SLAC-PUB-2206
October 1978
(I)

SOFTWARE FOR MICROCIRCUIT SYSTEMS*

Paul F. Kunz

Stanford Linear Accelerator Center
Stanford University, Stanford, California 94305, U.S.A.

INTRODUCTION

Modern Large Scale Integration (LSI) microcircuits are meant to be programmed in order to control the function that they perform. In reference[1], I have already discussed the basics of microprogramming and have studied in some detail two types of new microcircuits. In this course, I will explore the methods of developing software for these microcircuits. This generally requires a package of support software in order to assembly the microprogram, and also some amount of support software to test the microprograms and to test the microprogrammed circuit itself.

1. MICROPROGRAM ASSEMBLERS.

1.1 ASSEMBLERS IN GENERAL.

An assembler is a support software program which allows the programmer to write a program in a symbolic language. It does many tasks for the programmer which greatly relieves to tedium of writing programs. These tasks are illustrated in an example of assembly language program given in figure 1 which comes from the output of the IBM 360/370 Assembler. Of interest to us are the columns labeled 'LOC', 'OBJECT CODE', and 'SOURCE STATEMENT'. Under the column labeled 'LOC' is to relative address of an IBM 360/370 instruction which is represented in hexadecimal format under the column labeled 'OBJECT CODE'. The symbolic program is presented under the column labeled 'SOURCE STATEMENT'.

The first task of the assembler is to convert the symbolic operation codes into the machine binary code. For example, in figure 1 the operation code 'SR' was converted into the machine code '1B' and placed in the proper field of the machine instruction as shown at 1. A second task is to substitute for symbolic variable names the machine binary form. One can see at 1 in figure 1 that the symbols '3,2' have been converted to a binary form and placed in the proper fields of the machine instruction for the source and destination registers. Also, at point 2 of the figure the symbolic variable name 'ZAS(7)' has been substituted with the proper form of memory addressing. The third task is to substitute symbolic addresses with the binary addresses. In figure 1 at point 3, the instruction at location '17A' was given the symbolic label 'A22'. All references to this location made by the program used the symbolic name such as the one at location '19C'. The assembler substitutes for the symbolic name the actual address. This was also the case for the address of the symbolic variable given at point 2. The fourth task for the assembler is to supply after all the conversions and substitutions a complete binary program that can be loaded into the processor. In the case of the IBM assembler, this machine code is called the Object Code.

With the aid of the assembler, the programmer can write programs of great length and complexity which would be too difficult to write directly in machine code. Instructions can be inserted or moved without difficulty when all variables and branch addresses are referred to symbolically because the assembler will do the work of calculating the real addresses in generating the Object Code. The programmer is also almost completely relieved of having to know about the placement of the fields in the machine instruction and the details of addressing memory.

The assembler is generally given by or bought from the manufacturer of the computer when one receives the computer as part of a package of support software. It is written expressly for the computer. If one is to supply an assembler for a microprocessor of one's own design, then one has the problem of having an assembler written for that machine. Methods of doing this will be discussed in the next sections.

1.2 MICROPROCESSOR ASSEMBLERS IN GENERAL.

An assembler written expressly for a microprocessor should have the same general features as an assembler for a computer. One should realize, however, that there may be some major differences between the instruction set of a microprocessor and a that of a computer. Take for example the question of the operation codes. A computer typically has a set of about 100 to 400 instructions. Each instruction may have a few parameters, and these parameters are specified in a way which is common to many instructions. For example, the 'Subtract Register' (SR) instruction of the IBM 360/370 discussed above had two parameters, the source and destination registers. An 'Add Register' instruction has the same two parameters and they are specified in the same way.

A microprocessor might have an several orders of magnitude more operation codes if we tried to define them in the same way as computer operation codes. For example, a processor with a 2901A bit slice microprocessor requires that the source, function and destination fields be specified as was shown in figures 36 and 37 of reference [1]. Since each of these fields is 3 bits in length, there would be 512 combinations which could be considered operation codes. And since the CARRY-IN to the least significant bit also needs to be specified, we find that there are really 1024 combinations. The A and B addresses, of course, may be considered as operands, as were the source and destination registers with the IBM 360/370.

Besides the fields of the microinstruction defining the operation of the 2901A, we may find a microsequencer in the processor. The 2910 microsequencer, for example, has a 5 bit microinstruction

* Work supported by U. S. Department of Energy, EY-76-C-03-0515

(Presented at the 1978 CERN School of Computing, Jadwisin, Poland, May 28, - June 10, 1978.)

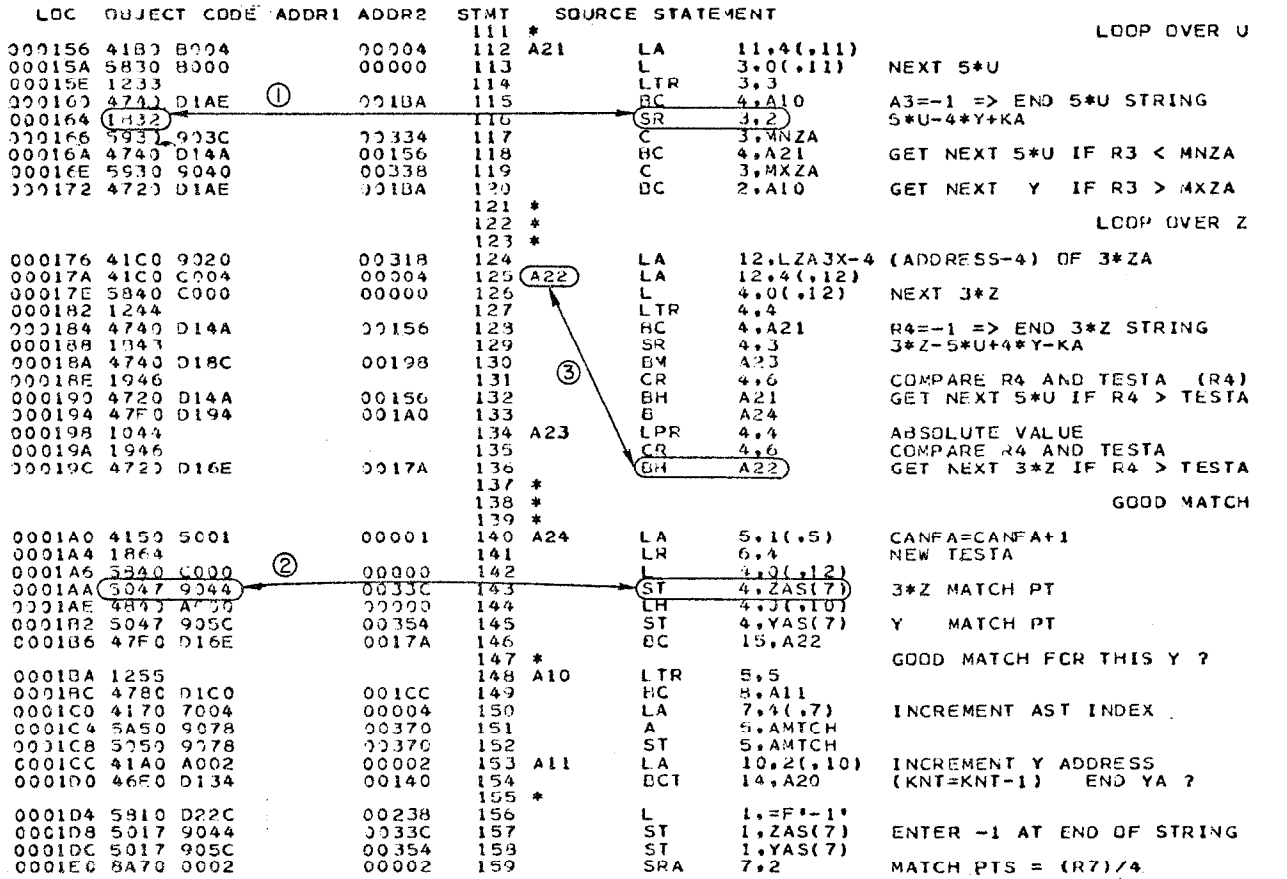

Figure 1: Example of Assembler Output.

code if we include the condition code enable bit along with the four next address instruction bits. If the next address control circuit were as shown in figure 40 of reference [1], then we must also condsider the four bits which control the condition code multiplexer. All the combinations of next address control must be joined with the 2901 instruction which would lead to a total of 524,288 operation codes.

Thus, it is general practice for microprocessor assemblers to divide the microinstruction into several fields each with its own set of operation codes and operands. Whereas an ordinary computer assembler generates one machine instruction for one operation code, the microassembler may expect several operation codes to be concatenated into one microinstruction.

