D0 Microcode User's Guide

Document: Unassigned
Date: June 1978
Version: 1.0

This guide is intended to provide a programmer's look at the D0 and at Micro, the assembler. Each section examines an isolated part of the machine, and contains examples of efficient coding techniques. Familiarity with the D0 Functional Specification and the Micro documentation is assumed. All comments should be addressed to the editor via Laurel.

Approval:

Carol Hankins

Carol Hankins, Editor

[illegible]

W. B. Kennedy, Manager

XEROX

SYSTEMS DEVELOPMENT DEPARTMENT
3408 Hillview Avenue / Palo Alto / California 94304

Table of Contents

Opening comments

You will never get to be a super microcoder until you memorize and understand the architecture of the machine. It is assumed that you have made some attempt to understand the D0 Functional Specification, and that you are now ready to program. This manual breaks the machine into small parts and describes the use of each section in detail. Accompanying each part are some examples of microcode which illustrate the features of the machine.

All numbers in this manual should be considered octal. When decimal is required, the number will be suffixed with a "D". Any number followed by a "B" is octal.

1.0 The ALU: Its Inputs and Output

Look at the circled part of the picture at the end of this manual and find the ALU. There are two inputs, labelled A and B. Notice where each come from. There is one output which goes back to T and R. The A input comes from the R registers through the cycler/masker. Also on this bus are the special-purpose R registers, such as APC, PCF... The B bus comes from T. Notice the way constants are put on the bus. You can have a constant *or* T - not both. Constants are eight bits - all of which must be contained in the left or right half of the word.

Next, find the signal coming in to the top of the ALU, ALUControl. There are two ways of controlling the ALU's operations: from the ALUF field of the microinstruction, or from a special box called SALUF. The ALU is really a unit with 64 operations, with 14 of the most common mapped into the ALUF field of the microinstruction. All 64 functions may be accessed by loading SALUF.

Here are the ALU operations:

ALUF	ALUOut =
0	B
1	A
2	A AND B
3	A OR B
4	A XOR B
5	A AND NOT B
6	A OR NOT B
7	A XNOR B
8	A+1
9	A+B
10	A+B+1
11	A-1
12	A-B
13	A-B-1
14	unassigned
15	use SALUF for ALU function

The cycler/masker is used to manipulate bits from an R register. This provides some standard shifting and masking operations. The following operations are provided:

LDF[r, pos, size] - right justify the field in r of length "size" beginning at bit "pos"
RSH[r, count] - right shift r by "count"
LSH[r, count] - left shift r by "count"
RCY[r, count] - right cycle r by "count"
LCY[r, count] - left cycle r by "count"
RHMASK[r] - r AND 377C
LHMASK[r] - r AND 177400
ZERO - a way to load a 0 on the bus
DISPATCH[r, pos, size] - see the section on the jump condition

Already you know how to write simple microinstructions for manipulating the ALU. Here are some examples of legal and illegal instructions:

Legal

```
T ← (R) + T;                  * an A input and a B input
R ← (R) + T;                  * can store into R or T
T←, R← (R) + T;               * can store into both
T ← LDF[R, 14, 4] + T;        * LDF is a cycler/masker function
R ← RHMASK[R] XOR T;          * RHMASK is a cycler/masker function
```

```
    R ← (R) + (377C);              * constant with lower 8 bits
    R ← (R) + (177400C);           * constant with upper 8 bits
    T ← (zero) + T + 1;            * this is the only way to add 1 to T.  zero is an
                                   * output of the cycler/masker, and A+B+1 is
                                   * an ALU function..
```

Illegal

```
    T ← T + (37C);                 * two B bus sources
    R ← R + (177777C);             * constant is more than 8 bits
    R ← R + (770C);                * the 8 bits cross a byte-boundary
    T ← LDF[R, 14, 4] + (37C);     * LDF uses F field and so does constant
    T ← RHMASK[T];                 * T is not on the A bus
    R ← T + 1;                     * no ALU function of this type
```

2.0 The Microinstruction

Since you would like to do more than arithmetic and logical functions, let's look at the non-memory microinstruction for other fields that you can use. The fields are:

NORMAL	0 or 1 - depends on if it's a memory operation or not
RMOD	used for addressing the special R registers
RSEL	used for R addressing. **NOTE: only 1 R address per m-i.**
ALUF	what the ALU is supposed to do
BSEL	what is supposed to be on the B bus (T or constant)
F1	special function
LR	load R
LT	load T
F2	special function
JC	jump control - call, goto, return, dispatch
JA	where to go next. **NOTE: addressing is 8 bits =>page-relative**

You know about RSEL, ALUF, BSEL, and the loading of R and T. Let's first discuss the branching mechanisms and the control logic of the D0. These use the JC and JA fields. Each microinstruction must indicate where to go next. If you do not instruct otherwise, a microinstruciton will be followed by the successor instruction in your program. You can modify this in many ways. A simple GOTO[label] will cause the JA field to contain the address of "label".

