

31 条指令单周期 CPU 设计

1. 实验介绍

在本次实验中，我们将使用 Verilog HDL 语言实现 31 条 MIPS 指令的 CPU 的设计和仿真。

2. 实验目标

- 深入了解 CPU 的原理。
- 画出实现 31 条指令的 CPU 的通路图。
- 学习使用 Verilog HDL 语言设计实现 31 条指令的 CPU。

3. 实验原理

***注意：**为了便于使用 Mars 进行测试，CPU 设计做如下规定：

Mars 中 CPU 采用冯诺依曼结构，我们使用它的 default 内存设置，故指令存储起始地址为：0x00400000，数据存储起始地址为 0x10010000，由于我们设计的 CPU 为哈佛结构，指令和数据的起始地址都为 0，所以最后请在 PC 模块、指令存储模块、数据存储模块进行地址映射（将逻辑地址转换为物理地址 0）。

1) 需要实现的 31 条 MIPS 指令，见图

Mnemonic Symbol	Format						Sample
Bit #	31..26	25..21	20..16	15..11	10..6	5..0	
R-type	op	rs	rt	rd	shamt	func	
add	000000	rs	rt	rd	0	100000	add \$1,\$2,\$3
addu	000000	rs	rt	rd	0	100001	addu \$1,\$2,\$3
sub	000000	rs	rt	rd	0	100010	sub \$1,\$2,\$3
subu	000000	rs	rt	rd	0	100011	subu \$1,\$2,\$3
and	000000	rs	rt	rd	0	100100	and \$1,\$2,\$3
or	000000	rs	rt	rd	0	100101	or \$1,\$2,\$3
xor	000000	rs	rt	rd	0	100110	xor \$1,\$2,\$3
nor	000000	rs	rt	rd	0	100111	nor \$1,\$2,\$3
slt	000000	rs	rt	rd	0	101010	slt \$1,\$2,\$3
sltu	000000	rs	rt	rd	0	101011	sltu \$1,\$2,\$3
sll	000000	0	rt	rd	shamt	000000	sll \$1,\$2,10
srl	000000	0	rt	rd	shamt	000010	srl \$1,\$2,10
sra	000000	0	rt	rd	shamt	000011	sra \$1,\$2,10
sllv	000000	rs	rt	rd	0	000100	sllv \$1,\$2,\$3
srlv	000000	rs	rt	rd	0	000110	srlv \$1,\$2,\$3
srav	000000	rs	rt	rd	0	000111	srav \$1,\$2,\$3
jr	000000	rs	0	0	0	001000	jr \$31
Bit #	31..26	25..21	20..16	15..0			
I-type	op	rs	rt	immediate			
addi	001000	rs	rt	Immediate(- ~ +)		addi \$1,\$2,100	
addiu	001001	rs	rt	Immediate(- ~ +)		addiu \$1,\$2,100	
andi	001100	rs	rt	Immediate(0 ~ +)		andi \$1,\$2,10	
ori	001101	rs	rt	Immediate(0 ~ +)		andi \$1,\$2,10	
xori	001110	rs	rt	Immediate(0 ~ +)		andi \$1,\$2,10	
lw	100011	rs	rt	Immediate(- ~ +)		lw \$1,10(\$2)	
sw	101011	rs	rt	Immediate(- ~ +)		sw \$1,10(\$2)	
beq	000100	rs	rt	Immediate(- ~ +)		beq \$1,\$2,10	
bne	000101	rs	rt	Immediate(- ~ +)		bne \$1,\$2,10	
slti	001010	rs	rt	Immediate(- ~ +)		slti \$1,\$2,10	
sltiu	001011	rs	rt	Immediate(- ~ +)		sltiu \$1,\$2,10	
lui	001111	00000	rt	Immediate(- ~ +)		Lui \$1, 10	
Bit #	31..26	25..0					
J-type	op	Index					
j	000010	address				j 10000	
jal	000011	address				jal 10000	

2) 单周期数据通路设计的一般性方法

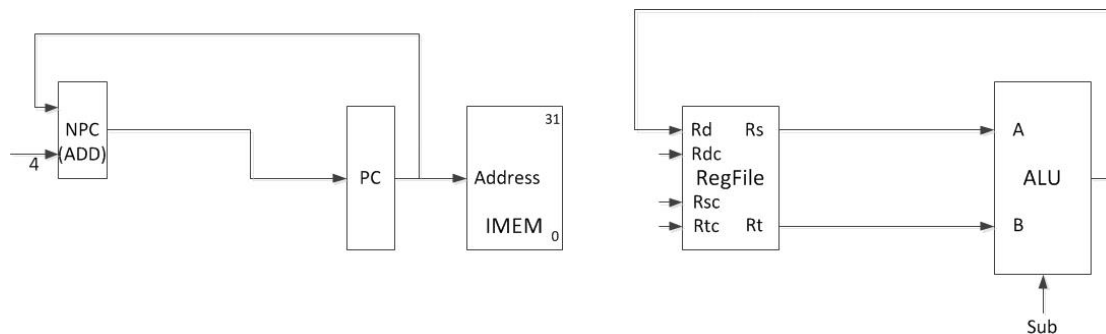


图 8.4.3 SUBU 指令通路

c) SLL

- 所需操作：取指令、 $R[rd] \rightarrow R[rt] \ll \text{unsign_ext}(\text{imm5})$ 、 $PC \rightarrow PC+4$
- 所需器件：PC 寄存器、指令存储器、寄存器文件、ALU、符号扩展模块 (S_EXT)

指令	PC	IM	RF	S_EXT	ALU	
			WData		A	B

指令	PC	IM	RF	S_EXT	ALU	
			WData		A	B
ADDU	PC+4	PC	ALU		RF.RD1	RF.RD2
SUBU	PC+4	PC	ALU		RF.RD1	RF.RD2
SLL	PC+4	PC	ALU	IM[10:6]	EXT	RF.RD2

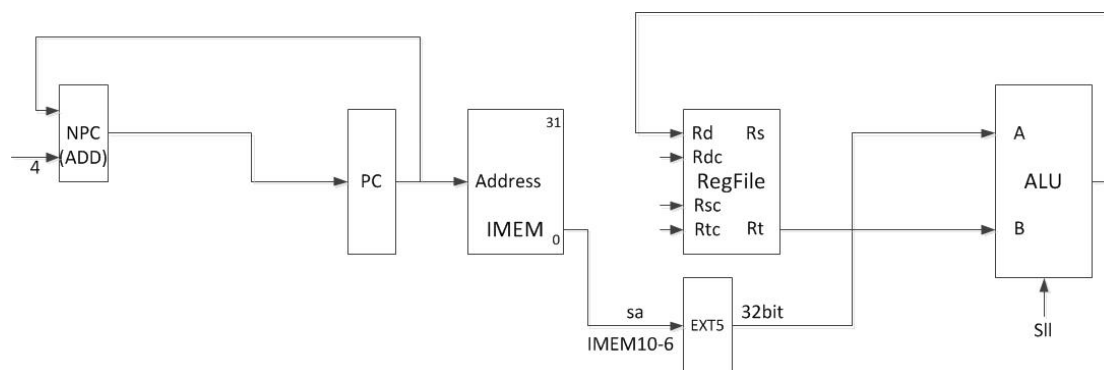


图 8.4.4 SLL 指令通路

d) ORI

- 所需操作：取指令、立即数符号扩展、 $R[rt] \rightarrow R[rs] \text{ or } \text{unsign_ext}(\text{imm16})$ 、 $PC \rightarrow PC+4$

