



Computer Organization

Lab10 CPU Design(2)

Data Path(2)
IFetch, Dmemory, ALU



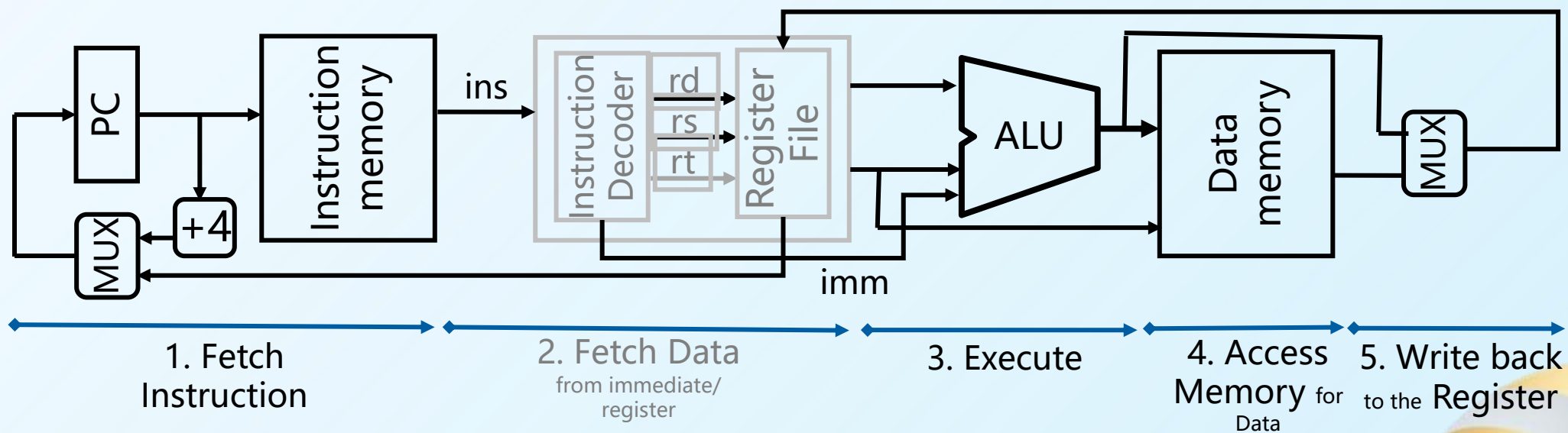
Topic

➤ CPU(2) -DataPath (2)

➤ Data-Memory

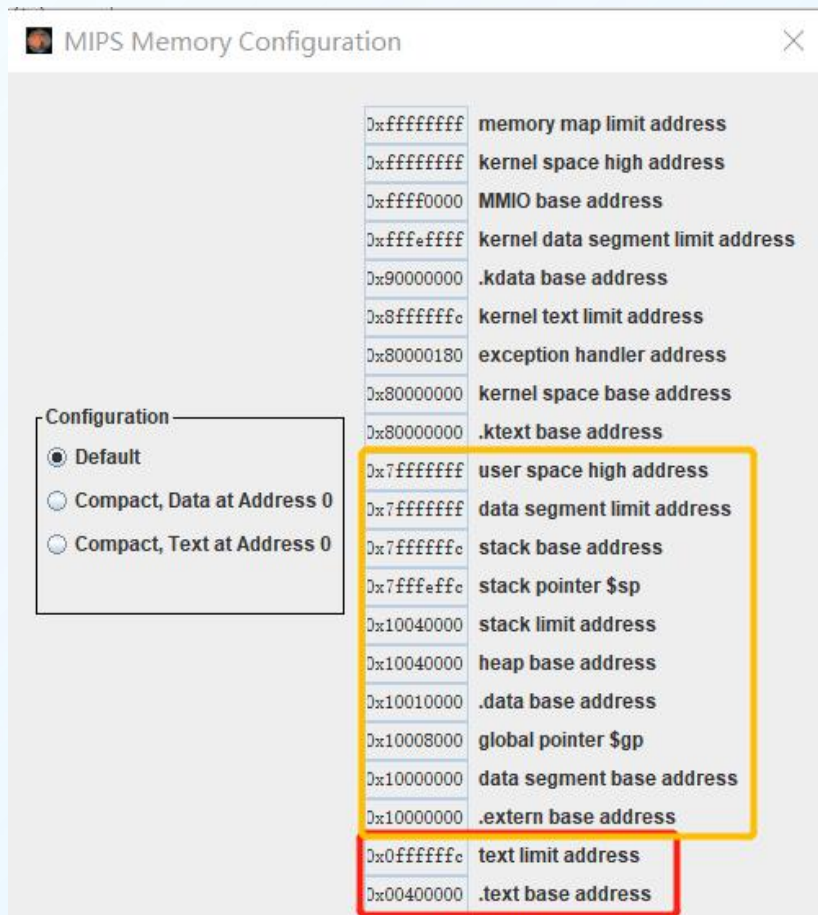
➤ IFetch

➤ ALU





Von Neumann structure vs Harvard structure



*Von Neumann architecture:
data and instruction share the same memory.*

The **Von Neumann architecture**, also known as the **Princeton architecture**, is centered around the fact that **data and instructions are mixed and addressed together, meaning that their data and instructions are stored at different addresses in the same memory**, and their widths must be the same. When transmitting instructions and data, both **share the same program bus and data bus**, so instructions and data cannot be operated simultaneously, and they can only be executed **sequentially**.

The data and instruction memory implemented in **Mars** (a simulator of MIPS32) is **Von Neumann architecture** (data and instruction share the same memory)

