Chapter 5  The processor: Datapath and control

Qing-song Shi

http://10.214.26.103  Email: zjsqs@zju.edu.cn
Chapter 5
The processor: Datapath and control
Chapter Five
The processor: Datapath and control

5.1 Introduction
5.2 Logic Design Conventions (skip)
5.3 Building a datapath
5.4 A Simple Implementation Scheme
5.5 A Multicycle Implementation
5.5 Microprogramming
5.6 Exception
What is the MIPS?

Microprocessor without Interlocked Pipeline Stages
5.1 Introduction

- We’ll look at an implementation of the MIPS™
- Simplified to contain only:
  - memory-reference instructions: \texttt{lw, sw}
  - arithmetic-logical instructions: \texttt{add, sub, and, or, slt}
  - control flow instructions: \texttt{beq, j}
- An Overview of the implementation
  - For every instruction, the first two steps are identical
    1. Fetch the instruction from the memory
    2. Decode and read the registers
  - Next steps depend on the instruction class
    - Memory-reference
    - Arithmetic-logical branches

What are steps?
How many FUN.?
Computer Organization

- CPU
- Memory
- I/O interface

Control unit

Datapath
- Path: multiplexors
- ALU
- Registers

......
An abstract view of the implementation of MIPS
Chapter Five
The processor: Datapath and control

5.1 Introduction
5.2 Logic Design Conventions (skip)
5.3 Building a datapath
5.4 A Simple Implementation Scheme
5.5 A Multicycle Implementation
5.5 Microprogramming
5.6 Exception
State Elements

- Unclocked vs. Clocked
- Clocks used in synchronous logic
  - when should an element that contains state be updated?
Our Implementation

- An edge triggered methodology
- Typical execution:
  - read contents of some state elements,
  - send values through some combinational logic
  - write results to one or more state elements
D-Latch for state

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
<th>S</th>
<th>R</th>
<th>Q(t + 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>hold</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Q=0: reset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Q=1: reset</td>
</tr>
</tbody>
</table>

D-Latch FUN. table

C | D | Q(t + 1) |
---|---|---------|
0 | X | hold    |
1 | 0 | Q=0: reset |
1 | 1 | Q=1: set |

D-Latch symbol
Chapter Five
The processor: Datapath and control

5.1 Introduction
5.2 Logic Design Conventions (skip)
5.3 Building a datapath
5.4 A Simple Implementation Scheme
5.5 A Multicycle Implementation
5.5 Microprogramming
5.6 Exception
The datapath there are ……

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 register</td>
<td>$s0-$s7,$t0-$t9, $zero,$a0-$a3, $v0-$v1</td>
<td>Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. $gp(28) is the global pointer, $sp(29) is the stack pointer, $fp(30) is the frame pointer, and $ra(31) is the return address.</td>
</tr>
<tr>
<td>30 Word Addresses signals line</td>
<td>Memory[0], Memory[4] , …… , Memory[4294967292]</td>
<td>Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers, such as those saved on procedure calls.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Register no.</th>
<th>Usage</th>
<th>Preserved on call</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>The constant value 0</td>
<td>n,.a.</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>Values for results and expression evaluation</td>
<td>no</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>Arguments</td>
<td>no</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>Temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>Saved</td>
<td>yes</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>More temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>Global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>Stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>Frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>Return address</td>
<td>yes</td>
</tr>
</tbody>
</table>
# ALU OP for MIPS machine language

<table>
<thead>
<tr>
<th>Name</th>
<th>Format</th>
<th>Example</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>R</td>
<td>0 18 19 17 0 32</td>
<td>add $s1, $s2, $s3</td>
</tr>
<tr>
<td>sub</td>
<td>R</td>
<td>0 18 19 17 0 34</td>
<td>sub $s1, $s2, $s3</td>
</tr>
<tr>
<td>lw</td>
<td>I</td>
<td>35 18 17 100</td>
<td>lw $s1, 100($s2)</td>
</tr>
<tr>
<td>sw</td>
<td>I</td>
<td>43 18 17 100</td>
<td>sw $s1, 100($s2)</td>
</tr>
<tr>
<td>and</td>
<td>R</td>
<td>0 18 19 17 0 36</td>
<td>and $s1, $s2, $s3</td>
</tr>
<tr>
<td>or</td>
<td>R</td>
<td>0 18 19 17 0 37</td>
<td>or $s1, $s2, $s3</td>
</tr>
<tr>
<td>nor</td>
<td>R</td>
<td>0 18 19 17 0 39</td>
<td>nor $s1, $s2, $s3</td>
</tr>
<tr>
<td>addi</td>
<td>I</td>
<td>12 18 17 100</td>
<td>addi $s1, $s2,100</td>
</tr>
<tr>
<td>ori</td>
<td>I</td>
<td>13 18 17 100</td>
<td>ori $s1, $s2,100</td>
</tr>
<tr>
<td>beq</td>
<td>I</td>
<td>4 17 18 25</td>
<td>beq $s1, $s2,100</td>
</tr>
<tr>
<td>bne</td>
<td>I</td>
<td>5 17 18 25</td>
<td>bne $s1, $s2,100</td>
</tr>
<tr>
<td>slt</td>
<td>R</td>
<td>0 18 19 17 0 42</td>
<td>slt $s1, $s2,$s3</td>
</tr>
<tr>
<td>j</td>
<td>J</td>
<td>2 2500</td>
<td>j 10000(see section 2.9)</td>
</tr>
<tr>
<td>jr</td>
<td>R</td>
<td>0 31 0 0 0 8</td>
<td>j Sra</td>
</tr>
<tr>
<td>jal</td>
<td>J</td>
<td>3 2500</td>
<td>jar 10000(see section 2.9)</td>
</tr>
</tbody>
</table>

Field size:
- 6bits
- 5bits

**R-format**
- R op rs rt rd shamt funct
- Arithmetic instruction format

**i-format**
- I op rs rt address
- Data transfer ,branch format

All MIPS instruction 32 bits
Instruction execution in MIPS

- **Fetch**:  
  - Take instructions from the instruction memory  
  - Modify PC to point the next instruction

- **Instruction decoding & Read Operand**:  
  - Will be translated into machine control command  
  - Reading Register Operands, whether or not to use

- **Executive Control**:  
  - Control the implementation of the corresponding ALU operation

- **Memory access**:  
  - Write or Read data from memory  
  - Only LW/SW

- **Write results to register**:  
  - If it is R-type instructions, ALU results are written to Rd  
  - If it is I-type instructions, Results are written to Rt
Instruction fetching three elements

How to connect?  Who?