指令	PC	IM	RF	S_EXT	ALU	
			WData		A	B

指令	PC	IM	RF	S_EXT	ALU	
			WData		A	B
ADDU	PC+4	PC	ALU		RF.RD1	RF.RD2
SUBU	PC+4	PC	ALU		RF.RD1	RF.RD2

SLL	PC+4	PC	ALU	IM[10:6]	EXT	RF.RD2
ORI	PC+4	PC	ALU	IM[15:0]	RF.RD1	S_EXT

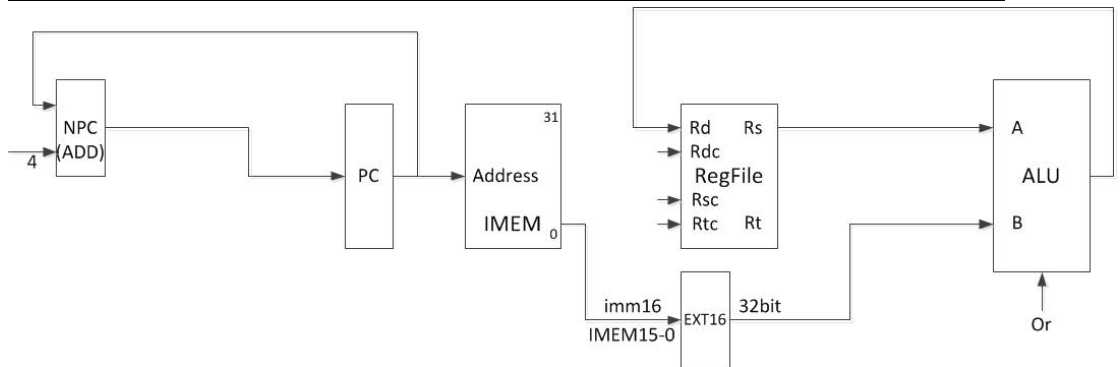


图 8.4.5 ORI 指令通路

- 所需器件：PC 寄存器、指令存储器、符号扩展模块、寄存器文件、ALU
- 更新表格

e) LW

- 所需操作：取指令、立即数符号扩展、 $R[rs] + \text{sign_ext}(\text{imm16})$ 、 $R[rt] \rightarrow \text{MEM}[R[rs] + \text{sign_ext}(\text{imm16})]$ 、 $\text{PC} \rightarrow \text{PC} + 4$

指令	PC	IM	RF	S_EXT	ALU	DM
			WData		A B	Data in addr

指令	PC	IM	RF	S_EXT	ALU	DM
			WData		A B	Data in addr
ADDU	PC+4	PC	ALU		RF.RD1 RF.RD2	
SUBU	PC+4	PC	ALU		RF.RD1 RF.RD2	
SLL	PC+4	PC	ALU	IM[10:6]	EXT RF.RD2	
ORI	PC+4	PC	ALU	IM[15:0]	RF.RD1 S_EXT	
LW	PC+4	PC	DM	IM[15:0]	RF.RD1 S_EXT	ALU

- 所需器件：PC 寄存器、指令存储器、符号扩展模块、寄存器文件、alu、数据存储器
- 更新表格

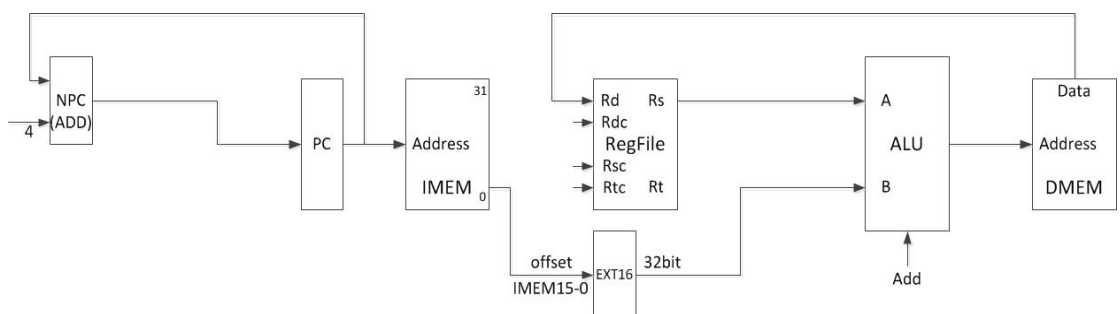


图 8.4.6 LW 指令通路

f) SW

- 所需操作： 取指令、立即数符号扩展、 $R[rs] + \text{sign_ext}(\text{imm16}) \rightarrow R[rt]$ 、 $PC \rightarrow PC + 4$

指令	PC	IM	RF	S_EXT	ALU		DM	
			WData		A	B	Data in	addr

指令	PC	I M	RF	S_EXT	ALU		DM	
			WData		A	B	Data in	addr
ADD	PC+ 4	P C	ALU		RF.RD 1	RF.RD 2		
SUB	PC+ 4	P C	ALU		RF.RD 1	RF.RD 2		
SLL	PC+ 4	P C	ALU	IM[10:6]]	EXT	RF.RD 2		
ORI	PC+ 4	P C	ALU	IM[15:0]]	RF.RD 1	S_EXT		
LW	PC+ 4	P C	DM	IM[15:0]]	RF.RD 1	S_EXT		ALU
SW	PC+ 4	P C		IM[15:0]]	RF.RD 1	S_EXT	RF.RD 2	ALU

- 所需器件： PC 寄存器、指令存储器、符号扩展模块、寄存器文件、alu、数据存储器
- 更新表格

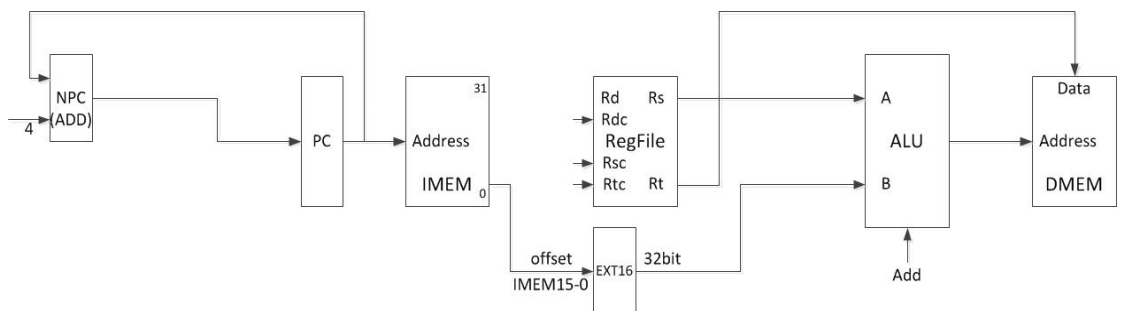


图 8.4.7 SW 指令通路

g) BEQ

- 所需操作： 取指令、立即数左移两位并且符号扩展 $\text{target_offset} \rightarrow \text{sign_extend}(\text{offset} \ll 2)$ 、 $\text{if} (GPR[rs] == GPR[rt]) \text{ then } PC \rightarrow PC + \text{target_offset} \text{ else } PC \rightarrow PC + 4$

指令	PC	IM	RF	S_EXT	ALU	DM
----	----	----	----	-------	-----	----

			WData		A	B	Data in	addr
--	--	--	-------	--	---	---	------------	------

指令	PC	I M	RF	S_EXT	ALU		DM	
			WData		A	B	Data in	addr
ADD U	PC+4	P C	ALU		RF.R D1	RF.R D2		
SUB U	PC+4	P C	ALU		RF.R D1	RF.R D2		
SLL	PC+4	P C	ALU	IM[10:6]	EXT	RF.R D2		
ORI	PC+4	P C	ALU	IM[15:0]	RF.R D1	S_EX T		
LW	PC+4	P C	DM	IM[15:0]	RF.R D1	S_EX T		ALU
SW	PC+4	P C		IM[15:0]	RF.R D1	S_EX T	RF.R D2	ALU
BEQ	PC+S_E XT	P C		IM[15:0]< <00	RF.R D1	RF.R D2		

