# Computer Organization

## Lab10 CPU(2) Minisys, Controller, ALU

2021 Spring term

wangw6@sustech.edu.cn

# Topics

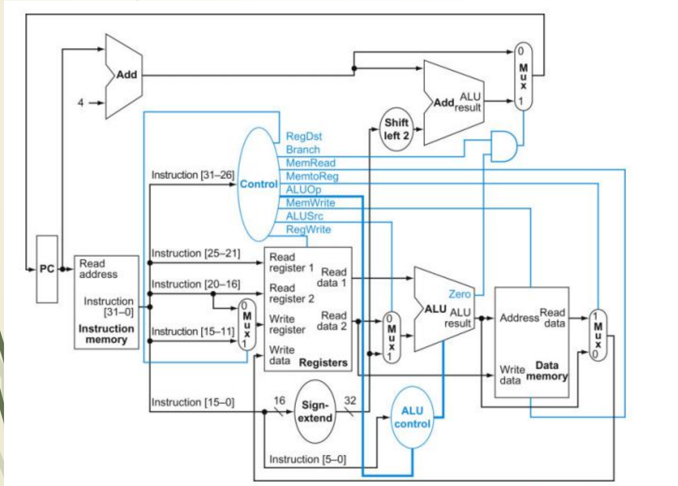
- CPU(2)
  - Controller
  - ALU
- Minisys
  - A Subset of MIPS32
  - The Assembler of Minisys

# Controller

Use opcode and funct code as input, generate the control signals which will be used in other modules.

**BASIC INSTRUCTION FORMATS**

| | | | | | | |
|---|---|---|---|---|---|---|
| R | opcode | rs | rt | rd | shamt | funct |
| | 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| I | opcode | rs | rt | immediate | | |
| | 31 26 | 25 21 | 20 16 | 15 0 | | |
| J | opcode | address | | | | |
| | 31 26 | 25 0 | | | | |




Source: H&P textbook

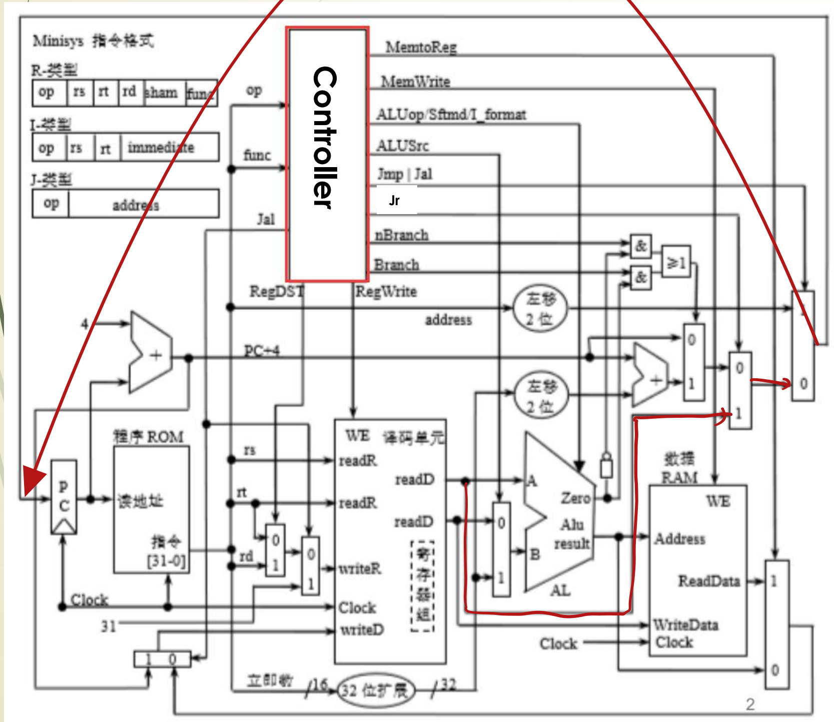
- part 1: get data from the instruction
  - address of registers: rs(Instruction[25:21]), rt(Instruction[20:16]) and rd(Instruction[15:11])
  - immediate(instruction[15:0])
  - shift mount(instruction[10:6])
  - address(instruction[25:0])
- **part 2: get and analyze code in the instruction**
  - opcode(instruction[31:26]), funct(bit[5:0])
  - **generate control signals** to control the instruction execution

# Controller continued

Why a controller is needed?