Another difference between ordinary computer instructions and microinstructions is the appearance of Don't Care fields in the microinstruction. In the case of the simple scanner described in section 3 of reference [1], the branch address field was not used for instructions in which the next address was taken as the next sequential address (CONTinue instructions). As we will see below, it is also convenient to have "Default" fields in the microinstruction. A microassembler must be able to handle these situations as well as the multiple operation codes.

2. GENERATION OF MICROPROGRAMS

There are many methods for generation of microprograms and we will study four of them. The choice of which method to use depends on many factors, such as the length of the expected programs, the execution efficiency required, the complexity of the instruction set, etc. And as with programming computers, a programming language that generally leads to efficient code such as assembly language may be rejected in favor of a programming language which is easier to use by a average programmer, such as FORTRAN.

2.1 HAND-CODED BINARY.

Hand-Coded binary programming is a method in which the programmer writes directly in the binary bit pattern of the processor's microinstruction set. This method was used in programming the simple scanner in section 3 of reference [1]. Figure 33 of reference [1] gave the instruction set of the processor. The program to perform a scan of devices with data was written in only six instructions as was shown in figure 34 of that reference.

The Hand-Coded binary method was a perfectly viable method in the case of the simple scanner. Unlike the methods which will be discussed in the following sections, it requires no support software in the form of assembler programs. The only support

software that may be needed is a way of actually loading the microprogram memory and even there one could think ways to get around using software to do this.

If the program should become very long this method can be very tedious. It is also relatively difficult to read the program many months after is was written. Modifying the program may also be very time consumming. For example if one inserted a few new instructions in the middle of a program, then one might need to change many other instructions in order to correct the branch address field for those instructions which have changed their address. The hand-coded binary method should probably only be used for very short and simple microprograms which do not need to be modified often.

2.2 DEDICATED MICROASSEMBLER.

A dedicated microassembler is an assembler which has been written to assemble programs for one microprocessor. If the processor is simple, and one does not expect an assembler with many of the sophisticated features we normally associate with assemblers that come with computers, then one can write a dedicated microassembler relatively quickly.

As an example of a dedicated microassembler, let us study a microprocessor designed by Guzik for use at experiment at FermiLab[2]. It's purpose was to read data from a CAMAC crate and make a decision on whether the event should be read out by the host computer. Figure 2 is a simplified block diagram of the processor which is based on the 2901A bit slice microprocessor and a 2909 microsequencer.

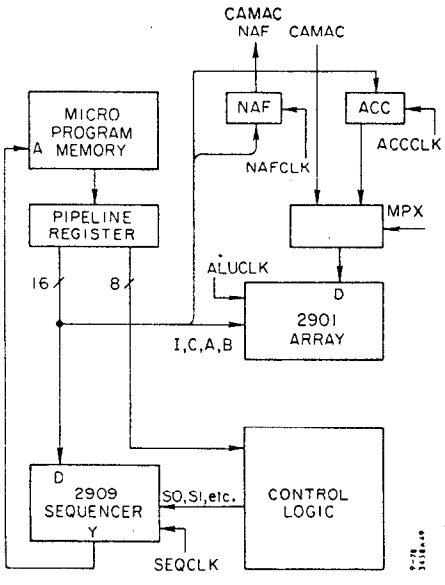

Figure 2: Block Diagram of Guzik's Processor.

As is shown in figure 3, the microinstructions are 24 bits in length with an 8 bit control field and a 16 bit operand field. The control field has bits that are routed directly to the control points within the processor. Two bits control the next address multiplexer of the 2909 (S0,S1). The SEN bit enables the stack file of the 2909. The CND bit allows the SIGN bit from the 2901A to be ORed with the least significant bit of the microinstruction address. This is the only form of conditional branching the processor can execute. The MPL bit was used for iterative multiplication. The DAT bit controls whether the operand field is clocked into the accumulator register or the CAMAC address register. The MPX bit controls the multiplexer at the input to the 2901A. It selects either the accumulator register which could be loaded from the 16 bit data operand field of the instruction or the data on the CAMAC READ lines. And finally, the MOD bit controlles whether the 2901 or a register is clocked. The operand field may be used for one of four purposes: a 16 bit data word, a 16 bit microinstruction address, a instruction for the 2901, or a CAMAC command. Figure 4 shows the bit assignments in the Operand field.

Figure 5 is an example of the processor's microassembly source code. The control field and operand fields are handled in different ways. For the control field the source is divided into 7 columns corresponding to the 6 control bits and the one 2 bit field. In these columns, the program writes a symbol to generate a '0' or '1'. These symbols are easily interpreted in terms of what the processor is controlling during that microinstruction. For some of the fields, the a blank means the Don't Care state of that subfield, while in others it means the Default value.

As discussed above, the operand field can have one of four different meanings. The first character of the operand field of the source code contains a symbol which tells the assembler which kind of operand follows. These symbols are

- $ for data operand
- * for microinstruction address
- & for 2901A instruction field
- # for CAMAC command

Within the rest of the operand field are further symbols to indicate parameters to put into the subfield, if any, of the operand. The microinstruction address may have a symbolic name or label and it is placed in the left most column of the instruction. The right most column is reserved for a comment field.

If we look in detail at a few instructions we should be able to see how the assembly language is used. The first instruction has the symbolic label 'START'. The MOD column has the symbol 'OPR' indicating that the 2901A will not function for this cycle. The DAT column has the symbol 'NAF' so that the operand field will be loaded into the CAMAC NAF register. The SEQ field has the symbol 'CNT' so that the next microinstruction will be taken from the microprogram counter register of the 2909. The operand field starts off with a '#', thus the operand contains the CAMAC address and function. The symbol 'N(3)' means that the station number subfield should be filled with a '3'. Similarly, the symbols 'A(0)' and 'F(0)' cause the subaddress and functions subfields to be filled with '0'. The net effect of this instruction is thus a CAMAC Read at station 3 subaddress 0.

The next instruction is similar to the first except that the operand field is loaded into the accumulator as is indicated by the symbol 'DAT' in the DAT column. The operand field starts with the symbol '$' to show that what follows in a single data word and the symbol 'AX' is used for this word. Thus this instruction loads a 16 bit data word from the

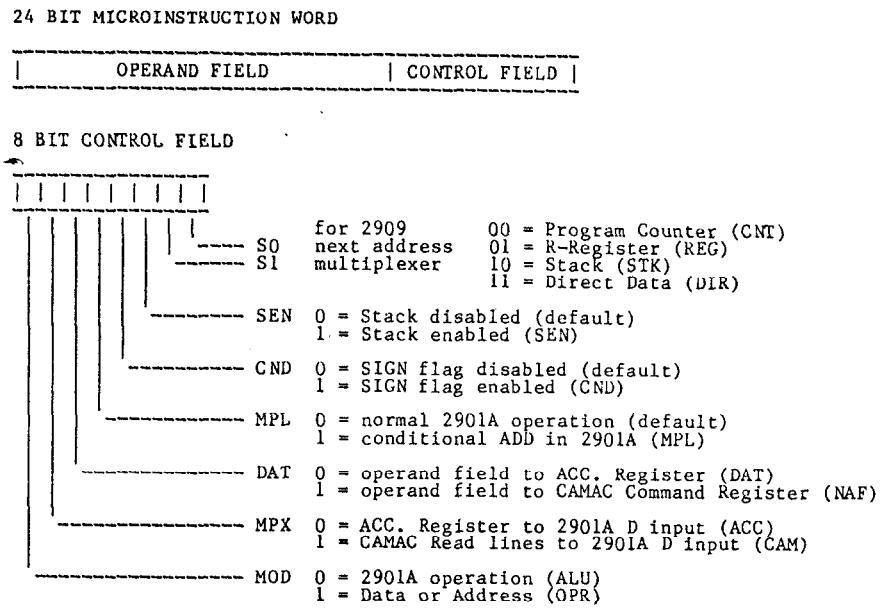

Figure 3: Control Field of Guzik's Microinstruction Word.