- 所需器件： PC 寄存器、指令存储器、符号扩展模块、寄存器文件、alu、数据存储器
- 更新表格

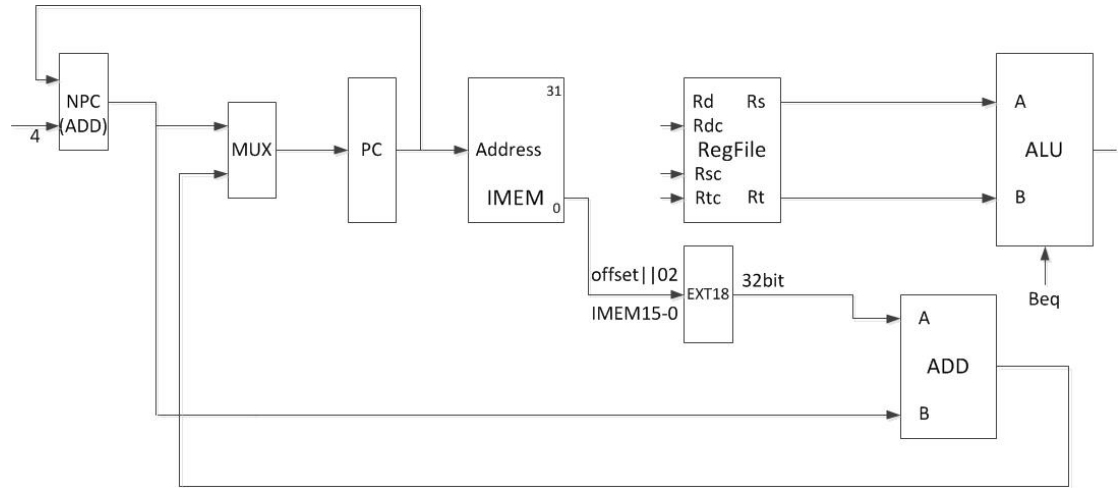


图 8.4.8 BEQ 指令通路

h) J

所需操作： 取指令、指令中的 26 位地址左移两位与 PC 的高四位合并为一个 32 为地址

指令	PC	IM	RF	S_EXT	ALU	DM
----	----	----	----	-------	-----	----

			WData		A	B	Data in	addr
--	--	--	-------	--	---	---	------------	------

指令	PC	I M	RF	S_EXT	ALU		DM	
			WData		A	B	Data in	ad dr
AD DU	PC+4	P C	ALU		RF. RD1	RF. RD2		
SU BU	PC+4	P C	ALU		RF. RD1	RF. RD2		
SL L	PC+4	P C	ALU	IM[10:6]]	EXT	RF. RD2		
OR I	PC+4	P C	ALU	IM[15:0]]	RF. RD1	S_E XT		
LW	PC+4	P C	DM	IM[15:0]]	RF. RD1	S_E XT		AL U
SW	PC+4	P C		IM[15:0]]	RF. RD1	S_E XT	RF. RD2	AL U
BE Q	PC+S_EXT	P C		IM[15:0]]<<00	RF. RD1	RF. RD2		
J	PC[31:28] IM[25:0]<<2	P C						

- 所需器件： PC 寄存器、指令存储器
- 更新表格

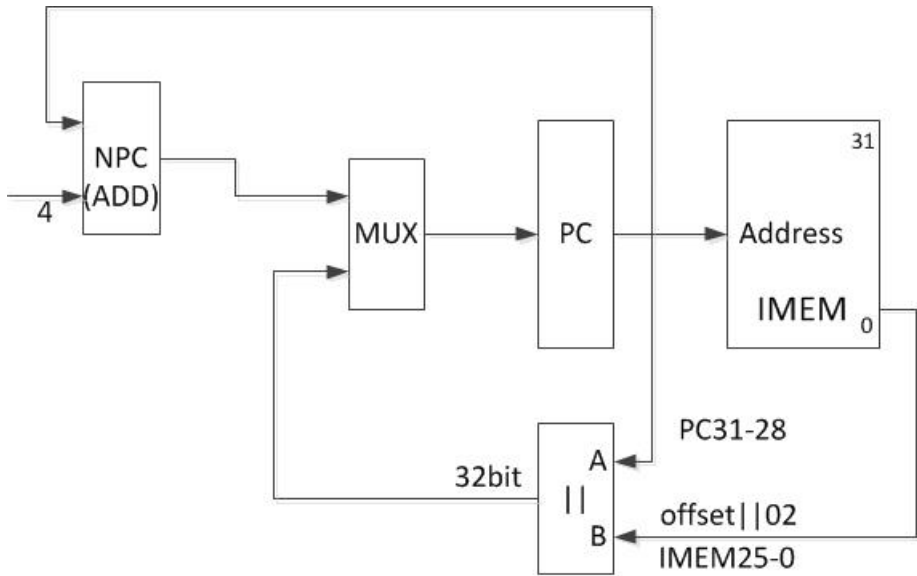


图 8.4.9 J 指令通路

i) 绘制通路图

若表格中某个输入端口有多个不同输入来源，则在此端口前加入一个多路选择器来选择各条指令所需的数据来源。根据上表，可画出数据通路图 8.4.10

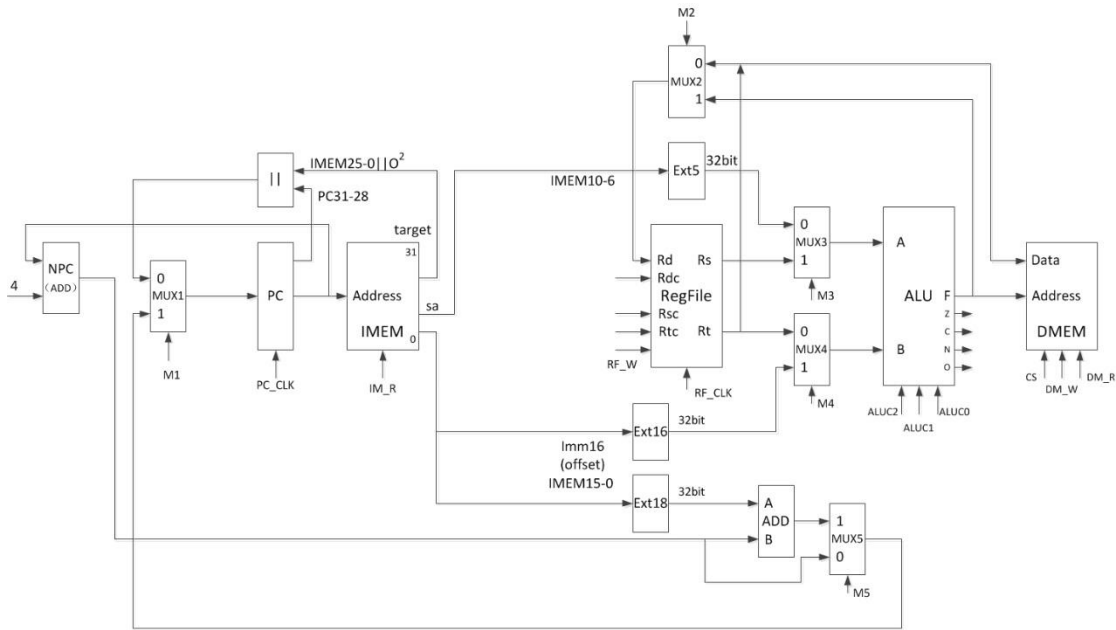


图 8.4.10 8 条指令数据通路图

- 3) 控制信号生成一般性方法
- 根据每条指令功能，在已形成的数据通路下，确定该条指令执行过程中所用部件的控制
 - 信号
 - 根据每条指令的所需控制信号列出 cpu 控制信号表
 - 写出布尔表达式
- 仍以 8 条指令为例：
- ADDU、SUBU、SLL、ORI、LW、SW、BEQ、J

a) 列出所有指令，各部件的所有的控制信号（即数据通路中没有被连接的信号）

	Addu	Subu	Sll	Ori	Lw	Sw	Beq	J
pcreg_ena								
i_ram_ena								
i_ram_wena								
d_ram_wena								
d_ram_ena								
rf_we								
rf_waddr								
rf_raddr1								
rf_raddr2								
s_ext_s								
alu_aluc [3:0]								

- b) 将选择器的选择信号加入表格（我们可以用选择器的输出目标来为选择器信号做一个暂时的命名）

	Addu	Subu	Sll	Ori	Lw	Sw	Beq	J
pcreg_ena								
i_ram_ena								
i_ram_wena								
d_ram_wena								
d_ram_ena								
rf_we								
rf_waddr								
rf_raddr1								
rf_raddr2								
s_ext_s								
alu_aluc [3:0]								
alu_a_mux								
alu_b_mux[1:0]								
rf_wdata_mux								
pc_mux[1:0]								

