Pipelining

Drawbacks of the Single Cycle Imp

- A single cycle machine has disadvantages such as: All instructions take the same time (CPI = 1), but some instructions are shorter than others:
  - ADD uses Instruction Memory, Register File, ALU, Register File
  - LW uses Instruction Memory, Register File, ALU, Data Memory, and Register file again...
- The cycle time of the machine is the time needed to execute the "longest" instruction.
- Note also that we're underutilizing functional units (the instruction memory, register file, and ALU sit idle while data memory is being read/written)
- We are violating our principle -- make the common case fast -- we're making the common case take as long as the most uncommon case...

Thought experiment

- Suppose we could design a machine whose cycle time varied, so that it was just long enough for each kind of instruction.
- What would our performance improvement be over the single cycle machine, given these numbers:

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>IMEM</th>
<th>Reg Read</th>
<th>ALU</th>
<th>DMEM</th>
<th>Reg Write</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Store</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>R-type</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>-</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Branch</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>-</td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

Thought Experiment 2

- GCC instruction mix: 22% loads, 11% stores, 49% R-format, 18% branches
  - Single-cycle cycle time = ??
  - Vari-cycle cycle time = ??
  - What's the speedup?
- Now suppose we add floating point, and that that our FP ALU takes 8 ns for add/sub and 16 ns for mult/div.
  - What would be the new cycle time of the single cycle machine?
  - How much faster would the variable clock machine be, given this mix: 25% loads, 15% stores, 30% R-format, 10% branches, 10% FP mult, 10 % FP add
  - Vari-cycle time = .26*40 + .14*35 + .31*30 + .10*25 + .09*80 + .10*40 = 38ns
  - What's the speedup?
Improving Performance

- Of course, it's really impractical to build a variable clock machine.
- As the ISA gets more complex, the single cycle shortcomings become more serious, so what do we do? Two approaches:
  - *Multiple cycle machine* (section 5.4): This is a way to approximate the effect of a variable clock, by letting instructions take different numbers of cycles to complete. For instance, loads might take 5 cycles because they use all 5 functional units, but adds might only take 4 cycles...
  - *Pipelining*: Observe that we're underutilizing our functional units -- e.g. the ALU sits idle while we access data memory. Find a way to work on several instructions at the same time.
  - CISC ISAs pretty much require a multi-cycle implementation. Why?
  - RISC ISAs are amenable to pipelining.

Pipelining Defined

- Basic metaphor is the assembly line:
  - Split a job $A$ into $n$ sequential subjobs ($A_1, A_2, ..., A_n$) with each $A_i$ taking approximately the same time.
  - Each subjob is processed by a different substation (resource), or equivalently, passes through a series of *stages*.
  - When subjob $A_1$ moves from stage 1 to stage 2, subjob $A_2$ enters stage 1, and so on.
- Laundry example:
  - Suppose doing a load of laundry, from beginning to end, takes 1.5 hours. How long does it take to do 3 loads of laundry?
  - If we split this job into 3 subjobs: washing (30 minutes), drying (30 minutes), folding+ironing (30 minutes).
  - With a pipeline, how long does each load of laundry take?
  - How long do 3 loads of laundry take? 10 loads? N loads?

Pipeline Performance

- The execution time for a single job can be longer, since each substage takes the same amount of time -- the time for the longest of any stages. Eg. it might not take a full 30 minutes to fold and iron clothes...
- However, throughput is enhanced because a new job can start at every stage time (and one job completes at every stage time).
- Pipelining enhances performance by increasing throughput of jobs, not by decreasing the amount of time each job takes.
- In the best case, throughput increases by a factor of $n$, if there are $n$ stages. This is optimistic, because:
  - Execution time of a job by itself could be less than $n$ stage times
  - We are assuming the pipeline can be kept full all of the time.

Pipeline Implementation

- The trick is to break the one long cycle into a sequence of smaller, hopefully equally-sized tasks. Traditionally, the cycle is broken into these 5 subjobs (pipe stages):
  - **IF**: *Instruction Fetch* -- get the next instruction
  - **ID**: *Instruction Decode* -- decode the instruction and read the registers
  - **EX**: *ALU Execution* -- utilize the ALU
  - **MEM**: *Memory Access* -- read/write memory
  - **WB**: *Write Back* -- write results to the register file
- On each machine cycle, each pipe stage does its small piece of work on the instruction that is currently inside of it. Each instruction now takes 5 cycles to complete.
Why it’s not (quite) so simple:

- We need to remember information about each instruction in the pipeline. This information has to flow from stage to stage. We accomplish this with pipeline registers.
- We need to deal with dependencies between instructions:
  - Data dependencies
  - Control dependencies
- We’ll ignore dependencies between instructions for now.
Example