| Module | How To Process | Instructions and Comments |
| --- | --- | --- |
| **Decoder** | Determine whether to write register or not | (lw), (R type instruction), (jal) |
| | Get the source of data to be written | 1. data memory(lw)<br>2. alu(R)<br>3. address of instruction ->$31 (jal) |
| | Get the address of register to be written | 1. rt(lw)<br>2. rd(R)<br>3. 31(jal) |
| **Memory** | Determine whether to write memory or not | (sw) |
| | Get the source of data to be written | register (sw) |
| **ALU** | Determine how to calculate the datas | add, sub, or, sll, sra, slt, branch ... |
| | Get the source of one operand from register or immediate extended | R(register), I(sign extended immediate) |
| **iFetch** | Determine how to update the value of PC register | 1. (pc+4)+immediate(sign extended) (branch,I type)<br>2. the value of $31 register (jr, J type)<br>3. {(pc+4)[31:28],lableX[25:2],2'b00} |
| **...** | ... | ... |

# Controller continued




**Q1:** How to determine the type of the instruction, R, I or J?

**Q2:** What's the usage of function code in the instruction?

**Q3:** How to generate these control signals?

**Q4:** What's the type of the circuit about Controller? A combinational logic or a seqencial logic?

0|0|0|0|0|0|0|0|0|0|0

# Minisys - A subset of MIPS32

| Type | Name | funC(ins[5:0]) |
|---|---|---|
| R | sll | 00_0000 |
| | sllv | 00_0100 |
| | srl | 00_0010 |
| | srlv | 00_0110 |
| | sra | 00_0011 |
| | srav | 00_0111 |
| | add | 10_0000 |
| | addu | 10_0001 |
| | sub | 10_0010 |
| | subu | 10_0011 |
| | and | 10_0100 |
| | or | 10_0101 |
| | xor | 10_0110 |
| | nor | 10_0111 |
| | slt | 10_1010 |
| | sltu | 10_1011 |

| Type | Name | opC(Ins[31:26]) |
|---|---|---|
| I | beq | 00_0100 |
| | bne | 00_0101 |
| | lw | 10_0011 |
| | sw | 10_1011 |
| | | |
| | addi | 00_1000 |
| | addiu | 00_1001 |
| | | |
| | andi | 00_1100 |
| | ori | 00_1101 |
| | xori | 00_1110 |
| | | |
| | slti | 00_1010 |
| | sltiu | 00_1011 |
| | lui | 00_1111 |

结果是地址

8 SW

运算性

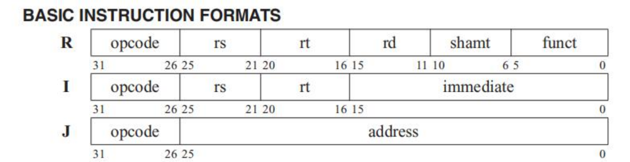
000

| Type | Name | opC(Ins[31:26]) | funC |
|---|---|---|---|
| J | jr | 00_0000 | 00_1000 |
| | jump | 00_0010 | |
| | jal | 00_0011 | |

NOTE:
The opC of R-Type instruction is 6'b00_0000

Minisys is a subset of MIPS32.