- c) 接着，我们将根据每条指令的功能以及数据通路填写这张表格：

Addu 所用部件：PC、IM、ALU、RF、rf_wdata_mux、alu_a_mux、alu_b_mux、pc_mux

未使用部件的控制信号可以使用 0 填写

取指令： pc_ena = 1

i_ram_ena = 1

i_ram_wena = 0

R[rd] R[rs]+R[rt]: rf_raddr1 = IM[25:21]

rf_raddr2 = IM[20:16]

rf_we = 1

rf_waddr = IM[15:11] //表示寄存器回写地址为 rd

rf_wdata_mux = 0 //表示数据来自于 IRAM 输出的[15:11]

alu_aluc = 0000 //表示 alu 作加法

alu_a_mux = 0 //表示数据来自于 RF 的 rdata1 输出

alu_b_mux = 00 //表示数据来自于 RF 的 rdata2 输出

PC PC+4: pc_mux = 00 //表示 PC 的来源为 PC+4

	Addu	Subu	Sll	Ori	Lw	Sw	Beq	J
pcreg_ena	1							
i_ram_ena	1							
i_ram_wena	0							
d_ram_wena	0							
d_ram_ena	0							
rf_we	1							
rf_waddr	IM[25:21]							
rf_raddr1	IM[20:16]							
rf_raddr2	IM[15:11]							
s_ext_s	0							
alu_aluc [3:0]	0000							
alu_a_mux	0							
alu_b_mux[1:0]	00							
rf_wdata_mux	0							
pc_mux[1:0]	00							

Subu 所用部件: PC、IM、ALU、RF、rf_wdata_mux、alu_a_mux、alu_b_mux、pc_mux

未使用部件的控制信号可以使用 0 填写

取指令: pc_ena = 1

i_ram_ena = 1

i_ram_wena = 0

R[rd] R[rs]+R[rt]: rf_raddr1 = IM[25:21]

rf_raddr2 = IM[20:16]

rf_waddr = IM[15:11]

rf_we = 1

rf_wdata_mux = 0

alu_aluc = 0001

alu_a_mux = 0

alu_b_mux = 00

PC PC+4: pc_mux = 00

SLL 所用部件: PC、IM、ALU、RF、S_EXT、rf_wdata_mux、alu_a_mux、alu_b_mux、pc_mux

取指令: pc_ena = 1

i_ram_ena = 1

```

i_ram_wena = 0
R[rd] R[rt] << unsign_ext(imm5): rf_raddr1 = xxxxx (IM[25:21])
rf_raddr2 = IM[20:16]
rf_we = 1
rf_waddr = IM[15:11]
rf_wdata_mux = 0
s_ext_s = 0 //表示 ext 0 扩展
alu_aluc = 0001
alu_a_mux = 1
alu_b_mux = 00
PC PC+4: pc_mux = 00
ORI 所用部件:PC、IM、ALU、RF、S_EXT、rf_wdata_mux 、alu_a_mux、alu_b_mux、
pc_mux
取指令: pc_ena = 1
i_ram_ena = 1
i_ram_wena = 0
R[rt] R[rs]+unsign_ext(imm16): rf_raddr1 = IM[25:21]
rf_raddr1 = xxxxx (IM[25:21])
rf_we = 1
rf_wdata_mux = 0
rf_waddr = IM[20:16] //表示回写目的寄存器为 rt
s_ext_s = 0
alu_aluc = 0001
alu_a_mux = 0
alu_b_mux = 01
PC PC+4: pc_mux = 00
LW 所用部件: PC、IM、ALU、RF、S_EXT、DM、rf_wdata_mux 、alu_a_mux、
alu_b_mux、
pc_mux
取指令: pc_ena = 1
i_ram_ena = 1
i_ram_wena = 0
R[rt] MEM[R[rs]+sign_ext(imm16)]: rf_raddr1 = IM[25:21]
rf_raddr2 = xxxxx (IM[20:16])
rf_we = 1
rf_wdata_mux = 1
rf_waddr = IM[20:16]
s_ext_s = 1
alu_aluc = 0000

```

```

alu_a_mux = 0
alu_b_mux = 01
d_ram_ena = 1
d_ram_wena = 0
PC PC+4: pc_mux = 00
SW 所用部件: PC、IM、ALU、RF、S_EXT、DM、rf_wdata_mux、alu_a_mux、
alu_b_mux、
pc_mux
取指令: pc_ena = 1
i_ram_ena = 1
i_ram_wena = 0
R[rt] MEM[R[rs]+sign_ext(imm16)]: rf_raddr1 = IM[25:21]
rf_raddr2 = xxxxx (IM[20:16])
rf_we = 0
rf_wdata_mux = x
rf_waddr = xxxxx //无回写, 未定义
s_ext_s = 1
alu_aluc = 0000
alu_a_mux = 0
alu_b_mux = 01
d_ram_ena = 1
d_ram_wena = 1
PC PC+4: pc_mux = 00
BEQ 所用部件: PC、IM、ALU、S_EXT、rf_wdata_mux、alu_a_mux、alu_b_mux、
pc_mux
取指令: pc_ena = 1
i_ram_ena = 1
i_ram_wena = 0
target_offset = sign_extend(offset || 0
2
): s_ext_s = 1
rf_raddr1 = IM[25:21]
rf_raddr2 = IM[20:16]
rf_we = 0
rf_wdata_mux = x
rf_waddr = xxxxx
alu_aluc = 0001
alu_a_mux = 0
alu_b_mux = 00

```

if (GPR[rs] == GPR[rt]) then zero→cu

PC = PC + target_offset pc_mux = 01

else

PC = PC + 4 pc_mux = 00

J 所用部件: PC、IM、pc_mux

取指令: pc_ena = 1

i_ram_ena = 1

i_ram_ena = 0

PC = PC[31:28] || IM[25:0]<<2: pc_mux = 10

	Addu	Subu	Sll	Ori	Lw	Sw	Beq	J
pcreg_ena	1	1	1	1	1	1	1	1
i_ram_ena	1	1	1	1	1	1	1	1
i_ram_wena	0	0	0	0	0	0	0	0
d_ram_wena	0	0	0	0	0	1	0	0
d_ram_ena	0	0	0	0	1	1	0	0
rf_we	1	1	1	1	1	0	0	0
rf_waddr	IM[25:21]	IM[25:21]	xxxxx	IM[25:21]	IM[25:21]	IM[25:21]	IM[25:21]	xxxxxx
rf_raddr1	IM[20:16]	IM[20:16]	IM[20:16]	xxxxx	xxxxx	xxxxx	IM[20:16]	xxxxxx
rf_raddr2	IM[15:11]	IM[15:11]	IM[15:11]	IM[20:16]	IM[20:16]	xxxxx	xxxxx	xxxxxx
s_ext_s	0	0	0	0	1	1	1	0
alu_aluc[3:0]	0000	0001	1110	0101	0000	0000	0011	0000
alu_a_mux	0	0	1	0	0	0	0	0
alu_b_mux[1:0]	00	00	00	01	01	01	00	00
rf_wdata_mux	0	0	0	0	1	x	x	0
pc_mux[1:0]	00	00	00	00	00	00	01	10

最后，列出布尔方程：

```

pc_ena = ADDU || SUBU || SLL || ORI || LW || SW || BEQ || J
i_ram_ena = ADDU || SUBU || SLL || ORI || LW || SW || BEQ || J
i_ram_wena = 0
d_ram_wena = SW
d_ram_ena = LW || SW
rf_we = ADDU || SUBU || SLL || ORI || LW
s_ext_s = LW || SW || BEQ
alu_aluc [0] = SUBU || ORI || BEQ
alu_aluc [1] = SLL || BEQ
alu_aluc [2] = SLL || ORI
alu_aluc [3] = SLL
alu_a_mux = SLL
alu_b_mux[0] = ORI || LW || SW
alu_b_mux[1] = 0
rf_wdata_mux = LW
pc_mux[0] = BEQ
pc_mux[1] = J
rf_raddr1 = IM[25:21] && (ADDU || SUBU || ORI || LW || SW ||
BEQ ) //此处为了表述简洁, 没有将 IM[25: 21]的每一位像 aluc 那样展开书
写, rf_raddr1 的信号也可以通过直接连线实现
rf_raddr2 = IM[20:16] && (ADDU || SUBU || SLL || BEQ ) //同上
rf_waddr = (IM[15:11] && (ADDU || SUBU || SLL )) || IM[20:16]
&&( ORI || LW))//同上, ps: rf_waddr 信号也可以使用选择器来实现

