KS10 FPGA Interrupts and the DSKAH Diagnostic

The Symptom:

The DSKAH DECSYSTEM-2020 BASIC INSTRUCTION DIAGNOSTIC (8) fails with the program counter 'stuck' at PC = 033316.

The Diagnostic Code

With the DSKAH source code in-hand, a quick analysis of the diagnostic program (shown below) reveals that the instruction at address 33316 is "`JRST .`" – which is a "Jump to self" type instruction. The diagnostic programs generally use this construct as a 'trap' to catch malfunctions.

A further analysis of the diagnostic program reveals that the diagnostic program's purpose is to test the KS10's interrupt system. To accomplish this, the program generates an interrupt and uses that interrupt to break out of the "`JRST .`" infinite loop. With this knowledge, it can be safely assumed that the interrupt is not occurring for some reason.

Two instructions are particularly important to the operation of this code:

1. The instruction at PC=33312 activates Interrupt 1 and enables the PDP10 Priority Interrupt (PI) system.
2. The instruction at PC=33313 creates software-generated NXM Interrupt on Interrupt 1.

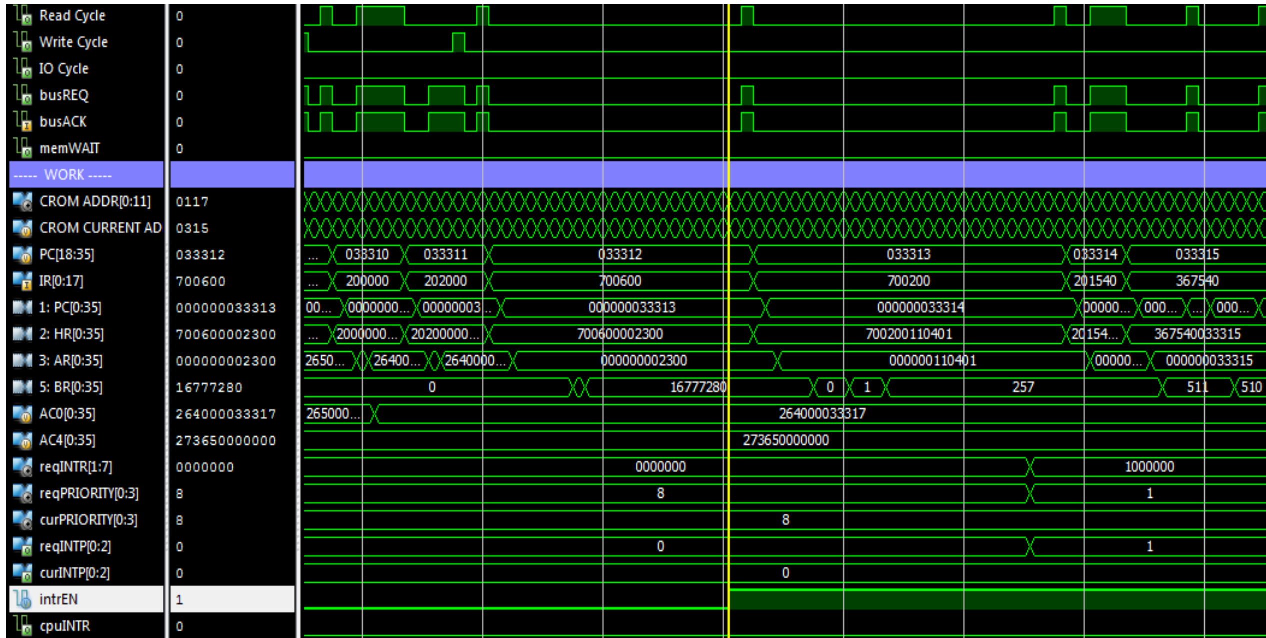
```
33303          HALTPI                  ;FILL INTERRUPT LOCATIONS WITH HALTS
33304          CLRPI
33305          CLRAPR
33306          MOVE    [JSP UUO]       ;SET TRAP TO HALT
33307          MOVEM   41              ;IN THE UUO TRAP LOCATION
33310          MOVE    [JSR TRP0A]     ;SET PROPER RECOVERY INST.
33311          MOVEM   42              ;INTO CH1 TRAP
33312          CONO    PI,2300         ;TURN ON CHAN1
33313          CONO    APR,LENXER!LSNXER!LAPRP1      ;CAUSE CACHE SWP DONE AND CHAN
33314          MOVEI   13,1000         ;SET UP LOOP OF TEN TO WAIT FOR INT.
33315          SOJG    13,.            ;AND WAIT^
33316          JRST    .               ;LOOP ON SELF^
33317   TRP0A: 0
33320   0047:  SKIPE   MONFLG          ;RESET FLAGS IF IN MONITOR
```

The Simulation

The Verilog Simulation is presented below. Of particular interest:

1. The Program Counter (PC) is shown in the 9th trace from the top which is labeled `PC[18:35]`. The "`[18:35]`" notation defines a range of bits.
2. The `PC[0:35]` signal which shown below that is an internal signal that is incremented right after the instruction is fetched. Therefore it is not the PC of the current instruction. During a jump or skip instruction it may also point to an instruction that is never fetched or executed. For the purpose of this example, it is confusing and is best ignored.
3. The `HR` register (labeled `HR[0:35]`) contains the OPCODE of the current instruction.
4. The `AR[0:35]` and `BR[0:35]` are shown.
5. The contents of `AC0` and `AC4` are shown below that. Notice that the contents of `AC0` are correctly modified by the instruction at `PC=033310`. AC4 is not used in this code and may be ignored. See listing file.

