CPU = IFETCH + CONTROLLER + DECODER + ALU + DATA-MEMORY + IO

取指令　控制器　指令译码　算术逻辑　内存　输入输出

组合逻辑：移位、加、减

时序逻辑：乘、除

赋值：①连续赋值：assign <wire> = <value>;

②过程赋值：initial、always 块内

非阻塞：<reg> <= <value>;

阻塞：<reg> = <value>;

程序结构：顺序、选择、循环

（循环过多时电路功耗大，有时甚至在综合时报错）

DUT：一个有输入输出的设计模块。　一般来说，testbench更复杂

Testbench：测试DUT，没有IO。

- **DUT is a designed module with input and output ports**
  - While do the design, non-synthesizable grammar means can’ t be convert to circuit, is NOT suggested!
  - DUT may be a top module using structured design which means the sub module is instanced and connected in the top module
- **Testbench is used for test DUT with NO input and output ports**
  - Instance the DUT, bind its ports with variable, set the states of variable which bind with inputs and check the states of variable which bind with outputs
  - Testbench is NOT a part of Design. It only runs in FPGA/ASIC EDA, so the un-synthesizable grammar can be used in testbench

# Module Design

- **Gate Level**
  - Implementation from the **perspective of gate-level structure** of the circuit, **using gates as components, connecting pins of gates**
  - Using **logical and bitwise operators or original primitive**(not , or , and , xor , xnor ..)
    - For example: not n1(na,a); xor xor1(c,a,b);
- **Data Streams**
  - Implementation from the **perspective of data processing and flow**
  - Using **continuous assignment**, pay attention to the correlation between signals, the difference between logical and bitwise operators
    - For example: assign z = ( x | y) ^ (a&b);
- **Behavior Level**
  - Implementation from the **perspective of the Behavior** of Circuits
  - Implemented in the **always** statement block
  - The variable which is assigned in the always block Must be **Reg type.**

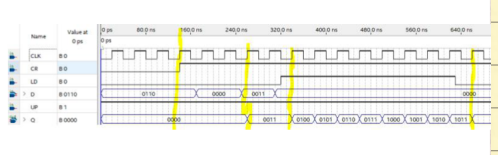
# Behavior Modeling

## 7 Behavior Modeling(if-else)

“if else” block can represent the priority among signals

From the overall structure, from top to bottom, priority decreases in turn

```
module updown_counter(D,CLK,CR,LD,UP,Q);
input [3:0]D;
input CLK,CR,LD,UP;
output reg [3:0] Q;

always @(posedge CLK )

if(!CR)
   Q=0;
   else if(!LD)
   Q=D;
   else if(UP)
   Q=Q+1;
   else
   Q=Q-1;
endmodule
```



**NOTIC:**

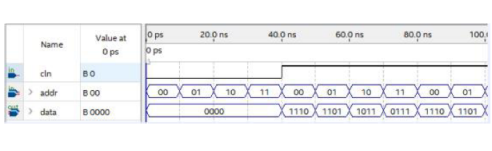
1) If there is no ‘else’ branch in the statement, latches will be generated while doing the synthesis.
2) Nested ‘if-else’ is NOT suggested, ‘case’ is suggested as an alternative.

不建议嵌套多层if-else，会造成延时大

## 8 Behavior Modeling(case)

```
module decorder(cln,data,addr);
input cln;
input [1:0] addr;
output reg [3:0] data;

always @(cln or addr )

begin

if(0==cln)
   data=4'b0000;
else
   case(addr)
   2'b00:data=4'b1110;
   2'b01:data=4'b1101;
   2'b10:data=4'b1011;
   2'b11:data=4'b0111;
   endcase

end

endmodule
```

| case | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| x | 0 | 0 | 1 | 0 |
| z | 0 | 0 | 0 | 1 |