```

4) 关于 CPU 测试

a) 前仿真单条指令的测试

在 modelsim 中, 可以使用如下代码初始化 iram:

```

initial begin
$readmemh("lout.txt", ram);
end

```

但 initial 块不可综合, 若下板时需要初始化 iram 请参看后面介绍的 ip 核使用示例。

测试指令时可在 cpu 的 testbench 中添加如图 8.4.11 代码打印结果, 将结果打印到文件中 (可以自行查找 verilog 的系统函数的使用方法)

```

integer file_output;
integer counter = 0;

initial begin
    //$dumpfile("mydump.txt");
    //$dumpvars(0,cpu_tb.uut.pcreg.data_out);
    file_output = $fopen("result.txt");
    // Initialize Inputs
    clk = 0;
    rst = 1;

    // Wait 100 ns for global reset to finish
    #50;
    rst = 0;
    // Add stimulus here
end

```

```

2      always @ (posedge clk)
3      begin
4          counter = counter + 1;
5
6          $fdisplay(file_output, "regfiles0 = %h", cputb.uut.rf.regfiles[0]);
7          $fdisplay(file_output, "regfiles1 = %h", cputb.uut.rf.regfiles[1]);
8          $fdisplay(file_output, "regfiles2 = %h", cputb.uut.rf.regfiles[2]);
9          $fdisplay(file_output, "regfiles3 = %h", cputb.uut.rf.regfiles[3]);
10         $fdisplay(file_output, "regfiles4 = %h", cputb.uut.rf.regfiles[4]);
11         $fdisplay(file_output, "regfiles5 = %h", cputb.uut.rf.regfiles[5]);
12         $fdisplay(file_output, "regfiles6 = %h", cputb.uut.rf.regfiles[6]);
13         $fdisplay(file_output, "regfiles7 = %h", cputb.uut.rf.regfiles[7]);
14         $fdisplay(file_output, "regfiles8 = %h", cputb.uut.rf.regfiles[8]);
15         $fdisplay(file_output, "regfiles9 = %h", cputb.uut.rf.regfiles[9]);
16         $fdisplay(file_output, "regfiles10 = %h", cputb.uut.rf.regfiles[10]);
17         $fdisplay(file_output, "regfiles11 = %h", cputb.uut.rf.regfiles[11]);
18         $fdisplay(file_output, "regfiles12 = %h", cputb.uut.rf.regfiles[12]);
19         $fdisplay(file_output, "regfiles13 = %h", cputb.uut.rf.regfiles[13]);
20         $fdisplay(file_output, "regfiles14 = %h", cputb.uut.rf.regfiles[14]);
21         $fdisplay(file_output, "regfiles15 = %h", cputb.uut.rf.regfiles[15]);
22         $fdisplay(file_output, "regfiles16 = %h", cputb.uut.rf.regfiles[16]);
23         $fdisplay(file_output, "regfiles17 = %h", cputb.uut.rf.regfiles[17]);
24         $fdisplay(file_output, "regfiles18 = %h", cputb.uut.rf.regfiles[18]);
25         $fdisplay(file_output, "regfiles19 = %h", cputb.uut.rf.regfiles[19]);
26         $fdisplay(file_output, "regfiles20 = %h", cputb.uut.rf.regfiles[20]);
27         $fdisplay(file_output, "regfiles21 = %h", cputb.uut.rf.regfiles[21]);
28         $fdisplay(file_output, "regfiles22 = %h", cputb.uut.rf.regfiles[22]);
29         $fdisplay(file_output, "regfiles23 = %h", cputb.uut.rf.regfiles[23]);
30         $fdisplay(file_output, "regfiles24 = %h", cputb.uut.rf.regfiles[24]);
31         $fdisplay(file_output, "regfiles25 = %h", cputb.uut.rf.regfiles[25]);
32         $fdisplay(file_output, "regfiles26 = %h", cputb.uut.rf.regfiles[26]);
33         $fdisplay(file_output, "regfiles27 = %h", cputb.uut.rf.regfiles[27]);
34         $fdisplay(file_output, "regfiles28 = %h", cputb.uut.rf.regfiles[28]);
35         $fdisplay(file_output, "regfiles29 = %h", cputb.uut.rf.regfiles[29]);
36         $fdisplay(file_output, "regfiles30 = %h", cputb.uut.rf.regfiles[30]);
37         $fdisplay(file_output, "regfiles31 = %h", cputb.uut.rf.regfiles[31]);
38
39         $fdisplay(file_output, "instr = %h", cputb.uut.iram.douta);
40         $fdisplay(file_output, "pc = %h", cputb.uut.pcreg.data_out);
41     end

```

图 8.4.11 testbench 示例代码

以下举一条指令的测试进行说明：

比如 addi 指令，测试指令如下：

b) 指令的边界数据测试

我们 CPU 测试时需要对各条指令进行边界数据进行测试以保证设计的 CPU 的正确性。

还是以 addi 指令为例，可以用以下汇编指令进行测试：

```
addi $1,$0,0x0001
addi $2,$0,0x8000
addi $3,$0,0x7fff
addi $4,$1,0xffff
addi $5,$2,0xffff
addi $6,$3,0x0006
```

```
addi $7,$0,0x000f
addi $8,$0,0x00f0
addi $9,$0,0x0f00
addi $10,$0,0xf000
addi $7,$0,0x000f
addi $8,$0,0x00f0
addi $9,$0,0x0f00
addi $10,$0,0xf000
```

```
addi $11,$0,0x5555
addi $12,$0,0xAAAA
addi $11,$11,0x5555
addi $12,$12,0xAAAA
```

```
addi $13,$0,0x000f
addi $14,$0,0x00ff
addi $15,$0,0x0fff
addi $13,$13,0x000f
addi $14,$14,0x00ff
addi $15,$15,0x0fff
```

```
addi $16,$0,0x0010
addi $17,$0,0x0011
addi $18,$0,0x0012
addi $19,$0,0x0013
addi $20,$0,0x0014
addi $21,$0,0x0015
addi $22,$0,0x0016
```