Conditional Branches

For the programmer's convenience, several branch conditions exist, and alter the flow of control when tested. There is a programming feature called DBLGOTO which has the form DBLGOTO[label1, label2, branch-condition]. If branch-condition is true, control will be transferred to label1, if not the next instruction will be label2. The processor requires that these two labels be one bit apart in their address. These are guaranteed to get you into trouble if you do not remember the instruction-placement constraints. The table below describes the placement constraints for "label1". Label1 will occupy an odd location if the condition is listed in the goes-to-odd column below:

JC,,JA	goes-to-odd	goes-to-even	BRANCHSHIFT	time	page
000	ALU#0	ALU=0	0	t3	18
001	CARRY	NOCARRY	0	t3	18
010	ALU<0	ALU>=0	0	t3	18
011	QUADOVF	INQUAD	0	t3	45
100	R<0	R>=0	0	t1	
101	R ODD	R EVEN	0	t1	
110	NOATTEN	IOATTEN	0	t3	62
111	MB	NOMB	0	t1	24
000	MPCARRY	NOMPCARRY	1	t3	
001	NOOVF	OVF	1	t3	18
010	BPCCHK	BPCNOCHK	1	t3	
011	ALU=<0	ALU>0	1	t3	

Note: GOTO[label1, branch-condition] is a degenerate case of DBLGOTO with label2 = current location + 1.

Note the "time" column. This is the time that this condition is available for testing. If t3 is listed, you should test this condition in the instruction *following* the instruction which could generate the condition. Conditions listed as t1 can be tested during the current microinstruction.

The BRANCHSHIFT column deals with special functions (in particular, F1), and will be discussed fully in the section on Special Functions.

The page column refers to the page in the D0 Functional Specification where more information about these conditions can be located.

Following are examples of instruction placement:

```
T ← (R) + (377C);
DBLGOTO[L1, L2, ALU#0];         * notice test during instruction following operation
mumble;
L1:  mumble2;                   * at an odd location (L2 OR 1)
L2:  mumble3;                   * at an even location

T ← (R) - T;
DBLGOTO[L1, L2, ALU>=0];
mumble;
L1:  mumble2;                   * at an even location
L2:  mumble3;                   * at an odd location (L1 OR 1)

DBLGOTO[L1, L2, R<0], LU ← R;   * notice test during instruction
mumble:
L1:  mumble2;                   * at an odd location (L2 OR 1)
L2:  mumble3;                   * at an even location
```

Subroutine Calls

There is a mechanism for one-level subroutines. These are accomplished by an instruction of the form CALL[label]. When a RETURN is executed, control will be given to the call instruction+1. CALLs must be on even words.

Example:

```
SUBR: R ← (R) + T + 1;
      T ← RHMASK[R], RETURN;          * next instruction will be "MUMBLE"
INIT: R ← (4C);
      T ← (37C);
      CALL[SUBR];
      MUMBLE;
```

Dispatch

DISPATCH is a cycler/masker function which allows the next instruction to be one of sixteen possible addresses. The lower four bits of APC are selected via DISPATCH[r, pos, size]. A trivial example of dispatch is as follows:

```
D: DISPATCH[RTMP, 12, 4];                 * dispatch on bits 12:16 of rtmp
   DISP[D0];                              * set up label for dispatch
   SET[DLOC, 20];                         * nail down the dispatch table

D0:    GOTO[X],  RNEXT ← (0C), AT[DLOC, 0];   * these don't all have to be
adjacent
D1:    RNEXT ← (1C), AT[DLOC, 1];
   GOTO[X],  t ← LDF[RNEXT, 3, 1];
```

```
D2:     GOTO[X],  RNEXT ← (2C), AT[DLOC, 2];
D3:     GOTO[X],  RNEXT ← (3C), AT[DLOC, 3];
D4:     GOTO[X],  RNEXT ← (4C), AT[DLOC, 4];
D5:     GOTO[X],  RNEXT ← (5C), AT[DLOC, 5];
D6:     GOTO[X],  RNEXT ← (6C), AT[DLOC, 6];
D7:     GOTO[X],  RNEXT ← (7C), AT[DLOC, 7];
D10:    GOTO[X],  RNEXT ← (10C), AT[DLOC, 10];
D11:    GOTO[X],  RNEXT ← (11C), AT[DLOC, 11];
D12:    GOTO[X],  RNEXT ← (12C), AT[DLOC, 12];
D13:    GOTO[X],  RNEXT ← (13C), AT[DLOC, 13];
D14:    GOTO[X],  RNEXT ← (14C), AT[DLOC, 14];
D15:    GOTO[X],  RNEXT ← (15C), AT[DLOC, 15];
D16:    GOTO[X],  RNEXT ← (16C), AT[DLOC, 16];
D17:    GOTO[X],  RNEXT ← (17C), AT[DLOC, 17];
```

