一、实验目标

使用 Verilog HDL 语言实现 31 条 MIPS 指令的 CPU 的设计和仿真，做到：
①深入了解 CPU 的原理。
②画出实现 31 条指令的 CPU 的通路图。
③学习使用 Verilog HDL 语言设计实现 31 条指令的 CPU。

所设计的 CPU 需要实现以下功能：
①能够正确执行 31 条 MIPS 指令，每次执行完一条指令后给出各寄存器的正确信息。
②具有完备性，能够正确处理边界指数据。
③对于较为复杂的随机指令序列，CPU 应该能够保证正确运行，并执行相应的功能。
④使用所给的 COE 文件初始化 IP 核的数据寄存器，所设计的 CPU 应当能够正确执行每一条指令，达到和 COE 文件所代表的程序相同的预期结果。
⑤能够用所给的 COE 文件通过网站上的测试。

二、总体设计

1. 需要实现的 31 条 MIPS 指令的种类、名称、格式及功能如下：

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

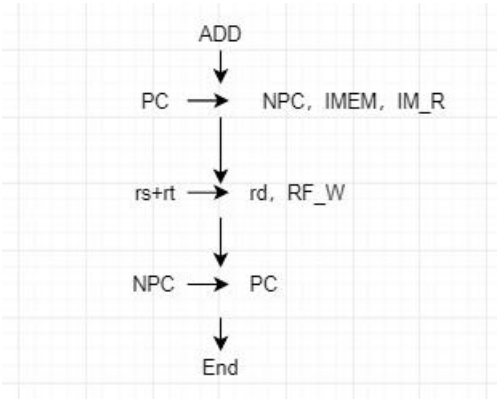
Bit #	31..26	25..21	20..16	15..0	
I-type	op	rs	rt	immediate	
addi	001000	rs	rt	Immediate(- ~ +)	addi $1,$2,100
addiu	001001	rs	rt	Immediate(- ~ +)	addiu $1,$2,100
andi	001100	rs	rt	Immediate(0 ~ +)	andi $1,$2,10
ori	001101	rs	rt	Immediate(0 ~ +)	andi $1,$2,10
xori	001110	rs	rt	Immediate(0 ~ +)	andi $1,$2,10
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. 各条指令设计：

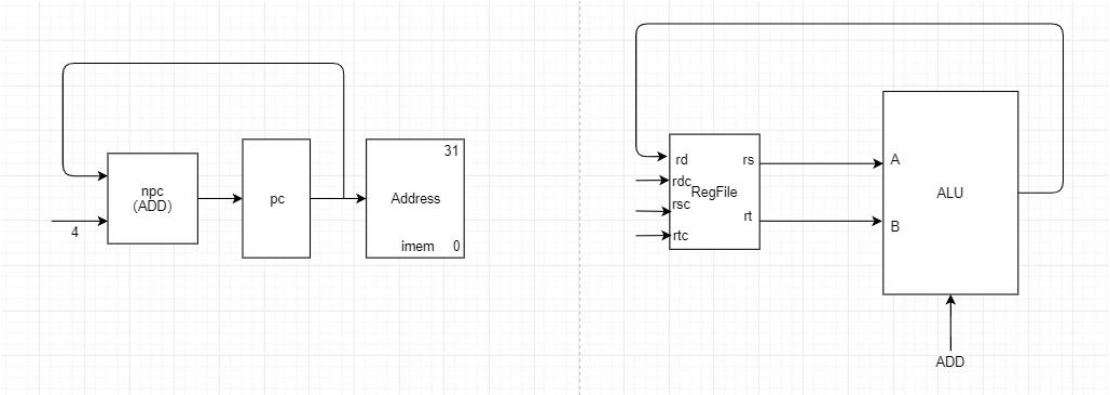
① ADD：

(1) 确定 ADD 所需的操作：取指令、R[rd]→R[rs]+R[rt] 、PC→PC+4



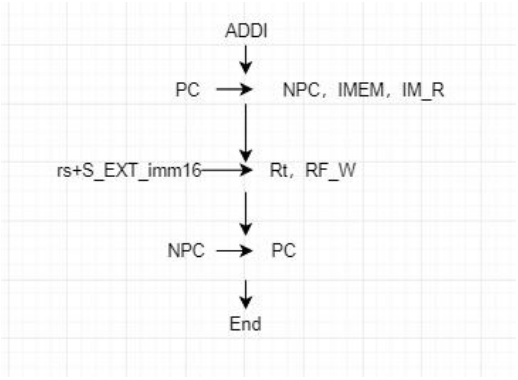
(2) 根据 ADD 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU)

(3) 根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



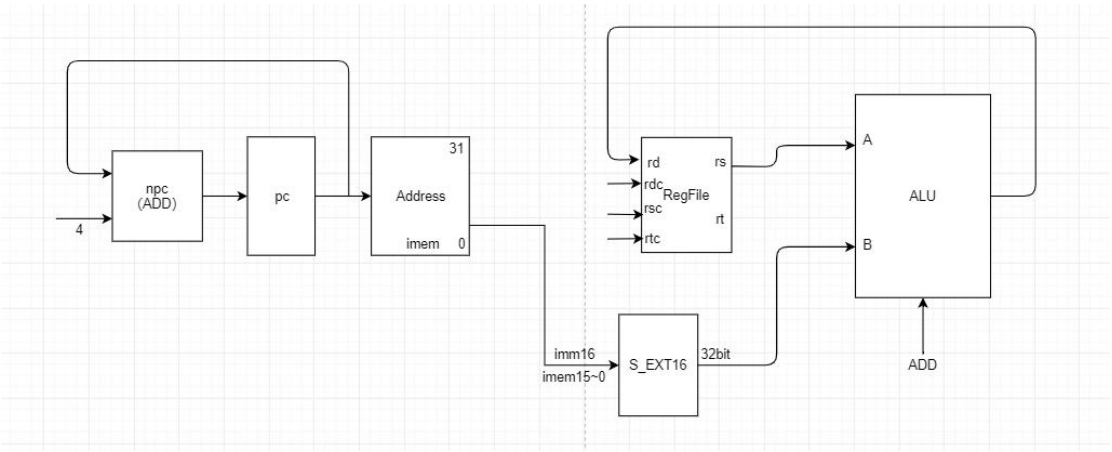
②ADDI：

(1)确定 ADDI 所需的操作：取指令、立即数符号扩展、R[rt]→R[rs]+signed_ext(imm16)、 PC→PC+4



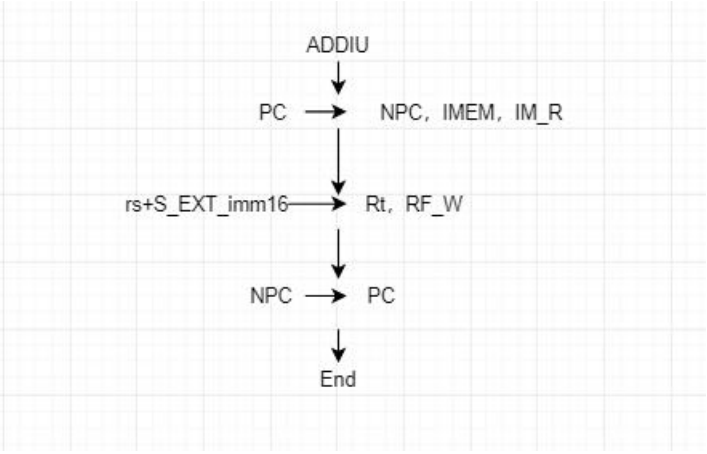
(2)根据 ADDI 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU)、有符号扩展元件(signed_ext16)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



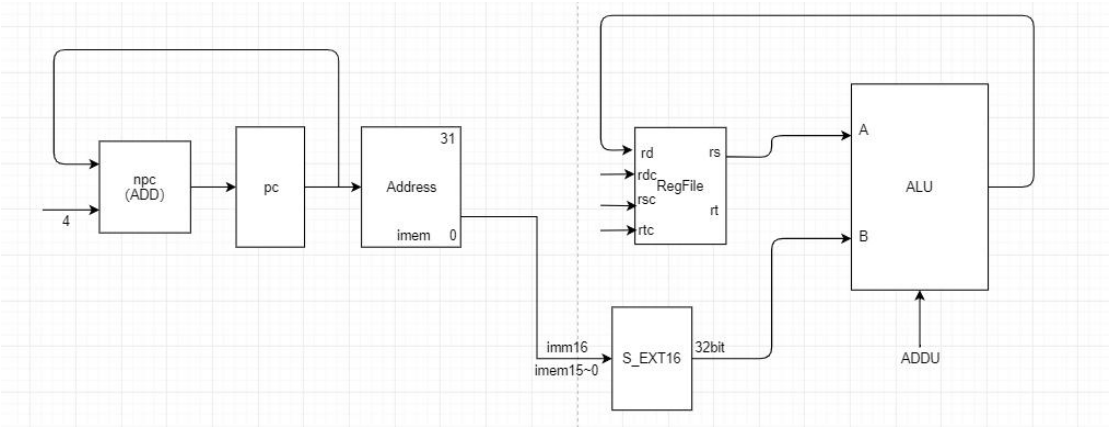
③ADDIU：

(1)确定 ADDIU 所需的操作：取指令、立即数符号扩展、R[rt]→R[rs]+signed_ext(imm16)、 PC→PC+4



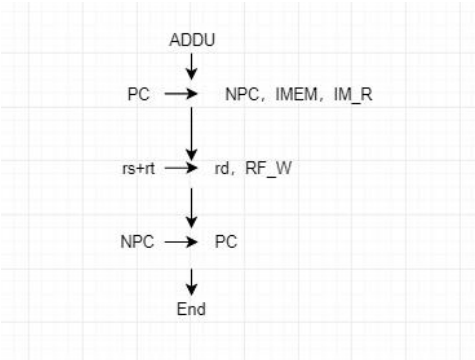
(2)根据 ADDIU 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU)、有符号扩展元件(signed_ext16)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



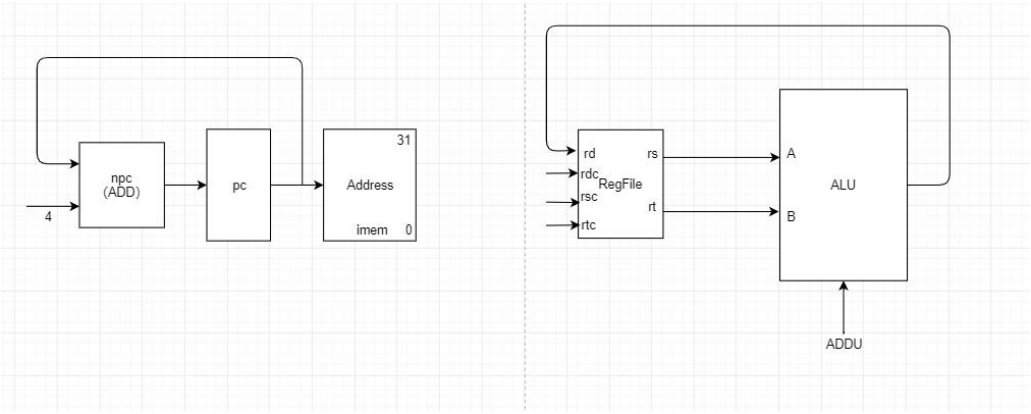
④ADDU：

(1)确定 ADDU 所需的操作：取指令、R[rd]→R[rs]+R[rt]、PC→PC+4



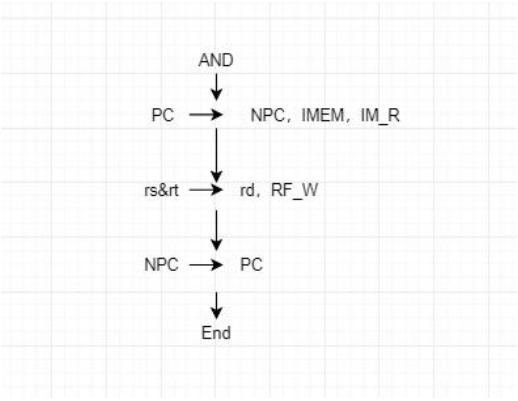
(2)根据 ADDU 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



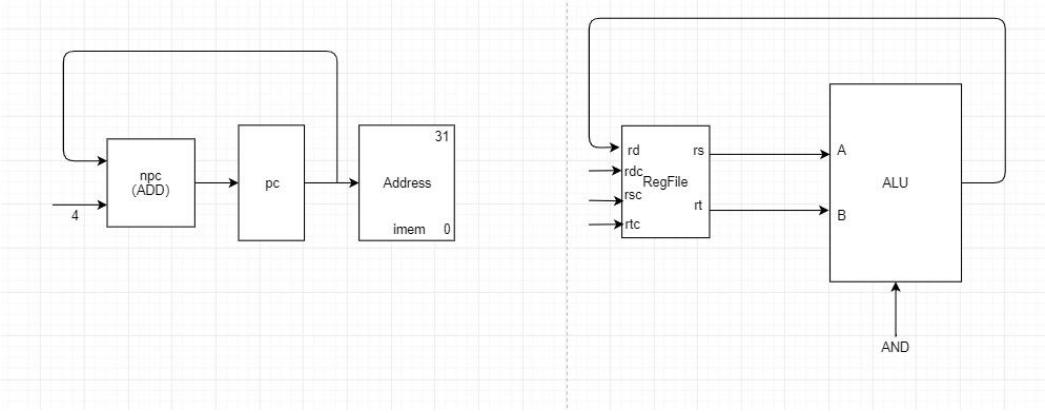
⑤AND：

(1)确定 AND 所需的操作：取指令、R[rd]→R[rs]&R[rt]、PC→PC+4



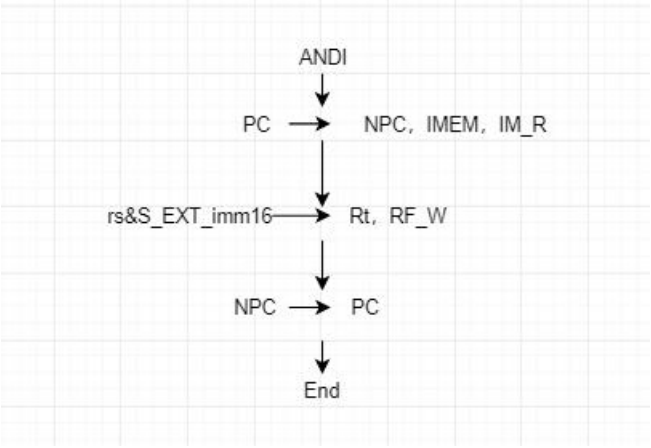
(2)根据 AND 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



