Instruction Pipelining; Interrupts

CS 350: Computer Organization & Assembler Language Programming

Lecture 16 [number fixed Thu 3/28]

Executing one instruction at a time

- Say we number the phases of the instruction cycle as 1: Fetch Instruction; 2: Decode Instruction, 3: Calculate Addresses, 4: Get Operands; 5: Execute operation; 6: Store Results
- In non-pipelined execution, as an instruction executes phase 1, the hardware for phases 2 – 5 is unused. Then hardware for phase 2 becomes active while hardware for phases 1 and 3 – 5 sit unused, and so on.
- Say each phase takes time $T_p$ to execute (in real life, different phases take different amounts of time).
- Let $K$ be the number of phases ($K = 6$ here). Then it takes $K \times T_p$ time to execute a single instruction. To execute $n$ instructions takes time $n \times K \times T_p$.
- Why use $K$ instead of just 6? If we break up the 6 phases into many smaller sub-phases, we can more-accurately track the timing. For example, it may take 15 to 20 clock cycles to retrieve a word of memory. Decoding an instruction will take much fewer cycles.

Instruction Pipelining

- In instruction pipelining, we try to keep the hardware for all phases active simultaneously by working on $K$ instructions. The instruction at address $A$ is nearly complete because it’s executing phase 6; the instruction at $A+4$ it is executing phase 5, the instruction at $A+8$ is executing phase 4 and so on.
- In general, if the instruction at $A$ is in phase $K$, and we number the instructions at $A, A+4, A+8, \ldots$ as instructions 0, 1, 2, …, then instruction $n$ is at $A+4n$ and is executing phase $K-n$.
- Assuming the pipeline is empty when we begin, instruction 0 finishes at time $K \times T_p$, but after that, instruction 1 finishes at time $(K+1) \times T_p$, instruction 2 finishes at time $(K+2) \times T_p$, …, and instruction $n-1$ finishes at time $(K+n-1) \times T_p$. So to execute $n$ instructions (numbered 0, 1, …, $n-1$) takes time $(K+n-1) \times T_p$.
- Clearly, $(K+n-1) \times T_p$ is a big improvement over the non-pipelined time $n \times K \times T_p$.
  - The speedup is $(n \times K \times T_p) / ((K+n-1) \times T_p) = n \times K / (K+n-1) \approx n \times K / (K+n) = K / (K+n) / n = K / (K/n+1) = n+1$ with the approximation becoming more accurate as $n$ increases.

Latency vs Throughput

- Latency is the time to execute one instruction; throughput is the number of instructions completed per unit time. Instruction pipelining aims to increase throughput, not decrease latency.
- One analogy uses everyday plumbing: Latency is the time it takes for water to come out of the faucet when you turn it on; throughput is the rate at which water comes out once it does start coming out. (Latency depends on the length of the pipe; throughput depends on the diameter.)
Problems with Pipelining

- The analysis we did assumes that we know exactly what \( n \) instructions we'll be executing in the future, plus the assumption that we could completely overlap the execution of different (or at least neighboring) instructions.

- Both of these assumptions can fail, of course.
  - Knowing the Future: If one of the future instructions is a conditional branch, then we won't know for sure what execution path to take until we do the test.
  - Overlapping Execution: If our instruction depends on the result of an earlier instruction, then at some point, we can't proceed if we don't have that earlier result.
    - For example, with `add t0, t1, t2; add t3, t4, t0`, we can't perform the second `add` until we know the value of \( t0 \) set by the first `add`. (This is the "Read after Write" problem.)
    - Interestingly, swapping the statements causes a different problem; with `add t3, t4, t0; add t0, t1, t2`, we can't complete the second `add` until \( t0 \) is used by the first `add`. (This is the "Write after Read" problem.)

Hazards, Bubbles, and Pipeline Stalls

- A hazard is a problem with one instruction that can cause incorrect execution of future instructions (unless we slow down the pipeline).

- A pipeline stall is a delay in executing an instruction to avoid a hazard.

- A bubble is the activity that occurs to implement the delay (e.g., don't read a new instruction into the instruction register, or don't actually execute an instruction, and so on).

Branch hazard: Not knowing which instructions will execute after a conditional branch.

- Flush the Pipeline: Easiest (and slowest) way to handle a branch hazard: Insert bubbles into the pipeline until we find out which way we're jumping.

- Branch prediction / Speculative execution: Predict which side of the conditional branch we'll take and continue on that basis. For example, we might decide that one direction of a branch is for continuing a loop and use that direction.
  - All the results of the operations after the jump have to be saved and not actually committed until we reach the actual branch. If we guessed correctly, then those saved results can be applied, otherwise we toss them.

Data hazard: A dependency between data in different instructions. (E.g., the read-after-write or write-after-read problems above.)

- Increase Latency: Add bubbles to the pipeline to guarantee that the dependency will be met correctly. For example, with the read-after-write hazard with \( t0 \), insert waits between writing \( t0 \) and reading \( t0 \).

