Notes and Updates

- Chapter 5 - Single and Multicycle Procs
- New Homework up
  - Hw and quiz return at the end of class
- Homework due NO LATER than 2:05pm
  - No handing it in at the end of class
- QUESTIONS!

- KQS cards
- How fast is class?
  - 1 too slow  3 OK  5 too fast

Chapter 5:
The Processor: Datapath and Control
The Single Cycle Processor
The Multicycle Processor
The Performance Big Picture

- Execution Time = Insts * CPI * Cycle Time
- Processor design (datapath and control) will determine:
  - Clock cycle time
  - Clock cycles per instruction

Starting today:
- Single cycle processor:
  - Advantage: CPI = 1
  - Disadvantage: long cycle time

What parts of MIPS?

- We won't implement all of MIPS
  - Memory instructions
  - Arithmetic/Logical (and just a subset of these, but you should be able to figure out how to add many of them)
  - BEQ and J (last)
- Basic load/store architecture with these steps:
  - Read PC and Fetch Inst
  - Read Registers
  - Do Math
  - Write memory/registers
  - Repeat
- Graphically?
Instruction Fetch Unit

Updating the PC for next instruction
- Sequential Code:
  - Branch and Jump:
    - We'll save branches for later, after adds, subs
The MIPS core subset

- **R-type**
  - add rd, rs, rt
  - sub, and, or, slt

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

   1. Read registers rs and rt
   2. Feed them to ALU
   3. Update register file

- **LOAD and STORE**
  - lw rt, rs, imm
  - sw rt, rs, imm

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

   1. Read register rs (and rt for store)
   2. Feed rs and imm to ALU
   3. Move data between mem and reg

- **BRANCH:**
  - beq rs, rt, imm

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

   1. Read registers rs and rt
   2. Feed to ALU to compare
   3. Add PC to disp; update PC

---

Register Transfer Language (RTL)

- Is a mechanism for describing the movement of data between storage elements
  - Gives us a precise way to describe various actions of our instructions
    - May be more than 1 RTL statement per inst
      - PC <= PC + 4
      - R[rd] <= R[rs] + R[rt]
Post Fetch Datapath for Reg-Reg Operations

- \( R[rd] \leftarrow R[rs] \ op \ R[rt] \)  
  - Example: \( add \ rd, rs, rt \)
  - \( R[a], R[b], \) and \( R[w] \) come from \( rs, rt, \) and \( rd \) fields
  - ALU operation signal depends on \( op \) and \( funct \)

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

```
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
ALU operation
ALU
Zero
ALU result
RegWrite
```

Post Fetch Datapath for Store Operations

- \( \text{Mem}[R[rs] + \text{SignExt}[\text{imm}16]] \leftarrow R[rt] \)  
  - Example: \( \text{sw} \ rt, rs, \text{imm}16 \)

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

```
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
RegWrite
Read data 1
Read data 2
ALU operation
ALU
Zero
ALU result
Sign extend
RegWrite
```
Putting together Store DP and RR DP

Post Fetch Datapath for Load Operations

\[
R[rt] \leftarrow \text{Mem}[R[rs]] + \text{SignExt}[\text{imm16}]
\]

Example: \text{lw } rt, rs, \text{imm16}
Putting together Load/Store DP and Reg-Reg DP

Datapath for Branch Operations

\[
\text{beq } rs, rt, \text{ imm}16 \quad \text{We need to compare Rs and Rt}
\]

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

[Diagram showing the datapath for branch operations]
Computing the Next Address

- PC is a 32-bit byte address into the instruction memory
  - Sequential operation: $PC_{31:0} = PC_{31:0} + 4$
  - Branch: $PC_{31:0} = PC_{31:0} + 4 + \text{SignExt}[\text{Imm16}] \times 4$

- We don’t need the 2 least-significant bits because:

---

Detour:

* Multiply -- That’s expensive!*

- Multiply the immediate by 4! Let’s try some possible values

0000 0001
0000 0010
0000 0011
0000 0100
0000 0100
1111 1111
Datapath for Branch Operations

```
beq rs, rt, imm16
We need to compare Rs and Rt
```

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

```
alu operation
alu zero
```

To branch control logic

---

All together: the single cycle datapath
R-Format/ Lw/ Sw/ BEQ

The R-Format (e.g. add) Datapath
The Load Datapath

The Store Datapath
Key Points

- CPU is just a collection of state and combinational logic
- We just designed a very rich processor, at least in terms of functionality
  - Know and understand
    - Basic flow
    - Control lines
    - Muxes - where and why needed
- Execution time = Insts * CPI * Cycle Time
  - where does the single-cycle machine fit in?
Adding Control Signals

DETOUR: Single Cycle Datapath

Warning! Text is inconsistent. MUX control signals sometimes have “1” on top, sometimes “0”. On exercises&tests, look carefully!
Control for instructions

Generating the control signals

PLA for control signals
ALU control bits: both opcode and funct

- Suppose ALU has 6 functions with control bits

<table>
<thead>
<tr>
<th>ALU control input</th>
<th>Function</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>And</td>
<td>and</td>
</tr>
<tr>
<td>0001</td>
<td>Or</td>
<td>or</td>
</tr>
<tr>
<td>0010</td>
<td>Add</td>
<td>add, lw, sw</td>
</tr>
<tr>
<td>0110</td>
<td>Subtract</td>
<td>sub, beq</td>
</tr>
<tr>
<td>0111</td>
<td>Slt</td>
<td>slt</td>
</tr>
</tbody>
</table>

- We’ll generate ALU control from opcode (bits 31-26) and funct field (bits 5-0) of instruction
- ALU doesn't need to know all opcodes -- we CHOOSE to summarize opcode with ALUOp (2 bits):
  - 00 - lw, sw  01 - beq  10 - R-format

