D0 GOTCHAs

8 February 1982

Filed on: [Indigo] <D0Docs>D0Gotchas.Press, .Bravo

There are a number of errors that Micro and MicroD do not check for. The following is a list of known illegal instruction combinations as well as instruction combinations that do not work like you might think they work. If you know of any others, please send a message to Ed <Fiala>. This list is incomplete.

Continuing From Faults

After a fault, the fault handler sometimes resumes the aborted instruction, but cannot do so in certain cases. Faulting in such situations must be avoided.

Two known problems are an abort following a LoadPage, or an abort of an instruction using the bypass kludge. Gotchas related to these are discussed below. In preparation for these, the situations when faults happen and when the fault handler continues from them are discussed here. Fortunately, the fault handler CAN continue correctly from the instruction following a Dispatch, UseCTask, or APCTask&APC_.

Stack Overflow and Underflow

No existing microcode system resumes from stack overflow or underflow. Pilot restarts the emulator at a trap address; other systems crash. Consequently, no lore for continuing from these has been developed.

MC2 Faults

An MC2 fault happens on uncorrectable storage failures (multiple errors) for any of the references that can cause such a failure; these are PFetch1/2/4, PStore1/2, and IOFetch4/16. This is considered to be a crash condition, so we don’t have to worry about resuming following such a fault.

Correctable failures (single errors) only happen when the LogSE bit is true in the map entry for the page and only when the reference experiencing the single error is a PFetch1/2/4. The fault handler would ALWAYS resume the program after single-error logging, if we allowed this feature to be used, but at the moment LogSE is only used in a limited context by the Initial microcode during storage testing.

Unfortunately, Pilot has about 10 or 15 LoadPages which might have LogSE faults abort the next mi, and several of these are not readily fixable. Continuation after the LoadPage at the buffer refill location (location 0) is particularly difficult to arrange. As a result, LogSE is ILLEGAL in Pilot and AMesa, so programmer’s need not worry about it at present. The fault handler will crash on any MC2 fault, assuming an uncorrectable error.

MC2 faults happen on about (not sure) the 18th cycle after a PFetch1/2/4, if the data is not touched sooner. If read through the ALU earlier than the 18th (?) cycle, then the fault is supposed to abort the instruction which touches the data. However, we have discovered that this next instruction completes before the fault on PFetch4 MC2 faults. For example in:

```
PFetch4[Base,RReg0];
T _ RReg0;      *This instruction SHOULD be aborted
Call[Foo];
```

When the PFetch4 experiences an MC2 fault, T _ RReg0 completes before the fault. In addition, TPC is written by the Call[Foo], even though that instruction is aborted by the fault, and this can be confusing during debugging. The extra instruction is executed because 8 cycles of transport for the PFetch4 is longer than the 5 cycle abort of the memory controller, so the next instruction completes before the fault begins. We don't know any reason why this bug is harmful because the corrected data is read.

MC1 faults

MC1 faults represent page or write protect faults. In isolation (i.e., in the absence of suspended cycles due to transport for previous references), MC1 faults seem to abort the fourth instruction (seventh cycle?) after a PFetch1/2/4 or PStore1/2 reference and the third after a PStore4.

However, transport for a preceding reference might suspend intervening cycles and advance the fault. A preceding PFetch2 might advance the fault to the third instruction or a PFetch4 to the second, ordinarily. If transport for a preceding PFetch4 intervened and experienced error correction, the fault would abort the instruction immediately after the reference!

It is true for Pilot, and probably for other microcode systems, that page and write protect faults are only supposed to happen on references INITIATED by the emulator, although, if the emulator tasks, an io task might be INTERRUPTED by the fault.

The fault handler will ALWAYS resume from such a fault if an io task was interrupted, (possibly changing the emulator's TPC or handling a nasty special case for a Return in the instruction being resumed). For Pilot, it will RARELY continue from such a fault if the emulator was interrupted--it only does this during instruction buffer refill for a NextInst and on jump opcodes. Continuation in these cases is not accomplished through the hardware's Restore function, so other restrictions commented in the code must be dealt with.

