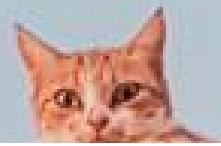


Computer Organization

Lab11 CPU(3) ALU, Clock, CPU-TOP

Topics

- CPU (3)
  - ALU
  - Clock
  - Build a Single Cycle CPU

Minisys - A subset of MIPS32

Type	Name	funC(ins[5:0])
R	sll	00_0000
	srl	00_0010
	sllv	00_0100
	srlv	00_0110
	sra	00_0011
	srav	00_0111
	jr	00_1000
	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
	slti	00_1010
	sltiu	00_1011
	andi	00_1100
	ori	00_1101
	xori	00_1110
	lui	00_1111

Type	Name	opC(Ins[31:26])
J	jump	00_0010
	jal	00_0011

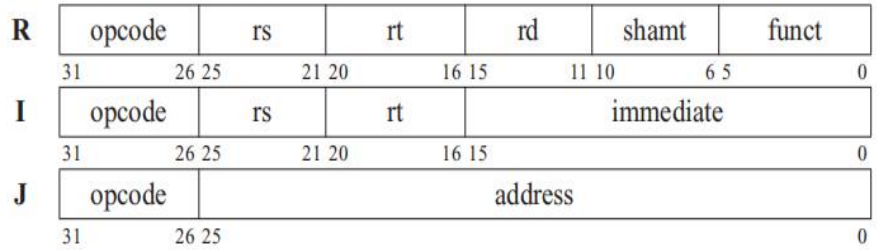
MIPS_Green_Sheet.pdf

NOTE:

Minisys is a subset of MIPS32.