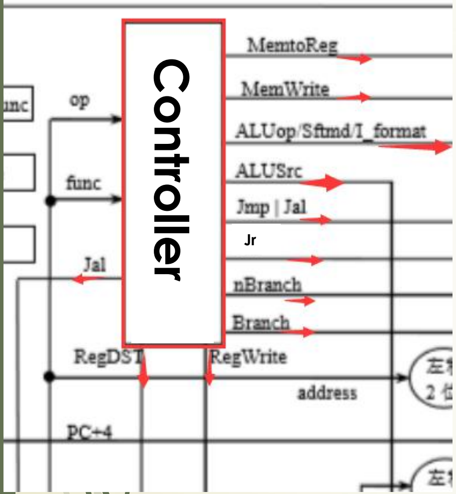
MIPS_Green_Sheet.pdf



# Controller continued

**BASIC INSTRUCTION FORMATS**

| | | | | | | |
|---|---|---|---|---|---|---|
| R | opcode (31–26) | rs (25–21) | rt (20–16) | rd (15–11) | shamt (10–6) | funct (5–0) |
| I | opcode (31–26) | rs (25–21) | rt (20–16) | immediate (15–0) | | |
| J | opcode (31–26) | address (25–0) | | | | |




```
input[5:0]   opcode;        // instruction[31..26]
input[5:0]   Function_opcode;          // instructions[5..0]
output       Jr;                      // 1 indicates the instruction is "jr", otherwise it's not "jr"
output       Jmp;           //  1 indicate the instruction is "j", otherwise it's not
output       Jal;                     //  1 indicate the instruction is "jal", otherwise it's not
output       Branch;        //  1 indicate the instruction is "beq" , otherwise it's not
output       nBranch;       //  1 indicate the instruction is "bne", otherwise it's not
output       RegDST;          // 1 indicate destination register is "rd",otherwise it's "rt"
output       MemtoReg;             //  1 indicate read data from memory and write it into register
output       RegWrite;     //  1 indicate write register, otherwise it's not
output       MemWrite;             //  1 indicate write data memory, otherwise it's not
output       ALUSrc;          // 1 indicate the 2nd data is immidiate (except "beq","bne")
output       I_format;      //  1 indicate the instruction is I-type but isn't  “beq","bne","LW" or "SW"
output       Sftmd;         //  1 indicate the instruction is shift instruction
//  if the instruction is R-type or I_format, ALUOp is 2'b10; if the instruction is  “beq”  or  “bne “, ALUOp is 2'b01;
//  if the instruction is  “lw”   or  “sw “, ALUOp is 2'b00;
output[1:0]  ALUOp;
```

# Controller continued

组合逻辑

"**Jr**" is used to identify whether the instruction is jr or not.

**Jr** =((Function_opcode==6'b001000)&&(Opcode==6'b000000)) ? 1'b1 : 1'b0;

| opCode | 001101 | 001001 | 100011 | 101011 | 000100 | 000010 | 000000 |
|---|---|---|---|---|---|---|---|
| Instruction | ori | addiu | lw | sw | beq | j | R-format |
| RegDST | 0 | 0 | 0 | x | x | x | 1 |

"**RegDST**" is used to determine the destination in the register file which is determined by rd(1) or rt(0)

**R_format** = (Opcode==6'b000000)? 1'b1:1'b0;

**RegDST** = R_format;

| opCode | 001xxx | 000000 | 100011 | 101011 | 000011 | 000010 | 000000 |
|---|---|---|---|---|---|---|---|
| Instruction | I-format | jr | lw | sw | jal | j | R-format |
| RegWrite | 1 | 0 | 1 | x | 1 | x | 1 |

"**RegWrite**" is used to determine whether to write registe(1) or not(0).

**RegWrite** = (R_format || Lw || Jal || I_format) && !(Jr)

可能读了之前
滞留的数据

# Controller continued

| Type | Name | opC(Ins[31:26]) |
|---|---|---|
| I | beq | 00_0100 |
| | bne | 00_0101 |
| | lw | 10_0011 |
| | sw | 10_1011 |
| | | |
| | addi | 00_1000 |
| | addiu | 00_1001 |
| | | |
| | andi | 00_1100 |
| | ori | 00_1101 |
| | xori | 00_1110 |
| | | |
| | slti | 00_1010 |
| | sltiu | 00_1011 |
| | lui | 00_1111 |

"**I_format**" is used to identify if the instruction is I_type(except for beq, bne, lw and sw). e.g. addi, subi, ori, andi…

**I_format** = (Opcode[5:3]==3'b001)?1'b1:1'b0;

| Instruction | ALUOp |
|---|---|
| lw | 00 |
| sw | 00 |
| beq,bne | 01 |
| R-format | 10 |
| I-format | 10 |

"**ALUOp**" is used to code the type of instructions described in the table on the left hand.

**ALUOp** = {(R_format || I_format),(Branch || nBranch)};

| Type | Name | funC(ins[5:0]) |
|---|---|---|
| R | sll | 00_0000 |
| | sllv | 00_0100 |
| | srl | 00_0010 |
| | srlv | 00_0110 |
| | sra | 00_0011 |
| | srav | 00_0111 |

**"Sftmd " is used to identify whether the instruction is shift cmd or not.**

**Sftmd** = (((Function_opcode==6'b000000)||(Function_opcode==6'b000010)

||(Function_opcode==6'b000011)||(Function_opcode==6'b000100)

||(Function_opcode==6'b000110)||(Function_opcode==6'b000111))
&& R_format)? 1'b1:1'b0;

# Practice1

- 1. Complete the Controller
- 2. Verify its function by simulation according to the following table.