Instruction memory  Program counter  Adder
Instruction fetching unit

PC → Read address

Instruction memory

Instruction

Add

4

CPU
More Implementation Details

Abstract / Simplified View:

- Registers
- Register #
- Data
- Memory
- Address
- Instruction
- ALU
- Data memory
- Data
Register File--Built using D flip-flops

- Read

Output from the register

5 bits
Read register number 1
Read register number 2
Write register

5 bits
Read data 1
Read data 2
Write data

32 bits
Reg. address

Data output

Read register number 1
Read register number 2
Read data 1
Read data 2
Reg. address

Write

Register file

Register 0
Register 1
Register n–1
Register n
Register File

- Write
  - Written to the register

Write signals
 rd or rt
  5 bits
Reg. address
  32 bits

Write signals: 

Register number: 
  0, 1, 2, ..., n - 1
  5 bits

Reg. address: 
  32 bits

Register data: 
  C, D

Write signals connect to:

- Register 0
- Register 1
- Register n - 1
- Register n
Register files

- Foundation element of Computer（Part of Datapath）
- Aggregation of many Registers
- Register address, Control signals: Read/Write
Module regs(clk, rst, reg_Rd_addr_A, reg_Rt_addr_B, reg_Wt_addr, wdata, we, rdata_A, rdata_B);

input clk, rst, we;
input [4:0] reg_Rd_addr_A, reg_Rt_addr_B, reg_Wt_addr;
input [31:0] wdata;
output [31:0] rdata_A, rdata_B;
reg [31:0] register [1:31]; // r1 - r31
integer i;

assign rdata_A = (reg_Rd_addr_A == 0)? 0 : register[reg_Rd_addr_A]; // read
assign rdata_B = (reg_Rt_addr_B == 0)? 0 : register[reg_Rt_addr_B]; // read
always @(posedge clk or posedge rst)
begin
  if (rst==1) begin // reset
    for (i=1; i<32; i=i+1)
      register[i] <= 0;
  end
  else begin
    if ((reg_Wt_addr != 0) && (we == 1)) // write
      register[reg_Wt_addr] <= wdata;
  end
end
endmodule
Path Built using Multiplexer

- R-type instruction Datapath
- I-type instruction Datapath
  - For ALU
  - For memory
  - For branch
- J-type instruction Datapath
  - For Jump

- First, Look at the data flow within instruction execution
R type Instruction & Data stream

- **op(6)**
- **rs(5)**
- **rt(5)**
- **rd(5)**
- **shamt**
- **func(6)**

### Registers
- Read reg. address1
- Read reg. address2
- Write reg. address
- Read data1
- Read data2
- Write data

### ALU
- ALU operation
- ALU result
- Zero

### Control
- RegWrite

### Function Table

<table>
<thead>
<tr>
<th>Bnega</th>
<th>op</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>and</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>Or</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>Add</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>Sub</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>Slt</td>
</tr>
</tbody>
</table>
I type Instruction & Data stream

- lw $t0, 200($s2)
- if $s2=1000, it will load word in element number 1200 to $t0
I type Instruction & Data stream of *beq*

- **op(6)**
- **rs(5)**
- **rt(5)**
- **offset**

**Registers**
- Read reg. address1
- Read reg. address2
- Write reg. address
- Read data1
- Read data2
- Write data

**ALU**
- ALU result
- Zero
- ALU operation
- Shift left 2

**ADD**
- To PC

**RegWrite**
- PC+4 from instruction datapath

**Sign extend**
- bit0-15
- bit16-20
- bit21-25
- offset
- rs
- rt

**alu operation**

- 32
- 3
- 16
Combine the datapath R & I type

**Registers**
- Read reg. address1
- Read data1
- Read reg. address2
- Read data2
- Write reg. address
- Write data

**ALU**
- ALU result
- Zero
- MUX
- Address
- Data
- Memory
- Write data
- Sign
- Extend
- MUX
- Bit 0-15
- Bit 11-15
- Bit 16-20
- Bit 21-25
- Shift left 2
- MenWrite
- MenRead
- 32bits data

**Decoded指令**
- op(6)
- rs(5)
- rt(5)
- Immediate data

**PC**
- Read address
- Instruction
- Memory

**Instruction Memory**
- Instruction

**MUX**
- MUX
- MUX
- MUX

**Mux**
- To PC

**RegWrite**
- RegWrite

**ALU Operation**
- ADD
- ADD
- ADD
- ADD
- To PC

**Memory**
- Read
- Instruction
- Instruction
- Memory
- Write data
Combine the datapath R & I type

- **PC**
- **Read address**
- **Instruction**
- **Instruction Memory**

**Registers**
- Read address1
- Read data1
- Read address2
- Read data2
- Write address
- Write data

**ALU operation**
- **ADD**
- **Shift left 2**

**Zero**
- **MenWrite**

**RegWrite**
- **MUX**

**MUX**
- bit0-15
- bit11-15
- bit16-20
- bit21-25

**32bits data**

**Instruction Memory**
- **Read address**
- **Instruction**

**Data Memory**
- **Read address**
- **Instruction**
- **Read data1**
- **Read data2**
- **Write address**
- **Write data**

**ALU result**
- **RegWrite**
- **ALU operation**

**RegWrite**
- **MUX**

**MenRead**
- **MenWrite**
Combine the datapath R & I type

[Diagram with various datapath components and connections, including RegWrite, MenWrite, MenRead, ALU operation, and instruction memory.]
Chapter Five

The processor: Datapath and control

5.1 Introduction
5.2 Logic Design Conventions (skip)
5.3 Building a datapath
5.4 A Simple Implementation Scheme
5.5 A Multicycle Implementation
5.5 Microprogramming
5.6 Exception
Building the Datapath

- Use multiplexors to stitch them together

Note: control signals  Page 306 F5.16
Building Control

Analyse for cause and effect

- Information comes from the 32 bits of the instruction
- Selecting the operations to perform (ALU, read/write, etc.)
- Controlling the flow of data (multiplexor inputs)
- ALU's operation based on instruction type and function code

### R-format instruction (add, sub, and, or, slt)

<table>
<thead>
<tr>
<th></th>
<th>3 1</th>
<th>2 1</th>
<th>25</th>
<th>21</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6 5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Op</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Rs</strong></td>
<td>6 bits</td>
<td>5bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Rt</strong></td>
<td>5bits</td>
<td>5bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Rd</strong></td>
<td>5bits</td>
<td>5bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Shamt</strong></td>
<td>5bits</td>
<td>6bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Funct</strong></td>
<td>6bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### I-format instruction (lw, sw, beq)

<table>
<thead>
<tr>
<th></th>
<th>6 bits</th>
<th>5bits</th>
<th>5bits</th>
<th>16bits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Op</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Rs</strong></td>
<td>5bits</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Rt</strong></td>
<td>5bits</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Immediate</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### J-format instruction (add, sub, and, or, slt)

<table>
<thead>
<tr>
<th></th>
<th>6 bits</th>
<th>26bits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Op</strong></td>
<td></td>
<td>address</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6 bits</th>
<th>26bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Instruction Code

<table>
<thead>
<tr>
<th>Instruction Code</th>
<th>Instruction</th>
<th>Field 1</th>
<th>Field 2</th>
<th>Field 3</th>
<th>Field 4</th>
<th>Field 5</th>
<th>Field 6</th>
<th>Field 7</th>
<th>Field 8</th>
<th>Field 9</th>
<th>Field 10</th>
<th>Field 11</th>
<th>Field 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>R 000000</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>00000</td>
<td>100000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>R 000000</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>00000</td>
<td>100010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>R 000000</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>00000</td>
<td>100100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>R 000000</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>00000</td>
<td>100101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>slt</td>
<td>R 000000</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>00000</td>
<td>101010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>I 100011</td>
<td>rs</td>
<td>rt</td>
<td>Immediate(displacement)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>I 101011</td>
<td>rs</td>
<td>rt</td>
<td>Immediate(displacement)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td>I 000100</td>
<td>rs</td>
<td>rt</td>
<td>Immediate(offset)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>J 000010</td>
<td></td>
<td></td>
<td>address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **R**: Register
- **I**: Immediate
- **J**: Jump
What should ALU do?

