ASSEMBLER FOR SIMULATION OF THE NEW RICE COMPUTER

-AP1/R2-

by

E. O. Mutschler
Computer Project
Rice University
Houston, Texas

Report # ORO-2572-21
AEC Contract AT-(40-1)-2572
June, 1969

ABSTRACT

One of the major problems of designing and building a new computer system is programming and debugging an operating system. The difficulty may be allocated somewhat by the use of a program which simulates, at some level, the computer under construction. One would like to be able to code routines that he writes symbolically to simplify coding, and to maintain routines in symbolic form to guard against differences between simulated machine language and actual machine language.

A hardware and software development effort is being carried on at Rice University on a new computer system which is based on the concept of tags for description and control and non-linear storage use. A simulator is being used to simulate the operation of the computer at a level somewhat above the microcode (the microcode in some respects having been designed from the logic of the simulator). An assembler has been written for symbolic input to the simulator, thus allowing the above-mentioned convenience of symbolic coding.

This assembly system: its relation to the simulator and, eventually, to the new hardware, is the subject of this paper. Topics covered include the criteria which caused the assembly language to assume its present form, the basic features of the language, and certain special features that have been found to be convenient, necessary, or aesthically pleasing. A section is included which concerns the changes in the language which will accompany its transferral to the new computer.

ASSEMBLER FOR SIMULATION OF THE NEW RICE COMPUTER
-AP1/R2-

INTRODUCTION

Work is currently being conducted on the construction of a new computer at Rice University. In conjuction with this hardware development, a software development effort is being carried out, in an effort to have an operating system of some nature available for use when the hardware becomes operational.

The means chosen to effect this development has been to simulate the basic functions of the proposed machine with a set of programs run on the present Rice Computer. To make the job of writing the routines in the system easier, the organization of the elements of the new computer has been rearranged somewhat, due primarily to the fact that any routines intended to be transferred to the new computer should be coded in symbolic form to be translated into either of the target languages: simulator instruction format, or actual new computer format.

To implement this aim, as well as to provide ease of coding system routines, an assembler has been written to run on the present Rice Computer which translates symbolic programs and loads them into an area of core store reserved for the "memory" of the new computer. It is envisioned that this assembler can be easily recoded into the assembly language of the new computer.

For the purposes of discussion, let us refer to the new computer as R2, and the assembly language as AP1/R2.

DESIGN CONSIDERATIONS

The development of the language has been more or less a multistage process. The first level of the development process consisted of defining a language with the following requirements: (1) It should be simple to implement. (2) It should allow fast translation, since a program retained only in symbolic form would have to be retranslated many times in simulation work. (3) It should be a 1-1 system, *i.e.* a symbolic assembly order should be a single machine order. The purpose of this is to represent to the programmer the machine exactly as it will be. In this way, it is hoped, weaknesses in the initial specification of the machine will become evident soon enough to be corrected in the hardware. This feature in the assembly system has led to several changes in machine specification, in fact.

Thinking on another level was that the language defined for simulation use would be a subset of a richer assembly language in R2 itself. In particular, it was thought that eventually a large portion of John Iliffe's Basic Language would be implemented here, due to the similarities between R2 and the definition of the Basic Machine.[1]

On a practical level, it was desired to eliminate some of the more undesirable features of the assembly system on the present computer. For example, one of the characteristics of the present assembly system is that undefined variables were stored as null words at the end of assembled programs. This was done without comment by the assembly system. Hence, any typographical or programming errors resulted in mysterious transfers to nonexistent instructions and references to nonexistent variables. It was proposed, therefore, that

undefined variables cause flags or error messages to be issued on the program listing. Another inconvenient feature was the fixed format of output. The programmer is forced to receive the output code listing and a copy of the symbol table, even though perhaps he is merely retranslating due to an unreadable output tape, etc. It was proposed, therefore, that all portions of the output process be conditioned on sense lights set by the programmer. Equivalent length programs in APl and APl/R2 would take 90 and 19 seconds, respectively, if all options in APl/R2 are exercised.

