Exceptions and interrupts

- An exception or interrupt is an unexpected event that requires the CPU to pause or stop the current program.
- Exception handling is the hardware analog of error handling in software.
  - Classes don't often talk about errors, so it's easy to forget them.
  - Proper error handling is very important in real systems, and a lot of effort is often devoted to error checking and recovery.
- We'll find that exception handling requires support from both hardware and software (the operating system) sides.

Exception handling

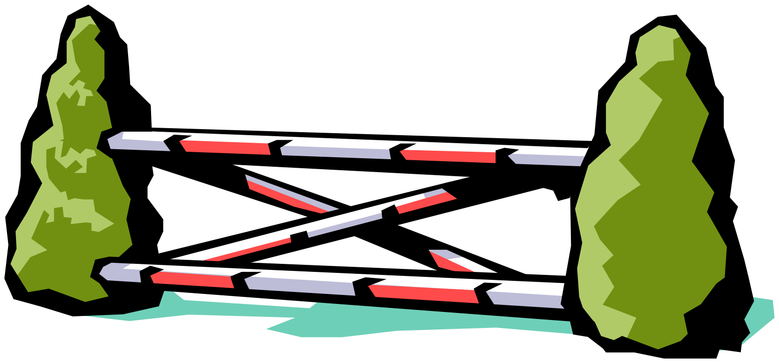
- Exceptions are typically errors that occur within the processor.
  - The CPU tries to execute an illegal instruction opcode.
  - An arithmetic instruction overflows, or attempts to divide by 0.
- There are two possible ways of resolving these errors.
  - The operating system can force a program with a serious error to quit.
  - Smaller errors may cause an error message to be sent to the program.
- Modern languages like Java allow programmers to "catch" exceptions so applications can deal with errors more gracefully.

Interrupt handling

- Interrupts are external errors that require the processor's attention.
  - The user presses the machine's sleep or reset buttons.
  - Peripherals and other I/O devices may need attention.
  - The virtual memory system needs to access the hard disk to complete a lw or sw instruction.
- These situations are not really errors.
  - They happen normally. (We'll talk about memory and I/O later on.)
  - The interrupted program usually needs to *resume* execution after the interrupt is handled.
- It is the operating system's responsibility to do the right thing, such as:
  - Save the current state and shut down the hardware devices.
  - Find and load the correct data from the hard disk
  - Transfer data to/from the I/O device, or install drivers.

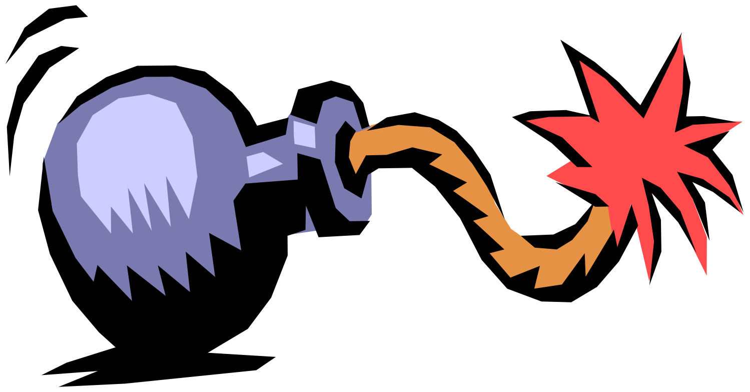
The hardware/software interface again

- The most uniform approach is to have the operating system handle both exceptions and interrupts.
- The operating system code contains an exception handler, which decides how exceptions and interrupts should be processed.
- The exception handler needs to know two things.
  - The cause of the exception (e.g., overflow or page fault).
  - What instruction was executing when the exception occurred. This helps the operating system report the error or resume the program.
- This is another example of interaction between software and hardware, as the cause and current instruction must be supplied to the operating system by the processor.

