XEROX

System Development Division
October 28, 1977

To: Simulator users

From: Crowther

Subject: D0 Simulator operating instructions

Documentation

The user of the simulator must be familiar with three other systems which are documented elsewhere:

1) The D0 Assembler System, documented on [Maxc]<Fiala>D0ASSEM.PUB.
2) The mesa system, documented on [maxc]<mesa-doc>.
1) The D0 itself, soon to be documented on [maxc]<thacker>???

This memo will assume that the reader knows how a D0 works, presumably from reading the documentation on the processor. Therefore I will casually use terms like the T register, and assume the reader understands their function. If the reader is unfamiliar with mesa, he is advised to get help from an expert in preparing his starting disk; thereafter mesa can be ignored.

Getting Started

To run the simulator you need a disk with the following modules on it:

Mesa.run (renamed runmesa.run if using Johnsson's exec)
Mesa.Image
wmanager.bcd
s.bcd

s.bcd is on [maxc]<crowther>s.bcd. Johnsson's exec is on [maxc]<johnsson>exec.run. All the other files can be found on the mesa directory on maxc. It is also helpful but not necessary to have an installed mesa debugger on the disk. With a debugger, if the system crashes the user gets some clue about what happened.

To run the simulator you must start the file s. If using the regular exec type the following (user types the bold characters, system the normal characters):

mesa
new filename s
start filename ESC

If using Johnsson's exec (which I recommend) type:

mesa s

After a delay for loading, the alto screen comes alive and is waiting for your instructions. In order to understand what to do now, you must realize that there are really three quite separate programs running in the machine at this instant.

1) There is the standard *mesa window package*, which is documented with the mesa system: it has complete control of the mouse, and will let you move the window(s) on the screen or create and destroy new windows. You can scroll any window in the normal mesa way.

2) There is a *D0 simulator*, which has a complete simulated state for a D0 (without any I/O). The D0 design is documented elsewhere. This simulator has a few features which the real D0 lacks: in particular, there is a control register which will start the machine when something is written into it. Depending on whether a 1,2, or 3 is written the simulated D0 will run for one cycle, one instruction, or forever (until a break is encountered). The D0 simulator communicates with the rest of the system through a pair

of routines which read and write simulated D0 memories. This is the only path into the simulator.

3) There is a *ddt*, which is a complex teletype-oriented user interface. The ddt accepts one character user commands, optionally preceded by a single parameter, converts them into commands to the simulator, and displays some sort of result at their completion. Since the ddt is trying to present a nice interface, it knows something about the format of D0 memory and D0 instructions, and can print the latter in a fairly reasonable way. It is often unnecessary to make a distinction between the ddt and the simulator, and I will occasionally confuse the two in the following. But sometimes the distinction is crucial for an understanding of the whole package.

Using ddt

You are now in the ddt and able to type ddt commands. The format of almost all commands is a single optional parameter followed by a single command character. The parameter is either an octal number or an alphanumeric string, while the command is either a punctuation character or a control character (written in this memo as Ctl X). The characters "+", "-", "*", and Space are really command characters, but their only effect is to help build up a complex parameter from a simple one. Using these commands one could type myStart+5 and use it as a parameter. In the use of strings the distinction between uppercase and lowercase is ignored, so that string, String, and STRING are all the same symbol.

Note that the parameter is specified in *octal*. The whole of the ddt operates in octal mode only, both on input and on output, and there are never any decimal numbers involved. I do not boast about this, but at least it is easy to remember that there are no options.

Load and Dump

The simulator is not of much use without a microcode program to simulate. The simulator will load the output of the micro/microd microcode assembly system (a ".mb" file). In addition the simulator load command expects there to be a source file (a ".mc" file), which it places into a second window on the alto display. The relevant ddt command is xxx Ctl L, which loads files xxx.mb and xxx.mc. The assembler and its input language are described elsewhere. Usually Ctl L is the very first command given to the ddt.

Examine and Change

After the load the user may wish to examine or change some of the memory locations in the simulated machine. He may examine a location by typing its address as the parameter and "/" as the command. The address may be specified either as an absolute octal number or as a symbol (which presumably came from the mb file out of the assembler, but see below for a way to define symbols in the ddt). Since there are several memories, the user is expected to precede the octal number with a single letter to indicate the desired memory. For example, i23/ would inspect register 23 of the instruction store. The defined memories are i (instruction), r (register), m (main), c (control=d0 hardware registers), and z (map - small prize to anyone coming up with a good name here). If the single letter is omitted it defaults to whatever the previous memory was. Normally the contents will print out in octal, but for the i memory that is not much use, so the i printout attempts to interpret the instruction symbolically. For the most part this is possible, but sometimes the meaning of an instruction depends on the context in which it occurs. One will frequently see a goto 45 interpreted as a []←T goto 45. Here the ddt is not smart enough to realize that the assembler specifies a T source for the alu and no store back when there is nothing else to do. (Of course the very next instruction might test the alu, so one cannot know for sure that this is a null operation).

