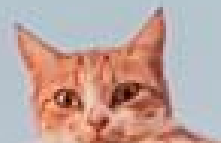


Computer Organization

Lab9 CPU Design1: ISA, Controller and Decoder

Topics

- CPU Design(1)
  - ISA, Assembler
  - Control Path
  - Data Path(1)
    - Decoder

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

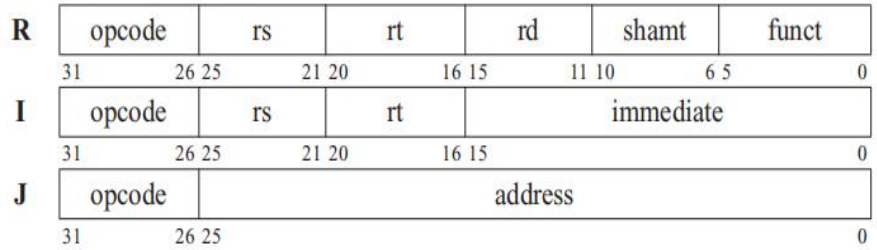
MIPS_Green_Sheet.pdf

NOTE:

Minisys is a subset of MIPS32.

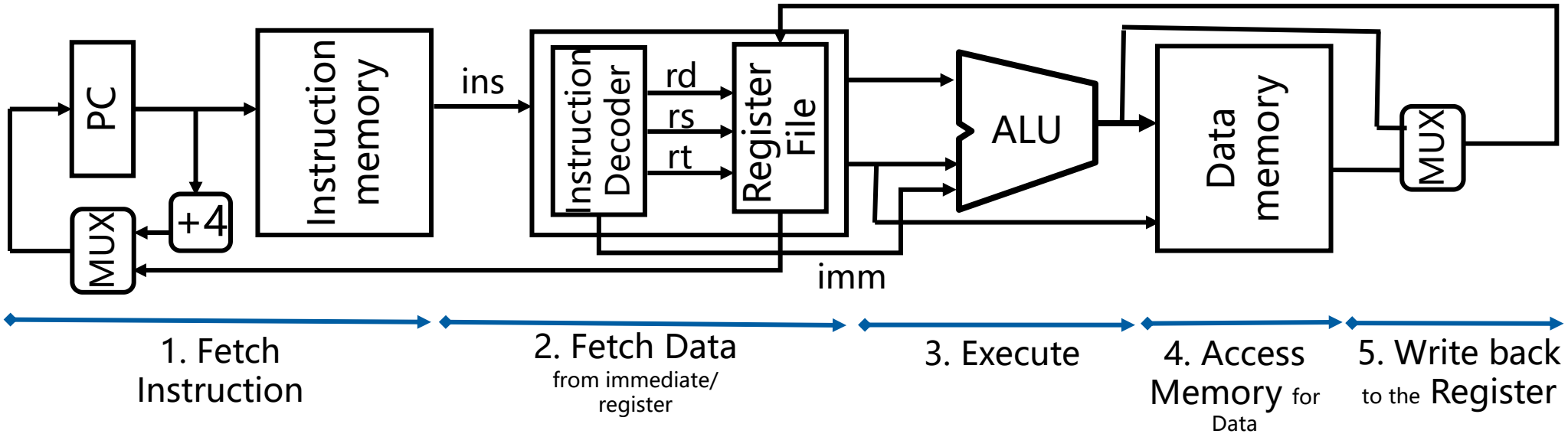
The **opC** of **R-Type** instruction is **6'b00_0000**