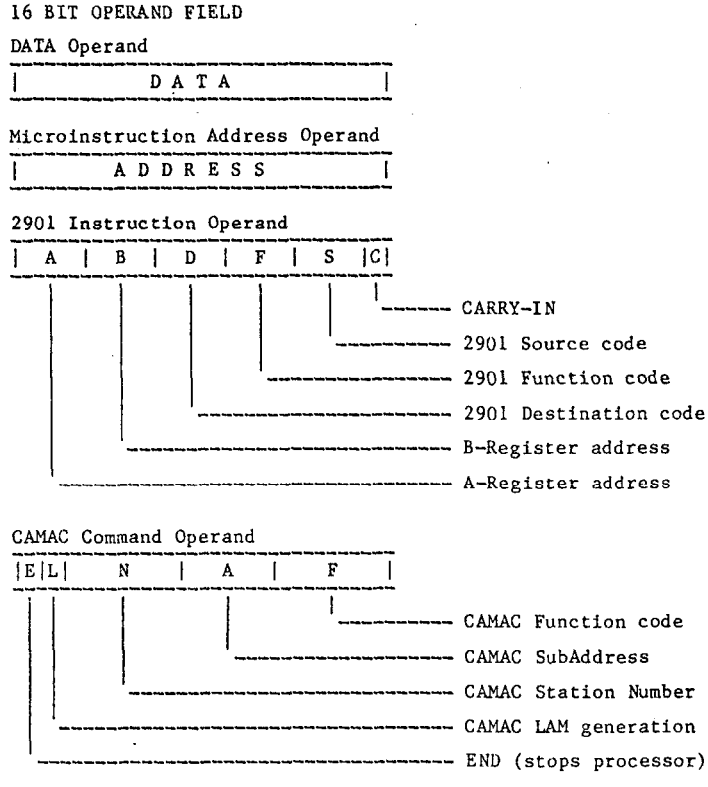

Figure 4: Operand Field of Guzik's Microinstruction Word.

operand field of the microinstruction into the accumulator register.

The 2901A will function in the next instruction as indicated by the symbol 'ALU' in the MOD column. The D input multiplexer selects the accumulator source as indicated by the 'ACC' in the MPX column. The operand field starts with the symbol '&' which means that the operand field source is be read as subfields of the 2901 instruction. In this instruction the 2901 source, function and destination codes are programmed to be 'D,0', 'OR', and 'F->Q'. The A and

```
//;LABEL:   MOD MPX DAT MPL CND SEN SEQ,  OPERAND FIELD          ;COMMENT
///
00/START:  OPR     NAF            CNT, #N(3) A(0) F(0)          ; NAFREG:=NAF(X1)
01/        OPR     DAT            CNT, $AX                      ; ACC:=AX
02/        ALU ACC                CNT, &A(X)B(X) D,0 OR FQ 0    ; Q:=ACC
03/        ALU CAM                CNT, &A(X)B(2) D,0 OR FBF 0   ; R2:=CAMAC(X1)
04/        OPR                SEN DIR, *MULTI                   ; JSR MULTI
05/        ALU                    CNT, &A(2)B(0) 0,A OR FBF 0   ; RQ:=R1
06/        OPR     NAF            CNT, #N(3) A(1) F(0)          ; NAFREG:=NAF(X2)
 .          .                      .     .
 .          .                      .     .
 .          .                      .     .
57/MULTI:  ALU                    CNT, &A(X)B(1) 0,B AND FBF 0  ; R1:=0
58/        ALU         MPL        CNT  &A(2)B(1) A,B ADD UFQ 0  ; COND ADD AND SHIFT
59/        ALU         MPL        CNT  &A(2)B(1) A,B ADD UFQ 0  ; COND ADD AND SHIFT
60/        ALU         MPL        CNT  &A(2)B(1) A,B ADD UFQ 0  ; COND ADD AND SHIFT
61/        ALU         MPL        CNT  &A(2)B(1) A,B ADD UFQ 0  ; COND ADD AND SHIFT
62/        ALU         MPL        CNT  &A(2)B(1) A,B ADD UFQ 0  ; COND ADD AND SHIFT
63/        ALU         MPL        CNT  &A(2)B(1) A,B ADD UFQ 0  ; COND ADD AND SHIFT
64/        ALU         MPL        CNT  &A(2)B(1) A,B ADD UFQ 0  ; COND ADD AND SHIFT
65/        ALU         MPL        CNT  &A(2)B(1) A,B ADD FBF 0  ; ADD
66/        OPR                    STK, $0.                      ; RETURN
```

Figure 5: Example of Guzik's Microprogram.

B register address are not used which is indicated by the 'X' in the symbol 'A(X)B(X)'. In the next instruction, however, the data on the CAMAC Read lines are loaded into register 2 of the 2901. Note the symbol 'CAM' in the MPX column and the operand field symbol 'A(X)B(2)'.

The next instruction illustrates how a subroutine call is programmed. The stack enable bit (SEN) is turned on with the next address multiplexer of the 2909 selecting the D inputs ('DIR' in column SEQ). The operand field must then contain a micro-instruction address so it starts with a '*'. The operand field contains the symbol 'MULTI' which is also used further down in the program to label a microinstruction address. The microassembler will substitute the binary address of MULTI into the operand field of this instruction.

It is left as an exercise to the reader to read the rest of the program. One might accuse this microassembler of being rather primitive, but the author feels it fits well to the task it must do. The processor was designed to execute a simple program and one can even notice that not all the functions of the LSI microcircuits were implemented in the circuit. Likewise, the microassembler only is capable of doing what the programmer needs: to write the relative short programs that this processor will be used for. Programming this processor with the assembler is considerable easier than using the hand-coded binary method and yet the assembler is not so complex that it is difficult to write.

2.3 META ASSEMBLERS.

Assemblers for microprocessors or computers perform very similar tasks. The code that must be written to write an assembler is also very similar. It is possible to divide the task into those parts which are the same for all machines and those parts which depend on the processor's instruction set. Then, if an assembler would be written to accept as input the definition of the processor, we could reuse this sort of assembler for many different processors. Such an assembler is called a an assembler assembler or a "meta-assembler".

A meta assembler operates in two phases, the definition phase and the assembly phase. The definition phase, which must be executed first, sets up tables with the programmer's defined set of instructions and their model format. That is to say, the programmer specifies the symbols which will be used in the assembly phase to produce the binary bit pattern of the microinstruction. The assembly phase then uses the output of the definition phase and the source program input and operates in the same way as an ordinary computer assembler.

In order to study the properties of a meta assembler, we will study in some detail a meta assembler called AMDASM which was written by Advance Micro Devices for users of their LSI microcircuits. We will only discuss only the some of the features of AMDASM in order to bring out the basic ideas. More details may be had in the reference manual which is included with the book on circuit specifications[3].

2.3.1 Definition phase example.

The definition phase of the AMDASM meta assembler has two basic statement types: the EQU statement and the DEF statement. An EQU statement is used to generate a symbolic name for a constant value or expression. An example would be:

```
S4: EQU  B#100
```

which sets the value of the symbol 'S4' to a 3 bit binary number '100'. Once the EQU statement is made, any further reference to the bit pattern '100' may be made by using the symbol 'S4'. A choice of four number systems may be made. The programmer selects which one by the letter in front of the '#' symbol as follows:

B for binary,
Q for octal,
D for decimal, and
H for hexadecimal.

The purpose of the EQU statement is the same as in ordinary assemblers, it relieves the programmer of the tedious task of always coding the bit pattern. Instead he can code a symbol which makes the program not only easier to write but also much easier to read and understand at a later date. Also, if the constant bit pattern need be changed, it may be done only at the EQU statement rather then throughout the program where that pattern may be used. Consider for example setting up the function code field of the 2901. We may write the following EQU statements:

```
ADD:    EQU  Q#0      ; R PLUS S
SUBR:   EQU  Q#1      ; S MINUS R
SUBS:   EQU  Q#2      ; R MINUS S
OR:     EQU  Q#3      ; R OR S
AND:    EQU  Q#4      ; R AND S
NOTRS:  EQU  Q#5      ; (NOT R ) AND S
EXOR:   EQU  Q#6      ; R EX-OR S
EXNOR:  EQU  Q#7      ; R EX-NOR S
```

The DEF statement is a model of the microinstruction that is to be generated by a symbol in the assembly phase. The microinstrucion word is broken up into fields of specified length with the sum of the lengths being equal to the length of the microinstruction word. There are three kinds of field specifications: constant, variable, and "Don't Care". A constant field is one which always supplies the same constant bit pattern each time the microinstruction is invoked in the assembly phase. A variable field is one in which a variable value may be supplied in the assembly phase. One feature of a meta assembler which is not common to an ordinary assembler, is that if the variable value is not explicitly stated in the assembly phase then a default value for the field is supplied by the assembler. The DEF statement also provides a mechanism for specifying the default value. The "Don't Care" fields are those which are unaffected or not needed by the model microinstruction.

