



# designing computers and digital systems

using pdpl6 register transfer modules

c. gordon bell

John grason

allen newell

with the assistance of

Daniel Siewiorek and

Ross Scroggs

Carnegie-Mellon University

design examples

Edoardo Berera, Robert Chen, Paolo Coraluppi, Michael Knudsen,

Andre Orban and Robert Van Naarden

[previous](#) | [contents](#) | [next](#)



[previous](#) | [contents](#) | [next](#)

- stepping motor 339
- stop error 79
- STOP-bit 237
- straight-line 130, 177
- stuck-at faults 402
- stuck-at-one 407
- stuck-at-zero 407
- subminicomputer 293, 322
- subprocess 80, 162
- subroutine 58, 70, 87, 118, 131, 162, 177, 297, 329, (see also K(subroutine call))
- subsystem 19
- subtraction 170
- sum-of-integers 25, 30, 95, 98, 307, 321, 379
- SUT 414
- swap 282
- switch 17, 45, 68, 87, 240
- switching circuit 356
- switching circuit components 378
- switching circuit level 33, 352
- symbol 260
- synchronization 101, 102, 107, 108, 124, 127, 209, 217, 222, 225, 226, 235, 251, 253, 256, 258, 273, 365, 390
- synchronous logic 378
- synchronous transmission 230, 232, 237, 240
- synthesis 188, 191
- system 20
- system testing 396, 414
- system under test 414
- T (see transducer)
- T(analog-to-digital) 68, 240
- T(digital-to-analog) 68, 211, 240
- T(general purpose interface) 69, 323, 330, 333
- T(input interface) 69, 211
- T(lights and switches) 45, 68
- T(output interface) 69
- T(serial interface) 68, 234, 323, 330, 372
- T modules 38, 68
- table-lookup 101, 133, 135
- tallying 80
- termination 80, 83, 136, 151, 261
- ternary 14
- test patterns 402
- testability 417
- testing 79, 80, 86, 116, 160, 396
- time base 102, 201, 205, 389
- time-based 188
- time-shared 83, 204, 280
- timing 32, 108, 230
- tone synthesis 246
- tradeoff 140, 145, 154, 204, 348, 406, 417
- transducer 15, 17, 30, 38, 230, 251, 339
- transfer operations 41, 44
- transform 195, 197
- transmission speed 232
- transmit 68, 230, 240, 243
- triangle waveform generator 201
- TTL 355, 359, 362
- TTL signals 41
- Turing machine 260
- two's complement 13, 38, 42, 90, 163, 164, 296
- two-Bus system 127, 217, 219
- two-port memory 217
- unary operations 281
- unboxed explicit notation 48
- unboxed implicit notation 48
- underflow 98, 99, 175
- understandability 83
- universality 352
- unwinding 130, 151
- validation of syntax 396
- valve 221
- variable time jitter 194
- voltage level 40
- wait-until 102
- walking one 402, 410
- walking zero 402, 410
- waveform 201
- waveform analysis 209, 219
- waveform analyzer 188, 194, 201
- waveform generator 108, 188, 191, 198, 251
- weighted average 117

table-lookup 101, 133, 135  
tallying 80  
tape 260  
tape transport 260, 264  
technology 40  
telephone companies 230  
Teletrola 246  
Teletype 68, 234, 246, 251

weighted average 117  
wired OR 362, 374  
wiring 30, 66, 116, 359  
wiring validation 397  
word 13  
word size 13  
write 260, 285, 404  
8421 code 177

- pulse 190, 359
- punch 251, 259
- Push 280, 281
- Put 226, 273, 274
  
- queue 273
  
- race 361
- read 260, 285, 404
- real time 198, 246
- receive 230, 240, 243
- receiver 68
- recompute 156
- record 195, 209
- recursion 19, 99
- recursive subroutine 281
- register 2, 10, 39, 41, 66, 268, 295, 385
- register transfer components (see RT-level components)
- register transfer level (see RT-level)
- Register Transfer Modules 3, 28, 300, 352
- reliability 153, 154, 158
- replace 281
- replace-two 282
- representation 10, 15, 16, 90, 132, 156, 177, 265, 340, 349
- residues 133
- result 44
- return addresses 280
- return from subroutine 298
- right shift 44
- righthand side 34, 42
- Robert Chen algorithm 180
- rotate 90
- RSI 44
- RT-level 2, 33, 110, 141
- RT-level components 3, 5, 7, 10, 352
- RT-level design 378
- RTM Bus 30, 39, 41
- RTM control structure 378
- RTM system 28, 32
- RTM system diagram 48
- RTM's (see Register Transfer
  
- S (see switch)
- S(AND|OR|NAND|NOR) 66
- S54181\N74181 75
- sampling 191, 194, 199, 240, 259, 335
- sampling frequency 259
- sampling oscilloscope 197
- sampling switchboard 240
- sawtooth waveform 198
- scientific notation 175
- scratchpad 67, 75, (see also M(scratchpad;16 words))
- secondary memory 303
- sequential circuit design 378
- sequential circuits 3, 355
- Sequential Search Strategy 288
- serial to parallel 244
- shift 82, 90, 110, 183, 282, 298
- shift-register 385
- sign-magnitude 90
- signal 268
- signal types 39
- signed numbers 140
- significance 175
- simplex 21
- simulation 116, 260, 294
- sin(t) 201
- single Bus parallelism 372
- SINGLE STEP 46, 75, 364
- Skip 296, 297
- software 140, 145, 146, 152, 209
- source 42
- space-time trade-off 130
- special purpose digital systems 352
- specialization 152
- speed 6, 121, 124, 126, 127, 128, 129, 131, 133, 141, 145, 148, 156, 203, 349
- square wave 108
- stack 280
- stack operations 281
- stage 124, 136
- START 45, 47, 75
- START-bit 237
- start-stop 232, 243
- state 260
- state assignment 382, 385

RTM system diagram 70  
RTM's (see Register Transfer  
Modules)  
ruggedness 355  
Run 79  
Russian Peasant's Algorithm 136,  
144, 151, 158, 385

state 200  
state assignment 382, 385  
state diagram 379, 389  
state minimization 380  
state table 380  
state transition table 382  
statistical measures 195

- merge 58, 156, 369, (see also  
K(serial merge), K(parallel merge))
- microcode 145
- microcoded 297
- microinstruction 69
- microprogramming 69, 73, 141, 145,  
293, 323
- minicomputer 2, 140, 145, 293, 308,  
321, 342
- modem 230, 240
- modifiability 153, 154, 158
- modulation 188, 190
- module set 352
- module testing 414
- modules 3, 28, 32, 34
- monitor 219
- mounting panel 30
- move 260
- multiplexing 219
- multiplication 110, 136, 173, 321,  
348
- multiplier 110, 377, 385
- multiply step 146, 161, 321
- musical terminal 246
  
- n-ary operations 282
- NAND 17, 41
- negative logic 41, 359
- nested subroutine 307
- noise 231
- nondigital behavior 251
- NOR 17, 41
- normalization 86, 175
- NOT 17
- notation 41, 48
  
- objectives 23
- octave 249
- on-off keying 191
- one's complement 90
- open subroutine 87, 96, (see also  
macro)
- operate instruction 297, 319
- operation code 145, 296
- operation-complete 28, 42
- operations 12, 15
- operator 265
- optimization 6, 154
  
- OVERFLOW\OVF 39, 45, 46, 75, 82,  
166
  
- paper tape punch 251, 259
- parallel to serial 244
- parallelism 73, 101, 121, 124, 127,  
141, 148, 160, 204, 206, 217,  
222, 303, 335, 372
- parameters 18
- parity 80, 83
- parity counter 260
- Patil 4
- PDP-11 342
- PDP-16 3, 13, 28, 40
- PDP-16 handbook 28, 30
- PDP-16/M 74, 145, 152, 293, 322
- PDP-8 308
- PDP-8/RTM 293
- performance 9, 23, 24, 82, 120,  
121, 125, 126, 132, 148, 153, 206,  
219, 280, 321, 332, 348, 350,  
417
- physical technology 4
- pins 40
- pipeline 124, 160, 280
- PMS 16, 17, 28, 162
- PMS level 3
- polling 107, 204, 205, 206, 209,  
222, 226, 277
- pop 280, 281
- pop-two 282
- port 20
- positive logic 41, 359
- power 153
- POWER CLEAR 39, 46, 47, 75, 359
- precision 163
- primary memory 303, 342
- prime factors 133
- printed circuit boards 40
- priority 206
- processor 15, 16, 17, 335
- processor state 300, 303, 308, 342,  
350
- producer 222, 223
- program 73, 145, 323, 331
- program controlled access 332
- Program Counter 69, 70, 294, 295,  
308, 323, 328, 342

operator 265  
optimization 6, 154  
OR 17, 41, 44, 58  
output 69  
overflow 92, 98, 166, 171, 175,  
177, (see also OVERFLOW\OVF)  
overflow test 409

Program Counter 69, 70, 294, 295,  
308, 323, 328, 342  
programmable 349, 350  
programmable clock 201  
programmed 74  
programming level 3  
Ps 335

445

[previous](#) | [contents](#) | [next](#)

intra-DMgpa register transfer operations 372  
 ISP 27, 145, 293, 301, 308, 348, 389  
 iterations 136, 151  
  
 Jump 297  
  
 K (see control)  
 K(arbiter) 216, 217, 219, 222  
 K(branch 2-way) 42, 47, 365, 378, 398  
 K(branch 8-way) 58, 368  
 K(bus control and termination combined) 74, 75  
 K(bus sense and termination) (see Kbus)  
 K(clock) 58, 188, 210, 246  
 K(delay) 58, 188, 253  
 K(delay; integrated) 58  
 K(diverge) 58  
 K(evoke) 41, 356, 359, 361, 377, 378, 398  
 K(for-loop) 95, 107, 218  
 K(function encoder) 75  
 K(macro) 58, 212  
 K(manual evoke) 58, 68, 369, 372  
 K modules 28, 38, 39, 58, 69, 356  
 K(no-operation\nop) 58  
 K(parallel merge) 58, 66, 70, 107, 127, 156, 223, 369, 378  
 K(PCS) 69, 142, 145, 152, 322, 323, 332, 348, 385  
 K(Programmable Command Sequencer) (see K(PCS))  
 K(Programmed Control Sequencer) (see K(PCS))  
 K(serial merge) 58, 66, 70, 369, 378  
 K(subroutine call) 58, 369  
 K(wait until) 107  
 Karnaugh maps 382  
 Kb2 (see K(branch 2-way))  
 Kbus 39, 45  
 Kbus test 412  
 keywords 80  
 Ksub' 369, 377, 378

lights 68  
 linearly-ordered 286  
 link 15, 17, 21, 28  
 linked list 285, 286  
 logarithm 133  
 logic 158  
 logic conventions 359  
 logic level 3, 40  
 logic operations 42, 282, 326  
 logical design 6  
 logical shift 90  
 look-up 158  
 loop 58, 100, 103, 130, 151, 177  
 Low 41, 359  
 LSI 44, 361, 377  
  
 M (see memory)  
 M(array) 323, 330  
 M(array)test (see memory test)  
 M(array;1024 word) 67  
 M(array;read only; 1024 word) 67  
 M(byte register) 67, 273, 330  
 M(byte register) test 406  
 M(constants) 66, 273, 323, 330  
 M(content addressable) 273, 285  
 M(queue) 223, 226, 273  
 M(scratchpad; 16 words) 67, 273, 323, 330  
 M(stack) 273, 280  
 M(transfer register) 66, 86, 273, 313, 323, 329, 372, 417  
 M(transfer register) test 406  
 M modules 34, 66  
 MA (see memory address register)  
 macro 58, 87, 96, 117, 131, 162, 212  
 Macromodules 4, 5, 156  
 maintainability 154  
 maintenance 153  
 maintenance testing 412  
 mantissa 175  
 MANUAL 46  
 manual testing 412  
 mapping 67, 141, 406  
 MB (see memory buffer register)  
 memory 15, 17, 30, 34, 67, 75, 135

Ksub' 369, 377, 378

L (see link)

latch 365, 368

left shift 44

lefthand side 34, 42

level change 359

LIFO 273, 280

MB (see memory buffer register)  
memory 15, 17, 30, 34, 67, 75, 135,  
141, 142, 156, 273, 280, 295,  
303, 306, 312, 323, 329  
memory address register 67, 312,  
317  
memory buffer register 312, 318  
memory test 399

- dummy operation 368
- duplex 21
- duplicate 282
  
- EBCDIC code 13, 437
- edge triggered flip flops 379
- educational systems 352
- effective address 313, 317
- effective address calculation 313, 314, 317, 347
- encoding 10, 69, 75, 142, 188, 247
- EPUT meter 195, 201, 204, 339, 389
- error 199
- error signals 274
- ERTM (see Extended RTM)
- Ev-con 79
- evaluation 23, 148
- event 201, 209
- event counting 205, 339
- event-driven 211
- Events Per Unit Time (see EPUT meter)
- evoke 28, 41, 42, 69, 75
- evoke-operation 28, 39, 42
- Exclusive OR 44
- execution 294, 301, 303, 307, 314, 339
- exhaustive testing 399, 402
- exponent 175
- exponential function 201
- Extended K modules 66
- Extended RTM 66, 87, 95, 107, 222
  
- facility sharing 118, 129, 148
- fault diagnosis 404
- fetching 281, 294, 301, 314, 339
- Fibonacci sequence 98, 307, 321
- field 66
- FIFO 273
- flag 124, 223, 268, (see also DM(flag))
- flip flop 359
- floating point 175
- flow meter 339
- flowchart 25, 28, 112, 379, 389
- FM 191
- for loop 95
  
- full adder 382
- full duplex 231, 332
- functions 98, 109
- fundamental mode 359, 361
  
- general register 342
- generality 152, 154, 158
- generation 98
- Get 226, 273, 274
  
- half duplex 21, 231, 335
- Halt 297
- hardware 121, 126, 128, 129, 140, 146
- hardware technology 355
- hardwired 69, 73, 74, 141, 142, 145, 146, 152, 160
- harmony 251
- hash code 285, 289
- hazard 361
- hexadecimal 136, 177
- hierarchy 3, 14, 19
- High 41, 359
- histogram recorder 209
- human operator interface 266
  
- IF 87
- if-then-else 87
- increment 296
- indirect address 317
- information 1, 10, 14, 15, 156
- information processing 1
- initialization 209, 210
- input 69
- input-output 323, 330, 349
- instruction 69, 71, 141, 294, 299, 313
- instruction format 295, 303, 306
- instruction set 296, 303, 307, 326, 347
- instruction-execution 317
- instruction-interpretation 298, 313, (see also interpretation, interpreter)
- integer 10, 13, 42
- Integrated Circuit 40, 75, 352, 356, 377
- integrating delay 58, 102

FM 191

for loop 95

fraction 175

frequency 203, 247

frequency division 246

frequency limit 201

frequency modulation 191

frequency shift keying 191

integrated circuit 40, 75, 352, 356, 377

integrating delay 58, 102

interface 69, 205, 233, 251, 265,

268, 294, 332, 335

interpretation 141, 152, 303, 306

interpreter 73, 294, 299, 301, 314,  
347

interrupt 314

443

[previous](#) | [contents](#) | [next](#)

- complementation (negation) 44, 164, 298  
(see also two's complement)
- completeness 114
- components 2, 9, 15, 20
- computer 1, 16, 17, 74, 140, 145,  
209, 237, 293, 301, 350, 352,  
414
- computer-related systems 352
- concurrency 127, 148, 160
- conditional execution 87
- console state 303
- constants 161
- consumer 222, 223
- content addressable memory 285
- control 15, 17, 26, 28, 33, 38, 69,  
73, 124, 127, 130, 141, 145, 156,  
160, 219, 221, 323, 328, (see  
also controller)
- control flow links 39
- control modules 377
- control signal timing 356
- control signals 39
- control steps 355
- controller 265, 379, (see also  
control)
- conversion 177
- conveyor-bin 265
- cooling 153
- correction 178
- correctness 114
- cos(t) 201
- cost 4, 6, 23, 24, 80, 82, 120, 121,  
124, 126, 128, 131, 133, 135,  
141, 142, 145, 148, 152, 153,  
156, 219, 239, 280, 321, 332,  
348, 349, 350, 355, 417
- cost-performance tradeoff (see  
cost, performance, tradeoff)
- counter 390, 415
- counting 80, 102, 127, 138
- Crtm-1 145, 152, 161, 293, 294
- Crtm-2 350
- Crtm-2/2 350
  
- D (see data operation)
- D(clock and calendar) 197
- D(decoder) 66, 67, 71
- D(Exclusive-OR/Equivalence) 66
- data communications 230
- data flowgraph 379
- data operations 16, 17, 30, 34, 39,  
145, 162, 281, 303, 306
- data rate 258
- DATA READY (see DR)
- Data register 67
- data signals 39
- data transfer 34, 332
- Data transmission 332
- data type 12, 15, 38, 39, 303, 306
- data-memory 28, 33
- debugging 355, 396, 397
- decoder (see K(decoder))
- decoding 188, 294, 301, 313
- delay 18, 58, 102, 108, 237, (see  
also K(delay))
- delay-in-progress 58
- delete 285
- deque 284
- design 1, 5, 9, 80, 110, 153, 293,  
352
- design documentation 30
- design for testability 416
- desk calculator 349
- destination 42
- devices 3
- digital design 3
- digital interface 231
- digital systems 1, 10, 14
- digital technology 1, 18
- digital voltmeter 197
- digital-to-analog 68
- direct memory access 308, 332,  
335, 349
- display 209, 210, 213, 349
- distortion 232
- DM(arithmetic and registers) (see  
DMar)
- DM(flag) 47, 66, 273, 323, 329, 372
- DM(general purpose arithmetic unit)  
(see DMgpa)
- DM modules 34, 66
- DMar 74, 75, 350, 374, 417
- DMgpa 42, 75, 323, 362, 374, 417
- DMgpa test 408
- DO statement 95, 106
- documentation 116

D(decoder) 66, 67, 71  
D(Exclusive-OR|Equivalence) 66  
D(NOT) 66  
D modules 66  
DA 46, 48, 68, 356, 364, 372, 398  
DA-disable 66, 68, 372  
DATA ACCEPTED (see DA)

DO statement 95, 106  
documentation 116  
DONE 39, 42, 45, 47, 48, 58, 359,  
364, 369, 415  
DONE ENABLE 364  
DR 47, 68, 356, 362, 397  
DS error 79

**INDEX**

- acceptance testing 396, 398
- accumulator 295, 308, 342
- activate 28, 42
- activate-next 30, 42
- adder 385
- addition 15, 164
- address 145
- address calculation 303, 306
- address integer 13
- alarm 79
- algorithm 6, 28, 30, 82, 110, 127, 132, 141, 148, 151, 154, 161, 379
- amplitude modulation 191
- analog 219, 335
- analog-to-digital 68, 259
- analog-to-digital sampling unit 240
- analysis 188, 194
- AND 17, 41, 44, 58, 296
- applications 352
- arbiter 209, 222, (see also K(arbiter))
- arbitration 365, (see also K(arbiter))
- arithmetic 42, 90, 177, 282, 323, 326, 349
- arithmetic operations 42, 163
- arithmetic shift 90
- array 67, 121, 160
- array parallelism 122
- ASCII code 13, 246, 435
- assembler 396
- assembly 153
- assembly language 145
- asynchronous logic 378
- asynchronous timing 372
- asynchronous transmission 230, 232, 233
- attribute 285, 422
- AUTO 46
- Auto restart 79
- Auto run 79
- AUTO/MANUAL 75, 364
- automated tester 412
- average response 195
- backslash 28, 419
- binary coded decimal (see BCD)
- binary operations 282
- Binary Search Strategy 288
- binary-to-BCD 181, 321, 332
- bit 2, 10, 14, 16
- bit numbering 39
- bit position 39
- bit sequence 11, 14
- bit vector 13, 39, 42
- Boolean 13, 28, 38, 39, 41, 42, 58, 66, 68, 69, 73, 87, 365, 368, 385, 398
- Boolean vector (see bit vector)
- borrow 166, 170
- boxed notation 48
- branch 42, 58, 69, 209, (see also K(branch 2-way), K(branch 8-way))
- BSR 45, 46, 75
- Bus 30, 32, 45
- Bus control sequencing 361
- Bus driver 362
- Bus receiver 362
- Bus Sense Register (see BSR)
- Bus-exclusive operation 41, 372
- busy-waiting 211, 225
- byte 13
- CAM 285
- carrier wave 191
- carry 75, 166, 177
- Cgr 342
- character 13, 68
- character string 13
- character transmission 241
- checkers 11
- checking 160, 161
- circuit level 3
- circular shift 90
- Clark 4, 5, 156
- clear 273, 274, 285, 297
- clock 58, 102, 108, 188, 201, 233, 241, 243, 340, 389
- clock-calendar 340
- clocked sequential design 379, 389
- closed subroutine 87, (see also

backslash 28, 419  
BCD 13, 177, 268, 349, 350  
BCD-to-binary 181, 307, 321, 332  
behavior 5, 7, 9, 21, 24, 32, 389,  
390  
binary 10, 14, 340

clocked sequential design 379, 389  
closed subroutine 87, (see also  
subroutine)  
coating thickness monitor 219  
code (see encoding)  
combinational circuits 3, 58  
communications 230  
communications link 240

441

[previous](#) | [contents](#) | [next](#)

- Ellis, R.A. and M.A. Franklin, High level logic modules: A qualitative comparison, Proc. of IEEE Computer Conference, September, 1972.
- Hill, F.J. and G.R. Peterson, Introduction to Switching Theory and Logical Design. John Wiley, 1968.
- Holland, S.A. and C.J. Purcell, The CDC STAR-100: A large scale network oriented computer system, Proc. IEEE Computer Conference, pp. 55-56, September, 1971.
- Jurgen, R.K., Minicomputer applications in the seventies (Process Control Applications), IEEE Spectrum, Vol. 7, No. 8, p. 44, August, 1970.
- Knuth, D.E., The Art of Computer Programming, Vol. 1 Fundamental Algorithms, Vol. 2 Seminumerical Algorithms, Addison-Wesley, Reading, Mass., 1968.
- Kohavi, Z., Switching and Finite Automatic Theory, McGraw-Hill Book Co., New York, 1970.
- Martin, J.T., Telecommunications and the Computer, Prentice-Hall, Englewood Cliffs, N.J., 1969.
- Murphy, D.E., Introduction to Data Communication, Digital Equipment Corporation, 1968.
- Patil, S.S., Coordination of Asynchronization Events, Project MAC (MIT) Report No. TR72, May, 1970.
- Peatman, J.B., The Design of Digital Systems, McGraw-Hill Book Co., New York, 1972.
- Phister, M., Logical Design of Digital Computers, John Wiley, New York, 1958.
- Tesler, L, PUB - The Document Compiler, Stanford Artificial Intelligence Laboratory Operating Note 70, Stanford University, Stanford, Calif., to be published.
- Ware, W.H., Digital Computer Technology, 2 Vol., John Wiley, New York, 1963.

## BIBLIOGRAPHY

In addition to material cited in this book, this bibliography includes a few additional references that are pertinent to register transfer level design.

Bjorner, D., Flowchart Machines, BIT, 10, pp. 415-442, 1970.

Barnes, J.H., R.M. Brown, M. Kato, D.J. Kuck, D.L Slotnick, and R.A. Stokes, The ILLIAC IV computer, IEEE Transactions on Computers, Vol. C-17, No. 8, pp. 746- 757, August, 1968.

Bartee, T.C., I.L Lebow and IS. Reed, Theory and Design of Digital Systems, McGraw-Hill Book Co., New York, 1962.

Bell, C.G., J.L Eggert, J. Grason, and P. Williams, The Description and use of register transfer modules (RTM's), IEEE Transactions on Computers, Vol. C-21, No. 5, pp. 495-500, May, 1972.

Bell, C.G. and J.Grason, Register transfer modules (RTM) and their design, Computer Design, pp. 87-94, May, 1971.

Bell, C.G. and A. Newell, Computer Structures: Readings and Examples, McGraw-Hill Book Co., New York, 1971.

ACM, Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, Publication of the ACM, June, 1970.

Breuer, M., Design Automation of Digital Computers, Vol. I Hardware, Prentice Hall, Englewood Cliffs, N.J., 1972.

Chapin, N., Flowcharts, Auerbach Publishers, Philadelphia, 1971.

Chu, Y., Computer Organization and Microprogramming, Prentice Hall, Englewood Cliffs, N.J., 1972.

Chu, Y., Introduction to Computer Organization, Prentice Hall, Englewood Cliffs, N.J., 1971.

Clark, W.A., Macromodular computer systems, Proc. SJCC, pp. 335-336, 1967 (introduction to a set of 6 papers, pp. 337-401, in same proceedings).

DEC, PDP-16 Computer Designer's Handbook, Digital Equipment Corporation, Maynard, Mass., 1971.

Dennis, J.B. and S.S. Patil, Computation Structures, notes for Subject 6.232, Massachusetts Institute of Technology, Department of Electrical Engineering, 1970.

[previous](#) | [contents](#) | [next](#)

[previous](#) | [contents](#) | [next](#)EBCDIC CODE  
(Key)

NUL	Null	PRE	Prefix(or ESC, Escape)	SI	Shift In
PF	Punch Off	SM	Set Mode	SMM	Start of Manual Message
HT	Horizontal Tab	PN	Punch On	DLE	Data Link Escape
LC	Lower Case	RS	Reader Step	DC1	Device Control 1
DEL	Delete	UC	Upper Case	DC2	Device Control 2
RES	Restore	EOT	End of Transmission	DC3	Device Control 3
NL	New Line	SP	Space	DC4	Device Control 4
BS	Backspace	SOH	Start of Heading	NAK	Negative Acknowledge
IL	Idle	STX	Start of Text	SYN	Synchronous Idle
CC	Cursor Control	ETX	End of Text	CAN	Cancel
DS	Digit Select	ENQ	Enquire	EM	End of Medium
SOS	Start of Significance	ACK	Acknowledge	SUB	Substitute
FS	Field Separator	BEL	Bell	IGS	Information Group Separator
BYP	Bypass	VT	Vertical Tabulation	IRS	Information Record Separator
LF	Line Feed	FF	Form Feed	IUS	Information Unit Separator
EOB	End of Block(or ETB, End of Transmission Block)	SO	Shift Out	IFS	Information Field Separator

438

[previous](#) | [contents](#) | [next](#)

EBCDIC CODE

Bit Positions

0, 1	00				01				10				11			
	00	01	10	11	00	01	10	11	00	01	10	11	00	01	10	11
2, 3																
4, 5, 6, 7																
0000	NUL	DLE	DS		SP	&	-									0
0001	SOH	DC1	SOS				/		a	j			A	J		1
0010	STX	DC2	FS	SYN					b	k	s		B	K	S	2
0011	ETX	DC3							c	l	t		C	L	T	3
0100	PF	RES	BYP	PN					d	m	u		D	M	U	4
0101	HT	NL	LF	RS					e	n	v		E	N	V	5
0110	LC	BS	ETB	UC					f	o	w		F	O	W	6
0111	DEL	IL	ESC	EOT					g	p	x		G	P	X	7
1000		CAN							h	l	y		H	Q	Y	8
1001		EM							i	r	z		I	R	Z	9
1010	SMM	CC	SM		c	!	:									
1011	VT				.	\$	#									
1100	FF	IFS		DC4	<	*	%	@								
1101	CR	IGS	ENQ	NAK	(	)	-	'								
1110	SO	IRS	ACK		+	;	>	=								
1111	SI	IUS	BEL	SUB		~	?	"								

437

[previous](#) | [contents](#) | [next](#)USASCII DATA TRANSMISSION CODE

(Key)

436

NUL = All zeros	VT = Vertical tabulation	SYN = Synchronous/idle
SOH = Start of heading	FF = Form feed	ETB = End of transmitted block
STX = Start of text	CR = Carriage return	CAN = Cancel (error in data)
ETX = End of text	SO = Shift out	EM = End of medium
EOT = End of transmission	SI = Shift in	SUB = Start of special sequence
ENQ = Enquiry	DLE = Data link escape	ESC = Escape
ACK = Acknowledgement	DC 1 = Device control 1	FS = Information file separator
BEL = Bell or attention signal	DC 2 = Device control 2	GS = Information group separator
BS = Back space	DC 3 = Device control 3	RS = Information record separator
HT = Horizontal tubulation	DC 4 = Device control 4	US = Information unit separator
LF = Line feed	NAK = Negative acknowledgement	DEL = Delete

[previous](#) | [contents](#) | [next](#)

TABLE 5. ASCII AND EBCDIC CHARACTER SET ENCODINGS

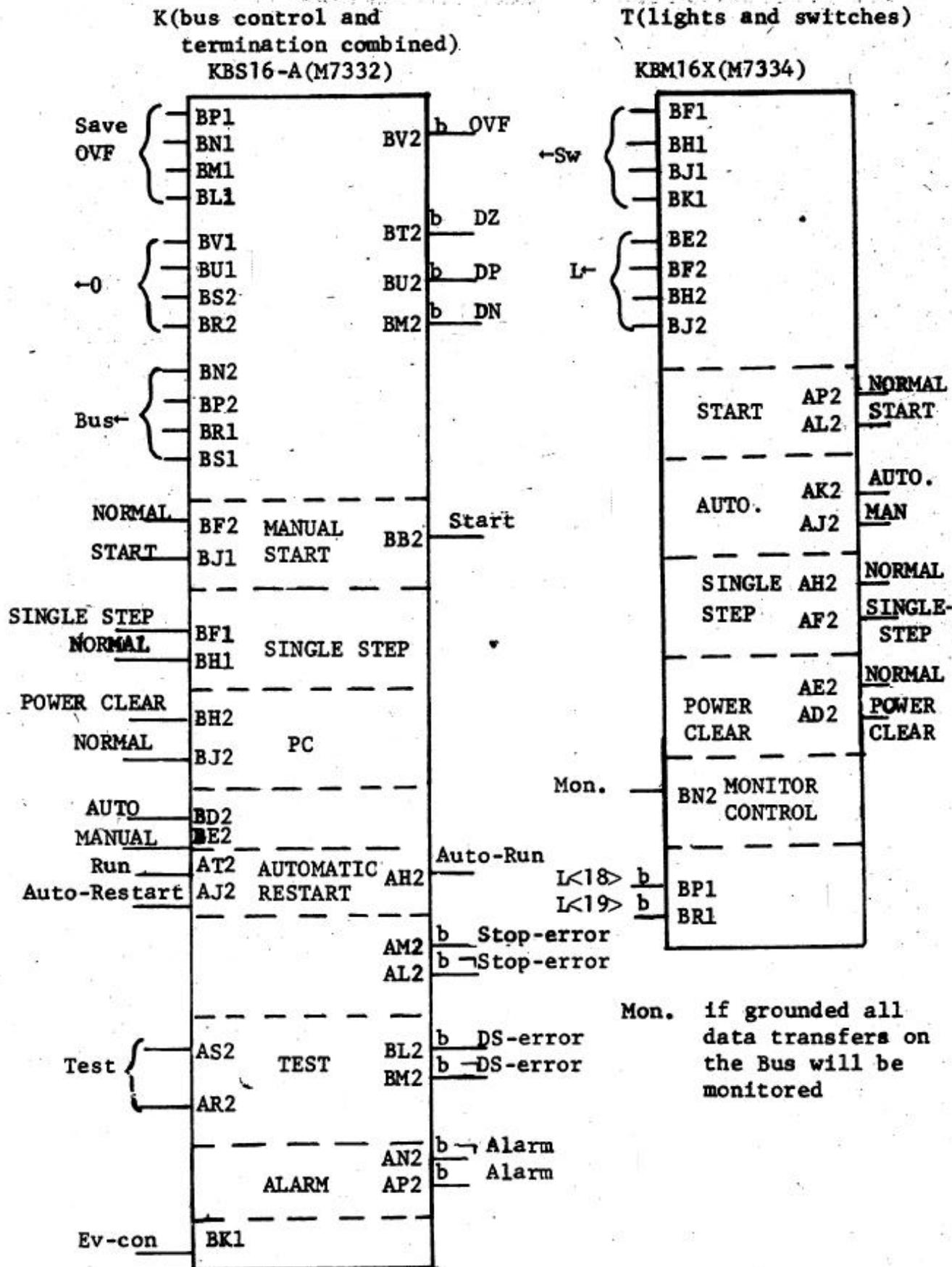
USASCII DATA TRANSMISSION CODE

Bit Positions				7	6	5	4	3	2	1
0	0	0	0	0	0	0	0	1	1	1
0	0	0	1	0	0	1	0	1	0	1
0	0	1	0	0	1	0	1	0	1	0
0	0	1	1	0	1	0	0	1	1	0
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	0	1	1	1	0	1	1
0	1	1	0	1	0	0	0	0	0	0
0	1	1	1	1	0	0	1	0	1	0
1	0	0	0	1	0	1	1	1	1	1
1	0	0	1	1	1	0	0	1	1	0
1	0	1	0	1	1	0	1	0	0	0
1	0	1	1	1	1	1	0	1	1	0
1	1	0	0	1	1	1	1	1	1	1
1	1	0	1							
1	1	1	0							
1	1	1	1							

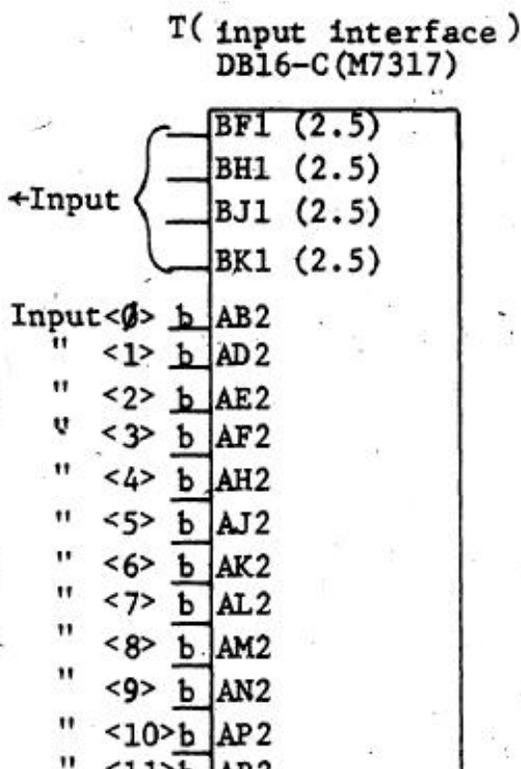
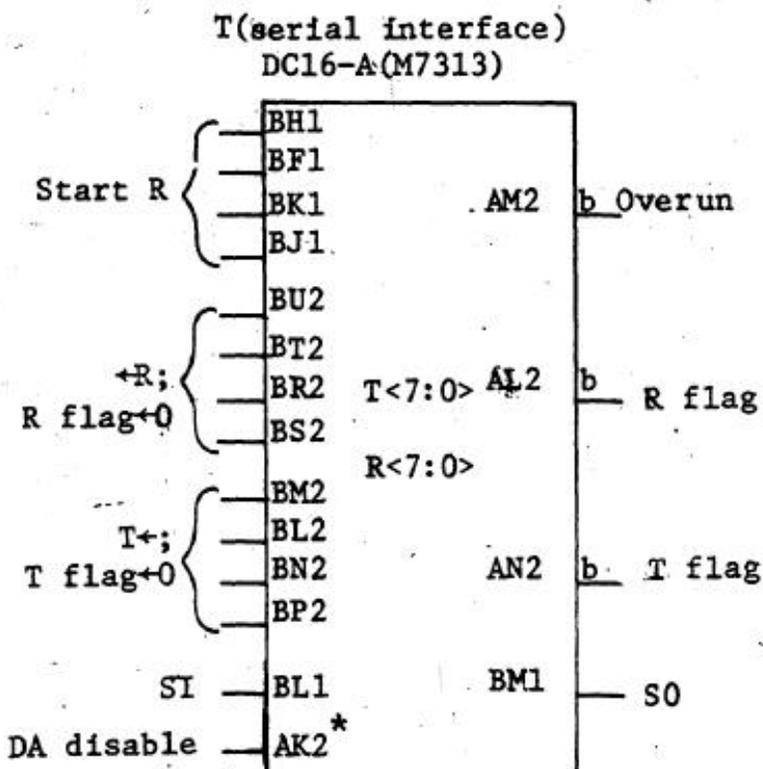
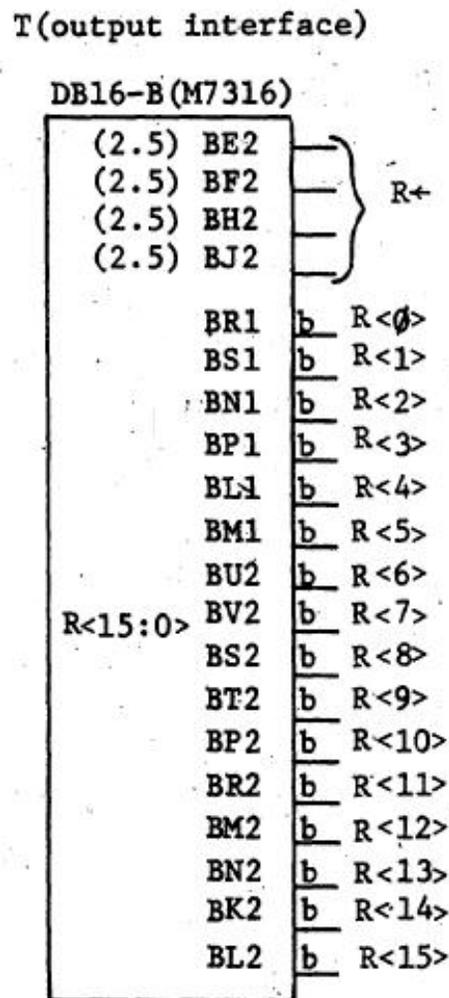
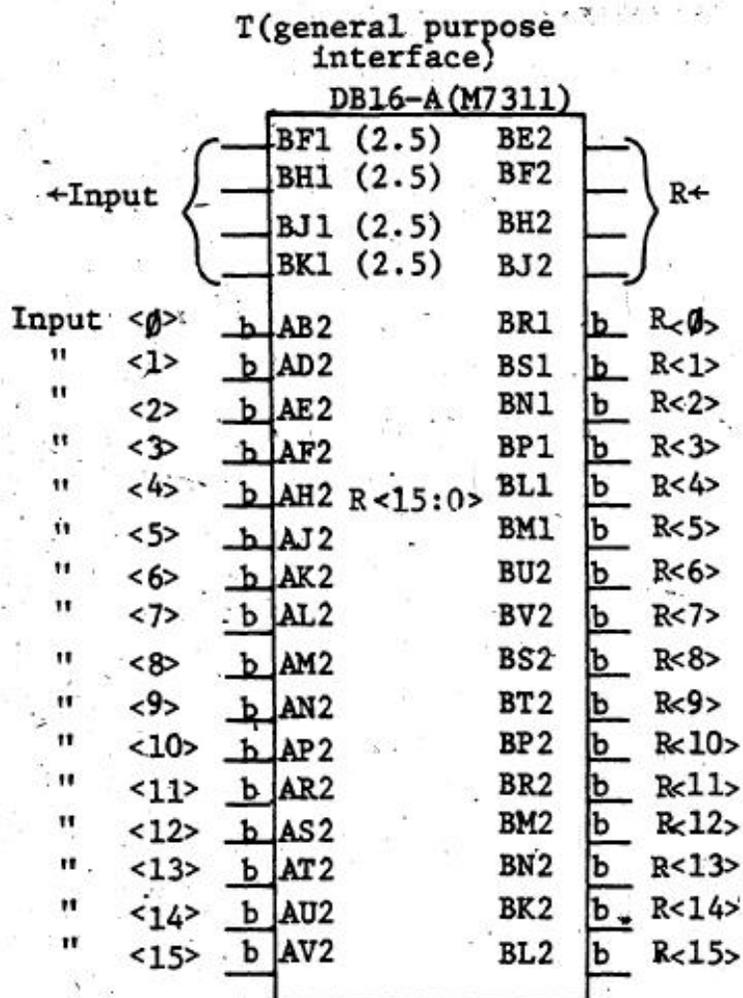
	0	1	2	3	4	5	6	7
NUL	DLE	SP	0	@	P	'	p	
SOH	DCL	!	1	A	Q	a	q	
STX	DC1	"	2	B	R	b	r	
ETX	DC3	#	3	C	S	c	s	
EOT	DC4	\$	4	D	T	d	t	
ENQ	NAK	%	5	E	U	e	u	
ACK	SYN	&	6	F	V	f	v	
BEL	ETB	'	7	G	W	g	w	
BS	CAN	(	8	H	X	h	x	
HT	EM	)	9	I	Y	i	y	
LF	SUB	*	:	J	Z	j	z	
VT	ESC	+	;	K	[	k	{	
FF	FS	,	<	L	\	l		
CR	GS	-	+	M	]	m	}	
SO	RS	.	>	N	^	n	~	
SI	US	/	?	O	-	o	DEL	

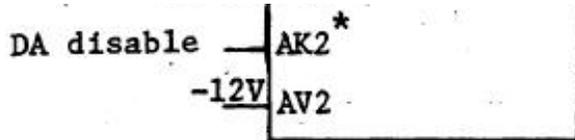
435



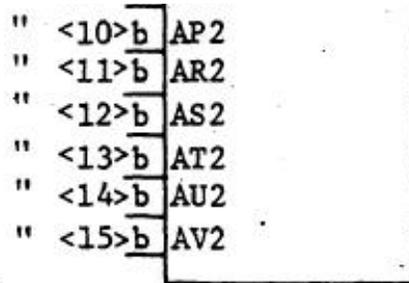


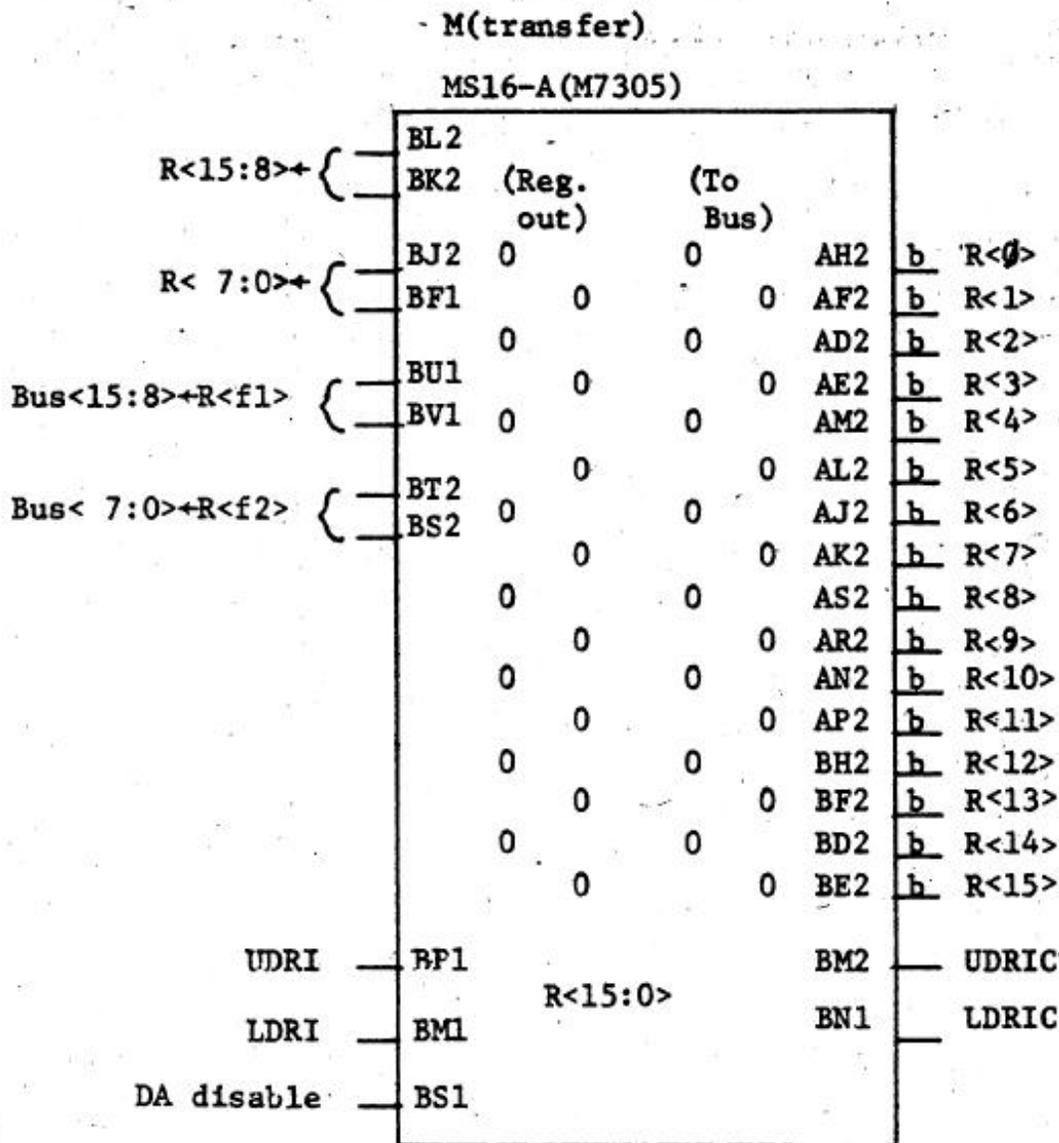
[previous](#) | [contents](#) | [next](#)





\* must be grounded to disable the paper tape reader timing circuit.  
 SI TTL compatible serial input.  
 SO TTL compatible serial output.





UDRI	upper 8 bits direct read in	LDR1	lower 8 bits direct read in
UDRIC	upper 8 bits direct read in control	LDRIC	lower 8 bits direct read in control

DA disable. When this is grounded, the timing circuit for R<15:8>+ and R<7:0>+ of the transfer register will be disabled.

In normal use UDRI and LDR1 should be grounded. If the timing circuit is disabled UDRI and LDR1 must be connected to the UDRIC and LDRIC outputs of another transfer register, which is receiving the same word and does not have its timing circuit disabled.

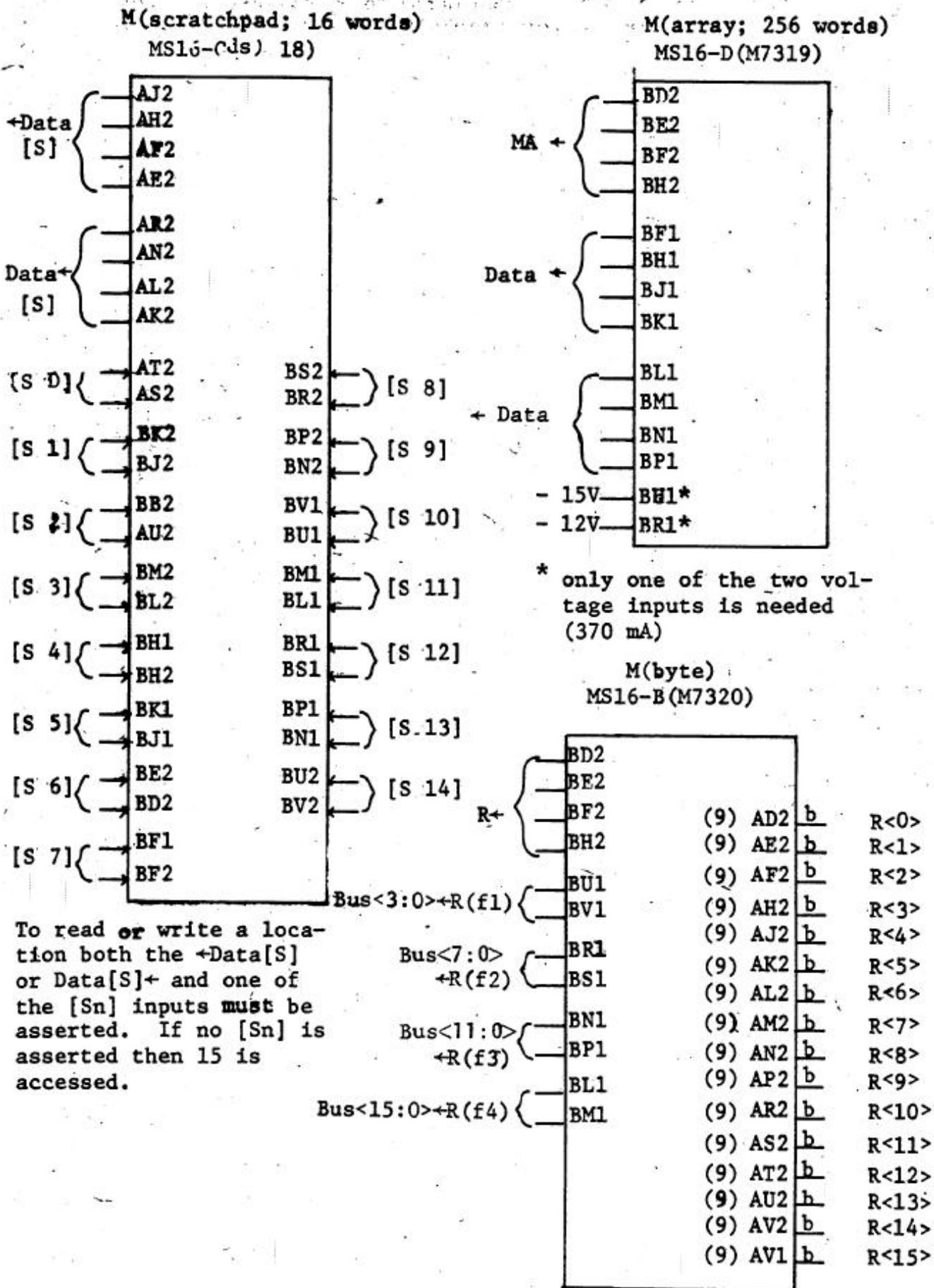
All outputs have 9 TTL unit loads each.

**All outputs have 9 TTL unit loads each.**

**Circuit board solder terminals permit bit interchanging during read operations only.**

432

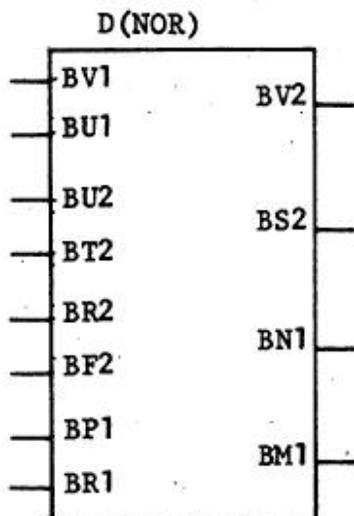
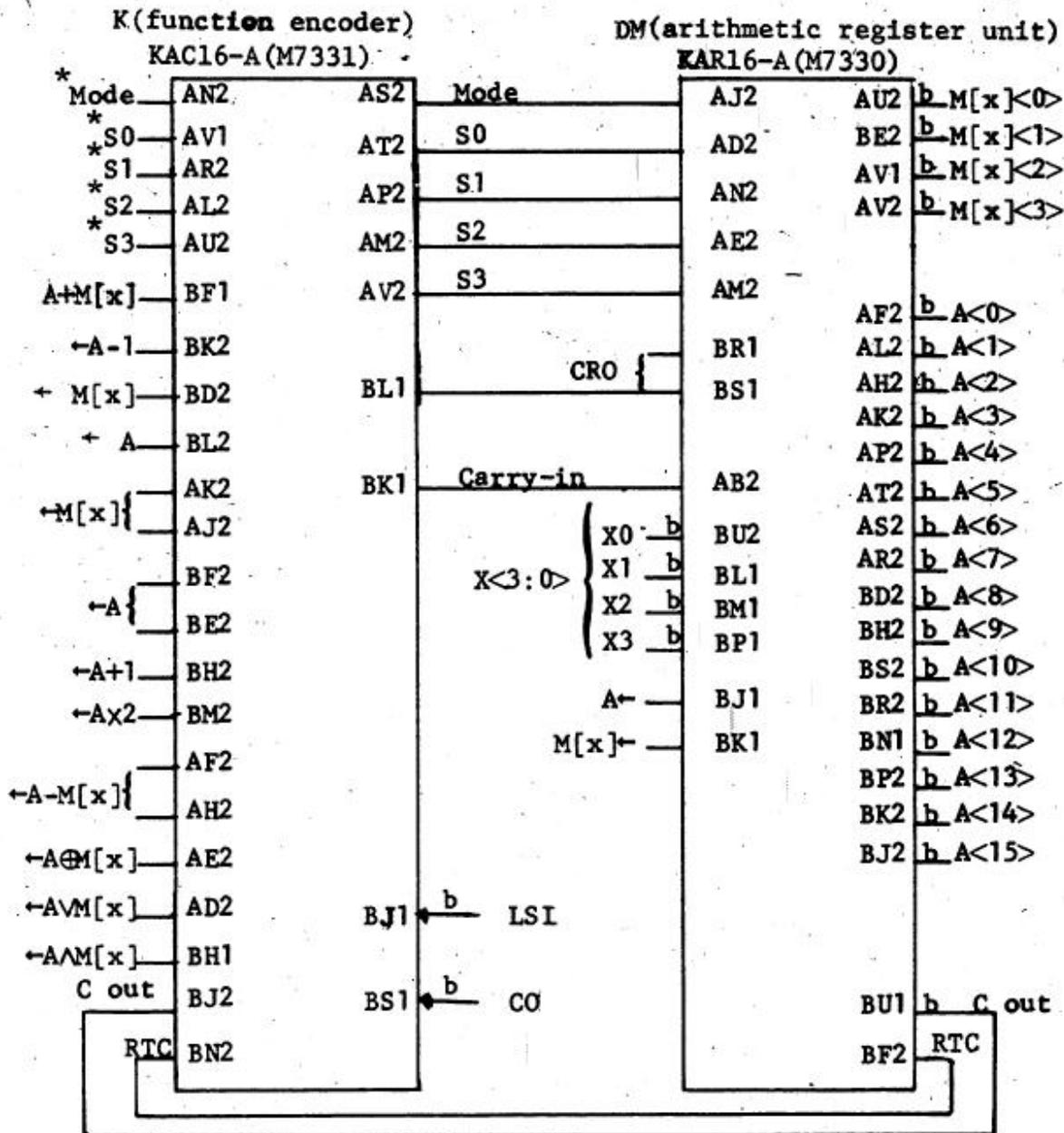
[previous](#) | [contents](#) | [next](#)



Circuit board solder terminals permit bit interchanging during read operations only.

431

[previous](#) | [contents](#) | [next](#)



\*S0 - \*S3, \*Mode. These inputs are used when evoking encoded functions (see Fig. 25 of Chap. 2)

.CRO. Common readout. If asserted it causes the result of the function to be put on the Bus.

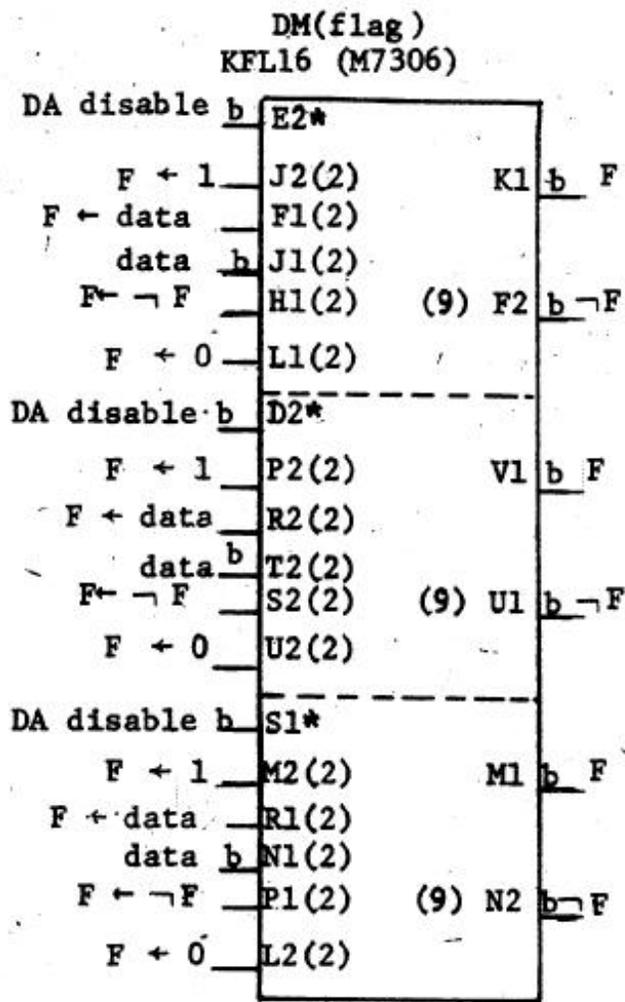
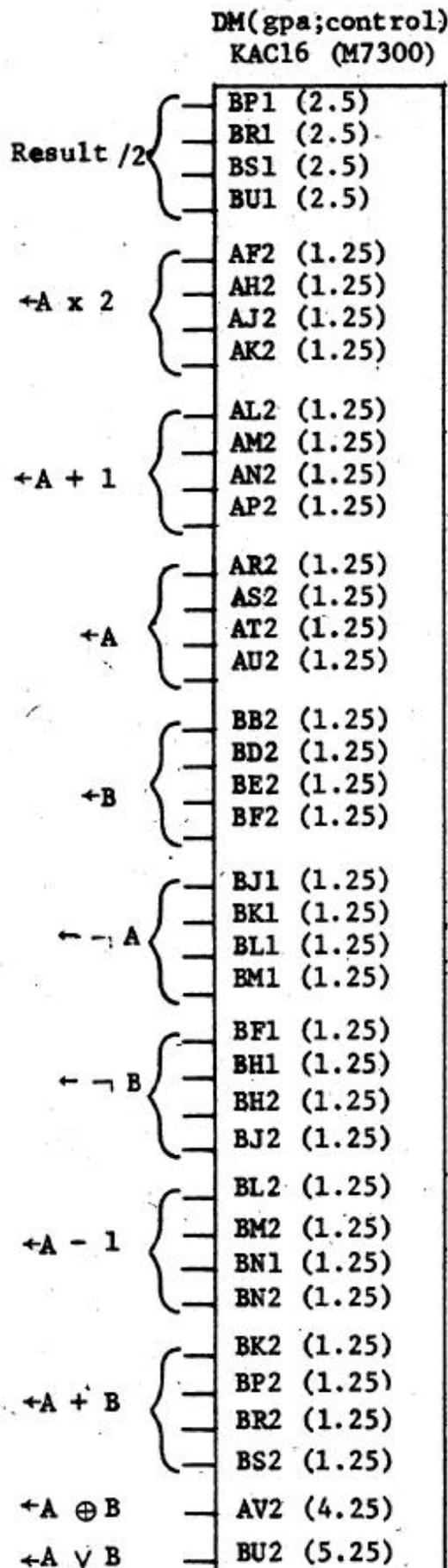
LSI. When doing a A x 2 it specifies what is shifted into Bit 0.

CO. When evoking a special function (\*S0 - \*S3), CO will be transferred without change to the Carry in on the DMaru.

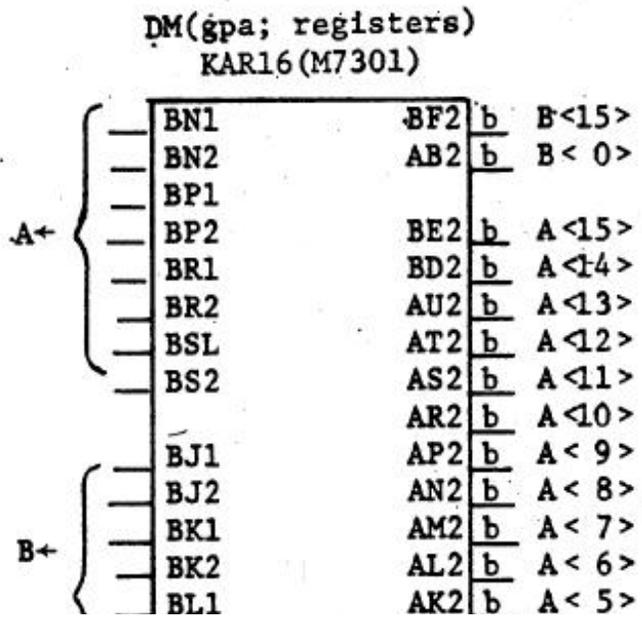
Carry in on the DMaru.

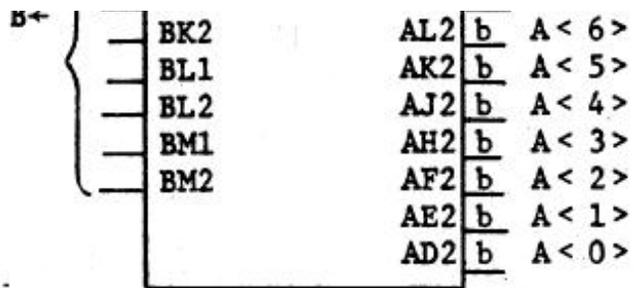
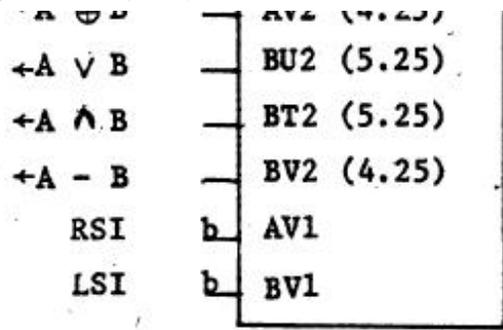
430

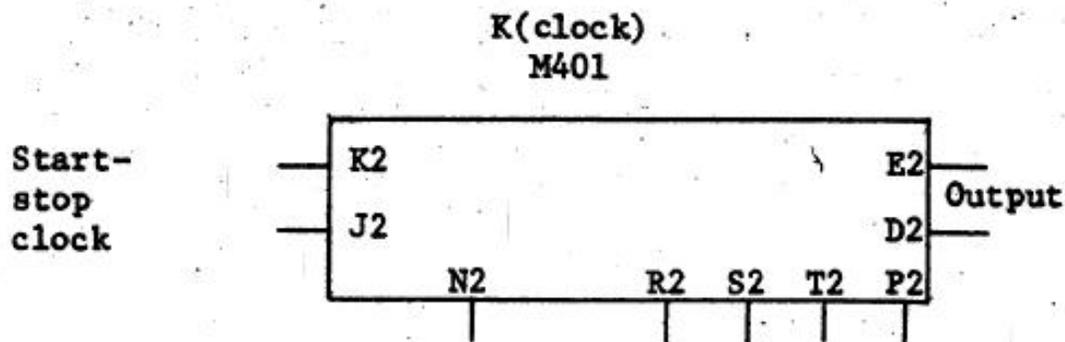
[previous](#) | [contents](#) | [next](#)



\* Must be grounded to disable the flag timing circuit.







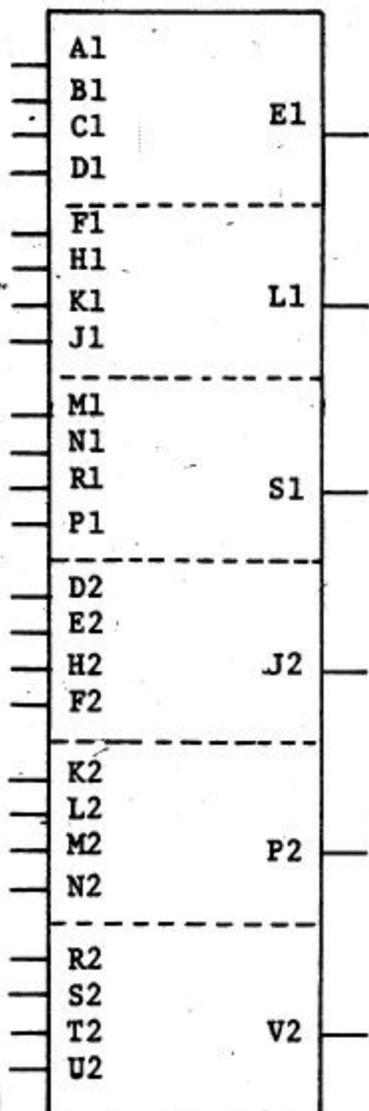
Frequency Range		Interconnections
1.5 MHz to 10 MHz	(100 pf)	NONE
175 KHz to 1.75 MHz	(1000 pf)	N2 - R2
17.5 KHz to 175 KHz	(.01 $\mu$ f)	N2 - S2
1.75 KHz to 17.5 KHz	(.1 $\mu$ f)	N2 - T2
175 Hz to 1.75 KHz	(1 $\mu$ f)	N2 - P2

Fine frequency adjustment is controlled by an internal potentiometer. For other ranges an external capacitor may be connected between N2 and ground.

A level change from High to Low with a fall time of less than 400 nsec. is required to enable the clock.

[previous](#) | [contents](#) | [next](#)

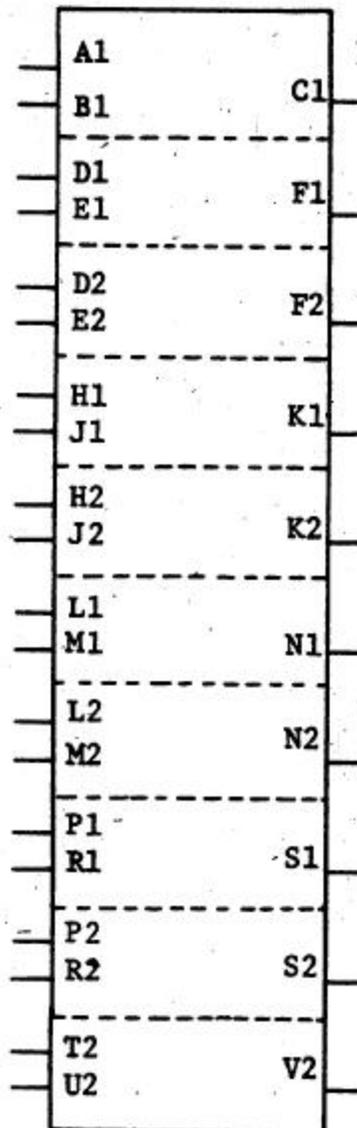
**K(serial merge; 4 input)  
KOR16-B (M1307)**



Each input represents 1.25 TTL unit load.

U1 and V1 are sources of logical "1" (i.e. +3 volts) Each can drive 20 TTL unit loads.

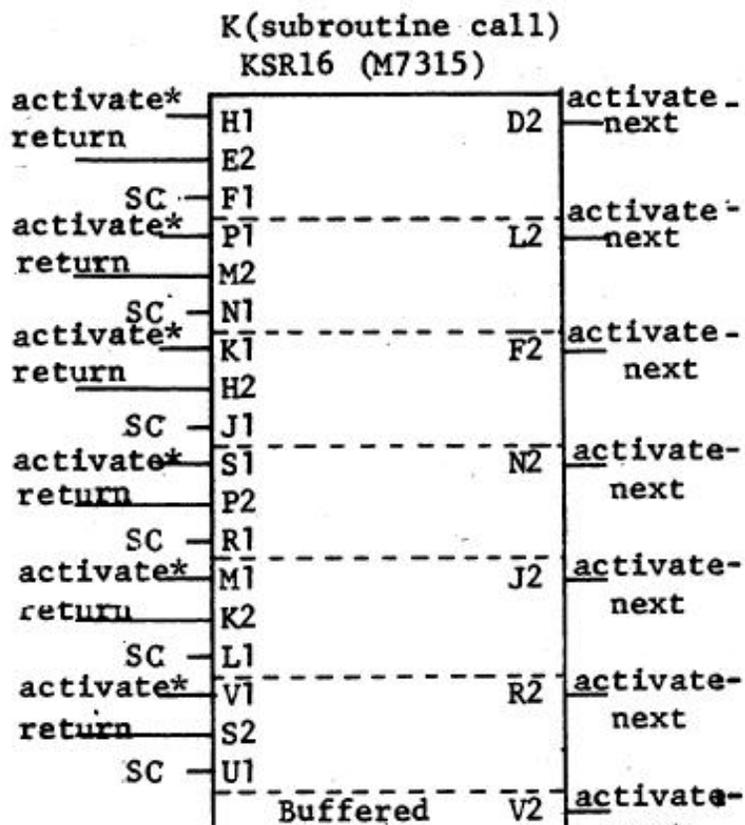
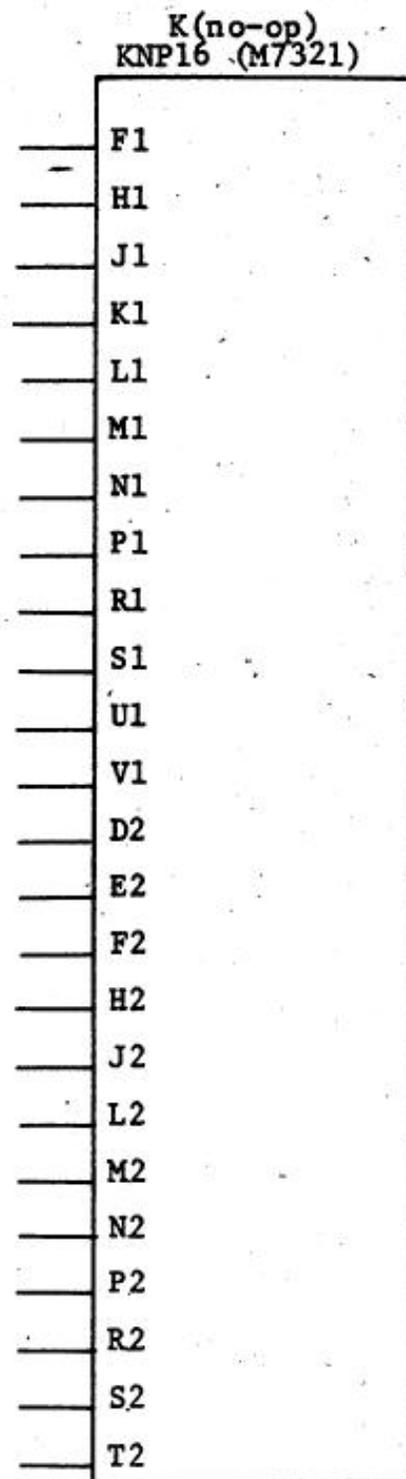
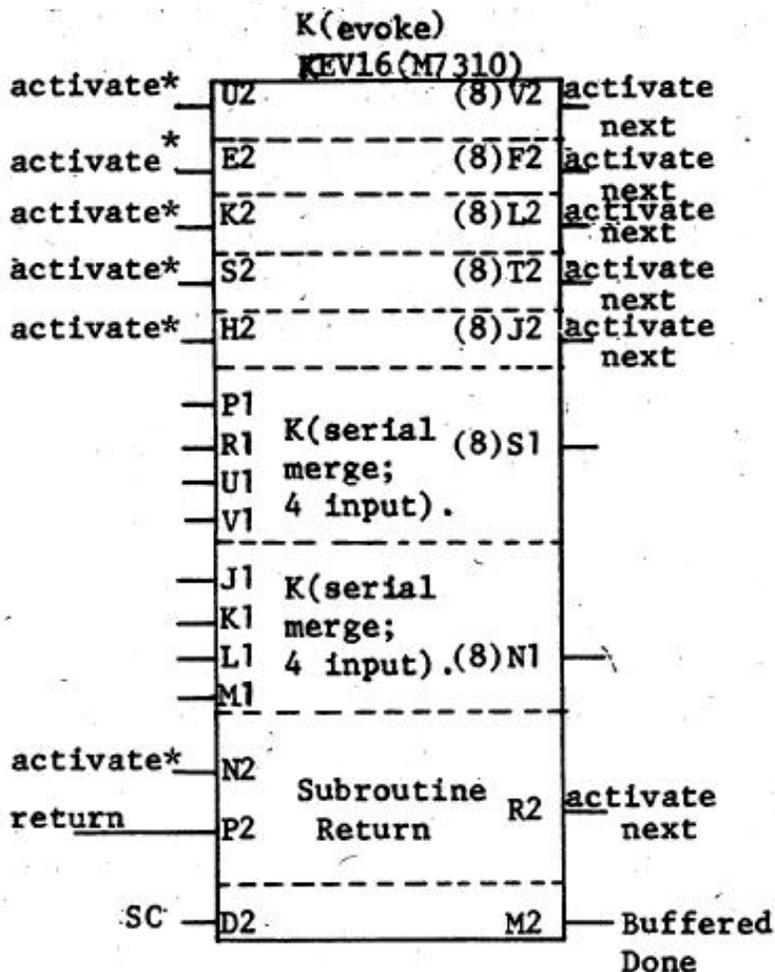
**K(serial merge; 2 input)  
KOR16-A (M1103)**



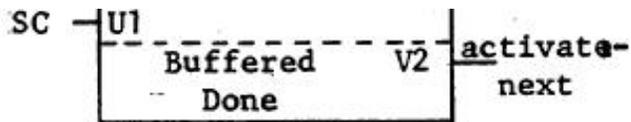
Each input represents 1.25 TTL unit load.

U1 and V1 are sources of logical "1" (i.e. +3 volts) Each can drive 20 TTL unit loads.

[previous](#) | [contents](#) | [next](#)



A delay of 100-200 ns occurs before the next evoke step is enabled.



SC . Start control input. This input must be connected to the Buffered done output if the subroutine return is to be used as a start circuit.

\*activate  $\equiv$  evoke operation for K(evoke)'s and activate  $\equiv$  call for K(sub)'s.



VZ U

Throughout this appendix, each input presents 1 TTL unit load and each output can drive 10 TTL unit loads except if otherwise specified in parentheses.

AL2	BBSR<2>
AJ2	BBSR<1>
AH2	BBSR<0>

[previous](#) | [contents](#) | [next](#)

[previous](#) | [contents](#) | [next](#)

<single-bit Boolean expression> <Boolean register>|  
 <-Boolean register>|  
 <register specifier>  
 <relational operator>  
 <register specifier>

A single-bit Boolean expression involves either a Boolean register or a pair of registers and a relational operator. The value of a Boolean expression is always either 0 or 1. Boolean expressions are usually used to condition two-way branches.

<triple-bit Boolean expression> <single-bit Boolean expression>□  
 <single-bit Boolean expression>□  
 <single-bit Boolean expression>

A triple-bit Boolean expression is composed of three single bit Boolean expressions and is used to condition eight-way branches.

[previous](#) | [contents](#) | [next](#)

**<PMS type>**

C\Computer  
 D\Data operation  
 DM\Data and Memory  
 K\Control  
 L\Link  
 M\Memory  
 S\Switch  
 T\Transducer

**<PMS component specifier>**    **<PMS type><attribute>**

See Chapter 2, Figures 16 and 17, for examples of the forms used to specify PMS components.

**<comment>**

A comment is given as clarification and explanation in RTM system flowcharts. Where needed, single curly brackets are used to separate comments from the flowcharts:

**<register expression>**

A register expression appears as the right hand side (source) in a register transfer statement. See the description of the modules in Chapter 2 for examples of the operations available with RTM's.

**<simple register transfer statement>**    **<register specifier>←  
<register expression>**

Register transfer statements indicate the transfer of data from a register expression (source) to a register specifier (destination). The register expression is evaluated with the current values of the registers in the expression, then the transfer is executed. The same register may appear on both sides of the statement and source and destination registers must agree as to data type.  
 e.g.  $A \leftarrow A + 1$   
 $B \leftarrow (A + B) / 2$

**<multiple register transfer statement>**    **<register specifier>←  
<register specifier>←  
<register expression>|  
<register specifier>←  
<multiple register transfer statement>**

Multiple registers may be specified as the destination of the value of a register expression.  
 e.g.  $T \leftarrow X \leftarrow A + B$

<multiple register  
transfer statement>

<Boolean register>

A Boolean register is a single bit register which stores only the values 0 and 1. There may be Boolean registers of the DMflag or single bits of other registers. e.g. Overflow-flag  
BSR<15>

423

[previous](#) | [contents](#) | [next](#)

<b>&lt;subscript range&gt;</b> [ <b>&lt;range&gt;</b> ]	<p>A subscript range represents a set of integer values over which an index can range. The values are in decimal unless explicitly shown otherwise.</p> <p>e.g. [1:1023] [J1:J2] [0:n-1]</p>
<b>&lt;subscript range list&gt;</b> <b>&lt;subscript range&gt;</b>   <b>&lt;subscript range list&gt;</b> - <b>&lt;subscript range&gt;</b>	<p>Multi-dimensional arrays, are specified with a subscript range list.</p> <p>e.g. [0:1023] [0:7][0:15][0:3]</p>
<b>&lt;list&gt;</b> <b>&lt;range&gt;</b>   <b>&lt;element&gt;</b>   <b>&lt;list&gt;</b> , <b>&lt;range&gt;</b>   <b>&lt;list&gt;</b> , <b>&lt;element&gt;</b>	<p>A list is a set of numbers and/or ranges and/or identifiers.</p> <p>e.g. S,T,R,3,2,1 Sign,14:1,Lsb</p>
<b>&lt;word specifier&gt;</b> <b>&lt;list&gt;</b>   <b>&lt;list&gt;</b> ↓ <b>&lt;base&gt;</b>	<p>A word specifier, using a list, provides "names" for the bits of a word. The base of the value stored of the value stored in the word may be indicated; if the base is omitted, binary is assumed.</p> <p>e.g. &lt;15:0&gt; &lt;3,2,1,0&gt;↓16 &lt;Sign,14:1&gt;</p>
<b>&lt;register specifier&gt;</b> <b>&lt;identifier&gt;</b>   <b>&lt;identifier&gt;</b> <b>&lt;word specifier&gt;</b>   <b>&lt;identifier&gt;</b> <b>&lt;subscript list&gt;</b>   <b>&lt;identifier&gt;</b> <b>&lt;subscript list&gt;</b>   <b>&lt;word specifier&gt;</b>	<p>A register specifier defines a register; these may be registers as in a DMgpa or an M(transfer) or a single word of memory.</p> <p>e.g. BSR S&lt;15:0&gt; Mem1[63] Table[47]&lt;7:0&gt;↓8</p>
<b>&lt;attribute value&gt;</b> <b>&lt;attribute&gt;</b>   <b>&lt;attribute name&gt;</b> : <b>&lt;attribute value&gt;</b>	<p>Attributes are descriptive information given about a PMS component. An attribute name serves to label an attribute value and is separated from the information by a colon.</p> <p>e.g. clock period: 35μseconds logic type: TTL</p>
<b>&lt;attribute list&gt;</b> <b>&lt;attribute&gt;</b>   <b>&lt;attribute list&gt;</b> ; <b>&lt;attribute&gt;</b>	<p>Several attributes may be grouped together in a list to provide information.</p>

<attribute list> <attribute>|  
<attribute list>;<attribute>

Several attributes may be grouped together in a list to provide information about a PMS component. Attributes are separated by semicolons in the list.

e.g. technology:MOS; cycle time:750nsec;  
sizes:1024/2048/4096 words

422

[previous](#) | [contents](#) | [next](#)

<b>&lt;multiword identifier&gt;</b>	<b>&lt;simple identifier&gt;-&lt;simple identifier&gt;  &lt;multiword identifier&gt;-&lt;simple identifier&gt;</b>	Multiword identifiers are formed using the hyphen to connect simple identifiers. There are no blanks between the simple identifiers and the hyphens.
<b>&lt;identifier&gt;</b>	<b>&lt;simple identifier&gt;  &lt;multiword identifier&gt;</b>	
<b>&lt;base&gt;</b>	<b>&lt;number&gt;</b>	A base (radix) is always expressed in decimal and is used in conjunction with the ↓.
<b>&lt;constant&gt;</b>	<b>&lt;number&gt;  &lt;number&gt;↓&lt;base&gt;</b>	If the base (radix) is not clear from context, it will be given with a base.
<b>&lt;simple integer expression&gt;</b>		A simple integer expression is an arithmetic integer that evaluates to have an integer value. e.g. $N-1$ $2*j$
<b>&lt;element&gt;</b>	<b>&lt;identifier&gt;  &lt;constant&gt;  &lt;simple integer expression&gt;</b>	
<b>&lt;range&gt;</b>	<b>&lt;element&gt;:&lt;element&gt;</b>	A range specifies the bounds of a set of values. The elements represent integers and the range represents all integers between the bounds inclusive.
<b>&lt;subscript&gt;</b>	<b>[&lt;element&gt;]</b>	A subscript represents a single integer value and is expressed in decimal unless explicitly shown otherwise. e.g. [3] [J] [n-1]
<b>&lt;subscript list&gt;</b>	<b>&lt;subscript&gt;  &lt;subscript list&gt;&lt;subscript&gt;</b>	A single element in a multi-dimensional array is specified by a subscript list. e.g. [3][2][5] [J][K]

e.g. [3][2][5]  
[J][K]

421

[previous](#) | [contents](#) | [next](#)

- :=** definition. The colon-equal is used in the form  $A := B$  which indicates that the value of  $A$  is defined by the expression  $B$ . The forms of  $A$  and  $B$  are not fixed but the form is used most often in defining registers, memories, processes and operations in terms of existing entities of these types. For example:
- ```
MWORD[0]<15:0>:= MBYTE[0]<7:0>□MBYTE[1]<7:0>
SP<15:0>\Stack-pointer-register:= R[6]
Increment-operation := (MA←Z; next A←MB; next MB←A+1)
```
- conditional-evoke. The right double-arrow is used in the form  $A \rightarrow B$  where  $A$  is a Boolean expression and  $B$  is usually a register-transfer-statement or a sequence of such statements. When  $A$  has the Boolean value 1 (true), the operation(s) indicated by the expression  $B$  is (are) executed. A form employing both the  $:=$  and  $\rightarrow$  is  $A := B \rightarrow C$ .  $A$  is a single-bit register that is assigned the value of the Boolean expression  $B$ . If  $B$  has the Boolean value 1, the expression  $C$  is executed.
- ← or →** transfer. The left or right arrow is used to indicate a data transfer. The forms  $A \leftarrow B$  and  $B \rightarrow A$  are equivalent. See Table 3, register transfer statement.
- { }** operation extension. Curly brackets are used to modify or describe an operation. For example:
- ```
A ← A/2 {arithmetic}
S ← R + T {double precision}
A ← A + B {mod 64}
```

TABLE 3. COMMON EXPRESSION FORMS USED IN RTM SYSTEMS

This table is intended to be a reference for various terms and expression forms used throughout the book. It is not intended to be a guide for writing ISP and PMS expressions; such information may be found in Bell and Newell (1971). In the following descriptions the vertical bar, |, is used to signify alternative forms. Angle brackets,  $\langle \rangle$ , are used to enclose the names of the forms that are being described, since some of them are more than one word long. This special use of the angle brackets is limited to the Expression and Definition columns of this table.

Expression	Definition	Explanation
$\langle$ simple identifier $\rangle$		A simple identifier is a string of letters and numbers from the character set. There is no restriction on the length of a simple identifier but

no restriction on the length of a simple identifier, but the first character of a simple identifier will often be capitalized.

420

[previous](#) | [contents](#) | [next](#)

- ;** general separator. The semi-colon is used to separate:

  - a. register-transfer-statements. Statements separated by a semi-colon are executed in parallel. For example, "A  $\leftarrow$  B ; B  $\leftarrow$  A" causes an interchange of the values of A and B.
  - b. attributes in a PMS component attribute list, e.g. see Table 3, attribute list.
  
- :** general separator. The colon is used to separate:

  - a. bounds of a range of values, e.g. see Table 3, range.
  - b. an attribute name and the associated attribute value, e.g. see Table 3, attribute.
  
- ,** list item separator. The comma is used to separate items in any type of list.
  
- next** sequential statement delimiter. The word "next" is used in conjunction with a semi-colon to indicate a sequential ordering of register-transfer-statements. For example, "A  $\leftarrow$  A + 1 ; next B  $\leftarrow$  A" indicates that the register A is incremented by one, then the new value is assigned to the B register.
  
- ?** unknown value. A question mark is used to indicate an unknown value in an equation.
  
- $\Phi$**  set value. Used in conjunction with  $\downarrow$  or base, the capital phi represents the set of all values that a digit with the given base can assume, e.g.,  $\Phi\downarrow 4$  represents the set of values: 0, 1, 2, 3. Most commonly used to denote a Boolean "don't care", i.e. a Boolean bit that can either be a 1 or a 0.
  
- word connector. The hyphen is used to establish multiword identifiers, e.g. see Table 3, multiword identifiers. When the hyphen is used to indicate subtraction, blanks will separate the operator and the operands, and parentheses will be used to eliminate possible ambiguities with multiword identifiers.
  
- $\square$**  register concatenator. The square box is used to define large registers in terms of smaller component registers. The registers defined, or those in the definition do not necessarily correspond to separate physical registers. This form is most frequently used to provide mnemonic names for subparts of larger registers. For example, Instruction-word<15:0> could be defined as Op-code<3:0> $\square$ Op-address<11:0>.
  
- ~** approximate range. The tilde is used to indicate an approximate range of values, e.g. 4 ~ 6 bits, 10 ~ 25 volts, ~ 1 $\mu$ sec.

of values, e.g. 4 ~ 6 bits, 10 ~ 25 volts, ~ 1 $\mu$ sec.

\ synonym separator. The back slash is used in an expression of the form A\B which indicates that B is to be considered as a synonym for A. The form of A and B is not fixed, but usually B represents a simpler form, an abbreviation, or mnemonic for A. For example, Bus-sense-register\BSR, DM(general-purpose-arithmetic)\DMgpa, Instruction-word<15:12>\Op-code.

**TABLE 1. COMMON ARITHMETIC, LOGICAL, AND RELATIONAL OPERATORS****Arithmetic**

+	addition
-	subtraction, negation
*	multiplication
/	division
↑	exponentiation
↓	base
mod	modulus
sqrt	square root
abs	absolute value

**Logical**

¬	NOT, complement
∧	AND
∨	OR, inclusive
⊕	XOR, exclusive OR
≡	equivalence

**Relational**

=	equal
≠	not equal
>	greater
≥	greater or equal
<	less
≤	less or equal

[previous](#) | [contents](#) | [next](#)

branch Boolean). A compromise for test set generation would be to analyze branch conditions for test data while using a random number generator for non- looping test data.

## CONCLUSION

Various aspects of the testing problem and the time-hardware tradeoff required to test have been explored. Testing -- both in actual check-out and debugging, and in constructing testing systems -- is a large part of the total design process. It does not make itself evident in the written word, but does as soon as physical systems are constructed. By devoting an entire chapter to the problem we have hoped to indicate its importance. We even suggest that testing is important enough to be included in the original design criteria, hence to effect the structure of the physical system. That is, there exists a testability-performance-cost tradeoff.

## PROBLEMS

1. Design several tests for the M(transfer), which are analogous-to those of the M(array).
2. Modify the M(transfer) test for testing S<-R<-C.
3. Modify the M(transfer) tests to also test M(byte).
4. Carry out the designs for automatic and comparison tests for the DMgpa.
5. Design a procedure that allows a person to generate tests for a new module. Try it for DMar.
6. Note that in the memory tests checking always follows writing, and that the 2 pairs of subroutines write pattern-write walking and check pattern-check walking are nearly identical. Carry out a design which makes them the same and selectable by a DMflag.

that the SUT is altered as little as possible. Altering the SUT has four distinct disadvantages:

1. It changes the system from what it was when the fault occurred, perhaps leading to an incorrect diagnosis.
2. The alterations themselves could be sources of error.
3. The alterations are time consuming.
4. Removal of the alterations from the SUT could itself inflict faults such as the accidental removal of a control wire.

There are several parameters of a SUT which determine testing ease. If these constructs exist, then the SUT need be altered very little. Thus these are items system designers should keep in mind:

1. Moderate loop sizes are easier to test since fewer control and data operations have to be tested.
2. All registers which appear in the righthand of an assignment in a loop should also appear as the lefthand of an assignment in the same loop. Thus the loop need not be exited to set up test data via ORing data to the Bus. This would speed up testing and cut back on the number of necessary alterations to the SUT.
3. After a branch the system should perform dissimilar data operations. Thus by observing the data on the Bus after a branch the direction of the branch can be determined without having to observe the branch Boolean.
4. Loops should be designed so that they exit on conditions which can be established by ORing data to the Bus. For example, a loop should exit on negative values of the Bus Sense Register rather than positive values since a leading one bit (negative value) can be OR'ed with a zero (positive value) to force a one on the Bus but not vice versa. Recall that since data appears in negative logic on the Bus, this one will appear as a logical low signal. As another example, loop exit conditions when sensing a bit in a register should consist of that register bit being one since a one can be OR'ed onto the Bus. Thus loops can be exited easily under control of the tester without requiring extra control wires to force the exiting condition.

Finally, consider the testing program. The program could analyze the flowchart input and decide on what test data to use. Such a program would have to be fairly sophisticated. An alternative would be to use a random number generator to select test values. To predict the outcome of a branch, the program would simulate both paths with the selected random number until the branch taken is uniquely predicted. The test is accepted or rejected depending on whether the desired portion of the branch was taken.

The random number test generator would require many simulations if the branch condition, was skewed to large or small numbers, as is often the case. For example, if the branch condition is  $n < 64$  (for a loop) the loop would be exited on a simulation with a probability of  $(2^{16} - 64)/(2^{16}) = 32704/32768$  (assuming a uniform distribution of random numbers) and re-entered with a probability of  $64/32768$ . Testing the looping condition would be very inefficient in simulation time. Often a loop contains a counter and only exits when the counter is zero or equal to some power or two (which can then be used as a

Now let us explore the spectrum of capabilities for the controller of the tester. The controller could inhibit the Bus DONE\DONE signal until it could analyze the data appearing on the Bus. Since all data transfers appear on the Bus, the controller can observe them all. Before releasing the DONE signal the controller could OR data to that already on the Bus. Thus with very little alteration of the SUT the controller can achieve a small measure of observability and controllability.

The fewer assumptions made about the nature of failures in the SUT the more observability and controllability that will have to be built into the controller. Figure 15 depicts some of the other capabilities that can be built into the controller. The further down the list an item is, the more the SUT has to be altered. Let us look briefly at these other controller capabilities. In order to tell which branch was taken it may be necessary to observe a selected number of Boolean values. To speed up the testing procedure it may be necessary to start the SUT at points other than the beginning; for example, at the beginning of a loop or 8-way branch using a ((manual evoke). Finally, some added control of the system can be achieved by wiring in some additional functions which do not exist in, the SUT. Additional functions such as clearing selected registers or setting selected Booleans could help place the SUT in states which facilitate testing.

### Observability

Monitor Bus

Observe selected Booleans

### Controllability

OR data to Bus

Control added Kme

Evoke single simple  
actions which are  
not already wired.

Fig. 15. Table of controller capabilities.

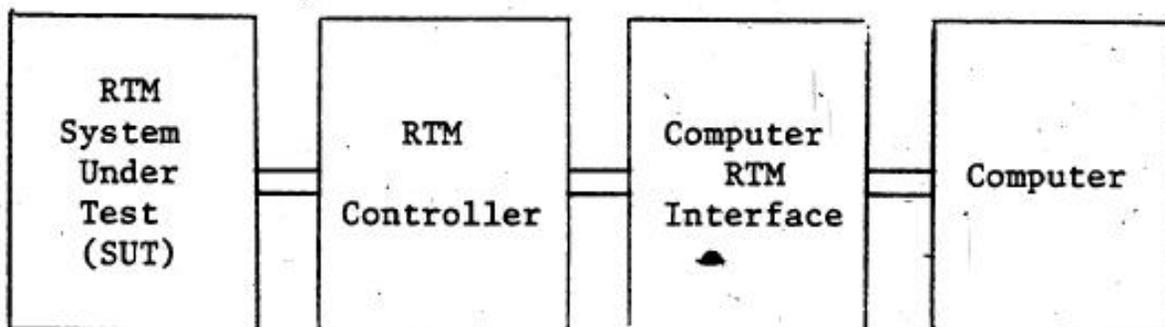
One other capability that could be added to the controller to speed up the testing process is a counter. Every branch of a SUT has to be examined for proper behavior. Normally the steps before a branch will be tested on the first time, down one path of a branch and need not be tested again. The counter can be set to .a value by the test program and the computer would not be interrupted by the controller until the number of DONE signals indicated in the counter have elapsed. This counter allows the SUT to proceed in almost normal speed over paths that have already been tested without wasting time for the interrupt request. and processing procedure of the computer.

Now consider, the test program in the computer. To be a general purpose tester, the program would

accept a description of the SUT, probably in flowchart form, and conduct the system test and diagnosis automatically. The program could simply simulate the actions of the SUT and, if complete observability and controllability were available, compare predicted data transfers with those from the SUT. Mismatches would indicate faults. Ideally, however, the program should make as much use of the wired-in control structure of the SUT as possible so

These are usually constructed around stored program computers. Also, such a tester could be used in debugging. A detailed design of an automated tester will not be presented. Rather, some of the considerations in the design of such a tester will be examined. Based upon the system failure assumptions, cost, and time limitations, the reader can design an automated tester to meet specifications for various types of systems.

The block diagram of a general purpose, automated tester is shown in Figure 14. The RTM system under test (SUT) is observed and, to varying degrees, controlled by a controller built from RTM's. The controller passes information about its observations to the Computer-to-RTM interface. A program in the computer then determines the next course of action and informs the controller. The controller performs the requested action and observes the results. The cycle is then repeated.



**Fig. 14.** Block diagram for a general purpose automated tester using a stored program computer.

Unlike acceptance testing where at most one or two modules at a time are under test, a system undergoing maintenance testing could have any number of faulty modules in addition to having any number of wires open or short circuited. The system testing problem is thus much more complicated than module testing. The assumptions made as to the nature of failures in the SUT largely establish the sophistication required for the test program and controller. For example, it could be assumed that all the modules have been tested (using module acceptance testers) and only the wiring needs to be checked. Checking the wiring is relatively easy since the controller need only apply voltage to a pin and observe whether the appropriate pins, as determined from the wiring list which would be given as data to the computer, also have a voltage appearing on them. Assuming that the modules are fault free, however, requires dismantling the SUT and a set of module testers.

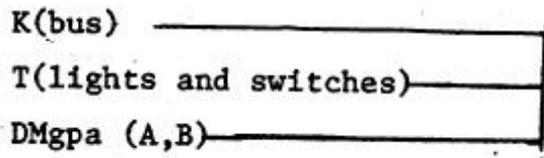
At the other extreme, no assumption about the SUT is made except that at some former time it performed according to specification. A very intelligent test program is required under this assumption

An assumption which is intermediate to these two extremes is that the data part is fault free (through

module acceptance testing) and only the control modules or wiring may be faulty. This assumption is of value when the data modules are relatively few in number (compared to the control modules) and may be tested on site or shipped to a manufacturing facility for testing.

```

    ↓ entry
    A ← Switches
    BSR ← A
    A ← 0
    A ← A+1; Set OVF
    ↓
    OVF yes → error
    ↓ no
    A ← 0
    A ← A-1; Set OVF
    OVF no → error
    ↓ yes
    A ← A+1
    A=0 no → error
    ↓ yes
    A ← A+1
    A=0 yes → error
    ↓ no
    A ← A-1
    A<0 yes → error
    ↓ no
    A ← A-1
    A>0 yes → error
    ↓ no
    A ← A+1
    A>0 no → error
    ↓ yes, exit
    A ← A-1
    A ← A-1
  
```



```
A ← A-1  
A < 0  $\xrightarrow{\text{no}}$  error  
      ↓ yes, exit
```

Fig. 13. RTM diagram for a K(bus) test.

All eight combinations for each bit should be tested if the arithmetic add is to be completely functionally tested. A set of test values which accomplishes this is:

A	0000....0	A	0000....0	A	1111....1		
B	0000....0	B	1111....1	B	0000....0		
A	0101....01	A	1010....10	A	1111....1	A	0101....01
B	0101....01	B	1010....10	B	1111....1	B	1111....11
A	1010....10	A	1111....11	A	1111....11		
B	1111....11	B	0101....01	B	1010....10		

Note that many of these test sets were used for the testing of logical shifting, and storing operations.

### Solution 3

The next step is to automate the manual test. Each function could be tested completely before proceeding to the next one. In some cases, especially the logical functions, several operations could be performed on a set of test data before the result was compared to the predicted result.

The test sets could be automatically generated. For example, an alternating zero and one vector could be stored as a constant and both alternating zero and one vectors could be generated by retrieving the data and performing a single end around shift. A memory could be used to store the predicted results of each test for comparison to the calculated test result.

### Solution 4

Instead of a memory to store the predicted test results a second DMgpa could be used as the predictor. This second DMgpa could be either a known non-faulty module or a second module under test.

### Kbus TEST

The bus sense module could be tested as a portion of any of the other module testers. However, a separate, straightforward Kbus sense test can be developed as shown in Figure 13.

First, the A register is loaded from the switches. The ability to load the bus sense register is then tested. The load zero on to the Bus is exercised and- the ability of the Bus to store the overflow from the DMgpa is tested. Finally the Bus tests  $BSR = 0$ ,  $BSR < 0$ , and  $BSR > 0$  are exercised.

## MAINTENANCE TESTING

Maintenance testing is performed on a system which is in the field and has already undergone debugging and module acceptance testing. Thus the system is known to have been operational at some time. The maintenance testing could be performed periodically to insure the system functions as specified. It also could be performed on a faulty system to facilitate repair and restoration.

In any event, maintenance testing can be conducted manually using techniques similar to those used in system debugging. However, manual testing can be very time consuming and usually requires an intimate knowledge of the system under test. For these reasons a general purpose, automated tester might be desirable.

412

[previous](#) | [contents](#) | [next](#)

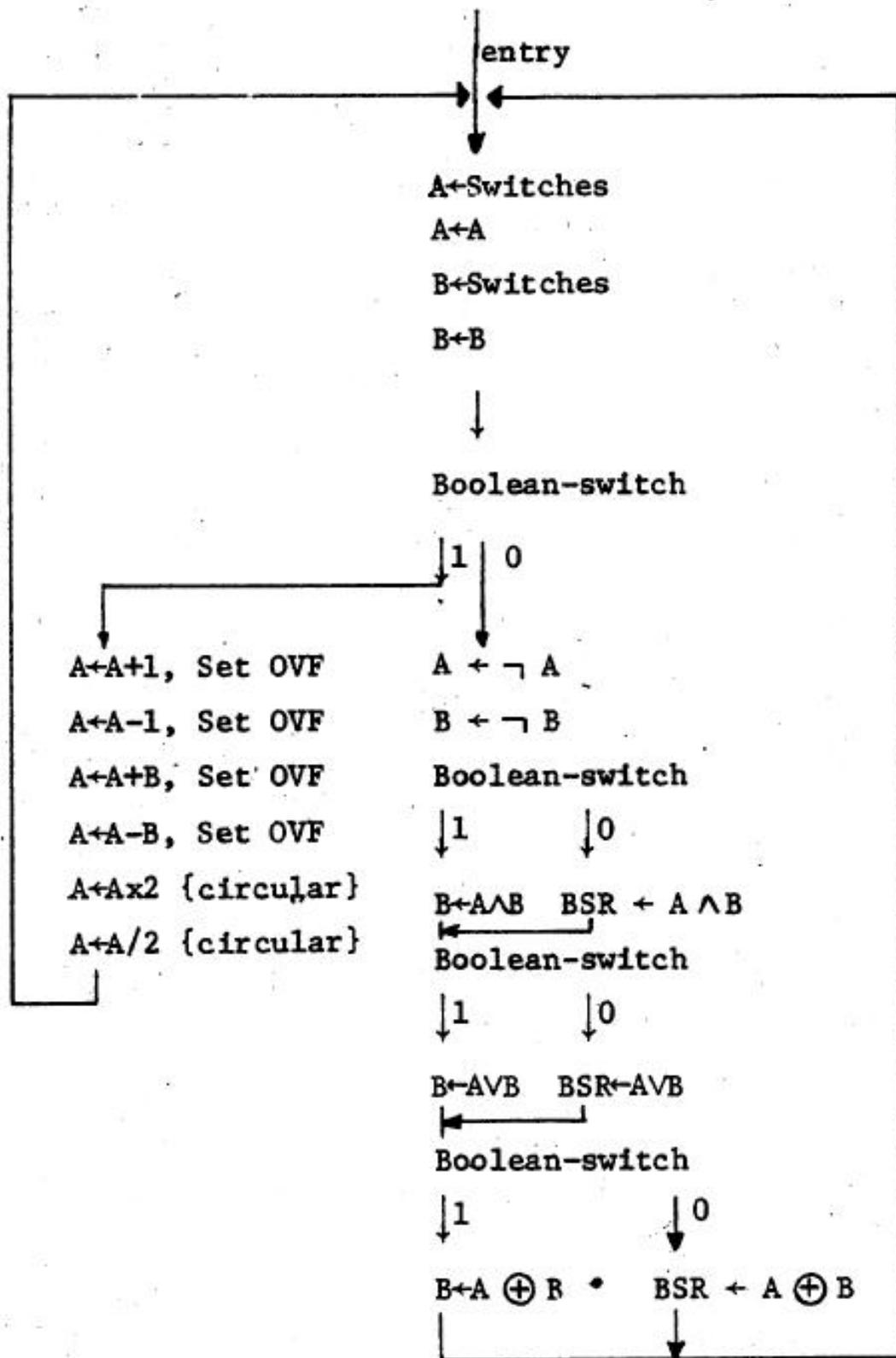


Fig. 12. RTM diagram for an improved manual DMgpa test.

one into a zero and the even bits' ability to shift a zero into a one. A second shift tests the odd bits' ability to shift a zero and the even bits' ability to shift a one. Next, patterns of alternating pairs of ones and zeroes are shifted to test the ability of the register to shift ones into ones and zeroes into zeroes. (If memory tests have already been planned for the register, these latter two patterns become redundant.)

Testing the more complex arithmetic operations is not as easy. For example, recall the table for the addition of two bits given in Chapter 5.

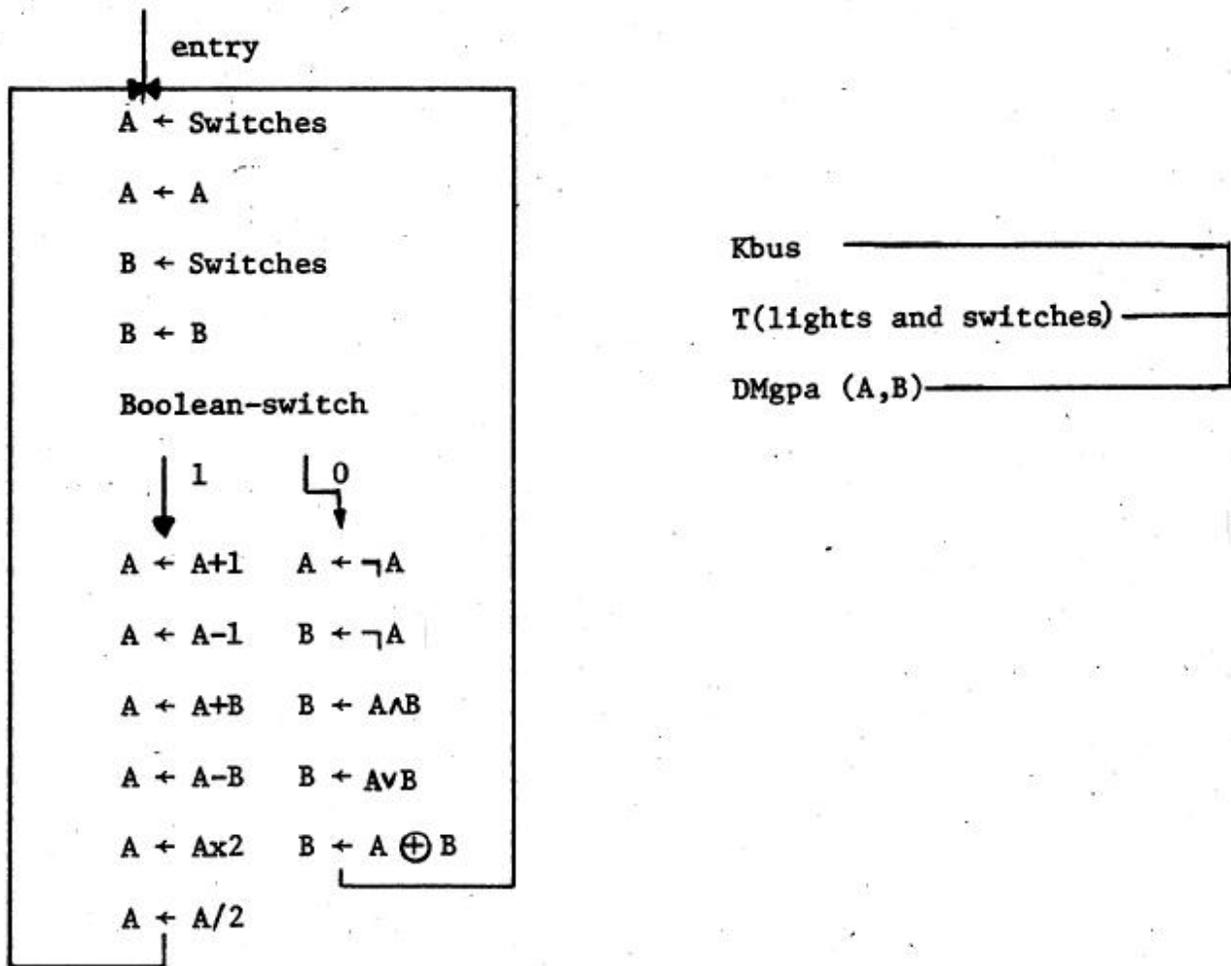


Fig. 11. RTM diagram for a manual DMgpa test.

1,  $A \leftarrow A+B$ ,  $A \leftarrow A-B$ . The overflow, and the three Bus indicators zero, positive, and negative, should also be connected to a light output. Also a K(branch) can be used in conjunction with Boolean switches to select one of  $B \leftarrow A \wedge B$ ,  $B \leftarrow A \vee B$ ,  $B \leftarrow A \sim B$ ,  $BSR \sim A \wedge B$ ,  $BSR \leftarrow A \vee B$ ,  $BSR \leftarrow A \sim B$ . When the Boolean switch is a 0 the contents of register B are not altered (the results of the operations are simply put on the Bus). Thus the contents of B can be different from the contents of A and the Exclusive OR function can be properly tested. These alternatives to solution 1 are shown in Figure 12.

For a memory cell a good set of tests consisted of all zeroes, all ones, walking zero and walking one. Since logical operations are performed on a bit-wise basis the following combinations for the A and B registers will test the logical operations exhaustively:

<b>A - all zeroes,</b>	<b>B - all zeroes</b>
<b>A - all zeroes,</b>	<b>B - all ones</b>
<b>A - all ones,</b>	<b>B - all zeroes</b>
<b>A - all ones,</b>	<b>B - all ones.</b>

To test the-shifter part of DMgpa, a pattern of alternating ones and zeroes is first used. The first shift tests the ability of half the bits, say the odd ones, to shift a

410

[previous](#) | [contents](#) | [next](#)

Subroutine: Write and Check starting at Limit and going to 0

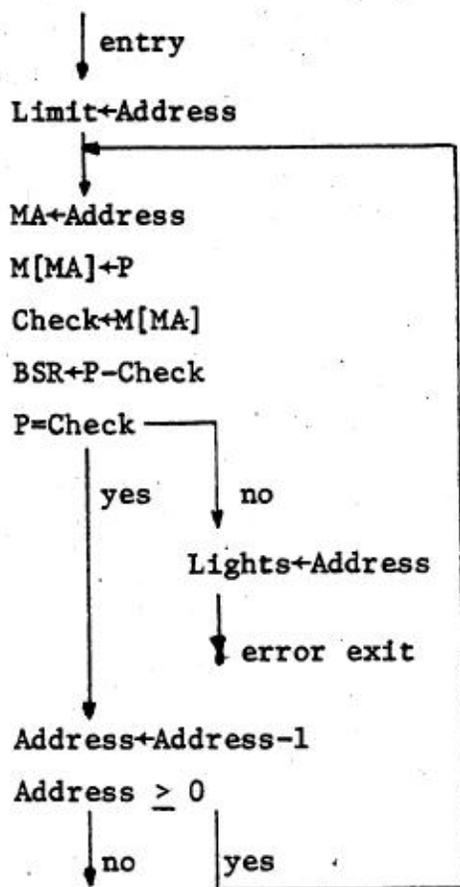


Fig. 10. Flowchart of automatic test Write and Check.

themselves. With the tester in the single step mode the contents of the A and B registers can be tediously checked with a voltmeter at the end of each step to insure that the predicted and actual contents of the registers coincide.

A manual Boolean-switch determines whether the data that has been stored in A and B will be used to test the arithmetic data operations or the logical data operations. Register A is wired for circular (end-around) shifting, (i.e., LSI is connected to A<0> and RSI to A<15>).

As in the previous manual testers, the DMgpa test should be run with several different sets of input data. For example if A contained all ones then  $A \leftarrow A+1$  would yield all zeroes and help to check carry propagation.

## Solution 2

Note, there is no test for overflow. Also, the Exclusive OR cannot be completely checked (i.e., just prior

to the  $B \leftarrow A \sim B$  operation  $B$  contains  $A \vee (A \wedge B)$  which is identical to  $A$ , thus  $B \leftarrow A \sim B$  will always yield all zeroes).

The overflow should be enabled for the operations  $A \leftarrow A+1$ ,  $A \leftarrow A$

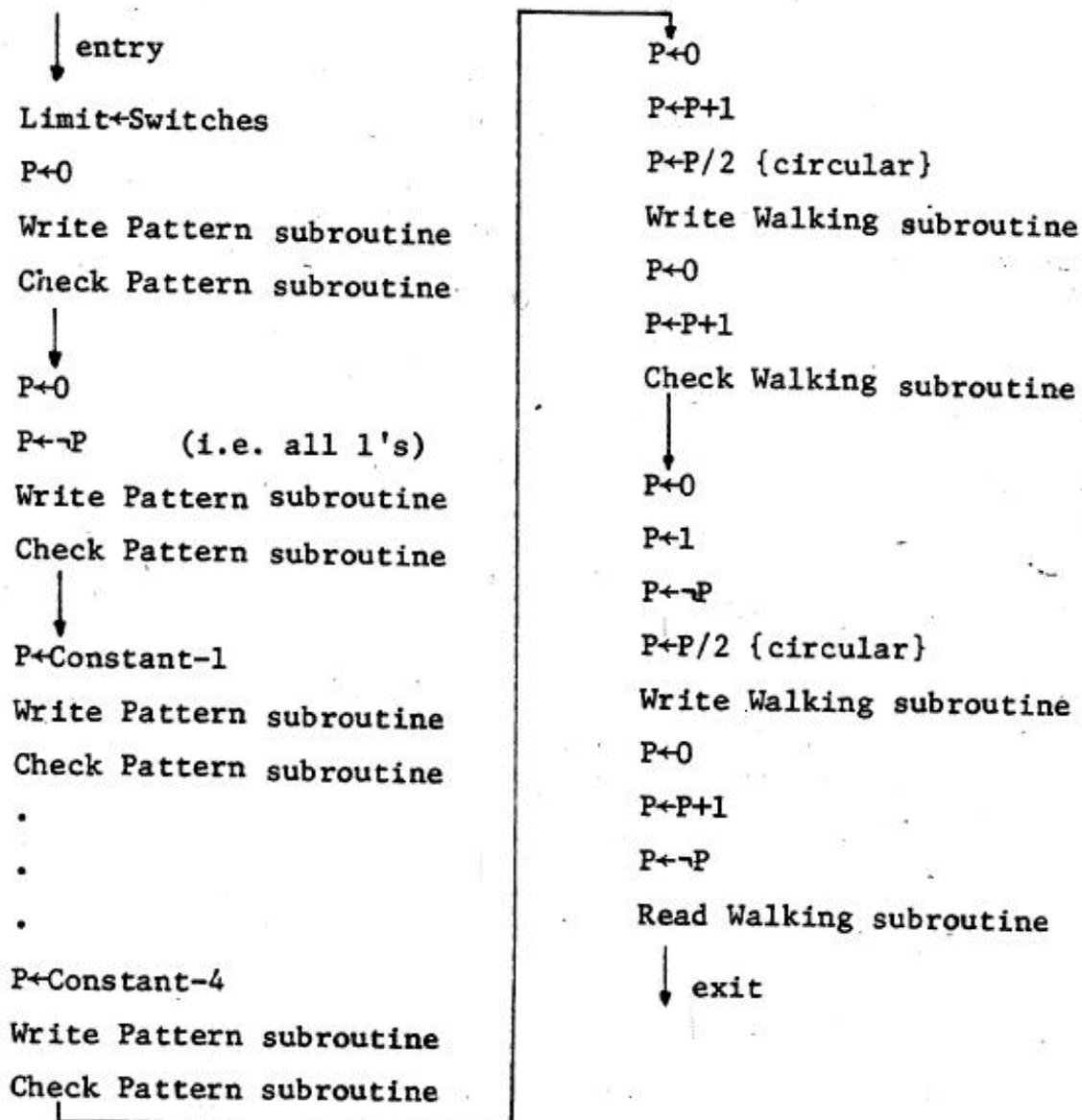


Fig. 9. Memory tests for 0's, 1's, and constants, and Walking 1 and Walking -1.

## DMgpa TEST

The DMgpa is yet a *further* generalization of the memory cell. It has two temporary storage cells, the contents of which are operated upon by many data transformation operations. These operations must be tested. The large number of data operations greatly complicate the required testing structure.

## Solution 1

The simplest test consists of manually exercising the various functions of the module and observing the results, as shown in Figure 11.

The A and B registers are loaded from the switches and stored back into

408

[previous](#) | [contents](#) | [next](#)

Subroutine: Check Walking starting at Address 0 and going to Limit

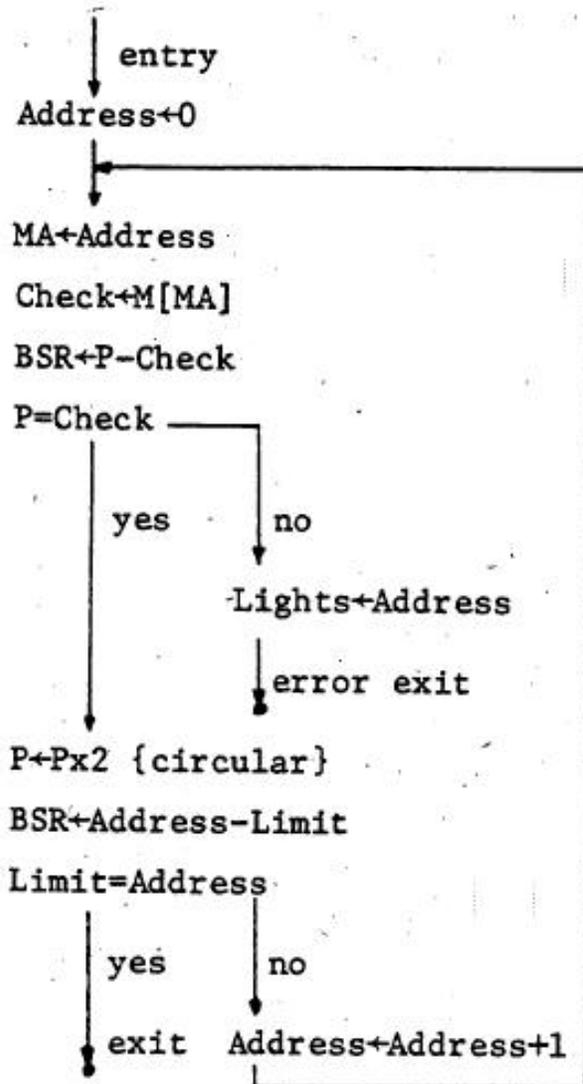


Fig. 8. Control part for Check Walking subroutine.

Since the transfer register contains only one memory word it therefore can be tested exhaustively. The automatic generation of tests would be very simple using a counter. But since the mapping function from input to output could be any bit mapping, the predicted outputs are difficult to generate. The simplest solution would be to store the predicted results in a memory. Storing results would require  $3 \cdot 2^{16}$  memory words ( $2^{16}$  test results for each of the full word, upper half word, and lower half word).

A subset of all possible tests would reduce the memory size drastically. The all-zeroes and all-ones tests

would serve to test stuck-at-one and stuck-at-zero faults, respectively. The all-ones test would also detect reading errors associated with reading and storing half words. Finally a walking/one test could be used to check the input to output bit mapping pattern. This scheme would require about 50 memory words.

An automatic tester which is independent of the mapping function could also be fabricated. Such a scheme would test two identical registers at the same time (or alternatively check one against a known good transfer register) using the registers themselves to provide the predicted test results.

Subroutine: Check Pattern starting at Address 0, and going to Limit

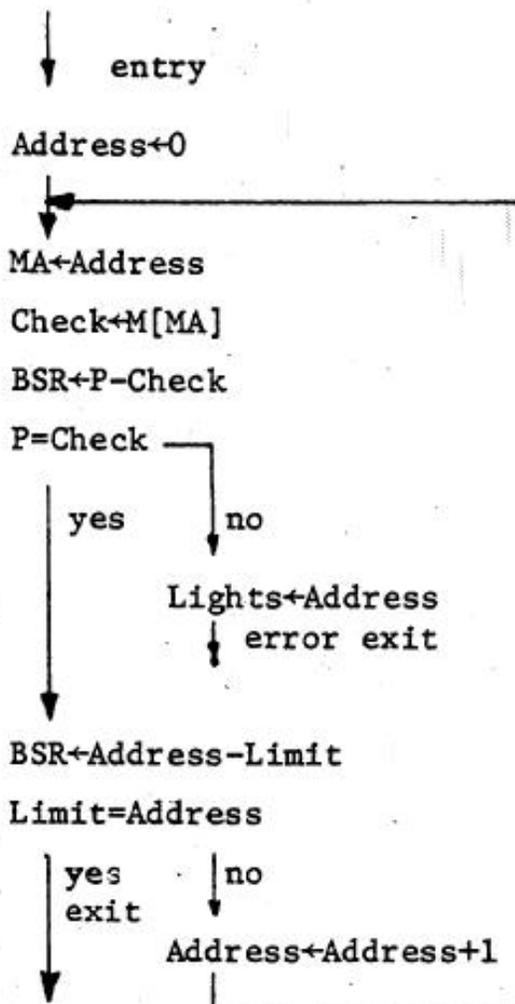


Fig. 7. Control part for Check Pattern subroutine.

## M(transfer) AND M(byte) TEST

The Mtr and Mbyte modules are used both for temporary storage and for mapping register bits into a different output configuration, as used in data packing operations. In testing M(transfer) and M(byte), tradeoffs exist between time to test, hardware, and the completeness of the test.

The transfer register can be considered to be a one word memory and thus be tested by the techniques established for memory testing. If the transfer register does not perform an identity mapping between its inputs and outputs, then the test is more complex -- also, a register transfer can be instructed to store half

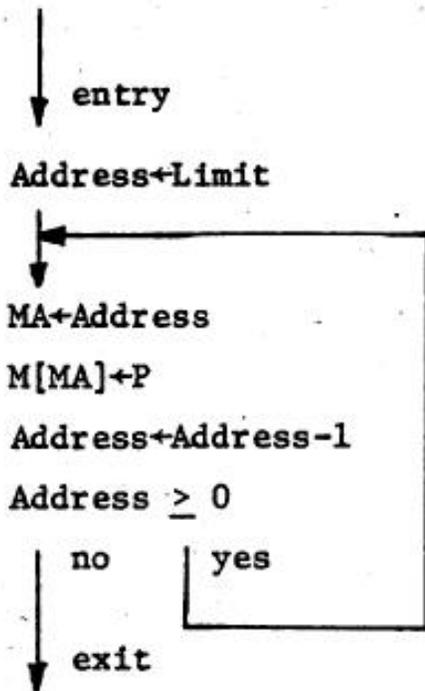
words.

Alternatively, a memory can be used to store the tests and predicted results. The resulting design is analogous to that for the memory test and will be left to the reader.

406

[previous](#) | [contents](#) | [next](#)

Subroutine: Write Pattern (in all cells starting at Limit and going to 0)



} Limit contains memory size  
 Pattern contains constant to write

Subroutine: Write Walking

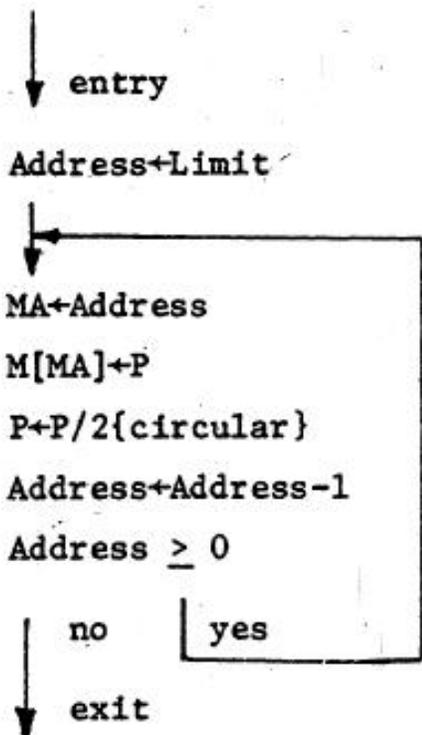


Fig. 6. Control part for write subroutines.

405

[previous](#) | [contents](#) | [next](#)

1. Address holds the current address being checked.
2. Limit holds the last address in the memory to check; it is constant and read from the switches.
3. Pattern\|P holds the pattern being checked depending on the test (i.e. 0's, 1's, constants 1,2,3, or 4, or walking 1, or 0).
4. Check is a temporary to be compared with constants.

Two subroutines, shown in Figure 6, are used to write the whole of memory with a test pattern. The first is for test patterns which are constant. Before calling the subroutine, Limit is initialized with the memory size. The constant test pattern is read into memory from Pattern\|P. The second subroutine generates a walking pattern of the input data by shifting Pattern to the right;

Likewise, two subroutines check the test patterns; they move through memory in opposite order from the order in which it was written. The read constant subroutine is shown in Figure 7. Register Address, the current memory address, is initialized to zero before the subroutine is called. The memory content of the current memory address is compared to what was written. A mismatch causes the current memory address to appear on the lights and the test system to halt. Otherwise, the end of memory is tested for by comparing the current memory address with the end of memory. The subroutine is in a loop until memory is completely tested.

The compare subroutine for walking test patterns is similar except the test pattern' is shifted to the left since it is being read in reverse order. This subroutine is given in Figure 8.

Finally, the main flowchart is shown in Figure 9. The memory size is first loaded into Limit from the switches, and must be a multiple of 16. The current memory address is initialized to the memory size and the test pattern is all zeroes. After the memory has been written the current memory address is initialized to zero and the memory is read in reverse order from the order in which it was written. Similarly, the all-one test pattern, the four test patterns of alternating blocks of ones and zeroes, and the walking-one and walking-zero test patterns are tried. Note the blocks of ones and zeroes are applied separately rather than in conjunction in two sets of two as suggested. These patterns could be applied as walking test patterns or could be applied in sets of two by alternating the read and write constant pattern subroutines to operate only on every other memory cell and calling each subroutine twice, once for each member of the sets of blocks of ones and zeroes. This exercise is left to the reader.

One should note that the walking-one (zero) constructed for the read operation assumes that the memory size is a multiple of the number of bits in the word (i.e. 16). Otherwise, since the read subroutine reads the memory in the opposite order from which it was written, provision must be made so that the proper constant is generated, i.e., the constant that was written into memory location zero.

Finally, the memory test scheme can be altered to test the maximum read- write speed of the memory instead of the long term storage ability, as was tested by the last solution. Instead of separate write and read subroutines, one write- read subroutine, as shown in Figure 10, would be used. The necessary alterations to the main test system flowchart are straightforward and left as an exercise for the reader.

Fault diagnosis can be assisted by observing the contents of the Lights which indicate the memory cell address when the mismatch occurred. Examining the contents of Pattern would indicate the test pattern that detected the failure and perhaps give a clue to the reason for failure.

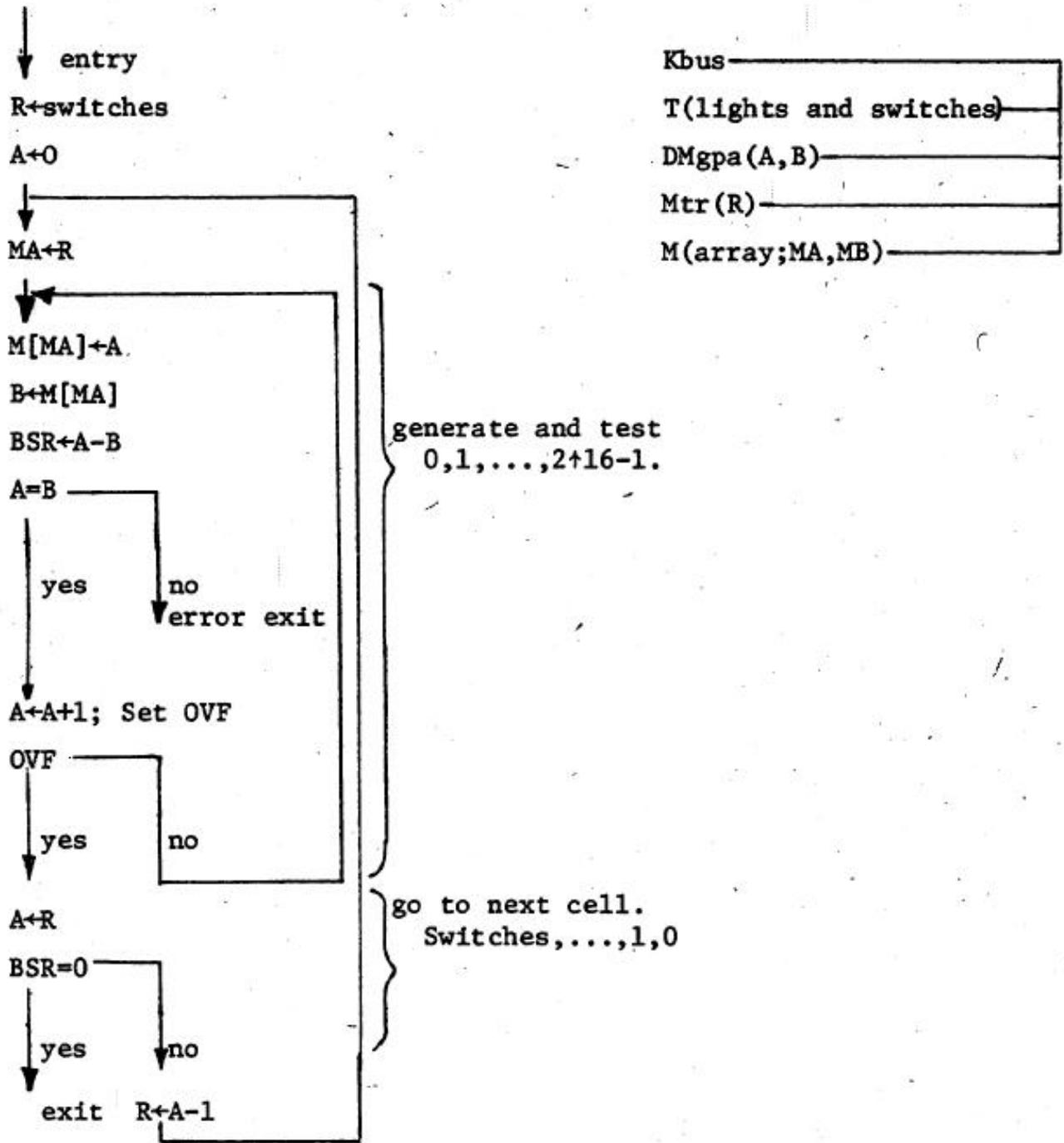


Fig. 4. RTM diagram of a memory array using exhaustive test.

[previous](#) | [contents](#) | [next](#)

Unlike the other memory tests, this solution-tests the long time storage ability of the memory since the entire memory is written into before it is read. The other two solutions tested, whether the memory could operate at maximum speed by writing into a cell and reading its contents immediately afterwards. Note further that this last solution reads into the memory in one order and writes out in another. This helps test the memory accessing mechanism.

All the solutions presented so far have consisted of a single write into a cell followed by a single read. Some faults could go undetected. For example, if all zeroes were read into a cell a /stuck-at-logical zero fault in that cell could go undetected.

#### Solution 4 - Exhaustive Testing

Thus for 100% confidence it would be necessary to test each memory cell exhaustively, as in this solution (Figure 4). The R register obtains the memory size from the switches. The inner loop starts with the A register containing zero and alternately writes and reads a memory cell until all possible combinations have been tried (i.e. until  $A = 2^{16}-1$ ). The outer loop insures that all memory cells are tested.

For a 4K, 16-bit memory this would require  $2^{12} * 2^{16} = 2^{28}$  tests. Even at one microsecond/test (which this does not achieve) this would require about 4.5 minutes. For larger memories exhaustive testing is prohibitive and even for this size memory it may be prohibitive.

#### Solution 5

A compromise is to apply to each cell several carefully selected tests so that if each cell passes these tests one can be fairly sure the memory is fault free. This solution summarizes all we have thus far learned.

To detect stuck-at faults (i.e., fixed at either 0 or 1) each cell should be tested with a pattern of all zeroes and a pattern of all ones. Historically, memories have also been tested for possible coupling between separate memory words. This has been accomplished by using the so-called walking-one and walking-zero patterns. In the walking-one pattern there is only one bit at a logical one level and the pattern written into successive memory words is:

```

0 0 0 ... 0 0 1
0 0 0 ... 0 1 0

1 0 0 ... 0 0 0

```

The walking-zero pattern is similar with the rules of one and zero interchanged. Other useful patterns, for detecting possible interference between bytes, are:

```

Pattern (1)  1 1 1 1    0 0 0 0    1 1 1 1    0 0 0 0
Pattern (2)  0 0 0 0    1 1 1 1    0 0 0 0    1 1 1 1
Pattern (3)  0 0 0 0    0 0 0 0    1 1 1 1    1 1 1 1
Pattern (4)  1 1 1 1    1 1 1 1    0 0 0 0    0 0 0 0

```

These last patterns are applied in sets, patterns (1) and (2) forming one set and patterns (3) and (4) another.

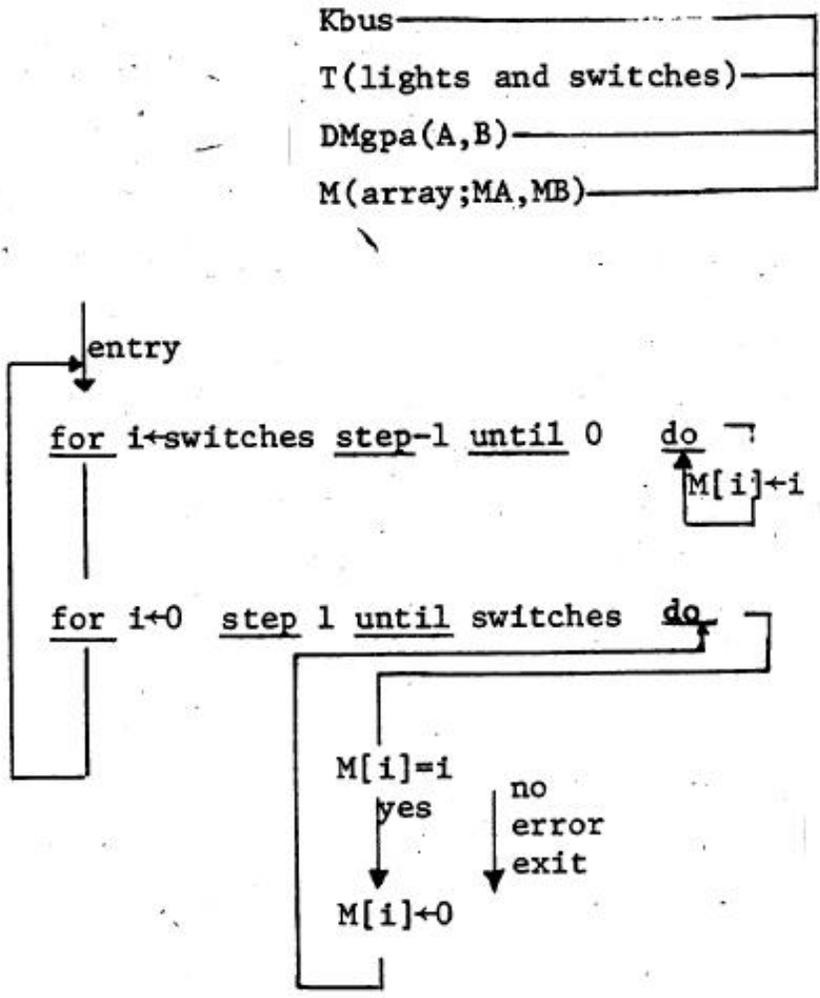
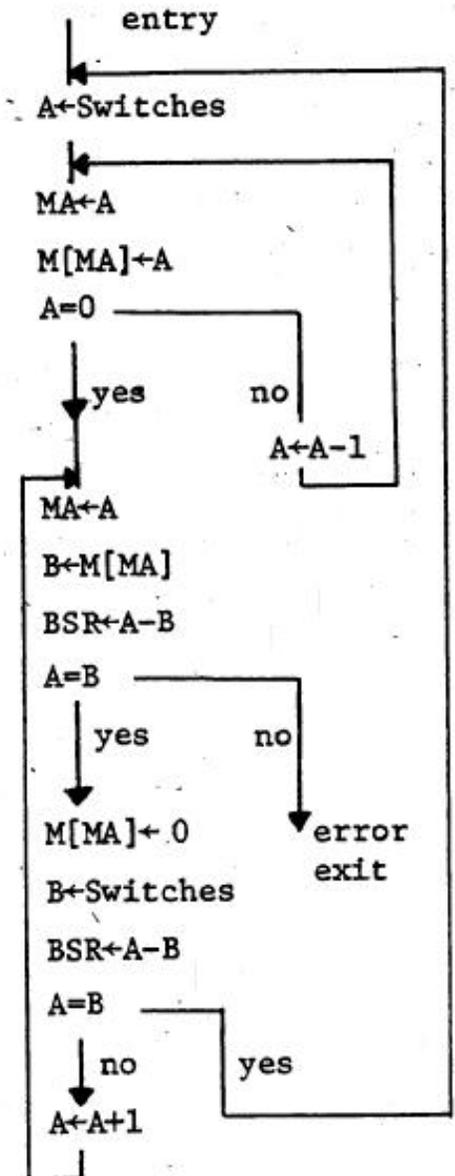
The data part of the test system is given in Figure 5. The registers are given names according to the function they perform in the checking:

memory 2 and compared. A mismatch indicates an error. The test iterates until each cell in the entire memory under test is examined.

This solution requires an extra memory. It would be desirable to generate the tests logically instead of relying on tests stored in a memory.

### Solution 3 - Testing With a Generated Function

By switching the method in which the tests are generated each memory cell can be tested for more than one value. This provides an automatic and faster test. Figure 3 shows this scheme. The final address is entered into register A via the switches. Each memory cell whose address is less than or equal to the final address has its address entered as data. Then starting from address zero the memory cells are read and compared to the data that was written in. A mismatch indicates an error. If the address of the current cell is equal to the final address shown on the switches the whole process cycles back to the start, otherwise more comparisons are made.



Kbus  
 T(lights and switches)  
 DMgpa(A,B)  
 M(array;MA,MB)

equivalent flowchart using K(for loop)

Fig. 3. RTM diagram of memory array test with generated data.

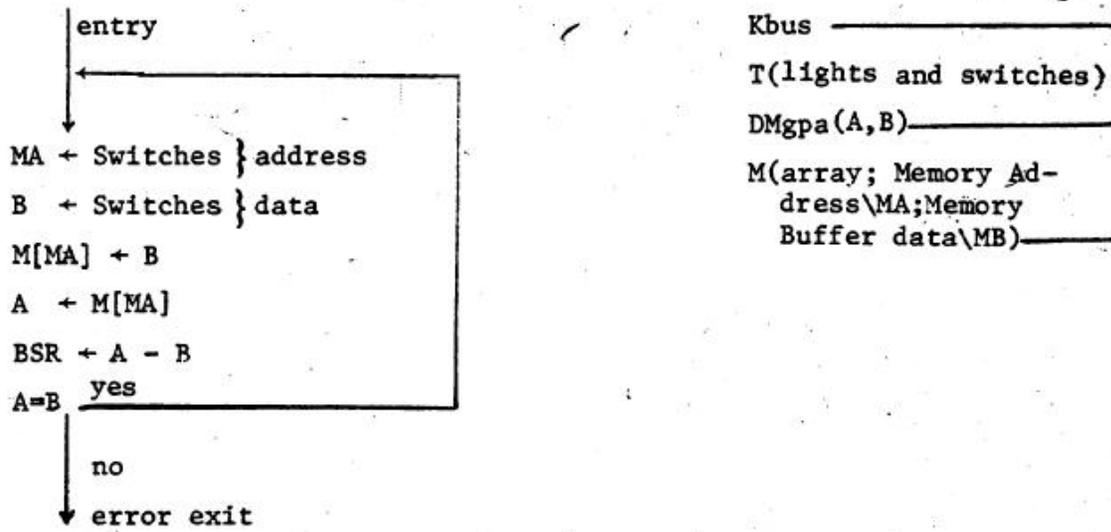


Fig. 1. RTM diagram of memory array test with manual inputs.

[previous](#) | [contents](#) | [next](#)

outputs are small, optimal testing is not significantly faster than exhaustive testing. Alternatively, these simpler modules can be tested as a part of a test system for a more complex module. Thus normal completion of the tasks of a test system imply all the modules forming the test system are acceptable. However, when several modules are concurrently under test an error indication might lead to a time consuming diagnosis to isolate the faulty module. For these reasons the following modules will be assumed to be tested exhaustively:

K(evoke)  
 K(branch-2-way)  
 K(branch-8-way)  
 K(serial merge)  
 K(parallel merge)  
 K(manual evoke)  
 K(subroutine)  
 DM(flags)  
 M(constants)

Example test systems using non-exhaustive schemes will be developed for the following modules:

M(array|registers|scratchpad)  
 DM(gpa)  
 M(tr)  
 K(bus)

## MEMORY TEST

The problem is to develop a test system for a read-write random access array memory module. Many of the concepts in test system design will be explored in detail in the following solution. Test systems for the other modules will thus only need to be sketched.

To test a module it is necessary to insure that the module performs its.. specified functions. The standard method for memory testing is to write a word into a memory cell and then later read the word and compare it to what was written. A mismatch indicates an error.

## Solution 1 - Manual Testing

An RTM solution is shown in Figure 1. The memory address to be tested is read into the MA register from the switches. Likewise the test data is read into the B register from the switches. The memory

address under test is written into and read out, and the two values compared. A mismatch indicates an error.

It is easy to see that this solution requires a significant amount of time to conduct the test, especially if the memory is large, because of the requirement of manual inputs. This method would be acceptable for M(transfer), however, since it has only one register. The system could operate at a higher speed by using a second switch register.

### Solution 2 - Testing Against Known Results.

In Figure 2, two memories are used: one stores the tests and the other is the system under test. The former could be a read only memory.

The memory size is input via the switches to R. The test data is retrieved from memory 1 and placed in memory 2. Next, the test data is retrieved from

becomes Low, then another data sender was also evoked and supplied the DR signal.

Care should be exercised when a register's content is checked by probing. For example, the latches of an M(transfer) are active as long as the evoke which enables them to read is active. Thus probing an M(transfer) while its read input is active could cause erroneous data to be entered.

Another way to detect incorrectly read data is to remove the receiver at each step in the flowchart and see if a DA signal is generated. If a DA signal appears, then a multiple read from the Bus was incorrectly performed at that flowchart step.

It should be noted that unless Boolean branch variables are brought out to light they are unobservable. An erroneous data transfer (appearing on the Bus) may be due to the system taking an incorrect branch. The branches can be forced to a one state by leaving their inputs floating. By removing the inputs to a Boolean branch of the flowchart, the system will take a path that is uniquely determined (assuming the K(branch) is not faulty). This will help detect wiring errors associated with the K(branch) modules. Observation of the Boolean inputs to the K(branch) modules at each branch situation of the flowchart will also help locate wiring errors.

Sometimes an RTM system will operate in manual-single step mode but will not operate properly at full speed. An oscilloscope can observe the K(evoke) following a K(branch). If the K(evoke) is supposed to be activated and is not, isolation is achieved by observing where the system first deviates from the flowchart. Checking earlier branch points is a good isolation procedure. If in doubt as to whether a timing problem actually exists, operate the system in manual-single step mode at different pulsing rates to see if the system operation is intermittently incorrect.

In order to facilitate debugging, the design should include testing facilities and a plan for testing. For example, software programs are debugged by dividing the program functions into several small subroutines and debugging the subroutines individually. By using K(subroutine) modules an RTM system can be divided into small, easy to test blocks. Also, (daisy) chaining wires is often suggested to distribute a signal to many pins. However, daisy chaining makes it difficult to trace wiring. For example, tracing a wire to a pin which has three wires attached to it means the traced wire could have one of two origins. If the traced wire does not have an active signal on it, detection of the relevant source wire is extremely difficult.

## ACCEPTANCE TESTING OF MODULES

After the modules have been manufactured they have to be tested to insure that they will meet specification. Normally this is done by the manufacturer, but in a laboratory environment it is possible to introduce defects. It will be assumed that an RTM system has been built that exercises all of the functions of the module to be tested. Thus a newly manufactured or an aged and suspected faulty module

would be plugged into the test system. Any detected fault in the module under test would cause the test system to indicate an error. If the test system completed its designed operation, then the module under test would be accepted as fault free. Some of the modules, such as K(branch) and K(evoke) are easy to test on a functional basis, while others, such as a DMgpa, are quite complex. The simpler modules can be functionally tested by observing the module outputs for all possible module inputs. Since the number of inputs and

numerically controlled wire-wrap machine. (Such a program has been written in the BASIC language, and is available from DEC.) In most instances the prototype will be a one-of-a-kind project and can be wired by hand. The following techniques can be used to validate the wiring.

1. Check to see that the Bus and T(lights and switches) to K(bus) are wired correctly. The lights and switches are the chief manual debugging aid.
2. Recheck every connection to reduce mistakes. (When the wiring is complete for a large system, the mounting panel is crowded with wires and wire tracing is difficult.)
3. The wiring may be rechecked by performing a continuity test (e.g., using an ohmmeter) between pins which, according to the wiring list, are connected. Also check shorts to ground. Check that each pin has the expected number of wires on it.
4. Insert the modules and apply power. If the power supply light does not go on, turn the supply off and recheck for open and short circuits.
5. With power on, check to see if the DR and DA are High (lights on). If the DR and DA are not High, the problem may be:
  - (a) Module(s) missing or not fully inserted.
  - (b) Modules may be touching each other on the back of the panel.
  - (c) Modules in wrong slot.
  - (d) Modules incorrectly wired.
  - (e) Bus voltage may be low because the Bus is being loaded by an erroneous signal. Try removing modules on a one-by-one basis until DR and DA become High (lighted).

## DEBUGGING

Test the system to see if it conforms to specifications. Place the system in the manual mode and verify that the system does what is expected by observing the lights and depressing the single step switch. Each time the single step switch is depressed the system will execute until the next K(evoke). step. (The system will not stop at K(branch) and K(merge) steps.) The system can be wired so that at each step the transferred data appears on the lights. The single step mode coupled with observing the lights is analogous to placing a print statement after each instruction in program debugging. Since each step is so

small system debugging is normally quite easy - although tedious.

As in program verification, the data used for RTM verification in this mode should be representative of what the system will encounter during actual operations. Therefore, debugging must be accompanied by a plan. For example, if the system is to read ASCII characters from a tape, valid eight level ASCII characters should be used as test data. All branches of the algorithm should be tried. Further, if the system is to interface with some external device, the input/output of the RTM system., should verify that the data exchange occurs in the proper sequence.

If an unexpected data transfer appears on the lights, then usually the sender or receiver of data on that step is in error or a wrong function was evoked. However, some wiring error other than those associated with the sender and receiver of data for that 'operation might exist. For example, if the Bus is shorter than planned for (common in laboratory experiments) the effect is to produce zeroes in the data transfer.

Incorrect data transfers could also result from multiple evokes being active. To check for this, remove the data sender wire and observe the DR light. If DR

## CHAPTER 9 SYSTEMS AND COMPONENT TESTING

by Daniel Siewiorek .

As anyone who has programmed a moderate-size problem on a digital computer knows, the design and coding of the problem is only half the job. Even in a large, fast system it may be impossible to test exhaustively all cases to validate a design. Insuring that a program performs as intended may be a formidable task. Assemblers and compilers help isolate syntactical errors but, are of little aid in detecting errors in logical flow. At program run time only a few errors, such as division by zero, will be detected by systems software and hardware. In consequence, the designer is currently faced with logically validating a system with little automatic assistance either in the planning or running stages.

Many of the same problems that face the programmer also face the hardware designer. Logical errors can be produced by errors in the fundamental design algorithm, as well as in the hardware implementation of the algorithm, e.g., wiring. errors. Before a system is correct the programmer must insure that' a program meets its specification; So, too, the logic designer must validate his design.

Logic design validation usually takes one of three forms: (1) system development, e.g., debugging the system prototype; (2) manufacturing acceptance testing of the individual components which make up the system; and (3) systems testing. Systems testing usually takes place after the system is first assembled. However, maintenance testing of the system after it has become operational is - also an important, aspect of system testing. Each of these three types of logic design validation requires different basic assumptions. The assumptions underlying logic validation and the various techniques to assist testing will be described below.

### SYSTEM DEBUGGING

The basic operations performed by the RTM's are analogous to those performed by a simple stored-program computer. The stored program is hard- wired in the control portion of the RTM system. Indeed, the flowchart used in designing an RTM system is similar to an assembly language program for the simple computer. Thus, many of the suggested techniques for RTM system debugging can be borrowed from software debugging techniques.

### VALIDATION OF SYNTAX

An assembler for a computer will check to see if a program consists of correctly formed and valid operations. The counterpart of syntax analysis in the RTM system is to check if only allowable functions are used. For example, in a DMgpa,  $A \leftarrow Ax2$  is available whereas  $A \leftarrow Bx2$  is not. A further consideration of the RTM syntax is that the proposed wire list performs the functions as specified in the flowchart (analogous to determining the octal machine codes for mnemonic instruction names.)

An assembler-like program could be written to check the proposed flowchart and wiring list. In lieu of such a program the validation must be done manually by checking the proposed operations against those performed by the RTM's and by checking the wiring list.

An RTM assembler-like program could further yield as output a tape for a

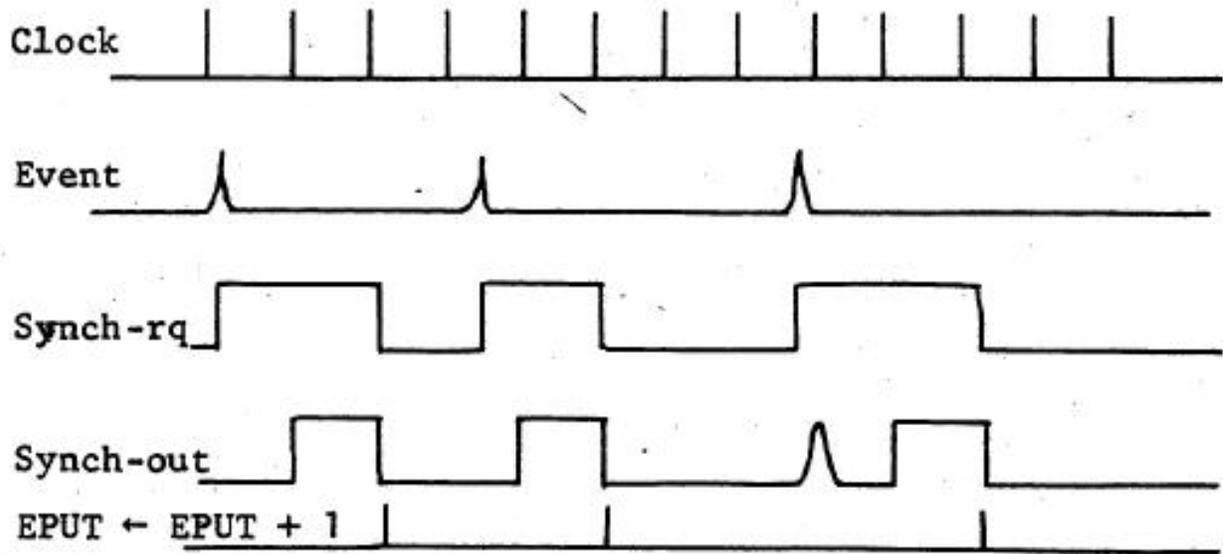
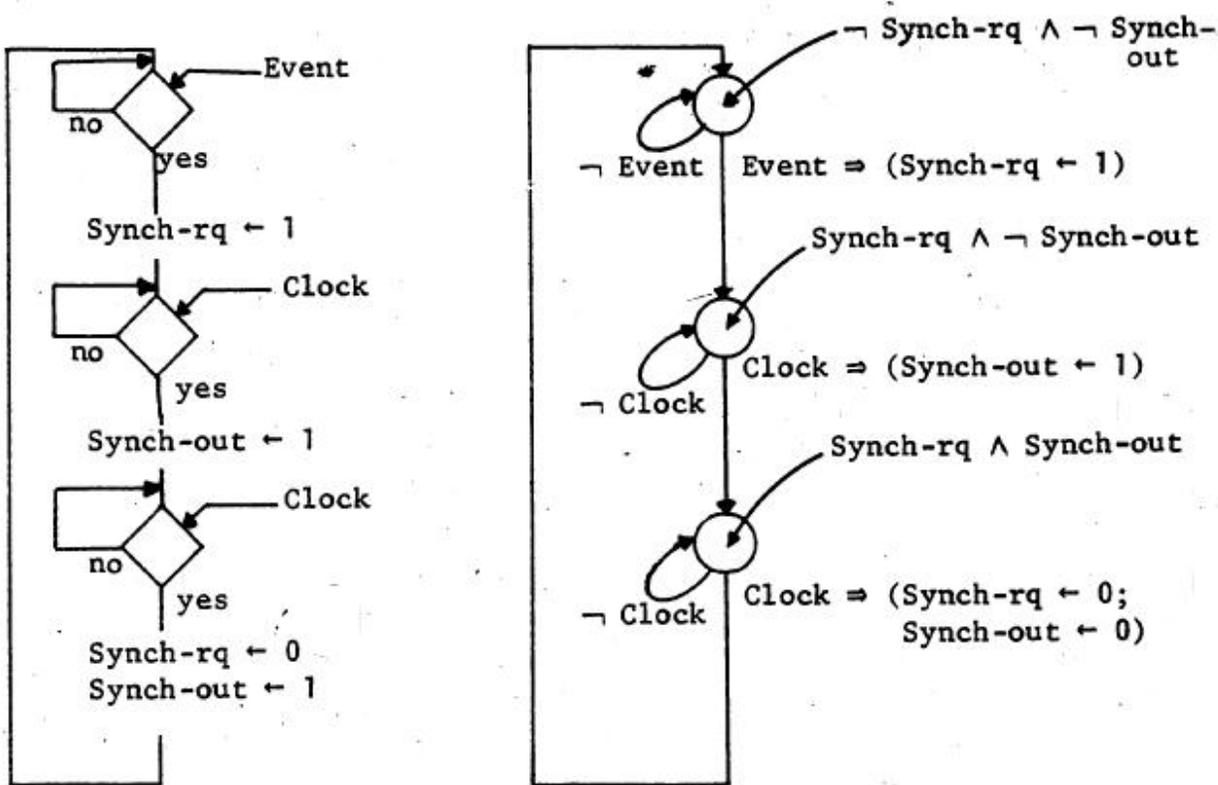


Fig. EPUTsc-5. Timing diagram for K(synchronizer).

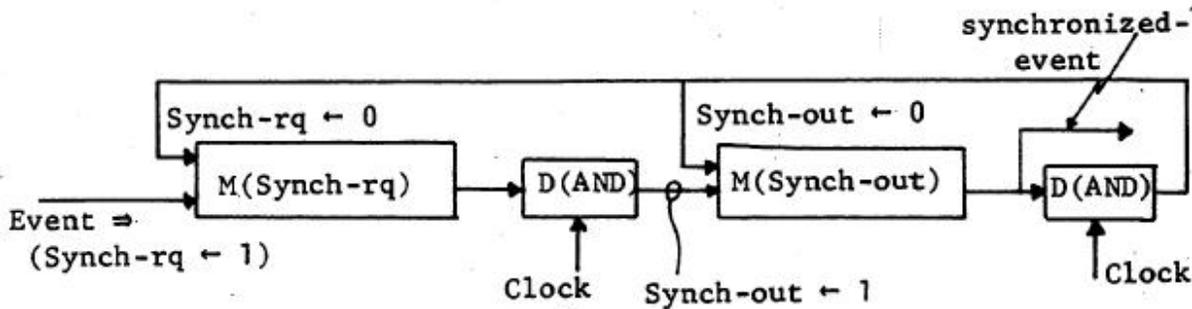


a. Flowchart

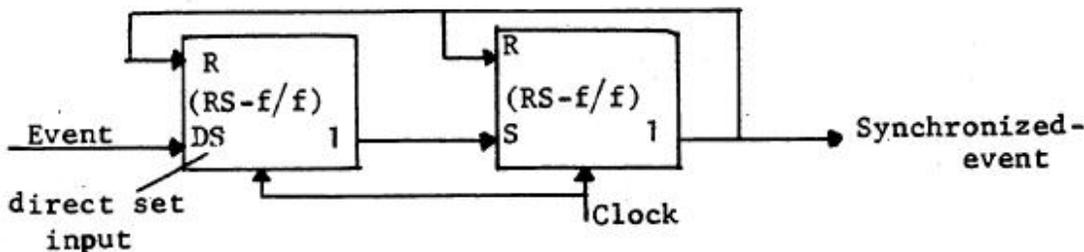
b. State diagram

Event  $\Rightarrow$  (Synch-rq  $\leftarrow$  1)  
 Clock  $\Rightarrow$  (Synch-rq  $\Rightarrow$  (Synch-out  $\leftarrow$  1);  
 Synch-out  $\Rightarrow$  (Synch-out  $\leftarrow$  0; Synch-rq  $\leftarrow$  0))

c. ISP description



d. Switching circuit



input

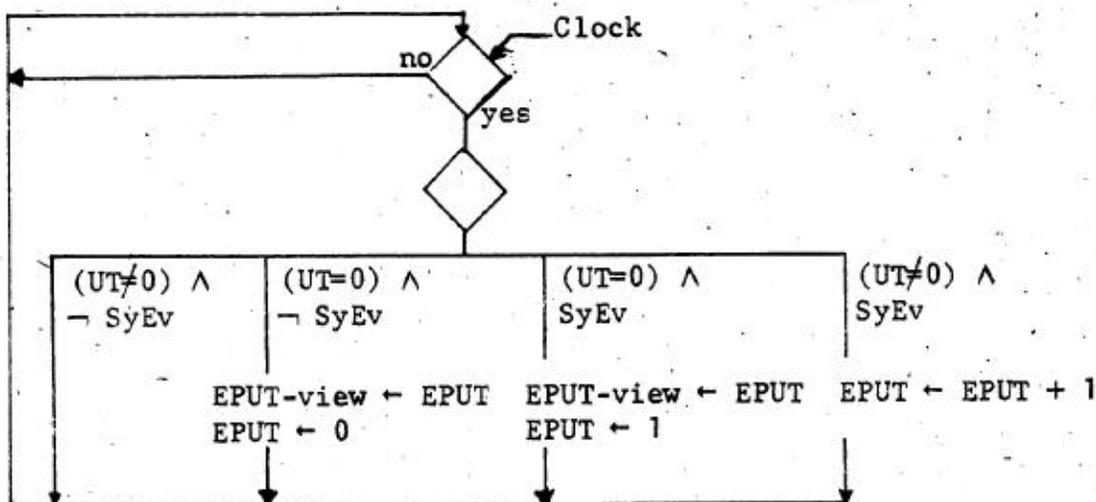
Clock

e. Switching circuit (conventional)

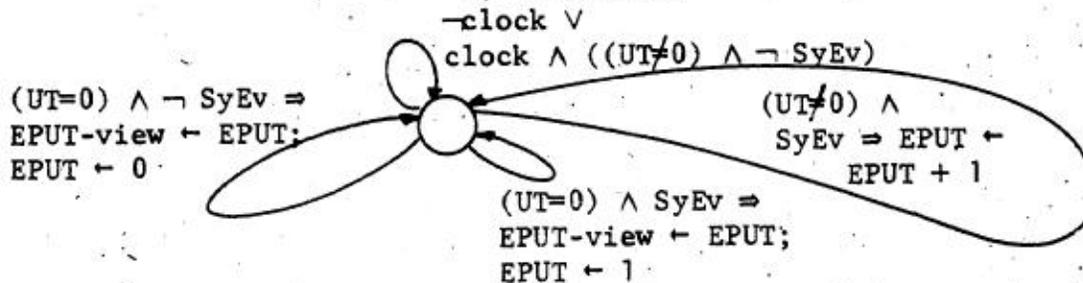
Fig. EPUTsc-4. Behavior and structure diagrams for K(synchronizer).

394

[previous](#) | [contents](#) | [next](#)



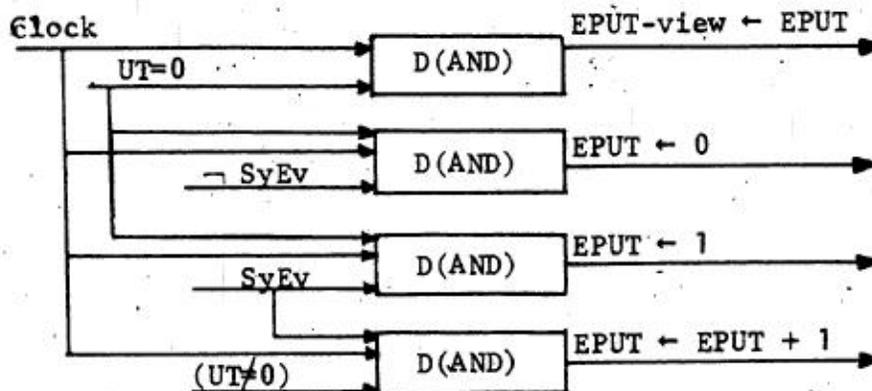
a. Flowchart



b. State diagram

$$\begin{aligned}
 clock \Rightarrow & ((UT=0) \Rightarrow (EPUT\text{-view} \leftarrow EPUT; \\
 & \neg SyEv \Rightarrow EPUT \leftarrow 0; \\
 & SyEv \Rightarrow EPUT \leftarrow 1)); \\
 (UT \neq 0) \wedge SyEv \Rightarrow & (EPUT \leftarrow EPUT + 1)
 \end{aligned}$$

c. ISP description

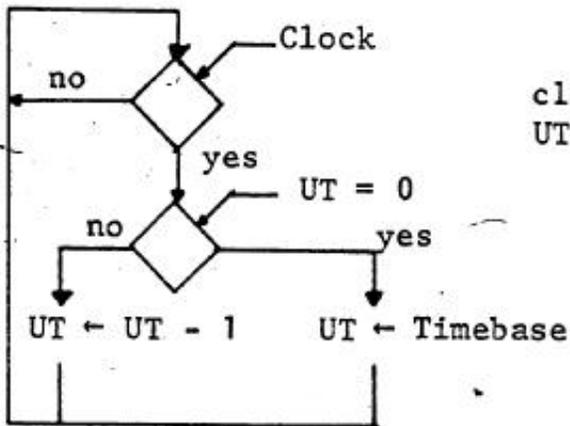


d. Switching Circuit

**Fig. EPUTsc-3. Behavior and structure diagrams for K(Events-Per-Unit-Time part).**

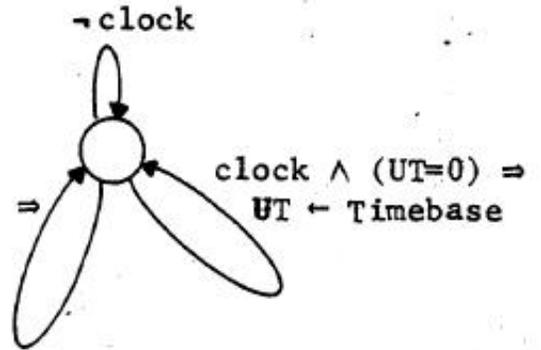
393

[previous](#) | [contents](#) | [next](#)



a. flowchart

$$\text{clock} \wedge (\text{UT} \neq 0) \Rightarrow \text{UT} \leftarrow \text{UT} - 1$$



b. state diagram

```
clock = ((UT=0) => (UT ← Timebase);
         (UT≠0) => (UT ← UT - 1))
```

c. ISP description

[previous](#) | [contents](#) | [next](#)

391

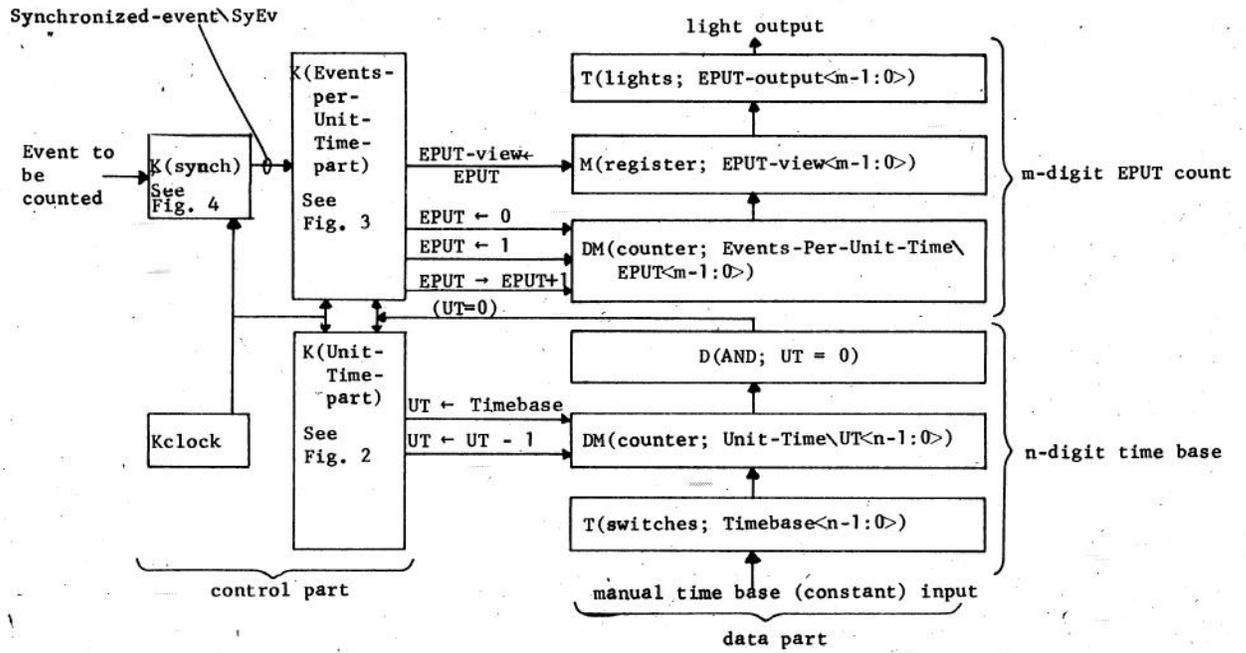


Fig. EPUTsc-1. PMS diagram of EPUT meter using switching circuits.

## SOLUTION

Figure EPUTsc-1 shows the PMS structure of the EPUT meter. The various parts are:

1. T(switches), called Timebase, holds  $n$  digits (of unspecified radix). These determine the length of time, in terms of clock pulses, the timebase, UT, is to run. This requirement is derived directly from the problem statement.
2. An  $n$ -digit DM(counter; Unit-Time\UT) is used to count the clock events, and derive the timebase. In this scheme, UT requires operations for being set to the Timebase switches (i.e.,  $UT \leftarrow \text{Timebase}$ ), and also UT must be counted down (i.e.,  $UT \leftarrow UT-1$ ).
3. The  $D(UT=0)$  network determines when  $UT=0$ , hence the completion of the time base is known.
4. For the EPUT counting section, an in-digit DM(counter;  $EPUT\langle m-1:0 \rangle$ ) holds the number of events occurring during the timebase.
5. M(register; EPUT-view) is an  $m$ -digit register that holds the number of counts which occurred during the previous timebase for display.
6. T(lights) show the output of EPUT-view.
7. K(Unit-Time. part) controls the two operations for the UT counter. It takes a clock input, and has only two outputs.
8. K(Events-Per-Unit-Time) part controls counting of EPUT and the viewing of EPUT results. Thus there are three operation outputs, and an input in synchronization with the clock which specifies when EPUT is to be counted (i.e.,  $EPUT \leftarrow EPUT+1$ ).
9. K(synchronization) takes events that occur at random with respect to the clock and synchronizes them with respect to the clock. In this way EPUT and UT counting are synchronized with respect to one another.

The behavior of the system is expressed as the behavior of three coupled control parts. Figures EPUTsc-2, 3, and 4 describe these control parts for Unit Time, EPUT, and Event synchronization. K(UT), the simplest, is shown in Figure EPUTsc-2. It has only one state, so there is no memory required to control its behavior. At each clock time, either UT is decremented (i.e.,  $UT \leftarrow UT-1$ ) or UT is reset to a new value (i.e.,  $UT \leftarrow \text{Timebase}$ ), depending on whether  $(UT=0)$  is true or not. This requirement comes by

noting that the Synchronized-event output must have a one clock time duration, otherwise it may not be used properly as an input in the EPUT circuit. With only two states the output would be on an indeterminate time. (Only long enough to be reset by the next clock event.)

Figure EPUTsc-3 describes the behavior and structure for the K(EPUT) part. This K control has only a single state which takes synchronized event inputs (SyEv) and then increments the EPUT counter accordingly. This control also transfers data to the EPUT-view register and resets EPUT.

Figure EPUTsc-4 is the most complex circuit, having 3 states, and is used to synchronize input events with the clock such that an output, Synchronized-event, is only present at the clock interval. Note that various cases of the circuit's behavior are given in the timing diagram of Figure EPUTsc-5. This synchronizer solves the same problem that occurs in the Punch problem and K(arbiter) of Chapter 5.

## ADDITIONAL PROBLEMS

1. What is the maximum event rate which the EPUT meter can accept in terms of the clock?
2. Design a meter which will count at twice this rate and still operate properly.

- 5. K controls the sequential register transfer process of the data part.
- 6. Register initialization is required for  $P \leftarrow 0$ ,  $MPD \leftarrow \text{Input-1}$  and  $MPR \leftarrow \text{Input-2}$ .
- 7. Register transfer operations are implied by the algorithm.

The design for the behavior of the control part of the system, which expresses the algorithm, is done in parallel with the PMS structure design.

Figure MPY-2 presents diagrams for the behavior of both the multiplier and the control part. The behavior is expressed in three forms:

- 1. The familiar flowchart, modified to reflect the possible structures which can be formed from RT switching circuits.
- 2. The state diagram, which indicates the number of states, and hence memory bits, required in the control. The state diagram is the conventional representation that permits a transformation to hardware.
- 3. The ISP description, which gives a linear (i.e., one dimensional) expression of the control's behavior. In effect, it expresses the conditions for evoking various register transfer operations. This description does not always permit a direct translation to hardware, although it can reflect the structure of hardware which is already designed.

The implementation of the control follows directly from either the flowchart or the state diagram. One flip flop is needed to hold the required two states of the control's state diagram. The circuit is activated when the entry signal is a one. Activation causes several initialization register transfers to occur. At each clock time, while there are still 1 digits in the multiplier, a multiplication step takes place in which either  $P \leftarrow P \times 2$  or  $P \leftarrow (P + MPD) \times 2$ , depending on whether  $MPR \langle 0 \rangle$  is 0 or 1. At the completion of the multiplication the exit signal is 1.

The performance of the multiplier is substantially greater than that of the RTM systems designed in Chapter 4 (see Figure 32). Assuming a clock time of 200 nanoseconds, the average multiplication time is  $200 * (1 + 7)$  or 1600 nanoseconds assuming 8-bit numbers. The cost of such a dedicated unit would no doubt be approximately the same as that of a DMgpa.

## ADDITIONAL PROBLEMS

1. Modify the design to tabulate various statistics about the way the multiplier is used, and the numbers

which are given it.

2. Design a multiplier for 2's complement signed numbers.
3. Design a multiplier which first interchanges the multiplier and multiplicand, depending on their magnitudes, in order to increase the multiplication speed.
4. Design a multiplier based on the 8-step process given in Chapter 4.
5. Carry out the design of a special multiplier that would be an RTM.

## EPUT METER

### PROBLEM STATEMENT

Design an EPUT meter of the type given in Chapter 5 using conventional sequential circuit components. Assume a clocked sequential design in which a high speed clock is used to supply the timebase. Assume the events to be counted arrive at random times with respect to the clock and that the event rate is about 1/4 that of the clock. The input events can be used directly to cause a register transfer operation. The time base is specified by a switch input and the EPUT output is to be placed in a separate output register and held there until another time base period is counted.

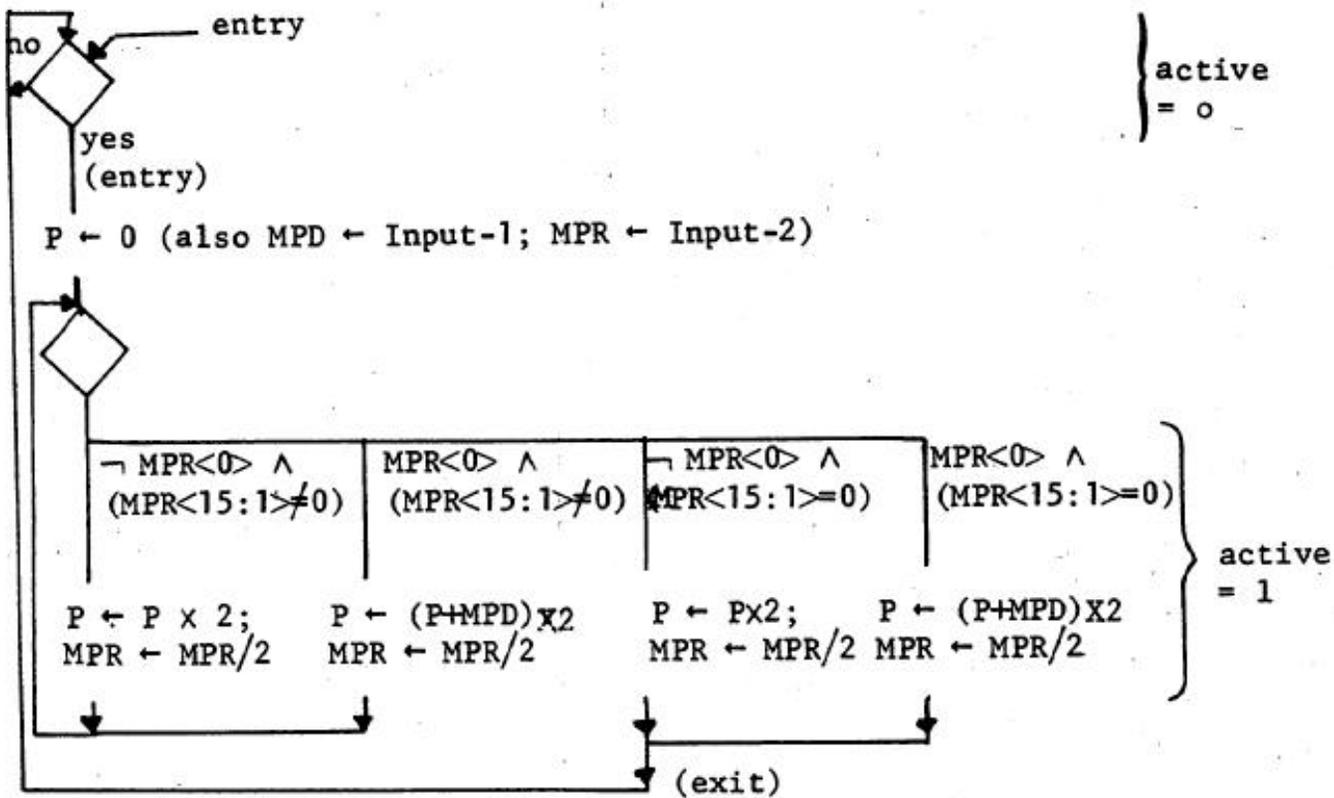
```

clock => (entry => (
    P ← 0;
    MPD ← Input - 1;
    MPR ← Input - 2;
    active ← 1);
    active => (
        ¬ MPR < 0 > => P ← P x 2
        MPR < 0 > => P ← (P + MPD) x 2
    exit (active ← 0)
    (MPR 15:1 = 0) ∧ active
exit :=

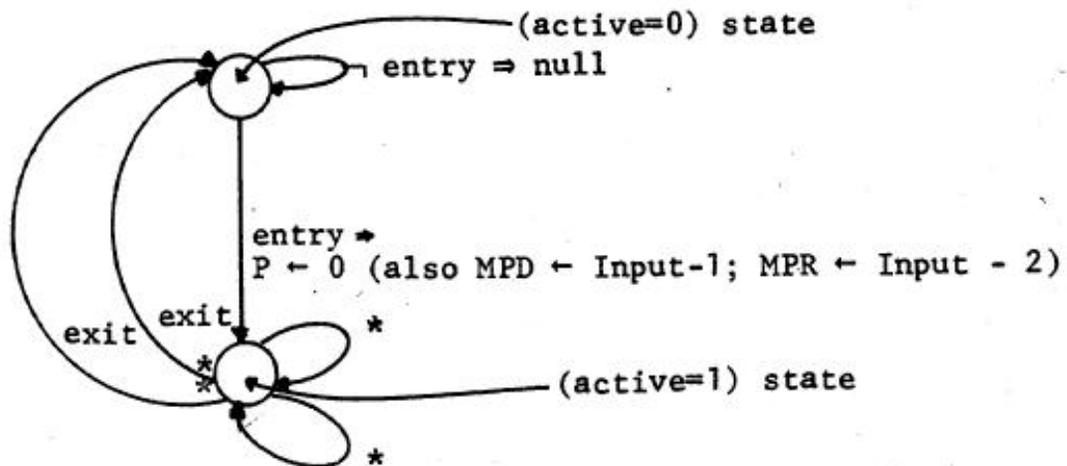
```

c. ISP description

[previous](#) | [contents](#) | [next](#)



a. Flowchart

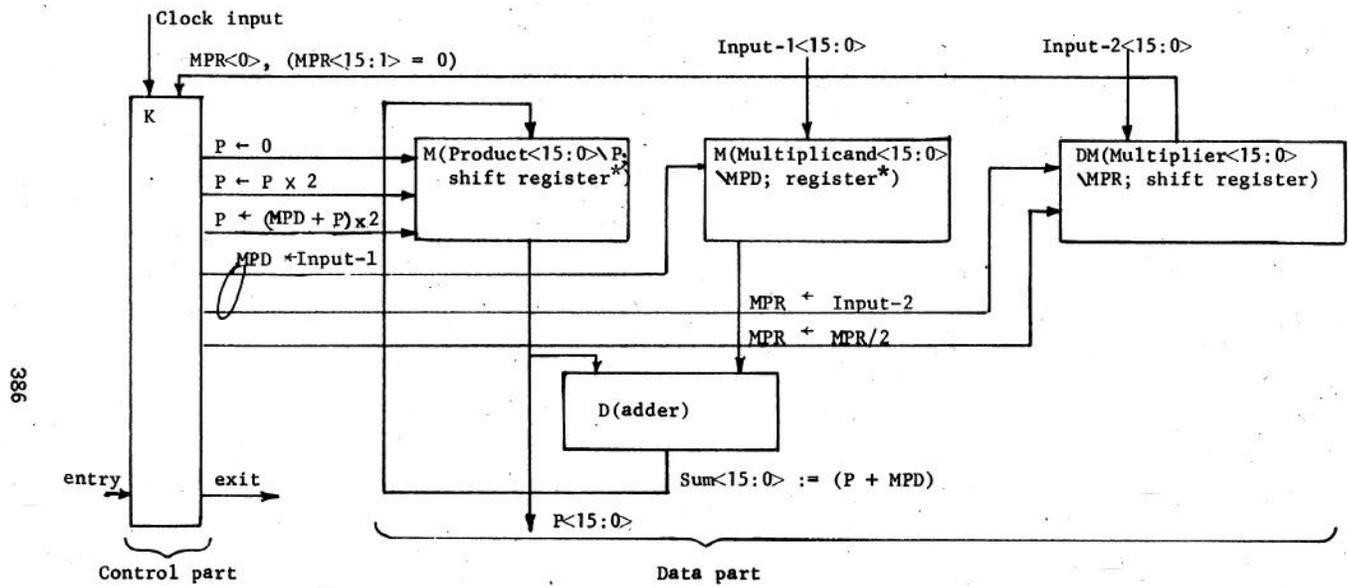


\* see flowchart

b. State diagram

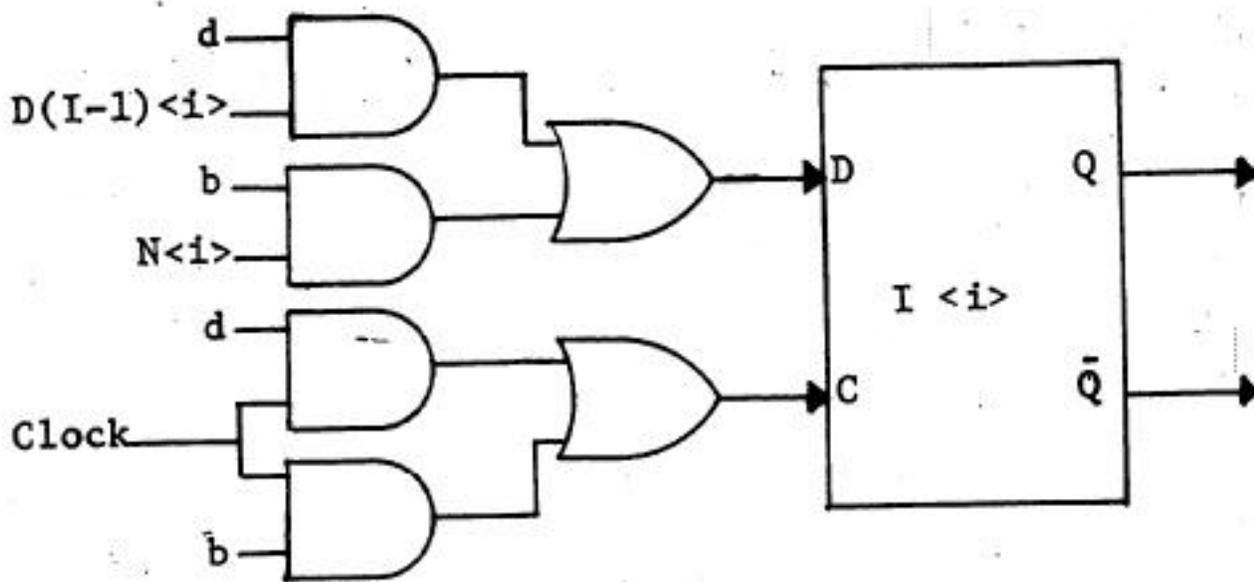
Fig. MPY-2 (Part 1 of 2).

[previous](#) | [contents](#) | [next](#)



\* e.g., R-S or J-K flip-flops.

Fig. MPY-1. PMS diagram of 16-bit multiplier using switching circuits (Russian Peasant's algorithm).



- Fig. SPI-11. Switching circuit for  $i$ th bit of  $I$  register, showing input data selection and clock control.

- The controller in the above problem does not generate a done (completion) signal. Modify the design so that it does.
- Try some other state assignments for the controller described above and compare the number of gates required for their implementation with the circuit of Figure SPI-10.
- The state diagram edges (transitions) roughly correspond to the boxes (actions) in the flowchart, whereas the flowchart edges (lines) correspond to the states of the state diagram. Show a flowchart form for Figure SPI-4.
- A memory could encode the next state, the output register transfer actions, and any inputs to sense, as was done in  $K(PCS)$ . Design a  $K(PCS)$ -like control part for doing this and subsequent control problems. Can  $K(PCS)$  do it directly? - with slight modification?

## A 16-BIT MULTIPLIER USING THE RUSSIAN PEASANT'S ALGORITHM

### PROBLEM STATEMENT

Design a 16-bit multiplier that implements the Russian Peasant's algorithm using conventional switching

circuit components. The algorithm and examples of its implementation are presented in Chapter 4. The product should be less than or equal to 16 bits, and the multiplier or multiplicand can be up to 16 bits.

## SOLUTION

Figure MPY-1 presents a PMS diagram of a multiplier. The structure of the design' is determined by the following:

- 1. M(registers) is based on lengths needed in the algorithm; these are derived from the problem constraints.
- 2. D(adder) is based on the requirement to form  $\text{Sum}_{\langle 15:0 \rangle} := (P + MPD)$ .
- 3. DM(Multiplier; shift-register) is based on the requirement for  $\text{MPR} \leftarrow \text{MPR}/2$ .
- 4. Booleans  $\text{MPR}_{\langle 0 \rangle}$  and  $(\text{MPR}_{\langle 15:1 \rangle} = 0)$  are needed to control the algorithm; that is,  $\text{MPR}_{\langle 0 \rangle}$  determines whether to use the D(adder) output or not, and  $(\text{MPR}_{\langle 15:1 \rangle} = 0)$  determines the completion of the iterative multiplication process.

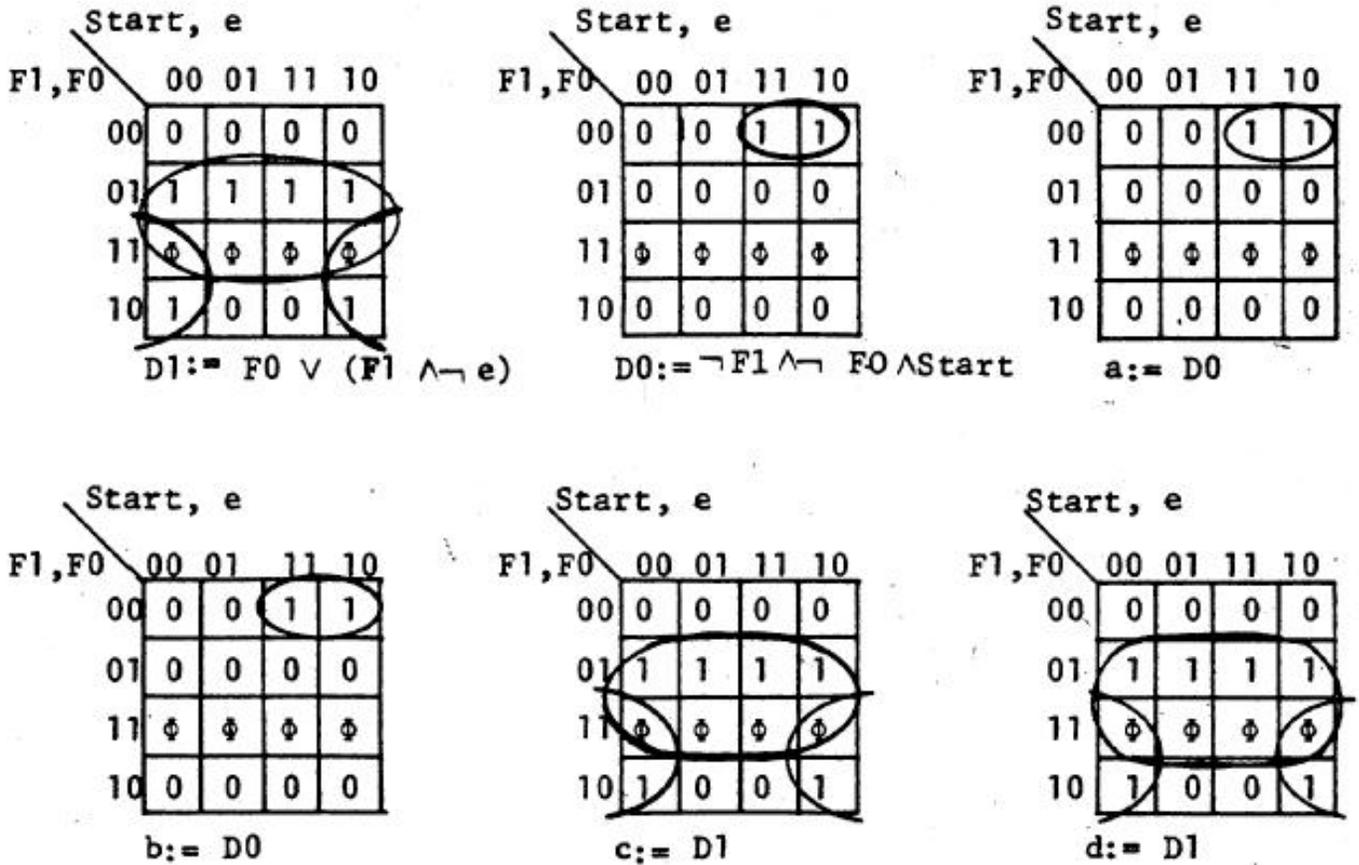


Fig. SPI-9. Karnaugh maps for the combinational circuit in the sum of integers controller (excitation and output).

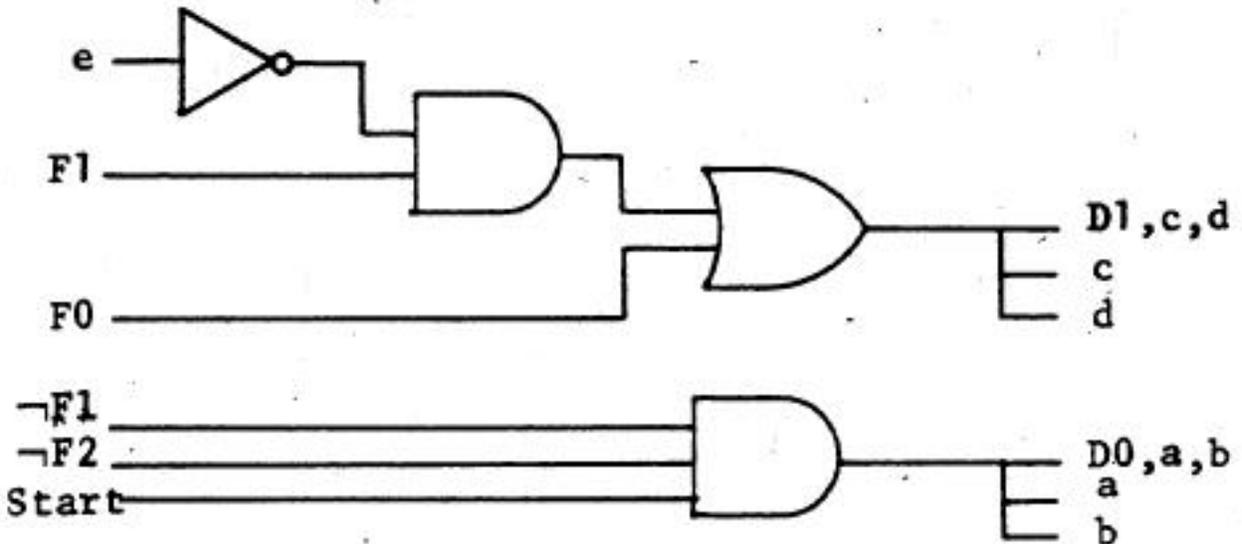


Fig. SPI-10. Switching circuit for combinational part of sum of integers controller.

