Computer Organization

Lab11/12 CPU(3) IFetch, Clock , I/O, Controller+

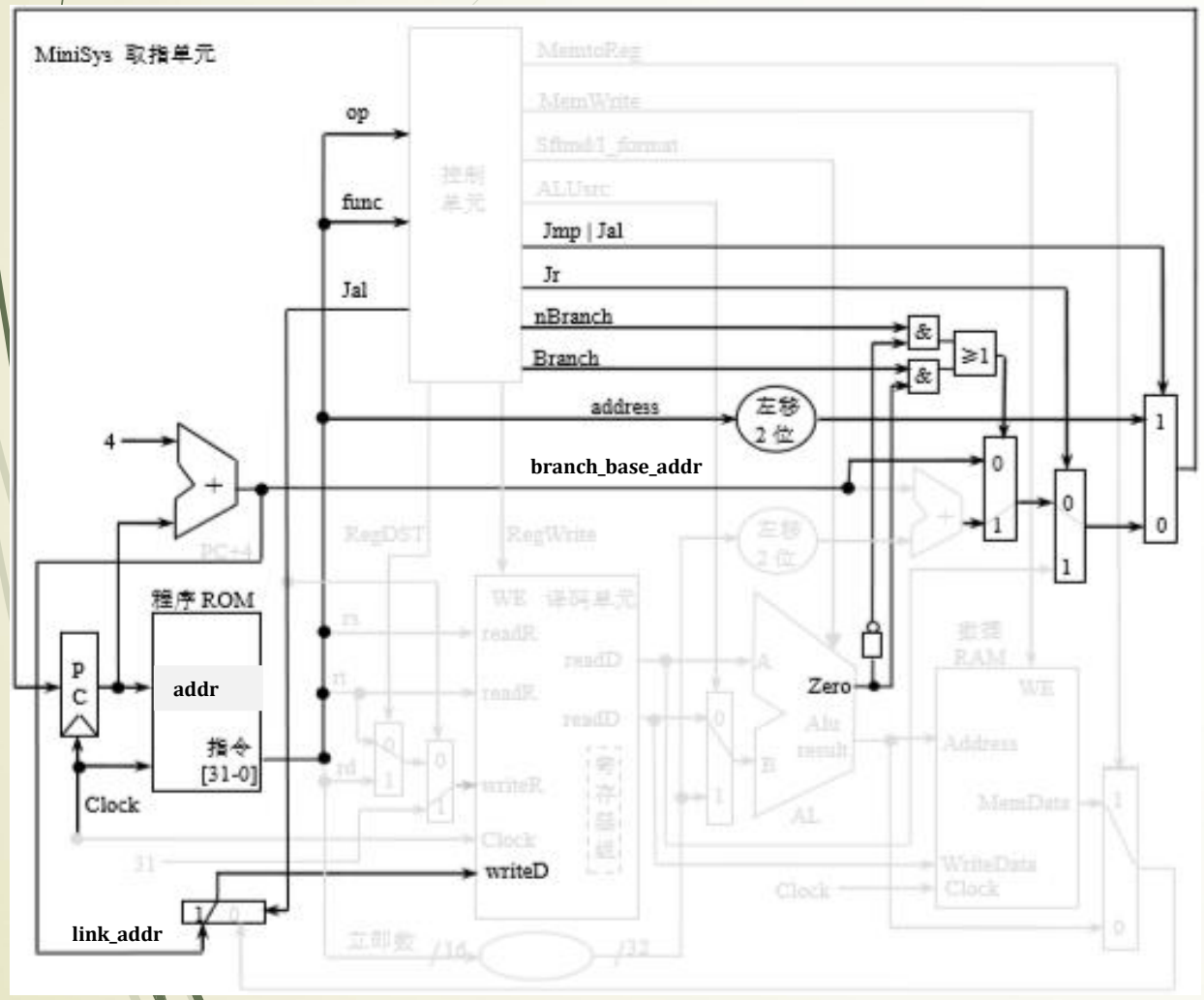
2021 Spring term

wangw6@sustech.edu.cn

Topics

- CPU (3)
  - IFetch
  - Clock
  - Build a Single Cycle CPU
  - I/O
  - Controller+

Instruction Fetch

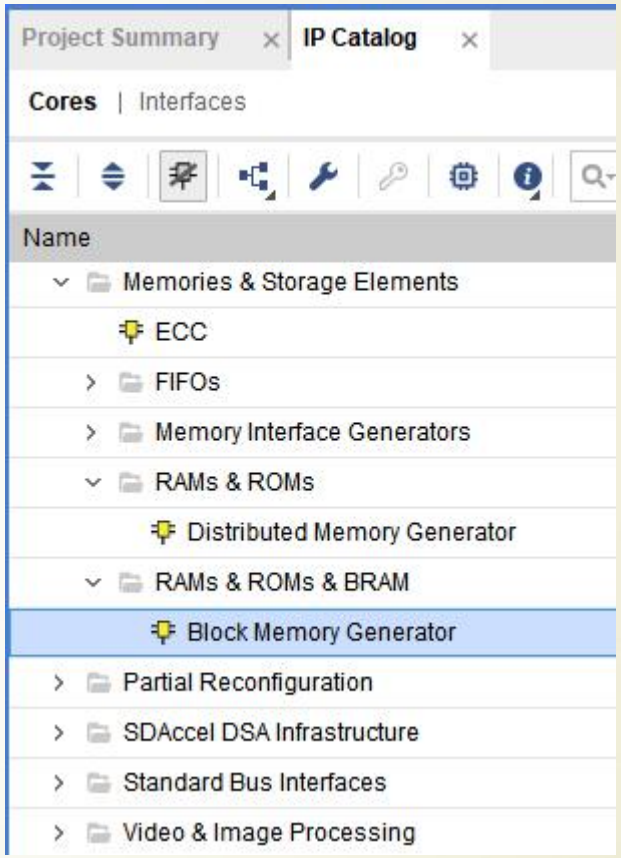


Instruction Fetch

- 1. **Using ROM as instruction memory**
- 2. **Update the value of the PC register**
  - Reset the value of the PC register
  - Add 4 to the value of the PC register
  - Update the value of the PC register according to the jump instructions
- 3. **Fetch the instructions according to the value of the PC register**

