CS 61C:
Great Ideas in Computer Architecture
Control and Pipelining

Instructors:
Vladimir Stojanovic and Nicholas Weaver
http://inst.eecs.Berkeley.edu/~cs61c/sp16
Datapath Control Signals

- **ExtOp:** “zero”, “sign”
- **ALUsrc:**
  - 0 ⇒ regB;
  - 1 ⇒ immed
- **ALUctr:** “ADD”, “SUB”, “OR”
- **MemWr:** 1 ⇒ write memory
- **MemtoReg:**
  - 0 ⇒ ALU;
  - 1 ⇒ Mem
- **RegDst:**
  - 0 ⇒ “rt”;
  - 1 ⇒ “rd”
- **RegWr:** 1 ⇒ write register
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[rd] \leftarrow R[rs] + R[rt]; \text{PC} \leftarrow \text{PC} + 4 )</td>
</tr>
<tr>
<td></td>
<td>ALUsrc=RegB, ALUctr=“ADD”, RegDst=rd, RegWr, nPC_sel=“+4”</td>
</tr>
<tr>
<td>sub</td>
<td>( R[rd] \leftarrow R[rs] - R[rt]; \text{PC} \leftarrow \text{PC} + 4 )</td>
</tr>
<tr>
<td></td>
<td>ALUsrc=RegB, ALUctr=“SUB”, RegDst=rd, RegWr, nPC_sel=“+4”</td>
</tr>
<tr>
<td>ori</td>
<td>( R[rt] \leftarrow R[rs] + \text{zero_ext}(\text{Imm16}); \text{PC} \leftarrow \text{PC} + 4 )</td>
</tr>
<tr>
<td></td>
<td>ALUsrc=Im, Extop=“Z”, ALUctr=“OR”, RegDst=rt,RegWr, nPC_sel=“+4”</td>
</tr>
<tr>
<td>lw</td>
<td>( R[rt] \leftarrow \text{MEM}[ R[rs] + \text{sign_ext}(\text{Imm16})]; \text{PC} \leftarrow \text{PC} + 4 )</td>
</tr>
<tr>
<td></td>
<td>ALUsrc=Im, Extop=“sn”, ALUctr=“ADD”, MemtoReg, RegDst=rt, RegWr, nPC_sel = “+4”</td>
</tr>
<tr>
<td>sw</td>
<td>( \text{MEM}[ R[rs] + \text{sign_ext}(\text{Imm16})] \leftarrow R[rs]; \text{PC} \leftarrow \text{PC} + 4 )</td>
</tr>
<tr>
<td></td>
<td>ALUsrc=Im, Extop=“sn”, ALUctr = “ADD”, MemWr, nPC_sel = “+4”</td>
</tr>
<tr>
<td>beq</td>
<td>if (R[rs] == R[rt]) then ( \text{PC} \leftarrow \text{PC} + \text{sign_ext}(\text{Imm16}) )</td>
</tr>
<tr>
<td></td>
<td>else ( \text{PC} \leftarrow \text{PC} + 4 )</td>
</tr>
<tr>
<td></td>
<td>nPC_sel = “br”, ALUctr = “SUB”</td>
</tr>
</tbody>
</table>
## Summary of the Control Signals (2/2)

### Control Signals Table

<table>
<thead>
<tr>
<th>op</th>
<th>R-type</th>
<th>I-type</th>
<th>J-type</th>
<th>We Don’t Care :-)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0000</td>
<td>00 0000</td>
<td>00 1101</td>
<td>10 0011</td>
<td>10 1011</td>
</tr>
<tr>
<td>add</td>
<td>sub</td>
<td>ori</td>
<td>lw</td>
<td>sw</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemWrite</th>
<th>nPCsel</th>
<th>Jump</th>
<th>ExtOp</th>
<th>ALUctr&lt;2:0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>?</td>
<td>x</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>?</td>
<td>x</td>
</tr>
</tbody>
</table>

### Appendices

See Appendix A for more details.

