Summary: Single-cycle Processor

- 5 steps to design a processor
  - 1. Analyze instruction set → datapath requirements
  - 2. Select set of datapath components & establish clock methodology
  - 3. Assemble datapath meeting the requirements
  - 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
  - 5. Assemble the control logic
    - Formulate Logic Equations
    - Design Circuits

A Summary of the Control Signals (1/2)

<table>
<thead>
<tr>
<th>inst</th>
<th>Register Transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>R[rs] ← R[rt] + R[rt]; PC ← PC + 4</td>
</tr>
<tr>
<td>sub</td>
<td>R[rs] ← R[rs] - R[rt]; PC ← PC + 4</td>
</tr>
<tr>
<td>ori</td>
<td>R[rs] ← R[rs] + zero_ext(Imm16); PC ← PC + 4</td>
</tr>
<tr>
<td>lw</td>
<td>R[rs] ← MEM[R[rs] + sign_ext(Imm16)]; PC ← PC + 4</td>
</tr>
<tr>
<td>sw</td>
<td>MEM[R[rs] + sign_ext(Imm16)] ← R[rt]; PC ← PC + 4</td>
</tr>
<tr>
<td>beq</td>
<td>if (R[rs] == R[rt]) then PC ← PC + sign_ext(Imm16)</td>
</tr>
</tbody>
</table>

A Summary of the Control Signals (2/2)

<table>
<thead>
<tr>
<th>R-type</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add, sub</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td></td>
<td>add, sub</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ori</td>
<td></td>
<td></td>
<td>add, sub</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step 4: Given Datapath: RTL → Control

In Review: A Single Cycle Datapath

- We have everything!
  - Now we just need to know how to BUILD CONTROL

Processor

Memory

Input

Output

A Summary of the Control Signals (1/2)

<table>
<thead>
<tr>
<th>Inst</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Rs</td>
</tr>
<tr>
<td>Sub</td>
<td>Rs</td>
</tr>
<tr>
<td>Ori</td>
<td>Rs</td>
</tr>
<tr>
<td>Lw</td>
<td>Rs</td>
</tr>
<tr>
<td>Sw</td>
<td>Rs</td>
</tr>
<tr>
<td>Beq</td>
<td>Rs</td>
</tr>
</tbody>
</table>

A Summary of the Control Signals (2/2)
**Boolean Expressions for Controller**

RegDst = add + sub
ALUSrc = ori + bs + sv
MemtoReg = bs
RegWrite = add + sub + ori + bs
MemWrite = sv
nPCsel = beq
Jump = jump
ExtOp = bs + sv
ALUctr[0] = sub + beq
(ALUctr is 00)
ExtOp = lw + sw
Jump = jump
nPCsel = beq
MemWrite = sw
MemtoReg = lw
ALUSrc = add

where,

type = op1, op2, op3, op4
ori = op1, op2, op3, op4
bs = op1, op2, op3, op4
sw = op1, op2, op3, op4
beq = op1, op2, op3, op4

add = type * func1 * func2
ori = type * func1 * func2
sub = type * func1 * func2
beq = type * func1 * func2

**Controller Implementation**

**Summary: Single-cycle Processor**

5 steps to design a processor
1. Analyze instruction set ➔ datapath requirements
2. Select set of datapath components & establish clock methodology
3. Assemble datapath meeting the requirements
4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
5. Assemble the control logic
   - Formulate Logic Equations
   - Design Circuits

**Review: Single cycle datapath**

- 5 steps to design a processor
- Analyze instruction set ➔ datapath requirements
- Select set of datapath components & establish clock methodology
- Assemble datapath meeting the requirements
- Analyze implementation of each instruction to determine setting of control points that effects the register transfer.