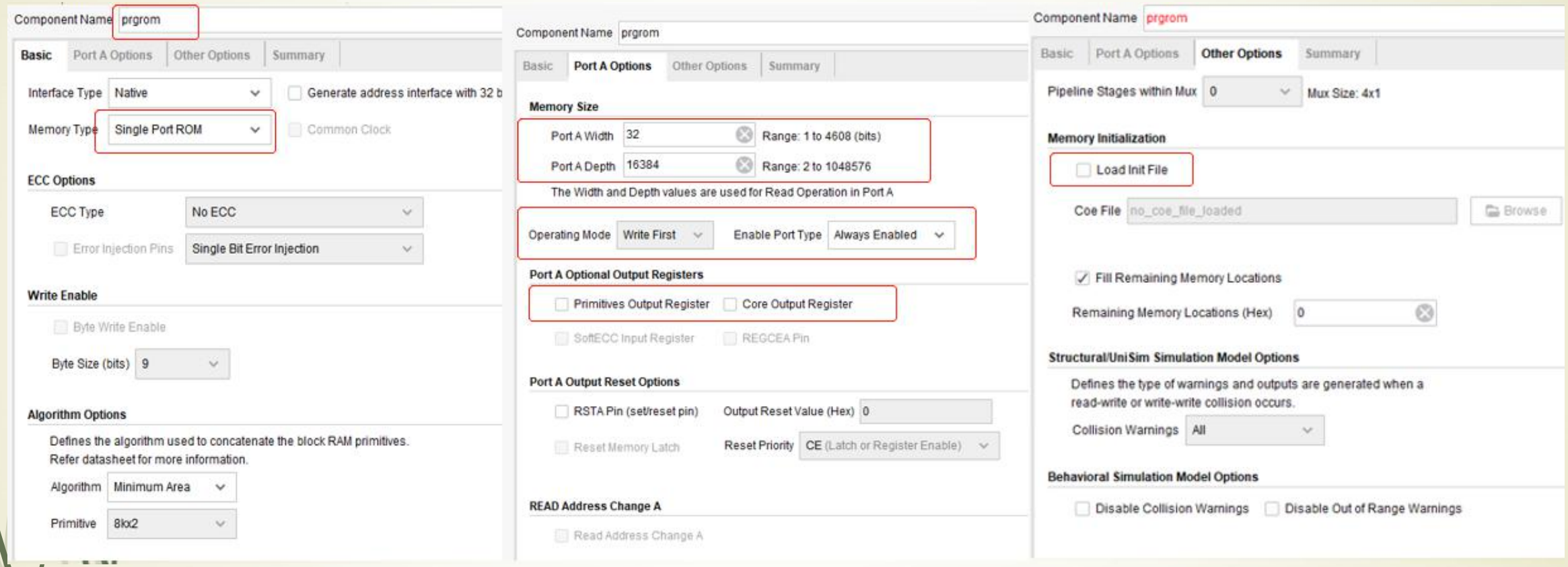
Using IP core As Instruction Memory

- **Step1**： **Find** the IP core(**Block Memory Generator**) in **IP Catalog**
- **Step2**： **Customize** the IP core
  - set name(component name), type
  - 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 same with which of the RAM IP core in lab9

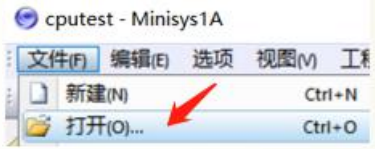
Customize the IP core



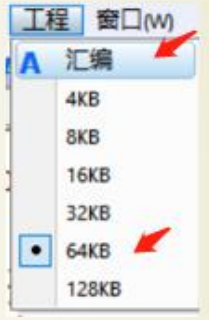
NOTE: set the init file of prgrom after this IP core has been added into vivado project, use the coe file on sakai site as the init file.

Using coe file to Initiate Instruction Memory

- **Option 1. Generate coe file by using MinisysAssembler**
  - Step1. **Open** the assembly **source file**



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



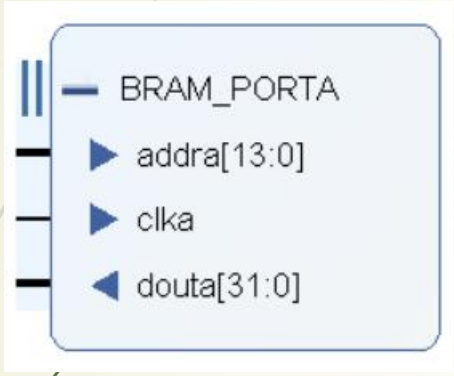
  - Step3. The coe files could be found at the sub-directory: "**output**"
    - The initial data of data memory could be found in file "dmem32.coe"
    - The machine code of Minisys instruction could be found in the file "**prgmip32.coe**"

- **Option 2. Get the coe file from sakai site (lab/lab11)**

https://sakai.sustech.edu.cn/portal/site/d50211ea-1586-4344-9d92-a6c42eb7f4e0/tool/b47e4c13-4220-4f61-b881-a211abee2a96?panel=Main

- **Copy the coe file to** the following directory, and re-generate the customized ip core again.
  - "VIVADO_PROJECT_DIR\VIVADO_PROJECT_NAME.srcs\sources_1\ip\prgrom"

Instance the IP core




```
prgrom instmem(
    .clka(clock),           // input wire clka
    .addra(PC[15:2]),       // input wire [13 : 0] addra
    .douta(Instruction)     // output wire [31 : 0] douta
);
```

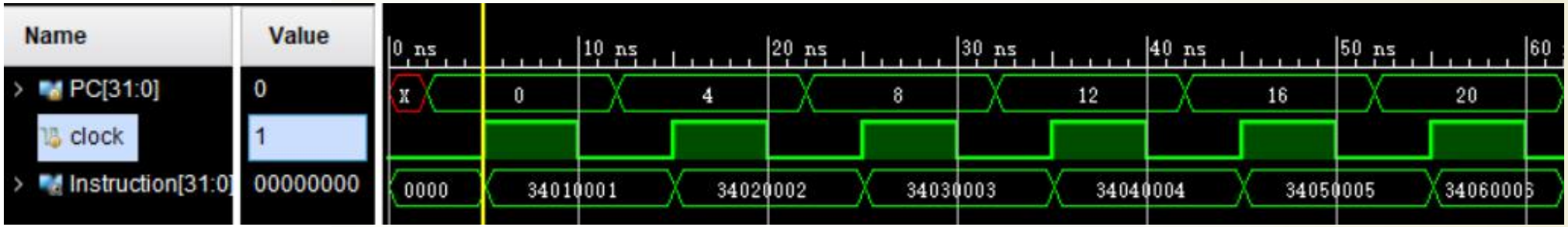
In One Cycle CPU, the process of geting instrcution should happen on the **posedge** of the clock. At this moment, IFetch module gets the instruction which is store at "**addra**" from the instruction memory "Instmem"