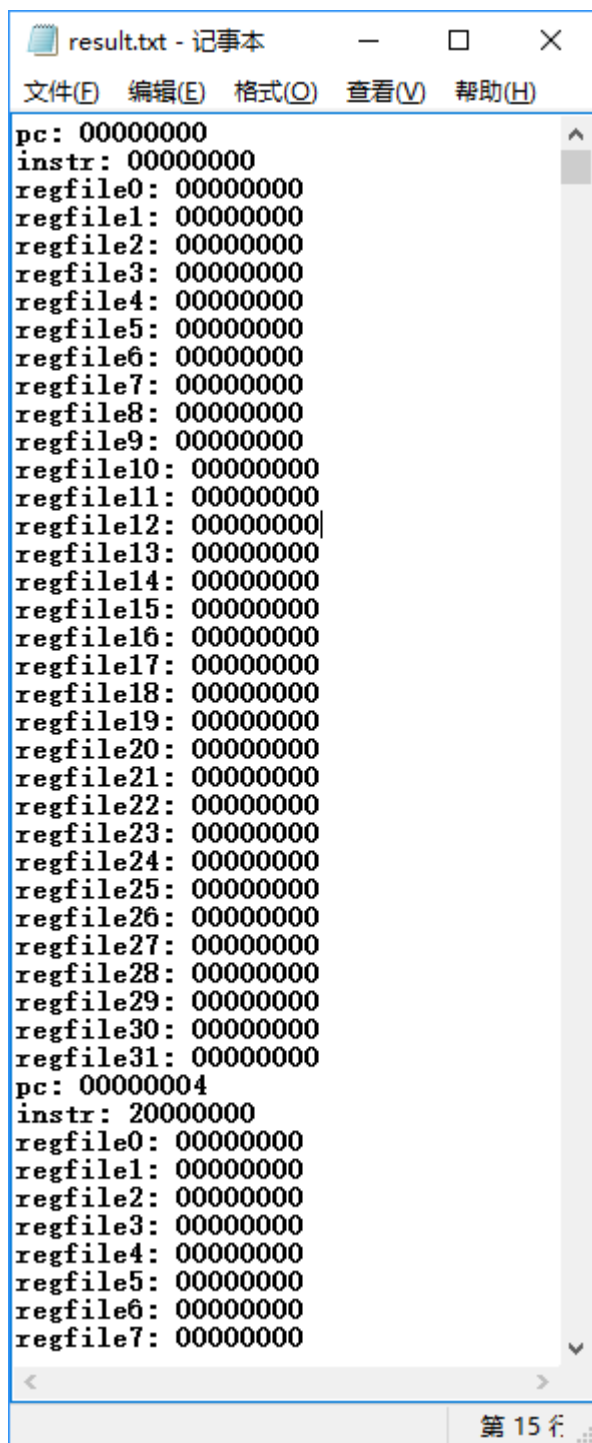
```
addi $23,$0,0x0017
addi $24,$0,0x0018
addi $25,$0,0x0019
addi $26,$0,0x001A
addi $27,$0,0x001B
addi $28,$0,0x001C
addi $29,$0,0x001D
addi $30,$0,0x001E
addi $31,$0,0x001F
```

```
addi $16,$16,0x0001
addi $17,$17,0xf001
addi $18,$18,0x8001
addi $19,$19,0x7001
addi $20,$20,0x0001
addi $21,$21,0x0211
addi $22,$22,0xab1
addi $23,$23,0xdc01
addi $24,$24,0xa001
addi $25,$25,0xb001
addi $26,$26,0xe001
addi $27,$27,0x7d01
addi $28,$28,0xcba1
addi $29,$29,0xdc1
addi $30,$30,0x7871
addi $31,$31,0x7771
```

d) 随机指令测试

可以自行编写一些符合 MIPS 规范的指令序列。将这序列分别放到 CPU 仿真状态下执行和 MARS 上去执行，分别产生两个执行结果文件，比较执行结果文件来判断 CPU 执行指令是否对。

用 Mars 运行 MIPS 汇编程序后会在 Mars 目录下生成一个 result.txt 文件，内容如图 8.4.12，为运行每条指令后的 指令地址：pc、指令十六进制码：instr、寄存器堆十六进制码：regfile；



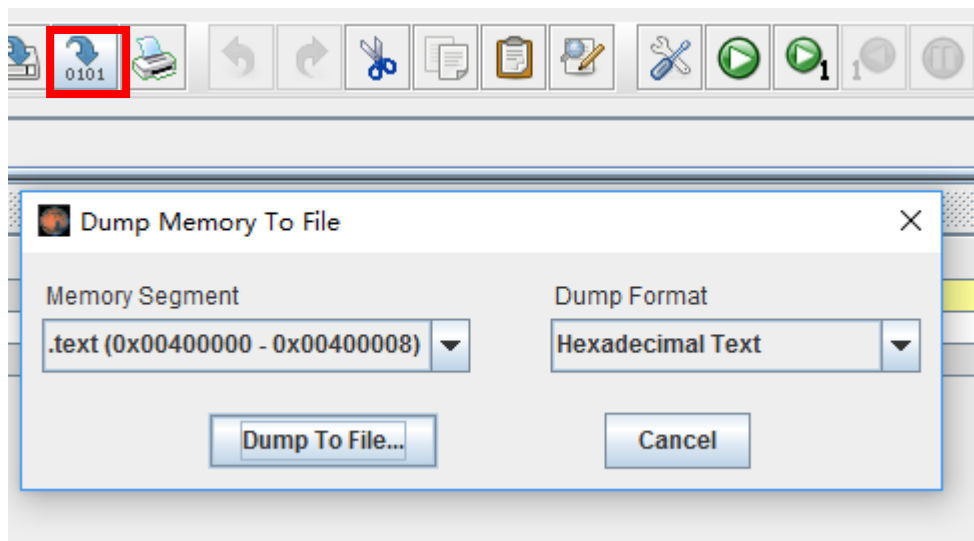
```
pc: 00000000
instr: 00000000
regfile0: 00000000
regfile1: 00000000
regfile2: 00000000
regfile3: 00000000
regfile4: 00000000
regfile5: 00000000
regfile6: 00000000
regfile7: 00000000
regfile8: 00000000
regfile9: 00000000
regfile10: 00000000
regfile11: 00000000
regfile12: 00000000
regfile13: 00000000
regfile14: 00000000
regfile15: 00000000
regfile16: 00000000
regfile17: 00000000
regfile18: 00000000
regfile19: 00000000
regfile20: 00000000
regfile21: 00000000
regfile22: 00000000
regfile23: 00000000
regfile24: 00000000
regfile25: 00000000
regfile26: 00000000
regfile27: 00000000
regfile28: 00000000
regfile29: 00000000
regfile30: 00000000
regfile31: 00000000
pc: 00000004
instr: 20000000
regfile0: 00000000
regfile1: 00000000
regfile2: 00000000
regfile3: 00000000
regfile4: 00000000
regfile5: 00000000
regfile6: 00000000
regfile7: 00000000
```

第 15 页

图 8.4.12 寄存器状态

用 Mars 导出编译后的 MIPS 指令十六进制格式文件，如图 8.4.13，点击红框处，导出格式选择十六进制。

图 8.4.13 导出文件



用自己的 CPU 运行导出的 MIPS 汇编程序，按 `cpu_tb.v` 中，如图 8.4.14，将 `pc`、`instr`、`regfile` 结果输出到文件，在 CPU 工程目录下生成另一个 `result.txt` 文件；

```
integer file_output;
integer counter = 0;

initial begin
    //$dumpfile("mydump.txt");
    //$dumpvars(0,cpu_tb.uut.pcreg.data_out);
    file_output = $fopen("result.txt");
    // Initialize Inputs
    clk = 0;
    rst = 1;

    // Wait 100 ns for global reset to finish
    #50;
    rst = 0;
    // Add stimulus here

    //#100;
    //$fclose(file_output);
end

always begin
    #50;
    clk = ~clk;
    if(clk == 1'b0) begin
        if(counter == 400) begin
            $fclose(file_output);
        end
        else begin
            counter = counter + 1;
            $fdisplay(file_output, "pc: %h", test.uut.pcreg.data_out);
            $fdisplay(file_output, "instr: %h", test.uut.pipe_if.ram_outdata);
            $fdisplay(file_output, "regfile0: %h", test.uut.pipe_id.rf.reg0.data_out);
            $fdisplay(file_output, "regfile1: %h", test.uut.pipe_id.rf.reg1.data_out);
            $fdisplay(file_output, "regfile2: %h", test.uut.pipe_id.rf.reg2.data_out);
        end
    end
end
```