Generating ALU control

<table>
<thead>
<tr>
<th>Instruction opcode</th>
<th>ALUOp</th>
<th>Instruction operation</th>
<th>Function code</th>
<th>Desired ALU action</th>
<th>ALU control input</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>00</td>
<td>load word</td>
<td>xxxxxxx</td>
<td>add</td>
<td>010</td>
</tr>
<tr>
<td>sw</td>
<td>00</td>
<td>store word</td>
<td>xxxxxxx</td>
<td>add</td>
<td>010</td>
</tr>
<tr>
<td>beq</td>
<td>01</td>
<td>branch eq</td>
<td>xxxxxxx</td>
<td>subtract</td>
<td>110</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>add</td>
<td>100000</td>
<td>add</td>
<td>010</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>subtract</td>
<td>100010</td>
<td>subtract</td>
<td>110</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>AND</td>
<td>100100</td>
<td>and</td>
<td>000</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>OR</td>
<td>100101</td>
<td>or</td>
<td>001</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>slt</td>
<td>101010</td>
<td>slt</td>
<td>111</td>
</tr>
</tbody>
</table>
Finally, Adding the jump instruction

- How do we get the new PC from a jump format instruction?
  - Top 4 bits - from PC (steal them, not add them)
  - Next 26 bits, from “immediate field” of J type
  - Last 2 bits: 00!

- When do we jump?
How could this change?

**KEY: CRITICAL PATH**

<table>
<thead>
<tr>
<th></th>
<th>I cache</th>
<th>Decode, R-Read</th>
<th>ALU</th>
<th>PC update</th>
<th>D cache</th>
<th>R-Write</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>-</td>
<td>.8</td>
<td>3.7</td>
</tr>
<tr>
<td>Load</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>1</td>
<td>.8</td>
<td>4.7</td>
</tr>
<tr>
<td>Store</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>3.9</td>
</tr>
<tr>
<td>beq</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>.1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Single-Cycle CPU Control Summary

- Easy, particularly the control
- Which instruction takes the longest? By how much? Why is that a problem?
- Execution time = insts * cpi * cycle time
- Real machines have much *more* variable instruction latencies than this small subset.
Why use multicycle design?

- Problem: In single-cycle design, cycle time must be long enough for longest instruction
- Solution: break execution into smaller tasks
  - each task takes a cycle;
  - different instructions require different numbers of cycles
- Another advantage: May need fewer logic blocks

Multicycle implementation

Goal: balance amount of work done each cycle.

<table>
<thead>
<tr>
<th></th>
<th>I cache</th>
<th>Decode, R-Read</th>
<th>ALU</th>
<th>PC update</th>
<th>D cache</th>
<th>R-Write</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>-</td>
<td>.8</td>
<td>3.7</td>
</tr>
<tr>
<td>Load</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>1</td>
<td>.8</td>
<td>4.7</td>
</tr>
<tr>
<td>Store</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>3.9</td>
</tr>
<tr>
<td>beq</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>.1</td>
<td>-</td>
<td>-</td>
<td>3.0</td>
</tr>
</tbody>
</table>

- How many cycles does each instruction type take?
  - Loads
  - Stores
  - RType
  - beq
Will multicycle design be faster?

<table>
<thead>
<tr>
<th></th>
<th>I cache</th>
<th>Decode, R-read</th>
<th>ALU</th>
<th>PC update</th>
<th>D cache</th>
<th>R-write</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>-</td>
<td>.8</td>
<td>3.7</td>
</tr>
<tr>
<td>Load</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>1</td>
<td>.8</td>
<td>4.7</td>
</tr>
<tr>
<td>Store</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>3.9</td>
</tr>
<tr>
<td>beq</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>.1</td>
<td>-</td>
<td>-</td>
<td>3.0</td>
</tr>
</tbody>
</table>

Single cycle design:
- Clock cycle time =
- time/inst =

Multicycle design:
- Clock cycle time =
- time/inst =

It depends on the program!

<table>
<thead>
<tr>
<th></th>
<th>Cycles needed</th>
<th>Instruction frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>4</td>
<td>60%</td>
</tr>
<tr>
<td>Load</td>
<td>5</td>
<td>20%</td>
</tr>
<tr>
<td>Store</td>
<td>4</td>
<td>10%</td>
</tr>
<tr>
<td>beq</td>
<td>3</td>
<td>10%</td>
</tr>
</tbody>
</table>

What is CPI assuming this instruction mix???

Single cycle design:
- Clock cycle time = 4.7 ns
- time/inst = 1 cycle/inst * 4.7 ns/cycle = 4.7 ns/inst

Multicycle design:
- Clock cycle time = 1 ns
- time/inst = CPI * 1 ns/cycle

=
The Five Cycles

- Five execution steps (some instructions use fewer)
  - **IF**: Instruction Fetch
  - **ID**: Instruction Decode (& register fetch & add PC+immed)
  - **EX**: Execute
  - **Mem**: Memory access
  - **WB**: Write-Back into registers

<table>
<thead>
<tr>
<th></th>
<th>I cache</th>
<th>Decode</th>
<th>ALU</th>
<th>PC update</th>
<th>D cache</th>
<th>R-W in</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>-</td>
<td>.8</td>
<td>3.7</td>
</tr>
<tr>
<td>Load</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>1</td>
<td>.8</td>
<td>4.7</td>
</tr>
<tr>
<td>Store</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>3.9</td>
</tr>
<tr>
<td>beq</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>.1</td>
<td>-</td>
<td>-</td>
<td>3.0</td>
</tr>
</tbody>
</table>

Partitioning the Single-Cycle Design