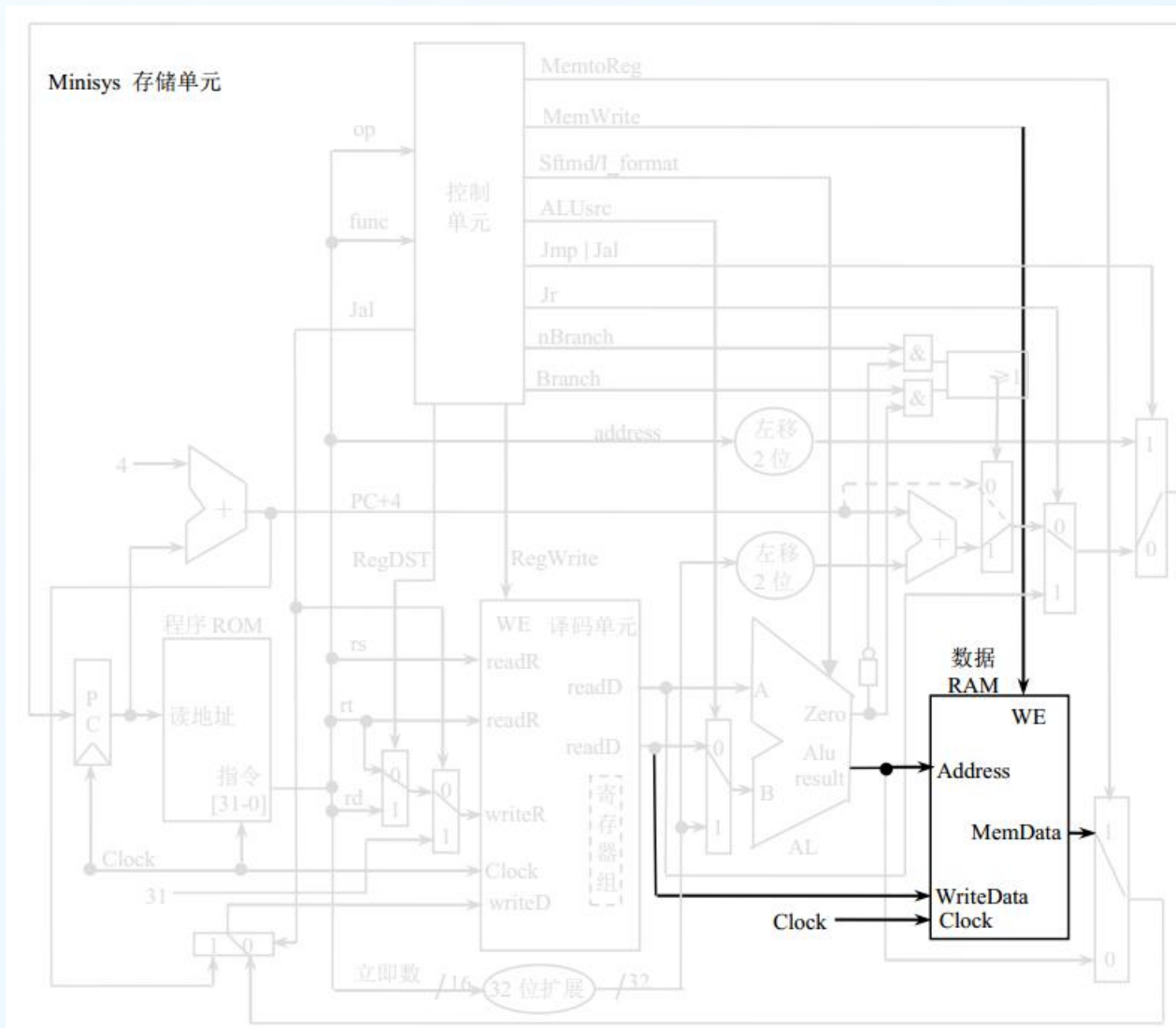
The characteristic of the **Harvard architecture** is that **data and instructions are stored separately in two different memories: data memory and program memory**.

The buses of the two are used separately, and the buses are divided into data bus and address bus of program memory, as well as data bus and address bus of data memory. And the width of instructions and data can be different, while instructions and data can work in **parallel**.

In the next few pages of the courseware, we will introduce the implementation of the **Harvard architecture**.

TIPS: Harvard architecture here is just a reference, not requirement!

Data-Memory



```
module dmemory32(readData,address,  
writedata,memWrite,clock);
```

```
input clock;    // 'Clock' signal
```

```
/* used to determine to write the memory unit or not,  
in the left screenshot its name is 'WE' */
```

```
input memWrite;
```

```
// the 'Address' of memory unit which is to be  
read/written
```

```
input[31:0] address;
```

```
// data to be written to the memory unit
```

```
input[31:0] writeData;
```

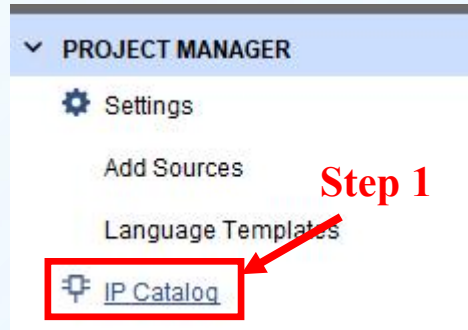
```
/* data to be read from the memory unit, in the left  
screenshot its name is 'MemData' */
```

```
output[31:0] readData;
```



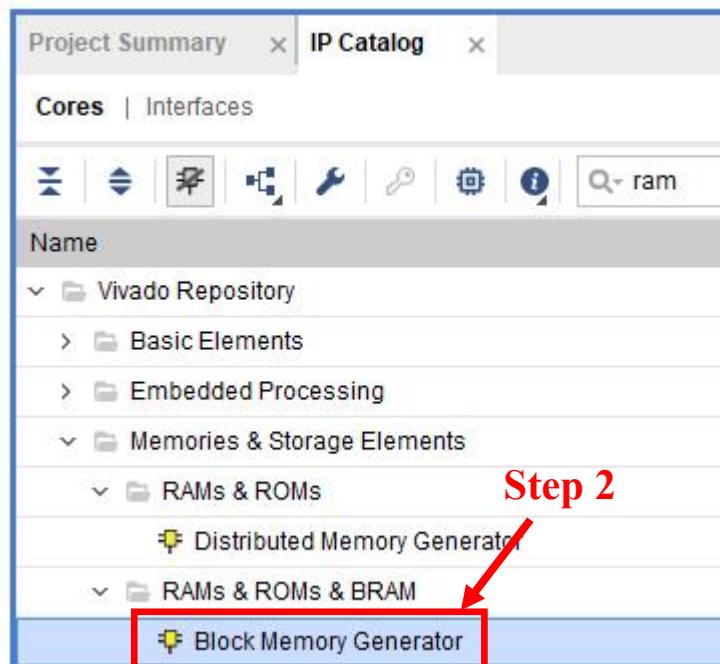
Using IP core: Block Memory

Using the IP core 'Block Memory' of Xilinx to implement the Data-memory.



Import the IP core in vivado project

1) in **“PROJECT MANAGER”** window
click **“IP Catalog”**



2) in **“IP Catalog”** window

> Vivado Repository

> Memories & Storage Elements

> RAMs & ROMs & BRAM

> **Block Memory Generator**



Customize Memory IP core

Component Name

Basic | Port A Options | Other Options | Summary

Interface Type: ☐ Generate address interface with 32 bits

Memory Type: ☐ Common Clock

ECC Options

ECC Type: ☐ Error Injection Pins:

Write Enable

☐ Byte Write Enable

Byte Size (bits):

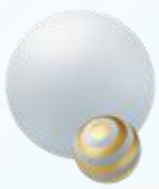
Algorithm Options

Defines the algorithm used to concatenate the block RAM primitives. Refer datasheet for more information.

