



# Computer Organization

---

## Lab9 CPU Design(1)

ISA  
Controller, Decoder



# Topic

## ➤ CPU Design(1)

### ➤ ISA

## ➤ Control Path

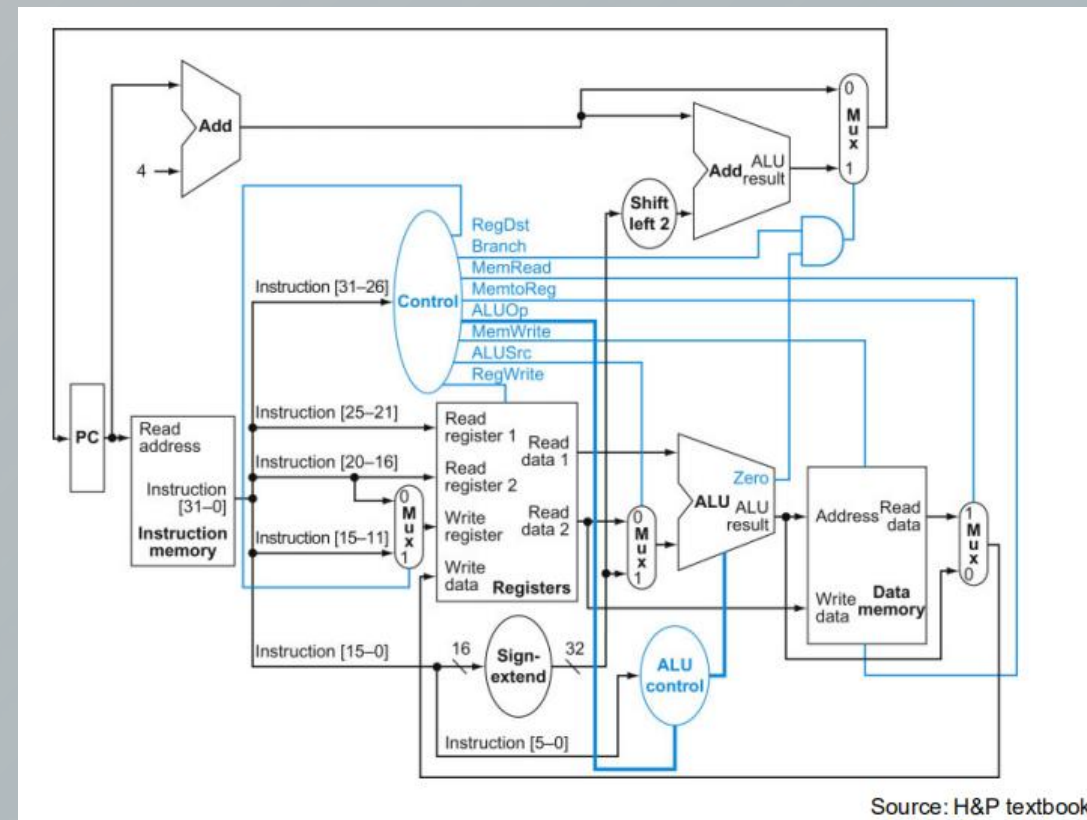
### ➤ Controller

### ➤ *Practice1*

## ➤ Data Path(1)

### ➤ Decoder

### ➤ *Practice2*





# Minisys - A subset of MIPS32

Type	Name	funC(ins[5:0])
R	sll	00_0000
	srl	00_0010
	sllv	00_0100
	srlv	00_0110
	sra	00_0011
	srav	00_0111
	jr	00_1000
	add	10_0000
	addu	10_0001
	sub	10_0010
	subu	10_0011
	and	10_0100
	or	10_0101
	xor	10_0110
	nor	10_0111
	slt	10_1010
	sltu	10_1011

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

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



MIPS\_Green\_Sheet.pdf

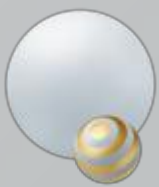
NOTE:

Minisys is a subset of MIPS32.