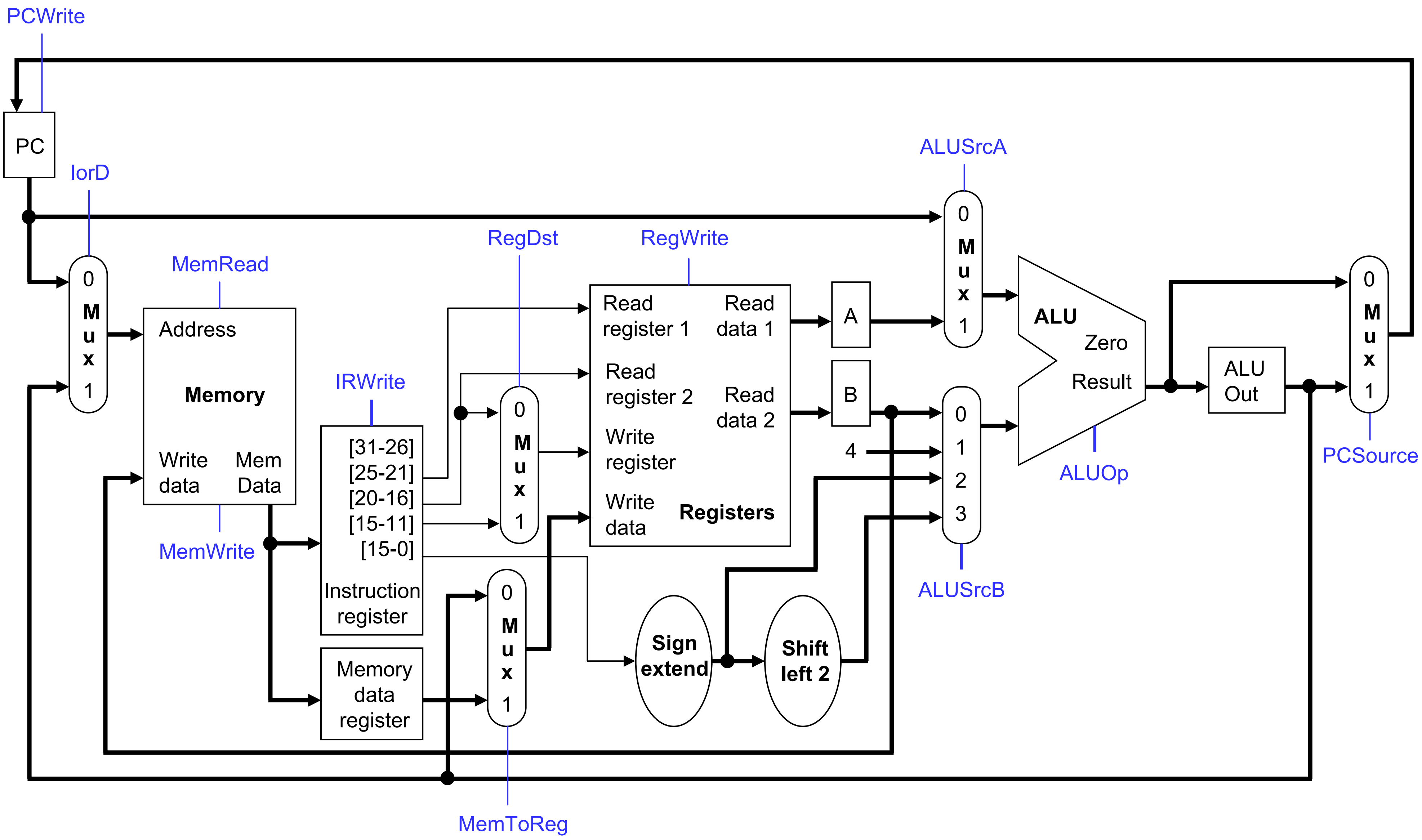


The CPU's basic responsibility

- The CPU first stores the relevant information for the operating system.
  - The exception is stored in a special Cause register.
  - The instruction that was executing when the exception occurred is stored in the EPC (exception program counter) register.
- Then the processor transfers control to the operating system by placing the exception handler's address into the PC.
- We'll look at how this might work in our multicycle datapath, and then also point out some issues with a pipelined datapath.



