Chapter 5
The Processor - Part II
Control
Control Unit

- **Functions:**
  - Select operations to be performed (ALU, read/write, etc.)
  - Control data flow (multiplexor inputs)

- **Major components:**
  - ALU control
    - ALU's operation is based on instruction type and function code
    - Will be designed first
  - Other controls

- **Input:**
  - Information comes from the 32 bits of the instruction

- **Output:**
  - Control signals
### Review: MIPS Instruction Format

#### R-type
```
<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>
</tbody>
</table>
```
- **op**: 6 bits
- **rs**: 5 bits
- **rt**: 5 bits
- **rd**: 5 bits
- **shamt**: 5 bits
- **funct**: 6 bits

#### I-type
```
<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></td>
</tr>
</tbody>
</table>
```
- **op**: 6 bits
- **rs**: 5 bits
- **rt**: 5 bits
- **immediate**: 16 bits

#### J-type
```
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>target address</td>
<td></td>
</tr>
</tbody>
</table>
```
- **op**: 6 bits
- **target address**: 26 bits
Observations on Instruction Format

- The *opcode* field is always in bits 31:26. will be referred to as (Op[5:0])
- For R-type, branch equal, and store instructions, the two registers to read are always *rs* and *rt* in bits 25:21 and 20:16 respectively
- For load and store instructions, the base register, *rs*, is always in bits 25:21
- For branch equal, load, and store, the 16-bit offset is always in positions 15:0
- The destination register is in one of two places
  - For load, the destination register (*rt*) is in position 20:16
  - For R-type instructions, destination register (*rd*) is in positions 15:11
  - => A MUX is needed to select which field of the instruction is used to indicate the destination register
Review: 1-Bit ALU Control Signals

- Supporting and, or, add, subtract, and nor
  - Operation
    - For and, or and add
      - 00 and
      - 01 or
      - 10 add
    - Binvert (invert the b-input)
      - For subtract and Nor = 1; otherwise = 0
        - 0 \( b \)
        - 1 \( \text{not}(b) \)
    - Ainvert (invert the a-input)
      - For nor = 1; otherwise = 0
        - 0 \( a \)
        - 1 \( \text{not}(a) \)
  - Summary

<table>
<thead>
<tr>
<th>Ainvert</th>
<th>Binvert</th>
<th>Operation</th>
<th>Combined</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0000</td>
<td>AND</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0001</td>
<td>OR</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0010</td>
<td>add</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>10</td>
<td>0110</td>
<td>subtract</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>0111</td>
<td>set-on-less-than</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>00</td>
<td>1100</td>
<td>NOR</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01</td>
<td>1101</td>
<td>set-on-less-than</td>
</tr>
</tbody>
</table>
ALU Control

Summary

<table>
<thead>
<tr>
<th>ALU control</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>AND</td>
</tr>
<tr>
<td>0001</td>
<td>OR</td>
</tr>
<tr>
<td>0010</td>
<td>add</td>
</tr>
<tr>
<td>0110</td>
<td>subtract</td>
</tr>
<tr>
<td>0111</td>
<td>set-on-less-than</td>
</tr>
<tr>
<td>1100</td>
<td>NOR</td>
</tr>
</tbody>
</table>

The ALU has 4 control inputs
- Allows 16 possible combinations
- Only 6 combinations are used.
- The rest could be used as don’t-care in minimization
ALU Control

Remember:

- ALU is needed for all instruction categories
  - *lw* and *sw* (I-Format):
    - Compute memory address
  - Arithmetic/logic (R-Format):
    - Perform arithmetic / logic operation
  - Branch(*beq*) (I-Format):
    - Subtract registers

- We need to find ALU control signals from the information in the instruction
ALU Control

How ALU control bits are set

- ALUOp = **00** or **01**
  - They are of I-type format
  - Depend on "op" field & does not depend on "funct" field

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ALUOp Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>10011</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
</tr>
</tbody>
</table>

  => Don’t care’s are used XXXXXX for funct field

- ALUOp code = **10**
  - Are of R-type instructions
  - Depend on "funct" field
  => funct code is used to set the ALU control input
### ALU Control