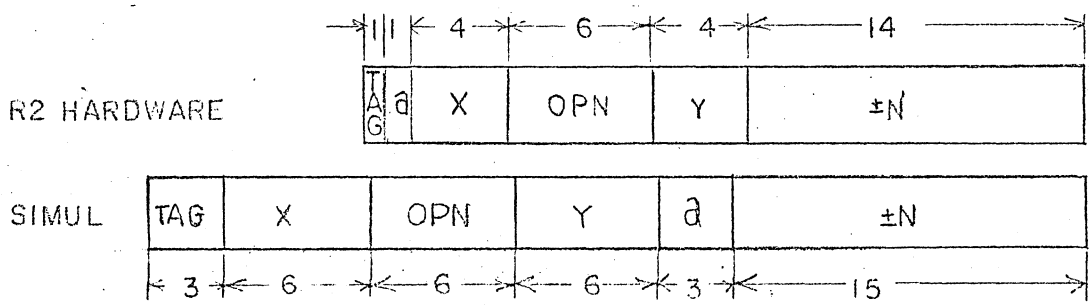
Some features were written into the new assembler with an eye toward increasing convenience for the programmer. One of these features is the option to translate while listing. This is feasible due to the 300 lpm. speed of our line printer when listing in a full alphanumeric character set. For short lines, the translation time "disappears" into the listing time. For long lines -- such as remarks, the gain is abrogated by the necessity of setting up a print matrix by software, and by the necessity of the assembly program to scan through the text, maintaining an upper/lower case indicator.

Another feature added for programmer convenience is the addition of a cross reference dictionary generating routine to the assembly system. At present, references only to user created variables are recorded, but it would not be difficult to add the recording of references to registers, which would aid greatly in charting the flow of information through registers and private storage locations.

THE HARDWARE AND THE SIMULATOR

In order to discuss the workings of the assembly system, we must say something about the proposed hardware and the program which simulates it. The hardware proposed is outlined in "Rice Computer-2 General Specifications", AEC Report # ORO-2572-19. A few relevant details are given here for reference.

In R2, computer instructions will be stored 2 to a word, each instruction being 30 bits in length. Since the arithmetic word of the present machine is only 54 bits, it was necessary for the simulator to operate on an instruction filling 39 bits of a word in the present memory. The extra nine bits are due to the fact that the R2 instruction is somewhat loosely packed in the word. The arrangement should be evident from the format illustrated here:



The tag bit in the instruction causes a simulated interrupt to occur, if the mode of the machine permits. Generally, this tag is used for tracing the instruction.

The X field is usually one of two items. It may be the designator for one of the sixteen general registers of the machine (Symbolically X0,X1,...,X15), or it may be an inflection on the operation code in the cases where the accumulator (X1) is assumed by the hardware to be the address of the first operand. This field, in the general case,

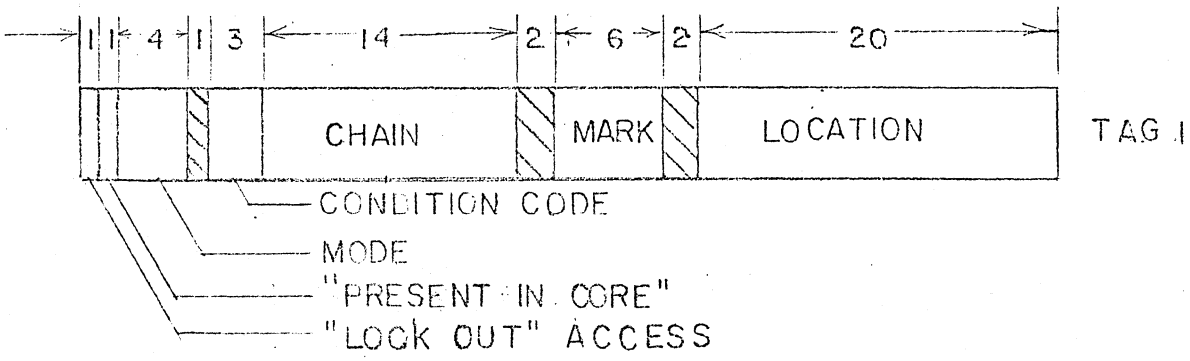
determines the destination of the result of the operation.

The OPN field contains the 6-bit operation code of the instruction, which is extended to 8 bits in certain classes of operations, particularly arithmetic orders. This is done by taking two bits of the X field for the operation code.

The Y and the ±N fields are the second operand designators. The Y field contains the designator for a register in the cases where this is meaningful. The ±N field determines an arithmetic value by which to displace the value of the register. If the value of ±N is -0 (octal 37777) then the content of Y is used, regardless of tag. If the "a" bit is on, ± N becomes a constant value inflected by the contents of the Y field: if Y=0 the value of ±N is used. If Y=1, ±N is an absolute memory location. If Y=2, ±N is a displacement relative to the instruction, in half-words.

In addition to instructions, Numeric, Control, and Address words are defined. The complete form of these elements may be found in the document on specifications for the new machine. The form currently used in the simulator is given for each here.