384

[previous](#) | [contents](#) | [next](#)

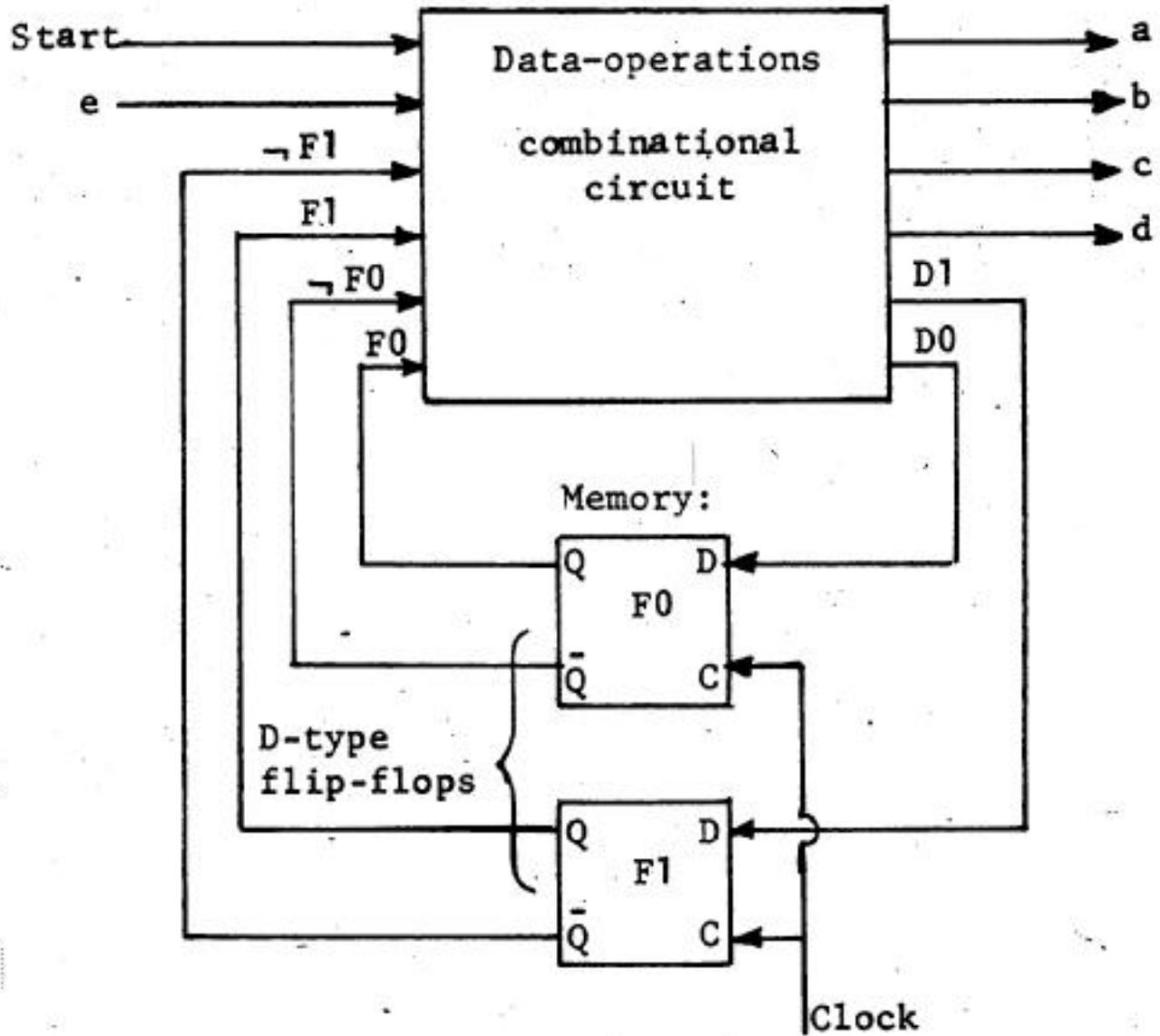


Fig. SPI-7. General configuration for sum of integers controller.

current state F1,F0	inputs			
	00	01	11	10
00	00, 0000	00, 0000	01, 1100	01, 1100
01	10, 0011	10, 0011	10, 0011	10, 0011
10	10, 0011	00, 0000	00, 0000	10, 0011

Fig. SPI-8. Transition table for sum of integers controller.

current state	inputs			
	Start ⚡ (I=0) 00	01	11	10
q <sub>0</sub>	q <sub>0</sub> , 0000	q <sub>0</sub> , 0000	q <sub>1</sub> , 1100	q <sub>1</sub> , 1100
q <sub>1</sub>	q <sub>2</sub> , 0011	q <sub>2</sub> , 0011	q <sub>2</sub> , 0011	q <sub>2</sub> , 0011
q <sub>2</sub>	q <sub>3</sub> , 0011	q <sub>0</sub> , 0000	q <sub>0</sub> , 0000	q <sub>3</sub> , 0011
q <sub>3</sub>	q <sub>2</sub> , 0011	q <sub>0</sub> , 0000	q <sub>0</sub> , 0000	q <sub>2</sub> , 0011

next state,  
outputs

Fig. SPI-5. Non-minimal state table for sum of integers controller.

current state	inputs			
	00	01	11	10
q <sub>0</sub>	q <sub>0</sub> , 0000	q <sub>0</sub> , 0000	q <sub>1</sub> , 1100	q <sub>1</sub> , 1100
q <sub>1</sub>	q <sub>2</sub> , 0011			
q <sub>2</sub>	q <sub>2</sub> , 0011	q <sub>0</sub> , 0000	q <sub>0</sub> , 0000	q <sub>2</sub> , 0011

next state,  
outputs

Fig. SPI-6. Minimal state table for sum of integers controller.

The next task is to realize this state table with hardware, using the switching circuit model shown in Figure SPI-7. Two flip flops are used in the model, because three states of the controller have to be encoded. The job of encoding the three states is called the state assignment task. It should be clear that, depending on which combinations of 1's and 0's in the two flip flops correspond to which states, different complexities of the combinational circuit in the model may result. For illustrative purposes, we arbitrarily choose a state assignment as shown in the state transition table in Figure SRI-8.

The task remains to design that combinational circuit. This can be done simply by constructing separate Karnaugh maps (see Hill and Peterson, 1968) for each of its outputs, and then selecting minimal Boolean equations for them, as shown in Figure SPI-9. Finally, the Boolean equations are translated into

switching circuit realizations, as shown in Fig. SPI-10.

The manner in which the outputs of the controller evoke the register transfers is shown in Figure SPI-11, where the steering logic for the  $i$ th bit of the I register is shown.

## ADDITIONAL PROBLEMS

1. Do the switching circuit design of the D's in the above problem.
2. A "full adder" is a combinational circuit that takes an addend bit, an augend bit, and a carry bit as inputs and produces a sum bit and a carry bit as outputs (assuming binary addition). Do the switching circuit design of the D's in the above problem using only full adders, wires, and Boolean constants (1's and 0's).

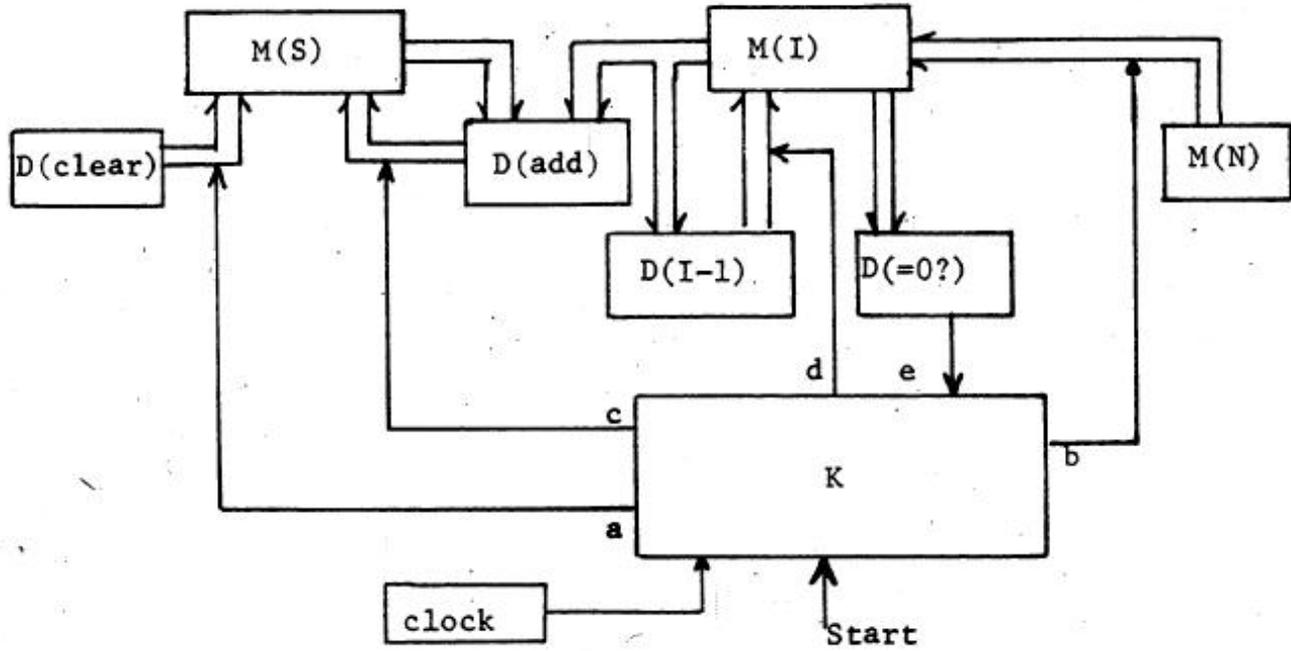


Fig. SPI-3. Register transfer level diagram of sum of positive integers to N.

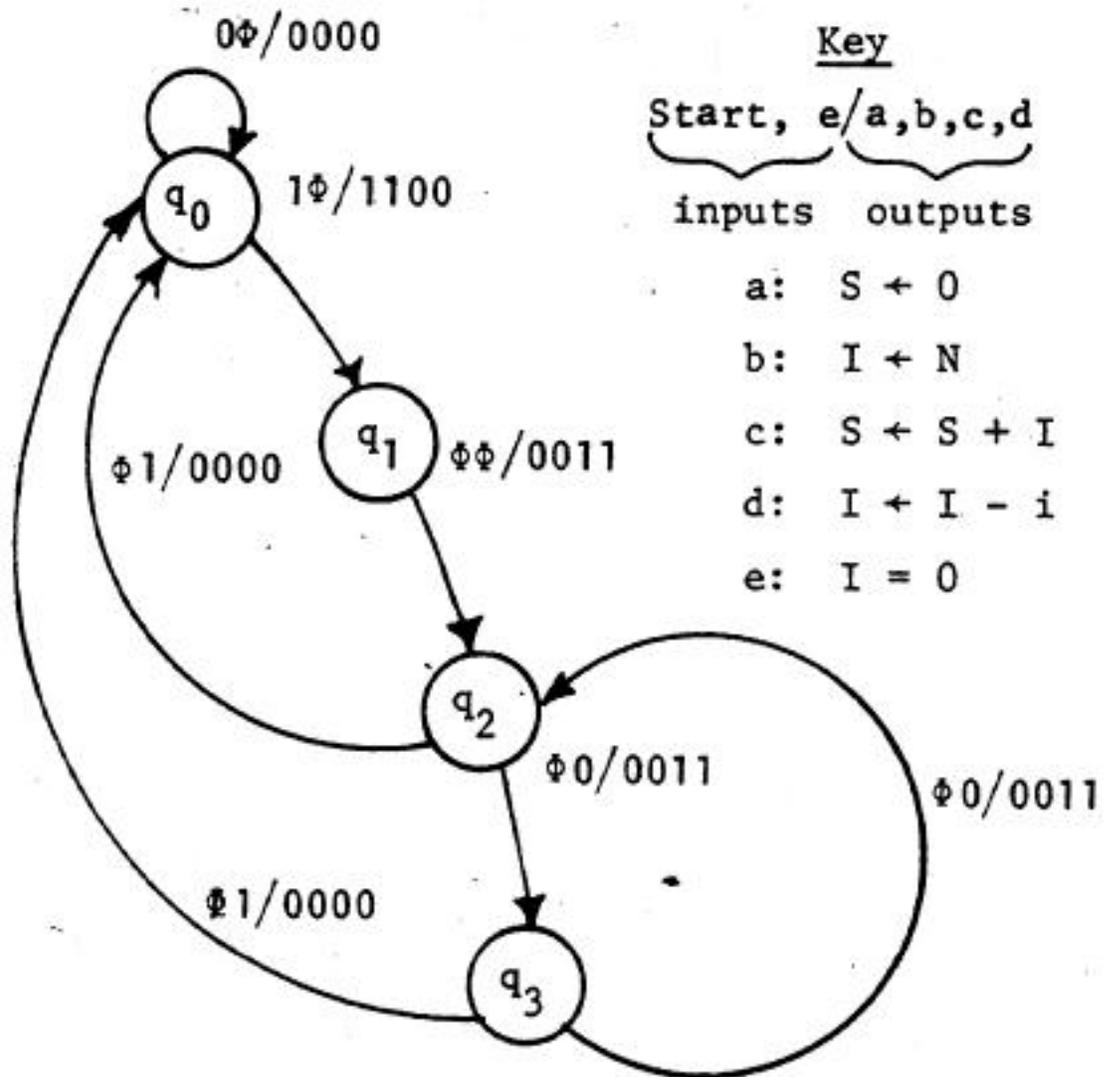


Fig. SPI-4. Non-minimal state diagram for sum of integers controller.

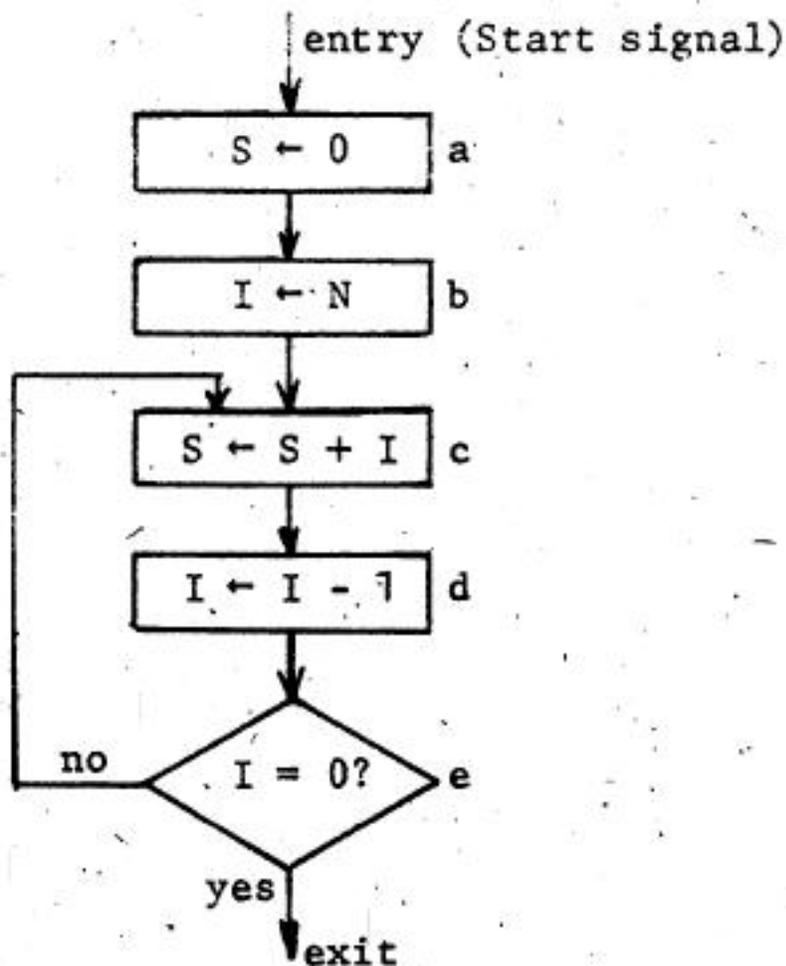


Fig. SPI-1. Flowchart for sum of positive integers to N.

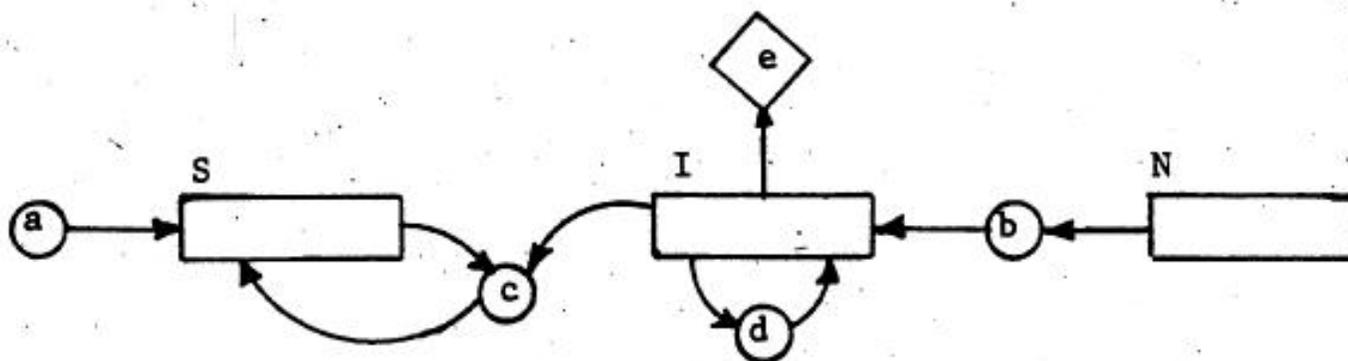


Fig. SPI-2. Data flow graph for sum of positive integers to N.

The next step in the design of the controller is to translate the state diagram into tabular form, as shown in Figure SPI-5. At this point in the design process, state minimization procedures can be applied to the state table (see Hill and Peterson, 1968 for details). In this case the state minimization procedures will

note that states  $q_2$  and  $q_3$  have identical sets of outputs in the table. Furthermore, if their next states are interchanged, the behavior of the system will be the same. Thus, states  $q_2$  and  $q_3$  are equivalent, and they are replaced by the single state  $q_2$ , as shown in the state table of Figure SPI-6. This can be proved to be the realization with the minimal number of states for this state table.

integrated logic circuits. Thus, the control scheme methodology used in this book can be applied directly to control parts as an alternative to the conventional state diagram approach.

## SUM OF POSITIVE INTEGERS TO N

### PROBLEM STATEMENT

Design a system to sum the positive integers to N, as described in Chapter 2, using conventional switching circuit components. Assume a

clocked sequential design. Further assume that the integer, N, is initially stored in a register (the N register) and that the result is placed in another register, the S register. Finally, assume that a Start signal, which is present for one clock period, initiates the operation of the system.

### SOLUTION

The flowchart for the algorithm is shown in Figure SPI-1, with each operation labelled by a lower case letter of the alphabet, for later reference. Instead of plunging ahead and postulating a register transfer level diagram for a system to implement this algorithm, it is instructive -to analyze first what data storage and transfer mechanisms will be needed. One way of doing this is to construct a data flow graph (Dennis and Patil, 1970), as shown in Figure SPI-2. In the graph the rectangles represent Memory registers, one for each of three variables in the algorithm; the circles and diamonds represent the data operations performed on the data (see the letter designations in Figure SPI-1); and the arrows represent data flow paths (links).

For this problem, the data flow graph provides a plausible model for the register transfer level system diagram. This is shown in Figure SPI-3, with a controller, K, added to properly sequence the operations. In the diagram, PMS notation is used to identify the various component types. Data paths are shown by double lines, and control signals are drawn as single lines and are labelled with the letters corresponding to the operation they evoke (see Figure SPI-1). It is assumed that the system will be built using D-type edge triggered flip flops. Thus the control lines are shown connected to the transfer paths, where they will effectively be used to strobe the appropriate data into the registers. The D's are assumed to be combinational circuits that are operating continuously, and need not be evoked.

The controller for the system will be designed using standard clocked sequential design techniques, as described, for example, in Hill and Peterson (1968), Kohavi (1971) and Peatman (1972). The first step in the design of the K is to translate the algorithm flowchart into a state diagram, as shown in Figure SPI-4. In the state diagram, each node corresponds to a state of the controller (a state is determined by the memory encoding in the controller part) and each graph edge (or arc or line) designates a transition from one state to another. All transitions are evoked by the occurrence of a clock pulse. The edges are labelled

with the value of the inputs to the controller (to the left of the slash) at the time of the transmission, and the outputs of the controller (to the right of the slash) at the time of the transition. Because trailing "waveform edge" triggered flip flops are used, the outputs do not change value until after the clock pulse that evoked the state transition has disappeared.

The reader may note that this state diagram is a bit sloppy. State q3 is really unnecessary, as the ~0/0011 transition from state q2 could have been directed back to state q2, giving a system with identical performance. We leave the state diagram this way to illustrate a later step in the design procedure.

## ■ CHAPTER 8 RT LEVEL DESIGN

### WITH CONVENTIONAL SWITCHING CIRCUIT COMPONENTS

Throughout this book we have illustrated register transfer level digital design using RTM's as a standard set of building blocks. We did this partly for uniformity and partly because the RTM notation lends itself to quickly understandable system diagrams. However, the concepts of register transfer level design are for the most part applicable for any well-chosen set of building blocks. We illustrate this by presenting here three of the designs that were done using RTM's earlier in the book, and implementing them using ad hoc techniques with conventional 'switching circuit components. We make no claim that all types of register transfer level design are represented in these problems. One was done by one of the authors and the other two by another of the authors, so only two of the many possible alternative register transfer level design styles are illustrated.

We are attempting to show that the principles illustrated in this book should be of use in all system designs, regardless of the set of primitives used. However, the choice of the set of primitives can have a considerable affect on the ease of design. Since no standard design approach or set of register transfer level, components is used in this chapter, design decisions are based on the characteristics of the problem at hand. Thus, the reader will see varying sizes and types of building blocks, varying notations, and varying synthesis approaches in the three problems. (The building blocks are about the same size as those currently available from integrated circuit manufacturers.) This flexibility allows the designer to optimize solutions locally in a way that is not possible with fixed sets of modules, such as RTM's. On the other hand, the price paid for this flexibility is the loss of standardization, with all of its attendant advantages.

### APPLICATIONS OF THE RTM CONTROL STRUCTURE TO CONVENTIONAL RT LEVEL DESIGN

It should be noted that while we have warned the reader that the methods of control used in this book are highly specific to RTM's, the RTM control structure also is easily used with conventional switching circuit design. In Chapter 7, where the internal operations of the various modules are described, it can be observed that certain common switching circuit components are used to implement each of the control operations. That is: Kevoke consists of a two state device utilizing two flip-flops; Kbranch requires a flip flop to sample the incoming Boolean variable, and two AND gates to route the output control signals; Ksubroutine is actually just a flip-flop to mark the state of control temporarily for one of several possible control return points; K(serial-merge) and K(parallel- merge) are just the negative logic OR and AND gates, respectively.

In introductory logical design texts on sequential circuit design, assumptions can often be made to simplify the control part, because only clocked, synchronous logic is used. On the other hand RTM's operate asynchronously, in that a single operation on the Bus can take an arbitrary amount of time. This

requirement adds to the control complexity. If one were to relax the requirements to only permit synchronous clocked logic, then it would be possible to use a single flip-flop for Kevoke, and no flip-flop would be required for the Kbranch, thus simplifying the modules.

With this view of the control elements, problems 3 and 4 of Chapter 7 can be considered using the control scheme of this book together with available

It is interesting to speculate on how the DMgpa might be redesigned subject to design goals other than those which influenced the PDP-16 set of modules. This topic is beyond the scope of this book, but the reader is invited to try the DMgpa design problem given in the problem section.

## PROBLEMS

1. Assume that the DMgpa. to be implemented is on a single large scale integrated circuit chip. Do an RT level design, such as the one shown in Figure 22, for a DMgpa. The PDP-16 DMgpa is composed of about 500 gate equivalents, but current LSI technology is capable of getting several thousand gate equivalents on a chip. There will be a pin limitations problem, since current LSI technology is limited to about 50 pins per chip. The two double height printed circuit boards which hold the current DMgpa have about 144 pins for input and output.

Thus, on the one hand, LSI provides greatly increased internal complexity. (For example, one might implement more than two registers, built-in register transfers (for Bus-exclusive operations), hardwired multiply, other data operations, etc.) But, on the other hand, the pin limitations change the use of multiple evoke-operations, parallel data transfers, Boolean outputs, etc. Perhaps the reader would like to change the RTM system ground rules (Bus and control part) in order to achieve his design. In any case, this problem opens up further the fascinating subject of the design of RT level module sets to investigate. Note that the DMar, described at the end of Chapter 2, is one solution to this problem..

2. Verify that Ksub' operates correctly for the cases of a single operation in the subroutine, and nested subroutine calls, as shown in Figure 20.

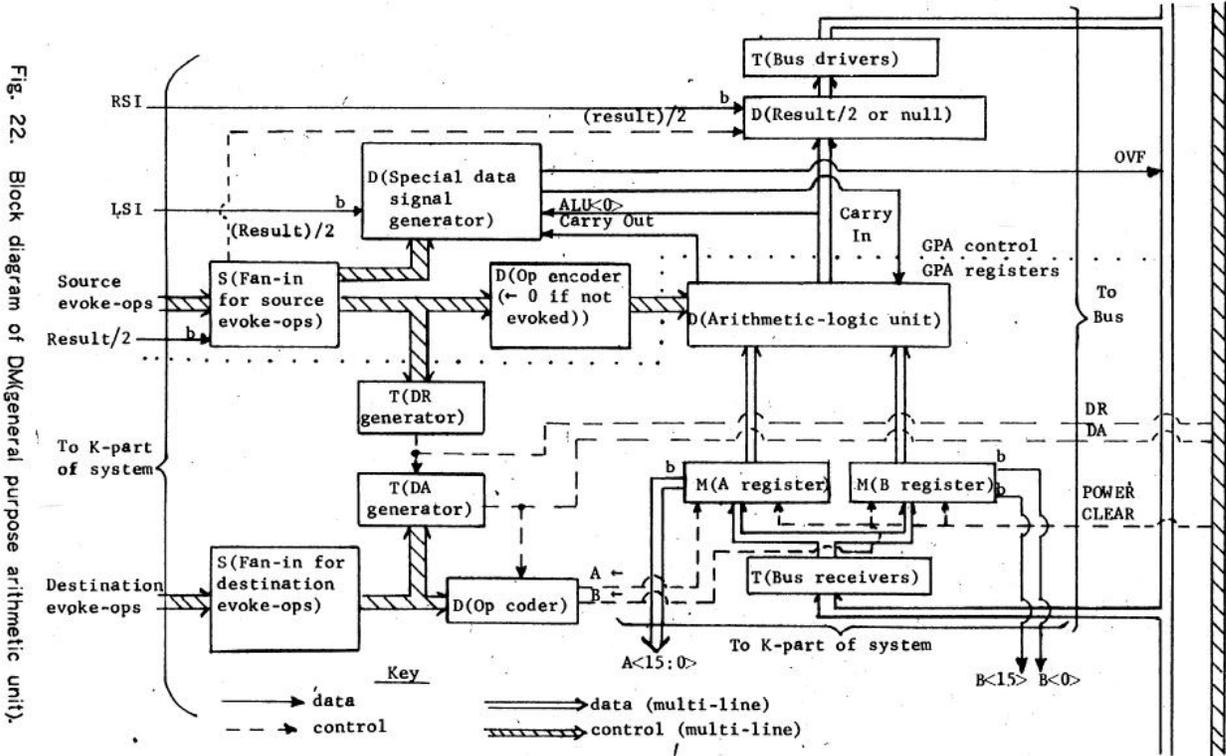
3. Take a standard integrated circuit catalog and try to make' some sense from it by expressing the functions and interconnections as register transfer operations.

4. Design an appropriate set of control modules for a particular set of integrated circuits, so that the logical design process is not the laborious process given in the usual logical design textbooks, resulting in non-modifiable designs.

5. Strictly speaking, the K(evoke) does not have three states, it has four. Find under which conditions the fourth state occurs, and illustrate it, using an RTM diagram and a timing diagram.

6. Design a fast, combinational circuit multiplier RTM.

[previous](#) | [contents](#) | [next](#)



376

Fig. 22. Block diagram of DMK general purpose arithmetic unit.

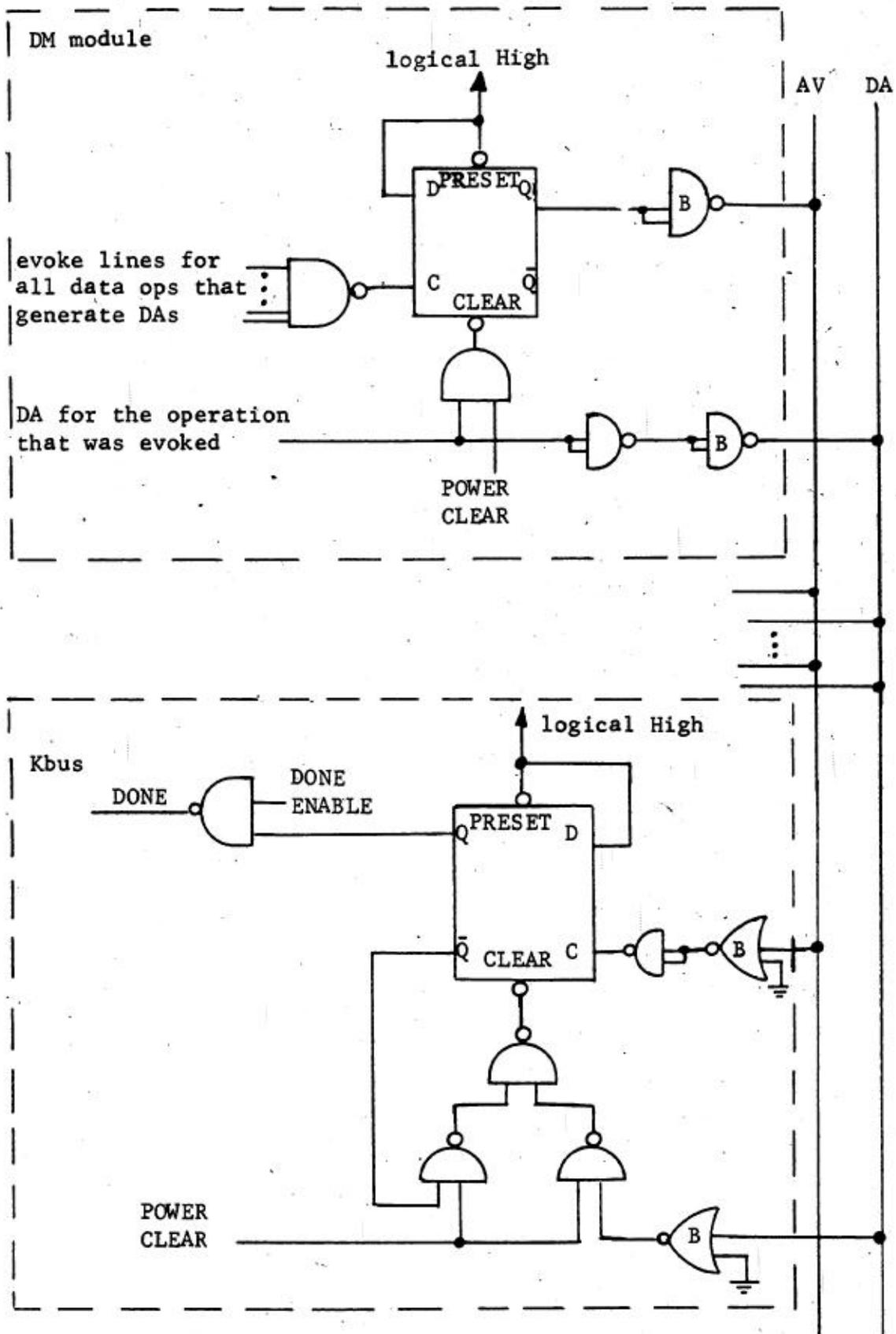


Fig. 21. Switching circuit showing a scheme for controlling parallel operations on one Bus.

on one Bus.

375

[previous](#) | [contents](#) | [next](#)

like to have a truly asynchronous Bus control scheme which waits for all DA's before proceeding. In such a scheme the DONE signal for a Bus could occur only after all the DA's of all operations being evoked had been generated.

A flexible and low cost scheme for providing multiple transfers from a single source is shown in Figure 21. Another control signal, called "Active", or "AV", has been added to the Bus. Whenever any operation of a DM module that generates a DA has been evoked, the AV bus line goes Low. (The control circuit to accomplish this for a DM module is shown at the top of Figure 21. Note the use of "Bus driver" and "Bus receiver" gates; like the other Bus lines, AV is effectively a negative logic "wired OR.") When the DA for a given module has been generated, its AV output is returned to a logical High. Because of the wired OR nature of the Bus, only when all modules have generated their DA signals does the AV Bus line go High. At this point the DONE signal is generated by the Kbus (see bottom of Figure 21). When all DA signals have been removed, the DA Bus line again goes High and resets the Kbus control circuitry, thus removing the DONE signal and completing the control cycle.

The scheme that is shown has too many gate delays in it. Since one would like to obtain the highest speed possible in an asynchronous system, the reader is invited to attempt a higher performance design to accomplish the same operational goals.

## DM(GENERAL PURPOSE ARITHMETIC UNIT)

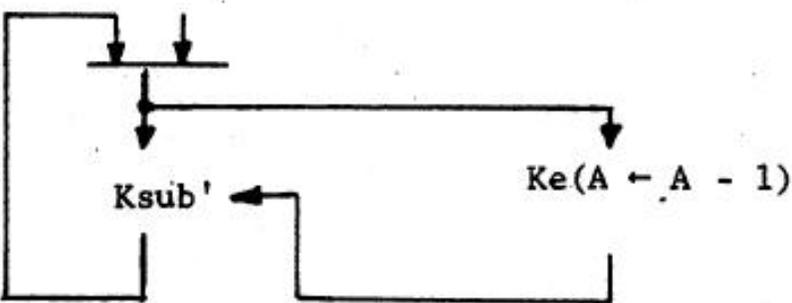
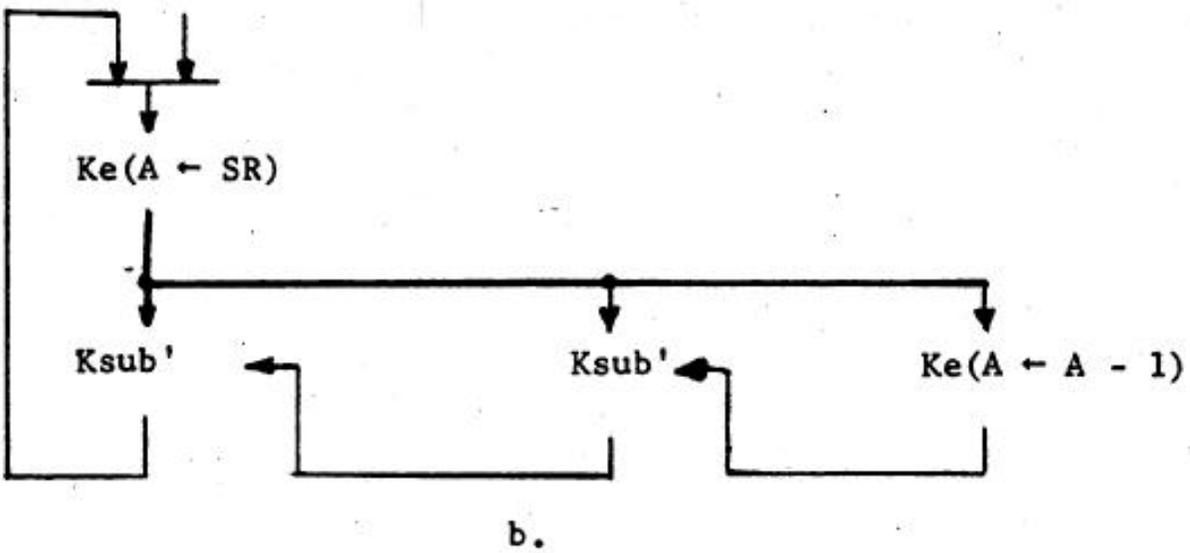
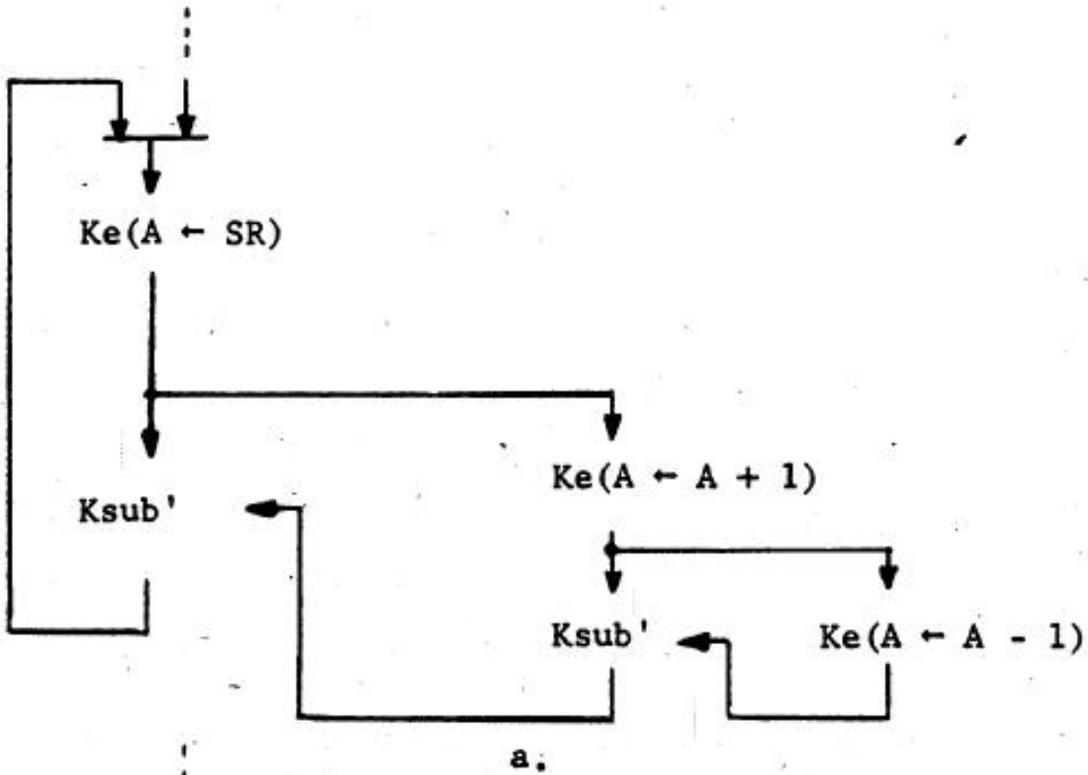
The logic level design of the DMgpa is of interest primarily because it is the most often used of the data-memory type modules. A block diagram of the DMgpa, with the PMS types of its components labeled, is given in Figure 22. The dotted line on the figure separates the logic assigned to the two printed circuit boards that constitute DEC's DMgpa - one is called the GPA Control and the other the GPA Registers.

The core of the DMgpa, the D(Arithmetic-logic unit), is implemented using two, three, or four 4-bit wide medium scale integrated circuit chips (for 8,12, or 16-bit word size), which are capable of 16 simple arithmetic (see DMar Chapter 2) and 16 Boolean logic operations on the two input registers, A and B. The operations are encoded from the evoke-operation inputs to the Dmgpa, and, because of pin limitations for the module, not all 32 possible operations are utilized or desired. The DMar does provide these.

As mentioned previously, all connections to the RTM Bus are made via negative logic wired OR connections, which require a special type of gate for interfacing to the rest of the DMgpa logic. These are shown as the T(Bus drivers) and T(Bus receivers) in Figure 22. On the input side, A and B registers only accept data from the Bus when they receive clock strobing control signals from their D(operation encoder). However, on the output side they are continuously broadcasting to the Bus, through the D(Arithmetic-logic unit) and the D(Result/2|null). Therefore when data source operations are being

evoked in a DMgpa, the operation code of the D(Arithmetic-logic unit) is locked to produce all zeros at its output, thus insuring that the Bus is free to carry data from other modules.

The D(Result/2|null) network had to be added to the DMgpa because the chips used for the D(Arithmetic-logic unit) have no right-shift operation - in fact they achieve left-shifting only by adding A to A. Thus the right shift is implemented using the Result/2 Boolean input. The T(DR generator) and T(DA generator) have already been shown earlier in Figure 7. The rest of Figure 22 is (hopefully) self-explanatory.





c.

Fig. 20. RTM system diagrams for testing Ksub'.

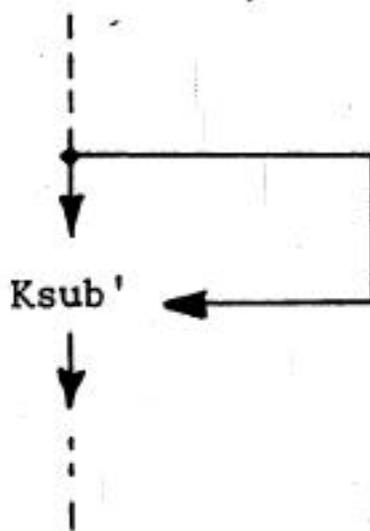


Fig. 19: RTM diagram of  $K_{sub}'$  wired to a trival subroutine.

One further problem with  $K_{sub}'$  is that if it is to be used as a  $K$ (manual evoke), flip flop  $F2$  has to have been initialized to the High output state. Since the activate input is normally held Low by the pushbutton, both inputs to  $F2$  may always have been High since the time power was turned on, in which case  $F2$  would be in an unknown state. This problem can be solved in one of two ways. First, if at least one  $K$ (evoke) had been activated prior to the use of the  $K$ (manual evoke), then a DONE signal would have been generated and  $F2$  would be set to High. Alternatively, POWER CLEAR and DONE could be connected through an OR gate to the  $S$  input of  $F2$ . This would allow  $F2$  to be initialized to Low when the circuit is used as a  $K$ (manual evoke), and to a High (by the uncomplemented output of  $F1$ ) when the circuit is used as a  $K_{sub}'$ .

## SINGLE BUS PARALLELISM

Various ways of achieving parallelism in an RTM system, using one or more Busses, were described in Chapters 2 and 4. Certain aspects of the switching circuit implementation of RTM systems affect the ease with which various types of parallelism are achieved. The single Bus parallelism case, where 'multiple destination assignments are made from a single source, e.g.,  $A \leftarrow B \leftarrow S + X$ , will be examined.

By disabling the DA output of all but one of the modules which receive the data, an operation of the preceding form is possible. For example, a NAND gate in the DA generation path would provide the facility for enabling or disabling the DA.

Currently the only PDP-16 modules (options) that have this DA-disable facility are the  $M$ (transfer

register), the DM(f lag), and the T(serial interface). However, future additions to the set of modules may result in further Bus-exclusive operations or Bus-utilizing operations that can be performed in parallel on a single-Bus system. For example, a DMgpa with an internal bus that links its registers could be used for intra-DMgpa register transfer operations. Such operations could then be evoked in parallel with other operations that required the main RTM Bus.

In either present or future cases, a potential engineering difficulty is introduced whenever the DA's are disabled, because asynchronous timing is violated. If an operation with a disabled DA takes longer than another operation which generates the DA, the former transfer may be incorrect. Thus, one would



Fig. 18. Switching circuit for  $K_{sub}'$ , a  $K_{sub}$  circuit based on the  $K(\text{evoke})$ .

371

[previous](#) | [contents](#) | [next](#)

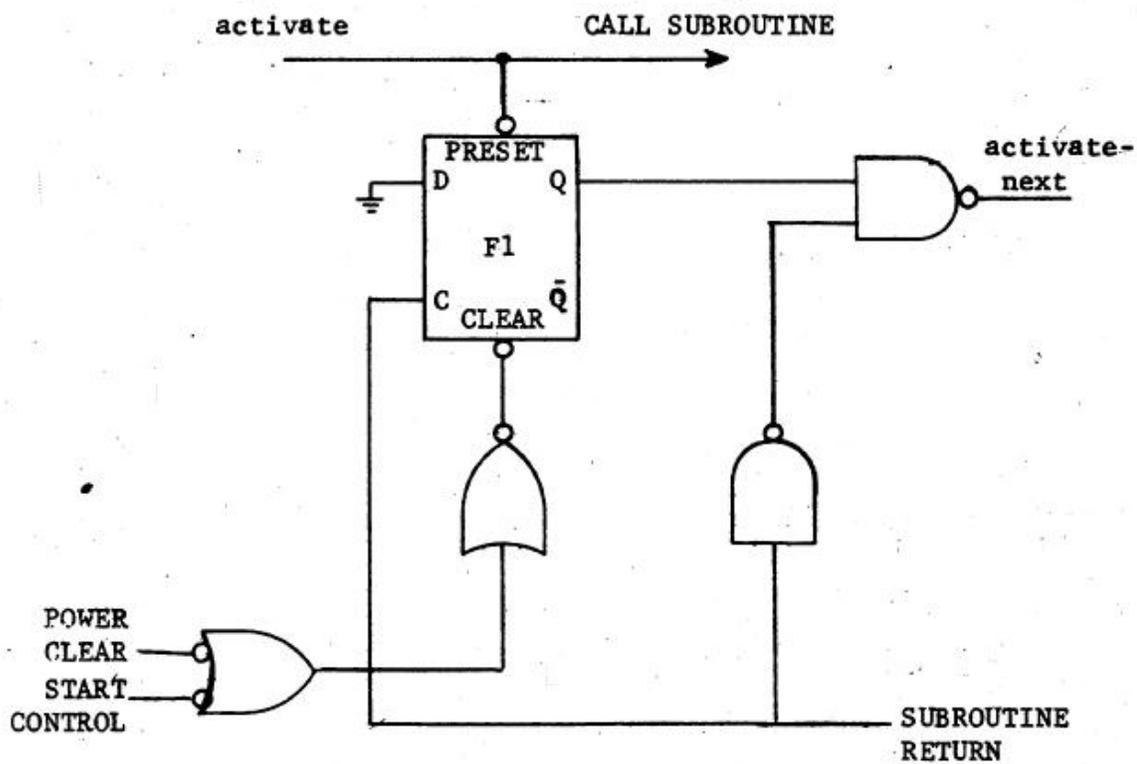


Fig. 15. Switching circuit for DEC K(subroutine call).

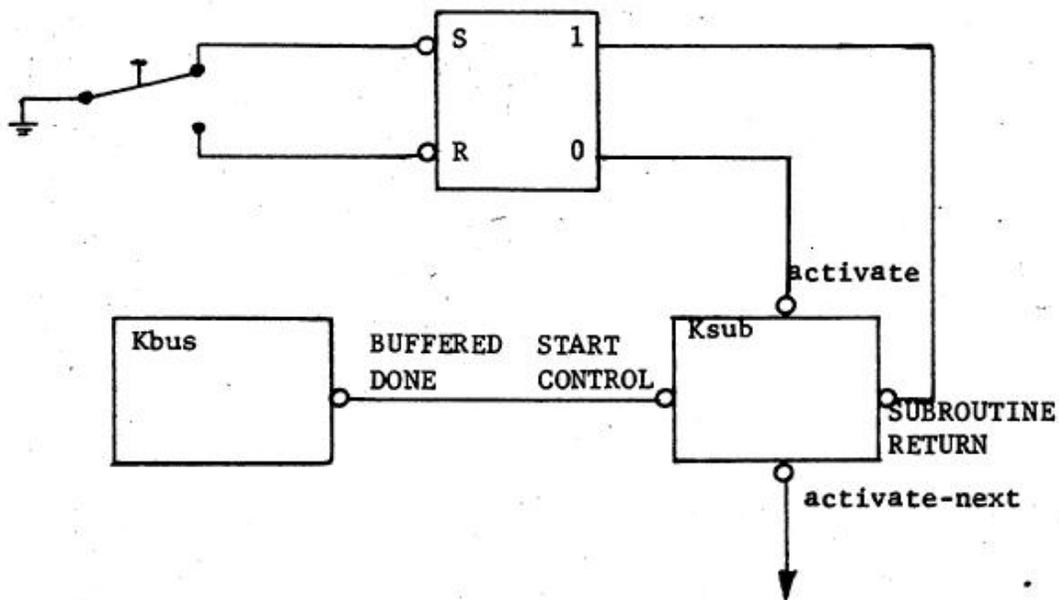


Fig. 16. RTM diagram showing use of Ksub as a K(manual evoke).

[previous](#) | [contents](#) | [next](#)

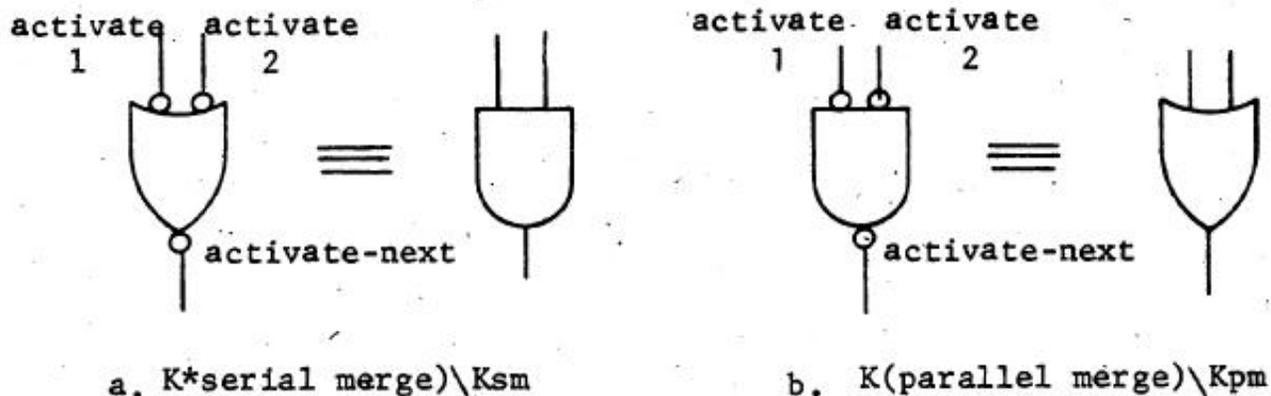


Fig. 14. Switching circuits for merge modules.

## SUBROUTINE CALL

The DEC circuit for the  $K(\text{subroutine call}) \setminus K_{sub}$  is shown in Figure 15. In essence,  $K_{sub}$  is just a flag (or Boolean variable) which remembers one of the possible callers to a subroutine. Its operation is fairly straightforward. When the module is activated, the flip flop,  $F_1$ , goes to the High state. Then, when the SUBROUTINE RETURN input is asserted by going to Low the activate-next output is asserted. After SUBROUTINE RETURN reverts to High,  $F_1$  returns to the low state and the  $K_{sub}$  is deactivated.

The PDP-16 handbook gives the diagram shown in Figure 16 for using the  $K_{sub}$  as a  $K(\text{manual evoke}) \setminus K_{me}$ . When the pushbutton is inactive, the activate input of the  $K_{sub}$  is Low, thus putting it in the active state, but with activate-next not asserted. When the button is pushed, the SUBROUTINE RETURN goes Low, thus causing activate-next to be asserted. When the data operation it evokes is finished, the BUFFERED DONE signal returns the  $K_{sub}$  to the inactive state, thus removing activate-next. When the pushbutton is released, the  $K_{sub}$  is rearmed<sub>1</sub> ready to act as a  $K(\text{manual evoke})$  the next time the button is pushed.

However, there is a problem with this circuit realization for the  $K_{sub}$ , which occurs when two calls to the same subroutine are made in series (see Figure 17). Suppose that the first of the two  $K_{sub}$ s is activated. At some time later the SUBROUTINE RETURN lines of both  $K_{sub}$ s are asserted, as the subroutine is completed. This causes the activate-next output of the first  $K_{sub}$  to be asserted, which is equivalent to the activate input for the second  $K_{sub}$ . However, the first SUBROUTINE RETURN signal has not yet been removed, and thus the activate-next output of the second  $K_{sub}$  is also asserted, incorrectly. Therefore, calls to the the same subroutine may not occur in direct succession.

The problem is resolved by interposing an event in the  $K_{sub}$  between when it is activated and when it delivers an activate-next output. A convenient event to use might be the occurrence of a DONE signal.

Thus, an alternative design for the Ksub would be to alter a Kevoke in the manner shown in Figure 18 to produce a circuit called Ksub'.

The use of the DONE signal constrains Ksub' to be used only with subroutines that contain at least one K(evoke), which is not a serious limitation. For example, Ksub' can not be used for the trivial subroutine shown in Figure 19, as it would never generate an activate-next. Figure 20 gives some other configurations in which Ksub' should be tested. A problem on this is given at the end of the chapter.

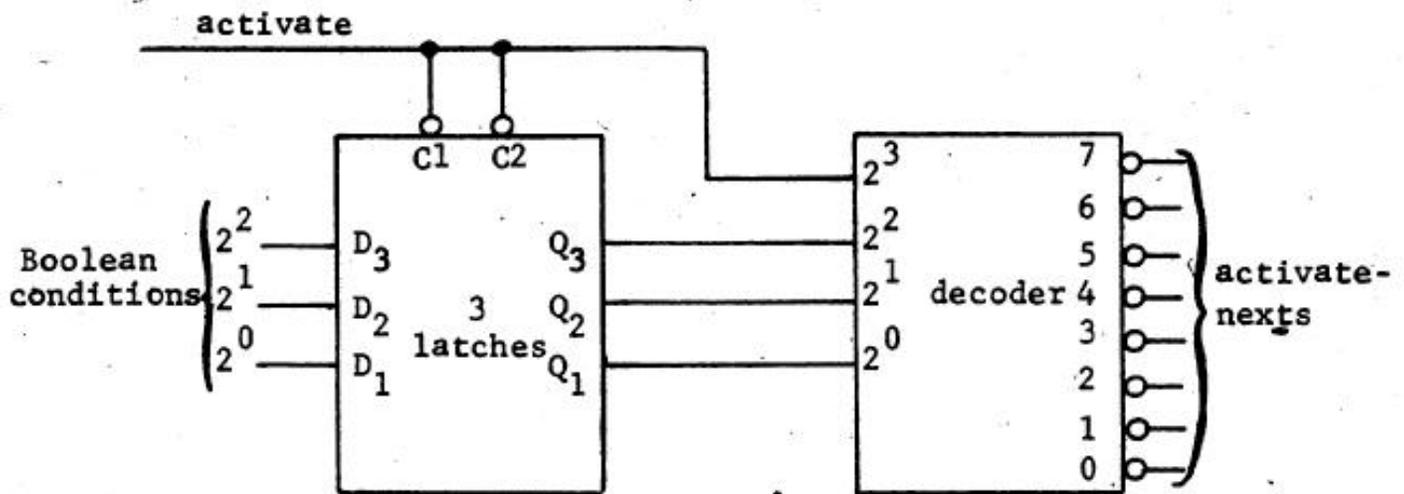


Fig. 13. Switching circuit diagram for DEC Kb8.

the flip flop F1, Kb2 cannot change its output again until a Ke is activated and the activate input to the Kb2 returns to High. Thus a Ke which evokes a dummy operation (e.g., BSR<-0) must be inserted into the loop, as shown in Figure 12b. Alternatively, a special K(No-op) module, when activated, provides a DA and an activate-next output after a delay of about 100-200 ns. This scheme is used in the Kwait Extended RTM of Chapter 3.

### Eight-Way Branch\Kb8

The Kb8 switching circuit is essentially just an extended version of the Kb2 circuit. Instead of one latch, it has three, one to hold each Boolean-condition input. The outputs of these latches go into a decoder which, when enabled, determines which of the 8 possible activate-next outputs should be asserted (see Figure 13.) The three latches and the decoder are each shown as black boxes in the figure because each is, in fact, implemented via a single integrated circuit chip. There are actually four latches on the latch chip, but only three are needed in this circuit. The two separate clocks on the chip (one for two of the latches, and one for the other two) are tied together, since the latches are used in parallel. The activate signal is used to enable the decoder as shown in the figure.

### MERGE MODULES

The use of level sensitive or DC logic makes the design of the merge modules simple. For the <(serial merge)\Ksm the DEC flowchart primitive is quite descriptive (see Figure 14a): namely it is a negative logic OR gate. Thus, when either activate 1 or activate 2 goes Low, the output is Low, passing control on

as activate-next. The negative logic OR gate is equivalent to an AND gate for positive logic. Similarly, the K(parallel merge) module is shown as a negative logic AND gate, which is equivalent to a positive logic OR gate. (The design of a K(parallel merge) with pulsed logic methods would require a much more complex circuit.)

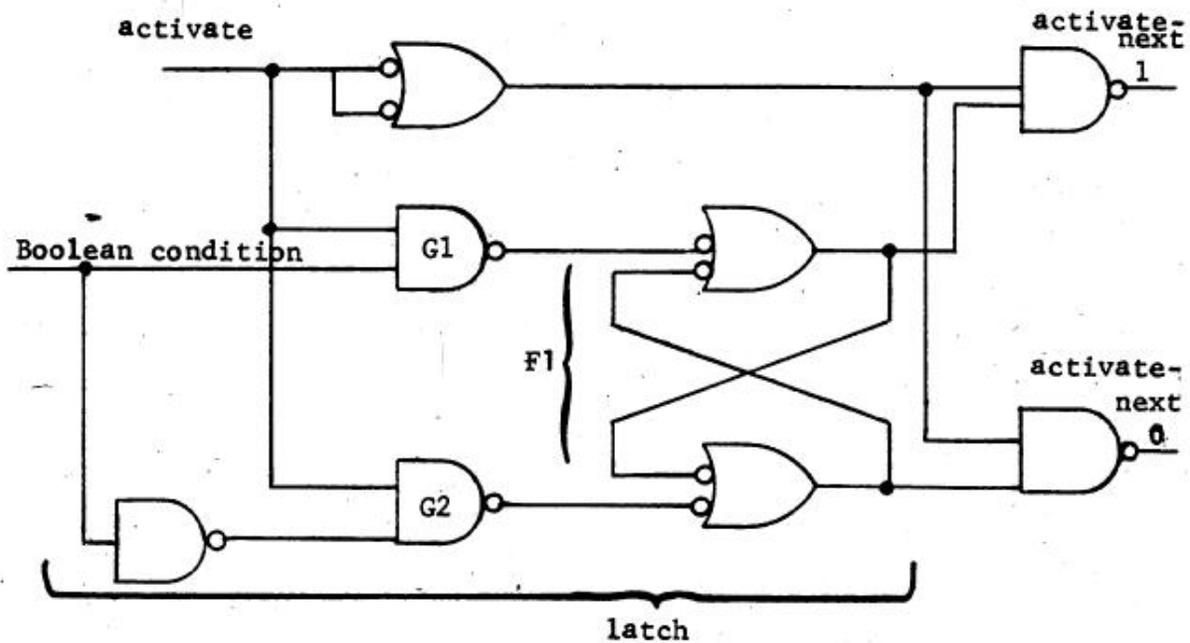
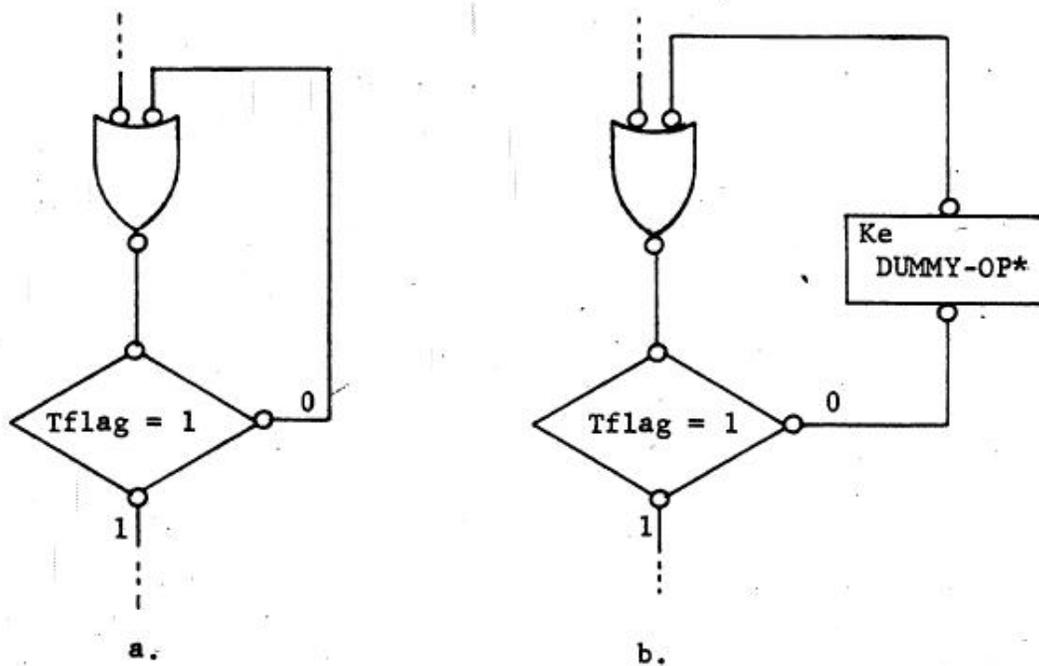


Fig. 11. Switching circuit for (DEC) Kb2.



\* e.g., BSR+0

Fig. 12. RTM diagrams illustrating the use of the Kb2 in small loops.

[previous](#) | [contents](#) | [next](#)

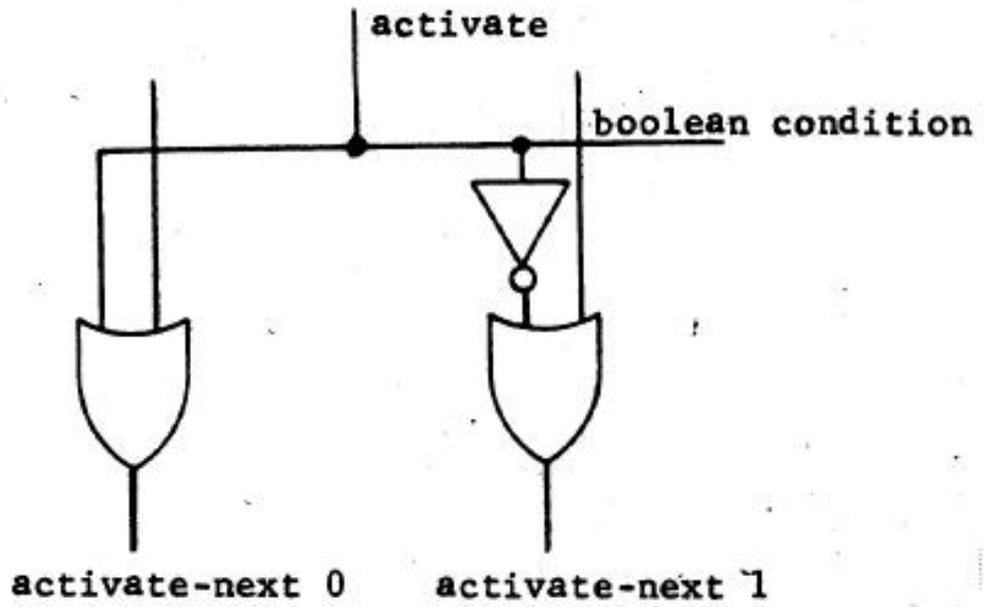


Fig. 9. Switching circuit for a naive Kb2.

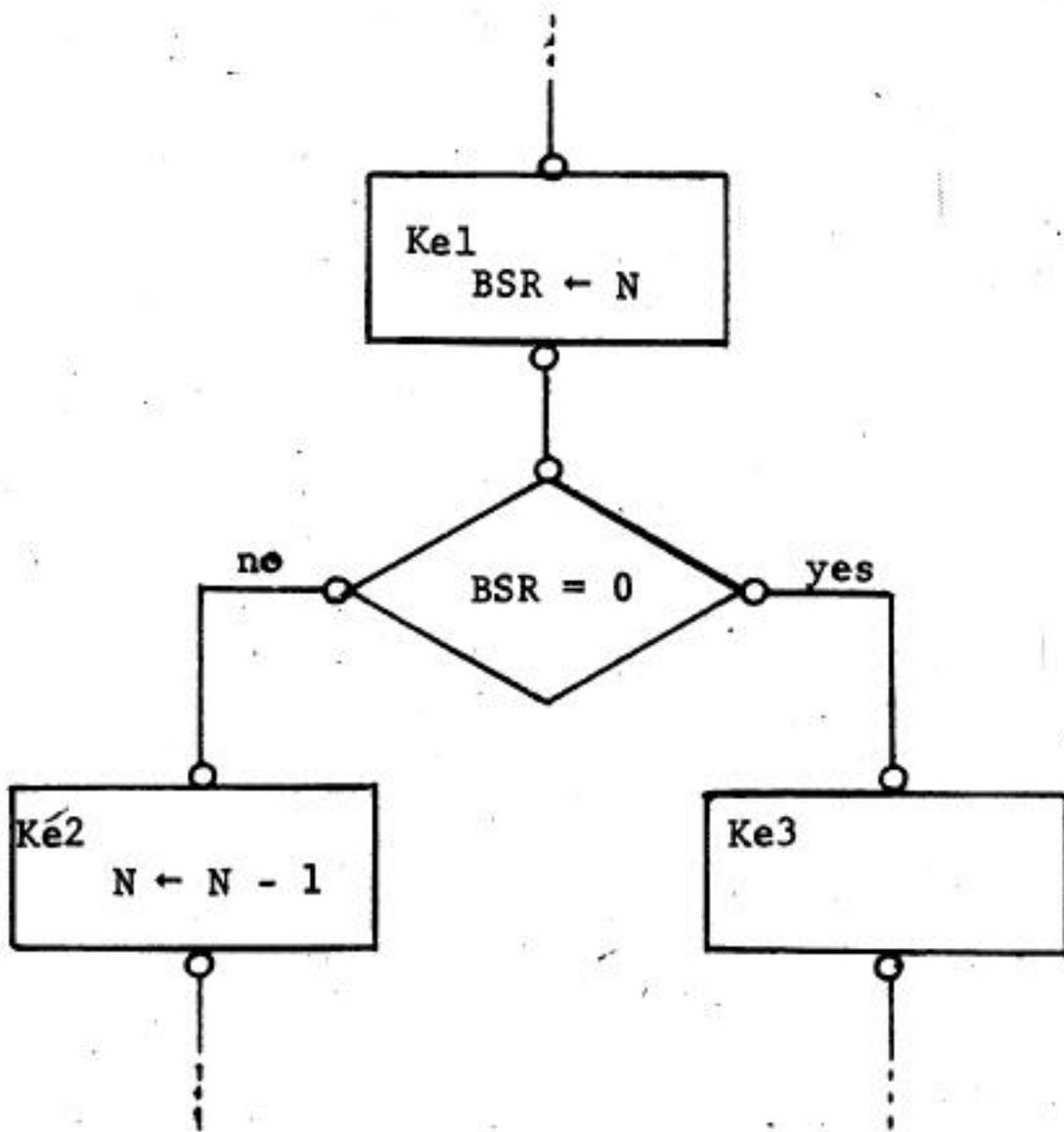


Fig. 10. RTM diagram for a Kb2 example.

is High and the RTM system operates continuously. When that switch is in the MANUAL position, the DONE ENABLE is normally Low, since the Q output of flip flop F1 is High. Thus, when a DATA ACCEPT (DA) signal is received, it cannot propagate to the K modules as a DONE signal (see Figure 7). In this mode of operation the active Ke is terminated by pressing the SINGLE STEP button. This process is characterized by the following steps:

1. Assume that the system is in MANUAL mode, and the DA line is Low, waiting to pass a DONE signal as soon as DONE ENABLE is received.
2. The SINGLE STEP button is pushed, thus activating the single shot shown in the circuit.
3. The single shot presets the flip flop F1 to  $\bar{Q} = \text{High}$ , thus causing the DONE ENABLE signal to be asserted, which then in turn allows DONE to be asserted.
4. After the DONE signal is received, the DA signal goes High and the next Ke is activated. The DA signal transition to high resets F1 to Low ( $\bar{Q}$  is High).
5. When DA goes Low again at the completion of the operation evoked by the current Ke, DONE ENABLE is again inhibited by  $\bar{Q}$  High. To activate the next Ke, the SINGLE STEP must again be pushed.

## BRANCH MODULES

### Two-Way Branch\Kb2

Upon first consideration, one might think that the design of a branch module would be trivial. For example, the circuit of Figure 9 would seem to be perfectly acceptable, and, indeed, it is provided the Boolean input is constant. However, a problem arises if the Boolean condition changes while the Kb2 is being activated. For instance, consider the sub-portion of an RTM system shown in Figure 10. Suppose that after Ke1 is activated, N is a 1. Then the Kb2 will activate Ke2 next, which evokes the action N4-N-1. After this action  $\text{BSR} = 0$ , and since it is possible that this would alter the Boolean input to the Kb2 before the activate-next output of Ke1 returns to a High, the Kb2 might inadvertently activate Ke3, thus causing chaos.

To avoid the above synchronization problem, a flip-flop, F1, is needed to hold the Boolean condition while the branching paths are activated. When the Kb2 is activated the inputs to F1 are disabled, thus the Kb2 will not change its output if the Boolean condition changes while Kb2 is being evoked. The combination of the enabling gates with the flip flop F1 constitutes a latch circuit (see Figure 11). As long as the activate input is High, the output of F1 tracks the Boolean-condition input. However, when

activate goes Low, Fl remembers the last value of the Boolean condition that appeared at its input.

An arbitration difficulty still exists for this circuit, however. In a multiple-Bus system (or a system in which Kb2 is examining an unsynchronized changing input) the Boolean condition might change just before Kb2 module is activated. This would make it uncertain whether the flip flop has settled properly before the branch output is selected. Synchronization problems of this sort were discussed in Chapter 5.

One slight inconvenience is introduced by the presence of the flip flop in the Kb2. Consider the subportion of an RTM system shown in Figure 12a. If Tflag is not equal to 1 when the Kb2 is activated, it is supposed to loop on itself, waiting for Tflag to change. However, once the Boolean condition has been locked into

ACCEPTED\DA signal is asserted on the Bus, again through a wired OR connection, signalling that data has been taken from the Bus.

4. The DA signal is sensed in the Kbus module. Before it is passed on to the Ke modules of the system, signalling final completion, it must pass through a NAND gate enabled by a special DONE ENABLE signal. The DONE ENABLE signal can be used to single-step an RTM system, as will be explained later in this section.
5. The Ke which initially activated  $Ke(T<-A)$  senses the DONE signal and removes the activate signal.
6. This now causes the DR signal to be removed. Note that this return of DR to High bypasses the delay  $T_r$ , as there is no need for a delay at this point. (In fact, speed is-now the concern.)
7. The DR being removed causes DA to be removed, by-passing the  $T_a$  delay.
8. The DONE signal is removed, and  $Ke(T<-A)$  now produces its activate-next output. At this point the cycle for the  $T<-A$  transfer is complete, although the Ke still has to complete the cycle for its activate-next signal.

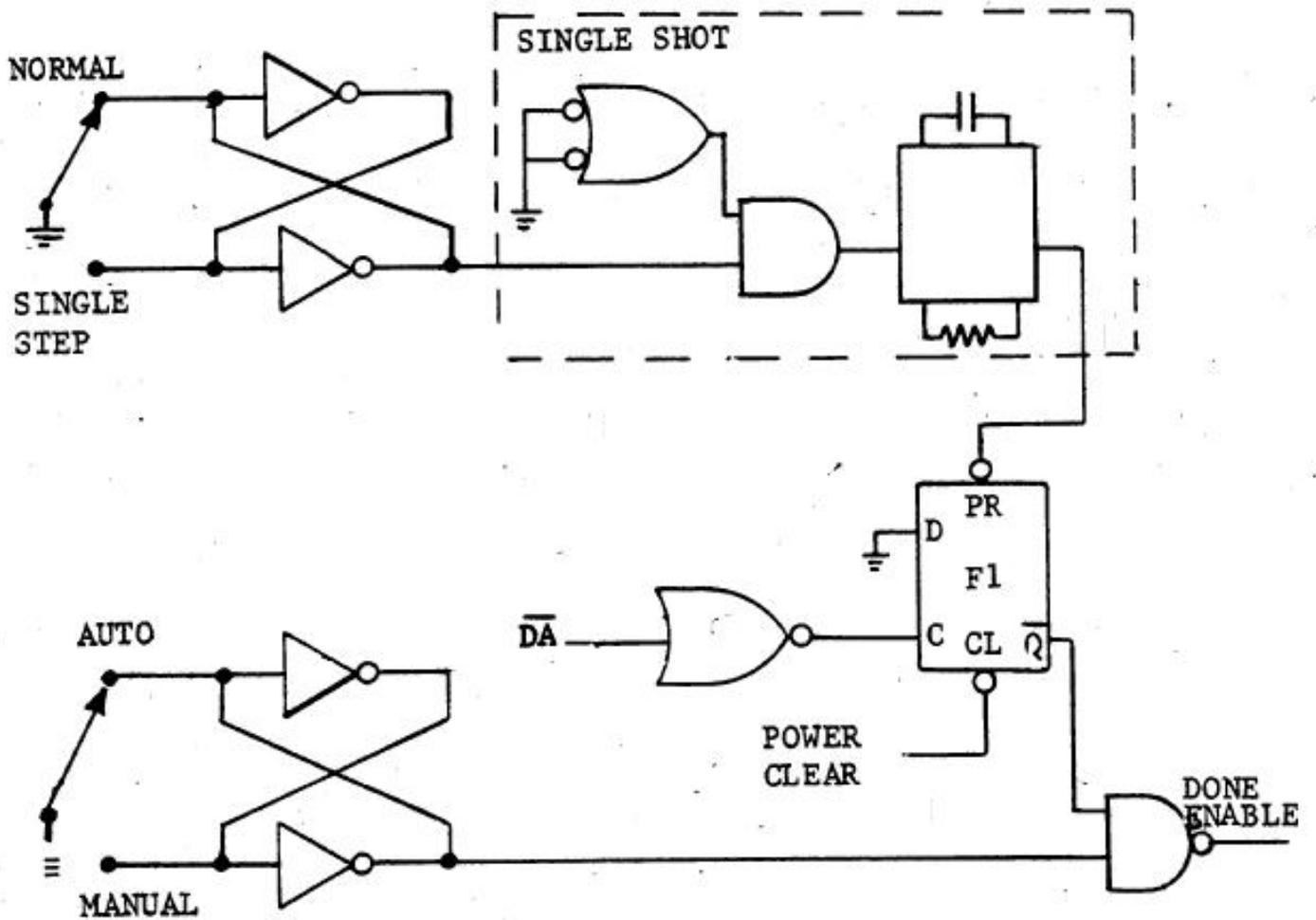
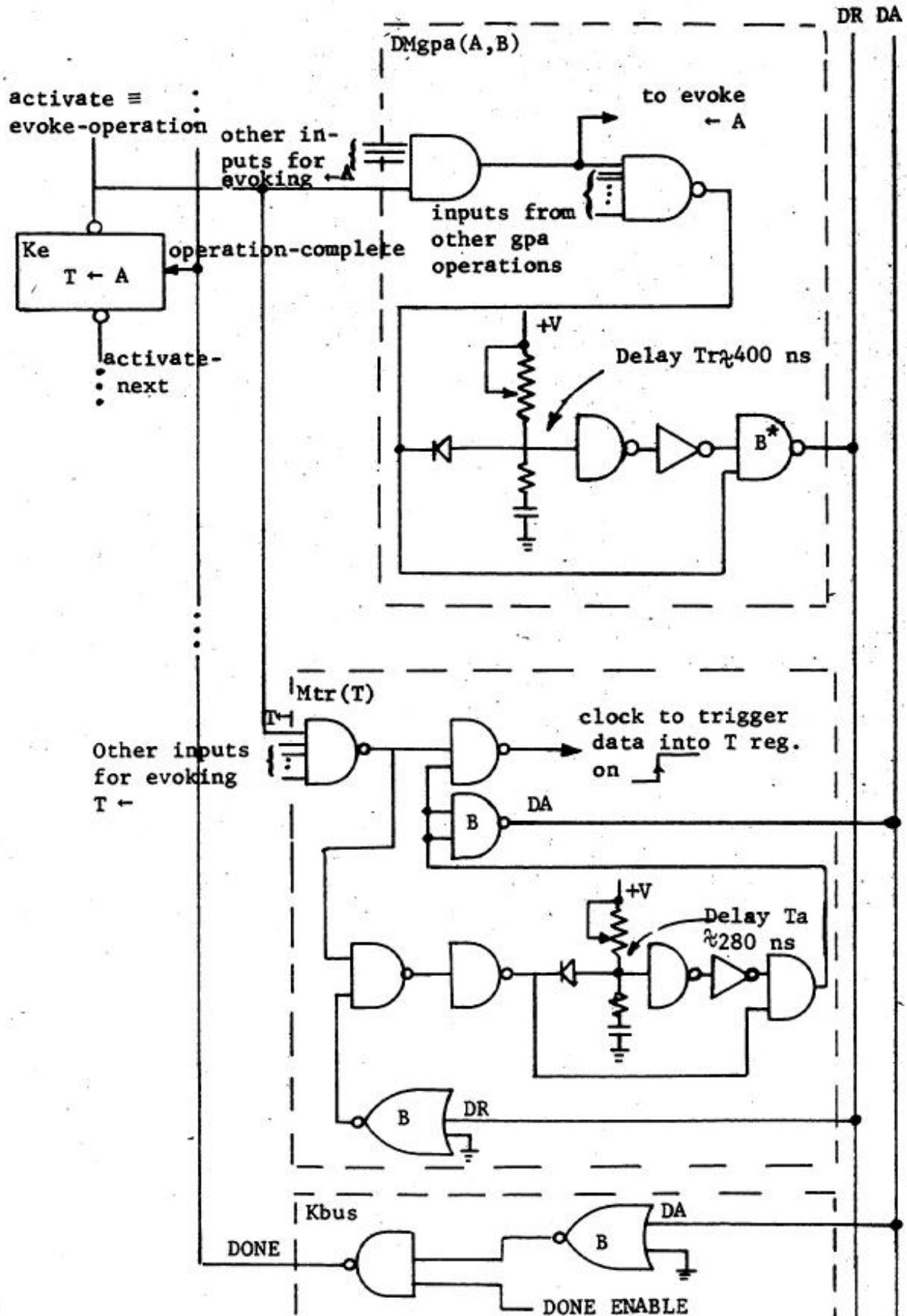
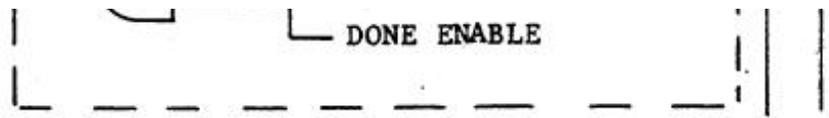


Fig. 8. Switching circuit for DONE ENABLE.

The circuit that generates the DONE ENABLE signal within the Kbus is shown in Figure 8. The AUTO/MANUAL switch and the SINGLE STEP pushbutton are located on the control panel for an RTM system or within T(lights and switches). When the AUTO/MANUAL switch is in the AUTO position the DONE ENABLE signal





\* The B denotes special bus driver and receiver gates.

Fig. 7. Switching circuit for Bus control sequencing.

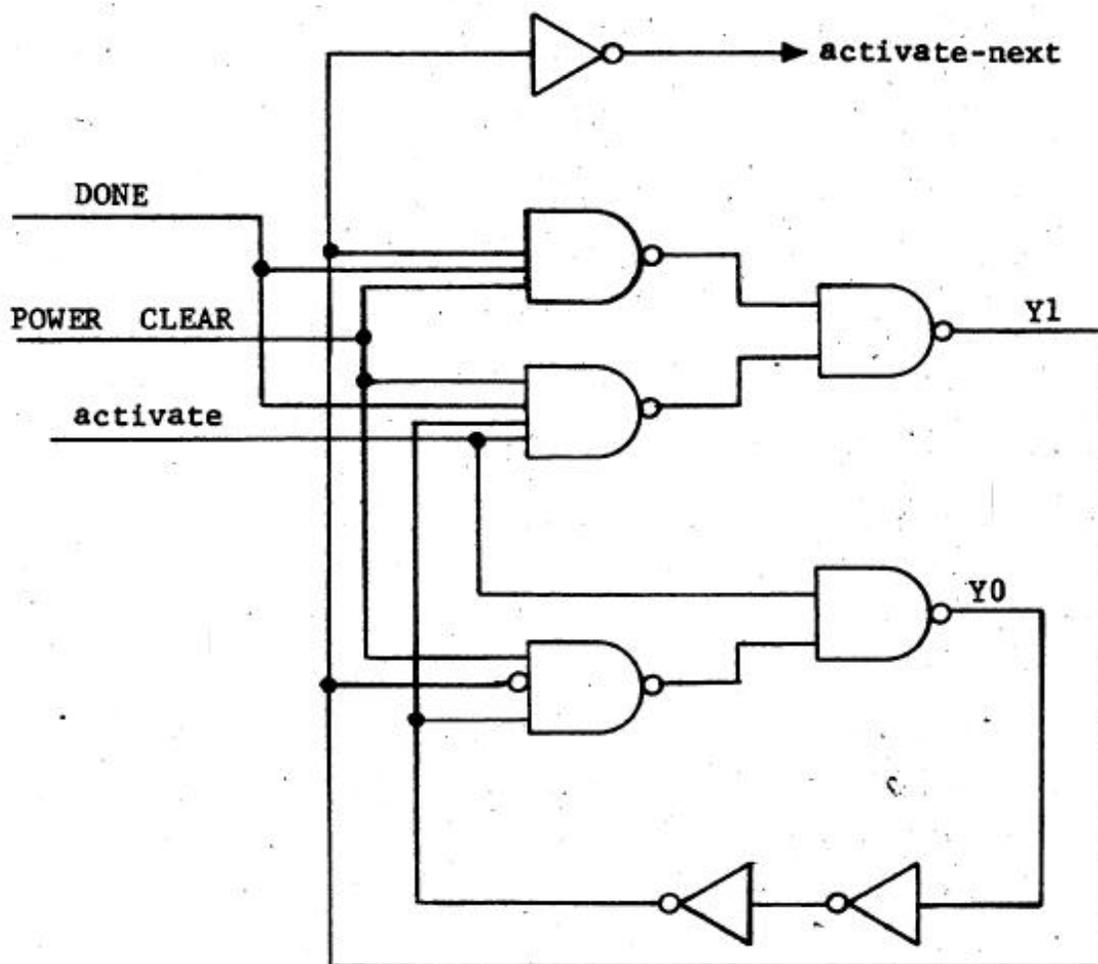


Fig. 6. Switching circuit for K(evoke) using fundamental mode design.

1. The  $K_e$  module is activated, and thus the  $\leftarrow A$  input of the source module,  $DM_{gpa}(A,B)$ , and the  $T \leftarrow$  input of the destination module, here,  $Mtr(T)$ , are asserted by the negative-going signal.
2. Inside the  $DM_{gpa}$  the  $\leftarrow A$  signal causes  $A$  to be placed on the Bus. This signal also goes through two levels of gates that collect the assertion signals for other source data operations that can be evoked in this module. The output goes through a special delay circuit, with delay  $T_r$ . This delay is long enough to allow the calculation of the source data and to gate it onto the Bus. The output of this delay circuit is the negative-going DATA READY\DR signal which is connected through a negative logic "wired-OR" to the BUS DR line. (The TTL logic used to implement RTM's allows such wired OR's. If the DR output of any module goes Low, the Bus DR line goes Low. Interfacing with the Bus is done using special "Bus driver" and "Bus receiver" gates,

marked with a B in the figure.)

3. Within the destination module, having had the  $T<$ - input asserted, it awaits the DR signal. When DR is sensed, a delay circuit, with delay  $T_a$ , is timed out to allow the input stages of the flip flops of the T register to settle. After the delay, the data is clocked into the T register, and the DATA

Whenever power is turned on or an RTM system is cleared, a Low pulse on the POWER CLEAR input clears flip flop F1 to  $Q = \text{Low}$ . In addition, since DONE is resting at logical High, flip flop F2 will be reset to  $Q = \text{High}$ . Thus  $(F1, F2) = (\text{Low}, \text{High})$  corresponds to state Q0. When the Ke is activated (i.e., the activate input goes Low), F1 is preset to a High via its Preset (PR) input. Thus,  $(F1, F2) = (\text{High}, \text{High})$  corresponds to state Q1. If the Ke is in state Q1 and DONE is asserted (i.e., DONE goes Low), F2 is set to the Low state and the Ke ends up in state Q2, i.e.,  $(F1, F2) = (\text{High}, \text{Low})$ . While in state Q2, when the DONE input returns to logical High the Ke asserts its activate-next output (i.e., activate-next goes Low). The next time the DONE input is asserted, the Ke returns to its inactive state, Q0, via the following chain of events. First, DONE goes Low and disables activate-next. When activate-next goes High it triggers F1 back into the Low state. F1 being Low will reset F2 to High as soon as DONE returns to High, and thus the Ke is again in state (Low, High). An interesting question is, what happens if DONE returns to High before F1 returns to Low? We would then get an undesired Low pulse on activate-next, i.e., a hazard. In the next section it will be seen that this hazard never occurs because of the way in which the DONE signal is generated.

### Fundamental Mode Design of K(evoke)

DEC implements the Ke with discrete memory elements because that mode of design is most suitable to the use of small and medium scale integrated circuits as switching circuit building blocks. Although this design is their current commitment, future generations of RTM's might be fabricated using LSI techniques. Thus, a completely custom-designed Ke might be feasible. To save gates, and for faster operation, one could implement in LSI a fundamental mode version of the Ke, such as the one shown in Figure 6. This circuit takes 8 gate equivalents (three of which are inverters), as contrasted with 9 for the design using discrete memory elements. Also, in this circuit the three operating states are defined somewhat differently (see Figure 5a). State q0 for the fundamental mode circuit is equivalent to state Q0 for the DEC circuit, and it is encoded as  $(Y1, Y0) = (\text{Low}, \text{Low})$ . State q1 is the state in which the Ke is evoked, but not producing an activate-next output, and is encoded  $(Y1, Y0) = (\text{Low}, \text{High})$ . State q2 is the state in which the Ke is producing an activate-next output, and is encoded  $(Y1, Y0) = (\text{High}, \text{Low})$ . The delay in the feedback path for Y0 prevents a hazard in the activate-next output during the-transition from state q1 to state q2. The reader should carry out a fundamental mode design in which this feedback delay is not necessary to avoid critical races or hazards. Such a circuit would be faster and more reliable than the one shown.

### BUS CONTROL SEQUENCING

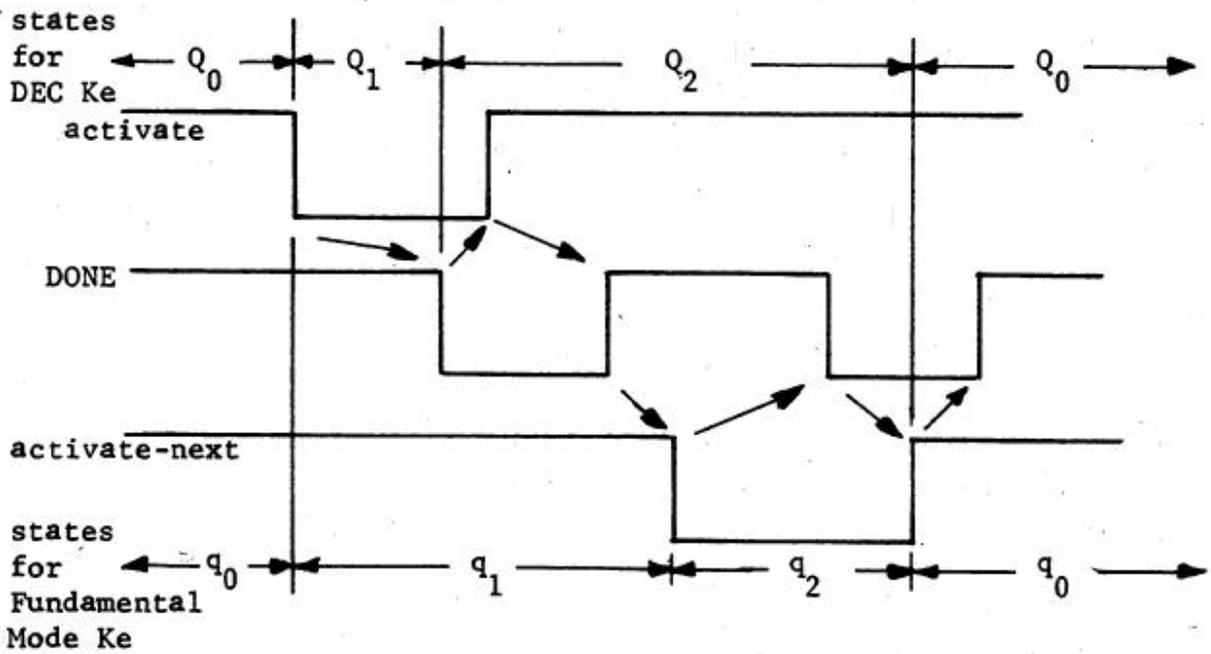
The description of the K(evoke) that was just given accounts for a significant portion of the behavior in the control signal timing diagram in Figure 4. Now, taking the switching circuit design of the K(evoke) as given, the circuit realization which accounts for the behavior in the rest of that diagram can be carried out. For example, consider the register transfer  $T \leftarrow A$  being evoked between an Mtr and a DMgpa in Figure 7. The figure shows simplified versions of the modules that carry out the sequencing of the

transfer correctly. Only the relevant portions of the modules are shown; the registers and the data operator circuits are not shown (although the control signals for them are).

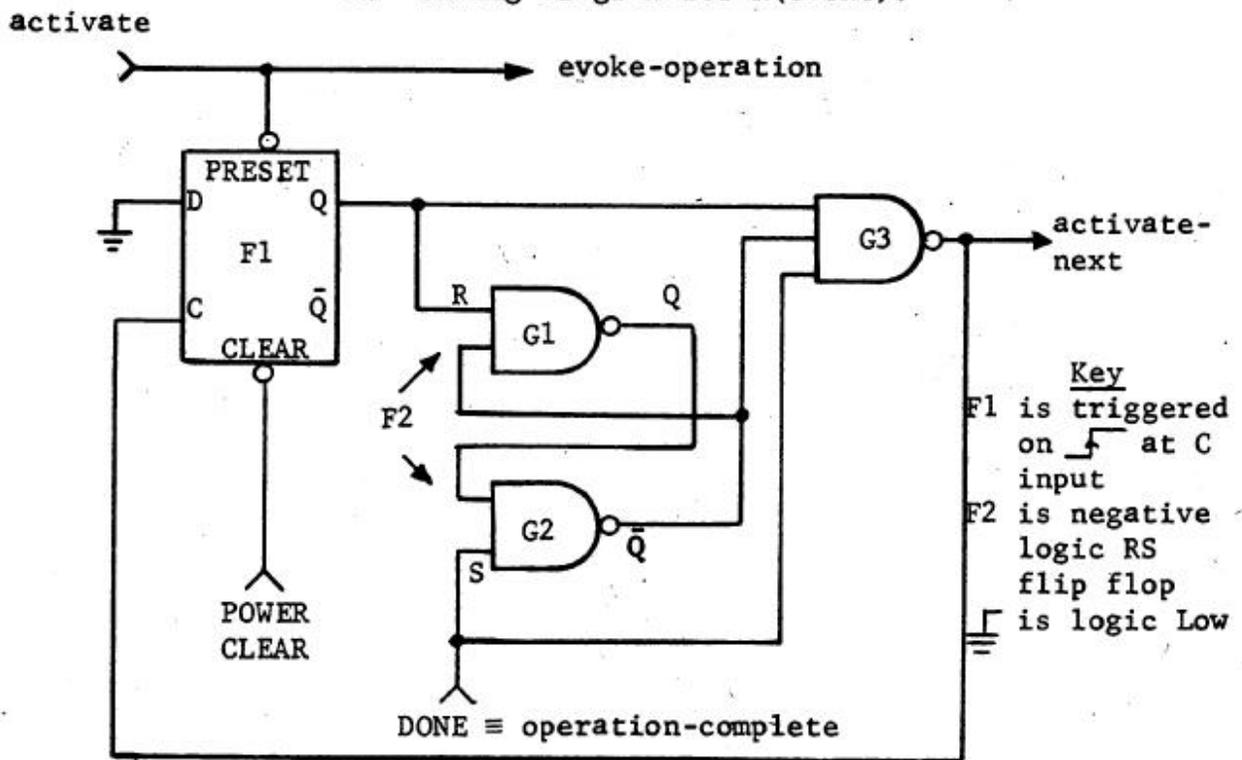
Keeping the diagram from Figure 4 in mind, the sequence of events for Figure 7 is as follows:

361

[previous](#) | [contents](#) | [next](#)



a. Timing diagram for K(evoke).



b. Switching circuit diagram for K(evoke).

Fig. 5. Diagrams for DEC K(evoke).

[previous](#) | [contents](#) | [next](#)

4. The Kbus module senses the DA signal and passes it on to all K(evoke) modules as the DONE signal.
5. When Ke1 senses the DONE signal as its operation-complete input, it removes activate 2, which causes DR to be removed, which causes DA to be removed, which causes DONE to be removed which causes Ke2 to now assert its output signal, activate-next 2.
6. Now the cycle for activate-next 2 is repeated in the same form as the cycle for activate-next 1 (activate 2).

Two questions might arise when examining the timing diagram: Why not use pulses instead of level changes as significant events? Why negative logic? In answer to the former, pulse mode design for the K(evokes) was tried and found to present several problems. First, for debugging, pulses are hard to observe, even using an oscilloscope, because they must be captured. Second, pulse mode logic is inherently slower than fundamental mode (level change) logic, because two transitions are required for an event. Also, certain modules, such as the K(parallel merge) are harder to design using pulse mode logic. Finally, single- stepping the system is more difficult using pulse mode logic.

Negative assertion was used because in the TTL logic used to implement RTM's, a floating (i.e., unconnected) input to a gate becomes a 0(High) signal.(2) Thus if an operation-evoke input to a DM module is left unconnected, this means that input is unevoked (disabled). Consequently, in wiring a system, only those inputs that correspond to evoked operations are connected, greatly saving effort (and causing less errors to be automatically designed in).

## DESIGN OF THE K(evoke)

Now that the fundamental signal sequencing method has been presented, we can look at the detailed logic design of the modules involved. The most basic module is the K(evoke)\Ke.

### Design of K(evoke) Using Discrete Memory Elements

Recall that the explicit input to a Ke is activate (this control), and its explicit outputs are evoke-operation (equivalent to activate) and activate-next (control). In addition, there are the two other inputs which do not appear on flowcharts: DONE (operation-complete) and POWER CLEAR. Both inputs are usually prewired for all K(evokes).

The timing diagram for a Ke, showing its behavior, is given in Figure Sa. It can be considered as a three state device: in state Q0 it is inactive; in state Q1 it is active, but disabled from producing an activate-next since the operation it has evoked is not yet completed; and in state Q2 it is active and enabled for producing an activate-next when DONE returns to a logical High. Thus the Ke cannot evoke another Ke

while DONE is asserted, as this could activate the second Ke. The three states of the Ke can be realized using two memory elements, as shown in the DEC implementation of the Ke in Figure 5b. The two memory elements are the edge triggered D-type flip flop F1, and the negative logic 2 NAND gate RS flip flop F2.

-----

2. In this chapter we continue to use the logic conventions established in Chapter 2. That is, for positive logic, the logical High level is asserted (a 1), and the logical Low level is not asserted (a 0). For negative logic, the logical Low level is asserted (a 1), and the logical High level is not asserted (a 0). All gate and flip flop symbols are to be interpreted according to the logic levels at their inputs and outputs (i.e., High and Low, not 1 and 0). Thus the symbol for a NAND gate designates that if all inputs are High, the output is Low, etc.

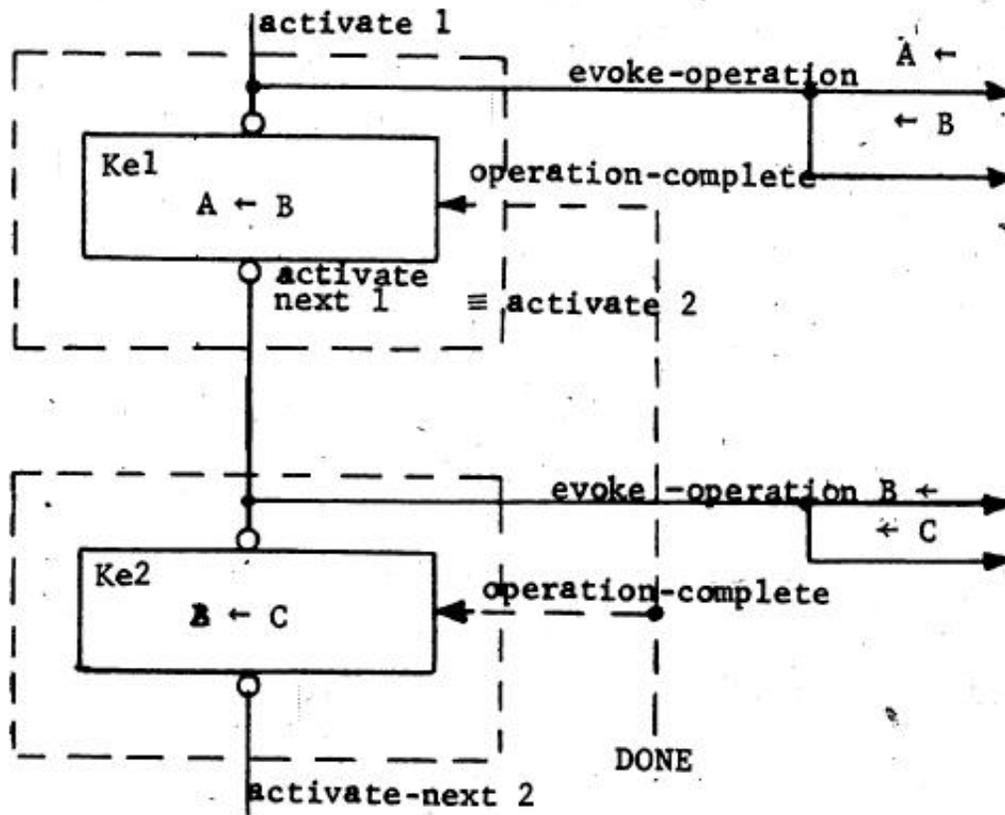
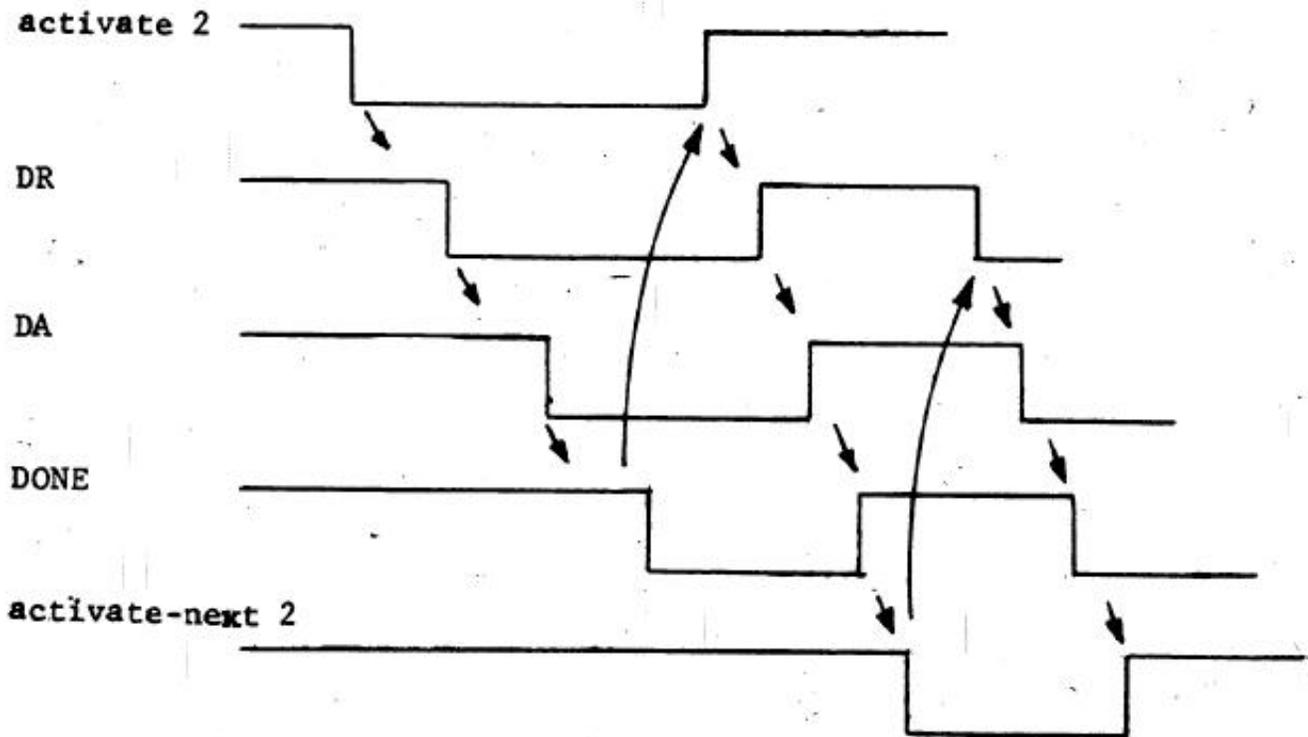


Fig. 3. RTM diagram of two K(evokes) in sequence.



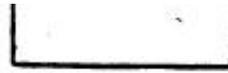


Fig. 4. Timing diagram for control signal sequence.

1. Logic: TTL. Acceptable for speed and noise immunity; internal constraints override; and cost is low (partially because of MSI).

2. Packaging: placed on DEC PC boards of 5 x 8 1/2" or 2 1/2 x 8 1/2" with 72 or 36 pins.

3. Wiring: wirewrap and push-on connections over wirewrap pins. Cost, manual, and automatic wiring considerations.

4. Logic interconnection rules: one kind of control signal. User has to count number of modules for power supply and certain bus loads. Very small number of rules compared to IC use.

5. Problem size: 4 ~ 100 control steps; 1 ~ 4 arithmetic registers; 100 variables; possibly read-only memory.

6. Word length: 8, 12, and 16 bit. Large enough for most applications; computer-related problems need 18, 24, 32, and 36 bits; internal and user--present de facto standard.

7. Universality and extendability: these modules are not a panacea. We provide escape to other systems, regular IC's, standard modules, PDP-11 computer components, and PDP-11 computer.

8. Selection of primitives: a basic register interconnection structure and data representation was first posited. The operations which formed a complete set for the data representation were then specified. With this basic module set, designs were carried out for benchmark problems.

9. Notations: modified PMS and ISP. The structure (interconnections) and the naming of the modules are essentially the PMS notation. The basic modification to PMS was to "box" the components to be more conventional.

10. Automatic (algorithmic) mapping of algorithm into hardware: the basic register transfer design archetype representation is a flowchart. The register transfer operations are expressed in the ISP language.

11. Parallelism\* and speed: provision was made for multiple busses and the modules are asynchronous. The applications classes put relatively low weight on speed. For teaching purposes, it was included because it is an important principle. A decision to use a bus, and thereby limit parallelism to the number of busses, was made for both cost and simplicity reasons. Most systems have parallelism of 1. Each arithmetic expression operates at its own speed in an asynchronous fashion.

\* Number of operations that can be made to occur at the same time.

set, designs were carried out for benchmark problems.

\* Number of operations that can be made to occur at the same time.

Fig. 2. Table of basic Register Transfer Module design decisions.

deviation would have created very high setup costs. Also, since most users of IC's have previously made the same decision, the manufacturing cost of TTL is lower. In fact, reasons for using, say ECL, such as higher speed and higher noise immunity, were not of sufficient importance to overcome the extra manufacturing, development, and setup costs.

Another constraint related to current MSI technology is word length. Integrated Circuit registers and data operations are commonly available in packages of four or eight bits. Furthermore, problems in the computer area usually dictate the use of word lengths of 8, 12, or 16 bits. Thus there was no conflict in choosing these word lengths for RTM's. If a need for word lengths of larger than 16 bits materializes, the word length can be extended by producing another set of modules.

## SET OF MODULES

In view of the constraints discussed above and shown in Figure 1, a final set of engineering decisions was made regarding the modules. Some of the more important ones are listed in the table in Figure 2.

## SWITCHING CIRCUIT DETAILS

This section deals with the switching circuit level design of RTM's. Much of this design is straightforward and conventional and will not be covered in any detail. Instead, those aspects of the design that give RTM's a unique, modular behavior will be presented. The intention herein is to assist the potential module designer, and also to provide enough detail so that one could extend the RTM concept by interfacing various components into the RTM framework.

## CONTROL SIGNAL TIMING DIAGRAMS

The most interesting and most important characteristic of RTM's is the method used to pass control among the K modules and between the K portion of a system and its DM part. That is, how do K modules activate one another, and how do they evoke operations in the DM part of the system? These questions can be answered by first looking at a control signal timing diagram for two sequentially connected K(evokes). Two such modules are shown in Figure 3.(1) They both evoke simple register transfers on the RTM Bus. The control signal sequence for their operation is shown in the, timing diagram in Figure 4. The diagram uses arrows to show the cause and effect relationship between events. Notice that each logic level change is an event, and signals are at the logical Low level for assertion, i.e., negative logic is used. The interpretation of the sequence is as follows:

- 1. The activate input to Ke2 is asserted by Kel, thus asserting the evoke- operation inputs to the appropriate DM modules to evoke the desired operation.
- 2. When the data from the source DM module (i.e., the source of C) is ready, that module asserts

the DATA READY\DR signal on the Bus.

3. When the receiving DM module (i.e., the module containing the B register) senses the DR signal, it accepts the data on the Bus and asserts the DATA ACCEPTED\DA signal on the Bus.

-----

1. Here the RTM boxed notation is superimposed on DEC's PDP-16 notation. Since we are discussing DEC's implementation of the modules, we shall use mostly the PDP-16 notation in this chapter.

## CONSTRAINTS

Constraints imposed on the choice of an RTM set by the application areas can be broken down into two types: problem-dictated constraints and user-dictated constraints. In addition, hardware technology and internal constraints are imposed by the realities of the production environment, in this case those at DEC.

These constraint categories are discussed in detail in the following sections. The manner in which these constraints manifest themselves for each of the application areas is shown in the table of Figure 1.

### Problem-Dictated Constraints

Within each application area, each digital system problem can be characterized by its hardware (and/or programming) requirements. Such constraints, for example, include word length, number base, and operation types. This set of constraints is a measure of the problem size.

We further measure problem size by the number of control steps (statements), the number of registers (and register operations), and the amount of memory needed for constants and variables (see Figure 1).

In terms of these problem requirements, RTM's were designed to accommodate up to 100 hardwired control steps, multiple arithmetic units, a small read-write memory of about 100 variables, and possibly a read-only memory. Other problem constraints, e.g., speed, noise, and reliability, also influenced the basic RTM design.

### User-Dictated Constraints

User-dictated constraints (see Figure 1) are both objective and subjective because they reflect how a human user views the modules (e.g., with respect to cost, ruggedness, and the ability to debug) for solving a particular class of problems.

The cost constraint is very important in most problem classes. Thus, decisions for the modules like packaging, fixed word length (8-, 12-, and 16-bit), and the method of interconnecting were all made to keep the cost low. RTM's are able to compete with relatively poorly packaged, overpriced, underproduced, or past produced computers just by having a simple structure and high volume production.

Perhaps the most important user requirement is the ability of RTM's to be used smoothly and rigorously in a design process. Such an ability directly affects design time and, therefore, cost. In the case of conventional logic design, the user selects registers, attaches data, operations and switches, and then designs a sequential circuit to evoke the system operations (see Chapter 8). Hopefully, such a design is based on a precise description such as a state diagram or a flowchart. Thus, since a logical design

problem is usually solved through the use of some high level description -- in this case a register transfer flowchart -- we chose such a high level description as the basic and only design documentation for RTM's. This particularly influenced the model for the control part of RTM systems. We wanted to make it difficult to build poor systems by poor design and documentation practices.

## Hardware Technology and Internal Constraints

The hardware technology and internal constraints dictate the parts from which the modules are to be constructed. Logic types other than TTL (e.g., DTL, RTL, ECL) were only briefly considered because TTL was the local standard, and

Design Requirements	Special Purpose Digital System	Computer Related	Educational
User requirements: cost design time debug facilities  inventory  user experience  basis for a methodology  Automatic techniques mechanical ruggedness wiring technique	relatively important very important yes  max flexibility (many modules) wide variation  some knowledge of flowcharts and registers  no access to a computer  can be relatively fragile  wire wrapped (automatically)	slightly important very important yes  max flexibility (many modules) highly trained  knowledge of registers, register operations, and flowcharts  access "limited", e.g. FORTRAN  see special purpose  see special purpose	very important unimportant yes (preferably, without scope) relatively few modules theory but no practice logic design, register transfer, programming  access to FORTRAN  needs to be drop-proof  plugboard (manually pluggable)

354

\* Tables and/or program memory (to reduce control states).  
 † Interfacing with common computers may dictate requirements.  
 ‡ Industrial applications require high noise immunity, low speed.  
 \*\* Due to temporary constructions with poor wiring, high noise immunity is required.

Fig. 1 (Page 2 of 2).

Design Requirements	Special Purpose Digital System	Computer Related	Educational
Typical equipment examples	A-D, phone-line interfaces to special equipment, tape to plotters, conveyor to controllers, ... instrument analysis and control	plotter, tape controls, display and FFT processors, peripherals for System/360, equipment for and emulation of computers (e.g. 1401)	multiplication, examples using parallelism, simple computers and desk calculators (e.g. BITRAN-6, PDP-8)
Equipment dictated requirements: control steps read-write memory read-only memory word length in bits speed noise immunity reliability space need for special modules  current solutions to problem  mechanical  arithmetic data types	2-20, 10-100, 100-2000 2-10, 10-100, 100-1000 0-4000* (application) < 16 < 1 MHz usually † relatively high ‡ medium relatively small analog, special IC's and level conversion hardwired, ... general purpose digital computers	4-200 0, 4000-16,000 0 8,12,16 (also 18,24,32,36) 0.5-5 MHz (parallelism) medium high should be small (interface to computers)  hardwired (with auto design aids) microprogramming	2-100 0-128, 128-4000 0 (possibly 8, 12, 16)† (to exhibit parallelism)† should be high ** low should be large no  special logic modules, special "teaching" computers, low cost computers free standing, human accessible all types of integers, Booleans, real, decimal
	separate & within racks	within the computer	
	unsigned integers, Booleans, decimal	usually 2's complement integers, Booleans	

Fig. 1 (Page 1 of 2). Table of specific problems and design requirements for digital systems.

## CHAPTER 7 THE DESIGN OF RTM's

The design of the RTM modules is presented with two goals: to communicate detailed information on the structure of the modules in order that the operation may be better understood; and to stimulate the reader to think about the process of choosing and designing a set of register-transfer level components for implementing digital systems. This latter goal is important both because it is likely that other sets of such components will be developed in the future (particularly some useful integrated circuits for control), and because all logical design should involve a conceptual structuring into modules. Thus, the chapter is organized into two major sections: the first presents the general engineering design considerations that entered into the choice of the RTM set of modules; and the second discusses the important aspects of the switching circuit realization of, the modules.

### CHOOSING A MODULE SET

#### APPLICATIONS

The design of a set of register-transfer level components presents unique problems that are not encountered in other digital components. For example, in choosing a set of gates and flip flops at the switching circuit level, the functions of the components are very basic; and many small sets exist that are universal, i.e., can be used, to realize any function (e.g., NAND|NOR|AND+NOT|OR+NOT). At the other extreme large scale computers are rarely designed for the primary purpose of being components, so they are usually universal in themselves.

However, register-transfer level components do not have functions as basic as those at the switching circuit level, yet they are used as components in larger systems. Thus, the choice of the functions that the modules perform becomes nontrivial. Furthermore, the type of systems that they will be employed in influences their design. Therefore potential applications became a big factor in determining the RTM set of modules. As for universality, there are many more possible universal sets at the RT level than at the switching circuit level. Thus, in a sense, the choice becomes more difficult.

Three main areas of application were considered in the design of the RTM set and the interconnection scheme: special purpose digital systems, computer-related systems, and educational systems. In Figure 1 we present a rough categorization of examples within each of these areas, showing those cases in which RTM's would be an appropriate solution.

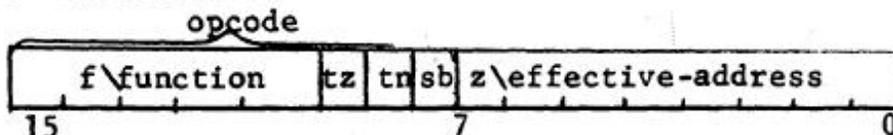
Within each application area there is a wide range of problems. In the special purpose digital systems area, problems range from A-D and D-A conversion to instrument control and analysis. In the computer-related area, problems range from controllers for plotters and card handling equipment to special processors and the emulation of older computers which are no longer produced, but which have adequately seasoned software. In the educational area, problems range from simple computer arithmetic

to the construction of small computers. The reader should be already quite familiar with this full range of problems, as they are covered in previous chapters of this book.

Depending on how the various application areas and their requirements are weighted, a different set of modules emerge. For example, weighting the educational requirement heavily might cause highly rugged components to be built. In the actual design of the modules all three applications areas were weighted about equally.

$M[0:15] \langle 15:0 \rangle$       the memory of DMar, and the only memory of the computer  
 $PC := M[0]$       the PC is mapped into the DMar  
 $AC := M[1]$       the AC is mapped into the DMar

$i \langle 15:0 \rangle$  instruction



$op\text{-}code \langle 7:0 \rangle := i \langle 15:8 \rangle$

$f \backslash \text{function} \langle 4:0 \rangle := op\text{-}code \langle 7:3 \rangle$

$tz \backslash \text{test-zero} := op\text{-}code \langle 2 \rangle$

$tn \backslash \text{test-negative} := op\text{-}code \langle 1 \rangle$

$sb \backslash \text{store-back} := op\text{-}code \langle 0 \rangle$

$z \langle 7:0 \rangle \backslash \text{effective-address} := i \langle 7:0 \rangle$

function provided by DMar

skip 2 words if zero

skip 2 words if negative

place AC back in M[z]

$Interpret := (i \leftarrow M[PC]; PC \leftarrow PC + 1; next$   
     Execute-instructions; next  
     Interpret)

Execute-instructions := (

$AC \leftarrow f(AC, M[z]); next$       carry out function specified by f

$(tz \wedge (AC=0)) \vee (tn \wedge (AC < 0)) \Rightarrow PC \leftarrow PC + 2; next$       skip

$sb \Rightarrow M[z] \leftarrow AC$       store result back in M[z]

Fig. Crtm-2/2-1. ISP description of Crtm-2/2, a half-finished simple RTM computer using the DMar.

[previous](#) | [contents](#) | [next](#)

The internal processor (calculator) state which the user sees is very important. Should there be a single register (accumulator), multiple internal registers, or a stack organized memory? Also, perhaps the calculator should be programmable, providing the capability to store the operations to be performed. Adding program capability to the calculator offers more interesting possibilities. By being programmable, both multiply and divide might be programmed rather than hardwired. The design should have operations which are accessible directly and via the program. Conditional operations are necessary for a programmable calculator. The programmability introduces several other problems: (1) How is a program loaded into memory? and (2) if inputs must go through a BCD to binary conversion, then instructions which utilize addresses (in binary form) become awkward to input in BCD.

## PROBLEMS

1. What are the cost and performance for a minimum non-programmable calculator which will just add, subtract, and multiply?
2. Design (1) and compare with a more elaborate calculator which still has only the three operations.
3. Design a programmable calculator which is based on principles similar to those used in the computers of this chapter.

### ■ Crtm-2/2: A SIMPLE RTM COMPUTER USING THE DMar

KEYWORDS: DMar, computer

At the end of Chapter 2 a new module, the DMar, was introduced. Since it has a 16-word scratchpad memory built in, one can build a simple RTM computer utilizing only a DMar, a T(lights and switches), and a Kbus for the data part. A partially completed (approximately 1/2) ISP description of such a computer, the Crtm-2/2, is given in Figure Crtm-2/2-1. The following problems are based on this computer.

## PROBLEMS

1. Finish the ISP description of the half-finished Crtm-2/2 and then implement it using RTM's, thus making Crtm-2.
2. Notice that the skip instructions skip two words rather than one. Why is this so? (Hint: Try writing some non-trivial programs for this machine.)
3. Design a version of Crtm-2 that only skips one word on skip instructions, yet is just as generally

programmable as the version that skips two words. Compare the compactness of similar programs in the two machines. Compare their cost; their speed.

4. Notice that the effective address of the Crtm-2/2 instruction word is 8-bits long. Design a version of the machine with a 256-word memory. How are the 16 words of the DMar scratchpad utilized to the best advantage in such a computer? Compare the cost and performance of this machine with those described earlier in this chapter.

6. Add input-output instructions and a direct memory access facility.
7. Suggest modifications to the DMgpa to improve the capability to implement Cgr. (E.g., add 4 to 8 registers as part of DMgpa.) Does DMar do it better?
8. Carry out a similar conversion task for the DEC PDP-11 based on the ISP description in the PDP-11 manual.

## SIMPLE DESK CALCULATOR

**KEYWORDS:** Desk calculator, BCD, arithmetic, programmable

This problem is the design of a simple desk calculator. The problem was given successfully as a laboratory exercise to juniors at Carnegie-Mellon University. The calculator should be capable of adding, subtracting, multiplying, and possibly dividing. Input numbers can be limited to two decimal digits while output may be up to four decimal digits. Since this is to be a decimal calculator, binary input and output is unacceptable. For example, to input the number  $12 \times 10$  from the switches the BCD form,  $0012 \times 16$ , should be used rather than the binary form,  $000A \times 16$ . A Users Manual for the design should be written so that someone unfamiliar with digital systems could use the calculator. The conventional T(lights and switches) may be used to encode the BCD input and output.

## DESIGN CONSIDERATIONS

Perhaps the first major decision in the design is whether data should be represented in binary or BCD form within the calculator. Since RTM's already use the binary representation for addition and subtraction, there is some merit to using binary, although a BCD to binary conversion routine is required. The BCD to binary conversion operation usually uses the division operation. Thus the division operation might be available within the calculator at almost no extra cost.

Various design objective functions need to be set, i.e., should the calculator perform at high speed, should it have low cost, or should it have features to make it easy to use (sophistication)? Consider for example, the speed, cost, sophistication trade-off. For high speed a fast multiply algorithm (e.g., those in Chapter 4) might be used. These are usually more costly than the algorithm which simply adds the multiplicand to itself while decrementing the multiplier. But the increased speed is hardly required, since at most 100 additions need be performed in the adding multiply algorithm. In any case, multiplication would - appear instantaneous to the user as long as it takes less than 100,000 microseconds. On the other hand, a fast multiply design might enable the sine function to be calculated rapidly using polynomial expansion. Again, an analysis might show that the number of operations required to achieve four decimal digit accuracy (all that can be represented under the design constraints of the problem) might be small enough to negate the advantages of a high speed multiplier.

Other features of the calculator also exhibit similar design trade-offs. The decimal display, signed number representation, and decimal point influence the complexity of the calculator. Negative number operations are desirable and relatively easy to implement, while the inclusion of a decimal point is more difficult. Error indications such as overflow or division by zero are very helpful. A structure whereby the results of the previous operation is used as one of the operands in the next operation is very convenient from a user's viewpoint (essentially necessary for, say, adding a column of figures).

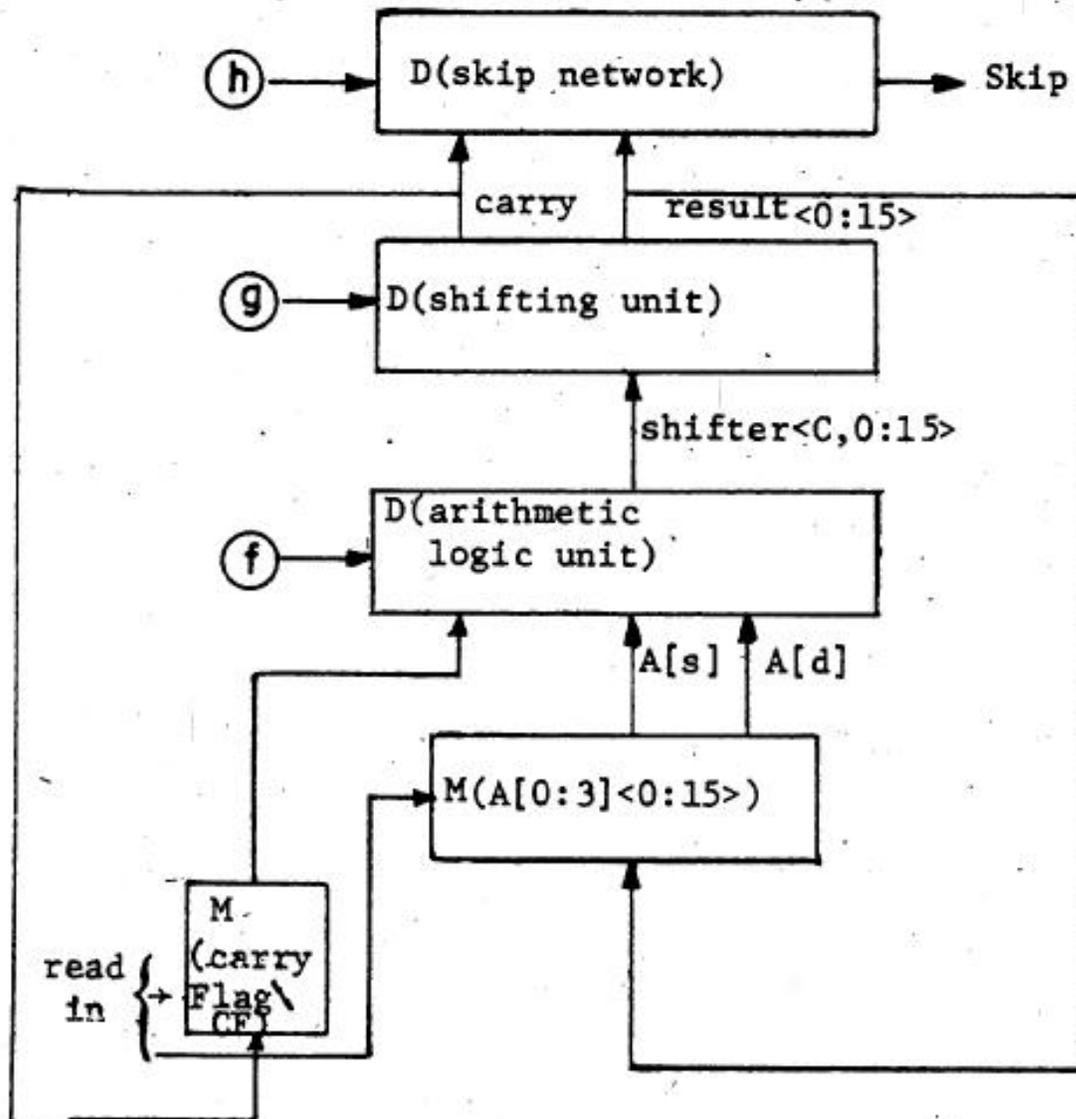


Fig. Cgr-3. PMS structure of the registers and data operators of Cgr.

and data operators (see Figure Cgr-3). First, some registers (one or two) are used as operands for the COMplement, NEGate, MOVE, INCrement, DECrement, ADd Complement, SUBtract, ADD, or AND instructions. The result appears as the name of a register, shifter, which is actually the input to shift gates. Second, the shift function may change the position of the bits (see the ISP), yielding a 17-bit result. Third, the result may be read back into the accumulator specified by the destination of the instruction,  $A[d]$ . Fourth, the programming counter  $P$ , may be incremented an extra time (i.e.,  $P \leftarrow P+1$ ) causing the next instruction to be skipped.

## ADDITIONAL PROBLEMS

1. Carry out the ISP to RTM translation for the Cgr. What are the cost and performance for this implementation?
2. What are the cost and performance using a K(PCS)?
3. Add multiply and divide instructions to Cgr,
4. Show one tradeoff in your implementation.
5. Add console keys and switches to your design.

348

[previous](#) | [contents](#) | [next](#)

```

Run ⇒ (i ← Mp[P]; P ← P + 1; next
      addressable-instructions ⇒ (
      ib ⇒ (e ← M[e] * else e ← e); next
      (x = 00) ⇒ e ← disp;
      (x = 01) ⇒ e ← P + disp;
      (x = 10) ⇒ e ← A[2] + disp;
      (x = 11) ⇒ e ← A[3] + disp); next
      Instruction-execution ** )

```

fetch instruction  
indirect address  
determine effective address, e

\* This process can be an indefinite indirect loop as specified in M[x] and Mi[a]

\*\*Not Instruction-execution in ISP, since all M[e] should be changed to Mp[e] because the effective address, e, is known.

Fig. Cgr-2. ISP description of interpreter part for a general register based minicomputer.

and for fetching/storing data: e and M[x]. The effective address calculation process, e, is invoked as the operands are required in individual instructions, whereas the process of fetching/storing data in memory is called for by M[x]. As an alternative description for effective address calculation and instruction fetching, Figure Cgr-2 shows an interpreter that carries out the effective address calculation process prior to Instruction-execution. This interpreter is similar to that used in the RTM implementation of the PDP-8. By carrying out this change in representation which fixes the value of the effective address, e, rather than leaving it to be calculated as a variable when needed, it is easier to go from the ISP description to the RTM implementation. There is still a process, M[x], defined in the main ISP for fetching/storing memory with indirect addressing.

The instruction-set is given in the Instruction-execution process part of the ISP. This process initially defines the instructions that use addresses: LDA, STA, ISZ, DSZ, JMP, JSR and IO, which are essentially those of the PDP-8. These instructions load and store the general register array, A, and control the program flow. Finally, the Operate group instructions are given, which allow operations to be performed on the A registers using source A[s] and destination A[d]. Fundamentally four sequential operations are carried out in the operate instruction:

1.  $\text{shifter} \leftarrow f(A[s], A[d])$  arithmetic function
2.  $\text{result} \leftarrow g(\text{shifter})$  shifting function
3.  $A[d] \leftarrow \text{result}$  optional restoring or not
4.  $h(\text{result}) \rightarrow (P \leftarrow P+1);$  skip test function, causes P to be incremented

The process is also shown pictorially by looking at the structure of the registers

[previous](#) | [contents](#) | [next](#)

```

ADC( := f = 4) => (shifter<0:15> ← A[d] + ¬A[d]; add complement
(A[d] > A[s] {unsigned}) => (shifter<c> ← ¬ ci else
shifter<c> ← ci));
SUB( := f = 5) => (shifter<0:15> ← A[d] - A[s]; subtract
(A[d] ≥ A[s] {unsigned}) => (shifter<c> ← ¬ ci else
shifter<c> ← ci));
ADD( := f = 6) => (shifter<0:15> ← A[d] + A[s]; add
(A[d] + A[s] ≥ 216) => (shifter<c> ← ¬ ci else
shifter<c> ← ci));
AND( := f = 7) => (shifter ← ci □ (A[d] ∧ A[s])); and
next do rotate group
L( := sh = 1) => r ← shifter × 2 {rotate}; left rotate
R( := sh = 2) => r ← shifter / 2 {rotate}; right rotate
S( := sh = 3) => r ← shifter<c,8;15,0:7>; swap bytes
next
(n = 0) => (CF ← A[d] ← r; load or not result to A
do skip group
SKP( := sk = 1) => P ← P + 1; always
SZC( := sk = 2) ∧ shifter<c> => P ← P + 1; carry
SNC( := sk = 3) ∧ ¬ shifter<c> => P ← P + 1; not carry
SZR( := sk = 4) ∧ (r<0:15>=0) => P ← P + 1; zero
SNR( := sk = 5) ∧ (r<0:15>≠0) => P ← P + 1; non-zero
SEZ( := sk = 6) ∧ (r<0:15>=0) ∧ shifter<c> => P → P + 1; positive
SBN( := sk = 7) ∧ ((r<0:15>≠0) ∧ ¬ shifter<c> => P ← P + 1; negative
) end arithmetic and logical functions
) end instruction-execution

```

Fig. Cgr-1 (Part 4 of 4).

[previous](#) | [contents](#) | [next](#)

[previous](#) | [contents](#) | [next](#)

### Instruction interpreter

Run  $\Rightarrow$  ( $i \leftarrow Mp[P]$ ;  $P \leftarrow P + 1$ ; next  
Instruction-execution)

does not consider interrupt scheme

fetch  
execute

### Instruction set

Instruction-execution := (

LDA( := op = 1)  $\Rightarrow$   $A[d] \leftarrow M[e]$ ;

load accumulator

STA( := op = 2)  $\Rightarrow$   $M[e] \leftarrow A[s]$ ;

store accumulator

ISZ( := lop = 00010<sub>2</sub>)  $\Rightarrow$   $M[e] \leftarrow m[e] + 1$ ;

increment and skip if zero memory

$((M[e] + 1) = 0) \Rightarrow P \leftarrow P + 1$ ;

DSZ( := lop = 00011<sub>2</sub>)  $\Rightarrow$   $M[e] \leftarrow M[e] - 1$ ;

decrement and skip if zero memory

$((M[e] - 1) = 0) \Rightarrow P \leftarrow P + 1$ ;

JMP( := lop = 0)  $\Rightarrow$  ( $P \leftarrow e$ );

jump

JSR( := lop = 1)  $\Rightarrow$  ( $A[3] \leftarrow P$ ;  $P \leftarrow e$ );

jump to subroutine; store P in AC[3]

IO( := op = 3)  $\Rightarrow$  not defined here;

input-output

( $i < 0$ ) = 1)  $\Rightarrow$  (

arithmetic and logical functions -

the operate instruction -

do function group

COM( := f = 0)  $\Rightarrow$  shifter  $\leftarrow$  ci  $\square$  ( $\neg A[s]$ );

complement

NEG( := f = 1)  $\Rightarrow$  (shifter<0:15>  $\leftarrow$  - A[s];

negative

(A[s] = 0)  $\Rightarrow$  (shifter<c>  $\leftarrow$   $\neg$  ci else shifter<c>  $\leftarrow$  ci));

MOV( := f = 2)  $\Rightarrow$  shifter  $\leftarrow$  ci  $\square$  A[s];

move

INC( := f = 3)  $\Rightarrow$  (shifter<0:15>  $\leftarrow$  A[s] + 1;

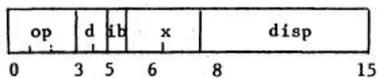
increment

(A[s] = 100000<sub>2</sub>)  $\Rightarrow$  (shifter<c>  $\leftarrow$   $\neg$  ci else shifter<c>  $\leftarrow$  ci));

Fig. Cgr-1 (Part 3 of 4).

345

[previous](#) | [contents](#) | [next](#)



```

ib := i<5>
x<0:1> := i<6:7>
disp<0:7> := i<8:15>
carry input\ci:=(carry=0) => CF;
                (carry=1) => 0;
                (carry=2) => ~CF;
                (carry=3) => 1)
    
```

Effective address calculation

```

e := (~ib => ei;
      ib => M[ei])

ei := ((x=00) => disp;
       (x=01) => P + disp;
       (x=10) => A[2] + disp;
       (x=11) => A[3] + disp)
    
```

```

M[x] := ((20<=x<27)&=>(Mp[x]+Mp[x]+1; next Mi[x]); main access process; auto decrement
         (30<=x<37)&=>(Mp[x]+Mp[x]-1; next Mi[x]); auto increment
         ~ (20<=x<37)&=> Mi[x]) direct
    
```

```

Mi[a] : (~M[a]<0> => Mp[a];
         M[a]<0> => M[Mp[a]<1:15>]) indirect word calculation
                                             fetches an address
    
```

Addressable instructions

- indirect bit
- index register selection
- long displacement addresses
- carry result input value

first effective address as calculated by instruction

- absolute
- relative
- indexed via A[2] or A[3]

Fig. Cgr-1 (Part 2 of 4).

Fig. Cgr-1 (Part 1 of 4). ISP description of a general register based minicomputer.

Processor State

P<0:15>  
 A[0:3]<0:15>  
 Carry\CF

Primary Memory

Mp[0:32767+10]<0:15>

Temporary registers used in description

shifter<c,0:15>  
 result<c,0:15>\r

Instruction format

i<0:15>  
 op<0:2> := i<0:2>  
 lop<0:4> := i<0:4>

l	s	d	f	sh	carry	n	sk
0	1	3	5	8	10	12	15

acs<0:1>\s := i<1:2>  
 acd<0:1>\d := i<3:4>  
 function\f<0:3> := i<5:7>  
 shift<0:1>\sh := i<8:9>  
 carry<0:1> := i<10:11>  
 no-load\n := i<12>  
 sk/skip<0:2> := i<13:15>

Program counter

Accumulators

physical memory

temporary data used in description  
 (and machine)

instruction bits

op code

long (extended) op code

Operate instruction

source accumulator specification

destination accumulator

controls operate instruction

shift control of operate

carry input of operate

whether result is retained

operate skip test

a DMgpa is used for updating the clock A clock count flag is set each 10 microseconds by the master clock, and the control part senses the state of the flag and then increments C[1:3] by 1.

The computer transmits a control word to the clock which signifies which of the two functions are to be performed. For resetting the clock, a non-zero word is transmitted to the clock, signifying that the next three values will be transmitted to reset C[1:3]. When this occurs, the sequence to reset the clock is invoked, and clock counting is stopped until all three words are placed in the clock registers.

A zero word is transmitted from the computer to the clock-calendar signifying that the three values of C are to be transmitted back. Unlike the case of resetting C, it is necessary for the clock to continue while the read-in process occurs. Also, the clock must not change during this process, since a carry might be generated and cause subsequent words to change. Note that adding a value to C is just a triple word addition by 1. While transmitting C[1:3] to the computer, counting is carried out in an intermediate register, and after C[1:3] is transmitted, the intermediate counts are added back to C to update it. (Alternatively, the clock might have transmitted C[3] to the computer on command and then continued to count C[3] while only keeping an overflow from C[3] for eventual update of C[1:2].)

## Problem

1. Design the mixed-base system, which provides time units that can be read directly by humans.

## A GENERAL REGISTER BASED MINICOMPUTER IMPLEMENTATION

**KEYWORDS:** General register, minicomputer, ISP

Cgr, a stored program minicomputer with multiple general purpose registers (i.e., Accumulators) is presented using ISP to give the reader an idea of another type of computer.(4) The implementation is similar in size to the DEC PDP-8 implementation.

## ISP DESCRIPTION

The implementation is to be based on the Instruction-set definition in Figure Cgr-1, given in ISP.

The processor state memory is five words + 2 bits: P<0:15>, A[0:3]<0:15>, Run and Carry-flag. The physical program primary memory is  $2^{15}$  words, declared as Mp[0:32767v10]<0:15>. Two 17-bit intermediate result registers, shifter and result, are needed in the description to correspond to outputs of Data operations in the actual machine.

Two instruction formats are given: the operate instruction that specifies the operations to be carried out

on the four Accumulator\A registers; and instructions which address operands for loading and storing the A registers and for modifying memory (i.e., incrementing and decrementing by 1). The addresses also specify locations for loading the program counter register\PC, for transferring control, and for calling subroutines.

There are two main processes for calculating addresses

-----

4. This machine is similar to the DEC PDP-11, but substantially simpler pedagogically. The PDP-11 manuals include ISP descriptions.

342

[previous](#) | [contents](#) | [next](#)



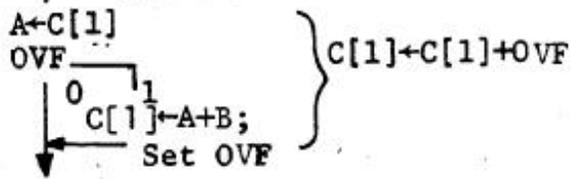


Fig. CC-1. RTM diagram of Clock-calendar interfaced to a computer using T(program controlled access).

## Problems

1. Assuming event inputs of the form illustrated by the flow meter, design a special system which counts these events into a computer's memory.
  - a. When an overflow of a channel count occurs (i.e., the number is greater than  $2^{16}-1$ ) interrupt the computer, giving the channel number.
  - b. Use two memory locations to store double precision counts.
2. In the above design allow another set of inputs from incremental shaft position. The encoders present counts of + 1 and -1 together with reference positions (e.g., 0).
3. Further modify the design of the special interface to control the movement of a set of stepping motors of the type given above. The desired incremental positions are given in a table in the computer's memory, and the function of the special processor is to make certain that the correct step commands are issued. (Note, a stepping motor of this type will supply return information as- to when it has completed a step process.)

## CLOCK-CALENDAR INTERFACED TO A COMPUTER

Design a clock-calendar, interfaced to a computer, using the T(program controlled) interface which is able to perform the following functions:

- 1. be reset under control of the computer to a specific time;
- 2. respond to queries for the time.

Assume a master clock of 100,000 KHz. Also, assume that the calender-clock is to operate for several years (e.g., 10) without resetting. There are at least two alternatives for representing time:

- 1. The clock can count in binary.
- 2. The clock can count in suitable (human) units: years, months, days, hours, minutes, seconds, fractional seconds.

The first alternative has simpler clock hardware and provides for simple operation (e.g., comparison, addition) in the computer. The second alternative is needed for human output. Also, in this latter case, the algorithm, is needed, in hardware, to deal with the calendar (leap years, and months). The length of the

clock would be some multiple of 16-bit words. The following bits would be needed for the mixed base representation:

- o 4 (10 years) + 4 (12 months) + 5 (31 days) + 5 (24 hours) +
- 6 (60 minutes) + 6 (60 seconds) + 10 (1000 milliseconds) +
- 7 (100 10-microseconds) = 47 bits or 3 16-bit words.

Actually since there are only

$$12 \times 31 \times 24 \times 60 \times 10^3 \times 10^2 = 3.19 \times 10^{12} \text{ 10-microseconds/year}$$

and  $2^{48} \approx 2.56 \times 10^{12}$ , then the 48-bit clock can count for 80 years without intervention in the full binary representation.

The structure of the clock for the binary alternative is given in Figure CC-1. The data part consists of the two systems: the minicomputer and the RTM Bus. - They are linked together by the full duplex T(program controlled) interface. The data part of the RTM system holds the three words of the clock, C[1:3], and

a table pointed to by the value (address) in memory location k2. Increase k2.

The behavior of the processor consists of the familiar fetch-execute cycle. The processor normally waits until the computer sets the start-flag, before proceeding. The instruction-pointer is picked up from memory location k1. The instruction-pointer is then used to address various instructions. Each instruction is then fetched, decoded, and executed. The 4 instruction types have very simple actions, as defined in the flowcharts.

When a value is found to be out of range, the scan address and the out of range value are placed in a list which is addressed by memory location k2. This location is continuously updated. At the completion of the process, the Halt instruction sets the Done-flag which signals the computer that the Scan cycle has been carried out.

## Problems

1. What is the maximum scanning rate, assuming all channels are within range, and all use the same range?
2. Modify the design to place the sampled value in the same 16-bit word as the channel number.

## SPECIALIZED PROCESSORS FOR EVENT COUNTING

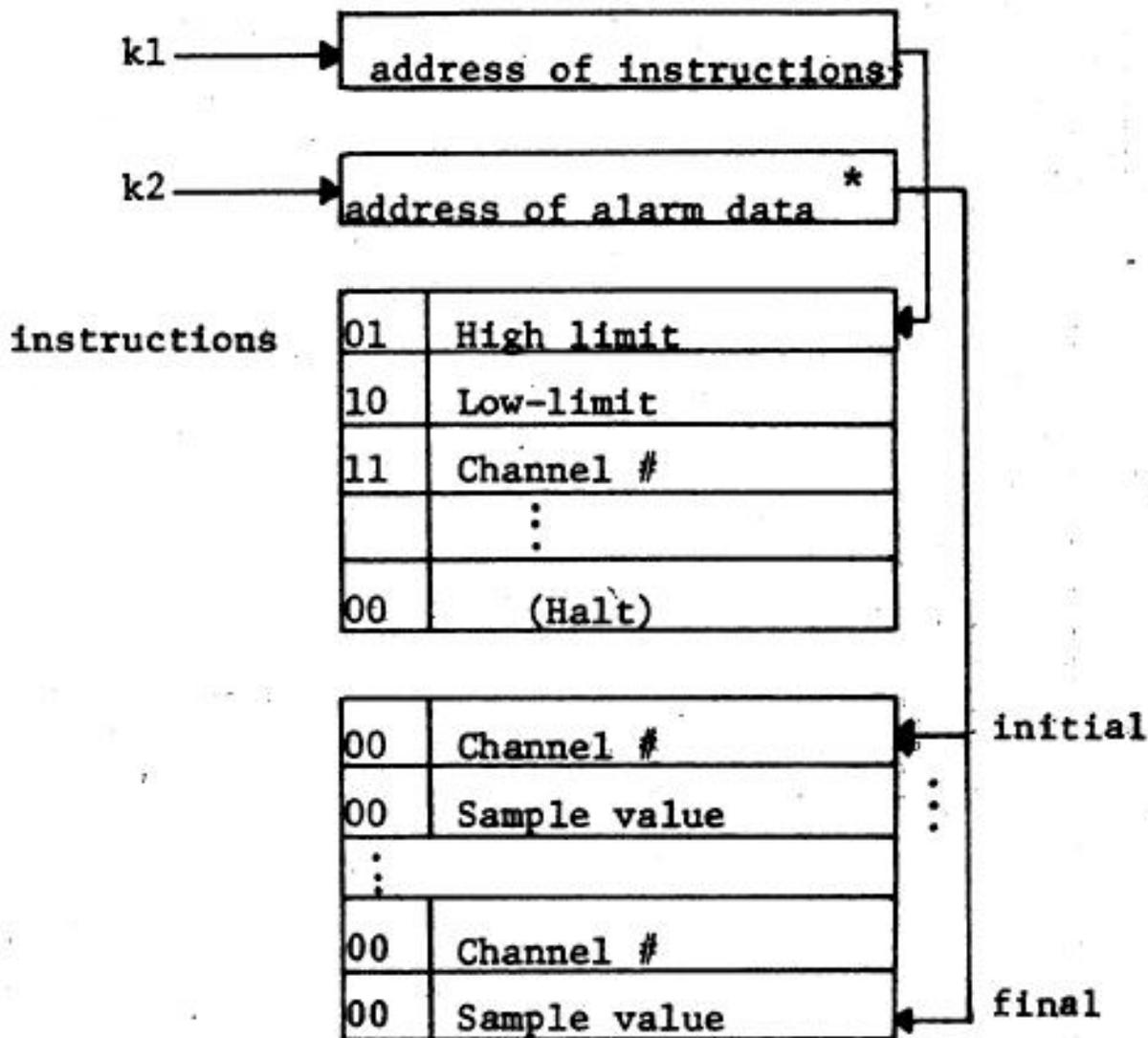
There are systems related to the EPUT meter described in Chapter 5 that are used in process control and communication. Event counting systems are often connected to a computer because other processing is carried out within the computer, including closed-loop control. The purpose of an RTM system used as an adjunct to a computer would be to lighten the processing load associated with the basically trivial, but time consuming, operation of counting. As we indicated in the EPUT meter example of Chapter 5, a minicomputer dedicated to counting could only handle 1/3 the counting rate of the RTM system.

Two types of event input signals exist in process control. First, uni-directional inputs count in one direction (i.e., only increment) and are encoded as either events or as pulse widths. Flow meters for gasses and liquids are typical devices. Each pulse output from the flow meter indicates an incremental movement of the meter, hence an incremental quantity of material has flowed. By integrating the output of the meter (i.e., counting the pulse outputs from it) the flow is recorded.

The second similar, but bi-directional, counting system of this type is the incremental stepping motor which controls either a rotational or linear position of a mechanism. For example, a stepping motor is given a command to move either forward or reverse. The motor itself has no output that can be used to detect the position of the mechanism being moved. There is usually some other part of the system which provides this feedback as to the actual position of the mechanism. Often, however, no incremental or

absolute feedback is used. The feedback as to the gross position for calibration is obtained from two limit reference switches indicating when the motor is in the extreme position.

Alternatively, there may be direct coupling of an input transducer to give the actual position (using an optical or magnetic position encoder, for example). This type of indicator has an output that indicates movement of +1 or -1 position, together with information about the absolute (limit) position.



\* Continuously points to next memory cell free

Fig. Ps-2. Memory and instruction formats for special processor.

list of instructions specifying input channels to sample, and limit values for the samples. Figure Ps-2 gives the format of how the instructions are interpreted, together with the structure of a program for Ps. The instruction table can contain any one of four instruction types as specified by bits <15:14>:

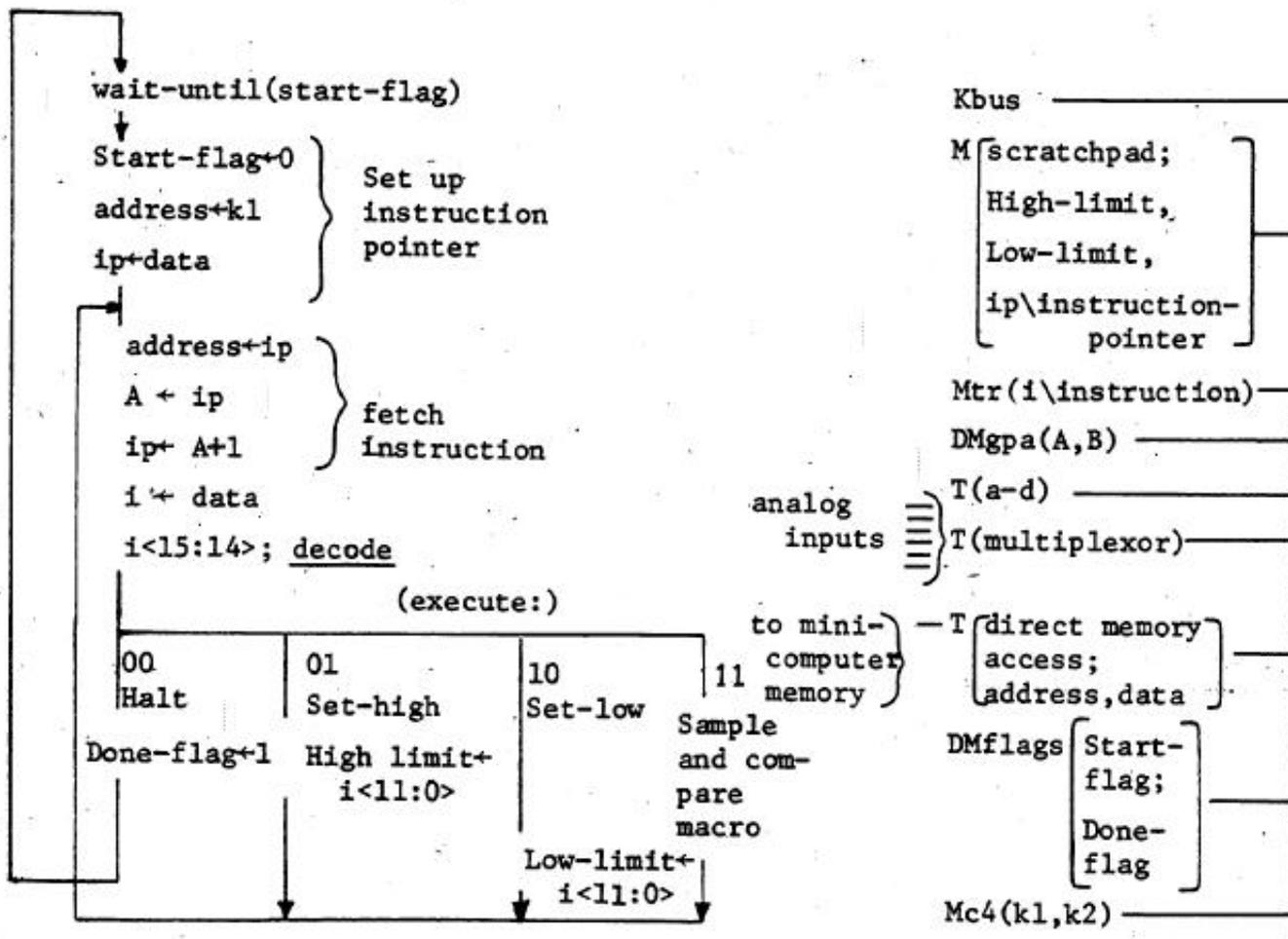
00 - Halt. Halt the comparison process, signal the computer by setting a Done-flag and then wait until the processor is reinitiated by the computer.

01 - Set high. Take value in bits <11:0> as a high limit value; incoming results will be

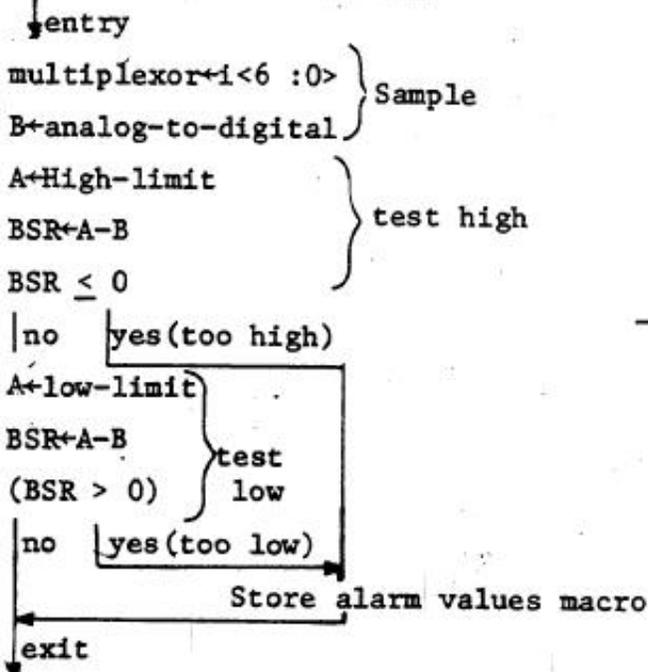
compared against this  
high value.

11 - Set Low. Take value in bits <11:0> as a low limit value: incoming results will be compared against low value.

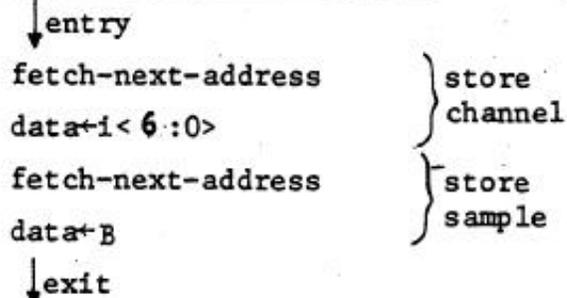
10 - Sample and Compare. Take bits <6:0> as line number, set up- an analog multiplexor (switch address), sample the data and compare against the high and low values. When an incoming value is tested and found to be out of range, place it and its address (channel) in



Sample-and-compare macro



Store-alarm-values macro



fetch-next-address subroutine

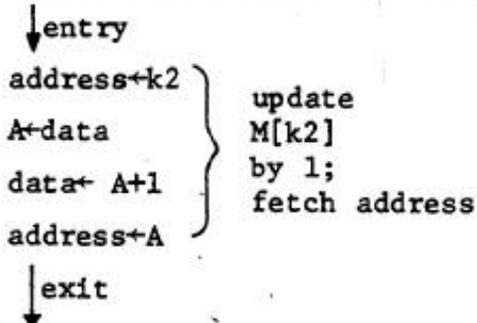


Fig. Ps-1. RTM diagram for a special processor to sample data and compare

**Fig. Ps-1. RTM diagram for a special processor to sample data and compare against limits.**

337

[previous](#) | [contents](#) | [next](#)



\* Mode specifies Reading or Writing. 0 specifies computer output (i.e. RTM read); 1 specifies computer input (i.e. RTM write)

Fig. CI-3. RTM diagram of RTM-minicomputer interface for direct memory access to computer - the T(direct memory access).

## Transfer of Data from the RTM System to the Computer

Figure C1-2c shows the flowchart of the RTM system for transmission to the minicomputer. The process is:

- 1. The RTM system first checks to see that the Outbuff is empty by waiting until the Outbuff-full flag is 0.
- 2. The RTM system places a word in Outbuff and sets the Outbuff-full flag to 1.
- 3. The Outbuff-full flag causes an interrupt to the minicomputer.
- 4. The minicomputer reads the Outbuff indirectly via its C-input register and resets the Outbuff-full flag to 0, denoting that another word can be transmitted from the RTM system.

Note that a flowchart for the input and output processes in the minicomputer would be the opposite of those in the RTM system.

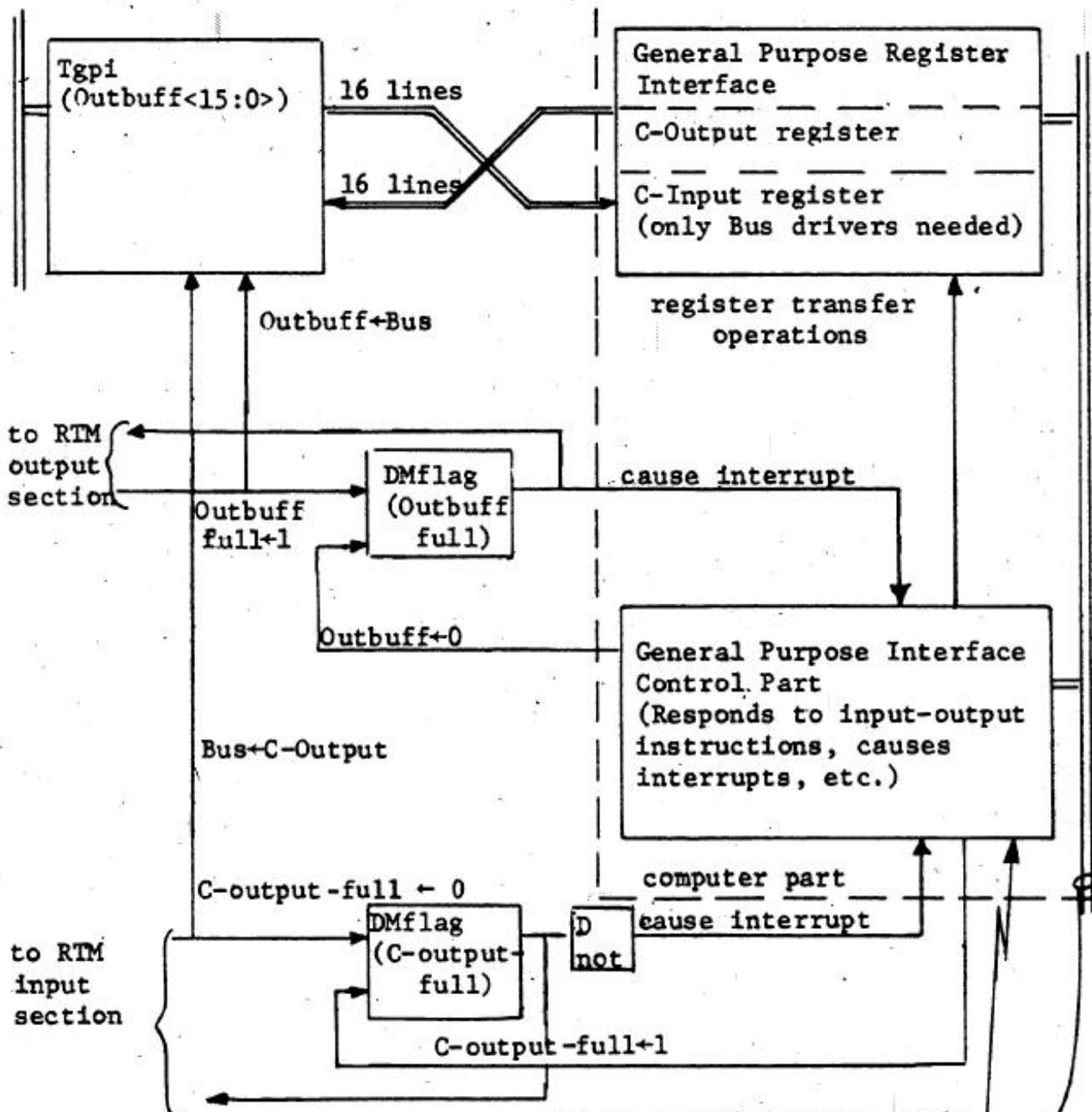
## RTM-MINICOMPUTER INTERFACE VIA DIRECT MEMORY ACCESS TO COMPUTER

The structure of an interface which can be used by an RTM system to access directly the minicomputer's memory is given in Figure C1-3a. The interface provides a capability that is essentially identical to that of the RTM system, i.e., having a large random access memory directly connected to its Bus. For this interface, standard minicomputer interface modules are not available directly; thus we simply postulate the registers and main functions. The control part of this interface is quite simple. The behavior of the interface is almost identical to the program controlled transfers except that hardware (within the minicomputer processor) controls the transfers instead of a program interpreted by the processor. Also, unlike the program controlled case, this channel is half-duplex; it can either transfer data to or from the computer's memory, but the transfers cannot be carried out simultaneously. This interface design is somewhat different from random access memories, even though the computer provides an identical function, because a flag signifies when the transfer is complete (DC-request). By examining the DC-request condition the RTM system can carry out several steps while waiting for service since the computer may take several microseconds to respond with the data. The two macro processes for reading and writing are given in Figure C1-3b and c. With the exception of the Kwait's, the behavior is identical to a random access memory.

## Ps, A SPECIAL PROCESSOR TO SAMPLE ANALOG INPUT DATA AND COMPARE AGAINST LIMITS

Figure Ps-1 shows the structure and behavior of a special processor interfaced to a computer, which

accepts analog data and checks the data against variable limits. The processor operates completely in parallel with a program in the computer, and the only interference is when the process accesses the computer's memory using the T(direct memory access). The structure of this hardwired process is similar to that of the hardwired processors in an IBM 1800. The process samples analog data, compares the data against high and low limits in a table, and signals when the out of limit condition occurs for each data point. The structure assumes that data is input-via an analog-to-digital converter with multiple selectable input channels. The data part also shows the various registers which the processor uses. The processor operates in much the same way as the processor of a stored program computer. There is a location in memory, k1, which contains an address of a data table That is interpreted as a



a. Interface structure

b. RTM+Computer macro

$\downarrow$   
 Kwait (C-Output-full)  
 Ke (Bus+C-Output-register;  
 C-output-full+0)  
 $\downarrow$  (causes interrupt)

c. Computer+RTM macro

$\downarrow$   
 Kwait ( Outbuff-full)  
 Ke (Outbuff+Bus; Outbuff  
 full+1)  
 $\downarrow$  (causes  
 interrupt)

**Fig. CI-2. RTM diagram of RTM-minicomputer interface for program controlled access - the T(program controlled access).**

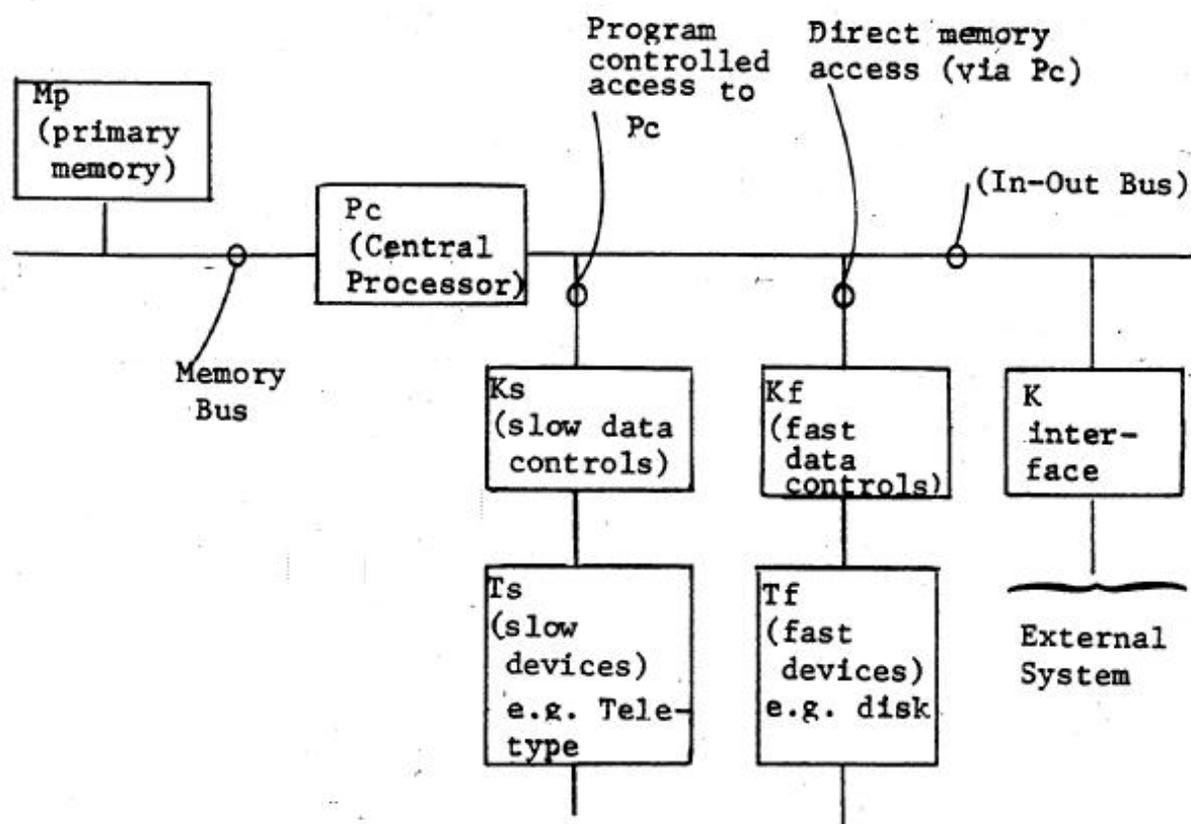


Fig. C1-1. PMS diagram of a simple minicomputer.

General Purpose Interface Control Part which determines input-output instructions and transmits status and interrupt requests to the computer.

A T(general purpose interface) RTM module is cross-coupled to the minicomputer data interface providing the two simultaneous paths for data (i.e., full duplex). The RTM part of the interface has only two DM(f lags): Outbuff-full, indicating that a word is in the RTM Outbuff register and is waiting to be taken by the computer; and C-output-full, indicating that a word is in the computer's Output register and waiting to be taken by the RTM system. Note that the C- input register is not really needed because Outbuff has the same data. Both of these flags cause interrupts to the minicomputer (i.e., when Outbuff-full = 1; and when C-output-full = 0).

### Transfer of Data from the Computer to the RTM System

Figure C1-2b shows the flowchart for the case of the computer transferring data to the RTM system. The process is:

1. The computer loads data into its C-output-register and sets the C-output- full flag to 1.

2. When the C-output-full is 1, the RTM control flow proceeds past the waiting loop (i.e., Kwait).
3. The RTM system takes the word from the computer's output register via the T(general purpose interface) input section and resets the C-output-flag to 0, interrupting the minicomputer.
4. The minicomputer either decides to ignore the interrupt or places another word in its Output-register.

## PROBLEMS

1. For the example program given above, show a possible pattern of bits in the K(PCS) program memory, assuming a particular assignment of operation codes (see Chapter 2).
2. Write the BCD-to-Binary and Binary-to-BCD conversion subroutines required in the example program.
3. Write a program in the 16/M so that a program held in a M(array; 1024 words) is interpreted the same way that Crtm-1 would interpret it. Modify 16/M to improve the ability to interpret Crtm-1.
4. Try problem 3 for a PDP-8 or your favorite minicomputer.
5. Solve any of the application problems given in other sections. Compare the conventional RTM, K(PCS), and 16/M cost and performance (recall the analyses in Chapter 4). 6. Respecify the 16/M in ISP.

## THE RTM-COMPUTER INTERFACES

An RTM system can be interconnected to a general purpose computer in many ways, depending on the system objectives (e.g., cost and performance) and the computer being interfaced. No doubt the simplest interface is the T(serial), which only involves connecting two pairs of wires between the computer and RTM system, providing full duplex data transmission at 10,000 bits/second. Two other interfaces will be described which provide closer coupling and higher data transmission rates. A particular minicomputer is used to illustrate the transmission scheme. The structure of the simple minicomputer is shown in Figure CI-1. All external devices are connected to the minicomputer by means of an In- Out Bus. This bus has two types of controllers:

**Ks** - Slow data transfer controllers - (Program controlled transfers) - Data transmission is accomplished by a program in Pc. Ks may request words by signalling using an interrupt request.

**Kf** - Fast data transfer controllers - (Direct memory access transfers) - Data transmission is accomplished by having Kf request words. Pc merely passes requests for data on to primary memory (Mp). In this mode the system looks like a random access memory to Kf.

In building an interface between RTM's and the minicomputer, the method used for transfer mainly depends on the needs of the RTM system for data. With minicomputers the programmed transfer method can be used for word transfer rates of perhaps 100,000 words/second, although this method is normally used at only 10,000 words/second because of the high processor utilization. The direct memory access methods are used for magnetic tapes and disks at speeds varying from 10,000 ~500,000 words/second.

In order to reduce processing time substantially the direct memory access method is usually used, and in some designs both interface methods are required. Both interface methods are presented in the following sections.

## RTM MINICOMPUTER INTERFACE VIA PROGRAM CONTROLLED ACCESS

Figure C1-2 gives most of the details for an interface between an RTM system and a minicomputer. The interface is full duplex in that data can be flowing quasi-simultaneously in both directions. The part of the interface nearest the minicomputer is constructed from two of the minicomputer's modules: a General Purpose Register Interface containing both an input and output register; and a

which holds the character being transmitted, and a Punch Flag, PF, which indicates when a new character can be loaded into T. The instructions for the transmitter part are:

<b>Slj←A</b>	<b>load the transmitter register Tj of serial interfacing from A&lt;7:0&gt;, clear PF (j=1,2)</b>
<b>IF PF, &lt;label&gt;</b>	<b>if the Punch Flag, PF, is true, then GOTO location called &lt;label&gt; (i.e, PC is set to address &lt;label&gt;)</b>

The receiver section contains an 8-bit register, <7:0> which holds the character while it is being received from a keyboard or the attached paper tape reader. The keyboard flag, KF, indicates when a new character has arrived in R. The instructions for the receiver section are:

<b>A←Slj</b>	<b>load the receiver register of serial-interfacing into A, clear KF (j=1,2)</b>
<b>TAPEj</b>	<b>start the paper tape reader and have it read 1 character</b>
<b>IF KF, &lt;label&gt;</b>	<b>if the Keyboard Flag, KF, is true, then go to location &lt;label&gt;</b>

#### PROGRAM EXAMPLE

This program waits until an external input is, i, it takes a number, 1, in BCD, sums the integers up to the input value, then outputs the sum, S, in BCD, where  $S=0+1+2+ \dots+i$ .

<b>SENSE,</b>	<b>IF EXT1 START</b>	<b>;WAIT FOR EXT1 INPUT ;BEFORE STARTING</b>
	<b>GOTO SENSE</b>	
<b>START,</b>	<b>A←GPI1</b>	<b>;TAKE IN DATA, IN BCD, ON INPUT 1</b>
	<b>CALL BCD-BIN</b>	<b>;CONVERT TO BINARY USING SUBROUTINE</b>
	<b>B←0</b>	<b>;BEGIN PROGRAM TO SUM INTEGERS TO INPUT</b>
<b>LOOP,</b>	<b>B←A+B</b>	
	<b>A←A-1</b>	
	<b>IF DN LOOP</b>	<b>;LOOP IF A IS NEGATIVE</b>
	<b>A←B</b>	<b>;END OF PROGRAM TO SUM INTEGERS</b>
	<b>CALL BIN-BCD</b>	<b>;CONVERT TO BCD USING SUBROUTINE</b>
	<b>GPI1←A</b>	<b>;OUTPUT</b>
<b>BCD-BIN,</b>		<b>;BCD TO BINARY SUB</b>

EXIT

BIN-BCD

;BINARY TO BCD SUB

EXIT

331

[previous](#) | [contents](#) | [next](#)

M(byte). The Byte Register BR<15:0> is a 16-bit register used as a temporary register. Various parts of it can be loaded into A.

<b>BR←A</b>	<b>load BR with A</b>
<b>A←BR</b>	<b>load A with BR</b>
<b>A←BR&lt;3:0&gt;</b>	<b>load A&lt;3:0&gt; with BR&lt;3:0&gt;, load A&lt;15:4&gt; with 0</b>
<b>A←BR&lt;7:0&gt;</b>	<b>load A&lt;7:0&gt; with BR&lt;7:0&gt;, load A&lt;15:8&gt; with 0</b>
<b>A←BR&lt;11:0&gt;</b>	<b>load A&lt;11:0&gt; with BR&lt;11:0&gt;, load A&lt;15:8&gt; with 0</b>

M(constants; 4 word, 24 word)\*. The Mc4 and Mc24 have access times of 0.2 and 1.0 microseconds, respectively.

<b>B←Cj</b>	<b>read constant j into B, where j=1,2,3 or 4 for Mc4</b>
<b>B←Kj</b>	<b>read constant j into B, where j=1,2,...,24 for Mc24</b>

M(scratchpad) 1\*, 2\*. The 16-word scratch pad is an optional read-write memory which is addressed in an explicit fashion as though each word were an independent register. The two scratch pads are denoted SP[1:16] and SP[17:32]. There are 16 instructions to read and 16 instructions to write each word of the 16 word scratch pad.

<b>SPj←A</b>	<b>load scratchpad register j with A</b>
<b>a←SPj</b>	<b>load A with scratchpad register j</b>
	<b>(j=1,...,16) for 1</b>
	<b>(j=17,...,32) for 2</b>

M(arrays). There are three memory arrays of 256, 256 and 1024 words. The cycle time of each memory is roughly two microseconds. They all operate in exactly the same way. Each array has a memory address register (MAR) which holds the address of a word being accessed. After MAR is loaded the word (specified by MAR) can be either read from A or written into A.

M(array; 256 words)\*.

**MAR1←A**            load the memory address of the first 256  
                          word memory with least significant 8 bits of A  
**A←MEM1**            read the word (data) accessed by MAR1 into A  
**MEM1←A**            write the word (data) accessed by MAR1 from A

M(array; 256 words)\*. The instructions: MAR2←A; A←MEM2, and MEM2←A are added.

M(array; 1024 words)\*. The instructions: MAR1K←A; A←MEM1K and MEM1K←A are added.

### Input-Output Instructions

T(general purpose interface) 1, 2, and 3\*.

**GPIj←A**            load the jth output register, GP from A  
**A←GPIj**            read in the jth input to A where j=1,2, or 3

T(serial interface) 1\*, 2\*. The transmitter part contains an 8-bit register, T<7:0>.

**MUX1**

set MUX to select the second group (MUX←1)

The <condition> input Boolean mnemonics of the first group, and main group (MUX0) are:

DZ	(BSR=0), result in Bus Sense zero
DP	(BSR>0), result in Bus Sense positive
DN	(BSR<0), result in Bus Sense negative
OVF	OVF is a one
A<1>	register A, bit 1, is a one
A<3>,A<5>,...,A<15>	other bits of A
FFj	Boolean flag (Flip-Flop) j is a one (j=1,2,...,6)
EXTj	external condition j is a one (j=1,2,...,6)
KFj	optional Keyboard Flag, KF, is one (j=1,2)
PFj	optional Punch Flag, PF, is one (j=1,2)
L	Link bit

The <condition> input mnemonics of the second, optional group (MUX 1), which give additional capability, are:

A<0>,...,A<14>,B<15>,B<0>	other bits of A, and B
EXTj	external condition j is one, (j=7,...,22)

The subroutine CALL and EXIT instructions allow a subroutine to be called and terminated (exited) respectively. The CALL instruction is two bytes, and EXIT one byte. They are specified:

<b>CALL&lt;label&gt;</b>	go to the subroutine addressed by the address <label> and place PC in the top register of the subroutine stack, pushing the registers of the stack down one position
--------------------------	--

<b>EXIT</b>	return from the current subroutine by placing the top register of the stack into the PC and popping the registers of the stack up one position
-------------	--

## Memory Instructions

DM(flag) FFI,FF2,FF3,FF4\*,FF5\*,FF6\*. The flag bits can be reset to 0 or set to 1 under program

control. The conditional jump instructions (above) also permit the flags to be tested and used to control branching.

$FF_j \leftarrow 0$	clear (reset) the $j$ th flag
$FF_j \leftarrow 1$	set the $j$ th flag

M(transfer). Although normally wired to perform a particular bit transformation, the M(transfer) can be wired for other uses. The normal instructions are:

$TR \leftarrow A$	load TR with A
$A \leftarrow TR$	load A with TR
$A \leftarrow TRU$	load $A\langle 15:8 \rangle$ with $TR\langle 15:8 \rangle$ , load $A\langle 7:0 \rangle$ with 0
$A \leftarrow TRL$	load $A\langle 7:0 \rangle$ with $TR\langle 7:0 \rangle$ , load $A\langle 15:8 \rangle$ with 0

<b>*B←A+1</b>	increment A, store in B
<b>*B←A+1(S)</b>	increment A, set OVF, store in B
<b>*B←A-1</b>	decrement A, store in B
<b>*B←A-1(S)</b>	decrement A, set OVF, store in B
<b>B←A+B</b>	add A to B
<b>B←A+B(S)</b>	add A to B, set OVF
<b>B←A-B</b>	subtract B from A
<b>B←A-B(S)</b>	subtract B from A, set OVF
<b>B←AXORB</b>	exclusive-or A to B
<b>B←AORB</b>	inclusive-or A to B
<b>B←AB</b>	and A to B
<b>B←NOTB</b>	negate B
<b>B←B/2</b>	rotate B and Link right
<b>B←B/2(S)</b>	rotate B and Link right, set OVF
<b>*B←AX2</b>	rotate A and Link left, store in B
<b>*B←AX2(S)</b>	rotate A and Link left, set OVF, store in B
<b>*B←A/2</b>	rotate A and Link right, store in B
<b>*B←A/2(S)</b>	rotate A and Link right, set OVF, store in B

**Instructions for the shift Link-bit:**

<b>L←0</b>	set Link to 0	<b>L←1</b>	set Link to 1
<b>L←OVF</b>	set Link with Overflow	<b>L←NOTL</b>	complement Link

**Instructions for the Bus Sense Register(BSR):**

<b>BS←A</b>	load BSR with A	<b>BS←B</b>	load BSR with B
-------------	-----------------	-------------	-----------------

**Control Instructions**

A special set of instructions controls the program flow. All instructions may affect the Program Counter\PC. The instructions are: unconditional branch (i.e., set PC) - the GOTO instruction; conditional branch (conditionally set PC) - the IF instruction for testing Boolean (bit) conditions; CALL subroutine; and EXIT from subroutine. The instructions which load the PC with a new address are two bytes long. Two byte instructions take 3.0 microseconds; and one byte instructions take 1.6 microseconds. The unconditional and conditional branch instructions are:

<b>GOTO &lt;label&gt;</b>	places the address <label> in the PC
<b>IF &lt;condition&gt;, &lt;label&gt;</b>	places the address <label> in the PC if Boolean (bit) <condition> selected by IF instruction is true; otherwise skip over<label>

There are two groups of Boolean input variables which can be tested by the above IF instruction. In order to specify which of the two groups is being tested, the MUX-bit (which is either 0 or 1) must be set properly to access either the first or second group, respectively. The first group is standard with the 16/M; the second group is optional. The MUX instructions are:

<b>MUX0</b>	set the MUX register for the selection of the first group of inputs (MUX←0)
-------------	---

328

[previous](#) | [contents](#) | [next](#)

Panel Row Number

D		C		B	A	Slot Number	
SPARE		SPARE		SPARE	SPARE		1
SPARE		SPARE			KBM16		2
KFL16 *		KEV16 *			KBS16		3
	PCS16-C				MS16-C *		4
	PCS16-C				MS16-C *		5
	PCS16-C				MR16A or MR16-D *		6
	PCS16-C *				MS16-D *		7
	PCS16-C *				MS16-D		8
	PCS16-C *				MS16-E *		9
	PCS16-A *				MS16-A		10
KFL16 *		KFL16 *			MS16-B		11
KOR16-B		KOR16-A			KAR16		12
KOR16-B		KOR16-B			KAC16		13
KOR16-B		KOR16-B			DB16-A		14
KOR16-B		KOR16-B			DB16-A		15
PCS16-B *		PCS16-B			DB16-A *		16
PSS16-B *		PCS16-B *			DC16-A		17
SERIAL ADAPT. *		KOR16-B			DC16-A *		18
PAR I/O Chan. 1		I/O MUX			PCS16-D		19
PAR I/O Chan. 3		PAR I/O Chan. 2			PCS16-D *	20	

\*Optional parts

Fig. 16m-3. Table of physical assignment of modules in the PDP-16/M.

[previous](#) | [contents](#) | [next](#)

The Arithmetic-Logic, Memory, and Input-Output parts respectively operate on, hold, and transfer outside the computer 16-bit data. Data is transferred among these parts via the RTM Bus, into which all the parts are connected.

## PHYSICAL ORGANIZATION

All of the registers accessible to the program (programmer) are given in Figure 16m-2. The components in the diagram correspond to the functional components in the standard PDP-16 set (i.e., RTM's). The optional modules and operations are indicated by asterisks. The actual physical layout of the computer with its modules is given in Figure 16m-3. Module locations are reserved and prewired for the basic machine plus the various options.

## INSTRUCTION SET

The instruction set for the PDP-16/M is taken from the basic set available from the K(PCS) and those instructions permitted by the specific DM, M, and T modules used. The set will be described in the order: Arithmetic-Logical (DM), Control (K), Memory (M), and Input-Output (T) parts.

The instruction-set is described using the assembler format of DEC. This, notation is slightly different from that of ISP used elsewhere in the chapter.

### Arithmetic and Logical Instructions

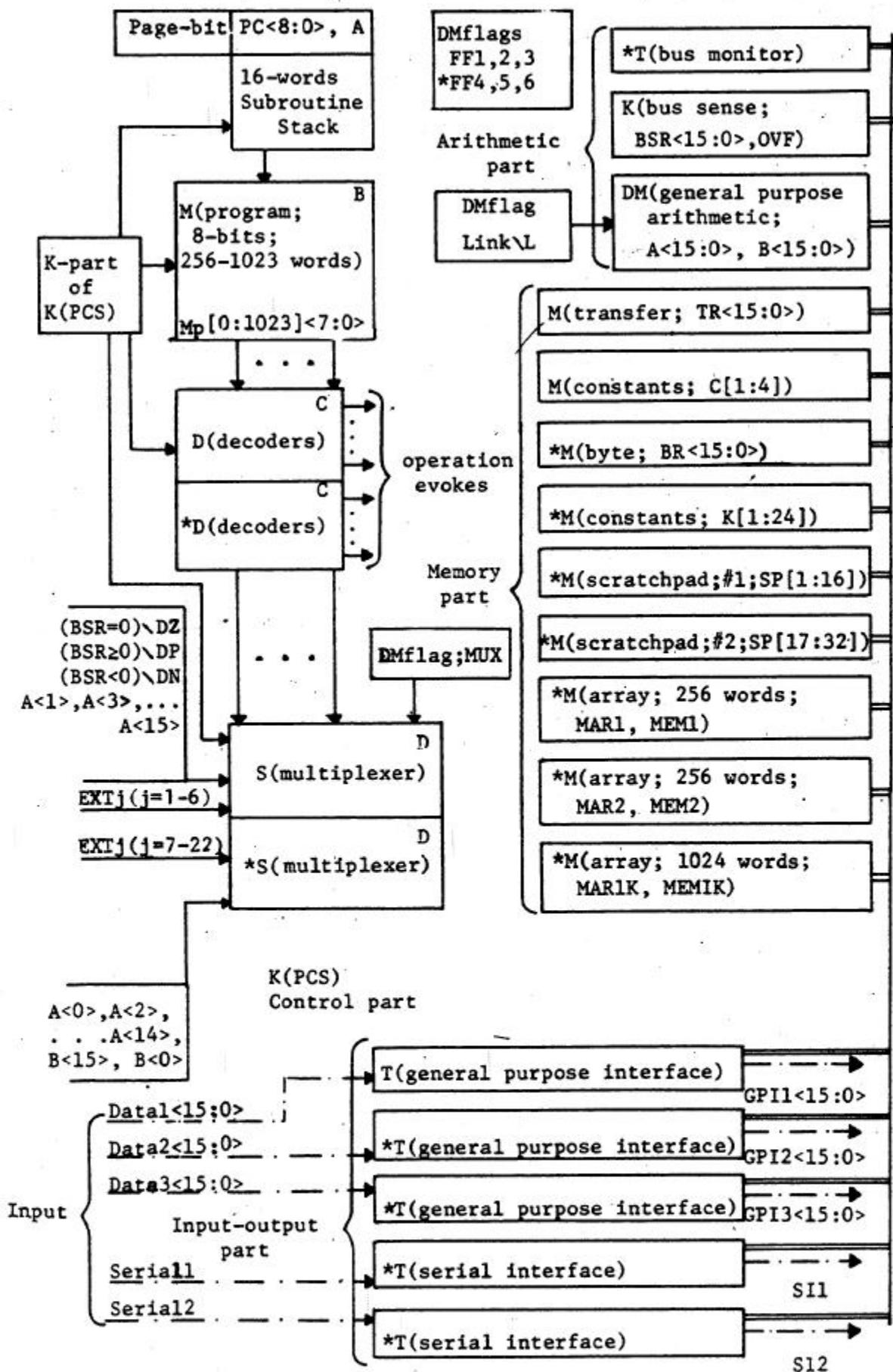
The following instructions are specified by a 1 byte operation code, and operate on a DMgpa and Kbus. Each instruction takes 2.1 microseconds. The instructions use the A and B registers of the DMgpa as arithmetic and logical operands, and operation results are placed in either A or B. Other instructions use and affect the Bus Sense Register\BSR\BS, Overflow-bit\OVF and a DMflag which holds the shift LINK bit\L

**Instructions for A(arithmetic-logical):**

$A \leftarrow 0$	clear A (transfer a 0 to A)
$A \leftarrow B$	store B in A
$A \leftarrow A+1$	increment A
$A \leftarrow A+1(S)$	increment A, set OVF
$A \leftarrow A-1$	decrement A
$A \leftarrow A-1(S)$	decrement A, set OVF
$A \leftarrow A+B$	add B to A
$A \leftarrow A+B(S)$	add B to A, Set OVF
$A \leftarrow A-B$	subtract B from A
$A \leftarrow A-B(S)$	subtract B from A, set OVF
$A \leftarrow A \oplus B$	exclusive-or B to A
$A \leftarrow A \vee B$	inclusive-or B to A
$A \leftarrow A \wedge B$	and B to A
$A \leftarrow \text{NOT } A$	negate A
$A \leftarrow A/2$	rotate A and Link right
$A \leftarrow A(S)$	rotate A and Link right, set OVF
$A \leftarrow A \times 2$	rotate A and Link left
$A \leftarrow A \times 2(S)$	rotate A and Link left, set OVF
$A \leftarrow B/2$	rotate B and Link right, store in A
$A \leftarrow B/2(S)$	rotate B and Link right, set OVF, store in A

**Instructions for B(arithmetic-logical):**

$B \leftarrow 0$	clear B
$B \leftarrow A$	store A in B



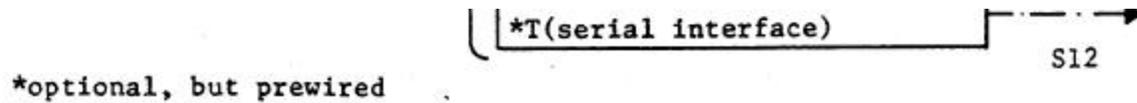


Fig. 16m-2. RTM diagram for the PDP-16/M.

- . Cost: 199
- . Word length: 16-bit data with 8-bit instructions
- . Modular and flexible: constructed from standard PDP-16 Modules, extensions via the PDP-16 Bus using PDP-11 and PDP-16 modules. The instruction set may also be changed.
- . Instruction times: 1.4 to 3 microseconds
- . Logic: TTL (ground and + 3 volts)
- . Environment: 0 to 70° C; 20 to 95 % relative humidity, non-condensing
- . Mounting: 10½" (high) x 19" (wide) x 14" deep with slides for RETMA 19" rack; weight approximately 35 lbs.
- . Power: 95 to 130 volts, single phase, 47 to 63 Hz., 3.0 amps; 190 to 260 volts, single phase, 47 to 63 Hz., 1.5 amps
- . Instructions: 127 basic with expansion to 200 in five types - register-transfer, unconditional and conditional branch, subroutine call and subroutine exit.
- . Arithmetic and Logic: 16-bit data words using two arithmetic accumulators (A and B) with LINK and Overflow (OVF) bits
- . Interface (input-output)
  - via standard computer:
    - 3 16-bit word-parallel full-duplex channels to 0.5 MHz.
    - 2 serial (8-bit) standard asynchronous full-duplex communications channels to 9600 baud; 26 input conditions can be sensed, and 32 output commands
  - via standard PDP-16 options: additional parallel and serial interfaces and new produce options via standard PDP-11 options; paper tape, printers, plotters, displays, etc.
  - via custom designed PDP-16 compatible modules: PDP-16 bus (21-wire, fully-interlocked)
- . Memory
  - for program: 256 to 1024 8-bit Programmable/Read Only-PROM/ROM
  - for 16-bit data: 3 to 7 bits (flags), 1 to 34 registers, 4 or 24 ROM constants, 256, 512 and/or 1024 word read-write arrays

Fig. 16m-1. Table of PDP-16/M attributes.

Programs are written in the 16/M's language and assembled (prepared) on another computer (e.g., the PDP-8). These programs are then either held in the PDP-8's memory and run in the 16/M while being debugged, or the PDP-8 prepares a program which is written into the Programmable Read Only Memory of the 16/M. The various attributes of interest to the system designer are given in Figure 16m-1.

The attributes that characterize the 16/M as a microprogrammed computer are somewhat tenuous. A microprogrammed computer is one whose K(interpreter) and instruction set are realized by a second computer (the microprogram computer). This latter computer is more primitive than regular computers in several ways: in having as its basic operation set the transfers between all the registers in the machine; in accessing random addressed (core) memories via address registers as external memories; in having read-only (hence non-modifiable) programs; and in evoking sets of parallel operations (like the OPR command of the Crtm-1). The 16/M has the first three of these characteristics, but not the fourth. Still, it appears as a genuinely more primitive computer than the minicomputers available on the commercial market

## LOGICAL ORGANIZATION

The organization of the PDP-16/M is shown in the RTM design of Figure 16m-2. PDP-16/M has four functional component parts, which are the same as those of the RTM set:

### Arithmetic Logic

K(bus) and DM(general purpose arithmetic) contain 3+ registers and arithmetic/logic operations capable of carrying out instructions on 16-bit data. The registers are: A and B which hold arithmetic results; the Link-bit\L which is used as, input data to A and B for shift operations; Overflow-bit\OVF, which holds overflow results from A and B; and Bus Sense\BSR which is set on every register data transmission and can be tested (i.e.,  $BSR=0$ ,  $BSR>0$ ,  $BSR<0$ ).

### Memory

These components are organized as bits (flags), registers, read only and read write arrays. The total memory size can be approximately 1,600 16-bit words. Only data is stored in the memory (not programs). The actual memory configuration can be up to: 6 bits - DM(flags), 2 word registers - M(transfer), 2 16-word scratchpads- M(scratchpad), 4+24 words of read only constants - M(constants), and  $256+256+1024$  (1536) words of read-write array - M(array).

### Input-Output

T(general purpose interface) and T(serial interface) communicate with the external systems. Up to three sets of 16-bit parallel input and output lines, and two sets of bit-serial input and output communications

equipment lines, are provided. Also, 26 Boolean (bit) inputs can be sensed and 32 output-evoke operations can be given.

## Control and program memory

The K(PCS) holds programs from 256 to 1024 words, 8 bits/word, in 256 word increments. It fetches instructions from its program memory and instructs the above three parts. It also senses input (bit) conditions from other parts of the system. Control has a Program Counter, PC, which points to the instruction being executed and a Subroutine Stack (16 words) to hold the previous value of the Program Counter when subroutines are called.