1. LoadPage in Emulator Task

Consequently, the only LoadPage restriction for the emulator is with respect to page and write protect faults, and it only applies in the rare situations when the emulator will be resumed. In this case, if the preceding reference is unknown, then LoadPage is illegal in any of the three instructions following a reference which might fault. If something is known about the preceding reference (or lack thereof), then it might be possible to relax this restriction and allow LoadPage to be used in the first or second instruction after a reference.

If you screw up, you will continue execution on the wrong page with no warning from the fault handler.

2. Task Switch Before LoadPage in Non-Emulator Task

A corollary to (1) above is that in an io task, a task switch must not have occurred during the three instructions preceding a LoadPage. This is because the Emulator can do a memory reference and a return on the same instruction. The non-Emulator task that runs must then wait at least three instructions for the page or write protect fault to occur.

The following combination is illegal:

```
                Call[TaskSwitch];
                LoadPage[0];
```

The fault handler will give a distinctive MP code if you violate this Gotcha.

3. Bypass Kludge When a Faulting Reference is Preceded by a PFetch4

Ordinarily, a PFetch4 would suspend only four cycles, so the page fault for a PFetch1/2/4 or PStore1/2 would be advanced no earlier than the second instruction after the reference, but if error correction occurs, a total of 8 cycles would be suspended for transport, and the fault would be advanced to the instruction immediately after the reference.

This means that the bypass kludge cannot be used following a reference which faults and might have PFetch4 transport occur between itself and the next instruction, if the fault handler will continue from the fault. Since continuation from MC2 faults and stack errors never happens, this is the only case to worry about.

4. Beware of H4 parity errors.

H4PE's are possible following Input, IOStore4, or IOStore16 references. They would ordinarily be fatal, crashing the system, but a 3 mb Ethernet controller routinely generates them. Also, the MC1 memory controller microcode has a bug that causes a wild branch after an H4PE, if another memory reference is starting at that time.

To cope with these hardware problems, the fault handler ALWAYS continues from H4PE's, under the assumption that they were generated by a 3 mb Ethernet input task, and the Ethernet microcode has an appropriate number of Nops following each Input or IOStore4 to ensure that MC1 doesn't take a wild branch and that an H4PE abort will not fall after a LoadPage.

The effect of this treatment is that any real H4PE goes undetected and may additionally trigger either a LoadPage gotcha or MC1 wild branch as a result of an H4PE. Problems are minimized by jumpering out H4PE's on machines with 3 mb Ethernet controllers.

Consequently, except in the 3 mb Ethernet microcode, H4PE's are assumed never to happen, so microcoders need not worry about any coding restrictions.

For the Ethernet task, an H4PE aborts the fourth instruction following an Input or the sixth following an IOStore4. If an Input is interlocked, the instruction after the interlock is aborted by the H4PE fault. The instruction preceding the abort cannot contain a LoadPage. A violation causes the ’LoadPage error’ number in the MP. (See Gacha #2.)

Another reference must not start until any reference that might generate an H4PE has finished. (All instructions that that can cause an H4PE use only MC1.) *This one is really nasty.* It can cause almost anything. A common result is an RM parity error.

The problem is a bit complicated to explain. When it is finished processing it’s part of an operation, the MC1 microcode normally jumps to 0 (an idle loop that waits for the next memory operation). If the H4PE bit is on, it jumps to 2 to report the error to the fault handler. It does this by ORing the 2 bit into the next address. The problem is that the dispatch to the first instruction of the next opcode may be happening at the same time. If so, it goes to the desired location OR 2, and you have to get out the listings to find out where that is.

An even more complicated symptom will happen if there is not much IO activity. If the wild branch happens, the H4PE flop is left set until another Input or IOStore happens to test it. This prevents data from getting written into memory, so the emulator can get really confused.

To avoid the MC1 wild branch requires 3 non-memory instructions after Input and 4 after IOStore4.

5. IOStrobe with RETURN

IOStrobe on the same instruction as RETURN does not work. The purpose of IOStrobe is to turn off the wakeup latch and go back to sleep until the next wakeup occurs (e.g., in the SA4000 code). If IOStrobe is on the same instruction as the RETURN, the latch will still be on when the RETURN is executed, and control may remain in the same task.

6. Minimum Instructions Between Timer Operations

When a timer instruction is executed (Load Timer or Add To Timer), it is necessary to execute at least 14 machine cycles (usually seven instructions) before another timer instruction can be executed.

