Box 27
folder 22

3 of 3

102749950

```
From:   ROCK::SUPNIK 24-AUG-1984 11:25
To:     WAFER::UHLER
Subj:   cutler's proposed high performance architecture

From:   WILBUR::CUTLER          23-AUG-1984 13:06
To:     ROCK::SUPNIK
Subj:   RE: performance of cvax and rigel
```

Things always loose in the translation!

We said the architecture was "like" the hypervax modle so people would have some idea how it worked. Our original thoughts were to do a four stage pipeline. The first stage was decode and decoded up to 3 specifiers at a time (decode means separated). The next stage had 3 address adders capable of doing displ(rb)[rx] in one cycle. The next stage had two tb's and two caches and fetched two operands in parallel. The final stage executed instructions. And of course there was some form of branch predicition.

Now after looking at the problem more throughly and doing some analytical modeling on actual flow data from microvax one we are convinced we can get 4x780 with the same pipeline and one operand decoded, adder, and fetcher. Specifiers are decoded one at a time, the address add (or register fetch) in done in the next cycle followed by the operand read and then the value is stored in one of three operand specifier buffers that are 16 bytes wide (they also act as merge registers for unaligned data). There are two sets of operand specifier buffers; one that is being written by the fetch and one that is being read by the execution stage.

Our data says we can get from 3.8-5x780 with this architecture depending on how well we do with branches. If we predict them all incorrectly then we get 3.8x780. If we predict them all correctly we get 5x780. The middle road is about 70% correct which gives us about 4x780.

The tpi we get is from 4-5.

The reasone this architecture works is because almost all operand fetching can be hidden behind instruction execution (or at least a large amount of it can).

I believe this is a simple architecture. It is not a complicated as Venus. Write comflict is handled by write in progress bits in both the cache and register file. There is a single write bus and it need not be arbitrated. There is an icache and a dcache. They are really the same cache and share control logic. However, two reads and one write can be done every cycle. There are two register files. One is in the address add stage and one in the execution stage. The one in the address add stage is dual port read one port write. The one in the execution stage is single port read and single port write (this is one required for access to the registers during string, decimal, and certain control instructions).

Now the address adder does everything but

```
        (r)+
        -(r)
        @(r)+
        @d(r)
```

and the context indexed form of these.

The modes that are not done by the address stage cause a trap in the execution stage when the instruction becomes the current instruction. All register backup is done in the execution stage. None is done in the address add stage. Thus there is a single source of write data and it is on the result bus no matter whether it is destined for the register file or memory. No arbitration of who gets to write memory or the register file now.

Complicated instructions like string, decimal, call, etc. stop the pipe. This means that no further instructions will be decoded until the execution unit says so. Thus the fetch stage cache and tb are available for fetching and storing results. This is necessary for proper memory conflict resolution. Also mtpr and mfpr stop the pipe.

I'll stop by on Monday or Tuesday of next week and talk to you about it.

I must say that I do not agree that the way we should be trying to build a fast Vax is by tightening the microcyle time. This does not have anywhere near the payback that an improved microarchitecture has. Yes it is important but we could never get 4x780 out of cmos (currently) by going to a microcyle of 50ns or less. We would have to have so many pipeline stages that branches would kill us.

dave

```
From:   ROCK::SUPNIK 13-DEC-1984 11:53
To:     WAFER::UHLER
Subj:   request for documentation

From:   WILBUR::CUTLER          13-DEC-1984 11:53
To:     ROCK::SUPNIK
Subj:   youyr comments
```

Thanks for the comments.

Although I know very little about Rigel it seems there is significant difference. The cahnges we made in our model for branch prediction, register counters, and decoding register destinations along with the previous specifier had MAJOR impact on performance. Stopping the pipe while you wait for a branch to be resolved could cost you up to 25% in performance assuming you have a four stage pipe and branches occur every 3-4 instructions.

We will run as many traces as we can and will model things in more detail.

Could you send me some documentation about Rigel?

d

```
From:   MIST::CUTLER 21-DEC-1984 13:46
To:     WAFER::UHLER
Subj:   fortran statistics
```

```
Instruction Frequency Data
This data was collected on 21-DEC-1984 09:55:14.62
This data was written to dua0:[cutler]mm32for.cod
Total number of instructions traced was     243068
Name        Count  Percnt Cumula

MOVL        29970  12.33  12.33
BNEQ        18521   7.62  19.95
BEQL        12412   5.11  25.06
CMPL        11910   4.90  29.96
MOVAB       11624   4.78  34.74
MOVZWL       9271   3.81  38.55
ADDL2        8902   3.66  42.21
BBC          8677   3.57  45.78
MOVZBL       6848   2.82  48.60
PUSHL        6500   2.67  51.28
CMPW         6433   2.65  53.92
RET          5423   2.23  56.15
BRB          5385   2.22  58.37
CALLS        5325   2.19  60.56
SOBGTR       4919   2.02  62.58
CMPB         4900   2.02  64.60
CLRL         4809   1.98  66.58
TSTL         4223   1.74  68.32
EXTZV        4133   1.70  70.02
MOVW         4020   1.65  71.67
RSB          3904   1.61  73.28
INCL         2544   1.05  74.32
BLBC         2525   1.04  75.36
AOBLEQ       2456   1.01  76.37
BRW          2402   0.99  77.36
ASHL         2240   0.92  78.28
SOBGEQ       2163   0.89  79.17
SUBL2        2058   0.85  80.02
JSB          1813   0.75  80.76
MOVB         1811   0.75  81.51
CMPZV        1802   0.74  82.25
BSBW         1793   0.74  82.99
BBS          1739   0.72  83.70
ADDL3        1690   0.70  84.40
BGTRU        1552   0.64  85.04
MOVAL        1469   0.60  85.64
BLBS         1266   0.52  86.16
BLEQU        1252   0.52  86.68
BLSS         1251   0.51  87.19
PUSHAW       1238   0.51  87.70
CASEB        1206   0.50  88.20
MOVQ         1204   0.50  88.69
BLSSU        1191   0.49  89.18
MULL2        1188   0.49  89.67
BLEQ         1129   0.46  90.14
CASEL        1055   0.43  90.57
BGEQ         1045   0.43  91.00
CLRB          989   0.41  91.41
PUSHAB        955   0.39  91.80
CLRQ          936   0.39  92.18
SUBL3         829   0.34  92.53
BGTR          784   0.32  92.85
BGEQU         731   0.30  93.15
```

BICB2	700	0.29	93.44
INSV	668	0.27	93.71
MOVC5	624	0.26	93.97
BICL2	612	0.25	94.22
SUBW3	587	0.24	94.46
POPR	540	0.22	94.68
BSBB	520	0.21	94.90
BISB2	517	0.21	95.11
DECL	516	0.21	95.32
TSTB	515	0.21	95.53
ADDW3	514	0.21	95.75
BITB	506	0.21	95.95
PUSHR	492	0.20	96.16
MULL3	485	0.20	96.36
MNEGL	475	0.20	96.55
CLRW	470	0.19	96.74
TSTW	470	0.19	96.94
BITW	462	0.19	97.13
MOVC3	452	0.19	97.31
BISW2	438	0.18	97.49
PUSHAL	410	0.17	97.66
MOVAQ	381	0.16	97.82
MTPR	278	0.11	97.93
BICL3	263	0.11	98.04
CASEW	244	0.10	98.14
PROBER	213	0.09	98.23
BBCC	207	0.09	98.32
PROBEW	189	0.08	98.39
BISL2	189	0.08	98.47
BBSS	183	0.08	98.55
SUBB2	174	0.07	98.62
ADDW2	162	0.07	98.68
DIVL3	153	0.06	98.75
BBSC	147	0.06	98.81
BVS	144	0.06	98.87
DECW	130	0.05	98.92
SKPC	129	0.05	98.97
INCB	111	0.05	99.02
AOBLSS	109	0.04	99.06
ROTL	106	0.04	99.11
ACBL	99	0.04	99.15
REI	96	0.04	99.19
SUBB3	92	0.04	99.23
EDIV	88	0.04	99.26
BISL3	87	0.04	99.30
INCW	86	0.04	99.33
MOVPSL	86	0.04	99.37
MFPR	82	0.03	99.40
LOCC	76	0.03	99.43
JMP	71	0.03	99.46
CVTWL	67	0.03	99.49
EMUL	67	0.03	99.52
EXTV	62	0.03	99.54
PUSHAQ	56	0.02	99.57
DIVL2	55	0.02	99.59
CVTBL	53	0.02	99.61
ADDB3	51	0.02	99.63
MOVZBW	50	0.02	99.65
CHMK	50	0.02	99.67
CMPC3	49	0.02	99.69

```
CHME            49    0.02   99.71
REMQUE          47    0.02   99.73
INSQUE          46    0.02   99.75
BITL            46    0.02   99.77
CVTLB           46    0.02   99.79
MULW2           41    0.02   99.81
XORL2           39    0.02   99.82
DECB            38    0.02   99.84
MOVAW           37    0.02   99.85
SUBW2           37    0.02   99.87
ADDB2           34    0.01   99.88
BICW2           34    0.01   99.90
ASHQ            33    0.01   99.91
BISW3           25    0.01   99.92
BBCCI           24    0.01   99.93
MNEGB           21    0.01   99.94
CVTLW           21    0.01   99.95
FFS             18    0.01   99.96
BISPSW          17    0.01   99.96
BVC             12    0.00   99.97
MOVTC           12    0.00   99.97
CALLG           11    0.00   99.98
ACBW             9    0.00   99.98
BICB3            9    0.00   99.98
MCOML            9    0.00   99.99
CVTWB            8    0.00   99.99
CMPV             7    0.00   99.99
XORL3            4    0.00  100.00
BBCS             4    0.00  100.00
BICW3            2    0.00  100.00
BBSSI            2    0.00  100.00
BISB3            1    0.00  100.00
MULW3            1    0.00  100.00
MNEGW            1    0.00  100.00
```

```
Instruction Size
Size        Count  Percnt Cumula

    1        9423   3.88   3.88
    2       62829  25.85  29.73
    3       63082  25.95  55.68
    4       48813  20.08  75.76
    5       23001   9.46  85.22
    6       10212   4.20  89.42
    7       16744   6.89  96.31
    8        7508   3.09  99.40
    9        1180   0.49  99.89
   10          72   0.03  99.92
   11          73   0.03  99.95
   12          43   0.02  99.96
   13          84   0.03 100.00
   14           1   0.00 100.00
   15           0   0.00 100.00
   16           3   0.00 100.00
   17           0   0.00 100.00
   18           0   0.00 100.00
   19           0   0.00 100.00
   20           0   0.00 100.00
   21           0   0.00 100.00
   22           0   0.00 100.00
   23           0   0.00 100.00
   24           0   0.00 100.00
   25           0   0.00 100.00
   26           0   0.00 100.00
   27           0   0.00 100.00
   28           0   0.00 100.00
   29           0   0.00 100.00
   30           0   0.00 100.00
   31           0   0.00 100.00
   32           0   0.00 100.00
   33           0   0.00 100.00
   34           0   0.00 100.00
   35           0   0.00 100.00
   36           0   0.00 100.00
   37           0   0.00 100.00
   38           0   0.00 100.00
   39           0   0.00 100.00
   40           0   0.00 100.00
Average Instruction Size =  3.65
```

```
Specifier Size
Size       Count  Percnt Cumula

    1     321707  74.95  74.95
    2      62904  14.66  89.61
    3      13744   3.20  92.81
    4       3409   0.79  93.61
    5      21449   5.00  98.60
    6       5998   1.40 100.00
Average Specifier Size =  1.50
Specifier Type (all)
Type       Count  Percnt Cumula

s^#0x      51034  11.89  11.89
s^#1x       9678   2.25  14.15
s^#2x       2282   0.53  14.68
s^#3x        933   0.22  14.89
[Rx]       21053   4.91  19.80
Rn        147093  34.27  54.07
(Rb)       32407   7.55  61.62
-(Rb)       2051   0.48  62.10
(Rb)+      15176   3.54  65.63
@(Rb)+      2125   0.50  66.13
b^(Rb)     46918  10.93  77.06
@b(Rb)      1775   0.41  77.47
w^(Rb)      5100   1.19  78.66
@w(Rb)        12   0.00  78.66
l^(Rb)     16769   3.91  82.57
@l(Rb)       152   0.04  82.61
Bdb        70350  16.39  99.00
Bdw         4303   1.00 100.00
Specifier Type (index)
Type       Count  Percnt Cumula

(Rb)        8308  39.46  39.46
-(Rb)          2   0.01  39.47
(Rb)+          0   0.00  39.47
@(Rb)+       133   0.63  40.10
b^(Rb)      2956  14.04  54.14
@b(Rb)       395   1.88  56.02
w^(Rb)      3276  15.56  71.58
@w(Rb)       133   0.63  72.21
l^(Rb)      5835  27.72  99.93
@l(Rb)        15   0.07 100.00
```

```
Memory Reads Per Instruction
Number     Count  Percnt Cumula

     0    169894  69.90  69.90
     1     64075  26.36  96.26
     2      8774   3.61  99.87
     3       292   0.12  99.99
     4         0   0.00  99.99
     5        33   0.01 100.00
     6         0   0.00 100.00
Average Memory Reads Per Instruction  =  0.34
Memory Writes Per Instruction
Number     Count  Percnt Cumula

     0    216590  89.11  89.11
     1     26478  10.89 100.00
     2         0   0.00 100.00
Average Memory Writes Per Instruction  =  0.11
Register Reads Per Instruction
Number     Count  Percnt Cumula

     0     89350  36.76  36.76
     1    108651  44.70  81.46
     2     40567  16.69  98.15
     3      4135   1.70  99.85
     4       365   0.15 100.00
     5         0   0.00 100.00
     6         0   0.00 100.00
     7         0   0.00 100.00
     8         0   0.00 100.00
     9         0   0.00 100.00
    10         0   0.00 100.00
    11         0   0.00 100.00
    12         0   0.00 100.00
Average Register Reads Per Instruction  =  0.84
Register Writes Per Instruction
Number     Count  Percnt Cumula

     0    154840  63.70  63.70
     1     88141  36.26  99.96
     2        87   0.04 100.00
Average Register Writes Per Instruction  =  0.36
```

```
Specifier Access Type
Type      Count  Percnt Cumula

read      195659  45.59  45.59
write      86433  20.14  65.72
modify     28360   6.61  72.33
addres     26433   6.16  78.49
vield      17673   4.12  82.61
branch     74653  17.39 100.00
Total number of operand specifiers was     429211
Number of nonfetch operand specifiers was     333944
Percent of nonfetch operand specifiers was  77.80
```

```
From:   MIST::CUTLER 21-DEC-1984 13:50
To:     WAFER::UHLER
Subj:   linker statistics
```

```
Instruction Frequency Data
This data was collected on 21-DEC-1984 10:00:30.14
This data was written to dua0:[cutler]mm32lnk.cod
Total number of instructions traced was     333074
Name         Count  Percnt Cumula

MOVL         51250  15.39  15.39
ADDL2        16023   4.81  20.20
MOVZBL       15514   4.66  24.86
BEQL         13224   3.97  28.83
MOVAB        13079   3.93  32.75
CMPL         11899   3.57  36.32
BLBC          8973   2.69  39.02
AOBLEQ        8514   2.56  41.58
BNEQ          7765   2.33  43.91
BBC           7469   2.24  46.15
MOVZWL        7452   2.24  48.39
CLRL          7000   2.10  50.49
BGTRU         6953   2.09  52.58
BBS           6787   2.04  54.61
RSB           6716   2.02  56.63
BRB           6078   1.82  58.45
PUSHL         6024   1.81  60.26
CMPB          5973   1.79  62.06
TSTL          5572   1.67  63.73
BLSSU         5201   1.56  65.29
RET           4552   1.37  66.66
CALLS         4318   1.30  67.95
BSBW          4159   1.25  69.20
EXTZV         4055   1.22  70.42
SUBL2         3822   1.15  71.57
BLBS          3692   1.11  72.68
MOVB          3602   1.08  73.76
INSV          3366   1.01  74.77
ADDL3         3301   0.99  75.76
MOVW          3222   0.97  76.73
MOVAL         3184   0.96  77.68
INCL          3119   0.94  78.62
SUBL3         2748   0.83  79.44
PUSHAB        2706   0.81  80.26
BGEQ          2338   0.70  80.96
PUSHR         2157   0.65  81.61
CMPW          2138   0.64  82.25
BGTR          2121   0.64  82.88
BRW           2085   0.63  83.51
POPR          2032   0.61  84.12
BSBB          1996   0.60  84.72
MOVQ          1932   0.58  85.30
BLEQ          1809   0.54  85.84
BLSS          1783   0.54  86.38
SOBGTR        1768   0.53  86.91
BICL2         1717   0.52  87.42
BGEQU         1675   0.50  87.93
BICB3         1492   0.45  88.38
CLRQ          1454   0.44  88.81
INCW          1425   0.43  89.24
MCOML         1402   0.42  89.66
MTPR          1396   0.42  90.08
BICL3         1294   0.39  90.47
```

BISB2	1260	0.38	90.85
JSB	1242	0.37	91.22
SOBGEQ	1098	0.33	91.55
BLEQU	1040	0.31	91.86
CASEW	1039	0.31	92.17
CLRW	1010	0.30	92.48
PROBEW	975	0.29	92.77
MOVC3	966	0.29	93.06
MOVC5	963	0.29	93.35
CMPC5	936	0.28	93.63
BICB2	934	0.28	93.91
ADDW2	916	0.28	94.18
ASHL	843	0.25	94.44
MNEGL	806	0.24	94.68
EDIV	714	0.21	94.89
MOVAQ	710	0.21	95.11
PROBER	673	0.20	95.31
TSTW	673	0.20	95.51
EMUL	664	0.20	95.71
BBCC	650	0.20	95.91
ROTL	611	0.18	96.09
DECL	584	0.18	96.26
MOVAW	572	0.17	96.44
DECW	539	0.16	96.60
BISW3	525	0.16	96.76
BBSS	525	0.16	96.91
EXTV	487	0.15	97.06
TSTB	433	0.13	97.19
BISL2	424	0.13	97.32
CLRB	396	0.12	97.44
ACBW	358	0.11	97.54
BBSC	353	0.11	97.65
CMPZV	299	0.09	97.74
BICW2	296	0.09	97.83
REI	294	0.09	97.92
SUBB3	292	0.09	98.00
CASEB	290	0.09	98.09
BISL3	289	0.09	98.18
MOVPSL	275	0.08	98.26
MFPR	268	0.08	98.34
INCB	263	0.08	98.42
JMP	256	0.08	98.50
REMQUE	255	0.08	98.57
CVTBL	253	0.08	98.65
CVTWL	231	0.07	98.72
INSQUE	221	0.07	98.78
PUSHAL	217	0.07	98.85
BVS	213	0.06	98.91
BISW2	209	0.06	98.98
PUSHAQ	208	0.06	99.04
CHMK	205	0.06	99.10
CMPV	185	0.06	99.16
MOVZBW	176	0.05	99.21
BITW	171	0.05	99.26
MULL2	129	0.04	99.30
DECB	125	0.04	99.34
SUBW3	123	0.04	99.37
ADDB3	118	0.04	99.41
CVTLB	118	0.04	99.44
MULW2	114	0.03	99.48

```
XORL2       114   0.03   99.51
SBWC        112   0.03   99.55
CHME        111   0.03   99.58
DIVL2       109   0.03   99.61
XORB2       105   0.03   99.64
BITB         94   0.03   99.67
ASHQ         84   0.03   99.70
AOBLSS       84   0.03   99.72
BBCCI        82   0.02   99.75
SUBW2        81   0.02   99.77
BITL         76   0.02   99.79
ADDW3        74   0.02   99.82
MULL3        69   0.02   99.84
FFS          69   0.02   99.86
BBCS         62   0.02   99.88
LOCC         55   0.02   99.89
CMPC3        53   0.02   99.91
ADDB2        49   0.01   99.92
PUSHAW       37   0.01   99.94
SUBB2        35   0.01   99.95
MNEGB        32   0.01   99.96
BICW3        31   0.01   99.96
XORL3        28   0.01   99.97
CASEL        19   0.01   99.98
MULW3        13   0.00   99.98
SKPC         11   0.00   99.99
DIVL3        11   0.00   99.99
BBSSI        11   0.00   99.99
CALLG         8   0.00   99.99
MNEGW         6   0.00  100.00
ACBL          3   0.00  100.00
MATCHC        2   0.00  100.00
BISB3         2   0.00  100.00
DIVW3         2   0.00  100.00
ADWC          2   0.00  100.00
```

```
 Instruction Size
 Size        Count  Percnt Cumula

      1      11562   3.47   3.47
      2      77185  23.17  26.64
      3     104811  31.47  58.11
      4      75220  22.58  80.70
      5      33308  10.00  90.70
      6       9373   2.81  93.51
      7      13379   4.02  97.53
      8       6726   2.02  99.55
      9       1391   0.42  99.96
     10         85   0.03  99.99
     11         22   0.01 100.00
     12         11   0.00 100.00
     13          1   0.00 100.00
     14          0   0.00 100.00
     15          0   0.00 100.00
     16          0   0.00 100.00
     17          0   0.00 100.00
     18          0   0.00 100.00
     19          0   0.00 100.00
     20          0   0.00 100.00
     21          0   0.00 100.00
     22          0   0.00 100.00
     23          0   0.00 100.00
     24          0   0.00 100.00
     25          0   0.00 100.00
     26          0   0.00 100.00
     27          0   0.00 100.00
     28          0   0.00 100.00
     29          0   0.00 100.00
     30          0   0.00 100.00
     31          0   0.00 100.00
     32          0   0.00 100.00
     33          0   0.00 100.00
     34          0   0.00 100.00
     35          0   0.00 100.00
     36          0   0.00 100.00
     37          0   0.00 100.00
     38          0   0.00 100.00
     39          0   0.00 100.00
     40          0   0.00 100.00
Average Instruction Size =  3.50
```

```
Specifier Size
Size       Count  Percnt Cumula

     1    479696  78.23  78.23
     2     93126  15.19  93.42
     3     15041   2.45  95.87
     4      5995   0.98  96.85
     5     17226   2.81  99.66
     6      2106   0.34 100.00
Average Specifier Size =  1.36
Specifier Type (all)
Type       Count  Percnt Cumula

s^#0x      63534  10.36  10.36
s^#1x       9913   1.62  11.98
s^#2x       3544   0.58  12.56
s^#3x       2018   0.33  12.88
[Rx]       13473   2.20  15.08
Rn        239508  39.06  54.14
(Rb)       51483   8.40  62.54
-(Rb)       3175   0.52  63.06
(Rb)+      24665   4.02  67.08
@(Rb)+      2550   0.42  67.49
b^(Rb)     74334  12.12  79.62
@b(Rb)      6557   1.07  80.69
w^(Rb)      8316   1.36  82.04
@w(Rb)        78   0.01  82.05
l^(Rb)     10908   1.78  83.83
@l(Rb)       265   0.04  83.88
Bdb        92264  15.05  98.92
Bdw         6605   1.08 100.00
Specifier Type (index)
Type       Count  Percnt Cumula

(Rb)        2461  18.27  18.27
-(Rb)         31   0.23  18.50
(Rb)+          0   0.00  18.50
@(Rb)+       404   3.00  21.49
b^(Rb)      1818  13.49  34.99
@b(Rb)      1071   7.95  42.94
w^(Rb)      2043  15.16  58.10
@w(Rb)      3865  28.69  86.79
l^(Rb)      1023   7.59  94.38
@l(Rb)       757   5.62 100.00
```

```
Memory Reads Per Instruction
Number      Count  Percnt Cumula

     0     222070  66.67  66.67
     1      99420  29.85  96.52
     2      11324   3.40  99.92
     3        260   0.08 100.00
     4          0   0.00 100.00
     5          0   0.00 100.00
     6          0   0.00 100.00
Average Memory Reads Per Instruction  =  0.37
Memory Writes Per Instruction
Number      Count  Percnt Cumula

     0     291474  87.51  87.51
     1      41600  12.49 100.00
     2          0   0.00 100.00
Average Memory Writes Per Instruction  =  0.12
Register Reads Per Instruction
Number      Count  Percnt Cumula

     0     111687  33.53  33.53
     1     145748  43.76  77.29
     2      68862  20.67  97.97
     3       5611   1.68  99.65
     4       1166   0.35 100.00
     5          0   0.00 100.00
     6          0   0.00 100.00
     7          0   0.00 100.00
     8          0   0.00 100.00
     9          0   0.00 100.00
    10          0   0.00 100.00
    11          0   0.00 100.00
    12          0   0.00 100.00
Average Register Reads Per Instruction  =  0.92
Register Writes Per Instruction
Number      Count  Percnt Cumula

     0     198184  59.50  59.50
     1     134197  40.29  99.79
     2        693   0.21 100.00
Average Register Writes Per Instruction  =  0.41
```

```
Specifier Access Type
Type      Count  Percnt Cumula

read      277942  45.33  45.33
write     132852  21.67  66.99
modify     44331   7.23  74.22
addres     34796   5.67  79.90
vield      24400   3.98  83.88
branch     98869  16.12 100.00
Total number of operand specifiers was     613190
Number of nonfetch operand specifiers was     476819
Percent of nonfetch operand specifiers was  77.76
```

```
From:   MIST::CUTLER 21-DEC-1984 13:55
To:     WAFER::UHLER
Subj:   sort statistics
```

```
Instruction Frequency Data
This data was collected on 21-DEC-1984 10:05:51.15
This data was written to dua0:[cutler]sort.cod
Total number of instructions traced was      402831
Name        Count  Percnt Cumula

MOVL        51261  12.73  12.73
CMPL        23039   5.72  18.44
BBC         20659   5.13  23.57
ADDL2       20420   5.07  28.64
BLSS        18915   4.70  33.34
BLSSU       18339   4.55  37.89
SUBL3       17640   4.38  42.27
RSB         16367   4.06  46.33
BNEQ        16216   4.03  50.36
BLBC        13632   3.38  53.74
MOVZWL      12864   3.19  56.94
EXTZV       12280   3.05  59.98
TSTL        11491   2.85  62.84
BEQL        11400   2.83  65.67
PUSHL       11352   2.82  68.48
MOVAL       10611   2.63  71.12
JSB          9885   2.45  73.57
DECL         8224   2.04  75.61
SOBGEQ       8138   2.02  77.63
BSBW         4674   1.16  78.79
MOVZBL       4066   1.01  79.80
BBS          3970   0.99  80.79
ADDL3        3704   0.92  81.71
CMPB         3538   0.88  82.59
BRB          3496   0.87  83.45
INCL         2876   0.71  84.17
CLRL         2656   0.66  84.83
POPR         2652   0.66  85.49
MTPR         2333   0.58  86.07
BSBB         2279   0.57  86.63
BRW          2254   0.56  87.19
BLBS         2069   0.51  87.70
MOVAB        2040   0.51  88.21
EXTV         1846   0.46  88.67
CMPW         1609   0.40  89.07
PUSHR        1533   0.38  89.45
PUSHAB       1420   0.35  89.80
BGEQU        1413   0.35  90.15
MOVW         1399   0.35  90.50
MOVB         1362   0.34  90.84
BGTRU        1355   0.34  91.17
CASEB        1352   0.34  91.51
SOBGTR       1316   0.33  91.84
SUBL2        1294   0.32  92.16
CMPZV        1180   0.29  92.45
BICB2        1105   0.27  92.72
MOVPSL       1017   0.25  92.98
MFPR         1009   0.25  93.23
BBSC         1002   0.25  93.48
BGTR          993   0.25  93.72
RET           966   0.24  93.96
BGEQ          957   0.24  94.20
BBCC          929   0.23  94.43
```

MOVQ	909	0.23	94.66
MOVC3	843	0.21	94.87
ROTL	838	0.21	95.07
BLEQU	821	0.20	95.28
BBSS	800	0.20	95.48
CASEW	782	0.19	95.67
ASHL	763	0.19	95.86
CLRQ	743	0.18	96.04
TSTB	742	0.18	96.23
PROBER	714	0.18	96.41
CMPC3	703	0.17	96.58
MNEGL	690	0.17	96.75
BLEQ	656	0.16	96.91
CALLS	642	0.16	97.07
CLRW	591	0.15	97.22
ADDW2	587	0.15	97.37
PROBEW	492	0.12	97.49
TSTW	469	0.12	97.60
BICL3	467	0.12	97.72
DECW	439	0.11	97.83
MULL3	430	0.11	97.94
BISB2	400	0.10	98.04
AOBLSS	360	0.09	98.13
CVTWL	357	0.09	98.21
SUBW3	345	0.09	98.30
REI	328	0.08	98.38
BISW2	283	0.07	98.45
BICL2	260	0.06	98.52
BVS	255	0.06	98.58
CLRB	252	0.06	98.64
INCW	251	0.06	98.70
ADDB3	239	0.06	98.76
INSQUE	233	0.06	98.82
INSV	228	0.06	98.88
CVTLB	226	0.06	98.93
MOVC5	225	0.06	98.99
MULW2	218	0.05	99.04
CHME	213	0.05	99.10
REMQUE	212	0.05	99.15
AOBLEQ	185	0.05	99.19
CMPV	163	0.04	99.24
BITB	158	0.04	99.27
BITL	149	0.04	99.31
BICB3	147	0.04	99.35
SUBW2	146	0.04	99.38
DECB	139	0.03	99.42
SUBB3	129	0.03	99.45
BITW	125	0.03	99.48
BISL3	122	0.03	99.51
CHMK	116	0.03	99.54
EDIV	113	0.03	99.57
ADDW3	110	0.03	99.60
EMUL	108	0.03	99.62
JMP	107	0.03	99.65
CVTBL	103	0.03	99.68
INCB	98	0.02	99.70
XORL3	93	0.02	99.72
MOVAQ	89	0.02	99.74
BISB3	85	0.02	99.77
PUSHAQ	77	0.02	99.79

PUSHAL	77	0.02	99.80
MOVZBW	75	0.02	99.82
BICW2	72	0.02	99.84
MULL2	72	0.02	99.86
BISL2	67	0.02	99.88
BBCCI	57	0.01	99.89
MOVAW	56	0.01	99.90
XORL2	56	0.01	99.92
ASHQ	46	0.01	99.93
LOCC	36	0.01	99.94
FFS	33	0.01	99.95
ACBW	32	0.01	99.95
ADDB2	31	0.01	99.96
DIVL2	20	0.00	99.97
MNEGB	15	0.00	99.97
BBCS	15	0.00	99.97
CASEL	14	0.00	99.98
PUSHAW	12	0.00	99.98
DIVL3	12	0.00	99.98
CALLG	12	0.00	99.99
MCOML	11	0.00	99.99
BBSSI	11	0.00	99.99
SUBB2	7	0.00	99.99
XORB2	6	0.00	99.99
BISW3	6	0.00	100.00
CVTLW	3	0.00	100.00
SVPCTX	2	0.00	100.00
CMPC5	2	0.00	100.00
BICW3	2	0.00	100.00
MNEGW	2	0.00	100.00
BVC	1	0.00	100.00
SKPC	1	0.00	100.00
MULW3	1	0.00	100.00
FFC	1	0.00	100.00

```
 Instruction Size
 Size        Count  Percnt Cumula

      1      17663   4.38   4.38
      2     100032  24.83  29.22
      3     101448  25.18  54.40
      4      81518  20.24  74.64
      5      59734  14.83  89.47
      6      15616   3.88  93.34
      7      21382   5.31  98.65
      8       4120   1.02  99.67
      9       1017   0.25  99.93
     10        100   0.02  99.95
     11        186   0.05 100.00
     12         14   0.00 100.00
     13          0   0.00 100.00
     14          1   0.00 100.00
     15          0   0.00 100.00
     16          0   0.00 100.00
     17          0   0.00 100.00
     18          0   0.00 100.00
     19          0   0.00 100.00
     20          0   0.00 100.00
     21          0   0.00 100.00
     22          0   0.00 100.00
     23          0   0.00 100.00
     24          0   0.00 100.00
     25          0   0.00 100.00
     26          0   0.00 100.00
     27          0   0.00 100.00
     28          0   0.00 100.00
     29          0   0.00 100.00
     30          0   0.00 100.00
     31          0   0.00 100.00
     32          0   0.00 100.00
     33          0   0.00 100.00
     34          0   0.00 100.00
     35          0   0.00 100.00
     36          0   0.00 100.00
     37          0   0.00 100.00
     38          0   0.00 100.00
     39          0   0.00 100.00
     40          0   0.00 100.00
Average Instruction Size =   3.56
```

```
Specifier Size
Size        Count  Percnt Cumula

      1     533131  74.08  74.08
      2     121259  16.85  90.93
      3      33007   4.59  95.52
      4       3910   0.54  96.06
      5      27499   3.82  99.89
      6        827   0.11 100.00
Average Specifier Size =   1.44
Specifier Type (all)
Type        Count  Percnt Cumula

s^#0x        70885   9.85   9.85
s^#1x        30401   4.22  14.07
s^#2x         2755   0.38  14.46
s^#3x         1025   0.14  14.60
[Rx]         24937   3.47  18.07
Rn          257110  35.73  53.79
(Rb)         24352   3.38  57.18
-(Rb)        11533   1.60  58.78
(Rb)+        12923   1.80  60.58
@(Rb)+       18811   2.61  63.19
b^(Rb)      100904  14.02  77.21
@b(Rb)        8720   1.21  78.42
w^(Rb)       10986   1.53  79.95
@w(Rb)          35   0.00  79.95
l^(Rb)        6836   0.95  80.90
@l(Rb)         221   0.03  80.93
Bdb         130239  18.10  99.03
Bdw           6960   0.97 100.00
Specifier Type (index)
Type        Count  Percnt Cumula

(Rb)          1030   4.13   4.13
-(Rb)            2   0.01   4.14
(Rb)+            0   0.00   4.14
@(Rb)+         389   1.56   5.70
b^(Rb)        1084   4.35  10.05
@b(Rb)       18025  72.28  82.33
w^(Rb)         986   3.95  86.28
@w(Rb)        2922  11.72  98.00
l^(Rb)         469   1.88  99.88
@l(Rb)          30   0.12 100.00
```

```
Memory Reads Per Instruction
Number     Count  Percnt Cumula

     0     258897  64.27  64.27
     1     121574  30.18  94.45
     2      22176   5.51  99.95
     3        182   0.05 100.00
     4          2   0.00 100.00
     5          0   0.00 100.00
     6          0   0.00 100.00
Average Memory Reads Per Instruction  =  0.41
Memory Writes Per Instruction
Number     Count  Percnt Cumula

     0     355111  88.15  88.15
     1      47720  11.85 100.00
     2          0   0.00 100.00
Average Memory Writes Per Instruction  =  0.12
Register Reads Per Instruction
Number     Count  Percnt Cumula

     0     154528  38.36  38.36
     1     175647  43.60  81.96
     2      54253  13.47  95.43
     3      18331   4.55  99.98
     4         72   0.02 100.00
     5          0   0.00 100.00
     6          0   0.00 100.00
     7          0   0.00 100.00
     8          0   0.00 100.00
     9          0   0.00 100.00
    10          0   0.00 100.00
    11          0   0.00 100.00
    12          0   0.00 100.00
Average Register Reads Per Instruction  =  0.84
Register Writes Per Instruction
Number     Count  Percnt Cumula

     0     271266  67.34  67.34
     1     131456  32.63  99.97
     2        109   0.03 100.00
Average Register Writes Per Instruction  =  0.33
```

```
Specifier Access Type
Type      Count  Percnt Cumula

read      329359  45.77  45.77
write     132292  18.38  64.15
modify     47102   6.55  70.70
addres     30507   4.24  74.94
vield      43174   6.00  80.93
branch    137199  19.07 100.00
Total number of operand specifiers was    719633
Number of nonfetch operand specifiers was    546693
Percent of nonfetch operand specifiers was  75.97
```

```
From:   ROCK::SUPNIK  2-JAN-1985 12:46
To:     WAFER::UHLER
Subj:   fyi

From:   WILBUR::CUTLER          2-JAN-1985 11:41
To:     ROCK::SUPNIK
Subj:   RE: update on rigel modelling
```

Let me respond to a couple of things:

1. 1k x 1 does indeed give > than 80% in most cases.

2. Branch prediction is good!

3. I have converted you traces and run them. I also changed the way strings are modelled to take into account the lengths of the strings. And you'll be happy to knwo that we also see 6-7 TPI. BUT these are only very small excerpts from the overall traces. They are VERY heavily weighted toward string and call return instructions. We have run the entire trace of the linker (not with system space however) and it is much better. If we look at the numbr of page faults and direct I/O's and multiply by 1ms and add the simulation time we still come out 6x780 for a complete liner trace!

4. You may not see any benefit for register counters since the traces you are using are so small. The greatest benefit we see is in Fortran programs. I still think this is a very important optimization and we intend to leave it in.

Thanks for the comments. I'll keep yopu posted on what we are doing. My spelling above is terrible (I never think to edit these things when I start).

dave

```
From:   MIST::SCHNORR              5-DEC-1984 05:06
To:     ULTRA::STRECKER,ROCK::SUPNIK,NOODLE::PARKER
Subj:   Frigate Simulator Description
```

1 OVERVIEW

The Frigate simulator consists of two programs; one to generate a trace file and one that reads the trace file and simulates the Frigate hardware pipeline. It should be kept in mind that the simulator does not actually execute programs. Rather it computes the number of cycles that would be required to execute the program on a Frigate machine.

The trace program is linked as a debugger with the program to be traced. It then gains control before the subject program and solicits what the name of the output file is to be and how many disk blocks of data are to be collected. The subject program is then traced and a data file is written that contains the opcodes, operand specifiers, and branch destinations of the executed instructions. Specifier displacements and immediate data are not written into the output file since they are not required by the simulator. At the end of the subject program or when the specified number of disk blocks of data have been collected an end of data sentinel is written and the data file is closed. The trace program then formats and prints instruction frequency, instruction size, specifier size, specifier type, memory read, memory write, register read and register write data.

The second part of the simulator is the program that simulates the actual hardware. This program allows several parameters such as data cache miss rate and branch prediction counter width to be specified and then reads the data file produced by the trace program. The simulator consists of five subroutines that simulate the individual pipeline stages and a short control program that calls each of the stage subroutines for each machine cycle. Instructions are prefetched, decoded, their operands fetched and then executed. Each activity proceeds in a pipelined fashion until it reaches the execution stage where it spends the number of cycles it takes to execute the respective instruction. Instructions are executed in this manner until the entire data file has been read. At the end of the simulation, statistics are output as to the number of cycles that were executed, the number of instructions executed, several branch statistics and data on the utilization of the pipeline stages.

2 INSTRUCTION CLASSIFICATION

All VAX instructions are classified into groups depending on how their execution affects pipeline activity. The intent is to have as few classes as possible and still execute the VAX instruction set efficiently. Class information will be stored in a ROM (or RAM) that is accessed using the instruction opcode value. The resultant information is then used to control pipeline operation while the instruction executes.

Eight instruction classes are defined:

1. Stop Decode - This instruction class inhibits the Decode stage from decoding further instructions. Explicit continuation from the execution unit is required before subsequent instructions will be decoded. The remaining specifiers for the subject instruction are decoded. Instructions in this class change global machine state (e.g. MTPR), interact with FPD (e.g. MOVC3), implicitly modify registers or contain multiple write destinations (e.g. EDIV).

   Instructions in this class include:

HALT	CVTPT	MOVP	EDIV	ASHP
REI	MULP	CMPP3	CASEB	CVTLP
BPT	CVTTP	CVTPL	CASEW	CALLG
RET	DIVP	CMPP4	POPR	CALLS
RSB	MOVC3	EDITPC	PUSHR	XFC
CVTPS	CMPC3	MATCHC	CHMK	ESCE
CVTSP	SPANC	LOCC	CHME	ESCF
CRC	SCANC	SKPC	CHMS	EMODG
ADDP4	MOVC5	EMODF	CHMU	POLYG
ADDP6	CMPC5	POLYF	CASEL	EMODH
SUBP4	MOVTC	EMODD	MTPR	POLYH
SUBP6	MOVTUC	POLYD	MFPR	LDPCTX
SVPCTX				

   These instructions take several cycles to execute and are generally infrequent. Note that RSB is also in the implied pop class.

2. Stop Fetch - This instruction class stops the Operand stage in the same way as the Decode stage is stopped by the previous class. Explicit continuation is required by the execution unit before further instruction operands will be fetched. These instructions read or modify destinations whose addresses cannot be calculated by the Operand stage (e.g. BBSS).

   This class includes:

ADAWI	INSQHI	REMQTI	BBSC	BBCCI
INSQUE	INSQTI	BBSS	BBCC	INSV
REMQUE	REMQHI	BBCS	BBSSI	BBC
BBS	EXTV	EXTZV	CMPV	CMPZV
FFS	FFC			

   Note that the branch on bit instructions in this class are also in the conditional branch class.

3. Conditional Branch - This instruction class conditionally branches to a destination based on source or condition code values. A subset of the instructions also modify the source value. The execution of these instructions is predicted in the Decode stage. If a branch is predicted to be taken then the destination address is computed by the Decode stage and

passed to the Prefetch stage.

This class includes:

BNEQ	BLSS	BGEQU	BBCS	BLBS
BEQL	BGTRU	BLSSU	BBSC	BLBC
BGTR	BLEQU	BBS	BBCC	
BLEQ	BVS	BBC	BBSSI	
BGEQ	BVC	BBSS	BBCCI	

Note that the branch on bit instructions that modify their source are also included in the stop fetch class.

4. Loop - This instruction class includes all the iterative loop instructions. This class is similar to the conditional branch class but differs in that the branches are always predicted to be taken. The branch destination address is computed by the Decode stage and passed to the Prefetch stage.

This class includes:

ACBB	ACBL	ACBD	AOBLSS	SOBGTR
ACBW	ACBF	ACBG	AOBLEQ	SOBGEQ
ACBH				

5. Unconditional Branch - This instruction class includes all the instructions that unconditionally branch to an address that can be calculated in the Decode or Address stage. The destination address is calculated in the Decode stage if it is PC relative and in the Address stage if it is indirect, context indexed or not relative to PC. The resultant address is passed to the Prefetch stage.

This class includes:

BSBB	JSB	BSBW
BRB	JMP	BRW

Note that BSBB, BSBW, and JSB are also in the implied push class.

6. Implied Push - This instruction class generates an implied push onto the stack after the final operand has been processed. This requires the decode stage to generate an autodecrement SP operand specifier.

This instruction class includes:

BSBB	JSB	PUSHAW	PUSHAQ	PUSHL
BSBW	PUSHAB	PUSHAL	PUSHAO	

Note that BSBB, BSBW, and JSB are also in the unconditional branch class.

7. Implied Pop - This class contains only the instruction RSB. The Decode stage generates an autoincrement SP operand specifier to remove the return address from the top of the stack.

   This class includes:

   RSB

   RSB is also in the stop decode class.

8. General - This instruction class contains all instructions that require no special processing. They cause no pipeline hazards and can be processed in a pipelined manner without any interference.

   This class includes:

NOP	CVTHD	INDEX	PROBER	PROBEW
CVTWL	CVTWB	MOVZWL	MOVAW	ADDF2
ADDF3	SUBF2	SUBF3	MULF2	MULF3
DIVF2	DIVF3	CVTFB	CVTFW	CVTFL
CVTRFL	CVTBF	CVTWF	CVTLF	MOVF
CMPF	MNEGF	TSTF	CVTFD	ADDD2
ADDD3	SUBD2	SUBD3	MULD2	MULD3
DIVD2	DIVD3	CVTDB	CVTDW	CVTDL
CVTRDL	CVTBD	CVTWD	CVTLD	MOVD
CMPD	MNEGD	TSTD	CVTDF	ASHL
ASHQ	EMUL	CLRQ	MOVQ	MOVAQ
ADDB2	ADDB3	SUBB2	SUBB3	MULB2
MULB3	DIVB2	DIVB3	BISB2	BISB3
BICB2	BICB3	XORB2	XORB3	MNEGB
MOVB	CMPB	MCOMB	BITB	CLRB
TSTB	INCB	DECB	CVTBL	CVTBW
MOVZBL	MOVZBW	ROTL	MOVAB	ADDW2
ADDW3	SUBW2	SUBW3	MULW2	MULW3
DIVW2	DIVW3	BISW2	BISW3	BICW2
BICW3	XORW2	XORW3	MNEGW	MOVW
CMPW	MCOMW	BITW	CLRW	TSTW
INCW	DECW	BISPSW	BICPSW	ADDL2
ADDL3	SUBL2	SUBL3	MULL2	MULL3
DIVL2	DIVL3	BISL2	BISL3	BICL2
BICL3	XORL2	XORL3	MNEGL	MOVL
CMPL	MCOML	BITL	CLRL	TSTL
INCL	DECL	ADWC	SBWC	MOVPSL
MOVAL	CVTHF	CVTFG	CVTFH	MOVAO
MOVO	CLRO	CVTLB	CVTLW	CVTDH
CVTGF	ADDG2	ADDG3	SUBG2	SUBG3
MULG2	MULG3	DIVG2	DIVG3	CVTGB
CVTGW	CVTGL	CVTRGL	CVTBG	CVTWG
CVTLG	MOVG	CMPG	MNEGG	TSTG
CVTGH	ADDH2	ADDH3	SUBH2	SUBH3
MULH2	MULH3	DIVH2	DIVH3	CVTHB
CVTHW	CVTHL	CVTRHL	CVTBH	CVTWH
CVTLH	MOVH	CMPH	MNEGH	TSTH

3 GENERAL OPERATION

3.1 Pipeline Activity

Pipeline stages take input from a previous stage, perform some transformation and produce output which is input for the next stage. If there is no input to act upon a stage is said to be idle; that is the previous stage produced no output and therefore there is nothing for the stage to do. If the next stage does not consume the data provided to it in a previous cycle then the preceding stage in the pipeline stalls.

In summary, a pipeline stage is idle if no data is provided for it to process and a pipeline stage stalls if the subsequent stage does not process previously provided data.

3.2 Instruction Cache And Translation Buffer

A separate instruction cache and translation buffer are used to access the instruction stream. All cache modeling done at DECwest suggests that the instruction stream cache hit rate will be upwards of 99%. In addition, two levels of buffering are employed. Therefore the instruction stream is not modeled explicitly. It is assumed that any instruction cache or translation buffer misses will have a minimal affect on performance.

There is one aspect of the Prefetch stage that is modeled that has to do with branches. The model keeps track of the virtual PC even though it does not use it to access the instruction translation buffer and cache. The virtual PC is used to determine if a translation buffer access is required when a branch takes place. In effect the virtual page number of the new PC is compared with the virtual page number of the old PC. If the comparison fails then an additional cycle is required to do the translation buffer access. Otherwise the page offset is concatenated with the previous physical page number to access the instruction cache. The reason for this is that it is not possible to access the translation buffer, access the instruction cache and write the prefetch buffer in one cycle.

3.3 Data Cache And Translation Buffer

A separate translation buffer and cache are used to fetch data from memory and write results. The organization of the cache is the write back scheme being used in Firefly at SRC. (The scheme employs two extra bits per cache line to keep track of shared and modified data. The shared bit indicates whether it is possible that the data might also be in another cache. The modified bit indicates whether the data has been modified but not written to memory.) There is no problem with cache coherence even in a multiprocessor configuration. DECwest modeling of the effects of write back caches suggests that memory write traffic can be cut by 60-70%. Thus it is possible to build a system without heavy demands for memory bus bandwidth.

VMS (and other operating systems) will not require any special code to manage the write back cache. The cache will be entirely transparent except when the power fails. An internal processor register will be provided so that VMS can sweep the cache and force all unwritten data to memory.

Modeling of the data translation buffer and cache is on the basis of how often a miss occurs and how many cycles it takes to process the miss. This data cache miss rate and the forced write rate are provided as parameters to the model.

3.4 Register File Write In Progress Counters

Four logical copies of the general purpose registers are maintained. Two of the copies reside in the Decode stage and two in the execution stage. These two copies can be thought of as dual port read single port write RAMs.

Register values that are needed for address calculation (base or index register values) are read from the RAMs located in the Decode stage and the value(s) is (are) passed to the Address stage. Register mode operands however are not actually read until the execution stage. This allows a major optimization with regard to allowing outstanding writes against register mode operands to be ignored since when the instruction reaches the execution stage the respective register will by definition have the most current value.