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

```
memory_initialization_radix = 16;
memory_initialization_vector =
34010001,
34020002,
34030003,
34040004,
34050005,
34060006,
34070007,
34080008,
34090009,
340a000a,
340b000b,
340c000c,
```




```
module prgrom_tb( );      //a reference for the testbench
   reg[31:0] PC;
   reg clock;
   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)
      #10 PC = PC+4;
      #10 $finish;
   end
endmodule
```

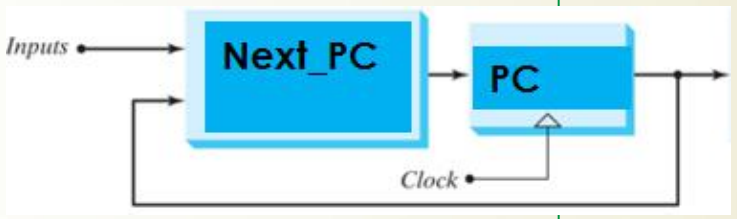
- The initial value of the PC register is always set as 0.
- Read the “Instruction” from “douta” port of Instruction memory “prgrom” on every posedge of the “clock”.
- In this tescase, the value of 'PC' is added with 4 each time.

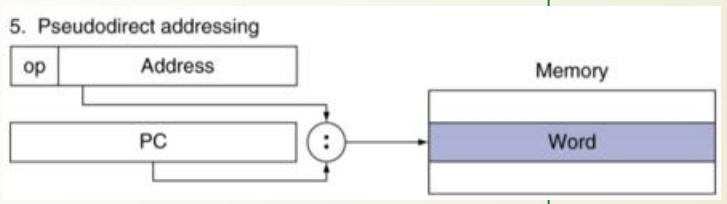
IFetch Module

```
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
   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

   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
```



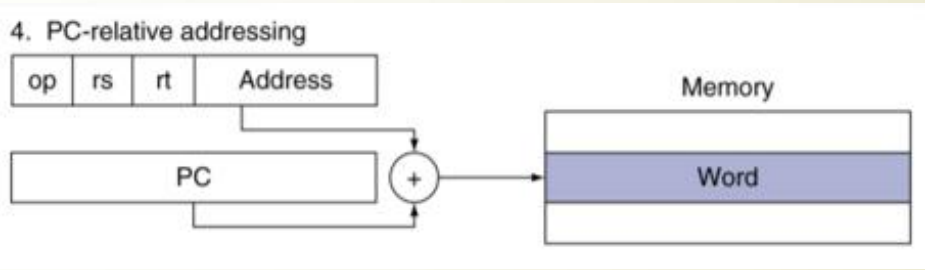


Q1: How to update 'Next_PC'?
Q2: When to update the value of the PC register?
Q3: Is this Minisys ISA a Harvard structure or Von Neumann structure

Prepare for Decoder and ALU

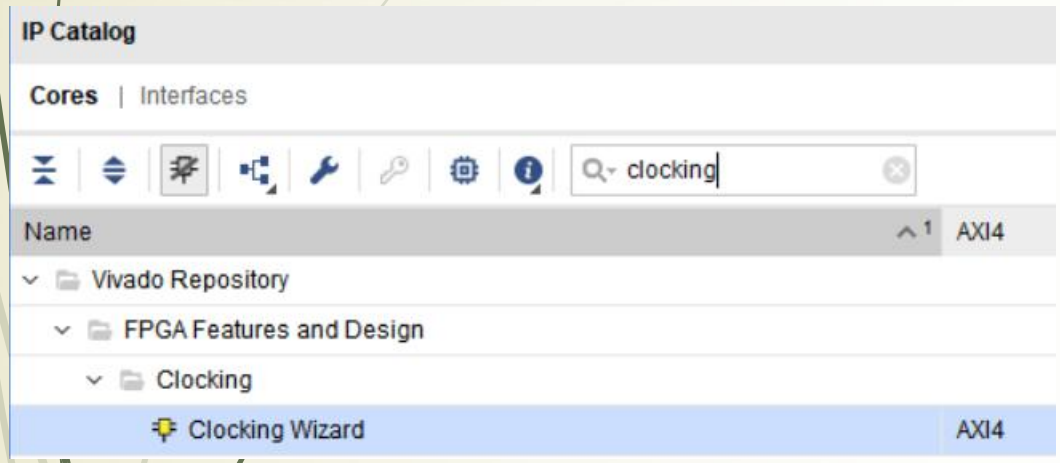
output[31:0] **branch_base_addr**; // (pc+4) to ALU which is used by branch type instruction
output[31:0] **link_addr**; // (pc+4) to decoder which is used by jal instruction

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



Don't forget to instance instruction memory, finish the port connection.

Clock



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

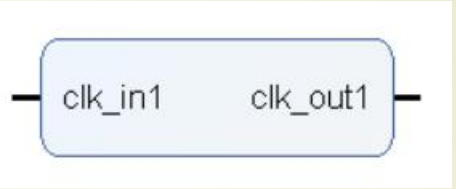
    1. The clock on the Minisys development board is **100Mhz**
    2. **A clock of 23Mhz is needed for the single clock cpu**

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

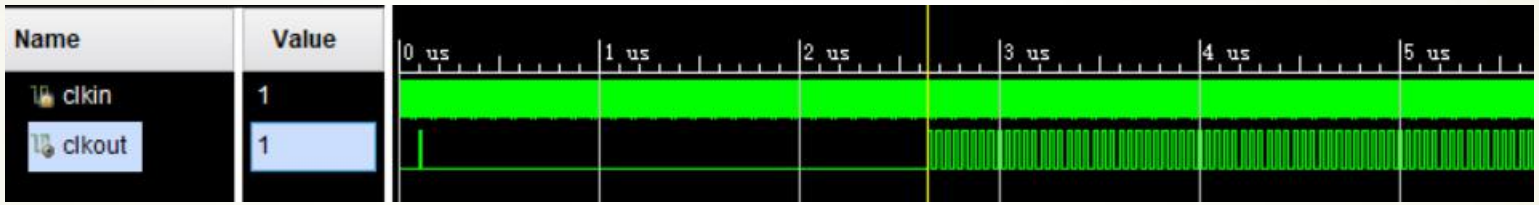
- [ ] reset
- [ ] power_down
- [ ] locked