6. The interrupts are enabled during the instruction at PC=033312. This is visible by examining the intrEN signal which is high-lighted near the bottom of the figure.
7. The CPU Interrupt is never asserted during the instruction al PC=033313. See cpuINTR signal at bottom of the figure. **This is a problem!**.



Background Information:

The KS10 (in fact all PDP10s) have a 7 level priority interrupt system with interrupts numbered 1 through 7. Interrupt 1 is the highest priority and Interrupt 7 is the lowest priority. Interrupt 0 is not valid and is used to represent an *interrupt not active* condition. Normally 3-bits would be sufficient to describe the interrupt state (7 priority levels) except that this representation is numerically awkward. If Interrupt 1 is the highest priority and Interrupt 7 is the lowest priority then an *interrupt not active* condition needs to be a numerically lower priority than the lowest interrupt – not higher. To work around this issue, the interrupt controller adds a fourth-bit which represents this *interrupt not active* condition and is numerically lower than the lowest interrupt priority. When an interrupt is requested that is of a higher priority than the current interrupt priority, then a CPU Interrupt signal is generated. This extra bit is stripped off once the interrupt priority comparison is evaluated.

The interrupt priority representation is summarized in the table below.

Priority	Interrupt Controller Representation	KS10 Priority Interrupt Representation	Notes
1	0001	001	Highest Priority
2	0010	010	
3	0011	011	
4	0100	100	
5	0101	101	
6	0110	110	
7	0111	111	Lowest Priority
	1000	000	Inactive

The DEC KS10 Priority Interrupt implementation uses a pair of TTL Priority Encoders and a 4-bit Magnitude Comparator. This implementation has been (mostly) replicated in the KS10 FPGA.

In the Verilog code example below, anyone conversant in the C Programming Language should recognize the *ternary if statement* which operates as follows:

```
variable = condition ? value_if_true : value_if_false
```

The Relevant Verilog Code:

```
 0: // Requested Priority
 1: wire [0:3] reqPRIORITY =
 2:            (~intrEN     ? 4'b1000 :  // Disabled
 3:             reqINTR[1] ? 4'b0001 :  // Highest priority
 4:             reqINTR[2] ? 4'b0010 :
 5:             reqINTR[3] ? 4'b0011 :
 6:             reqINTR[4] ? 4'b0100 :
 7:             reqINTR[5] ? 4'b0101 :
 8:             reqINTR[6] ? 4'b0110 :
 9:             reqINTR[7] ? 4'b0111 :  // Lowest priority
10:             4'b1000);               // Nothing active
11:
12: assign reqINTP = reqPRIORITY[1:3];
13:
14: // Current Priority
15: wire [0:3] curPRIORITY =
16:            (curINTR[1] ? 4'b0001 :  // Highest priority
17:             curINTR[2] ? 4'b0010 :
18:             curINTR[3] ? 4'b0011 :
19:             curINTR[4] ? 4'b0100 :
20:             curINTR[5] ? 4'b0101 :
21:             curINTR[6] ? 4'b0110 :
22:             curINTR[7] ? 4'b0111 :  // Lowest priority
23:             4'b1000);               // Nothing active
24:
25: assign curINTP = curPRIORITY[1:3];
26:
27: // If the requested interrupt priority is higher than
28: // the current interrupt priority, then an interrupt
29: // to the CPU is generated.
30:
31: reg cpuINTR;
32: always @(posedge clk or posedge rst)
33:   begin
34:      if (rst)
35:        cpuINTR <= 0;
36:      else if (clken)
37:        cpuINTR <= (reqINTP < curINTP);
37:   end
```

Analysis:

A quick glance reveals that the priority comparison is performed using the wrong two signals! The extra bit was added to represent the *no interrupt present* condition and then never used in the priority comparison - a simple coding mistake.

The "Fix":

The obviously correct 'fix' is to change line 37 as shown below:

```
31: reg cpuINTR;
32:   always @(posedge clk or posedge rst)
33:     begin
34:        if (rst)
35:          cpuINTR <= 0;
36:        else if (clken)
37:          cpuINTR <= (reqPRIORITY < curPRIORITY);
38:     end
```

The ‘one-liner’ code change is applied and the simulation is re-run.

Now note that the `cpuINTR` signal is asserted by the instruction at `PC=33313`, as it should be. Still there is no interrupt to the KS10 CPU.

This is progress but something is broken somewhere else, also.