Associated with each register number (excluding PC) is a Write in Progress Counter (4 bits) that is maintained by the Decode stage and which records the number of writes outstanding against the register. A counter is used so that it is possible to have multiple writes against a register outstanding (as opposed to a single bit which would allow only one outstanding write). It is not intuitive that multiple outstanding writes are a common occurrence, but consider the following instruction sequence:

```
        ADDL3   R0, R1, R2
        ADDL2   R3, R2
```

When the ADDL2 instruction is decoded register R2 will already have an outstanding write against it. But since R2 will actually have the correct value when the ADDL2 reaches the execution stage it is expedient to allow multiple writes against R2. This turns out to be a very common code sequence generated by most of our compilers.

The Decode stage monitors the write bus for register values and updates its copies of the registers when appropriate. It also decrements the respective Write in Progress Counter.

Register File Write in Progress Counters are not updated for register mode operand specifiers if instruction decode has been stopped. This is necessary to avoid deadlock for instructions with multiple write destinations (e.g. EDIV R0, R1, R2, (R2)). Write in Progress Counters, however, are always updated for autoincrement and autodecrement operand specifiers even when instruction decode is stopped. The Execution stage resumes pipeline activity when a consistent state has been reached.

3.5 Data Cache Write In Progress Bits

Each cache line in the data cache has associated with it a Write in Progress Bit. This bit is written by the Operand stage when a write destination operand is processed. The bit is cleared when a write on the write bus to the affected location is executed.

Write in Progress Bits allow subsequent operand reads to occur after a write has been processed if the read is from a different location than the write. If it is not, the Operand stage will stall until the Write in Progress Bit has been cleared.

Data Cache Write in Progress Bits are not updated if instruction decode has been stopped. This is necessary to avoid deadlock for instructions with multiple write destinations (e.g. EDIV R0, R1, (R2), @(R2)). The Execution stage resumes pipeline activity when a consistent state has been reached.

3.6 Autoincrement And Autodecrement Operand Specifiers

The processing of autoincrement and autodecrement operand specifiers requires the cooperation of the Decode, Address, Operand and Execution stages of the pipeline.

The Decode stage increments the respective Write in Progress Counter and passes the register number, register contents, autodecrement value (if required), and function to be performed to the Address stage. The Address stage computes the effective address and passes the register number, effective address, and function to be performed to the Operand stage. The Operand stage first passes the register number and autoincrement or autodecrement operation to the Execute state and then fetches the operand value in the next cycle if required (i.e. not address or yield access type). The Execution stage performs the autoincrement or autodecrement function and writes the new register value into its own copy of the Register File and on the write bus. The Decode stage then picks the value off the write bus, decrements the corresponding Write in Progress Counter, and writes the new value into its copy of the Register File.

All told, the processing of an autoincrement or autodecrement operand specifier takes one additional cycle. Subsequent decoding of operand specifiers is not blocked unless the register being autoincremented or autodecremented is the one required as a base or index register for a subsequent operand.

The actual autoincrement or autodecrement in the Execution stage is performed in the "context" of the instruction to which it belongs; that is, it is performed after the previous instruction has completed and before the next instruction has started execution. The previous contents of the respective register are saved in a register log when the Execution stage performs the increment or decrement operation. This register log is cleared at the end of an instruction.

3.7 Indirect Addressing

Indirect addressing requires two trips through the Address and Operand stages. The Execution stage gets involved only to pass the indirect address through the ALU and into the write latch. The Address stage then picks the address off the write bus, adds the index register if specified, and passes the effective address back to the Operand stage. The Execution unit always processes indirect addresses in the "context" of the instruction to which it belongs.

Indirect addressing occurs very infrequently (e.g. approximately 1-2% of the time) and is not expected to adversely affect performance.

4 PIPELINE OPERATION AND STAGES

The Frigate pipeline is executed by a control program that calls the stage subroutines in reverse order. This is necessary to propagate stalls correctly since each stage is not actually executed in parallel as it would be in real hardware.

This sequence, although it works nicely for stalls, causes problems when the output of a latter pipeline stage is to be acted upon by an earlier pipeline stage in the next cycle. The problem is that the earlier pipeline stage executes after the latter pipeline stage in the current cycle.

An example is register file writes on the write bus which must be recognized by the Decode and Address stages in the next cycle, not the current. The data written by the Execution stage in the current cycle is processed by the Decode and Address stages in the next cycle and the data written by the Execution stage in the previous cycle is processed by the Decode and Address stages in the current cycle. The model accomplishes this by inserting pipeline variables that delay the recognition of data until a subsequent cycle. In the case of register file writes this is done with a 2 deep array of register numbers and valid flags. All pipeline stages examine the first member of the array and the Execution stage writes the second member. At the end of each simulated cycle the second array member is copied to the first and the second is set invalid. Other pipeline variables are simply implemented as boolean variables. For example, when a new virtual instruction PC is available to the Prefetch stage, both the value and a flag called "prefetch_new_address" are set. The Prefetch stage sees the flag, copies the new virtual address and then clears the flag. In the next cycle it will actually start delivering the instruction information at the destination address.

4.1 Prefetch

The Prefetch stage reads the input file produced by the trace program and provides the Decode stage with opcode and operand specifier information on each cycle. This information is the actual opcode and operand specifier data, including register numbers, that was collected when the subject program was traced. As Prefetch reads the trace file it updates the virtual instruction PC by computing the length of each operand specifier. Two byte opcodes require one extra cycle to deliver the "escape" opcode to the Decode stage.

If the Decode stage has not processed the data delivered to it during a previous cycle then Prefetch stalls.

If a new virtual PC has been delivered by the Decode, Address or Execution stage then the old virtual PC's virtual page number is compared with the new virtual PC's page number and the new virtual PC is copied to the old virtual PC. If the virtual page numbers match (i.e. a translation buffer access is not required) the instruction data at the target address will be delivered to the Decode stage in the next cycle. If the page numbers do not match then a translation buffer cycle is required in the next cycle and the Prefetch stage will deliver the instruction data in the cycle after that.

It should be noted that the affects of Instruction Cache and Translation buffer misses are not modeled. It is assumed that this causes minimal degradation in performance.

If instruction decode is stopped by the Decode stage then Prefetch accumulates idle time when a new opcode is to be decoded. Subsequent operand specifiers for the current instruction are delivered until an instruction boundary is reached.

If the Decode stage has not processed data delivered in a previous cycle then Prefetch accumulates stall time.

If a new virtual PC is provided by one of the Decode, Address, or Execution stages the Prefetch accumulates wait time until a translation buffer and/or cache access can be done.

In all other cases Prefetch can do useful work and accumulates work time.

4.2 Decode

The Decode stage processes the opcode and operand specifier information provided by the Prefetch stage. It maintains two copies of the Register File and the associated Write in Progress Counters, determines when information is to be passed to the Address stage, predicts if conditional branches will be taken, and transmits the destination address for loop and unconditional branches and jumps to the Prefetch stage.

The Decode stage is modeled as a finite state machine with 4 states. The states are:

1. Process opcode and first operand specifier if any,

2. Process next operand specifier, branch destination, or jump address,

3. Execute implied push/pop to/from the current stack, and

4. Finish unconditional branch or jump instruction.

Decode starts at state 1 and cycles in state 2 if necessary until all operand specifiers have been processed. If required (PUSHL, RSB, ...) state 3 or 4 is entered to finish the instruction and then back to state 1 for the next opcode.

Decode always interrogates the write bus at the start of a cycle to determine if a register value is being written. If the write bus is valid (i.e. there is a register being written) then the respective register's Write in Progress Counter is decremented but never below zero. A decrement below zero could happen when a multiple write destination instruction has stopped the pipeline and a register mode destination was present. In this case the Write in Progress Counter was not incremented to avoid possible deadlock and when the instruction is finished the counter must not be decremented.

If the Address stage has not processed data from a previous cycle then the Decode stage stalls. Otherwise the operation associated with the current state is performed.

1. State 1 - Process opcode and first specifier if any.

   If no data has been supplied by the Prefetch stage (i.e. it is waiting for a translation buffer or cache access after a new virtual PC has been received and cannot deliver any data) or instruction decode has been stopped, then the Decode stage idles.

   If the opcode is a two byte opcode then the first byte is accepted in the current cycle and the two byte opcode flag is set. The second opcode byte will be delivered in the next cycle.

If the opcode is from the stop decode class or is an unconditional branch or jump the stop decode flag is set.

If the opcode has zero specifiers and is from the implied pop class, then the state number is set to 3 and the implied pop is executed immediately. Otherwise the opcode and a no operation function are passed to the Address stage. No operation implies that the Address and Operand stages perform no operation for the respective opcode.

If the opcode has one or more specifiers then the state number is set to 2, the specifier number is set to 1, and the first operand specifier is processed immediately if possible.

2. State 2 - Process next operand specifier, branch destination, or jump address.

   If no data has been supplied by the Prefetch stage then the Decode stage idles. Otherwise an action is executed depending on the specifier mode and register number. Specifier actions include:

   1. Modes 0, 1, 2, and 3 (short literal) - The short literal value, the specifier datatype, the specifier access type and a function of literal are passed to the Address stage. If the instruction is from the implied push class then the state number is set to 3. The autodecrement SP specifier will be generated in the next cycle. All instructions from the implied push class have exactly one operand specifier. If the instruction is not from the implied push class and this is the last operand specifier then the state number is set to 1. The next cycle will process the next opcode.

   2. Mode 4 (index) - If the index register or the base register (note there must always be a base register since short literals cannot be indexed) has a Write in Progress Count greater than 1 or both registers have a Write in Progress Count of 1, then the Decode stage waits for one of the registers to be written on the write bus. At that time the Write in Progress Counter will be adjusted. Otherwise the index register invalid flag is set equal to the value of the respective Write in Progress Counter. If the Write in Progress Counter is not zero (i.e. there are outstanding writes against the register but only one) then the Address stage will pick the value off the write bus when it is written. Index mode present is set and the base address is processed by executing its action routine.

   3. Mode 5 (register mode) - If the access type is write or modify and decoding of instructions is not stopped, then the Write in Progress Counters for the respective register(s) are incremented (note that up to 4 counters could be incremented). The register number, the

specifier datatype, the specifier access type and a function of register are passed to the Address stage. If the instruction is from the implied push class then the state number is set to 3. The autodecrement SP specifier will be generated in the next cycle. If the instruction is not from the implied push class and this is the last operand specifier then the state number is set to 1. The next cycle will process the next opcode.

4. Modes 6, 10, 12, and 14 (register deferred and byte, word and longword displaced) - If the Write in Progress Counter for the base register is greater than 1 then the Decode stage waits for the register to be written on the write bus. At that time the Write in Progress Counter will be adjusted. Otherwise if the opcode is from the unconditional branch class, an index register is not specified and the base register is PC, then the branch destination can be calculated immediately and sent to the Prefetch stage. If the unconditional branch is from the implied push class then the state number is set to 3 and the implied push is executed immediately. Otherwise the state number is set to 4 and the unconditional branch is finished. If the opcode is from the unconditional branch class and either an index register is specified or the base register is not PC, then the base register invalid flag is set to the value of the respective Write in Progress Counter and the branch destination flag is set. The base register invalid flag, the base register number, the base register value, the index register invalid flag, the index register number, the displacement value, the specifier datatype, the specifier access type and a function of fetch are passed to the Address stage. If the instruction is from the implied push class, then the state number is set to 3. Otherwise it is set to 4. The autodecrement SP specifier will be generated or the unconditional branch finished in the next cycle. If the instruction is not from the unconditional branch class, then the base register invalid flag is set to the value of the respective Write in Progress Counter. The base register invalid flag, the base register number, the base register value, the index register invalid flag, the index register number, the displacement value, and a function of fetch are passed to the Address stage. If the instruction is from the implied push class then the state is set to 3. The autodecrement SP specifier will be generated in the next cycle. If the instruction is not from the implied push class and this is the last operand specifier, then the state number is set to 1. The next cycle will process the next opcode.

5. Modes 7 and 8 (autoincrement and autodecrement) - If the base register is PC (only possible for mode 8 since autodecrement PC is illegal), then the action for mode 6 is executed. This is immediate mode addressing and the Operand stage actually fetches the immediate value. If

the base register is not PC and the respective Write in Progress Counter is greater than 1, then the Decode stage waits for the register to be written on the write bus. If the base register Write in Progress Counter is 0 or 1, then the base register invalid flag is set to the value of the Write in Progress Counter. The base register invalid flag, the base register number, the base register value, the index register invalid flag, the index register number, the specifier datatype, the specifier access type and a function of modify are passed to the Address stage. The Write in Progress Counter for the base register is incremented. If the opcode is from the unconditional branch and implied push classes, then the state number is set to 3 and the branch destination flag is set. The autodecrement SP specifier will be generated in the next cycle. If the opcode is from the unconditional class and not the implied push class, then the state number is set to 4 and the branch destination flag is set. The unconditional branch will be finished in the next cycle. If the instruction is from the implied push class and not the unconditional branch class, then the state number is set to 3. The autodecrement SP specifier will be generated in the next cycle. If the instruction is not from the unconditional branch or implied push class and this is the last specifier, then the state number is set to 1. The next cycle will process the next opcode.

6. Mode 9 (autoincrement deferred) - If the base register is PC, then the action for mode 6 is executed. This is absolute addressing and the address is treated like a longword displacement with no base register. If the base register is not PC and the respective Write in Progress Counter is greater than 1, the Decode stage waits for the register to be written on the write bus. If the base register Write in Progress Counter is 0 or 1, then the base register invalid flag is set to the value of the Write in Progress Counter. The base register invalid flag, the base register number, the base register value, the index register invalid flag, the index register number, the specifier datatype, the specifier access type and a function of indirect modify are passed to the Address stage. The Write in Progess Counter for the base register is incremented. If the opcode is from the unconditional branch and implied push classes, then the state number is set to 3 and the branch destination flag is set. The autodecrement SP specifier will be generated in the next cycle. If the opcode is from the unconditional branch class and not the implied push class, then the state number is set to 4 and the branch destination flag is set. The unconditional branch will be completed in the next cycle. If the instruction is from the implied push class but not from the unconditional branch class, then the state number is set to 3. The autodecrement SP specifier will be generated

in the next cycle. If the instruction is not from the unconditional branch or implied push class and this is the last specifier, then the state number is set to 1. The next cycle will process the next instruction.

7. Modes 11, 13, and 15 (indirect byte, word, and longword displaced) - If the Write in Progress Counter for the base register is greater than 1, then the Decode stage waits for the register to be written on the write bus. If the base register Write in Progress Counter is 0 or 1, then the base register invalid flag is set to the value of the Write in Progress Counter. The base register invalid flag, the base register number, the base register value, the index register invalid flag, the index register number, the displacement value, the specifier datatype, the specifier access type and a function of indirect fetch are passed to the Address stage. If the opcode is from the unconditional branch and implied push classes, then state number is set to 3 and the branch destination flag is set. The autodecrement SP specifier will be generated in the next cycle. If the opcode is from the unconditional branch class and not from the implied push class, then the state number is set to 4 and the branch destination flag is set. The unconditional branch will be finished the next cycle. If the instruction is from the implied push class and not from the unconditional branch class, then the state number is set to 3. The autodecrement SP specifier will be generated in the next cycle. If the instruction is not from the unconditional branch or implied push class and this is the last specifier, then the state number is set to 1. The next cycle will process the next instruction.

8. Branch Displacement (byte and word) - If the opcode is from the unconditional branch class, then the destination address is sent to the Prefetch stage and the stop decode flag is cleared. If the opcode is from the unconditional branch and implied push class then the state number is set to 3. The autodecrement SP specifier will be generated in the next cycle. If the opcode is from the uncondition branch class and not the implied push class, then the state number is set to 4. The unconditional branch will be completed in the next cycle. If the instruction is not from the unconditional branch class, then it is either from the conditional branch or loop class. The base register and index register register invalid flags are set false. The base register invalid flag, index register invalid flag, the specifier datatype, the specifier access type, computed destination address and a function of displacement are passed to the Address stage. The state number is set to 1. If the opcode is from the loop class, then it is always predicted as taken. The computed destination address is sent to the Prefetch stage. If the opcode is from the conditional branch class, then the branch prediction RAM

is accessed using bits 2 through 15 of the ending address of the conditional branch instruction itself (this is actually the address of the next instruction). If the high order bit of the prediction value is set, then the branch is predicted as taken. Otherwise it is predicted as not taken. If the branch is predicted taken, then the computed destination address is sent to the Prefetch stage. The prediction flag is passed to the Address stage. The next instruction is processed in the next cycle.

3. State 3 - Execute implied push/pop to/from the current stack.

   If the Write in Progress Counter for SP is greater than 1, then the Decode stage waits for the register to be written on the write bus. If the Write in Progress Counter is 0 or 1, then the base register invalid flag is set to the value of the Write in Progress Counter. If the opcode is from the implied pop class, then the specifier access is set to read. Otherwise the opcode is from the implied push class and the specifier access is set to write. The base register invalid flag, the register number 14, the specifier datatype (always longword), the specifier access type and a function of modify are passed to the Address stage. The state number is set to 1 and the Write in Progress Counter for SP is incremented. The next opcode will be processed in the next cycle.

4. State 4 - Finish unconditional branch or jump instruction.

   Unconditional branches and jumps, although executed in the Decode stage, cannot be evaporated. They must continue through to the Execution stage so that trace traps can occur if enabled (this may be eliminated later by sending the opcode through the pipe if and only if t-bit is set or the address mode is autoincrement, autodecrement or autodecrement deferred). The branch destination flag is set false. A function code of displacement is passed to the Address stage.

The Decode stage closely simulates what the actual hardware will do. It is believed to be very accurate.

If instruction decoding is stopped by the Decode stage itself or no input has been provided by the Prefetch stage, then the Decode stage accumulates idle time.

If the Address stage has not processed data delivered to it in a previous cycle, then the Decode stage accumulates stall time.

If the Write in Progress Count of a base register or index register is greater than 1, or both a base register and an index register are specified and their respective Write in Progress Counters are equal to 1, then the Decode stage accumulates wait time.

In all other cases Decode can do useful work and accumulates work time.

4.3 Address

The Address stage computes the effective address of an operand or passes through the data it receives to the Operand stage. It is capable of performing a 3 input add in one cycle (i.e. displacement, base register, and context shifted index register) and operates from a function and data supplied by the Decode stage. This stage processes indirect addressing.

The Address stage always interrogates the write bus at the start of a cycle to determine if a register value is being written that matches an invalid register that it requires to perform the address computation. Either the base or index register may be required but not both. The base register and index register invalid flags are used for this purpose. If the write bus specifies a partial write (i.e. byte or word), then the value is merged with the value passed to the Address stage by the Decode stage. Note that there can only be one outstanding write at this time. This is guaranteed by Decode and therefore the first write on the write bus that matches the invalid register number is the one required to complete the address calculation. As soon as the corresponding register value has been received the respective invalid flag is cleared.

If the Operand stage has not processed data from a previous cycle then the Address stage stalls.

If no data has been supplied by the Decode stage, then the Address stage idles.

If the Address stage has not yet received an indirect address from the Execution stage, then the Address stage waits.

If the base register or index register invalid flags are set, then the Address stage waits. Otherwise an action is performed according to the function specified by the Decode stage. Address stage actions include:

1. Displacement - The branch destination address is computed by adding the sign extended branch displacement with the displacement PC (the PC is provided through special logic that adjusts for the length of the branch displacement). The destination address and a function of displacement are passed to the Operand stage.

2. Fetch and Modify - The effective address is computed. If the branch destination flag is set, then the effective address is sent to the Prefetch stage and the stop decode flag is cleared. If the original function was fetch, then no further processing is necessary. Otherwise the effective address, the base register number, the specifier datatype, the specifier access type and a function of fetch or modify are

passed to the Operand stage.

3. Register - The base register number, the specifier datatype, the specifier access type and a function of register are passed to the Operand stage.

4. Literal - The literal value, the specifier datatype, the specifier access type and a function code of literal are passed to the Operand stage.

5. No Operation - A no operation function is passed to the Operand stage.

6. Indirect Fetch and Indirect Modify - The effective base address is calculated. This calculation does not include the index register if it is present. The context shifted contents of the index register will be added to the indirect address when it is received from the Execution stage. The indirect flag is set, the operand access type is set to read and the operand datatype is set to longword. The indirect flag will cause the Address stage to wait in subsequent cycles until cleared by the Execution stage. The effective address, the base register number, the specifier datatype, the specifier access type and a function of indirect fetch or indirect modify are passed to the Operand stage. When the indirect address is received from the Execution stage on the write bus the Address stage will add the context shifted index register, if any, to the indirect address and then pass the original specifier datatype, the original specifier access type and a function of fetch to the Operand stage. Note that while the Address stage is waiting for an indirect address the Decode stage is stalled because it cannot deliver new data to the Address stage.

4.4 Operand

The Operand stage reads operand values from memory, checks the validity of write destinations, increments register numbers, manages the Write in Progress Bits in the data cache and delivers subsequent zero longwords for short literals. It is also responsible for assigning pointer register numbers to address the operand buffers. The Operand stage operates from data and a function code passed to it by the Address stage.

It is worth explaining the function of the pointer registers and operand buffers although they are not actually modeled in the simulation. Three pointer register FIFO's are used to store pointer registers and operand status. One is for odd numbered source specifiers, one for even numbered source specifiers, and one for destinations. Each cycle, one of the source FIFO's and the destination FIFO can be written with a register number. The number that is written is either a general register number, an operand buffer number, or an address buffer number. Operand buffers are used to hold operand values. Address buffers are used to hold the physical address of the destination (this is always available since a translation buffer access is performed on the destination operand to determine if it is accessible). If the destination address crosses a page boundary, then two address buffer entries are used (i.e. two translation buffer accesses are required). A status code is also included with each register. The status code indicates if there was an access violation, translation not valid, translation buffer miss, a modify refuse (write access to a page that does not have the modify bit set) or an attempt was made to read an I/O address. (This latter type is a VERY sticky problem in a pipelined machine. Since we have such a wonderful I/O architecture that allows people to build devices where register reads have side effects we have to guarantee that I/O addresses are read exactly once. The way this is done is to dump the address of the operand into the allocated operand buffer and tag the pointer register with a status code that will cause the Execution stage to dispatch to a routine that will explicitly read the I/O address.) The Execution stage microword provides control over the reading of the pointer register FIFO's. For example, the microword for an ADDL3 would read the register numbers from the source 1, source 2, and destination FIFO's. It should be noted that ADDL2 would do the same thing since the modify source/destination operand would cause both the even numbered source FIFO and the destination FIFO to be written. The net effect is to allow the Operand stage to fetch operands somewhat ahead of the Execution stage (e.g. some elasticity is provided by the address and operand buffers) and provides parametric microcode in the Execution stage.

If no data has been supplied by the Address stage, then the Operand stage idles.

If there are no address or operand buffers available and one is required the Operand stage stalls until the Execution stage has emptied one.

If operand fetching has been stopped, then the Operand stage idles.

The Operand stage executes an action determined by the function code supplied to it by the Address stage. The following actions are performed:

1. No Operation - No operation is performed. In the simulation model this takes one cycle. In the real machine this will not require any cycles because the opcode dispatch information is kept separately from the operand specifier information. In the model it is not. This function is used for zero specifier opcodes.

2. Literal - The first longword of a short literal value is formatted (e.g. zero extended, shifted, bits inserted, etc.) in the Decode stage and passed through the Address stage. The Operand stage assigns an operand buffer and stores the first longword of the short literal in the buffer. If the context is quadword or octaword, then the Operand stage will

allocate additional operand buffers and deliver zero longwords in subsequent cycles. One cycle is required for each longword.

3. Fetch - If the access type is address or yield, then an operand buffer is allocated and the value received from the Address stage is written into the buffer. If the access type is read, write, or modify, then a translation buffer access must be performed to determine the physical address and accessibility of the operand (the translation buffer is not modeled). For each longword in the operand (and each longword takes at least one cycle) a random number is generated and compared with the data cache miss rate that was selected when the simulation was begun. If the random number is less than or equal to the data cache miss rate then 5 additional cycles (the cache fill time) will be spent fetching the operand value. If the random number is greater than the data cache miss rate then only 1 cycle is required to fetch the operand value. If the data cache misses there is also a probability that the location that will be displaced from the cache has been modified but not written. Another random number is generated and compared with the forced write rate which was also selected when the simulation was begun. If found to "miss" then 5 additional cycles are required to first write the current contents of the cache line and then read the new value. This amounts to 11 cycles in all if the data cache misses and a forced write is required. As longwords are fetched an operand buffer is allocated, the register number written into one or more of the FIFO's and the operand value placed in the operand buffer.

4. Modify - The register number of the general register that is to be modified is written into the appropriate FIFO with a status that encodes the context and whether the operation to be performed is an autoincrement or autodecrement. The Execution stage will dispatch to a routine that actually performs the operation when an attempt is made to read the respective FIFO. In the next cycle a fetch function is performed.

5. Indirect fetch - The indirect longword address is read and written into an operand buffer. The register number of the operand buffer and a status code that indicates that the operand buffer contains an indirect address is written into the operand buffer. The Execution stage will dispatch to a routine that writes the indirect address on the write bus when an attempt is made to read the respective FIFO.

6. Indirect modify - The register number of the general register that is to be updated is written into the appropriate FIFO with a status code that encodes a context of longword and autoincrement. In the next cycle an indirect fetch is performed.

7. Register - The general register number is written into the appropriate pointer FIFO. If the operand specifier is context quadword or octaword, then the register number is incremented in successive cycles and written into the same FIFO.

8. Displacement - An operand buffer is allocated and the displacement value is stored.

4.5 Execute

The Execution stage executes instructions, performs autoincrement and autodecrement operations, and writes indirect addresses on the write bus. The Execution stage closely models the real hardware but of course computes no answers. It does, however, keep track of which registers are being written by an instruction and at the end of instruction execution it writes the registers one per cycle on the write bus.

The Execution stage is modeled as a finite state machine with 4 states. The states are:

1. Dispatch

2. Execute

3. Clean Up

4. Register Write

Execute starts at state 1 and cycles there until an opcode and all its specifiers arrive. During the time that it waits it can perform autoincrements, autodecrements and send indirect addresses to the Address stage which each take 1 cycle. This is not exactly how the hardware will work but there should be no difference in the performance. The hardware actually starts the instruction early and then ends up waiting if an operand is not ready.

When a complete instruction has arrived state 2 is entered where the number of cycles estimated for the instruction are spent.

State 3 releases operand buffers, continues pipeline activity, and sends branch addresses to the prefetch stage. If a register value is to be written state 4 is entered. In state 4 a register value is transmitted on the write bus every cycle.

The Execution stage always performs the action associated with the current state.

1. State 1 - Dispatch. If there are no operand buffers that contain operands, then the Execute stage idles. Otherwise the operand buffers are examined one at a time to determine

if a complete instruction is present or there are autoincrements, autodecrements, or indirect addresses to process. If an autoincrement or autodecrement is found, then the operand buffer is removed and the register number is written on the write bus. This consumes the entire cycle. If an indirect address is found, then the operand buffer is removed and the indirect address is written on the write bus to signal the Address stage that the address it is waiting for is present (and thank God indirect addresses are infrequent - the whole pipeline is backed up while we sequence 1 indirect address through the pipeline). This also consumes the entire cycle. If no autoincrments, autodecrements, or indirect addresses are found before a complete instruction has been assembled then the state number is set to 2 and the cycle counter is set to the number of cycles the instruction is estimated to take. Estimates used in the simulator are as close to reality as possible since we must make judgements about the final performance of the actual hardware. State 2 is executed.

2. State 2 - Execute. The number of cycles remaining is decremented. If the result is nonzero, then the instruction execution is not complete. The number of cycles will be decremented again in the next cycle. If the remaining cycles is zero, then instruction execution is complete. If the instruction has no specifiers and is not from the implied pop class but is from the decode stop or fetch stop classes, then either the Decode or Operand stage is continued. If the opcode also caused a branch (e.g. REI) then a new PC is sent to the Prefetch stage. If the instruction has one or more specifiers or is from the implied pop class, then the state number is set to 3 and state 3 is executed.

3. State 3 - Clean Up. The operand buffers are released. If the instruction writes a destination register, then the state number is set to 4 and state 4 is executed. Otherwise if the instruction is from the decode or fetch stop classes, then either the Decode or Operand stage is continued. If the opcode also caused a branch (e.g. CHMK) then a new PC is sent to the Prefetch stage. If the instruction is from the conditional branch or loop classes and the branch was not predicted correctly, then the entire pipeline is flushed, the correct PC is sent to the Prefetch stage and the branch prediction RAM is updated. The state number is set to 1.

4. State 4 - Register Write. The destination register number is written on the write bus and the number of registers remaining to write is decremented. If the result is zero, then the state number is set to 3 and state 3 is executed. If the result is nonzero, then register number is incremented. The next register will be written in the next cycle.

[end of fb.rno]

1 OVERVIEW

The Frigate simulator consists of two programs; one to generate a trace file and one that reads the trace file and simulates the Frigate hardware pipeline. It should be kept in mind that the simulator does not actually execute programs. Rather it computes the number of cycles that would be required to execute the program on a Frigate machine.

The trace program is linked as a debugger with the program to be traced. It then gains control before the subject program and solicits what the name of the output file is to be and how many disk blocks of data are to be collected. The subject program is then traced and a data file is written that contains the opcodes, operand specifiers, and branch destinations of the executed instructions. Specifier displacements and immediate data are not written into the output file since they are not required by the simulator. At the end of the subject program or when the specified number of disk blocks of data have been collected an end of data sentinel is written and the data file is closed. The trace program then formats and prints instruction frequency, instruction size, specifier size, specifier type, memory read, memory write, register read and register write data.

The second part of the simulator is the program that simulates the actual hardware. This program allows several parameters such as data cache miss rate and branch prediction counter width to be specified and then reads the data file produced by the trace program. The simulator consists of five subroutines that simulate the individual pipeline stages and a short control program that calls each of the stage subroutines for each machine cycle. Instructions are prefetched, decoded, their operands fetched and then executed. Each activity proceeds in a pipelined fashion until it reaches the execution stage where it spends the number of cycles it takes to execute the respective instruction. Instructions are executed in this manner until the entire data file has been read. At the end of the simulation, statistics are output as to the number of cycles that were executed, the number of instructions executed, several branch statistics and data on the utilization of the pipeline stages.

2 INSTRUCTION CLASSIFICATION

All VAX instructions are classified into groups depending on how their execution affects pipeline activity. The intent is to have as few classes as possible and still execute the VAX instruction set efficiently. Class information will be stored in a ROM (or RAM) that is accessed using the instruction opcode value. The resultant information is then used to control pipeline operation while the instruction executes.

Eight instruction classes are defined:

1. Stop Decode - This instruction class inhibits the Decode stage from decoding further instructions. Explicit continuation from the execution unit is required before subsequent instructions will be decoded. The remaining specifiers for the subject instruction are decoded. Instructions in this class change global machine state (e.g. MTPR), interact with FPD (e.g. MOVC3), implicitly modify registers or contain multiple write destinations (e.g. EDIV).

   Instructions in this class include:

HALT	CVTPT	MOVP	EDIV	ASHP
REI	MULP	CMPP3	CASEB	CVTLP
BPT	CVTTP	CVTPL	CASEW	CALLG
RET	DIVP	CMPP4	POPR	CALLS
RSB	MOVC3	EDITPC	PUSHR	XFC
CVTPS	CMPC3	MATCHC	CHMK	ESCE
CVTSP	SPANC	LOCC	CHME	ESCF
CRC	SCANC	SKPC	CHMS	EMODG
ADDP4	MOVC5	EMODF	CHMU	POLYG
ADDP6	CMPC5	POLYF	CASEL	EMODH
SUBP4	MOVTC	EMODD	MTPR	POLYH
SUBP6	MOVTUC	POLYD	MFPR	LDPCTX
SVPCTX				

   These instructions take several cycles to execute and are generally infrequent. Note that RSB is also in the implied pop class.

2. Stop Fetch - This instruction class stops the Operand stage in the same way as the Decode stage is stopped by the previous class. Explicit continuation is required by the execution unit before further instruction operands will be fetched. These instructions read or modify destinations whose addresses cannot be calculated by the Operand stage (e.g. BBSS).

   This class includes:

ADAWI	INSQHI	REMQTI	BBSC	BBCCI
INSQUE	INSQTI	BBSS	BBCC	INSV
REMQUE	REMQHI	BBCS	BBSSI	BBC
BBS	EXTV	EXTZV	CMPV	CMPZV
FFS	FFC			

   Note that the branch on bit instructions in this class are also in the conditional branch class.

3. Conditional Branch - This instruction class conditionally branches to a destination based on source or condition code values. A subset of the instructions also modify the source value. The execution of these instructions is predicted in the Decode stage. If a branch is predicted to be taken then the destination address is computed by the Decode stage and

passed to the Prefetch stage.

This class includes:

BNEQ	BLSS	BGEQU	BBCS	BLBS
BEQL	BGTRU	BLSSU	BBSC	BLBC
BGTR	BLEQU	BBS	BBCC	
BLEQ	BVS	BBC	BBSSI	
BGEQ	BVC	BBSS	BBCCI	

Note that the branch on bit instructions that modify their source are also included in the stop fetch class.

4. Loop - This instruction class includes all the iterative loop instructions. This class is similar to the conditional branch class but differs in that the branches are always predicted to be taken. The branch destination address is computed by the Decode stage and passed to the Prefetch stage.

   This class includes:

ACBB	ACBL	ACBD	AOBLSS	SOBGTR
ACBW	ACBF	ACBG	AOBLEQ	SOBGEQ
ACBH				

5. Unconditional Branch - This instruction class includes all the instructions that unconditionally branch to an address that can be calculated in the Decode or Address stage. The destination address is calculated in the Decode stage if it is PC relative and in the Address stage if it is indirect, context indexed or not relative to PC. The resultant address is passed to the Prefetch stage.

   This class includes:

BSBB	JSB	BSBW
BRB	JMP	BRW

   Note that BSBB, BSBW, and JSB are also in the implied push class.

6. Implied Push - This instruction class generates an implied push onto the stack after the final operand has been processed. This requires the decode stage to generate an autodecrement SP operand specifier.

   This instruction class includes:

BSBB	JSB	PUSHAW	PUSHAQ	PUSHL
BSBW	PUSHAB	PUSHAL	PUSHAO	

   Note that BSBB, BSBW, and JSB are also in the unconditional branch class.

7. Implied Pop - This class contains only the instruction RSB. The Decode stage generates an autoincrement SP operand specifier to remove the return address from the top of the stack.

   This class includes:

   RSB

   RSB is also in the stop decode class.

8. General - This instruction class contains all instructions that require no special processing. They cause no pipeline hazards and can be processed in a pipelined manner without any interference.

   This class includes:

NOP	CVTHD	INDEX	PROBER	PROBEW
CVTWL	CVTWB	MOVZWL	MOVAW	ADDF2
ADDF3	SUBF2	SUBF3	MULF2	MULF3
DIVF2	DIVF3	CVTFB	CVTFW	CVTFL
CVTRFL	CVTBF	CVTWF	CVTLF	MOVF
CMPF	MNEGF	TSTF	CVTFD	ADDD2
ADDD3	SUBD2	SUBD3	MULD2	MULD3
DIVD2	DIVD3	CVTDB	CVTDW	CVTDL
CVTRDL	CVTBD	CVTWD	CVTLD	MOVD
CMPD	MNEGD	TSTD	CVTDF	ASHL
ASHQ	EMUL	CLRQ	MOVQ	MOVAQ
ADDB2	ADDB3	SUBB2	SUBB3	MULB2
MULB3	DIVB2	DIVB3	BISB2	BISB3
BICB2	BICB3	XORB2	XORB3	MNEGB
MOVB	CMPB	MCOMB	BITB	CLRB
TSTB	INCB	DECB	CVTBL	CVTBW
MOVZBL	MOVZBW	ROTL	MOVAB	ADDW2
ADDW3	SUBW2	SUBW3	MULW2	MULW3
DIVW2	DIVW3	BISW2	BISW3	BICW2
BICW3	XORW2	XORW3	MNEGW	MOVW
CMPW	MCOMW	BITW	CLRW	TSTW
INCW	DECW	BISPSW	BICPSW	ADDL2
ADDL3	SUBL2	SUBL3	MULL2	MULL3
DIVL2	DIVL3	BISL2	BISL3	BICL2
BICL3	XORL2	XORL3	MNEGL	MOVL
CMPL	MCOML	BITL	CLRL	TSTL
INCL	DECL	ADWC	SBWC	MOVPSL
MOVAL	CVTHF	CVTFG	CVTFH	MOVAO
MOVO	CLRO	CVTLB	CVTLW	CVTDH
CVTGF	ADDG2	ADDG3	SUBG2	SUBG3
MULG2	MULG3	DIVG2	DIVG3	CVTGB
CVTGW	CVTGL	CVTRGL	CVTBG	CVTWG
CVTLG	MOVG	CMPG	MNEGG	TSTG
CVTGH	ADDH2	ADDH3	SUBH2	SUBH3
MULH2	MULH3	DIVH2	DIVH3	CVTHB
CVTHW	CVTHL	CVTRHL	CVTBH	CVTWH
CVTLH	MOVH	CMPH	MNEGH	TSTH

3 GENERAL OPERATION

3.1 Pipeline Activity

Pipeline stages take input from a previous stage, perform some transformation and produce output which is input for the next stage. If there is no input to act upon a stage is said to be idle; that is the previous stage produced no output and therefore there is nothing for the stage to do. If the next stage does not consume the data provided to it in a previous cycle then the preceding stage in the pipeline stalls.

In summary, a pipeline stage is idle if no data is provided for it to process and a pipeline stage stalls if the subsequent stage does not process previously provided data.

3.2 Instruction Cache And Translation Buffer

A separate instruction cache and translation buffer are used to access the instruction stream. All cache modeling done at DECwest suggests that the instruction stream cache hit rate will be upwards of 99%. In addition, two levels of buffering are employed. Therefore the instruction stream is not modeled explicitly. It is assumed that any instruction cache or translation buffer misses will have a minimal affect on performance.

There is one aspect of the Prefetch stage that is modeled that has to do with branches. The model keeps track of the virtual PC even though it does not use it to access the instruction translation buffer and cache. The virtual PC is used to determine if a translation buffer access is required when a branch takes place. In effect the virtual page number of the new PC is compared with the virtual page number of the old PC. If the comparison fails then an additional cycle is required to do the translation buffer access. Otherwise the page offset is concatenated with the previous physical page number to access the instruction cache. The reason for this is that it is not possible to access the translation buffer, access the instruction cache and write the prefetch buffer in one cycle.

3.3 Data Cache And Translation Buffer

A separate translation buffer and cache are used to fetch data from memory and write results. The organization of the cache is the write back scheme being used in Firefly at SRC. (The scheme employs two extra bits per cache line to keep track of shared and modified data. The shared bit indicates whether it is possible that the data might also be in another cache. The modified bit indicates whether the data has been modified but not written to memory.) There is no problem with cache coherence even in a multiprocessor configuration. DECwest modeling of the effects of write back caches suggests that memory write traffic can be cut by 60-70%. Thus it is possible to build a system without heavy demands for memory bus bandwidth.

VMS (and other operating systems) will not require any special code to manage the write back cache. The cache will be entirely transparent except when the power fails. An internal processor register will be provided so that VMS can sweep the cache and force all unwritten data to memory.

Modeling of the data translation buffer and cache is on the basis of how often a miss occurs and how many cycles it takes to process the miss. This data cache miss rate and the forced write rate are provided as parameters to the model.

3.4 Register File Write In Progress Counters

Four logical copies of the general purpose registers are maintained. Two of the copies reside in the Decode stage and two in the execution stage. These two copies can be thought of as dual port read single port write RAMs.

Register values that are needed for address calculation (base or index register values) are read from the RAMs located in the Decode stage and the value(s) is (are) passed to the Address stage. Register mode operands however are not actually read until the execution stage. This allows a major optimization with regard to allowing outstanding writes against register mode operands to be ignored since when the instruction reaches the execution stage the respective register will by definition have the most current value.

Associated with each register number (excluding PC) is a Write in Progress Counter (4 bits) that is maintained by the Decode stage and which records the number of writes outstanding against the register. A counter is used so that it is possible to have multiple writes against a register outstanding (as opposed to a single bit which would allow only one outstanding write). It is not intuitive that multiple outstanding writes are a common occurrence, but consider the following instruction sequence: ?

```
        ADDL3   R0, R1, R2
        ADDL2   R3, R2
```

When the ADDL2 instruction is decoded register R2 will already have an outstanding write against it. But since R2 will actually have the correct value when the ADDL2 reaches the execution stage it is expedient to allow multiple writes against R2. This turns out to be a very common code sequence generated by most of our compilers.

The Decode stage monitors the write bus for register values and updates its copies of the registers when appropriate. It also decrements the respective Write in Progress Counter.

Register File Write in Progress Counters are not updated for register mode operand specifiers if instruction decode has been stopped. This is necessary to avoid deadlock for instructions with multiple write destinations (e.g. EDIV R0, R1, R2, (R2)). Write in Progress Counters, however, are always updated for autoincrement and autodecrement operand specifiers even when instruction decode is stopped. The Execution stage resumes pipeline activity when a consistent state has been reached.

3.5 Data Cache Write In Progress Bits

Each cache line in the data cache has associated with it a Write in Progress Bit. This bit is written by the Operand stage when a write destination operand is processed. The bit is cleared when a write on the write bus to the affected location is executed.

Write in Progress Bits allow subsequent operand reads to occur after a write has been processed if the read is from a different location than the write. If it is not, the Operand stage will stall until the Write in Progress Bit has been cleared.

Data Cache Write in Progress Bits are not updated if instruction decode has been stopped. This is necessary to avoid deadlock for instructions with multiple write destinations (e.g. EDIV R0, R1, (R2), @(R2)). The Execution stage resumes pipeline activity when a consistent state has been reached.

3.6 Autoincrement And Autodecrement Operand Specifiers

The processing of autoincrement and autodecrement operand specifiers requires the cooperation of the Decode, Address, Operand and Execution stages of the pipeline.

The Decode stage increments the respective Write in Progress Counter and passes the register number, register contents, autodecrement value (if required), and function to be performed to the Address stage. The Address stage computes the effective address and passes the register number, effective address, and function to be performed to the Operand stage. The Operand stage first passes the register number and autoincrement or autodecrement operation to the Execute state and then fetches the operand value in the next cycle if required (i.e. not address or yield access type). The Execution stage performs the autoincrement or autodecrement function and writes the new register value into its own copy of the Register File and on the write bus. The Decode stage then picks the value off the write bus, decrements the corresponding Write in Progress Counter, and writes the new value into its copy of the Register File.

All told, the processing of an autoincrement or autodecrement operand specifier takes one additional cycle. Subsequent decoding of operand specifiers is not blocked unless the register being autoincremented or autodecremented is the one required as a base or index register for a subsequent operand.

The actual autoincrement or autodecrement in the Execution stage is performed in the "context" of the instruction to which it belongs; that is, it is performed after the previous instruction has completed and before the next instruction has started execution. The previous contents of the respective register are saved in a register log when the Execution stage performs the increment or decrement operation. This register log is cleared at the end of an instruction.

3.7 Indirect Addressing

Indirect addressing requires two trips through the Address and Operand stages. The Execution stage gets involved only to pass the indirect address through the ALU and into the write latch. The Address stage then picks the address off the write bus, adds the index register if specified, and passes the effective address back to the Operand stage. The Execution unit always processes indirect addresses in the "context" of the instruction to which it belongs.

Indirect addressing occurs very infrequently (e.g. approximately 1-2% of the time) and is not expected to adversely affect performance.

4 PIPELINE OPERATION AND STAGES

The Frigate pipeline is executed by a control program that calls the stage subroutines in reverse order. This is necessary to propagate stalls correctly since each stage is not actually executed in parallel as it would be in real hardware.

This sequence, although it works nicely for stalls, causes problems when the output of a latter pipeline stage is to be acted upon by an earlier pipeline stage in the next cycle. The problem is that the earlier pipeline stage executes after the latter pipeline stage in the current cycle.

An example is register file writes on the write bus which must be recognized by the Decode and Address stages in the next cycle, not the current. The data written by the Execution stage in the current cycle is processed by the Decode and Address stages in the next cycle and the data written by the Execution stage in the previous cycle is processed by the Decode and Address stages in the current cycle. The model accomplishes this by inserting pipeline variables that delay the recognition of data until a subsequent cycle. In the case of register file writes this is done with a 2 deep array of register numbers and valid flags. All pipeline stages examine the first member of the array and the Execution stage writes the second member. At the end of each simulated cycle the second array member is copied to the first and the second is set invalid. Other pipeline variables are simply implemented as boolean variables. For example, when a new virtual instruction PC is available to the Prefetch stage, both the value and a flag called "prefetch_new_address" are set. The Prefetch stage sees the flag, copies the new virtual address and then clears the flag. In the next cycle it will actually start delivering the instruction information at the destination address.

4.1 Prefetch

The Prefetch stage reads the input file produced by the trace program and provides the Decode stage with opcode and operand specifier information on each cycle. This information is the actual opcode and operand specifier data, including register numbers, that was collected when the subject program was traced. As Prefetch reads the trace file it updates the virtual instruction PC by computing the length of each operand specifier. Two byte opcodes require one extra cycle to deliver the "escape" opcode to the Decode stage.

If the Decode stage has not processed the data delivered to it during a previous cycle then Prefetch stalls.

If a new virtual PC has been delivered by the Decode, Address or Execution stage then the old virtual PC's virtual page number is compared with the new virtual PC's page number and the new virtual PC is copied to the old virtual PC. If the virtual page numbers match (i.e. a translation buffer access is not required) the instruction data at the target address will be delivered to the Decode stage in the next cycle. If the page numbers do not match then a translation buffer cycle is required in the next cycle and the Prefetch stage will deliver the instruction data in the cycle after that.

It should be noted that the affects of Instruction Cache and Translation buffer misses are not modeled. It is assumed that this causes minimal degradation in performance.

If instruction decode is stopped by the Decode stage then Prefetch accumulates idle time when a new opcode is to be decoded. Subsequent operand specifiers for the current instruction are delivered until an instruction boundary is reached.

If the Decode stage has not processed data delivered in a previous cycle then Prefetch accumulates stall time.

If a new virtual PC is provided by one of the Decode, Address, or Execution stages the Prefetch accumulates wait time until a translation buffer and/or cache access can be done.

In all other cases Prefetch can do useful work and accumulates work time.

4.2 Decode

The Decode stage processes the opcode and operand specifier information provided by the Prefetch stage. It maintains two copies of the Register File and the associated Write in Progress Counters, determines when information is to be passed to the Address stage, predicts if conditional branches will be taken, and transmits the destination address for loop and unconditional branches and jumps to the Prefetch stage.

The Decode stage is modeled as a finite state machine with 4 states. The states are:

1. Process opcode and first operand specifier if any,
2. Process next operand specifier, branch destination, or jump address,
3. Execute implied push/pop to/from the current stack, and
4. Finish unconditional branch or jump instruction.

Decode starts at state 1 and cycles in state 2 if necessary until all operand specifiers have been processed. If required (PUSHL, RSB, ...) state 3 or 4 is entered to finish the instruction and then back to state 1 for the next opcode.

Decode always interrogates the write bus at the start of a cycle to determine if a register value is being written. If the write bus is valid (i.e. there is a register being written) then the respective register's Write in Progress Counter is decremented but never below zero. A decrement below zero could happen when a multiple write destination instruction has stopped the pipeline and a register mode destination was present. In this case the Write in Progress Counter was not incremented to avoid possible deadlock and when the instruction is finished the counter must not be decremented.

If the Address stage has not processed data from a previous cycle then the Decode stage stalls. Otherwise the operation associated with the current state is performed.

1. State 1 - Process opcode and first specifier if any.

   If no data has been supplied by the Prefetch stage (i.e. it is waiting for a translation buffer or cache access after a new virtual PC has been received and cannot deliver any data) or instruction decode has been stopped, then the Decode stage idles.

   If the opcode is a two byte opcode then the first byte is accepted in the current cycle and the two byte opcode flag is set. The second opcode byte will be delivered in the next cycle.

If the opcode is from the stop decode class or is an unconditional branch or jump the stop decode flag is set.

If the opcode has zero specifiers and is from the implied pop class, then the state number is set to 3 and the implied pop is executed immediately. Otherwise the opcode and a no operation function are passed to the Address stage. No operation implies that the Address and Operand stages perform no operation for the respective opcode.

