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

- We have everything except control signals (underline)
  - Today's lecture will show you how to generate the control signals

---

The Big Picture: Where are We Now?

- **The Five Classic Components of a Computer**

---

* Today's Topic: Designing the Control for the Single Cycle Datapath
Outline of Today's Lecture

° Recap and Introduction
° 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

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<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>
</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

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]
The Single Cycle Datapath during Add and Subtract

31 26 21 16 11 6 0

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>


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

31 26 21 16 0

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
</table>

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

The Single Cycle Datapath during Or Immediate

31 26 21 16 0

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
</table>

* R[rt] <- R[rs] or ZeroExt[imm16]
The Single Cycle Datapath during Load

* \( R[rt] \leftarrow \text{Data Memory} \ (R[rs] + \text{SignExt}[imm16]) \)

\[
\begin{align*}
\text{RegDst} & = 0 \\
\text{ALUctr} & = \text{Add} \\
\text{MemtoReg} & = 1 \\
\text{Rd} & = R[t] \\
\text{Rs} & = R[s] \\
\text{imm16} & = \text{imm16} \\
\text{Op} & = \text{Op}
\end{align*}
\]

The Single Cycle Datapath during Store

* \( \text{Data Memory} \ (R[rs] + \text{SignExt}[imm16]) \leftarrow R[rt] \)

\[
\begin{align*}
\text{RegDst} & = x \\
\text{ALUctr} & = \text{Add} \\
\text{MemtoReg} & = x \\
\text{Rd} & = R[t] \\
\text{Rs} & = R[s] \\
\text{imm16} & = \text{imm16} \\
\text{Op} & = \text{Op}
\end{align*}
\]
The 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

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”
Step 4: Given Datapath: RTL -> Control

A Summary of Control Signals

<table>
<thead>
<tr>
<th>op</th>
<th>00 0000</th>
<th>00 0110</th>
<th>10 0011</th>
<th>10 1011</th>
<th>00 0100</th>
<th>00 0010</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegDst</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RegWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MemWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Branch</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Jump</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ExtOp</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>ALUctr&lt;2:0&gt;</td>
<td>Add</td>
<td>Subtract</td>
<td>Or</td>
<td>Add</td>
<td>Add</td>
<td>Subtract</td>
</tr>
</tbody>
</table>

A Summary of the Control Signals

<table>
<thead>
<tr>
<th>Instruction&lt;31:0&gt;</th>
<th>ALUctr</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>ExtOp</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemWrite</th>
<th>Branch</th>
<th>Jump</th>
<th>ExtOp</th>
<th>ALUctr&lt;2:0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Add</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtract</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Add</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Subtract</td>
</tr>
</tbody>
</table>

R-type, ALU control selects by func

*The Concept of Local Decoding*
The Encoding of ALUop

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

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

### The Truth Table for ALUctr

<table>
<thead>
<tr>
<th>ALUop (Symbolic)</th>
<th>R-type</th>
<th>ori</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
<th>jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUop&lt;2:0&gt;</td>
<td>“R-type”</td>
<td>Or</td>
<td>Add</td>
<td>Add</td>
<td>Subtract</td>
<td>xxx</td>
</tr>
</tbody>
</table>

### The Logic Equation for ALUctr<2>

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

\[
= \neg X \land Z + X \land A \land B \land C \land D + X \land A \land \neg B \land C \land D
\]

\[
= \neg X \land Z + X \land \neg B \land C \land D
\]
The Logic Equation for ALUctr<1>

<table>
<thead>
<tr>
<th>ALUop bit&lt;2&gt; bit&lt;1&gt; bit&lt;0&gt;</th>
<th>func bit&lt;3&gt; bit&lt;2&gt; bit&lt;1&gt; bit&lt;0&gt;</th>
<th>ALUctr&lt;1&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  0  0</td>
<td>x  x  x  x</td>
<td>1</td>
</tr>
<tr>
<td>0  x  1</td>
<td>x  x  x  x</td>
<td>1</td>
</tr>
<tr>
<td>1  x  x</td>
<td>0  0  0  0</td>
<td>1</td>
</tr>
<tr>
<td>1  x  x</td>
<td>0  0  1  0</td>
<td>1</td>
</tr>
<tr>
<td>1  x  x</td>
<td>1  0  1  0</td>
<td>1</td>
</tr>
</tbody>
</table>

