Designing Single Cycle Control

Computer Science 104

Administrivia
° HW #4 Due March 21
° Midterm II March 26 (logic design, arithmetic, data path & control)
° Projects, Due April 20
° What to hand in:
  - Project description: e.g., interpretations of instructions and description of principles behind design
  - Source code (commented)
  - Design files (full project directory)
  - A description of each team member’s contribution.
  - Demonstration April 20 in class
° Reading: Chapter 5 & appendix C (control)
  - Appendix B.10 finite state machines next time

Recap: The MIPS Instruction Formats
° All MIPS instructions are 32 bits long. The three instruction formats:
  - R-type
    - op: operation of the instruction
    - rs, rt, rd: the source and destination registers specifier
    - shamt: shift amount
    - funct: selects the variant of the operation in the "op" field
    - address / immediate: address offset or immediate value
    - target address: target address of the jump instruction
  - J-type
    - op: operation of the instruction
    - rs, rt, rd: the source and destination registers specifier
    - shamt: shift amount
    - funct: selects the variant of the operation in the "op" field
    - address / immediate: address offset or immediate value
    - target address: target address of the jump instruction
  - I-type
    - op: operation of the instruction
    - rs, rt, rd: the source and destination registers specifier
    - shamt: shift amount
    - funct: selects the variant of the operation in the "op" field
    - address / immediate: address offset or immediate value
    - target address: target address of the jump instruction

Recap: The MIPS Subset
° ADD and subtract
  - add rd, rs, rt
  - sub rd, rs, rt
° OR imm:
  - ori rt, rs, imm16
° LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16
° BRANCH:
  - beq rs, rt, imm16
° JUMP:
  - j target

Recap: A Single Cycle Datapath

RTL: The Branch Instruction
° beq rs, rt, imm16
  - mem[PC] Fetch the instruction from memory
  - Cond <= R[rs] - R[rt] Calculate the branch condition
  - if (COND eq 0) Calculate the next instruction’s address
    - PC <= PC + 4 + (SignExt(imm16) x 4)
    - else
      - PC <= PC + 4
**Datapath for Branch Operations**

We need to compare Rs and Rt!

- compare Rs and Rt
- ALU compares the two values
- output is 0 or 1
- control signals to select ALU output

**Binary Arithmetic for the Next Address**

- In theory, the PC is a 32-bit byte address into the instruction memory:
  - Sequential operation: \( PC<31:0> = PC<31:0> + 4 \)
  - Branch operation: \( PC<31:2> = PC<31:2> + 4 + \text{SignExt}[\text{imm16}] \times 4 \)

- The magic number “4” always comes up because:
  - The 32-bit PC is a byte address
  - All our instructions are 4 bytes (32 bits) long

- In other words:
  - The 2 LSBs of the 32-bit PC are always zeros
  - There is no reason to have hardware to keep the 2 LSBs

- In practice, we can simplify the hardware by using a 30-bit PC:<br>
  - Sequential operation: \( PC<31:2> = PC<31:2> + 1 \)
  - Branch operation: \( PC<31:2> = PC<31:2> + 1 + \text{SignExt}[\text{imm16}] \)

  - In either case: Instruction-Memory-Address = \( PC<31:2> \text{ concat } "00" \)

**Next Address Logic: Expensive and Fast Solution**

- Using a 30-bit PC:
  - Sequential operation: \( PC<31:2> = PC<31:2> + 1 \)
  - Branch operation: \( PC<31:2> = PC<31:2> + 1 + \text{SignExt}[\text{imm16}] \)
  - In either case: Instruction-Memory-Address = \( PC<31:2> \text{ concat } "00" \)

**Next Address Logic, less expensive but slower**

**Why is this slower?**

**RTL: The Jump Instruction**

- \( \text{mem}[\text{PC}] \) Fetch the instruction from memory
- \( \text{PC} \leftarrow \text{PC}+4<31:28> \text{ concat target}+25:0 \text{ concat } "00" \) Calculate the next instruction’s address

**Instruction Fetch Unit**

- \( \text{PC} \leftarrow \text{PC}+4<31:28> \text{ concat target}+25:0 \text{ concat } "00" \)
Putting it All Together: A Single Cycle Datapath

- We have everything except control signals. (underlined)

The Big Picture: Where are We Now?