If the opcode has one or more specifiers then the state number is set to 2, the specifier number is set to 1, and the first operand specifier is processed immediately if possible.

2. State 2 - Process next operand specifier, branch destination, or jump address.

   If no data has been supplied by the Prefetch stage then the Decode stage idles. Otherwise an action is executed depending on the specifier mode and register number. Specifier actions include:

   1. Modes 0, 1, 2, and 3 (short literal) - The short literal value, the specifier datatype, the specifier access type and a function of literal are passed to the Address stage. If the instruction is from the implied push class then the state number is set to 3. The autodecrement SP specifier will be generated in the next cycle. All instructions from the implied push class have exactly one operand specifier. If the instruction is not from the implied push class and this is the last operand specifier then the state number is set to 1. The next cycle will process the next opcode.

   2. Mode 4 (index) - If the index register or the base register (note there must always be a base register since short literals cannot be indexed) has a Write in Progress Count greater than 1 or both registers have a Write in Progress Count of 1, then the Decode stage waits for one of the registers to be written on the write bus. At that time the Write in Progress Counter will be adjusted. Otherwise the index register invalid flag is set equal to the value of the respective Write in Progress Counter. If the Write in Progress Counter is not zero (i.e. there are outstanding writes against the register but only one) then the Address stage will pick the value off the write bus when it is written. Index mode present is set and the base address is processed by executing its action routine.

   3. Mode 5 (register mode) - If the access type is write or modify and decoding of instructions is not stopped, then the Write in Progress Counters for the respective register(s) are incremented (note that up to 4 counters could be incremented). The register number, the

specifier datatype, the specifier access type and a function of register are passed to the Address stage. If the instruction is from the implied push class then the state number is set to 3. The autodecrement SP specifier will be generated in the next cycle. If the instruction is not from the implied push class and this is the last operand specifier then the state number is set to 1. The next cycle will process the next opcode.

4. Modes 6, 10, 12, and 14 (register deferred and byte, word and longword displaced) - If the Write in Progress Counter for the base register is greater than 1 then the Decode stage waits for the register to be written on the write bus. At that time the Write in Progress Counter will be adjusted. Otherwise if the opcode is from the unconditional branch class, an index register is not specified and the base register is PC, then the branch destination can be calculated immediately and sent to the Prefetch stage. If the unconditional branch is from the implied push class then the state number is set to 3 and the implied push is executed immediately. Otherwise the state number is set to 4 and the unconditional branch is finished. If the opcode is from the unconditional branch class and either an index register is specified or the base register is not PC, then the base register invalid flag is set to the value of the respective Write in Progress Counter and the branch destination flag is set. The base register invalid flag, the base register number, the base register value, the index register invalid flag, the index register number, the displacement value, the specifier datatype, the specifier access type and a function of fetch are passed to the Address stage. If the instruction is from the implied push class, then the state number is set to 3. Otherwise it is set to 4. The autodecrement SP specifier will be generated or the unconditional branch finished in the next cycle. If the instruction is not from the unconditional branch class, then the base register invalid flag is set to the value of the respective Write in Progress Counter. The base register invalid flag, the base register number, the base register value, the index register invalid flag, the index register number, the displacement value, and a function of fetch are passed to the Address stage. If the instruction is from the implied push class then the state is set to 3. The autodecrement SP specifier will be generated in the next cycle. If the instruction is not from the implied push class and this is the last operand specifier, then the state number is set to 1. The next cycle will process the next opcode.

5. Modes 7 and 8 (autoincrement and autodecrement) - If the base register is PC (only possible for mode 8 since autodecrement PC is illegal), then the action for mode 6 is executed. This is immediate mode addressing and the Operand stage actually fetches the immediate value. If

the base register is not PC and the respective Write in Progress Counter is greater than 1, then the Decode stage waits for the register to be written on the write bus. If the base register Write in Progress Counter is 0 or 1, then the base register invalid flag is set to the value of the Write in Progress Counter. The base register invalid flag, the base register number, the base register value, the index register invalid flag, the index register number, the specifier datatype, the specifier access type and a function of modify are passed to the Address stage. The Write in Progress Counter for the base register is incremented. If the opcode is from the unconditional branch and implied push classes, then the state number is set to 3 and the branch destination flag is set. The autodecrement SP specifier will be generated in the next cycle. If the opcode is from the unconditional class and not the implied push class, then the state number is set to 4 and the branch destination flag is set. The unconditional branch will be finished in the next cycle. If the instruction is from the implied push class and not the unconditional branch class, then the state number is set to 3. The autodecrement SP specifier will be generated in the next cycle. If the instruction is not from the unconditional branch or implied push class and this is the last specifier, then the state number is set to 1. The next cycle will process the next opcode.

6. Mode 9 (autoincrement deferred) - If the base register is PC, then the action for mode 6 is executed. This is absolute addressing and the address is treated like a longword displacement with no base register. If the base register is not PC and the respective Write in Progress Counter is greater than 1, the Decode stage waits for the register to be written on the write bus. If the base register Write in Progress Counter is 0 or 1, then the base register invalid flag is set to the value of the Write in Progress Counter. The base register invalid flag, the base register number, the base register value, the index register invalid flag, the index register number, the specifier datatype, the specifier access type and a function of indirect modify are passed to the Address stage. The Write in Progess Counter for the base register is incremented. If the opcode is from the unconditional branch and implied push classes, then the state number is set to 3 and the branch destination flag is set. The autodecrement SP specifier will be generated in the next cycle. If the opcode is from the unconditional branch class and not the implied push class, then the state number is set to 4 and the branch destination flag is set. The unconditional branch will be completed in the next cycle. If the instruction is from the implied push class but not from the unconditional branch class, then the state number is set to 3. The autodecrement SP specifier will be generated

in the next cycle. If the instruction is not from the unconditional branch or implied push class and this is the last specifier, then the state number is set to 1. The next cycle will process the next instruction.

7. Modes 11, 13, and 15 (indirect byte, word, and longword displaced) - If the Write in Progress Counter for the base register is greater than 1, then the Decode stage waits for the register to be written on the write bus. If the base register Write in Progress Counter is 0 or 1, then the base register invalid flag is set to the value of the Write in Progress Counter. The base register invalid flag, the base register number, the base register value, the index register invalid flag, the index register number, the displacement value, the specifier datatype, the specifier access type and a function of indirect fetch are passed to the Address stage. If the opcode is from the unconditional branch and implied push classes, then state number is set to 3 and the branch destination flag is set. The autodecrement SP specifier will be generated in the next cycle. If the opcode is from the unconditional branch class and not from the implied push class, then the state number is set to 4 and the branch destination flag is set. The unconditional branch will be finished the next cycle. If the instruction is from the implied push class and not from the unconditional branch class, then the state number is set to 3. The autodecrement SP specifier will be generated in the next cycle. If the instruction is not from the unconditional branch or implied push class and this is the last specifier, then the state number is set to 1. The next cycle will process the next instruction.

8. Branch Displacement (byte and word) - If the opcode is from the unconditional branch class, then the destination address is sent to the Prefetch stage and the stop decode flag is cleared. If the opcode is from the unconditional branch and implied push class then the state number is set to 3. The autodecrement SP specifier will be generated in the next cycle. If the opcode is from the uncondition branch class and not the implied push class, then the state number is set to 4. The unconditional branch will be completed in the next cycle. If the instruction is not from the unconditional branch class, then it is either from the conditional branch or loop class. The base register and index register register invalid flags are set false. The base register invalid flag, index register invalid flag, the specifier datatype, the specifier access type, computed destination address and a function of displacement are passed to the Address stage. The state number is set to 1. If the opcode is from the loop class, then it is always predicted as taken. The computed destination address is sent to the Prefetch stage. If the opcode is from the conditional branch class, then the branch prediction RAM

is accessed using bits 2 through 15 of the ending address of the conditional branch instruction itself (this is actually the address of the next instruction). If the high order bit of the prediction value is set, then the branch is predicted as taken. Otherwise it is predicted as not taken. If the branch is predicted taken, then the computed destination address is sent to the Prefetch stage. The prediction flag is passed to the Address stage. The next instruction is processed in the next cycle.

3. State 3 - Execute implied push/pop to/from the current stack.

   If the Write in Progress Counter for SP is greater than 1, then the Decode stage waits for the register to be written on the write bus. If the Write in Progress Counter is 0 or 1, then the base register invalid flag is set to the value of the Write in Progress Counter. If the opcode is from the implied pop class, then the specifier access is set to read. Otherwise the opcode is from the implied push class and the specifier access is set to write. The base register invalid flag, the register number 14, the specifier datatype (always longword), the specifier access type and a function of modify are passed to the Address stage. The state number is set to 1 and the Write in Progress Counter for SP is incremented. The next opcode will be processed in the next cycle.

4. State 4 - Finish unconditional branch or jump instruction.

   Unconditional branches and jumps, although executed in the Decode stage, cannot be evaporated. They must continue through to the Execution stage so that trace traps can occur if enabled (this may be eliminated later by sending the opcode through the pipe if and only if t-bit is set or the address mode is autoincrement, autodecrement or autodecrement deferred). The branch destination flag is set false. A function code of displacement is passed to the Address stage.

The Decode stage closely simulates what the actual hardware will do. It is believed to be very accurate.

If instruction decoding is stopped by the Decode stage itself or no input has been provided by the Prefetch stage, then the Decode stage accumulates idle time.

If the Address stage has not processed data delivered to it in a previous cycle, then the Decode stage accumulates stall time.

If the Write in Progress Count of a base register or index register is greater than 1, or both a base register and an index register are specified and their respective Write in Progress Counters are equal to 1, then the Decode stage accumulates wait time.

In all other cases Decode can do useful work and accumulates work time.

4.3 Address

The Address stage computes the effective address of an operand or passes through the data it receives to the Operand stage. It is capable of performing a 3 input add in one cycle (i.e. displacement, base register, and context shifted index register) and operates from a function and data supplied by the Decode stage. This stage processes indirect addressing.

The Address stage always interrogates the write bus at the start of a cycle to determine if a register value is being written that matches an invalid register that it requires to perform the address computation. Either the base or index register may be required but not both. The base register and index register invalid flags are used for this purpose. If the write bus specifies a partial write (i.e. byte or word), then the value is merged with the value passed to the Address stage by the Decode stage. Note that there can only be one outstanding write at this time. This is guaranteed by Decode and therefore the first write on the write bus that matches the invalid register number is the one required to complete the address calculation. As soon as the corresponding register value has been received the respective invalid flag is cleared.

If the Operand stage has not processed data from a previous cycle then the Address stage stalls.

If no data has been supplied by the Decode stage, then the Address stage idles.

If the Address stage has not yet received an indirect address from the Execution stage, then the Address stage waits.

If the base register or index register invalid flags are set, then the Address stage waits. Otherwise an action is performed according to the function specified by the Decode stage. Address stage actions include:

1. Displacement - The branch destination address is computed by adding the sign extended branch displacement with the displacement PC (the PC is provided through special logic that adjusts for the length of the branch displacement). The destination address and a function of displacement are passed to the Operand stage.

2. Fetch and Modify - The effective address is computed. If the branch destination flag is set, then the effective address is sent to the Prefetch stage and the stop decode flag is cleared. If the original function was fetch, then no further processing is necessary. Otherwise the effective address, the base register number, the specifier datatype, the specifier access type and a function of fetch or modify are

passed to the Operand stage.

3. Register - The base register number, the specifier datatype, the specifier access type and a function of register are passed to the Operand stage.

4. Literal - The literal value, the specifier datatype, the specifier access type and a function code of literal are passed to the Operand stage.

5. No Operation - A no operation function is passed to the Operand stage.

6. Indirect Fetch and Indirect Modify - The effective base address is calculated. This calculation does not include the index register if it is present. The context shifted contents of the index register will be added to the indirect address when it is received from the Execution stage. The indirect flag is set, the operand access type is set to read and the operand datatype is set to longword. The indirect flag will cause the Address stage to wait in subsequent cycles until cleared by the Execution stage. The effective address, the base register number, the specifier datatype, the specifier access type and a function of indirect fetch or indirect modify are passed to the Operand stage. When the indirect address is received from the Execution stage on the write bus the Address stage will add the context shifted index register, if any, to the indirect address and then pass the original specifier datatype, the original specifier access type and a function of fetch to the Operand stage. Note that while the Address stage is waiting for an indirect address the Decode stage is stalled because it cannot deliver new data to the Address stage.

4.4 Operand

The Operand stage reads operand values from memory, checks the validity of write destinations, increments register numbers, manages the Write in Progress Bits in the data cache and delivers subsequent zero longwords for short literals. It is also responsible for assigning pointer register numbers to address the operand buffers. The Operand stage operates from data and a function code passed to it by the Address stage.

It is worth explaining the function of the pointer registers and operand buffers although they are not actually modeled in the simulation. Three pointer register FIFO's are used to store pointer registers and operand status. One is for odd numbered source specifiers, one for even numbered source specifiers, and one for destinations. Each cycle, one of the source FIFO's and the destination FIFO can be written with a register number. The number that is written is either a general register number, an operand buffer number, or an address buffer number. Operand buffers are used to hold operand values. Address buffers are used to hold the physical address of the destination (this is always available since a translation buffer access is performed on the destination operand to determine if it is accessible). If the destination address crosses a page boundary, then two address buffer entries are used (i.e. two translation buffer accesses are required). A status code is also included with each register. The status code indicates if there was an access violation, translation not valid, translation buffer miss, a modify refuse (write access to a page that does not have the modify bit set) or an attempt was made to read an I/O address. (This latter type is a VERY sticky problem in a pipelined machine. Since we have such a wonderful I/O architecture that allows people to build devices where register reads have side effects we have to guarantee that I/O addresses are read exactly once. The way this is done is to dump the address of the operand into the allocated operand buffer and tag the pointer register with a status code that will cause the Execution stage to dispatch to a routine that will explicitly read the I/O address.) The Execution stage microword provides control over the reading of the pointer register FIFO's. For example, the microword for an ADDL3 would read the register numbers from the source 1, source 2, and destination FIFO's. It should be noted that ADDL2 would do the same thing since the modify source/destination operand would cause both the even numbered source FIFO and the destination FIFO to be written. The net effect is to allow the Operand stage to fetch operands somewhat ahead of the Execution stage (e.g. some elasticity is provided by the address and operand buffers) and provides parametric microcode in the Execution stage.

If no data has been supplied by the Address stage, then the Operand stage idles.

If there are no address or operand buffers available and one is required the Operand stage stalls until the Execution stage has emptied one.

If operand fetching has been stopped, then the Operand stage idles.

The Operand stage executes an action determined by the function code supplied to it by the Address stage. The following actions are performed:

1. No Operation - No operation is performed. In the simulation model this takes one cycle. In the real machine this will not require any cycles because the opcode dispatch information is kept separately from the operand specifier information. In the model it is not. This function is used for zero specifier opcodes.

2. Literal - The first longword of a short literal value is formatted (e.g. zero extended, shifted, bits inserted, etc.) in the Decode stage and passed through the Address stage. The Operand stage assigns an operand buffer and stores the first longword of the short literal in the buffer. If the context is quadword or octaword, then the Operand stage will

allocate additional operand buffers and deliver zero longwords in subsequent cycles. One cycle is required for each longword.

3. Fetch - If the access type is address or yield, then an operand buffer is allocated and the value received from the Address stage is written into the buffer. If the access type is read, write, or modify, then a translation buffer access must be performed to determine the physical address and accessibility of the operand (the translation buffer is not modeled). For each longword in the operand (and each longword takes at least one cycle) a random number is generated and compared with the data cache miss rate that was selected when the simulation was begun. If the random number is less than or equal to the data cache miss rate then 5 additional cycles (the cache fill time) will be spent fetching the operand value. If the random number is greater than the data cache miss rate then only 1 cycle is required to fetch the operand value. If the data cache misses there is also a probability that the location that will be displaced from the cache has been modified but not written. Another random number is generated and compared with the forced write rate which was also selected when the simulation was begun. If found to "miss" then 5 additional cycles are required to first write the current contents of the cache line and then read the new value. This amounts to 11 cycles in all if the data cache misses and a forced write is required. As longwords are fetched an operand buffer is allocated, the register number written into one or more of the FIFO's and the operand value placed in the operand buffer.

4. Modify - The register number of the general register that is to be modified is written into the appropriate FIFO with a status that encodes the context and whether the operation to be performed is an autoincrement or autodecrement. The Execution stage will dispatch to a routine that actually performs the operation when an attempt is made to read the respective FIFO. In the next cycle a fetch function is performed.

5. Indirect fetch - The indirect longword address is read and written into an operand buffer. The register number of the operand buffer and a status code that indicates that the operand buffer contains an indirect address is written into the operand buffer. The Execution stage will dispatch to a routine that writes the indirect address on the write bus when an attempt is made to read the respective FIFO.

6. Indirect modify - The register number of the general register that is to be updated is written into the appropriate FIFO with a status code that encodes a context of longword and autoincrement. In the next cycle an indirect fetch is performed.

7. Register - The general register number is written into the appropriate pointer FIFO. If the operand specifier is context quadword or octaword, then the register number is incremented in successive cycles and written into the same FIFO.

8. Displacement - An operand buffer is allocated and the displacement value is stored.

4.5 Execute

The Execution stage executes instructions, performs autoincrement and autodecrement operations, and writes indirect addresses on the write bus. The Execution stage closely models the real hardware but of course computes no answers. It does, however, keep track of which registers are being written by an instruction and at the end of instruction execution it writes the registers one per cycle on the write bus.

The Execution stage is modeled as a finite state machine with 4 states. The states are:

1. Dispatch

2. Execute

3. Clean Up

4. Register Write

Execute starts at state 1 and cycles there until an opcode and all its specifiers arrive. During the time that it waits it can perform autoincrements, autodecrements and send indirect addresses to the Address stage which each take 1 cycle. This is not exactly how the hardware will work but there should be no difference in the performance. The hardware actually starts the instruction early and then ends up waiting if an operand is not ready.

When a complete instruction has arrived state 2 is entered where the number of cycles estimated for the instruction are spent.

State 3 releases operand buffers, continues pipeline activity, and sends branch addresses to the prefetch stage. If a register value is to be written state 4 is entered. In state 4 a register value is transmitted on the write bus every cycle.

The Execution stage always performs the action associated with the current state.

1. State 1 - Dispatch. If there are no operand buffers that contain operands, then the Execute stage idles. Otherwise the operand buffers are examined one at a time to determine

if a complete instruction is present or there are autoincrements, autodecrements, or indirect addresses to process. If an autoincrement or autodecrement is found, then the operand buffer is removed and the register number is written on the write bus. This consumes the entire cycle. If an indirect address is found, then the operand buffer is removed and the indirect address is written on the write bus to signal the Address stage that the address it is waiting for is present (and thank God indirect addresses are infrequent - the whole pipeline is backed up while we sequence 1 indirect address through the pipeline). This also consumes the entire cycle. If no autoincrments, autodecrements, or indirect addresses are found before a complete instruction has been assembled then the state number is set to 2 and the cycle counter is set to the number of cycles the instruction is estimated to take. Estimates used in the simulator are as close to reality as possible since we must make judgements about the final performance of the actual hardware. State 2 is executed.

2. State 2 - Execute. The number of cycles remaining is decremented. If the result is nonzero, then the instruction execution is not complete. The number of cycles will be decremented again in the next cycle. If the remaining cycles is zero, then instruction execution is complete. If the instruction has no specifiers and is not from the implied pop class but is from the decode stop or fetch stop classes, then either the Decode or Operand stage is continued. If the opcode also caused a branch (e.g. REI) then a new PC is sent to the Prefetch stage. If the instruction has one or more specifiers or is from the implied pop class, then the state number is set to 3 and state 3 is executed.

3. State 3 - Clean Up. The operand buffers are released. If the instruction writes a destination register, then the state number is set to 4 and state 4 is executed. Otherwise if the instruction is from the decode or fetch stop classes, then either the Decode or Operand stage is continued. If the opcode also caused a branch (e.g. CHMK) then a new PC is sent to the Prefetch stage. If the instruction is from the conditional branch or loop classes and the branch was not predicted correctly, then the entire pipeline is flushed, the correct PC is sent to the Prefetch stage and the branch prediction RAM is updated. The state number is set to 1.

4. State 4 - Register Write. The destination register number is written on the write bus and the number of registers remaining to write is decremented. If the result is zero, then the state number is set to 3 and state 3 is executed. If the result is nonzero, then register number is incremented. The next register will be written in the next cycle.

[end of fb.rno]

MAIL: BVAX

```
From:   RICKS::CALLANDER "Mike Callander 225-5591 HLO1-1/S11  17-Nov-1988 0843" 17-NOV-1988 08:43:04.09
To:     DURDAN,UHLER,SULLIVAN,MARCELLO
CC:
Subj:   "BVAX" simulations on XMI-2
```

Hugh,

I've completed the "BVAX" simulations you asked for. I ran the Mariah performance model at an 8ns cycle time with a 4KB PCACHE. I made 2 runs with different backup cache sizes. All other parameters(like cache flush frequency) were left the same as we use on Mariah.

/Mike

```
Run Mariah.1.1 - Mariah at 8ns with 128KB cache

                                       |<------ Computed Mariah ------>|
          |  Instn   | 780   |         |                | Sngl   Total|  | % bus  |
  Trace   |  Count   | TPI   |   |CPU| |  Cycles  |  TPI | x780 | x780 |  | Used   |
----------+----------+-------+   +---+ +----------+------+------+------+  +--------+
     MAIL |  284986  | 11.4  |   | 1 | |  5145886 | 18.1 | 15.8 | 15.8 |  |  17.6  |
----------+----------+-------+   +---+ +----------+------+------+------+  +--------+
   NLINKU |  375297  | 11.4  |   | 1 | |  4433011 | 11.8 | 24.1 | 24.1 |  |  10.9  |
----------+----------+-------+   +---+ +----------+------+------+------+  +--------+
    NFORT |  424294  | 10.8  |   | 1 | |  5219893 | 12.3 | 21.9 | 21.9 |  |  12.1  |
----------+----------+-------+   +---+ +----------+------+------+------+  +--------+
   RUNOFF |  403552  |  9.8  |   | 1 | |  4878005 | 12.1 | 20.3 | 20.3 |  |  12.1  |
----------+----------+-------+   +---+ +----------+------+------+------+  +--------+
     SORT |  402831  |  9.1  |   | 1 | |  3955004 |  9.8 | 23.2 | 23.2 |  |  11.7  |
----------+----------+-------+   +---+ +----------+------+------+------+  +--------+
 Summary  | 1890960  |       |         | 23631799 | 12.5 | 20.9 | 20.9 |  |        |
 Geo Mean|           | 10.5  |         |          | 12.6 | 20.8 | 20.8 |  |  12.7  |
                                         EF =100.0%

Run Mariah.1.2 - Mariah at 8ns with 1MB cache

                                       |<------ Computed Mariah ------>|
          |  Instn   | 780   |         |                | Sngl   Total|  | % bus  |
  Trace   |  Count   | TPI   |   |CPU| |  Cycles  |  TPI | x780 | x780 |  | Used   |
----------+----------+-------+   +---+ +----------+------+------+------+  +--------+
     MAIL |  284986  | 11.4  |   | 1 | |  4481662 | 15.7 | 18.1 | 18.1 |  |  14.4  |
----------+----------+-------+   +---+ +----------+------+------+------+  +--------+
   NLINKU |  375297  | 11.4  |   | 1 | |  4202547 | 11.2 | 25.5 | 25.5 |  |   9.0  |
----------+----------+-------+   +---+ +----------+------+------+------+  +--------+
    NFORT |  424294  | 10.8  |   | 1 | |  4740861 | 11.2 | 24.2 | 24.2 |  |   9.0  |
----------+----------+-------+   +---+ +----------+------+------+------+  +--------+
   RUNOFF |  403552  |  9.8  |   | 1 | |  4671781 | 11.6 | 21.2 | 21.2 |  |  10.9  |
----------+----------+-------+   +---+ +----------+------+------+------+  +--------+
     SORT |  402831  |  9.1  |   | 1 | |  3884460 |  9.6 | 23.6 | 23.6 |  |  11.0  |
----------+----------+-------+   +---+ +----------+------+------+------+  +--------+
 Summary  | 1890960  |       |         | 21981311 | 11.6 | 22.4 | 22.4 |  |        |
 Geo Mean|           | 10.5  |         |          | 11.7 | 22.3 | 22.3 |  |  10.7  |
                                         EF =107.5%
```

```
From:   RICKS::DURDAN "21-Nov-1988 1058" 21-NOV-1988 10:58:06.62
To:     @DIS:VECTOR_STAFF,UHLER
CC:
Subj:   BVAX - FYI

From:   RICKS::FISHER          21-NOV-1988 09:12
To:     DURDAN,GRUNDMAN
Subj:   BVAX

From:   RICKS::CASALETTO     18-NOV-1988 18:25
To:     AMNON,MARCELLO,SULLIVAN,LAPRADE
Subj:   FYI - direct and to the point.

From:   ROCK::SUPNIK "New location HLO2-3/C11  18-Nov-1988 1752" 18-NOV-1988 17:53
To:     RICKS::CASALETTO,WONDER::HARBERT
Subj:   BVAX will not happen - copy of message to MSB and SCO staff

From:   ROCK::SUPNIK          "New location HLO2-3/C11" 18-NOV-1988 17:51:57.86
To:     NM%CHIPS::WALKER,NM%SHARE::PALMER,NM%MSBIS::DEMMER,NM%HYDRA::DURVASALA
CC:     NM%AD::BIDERMANN,SUPNIK
Subj:   BVAX will not happen
```

I don't like being the bearer of bad news, but the thrashing about the short term VAX strategy has got to stop.

So, simply put: BVAX will not happen. Here's why.

1. There's no team. The project plan is built on the assumption that HLO would supply the architecture expertese and the chip designers. *NO ONE* from HLO is signed up: no architects, no chip designers.

   Further, no senior people are going to sign up. Why should anyone work on an ECL project that has bounced in and out of the corporate plans, that still is not officially approved, that is characterized as an insurance program for Aquarius and Raven, and that will be under constant political attack from another engineering group, when there is important, high payback, lower stress CMOS work to do?

2. The opportunity window has passed. To meet the proposed schedule, logic design had to start October 1. Right now, there's not even an architectural design. BVAX was a great opportunity in June, when it was proposed. It isn't any more.

3. There's no platform. Performance studies show that in the CMAX (XMI-1) box, BVAX delivers 13 vups - little more than Mariah. XMI-2 systems won't accomodate an ECL CPU. No ECL platform is planned for the required timeframe.

4. A direct CMOS to ECL translation won't work. Raven attempted, with the best intent in the world, a direct translation of the Rigel design from CMOS to ECL. It did not work out, for many reasons (see Appendix). BVAX cannot just translate CVAX; an architectural rework is needed. This will add more time to the schedule.

BVAX is vaporware. It's time to recognize that, and move on. Mariah PG's its first chip (floating point) NEXT MONTH. Let's put the limited budget, resources, and energy that are available into making Mariah, Raven, and NVAX successful.

Appendix: Why Direct Translation Fails

Here are some of the reasons why direct translation fails.

1. Certain CMOS structures don't work in ECL. For example, fully associative TB's can't be built. Changing the TB to direct mapped requires rethinking (and recoding) all the memory management algorithms and microcode.

2. Certain CMOS design practices don't work in ECL. For example, precharged busses with many sources translate into ultra-wide multiplexors, which are slow and costly in gates. Pass gate structures (like shifters) must be completely redesigned.

3. The ratio between the access time of regular structures and the target microcycle changes drastically. For example, in CVAX, the control store access is 50% of the microcycle. In Raven, it is 75%, and in BVAX it might be worse. This requires rethinking the amount of logic in the sequencing path, which in turn alters the entire control structure of the micromachine. The same effect occurs in the TB/cache path.

4. CMOS uses too many gates. Studies have shown that, without cache and control store, CVAX is 30k to 35k simple gates (and Rigel is 45k simple gates) - more than the Fujitsu arrays can handle. Reductions in gate count are needed, usually at the expense of a wider microword, to simplify decoding, and of more microcode, to reduce hardwired control.

MicroECL is a promising technology, but like VLSI MOS, it poses its own unique problems for chip designers and will require its own unique solutions. There won't be any free lunches.

/Bob Supnik

```
From:   RICKS::CASALETTO "27-Dec-1988 0913" 27-DEC-1988 09:08:40.38
To:     @AFLSTAFF
CC:
Subj:   FYI

From:   AD::BIDERMANN    22-DEC-1988 12:30
To:     @BVAX
Subj:   Hudson Support for BVAX


        +---+---+---+---+---+---+---+
        | d | i | g | i | t | a | l |    I N T E R O F F I C E   M E M O
        +---+---+---+---+---+---+---+

        TO: FRANK BOMBA                     DATE:   DECEMBER 20, 1988
            BILL DEMMER                     FROM:   BILL BIDERMAN
            SAS DURVASULA                   DEPT:   ADVANCED DEVELOPMENT
            BOB PALMER                      NET:    NULL::BIDERMANN
            BOB SUPNIK                      EXT:    225-5049
            LARRY WALKER                    L/MS:   HLO2-3/H3

        SUBJECT: BVAX SUPPORT


                At our meeting on December 2nd, I committed to attempt
        to find a team of 3 Hudson people to assist the BVAX
        development effort.  I have been unable to assemble this team
        due to the continuing uncertainty, perceived or otherwise, of
        the strategy surrounding the program.  Therefore, despite the
        experience which Hudson could gain and the impact on the
        program, I have come to the conclusion that we will not be
        able to assist in the development of BVAX as proposed.

                                            Sincerely,

                                            Bill Bidermann
```

```
From:   HYDRA::BOMBA "Soul of an Old Machine  30-Mar-1989 1443" 30-MAR-1989 15:01:48.95
To:     @DESREV.DIS,DURVASULA,MSBCS::NEUMAN
CC:
Subj:   BVAX Technical Review

 ---------------
| d i g i t a l |   I N T E R O F F I C E    M E M O R A N D U M
 ---------------

TO: Distribution                        DATE:   30 Mar 89
                                        FROM:   Frank Bomba
CC: Sas Durvasula                       DEPT:   BVAX Development
    Paul Neuman                         EXT:    226-6595
                                        LOC/MAIL STOP:  LTN1-1/G08

ATT: Review agenda

SUBJECT: BVAX Technical Review

The BVAX Project Team has scheduled a technical review for the afternoon
of April 27. We invite you to attend. The review will be held in the
TAY2 facility which is the lower of the two new Taylor St. buildings
accross the street from LTN2. The meeting will be in the Harvard
conference room from 1:00 to 5:00.

The purpose of this review is to get your critical feedback on our
design early in the project. Note we have not scheduled a more
lengthy, detailed review due to:

            - The nature of this implementation (copy wherever possible)

            - The expediency of our schedule and the value of our time
              as well as your own

You will receive a package of specifications before the review. If you
are unable to attend, we would still appreciate your comments on our
documentation. Feel free to send any comments via electronic mail or call:

                        Frank Bomba
                        DTN 226-6595
                        HYDRA::BOMBA

Attached is the agenda for the meeting. Since the review will move
quickly from one major topic to another, questions will be answered when
we can within the time limits. More detailed responses, especially those that
require additional work on our part will be answered as soon as possible
after the meeting by mail etc. Responses will be copied to all those
in attendance.

We sincerely hope you can take time to attend and help BVAX succeed.

Thanks,

Frank
```

AGENDA FOR BVAX TECHNICAL REVIEW -- APRIL 27, 1989

1:00 - 1:15	Welcome/introduction	- Bomba
1:15 - 2:15	System Overview XBP Module functional overview Performance modeling results	- Polzin
2:15 - 2:30	Break	
2:30 - 3:30	P-chip/R-chip/F-chip architecture and status - Chip CAD process - Key differences from CVAX/CFPA - Partitioning/gate count - Clocks - On chip and chip to chip critical paths - Custom cell plans - Microcode	- Schumann
3:30 - 3:45	C-chip functionality	- Polzin
3:45 - 4:00	X-chip functionality	- Keefer
4:00 - 4:45	Module Physical Design - Module CAD process - Layout - Module level critical timing paths - Thermal management	- Stefanski
4:45 - 5:00	Actions/issues	- Bomba

```
From:   HYDRA::BOMBA "Soul of an Old Machine  14-Apr-1989 1145" 14-APR-1989 11:51:21.36
To:     @JUSTMYGROUP.DIS,@PERFORMANCE.DIS,MSBCS::DEMMER
CC:
Subj:   BVAX PROJECT UPDATE
```

```
---------------
| d i g i t a l |   I N T E R O F F I C E   M E M O R A N D U M
---------------
```

TO: Bill Demmer

CC: Distribution

DATE: 14 April 89
FROM: Frank Bomba
DEPT: BVAX Development
EXT: 226-6595
LOC/MAIL STOP: LTN1-1/G08

ATT: Performance modeling results

SUBJECT: BVAX Project Update

I am happy to report excellent BVAX implementation progress over the past few months. As you know, this is due to the efforts of a rather small but dedicated team of individuals. Even more commendable is that this magnitude of progress has been made given the limited external support we have come to expect for this "unfunded" project.

At this point, BVAX can hardly be called vaporware:

o We have more than 50 percent of the CVAX processor gate-level design now translated to bipolar BVAX schematics.

o We have confidence that our design approach of direct gate mapping will work. We have confidence that we can make the design fit based on the Fujitsu information that we have today.

o We now have a running BVAX-specific performance model that again validates an achievable goal of 22+ VUPs per processor -- a commitment by the program made more than six months ago.

o We have done enough engineering investigation to know that the CMAX platform will accommodate a three-processor BVAX with minimal incremental changes (still impingement, three-processor cage support, and one additional BVAX regulator.)

o We will soon have draft specifications of all our key chips, module, and system available for a public design review later this month.

o We have had very positive comments on the technical aspect of our design approach from Strecker, Supnik, and Stewart and Rubinson.

o We are also investigating the good possibility of additional microcode support for certain commercial instructions to improve the COBOL performance of this machine.

We still do have a way to go. Design team staffing and the physical gate array CAD tool processes remain our key risks (outside of the obvious funding/MSB strategy). There is a willingness on the part of Kusik's team to help, but availability of resources is not clear. Only you can help us here...

In any case, I am pleased to report that at this time, BVAX remains a good opportunity for a minimized risk, time-to-market MSB product.

```
+---------------------------+ TM
|   |   |   |   |   |   |   |
| d | i | g | i | t | a | l |
|   |   |   |   |   |   |   |
+---------------------------+
```

INTEROFFICE MEMORANDUM

TO: Frank Bomba

CC: Kathy Harrington,
    Reinhard Schumann,
    Mark Stefanski

DATE: 12 April 1989
FROM: Steve Polzin
DEPT: BVAX Development
EXT: 226-6292
LOC/MAIL STOP: LTN1-1/G08

Subject: BVAX Performance Modeling Results

0.0 Executive Summary

A performance model of the BVAX CPU has been created based on the original CVAX model written by Joel Emer (in Pascal). The model represents a conservative model of the current BVAX design. It includes models of the XMI-2 bus protocol and XMA2 memory module. The model uses context switch cache flush and invalidate traffic similar to that implemented for the Rigel and Mariah performance models.

The model shows that a single BVAX CPU should yield 22.81 VUPs over the "Uhler-5" set of benchmark traces and 21.73 VUPs over the "Aqua-14" set of benchmark traces. The model also confirms earlier work by Mike Uhler and Rick Gillett that the XMI-2 can easily support three 22 VUP BVAX CPUs. The model is conservative in many ways, particularly in the modeling of floating-point instructions.

1.0 Detailed Results

BVAX Performance

Benchmark	VAX 11/780 TPI	BVAX TPI	BVAX VUPs @ 8nsec	
Mail	11.36	13.82	20.54	
Fort	10.8	11.92	22.65	
Link	11.4	11.26	25.31	
Runoff	9.8	11.51	21.29	
Sort	9.09	9.36	24.28	
	"Uhler-5" Average		22.81	
DirBrief	11.11	16.39	16.94	
Dir	11.11	14.29	19.43	
Hanoi	4.11	4.11	25.00	
LASL2D	20.0	20.25	24.69	*
LASL2S	9.91	14.37	17.24	*
EDT	12.2	12.83	23.77	
PL/1	9.8	11.42	21.45	
WHETD	16.44	19.4	21.19	*
WHETS	10.78	13.18	20.45	*
	"Aqua-14" Average		21.73	

* BVAX Model Uses 'Worse-Case' CFPA cycle counts.

2.0 BVAX Performance Model Details

The BVAX performance model is based on the original CVAX performance model written by Joel Emer. The XCP version of this model (modified by Doug Williams) was used as the base for BVAX. Extensive modifications were made to the XMI, memory, primary cache, secondary cache and statistics reporting routines. In addition, the 'µ-code' file that drives the model was changed for certain instructions to reflect the current CVAX implementation (Certain floating point instructions, MULL,DIVL).

The BVAX performance model implements the following features:

- Simplistic XMA2 Memory
    - 'Real' XMA2 cycle counts
    - Consecutive data cycles
    - Write recovery time
    - Single Memory Module
- 1 Mbyte, direct-mapped Write-back secondary cache (3 cycles on B-DAL given the current 12nsec Taa spec).
- 16 Entry Invalidate Queue
- 8 Entry Writeback Queue
- Context Switches are handled as follows:
    - Every 2msec, the primary cache and TB are completely flushed
    - Every 2msec, the secondary cache has half of its tags (chosen at random) written with an address of -1 (remains valid).
    - Also the secondary cache is initialized with all tags valid with address = -1 and 33% dirty.
- 8 other XMI commanders are modeled for invalidate traffic as follows:
    - Whenever the memory gets a request from BVAX , 8 invalidates are generated. 95% are random addresses, 5% are the same address of the last memory write. Read/write type invalidates are randomly generated.
- Primary Cache writes incur a 50% cycle 'stutter' as defined by Reinhard.
- F-Chip is modeled simply as a cycle count for a given instruction. The CFPA (and therefore the F-Chip) uses algorithms that are highly data dependent. The F-Chip cycle counts for floating point instructions are for 'worse-case' data patterns. The tight LINPACK and WHET benchmark loops should yield very close to 10x CFPA performance.

The current BVAX performance model reflects a conservative estimate of what we think the BVAX design is. Transfer times from the C-Chip to/from the X-Chip are conservative, as is the invalidate processing rate. The model includes the known restrictions on the BiCMOS rams that will comprise the secondary cache.

3.0 Future Work

The BVAX performance model will be maintained so that it will reflect the BVAX design as it evolves. Periodic updates will be made and the suite of benchmark traces will be re-run to verify the design.

The F-Chip routine could be updated to cycle according to the data pattern in the operands. It currently takes a simple cycle count from the .UCD (pseudo µcode) file.

We are currently investigating the addition of a few of the packed decimal instructions to take advantage of our 'extra' 400 or so µwords available in the R-Chip. I have obtained a set of COBOL benchmarks (GTE and the Jalics set) from Bhagyam Moses and we will attempt to quantify the performance of BVAX with and without the addition of these instructions across the various code sequences generated by the COBOL compiler (V3.3, V4.2/ins=generic, V4.2/ins=decimal, V4.2/ins=no_decimal).

To do this we will take the benchmarks and using the VAXITR program, run them on a Macho VAX (8800) and create a set of .ITR trace files. We will then modify the VAXEMUL image of VMS to not toggle the T bit in the PSL when entering and leaving instruction emulation routines and create another set of .ITR files that reflects the TRUE instruction sequence on a µVAX (6200). Finally, we will modify VAXEMUL again to only toggle the T bit on the instructions that we do NOT implement in µcode and create another set of traces from a 6200. By then applying these traces to the BVAX performance model with a new .UCD file that reflects the added µcode to support these packed decimal instructions we should be able to predict the performance of BVAX for certain commercial benchmarks.

4.0 Acknowledgment

My thanks to the following who have helped with this effort.

- Joel Emer and Doug Williams for creating the model.
- Mitch Rosich (SOC) for supplying the source
- Mike Callander (Mariah) for supplying the Mariah perf model sources and a bunch of valuable advice.
- Reinhard Schumann (BVAX) for p-cache and p/f/r chip specifics.
- Don Denning (BVAX) for help in wading through the real CVAX µcode to get the .UCD file in shape.
- Dwight Manley (Aquarius) for his Aquarius traces and also the VAXITR program.
- Kip Landingham (SPAG) for the set of COBOL traces.

```
From:   ROCK::UHLER          "Mike Uhler, DTN 225-4735, HLO2-3/C11" 24-APR-1989 08:15:39.48
To:     HYDRA::BOMBA
CC:     UHLER
Subj:   RE: BVAX PROJECT UPDATE
```

Although I haven't had time to go thru the BVAX specs in detail, here are two observations:

- The context switch rate that you indicate that you're using is 2ms. At an 8ns cycle time and 11.49 TPI (geometric mean of the Uhler 5), this amounts to a context switch every 21758 instructions if I've done the calculations correctly. For NVAX, we are using a 10000 instruction context switch frequency, which corresponds to 854 us on the same set of traces. So, although BVAX and NVAX are roughly the same speed in VUPS, you are using a context switch interval that is 2.3 times ours. Have you made any sensitivity runs to determine the performance impact of changing the frequency?

- You note that you are investigating adding certain of the packed decimal instructions back in to improve Cobol performance. Page 11-3 of Rev H of the SRM says "Instructions in an application extension group may be implemented or omitted only as a group". That means you can't add a few instructions in the group without adding them all. If you really intend to pursue this as a serious option, I suggest that you immediately submit an ECO to VAXB since this has the potential of becomming a giant rat hole.

/GMU

From: HYDRA::POLZIN "Steve Polzin in LTN1 24-Apr-1989 1139" 24-APR-1989 11:43:35.61
To: ROCK::UHLER
CC: BOMBA,SCHUMANN,STEFANSKI,POLZIN
Subj: RE: BVAX PROJECT UPDATE

Hi Mike,

Thanks for your questions to Frank, I'll try to answer them.

On context switches, I asked many people as to just what flush frequency I should use (I think I even talked to you about this). The numbers ranged from every 150,000 instructions (Simon Steely) to every 2msec (Mike Callander). The impression that I got was that context switch flush rates are functions of time and not absolute numbers of instructions. I'd be very interested to understand why you chose such a high rate of context switch flushing for NVAX. 2msec seems to be the number that was used on CVAX, Rigel and Mariah. The perf numbers that I published use a context switch rate of 20000 instructions. I've run some traces with context switch flush rate at every 10000 instructions and our "Uhler-5" average VUPs is 21.75, about a 4.6% decrease. Also, please note that we didn't simply implement a simple flush but rather I tried to emulate what you and Mike did for the Rigel and Mariah models (to stimulate writeback traffic).

On packed decimal, we are very aware of what the SRM says and we have talked with a number of people including Tom Eggers, Rich Brunner, Cheryl Wiecek, and Wayne Cardoza. The bottom line appears to be that the SRM statement was inserted in an attempt to establish some consistent expectations as to what a given processor performance would be. The architecture group and VMS have no objections to us implementing a few of the instructions (and would not oppose such an ECO to this affect when we get to that point). VMS currently tries DIVP in SYSBOOT to determine if the emulator image should be loaded. We certainly would not implement DIVP and VMS has also offered to conditionally load the emulator image based on our SID.

I agree that the whole issue of packed decimal instructions is a rat hole (ref Supniks memo on Cobol perf of 13-Apr). We are trying to 'patch' the problem given that we do have 400-500 extra μwords. If the problem of packed decimal instructions is resolved 'globally' (as Supnik suggests) then there would be no need. We agree with Bob on the issue of the VMS emulator routines. We've been going through them in detail and they could use a good clean up and/or re-coding from a hardware perspective and we might get 10-20% right there.

We are getting a very loud message from marketing that we have a problem. In July, the high-end VAX will be a μVAX. We are trying to see if by implementing only a few of the packed decimal instructions (CVTTP, CVTPT seem to be the most likely candidates), we can get a '90% solution'. I'd be very interested in your perspective on this issue.

Thanks for your comments.

s

From: ROCK::UHLER "Mike Uhler, DTN 225-4735, HLO2-3/C11" 8-MAY-1989 08:37:07.06
To: HYDRA::POLZIN
CC: UHLER
Subj: RE: BVAX PROJECT UPDATE

We've been quite busy lately, so I only got a chance to look over your mail this weekend.

Relative to context switch rates, here are the algorithms used for Rigel, Mariah, and NVAX:

Rigel: Full flush of TB and Pcache every 7500 instructions, random displacement of Bcache every 7500 instructions. At 11 TPI and 28ns cycle, this is 2.3ms between context switches.

Mariah: Full flush of TB and Pcache every 7500 instructions, full flush of Bcache every 35000 instructions. At 11 TPI and 20ns cycle, this is 1.5ms between TB and Pcache flushes and 7ms between Bcache flushes.

NVAX: Full flush of TB and VIC every 10000 instructions, run 50000 instructions worth of address traces thru the cache subsystem every 10000 instructions to displace data. At 7 TPI and 14ns cycle, this is 1ms between context switch.

You asked whether context switch rates were functions of time or number of instructions. The answer is a function of the kind of workload being executed. Early in Nautilus, Cheryl Wiecek worked with the VMS performance group to try to estimate the context switch rate for Nautilus. They came up with the following events that lead to a context switch:

- Conditional quantum end
- Rescheduling interrupt (process premption)
- One second conditional processing (VMS cleanup)
- Voluntary wait end due to I/O, event flag, process hibernation or suspension, etc.
- Involuntary wait end due to page fault, resource depletion, mutex wait, etc.

Of these, the first three are a function of time and the last two are a function of instructions. According to their analysis at that time (1984), most of the context switches were due to voluntary or involuntary waits of one form or another. This is just a restatement of what has always been observed: most processes block rather than expire quantum.

Therefore, we have always used instructions rather than cycles to trigger context switches. As you can see, this resulted in the context switch interval decreasing from 2.3ms in Rigel to about 1ms in NVAX. If you think that you can justify a different workload than we've been assuming, then 2ms as your context switch rate is reasonable. If not, then 4.6% error in performance is fairly large in my opinion, and it's not a good idea to set expectations and then fail to deliver.

I have no opinion about the viability of adding certain of the decimal instructions to improve performance because the only data that I've seen is Bob's memo. If you reach the conclusion that there are only a few instructions that really matter (CVTTP and CVTPT as you indicated), I'd be very interested in seeing the analysis. While we don't have 400 spare microwords in NVAX, we might be able to implement something, especially if there is a big leverage.

/GMU

```
From:   HYDRA::POLZIN "Steve Polzin in LTN1  08-May-1989 1922"  8-MAY-1989 19:25:11.20
To:     ROCK::UHLER
CC:     POLZIN
Subj:   RE: BVAX PROJECT UPDATE
```

Hi Mike,

Thanks for your explanation of modeling context switching. I tend to agree with you but we are in a weird situation. We are being forced into a rather awkward (for me at least) position of being an Aquarius alternative/replacement and hence I am being pressured to publish performance numbers for BVAX relative to Aqua.

In order to level the playing field I want to use the same assumptions that Dwight Manley did when he did the Aqua performance work. According to Dwight, he NEVER flushed any caches on any of the Aqua performance work. He simply flushed the TB on SVPCTX and LDPCTX. Therefore, when we compare to Aquarius, I do the same and BVAX comes in at 23 VUPs vs. 25.3 VUPs for Aqua running the "Uhler-5". Aquarius does much better on the "Aqua-14" set of benchmarks due to the hand-optimized LASL loop 2s and the Whetstones using lots of floating point (the one cycle 32x32 multiplier and 3 cycle 32x32 divider really scream).

I think I need to publish/advertise two sets of numbers: One set that never flushes for use when comparing BVAX to Aqua and then another set that flushes according to your specifications for use when comparing BVAX to Rigel/Mariah/NVAX. I'm running both sets now, it looks like the BVAX numbers to compare to Aqua come in around 23-24 VUPs and the BVAX numbers to compare to Rigel/Mariah/NVAX come in around 20-21 VUPs.

I'll let you know what we can come up with for decimal string performance. We've been spending most of our time fighting for our lives. Fortunately, Reinhard has more than 70% of the CVAX translated into Fuji already. Hopefully we will be done before the managers get around to canceling us.