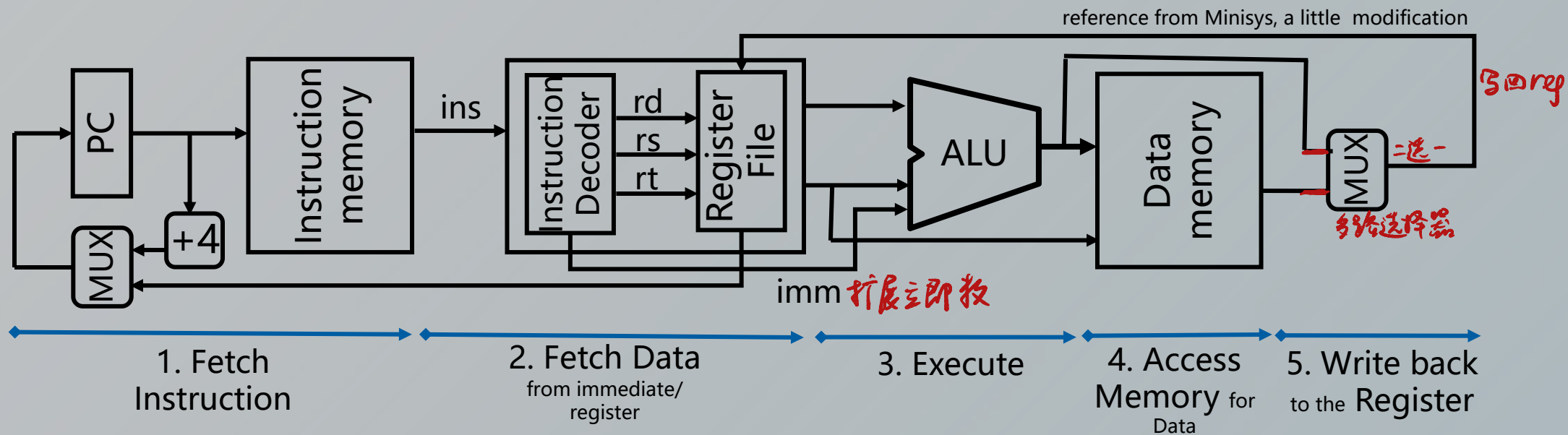
The **opC** of **R-Type** instruction is **6'b00\_0000**

## BASIC INSTRUCTION FORMATS

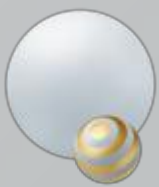
R	opcode		rs	rt		rd		shamt		funct	
	31		26 25	21 20		16 15		11 10		6 5 0	
I	opcode		rs	rt		immediate					
	31		26 25	21 20		16 15		0			
J	opcode		address								
	31		26 25	0							