- e.g. what should the ALU do with these instructions
- Example: lw $1, 100($2)

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>16 bit displacement</th>
<th>ALU op</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>(100011)35</td>
<td>2</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>sw</td>
<td>(101011)43</td>
<td>2</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>R</td>
<td>0000000(00)</td>
<td>rs(5)</td>
<td>rt(5)</td>
<td>rd(5)</td>
</tr>
</tbody>
</table>

- ALU control input
- 3 -types

<table>
<thead>
<tr>
<th>B negate</th>
<th>op</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>and</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>Or</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>Add</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>Sub</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>Slt</td>
</tr>
</tbody>
</table>

- Why is the code for subtract 110 and not 011?
Scheme of Controller

- **2-level decoder**

  - First Main decoder
    - ALU operation (3)
    - Defined at Chapter-3 ALU operation (3)

  - Second Decoder
    - Defined
    - ALU op (2)
    - Defined

  - Signals for Other Components (7)
    - Defined

  - op(6) rs(5) rt(5) rd(5) shamt func(6)
The ALU control is where and other signals.
### signals for datapath

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Effect when deasserted (=0)</th>
<th>Effect when asserted (=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RegDst</strong></td>
<td>Select register destination number from the rt(20:16) when WR</td>
<td>Select register destination number from the rd(15:11) when WB</td>
</tr>
<tr>
<td><strong>RegWrite</strong></td>
<td>None</td>
<td>Register destination input is written with the value on the Write data input</td>
</tr>
<tr>
<td><strong>ALUScr</strong></td>
<td>The second ALU operand come from the second register file output (Read data 2)</td>
<td>The second ALU operand is the sign-extended lower 16 bits of the instruction..</td>
</tr>
<tr>
<td><strong>PCSrc</strong></td>
<td>The PC is replaced by the output of the adder that computes the value PC+4</td>
<td>The PC is replaced by the output of the adder that computes the branch target.</td>
</tr>
<tr>
<td><strong>MemRead</strong></td>
<td>None</td>
<td>Data memory contents designated by the address input are put on the Read data output.</td>
</tr>
<tr>
<td><strong>MemWrite</strong></td>
<td>None</td>
<td>Data memory contents designated by the address input are replaced by value on the Write data input.</td>
</tr>
<tr>
<td><strong>MemtoReg</strong></td>
<td>The value fed to register Write data input comes from the Alu</td>
<td>The value fed to the register Write data input comes from the data memory.</td>
</tr>
</tbody>
</table>
Designing the Main Control Unit (First level)

- Main Control Unit function
  - ALU op (2)
  - Divided 7 control signals into 2 groups
    - 4 Mux
    - 3 R/W

<table>
<thead>
<tr>
<th>Instruction op code (6)</th>
<th>ALU control</th>
<th>LW</th>
<th>SW</th>
<th>Beq</th>
<th>R-type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>00</td>
<td>00</td>
<td>01</td>
<td>10</td>
</tr>
</tbody>
</table>

- ALU op (2)
- Mux (4)
- R/W (3)
- MemRead
- MemWrite
- RegWrite
- RegDst
- ALUScr
- PCSrc
- MemtoReg
Truth Table for Main decoder

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALU_op1</th>
<th>ALU_op0</th>
</tr>
</thead>
<tbody>
<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>
<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>
</tbody>
</table>
Circuitry of main Controller

- Simple combinational logic (truth tables)

<table>
<thead>
<tr>
<th>opcode</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>R-format</td>
</tr>
<tr>
<td>100011</td>
<td>lw</td>
</tr>
<tr>
<td>101011</td>
<td>sw</td>
</tr>
<tr>
<td>000100</td>
<td>beq</td>
</tr>
</tbody>
</table>

L/S 00
beq 01
R-type 10
Designing the ALU decoder (Second level)

- Must describe hardware to compute 3-bit ALU control input

<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>Load word</td>
<td>0010</td>
</tr>
<tr>
<td>SW</td>
<td>00</td>
<td>Store word</td>
<td>xxxxxxx</td>
<td>Store word</td>
<td>0010</td>
</tr>
<tr>
<td>Beq</td>
<td>01</td>
<td>branch equal</td>
<td>xxxxxxx</td>
<td>branch equal</td>
<td>0110</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>add</td>
<td>1000000</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>
</tbody>
</table>
Truth Table for ALU decoder

- Describe it using a truth table (can turn into gates):

<table>
<thead>
<tr>
<th>ALUOp</th>
<th>ALUOp0</th>
<th>Func field</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></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>
</tbody>
</table>

**Operation** 2 = \( ALU_{op0} + ALU_{op1}(\overline{F_3} \overline{F_2} F_1 \overline{F_0} + F_3 \overline{F_2} F_1 \overline{F_0}) \)

**Operation** 1 = \( ALU_{op1} \overline{F_3} F_2 \overline{F_1} \overline{F_0} + ALU_{op1} \overline{F_3} F_2 \overline{F_1} F_0 \)

**Operation** 0 = \( ALU_{op1} \overline{F_3} F_2 \overline{F_1} F_0 + ALU_{op1} F_3 \overline{F_2} F_1 \overline{F_0} \)
The ALU control signals---logic circuit

Operation 2 = \( ALU_{op0} + ALU_{op1} F_1 \)

Operation 1 = \( \overline{ALU_{op1}} + \overline{F}_2 \)

Operation 0 = \( ALU_{op1} + (F_0 + F_3) \)
Our Simple Control Structure

- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce right answer? right away
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path

We are ignoring some details like setup and hold times
The simple Datapath with the control unit

R-type
Op rs rt rd shamt Funct
I-type
Op rs rt Immediate
Jump-type
Op address

Instruction [31-0]
Shift left 2
jump address [31-0]
PC+4 [31-28]

Add
Shift left 2

Instruction [31-26]
Control

Instruction [25-21]
Read register 1
Read data 1

Instruction [20-16]
Read register 2
Write register
Write data

Instruction [15-11]
Read register
Write data

Instruction [10-6]
Zero
ALU
ALU result

Instruction [5-0]
ALU control

Registers

Add ALU result

Zero

ALU
result

Add

4

pc

Read
address

Instruction memory

Instruction

[31-0]