Thanks

s

1.0 CHIP SIZE

Frigate TB/cache chip -- Early Feasibility Study Results

The address comparator array is composed of 1 bit tag register and comparator cells ('comparand registers'), each about 58 microns wide by 100 microns high. By interleaving the data registers for two fully associative cells, we expect to obtain a height of only 64 comparand registers. This represents a height of about 250 mils. Thirty-two bits of comparand register abutted to 32 bits of interleaved data array, abutted to 32 more bits of comparand gives a width of about 220 mils. We do not have estimates of the carry ladder logic (which will add to the width) but we expect it to be less than 30 microns. The resulting chip is about 250 x 250 mils, exclusive of pads.

```
     2 Interleaved Fully Associative Cells

 |<----    5568 micron (219 mils)      ---->|

 +------------+-------------+------------+  ---
 |            |   Data 1    |            |   ^
 | Comparand 1 +-------------+ Comparand 2 |   | 100 micron
 |            |   Data 2    |            |   v (3.94 mil)
 +------------+-------------+------------+  ---
```

2.0 SPEED ESTIMATES

Given the high performance of the N-channel devices, we have adopted a "mostly N-channel" design approach. To determine a hit in the Frigate TB/cache chip, three important propagation times must be calculated:

1. the pull-down of precharged address lines going to the comparand registers,

2. the pull-down of all precharged missing hit lines low, and

3. the pull-down of the precharged common data bus lines.

SPICE simulation shows a 3 ns typical-typical transistor model time and 4 to 5 ns slow-slow transistor model time to pull-down the address lines. (These results were obtained using the Hudson CMOSTT.MOD and CMOSSS.MOD SPICE models.) This assumes the following:

- each address line is 6400 x 5 microns in metal 2
- address line resistance of 28 ohms
- address line capacitance of 2.3 pF, and
- a transistor gate capacitance loading 1.6 pF (64 transistors, each gate 2 x 12 micron).

The worst case performance for pulling down a hit line is when a single bit of the comparand register differs from the search address, resulting in a single XOR structure having to discharge the hit line. The SPICE simulation model yields 3 ns for the typical-typical case and 4 to 5 ns for the slow-slow case. This model assumes:

- an 1800 by 4 micron metal 1 comparand hit line,
- a comparand hit line resistance of 22 ohms,
- a comparand hit line capacitance of 0.40 pF,
- a regenerating inverter propagation delay,
- an 1800 by 4 micron metal 1 data array hit line,
- a data array hit line resistance of 22 ohms,
- a data array hit line capacitance of 0.40 pf, and
- a data array hit transistor gate capacitance loading of 0.86 pF.

Finally, the SPICE simulation for driving the common data bus lines yields a typical-typical time of 4 to 5 nsec, and 5 to 6 nsec for the slow-slow case. This assumes:

- data lines are in metal 2, 6400 by 5 microns,
- data line resistance is 28 ohms,
- data line capacitance is 2.3 pF, and
- the discharge path is through two series 2 by 12 micron gate n-channel transistors.

Neglecting input and output pad times, it appears that we should be able to register a hit and provide the data for that hit:

- in 3 + 3 + 5 = 11 ns for the typical-typical transistor case, and

- in 5 + 5 + 6 = 16 ns for the slow-slow transistor case.

After a hit has been determined, the Frigate TB/cache chip requires a rotation of the matching address and data to the input of their respective arrays. Note that the address lines are already charged with the address and the data lines are already charged with the data. The two major timing phases for the shift are:

- determining the extent of the registers participating in the shift, and

- the actual shift.

We expect to use Manchester-carry-ladder-like logic to determine the registers which must participate in the shift. We currently do not have any estimates of this timing. The comparand registers are made of 3 clocked-inverters with a cross-coupled pair in over-drive configuration. SPICE modeling indicates that these registers should be able to shift in 3 ns typically and in 4 to 5 ns for the slow case. This timing is not as critical as it is done in 'shadow' time. We are closely modelling the over-drive characteristics of this overdrive register.

3.0 OVERALL SPEED

A *very* preliminary model of address-in/comparand-hit/data-out gives a time of 9 ns. This assumes the timing generator starts discharging distribution capacitance in anticipation of final values. This might be possible with a self-timed timing generator. It should also be noted that not all capacitive loads are taken into account in this overall timing model.

{ end of mist:[kehl]tlb.rno tk 12/3/84 }

FRIGATE WORKING DESIGN DOCUMENT

VERSION 1.0

DIGITAL EQUIPMENT CORPORATION - CONFIDENTIAL AND PROPRIETARY

Abstract

This document describes the Frigate product. THIS
THIS
DOCUMENT IS CONFIDENTIAL. Do not distribute it outside
DOCUMENT IS CONFIDENTIAL
the company.

Issued by: DECwest Engineering

Revision History	Date	Reason for Change	Author
Version 1.0	4 Dec 84	Initial Distribution	P. Schnorr

CHAPTER 1

FRIGATE SYSTEM OVERVIEW

1.1 FRIGATE SYSTEM OVERVIEW

Frigate is a BI-based VAX hardware system under development at DECwest Engineering for FCS in September of 1987. The system consists of a processor which executes the complete VAX instruction set (without compatibility mode), a 64-bit Frigate System Bus (FSB), a memory subsystem consisting of a memory controller and up to eight memory arrays, a console subsystem, a BI adapter which supports up to two BI's, a FSB backplane, and a power system and package.

The kernel system consists of a single processor and console subsystem, a memory controller and 4 MB of memory, a BI adapter which includes support for a single BI backplane, a FSB backplane, and a power system and package.

Frigate will support symmetric multiprocessing for up to six processors in a single backplane.

1.2 CONSTRAINTS AND GOALS

Constraints are product attributes which define the minimum product; as such, they must be delivered, or the product is not viable. Constraints are:

1. FCS must occur by September, 1987 (Q1FY88)

2. Kernel cost must not exceed $20K

3. Single-processor performance must be at least four times the 11/780.

4. Processor must execute the VAX instruction set

5. System must be compatible with BI and BI options

6. System must support at least two BI's

7. Processor must be implemented on three F-Series modules maximum

8. System must include support for up to four processors in a single backplane; this implies commensurate:

   o main memory capacity

   o main memory bandwidth

   o system bus bandwidth

   o power, cooling and packaging

9. RAMP metrics must equal or better those of Nautilus

Goals define the product attributes beyond the minimum constraints, and are traded-off in the order in which they appear:

1. Single-processor performance six times the 11/780

2. Implement processor on two F-Series modules

3. Support up to six processors in a single backplane

4. FCS in March, 1987

5. $15K Kernel transfer cost

6. Support up to 4 BI's

1.3 SYSTEM DESCRIPTION

The major components of the Frigate System are implemented on F-Series (Nautilus) modules, which plug into the FSB (Frigate System Bus) Backplane. A FSB to BI Adapter Module, located in the FSB backplane, provides the interface to one or two BI backplanes via cables and a BI module located in each of the BI backplanes (Nautilus-style).

1.3.1 Frigate CPU

The Frigate CPU consists of F-Series modules which plug into the FSB Backplane. The CPU executes the entire VAX instruction set (not including compatibility mode); this includes hardware support for F, D, and G_Floating, with H_Floating and Decimal instructions implemented in microcode. The CPU is implemented as a five-stage machine with a basic cycle time of 100 nsec. To achieve the stated performance constraint (4x 11/780), Frigate must retire an instruction every 500 nsec, or once every five cycles.

The first stage in the Frigate processor pipeline accesses the Instruction TB and Cache for I-Stream data. The second stage decodes the I-Stream data at the rate of one instruction and one specifier, or one specifier (for subsequent specifiers), per cycle, for most specifiers (including context indexing). The next stage calculates operand addresses. The fourth stage accesses the Data TB and Cache for operand values, and the final stage executes instructions and writes results.

All writes are under the control of the final, execution stage. To facilitate pipelining, Write-In-Progress indications are maintained in the General Registers and Data Cache as appropriate.

The caching scheme used in the Frigate CPU includes separate Instruction and Data Caches, both implemented using a custom chip, called the Frigate TB/Cache Chip. Both the I and D Caches are fully associative and include least-recently-used replacement. They are organized as 512 quadwords (4 KB), with a fill size of 2 cache lines (128 bits).

The Data Cache is write-back, and implements the scheme utilized in Firefly under development at SRC. This scheme requires the storing of two additional bits with each cache line:

1. MODIFIED - indicates that this cache line has been locally modified

2. SHARED - indicates that another cache has a copy of this data

A CACHED signal on the FSB is asserted if read data is supplied by another cache when a Read With Cache Intent function is executed. This causes the SHARED bit to be asserted for this cache line.

The CPU is described in detail in the next chapter.

1.3.2 Frigate System Bus (FSB)

The Frigate System Bus (FSB) is a synchronous bus interconnecting the CPU(s), the Main Memory Controller, and the Frigate BI Adapter. The FSB is centrally arbitrated and controlled, and is TTL-based, utilizing FAST-family drivers and receivers. The basic cycle time is 100 nsec, and the data path is 64 bits wide, resulting in a theoretical bandwidth of 80 MB. The FSB can support up to six Frigate processors, a dual-BI Adapter, and the memory subsystem.

Address and Command information is time-multiplexed with data transfers; up to two quadwords may be transferred for every Command/Address cycle, yielding an effective bus bandwidth of 2/3(80MB) = 53 MB. Up to two transactions may be in progress on the bus at any time. All transfers are naturally aligned, such that no data rotation logic need be implemented in the memory controller.

The FSB supports the following basic transactions:

1. Read Quadword / Octaword
2. Read Quadword / Octaword Interlocked
3. Read Quadword / Octaword With Cache Intent
4. Write Quadword / Octaword
5. Write Quadword / Octaword Unlock
6. Write Quadword / Octaword Cached
7. Read Word / Longword
8. Read Word / Longword Interlocked
9. Write Masked Long
10. Write Masked Long Unlock

The FSB also includes the capability to handle various types of interrupts, including Interprocessor Interrupts.

1.3.3 Frigate Memory Controller

The Frigate Memory Controller is implemented on a single module, and provides the interface and control between the FSB and up to eight memory array modules, thereby controlling up to 128 Mbytes of Frigate system memory. It accepts commands, addresses, and data from the FSB, generates and checks ECC, and multiplexes this information to the Frigate memory arrays. Additionally, it decodes addresses to select the proper memory array to be enabled for a particular operation, and attempts to hide refresh cycles by selecting idle arrays for refresh.

With the Frigate writeback cache scheme causing only aligned 64-bit memory words to move between caches and the memory subsystem, the memory controller needs to implement only three basic operations:

1. Aligned octaword read,
2. Aligned quadword write, and
3. Aligned octaword write.

The memory controller also implements interlock capability on a memory line basis.

All memory ECC generation and checking is done by the memory controller. ECC checking is done in an 'offline' manner. Read data is assumed good, and transmitted to its destination while ECC checking takes place. If an ECC error is discovered, it is reported, and the data transfer is aborted. This effectively removes the time spent checking ECC from the critical path of memory reads, thereby increasing system throughput.

1.3.4 Frigate Memory Arrays

Frigate memory arrays are organized as arrays of 64 bit, ECC-corrected words. Since the Frigate writeback cache scheme causes write data to be merged in the cache, only 64-bit cache lines move between cache and memory, simplifying both the system bus and memory subsystem design. Memory arrays support 64-bit reads and writes only, allowing the economy of 64-bit ECC without any overhead due to read/modify/write operations.

Memory array cards will contain up to 16 MB of DRAM. This implies multiple 'banks' of 64-bit-wide arrays per array card, such that it is practical to simultaneously access two memory 'banks', and with buffering, to do aligned octaword reads and writes to memory. Utilizing octaword transfers increases maximum memory bandwidth from roughly 16M bytes/second to almost 32M bytes/second, and both the CPU cache fill size and FBI adapter operations are designed to capitalize on this feature.

1.3.5 Frigate BI Adapter (FBI)

The Frigate BI Adapter (FBI) serves as the interface between the Frigate System Bus (FSB) and up to two BI's. It is very similar to the Nautilus BI Adapter (NBI), in that it consists of two types of modules. The first, identified as the FBIA, is implemented on an F-Series module and resides in the FSB backplane. The other, identified as the FBIB, is a BI module which resides in a BI backplane. An FBIB is required for each BI included in a Frigate system (up to two). The FBIA connects to one or two FBIB modules as appropriate via cables.

The FBI:

1. Appears as a memory node to DMA devices on the BI;

2. Handles CPU memory requests in BI address space as a BI processor node;

3. Fields BI device interrupts

1.4 TECHNOLOGY

There are several key components utilized in the implementation of Frigate which are worthy of mention here.

1.4.1 F-Series Modules

Frigate uses the F-Series modules pioneered in Nautilus. These modules are similar in size to extended hex modules, and interface to the FSB backplane through 480-position ZIF connectors mounted on one side of the card.

1.4.2 Custom VLSI Chips

Frigate is envisioned to have at least two Custom VLSI Chips based on the Hudson CMOS I Process:

1.4.2.1 Frigate TB/Cache Chip

This chip implements a 128-entry, fully-associative cache structure, including least-recently-used replacement. It includes logic for data and address paths up to 32 bits wide, single cycle clear, locking of cache locations, and support of a write-back cache algorithm. The part is cascadeable in both width and depth. Each Frigate CPU includes 20 such chips.

1.4.2.2 Frigate FIFO Chip

This chip implements a 32-bit-wide by 16-entry-deep First-In-First-Out storage structure. It has separate input and output ports, and explicit control inputs for reading, writing, and advancing the FIFO. Standard cells or the Genesil design approach are possibilities here.

Additionally, an FPU chip based on the Hudson CMOS I process will be pursued, perhaps based on the MicroVAX FPU Chip. Alternatively, modifications to the Weitek 1164 Floating Point Multiplier and 1165 Floating Point ALU to completely implement DEC Floating Point are being worked with Weitek.

1.4.3 AMD 29300 Family

The main ALU and Register File in the Execution Unit are implemented using the AMD 29332 and AMD 29334, respectively. These parts have TTL-compatible outputs. The AMD 29332, packaged in a 168-pin PGA, implements a 32-bit data path, with two 32-bit input ports, and a 32-bit output port. The AMD 29334 is a dual-port-read, dual-port-write Register File, organized as 64 16-bit locations. It is packaged in a 120-pin PGA. Two Register File parts are utilized in each Frigate processor.

1.4.4 RAM Technology

Main memory RAMs will be industry-standard 256 Kbit dynamic RAMs; provision will be included to utilize the 1 Mbit RAMs when they become available.

The Frigate CPU is also dependent on the availability of 35 nsec 16K RAMs, preferably organized as 2Kx8.

1.4.5 Bipolar Gate Arrays

Wherever possible, the remaining logic is sized using the Motorola 2800ALS Bipolar Gate Array. This array uses the Mosaic II technology with three-layer metal, offering ECL internal speeds with FAST-equivalent I/O speeds, for about 3 watts of power. Current packaging is in 149-pin PGAs; alternative packaging will be pursued. Each Frigate CPU utilizes approximately 12 such gate arrays, of which 7 are unique designs.

1.4.6 Miscellaneous Buffering And Control Logic

The remainder of the data buffering and control logic will be accomplished with off-the-shelf components from the FAST logic family.

CHAPTER 2

FRIGATE PROCESSOR

2.1 FRIGATE CPU OVERVIEW

The Frigate CPU is a five-stage machine with a basic cycle time of 100 nsec. To achieve the stated performance goal, Frigate must retire an instruction every 500 nsec, or once every five cycles.

The five pipeline stages are:

1. Prefetch
2. Decode
3. Address Add
4. Operand Fetch
5. Execute

The CPU includes separate Instruction and Data Translation Buffers and Caches. The instruction TB and Cache are utilized by the Prefetch Stage to supply instruction stream data, while the Data TB and Data Cache, which is write-back, are used by the Operand Fetch Stage to access operands. Both TB/Cache structures use the same TB/Cache Chip as a building block; the chip organization is fully associative, with least-recently-used replacement.

The CPU also includes a four-port-readable and single-port-writeable general register structure. Two copies of the GPR's are accessed during operand specifier decode, and two other copies are available to the execution unit. The write logic is shared, and is under the control of the execution unit.

Write-In-Progress indications are associated with the Data Cache and the GPR's accessed during operand specifier decode. These are necessary to utilize the pipeline efficiently while providing only one write path to these structures; this write path is under the control of the execution unit, such that all register logging and fault recovery logic is implemented in one place.

2.1.1 Prefetch Stage

The Prefetch Stage uses a Prefetch PC to access the Instruction TB and Cache in parallel to continually supply instruction stream bytes to the Decode Stage via the Instruction Buffer. Major logic elements include the Instruction TB (ITB), Instruction Cache (ICache), Instruction Buffer Shifters, and the Instruction Buffer (IB).

Two cycles are required to traverse this stage from virtual address to IB data - one cycle to access the ITB for a physical address, and a second cycle to access the ICache and load the IB. However, hardware is implemented to access the ITB and ICache in parallel, and the extra cycle is paid only when accessing a different page than that previously accessed.

2.1.2 Decode Stage

The Decode Stage operates on the contents of the Instruction Buffer (IB) to parse an opcode and specifier or a subsequent specifier every cycle. Two copies of the GPRs are accessed for base and index register values as appropriate. In addition to the two GPR copies, this stage includes substantial opcode and specifier decode logic and the Decode PC Adder.

2.1.3 Address Add Stage

The Address Add Stage forms operand addresses (or passes operand data) from the specifier information presented by the Decode Stage. The major logic structure is a three-input adder; each input is presented via a mux which formats data based on control information supplied by the Decode Stage.

2.1.4 Operand Fetch Stage

The Operand Fetch Stage accesses the Data TB and Cache to obtain operands from memory. The Cache may be read AND written once every cycle. Writes occur during the first half of the cycle - data is presented via the DCache Data Bus, and the write (and WIP bit clear) occurs at the address specified on the DCache Address Bus. In parallel, the address generated by the Address Add Stage is presented to the TB and translated. During the second half of the cycle, the resulting physical address is used to access the cache to yield data and/or to mark a cache line as Write-In-Progress (WIP).

The Operand Fetch Stage also stages operands and builds the first microword (as a function of the specifier and opcode) for the Execution Stage.

2.1.5 Execution Stage

The Execution Stage operates on data supplied by the Operand Fetch Stage to produce the result specified by the instruction. All paths and register elements within this unit are 32-bits wide; the intent is to operate on two 32-bit input operands and produce a 32-bit result every cycle. Results are presented via the Write Latch and Write Bus; all writes to the GPR's and memory are under the control of this unit. Major elements of this unit include the main ALU and barrel shifter, a floating point unit (FPU), a set of Working Registers which includes copies of the GPRs (dual-port-readable, single-port-writeable), a set of Operand Buffers, a Register Log, and the main Control Store and microsequencer.

2.1.6 System Clock

The system clock structure consists of a free-running, four-phase clock with a 100 ns period. The four phases are labelled T0, T1, T2, and T3. The leading edge of T0 defines the start of a cycle. The leading edge of T1 occurs at T0 + 25, the leading edge of T2 occurs at T0 + 50, and the leading edge of T3 occurs at T0 + 75.

2.1.7 Pipeline Registers And Stall Signals

The output of each stage in the machine pipeline is a set of pipeline registers; these registers are also the input to the next stage in the pipeline. The pipeline registers at the output of each stage are manipulated by that stage as a group. Associated with each set of pipeline registers is a VALID bit, which indicates when these registers contain valid data; that is, when their contents may be used by the succeeding stage.

Each stage also implements a STALL signal, which is propagated to the previous stage in the pipeline. The assertion of such a signal indicates that a stage was unable to process the data presented by the preceding stage within the current cycle, such that the input state (ie: pipeline registers) must be maintained. The effect is to freeze the pipeline registers between the two stages. The propagation of these signals "back up the pipe" is one of the critical paths in the machine, and is integrally related to the clocks and clock skew.

A pipeline stage that cannot accomplish work because the pipeline registers at its input are not VALID is said to be IDLE. A pipeline stage which cannot accomplish work because the pipeline registers at its outputs are not available, as indicated by a STALL signal from the succeeding stage, is said to be STALLED. Conditions exist in the machine which cause a particular stage to cease processing until some sort of outside intervention occurs; a stage in this situation is said to be in the WAIT state.

2.2 FRIGATE TIMING CHART

The following timing chart illustrates the critical paths in Frigate from a timing standpoint.

```
Prefetch   0 ********************************************************************
              ICache Page Reg Prop  | Control Register Prop | Write Bus Prop
          10 -----------------------|-----------------------|     and Dist
             --------------------------------------------------------------------
          20   ICache Address Buf Prop      |    Write Bus Prefetch Latch Prop
             -------------------------------|------------------------------------
          30                                |        ITB Address Latch Prop
                 I Cache RAM Address        |------------------------------------
          40         Setup (Read)           |
                                            |       ITB RAM Address
          50 *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*         Setup (Read)
              Ctrl Reg Prop   |ICache Output|
          60 ----------------|  Latch & Dist|------------------------------------
             ICache Addr Buf |-------------| Data Comp | Addr Comp |
          70 ----------------|             |   Prop    |   Prop    | Protection
                             |             |-----------------------| Logic Prop
          80   ICache RAM    |  IB Shift   |        XXXXX          |
              Access (Write) |  and Setup  |------------------------------------
          90                 |             |
             Write Recovery ?|             |      Prefetch Valid Bit Setup
Decode     0 ********************************************************************
                  IB Propagation Delay          |         Write Bus Prop
          10 ---------------------------------|             and Dist
                                               |-----------------------------------
          20                                   |         Address Mux Prop
                   First-Level IB Decode       |-----------------------------------
          30                                   |
             ----------------------------------|          GPR Write Access
          40                                   |
                                               |
          50      Second-Level IB Decode       |-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
                                               |          Control Reg Prop
          60 ----------------------------------|-----------------------------------
             ----------------------------------|          Address Mux Prop
          70                  |                |-----------------------------------
                              |                |
          80 IB Shift Control |   Decode PC    |          GPR Read Access
                  Setup       |   Adder Prop   |
          90                  |
                              |95 ---------------------------------------------------
                                                  Register Setup
Add        0 ********************************************************************
                 Pipeline Register Prop         |         Write Bus Prop
          10 ----------------------------------|           and Dist
                                               |-----------------------------------
          20             xxxx                  |         Deskew Latch Prop
             --------------------------------------------------------------------
          30                  Address Adder Input Mux Prop
             --------------------------------------------------------------------
```

```
         40

         50
                            Address Adder Prop
         60

         70

         80
            -------------------------------------------------------------------
         90                 Virtual Address Mux Prop
                                  and VAR Setup
Fetch     0 *******************************************************************
                  VAR Prop (Read)          | PAR Prop (Write)| Write Bus Prop
         10 -------------------------------| and distribution|   and dist
                                           |-----------------|----------------
         20     TB RAM Address Setup       |      xxxx       | Rotator Prop
                      (Read)               |-----------------|----------------
         30                                |                 |DCache DBus Prop
            -------------------------------|Cache RAM Address|----------------
         40    DCache Address Bus Prop     | Setup  (Write)  | Cache RAM Data
                    and PAR Setup          |                 |    Setup
         50 *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
                PAR Prop (Read/WIP)        |
         60      and distribution          |
            -------------------------------|
         70                                |
                Cache RAM Address Setup    |
         80          (Read/WIP)            |   PA Valid for Cache Write
                                           |-----------------------------------
         90 -------------------------------|   DCache Address Bus Prop
             DCache Data Bus Prop, MDR Setup|        and PAR Setup
Execute1  0 *******************************************************************
            Uword Reg Prop | Operand Ptr and Control Reg Prop|  MDR Reg Prop
         10 ---------------|---------------------------------|-----------------
                                           |                 |
         20   Register File Address Setup  |Operand Register|  Rotator Prop
                                           | Address Setup  |
         30                                |                 |
            -------------------------------------------------------------------
         40             ALU Input Mux Prop                  |
            ------------------------------------------------|
         50                                                 |
                                                            |
         60                                                 |    xxxx
                                                            |
         70                  ALU Prop                       |
                                                            |
         80                                                 |
                                                            |---------
         90                                                 | OR RAM
                                                            |Data Setup
Execute2  0 *******************************************************************
            Write Latch Prop |Uword Reg Prop | CC Latch Prop |
 10 -----------------|---------------|---------------|
                     |               |  Branch Mux   |
 20      xxxx        |               |---------------|
    *-*-*-*-*-*-*-*-*|     xxxx      | CS Address Mux|
 30 Deskew Latch Prop|               |---------------|
    -----------------|               | CS Addr Latch |
 40                  |               |---------------|
                     |               |               |
 50                  |*-*-*-*-*-*-*-*|               |
        xxxxxx       | Ptr Reg Prop  |               |
 60                  |---------------|               |
                     |---------------| Control Store |
 70                  |               |  RAM Access   |
    -----------------| Register File |               |
 80   Register File  | RAM Addr Setup|               |
      RAM Data Setup |               |---------------|
 90 -----------------------------------| Ptr Mux Prop, |
           Write Recovery Time        |Uword Reg Setup|
100 ***************************************************************
```

Note: Execute2 occurs in parallel with Execute1 - it just wouldn't fit horizontally on the page!

2.3 PREFETCH STAGE

2.3.1 Prefetch Bus

The Prefetch Bus supplies bits <31:0> of a virtual address to the Instruction TB and Cache structure, from one of four possible sources:

1. The Write Bus via the Write Bus Prefetch Buffer
2. The Prefetch PC Register
3. The Branch Register
4. The Jump Register

The default Prefetch Bus driver is the Prefetch PC Register, which supplies the address of the next quadword beyond the one currently being processed. The bus enable lines are a function of the opcode and specifier decode logic and the execution unit (for TB management, branch correction, and interrupts).

2.3.2 ITB Address Latch

The ITB Address Latch captures bits <31:0> of the Prefetch Bus for presentation to the ITB and ICACHE. Bits <31:9> are presented to the ITB and bits <8:0> are presented directly to the ICACHE. The latch is open from T1 to T2, and is implemented to efficiently handle the capture of addresses supplied via the Write bus (the Write Bus Latch opens from T3 to T0). On a cache miss or access violation, the contents of this latch are passed down the pipeline to the execution unit via the ICache Address Latch and the IPC FIFO.

2.3.3 Instruction Translation Buffer (ITB)

The Instruction Translation Buffer (ITB) consists of a tag store and data store, and two sets of comparators. ITB Address Latch bits <31:9> are used to access the ITB for the corresponding PTE, from which bits <29:9> of a physical address are extracted. These bits are compared with ICache Page Register <29:9> to determine if the physical address presented to the Instruction Cache in parallel was valid. The validity of this physical address is also conditional on whether a TB hit occurs, and whether the access is allowed, as defined by the protection bits and processor mode.

The ITB itself consists of 256 entries, 128 for system space and 128 for process space, and is implemented with two custom Frigate TB/Cache chips. The organization is fully associative, with least-recently-used replacement.

2.3.3.1 Reading The ITB

Accessing the ITB to read a PTE proceeds as follows:

1. ITB Address Latch <31:9> are presented to the ITB to yield the corresponding PTE; ITB Address Latch <31> selects system or process space (one of the two TB/Cache chips).

2. In parallel:

    1. PFN<20:0> from the accessed PTE are compared with bits <29:9> of the ICache Page Register to determine if the physical address used to access the cache in parallel was correct;

    2. The Protection bits from the accessed PTE are examined relative to the processor mode to determine whether the access is allowed.

3. PTE <20:0> are unconditionally loaded into ICache Page Register <29:9> at the end of the cycle.

4. A valid indicator associated with the ICache Page Register is asserted as follows:

1. If the PTE is cached, the access is allowed, and the addresses match, the valid indication is asserted;

2. If the PTE is cached, the access is allowed, and the addresses don't match, the valid indication is not asserted - the result is that the cache access will be repeated in the next cycle, this time with the correct address;

3. If the PTE is cached but the access is not allowed, or the PTE is not cached, the valid indication is not asserted. Additionally, the contents of the ICache Address Latch are loaded unmodified into the IBPC Register (once the IB has been emptied), and a TB error indication is presented to the Decode Logic. The Decode logic propagates this error indication thru the control register pipeline, and the virtual address is propagated via the IPC FIFO, to the Execution Stage. The Prefetch unit then waits for execution unit intervention.

The ICache Page Register valid indication is ANDed with the Cache Hit signal from the parallel cache access to indicate to the IB Shift logic that the accessed cache line contains valid Instruction Stream data.

2.3.3.2 Writing The ITB

The ITB is always written under the control of the Execution Unit. The address to be written is presented via the Write Bus and Write Bus Prefetch Buffer and loaded into the ITB Address Latch (in fact, in most cases the address will already be there). In the next cycle the PTE to be written is presented via the Write Bus through a deskew latch, and the write into the appropriate ITB chip occurs, based on ITB Address Latch <31>.

When mapping is not enabled, the ITB will be managed as an identity map by the Execution Unit microcode.

2.3.4 ICache Page Register

The ICache Page Register is loaded every cycle from PFN <20:0> of the PTE accessed in the ITB. The output of this register is presented to the ICache Address Mux as bits <29:9> of the cache address to be read during the first half of every cycle. During the second half of the cycle, its contents are compared with the output of the ITB to determine if the cache access was valid.

2.3.5 ICache Address Bus

The ICache Address Bus consists of two portions, a page portion, which supplies bits <29:9> of the physical address to the cache, and the offset portion, which supplies bits <8:0> of the physical address.

The page portion has two sources, the ICache Page Register, which supplies the read address during the first half of every cycle, and the ICache Address Bus Latch, which supplies the write address during the second half of the cycle.

The offset portion also has two sources. During the first half of the cycle, the read address is sourced from ITB Address Latch <8:0>. During the second half of the cycle, the ICache Address Bus Latch supplies the write address.

Bits <29:3> of the ICache Address Bus are presented to the ICache for reading or writing the quadword cache lines. Bits <29:0> of this latch may also be driven onto the Internal Bus to access main memory on a cache miss.

[Note: The critical path reading the instruction cache is when the address is supplied via the write bus (bits <8:3> only). This path could be handled differently if it proves to be a problem as currently implemented. Note that no state element is necessary in this path since the cache output itself is captured at T2, and the Write Bus Latch does not re-open until T3.]

2.3.6 Instruction Cache

Bits <29:3> of the ICache Address Bus are used to access the Instruction Cache for the next quadword in the instruction stream. The Instruction Cache is implemented utilizing 8 Frigate TB/Cache chips, organized as 64 bits wide (+parity), and 512 locations deep (4 KB). It is fully associative, and implements the least-recently-used replacement algorithm.

The cache is accessed during the first half of every cycle for read data, regardless of the state of any valid / not valid indications; the cache output is captured at T2 in the ICache Output Latch. The cache hit/miss indication is ANDed with the ICache Page Register valid indication to generate an indication regarding the validity of the accessed cache line. This signal, ICACHE_LINE_VALID is used by the instruction buffer control logic to conditionally mark bytes valid in the Instruction Buffer at the end of the cycle.

The ICache may be accessed during the second half of each cycle to do a write or invalidate. The write address comes from the ICache Address Bus Latch, and the write data is sourced onto the ICache Data Bus from the Internal Bus via the Internal Bus Buffer.

2.3.7 ICache Output Latch

The ICache Output Latch captures the 64-bit output of the Instruction Cache. It is conditionally opened from T1 to T2 under the control of the Instruction Buffer control logic whenever the previous contents of the latch have been used by the IB, or are no longer needed (such as after a taken branch); this is indicated by the assertion of the signal LOAD_CACHE_LINE_BUF (defined below).

[Note: Conditioning the opening of this latch allows the following cache line to be accessed before the current contents of the ICache Output Latch have been loaded into the Instruction Buffer. Otherwise, the latch could be unconditionally opened every cycle.]

This latch is implemented in the Instruction Buffer Gate Arrays.

2.3.8 Instruction Buffer And Shifters

The Instruction Buffer and Shifter structure is utilized to provide the next 7 bytes in the instruction stream to the Decode Stage. The shifters are used to accomplish this by selecting the appropriate 7 bytes from the 22 bytes formed from the Instruction Buffer (14 bytes) and the output of the Instruction Cache latched in the ICache Output Latch (8 bytes).

The Instruction Buffer and Shifter is implemented through the use of four Instruction Buffer Gate Arrays. The structure is partitioned vertically, such that each gate array implements the entire structure for two of the eight bits in each byte. These gate arrays also include the ICache Output Latch.

2.3.9 ICache Address Latch

The ICache Address Latch is loaded from the Prefetch Bus whenever the signal LOAD_CACHE_LINE_BUF (defined below) is asserted. This latch captures the virtual address associated with the quadword in the ICache Output Latch. Its output is presented to the PC Offset logic to calculate the next Prefetch PC and IBPC.

[Note: This latch exists only to allow prefetching another cache line beyond the contents of the ICache Output Latch].

2.3.10 Prefetch PC Incrementer And Register

The Prefetch PC Incrementer supplies the contents of the ICache Address Latch + 8 to the Prefetch PC (PPC) Register every cycle. The Prefetch PC Register is the default driver of the Prefetch Bus.

2.3.11 Instruction Buffer PC Register (IBPC Register)

Associated with the Instruction Buffer (IB) is an Instruction Buffer PC Register, which identifies the next byte in the instruction stream to be operated on by the Decode Unit (ie: the byte in position 0 or 1 of the Instruction Buffer). This register is input to the Prefetch PC Adder in the Prefetch Stage, and the Decode PC Adder in the Decode Stage.

The IBPC Register is loaded as follows:

1. Bits <31:3> are loaded from the ICache Address Latch if the signal LOAD_CACHE_LINE is asserted; otherwise, the bits are re-circulated from the corresponding IBPC outputs;

2. Bits <2:0> are loaded from either the Prefetch PC Adder or the ICache Address Latch, depending upon whether or not a branch occurred, as indicated by the signal BRANCH_ADDRESS_BUF.

This logic is implemented in the PC Gate Array.

2.3.12 Prefetch PC Adder

The 3-bit Prefetch PC Adder is used to calculate the address of the first byte beyond the opcode (for instructions with no specifiers) or specifier currently being decoded (that is, the address of the first byte of the the opcode or specifier to be decoded in the next cycle). One input to the adder is Instruction Buffer PC Register <2:0>. The other input is PC_OFFSET<2:0> from the specifier decode logic.

The output of the Prefetch PC Adder is conditionally loaded into bits <2:0> of the IBPC Register at the end of the cycle.

Carry-out of this adder is one of the signals OR'd to generate the signal LOAD_CACHE_LINE. This signal causes bits <29:3> of the ICache Address Latch to be loaded into the corresponding bits of the IBPC.

This logic is implemented in the PC Gate Array.

[Note: The updating of the PC after quadword and octaword immediates is handled in the Decode PC Adder and via the Branch Register.]

2.3.13 Miscellaneous Control Lines

ICACHE_LINE_VALID - indicates that the contents of the ICache Output Latch are valid.

PREFETCH_PREGISTERS_VALID - specifies that the Prefetch Pipeline

Registers are valid.

PREFETCH_STALL - pipeline stall from Decode Stage; inhibits the updating of the Prefetch Pipeline Registers.

PC_OFFSET <2:0> - output of Decode Stage which specifies the number of bytes consumed by the Decode Stage, and therefore the number of bytes by which the Instruction Buffer PC is to be incremented and the Instruction Buffer is to be advanced, ie: right-shifted.

LOAD_OPCODE - output of Decode Stage which specifies that byte 0 of the Instruction Buffer is to be loaded. IB0 always contains the opcode of the instruction being decoded, and is therefore not loaded every cycle.

BRANCH_ADDRESS - signal from decode stage indicating that the next prefetch address is not-sequential. Used to select the source of the Prefetch Bus and to generate the signal LOAD_CACHE_LINE.

BRANCH_ADDRESS_BUF - Registered version of BRANCH_ADDRESS used in the next cycle to select the source of IBPC<2:0>.

LOAD_CACHE_LINE - the OR of the carry from the Prefetch PC Adder and BRANCH_ADDRESS. Specifies that IBPC Register bits <29:3> be loaded with the output of the ICache Address Latch.

LOAD_CACHE_LINE_BUF - Registered version of LOAD_CACHE_LINE used in the next cycle to conditionally open the ICache Output Latch and ICache Address Latch.

IBn Valid - valid bits associated with each of the bytes in the Instruction Buffer, IBn where n = 0 to 13.

2.4 DECODE STAGE

The Decode Stage operates on the contents of the Instruction Buffer (IB) and Instruction Buffer PC Register (IBPC) to generate operand information for the Address Add Stage, and control information for the Prefetch Stage. Major structures include the operand decode logic, the specifier decode logic, the Decode PC Adder and two copies of the General Registers.

2.4.1 Instruction Buffer

The Instruction Buffer (IB) is 14 bytes wide. Up to 7 bytes (the low-order bytes) may be consumed by the Decode Stage in any one cycle; the remaining 7 bytes in the buffer are used to insure that, whenever possible, the next 7 bytes in the IStream will be available to the decode logic in the next cycle. Each byte may be loaded from any succeeding byte in the IB and ICache Output Latch.

The 7 bytes available to the decode logic are identified as IBn, n = 0 to 6. IB0 always contains the opcode of the instruction being decoded.

The Instruction Buffer and Shifter is implemented through the use of four Instruction Buffer Gate Arrays. The structure is partitioned vertically, such that each gate array implements the entire structure for two of the eight bits in each byte. These gate arrays also include the ICache Output Latch.

[Note: The width of the IB beyond the 7 bytes which may be used by the Decode Stage in any one cycle is a function of the fact that the ICache Output Latch which supplies the IB cannot be re-loaded until all 8 bytes in the Latch have been used. Simulation is under way to determine the optimum IB length.]

2.4.2 Instruction Decode

The instruction decode unit is implemented as a state machine with twelve major states corresponding to the type of decode being performed. The twelve states are defined by four state signals, collectively called the decode state, which are generated each cycle by the state machine:

1. SECOND_OPCODE - when asserted, indicates that the opcode byte being decoded is the second byte of the opcode; deasserted indicates that the byte is to be interpreted as the first byte of the opcode.

2. SPECIFIER<2:0> - identifies the position of the specifier being decoded, as follows:

    1. 000 - First Specifier - decode first specifier

    2. 001 - Second Specifier - decode second specifier

    3. 010 - Third Specifier - decode third specifier

    4. 011 - Fourth Specifier - decode fourth specifier

    5. 100 - Fifth Specifier - decode fifth specifier

    6. 101 - Sixth Specifier - decode sixth specifier

    7. 110, 111 - Undefined

The quiescent state of the machine is SECOND_OPCODE not asserted and SPECIFIER<2:0> = 0, specifying decode first specifier (the machine enters this state upon initialization, and at the end of each instruction). In this state, there are six possible interpretations of the data in the instruction buffer (ignoring quadword and octaword immediates for the moment).

[Note: This assumes that there is no decode of the instruction stream done on the input side of the instruction buffer to align data - alignment is solely under the control of control logic in the decode stage on the output side of the instruction buffer. To do otherwise would add intolerable delay in the prefetch stage.]

The six possible data formats are:

1. Single-byte opcode, no specifiers

x	x	x	x	x	x	x	Opcode

2. Single-byte opcode, first specifier is not index mode

x	x	Disp 3	Disp 2	Disp 1	Disp 0	Base SPC	Opcode

3. Single-byte opcode, first specifier is index mode

x	Disp 3	Disp 2	Disp 1	Disp 0	Base SPC	IndexSPC	Opcode

4. Single-byte branch opcode

x	x	x	x	x	Disp 1	Disp 0	Opcode

5. Double-byte opcode, first specifier is not index mode

x	Disp 3	Disp 2	Disp 1	Disp 0	Base SPC	Opcode 1	Opcode 0

6. Double-byte opcode, first specifier is index mode

Disp 3	Disp 2	Disp 1	Disp 0	Base SPC	IndexSPC	Opcode 1	Opcode 0

The double-byte opcode cases are handled by detecting these formats (IB0 = FD), specifying a single-byte advance (shift) of the instruction stream, and specifying that the next decode state be SECOND_OPCODE asserted, SPECIFIER<2:0> unchanged (specifying decode first specifier). The effect is that the instruction buffer contents in the next cycle will be identical to one of the single-byte opcode formats, with the decode state specifying SECOND_OPCODE.

The net result of this strategy is that there are only four possible interpretations of the instruction buffer by the specifier decode logic when SPECIFIER<2:0> indicates that the first specifier is to be decoded. This means:

1. The index mode specifier, if there is one, always appears in IB1;

2. The base operand specifier appears in IB2 or IB1;

3. Displacements appear as follows:

    1. Branch Displacements in IB1 or IB2|IB1;

    2. Byte Displacements in IB2 or IB3;

    3. Word Displacements in IB3|IB2 or IB4|IB3;

    4. Longword Displacements, or the first longword of a quadword or octaword immediate, in IB5|IB4|IB3|IB2 or IB6|IB5|IB4|IB3.

To minimize the multiplexing in the decoder stage, it is desirable to maintain this positioning for all decode states, that is, independent of which specifier is being decoded.

This is done by shifting one less byte-position when transitioning into a SPECIFIER state other than that specifying the decode of the first specifier.

In this manner, an opcode and specifier may be decoded in one cycle, for all cases except quadword and octaword immediate mode specifiers. One additional cycle is required for the quadword immediate case, and three additional cycles for the octaword immediate case. During such cycles, the immediate value is extracted one longword at a time from IB4|IB3|IB2|IB1.

2.4.3 Opcode Decode Logic

The opcode decoder implemented in the Decode Stage is logically a 4K by n PROM with the following inputs:

1. IB0<7:0>

2. DECODE_STATE<3:0> (=SECOND_OPCODE|SPECIFIER<2:0>)

It's output includes the following fields:

1. DATA_TYPE<3:0> - specifies the data type of the specifier being decoded; DATA_TYPE<3:0> = F is used to indicate that the instruction has no specifiers (ie: no first specifier exists);

2. ACCESS_TYPE<2:0> - specifies the access type of the specifier being decoded;

3. NEXT_DECODE_STATE<3:0> - specifies the next decode state to be loaded into the Decode State Register.

4. LOAD_OPCODE - specifies that an opcode is to be loaded into IB0; this affects the PC Offset, and gates the clocking of IB0.

Additionally, the opcode decode logic will decode instruction class, particularly as it affects pipeline waits.

2.4.4 Specifier Decode Logic

IB1 contains the index specifier, if there is one, or may contain a base specifier. Additionally, IB2 may contain a base specifier for index mode. The only fast decode needed, therefore, is a determination of whether index mode is specified; the signal INDEX_MODE is asserted if IB1<7:4> = 4. This signal is used as a select on the muxes at the inputs of the base specifier GPRs and the displacement register.

The remainder of the specifier decode is accomplished via a logic structure with the following inputs:

1. IB1<7:4>

2. IB2<7:4>

3. DATA_TYPE<3:0>

4. ACCESS_TYPE<2:0>

5. LOAD_OPCODE

[Note: The 8-bit combination of IB1<7:4> and IB2<7:4> can be reduced to five bits by decoding IB1<7:4> = 4 (Index Mode) quickly, and using it to select a mux between IB1 and IB2.]

The specifier decode logic generates:

1. PC_OFFSET<2:0> - input to the Prefetch Stage to specify the number of bytes to advance (right shift) the Instruction Buffer, and input to the Prefetch PC and Decode PC Adders to calculate the appropriate updated PC's;

2. Control signals for subsequent stages propagated through the machine via Function Registers defined below;

3. Displacement Multiplexer select bits to specify the format of short literals or displacements in the Displacement Register;

4. WIP Control Bits - to specify the number of sequential WIP counters to be incremented or checked in a particular cycle;

5. Branch Control Bits used by the Branch Logic to specify how to set up the Decode PC Adder, whether the Branch or Jump Register is driving the Prefetch Bus, and when the branch address is valid.

2.4.5 Decode PC Adder

The Decode PC (DPC) Adder operates on the contents of the IBPC to generate the updated PC to be used by the Address Add Stage for address calculations, or to generate the target address to be supplied to the Prefetch Stage for some classes of branches.

One input to the DPC Adder is the IBPC Register, which supplies the address of the first byte of the I-Stream to be processed by the Decode Stage in this cycle. The other input is a mux which is setup based on opcode information (branch or no branch), specifier size and specifier type to supply the appropriate addend. The output of the DPC Adder is input to the Branch Register, Decode PC (DPC) Register, and IPC FIFO.

The Decode PC Adder is implemented in the PC Gate Array.

2.4.6 Branch Logic

When a program flow change is to be made, the Prefetch Stage is notified (by the assertion of the signal BRANCH_ADDRESS) that the next I-Stream Address will come from either the Branch Register or Jump Register. In response to this signal, the Prefetch PC Register no longer sources the Prefetch Bus, and a flush of the Instruction Buffer occurs. The latter is accomplished by marking all valid indications invalid.

The Branch Register is loaded from the Decode PC Adder in the Decode Stage. It provides the Branch Address (ie: drives the Prefetch Bus) when the signal JUMP_REGISTER is not asserted by the decode logic.

The next I-Stream address is supplied via this path for all conditional and unconditional branches and loop instructions that are absolute or PC-relative, and as the result of decoding quadword and octaword immediate mode specifiers. For these cases, the Decode PC Adder is set up to supply one of:

1. PC + 1 + SEXT(IB1)

2. PC + 2 + SEXT(IB2|IB1)

3. PC + 8 (quadword immediate)

4. PC + 16 (octaword immediate)

Loop instructions are always predicted to be taken, hence the target branch address is always supplied for these instructions. A 4K x 1 RAM is used for prediction of conditional branches. Bits <13:2> of the address are presented to this RAM, which contains a single bit indicating whether a branch was taken the last time the corresponding location was accessed. Based on this bit, either the target branch address or the incremental address are supplied via the Branch Register. The RAM is corrected by the Execution Unit microcode via the Write Bus when an incorrect prediction is made, and the pipeline is flushed.

The Jump Address Register is loaded from the output of the 3-input adder in the Address Add Stage. It drives the Prefetch Bus when the signal JUMP_REGISTER is asserted. This path is used to supply the next I-Stream address for JMP's and JSB's that are not PC-relative or absolute.

The Branch Logic, except for the Jump Address Register and the Prediction RAM, is implemented in the PC Gate Array.

2.4.7 Decode PC Register

This register presents the address of the first byte in the instruction stream beyond the specifier or opcode (for instructions with no specifiers) being decoded. It is loaded from the output of the DPC Adder, and conditionally drives the Base Bus, which is one of the inputs to the three-input adder in the Address Add Stage. The other source for Base Bus data is the Base Register; the specifier decode logic specifies the source.

2.4.8 Displacement Multiplexer And Register

The Displacement Register is used to pass sign-extended displacements and formatted literals to the Address Add stage. Additionally, when the base specifier mode is autodecrement, the two's complement of the size is passed via this logic.

The inputs to this register are four byte-wide multiplexers, defined as follows:

Spec Mode	Data Type	Byte3	Byte2	Byte1	Byte0
s^#lit	BWLQO	0	0	0	00|IB1<5:0>
	FD	0	0	010000|IB1<5:4>	IB1<3:0>|0000
	G	0	0	01000000	0|IB1<5:0>|0
	H	IB1<2:0>|00000	0	01000000	00000|IB1<5:3>
-(Rn)	B	FF	FF	FF	FF
	W	FF	FF	FF	FE
	LF	FF	FF	FF	FC
	QDG	FF	FF	FF	F8
	OH	FF	FF	FF	F0
INDEX	B	SEXT(IB3<7>)	SEXT(IB3<7>)	SEXT(IB3<7>)	IB3
	W	SEXT(IB4<7>)	SEXT(IB4<7>)	IB4	IB3
	L	IB6	IB5	IB4	IB3
SUBSEQUENT IMMEDIATE (after first longword)		IB4	IB3	IB2	IB1
ALL OTHERS	B	SEXT(IB2<7>)	SEXT(IB2<7>)	SEXT(IB2<7>)	IB2
	W	SEXT(IB3<7>)	SEXT(IB3<7>)	IB3	IB2
	L	IB5	IB4	IB3	IB2

The Displacement Multiplexer and Register are implemented in the Displacement Gate Array. The pinouts include:

1. 48 data inputs (IB<6:1>)

2. 4 data type inputs

3. 3 format control inputs

4. 1 clock enable

5. 1 clock

The Displacement Register must also be capable of supplying 0. The Displacement Register is one input to the three-input adder in the Address Add Stage.