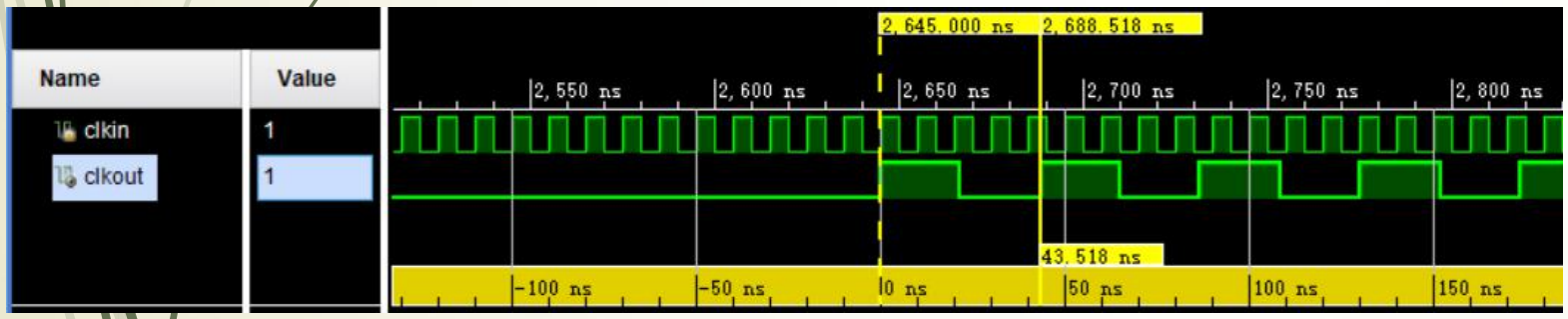
Reset Type

- (•) 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 reach stability.





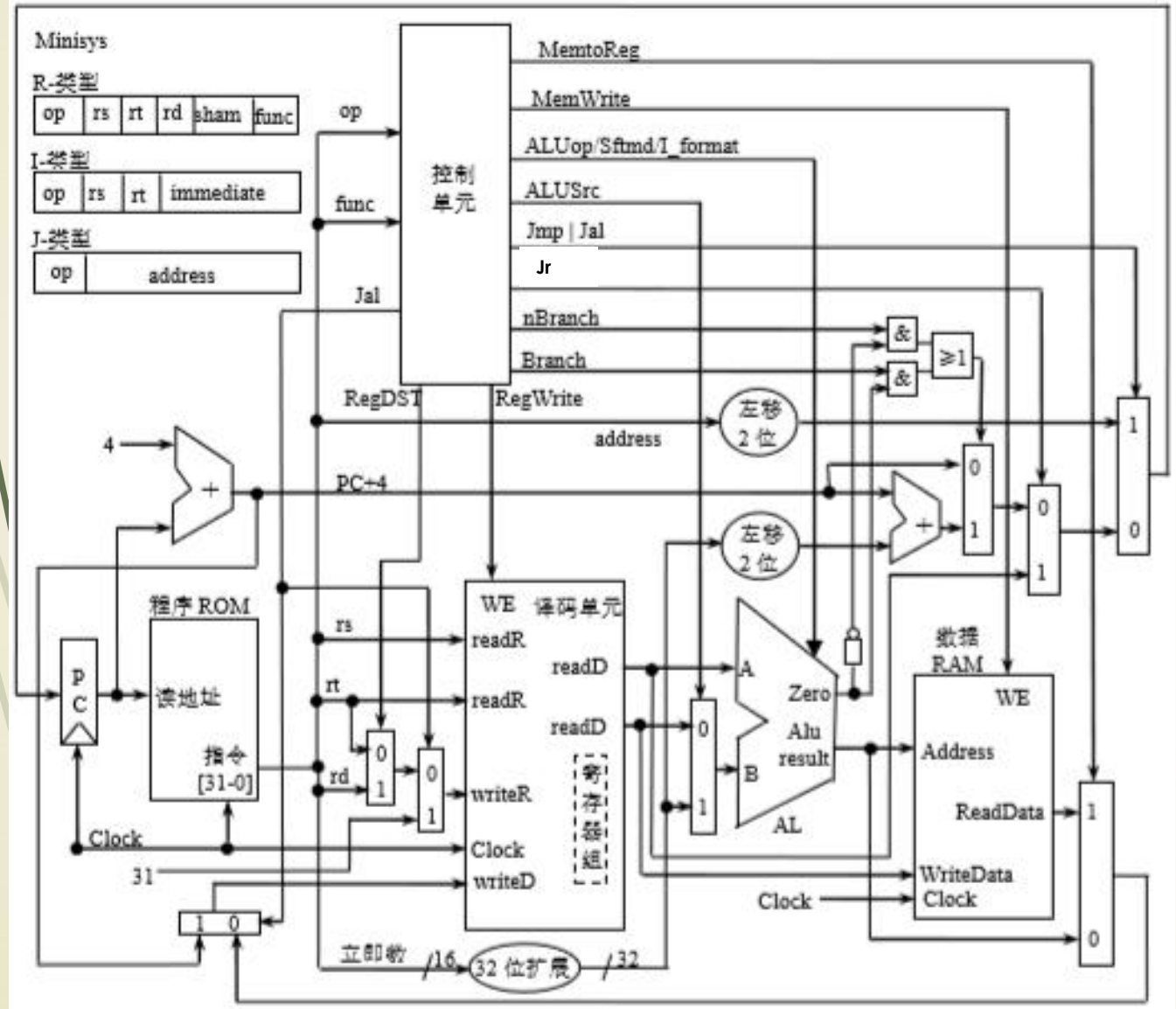

```
// a reference for test the ip core 'cpuclk'
module cpuclk_tb(    );
   reg clkin;
   wire clkout;
   cpuclk
clk1(.clk_in1(clkin),.clk_out1(clkout));
   initial begin
      clkin = 1'b0;
   end
   always #5 clkin=~clkin;
endmodule
```

Single Cycle CPU

- Determine input and output **ports** of the CPU
  - **Clock** signal, **reset** signal
  - **Input** port
  - **Output** port
- Determine the **sub-module** inside the CPU
  - Clock module
  - ifetch, controller, decoder, execution/alu,memory , IO processing
- **Build** CPU top-level module
  - Notes the relationship on the sub-modules inside the CPU, especially the control signal, data, 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