TIPS: Minisys1Assemblerv2.2 could help to generate the corresponding instructions

| time(ns) | opcode | function_opcode | instruction |
|---|---|---|---|
| 0 | 6'h00 | 6'h20 | add rd,rs,rt //RegDST=1，RegWrite=1，ALUSrc=0，ALUOp=10 |
| 200 | 6'h00 | 6'h08 | jr rs //RegDST=1，RegWrite=0，ALUSrc=0，ALUOp=10，jr=1， |
| 400 | 6'h08 | 6'h08 | addi rt,rs,imm //RegDST=0，RegWrite=1，ALUSrc=1， I_format=1 |
| 600 | 6'h23 | 6'h08 | lw rt,imm(rs) //RegDST=0，RegWrite=1，ALUSrc=1，ALUOp=00，MemtoReg=1 |
| 800 | 6'h2b | 6'h08 | sw rt,imm(rs) //RegDST=0，RegWrite=0，ALUSrc=1，ALUOp=00，MemtoReg=0, MemWrite=1 |
| 1050 | 6'h04 | 6'h08 | beq rs,rt,label //RegDST=0，RegWrite=0，ALUSrc=0，ALUOp=01，Branch=1 |
| 1250 | 6'h05 | 6'h08 | bne rs,rt,label //RegDST=0，RegWrite=0，ALUSrc=0，ALUOp=01，Branch=0, nBranch=1 |
| 1500 | 6'h02 | 6'h08 | j label //RegDST=0，RegWrite=0，ALUSrc=0，ALUOp=00，Branch=0, nBranch=0, Jmp=1 |
| 1700 | 6'h03 | 6'h08 | jal label //RegDST=0，RegWrite=1，ALUSrc=0，ALUOp=00，Branch=0, nBranch=0, Jmp=0, Jal=1 |
| 1950 | 6'h00 | 6'h02 | srl rd,rt,shamt //RegDST=1，RegWrite=1，ALUSrc=0，ALUOp=10，sftmd=1 |

Minisys1Assemblerv2.2(An Asambler on Minisys) could be found at below address:
https://sakai.sustech.edu.cn/portal/site/d50211ea-1586-4344-9d92-a6c42eb7f4e0/tool/b47e4c13-4220-4f61-b881-a211abee2a96?panel=Main

# Tips: a reference to build a testbench

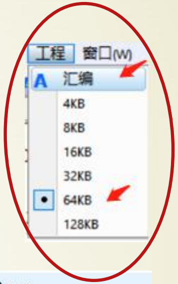
```
module control32_tb
	//reg type variables are use for binding with input ports
	reg [5:0] Opcode,Function_opcode;
	//wire type variables are use for binding with output ports
	wire [1:0] ALUOp;
	wire Jr,RegDST,ALUSrc,MemtoReg,RegWrite,MemWrite,Branch,nBranch,Jmp,Jal,I_format,Sftmd;

	//instance the module "control32", bind the ports
	control32 c32
	(Opcode,Function_opcode,
	Jr,Branch,nBranch,Jmp,Jal,
	RegDST,MemtoReg,RegWrite,MemWrite,
	ALUSrc,ALUOp,Sftmd,I_format);

	initial begin
		//an example: #0 add $3,$1,$2.  get the machine code of 'add $3,$1,$2'
			// step1: edit the assembly code, add "add $3,$1,$2"
			// step2: open the assembly code in Minisys1A assembler, do the assembly procession
			// step3: open the "output/prgmips32.coe" file, find the related machine code of 'add $3,$1,$2'
		//in "0x00221820", 'Opcode' is 6'h00,'Function_opcode' is 6'h20
		Opcode = 6'h00;
		Function_opcode = 6'h20;

		#200 //...
	end
endmodule
```

Tips:
The codes on Page 6 is another chooice

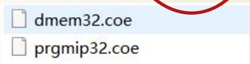
# How To Use "Minisys1Assemblerv2.2 "

- Step1. **Open** the assembly **source file**


- Step2. "工程"-》"**64KB**" (the size of Instruction memory and data memory)
  -》"**A 汇编**"




- Step3. The coe files could be found at the sub-directory: "**output**"
  - The initial data of data memory could be found in file "dmem32.coe"
  - The machine code of Minisys instruction could be found in the file "**prgmip32.coe**"