**How We Build The Controller**

RegDst = add + sub
ALUSrc = ori + bs + sv
MemtoReg = bs
RegWrite = add + sub + ori + bs
MemWrite = sv
nPCsel = beq
Jump = jump
ExtOp = bs + sv
ALUctr[0] = sub + beq
(ALUctr is 00)
ExtOp = lw + sw
Jump = jump
nPCsel = beq
MemWrite = sw
MemtoReg = lw
ALUSrc = add

where,

type = op1, op2, op3, op4
ori = op1, op2, op3, op4
bs = op1, op2, op3, op4
sw = op1, op2, op3, op4
beq = op1, op2, op3, op4

add = type * func1 * func2
ori = type * func1 * func2
sub = type * func1 * func2
beq = type * func1 * func2

Processor Performance

- Can we estimate the clock rate (frequency) of our single-cycle processor? We know:
  - 1 cycle per instruction
  - lw is the most demanding instruction.
  - Assume these delays for major pieces of the datapath:
    - Instr. Mem, ALU, Data Mem: 2ns each, regfile 1ns
    - Instruction execution requires: $2 + 1 + 2 + 2 + 1 = 8$ns
  - $\Rightarrow$ 125 MHz

- What can we do to improve clock rate?
- Will this improve performance as well?
  - We want increases in clock rate to result in programs executing quicker.

Gotta Do Laundry

- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, fold, and put away
  - Washer takes 30 minutes
  - Dryer takes 30 minutes
  - “Folder” takes 30 minutes
  - “Stasher” takes 30 minutes to put clothes into drawers

Sequential Laundry

- Sequential laundry takes 8 hours for 4 loads

Pipelined Laundry

- Pipelined laundry takes 3.5 hours for 4 loads!

General Definitions

- Latency: time to completely execute a certain task
  - for example, time to read a sector from disk is disk access time or disk latency
- Throughput: amount of work that can be done over a period of time

Pipelining Lessons (1/2)

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
  - Multiple tasks operating simultaneously using different resources
  - Potential speedup $= \frac{\text{Number pipe stages}}{\text{Number tasks}}$
  - Time to “fill” pipeline and time to “drain” it reduces speedup: 2.3X v. 4X in this example
Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?

Pipeline rate limited by slowest pipeline stage

Unbalanced lengths of pipe stages reduces speedup

### Administrative

- HW8 due tomorrow
- Project 2 due next Monday
- Newsgroup problems
- Reminder: Midterm regrades due today

### Problems for Pipelining CPUs

- Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle
  - **Structural hazards**: HW cannot support some combination of instructions (single person to fold and put clothes away)
  - **Control hazards**: Pipelining of branches causes later instruction fetches to wait for the result of the branch
  - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline (missing sock)
- These might result in pipeline stalls or “bubbles” in the pipeline.

### Structural Hazard #1: Single Memory (1/2)

- Read same memory twice in same clock cycle
Structural Hazard #1: Single Memory (2/2)

- **Solution:**
  - Infeasible and inefficient to create second memory
  - (We’ll learn about this more next week)
  - So simulate this by having two Level 1 Caches (a temporary smaller [of usually most recently used] copy of memory)
  - Have both an L1 Instruction Cache and an L1 Data Cache
  - Need more complex hardware to control when both caches miss

Structural Hazard #2: Registers (1/2)

- Two different solutions have been used:
  1) RegFile access is VERY fast: takes less than half the time of ALU stage
     - Write to Registers during first half of each clock cycle
     - Read from Registers during second half of each clock cycle
  2) Build RegFile with independent read and write ports

- Result: can perform Read and Write during same clock cycle

Things to Remember

- **Optimal Pipeline**
  - Each stage is executing part of an instruction each clock cycle.
  - One instruction finishes during each clock cycle.
  - On average, execute far more quickly.

- **What makes this work?**
  - Similarities between instructions allow us to use same stages for all instructions (generally).
  - Each stage takes about the same amount of time as all others: little wasted time.
The Single Cycle Datapath during Jump

- New PC = \{ PC[31..28], target address, 00 \}

Instruction Fetch Unit at the End of Jump

- New PC = \{ PC[31..28], target address, 00 \}

How do we modify this to account for jumps?

Query
- Can Zero still get asserted?
- Does nPC_sel need to be 0?
- If not, what?