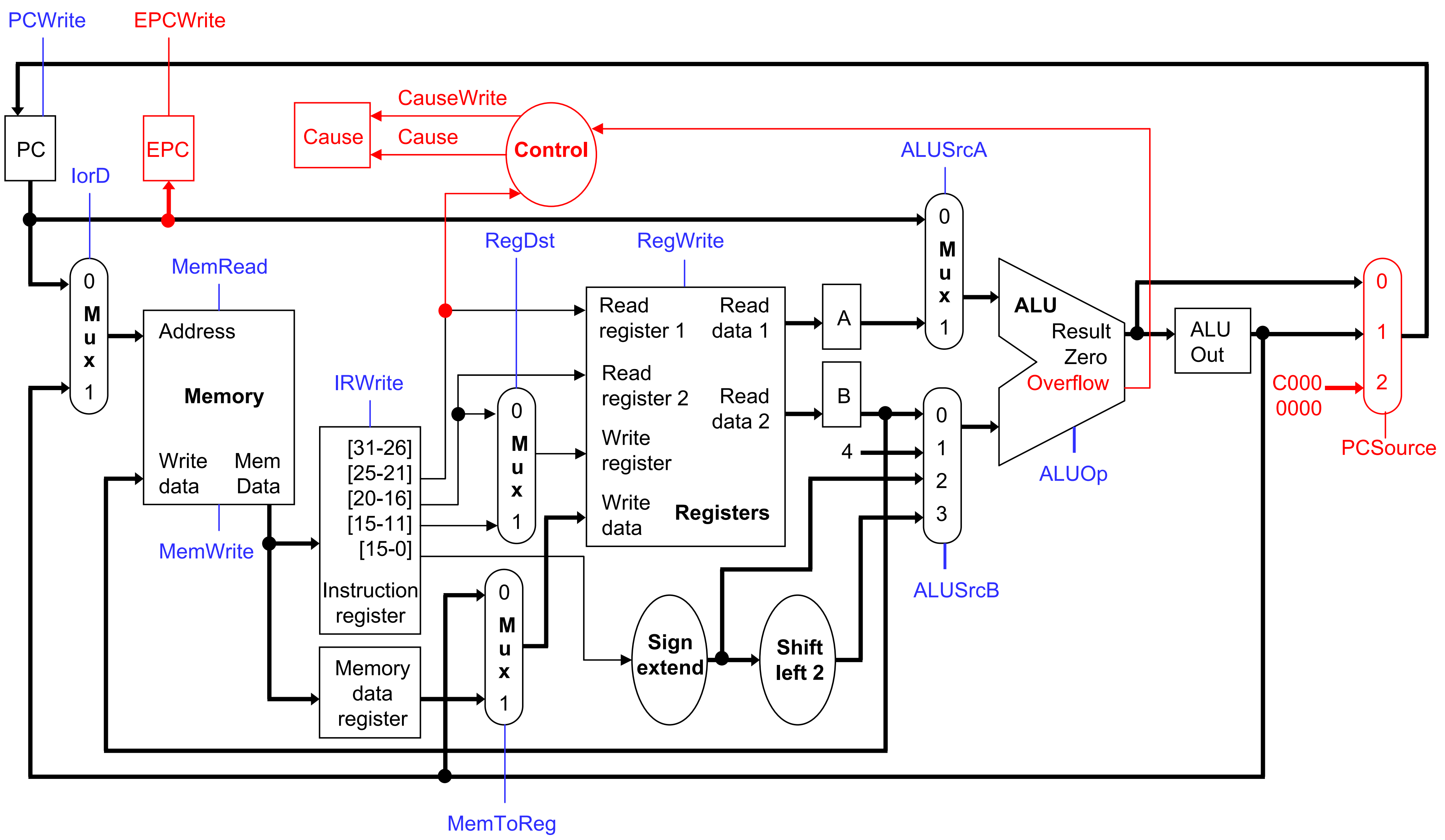
The original multicycle datapath



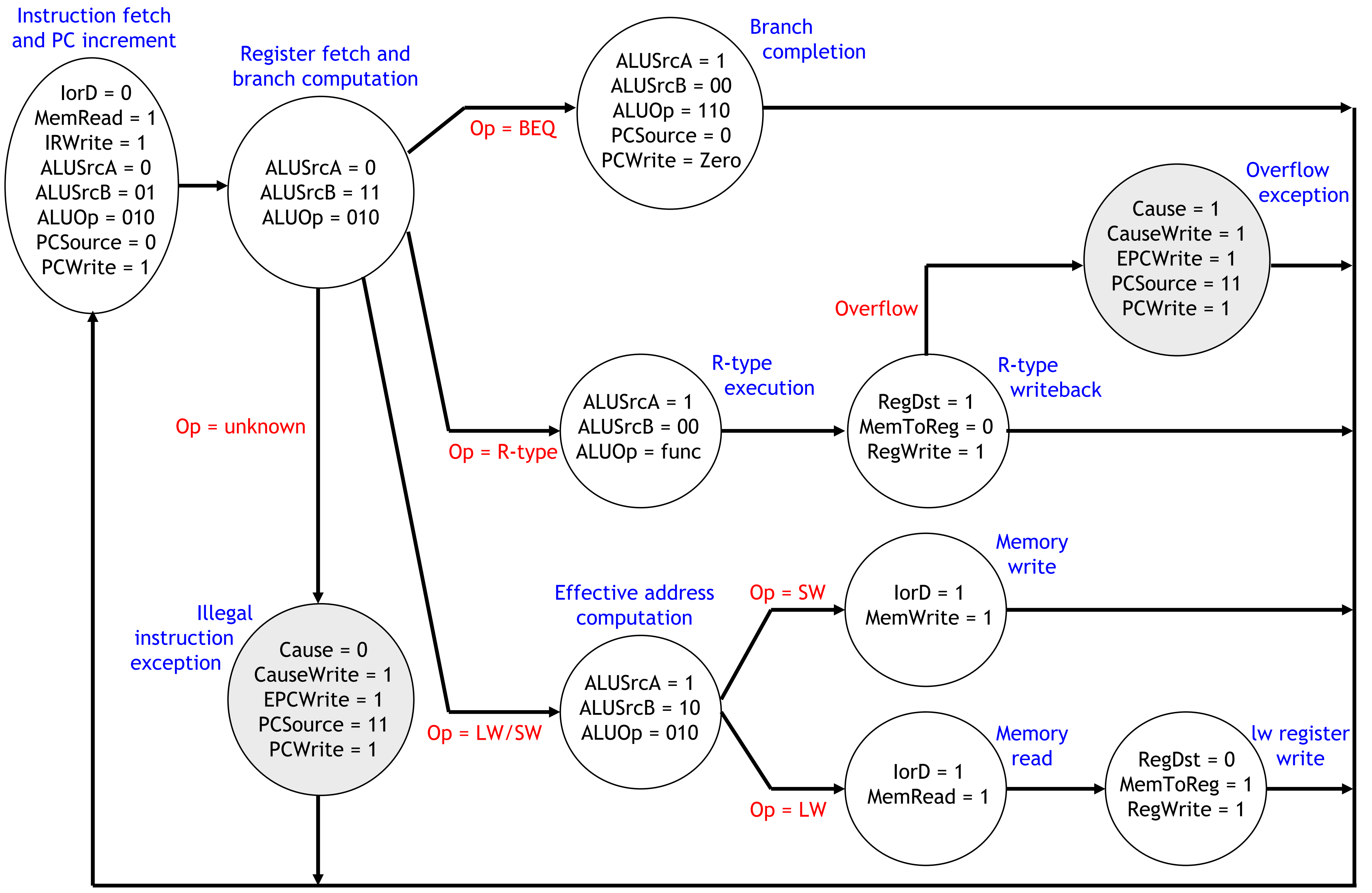
Exceptions for our multicycle CPU

- There are only two possible exceptions in our simple processor.
  - An illegal instruction can be detected by the control unit when it tries to decode the instruction word.
  - We can modify the ALU to generate an overflow signal for arithmetic overflows. The control unit then checks the overflow signal and causes an exception if needed.
- If an exception occurs, the control unit needs to do several things.
  1. Store the current PC in the EPC register (the diagram on the following page actually stores PC + 4 for simplicity).
  2. Store the exception cause in the Cause register.
     - We'll say illegal instructions are Cause = 0.
     - Overflows will be represented by Cause = 1.
  3. Finally, set the PC to the address of the operating system's interrupt handler. This address must be known by the CPU; we'll assume it's at memory location C0000000 in hexadecimal.