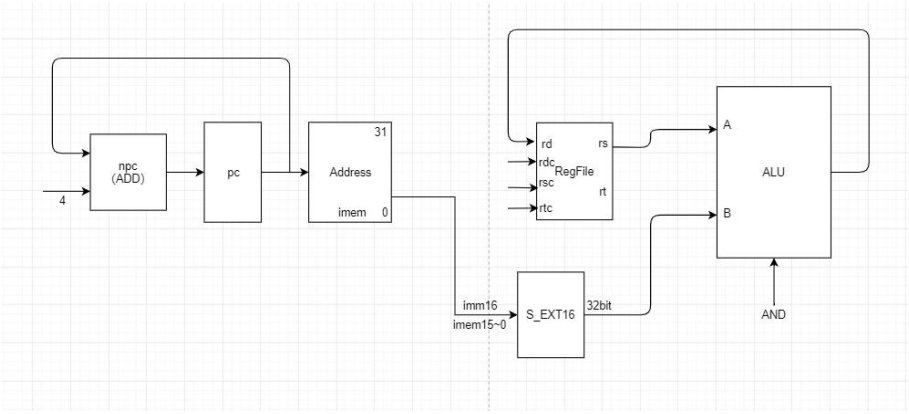
⑥ANDI：

(1)确定 ANDI 所需的操作：取指令、立即数符号扩展、R[rt]→R[rs]& unsign_ext(imm16)、 PC→PC+4



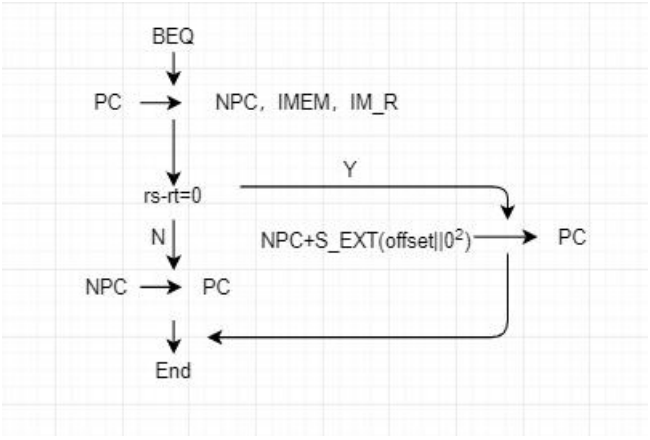
(2)根据 ANDI 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory)、寄存器堆(regfile)、算术逻辑单元(ALU) 、无符号扩展元件(ext16)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



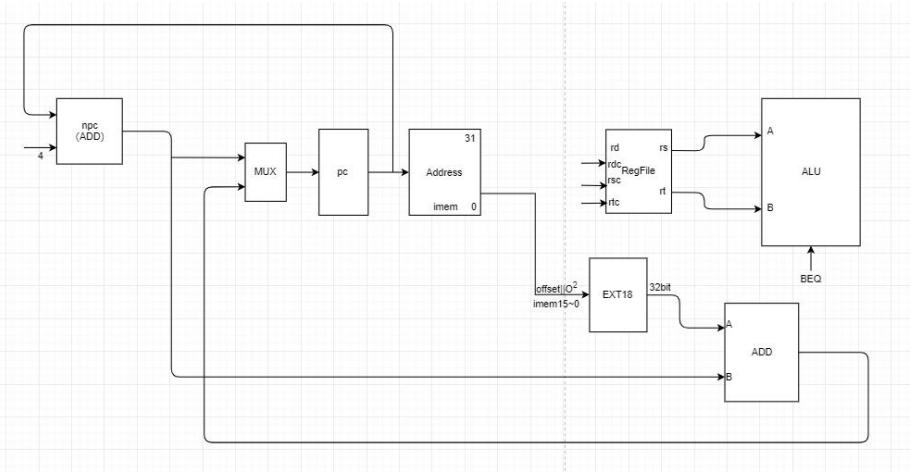
⑦BEQ：

(1)确定 BEQ 所需操作：取指令 、立即数左移两位并且符号扩展 target_offset →sign_extend(offset||02)、if （GPR[rs]==GPR[rt]） then PC→PC + target_offset else PC→PC+4



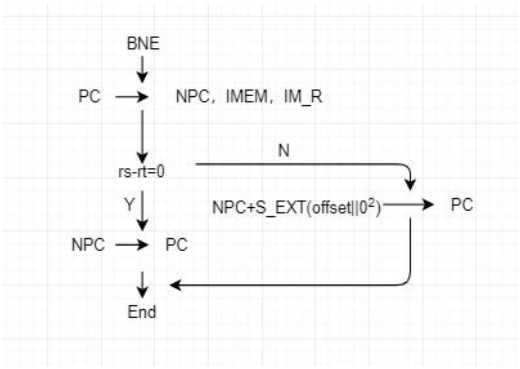
(2)根据 BEQ 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU)、有符号扩展元件(ext18)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



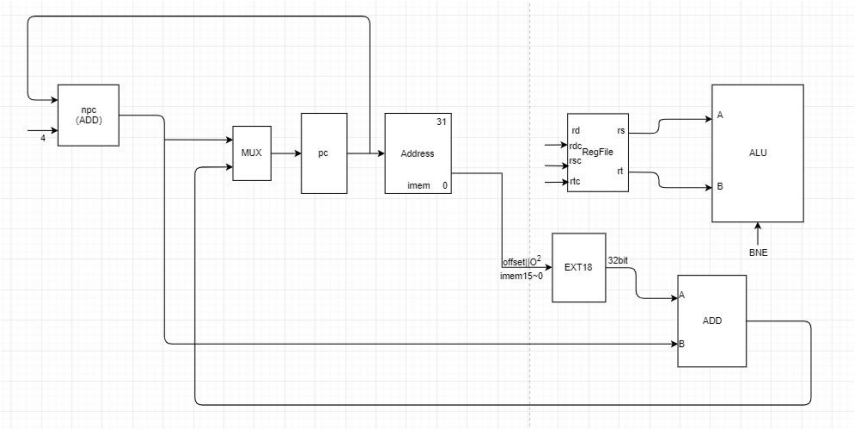
⑧BNE：

(1)确定 BNE 所需操作：取指令 、立即数左移两位并且符号扩展 target_offset →sign_extend(offset||02)、if （GPR[rs]!=GPR[rt]） then PC→PC + target_offset else PC→PC+4



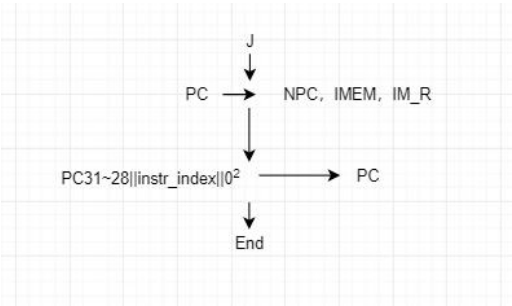
(2)根据 BNE 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU)、有符号扩展元件(ext18)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



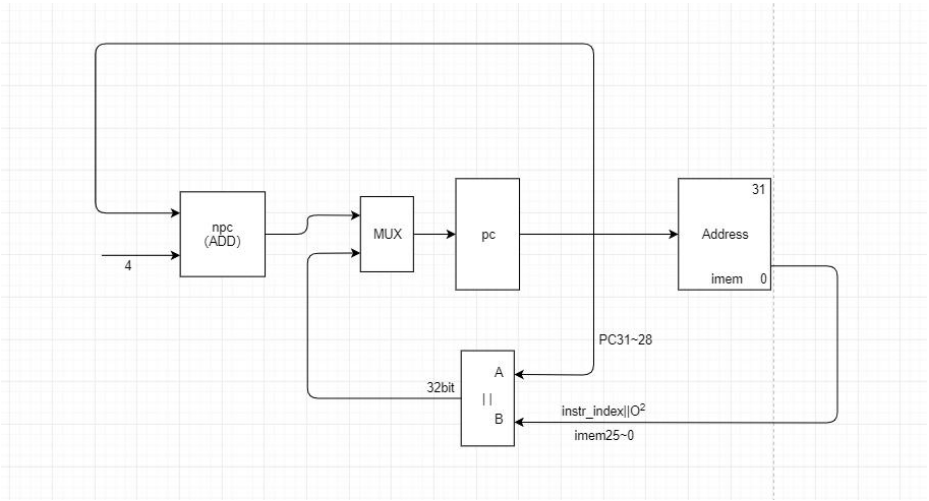
⑨J：

(1)确定 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 位地址 ，将地址送至 PC 寄存器



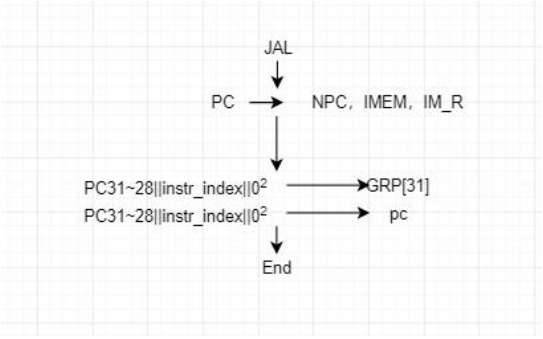
(2)根据 J 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



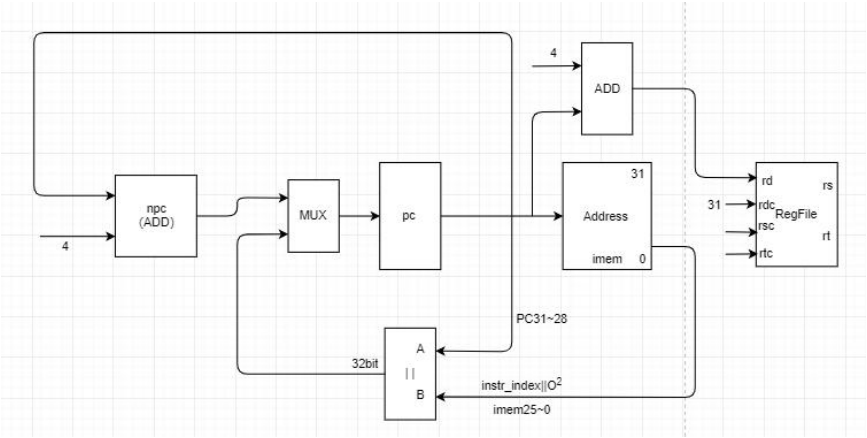
⑩JAL：

(1)确定 JAL 所需操作：取指令、指令中的 26 位地址左移两位与 PC+4 的高四位合并为一个 32 位地址，将地址送至 PC 寄存器和第 31 号寄存器



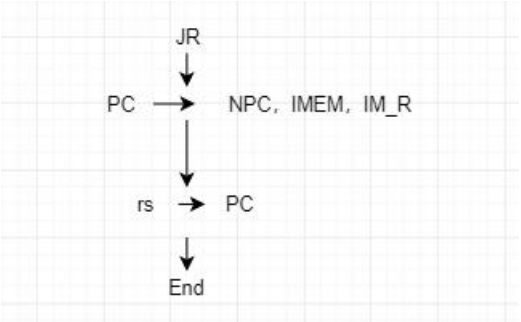
(2)根据 JAL 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆(regfile)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



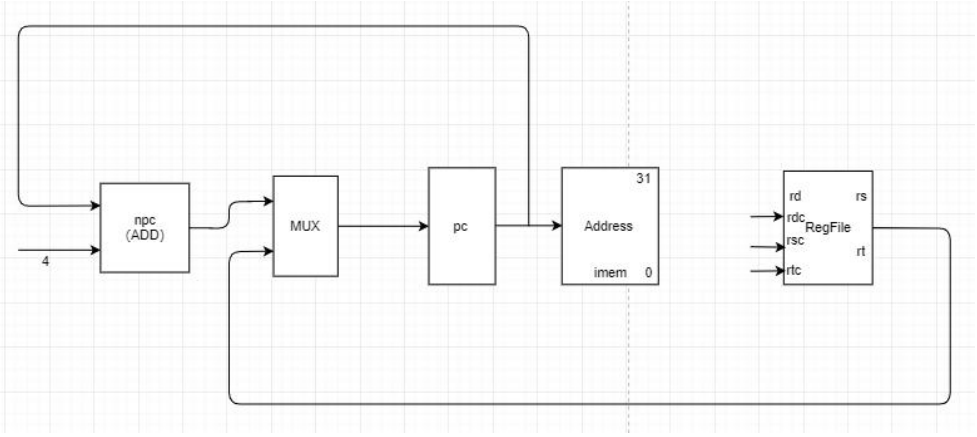
⑪JR：

(1)确定 JR 所需操作:取指令、指令中的 rs 寄存器的内容送至 PC 寄存器,PC→GRP[rs]