- How the ALU control bits are set depends on ALUOp control bits and the different function codes for R-type instructions.
- See fig 5.12, p. 302

<table>
<thead>
<tr>
<th>Instruction Opcode</th>
<th>ALUOp</th>
<th>Instruction operation</th>
<th>Funct field</th>
<th>Desired ALU action</th>
<th>ALU control input</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>00</td>
<td>load word</td>
<td>XXXXXXX</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>sw</td>
<td>00</td>
<td>store word</td>
<td>XXXXXXX</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>Branch equal</td>
<td>01</td>
<td>branch equal</td>
<td>XXXXXXX</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>add</td>
<td>100000</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>subtract</td>
<td>100010</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>AND</td>
<td>100100</td>
<td>and</td>
<td>0000</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>OR</td>
<td>100101</td>
<td>or</td>
<td>0001</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>set on less than</td>
<td>101010</td>
<td>set on less than</td>
<td>0111</td>
</tr>
<tr>
<td>R-type</td>
<td>11</td>
<td>nor</td>
<td>100111</td>
<td>nor</td>
<td>1100</td>
</tr>
</tbody>
</table>
ALU Control

- Truth table for the 4 ALU control bits
  - See Fig 5.13, p. 302
  - Once the truth table is constructed, it can be optimized and turned into gates

![Truth table for the 4 ALU control bits](image)

**FIGURE 5.13** The truth table for the three ALU control bits (called Operation). The inputs are the ALUOp and function code field. Only the entries for which the ALU control is asserted are shown. Some don’t-care entries have been added. For example, the ALUOp does not use the encoding 11, so that truth table can contain entries 1X and XX, rather than 10 and 01. Also, when the function field is used, the first two bits (F5 and F4) of these instructions are always 10, so they are don’t-care terms and are replaced with XX in the truth table.
Truth Tables for ALU Control Bits

- We need to find the logical functions for O3, O2, O1, and O0

<table>
<thead>
<tr>
<th>ALUOp</th>
<th>ALUOp0</th>
<th>F5</th>
<th>F4</th>
<th>F3</th>
<th>F2</th>
<th>F1</th>
<th>F0</th>
<th>O3</th>
<th>O2</th>
<th>O1</th>
<th>O0</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>lw/sw</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>beq</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</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>
<td>add</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>sub</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>and</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>or</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>sll</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>nor</td>
</tr>
</tbody>
</table>
Truth Tables for ALU Control Bits

For Operation $O3 = 1$

<table>
<thead>
<tr>
<th>ALUOp</th>
<th>ALUOp0</th>
<th>F5</th>
<th>F4</th>
<th>F3</th>
<th>F2</th>
<th>F1</th>
<th>F0</th>
<th>O3</th>
<th>O2</th>
<th>O1</th>
<th>O0</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>lw/sw</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>beq</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>add</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>sub</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>and</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>or</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>sll</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>nor</td>
</tr>
</tbody>
</table>

- $O3 = ALUOp1 \cdot ALUOp0 \cdot F3' \cdot F2 \cdot F1 \cdot F0$
Truth Tables for ALU Control Bits

For Operation O2 = 1

O2 = ALUOp1.ALUOp0
Truth Tables for ALU Control Bits

For Operation O1 = 1

- O1 = ALUOp1’. ALUOp0’ + ALOP1.ALUOp0.F2’.F0’
### Truth Tables for ALU Control Bits

- For Operation $O_0 = 1$

<table>
<thead>
<tr>
<th>ALUOp</th>
<th>Funct field</th>
<th>Operation</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUOp1</td>
<td>ALUOp0</td>
<td>F5</td>
<td>F4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALUOp</th>
<th>Funct field</th>
<th>Operation</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUOp1</td>
<td>ALUOp0</td>
<td>F5</td>
<td>F4</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