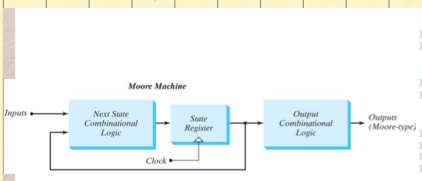
**NOTIC:**

**Without defining 'default' branches and declearing all situations under the “case”, latches will be generated while doing the synthesis.**

若缺少对分支的描述（如else和default），则EDA工具转换时会自动添加锁存器。

# Sequential Circuit : FSM

一般能用组合就不用时序.




```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
module moore_2b(input clk,rst_n,x_in, output[1:0] state,next_state);
reg [1:0] state,next_state;
parameter S0=2'b00,S1=2'b01,S2=2'b10,S3=2'b11;
always @(posedge clk,negedge rst_n) begin
    if(!rst_n)
        state <= S0;
    else
        state <= next_state;
end
always @(state,x_in) begin
    case(state)   <-当前状态
    S0: if(x_in) next_state = S1; else next_state = S0;
    S1: if(x_in) next_state = S2; else next_state = S1;
    S2: if(x_in) next_state = S3; else next_state = S2;
    S3: if(x_in) next_state = S0; else next_state = S3;
    endcase
end
endmodule
```

输入 ↑ 下一状态

下一个状态是由当前状态+输入决定.

米利/摩尔模式都不建议一段式

# Constant

- Expression
  - <bit width>' <numerical system expression><number in the numerical system >
    - Numerical system expression
      - B / b : Binary
      - O / o : Octal
      - D / d : decimal
      - H / h : hexadecimal
  - ' <numerical system expression><number in the numerical system >
    - The default value of bit width is based on the machine-system(at least 32 bit)
  - <number> : default in decimal
    - The default value of bit width is based on the machine-system(at least 32 bit)
- **X( uncertain state) and Z(High resistivity state)**
  - The default value of a wire variable is Z before its assignment
  - The default value of a reg variable is X before its assignment
- **Negative value**
  - Minus sign must be ahead of bit-width
    - -4' d3 (is ok) while 4' d-3 is illegal
- **Underline**
  - Can be used between number but can NOT be in the bit width and numerical system expression
    - 8' b0011_1010 (is ok) while 8' _b_0011_1010(is illegal)
- Parameter (symbolic constants)
  - Used for improving the readability and maintainability
  - Declare an identifier on a constant
  - Parameter p1=expression1,p2=expression2,..;

# Variable

- **Wire**
  - Net, Can' t store info ,must be driven (such as continuous assignment)
  - The input and output port of module is wire by default
  - Can NOT be the type of left-hand side of assignment in initial or always block
- **Reg**
  - MUST be the type of **left-hand** side of assignment in initial or always block
  - The default initial value of reg is an indefinite value X. Reg data can be assigned positive values and negative values.
  - When a reg data is an operand in an expression, its value is treated as an unsigned value, that is, a positive value.
  - For example, when a 4-bit register is used as an operand in an expression, if the register is assigned-1. When performing operations in an expression. It is considered to be a complement representation of + 15 (- 1)

```
wire a;
wire [7:0] b;
wire [4:1] c,d;
```

```
module sub_wr();   <- 输入应为wire
  input reg in1,in2;
  output out1;
  output out2;
endmodule
```

Error: Port in1 is not defined
Error: Non-net port in1 cannot be of mode input
Error: Port in2 is not defined
Error: Non-net port in2 cannot be of mode input

```
23  module test();
24      wire i1,i2;
25      wire o2;
26      reg o1;
27      sub_wr s1(.in1(i1),.in2(i2),
28      .out2(o2),
29      .out1(o1));   -> 输出端口应用wire
30  endmodule
31
32  module sub_wr(in1, in2, out1, out2);
33      input in1, in2;
34      output out1, out2;
35  endmodule
```

```
module sub_wr(in1,in2,out1,out2);
  input in1,in2;
  output out1;
  output reg out2;

  assign in1 = 1'b1;

  initial begin
    in2 = 1'b1;
  end
endmodule
```

initial内赋值应用reg

sim_1 (2 errors)
[VRFC 10-529] concurrent assignment to a non-net o1 is not permitted [test.v:29]