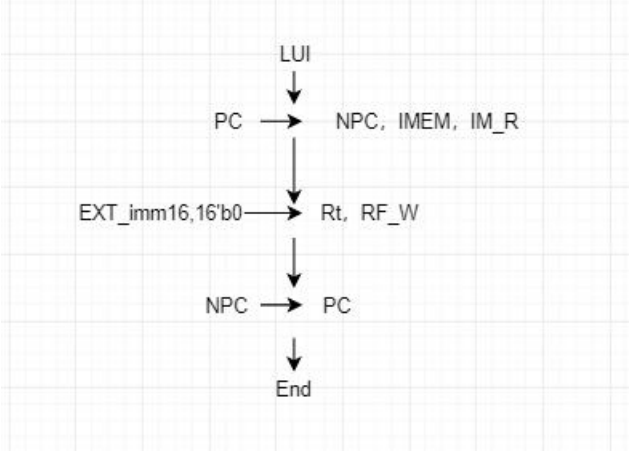
(2)根据 JR 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆(regfile)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



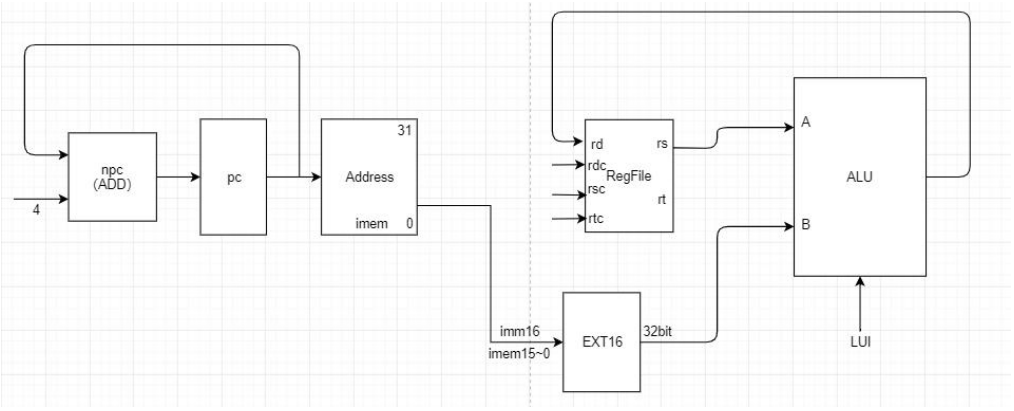
⑫LUI：

(1)确定 LUI 所需的操作：取指令、立即数符号扩展、R[rt] → unsign_ext(imm16)<<16、 PC→PC+4



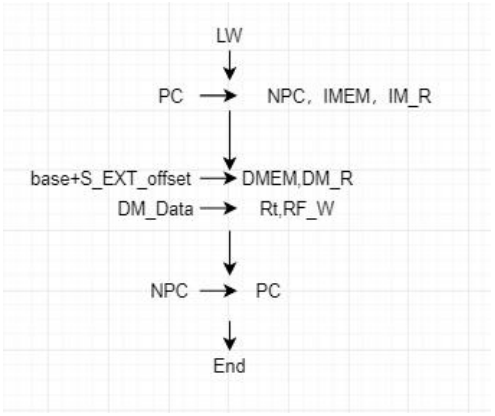
(2)根据 LUI 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory)、寄存器堆(regfile)、算术逻辑单元(ALU) 、无符号扩展元件(ext16)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



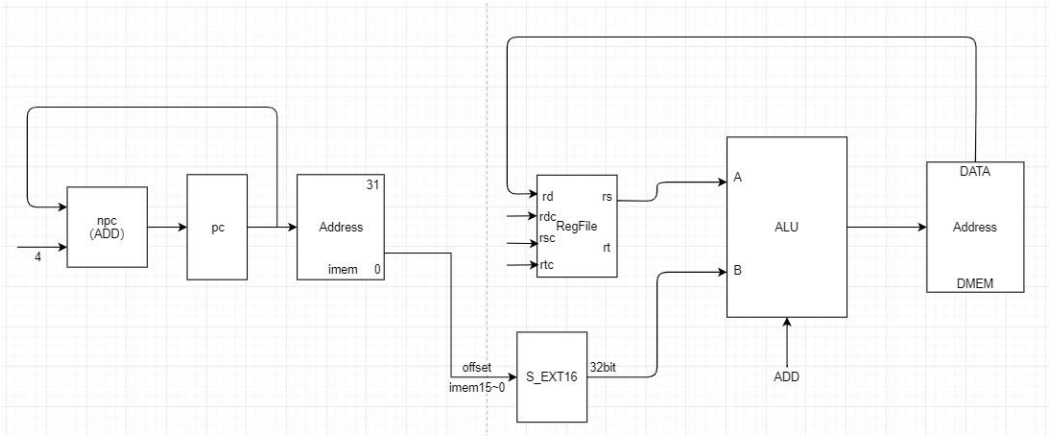
⑬LW：

(1)确定 LW 所需的操作：取指令、立即数符号扩展、R[rs]+sign_ext(imm16)、R[rt]→MEM[R[rs]+sign_ext(imm16)]、PC→PC+4



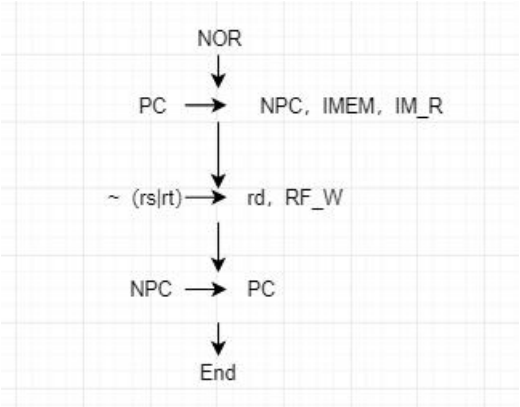
(2)根据 LW 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU) 、有符号扩展元件(ext16)、数据存储器(data memory)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



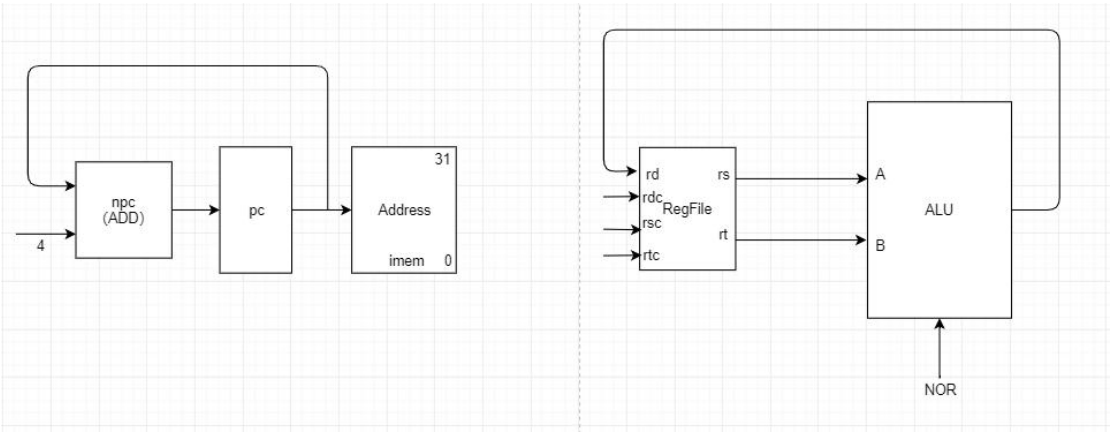
⑭NOR：

(1)确定 NOR 所需的操作：取指令、R[rd]→~(R[rs]|R[rt])、PC→PC+4



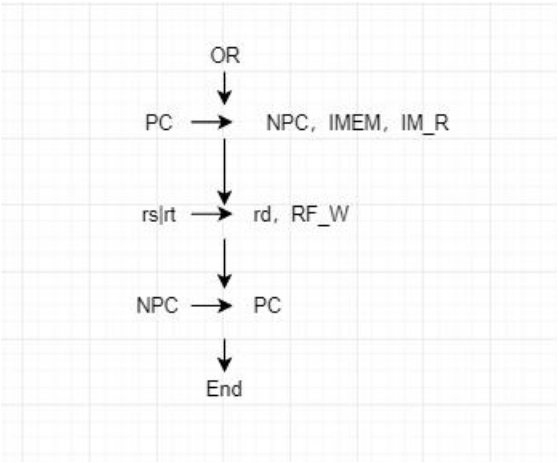
(2)根据 NOR 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



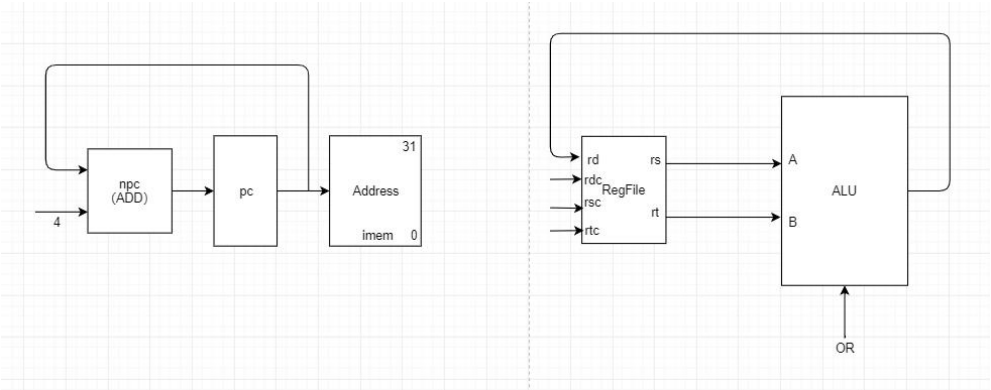
⑮OR：

(1)确定 OR 所需的操作：取指令、R[rd]→R[rs]|R[rt]、PC→PC+4



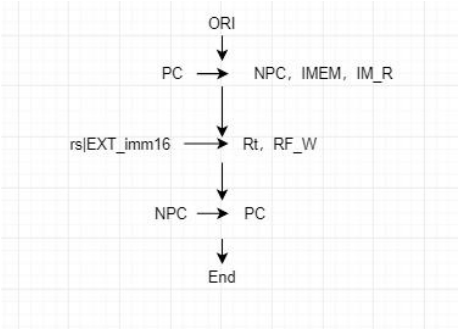
(2)根据 OR 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



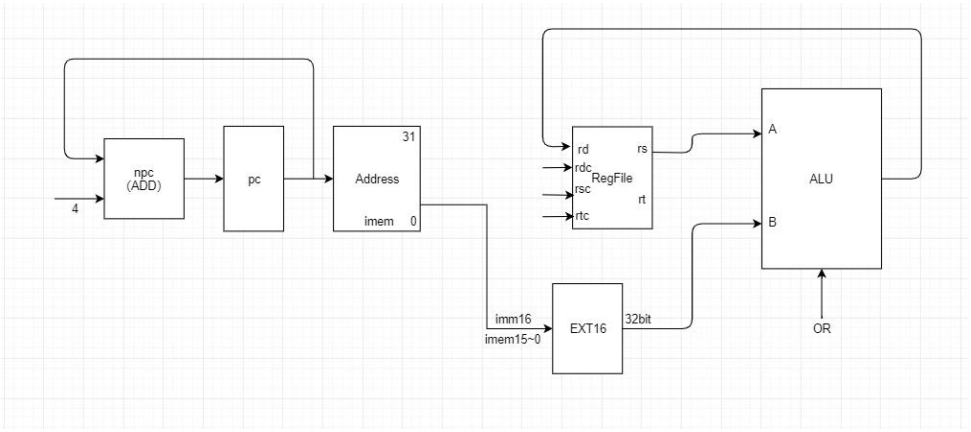
⑯ORI：

(1)确定 ORI 所需的操作：取指令、立即数符号扩展、R[rt]→R[rs]|unsign_ext(imm16)、 PC→PC+4



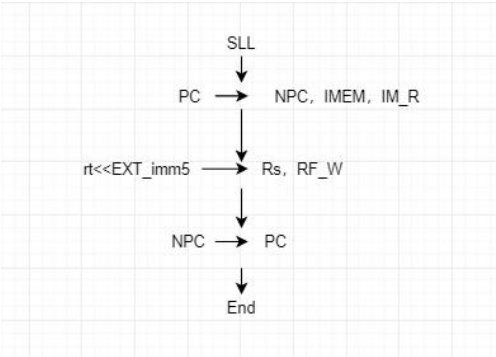
(2)根据 ORI 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU)、无符号扩展元件(ext16)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