BASIC INSTRUCTION FORMATS



4 Data Path

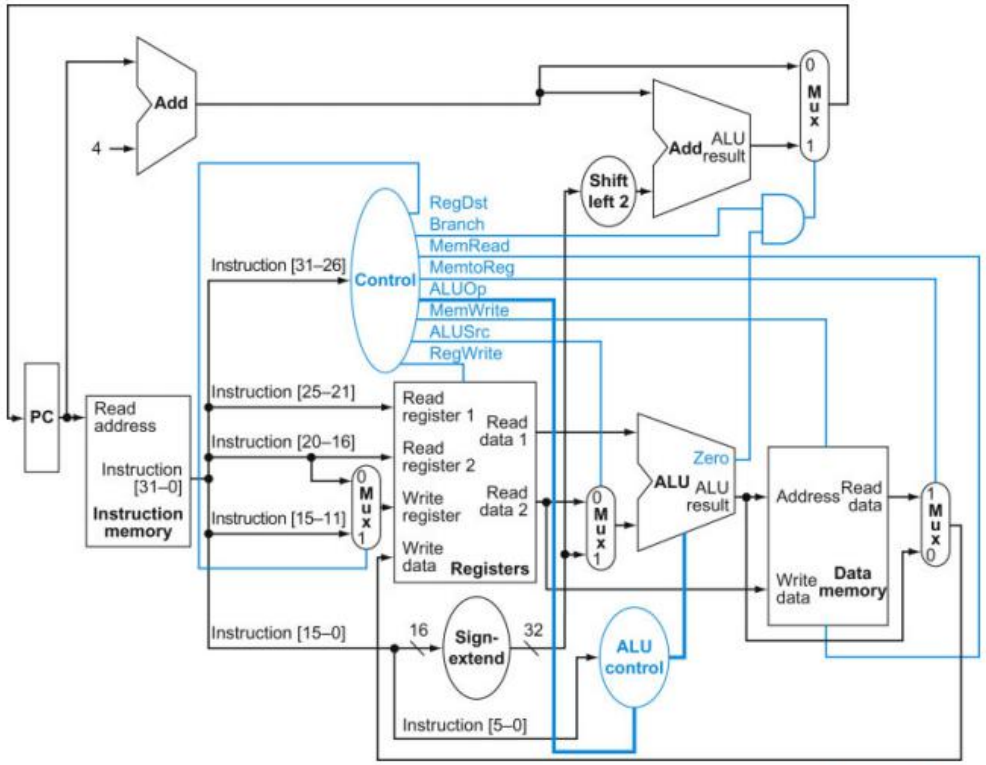
reference from Minisys, a little modification



	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		

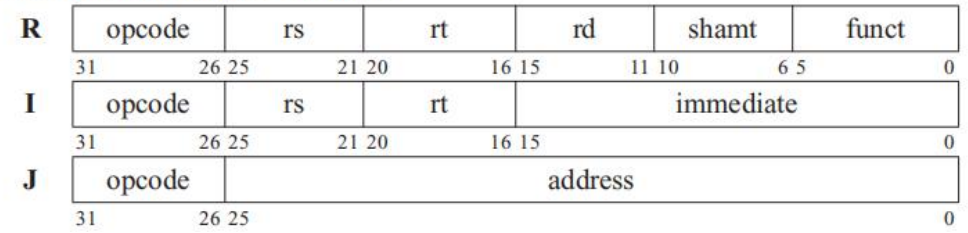
CONTROL PATH

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

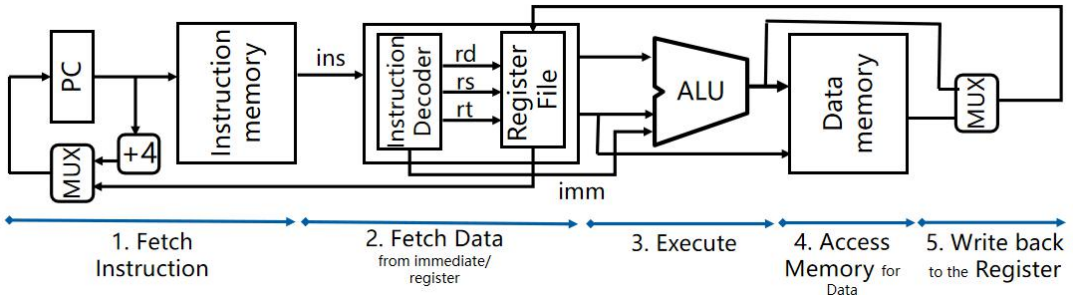


Instruction Analysis:

- **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])

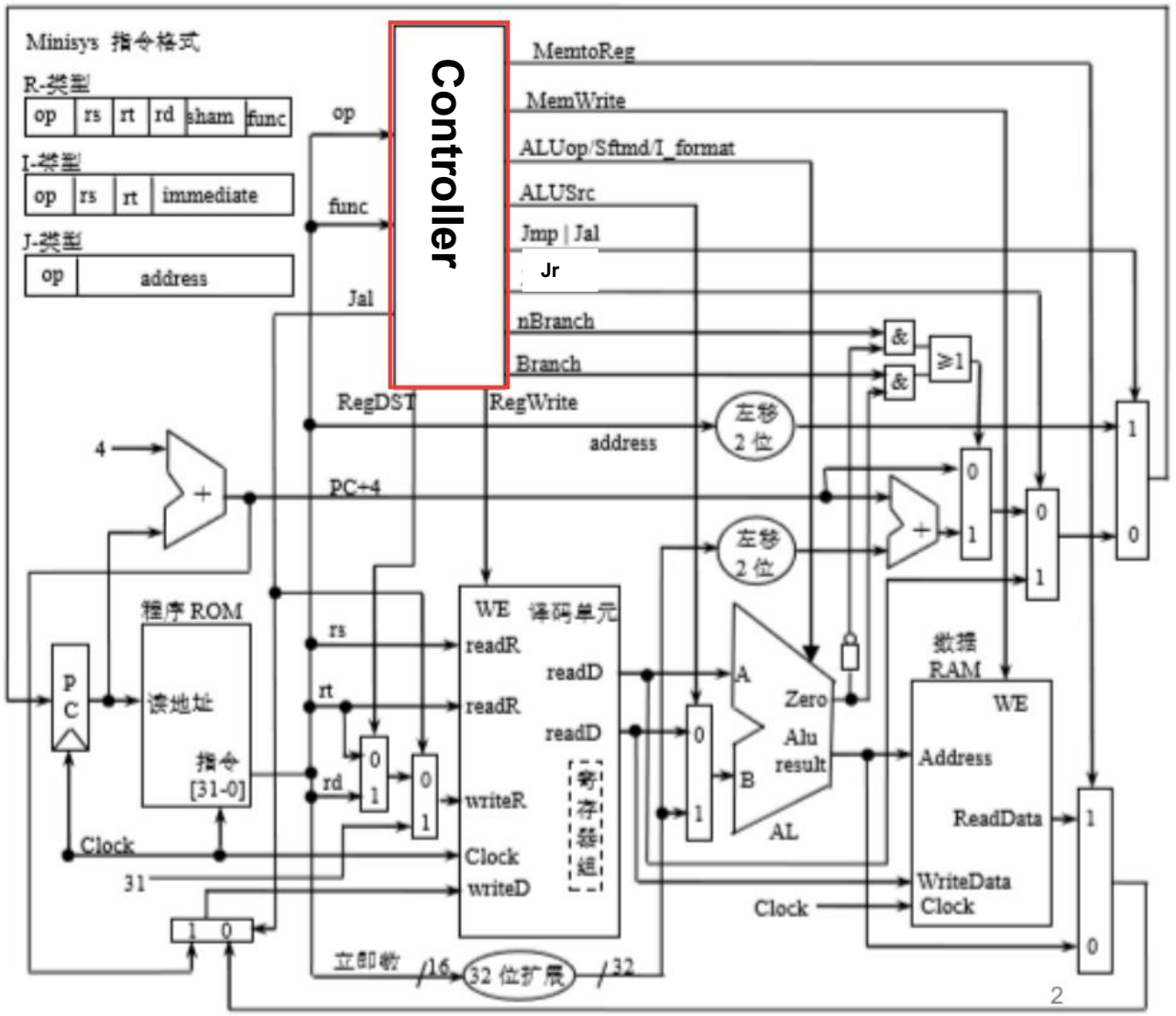
Controller

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]) 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])
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 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])
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)