Instruction

[25-0]

jump

[31-0]

jump

[31-26]

jump

[31-28]

jump

[31-0]

jump

[31-26]

jump

[31-28]

jump

[31-0]

jump

[31-26]

jump

[31-28]

jump

[31-0]

jump

[31-26]

jump

[31-28]

jump

[31-0]

jump

[31-26]

jump

[31-28]

jump

[31-0]

jump

[31-26]

jump

[31-28]

jump

[31-0]

jump

[31-26]

jump

[31-28]

jump

[31-0]

jump

[31-26]

jump

[31-28]

jump

[31-0]

jump

[31-26]

jump

[31-28]

jump

[31-0]

jump

[31-26]

jump

[31-28]

jump

[31-0]

jump

[31-26]

jump

[31-28]

jump

[31-0]

jump

[31-26]

jump

[31-28]

jump

[31-0]

jump

[31-26]

jump

[31-28]

jump

[31-0]

jump

[31-26]

jump

[31-28]

jump

[31-0]

jump

[31-26]

jump

[31-28]

jump

[31-0]

jump

[31-26]

jump
The Datapath in operation for load

Instruction [15-0]

Add

Instruction [31-26]

Control

Instruction [31-0]

Register memory

Instruction [25-21]

Zero

ALU

ADD

Instruction [20-16]

ALUOp

MemRead

MemToRead

Instruction [15-11]

RegWrite

ALUSrc

MemWrite

Instruction [10-6]

RegDst

Branch

Instruction [5-0]

ALUcontrol

Sign extend

Zero

ALU result

Address

Read data

Data memory

Write data

jump

jump

Instruction [25-0]

Shift left 2

Instruction [31-28]

Instruction [31-26]

Control

Instruction [31-0]

Register memory

Instruction [25-21]

Zero

ALU

ADD

Instruction [20-16]

ALUOp

MemRead

MemToRead

Instruction [15-11]

RegWrite

ALUSrc

MemWrite

Instruction [10-6]

RegDst

Branch

Instruction [5-0]

ALUcontrol

Sign extend

Zero

ALU result

Address

Read data

Data memory

Write data

jump
The Datapath in operation for store

store instruction

R-type

I-type

Jump-type

Op rs rt Immediate

Control

Add

pc

Instruction memory

Instruction [31-0]

Instruction [25-0]

Instruction [25-21]

Instruction [20-16]

Instruction [15-11]

Instruction [15-0]

Instruction [31-26]

Instruction [5-0]

Add data 1

Read register 1

Read register 2

Read data 1

Write register

Write data

Read address

mul

ALUOp

MemRead

MemtoRead

RegWrite

MemWrite

ALUSrc

RegDst

Branch

jump

jump address[31-0]

Shift left 2

PC+4[31-28]

Add ALU result

Shift left 2

0 MUX 1

1 MUX 0

0 MUX 0

1 MUX 1

1 MUX 0

0 MUX 1

Address

Write data

Data memory

Zero

ALU

ALU result

Address

Read data

Registers

alu

sign extend

16

32

alu control
The Datapath in operation for beq instruction

beq instruction

R-type

I-type

Op rs rt Immediate

Op address

Add

Instruction [31-26]

Control

Instruction [25-21]

Instruction [20-16]

Instruction [15-11]

Instruction [15-0]

Instruction [5-0]

Add ALU result

Shift left 2

Add ALU

MUX 1

MUX 0

Jump-type

Op address

RegDst Branch

MemRead MemtoRead ALUOp

ALUSrc RegWrite

Zero ALU result

ALU control

Write data

Address Read data

Data memory

Write register

Read data 1

Read register 1

RegWrite

Instruction memory

Instruction [31-0]

Instruction [25-0]

Shift left 2

jump address [31-0]

jump address [28-26]

PC+4 [31-28]

Add register

Write data 1

Read register 2

Instruction [25-21]

Instruction [20-16]

Instruction [15-11]

Instruction [15-0]

16

32

Sign extend

Registers

Bit 16

Bit 32

Instruction memory

Instruction [31-0]
**j instruction**

- **Instruction format**
  
  $j$ Label

  \[(000010)_2 \quad 26 \text{ bits address}\]

- **Implementation**

  \[pc = pc_{28\sim31} \# 26\text{bits-address} \times 4\]
The Datapath in operation for Jump

### Jump Instruction

- **Op**: address
- **R-Type**: 
  - `fshl` 
  - `mt nc t` 
  - `immediat` 
- **Jump-type**: 
  - PC+4[31-28] 
  - Jump-type 

**Instruction memory**

**Instruction [25-0]**

**Instruction [25-21]**

**Instruction [20-16]**

**Instruction [15-11]**

**Instruction [10-0]**

**Instruction [5-0]**

**Shift left 2**

**Jump address [31-0]**

**Add**

**Add ALU result**

**Shift left 2**

**1 MUX 0**

**0 MUX 1**

**jump instruction**

**Add ALU result**

**Zero ALU result**

**Address Read data**

**Data memory Write data**

**ALU control**

**Control**

**RegDst**

**Branch**

**MemRead**

**MemtoRead**

**ALUOp**

**MemWrite**

**ALUSrc**

**RegWrite**

**Read address**

**ALU**

**ALUSrc**

**RegWrite**

**RegDst**

**Branch**

**Zero**

**Jump address [31-0]**

**Add data 2**

**ALU result**

**Write data 2**

**Write register 2**

**Read register 2**

**Instruction [25-0]**

**Instruction memory**

**Instruction [31-26]**

**Instruction [25-21]**

**Instruction [15-11]**

**Instruction [15-0]**

**Instruction [10-0]**

**Instruction [5-0]**

**Shift left 2**

**PC+4[31-28]**
Calculate cycle time assuming negligible delays except:
- memory (2ns), ALU and adders (2ns), register file access (1ns)
Performance in Single Cycle Implementation

- Let’s see the following table:

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>Instruction memory</th>
<th>Register read</th>
<th>ALU</th>
<th>Data memory</th>
<th>Register write</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td>1</td>
<td>6 ns</td>
</tr>
<tr>
<td>Load word</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>8 ns</td>
</tr>
<tr>
<td>Store word</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td>7 ns</td>
</tr>
<tr>
<td>Branch</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td>5 ns</td>
</tr>
<tr>
<td>Jump</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 ns</td>
</tr>
</tbody>
</table>

- The conclusion:

Different instructions needs different time. The clock cycle must meet the need of the slowest instruction. So, some time will be wasted.
The CPU Performance Equation

\[ \text{CPU time} = \text{CPU clock cycles for a program} \times \text{Clock cycle time} \]

\[ \text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}} \]

\[ \text{CPU time} = I \cdot \text{CPI} \cdot \tau \]
CPI = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}}

\text{CPU time} = \text{Instruction count} \times \text{Clock cycle time} \times \text{Cycles per instruction}

\text{CPU time} = \frac{\text{Instruction count} \times \text{Clock cycle time}}{\text{Clock rate}}

\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}
CPU performance is dependent upon three characteristics:
- clock cycle (or rate)
- clock cycles per instruction
- and instruction count.

