The Case Of The Missing PLA Term, or, Microcode Bugs I Have Known

Bob Supnik, 24-Sep-2004

Introduction

Perusing a recently scanned copy of a later PDP-11 system manual (the PDP-11/84), I was surprised to see two entries in the “PDP-11 differences” list that I had never seen before:

55. The ASH instruction with a source operand of octal 37 (shift left 31 decimal times) will cause the register to be shifted right instead of left.
56. The ASHC instruction with an octal value of 37 (shift left 31 decimal times) in source operand bits 5:0 and bits 15:6 of the operand being non-zero, will cause the register to be shifted right instead of left.

Both new entries had a single check mark in the column for the J-11. These weren’t “differences”, these were bugs in the microcode that had been discovered too late to be fixed before the J-11 was in general release.

Microcode bugs were an inherent risk in a ROM-based VLSI microprocessor. Large-scale microprogrammed machines used PROM chips, which could be replaced, or RAM chips, which could be reloaded, for control store. But microprocessor control stores, once fabricated, were fixed for all time.

This paper documents the microcode bugs in DEC microprocessors that got out into general release.

Bad-Mannered Testing: F-11 MULP/DIVP

The Commercial Instruction Set (CIS) was a late addition to the PDP-11 architecture. Modeled after the commercial instructions in the VAX, CIS was intended to boost the PDP-11’s COBOL performance. It provided an extensive set of string and decimal instructions; indeed, its capabilities were more complete than its VAX counterparts.

The F-11 (11/23) microprocessor was the first PDP-11 to implement CIS. CIS was a relatively late addition to the project (always a danger sign). The F-11 didn’t offer much hardware support for CIS: a decimal adjust microinstruction was about the extent of its capabilities. As a result, the microprograms for CIS were large. CIS required six extra microcode control store chips (six times the size of the base instruction set). MULP and DIVP required two chips just by themselves – as much as the entire floating point instruction set.

Almost all of the decimal instructions were structured as loops that counted down by bytes or nibbles. These loops were always interruptible; if an interrupt occurred, the instruction was “packed up” into the general registers, and PSW<fpd> was set. A typical loop would to round the nibble count to even, divide by two, and then count down by one, testing against zero for end of loop. However, a few loops simply rounded and counted down by two, without dividing. This worked fine – provided that the instruction wasn’t interrupted, and that the interrupt-level program didn’t tamper with the state saved in the registers.

All was well until the code was tested with a program called BADMAN. Intended to flush out latent microcode bugs, BADMAN deliberately mangled the state saved in the general registers, set PSL<fpd>, and tried to see what would happen. When BADMAN set the saved loop count of one of these count-by-two loops to an odd value, the microprogram ran forever. It was still interruptible – control was never lost – but the instruction never completed.

This problem was found after the F11 CIS option shipped, and after DEC’s interest in CIS for the PDP-11 had waned. It was never fixed.

The Case of the Missing PLA Term: J-11 ASH/ASHC

Like its predecessors the LSI-11, F-11, and T-11, the J-11 used an elegant control store structure that contained both ROM words and PLA terms. The PLA provided great flexibility and conciseness in implementing the nearly (but not quite) regular instruction set of the PDP-11. A base micro-operation operation that depended only on the PDP-11 opcode, e.g.,

```
ADD:                          ; Do ADD.
=10********0
PLA0  [^0 111 0X0, ^0 110 XXX XXX XXX XXX]
      ADD.W*      [RF, RE],   ; Override: RF to RSRC if Source Mode 0,
                              ;  RE to RDST if Destination Mode 0,
                  NAF/NOP-PF  ;  NAF to ID1 (477) if Dest. Modes 00-06
```

would be modified by other PLA terms reflecting instruction modes:

```
DOP-SM0:                      ; Turns on at DOP execution.
=10********0                  ; Register select override for Source
                              ; Mode 0.
PLA0  [^0 111 0X0, ^X XXX 000 XXX XXX XXX]
      NOP.B [RSRC, RE],       ; Override RF to RSRC (PDP11 register).
                  NAF/1777    ; Do not affect NAF.
```

All the PLA terms that were selected drove their outputs, which ‘wire ANDed’ together. Thus, any particular part of the base micro-instruction (source register, destination register, next address) could be overridden; and the overrides would apply to all base micro-instructions with appropriate addresses.