Multicycle datapath with exceptions



Multicycle control unit changes

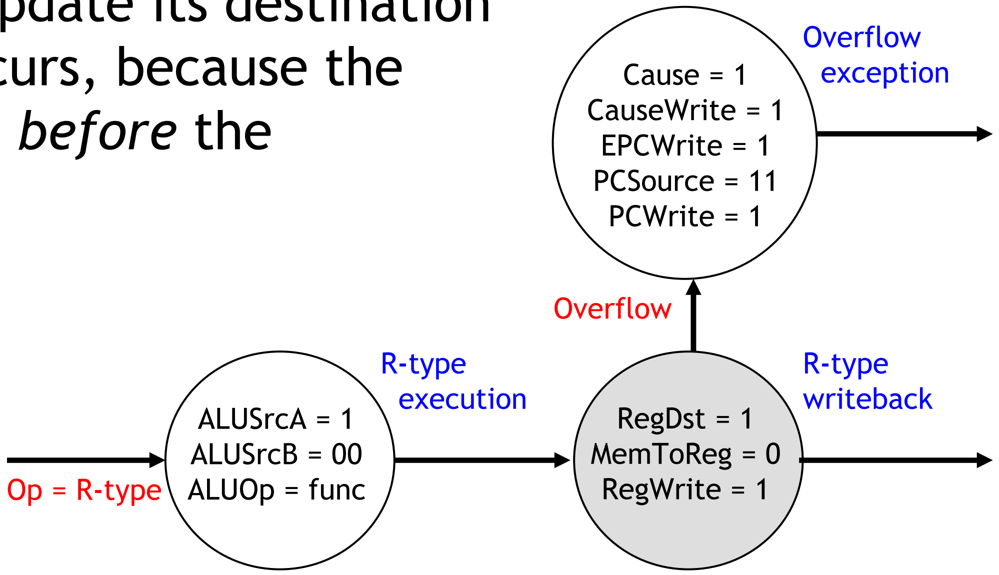


When to interrupt the processor

- An arithmetic instruction will update its destination register even if an overflow occurs, because the "R-type writeback" step occurs *before* the overflow exception step.
- This may not always be what is desired. For example, if the overflowed instruction was

  ```
  add $t1, $t1, $t2
  ```

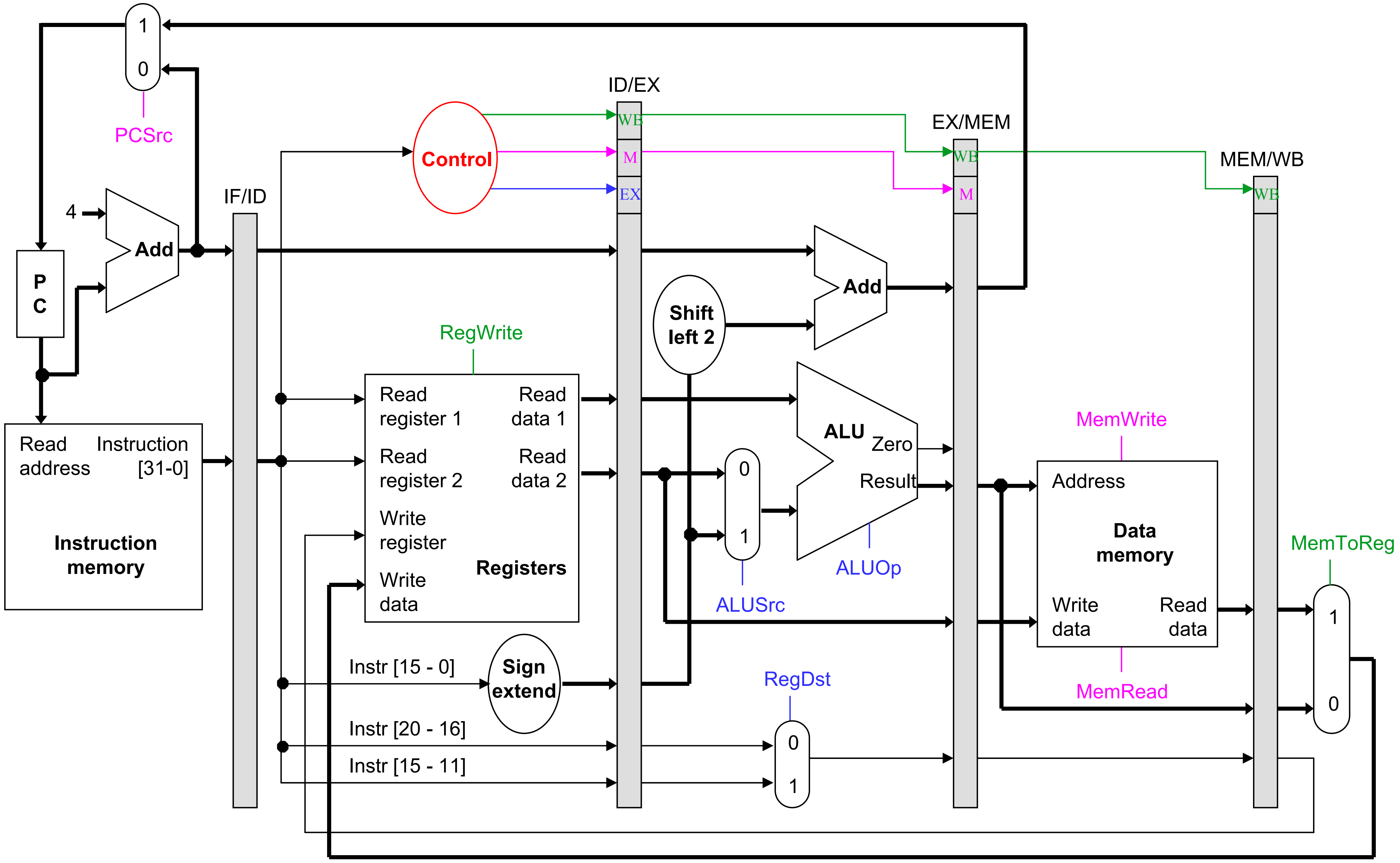
  the writeback stage would overwrite the original value of $t1 before the exception occurs, possibly making debugging much harder.