7. Loading the Source of PStore4

There must be at least one non-memory instruction between the loading of the source of the PStore4 and the PStore4; otherwise the processor may not be able to deposit the data in R before the memory controller needs it.

The following combination is illegal:

```
R0 _ ... ;
PStore4[Base,R0];
```

8. Loading the Source of PStore1 or PStore2

There must be at least one non-memory instruction between the loading of the source of a PStore1 or PStore2 if the PStore might be followed immediately by another memory reference.

The following combination is illegal:

```
Temp _ Temp-1;
PStore1[Base,Temp];
Input[...];
```

In this example, the write of Temp will not be done until the Input is finished. Since the Input won't start until the PStore is finished, the PStore will store the OLD value of Temp.

9. Task Switch after PStore1 or PStore2

As corollary of (6) above, a PStore1 or PStore2 may not be followed immediately by a task switch if there was not a non-memory instruction between the loading of the source and the PStore.

The following combination is illegal:

```
R0 _ ... ;
PStore1[Base,R0], RETURN;
```

10. Loading the Odd Base Register

At least one non-memory instruction must be executed between loading the odd base register and a memory reference that uses the base register.

The following combination is illegal:

```
BasePairOdd _ ... ;
PStore1[BasePairEven,R0];
```

11. Loading the Even Base Register

At least one non-memory instruction must be executed between loading the even base register and a memory reference that uses the base register. Exception: if it can be guaranteed that the memory reference will not be aborted, or the memory reference uses DF2 addressing with a displacement of zero.

The following combination is illegal:

```
BasePairEven _ ... ;
PStore1[BasePairEven,R0];
```

12. Loading the Displacement Before a Memory Reference

Loading the displacement immediately before a memory reference works because of the bypass hardware:

```
T _ Displacement;
PStore1[Base,Source];
```

In the above example, T does not actually get written until after the memory instruction. Because there is a write of T pending when the PStore is executed, bypassing is invoked, and the output of the ALU is correctly used in the calculation of the memory address.

Loading the displacement before a memory instruction will not work if there are only other memory instructions before the displacement is needed.

The following combination does not work:

```
T _ Displacement;
Input[...];
PStore1[Base,Source];
```

In the above example, T does not actually get written until after all memory instructions have been executed. Because there is a write of T pending when the PStore is executed, bypassing is invoked, and the displacement used in the calculation of the PStore address is the output of the ALU left over from the Input (which is junk). The store is done, but not to the place you expected!

13. Reading a Register Following a Memory Reference

If an RM register or T is loaded in the instruction preceding a memory reference, it cannot be read in the instruction immediately following the reference.

The following combination does not work:

```
DiskAddress _ T;
PFetch1[...];
T _ LSH[DiskAddress,4];
```

In this example, the bypassed value written to T is not the value written, but the result of the base register addition done by the memory reference.

In some cases this can be thought of as a hardware feature rather than a bug. In the Memory section of the D0 Hardware Manual, uses of this feature (known as the "bypass kludge") are described in detail.

14. Testing IOAtten After a Task Switch

A microprogram must not test IOAtten in any instruction following a task switch. This is because the time required for CTask[0..3] to reach a controller and be returned to the processor as IOAtten is longer than one cycle.

The following combination is illegal:

```
                Call[TaskSwitch];
                GoTo[Attention,IOAtten];
```

Also note that the UTVFC (display controller) has special restrictions regarding the memory reference which must be issued before testing IOAtten. Study the comments in the display modules as well as the hardware manual to find out how these work.

15. Issuing IOStrobe After a Task Switch

? IOStrobe should not be issued until the third (second?) instruction following a task switch (?Not sure about this one?).

16. Storing into the IO Register of Another Task

It is not possible to store into the IO register of a lower priority task. In particular, the form:

```
         T _ xx;
         Output[reg];
```

works fine from task 0. From any other task, the current CTask gets ORed into the high four bits even if you don’t want them there.