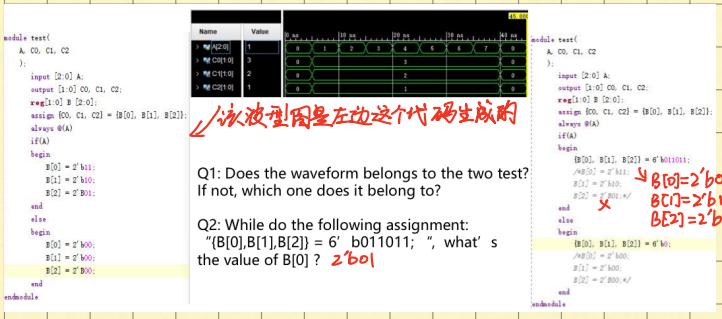
Error: procedural assignment to a non-register in2 is not permitted, left-hand side should be reg/integer/time/genvar

除非必须使用reg，否则一律用wire

凡有在always、initial被赋值的对象必须为reg.

# Memory

memory 即寄存器组. reg [1:0] A[2:0] 即表明A是每个有[1:0]宽的3个寄存器组.




波形图是左边这个代码生成的

Q1: Does the waveform belongs to the two test? If not, which one does it belong to?

Q2: While do the following assignment: "{B[0],B[1],B[2]} = 6' b011011; ", what' s the value of B[0] ? 2'b01

B[0]=2'b01
B[1]=2'b10
B[2]=2'b11

# Operator

位运算: &、^、~、|

逻辑运算: &&、||、!

highest priority
! ~
* / %
+ -
<< >>
< <= > >=
== != === !==
&
^ ^~
|
&&
||
? :
lowest priority

- Bit splicing operator { }
- Multiple data or bits of data are separated by commas in order, then using braces to splice them as a whole.

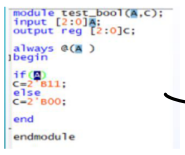
Such as:

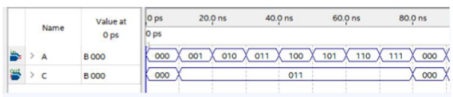
- {a, B [1:0], w, 2'b10} // Equivalent to {a, B [1], B [0], w, 1'b1, 1' b0}

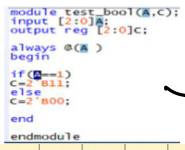
Repetition can be used to simplify expressions

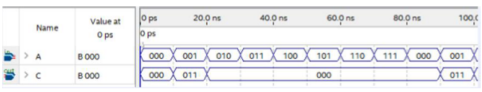
- {4 {w} } // Equivalent to {w, w, w, w}
- { b, {2 {x, y} } } // Equivalent to {b, x, y, x, y}

When numeric values are used for conditional judgment, non-zero values represent logical truth and zero values represent logical false.



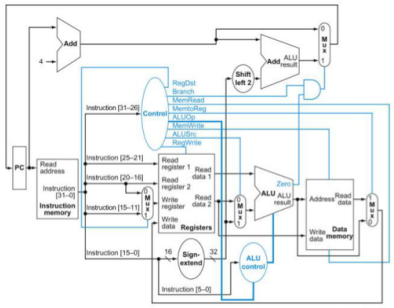



单周期CPU：基本指令在一个时钟周期内完成
多周期CPU：基本指令在多个时钟周期内完成

## Control Path & Data Path




Source: H&P textbook

**Control Path**: Interprete instructions and generate signals to control the data path to execute instructions

**Data Path**: The parts in CPU with componets which are involved to execute instructions

取址 → 分析 → 执行
IFetch Controller+Decoder ALU

## Minsys : 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 |

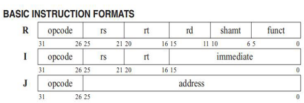
MIPS_Green_Sheet.pdf

NOTE:

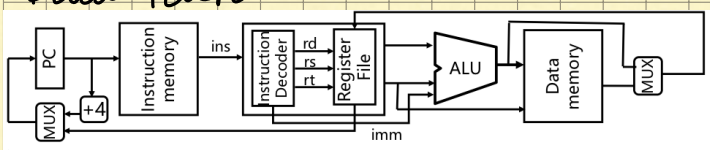
Minisys is a subset of MIPS32.