The PLA decode mechanism was so powerful that in the J11, its use was extended from decoding instructions to decoding arbitrary information. The microcode could load the PLA input register (PIR) from the data path. This technique was faster than testing bits and branching and was extensively used in computationally intensive instructions such as floating point, ASH, and ASHC.

The basic algorithm for ASH was to use PLA overrides to modify a load counter instruction based on bits<5:0>:

```
ASH1:       ; Default for EXTRX-X, ASHX-0-OVR and ASHX-NEG-1 PLAs.
=10********0
PLA0  [^1 011 110, ^X]
      LCNTR.W     [037, RE],  ; Load CNTR with shift count.
                  NAF/ASH-R   ; Overridden to NOP-PF1 for no shift or
                              ;  to ASH-L for positive shift or
                              ;  to ASH-R1 for negative 1 shift.
```

These overrides were used throughout the microcode to “extract” the value in the PIR and use the value to change microcode constants.

```
ASHX-0-OVR:                   ; SRC<5:0> = 0.  No shift.
PLA0  [^1 01X 110, ^X XXX XXX XXX 000 000]
      NOP.W [RF, RF],         ; Do not override microinstruction.
                  NAF/ASHC-NO ; Override to NOP-PF1 for ASH or
                              ;  to ASHC-NO for ASHC.

EXTR0-0:                      ; Override on 0 in PIR<5> and PIR<0>.
PLA0  [^1 011 1XX, ^X XXX XXX XXX 0XX XX0]
      LBIS.B [336, RF],       ; Override literal<5,0> (SPL, ASHX).
            NAF/1775    ; Override NAF<1> for left shift (ASH and ASHC)

EXTR0-1:                      ; Override on 0 in PIR<5> and PIR<1>.
PLA0  [^1 011 1XX, ^X XXX XXX XXX 0XX X0X]
      LBIS.B [275, RF],       ; Override literal<6,1> (SPL, ASHX).
            NAF/1775    ; Override NAF<1> for left shift (ASH and ASHC)

EXTR0-2:                      ; Override on 0 in PIR<5> and PIR<2>.
PLA0  [^1 011 1XX, ^X XXX XXX XXX 0XX 0XX]
      LBIS.B [173, RF],       ; Override literal<7,2> (SPL, ASHX).
            NAF/1775    ; Override NAF<1> for left shift (ASH and ASHC)

EXTR0-3:                      ; Override on 0 in PIR<5> and PIR<3>.
PLA0  [^1 011 11X, ^X XXX XXX XXX 0X0 XXX]
      LBIS.B [367, RF],       ; Override literal<3>.
            NAF/1775    ; Override NAF<1> for left shift (ASH and ASHC)

EXTR0-4:                      ; Override on 0 in PIR<5> and PIR<4>.
PLA0  [^1 011 11X, ^X XXX XXX XXX 00X XXX]
      LBIS.B [357, RF],       ; Override literal<4>.
            NAF/1775    ; Override NAF<1> for left shift (ASH and ASHC)

ASHX-NEG-1:                   ; SRC<5:0> = -1.  One right shift only.
PLA0  [^1 01X 110, ^X XXX XXX XXX 111 111]
      NOP.W [RF, RF],         ; Do not override microinstruction.
            NAF/ASHC-R1 ; Override to ASH-R1 or ASHC-R1.

EXTR1-0:                      ; Override on 1 in PIR<5> and PIR<0>.
PLA0  [^1 011 11X, ^X XXX XXX XXX 1XX XX1]
```

```
      LBIS.B [376, RF],       ; Override literal<0>.
            NAF/1777    ; Do not override NAF.

EXTR1-1:                      ; Override on 1 in PIR<5> and PIR<1>.
PLA0  [^1 011 11X, ^X XXX XXX XXX 1XX X1X]
      LBIS.B [375, RF],       ; Override literal<1>.
            NAF/1777    ; Do not override NAF.

EXTR1-2:                      ; Override on 1 in PIR<5> and PIR<2>.
PLA0  [^1 011 11X, ^X XXX XXX XXX 1XX 1XX]
      LBIS.B [373, RF],       ; Override literal<2>.
            NAF/1777    ; Do not override NAF.

EXTR1-3:                      ; Override on 1 in PIR<5> and PIR<3>.
PLA0  [^1 011 11X, ^X XXX XXX XXX 1X1 XXX]
      LBIS.B [367, RF],       ; Override literal<3>.
            NAF/1777    ; Do not override NAF.

EXTR1-4:                      ; Override on 1 in PIR<5> and PIR<4>.
PLA0  [^1 011 11X, ^X XXX XXX XXX 11X XXX]
      LBIS.B      [357, RF],        ; Override literal<4>.
            NAF/1777    ; Do not override NAF.
```