# Data Path

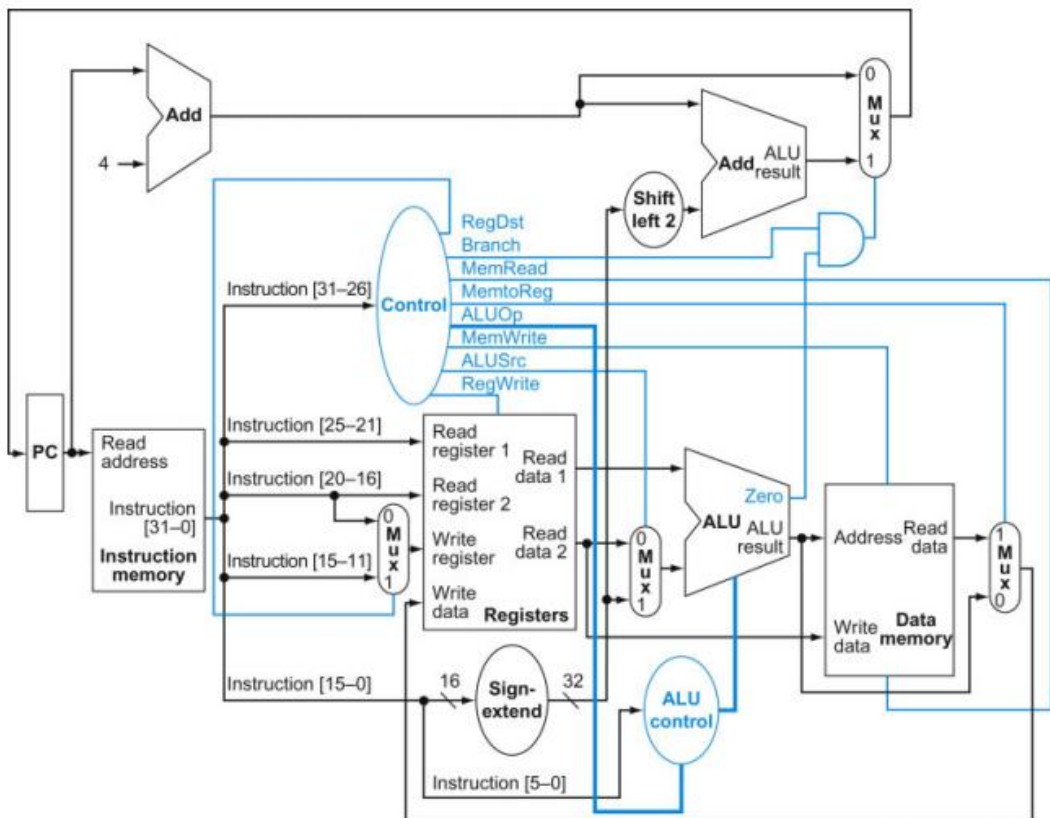


	1. Instruction fetch	2. Data Fetch	3. Instruction Execute	4. Memory Access	5. Register WriteBack
add[R]	Y	Y	Y		Y
addi[I]	Y	Y	Y		Y
sw[I]	Y	Y	Y	Y	
lw[I]	Y	Y	Y	Y	Y
branch[I]	Y	Y	Y		
jump[J]	Y	Y	Y		
jal [J]	Y	Y	Y		



# Control Path

'**opcode**' and '**funct**' are inputs, '**Control**' generate the control signals which will be used in other modules.



Source: H&P textbook

## BASIC INSTRUCTION FORMATS

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

## Instruction Analysis:

产生控制信号

➤ 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

获取数据信息

➤ Part 2: get data/information about the 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])

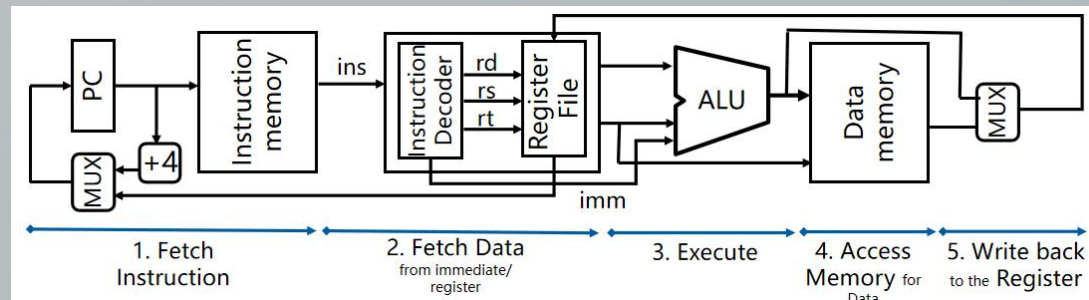
➤ **address**(instruction[25:0])