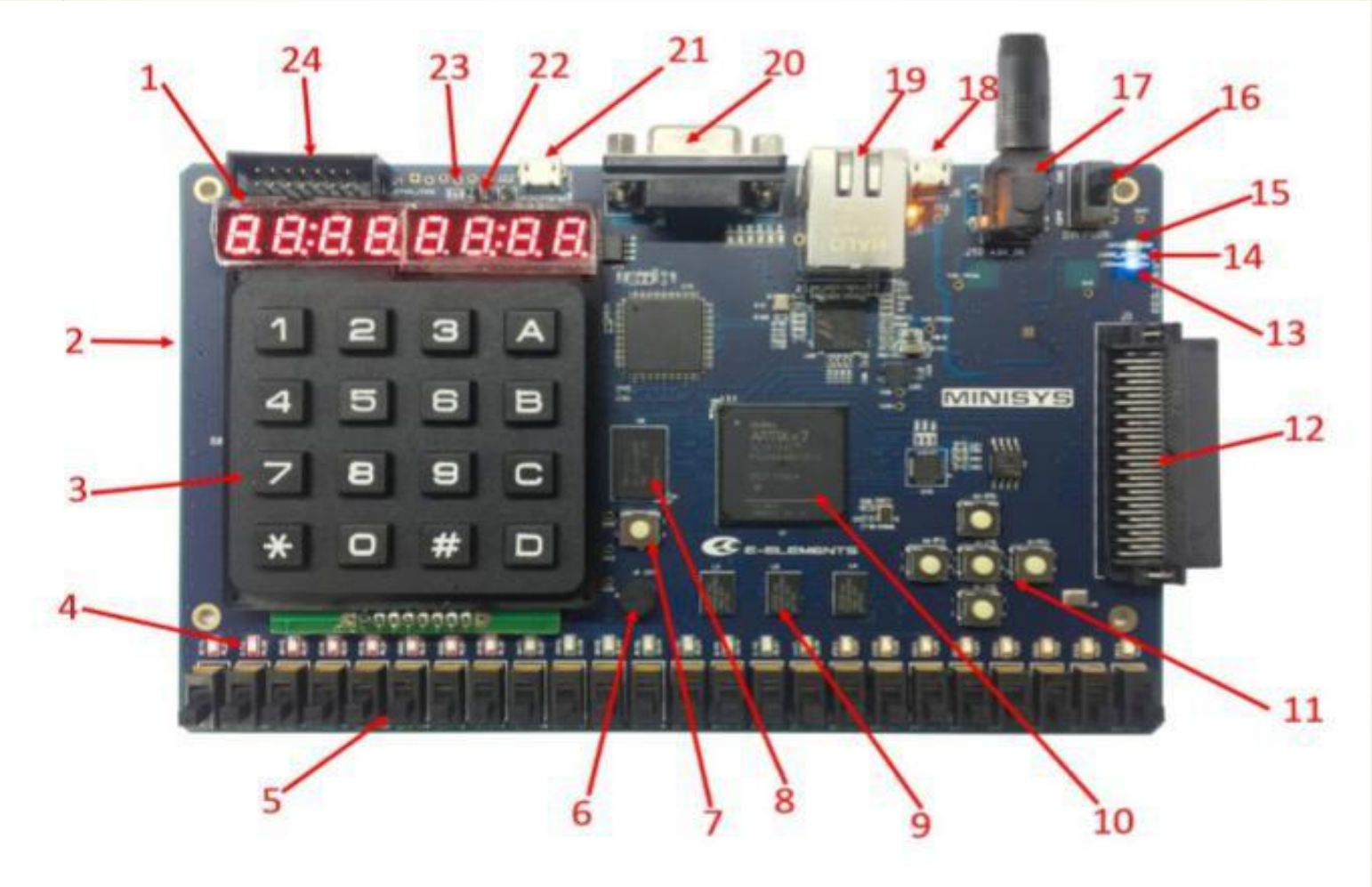


Create a CPU **top** module

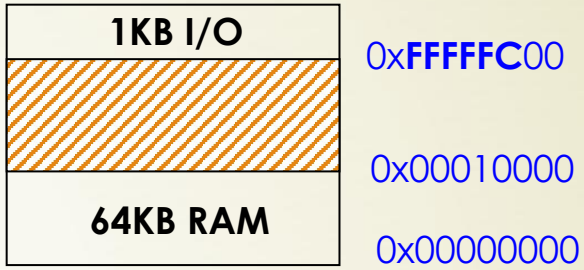
1) Instantiating the **clock**, **decoder**, execution unit/**ALU**, **ifetch** and **controller** units.

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

I/O Interface



I/O Share Part of the Data Bus Address



The space of 32 bits address bus is 4GB(0x0000_0000~0xFFFF_FFFF), a high of **1024** bytes(0xFFFF_FC00~0xFFFF_FFFF) is designed to be allocated for the **I/O**. **C**hip **S**elect and **address** are specified by specifying **10** IO port lines.

Here is an example for **24** LED lights and **24** DIP switches, both of them are divided into two groups, all the ports in one group share the same address.

1. The CS(Chip Select) signal of the LED light is **ledCtrl**
2. The CS(Chip Select) signal of the DIP switch is **switchCtrl**