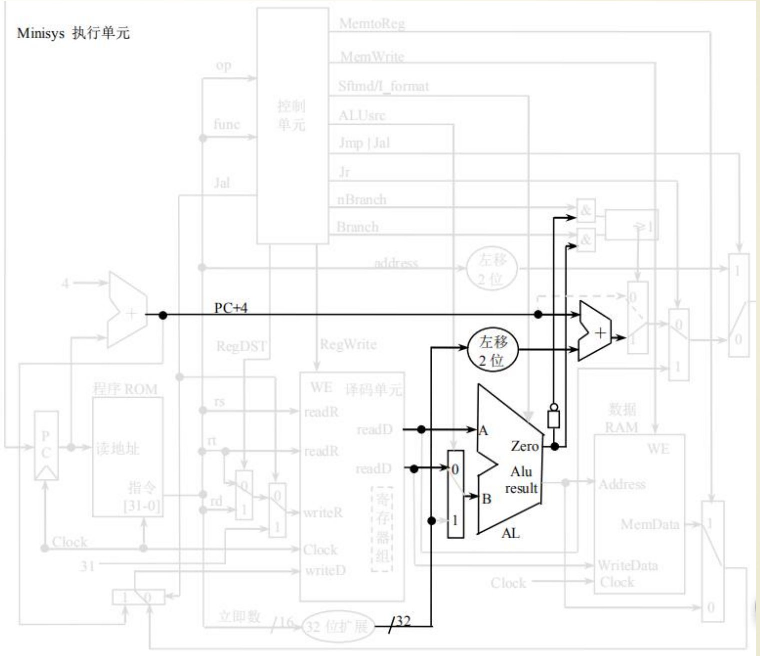



Following screenshot is an example, the machine code is recorded in **hexadecimal**.

```
memory_initialization_radix = 16;
memory_initialization_vector =
34010001,
34020002,
34030003,
34040004,
34050005,
34060006,
34070007,
34080008,
34090009,
340a000a,
```

# ALU

- Determine the function and the inputs and outputs
- A MUX for operand selection
- **Operation**
  - ALU_control
  - Operation
    - **Arithmetic and Logic** calculation
    - **Shift** calculation
    - **Special** calculation (slt,lui)
    - **Address** calculation



# Inputs Of ALU

| Instruction | ALUOp |
|---|---|
| lw | 00 |
| sw | 00 |
| beq,bne | 01 |
| R-format | 10 |
| I-format | 10 |

```
module Executs32 ( );
// from decoder
    input[31:0]  Read_data_1;         //the source of Ainput
    input[31:0]  Read_data_2;         //one of the sources of Binput
    input[31:0]  Sign_extend;         //one of the sources of Binput

// from ifetch
    input[5:0]   Function_opcode;     //instructions[5:0]
    input[5:0]   Opcode;              //instruction[31:26]
    input[4:0]   Shamt;               // instruction[10:6], the amount of shift bits
    input[31:0]  PC_plus_4;           // pc+4

// from controller
    input[1:0]   ALUOp;    //{ (R_format || I_format) , (Branch || nBranch) }
    input        ALUSrc;   // 1 means the 2nd operand is an immedite (except beq，bne)
    input        I_format; // 1 means I-Type instruction except beq, bne, LW, SW
    input        Sftmd;    // 1 means this is a shift instruction
    input        Jr;       // 1 means this is a jr instruction
```

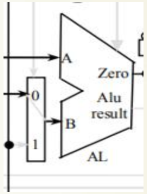
# Outputs And Variable of ALU continued

```
output Zero;                                  // 1 means the ALU_reslut is zero, 0 otherwise
output[31:0]    reg ALU_Result;               // the ALU calculation result
output[31:0]    Addr_Result;                  // the calculated instruction address
```

```
wire[31:0]    Ainput,Binput;                  // two operands for calculation

wire[5:0]     Exe_code;     // use to generate ALU_ctrl. (I_format==0) ? Function_opcode : { 3'b000 , Opcode[2:0] };
wire[2:0]     ALU_ctl;      // the control signals which affact operation in ALU directely
wire[2:0]     Sftm;         // identify the types of shift instruction, equals to Function_opcode[2:0]

reg[31:0]     ALU_output_mux;                 // the result of arithmetic or logic calculation
reg[31:0]     Shift_Result;                   // the result of shift operation

wire[32:0]  Branch_Addr;      // the calculated address of the instruction, Addr_Result is Branch_Addr[31:0]
```

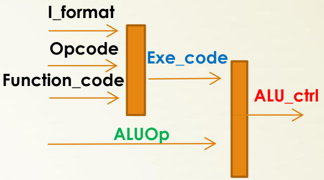
# The Selection On Operand2

- Two operands: Ainput and Binput.
- **Binput** is the output of 2-1 MUX:
  - “**Sign_extend**” and “**Read_data_2**” are from decoder.
  - The output of the MUX is determined by “**ALUSrc**”.




```
input[31:0]  Read_data_1; // from decoder
input[31:0]  Read_data_2; // from decoder
input[31:0]  Sign_extend;  // from decoder
input          ALUSrc;         // from controller, 1 means the operand2 is an immediate

assign  Ainput = Read_data_1;
assign  Binput = (ALUSrc == 0) ? Read_data_2 : Sign_extend[31:0];
```