Output Followed by PStore4

An Output instruction followed by a PStore4 instruction will cause all types of memory errors, register parity errors, and generally unpredictable behavior. The problem is that this instruction sequence will cause both MC1 and MC2 to access RM at the same time. Have a look at the memory timing chart on page 63 of the D0 Hardware Manual. Suppose the Output instruction is started. MC1 runs for two cycles. Two cycles later, MC2 starts, and the next instruction also begins execution. Suppose the next instruction is PStore4. MC1 will start right away because the Output has finished with MC1. MC1 will look and see if MC2 will reference RM in the next cycle. MC2 will reference RM five cycles later, but MC1 does not look that far ahead, so it starts. If you line up RM references for MC1 and MC2, you will see that they occur simultaneously if the PStore4 is executed right after the Output or one instruction later.

When an Output instruction is executed, a certain number of instructions must be executed before a PStore4 or Task switch can occur. The number of instructions following the Output depends on what the memory is doing. There are three cases to consider, and these are described separately below.

17. Output Preceded by any Memory Instruction Other Than IOFetch or Output

If in a segment of code between task Returns, the memory instruction immediately preceding Output is any memory operation other than IOFetch4, IOFetch16, or Output, then at least two non-memory instructions must lie between the Output and a PStore4. Similarly, a task Return can be performed on or after the second instruction following the Output.

18. Output Preceded by IOFetch or Output

If in a segment of code between task Returns, the memory instruction immediately preceding Output is IOFetch4, IOFetch16, or Output, more than two instructions may be required following the Output before a PStore4 or task Return can be executed. This is because the promotion of the Output instruction from MC1 to MC2 in the memory system is deferrred for a number of clocks due to the preceding memory operation. This causes the RM access in the Output to occur later in time than if the memory pipe had not been busy. The actual number of instructions required after the Output is given by the number of non-memory instructions lying between the Output and the preceding memory instruction.

Preceding Memory Instruction	**Intervening Non-memory Instructions**	**Non-memory Instructions required following Output**
IOFetch4 or Output	2 or less	5
	3	4
	4	3
	5 or more	2
IOFetch16	8 or less	5
	9	4
	10	3
	11 or more	2

Regardless of the proximity of an interfering prior memory operation, it is always legal to interlock the Output to avoid unacceptable instruction count requirements prior to the Output (it won’t take any less total machine time; it just causes a processor suspension until the Output is complete). Specifically, the sequences:

```
Output[foo];
foo_foo;
PStore4[...];

Output[foo];
foo_foo, RETURN;
```

are legal.

19. Output Alone

If in a segment of code between task Returns, Output is the only reference,then it must assume that the last instruction in the previous task could have been IOFetch16. Therefore, it must obey the instruction count rule given in number 18 releative to IOFetch16, or do the interlock following the Output.

20. At Least One Non-Memory Instruction in a Task

A task must execute at least one non-memory instruction. This follows from (11) above. Consider the following sequence of instructions in tasks A and B:

```
TASK A:

      T _ xx, RETURN;

TASK B:

      Memory Instruction;
      Memory Instruction;
      Memory Instruction, RETURN;

TASK A Continues:

      R _ T;
```

When Task A continues, T will not have been assigned because of the bypass problem.

21. Minimum Time Between Maintenance Panel Operations

There must be at least 800 nanoseconds between a ClearMPanel and an IncMPanel operation, and at least 400 nanoseconds between an IncMPanel and another IncMPanel operation. Since clock period may be as little as 70 nanoseconds, 12 cycles (6 instructions) must be allowed between ClearMPanel and IncMPanel and 6 cycles (3 instructions) between two IncMPanel's.

22. Return Following Dispatch

Beware of a Return statement following a Dispatch. This is equivalent to a notify (i.e., load APCTask&APC and then Return). The following example illustrates the point:

```
MsgStatus _ (MsgStatus) + 1;
Dispatch[MsgStatus,7,4], Skip[ALU#0];
      MsgStatus _ (Zero) - 1, RETURN;
Disp[MsgState];
```

In the above example, the Return statement causes a jump to Control Store location zero. The following example will produce the desired results:

```
MsgStatus _ (MsgStatus) + 1;
Dispatch[MsgStatus,7,4], Skip[ALU#0];
      MsgStatus _ (Zero) - 1, GoTo[RETURN];     *Delay one
      instruction.
Disp[MsgState];
```

23. Timer Instruction With Conditional Branch

It is illegal to have a conditional branch in the same instruction as a LoadTimer or AddToTimer instruction. This may cause the timer instruction to fail. For example:

```
Skip[ALU#0], AddToTimer[RTimer];
```

24. Loading PCF Before NextInst/NextData

PCF cannot be loaded in the instruction immediately preceding a NextInst or NextData instruction. PCF gets loaded, but the decision to cause a trap is made at the time MIR is loaded with the NextInst or NextOp instruction. This means that if the 10-bit was on before PCF was loaded, a trap will occur.