- The Five Classic Components of a Computer

Outline of Remainder of Today's Lecture

- Control for Register-Register & Or Immediate instructions
- Control signals for Load, Store, Branch, & Jump
- Building a local controller: ALU Control
- The main controller
- Summary

RTL: The ADD Instruction

- add rd, rs, rt
  - mem[PC] Fetch the instruction from memory
  - PC <- PC + 4 Calculate the next instruction’s address

Instruction Fetch Unit at the Beginning of Add / Subtract

- Fetch the instruction from Instruction memory: Instruction <- mem[PC]
  - This is the same for all instructions

The Single Cycle Datapath during Add and Subtract

- R[rd] <- R[rs] + / - R[rt]
Instruction Fetch Unit at the End of Add and Subtract

- PC <- PC + 4
- This is the same for all instructions except: Branch and Jump

The Single Cycle Datapath during Or Immediate

- R[rt] <- R[rs] or ZeroExt[imm16]

The Single Cycle Datapath during Load

- R[rt] <- Data Memory (R[rs] + SignExt[imm16])

The Single Cycle Datapath during Store (fill it in)

- Data Memory (R[rs] + SignExt[imm16]) <- R[rt]

The Single Cycle Datapath during Branch

- if (R[rs] · R[rt] == 0) then Zero <- 1; else Zero <- 0
- if (Zero == 1) then PC = PC + 4 + SignExt[imm16]*4; else PC = PC + 4

Instruction Fetch Unit at the End of Branch

- if (Zero == 1) then PC = PC + 4 + SignExt[imm16]*4; else PC = PC + 4
- Assume Zero = 1 to see the interesting case.
The Single Cycle Datapath during Jump

- Nothing to do! Make sure control signals are set correctly!

Instruction Fetch Unit at the End of Jump

- PC <- PC+<31:28> concat target+<25:0> concat "00"

A Summary of the Control Signals

<table>
<thead>
<tr>
<th>Signal</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegDst</td>
<td>x</td>
<td>Access data register</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>x</td>
<td>Store result to reg</td>
</tr>
<tr>
<td>Mux</td>
<td>x</td>
<td>Multiply or add</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>x</td>
<td>Source of ALU input</td>
</tr>
<tr>
<td>RegWr</td>
<td>0</td>
<td>Write result to reg</td>
</tr>
<tr>
<td>ALUctr&lt;2:0&gt;</td>
<td>x</td>
<td>ALU operation</td>
</tr>
<tr>
<td>ALUop&lt;N:0&gt;</td>
<td>1 00 0 10 0 00 0 00 0 01</td>
<td>ALU operation symbols</td>
</tr>
</tbody>
</table>

The Concept of Local Decoding

Decoding the “func” Field

- In this exercise, ALUop has to be 2 bits wide to represent:
  - (1) “R-type” instructions
  - (2) “I-type” instructions that require the ALU to perform:
    - (3) Add, and (4) Subtract

- To implement the full MIPS ISA, ALUop has to be 3 bits to represent:
  - (1) “R-type” instructions
  - (2) “I-type” instructions that require the ALU to perform:
    - (3) Add, and (4) Subtract, and (5) And (Example: andi)

The Encoding of ALUop

<table>
<thead>
<tr>
<th>ALUop&lt;2:0&gt;</th>
<th>1 00 0 10 0 00 0 00 0 01</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUop&lt;2:0&gt;</td>
<td>10 1010</td>
</tr>
<tr>
<td>ALUop&lt;2:0&gt;</td>
<td>10 0100</td>
</tr>
<tr>
<td>ALUop&lt;2:0&gt;</td>
<td>10 0010</td>
</tr>
<tr>
<td>ALUop&lt;2:0&gt;</td>
<td>10 0000</td>
</tr>
</tbody>
</table>

Decoding the “func” Field

<table>
<thead>
<tr>
<th>Function</th>
<th>ALUop&lt;2:0&gt;</th>
<th>Instruction Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>J-type</td>
<td>10 1010</td>
<td>Jump</td>
</tr>
<tr>
<td>I-type (add)</td>
<td>10 0101</td>
<td>add</td>
</tr>
<tr>
<td>I-type (sub)</td>
<td>10 0011</td>
<td>sub</td>
</tr>
<tr>
<td>I-type (and)</td>
<td>10 0001</td>
<td>and</td>
</tr>
<tr>
<td>R-type</td>
<td>10 0000</td>
<td>ori</td>
</tr>
<tr>
<td>R-type</td>
<td>10 0100</td>
<td>lw</td>
</tr>
<tr>
<td>R-type</td>
<td>10 1000</td>
<td>sw</td>
</tr>
<tr>
<td>R-type</td>
<td>11 1000</td>
<td>beq</td>
</tr>
<tr>
<td>R-type</td>
<td>11 1010</td>
<td>jump</td>
</tr>
<tr>
<td>R-type</td>
<td>11 1100</td>
<td>jump</td>
</tr>
<tr>
<td>R-type</td>
<td>11 1110</td>
<td>jump</td>
</tr>
<tr>
<td>R-type</td>
<td>11 1111</td>
<td>jump</td>
</tr>
</tbody>
</table>

