CS150 Fall 2010 Project Checkpoint 1 – version0.08

Interrupts!

Interrupts provide a mechanism for devices external to the processor to steal processor cycles without the “normal” software knowing about it.

You will be implementing interrupts in Austin’s Pipelined MIPS (hereafter APMIPS) implementation. There are many different versions of the MIPS processor, and we’ll try to stay consistent with what’s in your book, and what’s in the simplest of the commercially available MIPS processors.

When an interrupt occurs the current instruction is completed and then the PC is loaded with the interrupt vector, which is the address of the interrupt service routine (ISR). When the ISR is done, execution resumes at the next instruction as if there had never been an interruption. In a pipelined processor, the definition of “current instruction” can be a little fuzzy, so you get to decide what the simplest implementation is. The interrupt must save the address of the next instruction. On the MIPS, this is done in a register called EPC, or Exception Program Counter. For our implementation there will be only one ISR, at address 0x0040.000C. The type of interrupt (timer, UART, etc.) is specified in the Cause register with a set of bits that indicate an Interrupt Pending (IP). The ISR will need to examine the Cause register to figure out what caused the interrupt. Interrupts can be enabled and disabled altogether using a single Interrupt Enable (IE) bit in the Status register. Individual interrupts can be turned on and off using an Interrupt Mask (IM) which is also in the Status register. All of these named registers live in a register file like structure called CoProcessor0.

When a device requests an interrupt, the corresponding bit is set in the Cause register. This does not necessarily generate an interrupt though. If the interrupt mask for that interrupt is zero, or if interrupts are disabled, then no interrupt will be generated. Interrupts are disabled (at least initially) by the hardware in the ISR. So for example, if a device requests an interrupt while the ISR is executing (servicing another interrupt), then the IE bit will be set to zero, and a new interrupt will not be generated. As soon as the ISR is finished and interrupts are re-enabled in software, then the pending interrupt will cause a new interrupt to be generated (assuming the interrupt mask enables interrupts from that particular source).

The ISR is responsible for clearing the appropriate IP bit in the Status register.

CoProcessor 0

CP0 is really just a collection of 32 registers that are not a part of the normal register set for the MIPS. In some MIPS implementations there are actually completely separate register files called shadow registers which contain copies of all of the usual MIPS registers (like $s0, $t0, etc.), but that’s a separate capability from what CP0 provides. We won’t be using shadow registers, but we will use CP0. We won’t need most of the registers in CP0. The ones we will need are listed below:

EPC register (CP0 register 4)

The Exception PC register, this stores the PC of the interrupted instruction.

Count register (CP0 register 9)

The Count register increments at the CPU clock rate.

Compare register (CP0 register 11)

Used to set the timer.

Status register (CP0 register 12)

```
Status<15:10> Hardware interrupt mask, or IM<7:2>. 1=enabled, 0=disabled.  1-1 match with Cause <15:10>
   Status<15> timer
   Status<14> RTC
   Status<13>
   Status<12>
   Status<11> UART1
   Status<10> UART0
Status<9:8> Software interrupt mask, IM<1:0>.  We won't be using these.
Status<0> Interrupt enable. 1=enabled, 0=all interrupts disabled
```

Cause register (CP0 register 13)

```
Cause<6:2> - exception code.  0=Hardware Interrupt
Cause<15:10> - IP, Interrupt Pending hardware source, 1 bit per source
   Cause<15> timer
   Cause<14> RTC
   Cause<13>
   Cause<12>
   Cause<11> UART1
   Cause<10> UART0
```

Timer

The Coprocessor should generate a timer interrupt when Counter=Compare, and a Real Time Clock (RTC) interrupt when Count rolls over from 0xFFFF_FFFF to 0x0.
When the Compare register is equal to the Count register, Cause<15> (interrupt pending 7, or IP7) is set. If both Status<15> (interrupt mask 7, or IM7) and Status<0> (interrupt enable) are set, then an interrupt is generated immediately.
Writing to the Compare register clears Cause<15>.
When the Count register rolls over, an RTC ISR should cause the software Real Time Clock to increment.

New Instructions

You will need to implement several new instructions to get interrupts to work.

mfc0, mtc0

These instructions provide a mechanism for moving data to and from the CP0 registers.

From Appendix B of your book, both instructions use opcode 010000 (16), and the value in rs determines if the instruction is move from (rs=0) or move to (rs=4) CP0. The co-processor register is always in rd, and the MIPS register is in rt.

```
mfc0 k0, Cause      # opcode=16, rs=0, rd=13, rt=26; get the Cause register

mtc0 k1, Status     # opcode=16, rs=4, rd=12, rt=27; put k1 in Status register
```

Adding these two instructions should not take much work. Adding two entries to the case statement in the Control module, and adding a few control signals should be the most straightforward way to handle them.

eret

Instead of needing an eret instruction, we can be clever and save ourselves some work. We merely use a mfc0 instruction to grab the value of our EPC from CP0 and then jr to this value while re-enabling interrupts with a mtc0 in the delay slot of the jr.