# ALU_ctrl




- Design :
  - lots of operations need to be processed in ALU
  - to reduce the burden of the controller, the controller and ALU produce control signals which affect the alu operation together
- Implements :
  - **ALUOp**(1st level control signal) : **generated by Controller** ( the basic relationship between instruction and operation )
    - bit1 to identify if the instruction is R_format/ I_format, otherwise means neither
    - bit0 to identify if the instruction is beq/ bne, otherwise means neither
    - **ALUOp = { (R_format | | I_format) , (Branch | | nBranch) }**
  - **Exe_code**(2nd level control signal) : **generated by ALU**
    - according to the instruction type( I-format or not)
    - **Exe_code = (I_format==0) ? Function_opcode : { 3'b000 , Opcode[2:0] };**

    tips: **Opcode** is **instruction[31:26]**, function_opcode is **instruction[5:0]**

    tips: **I_format** is **1** means this is the **I-type instruction except beq,bne,lw and sw**.
  - **ALU_ctrl** : **generated by ALU** based on **ALUOp** and **Exe_code**

    **specify most of the operation details in ALU**

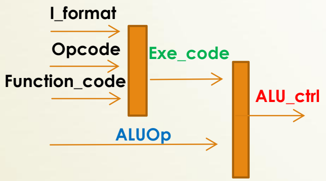
| Instruction | ALUOp |
|---|---|
| lw | 00 |
| sw | 00 |
| beq,bne | 01 |
| R-format | 10 |
| I-format | 10 |

# ALU_ctrl continued

| Exe_code[3..0] | ALUOp[1..0] | ALU_ctl[2..0] | 指令助记符 |
|---|---|---|---|
| 0100 | 10 | 000 | and,andi |
| 0101 | 10 | 001 | or,ori |
| 0000 | 10 | 010 | add,addi |
| xxxx | 00 | 010 | lw, sw |
| 0001 | 10 | 011 | addu, addiu |
| 0110 | 10 | 100 | xor,xori |
| 0111 | 10 | 101 | nor,lui |
| 0010 | 10 | 110 | sub, slti |
| xxxx | 01 | 110 | beq，bne |
| 0011 | 10 | 111 | subu, sltiu |
| 1010 | 10 | 111 | slt |
| 1011 | 10 | 111 | sltu |

**ALUOp** =
**{ (R_format | | I_format) , (Branch | | nBranch) }**

**Exe_code** =
**(I_format==0) ?**
**Function_opcode : { 3'b000 , Opcode[2:0] };**
// Function_opcode equals to Instruction[5:0]
// Opcode equals to Instruction[31:26]




```
assign   ALU_ctl[0] = (Exe_code[0] | Exe_code[3]) & ALUOp[1];

assign   ALU_ctl[1] = ((!Exe_code[2]) | (!ALUOp[1]));

assign   ALU_ctl[2] = (Exe_code[1] & ALUOp[1]) | ALUOp[0];
```

# ALU_ctrl continued

**Type1**

**The same operation in ALU with different operand source**

sometimes the instructions share the same calculation operation but with different operand source, such as “and” and “andi”, “addu” and “addui”.

the same operation but different operand source: **ALU_ctrl** is same

**add** vs **addi**
**addu** vs **addiu**
**and** vs **andi**
**or** vs **ori**
**xor** vs **xori**
**slt** vs **sltu** vs **sltiu**

| Exe_code[3..0] | ALUOp[1..0] | ALU_ctl[2..0] | 指令助记符 |
|---|---|---|---|
| 0100 | 10 | 000 | and,andi |
| 0101 | 10 | 001 | or,ori |
| 0000 | 10 | 010 | add,addi |
| xxxx | 00 | 010 | lw, sw |
| 0001 | 10 | 011 | addu, addiu |
| 0110 | 10 | 100 | xor,xori |
| 0111 | 10 | 101 | nor,lui |
| 0010 | 10 | 110 | sub, slti |
| xxxx | 01 | 110 | beq，bne |
| 0011 | 10 | 111 | subu, sltiu |
| 1010 | 10 | 111 | slt |
| 1011 | 10 | 111 | sltu |

# ALU_ctrl continued

- **Type2**

**The same operation in ALU with different destination**

- The **ALU_ctrl** code is same for both "lw","sw" and "add" :
  - the operation of "lw" and "sw" in ALU is calcuation the address based on the base address and offset which is same as in "add" operation.