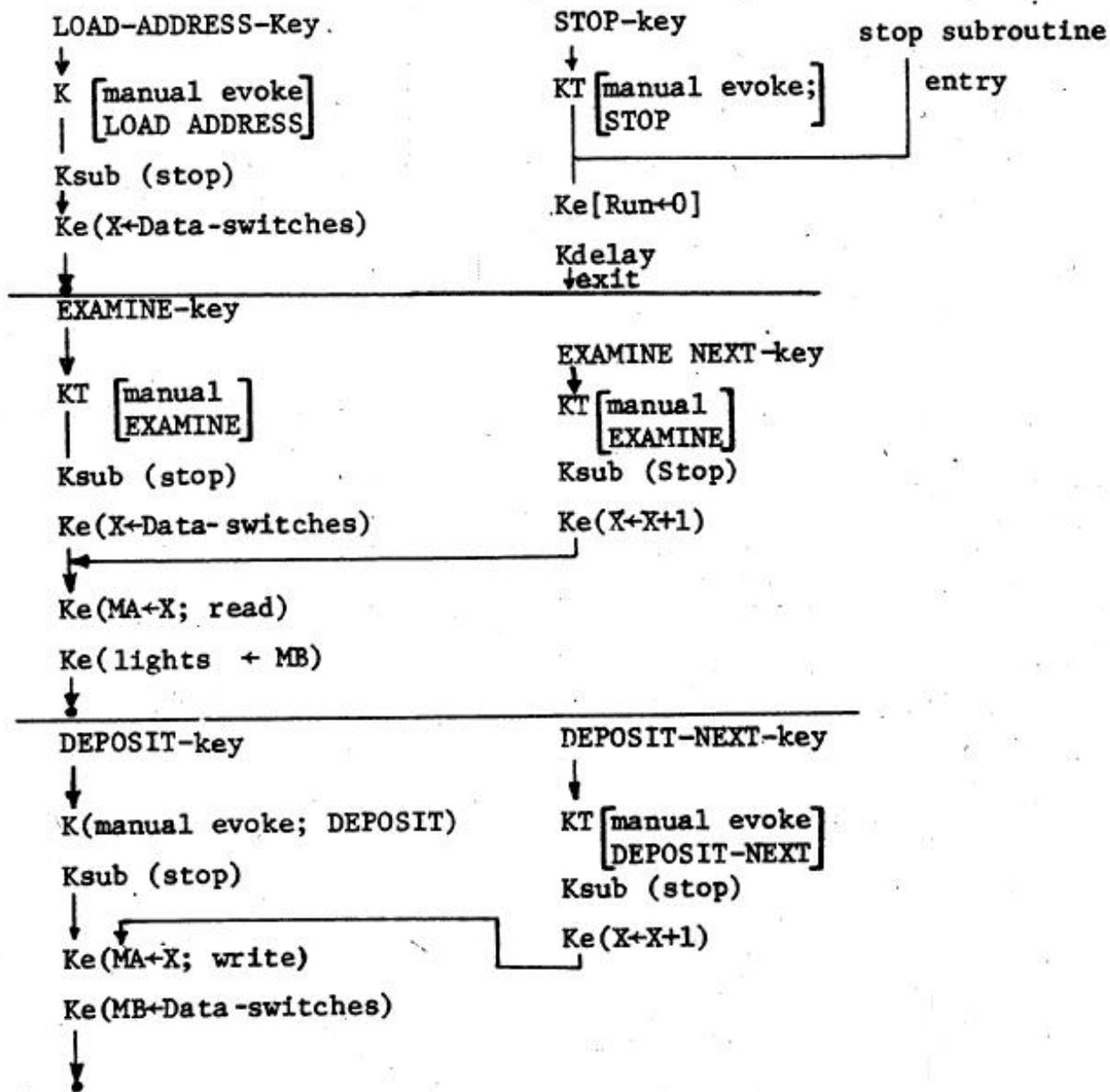


Fig. 8-10. Control part for manual-key macros.

Bitran-6 described in Chu (1971), write the ISP, convert the ISP to RTM, and construct it.

### THE PDP-16/M - A SUBMINICOMPUTER BASED ON RTM'S

The PDP- 16/M is a particular configuration of modules interconnected to behave in a fashion similar to a microprogram computer. The user (programmer) expresses register transfer operations in terms of the

fixed set of operations that are hardwired into the 16/M. The computer is based on the Program Controlled Sequencer, K(PCS), control module (see Chapter 2), and microinstructions are stored in the K(PCS) memory. Although the microinstruction set is fixed, it can be extended by rewiring. Also, additional data part modules can be added to change the basic structure of the 16/M. Behavior is expressed in about the same way as with the RTM flowchart.

is either a K(conditional-execute),  $K_{ce}(i < 4 \Rightarrow AC-0)$ , or a  $K_{b2}(IR < 4)$  connected to a  $K_e(AC < -0)$ , followed by a K(serial merge; 2 input). The remaining instructions are executed similarly. For the rotate instructions, a rotate subroutine is defined and called once; if  $IR < 10$  equals one, rotate is called again.

## Manual Key Processes

Figure 8-10 gives the six processes that define the action taken when each one of the six console keys is depressed. These processes are of interest because they occur in parallel with the main execution process. There is, however, an assumption that no two of the keys are depressed at exactly the same time. All of these processes call the stop subroutine (also the same as the STOP-key), which first sets the Run Boolean to 0 so that the interpretation loop will be broken (and the machine will stop) prior to the execution of the action implied by the key strike.

## CONCLUSIONS

In the above section we have shown another implementation of a computer, given a set of primitives. In this case, very few design decisions were exercised in implementing an Instruction set with the RTM's. The problem of translating the ISP into an RTM implementation was of interest. We were concerned with making the translation easily, cheaply, and clearly. Actually, the ISP description (Fig. 8-1) has been manipulated more than trivially in restructuring the interpreter to avoid the use of closed subroutines. A different initial ISP description would have minimized this factor. There can be other implementations that execute instructions faster. By adding more busses (and possibly more memories) the system would have more parallelism. Such an implementation might look more like pipelined machines or those machines with instruction look-ahead units; one unit would be concerned with fetching as many instructions as 'possible, while the other unit independently executed the instructions. These techniques would create a computer which might be considered larger than a minicomputer. From a pedagogical viewpoint such machines are quite interesting.

## PROBLEMS

1. What is the cost and performance of PDP-8/RTM?
2. Write programs for the PDP-8/RTM for the sum-of-integers, multiply, binary BCD, BCD-binary, and Fibonacci number algorithms.
3. Modify the PDP-8/RTM to have built-in multiplication. Show several incremental designs which trade off cost-performance. One intermediate possibility is to have a design which requires 12 multiply steps, and thus might be considered in the middle between hardware and software.
4. Modify the PDP-8/RTM to include facilities to convert between BCD and binary. Show several

designs and show incremental cost-performance tradeoff.

5. How can PDP-8/RTM be modified to have higher performance?
6. Given the programming manual or your knowledge of a minicomputer, describe the minicomputer using ISP.
7. Given an ISP of a minicomputer, convert the ISP into an RTM implementation.
8. Given the ISP of the Crtm-1, the PDP-8, the general register minicomputer, or any other, write a Fortran program to simulate the behavior, of the minicomputer. List the steps required in the translation from ISP to Fortran.
9. Design a multiple Bus RTM PDP-8 that operates faster. How fast will it operate?
10. Given a computer used for educational purposes (e.g. the simple computer or

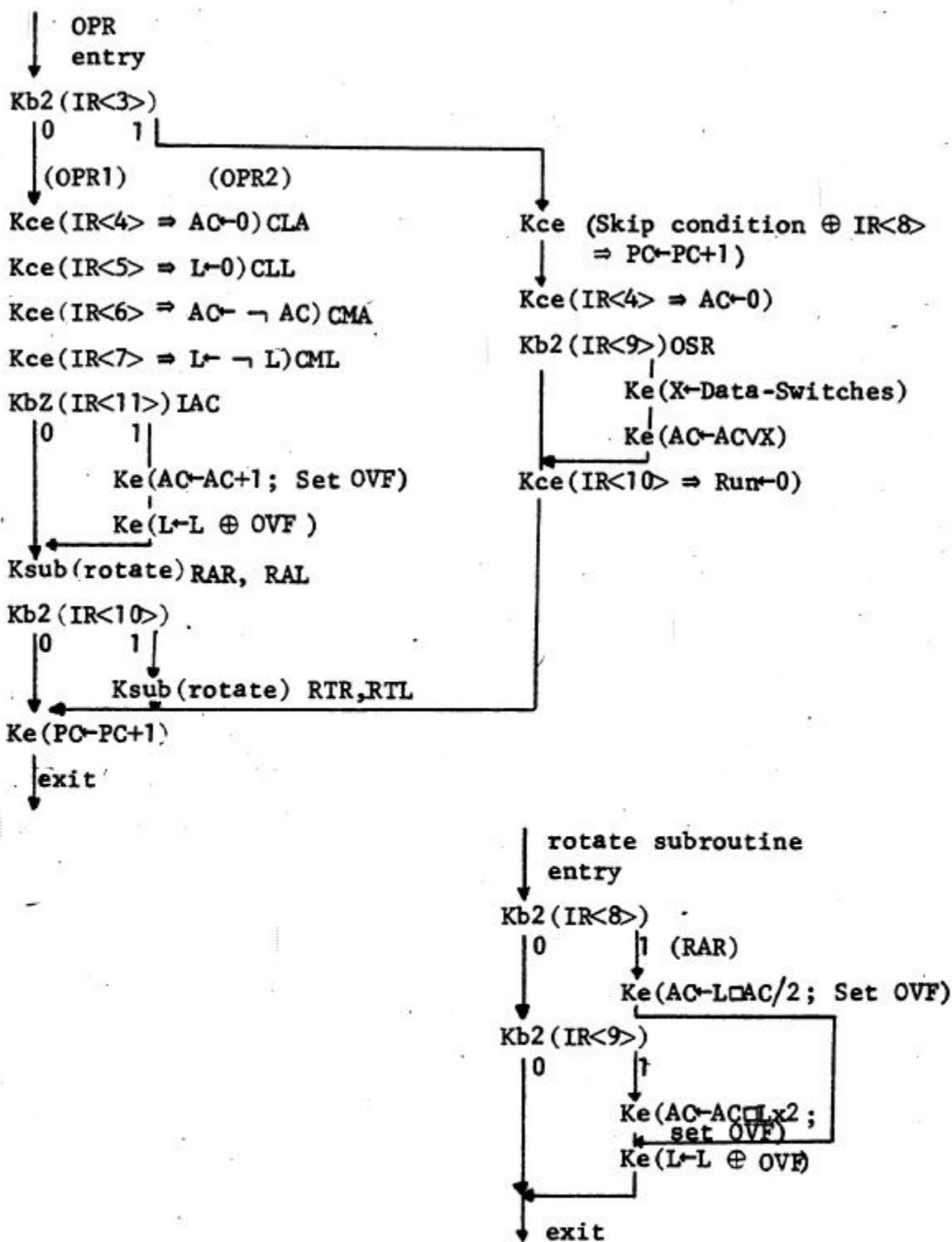


Fig. 8-9. Control part for OPR-instruction-execution macro.

[previous](#) | [contents](#) | [next](#)

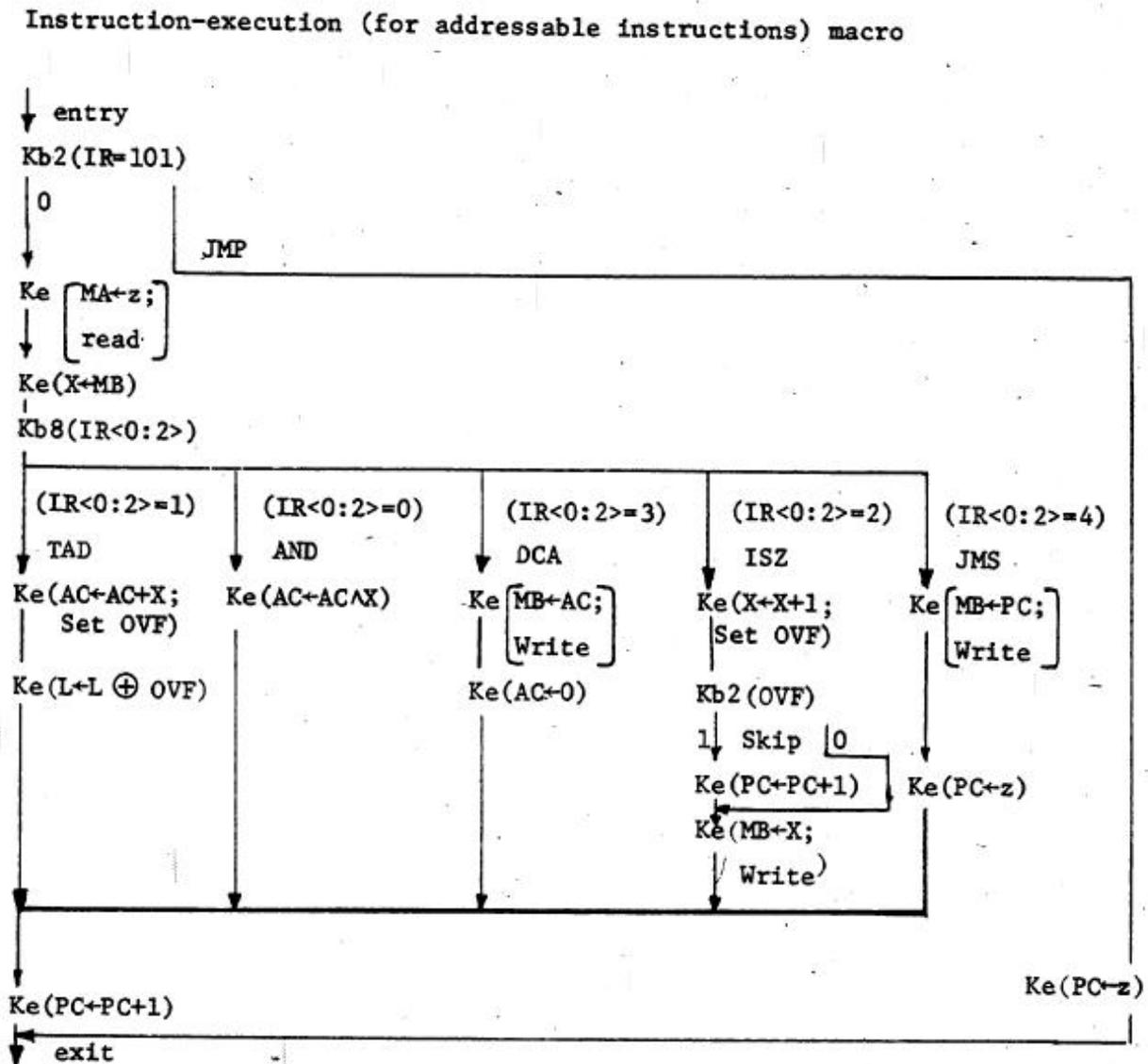


Fig. 8-8. Control part of Instruction-execution macro (for addressable instructions).

### Operate Instruction-Execution Process

Figure 8-9 gives the implementation of the operate (OPR) instruction. Operate actually consisted of two (really three, but we exclude the instructions for the control of the optional Extended Arithmetic element) sub-instruction sets and the first Kb2 decides which of the two sets is to be executed. The execution of these microcoded instructions is carried out in a direct translation from the ISP description using K(conditional evoke) ERTM modules.

For example, the instruction:

**cla (:= i<4> = 1) → (AC←0);**

319

[previous](#) | [contents](#) | [next](#)

**Effective-address-calculation macro**

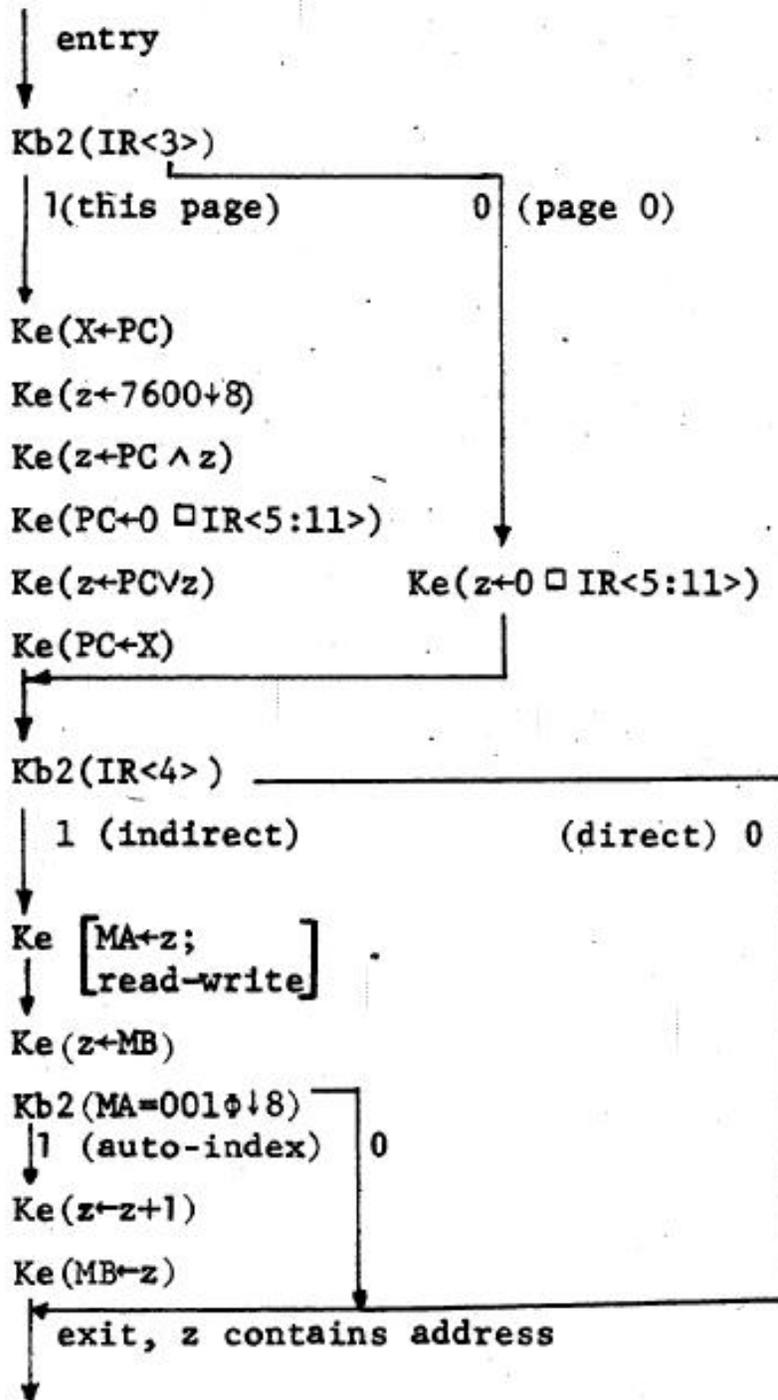


Fig. 8-7. Control part of effective address calculation.

contents of memory, specified by  $z$  (via  $MA$ ) from its Memory Buffer register,  $MB$ , into register  $X$  ( $X \leftarrow MB$ ). The actual execution of each particular instruction is a direct translation from the corresponding

ISP statement. For example, JMS ( $M[z] \leftarrow PC; PC \leftarrow z+1$ ) translates into:

↓	
Ke( $MB \leftarrow PC; write$ )	MA contains z, i.e., $M[z] \leftarrow PC$
Ke( $PC \leftarrow z$ )	
Ke( $PC \leftarrow PC+1$ )	$PC \leftarrow z+1$
↓	

START-Key and CONTINUE-Key processes for example, note that the subroutine, stop, is first called in order to insure that the computer is not started when it is already running.

The basic part of the control flow which is of interest here is the Instruction- fetch process, consisting of the control steps  $MA \leftarrow PC$ ; read; and  $IR \leftarrow MB$ . We have omitted incrementing the Program Counter (i.e.,  $PC \leftarrow PC+1$ ) at this time because the Effective-address-calculation process which deviates from the ISP description requires a value based on the location of the current instruction. In this diagram there are still three large remaining control processes: Effective-address- calculation, Instruction-execution (for Addressable instructions), OPR and IOT execution.

### Effective-address-calculation-process

In the initial description (above) of the control part for the effective address calculation, the formal ISP description was redefined in a form which could be converted to a simple RTM control process without using closed subroutines and K(subroutine call). Now, the problem is to take the basic ISP (above) and convert it into the RTM control. Such a conversion is quite direct; the RTM version is given in Figure 8-7.

The first part of the ISP description is concerned with determining an address which is either on page 0 or the currently active page. Thus, in the ISP description the page 0 bit is examined, and if 0, the address is the page-address,  $IR \langle 5:11 \rangle$ . If the page 0 bit is 1, then the address is on the current page,  $PC \langle 0:4 \rangle \parallel IR \langle 5:11 \rangle$ . In the RTM implementation this is just a sequential process which begins by testing  $IR \langle 3 \rangle$ , the page 0 bit, and forms one of the above corresponding addresses. The formation of the address on page 0 is just  $z \leftarrow IR \langle 5:11 \rangle$ . The address formation on the current page requires several steps in order to accomplish the concatenation. First, the PC must be preserved in a temporary register while the calculation is made;  $X \leftarrow PC$  saves the PC (and  $PC \leftarrow X$  restores PC, on completion of the concatenation process). In order to get at the most significant bits of the PC a constant mask 7600v8 is put in the register z and the result of  $z \leftarrow PC \wedge X$  isolates  $PC \langle 0:4 \rangle$ . Finally,  $IR \langle 5:11 \rangle$  is placed in the register normally holding PC, and z (which holds  $PC \langle 0:4 \rangle$ ) is OR'd with  $IR \langle 5:11 \rangle$  to give the final operation needed for concatenation.

In the second part of the Effective-address-calculation process the indirect address bit  $IR \langle 4 \rangle$  is examined, and if 0, then z already contains the effective address; otherwise  $M[z]$  contains the effective address, z. As a side effect of indirect addressing, if a reference is made to locations 10v8 to 17v8, then a 1 is added to the memory location (auto-indexing). In this part of the calculation there is a direct translation of the ISP statements into the corresponding RTM control statement implementation, since no temporary registers are involved.

### Instruction-Execution Process for the Addressable Instructions

Figure 8-8 gives the RTM implementation of this process which is a more direct translation from the ISP description than in the preceding cases. The first control, Kb2, decides whether a memory reference is required (the jump instruction is the only instruction with an address that does not reference memory). For the execution of the jump instruction (JMP mnemonic with op code equal to 5), the execution is merely PC4-z (identical to the ISP definition).

For the remaining instructions, which actually reference memory, the loading of the Memory Address register, MA, has been done initially and causes the memory to read (MA<-z; read). Next a Kevoke specifies the loading of the

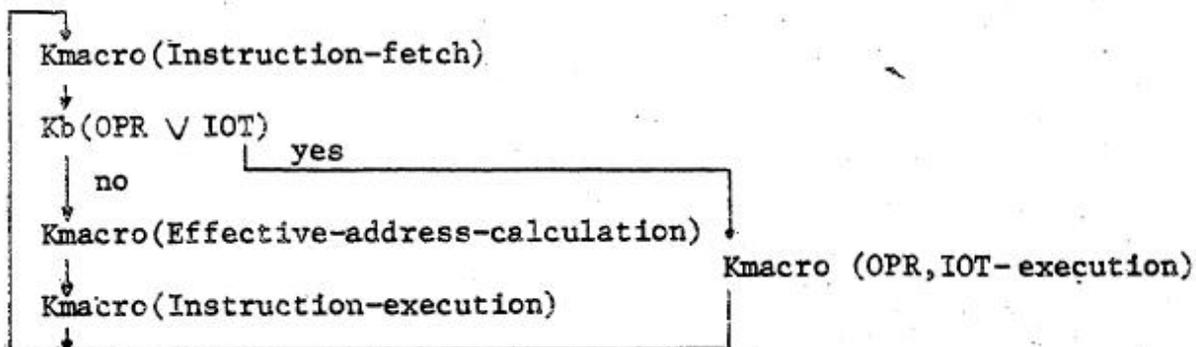


Fig. 8-5. Control part for PDP-8/RTM.

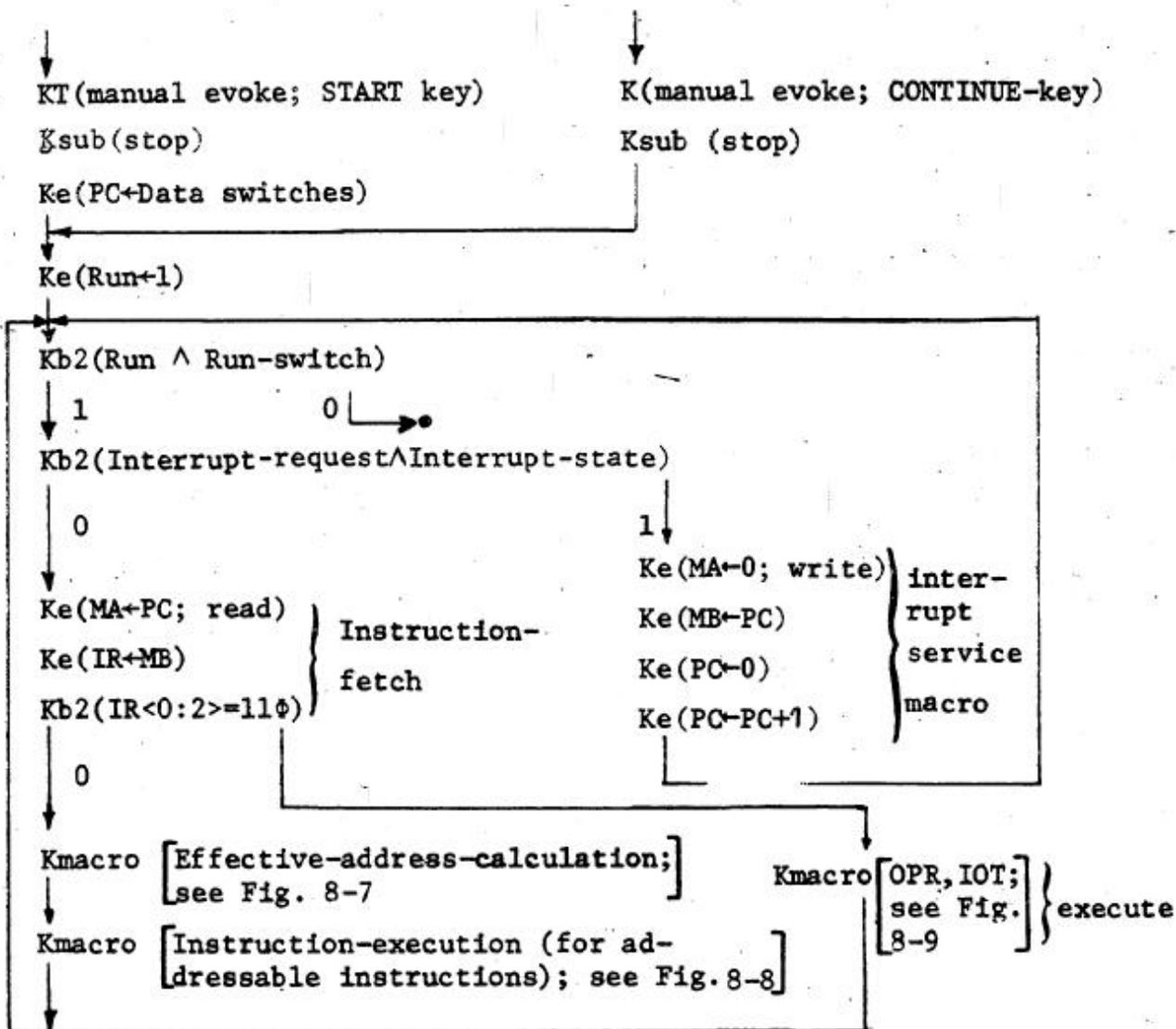
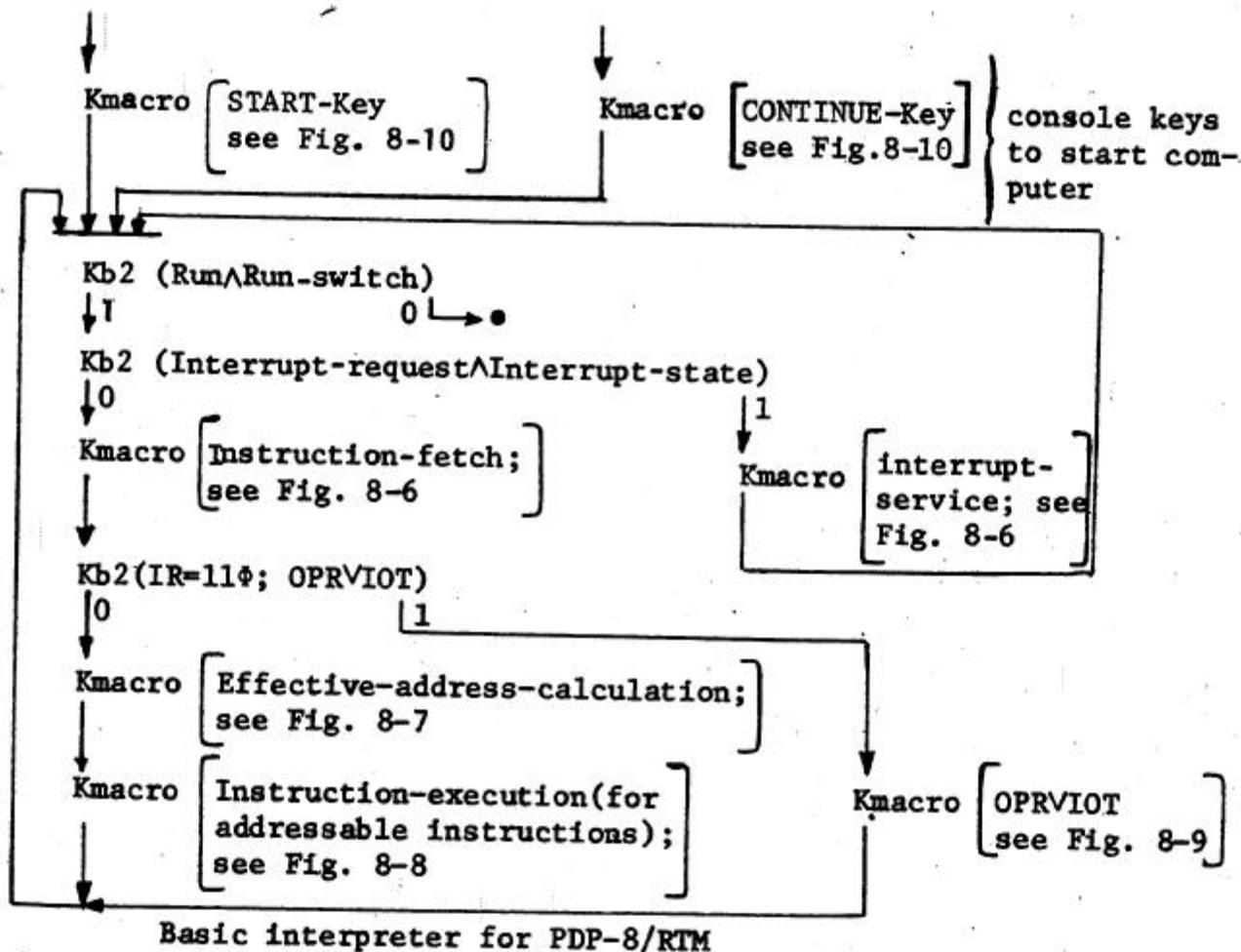


Fig. 8-6. Control part for PDP-8/RTM, (including START, CONTINUE keys, Instruction-fetch, and Interrupt service).

316

[previous](#) | [contents](#) | [next](#)



↓ Kmacro [LOAD-ADDRESS-Key see Fig. 8-10]	↓ Kmacro [STOP-Key see Fig. 8-10]
↓ Kmacro [EXAMINE-Key see Fig. 8-10]	↓ Kmacro [EXAMINE-NEXT-Key see Fig. 8-10]
↓ Kmacro [DEPOSIT-Key see Fig. 8-10]	↓ Kmacro [DEPOSIT-NEXT-Key see Fig. 8-10]

Fig. 8-4. Control part for PDP-8/RTM interpreter.

actions when the user strikes various console keys. These are not given in the ISP, but are given in the figure for completeness.

Figure 8-6 is basically the same as Figure 8-4, except that the macros START-Key, CONTINUE-Key,

Instruction-fetch, and Interrupt-service are completely defined in terms of physical K's. (The other Key processes of Figure 8-4 are not defined here, but are given in detail in Figure 8-10). With the

315

[previous](#) | [contents](#) | [next](#)

**Effective-address calculation; next  
Instruction-execution)**

**operand address calculation  
execute**

where the subsequent effective address values  $z'$  can now be defined simply as:

**$z' := z$**

Thus, in the modified description of the ISP, all other parts of Figure 8-1 remain the same, and all we have done is to allow the value of the effective address to be precalculated to the value,  $z$ . The Effective-address-calculation. process for this value becomes:

**Effective-address-calculation := (**

<b><math>\neg ib \rightarrow (z \leftarrow z'' );</math></b>	<b>direct address</b>
<b><math>ib \wedge (10 \downarrow 8 \leq z'' \leq 17 \downarrow 8) \rightarrow M[z''] \leftarrow M[z''] + 1; next</math></b>	<b>auto index indirect</b>
<b><math>ib \rightarrow z \leftarrow M[z'']</math></b>	<b>indirect</b>

The reader should note that the value  $z$  appears several times in the above process. Instead of continuing to calculate the value repeatedly, we can as in the method used above, first calculate the value for  $z''$  (and store it in  $z$  temporarily) prior to executing the body of the process. Thus the process now becomes:

**Effective-address-calculation := (**

<b><math>\neg \text{page 0 bit} \rightarrow (z \leftarrow 0 \square \text{page address});</math></b>	<b>page 0</b>
<b><math>\text{page 0 bit} \rightarrow (z \leftarrow \text{this page} \square \text{page address}); next</math></b>	<b>this page</b>
<b><math>\neg ib \rightarrow ;</math></b>	<b>direct</b>
<b><math>ib \wedge (10 \downarrow 8 \leq z \leq 17 \downarrow 8) \rightarrow (M[z] \leftarrow M[z] + 1); next</math></b>	<b>auto-index</b>
<b><math>ib \rightarrow (z \leftarrow M[z])</math></b>	<b>indirect</b>

We now have an ISP description that evaluates functions in a sequential manner, using a procedure mechanism rather than making excessive calls (as was the case in the original description of Figure 8-1). The problem of converting the ISP description to an RTM control flow diagram which will be used to control the data part in the previous section can now be solved.

Basic Control Process for PDP-8/RTM Interpreter

The complete control process is given in Figure 8-4 using several complex secondary processes (macros). These secondary processes are defined precisely in subsequent figures. (Figure 8-4 identifies the specific figures for the detailed secondary macros.) The actual Kmacro structure would be substituted for the Kmacro call in a higher level figure. The reader should note that the PDP-8/RTM implementation does not include the definition of the i/o transfer instruction, IOT, nor does it include the various console keys.

The basic process of the interpreter is a loop (which is unbroken so long as the Run-switch AND the Run Booleans are true; otherwise the process stops). If an Interrupt request Occurs AND the machine is in the Interrupt state, then the Interrupt-service process is evoked.

The main interpretation, or fetch-execute cycle, is the just the sequence consisting of the four control parts shown in Figure 8-5.

The remaining control parts in the figure are concerned with carrying out

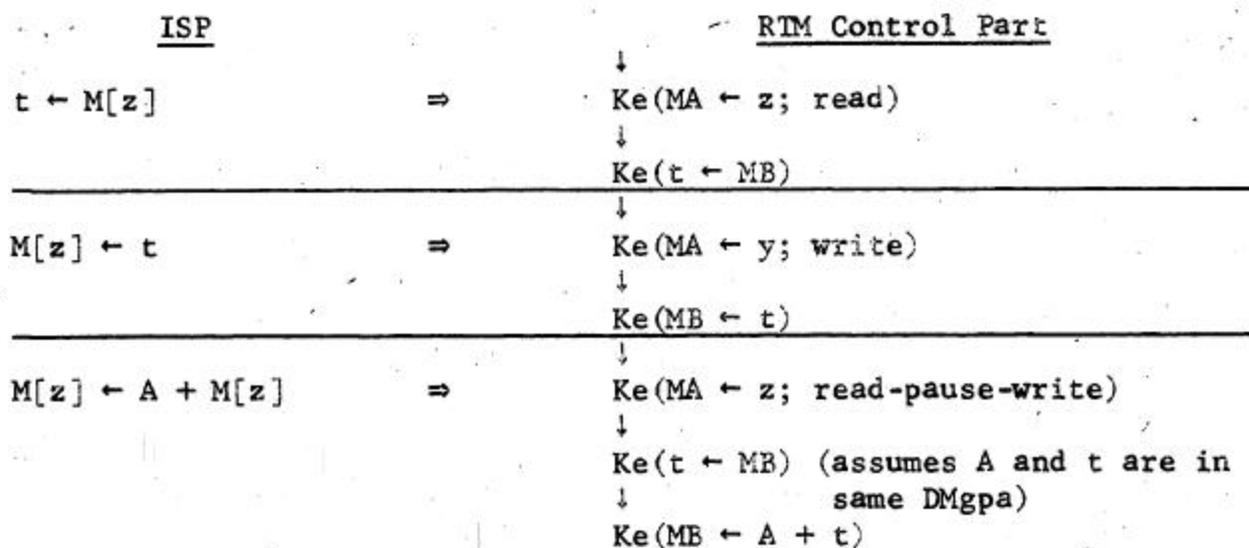


Fig. 8-3. ISP to RTM transformation for memory access.

The instruction,  $i$ , is stored in an M(transfer) labelled IR. We can deduce from the ISP instruction formats the various decoding and register partitioning operations required to separate the instruction more completely; two parts of the M(transfer) are declared: the subregister IR<5:11>, which is used to hold the page-address, and the IR<0:2> bits, which form the op code part of the instruction. Note that the specially named bits, e.g., page 0 bit, indirect bit, sma, sza and snl, are all available as Booleans at the output of the IR.

The final data part is the effective address,  $z$ , which we have discussed above. Here we include it as part of the DMgpa which holds the PC. Three other modules are connected to the data part: the Kbus, the T(serial interface) for a Teletype, and an M(constants). (We place M(constants) in the structure now because we have already worked the problem and know it is required.)

Now we have at least enough memory modules to hold the state (registers) of the machine. We also have a certain amount of flexibility in assigning the various registers to the appropriate DM and M modules. The Instruction-interpretation process can now be defined in terms of the actions implied by each of the instructions in the ISP.

## CONTROL PART

There are basically two approaches to the Instruction-interpretation process descriptions. The first method, with many subprocesses (macros), was used in the ISP description of Figure 8-1, because this approach is suited to the human reader. In this approach the Effective-address-calculation process is not invoked until an address is needed in the description of an instruction. Unfortunately, this description is

not well suited to hardware, since it makes repeated subroutine calls to the Effective-address-calculation process for the value,  $z$ . The second approach, which will be used in the following, first calculates the effective address as part of the process of the interpreter, prior to the execution of the instruction. Thus, the ISP interpreter should be modified to:

```
Run  $\wedge \neg(\text{Interrupt-request} \wedge \text{Interrupt-state}) \rightarrow ($     no interrupt
      instruction  $\leftarrow M[\text{PC}]; \text{PC} \leftarrow \text{PC} + 1; \text{next}$     fetch
```

313

[previous](#) | [contents](#) | [next](#)

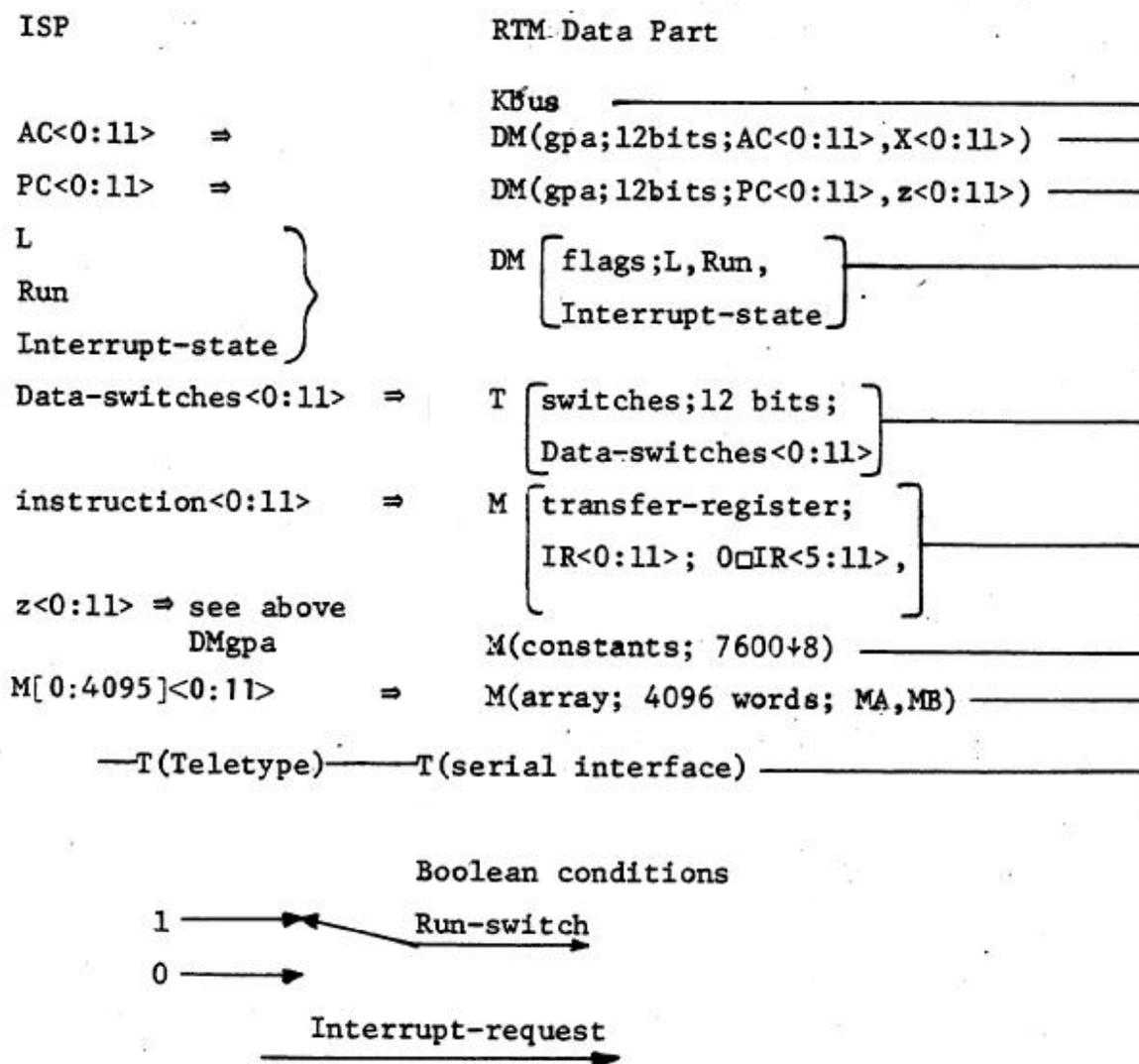


Fig. 8-2. RTM data part of PDP-8/RTM registers.

Three DMflag's hold the following 1-bit registers: Link\L, Run, and Interrupt state.

The main program memory, M, is a core memory array consisting of 4096 12-bit words.(3) There are two registers within M, namely, MA and MB, to hold the address and data especially for the array. In this case, the RTM implementation introduces other registers that are not part of the ISP description because of hardware memory idiosyncrasies. The control parts of the ISP to RTM transformation of the memory accesses for reading, writing, and reading-pause- writing (where a memory expression appears on both sides of an ISP expression) are shown in Figure 8-3. The read, write, and read-pause-write commands are needed to tell the core memory which part(s) of its inherent read-write cycle to use in each case.

The Data-switches, a transducer, is a 12-bit switch register which takes human input and converts it into RTM readable form.

-----

3. Although the basic set of RTM's has no such module, the only difference between it and the conventional M(array) is the number of words.

312

[previous](#) | [contents](#) | [next](#)

**Operate Instruction Set**

The microprogrammed operate instructions: operate group 1, operate group 2, and extended arithmetic are defined as a separate instruction set.

```

Operate_execution := (
  cla := i<4> = 1) → (AC ← 0);           clear AC. Common to all operate instructions.
  opr_1 := i<3> = 0) → (
    cli := i<5> = 1) → (L ← 0); next    operate group 1
    μ clear link
    cma := i<6> = 1) → (AC ← ¬ AC);     μ complement AC
    cml := i<7> = 1) → (L ← ¬ L); next  μ complement L
    lac := i<11> = 1) → (LQAC ← LQAC + 1); next μ increment AC
    ral := i<8:10> = 2) → (LQAC ← LQAC × 2 {rotate}); μ rotate left
    rtl := i<8:10> = 3) → (LQAC ← LQAC × 22 {rotate}); μ rotate twice left
    rar := i<8:10> = 4) → (LQAC ← LQAC / 2 {rotate}); μ rotate right
    rtr := i<8:10> = 5) → (LQAC ← LQAC / 22 {rotate}); μ rotate twice right
  opr_2 := i<3,11> = 10) → (
    skip condition ⊕ (i<8> = 1) → (PC ← PC + 1); next μ AC, L skip test
    skip condition := ((sma ∧ (AC < 0)) ∨ (sza ∧ (AC = 0)) ∨ (snl ∧ L))
    nsr := i<9> = 1) → (AC ← AC ∨ Data switches); μ "or" switches
    hlt := i<10> = 1) → (Run ← 0); μ halt or stop
  EAE := i<3,11> = 11) → EAF_Instruction_execution) optional EAE description

```

Fig. 8-1 (Part 3 of 3).

**Effective Address Calculation Process**

```

z<0:11> := (
  ¬lb → z'';
  lb ∧ (108 ≤ z'' ≤ 178) → (M[z''] ← M[z''] + 1; next);
  lb → M[z''])
z'<0:11> := (¬ lb → z''; lb → M[z''])
z''<0:11> := (page_0_bit → this_page_address;
             ¬page_0_bit → 0page_address)

```

*effective**auto indexing**direct address*μ *microcoded instruction or instruction bit(s) within an instruction***Instruction Interpretation Process**

```

Run ∧ ¬ (Interrupt_request ∧ Interrupt_state) → (
  Instruction ← M[PC]; PC ← PC + 1; next
  Instruction_execution);
Run ∧ Interrupt_request ∧ Interrupt_state → (
  M[0] ← PC; Interrupt_state ← 0; PC ← 1)

```

*no interrupt interpreter**fetch**execute**interrupt interpreter***Instruction Set and Instruction Execution Process**

```

Instruction_execution := (
  and (: = op = 0) → (AC ← AC ∧ M[z]);
  tad (: = op = 1) → (L⊖AC ← L⊖AC + M[z]);
  isz (: = op = 2) → (M[z'] ← M[z] + 1; next
                    (M[z'] = 0) → (PC ← PC + 1));
  dca (: = op = 3) → (M[z] ← AC; AC ← 0);
  jms (: = op = 4) → (M[z] ← PC; next PC ← z + 1);
  jmp (: = op = 5) → (PC ← z);
  iot (: = op = 6) → (
    io_p1_bit → IO_pulse_1 ← 1; next
    io_p2_bit → IO_pulse_2 ← 1; next
    io_p4_bit → IO_pulse_4 ← 1);
  opr (: = op = 7) → Operate_execution

```

*logical and**two's complement add**index and skip if zero**deposit and clear AC**jump to subroutine**jump**μ in out transfer, microprogrammed to generate up to 3 pulses to an io device addressed by IO\_select**the operate instruction is defined below end instruction execution*

Fig. 8-1 (Part 2 of 3).

310

Fig. 8-1 (Part 1 of 3). ISP description of PDP-8.

<b>Pc State</b>		
AC<0:11>		Accumulator
L		Link bit/AC extension for overflow and carry
PC<0:11>		Program Counter
Run		1 when Pc is interpreting instructions or "running"
Interrupt_state		1 when Pc can be interrupted; under programmed control
IO_pulse_1; IO_pulse_2; IO_pulse_4		IO pulses to I/O devices
<b>Mp State</b>		
Extended memory is not included.		
M[0:777] <sub>8</sub> <0:11>		special array of directly addressed memory registers
Page_0[0:177] <sub>8</sub> <0:11> := M[0:177] <sub>8</sub> <0:11>		special array when addressed indirectly, is incremented by 1
Auto_index[0:7]<0:11> := Page_0[10 <sub>8</sub> :17 <sub>8</sub> ] <sub>8</sub> <0:11>		
<b>Pc Console State</b>		
Keys for start, stop, continue, examine (load from memory), and deposit (store in memory) are not included.		
Data switches<0:11>		data entered via console
<b>Instruction Format</b>		
instruction/1<0:11>		
op<0:2>	:= 1<0:2>	op code
indirect_bit/lb	:= 1<3>	0, direct; 1 indirect memory reference
page_0_bit/p	:= 1<4>	0 selects page 0; 1 selects this page
page_address<0:6>	:= 1<5:11>	
this_page<0:4>	:= PC'<0:4>	
PC'<0:11>	:= (PC<0:11> - 1)	
IO_select<0:5>	:= 1<3:8>	selects a T or Ms device
io_1_bit	:= 1<11>	these 3 bits control the selective generation of -3 volts,
io_2_bit	:= 1<10>	0.4 μs pulses to I/O devices
io_4_bit	:= 1<9>	
sma	:= 1<5>	μ bit for skip on minus AC, operate 2 group
sza	:= 1<6>	μ bit for skip on zero AC
snl	:= 1<7>	μ bit for skip on non zero link

309

9. Modify Crtm-1 to include a Direct Memory Access (DMA) facility. With DMA, another system can communicate with a computer by reading directly into the computer's memory. Thus, the DMA provides a computer with the capability of behaving as an M(array).

10. Read about the two ERTM's described in the last section which assume program interrupt and DMA capabilities for a computer. Modify the two ERTM's to be compatible with the design of problems 8 and 9 above.

## PDP-8/RTM IMPLEMENTATION

In this section we show how the DEC PDP-8, a common small minicomputer, can be implemented using RTM's. The goal of designing the PDP-8 from RTM's is mainly pedagogical. One would not expect to see a production, model computer constructed from RTM's due to considerations of production economy and technology.

The computer will be defined using ISP. In the implementation of any computer, we feel it is necessary to have a relatively formal (non-natural language text) description. Such a description enables the machine to be well defined. ISP satisfies this goal. Figure 8-1 gives the ISP description of the PDP 8, taken from Bell and Newell (1971).

A premise of, the RTM implementation is that one can take the formal description of the machine, in ISP (or some other language), and in an algorithmic fashion generate the implementation. In reality, we have not yet accomplished this goal, but the description will proceed in a quasi-formal way as though this were possible, and thus we will proceed to generate all the given parts from the ISP description. We are ignoring the various input-output devices.

## DATA PART

For each register declared in the/ defining ISP description, a corresponding physical register is required to contain the actual condition. Using this procedure we get an RTM structure for the data part as shown in Figure 8-2. In this figure the two registers, AC\Accumulator and PC\Program Counter, of the ISP processor state are assigned to two DMgpa's. (In Crtm-i these were named A and P.) Two DMgpa's are used, not because of the innate character of AC or PC, but 'because of the operations to be performed upon them. We know that binary arithmetic and logical operations are performed on AC, necessitating a second operand register. Lack of the second operand register  $X_{0:11}(2)$  would require undue movement of data to get the operands into the register to be operated on. Thus, we have made the first implicit design decision (which we hope will result in a short, simple control part). As the second decision, the PC is assigned to a second DMgpa, based on the knowledge that PC is to be incremented by one (i.e., it is a counter). For the time being, no other assignments are made to the second registers in the two DMgpa's, and they are considered to be temporary registers, X and z, being synonymous with the B

registers of DMgpa's. We might expect that having an additional, separate DMgpa for the effective address calculation process (i.e. for z) would simplify the control; unfortunately it doesn't. There would only be one less Kevoke in the control part; therefore, we would rather increase the control part by one unit, in order to save an extra DMgpa.

-----

2. Note, bits in the PDP-8 are numbered from left to right (most to least significant).

308

[previous](#) | [contents](#) | [next](#)

the instruction, especially if it is to be calculated only once and will have a fixed value independent of anything that happens while executing instructions. Console activity can also be described in the interpreter, e.g., the effect of a switch that permits stepping through the program under manual control or interrogating and changing memory.

The normal statement for the simple computer interpreter is just the loop:

<b>Interpret :=</b>	
<b>(i ← M(P); P ← P + 1; next</b>	<b>fetch</b>
<b>Instruction-interpretation; next</b>	<b>execute</b>
<b>Run-switch → interpret)</b>	<b>(loop)</b>

which corresponds essentially to the loop expressed as a flowchart in Figure Crtm-4.

### Instruction-Set and Instruction Execution Process

The instruction set and the process by which each instruction is executed are usually given together in a single definition; this definition is called Instruction- execution in most ISP descriptions. This Instruction- execution usually includes the definition of the conditions for execution, i.e., the operation code, value, the name of the instruction (usually a comment), the mnemonic alias by which it is known, and the process for the instruction's execution. Thus, an individual instruction typically has the form:

<b>AND(:= op = 0) → (A ← A ∧ M[z])</b>	<b>logical AND</b>
--	--------------------

With this format for the instruction, the entire instruction set is simply a list of all the instructions. On any particular execution, as evoked by the Instruction- execution process, typically one and only one operation code correlation will be satisfied, hence one and only one instruction will be executed. Instruction decoding occurs locally and in parallel as each instruction recognizes whether it is to be executed. However, if more than one operation code is satisfied then this implies that parallel activity is initiated. (This is how parallel acting microcodes are described.)

### PROBLEMS

1. Write a program for Crtm-1 to compute the sum-of-integers and the Fibonacci numbers. Modify these to be subroutines. Show the numeric values for instructions of the multiply subroutine for Crtm-1 given in Ch. 4.
2. Add the Run-switch, described in the ISP description, to the RTM diagram of Crtm-1.

3. Add a link bit, L', like the PDP-8 has, together with instructions to manipulate it, to Crtm-1.
4. Since subroutines cannot be nested (i.e., call one another), nor can a subroutine find out from where it was called, their use tends to be awkward and limited in Crtm-1. Correct the design to fix this (e.g., allow L to be read into A).
5. Add an instruction to Crtm-1 to be able to load data from T(lights & switches).
6. Give Crtm-1 the capability of inputting and outputting data through T(serial interface) and T(lights & switches) under console control.
7. Add instructions to Crtm-1 to improve the performance of BCD-Binary conversion.
8. The PDP-8 design has a capability, called program interrupt, which allows an external event Boolean signal to request that a program beginning in memory location 1 be run. Modify Crtm-1 to include this capability.

performs the second data transmission after the first one (provided  $M[z]=0$ ).

## Memory Declarations

Memory is defined by giving a memory declaration as shown in Figure Crtm-5. This declaration is essentially the convention which prevails in this book.

$M[0:1777\downarrow 8]<15:0>$

## Instruction Format

Instruction formats are declared in the same fashion as memory and are not distinguishable as special non-memory entities. The instructions are carried in a register; thus it is natural to declare them by giving names to the various parts of the instruction register. Usually only a single declaration is made, the instruction  $i$ , followed by the declarations of the parts of the instruction; the operation code, the address field, indirect bit, etc. This corresponds to the conventional box diagram of Figure Crtm-2.

## Operand Address Calculation Process

In all processors instructions make use of operands. In most conventional processors, the operand is usually in memory defined as  $M[z]$ , where  $z$  is the effective address. It is defined in ISP by giving the process that calculates it. This process may involve only accesses to primary memory (possibly indexed), but it may also involve side effects, i.e., the modification of either primary memory or processor memory (e.g., by incrementing a register). Note that the effective address is calculated whenever its name is encountered during the evaluation of an ISP expression (either in an instruction or in the interpretation expression). That is, it is evaluated on demand.

The PDP-8 ISP in the following section has an address calculation process, whereas Crtm-1 does not. In Crtm-1, bits  $<9:0>$  of the instruction are simply used to specify a memory location,  $z$ .

## Data-types

In more complex computers it is worthwhile to describe data-types in terms of various field within a word. For example, sign-magnitude and sign-magnitude floating-point data have 2 and 4 parts of data, respectively, that must be manipulated. Crtm-1, the PDP-8, and indeed most minicomputers do not have sufficiently complex data-types to require declaration.

## Data-type operations

Given that data-type declarations are made, it is necessary to define operations on them. (e.g., +,-,x/, movement). The reader is invited to look elsewhere for this aspect of ISP (Bell and Newell, 1971).

### Instruction Interpretation Process

The instruction interpretation expression and the instruction set constitute a single ISP expression that defines the processor's action. In effect, this single expression is evaluated and all the other parts of the ISP description of a processor are evoked as indirect consequences of this evaluation. Simple interpreters without interrupt facilities have just the familiar cycle of fetch-the- instruction and execute-the- instruction. In more complex processors the conditions for trapping and interrupting must also be described. The effective address calculation may also be carried out in the interpreter prior to executing

Instruction-execution process

```

Instruction-execution := (
  AND (:= op = 0) ⇒ (A ← A ∧ M[z]);           logical and
  ADD (:= op = 1) ⇒ (A ← A + M[z]);           two's complement add
  ISZ (:= op = 2) ⇒ (M[z] ← M[z] + 1; next;   increment and skip if zero
                    (M[z] = 0) ⇒ P ← P + 1);
  DCA (:= op = 3) ⇒ (M[z] ← A; A ← 0);       deposit and clear A
  JMS (:= op = 4) ⇒ (L ← P; P ← z);          jump to subroutine
  JMP (:= op = 5) ⇒ (P ← z);                 jump
  HLT (:= op = 6) ⇒ ;                         stops computer
  OPR (:= op = 7) ⇒ (
    i<4> ∧ (A=0) ⇒ (P ← P+1);                 skip if A=0
    i<3> ∧ (A<15>) ⇒ (P ← P+1); next         skip if A negative
    i<10> ⇒ (A ← 0); next                    clear A
    i<9> ⇒ (A ← ¬ A); next                   complement A
    i<8> ⇒ (A ← A+1); next                   add 1 to A
    i<7> ⇒ (A ← A/2); next                   shift right A
    i<6> ⇒ (A ← A×2){logical}; next         shift left logical A
    i<5> ⇒ (P ← L);                          return from subroutine
  ) end Instruction-execution

```

Fig. Cr1m-6. (Part 2 of 2).

305

Fig. Cr1m-6. (Part 1 of 2). ISP description of Cr1m-1.

Processor state

A<15:0>\Accumulator  
 P< 9:0>\Program Counter  
 L< 9:0>\Subroutine Link

holds return address of subroutinePrimary memoryM[0:1777<sub>g</sub>]<15:0>Instruction format

Instruction<15:0>\i  
 Op<15:13> := i<15:13>  
 unused<2:0>:=i<12:10>  
 Effective-address<9:0>\z:=i<9:0>

Switches

Start key  
 Continue key  
 Run-switch

switch-actions

Start-key ⇒ (P ← 0; Interpret)  
 Continue-key ⇒ (Interpret)

Interpreter

Interpret :=  
 (i ← M[P]; P ← P+1; next  
 Instruction-execution; next  
 Run-switch ⇒ Interpret)

fetchexecute

The main virtue of ISP, and indeed the reason to introduce it to the reader, is the fact that it is a specification for both Instruction-sets and complete computers. An ISP description is usually completely independent of an implementation, thus serving as a definition of the computer, whether using an RTM, a microprogram, a conventional register technology implementation, or even a simulation program in Fortran. By adding more detail to the ISP definition an implementation can be implied.

A computer is completely defined at the programming level by giving both its instruction set and its interpreter in terms of basic operations, data types, and the system's memory. For clarity the ISP description is usually given in the following order:

Declare the system's memory:

Processor state (the information necessary to restart the processor if stopped between instructions, e.g., general registers, program counter, index registers).

Primary memory state (the program and data memory directly addressable from the processor).

Console state (any external keys, switches, lights, etc., that affect the interpretation process).

Secondary memory (the disks, drums, dectapes, magnetic tapes, etc.).

Transducer state (memory available in the peripherals that is assumed in the instructions of the processor).

Declare the instruction format.

Define the operand address calculation process.

Declare the data types.

Declare the operations on the data-types.

Define the instruction interpretation process including interrupts, traps, etc.

Define the instruction set and the instruction execution process (provides an ISP expression for each instruction).

Thus, the computer system is described by first declaring memory, data-types and primitive data operations. The instruction interpreter and the instruction-set is then defined in terms of these memory entities. In this sense, the ISP notation is similar to that used in higher level programming languages. Its statements define entities by means of expressions involving other entities in the system.

ISP expressions are inherently interpreted in parallel, reflecting the underlying - parallel nature of hardware operations. This is an important difference between ISP and standard programming languages, which are inherently serial. For example, in

$$\text{JMS} := (P \leftarrow z; L \leftarrow P)$$

both righthand sides of the data transmission operators ( $\leftarrow$ ) are evaluated in parallel. The values are then transmitted. Thus the old value of P would go to L, and the new P would contain z. Serial ordering of processing is indicated by using the term "next". For example,

$$\text{ISZ} := (M[z] \leftarrow M[z]+1; \text{next} \\ (M[z]=0) \rightarrow P \leftarrow P+1)$$

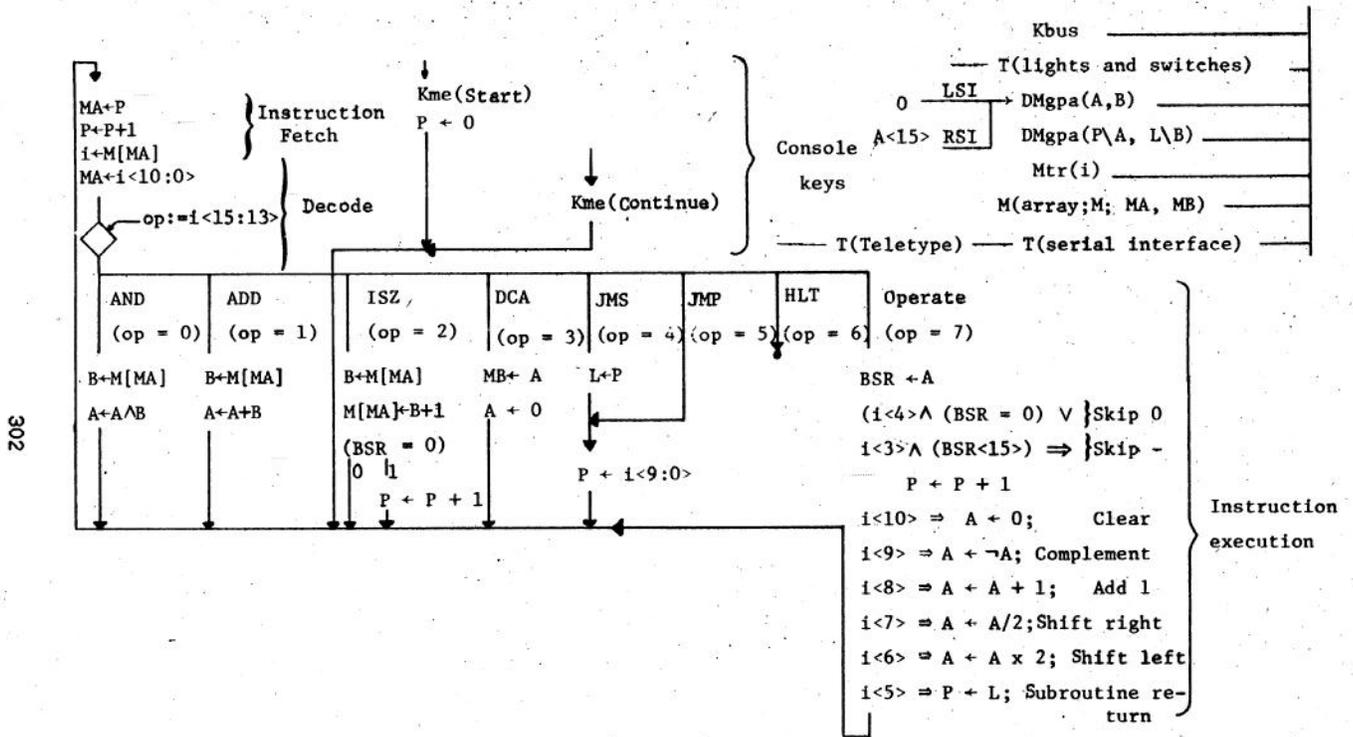


Fig. Crtm-5. RTM diagram of Crtm-1.

needed in the interpretation of the instructions are:  $i$ , an instruction holding register; and  $B$ , a temporary register used for binary operations on  $A$  (e.g., ADD, AND). Also connected to the RTM Bus are the read-only and read-write memories and the Teletype, as well as a special input/output register interface to the remainder of the system. Instructions for manipulating these are left as an exercise for the student.

The method of defining the interpreter can be seen from the RTM diagram (Figure Crtm-5). The control part consists of four sub-parts: the start and continue keys, which are used to initialize the processor to start at location 0 and to restart the processor; the instruction fetch; the instruction decoder; and the instruction execution. The instruction fetch consists of picking up the instruction from the memory word addressed by the program Counter,  $P$ , and incrementing  $P$  to point to the next instruction. The instruction is placed in the register, which has been specially wired to a Kb8 to allow decoding of the three most significant bits. Individual bits of  $i$  can be tested for the Operate (OPR) instruction, and the address field,  $i\langle 9:0 \rangle$ , can be read.

After the instruction is fetched and placed in  $i$ ,  $Ke(MA\langle i\langle 9:0 \rangle \rangle)$  is evoked to address data referenced by the instruction. Immediately following this evoke operation,  $K(\text{branch}; 8\text{-way})$  allows control to move to the one path corresponding to the operation code of the instruction being interpreted -- that is, the instruction is decoded and control is transferred to execute it. After the execution of the appropriate instruction, control returns to fetch the next instruction. For example, executing the ADD (two's complement add) instruction consists of loading the data from memory into the temporary register,  $B$  (i.e.,  $B\langle MB \rangle$ ) and then adding  $B$  to the accumulator,  $A$  (i.e.,  $A\langle A+B \rangle$ ).

For the Operate instruction, which does not reference memory, each bit of the address part of the instruction specifies an operation to be carried out on the accumulator (test for - or 0, clear, complement, add one, shift right or left, or return from the subroutine). Each bit is tested in the sequence shown, and if a one, the corresponding operation is carried out. If the instruction code with  $op = 6$  is given, the computer halts; pressing continue restarts it.

The instruction set shown above is straightforward and simple. Subroutine return addresses are stored in a link register,  $L$ . Thus to call subroutines at a depth of more than one level requires the called subroutine to save the link register in a temporary location. But there is no way of storing this register; thus it is difficult to call a subroutine and pass parameter information (e.g., the location of data). A problem is given to correct these design faults.

## USING ISP TO DEFINE THE COMPUTER

The ISP notation can be used alone, in a linear text fashion, as an alternative description of the stored program computer. In fact, ISP was initially developed solely for the purpose of defining the instruction-set of a computer. The difference between an ISP and an RTM description appears fairly slight, with only a few more RTM actions required to implement the ISP. Also, an ISP description usually does not imply

an implementation or structure. The main difference between the ISP and RTM description occurs because ISP is fundamentally a 1- dimensional or linear-text representation. For some descriptive applications this shortens the description, while in other cases the two are nearly the same size. The reader should not have a serious problem in understanding the ISP description because the various RTM register declarations and actions used in this book have been borrowed from ISP. The ISP description of Crtm-1 is given in Figure Crtm-6. In this description a Run-switch has been added for manually interrupting the computer.

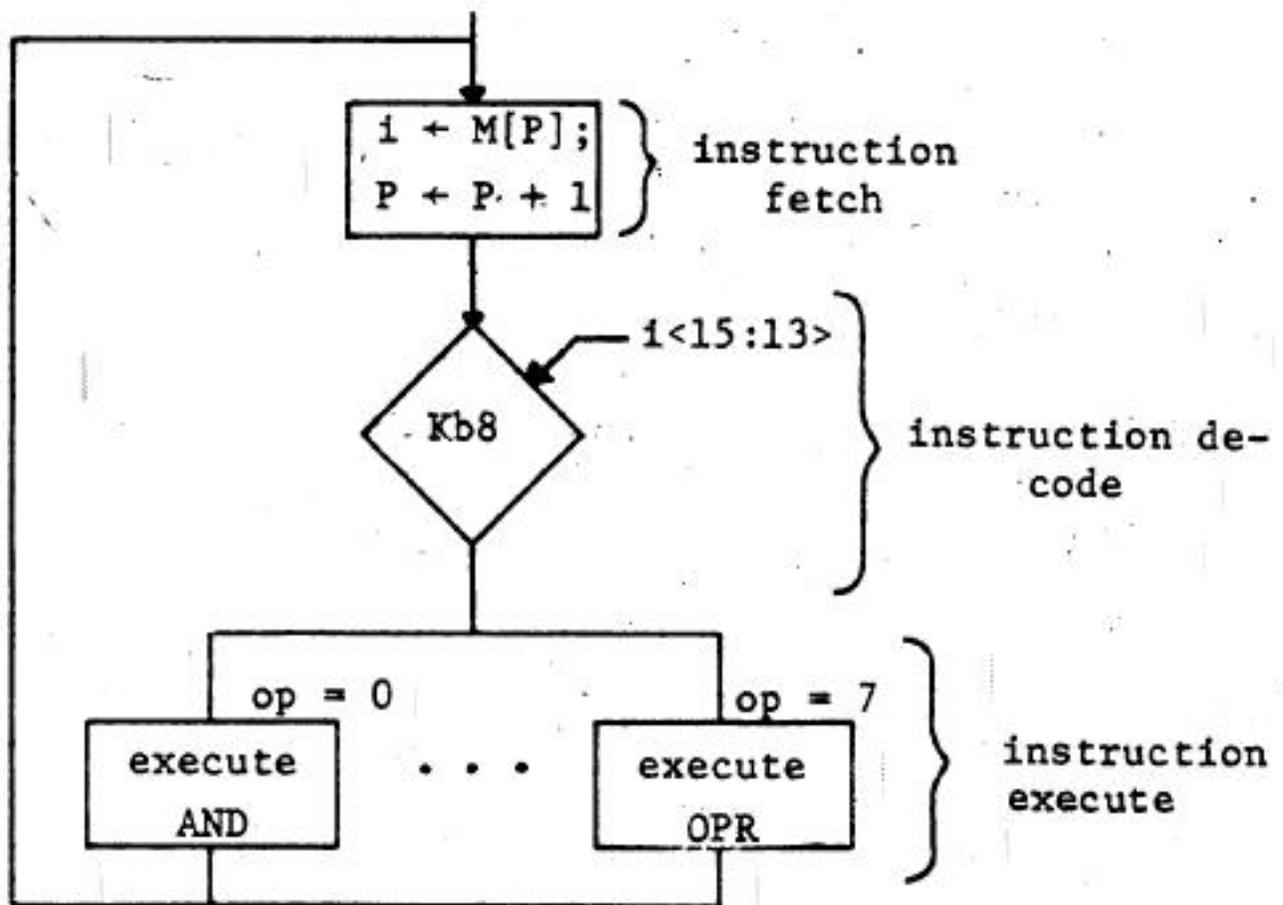


Fig. Crtm-4. Control part showing Crtm-1 fetch, decode, and execute processes for instruction interpretation.

## RTM IMPLEMENTATION OF Crtm-1

Figure Crtm-5 shows the RTM diagram for the small general purpose, stored program computer that we have been calling Crtm-1. It was initially constructed as an application experiment to demonstrate the feasibility of RTM's and to investigate systems problems. The process of specifying the machine took approximately two hours (with three people). The computer was wired, and aside from minor system/circuit problems (for which the experiment was designed) the computer operated essentially when power was supplied, since there were no logic errors.

The computer was designed for an actual application which had 300 constants, 600 control steps and 16 variables. (An alternative approach would have been to hard-wire the 600 control steps directly, thereby operating faster, but being more expensive and less flexible.) The computer has only 24 K(evoke) and 10 K(branch) modules. (By way of comparison, RTM interpreters require about 90 control modules, 2 DMgpa's, and core memory to emulate the PDP-8 minicomputer.) Since the price ratio of a single

hardwired control to a single read-only memory control word is approximately 10:1, and since the overhead price of a 1000-word read-only memory is about the same as 100 controls, it was cheaper in the above application to use RTM's to first build an interpreter, (i.e., a stored program digital computer) and then let the computer program execute the control steps.

The data part of the machine is shown in the upper right of Figure Crtm-5. Three DM-type RTM modules hold the processor state and temporary registers. The processor state, that part of memory accessible to and controlled by the program, includes: A, the accumulator; P, the program counter; and L, the register used to hold subroutine return addresses (links). The temporary registers

Assembly language format	memory cell number in octal	value of memory cell	equivalent action	direct action
Start, Clear	10	111 001 0000000000	A←0	} z ← x + y
ADD x	11	001 000 0001000000	A←A+M[100]	
ADD y	12	001 000 0001000001	A←A+M[101]	
DCA z	13	011 000 0010000000	M[200]←A; A←0	
⋮				
x,	3	100 000 0000000011	-	
y,	4	101 000 0000000100	-	
⋮				
z	-	200 ? before; 7 after	-	

Fig. Crtm-3. Program for Crtm-1 computing  $z \leftarrow x + y$ .

1. The interpreter picks up (fetches) the instruction in 10, loading it into K(interpreter). P is increased by 1, becoming .11.
2. K(interpreter) determines that the instruction is an OPR type (i.e., 111 001 0000000000) and furthermore proceeds to execute the proper register transfer called for. This causes A to be cleared (i.e.,  $A \leftarrow 0$ ).
3. The interpreter fetches the instruction addressed by P (i.e., 11). P is increased by 1 (P becomes 12).
4. The interpreter carries out (executes) the instruction in 11 (i.e., 001 000 0001000000), which is ADD 100. This causes data in memory location 100 to be added to A. Now, A contains a 3.
5. The instruction ADD 101 in location 12 is fetched, and P is advanced to 13.

6. The instruction is executed causing a 4 to be added to A. A now contains 7.
7. The instruction, DCA 200, in location 13 is fetched, and P advanced to 14.
8. The instruction is executed, causing a 7 to be placed in location 200. A is reset to 0.
9. The instruction in location 14 is fetched and the process continues.

This process of fetching an instruction, determining what it is (i.e., decoding it), and then executing it, is called instruction interpretation, and can be expressed by a flowchart as shown in Figure Crtm-4. There are only 1+1+8 basic boxes.

$$(i\langle 3 \rangle \wedge A\langle 15 \rangle) \rightarrow (P \leftarrow P+1)$$

Clear A microcoded operate instruction. Reset the Accumulator to 0. The micro-operation is selected by bit 10. The ISP expression defining clear A is:

$$i\langle 10 \rangle \rightarrow A \leftarrow 0$$

Complement A microcoded operate instruction. Complement the Accumulator and place the result back in the Accumulator. The micro-operation is selected by bit 9. The ISP expression defining complement A is:

$$i\langle 9 \rangle \rightarrow (A \leftarrow \neg A)$$

Add 1 to A microcoded operate instruction. Add a 1 to the Accumulator and place the result back in the Accumulator. The micro-operation is selected by bit 8. The ISP expression defining add 1 to A is:

$$i\langle 8 \rangle \rightarrow (A \leftarrow A+1)$$

Shift right A microcoded operate instruction. Shift the Accumulator right 1 position; i.e., divide the Accumulator by 2, and place the result in the Accumulator. This is an arithmetic shift (i.e., the sign is shifted in). The micro-operation expression defining shift right A is:

$$i\langle 7 \rangle \rightarrow (A \leftarrow A/2\{\text{arithmetic}\})$$

Shift left A logical microcoded operate instruction. Shift the Accumulator left 1 position, i.e., like multiplying by 2, and place the result in the Accumulator. This is a logical shift since a 0 is shifted in. The micro-operation is selected by bit 6. The ISP expression defining shift left logical A is:

$$i\langle 6 \rangle \rightarrow (A \leftarrow AX2\{\text{logical}\})$$

Return from subroutine microcoded operate instruction. Place the subroutine Link register in the Program Counter. This causes the computer to jump back to where a JMS instruction was given. The micro-operation is selected by bit 5. The ISP expression for defining return from subroutine is:

$$i \langle 5 \rangle \rightarrow (P \leftarrow L)$$

## Instruction-Interpreter

Having given the definition of the computer's memory and registers, together with the definition of each of the instructions, only the process for interpreting the instruction set remains to be described. The K(interpreter) part was included in Figure Crtm-1. Before defining the behavior of the interpreter more formally, let us look briefly at a program stored in Crtm's memory (see Figure Crtm-3). This program is 4 instructions long and begins in location 1018 (all numbers shown are base 8). It assumes that there are three additional variables, x, y, and z, held in locations 100, 101, and 200. If we start the program in location 10, by somehow having the Program Counter\P contain 10, the following behavior is observed:

298

[previous](#) | [contents](#) | [next](#)

$$(M[z]=0) \rightarrow P \leftarrow P+1$$

Deposit and Clear Accumulator (DCA) instruction. The Accumulator is stored.. in the memory word addressed by the effective address. The Accumulator is then reset to hold the value 0. The instruction is selected when the opcode is 3. The ISP expression defining DCA is:(1)

$$DCA(:= op = 3) \rightarrow (M[z] \leftarrow A; A \leftarrow 0)$$

Jump to Subroutine (JMS) instruction. Place the Program Counter in the Subroutine Link register. Next, reset the Program Counter to the value specified by the effective address. The instruction is selected when the opcode is 4. The ISP expression defining JMS is:

$$JMS(:= op = 4) \rightarrow (L \leftarrow P; P \leftarrow z)$$

Jump (JMP) Instruction. Reset the Program Counter to the value specified by the effective address. The instruction is selected when the opcode is 5. The ISP expression defining JMP is:

$$JMP(:= op = 5) \rightarrow (P \leftarrow z)$$

Halt (HLT) instruction. Stop the computer by not causing any subsequent action. The instruction is selected when the opcode is 6. The ISP expression defining HLT is:

$$HLT(:= op = 6) \rightarrow$$

Operate(OPR) microcoded instructions. The OPR instruction is actually a set of instructions which can be given by placing ones in various bit positions to cause certain actions called micro-operations. Any number of micro-operations can be executed in a single OPR instruction. The instruction is selected when the opcode is 7. The ISP expression defining OPR is:

$$OPR(:= op = 7) \rightarrow \text{begin operate}$$

Micro-operations are executed in the sequence described below:

Skip if A=0 microcoded operate instruction. If the Accumulator is zero, a 1 is added to the Program Counter causing a skip. The micro-operation is selected by bit 4. The ISP expression defining skip if A=0 is:

$$(i < 4) \wedge (A = 0) \Rightarrow (P \leftarrow P + 1)$$

Skip if A is negative microcoded operate instruction. If the Accumulator is negative, a 1 is added to the Program Counter, causing a skip. The micro-operation is selected by bit 3. The ISP expression defining skip if A negative is:

-----

1. The semicolon in the instruction indicates that the operations can be executed perfectly in parallel. That is,  $M[z]$  gets the old value of A, not the zero. This type of parallelism is easily achieved using registers composed of edge triggered or master slave flip flops. If a definite sequence of actions is implied. The word "next" is used following the semicolon.

the instruction format. There are three main parts: the 3-bit operation code, bits <15:13>, which define one of 8 instructions; 3 unused bits <12:10>; and ten address bits, <9:0>, which specify the location in memory of an operand. This address of the location is called the effective address.

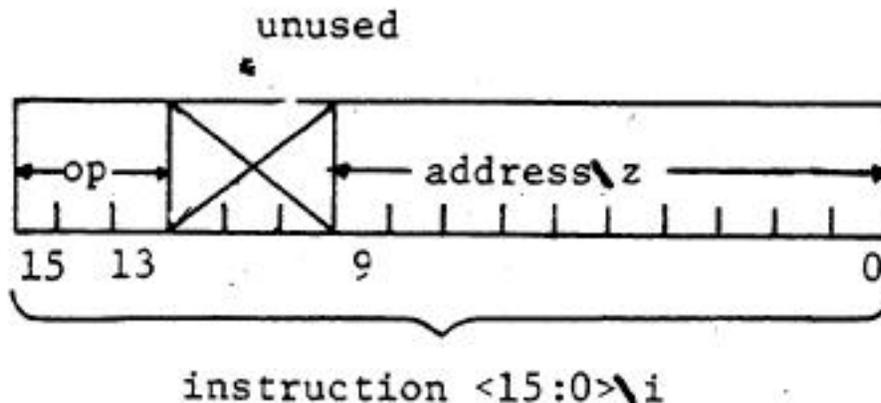


Fig. Crtm-2. Instruction format for Crtm-1.

The particular assignment of operation code bits to the eight possible instructions, and the behavior of the instructions will be given below. The complete set of instructions is called the Instruction-set. To describe the action of the instructions, the ISP notation will be used, which gives both the definition of the value of the instruction code and the behavior of it. The ISP format is: Instruction-name (:= op = value) => (action of instruction). There is a name for the instruction, a value for the op code corresponding to the name, and each instruction has a certain action (behavior) expressed as a series of register transfer operations. (The complete ISP description is given in a later section.)

### Instruction-Set

The instruction-set for Crtm-1 is:

Logical AND instruction. The operand in memory accessed by the effective address is ANDed with the Accumulator and the result is stored in the Accumulator. The instruction is selected when the opcode is 0. The ISP expression defining ADD is:

$$\text{AND}(\text{:= op} = 0) \Rightarrow (A \leftarrow A \wedge M[z])$$

Two's Complement Add (ADD) instruction. The operand in memory accessed by the effective address is added to the Accumulator and the result is stored in the Accumulator. The instruction is selected when the opcode is 1. The ISP expression defining ADD is:

$$\text{ADD}(\text{:= op} = 1) \rightarrow (A \leftarrow A + M[z])$$

Increment and Skip if Zero (ISZ) instruction. The operand in memory accessed by the effective address is added to a one, and the result is stored back in memory. If the result is zero, a one is added to the Program Counter causing the next instruction to be skipped. The instruction is selected when the opcode is 2. The ISP expression defining ISZ is:

$$\text{ISZ}(\text{:= op} = 2) \rightarrow (M[z] \leftarrow M[z] + 1; \text{next})$$

296

[previous](#) | [contents](#) | [next](#)

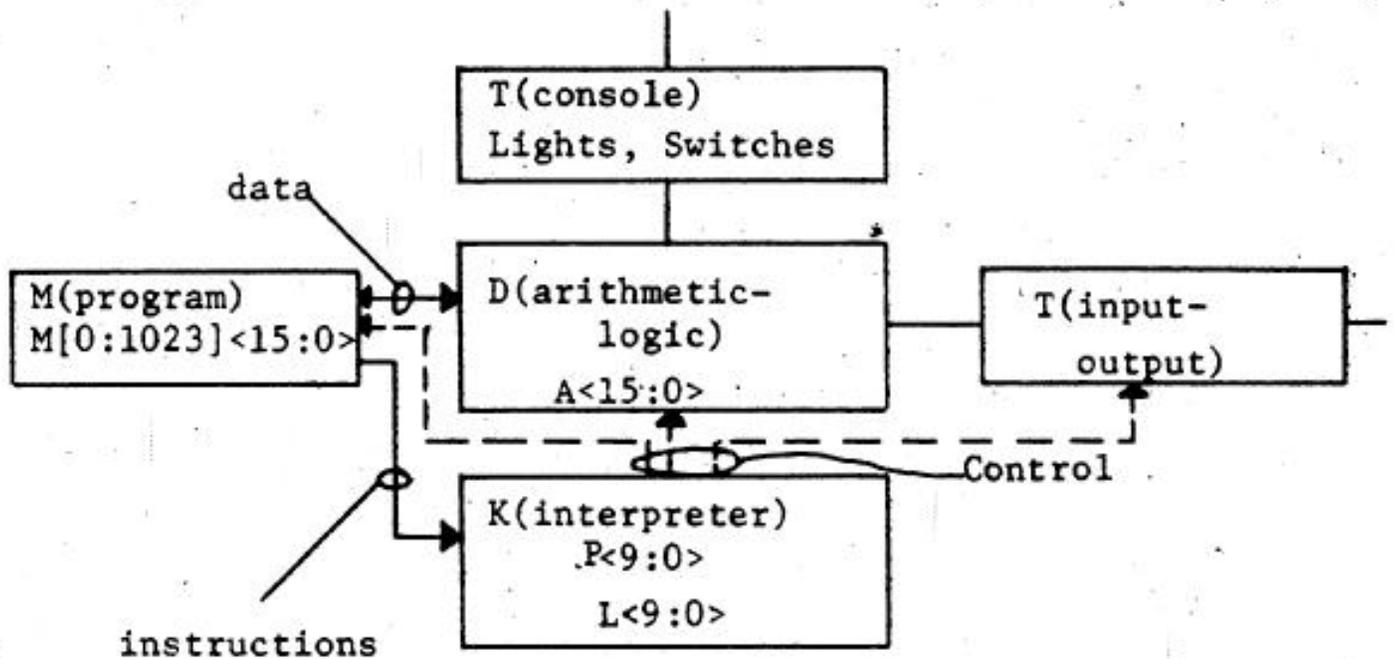


Fig. Crtm-1. PMS diagram showing four basic components of a general purpose stored program computer.

## Registers and Memory

In describing a computer, the common method is to describe first all the registers within the above parts which are accessible by the program (i.e., all the registers that can be affected by an instruction).

Crtm-1 has the following registers:

**A\Accumulator<15:0>** - a 16-bit register in the arithmetic-logical unit which holds intermediate results of calculations. In Crtm-1, A is the only arithmetic register which can be manipulated under program control.

**P\Program-Counter<9:0>** - A 10-bit register which is part the computer's control component, and which points to (accesses) the next instruction to be carried out.

**L\Link<9:0>** - The subroutine link register. This register is part of the control component and holds the return address when a subroutine is called. In this way, the program knows where to come back to at the completion of a subroutine.

**M[0:1777v8]<15:0>** - The array memory which holds the program and the data which the program operates on.

**Data-switches<15:0>** - A 16-bit external switch register, which can be read into the A register under program control (i.e., gets data into the machine). Also, the switch register, can tell the computer at

which location in memory to start.

### Instruction Format

Instructions in Crtm-1 manipulate the above registers and program memory. As has been outlined already, the control component contains a mechanism for fetching instructions pointed to by the Program Counter. These instructions are then interpreted and operations are performed.

The instructions are stored in M. Figure Crtm-2 shows the assignment (meaning) of bits in the instruction word. This assignment of bits is usually called

system (in this case a computer). Up until now, the RTM diagram has provided a two-dimensional representation schema for behavior and structure. The reader has not been burdened *with* learning a register-transfer language because they are quite cumbersome for expressing digital systems. Since most-computers behave simply, a one-dimensional representation is usually adequate.

The motivation for introducing ISP notation is that it serves as the formal basis for defining computers. A computer can be defined precisely and relatively easily in ISP. The section on the PDP-8/RTM is based on the ISP description; the specification in ISP is converted into an RTM implementation in a relatively formal way. ISP's of many larger computers (e.g IBM 7090, CDC 6600) can be found in Bell and Newell (1971).

## COMPUTER SIMULATION

An ISP description can also be converted to a programming language (e.g., Fortran) rather easily, so that a computer may be simulated, and the design and description verified. Subsequent design problems are given to describe computers in ISP, carry out ISP-to-RTM, and ISP-to-Fortran conversions.

## COMPUTER INTERFACING

A section presents several extended RTM's for use in interfacing an RTM system to a computer. With these interfaces, it is possible to build combined systems in which some operations are carried out within the computer (i.e., programmed) and some operations are done in the specialized system (hardware). Thus, the hardware-software trade-off can be exploited in a design.

### Crtm-1: A SMALL, GENERAL PURPOSE, STORED

### PROGRAM COMPUTER DESIGNED USING RTM'S

In this section, we present three aspects of computers using Crtm-1. -The first section describes the computer in a conventional way, without regard to the fabrication. The second section describes the implementation using RTM's; hence, the reader has another opportunity to understand how the computer operates. Finally, the structure and behavior of Crtm-1 are defined again using ISP. This last section will be referenced in subsequent sections for designing computers based on their ISP descriptions.

## DESCRIPTION OF CRTM- 1

A computer is usually considered in terms of a structure constructed from the four components shown in Figure Crtm-1; these are: a D for carrying out arithmetic and logical operations; an M for storing the

program and information (data) for the program; a T, for communicating with the outside world, which includes a console for direct human control of the computer; and a K, the interpreter, which defines the behavior of the components through the use of instructions. The interpreter operates by picking up (i.e., fetching) instructions from the computer's memory, examining (i.e., decoding) them and then carrying out (i.e., executing) the operation they specify. After each instruction is interpreted, a register within the interpreter, called the Program Counter (also called the instruction address, the instruction location counter, etc.), selects (accesses) the next instruction to be interpreted. Control then repeats the fetching-decoding-execution process for this new instruction. This simple 3 step sequential process is basically how all current, general purpose, stored program digital computers operate.

## CHAPTER 6 COMPUTER DESIGN EXAMPLES

The goal of this chapter is to give the reader insight into the operation of computers. Again RTM's are used -- here for fabricating computers. That we have used RTM's before in a slightly different context does not seem to limit our ability to present new concepts with them. In fact, since we use a fixed set of components, now well-known to the reader, it is feasible to carry out the description and analysis at a much deeper level than when we have to worry about implementation details. In describing a computer we can show the details of the structure and behavior by allowing the reader to look at the complete internal structure of the computer.

### COMPUTERS

Several computers are presented for various reasons. Holding with the policy that repetition reinforces learning, we also include at least two problems to reinforce concepts. Three C's are given: the Crtm-1, the first minicomputer constructed from RTM's; the PDP-8/RTM, based on a common minicomputer, the PDP-8; and the PDP-16/M, a subminicomputer which is microprogrammed.

The Crtm-1 is given both for historical and pedagogical reasons. Besides having been the first, it is nearly the simplest computer that can be constructed from RTM's, requiring only about 200 control wires. Because it is so simple, we believe that anyone who is unfamiliar with general purpose stored program computers can understand them using Crtm-1 as a model. Computers of about its size take about 8 to 12 hours to define, document, wire, and get operational. Therefore, we hope every reader who has not had the joy of designing and implementing a language or a computer will take the time to design and build one with RTM's, or at least carry out the paper exercise. He should then spend at least as long writing programs for it, and possibly re-iterate the design.

The PDP-8/RTM is given to provide an example of an existing minicomputer, which was designed independently (actually, long ago in 1965) of the work in RTM's. Educational examples are almost always oversimplified. This is especially true of computers, since they have many seemingly irrelevant details. But these details are in fact an essential part of the design -- of what makes the computer useful in the real world. Consequently, it is important to design and analyze an actual computer.

The PDP-16/M is presented because in applying the PDP-16 (i.e., RTM's) to a problem, a 16/M configuration might be the best solution. Also, since the 16/M is like a microprogram computer with no higher level instruction-set, it gives another view of microprogramming.

In terms of gradual transition from hardwired RTM systems to fully developed computers, the order should be 16/M, CRTM-1, PDP-8/RTM. We place the 16/M at the end in this chapter for reasons of pedagogy.

Several additional small computers are described briefly as problems: a general register structure machine, a desk calculator, and a machine using the DMar.

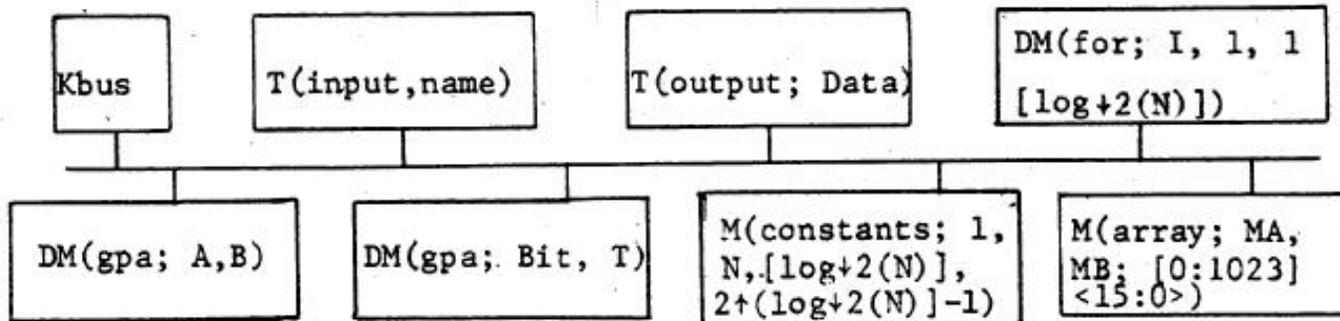
## ISP NOTATION, AND ISP TO RTM CONVERSION

In describing Crtm-1, a notation for computer instruction-set description, ISP, is introduced. ISP should be relatively familiar by now, since it is the programming-language-like notation used in the RTM flowchart and data part diagrams. By using the notation, a one-dimensional text string (such as in conventional programming) expresses the structure and behavior of a digital

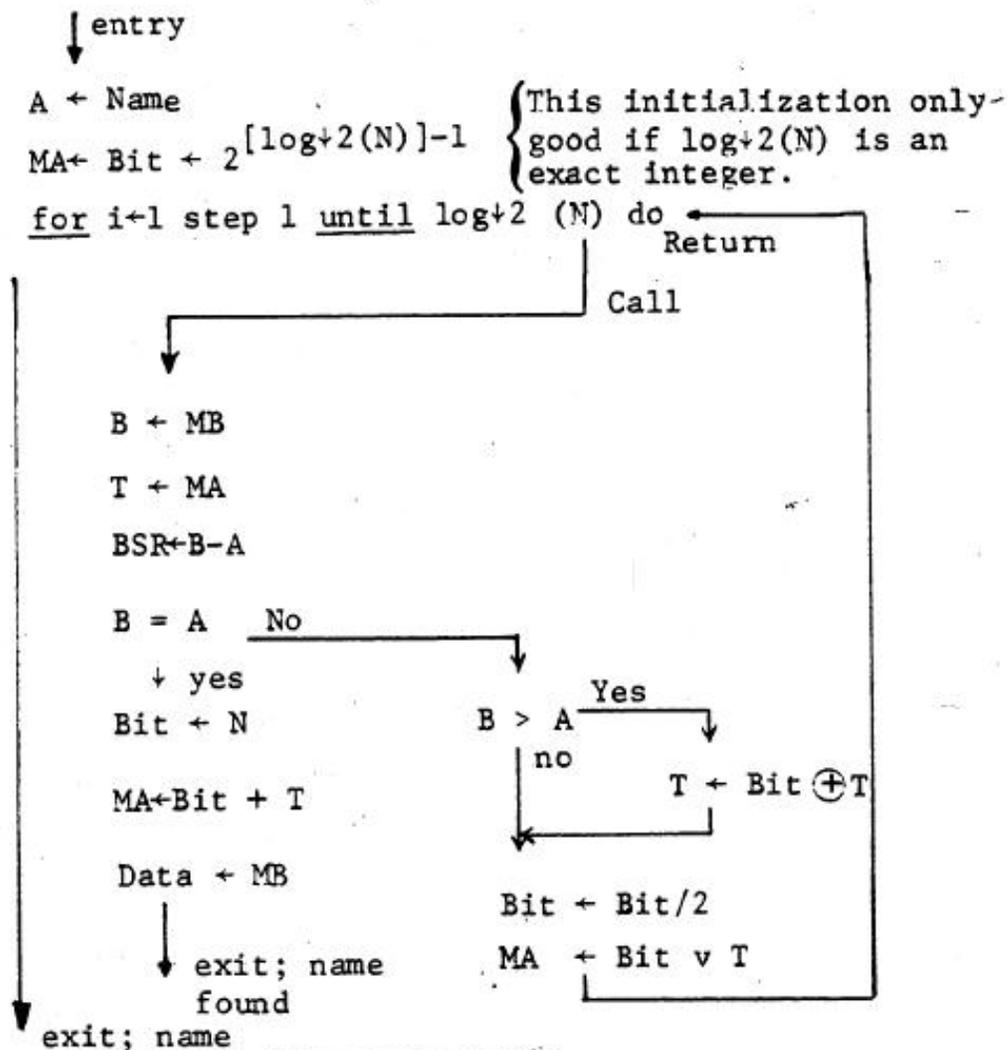
[previous](#) | [contents](#) | [next](#)

3. Extend any of the search schemes to access all entries between a starting entry name, Name-lo, and an ending entry name, Name-hi. Develop a scheme which accesses data by entry name and attribute.
4. The initialization scheme for the binary search strategy implemented in Figure MCA-8 guides the search properly only if  $\log_2(N)$  is an exact integer. Devise an initialization scheme that works for any N.
5. Design an M(content addressable) that uses a hash function. Implement the operations shown in Figure MCA-1. Predict the performance of your system as a function of the fullness of the memory.

[previous](#) | [contents](#) | [next](#)

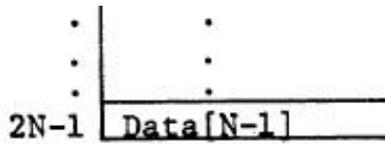


a. Data part



b. Control part

not found 0	Entry-name[0]
1	Entry-name[1]
.	.
.	.
.	.
N-1	Entry-name[N-1]
N	Data[0]
N+1	Data[1]
.	.
.	.



c. Memory organization

Fig. MCA-8. RTM diagrams of control part, data part and memory layout for an M(content addressable) with a binary search strategy.

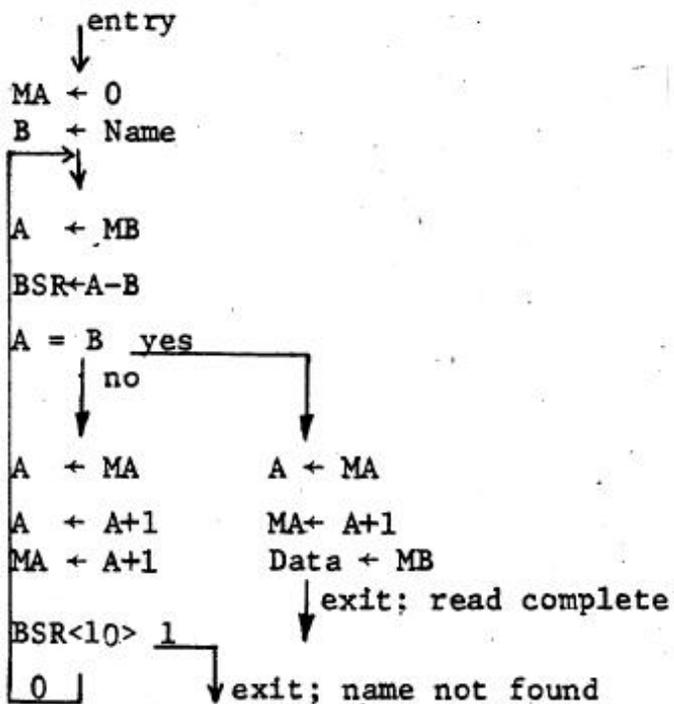


Fig. MCA-7. Basic flowchart for a binary search strategy on an increasing order list of entry names.

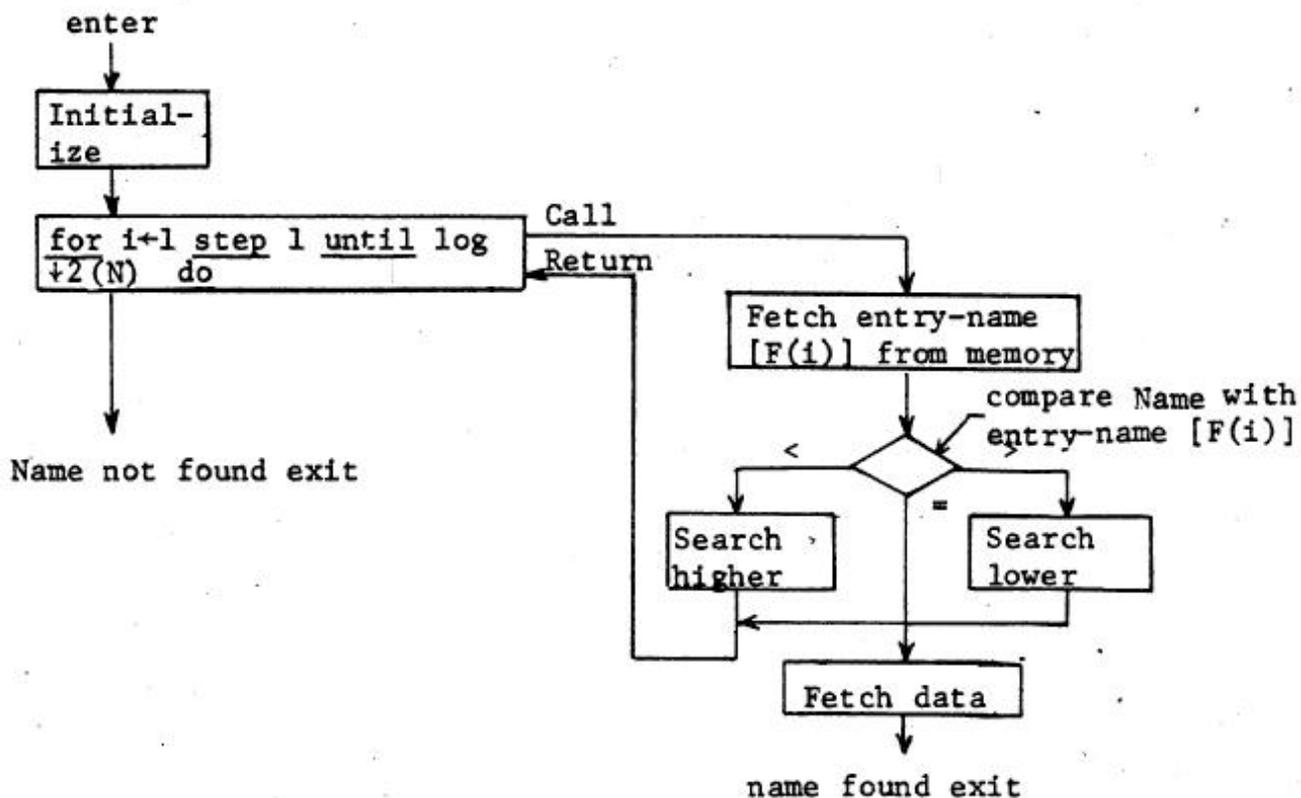


Fig. MCA-6. RTM diagram of the control part for a sequential search of a read-only CAM with unordered data.

only CAM with unordered data.

290

[previous](#) | [contents](#) | [next](#)

## Solution 2 - A Content Addressable Memory Using a Binary (Logarithmic) Search Strategy.

The design of an effective CAM requires that the search time be as small as possible. In the previous example, the data was unordered and the average number of comparisons to a match was  $N/2$ . However, if the data can be kept ordered, the search can be carried out with at most  $\log_2(N)$  (the integer part) comparisons for a CAM with  $N$  entry names. This technique is useful for example for converting analog inputs to digital values or for any table look-up scheme (e.g.,  $\sqrt{x}$ ,  $\sin(x)$ ).

Since the data is monotonic, a series of trials regarding the location of the desired entry name can be made. The trials are ordered such that each trial divides the search space in half and the succeeding trial is above or below the current point depending on the relationship between the entry name at the division point and the entry name being accessed. Figure MCA-7 shows the overall control strategy for a binary search. The main loop is controlled by a  $K$  (for loop) which steps through the search space for up to  $\log_2(N)$  trials and assumes that the entry names are stored in increasing order. The first trial compares the desired entry name (Name) with the  $N/2$  entry name in the memory. The search terminates on a match; otherwise, the  $N/4$  entry name or the  $3N/4$  entry name is checked, depending on whether Name was greater or less than  $N/2$ . The search continues in this fashion until a match is found or until the List has been exhaustively subdivided.

Figure MCA-8 shows the data part, control part, and memory organization for an M(content addressable) with a binary search scheme.

## Solution 3 - A Content Addressable Memory Using Hash Coding for Direct Addressing.

There is another accessing scheme, called hash code addressing which would provide faster access than either the linear or binary search. A hash function,  $H$ , is applied to an entry name to produce an address,  $H(\text{Name})$ , of a physical location in the memory space. For the design presented here, the entry name is a sixteen-bit word which must be mapped into a ten-bit address. The purpose of the hash function is to provide rapid access to unordered data with little or no searching. The memory can be organized as in Figure MCA-2, but any empty cells must have blank entry name fields.

The choice of the hash function depends on the distribution of the names in the address space. A desirable hash function is one which randomly distributes the entry names throughout the address space without bunching the entry names in dense groups. A collision occurs when two entry names are mapped into the same address. The object of the hash function is to avoid collisions; However, when a collision does occur, some scheme must be developed to provide an alternative location for the new entry name. A simple scheme is to start from- the collision point and consider successive addresses until an available slot is found.

When reading from the memory, the hash function is applied to the entry name and if the entry name at

that address does not match the given name, then whatever scheme was applied to write into the memory is used to search for the given entry name. If a blank entry is encountered before the entry name is found, the name does not exist in the memory.

## ADDITIONAL PROBLEMS

1. For both Solution-1 and Solution -2, implement all of the operations shown in Figure MCA-1.
2. Design an M(content addressable) which uses a linked list to organize the memory.

## SOLUTIONS

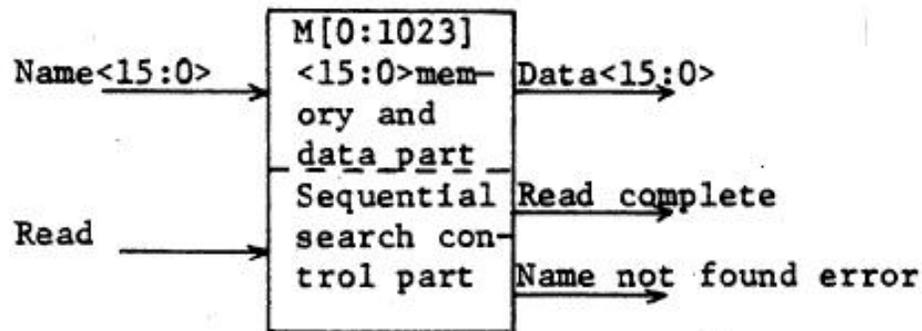


Fig. MCA-4. PMS diagram of the general structure of a read-only, content addressable memory with a sequential search strategy.

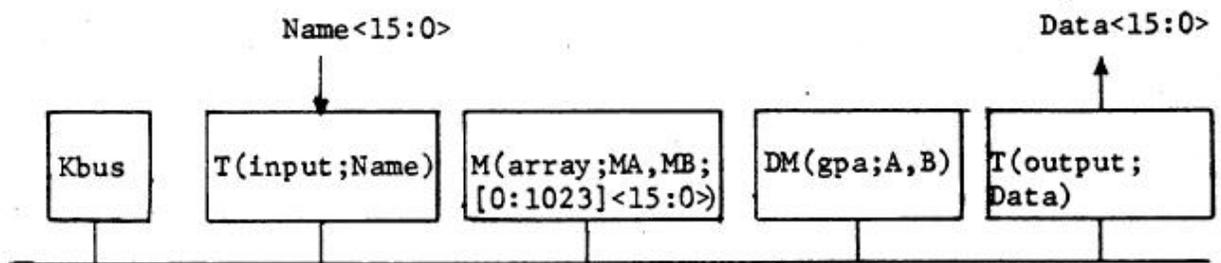


Fig. MCA-5. RTM diagram of the data part of a read-only CAM with a sequential search strategy.

The solutions presented will be given in terms of the search strategy employed in accessing the memory.

#### Solution 1 - A Content Addressable Memory Using a Sequential Search Strategy.

The design presented here will be a read-only, content addressable memory based on a 1024 word  $M(\text{array})$ , using the organization scheme shown in Figure MCA-2. This scheme provides for 512 sixteen-bit entry names and 512 sixteen-bit data entries. The entry names will appear unordered and will be unique. When a read command occurs, a register,  $Name$ , contains the entry name to be accessed; at the completion of the search, the data associated with the entry name is placed in a register,  $Data$ . If the entry name is not found in the memory, an error exit is taken. The general structure of the system is shown in Figure MCA-4 and the specific data part is shown in Figure MCA-5.

The search scheme is straightforward and is shown in Figure MCA-6. The search steps through the physical memory addresses and at each step checks for a match. The search is terminated when a match is found or when the entire memory has been searched. The average number of comparisons before a match is 256. We use the operation 4-MA here for compactness of presentation, although strictly speaking this operation is not available in RTM memories.

[previous](#) | [contents](#) | [next](#)

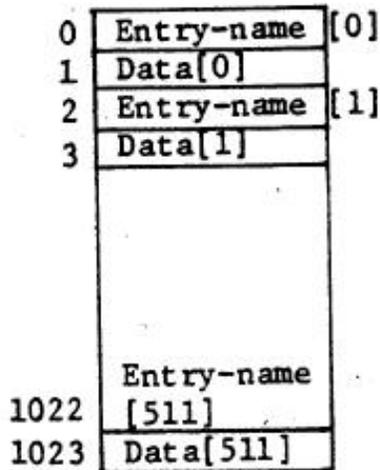


Fig. MCA-2. Example of mapping of content addressable memory cells into a physical memory array.

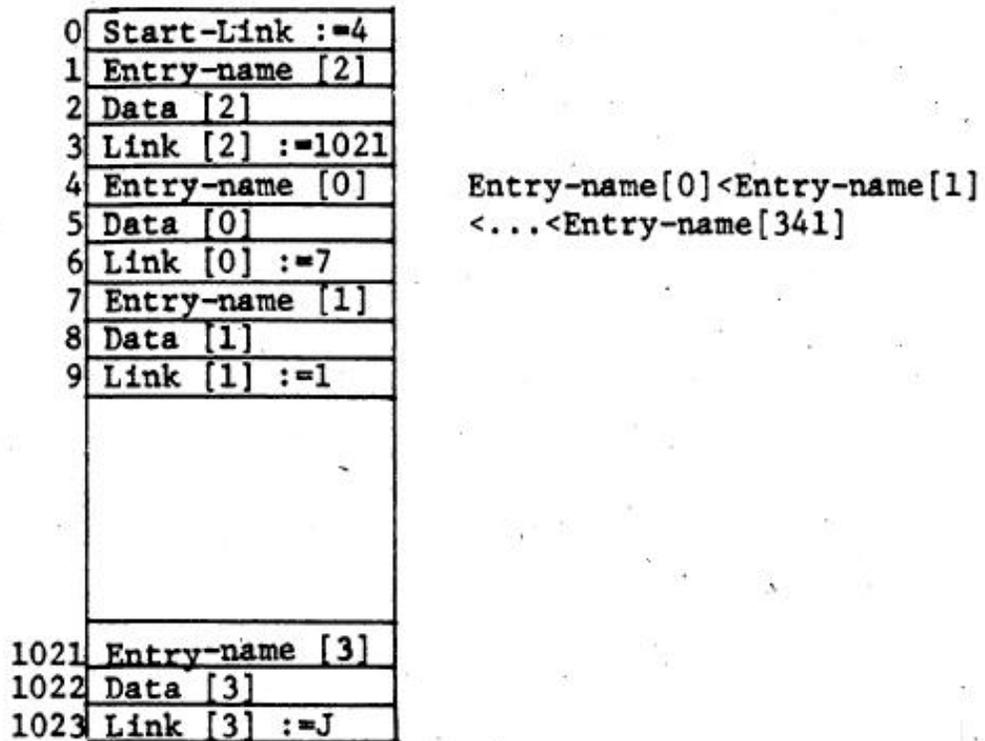


Fig. MCA-3. Example of a linked list ordering of content addressable memory cells.

[previous](#) | [contents](#) | [next](#)

Figure MCA-1 shows the general structure of a system with these operations. The memory might be organized in one of the following ways:

1. Linearly-unordered - Each time a new entry name is to be written into the memory, it is appended to the existing list as an entry name - data pair. Figure MCA-2 shows such a memory organization.
2. Linearly-ordered - Each time that a new entry name is to be written into the memory, it is added so that the entry names appear in some fixed order. The same entry name-data pair could be used as in Figure MCA 2, except that the entry names would be ordered. The simplest scheme for entering data in an ordered list is to find the proper place for the entry name-data pair, then move all entries which appear after the new entry down two spaces in memory and add the new data.

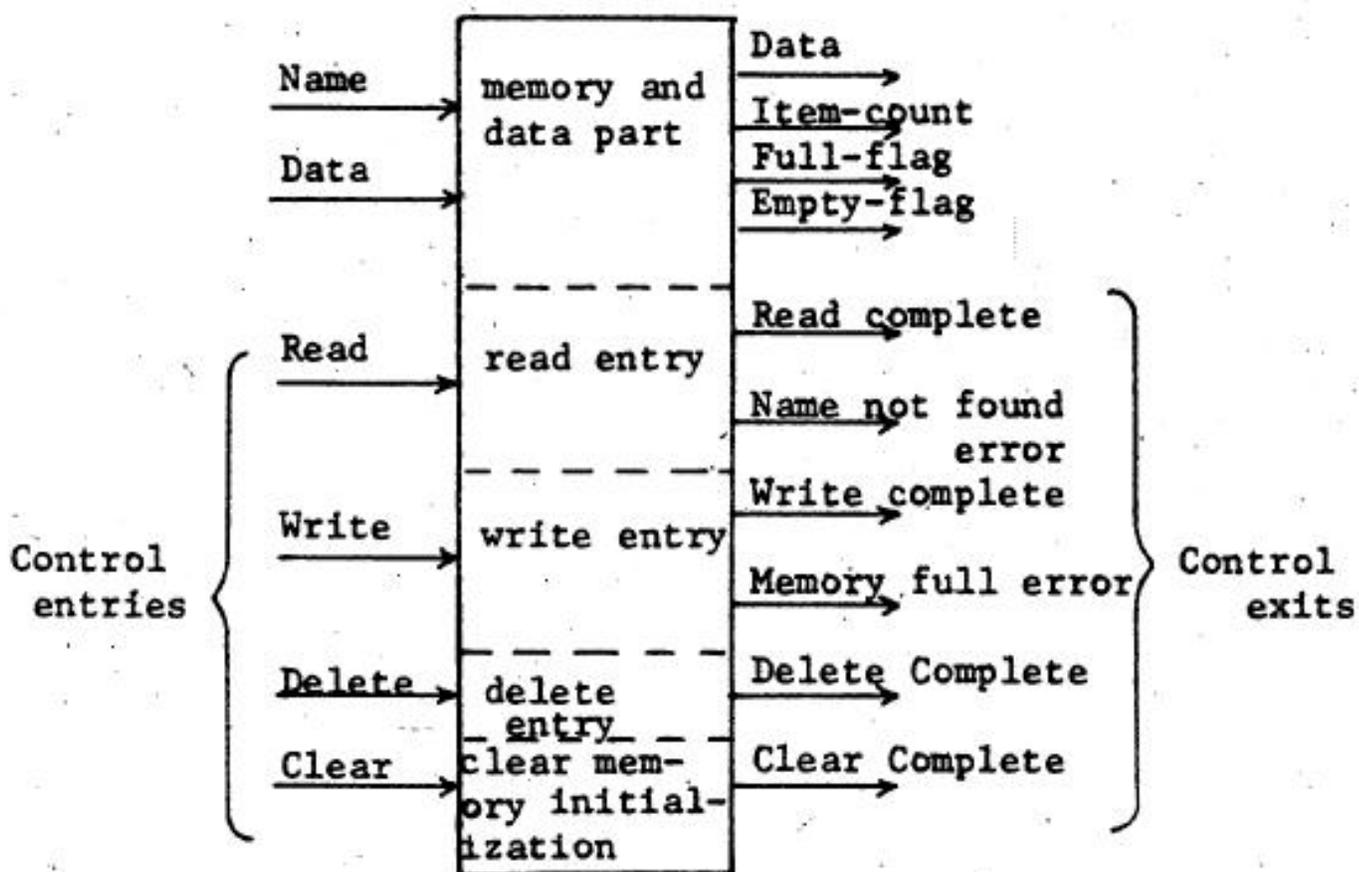


Fig.. MCA-1. PMS diagram of the general structure of an M(content addressable).

3. Linked-list - If ordered entry names are desired but the moving required in the previous case is not allowable, then a linked list structure might be feasible. Associated with each entry name-data pair is a link which points to the next entry in the list. The link most likely to be used is the physical memory address. Adding a new entry involves stepping through the linked list to find the

proper location, entering the new entry at an available memory location, setting the link of the new entry to have the value of the link of its predecessor, then changing the link of the predecessor so that it points to the new entry. Deleting an entry simply involves setting the link of the predecessor to the value of the link of the entry that is to be deleted. Figure MCA-3 shows an example of a linked list, organized memory.

## M(CONTENT ADDRESSABLE)

**KEYWORDS:** Content addressable memory\CAM, entry name search, hash code, linked list.

With array memories, each item is accessed with an explicit address which is an index that must lie in a specific range of values, i.e., zero to  $n-1$  where  $n$  is the number of words in the memory. With queues and stacks, items were accessed at the front and rear or at the top of the memory. Another type of memory is the content addressable memory which is a type of associative memory. The content addressable memory, as its name implies, is addressed by content rather than by an implicit or explicit address. Each cell of a content addressable memory has two parts: the entry name by which the cell is addressed, and the data associated with the entry name. With such a memory, there is no restriction on the address range and the entry names need not appear in any specific order.

A telephone directory is an example of a read-only content addressable memory\CAM, (ignoring the fact that some people often write in telephone books). The entry names are persons' names and the associated data are their telephone numbers and addresses. In one, sense, the telephone directory is organized as a conventional, linearly addressed memory; there are explicitly numbered pages with columns and lines and alphabetic keys at the tops of the pages. Various search strategies that take advantage of the alphabetic order of the entry names may be 'used to locate a particular entry name. If, however, someone were searching for a name associated with a particular telephone number, on the average, half of the directory would have to be searched to find the name associated with the given number.

There are numerous applications for content addressable memories both in hardware and software. For example, the symbol table of a compiler or assembler is such a memory. Some of the entry names are the operation codes (e.g., ADD); associated with each entry name (op code) is data used by the compiler or assembler. Consider a hardware character converter which translates character sets. The source characters would be the entry names into a CAM and the target characters would be the associated data.

Another type of organization for a CAM is by entry name and attribute; each entry name has an associated set of attributes in addition to the data (the data" may be just the attributes). This allows multiple instances of the same entry name with different attributes for the different cases; whole classes of data could be accessed by providing an entry name and desired attributes. The data might also be processed as a function of its entry name and attributes.

## PROBLEM STATEMENT

Design an M(content addressable) using an M(array).

## DESIGN CONSIDERATIONS

The principle design issues are memory organization, search strategy and available operations.

The following is a set of useful operations:

- 1. Read (search) - locate a particular entry name and read its associated data.
- 2. Write - enter a new entry name and data into the memory. If the entry name already exists, overwrite the old data.
- 3. Clear (initialize) - delete all entries in the memory.
- 4. Delete - delete an instance of a particular entry name.

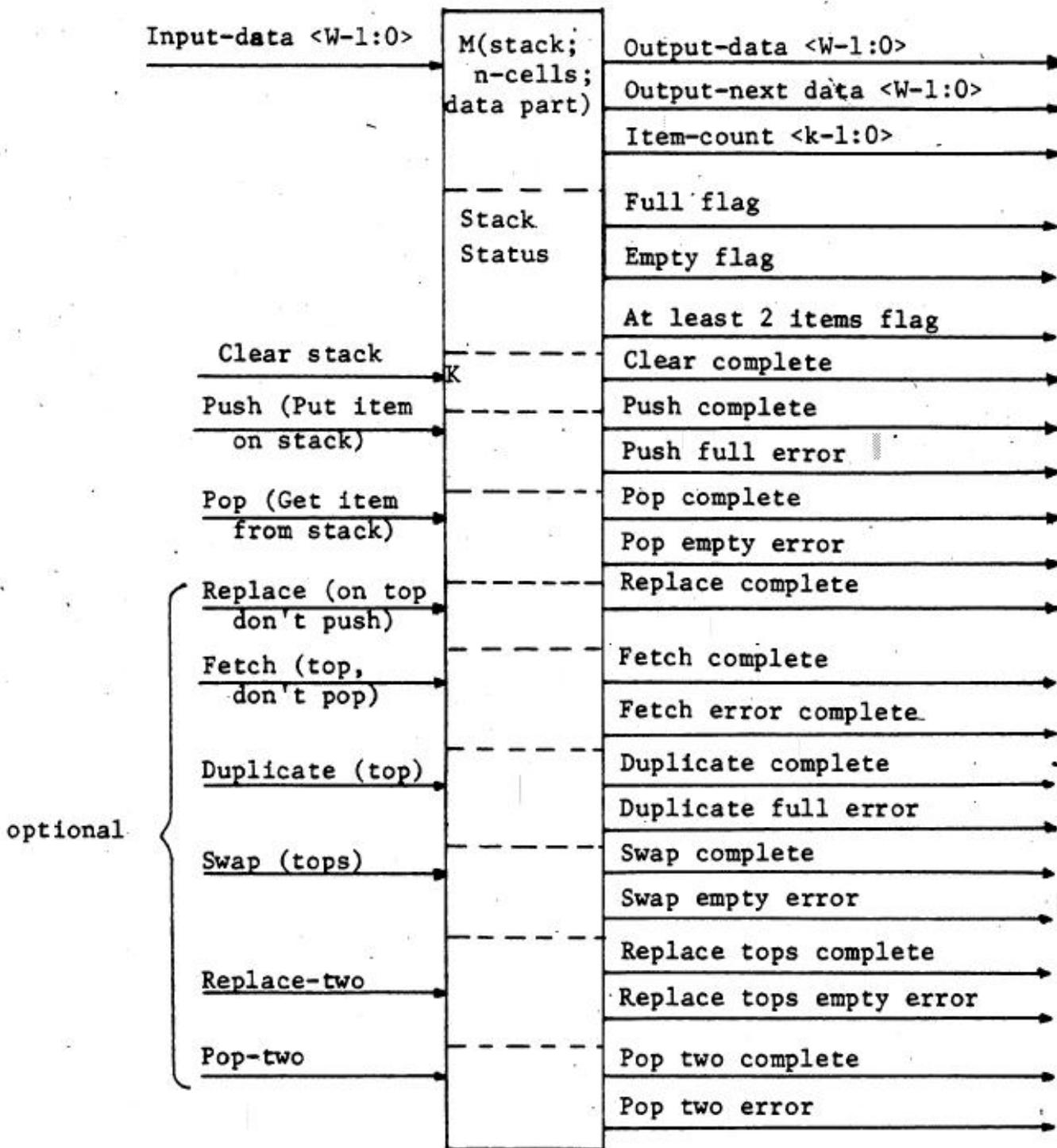


Fig. MS-4. RTM diagram of the general structure of an M(stack).

3. A deque (double ended queue) is another type of memory which has the operations associated with a stack, but provides access to both ends of the memory; design an M(deque) using an M(array).

4. Design an M(double stack) which shares a common memory between two stacks. Extend the design to

n stacks sharing a single memory. (Knuth, 1968)

284

[previous](#) | [contents](#) | [next](#)

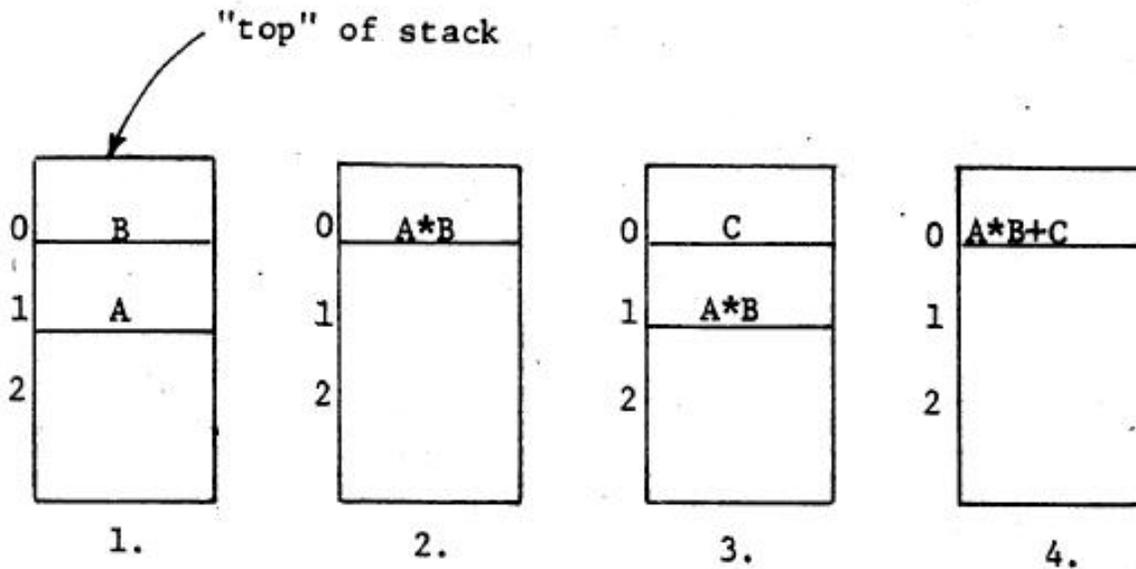


Fig. MS-2. Example of arithmetic expression evaluation using an M(stack).

	<u>Stack Before</u>	<u>Item Count</u>	<u>Stack After (assuming correct ops)</u>	<u>Item Count</u>
Clear	a,b,c,...	N	∅ (empty)	0
Push	a,b,c,...	N	id,a,b,c,...	N+1
Pop	a,b,c,...	N	b,c,...	N-1
Replace	a,b,c,...	N	id,b,c,...	N
Fetch	a,b,c,...	N	a,b,c,...	N
Duplicate	a,b,c,...	N	a,a,b,c,...	N+1
Swap	a,b,c,...	N	b,a,c,...	N
Replace-two	a,b,c,...	N	id,c,...	N-1
Pop-two	a,b,c,...	N	c,...	N-2

Fig. MS-3. Table of stack values for various stack operations.

**Fig. MS-3. Table of stack values for various stack operations.**

283

[previous](#) | [contents](#) | [next](#)

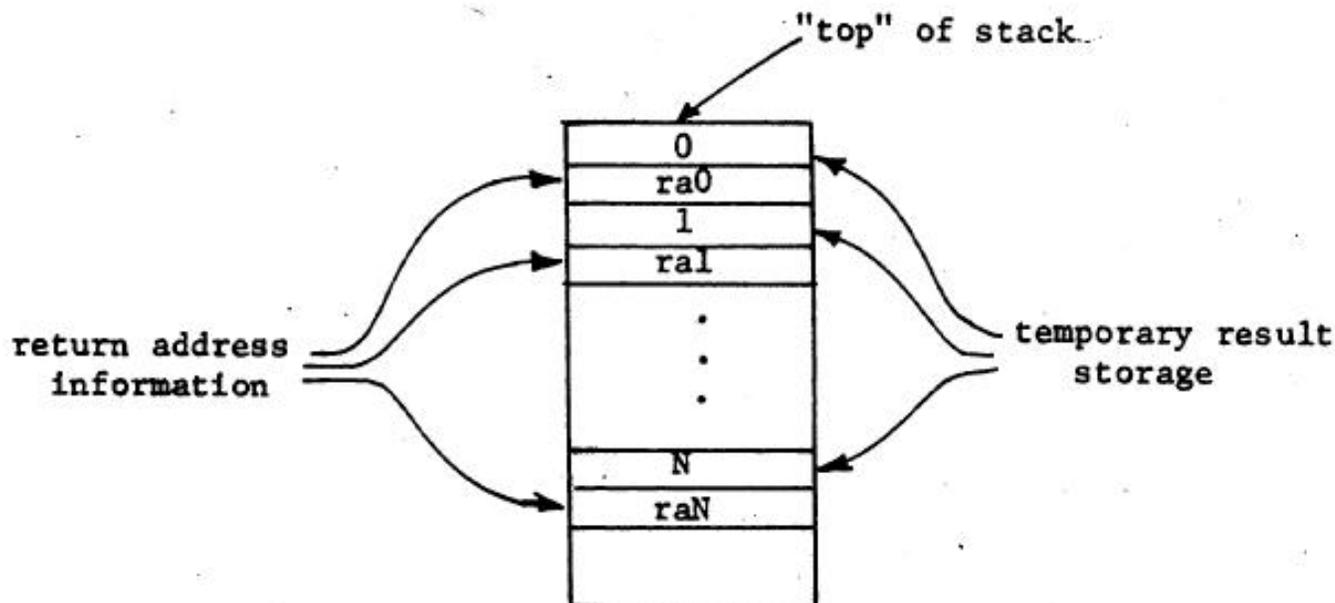


Fig. MS-1. Data in M(stack) for recursive sum-of-integers-to-N subroutine.

e. Duplicate (equivalent to Fetch, then Push of that element) which duplicates the top element of the stack so that the top two elements are the same.

f. Arithmetic, logical and shift operations such as complement, increment, NOT, right shift, left shift, performed on the top of the stack, with the result replacing the top.

2. Binary - These operations affect the top two elements of the stack.

a. Swap (equivalent to Pop-Pop, then two Pushes of the items in reverse order) which interchanges the top two elements of the stack.

b. Pop-two (equivalent to Pop-Pop) which removes the top two elements of the stack

c. Replace-two (equivalent to Pop-Pop-Push) which replaces the top two elements of the stack

d. Arithmetic and logical, such as addition, subtraction, multiplication, division, AND, OR, XOR. These operations replace the top two elements of the stack with a single result.

3. N-ary - These operations can be composites of the unary and binary operations though certain n-ary functions which replace the top n elements of the stack with a single result

are possible.

- a.  $\text{Max}(a,b,c,\dots,n)$  and  $\text{Min}(a,b,c,\dots,n)$
- b.  $\text{Sum}(a,b,c,\dots,n)$  and  $\text{Product}(a,b,c,\dots,n)$
- c. Clear

Examples of several of the above operations are shown in Figure MS-3 and an RTM diagram of an M(stack) is shown in Figure MS-4. The stack, unlike the queue, does not allow simultaneous operations since all accesses to the stack are made at the top of the stack

## PROBLEMS

1. Design an M(stack) using an M(array), with many of the operations described above. Consider the problem of handling invalid operation requests.
2. Design an M(stack) using a number of independent subsystems for each level in the stack.

keep track of which memory belongs to which subroutine is facilitated by the stack.

Another important application for stacks is in the implementation of recursive subroutines, i.e., those which call themselves. For example, in the sum-of-integers-to-N problem, the computation

$$\sum_{i=0}^N i$$

could alternately be written as

$$N + \sum_{i=0}^{N-1} i$$

The process could be repeated, e.g., with the next term

$$N + N - 1 + \sum_{i=0}^{N-2} i \quad \text{until the form of the expression was}$$

$$N + N - 1 + \dots + 2 + 1 + \sum_{i=0}^0 i$$

at which point the expression could be evaluated. An Algol procedure to perform this recursive condition is:

```
integer procedure SUM-OF-INTEGERS(N);
  integer N;
  begin
    if N ≤ 0 then SUM-OF-INTEGERS := 0 else
      SUM-OF-INTEGERS := N+ SUM-OF-INTEGERS (N-1);
  end
```

The above recursive procedure requires both temporary information identifying at which of the N calls it is currently executing, and the intermediate value of the computation it is performing. For each procedure call, a temporary memory call for N is created and the return address to the point of call is saved. Figure MS-1 shows what the stack could look like for the innermost call of this procedure.

A simple, stack-based arithmetic unit will illustrate the use of stack for evaluating arithmetic expressions. An arithmetic expression of the form  $(A*B)+C$  is said to be in operator infix notation, that is, the operators are between its operands. The expression could alternatively be written in functional notation as  $\text{Plus}(\text{Times}(A,B),C)$  or as  $+*ABC$ , which is operator prefix notation. A third possibility is operator

postfix notation which, for the example, would be  $AB^*C^+$ . In this last form there is strict left to right evaluation of the expression and this form is ideally suited for implementation on a stack. Figure MS-2 gives an example illustrating the use of the stack for storing operands while evaluating arithmetic expressions.

Three classes of operations are performed on stacks:

1. Unary - These operations affect only the top element on the stack.
  - a. Push (put |store |write) which adds an item to the top of the stack.
  - b. Pop(get |load |read) which removes the top item from the stack.
  - c. Replace (equivalent to Pop-Push) which replaces the top item on the stack.
  - d. Fetch (equivalent to Pop, then Push of the same element) which reads the top element of the stack without removing it.

N-1, which indicates that no items are in the queue; Empty-flag to true; full-flag to false; Item-count to 0; and the Clear-request flag to false.

The Put subprocess, shown in Figure MQ-4c, performs several tasks: increments the Item-count by one; sets the Empty-flag to false; if item-count equals N, sets the Full-flag to true; increments Put-ptr by one (mod N) and stores the data item from Put-buff at this address (rear of the queue); and sets the Put-request flag to false.

The Get subprocess, shown in Figure MQ-4d, performs these tasks: decrements the Item-count by one; sets the full-flag to false and the Empty-flag to true if the Item-count is zero; increments Get-ptr by one (mod N) and takes the data item at this address (front of the queue) and places it in Get-buff; sets the Get-request flag to false.

Figure MQ-5 shows various examples of usage of the queue.

## ADDITIONAL PROBLEMS

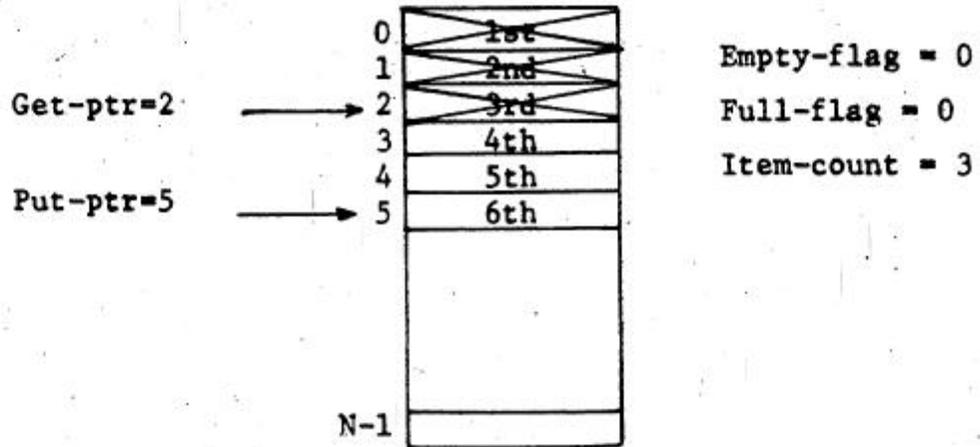
1. Carry out a design which minimizes cost by storing control information about the queue (i.e., Put-ptr, Get-ptr, Item-count) in the memory array.
2. Design an M(queue) which has a number of data parts (stages) which transfer data from stage to stage in a pipeline fashion. Carry out the design for 2,3,...,n data parts.
3. Do a cost/performance analysis to determine the feasibility of always keeping Get-buff filled.
4. Design a time-shared M(queue) which provides for data inputs from multiple sources and data outputs to multiple sinks. Assume that there is a fixed amount of memory to be shared which is shared equally among all queues. Alternatively consider a design with a fixed amount of memory dynamically shared among the queues.
5. Determine the rate at which Put, Get and Clear operations can be processed. In the system described above, an operation request is lost if it is given while a previous request of the same type is being handled. Design an M(queue) that can stack up to K requests of the same type without losing a request.
6. Use K(arbiter) to design M(queue).

## M(STACK) LAST-IN, FIRST-OUT MEMORY

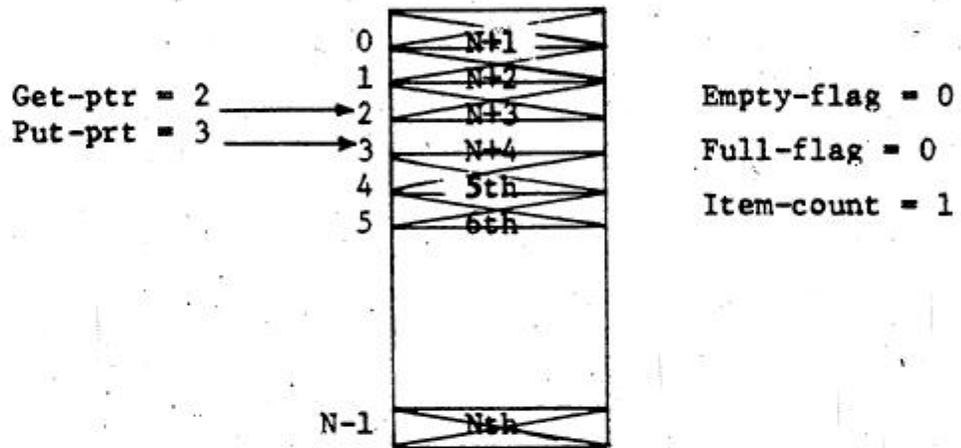
**KEYWORDS:** Stack, LIFO, push, pop, memory

The usual example given of a stack is that of a pile of plates stacked on top of a spring-loaded rack, to which new plates are always put on or taken from the top. In the stack memory, words would correspond to the plates. The stack is used less than the queue in hardware structures, but it is quite useful in programming and in studying theoretical models of computation. Stacks have been used as the main memory in computers, and in desk calculators for holding intermediate results. The two applications of the stack that are most frequently encountered in programming are saving procedure (or subroutine) return addresses, saving temporary information which subroutines use, and evaluating nested arithmetic expressions.

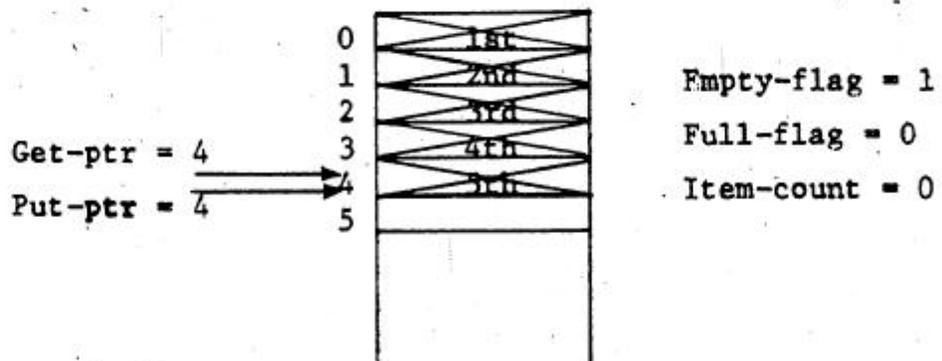
The advantage of the stack as a memory for saving return addresses and temporary information is that space is not required within each subroutine to hold that information. All subroutines use a common memory and the total requirement for the memory is equal to the space required by all subroutines active at a given moment - a dynamic requirement. The bookkeeping required to



a. M(queue) after 6 Puts and 3 Gets.



b. M(queue) after N + 4 Puts and N + 3 Gets.



c. M(queue) after  
5 Puts and 5 Gets

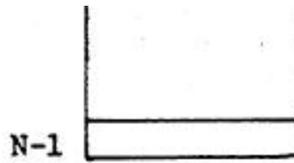


Fig. MQ-5. Examples of M(queue) usage.

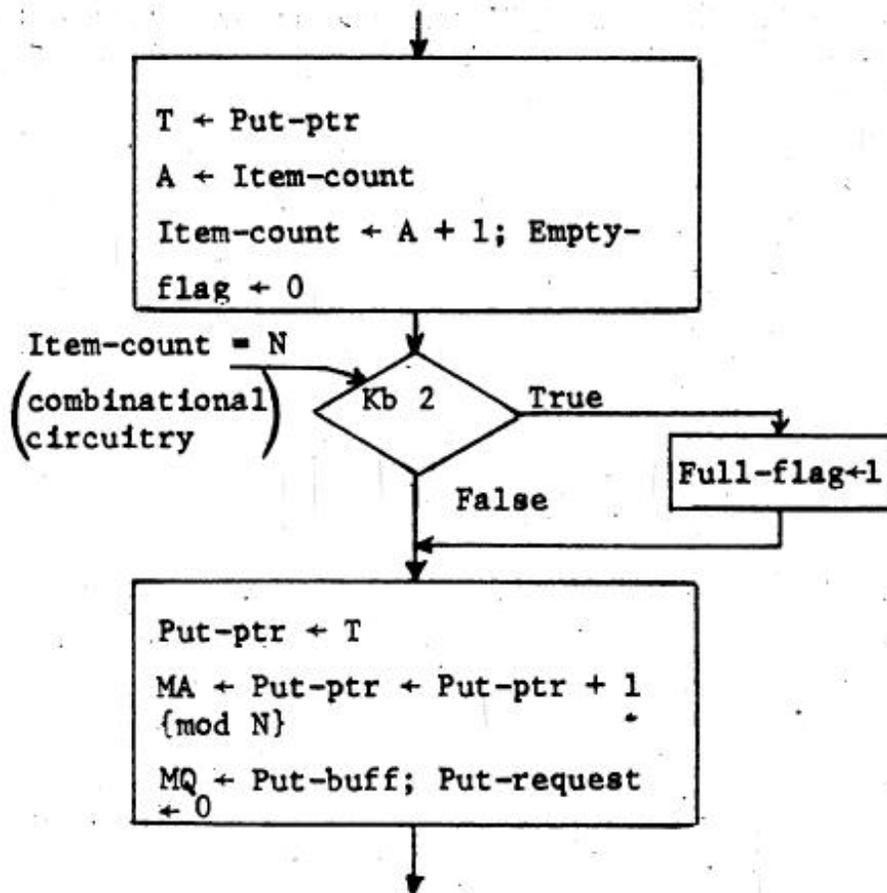
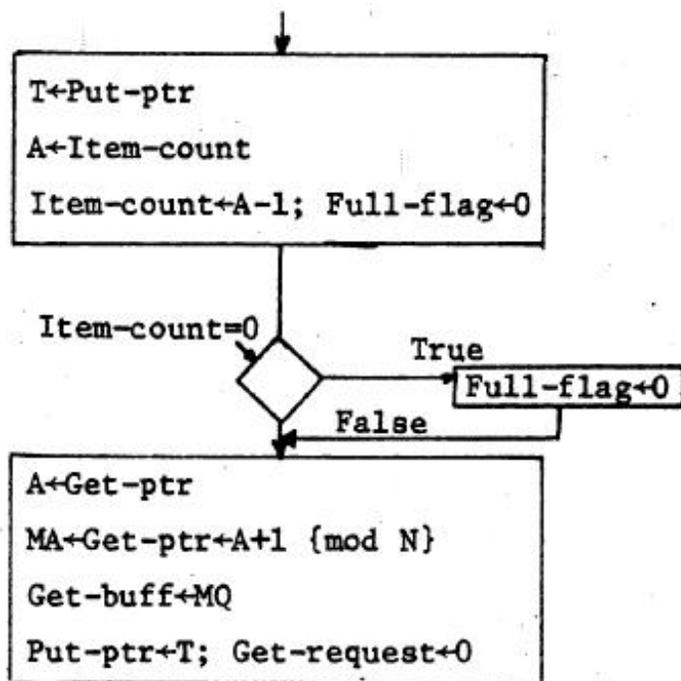


Fig. MQ-4c. RTM diagram of the Put subprocess for M(queue).



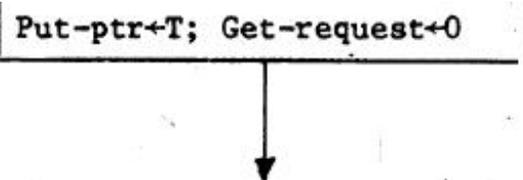


Fig. MQ-4d. RTM diagram of the Get subprocess for M(queue).

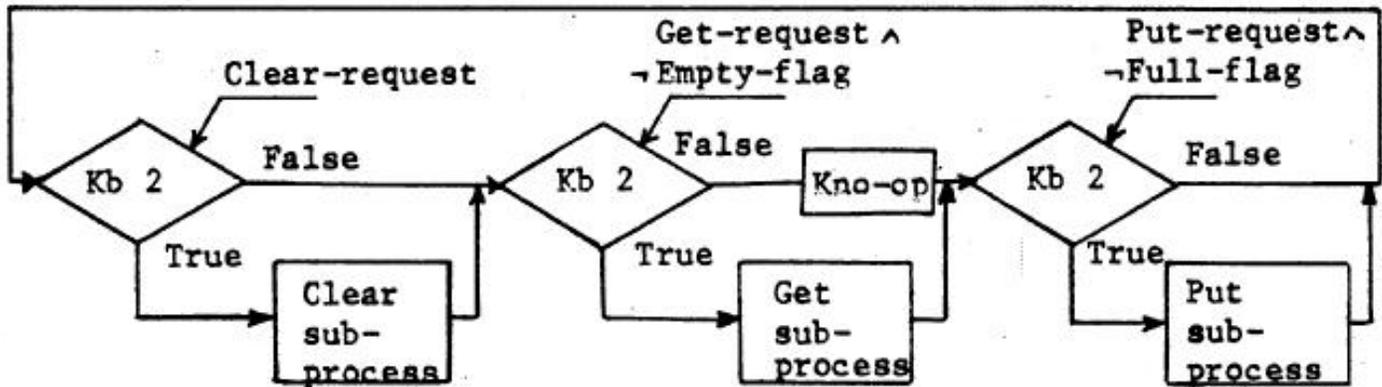


Fig. MQ-4b. RTM diagram of the Clear subprocess for M(queue).

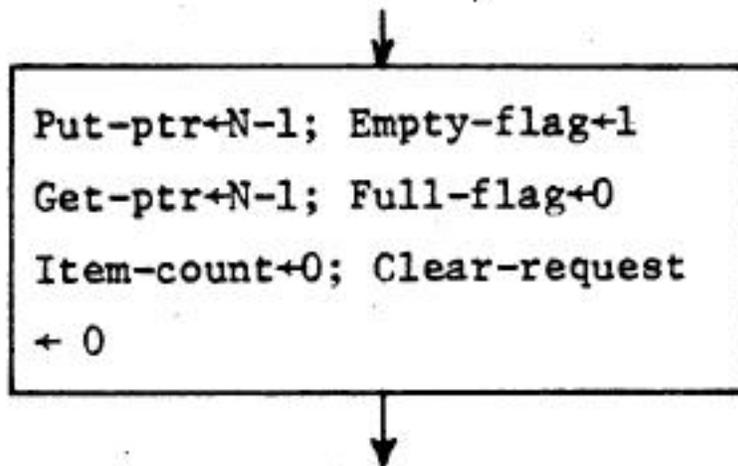


Fig. MQ-4a. RTM diagram of the polling control for M(queue).

5. two single word registers of a DMgpa, Get-ptr and Put-ptr, which act as pointers to the front of the queue (for Get operations) and to the rear of the queue (for Put operations), respectively;
6. two constant registers, N and N-1, where N is the maximum number of items that the queue can hold.

The control part interface, shown in Figure MQ-3, consists of three subroutines which are called by the system using the queue. In each subroutine, the appropriate request flag is set to one when a call is made on the subroutine, then control loops through a no-op module until the request flag is set to zero. The completion signal is then given. An independent central control part controls the actual manipulations on the queue. Communication between the interface subroutines and the central control is through the request flags. The central control part polls the request flag (see Figure MQ-4a) and on finding a request,

initiates the appropriate subprocess. At the completion of the subprocess, the request flag is set to zero which causes the interface subroutine to issue a completion signal. Then polling continues. Notice that simultaneous requests can be entered, but that only one operation subprocess is executed at a time and that requests for invalid operations (i.e., Put with a full queue, Get with an empty queue) are ignored in the polling loop until the operation can be completed successfully.

The Clear subprocess, shown in Figure MQ-4b, sets: Get-ptr and Put-ptr to

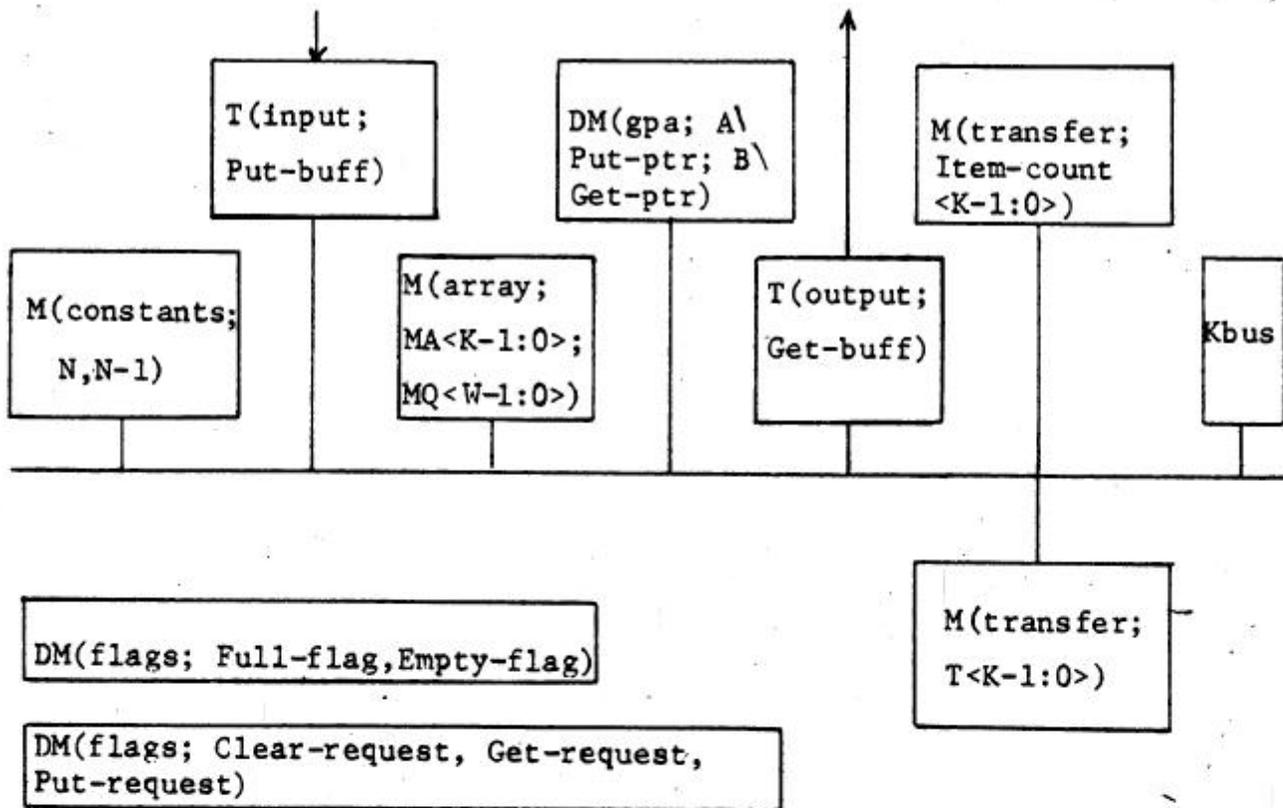


Fig. MQ-2. RTM diagram of the data part of an M(queue).

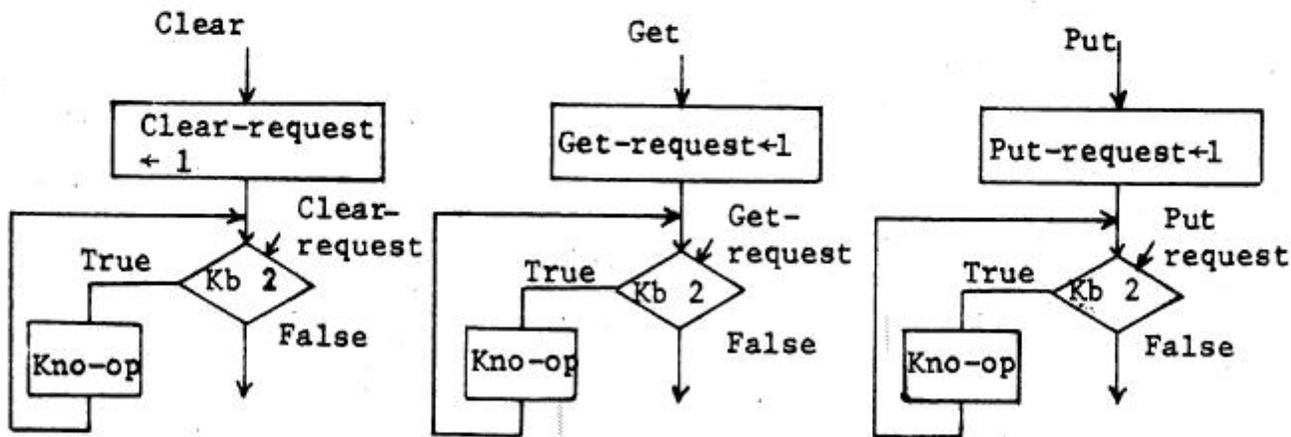
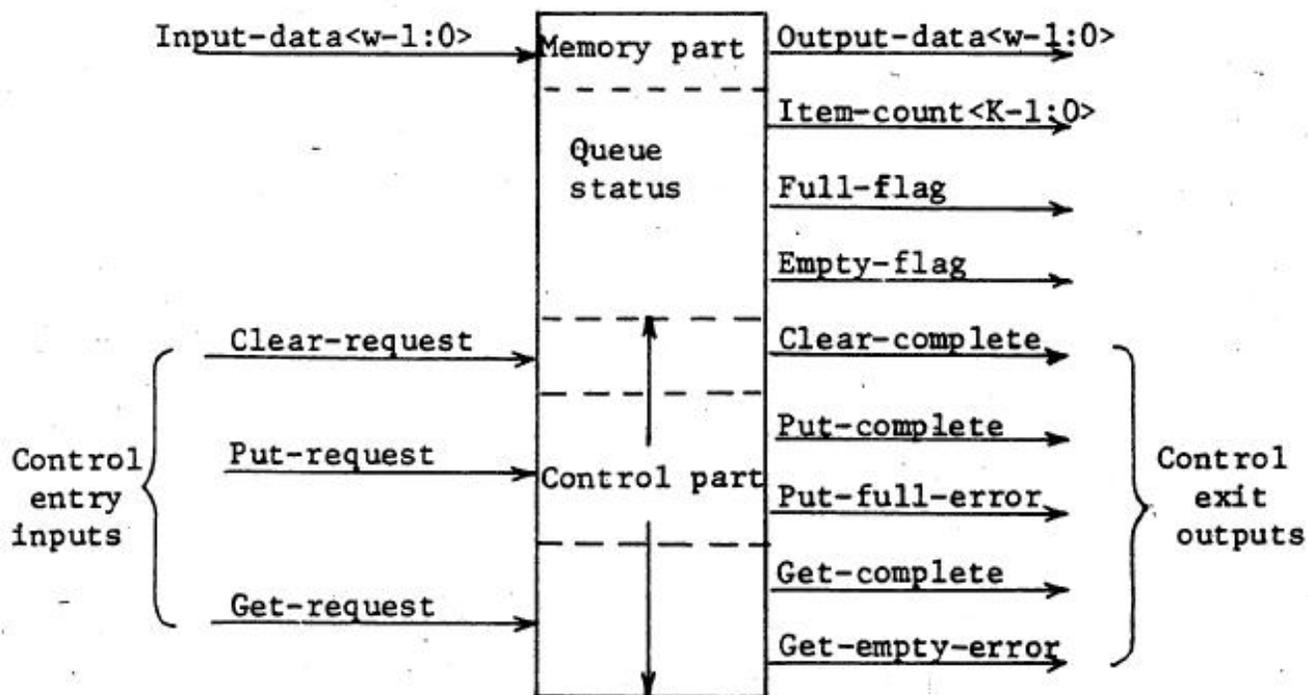


Fig. MQ-3. RTM diagram of the control part of an M(queue) for initiating Clear, Get and Put subprocesses.

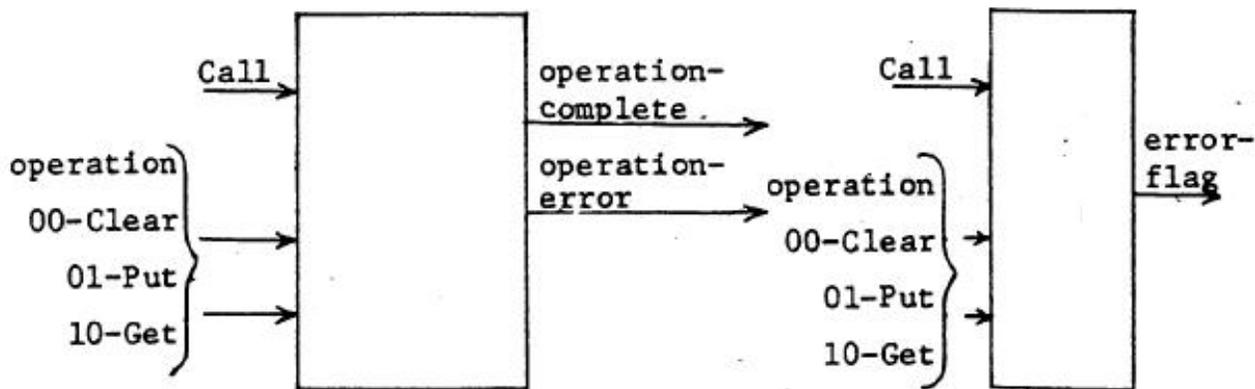
[previous](#) | [contents](#) | [next](#)

<u>Operation</u>	<u>Mq before operation</u>	<u>Mq after operation</u>
Clear	a,b,c,...,x,y	$\phi$ (empty).
Get	a,b,c,...,x,y	a,b,c,...,x
Put	a,b,c,...,x,y	z,a,b,c,...,x,y

a. Behavior table of Mq operations



b. PMS diagram of an M(queue) which allows simultaneous Put and Get requests.



c. PMS diagram of control part of

d. PMS diagram of the control

- c. PMS diagram of control part of on M(queue) with one operation call at a time.
- d. PMS diagram of the control part of an M(queue) without operation completion.

Fig. MQ-1. RTM diagrams and behavior table for M(queue; n words; w bits/word; operations: Clear, Get, Put).

There are no explicit names or addresses given to data in the memory. Putting is always done at the rear of the queue and Getting is always done at the front of the queue. Figure MQ-1a shows the behavior of a queue.

Figure MQ-1b gives a PMS diagram of an  $M(\text{queue})$ , with  $n$  words and  $w$  bits per word. The queue is composed of two parts: the memory for storing the data items in the queue, and the control part which executes the Get and Put processes. The various links and signals are:  $w$ -bit input and output links at the memory part; a  $k$ -bit item count which is a count of the current number of words in the queue; two Boolean flags which indicate an empty or full queue - these are used by external processes to determine whether an item can be taken from or placed into the queue, before actually making a request for an access operation; Put- and Get-request input control signals, and Put- and Get- complete output control signals; optional Put-full-error and Get-empty-error output signals which indicate an invalid operation request, i.e., Putting into a full queue or Getting from an empty queue; and a Clear-request (and Clear-complete) signal which evokes an operation that initializes the queue, i.e., in effect removes all items from the queue.

Figure MQ-1c shows an alternative method for making operation requests to the control part; a single call request is given with a parameter to specify which operation, Get, Put, or Clear, is being requested. In the former control part, simultaneous Put and Get requests could be handled while in the latter case, only single commands can be handled. Figure MQ-1d shows another control part which accepts single operation requests but does not issue an operation completion signal. It only sets an error-flag when invalid operation requests are issued. The assumption allowing this design is that it is the responsibility of the controlling process to wait a certain amount of time after issuing an operation request before issuing a second request. This time is the processing time of the operation requested.

An alternative specification for the design that allows simultaneous Get and Put requests with explicit error returns might be a system which does not issue error signals, but holds queue operation requests until they can be satisfied. For example, if a Put request were entered when the queue was full, that request would be held until a Get request was issued that would free a place in the queue, then the Put would be executed and the completion signal given. A similar process would be performed if a Get request was issued when the queue was empty. Clearly, error signals are not necessary in this design.

## PROBLEM STATEMENT

Design an  $M(\text{queue})$  using any of the control parts described in the introduction.

## SOLUTION

Figures MQ-2 and MQ-3 present the data and control parts which form the interface between the  $M(\text{queue})$  and the system that uses it. The data part consists of:

1. two one-word buffers, Get-buff and Put-buff, which hold data items to be read from or written into the queue respectively;
2. a one-word register, Item-count, which holds the count of the numbers of words in the queue;
3. an array memory which stores the data in the queue;
4. two Boolean flags, Full-flag and Empty-flag, which are used to indicate a full or empty queue respectively;

that the T(general purpose interface) output, Bin, holds the number of the current bin being filled. T is used to detect whether random or sequential addressing is used to input since it holds the old value of the ID switches.

The control part for the controller process is shown in Figure BIN-7. Note the register, temporary\T, holds the physical number of the conveyor cell being considered. The controller waits for the Advance-Conveyor-Flag signal from the conveyor, then \decrements the CHP and inserts the item number in the cell corresponding to that of the conveyor. It then checks all 63 conveyor positions to find those parts that may be ejected into each bin. New items enter the conveyor at the position. marked by the CHP. A count is kept of the items in M[OJ that could not be binned and the Over-30-alarm-f lag is set when this number reaches 30.

## ADDITIONAL PROBLEMS

1. Carry out a study to determine parameters for specifying a class of conveyor-bin systems.
2. What is the approximate rate at which the conveyor can move and not be limited by the controller?

## MEMORIES

This section introduces three new memories, namely M(queue), M(stack), and M(content addressable), into the set of those that have already been introduced as primitive RTM components: DM(flag) - a single bit Boolean; M(transfer), M(byte), and M(constants) - sixteen bit registers; and M(scratchpad) and M(array). In the primitive memories, a particular piece of data is accessed explicitly. The new memories use the following accessing schemes: M(queue) - the first item placed in the memory is the first item removed (FIFO); M(stack) - the last item placed in the memory is the first item removed (LIFO); M(content addressable) - an item of data is accessed by its value rather than by name, a form of associative memory.

The M(queue) presents an interesting synchronization problem. This memory has two independent processes: one for input (called Put), and one for output (called Get). These theoretically could be used simultaneously, but since one physical memory is used to hold the data, it is necessary to synchronize the usage requests. The extended RTM, K(arbiter), previously described, solves this synchronization problem.

With both the M(queue) and M(stack), items are not addressed explicitly; rather the "next" item is accessed. The M(content addressable), though, implies that some searching technique must be used to retrieve data.

An excellent description of queues, stacks and other types of memories is given in Chapter 2 of Knuth

(1968).

## M(Queue) FIRST-IN FIRST-OUT MEMORY

**KEYWORDS:** Queue, FIFO, put, get, clear, memory, synchronization

Queues appear in many different systems ranging from barber shops and grocery store check-out stations to components within computers and their programs. For information processing systems, an M(queue)\Mq is usually a fixed-size array memory with the following basic operations: Put(write) - for entering items into the queue; Get(read) - for taking items out of the queue.

273

[previous](#) | [contents](#) | [next](#)

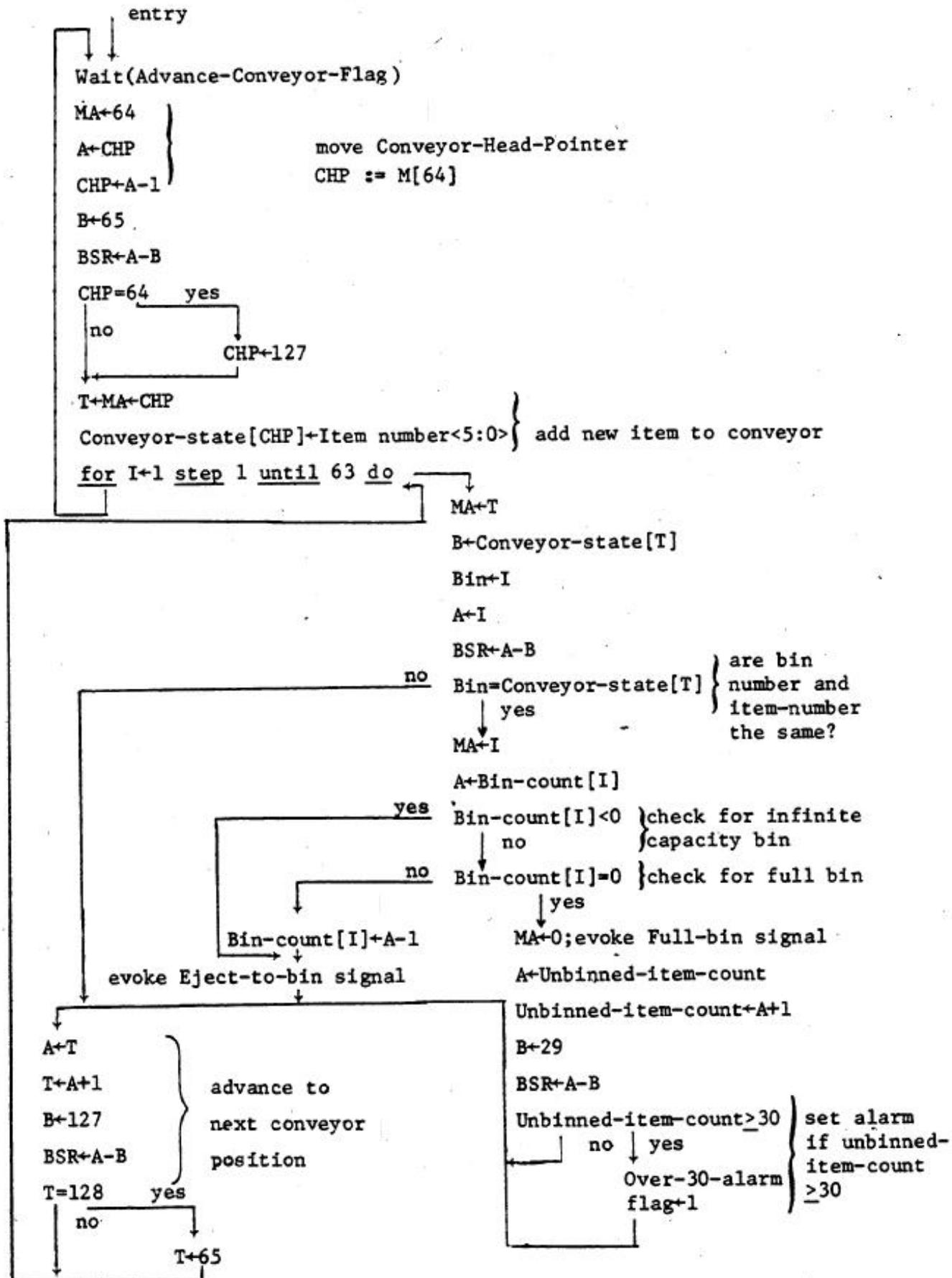
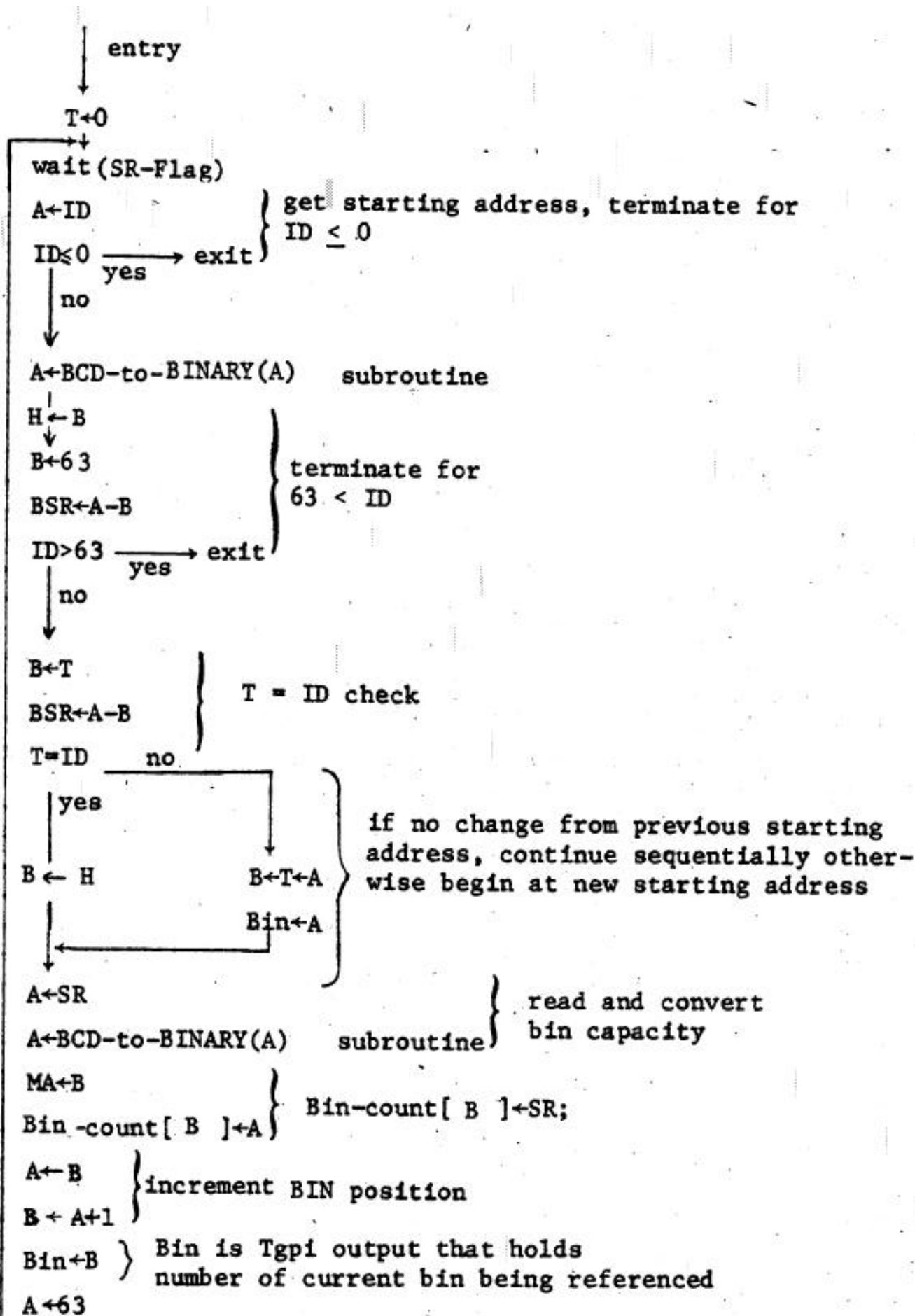


Fig. BIN-7. RTM diagram of the control part for the controller process.

272

[previous](#) | [contents](#) | [next](#)



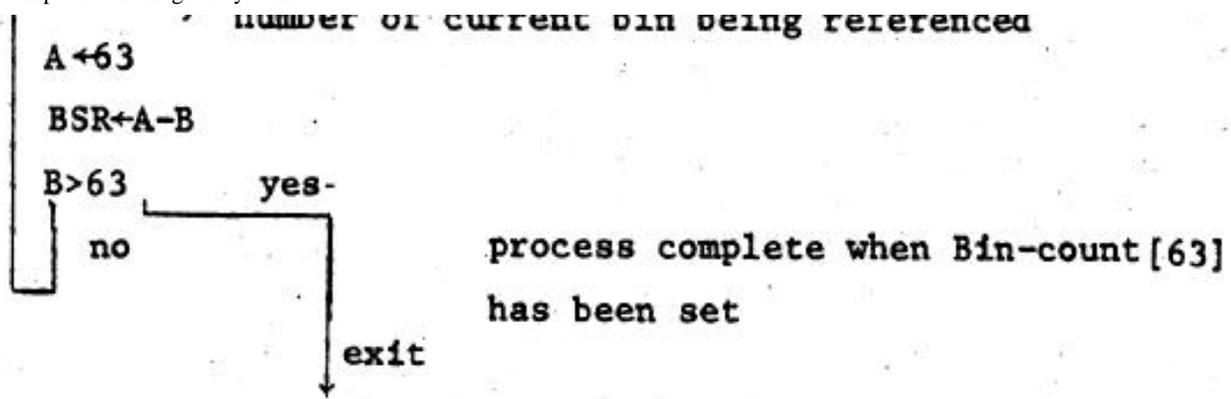
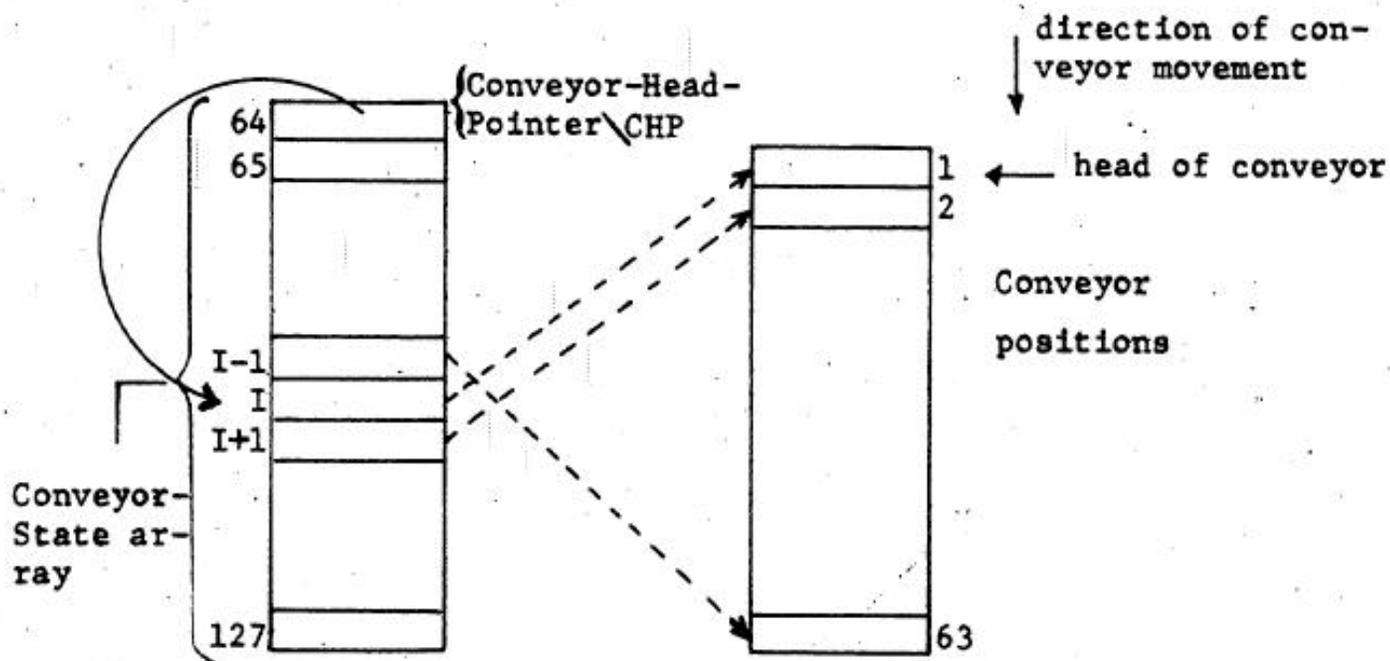


Fig. BIN-6. RTM diagram of the control part for modifying the bin-count array.



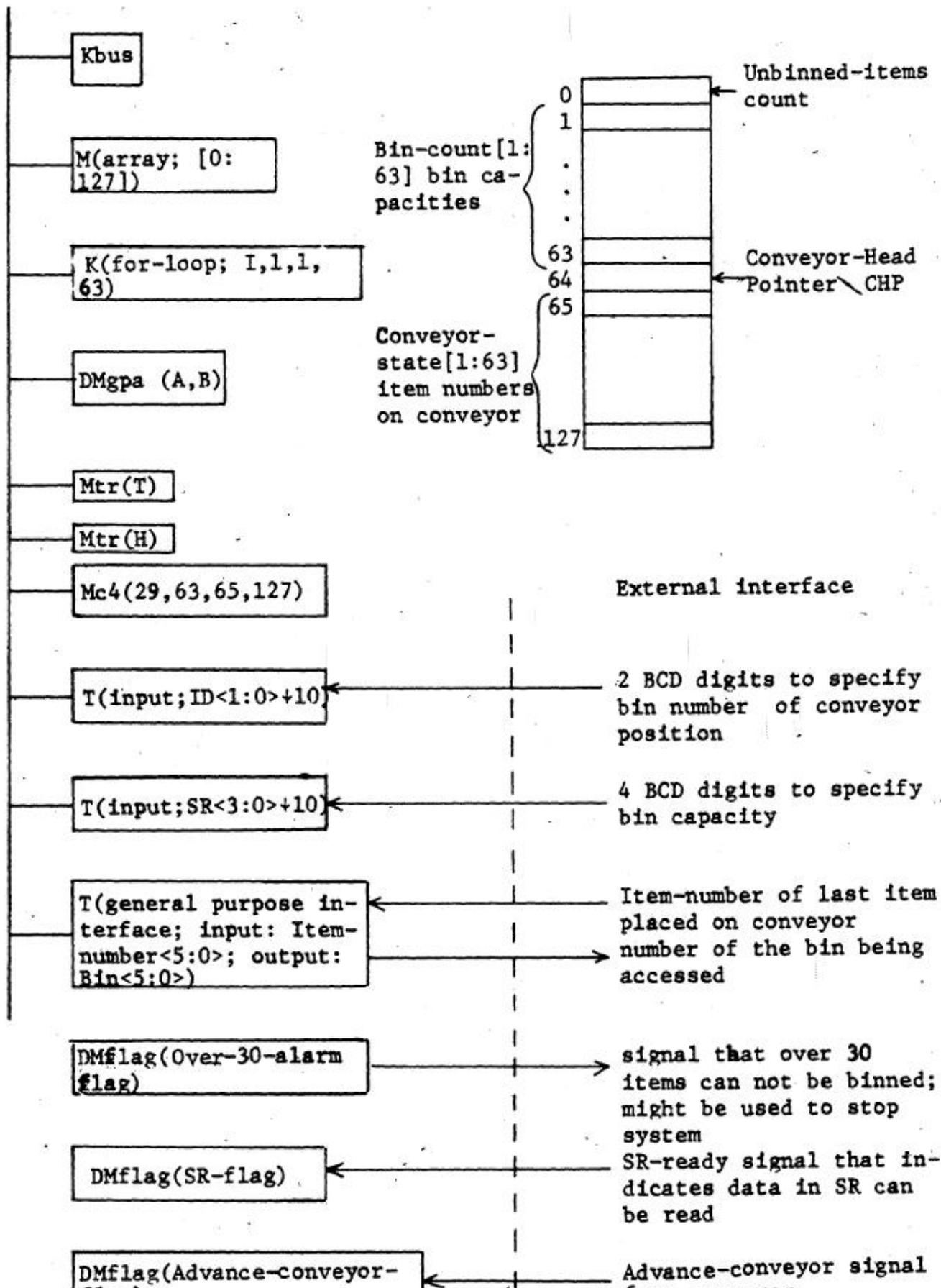
Decreasing Conveyor-Head-Pointer by 1 corresponds to advancing conveyor 1 position.

Fig. BIN-5. Mapping of conveyor-state-array [65:127] to conveyor positions [1:63].

6. a six-bit number, Item-number  $\langle 5:0 \rangle$ , which is input from the conveyor system to the controller indicating the item-number of the part just placed on the conveyor;
7. a six-bit number, Bin  $\langle 5:0 \rangle$ , which is output from the controller to the conveyor indicating a bin number;
8. an evoke-signal, Eject-to-bin, which signals the conveyor to read the number on Bin  $\langle 5:0 \rangle$  and eject the part on the conveyor in front of that bin into it;
9. an evoke signal, Full-bin, which signals that Bin  $\langle 5:0 \rangle$  contains the number of a bin that is full and cannot accept the item on the conveyor in front of it;
10. a four position switch, operation-mode-switch, which indicates the mode of operation for the controller.

The control part of the system can now be specified in terms of the problem description and the data part.

Figure BIN-6 gives the control part for manually modifying the Bin-count array; the control part for modifying the Conveyor- state-array is almost identical. The general operation of these two processes, which is a product of the design process, not the initial specification, is that ID specifies a starting bin (conveyor) position, and each time the SR flag is set, the contents of SR are read into the Bin-count (Conveyor-state) array. The bins (conveyor positions) are filled sequentially starting from ID until a new value for ID is entered, at which point the process is repeated. The two processes stop when the final Tgpi output Bin (conveyor) position has been referenced. Note



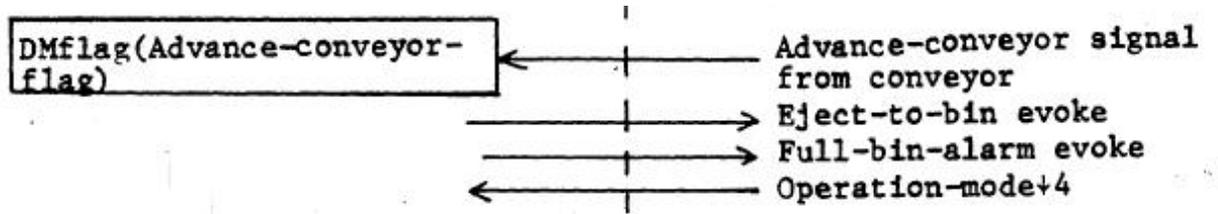


Fig. BIN-4. RTM diagram of the data part for the conveyor-bin-system.

## PROBLEM STATEMENT

Design the controller for the conveyor-bin system, described above.

## DESIGN CONSIDERATIONS

Some type of scheme must be developed to represent the capacities of the bins; therefore, the following assumptions will be made: finite capacity bins are limited to contain no more than 9999 items and will be represented by a positive integer that represents the current number of available spaces in the bin. Zero, therefore, indicates a full bin; open-ended bins with infinite capacity will be represented by a bin count of -1.

As stated in the introduction, BCD numbers will be used at the human operator interface, therefore the BCD conversion routines developed earlier will be used.

## SOLUTION

The data part of a controller for the conveyor-bin-system is shown in Figure BIN-4. A 128 word memory array (not part of the RTM set, but assumed for this problem)- stores two 63 word arrays: words [1:63] are bin-counts which indicate the capacities (as explained above) of the 63 bins; words [65:127] correspond to the 63 conveyor positions (referred to as the Conveyor-state-array); word [0] contains the number of items in the holding hopper which cannot be placed in bins -- when this number exceeds 30, the over-30-alarm-flag is set. Since the conveyor is moving, some relationship between memory words and conveyor positions must be made. One alternative is to assign each memory word to a fixed physical position and each time the conveyor moves, the Item-numbers in the cells are shifted up one position so that items in memory correspond to bin positions.

A second alternative, which was adopted in this design, assigns each memory cell to correspond to a conveyor cell. A pointer, the Conveyor-Head Pointer, CHP, is used to point to the head (i.e., the first position) of the conveyor at a given instant in time. Each time the conveyor moves forward one position, CHP is decreased by one; thus the relative distance of a particular item on the conveyor to the head is increased by one. Thus memory cells are not moved. CHP is stored in word [64] of the memory array, and must be counted modulo 63. New items enter the conveyor at the cell pointed to by CHP (see Figure BIN-5).

By storing all of the above data in a single array, an added benefit is gained. If the memory is non-volatile (i.e., it does not change when power is turned off), then the controller can be shut down and restarted at some later time in the same state as when it was turned off.

The following registers, flags, and signals form the interface between the controller and the external

system (i.e., conveyor and operator):

1. a register, ID, which is a two digit BCD switch that addresses either a bin or a conveyor position when it is necessary to change the state of the system manually;
2. a register, SR, which is a four digit BCD switch that is used to input a bin capacity or an item number;
3. a flag, Advance-conveyor-flag, which is set by the conveyor externally and signals the controller that the conveyor has advanced one position;
4. a flag, SR-flag, which is set by the operator and signals the controller that the operator would like to have data read in from the SR register;
5. a flag, Over-30-alarm-flag, which signals that over 30 unbinned items are in the holding hopper;

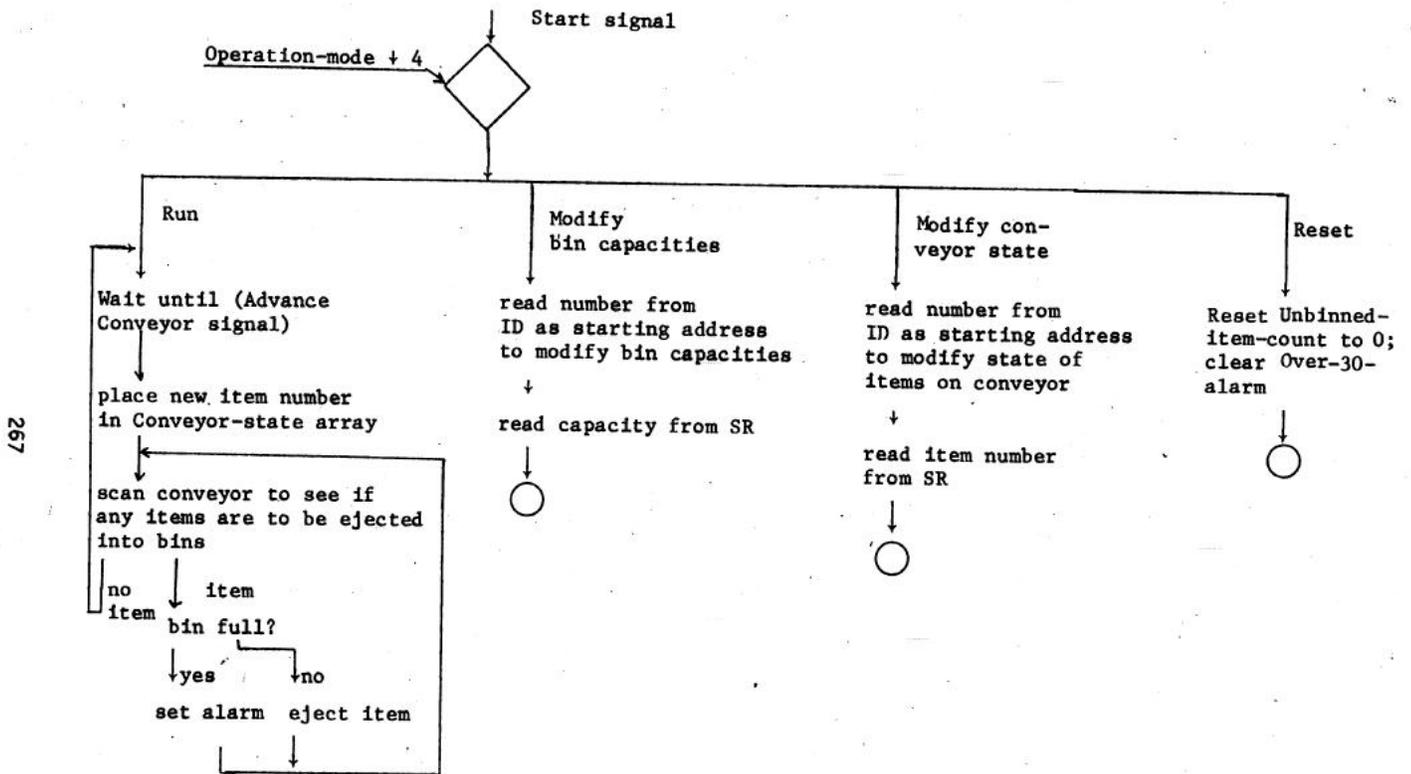


Fig. BIN-3. Flowchart showing basic behavior of the four modes of operation.

The inputs and outputs of the controller for the system are shown in Figure BIN-2. The conveyor gives an Advance-conveyor signal which signifies that all items have been moved up one position, and an Item-number is supplied signifying the part number just placed on the conveyor. The numbers 1 to 63 will be used for item numbers and 0 will indicate an empty position on the conveyor.