Changing Pages

As noted above, a micro-instruction does not know what page it is on, and can only jump to addresses on its current page. There are ways to circumvent this at assembly-time and at run-time.

Assembly-time

ONPAGE[xx] - directs assembler to put this instruction on page xx.
AT[nn] - the assembler assumes that it has been given a 12 bit address, and puts this instruction on the page indicated by the top four bits, and the offset of the last eight bits.

Run-time

LOADPAGE[n] - this is to be done before *every* branch that will be on a different page. This includes GOTO, DBLGOTO, CALL, DISPATCH.

Notify

When one wants to jump to a specific location in a specific task, APCTASK&APC are loaded with the desired information, and a RETURN is executed.

```
        R ← (20C);               * set up the location you want to get to
        R ← (R) OR (160000C);    * OR in the task number = 16
        APCTASK&APC ← R;
        RETURN;

L1      T ← (0C), AT[20];
```

After execution of the first block of code, control will be transferred to L1 with task 16 active.

3.0 Functions

The F-field decodes are as follows:

CODE	F1	F2	GROUP B
00	BBFA	REGSHIFT	unused
01	RS232←	STKP←	RESETERRORS
02	LOADTIMER	FREEZERESULT	INCMPANEL
03	ADDTOTIMER	STACKSHIFT	CLEARMPANEL
04	unused	CYCLECONTROL←	GENSRCLOCK
05	LOADPAGE	SB←	RESETWDT
06	unused	DB←	BOOT
07	GROUP B	NEWINST	SETFAULT
10	no-op	BRANCHSHIFT	APC&APCTASK←
11	WFA	SALUF←	RESTORE
12	BBFB	no-op	RESETFAULT
13	WFB	MNBR←	USECTASK
14	RF	PCF←	WRITECS0&2
15	BBFBX	RESETMEMERRS	WRITECS1
16	NEXTINST	USECOUTASCIN	READCS
17	NEXTOP (NEXTDATA)	PRINTER←	unused

Functions can be best be explained by division into categories. Following each group name will be the pages in the D0 Functional Specification where more information in available. The only function groups expected to be of use to the programmer are: Useful, ALU, and Sneaky. The rest of the functions should remain unused by most code. In addition, registers used for the Mesa emulator and BitBLT should be avoided.

Useful functions:
APCTASK&APC← - used to directly load this register from ALUA. This is the recommended way to do a *notify* of a different task at a different location. It is usually followed by a RETURN.
USECTASK - forces the next instruction to be taken from the current task. This usually preceeds a RETURN, and prohibits task switching.
LOADPAGE - uses F2 for an argument. This statement should preceed all CALLs or GOTOs which reference a different page.

ALU functions (p. 17-18):
FREEZERESULT - inhibits loading of RESULT register. This is used to save the output of the ALU from one instruciton to the next.
USECOUTASCIN - use carry out as carry in.
SALUF← - can expand the ALU to its full capabilities.

Sneaky functions: THESE GET SET WITHOUT YOUR KNOWLEDGE!!!!!!!
REGSHIFT - Set by accessing certain R registers (PRINTER, DB, SB, MNBR) and invoking BBFB.
STACKSHIFT - Set when operations like STACK&+1 are used
BRANCHSHIFT - Set by certain branch conditions: MPCARRY, NOOVF, BPCCHK, ALU=<0.