Controller continued



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

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

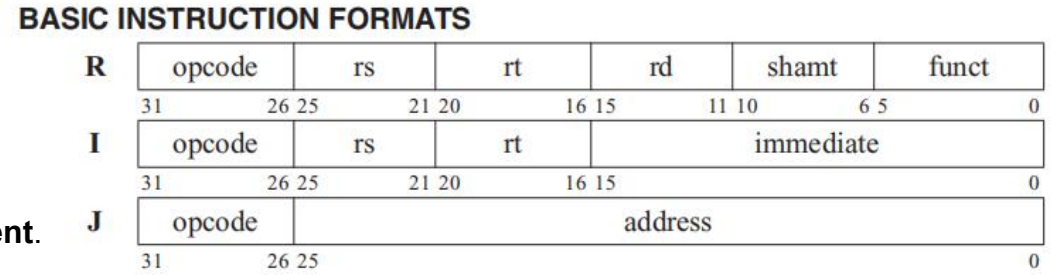
Q3: How to generate these control signals?

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.

Controller continued

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



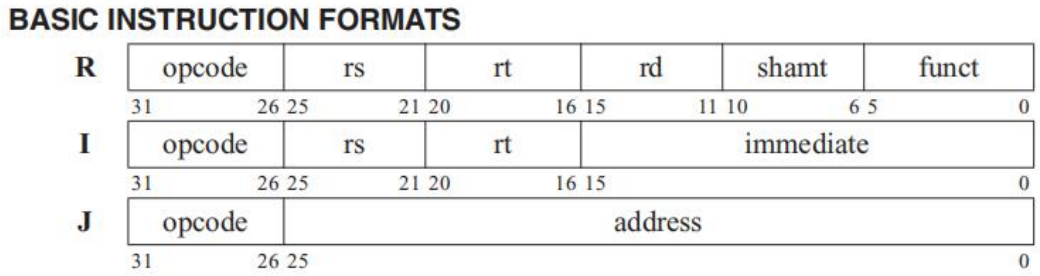

```
input[5:0]   Op;          // instruction[31..26], opcode
input[5:0]   Func;        // instructions[5..0], funct
output       Jr ;           // 1 indicates the instruction is "jr", otherwise it's not "jr"
output       Jmp;           // 1 indicate the instruction is "j", otherwise it's not
output       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 ?

Controller continued



"**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)

Controller continued

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

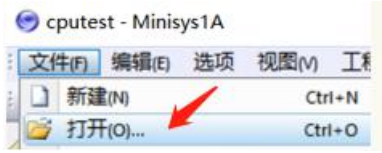
TIPS: Minisys1Assemblerv2.2(An Asambler on Minisys) could help to generate the corresponding instructions(32bit), it could be found at below address:
https://bb.sustech.edu.cn/webapps/blackboard/content/listContentEditable.jsp?content_id=_281670_1&course_id=_3602_1

Tips: a reference to build a testbench

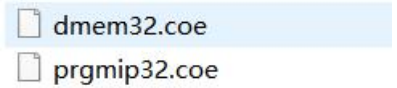
```
module control32_tb
    //reg type variables are use for binding with input ports
    reg [5:0] Opcode,Function_opcode;
    //wire type variables are use for binding with output ports
    wire [1:0] ALUOp;
    wire Jr,RegDST,ALUSrc,MemtoReg,RegWrite,MemWrite,Branch,nBranch,Jmp,Jal,I_format,Sftmd;