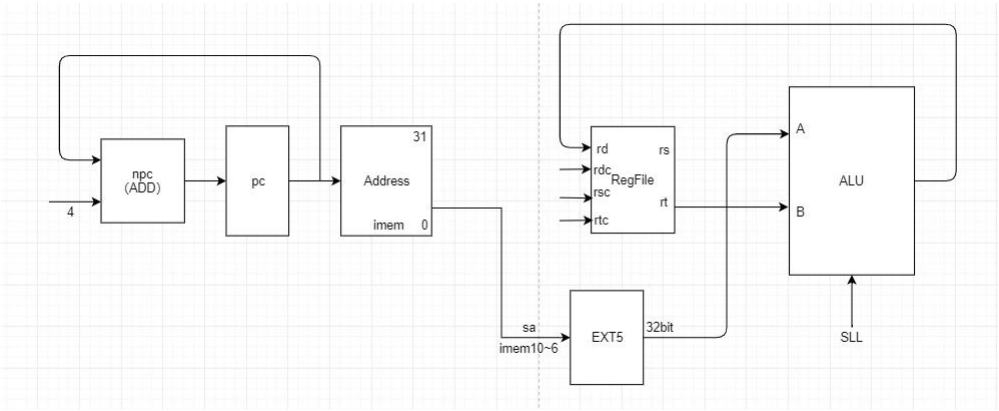
⑰SLL：

(1)确定 SLL 所需的操作：取指令、立即数符号扩展、R[rd]→R[rt]<< unsign_ext(shamt)、 PC→PC+4



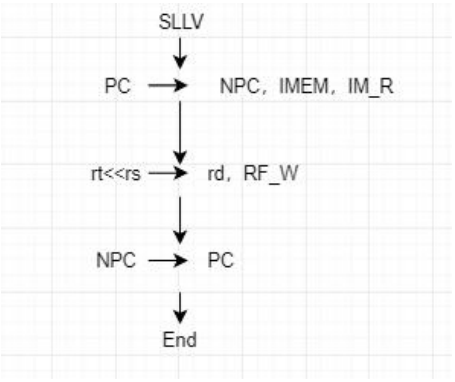
(2)根据 SLL 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU)、无符号扩展元件(ext5)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



⑱SLLV：

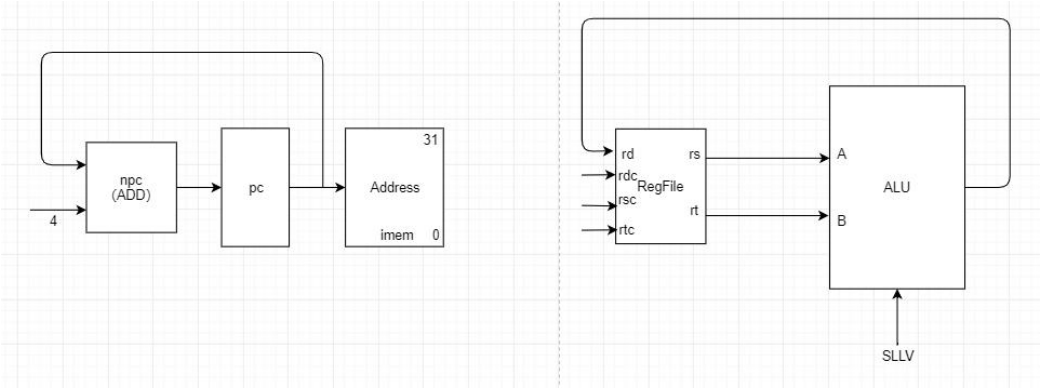
(1)确定 SLLV 所需的操作：取指令、R[rd]→R[rt]<<R[rs]、PC→PC+4



(2)根据 SLLV 的操作确定所需器件，PC 寄存器、指令存储器（instruction

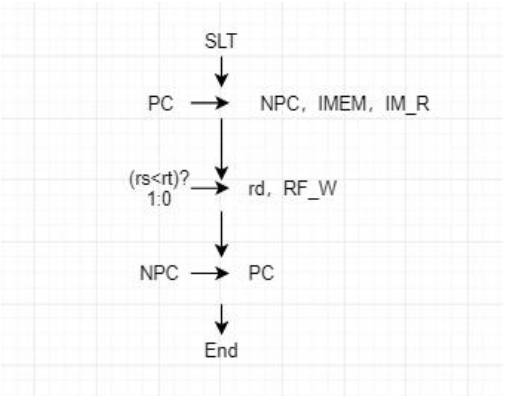
memory）、寄存器堆（regfile）、算术逻辑单元(ALU)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



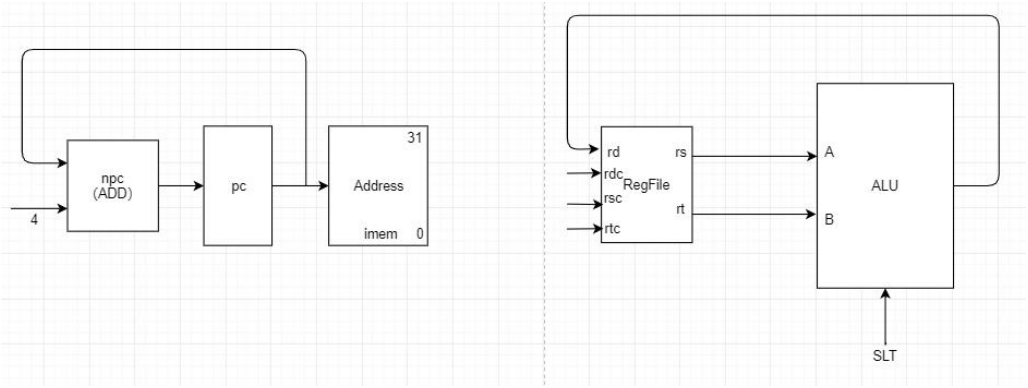
⑲SLT：

(1)确定 SLT 所需的操作：取指令、if (R[rs]<R[rt]) then R[rd]=1 else R[rd]=0、PC→PC+4



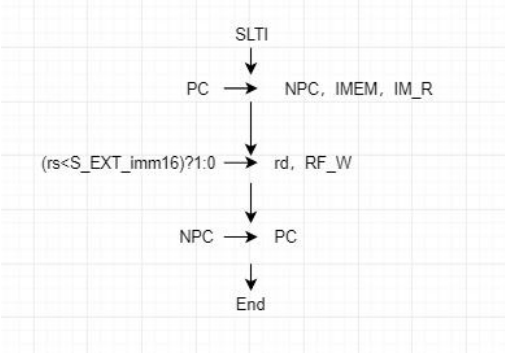
(2)根据 SLT 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



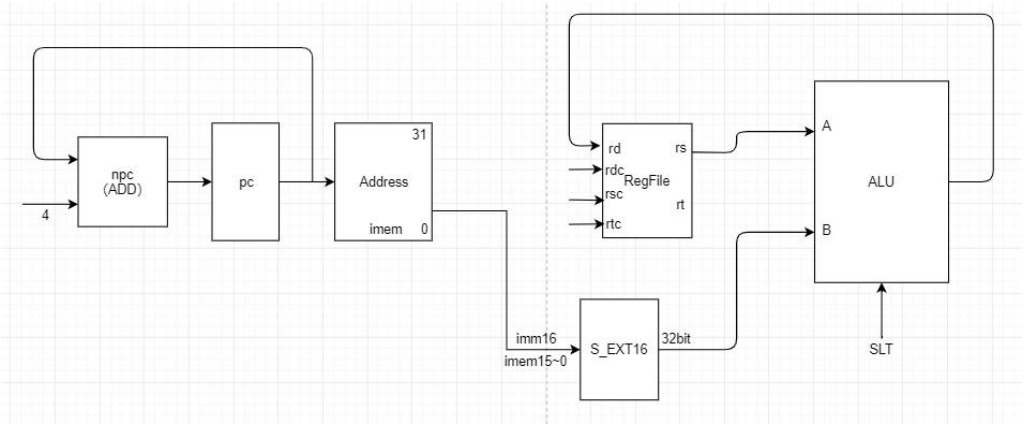
⑳ SLTI：

(1)确 SLTI 所需的操作：取指令、立即数符号扩展、if (R[rs]< signed_ext(imm16)) then R[rd]=1 else R[rd]=0、 PC→PC+4



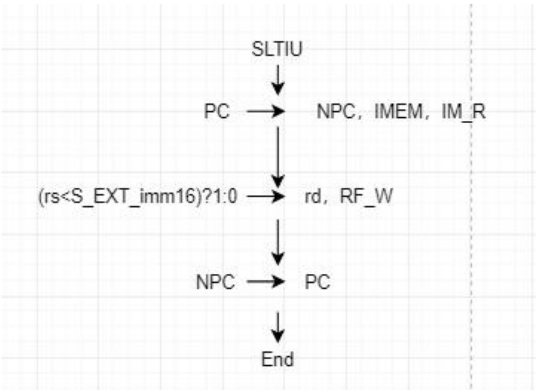
(2)根据 SLTI 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU)、有符号扩展元件(s_ext16)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



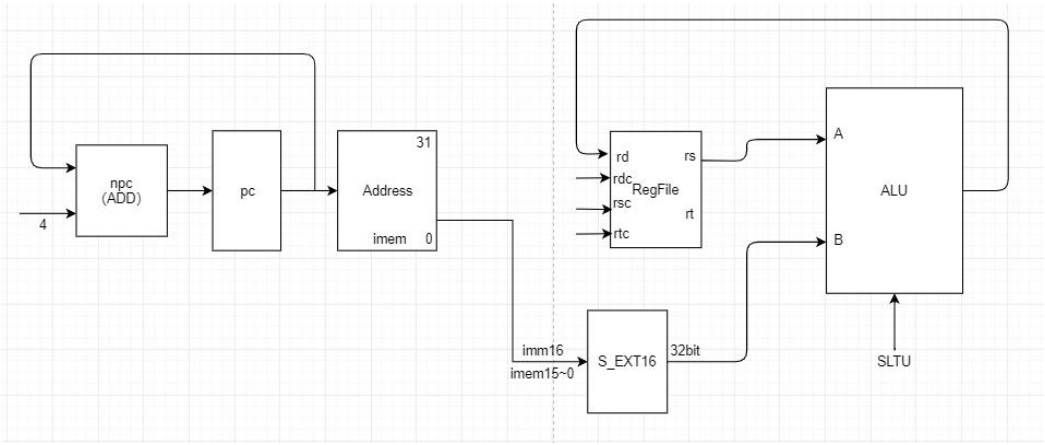
㉑ SLTIU：

(1)确 SLTIU 所需的操作：取指令、立即数符号扩展、if (R[rs]< signed_ext(imm16)) then R[rd]=1 else R[rd]=0、 PC→PC+4



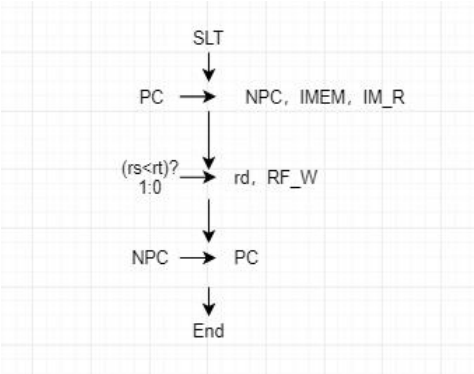
(2)根据 SLTIU 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU)、有符号扩展元件(s_ext16)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



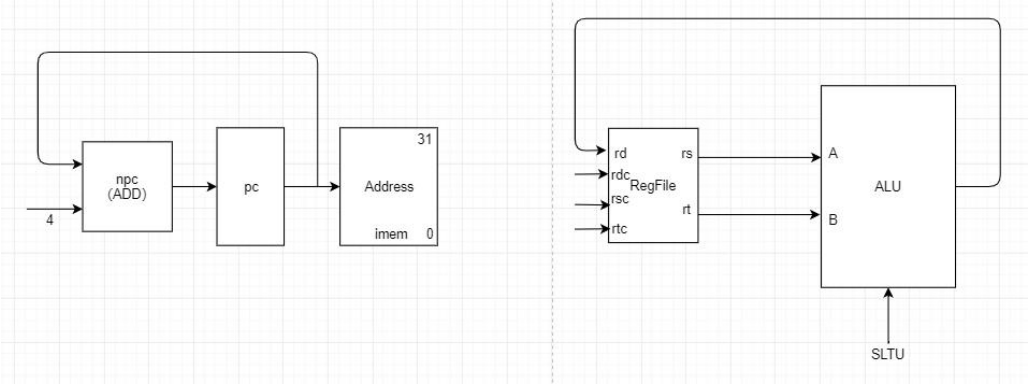
㉒SLTU：

(1)确定 SLTU 所需的操作：取指令、if (R[rs]<R[rt]) then R[rd]=1 else R[rd]=0、PC→PC+4



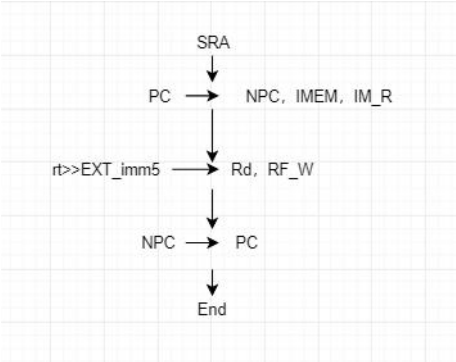
(2)根据 SLTU 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



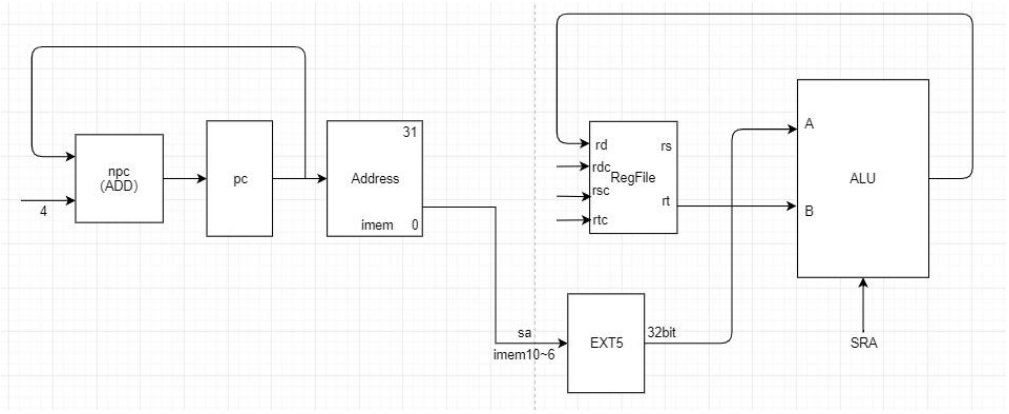
㉓SRA：

(1)确定 SRA 所需的操作：取指令、立即数扩展、R[rd]→R[rt]>>sign_ext(shamt)(arithmetic)、PC→PC+4



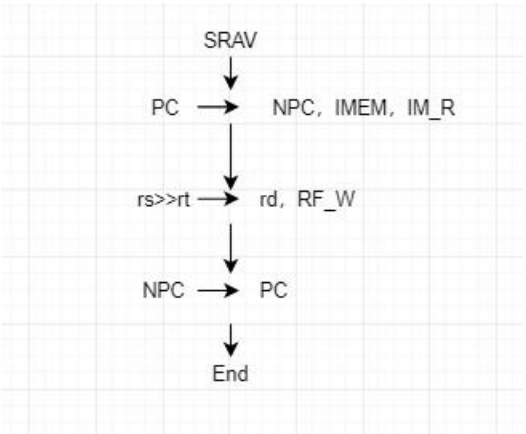
(2)根据 SRA 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU)、有符号扩展元件(s_ext5)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