Algorithm: Primitive:

Customize memory IP core

- 1) Component Name: **RAM**
- 2) Basic settings:
 - Interface Type: **Native**
 - Memory Type: **Single-port RAM**
 - ECC options: **no ECC check**
 - Algorithm options: **Minimum area**



Customize Memory IP core continued

Component Name RAM

Basic Port A Options Other Options Summary

Memory Size

Write Width 32 Range: 1 to 4608 (bits)

Read Width 32

Write Depth 16384 Range: 2 to 1048576

Read Depth 16384

Operating Mode Write First

Enable Port Type Always Enabled

Port A Optional Output Registers

☐ Primitives Output Register ☐ Core Output Register

☐ SoftECC Input Register ☐ REGCEA Pin

Port A Output Reset Options

☐ RSTA Pin (set/reset pin) Output Reset Value (Hex) 0

☐ Reset Memory Latch Reset Priority CE (Latch or Register Enable)

3) PortA Options settings:

➤ Data read and write **bit width**:

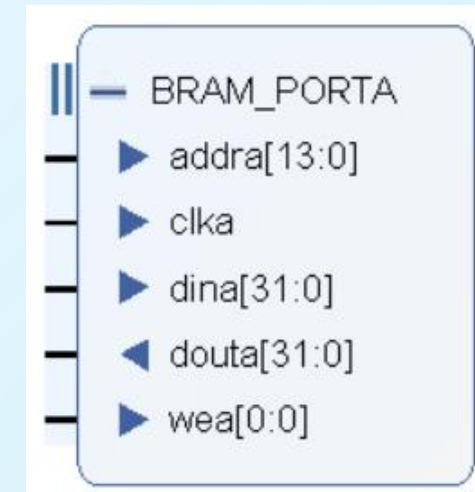
32 bits (4Byte)

➤ Write/Read **Depth**: **16384 (64KB)**

➤ Operating Mode: **Write First**

➤ Enable Port Type: **Always Enabled**

➤ PortA Optional Output Registers: **NOT SET**

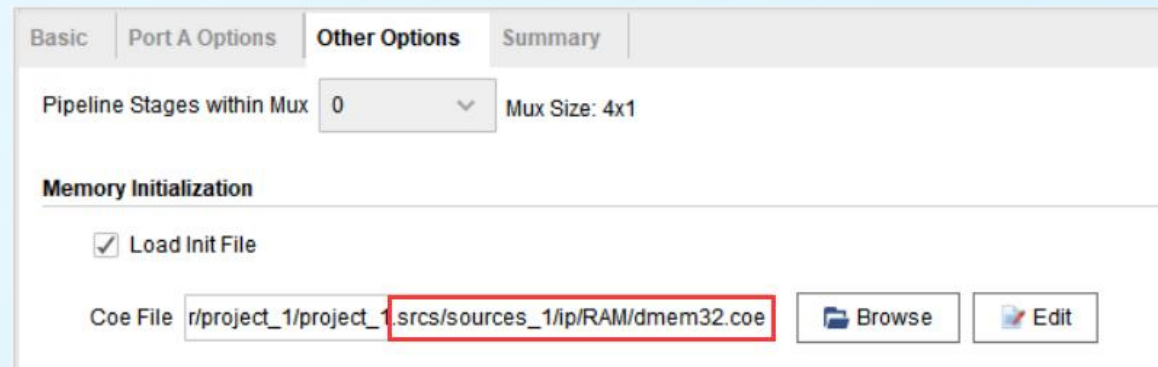
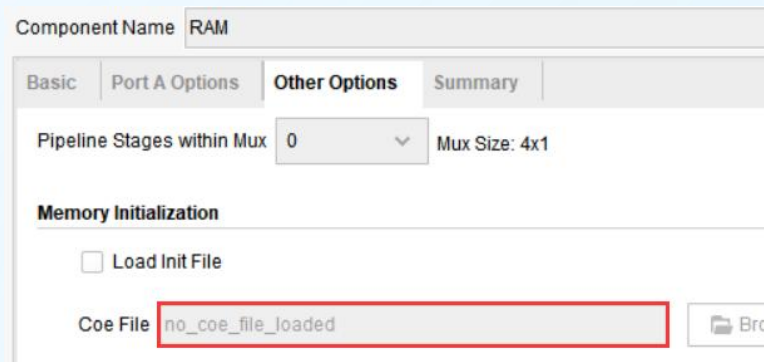




Customize Memory IP core continued

4) Other Options settings:

- 1. When **specifying the initialization file** for customize the RAM on the 1st time, the IP core RAM just customized **WITHOUT initial file** and **corresponding path**, so set it to **no initial file** when creating RAM.
- 2. **After** the RAM IP core created
 - 2-1. **COPY** the initialization file **dmem32.coe** to **projectName.srcs/sources_1/ip/ComponentName**. (“projectName.srcs” is under the project folder, “componentName” here is ‘RAM’)
 - 2-2. Double-click the newly created RAM IP core, **RESET** it with the **initialization file**, select the **dmem32.coe** file that has been in the directory of projectName.srcs/sources_1/ip/RAM.



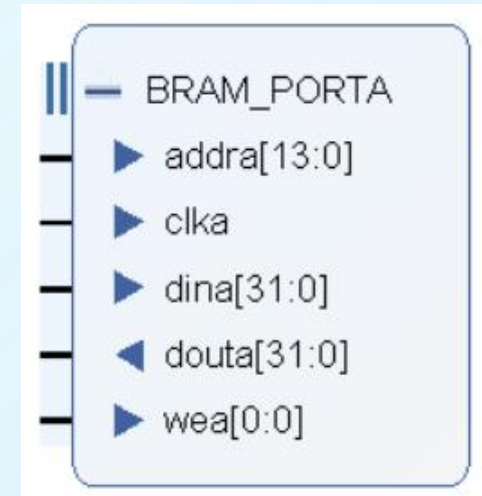
Tips: “dmem32.coe” file could be found in the directory “labs/lab10” of course blackboard site

Design Module With Memory IP Instanced

```
// Part of dmemory32 module
//Create a instance of RAM(IP core), binding the ports
RAM ram (
    .clka(clk),                // input wire clka
    .wea(memWrite),           // input wire [0 : 0] wea
    .addra(address[15:2]),     // input wire [13 : 0] addra
    .dina(writeData),         // input wire [31 : 0] dina
    .douta(readData)          // output wire [31 : 0] douta
);
```

*/*The 'clock' is from CPU-TOP, suppose its one edge has been used at the upstream module of data memory, such as IFetch, Why Data-Memroy DO NOT use the same edge as other module ? */*

```
assign clk = !clock;
```



Q: In the five stages of instruction processing, what operations must be arranged on the edge of the clock?
What's your design for a one-cycle CPU?