| Exe_code[3..0] | ALUOp[1..0] | ALU_ctl[2..0] | 指令助记符 |
|---|---|---|---|
| 0100 | 10 | 000 | and,andi |
| 0101 | 10 | 001 | or,ori |
| 0000 | 10 | 010 | add,addi |
| xxxx | 00 | 010 | lw, sw |
| 0001 | 10 | 011 | addu, addiu |
| 0110 | 10 | 100 | xor,xori |
| 0111 | 10 | 101 | nor,lui |
| 0010 | 10 | 110 | sub, slti |
| xxxx | 01 | 110 | beq，bne |
| 0011 | 10 | 111 | subu, sltiu |
| 1010 | 10 | 111 | slt |
| 1011 | 10 | 111 | sltu |

# ALU_ctrl continued

- Type2 continued

**The same operation in ALU with different destination**

- "beq", "bne" vs "sub":
  - the destionation of "beq" and "bne" is addr_reslut not the "ALU_reslut"
- "subu" vs "slt" , "sltu"
  - the destination of "slti" and "sltiu" is Zero. **I_format is used here to distinguish these two types**
- "sub" vs "slti", "subu" vs "sltiu":
  - same as upper instruction, **Function_opcode**(3)=1 of slt and sltu **could be used as distinguishment**

Beq $R, $S, Loop.

| Exe_code[3..0] | ALUOp[1..0] | ALU_ctl[2..0] | 指令助记符 |
|---|---|---|---|
| 0100 | 10 | 000 | and,andi |
| 0101 | 10 | 001 | or,ori |
| 0000 | 10 | 010 | add,addi |
| xxxx | 00 | 010 | lw, sw |
| 0001 | 10 | 011 | addu, addiu |
| 0110 | 10 | 100 | xor,xori |
| 0111 | 10 | 101 | nor,lui |
| 0010 | 10 | 110 | sub, slti |
| xxxx | 01 | 110 | beq，bne |
| 0011 | 10 | 111 | subu, sltiu |
| 1010 | 10 | 111 | slt |
| 1011 | 10 | 111 | sltu |

# ALU_ctrl continued

- **Type3**

**Some** instructions' **ALU_ctrl code** is the same as others, but with **different operation** in ALU. For these instructions, make sure they can be identified to avoid wrong operations:

  - **shift** instrucitons: could be identified by the input port "**sftmd**"
  - **lui** : whose ALU_ctrl code is the same as "nor", but could be identified by "**I_format**"
  - jr : could be identified by the input port "jr", not excute in ALU
  - j : could be identified by the input port "jmp", not excute in ALU
  - **jal** : could be identified by the input port "jal", excute in both ifetch and ALU

# Practice2-1 Arithmatic and Logic calculation

- Complete the following code according to the table on the right hand

```
reg[31:0] ALU_output_mux;
always @(ALU_ctl or Ainput or Binput)
begin
case(ALU_ctl)
    3'b000:ALU_output_mux =? ? ?
    3'b001:ALU_output_mux =? ? ?
    3'b010:ALU_output_mux =? ? ?
    3'b011:ALU_output_mux =? ? ?
    3'b100:ALU_output_mux =? ? ?
    3'b101:ALU_output_mux =? ? ?
    3'b110:ALU_output_mux =? ? ?
    3'b111:ALU_output_mux =? ? ?
    default:ALU_output_mux = 32'h00000000;
endcase
end
```

| Exe_code[3..0] | ALUOp[1..0] | ALU_ctl[2..0] | 指令助记符 |
|---|---|---|---|
| 0100 | 10 | 000 | and,andi |
| 0101 | 10 | 001 | or,ori |
| 0000 | 10 | 010 | add,addi |
| xxxx | 00 | 010 | lw, sw |
| 0001 | 10 | 011 | addu, addiu |
| 0110 | 10 | 100 | xor,xori |
| 0111 | 10 | 101 | nor,lui |
| 0010 | 10 | 110 | sub, slti |
| xxxx | 01 | 110 | beq，bne |
| 0011 | 10 | 111 | subu, sltiu |
| 1010 | 10 | 111 | slt |
| 1011 | 10 | 111 | sltu |

# Shift Operation

- There are 6 shift instructions, listed in the table on the left hand below
- Ainput, Binput/shamt are the operand of shift operation

| sftm[2:0] | process |
|---|---|
| 3'b000 | sll rd, rt, shamt |
| 3'b010 | srl rd, rt, shamt |
| 3'b100 | sllv rd, rt, rs |
| 3'b110 | srlv rd, rt, rs |
| 3'b011 | sra rd, rt, shamt |
| 3'b111 | srav rd, rt, rs |
| other | not shift |

```
input[4:0]   Shamt;      // from ifetch, instruction[10:6], its value is shift amount
input        Sftmd;      // from controller, 1 means this is a shift instruction
input[5:0]   Function_opcode;     //r-type instruction, instruction[5:0]
reg[31:0]    Shift_Result;              //the result of shift operation
wire[2:0]    Sftm;

assign Sftm = Function_opcode[2:0];    //the code of shift operations
```