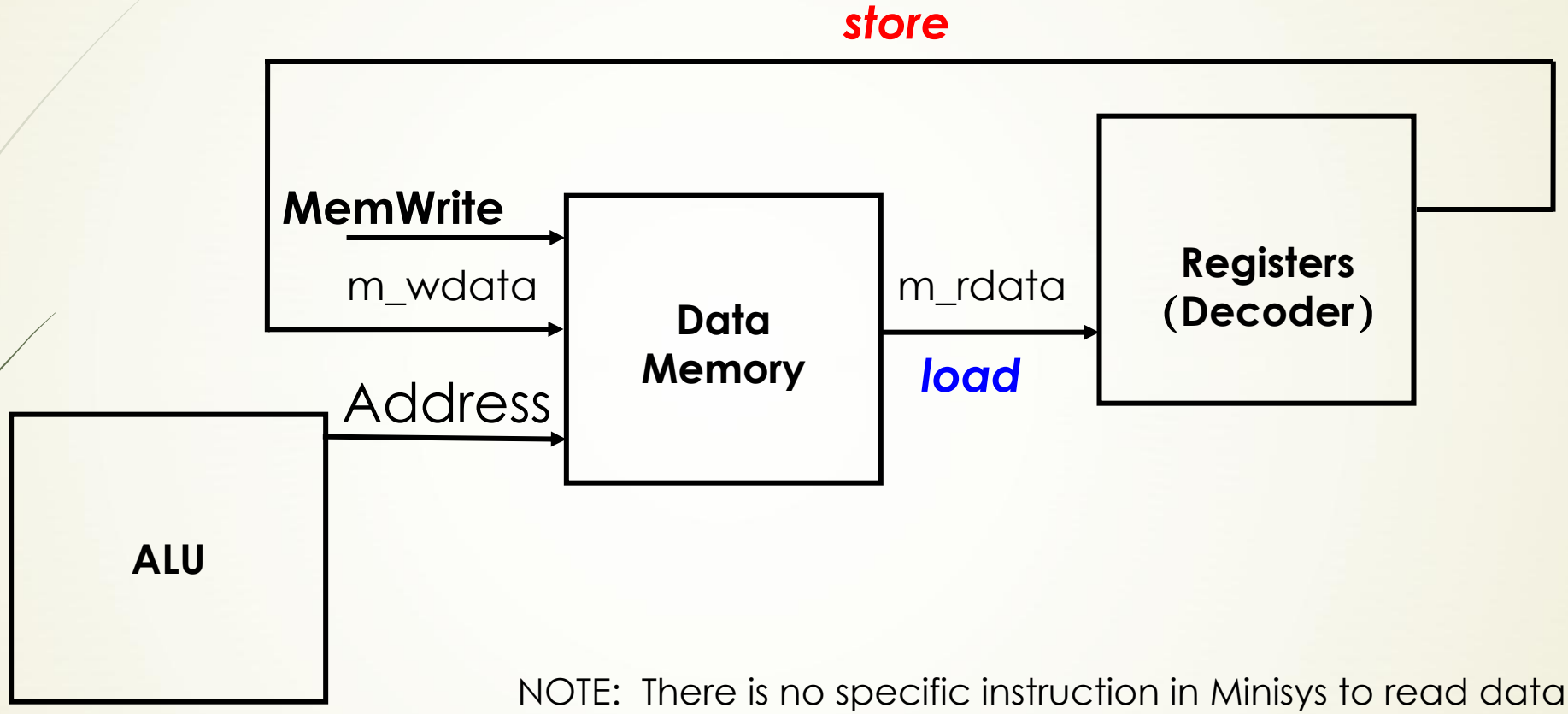
Range	LED(1~16)	LED(17~24)	Switch(1~16)	Switch(17~24)
Address	0xFFFFFC60	0xFFFFFC62	0xFFFFFC70	0xFFFFFC72

Note:

*1. In the computer field, there are usually two schemes for I/O address space design: I/O and memory **unified addressing** or **I/O independent addressing**. However there is no dedicated I/ O instruction in current Minisys-1. Here, both LW and SW instructions are used for RAM access and I/O access, which means Minisys-1 can only use I/O unified addressing.*

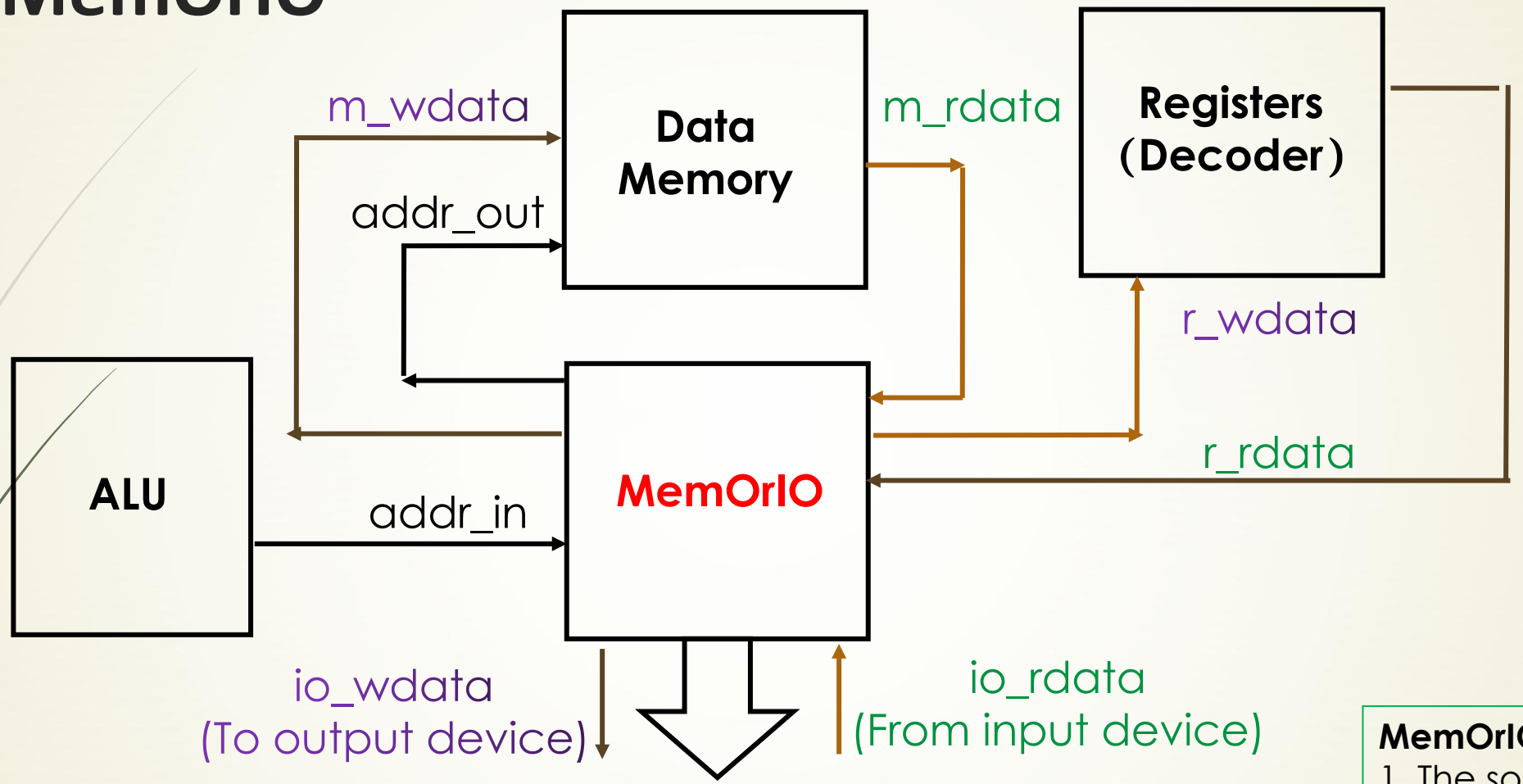
2. It is just a way for IO address implementation, but not the only choice.