2.4.9 General Processor Registers

Two copies of the GPRs are implemented in the Decode Stage, one to supply the register contents associated with the base specifier, and one to handle the index specifier. Each GPR set may be read AND written once every cycle; writes occur during the first half of the cycle, and reads occur during the second half of the cycle.

Associated with the GPRs in this stage are 15 Write-In-Progress (WIP) counters. These 4-bit counters maintain the number of outstanding writes to the corresponding GPR (except PC). Attempting to read a GPR whose WIP Counter is non-zero for any mode but register mode causes a pipeline stall.

Read and WIP bit addresses for the two GPR copies come from bits <3:0> of IB1 for the copy associated with the index specifier, and ((IB2*INDEX_MODE)+(IB1*/INDEX_MODE)) for the copy associated with the base specifier.

Write Addresses and the corresponding data size come from the address portion of the Write Bus. The three data size bits are used to specify a byte, word, or longword write. Write data comes from Write Bus <31:0>. When a write occurs, the corresponding WIP counter is decremented.

Multiple WIP counters (up to four) may be read and/or incremented in a single cycle, based on the WIP control bits, which are a function of data type, access mode, and specifier type. Only one WIP counter need be decremented per cycle.

A non-zero WIP counter does not cause a stall if the mode is register.

WIP counters may be read AND set during the second (read) half of the cycle (ie: read at the address supplied by the index specifier mux, and read and/or incremented at the address supplied by the base specifier mux). The WIP counters are decremented during the first (write) half of the cycle.

A register bypass mechanism is implemented to gain a cycle when a stall occurs due to a non-zero WIP counter. The base and index register numbers are captured in the Rb and Ri Registers, respectively, for presentation to the Address Add stage. A valid bit associated with each of these registers indicates whether the Decode Stage was able to supply valid register data. If the valid bit is not set, the Address Add Stage stalls and watches the Write Bus for the updated register contents.

The bypass mechanism is utilized only when there is one outstanding write to a requested register. That is, it is used only when one or the other of the base or index registers is not available; it cannot be used when both are not available, since the Address Add stage has no state element available to store the first operand that becomes available while waiting for the second. Thus, when both register values are not available, the Decode Stage stalls until one of them becomes available, and then transmits the specifier information to the Address Add Stage, which waits for the other to become available via the Write Bus.

Similarly, the bypass mechanism cannot be utilized when a WIP counter is greater than one.

Logic prevents the WIP counters from being decremented beyond zero.

2.4.10 Rbase Multiplexer

The Rbase Multiplexer selects the address presented to the base specifier copy of the GPRs; the output of this mux is also captured in the Rb Register. The mux selects one of three possible address sources:

1. IB1<3:0> - read / WIP (increment/check) cycle and not index mode
2. IB2<3:0> - read / WIP (increment/check) cycle and index mode
3. Write Bus Address - write and WIP decrement cycle

2.4.11 Base Register

The 32-bit Base Register contains the contents of the GPR specified by the Rbase Multiplexer. It is loaded directly from the base specifier copy of the GPRs at the end of every cycle. It conditionally drives the Base Bus, which is one of the inputs to the three-input adder in the Address Add Stage. The Base Register drives the Base Bus whenever the contents of a GPR other than PC are required to calculate the base address.

2.4.12 Rindex Multiplexer

The Rindex Multiplexer selects the address presented to the index specifier copy of the GPRs; the output of this mux is also captured in the Ri Register. The mux selects one of two possible address sources:

1. IB1<3:0> - read / WIP (check only) cycle
2. Write Bus Address - write and WIP decrement cycle

2.4.13 Index Register

The 32-bit Index Register contains the contents of the GPR specified in the index operand specifier, that is, Index Register = (IB1<3:0>). It is loaded directly from the index specifier copy of the GPRs every cycle. It is one of the inputs to the three-input adder in the Address Add Stage.

2.4.14 Rb Register

This 4-bit Register is loaded from the Rbase Multiplexer every cycle. It is used to pass the base register number to the Address Add Stage. To implement the register bypass mechanism, a valid bit is associated with this register. If set, the required register contents were supplied by the Decode Stage via the Base Register. If clear, the register contents were not available due to an outstanding write. In this case, the Address Add Stage monitors the Write Bus for the updated register contents, and stalls until such data is available.

2.4.15 Ri Register

This 4-bit Register is loaded from the Rindex Multiplexer every cycle. It is used to pass the index register number to the Address Add Stage, and exists, along with the associated Valid Bit, only to implement the write bus register bypass mechanism. If the Valid Bit is set, the required register contents were supplied by the Decode Stage via the Index Register. If clear, the register contents were not available due to an outstanding write. In this case, the Address Add Stage monitors the Write Bus for the updated register contents, and stalls until such data is available.

2.4.16 IPC FIFO

The IPC Buffers are organized as a 32-bit-wide FIFO, loaded under the control of the Decode Stage, and read under the control of the Execution Stage. These buffers combine the functions of pipeline PC registers and buffering at the output of the Operand Fetch Stage into a single structure. There are 8 32-bit-wide buffers, organized as a FIFO which may be read and written every cycle. Read data is available to the Execution Unit during the first half of the cycle, and the structure may be written by the Decode Stage during the second half of the cycle. This write occurs from the output of the DPC Adder under the control of the decode state machine.

The Virtual PC used to access the ITB is loaded into the IPC Buffers and passed to the Execution Stage for managing the ITB when necessary.

The PC Buffers are not explicitly addressable by the Execution Unit microcode, in that only the Top-of-Fifo is available to be read. However, the removal of an entry from the FIFO is explicitly controlled by the microcode, in order to manage First-Part-Done cases.

[Note1: Currently, this structure is read by the Execution Stage via the Write Bus to save pins. If this proves to be a bottleneck, dedicated pins could be used, in which case the structure could also be moved closer to the Execution Stage.]

[Note2: The depth of this structure is TBD as the result of simulation. It seems like 8 is a reasonable number, effectively allowing us to buffer up to 6 instructions (4 instructions is probably enough), and hold 2 in the pipeline.]

2.4.17 Control Registers

2.4.17.1 Opcode Register

This nine-bit register contains a single bit indicating single- or double-byte opcode, and the contents of IB0. It is loaded every cycle, and is propagated through the machine for use in succeeding stages.

2.4.17.2 Address Add Function Register

This register contains control information to be used by the Address Add Stage to construct the operand or operand address in the next cycle. The control information for the Address Add Stage is fully decoded in the Decode Stage so as to keep decode out of the critical path in the Address Add stage. The register is made up of the following fields:

1. Base Mux Control
    1. Base Register
    2. Decode PC Register
    3. Write Bus
    4. 0
2. Index Mux Control
    1. Index Register
    2. Index Register left shift by 1
    3. Index Register left shift by 2
    4. Index Register left shift by 3
    5. Index Register left shift by 4
    6. Write Bus
    7. Write Bus left shift by 1
    8. Write Bus left shift by 2
    9. Write Bus left shift by 3
    10. Write Bus left shift by 4
    11. 0
3. BASE_VALID - indicates that the Base Register is valid (used by register bypassing logic)
4. INDEX_VALID - indicates that the Index Register is valid (used by register bypassing logic)

2.4.17.3 Operand Fetch Function Register

The contents of the Operand Fetch Function Register are passed through a pipeline register in the Address Add Stage to ultimately be used by the Operand Fetch Unit.

The Operand Fetch Stage must have some knowledge of the specifier type and position, size, and access mode in order to allocate operand buffer locations and pointers appropriately, in addition to being able to access the data TB and Cache. Exact definition and encoding of this information TBD, but worst case it is no more than the 9-bit Opcode Register, four bits of specifier mode, and a synchronization signal (to indicate the start of a new instruction).

2.4.17.4 Execution Function Register

TBD control and status information from the decode logic in the Decode Stage to be passed via pipeline registers to the Execution Stage.

2.4.18 Miscellaneous Control Signals

DECODE_STALL - input to Prefetch Stage to inhibit update of Prefetch Pipeline Registers.

DECODE_VALID - specifies that the contents of the (pipeline) registers at the output of the Decode Stage are valid, ie: usable by the Address Add Stage.

2.5 ADDRESS ADD STAGE

2.5.1 Inputs

The following pipeline registers described in the previous section are inputs to the Address Add Stage; associated with these pipeline registers is the signal DECODE_VALID, which indicates that all of these registers contain valid data.

1. Displacement Register
2. Base Register
3. Index Register
4. Decode PC Register
5. Rb Register - passed unmodified to Operand Fetch Stage, and used in the Write Bus bypass mechanism for the Base Register
6. Ri Register - used only to implement the Write Bus bypass mechanism for the Index Register
7. Address Add Function Register (includes Index and Base Register Valid indications)
8. Opcode Register - passed unmodified to the Operand Fetch Stage

9. Operand Fetch Function Register - passed unmodified to the Operand Fetch Stage

10. Execution Function Register - passed unmodified to the Operand Fetch Stage

The signal FETCH_STALL is received from the Operand Fetch Stage to indicate that the pipeline registers at the output of the Address Add Stage should not be updated.

2.5.2 Outputs

The following pipeline registers are input to the Operand Fetch Stage; associated with these registers is the signal ADD_VALID, which indicates that these registers contain valid data.

1. Virtual Address Register - generally loaded with the output of the adder in the Address Add Stage for presentation to the Operand Fetch Stage.

2. Register Number Register - generally loaded with the unmodified contents of the Rb Register at the output of the Decode Stage.

3. Opcode Register - passed unmodified from Decode Stage

4. Operand Fetch Function Register - passed unmodified from Decode Stage

5. Execution Function Register - passed unmodified from Decode Stage

The Jump Register is loaded with the output of the adder in the Address Add Stage, but is not a pipeline register between the Address Add and Operand Fetch Stages. Rather, it is used to present Jump Addresses to the Prefetch Stage via the Prefetch Bus.

The signal ADD_STALL is input to the Decode Stage to inhibit the loading of the pipeline registers at the input of the Address Add Stage.

2.5.3 Operation

This stage includes a 32-bit 3-input adder. The inputs to the adder are formatted under the control of the Address Add Function Register. This control information results in three operands being input to the adder each cycle; the output of the adder is loaded into the Jump Register and the Virtual Address Register.

2.5.4 Base Multiplexer

The Base Multiplexer is controlled by the Base Mux Control Field in the Address Add Function Register, and the Base Register bypass logic. The Base Mux Field explicitly selects one of the following values to be presented to the Address Adder:

1. Base Register
2. Decode PC Register
3. Write Bus
4. 0

Additionally, when the Base Register is specified as the source and the BASE_VALID indication is not set, the bypass logic monitors the Write Bus for a write to the address specified in the Rb Register. When this address is detected, the data is routed directly through the Write Bus input to the Address Adder.

Note that to facilitate partitioning, the multiplexing between the Base Register and the DPC Register is done using a tri-state bus called the Base Bus.

2.5.5 Index Multiplexer

The Index Multiplexer provides the capability to shift the Index Register or Write Bus by 0, 1, 2, 3, or 4. The Write Bus is never explicitly selected, but is used by the Ri bypassing logic when necessary. The output of this mux may also be forced to zero. The Index Mux Control Field of the Address Add Function Register controls this mux.

2.5.6 Address Adder

The Address Adder is a 3-input adder which produces 32-bit results. The three inputs are:

1. Base Mux <31:0>
2. Displacement Register <31:0>
3. Index Mux <31:0>

The adder always performs a three input add, and its output is loaded into the Virtual Address Register and Jump Register every cycle. Bypassing is accomplished by specifying that the appropriate input mux(s) supply zero.

The Address Adder is implemented in 3 identical Gate Arrays. The low-order 16 bits of the three-input add are accomplished in one gate array, and the add of the high-order 16 bits is done in each of the other two gate arrays, one with carry-in hard-wired low, the other with carry-in hard-wired high. The carry-out from the low-order portion of the operation determines which of the high-order results will be used.

The Address Adder Gate Arrays include the Index and Base Muxes described above, and the corresponding register bypassing logic.

2.5.7 VAR Multiplexer

The VAR Multiplexer presents a virtual address to the Virtual Address (VAR) Register from one of three possible sources:

1. Address Adder

2. Virtual Address Adder

3. Write Bus via Deskew Latch

The Virtual Address Adder provides the capability to add 4 or 8 to the contents of the VAR under the control of the Operand Fetch state sequencer or Execution Unit microcode. The Write Bus is selected as the source under Execution Unit microcode control.

2.5.8 Virtual Address Register

The Virtual Address Register (VAR) is loaded from the VAR Multiplexer. The output of the VAR is presented to the Data TB and Cache structure for accessing memory operands, and is one of the inputs to the Bypass Multiplexer.

2.5.9 Register Number Incrementer And Register

The 4-bit Register Number Register is loaded from the Rb Register at the output of the Decode Stage. The contents of this register may be incremented by one under the control of the Operand Fetch state machine.

2.5.10 Jump Register

The 32-bit Jump Register is loaded directly from the Address Adder. It is used to present VIPCs via the Prefetch Bus for JMP and JSB instructions that are not absolute or PC-relative.

/* Other cases TBS */

2.6 OPERAND FETCH STAGE

This stage performs three major functions:

1. Accesses the data TB and Cache structure for operand data;
2. Presents operands to the Execution Stage based on the operand size and specifier position, and assigns corresponding operand pointers;
3. Presents a dispatch address to the main control store based on the specifier (for specifier completion) or opcode.

2.6.1 Inputs

The following pipeline registers described in the previous section are inputs to the Operand Fetch Stage; associated with these pipeline registers is the signal ADD_VALID, which indicates that all of these registers contain valid data.

1. Virtual Address Register
2. Register Number Register
3. Opcode Register
4. Operand Fetch Function Register
5. Execution Function Register

2.6.2 Outputs

The following pipeline registers are input to the Execution Stage; associated with these registers is the signal FETCH_VALID, which indicates that these registers contain valid data.

1. Memory Address Register
2. Bypass Register
3. Memory Data Register
4. Rotator Control Register
5. Pointer Registers
    1. Source1
    2. Source2
    3. Destination
    4. Operand Write
6. Microword Register (contents supplied by Operand Fetch Stage if based on specifier or opcode; otherwise supplied by Execution Stage).

The signal FETCH_STALL is input to the Address Add Stage to inhibit the loading of the pipeline registers at the input of the Operand Fetch Stage.

2.6.3 Data Manipulation

The output of the 3-input Adder in the Address Add Stage is passed to the Operand Fetch Stage via the Virtual Address Register. The Operand Fetch Stage operates on the contents of this register to:

1. Access the TB to produce a physical address; this address may be used to access the data cache for data or may be passed directly to the Address FIFO; or
2. Access the cache or (on a cache miss) main memory for memory data which is loaded into the Memory Data Register; or
3. Pass the contents of the Virtual Address Register directly to the Bypass Register or Address FIFO.

2.6.4 Bypass Multiplexer

The two-to-one Bypass Multiplexer provides a path for operands around the data TB and Cache structures. Its inputs are the Virtual Address Register and the Physical Address Register. Its output is input to both the Bypass Register and the Address FIFO. Operand Data is passed via this multiplexer from the Virtual Address Register to the Bypass Register, and virtual and physical addresses may be loaded into the Address FIFO via this mux.

2.6.5 Bypass Register

The 32-bit Bypass Register is loaded from the Bypass Multiplexer. It is one of the three possible sources for the Operand Bus, and is used to pass virtual and physical addresses, as well as operand data from the Address Add Stage, to the Execution Unit.

2.6.6 Address Buffers

The Address Buffers are organized as a 32-bit-wide FIFO which may be read and written every cycle. There are 8 buffers which may be read during the first half of the cycle for presentation to the Physical Address Register or the Execution Unit, and which may be loaded during the second half of the cycle from the Bypass Multiplexer. The output may be used to drive the DCache Address Bus when accessing cache, or may be presented to the main ALU for manipulation via the Address Register and Operand Bus. Reads are under the control of the Execution Unit microcode.

2.6.7 Memory Address Register

The Memory Address Register is loaded from the top of the Address FIFO. This Register is used to pass virtual and physical addresses to the Execution unit via the Operand Bus under Execution microcode control.

2.6.8 Data Translation Buffer

The Data Translation Buffer (DTB) consists of 256 entries, 128 for system space and 128 for process space, and is implemented with two Frigate TB/Cache Chips. The organization is fully associative, with least-recently-used replacement. VAR <31:9> are presented to the DTB during the first half of the cycle, yielding the corresponding PTE. During the second half of the cycle, access validity is checked while PFN <20:0> are presented to the Data Cache via the Physical Address

Register.

2.6.9 Physical Address Register

The Physical Address Register (PAR) captures the physical address to be presented to the Data Cache twice every cycle. The input to the PAR is the DCache Address Bus, which may be sourced by DTB|VAR<8:0>, the Write Bus via a deskew latch, or the top of the Address FIFO.

2.6.10 Data Cache

The Data Cache is organized as 64 bits wide (+parity) by 512 locations deep (4 KB), and is implemented utilizing 8 Frigate TB/Cache chips. It is fully associative, with least-recently-used replacement. The Data Cache implements the write-back scheme being used in Firefly at SRC. This includes implementing two additional status bits, the Modified bit and the Shared bit. The Modified bit indicates that a cache line has been written locally, while the shared bit indicates that another processor also has this line cached, such that any writes to this line must be broadcast on the FSB.

To facilitate pipelining, Write-in-Progress bits are associated with each 64-bit cache line. These bits are set as the result of a modify or write specifier access type during the pipeline read cycle. They are cleared when the cache line is written, or may be cleared as a group when flushing the pipeline.

Writes and the clearing of WIP bits occur during the first half of the cycle, and reads and the setting of WIP bits occur during the second half of the cycle.

2.6.11 Memory Data Register

The Memory Data Register is 64-bits wide. Its input is the DCache Data Bus, and its output goes to the Output Rotators, which source the Operand Bus.

2.6.12 Rotator Control Register

The Rotator Control Register is input to the Output Rotators to specify the extraction of a 32-bit quantity from the 64-bit Memory Data Register in the next cycle.

2.6.13 Dispatch Control Logic

The Dispatch Control Logic manipulates the Pointer FIFOs and controls the generation of the specifier and opcode dispatch microwords. Inputs to this logic include the Operand Fetch Function Register, Opcode Register, and Register Number Register. This logic manipulates the three Pointer FIFOs, and generates the addresses to be loaded into them. Additionally, this logic generates the dispatch address to potentially be used to supply the dispatch microword at the end of the cycle, either directly to handle specifier completion, or indirectly via the opcode.

2.6.14 Dispatch FIFOs

The Dispatch FIFOs include a command FIFO, which buffers the Opcode Register (and any other relevant control information), and the Operand Pointer FIFOs, of which there are three. The Operand Pointer FIFOs are loaded under the control of the Dispatch Control Logic with the address of the GPR or Operand Buffer into which the operand data is to be placed, or with an indication that an address has been loaded into the Address FIFO. Each of these pointer FIFO's are eight bits wide (5 address bits and 3 status bits) and 16-deep; they are identified as follows:

1. Source1 Pointer FIFO - used to address the Source1 Operand Buffer, or the Source1 copy of the GPRs. Pointers to source specifiers which appear in positions 1,3 and 5 in the instruction are loaded into the Source1 Pointer FIFO. It is one of two sources for Source1 read addresses, and as such its output goes to the Source1 Pointer Multiplexer.

2. Source2 Pointer FIFO - used to address the Source2 Operand Buffer, or the Source2 copy of the GPRs. Pointers to source specifiers which appear in positions 2, 4 and 6 in the instruction are loaded into the Source2 Pointer FIFO. It is one of two sources for Source2 read addresses, and as such its output goes to the Source2 Pointer Multiplexer.

3. Destination Pointer FIFO - used to specify a read from the Address FIFO, or to supply a register number as a write destination. It specifies the location(s) to be written by the corresponding instruction, and thus may contain either a GPR number or an indication that the write is to memory, such that the address at the top of the Address FIFO should be read. Its output goes to the Destination Pointer Multiplexer.

The Address FIFO is loaded during the Operand Fetch Stage at the same time as the Address FIFO bit is set in the Destination Pointer FIFO. The Operand Buffers, however, are loaded in the next cycle over the Operand Bus from either the Bypass Register or the Memory Data Register (via the rotators) at the address specified in the Operand

Write Pointer Register. There is also a bypass mechanism implemented which allows either of these outputs (including rotated memory data) to be used by the Execution Unit in this cycle.

The Operand Fetch Stage handles specifiers which require more than one cycle to produce the requested specifier data (except for quadword and octaword immediates). Logic in this stage interprets size, access mode, and specifier position in the instruction to supply operands via the Operand Buffers and Pointer FIFOs symmetrically to the Execution Unit. This logic also guarantees that all entries in the Pointer FIFOs and Operand Buffers for a particular specifier go into the same set of FIFOs and Buffers.

2.6.15 Source1 Pointer Multiplexer And Register

The Source1 Pointer Multiplexer specifies the Source1 Read Address used to access the Register File or Temporary Registers in the execution stage. It is 7 bits wide and selects the next address to be loaded into the Source1 Pointer Register from one of two sources:

1. Source1 Pointer FIFO (5 bits) - indirect read of GPR or Source1 Operand Buffer

2. Source1 Field of Microword Bus (7 bits) - explicit read of Register File or Temporary Register under microcode control

2.6.16 Source2 Pointer Multiplexer And Register

The Source2 Pointer Multiplexer specifies the Source2 Read Address used to access the register file in the execution stage. It is 6 bits wide and selects the next address to be loaded into the Source2 Pointer Register from one of two sources:

1. Source2 Pointer FIFO (5 bits) - indirect read of GPR or Source2 Operand Buffer

2. Source2 Field of Microword Bus (6 bits) - explicit read of Register File under microcode control

2.6.17 Destination Pointer Multiplexer And Register

The Destination Pointer Multiplexer is used to specify the address to be written. It is 8 bits wide and selects the next address to be loaded into the Destination Register from one of two sources:

1. Destination Pointer FIFO (5 bits) - indirect write of GPR, or write to memory at address at top of Address FIFO

2. Destination Field of Microword Bus (8 bits) - explicit write of any Write Bus destination under microcode control

2.6.18 Operand Write Pointer Register

This 4-bit-wide register contains the address of the next Operand Buffer to be written; that is, the address of the next available entry in the Operand Buffer FIFO. It is supplied from the Dispatch Control Logic.

2.6.19 Dispatch Microwords

The Dispatch Microword is the first microword to be executed in response to a new instruction, or when execution intervention is required to supply the required operand(s). A bit in the previous microword enables dispatch; this causes the next microaddress to be supplied to the main control store from the dispatch logic. This microaddress is a function of the opcode or a specifier associated with the opcode, determined by whether the operands associated with a particular instruction were supplied by the preceding stages in the pipeline (ie: without the help of the execution unit). Execution unit intervention occurs as a function of the specifier mode (ie: autoincrement mode) or because of an error which occurred in a preceding pipeline stage while attempting to process this specifier.

In some cases, exceptional conditions occur in the Operand Fetch Stage too late to affect the dispatch microaddress. These cases trap the microsequencer and generate the appropriate microword directly via hard-wired logic.

2.6.20 Microword Bus

The Microword Bus supplies the next microword to be loaded into the Microword Register and the Pointer Registers. Most often, the microword comes from the main control store PROM/RAM structure; in dispatch cases, portions of the microword come from the Pointer FIFOs or hard-wired logic.

2.6.21 Microword Register

Defined in Execution Unit Section.

2.6.22 Operand Fetch Stage Sequencer

The Operand Fetch Stage includes an n-bit finite state sequencer to handle multiple cycle accesses, such as octaword reads from cache.

2.7 EXECUTION STAGE

The Execution Stage operates on up to two 32-bit operands per cycle to produce a 32-bit result. It is micro-program controlled. Major functional units include the ALU, main control store, main microsequencer, the Register File (which include Operand Buffers, copies of the GPRs, and working registers), a set of Temporary Registers, and a Register Log. Results are presented on the Write Bus via the Write Latch; all writes to GPRs and memory are under the control of the Execution Unit.

2.7.1 Inputs

The following pipeline registers described in previous sections are inputs to the Execution Stage; associated with these pipeline registers is the signal FETCH_VALID, which indicates that all of these registers contain valid data.

1. Memory Address Register (top of Address Buffer FIFO)

2. Bypass Register

3. Memory Data Register

4. Rotator Control Register

5. Pointer Registers:

    1. Source1

    2. Source2

    3. Destination

    4. Operand Write

6. Microword Register (supplied by Operand Fetch Stage if based on specifier or opcode, otherwise supplied by Execution Stage).

2.7.2 Outputs

The Execution Unit writes results over the 32-bit Write Bus via the Write Bus Latch, which contains valid data from T0 to T3.

2.7.3 Data Rotators

The Data Rotators extract a 32-bit quantity from the 64-bit Memory Data Register every cycle, based on the contents of the Rotator Control Register. The output is one of the sources of Operand Bus data.

2.7.4 Operand Bus

The Operand Bus is the medium over which operand data is passed to the Execution Stage. The Operand Bus is an input to both ALU Input Muxes; and supplies data to be written to the Operand Buffers. The Operand Bus may be driven from the Memory Data Register (via the rotators), the Memory Address Register, or the Bypass Register.

2.7.5 ALU

Two 32-bit source operands are supplied to the ALU every cycle via the Source1 and Source2 Input Multiplexers.

A 32-bit result is input to the Write Latch, and condition codes are captured, at the end of the cycle.

The ALU is controlled directly by the ALU Function Field in the microword. Functions and encoding TBD.

The ALU is implemented using the AMD 29332.

2.7.6 FPU

The FPU structure will provide hardware support for F, D, and G_Floating Point instructions. FPU inputs are the 32-bit Source1 and Source2 Input Muxes; the FPU output will drive the Write Bus via the Write Latch under microcode control.

2.7.7 Source1 Input Mux

The Source1 Input Mux provides one of the 32-bit source operands to the ALU and FPU every cycle. Source1 may come from one of four places:

1. Register File
2. Temporary Registers
3. Write Bus
4. Operand Bus

The selects on the mux are controlled by the microword and the output of bypass logic which monitors addresses associated with the Write and Operand Buses.

2.7.8 Source2 Input Mux

The Source2 Input Mux provides one of the 32-bit source operands to the ALU and FPU every cycle. Source2 may come from one of five places:

1. Register File
2. Register Log
3. State Gate Array
4. Operand Bus
5. Write Bus

The selects on the mux are controlled by two bits in the microword and the output of bypass logic which monitors addresses associated with the Write and Operand Buses.

2.7.9 Reserved Operand Detection

Reserved operand detection will be done in hardware under the control of an enable bit in the microcode. Hardware will monitor the outputs of both Source Input Muxes.

2.7.10 Register File

The Register File is two-port-readable and two-port-writeable, and is implemented using two AMD 29334 Register File chips. It is 32 bits wide and 64 entries deep, and can be read and written on both ports every cycle. Reads occur during the first half of the cycle, and writes occur during the second half of the cycle.

The Register File includes copies of the GPR's, 32 Working Registers, and 16 Operand Buffers. The Operand Buffers are managed as a FIFO structure by the Dispatch Control Logic. The Operand Write Pointer Register contains the address of the next available entry in this Operand Buffer FIFO structure, into which an operand supplied from the Operand Fetch Stage via the Operand Bus may be written every cycle. This address is also maintained in the appropriate Pointer FIFO, so that this location (ie: the top of the FIFO) may be indirectly read by the microcode when the time comes.

The other write port is under the control of the execution microcode. The address is supplied from the Destination Pointer Register, and the data is supplied from the Write Bus via a deskew latch.

The two read ports are inputs to the two Source Input Muxes, and are addressed by the Source1 and Source2 Pointer Registers.

2.7.11 Temporary Registers

The Temporary Register File consists of 64 32-bit entries, addressed by the Source1 Pointer Register. This register file is one of the inputs to the Source1 Input Mux. Write data is supplied from the Write Bus.

2.7.12 Register Log

The Register Log is a LIFO structure which is used to capture register contents for backup. It is 36 bits wide and 7 entries deep.

2.7.13 State Gate Array

The State Gate Array contains miscellaneous logic and state information which may be input to the ALU data path.

2.7.14 Control Store

The Control Store is 8K deep and 96 bits wide. Initialization microcode will be implemented in PROM; the remainder of the control store will be implemented in RAM, and will be loadable under the control of the console subsystem.

Microword encoding TBS.

2.7.15 Microsequencer

The microsequencer controls the generation of microaddresses which are used to select the next microword from the control store. The functions provided are:

1. Jump to address
2. Jump to subroutine
3. Branch
4. CASE
5. Return from subroutine

A 13-bit address is specified for all microinstructions except return. Branch instructions go to either the specified target address or to the current address plus one. Case instructions go to the specified address with hardware conditions logically OR'ed into the low 4 bits.

The microaddress may also come from either the instruction/specifier decode logic, or from the microtrap logic for certain late-occuring error conditions.

2.7.16 Write Bus

The Write Bus is driven by the Write Latch, and is valid from approximately T0+15 to T3. It is driven by the Write Bus Latch. This is the main write path in the CPU, and is distributed to all stages in the machine. Thus, it generally needs to be received by a deskew latch. In particular, the Write Bus goes to:

1. the Prefetch Stage for ITB manipulation and to supply branch addresses
2. the Decode Stage for GPR writes

3. the Address Adder in the Address Add Stage

4. the Operand Fetch Stage to update the TB, and to supply write data to the cache and main memory

2.8 FSB PORT

The Frigate System Bus (FSB) Port is the interface between the 64-bit Internal Bus in the Frigate CPU, and the 64-bit FSB. The port includes input and output buffers, and port control logic.

```
VAX/VMS  SUPNIK     CP  5-DEC-1984 14:21   LPA0:  5-DEC-1984 15:00   UVWDS:[SUPNIK.FRIGATE]CP.RNO;15      VAX/VMS
VAX/VMS  SUPNIK     CP  5-DEC-1984 14:21   LPA0:  5-DEC-1984 15:00   UVWDS:[SUPNIK.FRIGATE]CP.RNO;15      VAX/VMS
VAX/VMS  SUPNIK     CP  5-DEC-1984 14:21   LPA0:  5-DEC-1984 15:00   UVWDS:[SUPNIK.FRIGATE]CP.RNO;15      VAX/VMS

VAX/VMS  SUPNIK     CP  5-DEC-1984 14:21   LPA0:  5-DEC-1984 15:00   UVWDS:[SUPNIK.FRIGATE]CP.RNO;15      VAX/VMS
VAX/VMS  SUPNIK     CP  5-DEC-1984 14:21   LPA0:  5-DEC-1984 15:00   UVWDS:[SUPNIK.FRIGATE]CP.RNO;15      VAX/VMS
VAX/VMS  SUPNIK     CP  5-DEC-1984 14:21   LPA0:  5-DEC-1984 15:00   UVWDS:[SUPNIK.FRIGATE]CP.RNO;15      VAX/VMS

                  SSSS  U   U  PPPP   N   N  III  K   K
                 S      U   U  P   P  N   N   I   K  K
                 S      U   U  P   P  NN  N   I   K K
                  SSS   U   U  PPPP   N N N   I   KKK
                     S  U   U  P      N  NN   I   K K
                     S  U   U  P      N   N   I   K  K
                 SSSS   UUUUU  P      N   N  III  K   K

    CCCCCCCC  PPPPPPPP
    CCCCCCCC  PPPPPPPP
  CC          PP      PP
  CC          PP      PP
  CC          PP      PP
  CC          PP      PP
  CC          PPPPPPPP
  CC          PPPPPPPP
  CC          PP
  CC          PP
  CC          PP            ....
  CC          PP            ....
    CCCCCCCC  PP            ....
    CCCCCCCC  PP            ....

  RRRRRRRR    NN      NN    OOOOOO        ;;;;        11      5555555555
  RRRRRRRR    NN      NN    OOOOOO        ;;;;        11      5555555555
  RR      RR  NN      NN  OO      OO      ;;;;      1111      55
  RR      RR  NN      NN  OO      OO      ;;;;      1111      55
  RR      RR  NNNN    NN  OO      OO                  11      555555
  RR      RR  NNNN    NN  OO      OO                  11      555555
  RRRRRRRR    NN  NN  NN  OO      OO      ;;;;        11            55
  RRRRRRRR    NN  NN  NN  OO      OO      ;;;;        11            55
  RR  RR      NN    NNNN  OO      OO      ;;;;        11              55
  RR  RR      NN    NNNN  OO      OO      ;;;;        11              55
  RR    RR    NN      NN  OO      OO        ;;        11      55      55
  RR    RR    NN      NN  OO      OO        ;;        11      55      55
  RR      RR  NN      NN    OOOOOO        ;;        111111      555555
  RR      RR  NN      NN    OOOOOO        ;;        111111      555555

                  SSSS  U   U  PPPP   N   N  III  K   K
                 S      U   U  P   P  N   N   I   K  K
                 S      U   U  P   P  NN  N   I   K K
                  SSS   U   U  PPPP   N N N   I   KKK
                     S  U   U  P      N  NN   I   K K
                     S  U   U  P      N   N   I   K  K
                 SSSS   UUUUU  P      N   N  III  K   K

VAX/VMS  SUPNIK     CP  5-DEC-1984 14:21   LPA0:  5-DEC-1984 15:00   UVWDS:[SUPNIK.FRIGATE]CP.RNO;15      VAX/VMS
VAX/VMS  SUPNIK     CP  5-DEC-1984 14:21   LPA0:  5-DEC-1984 15:00   UVWDS:[SUPNIK.FRIGATE]CP.RNO;15      VAX/VMS
VAX/VMS  SUPNIK     CP  5-DEC-1984 14:21   LPA0:  5-DEC-1984 15:00   UVWDS:[SUPNIK.FRIGATE]CP.RNO;15      VAX/VMS
```

COST GOAL FOR KERNEL SYSTEM = 20K

1.2 TPI < 4?

CYCLE TIME ≈ 70ns

LOW-END SYSTEM = 725 CABS

HIGH END SYSTEM = NAUTILUS CABS

OPTIMIZED REGISTER-MODE DESTINATIONS

Summary of three alternative cache organizations:

>>> Calculated I-stream and data stream effective access times

The following calculations utilize data obtained from cache simulations to estimate the effective access times for I-stream and data accesses. I-stream accesses are broken into cache hits and misses, and the effective I-stream access time is calculated as the weighted average of these two cases. Since the data cache is write-back, data cache accesses are a weighted average of:

1) read and write cache hits
2) read and write misses that cause cache fills to occur, and
3) data writebacks and writes of unwritten modified cache lines.

Finally, from the actual system memory demand thus calculated, and an estimated instruction execution rate, bus demand is calculated as system memory demand divided by elasped time.

Assumptions:

```
        a) read time                      = 500 nsec
        b) write time (writeback)         = 400 nsec
        c) cache hit read cost            = 100 nsec
        d) cache hit write cost           = 100 nsec
        e) instruction execution time     = 400 nsec
```

The following hit rate and writeback data taken from EPASMM.LOG, a complete trace of the EPASCAL compiler compiling a simple program. Of all of the benchmark programs input to the cache simulator program, this seems to be most representative of a large system programs, and proved to be the most taxing in terms cache performance.

Overall statistics:

```
        total number of instructions               455,914
        total number of reads                      540,148
                I-stream reads             309,439
                data reads                 230,709

        total number of writes                     116,843
```

Analysis:

Cost of I-stream reads =

```
  (I-stream hit rate * 100 nsec) +
  (I-stream miss rate * 500 nsec)  =  Average I-stream Read Time
```

Cost of data reads and writes =

```
  { ((read data hits + write data hits) * 100 nsec) +

    ((read data misses + write cache fills) * 500 nsec) +
    ((data writebacks + unwritten modified cache lines) * 400 nsec) }

   / (total number of data reads + total number of data writes)  =  Average Data Cache Access Time


  Bus Demand =
  ------------

  { (read data misses + write cache fills) * fill size +
        (data writebacks + unwritten, modified cache lines) * cache line size }
        divided by (elapsed time)

      = bus demand in bytes / second


Case 1 -- 4k byte, fully associative, LRU cache
-----------------------------------------------

        I-stream hit rate         = 96.5%
        data hit rate             = 95.9%

        data writebacks           = 7351
        write cache fills         = 4037
        unwritten modify lines    = 4073

for I-stream: (cache model assumes straight 8-byte I buffer)

  0.965 * 100 nsec + (1 - 0.965) * 500 nsec

   = 114 nsec effective I-stream read access time
     -------------------------------------------

for data reads and writes:

Average Data Cache Access Time =

 { (231k * 0.959 + 117k - (7351 + 4073)) * 100 +  (231k * 0.041 + 4037) * 500 + (7351 + 4073) * 400 }
----------------------------------------------------------------------------------------------------------
                                              (231k + 117k)

  = 127 nsec Average Data Cache Access Time
    ---------------------------------------------

Bus Demand = { (231k * 0.041 + 4037) * 16 + (7351 + 4073) * 8 } /
                (455,914 instr. * 400 nsec / instr.)

         = 1.69 Mb / sec
           -------------

CASE 2 -- 32 kbyte direct map cache
------------------------------------

        I-stream hit rate         = 97.0%
        read data hit rate        = 96.0%


        data writebacks           = 2710
        write cache fills         = 3988
        unwritten modify lines    = 8616
```

```
for I-stream: (cache model assumes straight 8-byte I buffer)

        0.97 * 100 nsec + (1 - 0.97) * 500 nsec

        = 112 nsec effective I-stream read time
          ------------------------------------

for data reads and writes:

Average Data Cache Access Time =

  { (231k * 0.960 + 117k - (2710 + 8616)) * 100 + ((231k * 0.040) + 3988) * 500 + (2710 + 8616) * 400 }
--------------------------------------------------------------------------------------------------------
                                             (231k + 117k)

  = 126 nsec Average Data Cache Access Time
    ---------------------------------------

Bus Demand = { (231k * 0.040 + 3988) * 16 + (2710 + 8616) * 8 } /
                (455,914 instr. * 400 nsec / instr.)

          = 1.66 Mb / sec
            -------------


CASE 3 -- 16 kbyte 2-way Set-Associative cache
----------------------------------------------

        I-stream hit rate        = 97.1%
        read data hit rate       = 96.2%


        data writebacks          = 2728
        write cache fills        = 3790
        unwritten modify lines   = 8130

for I-stream: (cache model assumes straight 8-byte I buffer)

        0.971 * 100 nsec + (1 - 0.971) * 500 nsec

        = 111.6 nsec Average I-stream Read Access Time
          --------------------------------------------

for data reads and writes:

Average Data Cache Access Time =

  { (231k * 0.962 + 117k - (2728 + 8130)) * 100 + (231k * 0.038 + 3790) * 500 + (2728 + 8130) * 400 }
--------------------------------------------------------------------------------------------------------
                                             (231k + 117k)

  = 125 nsec Average Data Cache Access Time
    ---------------------------------------

Bus Demand = { (231k * 0.038 + 3790) * 16 + (2728 + 8130) * 8 } /
                (455,914 instr. * 400 nsec / instr.)

          = 1.58 Mb / sec
            -------------
```

Cache Performance Summary

Cache	I-stream Read	Data Read/Write	Bus Demand	Packages	Board Space	Power
4kb FALRU	114.0 nsec	127 nsec	1.69 Mb / sec	32	30.9 sq.in.	20.1 watts
32kb direct map	112.0	126	1.66	122	55.8	41.5
16kb 2-way SA	111.6	125	1.58	224	98.4	70.4

{ end of cache performance summary -- 11/28/84, mist::[butts.deceast]cp.rno }

Custom 4k byte Fully Associative, LRU Data TB/cache
----

1) Data Cache TB

a) Description

1) 128 fully associative, least recently used (FALRU) system entries,
   128 FALRU process entries
2) single cycle clear of either system or process entries
3) full internal byte parity

b) Parts Count, Board Space, and Power Estimates

Virtual Address Register	74F374	4	20 pin	2.0 sq.in.	1.38 watts
TB FALRU chips	xxxxxx	2	100 pin LCC	4.0	2.00
Physical Address Register	74F374	4	20	2.0	1.38
Hit Buffer	74F244	1	20	0.5	0.37
		11	packages	8.5 sq.in.	5.13 watts

c) Timing

	0.0 nsec	VA available at input to virtual address register
	8.0	VA available at output of virtual address register
	8.0	VA available at TB FALRU chips
**	33.0	Data available at output of FALRU chips

	0.0 nsec	VA available at input to virtual address register
	8.0	VA available at output of virtual address register
	8.0	VA available at TB FALRU chips
	33.0	Data available at output of FALRU chips
**	40.0	Hit indication available at output of hit buffer

	0.0 nsec	VA available at input to virtual address register
	8.0	VA available at output of virtual address register
	8.0	VA available at TB FALRU chips
	33.0	Tag, data, and hit indication available at output of FALRU chips
**	41.0	PTE available at output of physical address register
**	tbs	parity indication available

2) Data Cache

a) Description

1) 4k bytes fully associative, least recently used
2) 64 bit cache lines
3) fill size is 128 bits
4) writeback
5) control bits = valid, modified, shared, and WIP
6) direct clear on WIP bits, all others must be cleared via explicit writes

b) Parts Count, Board Space, and Power Estimates

Physical Address Buffer	74F244	4	20	2.0	1.47
FALRU Chip selector	74F139	1	16	0.4	0.08
FALRU data cache chips	xxxxx	8	100 pin LCC	16.0	12.00
Memory Data Register	74F374	8	20	4.0	1.38

```
                                                              ---------       ---------      ---------
                                                              21 packages     22.4 sq.in.    14.93 watts
  c) Timing

         41.0           physical address (PA) available at output of PA register
         48.0           PA available at output of PA buffers
         52.0           PA at data cache address bus
         55.0           PA at data cache FALRU chips
**       80.0           Data and hit indication available at output of data cache FALRU chips

         41.0           physical address (PA) available at output of PA register
         48.0           PA available at output of PA buffers
         52.0           PA at data cache address bus
         55.0           PA at data cache FALRU chips
         80.0           Data and hit indication available at output of data cache FALRU chips
**       87.0           Hit indication available at output of hit buffer

         41.0           physical address (PA) available at output of PA register
         48.0           PA available at output of PA buffers
         52.0           PA at data cache address bus
         55.0           PA at data cache FALRU chips
         80.0           Tag, data, and hit indication available at output of data cache FALRU chips
**       88.0           data available at output of memory data latch

{       end of custom 4k FALRU cache,   11/29/84        MIST::[butts.cache]falrucc.rno  }
```

Frigate TB/cache Chip -- Description of Operation

The fully associative, least-recently-used (FALRU) Frigate TB/cache chip is intended to be a fast, general purpose, integrated cache building block for use in the Frigate processor. The design incorporates features that allow it to be used in the instruction and data cache translation buffers, and in the instruction and data caches. The 128 entry by 32 bit design incorporates single cycle direct clear, support for locking of cache locations (as required for use in a pipelined processor), and logic for the support of a writeback cache scheme. The replacement algorithm is true least recently used (LRU).

This particular design has evolved from a discussion of cache organizations found in Peter Kogge's book "The Architecture of Pipelined Computers," pages 260-262. Kogge suggests that a FIFO-like scheme, altered to recirculate the most recently used cache entry to the top of the FIFO, can be used to implement the least-recently-used replacement algorithm in hardware. New entries are written to the top of the FIFO, with all other entries being forced down one location. If the FIFO is full, the oldest entry is forced out the bottom and lost. This design emulates the FIFO with a shift register-like organization, in which each shift register location holds a cache entry's address tag, valid and other status bits, and data.

Each entry has a comparator, which compares the presented address with the stored address tag. Comparison of the presented address with the address tag of each cache entry proceeds simultaneously. When a hit occurs, the data of the matching entry is enabled onto the common data bus, and transmitted by output drivers to the external data pins. The matching data is also presented via the common data bus as input to the top location of the entry shift register. The hit indication is used to enable the matching entry and all entries above the matching entry to shift when the clock arrives, thus shifting the matching entry into the top location and all successive entries down to the matching entry, down by one shift register location.

Since the address tag can be re-created from the matching address, and the valid bit can be re-marked as valid, there is no need to include additional bus structures to recirculate the matching entry's tag and valid bit to the top of the shift register. Defining the valid bit of the top entry from an external pin also allows an entry to be marked invalid, thereby giving a mechanism for invalidating single TB entries. While this has the undesirable feature of causing a 'dead' entry to exist within the TB until it finally is shifted out the bottom, it costs no additional logic, and with a reasonably sized TB, results in a negligible performance loss.

Writes are done by asserting the write data on the common data bus, the corresponding address on the FALRU cache chip address pins, and asserting the valid bit on the valid in pin. Write enable is ORed with the hit indication from the bottom ('oldest') entry, causing all the entries of the shift register to shift with the rising clock edge. The new data is written into the top entry, all intermediate entries are shifted down one entry, and, if the chip is full, the oldest entry is lost.

The addition of a second single cycle direct clear status bit, the write-in-progress (WIP) bit, along with the means to read and write this bit upon a cache hit, gives the FALRU chip a mechanism for marking a cache location as having a pending write. This hardware simplifies the management of a pipelined processor's accesses to cache, and allows stalling the processor only in cases where a write to cache is pending and a successive instruction in the pipeline attempts to access the same memory location. The ability to independently clear all WIP bits in a single cycle prevents this feature from becoming a performance bottleneck during pipeline flushes.

Two additional status bits, along with a final register to buffer entries being lost out the bottom of the shift register, allow the operation of the FALRU cache chip in a writeback cache mode. The first status bit is used to mark a modified cache entry as needing eventual writeback to system memory. The second bit marks a cache entry as being shared between two or more caches in the system, and as needing to have any modifications of this location broadcast to the entire system. These two status bits are treated as two additional data bits, and must be initialized by the processor before cache operation is begun. An OR-AND structure on the input of each of these bits allows either bit to be reset upon a cache entry write, simply recirculated with its current value, or set upon a hit. Only the shared bit is available externally, as the state of the modified bit is reflected by the state of the writeback flag.

When an entry is shifted out of the shift register into the lost entry buffer, its valid and modify bits are ANDed together to create an external writeback flag. If the entry is both valid and modified, the writeback flag is asserted, and the lost entry address tag and data must be read from the FALRU cache chip before any successive cache chip writes are allowed, else the modified data will be lost. Two additional external signals exist to allow the reading of the address tag and data from the lost entry buffer.

To provide a means of monitoring the integrity of the operating FALRU cache chip, parity is generated in roughly eight bit groups of the address tag, status bits, and data. This parity is checked upon a hit, or when data is read from the lost entry buffer. A discrepancy in parity is reported externally by a parity error flag signal. A means will be provided to allow the testing of the parity generation and checking logic.

Testing of the FALRU cache chip will involve several phases. Phase I will operate the FALRU cache chip as a shift register only, presenting test patterns designed to test the integrity of each bit of memory in the address tag, status bits, and data portions of each entry. Phase II will attempt to determine that each entry's comparator is fully functional, and can produce a hit indication and cause the correct data and status to be produced at the data I/O pins. Phase III will confirm the feedback of matching entries to the top of the shift register. A final test will be of a statistical nature. It will put the chip through a sequence of writes and reads, and then read the entire state of the chip and compare it against the correct state. By operating the FALRU cache chip in a non-standard mode, the number of test vectors can be greatly reduced. For example, there is no need to have the chip do the shift re-organization during the comparator test phase.

FALRU cache chips are cascadable in both width and depth to allow the construction of larger caches. To increase the width of a cache entry, chips may be accessed in parallel by addressing several chips with the same address, and routing a slice of the wider cache word to each individual chip. To increase the depth of a cache, the least significant address bits can be used to select which chip or chips routing perform the given operation. This assumes an even distribution of accesses are made to each chip; clearly cases exist where the distribution of data accesses is skewed, such as regularly accessing a data array. The impact of regular accessing upon a multiple FALRU cache chip based system can be estimated by examining the cache performance simulation for a cache of the corresponding smaller size. Thus, the 4k byte Frigate data cache would operate worstcase much like a 1k byte cache, and cache performance simulation data predicts that the data cache hit rate will fall from 96% to 92% under such conditions.

