Multi-cycle Implementation

Issues With the Single Cycle Implementation

- All instructions take the same time (CPI = 1), but some are actually *shorter* than others...
  - ADD uses Instruction Memory, Register File, ALU, Register File
  - LW uses Instruction Memory, Register File, ALU, Data Memory, Register file again...
- The cycle time of the machine has to be the cycle time of the "longest" instruction
- We are violating an important principle: Make the common case fast.

Thought Experiment

- Suppose we could design a machine whose cycle time varied, so it was just long enough for each kind of instruction.
- Suppose the following times are realistic (in nanoseconds):
  - Memory access: 2
  - Register Read/Write: 1
  - ALU: 2
- Fill in the table on the next page

Thought Experiment 2

Instruction type	IMEM	Reg Read	ALU	DMEM	Reg Write	Total
Load						
Store						
R-type						
Branch						

Thought Experiment 3

- A given benchmark (say GCC) has this mix: 20% loads, 10% stores, 50% R-format, 20% branches
  - What's the single-cycle time?
  - What's the vari-cycle time?
  - What's the speedup?
- Suppose we add floating point instructions, and it takes 8ns to do an FP add and 16 ns to do a FP mult.
  - New cycle time of single cycle machine?
  - Another mix: 25% loads, 15% stores, 30% R-format, 10% branches, 10% FP add, 10% FP mult

Making it Better

- Of course, it's impractical to build a vari-cycle machine.
- What to do?
  - *Multiple cycle implementation* (section 5.4): Approximate the effect of a variable clock, by letting instructions take different numbers of cycles to complete.
  - *Pipelining*: Observation: We're underutilizing functional units (eg. the ALU is idle while we're accessing memory). Find a way to work on multiple instructions at the same time.
- CISCS pretty much require a multi-cycle implementation. Why?
- RISCS are amenable to pipelining...