Function Verification by simulation

```
//The testbench module for dmemory32
module ramTb( );
reg clock = 1'b0;
reg memWrite = 1'b0;
reg [31:0] addr = 32'h0000_0010;
reg [31:0] writeData = 32'ha000_0000;
wire [31:0] readData;

dmemory32 uram
    (clock,memWrite,addr,writeData,readData);
always #50 clock = ~clock;

initial fork
    #120 memWrite = 1'b1;
    #200
        writeData = 32'h0000_00f5;
    #400
        memWrite = 1'b0;
    // ... to be completed
join

endmodule
```

NOTE:

Using bind port with name is Suggested!!

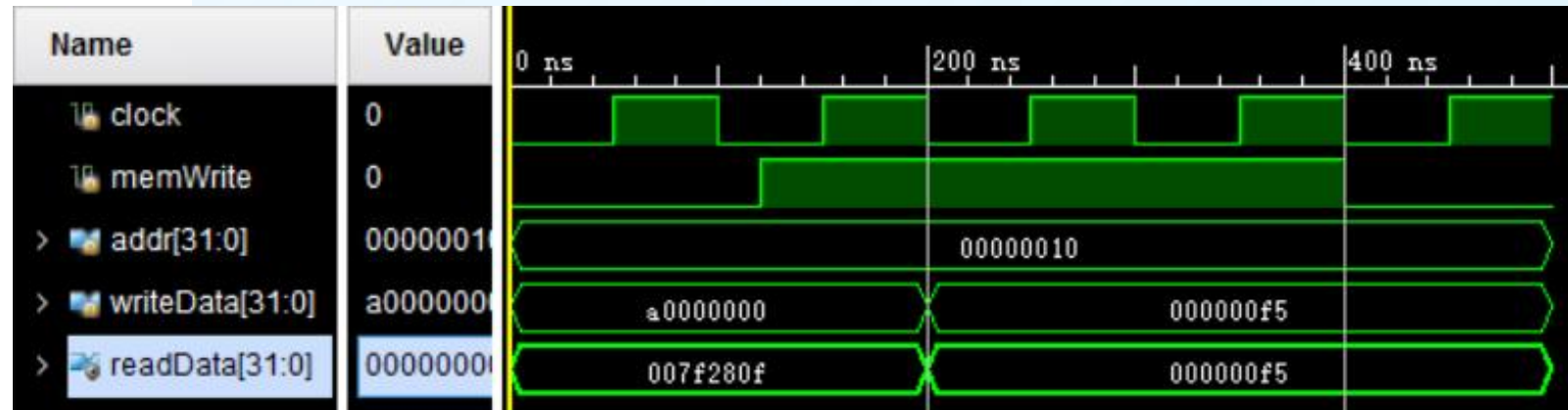
- 1) Set “**memWrite**” to 1'b0 means to read the data from the RAM unit identified by “**addr**”.
- 2) Set “**memWrite**” to 1'b1 and “**writeData**” to 0x0000_00f5 which means to write data 0xa000_00f5 to the RAM unit identified by “**addr**”.

Q1. While instance the module on page 4(module **dmemory32**(readData,address,writedata,memWrite,clock)) and using sequential binding as the testbench on the left hand, What will happen ?

Q2. While the data has been written to the RAM unit, would it be recorded to the initial data file(dmем32.coe)?

Function Verification by simulation continued

```
1 memory_initialization_radix = 16;
2 memory_initialization_vector =
3 007f2812,
4 007f2811,
5 007f2810,
6 007f2810,
7 007f280f,
8 00000001,
9 00000002,
10 00000003,
11 00000005,
12 00000006,
13 00000007,
14 0000ffff,
15 00000000,
16 00000000,
```



Q1: On which edge of clock does the read and write operations occur? posedge or negedge?

Q2: What's value will be get while read the memory according to the "addr" 0x0000_0020?
how about "addr" 0x0000_0016?

Tips: "dmem32.coe" file could be found in the directory "labs/lab10" of course blackboard site



Practice1

1. Build the data memory module.
2. Verify its function by simulation

(NOTE: The testbench on page 9 is JUST a reference)

- **Read** the values one by one from memory unit where are specified in the red box of the screenshot on the right hand.
- Write a word(value is 0x1000_0000) to the memory unit where is specified in the blue box of the screenshot on the right hand, then read it out.

3. List all the signals which are needed for data-memory module

```
1 memory_initialization_radix = 16;
2 memory_initialization_vector =
3 007f2812,
4 007f2811,
5 007f2810,
6 007f2810,
7 007f280f,
8 00000001,
9 00000002,
10 00000003,
11 00000005,
12 00000006,
13 00000007,
14 0000ffff,
15 00000000,
16 00000000,
17 00000000,
18 00000000,
19 00000000,
20 00000000,
21 00000000,
```

