Computer Science 324
Computer Architecture
Mount Holyoke College
Fall 2007

Topic Notes: Exceptions and Interrupts

Exceptions and Interrupts

We have seen how to build a relatively simple datapath and control system. The same ideas and techniques can be used to build computers that implement a more complete instruction set, such as the full MIPS ISA.

Our implementation so far assumes that everything will go as planned.

- all instructions are valid
- arithmetic operations complete correctly
- all data is immediately available
- the operating system has not yet become involved

Conditions that cause an "unusual" operation – one that does not simply proceed to the next instruction specified by the `PC` – are called *exceptions* or *interrupts*.

Our text uses the term "exception" for an event generated by the processor and "interrupt" for one coming from outside. However, different texts and ISAs use different conventions. We will use the terms pretty much interchangeably.

So far, there are only two things we have seen that can cause an exception: arithmetic overflow and the attempted execution of an undefined instruction. We will see others soon and consider them when they arise.

Data Path Augmentation for Exceptions

The MIPS approach to handling exceptions involves several steps:

1. the address of the instruction that caused the exception is stored in a new register: the *exception program register* or `EPC`
2. control is transferred to the operating system to take appropriate action by loading the `PC` with a specified address
3. the OS handles the situation, which may involve

   (a) providing a service to the user program (e.g., reading some information from an I/O device)
   (b) handling an overflow
   (c) stopping a user program that has executed an illegal instruction

4. if the offending program is allowed to continue, the OS restarts its execution based on the value in the `EPC`

In the MIPS ISA, the OS is informed of the exception's cause by storing a value in an additional register `Cause`.

Other systems use an *interrupt vector*, which specifies different target addresses to be used by the OS based on the reason for the interrupt/exception.

We will consider the MIPS approach, and look at how to augment our data path and control to handle the two exception types that could happen in our MIPS subset.

The two additional registers are, as mentioned above:

1. `EPC`: holds the 32-bit address of the instruction that was being executed when the exception occurred
2. `Cause`: records the reason for the exception

For our simplified system, there are only two causes. We will store a 0 in `Cause` for an undefined instruction and a 1 for an arithmetic overlow.

This also requires some new control lines and options:

- `EPCWrite` controls the writing of `EPC`
- `CauseWrite` controls the writing of `Cause`
- `IntCause` determines whether to store a 0 or a 1 in `Cause`
- a 4th option for `PCSource` to load the `PC` with the OS exception handling code address: 80000180_{16}.
- a new ALU output to compute `PC`-4 since we want to store the address of the currently executing instruction, and we will detect the exception conditions after we have already incremented the `PC` – fortunately this is easy, since we can already feed in the `PC` to the first ALU input, we can already feed in the constant 4 to the second ALU input, and we can already force the ALU to perform subtraction

Figure 5.39 shows the data path with these enhancements.

Control Augmentation for Exceptions

Next, we need to augment our control to implement this. For the finite state control, we need to make two additions:

1. a new state where we go if the instruction is something other than one of our defined instructions when transitioning from state 1
2. a new state where we go from state 7 if the ALU operation results in overflow

Figure 5.40 shows the finite state control with these additions.

The control lines in the new states are as we would expect.

The transition from 1 to 10 in the illegal instruction case is fairly straightforward. We simply make 10 the next state for all undefined opcodes.

The transition from 7 to 11 is more complicated – we need to look back at the ALU and note that it has an additional output line called `Overflow`. Our control in this case needs to examine the value of this line when determining whether to proceed to 7 on a successful ALU operation or to 11 to handle the overflow condition.

There are complications we are not addressing as well – the MIPS ISA states that an instruction that causes an overflow should have no effect. However, we have already written the result to a register in state 7. So this is a case where we really need to check overflow of the ALU result (from the previous cycle) before doing our write back operation, but this isn't the way the finite state control works. We won't worry about this for now.

If we are using a microprogrammed control, the changes are similar.

- our microinstruction size increases by two bits for the two new control lines that need to be specified
- our dispatch table for instruction decode simply needs to be set to the address of the microinstruction that deals with the bad instruction case
- we would also need to add some capabilities to the microprogrammed control to branch conditionally based on the overflow line, possibly adding a 4th case to the sequencing where we define a conditional jump based on whether the `Overflow` flag is set.

We will return to the ideas of exception handling as they come up in our remaining topics.