As an example of the DEF statement, consider the simple scanner example again. Figure 33 of reference [1] defines the fields associated with that processor. Using the AMDASM assembler we can define the following microinstructions to handle the branching portion of the microinstruction as follows:

```
CONT:    DEF  4X,B#00,7X     ;CONTINUE
BRDV:    DEF  4VH#F,B#01,7X  ;BR DATA VALID
BRDONE:  DEF  4VH#F,B#10,7X  ;BR DONE
JUMP:    DEF  4VH#F,B#11,7X  ;JUMP
```

In the first statement, the symbol 'CONT' is a microinstruction with 3 fields which are separated by a ','. The first and last are Don't Care fields which is indicated by the 'X'. These fields are 4 and 7 bits in length respectively. The second field is 2 bits in length and contains the constant value '00' binary. The same number system specification as the EQU statements are used in the DEF statements. In the remaining statements the first field is a variable field four bits in length. The default value will have the value 'F' hexadecimal. The second and third fields in these statements are like those of the first statement except the constant value is different. A complete input to the definition phase of the AMDASM assembler for the simple scanner is shown in figure 6 .

```
;AMDASM MICRO ASSEMBLER EXAMPLE
;SIMPLE SCANNER
;
; DEFINE WORD LENGTH
;
        WORD    13
;
; DEVICE CONTROL EQUATES
;
IDAC:   EQU     B#1
NEXT:   EQU     B#1
ZDAC:   EQU     B#1
;
; NEXT ADDRESS CONTROL DEFINITIONS
;
CONT:   DEF     4X,2B#00,7X     ; CONTINUE
BRDV:   DEF     4VX,2B#01,7X    ; BRANCH IF 'DATA-VALID'
BRDONE: DEF     4VX,2B#10,7X    ; BRANCH IF 'DONE'
JUMP:   DEF     4VX,2B#11,7X    ; JUMP
;
; MEMORY CONTROL DEFINITIONS
;
ZMAC:   DEF     6X,Q#4,4X       ; ZERO MEMORY ADDRESS COUNTER
IMAC:   DEF     6X,Q#2,4X       ; INCREMENT MEMORY ADDRESS COUNTER
WMEM:   DEF     6X,Q#1,4X       ; WRITE TO MEMORY
MNOP:   DEF     6X,Q#0,4X       ; MEMORY NO OPERATION
;
; DEVICE CONTROL DEFINITIONS
;
DEVICE: DEF     9X,1VB#0,1VB#0,1VB#0,1X  ; VARIABLE FIELDS ARE:
;                                        ; 'NEXT' PULSE
;                                        ; INCREMENT DEVICE COUNTER
;                                        ; ZERO DEVICE COUNTER
;
; STOPPING CONTROL
;
STOP:   DEF     12X,1VB#0       ; SET TO '1' TO STOP
;
END
```

Figure 6: Definition Phase Input for Simple Scanner.

x

2.3.2 Assembly phase example.

The assembly phase reads the source program statements, substitutes values for the constants and labels, and generates the bit pattern which is to be loaded ino the microprogram memory. The symbols used in the source statements must be either those defined in the definition phase or from EQU statements given in the assembly phase. The meta assembler in this phase looks very much like an ordinary assembler except for its ability to overlay or concatenate several microinstructions into a single microinstruction word. This feature is made clear by considering statements for the program of the simple scanner.

The program flow of the simple scanner is given in figure 31 of reference [1]. In the first instruction, the processor must reset the DEVICE COUNTER and ADDRESS COUNTER and send the first NEXT signal. The next instruction will be the next sequential address. This instruction may be coded as

```
;AMDASM MICRO ASSEMBLER EXAMPLE
;SIMPLE SCANNER
;
; PROGRAM PHASE
;
START:  CONT        & ZMAC  & DEVICE NEXT,,ZDAC ; RESET ALL
DVCHK:  BRDV DVON   & MNOP  & DEVICE            ; TEST 'DATA-VALID'
        BRDONE FINI & MNOP  & DEVICE            ; TEST 'DONE'
NXTDV:  JUMP DVCHK  & MNOP  & DEVICE NEXT,IDAC  ; TRY NEXT DEVICE
DVON:   CONT        & WMEM  & DEVICE            ; WRITE TO MEMORY
        JUMP NXTDV  & IMAC  & DEVICE            ; INCREMENT MEM ADR
FINI:   JUMP START  & STOP 1                    ; STOP
;
END
```

Figure 7: Assembly Phase Example for Simple Scanner.

shown in the first program statement of figure 7 . Reading this statement from left to right, it is explained as follows. The 'START:' is a label for this microinstruction address. It does not effect the instruction generated for this location in any way. The 'CONT' is a symbol from the definition phase which sets bits 4 and 5 to '00' and leaves all the other bits in the Don't Care condition. The symbol '&' means concatenation or overlay is to be performed by the assembler. The symbol 'ZMAC' is also from the definition phase and it sets bits 6 through 8 to '100'. Because the overlay symbol has been used the bits affected by both the 'CONT' and 'ZMAC' will be set by this instruction. Continuing to the right another '&' symbol follows which means that even more bits are to be set. The 'DEVICE' symbol comes next and from the definition phase we see that it sets three variable fields. Only two variable symbol names follow: 'NEXT' and 'ZDAC'. The value of the variable 'NEXT' is equal to '1' because of the EQU statement in the definition phase. Thus the with combination 'DEVICE NEXT', the assembler will generate a '1' in bit 9 of the microinstruction. The next variable field of the 'DEVICE' instruction is not specified as we can see by the two commas occurring in a row. Thus the assembler will use the default value for this field and generate a '0' in bit 10. The last variable field is specified, and from the EQU statement for 'ZDAC', one can see that the assembler should generate a '1' in bit 11.

The next instruction is labeled 'DVCHK' and it overlays a 'BRDV', 'MNOP', and 'DEVICE' microinstructions into this single instruction. The 'BRDV' contains a variable field which is the first bits of the microinstruction. The variable to be used is 'DVON'. This symbol is used as the statement label for the fifth microinstruction in the program assembly. Thus the assembler takes the address of this instruction as the value of the variable DVON and puts this value in bits 0 through 3. The net effect of this code is for the program to branch to the instruction labeled 'DVON' if the 'DATA-VALID' signal is being received while doing no operation on either the buffer memory control or the device control signals. Step 1 on the flow diagram in figure 31 of reference [1] corresponds to this instruction.

One should now be able to follow the complete program for the simple scanner shown in figure 7 . Note that for the fields of the microinstruction which control the buffer memory and the device counter, I have used two different methods of setting the bits in order to illustrate variable substitution. I could have used either method for both fields or combined them into one microinstruction model.

One would think that the program given in figure 7 would yield the same bit pattern for the microinstructions as shown in figure 34 of reference [1]. One mistake, however, has been made in the program as shown. It illustrates that microprogramming with a meta assembler is not as easy at it seems. The 'STOP' microinstruction model was used in the program statement in order to halt the processor. The use the 'STOP' symbol followed by the variable field '1' is correct in the last statement of the program shown in figure 7 . In all the other instructions, however, this bit of the microinstruction is left in the Don't Care state because the symbol 'STOP' does not appear. The default value '0' which is desired for these instructions will not be invoked unless the symbol 'STOP' appears in the source statement. As will be seen in section 3, all the Don't Care fields must be translated to either '0' or '1' when they are moved into the microinstruction memory since obviously a memory can't store a Don't Care. Thus, unless all Don't Cares are translated to '0', the processor will not function pass the first instruction. Figure 8 shows the program corrected with the symbol 'STOP' appearing in each statement. In all but the last statement the default value '0' is generated because no variable is specified.

```
;AMDASM MICRO ASSEMBLER EXAMPLE
;SIMPLE SCANNER
;
; PROGRAM PHASE
;
START:  CONT        & ZMAC & DEVICE NEXT,,ZDAC & STOP   ; RESET ALL
DVCHK:  BRDV DVON   & MNOP & DEVICE            & STOP   ; TEST 'DATA-VALID'
        BRDONE FINI & MNOP & DEVICE            & STOP   ; TEST 'DONE'
NXTDV:  JUMP DVCHK  & MNOP & DEVICE NEXT,IDAC  & STOP   ; TRY NEXT DEVICE
DVON:   CONT        & WMEM & DEVICE            & STOP   ; WRITE TO MEMORY
        JUMP NXTDV  & IMAC & DEVICE            & STOP   ; INCREMENT MEM ADR
FINI:   JUMP START  & MNOP & DEVICE            & STOP 1 ; STOP
;
END
```

Figure 8: Assembly Program for Simple Scanner Corrected.

2.3.3 Other features in meta assemblers.

Even a meta assembler as simple as that part of the AMDASM assembler described above can make programming a microprocessor much easier when compared to hand-coded binary. It may also be used for many different microprocessor projects and programs can be easily be modified when a processor is modified. There are other desirable features one would like to have in a meta assembler which would make programming even easier and a few of these features are described below.