The **opC** of **R-Type** instruction is **6'b00_0000**

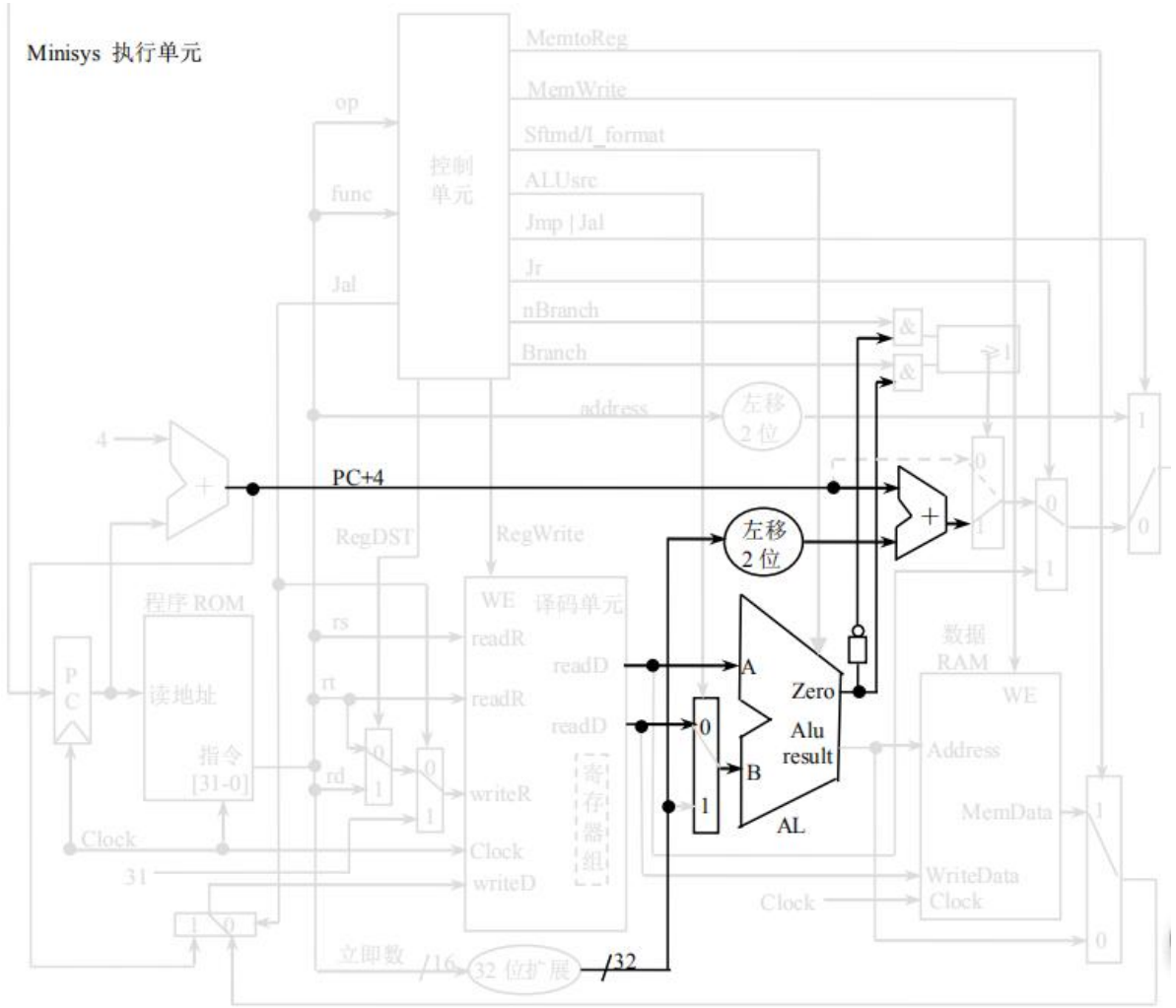
BASIC INSTRUCTION FORMATS



ALU

Q: Is the ALU a commbinatorial logic and sequential logic?

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



Tips: follow design is a reference ONLY, not required.

Inputs Of ALU

```
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]   Opcode;               //instruction[31:26]
    input[5:0]   Function_opcode;      //instructions[5:0]
    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 immediate (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
```

Outputs And Variable of ALU continued

Q1: What's the destination of the outputs of ALU?

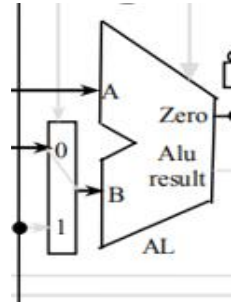
```
output[31:0]    reg ALU_Result;     //  the ALU calculation result
output          Zero;               // 1 means the ALU_reslut is zero, 0 otherwise
output[31:0]    Addr_Result;        //  the calculated instruction address
```

Q2: How to determine the data type of following variable?

```
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]       Shift_Result;               //  the result of shift operation

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

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

The Selection On Operand2

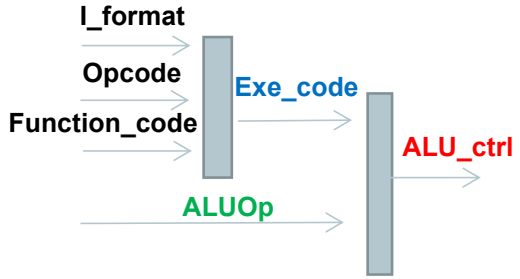
- Two operands: Ainput and **Binput**.
- **Binput** is the output of 2-1 MUX:
  - "**Sign_extend**" and "**Read_data_2**" are from Decoder.
  - The output of the MUX is determined by "**ALUSrc**".




```
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 generation

- 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(1):
  - **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) }**

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

      // "I_format" is used to identify if the instruction is I_type(except for beq, bne, lw and sw).

ALU_ctrl generation continued

- Implements(2) :
  - **Exe_code**(2nd level control signal): according to the instruction type(I-format or not):

Exe_code = (I_format==0) ?

function_opcode :

{ 3'b000 , Opcode[2:0] };

Tips

1) I_format is **1** means this is the **I-type instruction except beq,bne,lw and sw**.

