

Lab 2: Processor Exceptions

Kyle Swanson

February 17, 2015

The goal for this lab was to experiment with and better understand processor exceptions and system faults. We ran code that caused faults, code that handled the faults, and code that used the Mode Clock Gating, which prevented the fault. During these experiments, we made sure to investigate and better understand how the different registers work, how they relate to the vector table, context switching, and fault handling.

In the first portion of the lab, we used code that intentionally caused a fault. It simply started a main function, which then branched into an aptly named *CauseMemFault* branch. As far as I understand, this caused the fault by trying to write to a portion of memory that is responsible for the LED on the board. However, if the GPIO port is not turned on, it causes a hard fault when you try to write to its portion of memory. The fault causing portion was *MOVS R1, #0x02U*, which if my assembly memory serves right, is trying to move 0x02 into the first register. See figure 1 for more details.

CauseMemFault

```
LDR      R0, datF          ; GPIOF data all bits
MOVS     R1, #0x02U        ; LED red
STR      R1, [R0]          ; BOOM!
NOP                               ; Pause, wait for it...!
BX       LR                ; Return to caller
```

Figure 1: The hard fault causing portion of code.

While we were executing this first portion, I noticed some changes over the first lab. First, the program counter was much higher than the first lab, in this instance it was `0x00000044`, I suspect that this is due to the extra code to define the handlers above the actual executing code. Just an interesting note I thought. Next we stepped through the code until the exception happened. According to the ISPR, the exception was `0x003` which according to the TI documentation, is a Hard Fault. Looking at the disassembly view, the program stopped changing states at the `BusFaultHandler`. I noted that the location of the stack pointer, `0x20007FE0` contained the same information as register 0. I suppose this is the information that was at the position on the stack, and also in that register when the hard fault occurred.

R0	=	0x400253FC
R1	=	0x00000002
R2	=	0x00000000
R3	=	0x00000000
R4	=	0x00000000
R5	=	0x00000000
R6	=	0x00000000
R7	=	0x00000000
R8	=	0x00000000
R9	=	0x00000000
R10	=	0x00000000
R11	=	0x00000000
R12	=	0x00000000
SP	=	0x20007FE0
LR	=	0xFFFFFFFF
<input checked="" type="checkbox"/> APSR	=	0x00000000
<input type="checkbox"/> IPSR	=	0x00000003
Exception_Number	=	0x003
<input type="checkbox"/> EPSR	=	0x01000000
T	=	1
ICIIT	=	0b00000000
PC	=	0x00000042
<input checked="" type="checkbox"/> PRIMASK	=	0x00000000
<input checked="" type="checkbox"/> BASEPRI	=	0x00000000
<input checked="" type="checkbox"/> BASEPRI_MAX	=	0x00000000

Figure 2: Status of the registers after the hard fault.

In the next portion of the lab, we created a HardFault Handler. For the first part of this portion, we compared the vector table of our code, to the vector table found in the documentation. We found that the *Stack Pointer*, *Reset*, and *Bus Fault* all corresponded to the documentation.