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

**BASIC INSTRUCTION FORMATS**




## Data Path



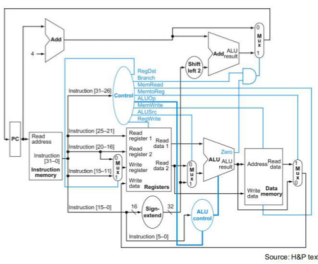

1. Fetch Instruction
2. Fetch Data from immediate/ register
3. Execute
4. Access Memory for Data
5. Write back to the Register

| | Instruction fetch | Data Fetch | Instruction Execute | Memory Access | Register WriteBack |
|---|---|---|---|---|---|
| add[R] | Y | Y | Y | | Y |
| addi[I] | Y | Y | Y | | Y |
| store[I] | Y | Y | Y | Y | |
| load[I] | Y | Y | Y | Y | Y |
| branch[I] | Y | Y | Y | | |
| jump[J] | Y | Y | Y | | |
| jal [J] | Y | Y | Y | | |

CPU(单周期) 1 cycle = 1 posedge + 1 negedge
posedge (I FETCH) 该周期的上升沿获得指令
negedge (Data memory) 该周期的下降沿获得数据
next-cycle : posedge (Decoder writeback) 在下个周期上升沿更新
下个周期的行为不受影响.
I FETCH(下一个) 与 write back(这一个) 同时不冲突.
需时间很少.

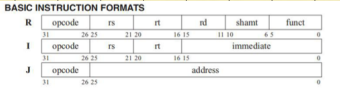
## Control Path

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



Source: H&P textbook

**BASIC INSTRUCTION FORMATS**



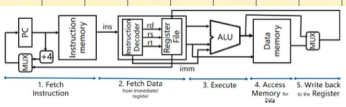

**Instruction Analysis:**

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

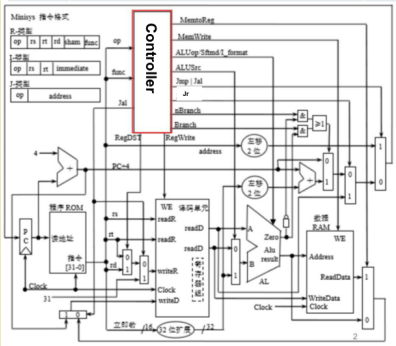
# Controller

## Controller

Why a controller is needed?




| Module | How To Process | Instructions and Comments |
|---|---|---|
| IFetch | Determine how to update the value of PC register | 1. (pc+4)+immediate(sign extended) (bne [I])<br>2. the value of $31 register (jr [R])<br>3. {(pc+4)[31:28],lableX[25:2],2'b00} ( jal, j [J])<br>4. pc+4 (other instruction except branch, jr, j and jal [R\I]) |
| Decoder | Determine whether to write register or not | lw [I], R type instruction except jr [R], jal[J] |
| | Get the source of data to be written<br>Get the address of register to be written | 1. Data memory (lw[I]) -> rt<br>2. ALU ([R]) -> rd<br>3. address of instruction (jal[J]) ->31 |
| | Determine whether to get immediate data from the instruction and expand it to 32bit | add([R]) vs addi([I]) |
| Memory | Determine whether to write memory or not | (sw[I]) vs lw[I] |
| | Get the source of data to be written | rs of registers (sw[I]) |
| | Get the address of memory unit to be written | the output of ALU (sw[I]) |
| 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) |




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

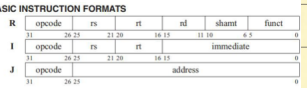
**Q2:** What's the usage of function code in the instruction?