2) Opcode is **instruction[31:26]**

3) function_opcode is **instruction[5:0]**

Q. Could the 'Exe_code' be generated by Controller or by ALU? What's your choic?

Type	Name	funC(ins[5:0])
R	sll	00_0000
	srl	00_0010
	sllv	00_0100
	srlv	00_0110
	sra	00_0011
	srav	00_0111
	jr	00_1000
	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
		I-Format
	addi	00_1000
	addiu	00_1001
	slti	00_1010
	sltiu	00_1011
	andi	00_1100
	ori	00_1101
	xori	00_1110
	lui	00_1111

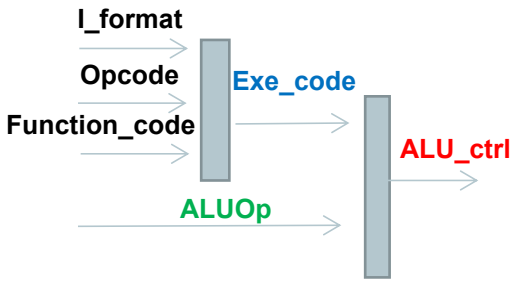
ALU_ctrl generation continued

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

- **ALU_ctrl**： based on **ALUOp** and **Exe_code,** **specify most of the operation details in ALU**

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

Exe_code =
(I_format==0) ?
Function_opcode :
{ 3'b000 , Opcode[2:0] };




```
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 usage

➢ **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 usage continued

- **Type2:** The same **operation** in ALU with different **destination**

The **ALU_ctrl** code is same(**3'b010**) for both "**lw**","**sw**", "**add**" and " **andi**":

- *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 usage continued

- **Type2 continued:** The same **operation** in ALU with different **destination**
  - "beq", "bne" vs "sub" (destionation):
    - "beq" and "bne" : Addr_reslut
    - "sub" : "ALU_reslut"
  - "subu" vs "slt" , "sltu" (destionation)
    - "slti" and "sltiu" :Zero.

    I_format is used here to distinguish these two types
  - "sub" vs "slti", "subu" vs "sltiu":

    same as upper instructions,
    Function_opcode(3)=1 of slt and sltu **could be used as distinguishment**

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 usage 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", not excute in ALU

Practice1-1: Arithmatic and Logic calculation

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

```
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

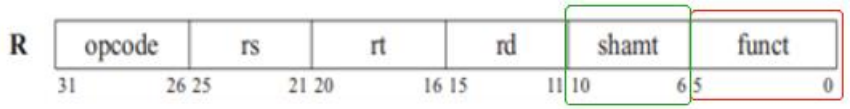
***Tips**: While ALU_ctrl is 3'b101, One of the implements is to execute only 'nor', make other procedure do the 'lui'*

Shift Operation

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

There are 6 shift instructions, listed in the table on the left hand。

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[5:0]   Function_opcode;      //from IFetch,R-type instruction, instruction[5:0]
input        Sftmd;                // from Controller, 1 means this is a shift instruction
wire[2:0]    Sftm;
assign Sftm = Function_opcode[2:0];   //the code of shift operations

reg[31:0]    Shift_Result;              //the result of shift operation
```

Practice1-2: Shift Operation

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

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    00010
               3'b110:Shift_Result = ???;                      //Srlv rd,rt,rs    00110
               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) Execute the **setting** type instructions (**slt**、**sltu**、**slti** and **sltiu**)
   - get **ALU_output_mux**, and set the value of the **output port "ALU_result"**
2) Execute the **lui** operation
   - get result of "lui" execution, and set the value to the **output port "ALU_result"**
3) Execute the **shift** operation
   - get "**Shift_Result**", set its value to the **output port "ALU_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*

Practice 1-3: Determine the output “ALU_Result ”

Complete the following code according to the code annotation

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

    //lui operation
    else if((ALU_ctl==3'b101) && (I_format==1))
                ALU_Result[31:0]=  /*to be completed*/;

    //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
```

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

Practice 1-4: determine the output “Addr_result ” and “Zero”

The values of “**Addr_result**” and “**Zero**” are waiting to be detemined.

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

- “**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 branch type instruction.