There are other ways to inspect memory. One can type linefeed(LF) to inspect the next location, and "↑" to inspect the previous. One can type TAB to inspect the location specified as the destination in the previous instruction printout (but watch out for Call and Return, which may not do what you expect - the ddt is only looking at memory, not executing it). TAB is the most useful way of examining instruction memory.

One can change memory. This is particularly useful for setting up test cases during a debugging session. The method is to examine the desired memory location by any of the methods described above, and then to type a new contents followed by a carriage return (CR). For example, r15/123 456CR will change r memory location 15 from 123 to 456. LF, ↑, and TAB will work just as CR, and in addition will go on to inspect a new location. Unfortunately, the only form of input is octal numbers and predefined symbols: one cannot type in new instructions in anything like a natural format. It is possible to change i memory by typing in a 16 digit octal number (yes, the input is triple precision), but in practice that is too painful to attempt. I tried to build an instruction read in routine, but it was just too hard. Maybe someday.

Simulator Execution

Eventually one tires of looking at the program and decides to run the simulator on it. The easiest way to do this is to type the start address followed by Ctl G. If you are just learning the simulator I do not recommend this way, but if you use it, the simulator will execute instructions as the program directs until one of three things happens:

1) a breakpoint is encountered. You may set breakpoints in the file loaded from the assembler, or you may set them by typing address Ctl B. The simulator will stop with the breakpoint instruction about to be executed.
2) one of many illegal instructions is encountered.
3) you type a backspace(bs).

Another way to run the simulator is to type address Ctl S. This will prime the simulator to start at the specified address, but will actually execute nothing. Another Ctl S without a parameter will step the simulator forward one instruction. In this way one can step the program forward one instruction at a time. I recommend this mode when first learning the simulator.

When the simulator stops execution and returns to the ddt, all of the active registers of the simulated D0 are accessible, as well as all of the memories described before. One can type t/ or apc/ and see what is currently in these registers. One can even change these registers in mid stride. At each return to ddt two especially useful registers are automatically printed. These are the register holding the address of the next instruction to be executed, and the MIR which holds the instruction itself.

I want to tell you at what part of its cycle the simulated d0 stops, for that is vitally important for understanding what the various registers mean. In order to do that I must explain a little bit about how the simulator treats time. The answer will sort of turn out to be that the D0 has stopped just after the start of cycle zero of the machine, so that all of the registers which are loaded at time zero have actually been loaded, but none of the gates which hang off of those registers have yet started to change.

At the beginning of every cycle the simulator starts with a record which contains the complete state of the D0. This record contains things like h1 and cia and apc. It first executes a set of procedures whose job is to compute various gating functions from that record. For example, the actual r address specified and the output of the cycle/masker. It next executes another set of procedures whose job is to compute a new record which will be the state of the machine at the start of the next cycle. This new record is kept completely separate from the old one until the very end of the simulated cycle. Finally, the new record is copied into the old one (with due care for the abort case), and the cycle repeats. When the machine stops, the copy over has *not* happened, but the ddt is

looking at the *new* registers. Normally the simulator stops at the end of cycle 3, but because the ddt is looking at the new register it seems that it is the beginning of cycle 0. Actually, since an abort can prevent the normal loading of some of the registers, one must take care when interpreting the ddt output. The ddt is willing to display not only the contents of the new state record, but also any of the gating functions which seem to be of interest. There are approximately 60 values which can be examined in this way.

One of the entries in the state record is a 2 bit counter (called *cycle*) which cycles through the four stages of the instruction being executed. Cycle is used to set one of four corresponding booleans called time0, time1, time2, and time3. The booleans in turn are used to decide whether a particular part of the simulator logic should execute. Thus after four passes through the main loop of the simulator one instruction will be completely executed. To mimic the D0 overlap, the simulator sets another of the time booleans on each pass. This will force the execution of logic corresponding to the appropiate cycle of the overlapped instruction, and because of the nature of the D0 design the two cycles will not conflict. But the simulator will work equally well if the overlap is not called for, which means that it is easy to run the D0 simulator in a non-overlapped mode. The value of such a mode is in the debugging of microcode: it is much easier to understand what is going on if you have all of the variables relevant to the current instruction at hand, instead of seeing half of them as they have been stepped on by the next instruction. There is a control register (c1 = "overlap") which can be set to zero for overlapped mode and one for nonoverlapped. The default is *nonoverlapped.*

I recommended the single step mode for the initial experience with the simulator because I found the various registers did not always have the values I expected, even when the simulator was working correctly. With the single step mode one is at least confident where the program has stopped and by what path it got there. One final caution: the main memory is of course asynchronous, and the result does not always show up until several instructions have been executed. If you stop just after computing your final answer, you may never get to see it!

Command Strings

The user has the ability to enter a string (called a *command string*) for the ddt to remember. He can later specify that the ddt execute the whole string as though it had been typed from the keyboard. A typical string might single step the simulator and print out several registers for the user to examine. The ddt has storage for four such strings, labeled 0,1,2,and 3. The systax for command string entry is # Ctl Z <command string> Ctl Z, where # is the string label. The syntax for executing the command string is # ESC. An ESC with no label repeats the last command string.

The rest of this memo lists and describes each of the ddt commands, including all those mentioned above plus a few other less used ones. Following the ddt commands is a list of the simulator memories, with special emphasis on the 64 simulator control registers.