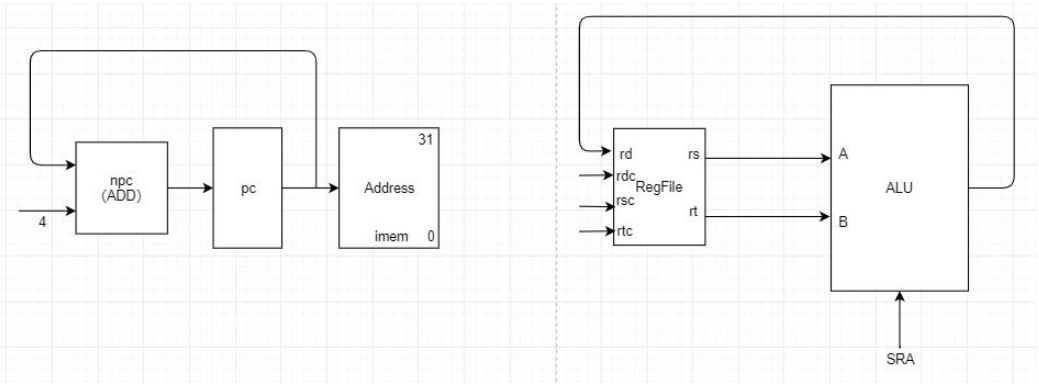
㉔SRAV：

(1)确定 SRAV 所需的操作：取指令、R[rd]→R[rt]>>R[rs](arithmetic)、PC→PC+4



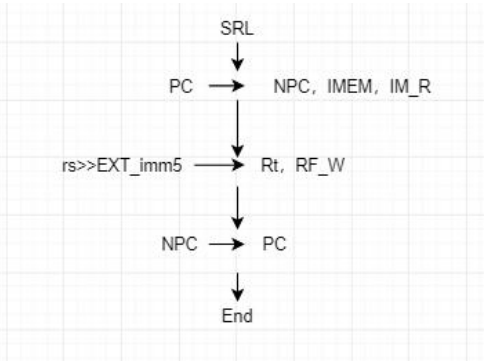
(2)根据 SRAV 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



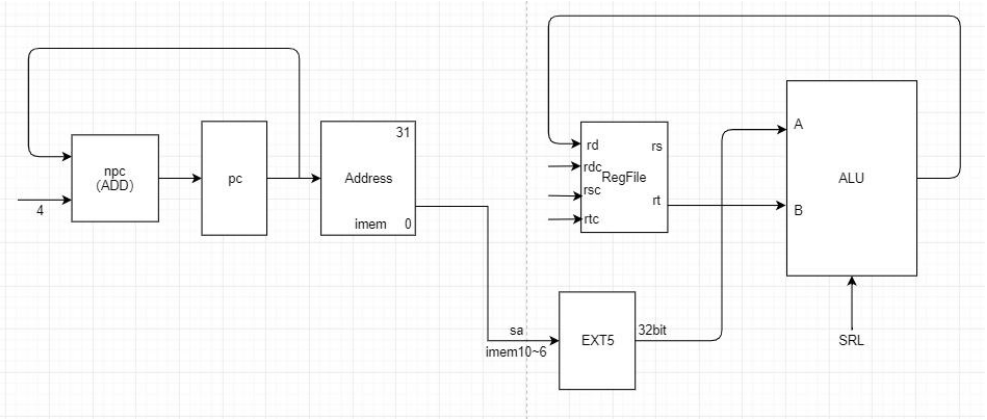
㉕SRL：

(1)确定 SRL 所需的操作：取指令、立即数扩展、R[rd]→R[rt]>> sign_ext(shamt)(logic)、PC→PC+4



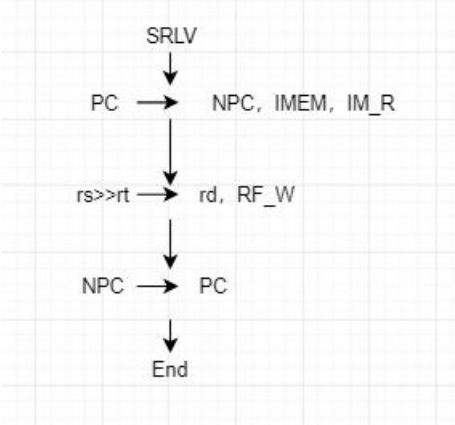
(2)根据 SRL 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU)、有符号扩展元件(s_ext5)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



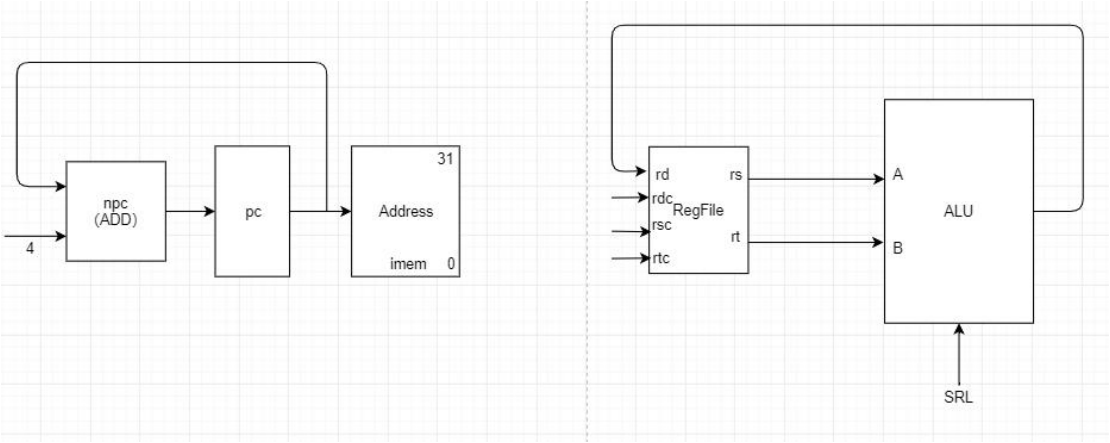
㉖SRLV：

(1)确定 SRLV 所需的操作：取指令、R[rd]→R[rt]>>R[rs](logic)、PC→PC+4



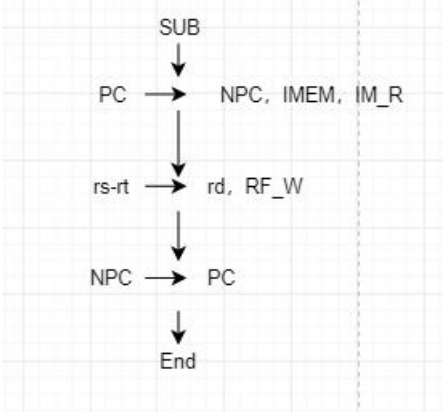
(2)根据 SRLV 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



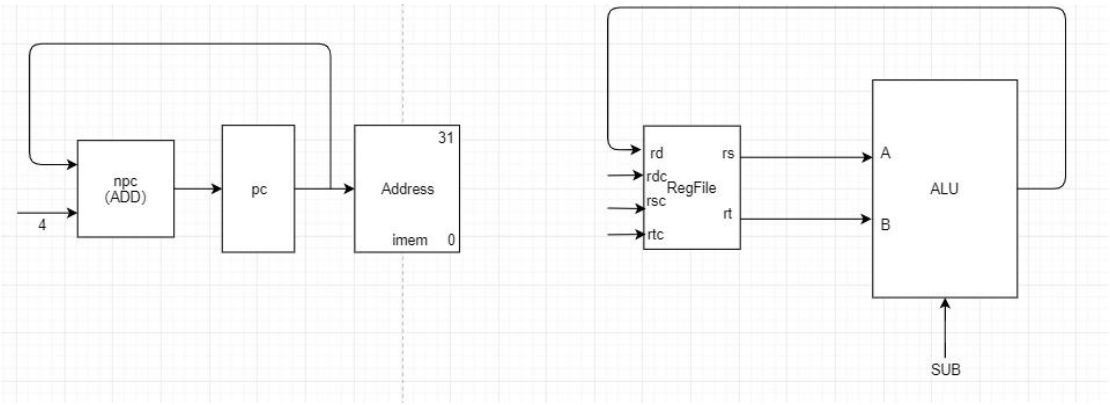
㉗SUB：

(1)确定 SUB 所需的操作：取指令、R[rd]→R[rs]-R[rt] 、PC→PC+4



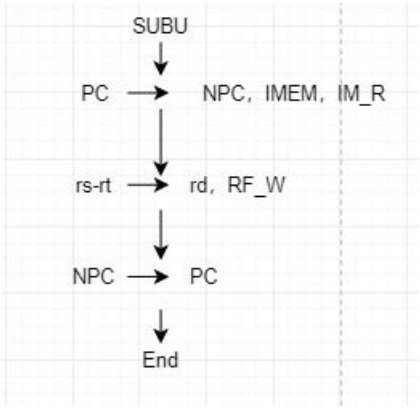
(2)根据 SUB 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



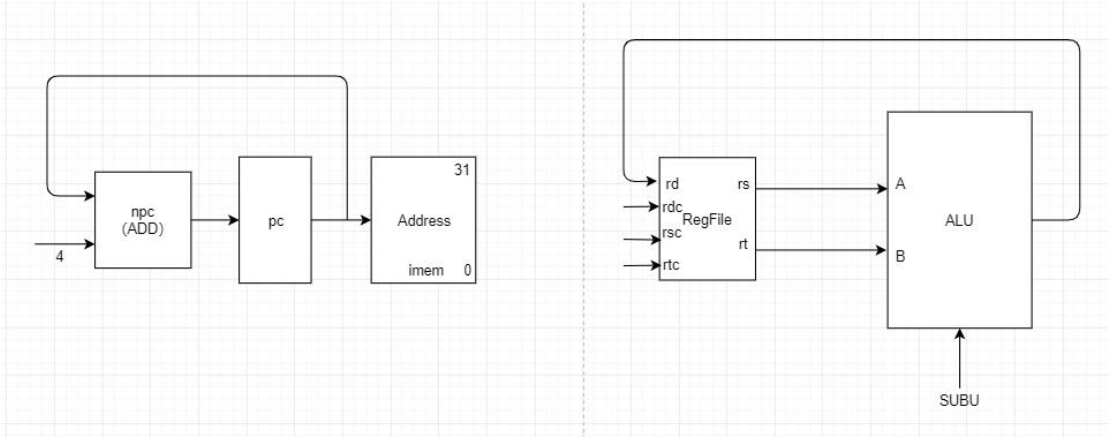
㉘SUBU：

(1)确定 SUBU 所需的操作：取指令、R[rd]→R[rs]-R[rt] 、PC→PC+4



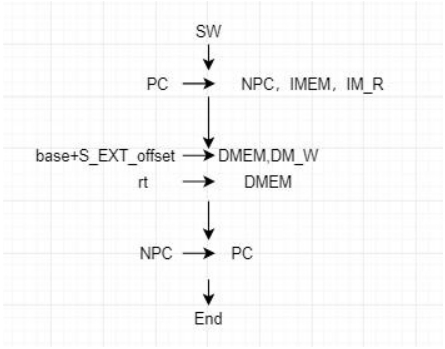
(2)根据 SUBU 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



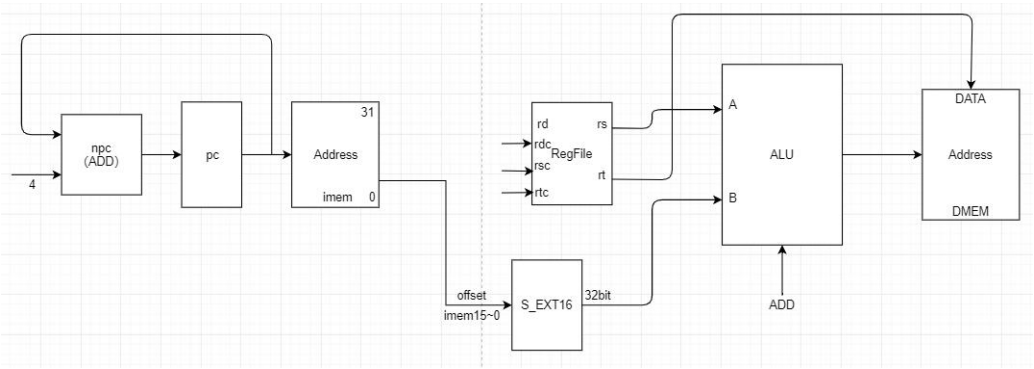
㉙ SW：

(1)确定 SW 所需的操作：取指令、立即数符号扩展、R[rs]+sign_ext(imm16)、MEM[R[rs]+sign_ext(imm16)]→R[rt]、PC→PC+4