- Out-of-Order Execution: Instead of adding bubbles to the pipeline, execute parts of instructions that don't have the dependency. For example, say with read-after-write, the write is at instruction \( n \) and the read is at \( n+1 \). Instead of adding do-nothing delays into the pipeline, execute parts of instructions \( n+2 \), and/or \( n+3 \), etc. to keep the pipeline full. The results of these future instructions are queued up and applied one after the other, once the time is right: The results of instruction \( n+2 \) are kept, uncommitted, until instruction \( n+1 \) finishes. Similarly, results of \( n+3 \) wait until \( n+2 \) finishes, and so on.
Interrupts

- An interrupt is a signal that says that normal processing might need to be stopped because we have to do something.
- One analogy is answering a phone call — we're doing something, the phone rings, we answer it, hang up, and then go back to what we were doing. The phone ring is the interrupt and the answering and talking is interrupt handling.
- In a computer, an interrupt generally comes from I/O finishing.
  - To do I/O, we tell the device to start doing something. (Read a character from the keyboard)
  - Rather than wait for it to be done, we go off and do something else for a while.
    - Maybe execute someone else's program.
  - When the I/O action finishes, we get an interrupt (someone typed in a character)
    - The interrupt handler figures out which I/O action finished and lets the original program continue on.
      (Your program was executing a read instruction; it finally finishes.)
- Other interrupts include the wall clock ticking (timer interrupt), various hardware problems (someone turned off the power).

[Section below added Thu 3/28]

- Historically, I/O status was checked using polling, where we repeatedly ask a device if it's finished yet. ("Are we there yet? Are we there yet? …"). Polling is much less efficient than modern interrupt-driven I/O unless we know that the time we'll have to wait is less than the time it takes to handle an interrupt (notice the signal, pause the current program, change to the interrupt handler, run its code, restore the paused program, and go back).
- To support interrupts, the control unit has to be modified. An interrupt signal line has to be added, and the instruction cycle has to be modified:
  - Between the last step of the current instruction cycle and the Fetch instruction phase of the next cycle, we add a check for an interrupt signal.
  - If no interrupt has been signaled, we just continue with the instruction cycle as usual.
- Not all operations can be interrupted. For example, if the list of active programs kept by the OS is implemented using linked lists, then we must not stop in the middle of a linking/unlinking operation, else the list can be scrambled.
- Can an interrupt handler be interrupted? In general, people assign priority levels to interrupts so that a higher priority interrupt can interrupt an interrupt handler for a lower-priority interrupt. (For example, a power failure interrupt is more important than the regular clock timer ticking interrupt.) This also takes care the situation when we're doing something that should never be interrupted: Just set the priority threshold for accepting an interrupt to be higher than any actual interrupt.
- Note we don't want just any program performing dangerous operations like modifying the interrupt threshold or starting or stopping an I/O operation. To handle this, some machine instructions can only be run when in a privileged mode of the computer. In general, the OS runs in privileged mode, which is why it gets to perform I/O directly instead of through a library routine like scanf or printf in C.
• There's a register commonly called the **Processor Status Register (PSR)** that contains a bit that indicates whether the program is running in privileged or user mode. When an interrupt occurs, the hardware automatically turns that bit on so that the interrupt handling code can run in privileged mode. When it finishes, interrupt handling code can turn the bit off if it's returning to user-mode code. (It could have been OS code that was interrupted, so we don't always just turn off the privileged mode bit.)

• For a user to ask the OS to run privileged code (like doing some I/O), computers generically have some sort of "TRAP" instruction that is like an interrupt in that it causes a change to privileged mode and runs OS-maintained trap-handling code. Normal interrupts are asynchronous (can happen at any time), so TRAPs are often called "synchronous" or "soft" interrupts. The MIPS architecture has a **SYSCALL** instruction for to let users request OS services. (There's also a different kind of instruction for code debugging.)

### Study Questions

1. Why does instruction pipelining increase throughput but not decrease latency?
2. What are the following: Pipeline stall, bubble, hazard, branch hazard, data hazard.
3. What is the read-after-write problem? The write-after-read problem? Give an example of each.
4. Consider the code below. Where and can it cause pipeline stalls? How might the code be modified so that pipelining is more effective? (Feel free to use other temporary registers.)

   ```
   add  $t0,$s0,$s1
   sw   $t0,X
   add  $t0,$s2,$s3
   sw   $t0,Y
   lw   $s4,X
   lw   $s5,Y
   ```

5. What is a (hardware) interrupt? How is the control unit modified to handle interrupts?
6. One thing done before running interrupt-handling code is to save the user's register values, since the interrupt-handling code might change them. Is it okay to allow a higher-priority interrupt interrupt this register saving (and symmetric storing) action?
7. What is privileged vs user mode and why is it used? When is it set/reset? Where is it stored?

### Selected Answers

1. Pipelining increases throughput because it allows multiple instructions to execute simultaneously instead of sequentially. It doesn't decrease latency because each instruction's execution is basically unchanged.
2. See lecture body.
3. See lecture body.
4. First, the code uses $t0 for two different problems, setting $X = $s0 + $s1 and $Y = $s2 + $s3. Using $t0 in add $t0,$s2,$s3 introduces a dependency (write-after-write) that isn't really necessary. We can use a different temporary register $t1 for the second addition. Second, the code stores a value from $t0 to X and then loads $s4 from X; it would be much faster to simply copy the value from $t0 to $s4. Similarly, we can copy $t1 to $s5. Making these changes gives us
Another change we can make is to move the setting of \$s4 upward so that it might overlap with the `sw` of \$t0. This gives us:

```assembly
add \$t0,\$s0,\$s1
sw \$t0, X
add \$t1,\$s2,\$s3
sw \$t1, Y
add \$s4,\$t0, \$zero
add \$s5,\$t1, \$zero
```