DDT Commands

editing
DEL, BS ,Ctl A all abort the current command
CR with no parameter does nothing but move the display

inspect and change:(change only if explicit parameter)
/ inspect
CR change
LF change and inspect next
↑ change and inspect previous
TAB change and inspect jump address

> same as CR (for wasting less display space)

building complex parameters
+ plus
- minus
* times
Sp plus

load/dump
Ctl L load
Ctl D dump (hasn't worked since 36 bit D0 change)

D0 control
Ctl G run D0
Ctl S step D0
Ctl B set breakpoint at parameter
Ctl C clear breakpoint at parameter
Ctl W set task level to parameter (W stands for Wakeup Task)
BS halt a running simulator

ddt control
Ctl Q exit ddt(quit) (shift swat is faster)
? type list of commands
= retype the last thing in octal
Ctl Z enter a command string terminated by another Ctl Z
ESC play the command string through the ddt
Ctl T type out the command string
Ctl R type out all the R memory symbols with their values.
xx: yyCR define symbol xx to have value yy.

The simulator memories

i memory: 4K of 48 bit words
1) only 36 bits are used
2) the parity bit holds breakpoint information. IF YOU USE THE PARITY BIT FOR DATA, BEWARE - CLEAR ALL BREAKS WILL CLEAR IT.
3) the simulator keeps i memory on the disk, and caches two 256 word pages in core.

m memory: 2K of 16 bit words

r memory: 256 16 bit words

z memory: nominally 16K of 13 bit words pointing from virtual to real addresses. Actually 8 words pointing from real to virtual addresses. Searching the 8 words slows the simulator down a little, but not as much as keeping the z memory on disk.

t memory: 16 16 bit words
only t(ctask) can be read and written by the ddt

c memory: 64 48 bit words
1) most of these addresses have only 16 bits of memory behind them, but a couple have more
2) most of these addresses are implemented by a table of 64 pointers to various structures in the simulator data region. In particular, there are a lot of pointers into the output version of the state vector, and a lot into the computed gating functions.

The C Memory in detail

The following list gives the c address, followed by the ddt symbol for that address, followed by a brief description of the register. There is no longer any method behind the ordering of these registers.

c00 none: D0 control. read yields zero, write 1-4 does command
1=> run (Ctl G)
2=> step (Ctl S)
3=> single cycle
4=> clear all breaks
c01 overlap: zero like D0, 1 non overlap mode.
c02 pc: establish a new i store pc - used in Ctl G and S
c03 break: set break
c04 clearBreak:clear break
c05 none: write a 1 to clear whole machine (all memories)
c06 stkp: see D0 manual.
c07 pcf: see D0 manual.
c10 cycleCtl: see D0 manual.
c11 sstack: see D0 manual.
c12 sb: see D0 manual.
c13 t: see D0 manual. Access is to T(Ctask).
c14 h1: see D0 manual.
c15 h2: see D0 manual.
c16 none: unused.
c17 alua: see D0 manual. Alua represents gates, not a register.
c20 alu: see D0 manual.
c21 none: unused
c22 mir: see D0 manual.
c23 rselGates: the r address, as computed from MIR
c24 jumpGates: the jump address, as computed from MIR
c25 cycle: an instruction cycle counter running from 0 to 3
c26 apc: see D0 manual.
c27 flags: A set of bits indicating control conditions
001=>abort
040=>steal
100=>r write back
200=>t write back
400=>time3
1000=>dispatch
2000=>freeze
c30 ctask: see D0 manual.
c31 none: unused
c32 db: see D0 manual.
c33 pcx: see D0 manual.
c34 none: unused
c35 none: unused
c36 none: address - an intermediate PC useful for debugging
c37 saluf: see D0 manual.
c40 minnbr: see D0 manual.
c41 page: see D0 manual.
c42 cia,next: see D0 manual.
c43 tpc: see D0 manual. tpc(task0) is shown.
c44 conds: a set of booleans related to skip conditions
001=>attention
002=>r neg
004=>r odd
010=>a carry

020=>a neg
040=>a zero
100=>overflow
c45 apcTask: see D0 manual.
c46 none: mc2 memory page
c47 none: mc2 memory address
c50 none: mc2 function
c51 none: mc2 r address
c52 none: mc2 going 0=not going, non 0 counts cycles
c53 none: mc1 memory page
c54 none: mc1 memory address
c55 none: mc1 function
c56 none: mc1 r address
c57 none: mc1 going 0=not going, non 0 counts cycles
c60 none: unused
c61 clock: counts 70 ns cycles
c62 none: unused
c63 csData: see D0 manual.
c64 csin: see D0 manual.
c65 csinExtention: see D0 manual.
c66 none: unused
c67 none: unused
c70 none: unused
c71 nextm7: address executed 7 inst ago
c72 nextm6: address executed 6 inst ago
c73 nextm5: address executed 5 inst ago
c74 nextm4: address executed 4 inst ago
c75 nextm3: address executed 3 inst ago
c76 nextm2: address executed 2 inst ago
c77 nextm1: address executed last