Notes and Updates

- Lab due Monday!
  - Out of town this weekend
- Chapter 5 - Single and Multicycle Procs
- Homework due NO LATER than 2:05pm
  - No handing it in at the end of class
- KQS cards from last time
- Recommended, print out slides, drawing
- Office hours: Tues 3:30-5, Wed 12-1:30
- QUESTIONS!

Most of the Single Cycle Datapath

![Datapath Diagram]
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 =

## What if something changes?

<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>

### 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 = 4.7ns
\[
\text{time/inst} = \frac{1 \text{ cycle}}{\text{inst}} \times 4.7 \text{ ns/cycle} = 4.7 \text{ ns/inst}
\]

Multicycle design:
Clock cycle time = 1 ns
\[
\text{time/inst} = \text{CPI} \times 1 \text{ ns/cycle}
\]

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, 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>
Partitioning the Single-Cycle Design

Adding State Elements

Since we reuse logic (e.g. ALU), we need to store results between states

Need extra registers when:
- signal is computed in one clock cycle and used in another, AND
- the inputs to the combinational circuit can change before the signal is written into a state element.
  - We only require them to be "held" for 1 cycle
Where to add registers (more or less)

Complete Multicycle Datapath
Summary of execution steps

<table>
<thead>
<tr>
<th>Step</th>
<th>R-type</th>
<th>Memory</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td>IR = Mem[PC]</td>
<td>PC = PC + 4</td>
<td></td>
</tr>
<tr>
<td>Instruction Decode/</td>
<td>A = Reg[IR[23-21]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>register fetch</td>
<td>B = Reg[IR[20-16]]</td>
<td>ALUout = PC + (sign-extend(IR[15-0]) &lt;&lt; 2)</td>
<td></td>
</tr>
<tr>
<td>Execution, address</td>
<td>ALUout = A op B</td>
<td>ALUout = A + sign-extend(IR[15-0])</td>
<td>If (A==B) then PC=ALUout</td>
</tr>
<tr>
<td>computation, branch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>completion</td>
<td>Reg[IR[15-11]] = ALUout</td>
<td>memory-data = Mem(ALUout) or Mem[ALUout] = B</td>
<td></td>
</tr>
<tr>
<td>Memory access or R-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>type completion</td>
<td>Reg[IR[20-16]] = ALUout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write-back</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This is Register Transfer Language (RTL)

"High level" description of changes to state elements

We'll go through these in exacting detail
And translate them to "low level" control signal settings
Modern design tools do this automatically

Control Lines (page 384):

- 1-bit
  - RegDst, RegWrite, MemRead, MemWrite
  - ALUSrcA: top ALU operand from reg or PC?
  - MemtoReg: data to write register from ALUout or MDR(memory)?
  - IorD: Runs to Memory to indicate reading/writing Instruction or Data
  - IRWrite: Buffer register for Instr should be written
  - PCWrite: PC should written absolutely
  - PCWriteCond: PC should be written, maybe
Control Lines (page 324):

- **2-bit**
  - ALU Op: 00 add, 01 sub, 10 funct based
  - ALUSrcB:
    | What selected | What type of op uses? |
    |---------------|-----------------------|
    | 00            |                       |
    | 01            |                       |
    | 10            |                       |
    | 10            |                       |
  - PCSrc

Cycle 1: Instruction Fetch

Datapath RTL:
Control: IorD MemRead MemWr IRwrite ALUsrcA ALUsrcB
Control for IF cycle

- We’ll design a Finite State Machine to detail our control logic.
  - Basically a sequential logic function OR
  - A program

\[
\begin{align*}
\text{MemRead} \\
\text{ALUsrcA} &= 0 \\
\text{IorD} &= 0 \\
\text{IRwrite} \\
\text{ALUsrcB} &= 01 \\
\text{ALUop} &= 00
\end{align*}
\]

Detour: Finite State Machines (Programs?)