# Controller

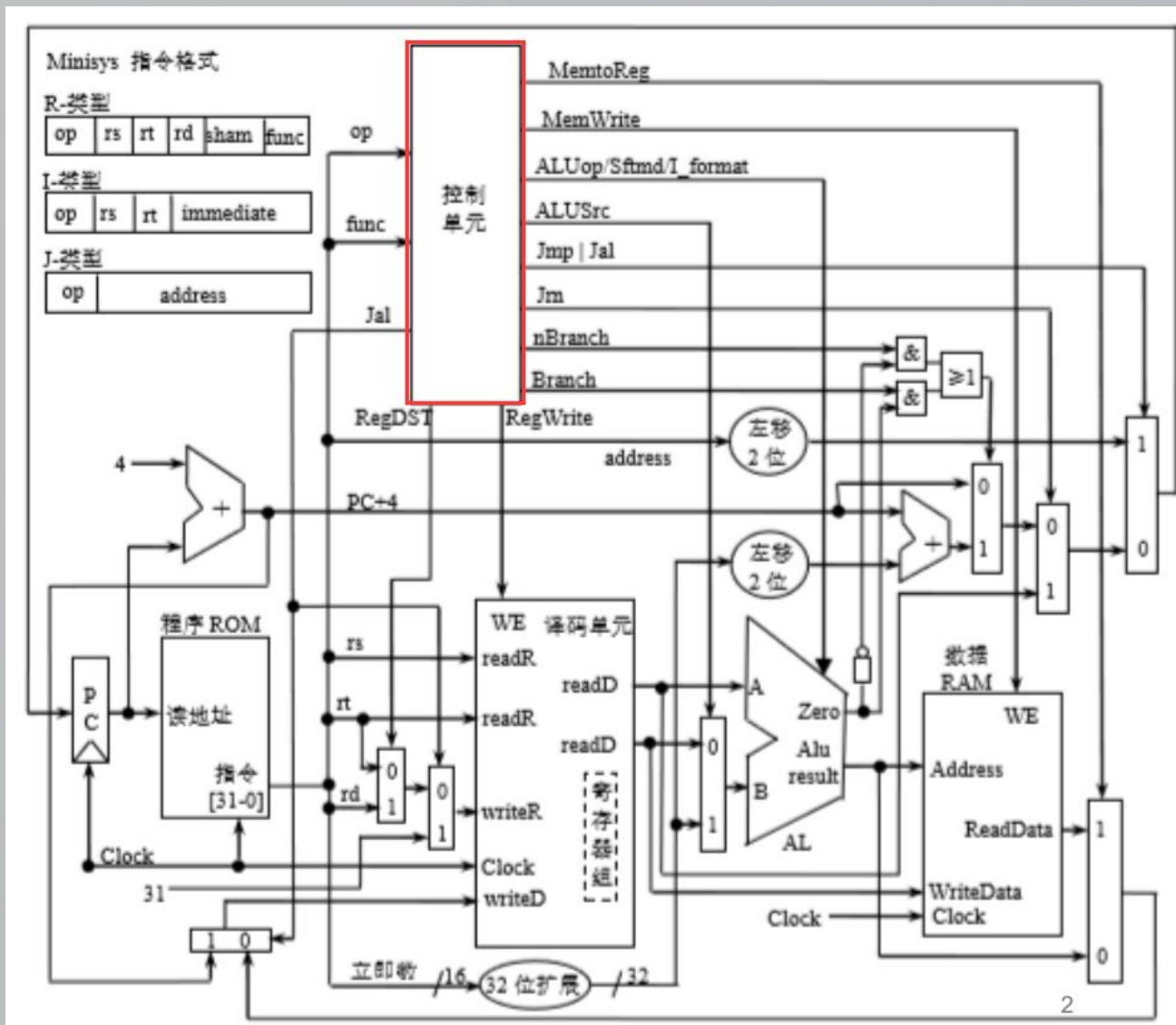
Why a controller is needed?



Module	How To Process	Instructions and Comments
<b>IFetch</b>	Determine how to update the value of PC register	1. (pc+4)+immediate(sign extended) (bne [I]) 2. the value of \$31 register (jr [R]) 3. {(pc+4)[31:28],lableX[25:2],2'b00} (jal, j [J]) 4. pc+4 (other instruction except branch, jr, j and jal [R\I])
<b>Decoder</b>	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 Get the address of register to be written	1. Data memory (lw[I]) -> rt 2. ALU ([R]) -> rd 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])
<b>Memory</b>	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])
<b>ALU</b>	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)



# Controller continued



**Q1:** How to determine the type of the instruction(R, I or J) from the 32bits machine code? 高6位

**Q2:** What's the usage of function code in the instruction? 区分R类型指令功能

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

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

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



# 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
                                写入寄存器类型
output RegDST;           // 1 indicate destination register is "rd"(R),otherwise it's "rt"(I)
                                写入数据来源.
output MemtoReg;         // 1 indicate read data from memory and write it into register
                                是否需要写 reg.
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
    
```

## BASIC INSTRUCTION FORMATS

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

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

```

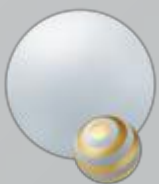
output I_format; // I_format is 1 bit width port
/* 1 indicate the instruction is I-type but isn't
   "beq","bne","LW" or "SW" */
    
```

```

output[1:0] ALUOp; // ALUOp is multi bit width port
/* 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 these output ports ?