图 8.4.14 结果文件

比对 1 和 2 中生成的两个结果文件，比对可使用 UltraCompare 或 Notepad++等工

具。

如图 8.4.15，对比发现在 pc 为 8，执行指令 3c038000 后 \$1 寄存器的结果出现了错误，需要进一步调试修改。

例子中运行的指令为：

```
sll $0,$0,0
addi $1,$0,0xf
addi $0,$1,0x1
```

结果发现第三条指令出现错误，\$0 寄存器结果异常，因为零号寄存器恒置零，所以应修改寄存器堆中对零号寄存器的写操作。

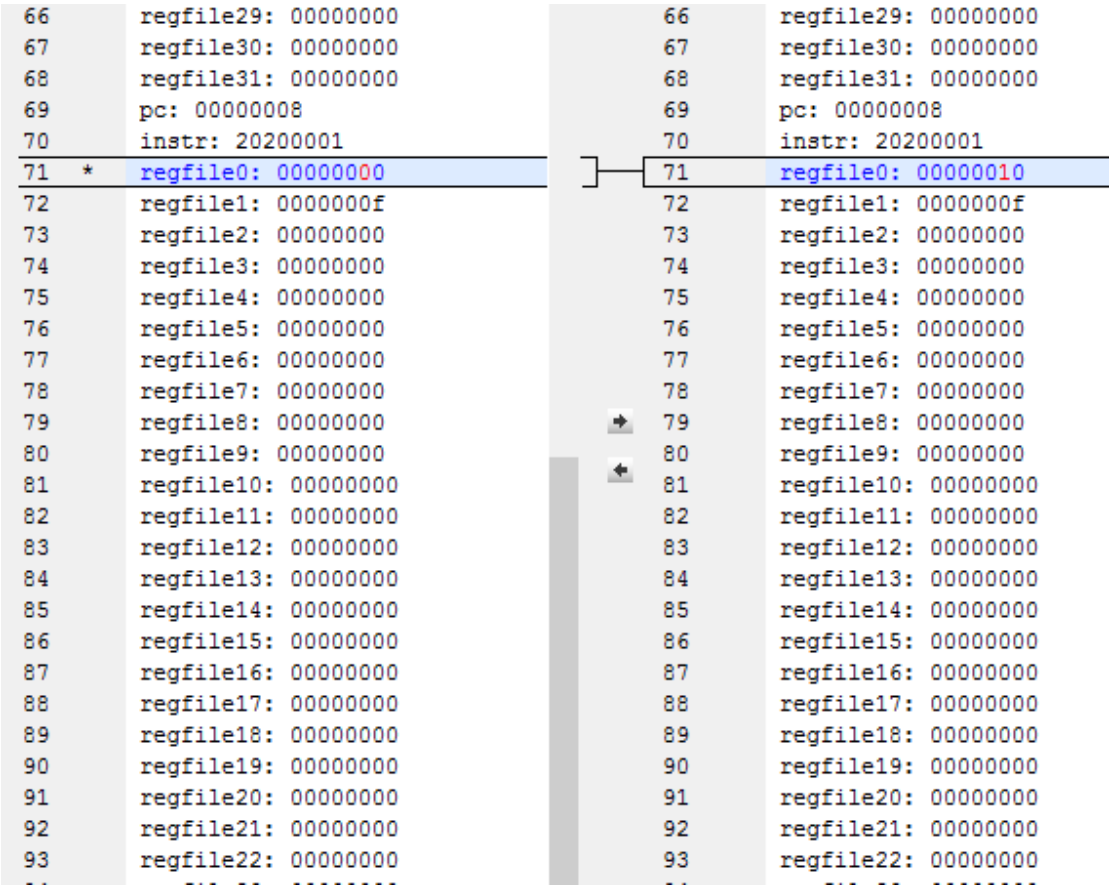


图 8.4.15 对比情况

e) 后仿真测试

Modelsim 的仿真分为前仿真和后仿真，下面具体介绍一下两者的区别。

- 前仿真
前仿真也称为功能仿真，主旨在于验证电路的功能是否符合设计要求，其特点是不考虑电路门延迟与线延迟，主要是验证电路与理想情况是否一致。可综合 FPGA 代码是用 RTL 级代码语言描述的，其输入为 RTL 级代码与 Testbench。
- 后仿真
后仿真也称为时序仿真或者布局布线后仿真，是指电路已经映射到特定的工艺环境以后，

综合考虑电路的路径延迟与门延迟的影响,验证电路能否在一定时序条件下满足设计构想的过程, 是否存在时序违规。其输入文件为从布局布线结果中抽象出来的门级网表、Testbench 和扩展名为 SDO 或 SDF 的标准时延文件。SDO 或 SDF 的标准时延文件不仅包含门延迟,还包括实际布线延迟,能较好地反映芯片的实际工作情况。一般来说后仿真是必选的,检查设计时序与实际的 FPGA 运行情况是否一致,确保设计的可靠性和稳定性。选定了器件分配引脚后在做后仿真。

f) 下板测试

由于我们自行编写的指令 RAM 用来初始化内存的 initial 指令是不可综合的,无法在开发板上运行,所以,我们可以使用 Vivado 提供的 ip 核来替换我们的 ram,其可以使用一个 coe 文件来初始化内存。

Coe 为初始化 ROM 的配置文件,以下为 coe 文件格式实例

memory_initialization_radix=16; #16 表示指令以 16 进制表示,可以按需求更改

memory_initialization_vector= #由此开始为指令序列,以“,”隔开“;”结束

00000000,

241d03fc,

0800001a;

跟随以下步骤来添加 IP 核:

如图选择 IP 核列表

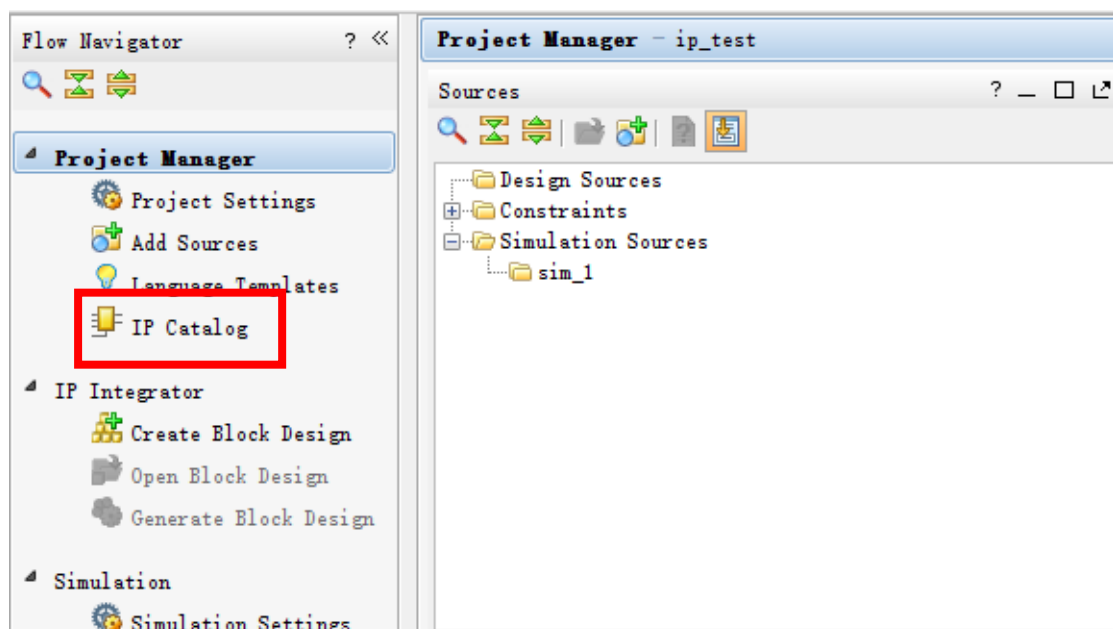


图 8.4.12 IP 核列表选择

在列表中选择 Memories & Storage Elements 中的 RAMs & ROMs & BRAM, 双

击 Block Memory Generator

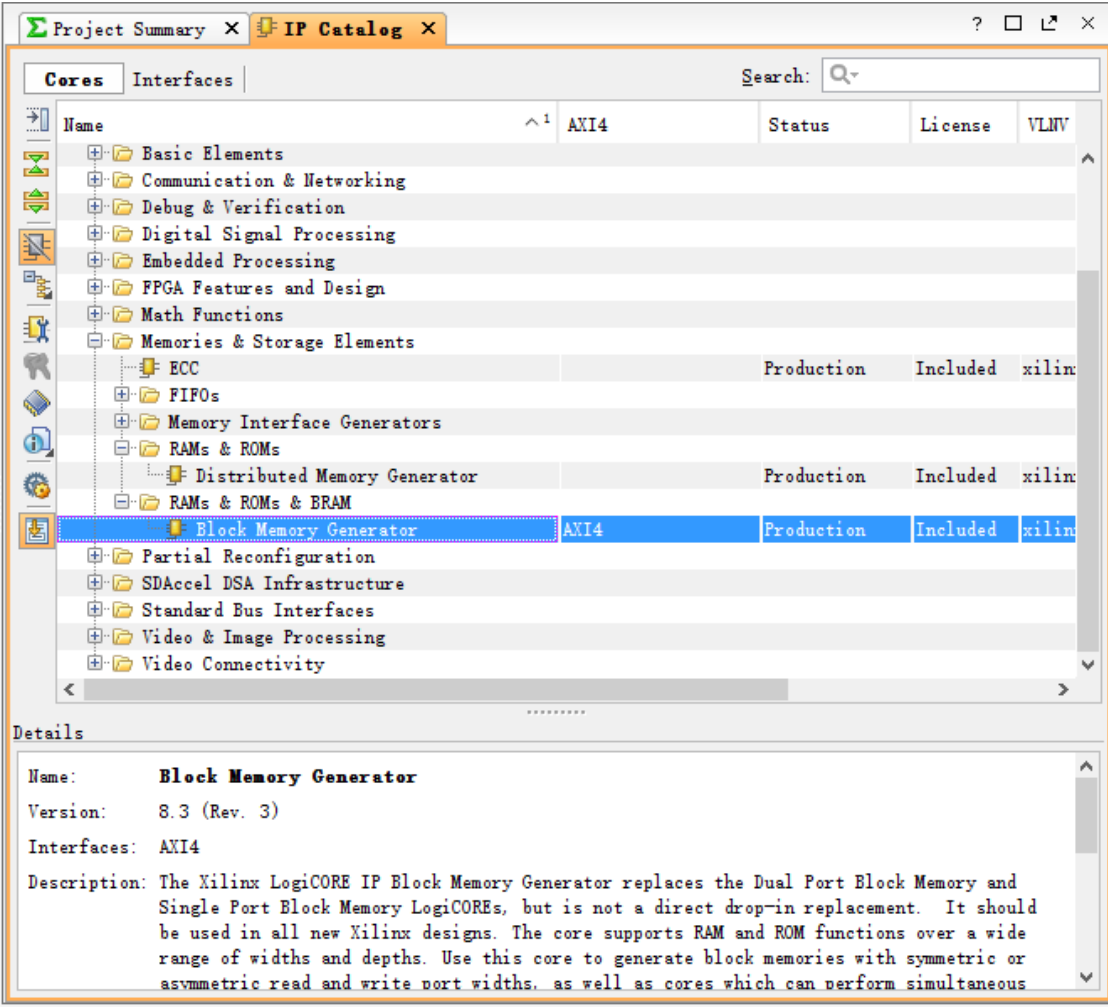


图 8.4.13 列表选择

根据图 8.4.14 配置 ip 核，下图均为推荐配置，可根据自己的需求来更改

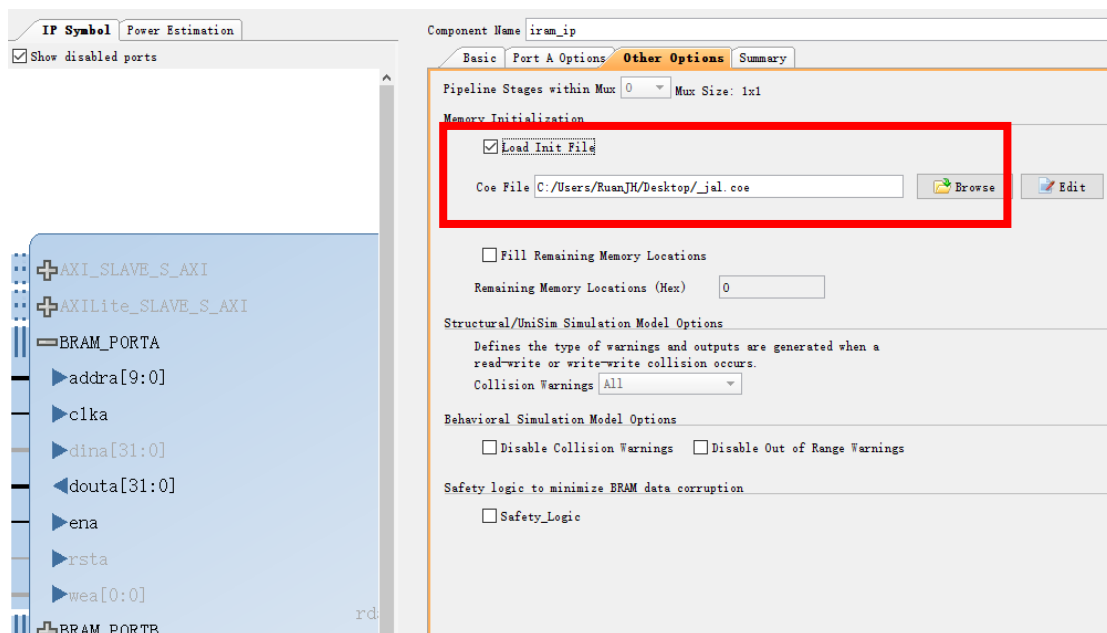
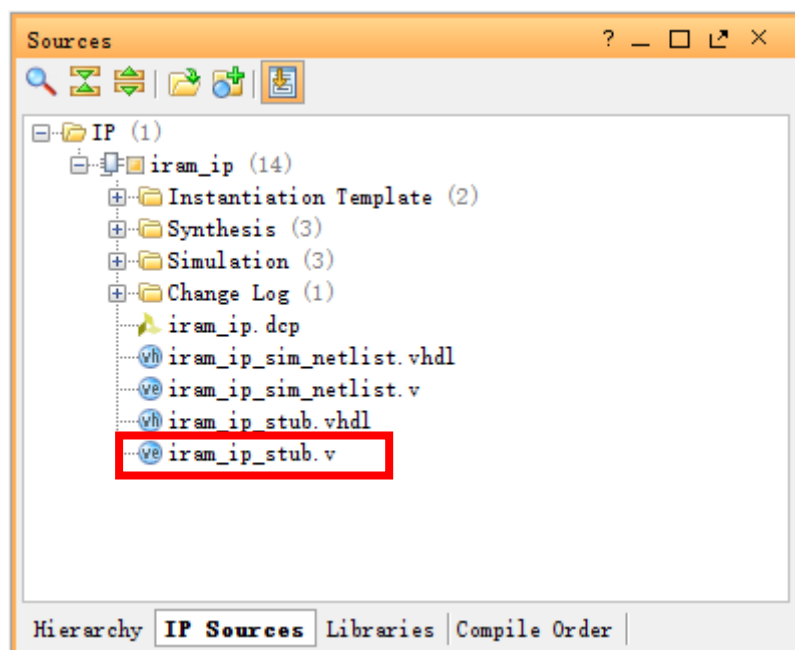


图 8.4.15 IP 核推荐配置

点击生成的 IP Sources，可以看到如图 8.4.16 选项，可以根据 ip 核的接口来使用生成的模块（双击 ip 核模块，可以对 ip 核进行修改）



```

15 // Please paste the declaration into a Verilog source file or add the file as an a:
16 (* x_core_info = "blk_mem_gen_v8_3_3,Vivado 2016.2" *)
17 module iram_ip(clka, ena, addra, douta)
18 /* synthesis syn_black_box black_box_pad_pin="clka, ena, addra[9:0], douta[31:0]" */;
19     input clka;
20     input ena;
21     input [9:0]addra;
22     output [31:0]douta;
23 endmodule
24

```

图 8.4.16 IP 核模块选择

在 CPU 调用 IP 核时可参照如图 8.4.17 方式

```

31
32 cpu sccpu(clk, reset, inst, c
33 iram_ip iram(.clk(clk),
34             .ena(ena),
35             .addr(addr),
36             .dout(dout)
37             );
38 dmem scdmem(~clk, reset, DM_
39
40 endmodule

```

图 8.4.17 IP 核调用方式

4. 实验步骤

1. 新建 Vivado 工程，编写各个模块。
2. 用 ModelSim 前仿真逐条测试所有指令。
3. 用 ModelSim 进行后仿真测试。
4. 配置 XDC 文件，综合下板，并观察实验现象。
5. 按照要求书写实验报告。