C=12=8+4

# Practice2-2 Shift Operation

Complete the following code, taking the table on the left hand as reference

0011 20 1100011 0000

| sftm[2:0] | process |
|---|---|
| 3'b000 | sll rd, rt, shamt |
| 3'b010 | srl rd, rt, shamt |
| 3'b100 | sllv rd, rt, rs |
| 3'b110 | srlv rd, rt, rs |
| 3'b011 | sra rd, rt, shamt |
| 3'b111 | srav rd, rt, rs |
| other | not shift |

```
always @* begin  // six types of shift instructions
      if(Sftmd)
            case(Sftm[2:0])
              3'b000:Shift_Result = Binput << Shamt;          //Sll rd,rt,shamt  00000
              3'b010:Shift_Result = ???;                      //Srl rd,rt,shamt  00010
              3'b100:Shift_Result = Binput << Ainput;         //Sllv rd,rt,rs 000100
              3'b110:Shift_Result = ???;                      //Srlv rd,rt,rs 000110
              3'b011:Shift_Result = ???;                      //Sra rd,rt,shamt 00011
              3'b111:Shift_Result = ???;                      //Srav rd,rt,rs 00111
              default:Shift_Result = Binput;
            endcase
      else
            Shift_Result = Binput;
   end
```

# Get the Output of ALU

The operations of ALU include:

- 1） do the setting type instruction （slt、sltu、slti and sltiu ）
  - get ALU_output_mux, and set the value of the output port "Zero"
- 2） do the lui operation
- 3） do the shift operation
  - get shift amount, do the shift operation according to the "Sftm", set the value of the output port "Shift_Result"
- 4） do the basic arithmetic and logic calculation
  - get ALU_output_mux, set its value to the output port "ALU_result"

**Tips:** Exe_code[3..0]、ALUOp[1..0] and ALU_ctl[2..0] are used to identify the types of operation

# Get the Output of ALU continued

```
always @* begin
        //set type operation (slt, slti, sltu, sltiu)
        if(((ALU_ctl==3'b111) && (Exe_code[3]==1))||((ALU_ctl[2:1]==2'b11) && (I_format==1))
                ALU_Result = (Ainput-Binput<0)?1:0;

      //lui operation
      else if((ALU_ctl==3'b101) && (I_format==1))
                        ALU_Result[31:0]={Binput[15:0],{16{1'b0}}};

      //shift operation
      else if(Sftmd==1)
                        ALU_Result = Shift_Result  ;

      //other types of operation in ALU (arithmatic or logic calculation)
      else
                        ALU_Result = ALU_output_mux[31:0];
   end
```

# Practice2-3 Complete ALU

- The values of “**Addr_result**” and “**zero**” are still not detemined.

  Complete the ALU code.

  - “**zero**” is a signal used by “**ifetch**” to determine whether to use the value of “**Addr_reslut**” to update **PC** register or not.

    TIPS: Minisys only support “beq” and “bne” in the conditional jump instruction.

  - “**Addr_result**” is calculated by ALU when the instruction is “**beq**” or “**bne**”.

    TIPS: Addr_reslut should be the sum of pc+4 and the immediate in the instruction.

# Practice2-4 Function Verification on ALU

Build a testbench to verify the function of ALU. (TIPS: p11 could be a reference)
Take the testcases described in bellow table as reference
More testcases are suggested for function verification

| Time (ns) | Instruction | A input | B input | Results(includes 'Zero') |
|---|---|---|---|---|
| 0 | add 0 | 0x5 | 0x6 | ALU_Result = 0x0000_000b, Zero=1'b0 |
| 200 | addi 4 | 0xffff_ff40 | 0x3 | ALU_Result = 0xffff_ff43, Zero=1'b0 |
| 400 | and 8 | 0x0000_00ff | 0x0000_0ff0 | ALU_Result = 0x0000_00f0, Zero=1'b0 |
| 600 | sll C | 0x0000_0002 | 0x3 | ALU_Result = 0x0000_0010, Zero=1'b0 |
| 800 | lui 10 | 0x0000_0040 | 0x10 (16) | ALU_Result = 0x0040_0000, Zero=1'b0 |
| 1000 | beq 14 | The value of Ainput is same with that of Binput. Zero = 1'b1<br>**Addr_Result**: the address of current instruction is 0x0000_0006, the offset is 0x0000_0004, 'Addr_Result' should be 0x0000_000a | | |

18