BitBLT (p. 22):
BBFA - sets up 3 bit dispatch based on SB, DB, MNBR for the next instruction
BBFB - update of the x level from MW. SETS REGSHIFT!
BBFBX - update of the main level from MW
SB← - loaded from A bus
DB← - loaded from A bus
MNBR← - loaded from A bus

Mesa (p. 20-22):
- STKP ← - loaded from A bus
- WFA - Mesa Write Field
- WFB - same
- RF - Mesa Read Field controlling the cycler/masker directly from field descriptor
- NEXTINST
- NEINST
- NEXTOP
- PCF← - loaded from A bus
- CYCLECONTROL← - loaded from A bus. This is a way for controlling the cycler/masker. It also loads SBX[0:5], DBX[0:1].

Auxiliary Registers (p. 27-30):
- RS232← - from B bus
- PRINTER← - loaded from A bus

Modification of Control Store (p. 39-40):
- WRITECS0&2 - preceeded by APCTASK&APC ← x, where x is the address you want to write in. Word 0 is written from A bus, word 2 from B bus.
- WRITECS1 - item on A bus written to word 1 in location in APCTASK&APC
- READCS - reads a word from the control store. For the values 0, 1, 3 of T, you get word 0, 1, and 2 from control store into CSData.

Timer functions (p. 30-33):
- LOADTIMER - loads a timer from ALUA; bits[0:3] state, [4:11] data, [12:15] slot
- ADDTOTIMER - increments a timer from ALUA

Maintenance panel:
- CLEARMPANEL - clears the maintenance panel
- INCMPANEL - increments the maintenance panel

System functions:
- RESETERRORS (p.38) - clears the freeze on CIA, RESULT and resets PARITY and the fault logic
- RESETMEMERRS - clears the memory error logic
- RESETFAULT -
- RESTORE (p.38) - loads RESULT regiser from H2, loads APCTASK&APC from ALUA. This is to restore the machine state after a FAULT.
- GENSRCLOCK - clock out bits to the IO controllers
- RESETWDT (p.27) - reset Watchdog timer
- BOOT - initiate a software boot
- SETFAULT - cause a fault to occur

4.0 Memory Use

This section is an amplification and correction of the D0 Functional Specifications of January 16, 1978. Its purpose is to provide an interim guide to the proper use of memory and IO operations. You should assume that any topic not covered here is considered correct in the manual. If you follow the guidelines listed, you should not run into any trouble. Ignoring them will get you into funny situations which involve timing problems, interlocking and bypassing. The three main topics to be discussed here will be quadword alignment, bypassing, and the memory interlock feature.

Words and phrases in italics are meant to convey a special meaning. If one wanted to avoid trouble, verbs should be read as "must".

Comments on style

Since the memory operates in parallel with the processor, there are certain hardware features that have been provided that will prevent you from accessing a location which is an operand in the memory operation which is running concurrently, if these features are used correctly. When an instruction following a memory operation attempts to use data from that operation, the instruction aborts (this means that time freezes until the operation is complete). Efficient microcoders will not write code in this manner, but will use the cycles between a memory operation and use of the data for other necessary code. A forthcoming section will list the maximum execution time for memory and IO instructions.

Quadword alignment

Memory operations dealing with transference of more than one word *should* adhere to double- or quadword alignment. The memory instruction has two fields involving R registers: the base register field and the SRC/DEST field. The base registers *should* be an even and odd word pair. The even word is the page and displacement, and the odd word contains the upper bits of the virtual address.

> If the base pointer is denoted BP[0:23], bits 0:7 in the odd register hold BP[0:7], and bits 8:15 hold the BP[0:7]+1. **This is incorrect in the manual.**
>
> You should also remember the caution in section 5.5 about bits 0,1 and 8,9 of the base register. For safe use, these four bits should be set to 0 by the programmer.

In general, you *should* always have aligned registers and memory. You can transfer data to a non-quad (or double) aligned R register, but it will defeat the interlock(see below). If the memory is not quad or double aligned, the memory will pick the smallest outer bound matching your request and transfer those words to you. For example, executing a PFetch4 with a memory address of 2 will **not** give you words 2,3,4, and 5. It will give you 0,1,2, and 3.

> Do not use register 0 in any task block for a SRC/DEST. This forces use of the stack.
>
> The signal QUADOVF is generated <u>only</u> in the following situation: The stack is used for a PFetch2 or PStore2 with memory address equal to 3 mod 4. This is **not** a general signal which occurs whenever you cross a quadword boundary.

Bypassing

A non-memory instruction is broken into four cycles: cycle 0 reads the R registers or T, cycles 1 and 2 are taken by the ALU operation, and cycle 3 writes R or T. Since another instruction begins at the beginning of cycle 2, data needed for this instruction will not have been written when the read occurs. The hardware notices this, grabs the needed data for the current instruction, and does the write during cycle 1. The bypass is only good for the following instruction. Bypassing only allows data to be used from one instruction to the next; it does not imply storing.