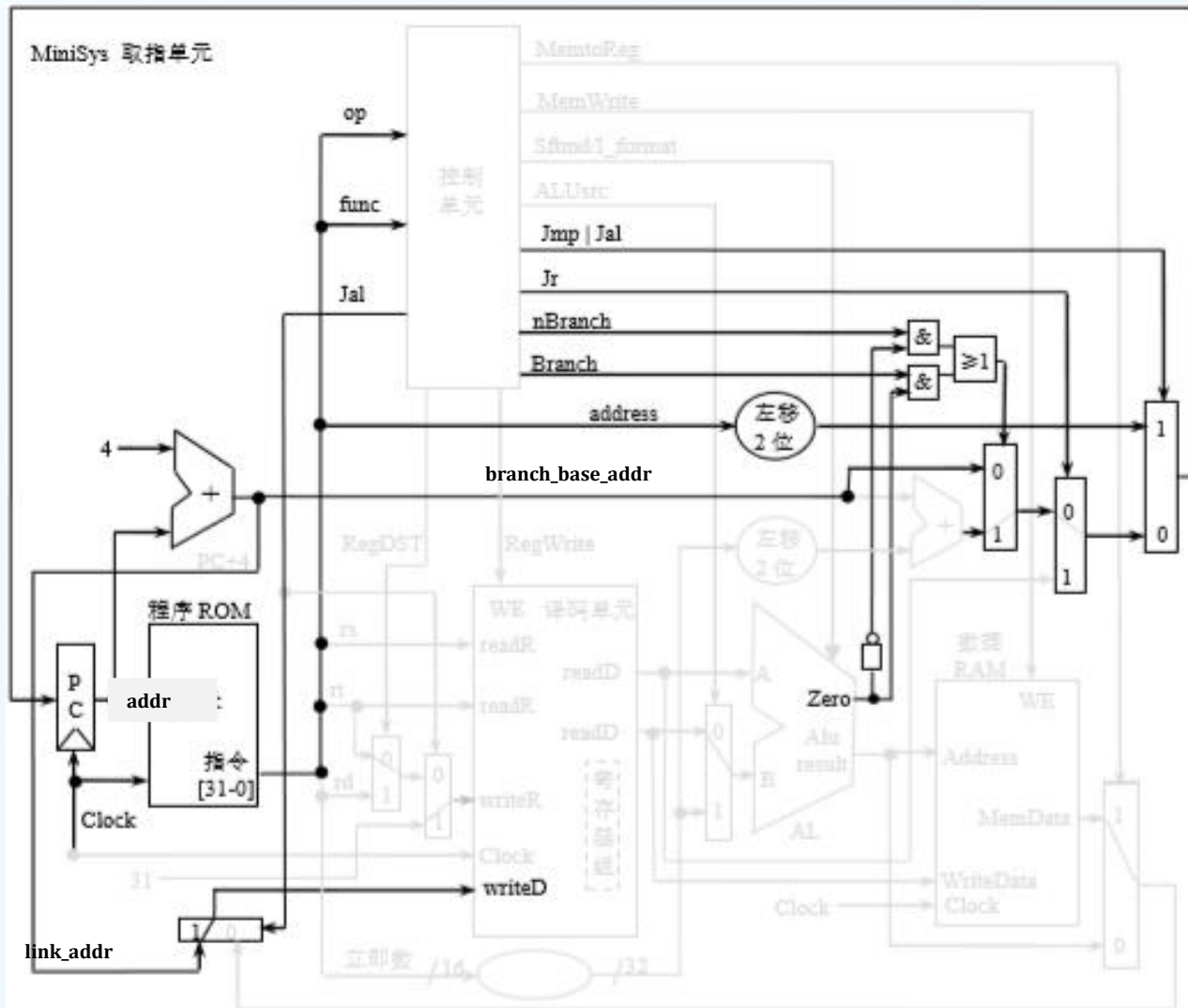
read these initial value

write this word with 0x1000_0000 then read it

name	from	to	bits	function
clock	CPU-TOP	Data Memory	1	data memory write is sensitive with its negedge
rdata	Data Memory	Decoder	32	the word read from the data memory and send to decoder
memoryWrite	Controllor	Data Memory	1	1'b1 means to write the memory unit, else means not to write
address	ALU	Data Memory	32	the address which is used to identify the memory unit to be read or written
...				

Tips: “**dmem32.coe**” file could be found in the directory “**labs/lab10**” of course blackboard site

Instruction Fetch



The function of Instruction Fetch module

- 1. **Store** the instructions(machine-code)
- 2. **Update** the value of the PC register
 - Reset
 - PC+4
 - Update the value of the PC register according to the jump instructions
 - branch(beq,bne) [I-type]
 - jal, j [J-type]
 - jr [R-type]
- 3. **Fetch** the instructions according to the value of the PC register



Using IP core As Instruction Memory

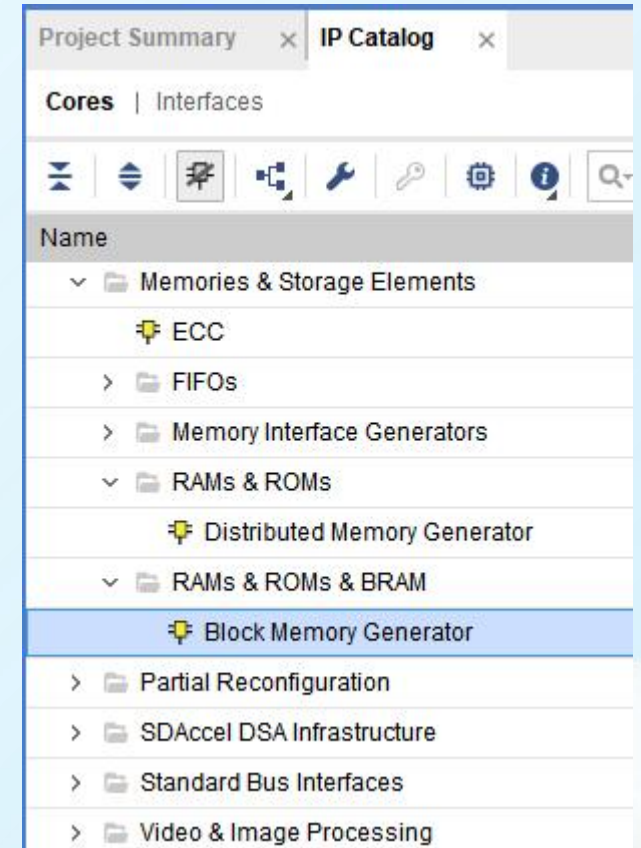
Step1: Find the IP core(Block Memory Generator) in IP Catalog

Step2: Customize the IP core

- set **name**(component name), **type**(ROM)
- set features of the ROM(**width** and **depth**), **operation mode** and **register output**
- set **initial file**

Step3: Generate the IP core, then it will be added to vivado project automatically

Tips: The setting steps of ROM IP core are almost same as which of the RAM IP core in Data-memory except the type is



Customize the IP core

Component Name: prgrom

Basic | Port A Options | Other Options | Summary

Interface Type: Native

Memory Type: Single Port ROM

ECC Options

ECC Type: No ECC

Error Injection Pins: Single Bit Error Injection

Write Enable

Byte Write Enable

Byte Size (bits): 9

Algorithm Options

Defines the algorithm used to concatenate the block RAM primitives. Refer datasheet for more information.

Algorithm: Minimum Area

Primitive: 8kx2

Component Name: prgrom

Basic | Port A Options | Other Options | Summary

Memory Size

Port A Width: 32 Range: 1 to 4608 (bits)

Port A Depth: 16384 Range: 2 to 1048576

The Width and Depth values are used for Read Operation in Port A

Operating Mode: Write First

Enable Port Type: Always Enabled

Port A Optional Output Registers

Primitives Output Register

Core Output Register

SoftECC Input Register

REGCEA Pin

Port A Output Reset Options

RSTA Pin (set/reset pin)

Output Reset Value (Hex): 0

Reset Memory Latch

Reset Priority: CE (Latch or Register Enable)

READ Address Change A

Read Address Change A

Component Name: prgrom

Basic | Port A Options | Other Options | Summary

Pipeline Stages within Mux: 0 Mux Size: 4x1

Memory Initialization

Load Init File

Coe File: no_coe_file_loaded

Fill Remaining Memory Locations

Remaining Memory Locations (Hex): 0

Structural/UniSim Simulation Model Options

Defines the type of warnings and outputs are generated when a read-write or write-write collision occurs.