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

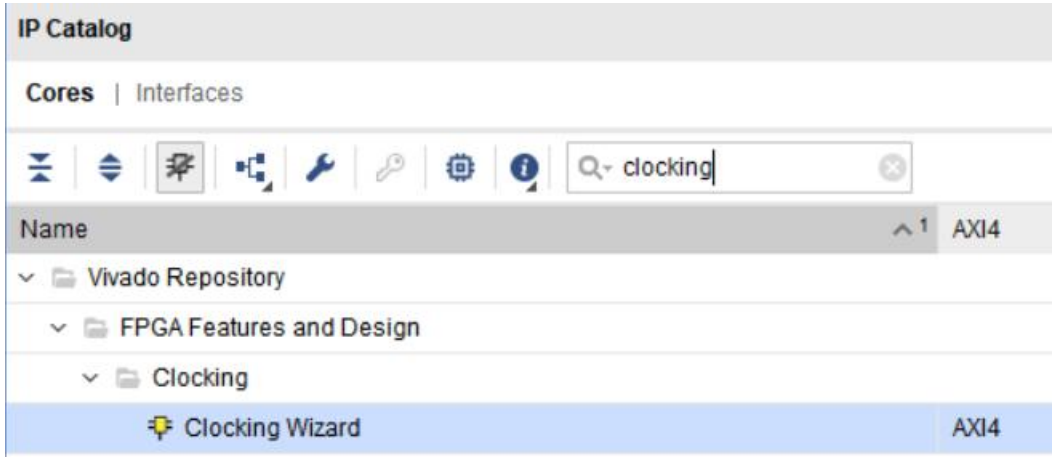
TIPS: **Addr_reslut** should be the sum of **pc+4**(could be get from **PC_plus_4**) and the **immediate** in the instruction.

Practice 1-5: Function Verification on ALU

Build a testbench to verify the function of ALU.
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	0x5	0x6	ALU_Result = 0x0000_000b, Zero=1'b0
200	addi	0xffff_ff40	0x3	ALU_Result = 0xffff_ff43, Zero=1'b0
400	and	0x0000_00ff	0x0000_0ff0	ALU_Result = 0x0000_00f0, Zero=1'b0
600	sll	0x0000_0002	0x3	ALU_Result = 0x0000_0010, Zero=1'b0
800	lui	0x0000_0040	0x10 (16)	ALU_Result = 0x0040_0000, Zero=1'b0
1000	beq	The value of Ainput is same with that of Binput. Zero = 1'b1 Depends on your design **Addr_Result**: should be the sum of **pc+4**(could be get from **PC_plus_4**) and the **immediate** in the instruction		

Clock



1. Add **PPL clock IP core** to generate a clock

    1. The clock on the Minisys development board is **100Mhz**
    2. **A clock of 23Mhz is needed for the single clock cpu**

Functional Verification

1） Create a verilog design module to perform instance and port binding on the IP core.

2） Set up testbench to verify whether the output signal is a 23Mhz clock signal while the input signal is 100Mhz.

Clock continued

Component Name: cpuclk

Clocking Options | Output Clocks | Port Renaming | PLLE2 Settings | Summary

Clock Monitor

- [ ] Enable Clock Monitoring

Primitive

- ( ) MMCM
- (•) PLL

Clocking Features

- [x] Frequency Synthesis
- [ ] Minimize Power
- [x] Phase Alignment
- [ ] Dynamic Reconfig
- [ ] Safe Clock Startup

Jitter Optimization

- (•) Balanced
- ( ) Minimize Output Jitter
- ( ) Maximize Input Jitter filtering

Clocking Options | Output Clocks | Port Renaming | PLLE2 Settings

The phase is calculated relative to the active input clock.

