MIPS ISA and Single Cycle Datapath

Computer Science 104

Outline of Today’s Lecture

- Homework #5
- The MIPS Instruction Set
- Datapath and timing for Reg-Reg Operations
- Datapath for Logical Operations with Immediate
- Datapath for Load and Store Operations
- Datapath for Branch and Jump Operations
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 register specifiers
  - 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

<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>6 bits</td>
</tr>
</tbody>
</table>

- **I-type**:  
  - op: operation of the instruction
  - rs, rt: the source and destination register specifiers
  - immediate: address offset or immediate value

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- **J-type**:  
  - op: operation of the instruction
  - target address: target address of the jump instruction

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>target address</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

The MIPS Subset (We can't implement them all!)

ADD and subtract:
- add rd, rs, rt
- sub rd, rs, rt

OR immediate:
- ori rt, rs, imm16

LOAD and STORE:
- lw rt, rs, imm16
- sw rt, rs, imm16

BRANCH:
- beq rs, rt, imm16

JUMP:
- j target

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>target address</td>
<td>6 bits</td>
</tr>
</tbody>
</table>
An Abstract View of the Implementation

Clocking Methodology

- All storage elements are clocked by the same clock edge
- Cycle Time $\geq$ CLK-to-Q + Longest Delay Path + Setup + Clock Skew
- Longest delay path = critical path
An Abstract View of the Critical Path

- Register file and ideal memory:
  - The CLK input is a factor ONLY during write operation
  - During read operation, behave as combinational logic:
    - Address valid => Output valid after “access time.”

---

Overview of the Instruction Fetch Unit

- The common RTL operations
  - Fetch the instruction: mem[PC]
  - Update the program counter:
    - Sequential Code: PC <- PC + 4
    - Branch and Jump: PC <- “something else”
RTL: The ADD Instruction

° add rd, rs, rt

• mem[PC] Fetch the instruction from memory

• R[rd] <- R[rs] + R[rt] The ADD operation

• PC <- PC + 4 Calculate the next instruction's address

RTL: The Load Instruction

° lw rt, rs, imm16

• mem[PC] Fetch the instruction from memory

• Address <- R[rs] + SignExt(imm16) Calculate the memory address

• R[rt] <- Mem[Address] Load the data into the register

• PC <- PC + 4 Calculate the next instruction's address
**RTL: The ADD Instruction**

![Add Instruction Diagram](image)

- **add** \( rd, rs, rt \)
  - **mem[PC]**: Fetch the instruction from memory
  - **\( R[rd] \leftarrow R[rs] + R[rt] \)**: The actual operation
  - **\( PC \leftarrow PC + 4 \)**: Calculate the next instruction’s address

**RTL: The Subtract Instruction**

![Subtract Instruction Diagram](image)

- **sub** \( rd, rs, rt \)
  - **mem[PC]**: Fetch the instruction from memory
  - **\( R[rd] \leftarrow R[rs] - R[rt] \)**: The actual operation
  - **\( PC \leftarrow PC + 4 \)**: Calculate the next instruction’s address
Datapath for Register-Register Operations

- \( R[rd] \leftarrow R[rs] \text{ op } R[rt] \)  
  - Example: \( \text{add rd, rs, rt} \)
  - \( Ra, Rb, \text{ and } Rw \) comes from instruction’s \( rs, rt, \text{ and } rd \) fields
  - \( \text{ALUctr and RegWr: control logic after decoding the instruction fields: op and func} \)

### Datapath Diagram

- **RegWr**, **Rd**, **Rs**, **Rt**: 
- **busW**: 32 bits
- **clk**: Clock
- **busA**: 32 bits
- **busB**: 5 bits
- **32-bit Registers**: \( R_r, R_b, R_w \)
- **ALUctr**: 6 bits
- **Result**: 32 bits

#### Instruction Set

- **ori**: \( r_t, r_s, \text{imm16} \)

### RTL: The OR Immediate Instruction

- **ori**: \( r_t, r_s, \text{imm16} \)
  - **mem[PC]**: Fetch the instruction from memory
  - **R[rt] \leftarrow R[rs] \text{ or ZeroExt(imm16)}**: The OR operation
  - **PC \leftarrow PC + 4**: Calculate the next instruction’s address