\* ALUctr<1> = \neg ALUop<2> & \neg ALUop<1> + ALUop<2> & \neg \text{func}<2> & \neg \text{func}<0>

The Logic Equation for ALUctr<0>

<table>
<thead>
<tr>
<th>ALUop bit&lt;2&gt; bit&lt;1&gt; bit&lt;0&gt;</th>
<th>func bit&lt;3&gt; bit&lt;2&gt; bit&lt;1&gt; bit&lt;0&gt;</th>
<th>ALUctr&lt;0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  1  x</td>
<td>x  x  x  x</td>
<td>1</td>
</tr>
<tr>
<td>1  x  x</td>
<td>0  1  0  1</td>
<td>1</td>
</tr>
<tr>
<td>1  x  x</td>
<td>1  0  1  0</td>
<td>1</td>
</tr>
</tbody>
</table>

\* ALUctr<0> = \neg ALUop<2> & ALUop<1> + ALUop<2> & \neg \text{func}<3> & \text{func}<2> & \neg \text{func}<1> & \text{func}<0> + ALUop<2> & \text{func}<3> & \text{func}<2> & \text{func}<1> & \neg \text{func}<0>

The ALU Control Block

![Diagram of the ALU Control Block]

\* ALUctr<2> = \neg ALUop<2> & ALUop<0> + ALUop<2> & \neg \text{func}<2> & \text{func}<1> & \neg \text{func}<0>

\* ALUctr<1> = \neg ALUop<2> & \neg ALUop<1> + ALUop<2> & \neg \text{func}<2> & \text{func}<1> & \neg \text{func}<0>

\* ALUctr<0> = \neg ALUop<2> & ALUop<1> + ALUop<2> & \neg \text{func}<3> & \text{func}<2> & \neg \text{func}<1> & \text{func}<0> + ALUop<2> & \text{func}<3> & \text{func}<2> & \text{func}<1> & \neg \text{func}<0>

Step 5: Logic for each control signal

\* nPC_sel <= if (OP == BEQ) then EQUAL else 0

\* ALUsrc <= if (OP == “Rtype”) then “regB” else “immed”

\* ALUctr <= if (OP == “Rtype”) then \text{func} \begin{cases} 
  \text{“OR”} \quad & \text{elseif (OP == ORi) then “add”} \\
  \text{elseif (OP == BEQ) then “sub”}
\end{cases}

\* ExtOp <= __________

\* MemWr <= __________

\* MemtoReg <= __________

\* RegWr: <= __________

\* RegDst: <= __________
Step 5: Logic for each control signal

° nPCSel <= if (OP == BEQ) then EQUAL else 0
° ALUsrc <= if (OP == “Rtype”) then “regB” else “immed”
° ALUctr <= if (OP == “Rtype”) then funct
  elseif (OP == ORi) then “OR”
  elseif (OP == BEQ) then “sub”
  else “add”
° ExtOp <= if (OP == ORi) then “zero” else “sign”
° MemWr <= (OP == Store)
° MemtoReg <= (OP == Load)
° RegWr: <= if ((OP == Store || (OP == BEQ)) then 0 else 1
° RegDst: <= if ((OP == Load || (OP == ORi)) then 0 else 1
Putting it All Together: A Single Cycle Processor

Worst Case Timing (Load)

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

° Single cycle datapath => clocks per instruction=1, clock cycle time => long

° 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

° Control is the hard part

° MIPS makes control easier
  • Instructions same size
  • immediates have same size & location
  • Source registers always in same place
  • Operations always on registers/immediates