{ MIST::[butts.frigate]falrudes.rno Nov. 29, 1984 hbb }

```
          0000         .title  opcode
          0000
          0000   ;
          0000   ; local symbols
          0000   ;
          0000
00000001  0000   brch = 1                              ;stop decode and branch after execution
00000002  0000   cont = 2                              ;stop decode and continue after execution
00000004  0000   cond = 4                              ;conditional branch
00000008  0000   uncd = 8                              ;stop pipe and unconditionlly branch
00000010  0000   push = 16                             ;implied push for unconditional branch
00000020  0000   ftch = 32                             ;stop operand fetch and continue after execu
00000040  0000   pop = 64                              ;implied pop for return from subroutine
00000080  0000   loop = 128                            ;looping instruction - predicted taken
00000100  0000   dest = 256                            ;register destintion optimization allowed
          0000
00000000  0000   b = 0                                 ;byte
00000001  0000   w = 1                                 ;word
00000002  0000   l = 2                                 ;longword
00000003  0000   q = 3                                 ;quadword
00000004  0000   o = 4                                 ;octaword
          0000
00000000  0000   rd = 0                                ;read access
00000001  0000   wr = 1                                ;write access
00000002  0000   md = 2                                ;modify access (read and write)
00000003  0000   ad = 3                                ;address access
00000004  0000   vd = 4                                ;vield access
00000005  0000   bd = 5                                ;branch displacement
          0000
          0000   ;
          0000   ; This program defines the opcode data base.
          0000   ;
          0000
          0000         .macro  define  name, cycles, access, datatype, flags
          0000   n = 0
          0000         .irpc   x,<name>
          0000   n = n + 1
          0000         .endr
          0000         .ascii  /name/
          0000         .rept   6 - n
          0000         .ascii  / /
          0000         .endr
          0000         .byte   cycles
          0000   n = 0
          0000         .irp    x,<access>
          0000   n = n + 1
          0000         .endr
          0000         .byte   n
          0000         .irp    x,<access>
          0000         .byte   x
          0000         .endr
          0000         .blkb   6 - n
          0000         .irp    x,<datatype>
          0000         .byte   x
          0000         .endr
          0000         .blkb   6 - n
          0000   n = 0
```

```
    0000                 .irp     x,<flags>
    0000        n = n ! x
    0000                 .endr
    0000                 .word    n
    0000                 .endm    define
    0000
    0000        ;
    0000        ; Define opcode data base
    0000        ;
    0000
00000000                 .psect   $data,rd,wrt,pic,long,noshr,noexe,lcl,con,rel
    0000        opcode::
    0000                 define   HALT,10,<>,<>,<brch>
    0016                 define   NOP,0,<>,<>
    002C                 define   REI,10,<>,<>,<brch>
    0042                 define   BPT,10,<>,<>,<brch>
    0058                 define   RET,30,<>,<>,<brch>
    006E                 define   RSB,1,<>,<>,<brch,pop>
    0084                 define   LDPCTX,50,<>,<>,<cont>
    009A                 define   SVPCTX,50,<>,<>,<cont>
    00B0                 define   CVTPS,100,<rd,ad,rd,ad>,<w,b,w,b>,<cont>
    00C6                 define   CVTSP,100,<rd,ad,rd,ad>,<w,b,w,b>,<cont>
    00DC                 define   INDEX,8,<rd,rd,rd,rd,rd,wr>,<l,l,l,l,l,l>,<dest>
    00F2                 define   CRC,100,<ad,rd,rd,ad>,<b,l,w,b>,<cont>
    0108                 define   PROBER,10,<rd,rd,ad>,<b,w,b>,<dest>
    011E                 define   PROBEW,10,<rd,rd,ad>,<b,w,b>,<dest>
    0134                 define   INSQUE,15,<ad,ad>,<b,b>,<ftch>
    014A                 define   REMQUE,15,<ad,wr>,<b,l>,<ftch>
    0160                 define   BSBB,1,<bd>,<b>,<uncd,push>
    0176                 define   BRB,1,<bd>,<b>,<uncd>
    018C                 define   BNEQ,1,<bd>,<b>,<cond>
    01A2                 define   BEQL,1,<bd>,<b>,<cond>
    01B8                 define   BGTR,1,<bd>,<b>,<cond>
    01CE                 define   BLEQ,1,<bd>,<b>,<cond>
    01E4                 define   JSB,1,<ad>,<b>,<uncd,push>
    01FA                 define   JMP,1,<ad>,<b>,<uncd>
    0210                 define   BGEQ,1,<bd>,<b>,<cond>
    0226                 define   BLSS,1,<bd>,<b>,<cond>
    023C                 define   BGTRU,1,<bd>,<b>,<cond>
    0252                 define   BLEQU,1,<bd>,<b>,<cond>
    0268                 define   BVC,1,<bd>,<b>,<cond>
    027E                 define   BVS,1,<bd>,<b>,<cond>
    0294                 define   BGEQU,1,<bd>,<b>,<cond>
    02AA                 define   BLSSU,1,<bd>,<b>,<cond>
    02C0                 define   ADDP4,100,<rd,ad,rd,ad>,<w,b,w,b>,<cont>
    02D6                 define   ADDP6,100,<rd,ad,rd,ad,rd,ad>,<w,b,w,b,w,b>,<cont>
    02EC                 define   SUBP4,100,<rd,ad,rd,ad>,<w,b,w,b>,<cont>
    0302                 define   SUBP6,100,<rd,ad,rd,ad,rd,ad>,<w,b,w,b,w,b>,<cont>
    0318                 define   CVTPT,100,<rd,ad,ad,rd,ad>,<w,b,b,w,b>,<cont>
    032E                 define   MULP,100,<rd,ad,rd,ad,rd ad>,<w,b,w,b,w,b>,<cont>
    0344                 define   CVTTP,100,<rd,ad,ad,rd,ad>,<w,b,b,w,b>,<cont>
    035A                 define   DIVP,100,<rd,ad,rd,ad,rd,ad>,<w,b,w,b,w,b>,<cont>
    0370                 define   MOVC3,100,<rd,ad,ad>,<w,b,b>,<cont>
    0386                 define   CMPC3,100,<rd,ad,ad>,<w,b,b>,<cont>
    039C                 define   SCANC,100,<rd,ad,ad,rd>,<w,b,b,b>,<cont>
    03B2                 define   SPANC,100,<rd,ad,ad,rd>,<w,b,b,b>,<cont>
    03C8                 define   MOVC5,100,<rd,ad,rd,rd,ad>,<w,b,b,w,b>,<cont>
```

```
03DE    define  CMPC5,100,<rd,ad,rd,rd,ad>,<w,b,b,w,b>,<cont>
03F4    define  MOVTC,100,<rd,ad,rd,ad,rd,ad>,<w,b,b,b,w,b>,<cont>
040A    define  MOVTUC,100,<rd,ad,rd,ad,rd,ad>,<w,b,b,b,w,b>,<cont>
0420    define  BSBW,1,<bd>,<w>,<uncd,push>
0436    define  BRW,1,<bd>,<w>,<uncd>
044C    define  CVTWL,1,<rd,wr>,<w,l>,<dest>
0462    define  CVTWB,1,<rd,wr>,<w,b>,<dest>
0478    define  MOVP,100,<rd,ad,ad>,<w,b,b>,<cont>
048E    define  CMPP3,100,<rd,ad,ad>,<w,b,b>,<cont>
04A4    define  CVTPL,100,<rd,ad,wr>,<w,b,l>,<cont>
04BA    define  CMPP4,100,<rd,ad,rd,ad>,<w,b,w,b>,<cont>
04D0    define  EDITPC,100,<rd,ad,ad,ad>,<w,b,b,b>,<cont>
04E6    define  MATCHC,100,<rd,ad,rd,ad>,<w,b,w,b>,<cont>
04FC    define  LOCC,100,<rd,rd,ad>,<b,w,b>,<cont>
0512    define  SKPC,100,<rd,rd,ad>,<b,w,b>,<cont>
0528    define  MOVZWL,1,<rd,wr>,<w,l>,<dest>
053E    define  ACBW,3,<rd,rd,md,bd>,<w,w,w,w>,<loop>
0554    define  MOVAW,1,<ad,wr>,<w,l>,<dest>
056A    define  PUSHAW,1,<ad>,<w>,<push>
0580    define  ADDF2,2,<rd,md>,<l,l>,<dest>
0596    define  ADDF3,2,<rd,rd,wr>,<l,l,l>,<dest>
05AC    define  SUBF2,2,<rd,md>,<l,l>,<dest>
05C2    define  SUBF3,2,<rd,rd,wr>,<l,l,l>,<dest>
05D8    define  MULF2,3,<rd,md>,<l,l>,<dest>
05EE    define  MULF3,3,<rd,rd,wr>,<l,l,l>,<dest>
0604    define  DIVF2,13,<rd,md>,<l,l>,<dest>
061A    define  DIVF3,13,<rd,rd,wr>,<l,l,l>,<dest>
0630    define  CVTFB,2,<rd,wr>,<l,b>,<dest>
0646    define  CVTFW,2,<rd,wr>,<l,w>,<dest>
065C    define  CVTFL,2,<rd,wr>,<l,l>,<dest>
0672    define  CVTRFL,2,<rd,wr>,<l,l>,<dest>
0688    define  CVTBF,2,<rd,wr>,<b,l>,<dest>
069E    define  CVTWF,2,<rd,wr>,<w,l>,<dest>
06B4    define  CVTLF,2,<rd,wr>,<l,l>,<dest>
06CA    define  ACBF,5,<rd,rd,md,bd>,<l,l,l,w>,<loop>
06E0    define  MOVF,1,<rd,wr>,<l,l>,<dest>
06F6    define  CMPF,2,<rd,rd>,<l,l>,<dest>
070C    define  MNEGF,1,<rd,wr>,<l,l>,<dest>
0722    define  TSTF,1,<rd>,<l>
0738    define  EMODF,6,<rd,rd,rd,wr,wr>,<l,b,l,l,l>,<cont>
074E    define  POLYF,100,<rd,rd,ad>,<l,w,b>,<cont>
0764    define  CVTFD,2,<rd,wr>,<l,q>,<dest>
077A    define  RESRV,0,<>,<>,<brch>
0790    define  ADAWI,5,<rd,vd>,<w,w>,<ftch>
07A6    .rept   3
07A6    define  RESRV,0,<>,<>,<brch>
07A6    .endr
07E8    define  INSQHI,20,<ad,ad>,<b,b>,<ftch>
07FE    define  INSQTI,20,<ad,ad>,<b,b>,<ftch>
0814    define  REMQHI,20,<ad,wr>,<b,l>,<ftch>
082A    define  REMQTI,20,<ad,wr>,<b,l>,<ftch>
0840    define  ADDD2,4,<rd,md>,<q,q>,<dest>
0856    define  ADDD3,4,<rd,rd,wr>,<q,q,q>,<dest>
086C    define  SUBD2,4,<rd,md>,<q,q>,<dest>
0882    define  SUBD3,4,<rd,rd,wr>,<q,q,q>,<dest>
0898    define  MULD2,7,<rd,md>,<q,q>,<dest>
08AE    define  MULD3,7,<rd,rd,wr>,<q,q,q>,<dest>
```

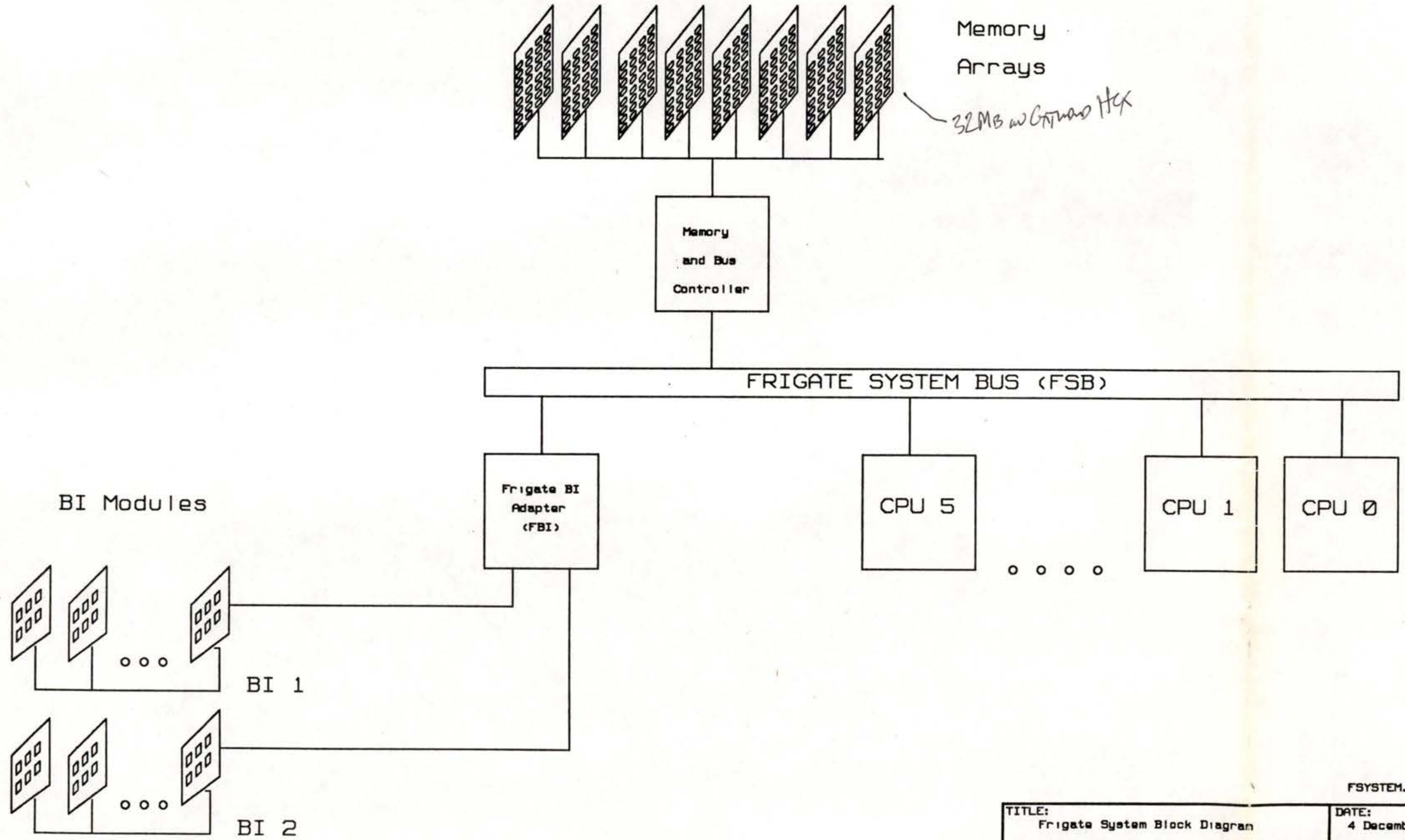
```
08C4          define  DIVD2,31,<rd,md>,<q,q>,<dest>
08DA          define  DIVD3,31,<rd,rd,wr>,<q,q,q>,<dest>
08F0          define  CVTDB,3,<rd,wr>,<q,b>,<dest>
0906          define  CVTDW,3,<rd,wr>,<q,w>,<dest>
091C          define  CVTDL,3,<rd,wr>,<q,l>,<dest>
0932          define  CVTRDL,3,<rd,wr>,<q,l>,<dest>
0948          define  CVTBD,3,<rd,wr>,<b,q>,<dest>
095E          define  CVTWD,3,<rd,wr>,<w,q>,<dest>
0974          define  CVTLD,3,<rd,wr>,<l,q>,<dest>
098A          define  ACBD,5,<rd,rd,md,bd>,<q,q,q,w>,<loop>
09A0          define  MOVD,3,<rd,wr>,<q,q>,<dest>
09B6          define  CMPD,3,<rd,rd>,<q,q>,<dest>
09CC          define  MNEGD,3,<rd,wr>,<q,q>,<dest>
09E2          define  TSTD,2,<rd>,<q>
09F8          define  EMODD,10,<rd,rd,rd,wr,wr>,<q,w,q,l,q>,<cont>
0A0E          define  POLYD,100,<rd,rd,ad>,<q,w,b>,<cont>
0A24          define  CVTDF,2,<rd,wr>,<q,l>
0A3A          define  RESRV,0,<>,<>,<brch>
0A50          define  ASHL,2,<rd,rd,wr>,<b,l,l>,<dest>
0A66          define  ASHQ,4,<rd,rd,wr>,<b,q,q>,<dest>
0A7C          define  EMUL,4,<rd,rd,rd,wr>,<l,l,l,q>,<dest>
0A92          define  EDIV,18,<rd,rd,wr,wr>,<l,q,l,l>,<cont>
0AA8          define  CLRQ,2,<wr>,<q>
0ABE          define  MOVQ,3,<rd,wr>,<q,q>,<dest>
0AD4          define  MOVAQ,1,<ad,wr>,<q,l>,<dest>
0AEA          define  PUSHAQ,1,<ad>,<q>,<push>
0B00          define  ADDB2,1,<rd,md>,<b,b>,<dest>
0B16          define  ADDB3,1,<rd,rd,wr>,<b,b,b>,<dest>
0B2C          define  SUBB2,1,<rd,md>,<b,b>,<dest>
0B42          define  SUBB3,1,<rd,rd,wr>,<b,b,b>,<dest>
0B58          define  MULB2,2,<rd,md>,<b,b>,<dest>
0B6E          define  MULB3,2,<rd,rd,wr>,<b,b,b>,<dest>
0B84          define  DIVB2,17,<rd,md>,<b,b>,<dest>
0B9A          define  DIVB3,17,<rd,rd,wr>,<b,b,b>,<dest>
0BB0          define  BISB2,1,<rd,md>,<b,b>,<dest>
0BC6          define  BISB3,1,<rd,rd,wr>,<b,b,b>,<dest>
0BDC          define  BICB2,1,<rd,md>,<b,b>,<dest>
0BF2          define  BICB3,1,<rd,rd,wr>,<b,b,b>,<dest>
0C08          define  XORB2,1,<rd,md>,<b,b>,<dest>
0C1E          define  XORB3,1,<rd,rd,wr>,<b,b,b>,<dest>
0C34          define  MNEGB,1,<rd,wr>,<b,b>,<dest>
0C4A          define  CASEB,6,<rd,rd,rd>,<b,b,b>,<brch>
0C60          define  MOVB,1,<rd,wr>,<b,b>,<dest>
0C76          define  CMPB,1,<rd,rd>,<b,b>,<dest>
0C8C          define  MCOMB,1,<rd,wr>,<b,b>,<dest>
0CA2          define  BITB,1,<rd,rd>,<b,b>,<dest>
0CB8          define  CLRB,1,<wr>,<b>
0CCE          define  TSTB,1,<rd>,<b>
0CE4          define  INCB,1,<md>,<b>
0CFA          define  DECB,1,<md>,<b>
0D10          define  CVTBL,1,<rd,wr>,<b,l>,<dest>
0D26          define  CVTBW,1,<rd,wr>,<b,w>,<dest>
0D3C          define  MOVZBL,1,<rd,wr>,<b,l>,<dest>
0D52          define  MOVZBW,1,<rd,wr>,<b,w>,<dest>
0D68          define  ROTL,1,<rd,rd,wr>,<b,l,l>,<dest>
0D7E          define  ACBB,3,<rd,rd,md,bd>,<b,b,b,w>,<loop>
0D94          define  MOVAB,1,<ad,wr>,<b,l>,<dest>
```

```
0DAA   229        define  PUSHAB,1,<ad>,<b>,<push>
0DC0   230        define  ADDW2,1,<rd,md>,<w,w>,<dest>
0DD6   231        define  ADDW3,1,<rd,rd,wr>,<w,w,w>,<dest>
0DEC   232        define  SUBW2,1,<rd,md>,<w,w>,<dest>
0E02   233        define  SUBW3,1,<rd,rd,wr>,<w,w,w>,<dest>
0E18   234        define  MULW2,2,<rd,md>,<w,w>,<dest>
0E2E   235        define  MULW3,2,<rd,rd,wr>,<w,w,w>,<dest>
0E44   236        define  DIVW2,17,<rd,md>,<w,w>,<dest>
0E5A   237        define  DIVW3,17,<rd,rd,wr>,<w,w,w>,<dest>
0E70   238        define  BISW2,1,<rd,md>,<w,w>,<dest>
0E86   239        define  BISW3,1,<rd,rd,wr>,<w,w,w>,<dest>
0E9C   240        define  BICW2,1,<rd,md>,<w,w>,<dest>
0EB2   241        define  BICW3,1,<rd,rd,wr>,<w,w,w>,<dest>
0EC8   242        define  XORW2,1,<rd,md>,<w,w>,<dest>
0EDE   243        define  XORW3,1,<rd,rd,wr>,<w,w,w>,<dest>
0EF4   244        define  MNEGW,1,<rd,wr>,<w,w>,<dest>
0F0A   245        define  CASEW,6,<rd,rd,rd>,<w,w,w>,<brch>
0F20   246        define  MOVW,1,<rd,wr>,<w,w>,<dest>
0F36   247        define  CMPW,1,<rd,rd>,<w,w>,<dest>
0F4C   248        define  MCOMW,1,<rd,wr>,<w,w>,<dest>
0F62   249        define  BITW,1,<rd,rd>,<w,w>,<dest>
0F78   250        define  CLRW,1,<wr>,<w>
0F8E   251        define  TSTW,1,<rd>,<w>
0FA4   252        define  INCW,1,<md>,<w>
0FBA   253        define  DECW,1,<md>,<w>
0FD0   254        define  BISPSW,2,<rd>,<w>
0FE6   255        define  BICPSW,2,<rd>,<w>
0FFC   256        define  POPR,20,<rd>,<w>,<cont>
1012   257        define  PUSHR,20,<rd>,<w>,<cont>
1028   258        define  CHMK,10,<rd>,<w>,<brch>
103E   259        define  CHME,10,<rd>,<w>,<brch>
1054   260        define  CHMS,10,<rd>,<w>,<brch>
106A   261        define  CHMU,10,<rd>,<w>,<brch>
1080   262        define  ADDL2,1,<rd,md>,<l,l>,<dest>
1096   263        define  ADDL3,1,<rd,rd,wr>,<l,l,l>,<dest>
10AC   264        define  SUBL2,1,<rd,md>,<l,l>,<dest>
10C2   265        define  SUBL3,1,<rd,rd,wr>,<l,l,l>,<dest>
10D8   266        define  MULL2,2,<rd,md>,<l,l>,<dest>
10EE   267        define  MULL3,2,<rd,rd,wr>,<l,l,l>,<dest>
1104   268        define  DIVL2,17,<rd,md>,<l,l>,<dest>
111A   269        define  DIVL3,17,<rd,rd,wr>,<l,l,l>,<dest>
1130   270        define  BISL2,1,<rd,md>,<l,l>,<dest>
1146   271        define  BISL3,1,<rd,rd,wr>,<l,l,l>,<dest>
115C   272        define  BICL2,1,<rd,md>,<l,l>,<dest>
1172   273        define  BICL3,1,<rd,rd,wr>,<l,l,l>,<dest>
1188   274        define  XORL2,1,<rd,md>,<l,l>,<dest>
119E   275        define  XORL3,1,<rd,rd,wr>,<l,l,l>,<dest>
11B4   276        define  MNEGL,1,<rd,wr>,<l,l>,<dest>
11CA   277        define  CASEL,6,<rd,rd,rd>,<l,l,l>,<brch>
11E0   278        define  MOVL,1,<rd,wr>,<l,l>,<dest>
11F6   279        define  CMPL,1,<rd,rd>,<l,l>,<dest>
120C   280        define  MCOML,1,<rd,wr>,<l,l>,<dest>
1222   281        define  BITL,1,<rd,rd>,<l,l>,<dest>
1238   282        define  CLRL,1,<wr>,<l>
124E   283        define  TSTL,1,<rd>,<l>
1264   284        define  INCL,1,<md>,<l>
127A   285        define  DECL,1,<md>,<l>
```

```
1290    define  ADWC,1,<rd,md>,<l,l>,<dest>
12A6    define  SBWC,1,<rd,md>,<l,l>,<dest>
12BC    define  MTPR,10,<rd,rd>,<l,l>,<cont>
12D2    define  MFPR,10,<rd,wr>,<l,l>,<cont>
12E8    define  MOVPSL,10,<wr>,<l>
12FE    define  PUSHL,1,<rd>,<l>,<push>
1314    define  MOVAL,1,<ad,wr>,<l,l>,<dest>
132A    define  PUSHAL,1,<ad>,<l>,<push>
1340    define  BBS,7,<rd,vd,bd>,<l,b,b>,<cond,ftch>
1356    define  BBC,7,<rd,vd,bd>,<l,b,b>,<cond,ftch>
136C    define  BBSS,7,<rd,vd,bd>,<l,b,b>,<cond,ftch>
1382    define  BBCS,7,<rd,vd,bd>,<l,b,b>,<cond,ftch>
1398    define  BBSC,7,<rd,vd,bd>,<l,b,b>,<cond,ftch>
13AE    define  BBCC,7,<rd,vd,bd>,<l,b,b>,<cond,ftch>
13C4    define  BBSSI,12,<rd,vd,bd>,<l,b,b>,<cond,ftch>
13DA    define  BBCCI,12,<rd,vd,bd>,<l,b,b>,<cond,ftch>
13F0    define  BLBS,3,<rd,bd>,<l,b>,<cond>
1406    define  BLBC,3,<rd,bd>,<l,b>,<cond>
141C    define  FFS,7,<rd,rd,vd,wr>,<l,b,b,l>,<ftch>
1432    define  FFC,7,<rd,rd,vd,wr>,<l,b,b,l>,<ftch>
1448    define  CMPV,7,<rd,rd,vd,rd>,<l,b,b,l>,<ftch>
145E    define  CMPZV,7,<rd,rd,vd,rd>,<l,b,b,l>,<ftch>
1474    define  EXTV,7,<rd,rd,vd,wr>,<l,b,b,l>,<ftch>
148A    define  EXTZV,7,<rd,rd,vd,wr>,<l,b,b,l>,<ftch>
14A0    define  INSV,11,<rd,rd,rd,vd>,<l,l,b,b>,<ftch>
14B6    define  ACBL,3,<rd,rd,md,bd>,<l,l,l,w>,<loop>
14CC    define  AOBLSS,3,<rd,md,bd>,<l,l,b>,<loop>
14E2    define  AOBLEQ,3,<rd,md,bd>,<l,l,b>,<loop>
14F8    define  SOBGEQ,3,<md,bd>,<l,b>,<loop>
150E    define  SOBGTR,3,<md,bd>,<l,b>,<loop>
1524    define  CVTLB,1,<rd,wr>,<l,b>,<dest>
153A    define  CVTLW,1,<rd,wr>,<l,w>,<dest>
1550    define  ASHP,100,<rd,rd,ad,rd,rd,ad>,<b,w,b,b,w,b>,<cont>
1566    define  CVTLP,100,<rd,rd,ad>,<l,w,b>,<cont>
157C    define  CALLG,30,<ad,ad>,<b,b>,<brch>
1592    define  CALLS,30,<rd,ad>,<l,b>,<brch>
15A8    define  XFC,10,<>,<>,<brch>
15BE    define  ESCD,10,<>,<>
15D4    define  ESCE,10,<>,<>,<brch>
15EA    define  ESCF,10,<>,<>,<brch>
1600    .rept   50
1600    define  RESRV,0,<>,<>,<brch>
1600    .endr
1A4C    define  CVTDH,20,<rd,wr>,<q,o>
1A62    define  CVTGF,3,<rd,wr>,<q,l>
1A78    .rept   12
1A78    define  RESRV,0,<>,<>,<brch>
1A78    .endr
1B80    define  ADDG2,4,<rd,md>,<q,q>
1B96    define  ADDG3,4,<rd,rd,wr>,<q,q,q>
1BAC    define  SUBG2,4,<rd,md>,<q,q>
1BC2    define  SUBG3,4,<rd,rd,wr>,<q,q,q>
1BD8    define  MULG2,7,<rd,md>,<q,q>
1BEE    define  MULG3,7,<rd,rd,wr>,<q,q,q>
1C04    define  DIVG2,28,<rd,md>,<q,q>
1C1A    define  DIVG3,28,<rd,rd,wr>,<q,q,q>
1C30    define  CVTGB,3,<rd,wr>,<q,b>
```

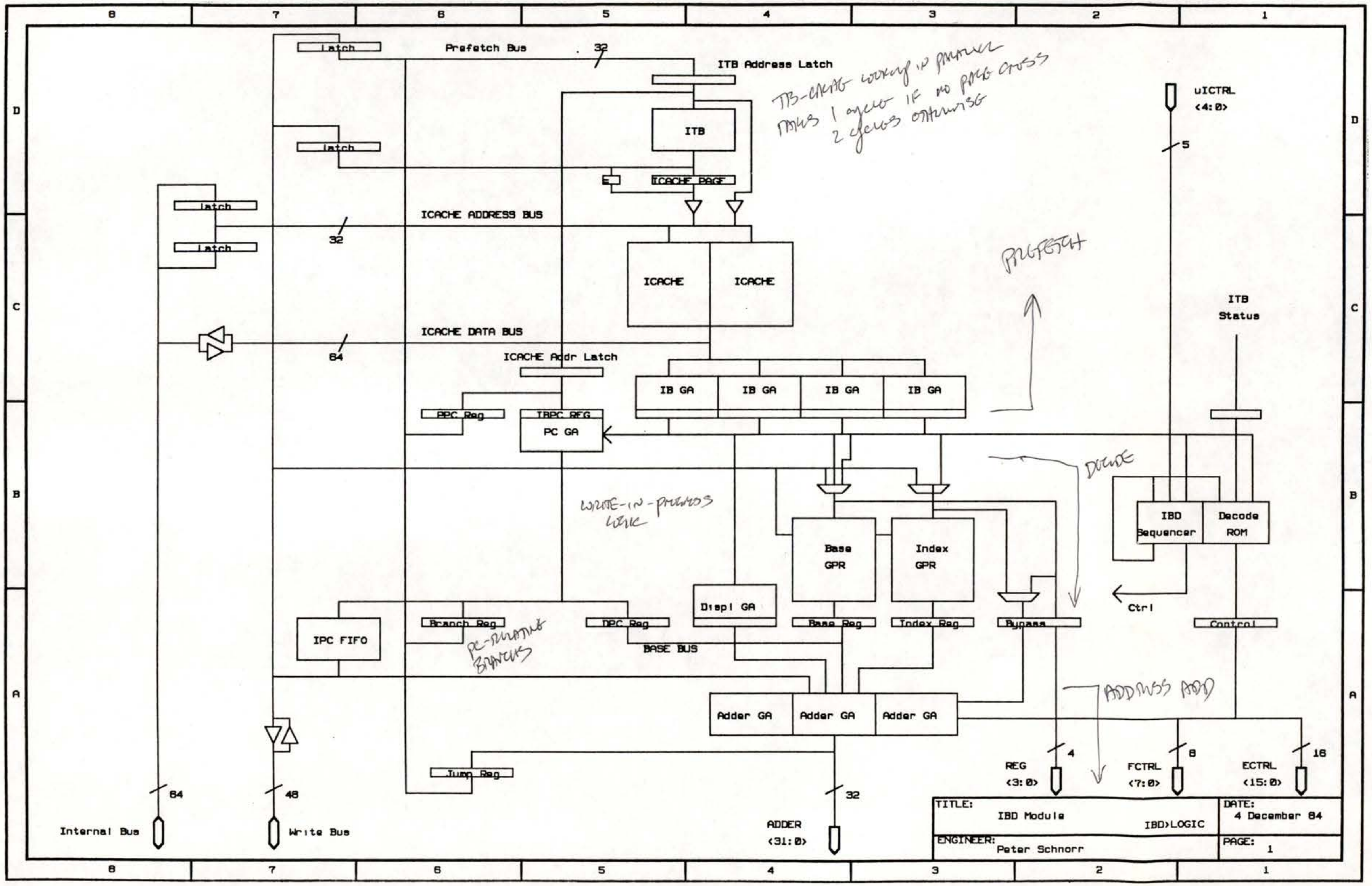
```
1C46          define  CVTGW,3,<rd,wr>,<q,w>
1C5C          define  CVTGL,3,<rd,wr>,<q,l>
1C72          define  CVTRGL,3,<rd,wr>,<q,l>
1C88          define  CVTBG,3,<rd,wr>,<b,q>
1C9E          define  CVTWG,3,<rd,wr>,<w,q>
1CB4          define  CVTLG,3,<rd,wr>,<l,q>
1CCA          define  ACBG,5,<rd,rd,md,bd>,<q,q,q,w>,<loop>
1CE0          define  MOVG,3,<rd,wr>,<q,q>
1CF6          define  CMPG,3,<rd,rd>,<q,q>
1D0C          define  MNEGG,3,<rd,wr>,<q,q>
1D22          define  TSTG,2,<rd>,<q>
1D38          define  EMODG,10,<rd,rd,rd,wr,wr>,<q,w,q,l,q>,<cont>
1D4E          define  POLYG,100,<rd,rd,ad>,<q,w,b>,<cont>
1D64          define  CVTGH,20,<rd,wr>,<q,o>
1D7A          .rept   9
1D7A          define  RESRV,0,<>,<>,<brch>
1D7A          .endr
1E40          define  ADDH2,8,<rd,md>,<o,o>
1E56          define  ADDH3,8,<rd,rd,wr>,<o,o,o>
1E6C          define  SUBH2,8,<rd,md>,<o,o>
1E82          define  SUBH3,8,<rd,rd,wr>,<o,o,o>
1E98          define  MULH2,14,<rd,md>,<o,o>
1EAE          define  MULH3,14,<rd,rd,wr>,<o,o,o>
1EC4          define  DIVH2,56,<rd,md>,<o,o>
1EDA          define  DIVH3,56,<rd,rd,wr>,<o,o,o>
1EF0          define  CVTHB,6,<rd,wr>,<o,b>
1F06          define  CVTHW,6,<rd,wr>,<o,w>
1F1C          define  CVTHL,6,<rd,wr>,<o,l>
1F32          define  CVTRHL,6,<rd,wr>,<o,l>
1F48          define  CVTBH,6,<rd,wr>,<b,o>
1F5E          define  CVTWH,6,<rd,wr>,<w,o>
1F74          define  CVTLH,6,<rd,wr>,<l,o>
1F8A          define  ACBH,10,<rd,rd,md,bd>,<o,o,o,w>,<loop>
1FA0          define  MOVH,6,<rd,wr>,<o,o>
1FB6          define  CMPH,6,<rd,rd>,<o,o>
1FCC          define  MNEGH,6,<rd,wr>,<o,o>
1FE2          define  TSTH,4,<rd>,<o>
1FF8          define  EMODH,20,<rd,rd,rd,wr,wr>,<o,w,o,l,o>,<cont>
200E          define  POLYH,100,<rd,rd,ad>,<o,w,b>,<cont>
2024          define  CVTHG,20,<rd,wr>,<o,q>
203A          .rept   5
203A          define  RESRV,0,<>,<>,<brch>
203A          .endr
20A8          define  CLRO,4,<wr>,<o>
20BE          define  MOVO,6,<rd,wr>,<o,o>
20D4          define  MOVAO,1,<ad,wr>,<o,l>
20EA          define  PUSHAO,1,<ad>,<o>,<push>
2100          .rept   24
2100          define  RESRV,0,<>,<>,<brch>
2100          .endr
2310          define  CVTFH,20,<rd,wr>,<l,o>
2326          define  CVTFG,3,<rd,wr>,<l,q>
233C          .rept   92
233C          define  RESRV,0,<>,<>,<brch>
233C          .endr
2B24          define  CVTHF,20,<rd,wr>,<o,l>
2B3A          define  CVTHD,20,<rd,wr>,<o,q>
```

```
                        2B50    400             .rept   8
                        2B50    401             define  RESRV,0,<>,<>,<brch>
                        2B50    402             .endr
                        2C00    403
                        2C00    404             .end
```

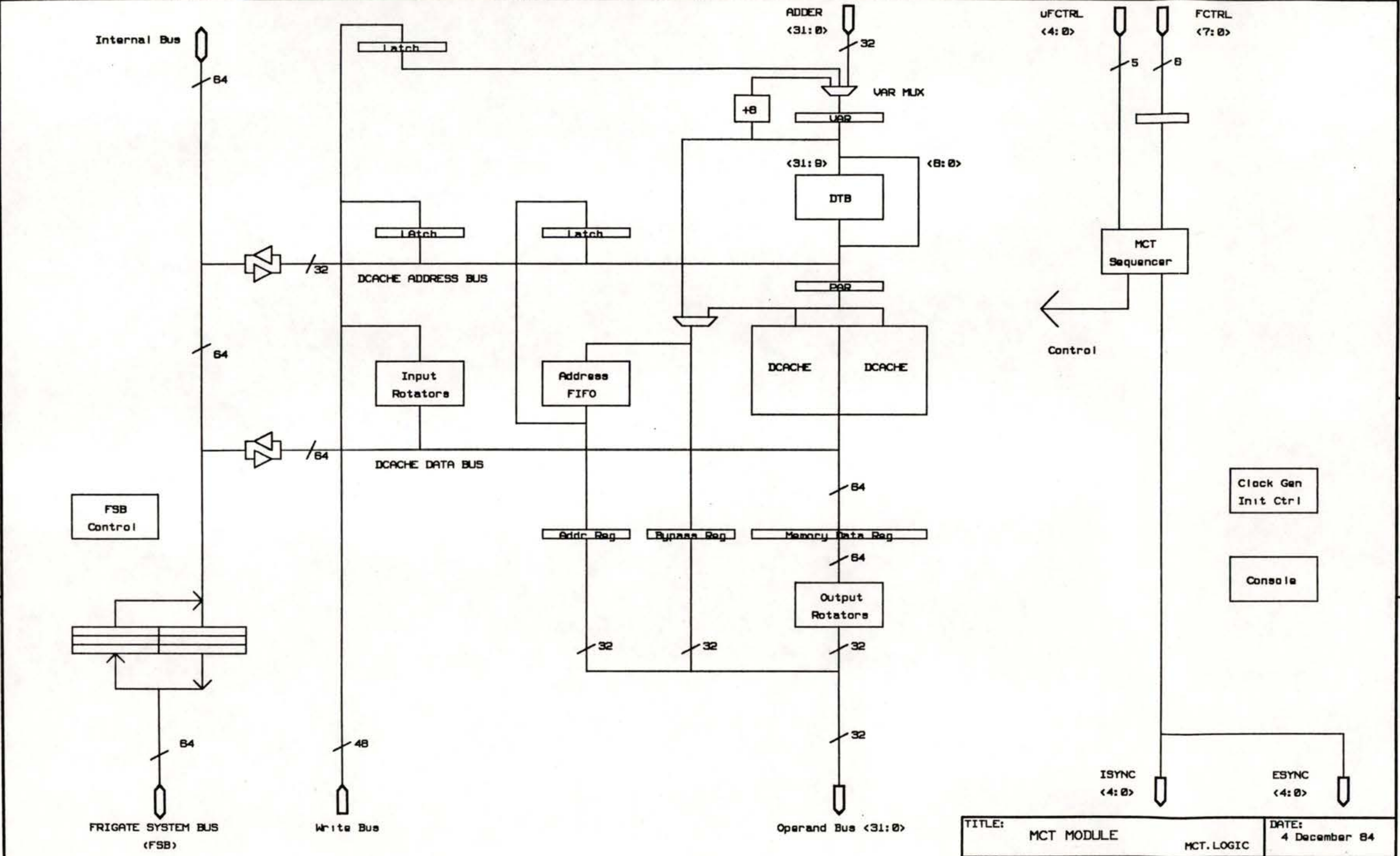


FSYSTEM.LOGIC

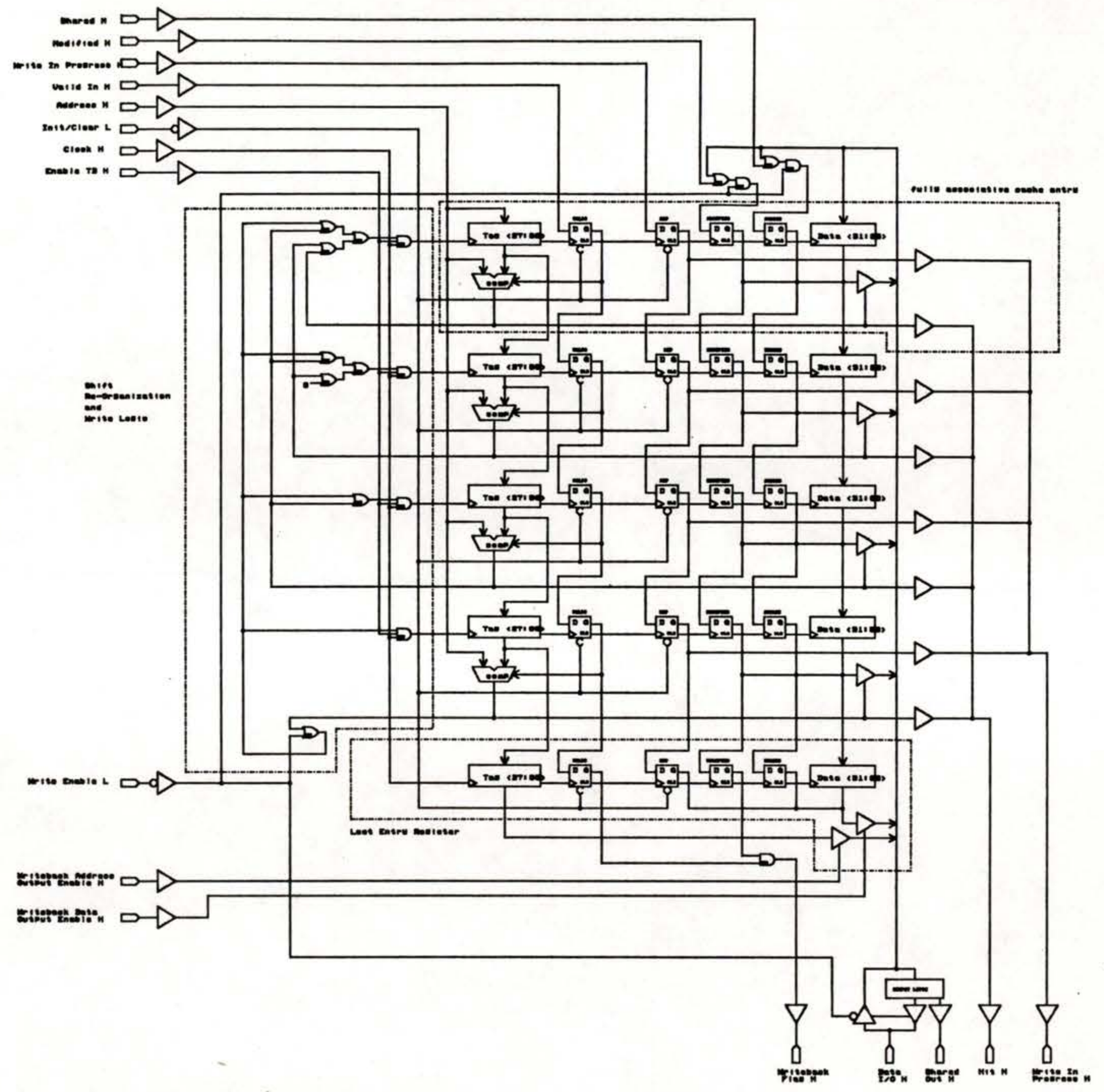
TITLE: Frigate System Block Diagram	DATE: 4 December 84
ENGINEER: Peter Schnorr	PAGE: 1



TITLE: IBD Module IBD>LOGIC	DATE: 4 December 84
ENGINEER: Peter Schnorr	PAGE: 1



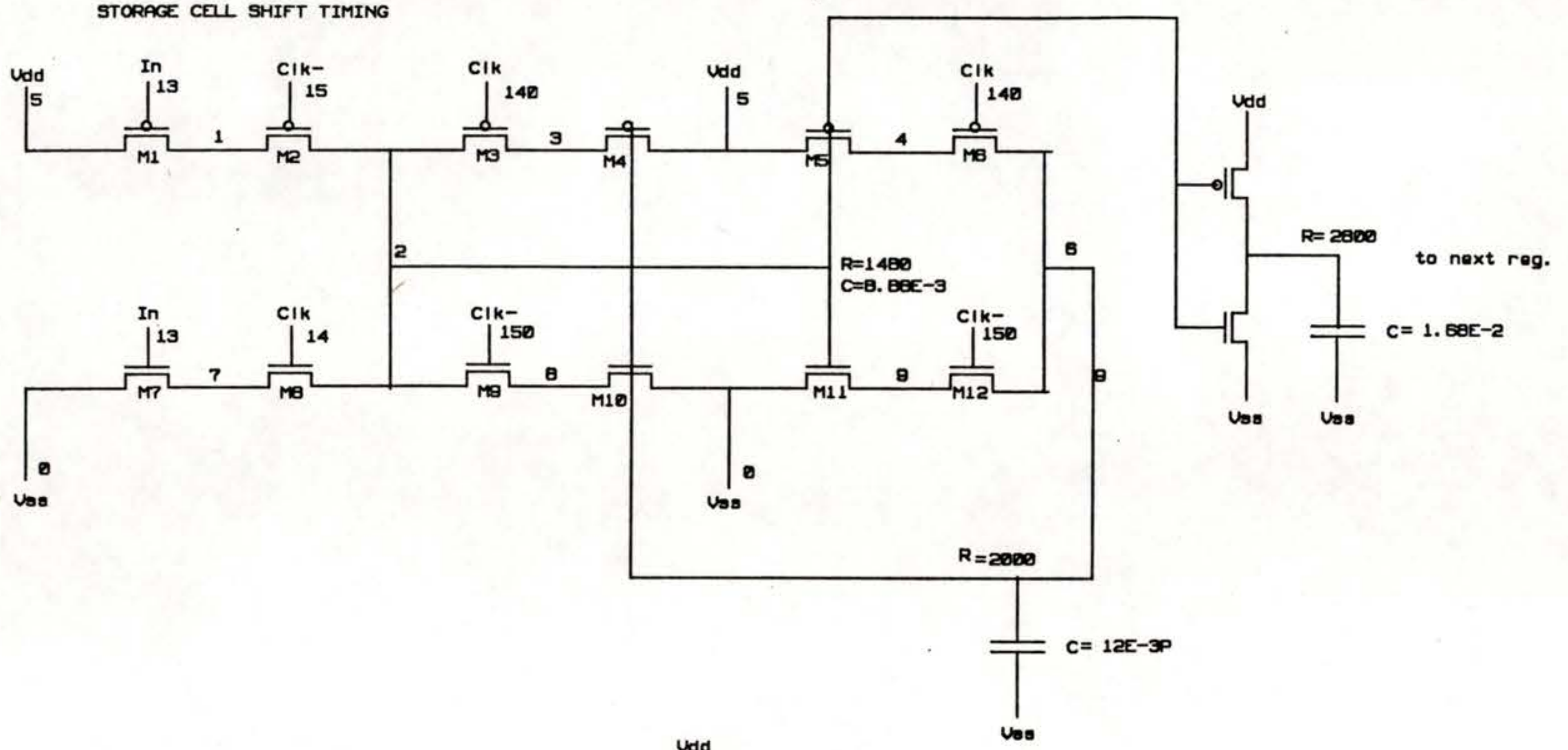
TITLE: MCT MODULE	MCT.LOGIC	DATE: 4 December 84
ENGINEER: Peter Schnorr		PAGE: 1 of 1



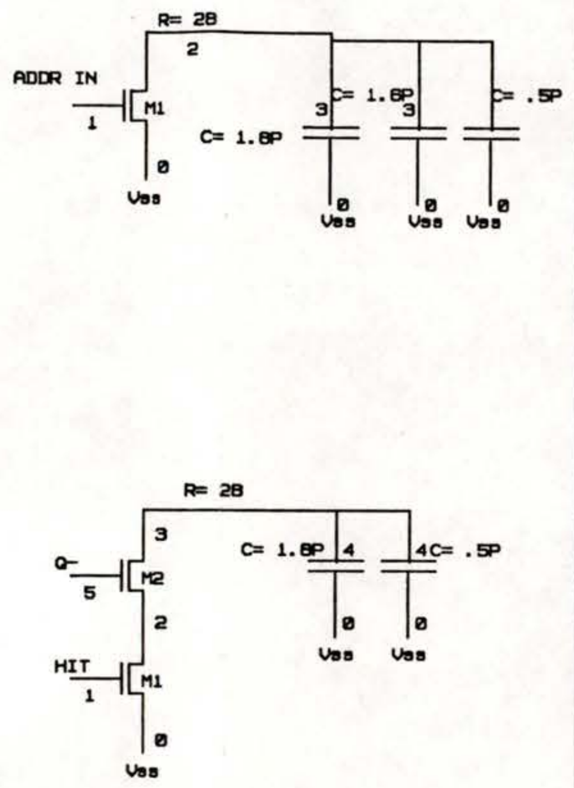
4 Entry General Purpose TB or Cache Chip