It is difficult to change one parameter in complete isolation from others because the basic technologies involved in changing each characteristic are interdependent:
- *Clock cycle time*—Hardware technology and organization
- *Clock cycle time*—Hardware technology and organization
- *Instruction count*—Instruction set architecture and compiler technology
MIPS (million instruction per second)

MIPS = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}

\text{Execution time} = \frac{\text{Instruction count}}{\text{MIPS} \times 10^6}

× \textbf{The bigger the MIPS, the faster the machine.}

- Three problems with MIPS:
  - MIPS is dependent on the instruction set, making it difficult to compare MIPS of computers with different instruction sets.
  - MIPS varies between programs on the same computer.
  - Most importantly, MIPS can vary inversely to performance!
Single Cycle Problems

- what if we had a more complicated instruction like floating point?
  
  If so, the waste of time will be more serious.

- wasteful of area. The reason is the following:
  
  - Let’s see the instruction ‘mult’. This instruction needs to use the ALU repeatedly.
  
  - But, in the single cycle implementation, one ALU can be used only once in one clock cycle.
  
  - So, the instruction ‘mult’ will need many ALUs. The CPU will be very large.
One Solution for Single Cycle Problems

- One Solution:
  - Use a smaller cycle time
  - Let different instructions take different numbers of cycles
- a Multicycle datapath:
Chapter Five

The processor: Datapath and control

5.1 Introduction
5.2 Logic Design Conventions (skip)
5.3 Building a datapath
5.4 A Simple Implementation Scheme
5.5 A Multicycle Implementation
5.5 Microprogramming
5.6 Exception
Multicycle Approach

• Break up the instructions into steps, each step takes a cycle
  – balance the amount of work to be done
  – restrict each cycle to use only one major functional unit
• At the end of a cycle
  – store values for use in later cycles
  – introduce additional internal registers
Analyse events: Five Execution Steps

- **IF**: Instruction Fetch
- **ID**: Instruction Decode and Register Fetch
- **EX（BC）**: Execution, Memory Address Computation, or Branch Completion
- **MEM（WB）**: Memory Access or R-type instruction completion
- **WB**: Write-back step

*INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!*
Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register.
  - \( IR = \text{Memory}[PC] \);
- Increment the PC by 4 and put the result back in the PC.
  - \( PC = PC + 4 \);
- Can be described simply using RTL "Register-Transfer Language"
  
  \[
  IR = \text{Memory}[PC]; \\
  PC = PC + 4;
  \]
- Can we figure out the values of the control signals?
- What is the advantage of updating the PC now?
Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:
  
  \[
  \begin{align*}
  A &= \text{Reg[IR[25-21]];} \\
  B &= \text{Reg[IR[20-16]];} \\
  \text{ALUOut} &= \text{PC} + (\text{sign-extend(IR[15-0])} \ll 2);
  \end{align*}
  \]

- We aren't setting any control lines based on the instruction type
  (we are busy "decoding" it in our control logic)
Step 3 (instruction dependent)

- ALU is performing one of three functions, based on instruction type
  - Memory Reference ( lw / sw ):
    \[ \text{ALUOut} = A + \text{sign-extend}(IR[15-0]) \]
  - R-type:
    \[ \text{ALUOut} = A \text{ op } B \]
  - Branch:
    \[ \text{if (A==B) PC = ALUOut} \]
  - jump:
    \[ \text{pc} = \text{pc31-28} + \text{IR25-0} \ll 2 \]
Step 4 (R-type or memory-access)

- Loads and stores access memory
  
  \[
  \text{MDR} = \text{Memory}[\text{ALUOut}]; \quad \# \text{ for lw}
  \]

  or

  \[
  \text{Memory}[\text{ALUOut}] = B; \quad \# \text{ for sw}
  \]

- R-type instructions finish

  \[
  \text{Reg}[rd]=\text{Reg}[\text{IR[15-11]}] = \text{ALUOut};
  \]

*The write actually takes place at the end of the cycle on the edge*
Write-back step (step 5)

- `lw`
  - `Reg[rt]=Reg[IR[20-16]]= MDR;`

*What about all the other instructions?*
### Summary:

<table>
<thead>
<tr>
<th>Step name</th>
<th>Action for R-type instructions</th>
<th>Action for memory-reference instructions</th>
<th>Action for branches</th>
<th>Action for jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td></td>
<td>IR = Memory[PC]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC = PC + 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction decode/ register fetch</td>
<td></td>
<td>A = Reg [IR[25-21]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>B = Reg [IR[20-16]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALUOut = PC + (sign-extend (IR[15-0]) &lt;&lt; 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution, address computation, branch/jump completion</td>
<td>ALUOut = A op B</td>
<td>ALUOut = A + sign-extend (IR[15-0])</td>
<td>if (A == B) then PC = ALUOut</td>
<td>PC = PC [31-28] II (IR[25-0] &lt;&lt; 2)</td>
</tr>
<tr>
<td>Memory access or R-type completion</td>
<td>Reg [IR[15-11]] = ALUOut</td>
<td>Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B</td>
<td>Load: Reg[IR[20-16]] = MDR</td>
<td></td>
</tr>
<tr>
<td>Memory read completion</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Simple Questions

- How many cycles will it take to execute this code? (12)
  
  ```
  lw $t2, 0($t3)
  lw $t3, 4($t3)
  beq $t2, $t3, Label       #assume not
  add $t5, $t2, $t3
  sw $t5, 8($t3)
  ```
  
  Label: ...

- What is going on during the 8th cycle of execution?
  - Answer: Calculating memory address

- In what cycle does the actual addition of $t2 and $t3 take place?
  - No. 9

---

 RAW TEXT END
Reusing Resource

- We will be reusing functional units
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- We will use a finite state machine for control
Scheme of Controller of Multicycle
How does it control in Multicycle Approach
<table>
<thead>
<tr>
<th>Signal name</th>
<th>Effect when deasserted(=0)</th>
<th>Effect when asserted(=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegDst</td>
<td>Select register destination number from the rt(20:16) when WR.</td>
<td>Select register destination number from the rd(15:11) when WB.</td>
</tr>
<tr>
<td>RegWrite</td>
<td>None</td>
<td>Register destination input is written with the value on the Write data input</td>
</tr>
<tr>
<td>ALUScrA</td>
<td>The first ALU operand is the PC</td>
<td>The first ALU operand come from the A register.</td>
</tr>
<tr>
<td>MemRead</td>
<td>None</td>
<td>Memory contents at the location specified by the address input is put on the Memory data out.</td>
</tr>
<tr>
<td>MemWrite</td>
<td>None</td>
<td>Memory contents at the location specified by the address input are replaced by value on the Write data input.</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>The value fed to register Write data input comes from the ALUOut</td>
<td>The value fed to the register Write data input comes from the MDA.</td>
</tr>
<tr>
<td>IorD</td>
<td>The PC is used to supply the address to the memory unit.</td>
<td>ALUOut is used to supply the address to the memory unit.</td>
</tr>
<tr>
<td>IRWrite</td>
<td>None</td>
<td>The output of memory is written into the IR.</td>
</tr>
<tr>
<td>PCWrite</td>
<td>None</td>
<td>The is written;the source is controlled by PCSource.</td>
</tr>
<tr>
<td>PCWriteCond</td>
<td>None</td>
<td>The PC is written if the zero output from the ALU is also active.</td>
</tr>
</tbody>
</table>
## Control signals

