CS 61C: Great Ideas in Computer Architecture (Machine Structures)
MIPS Control, MIPS Pipelining Intro

Instructor:
Michael Greenbaum
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;
```

```
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
```
Review: A Single Cycle Datapath

[Diagram of a single cycle datapath]
Agenda

• Overview of Control Signals
• Administrivia
• Control Implementation
• Break
• Pipelining Intro
Processor Design Process

• Five steps to design a processor:
  Step 1: Analyze instruction set to determine datapath requirements
  Step 2: Select set of datapath components & establish clocking methodology
  Step 3: Assemble datapath components to meet the requirements
  **Step 4: Analyze implementation of each instruction to determine setting of control signals that implements the register transfer**
  Step 5: Assemble the control logic
A Single Cycle Datapath

- We have everything but the **values of control signals**

---

RegDst

RegWr

Instr fetch unit

MemtoReg

MemWr

Data Memory

ExtOp

ALUSrc
Values of the Control Signals

- \text{nPC\_sel:} 
  - \text{“+4” 0} \Rightarrow \text{PC} \leftarrow \text{PC} + 4 
  - \text{“br” 1} \Rightarrow \text{PC} \leftarrow \text{PC} + 4 + \{\text{SignExt(Im16)} , 00 \}
Values of the Control Signals

- **ExtOp**: zero, sign
- **ALUsrc**: 0 → reg; 1 → imm
- **ALUctr**: ADD, SUB, OR

  - **MemWr**: 1 → write memory
  - **MemtoReg**: 0 → alu; 1 → mem
  - **RegDst**: 0 → rt; 1 → rd
  - **RegWr**: 1 → write register
RTL: The **Add** Instruction

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
<td></td>
</tr>
<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>
<td></td>
</tr>
</tbody>
</table>

**add rd, rs, rt**

- MEM[PC]  
  Fetch the instruction from memory
- R[rd] = R[rs] + R[rt]  
  The actual operation
- PC = PC + 4  
  Calculate the next instruction’s address
Single Cycle Datapath during Add

R[rd] = R[rs] + R[rt]
Instruction Fetch Unit for Add

- \( \text{PC} = \text{PC} + 4 \)
  - Same for all instructions except: Branch and Jump

\[
\begin{align*}
\text{Inst Address} & \quad \text{Inst Memory} \\
\text{Adder} & \quad \text{Mux} \\
\text{Adder} & \quad \text{PC Ext} \\
\text{nPC\_sel=+4} & \quad \text{clk} \\
\text{imm16} & \quad \text{PC} \quad 00
\end{align*}
\]
Single Cycle Datapath during Or Immediate

- \( R[rt] = R[rs] \text{ OR ZeroExt}[\text{Imm16}] \)
• \( R[rt] = R[rs] \text{ OR} \text{ ZeroExt}[\text{Imm16}] \)
Single Cycle Datapath during Load

- \( R[rt] = \text{Data Memory} \{R[rs] + \text{SignExt}[\text{imm16}] \} \)
Single Cycle Datapath during Load

- \( R[rt] = \text{Data Memory} \{R[rs] + \text{SignExt[imm16]}\} \)
Single Cycle Datapath during Branch

- if (R[rs] - R[rt] == 0) then Zero = 1; else Zero = 0
Single Cycle Datapath during Branch

- if \((R_{rs} - R_{rt} == 0)\) then Zero = 1; else Zero = 0
Instruction Fetch Unit at the End of Branch

- if (Zero == 1) then PC = PC + 4 + SignExt[imm16]*4 ; else PC = PC + 4

Q: What logic gate?
Summary: Datapath’s Control Signals

- **ExtOp:** “zero”, “sign”
- **ALUsrc:**
  - 0 \(\Rightarrow\) regB;
  - 1 \(\Rightarrow\) immed
- **ALUctr:** “ADD”, “SUB”, “OR”
- **MemWr:** 1 \(\Rightarrow\) write memory
- **MemtoReg:**
  - 0 \(\Rightarrow\) ALU;
  - 1 \(\Rightarrow\) Mem
- **RegDst:**
  - 0 \(\Rightarrow\) “rt”;
  - 1 \(\Rightarrow\) “rd”
- **RegWr:** 1 \(\Rightarrow\) write register
Administrivia

• HW3 Due Wednesday at midnight
• Project 2 Part 2 due Sunday.
  – Slides at end of July 12 lecture contain useful info.
  – Project 2 Extra Credit: If you have time after finishing Part 2, try to make your program as fast as possible! Top submissions in the class get lots of extra credit, substantial improvement will get some.
• Lab 12 cancelled!
  – Replaced with free study session where you can catch up on labs / work on project 2.
  – The TA’s will still be there.
cs61c in the News

• Use power dissipation from servers as source of heating?
• So-called “Data Furnaces.” Sell the heat to consumers.
• “Additionally, such a setup would also provide lower network latency as the storage and computation systems can be located closer to areas of high population density and therefore those using them.”

Processor Design Process

• Five steps to design a processor:
  Step 1: Analyze instruction set to determine datapath requirements