Example code

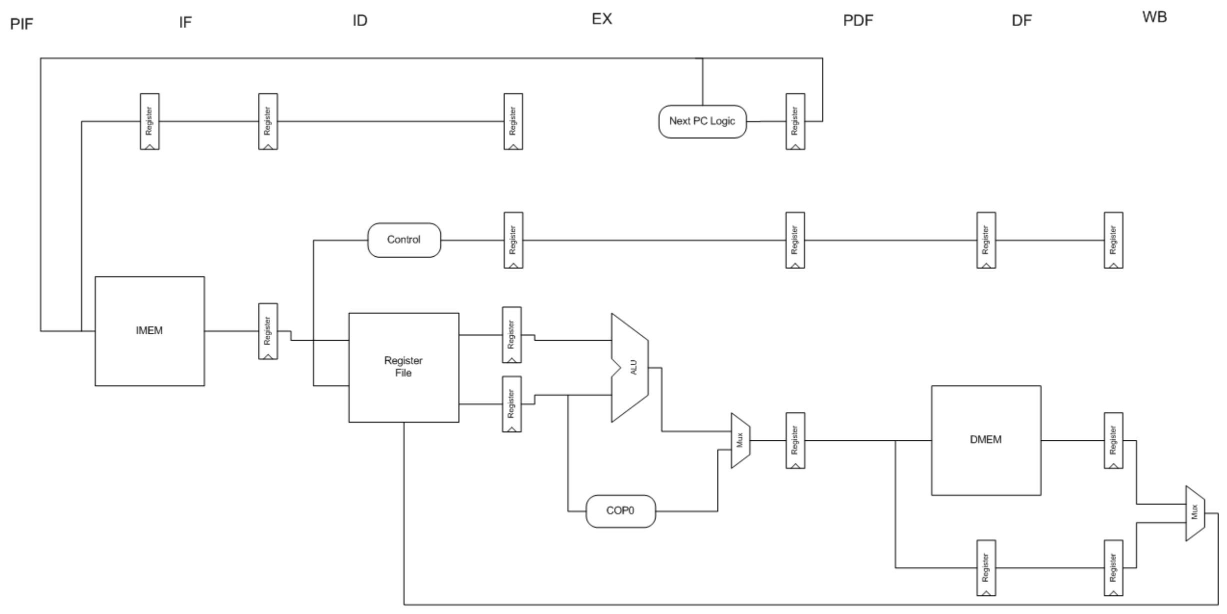
From the MIPS M4K software users manual [1], this code converts a single interrupt handler into a vectored interrupt.

```
IVexception:
       mfc0 k0, Cause              /* Read Cause register for IP bits */
       mfc0 k1, Status             /* and Status register for IM bits */
       andi k0, k0, M_CauseIM      /* Keep only IP bits from Cause */
       and k0, k0, k1              /* and mask with IM bits */
       beq k0, zero, Dismiss       /* no bits set - spurious interrupt */
       clz k0, k0                  /* Find first bit set, IP7..IP0; k0 = 16..23 */
       xori k0, k0, 0x17           /* 16..23 => 7..0 [ksjp: should this be srl?] */
       sll k0, k0, VS              /* Shift to emulate software IntCtlvs */
       la k1, VectorBase           /* Get base of 8 interrupt vectors */
       addu k0, k0, k1             /* Compute target from base and offset */
       jr k0                       /* Jump to specific exception routine */
       nop
Dismiss:
       eret          /* return to interrupted code */

SimpleInterrupt:
/*
* Process the device interrupt here and clear the interupt request
* at the device. In order to do this, some registers may need to be
* saved and restored. The coprocessor 0 state is such that an ERET
* will simple return to the interrupted code.
*/
       eret                        /* Return to interrupted code */
```

APMIPS

A large portion of the work is going to be in augmenting the APIMIPS datapath. APMIPS is a slightly modified five stage MIPS like the one developed in DDCA or P&H. In both of the previously mentioned formulations of the five stage MIPS, asynchronous reads for instruction and data memories are assumed. Since the block RAMs on the Xilinx FPGAs are synchronous read, we merely add another stage for each memory to delay the pipeline by one cycle. This gives the following stages, PIF(Pre-Instruction Fetch), IF(Instruction Fetch), ID(Instruction Decode), EX(Execute), PDF(Pre-Data Fetch), DF(Data Fetch), and WB(Write Back). A high level diagram of the datapath, with the complicated control and hazard resolution logic removed, is included below.



You will mostly be working with the "Next PC Logic" and connecting the CoProcessor as shown. The first step will be attaching the CP0. The most straightforward way to accomplish this is to keep the CoProcessor completely contained within the Execute stage. Once you can read and write the registers contained in CP0, the next step is to make the processor jump to the correct address when an interrupt is requested, and store the correct PC in the EPC register located in CP0. Beware of the architectured delay slot as well as the two instructions after the delay slot that are killed. If an interrupt happens during any of three instructions directly following a branch or jump, you should delay the interrupt to make sure that the branch or jump is actually taken. Otherwise when you return to the interrupted PC the jump will not be taken and the ISA will be violated.

Prelab

Create a block diagram of your CoProcessor0, as well as your APMIPS datapath with the CoProcessor attached, showing all control lines, etc.

Make sure that you understand the example code.

Checkoff

Implement a 1 second clock, and send the time out UART1 (or UART0 if the UART1 connector isn't ready yet).

Implement an interrupt driven UART0 interface. Bring the appropriate IP bit out to a pin and measure the interrupt latency and ISR duration using an oscilloscope.

Use the timer to count the number of cycles necessary to count to 100,000,000. The result should be sent out UART0. Counting should be in four different modes: 1) using a register to hold the count; 2) using a memory location to hold the count; 3) using a register to hold the count, but with a function call to implement "+1"; 4) using a memory location to hold the count, and a function call to implement "+1". Use UART0 to set which mode the program counts in. Pressing 'r' should have the next count use the register, pressing 'm' should have the next count use the memory location, pressing 'f' should use the function call for +1, and pressing 'a' should use addi for +1.

Does the number of cycles in a given mode depend on how many times you pressed a key? Is this result consistent with the measurements from the oscilloscope?

References

[1] MIPS M4K Software Users Manual, available from mips.com