- $O_0 = \text{ALUOp1}$
ALU Control: Logic Circuit

- Based on “funct” code & ALUOp
- Details in Appendix C2 (Have some errors/incomplete)
- \( O_3 = \text{ALUOp}_1 \cdot \text{ALUOp}_0 \cdot F_3'.F_2.F_1.F_0 \)
- \( O_2 = \text{ALUOp}_1.\text{ALUOp}_0 \cdot F_3'.F_1 \)
- \( O_1 = \text{ALUOp}_1 .F_2'.F_0' + \text{ALUOp}_1' \)
- \( O_0 = \text{ALUOp}_1 \)
- Exercise:
  - Draw the correct logic diagram

Exercise:
- Draw the correct logic diagram
ALU Control

- The datapath with necessary multiplexors and all control lines
ALU Control

- 9 control lines
  - 2 for ALUOp
    - 00 for load/store
    - 10 for R-Format
    - 01 for other operations
  - RegDst
  - RegWrite
  - ALUSrc
  - PCSrc
  - MemRead
  - MemWrite
  - MemToReg
- All signals except PCSrc are set from the opcode field
- PCSrc is set when the code is for a branch instruction and Zero signal is set
  - To generate PCSrc signal, we use an AND gate with the “zero” signal from ALU
- See fig. 5.16, p. 306 for the effect of the control signals when they are asserted or de-asserted respectively
ALU Control

- Effect of the control signals when they are asserted or de-asserted respectively
- See fig. 5.16, p. 306

<table>
<thead>
<tr>
<th>Signal name</th>
<th>If de-asserted (0)</th>
<th>If asserted (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegDst</td>
<td>Destination register &lt;= rt field (20-16)</td>
<td>Destination register &lt;= rd field (15-11)</td>
</tr>
<tr>
<td>RegWrite</td>
<td>None</td>
<td>Write data input =&gt; Write register</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>Reg Data2 =&gt; 2nd ALU operand</td>
<td>Sign-extnd 16-bits of instruction =&gt; 2nd ALU operand</td>
</tr>
<tr>
<td>PCSrc</td>
<td>PC &lt;= PC + 4 (from adder)</td>
<td>PC &lt;= Branch target (from adder)</td>
</tr>
<tr>
<td>MemRead</td>
<td>None</td>
<td>Mem[Address] =&gt; to Read data output</td>
</tr>
<tr>
<td>MemWrite</td>
<td>None</td>
<td>Mem[Address] &lt;= value on Write data input</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>Value to Write data input comes from AL</td>
<td>Value to Write data input comes from data memory</td>
</tr>
</tbody>
</table>
ALU Control

- The simple datapath with the control unit
Main Control Unit

Input:
- 6-bit opcode field from the instruction

Output:
- 9 signals
  - 3 signals for multiplexors
    - RegDest
    - MemtoReg
    - ALUSrc
  - 3 signals to control reads & writes in register file & data memory
    - RegWrite
    - MemRead
    - MemWrite
  - 1 signal to control whether to Branch or not
    - Branch
  - 2 signals for ALU
    - ALUOp
Main Control Unit

- Mapping the Functions to Gates
- See Fig C.2.5, Appendix C, p. c-8

Input (6-bits Op code)

<table>
<thead>
<tr>
<th>Op5</th>
<th>Op4</th>
<th>Op3</th>
<th>Op2</th>
<th>Op1</th>
<th>Op0</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>100011</td>
<td>101011</td>
<td>000100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output (Control signals)

- RegDst
- ALUSrc
- MemtoReg
- RegWrite
- MemRead
- MemWrite
- Branch
- ALUOp1
- ALUOp0
Main Control Unit

- Control lines needed by each instruction
  - R-Format \((\text{add, sub, AND, OR, & slt})\)
    - Source register: \(rs & rt\)
    - Destination register: \(rd\)
    - Writes a register (RegWrite = 1) but neither reads nor writes data memory
    - ALUOp for R-Type format = 10 indicating that ALU control should be generated from function field
  - When Branch:
    - Control signal = 0, \(PC <= PC + 4\);
    - Otherwise it is replaced by Branch target
  - For \(lw\)
    - MemRead = 1,
  - for \(sw\)
    - MemWrite = 1
  - See Fig 5.18, p. 308