# Controller continued

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

**“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

$RegWrite = (R\_format || Lw || Jal || I\_format) \&\& !(Jr)$

## BASIC INSTRUCTION FORMATS

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



# Controller continued

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

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;



# Practice1

1. Implement the sub-module of CPU: Controller.
2. Verify the Controller's function by simulation.

**NOTE: Following table is Not a complete set of tests, just a reference.**

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

3. List the signals which are used by the Controller(NOTE: Signals' name are determined by designer, following table could be used as a reference)

name	from	to	bits	function
opCode	IFetch	Controller	6	the opcode of the 32bits instruction
regWrite	Controller	Decoder	1	1 indicate write register(R,I(lw)), otherwise it's not
...				



# Tips: a reference to build a testbench

```
module control32_tb
```

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

```
    //instance the module "control32", bind the ports
```

```
    control32 c32
```

```
    (Opcode, Function_opcode,  
     Jr, Branch, nBranch, Jmp, Jal,  
     RegDST, MemtoReg, RegWrite, MemWrite,  
     ALUSrc, ALUOp, Sftmd, I_format);
```

```
    initial begin
```

```
        //an example: #0 add $3,$1,$2. get the machine code of 'add $3,$1,$2'
```

```
        // step1: edit the assembly code, add "add $3,$1,$2"
```

```
        // step2: open the assembly code in Minisys1Assembler2.2, do the assembly procession
```

```
        // step3: open the "output/prgmips32.coe" file, find the related machine code of 'add $3,$1,$2'
```

```
        //in "0x00221820", 'Opcode' is 6'h00, 'Function_opcode' is 6'h20
```

```
        Opcode = 6'h00;
```

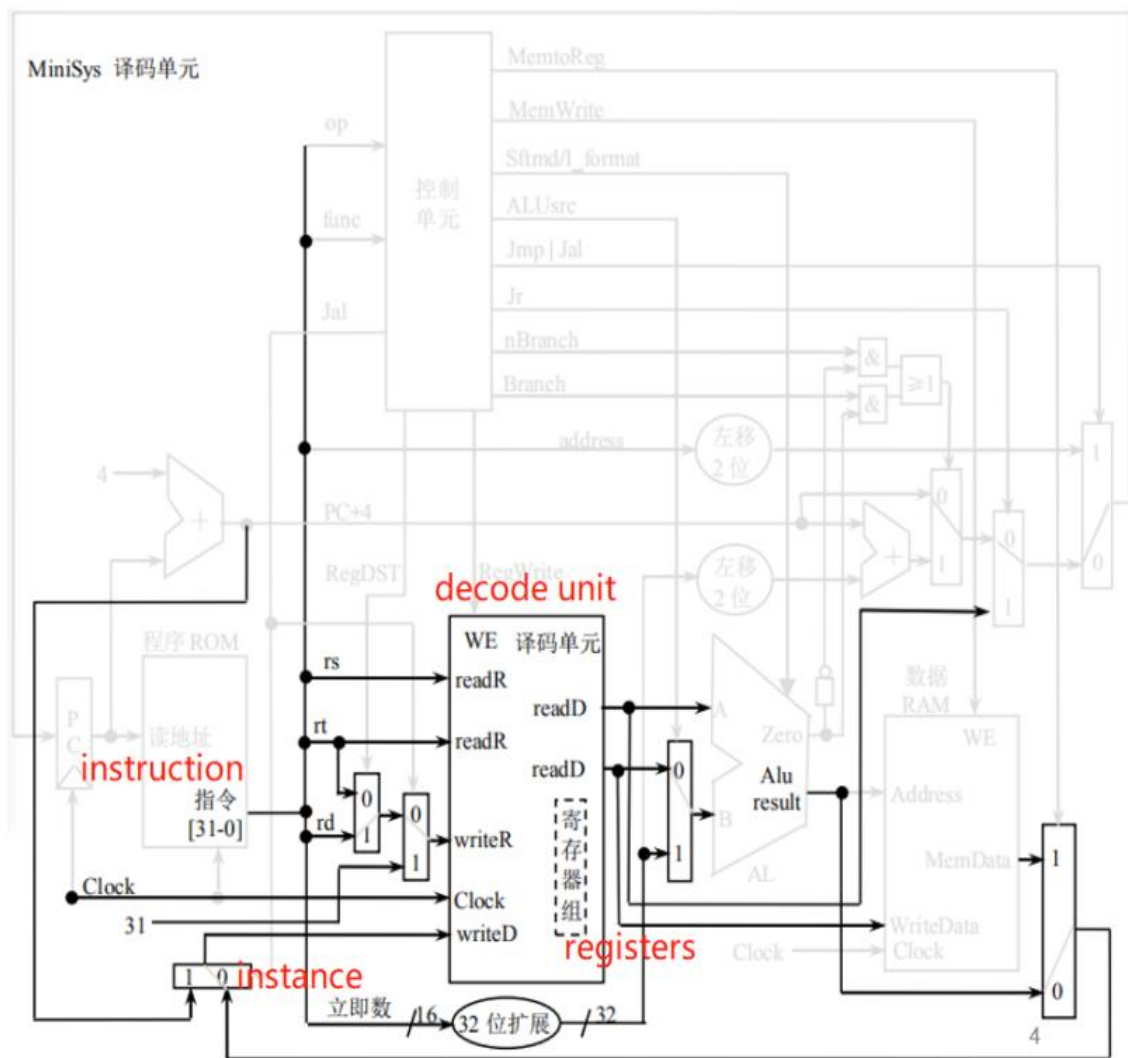
```
        Function_opcode = 6'h20;
```

```
        #200 //...
```

```
    end
```

```
endmodule
```