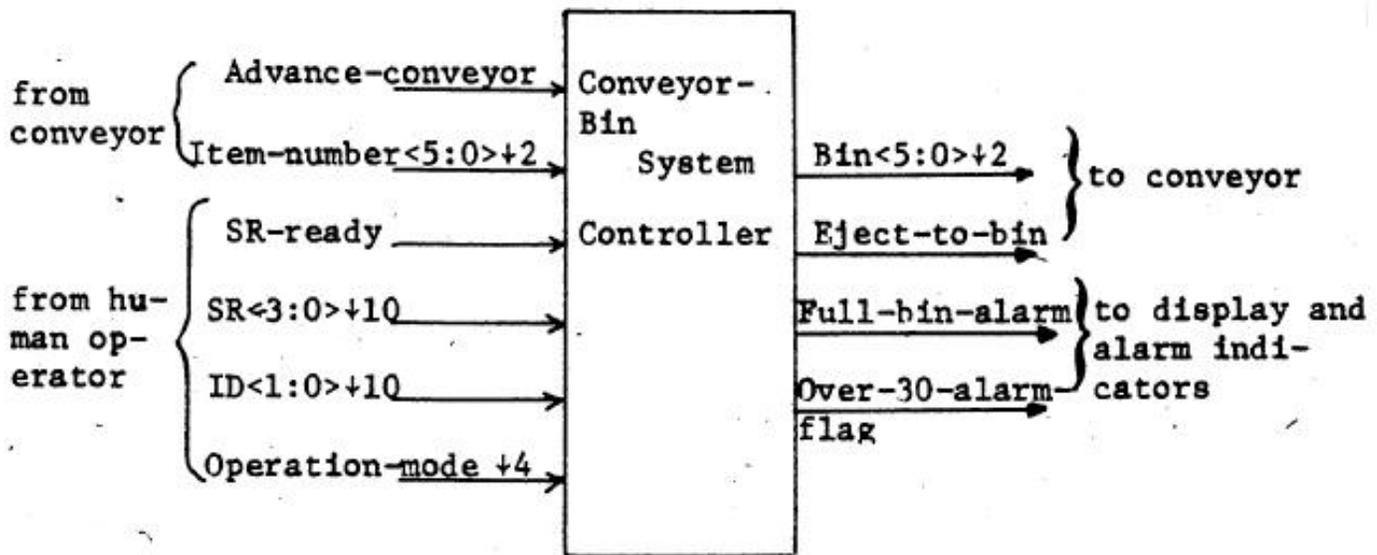


Fig. BIN-2. PMS diagram showing inputs and outputs to the conveyor-bin-system controller.

The remaining inputs come from the human operator interface. The number, ID, identifies a bin or conveyor position number and the Operation-mode switch allows the operator to select one of four possible modes for operating the system. The switch-input SR is used by the operator to input information about the system. This includes items on the conveyor when it is started and the number of available spaces for each bin. The signal, SR-ready, indicates that data is available to be read from SR. All inputs from the human interface are encoded in Binary-Coded-Decimal and must be converted to binary by the controller.

The outputs from the controller include: the bin number being specified by the controller; the Eject-to-bin signal which causes the specified bin to be opened and the item on the conveyor placed in the bin; the Full-bin-alarm signal which initiates an audible alarm and the display in lights of the bin number at any bin each time an item cannot be placed in the bin because it is full; and the Over-30-alarm-flag which might be used to shut down the system when over 30 unplaceable items are in the special holding hopper.

The four modes of operation (shown in Figure BIN-3) for the system are:

1. Automatic-run - in which the controller is running the system.

2. **Modify-bin-count** - in which the operator may change the specification of the capacity of any of the bins.
3. **Modify-conveyor-state** - in which the operator may change the status of the items on the conveyor.
4. **Reset-unbinned-item-count** - in which the **Over-30-alarm-flag** is Reset to zero.

## CONTROLLER FOR A CONVEYOR-BIN SYSTEM

**KEYWORDS:** Controller, operator, interface, representation

This problem originated as a controller for a conveyor carrying lumber to be sorted into bins. Since it is typical of sorting conveyor-bin problems, the parameters of the lumber system will be used. The problem is presented for its practical significance and its interesting human interface and representation problems.

A production process creates items to be placed in 63 bins. When an item is placed on the conveyor, it is given an item number which identifies the bin into which it is to be placed. Faulty items are also placed on the conveyor and transferred to a reject-bin, it is assumed that the conveyor moves in discrete steps, and after each step, a new item is placed on the conveyor and all items move up one position. If items opposite new bin positions have the corresponding item number, they are ejected into the respective bins. A diagram of the system is shown in Figure BIN-1

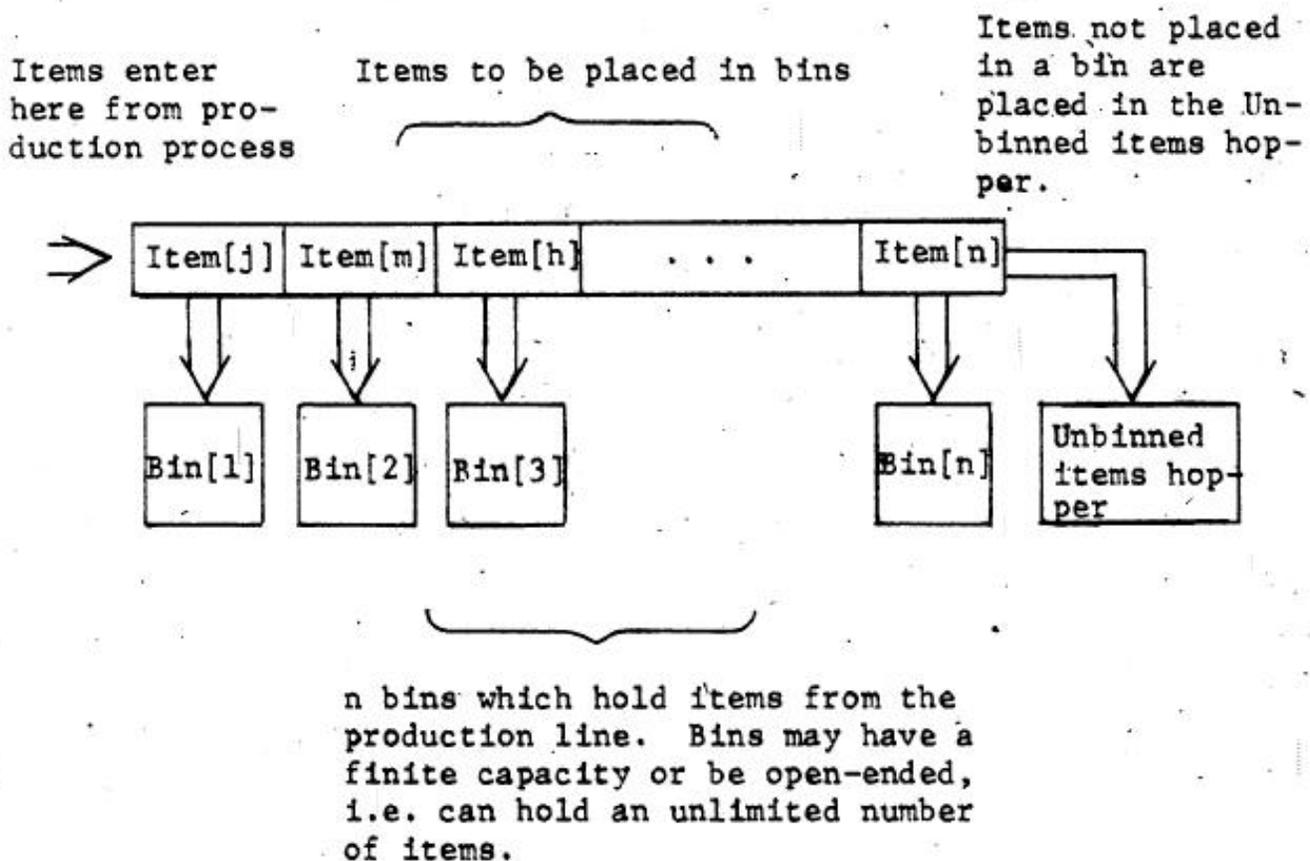


Fig. BIN-1. Flow diagram for a conveyor-bin-system.

The bins are of two types: those with a finite capacity which is set when the process is initialized, and those which are open-ended and have an infinite capacity. This latter type is used for the reject bin and

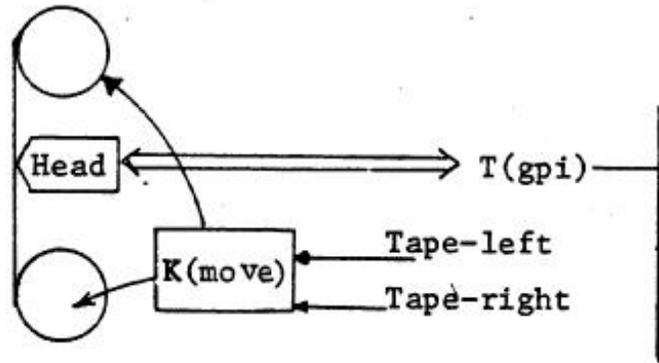
may be used in other positions. When a bin is filled, any items which are assigned to that bin ride to the end of the conveyor and are placed in a special holding hopper called the Unbinned items hopper. After some fixed number of items are placed in this hopper, an alarm is given which can be used to stop the system so that the bins can be emptied and the items in the holding hopper placed on the conveyor again. Each time an attempt is made to place an item in a bin that is filled, a signal is given to indicate the condition.

## EXTENSION

The most useful extension to the parity counter design would allow tapes longer than sixteen bits to be processed. The device that would allow this extension is a Turing Tape Transport. Such a device is shown in Figure TUR-5 and is a close approximation to an actual drive known as an incremental tape drive. With a tape device such as this each character of the alphabet 0, 1, and B may be represented (assume two bits represent a character as  $0 = 00$ ,  $1 = 01$  and  $B = 10$  or  $11$ ). In fact most tapes are 7 or 9 bits wide which would allow alphabets with  $2^7$  and  $2^9$  characters respectively. Design a controller for the Turing Tape Transport to compute parity using the quintuple of Figure TUR-2.

## ADDITIONAL PROBLEMS

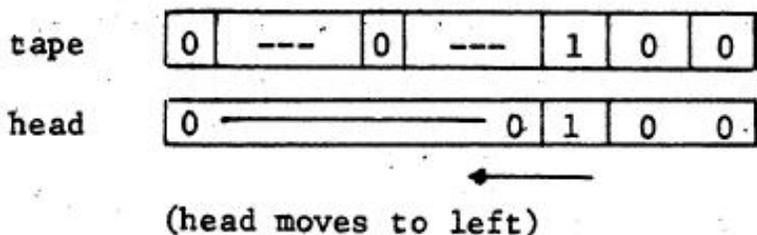
1. Design a Turing machine which computes parity without erasing the tape.
2. Design a Turing machine which adds one to a binary number stored on the tape and leaves the result on the tape. The read head will be placed under the least significant bit of the word initially and the termination symbol, B, will mark the left end of the word.
3. Design a Turing machine which adds two binary numbers together. Assume that special characters delimit the numbers on the tape (i.e. increase the alphabet).
4. The parity counter essentially had its quintuples hardwired in the design. Design a Turing machine simulator that operates from quintuples stored in a memory array, i.e., this machine should operate for any set of quintuples. The major issue here is the representation of the quintuples and the method of table lookup to determine the appropriate action for each current state - symbol read pair. Such a machine resembles K(PCS), described in Chapter 2.



Commands for Turing tape drives:

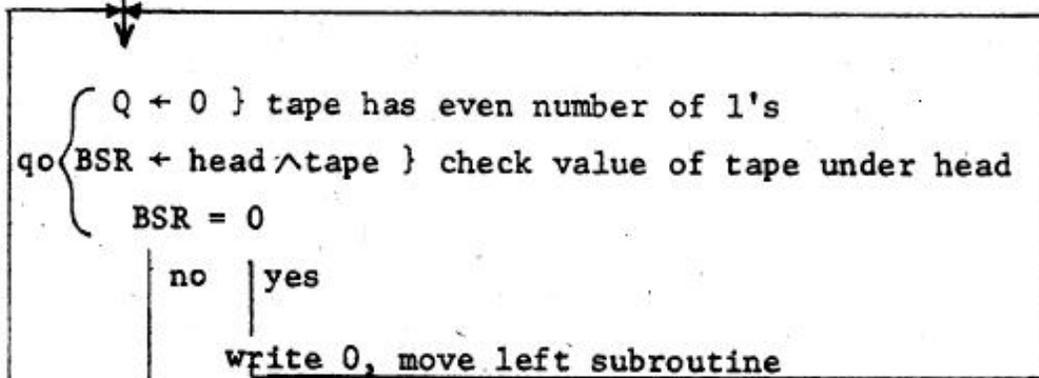
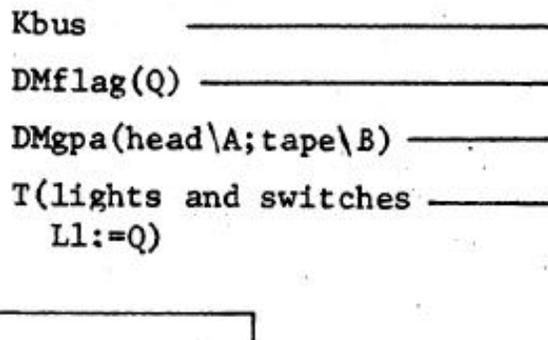
1. Bus  $\leftarrow$  Head (read value of a tape square)
2. Head  $\leftarrow$  Bus (write value of a tape square)
3. Tape-left; Tape-right evokes

Fig. TUR-5. An RTM T(Turing tape drive) which is similar to an incremental tape drive.

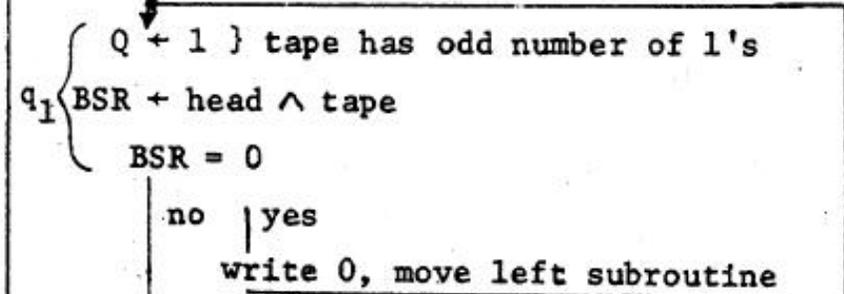


entry  
 ↓  
 head ← 0  
 head ← head + 1  
 tape ← switches

} place a head next to a short tape



write 0, move left subroutine



write 0, move left subroutine

write 0, move left subroutine

entry  
 ↓  
 tape ← head ∨ tape }  
 tape ← head ⊕ tape } write 0  
 head ← head x 2; set OVF } move left  
 OVF

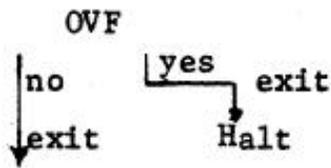
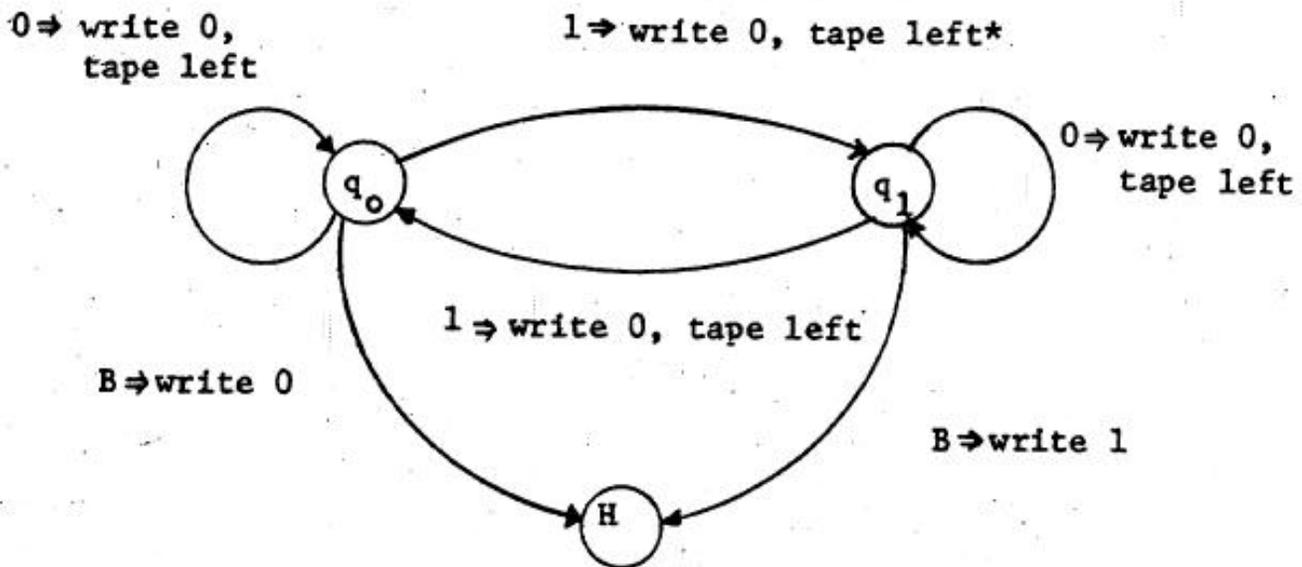


Fig. TUR-4. RTM diagram of a system which simulates a parity counter Turing machine.

	$Q_i$	$S_j$	$Q_{ij}$	$S_{ij}$	$D_{ij}$
1.	0	0	0	0	left
2.	0	1	1	0	left
3.	0	B	Halt	0	-
4.	1	0	1	0	left
5.	1	1	0	0	left
6.	1	B	Halt	1	-

Fig. TUR-2. Table of quintuples specifying a parity counter Turing machine.



\* This notation (1 ⇒ write 0, tape left) means when a 1 is encountered (in this case in state  $q_0$ ), write a 0 on the tape, and go left.

... state q<sub>0</sub>, write a 0 on the tape, and go left.

**Fig. TUR-3. State diagram representation for the parity counter Turing machine.**

current state	symbol read	new state	symbol written	direction to move
$Q_i$	$S_j$	$Q_{ij}=G(Q_i, S_j)$	$S_{ij}=F(Q_i, S_j)$	$D_{ij}=D(Q_i, S_j)$

Fig. TUR-1. Format for specification of Turing machine quintuples.

l's. The machine will also write 0's on all squares of the tape until a termination character, B, is encountered; this symbol will be written over by a 0 or 1 to denote even or odd parity, respectively, of the tape. The symbols 0, 1, B are the only symbols that will appear on the tape. The machine will start at the rightmost end of the tape and move left until the 'termination character is detected. The table presented in Figure TUR-2 shows six quintuples which express the behavior of the parity counter Turing machine. A state diagram of the parity counter is shown in Figure TUR-3.

## PROBLEM STATEMENT

Design a parity counter using RTM's which "simulates" the Turing machine.

## DESIGN CONSIDERATIONS

The tape representation is the most important design issue for this problem. There are three tape symbols, 0, 1, and B, requiring two bits to represent uniquely the values. For the parity counter, however, the termination symbol, B, occurs only once on the tape. Possibly an alternative method could be found to mark the end of the tape, thereby doubling the density of the tape since one bit will suffice to represent 0 and 1. The infiniteness of the tape also presents a problem. The largest data element that can be operated on with RTM's is a sixteen bit word. The parity counter, in this problem, will be restricted to tapes with sixteen squares. The overflow bit will be used to mark the end of the sixteen square tape which will be held in a register of a DMgpa.

## SOLUTION

A parity counter is shown in Figure TUR-4. The tape is held in the B register of a DMgpa and the read head is represented as a single one bit in the A register. The tape is read by ANDing the tape and the read head. If the result is zero, the bit under the read head is zero; if the result is non-zero, the bit under the read head is a one. The symbol 0 is written on the tape by ORing, then XORing the tape and the read head together. A left shift moves the read head; when the bit is shifted into the OVF bit, the machine halts. The action of writing a zero and shifting the read head left is implemented as a subroutine.

An initial tape is entered through the switches of a T(lights and switches). The state of the control for the machine, either 0. or 1, is held in the DMflag(Q) and is displayed at light L1. When the machine halts, the light will show the parity of the tape, 0 = even and 1 = odd.

The device is to take a fixed number of 8-bit analog samples, here 512. The samples are first stored in a fast memory and then punched on paper tape with an 8-bit identifying number preceding the samples. This number, the Sample Identification, is supplied from an external source.

## ADDITIONAL PROBLEMS

1. Design a system in which the analog input is multiplexed for eight inputs.
2. Modify your system so that the identifying number has four 8-bit characters.
3. What is the maximum sampling rate that might be accomplished using the same a-to-d converter, and memory? What would such a design look like?
4. There is no way to tell if the data punched is the same as that sampled and read. Paper tape systems of this type have been known to make errors (e.g., 1 character in  $10^6$  punched is not unheard of). Design a scheme for detecting whether an error has been transmitted in the data which was punched. Design a scheme that assumes 1 character has been erroneously punched with a single bit error. Add enough information so that the error can be corrected.
5. Assuming the analog-to-digital converter samples are variable from 8 to 12 bits, modify the design to accommodate the variable sample accuracy.

## TURING MACHINE SIMULATOR

**KEYWORDS:** State, symbol, tape, transport, Turing machine

In 1936 A. M. Turing in a paper on the theory of computability, defined a class of abstract machines which we now call Turing machines. These machines have both theoretical and pedagogical applications in the study of the nature of computation. Turing machines provide an effective method for expressing a computational procedure; these model initial, terminal and looping conditions as well as the basic computational steps of the procedure. A Turing machine has a finite number of states (resembling a conventional controller) and is coupled by a single read-write head to a tape (resembling a magnetic tape) which is infinitely long and consists of squares which may have symbols from a finite alphabet written on them. The Turing machine executes three basic functions:

- 1. Read - read the symbol from the square of the tape under the head.
2. Write - write a symbol on the square of the tape under the head.

3. Move - move the read head one square to the left or right on the tape.

The operation of the Turing machine proceeds as follows:

- 1. Read a symbol from the tape.
2. Using the current machine state and symbol read as parameters, compute:
  - a. a symbol to write on the tape (possible the same symbol as was read);
  - b. a new machine state (possibly the same as the current state);
  - c. a direction to move along the tape (left or right).
3. Loop to step 1.

The functions for computing new machine states, symbols, and directions are usually expressed in tabular form. Each combination of machine state and symbol has an associated triple which indicates the new machine state, symbol to write, and direction of movement resulting from that combination of parameters; the complete entry is referred to as a quintuple. The table format is shown in Figure TUR-1.

A parity counter will be developed as an example of a particular Turing machine. The purpose of the parity counter is to count the number of 1's, modulo 2, that are on a tape of 0's and 1's, i.e., detect an odd or even number of

interconnected sequential elements. Use a K(wait-until) and conventional RTM's to solve this problem.

3. An important factor contributing to the unreliability of the punch mechanism is wear. This can be decreased significantly by having the punch off when not punching. Modify the design so that the punch power is automatically turned on and off when the punch is to be used and is not in use, respectively. The punch requires about four seconds to come to speed when started from rest. As an alternative, a circuit might be designed which senses the Punch Synch pulses and detects when the punch is at speed (at speed, Punch Synch signals should occur each  $1/10$  seconds). Note, the punch should not be immediately turned off if a character is missed, but rather the punch should wait until no characters have arrived for say two seconds since, given that a character has been punched, the probability is high that another character will arrive to be punched shortly thereafter. These circuits can be constructed using either an all digital or a control delay (particularly the integrating delay) approach.

4. Design an interface to control the following asynchronous punch. Assume the punch has similar signals to those in Figure PH-3. Anytime the physical punch is free and turned on for 4.5 ins., punch magnet energizing current can be supplied which signifies that punching is to occur. At the end of the punch cycle the Punch Synch Pulse signifies that the punching is completed, and a new punch cycle can be started. Thus, a "model" for this behavior is that when the punch currents are energized a  $1/110$  second delay within the punch is evoked, and at the end of the delay the Punch Synch Pulse is emitted by the punch. Why is the design of an asynchronous punch interface fundamentally simpler than the one discussed above?

## SAMPLED ANALOG INPUT TO PAPER TAPE CONVERTER

**KEYWORDS:** Sampling frequency, punch, analog-to-digital conversion.

A device which can sample an analog waveform a number of times and record the samples in a permanent memory for later analysis is a common scientific instrument. The signal to begin the sampling process is usually evoked externally, and the samples are taken in synchronism with a clock that occurs at the sampling frequency. The inputs to the system are the clock (sampling frequency), the analog signal to be sampled, the signal to start the process, and a signal to denote when the process has been completed. Another necessary input (here an 8-bit number named Sample Identification) identifies the particular set of samples.

## PROBLEM STATEMENT

Design a system to sample and store values of an analog waveform. This particular problem is derived from an actual one, so values have been fixed for the number of samples, maximum sampling clock frequency, etc. At the end of the discussion extensions are proposed which give an excursion into the design alternatives of the system.

The sampling rate is very high relative to the rate at which data can be punched on the paper tape. The paper tape punch operates at approximately one 8-bit character each 1/110 second (i.e., ~10 ms./character). The minimum sampling time is the time of the analog-to-digital conversion plus the time to store the samples in memory in this design. (The design could be modified to operate at a still higher rate.)

to, wait a sufficiently long time for the deciding circuit to decide, thus giving a low probability of undecidability. In many cases being undecided when a decision is made is acceptable because the circuits which make the decision have a value, and it does not matter which decision was made. The reason that a problem occurs in the punch circuit is that the decision is sent to two places which interpret the decision and each may interpret it differently.

Another solution to the problem involves placing a circuit that examines the outputs of a circuit and then gives a signal at a time when the decision is ready and can be examined.

## MORE ON THE SYNCHRONIZATION PROBLEM

Although the reader may think the synchronization problem has been belabored, this is probably not the case. Almost all digital systems require the solution to this problem because they interface with the outside world. Many digital systems engineers do not know that digital circuits behave in the somewhat capricious and arbitrary manner described above when marginal input commands are given.

Some of the phenomena that engineers have used in the past to account for unpredicted behavior are: (1) circuit noise, (2) faulty power supplies, (3) local keypunch (or other equipment) noise, (4) static electricity by the operators who touch the machine, (5) thunderstorms, (6) power failure, (7) noisy input power, (8) intermittent wiring or board failures, (9) heavy cosmic ray showers, (10) someone slamming a door, etc. The point is that many of these failures were probably due to the synchronization problem.

Now, it is important to consider how often an error such as the punch synchronization error might occur. The probability that such an event will occur is the product of several independent events. The events are:

1. A narrow pulse width window at the end of the synchronization pulse. Assume that the window is say 10 ns wide. The probability of this event is:

$$10\text{ns}/(1/110\text{sec})= 1.1 \times 10^{-6}$$

2. Assume that a weak pulse of 10 nanoseconds duration causes only an occasional error, of 1 in 100.
3. Assume that only the first character is likely to cause an error because synchronization occurs after the first character is punched. Also assume that characters are punched in blocks of 100 characters. Thus, the probability of a first character is  $1 \times 10^{-2}$ .

Therefore the probability that a given character will be punched erroneously is about  $1.1 \times 10^{-10}$ . We feel that the given design is satisfactory because this probability of failure is small compared to other

errors having to do with the handling of paper tape. The errors one might expect with paper tape would be in the range of one error each  $10^5 \sim 10^7$  characters. Also, the time between failures is roughly  $10^8$  seconds; which is in the order of the punch's life.

## ADDITIONAL PROBLEMS

1. Even though we need not worry about the errors caused in the punch, consider what happens when a drum is transferring at a data rate  $10^3 \sim 10^4$  faster.
2. The synchronization problem can be solved using an RTM control structure (together with the various transducers required in the previous solution to supply the higher currents). This structure is identical to that of Figure PH-3, except that the implementation of the actual control part is with RTM's instead of

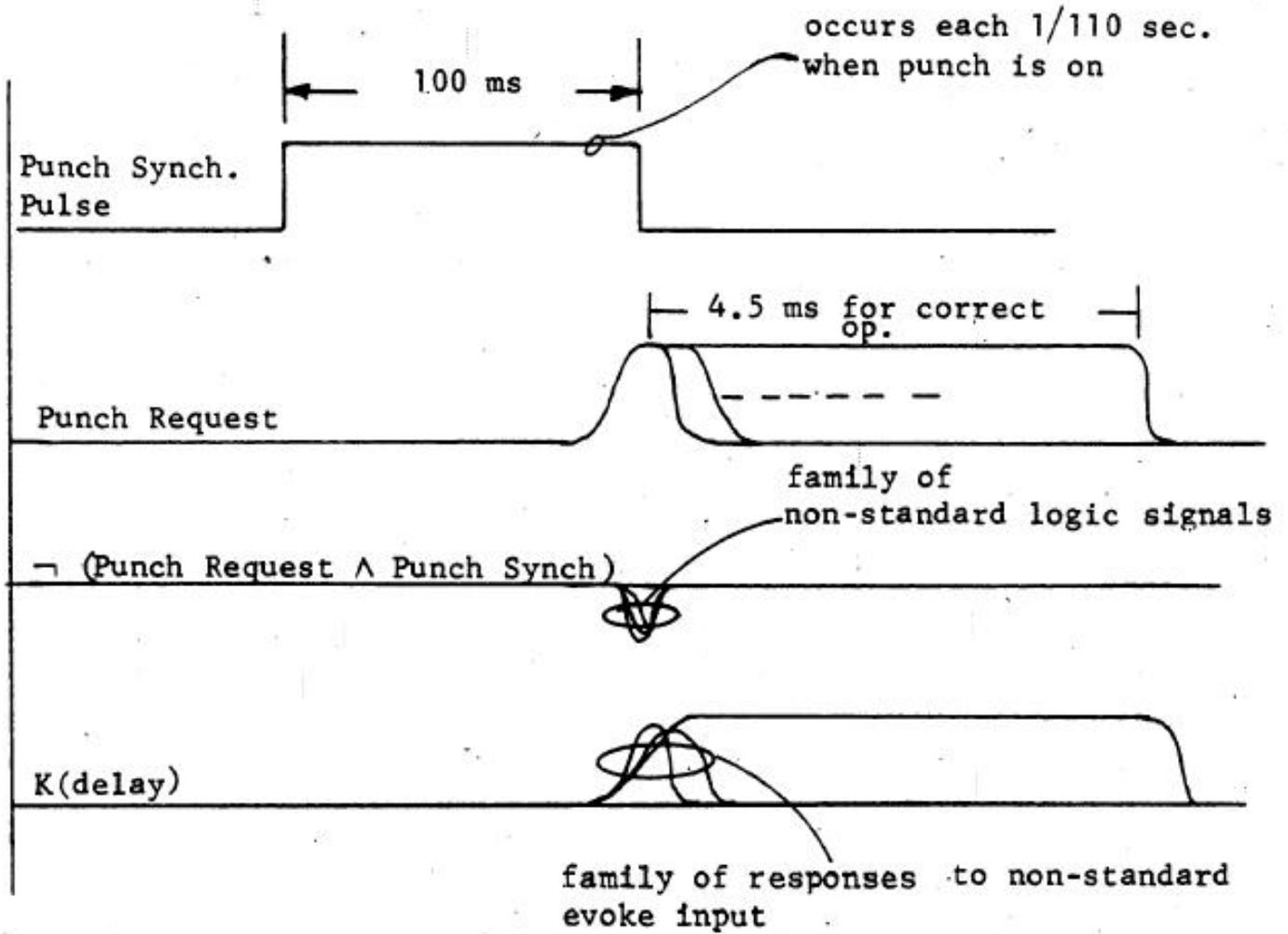


Fig. PH-5. Synchronization of two processes.

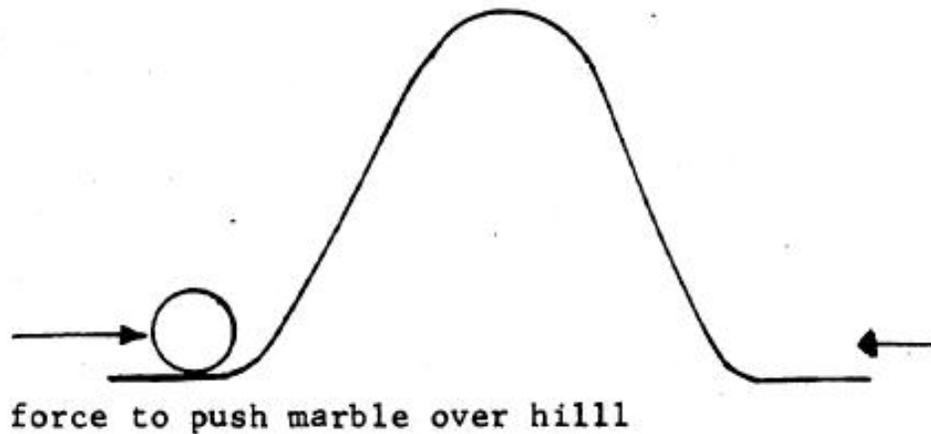


Fig. PH-6. Mechanical dual of memory state undecidability.

Fig. PH-6. Mechanical dual of memory state undecidability.

257

[previous](#) | [contents](#) | [next](#)

the punch process is a parallel merge of the Punch Synch Pulse and the Punch- Request. That is, when both of these signals are present the punching process is started.

## SOLUTION 2

It would seem that the above process and the design as specified in Figure PH-4 are completely correct. To a certain extent this is true, as long as we believe that all systems behave in a fundamentally digital manner. But this example serves to illustrate a synchronization problem inherent in digital systems which have to synchronize two independent processes. Here the Punch Synch and the Punch-Request are being synchronized to create the punching process. When what we think of as a fundamentally digital event (i.e., it is either present or absent) is examined more closely, it is found not to be digital at all. There is some probability that a partial event will occur. The synchronization process can be examined by looking at Figure PH-5. Here, we illustrate the one event in possibly  $10^{10}$  which is erroneous. It can occur as follows:

1. The punch is outputting normal Punch Synchronization pulses of 100 microseconds duration, each 1/110 seconds.
2. A Punch command is given by another process (i.e., the Punch Request becomes 1) at a time near the end of a Punch Synch as shown in the figure.
3. The condition for starting the 4.5 millisecond punch delay is a NAND gate with inputs from conditions 1 and 2 above. The output of this gate is a very small signal (fundamentally not a 1 or 0) which causes the delay to be evoked with some uncertainty. That is, there may not be enough energy in the input signal which is to evoke the punching process. Thus one of the following Occurs: (1) the punch does not operate at all, which is an acceptable condition because it will be evoked the next time by the Punch Synch Pulse; (2) the punch is evoked, which is also acceptable because that was the condition desired; or (3) the punch character signal rises for a few nanoseconds or microseconds and then falls, causing erroneous behavior because the pulse had insufficient energy to fire the delay. For this operation the punch signal may not be long enough to cause punching to occur, but may be long enough to turn off the Punch Request. Therefore, the punch does not operate, but the Punch Ready Flag is set indicating punch completion.

This phenomenon has been discussed in the literature only briefly. A nice mechanical analog of this system (pointed out by the Macromodule group at Washington University, St. Louis) is a marble which is being pushed over a frictionless hill by impulses on each side (shown in Figure PH-6). Here, a strong force normally pushes the marble so that it will rise and go to the other side of the hill. If the force is too small, it will fall back; if too large, it will always go over the hill. If the force is just at the right magnitude, the marble will go to the top of the hill, remain there for a time which is undecidable except by a probability measure and then fall to either side. Circuit solutions to the problem place amplifiers at

the inputs (which increase the probability that a good pulse will always be emitted). Other circuit solutions increase the storage device circuit gain (analogous to making a pointed hill). Another solution is to add a specification to all multiple state devices which indicates the probability of deciding a state within a certain time, given a particular pulse input amplitude and duration.

The solution used in the RTM circuits when synchronization problems arise is



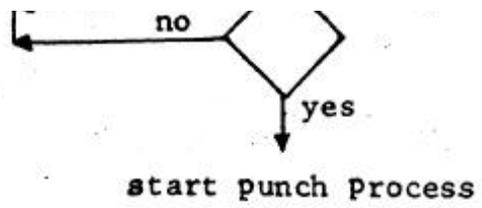


Fig. PH-4. Control behavior of conventional logic.

254

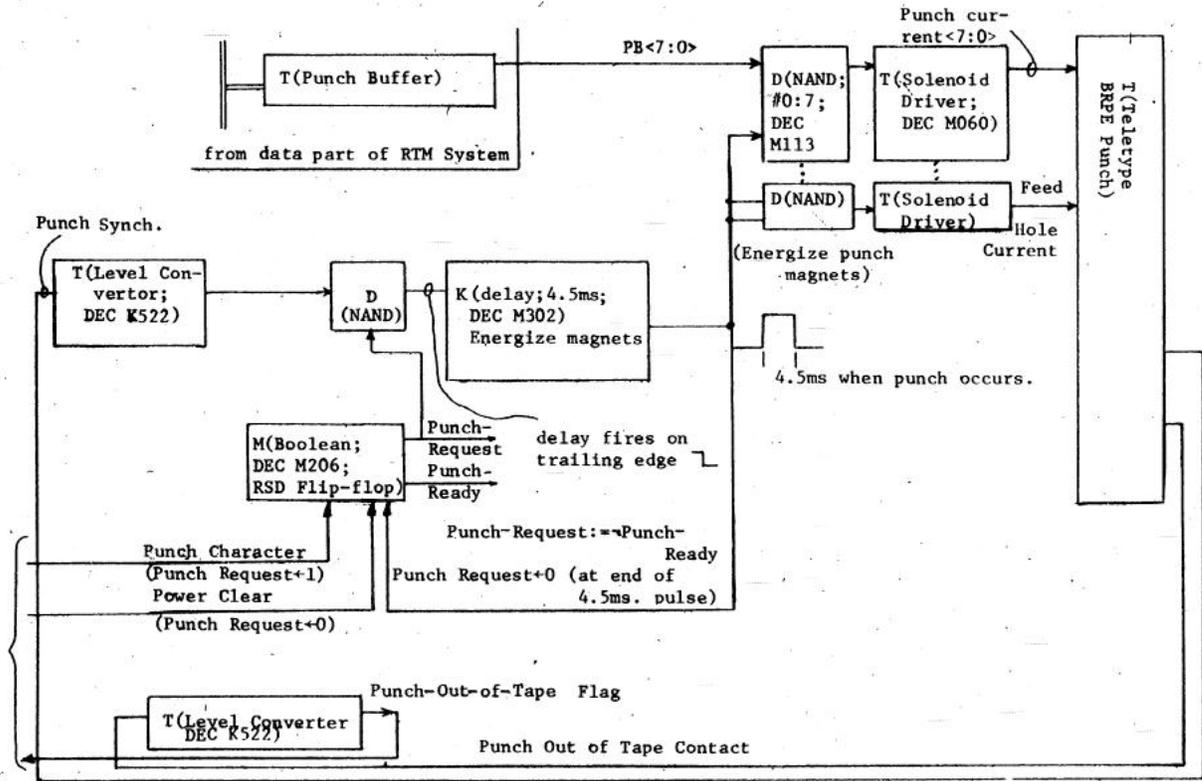


Fig. PH-3. T(paper Tape Punch) using Conventional Logic.

accepted as given, and the RTM control must adapt the punch to the RTM system. - Alternatively, it may be necessary to modify the definition of the punch as an RTM component if the design indicates such a solution.

The Teletype BRPE punch has eight punch magnets (solenoids). The punch solenoids require a current of several hundred milliamps, and the current is supplied from a +30 volt power supply. In order to convert logic signals to high currents the control portion of Figure PH-2 must contain T(solenoid driver)'s so the Punch Buffer data signals can be matched to the punch solenoids. A ninth solenoid current must be supplied at the same time as the data in order to punch a small hole, called the feed or sprocket hole. The hole is between data bits 3 and 4 and is used by a paper tape reader as a synchronization signal. For some readers the sprocket hole is used to pull the tape through the reader. The punch has a pulse output signal of several hundred microseconds duration and amplitude of about 30 volts. This Punch Synchronization Pulse occurs when it is possible to punch a character on the tape. In essence, one can think of the punch as a flywheel that rotates at the rate of 110 revolutions per second, and only during a particular portion of the flywheel can punching occur. In order to punch (when a punch request is pending) the punch magnets are energized for exactly 4.5 milliseconds following the Punch Synchronization Pulse. No characters are punched and the tape is not moved if none of the nine punch magnets are energized.

## SOLUTION 1

With this background a straightforward design of the RTM control can be posited. T(solenoid driver)'s and T(level converter)'s are required to energize the punch and convert the return signals from the punch so that they are compatible with the logic system. The solenoids are timed by a K(delay) of 4.5 milliseconds. A DMflag is used for the Punch-Request Flag. D(AND)'s are used to conditionally operate (switch) the respective solenoids so that the correct information will be punched corresponding to the bits in the Punch Buffer. Figure PH-3 shows the structure of the RTM punch control in terms of conventional combinational and sequential logic elements. The parts are taken from the DEC catalog.

The behavior, of this control is expressed in the flowchart of Figure PH-4. There are two independent (unsynchronized) processes: the running of the punch and the user RTM process which issues the command to punch a character. The two processes achieve some natural synchronism after the first character has been punched, since the RTM process must wait until the punching is complete before proceeding. The punch can be thought of as a clock pulse which occurs each 1/110 seconds and lasts for about 100 microseconds, the duration of the Punch Synchronization Pulse.

The other independent process of the RTM system is the occurrence of the Punch-Request being set to 1. The two processes are synchronized by a polling process in the righthand part of Figure PH-4. 'While the Punch Synch Pulse is present the Punch-Request is polled and, if present, the punching process is initiated. If the Punch Synch Pulse becomes 0 (indicating the end of the 100 microsecond pulse) the polling stops, and punching cannot occur until approximately 1/i 10 seconds later. The punch process is

the four step sequence of turning on the delay, delaying 4.5 ins., turning off the delay and resetting the Punch-Ready Flag to 1 (signifying that a new character can be loaded into the Punch Buffer). As an alternative way of looking at the synchronization between the two processes (used in this implementation), starting

loaded with a character and a command, Punch Character, is given. This has the effect of setting a DMflag; called the Punch-Request Flag, to a 1. The punching mechanism, sensing the Punch-Request Flag as a 1, proceeds to punch the character.

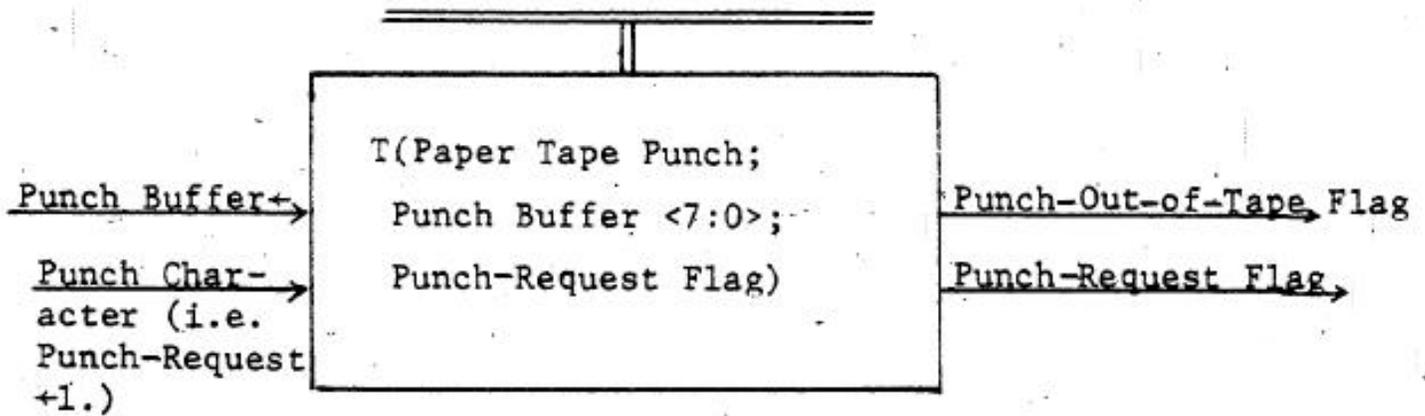
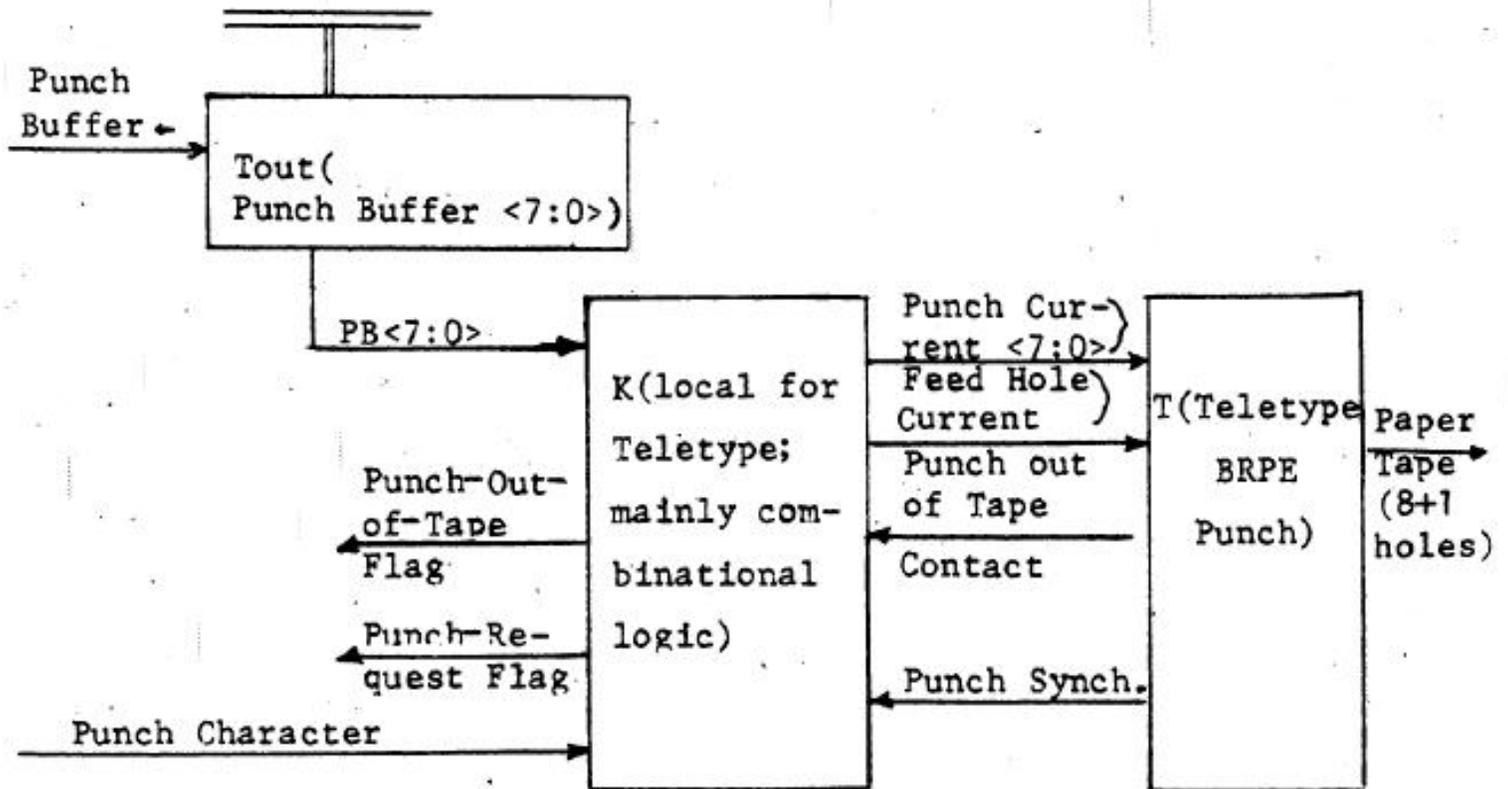


Fig. PH-1. Module diagram of T(Paper Tape Punch).



## Fig. PH-2. PMS diagram of RTM Module T(Paper Tape Punch).

When completed with the punching the Punch Request Flag is reset to 0. Another signal is available from the punch, indicating that it is out of paper tape. This signal originates from the closure of a switch contact pair. The Punch Buffer is loaded from the least significant eight bits of the data Bus. In order to define the operation of the RTM punch control the Teletype punch will be

The Octave bits (Note  $\langle 1:0 \rangle$ ) are placed into two DMflag's before dividing N' by 4. Later they control the left-shifting of N" into lower octaves (the more shifts, the lower the octave). This scheme avoids using an M(transfer).

## ADDITIONAL PROBLEMS

1. Design an advanced instrument to play 3-part harmony over a three octave range. Then extend to four parts and/or four octaves.
2. Design an instrument whose note frequencies are more accurate. A read-only memory may be used.
3. For these more complex instruments what code would be used to control the device? Is it necessary to transmit at a higher data rate? Can you encode each event (note or rest) into two characters?
4. How would you modify the basic waveform generator to vary volume? - to change the shape of the waveform to better approximate a sine wave? - to approximate other (variable) musical instruments? - to change waveform under computer control?
5. It was assumed in the design above that when the same note is received two or more times in succession, no perceptible break would occur, and one continuous note would sound. Is this really so? If not, how would you redesign to make it so?

## DESIGN OF INTERFACE TO A PAPER TAPE PUNCH

**KEYWORDS:** Paper tape punch, synchronization, nondigital behavior, transducer.

Several problems assume the existence of a T(paper tape punch); therefore we will design an interface for one. A unit of this type appears to be similar to the T(Teletype ASR 33; printer; keyboard; paper tape reader; punch). The design problem is given for several reasons; it is typical of the devices interfaced to digital systems, yet it is fairly simple; there are interesting pitfalls in the obvious synchronization technique, hence there will be extensions to the problem to increase the reliability; and the design will be carried out with both conventional and RTM logic. In any case, it is necessary to go outside the RTM logic framework for Transducer modules (e.g., to interface to the high current punching magnets).

## PROBLEM STATEMENT

Design a system which will serve as an interface between the Teletype Model BRPE paper tape punch described below and the RTM system. The punch will appear as a conventional module in the RTM system.

## DESIGN CONSIDERATIONS

The overall structure of a T(paper tape punch) as seen by an RTM user is shown in Figure PH-1. A register, the Punch Buffer, holds the eight bit character to be punched. The character is eventually punched on a one inch wide paper tape, and 10 characters are punched per linear inch of tape. The physical punch is manufactured by Teletype Corporation as the Model BRPE. The punch operates at the rate of 110 characters per second.

Figure PH-2 shows the internal structure of the T(paper tape punch) system of Figure PH-1 in the next stage of detail. In this figure the signals to the physical Teletype BRPE punch are depicted along with the logic that performs the actual punching process. In order to punch a character the Punch Buffer is

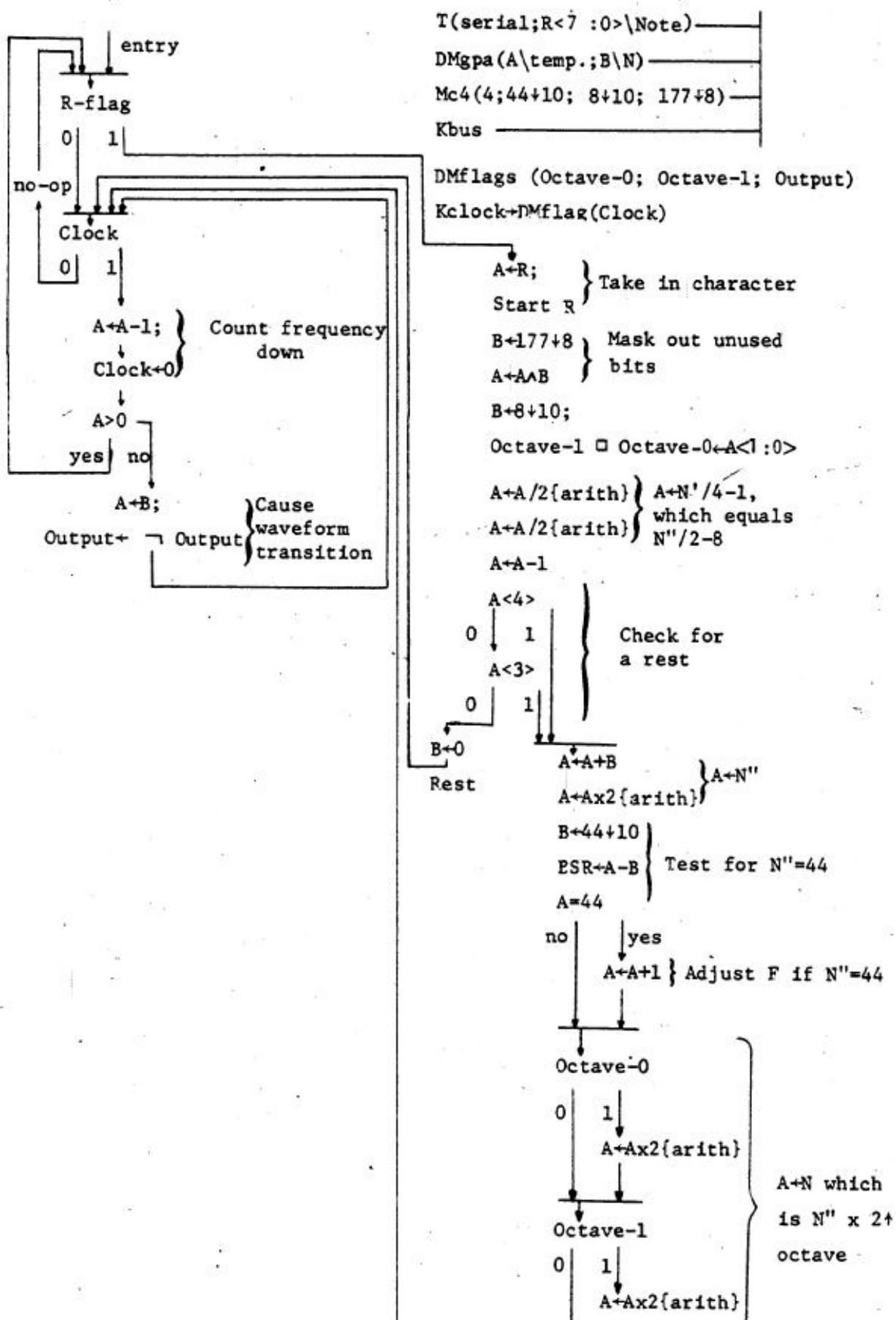


Fig. Tel-3. RTM diagram for Teletrola

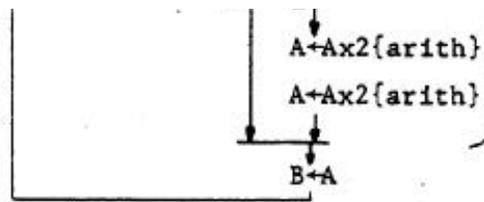


Fig. Tel-3. RTM diagram for Teletrola.

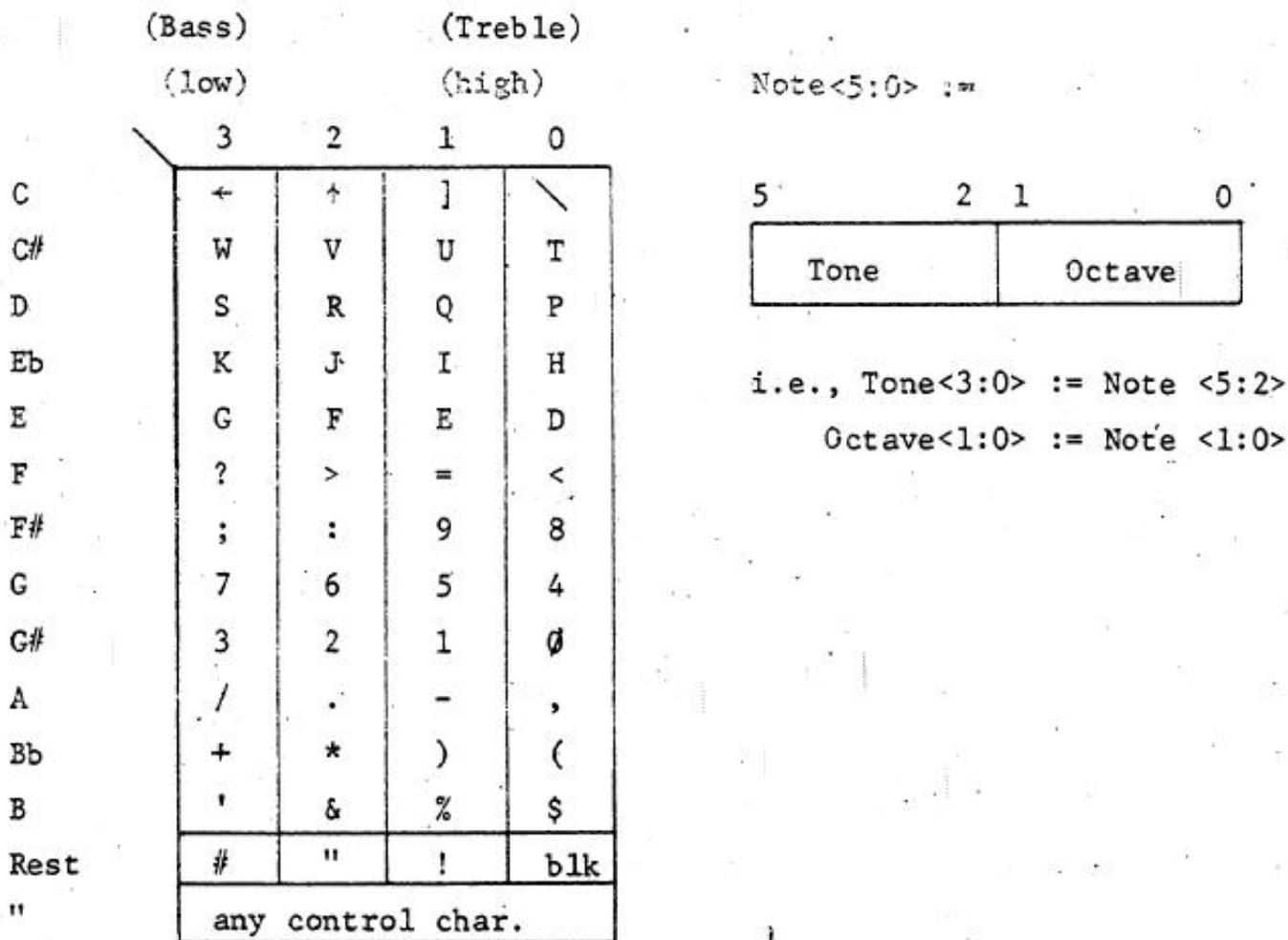


Fig. Tel-2. Table of ASCII characters for 'tones and octaves.

All this needn't be computed at the computer transmitting end -- the character for any given pitch can be looked up in a table such as Figure Tel-2. However, since the device at the receiving end has no memory for such a table, it must compute to "decode" the characters into notes. Treating the incoming character as a binary number N', we can recover N by the formula (see Figure Tel-1.):

$$N'' = (N'/4 + 7) * 2 = N'/2 + 14$$

Note that the octave bits are shifted out when N'/4 is done, so they have to be stored before-hand. In the special case of tone F, whose N'' is 45, we check for N'' = 44 and then add 1. The N for the correct note (tone plus octave) is then:

$$N = N'' * 2 ^ \text{octave}$$

Note that octave 3 is the lowest (bass), 0 is the highest (treble).

The RTM terminal design in Fig. Tel-3 implements the above scheme and treats any character below decimal 36 (octal 44) as a rest. Rests are actually a tone 32 times the highest note's frequency, well above the range of human hearing. This system sounds the last note (or rest) transmitted until another comes in. The T(serial) holds the latest input note, N'.

The output is a square wave from DMflag(Output). Dividing by N is accomplished by decrementing the A register by 1 on each clock pulse. When A reaches zero, A is reloaded with N from the B register, and Output makes a transition to give half an output cycle.

248

<u>Tone</u>	<u>N</u>	<u>N/2</u>	<u>N/2-15</u>	<u>4*(N/2-15)</u>	<u>(N'=2N-28)*</u>	<u>N' (octal)</u>	<u>Ideal N</u>
C	60	30	15	60	92	134	60.0 (defn.)
C#	56	28	13	52	84	124	56.632
D	54	27	12	48	80	120	53.454
Eb	50	25	10	40	72	110	50.454
E	48	24	9	36	68	104	47.622
F	45	22 <sup>+</sup>	7 <sup>+</sup>	28 <sup>+</sup>	60	74	44.949
F#	42	21	6	24	56	70	42.426
G	40	20	5	20	52	64	40.045
G#	38	19	4	16	48	60	37.798
A	36	18	3	12	44	54	35.676
Bb	34	17	2	8	40	50	33.674
B	32	16	1	4	36	44	31.784
(Rest)	≤31	15	0	0	≤32	≤40	- -

\*  $N = N' / 2 + 14$

Fig. Tel-1. Table of divider values (N) for tones, and derivation of transmission code N' for treble octave.

5. The output feeds into a conventional audio amplifier and speaker.

## Musical Requirements

1. Only one musical part (note) need be played at a time (no harmony).
2. (Forced by technical restriction 3) "Tempo" must be determined by the incoming character rate. Each character corresponds directly to one note -- nominally a 16th, 24th (triplet 16th) or 32nd note. The last character. (note) received .is played until a new note is received and overwrites it. Thus all notes are "legato, that is, continuous-sounding. "Staccato" notes and other phrasing (breaks) are sounded with rests.
3. At least one character must produce a "rest" or null-note (no sound produced). Carriage-return (octal 15) and line-feed (octal 12) should also evoke rests, due to computer output-buffer line-length limitations.
4. Intonation (pitch accuracy) need not be perfect, but should be close enough on all 12 tones of the chromatic scale (includes sharps and flats) that it will not offend a musical ear (the lack of harmony gives extra leeway). Absolute pitch is not important (this can be adjusted from the clock), but the tones should be in tune with one another (relative pitch). A 1.0% error or less is desired, where error is defined as follows: If the desired frequency for perfect relative pitch for a tone is  $d$ , and the actual frequency is  $a$ , then percentage error =  $|1-d/a|*100$ . This error measure is derived from the fact that in music ratios of frequencies, rather than frequency differences, determine tone intervals.
5. Pitch range should be at least three or four chromatic octaves.
6. It is acceptable to generate tones with a square wave, hence tone quality (timbre) is quite simple.

## SOLUTION

One method to generate a variety of frequencies (pitches) is to use a clock running at a constant frequency,  $F$ , much higher than desired, and divide  $F$  by various integer numbers. To divide the clock by  $N$  means that on every  $N$ th pulse from the clock, the divider outputs one pulse. Thus output frequency varies inversely with  $N$ .

$F$  must be an integral multiple of each output frequency desired. However, the 12 tones of the chromatic scale are not all in integral ratios with one another, making exact pitches impossible by the divider method, so some approximation must be used.

Figure Tel-1 shows a tolerable approximation for one octave, based on 60- that is, the lowest note, C, is F/60. To get four octaves, one can encode the 6 bits of a character as shown in the format in Figure Tel-2. Here, we shall use the word Note to denote the combination of a tone (one of the 12 in the chromatic scale) and the octave in which it is to be played. Thus bits 5 to 2, of Note <5:0> give the value of N for the note (Tone <3:0>), and bits 1 to 0 give the desired octave (Octave <1:0>): In Figure Tel-1, N ranges from 32 to 60. The problem is to fit this into the 4 bits of Tone <3:0>. This can be done by dividing N by 2, giving a range of 16 to 30. Then, subtracting 15 gives a range of 1-15 which just fits into 4 bits. If Note <5:0> is treated as an integer, then the range 1-15 of Tone <3:0> becomes the range 4-60 in Note <5:0>. Adding the codes for the 4 possible octaves gives Note a range of 4-63.

Recalling technical restriction 2, the 64 character subset, octal 040 through 137, is used in transmitting notes to Teletrola. Thus, if we add decimal 32 to the decimal 4-63 range for Note, we get decimal 36-95 (octal 044-137), which encodes into the desired character set.

- sampled data with transfer to the transmit buffer, T. The transmit and receive buffers are registers in a DMgpa. (Note that in this problem an 8-bit DMgpa is used.)

The control part of the system is comprised of a principal control loop and three subroutines. The control loop is a polling process which checks the DMflags, T-clock, and R-clock, for an indication that data is to be transmitted or received. On transmission, the data in the DMflag (T-data) is transferred to the DMflag (T-out), the data in the transmit buffer, T, is shifted left one bit, and the most significant bit (shifted to OVERFLOW) is placed in T-data prior to the next clock signal. On receiving input, each new bit is shifted into the receive buffer, R. When the start bit is shifted out of R into OVERFLOW, the nine bits of the message have been received. Thus, when OVERFLOW is set, the 5-bit channel number is transferred to the analog multiplexor switch which selects the appropriate channel, and an analog sample, is taken. The T(analog-to-digital) converts the analog sample to digital form and the data is transferred to the transmit buffer, T.

## TELETROLA, A MUSICAL TERMINAL

Design by Michael Knudsen

**KEYWORDS:** Tone synthesis, frequency division, Teletype

Most of us are familiar with devices known as electronic music synthesizers. These usually consist of various analog tone generators driven by a human playing a keyboard. In this problem we shall deal with the design of a synthesizer which generates tones digitally, and is driven by incoming, serial digital data of the form used by a Teletype.

### PROBLEM STATEMENT

Design a musical instrument which connects to a bit-serial communications line and interprets incoming characters as notes to be played. For the purposes of this problem, assume the bit serial line comes from a Teletype, hence the name Teletrola for the musical instrument. Since nearly all computers have interfaces for low-speed (10, 15 or 30 characters/sec) typewriter-like terminals and use bit-serial (or Teletype) format data, Teletrola could be easily interfaced with a computer.

The design should adhere to the following restrictions:

### Technical Restrictions

- 1. The terminal is a 10, 15 or 30 char/sec Teletype using the 6-bit subset of the 7-bit ASCII code of book Table 5. The data which gets transmitted to the device has only the 64-character subset, octal 040 through 137, plus carriage-return (015) and line-feed (012). This includes capital letters,

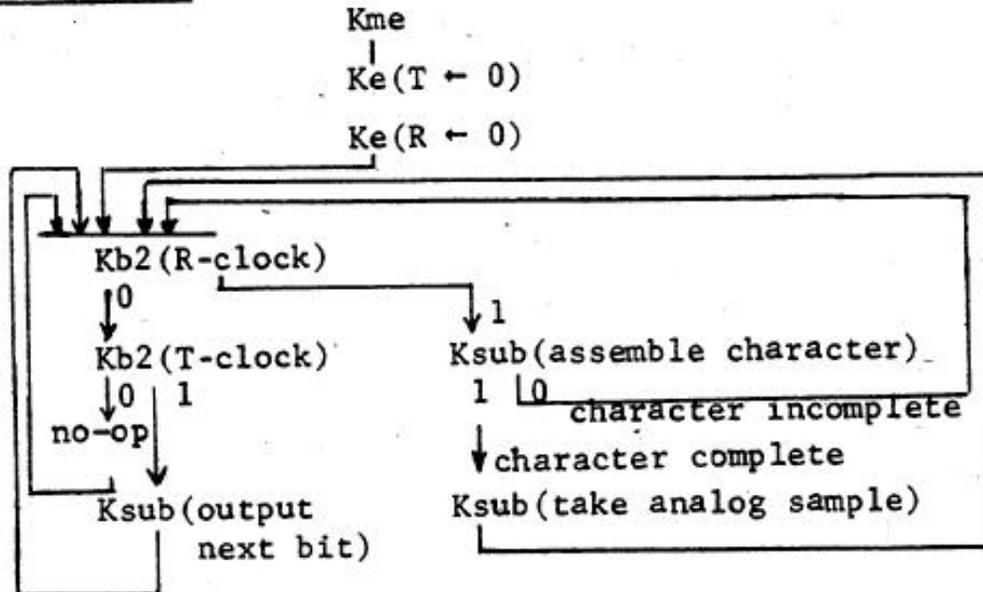
special characters, and digits, but no small letters or other control codes.

2. No buffer memory is available to store incoming data. Characters must be processed (played) as received.

3. No addressable read-only memory (ROM) is available for table-lookup operations, just a few constants registers. Try for one M(constants; 4- words).

4. An adjustable Kclock is used to set a DMflag, to provide a real time measure.

control part



subroutine: output next bit

```

↓
Ke(T-clock ← 0)
↓
Ke(T-out ← T-data)
↓
Ke(T ← TX2 ; Set OVERFLOW){logical}
↓
Ke(T-data ← OVERFLOW)
↓

```

subroutine: assemble character

```

↓
Ke(R-clock ← 0)
↓
Ke(R ← Rx2; LSI := R-data; Set OVERFLOW)
↓
Kb2(OVERFLOW)
↓ 1      ↓ 0
↓        ↓
character character
complete incomplete

```

subroutine: take analog sample

```

↓
Ke(R ← R/2) {logical}
↓
Ke(R ← R/2) {logical}
↓
Ke(R ← R/2) {logical}
↓
Ke(mpx ← R)
↓
Ke(R ← 0)
↓

```

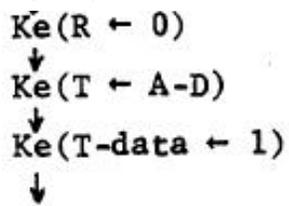


Fig. ADSU-3b. RTM system diagram of a remote analog-to-digital sampling unit (control part).

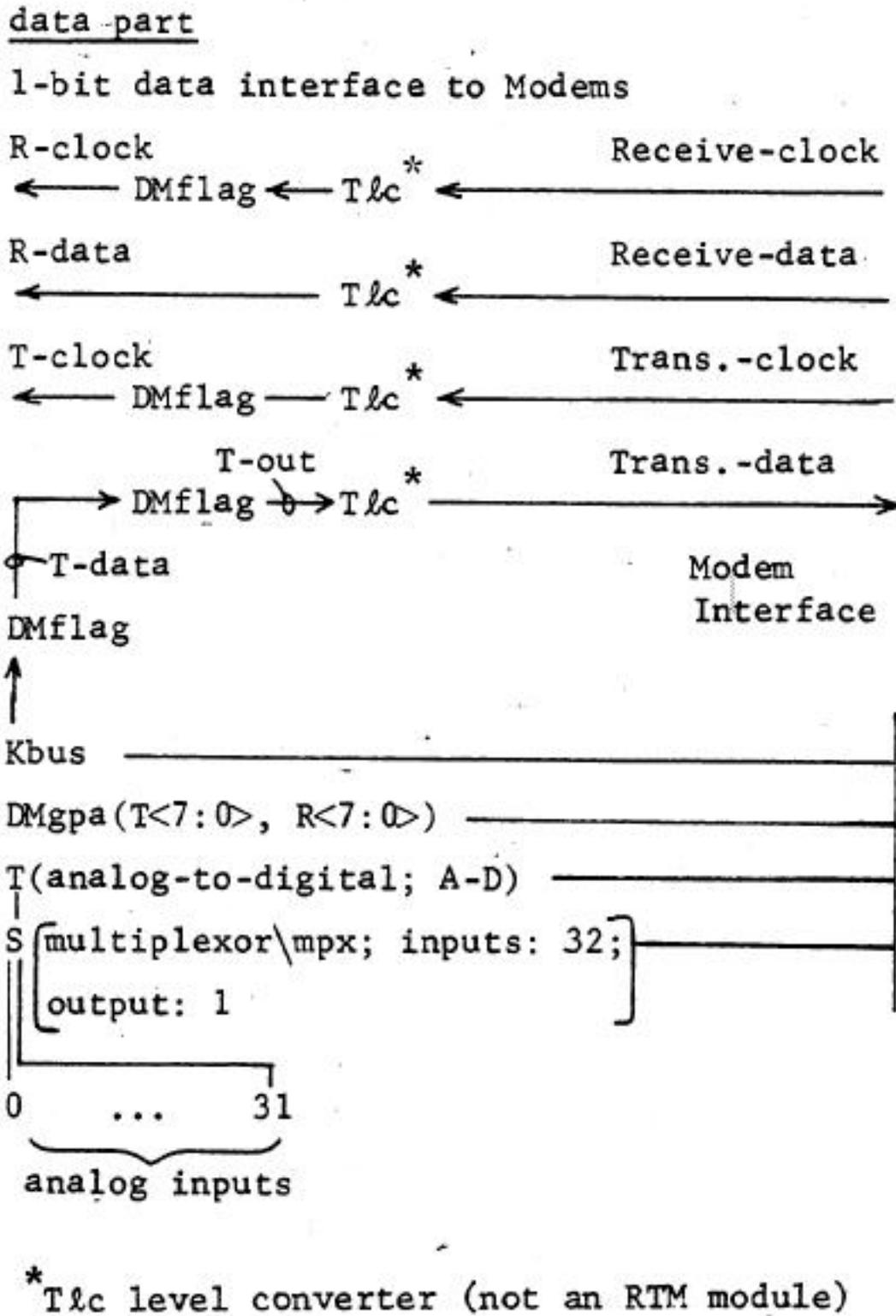


Fig. ADSU-3a. RTM system diagram of a remote analog-to-digital sampling unit (data part).

digital form and are stored in the DMflags, R-clock, and T-clock, respectively. Whenever a clock signal is present on the line, the appropriate Boolean register (flag) is set to one. The data on the

incoming data line, Receive-data, is valid at the time of the clock transition, i.e., when the DMflag(R-clock) is set the signal has been converted and is available in digital form. When the DMflag(T-clock) has been set, outgoing data from the sampling unit is transferred from the DMflag(T-data) to the DMflag(T-out); T-clock is set to zero immediately before the transfer.

2. The data part associated with the Bus carries out three functions: serial to parallel conversion of the analog channel number; sampling of the appropriate analog channel (the multiplexor\mpx register is set to the desired channel number); and parallel to serial conversion of the

Certain characters may be used as control characters, e.g., start of header (an identifier), start of text, end of text, end of block, end of transmission, acknowledgment (message o.k.), negative acknowledgement, parity check, etc. When no data is being transmitted, a special, idle line (synchronizing) character is continuously transmitted. Data characters are, of course, those characters not reserved for control. In essence, the component interfaced to a modem looks at the data through a continuous 8-bit wide window; synchronization is important so that the component only looks through the window to retrieve a character at proper intervals.

2. Start-stop - With the start-stop scheme, an idle line is signified by a constant string of zeros (ones). A character is transmitted by sending a one (zero) followed by the data bits of the fixed length word.

It should be remembered that the components connected to the modems transmit their data to the modems by one of the methods described above. It is of no concern to the components how the modems communicate with one another. The transmission data rate, though, is determined by the link between the modems. The rate for a telephone line is 2400 bits/second, although rates of 4800 and 9600 bits/second are also used. For even higher data rates, special communications lines (and modulation techniques) are employed; these methods provide rates of 40~50 Kilobits/second, and megabit rates are possible.

In the design presented below, characters will be transferred using the start-stop transmission scheme. A request from the central unit for a sample from the remote unit is signified by a 1(start bit), followed by the 5-bit channel number, followed by three 0's (stop bits); i.e., lxxxxx000 in which xxxxx represents a 5-bit channel number. The remote sampling unit takes a sample from the appropriate channel, converts the analog sample to digital form, and transmits the data in the format lyyyyyyyy where the 1 is the start bit and yyyyyyyy is the 8-bit digital encoding of the sample. Fig. ADSU-2 shows a diagram of the modem and the remote sampling unit.

There are two other considerations concerning the timing of the system that should be noted. The remote sampling unit must have sufficient time to process either incoming or outgoing bits of information. Obviously the system cannot function properly if the modem is transmitting the sampling unit information faster than the sampling unit can process the data. Equally important, the modem cannot be allowed to request data from the sampling unit at a higher rate than the sampling unit can provide the data. A check of the execution times of the subroutines, comprising the remote sampling unit (see below) will indicate that even at a moderately high transmission rate, e.g., 10,000 bits/second, the inter-bit time gap is sufficient for all processing that is necessary. A second assumption that must be made is that the transmit clock operates at an equal or faster rate than the receive clock; if not, a potentially infinite buffer would be needed at the sampling unit to hold the build up of bits that would occur. (Show how this build up would occur.) For this problem, assume that the transmit clock of the modem is faster than the receive clock by 0.1%. (Show why this time differential does not lead to any difficulties).

## SOLUTION

An RTM implementation of the remote analog-to-digital sampling unit is shown in Figure ADSU-3. The system has two data parts:

1. There are four i-bit interfaces to the communications modem. The two clock signals from the modem, Receive and Transmit, are converted to

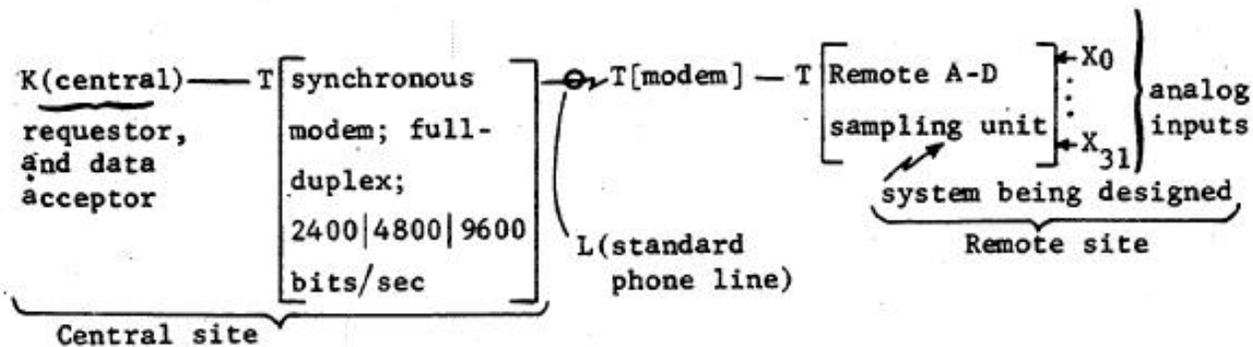


Fig. ADSU-1. PMS diagram of a remote sampling unit with communication facilities to a central control unit.

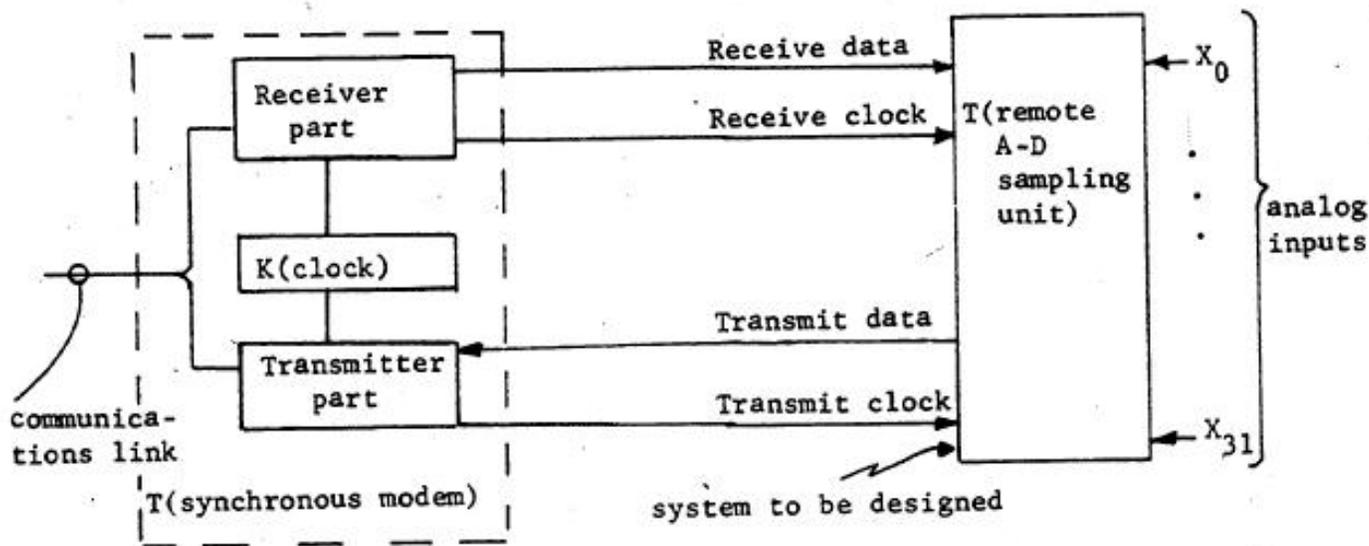


Fig. ADSU-2. PMS diagram of the remote sampling unit and its interface to a modem.

[previous](#) | [contents](#) | [next](#)

This problem is concerned with the conversion of analog data to digital form at a remote sampling unit, and the subsequent transmission of the digital data via a full duplex, synchronous communications link to a central controlling unit. A complete system, shown in Figure ADSU-1 consists of these components: (1) a central controlling unit (actually a computer) which issues commands to and receives data from the remote sampling unit; (2) a synchronous modem at the central unit which converts digital signals originating at the central unit to a form suitable for transmission along the communications link and vice versa; (3) a communications link to the remote sampling unit; (4) a synchronous modem at the remote unit which converts signals in the communications link format to digital form and vice versa; (5) a remote analog-to-digital conversion unit and switch (multiplexor) which selects one of 32 analog inputs for sampling and converts the analog signal to digital form for transmission back to the central unit.

The general operations sequence of the system is as follows:

1. The central unit transmits a code, which specifies a channel to sample, to the remote sampling unit via the modems and the synchronous communications link.
2. The remote sampling unit samples the channel specified by the code, converts the analog data to digital form, and transmits the result back to the central control unit via the modems and the synchronous communications link.

## Synchronous Modems

A synchronous communications modem performs two types of data representation conversion. A modem can convert signals arriving along a communications link to standard digital form by frequency shift keying (frequency modulation), for use by the equipment connected to it. The modem can also perform the inverse conversion -- it accepts standard digital data and converts it to a form suitable for transmission along the communications link. The communications link is full duplex, that is, information can be transmitted and received simultaneously by the modem attached to the link. Although a synchronous, full duplex modem, such as a Bell System 201A or 201B (2400 bits/sec), is a relatively complex component internally, only the two pairs of signals (see Figure ADSU-2) which form the interface to the sampling unit will be considered in this problem. Thus, stated information about the modem, such as it being on or off, whether it has just come on, etc., will be ignored. The two modems and the communications link (phone line) in Figure ADSU-1 can be thought of as a strictly digital link, with clocks that control the data flow along the link. A modem has two clocks: a transmit clock, which determines when the modem transmits a bit along the link, and a timing signal (clock), which is set when the modem detects the arrival of a bit from the communications line. The communications line is never idle; a 0 or 1 must always be transmitted.

## PROBLEM STATEMENT

Design the remote analog-to-digital sampling unit.

## DESIGN CONSIDERATIONS

A scheme must be developed for transmission of data from the remote sampling unit to the control unit. Two possible schemes which are commonly used are:

1. Character transmission - A fixed length character is always transmitted. A common size is 8 bits/character which yields 256 possible characters.

3.(2) Examine the feasibility of a digitally controlled, sampling switchboard for communications lines carrying {asynchronous| synchronous) communications data in the form found at the digital interface side of a modem (see Figure TRAN-8). Switching can be performed by sampling the incoming lines at a very high rate and transferring the samples to outgoing lines. The virtual interconnection of the links  $i$  and  $j$  is shown in Figure TRAN-9. Here, the receiver of  $i$  is connected to the transmitter of  $j$  and the receiver of  $j$  to the transmitter of  $i$ . The system would require a 1-bit memory for each half of each line at the transmitter, since the outgoing line must be held between sample times and the switching process consists of taking incoming samples at the receiver and passing them on to the transmitter bit.

The most important parameter of the design is to make the sampling process time,  $t_s$ , for all the lines short so that a high sampling frequency is possible. The number of lines,  $n$ , that can be switched is a function of the maximum allowable distortion,  $d$ . The percentage of the variation in the bit-time the sampled transition times occurs and the bit data rate,  $r$ , determine  $d$ .

Another problem of the design is determining the control part of the system. As indicated in Figure TRAN-8, a standard full duplex communications link would give the control information concerning the switching paths. That is, the control link gives information designating that a line,  $i$  is to be connected to another line,  $j$ . This link could also transfer switch status information. The sampling switch uses this information to carry out the switching (by sampling). (Why each pair is to be connected is another story and problem.) The switching request information is usually contained on the incoming line. Thus information as to which incoming line is to be switched to another line is a difficult problem. You can determine designs which perhaps, ignore the problem-but hopefully you can solve it. Some solutions may require a modem signal that tells when a line is turned on.

## ANALOG TO DIGITAL CONVERTER

Assuming that only a T(digital-to-analog) converter and a D(analog comparator) are available, design a family of T(analog-to-digital) converters. A D(analog comparator) has two analog inputs,  $I_1$  and  $I_2$ , and one Boolean digital output. The relationship between output and input is  $B := I_2 > I_1$ ; that is, if  $I_2 \leq I_1$  then  $B$  will be false\0, and if  $I_2 > I_1$ , then  $B$  will be true\1.

By interconnecting these two components, finding a digital value corresponding to an analog input can be considered as a kind of guessing game. A digital value is given to the T(d-a), and the control for the conversion is able to find out whether the analog input is greater than or equal to the number given. Thus by systematic searching the value of the input can be found.

Examine various search strategies, determining the cost and conversion time. (Two obvious candidates are:  $t \sim k_1 \times \text{input-value}$  and  $t \sim k_2 \times \log_2(b)$ , where  $b$  is the number of bits in the answer.)

## REMOTE ANALOG-TO-DIGITAL SAMPLING UNIT

## WITH MULTIPLE CHANNEL INPUT

**KEYWORDS:** Synchronous, communications link, transmit, receive, modem, sample

-----

2. This design was suggested by Mr. William Broadley, manager of the Carnegie Mellon University Computer Science Department Engineering Laboratory.

240

[previous](#) | [contents](#) | [next](#)

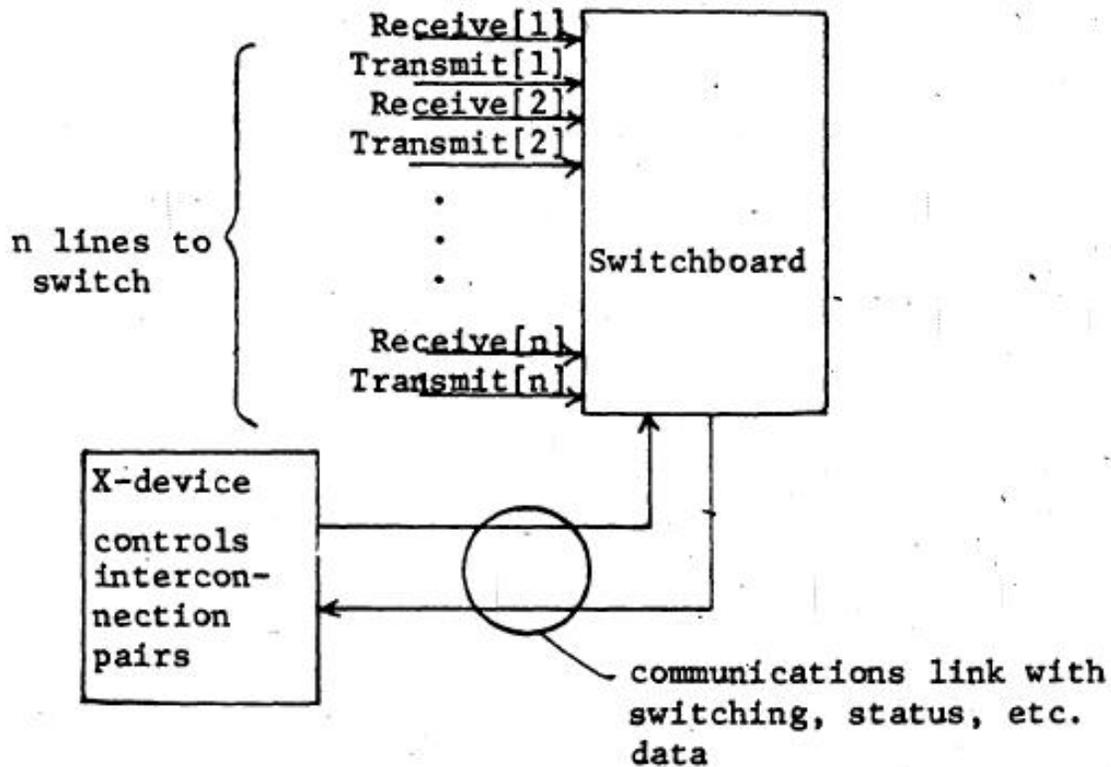


Fig. TRAN-8. PMS diagram of a switchboard for digital data communications.

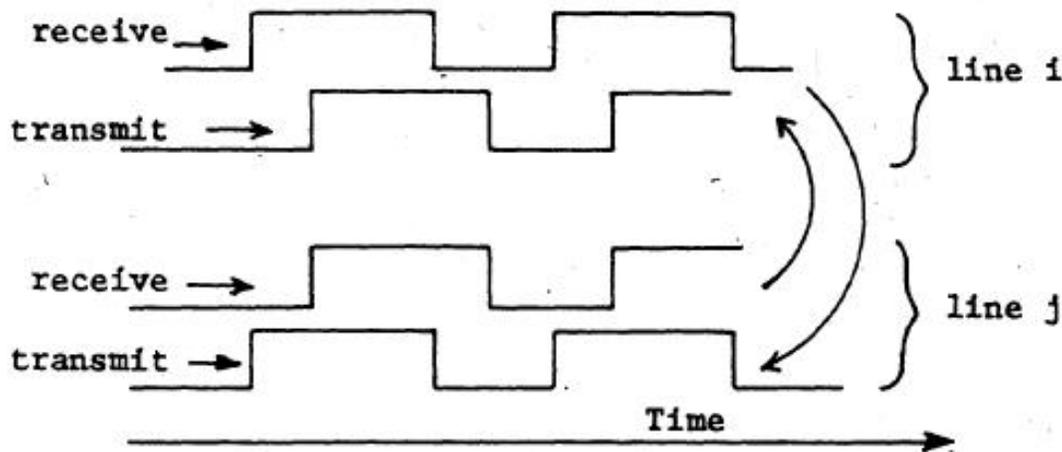
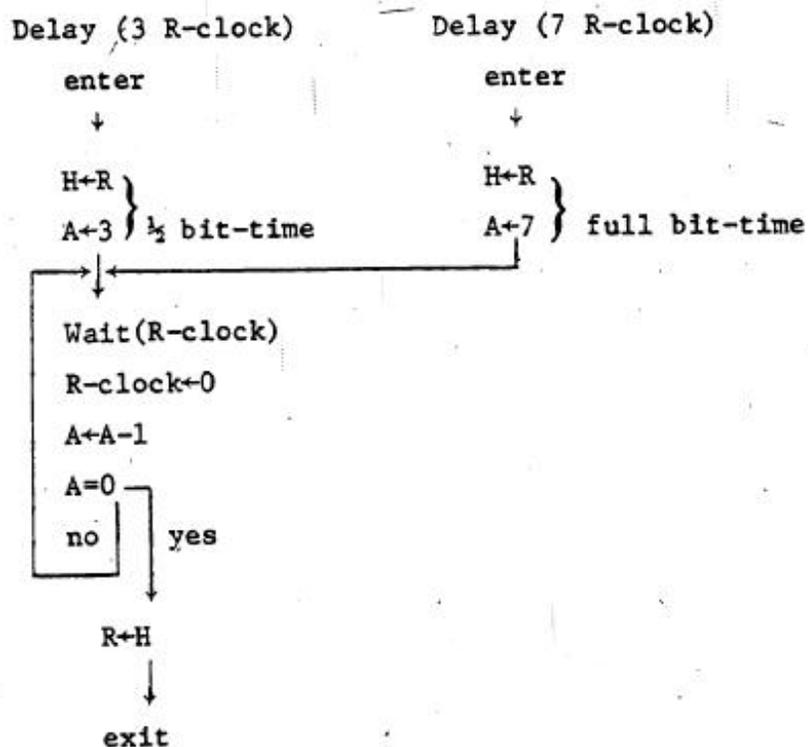
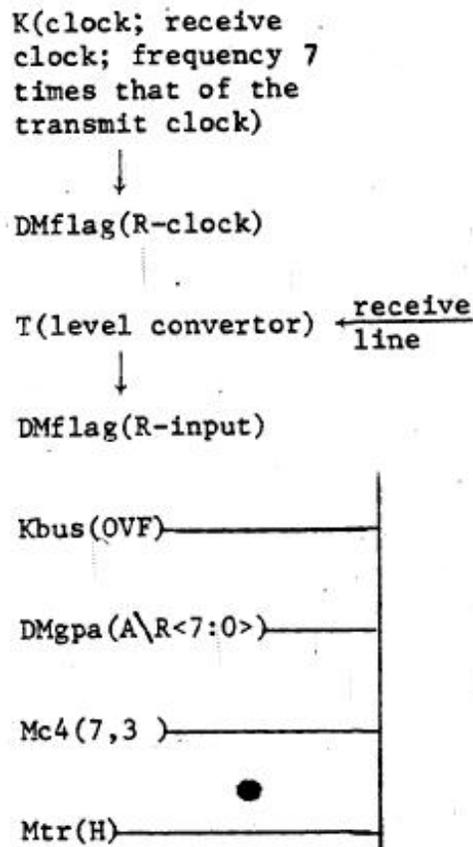
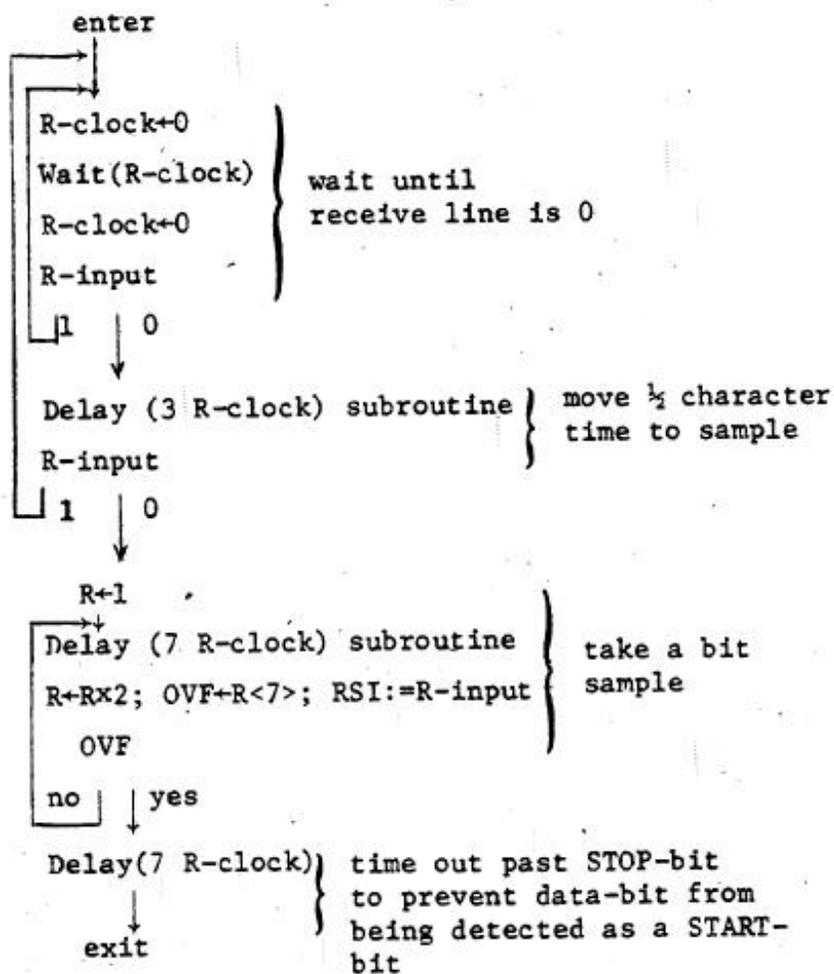


Fig. TRAN-9. Timing diagram showing interconnection of full-duplex links.

Some of the design considerations are: RTM system cost, total system cost, interference time taken away from the computer for bit or character accessing, computer program time. At one extreme, the communications part of the system would transfer data to the processor on a character-by-character basis.

At the other extreme, the external system would perhaps be a specialized programmable processor, placing all messages in the computer's primary memory. Alternatively, the system might be given buffers of messages to input and output. It might perform character-by-character conversion for different line speeds, line protocols and character sets. For example, a carriage return character usually signifies that the processor is to be informed when handling typewriter-like devices.



**exit**

**Fig. TRAN-7. RTM diagram of the receive part of a communications transducer.**

238

[previous](#) | [contents](#) | [next](#)

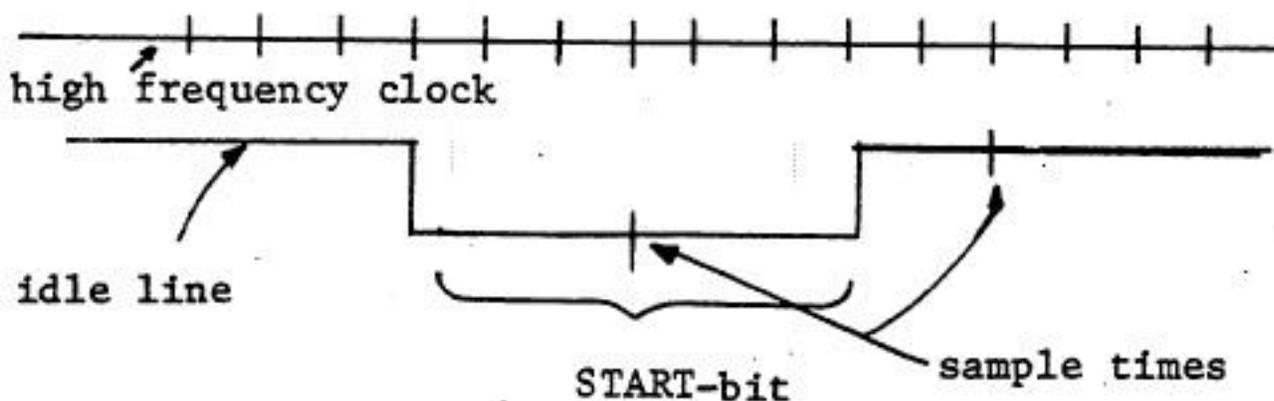


Fig. TRAN-6. Timing diagram showing the sampling time for detecting (receiving) the START-bit.

is illustrated by the waveform in Figure TRAN-6. The first 0 is detected, indicating the presence of the START-bit; at this time a bit counter is set to count to three, and a delay of  $3/7$  bit-times is measured. At the end of the delay, the center of the bit is marked (on the average). Now, subsequent sampling times are derived by counting delays of  $7/7$  bit-times. (Some counting schemes use a frequency of an even number for detecting the START-bit, e.g., eight. Why is this less desirable?)

Figure TRAN-7 shows a subroutine for receiving an 8-bit character with two STOP-bits. That is, it would receive data of the type sent by the transmitter of Figures TRAN-2,-3,-4 or by T(serial interface) modules (provided the clock frequencies agree).

The receiving process has four major parts: detecting the START-bit; verifying that the START-bit is present at the center of the bit sample time; sampling the eight data bits; and moving past one STOP-bit before ending the subroutine. If a 0 is not present at the center of the START bit, the subroutine is reinitiated. The eight data-bits are counted by placing a 1 in the R shift register initially, and waiting until it is shifted to the overflow-bit. Two delay subroutines measure delays of  $3/7$  and  $7/7$  of the bit-time. The final delay of  $7/7$  bit-times after the 8th bit insures that when reentering the subroutine the last data bit won't be detected as a start bit.

## PROBLEMS AND APPLICATIONS

1. Explore the design space of Vs for synchronous communications. Use the control characters from the ASCII character set to "frame" (i.e., synchronize the link).
2. For communications message switching applications computers are usually required because there are switching, buffering, and character conversion tasks to be done. These applications usually involve multiple lines (e.g., 20 100). If a hardwired device is interfaced to a computer for each line, the hardware cost (and perhaps program execution time) can be quite large. To reduce the hardware cost, a single,

physical T can be time-shared to provide the interface to a number of physically independent lines. Explore the design space for a set of systems which connect to a computer to provide the. interface to {asynchronous| synchronous} communications links. Give one design in detail.

236

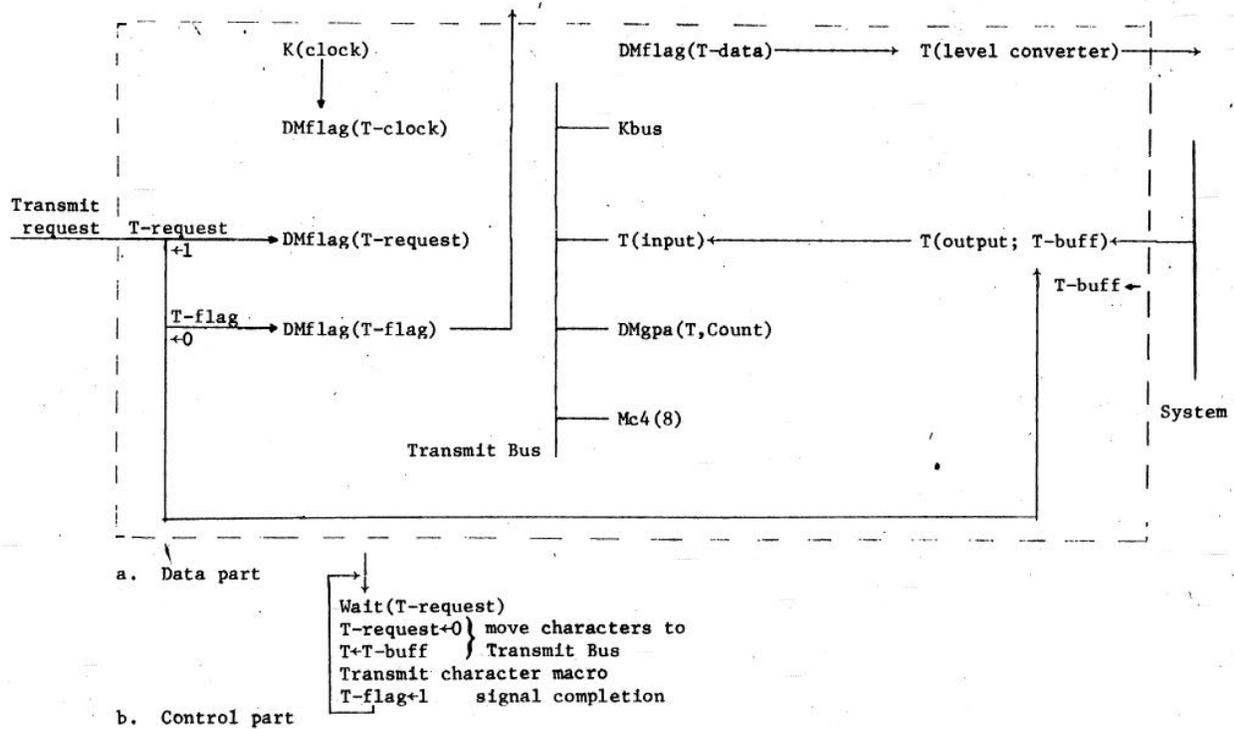


Fig. TRAN-5. RTM diagram of the transmit part of a communications transducer.

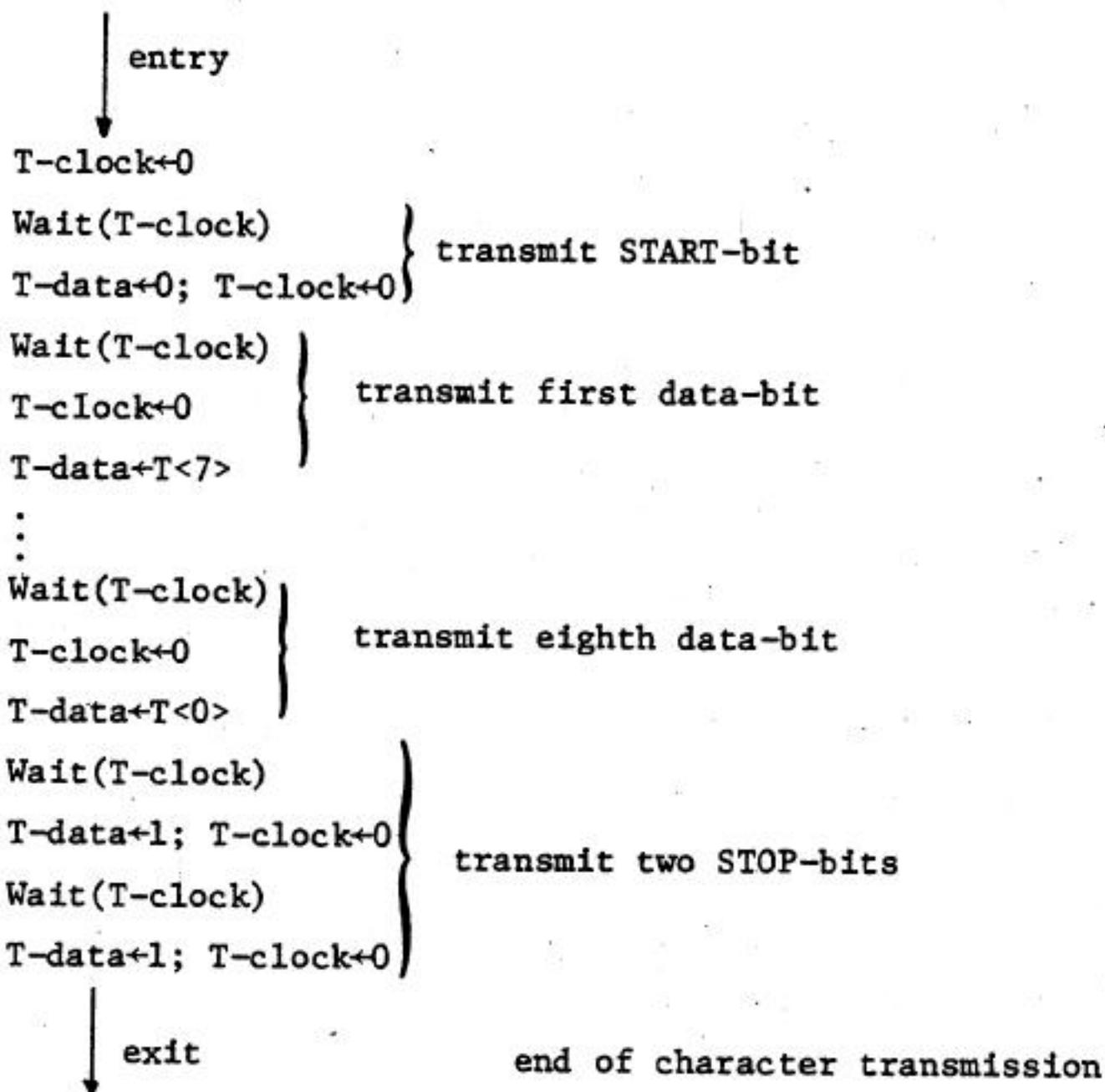


Fig. TRAN-4. RTM diagram of the control part of a subprocess to transmit one, eight-bit character in a synchronous, eleven-bit code format using straight-line control.

the second Bus and the communications between Busses, thereby providing a cheaper solution. The process (Figure TRAN-5) provides another example of the synchronization problem; the output process takes data from T-buff, moves it via the transmitter Bus (on left) into T for output. The DMflag (T-request) is the synchronizing variable, and indicates that a character in T-buff is to be output.

The receiving process is usually not as simple as transmission because there is a fundamental problem of determining when a bit arrives. That is, a time base (clock) has to be derived from which subsequent bit-times are marked. The scheme used here, which is the one used in most digital systems, is to use a high frequency clock (some integer multiple of the transmitter clock) to sample the line, detecting the first occurrence of the START-bit. The frequency of the clock used in this design is seven times the transmitter clock frequency. The method

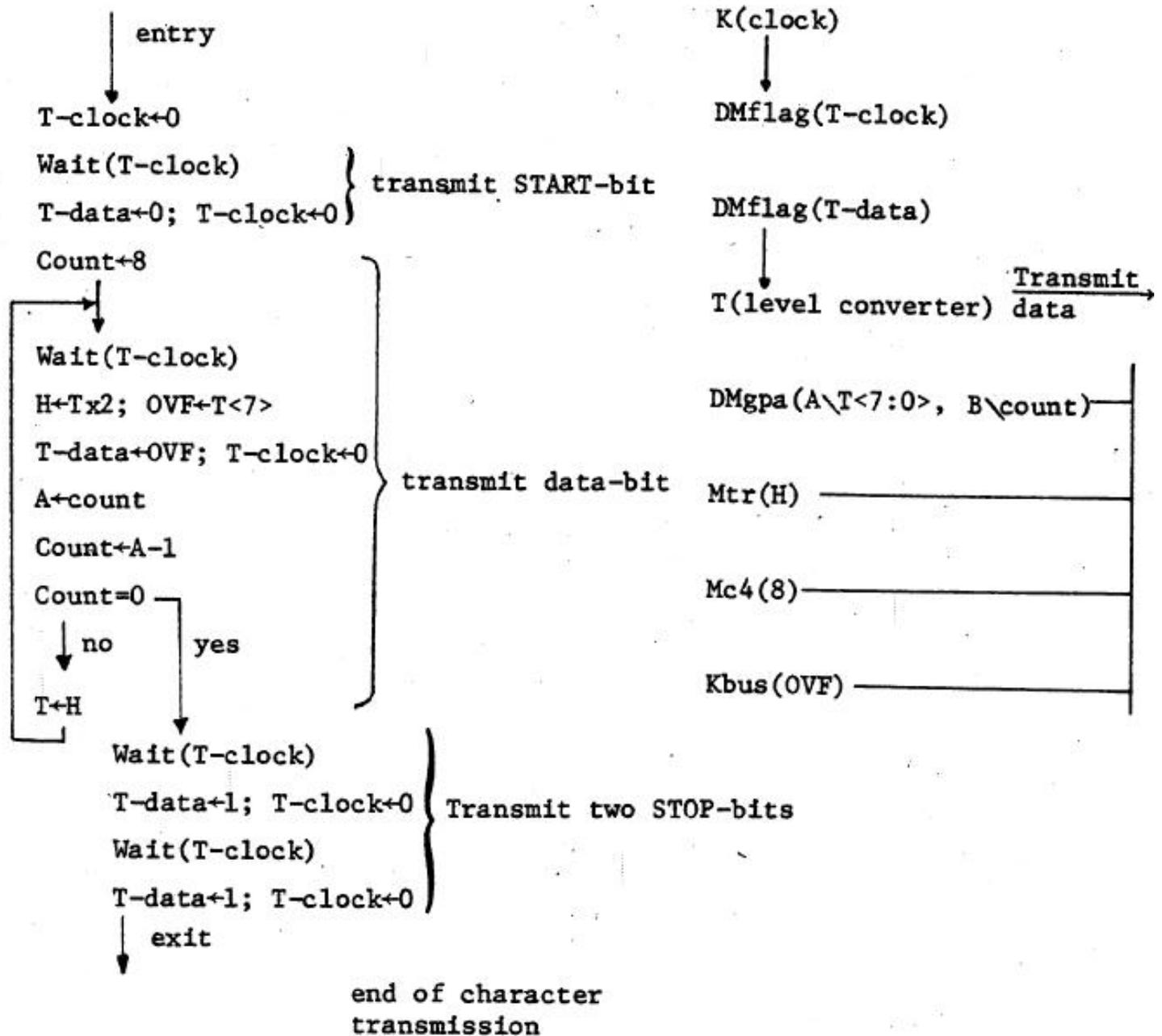


Fig. TRAN-3. RTM diagram of a subprocess to transmit one eight-bit character in a synchronous, eleven-bit code format using a loop control.

A system of this type could be used to transmit to a Teletypewriter, for example (in which case the clock interval would be 1/110 sec. or about 9.9 msec.). Note that while the above scheme is presented with a goal of pedagogical clarity it would undoubtedly not be a real design in RTM's because there is a more cost effective solution. This solution merely examines each bit in turn and transmits it without needing a DMgpa. A flowchart for this scheme is given in Figure TRAN-4.

Figure TRAN-5 takes the scheme given in Figure TRAN-3 and interconnects it to a second bus via a T(gpi) to provide an overall system which behaves as the T(serial interface) module. Using the scheme of Figure TRAN-4 would eliminate

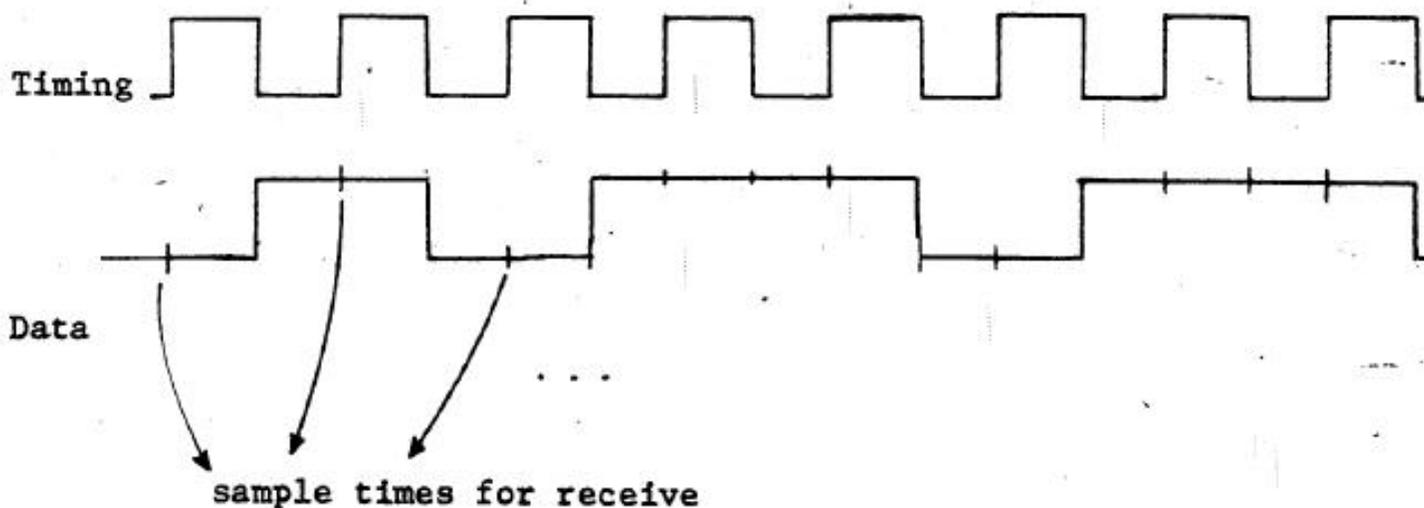


Fig. TRAN-2. Timing diagram showing a synchronous transmission technique.

Disadvantages of the technique are:

- a. Characters must be sent synchronously, not asynchronously, as they become available (which is desirable for some real time and mechanical applications) although start/stop synchronous transmission is possible.
- b. One bit-time added to or missing from the data-bit stream can cause the entire message to be faulty.
- c. The common-carrier equipment to accommodate this mode of operation is more expensive than the equipment required for asynchronous modes of operation.
- d. Mechanical equipment cannot readily transmit or receive this format directly without electronics.

## COMMUNICATION LINE INTERFACE TO RTM'S

The T(serial interface) module in the basic module set, described in Chapter 2, behaves as described in the previous section on asynchronous communications. Here the process for transmitting characters, i.e., the output part of the full duplex transducer, will be described. This system, is similar to the clock and waveform generation type of system, because the problem is to transmit the character data bits in synchronism with the transmit clock. The clock of the synchronous system, in effect, marks the bit-time boundaries, whereas the asynchronous system is self-clocking.

Before examining a system which behaves as the T(serial interface) module, a simple system for asynchronous data transmission will be presented. One such system is given in Figure TRAN-3. It is a subroutine which, when called, outputs a character from a register, T. The subroutine is purely sequential; when called, it waits for a clock tick ( $T\text{-clock} = 1$ ), then transmits the START-bit. Next, the eight data-bits of T are transmitted, followed by the two STOP-bits. The first operation resets T-clock so that bit output is in synchronism with the clock transitions.

as having two ports: one for transmitting and one for receiving. The ports can be thought of as consisting of a single wire, or at most two wires. These simplifications are possible and reasonably accurate.

There are two signalling methods for data transmission among communications links: asynchronous and synchronous. The application (type of data transmitted and data rate) and history determine which of the methods are used. The following description of these two signalling methods is based on Murphy and Kallis (1968).

In asynchronous serial data transmission, characters (encoded into streams of bits) are transmitted one at a time, whenever the line is not busy. A continuous stream of characters can be sent, end to end, but this is not necessary. The presence of each character is indicated by special start/stop codes, which appear on the line. Asynchronous transmission has the following advantages:

- a. Data is easily generated and detected by electromechanical equipment (e.g. Teletype keyboards and printers).
- b. Characters can be sent at an "asynchronous" rate (i.e. at will, as long as the line is not busy), because each character carries its own synchronizing information.

The disadvantages of this technique are:

- a. It is distortion sensitive. The receiver depends upon incoming signal sequences becoming synchronized. Any distortion in these sequences will affect the reliability with which the character is assembled; hence characters are usually limited to eight bits.
- b. It is speed limited. To accommodate distortion, transmission speeds can only go as high as about 2000 baud (bits/sec).
- c. It is inefficient. At least 10 bit-times are required to send 8 data bits. If a 2-bit start/stop code is used, 11 bit times are required to transmit 8 data bits.

In the synchronous serial technique, a continuous stream of characters is sent over the line, and there is usually no need for special start/stop codes for each character. Instead, characters are separated by transmitting a unique code at the beginning of a character stream which, when recognized, causes the receiver to lock in (frame) the incoming bits and assemble them as characters. As in the asynchronous technique, the character length (in bits) is fixed. Other full character codes signify other conditions concerning the data stream.

Unlike the asynchronous technique, a synchronizing signal must be provided along with the data bit stream. This signal is in the form of a clock which signifies when each data bit can be transmitted or has

been received; bits are transmitted only at clock times, and at each clock time, either a 0 or a 1 must be transmitted. This clock signal can either be provided by the transmitter, or by some separate source that the transmitter uses for timing. The format for this type of transmission is shown in figure TRAN-2.

In the format shown, the transmitter presents data to the line on the negative going (+ to -) transition of the timing signal, and the receiver samples the data line on the timing signal positive transition (- to +).

The advantages of the synchronous serial techniques are:

- a. A common timing source can be used for both transmitter and receiver, hence clock-synchronizing logic is minimal.
- b. Efficiency is increased, since there are no bit-times wasted with the use of start/stop code bits. All bits on the line are data, with the exception of a single synchronizing pattern at the beginning of the bit stream.
- c. There is low distortion sensitivity, due to the timing being provided with the data, allowing higher speeds.

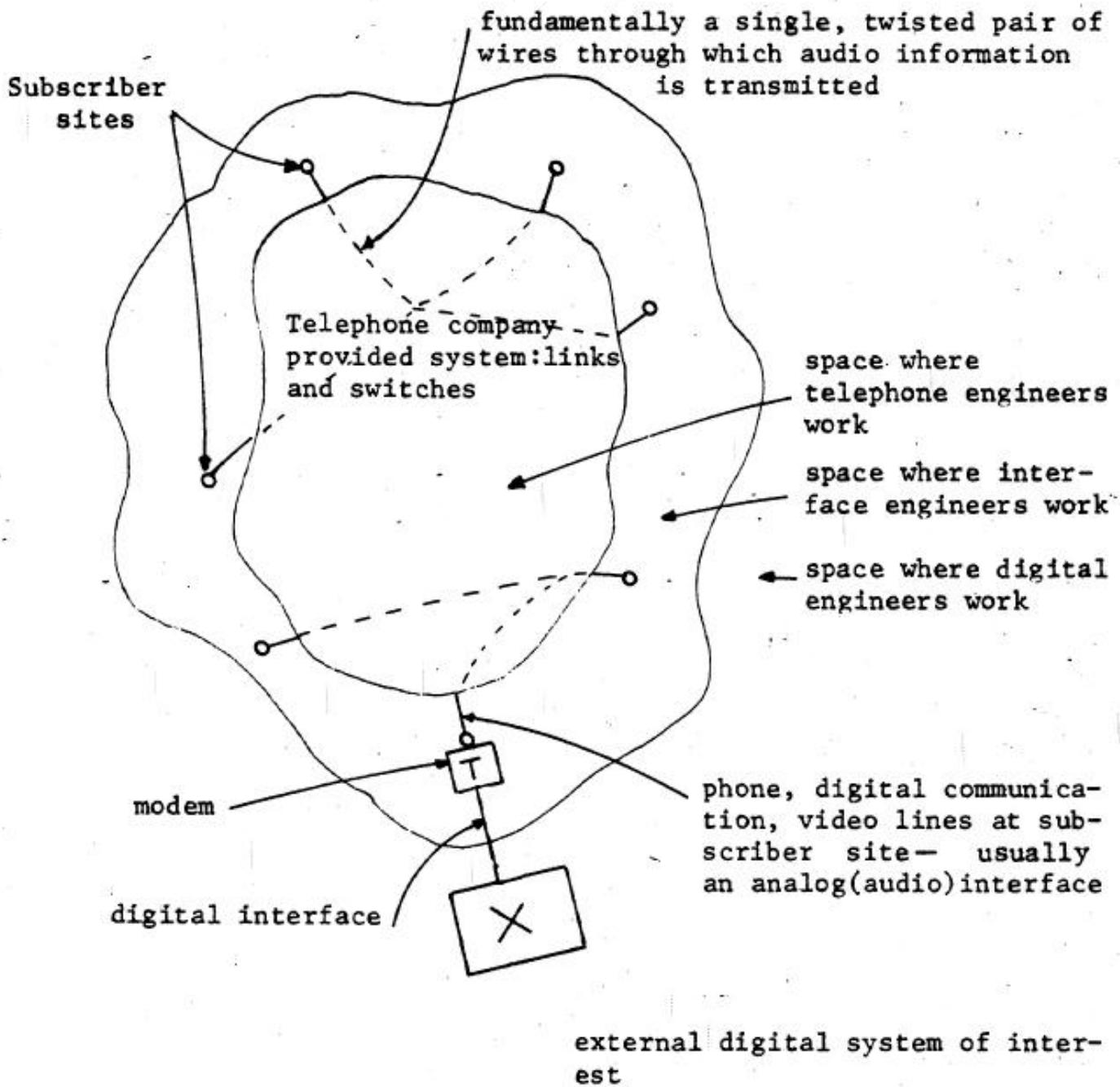


Fig. TRAN-1. The communications space environment.

criterion. For most designers, however, the digital interface begins at the digital data transmission side of the modem. There are still several concerns at the modem interface:

1. Data is transmitted on a one-bit at a time (serial) basis.

2. We can have either the equivalent of two simultaneous links -- full duplex, or a link which is either receiving or transmitting, but whose direction can be reversed -- half duplex.
3. The modems and telephone links add noise, hence the data received is not exactly the data transmitted.
4. There is a large number of different types of modems and corresponding interfaces.

Ignoring the error problem and various modem interface signals (e.g., those that indicate whether the unit is turned on or off), the modem can be considered

subprocesses check the flags and issue an error return if the operation cannot be successfully completed. Note that the complete signal is given when the error signal is set (why?). Can this spurious complete signal be eliminated?

2. The K(arbiter) presented above used a polling loop to check the request flags. In some applications, however, such a loop might not be fast enough to allow efficient use of the shared facility. Design a combinational circuit to replace the polling loop in the K(arbiter). The rest of the structure can remain unchanged.

## TRANSDUCERS FOR DATA COMMUNICATIONS SYSTEMS

**KEYWORDS:** Synchronous, asynchronous, communications, timing, receive, transmit.

This section will examine a class of transducers which are used to transfer data between physically separate sites via communications links, usually conventional telephone lines. They are presented because nearly all digital systems engineers eventually are involved with digital data communications and this section will serve as a brief introduction. It is also hoped that when users understand how really simple data communications are, these concepts, formats, and interfaces will be used instead of designing new interfaces which are incompatible with everything else.

James Martin has written a series of books (e.g., 1969) on digital communications. Knowledge in this area is based on classical communications theory together with a large variety of equipment. The digital communications professional should know: classical communications theory, various telephone company line tariffs and policy, equipment that is available and which can be legally interfaced to lines provided by the telephone communications companies, what types of equipment can use the facilities, etc.

An overview of the system under study is given in Figure TRAN-1. The telephone companies (i.e., communications carriers) fundamentally provide communications lines (links) between pairs and larger groups of subscribers, allowing them to communicate audio, picture phone, video, and digital data information. They also provide facsimile devices and Teletypes for the transmission of printed information. Finally, they allow subscribers to select connections to other subscribers (i.e., switching). To be considered along with the telephone companies are federal, state and local regulating commissions, etc. This large, internal mass, Figure TRAN-1, is considered as given -- with no clear input or output.

As one moves away from the central hard-core of communications suppliers that fundamentally provide a switched, twisted pair of wires between one site and another, the next system obstacle is encountered. Since the twisted pair of wires usually transmits data in the audio frequency range of 100-3000 Hz, there has to be some transducer to convert information in that frequency domain into the 0 and 1 needed by the digital engineer. The transducer which converts this audio frequency data into digital data bits (0 and 1) is called a modem. The conversion encoding is implemented by using one of the following': amplitude

modulation (AM) with on-off keying; frequency modulation (FM), using frequency shift keying; and phase modulation (PM). For the vast number of people who profess to be communications engineers and who do not work for the telephone company or the government (which regulates the telephone companies) the modem is simply another given component. There are exceptions, since a ruling of one of the regulatory agencies does allow a user to provide his own modem in certain cases; user provided modems are common when low cost is a design

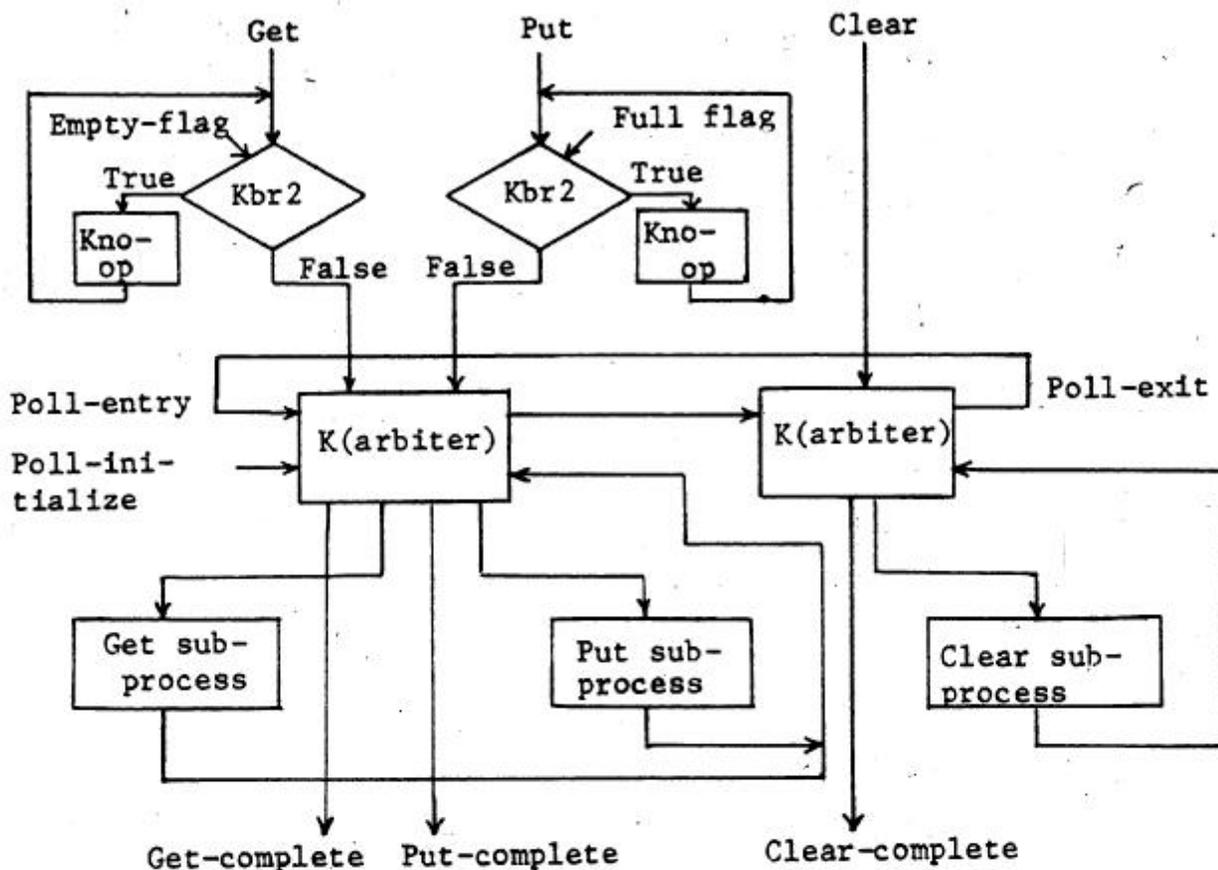
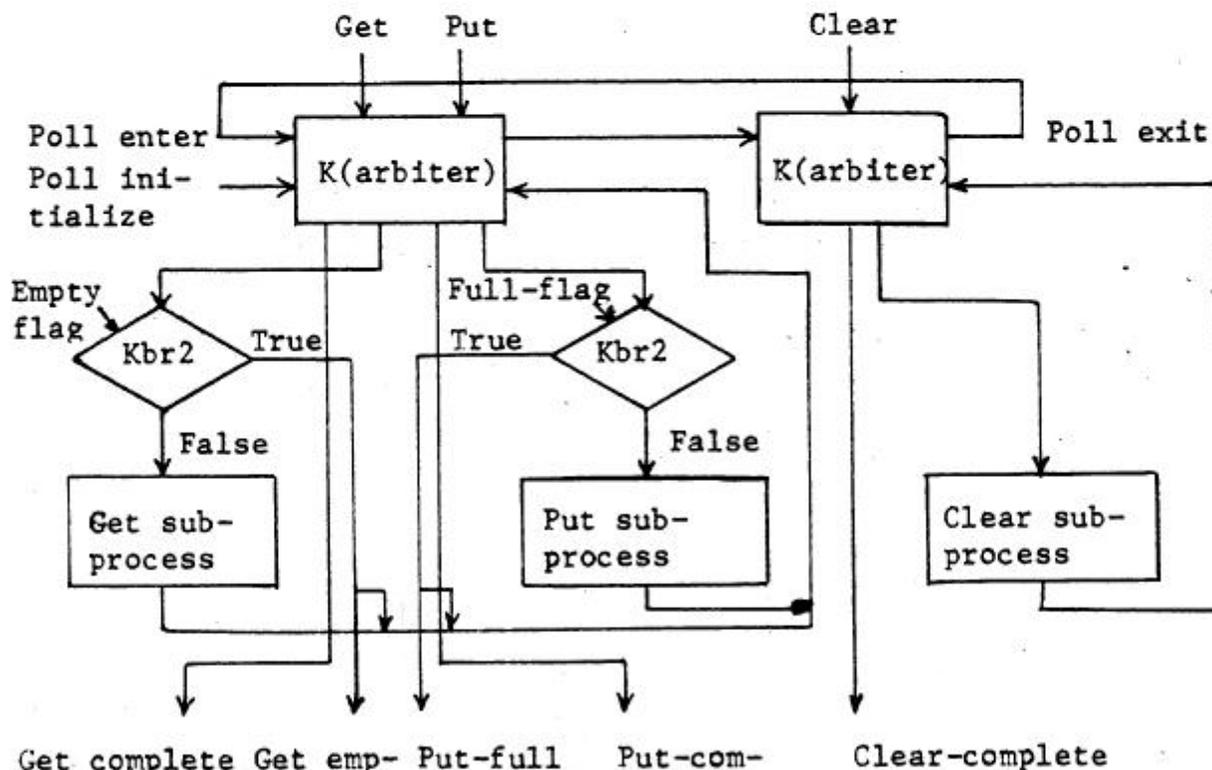


Fig. ARB-7b. RTM diagram of an M(queue), using a K(arbiter), that holds Get and put requests until they can be successfully completed.



Get complete    Get emp-    Put-full    Put-com-    Clear-complete  
                  ty er-    error    plete  
                  ror

Fig. ARB-7c. RTM diagram of an M(queue), using a K(arbiter, that issues error signals when a Get or Put request cannot be successfully completed.

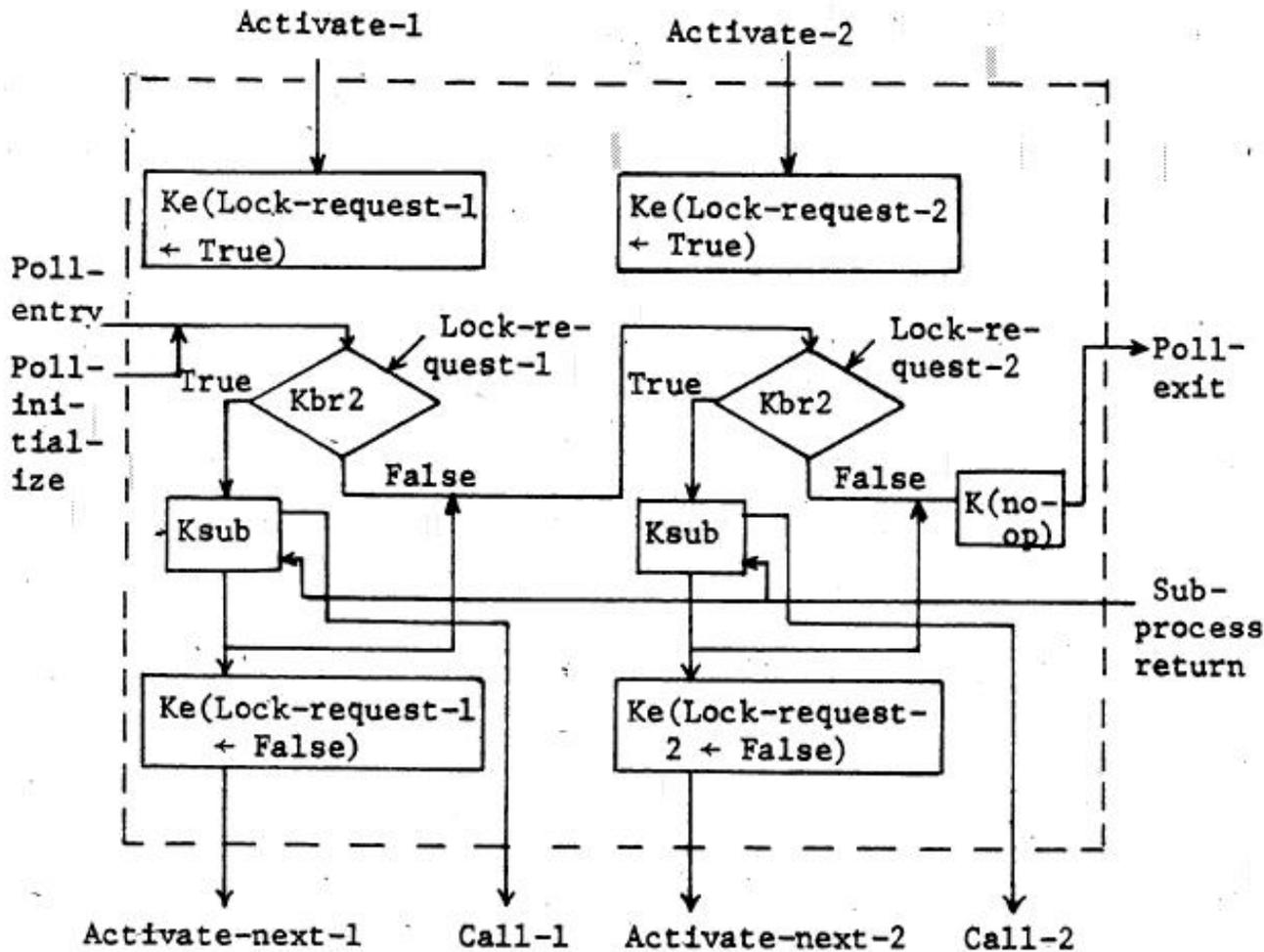
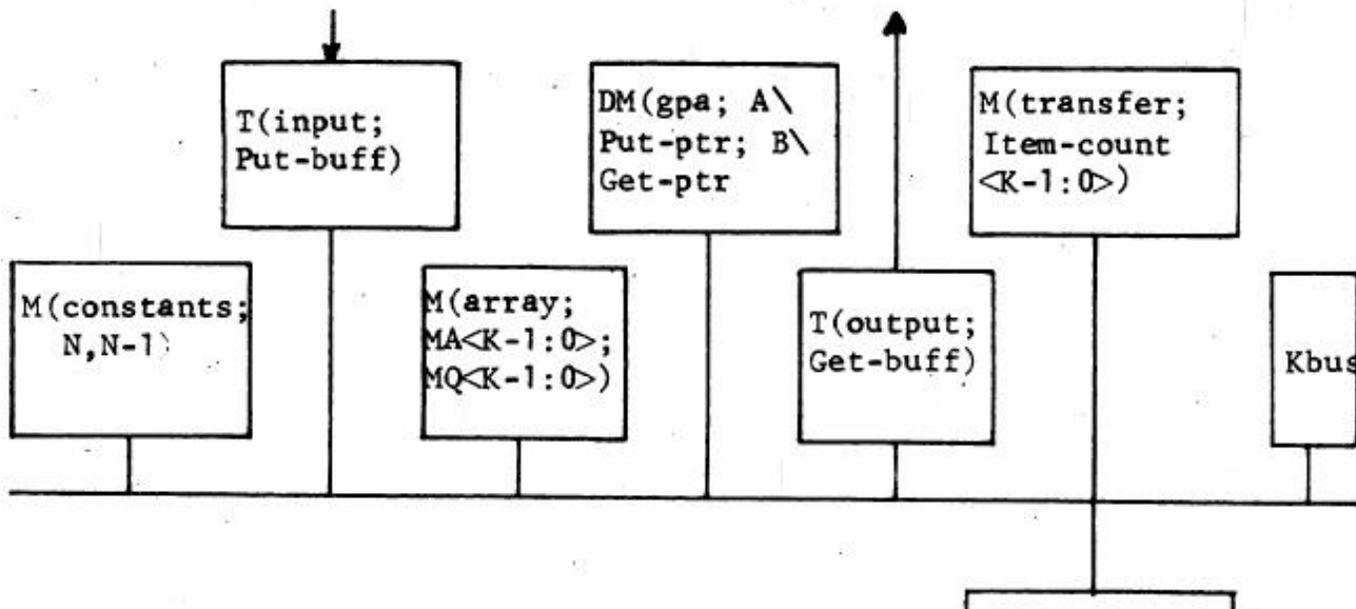


Fig. ARB-6. Module diagram showing details of the K(arbiter).



DM(flags; Full-flag, Empty-flag)

M(transfer;  
T<K-1:0>)

Fig. ARB-7a. RTM diagram of data part for an M(queue) using a K(arbiter).

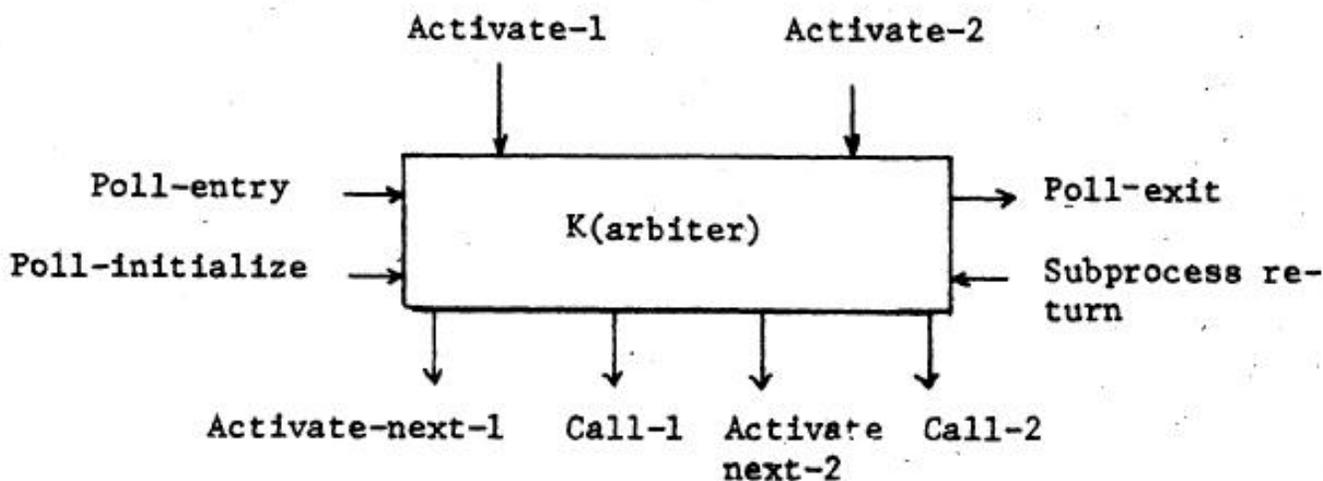


Fig. ARB-4. Module diagram showing the general structure of the K(arbiter).

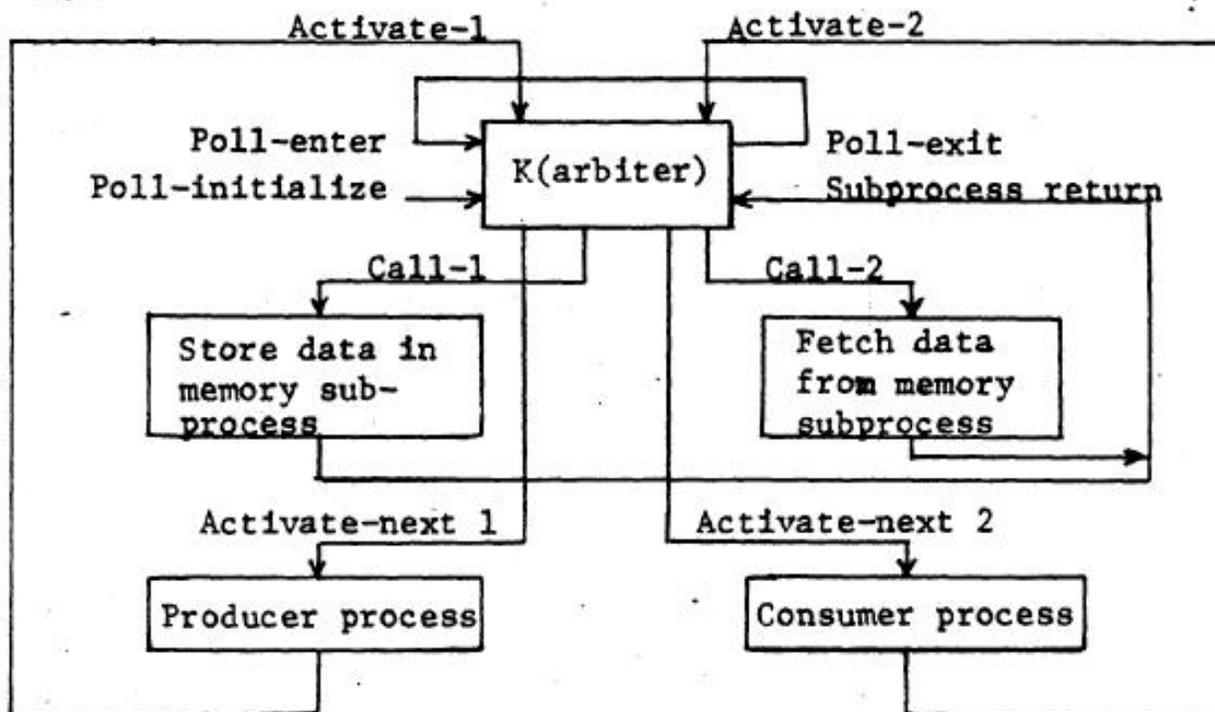


Fig. ARB-5. RTM diagram of a Producer-Consumer system using a K(arbiter) for synchronization.

Diagram of a Producer-Consumer system using a semaphore for synchronization.

227

[previous](#) | [contents](#) | [next](#)

request for a fetch or store is detected in the central polling loop, the central process gives the requesting process access to the common memory. The producer and consumer processes enter a wait-loop which allows control to proceed when the request flag is set to false by the memory access facility as it completes its use of the memory. Since the polling loop in the central process inspects the flags sequentially, there is no way that both the producer and consumer can gain access to the common memory at the same time. This solution will be used to solve the synchronization problem in the M(queue).

## PROBLEM STATEMENT

Design a module, a K(arbiter), which performs the function of examining a number of possibly simultaneous calls for use of a shared resource facility. The module should grant a calling accessor the next access to the facility, thereby locking all other calls out, and then unlock the facility, allowing access by other processes after the current user of the shared facility completes its use of the facility.

## SOLUTION

Figure ARB-4 shows the general structure of a K(arbiter) module for two inputs. When an activate is acknowledged by the K(arbiter), the appropriate subprocess which uses the common facility is called and entered: On exit from the subprocess, the return signal is given to the K(arbiter) which unlocks the facility and sends control to the appropriate activate-next part. Multiple K(arbiter) modules of this type could be interconnected, providing more than two inputs. Figure ARB-5 shows how the K(arbiter) is used to solve the producer- consumer synchronization problem.

Figure ARB-6 shows an RTM implementation of a K(arbiter) based on an equal priority polling scheme. The poll-entry input and poll-exit output are used to connect multiple K(arbiters) together; for a single module, they are connected to each other; the poll-initialize input starts the polling process. The control flow from an activate input to the activate-next output is broken, since the activate inputs only set a request flag. The request flags are detected by the polling process which passes control to a Ksub to call the particular subprocess to use the shared facility. On completion, the subprocess issues a return signal which restarts the polling process and issues the appropriate activate-next signal. Alternatively, both calling processes could call the same subprocess which uses the common facility. Examples of this type of usage are in arithmetic function evaluation.

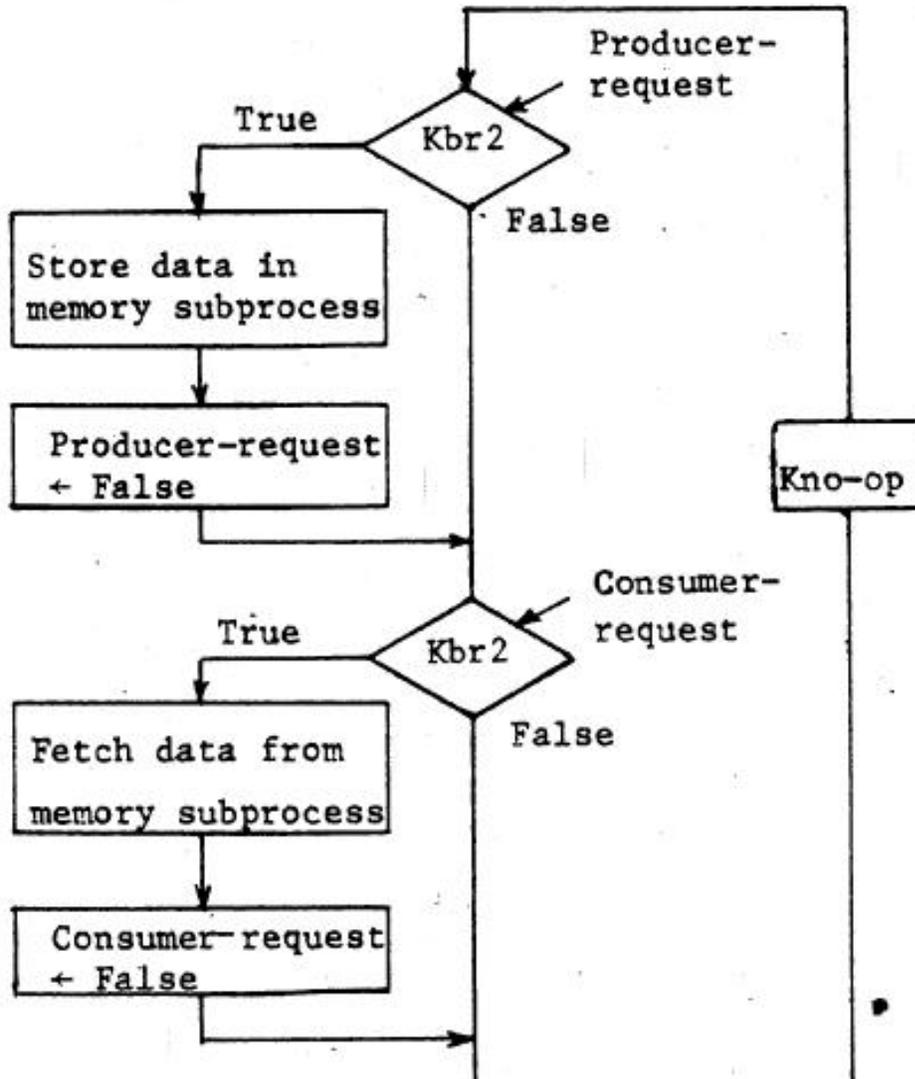
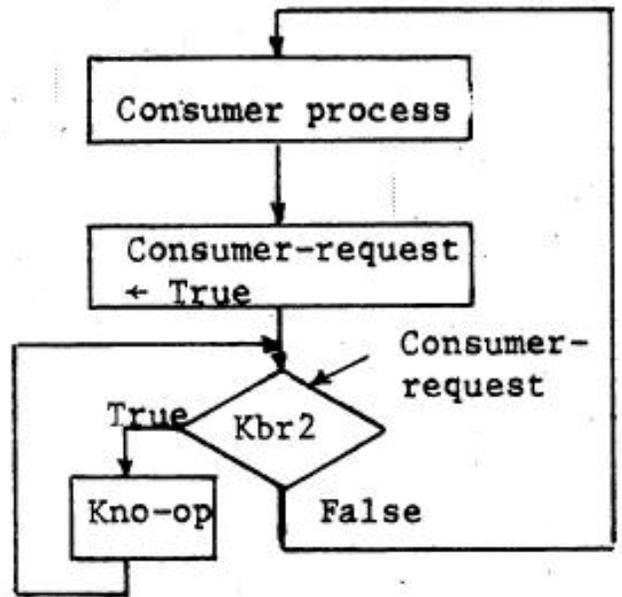
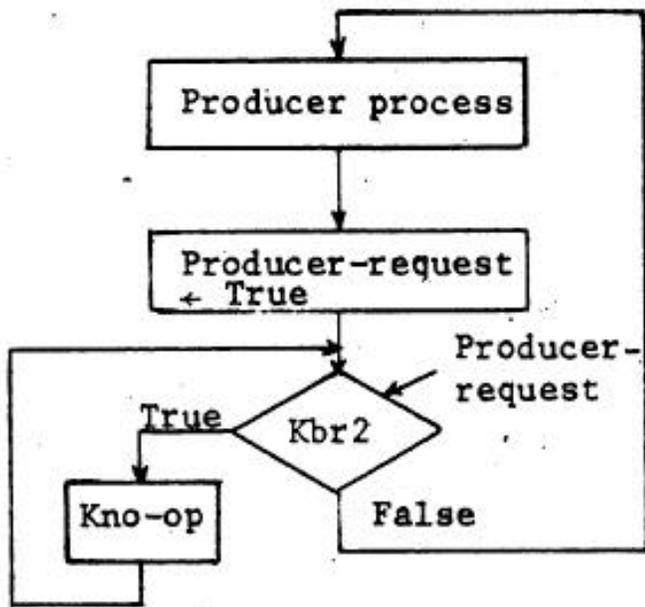
## EXTENSIONS AND ADDITIONAL PROBLEMS

An M(queue) based on the K(arbiter) is shown in Figure ARB-7. The data part of the design is given in Figure ARB-7a; this structure is essentially that of the M(queue) problem except that no additional request-flags are needed since the K(arbiter) has internal request flags. Figure ARB-7b shows the control structure of the design; the three subprocesses, Get, Put and Clear, are as defined in the M(queue)

problem except that the subprocesses do not have to reset the request flag on completion. A Get call is not issued to the K(arbiter) until the Empty-flag is false, likewise a Put call is not made until the Full-flag is false.

## Problems

1. Figure ARB-7c shows a control structure for a design which does not hold Put and Get calls until they can be successfully completed. Rather, the Get and Put



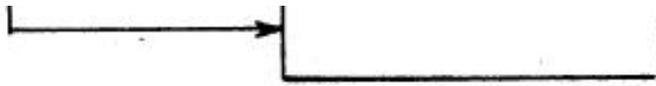


Fig. ARB-3. RTM diagram of a Producer-Consumer system using a "busy waiting" polling loop for synchronization.

time find the common memory free, hence both would proceed to access the facility causing erroneous operation.

This synchronization problem does have a solution which involves polling (commonly called "busy-waiting" as described in the EPUT problem). In this design, shown in Figure ARB-3, the central process polls requests (indicated by the flags) from the producer and consumer for use of the common memory. As a

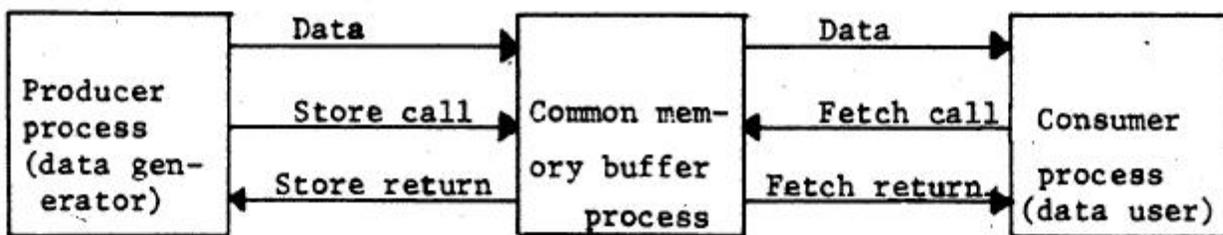


Fig. ARB-1. Producer-Consumer system with a shared memory facility.

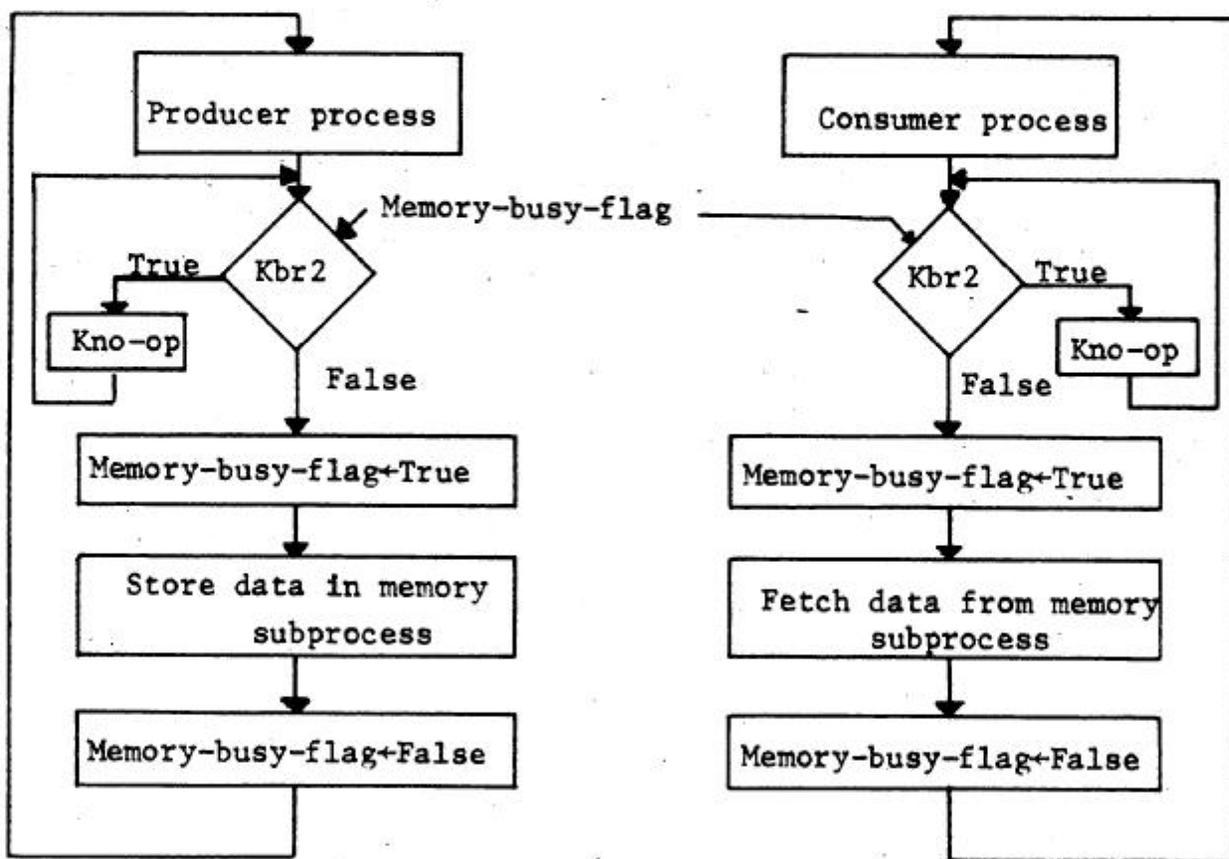


Fig. ARB-2. RTM diagram of a Producer-Consumer system using a memory-busy-flag.

[previous](#) | [contents](#) | [next](#)

## **K(ARBITER) - AN ERTM FOR SYNCHRONIZATION OF MULTIPLE PROCESSES**

**KEYWORDS:** Arbiter, producer, consumer, polling, synchronization, parallelism.

The purpose of the K(arbiter) module is to arbitrate multiple requests for use of a common set of facilities. A polling structure which examines such requests in turn, e.g. as described in the M(queue) problem (later in this chapter), solves the same type problem as the K(arbiter), but the K(arbiter) saves hardware and makes the solution of timing and synchronization problems quite apparent. These synchronization problems arise from the need (desire) to achieve a high degree of parallelism in digital systems.

The K(arbiter) module is similar to a parallel merge in that it waits for a specific condition to occur before proceeding, but unlike the parallel merge, its use is not predicated on lock-step parallelism. The form of parallelism it addresses is referred to as concurrency; in essence two or more processes are active concurrently, and, at some time, must share a common set of facilities (resources). In the M(queue) problem, the three control subprocesses each access a common data part, hence only one control part is allowed to be active at a given time. On the other hand, in the histogram recorder problem, a record updating process and display process could run concurrently except when, both required access to the shared memory.

The basic structure of this latter type of system is shown in Figure ARB-1. There is no coordination required by the two processes except that they cannot access the common memory (essentially a queue) at the same time. The producer process generates data and makes calls to store the data in the common memory while the consumer process makes calls to fetch data, to use in the process, from the memory. The producer and consumer processes call the common memory buffer process at different rates and neither is synchronized with it or with each other. A common process controls access to the central memory in such a way that only one process (producer or consumer) can use it at a time, thereby protecting the two processes from one another. Without such protection, the processes might incorrectly change a controlling variable in the central process, thereby causing erroneous action by either the producer or consumer.

Suppose that there is a flag that indicates whether the common facility is in use or not. Each process can examine the flag, and if the memory is free, then that process can set the flag to indicate that the memory is in use at that time, thereby inhibiting the other process from gaining access to it. When the process has finished using the memory, it can reset the flag to indicate that the facility is not in use, thereby allowing the waiting process to access the memory. An implementation based on such a flag is shown in Figure ARB-2. There is a slight' flaw in the design, however, that could make the system operate incorrectly. When examining the flag, both the producer and consumer might at the same

[previous](#) | [contents](#) | [next](#)

## USE OF THE COATING THICKNESS MONITOR IN CONJUNCTION WITH A STORED PROGRAM COMPUTER

Advantages from building an RTM based system usually include lower cost than a stored program computer for a fixed small task, and faster operation than a computer because of specialization to the task. The CTM is a good example of how a system can be hardwired to give very high performance. Another benefit of hardwiring is involved in interfacing, because a computer usually has to be specialized to meet interface constraints. On the other hand the RTM system once hardwired, is relatively inflexible for month to month changes, and large algorithms are not particularly suited to the structure because of limited control and processing structures. The analysis algorithm implemented in the CTM is getting to the complexity limit that one might wish to hardwire.

By using a computer with an RTM system one might combine the advantages of both high performance processing, and flexibility in the analysis and control algorithms. The time-consuming functions in this problem are the reading of the beta gages and storing them in memory, and the averaging and distribution analysis. Accepting switch inputs, converting them from decimal, and displaying results happen infrequently compared to the constant stock movement.

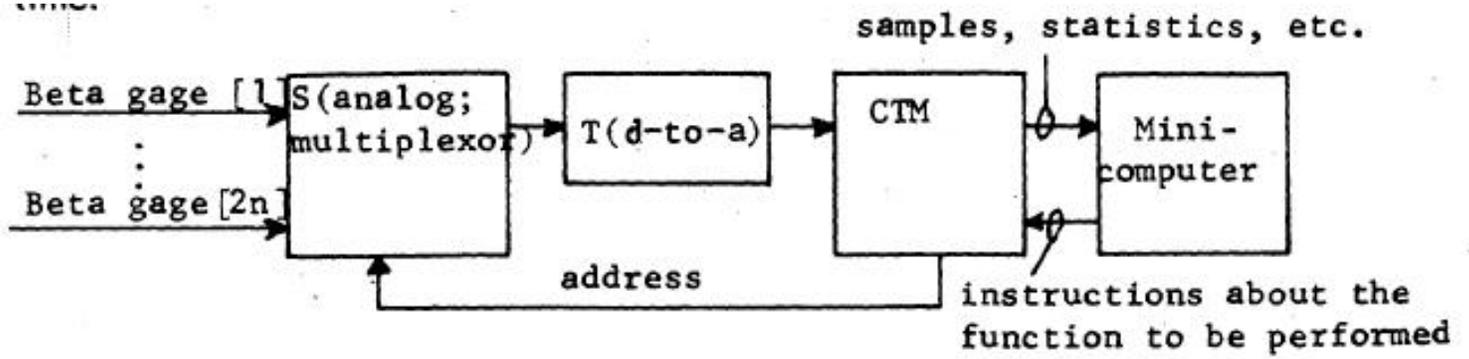
### PROBLEM STATEMENT

Assume an RTM system is attached to a small computer to provide the CTM functions, and design a system to provide the functions of the original monitor. You may use any of the small computers described in Chapter 6. The input information about the material,  $K_j$ , the number of samples to be taken in the computation of the averages, the allocation of memory for results, and the various analysis functions would all be specified as parameters to the RTM system. Also, the RTM system might check for out of tolerance limits. Note, the RTM system would use the small computer's memory on a shared basis to store results.

### DESIGN CONSIDERATIONS

It is important in this design not to give up any of the computer's flexibility. Doing too much automatically will often lead to a system that cannot be changed in the future, hence a large amount of automation is undesirable. Another problem which will arise is that constant use of the computer's memory by the hardwired portion of the system will cause its processor to stop, hence the design should allow some computation (say 50%) while the memory is being utilized. The gross structure of such a system is shown in Figure CTM-3.

There are several types of designs that might be examined. One would be similar to the alarm scanner (see Chapter 6) which performs one instruction at a



- Fig. CTM-3. PMS diagram giving structure of a CTM based on RTM's and a minicomputer.

readings by an appropriate constant. For the  $j$ th station the thickness of the material is, thickness =  $K_j$  ( $B_{2j} - B_{2j-1}$ ). Here, it may be assumed that the beta gage reading is known to only 1 part in 256 (8-bits) and the value of  $K_j$  is also only known to 8 bits. Prior to a run, these constants (the  $K$ 's) are read manually into the CTM via switches.

There are many different types of outputs that a monitor of this type could display. Some possible measurements on the samples are:

1. The instantaneous value of the coating at a particular station.
2. The minimum thickness -- measured since start-up.
3. The maximum thickness -- measured since start-up.
4. The average value of thickness of the last  $m$  samples, where  $1 \leq m \leq 256$ .
5. A distribution of the thickness (which can be output on paper tape after a certain time period). This distribution gives a measure of the consistency of the coating.

## PROBLEM STATEMENT

Design an  $n$  station CTM to display as many of the measurements listed above as possible.

## COATING THICKNESS CONTROL

After exploring the design space of CTM's in the previous section, it should be clear that a hardwired structure provides for a very fast computation of various parameters about a coating station's behavior. In addition, one might want to use the thickness information to aid in controlling the amount of coating material being applied at a given station. As the speed of the process increases, it might be necessary to do this automatically. Suppose that the coating material flow valves are also under control of the CTM. In PMS terminology the two functions, data operations (D) and control (K), are being performed, hence the system becomes a DK(coating thickness).

Assume a coating valve has a ramp shaped time characteristic to change from one opening to another, where the time constant of the ramp is quite long. A valve of this type might be controlled by a stepping motor which takes a fixed time ( $t_m$ ) for one step. At any moment of action, the valve can be left alone, opened one step, or closed one step. The two inputs to a given valve might be: Move one step, and a Boolean which indicates close (if 0) and open (if 1). Two Boolean outputs could indicate whether the valve is Fully Open or Fully Closed. A third output, an event, could be derived from the valve indicating

that the stepping process has completely taken place. Alternatively, a valve might have the same input control structure, but be continuous in nature. That is, it would take Boolean inputs to indicate movement and direction, but then would merely continue moving until a stop is reached. (The stop would prevent the valve from proceeding beyond its travel.) With a scheme of this type there would be no feedback about the actual position of the valve except via the measurements taken at the Beta gages.

## PROBLEM STATEMENT

Design a control system which would operate the valves. Since the valves are incompletely specified, either make some assumptions about their parameters, or leave the actual control algorithm free to be specified when more information is known. The data from the thickness monitor would be used to manage the control. Using this data together with the tolerance limits of the coating, assume the DK(coating thickness) would have parameters which would be used to specify the actual thickness to be maintained.

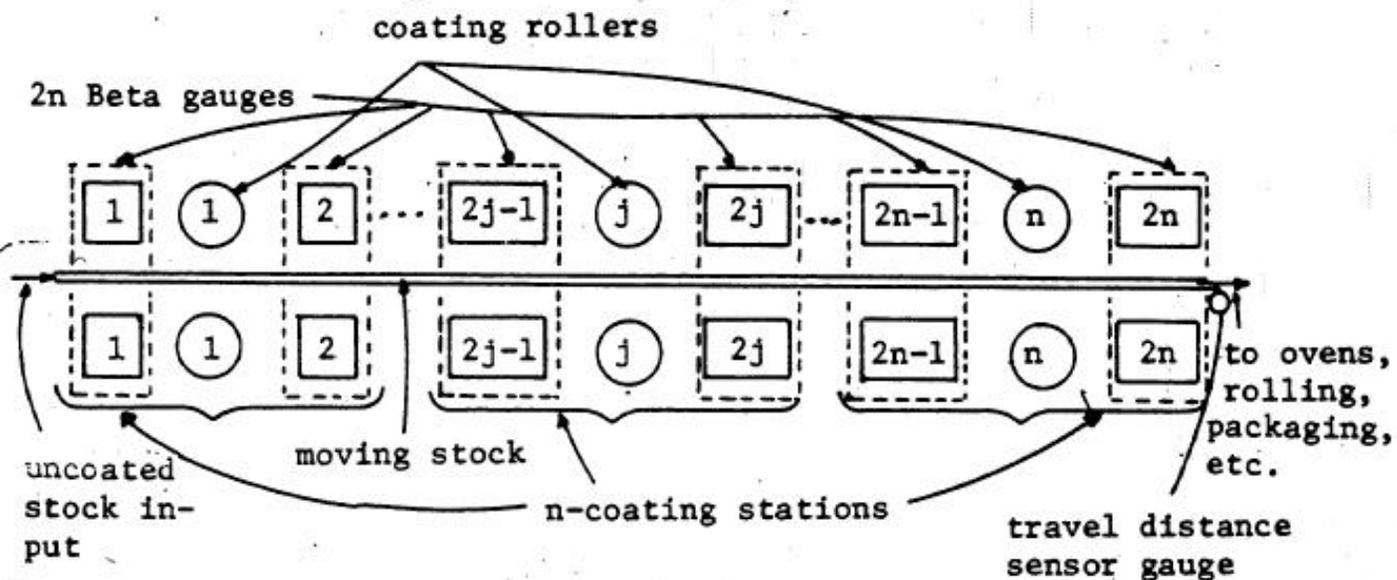
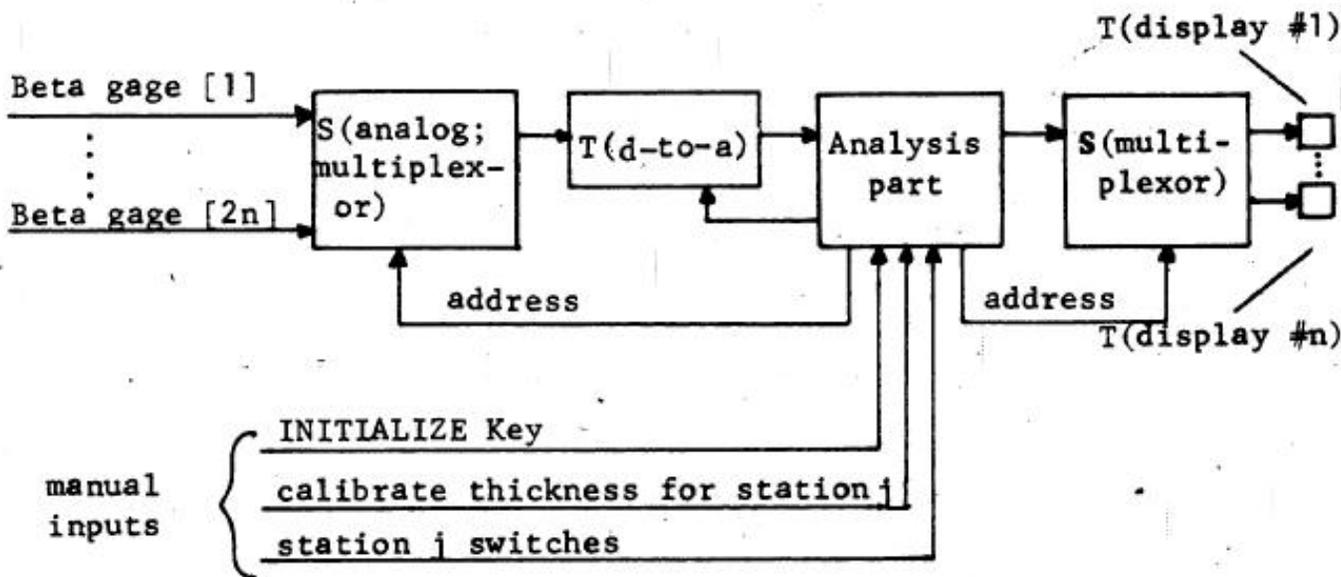


Fig. CTM-1. N-stage continuous coating process.



Parameters for each of the inputs to the CTM from the pairs of beta gages are operated on independently to produce an output for that pair of gages. The only coupling among the coating stations is the fact that the sheet moves through all coating stations at the same rate. That is, there is a velocity detector which produces a pulse every time a certain length of stock,  $l_s$  (in cm.) has passed. The velocity of the stock is  $v$

(in cm/sec). Therefore, if all beta gages are sampled each time one of these pulses arrives, the time between samples,  $t_s$ , is  $t_s = I_s/v$ . Since there are  $n$  coating stations, the amount of time for the output computation is  $t_c = t_s/n = I_s/n*v$ , assuming the output is updated each time a sample is taken.

The current coating thickness at a coating station,  $j$ , is computed by multiplying the difference between its initial,  $B_{2j-1}$ , and final,  $B_{2j}$  beta gage

- manual switch as a variant of the display as indicated in Solution 4. What is the maximum allowable event rate?
3. Define the action of  $K_{\text{sub}}(\text{Process-cell-overflow})$  for Figure Hist-6 and draw the flowchart of its behavior.
  4. Carry out the design indicated in Solution 1 to increase the update rate (i.e., decrease the time between events). This is accomplished by polling the update- request process within the  $K_{\text{macro}}(\text{Display})$ . What is the maximum allowable event rate?
  5. Design the control parts for the two-Bus scheme as indicated in Solution 3.
  6. Design a two-Bus histogram recorder using a  $K(\text{arbiter})$ . Minimize the time spent in the common control section. What is the maximum allowable event rate?
  7. Plot the cost and performance for the various solutions.
  8. Compute and display the integral of the distribution. Form the cumulative probability distribution:  $FF(X[J]) = \sum_{I=0}^J F(X[I])$  for all  $J = 0, 1, \dots, N$  such that  $FF(X[N]) = 1$ . This process would be initiated by pressing a  $K(\text{manual evoke})$ .
  9. Compute the mean of the distribution described in problem 8.
  10. Compute the median of the distribution described in problem 8.
  11. Using the  $T(\text{paper tape})$  described in the transducer section, punch the contents of the 16 bit switch register, followed by the  $2 \times 1024$  8-bit characters in the histogram memory. Punch a 16-bit sum check of all this preceding data;

## A MONITOR TO MEASURE THE THICKNESS OF A COATING PROCESS

**KEYWORDS:** Monitor, control, waveform analysis, multiplexing

This problem involves the design of a system for the analysis of multiple, but similar waveforms. The process from which the waveforms arise is a continuous, multiple station coating mill (see Figure CTM-1).(1) The coatings are on sheet stock such as paper, cloth, plastic, and tin which move through the mill. Since this is a continuous process it is difficult (too costly and impractical) to test the thickness of the

coating at completion. However, controlling the coating thickness is extremely important because material will be wasted if too much is applied, and similarly, the quality of the product may be unacceptable if too little is applied. We will discuss and propose the design problem for this task, but will not present detailed solutions.

A special purpose digital system, which we shall call a coating thickness monitor (CTM), can permit the nondestructive testing of the coating thickness. In this method, coating thicknesses are measured by beta gages that provide an electrical (analog voltage) output related to the mass of material in the gage measuring gap. Gages are placed before and after each coating station as shown in Figure CTM-1.

A PMS diagram of a possible general structure for the CTM is given in Figure CTM-2. The various beta gage voltages are selected via an input S(multiplexor). The results of analyzing the beta gage waveforms are displayed at the various T(display)'s via an output S(multiplexor). In general, for an n station coating mill, there would be 2n input beta gages and n T(display)'s. Thus the main emphasis of this design is to show how a single system can appear to be n, independent systems by time-sharing a single CTM.

-----  
1. This problem was derived from Jurgen (1970).

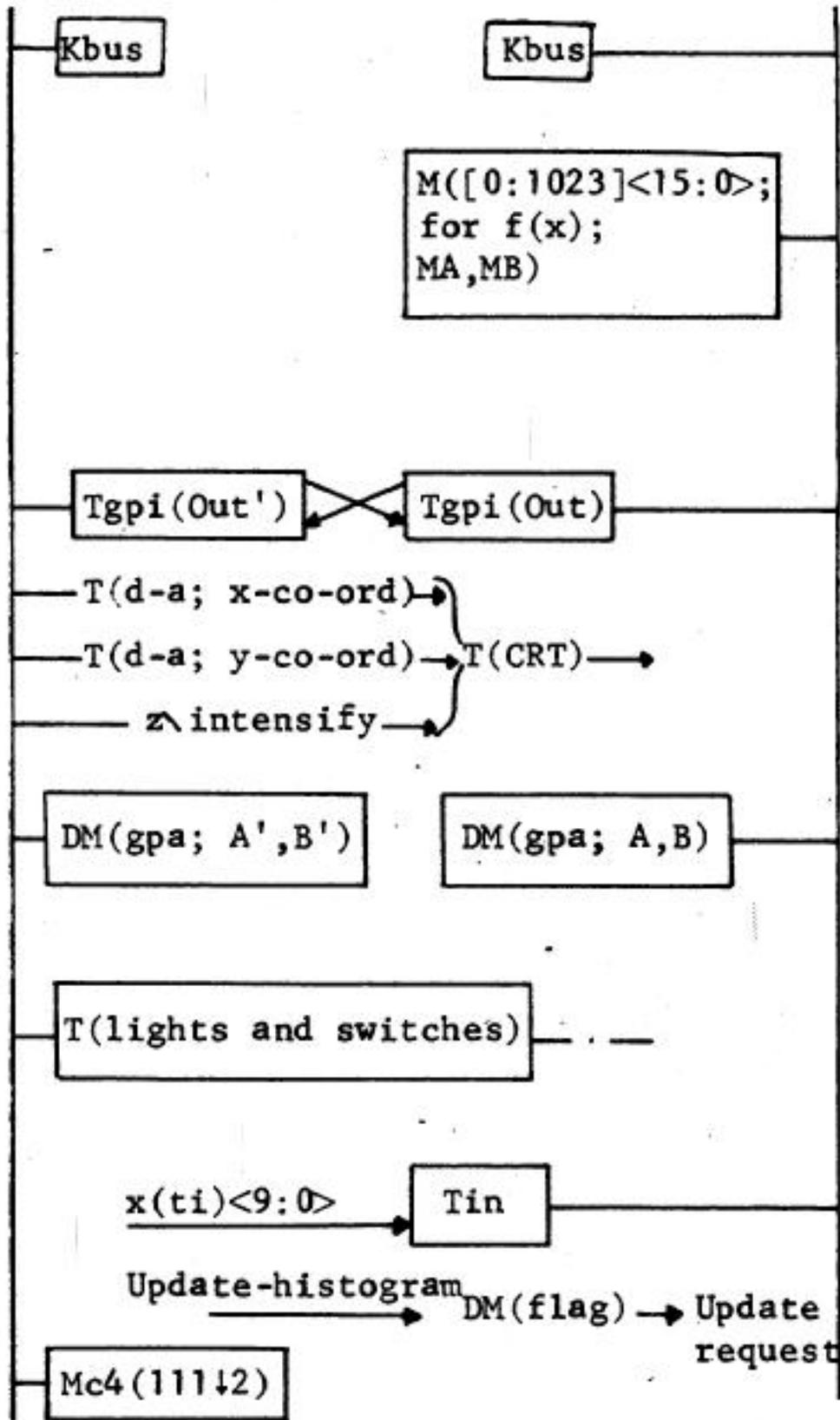


Fig. Hist-10. RTM diagram data part of a two-Bus histogram recorder for a simultaneous sample and

display.

3. Install, a switch in the display process such that when the switch is on, the process resets the memory word to zero after fetching the Y-coordinate (record). This feature can be implemented by changing the display process slightly.

#### ADDITIONAL PROBLEMS

1. Use a K(for-loop) and design the Kmacro(Initialize-memory)
2. Show how memory-initialization would be carried out more simply using a

218

[previous](#) | [contents](#) | [next](#)

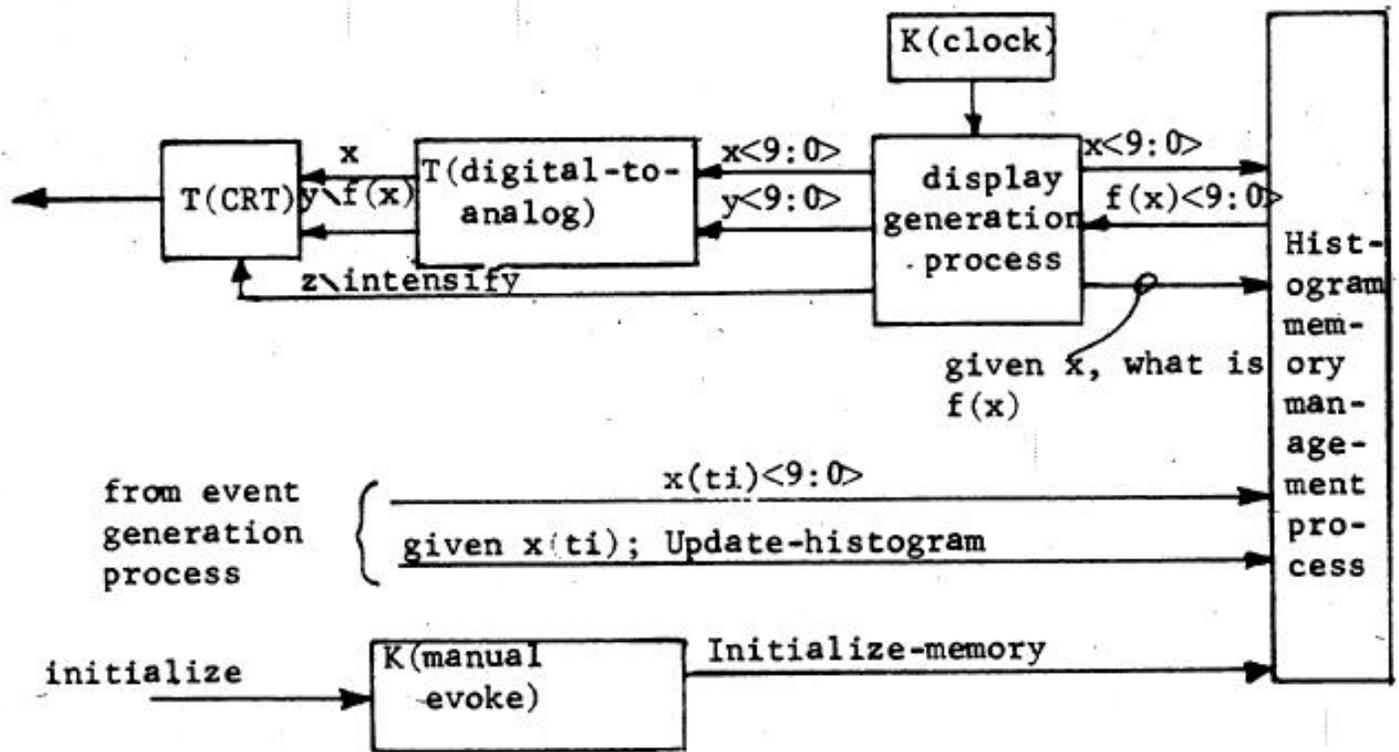


Fig. Hist-9. PMS diagram of a two-Bus RTM histogram recorder (independent update and display processes).

### SOLUTION 3

This scheme divides the histogram recorder into two parallel subprocesses: generation of display data and memory management; each subprocess employs its own Bus. This allows the record update rate to be increased while maintaining a constant display rate. The memory management process can initialize the memory, update the records, and supply data to the display process. The display process prepares the data for output, including any necessary scaling, and manages the display unit. The overall structure of a two-Bus system is shown in Figure Hist-9. The structure of the data part of this scheme is given in Figure Hist-10. The control part is almost the same as that given in the first solution. Minimum synchronization is needed between the memory and display processes; a straightforward method of synchronizing the two processes could employ request flags for use of the memory facility. The details of the control part for the two-Bus scheme are not given.

### SOLUTION 4

This scheme employs a two-Bus structure with a shared, two-port memory. Each port of the memory is connected to a Bus. In this design, the display process and the record update process are not connected via control. The two- port memory resolves conflicts for access in much the same way as the K(arbiter) did in Solution 2. The initialization process can be incorporated into the design in one of several ways:

1. Use a K(arbiter) on one of the Busses. The initialization process and one of the other processes will share that Bus.
2. Extend the number of ports to three to allow concurrent execution of all three processes.

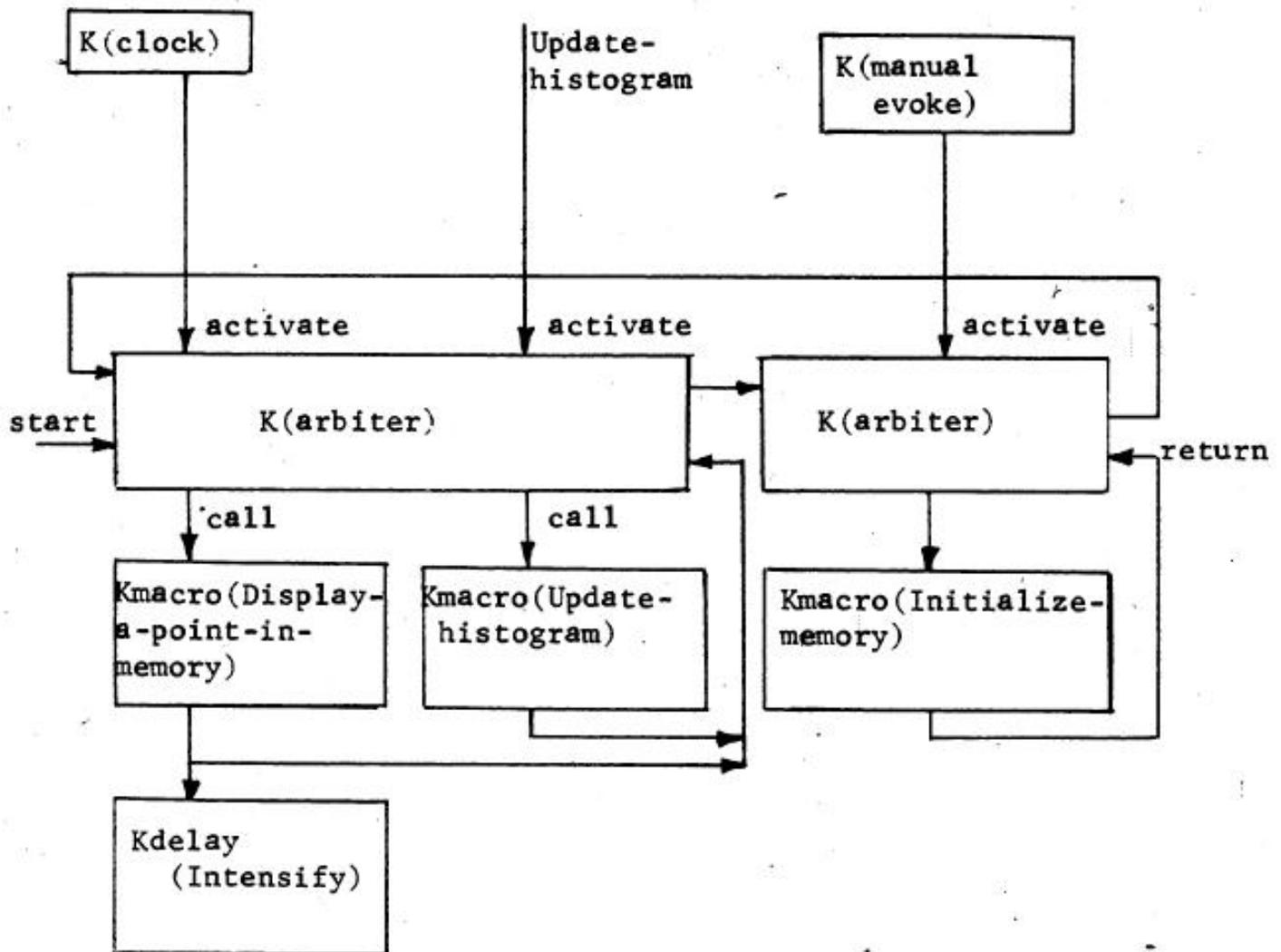


Fig. Hist-8. RTM diagram control part for a one-Bus histogram recorder using a K(arbiter).

The update rate can also be increased by polling the update request one or more times during the Kmacro(Display). In this way, any requests for update are honored, and merely interrupt the displayed points.

## SOLUTION 2

A variant of the previous scheme can be designed using the K(arbiter) module, described later in this chapter. The data part in Figure Hist-3 is assumed. The K(arbiter) module resolves conflicts for use of the Bus containing the memory. A shared resource. The main control part of the system is shown in Figure Hist-8.

This scheme uses the same processes as defined in the above solution: initialization, histogram update, and display of Figures Hist-5, 6 and 7. When an initialization request is given, the K(arbiter) is locked (if not already locked) and the initialization process is started. The process unlocks the K(arbiter) on completion. At this point another process request may lock the K(arbiter) and proceed. The histogram update process locks and unlocks ((arbiter) in the same manner as the initialization process. The display process operates in a similar way.

could not read the d-a register x-co-ord onto the Bus, but we do it here for compactness of representation.