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDest</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Banch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-Format</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(lw)</td>
<td>0</td>
<td>1</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>(sw)</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(beq)</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Datapath Operation for R-Format (add)

Add $t1, $t2, $t3

- **Step 1:**
  - Fetch instruction from instruction memory
  - Increment program counter
- **Step 2:**
  - Decode result is *Add* operation
  - Read two registers $t2$ & $t3$ from register file
- **Step 3:**
  - ALU operates on data, using "func*" field code to generate the ALU function
- **Step 4:**
  - Idle
- **Step 5:**
  - Write result of step 3 from ALU into register $t1$

**Note:**
- Step 1 is the same for all instructions
Datapath Operation for R-Format \((\text{add})\)

\textit{Add }$t_1$, $t_2$, $t_3$\textit{ (Fig. 5.19, p. 309)
Datapath Operation for R-Format *\( (add) \)*

- **Step:** Fetch instruction & increment PC
Datapath Operation for R-Format (add)

- **Step2**: Read registers from register file
  - Main control unit uses the opcode to determine the control line setting
  - These units become active in addition to the units during the instruction fetch step
Datapath Operation for R-Format *(add)*

- **Step 3:** Perform ALU operation
  - Opcode fields are also used to set control lines

![Datapath Diagram](image-url)
Datapath Operation for R-Format *(add)*

- **Step4**: idle
Datapath Operation for R-Format (add)

- Step5: Write result from ALU into register
Datapath Operation for I-Format \( (\textit{slti}) \)

\[ \textit{slti} \ $t1, \ $t2, \ 33 \]

- **Step 1:**
  - Fetch instruction from instruction memory
  - Increment program counter
- **Step 2:**
  - Decode result is \( \textit{slti} \) operation
  - Read two registers \( \$t2 \) from register file
- **Step 3:**
  - ALU operates on data, using "\textit{funct}" field code to generate the ALU function
  - Compare value retrieved with immediate operand value 33.
- **Step 4:**
  - Idle
- **Step 5:**
  - Write result of step 3 from ALU into register \( \$t1 \)
Datapath Operation for I-Format \((lw)\)

\(lw\ $t1,\ offset($t2)\)

- **Step 1:**
  - Fetch instruction from instruction memory
  - Increment program counter
- **Step 2:**
  - Decode result is \(lw\) operation
  - Read register \($t2\) from register file
- **Step 3:**
  - ALU operates on data to build an address
  - Compute sum of \($t2\) & sign-extended 16-bits (offset) of instruction
- **Step 4:**
  - Retrieve data located in the calculate address from memory
- **Step 5:**
  - Write memory contents into destination register \($t1\)
Datapath Operation for I-Format (*lw*)

### lw $t1, offset($t2) (Fig 5.20, p. 310)

- Exercise: Highlight the steps!
Datapath Operation for I-Format \( (sw) \)

\[ sw \, t1, \, 33(t2) \]

- **Step 1:**
  - Fetch instruction from instruction memory
  - Increment program counter
- **Step 2:**
  - Decode result is \( SW \) operation
  - Read register contents of \( t1 \) and \( t2 \) from register file
- **Step 3:**
  - ALU operates on data to build an address
  - Compute sum of \( t2 \) & sign-extended 16-bits (offset) of instruction
- **Step 4:**
  - Write data into the computed memory address
- **Step 5:**
  - Idle
Datapath Operation for I-Format \((sw)\)

Exercise: Highlight the steps!