# Data Path(1) 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

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





# Decoder continued

## BASIC INSTRUCTION FORMATS

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

## ➤ Registers(Register File)

### -Inputs

#### ➤ read address

- [R] add: rs,rt
- [R] jr : [31]
- [I] addi: rs

#### ➤ write address

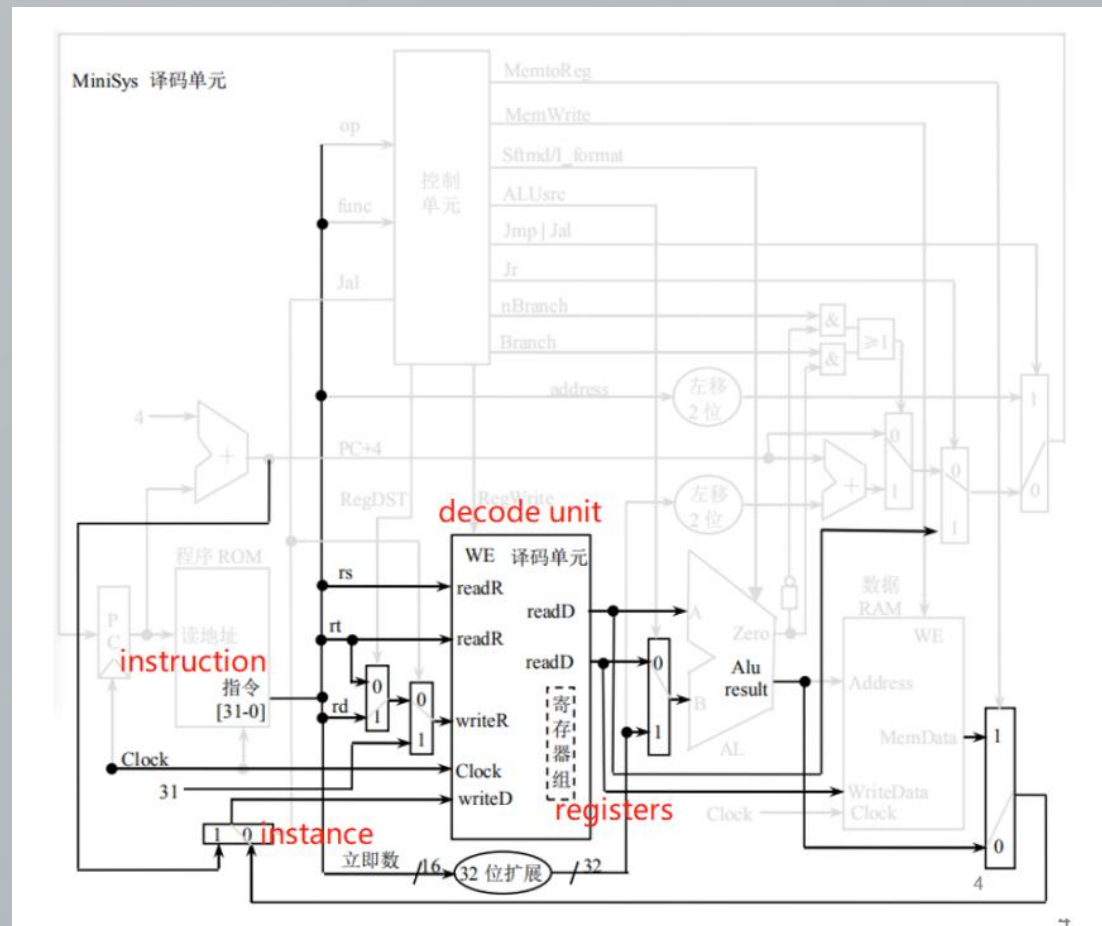
- [R] add: rd
- [J] jal : [31]
- [I] addi: rt