    //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
```

How To Use “Minisys1Assemblerv2.2 ”

- Step1. **Open** the assembly **source file**

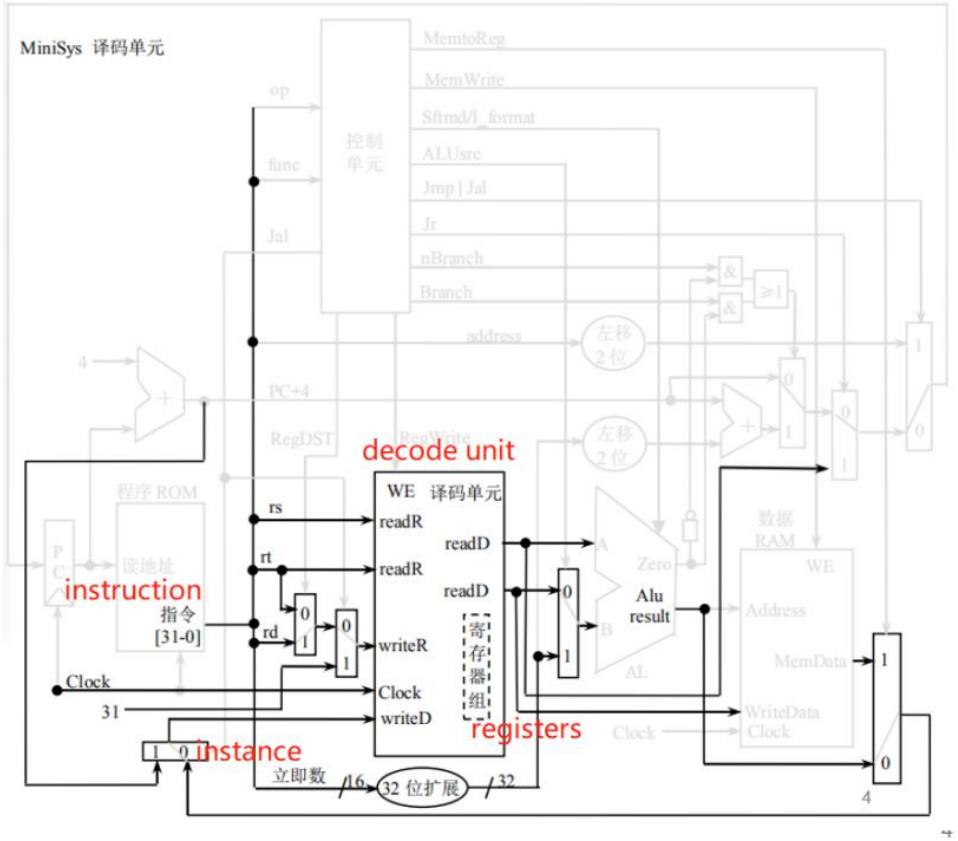


- Step2. “**工程**”-》“**64KB**”（the size of Instruction memory and data memory）-》“**A 汇编**”
- Step3. The coe files could be found at the sub-directory: ”**output**”
  - The initial data of data memory could be found in file “dmem32.coe”
  - The machine code of Minisys instruction could be found in the file “**prgmip32.coe**”
  - Following screenshot is an example, the machine code is recorded in **hexadecimal**.




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

14 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**

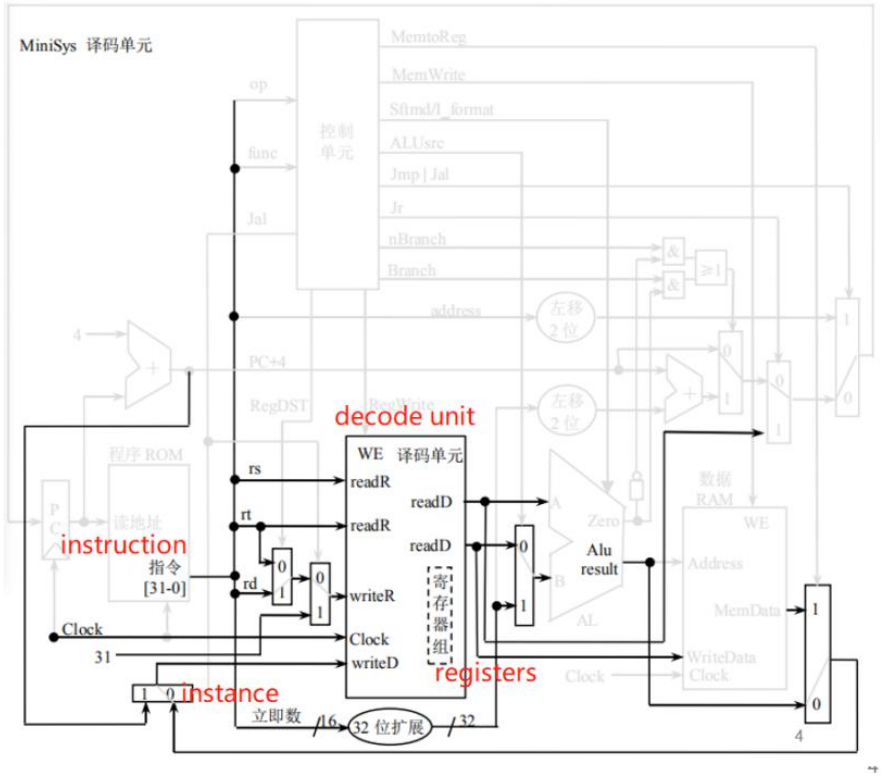