If the instruction following a store is a memory instruction, the write will be delayed for another two cycles. This means that the store will not take place until cycle 1 of the instruction following the memory operaton. As an example, consider a sequence of three instructons, the middle one being the memory operation. A memory instruction reads R registers in cycle 0 for bits 8:23 of the virtual address, and in cycle 1 for the upper bits. Since the R memory cannot be read and written in the same cycle, the second read required by the memory operation forces the write of instruction 1 to occur in cycle 1 of the third instruction. The bypass of data from instruction 1 to instruction 2 will work, and give data to the memory operation for its cycle 0 read, but not its cycle 1 read. This is why you can load an even base register before a memory operation, but not an odd base register.

```
  Read                    Write-can't take place because of read for memory
/-------/-------/-------/-------/
                  Read    Read
                /-------/-------/
                                  Read   Write - from above is done here
                                /-------/-------/-------/-------/
```

Memory Interlock

The memory interlock is provided to protect you from accessing data which may not have been operated on by a preceeding memory operation. Use of quad or double word aligned registers will make this work smoothly; nonaligned registers defeat the interlock. The actual R register address is compared (with appropriate low order bits omitted if the operation is double or quad) with R addresses in MC1 and MC2, and the instruction is aborted until the memory is finished. If you like to gamble, you can use non-aligned registers and access those protected by the interlock in the next instruction, but wait until some time later to access the other registers.

Examples

Proper use of the memory will look like the following:

```
RV[rbaseEven, 10];
RV[rbaseOdd, 11];
RV[rsrc2, 12];
RV[rsrc4, 14];
RV[rbaseEven2, 20];
RV[rbaseOdd2, 21];
RV[rtmp];

1.      rbaseOdd ← value1;              * set up Odd register first
        rbaseEven ← value2;             * set up even register - bypass will get proper
        PFetch2[rbaseEven, rsrc2];      *     value to mem op in cycle 0
        t ← rsrc2;                      * when memory done, this instr will be executed

2.      rbaseOdd ← value1;
        rbaseEven ← value2;
        PFetch4[rbaseEven, rsrc4];
        t ← rsrc4;

3.      rbaseOdd ← value1;
        rbaseEven ← value2;
        PFetch2[rbaseEven, rsrc2];
        rbaseOdd2 ← value3;          * this could also be rbaseEven2 ← mumble
        rtmp ← t;                    * need this to be sure store is accomplished
        PFetch2[rbaseEven2, rsrc2];
```

Improper use of memory:

```
1.      rbaseEven ← value2;
        rbaseOdd ← value1;           * mem op needs this in cycle 1, but bypass
        PFetch2[rbaseEven, rsrc2];   *    only works for cycle 0

2.      rsrc2 ← value3;              * set up a register to be stored
        PStore2[rbaseEven, rsrc2];   * rsrc2 will not have been written when this
                                     *    begins  (note 1, page 47)

3a.     rtmp ← value3;               * this will not work because of the bypassing
        MemOp[rbaseEven, rsrc2];     *  mentioned above.  Writing of rtmp is in cycle
        t ← rtmp;                    *  1 of this instruction

3b.     t ← value3;                  * same reasons as 3a.
        MemOp[rbaseEven, rsrc2];
        rtmp ← t;
```

5.0 Getting Started

Most of the information which you will need will be present on Iris. We have a directory called D0. This is the first place you should go and look for any programs or documentation that you need. There is also a microcoder's distribution list which is on [maxc]<secretary>d0users.dl. You will receive notification of new programs or updates via this distribution list.

There are two files on [iris]<D0>which can be used to create a microcoder's disk. If you have a virgin disk, you should obtain a copy of <alto>newdisk.cm from your local file server. After running this, get [Iris]<D0>newmidasdisk.cm or <D0>newsimdisk.cm. This will give you all files you need to use for microcoding. For a disk already containing an operating system, FTP, Chat, Bravo, and other basic programs, you need to run [Iris]<D0>midasdisk.cm or <D0>simdisk.cm. This will provide you with enough Mesa to run the simulator, and all needed microcode files.

The first document to be read is the D0 Processor Functional Specification, an old copy of which is on [maxc]<Thacker>manual.press. This explains the hardware and also gives you pictures of the architecture which are useful to look at while coding. After reading this, you should get the D0 Assembler manual ([maxc]<D1Docs>d0assem.ears) to familiarize yourself with the microcode syntax. At this time, you should be able to write a simple program.

