Working on the Pipeline
Datapath Control Signals

- MemWr: $1 \Rightarrow$ write memory
- MemtoReg: $0 \Rightarrow$ ALU; $1 \Rightarrow$ Mem
- RegDst: $0 \Rightarrow$ ‘rt’; $1 \Rightarrow$ ‘rd’
- RegWr: $1 \Rightarrow$ write register

- ALUctr: "add", "sub", "OR",...
- Extender: $0 \Rightarrow$ zero-ext; $1 \Rightarrow$ sign-ext
- nPC_sel: $0 \Rightarrow$ pc+4; $1 \Rightarrow$ branch-if-equal
Summary of the Control Signals (1/2)

<table>
<thead>
<tr>
<th>Inst</th>
<th>Register Transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add</strong></td>
<td>R[rd] ← R[rs] + R[rt]; PC ← PC + 4 &lt;br&gt; ALUsrc=RegB, ALUctr=&quot;ADD&quot;, RegDst=rd, RegWr, nPC_sel=&quot;+4&quot;</td>
</tr>
<tr>
<td><strong>sub</strong></td>
<td>R[rd] ← R[rs] − R[rt]; PC ← PC + 4 &lt;br&gt; ALUsrc=RegB, ALUctr=&quot;SUB&quot;, RegDst=rd, RegWr, nPC_sel=&quot;+4&quot;</td>
</tr>
<tr>
<td><strong>ori</strong></td>
<td>R[rt] ← R[rs] + zero_ext(Imm16); PC ← PC + 4 &lt;br&gt; ALUsrc=Im, Extop=&quot;Z&quot;, ALUctr=&quot;OR&quot;, RegDst=rt, RegWr, nPC_sel=&quot;+4&quot;</td>
</tr>
<tr>
<td><strong>lw</strong></td>
<td>R[rt] ← MEM[ R[rs] + sign_ext(Imm16)]; PC ← PC + 4 &lt;br&gt; ALUsrc=Im, Extop=&quot;sn&quot;, ALUctr=&quot;ADD&quot;, MemtoReg, RegDst=rt, RegWr, nPC_sel = &quot;+4&quot;</td>
</tr>
<tr>
<td><strong>sw</strong></td>
<td>MEM[ R[rs] + sign_ext(Imm16)] ← R[rs]; PC ← PC + 4 &lt;br&gt; ALUsrc=Im, Extop=&quot;sn&quot;, ALUctr = &quot;ADD&quot;, MemWr, nPC_sel = &quot;+4&quot;</td>
</tr>
<tr>
<td><strong>beq</strong></td>
<td>if (R[rs] == R[rt]) then PC ← PC + sign_ext(Imm16)</td>
</tr>
</tbody>
</table>
Summary of the Control Signals (2/2)

See Appendix A

<table>
<thead>
<tr>
<th></th>
<th>func 00000</th>
<th>func 00010</th>
<th>We Don’t Care :-)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>1</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>sub</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>ori</td>
<td>0</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>sw</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>beq</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>jump</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
</tbody>
</table>

- RegDst: 1 1 0 0 x x x
- ALUSrc: 0 0 1 1 1 0 x
- MemtoReg: 0 0 0 1 x x x
- RegWrite: 1 1 1 1 0 0 0
- MemWrite: 0 0 0 0 1 0 0
- nPCsel: 0 0 0 0 0 1 ?
- Jump: 0 0 0 0 0 0 1
- ExtOp: x x 0 1 1 x x
- ALUctr<2:0>: Add Subtract Or Add Add Subtract x

<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>add, sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-type</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td>immediate</td>
<td>ori, lw, sw, beq</td>
</tr>
<tr>
<td>J-type</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>target address</td>
<td>jump</td>
</tr>
</tbody>
</table>
Boolean Expressions for Controller

RegDst    = add + sub
ALUSrc    = ori + lw + sw
MemtoReg  = lw
RegWrite  = add + sub + ori + lw
MemWrite  = sw
nPcSel    = beq
Jump      = jump
ExtOp     = lw + sw
ALUctr[0] = sub + beq  (assume ALUctr is 00 ADD, 01 SUB, 10 OR)
ALUctr[1] = or

Where:

rtype = ~op_5 • ~op_4 • ~op_3 • ~op_2 • ~op_1 • ~op_0,
ori  = ~op_5 • ~op_4 • op_3 • op_2 • ~op_1 • op_0
lw   = op_5 • ~op_4 • ~op_3 • ~op_2 • op_1 • op_0
sw   = op_5 • ~op_4 • op_3 • ~op_2 • op_1 • op_0
beq  = ~op_5 • ~op_4 • ~op_3 • op_2 • ~op_1 • ~op_0
jump = ~op_5 • ~op_4 • ~op_3 • ~op_2 • op_1 • ~op_0

add = rtype • func_5 • ~func_4 • ~func_3 • ~func_2 • ~func_1 • ~func_0
sub = rtype • func_5 • ~func_4 • ~func_3 • ~func_2 • func_1 • ~func_0

How do we implement this in gates?
Controller Implementation

“AND” logic

- opcode
- func
- add
- sub
- ori
- lw
- sw
- beq
- jump

“OR” logic

- RegDst
- ALUSrc
- MemtoReg
- RegWrite
- MemWrite
- nPCsel
- Jump
- ExtOp
- ALUctr[0]
- ALUctr[1]
P&H Figure 4.17
Summary: Single-cycle Processor