  Step 2: Select set of datapath components & establish clocking methodology
  Step 3: Assemble datapath components to meet the requirements
  Step 4: Analyze implementation of each instruction to determine setting of control signals that implements the register transfer
  **Step 5: Assemble the control logic**
Given Datapath: RTL → Control

Instruction<31:0>

Inst Memory
Adr

Op Fun Rt Rs Rd Imm16

nPC sel RegWr RegDst ExtOp ALUSrc ALUctr MemWr MemtoReg

DATA PATH

Control
Summary of the Control Signals (1/2)

<table>
<thead>
<tr>
<th><strong>inst</strong></th>
<th><strong>Register Transfer</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add</strong></td>
<td>( R[rd] \leftarrow R[rs] + R[rt]; ) ( PC \leftarrow PC + 4 )</td>
</tr>
<tr>
<td></td>
<td>( ALU_{src}=RegB, ) ( ALU_{ctr} = “ADD”; ) ( RegDst=rd, ) ( RegWr, ) ( nPC_sel = “+4” )</td>
</tr>
<tr>
<td><strong>sub</strong></td>
<td>( R[rd] \leftarrow R[rs] - R[rt]; ) ( PC \leftarrow PC + 4 )</td>
</tr>
<tr>
<td></td>
<td>( ALU_{src}=RegB, ) ( ALU_{ctr} = “SUB”; ) ( RegDst=rd, ) ( RegWr, ) ( nPC_sel = “+4” )</td>
</tr>
<tr>
<td><strong>ori</strong></td>
<td>( R[rt] \leftarrow R[rs] + \text{zero_ext(Imm16)}; ) ( PC \leftarrow PC + 4 )</td>
</tr>
<tr>
<td></td>
<td>( ALU_{src}=\text{Im}, ) ( Extop = “Z”, ) ( ALU_{ctr} = “OR”; ) ( RegDst=rt, ) ( RegWr, ) ( nPC_sel = “+4” )</td>
</tr>
<tr>
<td><strong>lw</strong></td>
<td>( R[rt] \leftarrow \text{MEM}[R[rs] + \text{sign_ext(Imm16)]}; ) ( PC \leftarrow PC + 4 )</td>
</tr>
<tr>
<td></td>
<td>( ALU_{src}=\text{Im}, ) ( Extop = “sn”, ) ( ALU_{ctr} = “ADD”; ) ( MemtoReg, ) ( RegDst=rt, ) ( RegWr, ) ( nPC_sel = “+4” )</td>
</tr>
<tr>
<td><strong>sw</strong></td>
<td>( \text{MEM}[R[rs] + \text{sign_ext(Imm16)]} \leftarrow R[rs]; ) ( PC \leftarrow PC + 4 )</td>
</tr>
<tr>
<td></td>
<td>( ALU_{src}=\text{Im}, ) ( Extop = “sn”, ) ( ALU_{ctr} = “ADD”; ) ( MemWr, ) ( nPC_sel = “+4” )</td>
</tr>
<tr>
<td><strong>Beq</strong></td>
<td>if ( (R[rs] == R[rt]) ) then ( PC \leftarrow PC + \text{sign_ext(Imm16)]</td>
</tr>
<tr>
<td></td>
<td>else ( PC \leftarrow PC + 4 )</td>
</tr>
<tr>
<td></td>
<td>( nPC_sel = “br”, ) ( ALU_{ctr} = “SUB” )</td>
</tr>
</tbody>
</table>
### Summary of the Control Signals (2/2)

See Appendix A

<table>
<thead>
<tr>
<th>func</th>
<th>10 0000</th>
<th>10 0010</th>
<th>We Don’t Care :-)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00 0000</td>
<td>00 0000</td>
<td>00 1101</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>sub</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ori</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>jump</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

#### R-type
- **op**: add, sub
- **rs**, **rt**: ori, lw, sw, beq
- **rd**: jump
- **shamt**, **funct**: target address

#### I-type
- **op**: add, sub
- **rs**, **rt**: ori, lw, sw, beq
- **immediate**: jump

#### J-type
- **op**: add, sub
- **target address**: ori, lw, sw, beq
- **jump**: jump
Boolean Expressions for Controller

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

Where:

\[
\begin{align*}
\text{rtype} &= \overline{op_5} \cdot \overline{op_4} \cdot \overline{op_3} \cdot \overline{op_2} \cdot \overline{op_1} \cdot \overline{op_0}, \\
\text{ori} &= \overline{op_5} \cdot \overline{op_4} \cdot op_3 \cdot op_2 \cdot \overline{op_1} \cdot op_0 \\
\text{lw} &= op_5 \cdot \overline{op_4} \cdot \overline{op_3} \cdot \overline{op_2} \cdot op_1 \cdot op_0 \\
\text{sw} &= op_5 \cdot \overline{op_4} \cdot op_3 \cdot \overline{op_2} \cdot op_1 \cdot op_0 \\
\text{beq} &= \overline{op_5} \cdot \overline{op_4} \cdot \overline{op_3} \cdot op_2 \cdot \overline{op_1} \cdot \overline{op_0} \\
\text{jump} &= \overline{op_5} \cdot \overline{op_4} \cdot \overline{op_3} \cdot \overline{op_2} \cdot op_1 \cdot \overline{op_0} \\
\text{add} &= \text{rtype} \cdot \text{func}_5 \cdot \overline{\text{func}_4} \cdot \overline{\text{func}_3} \cdot \overline{\text{func}_2} \cdot \overline{\text{func}_1} \cdot \overline{\text{func}_0} \\
\text{sub} &= \text{rtype} \cdot \text{func}_5 \cdot \overline{\text{func}_4} \cdot \overline{\text{func}_3} \cdot \overline{\text{func}_2} \cdot \text{func}_1 \cdot \overline{\text{func}_0}
\end{align*}
\]

How do we implement this in gates?
Controller Implementation

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>fun</td>
</tr>
</tbody>
</table>