The Single Cycle Datapath during Jump

- Nothing to do! Make sure control signals are set correctly!
The Truth Table for ALUctr

<table>
<thead>
<tr>
<th>ALUop (Symbolic)</th>
<th>func&lt;3:0&gt;</th>
<th>Instruction Op.</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>Or, Add, Subtract</td>
<td></td>
</tr>
</tbody>
</table>

The Logic Equation for ALUctr<2>

\[ ALUctr<2> = \overline{ALUop<2>} \land \overline{ALUop<0>} + \]
\[ ALUop<2> \land \overline{func<2>} \land \overline{func<0>} + \]
\[ ALUop<2> \land \overline{func<3>} \land \overline{func<2>} \land func<1> \land func<0> \]

This makes func<3> a don't care

The Logic Equation for ALUctr<1>

\[ ALUctr<1> = \overline{ALUop<2>} \land \overline{ALUop<1>} + \]
\[ ALUop<2> \land \overline{func<2>} \land \overline{func<0>} \]

The Logic Equation for ALUctr<0>

\[ ALUctr<0> = \overline{ALUop<2>} \land \overline{ALUop<0>} + \]
\[ ALUop<2> \land \overline{func<3>} \land \overline{func<2>} \land \overline{func<1>} \land \overline{func<0>} + \]
\[ ALUop<2> \land \overline{func<3>} \land \overline{func<2>} \land func<1> \land func<0> \]

The ALU Control Block

* ALUctr<2> = \overline{ALUop<2>} \land \overline{ALUop<0>} + \]
* ALUop<2> \land \overline{func<3>} \land \overline{func<2>} \land func<1> \land \overline{func<0>}

The "Truth Table" for the Main Control

<table>
<thead>
<tr>
<th>op</th>
<th>imm</th>
<th>sel</th>
<th>type</th>
<th>B-type</th>
<th>R-type</th>
<th>Shift</th>
<th>add</th>
<th>inc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Truth Table for the Main Control

<table>
<thead>
<tr>
<th>RegDst</th>
<th>ALUOp</th>
<th>ALUOp &lt;2&gt;</th>
<th>ALUOp &lt;1&gt;</th>
<th>ALUOp &lt;0&gt;</th>
<th>ALUOp (Symbolic)</th>
<th>B-type</th>
<th>sel</th>
<th>op</th>
<th>add</th>
<th>inc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Add</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>X</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>0</td>
<td></td>
<td></td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
**The “Truth Table” for RegWrite**

<table>
<thead>
<tr>
<th>op</th>
<th>R-type</th>
<th>ori</th>
<th>lw</th>
<th>beq</th>
<th>jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

RegWrite = (R-type) + ori + lw

\[ \text{RegWrite} = \neg\text{op}_5 \land \neg\text{op}_4 \land \neg\text{op}_3 \land \neg\text{op}_2 \land \neg\text{op}_1 \land \text{op}_0 \text{(R-type)} + \neg\text{op}_5 \land \neg\text{op}_4 \land \text{op}_3 \land \text{op}_2 \land \neg\text{op}_1 \land \text{op}_0 \text{(ori)} + \text{op}_5 \land \neg\text{op}_4 \land \neg\text{op}_3 \land \neg\text{op}_2 \land \text{op}_1 \land \text{op}_0 \text{(lw)} \]

---

**Implementation of the Main Control**

**Putting it All Together: A Single Cycle Processor**

**Worst Case Timing: lw $1$, $2$(offset)**

**Drawback of this Single Cycle Processor**

- Long cycle time:
  - Cycle time must be long enough for the load instruction:
    - PC's Clock-to-Q + Instruction Memory Access Time + Register File Access Time + ALU Delay (address calculation) + Data Memory Access Time + Register File Setup Time + Clock Skew
- Cycle time is much longer than needed for all other instructions

**Summary**

- What's ahead
  - Multicycle processors
  - Memory
  - Input / Output

---

** CPS 104 37**

** CPS 104 38**

** CPS 104 39**

** CPS 104 40**

** CPS 104 41**

** CPS 104 42**