\textit{sw} \$t1, offset(\$t2)
Datapath Operation for I-Format (*beq*)

*beq* $t1, $t2, offset*  (Fig. 5.21, p. 311)

- **Step 1:**
  - Fetch instruction from instruction memory
  - Increment PC
- **Step 2:**
  - Decode instruction result in *beq*
  - Read contents of $t1, $t2 registers from register file
- **Step 3:**
  - ALU subtract the two values from registers
  - Add PC+4 to sign-extended lower 16-bits of offset
- **Step 4:**
  - Idle
- **Step 5:**
  - The zero result from ALU is used to decide which adder result to store into PC
Datapath Operation for I-Format \textit{(beq)}

\textit{beq} \$t1, \$t2, offset

- Exercise: Highlight the steps!
Datapath Operation for J-Format \( (j) \)

- Similar to branch, but computes PC differently and no comparison needed

**\( j \) label**

- **Step 1:**
  - Fetch instruction from instruction memory
  - Increment PC by 4
- **Step 2:**
  - Decode instruction result is \( j \)
  - Retrieve value of jump target (label)
- **Step 3:**
  - Shift the label 26 bits left by 2 bits (to get byte count instead of word count)
  - Concatenate the upper 4 bits of PC +4 as the high-order bits (to get the complete memory address)
- **Step 4:**
  - Idle
- **Step 5:**
  - Store the calculated address into PC
Lower-order 2 bits of a jump address are always 00
Single Cycle Implementation

Exercise:
Assuming we have the following instruction mix:
- Load 25%
- Store 10%
- ALU instructions 45%
- Branch 15%
- Jump 5%

Calculate cycle time assuming negligible delays except for the following:
- memory (200 ps) (ps = Pico Second)
- ALU and adders (100 ps)
- Register file access (50 ps)

Note: For single-cycle instruction, CPI = 1

Answer:
- CPU execution time = Instruction count x CPI x Clock cycle time
  = Instruction count x 1 x Clock cycle time
  = Instruction count x Clock cycle time

(Complete answer in pp. 315-317)
Single Cycle Problems

- Single cycle instruction is not used in modern computers
- Disadvantages:
  - Inefficient in both performance and HW cost
  - Clock cycle must have the same length for every instruction
  - Clock cycle determined by the longest possible path in the machine
    - Load instruction has 5 functional units in series
  - Several other instruction classes could fit in shorter clock cycle
  - For more complicated instruction like floating point it will be even harder
  - Violates the key design principle “Make common case faster”
  - Functional units must be duplicated, since each can be used only once per cycle
Single Cycle Problems

Solution:
- Use a “smaller” cycle time
- Have different instructions take different numbers of cycles
- “Multi-cycle” datapath
Multicycle Implementation

- Each instruction is broken into a series of steps corresponding to the functional unit operations needed
- Each step executes in one clock cycle
- Advantages:
  - Allows functional units to be used more than once per instruction
  - Help reduce HW cost
Multicycle Implementation

- Major difference from single cycle datapath:
  - Single memory unit used for both instruction and data
  - Single ALU instead of three
  - One or more registers are added after every major functional unit to hold the output of that unit until the value is used in subsequent clock cycle
- Added registers
  - **Instruction register (IR)**
    - Saves output from memory for instruction read
  - **Memory data register (MDR)**
    - Saves output from memory for data read
  - **A and B registers**
    - Hold the register operands read from register file
  - **ALUOut register**
    - Holds the output from ALU
Multicycle Implementation

- Multicycle datapath handling basic instructions
  - Supports normal PC increment
  - One MUX is added and one is increased in size
  - A different set of control signals is needed
Multicycle Implementation

- Multicycle datapath including control signals
Multicycle Implementation

- Multicycle datapath supporting jump and branch instructions
Breaking Instructions

- Each step will contain at most
  - One ALU operation
  - One register file access
  - One memory access
- At the end of each cycle, data values needed for subsequent cycle must be stored into a register
- Since the design is edge triggered, current value of a register can be read during that clock cycle period until the next edge
- All operations listed in one step occur in parallel
- Successive steps occur in a series of different clock cycles
- Each MIPS instruction will need 3 to 5 steps
Breaking Instructions

Steps:
1. Instruction fetch
2. Instruction decode & register fetch
3. Execution, memory address computation, or branch completion
4. Memory access or R-type instruction completion
5. Memory read completion