Given that you've now written a program, you need to assemble it. Actual assembly is accomplished by two programs: Micro and MicroD ([iris]<d0>micro.run, microd.run). Micro is the main assembler; MicroD's function is instruction placement in the microstore. Micro is a very general microcode assembler, and it accepts language features from a file called D0lang.mc ([iris]<d0>D0lang.mc). This file is assembled with each of your microcode files. If your file is named Test, you would assemble it in the following manner:

```
Micro D0lang Test
```

Assuming you got no errors, you would then proceed with

```
MicroD Test
```

At this time you have a file called Test.mb which is ready for loading into the D0 or for use with the simulator.

Since we don't have a multitude of D0's yet, we need to be able to debug our programs on an Alto. This is possible via the D0 Simulator ([iris]<d0>s.bcd). There is a very readable document on how to use the simulator, [iris]<d0>s.press. The simulator closely tracks the D0 and any changes made to it. Once your program runs through the simulator, you can be very confident that it will work on the D0. The simulator has a feature for running in non-overlap mode which is most useful for debugging.

On the D0, microcode programs will be run and debugged with a program called Midas. Midas has its own documentation which you should read on [iris]<d0>midas.press. The Midas system is in the form of a "dump" file and is on [iris]<d0>midasrun.dm. If most of the information in this memo is new to you, don't bother getting into Midas yet.

6.0 Caveats

You must execute a TASK function every 12 microinstructions to insure that data from higher priority devices is not lost.

NEVER use more than sixteen R registers for a given task.

If you are writing microcode which will be incorporated into a release, you must "check out" a prefix from me. This prefix will occur before your labels and register names.

Anyone who does not follow the above rules will receive no help from me whatsoever.

7.0 Suggested programming style

It is highly unlikely that you will be the only person reading your code, so below are some suggestions which will make your fellow coder's life easier.

As mentioned in the Caveat section, if this piece of code will ever be in a microcode release, you must check out a prefix from me. This prefix is to be used in front of *all* R register names and labels. Given that they all begin with this prefix, they can still be named something which suggests their function. It is possible to define many names for a particular R register (by executing as many RV's as are necessary), and if your code can be sectioned in a reasonable manner, you may want to try this technique.

If you use names which are a concatenation of two or more syllables, you might consider using lower case letters and having the next syllable begin with upper case. This produces quite readable text. If you use lower case, you must call Micro with the "/u" switch on the command line.

The micro-assembler, Micro, makes it quite easy to define constants and assign English-like names to arbitrary sets of bits. There are two facilities for accomplishing this. The macro MC[name, number] defines a constant; i.e., every time the assembler finds "name", it substitutes the number appended with a "C". SET[name, number] will give you the number without a "C" which is suitable for use as a parameter.

Examples:

```
MC[bitMask, 200];         * used for expressions like RTMP←(RTMP) OR (bitMask);
MC[sectorMask, 16400];       * T←(DiskAddr) AND (sectorMask);

SET[myPage, 3];           * use this as a parameter as in ONPAGE[myPage];
SET[dispLoc, 200];        * DISP[dispLoc];
```

It should be noted that constants formed in this manner must still adhere to the eight bit limit discussed in the section on constants.

It is suggested that you begin each of your modules with a SETTASK and an ONPAGE. Parameterization will make these easy to change later on. You should also begin your modules with a "notify" to get you to the proper task and location for the start of your code.