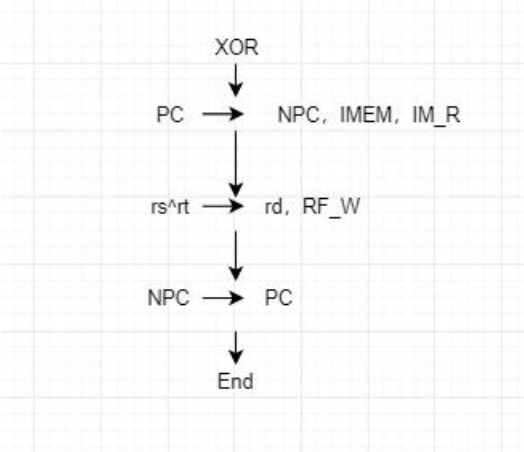
(2)根据 SW 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory)、寄存器堆（regfile)、算术逻辑单元(ALU)、有符号扩展元件(s_ext16)数据存储器

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



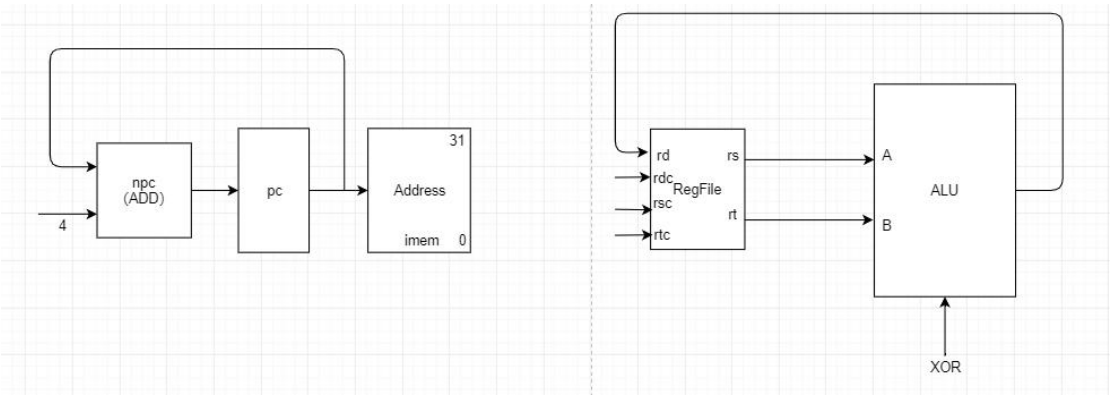
㉚ XOR：

(1)确定 XOR 所需的操作：取指令、R[rd]→R[rs]^R[rt]、PC→PC+4



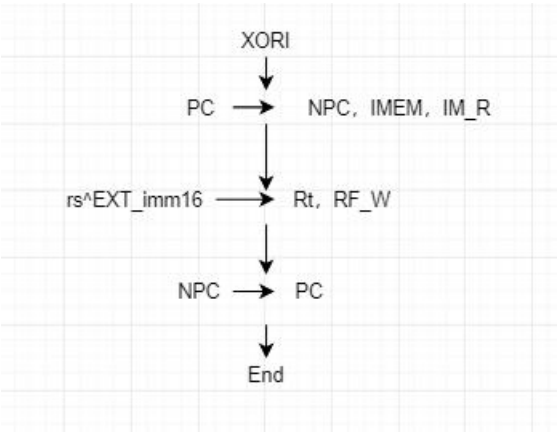
(2)根据 XOR 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory）、寄存器堆（regfile）、算术逻辑单元(ALU)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



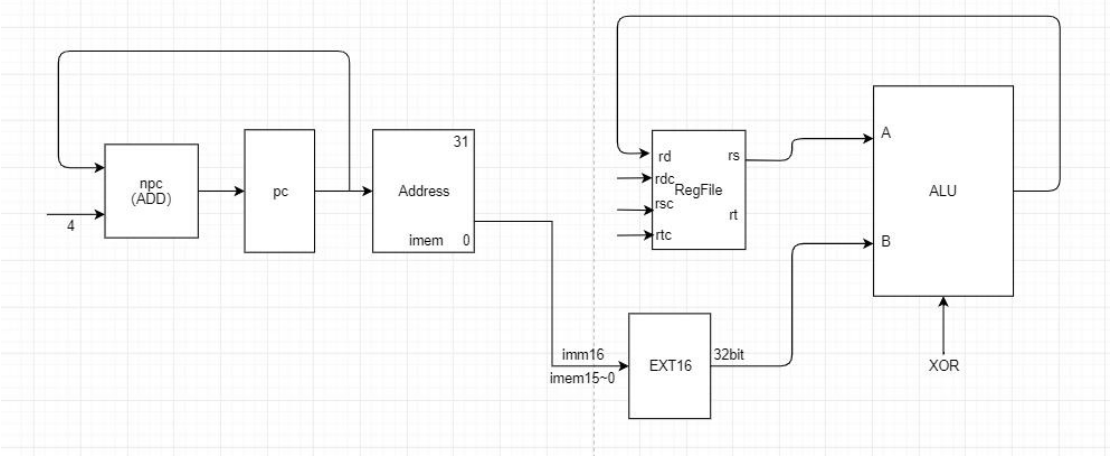
㉛XORI：

(1)确定 XORI 所需的操作：取指令、立即数符号扩展、R[rt]→R[rs]^ unsign_ext(imm16)、 PC→PC+4



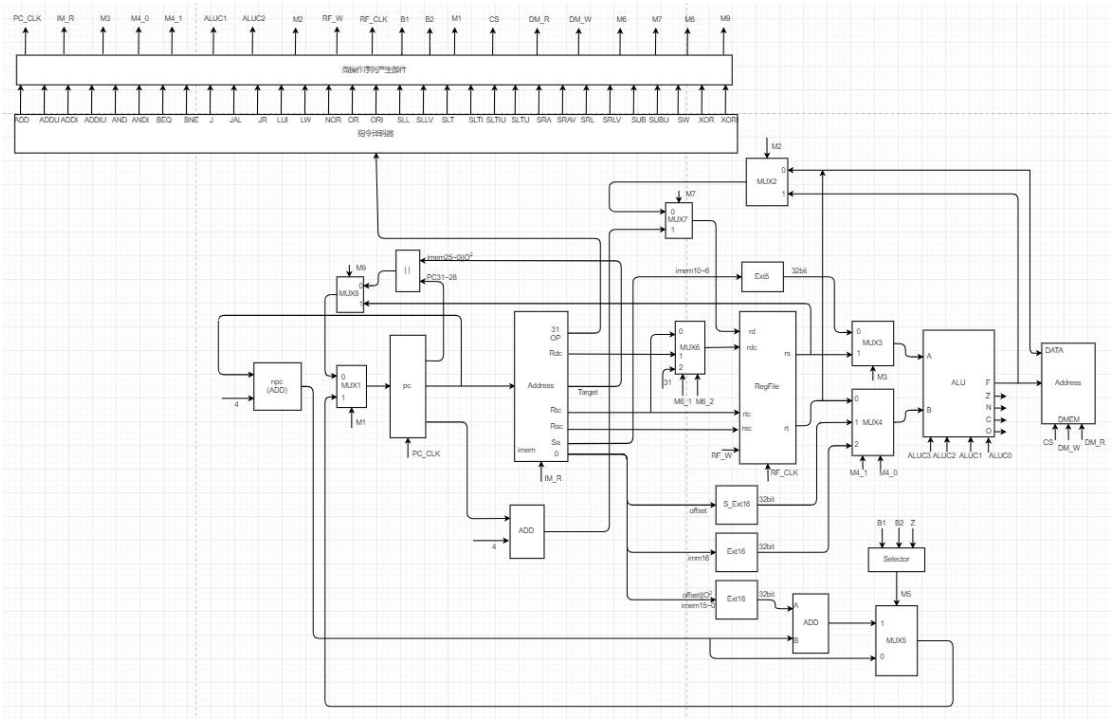
(2)根据 XORI 的操作确定所需器件，PC 寄存器、指令存储器（instruction memory)、寄存器堆（regfile)、算术逻辑单元(ALU)、无符号扩展元件(ext16)

(3)根据指令所需用到的操作及部件的输入输出关系，可以得到如下数据通路：



3.CPU 整体通路

综合考虑各条指令所需数据通路，可以构建 CPU 的整体数据通路如下：



4.控制信号的产生

根据各条指令所需要的数据通路，可以得到控制器的控制信号如下各表所示：

	ADD	ADDU	SUBU	SUB	AND
PC_CLK	1	1	1	1	1
IM_R	1	1	1	1	1
Rsc4-0	IM25-21	IM25-21	IM25-21	IM25-21	IM25-21
Rtc4-0	IM20-16	IM20-16	IM20-16	IM20-16	IM20-16
Rdc4-0	IM15-11	IM15-11	IM15-11	IM15-11	IM15-11
M1	1	1	1	1	1
M2	1	1	1	1	1
M3	1	1	1	1	1
M4_0	0	0	0	0	0
M4_1	0	0	0	0	0
B1	0	0	0	0	0
B2	0	0	0	0	0
M5	0	0	0	0	0
M6_1	1	1	1	1	1
M6_2	0	0	0	0	0
M7	0	0	0	0	0
M8	X	X	X	X	X
RF_W	1	1	1	1	1

RF_CLK	1	1	1	1	1
A0	0	0	1	1	0
A1	1	0	0	1	0
A2	0	0	0	0	1
A3	0	0	0	0	0
DM_w	0	0	0	0	0
DM_r	0	0	0	0	0
DM_cs	0	0	0	0	0

	OR	XOR	NOR	SLT	SLTU
PC_CLK	1	1	1	1	1
IM_R	1	1	1	1	1
Rsc4-0	IM25-21	IM25-21	IM25-21	IM25-21	IM25-21
Rtc4-0	IM20-16	IM20-16	IM20-16	IM20-16	IM20-16
Rdc4-0	IM15-11	IM15-11	IM15-11	IM15-11	IM15-11
M1	1	1	1	1	1
M2	1	1	1	1	1
M3	1	1	1	1	1
M4_0	0	0	0	0	0
M4_1	0	0	0	0	0
B1	0	0	0	0	0
B2	0	0	0	0	0
M5	0	0	0	0	0
M6_1	1	1	1	1	1
M6_2	0	0	0	0	0
M7	0	0	0	0	0
M8	X	X	X	X	X
RF_W	1	1	1	1	1
RF_CLK	1	1	1	1	1
A0	1	0	1	1	0
A1	0	1	1	1	1
A2	1	1	1	0	0
A3	0	0	0	1	1
DM_w	0	0	0	0	0
DM_r	0	0	0	0	0
DM_cs	0	0	0	0	0

	SLLV	SRLV	SRAV	ADDI	ORI
PC_CLK	1	1	1	1	1
IM_R	1	1	1	1	1
Rsc4-0	IM25-21	IM25-21	IM25-21	IM25-21	IM25-21
Rtc4-0	IM20-16	IM20-16	IM20-16	X	X
Rdc4-0	IM15-11	IM15-11	IM15-11	IM20-16	IM20-16
M1	1	1	1	1	1
M2	1	1	1	1	1
M3	1	1	1	1	1
M4_0	0	0	0	1	0
M4_1	0	0	0	0	1
B1	0	0	0	0	0
B2	0	0	0	0	0
M5	0	0	0	0	0
M6_1	1	1	1	0	0
M6_2	0	0	0	0	0
M7	0	0	0	0	0
M8	X	X	X	X	X
RF_W	1	1	1	1	1
RF_CLK	1	1	1	1	1
A0	X	1	0	0	1
A1	1	0	0	1	0
A2	1	1	1	0	1
A3	1	1	1	0	0
DM_w	0	0	0	0	0
DM_r	0	0	0	0	0
DM_cs	0	0	0	0	0

	SLTI	SLTIU	ADDIU	ANDI	XORI
PC_CLK	1	1	1	1	1
IM_R	1	1	1	1	1
Rsc4-0	IM25-21	IM25-21	IM25-21	IM25-21	IM25-21
Rtc4-0	X	X	X	X	X
Rdc4-0	IM20-16	IM20-16	IM20-16	IM20-16	IM20-16

M1	1	1	1	1	1
M2	1	1	1	1	1
M3	1	1	1	1	1
M4_0	1	1	1	0	0
M4_1	0	0	0	1	1
B1	0	0	0	0	0
B2	0	0	0	0	0
M5	0	0	0	0	0
M6_1	0	0	0	0	0
M6_2	0	0	0	0	0
M7	0	0	0	0	0
M8	X	X	X	X	X
RF_W	1	1	1	1	1
RF_CLK	1	1	1	1	1
A0	1	0	0	0	0
A1	1	1	0	0	1
A2	0	0	0	1	1
A3	1	1	0	0	0
DM_w	0	0	0	0	0
DM_r	0	0	0	0	0
DM_cs	0	0	0	0	0

	LUI	SRL	SLL	SRA	BEQ(Z=1)
PC_CLK	1	1	1	1	1
IM_R	1	1	1	1	1
Rsc4-0	IM25-21	X	X	X	IM25-21
Rtc4-0	X	IM20-16	IM20-16	IM20-16	IM20-16
Rdc4-0	IM20-16	IM15-11	IM15-11	IM15-11	IM15-11
M1	1	1	1	1	1
M2	1	1	1	1	1
M3	1	0	0	0	1
M4_0	0	0	0	0	0
M4_1	1	0	0	0	0
B1	0	0	0	0	1
B2	0	0	0	0	0
M5	0	0	0	0	1
M6_1	0	1	1	1	X
M6_2	0	0	0	0	0