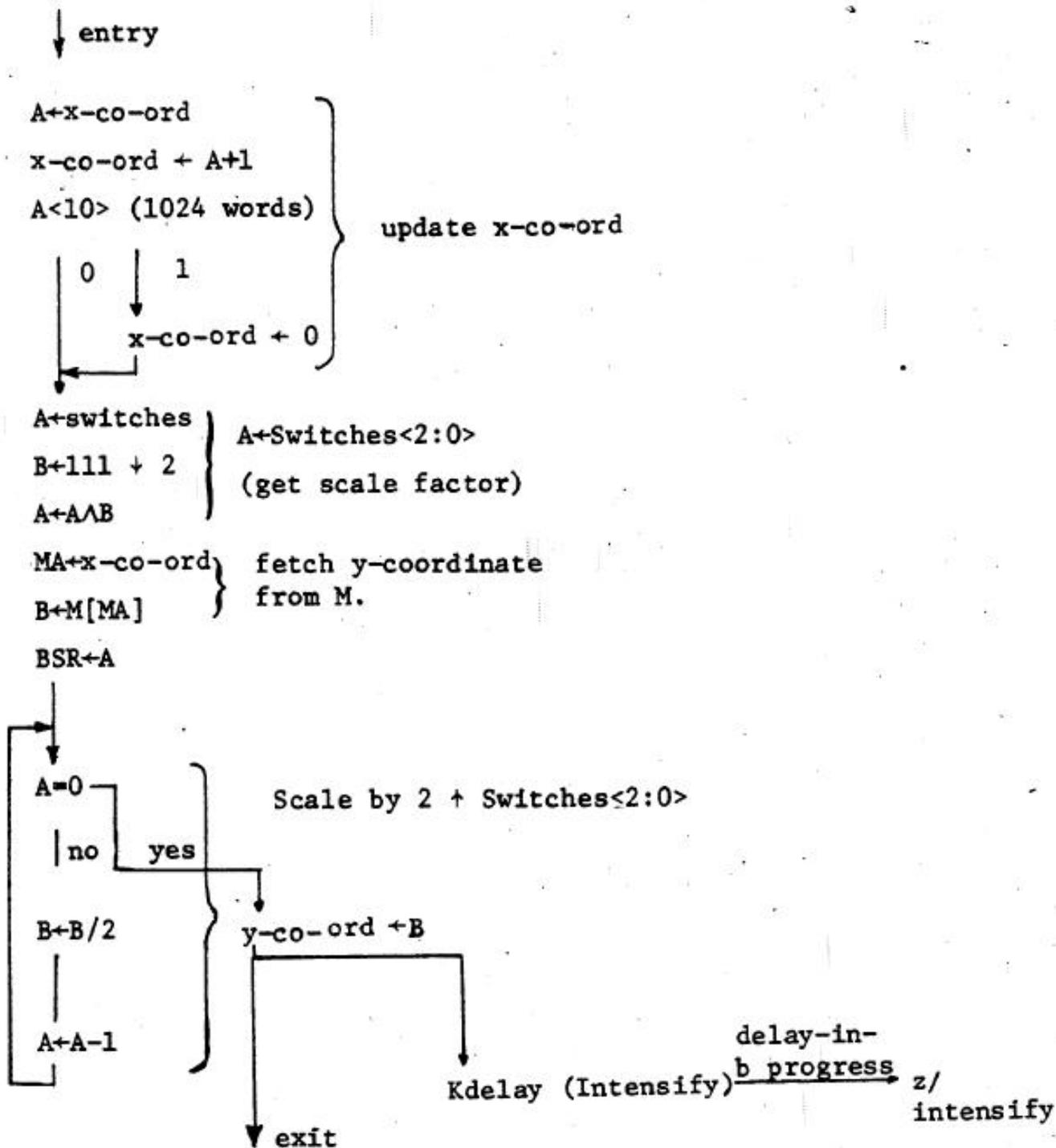


Fig. Hist-7. RTM diagram control part for a Display-point-in-memory macro.

With this scheme the display process takes approximately 30 steps, if six shifts are executed, with a total

time requirement of 20 microseconds. If record updating and display are operating concurrently, the display process limits the record update rate to an order of  $(1/20) \times (10^6)$  or 50 kilo-operations/second.. The rate may be improved by limiting the time during which record update requests are ignored.

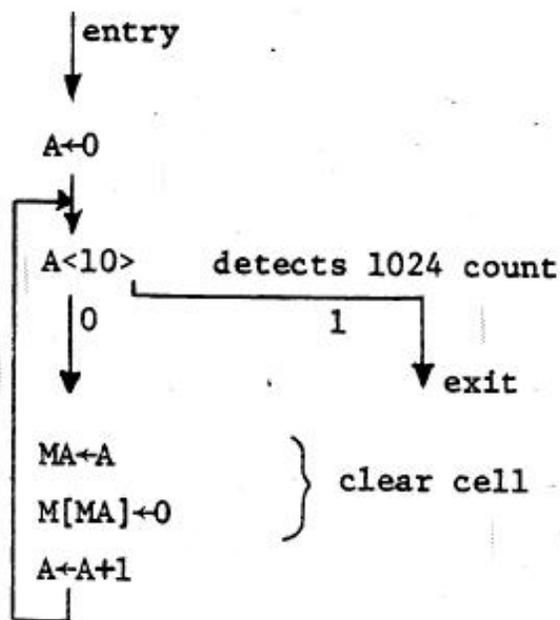


Fig. Hist-5. RTM diagram control part for initialize-memory macro (i.e. set all cells to 0).

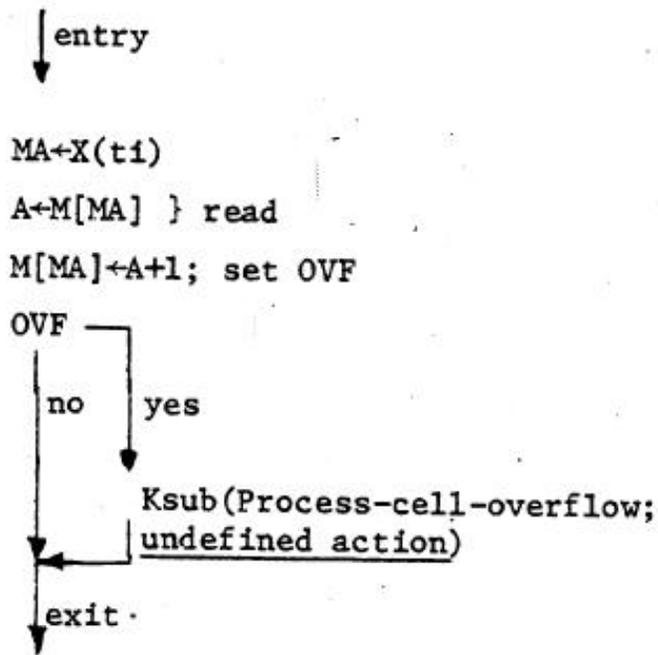


Fig. Hist-6. RTM diagram control part for update-histogram macro (i.e. add 1 to a

**Fig. Hist-6.** RTM diagram control part for update-histogram macro (i.e. add 1 to a particular count cell for  $X(t_i)$ ).

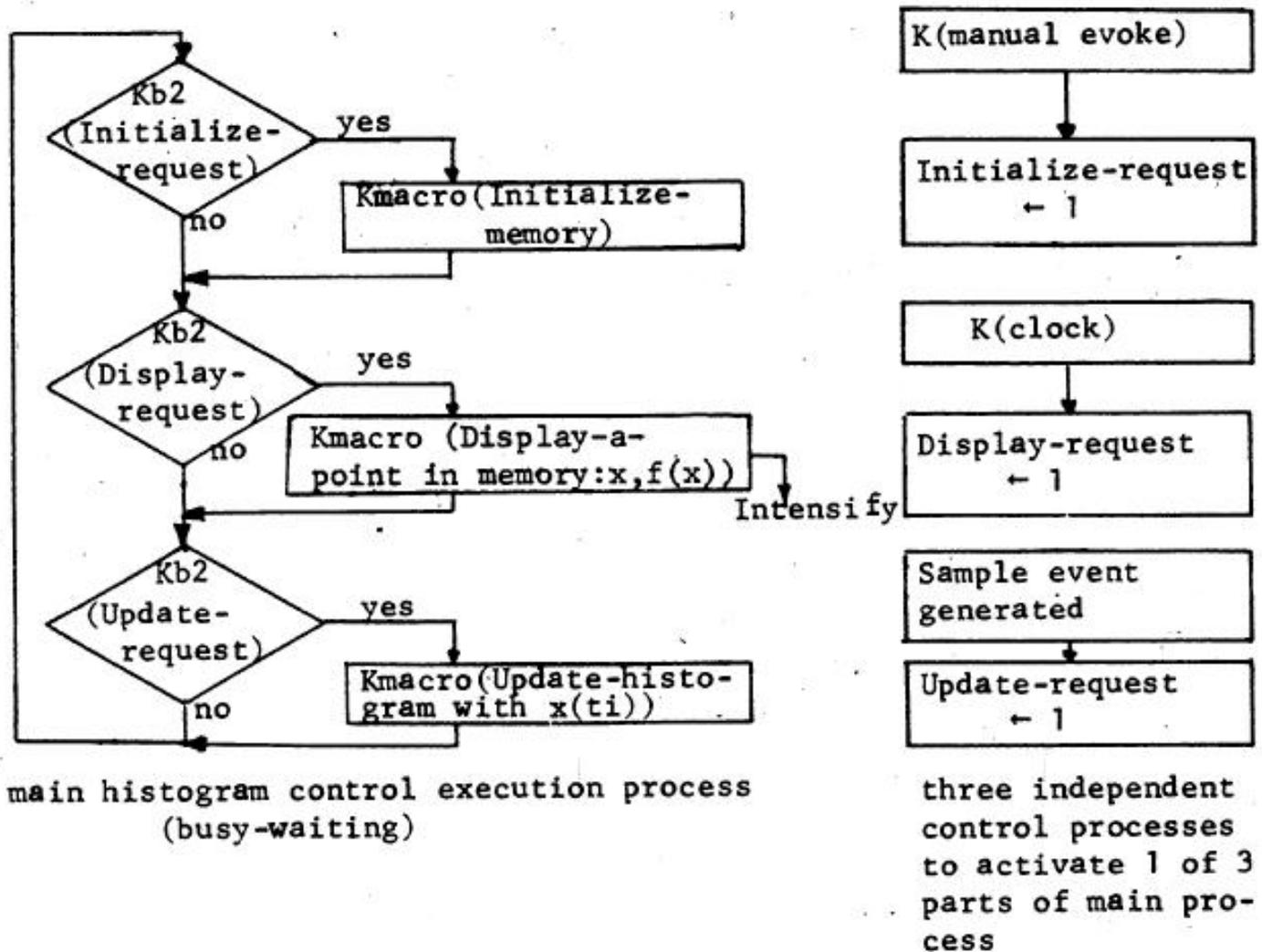


Fig. Hist-4. RTM diagram control part for a one-Bus histogram recorder using a busy-waiting scheme to synchronize facility requests.

Kmacro(Initialize-memory). The simplest process is the initialization of memory by clearing all cells to 0. (See Figure Hist-5.) It consists of a single loop to count from 0 to 1023, clearing cells 0,1,...,1023.

Kmacro(Update-histogram). This macro is shown in Figure Hist-6, and consists of taking an event,  $x(t_i)$ , and adding 1 to the cell selected by the event-value. This provides the function:  $M[x] \leftarrow M[x] + 1$ . Note that a subroutine for handling the event count overflow is indicated, but not defined.

Kmacro(Display-a-point-in-memory). Since the display unit accepts only 10 bits of data, the 16 bits of the memory word must be scaled for display. One possible scaling procedure is that of using only the 1-0 most significant bits of the word. With a prior knowledge of the possible range of values, these 10 bits are not necessarily the 10 high-order bits of the memory word. The data will be shifted to the right until

the 10 most significant bits are in the 10 low-order bits of the word. The switches  $\langle 2:0 \rangle$  will contain the number of shifts to apply to the memory word. Six shifts guarantee that all data is in the range zero to  $(2^{10})-1$ , but this many shifts may not be required for all cases. The control part is shown in Figure Hist-7 for  $\text{Kmacro}(\text{Display})$ . Strictly speaking, in the control part one

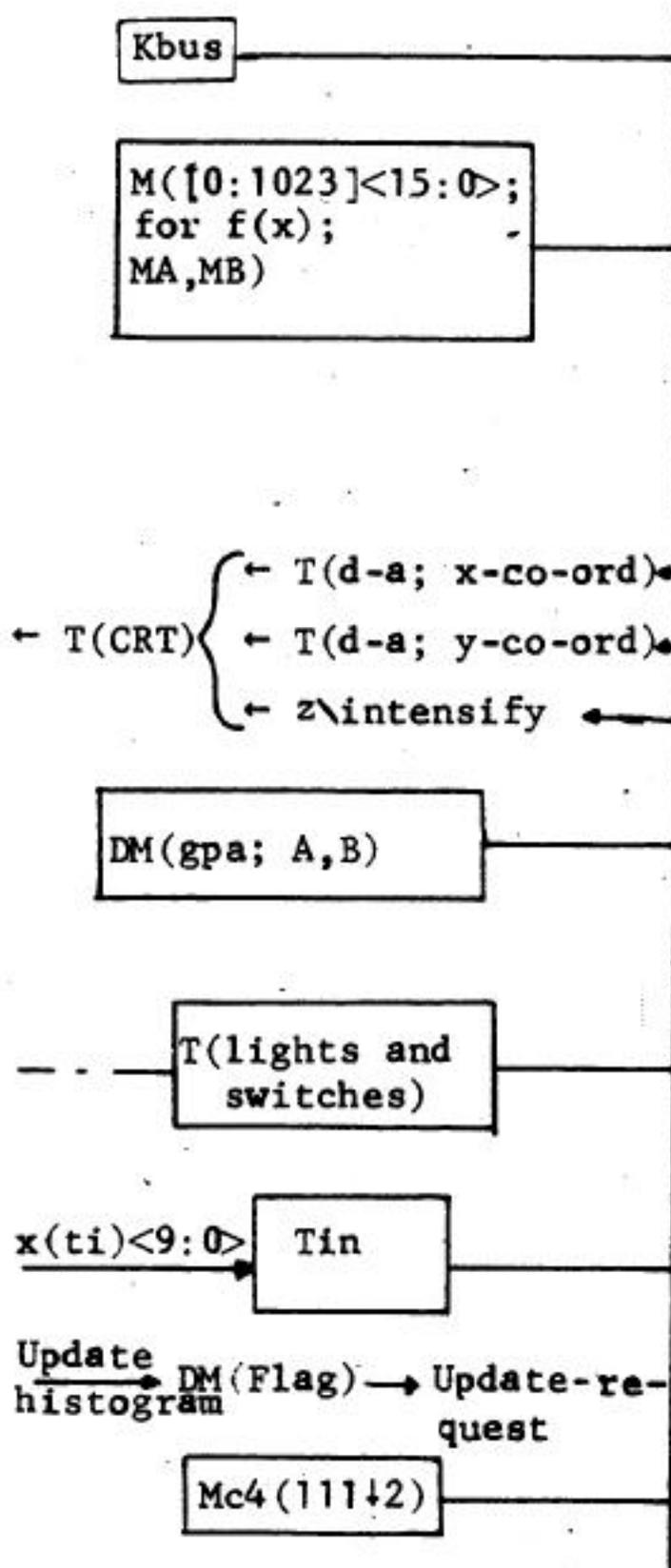


Fig. Hist-3. RTM diagram data part of a one-Bus histogram recorder.

## Control Part

The major control part is shown in Fig. Hist-4. It consists of four independent processes all of which run asynchronously with respect to one another. The main histogram control execution is shown on the left. It consists of a process which polls the other three processes, and if it finds any requests for activity, calls the appropriate Kmacro to carry out the action. On the right, the three independent processes make requests to be served.

## SOLUTION 1

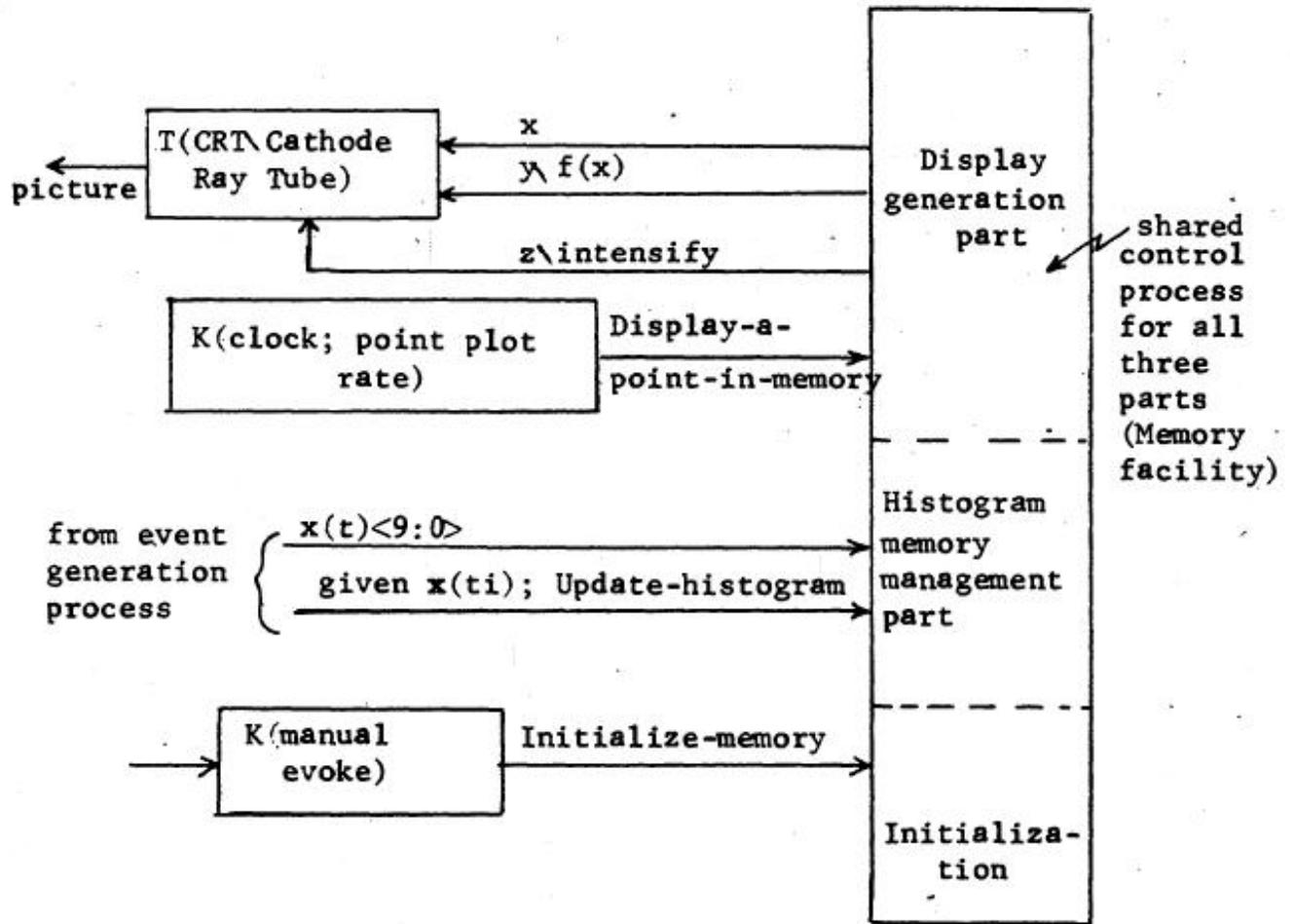


Fig. Hist-2. PMS diagram of a one-Bus RTM system histogram recorder.

The first system, shown in the PMS diagram of Figure Hist-2, has a single control process which drives the initialization, display and histogram-update processes. The system is capable of performing only one of the operations at a time, since each of the processes requires the common memory facility. The following synchronization schemes might be used to control the use of the common memory facility:

1. **Busily-waiting.** While idle, the memory facility is busily-waiting to be used, i.e., it is looking for requests for the facility from the processes. The memory only recognizes requests when it is idle. This prevents multiple processes from accessing the memory simultaneously.
2. **Event-driven.** In this scheme, the processes may perform some operations in parallel, but when a process requires the memory facility, it must wait until the memory is free before proceeding.

## Data Part

Figure Hist-3 shows the data part of a histogram recorder using one Bus. The various components are taken, to a large degree, from the problem statement. These components are: the memory to hold (record) the histogram; two T(d-a) converters to deflect the T(CRT) oscilloscope beam; a DMgpa to carry out the arithmetic of the process; a T(switch) to accept data about how the data might be manipulated (i.e., scaled); a T(input) for accepting event,  $x(t_i)$ , input; and an M(constants).

## PROBLEM STATEMENT

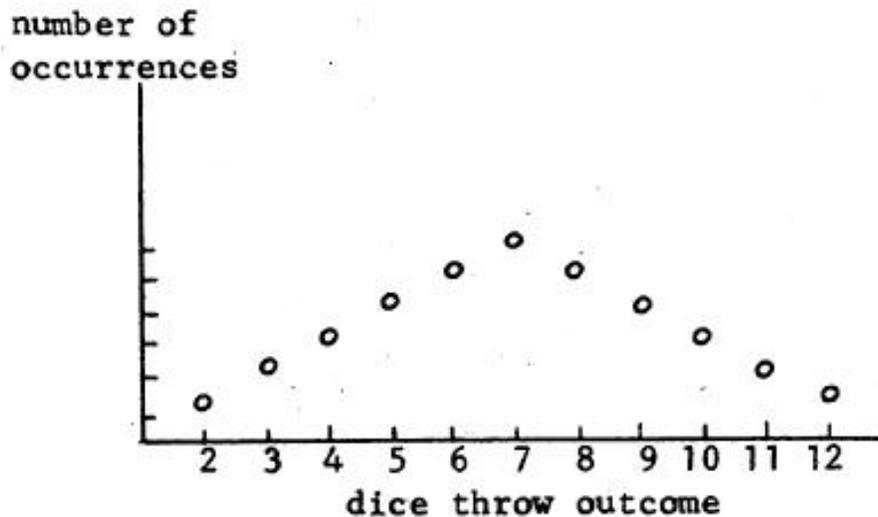


Fig. Hist-1. Example of a histogram which records the distribution of dice throws.

Design an RT system to implement a histogram recorder. The number of available memory words is 1024 with a word size of 16 bits.

## DESIGN CONSIDERATIONS

There are limitations which affect this design. If the number of event types,  $M$ , is greater than the number of memory words available, all possible events cannot be uniquely represented. In this case a record can be maintained for only a subset of the original set of events. A possible solution to this problem is to group the events into distinct classes such that the number of classes does not exceed the number of available memory words. A second limitation is that only a finite range of values may be stored in a record, given a word size of  $W$  bits. The histogram recorder can correctly maintain only those records with values in the range zero to  $(2^W)-1$ . The use of scaling or shorter time periods of observation are possible solutions to this problem.

Finally, it is important that the recorder operate in a short time to minimize the necessary duration of time between two events at  $t_i$  and  $t_i + \Delta t$ . In this way, the event occurrence rate can be high.

The following functions are necessary in the design:

1. Initialization - A switch should be provided which initiates a process to reset each word in

memory to zero.

2. Histogram update - This is a process which increments the histogram by one for each event,  $x(t_i)$ .

3. Display - A display process which reads a memory word and displays the value on a T(CRT display) should be incorporated in the design. This process can be used to successively plot each value of  $M[0:1023]<15:0>$  as shown in Figure Hist-1. A K(clock) should provide a signal to initiate the display process which should be capable of plotting a new point every 30 microseconds. (This fast plotting time is required so that the 1024 points can be plotted in about 30 milliseconds, providing a flicker rate of about 33 frames/second.)

figures are approximate). Assuming a random occurrence of events on all channels, the time to perform all operations is  $8 \times 1.8 + 9.35 \sim 24$  microseconds. This provides an aggregate counting rate on all channels of 42 KHz. This is not a strict bound; if the events do not occur randomly, the polling loop may execute consistently above or below the average figure with different performance characteristics. However, polling is clearly the limiting time-consuming process and any general performance increase will be gained only if the polling time is decreased.

## ADDITIONAL PROBLEMS

1. Design a solution that has a polling process consisting of a branch tree using combinational circuitry at the branch points. That is, a D network would carry out the computations at each of the branch points.
  - a. What effect does this have on the system performance?
  - b. Design the polling network so that it polls in a more equitable manner.
  - c. Write the initialization subroutine to set the event counts and time base.
2. Write a software, n-channel EPUT meter for your favorite computer. What are the performance characteristics of the program?
3. Design an output process for the n-channel EPUT meter; consider using BCD for the output data. How does this process affect the design, its accuracy, maximum frequencies, etc.?
4. Modify the EPUT meter of solution 1 to have independent time base generators for each channel. Assume a single K(clock) and that the values of the time bases are stored in a memory array. Give the performance characteristics of the system.
5. How would multiple DMgpa's affect any of the designs above?
6. Design a polling scheme which will find the selected bit in a constant time by using the binary searching technique. With this technique, the control first examines whether a flag is on in the first 1/2 of the word or not. It then proceeds to subdivide the search space in two each time. Such an approach can be done using either a combinational or register transfer approach.

## HISTOGRAM RECORDER

**KEYWORDS:** Display, event, histogram, initialization, synchronization, record, waveform analysis, arbiter.

A histogram recorder is a device, principally a memory, which records an occurrence count for each member of a set of discrete events. For the purposes of this problem, an event of this form at a particular time,  $t_i$ , has a 10-bit value, and is denoted  $x(t_i) < 9:0 >$ . A record is defined as the occurrence count, taken over a finite time period, for an event.

An example of a histogram is given in Figure Hist-1. The 11 events represent all of the possible outcomes from a single throw of a pair of dice. The values stored in the histogram are the records for each possible outcome over some time period, e.g., 1000 throws. Histograms are usually presented by such a visual display, hence the recorders described below will have display capabilities.

If the set of events contains  $M$  elements, the histogram recorder can be implemented as a memory of  $M$  words. The records for events  $E[0:M-1]$  can be uniquely assigned as the value of memory words  $MW[0:M-1]$ .

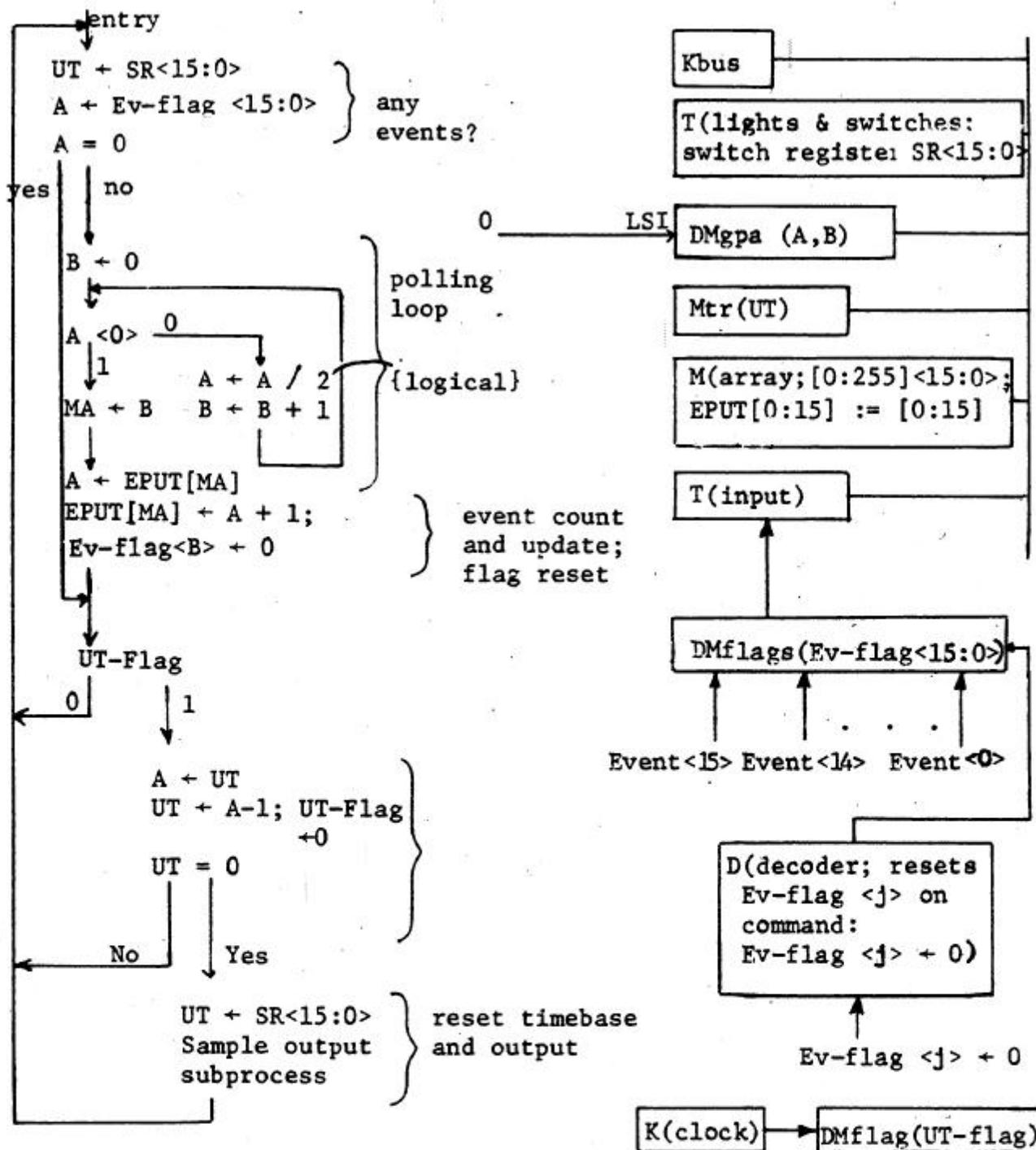
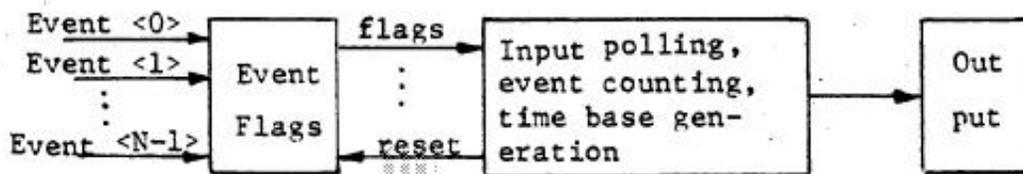
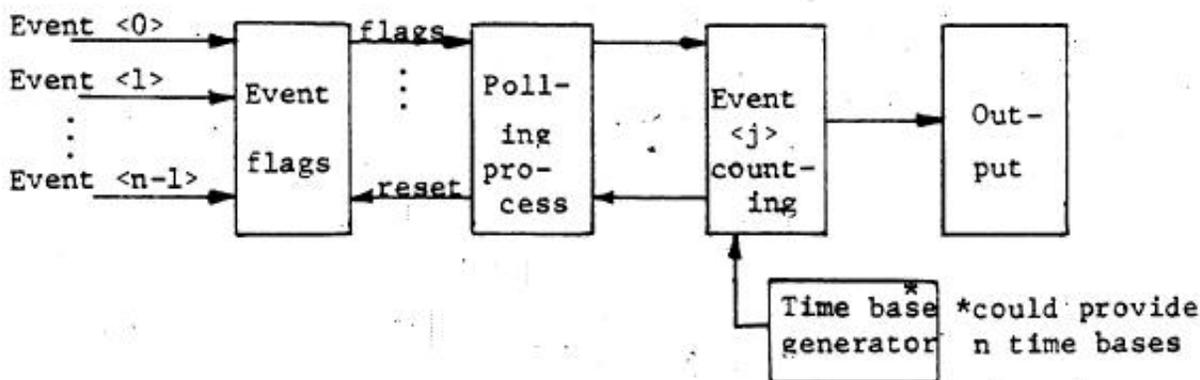


Fig. EPUT-5. An RTM diagram of an n-input time-shared EPUT meter with a common time base for all channels.

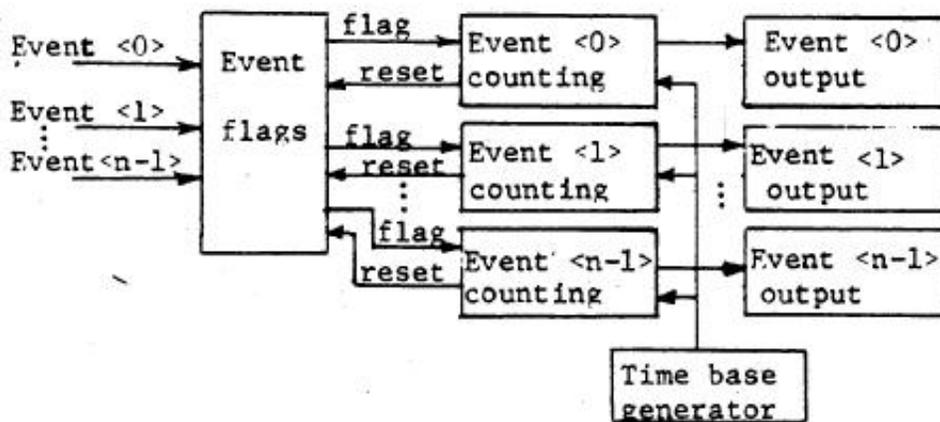
[previous](#) | [contents](#) | [next](#)



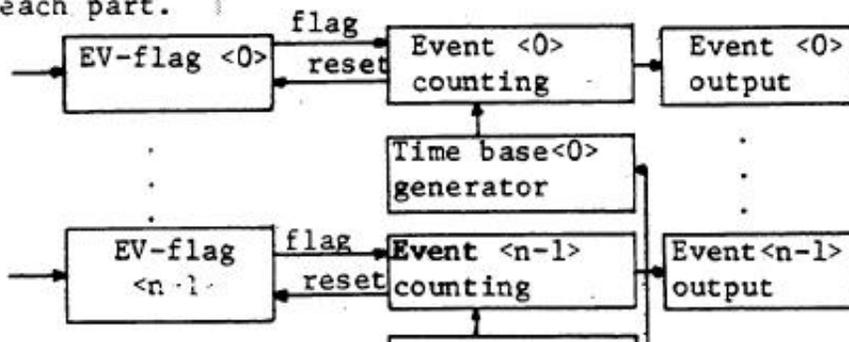
a. An RTM diagram of an n-input EPUT meter using all time-shared resources.



b. An RTM diagram of an n-input EPUT meter using a pipeline structure for separate polling and event counting.



c. An RTM diagram of an n-input EPUT meter using a time-shared time base generator and independent counting and display processes for each part.



d. RTM diagram of independent single input EPUT meters using a time-shared clock.

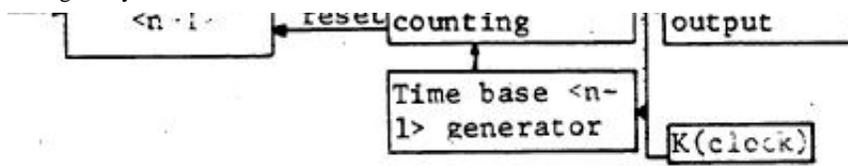


Fig. EPUT-4. Four RTM diagram of EPUT meters using various degrees of time-shared resources.

5. Result output - A process must be implemented to output the results of the EPUT meter. For this problem however, the output process will not be examined.
6. As in the case of the single-input EPUT meter of the previous problem, the limitations on the clock and *event* frequencies must be considered.
7. The extent to which the functions expressed in 2 to 5 above can be executed in parallel is of interest in the design problem.

## SOLUTIONS

The overall structure of four systems in the design space of time-shared EPUT meters will be examined before giving a detailed solution to the problem.

Figure EPLJT-4a shows a completely time-shared system which can be constructed using a single Bus. Each event count is operated independently and there can be either a single time base for all channels, or each channel can have a separate time base by time-sharing a central clock. With a single control structure and Bus, the systems capabilities are shared among all of the inputs.

Figure EPUT-4b shows a system which performs input polling and event counting in parallel. The counting and polling processes are still shared among all inputs. The time base generation shown is common to all the channels, although a time base generator for each channel could be constructed using a time-shared clock. The coupling between the time bases and the EPUT counting would be more complicated in the latter case than in the single time base situation.

Figure EPUT-4c shows a system with a shared polling process and clock. The system exhibits a high degree of parallelism in that each event has its own counting and output processes. A single time base is used for all inputs.

Figure EPUT-4d shows the extreme in parallel design;  $n$  independent systems sharing only a clock for time base generation.

### Solution 1

With this overview of possible time-shared EPUT meters, a specific system will be designed to illustrate solutions to the various design issues. Figure EPUT-5 presents a time-shared system which utilizes a single Bus and control for  $n$  inputs, and a common time base for all inputs. The operation of the polling process is of principal interest. The sixteen event flags,  $Ev\text{-}flag\langle 0:15 \rangle$ , are interfaced to the system through a T(input interface); on command, the event flags are transferred to the A register of a DMgpa. If A is non-zero, a shift and count loop is initiated to find the first bit (from the left) that is a one, then the

count for this event is incremented and the flag for the event is set (notice that the single command, Ev-flag<j><-0 is used). With this strategy, the events have implicitly assigned priorities, Ev-flag<15> is of highest priority, Ev-flag<14> next highest, and so forth until Ev-flag<0> with the lowest priority. If the higher numbered events occur at too high a rate, the lower numbered events will not be correctly counted. If, instead of always polling from the left, the process continued polling from the point at which an event was last detected, all channels would be handled more equitably.

The performance of this system is determined by the time required to read the event flags, poll the events and update the event and time base counts. The flags can be read in .7 microseconds, the polling loop has an initial overhead of .85 microseconds and each pass through the loop represents 1.8 microseconds, event count updating takes 5.7 microseconds, time period count updating takes 1.6 microseconds, and resetting the time base takes .5 microseconds(all timing

## PROBLEM STATEMENT

Design a system which behaves as  $n$  EPUT meters, i.e., a system in which  $n$  different event streams may be counted.

## DESIGN CONSIDERATIONS

1. The interface between the system and the external events is a straightforward extension of the single input system: a vector of DMflag's, one per event, will be used to signal event occurrences to the system. The flags will be denoted:  $Ev\text{-}flag\langle 0:n-1 \rangle$ . The flags may be reset in one of two ways: (1) each flag can be reset explicitly with one of the  $n$  independent commands:  $Ev\text{-}flag\langle 0 \rangle \leftarrow 0$ ,  $Ev\text{-}flag\langle 1 \rangle \leftarrow 0$ , ...,  $Ev\text{-}flag\langle n-1 \rangle \leftarrow 0$ ; (2) a decoder network can be used to reset the flags with a selection parameter used in the single command,  $Ev\text{-}flag\langle j \rangle \leftarrow 0$ . Figure EPUT-3 shows both, of these schemes. The solutions described below will assume that one of these two schemes has been implemented and that the number of inputs is sixteen.

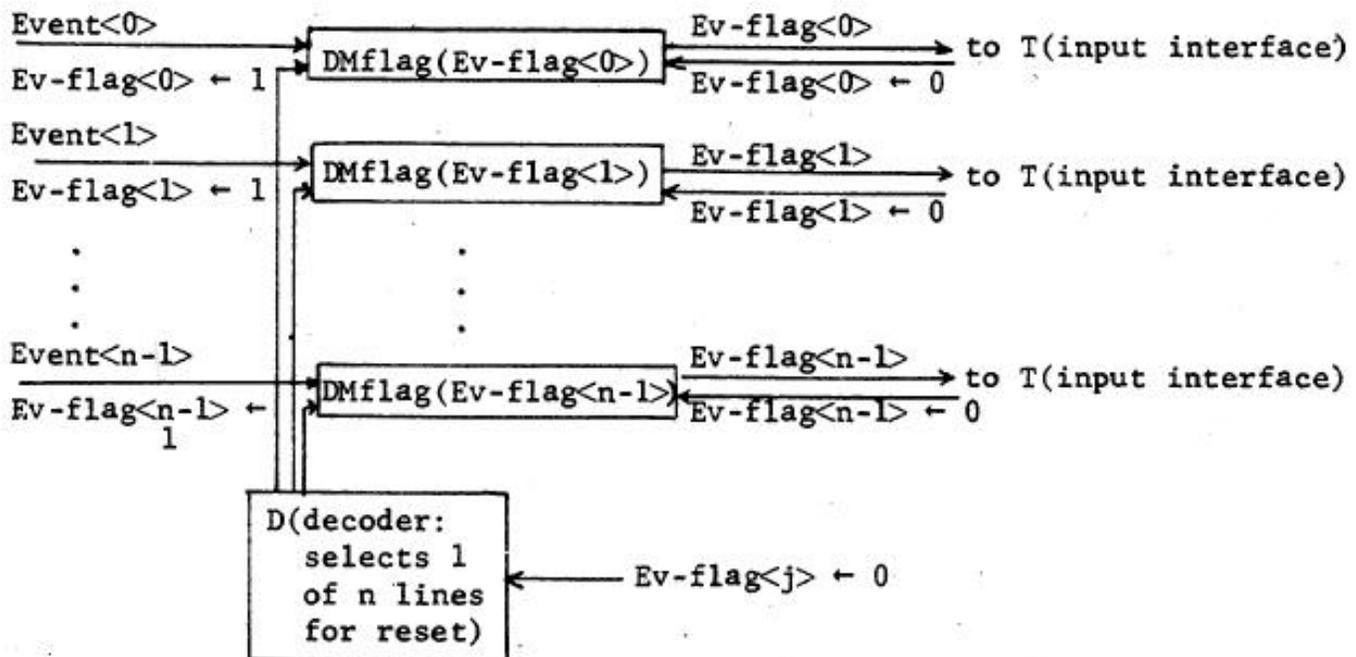


Fig. EPUT-3. An RTM system which provides two methods for resetting event flags.

2. Input polling - The input events must be sensed and a corresponding input channel determined. Some

polling strategy must be developed to detect event occurrences recorded in the event flag.

3. Event counting - Once an event has been detected at Ev-flag<j> the count for that event must be updated. A separate event count must be maintained for each event; the question is Where? Separate DMgpa registers could be used, but the cost would be prohibitive. Alternatively, the event counts could be held in a vector of memory words and a single DMgpa time-shared for incrementing the event counts.

4. Time base generation - Two possibilities exist for time base generation; either use a common time base for all channels or provide an independent time base for each channel.

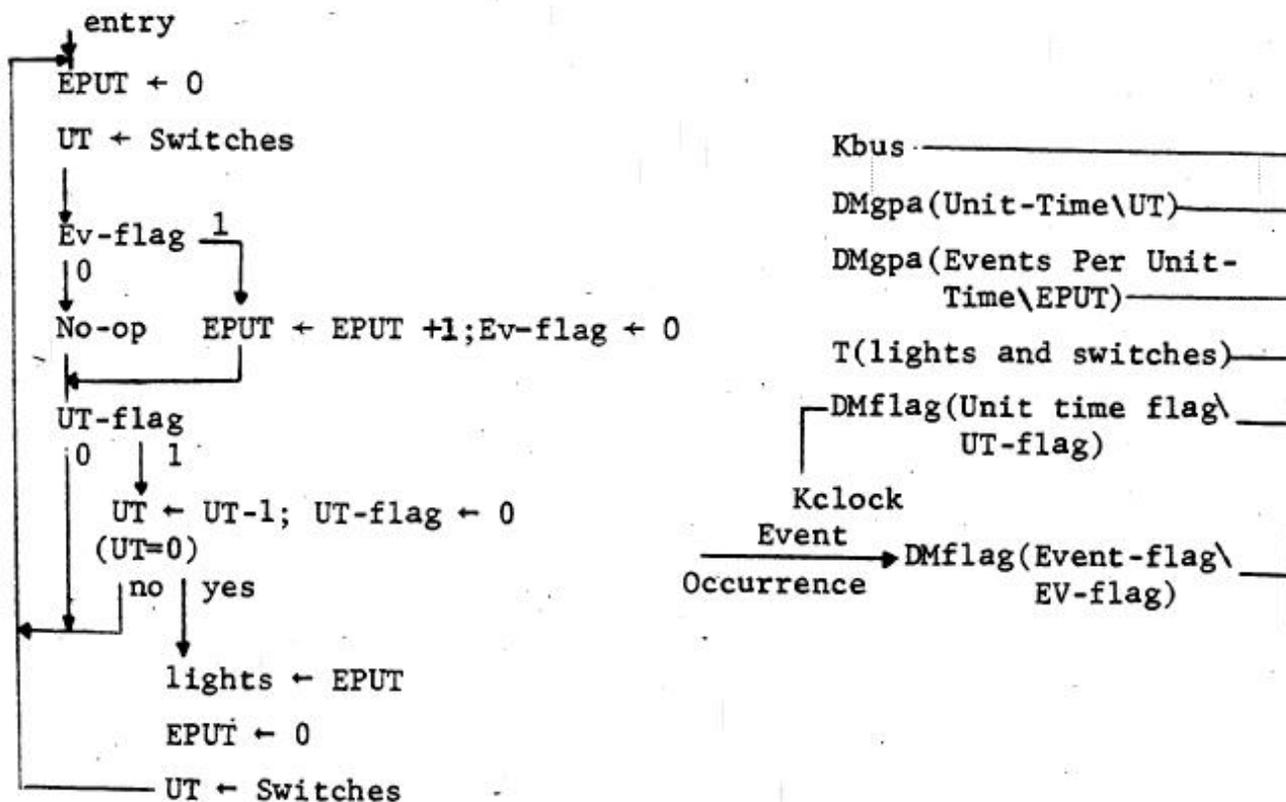


Fig. EPUT-2. An RTM system diagram of an Events Per Unit Time\EPUT meter.

### TIME-SHARED EPUT METER

**KEYWORDS:** Parallelism, time-sharing, polling

This design problem illustrates how the EPUT meter design of the previous problem might be modified such that a single system would give the appearance of being many independent EPUT meters. Such a system falls into the space of time-shared systems design, the main objective being the design of a single system which behaves as n independent systems. The gain from this design approach is that it is usually more economical to have a single system handle n inputs than to have n independent systems handle one input each. A drawback to the centralized, time-shared approach is that unless such systems are handling a close-to-capacity number of inputs, they are unnecessarily complex and expensive.

The technique of time-sharing usually focuses upon a resource which must be shared among all processes in the system. The trade-off between n independent systems and a single time-shared system is that each process in the time-shared system can use only 1/n of the shared resource; expense has been

traded for performance. Of course time-shared systems are not constrained to have a single instance of a shared resource; the  $n$  processes may share several instances of the same resource (approaching  $n$  independent systems in the limit), or the processes could share the services of many resources combined to form a single system.

in standard units. As the input events arrive (Figure EPUT-1b) they are counted and recorded in a memory (register). At the end of the time base, the results are displayed, the count is zeroed, and the next time base (counting period) is started. The system can also be used to measure the frequency of a sinusoidal input (Figure EPUT-1c) by adding an input section (Figure EPUT-1d) which amplifies the sinusoidal input (Figure EPUT-1e) to produce 0 and 1 signals only. These are subsequently differentiated and clipped to allow only positive voltages (Figure EPUT-1f), which are used as a clocking input to Ev-flag.

## PROBLEM STATEMENT

Design a general purpose EPUT meter.

## DESIGN CONSIDERATIONS

The limit on the frequency of events is perhaps the principle design issue in this problem. As mentioned above, the end of the time base initiates a display of the event count, the count is zeroed and the next time base is started. All of these operations must be completed before the next event occurrence if all events are to be recorded. If the operations are not completed sufficiently rapidly, an event which occurs during the processing of the operations would be lost if another event occurred before the first event was counted. A partial solution to this problem is to allow a short interval of time between successive time periods during which events are ignored. This scheme would yield correct results for each individual time period, but would not reflect a true total count. Even using this scheme, the event frequency is limited by the speed with which the meter can update its event counter and/or the clock count. Again, if two events occur while the event or clock count is being updated, the first event of the two will not be counted. Similarly, the clock pulse frequency cannot be so high that two clock pulses arrive before the clock count can be properly updated (decremented).

## SOLUTION

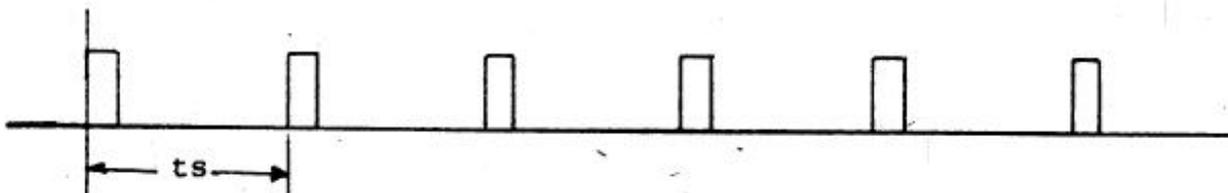
The structure (and behavior) of an EPUT meter is given in Figure EPUT-2. The data part requires two registers: UT to hold the unit time count (programmable clock); and EPUT to hold the event count. For greatest speed, two DMgpa's are used so that both registers can be incremented without extra transfer operations. The switch register in the T(lights and switches) holds the variable time base parameter for the programmable clock. The lights in the T(lights and switches) display the event count in binary. A clock pulse sets the DMflag (UT-flag) and an event occurrence sets the DMflag (EV-flag). If no interval is allowed between time base periods, the maximum event-frequency is approximately 250 KHz. If an interval is allowed between sampling periods, this rate can be increased to approximately 500 KHz. The maximum allowable frequency for the clock pulses is 1 MHz.

## ADDITIONAL PROBLEMS

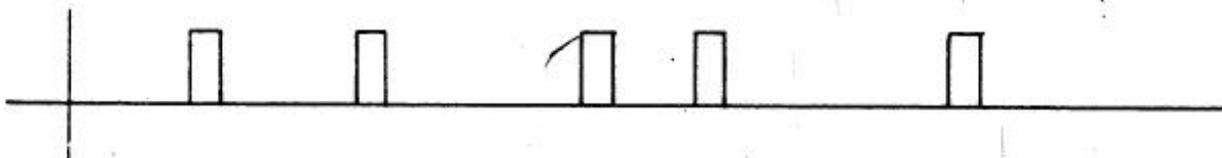
1. Now were the frequency rates for the EPUT meter described above computed?
2. Would an increase or decrease of the clock pulse frequency allow a higher event frequency?

203

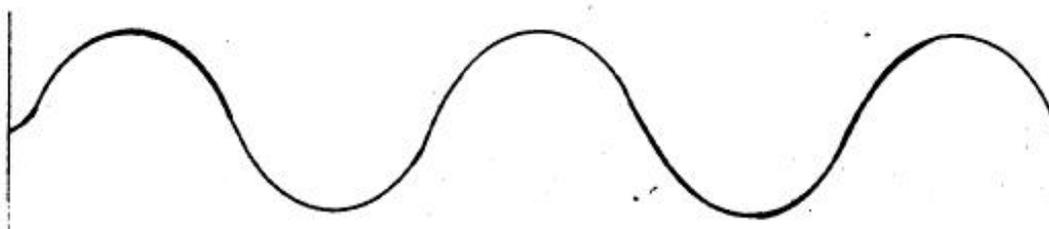
[previous](#) | [contents](#) | [next](#)



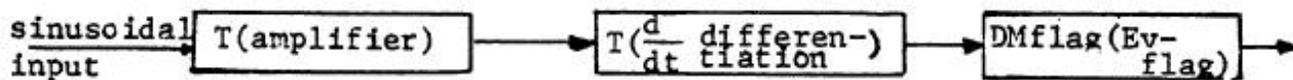
a. Graph of time base operation for an EPUT meter



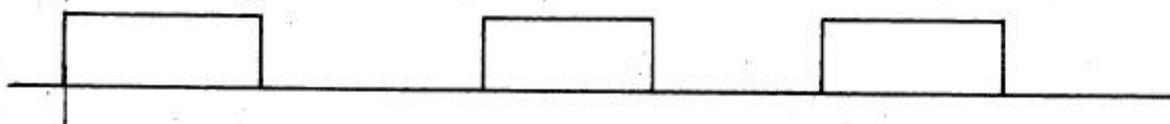
b. Graph of event occurrence input to an EPUT meter



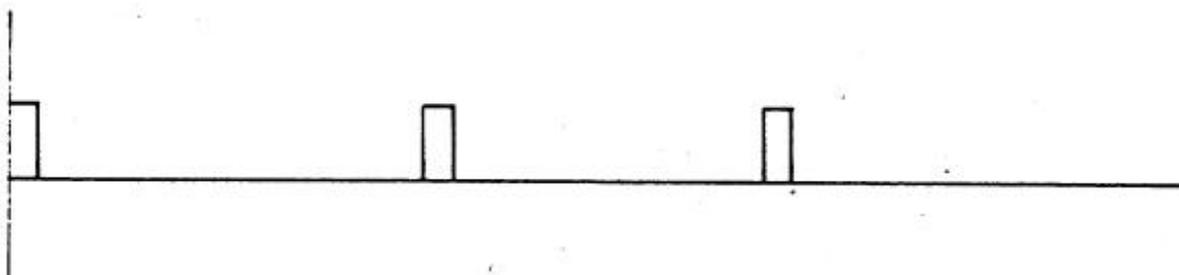
c. Graph of sinusoidal event occurrence input to an EPUT meter



d. RTM diagram of input section to process sinusoidal input to an EPUT meter



e. Graph of sinusoidal event occurrence input after amplification



f. Graph of sinusoidal event occurrence input after amplification and differentiation

and differentiation

Fig. EPUT-1. Examples of waveforms associated with EPUT meters.

202

[previous](#) | [contents](#) | [next](#)

desired, e.g., one part in  $2^{16}$ , the maximum frequency that can be counted is only 10 Hz. However, it should be noted that a 10-bit T(digital-to-analog) is not capable of such a fine resolution. For an error of one part in 1024( $\delta = 2^6$ ), the error is:  $2^6/2^{16} = 1/2^{10} \sim 10^{-3}$  or 0.1%. This provides for a maximum frequency of  $\sim 666$  Hz. Increasing the allowable error to one part in  $2^7$  ( $\sim 0.8\%$ ) yields a frequency limit of  $\sim 5$  KHz. By increasing the frequency to 10,000 Hz, the error increases to  $(10^4) \cdot (3/2) / 10^6 = 1.5\%$ . What can be done to further increase performance, considering the values for  $f$  and error? As has been indicated previously, the ability to make multiple simultaneous register assignments can significantly improve system performance. In this case, the two transfer operations would be performed as: analog-out  $\leftarrow$  output  $\leftarrow$  output +  $\delta$ . This procedure would halve the processing time and double the frequency.

Another parameter not yet examined is the frequency quantization for a fixed sample time,  $t_s$ . For a fixed sample time,  $t_s$ , the frequency formula becomes  $f = K \cdot \delta$  (where  $K = 1/(2^{16} \cdot t_s)$ ) which shows that the frequency varies linearly with respect to  $\delta$ . For  $t_s = 1.5$  microseconds, the frequency change per unit is:  $f\text{-change}/\delta = 1/(2^{16} \cdot 1.5 \cdot 10^{-6})$  10 Hz. The frequencies possible from this function generator with  $t_s = 1.5$  microseconds are: 10, 20, ...,  $10 \cdot \delta$ , ..., 666 KHz (independent of error). Different frequencies may be obtained by varying the sample time,  $t_s$ .

## ADDITIONAL PROBLEMS

1. In the above generator the amplitude remains fixed. Suppose the amplitude is to be varied over a range and step size by input parameters. What would the parameters be for such a generator? Design such a generator and analyze its characteristics.
2. In the generator presented above, the error varies with respect to the frequency. Using a programmable clock, design a generator which has a constant maximum error for all frequencies; the maximum error should be set by input parameters. Analyze the characteristics of your design.
3. Figure SG-3 shows the output of a triangle waveform generator. Design a generator with such an output and determine the relationship between  $\delta$ , Error,  $f$ , and  $f\text{-change}/\delta$ . How does the sample time,  $t_s$ , affect the performance of the generator.
4. Design a waveform generator for the exponential function,  $e^{-t}$ . Describe the error as a function of  $t$ .
5. Design a waveform generator for  $\sin(t)$ ,  $\cos(t)$  for  $0 \leq t \leq 2$ . Note that  $\sin$  and  $\cos$  can be defined in terms of each other by integrating.

## EVENTS PER UNIT TIME \EPUT METER

**KEYWORDS:** Waveform, time base, event, frequency limit

An EPUT meter is about the simplest waveform analyzer that can be constructed. It merely counts pulses (events) within a certain measured time period. A description of the structure of an EPUT meter was given in the introduction of time-based systems. The basis of the time period measurement is a clock which produces pulses at a frequency of  $1/t_s$  Hertz as shown in Figure EPUT-1a. The clock counts for a variable time to give the basic time unit measurement; that is, it forms the basis for a programmable clock. The time base may be any fraction of a second, but it is usually a multiple or submultiple of ten

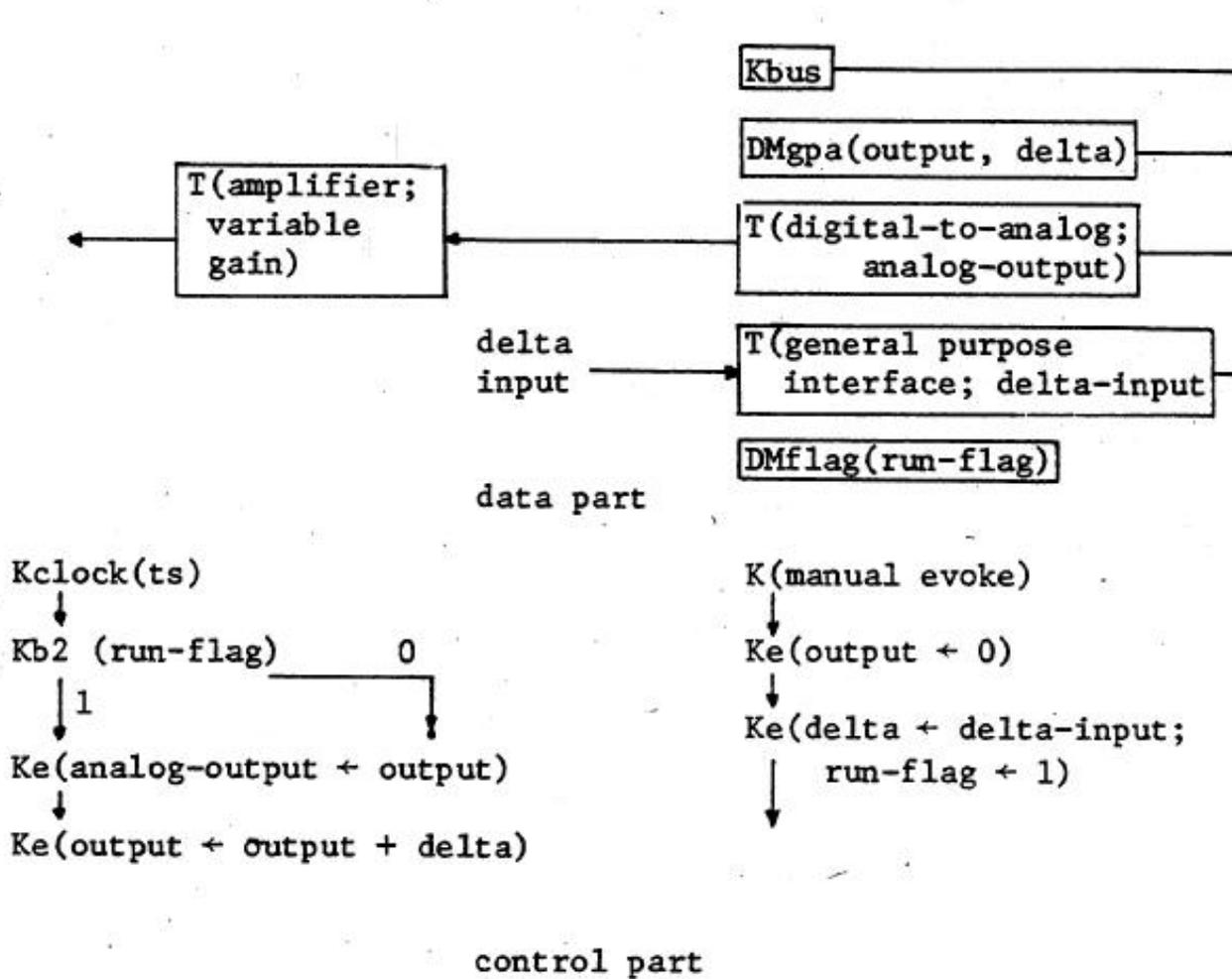
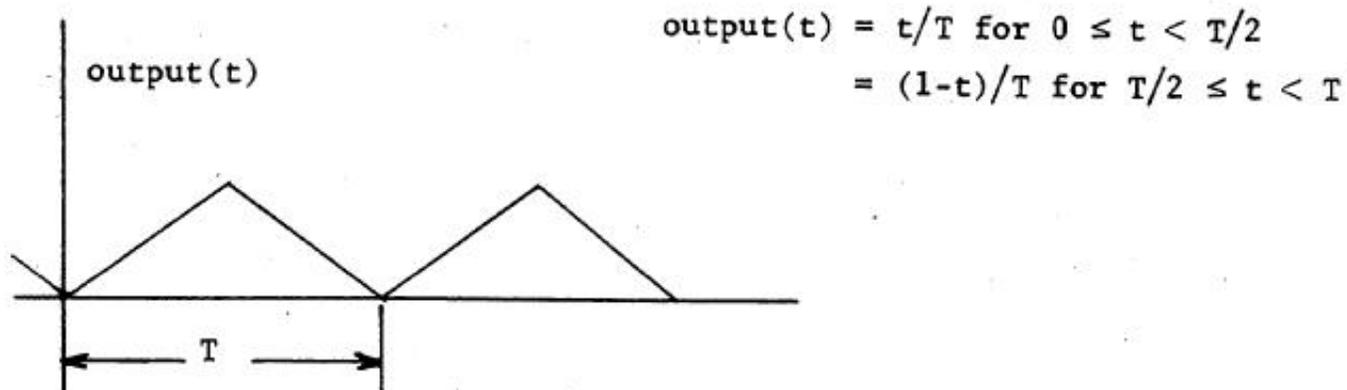


Fig. SG-2. An RTM diagram of a sawtooth waveform generator.



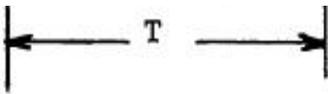


Fig. SG-3. Graph of an example of a triangle waveform.

$$\text{output}(t) = \frac{t}{T} \text{ for } 0 \leq t < T$$

$$\text{output}(t) = \text{output}(t+nT) \text{ for } n=0, \pm 1, \pm 2, \dots$$

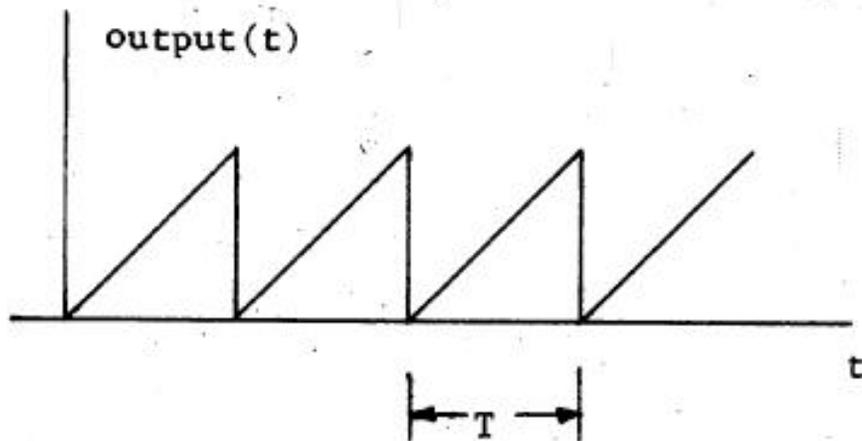


Fig. SG-1a. Graph of an example of a sawtooth (ramp) waveform.

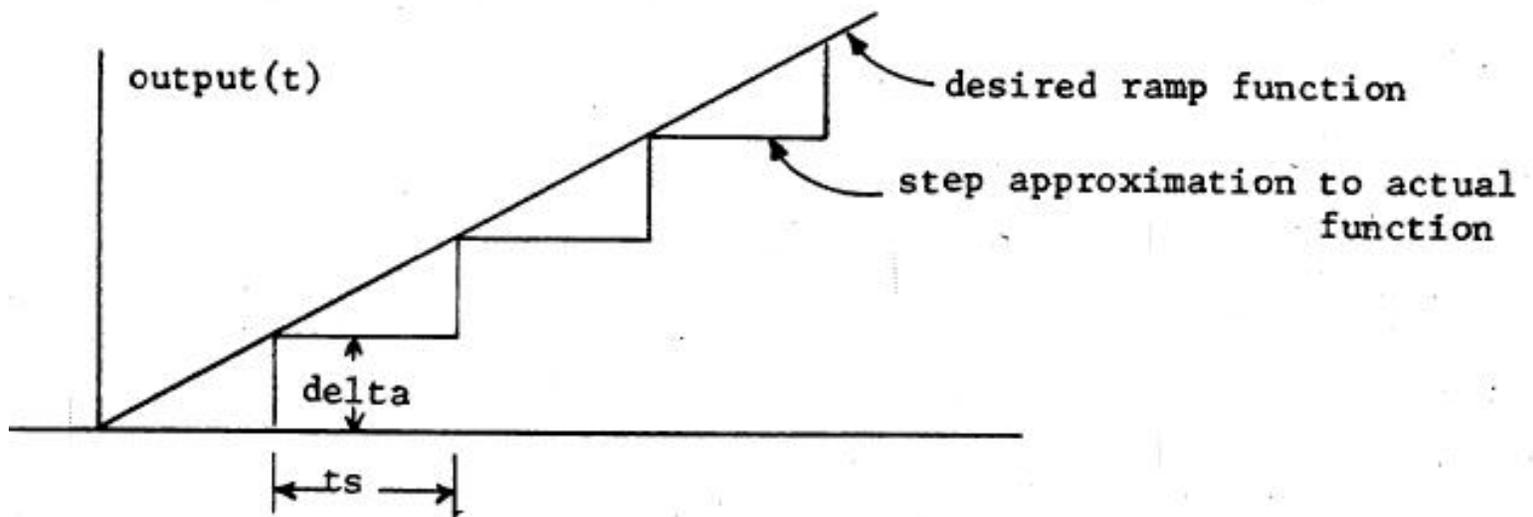


Fig. SG-1b. Graph of an example of a step approximation to a ramp function.

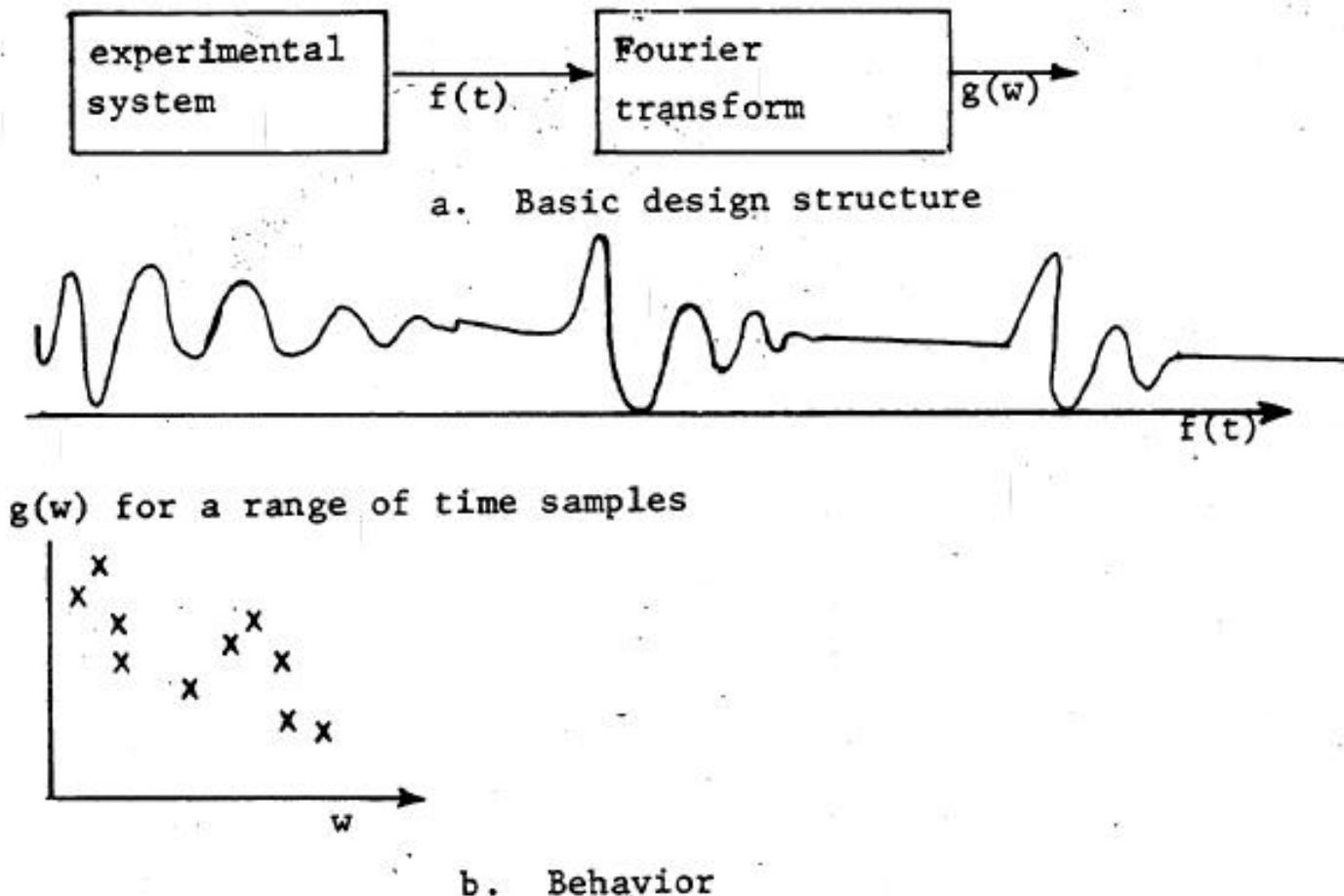
1. A fractional binary representation will be used to normalize the function so that the range 0 to  $(2^{16})-1$  represents the range 0 to 1.
2. The minimum period,  $T_{\min}$ , of the waveform equals  $(2^{16}/\delta) \cdot t_s$  seconds. The maximum

frequency,  $F$ , therefore equals  $1/T_{\min}$ .

3. The maximum fractional error is just the fractional step size;  $\text{error} = \Delta/2^{16}$ . The relationship between error and frequency is:  $f = \text{error}/t_s$  or  $\text{error} = f \cdot t_s$ .

4. The sampling time,  $t_s$ , generated by the clock is constrained to be no faster than the processes it controls, in this case roughly 1.5 microseconds (2 register transfers). Therefore the maximum frequency is  $f = \Delta/(2^{16} \cdot 1.5 \cdot 10^{-6})$  and the error is:  $\text{error} = f \cdot 1.5 \cdot 10^{-6}$ .

From the above characteristics, it can be seen that if extreme accuracy is



■ Fig. Time-11. RTM diagram of a system to compute fourier transform of a waveform.

functions for variable sweep time, automatic scaling, and taking statistical data. Provide for a synchronization input which triggers the scope to start the sweep. Also, provide for automatic triggering based on certain input signals.

### SAWTOOTH WAVEFORM (RAMP) GENERATOR

KEYWORDS: Real time, waveform generator

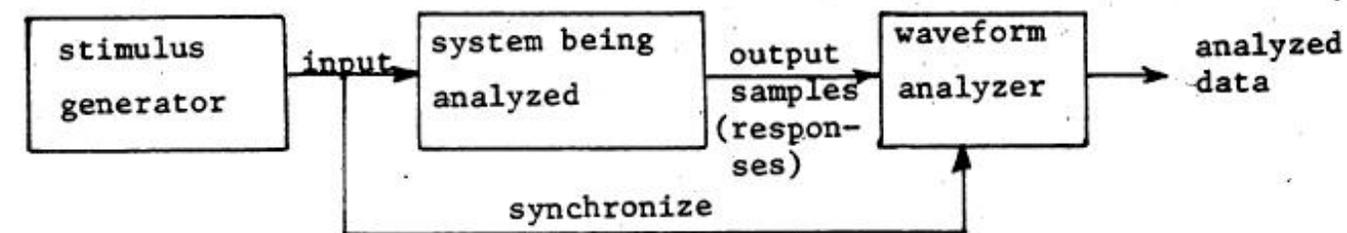
Figure SG-1a shows an example of a ramp function, output(t), and Figure SG- 1b shows the error which results from an approximation to the ramp function by a step function.

### PROBLEM STATEMENT

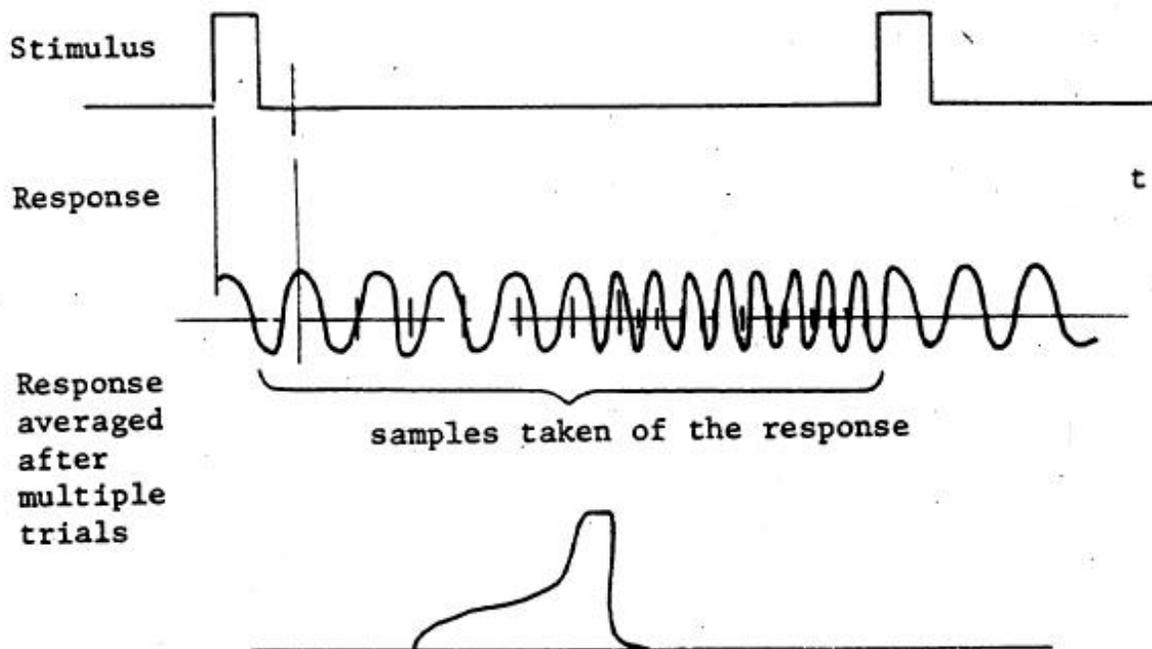
Design a sawtooth waveform (i.e., a ramp) generator with the peak amplitude for the function constant at one (i.e., normalized).

## SOLUTION

The characteristics of a single sawtooth waveform generator, shown in Figure SG-2 will be examined to introduce concepts common to many waveform generators. The generator is basically a register in a DMgpa which holds the value  $output(t)$ , to which an input increment (a slope),  $\delta$ , is added modulo  $2^{16}$ . The value of  $\delta$  (where  $1 \leq \delta < 2^{16}$ ) is added to output at its second intervals. The value for output is transferred to a T(digital-to-analog) at the sample intervals. The following characteristics can be formulated for the generator:



a. Basic design structure



b. Behavior graphs

Fig. Time-10. RTM diagram of a system for average response computation (stimulus-response system).

3. Design an RTM system to compute one of the common transforms: Fourier, fast Fourier, auto-correlation or cross-correlation.
4. Design a digital voltmeter (using a T(analog-to-digital)) which computes the maximum, minimum, average and peak to peak values, and the root mean square of the waveform. Note that for measuring a periodic function, a large number of samples must be taken to insure that the total error is no greater than the input quantization error.
5. Design a D(clock and calendar) which displays the time (in 1/100 second increments) and date. Two important principles are counting in mixed bases (i.e., 100,60,60,24,(28,29,30,31),12,-) and synchronizing independent processes, since counting and display occur nearly simultaneously. Assume that the system can be initialized with base values for seconds, minutes, hours,... at the occurrence of an

input signal. Also, assume that a master clock generates the time base for the D(clock and calendar) by providing a pulse at 10 microsecond intervals. All interface signals (inputs and outputs) should be given together. with any restrictions on the system.

6. Design a sampling oscilloscope using RTM's, which assumes a basic x-y plotting oscilloscope with z axis control and T(a-d) converters. The input function is sampled, held in memory, possibly operated on, and then displayed. Provide

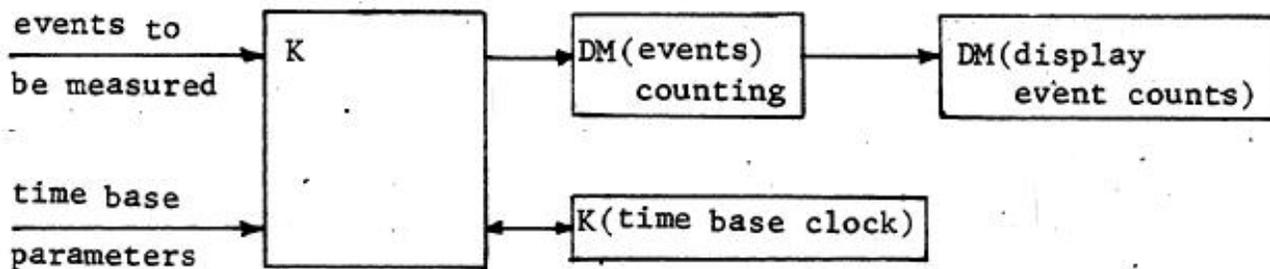


Fig. Time-7. RTM diagram of a Frequency/Event Per Unit Time/EPUT meter.

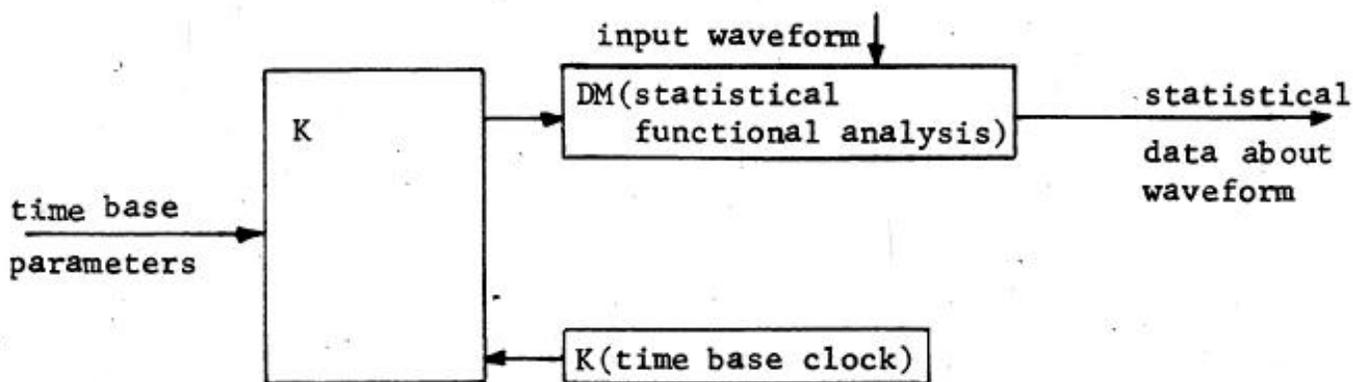
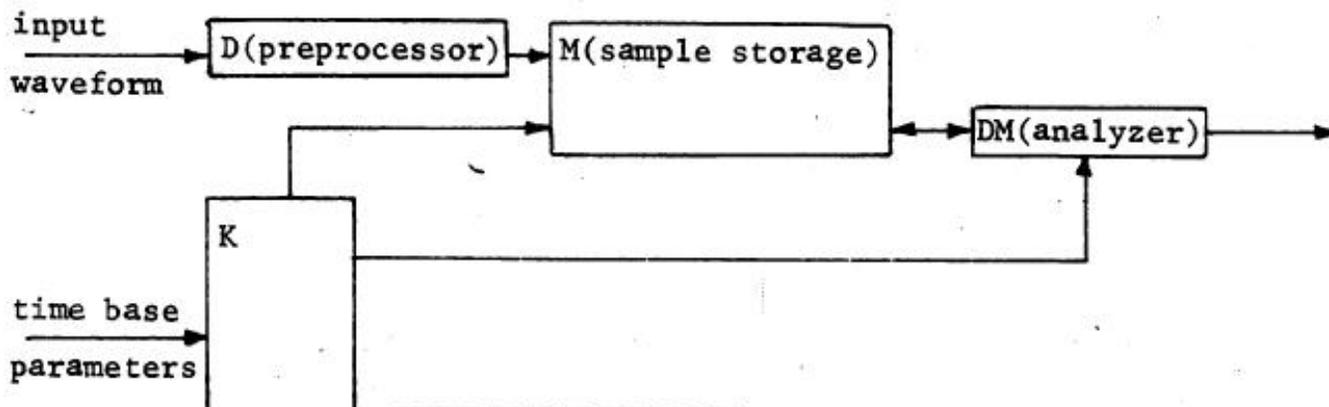


Fig. Time-8. RTM diagram of a system to perform waveform parameter measurement (e.g., mean, median, rms, etc).



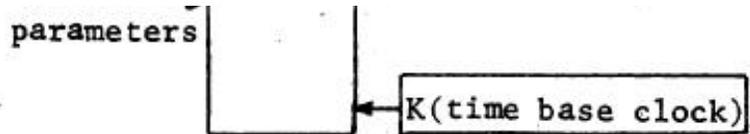


Fig. Time-9. RTM diagram of a system with waveform sample storage for post analysis.

3. Derivations of a quantized analog waveform, where the input signal indicates that a certain condition (event) has occurred.

The output from the waveform analyzer might be one (or more) of the following:

1. A count of the number of events that occurred over some unit time period. Such a system is called a frequency meter or Events Per Unit Time\EPUT meter. The basic structure is shown in Figure Time-7. A complete design problem will be presented in a later section.
2. Various statistical measures of the .waveform such as: maximum (peak), minimum, and average (mean) values, median, mode, standard deviation, root mean square, etc. The basic structure is shown in Figure Time-8.
3. A recording of the input waveform for later analysis. The design of Figure Time-9 shows that the input waveform samples are stored in a memory; then an analyses is performed on the data. One type of record which is kept is a histogram which records the number of occurrences of each sampled event type; statistical analysis could be performed on the histogram. A complete design for a histogram is presented in a later section.
4. An average response computation, which is an analysis of signals that are either periodic or of known time origin. The basic structure for such a system is shown in Figure Time-10a. A generator stimulates the system being analyzed by giving it an input at a known time. The system responds with an output waveform which is transferred to the waveform analyzer for the computation. In Figure Time-10b the system response output is shown as it might appear at the input to the waveform analyzer. The signal in the figure has a large amount of additive random noise - in fact, there is more noise than true signal. Taking a single sample of the waveform, therefore, gives little accurate information. Typically, the true signal might be 1/10 of the noise amplitude. By repeatedly evoking the system and adding each response to an accumulated sum of sample values, the noise can be averaged out. The effect is that the repeated sampling of each point along the waveform will drive the average value of the noise to zero. If the average noise value does go to zero, the average of the accumulated sum of samples will give a good approximation to the true signal value at the sample point.
5. A transform type analysis based on a vector of samples. Figure Time-11 shows the basic structure of an analyzer to compute Fourier transforms. The purpose of the system is to transform the time sampled results into some frequency domain (spectrum). Other transforms of this type include auto-correlation and cross-correlation which have similar design structures.

## PROBLEMS

I. Figure Time-2 shows various schemes for carrying information. Sketch functional diagrams for the parts of systems which might be used to transmit (encode) and receive (decode) the information expressed in those forms. Assume that various T's are available to produce and convert the signal forms. For example, one such transducer might take a time parameter as input and produce a sinusoidal output with this period.

2. Explore the design space of average response analyzers and show several alternatives with their performance characteristics. The systems should be capable of simultaneously sampling the input and displaying previous results.

3. The data operations macro which generates the output value requires a certain amount of time and this time must be considered when choosing the sampling time,  $t_s$ . If the sampling time is sufficiently small, the data operations macro may be re-activated before it has completed the processing initiated with the previous activate signal. This condition would cause erroneous operation.

4. If the data operations macro does not perform its processing in a constant amount of time, the output will appear at irregular intervals, rather than at the equally spaced intervals which are desired. This irregularity is called variable time jitter.

A design with the above modifications is shown in Figure Time-5. A K(manual evoke) is used to start the system and a switch in the control part loop allows the generator to be stopped. The variable time jitter is corrected by calculating an output value during a particular time delay, then outputting it at the beginning of the following time delay. This gives the constant spacing of the output values as desired even if the processing time is not constant. If one looks carefully at the loop, it is clear that a clock has been constructed. Replacing the delays by a clock, the design in Figure Time-6 is achieved.

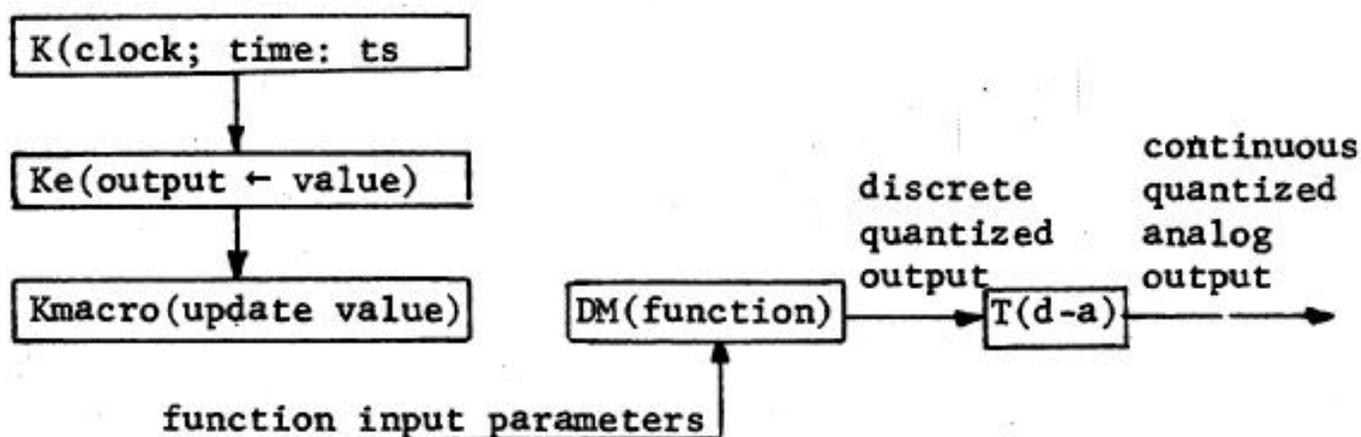


Fig. Time-6. RTM diagram of a waveform generator using K(clock) as a time base.

The problem of activating the data operations macro while it is still processing from the previous activate signal has been corrected in the design presented in Figure Time-5 via the K(parallel merge), but it still exists in the design of Figure Time-6.

## TIME FUNCTION ANALYSIS (WAVEFORM ANALYZERS)

Waveform analyzers take an input waveform,  $Input(t)$ , and analyze it by observing (sampling) the

waveform to extract information about certain defining properties of the waveform. This information is the output of the waveform analyzer. For the systems described here, the input waveform will either be encoded in digital form or may be readily encoded with a T(analog-to-digital). The types of input to the analyzer might be:

1. An analog voltage at a certain sampling time, which has been quantized to one of  $2^n$  levels (e.g., the output of the waveform generators described previously).
2. A digital signal, which indicates whether an analog waveform is above or below a given threshold (actually a special case of 1 above with 2 levels).

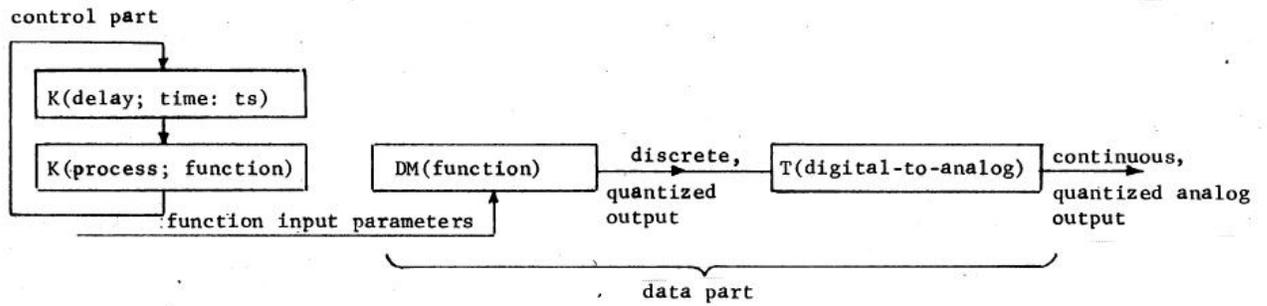
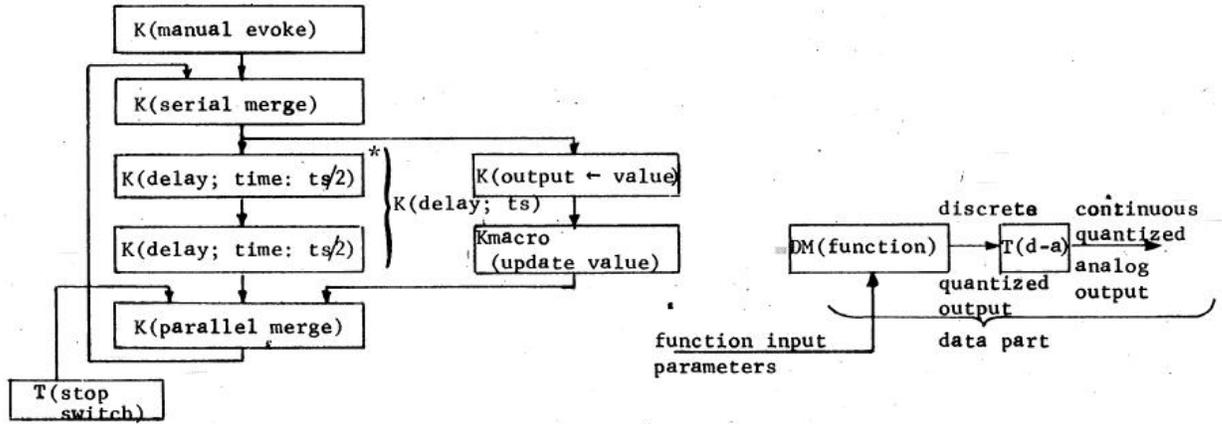


Fig. Time-4. RTM diagram of a waveform generator with a K(delay) time base (incomplete design).

193



\* alternatively may be programmable to give variable functions

Fig. Time-5. RTM diagram of a waveform generator with a K(delay) time base (improved design).

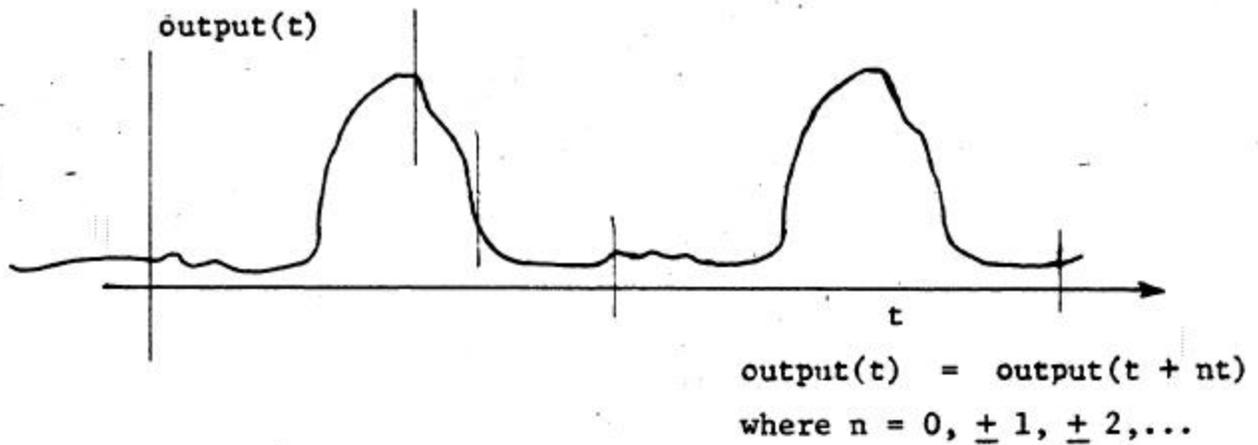


Fig. Time-3a. An example of a wave form function, Output(t), produced by a waveform generator.

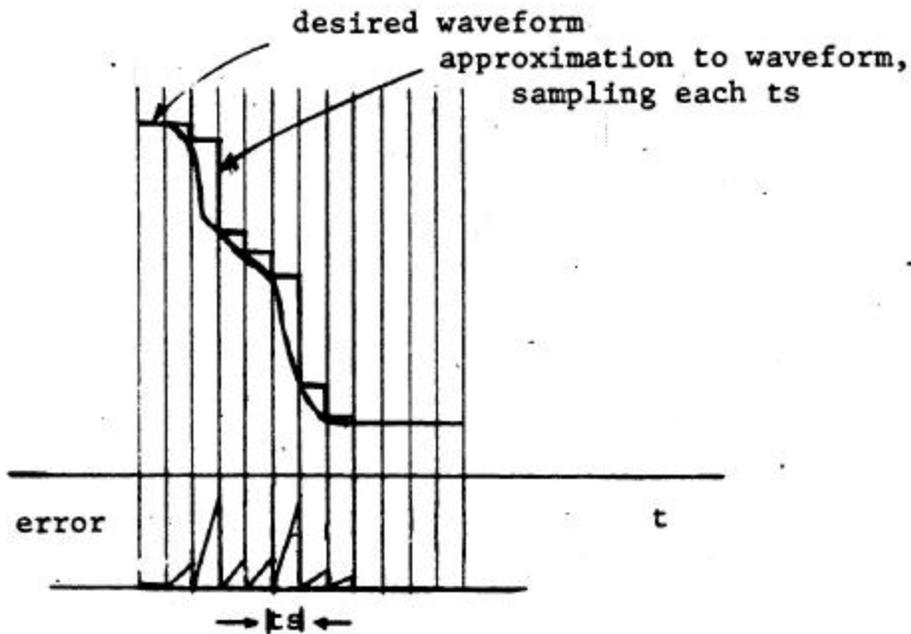


Fig. Time-3b. An example of discrete approximations to a continuous waveform.

[previous](#) | [contents](#) | [next](#)

The second type of scheme is characterized by the modulation of a carrier sine wave, as shown in Figures Time-2b, c, and e. In amplitude modulation (on off keying), as shown in Figure Time-2b, the carrier has an amplitude of 1 when a pulse is present and 0 when it is not. In the type of frequency modulation known as frequency shift keying, as shown in Figure Time-2c, a 0 is transmitted as a given frequency, and a 1 is transmitted as a second frequency, both with the same amplitude. In conventional frequency modulation (FM), as shown in Figure Time-2e, the instantaneous frequency is proportional to the value of the information being transmitted; that is, FM has a fixed carrier frequency and the value of the information transmitted is proportional to the frequency deviation from the fixed carrier frequency.

From these examples, it can be seen that while it is commonly recognized that communications systems transmit information in a continuous form, systems are also designed to transmit and receive data using digital (discrete) techniques.

## TIME FUNCTION SYNTHESIS (WAVEFORM GENERATORS)

Waveform generators produce as output functions of specified input parameters and time. In this book the dependence upon the input parameters is not usually expressed explicitly; the function is most often specified as output(t). An example of a periodic function, output(t), is shown in Figure Time-3a. Although the function is shown to be continuous, RTM and other digital system waveform generators are constrained to approximate the true function by a series of discrete samples. Such an approximation, which produces a sample every  $t_s$  seconds, is shown in Figure Time-3b. Approximation of a continuous function by a set of discrete samples introduces two types of errors: discretization or quantization errors, which arise from the use of only a finite number of samples (e.g.,  $2^8 \sim 2^{12}$ ) in the approximation; and sampling errors, which are introduced by the fact that between any pair of samples the function is held constant, while the true function may exhibit a non-linear behavior between the points. (Linearization between samples would be a better approximation, for example.) The error can be expressed in various ways: maximum value, average value, root mean square (rms), etc. The choice of sampling time,  $t_s$ , is dependent on: the specific waveform and its period  $T$ , the 'resolution (number of levels) of the discrete output (e.g.,  $2^w$  where  $w$  is the number of output bits from some digital device), and the maximum allowable error.

A waveform generator has the simple structure shown in Figure Time-4. A K(delay) generates the time base by producing a  $t_s$  seconds delay, after which a data operations macro is called which generates output(t) based on the input parameters and time. The output of the data operations macro can be used directly, (parallel by word pulse code modulation); or, if necessary, a T(digital-to- analog) converts the digital outputs into voltage steps that approximate the desired waveform (amplitude encoding). The design shown in Figure Time-4 must be modified, though, in order to provide a more complete and correct design. To that end, the following changes and considerations are necessary:

1. An activate signal is needed so that at power on, or by manual control, the generator can be

started. Another switch is needed so that the generator may be stopped.

2. Two K(delay)'s are needed in series to produce the desired delay because a K(delay) cannot be re-activated a short time after being active. The recovery of a delay is nominally 20% of the delay. Alternatively, a K(programmable delay) as described in Chapter 3 could be used.

state. This last analyzer is often called an Events Per Unit Time\EPUT meter though it is usually called a frequency meter when analyzing continuous waveforms (e.g., sinusoidal).

## MODULATION: CARRYING' INFORMATION IN A TIME-DEPENDENT FORM

In digital systems, information is usually transmitted in a direct parallel form; that is, there is a direct physical link for each bit of a data word, and each link has to carry only a 1 or a 0 during a word transfer. However, digital information may also be transmitted' serially by using one of the various types of modulation as shown in Figure Time-2. It will be seen that the modulation schemes are of. two basic types. The first type, exemplified in Figures Time-2a and 2d, is characterized by intermittent steplike transitions of the signal between two voltage values. In pulse code modulation, shown in Figure Time-2a, the high voltage values correspond to 1's and the low values correspond to 0's. In pulse duration modulation, shown in Figure Time-2d, the information is encoded in the length of the pulse.

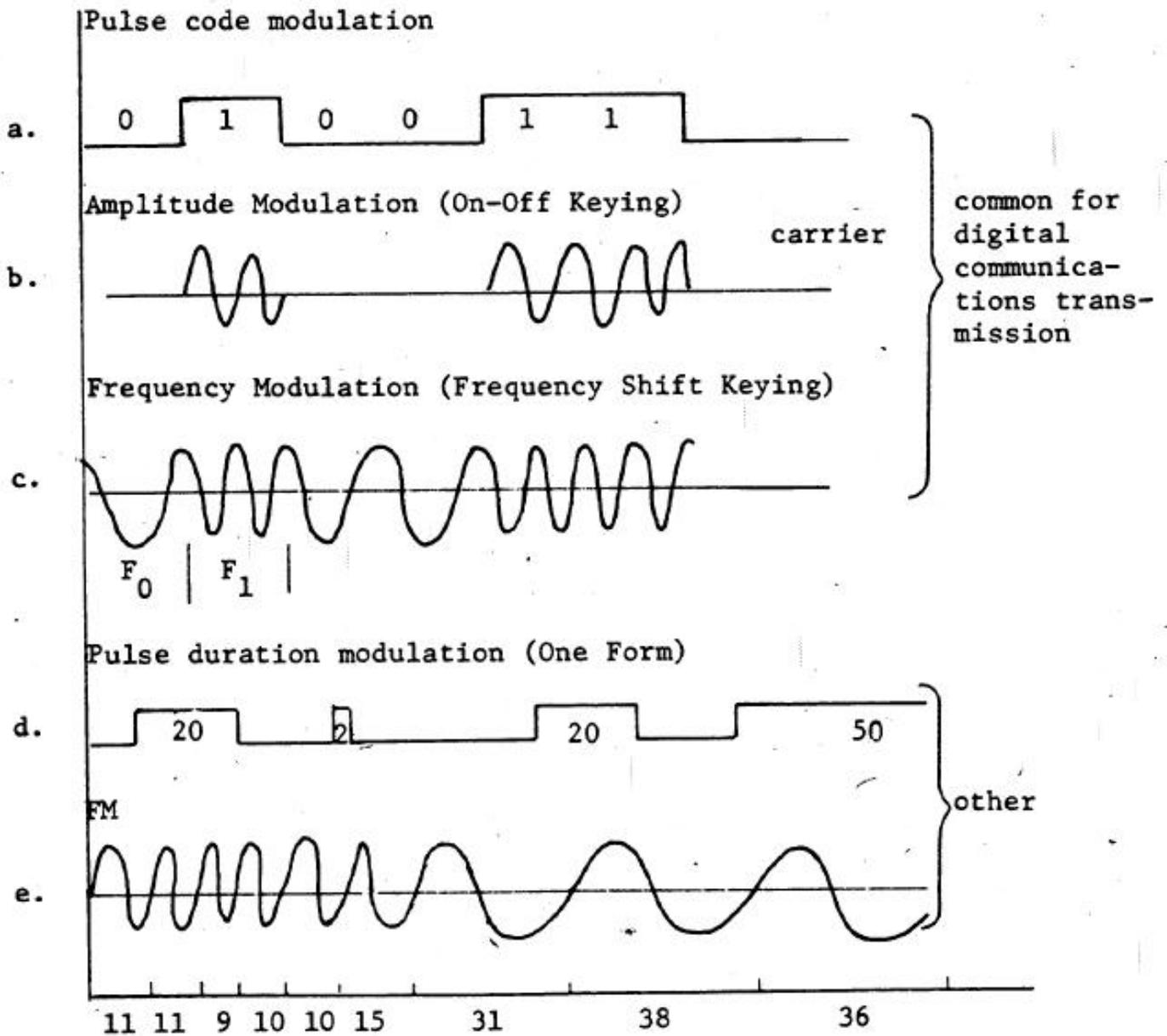
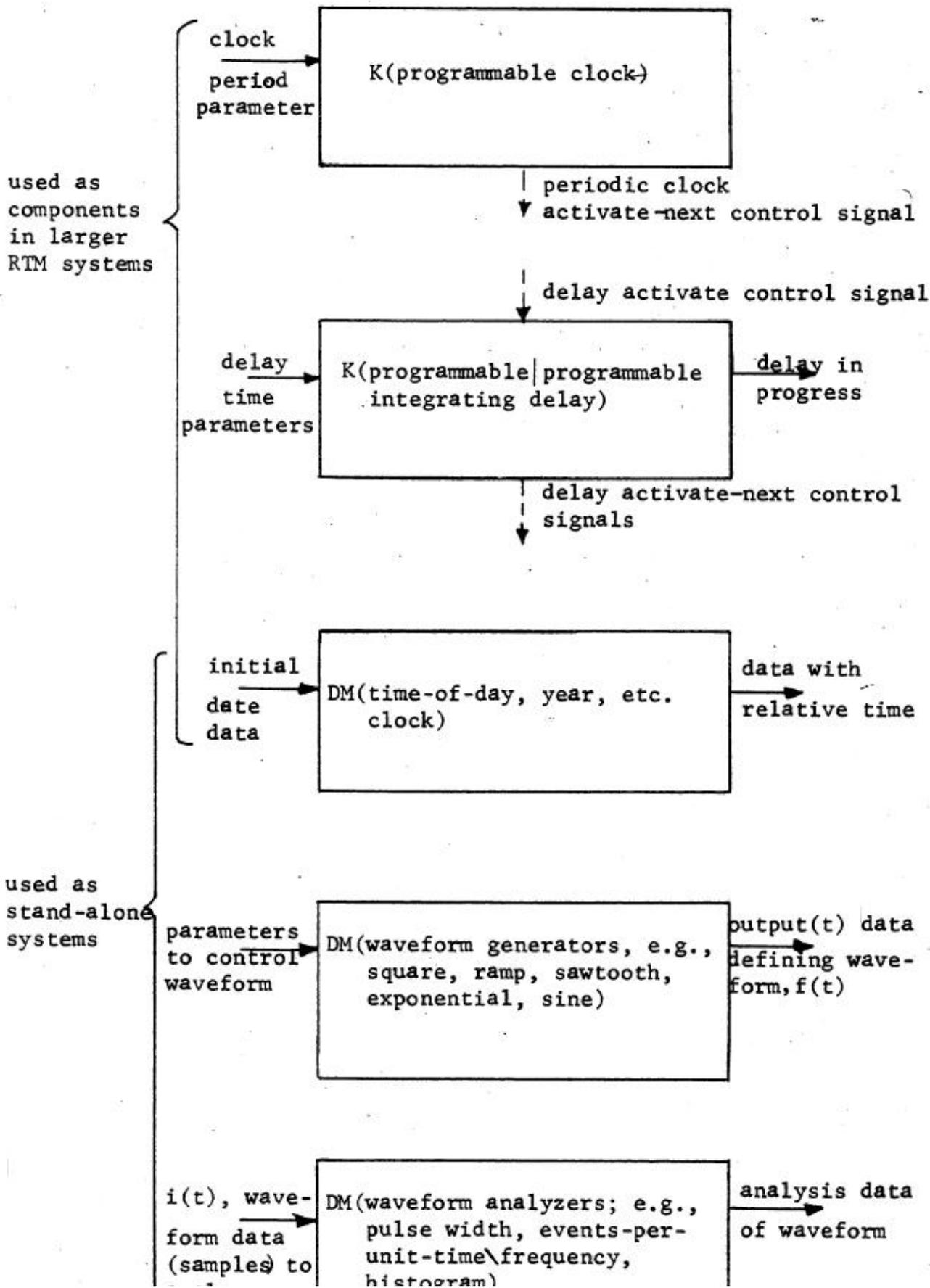


Fig. Time-2. Various modulation schemes for transmitting digital information.



form data  
(samples) to  
analyze

unit-time\frequency,  
histogram)

Fig. Time-1. RTM diagrams of various time-based components and systems.

When the result of a subtraction is negative, the 1000v10 is added back into the binary register and no increment is made to the BCD registers. The second, third and fourth calls on the subroutine cause the subtraction of 100v10, 10v10 and 1 respectively from the binary numbers and the addition of 100v16, 10v16 and 1 into the BCD register. Each subroutine call is complete when the result of a subtraction is a negative number.

## ADDITIONAL PROBLEMS

1. Develop RTM systems to perform BCD complementation, add 1, x10, /10, multiplication, division, single and double precision.
2. Perform an analysis which compares the cost/speed of the arithmetic flowcharts which operate on BCD numbers without conversion and those routines which perform the equivalent operations with conversion from BCD to binary on input and binary to BCD on output.
3. Design a DMgpa which would facilitate BCD arithmetic. Carry out problem 1 above, based on this design.

## TIME-BASED SYSTEMS: CLOCKS, TIMERS

### WAVEFORM GENERATOR, AND WAVEFORM ANALYZER

## INTRODUCTION

This section presents systems which synthesize (generate) and analyze time dependent functions. Carrying information in a time-dependent function is called modulation, converting information into such a function is called synthesis (encoding), and extracting information from such a function is called analysis (decoding). The primitive clock, delay, and integrating delay, classified as controls in Chapter 2, form the basis for time measurement, permitting encoding and decoding functions of time. The types of systems to be considered are shown in Figure Time-i. The first three (and variations) are, fundamentally, components for use in larger systems, whereas the last two might stand alone as independent systems.

The first two systems, K(clock) and K(delay), with and without variable time parameters, were presented as primitive RTM components in Chapter 2 and as design problems in Chapter 3.

The third system, a K(clock and calendar), maintains a clock and calendar 'relative to a base point. The time is made available as output data either on a continuous basis or on demand. Such a design problem is presented in a following section and such a clock is interfaced to a digital computer in Chapter 6.

The fourth type of system, the waveform synthesizer (generator), is usually a combination of a clock (or timer) and a Data operation part which computes an output value at times  $t, t + \Delta t, \dots$ . The output value, which is a function of any input parameters specified and the time, is either an analog (continuous) or a digital (discrete) waveform. The square wave generator presented in Chapter 3 is a simple example of a discrete waveform generator. Additional, more complex waveform generators will be presented in a following section.

The remaining system of interest, the D(waveform analyzer), is an analytic device which takes information from an input waveform,  $1(t)$ , and decodes it in an attempt to reduce the amount of information needed to represent the waveform; the reduced set of data is the output of the analyzer. For example, analyzers might measure the instantaneous amplitude of a waveform, waveform duration, or the number of occurrences per unit time that a signal has remained in a given

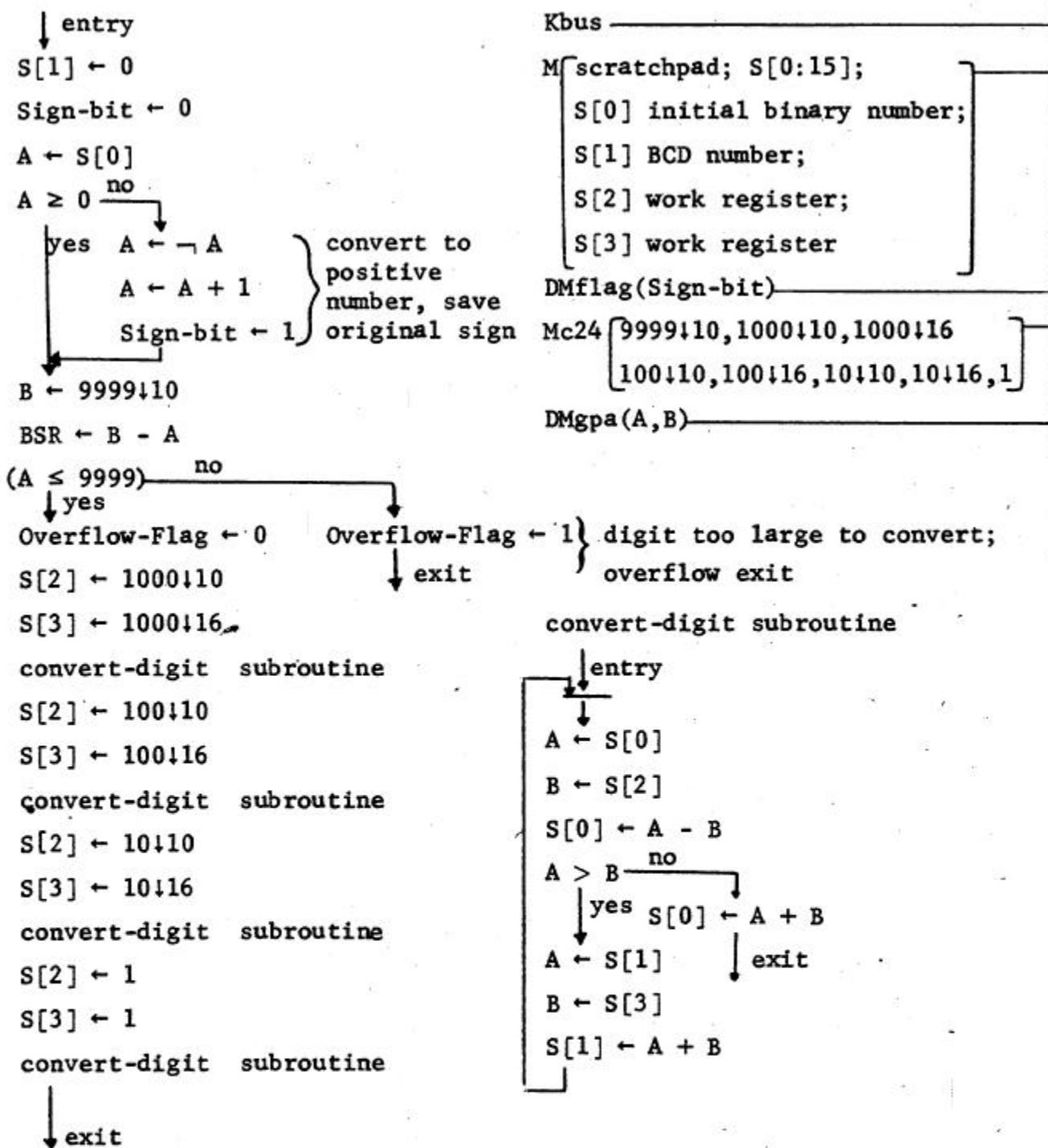


Fig. BCD-10. RTM diagram of a system to perform single word two's complement binary to BCD conversion by repeated subtraction.

[previous](#) | [contents](#) | [next](#)



[previous](#) | [contents](#) | [next](#)

$$\begin{array}{r}
 1 \quad - 1 \\
 10 \quad ] \quad - 2 \\
 10 \quad ] \\
 100 \quad ] \\
 100 \quad ] \quad - 4 \\
 100 \quad ] \\
 + 100 \quad ] \\
 \hline
 421110
 \end{array}$$

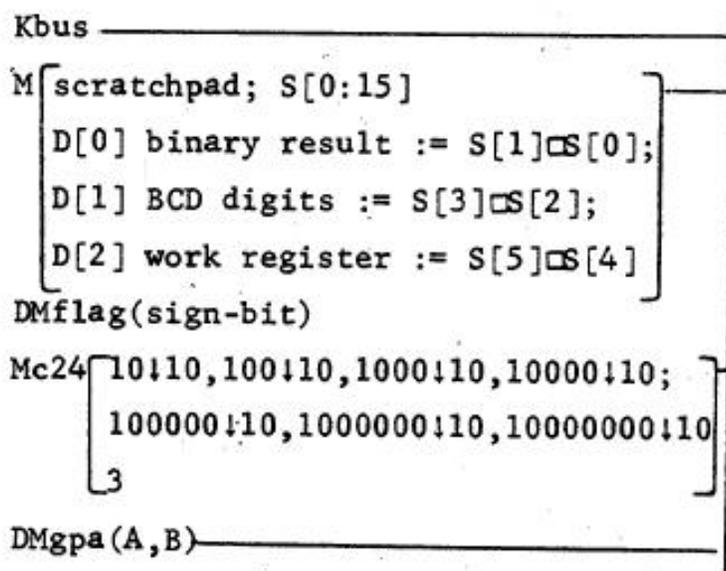
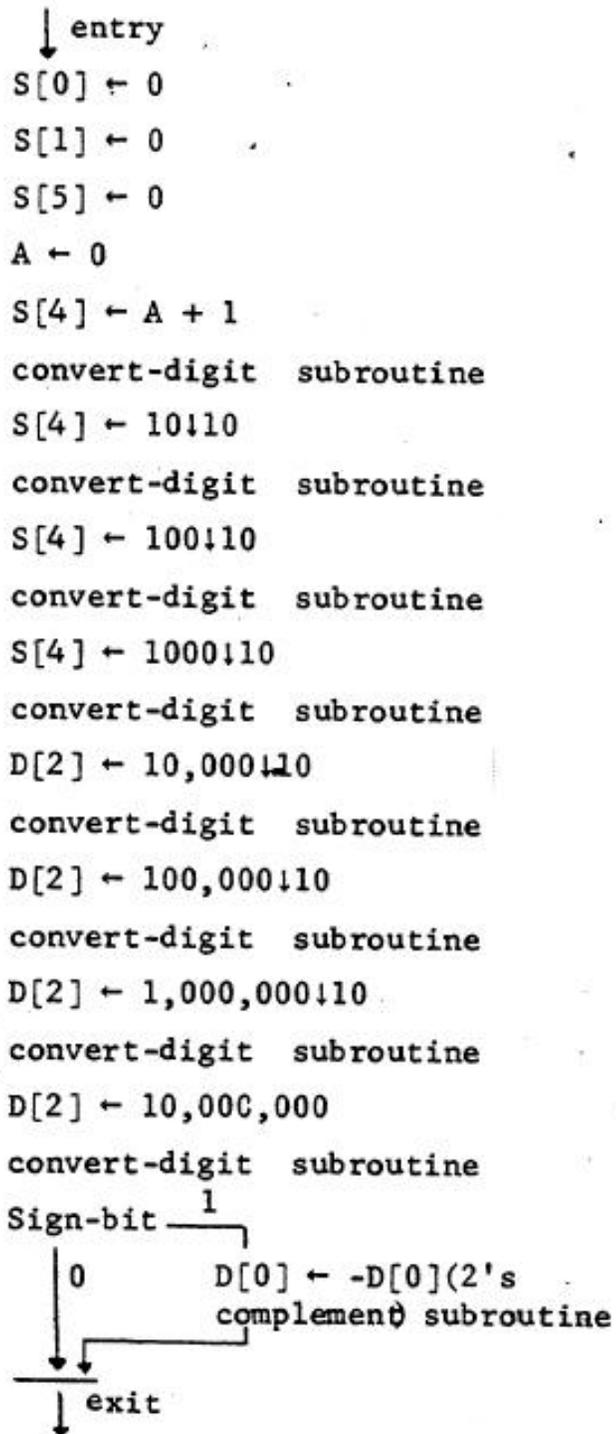
Convert 421+10 to binary.

Fig. BCD-7. An example of double precision BCD to binary conversion by repeated addition of powers of ten.

$  \begin{array}{r}  901 \\  10 \overline{) 9018} \\  \underline{90} \\  018 \\  \underline{10} \\  8  \end{array}  $	$  \begin{array}{r}  90 \\  10 \overline{) 901} \\  \underline{90} \\  01 \\  \underline{0} \\  1  \end{array}  $	$  \begin{array}{r}  9 \\  10 \overline{) 90} \\  \underline{90} \\  0  \end{array}  $	$  \begin{array}{r}  0 \\  10 \overline{) 9} \\  \underline{0} \\  9  \end{array}  $
first digit	second digit	third digit	fourth digit

Fig. BCD-9. An example of two's complement binary to BCD conversion by repeated division by ten.

[previous](#) | [contents](#) | [next](#)



convert-digit subroutine D[0] ← D[0] + D[2] \* D[1]<3:0>

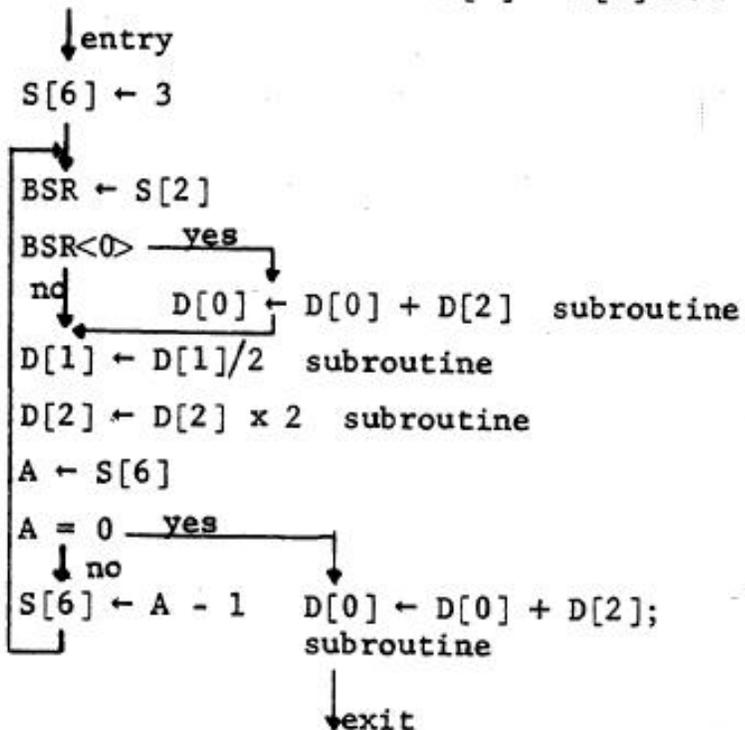


Fig. BCD-6. RTM diagram of a system to perform double precision BCD to two's complement binary conversion in straight line form.

Solution 2

Figure BCD-10 illustrates a straight-line single word two's complement binary to BCD conversion

flowchart. This scheme employs successive subtraction of powers of ten to form the BCD number. The sign of the binary number is once again saved in the Boolean register, Sign-bit; negative binary numbers are complemented and the check is made for binary numbers greater than  $9999_{10}$ . The first call on the subroutine successively subtracts  $1000_{10}$  from the binary numbers and adds  $1000_{16}$  to the BCD register for each  $1000_{10}$  subtracted.

```

↓ entry
D[0] ← 0
For I ← 1 step 1 until 8 do (multiply by 10, add next-digit subroutine)
Sign-bit ← 1
    ↓ 0
    D[0] ← -D[0] (2's complement subroutine)
    ↓ 1
    exit
    
```

multiply by 10, add next digit subroutine

```

↓ entry
D[1] ← D[1] * 16 {rotate} subroutine
D[2] ← D[1] <3:0> subroutine
D[0] ← D[0] * 10 subroutine
D[0] ← D[0] + D[2] subroutine
↓ exit
    
```

} next-digit

} D[0] ← D[0] \* 10 + next-digit

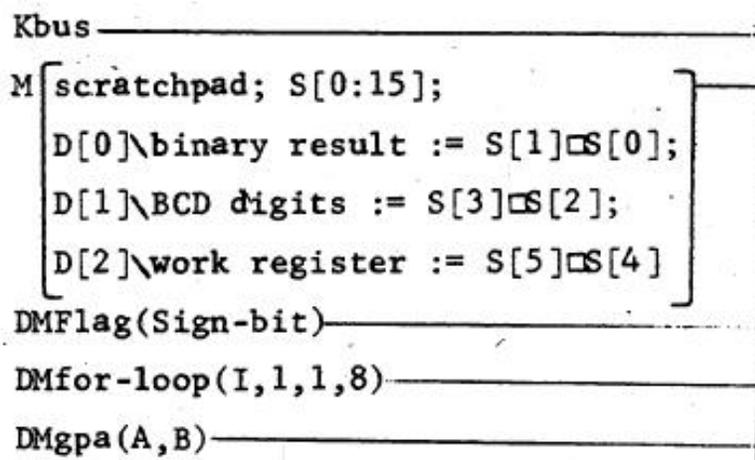


Fig. BCD-5. RTM diagram of a system to perform double precision BCD to two's complement binary conversion using a ((for-loop).

9999v10, since this is the largest four-digit BCD number than can be correctly stored. If the binary number is out of the proper range, an Overflow-flag Boolean is set to one.

The conversion process produces the digits by integer division by 10 of the binary number. The remainder at each step is a digit of the BCD number; the digits are produced in order from least

significant. The process is illustrated by the example in Fig. BCD-9 (Fig. BCD-9 is on the same page as Fig. BCD-7). Each digit is added into a partial sum as it is produced.



↓ exit

Fig. BCD-4. RTM diagram of a system to perform BCD to two's complement binary conversion.

$$S[0]<15:0>:=((((S[1]<3>)*10+S[1]<2>)*10)+S[1]<1>*10)+S[1]<0>)$$

## BINARY CODED DECIMAL TO BINARY CONVERSION

### Solution 1

An algorithm for converting a four-digit BCD number into standard two's complement binary format is shown in Figure BCD-4. The BCD number is held in the base 16 register,  $S[1]<3:0>v16$  with the sign bit held in a DMflag (Sign-bit). The resulting binary number is stored in the register  $S[0]<15:0>v2$  and is computed as follows:

$S[0]$  is 2's complemented if the sign bit of the BCD number was one (negative). The main control part calls a subroutine four times to convert each of the digits and forms the two's complement if necessary. The subroutine, on each call, multiplies the partially converted binary number by ten and adds in the next least significant BCD digit; this process is Homer's method for polynomial evaluation. The next digit is found by shifting the four most significant bits into the least significant four bits of the A register and masking the other digits out. The partially converted binary number is multiplied by ten by repeated shifting and addition. The last step of the subroutine adds the new digit into the partial sum.

### Solution 2

An algorithm for converting an eight-digit BCD number into a double word two's complement format is shown in Figure BCD-5. The eight BCD digits are held in the double word register  $D[1]$  which is comprised of the register pair  $S[3] \parallel S[2]$ ; the binary number will be held in the double word register  $D[0]$  which is formed by the register pair  $S[1] \parallel S[0]$ . The same subroutine as was developed in solution 1 is used to convert the eight digits; in this implementation, though, a K(for-loop) module is used to control the conversion routine. The conversion routine is divided into four subroutines: (1) circular shift to find next BCD digit; (2) extract the BCD digit; (3) multiply the partial sum by ten; and (4) add the BCD digit into the partial sum. The main routine also calls a subroutine to complement the binary number if the BCD number was negative.

### Solution 3

Figure BCD-6 shows a straight-line method for converting eight BCD digits to two's complement binary. The BCD number is held in a double-word register  $D[1]<7:0>v16$  and the binary result is stored in the double-word register  $D[0]<31:0>v2$ . The scheme adds  $10^l$  to the accumulated sum in  $D[0]$ ,  $D[1]<|>v10$  times for  $l=0,1,\dots,7$ . A subroutine is called by the main routine to complement the binary number if the BCD number was of negative sign. Figure BCD-7 gives an example of this scheme.

## BINARY TO BINARY CODED DECIMAL CONVERSION

### Solution 1

A routine to convert a single word two's complement binary number to BCD is given in Figure BCD-8. Since the BCD number is stored in sign-magnitude form, the sign of the binary number is saved in the DMflag Sign-bit and negative binary numbers are complemented so that all conversion is done on positive integers. The magnitude of the binary number must be less than or equal to

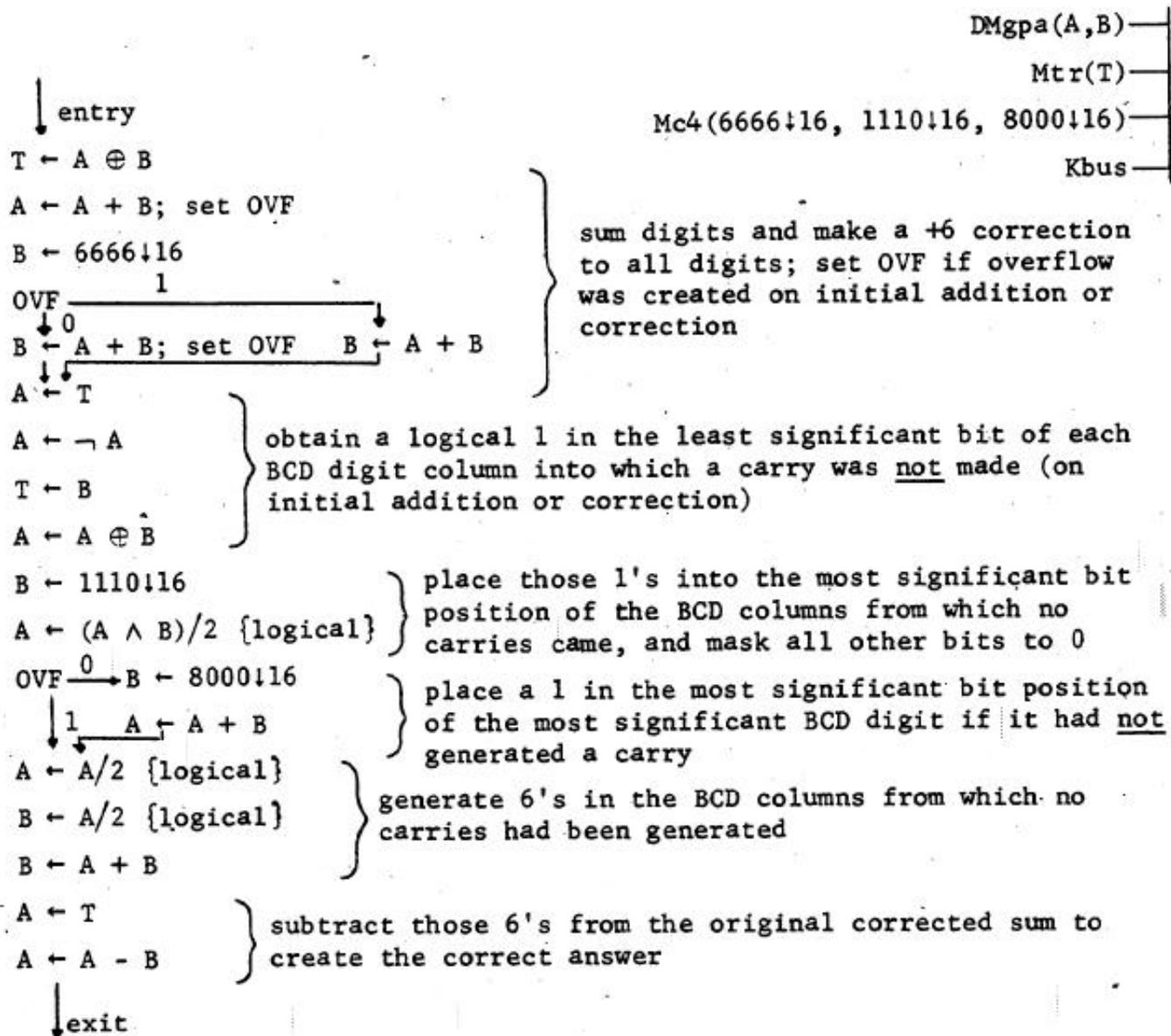


Fig. BCD-3. RTM diagram of a system designed by Robert Chen to perform BCD addition using binary operations; BCD numbers stored four digits per register.

Solution 2 (Robert Chen algorithm)

An algorithm for adding two four-digit BCD numbers is shown in Figure BCD

3. The numbers are held in the registers of a DMgpa; a temporary register, T, is also used. In this scheme the six correction is made to all sum digits, then a mask of sixes is created from those sum digits in which

no carry was produced. The sixes are then subtracted from the digits which required no correction.

This latter algorithm by Robert Chen still appears costly and time consuming compared to binary addition. Signed addition, subtraction, multiplication, and division will no doubt be more difficult. The alternative of converting from BCD to binary to perform the arithmetic operations and reconvertng from binary to BCO for output will now be considered.

↓ entry

A ← S[0]

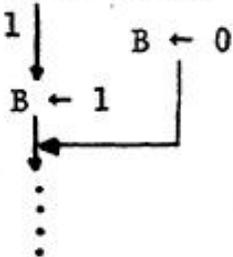
B ← T[0]

A ← A + B

B ← 1111 1111 1111 0110↓2

S[0] ← A + B; set OVF

OVF<sup>0</sup> → S[0] ← A - B



A ← S[N-1]

A ← A + B

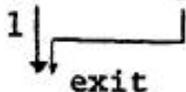
B ← T[N-1]

A ← A + B

B ← 1111 1111 1111 0110↓2

S[N-1] ← A + B; set OVF

OVF<sup>0</sup> → S[N-1] ← A - B



add least

significant BCD  
digits

add correction

to sum

subtract correction

if no carry was generated,  
(i.e., sum was ≤ 4 before  
correction) set

BCD carry

add carry from  
previous sum

add most

significant BCD digits

add correction

to sum

subtract correction

if no carry was generated

addend  
(sum)

S[0] First BCD digit  
⋮  
S[N-1] Nth BCD digit

augend

T[0] First BCD digit  
⋮  
T[N-1] Nth BCD digit

Fig. BCD-2. Control part of an RTM system to perform BCD addition using binary operations; BCD numbers stored one digit per word in a vector.

[previous](#) | [contents](#) | [next](#)

instead of modulo 10 as in proper decimal addition. To achieve the correct decimal result, a six must be added to each BCD digit of the sum which represents a decimal sum greater than nine. That is, sums in the range 10 to 15 (binary) should produce sums in the range 0 to 5 (decimal); addition of a six to each sum digit in the result will produce the desired effect. A carry into the next BCD sum digit must also be generated in the above case. For sums in the range 16 to 18 (binary) a carry has already been propagated to the next digit sum, but the six correction must still be added to adjust the BCD sum digits 0, 1, 2 to 6, 7, 8 respectively. If the sum of any pair of BCD digits is less than or equal to nine, no correction is necessary. Fig. BCD-1b summarizes the correction procedure for achieving proper BCD addition using binary operations.

BCD bits	<15:12>	<11:8>	<7:4>	<3:0>
A	4	0	9	9
B	5	3	9	1
Sum	9	4 <sup>1</sup>	2 (18)	10
Correction			6	6
Correct Sum	9	4	9 <sup>1</sup>	0 (16)

} digits shown are the base 10 equivalent of the 4-bit binary code

a. Example of addition of BCD numbers using binary arithmetic.

Sum \ S \ A+B	Carry to next digit	Correction to digit
$0 \leq S \leq 9$	no	no
$10 \leq S \leq 15$	yes (occurs when add 6)	yes, add 6
$16 \leq S \leq 18$	yes (already occurred)	yes, add 6

b. Rules for forming correct BCD sum using binary addition.

Fig. BCD-1. Illustration of BCD addition.

Solution 1 (S ← S+T)

A simple, direct method of BCD addition is illustrated in Figure BCD-2. Two vectors of registers, S[0:n-

1] and  $T[0:n-1]$ , hold the  $N$  digits of the two numbers to be coded; the  $j$ th digit is stored in bits  $\langle 3:0 \rangle$  of the  $j$ th register in the vector. The algorithm adds the first two digits ( $S[0]$  and  $T[0]$ ) together and then adds in a correction number,  $FFF6_{16}$ . If a carry is generated, it is propagated to the `BusOVERFLOW` and the sum is correct; otherwise the correction was unnecessary and it is removed. The carry is added to the sum of the next two digits. This process is repeated for each pair of digits ( $S[j]$  and  $T[j]$ ,  $j=0,1,..n-1$ ) to form the complete sum, which is stored in the  $S$  vector as it is generated.

## ADDITIONAL PROBLEMS

1. Implement higher precision floating point operations, e.g., a double word with an eight bit exponent and twenty-three bit mantissa.
2. Consider and implement a floating-point scheme in which the mantissa part is an integer instead. What are the advantages or disadvantages of this scheme?

## BINARY CODED DECIMAL (8421 CODE) ARITHMETIC AND CONVERSION'

**KEYWORDS:** Subroutine, overflow, carry, arithmetic, BCD; straight-line, loop, data-representation

Binary coded decimal (BCD) arithmetic will be discussed because it is a necessary adjunct to the standard binary system. BCD is often used outside the RTM system for representing data in a form convenient for human handling. In these examples each decimal digit is represented using four bits in the standard binary encoding, hence the name 8421 code. The sixteen bits of an RTM register will hold four BCD digits, but with no provision for a sign-bit. A Boolean register, Sign-bit, will be used to store the sign of the BCD number; a 0 will represent a positive sign and a 1 will represent a negative sign. Thus, for example, +9015 would be represented in binary coded decimal as 0 1001 0000 0001 0101.

Each BOD character is a four bit pattern. A simple notational extension of decimal digits is used to refer to each of the 16 possible patterns, called hexadecimal digits:

0 := 0000	4 := 0100	8 := 1000	C := 1100
1 := 0001	5 := 0101	9 := 1001	D := 1101
2 := 0010	6 := 0110	A := 1010	E := 1110
3 := 0011	7 := 0111	B := 1011	F := 1111

For example, 1D3Fv16 is the 16 bit word 0001 1101 0011 1111.

## PROBLEM STATEMENT

Design an RTM system to perform addition of two positive numbers expressed in binary coded decimal.

## DESIGN CONSIDERATIONS

These are two possible strategies for operating on numbers expressed in BCD with RTM's: operate on the numbers in BOD form inside the system; or convert the BCD numbers to standard binary form on input, operate on the binary numbers, and reconvert the binary numbers to BCD form on output. The systems described below include systems which perform addition directly on the BCD numbers and systems for BCD-to-binary and binary-to-BCD conversions.

## BCD ADDITION

The mechanics of BCD addition using binary registers will be briefly described to provide background for understanding the systems presented below. An example of BCD addition using binary registers is illustrated in Fig. BCD-1a. First, binary addition is performed producing  $A+B$ . This operation produces some correct BCD digit results; however, corrections must be made to some of the other digit sums to produce the correct result in BCD. Since binary addition has been performed, the addition for each pair of BCD digits has been modulo 16

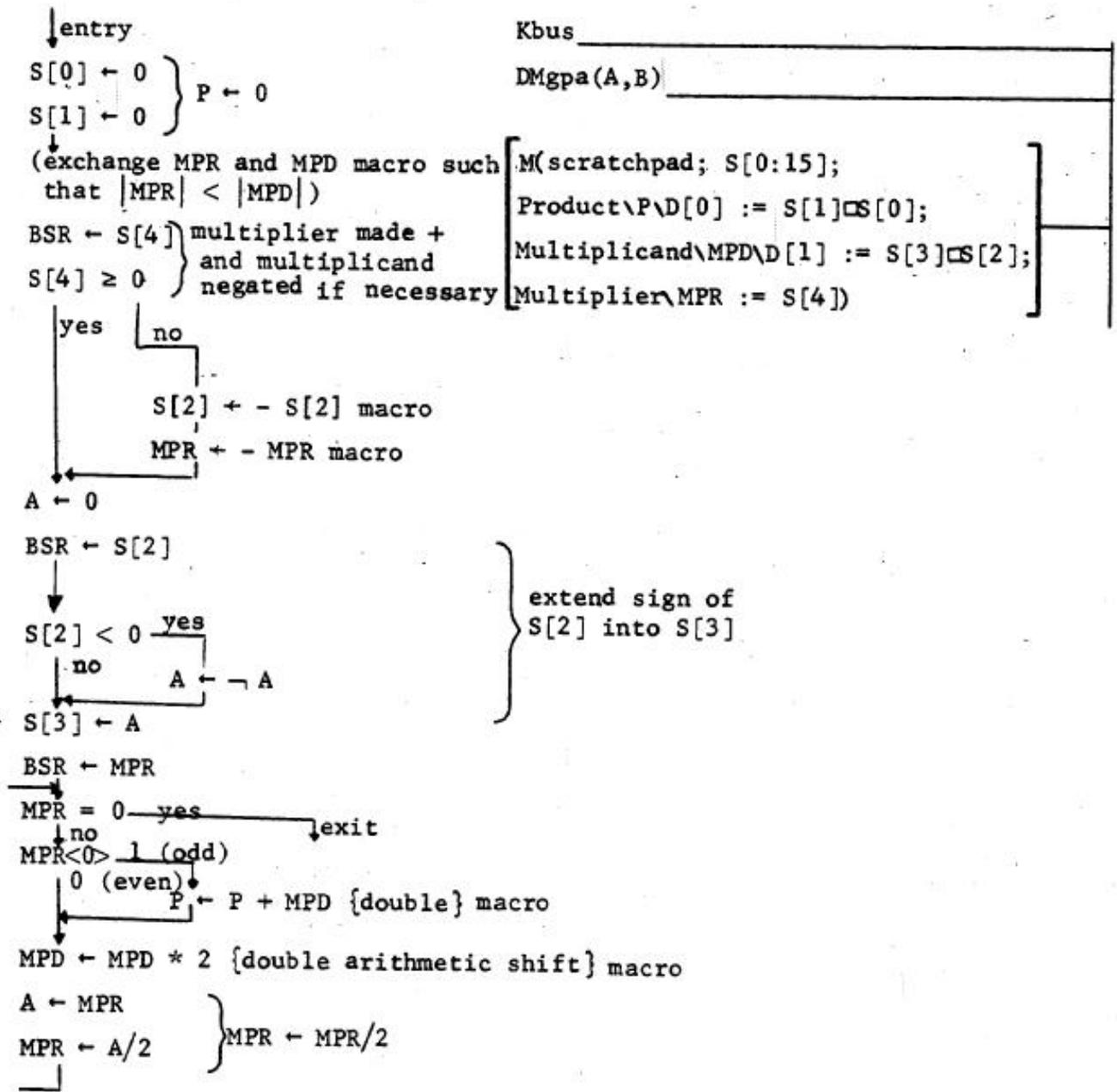


Fig. DATA-9. RTM subprocess for multiplication of sixteen-bit, two's complement numbers(Russian Peasant's Algorithm).

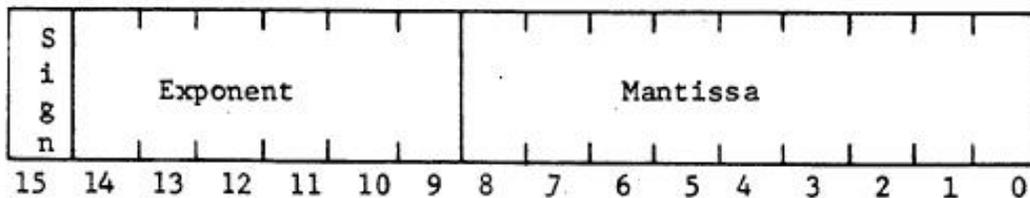


Fig. DATA-10 Sixteen-bit floating point data format

**Fig. DATA-10. Sixteen-bit, floating point data format.**

176

[previous](#) | [contents](#) | [next](#)

subroutine which implements the Russian Peasant's Algorithm will be presented here (see Figure DATA-9). Two sixteen-bit, two's complement numbers, initially held in registers S[2] and S[4], are multiplied together and the result placed in the double register D[0] defined as S[1] || S[0]. This subroutine requires double-word addition and left-shift operations. The reader should note the high overhead time for setting up the multiplication. Four steps are required:

initialization of P[D[0]]; exchange of multiplier and multiplicand (so that  $|MPR| < |MPD|$ ) to improve the performance of the algorithm; adjusting the sign (if necessary) of the multiplier and multiplicand so that the sign of the multiplier is positive; and extending the sign of the multiplicand into S(3).

## SCIENTIFIC (FLOATING POINT) NOTATION

**KEYWORDS:** Scientific notation, floating point, exponent, fraction, mantissa

A common representation used to express numeric data is the scientific (floating point) notation. Numbers in this form are written as  $d \cdot 2^j$  (for base two), where  $d$  is a binary fraction (mantissa) multiplied by  $2^j$ , where  $j$  is referred to as the exponent. For example:  $.1011 \cdot 2^6 = 101100.$ ,  $.0110 \cdot 2^{-3} = .0000110$ . Floating point notation is most often used for numbers of a fixed precision, but with a widely varying exponent.

One possible format for floating point numbers using a sixteen bit word is shown in Figure DATA-10. The exponent, R<14:9> is encoded in so-called excess 31 notation, that is, the true exponent of the number is 31 less than the number stored in the word; e.g., -31 is encoded as 0, -30 is encoded as 1, ..., 0 as 31, ..., and +32 is encoded as +63. This method avoids using a sign bit for the exponent. The mantissa, R<8:0>, is stored as an unsigned binary fraction with the binary point assumed to be immediately to the left of R<8>. The sign bit, R<15>, stores the sign of the mantissa. This format gives a range of numbers from  $-.000\ 000\ 001 \cdot 2^{-31}$  to  $.111\ 111\ 111 \cdot 2^{32}$  whereas a sixteen-bit, two's complement notation gives the range  $-(2^{15})$  to  $(2^{15})-1$ .

In addition to the common arithmetic operations (addition, subtraction, multiplication and division), one other operation is usually provided for floating point numbers, i.e., normalization. Normalized floating point numbers always have the left-most one-bit of the mantissa in R<8>, except of course for a zero fraction. To normalize a number, the mantissa is shifted to the left until a one-bit is in R<8> while the exponent is decreased by one for each shift. Using the format shown in Figure DATA-10, the range of normalized floating point numbers is  $-.100\ 000\ 000 \cdot 2^{-31}$  to  $+.111\ 111\ 111 \cdot 2^{32}$ .

## PROBLEM STATEMENT

Implement a set of subprocesses to provide floating point arithmetic operations for RTM systems; include addition, subtraction, multiplication, division, and normalization.

## DESIGN CONSIDERATIONS

There are three types of errors which can result from the floating point operations:

- 1. overflow - the exponent of the result of an operation is too large to be represented
- 2. underflow - the exponent of the result of an operation is too small to be represented
- 3. significance - normalization of a number causes exponent underflow.

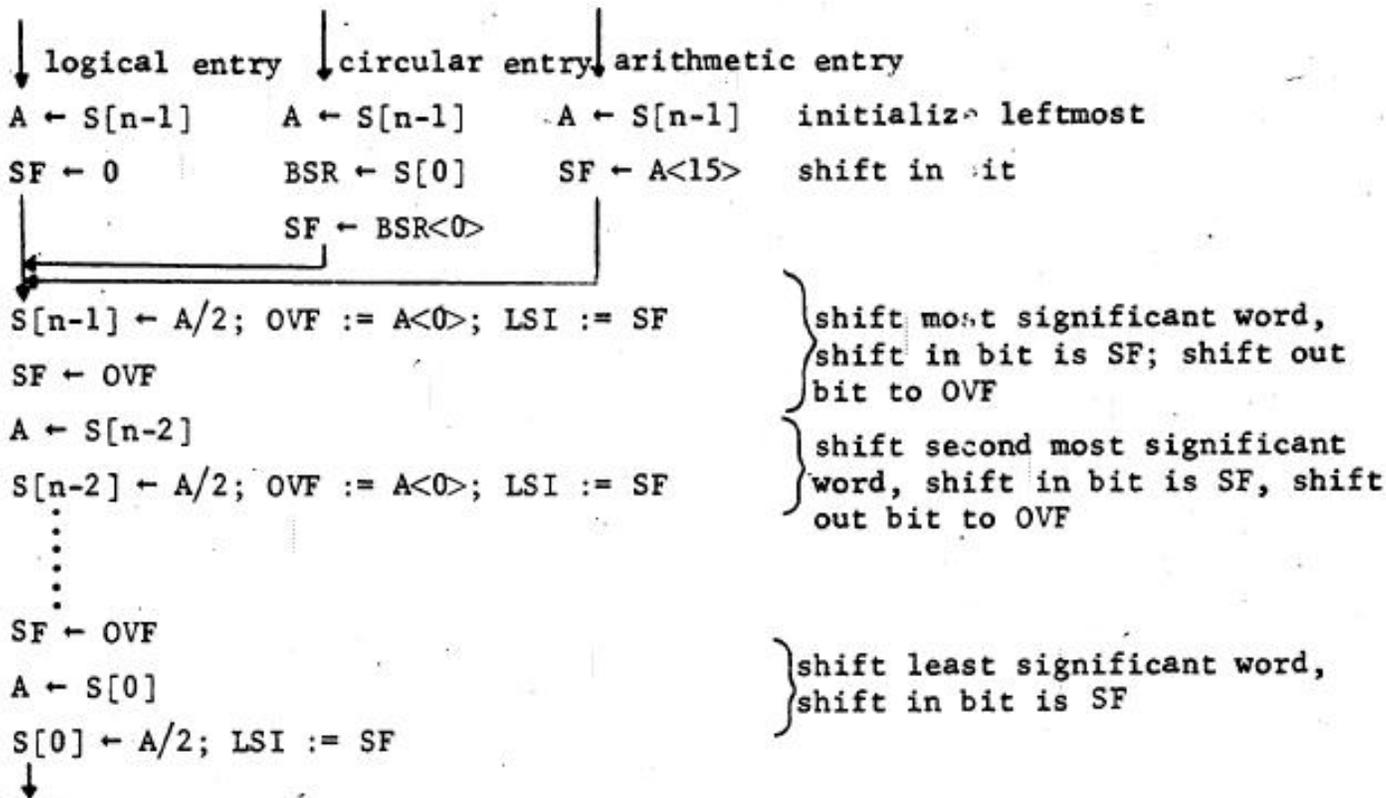
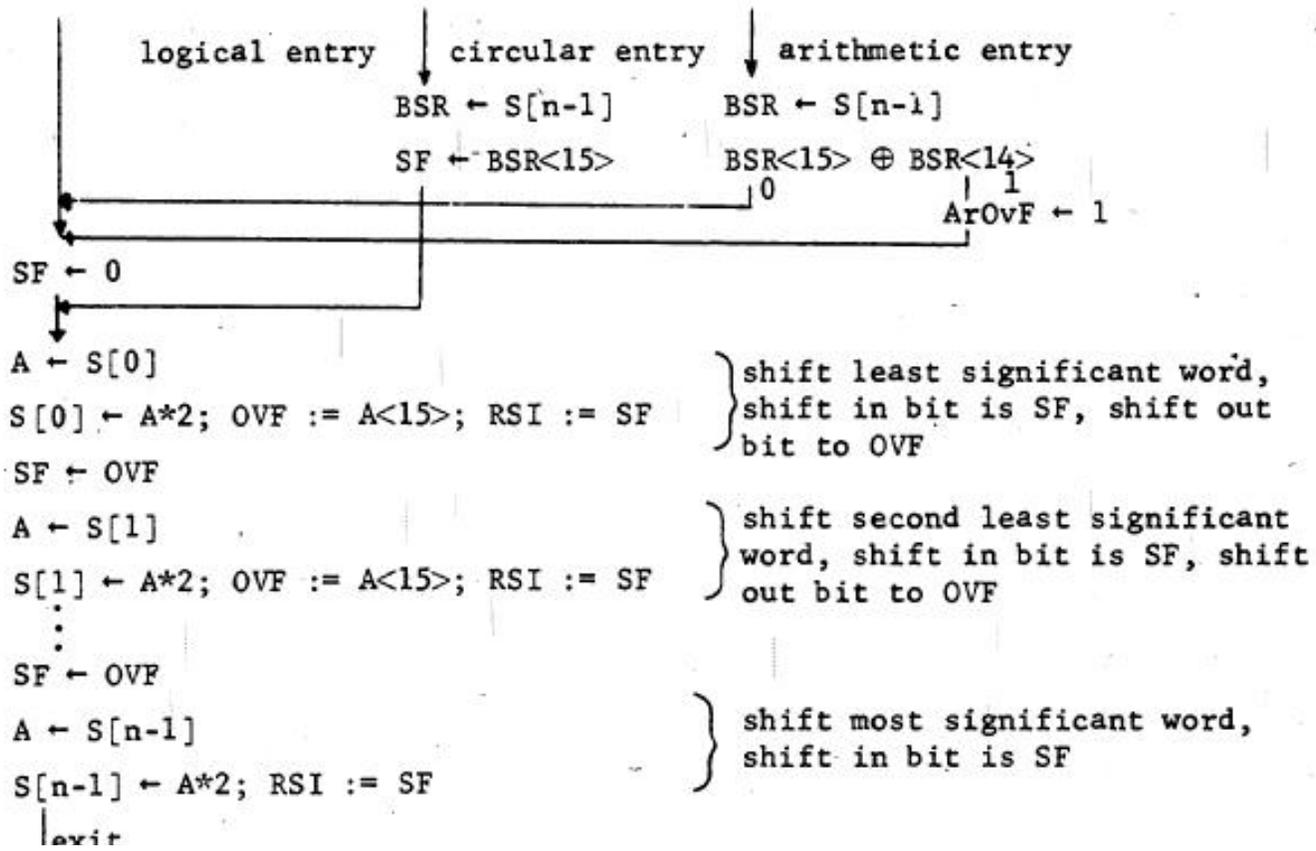


Fig. DATA-8a. RTM subprocess for n-word logical, circular and two's complement arithmetic right shift.



$S[n-1] \leftarrow A \times 2; RSI := SF$

↓ exit

Fig. DATA-8b. RTM subprocess for n-word, logical, circular and two's complement arithmetic left shift.

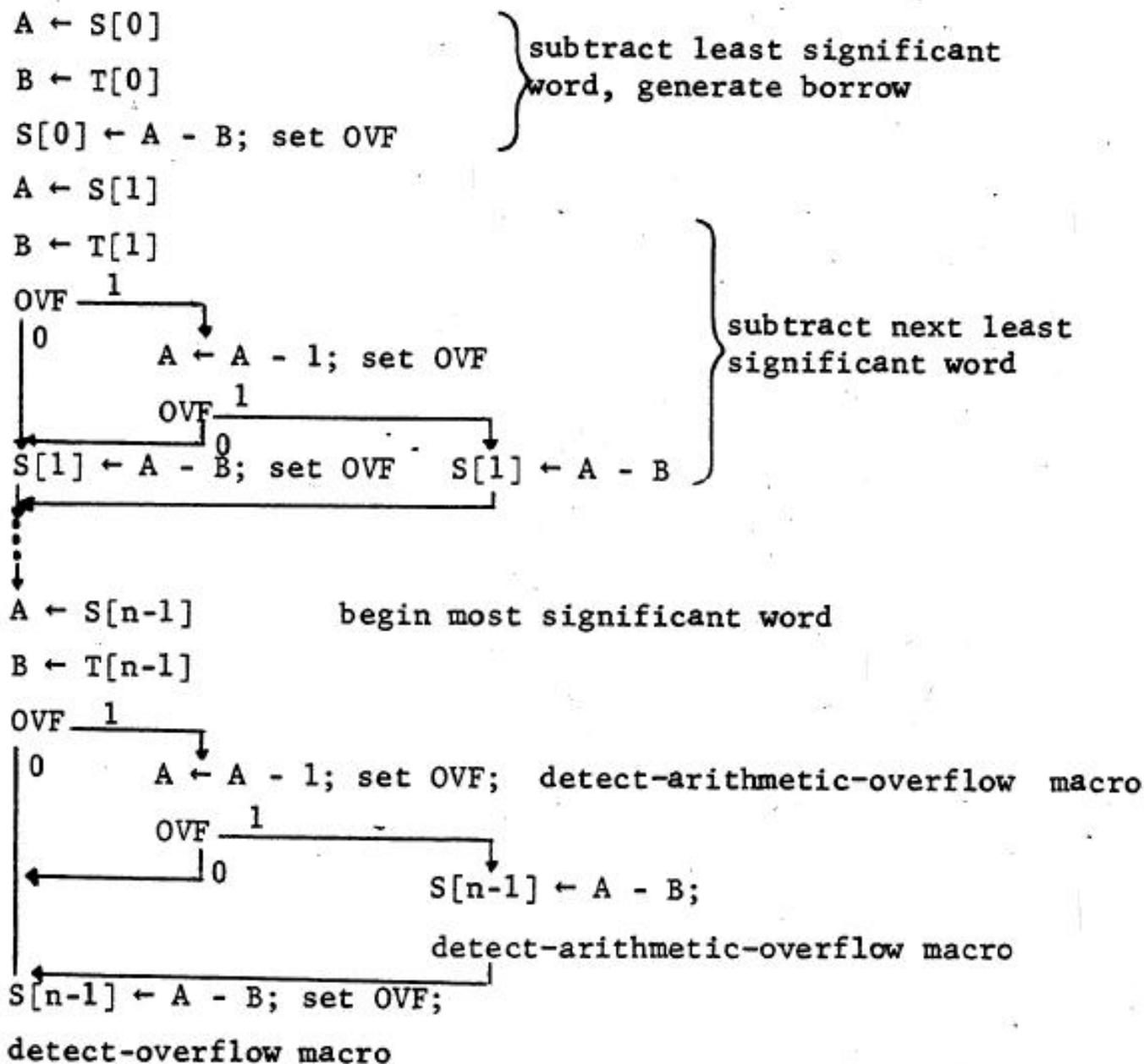


Fig. DATA-7. RTM subprocess for n-word, two's complement subtraction with arithmetic overflow detection.

### N-WORD PRECISION SHIFTING

The definitions of logical, circular, and two's complement arithmetic shifting were presented in Chapter 3 and knowledge of these will be assumed here. The only difficulty in n-word shifting is propagating a shift bit between registers. However any bit shifted out of a register may be saved in the K(bus sense) module OVERFLOW-flag(OVF), and the bit to be shifted into a register may be specified. Thus the

problem is easily solved.

Figure DATA-8a shows implementations for n-word logical, circular, and two's complement arithmetic right shifts. A flag, Shift-Flag\SF, is used to hold the bit being transferred between registers. Figure DATA-8b shows implementations for n-word logical, circular, and two's complement arithmetic left shifts. The ArOvF flag is set to one if the sign is changed on the arithmetic shift.

## MULTIPLICATION

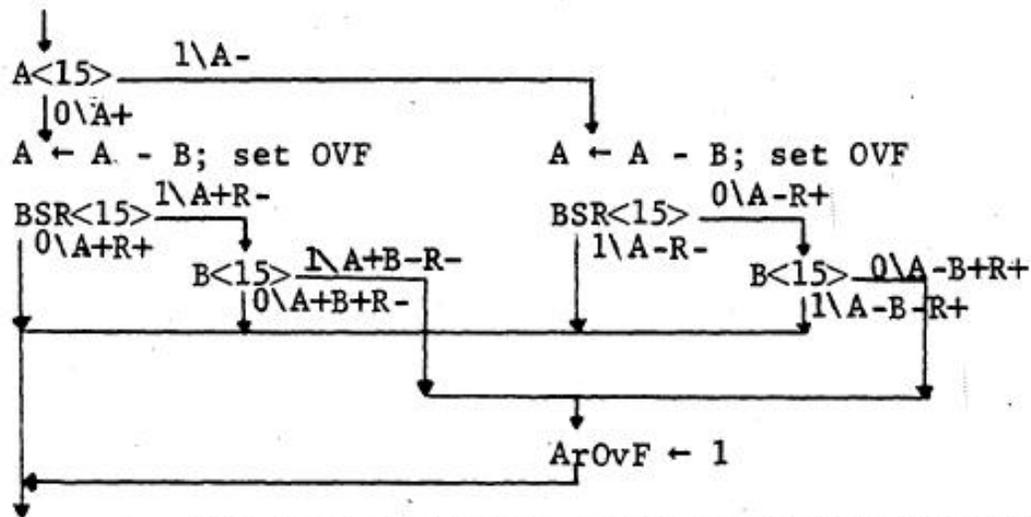
Since multiplication was discussed extensively in Chapter 3, a single

↓  
 $A \leftarrow S[0]$   
 $B \leftarrow T[0]$   
 $S[0] \leftarrow A - B$   
 ↓

a. RTM subprocess for unsigned integer subtraction with no borrow detection.

↓  
 $A \leftarrow S[0]$   
 $B \leftarrow T[0]$   
 $S[0] \leftarrow A - B; \text{ set OVF}$   
 ↓

b. RTM subprocess for unsigned integer subtraction with borrow in OVF.



c. RTM subprocess for two's complement subtraction with arithmetic overflow detection.

Fig. DATA-6. RTM subprocesses for single-word, unsigned integer and two's complement subtraction.

[previous](#) | [contents](#) | [next](#)

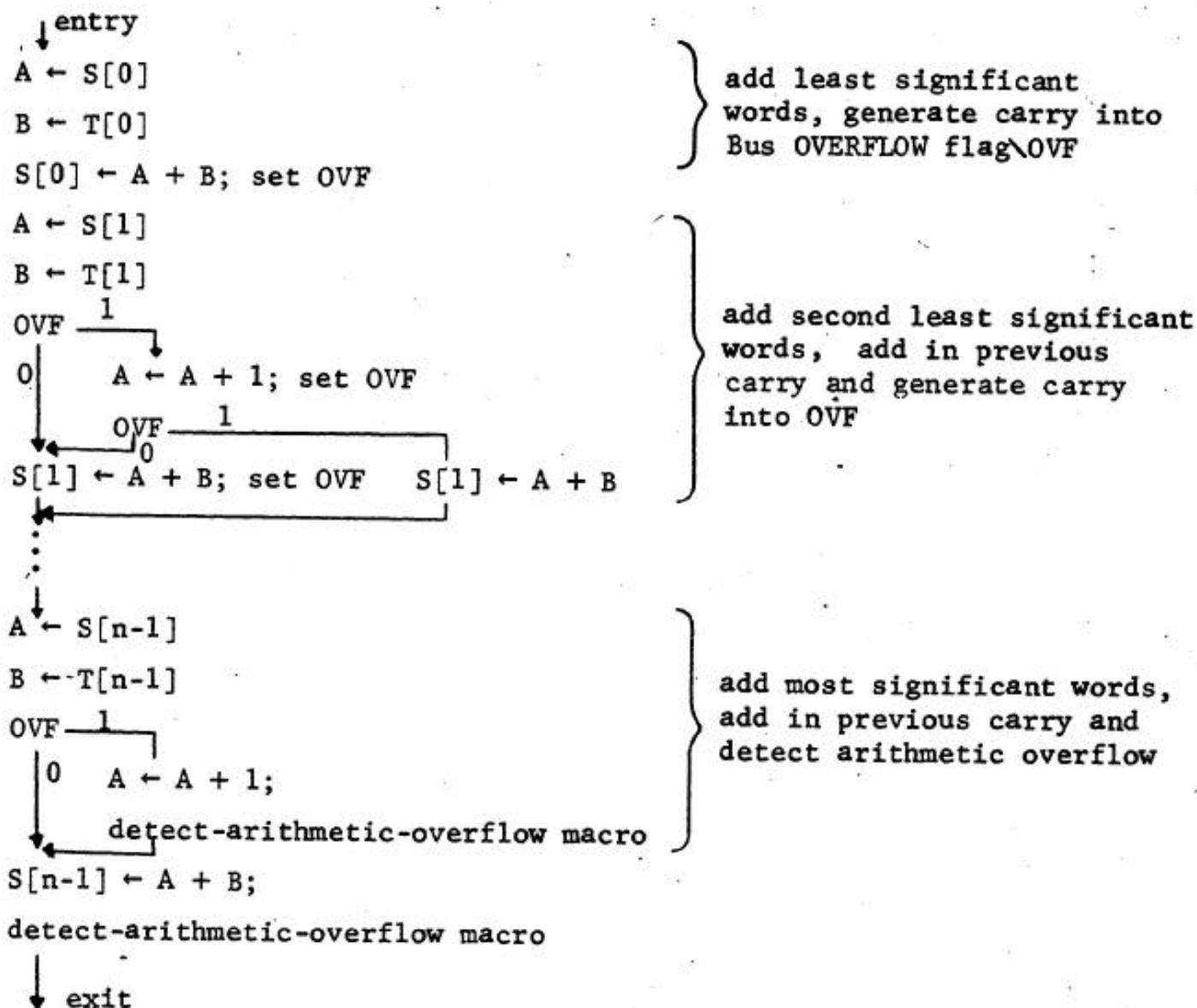
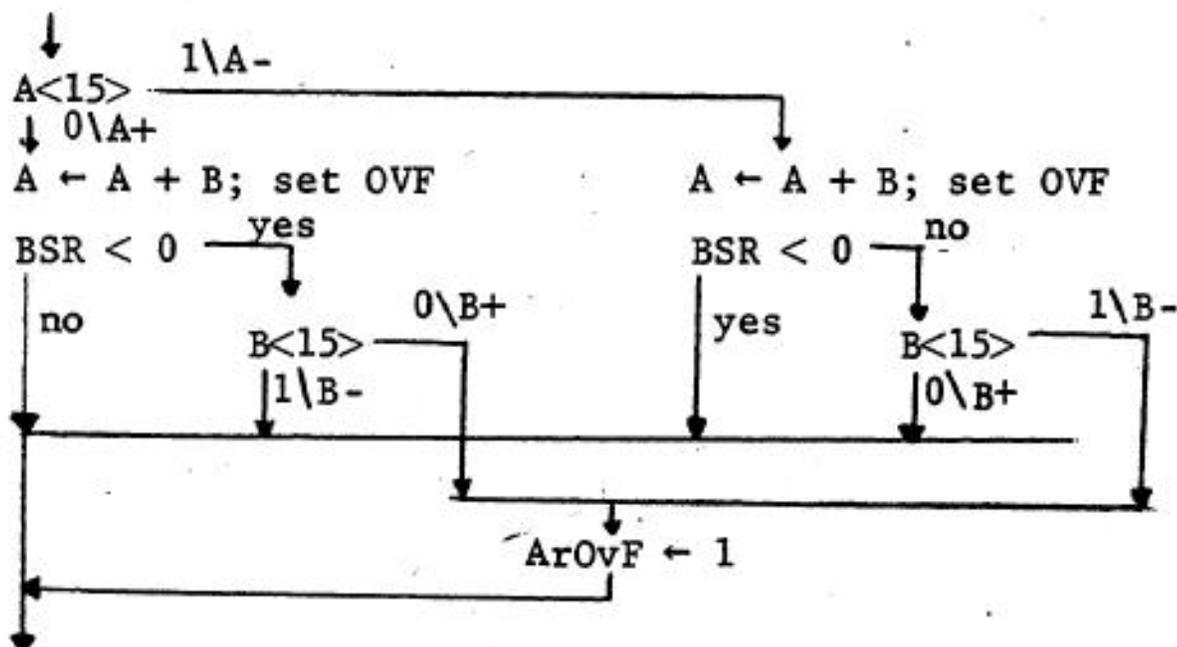


Fig. DATA-5. RTM subprocess for n-word, two's complement addition with arithmetic overflow detection.

Figures DATA-6a and 6b show subtraction of unsigned integers with and without borrow detection respectively. Figure DATA-6c shows a subtraction scheme with overflow detection for two's complement numbers which has the same logical structure as the addition scheme in Figure DATA-4h.

Figure DATA-7 shows an n-word subtraction process. The first n-1 pairs of words of the operands are subtracted using unsigned integer subtraction with borrow detection. A borrow produced in one subtraction will be subtracted from the next word of the operand, then the next subtraction is performed. A borrow may be generated (into the OVF flag) either when a previous borrow is subtracted from the A register or when the B register is subtracted from the A register, but the OVF flag can be set to 1 only

once for each pair of words (why?). Two's complement subtraction is used on the most significant pair of words in the operand. Overflow checks are needed when a borrow from the  $n-1$ st subtraction is subtracted from the A register and when the B register is subtracted from A.



- h. RTM subprocess for two's complement addition with arithmetic overflow detection using no sign-flags or combinational circuitry

Fig. DATA-4 (Part 3 Of 3). Several RTM subprocesses for one-word, two's complement addition.

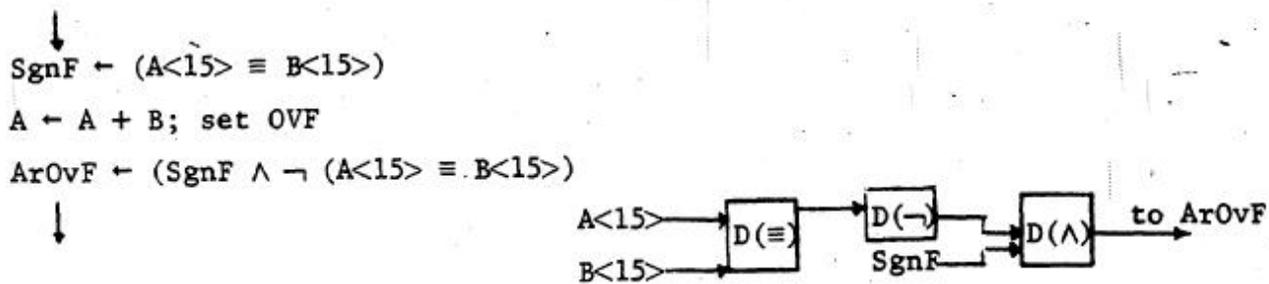
testing the sign of the result and the sign of B. The tests on each path are appropriate to the original sign of A which initiated control along the path.

Figure DATA-5 shows the process to perform two's complement addition of n word operands. The lower n-1 words of each operand contain sixteen-bit numbers, no sign is included. The sign for the operand is held in the most significant bit of the nth word. With this convention, unsigned integer addition is performed on the lower n-1 pairs of words, retaining any carry from a single addition and adding it into the sum of the next pair of words. The most significant pair of words are added using one of the schemes for two's complement addition developed above to detect arithmetic overflow. Note that a carry in one of the first n-1 additions is generated (into the OVF flag) either when a previous carry is added to the A register or when the A and B registers are added together, but that the OVF flag can be set to 1 only once for each pair of words (why?).

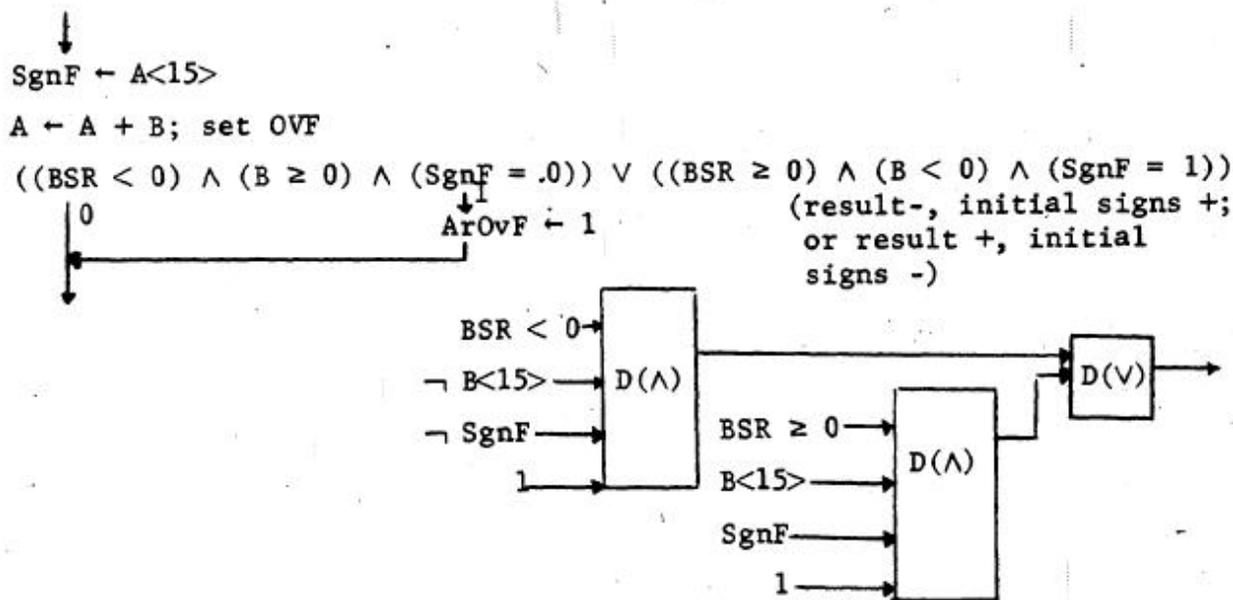
## SUBTRACTION

Figure DATA-6 shows two methods for performing unsigned integer subtraction and one method for overflow detection for two's complement numbers. For unsigned integers, the borrow out of a register is

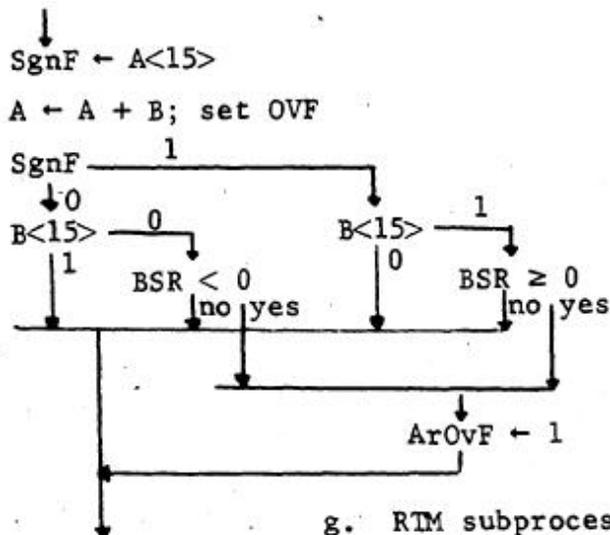
important, but the concept of overflow is meaningless. For one word, two's complement subtraction, the borrow out is not significant, but overflow must be considered. Subtracting a negative number from a positive number may produce a positive number out of the representable range, likewise subtracting a positive number from a negative number may produce a non-representable negative number. Figure DATA-3 shows the difference and borrow results for individual bit subtractions.



e. RTM subprocesses for two's complement addition with all combinational circuitry to set arithmetic overflow flag.



f. RTM subprocess for two's complement addition with a sign-flag for A and alternative combinational circuitry for arithmetic overflow detection.



g. RTM subprocess for two's complement addition with a sign-flag for A and no combinational circuitry for arithmetic overflow detection.

addition with a sign flag for overflow  
no combinational circuitry for arithmetic  
overflow detection.

Fig. DATA-4 (Part 2 of 3).

169

[previous](#) | [contents](#) | [next](#)



- d. RTM subprocess for two's complement addition with carry and arithmetic overflow detection using additional combinational circuitry and a sign-match flag.

Fig. DATA-4 (Part 1 of 3).

168

[previous](#) | [contents](#) | [next](#)

[previous](#) | [contents](#) | [next](#)

Inputs		Outputs for addition			Outputs for subtraction			
carry in	A<j>	B<j>	sum<j>	carry out	arithmetic overflow (j=15)	difference<j>	borrow out	arithmetic overflow (j=15)
borrow in								
0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	1	1
0	1	0	1	0	0	1	0	0
0	1	1	0	1	1	0	0	0
1	0	0	1	0	1	1	1	0
1	0	1	0	1	0	0	1	0
1	1	0	0	1	0	0	0	1
1	1	1	1	1	0	1	1	0

Fig. DATA-3. Table showing individual bit sums and differences; overflow condition for sign bits.

[previous](#) | [contents](#) | [next](#)

control (multiple branches), and designs which use combinational circuitry to compute a single Boolean value for the branch input.

To provide a basis for this section, Figure DATA-3 shows the complete table of sum (difference), carry out (borrow out), and arithmetic overflow results for all combinations of bit patterns at the  $j$ th bit position of a binary adder (subtractor) with input words A and B. The, arithmetic overflow is only dependent on the 15th bit position.

For the simplest case, addition of unsigned integers, carry and arithmetic overflow are identical. Figure DATA-4a shows that addition of unsigned integers with no carry\overflow detection can be performed in one operation. Of course, operations are necessary to load the A and B registers from the S and T memories and store the result, but these operations will not be shown explicitly except in the first few cases.

Figure DATA-4b shows unsigned integer addition with carry\overflow detection. Sixteen-bit unsigned integers can be represented in the range  $0 \leq x \leq (2^{16})-1$ . Overflow occurs when the sum of two integers is greater than or equal to  $2^{16}$ ; the carryout of the register into the Bus OVERFLOW flag signals this condition.

Various methods of detecting carry and arithmetic overflow in addition of two's complement numbers are shown in Figures DATA-4c to 4h. For signed numbers, when overflow occurs, the sign of the result is erroneous; the sum of two positive numbers yields a negative (erroneous) result, or the sum of two negative numbers yields a positive (erroneous) result. Any carry out of the register in signed-number addition is superfluous information, since any significant carry can be detected from the sign-bit. The sum of numbers of different sign will never produce overflow, but may generate a meaningless carry out of the register.

The scheme of Figure DATA-4c uses a flag, SgnF, to indicate whether the signs of the operands match. A match (both operands positive or both negative) indicates potential overflow since the magnitude of the result may not be representable. SgnF is set initially and the addition is performed, then if the sign of the result does not match that of the original operands, overflow has occurred and the flag, ArOvF, is set to I.

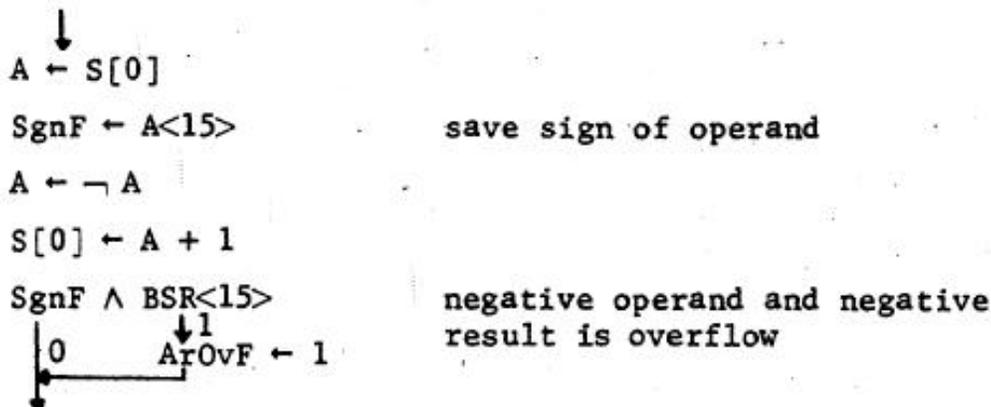
The scheme shown in Figure DATA-4d is similar to that in 4c, but more combinational circuitry is used to set the ArOvF flag. The scheme in Figure DATA-4e is a more direct form of the scheme in 4d, but it violates the condition that the ArOvF flag is not set (to zero) on successful operations.

In Figure DATA-4f, SgnF holds the original sign of the operand in the A register. Note that SgnF (i.e., B(15>)) indicates whether the signs match or not. The test for overflow is made using combinational circuitry to compare the original sign of A, in SgnF, the sign of B, in B<15>, and the sign of the result,

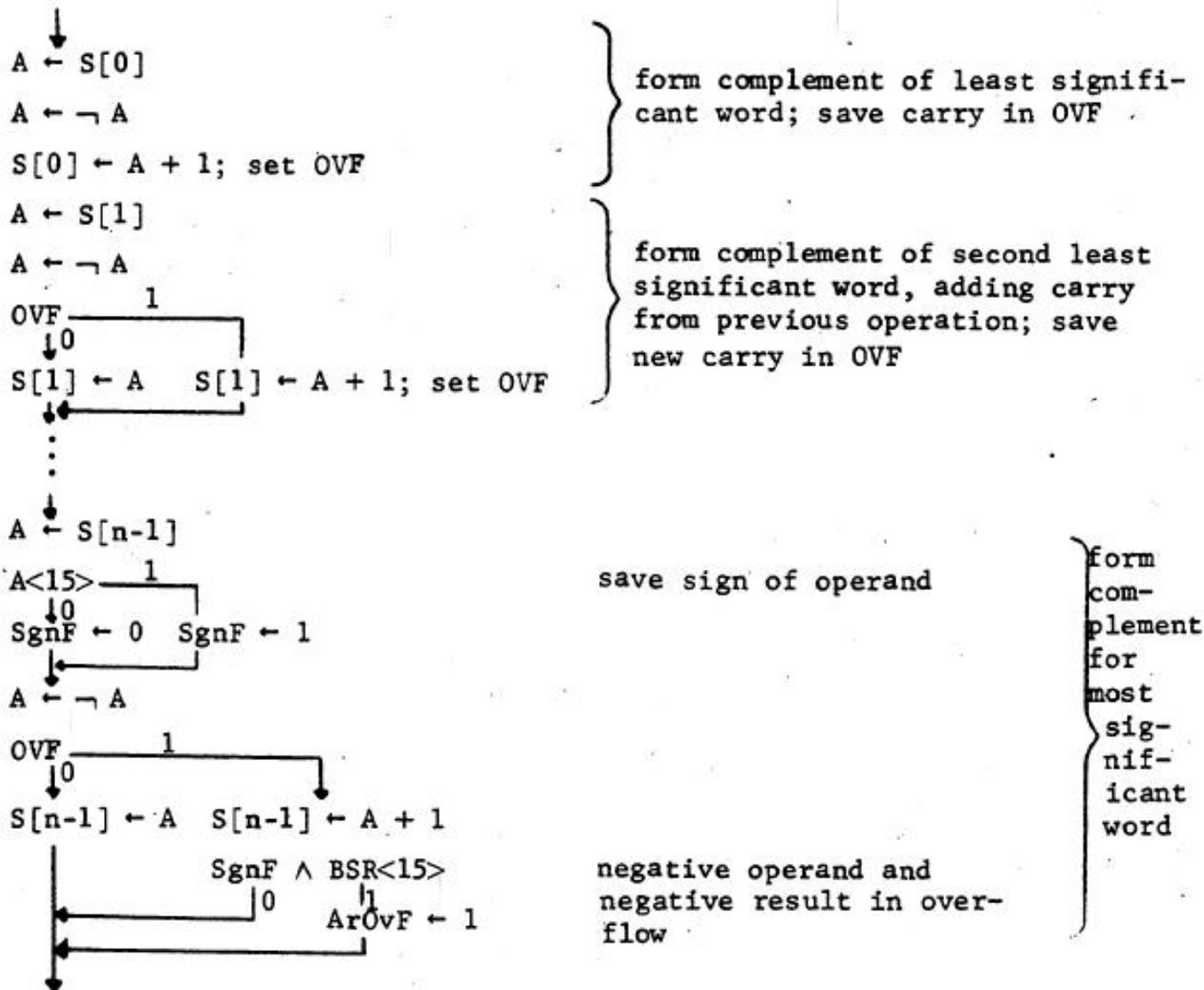
saved in the K(bus sense) and in A.

In Figure DATA-4g, the same principle for detecting overflow as was used in 4f is implemented, but without combinational circuitry. A set of two-way branches is used to successively check the original sign of A, the sign of B and the sign of the result. The ArOvF flag is set when an erroneous result sign is encountered.

The process shown in Figure DATA-4h uses the same principle for detecting overflow as the schemes shown in Figures DATA-4f and 4g, but this scheme does not use a sign flag. The sign of the operand in the A register causes one path to be taken for a negative number, the other for a non-negative number. The addition is performed on each of the paths, then the overflow check is made by



a. RTM subprocess to form single-word two's complement (negation) with overflow



b. RTM subprocess to form n-word two's complement (negation) with

b. KIM subprocess to form n-word  
two's complement (negation) with  
overflow detection.

Fig. DATA-2. RTM subprocesses to form single- and n-word two's complement  
(negation) with overflow detection.

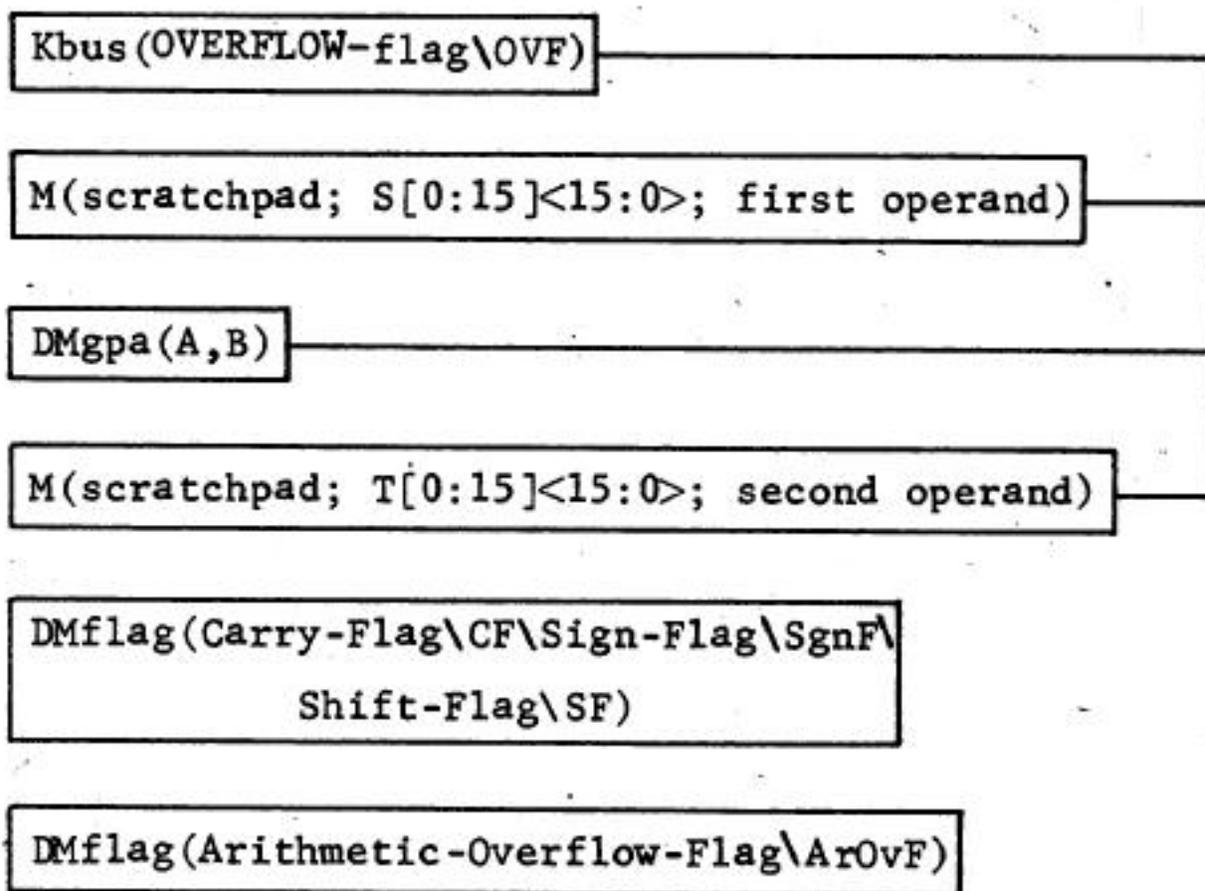


Fig. DATA-1. RTM data part used by all arithmetic subprocesses.

a Sign-Flag \SgnF in the addition and subtraction operations. A second, independent DMflag, Arithmetic-Overflow-Flag\ArOvF, retains arithmetic overflow information. The ArOvF flag is set to a one on arithmetic overflow, but a successful operation (no overflow) does not reset the flag to zero. This allows a sequence of arithmetic operations to be performed prior to checking the result of a previous operation.

## COMPLEMENTATION (NEGATION)

In two's complement notation, changing the sign of a word is more involved than for either the sign-magnitude or one's complement notations which require only a change of sign bit or bit complementation, respectively. In two's complement representation, the value of each bit is  $b_{<j>} * 2^j$ ; however the most significant bit has the value  $b_{<15>} * (-2^{15})$ . (Why?) Since two's complement notation is not symmetric about zero (there is no  $-0$ ), on negation all bits of the word. must be complemented, including  $b_{<15>}$ , and a one added to the result. For example, consider the negation of zero (all 0's) which is zero. The result of complementing all the bits results in sixteen 1's, but by adding a 1, the desired result of all 0's is obtained.

Using 16-bit, two's complement notation, numbers in the range  $-(2^{15}) \leq x \leq (2^{15})-1$  can be represented. Negating the smallest negative number,  $-(2^{15})$ , then, will cause arithmetic overflow since 2115 is not representable. Figure DATA-2 shows both n-word and one-word negation processes for two's complement notation.

## ADDITION

Two implementations of unsigned integer addition and several implementations for one-word, two's complement addition will be given. The examples for two's complement addition are given to illustrate the tradeoffs between sequential

operations on data not encoded in the regular 16-bit, two's complement integer form of the RTM system. (The shifting problem of Chapter 3 contains a discussion of two's complement representation.)

There are other forms of binary encoding for signed numbers such as sign- magnitude and one's complement (see the shifting problem in Chapter 3) which will be discussed only briefly here. Logical design texts, enumerate these other forms. They are omitted from extensive discussion both because most digital systems use the two's complement encoding and because we simply cannot be all-inclusive. The principal reason, though, for discussing two's complement operations is that RTM's were designed to facilitate arithmetic on data in this form.

Two's complement numbers will be presented in two forms: single word precision -- one 16-bit word; and n-word precision -- n 16-bit integers concatenated to form an n\*16-bit word with the sign bit held in the most significant bit of the most significant word. The n-word case is presented for the sake of generality. Two-word (double) precision, which is sufficient for most applications, is readily derived from the n-word case. Using both the one-word and n-word forms, subprocesses are given to perform addition, subtraction, complementation (negation), shifting, and multiplication. These subprocesses are easily understood and will be discussed only briefly. The problem-statement format used for most examples in this book will be returned to for the description of more complex subprocesses.

The binary coded decimal(BCD) encoding scheme using the 8421 code is presented since it is often needed outside the RTM system for devices used in human communications (lights, switches) and for some instrumentation transducers (e.g., shaft position indicators, digital voltmeters). BCD addition and conversion algorithms for BCD-to-binary and binary-to-BCD are given.

A short section presents the scientific notation (floating point) representation, which is used for data that varies over a wide exponent range. The section will introduce the problem of implementing floating point operations but will not give detailed solutions.

## DATA PART FOR THE ARITHMETIC OPERATIONS

In the design of the data operations subprocesses, a particular fixed structure for the data part of the system will be used. In this way, the operations can be considered as a collection of subprocesses, all of which operate with a common memory and arithmetic unit; Figure DATA-1 shows the common data part. Two scratch pad memories, S[0:15] and T[0:15] hold the operands. The arithmetic operations are performed in the A and B registers of the DMgpa. Register S[0] is used for monadic (unary) operations of the type,  $S[0] \leftarrow u(S[0])$ , e.g., shifting, incrementing, complementing, etc. The form for the dyadic (binary) operations is  $S[0] \leftarrow b(S[0], T[0])$ , which includes operations such as addition, subtraction, multiplication, etc. which require two operands. For n-word precision operations, registers S[n-1], ..., S[0] and T[n-1], ... T[0] hold the operands (most significant part to least significant). Note, however, that in some cases T might merely be a renaming of some of the registers in S. For example, using double

precision, S[1] and S[0] might hold the first operand and S[3] and S[2] hold the second, then T[1]:=S[3] and T[0]:=S[2].