First of all, one would like to remove the positional dependence of the variable substitutions. This may be accomplished if the assembler has what is called a 'keyword' ability. Consider the following source program statement which might come from a processor with a 2901A:

```
ALU=(A(1),B(3),OR,AB,FBF),BR=(Z,LOOP)
```

From the description of the 2901A, we can imagine that this statements calls for registers 1 and 3 to be selected for the A and B outputs respectively, an OR ALU function code with A and B as the two ALU operands, and the results loaded back into the register file. In the same microcycle the program should branch to the location labeled by the symbol 'LOOP' if the result is zero. The Keywords can be at least 'ALU=', and 'BR=' as well as 'A( )' and 'B( )'. A meta assembler which could correctly interpret the statement this way would be much easier to write programs for and the programs would be much easier to read. One could also imagine that symbols such as 'OR', 'AB', and 'FBF' could be defined in such a way that they would cause certain fields of the microinstruction to be generated independent of the position within the microprogram statement.

Another feature, which is sometimes very useful in ordinary assemblers, is the macro capability. It allows the programmer, in the assembly phase to define one or more instructions to be generated by a symbol defined as a macro. For example one could define a macro 'OR' which when written in a program statement thusly:

```
OR A(1),B(3)  BR=(Z,LOOP)
```

would generate exactly the same microinstruction shown in the previous paragraph. Some ordinary assemblers with macro capabilities allow the user to test for the number of operands. With such a capability in a meta assembler, the statement

```
OR A(1),D,B(3)  BR=(Z,LOOP)
```

could be assembled as

```
ALU=(A(1),B(3),OR,DA,FBF), BR=(Z,LOOP)
```

where the use of the D inputs for one operand was understood by the assembler because there were three operands in the argument list.

Another use of the macro capability is the generation of multiple microinstructions with one macro. Consider a macro called 'LOAD' which when written in a statement like

```
LOAD Q,X4
```

would generate

```
MOPR=(X4,READ),CONT
ALU=(A(X),B(X),OR,DØ,FQ), CONT
```

which might be a memory read cycle with address X4 followed by passing the memory contents through the 2901 ALU in order to load it into the 2901 Q register.

One might also want an automatic default feature to avoid the kind of error that was made with the simple scanner program as shown in figure 7 . That is, with the automatic default feature, any model microinstruction not appearing in the microprogram statement would be set to its default field. Such a feature however, might lead to other kinds of programming errors if the programmer had to make a real choice of possible variables. On the other hand, in all the microprogram statement examples we have shown so far, one had to always specify a 'CONTINUE' for the next address selection. It would be easier for the programmer if the assembler defaulted to 'CONTINUE' unless a 'Branch' was explicitly stated.

2.3.4 Difficulties with meta assemblers.

It seems that with an assembler of the type described with the AMDASM assembler that microprogramming can be quite easy. One way of analyzing the difficulty is the consider some of the errors one might make and when these errors can be detected. Table 1 gives such a possible list of errors and also shows what kind of errors are unique to microprocessor assemblers as compared to ordinary assemblers.

2.3.5 Obtaining a meta assembler.

The meta assembler is a good, if not essential, starting point for developing the necessary microprograms. The question is: How does one obtain a meta assembler which has the capabilities required for microprocessors we would find in our laboratories. The decision is basicly whether to buy one or to write one.

One can write a meta assembler in FORTRAN or assembly language. It is not as formidable task as it might appear if one keeps the definition and assembly phases simple enough. Probably the most tedious part of the task is writing the code which recognizes the syntax of the character strings. One should consider a programming language which is good at this. The rest is simple.

Since the task of recongizing character strings is already done by the assembler one has with a computer, one might like to find a way to use this assembler to do most of the work. One way to use an existing assembler is to make use of its macro capabilities. This approach is highly desirable because many of the necessary capabilities of an assembler will be taken care of by the host assembler so the user will not need to rewrite them. These include the assignment of symbols to constant values, the handling of address labels, and the substitution of variable values into instructions. Also the writing of the assembler is made easier by the macro language capabilities for decoding of parameter fields and the general ability for manipulating character strings and bit patterns. There are also some higher level languages which have a macro capability, such as PL/1, which should be good for writing a meta assembler.

An example of this latter approach is the MIMIC assembler written at SLAC by Edward Frank[4]. It

TABLE 1 Errors in Microprogramming

DEFINITION PHASE	
Illegal character strings	Ordinary
Undefined symbols	"
Duplicate labels	"
Bad word length	Microassemblers
Bad field length	"
ASSEMBLY PHASE	
Illegal character strings	Ordinary
Undefined symbols or labels	"
Duplicate labels	"
Overlay error	Microassemblers
Misplaced variables	"
Incorrect variable specification	"
Missing variable with no default	"
Necessary field unspecified	"
RUN PHASE	
Bad field position	Microassemblers
Bad field length	"
Wrong constants	"
Bad timing	"
Don't Cares are not	"

makes extensive use of the capabilities of the IBM 360/370 assembler. It consists of a set of macro definitions so that, except for EQU statements, every symbolic source statement calls a MIMIC macro. The general format rules of the IBM Assembler program are preserved, along with its error handling capabilities.

In the definition phase of the MIMIC meta assembler, the programmer writes the definition of his fields with a set of MIMIC macros. The macros generate a table of symbol names and may generate other macros. The table of generated macros are "Punched" to an output file which is in fact the output of the definition phase and they may be saved for future use.

In the assembly phase of the MIMIC meta assembler, the programmer codes the program using the symbols defined in the definition phase. The "Punched" output file is the set of macros necessary for this phase. These macros assemble the bit pattern of the desired microinstruction into assembler constants. The binary output is generated by a "DC" (Define Constant) IBM assembler directive. It may then be saved using all the normal facilities of the operating system.

An alternative to writing a meta assembler is to rent one. There are several meta assemblers available from the time sharing computer services. In the U.S., for example, the Computer Sciences Corporation rents the AMDASM that we have already studied. Output is available on paper tape ready to program PROM's. The cost of such a service is about U.S. $100-$1000 per month depending on the amount of programming done. Despite the seeming high cost for the rental, this may be the best answer when one has a small project and does not expect to need the meta assembler for other projects.

The same meta assemblers that one can rent are generally available to buy in the form of either a FORTRAN source program or a Load module for a mini or micro computer. In addition, there are other meta assembler that one can buy. Many of the semiconductor manufacturers, realizing that potential users of their LSI microcircuits need programming aids, are also in the business of selling meta assemblers. For example, Signetics sells a meta assembler written in FORTRAN, while Advanced Micro Devices sells an assembly program written for the 8080. There are also independent Software houses which have developed meta assemblers which are geared for the microprocessor logic engineer. Even IBM has a software product which is a meta assembler. It requires that the installation has APL. The cost of these meta assemblers is around U.S. $2000. For a survey of commercially available meta assemblers, the reader is referred to a recent article by V. M. Powers and J. H. Hernandez [5].

It is interesting to note that the interest in meta assemblers has increased dramatically since the availability of the LSI microcircuits. The semiconductor manufacturers find themselves not only supplying the circuits but also some software to help their customers use them. In addition, many companies are publishing extensive application notes which give examples for the use of their circuits. Whereas microprogramming was the domain of a few academics and professional computer designers in the past, it is now the focus of a large public education campaign waged by semiconductor companies interested in bring the technique to the largest body of people as possible.

Another method to obtain a meta assembler is to "steal" one. That is to say, get a copy of one from someone which is willing to give you a meta assembler he has written. Since the LSI microcircuits are becoming more and more in use in the High Energy Physics Laboratories, one will certainly find that some good meta assemblers will be developed. I have already mentioned the MIMIC meta assembler written at SLAC, for example, which may be used by anybody who has access to an IBM 360/370. Another one has been written by W. Wimmer at DESY[6].

2.4 EMULATION OF EXISTING COMPUTER.

For a given project in which a microprocessor is to perform some task, one has a generally a great deal of design flexibility. That is to say there are many ways in which the circuit could be designed to perform the task. Given a processor with a reasonable instruction set, the programmer can accomplish almost any calculation he desires. One microprocessor on the market (the 8X300, made by Signetics) has only 8 machine instructions and yet one could imagine doing very complicated calculations with it. Obviously, if the main feature of a

microprocessor is speed of calculation for one particular problem, then the choice between many possible designs becomes smaller. An example is the M7 processor[7] which was designed to calculate the effective mass from two particles in the detector. It has the capability of doing two multiplications and one addition in a single cycle.

For a microprocessor which is designed for general purpose, there is generally a larger choice of design. However, there is one choice which can greatly reduce the programming difficulty. It is to emulate an existing computer which one has access to, so that software development can be done on it. An example comes from my own work with a microprocessor called the 168/E which I have described in section 5.2 of reference [1]. The rational behind the 168/E was the following. A general purpose microprocessor must have three basic features:

1. arithmetic and logical operations perhaps with a register file or accumulator,

2. a means of memory addressing with efficient indexing capabilities, and

3. a means of condition branching on previous results.

The design of the 168/E chosen so that all these conditions was satisfied in a fashion which is so similar to the instruction set of the IBM 360/370 series of computers that one can easily translate IBM 360/370 programs into microprograms for the 168/E.