DECwest Engineering
October 2, 1984
rev Nov. 11, 1984 -- write back
address and data registers

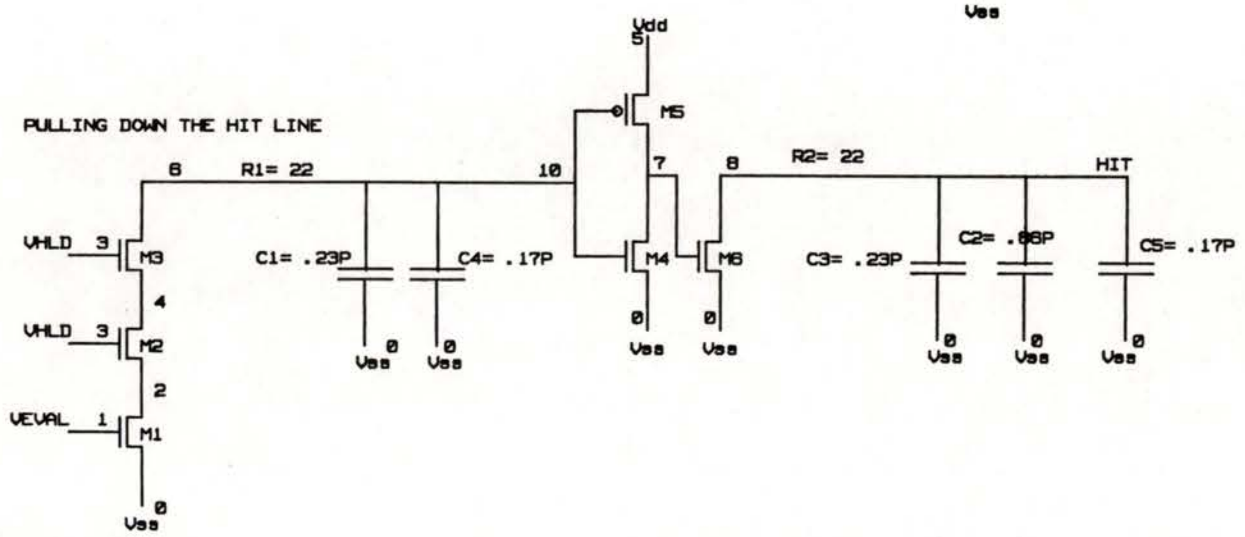
STORAGE CELL SHIFT TIMING



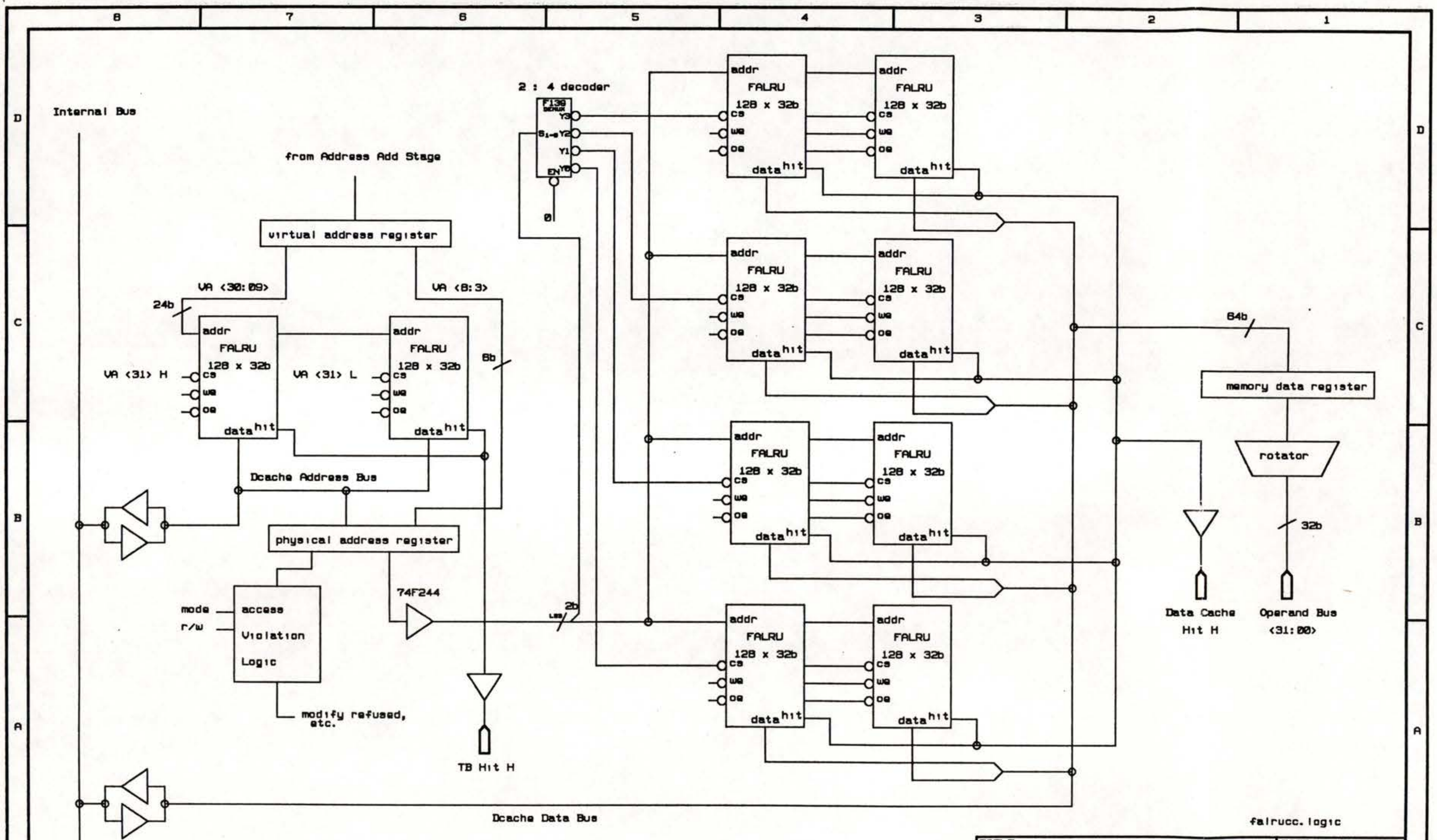
PULLING DOWN THE ADDRESS LINE



PULLING DOWN THE HIT LINE



TITLE: TLB Spice Model	DATE: 11-27-84
ENGINEER: Ted Kehl	PAGE:

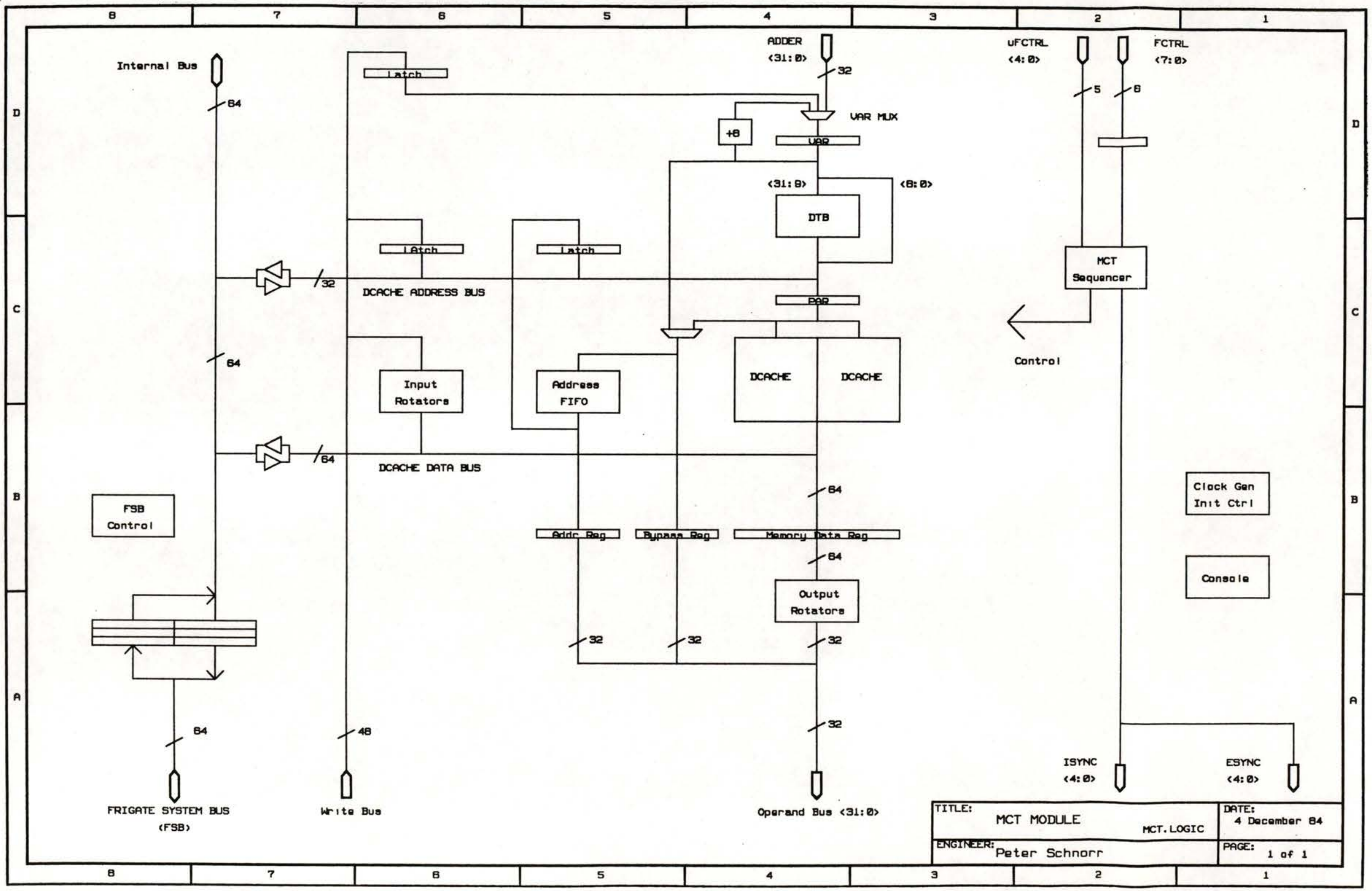


TITLE: Frigat2 FALRU TB/cache Chip

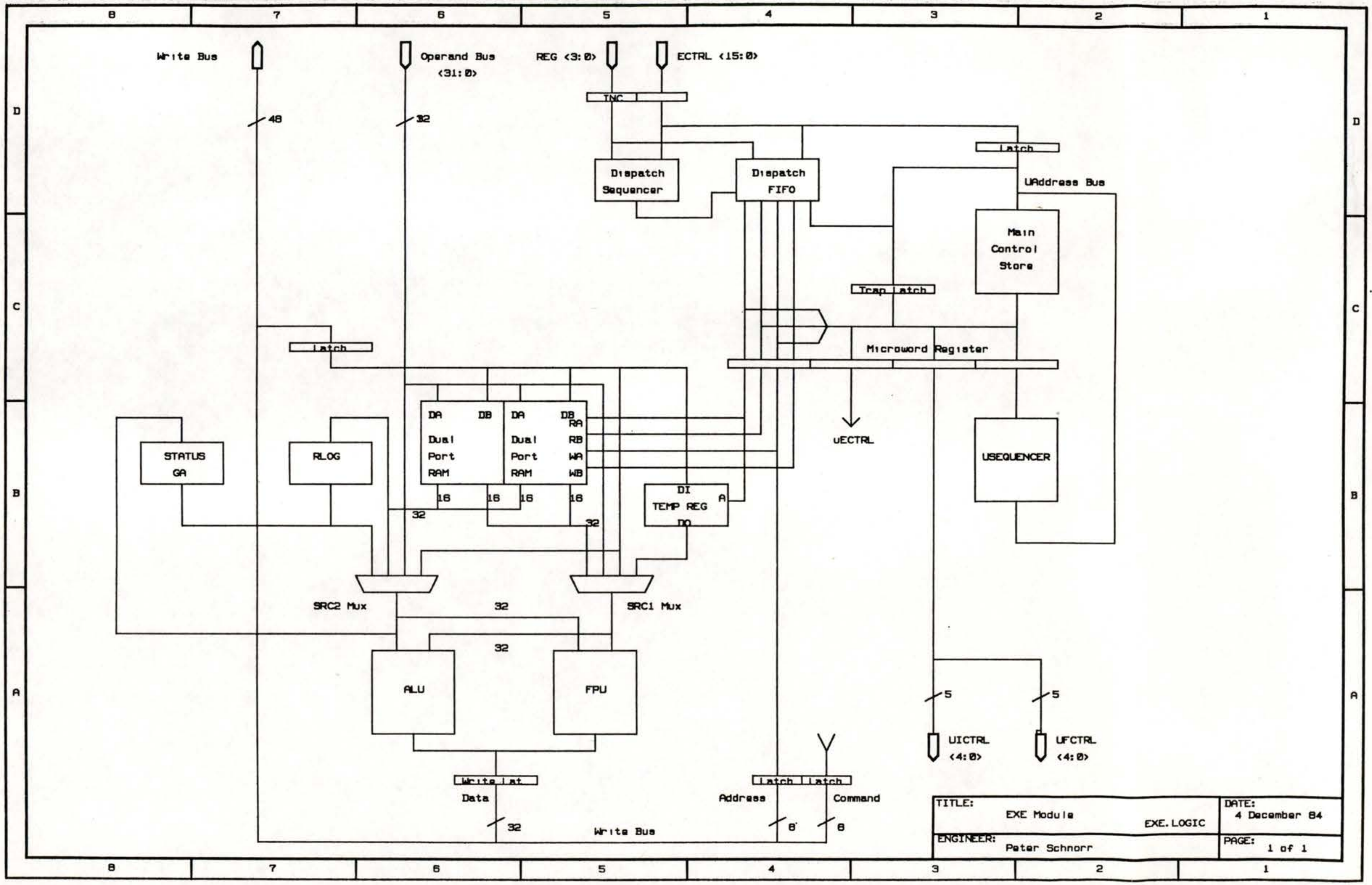
DATE: Nov. 29, 1984

ENGINEER: Bruce Butts

PAGE: 1/1



TITLE: MCT MODULE MCT.LOGIC	DATE: 4 December 84
ENGINEER: Peter Schnorr	PAGE: 1 of 1



TITLE: EXE Module EXE.LOGIC

DATE: 4 December 84

ENGINEER: Peter Schnorr

PAGE: 1 of 1

```
Frigate pipeline simulation model analysis of file dua1:[cutler]pjacobi.cod
Simulation was run on 29-NOV-1984 09:20:27.73
Data cache miss rate is set at  0%
Data cache miss forced write rate is set at  0%
Static branch prediction was used to predict conditional branches

Total number of simulation cycles =  38244387
Total number of instructions executed =  13569313
Average number of cycles per instruction =  2.82

Number of instructions that stop decode =       366860
Number of instructions that stop fetch =          636
Total number of branching instructions =      822576
Number of branches targets within the same virtual page =    717362 — PESSIMISTIC
Percent branches targets within the same virtual page =  87.21
Number of conditional branch instructions =      707332
Percent conditional branch instructions =  85.99
Percent of branches predicted correctly =  91.99
Percent of branches incorrectly predicted =   8.01
Number of unconditional branches =       85893
Percent unconditional branches =  10.44
Number of instructions that stop pipe and then branch =       29351
Percent stop and branches =   3.57

          Pipeline Utilization Cycles

Stage          Idle     Stall      Wait      Work
Prefetch    1381562   4322688    803487  31736650
Decode      2239642   2007957   2284080  31712708
Address     4524255    476418   1512348  31731366
Operand     6120074      7372         0  32116941
Execute    11756385         0         0  26017291

Autoinc/dec register write wait cycles =          0
Register base wait cycles =          0
Double invalid register wait cycles =   2284080
Indirect autoinc/dec register write wait cycles =          0

        Pipeline Utilization Percent

Stage      Idle Stall  Wait  Work
Prefetch    3.6  11.3   2.1  83.0
Decode      5.9   5.3   6.0  82.9
Address    11.8   1.2   4.0  83.0
Operand    16.0   0.0   0.0  84.0
Execute    30.7   0.0   0.0  68.0
$ run fb
dua1:[cutler]pjacobi
0
0
10
1
2
```

```
Frigate pipeline simulation model analysis of file dua1:[cutler]pjacobi.cod
Simulation was run on 30-NOV-1984 04:50:32.76
Data cache miss rate is set at  0%
Data cache miss forced write rate is set at  0%
Dynamic branch prediction was used to predict conditional branches
        Branch table size is  1024 entries
        Branch counter width is    1 bits
        Branch block size is    4 bytes

Total number of simulation cycles =  38212106
Total number of instructions executed =  13569313
Average number of cycles per instruction =  2.82

Number of instructions that stop decode =       36686
Number of instructions that stop fetch =          636
Total number of branching instructions =     822576
Number of branches targets within the same virtual page =     679332
Percent branches targets within the same virtual page =  82.59
Number of conditional branch instructions =     707332
Percent conditional branch instructions =  85.99
Percent of branches predicted correctly =  94.38
Percent of branches incorrectly predicted =   5.62
Number of unconditional branches =      85893
Percent unconditional branches =  10.44
Number of instructions that stop pipe and then branch =       29351
Percent stop and branches =   3.57

            Pipeline Utilization Cycles

Stage          Idle      Stall       Wait      Work
Prefetch    1381005    4309803     881155  31640143
Decode      2297082    1996487    2284075  31634462
Address     4566146     474583    1503683  31667694
Operand     6132400       5993          0  32073713
Execute    11724104          0          0  26017291

Autoinc/dec register write wait cycles =          0
Register base wait cycles =            0
Double invalid register wait cycles =    2284075
Indirect autoinc/dec register write wait cycles =          0

        Pipeline Utilization Percent

Stage       Idle Stall  Wait  Work
Prefetch     3.6  11.3   2.3  82.8
Decode       6.0   5.2   6.0  82.8
Address     11.9   1.2   3.9  82.9
Operand     16.0   0.0   0.0  83.9
Execute     30.7   0.0   0.0  68.1
$ run fb
dua1:[cutler]pjacobi
0
0
10
2
2
```

```
Frigate pipeline simulation model analysis of file dua1:[cutler]pjacobi.cod
Simulation was run on  1-DEC-1984 00:28:29.43
Data cache miss rate is set at  0%
Data cache miss forced write rate is set at  0%
Dynamic branch prediction was used to predict conditional branches
        Branch table size is  1024 entries
        Branch counter width is    2 bits
        Branch block size is    4 bytes

Total number of simulation cycles =  38108771
Total number of instructions executed =  13569313
Average number of cycles per instruction =  2.81

Number of instructions that stop decode =       36686
Number of instructions that stop fetch =          636
Total number of branching instructions =      822576
Number of branches targets within the same virtual page =      707199
Percent branches targets within the same virtual page =  85.97
Number of conditional branch instructions =      707332
Percent conditional branch instructions =  85.99
Percent of branches predicted correctly =  96.65
Percent of branches incorrectly predicted =   3.35
Number of unconditional branches =       85893
Percent unconditional branches =  10.44
Number of instructions that stop pipe and then branch =       29351
Percent stop and branches =   3.57

          Pipeline Utilization Cycles

Stage          Idle      Stall        Wait       Work
Prefetch    1381085    4317812      823796   31586078
Decode      2224093    2004492     2284073   31596113
Address     4477388     475963     1511678   31643742
Operand     6038039       7372           0   32063360
Execute    11620769          0           0   26017291

Autoinc/dec register write wait cycles =            0
Register base wait cycles =            0
Double invalid register wait cycles =    2284073
Indirect autoinc/dec register write wait cycles =            0

       Pipeline Utilization Percent

Stage       Idle Stall  Wait  Work
Prefetch     3.6  11.3   2.2  82.9
Decode       5.8   5.3   6.0  82.9
Address     11.7   1.2   4.0  83.0
Operand     15.8   0.0   0.0  84.1
Execute     30.5   0.0   0.0  68.3
$ run fb
dua1:[cutler]pjacobi
0
0
10
3
2
```

```
Frigate pipeline simulation model analysis of file dual:[cutler]pjacobi.cod
Simulation was run on  1-DEC-1984 19:48:16.37
Data cache miss rate is set at  0%
Data cache miss forced write rate is set at  0%
Dynamic branch prediction was used to predict conditional branches
       Branch table size is  1024 entries
       Branch counter width is    3 bits
       Branch block size is    4 bytes

Total number of simulation cycles =  38109608
Total number of instructions executed =  13569313
Average number of cycles per instruction =  2.81

Number of instructions that stop decode =       36686
Number of instructions that stop fetch =          636
Total number of branching instructions =      822576
Number of branches targets within the same virtual page =     707247
Percent branches targets within the same virtual page =  85.98
Number of conditional branch instructions =     707332
Percent conditional branch instructions =  85.99
Percent of branches predicted correctly =  96.63
Percent of branches incorrectly predicted =   3.37
Number of unconditional branches =      85893
Percent unconditional branches =  10.44
Number of instructions that stop pipe and then branch =      29351
Percent stop and branches =   3.57

          Pipeline Utilization Cycles

Stage          Idle     Stall       Wait      Work
Prefetch   1381129   4317785     823540  31587154
Decode     2224181   2004466    2284071  31596890
Address    4477791    475956    1511659  31644202
Operand    6038374      7372          0  32063862
Execute   11621606         0          0  26017291

Autoinc/dec register write wait cycles =          0
Register base wait cycles =           0
Double invalid register wait cycles =    2284071
Indirect autoinc/dec register write wait cycles =          0

       Pipeline Utilization Percent

Stage      Idle  Stall  Wait  Work
Prefetch    3.6   11.3   2.2  82.9
Decode      5.8    5.3   6.0  82.9
Address    11.7    1.2   4.0  83.0
Operand    15.8    0.0   0.0  84.1
Execute    30.5    0.0   0.0  68.3
$ run fb
dual:[cutler]pjacobi
0
0
10
4
2
```

```
Frigate pipeline simulation model analysis of file dual:[cutler]pjacobi.cod
 Simulation was run on  2-DEC-1984 14:46:15.96
 Data cache miss rate is set at  0%
 Data cache miss forced write rate is set at  0%
 Dynamic branch prediction was used to predict conditional branches
        Branch table size is  1024 entries
        Branch counter width is    4 bits
        Branch block size is    4 bytes

Total number of simulation cycles =  38111133
Total number of instructions executed =  13569313
Average number of cycles per instruction =  2.81

Number of instructions that stop decode =       36686
Number of instructions that stop fetch =          636
Total number of branching instructions =     822576
Number of branches targets within the same virtual page =     706763
Percent branches targets within the same virtual page =  85.92
Number of conditional branch instructions =    707332
Percent conditional branch instructions =  85.99
Percent of branches predicted correctly =  96.60
Percent of branches incorrectly predicted =   3.40
Number of unconditional branches =      85893
Percent unconditional branches =  10.44
Number of instructions that stop pipe and then branch =      29351
Percent stop and branches =   3.57

           Pipeline Utilization Cycles

Stage          Idle      Stall       Wait       Work
Prefetch    1381163    4317744     824336   31587890
Decode      2225102    2004432    2284074   31597525
Address     4479036     475937    1511644   31644516
Operand     6039656       7372          0   32064105
Execute    11623131          0          0   26017291

Autoinc/dec register write wait cycles =          0
Register base wait cycles =           0
Double invalid register wait cycles =    2284074
Indirect autoinc/dec register write wait cycles =          0

        Pipeline Utilization Percent

Stage       Idle  Stall  Wait  Work
Prefetch     3.6  11.3   2.2  82.9
Decode       5.8   5.3   6.0  82.9
Address     11.8   1.2   4.0  83.0
Operand     15.8   0.0   0.0  84.1
Execute     30.5   0.0   0.0  68.3
$ run fb
dual:[cutler]pjacobi
0
0
12
2
2
```

```
Frigate pipeline simulation model analysis of file dua1:[cutler]pjacobi.cod
Simulation was run on  3-DEC-1984 09:19:49.71
Data cache miss rate is set at  0%
Data cache miss forced write rate is set at  0%
Dynamic branch prediction was used to predict conditional branches
       Branch table size is  4096 entries
       Branch counter width is    2 bits
       Branch block size is    4 bytes

Total number of simulation cycles =  38108703
Total number of instructions executed =  13569313
Average number of cycles per instruction =  2.81

Number of instructions that stop decode =      36686
Number of instructions that stop fetch =         636
Total number of branching instructions =     822576
Number of branches targets within the same virtual page =       707229
Percent branches targets within the same virtual page =  85.98
Number of conditional branch instructions =    707332
Percent conditional branch instructions =  85.99
Percent of branches predicted correctly =  96.65
Percent of branches incorrectly predicted =   3.35
Number of unconditional branches =      85893
Percent unconditional branches =  10.44
Number of instructions that stop pipe and then branch =       29351
Percent stop and branches =   3.57

           Pipeline Utilization Cycles

Stage          Idle      Stall       Wait      Work
Prefetch    1381089    4317820     823729  31586065
Decode      2224025    2004493    2284080  31596105
Address     4477315     475963    1511681  31643744
Operand     6037966       7372          0  32063365
Execute    11620701          0          0  26017291

Autoinc/dec register write wait cycles =          0
Register base wait cycles =           0
Double invalid register wait cycles =   2284080
Indirect autoinc/dec register write wait cycles =          0

       Pipeline Utilization Percent

Stage      Idle Stall  Wait  Work
Prefetch    3.6  11.3   2.2  82.9
Decode      5.8   5.3   6.0  82.9
Address    11.7   1.2   4.0  83.0
Operand    15.8   0.0   0.0  84.1
Execute    30.5   0.0   0.0  68.3
$run fb
dua1:[cutler]pjacobi
0
0
12
4
2
```

```
Frigate pipeline simulation model analysis of file dua1:[cutler]pjacobi.cod
Simulation was run on  4-DEC-1984 04:48:54.57
Data cache miss rate is set at  0%
Data cache miss forced write rate is set at  0%
Dynamic branch prediction was used to predict conditional branches
        Branch table size is  4096 entries
        Branch counter width is    4 bits
        Branch block size is    4 bytes

Total number of simulation cycles =  38111020
Total number of instructions executed =  13569313
Average number of cycles per instruction =  2.81

Number of instructions that stop decode =       36686
Number of instructions that stop fetch =          636
Total number of branching instructions =     822576
Number of branches targets within the same virtual page =      706788
Percent branches targets within the same virtual page =  85.92
Number of conditional branch instructions =     707332
Percent conditional branch instructions =  85.99
Percent of branches predicted correctly =  96.61
Percent of branches incorrectly predicted =   3.39
Number of unconditional branches =       85893
Percent unconditional branches =  10.44
Number of instructions that stop pipe and then branch =      29351
Percent stop and branches =    3.57

           Pipeline Utilization Cycles

Stage          Idle      Stall        Wait      Work
Prefetch   1381167    4317744     824273  31587836
Decode     2225026    2004426    2284080  31597488
Address    4478949     475931    1511646  31644494
Operand    6039549       7372          0  32064099
Execute   11623018          0          0  26017291

Autoinc/dec register write wait cycles =          0
Register base wait cycles =           0
Double invalid register wait cycles =   2284080
Indirect autoinc/dec register write wait cycles =         0

       Pipeline Utilization Percent

Stage      Idle Stall  Wait  Work
Prefetch    3.6  11.3   2.2  82.9
Decode      5.8   5.3   6.0  82.9
Address    11.8   1.2   4.0  83.0
Operand    15.8   0.0   0.0  84.1
Execute    30.5   0.0   0.0  68.3
$ run fb
dua1:[cutler]pjacobi
0
0
14
2
2
```

```
Frigate pipeline simulation model analysis of file dual:[cutler]pjacobi.cod
Simulation was run on  5-DEC-1984 00:11:18.76
Data cache miss rate is set at  0%
Data cache miss forced write rate is set at  0%
Dynamic branch prediction was used to predict conditional branches
        Branch table size is 16384 entries
        Branch counter width is    2 bits
        Branch block size is    4 bytes

Total number of simulation cycles =  38108678
Total number of instructions executed =  13569313
Average number of cycles per instruction =  2.81

Number of instructions that stop decode =      36686
Number of instructions that stop fetch =         636
Total number of branching instructions =     822576
Number of branches targets within the same virtual page =     707237
Percent branches targets within the same virtual page =  85.98
Number of conditional branch instructions =     707332
Percent conditional branch instructions =  85.99
Percent of branches predicted correctly =  96.65
Percent of branches incorrectly predicted =   3.35
Number of unconditional branches =      85893
Percent unconditional branches =  10.44
Number of instructions that stop pipe and then branch =      29351
Percent stop and branches =   3.57

          Pipeline Utilization Cycles

Stage          Idle     Stall       Wait      Work
Prefetch    1381095   4317819     823715  31586049
Decode      2224011   2004493    2284080  31596094
Address     4477295    475963    1511681  31643739
Operand     6037942      7372          0  32063364
Execute    11620676         0          0  26017291

Autoinc/dec register write wait cycles =          0
Register base wait cycles =          0
Double invalid register wait cycles =   2284080
Indirect autoinc/dec register write wait cycles =         0

       Pipeline Utilization Percent

Stage      Idle Stall  Wait  Work
Prefetch    3.6  11.3   2.2  82.9
Decode      5.8   5.3   6.0  82.9
Address    11.7   1.2   4.0  83.0
Operand    15.8   0.0   0.0  84.1
Execute    30.5   0.0   0.0  68.3
$ run fb
dual:[cutler]pjacobi
0
0
14
4
2
```

```
Frigate pipeline simulation model analysis of file dua1:[cutler]pjacobi.cod
Simulation was run on  5-DEC-1984 19:46:22.92
Data cache miss rate is set at  0%
Data cache miss forced write rate is set at  0%
Dynamic branch prediction was used to predict conditional branches
        Branch table size is 16384 entries
        Branch counter width is    4 bits
        Branch block size is    4 bytes

Total number of simulation cycles =  38110977
Total number of instructions executed =  13569313
Average number of cycles per instruction =  2.81

Number of instructions that stop decode =      36686
Number of instructions that stop fetch =        636
Total number of branching instructions =    822576
Number of branches targets within the same virtual page =     706816
Percent branches targets within the same virtual page =  85.93
Number of conditional branch instructions =     707332
Percent conditional branch instructions =  85.99
Percent of branches predicted correctly =  96.61
Percent of branches incorrectly predicted =   3.39
Number of unconditional branches =      85893
Percent unconditional branches =  10.44
Number of instructions that stop pipe and then branch =      29351
Percent stop and branches =   3.57

        Pipeline Utilization Cycles

Stage          Idle     Stall      Wait     Work
Prefetch    1381173   4317763    824219 31587822
Decode      2224969   2004432   2284080 31597496
Address     4478883    475931   1511652 31644511
Operand     6039478      7372         0 32064127
Execute    11622975         0         0 26017291

Autoinc/dec register write wait cycles =          0
Register base wait cycles =          0
Double invalid register wait cycles =   2284080
Indirect autoinc/dec register write wait cycles =         0

       Pipeline Utilization Percent

Stage      Idle Stall  Wait  Work
Prefetch    3.6  11.3   2.2  82.9
Decode      5.8   5.3   6.0  82.9
Address    11.8   1.2   4.0  83.0
Operand    15.8   0.0   0.0  84.1
Execute    30.5   0.0   0.0  68.3
$ run fb
dua1:[cutler]pjacobi
5
33
12
2
2
```

5 CYCLE REFILL

```
Frigate 6 stage pipeline simulation model analysis of file ph.cod
Simulation was run on 30-NOV-1984 14:46:31.92
Data cache miss rate is set at  5%
Data cache miss forced write rate is set at 33%
Dynamic branch prediction was used to predict conditional branches
     Branch table size is  4096 entries
     Branch counter width is    2 bits
     Branch block size is    4 bytes

Total number of simulation cycles =   1555149
Total number of instructions executed =     629924
Average number of cycles per instruction = 2.47

Number of instructions that stop decode =          259
Number of instructions that stop fetch =          169
Total number of branching instructions =     195276
Number of branches targets within the same virtual page =     194926
Percent branches targets within the same virtual page =  99.82
Number of conditional branch instructions =     130058
Percent conditional branch instructions =  66.60
Percent of branches predicted correctly =  58.39
Percent of branches incorrectly predicted =  41.61
Number of unconditional branches =     65042
Percent unconditional branches =  33.31
Number of instructions that stop pipe and then branch =          176
Percent stop and branches =   0.09

          Pipeline Utilization Cycles

Stage          Idle       Stall       Wait      Work
Prefetch        9241     216135     163203    1166570
Decode        225611     216642        216    1112680
Address       279846     128659      87320    1058324
Translate     421228     129250          0    1004671
Operand       449807          0          0    1105342
Execute       915682          1          0     639375

Autoinc/dec register write wait cycles =          0
Register base wait cycles =          0
Double invalid register wait cycles =          216
Indirect autoinc/dec register write wait cycles =          0

       Pipeline Utilization Percent

Stage      Idle Stall  Wait  Work
Prefetch    0.6  13.9  10.5  75.0
Decode     14.5  13.9   0.0  71.5
Address    18.0   8.3   5.6  68.1
Translate  27.1   8.3   0.0  64.6
Operand    28.9   0.0   0.0  71.1
Execute    58.9   0.0   0.0  41.1
```

5 CYCLE REFILL

```
Frigate 5 stage pipeline (tb/cache) simulation model analysis of file ph.cod
Simulation was run on  4-DEC-1984 17:52:55.66
Data cache data miss rate is set at  5%
Data cache address miss rate is set at 50%
Data cache miss forced write rate is set at 33%
Dynamic branch prediction was used to predict conditional branches
    Branch table size is  4096 entries
    Branch counter width is    1 bits
    Branch block size is    4 bytes

Total number of simulation cycles =    1505467
Total number of instructions executed =    628924
Average number of cycles per instruction = $\boxed{2.39}$

Number of instructions that stop decode =         259
Number of instructions that stop fetch =        169
Total number of branching instructions =    195276
Number of branches targets within the same virtual page =    194940
Percent branches targets within the same virtual page =  99.83
Number of conditional branch instructions =    130058
Percent conditional branch instructions =  66.60
Percent of branches predicted correctly =  66.69
Percent of branches incorrectly predicted =  33.31
Number of unconditional branches =    65042
Percent unconditional branches =  33.31
Number of instructions that stop pipe and then branch =       176
Percent stop and branches =   0.09

        Pipeline Utilization Cycles

Stage          Idle     Stall      Wait      Work
Prefetch       8915    259072    152410   1085070
Decode       204391    258959       162   1041955
Address      247492    214942     44142    998891
Operand      314298         0         0   1191169
Execute      865975        26         0    639375

Autoinc/dec register write wait cycles =          0
Register base wait cycles =          0
Double invalid register wait cycles =        162
Indirect autoinc/dec register write wait cycles =         0

       Pipeline Utilization Percent

Stage      Idle Stall  Wait  Work
Prefetch    0.6  17.2  10.1  72.1
Decode     13.6  17.2   0.0  69.2
Address    16.4  14.3   2.9  66.4
Operand    20.9   0.0   0.0  79.1
Execute    57.5   0.0   0.0  42.5
```

7 CYCLE REFILL

```
Frigate 4 stage pipeline (tb|cache, register destination) simulation model analy
Simulation was run on  7-DEC-1984 10:35:27.12
Data cache data miss rate is set at  5%
Data cache address miss rate is set at 50%
Data cache miss forced write rate is set at 33%
Dynamic branch prediction was used to predict conditional branches
    Branch table size is  4096 entries
    Branch counter width is    1 bits
    Branch block size is    4 bytes

Total number of simulation cycles =   1261300
Total number of instructions executed =    628924
Average number of cycles per instruction =  2.01

Number of instructions that stop decode =         259
Number of instructions that stop fetch =         169
Total number of branching instructions =    195276
Number of branches targets within the same virtual page =     194944
Percent branches targets within the same virtual page =  99.83
Number of conditional branch instructions =    130058
Percent conditional branch instructions =  66.60
Percent of branches predicted correctly =  66.69
Percent of branches incorrectly predicted =  33.31
Number of unconditional branches =      65042
Percent unconditional branches =  33.31
Number of instructions that stop pipe and then branch =         176
Percent stop and branches =   0.09

         Pipeline Utilization Cycles

Stage          Idle     Stall      Wait      Work
Prefetch       8555    232039    152399    868307
Decode       203792    232204       108    825196
Operand      226751         0       327   1034222
Execute      621804        30         0    639375

Autoinc/dec register write wait cycles =          0
Register base wait cycles =          0
Double invalid register wait cycles =          108
Indirect autoinc/dec register write wait cycles =          0

        Pipeline Utilization Percent

Stage      Idle Stall  Wait  Work
Prefetch    0.7  18.4  12.1  68.8
Decode     16.2  18.4   0.0  65.4
Operand    18.0   0.0   0.0  82.0
Execute    49.3   0.0   0.0  50.7
Is '%SYSTEM-S-NORMAL, normal successful completion'
Command syntax error at or near 'EXITEXIT'
```