8.0 Sample programs

```
file: [iris]<d0>sample.mc
This file consists of sample programs, which are each prefaced
with what I hope they will illustrate. The sections can each
be broken out (code between TITLE and END) and be assembled
and run through the Simulator, if you wish.

TITLE[Sample1];
* This code takes the number in R register RNum and multiplies
* it by 10. This is accomplished by multiplying it first by
* 8, multiplying a copy of it by 2, and adding the results.

RV[RNum];
RV[RTemp];      *just a temporary register

INIT:  RNum ← (4C);     *initialize it
START:    T ← RNum;    *need to copy it into T to get it to RTemp
    RTemp ← T;   *RTemp = RNum
    RTemp ← LSH[RTemp, 3];   *RTemp = 8*RNum
    RNum ← LSH[RNum, 1];     *RNum = 2*RNum
    T ← RTemp;   *put in T so we can add them
    RNum ← (RNum) + T;
    GOTO[START];
END.

%
Now we try and make the above a bit more efficient.
%

TITLE[Sample2];

RV[RNum];
RV[RTemp];

INIT:  T ← RNum ← (4C);    *loading T is free
TIMES10: RTemp ← T;
    T ← RTemp ← LSH[RTemp, 3];
    RNum ← (LSH[RNum, 1]) + T; *shifting is on A-bus
    GOTO[TIMES10];

END.

%
Moving right along, let's look at branching. The important
things to remember about branching are that ALU conditions
are available at t3 (after cycle 2) and are saved, while R
conditions are available at t1, and are destroyed after this
time.

In the next program, we're going to use two subroutines. GETVAL is
totally mythical - assume it gets a number from somewhere and puts
it in T. TIMES10 is the above code made into a subroutine. The
following program reads a count via GETVAL, then calls GETVAL to
give it numbers which it makes positive if they aren't, and then
multiplies them by 10. When finished with that loop, it goes back
up to get another count.
%
```

```
TITLE[Sample3];

RV[RNum];
RV[RCount];
RV[RTemp];

START:  CALL[GETVAL];
    RCount ← T;
    GOTO[DONE, R<0], LU ← RCount;    *way to put something on bus
* could have tested on T above via
*   GOTO[DONE, ALU<0];

AGAIN:  NOP;              *see below for explanation
    CALL[GETVAL];
    RNum ← T;
* again, could have tested on T as above
    GOTO[MULR, R>=0], LU ← RNum;    *if it's positive, jump
    RNum ← (RNum) XOR (100000C); *make it positive
MULR:   CALL[TIMES10];
    RCount ← (RCount) - (1C); *decrement count      DBLGOTO[AGAIN, DONE,
ALU#0];
DONE:   GOTO[START];

TIMES10: T ← RNum;
    RTemp ← T;
    T ← RTemp ← LSH[RTemp, 3];
    RETURN, RNum ← (LSH[RNum, 1]) + T;

END.


%
Many errors can be avoided by understanding the branching logic.
CALL's always have to be at even locations.  DBLBRANCH and DBLGOTO
go to odd locations if true, and even if false.  The DBLGOTO
which is right before the label DONE is supposed to go to AGAIN
if true, and DONE if false.  At AGAIN, we really want to do a
CALL[GETVAL], but since the branching logic dictates that AGAIN
be placed at an odd location, we have to put in a NOP.
%
```

9.0 Error Messages

Micro occasionally produces rather baroque error messages. The following are the ones most commonly received when beginning:

RREGISTER+B Undefined - a missing set of parentheses around the "A" field of the the ALU function in the instruction. This comes from a statement like T ← RTEMP + (1C), where the above message would be RTEMP+B Undefined.

Field RSEL2 already used - this usually results from referring to two R registers in the same statement. There is only space for one in the micro-instruction. RTEMP ← (RADDR) + (T) is illegal.

Illegal constant - a constant in a microinstruction can only be 8 bits, either the upper or lower 8. If you need a constant which is longer you need to do it in two instructions.

T+B Undefined - you are trying to put two things on the B bus. Look at the diagram of the D0. An instruction of the form T ← T + (377C) is not possible, since T is on the B bus, and so is the constant.