Output Clock	Port Name	Output Freq (MHz)	
		Requested	Actual
[x] clk_out1	clk_out1	23.000	23.000

Component Name: cpuclk

Clocking Options | Output Clocks | Port Renaming | PLLE2 Settings | Summary

Enable Optional Inputs / Outputs for MMCM/PLL

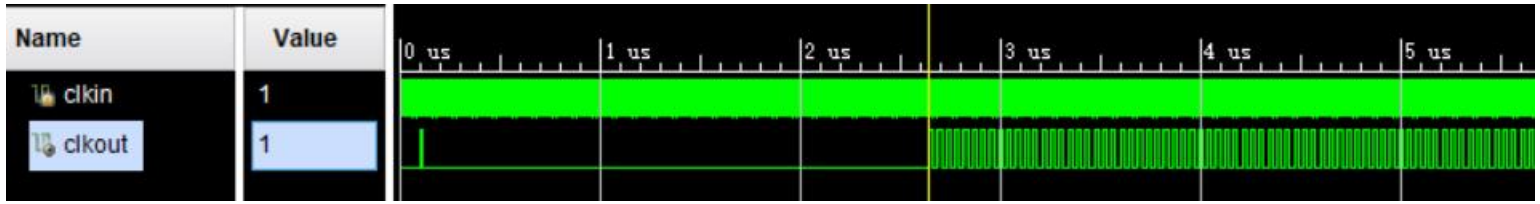
- [ ] reset
- [ ] power_down
- [ ] locked

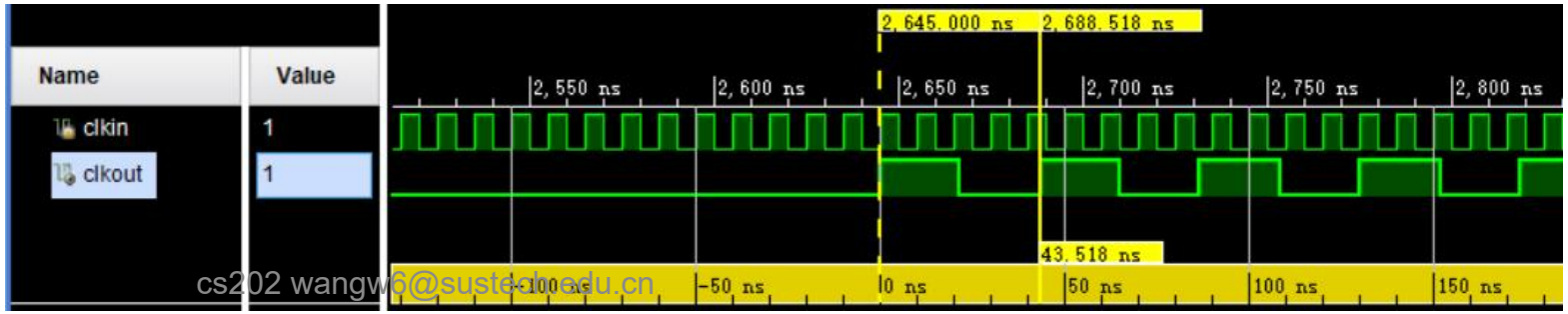
Reset Type: (•) Active High ( ) Active Low

The Function Verification of “cpuclk”

NOTE: The output of IP core 'cpuclk' need to work for a period of time to achieve stability.

```
// a reference testbench for 'cpuclk'
module cpuclk_tb(    );
    reg clkin;
    wire clkout;
    cpuclk clk1( .clk_in1(clkin), .clk_out1(clkout) );
    initial       clkin = 1'b0;
    always #5 clkin=~clkin;
endmodule
```