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

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

Tips: parts of the answer could be found on page 3 of this slides.

BASIC INSTRUCTION FORMATS




## Controller continued

**NOTES**: The design of Controller in this slides is **ONLY a reference, NOT a requirement**.

```
input[5:0]  Op;        // instruction[31..26], opcode
input[5:0]  Func;      // instructions[5..0], funct
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"(R),otherwise it's "rt"(I)
output      MemtoReg;  // 1 indicate read data from memory and write it into register
output      RegWrite;  // 1 indicate write register(R,I(lw)), 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      Sftmd;     // 1 indicate the instruction is shift instruction
```

Q1: Which type of design style on port would you prefer: 1bit or Coded multi bit wide port ?

```
output      I_format;
/* 1 indicate the instruction is I-type but isn't
 "beq","bne","LW" or "SW"  */
output[1:0]  ALUOp;
/*  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;  */
```

Q2: What's the destinaion sub-module of this output port ?

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

**Jr** =((Opcode==6'b000000)&&(Function_opcode==6'b001000)) ? 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)

| 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 |

"**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 |
| | srl | 00_0010 |
| | sllv | 00_0100 |
| | 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)

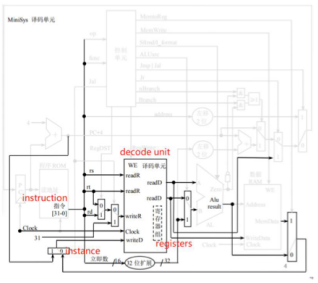
||(Function_opcode==6'b000110)||(Function_opcode==6'b000111))
&& R_format)? 1'b1:1'b0;

组合逻辑

Branch和nBranch不能合并的原因：
要区分有分支/无分支

用assign. 因为定义时output Jr未指定类型，默认为wire.

# Decoder




- **Get data** from the instruction directly or indirectly
  - opcode, function code : how to get data, where to get data
  - **immediate data** in the instruction([15:0]), (e.g. immi = Instruction[15:0]) need to be **signextended to 32bits**
  - **data in the register**, the address of the register is coded in the instruction. e.g. rs = Instruction[25:21];
  - **data in the memory**, the address of the memory unit need to be calculated by ALU with base address stored in the register and offset as immediate data in the instruction
- **Read/Write data from/to Register File**

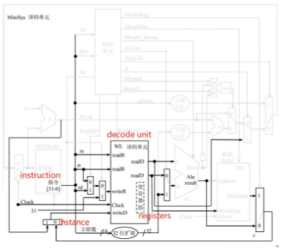
**BASIC INSTRUCTION FORMATS**

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

15 **Decoder continued**

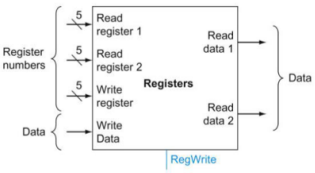
**BASIC INSTRUCTION FORMATS**

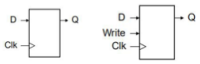
| | 31–26 | 25–21 | 20–16 | 15–11 | 10–6 | 5–0 |
|---|---|---|---|---|---|---|
| R | opcode | rs | rt | rd | shamt | funct |
| I | opcode | rs | rt | immediate | | |
| J | opcode | address | | | | |




- **Register File-Outputs**
  - read data1
  - read data2
  - extended Immi
- **Register File-Inputs**
  - **read address**
    - [R] add: rs,rt
    - [R] jr : [31]
    - [I] addi: rs
  - **write address**
    - [R] add: rd
    - [J] jal : [31]
    - [I] addi: rt
  - **write data**
    - [R] add: data from alu_result
    - [I] lw: data from memory
  - **control signal**
    - [R]/[I]/[J] writeRegister
    - [J]jal : jal
    - [I] lw : memToReg
    - [R]/[I]: rd vs rt

**Register File:**
Almost all the instructions need to read or write register file in CPU
32 common registers with same bitwidth: 32




Q1:
How to avoid the confliction between register read and register write?