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUOp</td>
<td>00</td>
<td>The ALU performs an add operation.</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>The ALU performs a subtract operation.</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>The funct field of the instruction determines the ALU operation.</td>
</tr>
<tr>
<td>ALUScrB</td>
<td>00</td>
<td>The second input to the ALU comes from the B register.</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>The second input to the ALU is the constant 4.</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>The second input to the ALU is the sign-extended, lower 16 bits of the IR.</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>The second input to the ALU is the sign-extended, lower 16 bits of the IR shift 2 bits.</td>
</tr>
<tr>
<td>PCSource</td>
<td>00</td>
<td>Output of the ALU (PC+4) is sent to the PC for writing.</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>The contents of ALUOut (the branch target address) are sent to the PC writing.</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>The jump target address (IR[25:0] shifted left 2 bits and concatenated with PC+4[31:28]) is sent to the PC for writing.</td>
</tr>
</tbody>
</table>
seq:  pc = pc + 4
beq: pc = pc + offset * 4
j: pc = pc_{31:28} + IR_{25:0} \ll 2
Implementing the Control

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed

- Let’s go over the main control unit in the single-cycle implementation.
• Review for singlecycle

• Main control unit in the single-cycle implementation

R-type 0
lw 35
sw 43
beq 4
• Review

• Now, let’s look at the ALU control unit. It does not need to changed.
- As same as the single-cycle

- So, the following truth table remains the same.

<table>
<thead>
<tr>
<th>ALUOp1</th>
<th>ALUOp0</th>
<th>F5</th>
<th>F4</th>
<th>F3</th>
<th>F2</th>
<th>F1</th>
<th>F0</th>
<th>Operation</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>010</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>110</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>010</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>110</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>000</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>001</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>111</td>
</tr>
</tbody>
</table>
Review: Finite state machines

- Finite state machines:
  - a set of states
  - next state function (determined by current state and the input)
  - output function (determined by current state and possibly input)
• **Difference in controller**

• **Do something different in each cycle**
  • **Two main ways to implement the control**
    – 1. Finite state machine
    – 2. Use microprogramming
  • Next, we will discuss the first way.
State diagram for Instruction execute flow

Ref Fig5.31 (p.332)

Start

Instruction fetch/decode and registerfetch
Figure(5.32)

Memory access instructions
Figure(5.33)

R-type instructions
Figure(5.34)

Branch instructions
Figure(5.35)

Jump instructions
Figure(5.36)
Instruction fetch / decode and Reg fetch

Memory reference FSM (Figure 5.33)
R-type FSM (Figure 5.34)
Branch FSM (Figure 5.35)
Jump FSM (Figure 5.36)

0

Instruction fetch

MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00

1

Instruction decode/Register fetch

ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00

(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = BEQ)
(Op = 'JMP')
**lw** and **sw**

- **Memory address computation**
  - $\text{Op} = \text{'LW'}$ or $\text{Op} = \text{'SW'}$

- From state 1

- To state 0 (Figure 5.32)

**Transition States**

- **State 2**
  - $\text{ALUSrcA} = 1$
  - $\text{ALUSrcB} = 10$
  - $\text{ALUOp} = 00$

- **State 3**
  - Memory access
  - $\text{MemRead lorD} = 1$

- **State 4**
  - Write-back step
  - $\text{RegWrite MemtoReg} = 1$
  - $\text{RegDst} = 0$

- **State 5**
  - Memory access
  - $\text{MemWrite lorD} = 1$
$R$-type

From state 1

$Op=R$-type

6

$ALUSrcA=1$

$ALUSrcB=00$

$ALUOp=10$

R-type completion

7

$RegDst=1$

$RegWrite$

$MemtoReg=0$

To state 1

(Figure 5.32)
Branch

From state 1

Op='BEQ'

ALUSrcA=1
ALUSrcB=00
ALUOp=01
PCWriteCond
PCSource=01

To state 1

(Figure 5.32)
J-type

From state 1

Op='J'

9

PCSource=10

Jump completion

To state 1

(Figure 5.32)
Graphical Specification of FSM

Instruction fetch

Instruction decode/register fetch

ALUSrcA = 11
ALUSrcB = 0
ALUOp = 00
PCWrite
PCSource = 01

ALUSrcA = 0
ALUSrcB = 10
ALUOp = 00

ALUSrcA = 1
ALUSrcB = 0
ALUOp = 10

ALUSrcA = 1
ALUSrcB = 0
ALUOp = 01

MemRead
ALUSrcA = 0
lrd = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00

MemWrite
ALUSrcA = 1
lrd = 1
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 01

RegDst = 0
RegWrite
MemtoReg = 1

RegDst = 1
RegWrite
MemtoReg = 0

Write-back step

Memory address computation

Memory access

Memory access

R-type completion

Execution

Branch completion

Jump completion

beq

lw

SW

R-type

jump
The truth table for the 16 datapath control outputs, which depend only on the state inputs.

<table>
<thead>
<tr>
<th>Outputs</th>
<th>Input values (S[3–0])</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000</td>
</tr>
<tr>
<td>PCWrite</td>
<td>1</td>
</tr>
<tr>
<td>PCWriteCond</td>
<td>0</td>
</tr>
<tr>
<td>IorD</td>
<td>0</td>
</tr>
<tr>
<td>MemRead</td>
<td>1</td>
</tr>
<tr>
<td>MemWrite</td>
<td>0</td>
</tr>
<tr>
<td>IRWrite</td>
<td>1</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>0</td>
</tr>
<tr>
<td>PCSource1</td>
<td>0</td>
</tr>
<tr>
<td>PCSource0</td>
<td>0</td>
</tr>
<tr>
<td>ALUOp1</td>
<td>0</td>
</tr>
<tr>
<td>ALUOp0</td>
<td>0</td>
</tr>
<tr>
<td>ALUSrcB1</td>
<td>0</td>
</tr>
<tr>
<td>ALUSrcB0</td>
<td>1</td>
</tr>
<tr>
<td>ALUSrcA</td>
<td>0</td>
</tr>
<tr>
<td>RegWrite</td>
<td>0</td>
</tr>
<tr>
<td>RegDst</td>
<td>0</td>
</tr>
</tbody>
</table>
The logic equations for the control unit shown in a shorthand form