Collision Warnings: All

Behavioral Simulation Model Options

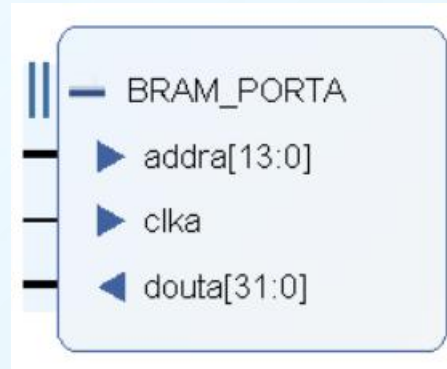
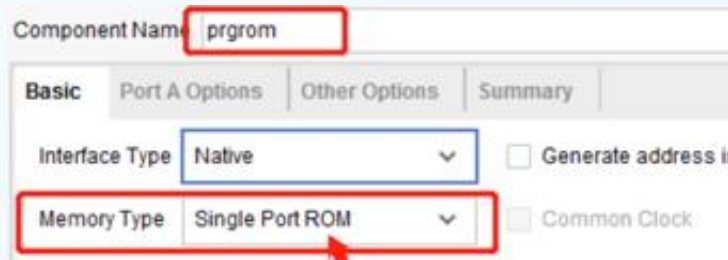
Disable Collision Warnings

Disable Out of Range Warnings

NOTE: set the init file of prgrom after this IP core has been added into vivado project.
Same steps as the RAM IP core used in Data-memory.



Instance the IP core



```
prgrom instmem(  
    .clka(clock),  
    .addra(PC[15:2]),  
    .douta(Instruction)  
);
```

In One Cycle CPU, the process of **getting instruction** should **happen** on the **posedge** of the clock. At this moment, IFetch module gets the instruction which is store at “**addra**” from the instruction memory “Instmem”

Q: **Why using PC[15:2] instead of PC[13:0] to bind with port “addra”?**

TIPS: The same reason as the address bus used in Data-memory

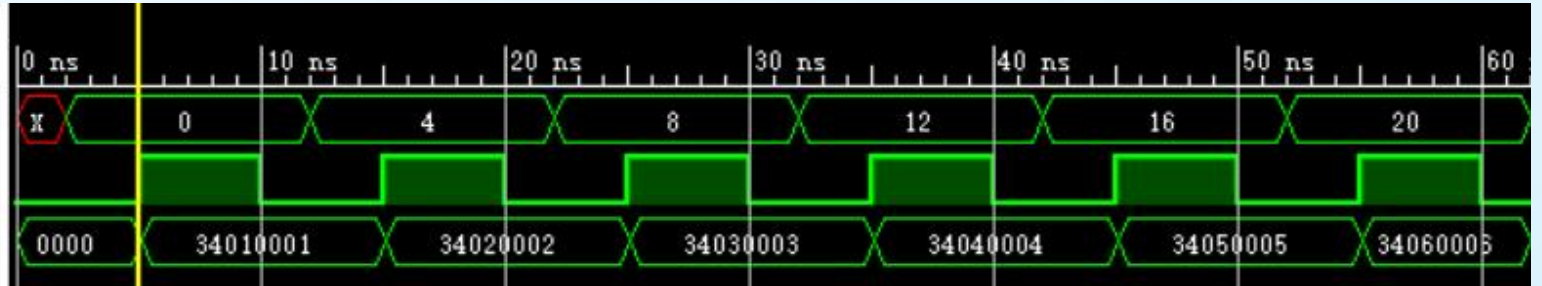
The Function Verification of “prgrom”

prgmip32.coe

Tips: “prgmip32.coe” file could be found in the directory “labs/lab10” of course blackboard site

```
1 memory_initialization_radix = 16;
2 memory_initialization_vector =
3 34010001,
4 34020002,
5 34030003,
6 34040004,
7 34050005,
8 34060006,
9 34070007,
10 34080008,
11 34090009,
12 340a000a,
13 340b000b,
14 340c000c,
```

Name	Value
PC[31:0]	0
clock	1
Instruction[31:0]	00000000



```
module prgrom_tb( ); //a reference for the testbench ?
    reg[31:0] PC;
    reg clock=1'b0;
    wire [31:0] Instruction;
    prgrom instmem(.clka(clock),.addra(PC[15:2]),.douta(Instruction));
    always #5 clock = ~clock;
    initial begin
        clock = 1'b0;
        #2 PC = 32'h0000_0000;
        repeat(5) begin
            #10 PC = PC+4;
            #10 $finish;
        end
    end
end
endmodule
```

- Read the “Instruction” from “douta” port of Instruction memory “prgrom” on every posedge of the “clock”.
- In this testcase, the value of 'PC' is added with 4 each time.
- Q: How many instructions would be fetched in this testbench ?



IFetch Module

```
module IFetc32(Instruction, branch_base_addr, link_addr,
clock, reset,
Addr_result, Read_data_1, Branch, nBranch, Jmp, Jal, Jr, Zero);

    output[31:0] Instruction;           // the instruction fetched from this module to Decoder and Controller
    output[31:0] branch_base_addr;    // (pc+4) to ALU which is used by branch type instruction
    output[31:0] link_addr;           // (pc+4) to Decoder which is used by jal instruction

//from CPU TOP
    input      clock, reset;           // Clock and reset
// from ALU
    input[31:0] Addr_result;           // the calculated address from ALU
    input      Zero;                   // while Zero is 1, it means the ALUresult is zero

// from Decoder
    input[31:0] Read_data_1;           // the address of instruction used by jr instruction

// from Controller
    input      Branch;                 // while Branch is 1,it means current instruction is beq
    input      nBranch;                // while nBranch is 1,it means current instruction is bnq
    input      Jmp;                    // while Jmp 1, it means current instruction is jump
    input      Jal;                    // while Jal is 1, it means current instruction is jal
    input      Jr;                     // while Jr is 1, it means current instruction is jr
```

Update the Value of the PC register

```
reg[31:0] PC, Next_PC;
```

```
always @* begin
```

```
    if(((Branch == 1) && (Zero == 1)) || ((nBranch == 1) && (Zero == 0))) // beq, bne
```

```
        Next_PC = ... // the calculated new value for PC
```

```
    else if(Jr == 1)
```

```
        Next_PC = ... // the value of $31 register
```

```
    else Next_PC = ... // PC+4
```

```
end
```

```
always @(... clock) begin
```

```
    if(reset == 1)
```

```
        PC <= 32'h0000_0000;
```

```
    else begin
```

```
        if((Jmp == 1) || (Jal == 1)) begin
```

```
            PC <= ...;
```

```
        end
```

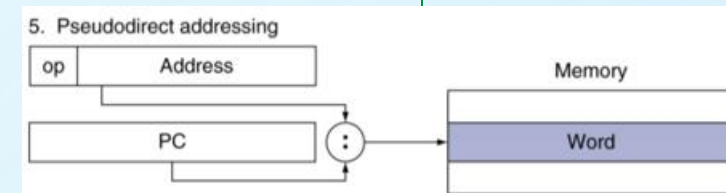
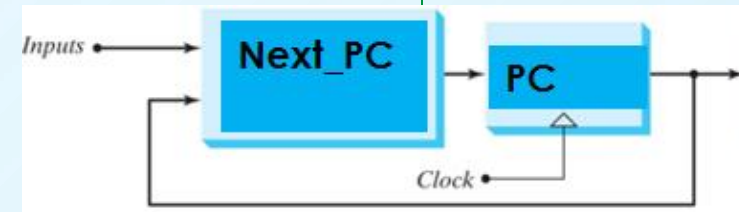
```
    else PC <= ...;
```

```
end
```

Q1: Complete the code to update 'Next_PC'

Q2: Could be 'PC' ready while read the 'prgrom'? Determine when to update the value of the PC register.