Q2: Which kind of circuit is this register file, combinatorial circuit or sequential circuit?

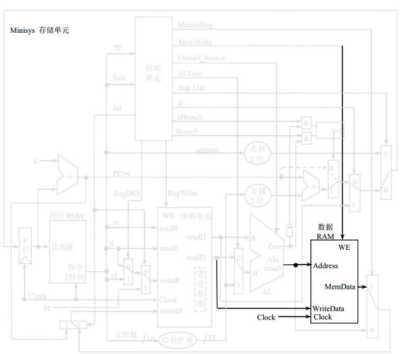
Q3: How to determine the size of address bus on register file?

```
//verilog tips:
reg[31:0] register[0:31];
assign Read data 1 = register[Read register 1];
register[Write register] <= WriteData;
```

decoder模块是时序逻辑
① 从指令中获取数据
② 从通用寄存器中取数据
③ 做立即数扩展
④ 做寄存器写入
执行lw指令时，regWrite信号有效，decoder做先读后写操作

是否做写操作
读/写的判断取决于 { regWrite, clk edge }
何时做操作

# 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**;

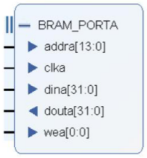
*// the address of memory unit which is tobe read/writen*
input[31:0] **address**;

*// data tobe wirten 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**;

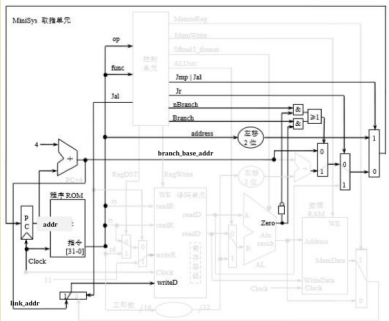
# With Memory IP Instanced

```
// 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?*

# Instruction Fetch



Instruction Fetch

- 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

# Instance the IP core


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

NOTES:
"**prgrom**" is the IP core which is generated in vivado follow the steps on page 13, 14 of this slides.

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

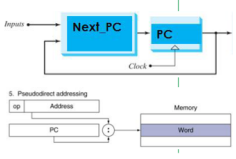
# IFetch Module

```
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

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

```
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
```

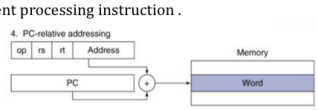



Q1: Complete the code to update 'Next_PC'
Q2: Could be 'PC' ready while read the "prgrom"? Determine when to update the value of the PC register.
Q3: Is this Minisys ISA a Harvard structure or Von Neumann structure take a look at the initial value of PC

## 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 .




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

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

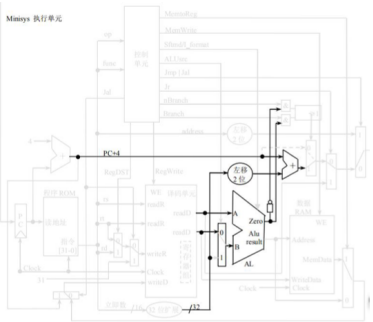
ALU

## ALU

- 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

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

组合逻辑




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?

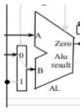
```
output[31:0]    reg ALU_Result;   //  the ALU calculation result
output          Zero;             // 1 means the ALU_reslut is zero, 0 otherwise
output[31:0]    Addr_Result;      //  the calculated instruction address
```