#### ➤ 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



## ➤ Registers(Register File)

### -Outputs

- read data1
- read data2
- extended Immi

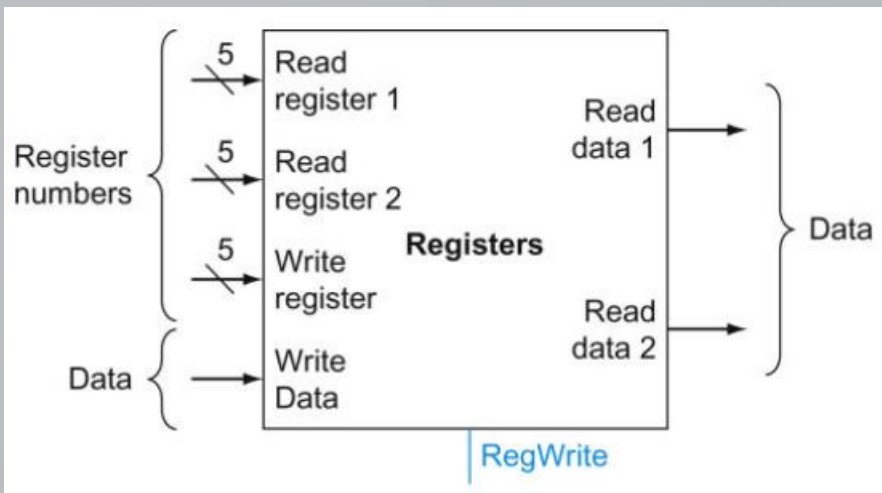


# Decoder continued

## Register File(Registers):

Almost all the instructions need to read or write register file in CPU;

32 common registers with same bitwidth: 32



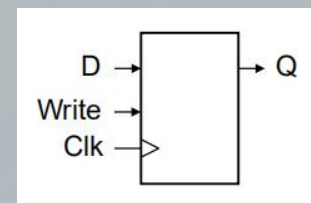
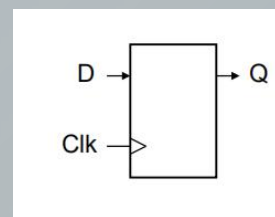
//verilog tips:

reg <sup>32位</sup>31:0; register <sup>32位</sup>0:31; 寄存器组

assign Read data 1 = register[Read register 1];  
register[Write register] <= WriteData;

## BASIC INSTRUCTION FORMATS

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



Q1:

How to avoid the conflict 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?