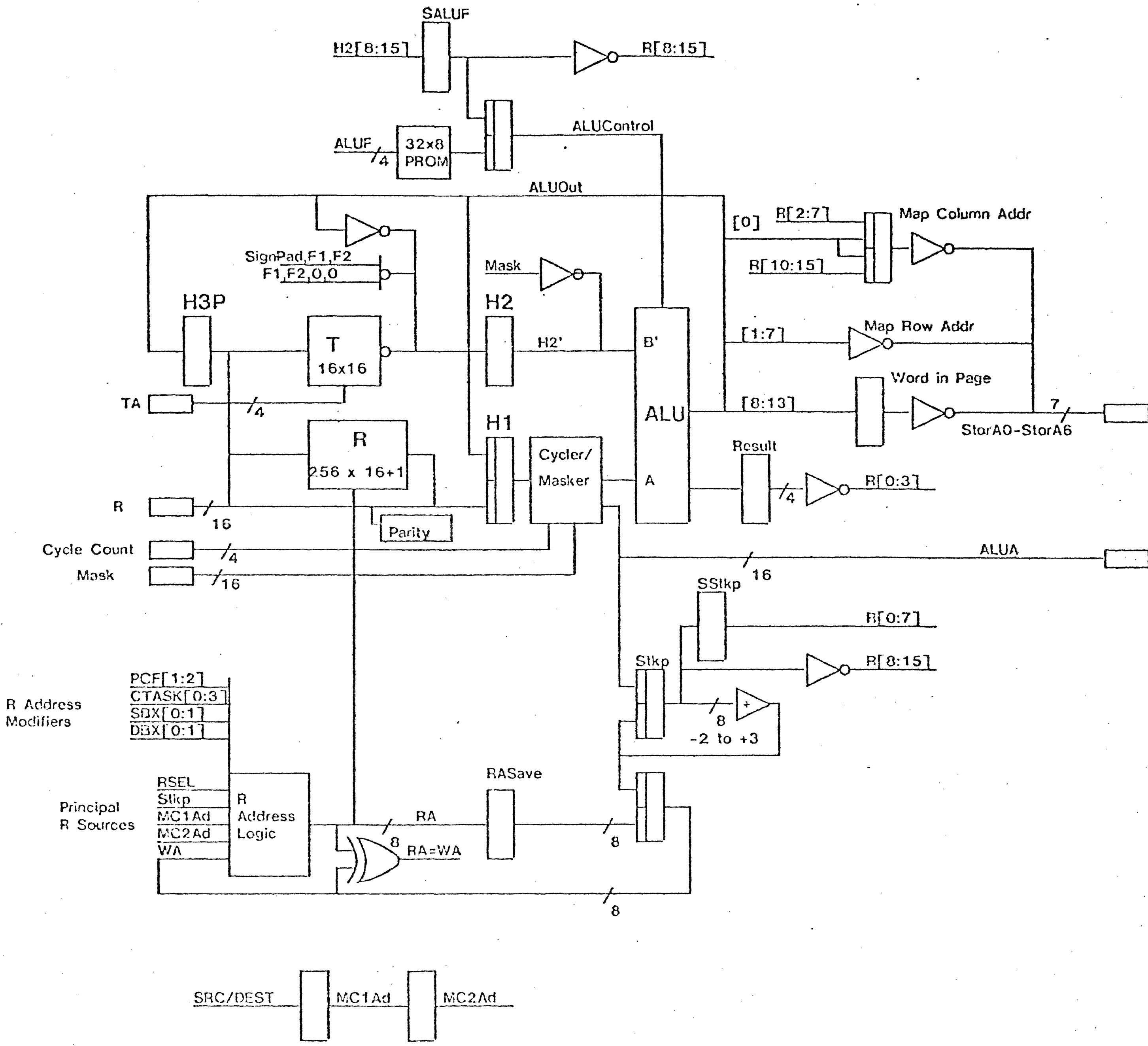
Field BS already set - Bsel is 0 or 1 for a constant, and 3 for the cycler/masker. Thus, RTEMP ← RSH[RTEMP, 1] AND (2C) would produce this message. This statement also produces "F1.used.twice".

MicroD is the part of the assembler which places the instructions in their final locations. Any messages received from MicroD are because of placement constraints. The following are the most common:

Attempted to link LabelX with LabelY - you probably have two DBLGOTOs which require LabelX or LabelY to be on an even location for one and an odd location for the other: e.g. DBLTOGO[LabelX, LabelY, alu#0];
DBLGOTO[LabelX, LabelZ, alu>=0];

Impossible allocation constraints - Most likely there are two CALLs in sequence.

A printout of all locations on two pages - This probably results from doing a CALL to a different page not being preceeded by a LOADPAGE.


SALUF
H2[8:15]
R[8:15]
ALUF
4
32x8 PROM
ALUControl
ALUOut
[0]
R[2:7]
Map Column Addr
R[10:15]
SignPad,F1,F2
F1,F2,0,0
Mask
H3P
T
16x16
H2
H2'
B'
[1:7]
Map Row Addr
Word in Page
TA
4
ALU
[8:13]
7
StorA0-StorA6
R
256 x 16+1
H1
Cycler/
Masker
A
Result
4
R[0:3]
R
16
Parity
Cycle Count
4
Mask
16
ALUA
16
SStkp
R[0:7]
Stkp
R[8:15]
8
-2 to +3
PCF[1:2]
CTASK[0:3]
SBX[0:1]
DBX[0:1]
R Address Modifiers
RSEL
Stkp
MC1Ad
MC2Ad
WA
Principal R Sources
R Address Logic
RA
8
RA=WA
RASave
8
8
SRC/DEST
MC1Ad
MC2Ad