The advantages of emulation can be seen if one follows the steps for writing and debugging a program. First, a program is written in a language which is available for the emulated computer such as FORTRAN or assembly. One can then use a complier that exists for this machine to generate object code. Then the program is tested with real or simulated data on the emulated computer using all of that computer's resources for program debugging such as line printers, interactive terminals, graphics, and histogram and debug software packages. The working program can then be translated from the object code into the microprogram code of the microprocessor and when loaded into it, it should work the first time.

The reduction in overall effort can be quite substantial. For one thing, no software documentation for a microprocessor instruction set need be written or maintained. There would be no need to write an assembler or use a meta assembler. Simulators for a microprocessor are sometimes written to aid program testing, but no such simulators need to be written for a emulating microprocessor since the emulated computer serves this function. One should also realize that no retraining of programmers need be done which is important since most programmers only get very proficient with a particular machine language after they have much experience with it. The greatest advantage of all is that because of the emulation of another computer, the experimenters on the project can program the microprocessor themselves in a higher level language they understand, such as FORTRAN.

The emulation technique requires a restriction of the design of the microprocessor so that it best corresponds to the emulated computer, which is sometimes called the "target machine". The translator, which is the only special software which needs to be written, is a partner with the hardware design to produce the desired result. The net result is that one has a processor which is considerably easier to program yet is probably no slower in execution speed nor not much more costly then designs based on the same LSI microcircuits. They will be slower, however, than processors with a lot of dedicated high speed arithmetic units.

3. POST-PROCESSING MICROPROGRAM ASSEMBLY.

After assembling a program for a microprocessor on a host computer, one is ready to load it into the microprogram memory of the processor. This task is simple in concept but nevertheless requires some additional software which must be considered as part of the software for microcircuits.

The output from the assembly stage is a object code file containing the bit pattern to be loaded into the microprogram memory. It is in a form which is probably convenient for storage on the host computer such as a disk file. It must be transformed into the format for loading into the processor whose memory may be in the form of Programmable Read Only Memory (PROM) or as a writeable control store (RAM). It also must be transported from the host computer to the microprocessor.

The programming of PROMs is a software task which must have knowledge of the microinstruction format and the particular PROM circuit chosen. First, the "Don't Care" states which may still exist in the program must be changed to either a logic '0' or '1' since the memory circuit can only store one state or the other. Some PROM circuits invert or complement all of its outputs and in such cases one would need to complement the object program file before loading into the PROM. Both of these problems are relatively easy to take care of during program transport.

The number of bits available in PROM circuits is increasing all the time. Yet, in most cases one can not put all of the microprogram into one single circuit. For example, the microinstruction for the simple scanner processor (figure 33 of reference [1]) has a width of 13 bits and requires at least 6 words. A survey of the available memory circuits shows that the economical choice for the simple scanner is to use PROMs with 32 words of 8 bits. There simply does not exist on the market a memory circuit with 6 words of 13 bits. Thus the implementation of the instruction memory may be as shown in figure 9 . Here two memories each containing 8 bits have been placed in parallel so that the first has outputs for the first 8 bits and the second contains the remaining 5 bits. Three bits of the second PROM are not used and we are force to waste these bits. The same microprogram address is applied to both circuits. Since the next address logic of the simple scanner has an address field of only four bits, we have one address input line on both memories which is unused. In figure 9 we have tied that input to ground which forces that address line to '0'.

The simple scanner example is a case where the program is smaller but the program instruction width is larger than any available memory circuit. When this PROM is programmed it is inserted into a special circuit which applies the required voltages to blow the fuses within the circuit or deposits the charge in the cells of EPROMs. In either case, the programming circuit needs as input from the object code only the bits from the whole instruction word which will be placed into one PROM circuit. This means one of the post-processing steps is to take the object file and generate a programming file for each PROM which is required to contain the whole instruction word.

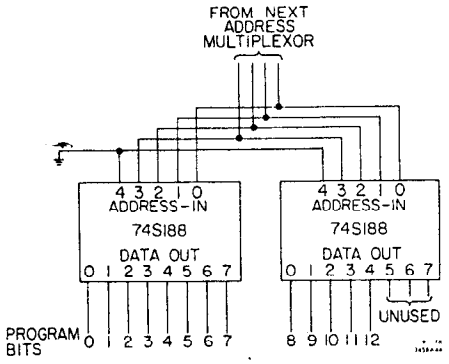

Figure 9: PROM layout for Simple Scanner.

Some microprocessors will have programs which are longer than the number of words available in PROMs. In this case, one would use multiple PROMs to contain the whole program as shown in figure 10 . In this figure, we have illustrated how one can use the same 32 word by 8 bit PROMs to make a program memory of 64 words by 16 bits. The low order 5 bits of the micro-program address are bused to each memory circuit. The most significant bit and its complement is generated so that a "Chip-Select" signal is sent to only one bank of PROMs at a time. Most memory circuits have a "Chip-Select" input which disables the output of the circuit when a False signal is received by the circuit. One can then tie the outputs of two circuits together to form what is called a "Wired-Or". Since only one memory bank is "Chip-Selected" at any time, the two circuit banks in figure 10 act as if they were one memory circuit of 64 words in length. The Wired-OR function is generally accomplished by having the outputs of the PROMs being either an Open Collector or a Tri-State output.

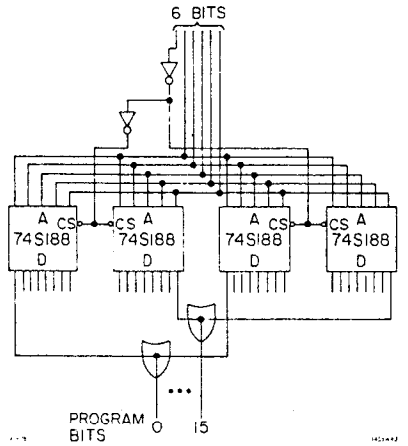

Figure 10: Expanding Memory Space.

Thus in transporting the microprogram object file to the PROM burner, we must reformat the file to take into account the width of each memory circuit, its position in the microinstruction word, and its depth in the memory address length. If the microprogram memory is to be implemented as RAM memory, then one has other needs in the transport of the file. An interface needs to be designed between the host computer and the memory bus of the microprocessor. In many cases, as will be discussed later, this interface may contain a micro or mini computer. In such cases, one must provide for the software for these two computers to communicate with each other and software which enables the computer directly attached to the microprocessor program memory to write into that memory. The software necessary is generally simple and straightforward. Yet it must be written, tested and debugged and as we all know even the simplest of software programs can sometimes lead to days or weeks of effort. All of this software should be considered as part of the software necessary for microcircuits even though it has nothing to do with the programs that run in the microprocessor. But once ths software is functioning properly, it may be reused for many different microprocessor projects.

4. SOFTWARE FOR TESTING MICROCIRCUITS.

Testing the microprogram may be a very difficult task, especially at the early stages of the project development when the microprocessor itself is not know to function properly. With ordinary random logic design if one does not get the correct results, then the fault must lie in the hardware. With ordinary micro or mini computers, if one does not get the correct result, then the fault must lie in the software. With microprogrammed processors, if one does not get the correct result, then the fault could be in either the hardware or software and one must try to isolate the problem. Thus, as well as carefully designing the processor and writing programs for it, one should also carefully design a means of testing the processor, and testing the programs that will run on it.

Let us consider for a moment, what the hardware faults in the processor might be. First of all there may simply be a logic design error, for example a circuit may not perform as expected because one did not read the specifications carefully enough. Secondly, there may errors in the fabrication of the processor, for example wires may be misplaced. There may be errors in the timing, for example, some results may be strobed into a register before they are ready. There may be "glitches", that is to say, noise pickup on some lines so that the wrong results are strobed into register or a clock input is generated at the wrong time. Although unlikely, there may even be some bad IC packages that need to be replaced with working ones.

4.1 TEST BOXES.

In the classical random logic design, one can usually "drive" the circuit with a pulse generator, and examine the functioning of the processor with an oscilloscope or logic probe. If the circuit has multiple input sources, one could build a test box to generate these inputs. The test bench setup would look like the one shown in figure 11 . If we try to apply this same testing technique to micro-processor, we will undoubtedly run into some problems. The microprocessor is "driven" by its program and the processor operates on data coming from or going to external devices or memory. Thus we have three separate subsystems as shown in figure 12 and each of the subsystems must be tested to see if they function properly.