Corresponding Operation of LW/SW



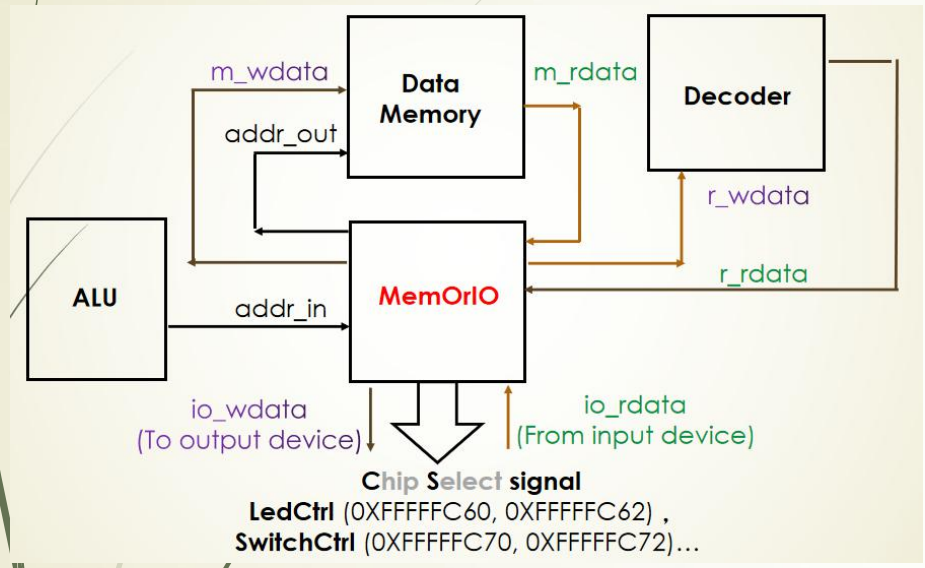
NOTE: There is no specific instruction in Minisys to read data from input ports and write data to output ports.
To implement the read/write process on I/O, it needs to share the load/store instructions in Minisys.

MemOrIO



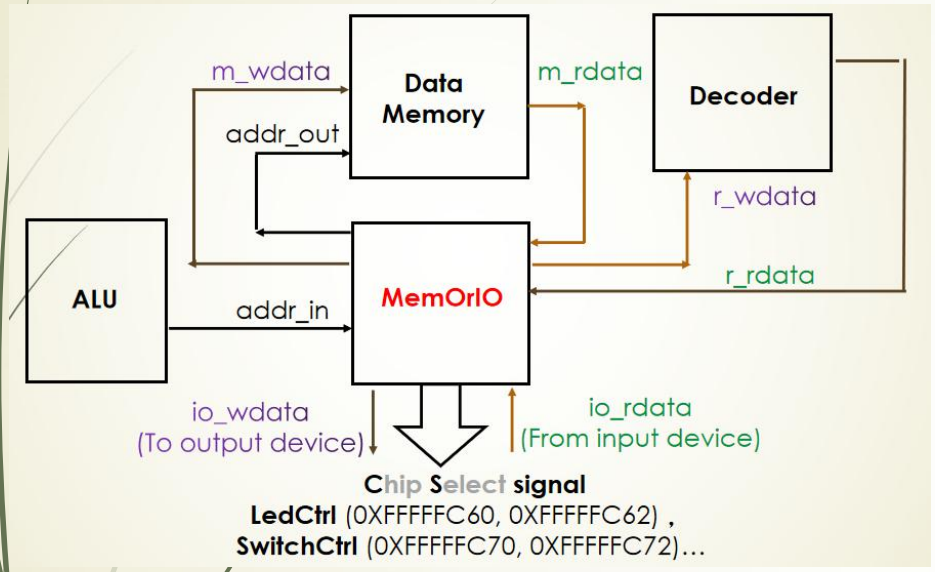
MemOrIO determine:
1. The source of r_rdata
2. The destination of r_wdata

MemOrIO continued




```
module MemOrIO( mRead, mWrite, ioRead, ioWrite,addr_in, addr_out,
m_rdata, io_rdata, r_wdata, r_rdata, write_data, LEDCtrl, SwitchCtrl);

input mRead;                    // read memory, from control32
input mWrite;                   // write memory, from control32
input ioRead;                   // read IO, from control32
input ioWrite;                  // write IO, from control32

input[31:0] addr_in;            // from alu_result in executs32
output[31:0] addr_out;          // address to memory

input[31:0] m_rdata;            // data read from memory
input[15:0] io_rdata;           // data read from io,16 bits
output[31:0] r_wdata;           // data to idecode32(register file)

input[31:0] r_rdata;            // data read from idecode32(register file)
output reg[31:0] write_data;    // data to memory or I/O  (m_wdata, io_wdata)
output LEDCtrl;                 // LED Chip Select
output SwitchCtrl;              // Switch Chip Select
```