- Five 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
Single Cycle Performance

- Assume time for actions are
  - 100ps for register read or write; 200ps for other events

<table>
<thead>
<tr>
<th>Instr</th>
<th>Instr fetch</th>
<th>Register read</th>
<th>ALU op</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td>100 ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
</tbody>
</table>

- What can we do to improve clock rate?
- Will this improve performance as well?
  Want increased clock rate to mean faster programs
Gotta Do Laundry

- Alice, Bob, Carol, and 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!
Pipelining Lessons (1/2)

- Pipelining doesn’t help latency of single task, it helps **throughput** of entire workload
- Multiple tasks operating simultaneously and independently using different resources
- Potential speedup = Number pipe stages
- Time to “fill” pipeline and time to “drain” it reduces speedup: $2.3x (8/3.5)$ v. $4x (8/2)$ in this example
Pipelining Lessons (2/2)

- Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?
  - Pipeline rate limited by slowest pipeline stage
- Suppose Bob doesn't bother folding his laundry?
  - Idle steps in the pipeline don't enable others to fold
- Unbalanced lengths and idle stages reduces speedup
Execution Steps in MIPS Datapath

• 1) **IFetch/IF**: Instruction Fetch & Increment PC
• 2) **Dcd/ID**: Instruction Decode & Read Registers
• 3) **Exec/EX**:
  - Mem-ref: Calculate Address
  - Arith-log: Perform ALU Operation
• 4) **Mem**:
  - Load: Read Data from Memory
  - Store: Write Data to Memory
  • Memory is now synchronous
• 5) **WB**: Write Data Back to Register
Single Cycle Datapath

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Write Back
Pipeline registers

- Need registers between stages
- To hold information produced in previous cycle
More Detailed Pipeline
IF for Load, Store, …
ID for Load, Store, …
EX for Load
MEM for Load
WB for Load – Oops!

Wrong register number!
Corrected Datapath for Load
Pipelined Execution Representation

- Every instruction must take the same number of steps, so some stages will idle.
- e.g. MEM stage for any arithmetic instruction.

<table>
<thead>
<tr>
<th>Time</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>
Graphical Pipeline Diagrams

- Use datapath figure below to represent pipeline:

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Write Back
Graphical Pipeline Representation

- RegFile: left half is write, right half is read

Time (clock cycles)
Pipelining Performance (1/3)

- Use $T_c$ ("time between completion of instructions") to measure speedup

\[ T_{c,\text{pipelined}} \geq \frac{T_{c,\text{single-cycle}}}{\text{Number of stages}} \]

- Equality only achieved if stages are balanced (i.e. take the same amount of time)
- If not balanced, speedup is reduced
- Speedup due to increased throughput
  - *Latency* for each instruction does not decrease
Pipelining Performance (2/3)

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages

<table>
<thead>
<tr>
<th>Instr</th>
<th>Instr fetch</th>
<th>Register read</th>
<th>ALU op</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td>100 ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
</tbody>
</table>

- What is pipelined clock rate?
  - Compare pipelined datapath with single-cycle datapath
Pipelining Performance (3/3)

Single-cycle

\[ T_c = 800 \text{ ps} \]
\[ f = 1.25 \text{GHz} \]

Pipelined

\[ T_c = 200 \text{ ps} \]
\[ f = 5 \text{GHz} \]
Clicker/Peer Instruction

Logic in some stages takes 200ps and in some 100ps. Clk-Q delay is 30ps and setup-time is 20ps. What is the maximum clock frequency at which a pipelined design can operate?

- A: 10GHz
- B: 5GHz
- C: 6.7GHz
- D: 4.35GHz
- E: 4GHz
Project 3.1...

- Project 3.1 will be released in a couple of hours
  - In project 3, you will build a CPU in logisim.
- 3.1 is the ALU and register file
- 3.2 is putting together the control logic
We Grossly Simplified The Project... Why?

- Last semester and last year it was building effectively a full MIPS
  - Now it is a much smaller architecture with narrower words: Why are we cheating you out of the experience?
- We use logisim for pedagogical reasons
  - Almost all design these days uses "HDL" (High-Level Design Languages) like VHDL and Verilog
- In an HDL, doing a 32b, 32 register register file is no harder than doing a 16b, 8 register one
  - But in logisim, it is at least 4x more work... And 4x more chance to make an error
Why Nick's Ph.D. Was An Incredibly Stupid Idea...

- My Ph.D. was on a highly pipelined FPGA architecture
- FPGA -> Field Programmable Gate Array: Basically programmable hardware
- The design was centered around being able to pipeline multiple independent tasks
  - We will see on Wednesday how to handle "pipeline hazards" and "forwarding:
    1. add $s0 $s1 $s2
    2. add $s3 $s0 $s4
  - This is critical to get real performance gains
  - But my dissertation design didn't have this ability
- I also showed how you could use the existing registers in the FPGA to heavily pipeline it automatically
But pipelining is \textit{not} free!

- Not only does pipelining not improve latency...
  - It actually makes it worse!

- Two sources:
  - Unbalanced pipeline stages
  - The setup & clk->q time for the pipeline registers

- Pipelining only independent tasks also can't "forward"

- So \textit{independent} task pipelining is only about reducing cost
  - You can always just duplicate logic instead

- \textbf{Latency is fundamental, independent task throughput} can always be solved by throwing $$ at the problem

- So I proved my Ph.D. design was \textbf{no better} than the conventional FPGA on throughput/$ and far far far worse on latency!