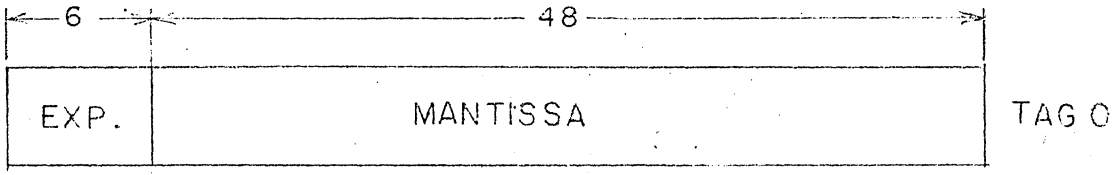
The Control element defines a set of instructions (and invariant data structures associated with these instructions) has the form:



CONTROL ELEMENT

Since these elements currently may not be directly generated by the programmer, and are created only by the Assembler and System when a program is placed in memory, and by certain types of control jumps, it will not be further discussed here.

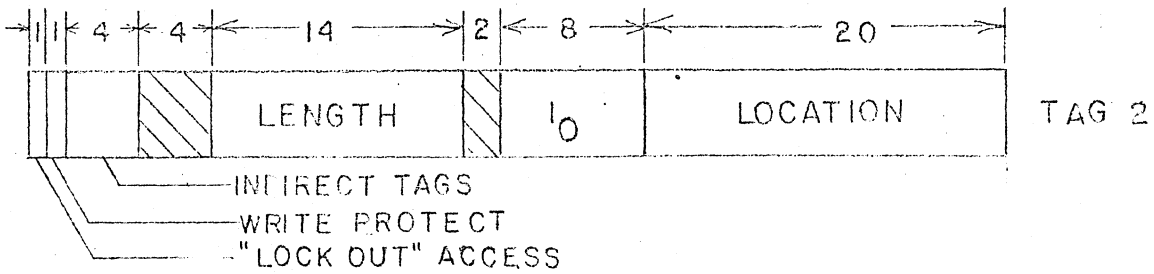
A numeric element is a standard data word, with special hardware tags in R2 to distinguish it as such. In the simulator, however, it has the following format:



NUMERIC ELEMENT

This is just a standard word on the present machine. Again, though other bits are defined for the hardware, they are not in the simulator, and hence not in the Assembler.

An Address element is much like the Rice Computer codeword or the Burroughs Descriptor. In this case, it may be relativised, as compared to codewords, which are always absolute. This relativisation is expected to be very useful in R2. As presently in the language, the Address is of the form:



ADDRESS ELEMENT

However, its form is expected to change soon, since the

definition of the operating system will require the indirect addressing feature.

THE ASSEMBLY SYSTEM: BASIC FEATURES

The Assembler itself is basically a two pass assembler, operating much in the way of normal assemblers. On the first pass, the user-generated text is scanned and condensed. The text that is scanned is much in the form of normal assemblers. Its format is:

LOCN	X	OPN	Y±N	TG/COMMENT
|CR	|1st tab	|2nd tab	|3rd tab	|4th tab

where the LOCN field is the label one associates with the line, X,Y,OPN, and ±N are defined as for the hardware, and TG allows placing of tags on instructions.

During the first pass, any direct translation possible is performed, and the partially formed orders and parameters for second pass evaluation are stored in a vector of intermediate results. A symbol table is built up, constants are evaluated, and data is taken to compile a cross reference dictionary. The input text may be listed during this phase.

On the second pass, the intermediate results vector is processed. Pseudo operations are performed, which may, for example, cause an arbitrary structure, which may be a matrix or vector of constants, to be placed in the output code. Symbolic references from the first pass are evaluated, based on the symbol table compiled during the initial scan. Results of these operations are stored into a final code vector. When this has been done, a request for storage space in the vector serving as the memory for the new machine is made. If the

request can be satisfied, the final code is moved into the new memory. A control element is generated to describe the segment, based on the segment command following the program. For example, consider the following sequence of text presented to the system.

$INST_1$
:
$INST_N$
<ijk

The system would be informed that it had an assembly on its hands, and control would be given to the assembler. On encountering <ijk in the input stream, the code generated would be loaded into the new memory, and control would return to the operating system.

The options mentioned earlier, if exercised, will be filled during the second pass before control returns to the system. The programmer may receive an octal listing of his output code, followed by his portion of the symbol table. He may also receive the cross reference dictionary, which defines each variable used, and lists its definition point and occurrences within the program. Further, he may elect to have the final code vector and other relevant information punched on paper tape, thus perhaps allowing assembly to be restarted at the point of loading into the memory at some future date.