- Trace the execution of this 3 instruction sequence:

\[
\begin{align*}
\text{lw} & \quad 10, 16(1) \\
\text{sub} & \quad 11, 2, 3 \\
\text{sw} & \quad 12, 16(4)
\end{align*}
\]

Instruction Fetch

- We'll next describe the operation of the pipeline in some detail, using pseudocode.
- Instruction Fetch and Decode is the same for all instructions:
- **Instruction Fetch Stage.** The IF/ID register needs to hold 2 pieces of information:

\[
\begin{align*}
\text{IF/ID.IR} & \leftarrow \text{IMemory[PC]} \\
\text{if (EX/MEM.ALUResult == 0)} & \\
& \quad \text{PC} \leftarrow \text{EX/MEM.TargetPC} \\
\text{else} & \\
& \quad \text{PC} \leftarrow \text{PC} + 4 \\
\text{IF/ID.nPC} & \leftarrow \text{PC}
\end{align*}
\]

Instruction Decode

- **Instruction Decode Stage.** The ID/EX register needs to hold 6 pieces of information. Let A be input 1 of the ALU and B be input 2.

\[
\begin{align*}
\text{ID/EX.nPC} & \leftarrow \text{IF/ID.nPC} \\
\text{ID/EX.A} & \leftarrow \text{Reg[IF/ID.IR[25:21]] (i.e. read rs)} \\
\text{ID/EX.B} & \leftarrow \text{Reg[IF/ID.IR[20:16]] (i.e. read rt)} \\
\text{ID/EX.Imm} & \leftarrow \text{sign-extend[IF/ID.IR[15:0}}} \\
\text{ID/EX.rd} & \leftarrow \text{IF/ID.IR[15:11]} \\
\text{ID/EX.rt} & \leftarrow \text{IF/ID.IR[20:16]}
\end{align*}
\]

- Note that we start computing immediate, even though we might not need it, and it might be "garbage"

ALU Execution

- **ALU Execution Stage** either computes the memory address for load/stores, the value for arithmetic instructions, or whether a branch is being taken.
- The EX/MEM register needs to hold 4 pieces of information:

\[
\begin{align*}
\text{EX/MEM.B} & \leftarrow \text{ID/EX.B} \\
\text{EX/MEM.WriteReg} & \leftarrow \text{ID/EX.rd (for R-type)} \\
& \quad \text{or ID/EX.rt (for Load)} \\
\text{EX/MEM.TargetPC} & \leftarrow (4*\text{ID/EX.Imm}) + \text{ID/EX.nPC} \\
\text{EX/MEM.ALUResult} & \leftarrow "\text{ALU Result}"
\end{align*}
\]

ALU Result is either

\[
\begin{align*}
\text{ID/EX.A + ID/EX.Imm} & \quad (\text{for lw, sw}) \\
\text{ID/EX.A op ID/EX.B} & \quad (\text{for R-type}) \\
\text{ID/EX.A - ID/EX.B} & \quad (\text{for beq})
\end{align*}
\]
Memory Access and WriteBack

- **Memory Access Stage.** The MEM/WB register needs to hold 3 pieces of information:
  - MEM/WB.MemData <- DMemory[EX/MEM.ALUResult]
  - MEM/WB.ALUResult <- EX/MEM.ALUResult
  - MEM/WB.WriteReg <- EX/MEM.WriteReg
  - DMemory[ALUResult] <- EX/MEM.B (for stores)

- **Write Back Stage.** All we need to do here is write back the results (either from an ALU op or memory load) to the destination register:
  - if ins was Load
    - Reg[MEM/WB.WriteReg] <- MEM/WB.MemData
  - else if ins was R-type
    - Reg[MEM/WB.WriteReg] <- MEM/WB.ALUResult

Summary of Simple Pipeline

- In stages 1 and 2 (IF and ID) the information to be kept is the same for all instructions.
- But the pipeline registers must accommodate the “maximum” amount of information that we might need to maintain.
- For example, if we have a store, the contents of rs must be kept in EX/MEM, which is not necessary in the case of loads or arithmetic instructions.
- Instructions must pass through all stages even if there is nothing to be done in that stage.
- The pipeline takes 4 cycles before it is operating at full efficiency.

Control

- The basic idea with pipeline control is to pass along control information from stage to stage:
  - control unit
    - IF/ID: RegWrite, MemToReg
    - ID/EX: MemRead, MemWrite, RegWrite, RegDest
    - EX/MEM: ALUSrc, ALUops
    - MEM/WB: MemToReg, RegWrite, RegDst

Control in the Ideal Case

- Control signals must be split among the 5 stages:
  - IF: nothing special to control (always read instruction, increment PC)
  - ID: nothing special, same for all instructions
  - EX: control signals for ALUSrc and ALUop are needed.
  - MEM: Controls for MemRead, MemWrite, and Branch are needed here
  - WB: Controls for what to write (MemToReg) and RegWrite and RegDst needed here