### Instruction Format

- **ori**
  - op, rs, rt, immediate
Datapath for Logical Operations with Immediate

\[ R_{rt} \leftarrow R_{rs} \text{ op} \text{ZeroExt}[\text{imm16}] \]

**Example:** ori\( \quad rt, rs, \text{imm16} \)

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

RTL: The Load Instruction

\[ \text{l}w\quad rt, rs, \text{imm16} \]

- **mem[PC]**  
Fetch the instruction from memory

- **Address \leftarrow R_{rs} + \text{SignExt}(\text{imm16})**  
Calculate the memory address

- **R_{rt} \leftarrow \text{Mem}[\text{Address}]**  
Load the data into the register

- **PC \leftarrow PC + 4**  
Calculate the next instruction’s address
### Datapath for Load Operations

- \( R[rt] \leftarrow Mem[R[rs] + \text{SignExt}[\text{imm16}]] \)

**Example:** \( lw \ rt, rs, \text{imm16} \)

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

### RTL: The Store Instruction

- \( sw \ rt, rs, \text{imm16} \)

1. **mem[PC]**: Fetch the instruction from memory
2. **Address \leftarrow R[rs] + \text{SignExt}(\text{imm16})**: Calculate the memory address
3. **Mem[Address] \leftarrow R[rt]**: Store the register into memory
4. **PC \leftarrow PC + 4**: Calculate the next instruction’s address
Datapath for Store Operations

- \( \text{Mem}[R[rs] + \text{SignExt}(\text{imm16})] \leftarrow R[rt] \)

Example: \( \text{sw} \ rt, rs, \text{imm16} \)

### RTL: The Branch Instruction

- \( \text{beq} \ rs, rt, \text{imm16} \)

- **mem[PC]**
  
  Fetch the instruction from memory

- **Cond \( \leftarrow \) R[rs] - R[rt]**
  
  Calculate the branch condition

- **if (COND eq 0)**
  
  Calculate the next instruction's address

  - \( \text{PC} \leftarrow \text{PC} + 4 + (\text{SignExt}(\text{imm16}) \times 4) \)

  - **else**

  - \( \text{PC} \leftarrow \text{PC} + 4 \)
Datapath for Branch Operations

- **beq rs, rt, imm16**

  We need to compare Rs and Rt!

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

**ALUctr**

- **Clk**
- **busW**
- **RegWr**
- **Rs**
- **Rt**
- **Rw**
- **Ra**
- **Rb**

**32 32-bit Registers**

**Extender**

- **imm16**
- **busA**
- **busB**

**Mux**

- **32**
- **16**

**ALU**

- **32**

**Zero**

**Next Address Logic**

- **Clk**
- **PC**
- **Branch**
- **To Instruction Memory**

**Binary Arithmetic for the Next Address**

- **In theory,** the PC is a 32-bit byte address into the instruction memory:
  - **Sequential operation:** \( \text{PC}^{31:0} = \text{PC}^{31:0} + 4 \)
  - **Branch operation:** \( \text{PC}^{31:0} = \text{PC}^{31:0} + 4 + \text{SignExt}[\text{Imm16}] \times 4 \)

- **The magic number “4” always comes up because:**
  - The 32-bit PC is a byte address
  - And 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<31:2>:
  - **Sequential operation:** \( \text{PC}^{31:2} = \text{PC}^{31:2} + 1 \)
  - **Branch operation:** \( \text{PC}^{31:2} = \text{PC}^{31:2} + 1 + \text{SignExt}[\text{Imm16}] \)
  - **In either case:** Instruction-Memory-Address = PC<31:2> 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 = $PC_{31:2}$ concat “00”
RTL: The Jump Instruction

- mem[PC] Fetch the instruction from memory
- PC ← PC+4<31:28> concat target<25:0> concat <00> Calculate the next instruction's address

Instruction Fetch Unit

- PC<31:2> ← PC+4<31:28> concat target<25:0>
Putting it All Together: A Single Cycle Datapath

° We have everything except control signals.