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).**

- 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 |

| 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] };**

ALU_ctrl相同，则指令行为相同.


```
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 |

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

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

| 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 |

- *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.*

- **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

# 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 |

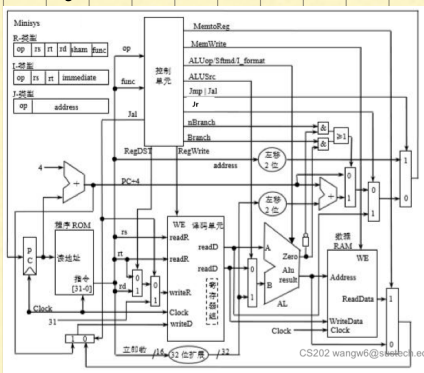
| sftm[2:0] | process |
|---|---|
| 3'b000 | sll rd, rt, shamt |
| 3'b010 | srl rd, rt, shamt |
| 3'b100 | sllv rd, rt, rs |
| 3'b110 | srlv rd, rt, rs |
| 3'b011 | sra rd, rt, shamt |
| 3'b111 | srav rd, rt, rs |
| other | not shift |

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

Ainput, Binput/shamt are the operand of shift operation


```
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
```

# Single Cycle CPU




**Build** a **CPU top** module

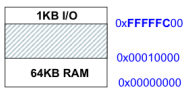
1) **Instantiating** the sub-modules: **clock**, **Decoder**, execution unit/**ALU**, **IFetch, Controller** and **Data-Memory**.

2) Complete the inter-module connection inside the CPU and the **binding** to the CPU **port.**

**Q. How to test the CPU ? how to determine the program, the data, how to check the result?**

# IO Share Part of the Data Bus Address

## I/O Share Part of the Data Bus Address




The space of **32** bits address bus is **4GB**(0x0000_0000~0xFFFF_FFFF)

**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 on Minisys board, both of them are divided into two groups, all the ports in one group share the same address.

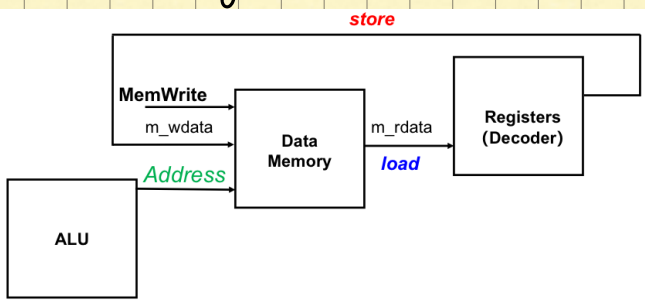
1. The CS(Chip Select) signal of the LED light is **ledCtrl**
2. The CS(Chip Select) signal of the DIP switch is **switchCtrl**

| 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.*

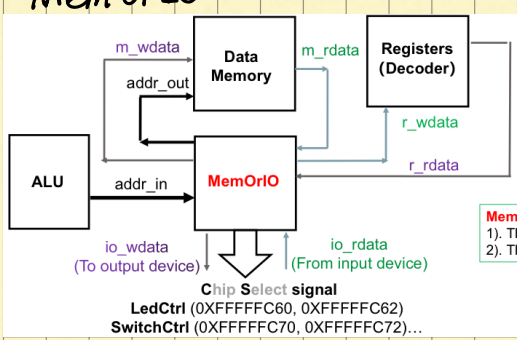
2. It is just a way for IO address implementation, but not the only choice.

# Corresponding Operation of LW/SW




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

# Mem Or IO



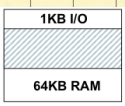

**MemOrIO** determine:
1). The source of r_rdata
2). The destination of r_wdata

```
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 Controller
input mWrite;               // write memory, from Controller
input ioRead;               // read IO, from Controller
input ioWrite;              // write IO, from Controller

input[31:0] addr_in;        // from alu_result in ALU
output[31:0] addr_out;      // address to Data-Memory

input[31:0] m_rdata;        // data read from Data-Memory
input[15:0] io_rdata;       // data read from IO,16 bits
output[31:0] r_wdata;       // data to Decoder(register file)

input[31:0] r_rdata;        // data read from Decoder(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
```

```
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
```

# 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
      ...
```

**1) Modify** the logic of the '***MemWrite***'
**2) Add** '***MemRead***', '***IORead***' and '***IOWrite***' signals
**3) 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;
```