- Set of inputs and outputs
- Next-state function that maps current state and (possibly) inputs to a next state and (possibly) outputs

\[
\begin{array}{c|ccc}
\text{OPEN} & \text{NEITHER} & \text{FRONT} & \text{REAR} \\
\text{CLOSED} & \text{C} & \text{O} & \text{O} \\
\end{array}
\]

Problem: Automatic door sensor

- Front pad
- Rear pad
Outline of FSM for Multicycle CPU

Cycle 1

MemRead
ALUsrcA = 0
IorD = 0
IRwrite
ALUsrcB = 01
ALUop = 00
PCwrite
PCsource = 00

Cycle 2

Any inst

Cycle 3

Cycle 2: Instruction Decode
(and register fetch)

A = Reg[IR[25-21]]
B = Reg[IR[20-16]]
ALUout = PC + (sign-extend (IR[15-0]) << 2)

We compute target address even though we don’t know if it will be used
- Operation may not be branch
- Even if it is, branch may not be taken

Why NOW?

Why for ALL?
Cycle 2: Instruction Decode cycle

A = Register[IR[25-21]]
B = Register[IR[20-16]]
ALUout = PC + (sign-extend (IR[15-0]) \ll 2)

Control for first two cycles

Instruction Fetch, state 0

Start
MemRead
ALUsrcA = 0
IorD = 0
IRWrite
ALUsrcB = 01
ALUOp = 00
PCWrite
PCSource = 00

After cycle 2, we can treat different instructions separately

Instruction Fetch
Opcode = LW or SW
Opcode = R-type
Opcode = BEQ
Opcode = JMP

Memory Inst
R-type Inst
Branch Inst
Jump Inst
FSM
FSM
FSM
FSM
Cycle 3 for beq: Execute

- In cycle 1, PC was incremented by 4
- In cycle 2, ALUout was set to branch target
- This cycle, we conditionally reset PC: \( \text{if} \ (A==B) \ \text{PC}=\text{ALUout} \)

FSM state for cycle 3 of beq

from state 1

To state 0
**R-type instruction review**

- **Cycle 1 (Fetch)**
  \[ PC = PC + 4 \]
  \[ IR = MEM[PC] \]

- **Cycle 2 (Decode)**
  \[ ALUout = PC + (sign-extend (IR[15-0]) \ll 2) \]
  \[ A = Reg[IR[25-21]] \]
  \[ B = Reg[IR[20-16]] \]

- **Cycle 3 (EXecute)**
  \[ ALUout = A \text{ op } B \]

- **Cycle 4 (WriteBack)**
  \[ Reg[IR[15-11]] = ALUout \]

Then R-type instruction is finished.

---

**R-type EXecution Then WriteBack**

**Cycle 3**: \[ ALUout = A \text{ op } B \]

**Cycle 4**: \[ Reg[IR[15-11]] = ALUout \]
FSM states for R-type Instructions

from state 1

Execution

ALUsrcA = 1
ALUsrcB = 00
ALUop = 10

WriteBack

To state 0

Load and Store

- **EXecute (cycle 3): compute address**
  \[ \text{ALUout} = A + \text{sign-extend} (\text{IR}[15-0]) \]
- **Mem (cycle 4): access memory**

- **WB (cycle 5, only for load): write register**
  \[ \text{Reg[IR}[20-16]] = \text{MDR} \]
Cycle 3 for lw and sw: Address Computation

\[ \text{ALUout} = A + \text{sign-extend}(\text{IR}[15-0]) \]

1/5

Cycle 4 for Store: Memory Access

\[ \text{Memory}[\text{ALUout}] = B \]

2/5
Cycle 4 for Load: Memory Access

Memory Data Register = Memory[ALUout]

Cycle 5 for load: WriteBack

Reg[IR[20-16]] = memory-data
from state 1

Memory
Instruction
States

MemRead
IorD = 1

MemWrite
IorD = 1

write-back

5/5!!!!

Cycle 3 for Jump

PC = PC[31-28] | (IR[25-0] <<2)
**Cycle 3 JMP FSM state**

- From state 1
- PCWrite
- PCSource=10

---

**RECAP:**

**Control of Multicycle Implementation**
- We’ve been building up a Finite State Machine (FSM)
- FSM shows a succession of states, transitions between states (based on inputs), and outputs for each state.
- First two states are the same for every instruction, later states depend on opcode
The Complete FSM

page 339