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

- **I-type**

  - op: operation of the instruction
  - rs, rt: the source and destination registers specifier
  - immediate: address offset or immediate value

- **J-type**

  - op: operation of the instruction
  - 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

RTL: The ADD Instruction

° 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: \( \text{Instruction} \leftarrow \text{mem}[\text{PC}] \)
  - This is the same for all instructions

**The Single Cycle Datapath during Add and Subtract**

- \( \text{R}[rd] \leftarrow \text{R}[rs] + / - \text{R}[rt] \)

- \( \text{ExtOp} = x \)
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

- \( R[rt] \leftarrow R[rs] \) 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

Instruction Fetch Unit at the End of Branch
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>func</th>
<th>10 0000</th>
<th>10 0010</th>
<th>We Don’t Care :-)</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>00 0000</td>
<td>00 0000</td>
<td>00 1101</td>
</tr>
<tr>
<td>add</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>sub</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ori</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>lw</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>sw</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>beq</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>jump</td>
<td>x</td>
<td>x</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       |
Branch   | 0       | 0       | 0       | 0       | 0       | 0       | 1       |
Jump     | 0       | 0       | 0       | 0       | 0       | 0       | 1       |
ExtOp    | x x     | 0       | 1       | 1       | x x     | x x     | x x     |
ALUctr<2:0> | Add | Subtract | Or | Add | Add | Subtract | xxx |

R-type

I-type

J-type

The Concept of Local Decoding

<table>
<thead>
<tr>
<th>op</th>
<th>00 0000</th>
<th>00 1101</th>
<th>10 0011</th>
<th>10 1011</th>
<th>00 0100</th>
<th>00 0010</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>ori</td>
<td>lw</td>
<td>sw</td>
<td>beq</td>
<td>jump</td>
<td></td>
</tr>
<tr>
<td>RegDst</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
ALUSrc   | 0       | 1       | 1       | 1       | 0       | x       |
MemtoReg| 0       | 0       | 1       | x       | x       | x       |
RegWrite | 1       | 1       | 1       | 0       | 0       | 0       |
MemWrite | 0       | 0       | 0       | 1       | 0       | 0       |
Branch   | 0       | 0       | 0       | 0       | 1       | x       |
Jump     | 0       | 0       | 0       | 0       | 0       | 1       |
ExtOp    | x       | 0       | 1       | 1       | x       | x       |
ALUop<N:0> | “R-type” | Or | Add | Add | Subtract | xxx |

Main Control

aluop

ALU Control

Local

aluctrl

eps 104 41
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)

Decoding the “func” Field

### Table: ALUop (Symbolic) and ALUop<2:0>

<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>

| ALUop<2:0>       | 1 00 | 0 10 | 0 00 | 0 00 | 0 01 | xxx  |

### Table: R-type and Instruction Operation

<table>
<thead>
<tr>
<th>R-type</th>
<th>Instruction Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 0000</td>
<td>add</td>
</tr>
<tr>
<td>10 0010</td>
<td>subtract</td>
</tr>
<tr>
<td>10 0100</td>
<td>and</td>
</tr>
<tr>
<td>10 0101</td>
<td>or</td>
</tr>
<tr>
<td>10 1010</td>
<td>set-on-less-than</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALUctr&lt;2:0&gt;</th>
<th>ALU Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>And</td>
</tr>
<tr>
<td>001</td>
<td>Or</td>
</tr>
<tr>
<td>010</td>
<td>Add</td>
</tr>
<tr>
<td>110</td>
<td>Subtract</td>
</tr>
<tr>
<td>111</td>
<td>Set-on-less-than</td>
</tr>
</tbody>
</table>
## 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>
</tr>
</thead>
<tbody>
<tr>
<td>ALUop&lt;2:0&gt;</td>
<td>1 0 0</td>
<td>0 10</td>
<td>0 00</td>
<td>0 00</td>
<td>0 01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALUop&lt;2:0&gt;</th>
<th>funct&lt;3:0&gt;</th>
<th>Instruction Op.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0000</td>
<td>add</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0010</td>
<td>subtract</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0100</td>
<td>and</td>
</tr>
<tr>
<td>0 1 1</td>
<td>0101</td>
<td>or</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1010</td>
<td>set-on-less-than</td>
</tr>
</tbody>
</table>

### ALUop<2:0>

<table>
<thead>
<tr>
<th>ALUop&lt;2:0&gt;</th>
<th>funct&lt;3:0&gt;</th>
<th>Instruction Op.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0000</td>
<td>add</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0010</td>
<td>subtract</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0100</td>
<td>and</td>
</tr>
<tr>
<td>0 1 1</td>
<td>0101</td>
<td>or</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1010</td>
<td>set-on-less-than</td>
</tr>
</tbody>
</table>