25. T _ Stack

In the Emulator, the first reference to the Stack in a byte code must not be:

```
T _ Stack;
```

If a PFetch2 is pending to the Stack, the processor may read the R location before the second word has been transferred to the stack. To prevent this from occurring, the first reference to the Stack must also decrement the stack pointer:

```
T _ Stack&-1;
```

Decrementing the stack pointer causes an instruction to be aborted if a PFetch2 is pending to the stack.

26. Putting a Negative Number on the Stack

The instructions:

```
Stack&+1 _ (Zero) - T;
Stack&+1 _ (Zero) - 1;
```

do not work when the stack is empty because the Zero causes the hardware to read the stack, possibly causing a stack underflow trap, even though the value read from the stack is unused.

To put -1 on the stack, the instruction:

```
Stack&+1 _ (Stack&+1) OR NOT (0C);
```

does not work if the stack is empty either.

27. Beware of Fetches to the Stack

The instructions:

```
PFetch4[Base,Stack0];
```

Will fetch four words to the stack registers. If this is followed by a stack instruction:

```
Stack&+1 _ ...
```

there will be no interlock between the PFetch4 and the write to the stack. This means the write to the stack will be overwritten by the fetch. The way to avoid this is by means of an explicit interlock:

```
Stack0 _ Stack0;
Stack&+1 _ 0C;
```

28. Writing a Register From a Constant or T Does Not Interlock a PFetch

Assigning a constant or T to a register does not interlock a PFetch1/2/4 to that register. If you assign a constant to a register and a fetch is pending on that register, the constant will be overwritten by the fetch. For example:

```
PFetch4[Base,R0];
R2 _ 0C;
```

In the above example, R2 will be overwritten by the fetch. Reading a register that is being fetched will cause it to interlock:

```
PFetch4[Base,R0];
R2 _ R2;
R2 _ 0C;
```

29. Testing the R<0 or R Odd Branch Condition Does Not Interlock a PFetch

You will defeat both the hardware interlock and the assembler’s no-interlock warning, if you write an R<0 or R Odd branch condition as in the following example:

```
PFetch4[Base,R0];
R2, T _ 0C, GoTo[Next,R<0];
```

or in the following:

```
R2, GoTo[Next,R<0];
```

In this case, you will encounter Gotcha 28. However, if you make a practice of writing your R<0 and R Odd branch tests as follows:

```
A _ R2, T _ 0C, GoTo[Next,R<0];
```

then the assembler will warn you with a 'no register interlock' message. And if you write your RM branch conditions as:

```
LU _ R2, GoTo[Next,R<0];
```

the hardware interlock will safely occur.

30. Base Register Now Interlocks.

The D0 Hardware Manual is wrong in section 5.3 R Interlocking - where it says:

"Note that the interlock comparators are not activated during memory reference instructions. This means that if a base register is fetched into RM and then used in a memory reference instruction without being explicitly read first, the old version of the base register will be used if the base register has not been filled by the memory. This situation must be avoided by the programmer."

A hardware modification in 1981 was made so that the base register used by a reference interlocks in exactly the same way as reading that register in a non-reference instruction. This improvement is very valuable for the Alto emulator and in some other situations.

As a result of this hardware change, however, you can now get unwanted interlocks for Input, Output, and ReadPipe references where the base register isn't used.

In Output[RReg,F2], for example, the assembler supplies register 0 in the appropriate group of 20b registers as the default base register. If this Output were preceded by a PFetch into that register, then the Output would unintentionally be held until the PFetch finished. To avoid this, supply a third argument to Input, Output, or ReadPipe, which will be used as the base register for the instruction. For example:

```
Output[RReg,0,RConst];
```

uses RConst instead of register 0 as the base register.