<table>
<thead>
<tr>
<th>Output</th>
<th>Current states</th>
<th>Op</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCWrite</td>
<td>state0 + state9</td>
<td></td>
</tr>
<tr>
<td>PCWriteCond</td>
<td>state8</td>
<td></td>
</tr>
<tr>
<td>lorD</td>
<td>state3 + state5</td>
<td></td>
</tr>
<tr>
<td>MemRead</td>
<td>state0 + state3</td>
<td></td>
</tr>
<tr>
<td>MemWrite</td>
<td>state5</td>
<td></td>
</tr>
<tr>
<td>IRWrite</td>
<td>state0</td>
<td></td>
</tr>
<tr>
<td>MemtoReg</td>
<td>state4</td>
<td></td>
</tr>
<tr>
<td>PCSource1</td>
<td>state9</td>
<td></td>
</tr>
<tr>
<td>PCSource0</td>
<td>state8</td>
<td></td>
</tr>
<tr>
<td>ALUOp1</td>
<td>state6</td>
<td></td>
</tr>
<tr>
<td>ALUOp0</td>
<td>state8</td>
<td></td>
</tr>
<tr>
<td>ALUSrcB1</td>
<td>state1 + state2</td>
<td></td>
</tr>
<tr>
<td>ALUSrcB0</td>
<td>state0 + state1</td>
<td></td>
</tr>
<tr>
<td>ALUSrcA</td>
<td>state2 + state6 + state8</td>
<td></td>
</tr>
<tr>
<td>RegWrite</td>
<td>state4 + state7</td>
<td></td>
</tr>
<tr>
<td>RegDst</td>
<td>state7</td>
<td></td>
</tr>
<tr>
<td>NextState0</td>
<td>state4 + state5 + state7+ state8 + state9</td>
<td>(Op = 'lw') + (Op = 'sw')</td>
</tr>
<tr>
<td>NextState1</td>
<td>state0</td>
<td></td>
</tr>
<tr>
<td>NextState2</td>
<td>state1</td>
<td></td>
</tr>
<tr>
<td>NextState3</td>
<td>state2</td>
<td></td>
</tr>
<tr>
<td>NextState4</td>
<td>state3</td>
<td></td>
</tr>
<tr>
<td>NextState5</td>
<td>state2</td>
<td></td>
</tr>
<tr>
<td>NextState6</td>
<td>state1</td>
<td></td>
</tr>
<tr>
<td>NextState7</td>
<td>state6</td>
<td></td>
</tr>
<tr>
<td>NextState8</td>
<td>state1</td>
<td></td>
</tr>
<tr>
<td>NextState9</td>
<td>state1</td>
<td></td>
</tr>
</tbody>
</table>
Implemented using a block of combinational logic and a register to hold the current state.
Chapter Five

The processor: Datapath and control

5.1 Introduction
5.2 Logic Design Conventions (skip)
5.3 Building a datapath
5.4 A Simple Implementation Scheme
5.5 A Multicycle Implementation
5.5 Microprogramming
5.6 Exception
Microinstruction format

- Microinstruction
  - control signal
  - position of next Microinstruction

- Representation
  - split into some fields
    - next position
      - sequential
      - dispatch table (jump)
## Microinstruction format

<table>
<thead>
<tr>
<th>Field name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ALU control</strong></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>ALUOp = 00</td>
</tr>
<tr>
<td>Subt</td>
<td>ALUOp = 01</td>
</tr>
<tr>
<td>Func code</td>
<td>ALUOp = 10</td>
</tr>
<tr>
<td><strong>SRC1</strong></td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td>ALUSrcA = 0</td>
</tr>
<tr>
<td>A</td>
<td>ALUSrcA = 1</td>
</tr>
<tr>
<td><strong>SRC2</strong></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>ALUSrcB = 00</td>
</tr>
<tr>
<td>4</td>
<td>ALUSrcB = 01</td>
</tr>
<tr>
<td>Extend</td>
<td>ALUSrcB = 10</td>
</tr>
<tr>
<td>Extshft</td>
<td>ALUSrcB = 11</td>
</tr>
<tr>
<td><strong>Register control</strong></td>
<td></td>
</tr>
<tr>
<td>Read (Reg fetch)</td>
<td>A=Reg[IR_{25-21}], B=Reg[IR_{20-16}]</td>
</tr>
<tr>
<td>Write ALU</td>
<td>RegWrite, RegDst = 1, MemtoReg = 0</td>
</tr>
<tr>
<td>Write MDR</td>
<td>RegWrite, RegDst = 0, MemtoReg = 1</td>
</tr>
</tbody>
</table>
## Microinstruction format

<table>
<thead>
<tr>
<th>Memory</th>
<th>Read PC</th>
<th>MemRead, lorD = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read ALU</td>
<td>MemRead,</td>
<td>lorD = 1</td>
</tr>
<tr>
<td>Write ALU</td>
<td>MemWrite,</td>
<td>lorD = 1</td>
</tr>
<tr>
<td></td>
<td>ALU</td>
<td>PCSource = 00,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PCWrite</td>
</tr>
<tr>
<td>PC write control</td>
<td>ALUOut-cond</td>
<td>PCSource = 01,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PCWriteCond</td>
</tr>
<tr>
<td>jump address</td>
<td></td>
<td>PCSource = 10,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PCWrite</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Seq</td>
<td>AddrCtl = 11</td>
</tr>
<tr>
<td></td>
<td>Fetch</td>
<td>AddrCtl = 00</td>
</tr>
<tr>
<td></td>
<td>Dispatch 1</td>
<td>AddrCtl = 01</td>
</tr>
<tr>
<td></td>
<td>Dispatch 2</td>
<td>AddrCtl = 10</td>
</tr>
</tbody>
</table>
## Steps of Instructions

<table>
<thead>
<tr>
<th>Step name</th>
<th>Action for R-type instructions</th>
<th>Action for memory-reference instructions</th>
<th>Action for branches</th>
<th>Action for jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td></td>
<td>IR = Memory[PC]</td>
<td>PC = PC + 4</td>
<td></td>
</tr>
<tr>
<td>Instruction decode/register fetch</td>
<td></td>
<td>A = Reg [IR[25-21]]</td>
<td>B = Reg [IR[20-16]]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALUOut = PC + (sign-extend (IR[15-0]) &lt;&lt; 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution, address computation, branch/jump completion</td>
<td>ALUOut = A op B</td>
<td>ALUOut = A + sign-extend (IR[15-0])</td>
<td>if (A == B) then PC = ALUOut</td>
<td>PC = PC [31-28] II (IR[25-0] &lt;&lt; 2)</td>
</tr>
<tr>
<td>Memory access or R-type completion</td>
<td>Reg [IR[15-11]] = ALUOut</td>
<td>Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory read completion</td>
<td>Load: Reg[IR[20-16]] = MDR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A specification methodology