M7	0	0	0	0	X
M8	X	X	X	X	X
RF_W	1	1	1	1	0
RF_CLK	1	1	1	1	0
A0	X	1	X	0	1
A1	0	0	1	0	0
A2	0	1	1	1	0
A3	1	1	1	1	0
DM_w	0	0	0	0	0
DM_r	0	0	0	0	0
DM_cs	0	0	0	0	0

	BNE(Z=0)	LW	SW	J	JR	JAL
PC_CLK	1	1	1	1	1	1
IM_R	1	1	1	1	1	1
Rsc4-0	IM25-21	IM25-21	IM25-21	X	IM25-21	X
Rtc4-0	IM20-16	X	IM20-16	X	X	X
Rdc4-0	IM15-11	IM15-11	X	X	X	X
M1	1	1	1	0	0	0
M2	1	0	1	X	X	X
M3	1	1	1	X	X	X
M4_0	0	1	1	X	X	X
M4_1	0	0	0	X	X	X
B1	0	0	0	0	0	0
B2	1	0	0	0	0	0
M5	1	0	0	X	X	X
M6_1	X	0	1	X	X	0
M6_2	0	0	0	X	X	1
M7	X	0	0	X	X	1
M8	X	X	X	0	1	0
RF_W	0	1	0	0	0	1
RF_CLK	0	1	0	0	0	1
A0	1	0	0	X	X	X
A1	0	0	0	X	X	X
A2	0	0	0	X	X	X
A3	0	0	0	X	X	X
DM_w	0	0	1	0	0	X
DM_r	0	1	0	0	0	X

DM_cs	0	1	1	0	0	X

其中各个控制信号的产生方式以及各个控制信号的具体作用可以参见后面具体模块(如微操作序列产生器)的分析。

三、主要模块设计

1.算术逻辑单元 ALU

负责执行算术逻辑运算的模块。根据控制信号 ALUC[4:0]的数值执行不同的算术逻辑运算，其中控制信号与所执行的运算的关系如下：

ALUC	执行运算
0000	ADDU
0010	ADD
0001	SUBU
0011	SUB
0100	AND
0101	OR
0110	XOR
0111	NOR
100X	LUI
1011	SLT
1010	SLTU
1100	SRA
111X	SLL/SLR
1101	SRL

同时 ALU 模块具有四位标志位：

①zero：运算结果为 0 则 zero=1，否则 zero=0

②carry：无符号数加法运算发生上溢出则 carry=1，无符号数减法发生下溢出则 carry=1，无符号数比较运算 a-b<0 则 carry=1,移位运算 carry 为最后一次被移出的位的数值，其余情况 carry=0

③negative：运算最终结果最高位为 1 或有符号数比较运算 a-b<0 则 negative=1,否则 negative=0

④overflow：有符号数加减法发生溢出时 overflow=1，否则 overflow=0

源代码如下：

```
module alu(
input [31:0]a,
input [31:0]b,
```

```
input [3:0]aluc,
output [31:0]r,
output z,
output c,
output n,
output o
);

reg [32:0] r1;
//行为级 ALU,r1 暂存运算结果，r2 辅助运算便于判断是否溢出，ta 和 tb 将 a、b 转化为有符号数便于进行有符号数运算

wire signed [31:0] ta;
wire signed [31:0] tb;
assign ta=a;
assign tb=b;

always@ (aluc or a or b)
begin
  casez(aluc)
    4'b0000:begin r1=a+b;end//ADDU，运算过程中使用双符号位便于判断溢出
    4'b0010:begin r1={a[31],a}+{b[31],b};end//ADD
    4'b0001:begin r1=a-b;end//SUBU，运算过程中使用双符号位便于判断溢出
    4'b0011:begin r1={a[31],a}-{b[31],b};end//SUB
    4'b0100:begin r1=a&b;end//AND
    4'b0101:begin r1=a|b;end//OR
    4'b0110:begin r1=a^b;end//XOR
    4'b0111:begin r1=~(a|b);end//NOR
    4'b100?:begin r1={b[15:0],16'b0};end//LUI
    4'b1011:begin r1=(ta<tb)?1:0;end//SLT
    4'b1010:begin r1=(a<b)?1:0;end//SLTU
    4'b1100:begin if(a==0) {r1[31:0],r1[32]}={b,1'b0};else {r1[31:0],r1[32]}=tb>>>(ta-1);end//SRA，运算过程中附加一位实现记录最后移出去的一位
    4'b111?:begin r1=b<<a;end//SLL/SLR
    4'b1101:begin if(a==0) {r1[31:0],r1[32]}={b,1'b0};else {r1[31:0],r1[32]}=b>>(a-1);end//SRL，运算过程中附加一位实现记录最后移出去的一位
    default:;
  endcase
end

assign r=r1[31:0];
assign  c=(aluc==4'b0000|aluc==4'b0001|aluc==4'b1010|aluc==4'b1100|aluc==4'b1101|aluc==4'b1111|aluc==4'b1110)?r1[32]:1'b0;// 无符号数加法运算发生上溢或无符号数减法发生下溢或无符号数比较运算 a-b<0 时该标志位为 1。移位运算该标志位为最后一次被移出的位的数值。
assign z=(r1==32'b0)?1:0;//z=1 表示运算结果为 0,z=0 表示运算结果不为 0，所有运算均影响此标志位。
```

```
assign n=r[31];//有符号数运算，操作数和结果均用二进制补码的形式表示，n=1 表示结果为负数，n=0 表示结果为正数或零。其他运算最终结果 r[31]为 1 则 n 为 1。
assign o=(aluc==4'b0010|aluc==4'b0011)?(r1[32]^r1[31]):1'b0;//对于有符号加减法运算，操作数和运算结果均用二进制补码的形式表示，有溢出时该标志位 o=1。

Endmodule
```

2.PC 寄存器

通过实例化 32 个并行 D 触发器构成 32 位的 PC 寄存器，在时钟上升沿若写信号有效则更新寄存器值，在 reset 信号下降沿异步清零。由于我们设计的 MIPS CPU 为哈佛结构，指令地址从 0x00000000 开始，而测试所使用的 Mars 模拟器为冯诺依曼结构，指令的地址从 0x00400000 开始，因此需要设置 PC 寄存器的第 23 位对应的 D 触发器具有和其他触发器不同的特性。在其他触发器清零时，该触发器置 1，从而使得此时 PC 寄存器中的值为 0x00400000，实现逻辑地址到物理地址的映射。

源代码如下：

```
module pcreg2(clk,rst,ena,data_in,data_out);
input clk,rst,ena;
input [31:0] data_in;
output [31:0] data_out;
wire [31:0] data_out;
wire [31:0] d_data_out;

D_FF uut0(clk,data_in[0],rst,ena,data_out[0],d_data_out[0]);//依次实例化 32 个 D 触发器，并行实现 32 位 pc 寄存器
D_FF uut1(clk,data_in[1],rst,ena,data_out[1],d_data_out[1]);
D_FF uut2(clk,data_in[2],rst,ena,data_out[2],d_data_out[2]);
D_FF uut3(clk,data_in[3],rst,ena,data_out[3],d_data_out[3]);
D_FF uut4(clk,data_in[4],rst,ena,data_out[4],d_data_out[4]);
D_FF uut5(clk,data_in[5],rst,ena,data_out[5],d_data_out[5]);
D_FF uut6(clk,data_in[6],rst,ena,data_out[6],d_data_out[6]);
D_FF uut7(clk,data_in[7],rst,ena,data_out[7],d_data_out[7]);
D_FF uut8(clk,data_in[8],rst,ena,data_out[8],d_data_out[8]);
D_FF uut9(clk,data_in[9],rst,ena,data_out[9],d_data_out[9]);
D_FF uut10(clk,data_in[10],rst,ena,data_out[10],d_data_out[10]);
D_FF uut11(clk,data_in[11],rst,ena,data_out[11],d_data_out[11]);
D_FF uut12(clk,data_in[12],rst,ena,data_out[12],d_data_out[12]);
D_FF uut13(clk,data_in[13],rst,ena,data_out[13],d_data_out[13]);
D_FF uut14(clk,data_in[14],rst,ena,data_out[14],d_data_out[14]);
D_FF uut15(clk,data_in[15],rst,ena,data_out[15],d_data_out[15]);
D_FF uut16(clk,data_in[16],rst,ena,data_out[16],d_data_out[16]);
D_FF uut17(clk,data_in[17],rst,ena,data_out[17],d_data_out[17]);
D_FF uut18(clk,data_in[18],rst,ena,data_out[18],d_data_out[18]);
D_FF uut19(clk,data_in[19],rst,ena,data_out[19],d_data_out[19]);
```

```
D_FF uut20(clk,data_in[20],rst,ena,data_out[20],d_data_out[20]);
D_FF uut21(clk,data_in[21],rst,ena,data_out[21],d_data_out[21]);
D_F uut22(clk,data_in[22],rst,ena,data_out[22],d_data_out[22]);//为了使基址为 32'h00400000,x 需要设置一个特殊触发器
D_FF uut23(clk,data_in[23],rst,ena,data_out[23],d_data_out[23]);
D_FF uut24(clk,data_in[24],rst,ena,data_out[24],d_data_out[24]);
D_FF uut25(clk,data_in[25],rst,ena,data_out[25],d_data_out[25]);
D_FF uut26(clk,data_in[26],rst,ena,data_out[26],d_data_out[26]);
D_FF uut27(clk,data_in[27],rst,ena,data_out[27],d_data_out[27]);
D_FF uut28(clk,data_in[28],rst,ena,data_out[28],d_data_out[28]);
D_FF uut29(clk,data_in[29],rst,ena,data_out[29],d_data_out[29]);
D_FF uut30(clk,data_in[30],rst,ena,data_out[30],d_data_out[30]);
D_FF uut31(clk,data_in[31],rst,ena,data_out[31],d_data_out[31]);

endmodule

module D_F(CLK,D,RST_n,ena,Q1,Q2);//单个 D 触发器
input CLK,D,RST_n,ena;
output Q1,Q2;
reg Q1,Q2;

always @(posedge CLK or posedge RST_n)//为了使基址为 32'h00400000,x 需要设置一个特殊触发器
begin
   if ((RST_n==0)&&(ena==1)) begin
      if (D==1) begin
         Q1=1;
         Q2=0;
      end
      else begin
         Q1=0;
         Q2=1;
      end
   end
   else
   if (RST_n==1) begin
      Q1=1;
      Q2=0;
   end
end

endmodule
```

3.寄存器堆 regfile

通过实例化 32 个并行 32 位寄存器(可实例化 PC 寄存器)得到寄存器堆。其中第 0 号寄

存器恒置 0 因此将该寄存器的写信号恒置为无效，不允许修改该寄存器的值。寄存器堆设置有两个读地址接口 addr1、addr2 可以一次读出寄存器堆中的两个寄存器的数据 wdata1、wdata2，设置有一个写地址接口，在写信号有效时将对应的寄存器的值更新为 rdata。reset 信号下降沿时，将寄存器堆所有寄存器数据清零。