### Instruction Encoding

- **R-type**: `op rs rt rd shamt funct` (add, sub, ori, lw, sw, beq)
- **I-type**: `op rs rt immediate` (ori, lw, sw, beq)
- **J-type**: `op target address` (jump)
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:

\[
\begin{align*}
\text{rtype} &= \neg \text{op}_5 \cdot \neg \text{op}_4 \cdot \neg \text{op}_3 \cdot \neg \text{op}_2 \cdot \neg \text{op}_1 \cdot \text{op}_0, \\
\text{ori}  &= \neg \text{op}_5 \cdot \neg \text{op}_4 \cdot \text{op}_3 \cdot \text{op}_2 \cdot \neg \text{op}_1 \cdot \text{op}_0 \\
\text{lw}   &= \text{op}_5 \cdot \neg \text{op}_4 \cdot \neg \text{op}_3 \cdot \neg \text{op}_2 \cdot \text{op}_1 \cdot \text{op}_0 \\
\text{sw}   &= \text{op}_5 \cdot \neg \text{op}_4 \cdot \text{op}_3 \cdot \neg \text{op}_2 \cdot \text{op}_1 \cdot \text{op}_0 \\
\text{beq}  &= \neg \text{op}_5 \cdot \neg \text{op}_4 \cdot \neg \text{op}_3 \cdot \text{op}_2 \cdot \neg \text{op}_1 \cdot \neg \text{op}_0 \\
\text{jump} &= \neg \text{op}_5 \cdot \neg \text{op}_4 \cdot \neg \text{op}_3 \cdot \neg \text{op}_2 \cdot \text{op}_1 \cdot \neg \text{op}_0 \\
\text{add}  &= \text{rtype} \cdot \text{func}_5 \cdot \neg \text{func}_4 \cdot \neg \text{func}_3 \cdot \neg \text{func}_2 \cdot \neg \text{func}_1 \cdot \neg \text{func}_0 \\
\text{sub}  &= \text{rtype} \cdot \text{func}_5 \cdot \neg \text{func}_4 \cdot \neg \text{func}_3 \cdot \neg \text{func}_2 \cdot \text{func}_1 \cdot \neg \text{func}_0
\end{align*}
\]

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

- Clock period is?

<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>

- Clock rate (cycles/second = Hz) = 1/Period (seconds/cycle)
Single Cycle Performance

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

- Clock period is?

<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
Levels of Representation/Interpretation

- **High Level Language Program (e.g., C)**
- **Assembly Language Program (e.g., MIPS)**
- **Machine Language Program (MIPS)**

**Compiler**

**Assembler**

**Machine Interpretation**

**Hardware Architecture Description (e.g., block diagrams)**

**Architecture Implementation**

**Logic Circuit Description (Circuit Schematic Diagrams)**

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```asm
lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)
```

Anything can be represented as a *number*, i.e., data or instructions

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```
No More Magic!

- Application (ex: browser)
- Operating System (Mac OSX)
- Compiler
- Assembler
- Processor
- Memory
- I/O system
- Datapath & Control
- Digital Design
- Circuit Design
- Transistors

Courses:
- CS61A
- CS61B
- CS61C
- EE40
- Phys 7B
Administrivia

- Project 2-2 due 3/8 @ 23:59:59 (Tue)
- Guerrilla Sessions: MIPS CPU
  - Wed 3/09 3 - 5 PM @ 241 Cory
  - Sat 3/12 1 - 3 PM @ 651 @ 611 Soda

Review Grades for Midterm 1

- Minimum: 6.0
- Median: 68.0
- Maximum: 98.0
- Mean: 64.97
- Std Dev: 19.57
**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!
Pipelining doesn’t help latency of single task, it helps throughput of entire workload.

- Multiple tasks operating simultaneously 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
- Unbalanced lengths of pipe stages reduces speedup
Execution Steps in MIPS Datapath

1) **IFetch**: Instruction **Fetch**, Increment PC
2) **Dcd**: Instruction **Decode**, Read Registers
3) **Exec**:
   Mem-ref: Calculate Address
   Arith-log: Perform Operation
4) **Mem**:
   Load: Read Data from Memory
   Store: Write Data to Memory
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 same number of steps, so some stages will idle
  – e.g. MEM stage for any arithmetic instruction
Graphical Pipeline Diagrams

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

• Use datapath figure below to represent pipeline:
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
• 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**

\( T_c = 800 \text{ ps} \quad f = 1.25\text{GHz} \)

- \( \text{lw} \ $1, 100($0) \)
- \( \text{lw} \ $2, 200($0) \)
- \( \text{lw} \ $3, 300($0) \)

**Pipelined**

\( T_c = 200 \text{ ps} \quad f = 5\text{GHz} \)

- \( \text{lw} \ $1, 100($0) \)
- \( \text{lw} \ $2, 200($0) \)
- \( \text{lw} \ $3, 300($0) \)
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