BASIC INSTRUCTION FORMATS

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				

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				

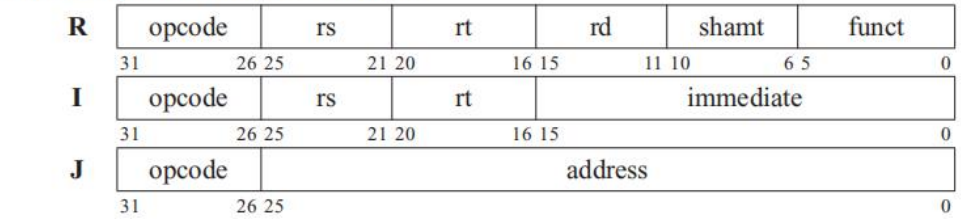


- **Register File-Outputs**
  - read data1
  - read data2
  - extended Immi

- **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

Decoder continued

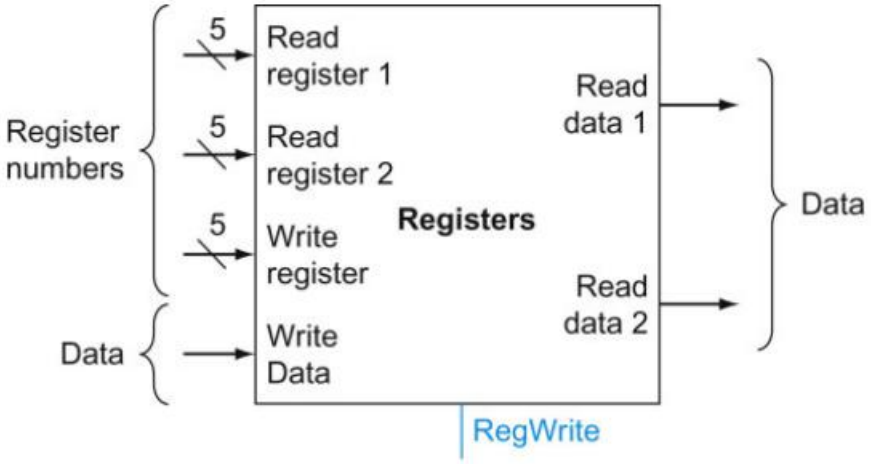
BASIC INSTRUCTION FORMATS

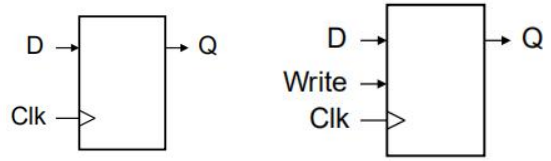


Register File:

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

32 common registers with same bitwidth: 32





Q1:

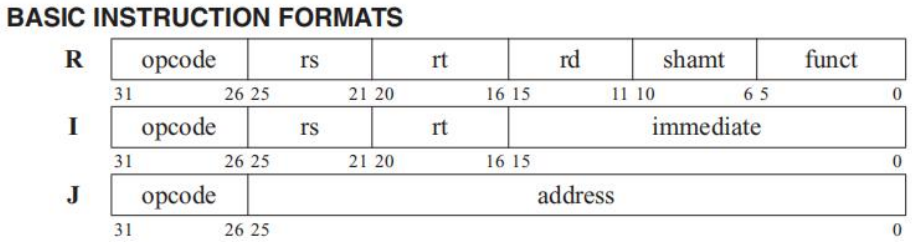
How to avoid the confliction between register read and register write?

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;
```