Since the arithmetic operations are carried out in the A and B registers of the DMgpa, extra operations are required to load the operands from S and T and store the results back. A DMflag is used to hold carry information, Carry Flag\CF, in some of the multiple precision operations and it is also used as

## CHAPTER 5 ADVANCED DESIGN EXAMPLES

This chapter contains problems organized in terms of PMS component types. The first set of problems treat the Data operations\D type component. These problems are used as a basis for subsequent sections since they illustrate more complex data operations for binary and binary coded decimal (BCD) data types. Since BCD is so important at the system interface to humans, and also since the internal operations of RTM's and in two's complement binary, it is important to understand both the BCD operations and binary-BCD conversion operations.

The next set of problems in the chapter describes systems in which time is the primary constraint. The encoding and decoding of information as a function of time underlies these systems. Two problem classes are presented: waveform generators (synthesizers) and event per unit time (EPUT) counters (analyzers). Subsequent problems give more complex systems for analyzing various waveforms. One such system records the histogram (distribution) of input samples; this system is functionally a compound DM since data operations and memory are central to it. Another system, the DM(coating thickness monitor) problem is similar to the histogram. The two system synchronization problem is illustrated in the time-based systems using an extended RTM, called the K(arbiter). The arbiter allows two independent control sequences to enter a common control, K(arbiter), in which one control sequence only is permitted to emerge at a time. In this way multiple independent systems can share common facilities in a co-ordinated way.

The Transducer section has four complete problem-examples which are also involved with timing and synchronization. Transducers by their function are interfaces between two systems - an inherent synchronization problem. The first problem considers various methods of digital communication via the telephone network An analog sampling unit at the end of a communication link is then presented. Next an interface to a paper tape punch is given. Finally, T(Teletrola), a device which .uses the data communication format of the ASCII code Teletype to produce a four-octave range of square wave tones is given. With Teletrola, any computer with Teletype output can easily synthesize music- like sounds.

Next, two controls\K are given. One control is for solving abstract problems, using the Turing Machine formulation. A Turing Machine tape transport (with infinite tape) is postulated as an "extended" RTM. Several Turing machine problems are solved for smaller (i.e., 16-cell) tape units. The other problem is concerned with the control of a conveyor which has a number of output stations. As items enter the conveyor a record of them is stored in the control, and then as they move along on the conveyor, they are ejected into the correct output station.

The final section presents three functionally different kinds of memories which use a conventional M(array): the queue, stack, and content addressable. Each is used in a particular way, according to function.

## DATA OPERATIONS SUBPROCESSES

### INTRODUCTION

This section is concerned with the design of subprocesses, i.e. macros and subroutines, that perform operations on data held in RTM memories. The data 'operation subprocesses are generally necessary because they provide the ability to perform operations not available in RTM's, like multiplication, division, and/or

162

[previous](#) | [contents](#) | [next](#)

13. Consider the problem of generating constants ,for an RTM system. Make a study of this problem in the manner of this chapter.

14. Do a study in the manner of this chapter on the problem of computation versus control flow for realizing Boolean functions, as illustrated in Figure 34. Be sure to estimate the cost and speed of the obvious ways to do the task, as a basis for comparison.

15. (A) Add a multiply instruction to the Crtm-1 computer of Chapter 6 so that. giving one instruction carries out the operation. (B) Add a multiply-step (i.e., an add an shift) and evaluate it against the other systems.

16. (A) Write down a process for converting an RTM system to a Fortran program for simulation. Carry out this conversion process for the various schemes in this chapter. (B) What must be true of the Fortran system so that the simulation is possible?

17. Unwind the control of the RP algorithm and evaluate it.

18. Checking the RTM System (Part 2). The problem is to check the actual system exhaustively. The difficulty is that there is no obvious way to know whether a given multiplication 4s right except by comparing it to a true source. One solution that .might have been suggested is to tie the system to a computer and use the computer (which presumably has a true multiply) to compare. This is a good solution, if an interface is available, and a terrible solution if not. A second solution that might have been suggested is to create two multipliers, say the 8-step and the RP, and play them against each other. If they are in error at least it will be in the same way! This is a reasonable solution, though inelegant and clearly not fool-proof. A third solution, and the one we recommend comes from considering the identity:

$$(A+B) * (A-B) = A^2 - B^2 = A*A - B*B$$

Thus, given A and B one can form:

$$(A+B) * (A-B) - A*A + B*B$$

If this is not zero, then there must be an error in the multiplier. Now do this, over all values of A and B that lead to legal products (since A+B must be less than 2^8). In this manner every product gets executed at least once and checked. (A) Design such a tester to exhaustively check the multipliers in this chapter. (B) Does this scheme guarantee that the multiplier is correct? (C) If not, what is the difficulty and can you fix it? (D) Does this whole exercise suggest a general way to test certain systems? What is it? (E) Can you list some other functions which might be checked this way? Some others that seem to you unlikely? (F) How do you know that your tester is correct?

19. Algorithms for Multiplication. Consider the identity used in Problem 12:

$$(A+B) * (A-B) = A*B - B*B.$$

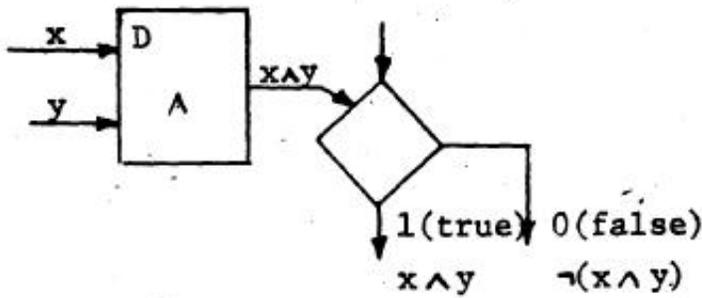
(A) Can this be used to design a multiplier? (B) Do so.

20. In this chapter we have not considered one of the simplest of all multiplication algorithms. This simply consists of adding the multiplicand to itself a number of times that is equal to the multiplier. Design a minimal cost RTM system to implement this algorithm.

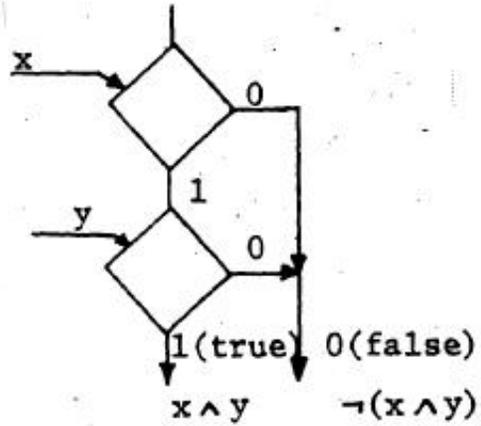
4. We have started with a hardwired version of multiply defined in an RT module system and compared it with various software implementations. But systems of RT modules themselves represent specializations. If one moves down to general logic-level design, one would confidently expect multipliers to be either cheaper, faster or both. (A) Search the literature on multipliers and compare their costs and speeds with those of the systems discussed in this chapter. (B) Compare the methods used with those used here. Can the algorithms used at the logic design level be applied at the RT level?
5. In the discussion of the pipeline and array, we only worked with the multiplication. (A) Pipeline the entire processes of Figure 7. (B) Show the array design noting that there is not necessarily a loop but possibly arrays of adders.
6. Conduct an analysis in the style of this chapter of the problem of calculating a quadratic function:  $F=A*X^2+B*X+C$ . (A) Do it assuming that the coefficients are fixed, so that the system takes  $X$  in as input and delivers  $F$  as output. Assume that numbers are 8-bit positive integers. This case might arise in practice; e.g., the quadratic function plays a fixed role in a large calculation, say, it computes a decision criterion. (B) Do it assuming that there are four inputs  $X$ ,  $A$ ,  $B$  and  $C$  with one output,  $F$ .
7. The 8-step multiplier has the property that it actually computes  $Z=X*Y$ . Make use of this feature to design a general polynomial evaluation system, assuming a fixed number of coefficients and an 8-bit input. Design the first, middle and last stages.
8. Reduction of Control. In unwinding the control loop, we engaged in a shift of knowledge from encoded data (the iteration variable) to position (in the control sequence). These are simply two different ways of maintaining the same state. (A) Can this be applied elsewhere? (B) Avoid the flags in the pipeline by generating all stages in parallel, and then feeding forward all at once. Does it help the speed?
9. General Concurrency. In seeking ways to introduce parallel computation into an algorithm we sought places in which there was no output from one part of a computation that was required as input to another. We did it by examining Figure 4. (A) Can you find a way to identify all the opportunities that arise in this fashion, just by a mechanical examination of the flowchart? (B) Does this lead you to another representation of the computation that is useful for looking for parallelism?
10. Forms of Parallelism. We introduced three forms of parallelism, two of which were special -- array parallelism and pipeline parallelism. The third, concurrency, we simply characterized as general parallelism. (A) Are there any other interesting special cases of parallelism? (Hint: Take the various metaphors for naming parallelism and ask about variations.) (B) Do they have any application to the multiplication problem? [Note: One answer is job shop parallelism, in which there are job queues established in front of specialized machines.]
11. Checking the RTM System. With the simulation we checked the RT flowchart completely. But we

certainly did not check the actual RT circuit. (A) Find a way to exhaustively check the RTM system itself after it has been assembled and seems to be working. (B) Why is this hard to do? (Hint: Examine the Fortran program used in the simulation to see why it is possible there.)

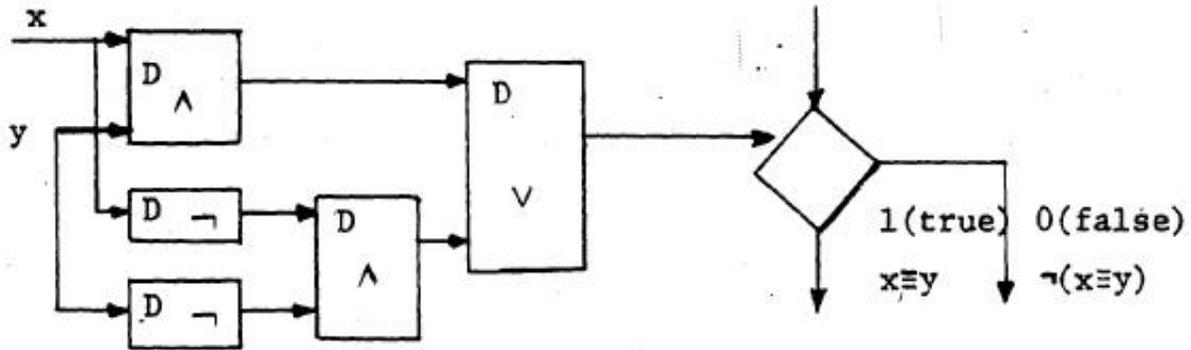
12. In the 8-step straight line implementation, the reader is invited to show how the number of control modules can be further reduced while keeping the essentially straightline nature of the structure by using a multiple level subroutine. This achieves a slightly better point in the local cost-performance space.



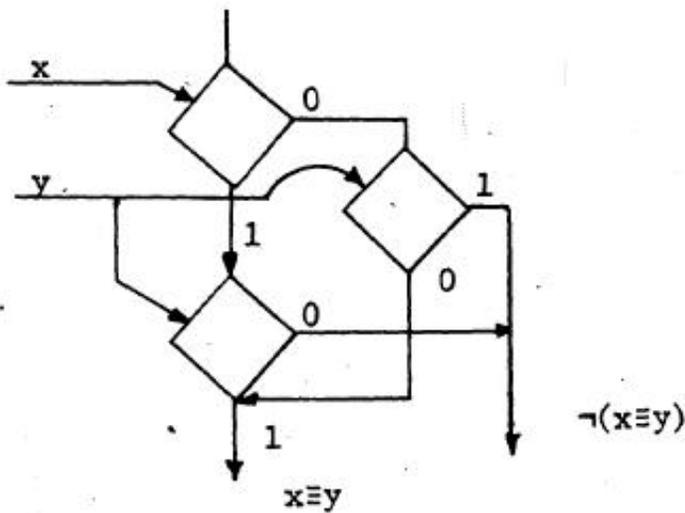
a. D(combational)-flowchart subsystem for D(AND)



b. Flowchart subsystem for D(AND)



c. D(combational)-flowchart subsystem for D(Equivalence - i.e. bit match)



d. Flowchart for D(Equivalence)

**Fig. 34. D(combination) and flowchart versus flowchart for determining Boolean variables.**

## INCREASING GENERALITY AND MODIFIABILITY

These criteria were not a part of our study. They clearly do not have much impact on a simple multiply operation. However, in more substantial systems they become important. The basic fact is that all generality costs in terms of hardware and (possibly) speed, since additional parametric inputs must be processed and either consulted during the computation or the computation adapted according to them. Thus, one rule to keep costs down and speed up is to decrease generality. However, if generality is required, and especially if the kinds- of amounts of generality are not clear at design time, then going to a software implementation provides one generally available solution.

## INCREASING RELIABILITY

Like generality and modifiability, reliability is an evaluative dimension that we did not explore. As we discussed earlier, it is not really an appropriate concern of alternative designs in most cases, in this book. In general, one should use a technology that has the right reliability characteristics so that straightforward design is appropriate. However, this is not always possible, and we list in the figure several standard strategies that are used to deal with unreliability.

## COMPUTE WITH LOGIC

Since Boolean operations are just selections on the basis of whether an operand is a 0 or 1, and since one can store the knowledge of a computation in the control path, it is possible to do all Boolean computations in the control part. In general, with a suitable functional system (e.g., our D, M, K labelling of components) there should be only a single way of doing basic operations and this should be with components of the appropriate functional label. Indeed, Boolean operations can be done with D components. They can, of course, be done with a DMgpa. However, this is extremely expensive, since a DMgpa processes 16 bits in parallel. Thus, there is a full set of combinatorial data operations, e.g., D(AND|OR|NAND|NOR). But the third way of computing with K's remains an alternative, which can sometime be of use in saving either time or hardware depending on the exact details of the design. Figure 34 illustrates how this is carried out.

## PROBLEMS

1. Reliability as another criterion. (A) Using the formulation that the probability of failure is roughly proportional to cost, derive several tables for the failures of the RTM modules. Make reliability calculations for two similar implementations of multiplication in order to see how different their reliabilities are. (B) Design a highly reliable version of multiply, assuming that you wanted to remain within RTM's.
2. The multiplication scheme using table look-up would appear to work with any sized component, e.g.,

2-bit, 3-bit, etc. With each component there is a trade-off of the size of memory for the table and the number of stages in the computation. (A) Design the multiplication system using 3-bit components. (B) If one considers  $N$ -bit numbers and  $K$ -bit components, can one decide in general which  $D$ -bit versions are plausible candidates?

3. In the RP algorithm we introduced an initial selection of the smaller operand to be the multiplier in order to speed up the system somewhat. (A) Is this addition worthwhile (since it takes more hardware)? (B) How can you answer this question so that the answer is usefully available to future designers who might want to use this implementation, say by being added as a note in a designer's notebook?

## DECREASING COST BY MERGING CONTROL PATHS

Whenever two control paths are similar or identical they can be shared. Usually this involves separation of the paths again at some points, so that some additional processing must be done. But if the shared path is long enough, there can be a slight decrease in cost. The modification of the concurrent version of multiply, where we got rid of one  $K(\text{parallel-merge})$  and put a  $Kb^2(C=O)$  in series, was essentially a small example of this.

## SAVING MEMORY

In the chapter we emphasized only cost and performance. On the surface there does not appear to be any reason to want to save memory (or contrarywise, to wish to use memory), since memory can be seen as simply one component of hardware costs. There are two reasons for separating it out. First, there are generally several ways to generate alternative systems that depend on increasing or decreasing memory and these options need to be available. However, the effect of these options on cost and performance is not predictable in many cases, so they cannot all be subsumed there. Second, because memory comes in large units, it is either very cheap (when excess capacity has been made available in the system by other demands) or very dear (when staying within an existing memory limit will keep the costs down, but exceeding it forces a large jump). In either case one may wish to consider explicitly the trade-off of memory for other processing.

### Recompute on Demand

This is really the opposite strategy from the table look up. Even though an algorithm forces the computation of certain data ~t some point, it recomputes it on subsequent demands rather than take the space to store it.

### Store Information in Control Part

By running a separate control path, knowledge that is implicitly known at the beginning of the control can be remembered all along the path. The other side of this coin occurs in trying to merge two paths to save cost, in which some memory may be required because running a joint path no longer permits knowing which source one came from.

### Compress Data

The total set of bits to hold the information may be much in excess of the logarithm (base 2) of the actual set of distinguishable states that occur. Thus, there is the opportunity to recode the data in a more compact representation and thus save memory space. This almost always costs additional processing, since the algorithm usually cannot work on the data in compressed form and hence a decoding operation

must be performed whenever the data is to be processed. Whether the encoding and decoding operations cost more than is saved by the memory depends on the details of the algorithm and the volume of data.

### Avoid Storing Unrequired Data

Often the question is not really one of compacting data in terms of a new encoding, but simply of recognizing that a large part of the data in the representation that is being used "automatically" with the problem will never be processed and therefore can be discarded. A related possibility is to recognize when an item of data has been used for the last time and thus its memory space can be used for another purpose.

then a macro is used in the implementation, `Kmacro(...)`. the data operations are directly determined from the algorithm and then the additional memory required is determined. (This is the correct order, since RTM data operations often have associated memory that obviates the need for separate memory.)

Thus it is always possible to obtain a starting design. This base design may not be the preferred one. The viewpoint of Figure 33 is that one will wish to explore variations in design before one settles on an appropriate version for actual implementation. But it is a fundamental feature of a good technology that there exists a straightforward way of producing at least one reasonable design from the specifications for a system. The functional notation for the RTM modules, the division into control and data parts, and the creation of the control part as isomorphic to a standard flowchart for specifying algorithms -- all these serve to provide this essential property of good technology for the RTM system used in this book. An essentially similar scheme underlines the system of the Macromodules of the Clark..

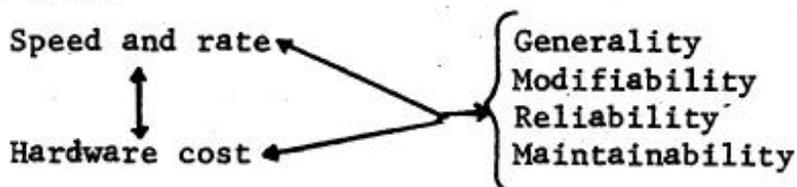
Figure 33 should be looked at as a proposer of plausible moves for wandering in the design space. There is no guarantee that a given move can be made for a given design or that, if made, it will produce a design that has the specified change. In many cases we know something about the conditions under which a move is applicable -- e.g., array parallelism requires a series of independent, almost identical computations on an array of data. We have not added such notes on feasibility to the figure because the conditions of applicability are not generally clear. Similarly, though sometimes it can be assured that a move will produce an effect in the direction under which it is listed, often it cannot and, in any event the size of the change is quite unpredictable. Equally important, the new system will differ along other evaluative dimensions as well. Indeed, the trade-off structure at the top of the figure guarantees that this will often happen in undesirable ways -- that is, in ways that decrease the system's value. Of course, as we know from the example in the chapter, occasionally, a move may yield a design that is preferred along all relevant dimensions. The point is that Figure 33 is a generator of moves and that the result of attempting such a move must be evaluated post hoc, after each new design has been constructed.

Most of the entries in Figure 33 are familiar from earlier in the chapter and further discussion of them would be repetitious. However, the new ones that have been added require some comment, although we cannot illustrate them extensively here.

## INCREASING SPEED BY MULTIPLE REPRESENTATIONS

Whenever data is to be used for several purposes, there is a conflict over the representation of the data -- to which of the several purposes it is adapted. This means that extra processing has to be done for one or more of the purposes due the presentation adopted. Often keeping multiple representations adapted to each purpose can save computation time. It invariably requires extra memory and usually requires extra processing to set up the representations. Thus, there needs to be a high rate of use to compensate for these disadvantages.

Multiplication is too simple to show this affect. In principle one could, imagine keeping around both the regular and logarithmic representations of numbers to be used in addition and multiplication, respectively.

**Trade-offs:****To increase speed:****Compute in parallel:**

Array parallelism

Pipeline parallelism

Functionally independent parts (concurrency)

Use table look-up

Unwind control loop

Use multiple representations

**To decrease cost:****Share facilities:**

Subroutines

Extract memory and share data operations

Merge control paths

Use software control

**To save memory:**

Recompute data when needed

Store information in control part

Compress data

**Avoid storing unrequired data:**

Store only essential data:

Extract and recode

Reuse memory after data has been used

**To increase generality and modifiability:**

Use software implementation

**To increase reliability:**

Use few parts

Duplicate computation by separate algorithms

Add checking stages

Use redundant data:

Error detection

Error correction

**Alternatives of unpredictable effect:****Create new algorithm:**

Use table look-up

Change data representation

Use table look-up  
Change data representation  
Compute with logic  
Compute with control part.

Fig. 33. Table of tradeoffs in digital systems design.

accurately known, since empirical frequencies of the types of failures are known and the costs of repair and replacement are also known. Sometimes no quantitative estimate can be made because we (computer science) do not yet understand what we want to mean by a given concept. Generalizability is an example.

Despite the restriction of the present analysis, we see that it goes quite a way. Certainly it is adequate enough to select an implementation that should be used within some particular larger system. Where basic data is available, accountings can be made of the performance and cost along additional dimensions. Observe, however, that if one of 30 systems is to be selected, having them evaluated on 100 dimensions will probably not help much in making the right selection -- two or three dimensions will dominate. The others are of interest to the decision only as a check that costs and performance on subsidiary dimensions are not out of bounds (which can often be checked without detailed calculations). A detailed accounting of the other costs may still be of interest however, even though they do not contribute to the decision.

## SUMMARY OF THE TYPES OF VARIATIONS

Our strategy in this chapter has been to raise some issues of RT-level design by taking a single task and exploring the space of possible designs. We have asserted the existence of recurrent options for creating alternative designs and that these options are a principal means for generating the design space. Each example design has been created in response to one of these recurrent patterns, in order to bring out as many of them as possible. In this last section we summarize these patterns in a more systematic way. We expand the list somewhat beyond the examples given. Although our single task of multiplication has served us well, it could hardly be rich enough to reveal all the useful patterns. Our summary, of course, can also hardly pretend to be exhaustive, but it can make evident a few more patterns.

Figure 33 provides the summary. At the top it gives the various major evaluative criteria amongst which trade-offs can occur. The existence of trade offs is guaranteed simply by the fact that several functions of the same variable (here the design, which varies over the design space) do not in general attain their respective optima all at the same place. Thus, to maximize on one criterion is to be less than maximum on all the others and in principle there should exist a trade-off of each criterion against all the others. The structure given in the figure expresses more than this, namely, the possibilities for significant trade-off. Thus, one regularly trades performance (in speed or rate) for hardware and vice-versa. The introduction of any significant special design for any of the criteria on the right -- generality, modifiability, reliability, maintainability -- leads to a trade-off with either performance or cost. But it is rare that one directly trades, say, reliability for generality, or speed for rate.

The remainder of Figure 33 is organized by listing the changes to be sought in order to effect changes in specified criteria. The viewpoint is that an implementation is given and the problem is to modify it. This may seem to jump over what would seem to be a major concern: how to get an implementation for an algorithm in the first place. But, as we illustrated in the first part of this chapter, there exists a

straightforward technique for going from an algorithm, expressed as a conventional flowchart, to an RTM design. This is essentially a one for one mapping from the algorithm into the control part of the RTM system. Whenever an operation is used in the algorithm that is not an RTM primitive,

## SUMMARY

The task we have examined (multiplication) is an extremely limited and specialized one (as indeed are all particular tasks). The quantitative comparisons from Figure 31 reflect a mixture of general factors and factors special to the task. The particular gains in both speed and cost which occurred with unwinding the control loop depend intimately on the algorithm. They were not there when we did the same task with different algorithms, either the RP algorithm or the 4-bit component algorithm. On the other hand, the degradations in the software system show up in some form in working with any task, though the exact numerical size of the factors due to each cause will vary both with task and with computer organization and implementation. Consequently, this evaluation exercise is meant in large part to be an example of how to explore and compare the options within the task given and the hardware available in the actual case.

A particular limitation of the entire analysis has been the decision to consider only two criteria: performance and parts cost. As Figure 8 of Chapter 1 made clear, there are a large number of distinct objectives that are of concern in an RT system. We have attended to one already on the performance side, operation time versus operation rate, though we have restricted ourself to operation rate in this section. Multiplication is too simple to have a profile of performances over varying inputs that makes a difference to the evaluation. The RP algorithm does take a variable length of time, performing very fast for simple inputs such as multiplication by 0 and 1, and in a rough way taking longer with larger operands; the 8-step algorithm takes an invariant time. But it is not easy to set up examples where such a difference in performance profiles is consequential. In general, however, it does matter, e.g., whether wait time in a batch computer is uniform for all jobs or proportional to job duration.

Other aspects of performance, such as reliability, are vital in general for digital systems, but cannot be affected by the sorts of variations under consideration here. Given a specific RT technology, such as the PDP-16 modules, reliability becomes a function simply of the total complexity of the system (thus becoming coupled with total parts cost). The reliabilities of the modules do vary, but not in a way that permits direct selection of modules to achieve high reliability. One can of course deliberately design for reliability, introducing redundant computations or checking 'stages. If such were a genuine concern, one would be well advised to expand the design space to include alternative technologies.

Although there are restricted variations of performance measures, especially in simple systems, there are a substantial number of costs to be considered. Total parts cost is, of course, the easiest one to obtain. But there are assembly costs, and parts cost if bought in bulk, design costs, modifiability costs, maintenance costs, power costs, cooling costs, and space costs. On large systems' all of these can be independent items and must be separately evaluated. On small systems and in a modular technology the situation is much as it is with reliability. Almost everything is fixed on a per module basis and therefore varies directly with total parts costs. In an actual accounting of the different costs, there will be some variation (i.e., the correlation with total parts cost is say, .8 rather than 1). But there is little margin for deliberately

designing to minimize some other cost without at the same time minimizing total parts cost.

Many aspects of system performance and cost are not specifiable in any quantitative form, e.g., maintainability or generalizability. Sometimes this is because there is not enough knowledge (usually in terms of field experience) to make an estimate. For many types of systems maintainability can be rather

only four on the average instead of eight, which are more than enough to compensate for slightly more processing in each iterative step. We also noted in the discussion earlier the several ways in which the RP appears to be a qualitatively preferable algorithm.

In case of the 4-bit component algorithm, we have only a single point, which is not worth -considering because of the tremendous shuffling of data required to find and add the components.

## HAROWIRE VERSUS SOFTWARE

We have a complete sequence for the RP algorithm for the hardwired case (which we can take as the standard 2 DMgpa implementation) and the four software versions. Comparing the K(PCS), which is the most basic form for microprogramming of control, the evaluation is straightforward: evoke and branch instructions take roughly a factor of 2.6 and 50 longer respectively than with a hardwired structure. Thus, the degradation factor depends on the relative use- of the two instructions. In this case the degradation is a factor of 6. A similar factor of 6 can be observed in comparing the Crtm implemented with a hardwired control with one implemented with a K(PCS) microprogram. Since the C(16/M) is the K(PCS) as far as its control structure is concerned, the performance effect of using a 16/M is to degrade the time to that of a 1 DMgpa organization. That is, the 16/M is a factor of 6 slower than the 1 DMgpa hardwired version. The other effect of the 16/M is a higher cost due to the overhead of the total physical assembly, costs that did not have to be included in the specially constructed hardwired system.

The Crtm case provides a final view of the total performance degradation that accrues to programming in a stored program computer. First, the necessity of fetching instructions from memory accounts for a factor of 2. Second, the interpretation of the instruction once it is obtained accounts of a factor of 4. Third, the inefficiency of coding caused by the minimal number of active registers (the single accumulator) accounts for a final factor of 2. Thus we get a total factor of degradation from the hardwired case of:

$$2(\text{fetch}) * 4 (\text{interpretation}) * 2 (\text{coding}) = 16.$$

For the microprogram implementation of Crtm-1 with K(PCS) we might expect:

$$16 * 6 (\text{microprogram}) = 96$$

## GENERALITY VERSUS SPECIALIZATION

Our example of a single input-output function (multiplication) does not lend itself well to exploring the costs of obtaining additional generality. However, the software systems do provide some indication of the issues. All of them provide more capability than the multiplication algorithm requires, for they can interpret any sequence of instructions within their memory limit, and thus perform many additional

functions. For these minimal computers, with minimal data operations and active registers, the cost of this generality shows up in the cost of unused memory and in the cost of the interpretation of unused instructions. For instance if we take the C(16/M) with a 256 word 'memory, the cost could be prorated over the various cells it uses. We have assigned to the multiply the cost of each of its memory cells and the full cost of any part of the system that it uses at all. Thus, it bears the full cost of the interpretation cycle. Thus, even if the system were fitted with exactly the right sized instruction memory, there would still be excess costs of generality. If the extra instructions were eliminated, we would get a system identical to a K(PCS) implementation with one DMgpa.

Russian Peasant's algorithm versus 8-step loop	1.0 ~ 1.25
Use of hardwiring versus K(PCS)	3 ~ 57 (avg. of 6)
Stored Program Computer versus RP loop	
Operation times (slower memory, and multiple steps/instruction)	.125
Inefficiency for extra operators	.5
Overhead of looping versus straight line	2
Sharing a single DMgpa versus two DMgpa	~.5
Concurrency in 8-step algorithm versus algorithm with built in loop control	2
Array parallelism (8 element versus 8-step straight line)	8
8 stage pipeline versus 8-step straight line (factor of 8, but overhead of factor 2.7)	3

Fig. 32. Table of performance factors for various RTM multipliers.

computation (the average). Using that formula and taking  $N=8$ , the overall gain would be about 3.2. The other two implementations are affected analogously, only with the degree of parallelism being 4 and 2 respectively, rather than 8, yielding gains of 2.1 and 1.4. The systems maintain their ordering, but the whole comparison has flattened out. The extreme case, the array, becomes less impressive relative to the others.

## UNWINDING THE CONTROL LOOP

The effect of taking operations that have to be controlled iteratively and replacing the explicit control by a straightline sequence is to remove the overhead. The improvement in performance is by a factor of two, since each step has a Kevoke for counting (overhead) and a Kevoke for the add and shift part. The speed improvement is to be expected. But there is a cost improvement as well, which is unexpected. It arises because the extra control hardware in the unwound loop is minimal, so that it does not offset all the savings from getting rid of the DMgpa for the iteration count. Note that the times are essentially the same.

## THE ALGORITHM

We have three algorithms to compare: the eight-step algorithm, the Russian Peasant algorithm (RP), and the 4-bit component algorithm. The 8-step and RP algorithms were each designed for both the 1 and 2 DMgpa cases, so we can see clearly the effect due to the algorithm -- they are about equal, and any significant improvement depends on the distribution of the numbers input to the RP algorithm. Any improvement comes about because of the automatic calculation of the termination condition and because the number of iterations are

8-bit, shift right algorithm

- 82 - 2 DMgpa (Fig. 4)
- 8a - 8 element array (Fig. 11)
- 8acl - 8 element array closed subroutine (no figure)
- 8p - 8 stage pipeline (Figs. 12 and 13)
- 82c - concurrent (Fig. 14)
- 81 - 1 DMgpa (Fig. 16)
- 8scl - straight line, closed subroutine (Fig. 17)
- 8sop - straight line, open subroutine (Fig. 17)

4-bit partial product with table lookup

t (Fig. 21)

Russian Peasant's Algorithm

- r24 - 2 DMgpa, 4 iterations (Fig. 23)
- r27 - 2 DMgpa, 7 iterations (Fig. 24)
- r17 - 1 DMgpa, 7 iterations (Fig. 24)
- r2K7 - 2 DMgpa, K(PCS), 7 iterations (Fig. 28)
- r1M7 - 1 DMgpa, C(16/M), 7 iterations (Fig. 29)
- rC7 - hardwired Crtm-1, 7 iterations (Fig. 30)

Fig. 31b. Legend for graph of Fig. 31a.

even the two .of a DMgpa). Thus, operands must be acquired from memory and a further degradation in speed occurs, accounting for another factor of .5 with the particular computer.

The main influence of parallelism can be observed by comparing the three designs with the

corresponding version of the direct eight-step algorithm: an eight element array multiplier; an eight stage pipeline; and the system for concurrently counting and taking multiplication steps. For the eight step array performance of the multiplier increases by a factor of 8. For the eight stage pipelined multiplier, a factor of eight is degraded through overhead by a factor of 2.7, giving an overall increase of a factor of three. Finally, in the case of concurrent loop control and multiplication step, there is a factor of two improvement, since the two dominant operations are being done concurrently.

Whether these factors are realizable in any particular application is dependent on the details of the larger algorithm. Recall the calculation made in the discussion of the array implementation with regard to a particular embedding

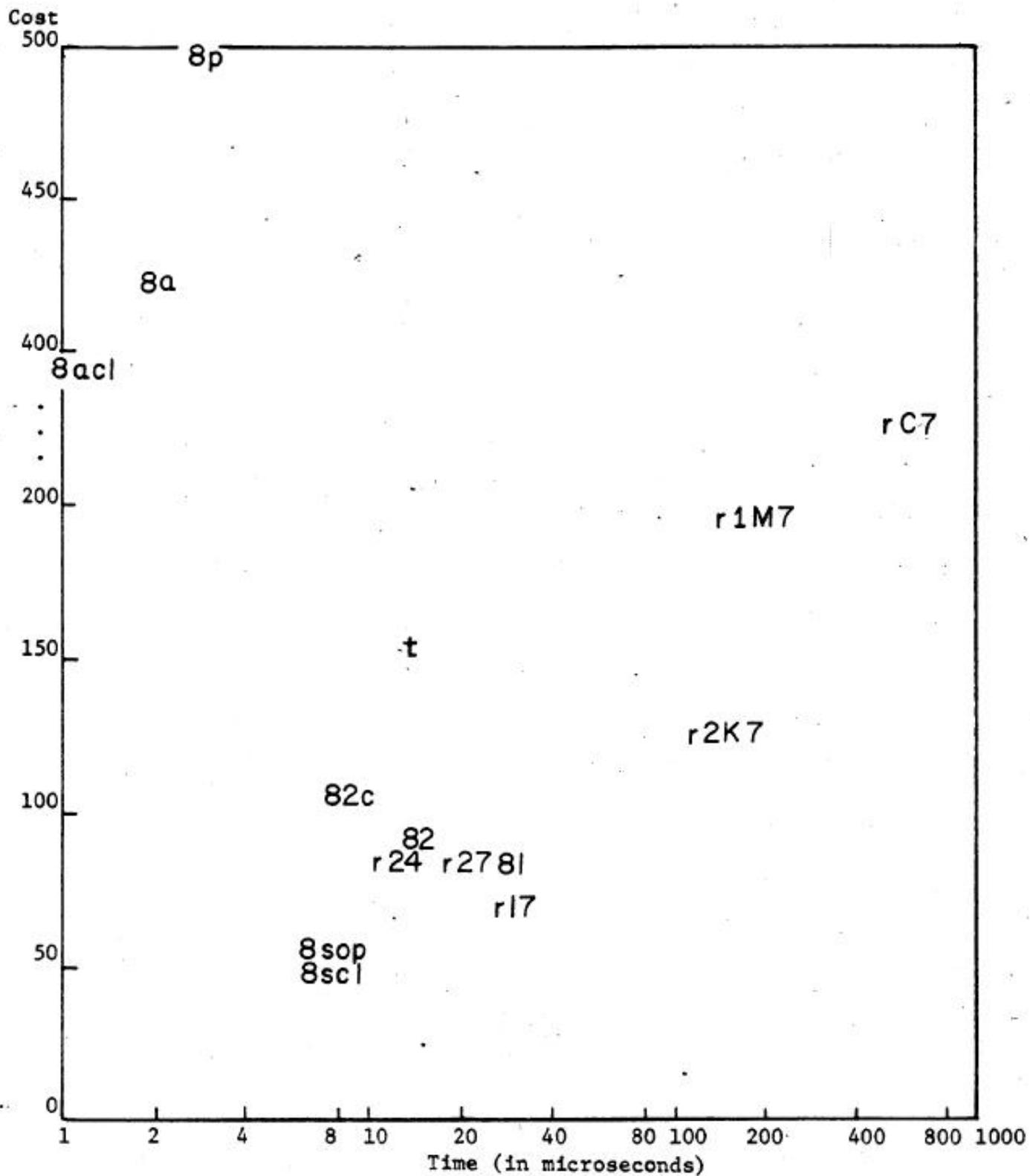


Fig. 31a. Graph of cost versus performance for various RTM multipliers.

[previous](#) | [contents](#) | [next](#)

operation. The final stage, realized in all large computers, is where a multiply exists as a primitive operation. The latter brings us full circle to where the multiply itself is realized in a hardwired form.

## EVALUATION OF THE DESIGN

We have now created a substantial number of designs, all ostensibly to do the same task. Our analysis is not complete without an attempt to compare the designs in some uniform way. Throughout we have been concerned primarily with operational performance, measured in terms of operation time or operation rate, and total system hardware cost, measured in terms of the costs presented in this book, which are representative of relative technological cost. These are not the only objectives of concern in a design -- recall the list of objectives in Chapter 1. We will return to this issue at the end of the section; for now let us work with the kinds of data we have made available in our analyses.

In Figure 31 we select most of the systems and plot them horizontally on effective operation time (the reciprocal of the operation rate) and vertically on total hardware cost. In discussing each of the separate systems, we did not always provide the explicit cost figure. These can be computed from each of the figures using Figure 17 from Chapter 2. Earlier we distinguished operation time and operation rate as two separate measures. We prefer to use operation rate in the figure, on the assumption that parallel organizations will be used in situations where the rate can be exploited. We prefer to express the operation rate as an effective operation time, since a designer's intuitions are built up in terms of microseconds per operation rather than millions of operations per second. RT-level systems are generally serial so that one adds operation times for a heterogeneous collection of operations. There is essentially no use for a number like 750,000 multiplications/second, since it never occurs that a system does nothing but multiplications for any period of time.

The most striking fact about the Figure 31 is that the operation times vary over a factor of 500, from 1 to 500 microseconds and the costs vary over a factor of 10, from 50 to 400. The total ranges, however, are less important than detecting the effects of the various types of alternatives that are generally available in designing an RT-level system. We can get some insight into the effect of parallelism and facility sharing, unwinding control loops, varying the algorithm, hardware versus software control, and generality versus specificity. Where quantitative factors can be made of the effect from the figure, these are summarized in the table of Figure 32.

## PARALLELISM

We take parallelism to include both replication of hardware to achieve concurrency factors greater than 1 and facility sharing to achieve concurrency factors less than 1.

The simplest effect of parallelism, facility sharing, is independent of algorithm and implementation. Some of the implementations are given as pairs in Figure 31, showing both the 1 and 2 DMgpa

arrangements. With shared resources extra time is required to carry out the sharing, consisting of shuffling data to and from the shared resource. In the two implementations for the eight-step algorithm that use the straight-line control, only two active registers are required. Since a single DMgpa provides these, only one DM module is required. Sharing in this form costs in time by a factor of about .5 to .6.

Facility sharing also occurs in the Crtm, since it has only a single register (not

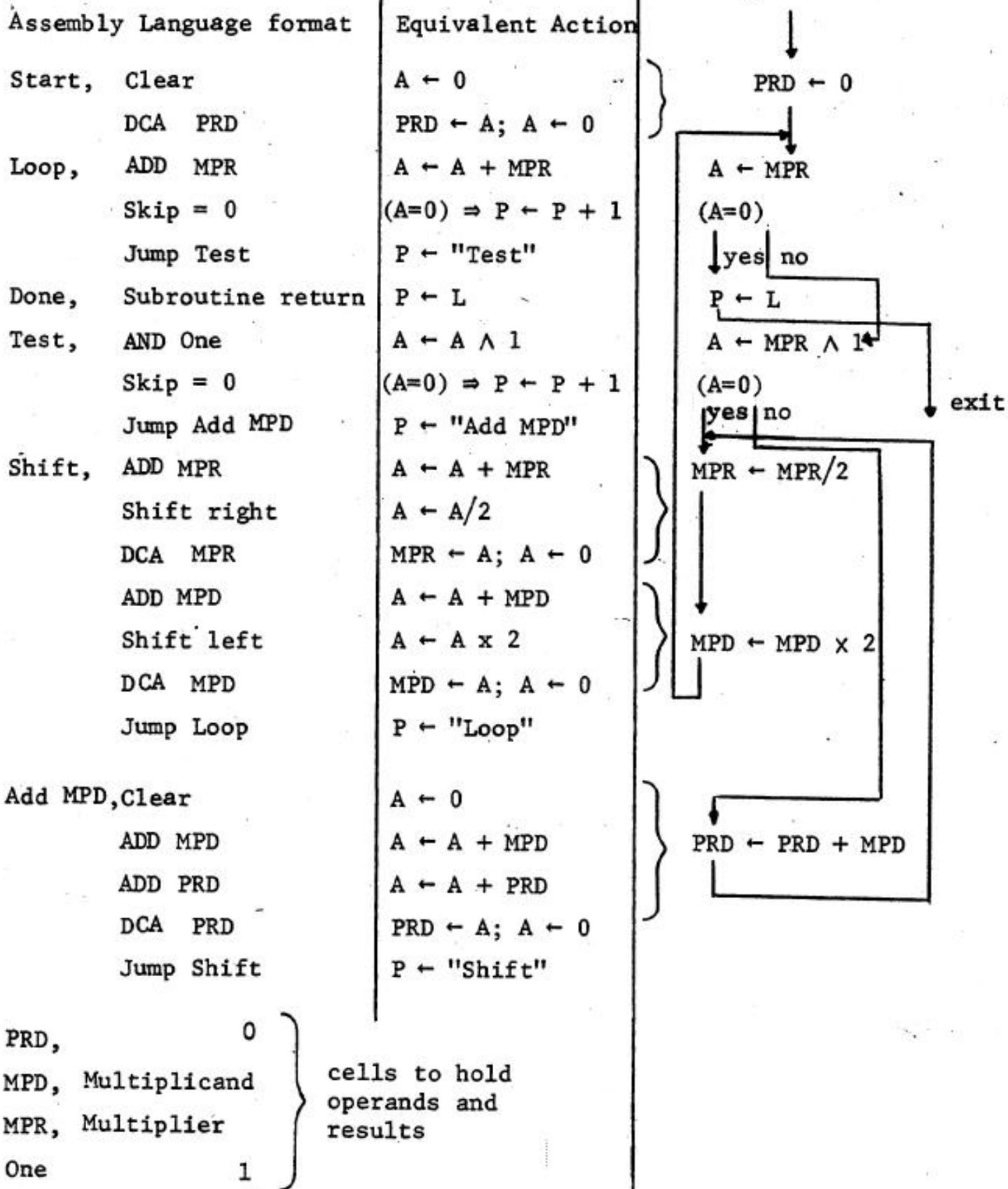


Fig. 30. Subroutine for multiplication using RP algorithm for Crtm-1 computer of Chapter 6.

## Chapter 6.

147

[previous](#) | [contents](#) | [next](#)

START,	$B \leftarrow 0$	} P initialize		S0\Multiplicand\MPD
LOOP,	$A \leftarrow S1$		} $S1 \leftarrow S1/2;$ Test if multiplier 0	S1\Multiplier\MPR
	IF BSR = 0 GOTO DONE			Product\P, held in B
	$A \leftarrow A/2(S)$			
	$S1 \leftarrow A$			
	$A \leftarrow S0$		} $S1 < 0 \Rightarrow B \leftarrow S0 + B$ $S0 \leftarrow S0 \times 2$	
	IF OVF GOTO ADD			
SHIFT,	$A \leftarrow A \times 2$			
	$S0 \leftarrow A$			
	GOTO LOOP			
ADD,	$B \leftarrow A + B$	} add multiplicand $B \leftarrow S0 + B$		
	GOTO SHIFT			
DONE,	EXIT	return from subroutine		

Fig. 29. Subroutine for multiplication using RP algorithm for PDP-16/M.

Even before making a quantitative estimate of the operation time, an important feature of the organization of small minicomputers can be seen directly from Figure 30. Since they only have a single arithmetic register (A) with the second operand coming from the primary memory, they require a large number of operations to load and store data. This is apparent from a horizontal comparison between the righthand side, expressing the essential computation, and the code, along the lefthand side. This extra quota of operations, which occurs throughout all the code, creates another factor of speed degradation to be multiplied in with the factor already present for accessing of control steps from memory and the interpretation loop for decoding each control step once obtained.

## SUMMARY

We now have three distinct software organizations to be compared, both against each other and against a hardwired organization. Rather than make that comparison in this subsection, it will prove worthwhile to step back and examine generally all of the solutions that we have obtained in the chapter.

Before we leave this implementation, however, it is worth noting that there is no single hardware-software tradeoff, but rather an entire spectrum of implementations, both on the hardwired side (which one might expect) but also on the software side, defined as those implementations that encode the control part of the algorithm into memory. Putting forth three distinct software implementations was done, in part, to make this point. In fact, we could have proceeded through at least two more stages. One would be a mini-computer organization where a multiply-step of some kind was given as a primitive

## C(16/M): A MICROCODE COMPUTER

Chapter 6 describes a small computer, the PDP-16/M, which consists of a particular collection of RT modules in which a fixed set of control operations have been preassigned in terms of a fixed data part. Only a certain subset of all possible transfers for the particular data part can be given. The control part is a K(PCS), so that the program is the set of control steps that are placed in the primary memory of the K(PCS).

The instruction set of the C(16/M), being identical in form to that of the K(PCS), is similar to the instruction sets typical in microprogramming. Thus, we can refer to the C(16/M) as a microprogram or microcode computer, to distinguish it from computers with conventional instruction formats, involving N-addresses, indexing, general registers, and various other addressing modes. Terminology is still unsettled in the microprogramming area, since the term microprogrammed computer refers not at all to the nature of the computer's instruction set, but to the fact that its interpreter is realized through another - computer, which interprets a code of its own (the microcode). Thus to call a computer a microcode computer is only describing it indirectly by means of a family resemblance to other computers which occur in a certain application (i.e., microprogramming). But the C(16/M) stands on its own as a particular small computer.

The program for the RP algorithm coded into the 16/M is given in Figure 29. It is similar to the hardwired version in Figure 24, since the 16/M uses only a single DMgpa. Consequently, extra memory transfers are required. The operation time required is longer than for the prior case, where the data structure could be adapted to the algorithm at hand. Thus, the preselection of a fixed set of data operations implies some costs to be paid later when particular algorithms are to be coded. The cost, by the way, is an opportunity cost (to use a phrase from economics), since it only shows up in terms of alternative organizations that were not available. If two DMgpa's were included in the 16/M, then real costs would be accrued in all those applications where only one (or none) DMgpa was used. For then actual equipment would be sitting idle.

## Crtm-1: AN RTM MINICOMPUTER

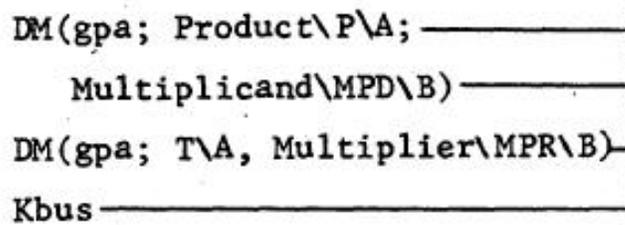
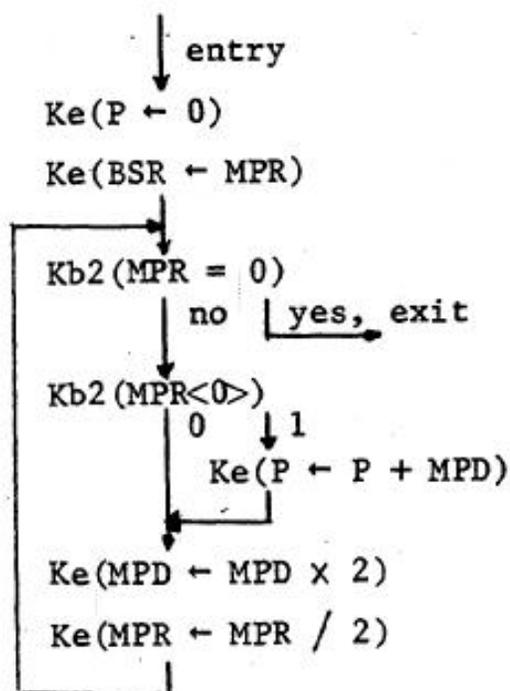
Multiplication is carried out using subroutines in some small minicomputers. One, called Crtm-1, is described in Chapter 6. We can use it to provide a third example of a software realization of multiplication. It differs from C(16/M) and from K(PCS) in having a typical minicomputer one-address instruction set.

In Figure 30, we show the subroutine for multiplication coded for the computer. We give it in three representations, since we have not formally defined the machine here. The lefthand side gives the code, in conventional assembly language format: the instruction location label (if it exists), followed by the operation code, followed by the address (or other argument, on occasion). In the center we give the equivalent ISP statements. The meaning of the operation codes can rather easily be inferred from this

representation. The righthand side gives a flowchart for the algorithm in the style we have been using throughout.

This computer could be implemented either with a hardwired control part or with a K(PCS), which would make it a microprogram computer. Between the two there is a speed-cost trade-off of roughly a factor of two in speed for roughly a factor of two in cost. Which choice is taken clearly effects the performance of the total system as a multiplier. Better stated, it provides two separate points on a cost-speed trade-off curve, to be compared against the points for C(16/M) and K(PCS).

Figure 28 gives the encoding for the RP multiplication algorithm, using the 2 DMgpa scheme of Figure 23, but without the initial selection of the multiplier. Due to the design of K(PCS), there is a direct mapping of each hardwired control step into an encoded state in the K(PCS) memory.



Start,  $P \leftarrow 0$  first of subroutine

$BSR \leftarrow MPR$

Loop, IF  $(BSR = 0)$ , GOTO Done

IF  $MPR < 0 >$ , GOTO Add

Shift,  $MPD \leftarrow MPD \times 2$

$MPR \leftarrow MPR / 2$

GOTO Loop

Done, EXIT

Add,  $P \leftarrow P + MPD$

GOTO Shift

Fig. 28. RTM diagram and subroutine for K(PCS) multiplier using RP algorithm.

144

[previous](#) | [contents](#) | [next](#)

used so that costs can be optimally prorated. Figure 27 compares the costs of the two schemes for implementing control, as a function of the number of control steps. This assumes an average cost of 0.6 per hardwired control step and an average of one 8-bit word in the memory for a software control step. The remarks of the earlier part of the section come into clear view here. With few control steps (less than about 80) the fixed costs of K(PCS) make it more costly; above this point the software scheme gets progressively more advantageous. The steps in the cost function of K(PCS) are caused by having to obtain control memory in discrete blocks of 256 words each.

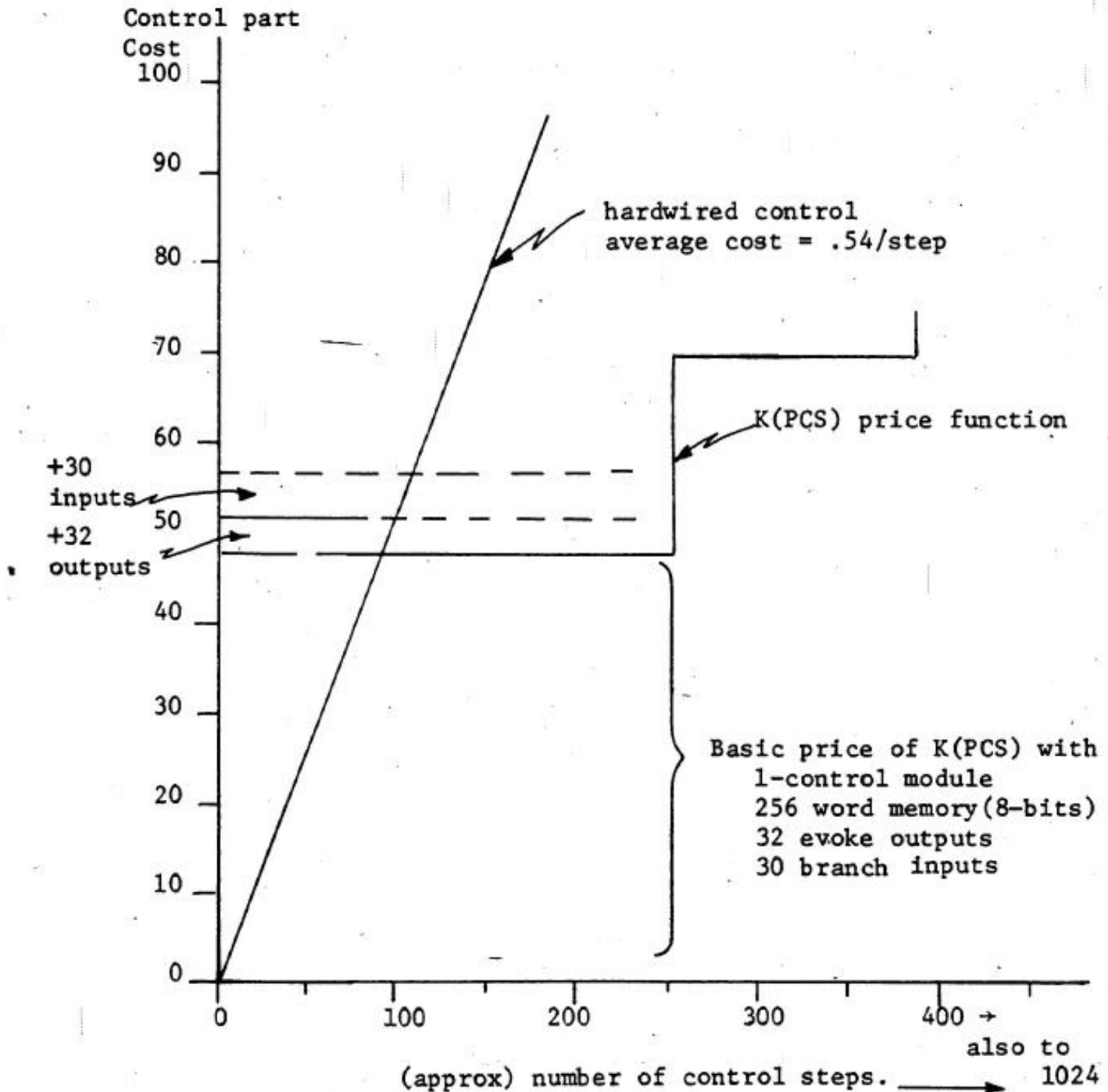


Fig. 27. Graph of cost of control part versus control steps for hardwired and K(PCS) cases.

143

[previous](#) | [contents](#) | [next](#)

examples of software systems, which implement their control by storing an encoded version of it in a memory. We will continue with multiplication as an example, since we have already produced an extensive analysis of hardwired versions. All of the software systems belong within the RT-level design domain. None has an instruction that simply evokes a multiplication operation; hence for each multiplication must be programmed. We will not be concerned here with the internal hardware structure of these software systems. Chapter 6 is devoted to computers and the systems used here are all analyzed in detail there. We take the software systems as given, and examine how they perform multiplication.

### K(PCS\PROGRAMMABLE COMMAND SEQUENCER)

The K(PCS\Programmable Command Sequencer) was introduced in Chapter 2 as an available RTM control module. It provides a general scheme to control RTM's from a memory rather than from a hardwired control part. The discussion there and in Chapter 6 provides a description of the encoding and gives the details of the module's internal operation.

#### Cost K(PCS) parts

Basic control part	15
32 evoke outputs (decoder)	5
30 branch inputs (multiplexor)	5
256 word memory	22.5

<u>Unit</u>	<u>Operation times K(PCS)</u>	<u>Operation times, for corresponding hardwired version</u>	<u>Factor Degradation</u>
Kevoke (including register transfers)	2.1	(0.8)	2.6
Kbranch (branch not taken)	1.6	(0.04)	40
Kbranch (branch taken)	3.0	(0.04)	75
K(subroutine call)	3.4	(--)	$\infty$
K(subroutine return)	2.0	(0.02)	10
Kserial merge	0	(0.03)	0

Fig. 26. Table of K(Programmable Control Sequencer) cost and performance.

With K(PCS) additional time is required to fetch the encoded control steps from memory, giving the longer operations times shown in the table of Figure 26. Costs are also given in the table, assuming that all the words of the memory are

(hereafter, hardwired) RT systems is the implementation of control. In hardwired systems, such as we have been considering so far throughout this chapter, the control part of the RT system holds the steps on the algorithm to be performed. There is a mapping from the algorithm to the structure of the control part. In software systems the steps of -the algorithm are held in a memory in some encoded form (conventionally called an instruction). There is a mapping from the algorithm to these instructions. What does exist in hardware is a control, part that interprets the encoding of the algorithm in memory (which is called the primary memory\Mp of the interpreting control system). The interpretation of an instruction not only involves evoking the data and memory operations encoded therein, but determining the next instruction to be interpreted, which is also encoded in the instruction. All digital computers are examples of software systems, of course. But the basic principles of interpretation can be realized in many partial guises. The recent flourishing of microprogrammed systems is only a beginning of the variations that are possible.

What does one get with software -- what is its side of the trade-off? Let us ignore the issues of flexibility- and reuseability, whereby a single physical system can be programmed to do any task and reprogrammed to do another. These issues are indeed critical and they would make software systems worthwhile, even if their performance was always inferior. But the emergence of RT-level design in terms of RT modules (e.g., the Macromodules of Clark and the present PDP-16 RTM's) is an initial attempt to provide both these advantages at the hardware RT level. The primary focus of attention should be on the performing systems themselves: what advantages does one gain from a software system and what advantages from a hardwired system?

The basic advantage of a software system is the cost per step of control. A K(evoke), the basic control step in a hardwired system, costs 0.6. A 1024 word read-only memory costs 85. Assuming one word per control step, this leads to an estimated cost of .08 per step. The amount of control per word depends on the encoding, of course, but one can as easily get higher densities as lower, depending on the system. Furthermore, as the memory gets larger, the cost per word decreases. For Mp(16 bit/w;  $2^{16}$  words) the cost is about .06 per word. Thus, software control is potentially an order of magnitude less expensive than hardwired control. The hardwired interpreter also costs, but it is amortized over all the instructions. As these increase, the contribution of the interpreter to the cost per control step becomes negligible. In sum, if the algorithm is larger than a certain modest size, then it is simply impractical to realize it by any means except software. (We have ignored other aspects, such as reliability and power consumption, that also operate to keep hardwired systems within limits; they act in concert with costs of control.)

The basic comparative advantage of hardwired control is speed. The software system must engage in an entire loop of interpretation, fetching an instruction and decoding it. While there are many ways to make this efficient, it is still fundamentally more expensive than a K(evoke). In addition to the basic cost of the interpretive loop, the software system foregoes any of the possibilities of parallelism that we exploited earlier in the chapter. These can be incorporated in software systems (as in the Illiac-IV, Barnes et al., 1968, or the CDC Star, Holland and Purcell, 1971), but require quite complex controls and appear to be

outside the RT-level design domain of the present book. (No one knows for sure, however, since it has hardly ever been tried.)

There are other aspects to the software-hardware trade-off, but they are best seen in the light of particular examples. Thus, we will look at three

the original-case. However, the opportunities depend on the algorithm. Thus, since there is no arithmetic step associated with the control and a test for termination is required, there is little to be gained from running a straightline version of Figure 23 (though it can be done). However, one can attempt a cheaper implementation by time sharing a single DMgpa. This is shown in Figure 24. It uses two more control steps (taking more time) by substituting two memory cells for one DMgpa.

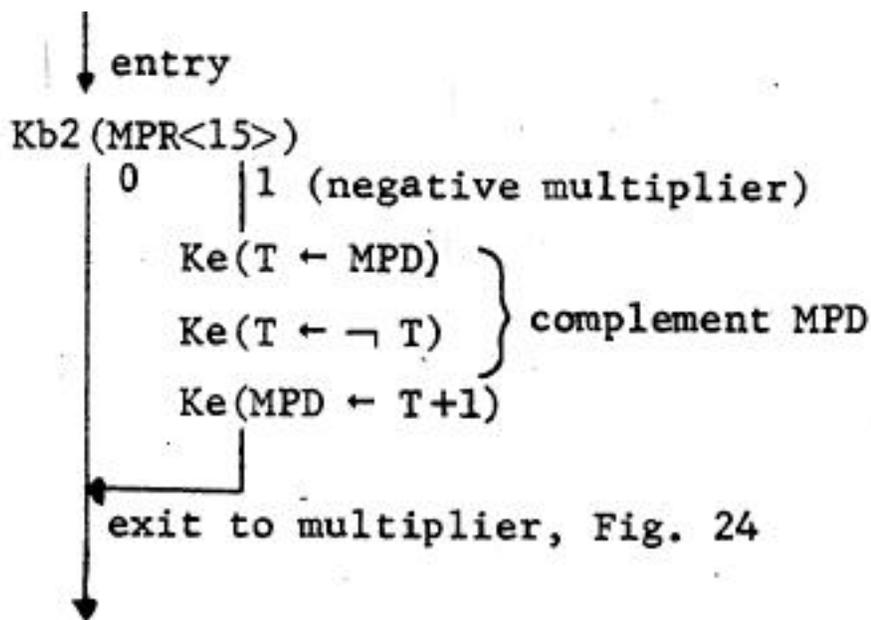


Fig. 25. Control part for signed multiplication.

The RP algorithm has several desirable properties that permit improvement of its specifications. It takes both its inputs in the same form, without requiring one to be shifted to the correct half word. It also will work on any 16-bit inputs, as long as the resulting product does not exceed 16 bits. If the latter happens it simply gives the correct low order 16 bits. Thus, none of the problems of specifying behavior in special cases arise, as they did for the original algorithm. Finally, with only a slight additional control, the algorithm can be extended to be used with signed numbers in a two's complement representation, whereas the schemes so far have permitted only positive integers. (See Chapter 5 for discussion of the representation.) We show the implementation of this in Figure 25, which involves only an additional initialization segment. The multiplier must be made positive in order to detect when to terminate. Therefore, the multiplier is complemented initially if it is negative.

## VARIATIONS IN THE IMPLEMENTATION OF CONTROL

One of the most frequently-discussed trade-offs is that between hardware and software. Reference to it occurs repeatedly in the critical discussion of any computer -- in analyzing why one feature was realized "in hardware" and some other feature was realized "in software". At the RT level of design this trade-off appears as the option of whether to build a "direct RT system" to do a given job or whether to obtain a minicomputer and simply program it. Since in all cases the functioning system is realized in hardware consisting of K's, D's, M's, etc., the first step is to be clear on what constitutes a software system (realized in hardware). Then we can discuss what is being traded for what. -

The essential feature distinguishing software RT systems from nonsoftware

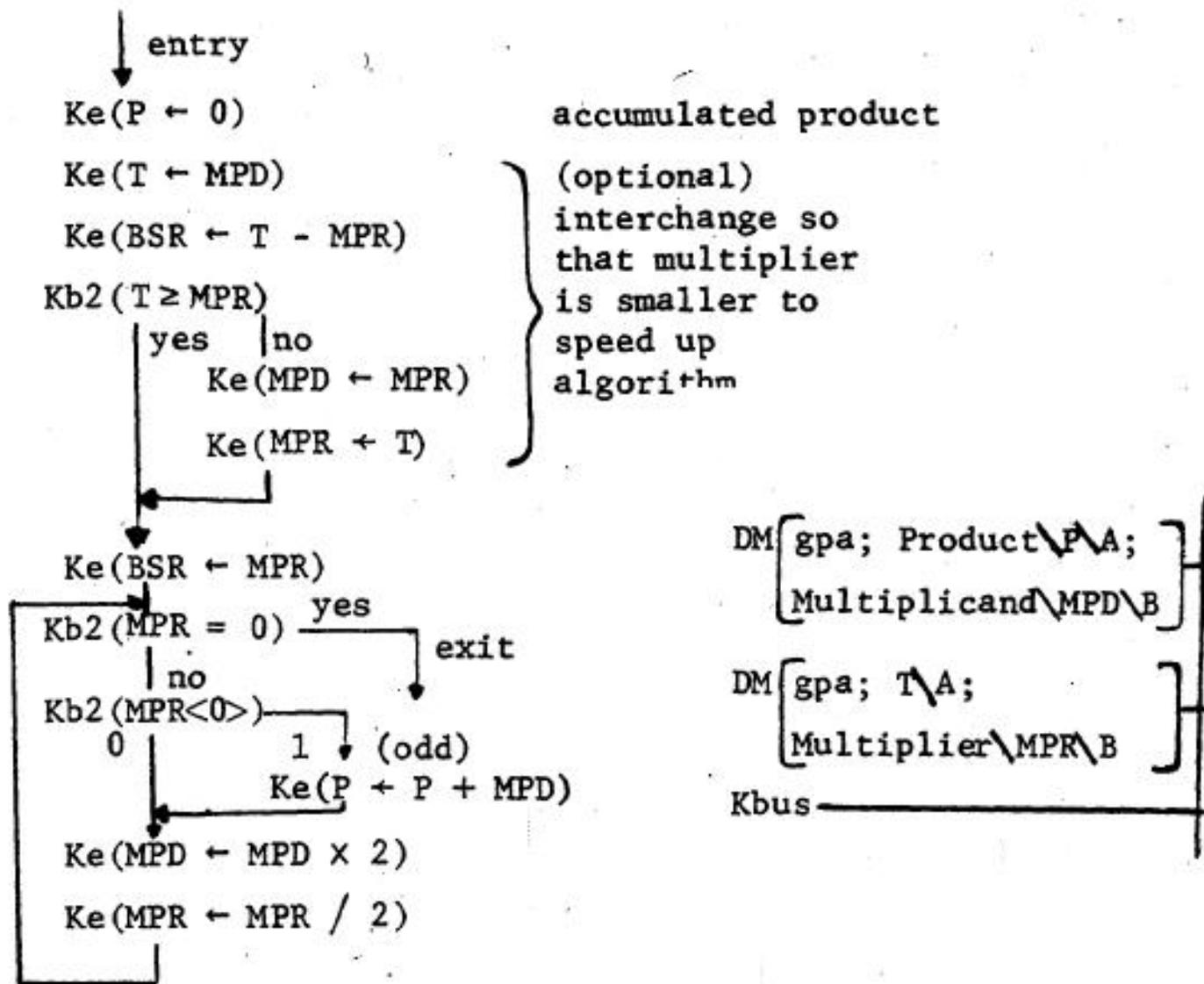


Fig. 23. RTM diagram of multiplier using RP algorithm.

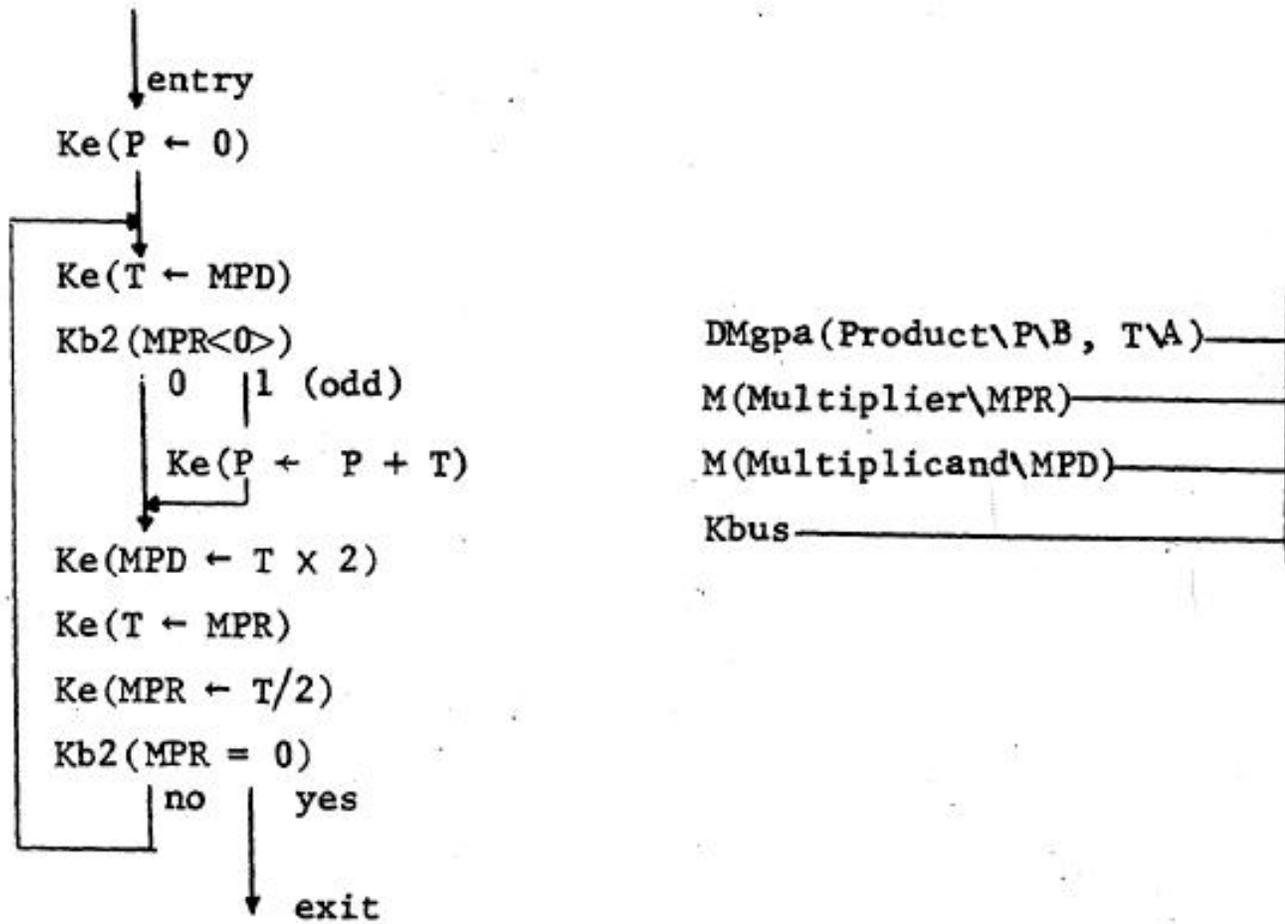


Fig. 24. RTM diagram of multiplier using RP algorithm, shared DMgpa.

Multiplier	Multiplicand	
	Multiplier odd	Multiplier even
67	12	
33	24	
16		48
8		96
4		192
2		384
1	768	

804

Multiplier	Multiplicand	
	Multiplier odd	Multiplier even
12		67
6		134
3	268	
1	536	

804

Fig. 22. Example of  $67 * 12$  multiplication using RP algorithm.

(the multiplier becoming zero), hence it is not necessary to keep a separate iteration count.

A system diagram based on the RP algorithm is given in Figure 23. The data part has roughly the same

structure as the original basic algorithm. While we eliminate one DMgpa for the iteration count, we must have a separate DMgpa for both the multiplier and multiplicand, so the implementation is still a 2-DMgpa, 1-Bus implementation. The multiplicand-multiplier interchange subprocess has been included to speed up the algorithm slightly. With the change of algorithm the performance has improved over the original. The multiplication step is now a little longer, but the number of steps is less (on the average), assuming small numbers.

Variations can be played on the implementation of the RP algorithm, just as in

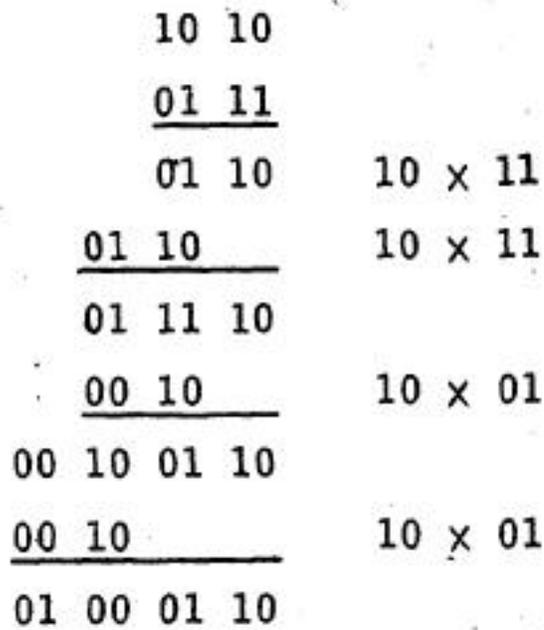


Fig. 21. RTM diagram for 8-bit multiplier using table look-up on half-word partial products.

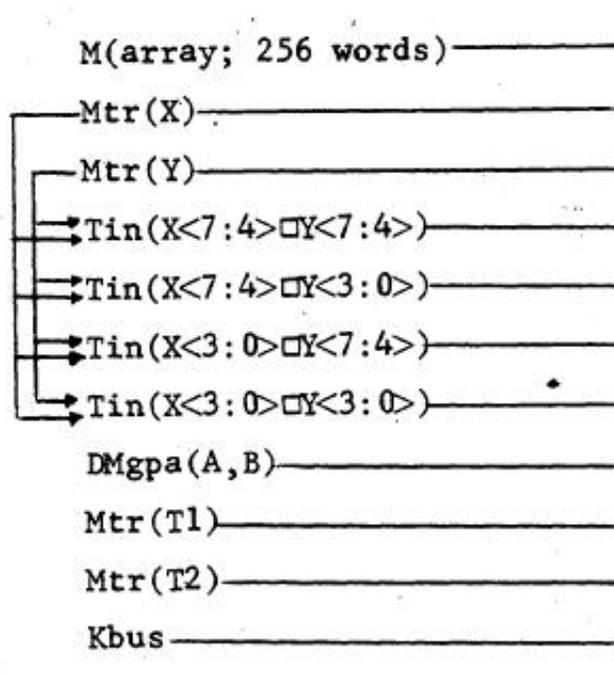
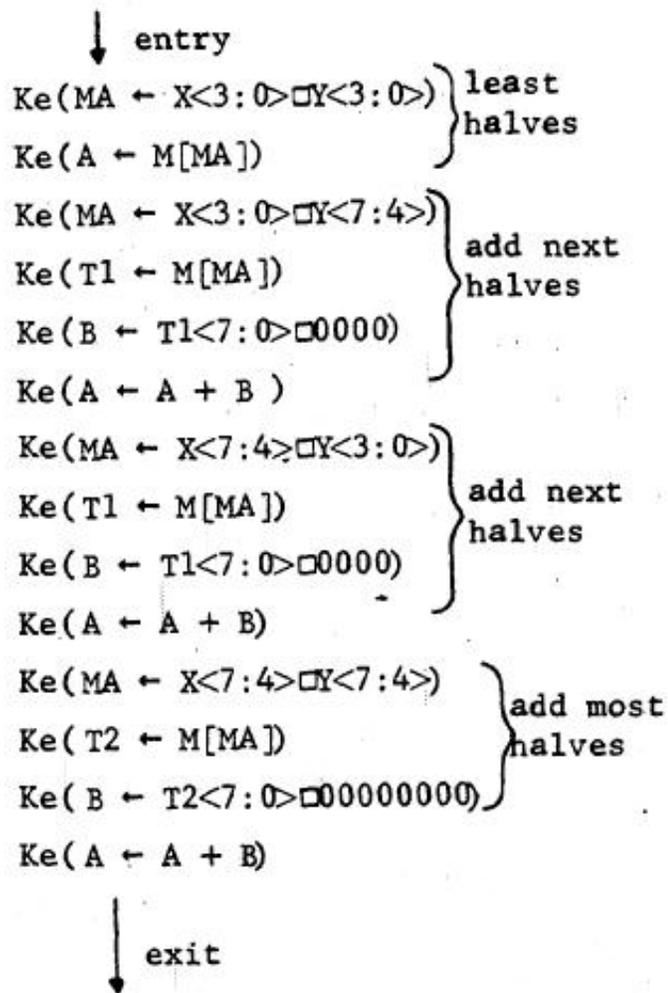


Fig. 20. Example of multiplication using half-word partial products.

bit stages, rather than the 1-bit stages of the original algorithm. Since the desired goal is to handle 8-bit numbers, the use of 4-bit multiplication implies that the table look-up is into a  $2^4 \times 2^4 = 2^8$  memory. RTM read-only memories are not currently available in the smaller size, but 1256 word read- write memory is available at 50, thus cutting the table cost a little bit.

Figure 20 shows the basic algorithm using an example:  $1010 * 0111$ . We use 2-bit components, so that each 4-bit number is made up of two 2-bit components, analogously to the 8-bit number being made up of two 4-bit components. However, there is no scaling up of the stages in the computation, since the components simply increase in size. Thus, three stages of computation are required, with shifts for each stage corresponding to the size of the component (2 in the example, 4 in the actual case). Figure 21 gives the RTM system for the 8-bit case with 4-bit components. Unlike the basic algorithm, there is no requirement for a test on the multiplier, since all cases are handled in a uniform way through the component multiplier. The problem with this implementation is that a significant amount of field shifting is required to manipulate the partial products. This raises the time and-cost both to a factor of 2 times the unwound loop case of Fig. 17. Normally when this scheme is used, all the shifting is done with wires and combinational circuitry. Here we have used the transfer fields of the  $M(\text{transfer})$ 's, and  $T$ 's specially wired to the Boolean outputs of the  $M(\text{transfer})$ 's.

This scheme with 4-bit components might appear to be like doing arithmetic in 4-bit digits -- i.e., serial hexadecimal arithmetic -- since the multiplication is done hexadigit by hexadigit. However, the additions are done in parallel in binary. If this, were not the case, the addition in each stage would have been a multistage process, making the scheme still more expensive.

#### ALTERNATIVE: THE RUSSIAN PEASANT'S ALGORITHM (RP)

Though we can only be illustrative, let us examine one other alternative scheme for multiplication. This algorithm, known in some parts as the Russian Peasant's Algorithm (Knuth, 1971) and hereafter abbreviated simply as the RP algorithm, works on two positive integers. It involves doubling the multiplicand and halving the multiplier, while accumulating only the multiplicands of odd multipliers to get the total product. Figure 22 provides an example of multiplying 67 by 12. The first row is obtained by placing the original multiplier (67) in the Multiplier column and the multiplicand (12) in the Multiplier-odd column because 67 is odd. The second row is obtained by dividing the multiplier by 2 and putting in the truncated value (33); the multiplier is doubled ( $2*12 = 24$ ) and placed in the odd side again since 33 is odd. The process is continued until the multiplier becomes 1 or 0. Then the product (804) is obtained by adding up all the entries in the Multiplier-odd column. The figure also gives the scheme run with 12 as the multiplier and 67 as the multiplicand.

The algorithm has a peculiar flavor when expressed in decimal notion, but it does not take much insight to see that it is well adapted to RTM implementation, and in fact works with the underlying binary representation of the two numbers. Note, this is similar to the algorithm of Figure 1. The number of

iterations in the algorithm is dependent on the number of 1-bits in the multiplier. It is likely that the numbers are small. Thus, assuming that the logarithms of the numbers are distributed uniformly, only an average of four iterations is required. In general the smaller of the two inputs will have fewer 1-bits (though not always), so the number of iterations can be reduced somewhat by taking the smaller number as the multiplier. The algorithm automatically generates its termination condition

obvious properties when it comes to computing all the things of interest in playing checkers.

## MEMORY VERSUS COMPUTATION: TABLE LOOK-UP

The one alternative algorithm that always exists for a simple input-output function is to store the values in memory and use a table look-up to obtain them. This was the strategy followed above in trying to obtain the logarithm and the antilogarithm. Of course, the values must be computed once in order to initialize the table and they must be stored in appropriate cells. If the function is to be used many times, as with multiplication, such one-time developmental costs are justified. But if a function is to be computed only a small number of times, compared to the total number of values in the table, then table look-up is a poor choice.

The prime determiner of the feasibility of a table look-up is the size of the table involved. This depends critically on the number of arguments, since values must be available for each combination of arguments, hence on the product of their ranges. Our arguments for the multiplication are 8-bit numbers, hence have a range of  $2^8$  or 256. This implies a reasonable sized table if only one argument is involved, as in the logarithm. However, if we apply the same strategy to the multiplication itself -- planning to do the whole job by table look-up -- then we need  $2^8 \times 2^8 = 2^{16} = 65,536$  cells. This is a large, hence expensive, memory, though it is still possible with available RTM components.

It would seem that the size of the table required settles the matter. But let us quantify the judgment. After all, table look-up is extraordinarily simple and fast, and its use here is even more advantageous than in the case above involving the logarithm. A  $2^{10}$  word read-only memory costs 85 (we clearly need no capability for writing). This is to be compared to a DMgpa at 31. Counting the extra control and Kbus's (at 14), we might consider that 2 DMgpa's are easily equivalent to 2110 words of memory. To compete with the pipeline which has 9 DMgpa's (whose operation rate of around 2 microseconds would be roughly equivalent to the table look-up scheme) implies a memory of 4 - 5000 words. But we require around 65,000 words. Thus, unless something can be done to decrease drastically the memory used, the table look-up scheme is out of bounds. This is true of a particular technology. If the relative costs of memory and data operations were to change significantly, then the solution might not be outlandish. There is nothing inherently wrong with using a 65,000 word table, if such tables are cheap enough and fast enough.

Can anything be done to decrease the amount of memory used? We can attempt to exploit some of the structure of multiplication. The most obvious is the commutativity of multiplication so that  $A * B$  is the same as  $B * A$ . Thus, we need only store approximately half the table. This would reduce us to a 32,000 word table which is still quite a way from the desired goal. Furthermore, we must now compute the address to the table from the inputs, rather than simply concatenating the two arguments A and B, which is all that is required in a pure table look-up. Such a calculation will take some precious time, and so reduce the effective gain from the memory reduction. Something much more drastic is needed.

Since any scheme must involve more processing than the pure look-up let us lower our sights to using 1000 words or less of memory. What could be done then? One could always do a smaller multiplication.  $2^{10}$  words permits numbers of size  $2^5$ , since  $2^5 \times 2^5 = 2^{10}$ . Thus one can multiply 5-bit numbers rapidly. This suggests breaking down the total multiplication into multi-

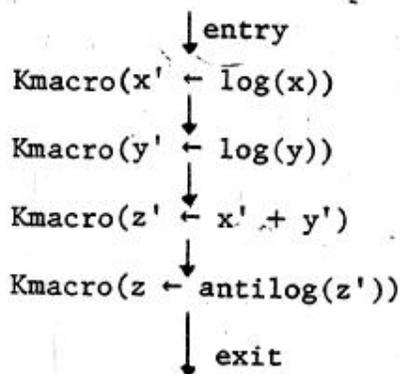


Fig. 18. Control part of incorrect 8-bit multiplier using logarithms.

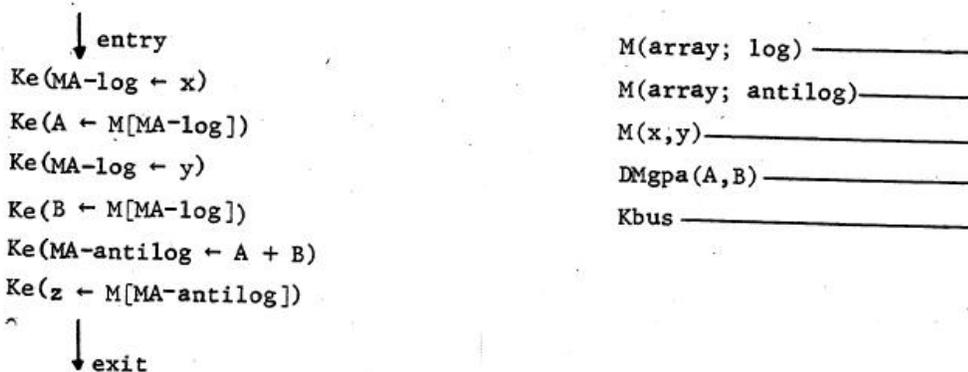


Fig. 19. RTM diagram of incorrect 8-bit multiplier using logarithms.

This is a unique representation up to the maximum number expressible by the basis (here  $2 * 3 * 5 = 30$ ). Both addition and multiplication can be expressed as simple algorithms, but it turns out to be difficult to compare numbers. However, a good deal of investigation has gone into residue arithmetic as a possible alternative internal representation of numbers.

For RTM's none of the above representations seem to be an alternative to the binary representation. Our example of multiplication was ill conceived to illustrate gains to be made by changing representation. If we had chosen the example of checkers (recall the two representations in Chapter 1), the story would have been quite different. Here the basic hardware system is not already specialized to the representation of checkerboards (as it is to numbers). Each different internal representation of the board has distinctly different and non-

only because of external constraints of human readability. Internally, they simply add problems, rather than provide processing options.

The first alternative that comes to mind is the logarithm. Given that numbers were represented internally as their logarithms, then multiplication would simply be addition of logarithms:

$$\log(X * Y) = \log(X) + \log(Y)$$

This yields the result in the form that is wanted, i.e., the logarithm of the product, which is in the appropriate logarithmic representation. The difficulty, of course, is that if any additions or subtractions are to be done on the resulting product, then the representation is impossible -- direct algorithms for adding two numbers represented as logarithms (and not involving multiplication) do not exist. Rather, it would be necessary to convert back to the numbers (by taking the antilogarithms) and then add.

Thus, one would use logarithms only when the total algorithm involved only multiplication (and division). If this were the case, the transformation to logarithms would undoubtedly be carried out at the level of the algorithm itself and would not be seen as occurring in the RT implementation.

It is worth considering briefly the feasibility of developing an implementation of multiplication that input and output numbers in binary, but operated internally in terms of logarithms. The basic scheme is shown in Figure 18. It requires a conversion from numbers on input and from the logarithm back to numbers on output. If these conversions take much time and hardware, then there can be no advantage in the scheme. Given the small amount of hardware in the basic multiply implementation itself, it would seem that the only possibility lies in a fast, though possibly expensive, scheme that would produce a fast total algorithm, thus competing with some of the parallel algorithms that take considerable hardware.

The above considerations dictate the use of table look-up, which is the only fast scheme for arbitrary algorithms. Let us suppose that we use sufficient memory so that each table look-up is done in the most efficient way possible, by a direct memory access with the argument as address. This is certainly possible for taking the logarithm of the inputs, where it requires a table of size  $2t8$ . It is much more complex, however, on the output side, where the argument is a 16-bit logarithm of the product. Still, continuing with the assumption, we get the basic implementation shown in Figure 19. It can be seen that the time is about  $\frac{1}{2}$  multiply steps, and the cost is  $256 + 16,384$  read-only memory cells. Thus, it yields less than a factor of 2 in speed over the basic algorithm and provides little with which to compete against the array and pipeline parallel schemes. (Note that it is also incorrect as it stands because of the 0 factor case.)

Thus, we can finally lay to rest the use of the logarithm. One could have predicted the outcome, perhaps, since multiply yields a simple algorithm and the mere introduction of a function such as the logarithm bodes inescapable complexities. We carried out the analysis a bit to indicate the regard one should have for the actual examination of alternatives, rather than their dismissal on general grounds,

There exist other representations of numbers. For instance, a number can be expressed as the product (with repetition) of its prime factors. Multiplication then reduces to summing of exponents. The complexities here turn out to be worse than for logarithms.

Numbers can also be represented as the residues taken modulo a basis set of primes. The number 18 for instance would be represented as the vector (0,0,3) where the basis is (2, 3, 5), since:

$$18 = 0 \pmod{2}, 18 = 0 \pmod{3}, 18 = 3 \pmod{5}$$

The basic cause of the high performance can be traced to a combination of three things. First, DMgpa is an expensive module, so its replacement can support a substantial amount of control. Second, the number of iterations was only eight, so the total size of the unwound control was small. Third, since the number of steps in the iteration loop was fixed, the test of termination was required only because the loop was introduced. Normally, each stage in the unwound control would not only have to evoke the basic processing step, but to test for termination. This increases the cost of the control step considerably. Removing these last two conditions would increase the cost of the control beyond the the saving from a DMgpa, returning the situation to a trade-off between speed and cost.

## VARIATIONS IN THE ALGORITHM

Throughout the chapter we have taken as given the algorithm for doing multiplication. But clearly there are many ways to perform multiplication. From an investigation of the properties of one algorithm essentially nothing can be known about the others. It might seem simple to define the essential information processes involved in an operation as basic as multiplication, where the function is easily defined mathematically (by axioms), free of any particular way of computing it. But no such results are yet available in computer science, though the topic currently is highly active. Indeed, 'recently new algorithms have been found for the multiplication of two matrices which take less time than anyone would have predicted earlier.

The upshot is that we cannot offer any systematic view for finding-alternative algorithms or knowing that an algorithm in hand is as good as can be expected. The best we can do is illustrate issues. This is not as crippling as it might sound. Design at the RT level is very much the implementation of algorithms defined externally to the design and not, like programming, very much the creation of new fundamental algorithms. The reasons for this lies in the relative simplicity of the algorithms realized at the RT level compared to those realized routinely by programming (which can have thousands of instructions).

## ALTERNATIVE REPRESENTATIONS

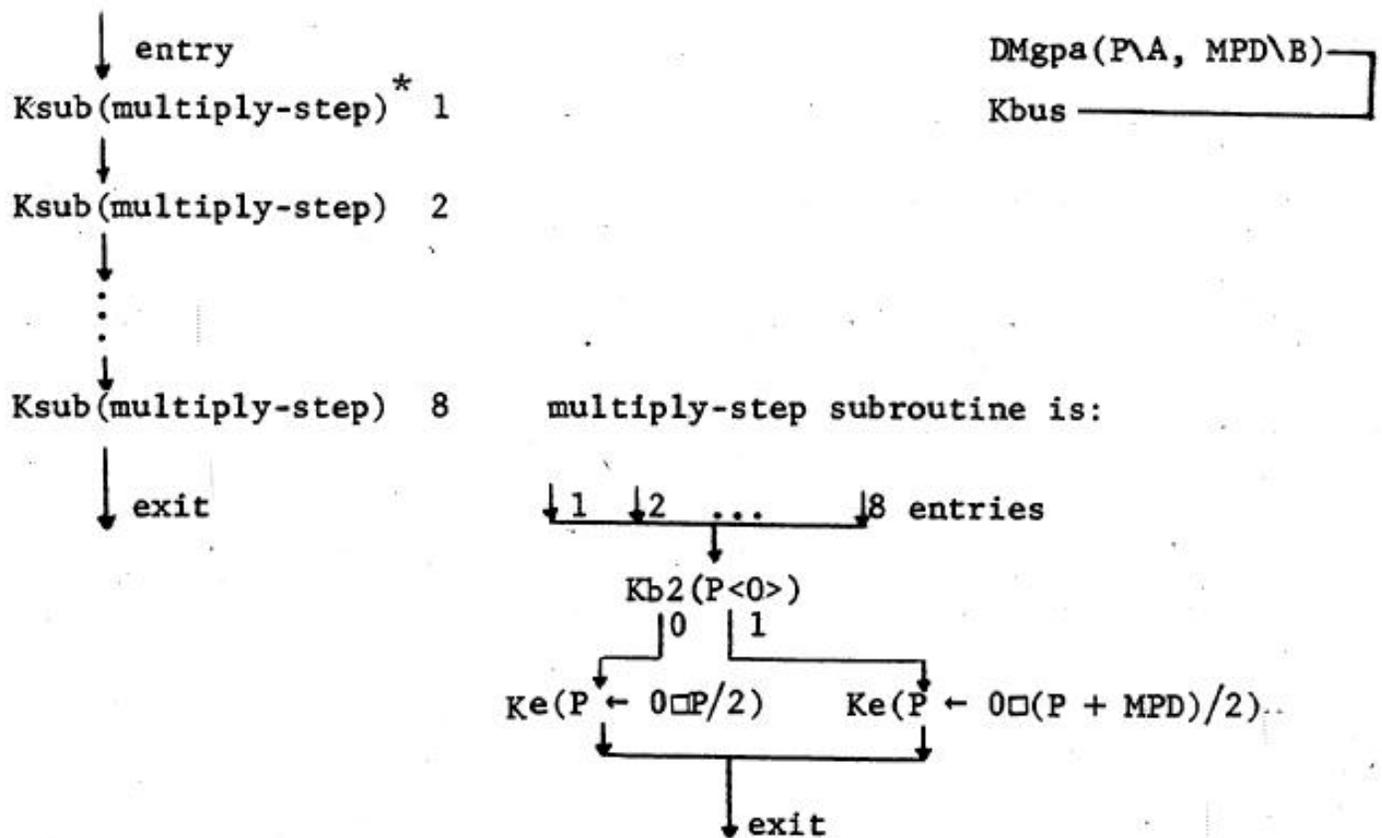
In general an algorithm can be realized in RTM's with any basic set of data operations on a set of data structures capable of holding the required information. At one level, the data structure and data operations are completely fixed, given by the RTM system. But, in fact for any realistic information processing task there is a design issue of how to represent the information of the task in terms of the basic bit vectors of the RT level. Though occasionally the choice of data representation and algorithm are separate design decisions, usually they are tightly coupled. Given the algorithm, one chooses just that data representation that makes the processing efficient. To do otherwise invariably generates needless operations of extraction and encoding, just to deal with the inappropriate data representation.

Conversely, if the data representation is selected first, then it determines a class of algorithms as the natural candidates to do a task. Thus, the data representation can be used as means of generating possible

alternative algorithms. Applying this to multiplication leads to searching for the different possible representations of numbers. It is clear that various familiar encodings of numbers, such as binary coded decimal, will not provide interesting alternatives in comparison with the basic binary coding already used. They exist

multiply step, using Ksubroutine. Alternatively, a form of open subroutine organization can be used in which 8 groups consisting of a K(branch), two K(evoke)'s and a K(serial-merge) are interconnected. The former requires less control components, but the latter is somewhat faster.

The reader will discover in Chapter 7 that with 1972 PDP-16's two Ksub's that evoke the same subroutine in series are not permitted; thus the 8-step straight line implementation using Ksub's cannot be built. However, a simple modification of Ksub, presented in Chapter 7, corrects this problem. Alternatively, placing a Kevoke with a dummy operation (e.g. BSR<-0) between the Ksub's will solve the problem, but this will slow the system down.



\* Note: These subroutines could alternatively be macros with 1 Kb2, 2 Ke, and 1 Ksm.

Fig. 17. RTM diagram of 8-bit multiplier with open and closed subroutine implementations.

The implementation of Figure 17 actually dominates the basic solution of Figure 4, being both faster and

less expensive. This shows that one cannot always depend on the general rule, which we have been illustrating continually, that to explore a design space is to explore the trade-offs that exist between the various objectives (here speed and cost). Often one finds designs which are both slower and costlier. Occasionally, as here, one finds designs which are faster and cheaper.

multiplication had been part of a larger system with several variables to be stored, then an M(16 word; scratchpad) could have been used and the savings would have been even greater. Conceivably, the larger memory might already be required by the other demands of the system, so that the additional memory cost for the cheap multiply would in fact be zero.

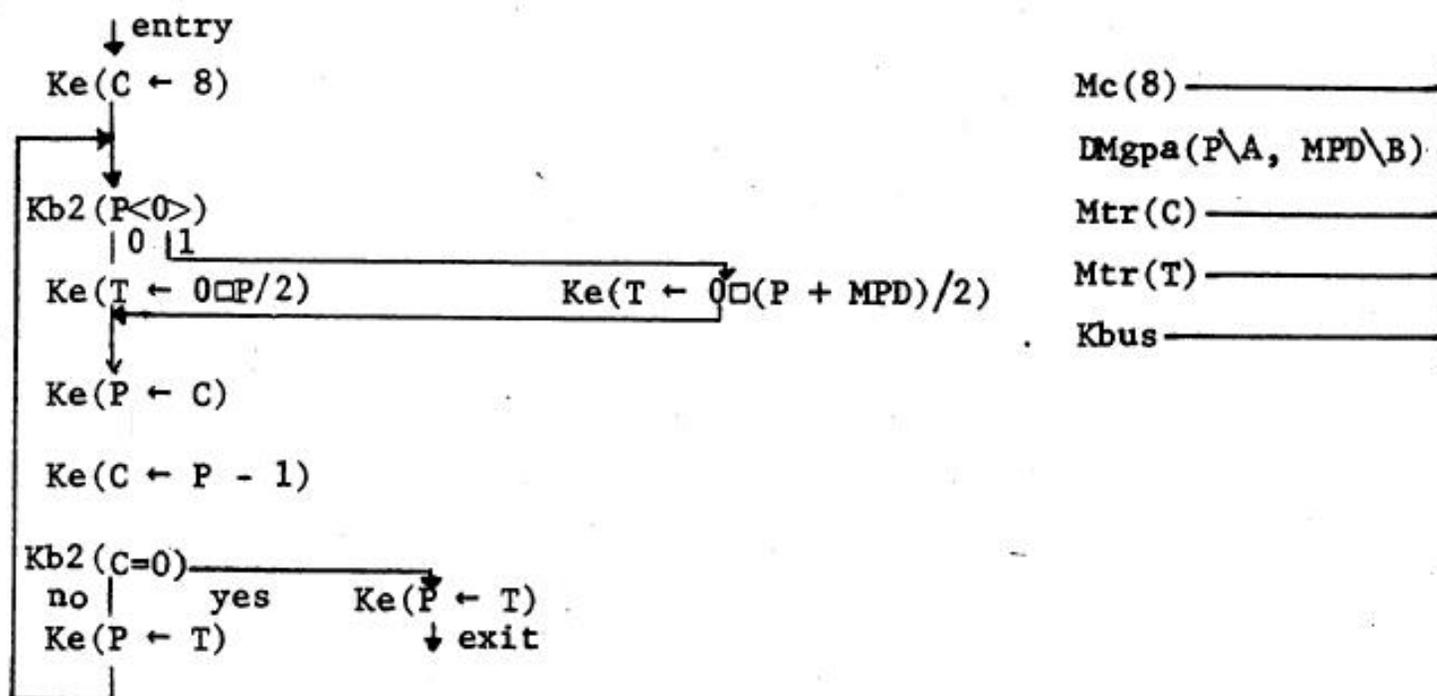


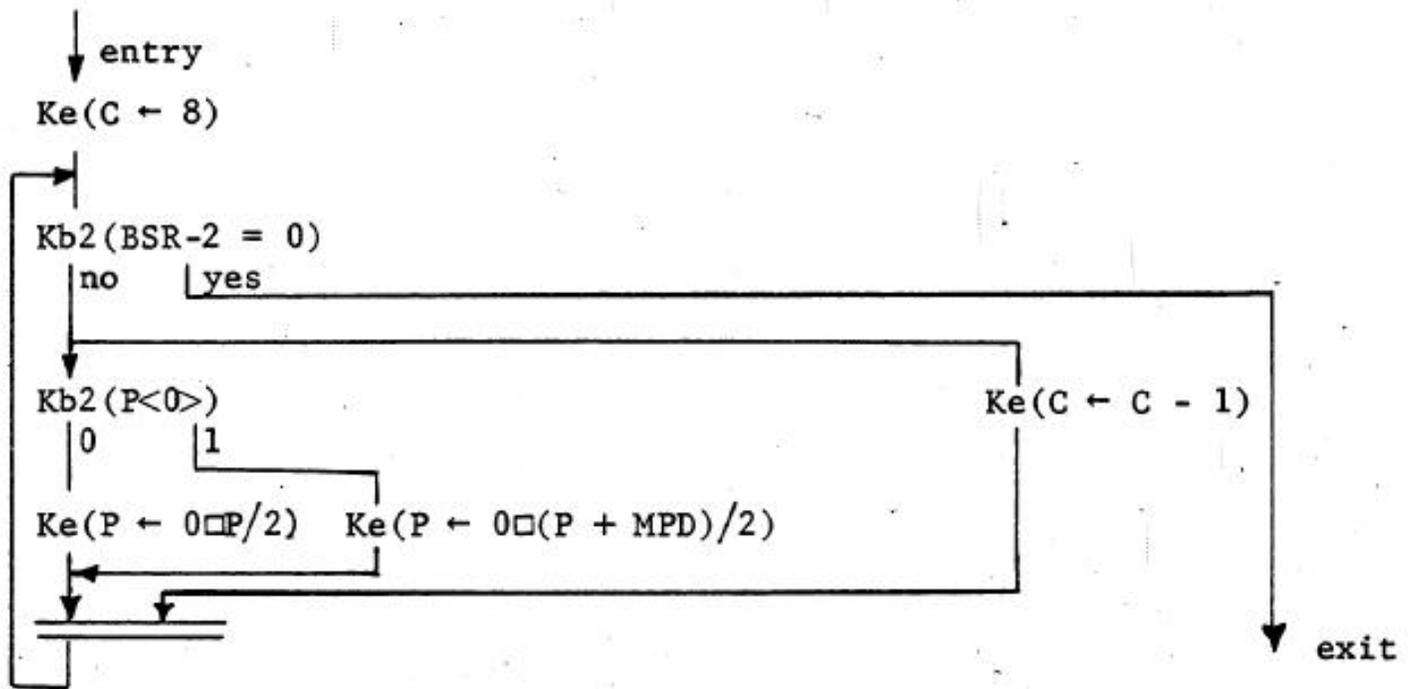
Fig. 16. RTM diagram of 8-bit multiplier, shared use of a common DMgpa.

## REDUCTION OF THE CONTROL PART: UNWINDING LOOPS

In general, it is difficult to make changes in the control part of an algorithm per se (i.e., not associated with some other change in the data part, such as in all the prior examples). Given that the algorithm and the data part are both fixed, then each module in the control part tends to perform some specific step in the algorithm, which function cannot be eliminated. Any attempt to provide an alternative way of performing such a function would also involve control and would negate any conjectured savings. (Try sharing Ke's or Kb's.)

One exception to this that arises with great regularity is avoiding control operations associated with an iteration by unwinding the loop. This is a standard technique in programming, where it forms part of a general space-time trade-off. Some of the time spent controlling the loop can be avoided if the program is written as straight-line code with the requisite number of copies of the loop-body (which take extra memory). The same philosophy exists in RT-level design, where the trade-off is in the saving of hardware to do the loop calculations versus the extra control, which must be replicated for each iteration.

Figure 17 shows the application of this to multiply. The control loop has been removed and with it the DMgpa(C). The steps have been strung out as a sequence of 8 subprocesses. These can be defined as subroutine calls on the



- Fig. 15. RTM diagram of 8-bit multiplier, concurrent implementation, test at beginning of loop.

## FACILITY SHARING

As in the earlier section, we may not be concerned about obtaining more speed (by paying more hardware). Rather, we can be concerned about doing the operation more cheaply, being prepared to give up some speed in the process. Such a decision need not occur because of any general weighing of speed versus cost in an overall objective function. It can occur just as well because we are working on a part of a larger system that is speed critical or speed non critical. For instance, it is quite possible that the multiplication in question is being done in parallel with another process that takes much longer. Consequently, no increase in speed will make any improvement at all and no decrease in speed (up to some limit) will detract from overall performance at all. Hence, one wants to implement the cheapest version of multiply possible consistent with the lower limit on speed.

In such a case, the general strategy is to look for facilities to share. In the present case, taking Figure 4 as the defining RTM flowchart, we observe two DMgpa's, one for control and one for the multiplication step. These become candidates for sharing. We can anticipate that some additional control will be necessary to share facilities (as it was with subroutines, for instance). But DMgpa's are relatively expensive, so that we can undoubtedly afford additional control and still reduce cost.

The present case turns out to involve more than control, however. The DMgpa's perform two functions: the arithmetic operation and the memory operation. The arithmetic operation can be shared, but the

memory cannot, since the registers of both DMgpa's are fully occupied throughout the computation with holding the relevant data. Thus, to share the arithmetic part, by having only a single DMgpa, implies that one must introduce additional modules to provide the memory. Figure 16 shows the resulting implementation, using two M(transfer register)'s. Since these cost roughly 1/3 of a DMgpa, the resulting system is still somewhat less expensive overall, even with the additional Ke's for control. If the

stream) detects both that the two streams have been completed and also that the C-stream found that  $C=0$ , hence that the computation should terminate. This two-Bus parallel implementation is approximately twice as fast as the one-Bus serial version, but requires extra hardware in the form of an extra Kbus and two K(parallel-merge)'s.

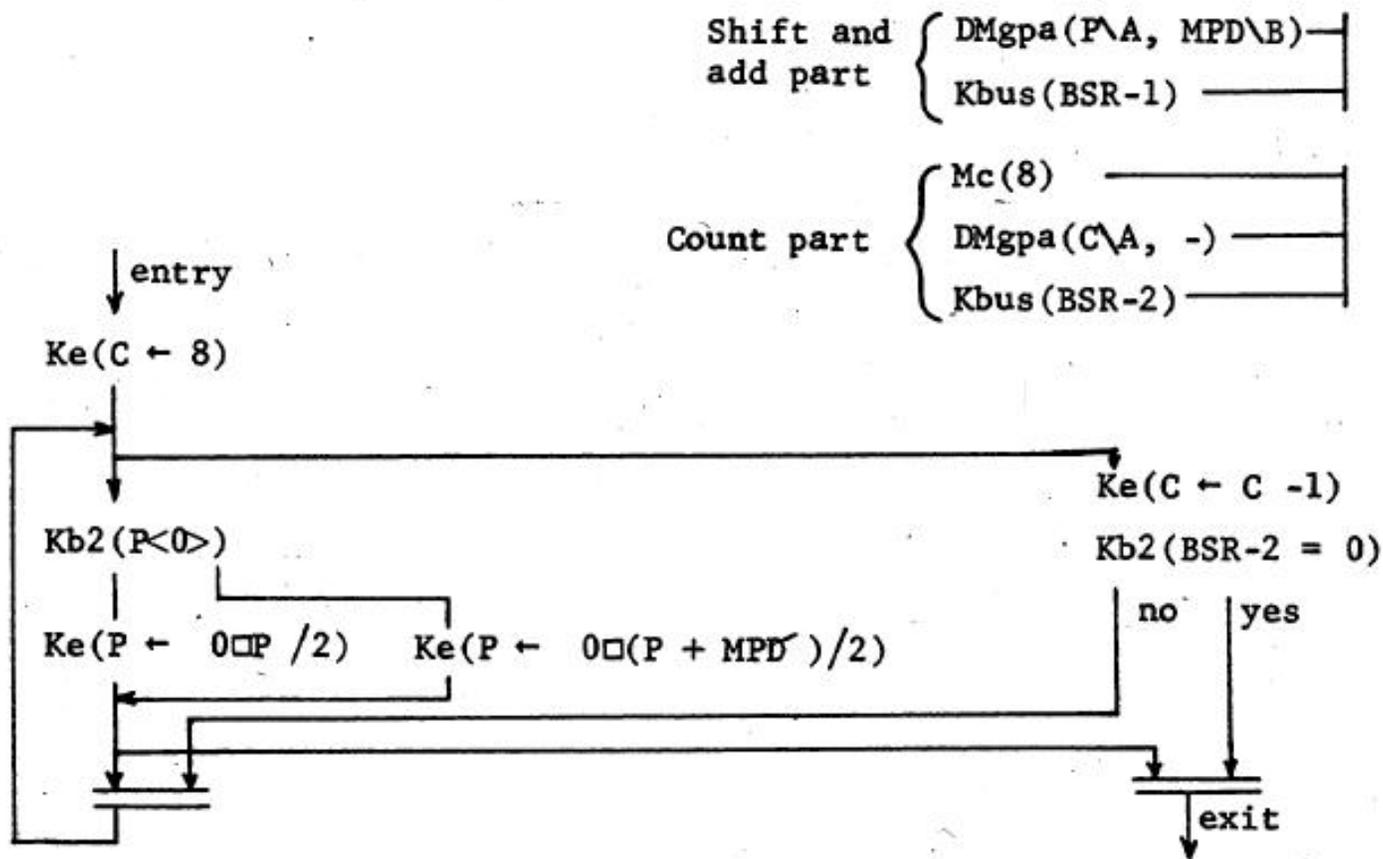


Fig. 14. RTM diagram of 8-bit multiplier, concurrent implementation, test at end of loop.

One can attempt to avoid some of the extra cost by making minor adjustments in the control structure. For instance, one of the Kpm's can be eliminated if the loop control is divorced from the synchronization. This can be done by putting the test prior to entering the loop, as shown in Figure 15. Unfortunately, this structure requires one additional control step in the P-MPD control stream, thus slowing up the system slightly.

The type of parallelism exhibited here is really the general case -- functionally diverse computations that

do not depend on each other for data (or do not affect data used by each other, such as updating in midstream) are done by independent machinery. Synchronization, when it is finally required, is forced by means of parallel-merges. This is often called concurrency, although the term parallelism itself is often used as well; terminology is not yet standardized. Our example again illustrates a rather general rule: that there is always a tradeoff between speed and hardware. The increase in speed (in the two-Bus system) costs hardware; the attempt to avoid some of that cost has to give back some of the speed gain.

actual facts on which a design choice must be based. But the alternatives almost always exist to be considered, since they reflect pervasive qualitative characteristics of RT systems: the occurrence of an operation more than once in a larger system; the prevalence of array data; and the existence of serial computation.

Having set out to study the design of RTM systems to multiply, it may seem odd that we concentrated first on the use of multiply as a subsystem within a larger algorithm, rather than simply taking multiply as an autonomous computation and considering alternatives that arise for its internal structure. Our reason for so doing is to counteract the natural tendency to consider multiply as a new unit, just as if a DM(multiply) had been defined as a basic RTM module. This latter is a powerful notion and in programming it is almost universally the right thing to do. In RT design it is also appropriate to a degree. Every designer should have a personal library of such schemes, designed, checked out and documented. However, as the present section shows, the larger system should always be taken into account in determining the way the subsystem is to be implemented. In the examples given above the internal structure was substantially modified in response to whether to implement the algorithm with one or another degree of parallelism. Thus, the level at which the unit should exist is that of the conceptual RTM flow chart, such as Figure 9. In fact, one could even write simply K(...) for Kmacro(...), indicating that no decision had been made on how to realize the computational function.

With this fundamental point made, we can henceforth restrict our concern to the function of multiplication considered in isolation.

## VARIATIONS IN IMPLEMENTING THE BASIC ALGORITHM'

In this section we ask what alternatives for implementation regularly occur if we take a given algorithm, such as that given by Figure 3 for multiplication, and consider the calculation of an output from a single input. The basic option is given by a direct one-to-one mapping of the algorithm into an RTM flowchart; it was already presented in Figure 4. What other alternatives exist?

### GENERAL PARALLELISM: CONCURRENCY

Given that a certain set of operations has to be performed, speed comes from doing some of them at the same time -- in parallel. What prevents simply doing everything at once is that some operations depend for their inputs on the outputs of other operations. Such operations are forced into a sequential relationship. Thus the opportunity for parallelism arises from operations that do not depend on each other. Examination of Figure 3 shows that the calculations on C are independent of the calculations on P and MPD, providing only that the

two sets of calculations remain locked together as a whole (only one decrement of C per multiplication step). Thus, these two sequences can be separated into two parallel streams.

Figure 14 shows the implementation, with separate data-memory parts for shifting and adding, and for counting. The parallelism begins at the output of the K(serial merge) following  $K_e(C < -8)$ , control branching into two paths. Control must be synchronized again at the end of the sequences. This is done by a K(paralled-merge). Actually, two Kpm's are required. The left hand one (under the P-MPD control stream) detects both that the two streams have been completed, and that the C-stream found that C was not 0, hence that another iteration of the loop is required. The Kpm at the right (under the C control

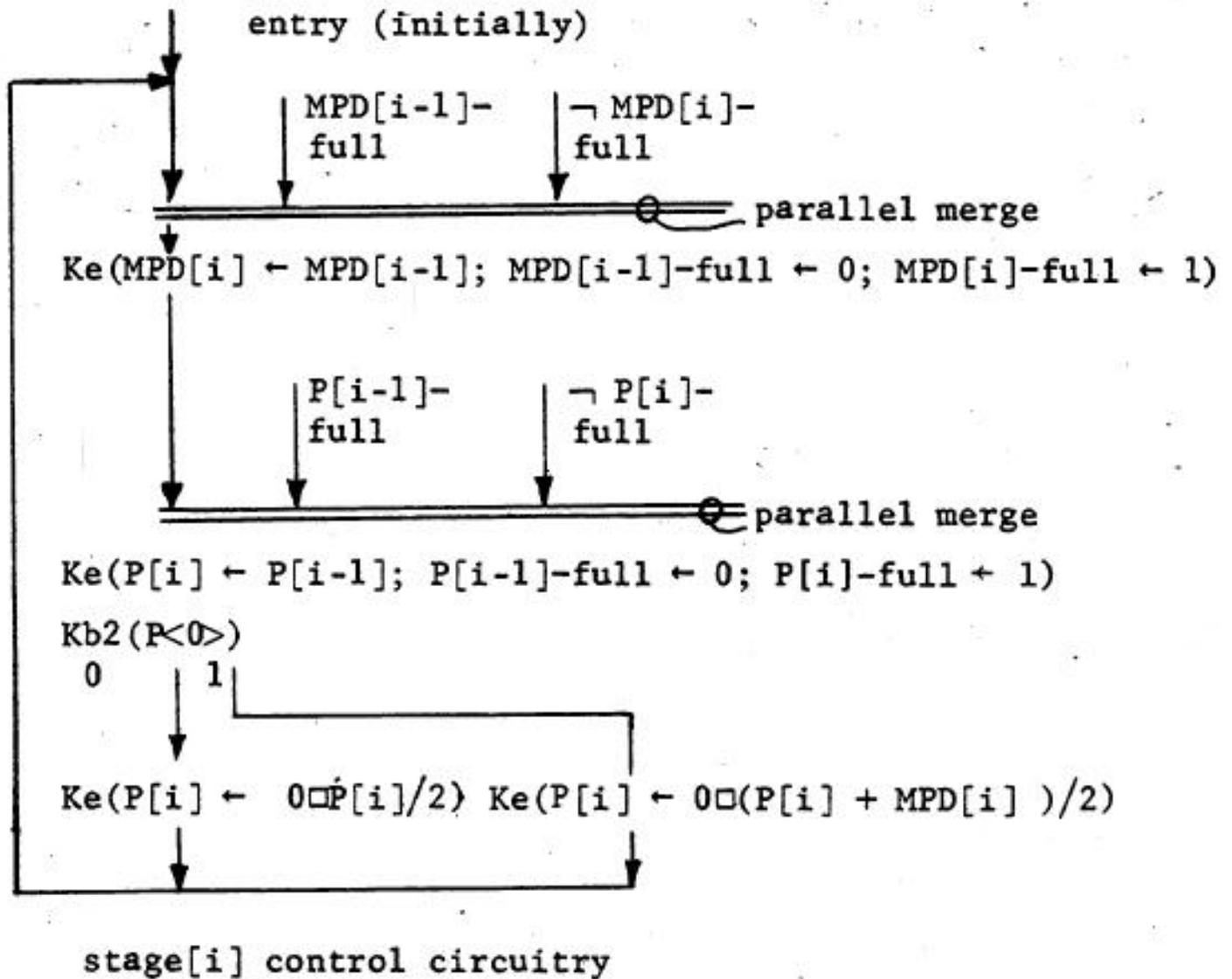


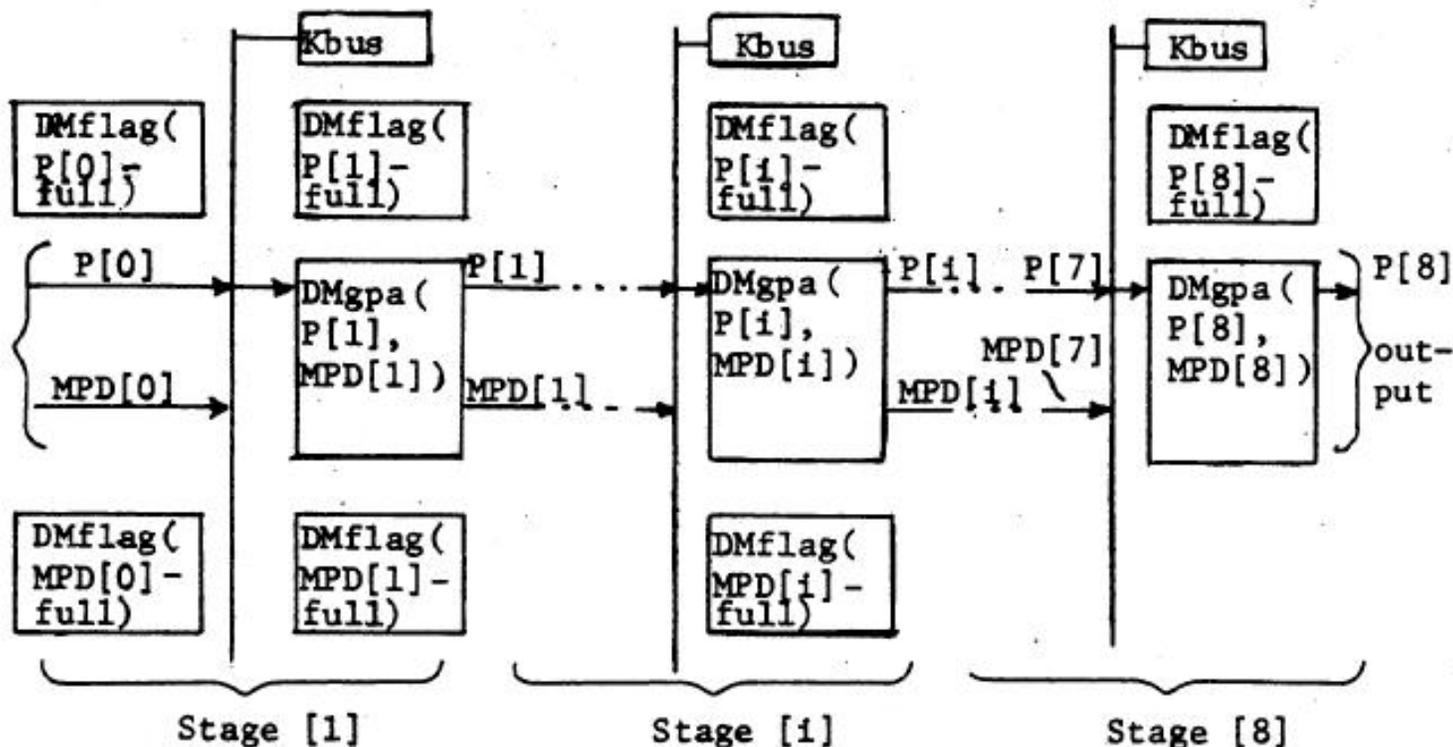
Fig. 13. Control part for 8-stage pipeline multiplier.

[RREPEAT OF 00000140.htm]

below that predicted from the operation time. A common example in memory technology occurs in magnetic core random access memories, which deliver the contents of the addressed work in, say, .9 microseconds, but require another .4 microseconds to finish the rewrite cycle. This gives an access time of .9 and an access rate of 1/1.3, equivalent to 770,000 accesses per second, compared to a "predicted" 11.9 or 1,110,000 accesses per second. Often only one of these two performance measures is important, but only an analysis of the larger system can tell.

## SUMMARY

We have devoted this section to examining how a subsystem, such as multiplication, might occur in a larger algorithm. Several alternatives almost always make themselves felt at the RT-level of design: minimizing the hardware by using subroutines; mapping each occurrence into separate hardware (macros); speeding up the algorithm by pipelining; and maximizing speed by complete duplication of hardware for independent operations (array parallelism). These alternatives provide a set of trade-offs between hardware (cost) and speed, though where each alternative stands in the trade-off (how much hardware for how much speed) depends on the details of both the subsystem and the larger system in which it occurs. Only detailed analysis can reveal the



**[REPEAT OF 00000139.htm]**

Fig. 12. Data part for 8-stage pipeline multiplier.

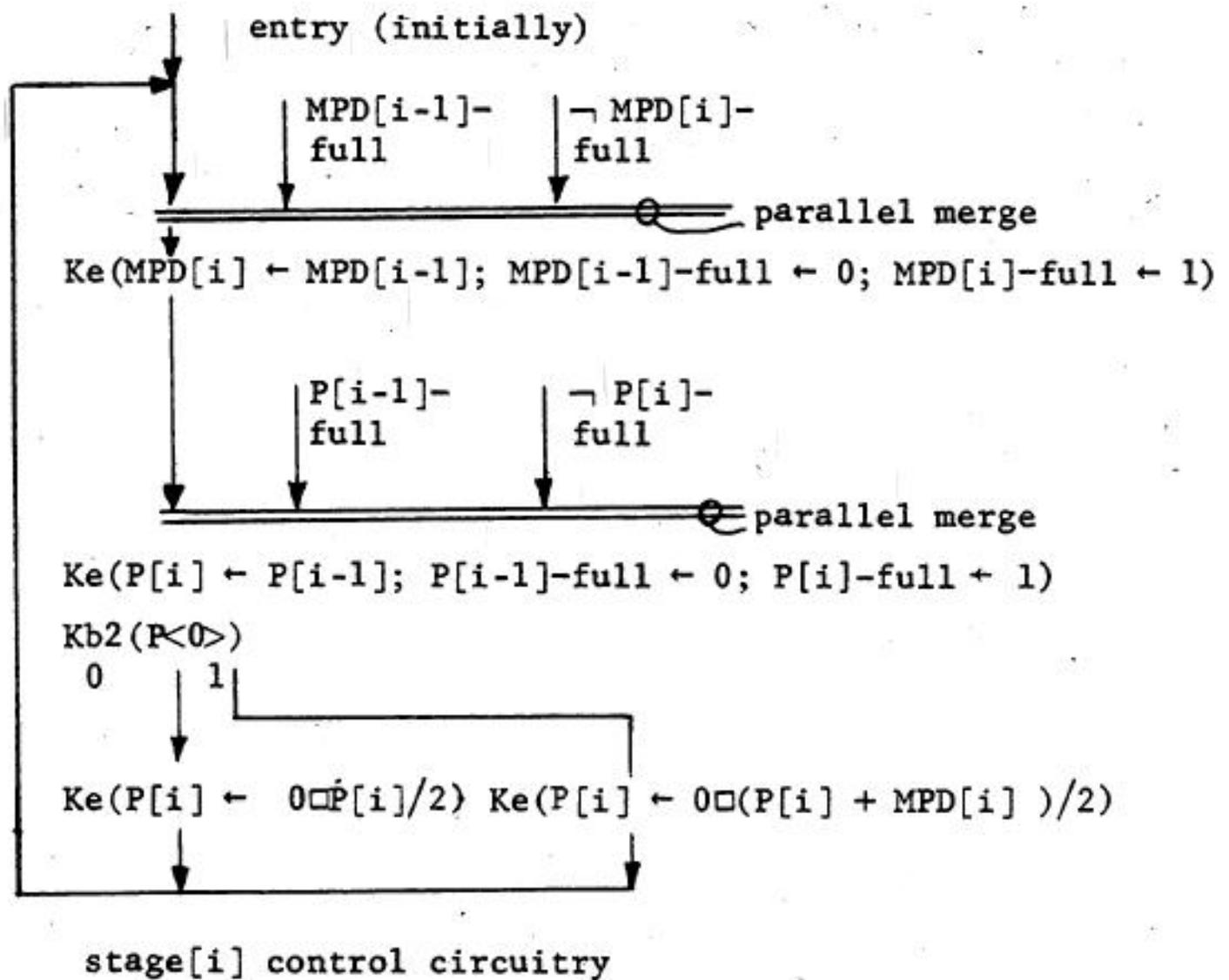
transfers data and of setting the flags of stage  $i$  to 1 after its computation is complete. Each stage operates as soon as its input data is available and its just-computed outputs have been used. Thus, the whole pipeline moves as fast as possible, compatible with not mixing up the computations.

The alert reader will have noticed that since the flag outputs are positive logic and the  $K_e$  control flow outputs are negative logic, the complements of the named flag outputs are the ones that must enter the  $K(\text{parallel merge})$ 's. Even so, there are some problems in using  $K(\text{flag})$ 's to enable  $K(\text{parallel merge})$ 's. Chapter 7 must be read to discover why. After reading Chapter 7, return to this example and, if necessary, alter it so that the  $K(\text{parallel merge})$ 's are enabled properly.

The pipeline structure is considerably more complicated than the array case from a control viewpoint. The limit to the parallelism is set by the structure of the algorithm -- here, 8 stages -- rather than by the size of the data array being processed. The time to perform a given stage is 2.7 microseconds, computed from the control sequence of Figure 13, so that the time to get an answer is  $8 * 2.7 = 21.6$  microseconds. However, providing the data is fed into the pipeline rapidly enough, the system will finish a'

multiplication every 2.7 microseconds.

The pipeline makes clear that one must always consider two distinct performance measures for a process. One is the time taken to produce a result, measured from the time at which the input, is presented -- this is the operation time. The other is the rate at which results are produced -- this is the operation rate. In ideal serial systems the operation rate is simply the reciprocal of the time for each operation:  $1/\text{operation-time}$ . But in general the two can vary independently. In parallel systems, as we have seen, the operation rate can be increased many fold. To do so usually implies an appropriate organization of the input data. But also, systems sometimes produce their outputs prior to the time they are prepared to accept the next input, thus reducing the operation rate



■ Fig. 13. Control part for 8-stage pipeline multiplier.

below that predicted from the operation time. A common example in memory technology occurs in magnetic core random access memories, which deliver the contents of the addressed work in, say, .9 microseconds, but require another .4 microseconds to finish the rewrite cycle. This gives an access time of .9 and an access rate of 1/1.3, equivalent to 770,000 accesses per second, compared to a "predicted" 11.9 or 1,110,000 accesses per second. Often only one of these two performance measures is important, but only an analysis of the larger system can tell.

## SUMMARY

We have devoted this section to examining how a subsystem, such as multiplication, might occur in a larger algorithm. Several alternatives almost always make themselves felt at the RT-level of design: minimizing the hardware by using subroutining; mapping each occurrence into separate hardware (macros); speeding up the algorithm by pipelining; and maximizing speed by complete duplication of hardware for independent operations (array parallelism). These alternatives provide a set of trade-offs between hardware (cost) and speed, though where each alternative stands in the trade-off (how much hardware for how much speed) depends on the details of both the subsystem and the larger system in which it occurs. Only detailed analysis can reveal the

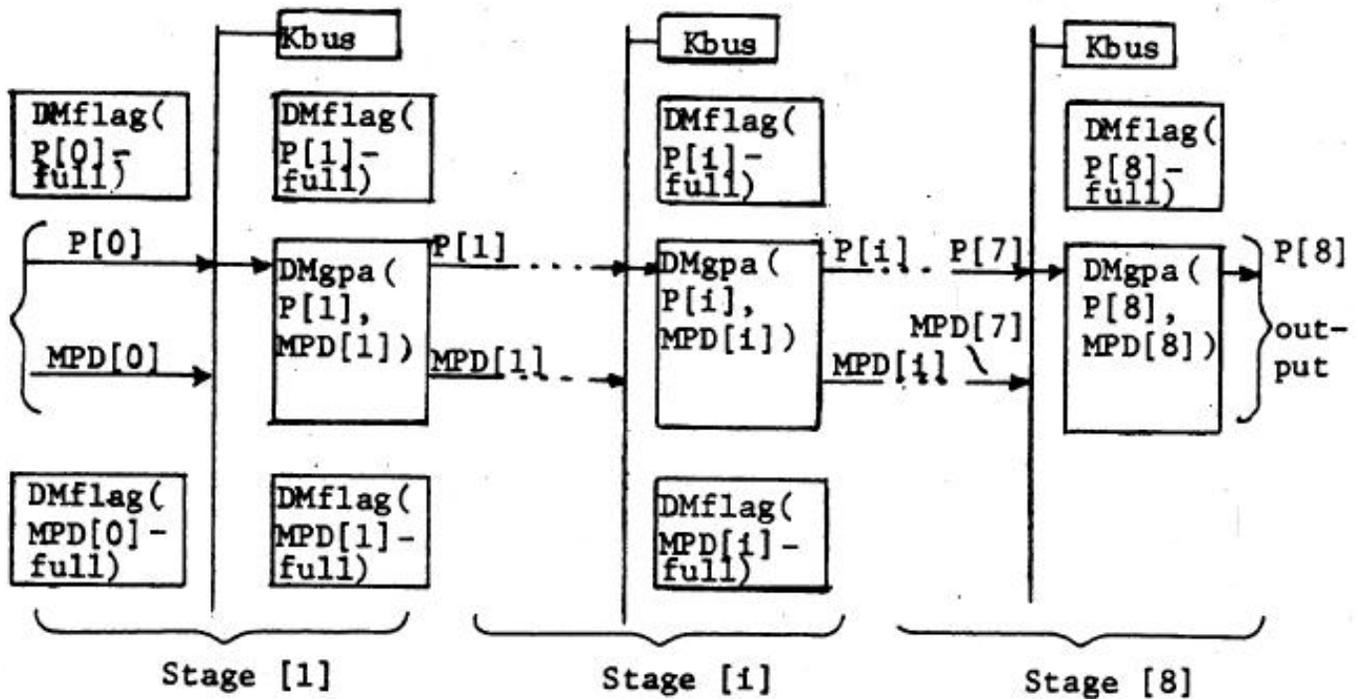


Fig. 12. Data part for 8-stage pipeline multiplier.

transfers data and of setting the flags of stage  $i$  to 1 after its computation is complete. Each stage operates as soon as its input data is available and its just-computed outputs have been used. Thus, the whole pipeline moves as fast as possible, compatible with not mixing up the computations.

The alert reader will have noticed that since the flag outputs are positive logic and the  $K_e$  control flow outputs are negative logic, the complements of the named flag outputs are the ones that must enter the  $K(\text{parallel merge})$ 's. Even so, there are some problems in using  $Kflag$ 's to enable  $((\text{parallel merge})$ 's. Chapter 7 must be read to discover why. After reading Chapter 7, return to this example and, if necessary, alter it so that the  $K(\text{parallel merge})$ 's are enabled properly.

The pipeline structure is considerably more complicated than the array case from a control viewpoint. The limit to the parallelism is set by the structure of the algorithm -- here, 8 stages -- rather than by the size of the data array being processed. The time to perform a given stage is 2.7 microseconds, computed from the control sequence of Figure 13, so that the time to get an answer is  $8 * 2.7 = 21.6$  microseconds. However, providing the data is fed into the pipeline rapidly enough, the system will finish a multiplication every 2.7 microseconds.

The pipeline makes clear that one must always consider two distinct performance measures for a process.

One is the time taken to produce a result, measured from the time at which the input, is presented -- this is the operation time. The other is the rate at which results are produced -- this is the operation rate. In ideal serial systems the operation rate is simply the reciprocal of the time for each operation:  $1/\text{operation-time}$ . But in general the two can vary independently. In parallel systems, as we have seen, the operation rate can be increased many fold. To do so usually implies an appropriate organization of the input data. But also, systems sometimes produce their outputs prior to the time they are prepared to accept the next input, thus reducing the operation rate

both on the objectives of the design (how much speed is worth) and on how much gain can be made in the overall time of the algorithm.

Array parallelism is not the most general form of parallel computation, defined as many independent-heterogeneous computations going on concurrently. It is distinguished by the fact that all the parallel computations are 'identical, hence there need only be a single control part for them all. This has often been likened to a symphony orchestra with a single conductor (the control) and many instruments. As we have seen, even in our simple example, the computations need not be completely identical. Often a small amount of control can be included in each instrument and still most of the control can be put in the hands of the single conductor. The problem of so orchestrating a computation to make this possible can pose a major challenge in the analysis - of the underlying algorithm.

## PARALLELISM: PIPELINES

The cost in hardware of array parallelism is very high. There exists another alternative for parallel organization, commonly called pipelining, that uses less equipment though (in consequence) obtains less speedup. It exploits the fact that any sequential flow diagram can be viewed as a sequence of work stations, through which the data to be processed flows. Each work station is active at the same time (i.e., in parallel), but doing its particular job on a different input. Thus, the computation flows along, as in a pipeline. An equally apt metaphor would have been assembly-line processing, since this is the same organization that is used in all mass production assembly lines (e.g., for automobiles). To make use of pipelining we need a continuing stream of inputs, for the efficiency depends on not having holes in the stream of data. But this is exactly the-situation we have in the averaging algorithm of Figure 7.

We could pipeline the entire process of Figure 7, but to keep our attention on multiplication let us just build a pipeline for multiplication, viewing the total flow diagram as providing a context in which the demand for multiplications are provided at a sufficiently high rate to justify a pipeline multiplier. Note that to do averaging, the final stage simply has to accumulate a running sum, which eventually gets multiplied by  $1/W$ , or, more correctly divided by  $W$ .

The basic stage in the multiplication algorithm is the multiplication-step, which forms the body of the loop in the basic flow diagram of Figure 8. Rather than executing a loop to use the same hardware (the DMgpa's) for each stage, we now wish to form our pipeline by providing separate parallel systems for each stage. Figure 12 shows the organization. There are 8 iteration steps in the algorithm, hence there are 8 stages in the pipeline. Each step consists of taking in  $P$  and  $MPD$ , looking at  $P<0>$ , and then transferring data (the new values of  $P$  and  $MPD$ ) to the next stage. Each stage essentially duplicates the data and memory part of the basic iteration step, here the DMgpa and the Kbus. Memory used for control will in general be different. Here, there is no need for the  $Mc$  used to hold the iteration count; but there are two 1-bit flags,  $P$ -full and  $MPD$ -full, that are required for the pipeline control.

The control for a single stage is given in Figure 13. The two flags for each stage permit the synchronization for the stages to operate at the same time yet pass data to one another. Stage  $i$  (in the figure) does not transfer data from the prior stage ( $i-1$ ) to its own registers,  $P[i]$  and  $MPD[i]$ , until the  $i-1$  registers are full (as guaranteed by the  $i-1$  flags being 1) and its own registers have had their contents transferred to the  $i+1$  stage (as guaranteed by the  $i$  flags being 0). Thus, stage  $i$  has the responsibility for setting the flags of stage  $i-1$  to 0 after it

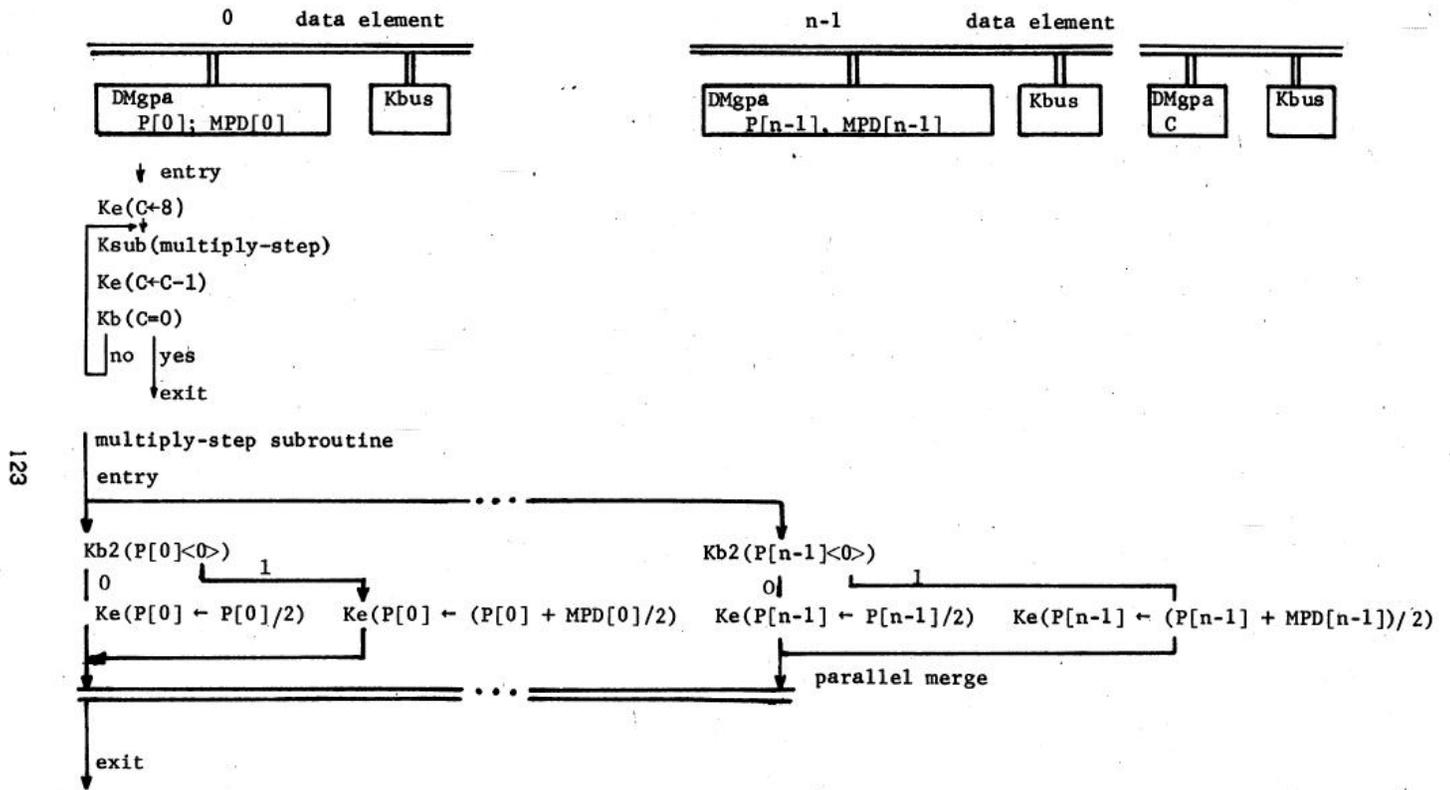


Fig. 11. RTM diagram for an n-element array (vector) of 8-bit multipliers.

Much more important is to recognize that the loop in the algorithm is basically a way of getting  $N$  independent operations performed (multiplying the  $X$ 's by the  $W$ 's). Thus, all of the multiplications could be carried out at the same time -- in parallel to use the accepted term. This will radically alter the total time to perform the algorithm, since what took  $N$  time intervals originally takes only one with the parallel implementation. The amount of gain depends on the size of  $N$ , and can reach any size if  $N$  is large enough. However, the gain, also depends on the rest of the algorithm and how it is implemented. For instance, the weighted  $X$ 's still have to be summed and multiplied by  $1/W$ . Since these operations depend on each other they cannot simply be separated into parallel computation streams. Thus, to evaluate the actual gain requires that we have a total implementation to analyze.

To implement the parallel scheme requires separate data-memory parts for each independent multiplier, in particular a DMgpa and a Kbus. The straightforward way would be also to have independent control parts. This would require a  $K$  parallel-merge at the end to be sure that the summation loop did not begin until all the multiplications had been completed.

However, each of the parallel parts has an identical control -- being simply copies of the same algorithm. Thus, one need only construct a single basic control part. Figure 11 gives the RTM structure of this implementation. Note that  $K$  parallel-merges are required in order to coordinate each step. One can certainly not depend on the systems to remain synchronized by themselves, even if the components are of identical physical construction. Note also that separate control must exist for each test on  $P<0>$ , since which way control goes is data dependent. The basic control loop is not data dependent, which permits it to be factored out. The basic loop of the Figure 9 is still needed, but it now encompasses only the summation of the weighted  $X$ 's, and we do not show it. Similarly, the final multiplication by  $1/W$  still occurs in series and is not shown.

We can now make a rough estimate of the increase in speed from the parallelism. The serial scheme takes:

$$t_{\text{serial}} = N \cdot t_{\text{multiply}} + N \cdot t_{\text{sum-loop}} + t_{\text{multiply}} + t_{\text{set-up}}$$

$$t_{\text{parallel}} = t_{\text{multiply}} + N \cdot t_{\text{sum-loop}} + t_{\text{multiply}} + t_{\text{set-up}}$$

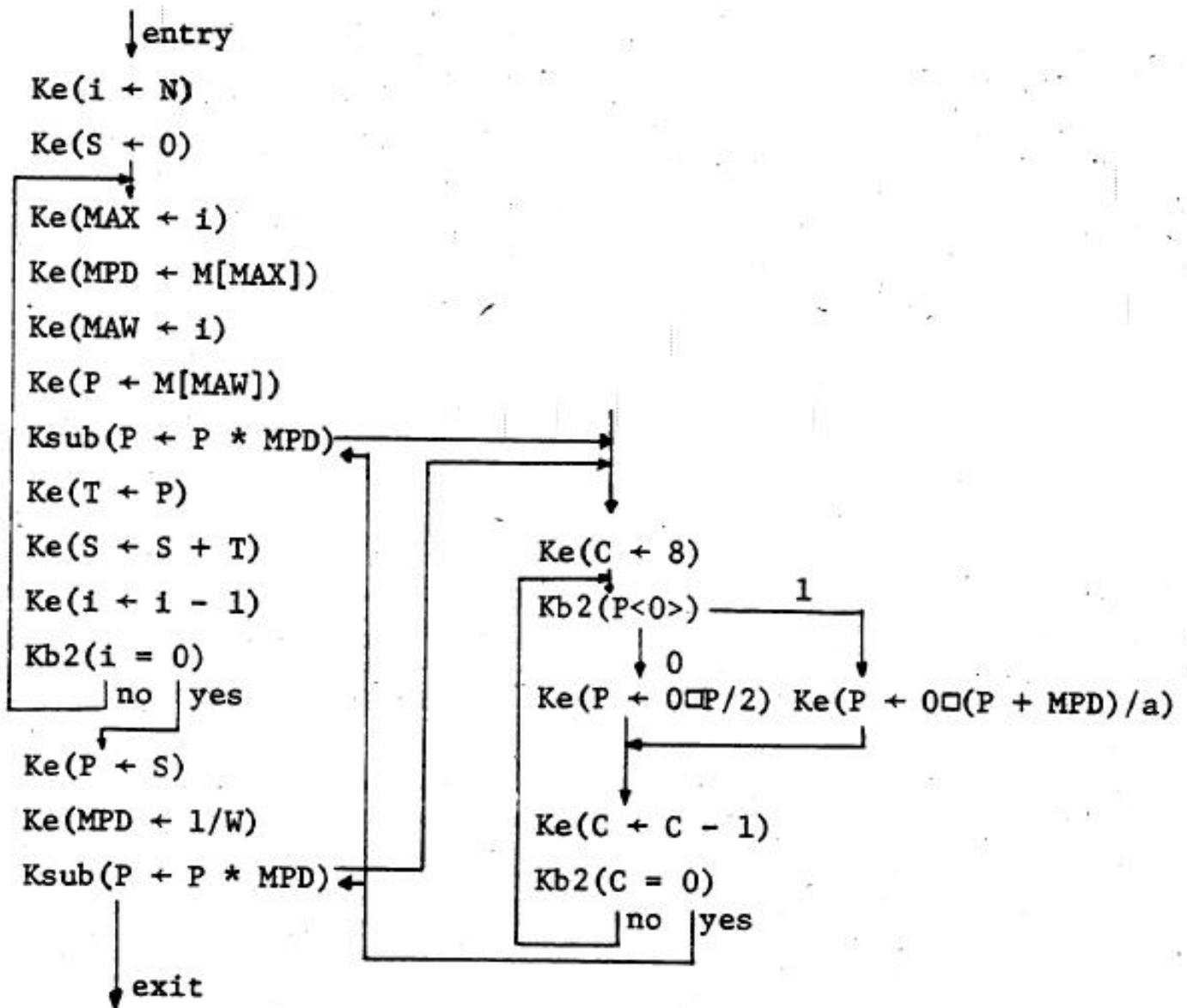
The two set-up times and the time for the summation loop are small fractions,  $f$ , of the multiply time and we can approximate them by  $f \cdot t_{\text{multiply}}$ . Thus we get:

$$t_{\text{parallel}}/t_{\text{serial}} = (2 + f \cdot N + f)/(N + 1 + f \cdot N + f)$$

As  $N$  gets very large -- thus getting the most parallelism -- the increase in speed becomes like  $f/(1+f)$  (simply ignoring all terms without an  $N$ ). If the small operations, represented by  $f$ , were such that  $f \approx 1/8$  (e.g., like a single multiply step) then this limiting advantage would be  $1/9$ . We see that the remaining

structure of the algorithm puts a limit to how much advantage is to be obtained by parallelism. While  $N$  is still small the advantage is about  $2/(N+1)$ , reflecting the fact that at least two multiplications must be done in series. If we consider only the multiplication time as the measure of performance, the ratio  $t_{\text{parallel}}/t_{\text{serial}} - 1/N$ . That is, we achieve a factor of  $N$  speed-up.

This form of parallelism, which we can call array parallelism, arises repeatedly in RT-level design. Many of the data structures that occur in practice consist of one or two dimensional arrays, and algorithms often specify that the same operation be done independently to each element of the array. Then the option always exists of factoring out the independent part and doing it in parallel, thus reducing that part of the time cost by a factor of  $N$ , the size of the array. One pays substantial additional hardware for this, and whether it is worth it depends



■ Fig. 10. Control part to compute weighted averages with multiply subroutine of Fig. 4.

make this design alternative easily available, since in general when the subsystem is large or used in many different places, subroutining is preferred. The Ksubroutine permits the nesting of subroutines to any level. However, it does not permit recursive subroutines, that is, subroutines that call themselves or call other subroutines that call them. In this respect the Ksubroutine is exactly parallel to its software counterpart in some programming languages, the conventional subroutine-call instruction, differing only in that its control is in unique hardware and that the subroutine is hardware rather than a program.

## PARALLELISM: ARRAYS

We used subroutines above in response to a concern with using too much hardware, giving up a small amount of operating speed. But our concern might not be with hardware cost at all. Instead, we might wish the algorithm of Figure 7 to run as fast as possible and are prepared to use additional hardware. One choice, of course, is simply not to use the subroutines, but to use the original implementation with macros (Figure 9). However, as we just noticed, this helps only a little.

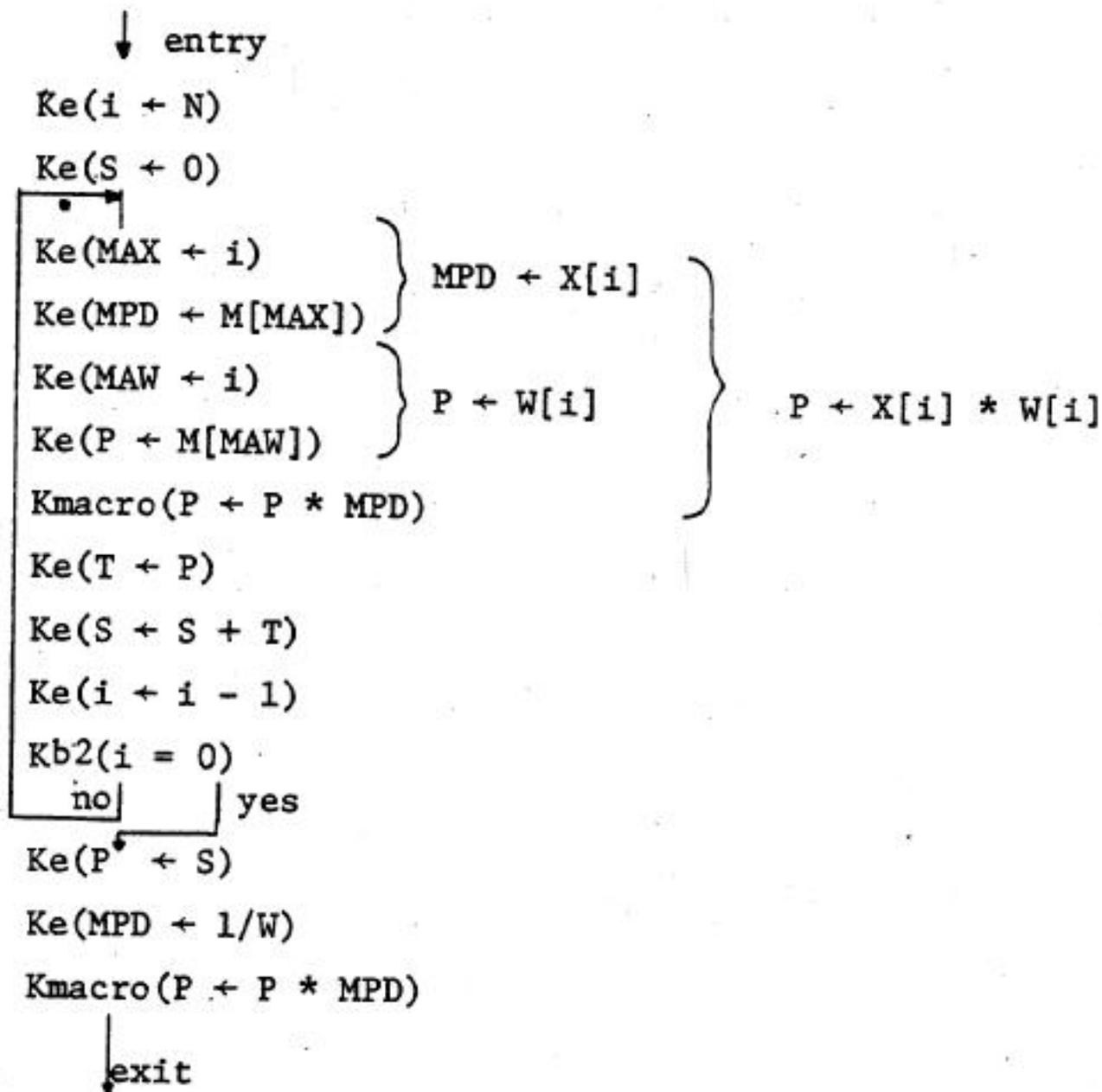


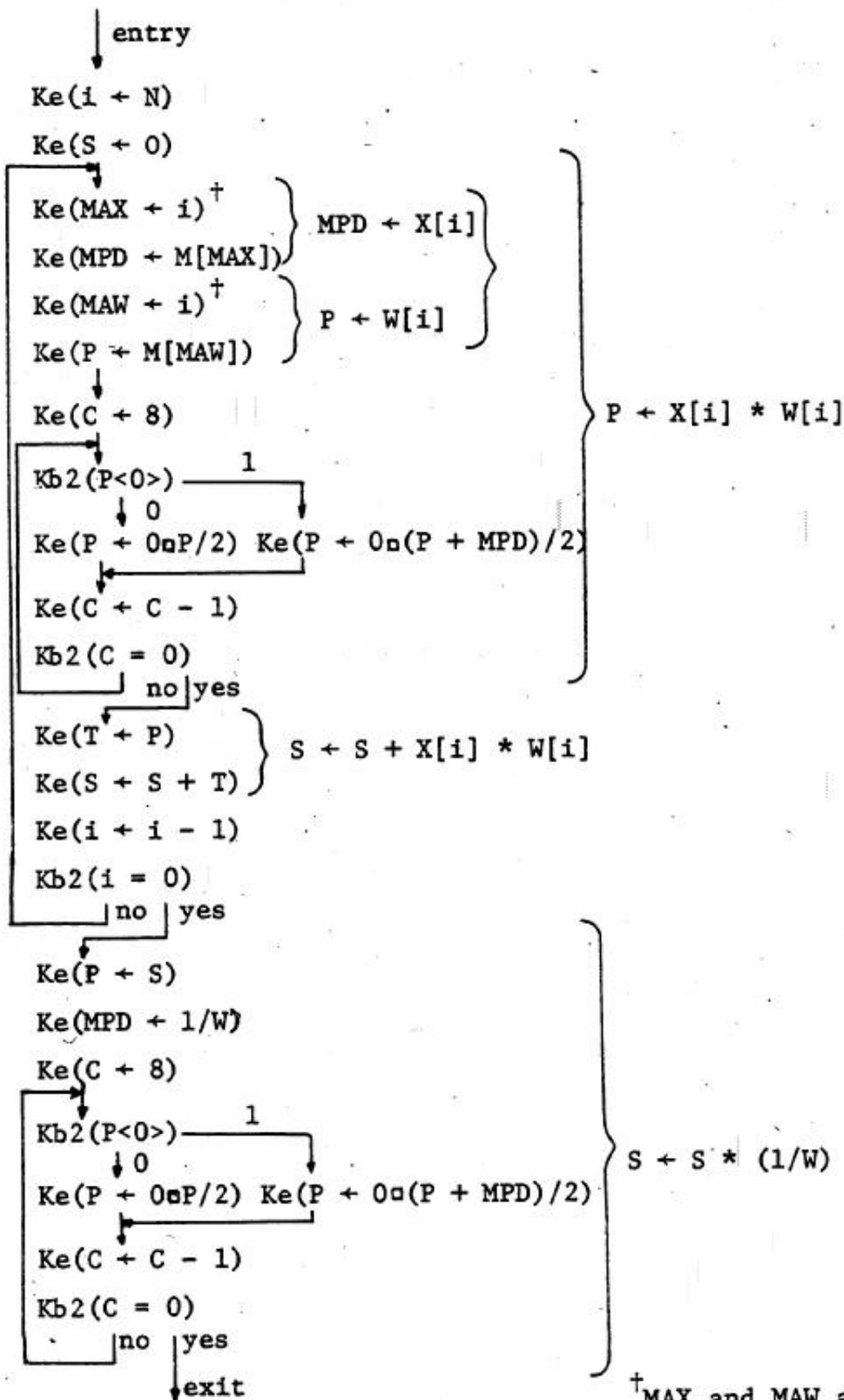
Fig. 9. Control part to compute weighted averages using multiply macro.

end of the subroutine. A return is made to all calling modules simultaneously and only the  $K_{sub}$  that actually made the call is active and will respond to pass control on to the next module.

The option of whether to share facilities via subroutines or whether to use separate hardware for each occurrence (macros) is one of the recurrent design alternatives in RT-level design. (It occurs as well in

programming, under the same rubric of subroutines versus macros; originally it was known as the alternative between open and closed subroutines, where an open subroutine was a form of what is now termed a macro.) As can be seen by comparing Figures 8 and 10, there is a very slight cost to be paid for using subroutines, both in time and in hardware. The cost in time is about 0.02 microseconds per call. The cost in hardware is the use of a Ksubroutine instead of a Kevoke at each occurrence plus the one Kserial-merge associated with the subsystem. In general this hardware cost is minor compared to the hardware savings from not having the additional hardware for each copy of the subsystem. However, it does put a lower bound on how trivial a subsystem can be effectively used as a subroutine.

The Ksubroutine was added to the basic set of RTM modules precisely to



† MAX and MAW are the memory address registers of the M(array)'s for holding the X's and W's respectively

holding the X's and W's  
respectively

**Fig. 8.** Control part to compute weighted averages with multiply macro of Fig. 4. substituted.

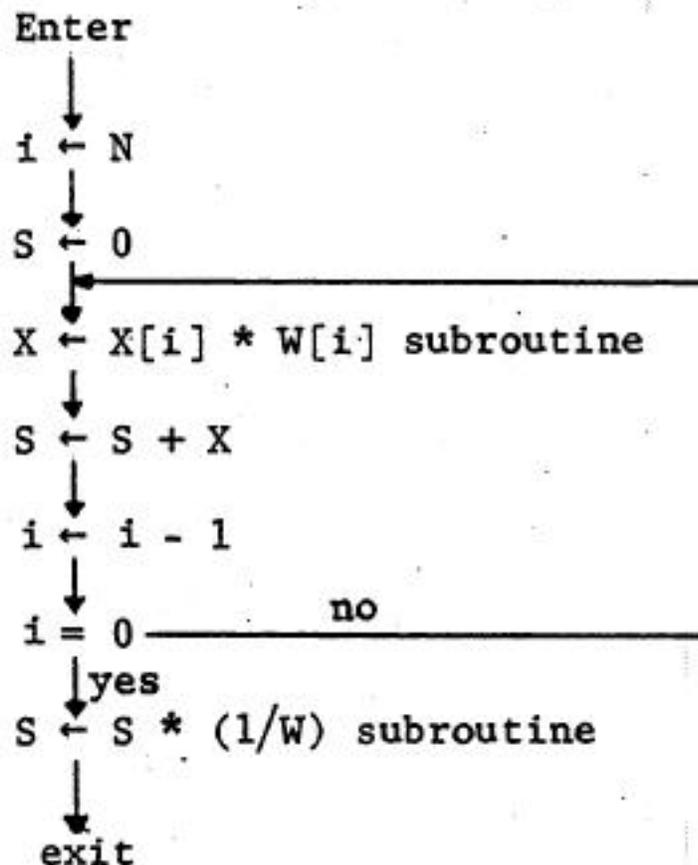


Fig. 7. Flowchart to compute weighted average.

multiplications are not. We can simply treat these expressions (the  $X \leftarrow X[i] * W[i]$  and the  $S \leftarrow S * (1/W)$ ) as standing for more complex RTM circuits. Thus, a direct mapping of the flowchart leads to Figure 8 in which we have replaced the two multiplication expressions by the appropriate memory array accesses and also copies of the multiply subprocess of Figure 4. We have left out the data-memory part of the system, since the structure is apparent from the control part. There will be separate DMgpa's and Mc's for each of the two multiplications, as well as components for the other operations in the flowchart. Two M(array)'s are necessary to hold the X's and the W's. We assume that the X's and  $1/W$  are originally stored in the 8 most significant bits of their words, so that they load MPD properly.

In creating an RTM flowchart for the algorithm of Figure 7 there is no reason not to have written Figure 9, rather than Figure 8. Here we have not yet replaced the multiplication expressions by the subsystems. Instead we have simply indicated that there is something that evokes that multiplication by using the Kmacro that was introduced in Chapter 2. Thus, provided we have recorded someplace that  $Kmacro(P \leftarrow P * MPD)$  is the structure of Figure 4 (with names appropriately changed), Figures 8 and 9 give the same information.

## SHARED FACILITIES: SUBROUTINES

It may have occurred to you that we were providing a lot of hardware for each multiplication, and that it would be nice to be able to share a single multiplication facility between the two uses of multiplication. This is exactly what the special control module `Ksubroutine\Ksub` permits. Figure 10 shows the algorithm of Figure 7 accomplished in this fashion, giving links explicitly to the single version of the multiply subsystem. This subsystem is identical to the system of Figure 4, except that there is a `Kserial-merge` at the beginning to handle the multiple calls. No additional coordinating structure is required at the

```

00100          INTEGER C,MPD,P,N1,N2
00150          LOGICAL TEST
00200          C GENERATE TEST NUMBERS FOR N1 AND N2
00300          DO 10 N1=0,255
00400          DO 10 N2=0,255
00600          P=N1
00700          MPD=N2*2**8
00750
00800          C 1 BUS 8-STEP MULTIPLIER
00900          C=8
00925          C SERIAL MERGE
00950          1          CONTINUE
01000          TEST=MOD(P,2).EQ.0
01100          IF (TEST) P=P/2
01160          IF (.NOT.TEST) P=(P+MPD)/2
01200          C SERIAL MERGE
01250          2          CONTINUE
01300          C=C-1
01400          IF((C.NE.0)) GOTO 1
01450
01500          C TEST ANSWERS
01600          IF((N1*N2).NE.P) GOTO 3
01700          10         CONTINUE
01800          C PRINT  ERRORS
01900          3          TYPE 4,N1,N2,MPD,P
02000          4          FORMAT (5I10)
02100          END
02200

```

Fig. 6. Fortran program to test 8-bit RTM multiplier of Fig. 4.

To have a concrete task to discuss, Figure 7 gives an algorithm for taking a weighted average. The components of a data vector,  $X[i]$ , are each multiplied by weights,  $W[i]$ , which are then summed (forming the vector dot product) and divided by the total weight,  $W$ . We will assume that the data remains within 8 bits, to avoid distraction from the central points. Since multiplication is available and not division, we have written the flowchart to use multiplication by  $1/W$ . The weights are themselves constant, hence  $1/W$  is a given constant, just as  $W$  is. Since we are working with integers,  $11W$  will be a

fraction between 0 and 1, hence not representable. However, we can simply take  $11W$  to be the fractional part (as an integer), which is equivalent to rescaling the expression. Actually since such a representation will cause unnecessary errors, we would have to use the integer division system to do the task properly.

## THE NATURAL MAPPING: MACROS

One can attempt to produce an RTM flowchart directly from Figure 7, just as we did for the multiplication itself. However, unlike Figure 3, not every component of Figure 7 is a primitive module -- in particular, the two

design is equally important. Major errors can often be detected by testing the design with a small set of input values that provide results that are known independently. For the actual system this can be done by running it on the selected inputs; for earlier (paper) stages of the design this can be done by hand simulating the operation.

The most important property of these test cases (besides the fact that the results must be independently known so a true check is possible) is that they exercise all the logical paths in the program. It is not sufficient that each K(branch) have each output taken at some time, though that is surely necessary. Each combinatorial possibility for tracing a path through the program must occur. In addition, an attempt must be made to include both special values and general values of inputs, in terms of the operations known to occur. For arithmetic 0 and 1 are always special, as well as the numbers at the maximum and minimum of the range. With all these conditions, the number of test cases can be quite large, especially with systems that have any degree of cascaded branching.

Hand simulation is an untrustworthy process, especially when what is desired is an assurance of correctness. It can often be replaced by using a computer to do the simulation. Figure 6 gives a FORTRAN program for checking the RTM flowchart stage of our multiplier design. The program is directly mapped from the RTM structure of Figure 4, and it is not taken from the flowchart describing the algorithm (though that could be checked too, if desired). In our case the two flowcharts are essentially identical, but that will not be the case most of the time. Consequently, care must be taken in the simulation program to reflect each of the RTM actions correctly.

The program of Figure 6 actually proves the correctness of the RTM flowchart of Figure 4 (assuming the translation to the simulator was made correctly), for it exhaustively checks all possible inputs. All  $2^8$  values of N1 are tested against all  $2^8$  values of N2, for a total of  $2^{16}$  test cases. This requires only about 30 seconds of time on a large computer (IBM 360/65 or DEC PDP-10), well worthwhile to attain completeness. However, this is a very special case, as consideration of the time required to exhaustively check a full 16-bit multiplier will show. The point of our example is that the notion of exhaustive testing should not be excluded automatically, just because it usually takes too long; sometimes it can be done.

The final step in a design, even after checking out the system, is documentation. The notation of Figure 4 serves this purpose well -- one of the side benefits from a systematic scheme of RT-level design. The wiring pin numbers and module locations can be conveniently entered on the same diagram, as was done in Figure 7 of Chapter 2 for the sum-of-integers problem. It is desirable to have as few separate documents as possible, since changes and corrections then have to be entered in several places. However, if handwiring is to be done in implementing a design, it is usually convenient to prepare a separate wiring list.

## VARIATIONS IN THE USE OF MULTIPLICATION

We have now designed a scheme for multiplying. Let us resist for the moment casting it into concrete hardware. Some of the recurrent issues in RT-level design occur when one considers doing some multiplication within a larger computation. To consider multiplication as analogous to another basic module (e.g., a DM(multiply)) is certainly possible and often useful. But it can pre-empt some other options.

if inputs are physically possible they will in fact occur at some time and someone will want to be alerted to the possibility of aberrant behavior by a subsystem such as our multiplier. This incompleteness in specifications is much more serious when it occurs for a general purpose component, such as our multiplier, than for a subsystem to be used only within a specific arrangement, where the inputs that will occur in the environment can be known.

We have three choices in completing the specification. One is simply, to enunciate explicitly the conditions under which the system will produce an error. With this solution we would say: Our system multiplies two 8-bit numbers located in registers P and MPD in such and such fashion; if the other halves of the registers are non-zero an error will occur. This is not very satisfactory, but at least it is honest. The second choice is to make explicit what the output actually is, especially if it is understandable so that its effects could be detected. With this solution we would say: Our system multiplies two 8-bit numbers located in the low-order half of P and the high-order half of MPD, assuming the other halves are zero; if the high-order half of P is non-zero, it is added to the product; if the low-order half of MPD is non-zero the result is garbage. This choice is somewhat more useful. The third and conceptually most satisfactory choice is to specify appropriate behavior on all inputs and then modify the design to achieve it. For instance, we could enforce the assumptions by setting the two halves to be zero. With this solution we would say: Our system multiplies two 8-bit numbers located in the low-order half of P and high-order half of MPD; a larger number in P is truncated to the least significant 8 bits; the low-order bits of MPD are set to zero. The trouble with solutions of this type is that they cost both in time and in hardware to implement. For instance, Figure 5 shows the control part for an initialization process for the multiplier to meet these new specifications. It must be judged whether the additional costs are worth the cleanliness of the behavior specification.

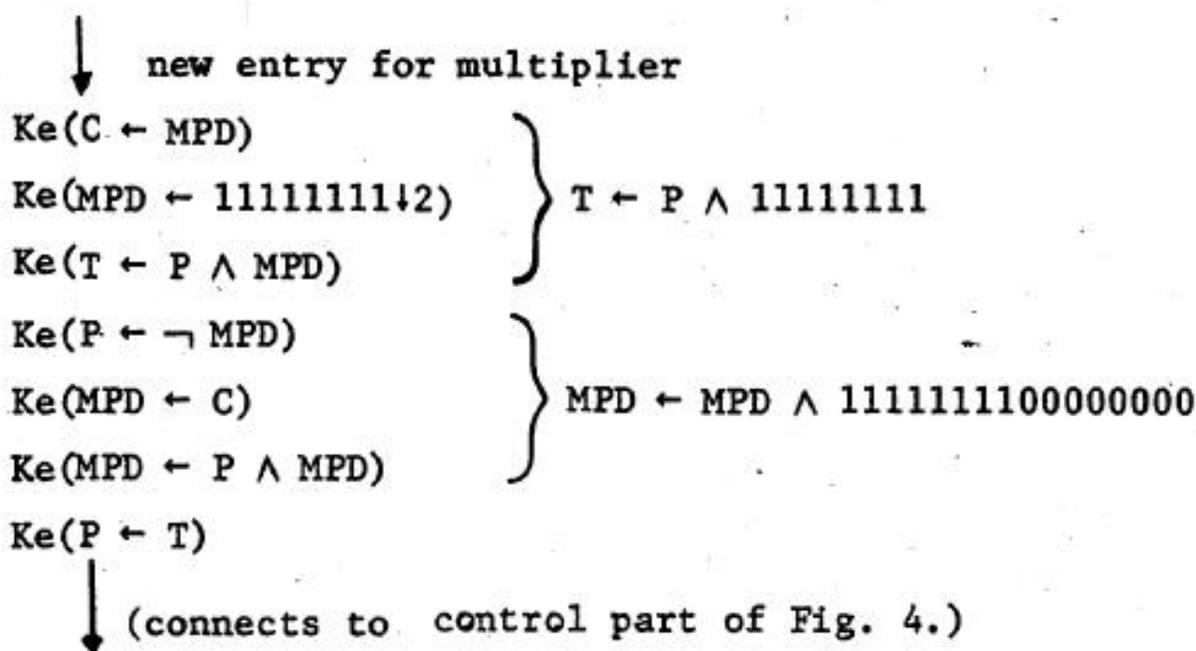


Fig. 5. Control part of initialization process for multiplier.

We emphasize the completeness and correctness of the specification because it is so easily overlooked, being taken as a given. Correctness of the rest of the

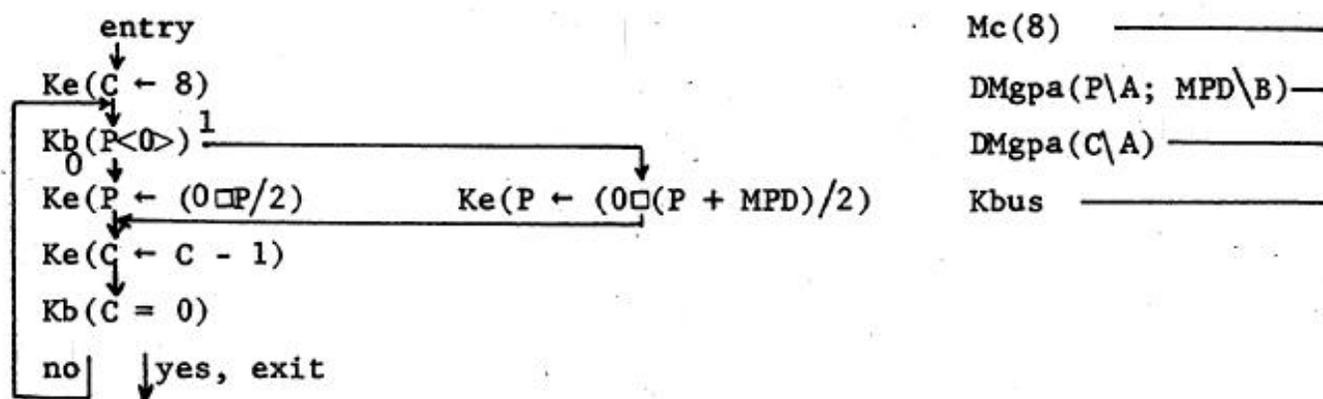


Fig. 4. RTM diagram for 8-bit shift right multiplier.

Figure 3 into a control step with a suitable K in the RTM flowchart: Ke's are used to evoke the OM's and Kb's are used to test for the conditions and branch control as shown in Figure 4.

Figure 4 completely specifies the design of an RTM multiplier system. The next step would be to implement the system by obtaining the indicated modules, assigning them to physical locations and interconnecting them. This step was described in Chapter 2 for the sum-of-integers example. It is sufficiently mechanical that it need not be repeated again here. Indeed, this step has been automated (providing a wire list), and the operation of the program that does this is described in the PDP-16 handbook.

The design, though completely specified and perhaps even built, is not yet complete. Two more steps remain. The first is to verify the correctness of design. If the specifications are correct and complete, and if the algorithm is correct, and if all the design steps described above have been taken correctly, up to and including the physical interconnection of the modules -- then of course the system should operate correctly. But all these if's are exactly what has to be checked.

Notice, for instance, that we have included the specifications as part of what must be verified. Though they usually form the assumed starting point, they can never in fact be assumed. With multiplication, the function is so well known that it seems hard to imagine what could be missing. But what if the inputs are greater than 8 bits? Physically, the multiplier could be 16 bits. Similarly, though with our particular conventions it is not really possible for the multiplicand to be greater than 8 bits, the 8 least significant digits could be non-zero and this would invalidate the result. We did not specify what the system output should be in these cases.

It can be objected that, since we assumed certain things, we are not responsible if the assumptions are not

met. But such a cavalier attitude will not do. Our specifications are incomplete, since they do not specify what the behavior is for all physically possible inputs to the system. We can be sure that

**Registers**

$P<15:0>$ \ Product	16-bit product and 8-bit multiplier (i.e., $MPR<7:0>$ )
$MPD<15:0>$ \ Multiplicand	only 8 bits are used (i.e., $MPD<15:8>$ )
$C<2:0>$ \ Counter	8-state iteration counter

**Values at entry**

$P<7:0>$ \ Multiplier	
$P<15:8>$	must be 0
$MPD<15:8>$ \ Multiplicand	
$MPD<7:0>$	must be 0
$C<2:0>$	undefined value

**Values at exit**

$P<15:0>$ \ Product	
MPD	unchanged value
C	0

**Algorithm**

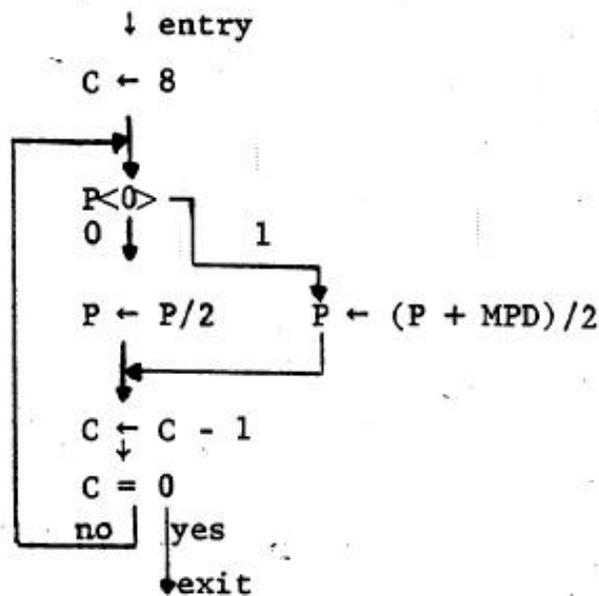


Fig. 3. Algorithm for 8-bit shift right multiplication.

[previous](#) | [contents](#) | [next](#)

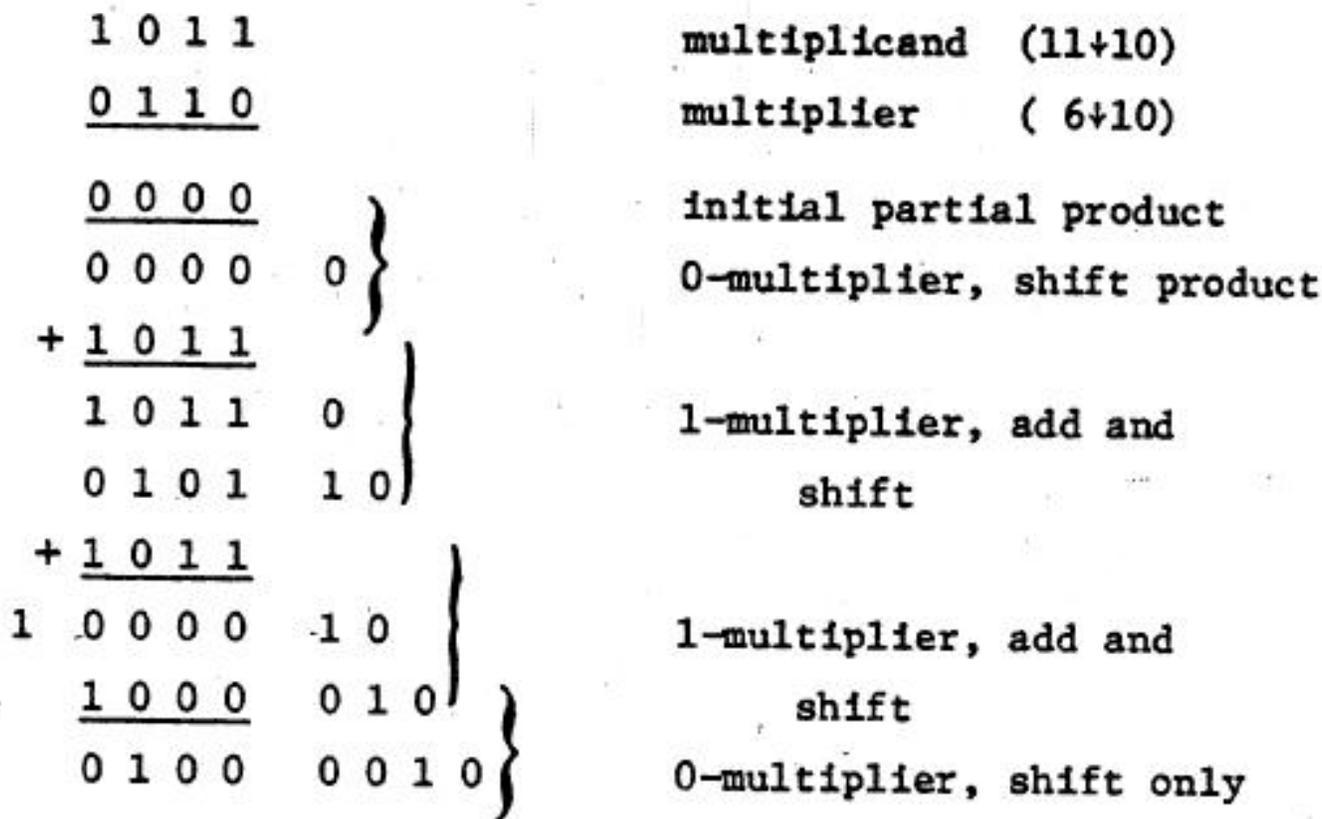


Fig. 2. Example of multiplication scheme by shifting result right.

Accepting this as a rough idea, the next step in the design is to specify the algorithm precisely. This can be done easily by a flowchart (or diagram) augmented by a declaration of the specific data required. This is given in Figure 3 for the algorithm of Figure 2, using 8-bit arguments with a 16-bit product. Two registers are needed for input: MPD for the multiplicand and P for the multiplier. The product is also developed in P. Notice, however, that the input data is positioned in an unconventional way: the multiplier is in its conventional position at the right side of the word; but the multiplicand is in the left-most 8 bits of MPD. Furthermore, the right-hand least significant bits of the multiplicand register,  $MPD\langle 7:0 \rangle$ , must be 0 initially. We adopt these constraints in the hope, mentioned above, of achieving a somewhat more efficient algorithm. The flow diagram in Figure 3, written below the data declarations, is simply a straightforward description of the algorithm used in Figure 2. It is necessary to introduce one other 3-bit register,  $C\langle 2:0 \rangle$ , to keep track of the iterations in order to exit the loop when the multiplication is finished.

With the algorithm specified, the next step in the design is to choose an appropriate set of M and D modules to hold the data and to carry out the operations. A straightforward selection leads to a DM<sub>gpa</sub> for the arithmetic operations on P and MPD and another DM<sub>gpa</sub> for the subtraction on C. Since the

DMgpa's have sufficient memory for their operands, no additional M's are required for MPD, P and C. (This simple example makes it clear that one should always select the D's first with RTM's, rather than a set of M's to hold the data, since D's are often combined with some memory.) However, the system does require the constant 8 to initialize C, so that an Mc is needed in addition to the two DMgpa's. This data-memory part is shown in Figure 4.

Given the organization of data and memory, the next step in the design is to construct the RTM flowchart, which specifies the control part of the system. In the present case this can be done by mapping each step of the flowchart of

1 0 1 1	1110	multiplicand
x 0 1 1 0	6110	multiplier
<u>0 0 0 0</u>	0-lsb	} least significant bit digits of multiplier
1 0 1 1	1	
1 0 1 1	1	
<u>0 0 0 0</u>	0-msb	} most significant bit
1 0 0 0 0 1 0	66110	final product

1a. Example of conventional method of multiplication for binary data.

1 0 1 1	
<u>0 1 1 0</u>	
0 0 0 0	initial partial product
<u>0 0 0 0</u>	first multiplicand addend (lsb = 0)
0 0 0 0	first digit partial product
<u>1 0 1 1</u>	
1 0 1 1 0	second digit partial product
<u>1 0 1 1</u>	
1 0 0 0 0 1 0	third digit partial product
<u>0 0 0 0</u>	
1 0 0 0 0 1 0	final product

1b. Example of multiplication using sequential process.

partial product	multiplicand	multiplier	
00000000 } +	00001011	0110	
<u>00000000</u> } +	00010110	0011	} first step
00010110 } +	00101100	0001	
<u>00010110</u> } +	01011000	0000	} second step
00101100 } +			
<u>01000010</u>			} third step

1c. Example using multiplication algorithm.

Fig. 1. Example of conventional multiplication scheme (shift left).

111

[previous](#) | [contents](#) | [next](#)

## CHAPTER 4 EXPLORING RTM DESIGN ISSUES

You have now been introduced to the set of basic RT modules and have designed a number of simple devices and systems, no doubt relying mostly on native intelligence. As we commented in the first chapter, in any sufficiently complex domain there is no systematic way of doing design. The design specifications and objective functions are too varied and the combinatorial possibilities for composing new structures are too immense. Still, much more can be said than just asserting 'that each design problem is an isolated puzzle. Candidate designs can be evaluated quantitatively to reveal explicitly the extent to which the objective functions are satisfied. The space of possible designs can be explored systematically (even if not exhaustively) to obtain a clear impression of the properties of the accepted design in contrast to the ones rejected. Furthermore, recurrent patterns arise in the space of possible designs that help in understanding whether further possibilities exist to be investigated.

This chapter is devoted to indicating the systematic characteristics of RT-level design. Consonant with the rest of the book we will proceed primarily by example, rather than by detailed theoretical description. The entire chapter will be devoted to the design of a single device -- the multiplier. We will endeavor to carry this through in a way that exhibits good design as well as revealing many of the basic patterns of design alternatives.

### BASIC DESIGN OF A MULTIPLIER

We take as given the desired goal of multiplying any two 8-bit positive integers to form a 16-bit product. We will consider that such a system for multiplication is to be used in many other systems that will be constructed later. Thus, the data is to be represented in standard form, as 8 and 16 bit vectors in a word, and the fixed character of the PDP-16 data organization with the Bus determines almost completely the way input and output will occur to our system.

The first design decision is the algorithm for doing multiplication. Starting with what we all know about multiplication, Figure 1a shows by example (on 4-bit numbers) the standard method of multiplication, using the multiplicand 11 and multiplier 6. For machine implementation, the algorithm consists of forming a series of partial products for each digit of the multiplier, shifting each one, to the left one place, and then adding them all up to obtain the final product as shown in Figure 1b. Since the hardware adds all the bits of a word in parallel, but does not add up a column of digits (all corresponding digits of the partial products) all at once, the algorithm has been reformed to accumulate the partial result at each stage. Furthermore, multiplication by a 0-bit does not add anything and multiplication by a 1-bit is equivalent to simply adding the multiplicand. Thus, the algorithm reduces fundamentally to adding the multiplicand whenever the multiplier bit is 1 and then shifting the multiplier and multiplicand one place, until all the bits of the multiplier have been processed, as shown in Figure 1c.

Additional insight into adapting this standard manual multiplication algorithm to a binary machine with

parallel addition can be obtained by considering the right shift scheme of Figure 2. Here we shift the partial result and the multiplier to the right at each stage. But since the partial result grows by 1 bit each step and the multiplier shrinks by 1 bit each step, their total bit, size remains constant (at 8 in the example of the figure). Thus, if the multiplier and product can be coupled together it may be possible to shift them in the same operation.

## SOLUTION

Figure SW-2 shows an implementation of a waveform generator for the function which uses a  $K(\text{clock})$  for the time base. In order to change the period of the waveform, the clock must be changed. The  $K(\text{clock})$  would have either a fixed period waveform or be variable using a potentiometer. A programmable clock (given in the preceding section) would provide the variability needed for such a generator. Note with a  $K(\text{clock})$  there is no data part which uses an RTM Bus. The only evoke complements the  $\text{Output}(t)$  function which is held in a  $\text{DMflag}$ . The function simply is 0,1,0,1,... with every clock period count.

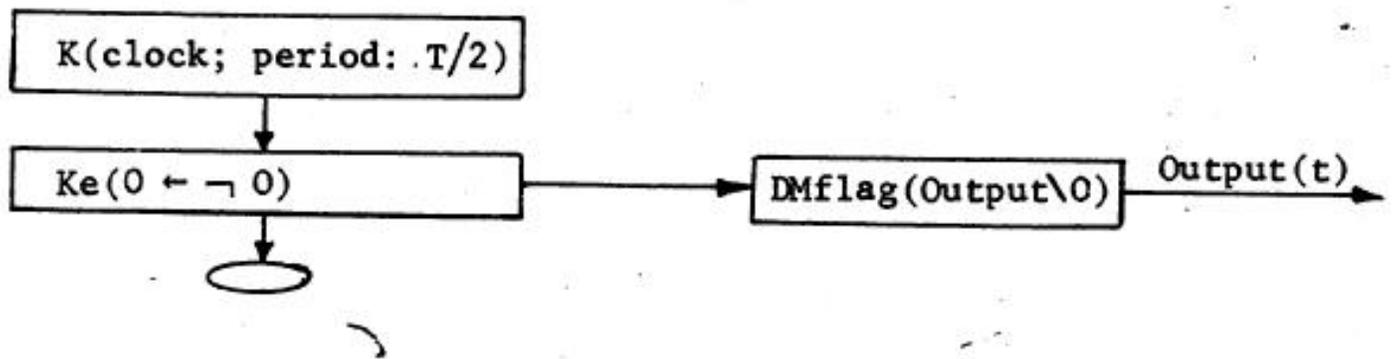


Fig. SW-2. RTM diagram of square wave generator using clock 'time base.

## ADDITIONAL PROBLEMS

1. Design a square wave generator which uses the  $K(\text{delay})$  to specify the time. Note, two  $K(\text{delay})$ 's are needed because of the recovery time restrictions of  $K(\text{delay})$ .
2. Design a square wave generator using the  $K(\text{programmable delay})$  given in the problems of the clock-delay.
3. Design a square wave generator in which the period,  $T$ , consists of two variable parts,  $t\text{-off}$  and  $t\text{-on}$ . That is,  $T = t\text{-on} + t\text{-off}$ .

The  $\text{Output}(t) = 0$  for  $0 \leq t \leq t\text{-off}$

$$= 1 \text{ for } t\text{-off} \leq t \leq T$$

Use either a  $K(\text{clock})$  and  $K(\text{delay})$  or two  $K(\text{delay})$ 's.

4. Design a complete system in which the output of the waveform generator, time t-off, is 1, and time t-on is proportional to the next Fibonacci number. Each time the waveform changes, a one is added to the number used to generate the Fibonacci number. That is: the Output(t) sequence will be:

off, on(0), off, on( 1), off, on (1), off, on(2), off, on(3) off ...

When the generator overflows it should be reset to start at 0.

5. What is the maximum rate for the clock in the case of the Fibonacci number generator? Can you redesign the system to increase this rate?

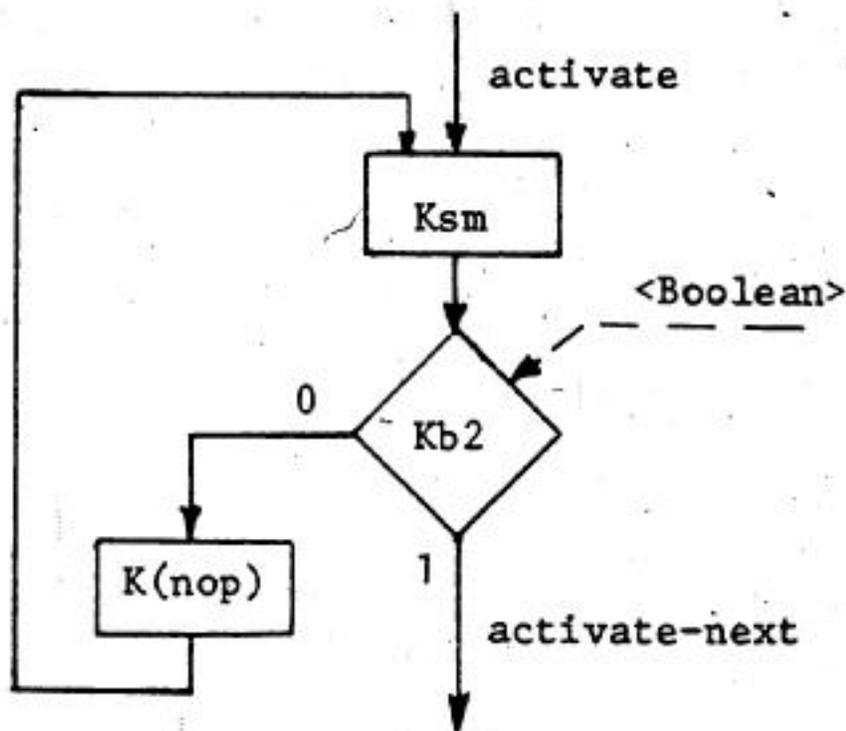


Fig. WU-2. RTM diagram of K(wait-until) ERTM.

### D(SQUARE WAVE GENERATOR)

**KEYWORDS:** Timing, clock, delay, waveform generator, synchronization

One important use of the K(clock) and K(delay) is in the design of waveform generators. These systems take input parameters and create an output waveform, Output(t), which varies with time. The simplest digital-type waveform that can be generated is the symmetrical square wave. It is represented graphically and functionally in Figure SW-1. With this function the sampling time between changes of the function is equal to half the period of the function,  $T/2$ . The function changes in discrete steps, hence, the Output(t) is 0 or 1, rather than being continuous.

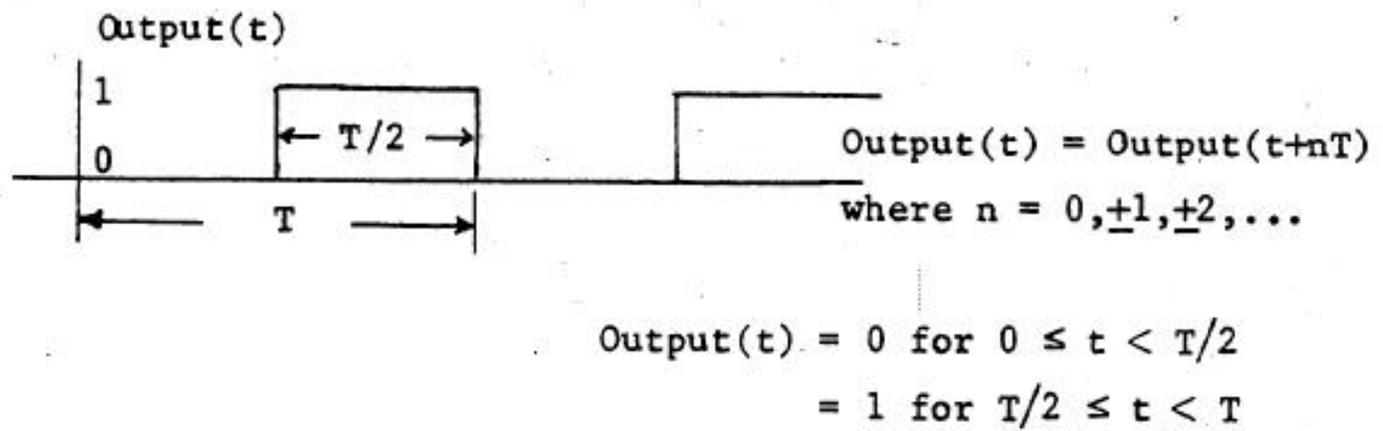


Fig. SW-1. Timing diagram of square wave function.