Q3: Is this Minisys ISA a Harvard structure or Von Neumann structure(take a look at the initial value of PC)



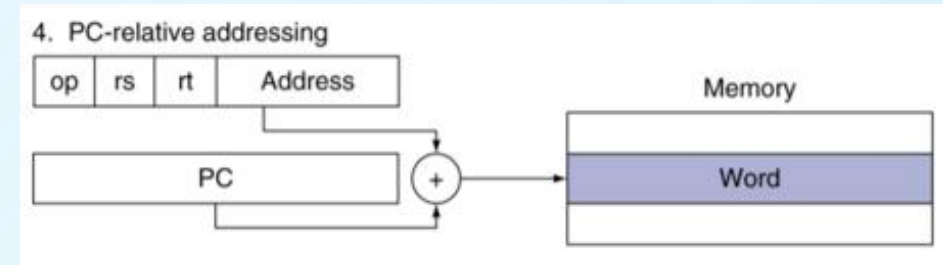
NOTES: The code here is JUST refence, NOT request.



Outputs of IFetch: Prepare for Decoder and ALU

output[31:0] **branch_base_addr**; // (pc+4) to ALU which is used by **branch** type instruction
output[31:0] **link_addr**; // (pc+4) to Decoder which is used by **'jal'** instruction

Here for “pc+4”, the value of ‘pc’ is the address of current processing instruction .



NOTES:

Don't forget to instance instruction memory, complete the port binding.

TIPS: The design here is for reference ONLY, NOT request.



Practice2

1. Make a Minisys source file with j, jal, jr, beq,bne and other NON-jumping instructions included.
2. Using the **Minisys1AssemblerV2.2** to assembler the source file on step 1, get the coe files .
3. Using the “prgmip32.coe” generated on step 2 as the initial file for the ROM in IFetch submodule to verify the its funciton:
 - 3-1) What’s the value of register PC while the reset is valid.
 - 3-2) While reset is invalid, on which edge of clock would the value of register PC be updated?
 - 3-3) What’s the updated value to register PC while the current instruction is j, jal, jr, beq,bne and other NON-jumping instructions.
 - 3-4) On which edge of clock would the instruction be fetched out?
 - 3-5) Is there any difference between the two output ports(“**branch_base_addr**” and “**link_addr**”)

Tips:1) There are j, jal, jr, beq,bne and other NON-jumping instructions in cputest.asm(which is in the Minisys1AssemblerV2.2.rar), you can modify it as an alternative to the 1st step.

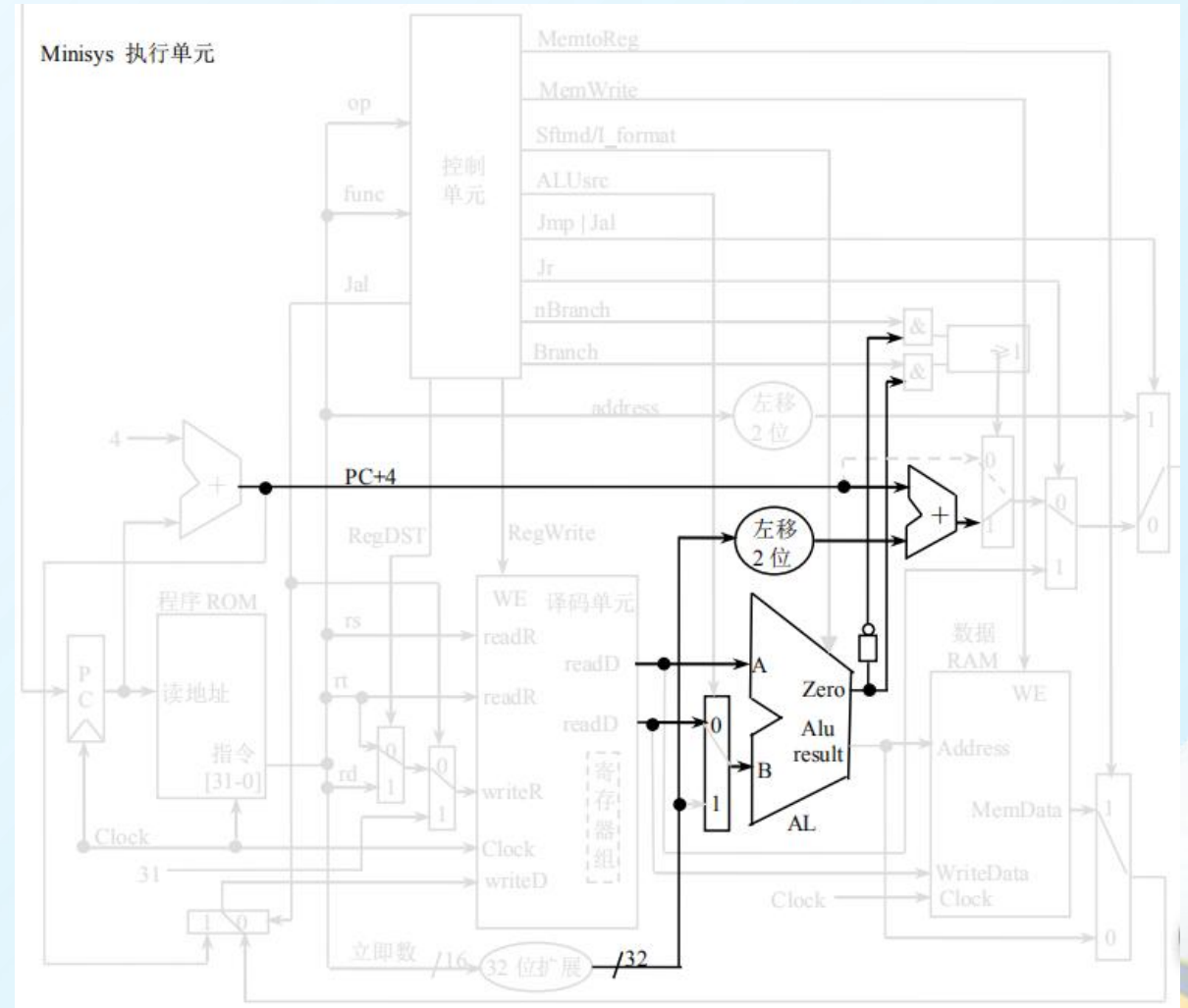
2) “**Minisys1AssemblerV2.2.rar**” could be found in the directory “**labs**” of course blackboard site



ALU

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

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



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

Minisys - A subset of MIPS32



MIPS_Green_Sheet.pdf

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

NOTE:

Minisys is a subset of MIPS32.