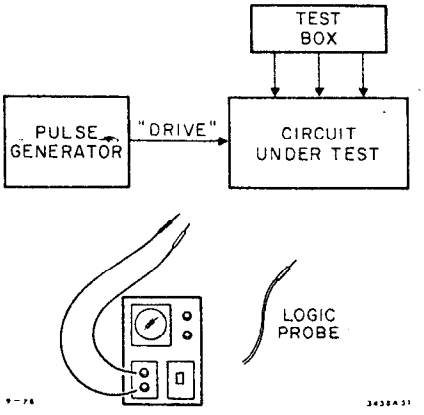

Figure 11: Traditional Random Logic Test Set-Up.

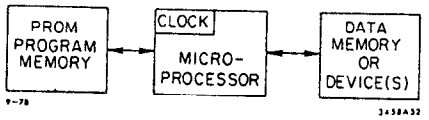

Figure 12: Microprocessor Subsystems to be Tested.

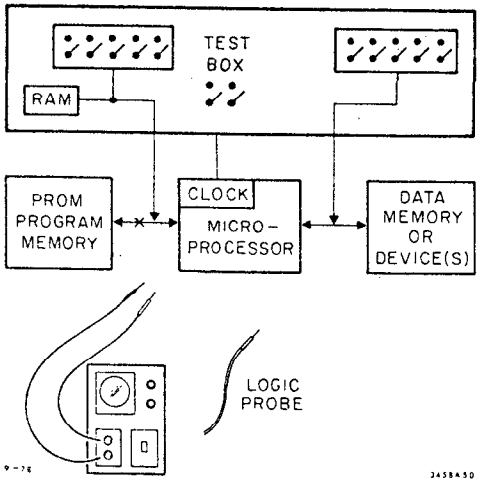

Figure 13: Microprocessor With Test Box.

One could follow the same approach as with random logic, that is by building a test box as shown in figure 13 . This test box can be more complex than the random logic test box, however. For example, if the microprocessor will eventually have a program memory in PROM circuits, the test box may contain a RAM memory so the user can put into it a variety of test programs. When using the RAM, one would disconnect the normal memory circuits from the microprocessor. Of course, some switches and lights need to be provided in order to read, write and modify the contents of the memory. Also the device or data memory may need to be simulated or at least a means of preloading a data pattern into them must be provided in order to test the processor.

A means of seeing what the processor has done must be provided as well as a means of controlling it. One could take as an example of the number of lights and switches that should be provided, those items that are used by a typical minicomputer. Table 2 lists what one finds on the front panel console of a PDP-11/20 minicomputer. Indicator lights are provided for the 18 address lines, the 16 data lines, and several other miscellaneous status conditions such as RUN, BUS, FETCH, etc. Sixteen switches are provided for entering data or address information which is controlled by the switches LOAD ADDRESS, EXAMINE, and DEPOSIT. The processor itself in controlled by the switches HALT/CONTINUE, SINGLE STEP, and START. These indicators and switches are about the minimal set that one could imagine in order to test programs on the PDP-11/20. A test box should at least contain these to test programs on a microprocessor. With a microprocessor, however, the test box might be even bigger than with a minicomputer because typically the program memory has a width greater than the data path and it is on a separate bus, so that the test box must have enough switches to handle both the program and data memory.

TABLE 2

PDP-11/20 Front Panel

INDICATOR LIGHTS:

- ADDRESS BUS (18)
- DATA BUS (16)
- RUN
- BUS
- FETCH

TOGGLE SWITCHES:

- DATA (16)
- LOAD ADDRESS
- EXAMINE
- HALT/CONTINUE
- SINGLE STEP
- DEPOSIT
- START

Such test boxes have been build in the past to control CAMAC. This approach has the advantage that the box can be specialized to the needs of a particular processor and it is a completely stand-alone system. On the other hand, the test box approach has certain disadvantages. It may have limited capability, since it itself is probably random logic. If one starts to add capabilities such as stopping the processor at certain address or loading the memory from some storage medium, the test box may be more complex than the processor itself. If the microprogram grows in size, it becomes extremely tedious and vulnerable to error to manually load the memory each time the power needs to be shut off to make a hardware change.

4.2 USE OF LOGIC ANALYZER.

Even with a very good test box, there are difficulties in testing a microprocessor with certain programs. For example, an error in either the hardware or software may cause the processor to jump to some unusually address and begin to do seemingly random operations that make it difficult to trace back to the source of the error. With an oscilloscope or logic probe one sees only one or two signals at a time and then only after the scope is triggered. If the processor halts after the error than one has only a single trace to see on the scope or one must use a storage scope.

A very useful instrument which aids in testing the microprocessor is a logic analyzer. It is an instrument which records in an internal memory the logic level (i.e. 0 or 1) of its input in fixed time intervals. Since with microprocessor circuits, we are always dealing with standard logic signals, it is usually sufficient to look only at the logic level rather than the real signal. The advantage of doing this is that an logic analyzer can be build with many more channels of input than an ordinary oscilloscope. Logic analyzers are available on the market with up to 16 independent input channels. The number of samples recorded is limited by the size of the internal memory and it is typically up to 1024 samples. The sampling rate, or inversely the time between the samples, is limited by the speed of the internal memory. With the analyzers available today, it is typically 20 nsec, and with some models 10 nsec. The analyzers generally have switch selectable thresholds for the standard logic families such as ECL and TTL. The analyzer does not, however, record short signals or glitches if they don't occur at the instance the signals are sampled. Some models have special input circuits called "glitch catchers" which take any transition in an input as the recorded level.

There are many other advantages of a Logic Analyzer over an ordinary oscilloscope. For example, a very important advantage is in triggering. With an oscilloscope, one can see the input signals for a time period after the trigger signal. Similarly, the logic analyzer can record its inputs after a trigger signal has arrived which is called the "Pre-Triggering" mode. But the analyzer can operate in the other sense, that it is, it can be continuously recording the input signals in a wrap around buffer and then stop recording when the trigger signal arrives. This mode is called "Post-Triggering" and it allows one to see all the input signals before the trigger signal. Some analyzers even allow one to put the triggering time in the middle of the memory storage so that one can see the input signals before and after the trigger time. Another important advantage is that once the analyzer is triggered, it can keep the recorded signals indefinitely, so that single shot events can easily been seen and studied.

The triggering abilities can be augmented by the use of a Word Recognizer which is frequently built into commercially available Logic Analyzers. This device allows the user to form his trigger on the combined state of many input conditions. The trigger can then be formed from something simple such as the transition of one signal input or as complex as particular bit pattern on the memory bus. Most available Word Recognizers accept up to 18 inputs to form the trigger and even allow one to make the trigger only after a number of occurrences of the same input pattern.

With most of the available Logic Analyzers, one has a choice of the way the recorded inputs are displayed. With an oscilloscope, one has only one mode. In this mode the signal levels are displayed vertically on the screen and time is displayed horizontally from left to right. Logic Analyzers can also display their data in this way and it is called the "Time Domain" mode. But they can also display their memory as data words with possibly a choice of binary, octal, or hexadecimal format. This mode is called the "Data Domain" and it is very useful for program development since it seems more like a program trace that the programmer is used to. Yet another mode is called the "Map" mode. It treats the input signals as one data word and plots a point on the screen for each possible data word. The most significant bits of the word form an displacement vertically while the least significant bits are used for a displacement horizontally. Thus one can get a feel of the flow of a program and the human eye can spot unusual events by points being very displaced from the normal pattern.

4.3 COMPUTER BASED DEVELOPMENT SYSTEM.

The logic analyzer greatly improves ones ability to find errors in the microprocessor even with a simple test box. But not all the problems are solved with it. One still has the problems of loading by hand the processor's memories and reading the results. Another approach is to use a micro or mini computer as the "test box". In this approach the design of the microprocessor would be made so that one could build an interface from a computer to the processor which allows the computer to gain access to the memories and various control points of the processor, for example the processor's clock. One can than emulate the test box functions with software in the computer. Such a setup is illustrated in figure 14 . The micro or mini computer in this setup is called the "Host Computer".

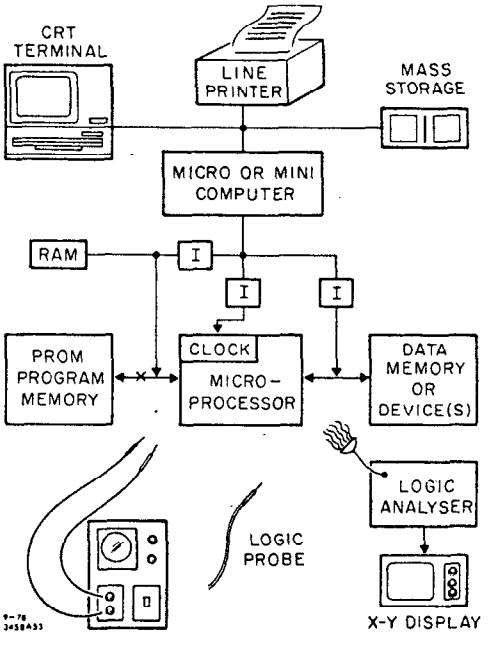

Figure 14: Computer Based Development System.