## PROBLEM STATEMENT

Design a waveform generator which will behave according to the square wave function.

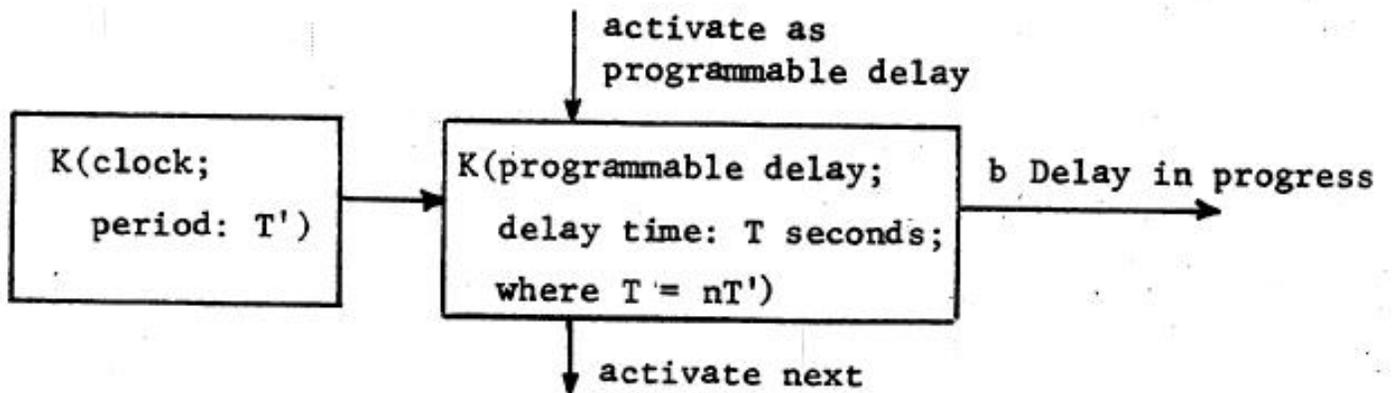


Fig. CD-3. Module diagram for `K(programmable delay)`.

### `K(WAIT-UNTIL)` EXTENDED RTM

**KEYWORDS:** Wait-until, ERTM, polling, synchronization, parallel merge

Conceptually the notion of waiting for a Boolean condition to occur (to be true) prior to proceeding is fundamental to synchronization between systems. Normally some external process causes a Boolean (flag) to be set to indicate the occurrence of an event. A second process, which is to be synchronized with the event, checks the Boolean input condition by polling, and when true, proceeds.

### PROBLEM STATEMENT

Design a module, `((wait-until))`, which carries out the waiting function with external characteristics as shown in the module diagram of Figure WU-1.

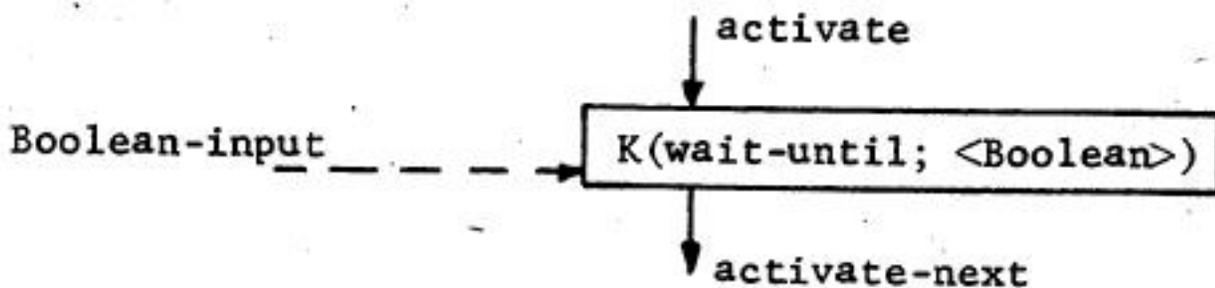


Fig. WU-1. Module diagram of K(wait-until).

## SOLUTION

This module is similar to the K(parallel merge) in that it is activated, waits until a Boolean input condition is satisfied, and then proceeds to activate the next module. Figure WU-2 shows the equivalent RTM macro formed from three conventional modules. For systems with an activate control signal identical to a Boolean, the K(wait-until) is identical to a K(parallel merge).

a DO loop causes a one time execution of the DO loop, they become slightly critical about meaning and consistency. Here, however, the problem is not just that 0 produces no actions; it is still necessary to check for a zero case prior to entering the loop, even if the loop detects 0 indexes prior to executing the loop. The following two solution implementations point this out.

#### SOLUTION 4

In Figure CD-2e the testing for  $n=0$  is performed before entering the loop, using the for loop ERTM previously described. Waiting for the clock will occur no times if  $n$  is 0. The structure has a more consistently defined behavior than Solutions 1 and 2 because the for loop is executed in a short time, generating a very large number of outputs per unit of time. The rate is uncontrolled, however, being determined by the time taken by the for loop control. That is, when  $n$  is 0 the part that synchronizes with the clock is never evoked. Somehow, this type of action also seems strange when compared with the behavior of a program. The difference can be seen between this case and that of programming; in programming one is interested in the number of times a process is to be executed and less interested in the control part (i.e., a program). Here the number of times a process completes is of interest and the control structure completion is giving erroneous outputs.

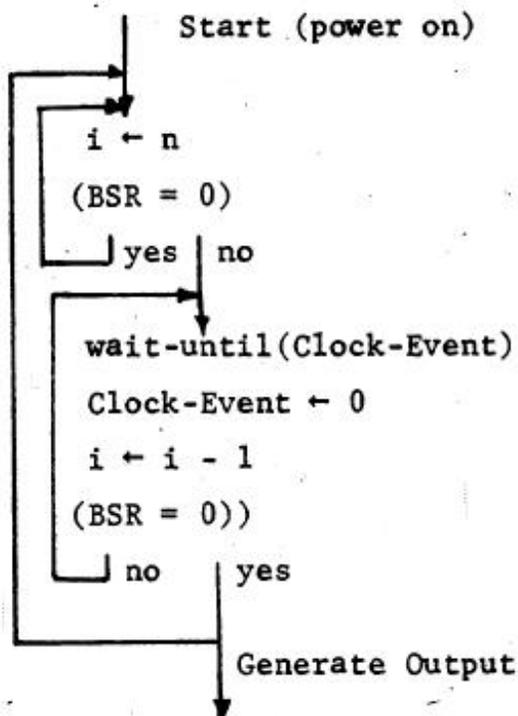
#### SOLUTION 5

Figure CD-2f finally solves the problem again, by first checking whether  $n$  is 0 before starting the for loop. In this way, erroneous outputs for this case are not generated. This rather lengthy discussion of five designs, only two of which are correct, illustrates concepts of number representation, loop control, and inter-process synchronization. In addition, it illustrates how a problem often becomes totally specified only when the design is carried out.

#### ADDITIONAL PROBLEMS

1. Design a K(programmable clock) which has only one control part (instead of two) that is activated from the K(clock; period: T). A DMflag (Clock-Event) might not be used, but instead each occurrence of the clock would cause the status of  $n$  and  $i$  to be checked and updated. Output control events would be given each time  $i$  is counted down to zero.
2. K(programmable(variable)delay). - A K(programmable delay) can be constructed which is similar to the K(programmable clock). The overall structure of such a device is given in Figure CD-3. The behavior is like the standard K(delay) of Chapter 2.
3. Design a clock that has two control parts that are similar to the control parts used in the K(clock) above. Determine the average delay time as a function of the input delay time,  $n$ . Take care to avoid a solution which has the behavior of  $(n,t)$  as  $(0,0), (1,0), (2,0\sim 1), (3,1\sim 2)\dots(n,n-2\sim n-1)$ .

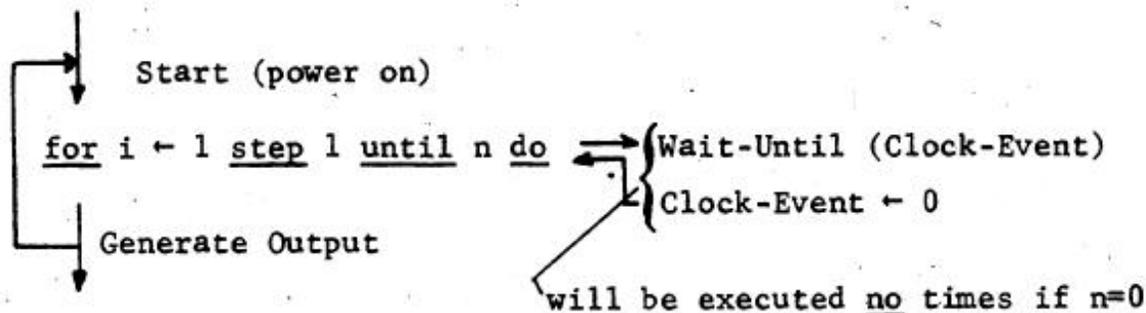
4. Design a delay with two control parts which uses the time base generator to cause the output activate-next signals as in the control part case of problem 1 above. The counting process is started by a flag set by the activate input which indicates that a time period is to be measured.
5. Design a K(integrating programmable delay) which is similar to that given in Chapter 2.



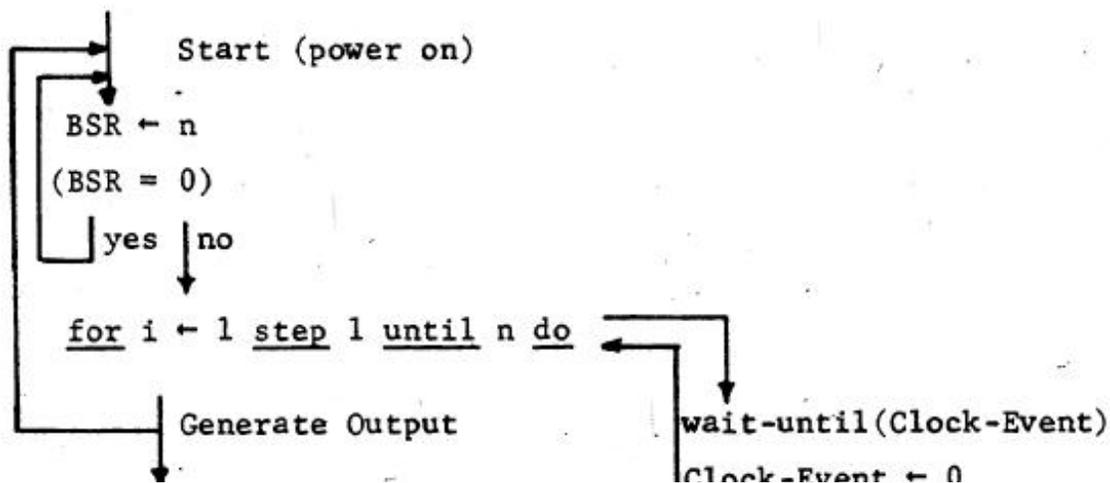
behavior:

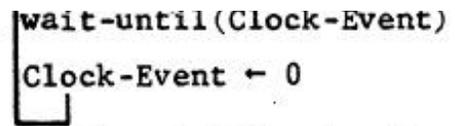
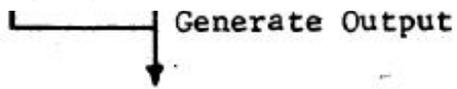
<u>n</u>	<u>t</u>
0	0
1	1
⋮	⋮
$2^{16}-1$	$2^{16}-1$

d. Control part for counting (unsigned numbers) using K(wait-until)



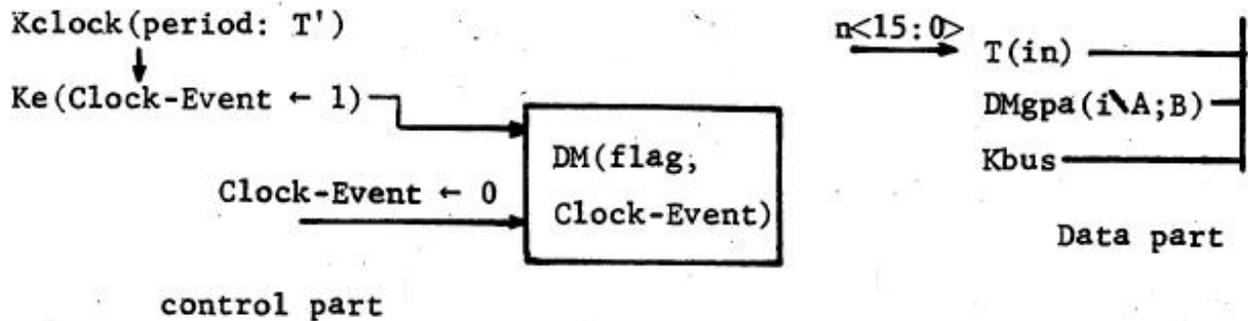
e. Control part for counting using K(wait-until) and K(for loop), ill-defined n=0



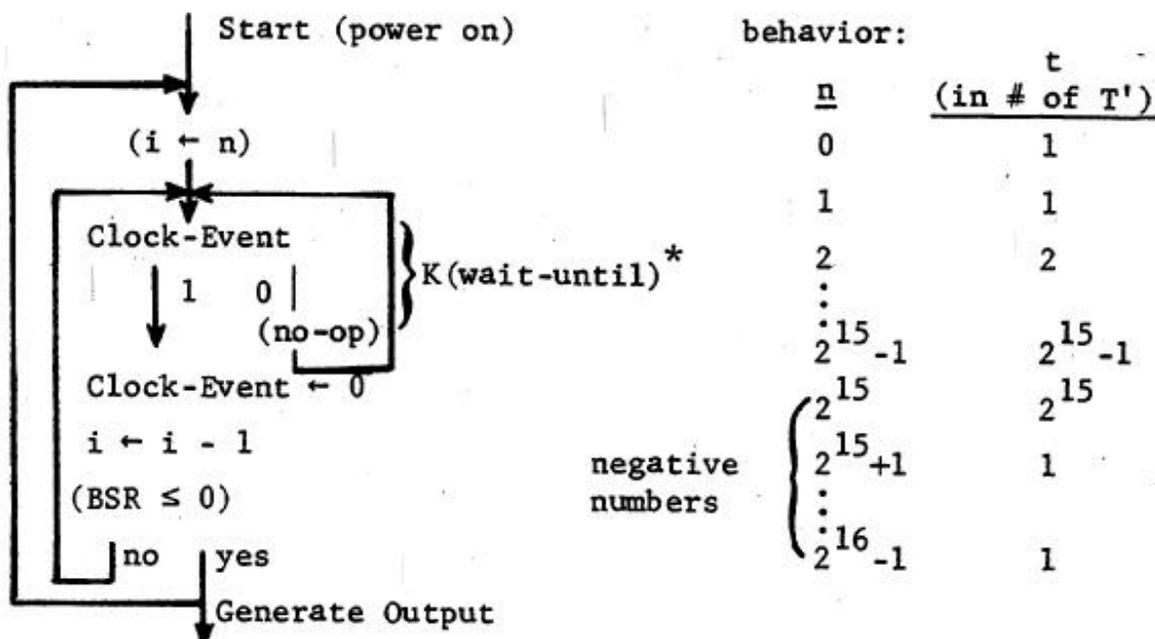


f. Control part for counting using K(wait-until) and K(for loop)

Fig. CD-2 (Part 2 of 2). RTM diagram for K(programmable (variable) clock) with 2 processes.

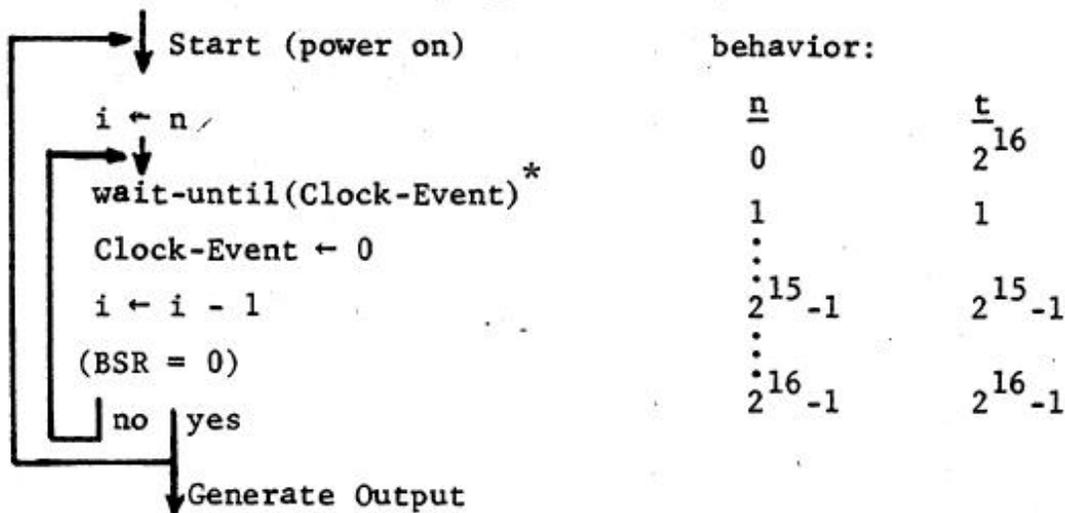


a. Data part and control part for time basis



\* Extended RTM macro - see following section

b. Control part for counter (signed numbers)-ill-defined



\* see following section

\* see following section

c. Control part for counting (unsigned numbers) using K(wait-until)-  
ill-defined for  $n=0$ .

Fig. CD-2 (Part 1 of 2).

104

[previous](#) | [contents](#) | [next](#)

## SOLUTION 1

This method assumes a variable,  $i$ , which can be counted down at each  $T'$  event; when  $i$  reaches 0 an output signal is generated and the count is reset to  $n$ , to begin the next countdown.

Figure CD-2a shows the data part of the design and one section of the control part that accepts time base clock pulse inputs from a conventional RTM  $K(\text{clock})$ . The data part requires an input interface,  $T(\text{input})$  to input the variable time count,  $n$ . A DMgpa contains  $i$ , the variable which is counted down to give the number of time base clock events. The time base is-generated by having the  $K(\text{clock})$  set the. DMflag, Clock-Event. That is, the Clock-Event flag is set each time a clock event occurs and a second, independent clock-timing control part counts these clock events.

The completely independent clock-counting control part can be designed directly; a design is given in Figure CD-2b. For this control part, count variable,  $i$ , is first initialized to  $n$ . A loop is then entered which waits until the Clock-Event flag becomes 1. Upon detecting the clock event,  $i$  is counted down by 1. The process is repeated for each occurrence of the Clock-Event flag becoming a 1. When  $i \leq 0$ , an output is generated and the clock counting process repeats. Thus, there are two independent control parts: the time basis generation which sets a Clock-Event flag; and a counting part which resets the Clock-Event flag.

There are several problems with this design as shown by the behavior in the figure which gives the number of outputs for each input value of  $n$ . If  $n$  is 0, the clock period is still 1, and if  $n$  is less than -1, then the clock period is also 1. One limitation arises from the fact that a signed two's complement number is assumed rather than an unsigned 16-bit number. In this way only half of the possible numbers are being used; since time is always positive, an unsigned convention should be used to give a greater range.

## SOLUTION 2

In Figure CD-2c the test is changed to a 0 test condition at the end of the loop, thus assuming an unsigned number (i.e., the other  $2^{15}$  numbers can also be used). Now a wider range of numbers can be used, but there is still a problem that a period of 0 is perhaps ill-defined. The behavior for the control flowchart given opposite the design shows that an input parameter value yields  $2^{16}$  unique periods. Perhaps a period of  $n=0$  should produce a better defined result. That is, for  $n=0$  either a large (infinite) number or zero of output events should be generated. The Figure CD-2c produced some low output rate. Note that in this design we have defined the  $K(\text{wait-until})$  macro which we will include in the set of Extended RTM's, because it is frequently used. The following example problem describes this macro.

## SOLUTION 3

The problem of an ill-defined case of  $n=0$  can be solved by first checking for 0 before the waiting loop

(see Figure CD-2d). A Kbranch has been added to check the initial condition before entering the loop.

One might think that the problem with this design is simply that we started off with a poor flowchart structure and then continued making modifications until a workable solution was found. The real problem, perhaps, is that the value of the number of times a loop is to be traversed should be checked before entering the loop. In this way anomolous cases will be sorted out without having the embarrassment of doing a loop one time when a parameter of 0 is specified. When neophyte Fortran programmers discover that giving an index count of 0 in

## ADDITIONAL PROBLEMS

1. What value is output in case of overflow? Is it the same value for all inputs that are large enough to create an overflow? What is the largest  $N$  for which the system can compute  $F(N)$ ? Given this value of  $N$ , is it clear why only one check was made for overflow in Solution 2?
2. Cost out the three proposals. Discuss the tradeoff between cost and speed. Why does it come out this way?
3. Is there a direct formula for calculating  $F(N)$ ? Find one and determine whether it provides an alternative basis for computing  $F(N)$ .
4. Suppose you had only 8 words of memory available (say as part of a larger core memory devoted to a total system). What sort of a fast Fibonacci generator could you build? Where would it come on the cost-speed graph of Problem 2?

- K(PROGRAMMABLE (VARIABLE) CLOCK)

**KEYWORDS:** Counting, clock, delay, integrating delay, wait-until, time-base, synchronization

The period  $T'$  of the basic RTM  $K(\text{clock})$  described in Chapter 2 is a constant, or more precisely, a variable which is set by a manual potentiometer adjustment. It seems desirable to have another kind of clock that has a period  $T$  that can be specified dynamically by a parameter from within an RTM system.

## PROBLEM STATEMENT

Design a clock, using RTM components, that has a variable period  $T$  that can be set by an RTM system.

## DESIGN CONSIDERATIONS

The proposed clock might have an overall RTM structure such as that shown in Figure CD-1. It would use a basic RTM ((clock) of constant period  $T'$  as an input. It would also have an input word,  $n$ , to specify the period  $T$  as a multiple of the base period  $T'$ , i.e.,  $T = n * T'$ , where  $n = 0, 1, 2, \dots, n\text{-max}$ . The clock would give an output control signal for each  $n$  counts of the basic clock, which has period  $T'$ . We can assume that  $n$  is held outside the system and is Accessible via a  $T(\text{input interface})$ .

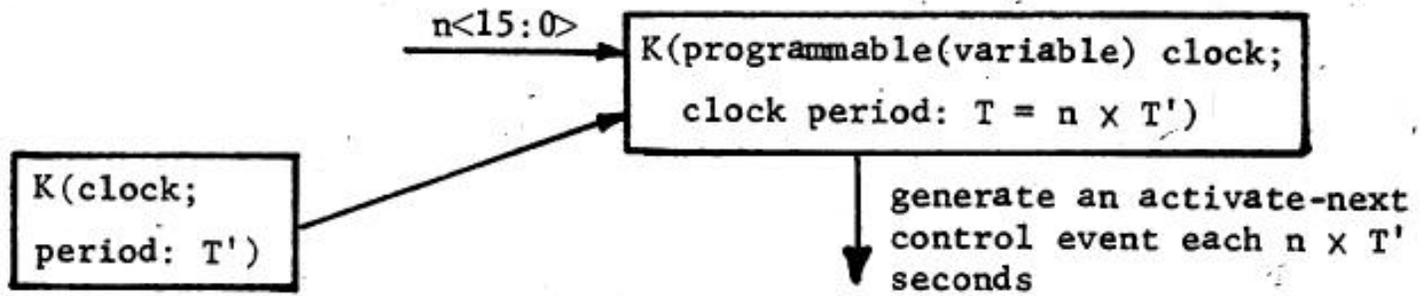


Fig. CD-1. Module diagram of K(programmable (variable) clock).

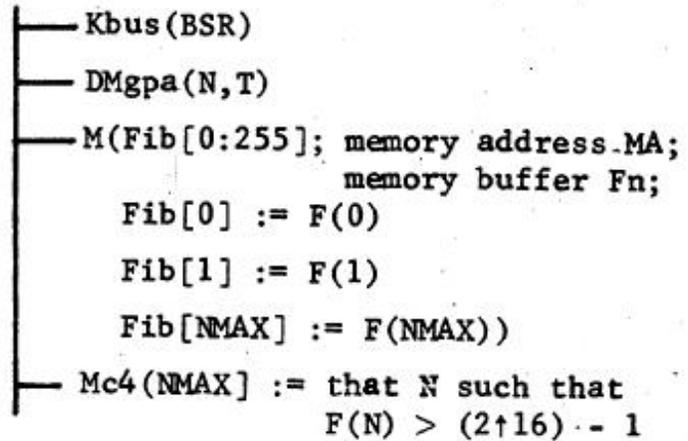
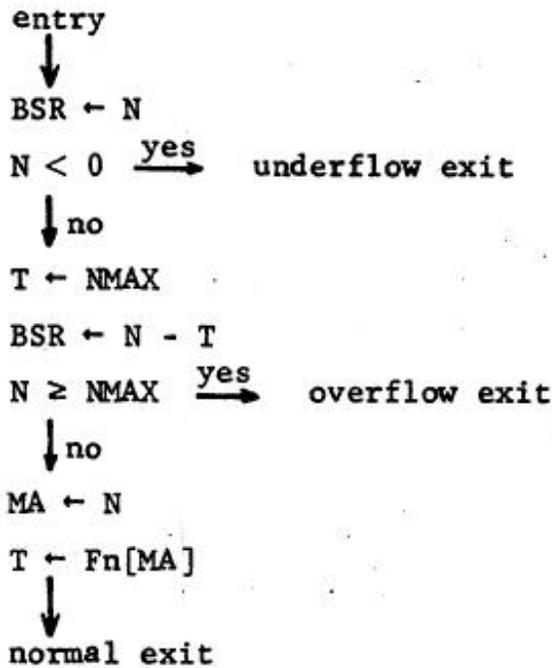


Fig. FIB-3. RTM system to compute the Nth Fibonacci number by a table look-up process.

### SOLUTION 2

This scheme employs a two-Bus structure and is shown in Figure FIB-2. This design uses the same preloop structure as that in the design of solution 1. The generation of the numbers in the sequence and the indexing on N are performed in parallel. K(parallel merge) modules provide the necessary synchronization of the two operations. The upper section of the loop generates the even-indexed terms of the sequence and the lower section generates the odd-indexed terms. Both sections of the loop decrement N, and the normal exit is taken when N = 0. As in solution 1, the detection of overflow passes control out the overflow exit. More parallel (concurrent) processes will be given in Chapter 4, and in subsequent design examples.

### SOLUTION 3

The fastest method for computing F(N) uses a table-lookup process and is illustrated in Figure FIB-3.

Those values of  $F(N)$  less than  $2^{16}$  may be stored and retrieved from a memory; the value for  $F(J)$  is stored in the  $j$ th word of the memory. The check for  $N < 0$  is made initially to prevent an illegal memory reference. A constant,  $NMAX$ , contains that value of  $N$  such that  $F(N) > (2^{16})-1$  and the overflow check is made against this value. With known  $N$ , this is just 2 Kevoke's.

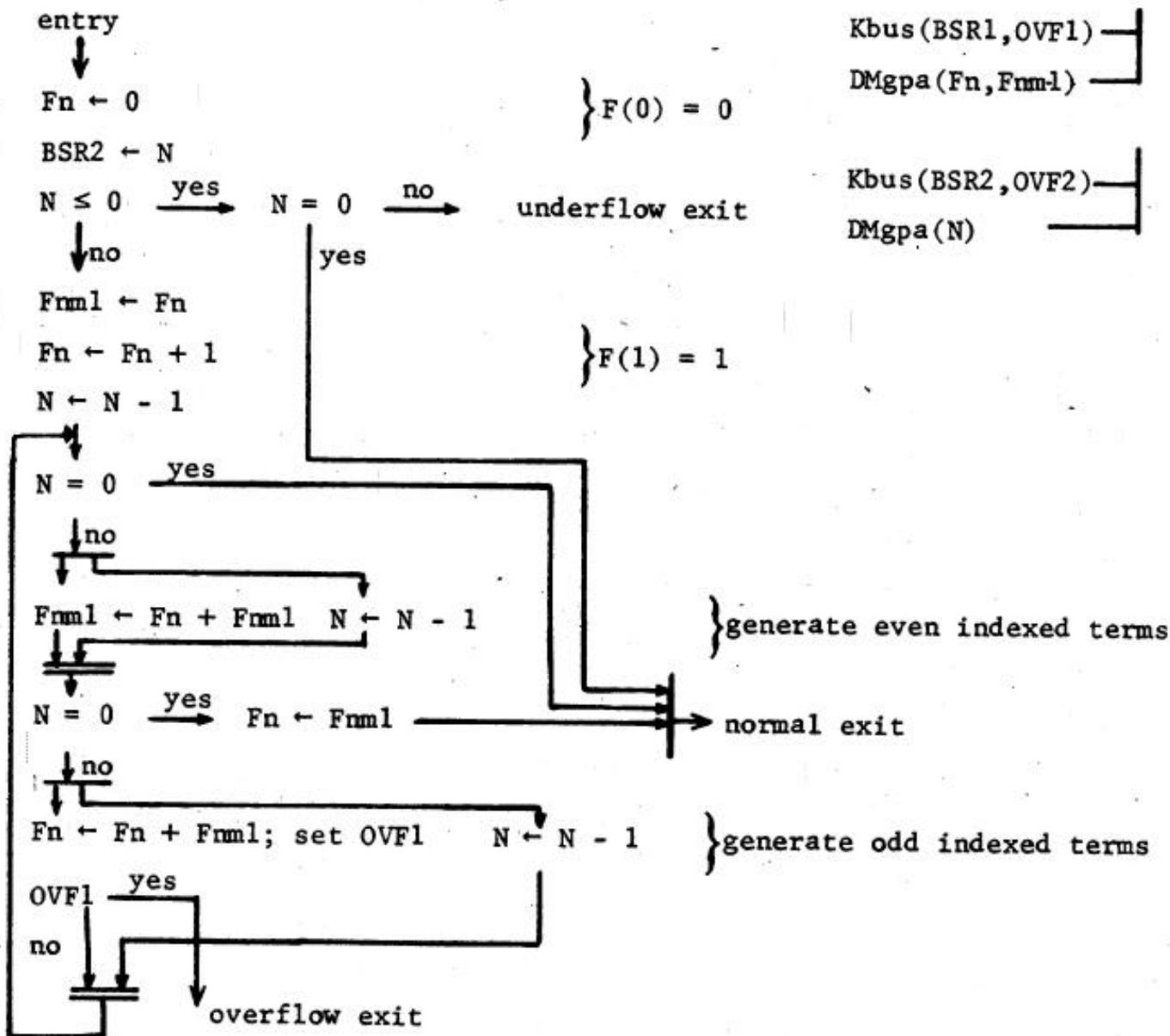


Fig. **FIB-2.** RTM system to compute the Nth Fibonacci number by a parallel process.

up according to the formula. The only concern is that all computations for  $F(1), F(2), \dots$  up to  $F(N)$  are being made, even though they are not themselves wanted. Having settled on a basic method does not determine fully the actual processing, as we show below by giving several designs.

**SOLUTION I**

Figure FIB-i shows a straightforward design for a Fibonacci number generator.  $F(0)$  and  $F(1)$  are generated at the beginning of the process and the normal exit is taken for  $N = 0, 1$ ; for  $N$  greater

than I but small enough not to cause overflow, the loop is performed  $N-1$  times yielding  $F(N)$ ,  $N$  is decremented by 1 on each pass through the loop, and the normal exit is taken when  $N = 0$ . For those  $N$  such that  $F(N) \geq 2^{16}$ , overflow will occur and the overflow exit is taken.

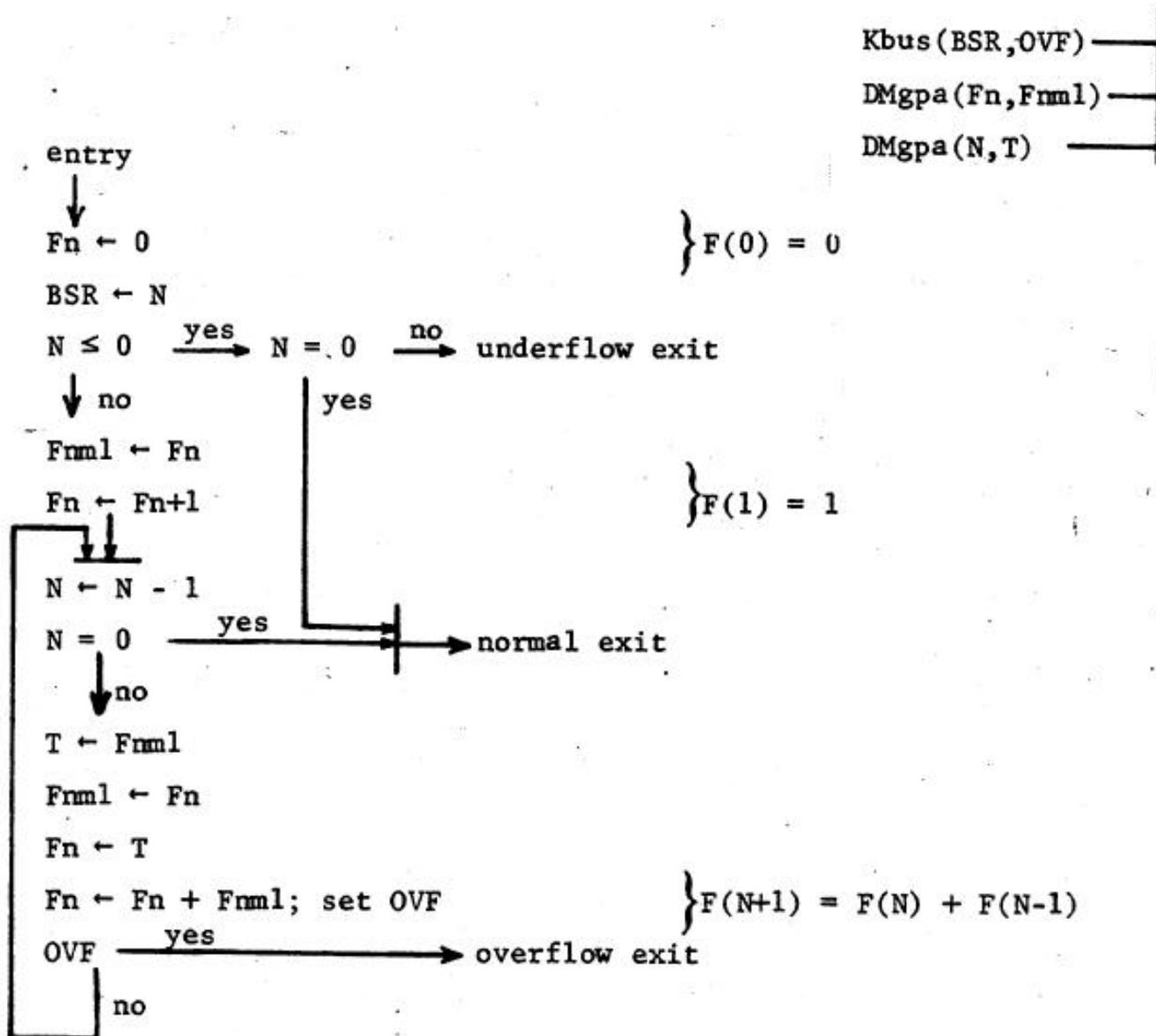


Fig. FIB-I. RTM system to compute the Nth Fibonacci number by a sequential process.

The second question to ask is about the various limits of the computation. The natural representation for  $F(N)$  is in a single 16-bit word, which means that  $F(N)$  cannot be greater than  $2^{16}-1$ . How fast does  $F(N)$  grow? We can answer that question by finding a simple upper bound for  $F(N)$ . For example:

$$F(N + 1) = F(N) + F(N-1)$$

$$F(N+1) \geq F(N-1) + F(N-1) = 2 * F(N-1)$$

We simply ignore the additional contribution from  $F(N)$  being greater than  $F(N-1)$ . The above formula shows that  $F(N)$  grows by at least a factor of 2 for each increment of 2 in  $N$ ; thus by the time  $N = 32$ ,  $F(N) \geq 2^{16}$  and has overflowed the word. Thus, the design specifications must include an overflow return as an additional part of the output representation. In addition a check should be made to insure that  $N \geq 0$ ; an underflow exit should be included for the condition that  $N < 0$ .

The third question is to decide on a basic method of computation. The simplicity of the recurrence relation suggests computing  $F(N)$  simply by adding it

least four variable locations is needed with a data operations part, a DMgpa, to carry out the updating of  $\langle v1 \rangle$ . The control part is shown on the left. In this implementation we have placed an additional data operations part to carry out the computation of  $(v1 - v4) * \text{sign}(v3)$ . This operation could have been carried out in the DMgpa however, Thus, we have defined the meaning (or semantics) of the Algol loop in terms of RTM primitives, although we have made the simplifying assumption that the variables are the two's complement 16-bit integers represented by the RTM system. The control part is somewhat like a K(subroutine), together with a number of implicit connections to the DM part to manipulate the controlling loop variables.

## APPLICATION

Figure FL-3 shows an implementation of the sum-of-integers problem, Chapter 2, using the K(for loop). Note that in this case only one control step has been saved. There is also an implicit assumption that the variable  $\langle v4 \rangle$  has been loaded into the DM part of the for loop module. Hopefully this implementation is somewhat easier to understand as it separates the control of the algorithm from the part of the algorithm performing the calculation.

## PROBLEMS

1. Redesign the K(for loop), Figure FL-2, using sequential Ke's to carry out the  $(vi - v4) * \text{sign}(v3) > 0$  computation.
2. Since there is a problem with the K(for loop) having access to variables which are used in the rest of the system (do part), as a practical matter almost all K(for loops) would have to be tailored to the particular application. But the customizing process variables would not have to be duplicated as they can be taken from other parts of the system. Show a special K(for-loop) for the sum- of-integers problem, Figure FL-3.
3. Use the K(for-loop) and K(conditional-execute) to solve the ones count problem.

## FIBONACCI NUMBER GENERATOR

**KEYWORDS:** Functions, generation, overflow, underflow

- The Fibonacci sequence is defined by the relations:
  - $F(N+1) = F(N) + F(N-1)$  with  $F(0) = 0$  and  $F(1) = 1$

The first few values of the sequence are: 0,1,1,2,3,5,8,... The Fibonacci sequence has a fascinating

history, showing up in many odd guises. For our purposes, they can illustrate the task of function generation where some implicit relationships are given, rather than a direct explicit formula.

## PROBLEM STATEMENT

Design a Fibonacci number generator that takes  $N$  as input and computes  $F(N)$ .

## DESIGN CONSIDERATIONS

The first design question is to ask about the interface between the system (the number generator) and the external world: what information is to be provided as input and output and in what representation? The matter appears quite simple here: the input is a 16-bit number held in a 16-bit word; similarly the output is a 16-bit number representing  $F(N)$ .

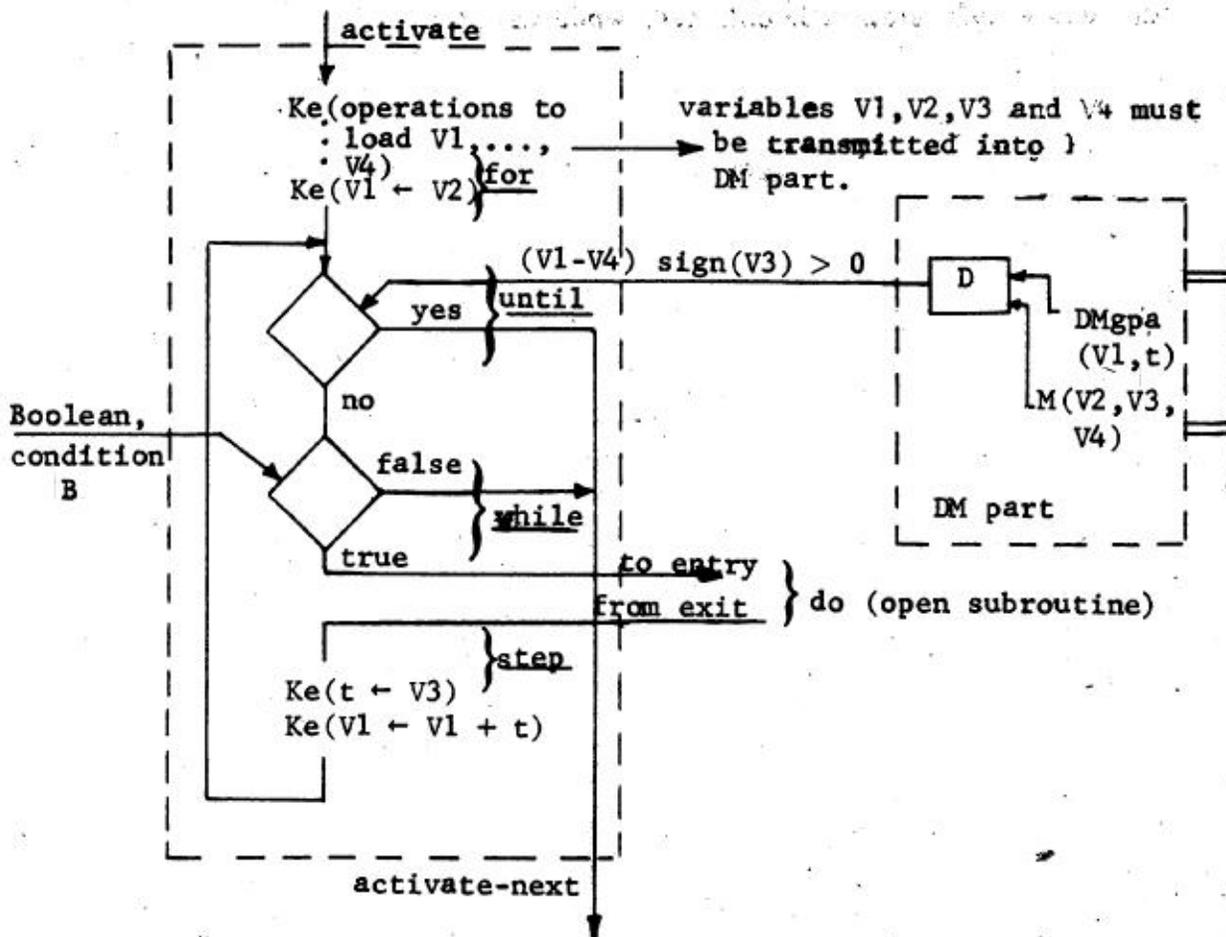


Fig. FL-2. RTM diagram of K(for loop) ERTM implementation.

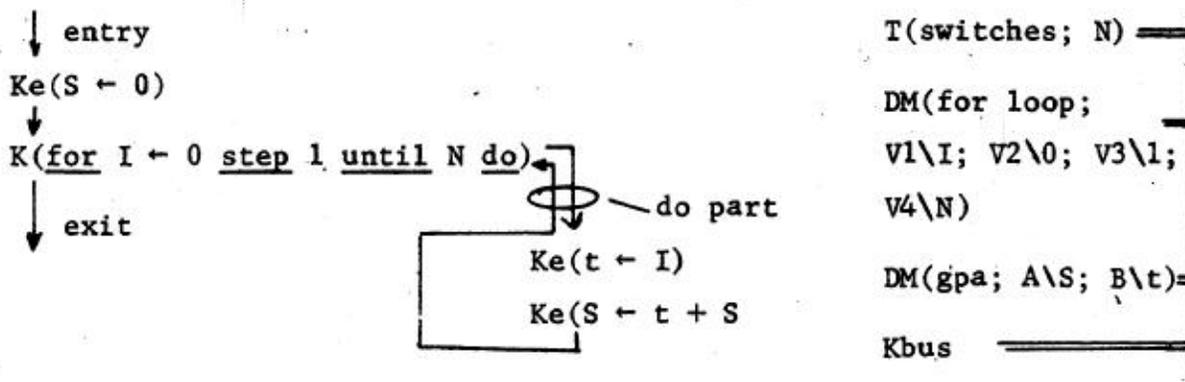


Fig. FL-3. RTM diagram for sum-of-integers using K(for loop).

[previous](#) | [contents](#) | [next](#)

**for <v1> ← <v2> step <v3> until <v4> while <B> do S;**

where the <v>'s indicate register real (floating point) or integer variables or constants which are substituted, <B> is a Boolean condition, and S is the statement or statements to be executed. In practice <v2>, <v3> and <v4> are often constants and can be positive or negative. Actually the Algol statement is even more general than this since <v2> can be replaced by a list of variables for <v1>, but we will not consider this case.

In terms of the for-loop the sum-of-integers problem, from Chapter 2, can be expressed in Algol as:

```
integer N,S,I;
S ← 0;
for I ← N step -1 until 0 do S ← S + I;
```

or alternatively

```
integer N,S,I;
S ← 0
for I ← 0 step 1 until N do S ← S + I;
```

Note that in these cases the while B part is not used. This part is a further test for deciding whether to continue iterating and may replace the until part.

## PROBLEM STATEMENT

The for-loop extended RTM is fairly complex and is classified as a K module, consisting of both a control and data part as shown in Figure FL-1.

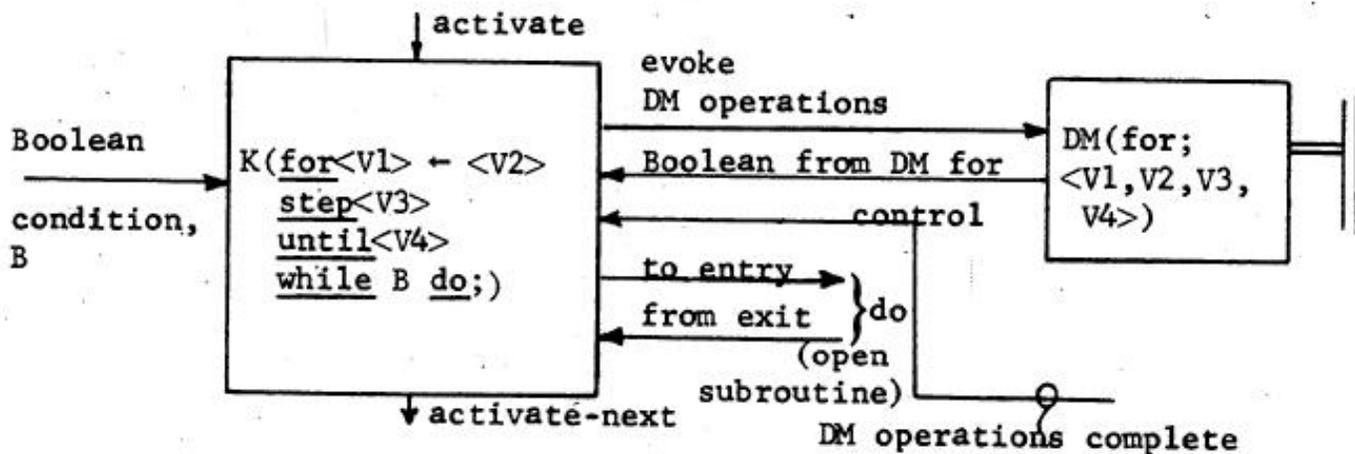


Fig. FL-I. Module diagram of K(for loop) ERTM.

Design a K(for-loop) which assumes a macro (open subroutine) will be used for the do part. Use standard RTM's to specify the structure. Assume that if  $\langle v3 \rangle$  is positive the test for until-completion is  $(v1-v4) > 0$  and for  $\langle v3 \rangle$  negative it is  $(v4-v1) > 0$ . These two conditions can be expressed as:  $(v1-v4) * \text{sign}(v3) > 0$ .

SOLUTION

Figure FL-2 shows an RTM implementation of the K(for-loop). Memory for at

is called. In this case the direction of the shift (i.e.,  $\times 2$  or  $/2$ ) is controlled by the sign of  $N$ . Also this sign determines whether incrementing (+1) or decrementing (-1) is to occur. The absolute value of  $N$  may lie outside the range of 16, thereby causing shifts in  $V$  which leave 0 as the result.

If we apply the K(conditional execute), the control part for this problem is slightly simplified as in Figure SO-4.

## ADDITIONAL PROBLEMS

1. The sign-magnitude representation for numbers was shown in Figure SO-1. Modify the design of the previous solution to perform  $\times(2^N)$  and  $/(2^N)$  operations using this representation.
2. Further generalize the previous problem so that it has three control inputs for the logical, circular and arithmetic shift cases.  $N$  specifies the direction and number of shifts to be carried out.
3. Can you find a way to decrease the time for the parameter shift operation case given in Figure SO-3 and problem I above?
4. A related pair of operations to logical shifting permits the loading, and storing of a field of data within a single word. Design the RTM system for loading and storing a field of up to 16 bits, which takes parameters bit,  $b$ , field length  $I+1$  (varying from 1 to 16 bits), and word,  $A$ , within a DMgpa. For loading, field  $A\langle b+1: b \rangle$  is placed in  $data\langle l:0 \rangle$ . On storing,  $data\langle l:0 \rangle$  replaces  $A\langle b+1: b \rangle$ , where  $0 \leq b \leq 15$  and  $b + 1 \leq 15$ .
5. Design RTM systems to test the various cases given in Problems 1 to 4.

## FOR LOOP EXTENDED RTM

**KEYWORDS:** For loop, DO statement, ERTM, sharing DM modules, sum-of-integers

An important control operation encountered in almost every application is a set of operations which are iterated a variable number of times. This is called a loop, and in this example we shall build an ERTM to carry out loop control. First, however several programming language structures related to this problem will be discussed.

### Fortran DO Statement

The following program uses Fortran to compute the sum-of-integers problem from Chapter 2:

```
      INTEGER N, S, I  
      S = 0  
      DO 10, I = N, 1, -1  
10    S = S + I
```

This may not be correct for some Fortran implementations because the DO statement increment is -1. It is given because it corresponds to our implementation in Chapter 2. The DO loop also has the problem that statement 10 must be interpreted at least once, although that causes no difficulty for the example shown.

### Algol For Statement

In Algol a more general form for loop control is allowed and, in addition, a loop can be executed zero times if desired. The general form of the Algol for statement is:

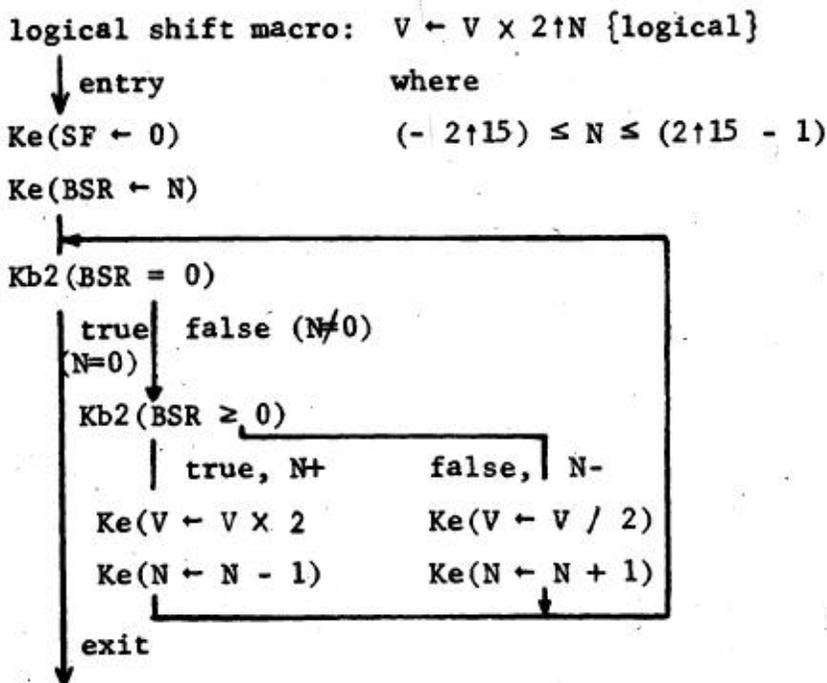
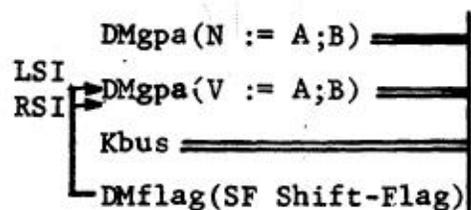
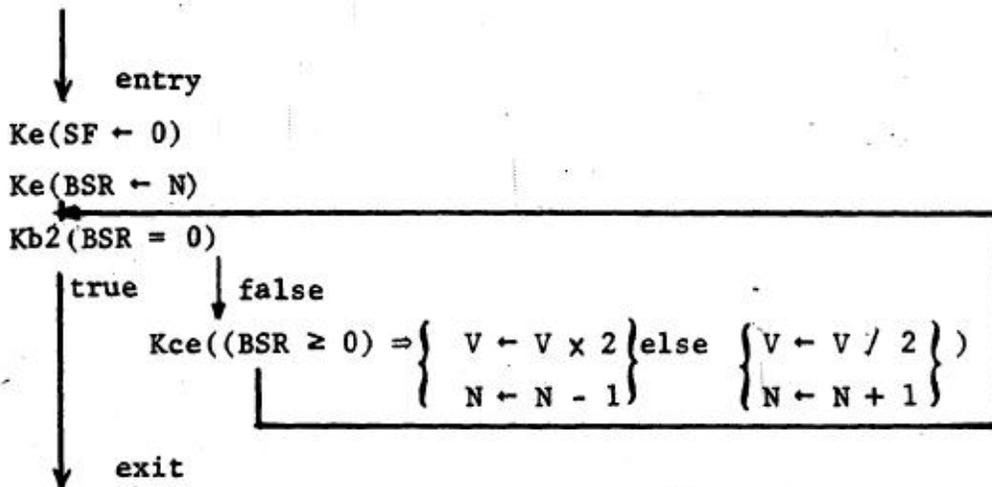


Fig. SO-3. RTM diagram for logical shift  $\times 2^{\uparrow N}$  macro.



**Fig. S0-4. Control part for logical shift  $x2\uparrow N$  assuming a K(conditional execute).**



[previous](#) | [contents](#) | [next](#)

circular shift is in fact necessary to put the correct digit into bit 0. Again, if the number 6 were originally represented in a register, as the register is shifted left three times its contents would be 12, 24, and 48 respectively. If a number is created during left shifting that exceeds the capacity of the register, this condition is indicated by setting an Arithmetic Overflow Flag (ArOvF). For 1's or 2's complement notation, this condition is detected when a different bit is transferred into the sign position. That is, take the set of large 16-bit positive numbers which have bit positions  $01\sim\dots\sim$ , where  $\sim$  indicates the bit may be either 0 or 1. This representation indicates numbers in the range  $2^{14} \leq n \leq 2^{15} - 1$ . Shifting left would call for  $2^{15} \leq n \leq 2^{16} - 1$ , which is impossible to represent in a signed 16-bit register, which only has room for storing positive numbers up to  $2^{15} - 1$ . Large negative numbers can cause overflow in a similar way. Notice that for sign-magnitude representation, leaving the sign alone and shifting bits 14:0 left does not cause the sign to be lost when an overflow results, but the result is still erroneous. Under these conditions the result is:

$$\text{result-number} - \text{original-number} \pmod{2^{15}}$$

## PROBLEM STATEMENT

Assume that a number to be shifted is held in the A register of a DMgpa. Design an RTM system (subprocess) to perform the three shift operations, logical, circular, and arithmetic (2's complement), for both the left ( $\times 2$ ) and right ( $/2$ ) cases as indicated in Figure SO-1. Assume that an arithmetic overflow will cause a 1 to be placed in the DMflag (Arithmetic-Overflow-Flag (ArOvF)).

## SOLUTION

The solution can be written down directly in terms of the data part statement as shown in Figure SO-1. The RTM diagram for the shift operations is given in Figure SO-2. In both the left and right shift cases the approach is to first establish the input bit, Shift-Flag (SF), to be shifted into A, and then carry out the appropriate shift operation. In the case of the arithmetic left shift ( $\times 2$ ) the ArOvF (flag) may be set.

## RELATED PROBLEM

The previous case only provided a single operation of  $\times 2$  or  $/2$ . More generally, the number of shifts is also a parameter in the operation. That is, consider the three operations:(1)

$$A \leftarrow A \times 2^N \text{ \{arithmetic\}}$$

$$A \leftarrow A \times 2^N \text{ \{logical\}}$$

$$A \leftarrow A \times 2^N \text{ (circular)}$$

where  $N$  may be either positive, zero, or negative, giving a  $\times 2^N$ , null or  $/2^N$  operation. Design an RTM subprocess which takes  $A$  and  $N$  as input parameters and performs the appropriate operation for the logical shift case.

## SOLUTION

Figure SO-3 shows the RTM diagram which solves the above problem. The parameter,  $N$ , to specify the number of shifts is held in a DMgpa when the macro

-----

1. Hereafter arithmetic shifts will be assumed to be 2's complement, unless otherwise stated.

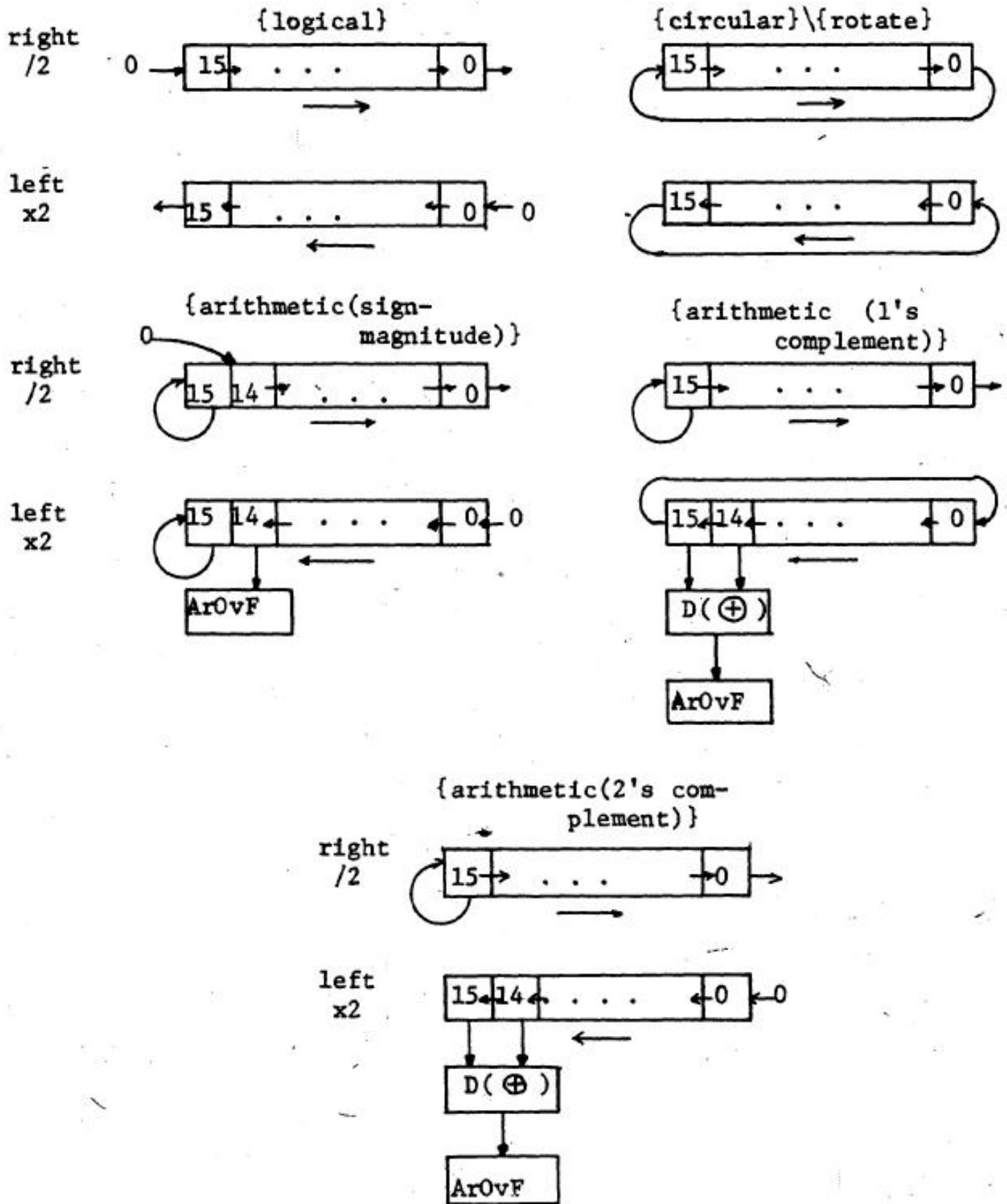


Fig. SO-1. Examples of shift operations for various data representations.



3. Can a Kce activate a Kce which activates a Kce which ...?
4. Use Kce's in the redesign of the ones counter, Figure OC-2.

## LOGICAL SHIFT, CIRCULAR SHIFT (ROTATE), AND ARITHMETIC SHIFT OPERATIONS

**KEYWORDS:** Number representation, single loop control,  $\times 2$ ,  $/2$ , arithmetic operations, shift, rotate

There are three basic operations, collectively known as shifts, that are often needed in digital systems design (especially general purpose computers). In shift operations bits are moved by one or more bit positions within a register, either to the left or to the right. Since the arithmetic value of a bit is either increased or decreased by a factor of two for each position that it is shifted, these operations are often denoted as either multiplication or division of the contents of the register by 2, for left and right shifts respectively. Strictly speaking, however, this is just a shorthand notation for what actually happens during the shift. Depending on the data-type of the word under consideration, there are actually three different kinds of shifts: logical shift, circular shift (rotate), and arithmetic shift. The actual paths for the movements of bits in the registers for three shift operations are shown pictorially in Figure S0-1.

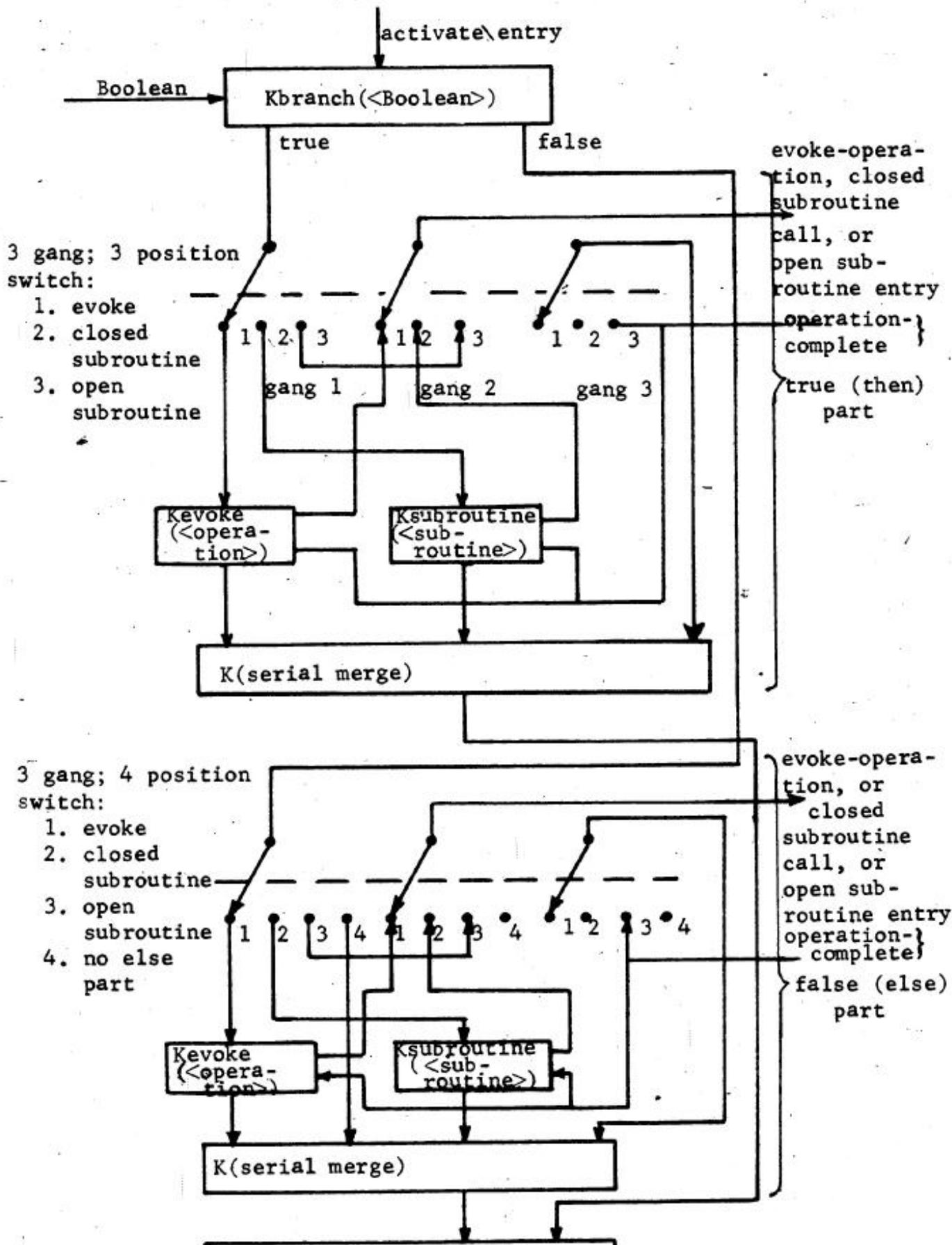
The logical shift merely moves the various bits to the right or left one position, and reads in a 0. The operation is not used for arithmetic, but might be used to isolate a field of bits (contiguous set of bits). The circular shift, also called rotate, just connects the end bits of the register and no bits are lost or introduced in the shifting process - they are merely moved.

The true arithmetic shift ( $\times 2$ ,  $/2$ ) is slightly more complicated. For one thing, there are three common integer number representations to consider, each of which is treated slightly differently: sign-magnitude, 1's complement, and 2's complement. In sign-magnitude representation, bit 15 is the sign bit (1 for -, 0 for +), and bits 14:0 are the binary magnitude of the number. In 1's complement notation, positive numbers are the same as in sign-magnitude notation, but negative numbers are represented as the 1's complement (bit by bit complement) of their positive counterpart. In 2's complement notation, positive numbers are again the same as in sign-magnitude notation, but negative numbers are represented as the 2's complement of their positive counterpart. The 2's complement of a number is the 1's complement, incremented by 1 (i.e., 2's complement = 1's complement + 1). Notice that in 1's complement and 2's complement notation the convention for the sign bit is maintained, i.e., bit 15 is 0 for positive numbers and 1 for negative numbers.

On dividing numbers in each of these representations by 2 (see Figure 50-1), the sign should not change, thus the sign remains the same during a right shift (or alternatively, one could claim that a copy of the sign is shifted into the sign position). Notice that in sign-magnitude notation, a 0 must be shifted into bit

14. Consider now an example in which the number 6 is originally represented in a register. As the register is shifted right three times, its resultant contents become 3, 1, and 0 respectively. Clearly information is being lost as bits are shifted out the right side of the register. In some digital systems this might possibly be indicated by an underflow flag indicating that a number which is smaller than the register can represent has occurred.

On multiplying numbers in each of these representations by 2, again the sign should not change, Furthermore, note that in 1's complement notation a left



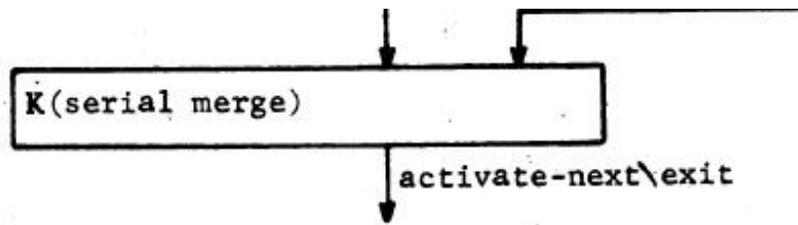


Fig. CE-3. RTM diagram for Extended RTM, K(conditional execute), general case.

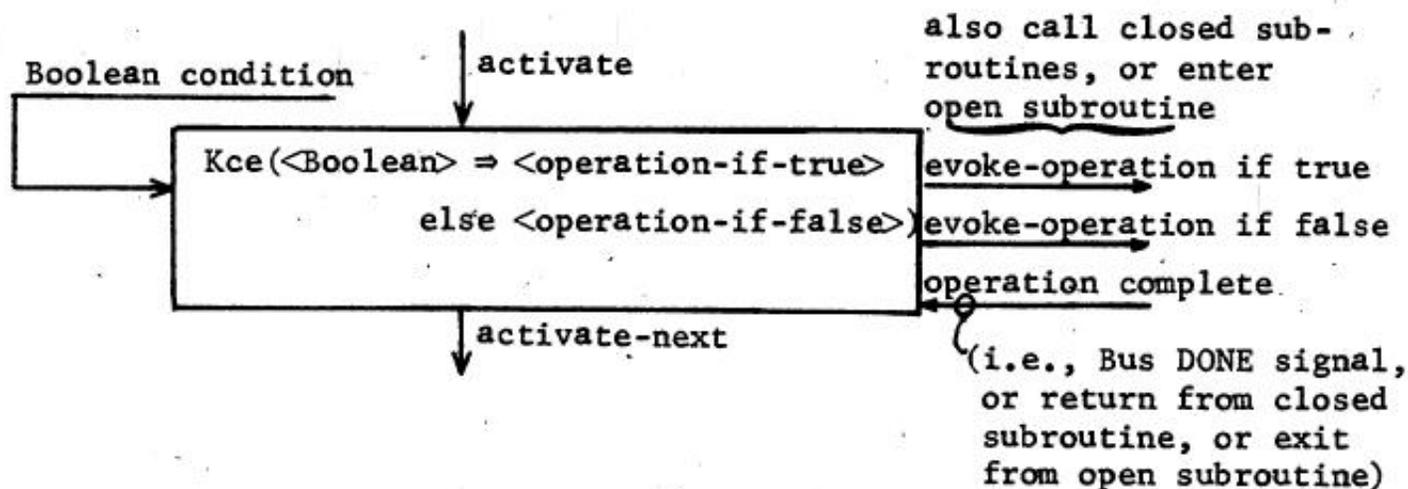


Fig. CE-1. Module diagram of Extended RTM, K(conditional execute).

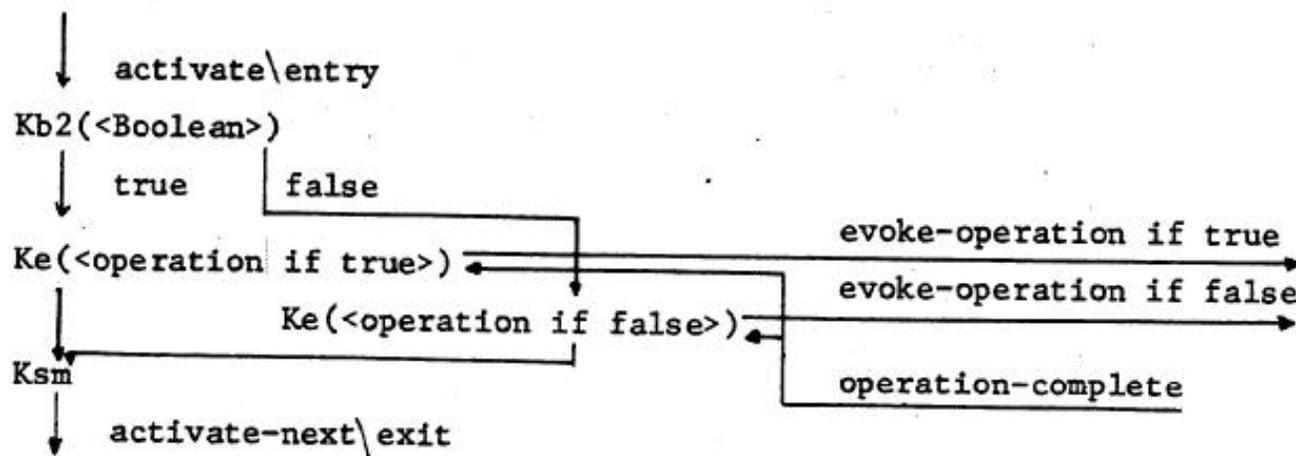


Fig. CE-2. RTM diagram of Extended RTM, K(conditional execute).

### ADDITIONAL PROBLEMS

1. Design a part of an RTM system using Kce (including Kce switch positions) for the following:

a.  $B \Rightarrow (C \leftarrow F)$ ;

b.  $B \Rightarrow (\text{multiply subroutine}) \text{ else } (X \leftarrow 0)$ ;

d.  $B \Rightarrow (\text{GOTO } Q) \text{ else } (\text{GOTO } Y)$ ; GOTO is used here to denote change of control flow; what might Q and Y be?

e.  $B \Rightarrow (X \leftarrow 1; \text{next } Y \leftarrow 1) \text{ else } (V \leftarrow 1; \text{next } Q \leftarrow 0);$

2. Design a Kce which only has an "if B then S" where S is a single operation that can be evoked.

is both the count and a number shifted with its most significant one in the bit 15 position. Design an RTM system to normalize a word.

## CONDITIONAL EXECUTE EXTENDED RTM

**KEYWORDS:** Extended RTM, conditional execution, if-then-else, IF, closed subroutine, open subroutine

### PROBLEM STATEMENT

Design an Extended RTM/ERTM which behaves similarly to the Algol "if B then S-true else S-false", and Fortran "IF" statements. The control is to sense the condition of a Boolean input, B, and depending on whether the input is true or false, evoke either the S-true or S-false control parts. A block diagram of the desired ERTM is shown in Figure CE-1.

### DESIGN CONSIDERATIONS

There are several possible structures for such an ERTM depending on the operations being initiated. These operations could be: a single register transfer operation as given in a K(evoke); a closed subroutine (subroutine) which would be called and produce a return as in a K(subroutine call); an open subroutine (macro) which would be called and produce an exit; and a transfer of control flow as in a K(branch).

### SOLUTION 1

Before presenting a general solution, we illustrate the simple case in which the operation is a K(evoke), i.e.  $Kce(B \Rightarrow \text{evoke-if-true else evoke-if-false})$ . The structure could be realized as shown in Figure CE-2. Namely, there are two K(evoke)'s, a K(branch; 2-way), and a K(serial merge).

### SOLUTION 2

A general solution is required to handle all cases stated in the above problem. Assuming such a module is to be specially fabricated and the module is to be statically switched to handle the various cases, two separate switches for specifying the true and false operations are required. Figure CE-3 shows the structure of a Kce which has those capabilities.

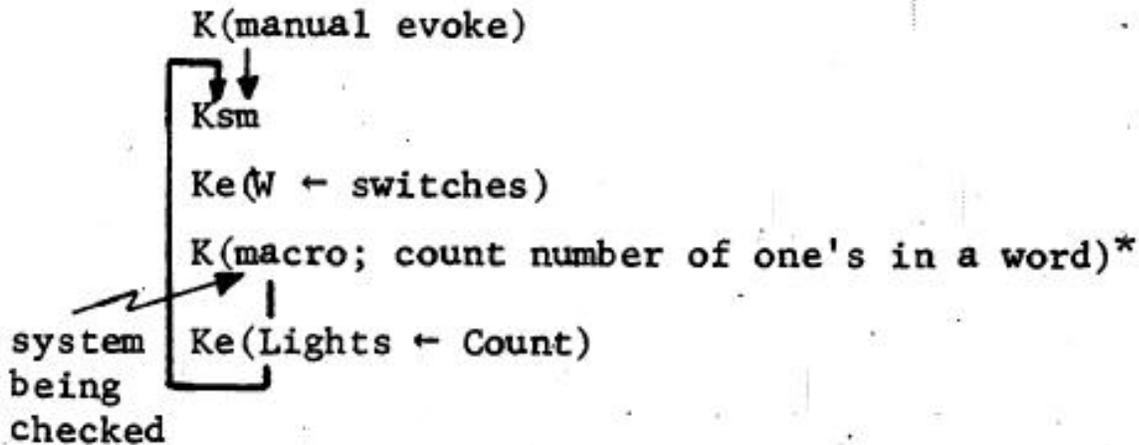
The implementation consists of four basic parts: the Kbranch makes the decision (if part); the true part then carries out the appropriate call if the Boolean is true; the false (else) part carries out the call if the Boolean is false; the merge brings the true and false parts together at completion. The false part is similar to the true part, except that it has an additional switch position which allows the false part to be bypassed,

permitting the statement: B=>operation-if- true. A three-position manual switch selects the type of output required for the true part (then case), giving a standard evoke operation (from the Kevoke), a subroutine call (from the Ksubroutine), or a call to a macro (direct). The three gangs. (or decks) of the three-position switch operate in parallel; that is, when the switch position is moved (shown in position 1) all decks are switched. The first gang selects the controlling input type; the second gang routes the output of the appropriate call control to the output (evoke-operation, subroutine call, or macro entry); and the third gang routes the exit signal back to the activate-next output when a macro is' used. The false (else) part fourth position is used as a bypass when only a true (then) part is used.

With these capabilities a K(conditional execute) can be used to activate any subprocess including the trivial subprocess consisting of a single K(evoke).

-T(lights and switches) -||  
 to remaining system

a. Data part



\* may alternatively be a K(subroutine call)

b. Control part

Fig. OC-6. RTM diagram to test the "Count number of ones in a word" RTM system.

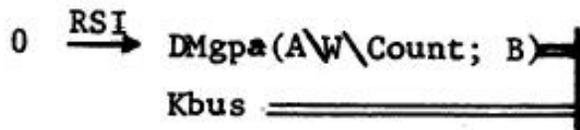
TESTING

The above four designs represent solutions to the isolated problem of designing subprocesses to compute the number of ones in a word. These designs create an additional problem of testing whether they in fact work. Each system can be checked by the additional components given in Figure OC-6. A T(lights and switches) is added to the data part. The control part causes the switches to be read into W, the subprocess to be executed, and the results to be displayed. By observing the relationship between lights and switches the system correctness can be verified to a certain extent.

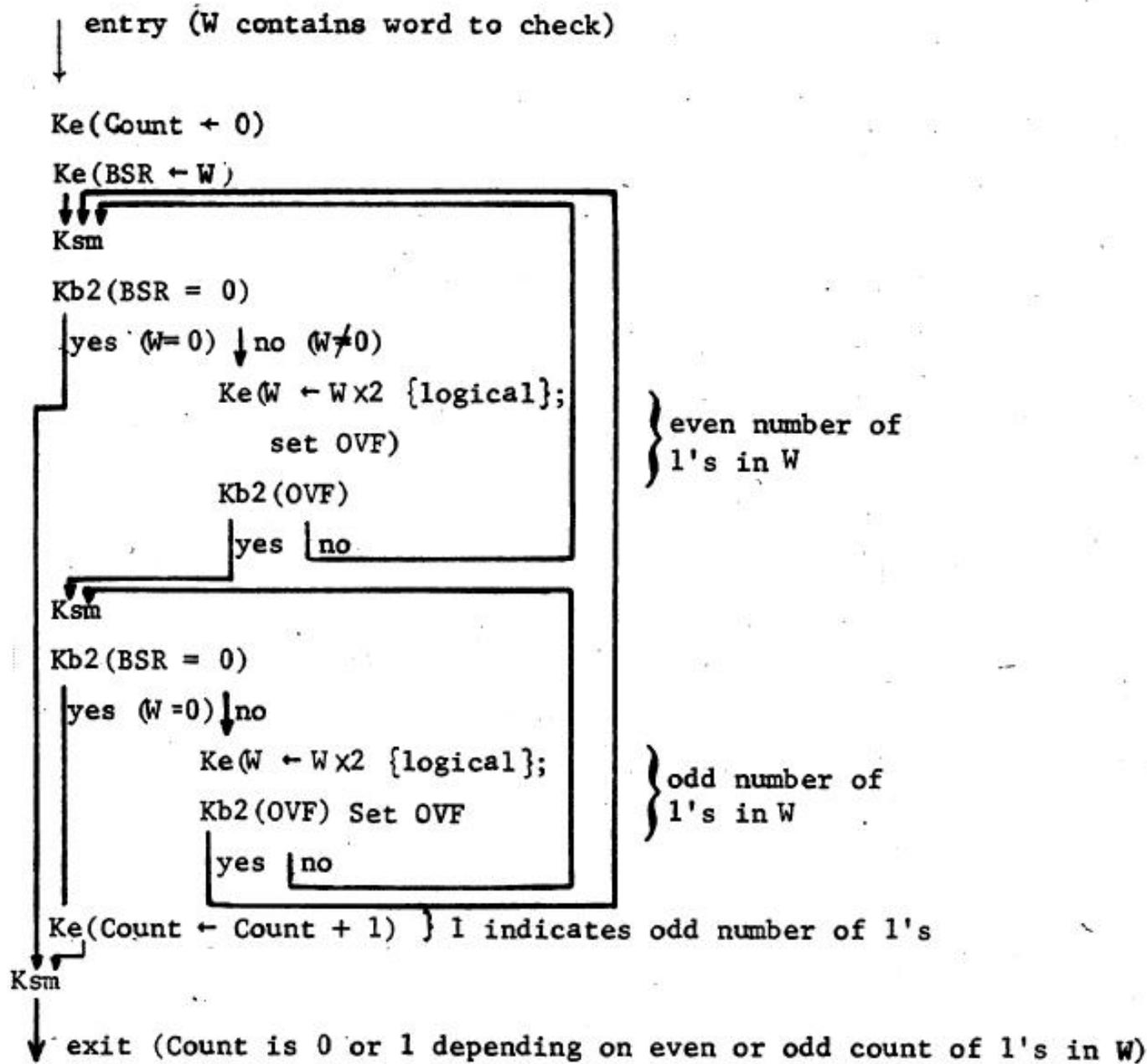
ADDITIONAL PROBLEMS

1. Why does Solution 2 take an additional loop time? What is the average time for Solution 2?
2. What is the operation time and system cost for Solution 3?

3. The solutions given assume that the data is shifted left (i.e.,  $A \leftarrow Ax2$ ). Since the B register of a DMgpa can be shifted right, then the best solution (in terms of minimum cost and time) may be based on an assignment of A\Count and B\Word\W in a single DMgpa. Thus the arithmetic operations:  $Count \leftarrow Count+1$ ; and  $W \leftarrow W/2$  are both permitted. Carry out this design.
4. Using the flowchart to hold data, as in Figure OC-4, together with 16 different constants, design an RTM system which counts the number of ones in a word without using shifting.
5. Using a specially wired M(transfer register), design a faster ones counter for both the count, and modulo 2 count cases.
6. Plot the time vs. cost on a graph for Solutions 1 to 3, and your designs in problems 3, 4 and 5.
7. Another operation, normalization, is used in certain arithmetic operations and is similar to counting ones. Normalization consists of determining a count which corresponds to the bit position of the most significant one in a word. The result



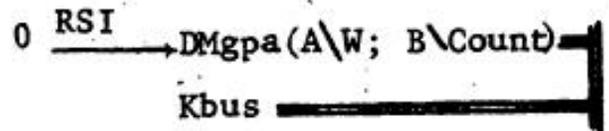
a. Data part



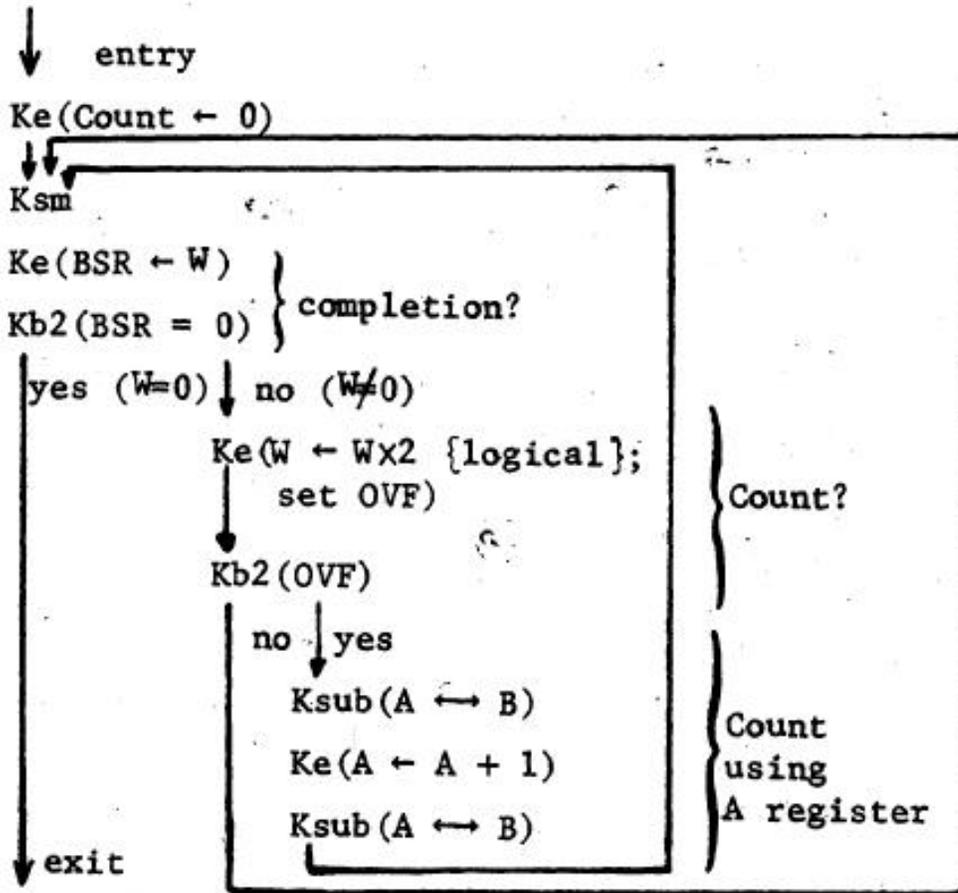
b. Control part

Fig. OC-5. RTM diagram to count the number of ones in a word, modulo 2 (i.e. parity), holding result within the control part.

Figure OC-5 gives an RTM diagram for a considerably faster method. Also note that only one register, A, is used for word W (at entry) and the result, Count (at exit). This method uses the flowchart to hold data; in this case, the location of the control within two loops indicates whether an odd or even number of ones have been detected. At the exit Count is either 0 or 1.

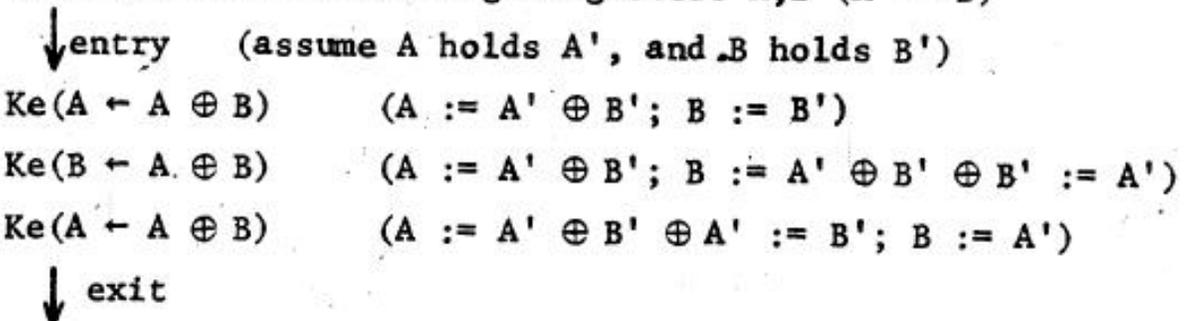


a. Data part



b. Control part

subroutine to interchange registers A,B (A ↔ B)



c. subroutine control part

**c. subroutine control part**

Fig. OC-4. RTM diagram for counting number of ones in a word using one DMgpa.

shown in Figure OC-3. In doing this re-ordering we have probably decreased the understandability of the algorithm, and we have certainly made it more difficult to describe whether it operates correctly. The algorithm is correct, but the testing for completion is slightly erratic, since either  $W$  or  $Count$  is checked for 0. Therefore the algorithm will take one more loop time than really needed to terminate.

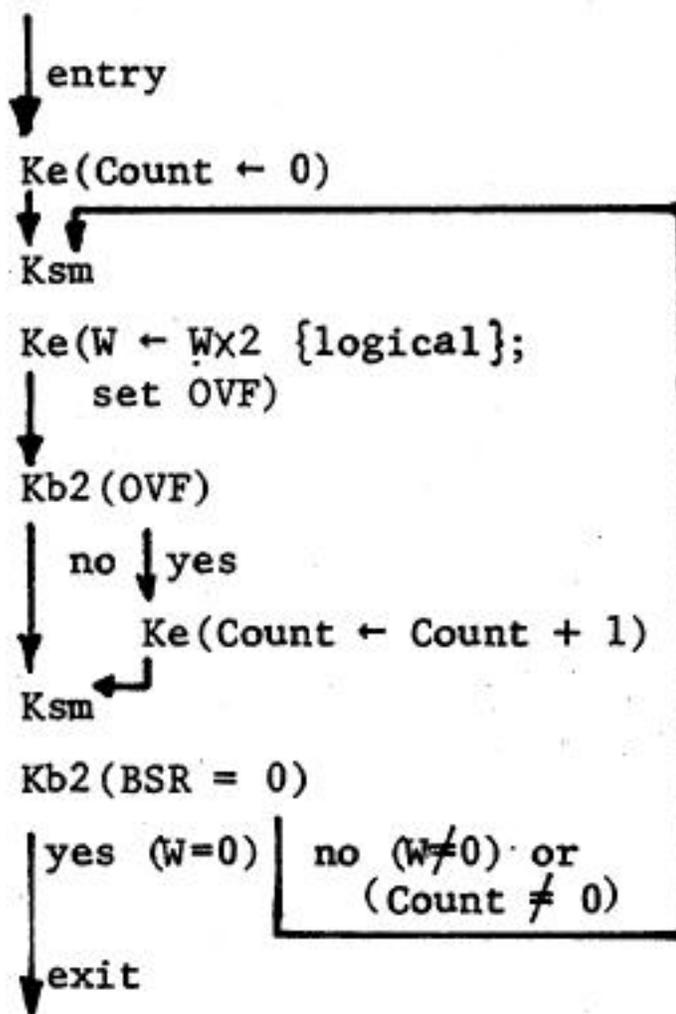


Fig. OC-3. RTM diagram (alternative) for counting ones in a word.

### SOLUTION 3

Figure OC-4 gives a solution which uses only a single DMgpa. That is, the DMgpa is time-shared, being used for arithmetic operations on  $Count$  and  $W$ . Since the A register can only be used for counting and shifting, the two registers must be interchanged before counting can be carried out in B. Thus time is traded-off for lower cost.

## RELATED PROBLEM

Another problem related to counting the number of ones in a word is counting the ones modulo 2, that is, counting whether an even number or odd number of ones appear in the word. The result is to be 0 for an even number and 1 for an odd number. A simple calculation of this form is often used when transmitting data via some error-prone information medium. By adding this count, called a parity bit, at the transmitter, and then checking the data and parity bit at the receiver, single bit transmission errors are detected.

## SOLUTION TO RELATED PROBLEM

The direct computation for parity based on the previous problem simply uses the previous solutions and takes the least significant bit of Count to indicate parity.

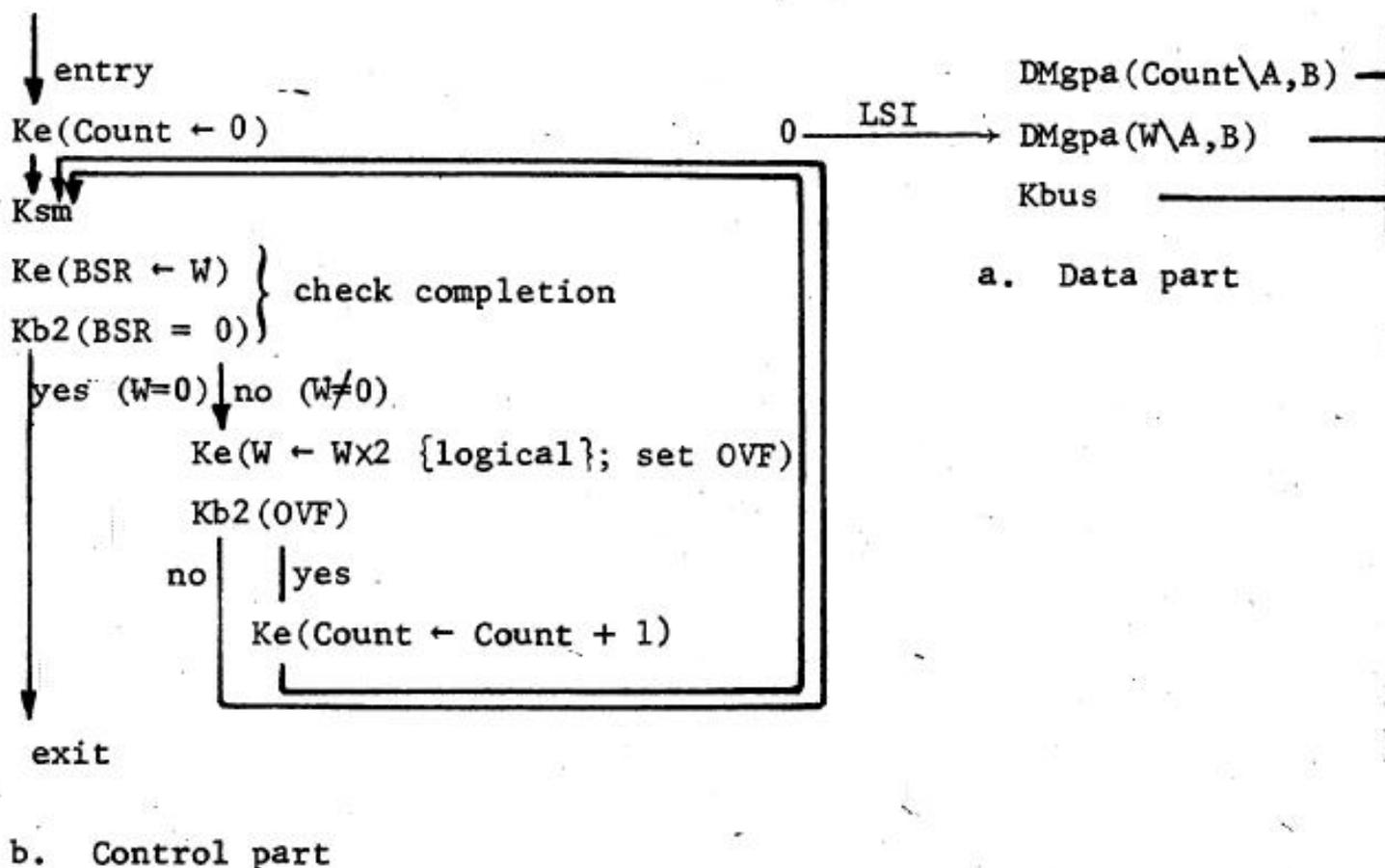


Fig. OC-2. RTM diagram for counting the number of ones in a word.

The first step of the algorithm resets the Count register. The next two steps are used to check whether the input word is a zero. When  $W$  is 0, there are no more 1's in the word, and the algorithm terminates (exits). If  $W$  is not 0, the main step of the control part shifts out (destroying) the most significant bit into the OVERFLOW flag\OVF for counting. OVF is then checked and counted if 1 (i.e.,  $\text{Count} \leftarrow \text{Count} + 1$ ). In shifting the result left (i.e., multiplying by 2) a 0 is input to the least significant bit  $A\langle 0 \rangle$ . This shift can be represented in a number of ways:  $(\text{OVF} \ll A \ll A \ll 0)$  or  $(A \langle 15:0 \rangle \leftarrow A \langle 14:0 \rangle \ll 0; \text{OVF} \leftarrow A \langle 15 \rangle)$  - signifying the actual bit transformation;  $(A \leftarrow A \times 2 \{\text{logical}\}; \text{Set OVF})$  - signifying a shift takes place, but modified by the name {logical} meaning a 0 is shifted into the least significant bit; or  $(\text{LSI} := 0; A \leftarrow A \times 2; \text{Set OVF})$  - signifying a shift where the actual shift input bit is specified (i.e., by the left shift\LSI input). The symbols "x2" here mean shifting left one bit position, not multiplying by 2. Multiplying is designated by "\*" in this book. We choose the form  $(A \leftarrow A \times 2 (\text{logical}); \text{Set OVF})$  for this example.

It should be noted that the implementation has a specific cost (consisting mainly of the 2 DMgpa's) and takes a certain time (approximately  $1 + 8 \times 3 + 7 \frac{1}{2} \times 2$  evoke times, or  $40 \text{ evokes} \times .83 \sim \text{s/evoke}$  or 35.2

microseconds for an average of eight l's randomly placed in a word).

## SOLUTION 2

A slight reordering of the steps of the algorithm can remove the step which accesses  $W$  to test if it is zero. Carrying out the re-ordering gives the design

82

[previous](#) | [contents](#) | [next](#)

input word				output word				
0101	1111	0000	0110	0000	0000	0000	0100	(8)
0000	0000	0000	0000	0000	0000	0000	0000	(0)
1111	1111	1111	1101	0000	0000	0000	1111	(15)

## SOLUTION 1

To simplify the design of the RTM system, an abstract algorithm can be determined. Since flowcharts are used in this book to express behavior, a flowchart of the algorithm is given in Fig. OC-1. The algorithm explains how such a system might operate, but need not in general be constrained by a physical / structure. However, since the purpose of this book is to discuss RTM's, assumptions made about operations on RTM systems will often affect how we write a flowchart.

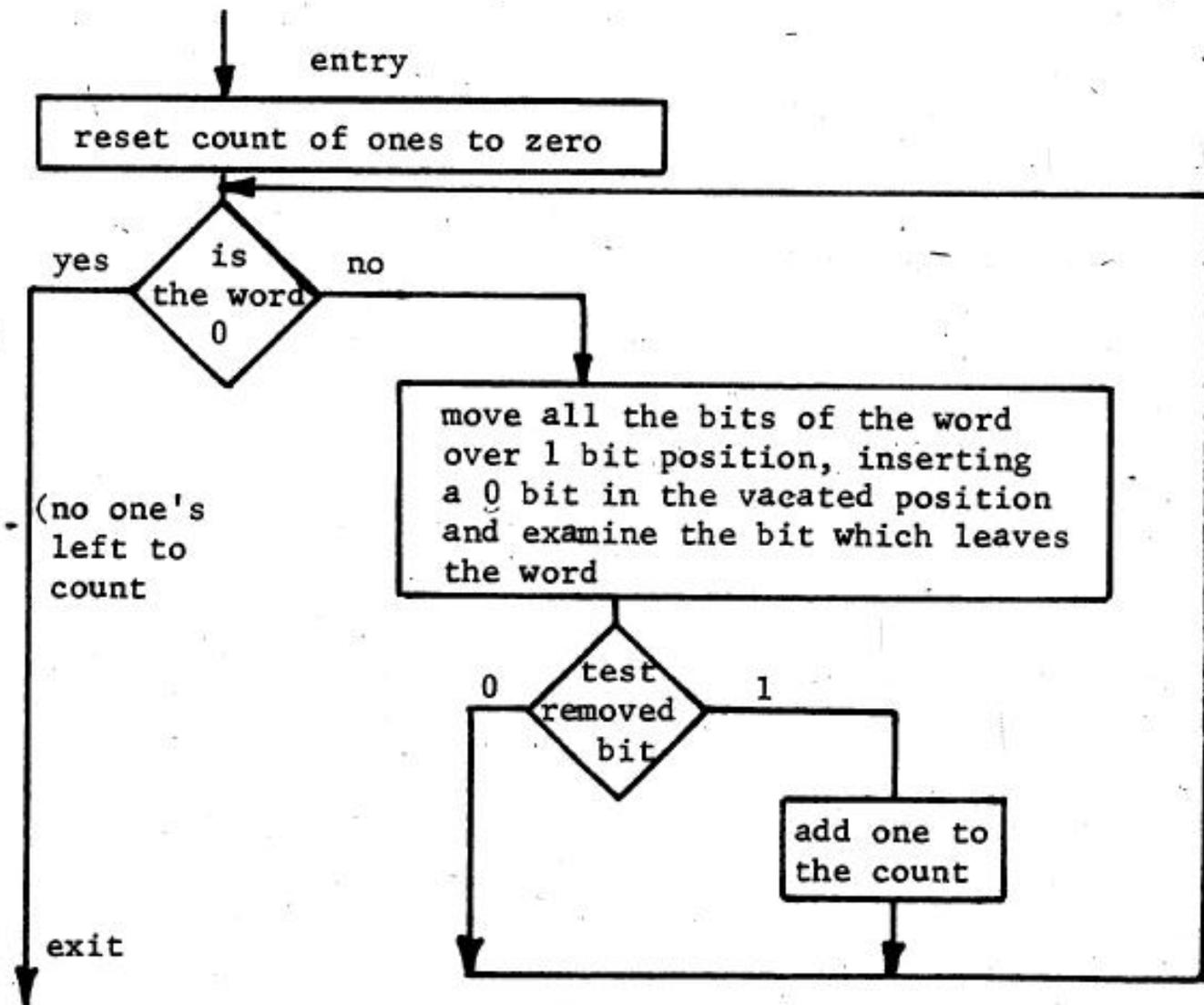


Fig. OC-1. Algorithm flowchart to count the number of one bits in a binary word.

Figure OC-2 gives a diagram of an RTM system which carries out the ones counting computation. Although only two registers are needed in the computation, two DMgpa's are used because of the asymmetric capabilities of the A and B registers of a DMgpa. At the entry to the control part, one DMgpa holds the word  $W$ , which is assigned to the A register by using the alias convention (i.e.,  $W \setminus A$  means that this register can be identified by either name). At the exit of the control part, register Count (the other DMgpa A register renamed) holds the number of 1's in  $W$  at entry. The subprocess destroys the data in  $W$ .

## CHAPTER 3 ELEMENTARY DESIGN EXAMPLES

The purpose of this chapter is to introduce the reader to the RTM design process by posing and solving a number of elementary RT-level design problems. The problems use the modules in Chapter 2 in a straight-forward fashion to hold Booleans (bits), and unsigned, and 2's complement signed integer data. The main concepts that are introduced are data representation and defining and structuring problems for solution; also included are some uses of subroutines, clocks for timing, and the related problem of synchronizing two independent systems.

Most of the designs in this chapter, being small, are actually subprocesses that occur in larger programs. For example, the first problem, counting the number of one bits in a word, would hardly ever be done for its own sake. Not until Chapter 5 do the problems become sizable enough to constitute entire information processing tasks. We do pose a couple of the problems in this chapter to have external input and output, thus being complete systems. But these are somewhat contrived, in order to give the flavor of total systems.

However, one should not view the designs of the present chapter simply as finger exercises. As we noted in Chapter 2, one builds up subprocesses out of the basic RTM's that are used again and again in larger systems. Some of the more important of these we have called Extended RTM's\ERTM's, and this chapter includes several of them. These ERTM's package various complex forms of control, the most complex one being the K(for loop) which evokes the operation of a subsystem for a variable number of iterations. Thus, these elementary examples contain important lessons about how to build complex control. They will be useful throughout the book both as actual modules (i.e., as ERTM's) and as models for larger systems.

The design examples are presented in a uniform format. A list of keywords appear first and sometimes a brief introduction is given. The problem is then stated, design considerations (possibilities) posed, and solutions given. In most examples, additional problems (exercises) are either posed or posed and solved.

### COUNTING THE NUMBER OF ONE BITS IN A WORD

**KEYWORDS:** Counting, parity, cost-performance tradeoff, storing data within a flowchart, termination, testing

The operation of counting the number of 1's in a word is usually needed when the word is considered to represent independent conditions (i.e., the word is a 16-bit Boolean vector). The 16 individual bits might represent the indices of stations requesting attention, hence a one's bit count would correspond to the total number of stations requesting attention at one time.

### PROBLEM STATEMENT

Design a system which will count the number of one bits in a 16-bit word (sometimes called tallying in a computer).

## DESIGN CONSIDERATIONS

The word is held in a register of a DMgpa. The word (data) may be destroyed in the calculation. The resulting count is held in a DMgpa register. For example, the following illustrates the relationship between input and output.

1. Stop error - a one when the system *is* no longer evoking operations. Invalid when using K(PCS).
2. DS error - a one when Test is evoked AND there is another operation in progress.
3. Alarm - a one when either Stop error or DS error is one.
4. Auto Run - a control activate output which occurs after using a power clear cycle. If this output is connected to the Run input, then the system will start automatically (Start will be evoked) on power up, or after the PC switch is depressed.

The additional inputs of Kbusc that Kbus doesn't have are:

1. Test - an evoke input used to determine whether the system already is active (see DS error output above).
2. Run - a control input signal to cause a system start. Any negative pulse of greater than 50 ns will start the system.
3. Auto restart - a high to low transition at this input will cause a power clear pulse to be generated. If Auto-Run is connected to Run this input clears the system and restarts it. This input can also come from the error flags.
4. Ev-con - if this pin is connected to the activate input of any Kevoke in an RTM system, then the system will halt when it gets to that module. At this point the user can switch the system to manual mode and single step for debugging purposes. The single step switch resets the system, so switching back to automatic will then allow the system to continue.

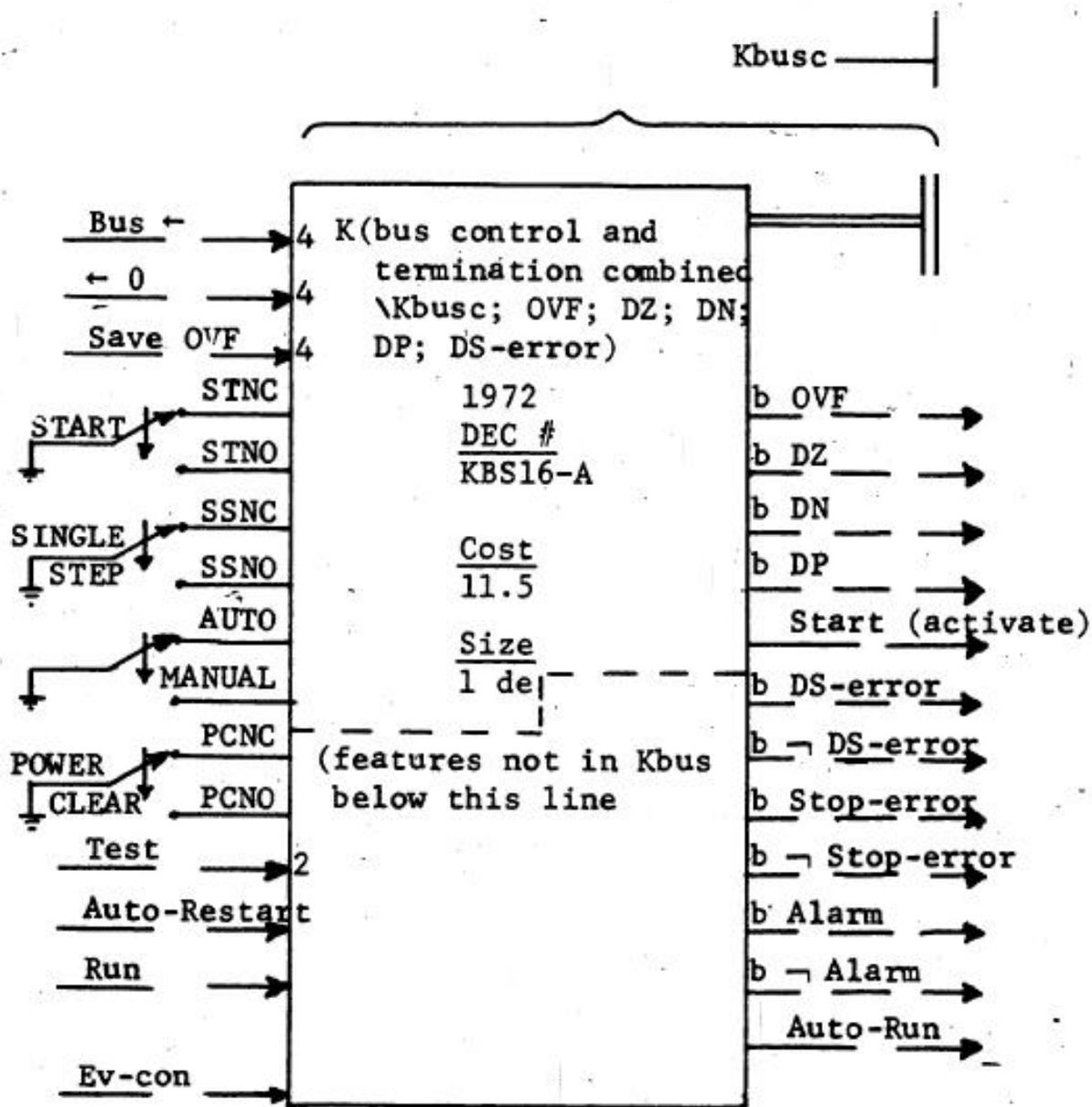


Fig. 26. Module diagram for K(Bus control and termination combined \Kbusc).

TRUTH TABLES

TABLE OF ARITHMETIC OPERATIONS						TABLE OF LOGIC FUNCTIONS					
FUNCTION SELECT				OUTPUT FUNCTION		FUNCTION SELECT				OUTPUT FUNCTION	
S3	S2	S1	S0	LOW LEVELS ACTIVE	HIGH LEVELS ACTIVE	S3	S2	S1	S0	NEGATIVE LOGIC	POSITIVE LOGIC
L	L	L	L	$F = A \text{ minus } 1$	$F = A$	L	L	L	L	$F = \bar{A}$	$F = \bar{A}$
L	L	L	H	$F = AB \text{ minus } 1$	$F = A+B$	L	L	L	H	$F = \overline{AB}$	$F = \overline{A+B}$
L	L	H	L	$F = A\bar{B} \text{ minus } 1$	$F = A+\bar{B}$	L	L	H	L	$F = \overline{A+B}$	$F = \bar{A}\bar{B}$
L	L	H	H	$F = \text{minus } 1 \text{ (2's complement)}$	$F = \text{minus } 1 \text{ (2's complement)}$	L	L	H	H	$F = \text{Logical } 1$	$F = \text{Logical } 0$
L	H	L	L	$F = A \text{ plus } (A+\bar{B})$	$F = A \text{ plus } A\bar{B}$	L	H	L	L	$F = \overline{A+B}$	$F = \bar{A}\bar{B}$
L	H	L	H	$F = AB \text{ plus } (A+\bar{B})$	$F = (A+B) \text{ plus } A\bar{B}$	L	H	L	H	$F = \bar{B}$	$F = \bar{B}$
L	H	H	L	$F = A \text{ minus } B \text{ minus } 1$	$F = A \text{ minus } B \text{ minus } 1$	L	H	H	L	$F = \overline{A+B}$	$F = A + B$
L	H	H	H	$F = A+\bar{B}$	$F = A\bar{B} \text{ minus } 1$	L	H	H	H	$F = A+\bar{B}$	$F = A\bar{B}$
H	L	L	L	$F = A \text{ plus } (A+B)$	$F = A \text{ plus } AB$	H	L	L	L	$F = AB$	$F = \overline{A+B}$
H	L	L	H	$F = A \text{ plus } B$	$F = A \text{ plus } B$	H	L	L	H	$F = A + B$	$F = \overline{A+B}$
H	L	H	L	$F = A\bar{B} \text{ plus } (A+B)$	$F = (A+\bar{B}) \text{ plus } AB$	H	L	H	L	$F = B$	$F = B$
H	L	H	H	$F = A+B$	$F = AB \text{ minus } 1$	H	L	H	H	$F = A+B$	$F = AB$
H	H	L	L	$F = A \text{ plus } A^\dagger$	$F = A \text{ plus } A^\dagger$	H	H	L	L	$F = \text{Logical } 0$	$F = \text{Logical } 1$
H	H	L	H	$F = AB \text{ plus } A$	$F = (A+B) \text{ plus } A$	H	H	L	H	$F = \bar{A}\bar{B}$	$F = A+\bar{B}$
H	H	H	L	$F = A\bar{B} \text{ plus } A$	$F = (A+\bar{B}) \text{ plus } A$	H	H	H	L	$F = AB$	$F = A+B$
H	H	H	H	$F = A$	$F = A \text{ minus } 1$	H	H	H	H	$F = A$	$F = A$

Fig. 25. Table of operations for S54181\N74181 (courtesy of Signetics).

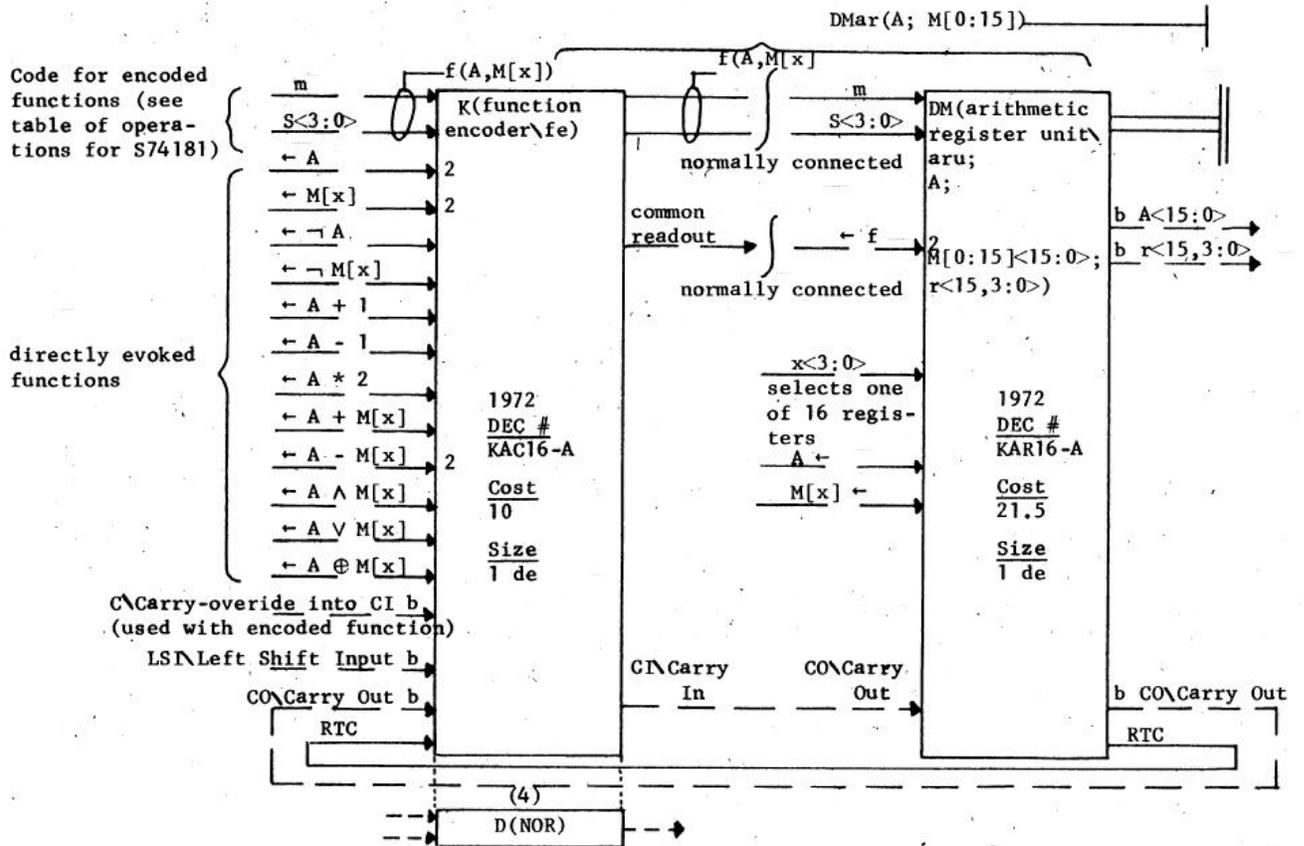


Fig. 24. Module diagram of DM(arithmetic and registers)\DMar.

available. One is a variation of the Dmgpa, and the other is a variation of the Kbus. Because of the importance of these modules we include a description of them here. However, because of their late arrival, none of the examples in the book utilizes these modules.

DM(arithmetic and registers)\DMar. Two modules, the DM(arithmetic register unit)\DMaru and the K(function encoder)\Kfe, taken together form a module which will be designated as the DM(arithmetic and registers)\DMar. Although the Dmaru can conceivably be used alone, the two modules will most likely be used together, hence they will be described together. Figure 24 describes the DMar, and should be consulted during the following description.

Briefly speaking, the DMar is much like a DMgpa with 16 addressable B registers, and the ability to receive encoded as well as direct evoke inputs for its functions. Actually, the DMar has 17 + 5/16, 16-bit registers available: A<15:0>, a 16 word scratchpad M[0:15]<15:0>, and 5 bits of the last result loaded into any of the positions of the scratchpad, designated r<15, 3:0>. Like the DMgpa, most operations are carried out on A. For operations on two registers, the second operand is held in the 16-word memory, M. The specific word of M is selected by a 4-bit input, X<3:0>, thus operations are actually on M[X]. Since all 16 combinations of X<3:0> are used, they may be addressed with either positive or negative logic. Thus, for example, they may be addressed using a Kevoke. Four NOR gates are provided on the Kfe to expand the inputs to X<3:0> if necessary.

DMar has essentially the same functions provided by DMgpa, with the exception that M[X] appears in place of B. There is no right shifting capability; however, an RTM subroutine can be written to do this. On the plus side, data can be written simultaneously into A and M[X] if desired.

DMar is implemented using a model S54181\N74181 Integrated Circuit which has the capability of implementing 16 arithmetic functions and 16 Boolean functions of the 2 input variables (in this case, A, and M[X]). The functions can be evoked in either of two ways: direct (e.g., <- A, <- M[X],..., <- A + M[X]); or encoded, in which the 5-bit code at M [ ] S<3:0> determines the operation. When used in the encoded mode, the code inputs must be present only during the desired operation and not at other times. Again, the NOR gates that are provided can be used to expand these inputs. The operations using this mode are given in Figure 25 for the logic case (m=0) and the arithmetic case (m=1). Whenever the code HHHHH (for all High) is used, the readout signal, <-f, on the DMaru must be separately evoked (see Figure 24). For all other functions (direct or encoded) <-f is automatically evoked via the Common readout signal from the Kfe. Whenever the encoded evoke is used, the C input must be used to specify the Carry In to the DMaru. For direct evokes, the Carry In is specified automatically, e.g. on <-A\*2 it takes the value of LSI, just as it does in the DMgpa.

K(bus control and termination combined)\Kbusc. Kbusc (see Figure 26) is a simplified version of the Kbus and the Bus terminator within a single (double height) board. Kbusc has no Bus Sense Register, although the result given in the most recent transfer on the Bus can be monitored via the following

signals: DZ\( $\text{result} = 0$ ); DN\( $\text{result} < 0$ ); DP\( $\text{result} > 0$ ); and OVF\OVERFLOW. The evoke input signals: Bus <-, <- 0, and Set OVF are also available. The above signals and the four basic switch inputs: POWER CLEAR\PC, AUTO-MANUAL, SINGLE STEP and START are identical to those of Kbus.

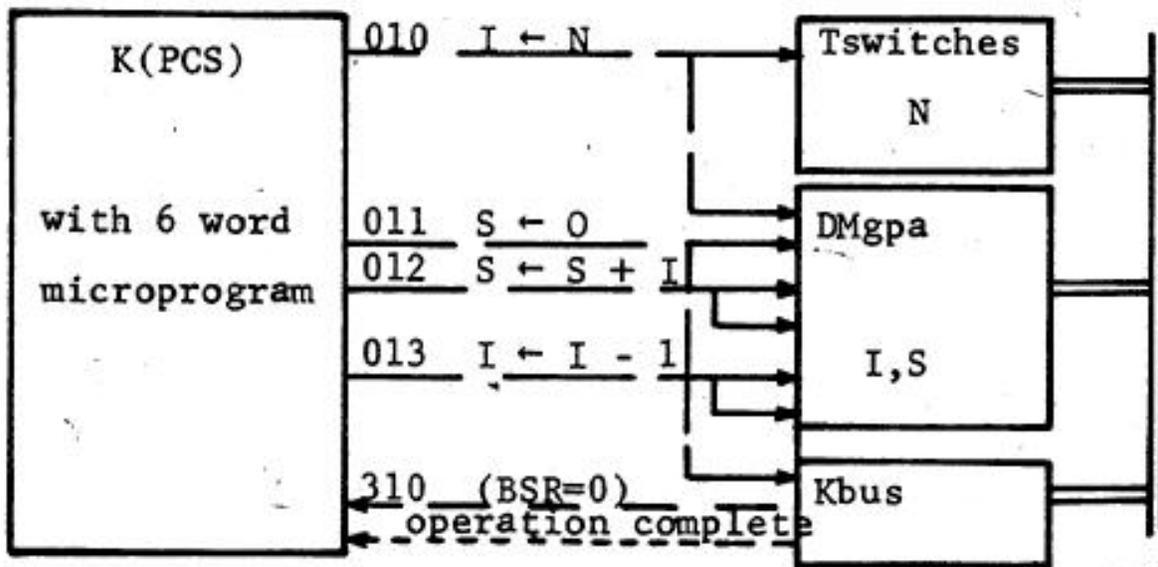
In addition, Kbusc has the following outputs that Kbus doesn't have:

75

[previous](#) | [contents](#) | [next](#)

<u>memory location</u>	<u>value (in octal)</u>	<u>action</u>
0	010	$I \leftarrow N$
1	011	$S \leftarrow 0$
2	012	$S \leftarrow S + I$
3	013	$I \leftarrow I - 1$
4, 5	310, 002	if ( $I \neq 0$ ) GOTO 2

a. Microprogram in K(PCS)



b. RTM diagram with K(PCS) for control part

Fig. 23. RTM system to compute sum-of-integers to N using K(PCS).

structure with particular instruction assignments for the data part components. Thus a given algorithm

can be specified in a number of ways using RTM systems:

1. hardwired
2. encoded in K(PCS) memory in terms of desired instructions
3. encoded in terms of PDP-16/M instructions (and structure)
4. programmed in terms of a hardwired computer
5. programmed in terms of a computer implemented using a K(PCS)
6. programmed in terms of a computer implemented using a PDP-16/M

TWO NEW MODULES: DMCARITHMETIC AND REGISTERS)

AND KCBUS CONTROC AND TERMINATION COMBINED)

Just before publication of this book, two new PDP-16 modules became

In a similar way, one of 29 Boolean (bit) input conditions (three of the possible conditions are reserved) can be sensed using the switch input structure. This function is:

$$\text{Test} := (\text{Boolean}[0] \wedge (\text{branch-code}=0) \vee$$

$$\vdots$$

$$\text{Boolean}[28] \wedge (\text{branch-code}=28))$$

With this function the branch-code selects one of 29 Booleans to test. The final part of the structure is the K(interpreter) which controls the action

of the other parts and forces the system to take on the desired behavior (see Figure 22). Note the behavior is expressed using a flowchart with register transfer operations. It is not an RTM system because it is not implemented with RTM parts, although it could be built in this fashion. The behavior is very similar to that of a general purpose stored program computer in that the interpreter picks up instructions sequentially (fetching), examines them (decoding), and then executes them. The execution process consists of taking one of four alternatives, depending on the instruction code. These alternatives correspond to evoke, conditional branch, subroutine call, and subroutine return, respectively. It is implicit in the execution of the evoke instruction that the ((interpreter) waits for a Bus DONE signal before proceeding.

## APPLICATION OF THE MICROPROGRAMMED CONTROLLER

Figure 23 shows the microprogram which computes the sum of integers to N. The program begins in location 0. Here, each operation is assigned a location in the memory in sequence. The convention for interpretation is that each instruction is picked up sequentially from memory. The sequence is broken by the final branch instruction, which returns control back to location 2. Note that the serial merge corresponds to a branch to a particular location. To convert the above microprogram to a microprogram subroutine only requires a subroutine return instruction in location 6. Doing this implies that location 0 is the beginning of the subroutine, i.e., a subroutine call 0 instruction specifies the function as a subroutine.

By looking at the previous example and structure of the microprogrammed controller, the reader may have some idea of its intended use. Whereas the hardwired structure can provide for parallelism in the control structure behavior, the microprogrammed control operates in a completely sequential fashion; each step is at least one access to the memory. Although the K(PCS) gives up parallelism, the cost is considerably less than that of a conventional RTM control part for larger systems. Also, the user commits himself to a fixed memory structure with the microprogrammed controller, whereas the hardwired structure can be modified somewhat. In Chapter 4 a graph is given for the cost of the control part versus

the number of control, steps for hardwired and K(PCS) implementations. The cross-over point for K(PCS) is approximately 80 control steps.

Chapter 6 presents the PDP-16/M computer which uses K(PCS) in a fixed

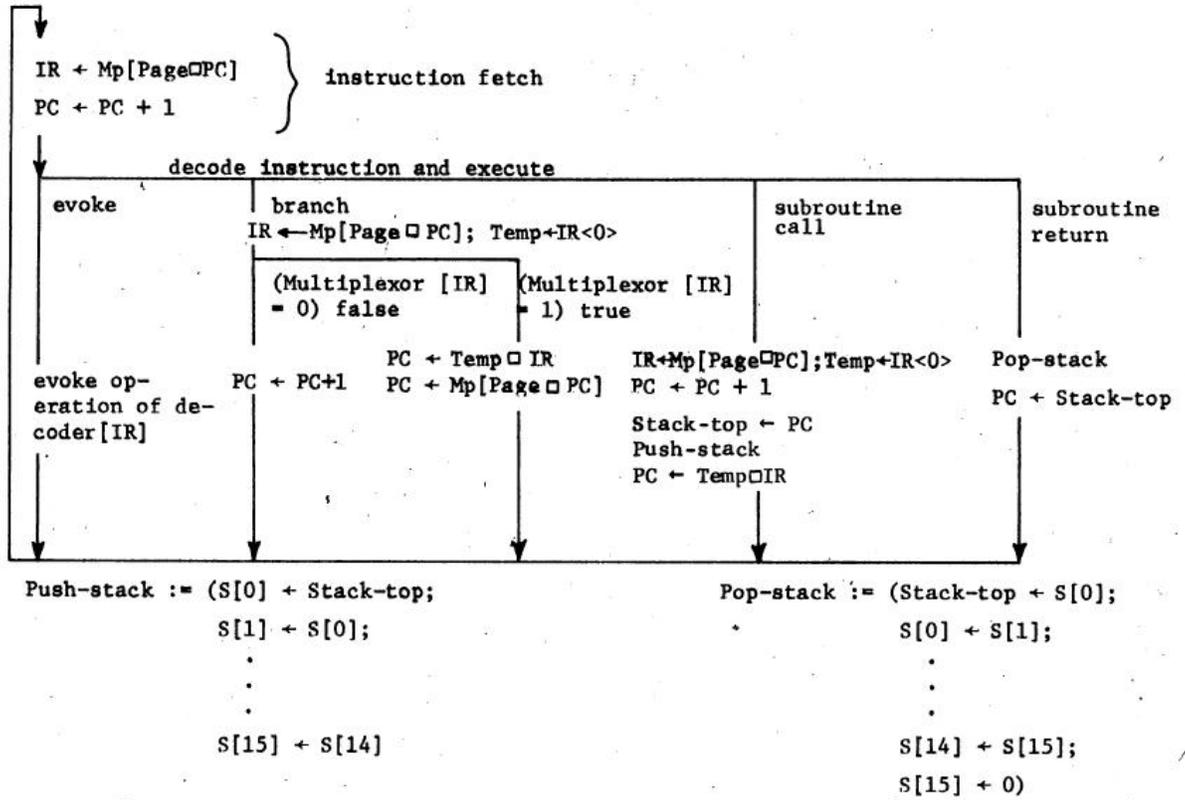


Fig. 22. Flowchart for the behavior of the K(PCS) (instruction interpretation process).

PC-Page - a bit that selects which of 512 words are to be used for the program - in effect, concatenated with PC.

IR<7:0>\Instruction Register - holds the current value of the instruction being performed

Mp[0:3][0:255]<7:0> - up to 4, 256-word, 8-bit/word memories which hold the instructions

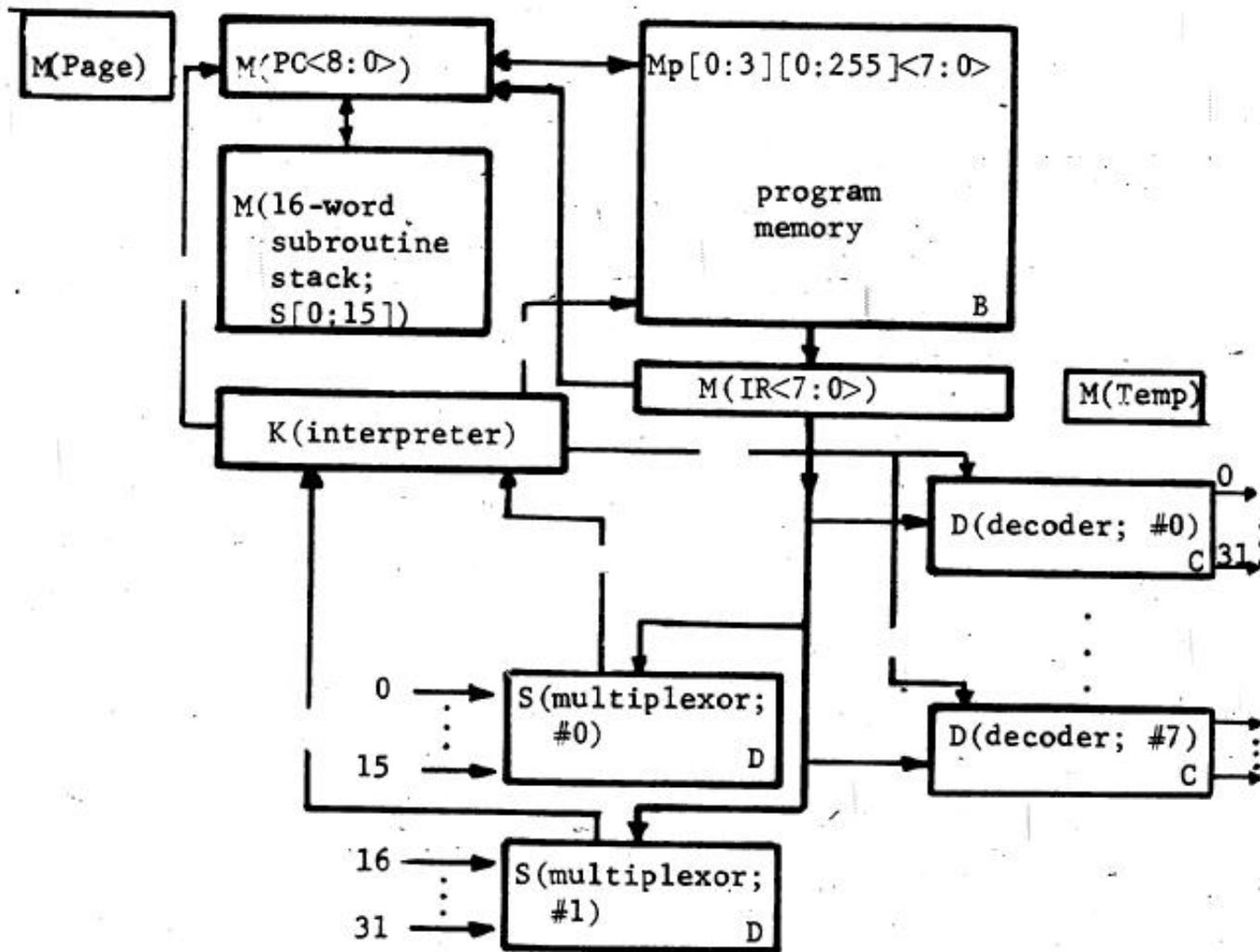


Fig. 21. PMS diagram of K(PCS).

The remaining parts of the structure are the various D(decoders) which evoke operations. The decoders "sense" the value of IR and create unique outputs. In effect, the decoders generate the following evoke functions:

**Evoke-output[0] :=(IR=0)**

**Evoke-output[255] := (IR = 255)**

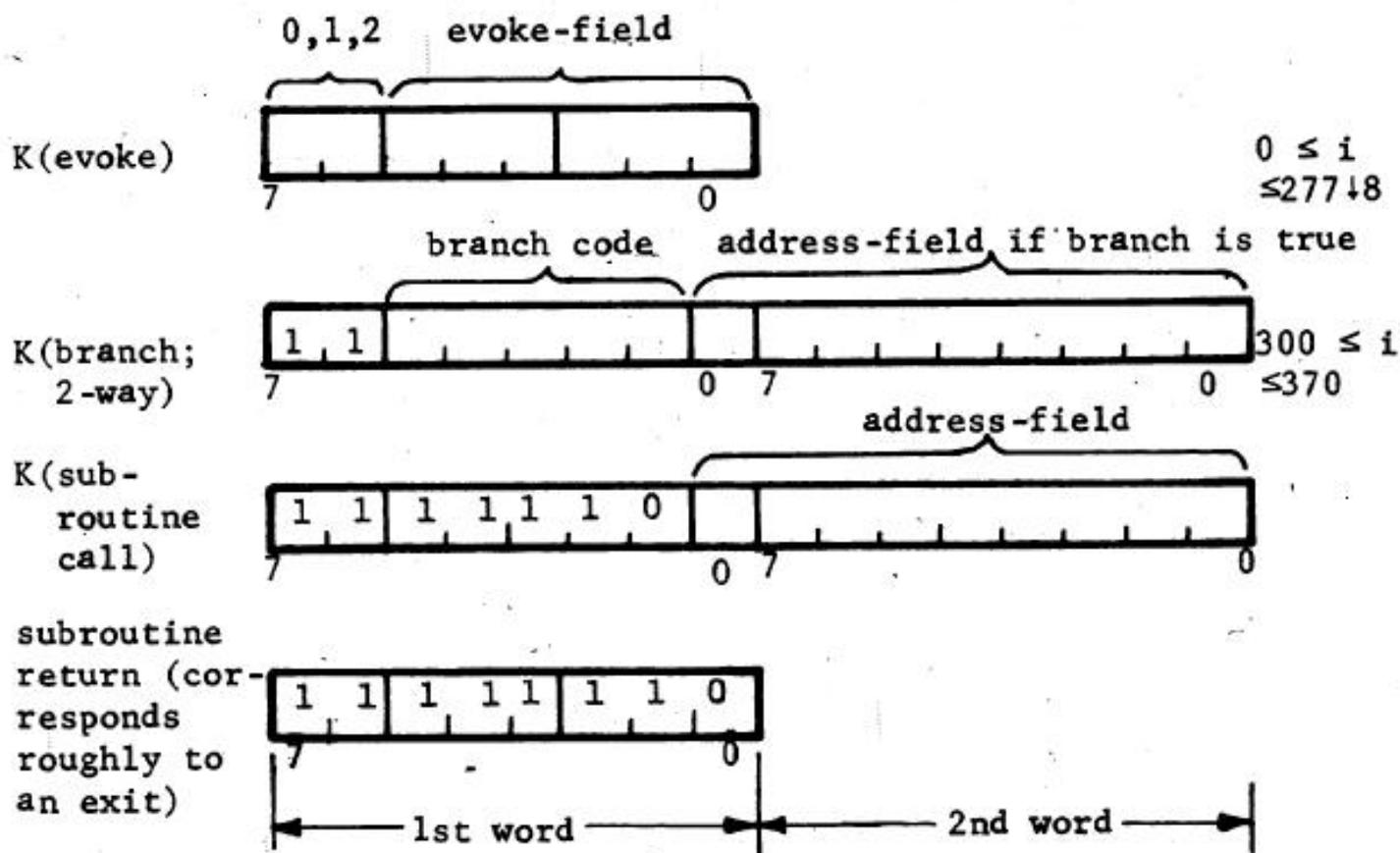


Fig. 20. Microinstruction formats for K(PCS).

## Subroutine Call and Subroutine Return

Two remaining instructions exist for calling a subroutine and specifying the return from a subroutine. A subroutine call has the effect of transferring the control (i.e., setting the Program Counter) to the address of the first instruction of a subroutine. At the completion of the subroutine an instruction, return, is given which returns control back to the instruction following the original subroutine call. Subroutine return is encoded  $376v8$ . Up to 16 subroutines can be called and the return addresses are placed in a subroutine stack which is attached to the Program Counter.

Note that there is an instruction, which corresponds to each of the hardwired operators: evoke, conditional (and unconditional) branch and subroutine call. Using these instruction formats, the instructions cannot be freely allocated to any memory locations, but instead, are fixed for sequential interpretation. There is no physical instruction for a ((serial merge); each occurrence corresponds to an address in the microprogram memory. Also, K(parallel merge) is impossible since there is but one

Program Counter for purely sequential control flow.

## THE STRUCTURE AND BEHAVIOR OF THE K(PCS)

Figures 21 and 22 show the structure and behavior of the K(PCS). Basically it has the following memories:

PC<8:0>\Program Counter - the instruction address pointer which selects the next instruction to be interpreted (selects 1 of 512 words)

T(general purpose interface)\Tgpi. The Tgpi provides for bidirectional transfer between external data and the RTM Bus. It contains a register,  $R_{<15:0>}$  to buffer outgoing data. The R register can be written by the Bus, i.e.  $R \leftarrow$ . The input data lines can be read directly onto the Bus, i.e.  $\leftarrow$  Input  $<15:0>$ .

T(input interface)\Tin. This is just the input part of the Tgpi.

T(output interface)\Tout. This is just the output part of the Tgpi.

## THE K(PROGRAMMED CONTROL SEQUENCER\PCS), A MICROPROGRAMMED CONTROLLER FOR A CENTRALIZED ENCODED CONTROL PART

The RTM control structure as realized with interconnected standard evoke, branch, merge, and subroutine call control modules has the interesting and useful characteristic that the abstract control flowchart is isomorphic to the physically wired control structure. Hence, specifying the behavior of the digital system specifies the control structure. This one-for-one mapping simplifies the system design and implementation processes. The control structure actions are specified by K-type components, and the notion of 'the next control step is specified by a wire which directs the next K module to be active. One alternative to this control structure, called microprogramming, encodes the control structure (i.e., flowchart) into a memory. An interpreting control unit reads instructions from the encoded memory and carries out the appropriate control steps on a one-at-a-time basis in much the same way as the hardwired structure. Each step, called a microinstruction, stored in the microprogrammed control's memory, corresponds roughly to a hardwired component. This microprogrammed control is implemented as a set of modules called the K(Programmed Control

Sequencer)\K(PCS).

### K(PCS) MICROINSTRUCTION ENCODING

The various instruction encodings for the K(PCS), and the K modules that they correspond to are given in Figure 20. An 8-bit/word memory array holds the microinstructions. The status of the K(PCS) is held in a 9-bit Program Counter\PC plus a 1-bit PC-page which points to the currently active control step (1 of 2110 words).

### Evoke Instructions

Each 8-bit memory cell can hold (encode) one of 192 ( $3 \times 64$ ) values which correspond to particular evoke functions (e.g.,  $A \leftarrow A+1$ ). Thus, a single cell corresponds identically to an instance of an evoke. Instructions 0 ~ 277v8 denote the coded range of evoke instructions.

## Branch Instructions

The branch instruction occupies two words and has two variable fields: a 5-bit branch field to select 1 of 32 input conditions and a 9-bit address field to select 1 of 512 memory addresses. If the input branch Boolean condition is true, the branch will be taken, and the control will proceed to the address specified in the branch instruction. If the condition is not true, then the instruction following the branch will be taken. Note that by fixing one branch condition at true, one can create an unconditional branch instruction. An unconditional branch instruction is often called a jump.

## T Modules

T(lights & switches). T(lights & switches) is an interface between an RTM system and the human user. It has a set of 16 toggle switches which may be set manually, and read by the Bus, i.e. <- switches<15:0>. In addition, there are 4 auxiliary toggle switches. (S1, S2, S3, and S4). These switches are wired single-pole-double-throw to ground, and are expressly intended to provide the Bus control switches for Kbus. However, with suitable adaptations they can be used in K(manual evoke)'s and as Boolean conditions. This module also has 20 display lights. Sixteen of these can be read from the Bus, i.e., Lights<15:0> <-. Of the other four, two correspond to the Bus timing signals, DA and DR, and the other two, L1 and L2, can be used to display Boolean data. If the Monitor enable input is grounded, then all Bus transfers will appear on the lights.

T(serial interface). This module is used as an interface to a standard Teletype, model 33, 35, or 37, either ASR or KSR, operating as either half duplex or full duplex. There are two registers, Transmit \T and Receive \R, which buffer the output to, and input from the Teletype, respectively.

If it is desired to receive a character from the Teletype papertape reader, the Start R operation is evoked. When the character has been received by the R register, Rflag goes to a 1. Then the character can be read onto the Bus, and Rflag cleared by a single evoke operation (i.e. <- R; Rflag <- 0). If another character is written into the R register before it is read, the Overrun flag goes to a 1.

To send a character to the Teletype, one must first monitor the Tflag. If it is a 1, the Teletype is ready to receive a character. Then one can send the character and clear the Tflag with a single evoke-operation (i.e. T <-; Tflag <- 0).

Since Start R is an operation that does not require the Bus, it may be evoked in parallel with some other operation that does. If so, the Boolean DA disable input must be set to 0, as in the DMflag and the Mtr described previously.

T(analog-to-digital converter)\Tad. Tad converts an analog input into digital form for entry into the RTM system. It contains a register, a-d<9:0>, which contains the digital value which is proportional to the analog input. Since it takes time to do an a-d conversion, the control may be operated in either of two ways. In the first way, the user may evoke the operation <- a-d<9:0>, and the conversion will be made, then the result read onto the Bus, all the time holding up control flow. In the second way, a command to initiate a conversion may be given (Start a-d), and then after the a-d flag signals that the conversion is complete (by going to 1), the <- a-d<9:0> operation is evoked. In the latter mode, the Bus may be used for other purposes while the conversion is going on. The <- a-d<9:0> operation also clears the a-d flag. Since this is not yet a standard PDP-16 module, it needs some additional steering logic to make it compatible. Thus, the user may implement the DA generation scheme for the Start a-d signal as he sees fit. The alternatives are: (1) generate a DA whenever Start a-d is evoked; (2) alternative (1) coupled with

an optional DA disable input; (3) no DA generated by Start a-d, so it must always be evoked in parallel with some DA generating operation. Similarly, a DR generation circuit must accompany <-a-d. Chapter 7 shows standard formats for DA and DR signal generation.

T(digital-to-analog)\Tda. The Tda contains a register, d-a, which when loaded forms an analog voltage proportional to the value in the register. The output analog signal is continuous. Thus the only functional command for the module is **d-a <-**. Here, as in the case of Tad, it is up to the user to provide a DA **generation circuit**.

Bus, respectively. These two fields can map the bits of the R register onto the selected half of the bus in any configuration. All unmapped bits can be wired to fill either 0's or 1's. Either one or both of the write operations can be evoked simultaneously. The same applies for the read operations. However, one cannot both read and write the same Mtr simultaneously.

Each bit of the R register is available as a Boolean output. The D(decoder) is often used with the P register to decode its contents. By using the Boolean DA disable input (see DMflag explanation) a register transfer can be made to multiple Mtr's at the same time, e.g.  $A \leftarrow B \leftarrow C \leftarrow D$ , where A, B, and C are all Mtr's and all registers (A,B,C, and D) are connected to the same Bus. The effect is;  $A \leftarrow D$ ;  $B \leftarrow D$ ;  $C \leftarrow D$ . The same caveat applies for zeroing the DA disable input here as applied for the DMflag. In addition, special wiring, not specified in the PDP-16 handbook, must accompany the Mtr, whether the DA disable is used or not. (See the pin number in Table 5.)

M(byte register)\Mbyte. The M(byte register) is similar to the Mtr, except it only has one write operation,  $R \leftarrow$ , and it has 4 read operations,  $\text{Bus}\langle 3:0 \rangle \leftarrow R\langle f1 \rangle$ ,  $\text{Bus}\langle 7:0 \rangle \leftarrow R\langle f2 \rangle$ ,  $\text{Bus}\langle 11:0 \rangle \leftarrow R\langle f3 \rangle$ , and  $\text{Bus}\langle 15:0 \rangle \leftarrow R\langle f4 \rangle$ . There is no DA disable on the Mbyte.

M(array;1024 word). This M is a 1,024 word, random access, solid state memory, declared  $A[0:1023]\langle 15:0 \rangle$ . It allows (indirect) expressions of the form  $\leftarrow A[X]$  and  $A[X] \leftarrow$ , for reading and writing, respectively. In order to specify the location in memory, the address, X, is first loaded into the memory address register, MA, by the operation  $MA \leftarrow$ . In fact sometimes we shall use these latter forms for the operations rather than the former. In addition, we sometimes use  $\leftarrow M[MA]$  and  $M[MA] \leftarrow$  (where M simply stands for Memory), or we use e.g.  $\leftarrow \text{CHP}$  or  $\text{Unbinned-item-count} \leftarrow$ , where CHP and Unbinned-item-count are aliases for specific memory locations. These different forms reflect the different ways that one can think about memory access.

Once the memory address has been specified, a command to transfer data is given. The data is transferred in or out of the Data register which is the interface to the storage part of the memory. Thus the two commands are  $\leftarrow \text{Data}$  and  $\text{Data} \leftarrow$  for reading and writing, respectively, which correspond to  $\leftarrow A[MA]$  and  $A[MA] \leftarrow$ .

M(array; read only; 1024 word). This is a 1,024 word braided wire, read only memory. It is similar to the previous M in behavior, except it does not allow write operations. This module required special interface to the PDP-16 system (see the PDP- 16 handbook).

M(array; 256-word). This M is just like the M(array; 1024-word), except it has fewer words.

M(scratchpad; 16 words). This memory is an array of 16 temporary storage registers,  $S[0:15]\langle 15:0 \rangle$ , which are essentially considered independently. The individual words are selected by one of 15 operation inputs,  $S[i]$  for  $I=0,1,\dots,14$ , and these are accompanied by a read or a write input. If no operation input is

selected, S[15] is selected. Only one register can be used at a time.

book that they are given the special name "Extended RTM's" (specifically, "Extended K modules"). These modules will be derived as they occur naturally in the design examples. However, as an illustration, Figure 19 constitutes a K(wait until).

## D Modules

D(decoder; inputs:4). The decoder is a Data-operation only component. It functions as a 4 input, 10 output BCD decoder, where the outputs correspond to the binary encoded value of the 4 inputs, i.e.,  $Out[0]:=1=0 \dots Out[9]:=1=9$ . The decoder can be used with the M(transfer register) (see below) to provide a set of Boolean outputs. It occupies a single height, single length board. The decoder can be enabled with a logical low signal.

D(NOT), D(AND|OR|NAND|NOR), D(Exclusive-OR| Equivalence). These modules are merely gates that can be used as needed for building small switching circuits to augment the larger modules, e.g. to calculate complex Boolean functions. They also function as K(serial merge) and K(parallel merge).

## DM modules

DM(flag)\DMflag. A DMflag contains a one bit register, F, that corresponds to the Boolean variable in programming languages. Alternatively, the F register can be thought of as storing one bit of data. Its Boolean output can be used to condition Kb2's and Kb8's. The operations that can be performed on F are  $F \leftarrow 0$ ,  $F \leftarrow 1$ ,  $F \leftarrow \neg F$ , and  $F \leftarrow \text{data}$ , where data is a Boolean input to DMflag. F and  $\neg F$  are available as Boolean outputs.

Since the DMflag does not use the Bus to transfer data, it is sometimes convenient to use a single K(evoke) to evoke both a DMflag operation and some other operation that does use the Bus, in parallel. For these situations a Boolean DA disable input is provided, which when set to 0 (logical low) prevents the DMflag from giving out a redundant DA Bus timing signal. Thus a typical operation incorporating this, type of "single-Bus parallelism" might be  $Ke(A \leftarrow A + 1; F \leftarrow \neg F; DA \text{ disable on DMflag})$ .(4)

## M Modules

M(constants;4 word)\Mc4. Mc4 is a module that has four 16-bit read only memory constants,  $C[0:3] \langle 15:0 \rangle$  which the user can read (i.e.,  $\leftarrow C[i]$ , where  $i=0,1,2,3$ ). Each constant is defined by wiring within the module by the user.

M(transfer register)\Mtr. Mtr is a single register,  $R \langle 15:0 \rangle$ , that can be read and written. For the write operations,  $R \langle 15:8 \rangle \leftarrow$  and  $R \langle 7:0 \rangle \leftarrow$ , either the upper half of the Bus, or the lower half can be written directly into the corresponding half of the register. For the read operations,  $Bus \langle 15:8 \rangle \leftarrow R \langle f 1 \rangle$  and

Bus<7:0> <- R<f2>, Mtr is constructed so that either of two fields (f 1 and. f 2) specified by user wiring, can be read selectively back onto the upper half or lower half of the

-----

4. Because of internal races, in 1972 PDP-16's the Boolean DA disable input to a DMflag cannot be zeroed directly by the evoke-operation output of a K(evoke). Instead, a Boolean signal, or a direct source of logical low must be used.

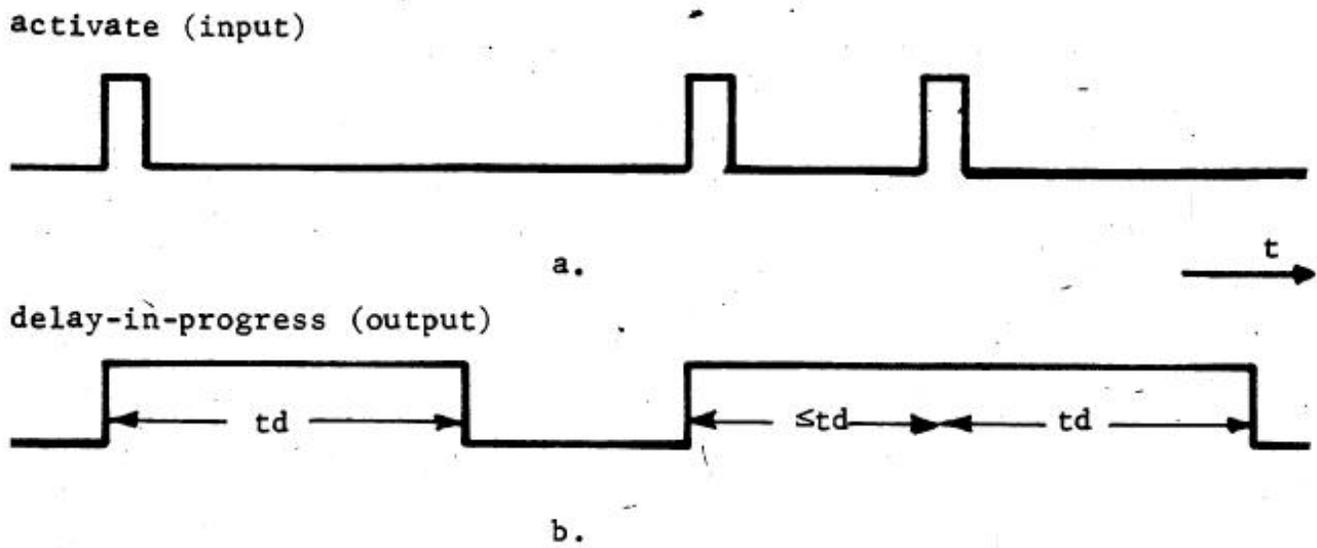


Fig. 18. Timing diagrams for K(delay; integrated).

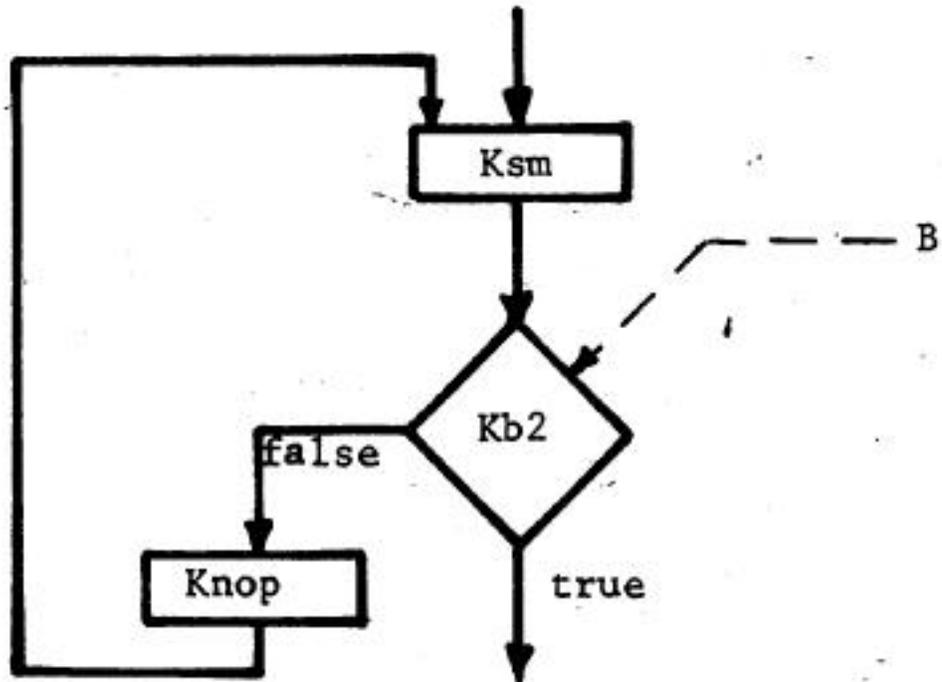


Fig. 19. RTM diagram for K(no-op)\Knop usage.

[previous](#) | [contents](#) | [next](#)

K(manual evoke)\Kme. Kme provides an interface between the RTM system and the human user. It allows a human to press a key which in turn creates a manually evoked activate-next control signal. This is not a PDP-16 module, but must be fabricated from other modules (see PDP-16 handbook or text, Chapter 7).

K(clock). This generates an activate-next signal at periodic intervals. The period is controlled by a potentiometer within the module. Care must be taken that the clock pulse is of the right width, so that it is properly synchronized with the DONE signal (see Chapter 7). Care must also be taken to avoid initiating a control path that is already active.

K(delay). When the delay activate input is applied, the K(delay) waits a period of time and then generates an activate-next output. The length of the delay period is controlled by a potentiometer within the module. The delay also has a Boolean output which indicates the delay is in progress. This module and the following module are not standard PDP-16's, so when physically implementing systems, care must be taken to insure compatibility.

K(delay; integrated). This module behaves in exactly the same way as the K(delay), but has an additional characteristic that permits it to be used in timing. All other modules are assumed to have only a single activate input signal before producing an output; this restriction is not present with the integrating delay. The function of the integrating delay is to delay from the most recent activate input signal. Thus, for example, for the activate input shown in Figure 18a, the delay-in-progress Boolean output is as shown in Figure 18b, where  $t_d$  is the characteristic delay of the module.

K(subroutine call)\Ksub. It is possible to have collections of K module sequences organized as subroutines so that they can be called from various places in the control portion of an RTM system. The module that calls these "hardwired" subroutines is similar to K(evoke), except that its call output is directed to a set of K modules, rather than to a DM module. When the hardwired subroutine is completed, it returns control to Ksub via the return link, and Ksub passes control on to the activate-next output. The use of RTM subroutines is illustrated later in greater detail.

K(no-operation \nop). This module is required in the PDP-16 system when a polling loop composed of a single K(branch) is being executed. That is, assuming the Boolean variable, B, is to change at some unknown time, and waiting must occur until that time, then the loop shown in Figure 19 is needed. The module serves no other function. The reason for this need is explained in Chapter 7.

K(macro). A K(macro) is not a single module, but effectively an open subroutine composed of other K modules, possibly in combination with other components. Whenever a K(macro) appears in an RTM control part, it indicates that the entire control part of the K(macro) is assumed to be inserted, in line, at that point (just like the open subroutine, or macro, of programming). This is in contrast to the K(subroutine call) which, whenever it appears in a control part, designates a call to some separate RTM

control part (just like the closed subroutine, or simply the subroutine of programming). Certain K(macro)'s i.e. K(arbiter), K(conditional execute), K(for-loop), K(manual-evoke), and ((wait-until), are used so often in this

Fig. 17. continued

Module Type	1972	Cost	Power (amps)	Size	time(nsec.)		
	DEC #				Read: Bus +	Write: + Bus	
T(lights and switches)	KBM16X				220	270	New version of KBM16 in PDP-16 handbook.
T(output interface)	DB16-B	7.5	.14	1 de		280	
T(serial interface)	DC16-A	32.5	.3 <sup>7</sup>	1 de	200	1500	

<sup>1</sup>w\width in number of slots: 1 or 2; h\height in slots:s\single\1 (2<sup>1</sup>/<sub>4</sub>" ) or d\double\2 (5<sup>3</sup>/<sub>16</sub>" );l\length: e\extend (8<sup>1</sup>/<sub>2</sub>" ) or -\regular (5<sup>1</sup>/<sub>2</sub>" )

<sup>2</sup>Total time for a register transaction: Kbus+Kevoke+read+write.

<sup>3</sup>Requires KBS16 + KTM16

<sup>4</sup>4 modules: PCS16-A Control: cost 15;  
PSC16-B ROM: cost ~ 22.5 per 256 steps;  
PCS16-C Decoder: cost ~ 5/32 instructions;  
PCS16-D: cost ~ 5/32 inputs

<sup>5</sup>Requires KAC16, KAR16, H851

<sup>6</sup>-15V: 37mA

<sup>7</sup>-15V: 100mA

<sup>8</sup>±15V: 25mA; -10.06V: 60mA;  
-15V: 60mA

<sup>9</sup>Unit cost figures used in examples. Actually proportional to selling price.

63

Fig. 17. continued

Module Type	1972	Cost	Power (amps)	Size	time(nsec.)		Comments
	DEC #				Read: Bus ←	Write: ← Bus	
<u>Constants:</u>							
M(constants; 4-word)	MR16-A	10	.16	1 de	200		
M(constants; 24-words)	MR16-C?	32	.4	1 de	1000		New version of MR16-C in PDP-16 handbook.
<u>Scratchpad:</u>							
M(scratchpad; 16-words)	MS16-C	20	0.49	2 de	270	600	
<u>Array organized:</u>							
M(array; 256-word)	MS16-D	50	.37 <sup>6</sup>	2 de	2200	1600	
M(array; 1024-word)	MR16-E	150	.37 <sup>6</sup>	2 de	2200	1600	
M(array; 1024-word; read only)	MS16-B	85	2.0	2 de	1000		Needs special interface to PDP-16 system.
<u>Transducers\T's</u>							
T(analog-to-digital)	A811	45			220		DEC module not in PDP-16 handbook.
T(digital-to-analog)	A618	35	.13 <sup>8</sup>	1 d-		270?	DEC module not in PDP-16 handbook.
T(general purpose interface)	DB16-A	10	.4	1 de	190	280	
T(input interface)	DB16-C	5	.26	1 de	190		

Fig. 17. Continued

Module type	1972 DEC #	Cost	Power (amps)	Size	time (nsec.)		Comments
					Read: Bus +	Write: + Bus	
<u>Data Operators (combinational)\</u>							
<u>D's</u>							
D(AND OR NAND NOR; inputs:2)	M112/10 etc. series	.15	.05	1 s			DEC modules not in PDP-16 handbook.
D(AND OR NAND NOR; inputs:4)		.2	.05	1 s			DEC modules not in PDP-16 handbook.
D(decoder; inputs:4)	M161	5.5	.12	1 s			DEC modules not in PDP-16 handbook.
D(Exclusive-OR Equivalence; inputs:2)	M112/10 etc. series	.4	.05	1 s			DEC modules not in PDP-16 handbook.
D(NOT)	M111/16	.1	.09	1 s			DEC modules not in PDP-16 handbook.
<u>Data-Memory Modules\DM's</u>							
DM(flags)	KFL16	4/3	.16	1 se	400	300	
DM(general purpose arithmetic)	3 <sup>5</sup>	31.5	.9	2 de	415	290	
<u>Memories\M's</u>							
<u>1 word:</u>							
M(byte)	MS16-B	11.5	.4	1 de	220	270	
M(transfer)	MS16-A	10	.4	1 de	220	270	

61

Fig. 17. continued

Module Type	1972 DEC #	Cost	Power (amps)	Size <sup>1</sup> (whl)	Time <sup>2</sup> (nsec)	Comments
<p><u>Extended:</u>                      K(arbiter)                      K(conditional-execute)                      K(for-loop)                      K(manual-evoke)                      K(wait-until)</p> <p><u>Other:</u>                      K(program controlled sequencer)</p>	PCS16 <sup>4</sup>	27 <sup>4</sup>		3 de		<p>modules, possibly in combination with some other components.</p> <p>Extended K modules are K(macros) that are used often in the book. Only the K(manual evoke) is in the PDP-16 handbook.</p> <p>Four modules that must be wired together.</p>

89

[previous](#) | [contents](#) | [next](#)



[previous](#) | [contents](#) | [next](#)

table of Figure 17 we give certain engineering data for the modules that are useful in this book, i.e., cost, size, power, and timing data. In addition, we give the 1972 PDP-16 model numbers for the modules. For the reader who is physically building PDP-16 systems, Table 5 at the end of the book gives pin numbers, fan-in, etc. for the modules. In the figures, the modules are given in the alphabetical order of Figure 17 for easy reference. However, in this section we describe the modules in an order convenient for exposition.

## Supplements to the Module Set

Most of the modules described in Figure 16 and 17 are manufactured by DEC as PDP-16's. A few modules are not actually PDP-16's, but are DEC modules that can be made compatible with the PDP-16 set by a small amount of switching circuit interfacing, so they are included as RTM's. A few other modules in the figures have not been built as of this writing, but they are planned additions to the PDP-16 set.

Two unique types of modules deserve special mention. The K(macro) modules are not really modules at all, but rather sets of specially interconnected modules that are treated as a unit. The other unique type of module is the Data Operator(combinational)\D. Since many of these are actually switching circuit level gates, they give one the flexibility to custom-design small combinational circuits where they are necessary to supplement the module set, e.g. for calculating complex Boolean conditions for Kb2 modules.

All of these special cases are pointed out by comments in Figure 1.7.

## K Modules

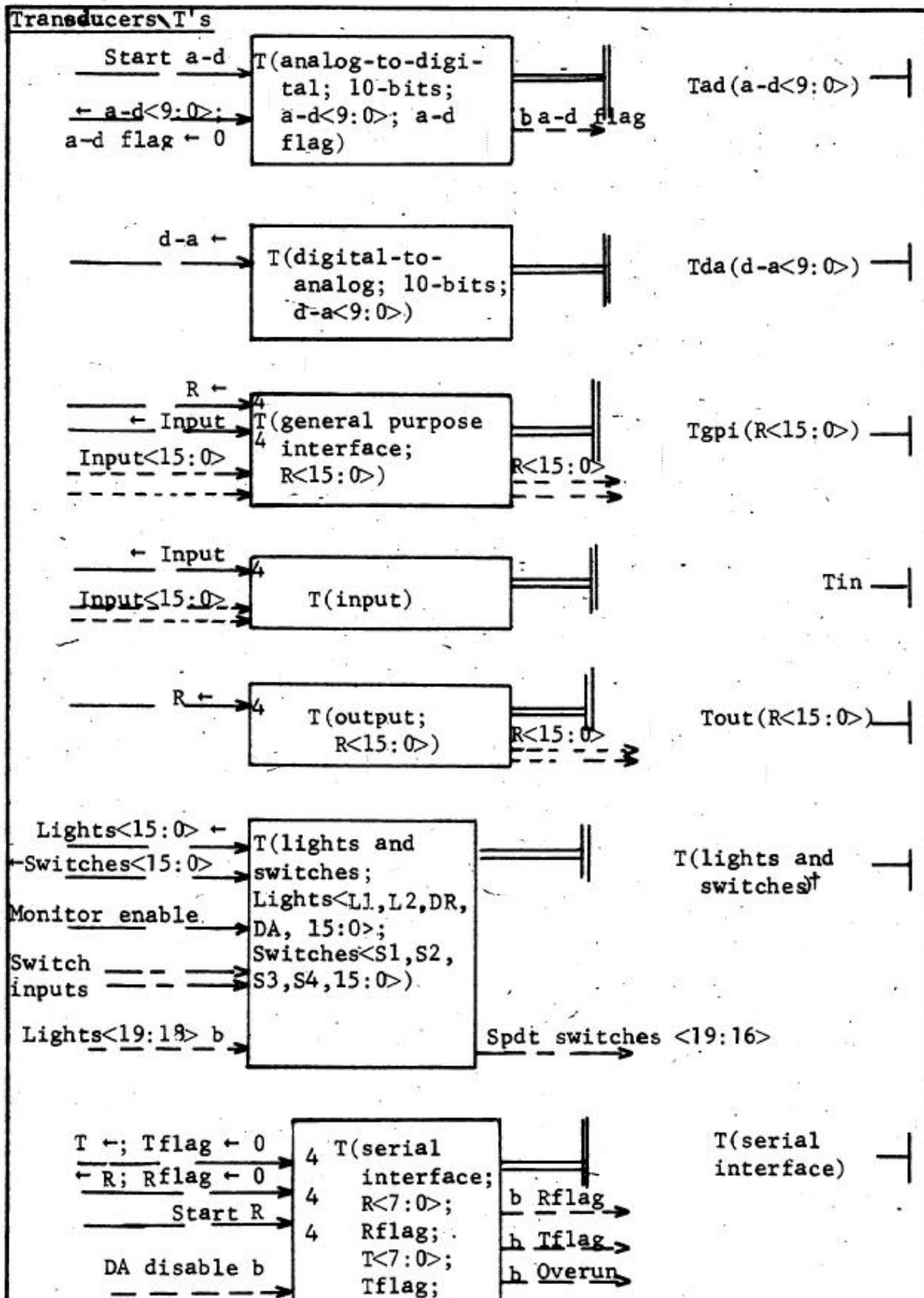
K(branch 8-way)\Kb8.- This module is similar to the two way branch described earlier, but it has three Boolean inputs and eight possible activate-next control / flow output ports. Each one of these eight ports corresponds to one of the eight possible combinations of the three Boolean input variables. The signal level conventions are identical to the two way branch.

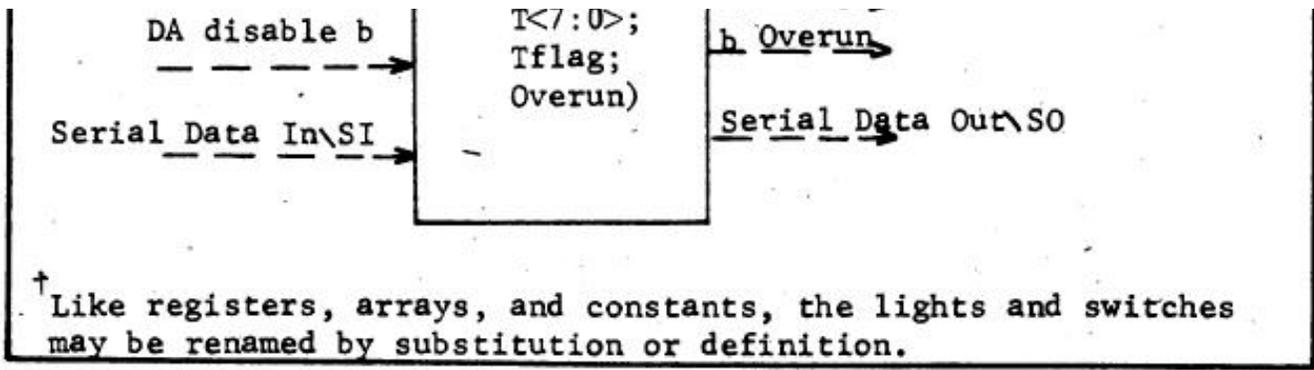
K(diverge). The K(diverge) is an implicit module in that it is merely the fan-out of a piece of wire. It is used whenever two or more parallel-operating control paths are to be activated. Parallel-operating RTM systems can be implemented by using several Busses, which may or may not be interfaced to allow passage of data back and forth. This subject will be taken up in greater detail in various examples, beginning in Chapter 4.

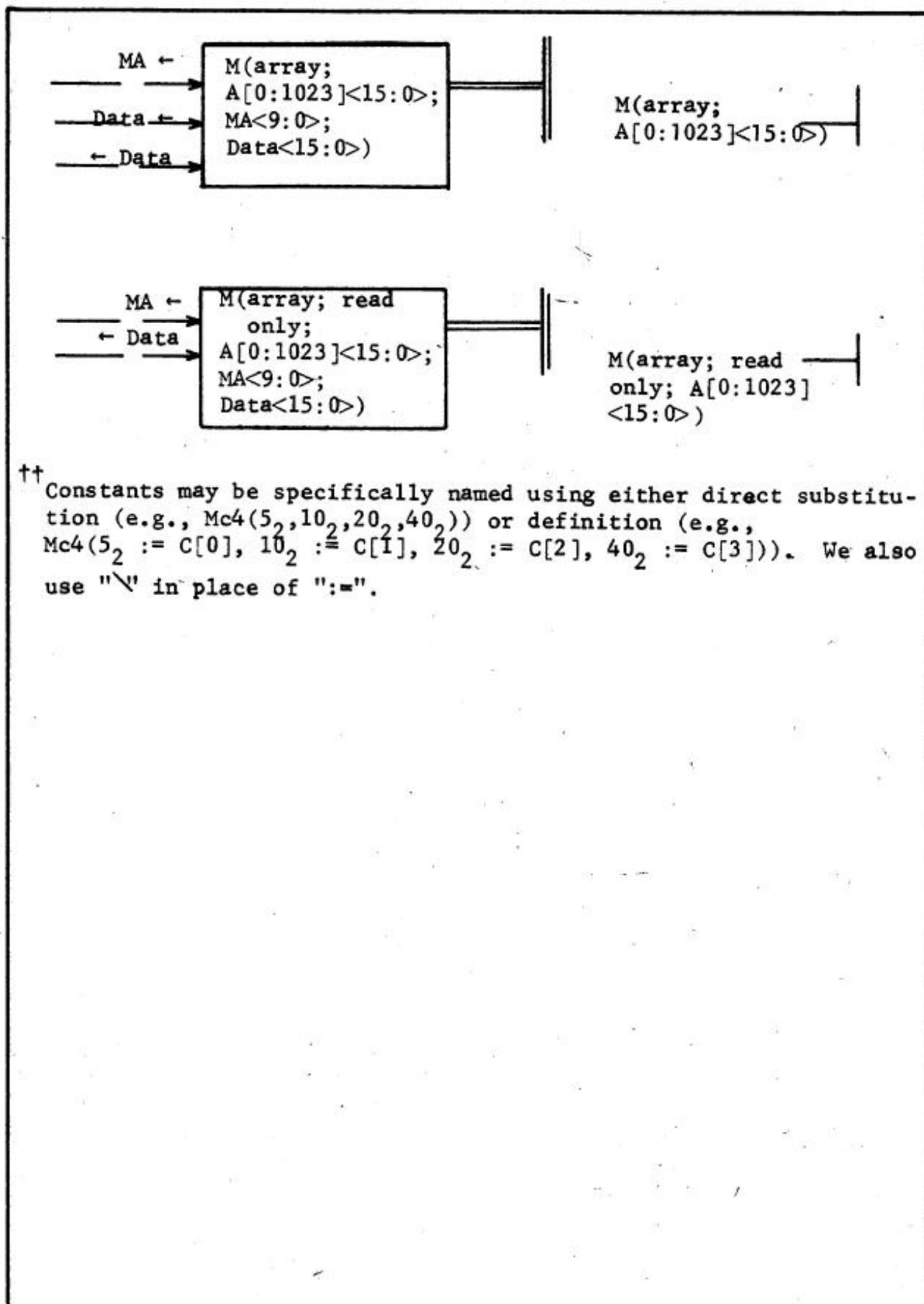
K(serial merge)\Ksm. This allows several control paths, only one of which may be active at one time, to come together and form a single control flow link. That is, if any one of the control flow inputs of a Ksm is activated, it will produce its active-next output and control will flow to the next K in the control path.

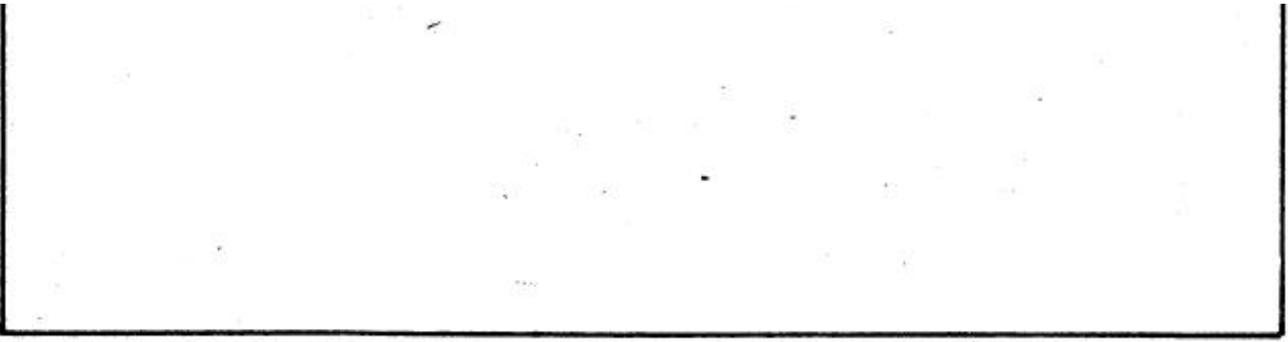
The serial merge is actually either a 2-input or 4-input negative logic OR gate. Several Ksm's are on a single height board.

K(parallel merge) \Kpm. This allows several control paths, all of which must be active simultaneously, to come together and form a single control flow link. That is, when all activate inputs have occurred, Kpm generates an activate-next signal and control will flow to the next K in the control path. The Kpm is actually a negative logic AND gate.

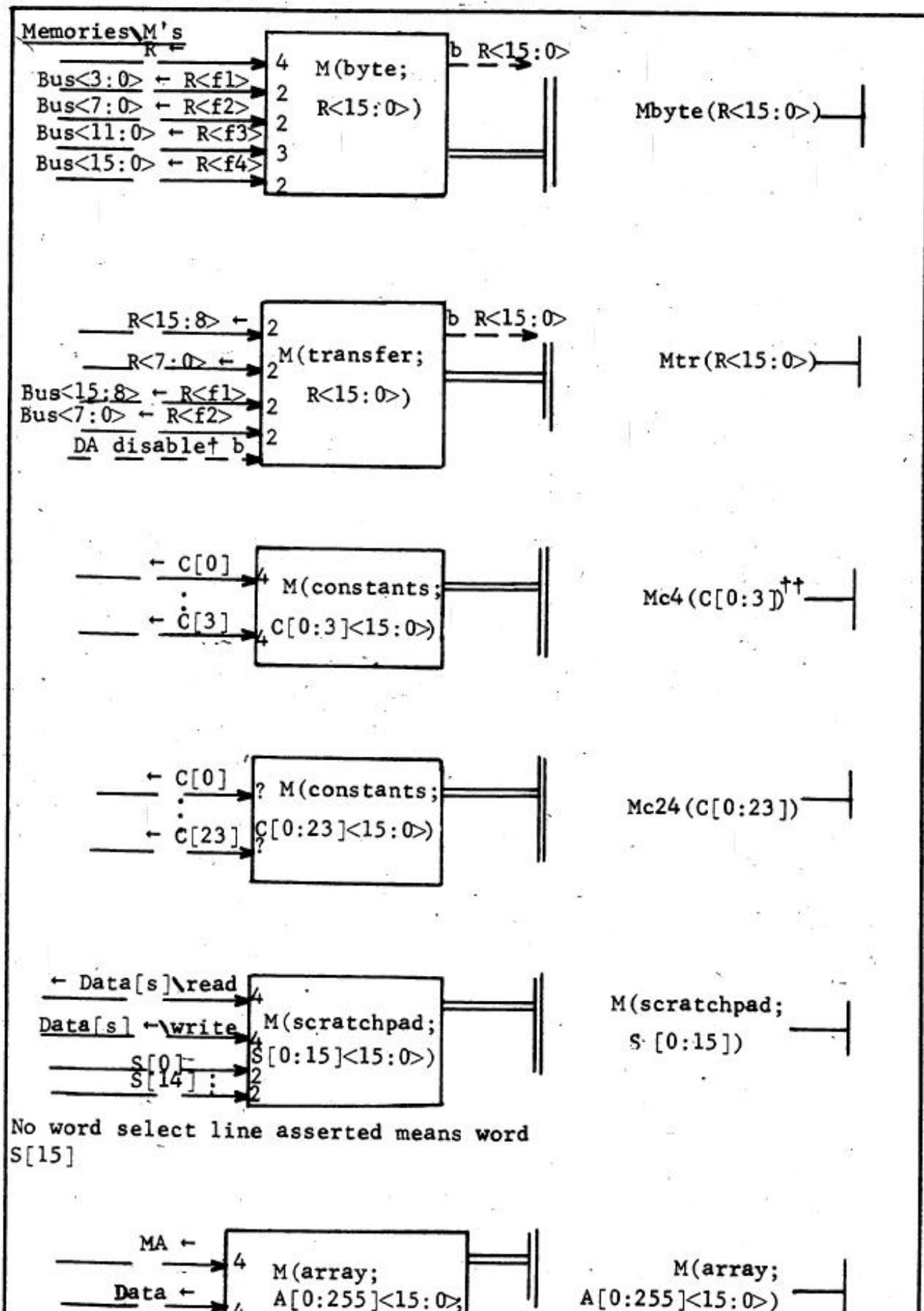


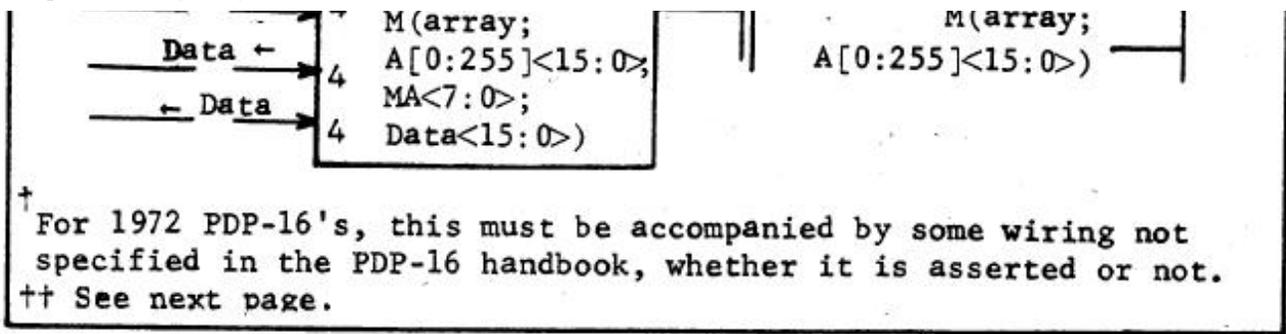


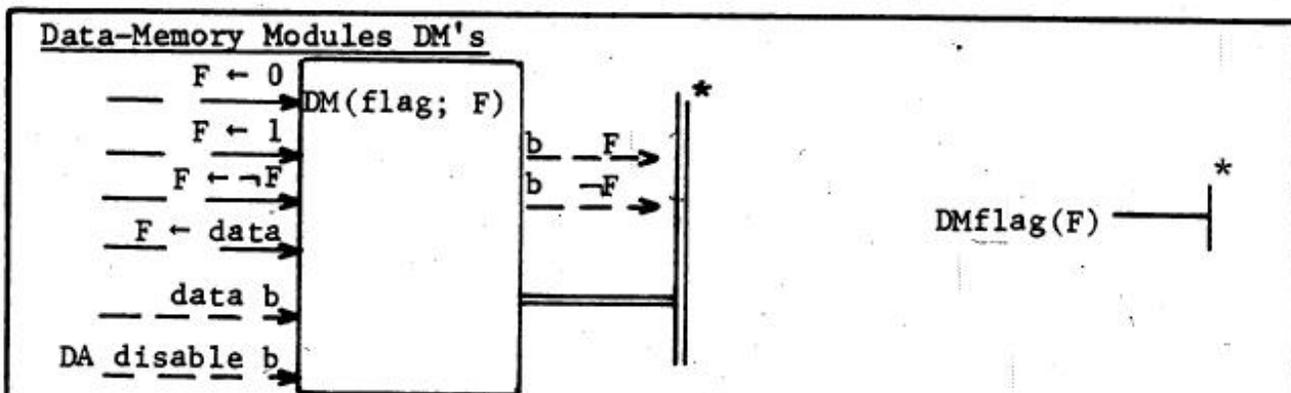




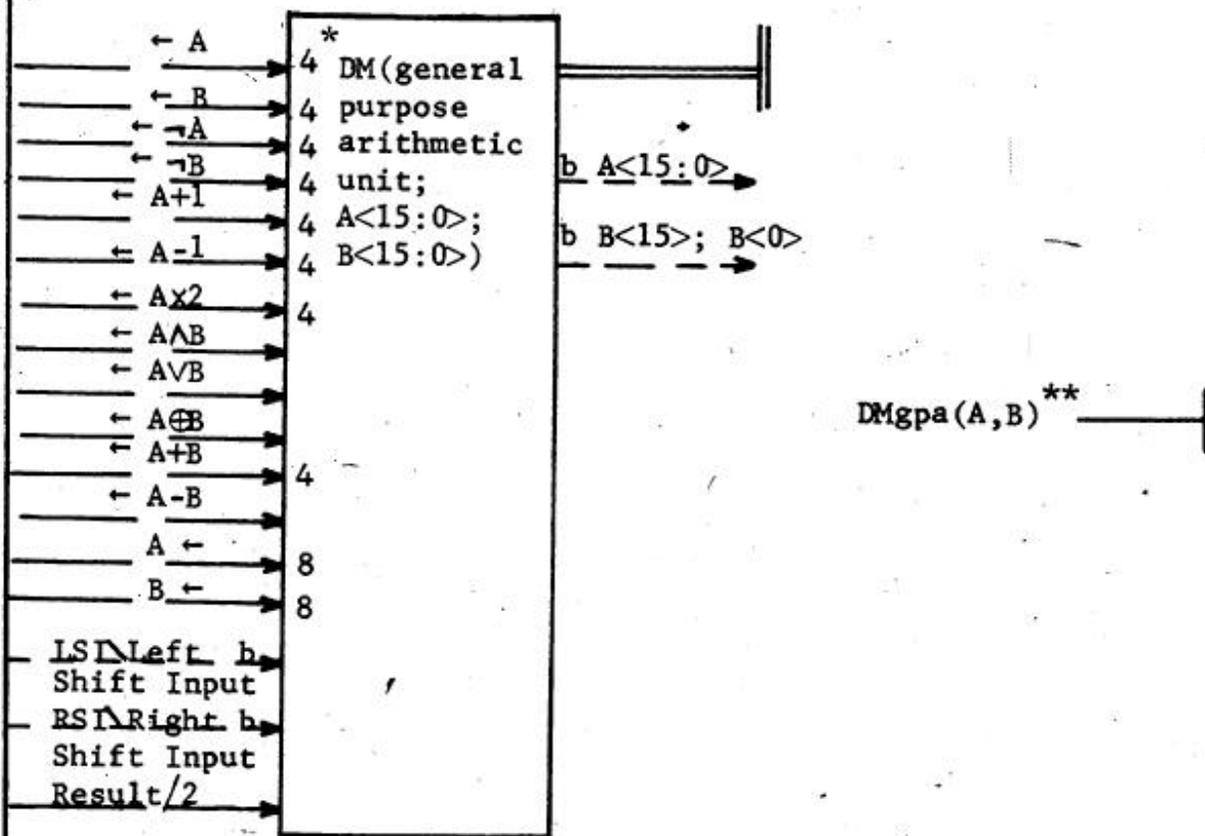
[previous](#) | [contents](#) | [next](#)







\* The DMflag connects to the Bus only to transmit the DA signal. Hence it is often drawn not connected to the Bus.

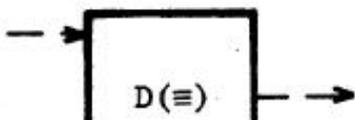
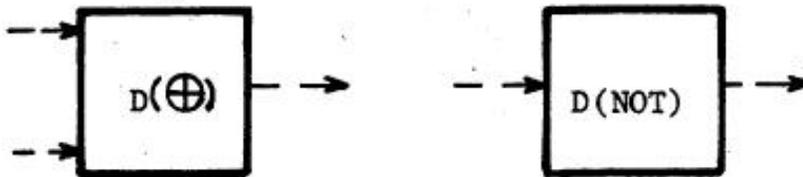
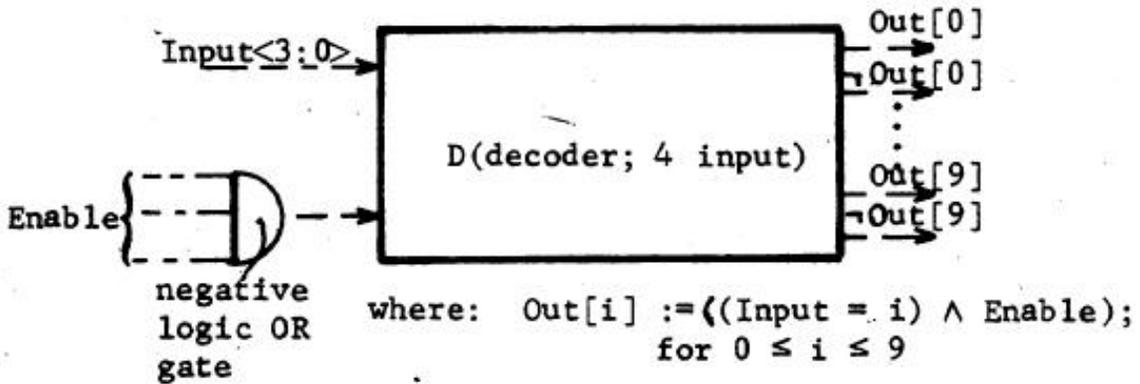
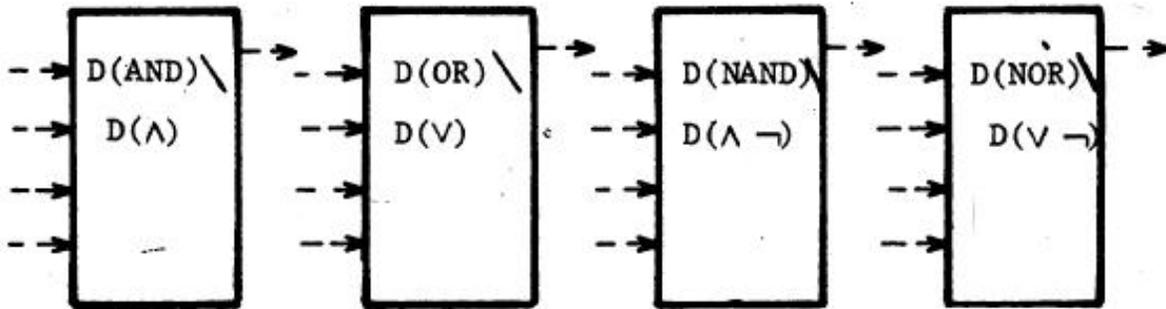
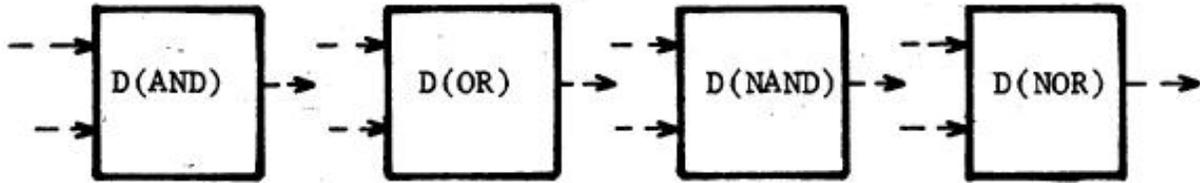


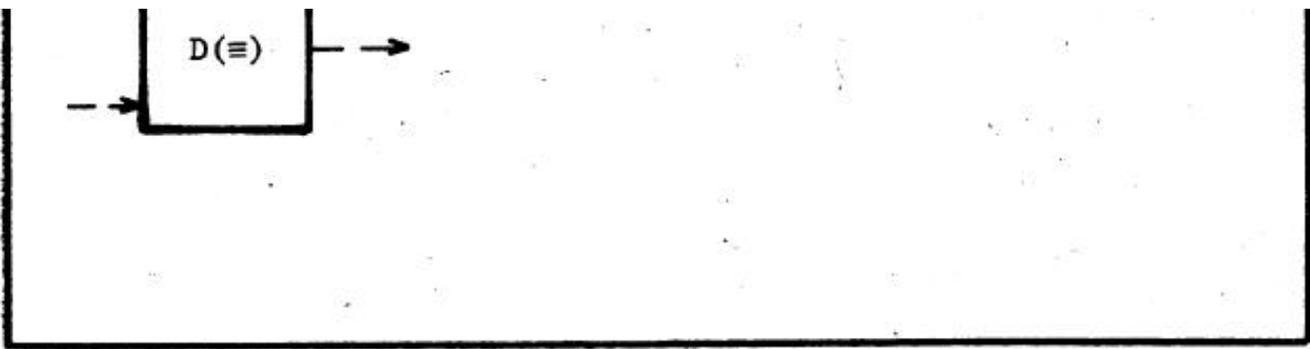
\* These numbers indicate the number of module inputs available for evoking the shown operation. The absence of a number denotes that there is one input. If more inputs are desired for 1972 PDP-16's, D(AND)'s must be used, since the operation evoke signals are negative logic.