See Fig 5.30, p. 329 for a summary of the steps
Breaking Instructions

1. Instruction fetch
   - Action taken (in Verilog terminology)
     - IR <= Memory[PC]
     - PC <= PC + 4
   - Control signals required
     - MemRead asserted
     - IRWrite asserted
     - IorD set to 0 (to select PC)
     - ALUSrcA set to 0
     - ALUSrcB set to 01 (to send 4 to ALU)
     - ALUOp set to 00 (choose addition)
     - PCSource set to 00
     - PCWrite asserted
2. Instruction decode & register fetch
   - Action taken
     - A <= Reg[IR[25:21]];  
     - B <= Reg[IR[20:16]];  
     - ALUOut <= PC + (sign-extend (IR[15:0]) << 2);
   - Control signals required
     - ALUSrcA set to 0
     - ALUSrcB set to 11 (sign extend and shift offset sent to ALU)
     - ALUOp set to 00 (choose addition)
Breaking Instructions

3. Execution, memory address computation, or branch completion
   - Determined by the instruction class
     - Memory reference instruction
       - Action taken
         - ALUOut <= A + sign-extend (IR[15:0]);
       - Control signals required
         - ALUSrcA set to 1
         - ALUSrcB set to 10 (sign extend and shift offset sent to ALU)
         - ALUOp set to 00 (choose addition)
     - Arithmetic-logical
       - Action taken
         - ALUOut <= A op B;
       - Control signals required
         - ALUSrcA set to 1
         - ALUSrcB set to 00
         - ALUOp set to 10(determined by funct field)
Breaking Instructions

3. Execution, memory address computation, or branch completion
   - Determined by the instruction class
     - Branch
       - Action taken
         - if (A == B) PC <= ALUOut;
       - Control signals required
         - ALUSrcA set to 1
         - ALUSrcB set to 00
         - ALUOp set to 01 (for subtract)
         - PCWrite asserted
         - PCSource set to 01
     - Jump
       - Action taken
         - PC <= {PC[31:28], IR[25:0], 2'b00});
       - Control signals required
         - PCWrite asserted
         - PCSource set to direct the jump address to the PC
4. Memory access or R-type instruction completion
   - Determined by the instruction class
     - Memory reference (load or store)
       - Action taken
         - MDR $\leftarrow$ Memory[ALUOut];
         - Memory [ALUOut] $\leftarrow$ B;
       - Control signals required
         - MemRead asserted (for load)
         - MemWrite asserted (for store)
         - IorD set to 1
     - Arithmetic-logical (R-type)
       - Action taken
         - Reg [IR [15:11]] $\leftarrow$ ALUOut;
       - Control signals required
         - MemtoReg set to 0
         - RegWrite asserted
Breaking Instructions

5. Memory read completion

- Load
  - Action taken
    - Reg [IR[20:16]] <= MDR;
  - Control signals required
    - MemtoReg set to 1 (to write result from memory)
    - RegWrite asserted (to cause a write)
    - RegDst set to 0 (to choose rt bits 20:16)
Finite State Machine

- Can be used to specify multi-cycle control
- A sequential logic function consisting of
  - A set of inputs
  - A set of outputs
  - A next state function that maps the current state and the input to a new state
  - An output function that maps the current state and the inputs to a set of asserted outputs
Finite State Machine

- The high-level view of the finite state machine control

Diagram:
- Start
- Instruction fetch/decode and register fetch (Figure 5.32)
  - Memory access instructions (Figure 5.33)
  - R-type instructions (Figure 5.34)
  - Branch instruction (Figure 5.35)
  - Jump instruction (Figure 5.36)
Finite State Machine

Instruction fetch and decode portion of every instruction is identical
Finite State Machine

- Controlling memory reference instructions
Finite State Machine

- R-type instructions

![Finite State Machine Diagram](image-url)
Finite State Machine

- Branch instructions
Finite State Machine

- Jump instructions

![Diagram of Finite State Machine with jump instructions](image)
Finite State Machine

- Finite state machine controllers are typically implemented using combinational logic and a register to hold current state.
Finite State Machine

- Complete finite state machine control for the datapath
Homework Assignment

- Textbook, p. 354,
  - In class:
    - Problems 5.1, 5.2
  - Homework:
    - Problems#: 5.8, 5.9, 5.10, & 5.32