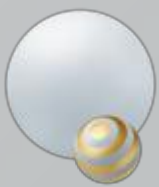
# Practice2

## BASIC INSTRUCTION FORMATS

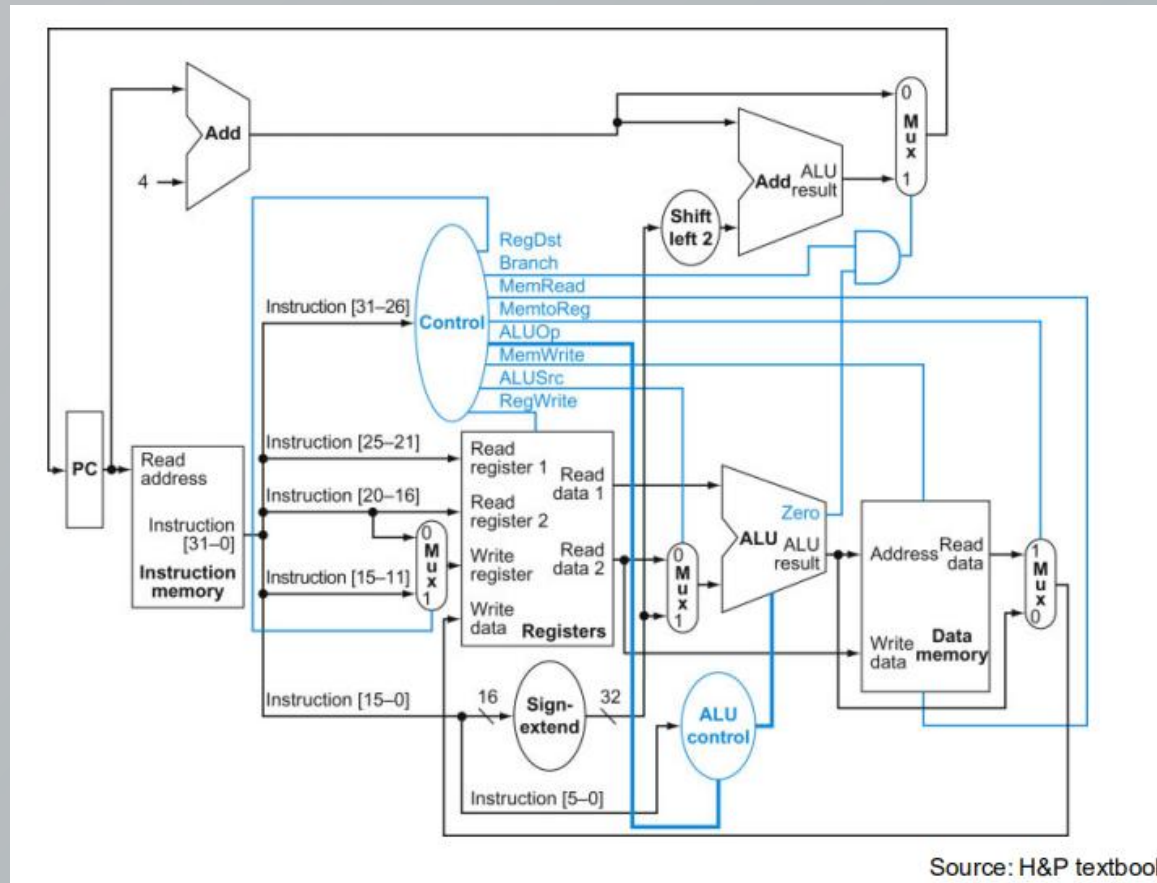
<b>R</b>	opcode	rs	rt	rd	shamt	funct
	31 26 25	21 20	16 15	11 10	6 5	
<b>I</b>	opcode	rs	rt	immediate		
	31 26 25	21 20	16 15			
<b>J</b>	opcode	address				
	31 26 25					

- 1. Implement the sub-module of CPU: Decoder
  - There are **32** registers(each register is **32bits**), All the registers are **readable** and **writeable** except \$0, **\$0 is readonly**.
  - The **reading** should be done at any time while **writing** only happens on the **posedge** of the clock.
  - The register file should support **R/I/J** type instructions(extend the immediate to be 32bits if needed).
    - such as: **add; addi; jr; lw, sw; jal;**
- 2. Verify its function by simulation. NOTES: The verification should be done on the full set of testcase.
- 3. List the signals which are used by the Decoder (NOTE: Signals' name are determined by designer)  
tips: following table could be used as a reference

name	from	to	bits	function
regWrite	Controller	Decoder	1	1 means write the register identified by writeAdress
imme	Decoder	ALU	32	the signextended immediate
readRegister1	IFetch	Decoder	5	the address of read register instruction[25:21]
...				



# Tips: Control Path & Data Path of CPU



**Control Path:** Interpret 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