### ALUop<2:0> Operation and ALUctr<2>

<table>
<thead>
<tr>
<th>ALUop&lt;2:0&gt;</th>
<th>funct&lt;3:0&gt;</th>
<th>Instruction Op.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0000</td>
<td>add</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0010</td>
<td>subtract</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0100</td>
<td>and</td>
</tr>
<tr>
<td>0 1 1</td>
<td>0101</td>
<td>or</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1010</td>
<td>set-on-less-than</td>
</tr>
</tbody>
</table>

### ALUctr<2>

<table>
<thead>
<tr>
<th>ALUop&lt;2:0&gt;</th>
<th>funct&lt;3:0&gt;</th>
<th>Instruction Op.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0000</td>
<td>add</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0010</td>
<td>subtract</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0100</td>
<td>and</td>
</tr>
<tr>
<td>0 1 1</td>
<td>0101</td>
<td>or</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1010</td>
<td>set-on-less-than</td>
</tr>
</tbody>
</table>

### ALUctr<2> = !ALUop<2> & ALUop<0> + ALUop<2> & !func<2> & func<1> & !func<0>

This makes func<3> a don’t care.
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 x x x x</td>
<td>0 0 0 0 0 0 0 0</td>
<td>1</td>
</tr>
<tr>
<td>0 x 1 x x x x</td>
<td>0 0 0 0 0 0 0 0</td>
<td>1</td>
</tr>
<tr>
<td>1 x x 0 0 0 0 0 1</td>
<td>0 0 0 0 0 0 0 0</td>
<td>1</td>
</tr>
<tr>
<td>1 x x 1 0 0 1 0 1</td>
<td>0 0 0 0 0 0 0 0</td>
<td>1</td>
</tr>
</tbody>
</table>

° ALUctr<1> = !ALUop<2> & !ALUop<1> + ALUop<2> & !func<2> & !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 x x x x x</td>
<td>0 1 0 1 0 1 0 1</td>
<td>1</td>
</tr>
<tr>
<td>1 x x 0 1 0 1 0 1</td>
<td>0 1 0 1 0 1 0 1</td>
<td>1</td>
</tr>
<tr>
<td>1 x x 1 0 1 0 1 1</td>
<td>0 1 0 1 0 1 0 1</td>
<td>1</td>
</tr>
</tbody>
</table>

° ALUctr<0> = !ALUop<2> & ALUop<0> + ALUop<2> & !func<3> & func<2> & !func<1> & func<0> + ALUop<2> & func<3> & !func<2> & func<1> & !func<0>
The ALU Control Block

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

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

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

The “Truth Table” for the Main Control (rotated)

<table>
<thead>
<tr>
<th>op</th>
<th>00 0000</th>
<th>00 1101</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>R-type</td>
<td>ori</td>
<td>lw</td>
<td>sw</td>
<td>beq</td>
<td>jump</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>RegWrite</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MemWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</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>x</td>
</tr>
<tr>
<td>ExtOp</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ALUop (Symbolic)</td>
<td>“R-type”</td>
<td>Or</td>
<td>Add</td>
<td>Add</td>
<td>Subtract</td>
<td>xxx</td>
</tr>
<tr>
<td>ALUop &lt;2&gt;</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>ALUop &lt;1&gt;</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>ALUop &lt;0&gt;</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
</tr>
</tbody>
</table>
The "Truth Table" for RegWrite

<table>
<thead>
<tr>
<th>op</th>
<th>00 0000</th>
<th>00 1101</th>
<th>10 0011</th>
<th>10 1011</th>
<th>00 0100</th>
<th>00 0010</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ori</td>
<td>lw</td>
<td>sw</td>
<td>beq</td>
<td>jump</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{RegWrite} = \text{R-type} + \text{ori} + \text{lw} \]

- \[ \text{RegWrite} = \text{R-type} \]
- \[ \text{RegWrite} = \text{ori} \]
- \[ \text{RegWrite} = \text{lw} \]

Implementation of the Main Control
Putting it All Together: A Single Cycle Processor

Worst Case Timing: \texttt{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
  • Pipelined processors
  • Memory
  • Input / Output