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

BASIC INSTRUCTION FORMATS

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



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

Q1: Who needs the calculation result of ALU?

```
output[31:0]  reg ALU_Result;    // the ALU calculation result
output        Zero;              // 1 means the ALU_result 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 affect operation in ALU directly

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

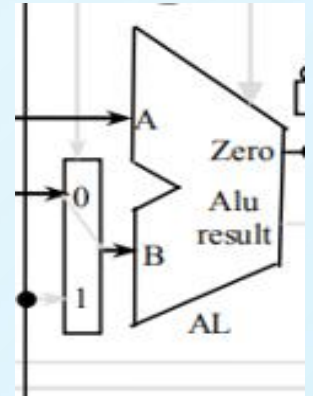
wire[32:0]    Branch_Addr;        // the calculated address of the instruction, Addr_Result is Branch_Addr[31:0]
```



The Selection On Operand2

Two operands of ALU: **Ainput** and **Binput**.

- **Ainput** is from the output port “Read_data_1” of **Decoder**
- **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**” which comes from **Controller**.



```
input[31:0] Read_data_1; // from Decoder
input[31:0] Read_data_2; // from Decoder
input[31:0] Sign_extend; // from Decoder
// from Controller, 1 means the Binput is an extended immediate, otherwise the Binput is Read_data_2
input      ALUSrc;
```

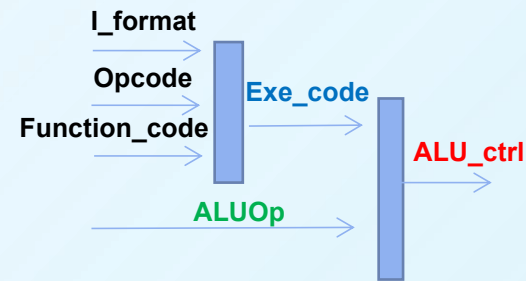
```
assign Ainput = Read_data_1;
assign Binput = (ALUSrc == 0) ? Read_data_2 : Sign_extend[31:0];
```



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).



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

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

### 3) `function_opcode` is `instruction[5:0]`

**Q.** Could the 'Exe\_code' be generated by Controller or by ALU? What's your choice?

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





# 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        |

➤ Implements(3)

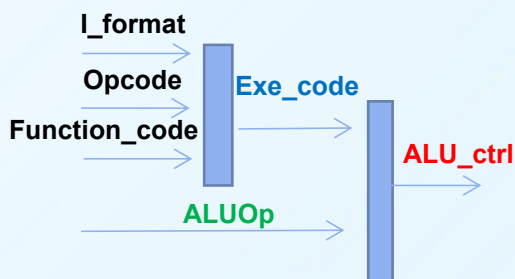
**ALU\_ctrl** : based on **ALUOp** and **Exe\_code**, specify most of the operation details in ALU

**ALUOp** =

{ (R\_format || I\_format) , (Branch || nBranch) }

**Exe\_code** = (I\_format==0) ?

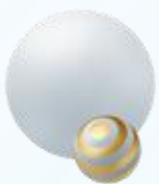
Function\_opcode :  
{ 3'b000 , Opcode[2:0] };



```
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

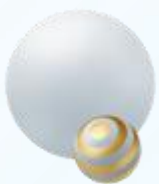
➤ **Type1:** The **same operation** in ALU with **different operand source**

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        |



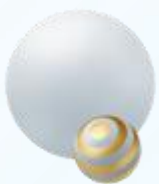
# ALU\_ctrl usage continued

- **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 calculation the address based on the base address and offset which is same as in “add” operation.*

| Exe_code[3..0] | ALUOp[1..0] | ALU_ctrl[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” (destination ):
  - “beq” and “bne” : Addr\_reslut
  - “sub” : “ALU\_reslut”
- “subu” vs “slt” , “sltu” (destination )
  - “slt” and “sltu” :Zero.

**I\_format** is used here to distinguish these two types

- “sub” vs “slt”, ”subu” vs “sltu”:

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** instructions: could be identified by the input port “**sftmd**”
- **lui** : whose ALU\_ctrl code is the same as “nor”, but could be identified by “**l\_format**”
- **jr** : could be identified by the input port “jr”, not execute in ALU
- **j** : could be identified by the input port “jmp”, not execute in ALU
- **jal** : could be identified by the input port “jal”, not execute in ALU



# Practice3-1: Arithmetic 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       |

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

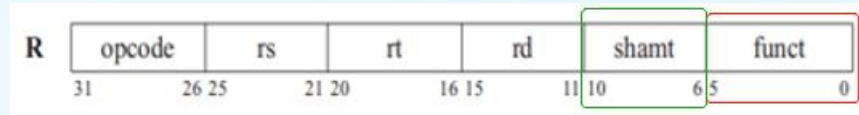


# Shift Operation

| 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
```



## Practice3-2: Shift Operation

Complete the following code, taking the table on the left hand 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 00100
 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
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 3-3: 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 (arithmetic 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 3-4: the output “Addr\_result ” and “Zero”

The values of “**Addr\_result**” and “**Zero**” are still not determined.

|              |                         |                                                |
|--------------|-------------------------|------------------------------------------------|
| output[31:0] | reg <b>ALU Result</b> ; | // the ALU calculation result                  |
| output       | <b>Zero</b> ;           | // 1 means the ALU result is zero, 0 otherwise |
| output[31:0] | <b>Addr Result</b> ;    | // the calculated instruction address          |

- “**Zero**” is a signal used by “**IFetch**” to determine whether to use the value of “**Addr\_result**” 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\_result** should be the sum of **pc+4**(could be get from **PC\_plus\_4**) and the **immediate** in the instruction.



## Practice 3-5: 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<br>Depends on your design<br><b>Addr_Result</b> : should be the sum of <b>pc+4</b> (could be get from <b>PC_plus_4</b> ) and the <b>immediate</b> in the instruction |             |                                     |