This sequence used the PLA mechanism elegantly and efficiently to decode both the amount and direction of the shift. For example, if the instruction being executed was ASH #10,Rn, then PLA overrides EXTR-0, EXTR-1, EXTR-2, and EXTR-4 would be selected. The load counter literal would be modified to

037 & 336 & 275 & 173 & 357 = 010

and the next address field (NAF) would be modified to left shift. If the instruction being executed was ASH #76,Rn, then PLA overrides EXTR1-1, EXTR1-2, EXTR1-3, and EXTR1-4 would be selected. The load counter literal would be modified to

037 & 375 & 373 & 367 & 357 = 1

since right shift expected a 1’s complement result. There was even a special case for ASH #77, to account for the expected 1’s complement counter value.

But what happened if <no> PLA term was selected? If the operand value is 037, then none of the left extract, right extract, or special case overrides is selected. The load counter instruction is unmodified. It loads the correct value (37) but fails to override the next address field. A right shift is executed, by default, instead of a left shift.

Another special case term was needed, to account for the ASH #37,rn:

```
ASHX-L-37:                    ; SRC<5:0> = 011111.
PLA0  [^1 01X 110, ^X XXX XXX XXX 011 111]
      NOP.W [RF, RF],         ; Do not override microinstruction.
```

```
            NAF/1775    ; Override NAF<1> for left shift (ASH and ASHC)
```

ASHC used a slightly different algorithm. It loaded the counter as well as the PIR from the data path and only used the extracts if source operands bits 15:6 were non-zero. Thus, ASHC only showed the effect of the missing PLA term if source operand bits 15:6 were non-zero.

How did these cases get overlooked? The PDP-11 never had a formal architectural exerciser like AXE for the VAX. Machines were verified by running diagnostics, hand tests, and system software. The two failing cases were fairly meaningless. ASH #37,Rn should clear Rn. With the bug, ASH #37,Rn would clear Rn for positive values, and set Rn to –1 for negative values. ASHC #xxxx37,Rn, where xxxx was non-zero, would be a meaningless looking instruction, because the shift count would appear out of range.

Whatever the reason, the bugs weren’t found until long after the J11 shipped. By then, it was too late to recall the thousands of chips in the field. The bugs became “differences”, indelible markers of the J11.

Always Check The Edge Cases: MicroVAX Passive Release

In June, 1985, the MicroVAX II system had been launched to great acclaim, when I got a call from the system group’s services leader. A MicroVAX II system at a customer site was failing unpredictably but regularly. Running diagnostics, swapping boards, and other standard service procedures had failed to find or cure the problem. Could I help?

I went to the customer site with personnel from the systems group. The first clue was that the problem always occurred in the immediate vicinity of a MOVCx instruction. The second was that the system had a third-party Qbus to Unibus converter: a configuration that DEC didn’t support and therefore had never tested. Looking at the setup with a scope, we determined that the problem occurred on a Unibus passive release (an interrupt cycle that was never completed). Why?

The interrupt flows for interruptible instructions like MOVCx and POLYx were very convoluted.