- appropriate if hundreds of opcodes, modes, cycles, etc.
- signals specified symbolically using microinstructions

<table>
<thead>
<tr>
<th>address</th>
<th>Label</th>
<th>ALU control</th>
<th>SRC1</th>
<th>SRC2</th>
<th>Register control</th>
<th>Memory</th>
<th>PCWrite control</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Fetch</td>
<td>Add</td>
<td>PC</td>
<td>4</td>
<td>Read PC</td>
<td>ALU</td>
<td>Seq</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td>Add</td>
<td>PC</td>
<td>Extshft</td>
<td>Read</td>
<td></td>
<td>Dispatch 1</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>Mem1</td>
<td>Add</td>
<td>A</td>
<td>Extend</td>
<td></td>
<td></td>
<td>Dispatch 2</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>LW2</td>
<td>Add</td>
<td>A</td>
<td>Extend</td>
<td></td>
<td>Read ALU</td>
<td>Seq</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Write MDR</td>
<td>Read ALU</td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>SW2</td>
<td></td>
<td></td>
<td></td>
<td>Write ALU</td>
<td></td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>Rformat1</td>
<td>Func code</td>
<td>A</td>
<td>B</td>
<td>Write ALU</td>
<td></td>
<td>Seq</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Write ALU</td>
<td></td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>BEQ1</td>
<td>Subt</td>
<td>A</td>
<td>B</td>
<td>ALUOut-cond</td>
<td>Fetch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>JUMP1</td>
<td></td>
<td></td>
<td></td>
<td>Jump address</td>
<td>Fetch</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 微指令编码

<table>
<thead>
<tr>
<th></th>
<th>ALU</th>
<th>SRC1</th>
<th>SRC2</th>
<th>RegDst</th>
<th>PCWrite</th>
<th>MemRead</th>
<th>Memory Control</th>
<th>Register Control</th>
<th>Seq</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Label</strong></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
</tr>
<tr>
<td><strong>ALU control</strong></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
</tr>
<tr>
<td><strong>SRC1 SRC2</strong></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
</tr>
<tr>
<td><strong>Register control</strong></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
</tr>
<tr>
<td><strong>PCWrite control</strong></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
</tr>
<tr>
<td><strong>Seq</strong></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
<td><img src="image" alt="" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALUOp1</th>
<th>ALUOp0</th>
<th>ALUSrcA</th>
<th>ALUSrcB1</th>
<th>ALUSrcB0</th>
<th>IRWrite</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>MemtoReg</th>
<th>MemRead</th>
<th>MemRead</th>
<th>IorD</th>
<th>PCSrc1</th>
<th>PCSrc0</th>
<th>PCWrite</th>
<th>PCWriteCond</th>
<th>Addrc1</th>
<th>Addrc10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 0 0 0 0 0 0 1 0 0 0</td>
<td></td>
<td>0 0 0 0 0 0 0 1 0 0</td>
<td></td>
<td>0 0 0 0 0 0 0 1 0 0</td>
<td></td>
<td>0 0 0 0 0 0 0 1 0 0</td>
<td></td>
<td>0 0 0 0 0 0 0 1 0 0</td>
<td></td>
<td>0 0 0 0 0 0 0 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Microprogramming

- What are the Microinstructions?
Details

<table>
<thead>
<tr>
<th>Dispatch ROM 1</th>
<th>Opcode</th>
<th>Opcode name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>0110</td>
<td>R-format</td>
<td></td>
</tr>
<tr>
<td>000010</td>
<td>1001</td>
<td>jmp</td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>1000</td>
<td>beq</td>
<td></td>
</tr>
<tr>
<td>100011</td>
<td>0010</td>
<td>lw</td>
<td></td>
</tr>
<tr>
<td>101011</td>
<td>0010</td>
<td>sw</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dispatch ROM 2</th>
<th>Opcode</th>
<th>Opcode name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100011</td>
<td>0011</td>
<td>lw</td>
<td></td>
</tr>
<tr>
<td>101011</td>
<td>0101</td>
<td>sw</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State number</th>
<th>Address-control action</th>
<th>Value of AddrCt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Use incremented state</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>Use dispatch ROM 1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Use dispatch ROM 2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Use incremented state</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>Use incremented state</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
</tbody>
</table>

Seeing to CD
Chapter Five

The processor: Datapath and control

5.1 Introduction
5.2 Logic Design Conventions (skip)
5.3 Building a datapath
5.4 A Simple Implementation Scheme
5.5 A Multicycle Implementation
5.5 Microprogramming
5.6 Exception
5.6 Exception

• The cause of changing CPU’s work flow:
  – Control instructions in program (bne/beq, j, jal, etc)
    It is foreseeable in programming flow
  – Something happen suddenly (Exception and Interruption)
    It is unpredictable

• Unexpected events
  – Exception: from within processor (overflow, undefined instruction, etc)
  – Interruption: from outside processor (input/output)
5.6 Exception

• Exception
  An Exception is a unexpected event from within processor.

• We follow the MIPS convention, using the term **exception** to refer to any unexpected change in control flow.

• Here we will discuss two types exceptions:
  – arithmetic overflow
  – undefined instruction
How Exceptions Are Handled

- When exception happens, the processor must do something.
- The predefined process routines are saved in memory when computer starts.
- Problem: how can CPU goto relative routine when an exception occurs.
- CPU should know
  - the cause of exception
  - which instruction generate the exception
How Exceptions Are Handled

- **Design**
  - add a register: *exception program counter (EPC)*
    save the address of the offending instruction
  - add a status register: *cause register (CauseReg)*
    hold a field that indicates the reason for the exception.
    \[
    \text{bit0} = \begin{cases} \ 0 & \text{undefined instruction} \\ \ 1 & \text{overflow} \end{cases}
    \]
  - *Another method is to use vector interrupts*

<table>
<thead>
<tr>
<th>Exception type</th>
<th>vector address</th>
</tr>
</thead>
<tbody>
<tr>
<td>undefined instr</td>
<td>c0 00 00 00 H</td>
</tr>
<tr>
<td>overflow</td>
<td>c0 00 00 20 H</td>
</tr>
</tbody>
</table>
How Control Checks for Exceptions

• add control signal
  – **CauseWrite** for CauseReg
  – **EPCWrite** for EPC
    \[ EPC = PC - 4 \text{ (completed by ALU)} \]

• process of control
  – CauseReg = 0 or 1
  – EPC = PC - 4
  – PC <--- address of process routine ( ex. c0000000 )
How Control Checks for Exceptions
How Control Checks for Exceptions

• detect exceptions
  – Undefined instruction
    when no next state is defined from state 1 for op value. New state 10 is introduced.
  – Overflow
    Overflow is occurred only in R-type instruction. Overflow is provided as an output from the ALU. This signal is used in the modified FSM to specify an additional state 11 for state 7.
10
IntCause = 0
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11

11
IntCause = 1
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11

To state 0 to begin next instruction