```

```
“AND” logic

```

```
opcodes:
- add
- sub
- ori
- lw
- sw
- beq
- jump

```

```
“OR” logic

```

```
output:
- RegDst
- ALUSrc
- MemtoReg
- RegWrite
- MemWrite
- nPCsel
- Jump
- ExtOp
- ALUctr[0]
- ALUctr[1]

```

“AND” logic

```

“OR” logic

```
An Abstract View of the Implementation

Control

Datapath

Ideal Instruction Memory

Instruction Address

Next Address

clk

Instruction

Rd 5
Rs 5
Rt 5

Rw Ra Rb

Register File

clk

A
32

B

32

32

ALU

Data Addr

Data In

clk

Ideal Data Memory

Data Out

Ideal Data Memory

Instruction

Control Signals

Conditions

clk

Instruction

Data

In

Out
Call home, we’ve made HW/SW contact!

- **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)

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

```
1010 1111 0101 1000
0000 1001 1100 0110
0101 1000 0000 1001
1100 0110 1010 1111
```
Agenda

• Overview of Control Signals
• Administrivia
• Control Implementation
• Break
• Pipelining Intro
Agenda

• Overview of Control Signals
• Administrivia
• Control Implementation
• Break
• Pipelining Intro
Single Cycle Performance

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

• Clock rate 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
Pipeline Analogy: Doing 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 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
Recall: Steps in Executing MIPS

1) **IF**: Instruction Fetch, Increment PC
2) **ID**: Instruction Decode, Read Registers
3) **EX**:
   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
Redrawn Single-Cycle Datapath

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

PC → instruction memory → registers → ALU → Data memory → PC

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

• Add registers between stages
  – Hold information produced in previous cycle
More Detailed Pipeline
IF for Load, Store, ...
ID for Load, Store, ...
EX for Load
MEM for Load

[Diagram of memory management process for load operations, including stages: IF/ID, ID/EX, EX/MEM, and MEM/WB, with instructions for addressing and data flow through registers and memory.]
WB for Load

Wrong register number
Corrected Datapath for Load
Every instruction must take the same number of steps, also called pipeline stages, so some will go idle sometimes.
Graphical Pipeline Diagrams

- Use datapath figure below to represent pipeline

```
+4

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

IF | ID | EX | Mem | WB
---|----|----|-----|-----
I$ | Reg| ALU| D$  | Reg|
```
Graphical Pipeline Representation

(In Reg, right half highlight read, left half write)

Time (clock cycles)
Pipeline Performance

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

- What is pipelined clock rate?
  - Compare pipelined datapath with single-cycle datapath

<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>
Pipeline Performance

Program execution order (in instructions)

Single-cycle ($T_c = 800\text{ps}$)

- lw $1, 100(0)$
- lw $2, 200(0)$
- lw $3, 300(0)$

Pipelined ($T_c = 200\text{ps}$)

- lw $1, 100(0)$
- lw $2, 200(0)$
- lw $3, 300(0)$

Time

- 200 ps
- 800 ps
- 200 ps
- 200 ps
Pipeline Speedup

• If all stages are balanced
  – i.e., all take the same time
  – Time between instructions\(_{\text{pipelined}}\) = Time between instructions\(_{\text{nonpipelined}}\) / Number of stages

• If not balanced, speedup is less

• Speedup due to increased throughput
  – Latency (time for each instruction) does not decrease
Instruction Level Parallelism (ILP)

- **ILP** – *e.g.*, **Pipelined Instruction Execution**
  - 5 stage pipeline => 5 instructions executing simultaneously, one at each pipeline stage
And in Conclusion, ...

• Can determine control signal values for each instruction.

• Control implementation: Just combinational logic (at least for a single cycle CPU...)

• Pipelining
  – Reduce critical path by inserting registers between each stage
  – N-stage pipeline => N operations done in parallel!
Bonus slides

• Note: You are still responsible for knowing the material covered in these slides.

• The slides will appear in the order they would have in the normal presentation.
Single Cycle Datapath during Store

- Data Memory \{R[rs] + \text{SignExt}[\text{imm16}]\} = R[rt]
Single Cycle Datapath during Store

- Data Memory \(\{R[rs] + \text{SignExt}[\text{imm16}]\} = R[rt]\)
• New PC = \{ PC[31..28], target address, 00 \}
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 }
Instruction Fetch Unit at the End of Jump

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

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