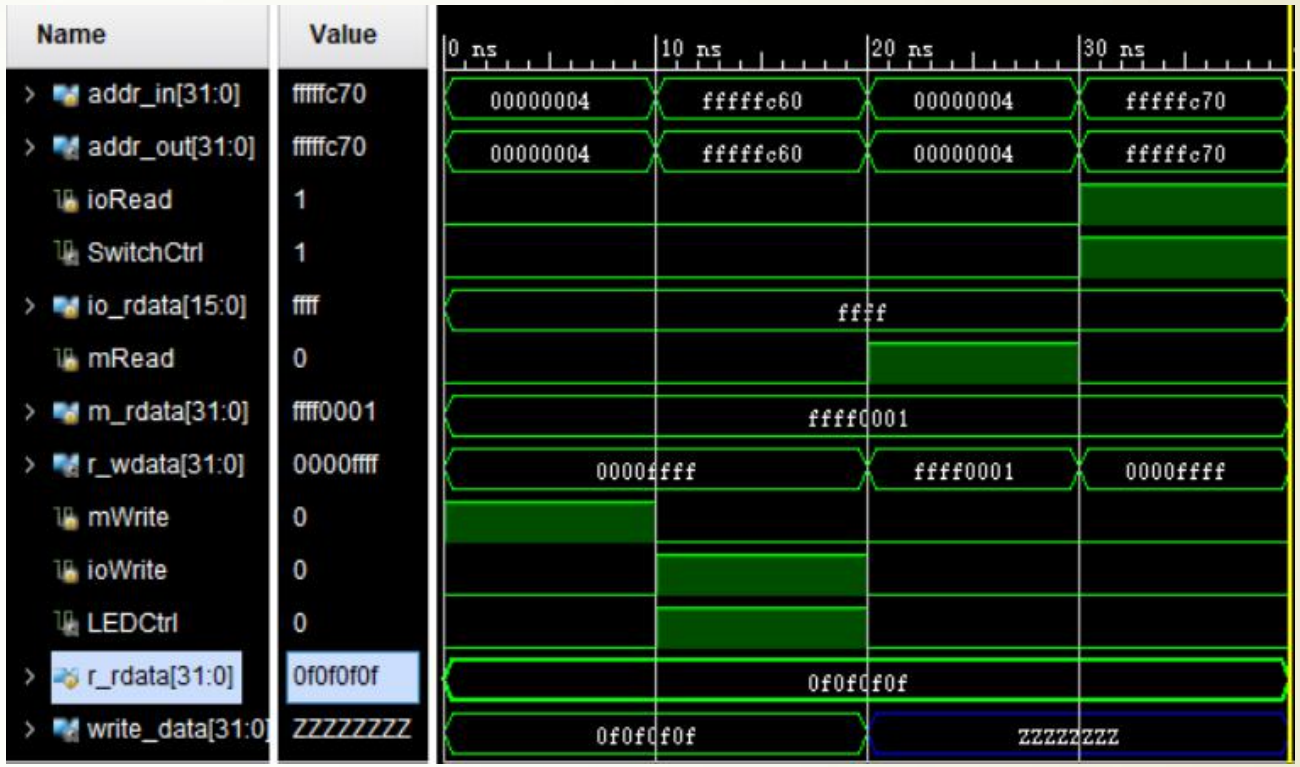
MemOrIO continued




```
assign addr_out= addr_in;
// The data wirte to register file may be from memory or io.
// While the data is from io, it should be the lower 16bit of r_wdata.
assign r_wdata = ? ? ?

// Chip select signal of  Led and Switch  are all active high;
assign LEDCtrl= ? ? ?
assign SwitchCtrl= ? ? ?

always @* begin
        if((mWrite==1)||(ioWrite==1))
            //wirte_data could go to either memory or IO. where is it from?
                write_data = ? ? ?
        else
                write_data = 32'hZZZZZZZZ;
end
endmodule
```

The Function Verification of MemOrIO

```
// a reference for the testbench of MemOrIO
module MemOrIO_tb( );
   reg mRead,mWrite,ioRead,ioWrite;
   reg[31:0] addr_in,m_rdata,r_rdata;
   reg[15:0] io_rdata;
   wire LEDCtrl,SwitchCtrl;
   wire [31:0] addr_out,r_wdata,write_data;

   MemoryOrIO umio(addr_out,addr_in,
mRead,mWrite,ioRead,ioWrite,
m_rdata,io_rdata,r_rdata,r_wdata,write_data,
LEDCtrl,SwitchCtrl);
```

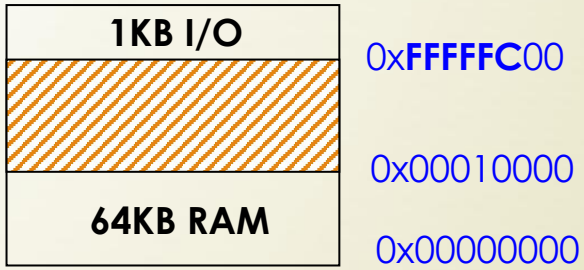



```
 initial begin    // r_rdata -> m_wdata(write_data)
     m_rdata = 32'h0xffff_0001;  io_rdata = 32'h0xffff;  r_rdata = 32'h0x0f0f_0f0f;   addr_in = 32'h4;{mRead,mWrite,ioRead,ioWrite}= 4'b01_00;
 #10   addr_in = 32'hffff_fc60;   {mRead,mWrite,ioRead,ioWrite}= 4'b00_01;          // r_rdata -> io_wdata(write_data)
 #10   addr_in = 32'h0000_0004;   {mRead,mWrite,ioRead,ioWrite}= 4'b10_00;          // m_rdata -> r_wdata
 #10   addr_in = 32'hffff_fc70;   {mRead,mWrite,ioRead,ioWrite}= 4'b00_10;          // io_rdata -> r_wdata(write_data)
 #10 $finish;
   end
endmodule
```

Controller +

Add new ports to Controller for IO reading and writing support.

```
module  control32(Opcode,Function_opcode,Jr,Branch,nBranch,Jmp,Jal,
Alu_resultHigh,RegDST,MemorIOtoReg,RegWrite,MemRead,MemWrite,IORead,IOWrite,
ALUSrc,ALUOp,Sftmd,I_format);
    ...
    input[21:0] Alu_resultHigh;     // From the execution unit Alu_Result[31..10]
    output MemorIOtoReg;    // 1 indicates that data needs to be read from memory or I/O to the register
    output RegWrite;        // 1 indicates that the instruction needs to write to the register
    output MemRead;         // 1 indicates that the instruction needs to read from the memory
    output MemWrite;        // 1 indicates that the instruction needs to write to the memory
    output IORead;          // 1 indicates  I/O read
    output IOWrite;         // 1 indicates  I/O write
    ...
```



Controller + continued

Modify the logic of the 'MemWrite', **add** 'MemRead', 'IORead' and 'IOWrite' signals, **change** 'MemtoReg' to 'MemorIOtoReg'.

```
 // The real address of LW and SW is Alu_Result, the signal comes from the execution unit
// From the execution unit Alu_Result[31..10], used to help determine whether to process Mem or IO
   input[21:0]  Alu_resultHigh;

    output      MemorIOtoReg;   //1 indicates that read date from memory or I/O to write to the register
    output      MemRead;        // 1 indicates that reading from the memory to get data
    output      IORead;                 // 1 indicates  I/O read
    output      IOWrite;                // 1 indicates  I/O write

    assign RegWrite = (R_format || Lw || Jal || I_format) && !(Jr) ;        // Write memory or write IO
    assign MemWrite = ((sw==1) && (Alu_resultHigh[21:0] != 22'h3FFFFF)) ? 1'b1:1'b0;
    assign MemRead = ? ? ?                  // Read memory
    assign IORead  = ? ? ?                  // Read input port
    assign IOWrite     = ? ? ?              // Write output port

// Read operations require reading data from memory or I/O to write to the register
    assign MemorIOtoReg = IORead || MemRead;
```

Practice

- 1. Generate IP core 'prgrom', build a testbench to do its function verification.

  TIPS: the reference could be found on Page 8
- 2. Generate IP core 'cpuclk', build a testbench to do its function verification.

  TIPS: the reference could be found on Page 14
- 3. Complete IFetch module, do its function verification.
- 4. Complete MemoryOrIO module, do its functon verification.
- 5. Complete Controller+ module, do its functon verification.