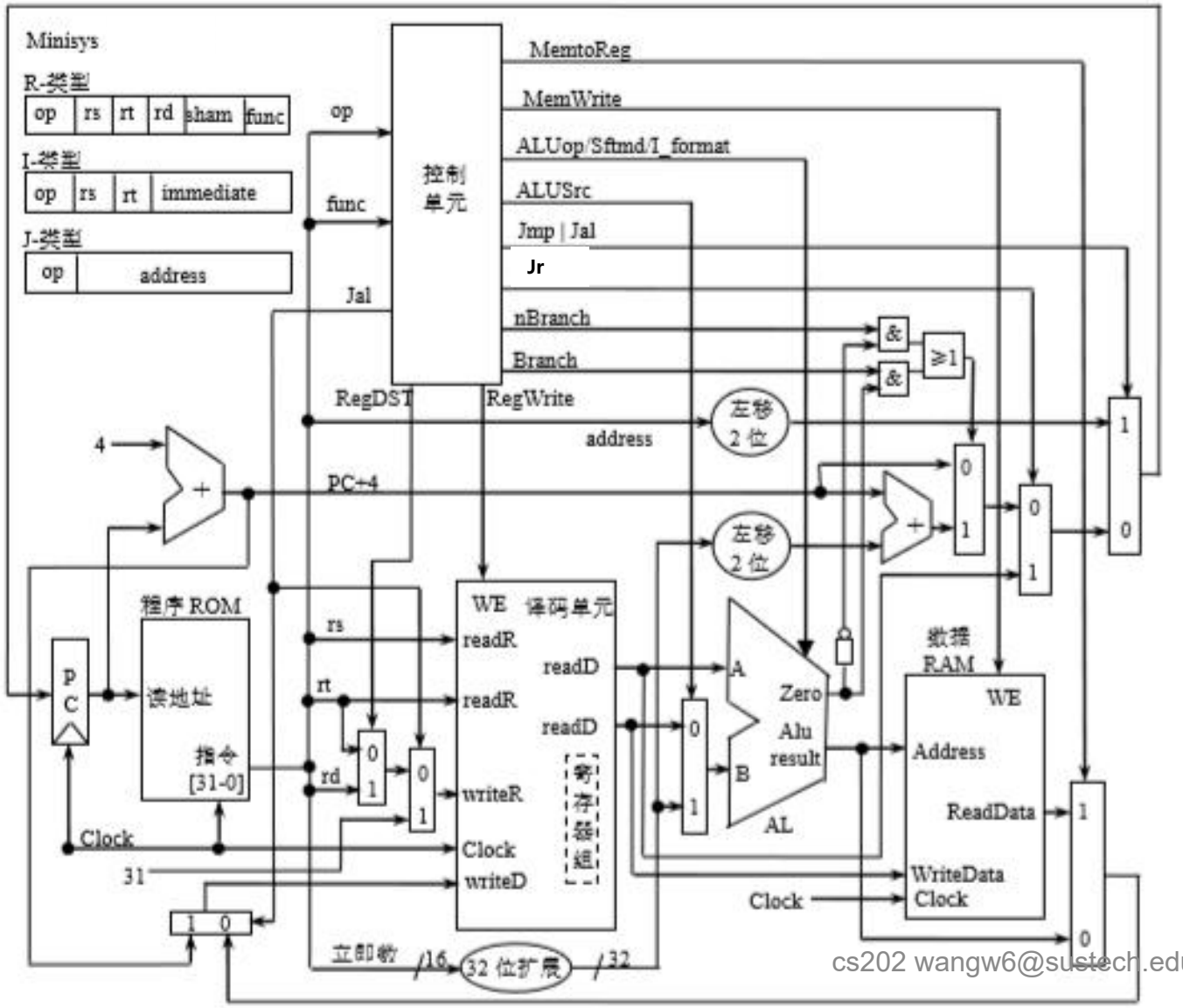


Single Cycle CPU

- Determine input and output ports of the CPU
  - **Clock signal, reset signal** //which sub-module use clock, reset
  - **Input port**
  - **Output port**
- Determine the sub-module inside the CPU
  - **Clock module**
  - **IFetch, Controller, Decoder, ALU, Dmemory,** (IO processing...
- Build CPU top-level module
  - **Notes the relationship on the sub-modules inside the CPU, especially the control signal, data,address, instruction between sub-modules.**
  - **Build CPU by using Structural Design in verilog or Block Design in Vivado.**

NOTE: Case sensitivity in Verilog syntax.

Single Cycle CPU continued



Create a CPU **top** module

1) Instantiating the **clock**, **IFetch, Controller, Decoder**, **ALU**, and **DataMemory** units.

2) Complete the inter-module **connection** inside the CPU and the connection between ports of sub-module and the CPU ports.