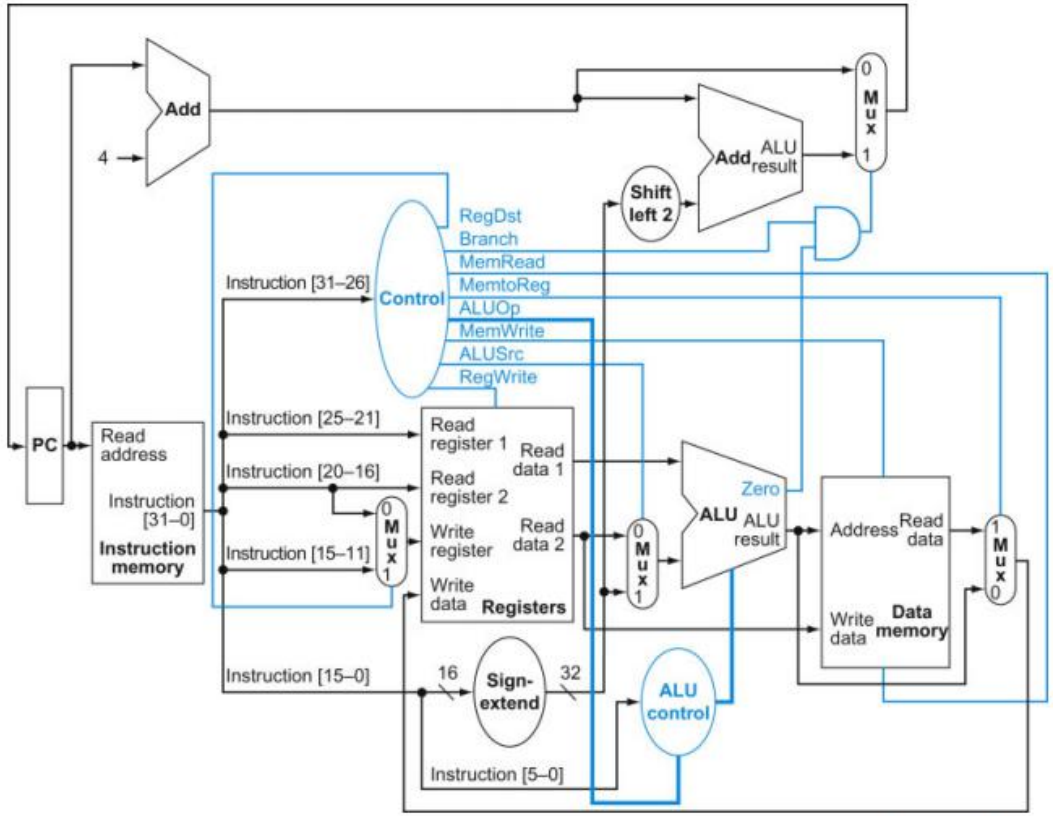
Practice2



- 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 in the decoder (NOTE: Signals' name are determined by designer)

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



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