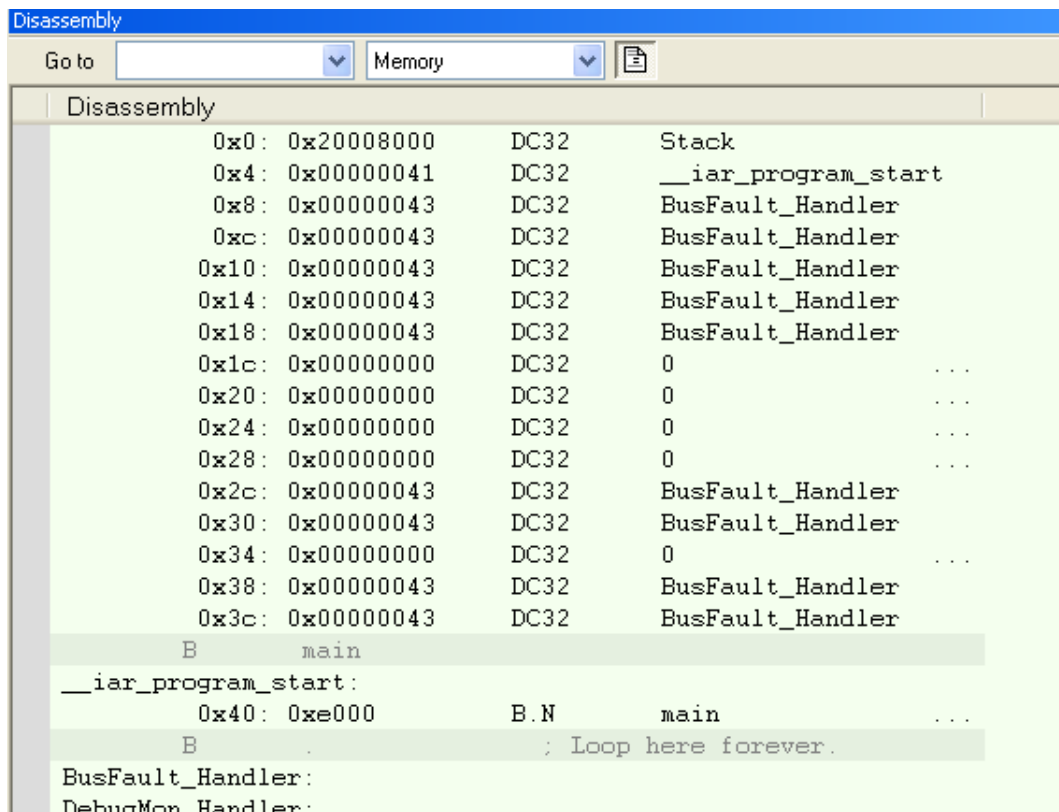


Figure 3: Vector table found in our program. Note 0x0, 0x4, & 0x14

Next we edited the code to add a hard fault handler. This simply captured a hard fault, and sent it into an infinite loop. In a real system, I think you would have this manage the fault, perhaps display an error code, I wonder if a Windows BSOD is an example of this. After we ran the code with this modified handler, we found that the handler was then executed when the

program caused the hard fault. We also found that the vector table was closer to that which was specified in the documentation. That is, the *Stack Pointer*, *Reset*, *Bus Fault*, and now the *HardFault* all corresponded to the documentation.

For the final experiment, we enabled Mode Clock Gating. To do this, we needed to inspect page 340 of the TI documentation. Note the figure below. According to the documentation, to enable mode clock gating, we needed to set the 6th bit in a certain register to 1. This was accomplished with the following assembly command: *MOVS R1, #0x20*. Note, $0x20 = 00100000$.

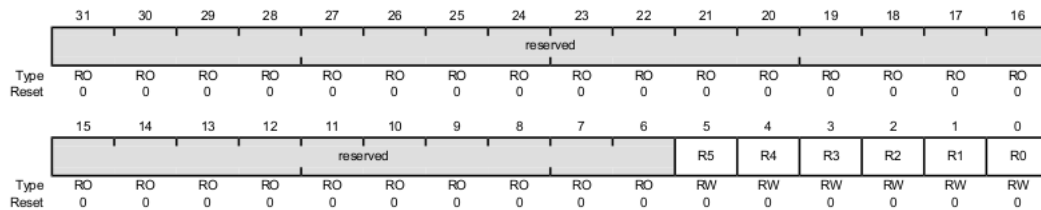


Figure 4: Clock Mode Gating diagram

Since the instructions now set mode clock gating, the *STR R1, [R0]* code segment no longer caused a hard fault. It successfully went through the *CauseMemFault* branch, and returned to the main, where it then looped forever.

I had a much easier time comprehending this lab than the 1st. I definitely have a better understanding of register manipulation, monitoring system faults, and their relation to the vector table.