1. The interruptible instruction set a global flag to indicate interruptibility.
2. The interruptible instruction periodically tested for an interrupt; if one was pending, the microcode branched to the interrupt “fault” entry.
3. The interrupt fault handler called a cleanup routine. If the global flag was set, the fault handler called an instruction specific cleanup routine.
4. For MOVCx, the instruction specific fault handler packed up instruction state into the general registers, set PSL<fpd>, and returned to the main interrupt flows.

The main interrupt flow issued a bus cycle to read the vector. If the bus cycle was aborted, or if the returned vector was zero, the interrupt flow simply exited to instruction decode. Instruction decode fetched the instruction, saw that PSL<fpd> was set, called the instruction-specific restart routine, and the microcode was back in business. Interruptibility had been extensively tested. Why was passive release failing?

The key was in that ambiguous phrase, “fetched the instruction”. MicroVAX, like all VAXen, had an instruction prefetch mechanism. When the microcode reached its execution flows, the prefetch mechanism was already pointing at the next instruction. All paths through the interrupt and exception microcode set a new PC, resetting the prefetch mechanism, *except for passive release*. Passive release just resumed execution. Therefore, the instruction that was fetched was the *next* instruction, not the interrupted instruction. If that instruction was not interruptible, PSL<fpd> was ignored. The MOVCx never completed, the registers were in the wrong state, and the program crashed.

Even though Qbus devices did not generate passive releases, the problem was regarded as serious enough to ECO every MicroVAX system in the field. A microcode revision was hurriedly generated and tested, new parts fabricated, and systems in the field upgraded.

The moral of this story was the importance of edge-case testing. Dynamic conditions, such as interrupts, DMA, and halts, were difficult to incorporate into architectural testing. These conditions needed to be tested explicitly, whether they were “impossible” or not.

There is *another* microcode bug in MicroVAX. It has never been seen in the field; and I’ll never tell where it is ;)

If An Exception Case Is Never Tried, Is It Really There? Rigel INSV

Rigel systems had been in the field for two years when testing with a new microcode verification tool, MAX, turned up a bug. In an INSV to a register, if the position operand was a reserved operand (that is, > 31), and the equation 32-size-position caused integer overflow, the INSV instruction would be treated like a NOP instead of causing a reserved operand fault. For this to happen, position (a longword) had to be 800000xy (hex), where xy <= 32-size. For example:

```
clrl   r3                  ; source
movl   #^x800000F,r4       ; position
movl   #^x3,r5             ; size
movl   #^x7004,r6          ; base
insv   r3,r4,r5,r6         ; should fault
```

The INSV should have generated a reserved operand fault. Instead, it was treated as a NOP, and the next instruction executed normally.

The problem occurred because of an insufficiently constrained n-way branch:

```
INSV.RMODE.1.255:
     ;------------------------------------; sc<5> = 0:
     [MD.T3] <-- [MD.T3] RROT (SC),      ; [6] rotate field
                                         ; surround by position
     CASE [WBUS.NZV] AT [INSV.RMODE.1]   ; case on 32 - (pos +
                                         ; size) test from [3]

;= ALIGNLIST 01** (INSV.RMODE.1, INSV.RMODE.2)
;  WBUS.NZVC set by subtract of bytes in longword --> V = 0
```

The alignlist was insufficiently constrained; it assumed the third bit (corresponding to the overflow condition code) was a don't care, that is, the bit would always be zero. In fact, the subtract was a longword subtract, because position was a longword parameter, and integer overflow could occur. By chance, the microcode instruction selected by an errant branch was benign, resulting in an effective NOP.

Because the bug had not been seen in the field, and caused an exception on all other VAXen, the bug was waived, and Rigel was never fixed.

The End Of An Era: NVAX and Alpha

By the time NVAX was designed, late-stage microcode bugs had become a sufficient annoyance, and silicon real estate had become sufficiently great, that the design team returned to a strategy used in early VAXen: a patchable control store. This made microcode bugs a thing of the past. Alpha carried the idea a stage further, by eliminating microcode altogether. Alpha's equivalent (PAL code) was always executed from main memory, allowing errors to be corrected by simple firmware updates. Patchable control stores (for CISC machines) and loadable firmware (for RISC machines) continue to be the preferred implementations to this day.