The use of a micro or mini computer for a test box may seem to be an expensive and overkill solution to a simple problem. But the cost of small computers is so low that they are probably not much more expensive then a test box with many indicator lights and switches. Through its software it offers many advantages over all but the most sophisticated test boxes. However, the software needs to written and debugged and one should consider this software as part of the software support for the microprocessor project.

Let us consider what software one might like to have for the host computer. The software can be organized as two set of programs, one set to move blocks of data as files and the other which allow the user to dynamically interact with the processor under test. First of all, one needs to be able to load the memory of the microprocessor from the memory of the host computer or some storage medium. The LOAD program may have a few options such as loading only a segment of the program or starting the load at some specified starting address. Since the processor's memory is also to be tested at some stage of the project, one should include a means of verifying that the memory was loaded correctly.

During the testing one would like to be able to examine and display individual words in the microprocessor program or data memory. The data should be formated for display on the host computer's terminal in a fashion that is easy to understand such as binary, octal, or hexadecimal or even a mixture in some applications. For example, the display program may recognize the fields in the microprocessor program memory and display them as mnemonics. One should also be able to modify individual words in order to correct program errors or to try different cases. All of the above features are available on most micro or mini computers that don't have front panels. For example, the DEC LSI-11 has a subset of the ODT program built into the processor to provide these functions. Finally, one should be able to read blocks of the microprocessor's memory and save it in the host computer or its storage medium. This would be useful after the program has been modified until it works and one would like to be able to reload it at a future date or use the saved file as input to a PROM burning program.

The next set of programs to be written on the host computer are programs to control the processor itself. The basic set are used to emulate the normal front panel console switches such as HALT, START, and SINGLE-STEP. One should also be able to load and display the microprogram counter and perhaps some other important registers. A very useful function would be to single step the processor and automatically display the contents of a register in order to trace where something went wrong. And finally, if one could cause the processor to stop when a certain address or data word is encountered, one would have all the capabilities that most computers have to trace down program or hardware difficulties.

The usefulness of computer based development systems has be recognized by many semiconductor manufacuturers. Some are offering for sale complete systems designed for microprocessor development. An example of such a system is the System/29 which is made by a company called Advance Micro Computers which is a partnership between Advanced Micro Devices and Siemens. This system is also described in a microcircuit specification book[3]. It includes a stand-alone 8080 microcomputer with floppy disks, CRT terminal, optional hardcopy printer, and a box in which one puts his own designed microprocessor. Prototype boards are available to simulate the eventual microprogram PROM with RAM and a standard next address logic board using a microsequencer. This system costs U.S. $25,000. This price includes the AMDASM meta assembler, an operating system for file manipulation in the 8080 microcomputer, and other software to aid in debugging the microprocessor under test.

The use of a micro or mini computer to be the controlling element for the microprocessor is not unique in the computer field. One can find similar examples in large computer systems. For example, the Amdahl 470/V6 computer has a Data General NOVA computer built into it's console display. The NOVA can read almost every register in the large machine. Another example is the Digital Equipment Corporation's VAX-11/780 in which an LSI-11 microcomputer handles the system terminal. In both computers the micro or mini computer handles the diagnostic routines and the system console. Neither machine has anything in the way we normally think of computer front panel consoles.

4.4 USE OF COMPUTER CENTER'S COMPUTER.

Given that one has a host computer that acts as the front panel for the microprocessor, one still has to consider on how one is going to store files and run the microprocessor's assembler program. This assembler program could well run on the host computer but not without some additional peripherals on this computer such as mass storage, and line printers. In High Energy Physics, most of the laboratories where one would be doing the microprocessor development work have a computer center and the question arises as to whether one should use the computer center for file storage and program development. In many cases a simple connection between the host computer and the computer center for the transfer of files is an approach which offers many advantages,

First of all, the peripherals are the most costly part of a computer system, especially with microcomputers since the CPU cost is very low. By using the computer center's peripherals one can greatly reduce the cost of the local computer system. Even with some local peripherals such as floppy disk, one generally has a more limited program development ability on a small micro or mini computer. One is also more likely to find useful cross assembler software available for the computer center computers than the inexpensive local computer.

All of the above advantages are obvious at first glance and may very well justify using the computer center for microprocessor program development. But there are also many hidden advantages which may be equally important from the point of view of hardware costs and more important in terms of man power costs. First of all, one can reduce the cost of the local computer to the bear minimum. This could mean that it would consist of only the CPU, some RAM memory, an interface to a terminal, an interface to the computer center, and a some ROM memory to get started. Since there will be no moving parts, the cost is only in inexpensive components. The reduction on peripherals also leads to a substantially reduced maintenance cost. Very little space in the laboratory would be required for the system which allows more room to work on the microprocessor or allows more flexability on where to place the equipment.

There is also a very large reduction of time consuming tasks for the personal involved in the project. All the facilities of the computer center

are presumably already known so there is no lost of time to retrain people with a new system. The computer center most likely has a better text editing system than those found on small systems. There will also be a file management system with much more space available to the user. Included with the file system will a routine data management system, that is, a system where by files are archived and backup tapes are regularly made. When batch jobs are to be submitted to do cross assembly work one can use the job entry system which is already well know including all the job control language that is necessary. The computer center will have many more peripherals and more kinds of peripherals than one would think of putting on a low cost local computer system. They may include high speed line printers, microfiche printers, graphic plotters and terminals, etc. One has global access to the computer center from any terminal in the laboratory rather than the one terminal for the local computer which allows more flexibility in working. The computer center is also set up to be multi-user, so that more than one person can work at a time on software for the microprocessor project and yet they can share the same data base files.

The cost of the connection between the local computer and the computer center can be minimized when one realizes that the data transmission rate need not be very high. For example one can let the local computer emulate an ordinary terminal from the point of view of the computer center. Data files could then be prepared before transmission with certain keyword characters so the local computer can take the characters as data rather than repeating them on its terminal. In this way no software for the connection need be written for the computer center's computer and the local computer need only have an ordinary terminal interface to the center and a very small amount of software. Such a simple connection was used for the local computer for the 168/E test bench shown in figure 15 . The terminal emulation program located in ROM is only 256 words in length.

Many laboratories have or are developing computer networks which would allow high speed data transmission between the computer center and the local computer. They offer better facilities in bringing down large programs with error checking and correcting on the transmitted data.

5. SUMMARY

We have seen that there are two parts of the software for microcircuits. The first part is the programs for the microprocessors, while the second is the software support programs. The support software can turn out to be much more extensive than the processor's programs. This does not mean that the user of microprocessors will drown in a ocean of support software, but he must simply learn how to swim in deep water.

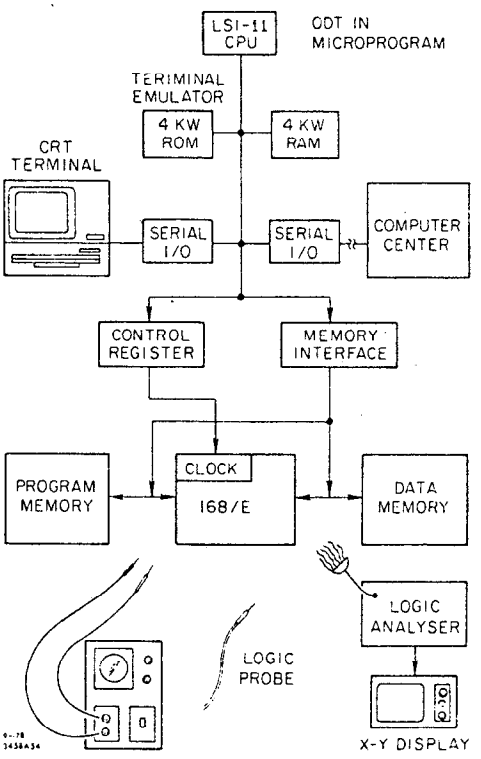

Figure 15: Test Bench for 168/E Project.

References

[1] Paul F. Kunz, "Microcircuits of High Energy Physics", Proc. of the 1978 CERN School of Computing, Jadwsin, Poland (May-June, 1978).

[2] Zbigniew Guzik, Private Communication.

[3] The Am2900 Family Data Book, Advanced Micro Devices, Inc., Sunnyvale, California 94086 (1978).

[4] Edward Frank, Private Communication.

[5] V. Michael Powers and Jose H. Hernandez, "Microprogram Assemblers of Bit-Slice Microprocessors", Computer, pp 108-120, (July 1978).

[6] W. Wimmer, DESY Report DV-78/04(1978), in German.

[7] Tom Droege, Transactions of I.E.E.E., NS-25, p 698,(1978).