10/11: Lecture Topics

- Slides on starting a program from last time
- Where we are, where we're going
- RISC vs. CISC reprise
- Execution cycle
- Pipelining
- Hazards

Where we've been:

- Architecture vs. implementation
- MIPS assembly
- Addressing modes, Instruction encoding
- Assembly, linking, and loading
- Chapters 1 & 3

Where we're going

- Make it fast
  - pipelining (chapter 6)
  - caching (chapter 7)
- Make it useful
  - Input/Output (chapter 8)
- Current research, Future trends
- Midterm October 27th

Where we're not going

- Performance: chapter 2
- Bit twiddling: chapter 4
- Datapath and control: chapter 5
  - important, but depends on a background in digital logic
- Multiprocessors: chapter 9

RISC vs. CISC

- *Reduced Instruction Set Computer*
  - MIPS: about 100 instructions
  - Basic idea: compose simple instructions to get complex results
- *Complex Instruction Set Computer*
  - VAX: about 325 instructions
  - Basic idea: give programmers powerful instructions; fewer instructions to complete the work

The VAX

- Digital Equipment Corp, 1977
- Advances in microcode technology made complex instructions possible
- Memory was expensive
  - Small program = good
- Compilers had a long way to go
  - Ease of translation from high-level language to assembly = good

VAX Instructions

- Queue manipulation instructions:
  - INSQUE: insert into queue
- Stack manipulation instructions:
  - POPR, PUSHR: pop, push registers
- Procedure call instructions
- Binary-encoded decimal instructions
  - ADDP, SUBP, MULP, DIVP
  - CVTPL, CVTLP (conversion)

The RISC Backlash

- Complex instructions:
  - Take longer to execute
  - Take more hardware to implement
- Idea: compose simple, fast instructions
  - Less hardware is required
  - Execution speed may actually *increase*
- PUSHR vs. sw + sw + sw

How many instructions?

- How many instructions do you *really* need?
- Potentially only one: subtract and branch if negative (sbn)
- See p. 206 of your book

Execution Cycle

- Five steps to executing an instruction:
  1. Fetch
     - Get the next instruction to execute from memory onto the chip
  2. Decode
     - Figure out what the instruction says to do
     - Get values from registers
  3. Execute
     - Do what the instruction says; for example,
       - On a memory reference, add up base and offset
       - On an arithmetic instruction, do the math

More Execution Cycle

4. Memory Access
   - If it's a load or store, access memory
   - If it's a branch, replace the PC with the destination address
   - Otherwise do nothing
5. Write back
   - Place the result of the operation in the appropriate register

Laundry

- Four steps to doing the laundry:
  - Wash, Dry, Fold, Put Away
- If each step = 30 min., 4 loads = _____

Pipelined Laundry

- Allow laundry *stages* to operate concurrently
- Now four loads takes _____

Latency vs. Throughput

- The *latency* of a load of laundry is 2 hours
  - Does not change with pipelining
- The *throughput* of the laundry system is
  - 1 loads/2 hours = .5 LPH without pipelining
  - 1 load/.5 hours = 2 LPH with pipelining
- The speedup is 4, the same as the number of stages (when stages are balanced)

Balancing the Stages

- What if the dryer takes an hour, while the other stages take 30 minutes?

- 1 load/1 hour = 1 LPH speedup = 2

Pipelining instructions

- We can overlap the five stages of the execution cycle
- Five different instructions can be executing simultaneously, if:
  - they are all in different stages
  - the stages are nearly balanced
  - nothing else goes wrong

What could go wrong?

- Structural hazards
  - Two instructions are incompatible
- Control hazards
  - We need to make a decision, but not all of the information is available
- Data hazards
  - We need to use the result of a previous computation for this computation

Structural Hazards

- Suppose a `lw` instruction is in stage four (memory access)
- Meanwhile, an `add` instruction is in stage one (instruction fetch)
- Both of these actions require access to memory; they could collide
- In practice, they don't, because of the design of the caching system

Control Hazards

- Suppose we have a `slt/bne` combination
- `slt` stores its result to a register in stage five
- `bne` needs that result at the beginning of stage four; it can't proceed
- Can *stall*, waiting for the result
- Can do *speculative execution*, and guess the result

Data Hazards

- Suppose we want to execute:

```
add $t2, $t0, $t1
add $t4, $t2, $t3
```

- The first addition doesn't store its result until the end of stage five
- The second addition wants to load its operands in stage two

Handling Data Hazards

- Again, you can *stall*
- You can use *data forwarding*
  - pass the data directly from stage 3 of the first add to stage 3 of the second add
- Sometimes, you can do *out-of-order execution*
  - reorder the instructions such that:
    - maintain correctness
    - avoid or reduce stalls