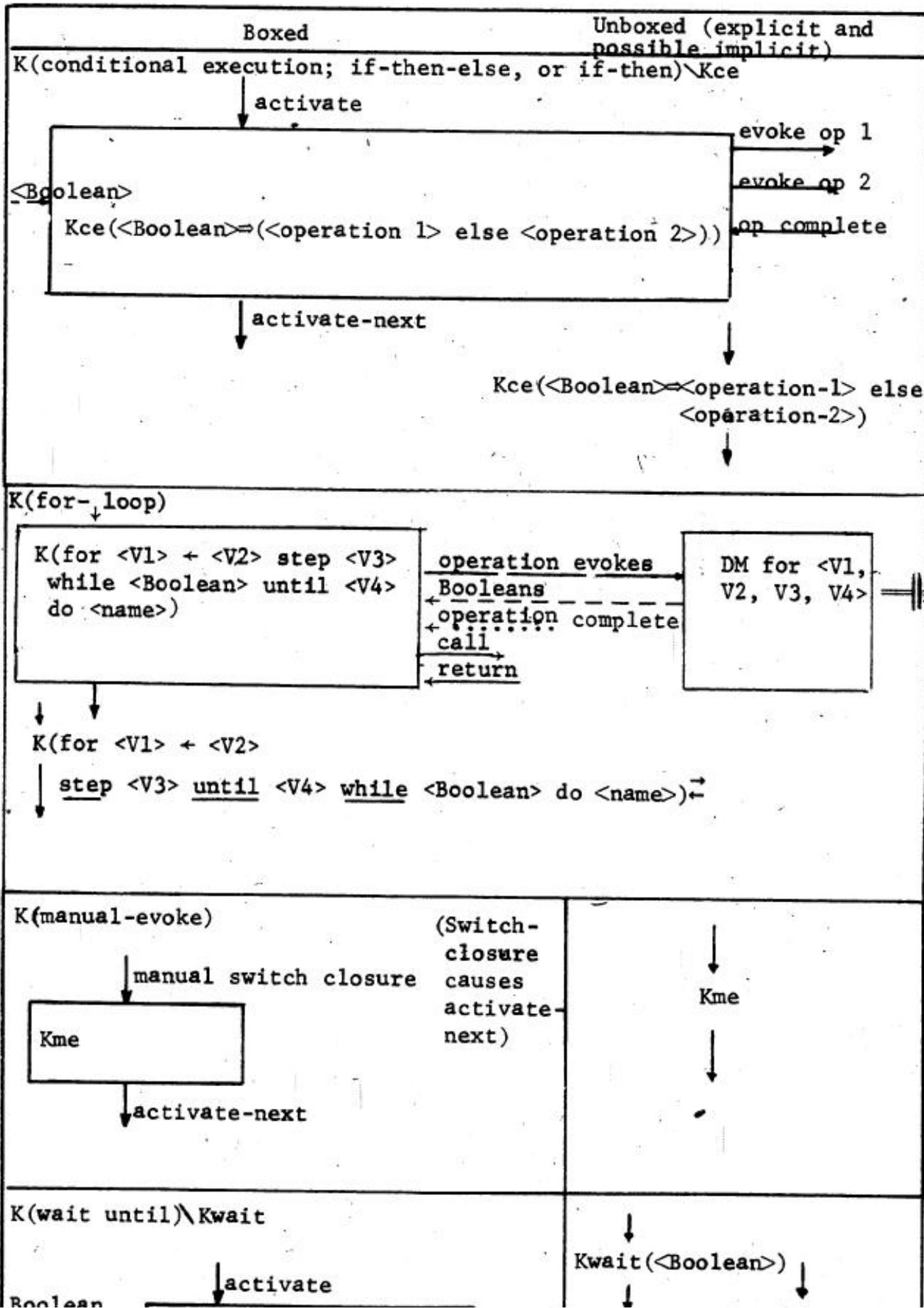
\*\* Where renaming is used for any registers or arrays, either direct substitution (e.g., DMgpa(X,Y), M(array; ZETA[0:256]<15:0>)) or definition (e.g., DMgpa(X := A, Y := B), M(array; ZETA[0:256]<15:0> = A[0:256]<15:0>)) may be used. We also use "\" in place of ":="

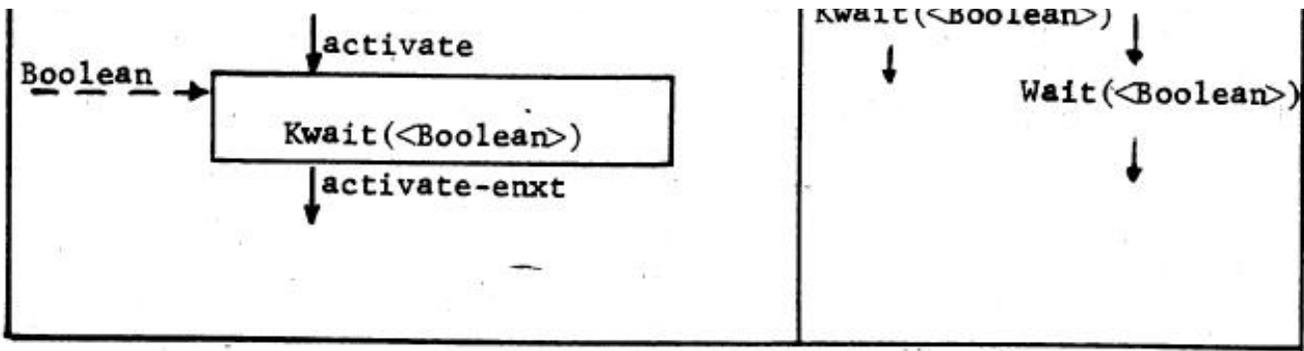
definition (e.g., `DMGPA(X := A, Y := B), M(array; ZETA[0:256]<15:0>  
= A[0:256]<15:0>))` may be used. We also use "`\`" in place of "`:=`"  
to denote an alias for a name.

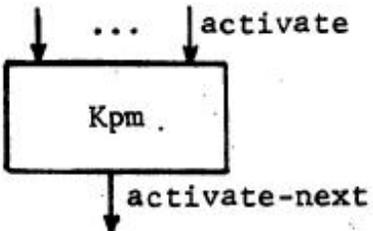
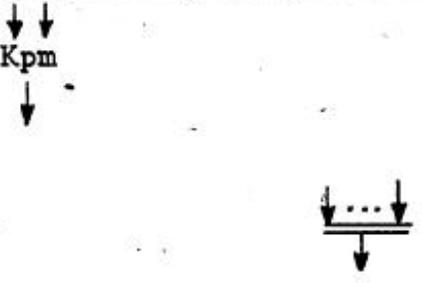
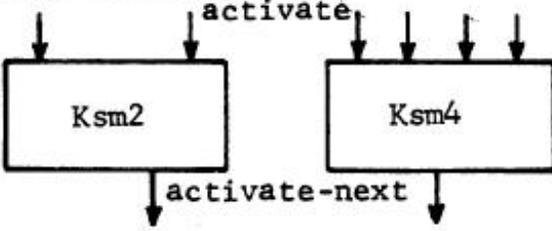
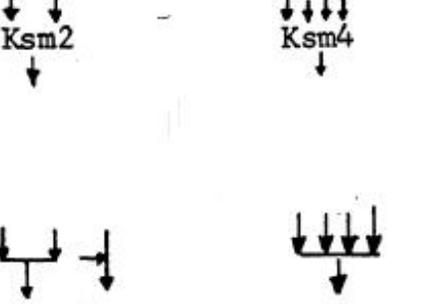
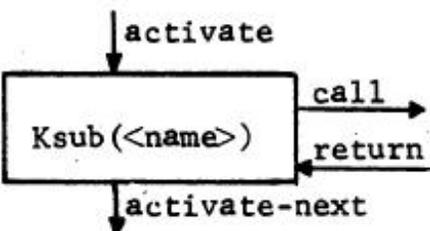
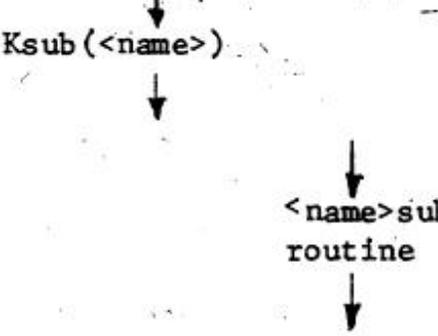
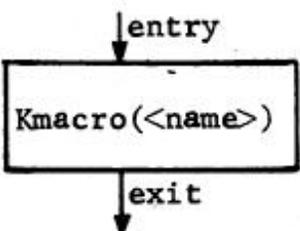
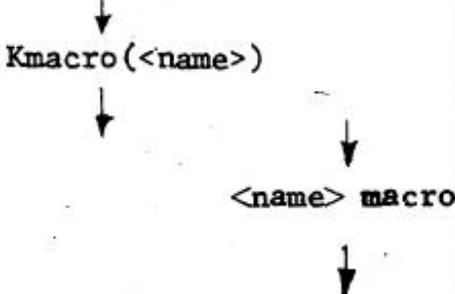
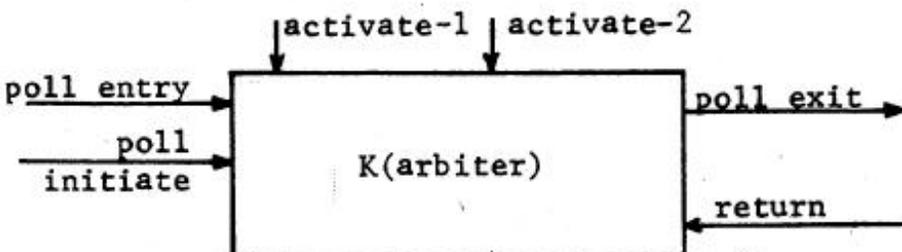
Data Operators \D's

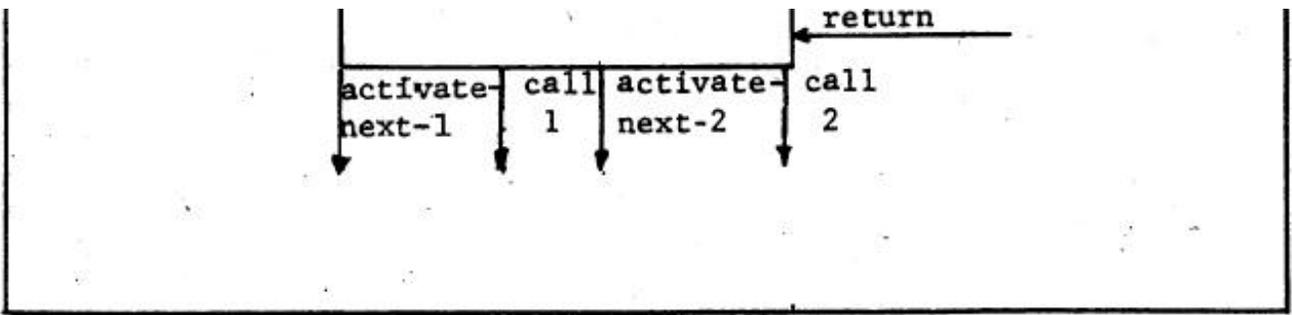


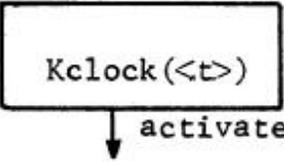
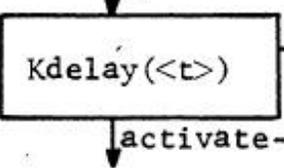
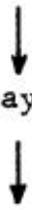
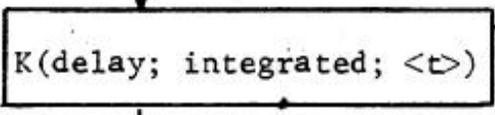
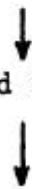
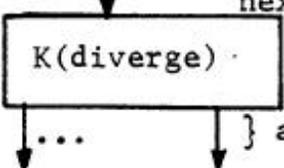
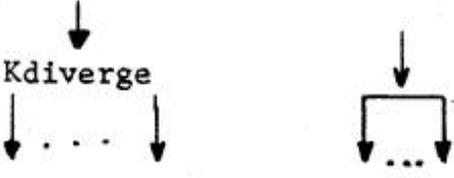
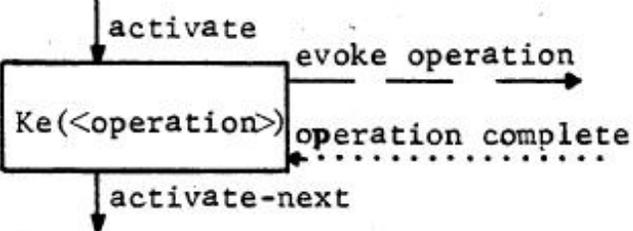
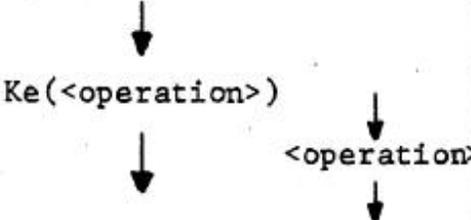
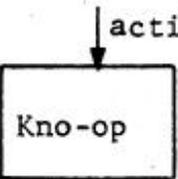






Boxed	Unboxed (explicit and possible implicit)
<p>K(parallel merge)\Kpm (If all activates, then activate-next.)</p> 	<p>Unboxed (explicit and possible implicit)</p> 
<p>K(serial merge)\Ksm (Any activate causes activate-next.)</p> 	
<p>K(subroutine)\Ksub (for calling closed subroutine)</p> 	
<p>K(macro)\Kmacro (Not a module; an open subroutine; substitute RTM control part called &lt;name&gt;.)</p> 	
<p>K(arbiter)</p> 	



Boxed	Unboxed (explicit and possible implicit)
<p><math>K(\text{clock}) \setminus K\text{clock}</math> (Emits activate-next signals at intervals of <math>t</math> seconds. The pulse width must be adjusted properly.)</p> 	<p><math>K\text{clock}(\langle t \rangle)</math></p> 
<p><math>K(\text{delay} \setminus K\text{delay}</math> (Activate causes activate-next after <math>t</math> seconds.)</p> 	<p><math>K\text{delay}(\langle t \rangle)</math></p> 
<p><math>K(\text{delay}; \text{integrated})</math></p> 	<p><math>K(\text{integrated delay}; \langle t \rangle)</math></p> 
<p><math>\cdot K(\text{diverge})</math> (Implicit modules - Activate causes all activate-nexts.)</p> 	<p><math>K\text{diverge}</math></p> 
<p><math>K(\text{evoke}) \setminus K\text{e}</math></p> 	<p><math>K\text{e}(\langle \text{operation} \rangle)</math></p> 
<p><math>K(\text{no-operation}) \setminus K(\text{no-op}) \setminus K\text{nop}</math> (null (delay) operation)</p> 	<p><math>K(\text{no-op})</math></p> 

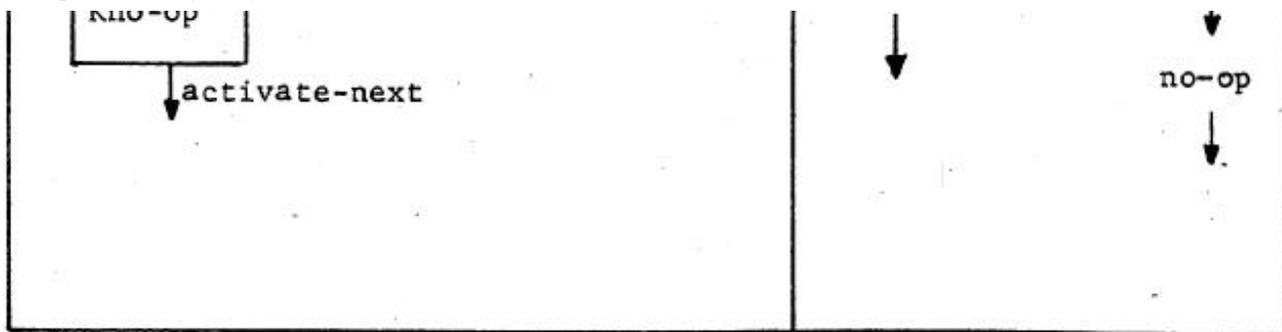
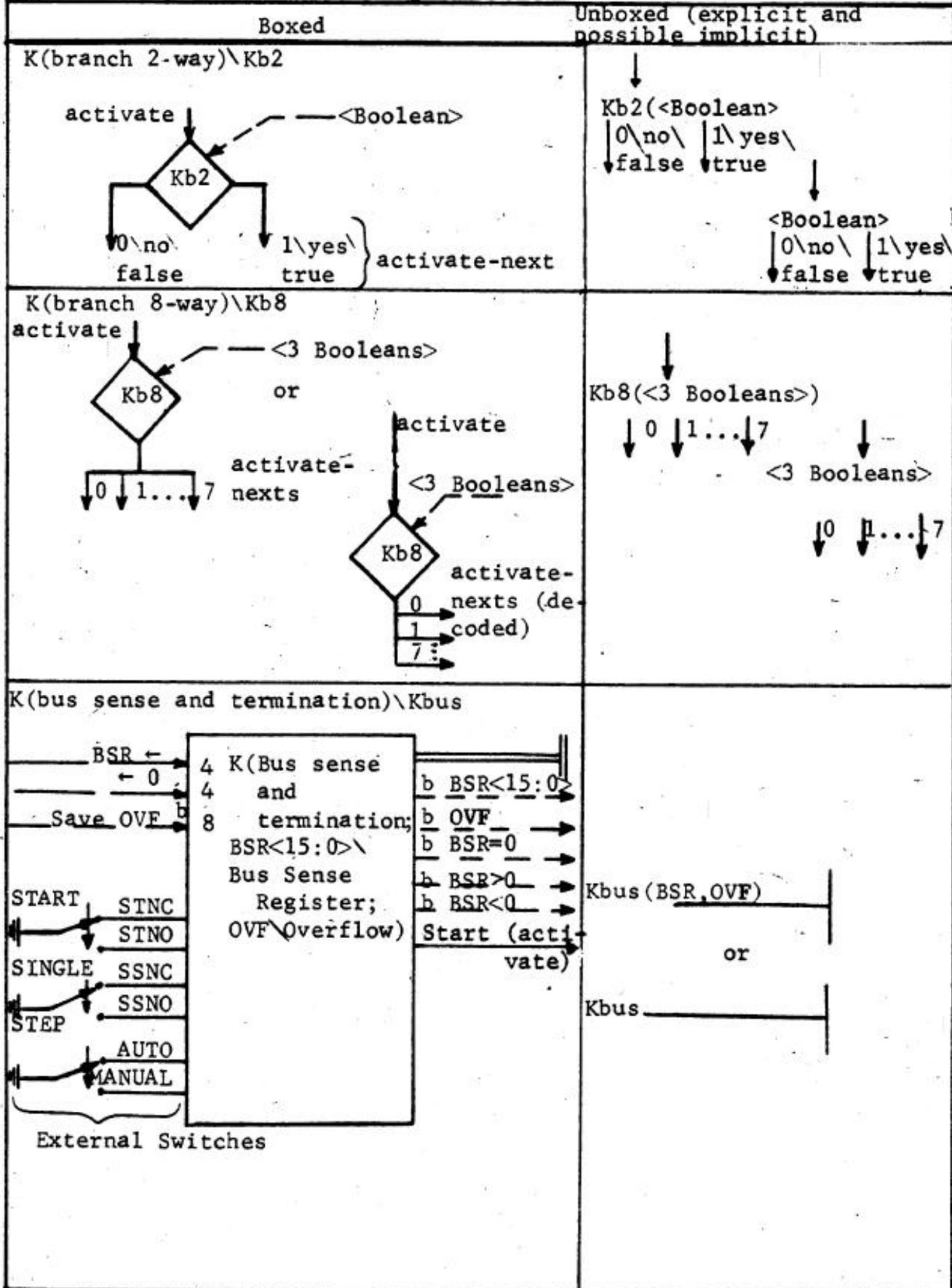


Fig. 16. Module diagrams for the PDP-16\RTM set.





[previous](#) | [contents](#) | [next](#)

4. The receiving module senses the DR signal and reads the data from the Bus.
5. The receiving module puts a Bus DATA ACCEPTED\DA signal on the Bus.
6. The Kbus module senses the DA signal and passes it on to all Ke module operation-complete inputs as the Bus DONE signal. Only the most recently evoked Ke responds to this signal.

Drawing boxes around modules in the RTM system diagrams is pretty, but the boxes are not really necessary. Furthermore, the slashed control lines linking the K part of the system to the DM part of the system can be inferred from the notation that accompanies each K module. Thus, throughout the rest of the book we shall draw RTM system diagrams as shown in Figure 15, which is the compact version of Figure 6. Notice that the K(serial merge)\Ksm is drawn simply as a merging of control arcs. Also, the Kb2 module reads  $I = 0$  instead of  $BSR = 0$ , since this is a better description of the condition being tested.

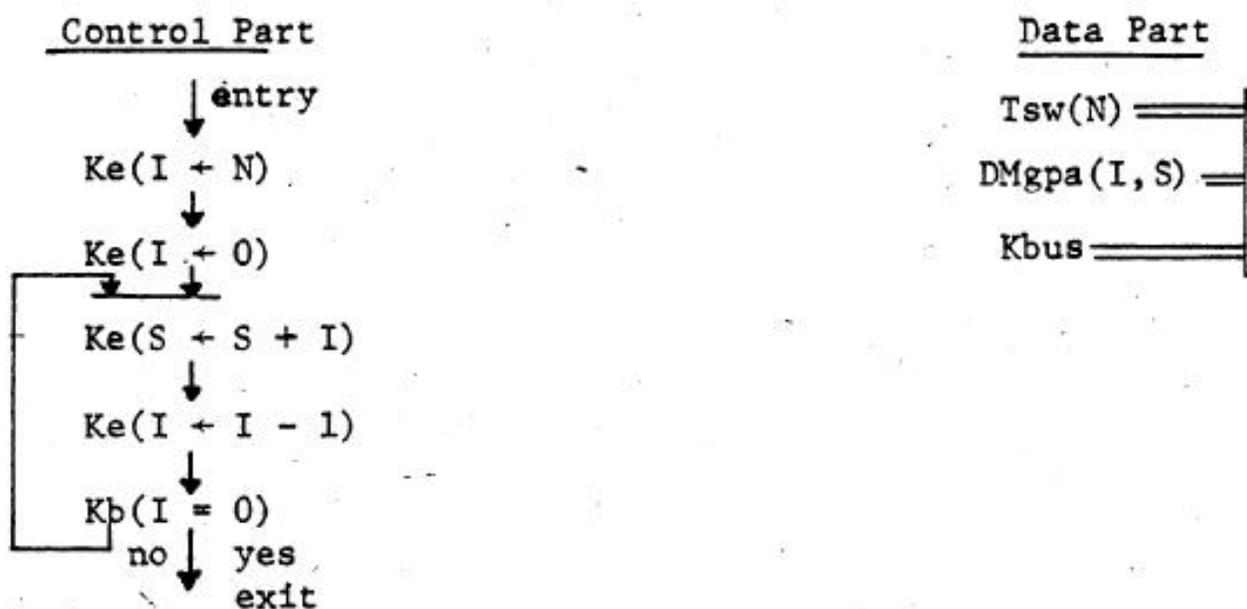


Fig. 15. RTM system diagram (compact) of Fig. 6.

## SUMMARY OF REMAINING MODULES

### Introduction

In this section we present descriptions of the remaining Register Transfer Modules. Block diagrams of all

RTM modules are given in Figure 16. We encourage the reader to use these block diagrams as module ready references. For the K modules, which are used to specify flowcharts, three types of notation are shown. The first is the PMS boxed form with signals, which we have used for clarity in most of the examples in this chapter. The second is the unboxed (explicit) form of notation, in which we have just removed the boxes, as was done in the example of Figure 15. In the third notation, unboxed (implicit), we also remove the PMS type designations from the statements, because they can be inferred from the flowchart and a knowledge of RTM's. We use the second and third notations throughout the remainder of this book.

The block diagrams of Figure 16 describe the modules at the RT level. In the

DMflag, below). The Kbus is a double height, extended length board. In addition, a special double height board must be connected at the opposite end of the Bus, to provide electrical termination.

## RTM SYSTEM OPERATION

Now that some basic modules have been described in detail we can illustrate how they function, when connected together by discussing the previous example in detail. The example is the one that was presented in Figure 6.

Notice that we have renamed the registers to correspond to the names used in the algorithm. For instance, the A and B registers of the DMgpa are renamed the I and S registers, respectively. The T(switch register) that is shown hasn't been described in detail yet, but it simply holds the value corresponding- to the binary switch value, and can be read, i.e.,  $\leftarrow N$ .

This particular implementation-operates as follows:

- 1. The power is turned on and all registers (except the manually set switches) are initialized to zero. This is carried out via the Bus POWER CLEAR line.
- 2. The manual START button is pressed and the Start control signal activates the first Ke module, which corresponds to the operation labeled  $I \leftarrow N$ . This module sends a  $\leftarrow N$  signal to the T(switch) and a  $I \leftarrow$  signal to the DMgpa.
- 3. Now the actual transfer is carried out under control of the Kbus, T(switch), and DMgpa. That is, the N register gates its stored information onto the Bus, and subsequently the DMgpa reads the information on the Bus into its I register. Notice that the Bus is fundamentally just a link (L), and not a register.
- 4. When the data transfer is completed, the first Ke receives the operation-complete signal (via DONE) and subsequently activates the- next Ke module,  $S \leftarrow 0$ .
- 5. The  $S \leftarrow 0$  Ke module is now in control of the system and the cycle of evoking an operation is repeated for this module, causing 0 to be read into the S register of the DMgpa.
- 6. Control is passed on to the  $S \leftarrow S + I$  Ke, through a K(serial merge \Ksm) module. (Here a K(serial merge) allows control flow links to merge into a single link. A Ksm module is used whenever control paths for the same Bus are merged.)
- 7. The next Ke causes I to be decreased by 1 (i.e.,  $I \leftarrow I - 1$ ).

8. The I = 0 Kb2 module utilizes the fact-that the result of the previous register transfer is stored in the BSR register, which can be tested for 0. If the register is non-zero, control flow activates the serial merge causing the loop to be traversed again(steps 6, 7, and 8). When BSR = 0, the loop is terminated and the process stops.

Although the synchronization of the Bus transfers is invisible to the RT level designer-user, it operates as follows:

- 1. A Ke module is activated and given control of the system.
- 2. The Ke sends the evoke operation control signal to the sending and receiving DM, M, or T modules.
- 3. The sending module gates its information onto the Bus and puts a Bus DATA READY\DR signal on the Bus.

Start signal to activate the first control module. When the AUTO/MANUAL switch is set to AUTO, the RTM system is in continuous operation (i.e., running). When the switch is set to MANUAL, then pulsing (depressing) the SINGLE STEP switch will cause one Bus transfer at a time to be executed (the SINGLE STEP switch does this by withholding the Bus DONE signal until it is pulsed).

3.It provides the reset signal, called POWER CLEAR, which initializes all modules when power is turned on. A switch can also be connected to provide this function manually.

4.It allows sense lights to be connected to the BSR register so that data transfers that use the Bus may be monitored.

5.It provides for a word source of zero, through the operation <-0.

6.It allows the transfer of data to the BSR for testing, through the operation BSR<-.

7.It forms the following Boolean outputs which are available after each control step using the Bus:

BSR<15:0> = 0 (detects whether the last word transferred was a zero)

BSR<15:0> > 0 (detects whether the last word transferred was positive)

BSR<15:0> < 0 (detects whether the last word transferred was negative)

BSR<15:0> (16 Booleans, one for each bit)

OVERFLOW\OVF (the carry out from an addition, borrow out from a subtraction, or the shift out from a shift)

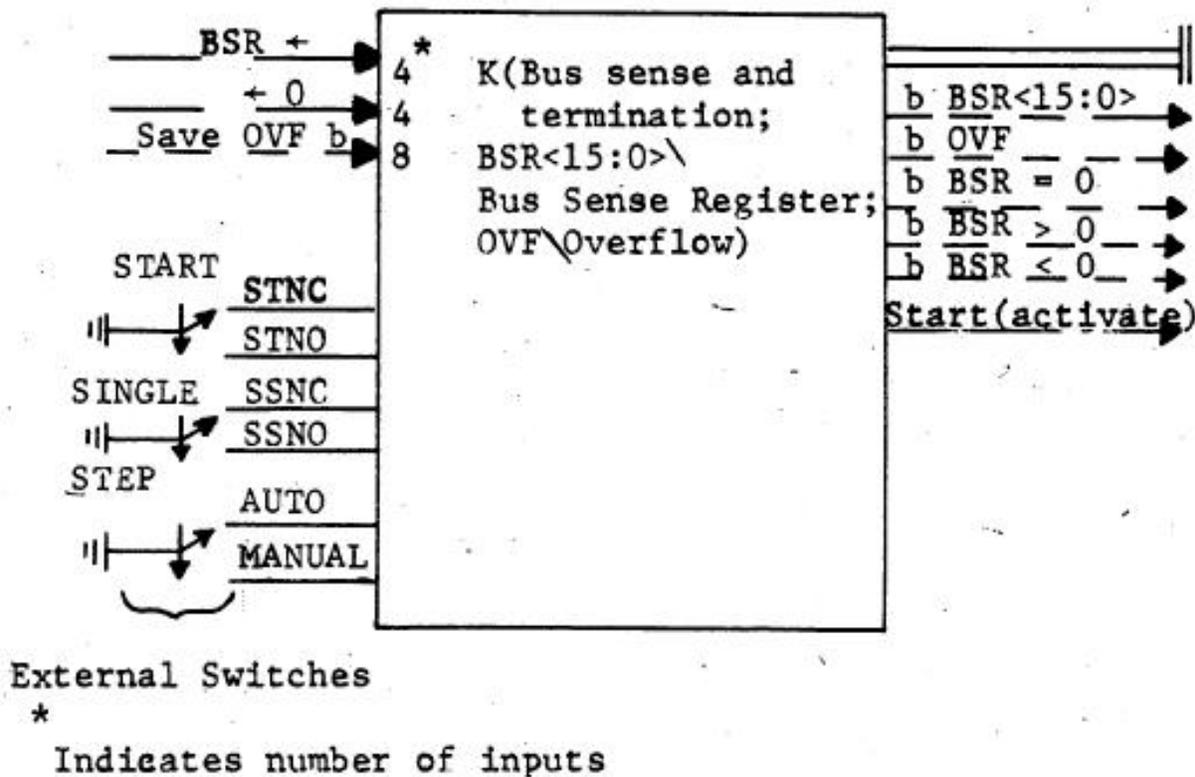


Fig. 14. Module diagram of K(bus sense and termination)

## module

Whenever the OVF signal on the Bus is to be saved in the OVF bit register, the Save OVF input to Kbus must be evoked. This operation does not cause a redundant DATA ACCEPTED\DA timing signal to be produced (see discussion of

Similarly, the source expression  $\leftarrow (RS \ll (A+B))/2$  indicates that the output will be the result (i.e.,  $(A+B)$ ), shifted right one bit, with RSI becoming the most significant bit of the output.

After a shift operation one may want to remember the bit that was shifted out. This can be accomplished by using the overflow facility. There is an OVERFLOW\OVF line on the RTM Bus, and each DMgpa has an output connected to this line.(3) After any given operation the value of OVF may be stored in a one bit OVF register in the Kbus module (see below) and be available for use as a Boolean output of Kbus. The values that OVF takes for DMgpa operations where it is of interest are shown in the table in Figure 13.

<u>DMgpa Operation</u>	<u>OVF</u>
$\leftarrow A + 1$	The carry out of the addition
$\leftarrow A + B$	The carry out of the addition
$\leftarrow A - 1$	The borrow out of the subtraction
$\leftarrow A - B$	The borrow out of the subtraction
$\leftarrow A \times 2$	A<15> before the shift
Result/2	Result<0> before the shift

Fig. 1-3. Table of values of OVF for associated DMgpa operations.

All 16 bits of the A register, and bits 15 and 0 of the B register (i.e.,  $A<15:0>$ ,  $B<15,0>$ ) are available as Boolean outputs of a DMgpa.

The DMgpa<sup>0</sup> consists of two double height, extended length boards interconnected via a rear connector.

K(bus sense and termination)\Kbus. Each independent Bus in an RTM system requires a centralized control module. It has a register, Bus Sense Register \BSR, which always contains the result of the last register transfer that took place via the Bus. Each bit of  $BSR<15:0>$  is available as a Boolean.

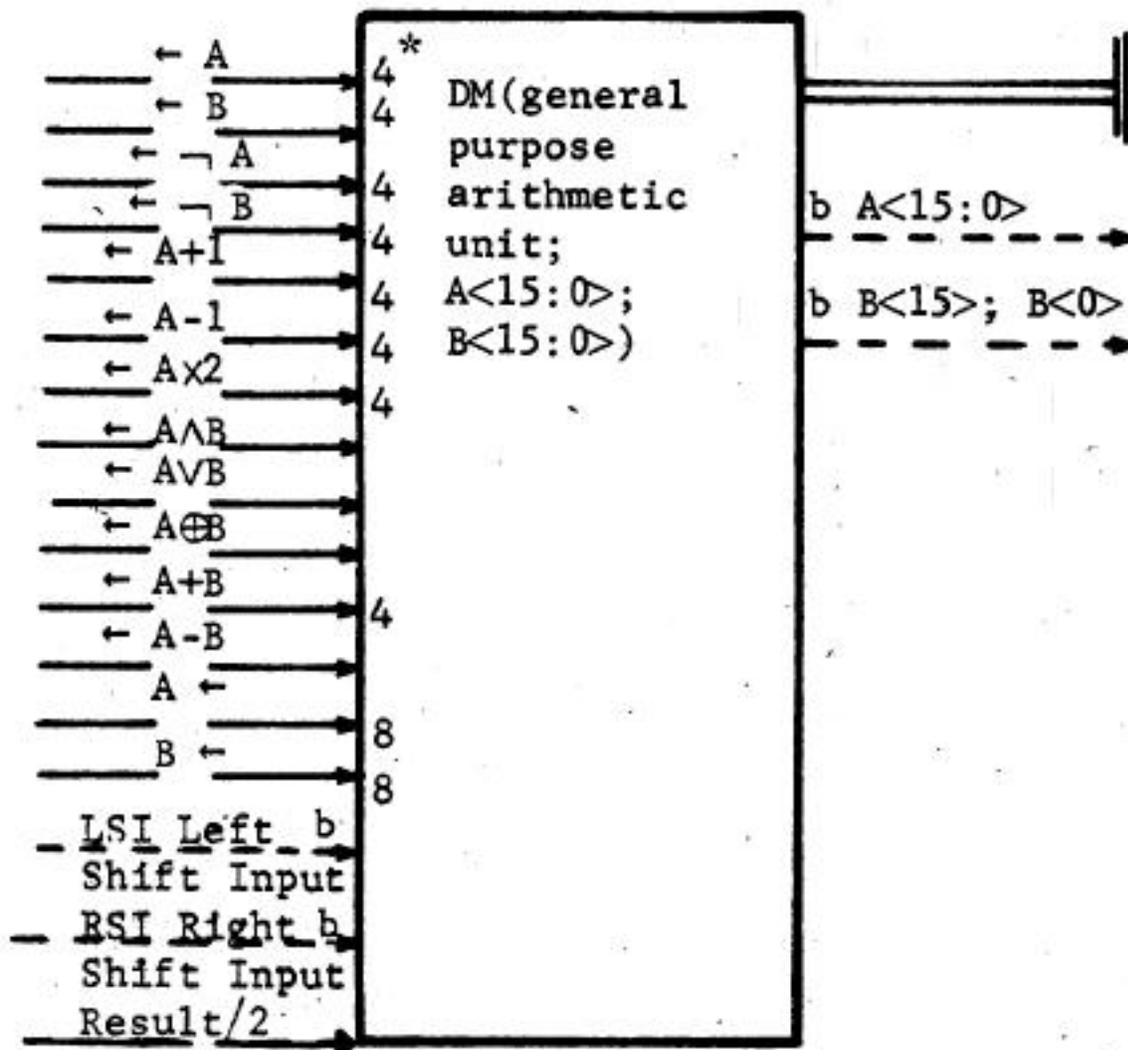
The block diagram of Kbus is shown in Figure 14. Kbus provides for the following functions:

1. It monitors all register transfer operations via the Bus transfer sequencing signals, and supplies the operation completion signal to the Ke modules to indicate that the requested function has been completed. This signal is called Bus DONE\DONE. As indicated, these signals are invisible to the

RT level user because they are a prewired part of the Bus.

2. It provides for manual control of an RTM system, if switches are connected to it. A set of suitable switches are provided in the T(lights and switches), described later. The manual START switch causes an output

-----  
3. For 1972 PDP-16's, the OVF Bus signal is not valid when more than one DMgpa is connected to a given Bus.



\*These numbers indicate the number of module inputs available for evoking the shown operation. The absence of a number denotes that there is one input. If more inputs are desired for 1972 PDB-16's, negative logic OR gates must be used, since the operation evoke signals are negative logic.

Fig. 12. Module diagram of DM (general purpose arithmetic unit).

conventional logic operators for a DMgpa are  $\leftarrow \neg A$ ,  $\leftarrow \neg B$ ,  $\leftarrow A \wedge B$ ,  $\leftarrow A \vee B$ , and  $A \oplus B$ . These are interpreted to be the bit parallel logical COMPLEMENT, AND, OR, and Exclusive OR, respectively.

Four data transfer operations are provided, namely  $\leftarrow A$ ,  $\leftarrow B$ ,  $A \leftarrow$ , and  $B \leftarrow$ . Finally, there are two shifting operators. The left shift is quite simple, i.e.,  $\leftarrow A \times 2$ . The right shift uses a special input in which the notation "(result)" appears, i.e. (result)/2. This input is always used in combination with a conventional evoke input coming from a Ke, and it signifies that a right shift is to be performed on the result of the conventionally evoked operation. For instance, a  $\leftarrow A+B$  accompanied by the input (result)/2 indicates that the operation  $\leftarrow (A+B)/2$  is to be performed.

The  $\leftarrow A \times 2$  and  $\leftarrow$  (result)/2 operators, strictly speaking, are not arithmetic operators. They simply imply that a logical one-bit shift to the left or right, respectively, is to be made. Since a shift implies that an end bit must be coming in from the outside, the two Boolean inputs left-shift-input\LSI and right-shift-input\RSI are provided (see Figure 12). For instance, the source expression  $\leftarrow (A[\text{LSI}]) \times 2$  indicates that the output will be the contents of the A register, shifted left one bit, with LSI becoming the least significant bit, i.e.,  $A\langle 0 \rangle$ .

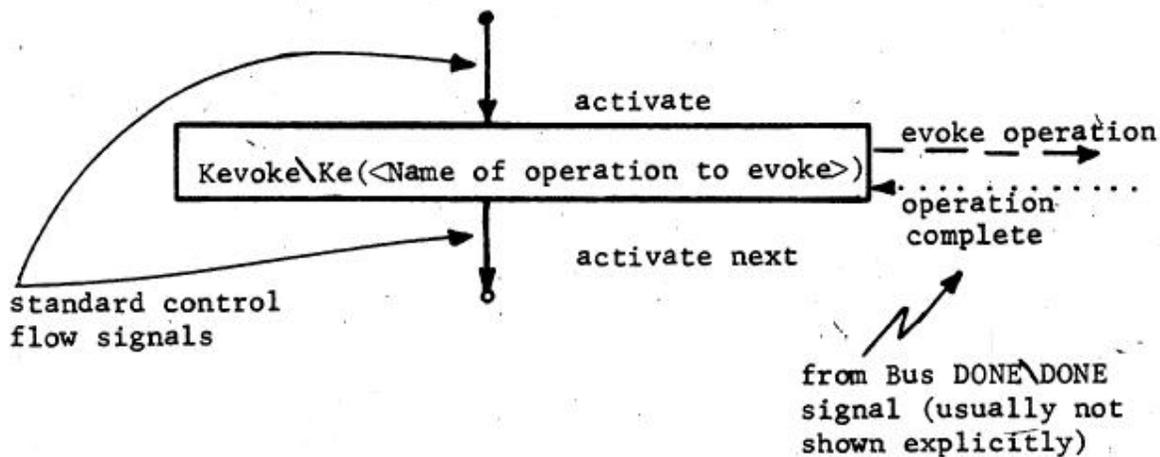


Fig. 10. Module diagram for K(evoke) module.

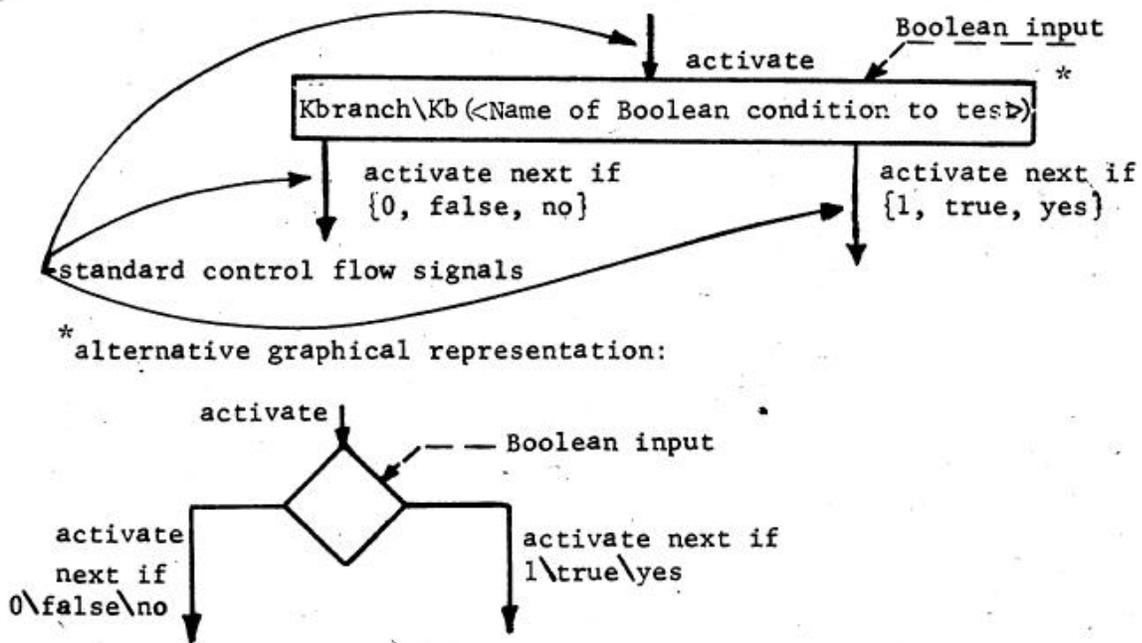


Fig. 11. Module diagram for K(branch 2-way) module

receives a signal, called "activate", at its input terminal which activates the control; (2) Ke sends a signal, called "evoke-operation", to the data-memory part of the system to evoke the desired operation(s); (2) (3) when the operation(s) is (are) completed, the Bus control module sends a signal, Bus DONE\DONE, back to the Ke module at its operation-complete input; (4) Ke then passes control on to the next control module via its output signal, called "activate-next".

The block diagram for a Ke, with its input and outputs labeled, is shown-in Figure 10. Although th? Kevoke has four terminals, the actual PDP-16 module has only three terminals since the activate input and evoke-operation output are the same terminal. The operation-complete is not explicitly shown in RTM system diagrams because in the physical implementation it is prewired (the K modules plug into the control part of the-Bus) and is common for all K modules. Several K(evokes) are mounted on a single height, double length board.

K(branch 2-way)\Kb2. A Kb2 provides for the branching of control flow based on the condition of a Boolean data input variable. Each time a Kb2 is activated, it in turn, on activate-next, activates either of the subsequent control modules attached to it, depending on whether the Boolean input is true (a Boolean 1) or false (a Boolean 0). The block diagram for Kb2 with its two inputs and two outputs is shown in Figure 11. Note that although the rectangle- indicates an RTM component in general, -we also use the diamond to represent the decision taken by the branch. This convention is compatible with the general flowchart conventions (and also PDP-16). Several Kb2's are mounted on a single height, double length board.

DM(general purpose arithmetic unit)\DMgpa. The DMgpa is the workhorse of the modules of the data part of an RTM system because it carries out arithmetic and logic operations. Like most DM modules it can accept data from the Bus and place it, unaltered, in one or both of its two registers, or it can perform (possibly null) data operations on the contents of its registers and place the result on the Bus. These operations and results correspond to the righthand side and lefthand side respectively of a register transfer statement, such as  $A \leftarrow B$ , or  $B \leftarrow A+B$ . The notation used for describing data operations lists the lefthand (destination) sides of these expressions as  $A \leftarrow$  and  $B \leftarrow$ . The corresponding righthand (source) sides are written,  $\leftarrow B$ ,  $\leftarrow A+B$ . The arrow is used in both cages to avoid confusion about whether the register is source or destination. Notice that the DMgpa allows the same register to occur in both the source and destination parts of an evoked register transfer operation if desired.

The block diagram for a DMgpa is shown in Figure 12. A DMgpa has two 16 bit registers, designated A and B. These are declared as  $A\langle 15:0 \rangle$  and  $B\langle 15:0 \rangle$  within the figure. For arithmetic operations these registers are assumed to hold two's complement integers. The conventional arithmetic operations for a DMgpa are  $\leftarrow A+B$ ,  $\leftarrow A-B$ ,  $\leftarrow A+1$ , and  $\leftarrow A-1$ . In RTM system diagrams the evoking of one or more of these operations is indicated by drawing a dashed link from a Ke to the DMgpa and labeling it with the operation to be evoked. A given operation may be called by any number of Ke's, and a different evoke link is shown for each instance of that operation. In Figure 12 one such link is shown for each of the

allowable DMgpa operations. Actually, in PDP-16 modules several operation inputs are available for each operation. The call for a given operation can be extended by using additional input gating logic (negative logic OR gates).

For logic operations registers A and B are assumed to hold bit vectors. The

-----

2. Data transfers with multiple sources and/or multiple destinations are also possible under certain conditions. This will be described later.

All logic voltages are standard TTL signals, 0 ~.8 volts for logical Low, and 2.0 ~ 3.6 volts for logical High. We shall use the terminology that a signal has the value 1 if it is asserted (i.e., active, equal to the binary digit 1, or equal to a Boolean 1), and the value 0 if it is not asserted (i.e. inactive, equal to the binary digit 0, or equal to a Boolean 0).

On the RTM Bus, logical Low signals correspond to 1's (assertion), and High signals are 0's. Likewise control part signals are logical Low for assertion, and the passing of control from one K module to another takes place at the moment of 0 to 1 (High to Low) transition of the signal. On the other hand, Boolean and other data signals are logical High for assertion. Finally, other signals (erg., signals from analog devices, Teletypes, etc.) have a variety of conventions and will be described as needed.

In describing switching circuit components, i.e., gates and flip flops, we shall refer to positive logic and negative logic implementations. Thus, for example, a positive logic OR gate will be referred to in a part of the system in which logical Highs correspond to 1's. If either input of the positive logic OR gate is a 1 (High), its output is a 1 (High). In a part of the system in which logical Lows correspond to 1's we shall refer to negative logic OR gates. If either input of a negative logic OR gate is a 1 (Low), its output is a 1 (Low). The same kind of reasoning applies for AND's, NAND's, NOR's, etc. Notice that a certain type of duality exists, e.g., a negative logic OR gate is a positive logic AND gate, etc.

## FOUR BASIC MODULES

### Introduction

This section describes in detail four modules with which it is possible to build non-trivial RTM systems. Using these modules we shall then discuss in the next section basic control and data flow in an RTM system.

To review the notation to be used: Registers are given by expressions such as  $A\langle 15:0 \rangle$ , which specify both the name and the set of bit positions. Concatenations of several registers into a single long one is indicated by the box, e.g.,  $A\langle 7:0 \rangle \boxed{\text{ } B\langle 7:0 \rangle}$  is a register of 16 bits. To give it a name and label its bits we could have written  $C\langle 16:0 \rangle := A\langle 7:0 \rangle \boxed{\text{ } B\langle 7:0 \rangle}$ . Thus  $:=$  is used to make declarations whereby the name on the left is defined by an expression (which may be arbitrarily complex) on the right. Transfers of bits from one register to another are indicated by the left-pointing arrow, e.g.  $B \leftarrow C$  indicates the transfer from register  $C$  to register  $B$ . Boolean conditions are given by equations, e.g.  $(BSR = 0)$  has the value 1 if the register called  $BSR$  contains the integer 0, otherwise it has the value 0. Boolean conditions are also given simply by the contents of a register; e.g.,  $B\langle 6 \rangle$ , when used as a Boolean condition, would have the value 1 if the bit in  $B\langle 6 \rangle$  was 1 and the value 0 if the bit was 0. If the reader encounters hereafter any notation that he does not understand, he should consult Tables 1, 2, and 3 at the back of this book.

## The Modules

**Kevoke\Ke.** The Ke module is the basic control module in an RTM system. Its function is to evoke (cause) a single register transfer operation in the data- memory part of the system, using the Bus, e.g.,  $C \leftarrow B$ ,  $A \leftarrow A+B$ . It may also simultaneously evoke a data operation that does not require the Bus, such as setting, complementing, or clearing a single bit Boolean register, e.g.,  $D \leftarrow \neg D$ , or it may evoke a Bus-exclusive operation by itself.

The operation sequence of a Ke module is as follows: (1) the Ke

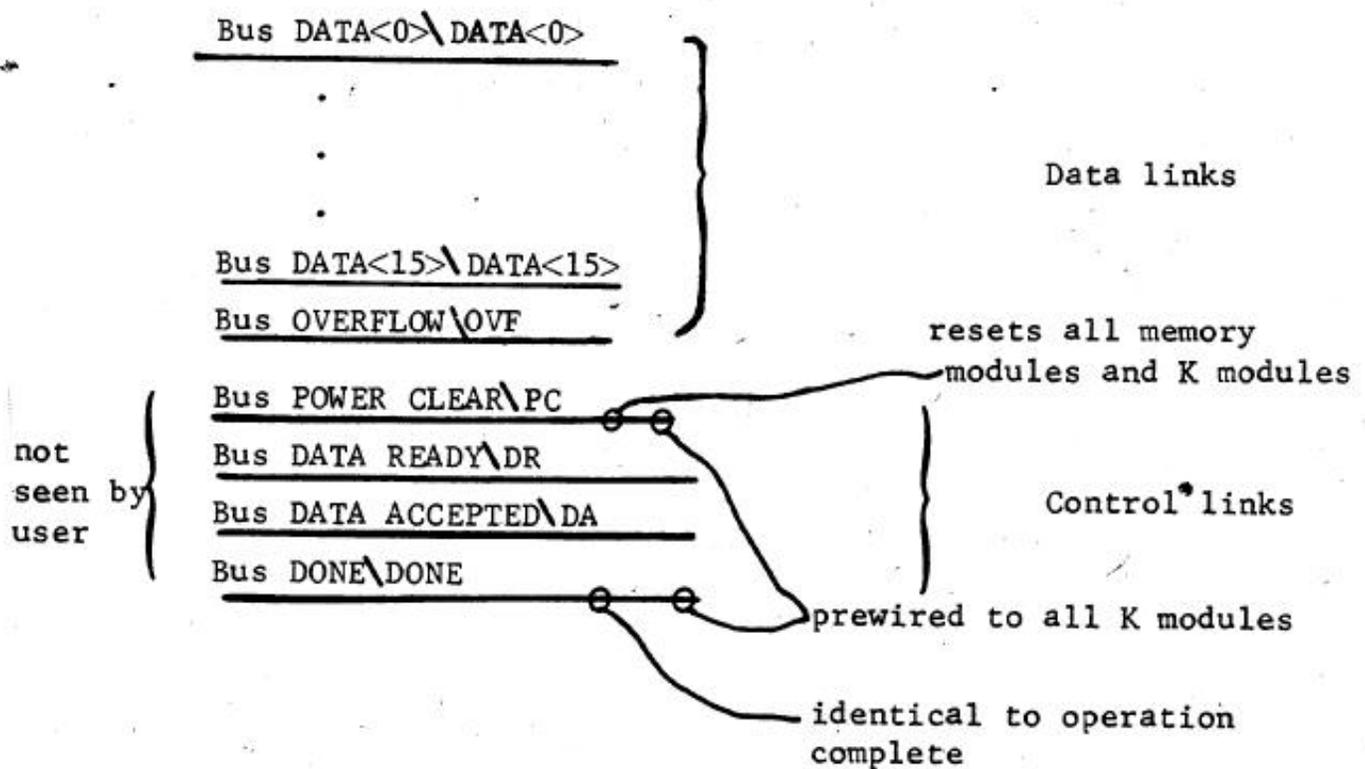


Fig. 9. RTM Bus signals.

RTM system, so they must be transduced. Examples are signals from analog devices, Teletypes, modems, lights, switches, etc. Figures 2 to 6 used ( — — — ), but in subsequent chapters these will be shown merely as \_\_\_\_\_ .

To summarize, in subsequent chapters only two types of links will usually be differentiated graphically: the RTM Bus (shown — or ===), and all other links ( — ). Normally the differences in these latter links is easily seen by the context. Generally, the signal names will be labeled on top of the 'links'.

### Physical Implementation of RTM's

The physical implementation of RTM's depends on the technology being employed. The current DEC PDP-16 modules are constructed using medium scale integrated circuits mounted on double sided printed circuit boards of 5" x 8 1/2" with 72 pins (double height, extended length) or 2 1/2" x 8 1/2" with 36 pins (single height, extended length) or 2 1/2" x 4 1/4" with 36 pins (single height, standard length). A few RTM's, e.g., the DMgpa, are actually two of these boards; these require board connectors on the top and

back of the printed circuit boards. Some RTM primitives, e.g., the Kevoke's, are so small that several are placed on a single height printed circuit board. The boards can be plugged into the back of 5 1/4" x 8 1/2" x 19" wireable panels, which contain a prewired Bus. The panel is plugged into a mounting rack which usually contains a power supply.

Logical Level - Voltage Level Conventions. In Chapter 7, we discuss the switching circuit level design of RTM's. At the RT level, which we are presently discussing, it is generally unnecessary for the reader to know the actual voltages that correspond to logic signals. However, we present them here for completeness. Detailed information is available in the PDP-16 handbook.

alternatively consider them to be unsigned integers or bit vectors). The 8, 12, or 16 bits of a register,  $R$ , are usually denoted  $R\langle 7:0 \rangle$ ,  $R\langle 11:0 \rangle$ , or  $R\langle 15:0 \rangle$ , respectively, thus giving both the range and the numbering to be assigned to each bit position in the register. For example, the bits of  $R\langle 7:0 \rangle$  are numbered from left to right,  $R\langle 7 \rangle, R\langle 6 \rangle, \dots, R\langle 0 \rangle$ . When interpreted as an integer, the  $j$ -th bit would have the value of  $2^j$ . In some cases, when needed, bits are numbered in the opposite order  $\langle 0:15 \rangle$ .

Associated with each data type is a full complement of appropriate data operations. This allows the user to selectively treat a 16-bit data word as a vector, for instance, if he limits himself to using only operations that apply to that data type. The data operations are more appropriately listed later, in association with the DM modules that implement them.

## Signal Types and Designations in Diagrams

The RTM Bus. The Bus carries the data among the registers for the various register transfer operations. All the DM, M, and T modules connect to it for inter-register transfers. Two signals, Bus DONE and Bus POWER CLEAR, connect to all control modules. The Bus is shown in Figure 9. There are two categories of signals on the Bus -- data signals, and control signals. The data signals correspond to the 16 bits of a data word and are denoted  $DATA\langle 15:0 \rangle$ . There is a 17th link which carries OVERFLOW\OVF (actually carry) information. There are four Bus control signals -- POWER CLEAR which initializes the RTM system when it is turned on, and three internal Bus transfer sequencing signals that are essentially invisible to the RT level user. K modules are implicitly connected (i.e., prewired) to the pertinent control part of the RTM Bus to use the sequencing signals. A special control module, Kbus, is connected to the Bus for the purpose of controlling and defining it.

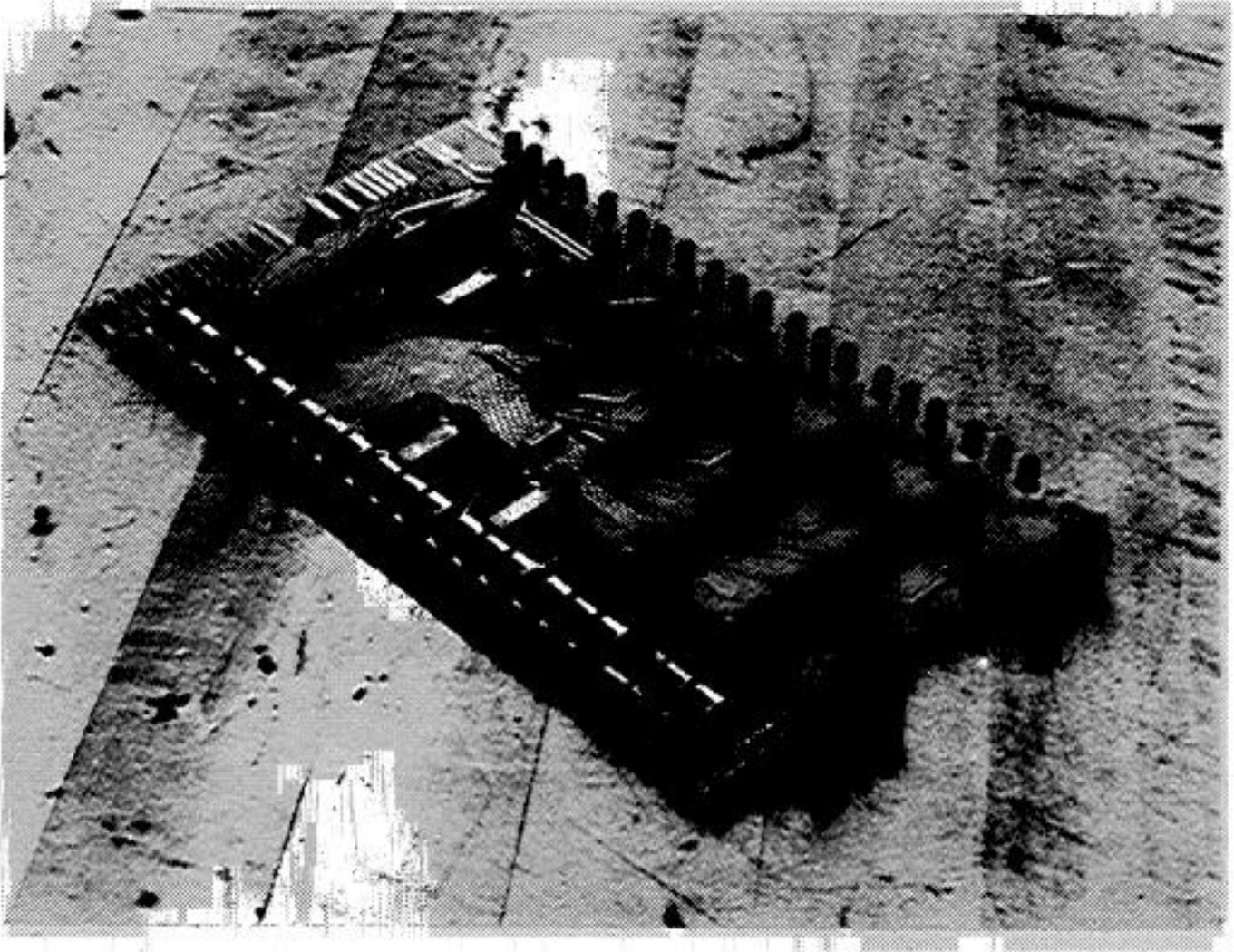
In the schematic diagrams used in this book, the Bus is denoted either as a single line or parallel lines (I or II) and a module connection to the Bus is denoted "—" or "==" (see Figure 6).

Control Part Signals. There are two types of control signals in RTM systems. One type is carried by the links that pass control from K module to K module, called control flow links. In the schematics, these links are shown as solid lines (—). The other type of signal is passed from K modules to the data part of an RTM system to evoke data operations, along evoke-operation links. These links are drawn in the RTM diagram of Figure 6 as ( — — — ); in subsequent chapters, these links will normally not be drawn, or if they are drawn, they will be shown as solid lines (possibly labelled). All the various styles of lines used in this chapter are used to avoid confusion while introducing new material. In subsequent chapters, however, we use mostly solid lines for compactness and simplicity.

Boolean and Other Data Signals. Single bit Boolean data signals may be passed between the control and the data parts of an RTM system and also solely within either part. A link carrying this type of signal is shown in Figures 2 and 6 as a dashed line, (-----); in subsequent chapters it will be shown as (b <Name>

or ( <Name> ). Conventional combinatorial switching circuits can be used to form functions of Booleans.

Other Signals. These signals are those whose form is not directly usable by the



memory (M). However, there are also modules with only an M part, e.g. core memory arrays, solid, state scratchpad memories, and read-only memories.

**K Modules.** K stands for control. K modules are responsible for evoking the various operations of the DM and M modules, including data transfers. K's are interconnected among themselves to determine the sequence in which the operations are evoked, and in addition they can use information from the DM and M modules to decide which operation to evoke next. K modules can be used to connect a series of operations together as a subroutine. They also synchronize control when there is more than one operation taking place at a time.

**T Modules.** These modules are transducers. They provide an interface to the environment outside an RTM system by changing external signals into RTM compatible signals; Examples include the Teletype

interface, analog/digital converters, lights, switches, other combinatorial and sequential switching circuits, etc. T modules generally plug into the RTM Bus and appear to the K's to be much like. DM's and M's.

## Data Types

RTM's provide for two basic data types: Booleans (single bit Boolean variables); and 8, 12, or 16-bit signed two's complement integers (the user may



Fig. 8a. Front view of sum-of-integers system: rack, mounting panel, and T(lights and switches).

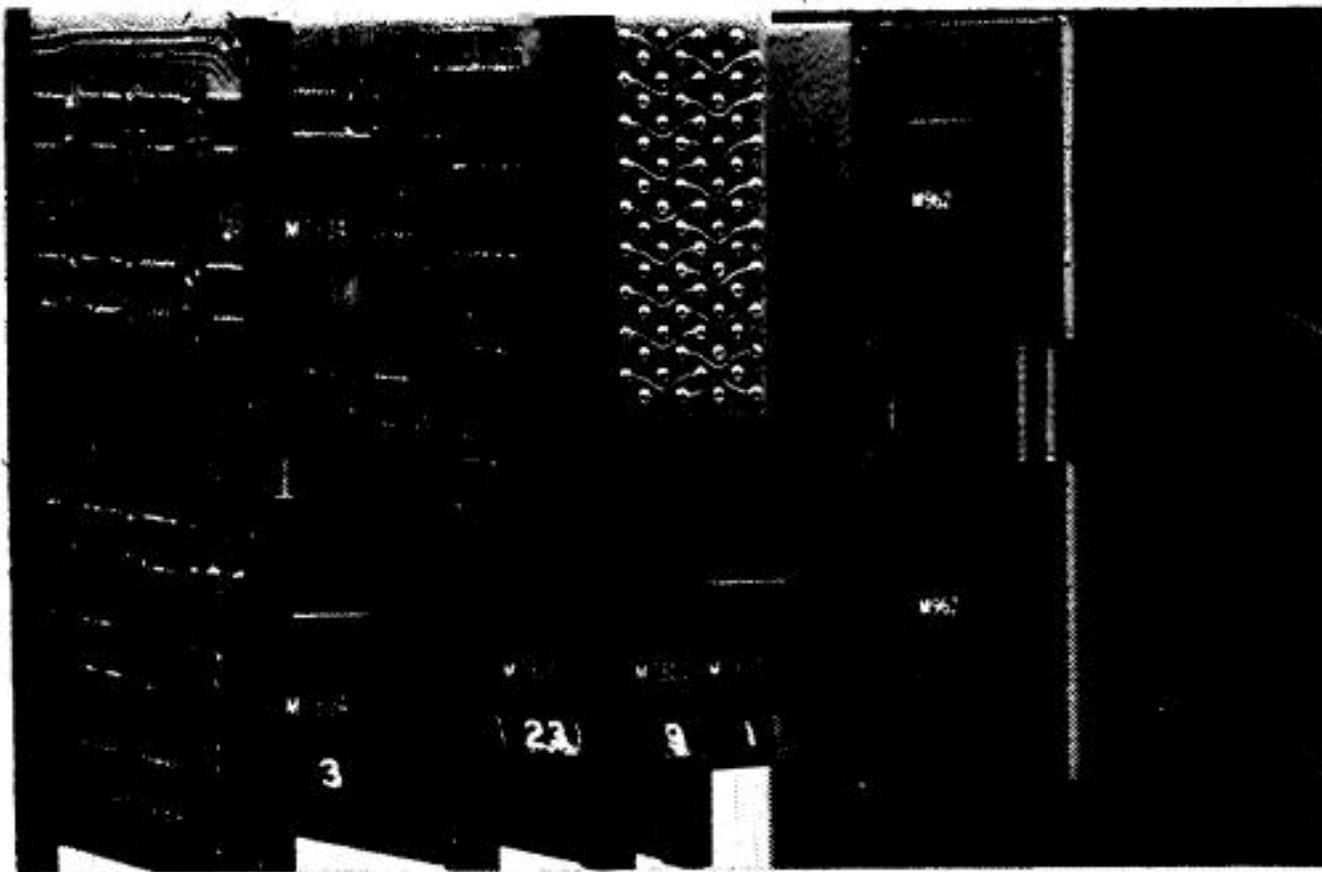


Fig. 8b. Rear view of sum-of-integers system, showing modules inserted into mounting panel.

<u>Operation</u>	<u>1st Pin</u>	wired to	<u>2nd Pin</u>
I ← N	7BB2		6BU2
	6BU2		5BN1
	5BN1		8BF1
S ← ∅	6BV2		6BE2
	6BE2		5BJ1
	5BJ1		7BR2
S ← S + I	6BF2		6BP1
	6BS1		6BK2
	6BK2		5BJ2
	5BJ2		4BK2
I ← I - 1	6BL2		6BS2
	6BS2		5BN2
	5BN2		4BL2
Branch	6BT2		9BH1
	9BE2		6BR1
Boolean	7BT2		9BH1
AUTO	8AK2		7BE2
MANUAL	8AJ2		7BD2
SINGLE STEP	8AF2		7BF1
NORMAL	8AH2		7BH1
START	8AL2		7BJ1

START

0ALZ

7BJ1

NORMAL

8AP2

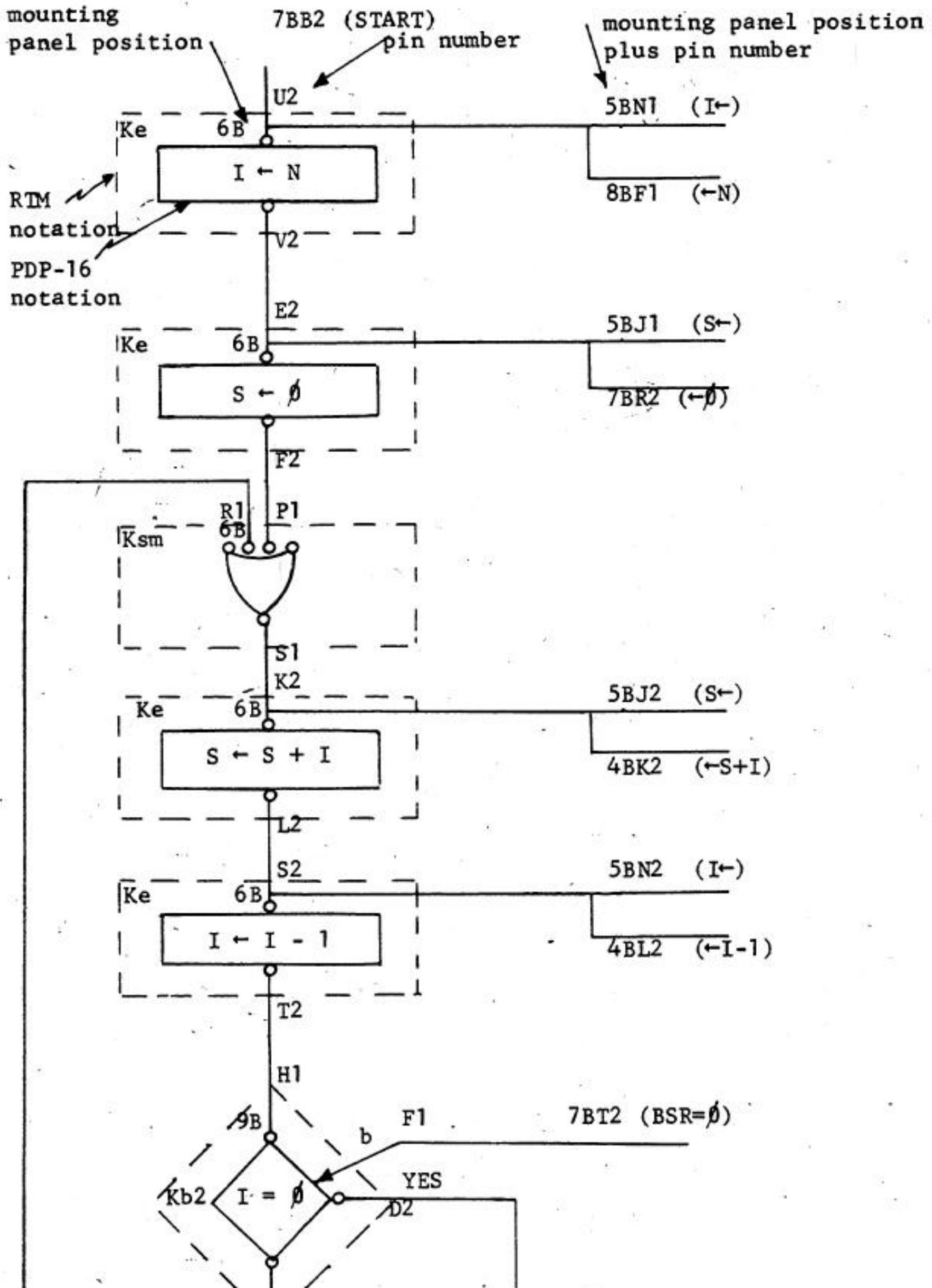
7BF2

POWER CLEAR

8AD2

8BB1

Fig. 7c. Wiring list for sum-of-integers to N.



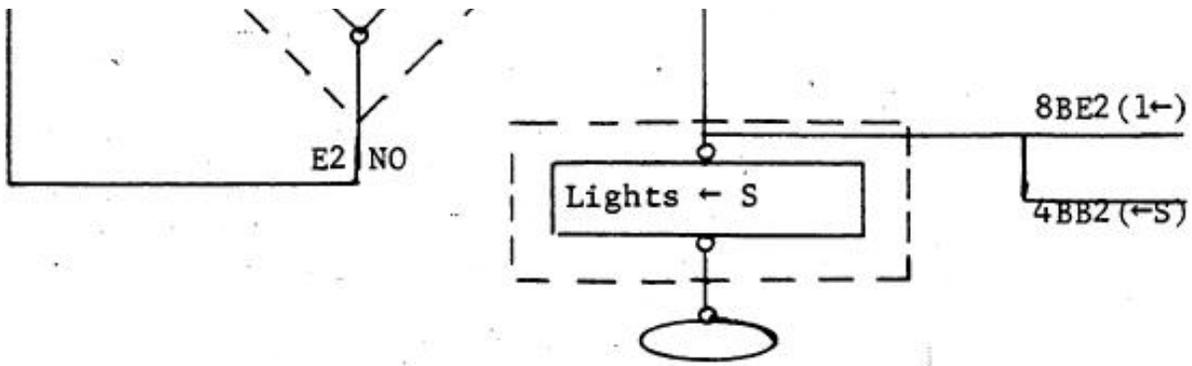


Fig. 7b. Control part, with pin numbers, for sum-of-integers to N. RTM and PDP-16 notation shown.

<u>Quantity</u>	<u>Name of Board</u>	<u>DEC #</u>	<u>Code # Shown on Board</u>	<u>Mounting Panel Position</u>
1	T(lights & switches)	KBM16X	M7334	8A,B
1	K(Bus sense)	KBS16	M7304	7A,B
1	DMgpa(registers)	KAR16	M7301	5A,B
1	DMgpa(control)	KAC16	M7300	4A,B
1	K(branch 2 way)	KB16-A	M7312	9B
1	K(evoke)	KEV16	M7310	6B
1	Bus termination	KTM16	M962	1A,B

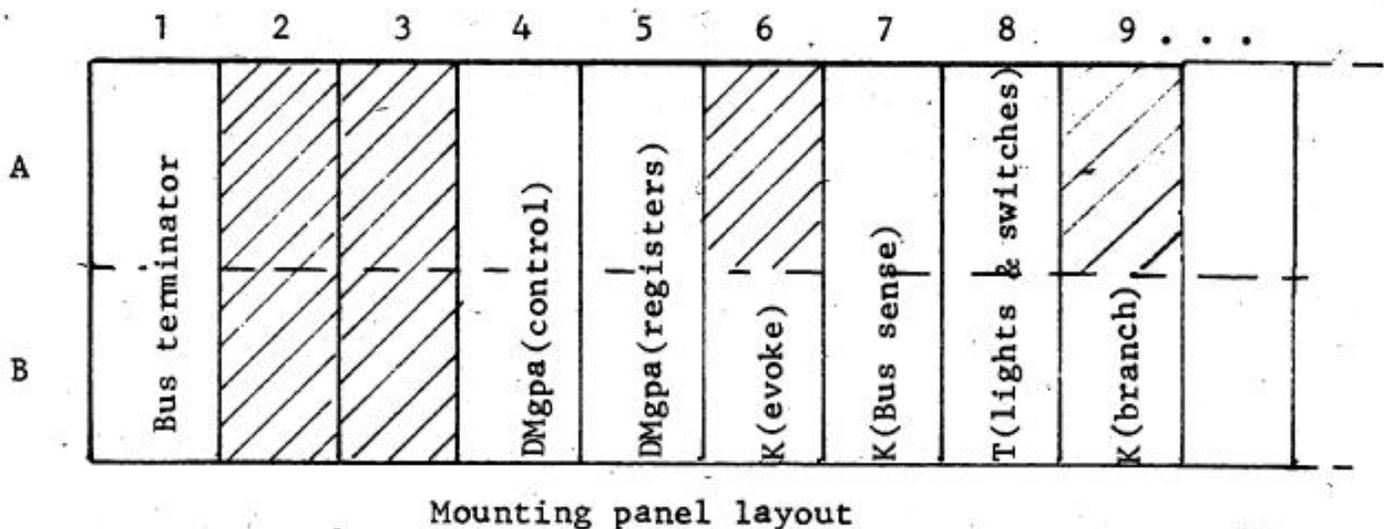


Fig. 7a. Parts list and mounting panel pin assignment for sun of integers to N.

### Categories of Modules

DM and M modules. DM stands for data operation combined with memory, and M stands for memory. The DM modules provide the register transfer gating paths and combinatorial switching circuits for performing simple arithmetic and logical functions on data. The expression "lefthand side <- righthand" side indicates that the integer or Boolean vector value of the righthand side is computed, or taken as a source, and placed in the register on the lefthand side.

The D (data operation) part of the DM module carries out evaluation of the righthand side of such an expression in which an integer or Boolean vector value is computed, e.g.,  $\leftarrow A+B$ ,  $\leftarrow A-B$ , or  $\leftarrow A+1$ . The M (memory) part of the DM module is just the registers (e.g., A, B) that hold data between statements. The operations on memory are usually just writing (M  $\leftarrow$ ) and reading ( $\leftarrow$  M). The use of bussed data transfers makes it necessary to combine data operations (D) with



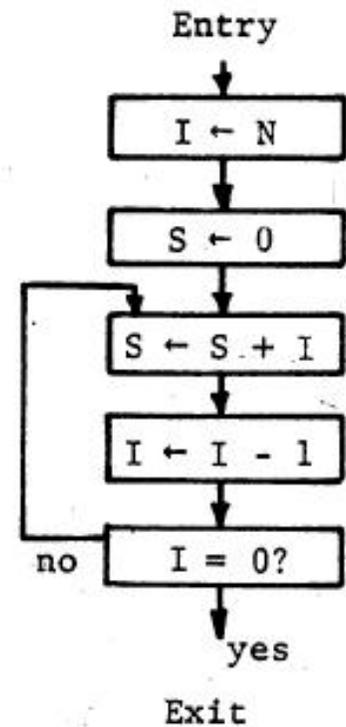
[previous](#) | [contents](#) | [next](#)

**Variables:**

N&lt;15:0&gt;/Constant

I&lt;15:0&gt;/Counter

S&lt;15:0&gt;/Sum

**Algorithm:**

$$S = \sum_{I=1}^{I=N} I$$

for  $N \geq 1$

Fig. 5. Algorithm flowchart for summing positive integers from 1 to N.

This brief introduction should have given an idea of what RTM's are, how they operate, and how they are used. It is highly recommended if the reader is planning on physically using these modules, that he first construct the above system in order to become familiar with the modules, conventions, etc. This would be subject to the reader's own prewired Bus conventions. Having gone this far, we are now in a position systematically to describe RTM's in detail. This is done in the next major section of this chapter.

### ■ RTM PRIMITIVES AND THEIR BEHAVIOR

## THE RTM SYSTEM

## Introduction

The RTM system is a set of about 35 different modules, plus a method of interconnecting modules via a common Bus that transfers data from module to module. This Bus also carries timing signals to interlock the register transfers -- thereby making the system timing independent. For a given system, some of the modules connect to the Bus in order to transfer data; this combination constitutes

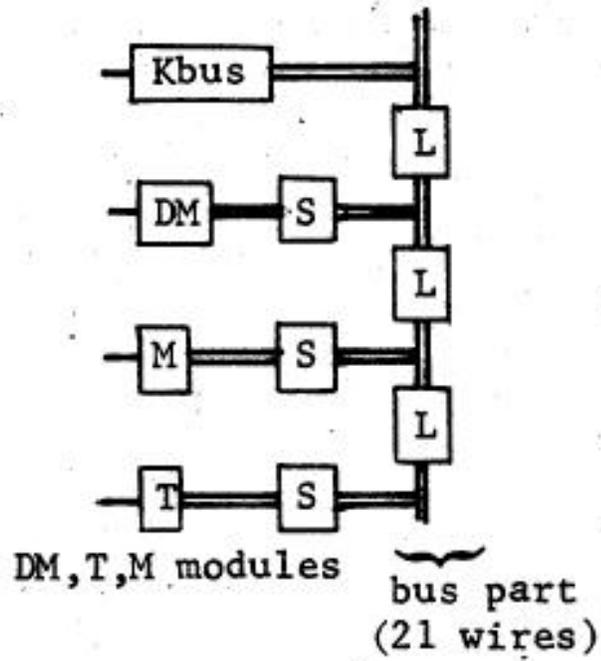


Fig. 3. PMS diagram (expanded) of RTM Bus.

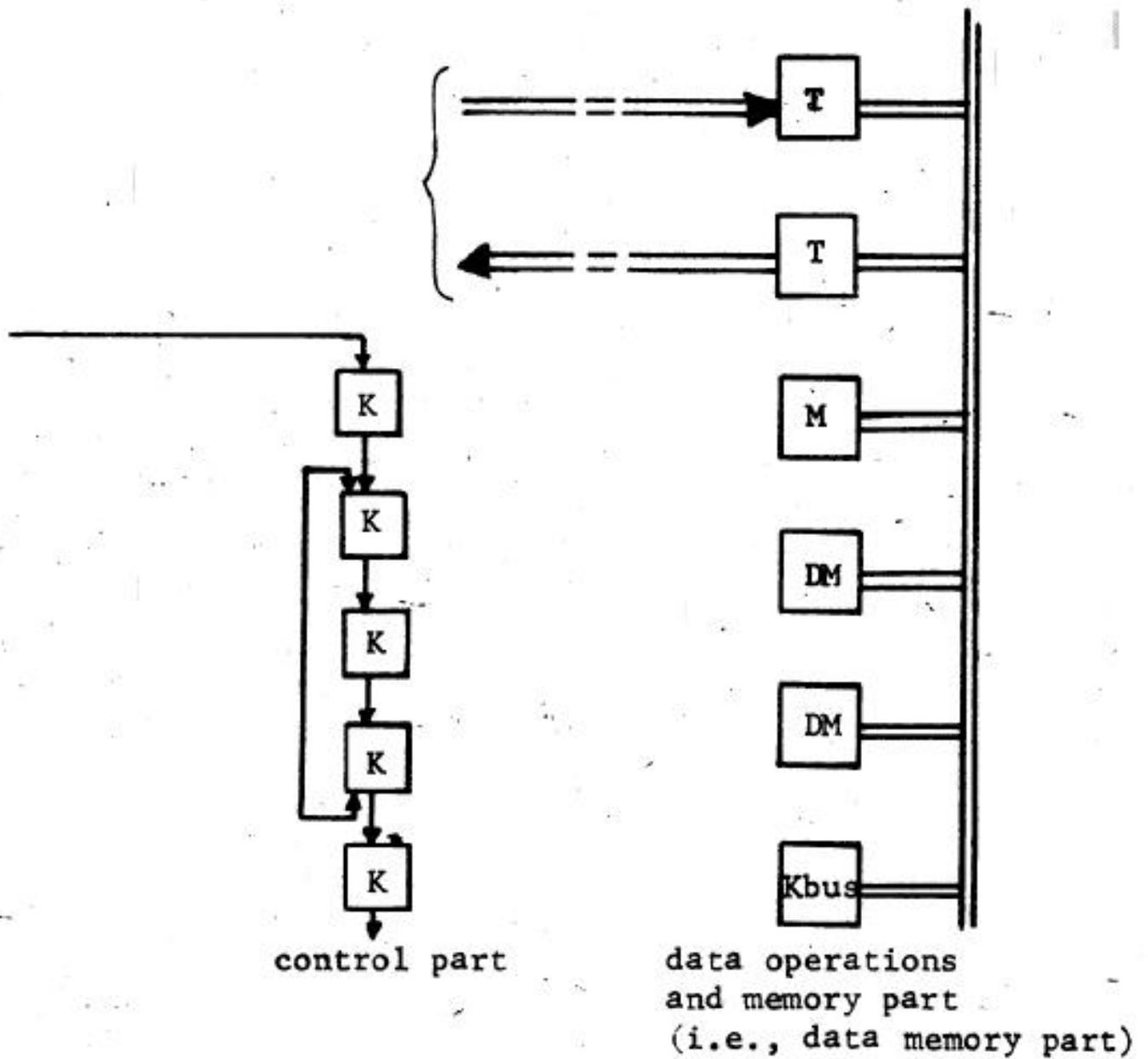


Fig. 4. PMS diagram of an RTM system without evoke-operation, operation complete, and Boolean data inks.

propagate control. via its "activate-next" output port: Information about the condition of the DM part (e.g., whether a certain memory location is zero) is also led back to the control part via Boolean data links (----) to affect the operation control sequence.

The data-memory portion of the system provides several functions: storing of data in memory (M) components; transmission of data to and from the outside world via transducer (T) modules, -- for example to or from analog, high voltage, low current or manual links (the links are shown as — -, — -); execution of data operations on various data cells via the RTM Bus \Bus, shown as heavy parallel lines (i.e., === ). Another module on the RTM Bus, the K(bus sense)\Kbus, controls the flow and timing of information and serves several other functions. While the bus is a common entity in digital systems engineering, it is not a primitive component, and the bus of the previous figure is more correctly shown in Figure 3. The RTM Bus is actually a set of 21 parallel links (wires) to which the M, DM, and T modules connect. Each module has a switch which allows it to be connected or not with the links. The actual function of each wire will be described later. For brevity, we shall normally present the Bus as in Figure 2.

In most diagrams only the K components connected by the control flow links will be given, together with the DM, M, and T components. The interconnection between the control and data-memory parts will be implied by the control part. Removing the interconnecting links, Figure 2 becomes Figure 4.

System design with the modules can best be illustrated with a simple example. Consider the algorithm for summing the positive integers from 1 to N, shown in Figure 5. Note that the computation is actually taken from N to 1, where  $N \geq 1$ . The diagram of an RTM system that implements this algorithm is shown in Figure 6. The design procedure for obtaining this design result from the algorithm is quite straightforward. First an appropriate set of data-memory (DM) modules from the possible set is selected, and each module is connected to the RTM Bus. For this problem, a T(swatches) module allows an external environment (a human) to specify the value N, and a DM(general purpose arithmetic unit) holds a counter, I, of the index to N, and the sum, S. The Bus requires a Kbus module attached to it, for sensing and controlling the Bus. Next the flowchart of the control algorithm is directly mapped into a network of control (K) modules. Then the wires implied by the algorithm statements and shown as dashed lines in the figure are run from the control part to the data and memory part to evoke the appropriate operations. This completes the RT level design.

The next step is the physical implementation of the system. It is usually the practice to use prewired panels, upon which the power, RTM Bus, and operation completion lines are already wired in some standard format. The only wires that have to be added to this panel are the control wires shown as solid and dashed lines in Figure 6. When the wiring has been completed, the DM, T, and K modules plug into the panel, and the panel is connected to an appropriate mounting rack, which contains the power supply.

Figure 7a gives the complete parts list for this example, plus the module locations on the mounting panel.

Figure 7b shows the RTM system diagram with complete information on pin numbers for wiring. In this figure we have superimposed the block diagram notation used in this chapter on the standard DEC notation for PDP-16's. This is for the benefit of the reader who is using the PDP-16 handbook. The only other place in this book that we refer to the DEC notation is in Chapter 7. Note that Figure 7b contains the complete design documentation of the system. However, if one wishes a more systematic wiring list, one can be derived, as shown in Figure 7c. Figures 8a, b, and c show photographs of various parts of the system.

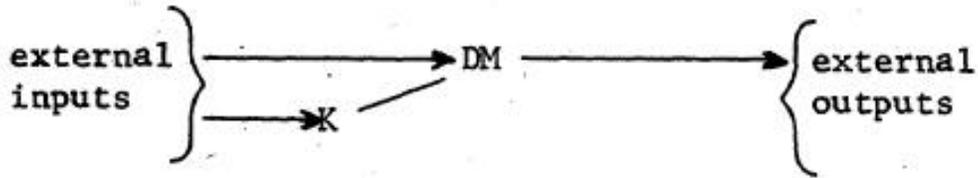
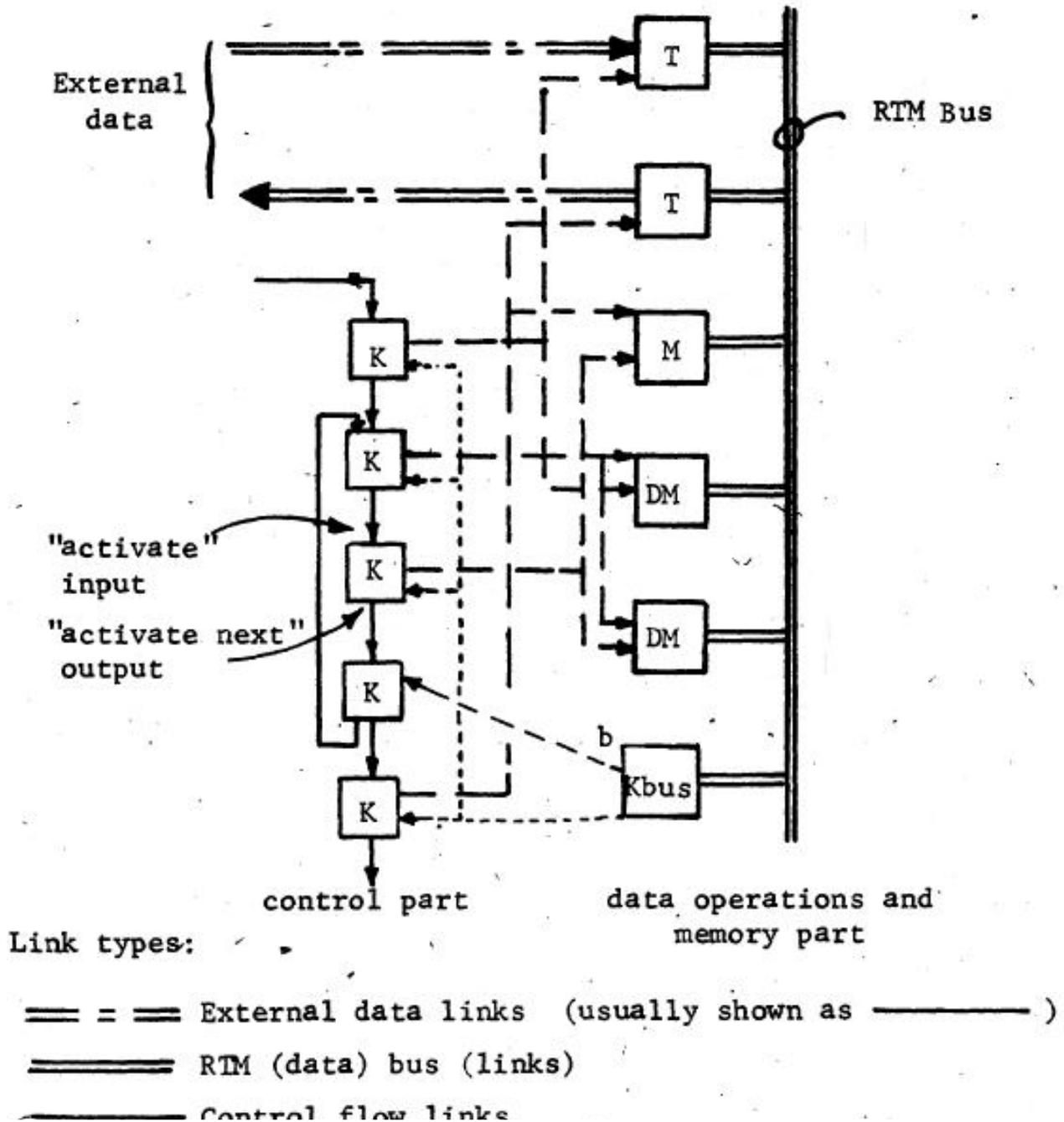


Fig. 1. PMS diagram of a typical RTM system.



==== KIM (data) BUS (links)

———— Control flow links

- - - - Boolean data links (usually shown as —————>)

- - - - - Operation complete links\Bus DONE (usually not shown)

- - - Evoke operation links

Fig. 2. PMS diagram of an RTM system showing various link and component types.

## ■ CHAPTER 2 PDP-16:

### A SYSTEM OF RT-LEVEL MODULES (RTM's)

This chapter presents a specific set of RT-level modules, called Register Transfer Modules (RTM's).<sup>(1)</sup> They will be used throughout the book as the given components out of which designs are to be constructed, except in Chapter 7, where we discuss the details of their construction, and in Chapter 8, where we discuss general RT-level design. The RTM components are the DEC PDP-16 modules, augmented by a few DEC modules that are not included formally in the DEC PDP-16 series. Throughout, we use the PMS notation for these modules, rather than the DEC names, believing the PMS names to be somewhat more descriptive functionally, hence making design easier. However, for some details of PDP-16's and their accessories, we shall refer the reader from time to time to the, "PDP-16 Computer Designer's Handbook" (DEC, 1971), or simply, to the PDP-16 handbook

This chapter differs from all those that follow in being a listing of information and specifications, rather than a series of design problems to be performed. It is a chapter that is to be used as a reference throughout the rest of the book and it should not be read through as if each of its pages should be understood in toto. We have organized it in several parts. The first introduces RTM's and describes a complete example system, carrying it all the way to photographs of the final constructed physical system. The second, gives the specifications of all modules, first a set of the four most important ones, and then description of the rest. To keep this part self contained as a reference, it is intentionally a little redundant with the preceding part that gives the example; The third part describes a major module, K(PCS), that provides a microprogrammed control. The final part describes two new modules that became available too late to be used in the designs of the book.

### A FIRST LOOK AT RTM'S

A typical system implemented with RTM's, having external inputs and outputs, is shown in Figure 1 in PMS notation. This embodies a control portion (K) and a data-memory portion (DM). At a more detailed level an RTM system usually has the form shown in Figure 2. Figure 2 shows that the control portion of the system is in fact a network of K modules which is connected to the data-memory portion of the system by another network of three types of links (i.e., -- evoke-operation, ....operation-complete, and ---- Boolean data). The control network is isomorphic to the flowchart of the control algorithm for the system. Each K module carries out one step of the algorithm by sending control signals to the data-memory portion of the system to evoke the desired operation(s), -- for example, evoking the transfer of data from register X to Y, expressed as  $Y \leftarrow X$ . As each step is completed, control is passed on to the next K module in the network via control flow links (—)

Control flow enters a K module via an input port called "activate" which activates it. The K module

requests that an operation be carried out in the data- memory part by using the evoke-operation link (--). As an operation is carried out in the DM part, it informs the control part of the completion so the next step can proceed. The operation-complete signal is used for this purpose (i.e., ...) The operation complete signal causes the currently active K module to

-----

1. We use the backslash (\) to separate a name from its accepted alias.

the addition of the assignment arrow. Thus in the figure we write that  $i \leftarrow i+1$ , meaning that the value of the variable  $i$  takes on a value equal to its current value, plus 1. If we were to use 'the equality sign,  $i = i + 1$ , this might be confused with the statement that the variable  $i$  is equal to itself plus 1 (which is universally false). Some programming languages make use of the equality sign in this way but it is preferable to use a separate notation. Thus, whenever the equality sign occurs it implies an assertion of the equality of two quantities (e.g., in the diamond in the figure where  $i = N+1$  asks if  $i$  and  $N+1$  have the same value).

The language of flowcharts is adequate for most of what we do in this book. However, it is not adequate for all tasks of specification. For instance, it is not adequate to specify a desired computer, prior to constructing an RT system that would realize the computer. For this one wants a language for describing computer instruction sets. Flowcharts have in fact been used for this on occasion but they are usually a bit awkward. Special languages have been proposed and in Chapter 6, which discusses the design of small computers, we will use one called ISP (for Instruction Set Processor), giving there as much of the notation as we need. The language was introduced in Bell and Newell (1971) where complete details can be found. Our reason for mentioning it here is to emphasize that there are many specification languages, adapted to the demands of different task environments.

## SUMMARY

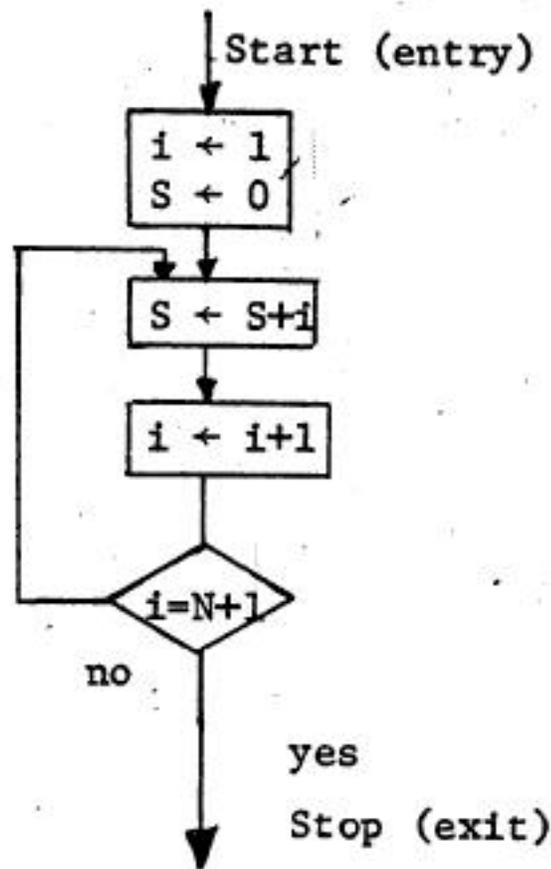
We have now provided a framework within which to proceed with the design of RT systems. We introduced digital systems, both what they are good for and what they are constructed of. We established that we would operate at the register-transfer level, distinguishing this from the many other levels at which digital systems could be considered. We posed the design task in general terms, just so you would be clear on what you were being asked to do. Then, we introduced the languages for the two main boundaries of the design task: the components out of which RT systems are to be built and the specifications for desired behavior.

There is nothing left to do now except start designing. Chapter 2 gives the full specifications of the PDP-16 modules, and from there you can begin.

■ 27

example, Chapin, 1971). However, we need only a few basic symbols. A rectangle specifies one or more actions, with the expression in the box describing the results of the actions in terms of the data on which they depend. A diamond specifies that a decision is to be made among one or more alternatives, depending on the value of the computation stated- in the box (we use the term box indifferently for rectangles and diamonds). The lines show the control flow, so that if one takes the action at a box then one follows the arrow leading out of the box to determine what action to take next. Several control lines lead out from a diamond, corresponding to the different decisions that can be made, and they are labeled accordingly. Several control lines may impinge on a single box or a single line since there is no reason why one should not arrive at an action (or decision) from many separate places.

The important thing to realize about flowcharts (assuming you do not already know all about them from programming) is that they express a single sequential scheme of computation, in which only one thing happens at a time (i.e., only a single box is active at a time) and where there is no memory at all of what particular sequence of activities in the past lead to the current state. One can follow the method shown in a flowchart by putting a button or similar token down at the active place (starting at the place marked entry or start) and looking only at the box containing the button and the lines that emanate from it.



**Fig. 9. Flowchart to sum integers from 1 to N.**

The flowchart only expresses the flow of control. The actions taken in the rectangles make changes in the data of the problem. (No changes in data ever occur in a diamond.) It is assumed that the current values of these variables are used in each box. Thus, values of the variables can be set in some rectangle, and then the flow of control made dependent on them via the expression in a diamond somewhere else. In Figure 9 this occurs when the variable  $i$  is finally increased until its value is  $N+1$ .

The language we use inside the boxes of a flowchart is standard algebra, with

processing. We want to connect to the external environment (to interface) at particular points and in a manner that is externally dictated, at least in part. We want to take from that external world various information, including commands, and to deliver various other information, including acts of control. We need to be able to state these things, without already giving the design of the system.

There is no universal natural language for stating system behavior. There cannot be, in fact, since this behavior is defined in terms of two things, each almost totally variable: (1) the subject matter that the information processing about; and (2) the form in which the specifications reside in the heads of those who want the jobs done (be they the designers themselves or some extern authorities or customers).

The important thing about a specification of behavior is that it should state what is wanted, without prejudicing the design any more than necessary. Thus, statements of behavior desired should be as much as possible in terms of the ultimate significance, which is to say, in terms of the intended subject matter interpretation of the information processed. Similarly, the ultimate specifiers the system have their knowledge in a particular way (usually in terms of subject matter concepts). They should make these specifications as definite and detailed as possible, but they should never be pressed to where they begin making design decisions about the digital system. Such decisions then masquerade as specifications, and freedom to create appropriate designs is lost.

This is not an idle worry. For many areas, especially non-mathematical one precise languages are not available for describing the jobs to be done. The language of information processing, as developed for digital systems, is the be language in which to say what one wants.. Consequently, the line between behavior specification and system design easily becomes blurred.

To specify desired behavior we will generally use the language of flowchart which is a language of sequential procedures. For example, suppose we wish construct a system to sum up the integers from 1 to N. We could express this, the first instance, by stating:

$$\text{Compute } S = \sum_{i=1}^{i=N} i$$

This is adequately precise. It does not, however, tell us how we are to compute it, that is, by what method. In the first instance, this is exact[y what is desired, since one does not want to prejudice the computation in any way. If one knows just a little mathematics, one knows that:

$$S = N*(N + 1)/2$$

One wishes the option of basing the processing on the latter formula as well as on the first one.

Settling on a method, either one of the above or some other, is only a first step toward the realization of an RT system that will compute  $S$ . All of the problems of the processing of data are still to be faced -- just as we saw them come to the fore in our miniature example of adding two numbers. We need a language to express the essential characteristics of the method (or algorithm, as it is also called). Such an expression will actually play the role of specifications for the design of the hardware system. Flowcharts are a useful language here, backed up by a language (such as algebra) used to designate the relations and functions of a substantive area.

Figure 9 shows flowcharts for a method of determining the sum of the first  $N$  integers. The conventions used in the figure are taken from those standard-in the field. Flowchart symbology is actually rather highly developed (see, for

the world to a few simple numbers. The solution is to understand what systems objectives are : they are guides to understanding and assessing system behavior in various partial aspects. Various measures for each type of objective are developed, and each shows something useful. Since all measures are partial and approximate (even conceptually), rough and ready measures that are easy to make, display and understand are often to be preferred to more exact and complex measures. Standard measures are to be developed and used, even If not perfect. Experience with how a measure behaves on many systems is often to be preferred to a better, but unique, measure with which no experience exists.

There is no need to treat systematically all the different system measures. We will illustrate a large number of them through the design examples we treat in Chapter 3. However, Figure 8 does provide a guideline, listing in one place many standard objectives on systems, cost, performance and structure, and giving brief definitions of the standard measures that are used for them.

## Cost components

designing

specifying

designing

prototyping

describing

designing the production system

producing

buying (parts)

assembling

testing

selling and distributing

using

understanding

purchasing

applying

operating environment (heat, humidity, vibration, color, power, space)

repairing

remodeling

## Performance

designing, producing, selling

using

environment

for a single task (operation times, operation rate, memory size, utilization)

for a set of tasks

reliability and error rate

mean time between failures (mtbf)

mean time to repair (mttr)

error rate (detected, undetected)

**Fig. 8. Cost and performance components for a system.**

## INPUT-OUTPUT BEHAVIOR

We want to design digital systems to do specific jobs of information

[previous](#) | [contents](#) | [next](#)

other systems, different from the one in Figure 6, that also did the same job. We might have provided two transducers and no memory, each number remaining available at its transducer until ready to be used by the adder. We might have provided for both numbers to have gone into the memory, insulating the adder entirely from the input 'process. With such a simple task there are only a limited number of non-foolish ways to do it. Still, there are several -- enough to show that the statement of the task to be done by the system is not to be confused with the description of the system that does the job.

The desired characteristics of a system are specified in two different ways: (1) by overall objectives on the system behavior and structure; and (2) by detailed specifications on the information processing to be accomplished. Both are important and are used jointly to state what is desired. We take up each in turn.

## OBJECTIVES AND EVALUATIONS

Objectives can often be stated as maximizing or minimizing some measure on a system. A system should be as reliable as possible, as cheap as possible, as small as possible, as fast as possible, as general as possible, as simple as possible, as easy to construct and debug as possible, as easy to maintain as possible -- and so on, if there are any system virtues that we have left out.

There are two deficiencies with such an enumeration. First, one cannot, in general, maximize all these aspects at once. The fastest system is not the cheapest system. Neither is the most reliable. The most general system is not the simplest. The easiest to construct is not the smallest, and so on. Thus; the objectives for a system must be traded off against each other. More of one is less of another and one must decide which of all these desirables one wasn't most and to what degree.(11)

The second deficiency is that each of these objectives is not so objective as it rooks. Each must be measured, and for complex systems there is no single satisfactory measurement. Even for something as standardized as costs there are difficulties. Is it the cost of the materials -- the components? Do we use a listed retail cost or a negotiated cost based on volume order? What about the cost of assembly? And should this be measured for the first item to be built, or for subsequent items if there are to be several? What about the costs of design? That is particularly tricky, since the act of designing to minimize costs itself costs money. What about cost measured in the time to produce the equipment? What about the cost of revising the design if it isn't right; this is a cost that may or may not occur. How do we assign overhead or indirect costs? And so on. In a completely particular situation one can imagine an omniscient designer knowing exactly which of these costs count and being able to put dollar figures on each to reduce them all to a common denominator. In fact, none of us knows that much about the world we live in and what we care about.

The dilemma is real: there is no reducing the evaluation of performance in

11. Belief in the perversity of nature that forces all good things to trade off against each other should not be carried to extremes. The field of digital systems provides a counter-example. Digital systems now, compared to digital systems even a few years ago, are: faster, cheaper, smaller, more reliable, simpler, and easier to maintain. In short, they are better in every way, and nothing had to be traded off in the final performing system to obtain them. Within this, there always exist small trade offs, e.g., between speed and cost. But this is barely significant against the major trend.

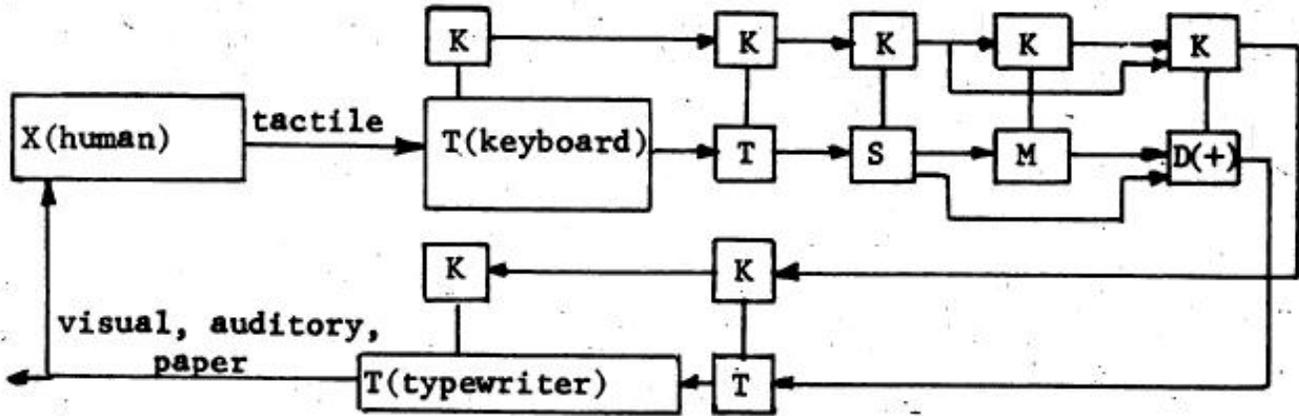


Fig. 6. PMS diagram of a calculator with typewriter output.

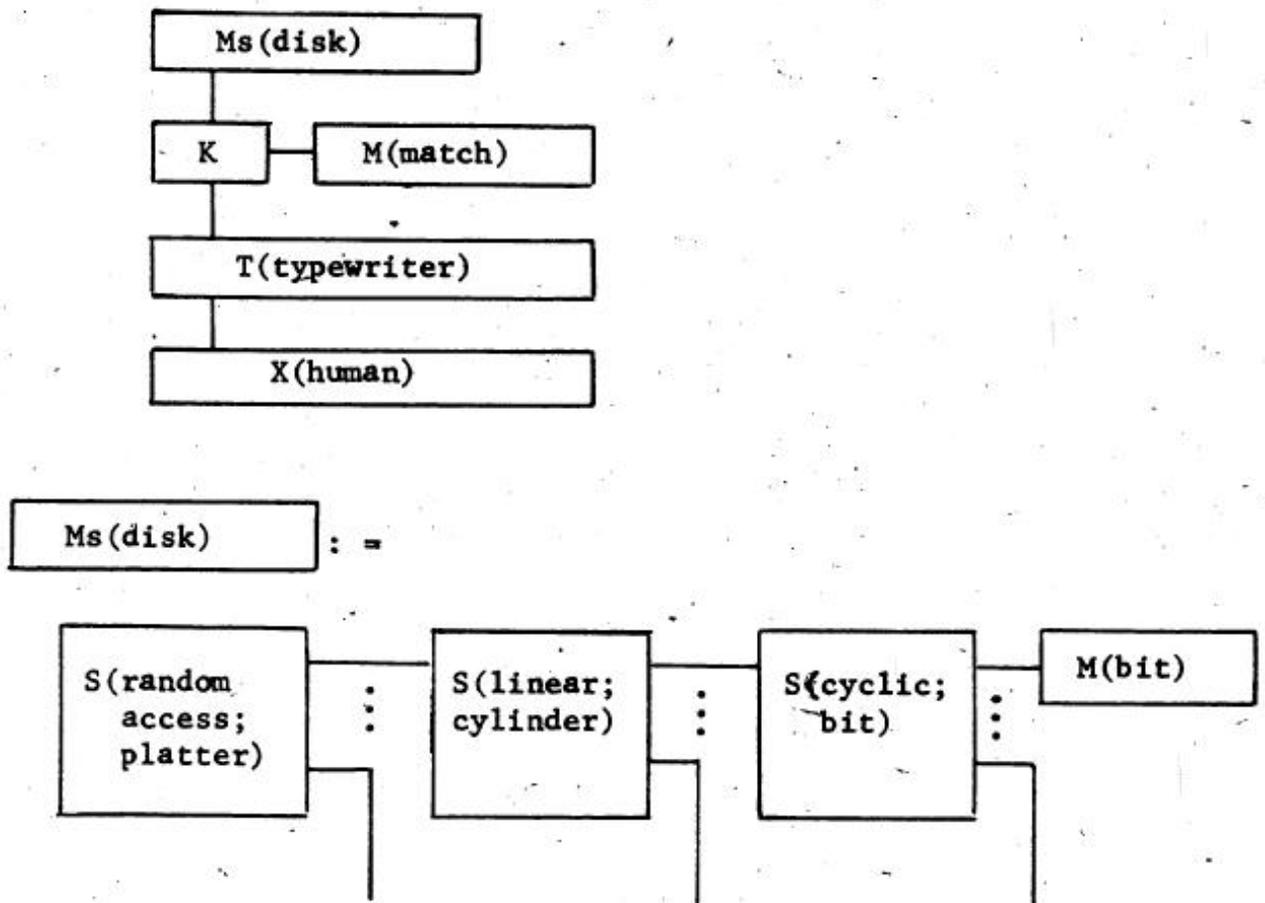


Fig. 7. PMS diagram of information retrieval system.

[previous](#) | [contents](#) | [next](#)

With the exception of the link itself, two components are in general physically distinct and thus their ports must be connected by a link. A perfect link would make this connection into an identity. In fact, there is always propagation delay so that the bit values at the port of the one component do not instantly assume the values of the corresponding bits of the other port to which it is connected. In modern digital technology these delays are often significant in the performance of the system. In high performance designs they always effect the design in detail. In much other design they can simply be lumped in with the speed of the components themselves. Though they degrade the overall performance somewhat, they do not effect the design. When this is true, as it will be throughout this book, the link itself is no longer a significant separate component in the system. It can be replaced by a simple connecting line that identifies two ports. It does, of course, still have a capacity of a certain number of bits. It also has directionality, permitting bits to be transmitted either just in one direction (called a simplex link), in both directions at the same time (called a duplex link), or in either direction, but only in one direction at a time (called a half-duplex link).

With the role of links cleared up, we can now show concretely how to connect digital systems together. Figure 6 shows the illustrative system for simple addition used earlier. We have made a box for each component, labeling it with the appropriate abbreviation. The boxes are a matter of style only, and we will often dispense with them. We have shown the direction of data flow in the links but omitted the data flow capacities. Links between controls and the components they control are invariably two way (at least half-duplex) since the controls both evoke their components and sense when they have completed operation.

Figure 7 shows a simple system for storing away data and then retrieving it. The component called X stands for the component outside the digital system in question, say a human at a keyboard or whatever. We often do not want to bother stating exactly what the nature of the external component is. Below the memory system we put in a blow-up of the M component itself, showing it as another system described in the same functional terms.

Both figures are still quite general, with the components labeled only by the gross functional category to which they belong. Before we can show the full detail, we need to know how to specify system behavior.

## SPECIFYING SYSTEM BEHAVIOR

To design one must specify what one wants of the to-be-constructed system. Recall that all the formulations of the design task had both a part that gave the technology and a part that gave the desired specifications. These specifications must be given in independent terms. Suppose, for the system of Figure 6, we had just said "We want a system that takes data from the keyboard, transduces it, ships it to a memory, transduces a second number, ships it and the number in the memory to the data operation,...(and so on through the details of the figure)." How would you ever know whether it did the task that was wanted? It simply is what is and there is no independent statement against which to

evaluate it. Worse yet, how would it ever have been designed in the first place? How would one have known what to write down in the diagram before it was written down?

In fact, of course, we did have an independent specification. We wanted to add two numbers given by a human and present the sum to him. But isn't this the same thing? Not at all. We could show it was not the same by exhibiting

21.

[previous](#) | [contents](#) | [next](#)

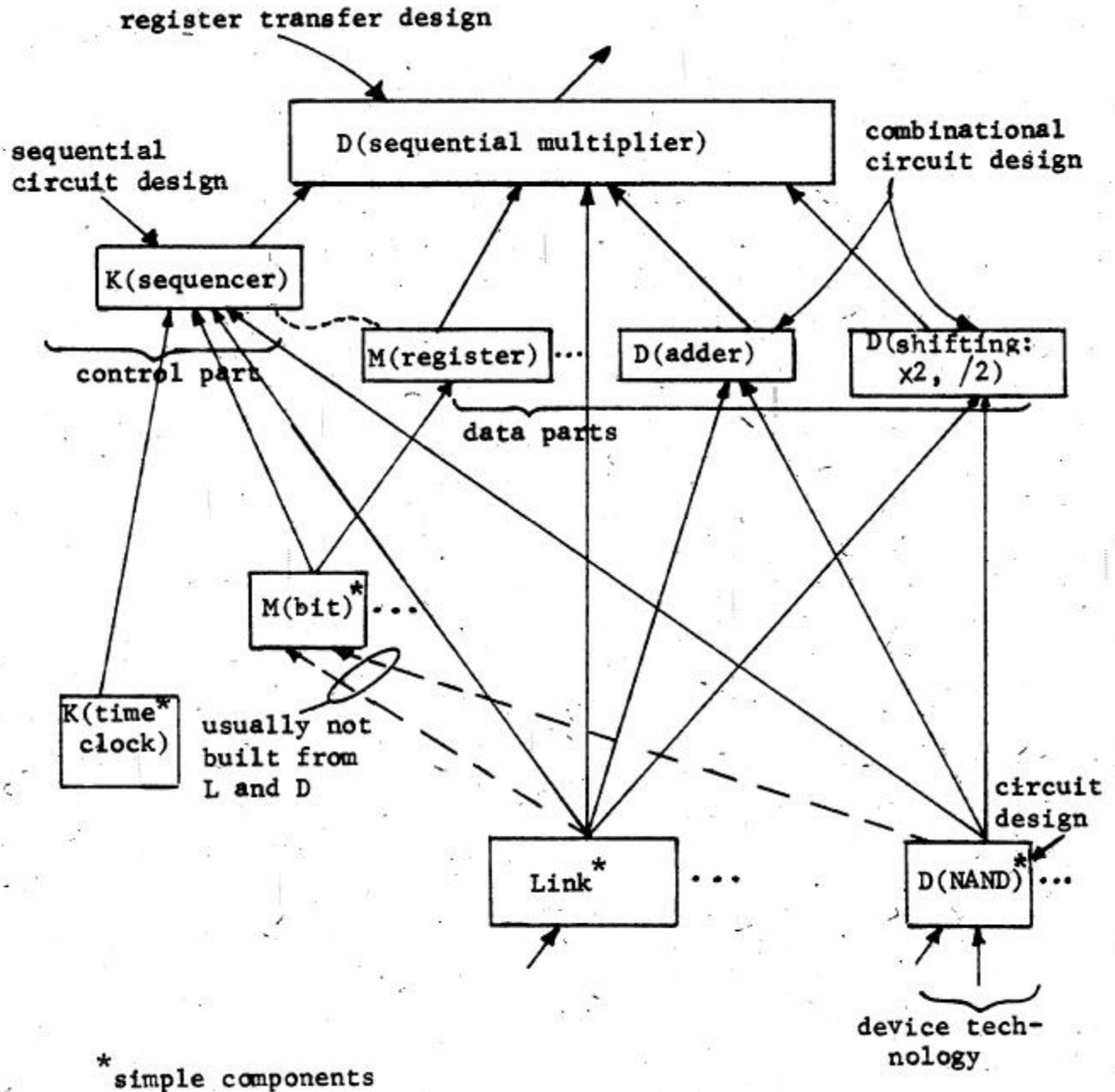


Fig. 5. Hierarchy of components used in logical design of a sequential multiplier.

structural scheme of description, we specify seven exact operations and then say everything can be built by combining them, here we specify seven functions and say everything can be decomposed into systems whose components are also only of these seven types.

## HOW TO CONNECT COMPONENTS INTO SYSTEMS

Each component, of whatever type, has a number of ports. Each port permits the transmission, in or out (sometimes both) of a set of bits that play some role in the operation of the component. To construct a system one connects the ports of one component with the ports of another. Conceptually, they become identical: two connected ports always have identical bit values.

16 set supplied by DEC, that will have memories of particular sizes, controls of particular character, various specific transducers, etc.

The second, and more important, reason why it is not necessary to catalogue all possible parameters and subtypes of each component type is that a scheme of functional description enjoys the important property of being recursive. Take any component (say a memory), however complex, and open it up. It will consist of a combination of subcomponents that can also be described in these same functional terms. A big, memory consists of a collection of smaller memories connected together by links and switches and made to operate by controls; if its internal representation (say as magnetic patches on a drum) is different from the external one, then there will be transducers both going into the internal form and coming out of it. The addressing will be accomplished by a switch (that, in fact, is what addressing is), and so on. A similar story will hold if you open up a complex control (say a disk controller), or a complex data operation (such as a multiplier), or a complex transducer (such as a line printer).

All the variety of each type of component is to be defined by all the different ways in which systems can be defined (in the same terms) that perform the function. Somewhere down at the bottom of such a decomposition -- of continually opening up each component and finding it to be an assemblage of subcomponents of the same type -- there must be some primitive components. We can consider these to be simple components: simple memories, simple transducers, simple links, etc. These simple components, which cannot be decomposed further conceptually, are all defined on single bits. They correspond to the operations available at the lowest logic level of digital systems: the primitives of combinational and sequential circuits.

Figure 5 shows how a sequential multiplier at the RT-level is hierarchically formed as combinations of K's, M(register)'s, L's, and D's. In Chapter 8, we have the design of such a multiplier; also Chapter 4 describes multipliers formed with our particular RT primitives (RTM's). Notice that there are several distinct design activities: the RT- level, sequential circuits (the K), combinational circuit (the D's), and electronic circuit (the NAND's and possibly the 1-bit M's). With respect to data operations there is no single unique set of primitives. We have given one that involves a single Boolean connective NAND.(10)

The exercise provided in Figure 5 is entirely for conceptual clarity. It once was the case that the physical components provided to a logic-level designer were simple components such as the NAND and he always had the task of specifying all systems made from these elements, including compound components that he might put together as intermediate subsystems in a design (such as a register consisting of a specified number of bits, which would be used throughout .a design). This is no longer true. What the manufacture provides is a collection of compound components, which then form the actual primitives that terminate the decomposition of a system into subsystems. Now the primitives are registers, adders, and shift networks, but the control part, the most difficult, is still left to the logical designer. Only rarely (as when one is designing a set of new primitives) does one design entirely from

simple components.

Again we have exhibited the peculiar feature of digital systems that they elude completely specific description. We have just been definite in one more respect: there are exactly seven types of components. But whereas in a

-----

10. NAND(x,y) is defined to be NOT(AND(x,y)) and all logical combinations can be defined in its terms, e.g., NOT(x) = NAND(x,x); OR(x,y) = NAND(NAND(x,x), NAND(y,y)), etc.

Occasionally one names by the structure of a system. For example in electrical circuit theory circuits such as Tee and Pi are so named because the structure resembles a T and  $\sim$ (Pi).

Given that you know a component is a memory (M), you know something about the behavior it can exhibit, but it is still mostly unspecified. You don't know how much memory (in total bits), how it is organized in terms of an addressing scheme (e.g., does it hold 1000 16-bit words or 2000 8-bit words). You don't know how fast it reads information (i.e., retrieves it) given an address, nor whether the time to access information is independent of the address (as in a random access memory) or linearly dependent (as in a magnetic tape). You don't know the details of how to control it. You also don't know other, more remote things: its reliability, its size, the effect of temperature, vibration, humidity, and so on. In short, all you know is that it is a device that will store information over time.

That is still a lot to know, since a link, transducer, control, or any other basic type of component cannot store information.(8) Thus, the labels are extraordinarily useful in design, since whenever information must be stored over time one knows an M must be involved. Similar statements hold for all the other types of components. They divide all the tasks to be done in a digital system into a set of exclusive categories -- the basic functions of L, T, M, D, K and S -- and whenever such a type of job must be done, it is known what kind of component must be used. Thus, they help to bridge the gap between the structure of the physical components and the behavior desired.

What we indicated for memories above is true of all the other component types. There is a great diversity of specific varieties of each component type. We could describe in a formal way all the various parameters and subtypes involved in each component type -- all the types of memories, of switches, of transducers, etc.(9) These further specifications are to make the component fit all the other conditions of the design: to have enough memory, to switch to all the required places, and so on. There are two important reasons why we do not have to provide all this detail here in a big catalogue. The first is that any particular digital technology (as supplied by a manufacturer) offers a finite set of basic components of each of these types (or special mixtures of them). All other systems must be built up by connecting these together. Thus a system's parameters become those that arise through combination of the basic set. For instance, Chapter 2 will present a set of RT-level modules, constituting the PDP-

-----

instance, knowing that a component is a resistor of 100 ohms does not tell anything about the modification of its behavior under high temperatures, or the point at which the component will disintegrate under vibration, or any of innumerable other aspects of its behavior that are relevant to its operation in a circuit under some circumstances.

8. Again, we must be careful. A link has a certain transmission delay and thus does have a small amount

of memory associated with it (much as a physical device designed to be a resistor has a little inductance also). Thus, a link could be used as a memory in a digital system. But to do so is strictly poor design or (in rare cases) exotic design. It can be ignored to a first approximation. Of course, a memory can be deliberately constructed out of links with long delays (that therefore act as low performance links if used just to transmit data). So-called delay line memories are just such devices.

9. See the Appendix of Bell and Newell (1971) where this is provided with some attempt at completeness.

L \Link Transmits data from one place to another without modification.

T \Transducer Changes the representation of data, either from a non-digital representation (external to the digital system) or from one bit representation to to another (within the system). Does not modify the information content.

M \Memory Stores data over time without modification.

D \Data operation Produces new information (in some bit representation) from a set of input data.

K \Control Evokes the operation of some component in a digital system in response to various input conditions and its current status.

S \Switch Changes the links that connect other components in a digital system, so as to reroute data.

P \Processor A digital system consisting of a set of operating components (D's, M's, K's, T's, S's, and L's) with a control(called the interpreter) that both reads an instruction to determine what operations to perform, and determines what next instruction to obtain, thus running autonomously from a memory that holds a program of instructions (called the primary memory, Mp).

C \Computer A digital system that includes at least one processor and its primary memory.

Fig. 4. PMS primitive component types.

We label components by the general functions they perform. This is to be contrasted with schemes that label components by a specific function, in which the exact behavior of the component can be determined from its label. Labelling by specific function occurs in the combinational and sequential logic level, where the components are labelled AND, OR, NOT, NAND, NOR, DELAY, etc. An AND - component takes two binary inputs and produces a binary output that is the logical AND of the inputs:

$$\text{AND}(0,0) = 0$$

$$\text{AND}(0,1) = 0$$

$$\text{AND}(1,0) = 0$$

$$\text{AND}(1,1) = 1$$

The output behavior of an AND component is completely defined as a function of its inputs. Similar definitions apply to the other components.

Another example of this labelling occurs in elementary circuit theory where components are called resistors, inductances and capacitances. There is no single resistor, but rather a family of them, each defined by a single numerical value (e.g., 25 ohms, 75 ohms, etc.). To know that a component is a

resistor of a specific value tells almost everything necessary to compute its behavior in a circuit.(7)

-----  
7. All such schemes are only approximations of actual physical systems. This should not be forgotten, even while making use of the simplified models. For

the transducer, link and memory control themselves? And nothing has been solved anyway, since who controls the controller? These questions are not quibbles or philosophical conundrums, but correspond to real design issues and options. Control can be associated with other components. If you look inside a control component, you may indeed find a subcomponent that controls it. Also, controls control each other. And the data itself may be used on occasion to effect control. But in all cases the question of control must be explicitly formulated and answered in specific terms. A major source of the power and generality of digital systems comes from making most components passive and inert -- memories, transducer, links, etc. -- and localizing control in a physically and logically separate component.

With the components introduced so far, the first number can be transduced and transmitted to memory, and the second number can be transduced (when typed by the human). The two bit representations are now to be added.. Addition is an operation on bit sequences that generates a genuinely new item of information -- the bit representation of a sum. In this case it should produce:  $110\ 0\ 1 + 1\ 1\ 00 = 1\ 00\ 1\ 0\ 1$ , which is the bit representation of  $25 + 12 = 37$ . The component that generates such new information is called a data operation and is abbreviated by D. More specifically we would use an adder here, which is simply a name given to a special subvariety of data operations.

To get the addition actually done requires some more links and some more control. The first number must be read out of the memory and transmitted to the data component (the adder). The second number must be transmitted from the keyboard to the data component. Here we have another issue: we set up the system to transmit the bit representation at the keyboard transducer to the memory; now we want it to go somewhere else. This requires another component, called a switch and abbreviated S, to reroute the data over a different link. This switch must also be controlled, so that on the first occasion it switches the data to the memory and on the second it switches it to the adder.

Now we have all the types of components necessary to finish the job. The results of the adder must be sent over a link to a transducer that converts a bit sequence to a human-readable display, say a sequence of type impacts on a typewriter. This total operation must be controlled. Possibly, also, the memory must be cleared and the switch at the keyboard transducer must be reset to send the next data to the memory again, so that the whole system can perform a new addition. These operations use additional components (such as controls and links) of the same types as have already been introduced.

The types of components we have just given are not only sufficient for our illustrative task, they are sufficient for the construction of all digital systems. They constitute our basic set of components. Figure 4 provides a summary. We have added two additional components at the bottom of the figure: a processor, abbreviated P, and a computer abbreviated C. Both of these stand for complex systems. They complete the basic set of abbreviations defined in this notation.(6)

6. The notation is also used to describe digital systems at the much higher level of equipment configurations. The notation itself is called the PMS notation (for processors, memories and switches, three of the important components). Its use in other contexts is described in Bell and Newell (197 1).

[previous](#) | [contents](#) | [next](#)

3. There may be different ways to represent information in bit sequences; which way is used may be important since what processing must be done depends on the bit representation and not just on the information.
4. Digital systems need only process bit sequences in the limited ways possible with 0's and 1's; but thereby they are able to effect any information processing.
5. Conventional encodings of information, called data types, exist (especially for numbers and alphabetic characters) and the operations of digital systems assume these data types in their processing.
6. Any physical structure that can take on binary values can represent bits and participate in a digital system. The actual physical character of the bit representations is safely buried inside the technology.

## WHAT COMPONENTS DO THE PROCESSING: L, T, M, D, K, S.

When a human thinks about a task of processing information he usually concentrates on the complex transformations to be done. If two numbers are to be added, the act of addition is the main thing. However, if you try to mechanize that task -- i.e., get a digital system to perform it -- then there are many other things that must be done besides just the addition. Let us list these things for even so simple a task as a single addition. As we do so, we will label them with the names used for them throughout the book

First, the two numbers to be added must be given to the system. These numbers are generated by the human who wants them added. He does not generate them as bit sequences, but he might use a keyboard in which he presses a certain key for the digit 1, another for 2, etc. For example, the human could press the sequence, key-2 key-5, to indicate that 25 was the first number to be added. Something must create from these key presses the bit sequence 1 1 0 0 1 (the standard binary bit encoding of the number 25). We call such a component a transducer, and abbreviate it by T.

To add two numbers the human must give them both -- that seems elementary. *Yet* it means that the keyboard must be used to input the second number (say 12) and the transducer used to convert it to a bit sequence (1 1 0 0). This must happen before the code for 25 can be used. Thus something must save this code (the 1 1 0 0 1) for later use. We call such a component a memory and abbreviate it by M.

Just having the memory is not enough, for the memory and the transducer are not in the same place and a bit-sequence at place 1 is not the same thing as a bit-sequence at place 2. Thus, there must be a component to transmit the data from the transducer to the memory. We call such a component a link and abbreviate it by L.

We do not yet have all the components necessary to get the 25 out of the way and into the memory. The transducer only developed the bit sequence (the 1 1 0 0 1) at a certain moment in time. Thus, the link should transmit it when and only when it is ready, and the memory should store it away when and only when it arrives at the memory end of the link. Thus, there must be a component that evokes all these operations at the right time and in the right sequence. We call such a component a control and abbreviate it by K.(5)

The need for control components may not be immediately obvious. Why don't

-----

5. We do not abbreviate it by C, since this character is used to stand for stored program computer.

rather than as the equivalent binary number representation. For example, the number 6235 would be represented in BCD as 01 100010001 10101 and in binary as 000110000101011. Needless to say, doing arithmetic in the two representations involves quite different bit processing..

The simplicity that comes to digital systems because they only have to process sequences of bits is incalculable.(4) Sequences of 0's and 1's are of sorts of things indeed and they can only be subjected to a small variety of fundamental transformations. All the rest comes out of compounding these. Although this compounding can become quite complex (indeed it has to, since ultimately any information processing, no matter how complex, can be done this way) the basic sorts of operations retain their simplicity.

We have described what is processed by digital systems in a very abstract way -- only as 0's and 1's -- and not at all as some specific physical structure. You will be forgiven if you conclude that we always promise to be concrete and specific and then manage to avoid it.. But the peculiarity lies in the nature of digital systems and digital technology, not in our writing habits. Any physical system that can be arranged to exhibit two possible states can be transformed into a digital technology out of which digital systems can be constructed. And in fact the physical phenomena actually used are of the most diverse kinds -- different in magnetic core memories, magnetic drums, transmission wires, transistors, storage tubes, and in all the subvarieties of solid state devices. All these can occur simultaneously within a single operating digital system, such as a large digital computer. At the lower levels of the digital system hierarchy, each of these technologies must be dealt with separately and requires its own scientific and engineering know-how. At the register transfer level all the differences have been smoothed over and all devices look alike in processing two abstract distinguishable states, called by convention 0 and 1. Differences still exist, but they all show up as parameters of speed, size, cost, reliability, etc.

It makes no sense to describe to you in detail, as some textbooks used to do, a particular lower level' technology (such as transistor logic), as if that made matters more concrete and real. The whole point is that 0's and 1's are real and one need not go below them. Someone has to go below the register transfer logic level, but that is simply a different design world with its own problems and its own rewards. We will say hardly anything more about such physical details.

We have dwelt on what a digital system processes, because it is Important to get fundamental matters straight. All that we have said in this section is probably known to you in some fashion. It is simply part of the accepted view of a digital system and does not appear as concrete rules, facts or principles about how to design digital systems. Yet it is important to understand each of these things. We summarize them below. If not so already, they should begin to appear to you to be commonplace and obvious:

1. Digital systems process information represented as sequences of bits.
2. Any definite information can be represented in sequences of bits.

4. Someone is sure to note that the term 'digital systems' should also cover systems that process discrete signals that have more than two values, e.g., ternary systems, which have three values, 0, 1 and -1. There is, of course, no issue of generality: binary systems can do all that ternary systems can do and vice versa. There would be some necessity to include such systems if (and only if) there was a ternary digital logic that was at all cost-effective. Categorically, there is little, chance in the immediate future for any other technology than a binary one. Consequently, digital systems can be restricted to binary systems.

types may be defined by the designer that are unique to a specific application. Figure 3 provides an outline of the basic ones. The basic data type is the word, which is a sequence of 16 bits. The PDP-16 system is built around this word size and thus all representations of information occur in its terms, either as multiples of 16 bits (if they take many words) or as various fractions of a word. The basic representation for a number is the 16-bit two's complement integer (details given in Chapter 3), which can encode the numbers from  $-2^{15}$  to  $2^{15}-1$ . The PDP-16 arithmetic modules take such integers as input and deliver them as output. Larger integers (say 32-bit integers) must be constructed out of 16-bit parts. Addresses into memories are also given as integers and form a conceptually distinct type. The size of such addresses depends on the memories used; we list 6, 8, and 10 bits, corresponding to memories of 64, 256 and 1024 words.

The second basic data type is the Boolean vector, which is simply the word treated as a set of bits. This can be broken down into fields of arbitrary size (less than 16 bits). The standard way to encode arbitrary, non-numerical information is to determine an upper bound to the variety (e.g., only five types of checker squares), then assign a field of the requisite number of bits to hold that variety (e.g., three bits to cover five types) and produce an arbitrary mapping of the possibilities into the bit sequences. Two special cases of this are the one-bit Boolean variable and the one-bit sign for signed integers.

word (16 bits)

two's complement integer (16 bits)

address integer (6, 8, 10 bits)

Boolean vector (16 bits)

field (any size  $\leq$  16 bits)

Boolean (1 bit)

integer sign (1 bit)

character string (arbitrary number of characters)

character (8 bits)

ASCII

## EBCDIC

binary coded decimal (4 bits) .

Fig. 3. Basic data types.

The third basic data type is the character string. All the alphabetic characters can be encoded. Unfortunately, due to historical reasons, there is more than one standard encoding of the alphabetic character set (thus destroying some of the gain to be obtained). The two standard ones (called EBCDIC and ASCII) are given in Table 5 (at the end of the book) for reference. They are both 8-bit codes, but with 7-bit (128 characters) subparts that cover most of the characters (capital and lower case letters, digits, basic punctuation marks, and some standard control signals). The eighth bit is reserved for a special purpose. Two 8-bit characters (also called bytes) fit into a single 16-bit word, but one almost always deals with sequences of many characters, corresponding to names, text, etc., thus requiring a sequence of words. Within the character code there is a 4-bit code for the 10 decimal digits, called binary coded decimal (BCD), which is just the first 10 binary digits (0000 to 1010). Sometimes decimal numbers are represented as sequences of BCD characters,,

1 R		2 R		3 R		4 R	
	5 R		6 R		7 R		8 R
9 R		10 R		11 R		12 R	
	13		14		15		16
17		18		19		20	
	21 W		22 W		23 W		24 W
25 W		26 W		27 W		28 W	
	29 W		30 W		31 W		32 W

1:	1 1 0	17:	0 0 0
2:	1 1 0	18:	0 0 0
3:	1 1 0	19:	0 0 0
4:	1 1 0	20:	0 0 0
5:	1 1 0	21:	1 0 0
6:	1 1 0	22:	1 0 0
7:	1 1 0	23:	1 0 0
8:	1 1 0	24:	1 0 0
9:	1 1 0	25:	1 0 0
10:	1 1 0	26:	1 0 0
11:	1 1 0	27:	1 0 0
12:	1 1 0	28:	1 0 0
13:	0 0 0	29:	1 0 0
14:	0 0 0	30:	1 0 0
15:	0 0 0	31:	1 0 0
16:	0 0 0	32:	1 0 0

Fig. 2. Encoding of checker position.

are called data types. To give a data type is to give a description of some types of information plus a rule for mapping that information into a sequence of bits. There are three independent reasons for having data types:

- (1) They simply provide pre-packaged solutions to be used by the designer for how to represent his information, among the many ways he might do it (several of which may be equally good). He is relieved of additional design decisions. In some cases, the conventional choices are actually better and the designer is relieved of having to rediscover that fact.
- (2) Given that a standard is used on all occasions, it provides for coordinating the representation used in separate parts of a large RT system. The designer need not explicitly check that he is using the same representation within various subsystems, which might then require additional processing to translate from one representation to another.
- (3) The basic operations provided in an RT technology must assume some representations for their input and output data. These impose a set of data types on the systems built up from these components. Extra

costs will occur with other data types because they must be translated into the form required by the given operations. A commitment to data types can be almost completely avoided by providing only the basic 'data types implicit in having bit sequences -- i.e., something corresponding to a sequence of bits (usually called the word) and the basic logical operations on a single pair of bits. However,; this substantially reduces the power of the resulting RT-systems technology and committing to higher data types is almost always worthwhile.

The systems we consider in this book, which are built out of PDP-16 modules, are quite restricted compared to the full range of digital processing (e.g., as done by large computers). Only a few data types show up, though additional data



amount of information.(3)

Any informational situation may be represented in terms of sequences of bits. As long as one can be definite about the things to be represented and the variety of these different things that can exist, one can invent some sort of correspondence to state these same things in terms of bit sequences. The limits are the total variety of things that have to be represented (e.g., 7 bits only distinguishes 128 things) and whether you can be definite about your description of them.

Standard ways exist for representing information in sequences of bits. These

-----

3. Can you find out why they differ? Actually, both representations take more bits than are necessary. Can you find better representations? What is the minimum number of bits required? Such questions are just intellectual games, of course, but they help to understand the issue of encoding. Actually, don't work too hard on the minimum; no one knows it exactly.

## RT-LEVEL COMPONENTS

A processing system always involves: (1) something to be processed; (2) a set of components that do the processing; and (3) a way to connect these components into a system. Digital systems process information. Therefore we should be able to identify all three of these things. Knowing them provides a clear idea of the nature of digital systems.

## WHAT IS PROCESSED: BITS

To say that digital systems process information is elliptical, for information only exists in so far as it is represented in some physical form. Digital systems actually process these physical representations of information. Furthermore, the representations it uses are highly specific: they consist of sequences of bits.

Something represents a bit if it can take on just one of two possible values, which are usually called zero (0) and one (1). Thus, if we had something (we might as well call it a register, which is what we will call it eventually) that can represent (hold) a sequence of seven bits, then it could take on various values, such as:

```

0 1 1 0 1 0 0
1 1 1 1 1 1 1
1 1 0 1 0 0 0
1 0 0 1 0 1 1

```

In fact, the register could take on any of  $2^7$  distinct patterns (where  $a^b$  is used to indicate that  $a$  is raised to the  $b$  power) corresponding to each bit-place taking on the value of 0 or 1 independently of the others. Thus, a seven bit register can represent one of 128 different situations ( $128 = 2^7$ ).

The representation of digital information is always in sequences of digital bits. All information must somehow be encode into such sequences. That can be a problem. If you want to. encode the age of a man in years; then you have to decide what pattern is 0 years old (i.e., less than a year), what pattern is 1 year old, what pattern 2, and so on. We could adopt the following code:

Age	Bit Sequence.
0	0 0 0 0 0 0 0
1	0 0 0 0 0 0 1
2	0 0 0 0 0 1 0
3	0 0 0 0 0 1 1
4	0 0 0 0 1 0 0
5	0 0 0 0 1 0 1
.	...
.	...
.	...
125	1 1 1 1 1 0 1
126	1 1 1 1 1 1 0
127	1 1 1 1 1 1 1

You may recognize this as the standard way of encoding the integers into a binary sequence. It is normally referred to as the binary number system. It takes seven bits to hold all the possible ages of modern man; a sequence of only six bits would have only permitted ages through 63. The integer value is just

search. This is why there may be no substitute for experience, since building up these networks of patterns must take place in the designers head, and requires a long immersion with the material to do it. Note that the immersion is in designing systems, not in abstract study of the technology. It is devoted to acquiring many partial solutions that jump little gaps from desired behaviors to patterns of components (subsystems) that achieve them.

We have not told you all we know, abstractly, about the task of design and how humans are able to do it. Perhaps we have told you enough to give you a feeling for why the book is written the way it is and how you should treat it. We can package some of this general advice in a way that may help to start you out in the right direction and may jog you occasionally if you get stuck. We offer this in Figure 1. But general strategies are no substitute for building up that network of special knowledge.

State explicitly the behavior desired.

Can you use a more precise language than English?

Everything desired cannot be said, but say some of it -- the most important.  
Design to the behavior you have explicitly specified.

You can always redesign it when new specifications are added.

Know the components at your disposal.

Design some little systems with each component, just to see how they operate.

Can you describe each component?

What is it for -- its function?

What peculiarities must be watched for?

Design at least some system, even if it is not the best system.

Improving, a system may be easier than designing one from scratch.

Use a separate component for each function to be performed.

Analyze the performance of any candidate system you design.

What is the function performed by each component?

Are all the functions necessary?

The cheapest and fastest parts of a system are those that don't exist.

Can one component perform several functions?

Can you quantify the behavior along each dimension of interest?

How does the system behave under extreme conditions?

How long does it take for the minimum task?

What are the maximum performances it can achieve?

How can it fail and what happens when it does?

Explore the design space.

Don't be afraid to design many variations for the same task.

Each actual design tells you something about other possibilities.

You learn nothing from a design that doesn't exist;

You can't even analyze its performance.

Fig. I. Good advice for designers.

appreciable way towards this goal, both in actual physical structure of the components and in the languages it provides for talking about them.

Jumping the remaining gap constitutes the primary act of designing. On what can it depend? Without knowing anything special, an intelligent person can still fashion designs. He can compare the list of specifications of what the components do with the list of specifications of what is wanted and see correspondences between them that indicate some necessary ingredients to the final system. As choices become apparent -- where there appears to be more than one way to accomplish a part of the task -- he can mentally explore forward, seeking some feature of each proposal that will let him discriminate which is preferable. Thought carries only so far, and if it doesn't get far enough to provide a clue, then a commitment must be made to one of the designs, carrying its development forward in a more explicit (hence time consuming) form. Backtracking to old choice points may occur (if they can be remembered), as it becomes apparent that the current candidate fails to meet the specifications.

The immense set of possible systems that can be fashioned and the depth of search required, measured in terms of design choices for an RT system of reasonable size, put a severe limit to how far unaided intelligence can proceed. For one thing, the process of taking a single design step is quite expensive if the characteristics of the components are only known indirectly, by having a list of them to consult. Thus, absolute familiarity with the components available contributes substantially to the ability to design. You will see this yourself when you must rummage in Chapter 2 for what components are possible, and then discover the swiftness later on, when all the components have been learned and come immediately to mind.

Familiarity with the desired features is equally important. Unlike the components, however, most of these will apply only to the particular design at hand. Working in a particular technology-- here, with a particular set of RT components -- means that the same components are used over and over again and their properties are fixed features, to be learned once and for all. With the desired features, the most important consideration is getting them out in the open where they will remain constant long enough to be analyzed and understood. In fact, as we noted, in most design it is not possible to state all the features desired -- one can always think of a few additional things it would be nice to have -- increased reliability, increased speed for a special class of input data, additional generality, or whatever. However, one cannot easily close the gap to a moving target. Stating some parts of the design specifications explicitly permits a design task to be posed and solved. Keeping them all implicit produces the equivalent of the moving target.

We have again focused on the two boundaries of the design tasks, the given components and the desired specifications. Knowing both of these are essential, but the gap is still too large in any real life design environment (such as RT-level design). What special knowledge helps to jump the gap? The major ingredient appears to be a very large number of patterns for subsystems that accomplish intermediate sized tasks -- exactly the sort of knowledge that is obtained by doing hundreds of designs and analyzing

their performance. We say appears, because the psychological evidence for what the master designer knows that the novice doesn't is not conclusive. But evidence from other areas (in particular, believe it or not, from our understanding of masters versus novices in playing chess), suggests strongly that the major role of experience is the building up of a very large number of functional patterns, so that they are recognized instantly in a new design situation, rather than having to be discovered by intelligent

If you have done much real design (whether of computer systems or other things) your own experience will probably not be entirely in accord with the above. Design seems a much more open activity, in which the emphasis appears to be on discovering ways to put the components together to obtain various specific features. It never seems possible to put all the desired features together at one time and think of it as an optimization problem. In fact, the model of a design problem that seems closest to actuality is one where there is an unlimited supply of desirable behaviors, some of which are more important, some less. The problem of design, then, is to obtain a system that has some of these desirable behaviors (the important ones) and, if there is any freedom left, it can be used to obtain some additional desired properties. We can even put this in the same way as the other formulations:

Given: (1) A technology  
(2) An ordered list of specifications of desired behavior.  
Create: A system that has as many as possible of the specifications, taken in order.

If the list, is really unlimited, there is no way to pose this problem as an optimization problem, since it is never possible to take all the constraints together. In fact, some of the constraints down the line may well be incompatible with some of the initial important ones. This formulation, though it still leaves some loose ends, captures an essential feature of design problems as posed by humans -- that humans can never know all that they want. The nature of our long term memories is such that we can never find out all that we know about something, even our, own desires. Thus, it often seems better to add a new specification to the list of desired properties of a proposed design, than simply to make the design better along the dimensions currently included in the list of desired specifications.

It can be seen from all formulations of the design task that a clear scheme for setting out the specifications desired and a clear scheme for describing the components out of which to construct systems, are of critical importance. Design is concerned with bridging the gap between these two types of information. If one simply selects arbitrarily a set of components and connects them together into a system there is only an infinitesimal chance that they will produce a performance of any interest at all. Thus, the essential property of the languages for describing the two boundaries of the design task -- the technology on the one side and the desired specifications on the other -- is to make it easy to jump across the gap. Given a description of components and subsystems, it should be easy to see what it is they do -- what functions they perform. Given a description of desired behavior, it should be easy to see what components satisfy the demands.

We will devote the last two sections of this first, chapter to these two boundary conditions: first, how to describe the RT-level components and second, how to describe desired behavior. In the best of all possible worlds these descriptions should be so close to each other that to express a desire is to set down the RT-level system that could accomplish it. In reality, of course, design is harder than this. Candidate designs must be proposed, which must then be evaluated to see how well they satisfy the specifications--

with the expectation that several iterations will be required before a tolerable design is discovered. However, as we remarked earlier,, a good design technology moves an

This form of the design problem shows up frequently. For instance, we spend all of Chapter 4 designing a multiplier, and it is easy to identify there all of the items in the problem statement above. Likewise most of the problems in the book have this form. But *there* are other variants of the design problem. For example:

Given: (1) An already-designed system.

(2) A specification of additional or modified behavior.

Create: A modification of the given system that exhibits the modified behavior.

Sometimes the added requirement is a specific bit of behavior --e.g., to display the result in a new useful way. Sometimes the added requirement is in terms of some measure of the system -- e.g., reduce its cost by half, or increase its speed by two.

Both types of design problems share some things in common. For one, the givens are essentially parts and the solution is to be constructed or composed out of these parts. Not all problems have this constructive character. For instance, solving a set of simultaneous equations seems quite different, as does (say) finding someone's house in an unknown city, or (say) deciding which used car to buy. Even many problems closely associated with RT-level systems are quite different. Determining if a given system actually performs a given processing task correctly is quite a distinct type of problem, as is determining which of two given systems is better for a given family of tasks. These latter *are* often called analysis problems and the design ones are called synthesis problems; though just assigning different names does not really help much.

An important feature of the design problems above is that a specification of the desired features of the system is given. It is usually -- and we do so, as well -- that being clear on the specification of the system is all important. If you don't know what you want, how can you build it? This leads to the view that a proper design problem has a complete specification, and that whenever the properties desired of a system are incomplete or vague, then the design problem is ill-posed. For example, if you look in books on logical design, problems such as the following are taken as prototypes of design problems:

Given: (1) A system of components for doing the

three logical functions, AND(X,Y), OR(X,Y), NOT(X).

(2) A cost for each component.

(3) A specific logical function of N logical variables  $F(X_1, X_2, \dots, X_N)$ .

Create: The logic network that computes F and

has minimum cost.

Algorithms are then developed for solving this minimization problem. In fact, a great deal of work in the area of logical design has gone into the investigation of such minimization problems. One is left with the impression that design problems are all ultimately mathematical problems of constrained optimization, where it is desired to optimize an objective function (e.g., minimize cost or maximize speed) subject to satisfying the constraints that the system produce certain specific behaviors;

or easier to assemble. However, most of these options exist only potentially, since only the PDP-16 modules are commercially available.(2) Although we will treat general register-transfer design (in Chapter 8) and the design of RT-level modules themselves (in Chapter 7), the book concentrates on design with a fixed system of RT-level primitives (namely, the PDP-16 modules).

## THE TASK OF DESIGN

Besides the two questions that you are entitled to ask, there is a third one:

### 3. How does one design systems of type X?

Whether you are entitled to an answer to this question or not is a moot point. In fact, you cannot obtain an adequate answer in specific and operational terms. Engineering textbooks on the design of X normally tell you all about the technology, analyzing its properties and what it can do. They exhibit some designs of systems that accomplish typical but simple objectives and analyze how these designs actually meet the objectives. They also pose many simple problems, some in design, some further analyzing the technology. Beyond this, they assume that you will use your native intelligence and discover how to design by yourself. You have been led to water; it is up to you not only to drink but to find out how to drink.

The difficulties in giving you an answer stem from two distinct sources. On the one hand, no one knows much about how design is actually accomplished by humans. The psychological processes involved are only a little studied and that only recently. On the other hand, what little is known, both about designing and educating, indicates that what must be learned consists of extensive cognitive structures built up in the context of attempting to design. Even if we knew all the right things (whatever they might be) and told them- all to you (and you listened) -- even so, this could serve only to let you construct for yourself the design situations in which to experience and learn, thus to grow into being a designer. In short, there may be no avoiding learning how to design by doing it.

In this light a textbook on design should operate as a guide to experience. It should serve to immerse you in a continual sequence of designs, contrived to expose you to the right sorts of things and to let you see how certain ways of designing lead to successful designs. This book, then, consists mostly of design problems, with various divisions of labor between what we design, which exhibit solutions for your understanding, and what we expect you to design, which let you obtain the experience yourself.

Most of the explicit things we have to say about design will occur in connection with the specific design examples. But a few words can be said about the nature of the task of design. The fundamental problem of RT-level design can be stated thus:

Given: (1) A technology consisting of a set of primitive

RT-level components and a method of assembling -  
them into systems.

(2) A specification of the behavior desired.

Create: A system out of the components that has the desired behavior.

-----

2. The Macromodules of Clark are available in a more limited way. A single large collection of them exists in his laboratory at Washington University in St. Louis, where they are being used for experimental studies.

The physical technology that underlies the register-transfer level is located at the basic logic level and permits the design of any combinational or sequential logic circuit. There are no natural physical RT-level components and the creation of complete sets of RT-level primitives is an act of relatively unfettered conceptual design. The limits that exist are only on the total complexity of the components, not on the logic and memory functions they are to perform.

Matters are still more complicated. There are many distinct physical technologies, even for the logic level. Much of the history of the computer and digital systems field can be traced to the spectacular development of these technologies, as they have increased in speed and reliability by orders of magnitude while decreasing in size and cost, also by orders of magnitude. The current array of technologies (various systems using MSI and LSI circuits) provide various cost/speed tradeoffs. They all have the property, stemming from the assembly of many basic logical elements on a single small physical surface (the chip), that they force the creation of register-transfer level components. That is, not only do they finally make possible RT-level components, they prohibit design in terms of a collection of physical components each representing a single basic logical element (a NAND gate or a flip-flop) that can be arbitrarily connected together to form larger systems.

RT-level technology is thus of recent origin (as opposed to the conceptual notions of the RT-level, which have been around since digital computers began). It has emerged only with the emergence of these underlying physical technologies. A great variety of RT-level components have been produced; they can be found via manufacturer's catalogues (Texas Instruments, ...; Fairchild Semiconductor...). But so recent is the technology that only two sets of complete RT-level primitives have been produced, the first being the Macromodules of Clark (1967) and the second being the PDP-16 RTM's used in the present book.(1)

To be concrete, we give a specification of the physical characteristics of the RT-level components out of which the digital systems in this book will be designed. Modules vary in size, but all are on boards a few inches on each side. They vary widely in the amount of logical function they contain, but range from tens up to a few thousand elementary logical operations and tens up to a few thousand bits of memory. They execute their operations (a transfer, an addition, etc.) in a few microseconds and the mean time between errors is of the order of  $10^{12}$  operations (i.e., measured in fractions of a year). Costs vary with the complexity of the module, but a complex module is about a hundred dollars. Useful systems can be built with a dozen modules and small computers can be constructed with about twenty. Thus, the costs of the physical components of designed systems is in the range of one to a few thousand dollars. Given the design and the modules, the physical systems can be assembled and checked out in a few days by a single person.

With these last specifications we have finally answered the second question on what digital systems are constructed of; It took some preparation, since we had to distinguish the particular slice of digital systems design we were concerned with. It even took a little cheating in order to be specific, since (just

as with the tasks digital systems perform) digital technologies are diverse and offer a range of options. Compared to the PDP-16 modules, RT-level modules could be larger or smaller, faster or slower, cheaper or more expensive, harder

-----

1. Other sets of RT-level primitives have been discussed in the theoretical literature (e.g., Patil, 1970), but none have led yet -to new sets of physical primitives that can be used for actual systems.

register-transfer components. We will present a scheme for describing any set of RT-level components. In practice, there will be available only a particular set, and the full reality of design requires that we settle on some such set. We will do this, using a set called the DEC PDP-16 Register Transfer Modules (also called RTM's), which were specified initially by one of the authors (GB) with collaboration from another one (JO) and developed at DEC by John Eggert, Robert Van Naarden, and Peter Williams. The entire second chapter is devoted to defining these modules.

Underlying any particular set of RT-level components is a technology that permits the construction of physical devices that perform as advertised for each component. It is this digital technology that dictates the speed, size, cost and reliability of the RT components. It is another peculiar feature of the RT-level of design of digital systems that the underlying physical technology can be used indifferently to produce many different sets of components. Tradeoffs in speed and cost exist, to be sure, but these are tradeoffs on the complexity of the processing accomplished by a proposed component. It very rarely happens that the technology is good (say) for registers of size 16 bits, but bad for registers of size 24 bits, or good for logical, but bad for arithmetic operations.

This separation of digital technology from the RT level reveals an important feature of digital systems: the hierarchy of levels of design. It is common to distinguish four main levels: the circuit level (of resistors, transistors, etc.); the logic level (of bits and AND'S and OR'S); the programming level (of addresses and instructions); and the level of total system configurations (of processors and drums and printers).. This last topmost level has no accepted name, though we have proposed elsewhere (Bell and Newell, 1971) that it be called the PMS level (for processors, memories and switches, three of its most important components). In addition to these main levels there are others. One can move downward below the circuit level to the physics of the devices. There are many levels within the programming level, e.g., the levels defined by the operating system, by the higher languages being used, etc. Most important to us, there are two levels of logic; the combinational and sequential level and the register-transfer level..

With such a hierarchy of levels, there are several technologies that simultaneously exist for digital systems -- a technology for circuits, for logic- design, for programming and so on. It is important to know what level of technology is being assumed in a work on digital system design. The lower the starting technology, the more freedom is available in the construction of systems, but correspondingly, the more difficult is the design task to obtain a system that performs the desired information processing job.

This book starts at the register-transfer level, which is the higher of the two logic levels. Most books on digital design start at some lower level. Recent books tend to start with the basic logic level of combinational and sequential circuits (e.g., Peatman, 1972). Earlier books, tended to start even lower down with the electronic circuits used to construct the basic logic elements, such as a flip-flop, an AND gate, and an inverter (e.g., Ware, 1963).

The gradual trend of digital design to start with higher and higher levels stems from three related aspects of the digital field: (1) the development of physical technologies that make it possible and economical to package physical components at higher levels; (2) the increasing complexity. of the systems to be designed, so that one simply cannot afford to start so far down; and (3). the gradual conceptual development of the field, so that it is possible to conceive of a base for design at the higher levels. Thus, this is the first book, as far as we know, that starts at the register-transfer level." But you may rest assured it will not be the last.

credited with the ability for general information processing. We use the more general term, but not to downgrade the computer. General purpose stored program digital computers are by far the most effective existing device for processing information above a certain level of complexity. But systems are always to be engineered to the actual tasks they are to perform. For many specific and limited information processing tasks, digital systems that do not have the classic organization of a stored program computer are cost-effective and are to be much preferred to the use of a general purpose computer. We do deal with computers and computer-like organizations in this book, as well as with other digital systems. But our concern is not with the large data processing tasks and scientific calculations that lead immediately to large computers, higher level programming languages, multiprogramming, time sharing and multi processing. Our focus is on lesser fry. The computer organizations we entertain are mini computer systems, where there is always the potential that some other form of digital system will prove the design of choice, given performance and cost objectives.

Despite our insistence on the propriety of a highly general answer to the first main question, you have a right to expect more, namely, that we should illustrate for you the kinds of information processing that can actually be achieved by digital systems that can actually be designed. A scan of the Table of Contents will show the array of different systems designed in the present book. Many of them, of course, do intermediate tasks, useful only if one is already in the midst of digital systems: multiplication, interfacing to computer, conversion of analog signals to digital form, etc. But there are also systems that do an externally defined task: a histogram recorder, a programmable desk calculator, a music synthesizer, a coating thickness monitor, a system for controlling the destinations of objects on a conveyor belt, a system for testing the correctness of a memory, and a Turing machine (a general information processing system that has played a strong role in the theoretical development of computer science). Besides these, there are designs for several small digital computers, which simply puts off the question of what final information processing tasks can be done. Even within the confines of a single book, the diversity of tasks essentially precludes categorization, except in terms of amount of processing and size of memory required.

## WHAT DIGITAL SYSTEMS ARE CONSTRUCTED OF

Digital systems are constructed out of registers, which hold information encoded in the form of arrays of bits. The registers are connected together by transfer paths, which permit the information in one register to be transmitted to another via processing units. These take as input sets of bits and provide as output some logical or arithmetic function of these bits. The level of digital systems corresponding to these components is called the register-transfer level (or RT level).

This is not quite all there is to it. For one thing, it is critically important exactly what components are available in the way of registers, processing units and transfer paths, and what are their precise properties, in terms of performance, connectivity, reliability, cost, size, and availability. Real design occurs with real components and much that makes design an intellectual challenge derives from the fact

that only certain particular components are available out of all possible components. The old saw about "If wishes were horses, beggars would ride" applies with a vengeance to design.

We will spend a goodly part of this first chapter describing the nature of

2

[previous](#) | [contents](#) | [next](#)

## ■ CHAPTER 1 INTRODUCTION

This book tells how to design digital systems. Whenever a book proposes to teach you how to design systems of type X, you are entitled to ask two questions and receive specific answers:

- 1. What do systems of type X do?
- 2. What is used to construct systems of type X?

Only then will you know whether you should be interested in designing such systems, whether you want systems that perform the stated tasks, and whether the technology proposed is reasonable. There are many other reasons why you might wish to read or study a book on the design of type-X systems: The beauty of the designs; the rigor of the intellectual discipline; the view afforded of a complex intellectual structure; the pleasure of mastering yet one more art. All these, though real, should be secondary. Design is the art of obtaining useful devices, and technologies are fashioned to make design practical without excessive creativity and intelligence. First things first.

Herewith we attempt to answer the two basic questions.

### WHAT DIGITAL SYSTEMS DO

Digital systems process information. They do calculations on information input in specified physical forms, delivering answers represented in some other (or the same) physical form. They sense the current state of physical devices, such as instruments or manufacturing tools, and control them to achieve given objectives. They acquire information at a given time and in a given place, and make that information available at some later time and in some place. They monitor ongoing physical processes at discrete times so that they perform as desired, recording errors when they occur and deactivating a process if it moves outside of stated bounds of safety and reliability.

If these statements seem to you too general -- that we should answer our question by stating-specific capabilities -- then it is you who must revise your notions, not us who must become more definite. For a peculiarity of digital systems is that they are capable of performing any specified task of processing information and, further, that they can perform any collection of such tasks, doing each when commanded. Not only are they capable of doing this, in some technical sense of possibility, but they are the main practical device for doing it in an ever widening domain of application.

There are many technologies for processing information and for sensing and controlling physical devices. For instance an autopilot on an airplane senses broadcasted external signals from which it can determine the plane's position and then manipulate the plane's controls to guide it along a predetermined path.

Currently these devices are entirely analog, i.e. continuous, working with the intensities and phase relationships of received electromagnetic signals, making all the computations of how to respond in terms of voltages in fixed circuits, and effecting the. controls by setting other voltages appropriately. These analog systems have been developed to a high state of performance. and reliability. However, add a little complexity to the task, especially if it involves remembering past behavior or arbitrary future plans (e.g., a flight plan), and make digital technology a little cheaper, and digital systems may become practical competitors for the task.

Digital systems include digital computers, which are the machines normally

■ 1

[previous](#) | [contents](#) | [next](#)

[previous](#) | [contents](#) | [next](#)

[CHAPTER 8 RT LEVEL DESIGN WITH CONVENTIONAL SWITCHING CIRCUIT COMPONENTS](#)

[Sum of Positive Integers to N](#)

[A 16-bit Multiplier Using the Russian Peasant's Algorithm](#)

[EPUT Meter](#)

[CHAPTER 9 SYSTEMS AND COMPONENT TESTING](#)

[System Debugging](#)

[Acceptance Testing of Modules](#)

[Memory Test](#)

[M\(transfer\) and M\(byte\) Test](#)

[DMgpa Test](#)

[Kbus Test](#)

[Maintenance Testing](#)

[Conclusion](#)

[Problems](#)

[TABLE 1 COMMON ARITHMETIC, LOGICAL, AND RELATIONAL OPERATORS](#)

[TABLE 2 COMMON SYMBOLS USED BOTH IN ISP AND PMS FORMS](#)

[TABLE 3 COMMON EXPRESSION FORMS USED IN RTM SYSTEMS](#)

[TABLE 4 RTM PIN NAMES, POWER, AND LOADING](#)

[TABLE 5 ASCII AND EBCDIC CHARACTER SET ENCODINGS](#)

[BIBLIOGRAPHY](#)

[INDEX](#)

[previous](#) | [contents](#) | [next](#)

- [Crtm-1: An RTM Minicomputer](#)  
[Summary](#)  
[Evaluation of the Design](#)  
[Summary of the Types of Variations](#)  
[Problems](#)

## CHAPTER 5 ADVANCED DESIGN EXAMPLES

- [Data Operations Subprocesses](#)  
[Scientific \(Floating Point\) Notation](#)  
[Binary Coded Decimal \(8421 Code\) Arithmetic and Conversion](#)  
[Time Based Systems: Clocks, Timers, Waveform Generators, and Waveform Analyzers](#)  
[Sawtooth Waveform \(Ramp\) Generator](#)  
[Events Per Unit Time \(EPUT\) Meter](#)  
[Time-shared EPUT meter](#)  
[Histogram Recorder](#)  
[A Monitor to Measure the Thickness of a Coating Process](#)  
[K\(Arbiter\), an ERTM for Synchronization of Multiple Processes](#)  
[Transducers for Data Communications Systems](#)  
[Analog-To-Digital Converter](#)  
[Remote Analog-To-Digital Sampling Unit with Multiple Channel Input](#)  
[Teletrola, A Musical Terminal](#)  
[Design of Interface to a Paper Tape Punch](#)  
[Sampled Analog Input to Paper Tape Converter](#)  
[Turing Machine Simulator](#)  
[Controller for a Conveyor-Bin System](#)  
[Memories](#)  
[M\(queue\) First-in First-out Memory](#)  
[M\(stack\) Last-in First-out Memory](#)  
[M\(content addressable\)](#)

## CHAPTER 6 COMPUTER DESIGN EXAMPLES

- [Crtm-1: A Small, General Purpose, Stored Program. Computer Designed Using RTM's Using ISP to Define the Computer](#)  
[PDP-8/RTM Implementation](#)

[The PDP-16/M-A Subminicomputer Based on RTM's](#)

[The RTM-Computer Interfaces](#)

[RTM Computer Interface via Program Controlled Access](#)

[RTM-Minicomputer Interface via Direct Memory Access to Computer](#)

[Ps, a Special Processor to Sample Analog Input Data and Compare Against Limits](#)

[Specialized Processors for Event Counting](#)

[Clock-Calendar Interfaced to a Computer](#)

[A General Register Based Minicomputer Implementation](#)

[Simple Desk Calculator](#)

[Crtm-2/2: A Simple RTM Computer Using the DMar](#)

## [CHAPTER 7 THE DESIGN OF RTM'S](#)

- [Choosing a Module Set](#)  
[Switching Circuit Details](#)

xii

[previous](#) | [contents](#) | [next](#)

## CONTENTS

### [PREFACE](#)

### [CHAPTER 1 INTRODUCTION](#)

- [What Digital Systems Do and What They are Constructed.. of RT-Level Components](#)  
[Specifying System Behavior](#)  
[Summary](#)

### [CHAPTER 2 PDP-16: A SYSTEM OF RT-LEVEL MODULES \(RTM'S\)](#)

- [A First Look at RTM's](#)  
[RTM Primitives and Their Behavior](#)
  - [The RTM System](#)
    - [Four Basic Modules](#)
    - [RTM System Operation](#)
    - [Summary of Remaining Modules](#)

[The K\(Program Controlled Sequencer\PCS\),- A Microprogrammed Controller for a Centralized Encoded Control Part](#)

[Two New Modules: DM\(Arithmetic and Registers\) and K\(Bus Control and Termination Combined\)](#)

### [CHAPTER 3 ELEMENTARY DESIGN EXAMPLES](#)

- [Counting the Number of Ones in a word](#)  
[Conditional Execute Extended RTM](#)  
[Logical Shift, Circular Shift \(Rotate\), and Arithmetic Shift Operations](#)  
[For Loop Extended RTM](#)  
[Fibonacci Number Generator](#)  
[K\(Programmable \(Variable\) Clock\)](#)  
[K\(Wait-Until\) Extended RTM](#)

[D\(Square Wave Generator\)](#)

[CHAPTER 4 EXPLORING RTM DESIGN ISSUES](#)

- [Basic Design of a Multiplier](#)
  - [Variations in the Use of Multiplication](#)
    - [The Natural Mapping: Macros](#)
    - [Shared Facilities: Subroutines](#)
    - [Parallelism: Arrays](#)
    - [Parallelism: Pipelines](#)
    - [Summary](#)
  - [Variations in Implementing the Basic Algorithm](#)
  - [General Parallelism: Concurrency](#)
  - [Facility Sharing](#)
  - [Reduction of the Control Part: Unwinding Loops](#)
  - [Variations in the Algorithm](#)
  - [Alternative Representations](#)
  - [Memory versus Computation: Table Look-up](#)
  - [Alternative: The Russian Peasant's Algorithm](#)
  - [Variations in the Implementation of Control](#)
  - [K\(PCS\Programmable Command Sequencer\)](#)
  - [C\(16/M\): A Microcode Computer](#)

[previous](#) | [contents](#) | [next](#)

insistence to one of the authors (GB) that there should be a reasonable way (especially for students) to build digital systems. This led to a National Science Foundation grant (GY-5160) for undergraduate laboratory equipment which was used to purchase RTM's.

We would also like to thank Professor Angel Jordan, head of the Electrical Engineering Department, and Professor Joseph Traub, head of the Computer Science Department for their support and encouragement. In addition, Mr. Paul Stockhausen, business manager of the Computer Science Department, patiently provided valuable support for this work.

AT CMU there have been a number of students and researchers who have contributed to the modules and to this book A simulator by Messrs. Bhandarkar, Goel, Rege, Schulte, and Staab contributed in -this regard. Mr. Paolo Coraluppi together with Messrs. Berera, Orban, and Philip set up The PDP-16 laboratory stations at CMU and got the first operational results with the modules in an educational environment.

We have particularly enjoyed the experimentation with the art of textbook production, by using our own computing facility for editing and typesetting -- it being the antithesis of the conventional textbook production process. .. The specific people who used the process and contributed to, the direct production are credited, following the author page. Other than costing less for production, it is also approximately 5 to 10 times faster. The editing and typesetting were done on line using the PDP-10, in conjunction with the SOS editor and PUB typesetting programs developed at Stanford University. The printing masters were created directly using a Xerox -LDX(Long Distance Xerography) printer which was developed at CMU. The printer has a 200 point/inch resolution and is driven directly by a PDP-11. All characters are thus generated point by point by scanning, and a page is printed in about 10 seconds. The pages are then photoreduced for lithographic offset printing. Professor Raj Reddy led the development of the LDX project. Lee Erman, Richard Neely, George Robertson, Philip Karlton, and Richard Johnsson provided consultation and-programming in making the system convenient to operate . The necessary News. Gothic Roman character sets were formed and input by Janice Karlton.

It is evident that we have had much help with this book. However, we alone remain responsible for the remaining errors and difficulties in the text and problems.

G.B. J.G. A.N.

July, 1972

[previous](#) | [contents](#) | [next](#)

permits some additional features of RT-level design to emerge that remain hidden if a fixed set of primitives is taken. It also reflects the fact that most RT-level design is still done ad hoc, without a set of primitives in mind. Chapter 9 is on the problem of debugging a system, designing systems for testability, and testing (verifying) that the individual modules carry out their required function. To a very real extent we have completely overlooked the constraint in all problems that these designs have to be tested. We hope to be forgiven for this, for we do not approve of such design. Our only excuse is that the testing constraint might obscure the problem. For the student who builds actual systems, testing will be a very important problem, after he builds his first system.

Various supporting and reference materials are gathered into Tables at the end of the book.

Ideally all of the example problems would have been verified by either construction or simulation. This is not the case. Only a few of the designs have been wired; these include sum-of-integers, multiplication, Teletrola (by Michael Knudsen), several of the computers, and various arithmetic algorithms. Simulation has also been used on the CMU RTM simulator, but is generally top costly. Also, some of the designs are pedagogical in nature, and presented only to show tradeoffs. Therefore, the reader will no doubt find a number of errors. We hope these will be fed back to us so that we may correct subsequent versions.

The list of acknowledgements with regards to the modules is long and we will attempt to be as complete as our memories allow. It would be difficult to acknowledge all the representational work which permits the simple description of these structures, because its origin is with the state machine and flowchart. Clark's Macromodules (with engineering care and assistance of Charles Molnar), provided the main impetus to the idea that fairly large modules could exist to build digital systems, these being smaller than computer components (i.e., Processor, Memory, Switches), but still much larger than gates and flip flops packaged into Integrated Circuits. Much of the work on expressing and implementing control was taken from the PDP-6 and PDP-10 -- especially the ideas of parallel and serial branches and subroutines.

Research on basic ideas underlying hardware modularity together with various applications, studies was supported by Advanced Research Projects Agency of the Office of the Secretary of Defense (F 44620-67-C-0058) and is monitored by the Air Force Office of Scientific Research.

The whole management group at DEC who, in fact, are on the line for deciding to produce such a collection of modules, are perhaps the only real pioneers that have anything to lose by building them. These include Kenneth Olsen, Stanley Olsen, Al Devault, and Fred Gould. Robert Pouliot and Bill Hogan finally decided to produce and distribute the textbook -- they also being responsible for module sales. This group is responsible for allowing us to experiment with the production of textbooks. The engineering group led by John Eggert, with the assistance of Peter Williams and Robert Van Naarden are to be commended for carrying basic ideas and specifications to a workable engineering solution.

Professor E. M. (Rod) Williams, former Head of Electrical Engineering at CMU, was a key factor in PDP-

16's existence, since they came about through his

[previous](#) | [contents](#) | [next](#)

thus to be charged to capital budgets rather than current expenses, and to be reviewed by a Facilities Committee. The success of minicomputers is gradually wearing down some of these myths, though it leaves others untouched. The sort of digital systems designed in this book give the lie to all of them. But even realities dispel myths only slowly. Thus, we feel it is legitimate to state the case for register-transfer systems designs in specific and unequivocal terms. This book is our attempt to do so.

The book is organized as follows: Chapter 1 provides an introduction, describing both what digital systems are and their applications. It includes a basic description of register-transfer level modules in the functional PMS language used throughout the book. It also introduces ways to specify the behavior of desired systems. It provides a set of diverse examples of digital systems to define concretely the scope of the book. Finally, it discusses the processes of design. Chapter 2 is a systematic description of the specific set of PDP-16 modules.

Chapter 3 gives the solutions of 8 small systems built from RTM's: These should give the reader a good feeling about the design process and what RTM systems look like, since the problems being solved are small and straightforward. These problems are presented in a relatively standard format which we carry throughout.

Chapter 4 is the main reference on the process of RTM design. It too uses a problem, that of a multiplier, to illustrate both the design process and the many trade-offs that are possible even with a limited set of modules. These tradeoffs include: hardware-microprogram-program; parallelism (time-size) in the form of concurrency, array processing, and pipeline processing; the algorithm; and memory-compute.

Chapters 5 and 6 proceed by cases to design a wide array of systems. Some are introduced at length with intensive discussion, some are given simply as problems. Always there are suggestions for further work, even where our own treatment is most complete. Generally, designs are separated from the statement of the design problem, so that a student can attempt the design first, before seeing how we approached the problem. Any of the readers mentioned earlier-in the preface should find some designs to fit his interest or tickle his fancy. We include the design of several real computers so that the mystery of the operation of computers can be dispelled once and for all, for all readers. In Chapter 6, we urge the reader who has never designed, built, and programmed his very own stored program computer to take the few days to do so. All in all, about one hundred design problems are posed and treated, and a similar number of tasks are posed.

Chapter 7 disc the design of register transfer modules themselves. This is necessary, not only to keep them from appearing to be black boxes, but because no one set of modules is appropriate for all tasks. As a by-product, we hope that some integrated circuit logical designer will read this and derive designs for control schemes that people can actually easily use with integrated circuit logical design. We also hope to encourage the logic design textbook writer to think about control schemes. Chapter 8 treats register transfer design generally, freed from the dependence on the PDP-16 modules. This chapter

[previous](#) | [contents](#) | [next](#)

(1971) for structure and behavior, knowledge of these, languages (notations) is not required. We have avoided using a register transfer language for defining behavior. We use instead the flowchart, with ISP language expressions. This method is equally formal and precise, yet shows 'the structure and parallelism without burying the reader in the language control syntax. We do invite the reader to describe and design RTM systems in his favorite (1 dimensional) register transfer language, as an exercise. Alternatively, he should simulate his design in a conventional language (e.g., Fortran). In a few instances, we have described systems in pure ISP language.

The book is aimed primarily at people who wish to design specific small digital systems -- either to learn about digital technology or to accomplish a particular task. The book is specifically designed to be used as either a primary or a secondary text in undergraduate and graduate courses on logical design. It can also be used for self study by students in departments of electrical engineering. and computer science. It is especially suited for the latter, if the department does not provide any courses that descend below the programming level to tell how digital systems really operate. Although not a requirement, a knowledge of a programming language will be helpful in reading the book.

When the book is used as a primary text, at the sophomore or junior level, as a first course, some minor amount of supplementary material on number representation and Boolean algebra may be useful. After the first four chapters, the book becomes topical and can be read in any order. We strongly recommend the style of design and analysis given in Chapter 4, and several weeks can be spent studying this chapter.

The book is also designed specifically for self study by engineers and scientists of all persuasions (civil, mechanical, chemical,..., physics, chemistry, biology,..) who deal with laboratory and industrial processes that need monitoring and control. As this book demonstrates, the design of RT-level systems is no more difficult than programming, the construction of working systems is no more onerous than debugging programs,, and the cost of such systems is low enough to be cost-effective for use with all manner of laboratory instrumentation, industrial instrumentation, and production processes. The performance that can be delivered in the way of useful bits processed per second per dollar is, for the right task in the right place, phenomenal. The design time appears to be cut at least a factor or 10, making it possible to design, wire, and test a simple computer or a desk calculator in about 20 hours.

If we appear to sell a little bit in the previous paragraph, that sour intent. There is a certain amount of missionary work to be done. The making of myths is a constant avocation of man. He establishes them continually as he packages his experiences in verbal formulas to make his world seem comfortable and familiar. They linger almost forever, being cut free from the old experiences that generated them. Some myths that cling to the computer field are: that hardware is much harder to construct than software, so always program rather than construct equipment; "that all digital systems are digital computers; that interfacing digital systems to the external world is complex and costly; that the only fit interfaces for digital systems. are punched cards in and printers out (recently modified to include Teletypes in and out); that digital systems are expensive facilities (i.e., capital equipment), rather than instruments or supplies,

[previous](#) | [contents](#) | [next](#)

of our own invention), and partly because these are the only system of RT modules that are readily available through commercial channels.

In addition, we believe that the emergence of the RT-level is still uncertain enough in its character that it should stay fairly close to its roots. The field is not quite ready for a general treatment of the register-transfer level. In fact, we have in preparation a more extended and scholarly treatment which includes other modules, microprogramming, and switching circuit RT design; the present book can be viewed as an initial attempt to characterize and create this level in an appropriate way.

Although the book uses only a single set of modules (aside from one chapter) and might appear limiting, we urge the reader to go beyond the book and see for himself how general RT level design is carried out. Only some imagination is needed to show that the representation and design techniques are applicable to other RT schemes. We believe the payoff for the reader will be far greater than, say, reading about RT languages for the simulation of hypothetical digital systems.

To those who are interested in microprogramming, RTM's provide a very interesting structure. Microprogrammed and RTM operations appear similar; at each step there are one or more register transfer actions, and input conditions select the next step. There are two substantial differences between RTM and microprogrammed systems: the RTM control part is hardwired (as opposed to being stored in a memory); and the RTM structure can be changed (as opposed to having a fixed data and control part).

A special microprogrammed control module called K(Programmed Control Sequencer) is also provided as an alternative to hardwiring the control part. Thus, for studying and using microprogramming, its advantages remain, but in addition the very structure can also be changed, providing substantial flexibility.

The book is written in an elementary "how-to" fashion. Its aim is to get people to design digital systems at the RT level. Our reasons for choosing this style of book can be inferred from the previous paragraphs. Let us note only that a design level exists and becomes real only if lots of designs get created in its terms. Thus a "how-to" book in the area of design plays the same role as does the treatise, with theory and supporting data, in an area of science. This has led us to make various assumptions about our readers.

Thus, we have not discussed number representation in any detail for several reasons. Positional notation for representing binary numbers is being presented in primary and secondary schools. In the rare cases where readers are not familiar with number representation, the concepts can either be "picked up" herein, or be obtained from any book that has anything to do with digital computers or digital systems. Boolean algebra is also not given for the same reasons. Both concepts are not used in substantial quantities, although the reader may feel they are needed to thoroughly understand all of the problems. The concepts of integers and Boolean algebra as given in a programming language (e.g., Fortran, Algol, Basic) are sufficient.

Although RTM's use parts of the PMS and ISP notation of Bell and Newell

[previous](#) | [contents](#) | [next](#)

## PREFACE

Digital technology continues to change and with it changes both our understanding of digital systems and the nature of the cost-effective ways to create them. The development has continued to be towards greater performance and the accomplishment of more complex information processing tasks. Simultaneously it has continued toward cheaper, more reliable systems, and the accomplishment .of an ever widening spectrum of tasks. This joint development gives no hint yet of slackening.

One particular item in this development is the current emergence of the register-transfer level (RT level) to the status of a full fledged system level. The conceptualization of digital systems in terms of registers and data transfers goes back to the beginnings of digital computers. But in practice logical design has been carried out in terms of combinational and sequential circuits, with the concepts of registers, functional units and transfer paths playing only a heuristic role. Examination of any of the standard books on logical design will confirm this, as will the degradation of the formal concern with the register-transfer level since the work of Bartee, Lebow and Reed (1962) which provided a high water mark in the explicit treatment of this level.

The current maturation of the register-transfer level is due primarily to the technology -- to MSI and LSI fabrication techniques. However, it is perhaps due also to the creation of systems of primitives at the RT level, first the Macromodules by Wesley Clark (1966) at Washington University and then by one of the authors, (see Bell and Grason, 1971). These have provided the conceptual scheme around which to conceive of design operating exclusively at the register-transfer level, without involvement of the lower logical levels of general combinational and sequential circuits. An additional conceptual tool, in our own case at least, has been the development of an appropriate language for describing RT-level components (so-called PMS notation, see Bell and Newell, 1971). This is a functional notation which has led us off in a somewhat different direction than that implicit in the earlier attempts to create a register-transfer level, e.g., the work of Bartee et.al., which was strongly algebraic in character (essentially formalizing the level as Boolean rings on vector elements).

This book is an attempt to bring this level of design to fruition. We have based it strongly on the use of a specific set of RT-level components, the DEC PDP- 16 modules which we call Register Transfer Modules (RTM's). Our reasons for using RTM's herein are partly pedagogical, partly proprietary (in both a corporate(1) and a personal sense--we like our own modules, they being children

-----

1. The system of modules used here was originally called Register Transfer Modules (RTM's) and was initially designed by one of us (GB) at Carnegie- Mellon University for educational purposes under an NSF equipment grant (GY 5160). They have been described in the literature (Bell and Grason, 1971;

Bell, Eggert, Grason, and Williams, 1972). The production version of these modules was developed by John Eggert and Peter Williams of DEC, using the name PDP-16. These differ slightly from the original RTM's, mostly for production reasons. The PDP-16 set is described in the present book, although the precise definition specification is given in DEC PDP-16 Computer Designer's Handbook, 1971.

v

[previous](#) | [contents](#) | [next](#)

[previous](#) | [contents](#) | [next](#)

[previous](#) | [contents](#) | [next](#)

to

Rod and NSF

Wes and Charlie

Ken, Stan, Fred, Al, Bill, and Bob

John, Peter, and  
Rob

Raj

Production Credits

Computer input: Mercedes Kostkas (CMU)

Computer programming: Richard Johnsson (CMU)

Cover: Elliot Hendrickson (DEC)

Drafts: Constance Dougherty, Dorothy Josephson, Geraldine Rutledge, and Mildred Sisko (CMU)

Figures: Constance Dougherty, Dorothy Josephson, Effie Lipanovich Geraldine Rutledge, and Andre Orban (CMU)

Layout Assistance: Lee Ambrose, Janice Karlton, Jennifer Grason

Layout, computer editing, and typesetting: Darlene Covaleski (CMU)

Problem proofreading: Mario Barbacci, Dileep Bhandarkar, Bob Chen, Ashok Ingli, Andre Orban, Satish Rege, Daniel Siewiorek, Jack Stephenson, and Ted Wagstaff (CMU)

Production: Dean Dombrowik, Dimitri Dimancesco, and Bob Pouliot (DEC)

Technical Editing Assistance: Heather Shoub (CMU)

[previous](#) | [contents](#) | [next](#)

[previous](#) | [contents](#) | [next](#)

Copyright © 1972

Digital Press

Digital Equipment Corporation

Library of Congress Number 72-  
89566

[previous](#) | [contents](#) | [next](#)

## [PREFACE](#)

## [CHAPTER 1 INTRODUCTION](#)

- [What Digital Systems Do and What They are Constructed.. of RT-Level Components](#)  
[Specifying System Behavior](#)  
[Summary](#)

## [CHAPTER 2 PDP-16: A SYSTEM OF RT-LEVEL MODULES \(RTM'S\)](#)

- [A First Look at RTM's](#)  
[RTM Primitives and Their Behavior](#)
  - [The RTM System](#)
    - [Four Basic Modules](#)
    - [RTM System Operation](#)
    - [Summary of Remaining Modules](#)

[The K\(Program Controlled Sequencer\PCS\),- A Microprogrammed Controller for a Centralized Encoded Control Part](#)

[Two New Modules: DM\(Arithmetic and Registers\) and K\(Bus Control and Termination Combined\)](#)

## [CHAPTER 3 ELEMENTARY DESIGN EXAMPLES](#)

- [Counting the Number of Ones in a word](#)  
[Conditional Execute Extended RTM](#)  
[Logical Shift, Circular Shift \(Rotate\), and Arithmetic Shift Operations](#)  
[For Loop Extended RTM](#)  
[Fibonacci Number Generator](#)  
[K\(Programmable \(Variable\) Clock\)](#)  
[K\(Wait-Until\) Extended RTM](#)

[D\(Square Wave Generator\)](#)

[CHAPTER 4 EXPLORING RTM DESIGN ISSUES](#)

- [Basic Design of a Multiplier](#)
  - [Variations in the Use of Multiplication](#)
    - [The Natural Mapping: Macros](#)
    - [Shared Facilities: Subroutines](#)
    - [Parallelism: Arrays](#)
    - [Parallelism: Pipelines](#)
    - [Summary](#)
  - [Variations in Implementing the Basic Algorithm](#)
  - [General Parallelism: Concurrency](#)
  - [Facility Sharing](#)
  - [Reduction of the Control Part: Unwinding Loops](#)
  - [Variations in the Algorithm](#)
  - [Alternative Representations](#)
  - [Memory versus Computation: Table Look-up](#)
  - [Alternative: The Russian Peasant's Algorithm](#)
  - [Variations in the Implementation of Control](#)
  - [K\(PCS\Programmable Command Sequencer\)](#)
  - [C\(16/M\): A Microcode Computer](#)

- [Crtm-1: An RTM Minicomputer](#)  
[Summary](#)  
[Evaluation of the Design](#)  
[Summary of the Types of Variations](#)  
[Problems](#)

## CHAPTER 5 ADVANCED DESIGN EXAMPLES

- [Data Operations Subprocesses](#)  
[Scientific \(Floating Point\) Notation](#)  
[Binary Coded Decimal \(8421 Code\) Arithmetic and Conversion](#)  
[Time Based Systems: Clocks, Timers, Waveform Generators, and Waveform Analyzers](#)  
[Sawtooth Waveform \(Ramp\) Generator](#)  
[Events Per Unit Time \(EPUT\) Meter](#)  
[Time-shared EPUT meter](#)  
[Histogram Recorder](#)  
[A Monitor to Measure the Thickness of a Coating Process](#)  
[K\(Arbiter\), an ERTM for Synchronization of Multiple Processes](#)  
[Transducers for Data Communications Systems](#)  
[Analog-To-Digital Converter](#)  
[Remote Analog-To-Digital Sampling Unit with Multiple Channel Input](#)  
[Teletrola, A Musical Terminal](#)  
[Design of Interface to a Paper Tape Punch](#)  
[Sampled Analog Input to Paper Tape Converter](#)  
[Turing Machine Simulator](#)  
[Controller for a Conveyor-Bin System](#)  
[Memories](#)  
[M\(queue\) First-in First-out Memory](#)  
[M\(stack\) Last-in First-out Memory](#)  
[M\(content addressable\)](#)

## CHAPTER 6 COMPUTER DESIGN EXAMPLES

- [Crtm-1: A Small, General Purpose, Stored Program. Computer Designed Using RTM's](#)  
[Using ISP to Define the Computer](#)  
[PDP-8/RTM Implementation](#)

[The PDP-16/M-A Subminicomputer Based on RTM's](#)

[The RTM-Computer Interfaces](#)

[RTM Computer Interface via Program Controlled Access](#)

[RTM-Minicomputer Interface via Direct Memory Access to Computer](#)

[Ps, a Special Processor to Sample Analog Input Data and Compare Against Limits](#)

[Specialized Processors for Event Counting](#)

[Clock-Calendar Interfaced to a Computer](#)

[A General Register Based Minicomputer Implementation](#)

[Simple Desk Calculator](#)

[Crtm-2/2: A Simple RTM Computer Using the DMar](#)

## [CHAPTER 7 THE DESIGN OF RTM'S](#)

[Choosing a Module Set](#)

[Switching Circuit Details](#)

■  
xii

[previous](#) | [contents](#) | [next](#)