Wr

Neg

```
Frigate pipeline simulation model analysis of file dba3:[cutler]icp.cod
Simulation was run on 30-NOV-1984 14:51:47.34
Data cache miss rate is set at  0%
Data cache miss forced write rate is set at  0%
Static branch prediction was used to predict conditional branches

Total number of simulation cycles =  48244771
Total number of instructions executed =   9948733
Average number of cycles per instruction =  4.85

Number of instructions that stop decode =     594655
Number of instructions that stop fetch =     578555
Total number of branching instructions =    3281253
Number of branches targets within the same virtual page =    2503697
Percent branches targets within the same virtual page =  76.30
Number of conditional branch instructions =    2218007
Percent conditional branch instructions =  67.60
Percent of branches predicted correctly =  47.29
Percent of branches incorrectly predicted =  52.71
Number of unconditional branches =     562427
Percent unconditional branches =  17.14
Number of instructions that stop pipe and then branch =     500819
Percent stop and branches =  15.26

        Pipeline Utilization Cycles

Stage          Idle     Stall      Wait      Work
Prefetch   15608059   7884557   3383247  21368908
Decode     19769907   7574364    307682  20592818
Address    20490443   5064202   2816705  19873421
Operand    28374475         0         0  19870296
Execute    18317844         0         0  29649233

Autoinc/dec register write wait cycles =         390
Register base wait cycles =       2448
Double invalid register wait cycles =      304844
Indirect autoinc/dec register write wait cycles =           0

      Pipeline Utilization Percent

Stage      Idle Stall   Wait   Work
Prefetch   32.4  16.3    7.0   44.3
Decode     41.0  15.7    0.6   42.7
Address    42.5  10.5    5.8   41.2
Operand    58.8   0.0    0.0   41.2
Execute    38.0   0.0    0.0   61.5
```

```
Frigate pipeline simulation model analysis of file dba3:[cutler]icp.cod
Simulation was run on  1-DEC-1984 01:45:52.07
Data cache miss rate is set at  0%
Data cache miss forced write rate is set at  0%
Dynamic branch prediction was used to predict conditional branches
     Branch table size is  1024 entries
     Branch counter width is    1 bits
     Branch block size is    4 bytes

Total number of simulation cycles =  45635643
Total number of instructions executed =    9948733
Average number of cycles per instruction =  4.59

Number of instructions that stop decode =      594655
Number of instructions that stop fetch =     578555
Total number of branching instructions =   3281253
Number of branches targets within the same virtual page =    2482444
Percent branches targets within the same virtual page =  75.66
Number of conditional branch instructions =    2218007
Percent conditional branch instructions =  67.60
Percent of branches predicted correctly =  83.94
Percent of branches incorrectly predicted =  16.06
Number of unconditional branches =      562427
Percent unconditional branches =  17.14
Number of instructions that stop pipe and then branch =      500819
Percent stop and branches =  15.26

          Pipeline Utilization Cycles

Stage          Idle      Stall      Wait       Work
Prefetch   15416307    7520299   3745347   18953690
Decode     19203392    7227109    307675   18897467
Address    19199712    4810706   2762002   18863223
Operand    25861947          0         0   19773696
Execute    15708716          0         0   29649233

Autoinc/dec register write wait cycles =          383
Register base wait cycles =        2448
Double invalid register wait cycles =      304844
Indirect autoinc/dec register write wait cycles =            0

        Pipeline Utilization Percent

Stage      Idle Stall  Wait  Work
Prefetch   33.8  16.5   8.2  41.5
Decode     42.1  15.8   0.7  41.4
Address    42.1  10.5   6.1  41.3
Operand    56.7   0.0   0.0  43.3
Execute    34.4   0.0   0.0  65.0
```

```
Frigate pipeline simulation model analysis of file dba3:[cutler]icp.cod
Simulation was run on  1-DEC-1984 07:07:04.84
Data cache miss rate is set at  0%
Data cache miss forced write rate is set at  0%
Dynamic branch prediction was used to predict conditional branches
    Branch table size is  1024 entries
    Branch counter width is    2 bits
    Branch block size is    4 bytes

Total number of simulation cycles =  45312139
Total number of instructions executed =   9948733
Average number of cycles per instruction =  4.55

Number of instructions that stop decode =    594655
Number of instructions that stop fetch =    578555
Total number of branching instructions =   3281253
Number of branches targets within the same virtual page =    2496623
Percent branches targets within the same virtual page =  76.09
Number of conditional branch instructions =   2218007
Percent conditional branch instructions =  67.60
Percent of branches predicted correctly =  87.75
Percent of branches incorrectly predicted =  12.25
Number of unconditional branches =     562427
Percent unconditional branches =  17.14
Number of instructions that stop pipe and then branch =     500819
Percent stop and branches =  15.26

        Pipeline Utilization Cycles

Stage          Idle      Stall      Wait      Work
Prefetch   15404601    7536476   3614994  18756068
Decode     18987400    7248587    307675  18768477
Address    18906870    4823568   2771748  18809953
Operand    25544147          0         0  19767992
Execute    15365212          0         0  29649233

Autoinc/dec register write wait cycles =        383
Register base wait cycles =       2448
Double invalid register wait cycles =     304844
Indirect autoinc/dec register write wait cycles =          0

      Pipeline Utilization Percent

Stage      Idle Stall  Wait  Work
Prefetch   34.0  16.6   8.0  41.4
Decode     41.9  16.0   0.7  41.4
Address    41.7  10.6   6.1  41.5
Operand    56.4   0.0   0.0  43.6
Execute    34.0   0.0   0.0  65.4
```

```
Frigate pipeline simulation model analysis of file dba3:[cutler]icp.cod
Simulation was run on  1-DEC-1984 22:29:54.65
Data cache miss rate is set at  0%
Data cache miss forced write rate is set at  0%
Dynamic branch prediction was used to predict conditional branches
     Branch table size is  1024 entries
     Branch counter width is    3 bits
     Branch block size is    4 bytes

Total number of simulation cycles =  45307889
Total number of instructions executed =    9948733
Average number of cycles per instruction =  4.55

Number of instructions that stop decode =      594655
Number of instructions that stop fetch =     578555
Total number of branching instructions =    3281253
Number of branches targets within the same virtual page =    2503397
Percent branches targets within the same virtual page =  76.29
Number of conditional branch instructions =    2218007
Percent conditional branch instructions =  67.60
Percent of branches predicted correctly =  87.73
Percent of branches incorrectly predicted =  12.27
Number of unconditional branches =     562427
Percent unconditional branches =  17.14
Number of instructions that stop pipe and then branch =     500819
Percent stop and branches =  15.26

          Pipeline Utilization Cycles

Stage          Idle       Stall       Wait       Work
Prefetch   15403250    7541086    3603642   18759911
Decode     18973620    7254226     307653   18772390
Address    18895486    4826390    2773646   18812367
Operand    25537742          0          0   19770147
Execute    15380962          0          0   29649233

Autoinc/dec register write wait cycles =          361
Register base wait cycles =         2448
Double invalid register wait cycles =      304844
Indirect autoinc/dec register write wait cycles =            0

        Pipeline Utilization Percent

Stage      Idle Stall  Wait  Work
Prefetch   34.0  16.6   8.0  41.4
Decode     41.9  16.0   0.7  41.4
Address    41.7  10.7   6.1  41.5
Operand    56.4   0.0   0.0  43.6
Execute    33.9   0.0   0.0  65.4
```

```
Frigate pipeline simulation model analysis of file dba3:[cutler]icp.cod
Simulation was run on  2-DEC-1984 15:13:57.76
Data cache miss rate is set at  0%
Data cache miss forced write rate is set at  0%
Dynamic branch prediction was used to predict conditional branches
    Branch table size is  1024 entries
    Branch counter width is    4 bits
    Branch block size is    4 bytes

Total number of simulation cycles =  45309127
Total number of instructions executed =   9948733
Average number of cycles per instruction =  4.55

Number of instructions that stop decode =     594655
Number of instructions that stop fetch =     578555
Total number of branching instructions =   3281253
Number of branches targets within the same virtual page =   2503046
Percent branches targets within the same virtual page =  76.28
Number of conditional branch instructions =   2218007
Percent conditional branch instructions =  67.60
Percent of branches predicted correctly =  87.62
Percent of branches incorrectly predicted =  12.38
Number of unconditional branches =     562427
Percent unconditional branches =  17.14
Number of instructions that stop pipe and then branch =     500819
Percent stop and branches =  15.26

          Pipeline Utilization Cycles

Stage          Idle      Stall       Wait       Work
Prefetch   15401045    7546070    3595337   18766675
Decode     18963984    7259152     307642   18778349
Address    18885628    4829628    2773921   18819950
Operand    25539574          0          0   19769553
Execute    15382200          0          0   29649233

Autoinc/dec register write wait cycles =          350
Register base wait cycles =        2448
Double invalid register wait cycles =       304844
Indirect autoinc/dec register write wait cycles =          0

        Pipeline Utilization Percent

Stage      Idle Stall  Wait  Work
Prefetch   34.0  16.7   7.9  41.4
Decode     41.9  16.0   0.7  41.4
Address    41.7  10.7   6.1  41.5
Operand    56.4   0.0   0.0  43.6
Execute    33.9   0.0   0.0  65.4
```

```
Frigate pipeline simulation model analysis of file dba3:[cutler]icp.cod
Simulation was run on  3-DEC-1984 09:19:46.49
Data cache miss rate is set at  0%
Data cache miss forced write rate is set at  0%
Dynamic branch prediction was used to predict conditional branches
    Branch table size is  4096 entries
    Branch counter width is     2 bits
    Branch block size is    4 bytes

Total number of simulation cycles =  45286993
Total number of instructions executed =    9948733
Average number of cycles per instruction =  4.55

Number of instructions that stop decode =     594655
Number of instructions that stop fetch =     578555
Total number of branching instructions =    3281253
Number of branches targets within the same virtual page =    2497312
Percent branches targets within the same virtual page =  76.11
Number of conditional branch instructions =    2218007
Percent conditional branch instructions =  67.60
Percent of branches predicted correctly =  88.06
Percent of branches incorrectly predicted =  11.94
Number of unconditional branches =     562427
Percent unconditional branches =  17.14
Number of instructions that stop pipe and then branch =      500819
Percent stop and branches =  15.26

           Pipeline Utilization Cycles

Stage          Idle       Stall       Wait       Work
Prefetch   15402690    7540240    3604234   18739829
Decode     18971106    7249544     307675   18758668
Address    18884504    4823725    2772777   18805987
Operand    25519094          0          0   19767899
Execute    15360066          0          0   29649233

Autoinc/dec register write wait cycles =          383
Register base wait cycles =        2448
Double invalid register wait cycles =      304844
Indirect autoinc/dec register write wait cycles =            0

       Pipeline Utilization Percent

Stage       Idle Stall  Wait  Work
Prefetch    34.0  16.6   8.0  41.4
Decode      41.9  16.0   0.7  41.4
Address     41.7  10.7   6.1  41.5
Operand     56.3   0.0   0.0  43.7
Execute     33.9   0.0   0.0  65.5
```

```
Frigate pipeline simulation model analysis of file dba3:[cutler]icp.cod
Simulation was run on  4-DEC-1984 09:08:27.94
Data cache miss rate is set at  0%
Data cache miss forced write rate is set at  0%
Dynamic branch prediction was used to predict conditional branches
    Branch table size is  4096 entries
    Branch counter width is    4 bits
    Branch block size is    4 bytes

Total number of simulation cycles =  45255738
Total number of instructions executed =   9948733
Average number of cycles per instruction =  4.55

Number of instructions that stop decode =      594655
Number of instructions that stop fetch =      578555
Total number of branching instructions =    3281253
Number of branches targets within the same virtual page =    2504099
Percent branches targets within the same virtual page =  76.32
Number of conditional branch instructions =    2218007
Percent conditional branch instructions =  67.60
Percent of branches predicted correctly =  88.21
Percent of branches incorrectly predicted =  11.79
Number of unconditional branches =    562427
Percent unconditional branches =  17.14
Number of instructions that stop pipe and then branch =     500819
Percent stop and branches =  15.26

        Pipeline Utilization Cycles

Stage           Idle      Stall       Wait       Work
Prefetch   15398960    7550495    3570224   18736059
Decode     18927432    7260664     307643   18759999
Address    18831525    4830287    2775132   18818794
Operand    25486385          0          0   19769353
Execute    15328811          0          0   29649233

Autoinc/dec register write wait cycles =          351
Register base wait cycles =        2448
Double invalid register wait cycles =       304844
Indirect autoinc/dec register write wait cycles =          0

       Pipeline Utilization Percent

Stage      Idle Stall  Wait  Work
Prefetch   34.0  16.7   7.9  41.4
Decode     41.8  16.0   0.7  41.5
Address    41.6  10.7   6.1  41.6
Operand    56.3   0.0   0.0  43.7
Execute    33.9   0.0   0.0  65.5
```

```
Frigate pipeline simulation model analysis of file dba3:[cutler]icp.cod
Simulation was run on  5-DEC-1984 13:23:25.41
Data cache miss rate is set at  0%
Data cache miss forced write rate is set at  0%
Dynamic branch prediction was used to predict conditional branches
    Branch table size is 16384 entries
    Branch counter width is    2 bits
    Branch block size is    4 bytes

Total number of simulation cycles =  45284017
Total number of instructions executed =   9948733
Average number of cycles per instruction =  4.55

Number of instructions that stop decode =    594655
Number of instructions that stop fetch =    578555
Total number of branching instructions =   3281253
Number of branches targets within the same virtual page =   2497910
Percent branches targets within the same virtual page =  76.13
Number of conditional branch instructions =   2218007
Percent conditional branch instructions =  67.60
Percent of branches predicted correctly =  88.08
Percent of branches incorrectly predicted =  11.92
Number of unconditional branches =    562427
Percent unconditional branches =  17.14
Number of instructions that stop pipe and then branch =    500819
Percent stop and branches =  15.26

          Pipeline Utilization Cycles

Stage          Idle       Stall        Wait       Work
Prefetch   15400701     7540277     3599865   18743174
Decode     18966616     7249586      307675   18760140
Address    18880704     4823780     2772926   18806607
Operand    25516235           0           0   19767782
Execute    15357090           0           0   29649233

Autoinc/dec register write wait cycles =         383
Register base wait cycles =        2448
Double invalid register wait cycles =     304844
Indirect autoinc/dec register write wait cycles =           0

       Pipeline Utilization Percent

Stage      Idle Stall  Wait  Work
Prefetch   34.0  16.7   7.9  41.4
Decode     41.9  16.0   0.7  41.4
Address    41.7  10.7   6.1  41.5
Operand    56.3   0.0   0.0  43.7
Execute    33.9   0.0   0.0  65.5
```

```
Frigate pipeline simulation model analysis of file dba3:[cutler]icp.cod
Simulation was run on  6-DEC-1984 14:28:18.31
Data cache miss rate is set at  0%
Data cache miss forced write rate is set at  0%
Dynamic branch prediction was used to predict conditional branches
    Branch table size is 16384 entries
    Branch counter width is    4 bits
    Branch block size is    4 bytes

Total number of simulation cycles =  45245263
Total number of instructions executed =    9948733
Average number of cycles per instruction =  4.55

Number of instructions that stop decode =     594655
Number of instructions that stop fetch =     578555
Total number of branching instructions =   3281253
Number of branches targets within the same virtual page =    2505091
Percent branches targets within the same virtual page =  76.35
Number of conditional branch instructions =   2218007
Percent conditional branch instructions =  67.60
Percent of branches predicted correctly =  88.31
Percent of branches incorrectly predicted =  11.69
Number of unconditional branches =     562427
Percent unconditional branches =  17.14
Number of instructions that stop pipe and then branch =     500819
Percent stop and branches =  15.26

          Pipeline Utilization Cycles

Stage          Idle      Stall       Wait       Work
Prefetch  15396796    7550358    3562880   18735229
Decode    18918395    7260724     307642   18758502
Address   18821451    4830263    2775370   18818179
Operand   25476351          0          0   19768912
Execute   15318336          0          0   29649233

Autoinc/dec register write wait cycles =         350
Register base wait cycles =      2448
Double invalid register wait cycles =     304844
Indirect autoinc/dec register write wait cycles =          0

      Pipeline Utilization Percent

Stage      Idle Stall  Wait  Work
Prefetch  34.0  16.7   7.9  41.4
Decode    41.8  16.0   0.7  41.5
Address   41.6  10.7   6.1  41.6
Operand   56.3   0.0   0.0  43.7
Execute   33.9   0.0   0.0  65.5
```

```
Frigate pipeline simulation model analysis of file dba3:[cutler]iep.cod
Simulation was run on  7-DEC-1984 19:05:30.53
Data cache miss rate is set at  5%
Data cache miss forced write rate is set at 33%
Dynamic branch prediction was used to predict conditional branches
     Branch table size is  4096 entries
     Branch counter width is    2 bits
     Branch block size is    4 bytes

Total number of simulation cycles =  46534008
Total number of instructions executed =   9948733
Average number of cycles per instruction =  4.68

Number of instructions that stop decode =     594655
Number of instructions that stop fetch =     578555
Total number of branching instructions =   3281253
Number of branches targets within the same virtual page =    2497254
Percent branches targets within the same virtual page =  76.11
Number of conditional branch instructions =    2218007
Percent conditional branch instructions =  67.60
Percent of branches predicted correctly =  88.06
Percent of branches incorrectly predicted =  11.94
Number of unconditional branches =     562427
Percent unconditional branches =  17.14
Number of instructions that stop pipe and then branch =     500819
Percent stop and branches =  15.26

          Pipeline Utilization Cycles

Stage          Idle      Stall       Wait       Work
Prefetch   15481525    8706728    3604414   18741341
Decode     19009284    8463580     300971   18760173
Address    18851696    6053578    2821067   18807667
Operand    25485359          0          0   21048649
Execute    16607081          0          0   29649233

Autoinc/dec register write wait cycles =          356
Register base wait cycles =        2440
Double invalid register wait cycles =      298175
Indirect autoinc/dec register write wait cycles =            0

        Pipeline Utilization Percent

Stage      Idle Stall  Wait  Work
Prefetch   33.3  18.7   7.7  40.3
Decode     40.9  18.2   0.6  40.3
Address    40.5  13.0   6.1  40.4
Operand    54.8   0.0   0.0  45.2
Execute    35.7   0.0   0.0  63.7
  CUTLER          job terminated at  8-DEC-1984 10:00:07.34

  Accounting information:
  Buffered I/O count:             160     Peak working set size:    600
  Direct I/O count:             53062     Peak page file size:     1044
  Page faults:                   2292     Mounted volumes:            0
  Elapsed CPU time:     2 06:42:05.08     Elapsed time:     7 19:08:43.19
```

```
Instruction Frequency Data
This data was collected on 30-NOV-1984 08:54:02.73
This data was written to db3:[cutler]phanoi.cod
Total number of instructions traced was  15208945

Name        Count  Percent Cumula

MOVL      3670441   24.13   24.13
BEQL      2097259   13.79   37.92
DECL      2097163   13.79   51.71
BRB       1572940   10.34   62.05
INCL      1048710    6.90   68.95
CMPL      1048657    6.90   75.84
BLEQ      1048656    6.89   82.74
SUBL2     1048650    6.89   89.63
SUBL3     1048605    6.89   96.53
MNEGL      524298    3.45   99.98
MOVB          201    0.00   99.98
RSB           149    0.00   99.98
JMP           137    0.00   99.98
MOVAB         137    0.00   99.98
CLRL          116    0.00   99.98
BBC           114    0.00   99.98
BNEQ          112    0.00   99.98
BLBC          105    0.00   99.98
CASEB          99    0.00   99.98
BLSS           87    0.00   99.98
CVTBL          85    0.00   99.99
MOVQ           84    0.00   99.99
MOVAL          84    0.00   99.99
CMPB           81    0.00   99.99
PUSHL          78    0.00   99.99
BRW            75    0.00   99.99
INSV           75    0.00   99.99
TSTL           73    0.00   99.99
CLRB           71    0.00   99.99
BICL2          71    0.00   99.99
BGTR           69    0.00   99.99
RET            64    0.00   99.99
MOVW           62    0.00   99.99
CALLS          62    0.00   99.99
BSBW           60    0.00   99.99
CMPW           59    0.00   99.99
ACBB           54    0.00   99.99
ADDL3          53    0.00   99.99
MOVZWL         50    0.00   99.99
MOVZBL         50    0.00   99.99
ADDL2          48    0.00   99.99
SOBGEQ         46    0.00   99.99
BISL2          45    0.00   99.99
JSB            36    0.00   99.99
PUSHAB         36    0.00  100.00
SUBW3          36    0.00  100.00
MOVC3          35    0.00  100.00
BSBB           33    0.00  100.00
ADDW2          32    0.00  100.00
MOVC5          27    0.00  100.00
PUSHAL         27    0.00  100.00
ASHL           26    0.00  100.00
DECW           23    0.00  100.00
BBS            23    0.00  100.00
```

```
BLBS            23     0.00 100.00
CLRQ            22     0.00 100.00
MCOMB           22     0.00 100.00
BISPSW          21     0.00 100.00
LOCC            20     0.00 100.00
BICB3           20     0.00 100.00
AOBLSS          20     0.00 100.00
BGEQ            19     0.00 100.00
CLRW            17     0.00 100.00
MULL2           16     0.00 100.00
POPR            15     0.00 100.00
PUSHR           15     0.00 100.00
SUBW2           13     0.00 100.00
EDIV            12     0.00 100.00
CHME            12     0.00 100.00
CVTLP           12     0.00 100.00
SUBB3           11     0.00 100.00
TSTW            11     0.00 100.00
EXTZV           11     0.00 100.00
BLSSU           10     0.00 100.00
PUSHAQ          10     0.00 100.00
ASHP            10     0.00 100.00
CVTPS            9     0.00 100.00
BLEQU            8     0.00 100.00
SKPC             7     0.00 100.00
DIVL3            7     0.00 100.00
AOBLEQ           7     0.00 100.00
EDITPC           5     0.00 100.00
CASEW            5     0.00 100.00
CHMK             5     0.00 100.00
CVTLW            5     0.00 100.00
ROTL             4     0.00 100.00
BITL             4     0.00 100.00
BBCS             4     0.00 100.00
EXTV             4     0.00 100.00
CALLG            4     0.00 100.00
CMPC5            3     0.00 100.00
ADDF2            3     0.00 100.00
SUBF2            3     0.00 100.00
DIVF2            3     0.00 100.00
CVTWF            3     0.00 100.00
CVTLF            3     0.00 100.00
MOVF             3     0.00 100.00
EMUL             3     0.00 100.00
TSTB             3     0.00 100.00
CVTBW            3     0.00 100.00
MULW3            3     0.00 100.00
DIVL2            3     0.00 100.00
CVTSP            2     0.00 100.00
PUSHAW           2     0.00 100.00
CVTFD            2     0.00 100.00
MOVD             2     0.00 100.00
MULL3            2     0.00 100.00
BICL3            2     0.00 100.00
BBCC             2     0.00 100.00
INSQUE           1     0.00 100.00
DIVD2            1     0.00 100.00
CVTLD            1     0.00 100.00
CVTDF            1     0.00 100.00
```

```
BISB2          1    0.00 100.00
INCB           1    0.00 100.00
```

```
Instruction Size

Size       Count  Percnt Cumula

   1         213    0.00   0.00
   2     7865297   51.71  51.72
   3     2098478   13.80  65.51
   4     1049684    6.90  72.42
   5     3146043   20.69  93.10
   6     1048972    6.90 100.00
   7         218    0.00 100.00
   8          16    0.00 100.00
   9          18    0.00 100.00
  10           5    0.00 100.00
  11           1    0.00 100.00
  12           0    0.00 100.00
  13           0    0.00 100.00
  14           0    0.00 100.00
  15           0    0.00 100.00
  16           0    0.00 100.00
  17           0    0.00 100.00
  18           0    0.00 100.00
  19           0    0.00 100.00
  20           0    0.00 100.00
  21           0    0.00 100.00
  22           0    0.00 100.00
  23           0    0.00 100.00
  24           0    0.00 100.00
  25           0    0.00 100.00
  26           0    0.00 100.00
  27           0    0.00 100.00
  28           0    0.00 100.00
  29           0    0.00 100.00
  30           0    0.00 100.00
  31           0    0.00 100.00
  32           0    0.00 100.00
  33           0    0.00 100.00
  34           0    0.00 100.00
  35           0    0.00 100.00
  36           0    0.00 100.00
  37           0    0.00 100.00
  38           0    0.00 100.00
  39           0    0.00 100.00
  40           0    0.00 100.00

Average Instruction Size =  3.17
```

```
Specifier Size

Size        Count  Percnt Cumule

     1   19404530   82.22  82.22
     2       1655    0.01  82.23
     3    3145973   13.33  95.56
     4    1048650    4.44 100.00
     5        320    0.00 100.00
     6          0    0.00 100.00

Average Specifier Size =   1.40


Specifier Type (all)

Type        Count  Percnt Cumule

s^#0x     2098106    8.89   8.89
s^#1x         138    0.00   8.89
s^#2x         107    0.00   8.89
s^#3x          96    0.00   8.89
[Rx]      4194438   17.77  26.66
Rn       12585969   53.33  79.99
(Rb)          332    0.00  79.99
-(Rb)          95    0.00  79.99
(Rb)+         355    0.00  79.99
@(Rb)+         26    0.00  79.99
b^(Rb)       1144    0.00  80.00
@b(Rb)        274    0.00  80.00
w^(Rb)        121    0.00  80.00
@w(Rb)          0    0.00  80.00
1^(Rb)        169    0.00  80.00
@1(Rb)         33    0.00  80.00
Bdb       4719537   20.00 100.00
Bdw           189    0.00 100.00


Specifier Type (index)

Type        Count  Percnt Cumule

(Rb)            0    0.00   0.00
-(Rb)           0    0.00   0.00
(Rb)+           0    0.00   0.00
@(Rb)+          0    0.00   0.00
b^(Rb)    3145734   75.00  75.00
@b(Rb)         54    0.00  75.00
w^(Rb)    1048650   25.00 100.00
@w(Rb)          0    0.00 100.00
1^(Rb)          0    0.00 100.00
@1(Rb)          0    0.00 100.00
```

```
Memory Reads Per Instruction

Number      Count  Percnt Cumula

     0   13110426  86.20  86.20
     1    2098237  13.80 100.00
     2        279   0.00 100.00
     3          2   0.00 100.00
     4          1   0.00 100.00
     5          0   0.00 100.00
     6          0   0.00 100.00

Average Memory Reads Per Instruction  =  0.14


Memory Writes Per Instruction

Number      Count  Percnt Cumula

     0   13111080  86.21  86.21
     1    2097863  13.79 100.00
     2          2   0.00 100.00

Average Memory Writes Per Instruction  =  0.14


Register Reads Per Instruction

Number      Count  Percnt Cumula

     0    4720151  31.04  31.04
     1    5245065  34.49  65.52
     2    3146489  20.69  86.21
     3    2097208  13.79 100.00
     4         27   0.00 100.00
     5          5   0.00 100.00
     6          0   0.00 100.00
     7          0   0.00 100.00
     8          0   0.00 100.00
     9          0   0.00 100.00
    10          0   0.00 100.00
    11          0   0.00 100.00
    12          0   0.00 100.00

Average Register Reads Per Instruction  =  1.17


Register Writes Per Instruction

Number      Count  Percnt Cumula

     0    7867335  51.73  51.73
     1    7341600  48.27 100.00
     2         10   0.00 100.00

Average Register Writes Per Instruction  =  0.48
  CUTLER        Job terminated at 30-NOV-1984 17:59:59.51
```

```
Instruction Frequency Data
This data was collected on 30-NOV-1984 10:40:05.24
This data was written to dba3:[cutler]pjacobi.cod
Total number of instructions traced was  13567199

Name        Count  Percnt Cumula

MOVF      2411961  17.78  17.78
SUBL3     1443589  10.64  28.42
ADDL2     1435118  10.58  39.00
MULL2     1434538  10.57  49.57
MULF3     1324354   9.76  59.33
MULF2     1197322   8.83  68.16
SUBF3      589739   4.35  72.50
ADDF3      587551   4.33  76.83
CMPL       581551   4.29  81.12
BEQL       571072   4.21  85.33
ADDL3      323167   2.38  87.71
ACBL       315503   2.33  90.04
ADDF2      273944   2.02  92.06
AOBLEQ     207153   1.53  93.58
BLEQ        59895   0.44  94.02
CMPF        59776   0.44  94.46
BICW2       59738   0.44  94.91
DIVF3       59384   0.44  95.34
SUBW2       46019   0.34  95.68
ADDW2       36388   0.27  95.95
BICL3       34442   0.25  96.20
MOVL        33645   0.25  96.45
BLSS        31227   0.23  96.68
DIVF2       30974   0.23  96.91
CLRL        30383   0.22  97.13
BGEQ        30330   0.22  97.36
RSB         25325   0.19  97.54
JMP         24799   0.18  97.73
JSB         24449   0.18  97.91
MOVZWL      23259   0.17  98.08
ROTL        23217   0.17  98.25
CLRB        23081   0.17  98.42
TSTB        23061   0.17  98.59
CVTFD       23060   0.17  98.76
DIVD2       23004   0.17  98.93
BRW         22711   0.17  99.10
BRB         13157   0.10  99.19
SUBF2       12235   0.09  99.28
BGTR        11005   0.08  99.37
MOVAL        6916   0.05  99.42
MULL3        6289   0.05  99.46
SUBL2        6008   0.04  99.51
PUSHAL       5158   0.04  99.55
CVTLF        5084   0.04  99.58
MNEGF        4199   0.03  99.61
CMPW         3958   0.03  99.64
EMUL         3935   0.03  99.67
EDIV         3850   0.03  99.70
INCL         3699   0.03  99.73
MOVAB        3236   0.02  99.75
CVTWL        2550   0.02  99.77
BICL2        1679   0.01  99.78
RET          1603   0.01  99.79
CALLS        1598   0.01  99.81
```

```
CVTBL        1505   0.01   99.82
DECL         1496   0.01   99.83
MOVW         1488   0.01   99.84
SUBW3        1371   0.01   99.85
INCB         1303   0.01   99.86
BLSSU        1276   0.01   99.87
POLYF        1267   0.01   99.88
CVTDF        1266   0.01   99.89
MOVAQ        1266   0.01   99.90
EMODF        1265   0.01   99.91
ADDD2        1265   0.01   99.91
BICW3        1265   0.01   99.92
MOVB         1111   0.01   99.93
TSTL          816   0.01   99.94
CASEB         761   0.01   99.94
BNEQ          676   0.00   99.95
BSBW          671   0.00   99.95
PUSHL         590   0.00   99.96
MOVQ          388   0.00   99.96
INSV          356   0.00   99.96
BLBC          354   0.00   99.97
BBC           353   0.00   99.97
CMPB          331   0.00   99.97
PUSHAB        296   0.00   99.97
BISL2         265   0.00   99.98
ASHL          262   0.00   99.98
MOVC3         220   0.00   99.98
BISPSW        199   0.00   99.98
CVTLP         195   0.00   99.98
CVTPS         185   0.00   99.98
BBCC          167   0.00   99.98
BBS           150   0.00   99.99
SOBGEQ        127   0.00   99.99
DIVL3         124   0.00   99.99
TSTF          121   0.00   99.99
CVTFL         110   0.00   99.99
BBSC          101   0.00   99.99
CLRQ           88   0.00   99.99
MOVZBL         87   0.00   99.99
MNEGL          87   0.00   99.99
SKPC           68   0.00   99.99
EXTZV          66   0.00   99.99
DIVL2          65   0.00   99.99
MOVD           57   0.00   99.99
BVC            55   0.00   99.99
ADDB2          55   0.00   99.99
MULB2          55   0.00  100.00
ACBB           54   0.00  100.00
CLRW           45   0.00  100.00
BSBB           42   0.00  100.00
MOVC5          41   0.00  100.00
BLBS           37   0.00  100.00
SOBGTR         37   0.00  100.00
XORW3          36   0.00  100.00
BLEQU          28   0.00  100.00
CHME           26   0.00  100.00
ASHP           24   0.00  100.00
DECW           23   0.00  100.00
PUSHR          23   0.00  100.00
```

```
MCOMB        22     0.00 100.00
POPR         22     0.00 100.00
AOBLSS       22     0.00 100.00
LOCC         20     0.00 100.00
BICB3        20     0.00 100.00
TSTW         17     0.00 100.00
XORB3        16     0.00 100.00
EDITPC       12     0.00 100.00
SUBB3        11     0.00 100.00
EXTV         11     0.00 100.00
PUSHAQ       10     0.00 100.00
CALLG        10     0.00 100.00
CHMK          9     0.00 100.00
BBCS          9     0.00 100.00
CASEW         5     0.00 100.00
CVTLW         5     0.00 100.00
BITL          4     0.00 100.00
CMPC5         3     0.00 100.00
CVTWF         3     0.00 100.00
CVTBW         3     0.00 100.00
MULW3         3     0.00 100.00
CVTSP         2     0.00 100.00
PUSHAW        2     0.00 100.00
BITB          2     0.00 100.00
MOVPSL        2     0.00 100.00
REI           1     0.00 100.00
INSQUE        1     0.00 100.00
CVTLD         1     0.00 100.00
BISB2         1     0.00 100.00
BICB2         1     0.00 100.00
```

```
Instruction Size

Size        Count  Percnt Cumula

   1        26929    0.20   0.20
   2       795271    5.86   6.06
   3      4750848   35.02  41.08
   4      3023492   22.29  63.36
   5      2720383   20.05  83.41
   6      1819204   13.41  96.82
   7       368493    2.72  99.54
   8         3640    0.03  99.57
   9        27021    0.20  99.76
  10        29467    0.22  99.98
  11         2451    0.02 100.00
  12            0    0.00 100.00
  13            0    0.00 100.00
  14            0    0.00 100.00
  15            0    0.00 100.00
  16            0    0.00 100.00
  17            0    0.00 100.00
  18            0    0.00 100.00
  19            0    0.00 100.00
  20            0    0.00 100.00
  21            0    0.00 100.00
  22            0    0.00 100.00
  23            0    0.00 100.00
  24            0    0.00 100.00
  25            0    0.00 100.00
  26            0    0.00 100.00
  27            0    0.00 100.00
  28            0    0.00 100.00
  29            0    0.00 100.00
  30            0    0.00 100.00
  31            0    0.00 100.00
  32            0    0.00 100.00
  33            0    0.00 100.00
  34            0    0.00 100.00
  35            0    0.00 100.00
  36            0    0.00 100.00
  37            0    0.00 100.00
  38            0    0.00 100.00
  39            0    0.00 100.00
  40            0    0.00 100.00

Average Instruction Size =   4.10
```

```
Specifier Size

Size        Count  Percnt Cumula

   1     24311236  77.29   77.29
   2      3963980  12.60   89.89
   3      3012911   9.58   99.47
   4        11399   0.04   99.50
   5       156449   0.50  100.00
   6            3   0.00  100.00

Average Specifier Size =    1.34


Specifier Type (all)

Type        Count  Percnt Cumula

s^#0x     1838020   5.84    5.84
s^#1x       24121   0.08    5.92
s^#2x        3144   0.01    5.93
s^#3x        5297   0.02    5.95
[Rx]      2909389   9.25   15.20
Rn       19914430  63.31   78.50
(Rb)      1274775   4.05   82.56
-(Rb)         536   0.00   82.56
(Rb)+      531060   1.69   84.25
@(Rb)+         44   0.00   84.25
b^(Rb)    3586808  11.40   95.65
@b(Rb)      43393   0.14   95.79
w^(Rb)       6592   0.02   95.81
@w(Rb)          0   0.00   95.81
1^(Rb)      27611   0.09   95.90
@1(Rb)      24547   0.08   95.97
Bdb        927273   2.95   98.92
Bdw        338939   1.08  100.00


Specifier Type (index)

Type        Count  Percnt Cumula

(Rb)        12703   0.44    0.44
-(Rb)           0   0.00    0.44
(Rb)+           0   0.00    0.44
@(Rb)+          1   0.00    0.44
b^(Rb)    2885231  99.17   99.61
@b(Rb)         55   0.00   99.61
w^(Rb)      11399   0.39  100.00
@w(Rb)          0   0.00  100.00
1^(Rb)          0   0.00  100.00
@1(Rb)          0   0.00  100.00
```

```
Memory Reads Per Instruction

Number     Count  Percnt Cumula

     0   7884891  58.12  58.12
     1   5364767  39.54  97.66
     2    316273   2.33  99.99
     3         2   0.00  99.99
     4      1266   0.01 100.00
     5         0   0.00 100.00
     6         0   0.00 100.00

Average Memory Reads Per Instruction  =  0.44


Memory Writes Per Instruction

Number     Count  Percnt Cumula

     0  10987263  80.98  80.98
     1   2579922  19.02 100.00
     2        14   0.00 100.00

Average Memory Writes Per Instruction  =  0.19


Register Reads Per Instruction

Number     Count  Percnt Cumula

     0    881304   6.50   6.50
     1   1895504  13.97  20.47
     2   8486543  62.55  83.02
     3   1706805  12.58  95.60
     4    597031   4.40 100.00
     5        12   0.00 100.00
     6         0   0.00 100.00
     7         0   0.00 100.00
     8         0   0.00 100.00
     9         0   0.00 100.00
    10         0   0.00 100.00
    11         0   0.00 100.00
    12         0   0.00 100.00

Average Register Reads Per Instruction  =  1.94


Register Writes Per Instruction

Number     Count  Percnt Cumula

     0   4080169  30.07  30.07
     1   9481929  69.89  99.96
     2      5101   0.04 100.00

Average Register Writes Per Instruction  =  0.70
  CUTLER        job terminated at 30-NOV-1984 19:13:03.84
```

FRIGATE
DATA

```
Instruction Frequency Data
This data was collected on  6-DEC-1984 22:11:37.56
This data was written to dual:[cutler]icp.cod
Total number of instructions traced was     9957421

Name        Count  Percnt Cumula

MOVL      1743507  17.51  17.51
BEQL       567772   5.70  23.21
BNEQ       485003   4.87  28.08
CMPW       441890   4.44  32.52
MOVZBL     333663   3.35  35.87
MOVAL      303766   3.05  38.92
CMPL       292282   2.94  41.86
CVTWL      265602   2.67  44.52
CLRL       212242   2.13  46.66
RSB        211168   2.12  48.78
MOVW       189319   1.90  50.68
BBC        181495   1.82  52.50
ADDL2      170716   1.71  54.22
BGTR       169925   1.71  55.92
BGEQ       165458   1.66  57.58
MOVAB      155238   1.56  59.14
CVTBL      152638   1.53  60.68
CMPB       138147   1.39  62.06
BRW        136873   1.37  63.44
BBS        136438   1.37  64.81
PUSHL      133715   1.34  66.15
BRB        127014   1.28  67.43
TSTL       117191   1.18  68.60
ADDL3      113561   1.14  69.74
BSBB       111727   1.12  70.87
AOBLSS     110415   1.11  71.97
MOVZBW     109056   1.10  73.07
BLSS       108715   1.09  74.16
MOVZWL     102328   1.03  75.19
BSBW        98629   0.99  76.18
MOVB        97518   0.98  77.16
BLEQ        94588   0.95  78.11
RET         94044   0.94  79.05
CASEB       93993   0.94  80.00
CALLS       91473   0.92  80.92
MULL3       87402   0.88  81.79
BLSSU       84698   0.85  82.64
BBCC        82935   0.83  83.48
BLBC        77627   0.78  84.26
INCL        75126   0.75  85.01
INSV        73498   0.74  85.75
AOBLEQ      70124   0.70  86.45
SUBL3       69871   0.70  87.15
SUBB3       69752   0.70  87.86
SOBGTR      63175   0.63  88.49
JMP         60888   0.61  89.10
TSTW        59291   0.60  89.70
SUBL2       56405   0.57  90.26
SUBB2       55494   0.56  90.82
EXTZV       55098   0.55  91.37
MNEGW       51944   0.52  91.90
CVTLW       43892   0.44  92.34
```

```
ACBL      42299   0.42  92.76
DECL      39371   0.40  93.16
MULL2     34942   0.35  93.51
JSB       26880   0.27  93.78
CLRB      26760   0.27  94.05
MOVAW     26677   0.27  94.31
BGTRU     22420   0.23  94.54
BICL2     20556   0.21  94.75
MOVC3     20555   0.21  94.95
ASHL      20362   0.20  95.16
SOBGEQ    18927   0.19  95.35
REMQUE    18727   0.19  95.53
MNEGL     18644   0.19  95.72
CVTBW     18097   0.18  95.90
DECW      18051   0.18  96.09
BLBS      17849   0.18  96.26
DIVL2     17263   0.17  96.44
MOVQ      16936   0.17  96.61
CLRW      16935   0.17  96.78
MCOML     16096   0.16  96.94
INSQUE    15393   0.15  97.09
MOVAQ     14413   0.14  97.24
LOCC      13656   0.14  97.38
BISB2     13335   0.13  97.51
POPR      13198   0.13  97.64
MOVC5     12444   0.12  97.77
INCW      12305   0.12  97.89
PUSHAL    12113   0.12  98.01
CMPZV     11031   0.11  98.12
BVS       11013   0.11  98.23
CLRQ      10822   0.11  98.34
ADDW2      9541   0.10  98.44
TSTB       9434   0.09  98.53
ADDW3      8999   0.09  98.62
PUSHR      8927   0.09  98.71
PUSHAB     8601   0.09  98.80
BGEQU      7313   0.07  98.87
EDIV       7244   0.07  98.95
SPANC      6780   0.07  99.01
BICW2      6528   0.07  99.08
BISL2      6282   0.06  99.14
EMUL       5916   0.06  99.20
CMPC5      5257   0.05  99.25
ADDB3      4891   0.05  99.30
BVC        4834   0.05  99.35
CASEW      4641   0.05  99.40
SUBW3      4418   0.04  99.44
PROBER     4261   0.04  99.49
BICB3      3848   0.04  99.52
BICB2      3791   0.04  99.56
CMPC3      3568   0.04  99.60
SUBW2      3394   0.03  99.63
ACBB       3258   0.03  99.67
CHME       2928   0.03  99.69
PUSHAQ     2852   0.03  99.72
EXTV       2834   0.03  99.75
BICW3      2757   0.03  99.78
MNEGB      2591   0.03  99.81
CALLG      2574   0.03  99.83
BLEQU      2466   0.02  99.86
```

```
BISW2         2149    0.02   99.88
BISB3         1989    0.02   99.90
MOVTC         1494    0.02   99.91
PUSHAW        1418    0.01   99.93
BISL3         1394    0.01   99.94
BICL3          944    0.01   99.95
BBSS           580    0.01   99.96
INCB           550    0.01   99.96
CVTLB          513    0.01   99.97
FFS            508    0.01   99.97
SKPC           450    0.00   99.98
CHMK           368    0.00   99.98
XORB2          358    0.00   99.98
MCOMB          254    0.00   99.99
DIVL3          191    0.00   99.99
ADDB2          176    0.00   99.99
DIVW2          156    0.00   99.99
BISW3          144    0.00   99.99
MCOMW          117    0.00   99.99
DECB           116    0.00  100.00
BITW            65    0.00  100.00
SUBP6           51    0.00  100.00
CVTPL           51    0.00  100.00
BITB            50    0.00  100.00
BBSC            36    0.00  100.00
CVTLD           26    0.00  100.00
MULW3           26    0.00  100.00
CASEL           25    0.00  100.00
CMPD            24    0.00  100.00
MOVD            14    0.00  100.00
BITL            14    0.00  100.00
CVTWB            9    0.00  100.00
MULD3            8    0.00  100.00
SUBD3            6    0.00  100.00
TSTD             6    0.00  100.00
DIVD2            3    0.00  100.00
CVTDL            3    0.00  100.00
CVTDF            3    0.00  100.00
ACBW             1    0.00  100.00
CVTLF            1    0.00  100.00
DIVB3            1    0.00  100.00
XORW2            1    0.00  100.00
XORL3            1    0.00  100.00
```

```
Instruction Size

Size        Count  Percnt Cumula

   1       305212    3.07   3.07
   2      2440090   24.51  27.57
   3      2194528   22.04  49.61
   4      2142629   21.52  71.13
   5      1418012   14.24  85.37
   6       523898    5.26  90.63
   7       424738    4.27  94.90
   8       411772    4.14  99.03
   9        38384    0.39  99.42
  10        52167    0.52  99.94
  11         3402    0.03  99.97
  12         2417    0.02 100.00
  13           32    0.00 100.00
  14           67    0.00 100.00
  15            0    0.00 100.00
  16           73    0.00 100.00
  17            0    0.00 100.00
  18            0    0.00 100.00
  19            0    0.00 100.00
  20            0    0.00 100.00
  21            0    0.00 100.00
  22            0    0.00 100.00
  23            0    0.00 100.00
  24            0    0.00 100.00
  25            0    0.00 100.00
  26            0    0.00 100.00
  27            0    0.00 100.00
  28            0    0.00 100.00
  29            0    0.00 100.00
  30            0    0.00 100.00
  31            0    0.00 100.00
  32            0    0.00 100.00
  33            0    0.00 100.00
  34            0    0.00 100.00
  35            0    0.00 100.00
  36            0    0.00 100.00
  37            0    0.00 100.00
  38            0    0.00 100.00
  39            0    0.00 100.00
  40            0    0.00 100.00

Average Instruction Size =   3.79
```

Specifier Size

Size	Count	Percnt	Cumula
1	12195247	68.41	68.41
2	3066385	17.20	85.61
3	1559437	8.75	94.36
4	331437	1.86	96.22
5	447160	2.51	98.73
6	226171	1.27	100.00

Average Specifier Size = 1.57

Specifier Type (all)

Type	Count	Percnt	Cumula
s^#0x	1475656	8.28	8.28
s^#1x	143289	0.80	9.08
s^#2x	69521	0.39	9.47
s^#3x	77850	0.44	9.91
[Rx]	1282570	7.19	17.10
Rn	6723786	37.72	54.82
(Rb)	467798	2.62	57.45
-(Rb)	80794	0.45	57.90
(Rb)+	991686	5.56	63.46
@(Rb)+	12854	0.07	63.54
b^(Rb)	2107775	11.82	75.36
@b(Rb)	132002	0.74	76.10
w^(Rb)	910439	5.11	81.21
@w(Rb)	23978	0.13	81.34
l^(Rb)	284528	1.60	82.94
@l(Rb)	37754	0.21	83.15
Bdb	2722547	15.27	98.42
Bdw	281060	1.58	100.00

Specifier Type (index)

Type	Count	Percnt	Cumula
(Rb)	265422	20.69	20.69
-(Rb)	0	0.00	20.69
(Rb)+	0	0.00	20.69
@(Rb)+	0	0.00	20.69
b^(Rb)	463799	36.16	56.86
@b(Rb)	49170	3.83	60.69
w^(Rb)	276170	21.53	82.22
@w(Rb)	5860	0.46	82.68
l^(Rb)	214441	16.72	99.40
@l(Rb)	7708	0.60	100.00

```
Memory Reads Per Instruction

Number      Count  Percnt Cumula

     0    6087915   61.14  61.14
     1    3574652   35.90  97.04
     2     289517    2.91  99.95
     3       5333    0.05 100.00
     4          4    0.00 100.00
     5          0    0.00 100.00
     6          0    0.00 100.00

Average Memory Reads Per Instruction  =  0.42


Memory Writes Per Instruction

Number      Count  Percnt Cumula

     0    8640952   86.78  86.78
     1    1316469   13.22 100.00
     2          0    0.00 100.00

Average Memory Writes Per Instruction  =  0.13
```

SPECTRUM RASA only

```
Register Reads Per Instruction

Number      Count  Percnt Cumula

     0    3077047   30.90  30.90
     1    4655170   46.75  77.65
     2    1790713   17.98  95.64
     3     319066    3.20  98.84
     4     111307    1.12  99.96
     5       4118    0.04 100.00
     6          0    0.00 100.00
     7          0    0.00 100.00
     8          0    0.00 100.00
     9          0    0.00 100.00
    10          0    0.00 100.00
    11          0    0.00 100.00
    12          0    0.00 100.00

Average Register Reads Per Instruction  =  0.97


Register Writes Per Instruction

Number      Count  Percnt Cumula

     0    6008515   60.34  60.34
     1    3941662   39.59  99.93
     2       7244    0.07 100.00

Average Register Writes Per Instruction  =  0.40
```

```
Specifier Access Type

Type       Count  Percnt Cumula

read     8125967  45.59  45.59
write    4417811  24.78  70.37
modify    854808   4.80  75.16
addres    879241   4.93  80.10
vield     544453   3.05  83.15
branch   3003607  16.85 100.00

Total number of operand specifiers was  17825887
Number of nonfetch operand specifiers was  13122708
Percent of nonfetch operand specifiers was  73.62
```

```
Instruction Frequency Data
This data was collected on 30-NOV-1984 08:53:51.46
This data was written to dba3:[cutler]icp.cod
Total number of instructions traced was    9948733

Name       Count  Percnt Cumula

MOVL     1739602  17.49  17.49
BEQL      564686   5.68  23.16
BNEQ      484989   4.87  28.04
CMPW      441337   4.44  32.47
MOVZBL    333848   3.36  35.83
MOVAL     303765   3.05  38.88
CMPL      292195   2.94  41.82
CVTWL     266086   2.67  44.49
RSB       211140   2.12  46.62
CLRL      210742   2.12  48.73
MOVW      189294   1.90  50.64
BBC       180912   1.82  52.45
ADDL2     171129   1.72  54.17
BGTR      169916   1.71  55.88
BGEQ      165458   1.66  57.55
MOVAB     156565   1.57  59.12
CVTBL     152638   1.53  60.65
CMPB      138236   1.39  62.04
BRW       137319   1.38  63.42
BBS       137010   1.38  64.80
PUSHL     133535   1.34  66.14
BRB       127044   1.28  67.42
TSTL      115383   1.16  68.58
ADDL3     113320   1.14  69.72
BSBB      111727   1.12  70.84
AOBLSS    110415   1.11  71.95
MOVZBW    109055   1.10  73.05
BLSS      108707   1.09  74.14
MOVZWL    102036   1.03  75.17
BSBW       98630   0.99  76.16
MOVB       97676   0.98  77.14
BLEQ       94587   0.95  78.09
CASEB      93969   0.94  79.03
RET        93894   0.94  79.98
CALLS      91323   0.92  80.90
MULL3      87398   0.88  81.77
BLSSU      84846   0.85  82.63
BBCC       82935   0.83  83.46
BLBC       77592   0.78  84.24
INCL       75121   0.76  85.00
INSV       73498   0.74  85.74
AOBLEQ     70124   0.70  86.44
SUBL3      69846   0.70  87.14
SUBB3      69754   0.70  87.84
SOBGTR     63175   0.64  88.48
JMP        60856   0.61  89.09
TSTW       59545   0.60  89.69
SUBB2      55494   0.56  90.25
EXTZV      55096   0.55  90.80
SUBL2      55020   0.55  91.35
MNEGW      51944   0.52  91.88
CVTLW      43892   0.44  92.32
```

```
ACBL       42299    0.43   92.74
DECL       39369    0.40   93.14
MULL2      34940    0.35   93.49
JSB        26851    0.27   93.76
CLRB       26760    0.27   94.03
MOVAW      26677    0.27   94.30
BGTRU      22422    0.23   94.52
BICL2      20556    0.21   94.73
MOVC3      20548    0.21   94.93
ASHL       20358    0.20   95.14
SOBGEQ     19115    0.19   95.33
REMQUE     18727    0.19   95.52
MNEGL      18643    0.19   95.71
MOVQ       18348    0.18   95.89
DECW       18051    0.18   96.07
CVTBW      17952    0.18   96.25
BLBS       17756    0.18   96.43
DIVL2      17263    0.17   96.60
CLRW       16926    0.17   96.77
MCOML      16096    0.16   96.94
INSQUE     15393    0.15   97.09
MOVAQ      14413    0.14   97.24
LOCC       13632    0.14   97.37
BISB2      13296    0.13   97.51
POPR       13197    0.13   97.64
MOVC5      12431    0.12   97.76
INCW       12305    0.12   97.89
PUSHAL     12089    0.12   98.01
CMPZV      11026    0.11   98.12
BVS        11013    0.11   98.23
CLRQ       10815    0.11   98.34
TSTB        9670    0.10   98.44
ADDW2       9541    0.10   98.53
ADDW3       8999    0.09   98.62
PUSHR       8927    0.09   98.71
PUSHAB      8452    0.08   98.80
BGEQU       7264    0.07   98.87
EDIV        7245    0.07   98.94
SPANC       6780    0.07   99.01
BICW2       6528    0.07   99.08
BISL2       6282    0.06   99.14
EMUL        5917    0.06   99.20
CMPC5       5488    0.06   99.26
ADDB3       4889    0.05   99.30
BVC         4834    0.05   99.35
CASEW       4641    0.05   99.40
SUBW3       4418    0.04   99.44
PROBER      4261    0.04   99.49
BICB3       3848    0.04   99.53
BICB2       3751    0.04   99.56
CMPC3       3568    0.04   99.60
SUBW2       3394    0.03   99.63
ACBB        3258    0.03   99.67
CHME        2928    0.03   99.70
PUSHAQ      2852    0.03   99.72
EXTV        2834    0.03   99.75
BICW3       2757    0.03   99.78
MNEGB       2591    0.03   99.81
CALLG       2574    0.03   99.83
```

```
BLEQU      2464    0.02   99.86
BISW2      2149    0.02   99.88
BISB3      1989    0.02   99.90
MOVTC      1470    0.01   99.91
PUSHAW     1418    0.01   99.93
BISL3      1394    0.01   99.94
BICL3       941    0.01   99.95
BBSS        580    0.01   99.96
INCB        551    0.01   99.96
CVTLB       513    0.01   99.97
FFS         508    0.01   99.97
SKPC        448    0.00   99.98
XORB2       358    0.00   99.98
CHMK        325    0.00   99.98
MCOMB       254    0.00   99.99
DIVL3       191    0.00   99.99
ADDB2       176    0.00   99.99
DIVW2       156    0.00   99.99
BISW3       144    0.00   99.99
MCOMW       117    0.00   99.99
DECB        116    0.00  100.00
BITW         65    0.00  100.00
SUBP6        51    0.00  100.00
CVTPL        51    0.00  100.00
BBSC         36    0.00  100.00
BITB         34    0.00  100.00
CVTLD        26    0.00  100.00
MULW3        26    0.00  100.00
CASEL        25    0.00  100.00
CMPD         24    0.00  100.00
MOVD         14    0.00  100.00
BITL         14    0.00  100.00
CVTWB         9    0.00  100.00
MULD3         8    0.00  100.00
SUBD3         6    0.00  100.00
TSTD          6    0.00  100.00
DIVD2         3    0.00  100.00
CVTDL         3    0.00  100.00
CVTDF         3    0.00  100.00
ACBW          1    0.00  100.00
CVTLF         1    0.00  100.00
DIVB3         1    0.00  100.00
XORW2         1    0.00  100.00
XORL3         1    0.00  100.00
```

```
Instruction Size

Size        Count  Percnt Cumula

    1      305034    3.07   3,07
    2     2434230   24.47  27,53
    3     2191878   22.03  49,57
    4     2142094   21.53  71,10
    5     1419257   14.27  85,36
    6      523420    5.26  90,62
    7      424538    4.27  94,89
    8      411787    4.14  99,03
    9       38381    0.39  99,42
   10       52167    0.52  99,94
   11        3378    0.03  99,97
   12        2397    0.02 100,00
   13          32    0.00 100,00
   14          67    0.00 100,00
   15           0    0.00 100,00
   16          73    0.00 100,00
   17           0    0.00 100,00
   18           0    0.00 100,00
   19           0    0.00 100,00
   20           0    0.00 100,00
   21           0    0.00 100,00
   22           0    0.00 100,00
   23           0    0.00 100,00
   24           0    0.00 100,00
   25           0    0.00 100,00
   26           0    0.00 100,00
   27           0    0.00 100,00
   28           0    0.00 100,00
   29           0    0.00 100,00
   30           0    0.00 100,00
   31           0    0.00 100,00
   32           0    0.00 100,00
   33           0    0.00 100,00
   34           0    0.00 100,00
   35           0    0.00 100,00
   36           0    0.00 100,00
   37           0    0.00 100,00
   38           0    0.00 100,00
   39           0    0.00 100,00
   40           0    0.00 100,00

Average Instruction Size =  3,80
```

```
Specifier Size

Size       Count  Percent Cumula

     1  12184479   68.39   68.39
     2   3065682   17.21   85.60
     3   1560462    8.76   94.36
     4    331484    1.86   96.22
     5    446636    2.51   98.73
     6    226167    1.27  100.00

Average Specifier Size =   1.57


Specifier Type (all)

Type       Count  Percent Cumula

s^#0x    1473957    8.27    8.27
s^#1x     143101    0.80    9.08
s^#2x      69491    0.39    9.47
s^#3x      77848    0.44    9.90
[Rx]     1282513    7.20   17.10
Rn       6717369   37.71   54.81
(Rb)      467911    2.63   57.44
-(Rb)      80795    0.45   57.89
(Rb)+     991926    5.57   63.46
@(Rb)+     12628    0.07   63.53
b^(Rb)   2106409   11.82   75.35
@b(Rb)    132097    0.74   76.09
w^(Rb)    911709    5.12   81.21
@w(Rb)     23978    0.13   81.35
l^(Rb)    284337    1.60   82.94
@l(Rb)     37727    0.21   83.15
Bdb      2719607   15.27   98.42
Bdw       281507    1.58  100.00


Specifier Type (index)

Type       Count  Percent Cumula

(Rb)      265398   20.69   20.69
-(Rb)          0    0.00   20.69
(Rb)+          0    0.00   20.69
@(Rb)+         0    0.00   20.69
b^(Rb)    463773   36.16   56.85
@b(Rb)     49170    3.83   60.69
w^(Rb)    276167   21.53   82.22
@w(Rb)      5860    0.46   82.68
l^(Rb)    214441   16.72   99.40
@l(Rb)      7704    0.60  100.00
```

```
. . . . .

Memory Reads Per Instruction

Number     Count  Percnt Cumula

     0   6080177   61.12  61.12
     1   3573927   35.92  97.04
     2    289359    2.91  99.95
     3      5266    0.05 100.00
     4         4    0.00 100.00
     5         0    0.00 100.00
     6         0    0.00 100.00

Average Memory Reads Per Instruction  =  0.42


Memory Writes Per Instruction

Number     Count  Percnt Cumula

     0   8632443   86.77  86.77
     1   1316290   13.23 100.00
     2         0    0.00 100.00

Average Memory Writes Per Instruction  =  0.13


Register Reads Per Instruction

Number     Count  Percnt Cumula

     0   3072137   30.88  30.88
     1   4651471   46.75  77.63
     2   1790468   18.00  95.63
     3    319025    3.21  98.84
     4    111514    1.12  99.96
     5      4118    0.04 100.00
     6         0    0.00 100.00
     7         0    0.00 100.00
     8         0    0.00 100.00
     9         0    0.00 100.00
    10         0    0.00 100.00
    11         0    0.00 100.00
    12         0    0.00 100.00

Average Register Reads Per Instruction  =  0.97


Register Writes Per Instruction

Number     Count  Percnt Cumula

     0   6003116   60.34  60.34
     1   3938372   39.59  99.93
     2      7245    0.07 100.00

Average Register Writes Per Instruction  =  0.40
  CUTLER        job terminated at 30-NOV-1984 14:50:05.31
```