THE ASSEMBLY SYSTEM: SPECIAL FEATURES

The operation code field of an R2 instruction is only six bits long, which would ordinarily imply an instruction repertoire of sixty-four basic operations. However, two

features of the machine organization greatly extend the basic instruction set. The first of these is the tagging of data words as to type, which allows operation codes to effectively be carried along with data they operate on. In other words, only one ADD order is needed for all types of data, since data tags cause modification of the "ADD" microcode. The second of these features is the use of the X,Y, and ±N field to act as instruction modifiers. This feature of the hardware could conceivably cause problems for the pregrammer, keeping modifications straight. One of the features of this assembly system is that unique mnemonics take the place of large numbers of modifiers, saving difficulties of numerically coding modifiers. Indeed, it is impossible to code some modifiers numerically, for reasons explained later.

A case of the usefulness of these mnemonics is in the case of the shift orders. In the present machine there are about 29 shift orders, of which 11 are assigned AP1 mnemonics. In R2 there will be two, with all combinations available by use of the X field. The AP1/R2 system will provide for the use of many shift mnemonics, which map onto the 2 shift orders and their inflection fields. For example

```
LLS     10
LRS     10
LUL     10
LUR     10
```

code into

	2nd tab	3rd tab	
00	12	001	00012 (octal)
01	12	001	00012
00	13	001	00012
01	13	001	00012

respectively.

As a matter of convenience, the three forms of jump orders are handled in a special way. The coder may use the letter "J" for any jump, with the contents of the X field. Advantage is taken of the fact that the content of the X field is unique for each. Without going into detail of meaning, the following examples show what is meant:

```
            J       LABEL 1
AT          J       LABEL 2
1           J       LABEL 3
X3          J       LABEL 4
|1st tab   |2nd tab  |3rd tab
```

are equivalent to:

```
UN          JCC     LABEL 1
AT          JCC     LABEL 2
01          JSM     LABEL 3
X3          JSL     LABEL 4
|1st tab   |2nd tab  |3rd tab
```

It is possible to do this becuase unlike the previous assembly system, this system can readily locate each field of the symbolic text at will.

Two forms of remarks are written into the assembly system. The first of these is remarks field of the fourth tab, which may also be used to set a tag on the instruction for selective tracing purposes. Text at the fourth tab is ignored unless the first two-letter word is "TG", in which case a trace tag is set on the instruction. The second form of remark is that which begins with a slash (/) at the carriage return position.

Although it is not the case now, another special feature of the assembly system will be the set of checks it employs to prevent improper operations on the part of the programmer. There will be no priviledged mode of operation in R2, only

a priviledged mode of assembly. It will be up to the assembly program to check for transfers into data words, for attempts to fetch instructions, for attempts to store into invariant constants, and to see that no transfers to undefined locations take place. These checks can be performed in the second phase of assembly, as location equivalence are being worked out. Essentially the only additional burden on the assembler is that it knows what order it is assembling.

Monitoring of potentially dangerous operations, such as storing to absolute core addresses or relative to the instruction will be aided by the requirement that operation codes and their inflections ordinarily be coded in symbolic form. Since the assembly program assembles only invariant code, this is sufficient to allow Phase I checks on these items.

A recent added feature of AP1/R2 is the generalized method of defining constants. The programmer equates a label to a set of constant elements, each element of which may be a set of elements. An element is enclosed in parentheses, which may be nested to an arbitrary level. The result is a tree of constants, of which special cases are single constants, vectors, and matrices. Any of the data forms permitted by the assembly program may be used with their proper prefixes -- none for decimal, "c" for octal, "b" for bit string, or "h" for hexadecimal. The ability of the assembly system to generate these structures rest on the relative nature of address elements in R2.

PLANS FOR RE-IMPLEMENTATION IN R2

Current plans call for re-coding the assembler in itself and running it under control of the simulation program. As a result, the assembler should be ready for use when the machine itself is ready. The primary problem becomes getting a version of itself initially in core. This is expected to be done by the old simulator, since the new and old computers will probably be coupled via a device channel for some period of time before R2 begins independent operation.

As the assembler is recoded, it is expected that many improvements will be made to the language, to make it a full assembly system. The checks mentioned above must be implemented, and also provision must be made for addressing variables in a manner compatible with the proposed operating system conventions must be included.

"Implicit" macro facilities for implementation of Basic Language constructs will be available.[2] For example, it may become possible to say

```
X1      LD      X2.7.1
```

which would expand to

```
X1      LD      X2
X1      DOT      7
X1      DOT      1
```

Duplicate macro facilities will also be used to code orders which will eventually become instructions in R2 in future years. When these orders are implemented, the assembler will simply stop treating them as macro orders, and substitute actual function codes.

Finally, proposals for extensions of the language even beyond the Basic Language include proposals for conditional assembly and explicit macro orders. It is not known at this time whether these will be included, or what form they will take.

NOTES

[1]John Iliffe, Basic Machine Principles, (London, MacDonald, 1968), pp. 47-73.

[2]Iliffe, p. 56.