Other kinds of exceptions

- Exceptions can occur in different instruction execution stages, depending on the actual instruction set architecture and datapath design.
  - For example, a virtual memory page fault might be raised during the instruction fetch or data memory stages.
  - External interrupts can also arrive at any time.
- Exception handling is a major challenge in CPU design.
  - Handling all of the possible exceptions can result in a very large state diagram and a very complex control unit.
  - Remember that you have to be careful not to make the control unit too complicated, or cycle times will increase.
- As an analogy, error handling in programs usually yields longer and more complex code.

Our pipelined datapath



Pipelining and exceptions

- In a pipelined design, exceptions are another form of control hazard— just like branches, they alter the normal program flow so it's not always clear what the next instruction should be.
- Handling exceptions is much more difficult with pipelining, since there are several instructions executing at once.
  - The control unit has to determine which of the several instructions in the pipeline caused the exception.
  - It's also possible for multiple exceptions to occur in the same cycle! For example, an instruction in its EX stage could overflow at the same time another instruction causes an illegal opcode exception.

Precise exceptions

- Deciding where to interrupt or stop the pipeline is also difficult. Ideally, a processor would implement precise exceptions.
  - All instructions before the offending one should complete execution.
  - The CPU should stop on the excepting instruction, storing its address in the EPC register.
  - Instructions *after* that one should not be executed.
- In a pipelined CPU, this means that the control unit must ensure some of the instructions in the pipeline complete, while others are flushed.
  - Again this can lead to very complex control.
  - Flushing for exceptions can limit the performance of deep pipelines, just like flushing for branches. If an instruction in stage 17 causes an exception, then the following 16 instructions are flushed and might need to be re-executed.
- Precise exceptions are not difficult to implement in a single or multicycle datapath, since there is only one instruction active in any given cycle.

Summary

- Exceptions and interrupts are different hardware events that force the CPU to either pause or stop the running program.
- The operating system and processor work together to handle exceptions.
  - The OS provides an exception handler to process the errors.
  - The processor records and passes the exception program counter and exception cause to the operating system.
- Handling all possible exceptions and interrupts can lead to complex and slow control units and processors.
- Exception handling presents special challenges for pipelined datapaths.
  - Multiple exceptions can occur simultaneously, so supporting precise exceptions is difficult.
  - Complicated control units and excessive flushing can reduce the CPU performance.