源代码如下：

```
module regfile(clk,rst,we,raddr1,raddr2,waddr,wdata,rdata1,rdata2);
input clk,rst,we;
input [4:0] raddr1;
input [4:0] raddr2;
input [4:0] waddr;
input [31:0] wdata;
output [31:0] rdata1;
output [31:0] rdata2;
wire [31:0] d;
wire [31:0] array_reg[0:31];

decoder srt (waddr,we,d);
pcreg uvw0 (clk,rst,0,wdata,array_reg[0]);//依次实例化寄存器中的各个寄存器，并行实现寄存器堆,特殊地有 0 寄存器不能被写入
pcreg uvw1 (clk,rst,d[1],wdata,array_reg[1]);
pcreg uvw2 (clk,rst,d[2],wdata,array_reg[2]);
pcreg uvw3 (clk,rst,d[3],wdata,array_reg[3]);
pcreg uvw4 (clk,rst,d[4],wdata,array_reg[4]);
pcreg uvw5 (clk,rst,d[5],wdata,array_reg[5]);
pcreg uvw6 (clk,rst,d[6],wdata,array_reg[6]);
pcreg uvw7 (clk,rst,d[7],wdata,array_reg[7]);
pcreg uvw8 (clk,rst,d[8],wdata,array_reg[8]);
pcreg uvw9 (clk,rst,d[9],wdata,array_reg[9]);
pcreg uvw10 (clk,rst,d[10],wdata,array_reg[10]);
pcreg uvw11 (clk,rst,d[11],wdata,array_reg[11]);
pcreg uvw12 (clk,rst,d[12],wdata,array_reg[12]);
pcreg uvw13 (clk,rst,d[13],wdata,array_reg[13]);
pcreg uvw14 (clk,rst,d[14],wdata,array_reg[14]);
pcreg uvw15 (clk,rst,d[15],wdata,array_reg[15]);
pcreg uvw16 (clk,rst,d[16],wdata,array_reg[16]);
pcreg uvw17 (clk,rst,d[17],wdata,array_reg[17]);
pcreg uvw18 (clk,rst,d[18],wdata,array_reg[18]);
pcreg uvw19 (clk,rst,d[19],wdata,array_reg[19]);
pcreg uvw20 (clk,rst,d[20],wdata,array_reg[20]);
pcreg uvw21 (clk,rst,d[21],wdata,array_reg[21]);
pcreg uvw22 (clk,rst,d[22],wdata,array_reg[22]);
pcreg uvw23 (clk,rst,d[23],wdata,array_reg[23]);
pcreg uvw24 (clk,rst,d[24],wdata,array_reg[24]);
pcreg uvw25 (clk,rst,d[25],wdata,array_reg[25]);
```

```
pcreg uvw26 (clk,rst,d[26],wdata,array_reg[26]);
pcreg uvw27 (clk,rst,d[27],wdata,array_reg[27]);
pcreg uvw28 (clk,rst,d[28],wdata,array_reg[28]);
pcreg uvw29 (clk,rst,d[29],wdata,array_reg[29]);
pcreg uvw30 (clk,rst,d[30],wdata,array_reg[30]);
pcreg uvw31 (clk,rst,d[31],wdata,array_reg[31]);

assign a0=array_reg[0];
assign a1=array_reg[1];
assign a2=array_reg[2];
assign a3=array_reg[3];
assign a4=array_reg[4];
assign a5=array_reg[5];
assign a6=array_reg[6];
assign a7=array_reg[7];
assign a8=array_reg[8];
assign a9=array_reg[9];
assign a10=array_reg[10];
assign a11=array_reg[11];
assign a12=array_reg[12];
assign a13=array_reg[13];
assign a14=array_reg[14];
assign a15=array_reg[15];
assign a16=array_reg[16];
assign a17=array_reg[17];
assign a18=array_reg[18];
assign a19=array_reg[19];
assign a20=array_reg[20];
assign a21=array_reg[21];
assign a22=array_reg[22];
assign a23=array_reg[23];
assign a24=array_reg[24];
assign a25=array_reg[25];
assign a26=array_reg[26];
assign a27=array_reg[27];
assign a28=array_reg[28];
assign a29=array_reg[29];
assign a30=array_reg[30];
assign a31=array_reg[31];

selector321 la1
(array_reg[0],array_reg[1],array_reg[2],array_reg[3],array_reg[4],array_reg[5],array_reg[6],array_reg[7],array_reg[8],array_reg[9],array_
reg[10],array_reg[11],array_reg[12],array_reg[13],array_reg[14],array_reg[15],array_reg[16],array_reg[17],array_reg[18],array_reg[19],a
rray_reg[20],array_reg[21],array_reg[22],array_reg[23],array_reg[24],array_reg[25],array_reg[26],array_reg[27],array_reg[28],array_reg
```

```
[29],array_reg[30],array_reg[31],raddr1[4],raddr1[3],raddr1[2],raddr1[1],raddr1[0],rdata1,1);
selector321 la2
(array_reg[0],array_reg[1],array_reg[2],array_reg[3],array_reg[4],array_reg[5],array_reg[6],array_reg[7],array_reg[8],array_reg[9],array_
reg[10],array_reg[11],array_reg[12],array_reg[13],array_reg[14],array_reg[15],array_reg[16],array_reg[17],array_reg[18],array_reg[19],a
rray_reg[20],array_reg[21],array_reg[22],array_reg[23],array_reg[24],array_reg[25],array_reg[26],array_reg[27],array_reg[28],array_reg
[29],array_reg[30],array_reg[31],raddr2[4],raddr2[3],raddr2[2],raddr2[1],raddr2[0],rdata2,1);

endmodule

module decoder(iData,iEna,oData);//译码器，确定地址所对应的是哪个寄存器
input [4:0] iData;
input iEna;
output [31:0] oData;

assign oData=(iEna==1)?(1<<iData):32'hzzzzzzzz;

endmodule

module
selector321(iC0,iC1,iC2,iC3,iC4,iC5,iC6,iC7,iC8,iC9,iC10,iC11,iC12,iC13,iC14,iC15,iC16,iC17,iC18,iC19,iC20,iC21,iC22,iC23,iC24,i
C25,iC26,iC27,iC28,iC29,iC30,iC31,iS4,iS3,iS2,iS1,iS0,oZ,ena);
input [31:0]
iC0,iC1,iC2,iC3,iC4,iC5,iC6,iC7,iC8,iC9,iC10,iC11,iC12,iC13,iC14,iC15,iC16,iC17,iC18,iC19,iC20,iC21,iC22,iC23,iC24,iC25,iC26,iC
27,iC28,iC29,iC30,iC31;
input iS4,iS3,iS2,iS1,iS0,ena;
output [31:0] oZ;
reg [31:0] oZ;
//存储所选择的寄存器中的内容

always@(ena or iS4 or iS3 or iS2 or iS1 or iS0 or iC0 or iC1 or iC2 or iC3 or iC4 or iC5 or iC6 or iC7 or iC8 or iC9 or iC10 or iC11 or
iC12 or iC13 or iC14 or iC15 or iC16 or iC17 or iC18 or iC19 or iC20 or iC21 or iC22 or iC23 or iC24 or iC25 or iC26 or iC27 or iC28
or iC29 or iC30 or iC31 )
begin
   if (ena==1) begin
      case ({iS4,iS3,iS2,iS1,iS0})
         5'b00000:oZ=iC0;
         5'b00001:oZ=iC1;
         5'b00010:oZ=iC2;
         5'b00011:oZ=iC3;
         5'b00100:oZ=iC4;
         5'b00101:oZ=iC5;
         5'b00110:oZ=iC6;
         5'b00111:oZ=iC7;
         5'b01000:oZ=iC8;
```

```
            5'b01001:oZ=iC9;
            5'b01010:oZ=iC10;
            5'b01011:oZ=iC11;
            5'b01100:oZ=iC12;
            5'b01101:oZ=iC13;
            5'b01110:oZ=iC14;
            5'b01111:oZ=iC15;
            5'b10000:oZ=iC16;
            5'b10001:oZ=iC17;
            5'b10010:oZ=iC18;
            5'b10011:oZ=iC19;
            5'b10100:oZ=iC20;
            5'b10101:oZ=iC21;
            5'b10110:oZ=iC22;
            5'b10111:oZ=iC23;
            5'b11000:oZ=iC24;
            5'b11001:oZ=iC25;
            5'b11010:oZ=iC26;
            5'b11011:oZ=iC27;
            5'b11100:oZ=iC28;
            5'b11101:oZ=iC29;
            5'b11110:oZ=iC30;
            5'b11111:oZ=iC31;
        endcase
    end
end

endmodule

module pcreg(clk,rst,ena,data_in,data_out);
input clk,rst,ena;
input [31:0] data_in;
output [31:0] data_out;
wire [31:0] data_out;
wire [31:0] d_data_out;

D_FF uut0(clk,data_in[0],rst,ena,data_out[0],d_data_out[0]);//依次实例化 32 个 D 触发器，并行实现 32 位 pc 寄存器
D_FF uut1(clk,data_in[1],rst,ena,data_out[1],d_data_out[1]);
D_FF uut2(clk,data_in[2],rst,ena,data_out[2],d_data_out[2]);
D_FF uut3(clk,data_in[3],rst,ena,data_out[3],d_data_out[3]);
D_FF uut4(clk,data_in[4],rst,ena,data_out[4],d_data_out[4]);
D_FF uut5(clk,data_in[5],rst,ena,data_out[5],d_data_out[5]);
D_FF uut6(clk,data_in[6],rst,ena,data_out[6],d_data_out[6]);
D_FF uut7(clk,data_in[7],rst,ena,data_out[7],d_data_out[7]);
```

```
D_FF uut8(clk,data_in[8],rst,ena,data_out[8],d_data_out[8]);
D_FF uut9(clk,data_in[9],rst,ena,data_out[9],d_data_out[9]);
D_FF uut10(clk,data_in[10],rst,ena,data_out[10],d_data_out[10]);
D_FF uut11(clk,data_in[11],rst,ena,data_out[11],d_data_out[11]);
D_FF uut12(clk,data_in[12],rst,ena,data_out[12],d_data_out[12]);
D_FF uut13(clk,data_in[13],rst,ena,data_out[13],d_data_out[13]);
D_FF uut14(clk,data_in[14],rst,ena,data_out[14],d_data_out[14]);
D_FF uut15(clk,data_in[15],rst,ena,data_out[15],d_data_out[15]);
D_FF uut16(clk,data_in[16],rst,ena,data_out[16],d_data_out[16]);
D_FF uut17(clk,data_in[17],rst,ena,data_out[17],d_data_out[17]);
D_FF uut18(clk,data_in[18],rst,ena,data_out[18],d_data_out[18]);
D_FF uut19(clk,data_in[19],rst,ena,data_out[19],d_data_out[19]);
D_FF uut20(clk,data_in[20],rst,ena,data_out[20],d_data_out[20]);
D_FF uut21(clk,data_in[21],rst,ena,data_out[21],d_data_out[21]);
D_FF uut22(clk,data_in[22],rst,ena,data_out[22],d_data_out[22]);
D_FF uut23(clk,data_in[23],rst,ena,data_out[23],d_data_out[23]);
D_FF uut24(clk,data_in[24],rst,ena,data_out[24],d_data_out[24]);
D_FF uut25(clk,data_in[25],rst,ena,data_out[25],d_data_out[25]);
D_FF uut26(clk,data_in[26],rst,ena,data_out[26],d_data_out[26]);
D_FF uut27(clk,data_in[27],rst,ena,data_out[27],d_data_out[27]);
D_FF uut28(clk,data_in[28],rst,ena,data_out[28],d_data_out[28]);
D_FF uut29(clk,data_in[29],rst,ena,data_out[29],d_data_out[29]);
D_FF uut30(clk,data_in[30],rst,ena,data_out[30],d_data_out[30]);
D_FF uut31(clk,data_in[31],rst,ena,data_out[31],d_data_out[31]);

endmodule

module D_FF(CLK,D,RST_n,ena,Q1,Q2);//单个 D 触发器
input CLK,D,RST_n,ena;
output Q1,Q2;
reg Q1,Q2;

always @(posedge CLK or posedge RST_n)
begin
   if ((RST_n==0)&&(ena==1)) begin
      if (D==1) begin
         Q1=1;
         Q2=0;
      end
      else begin
         Q1=0;
         Q2=1;
      end
   end
```

```
    else
    if (RST_n==1) begin
        Q1=0;
        Q2=1;
    end
end

endmodule
```

4.指令存储器 imem

通过实例化 2048 个 32 位寄存器得到，其控制信号 IM_R 有效时，根据 PC 寄存器的值取出对应地址的存储单元的指令信息。也可以使用 IP 核实现。

源代码如下：

```
module IMEM(
input clk,
input [10:0] addr,
input IM_R,
output [31:0] data_out
);
reg [31:0] mem [0:2047];

assign data_out = (IM_R==1)?mem[addr][31:0]:32'bzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz;

endmodule//该模块也可以使用 IP 核。
```

5.数据存储器 dmem

通过实例化 2048 个 32 位寄存器得到。其控制信号 DM_CS 有效时，允许对数据存储器进行读写。DM_W 信号有效时通过地址线接收到的地址，将对应地址的存储单元的数据更新为数据线上的数据。DM_R 信号有效时通过地址线接收到的地址，将对应地址的存储单元的数据送出到数据线上。

源代码如下：

```
module DMEM(
input clk,
input [10:0] addr,
input cs, r, w,
input [31:0] rdata,
output [31:0] wdata
);
reg [31:0] mem [0:2047];
```

```
always @(posedge clk) begin
    if (cs&w)
        mem[addr]=rdata;
end

assign wdata = r?mem[addr]:32'hzzzzzzzz;

endmodule
```

6.npc 寄存器

32 位寄存器，其中存储的值为 PC+4，为正常顺序执行指令的情况下，下一条指令的地址。其内容通过数据选择器 Mux5，在执行非跳转类指令时送到 PC 寄存器。

源代码如下：

```
module add4(
input [31:0] a,
output [31:0] r
);
assign r=a+4;//(npc，实质是+4 加法器)
endmodule
```

7.指令译码器

根据 PC 寄存器地址从指令寄存器中读取到的指令送入指令译码器，根据不同指令的指令格式的特点确定将要执行的指令是哪一种，用输出的 32 位信号代表 31 条指令，其中只有 1 位为高电平，代表了所要执行的指令。

源代码如下：

```
module instruction_decoder( input [31:0] instruction_code,output reg[31:0] instruction_type);
//指令译码器，输入为接收到的指令，输出为对应的指令类型
wire [11:0] temp;
assign temp = {instruction_code[31:26],instruction_code[5:0]};
//32~26 为 op，5~0 为 func，通过这两者判断指令类型，使用 32 位寄存器，表示是置 1 的位对应的指令类型

always @ (temp) begin
casez(temp)
    12'b000000100000:instruction_type=1<<0;//ADD
    12'b000000100001:instruction_type=1<<1;//ADDU
    12'b000000100010:instruction_type=1<<2;//SUB
    12'b000000100011:instruction_type=1<<3;//SUBU
    12'b000000100100:instruction_type=1<<4;//AND
    12'b000000100101:instruction_type=1<<5;//OR
    12'b000000100110:instruction_type=1<<6;//XOR
    12'b000000100111:instruction_type=1<<7;//NOR
```

```
    12'b000000101010:instruction_type=1<<8;//SLT
    12'b000000101011:instruction_type=1<<9;//SLTU
    12'b000000000000:instruction_type=1<<10;//SLL
    12'b000000000010:instruction_type=1<<11;//SRL
    12'b000000000011:instruction_type=1<<12;//SRA
    12'b000000000100:instruction_type=1<<13;//SLLV
    12'b000000000110:instruction_type=1<<14;//SRLV
    12'b000000000111:instruction_type=1<<15;//SRAV
    12'b000000001000:instruction_type=1<<16;//JR
    12'b001000??????:instruction_type=1<<17;//ADDI
    12'b001001??????:instruction_type=1<<18;//ADDIU
    12'b001100??????:instruction_type=1<<19;//ANDI
    12'b001101??????:instruction_type=1<<20;//ORI
    12'b001110??????:instruction_type=1<<21;//XORI
    12'b001111??????:instruction_type=1<<22;//LUI
    12'b100011??????:instruction_type=1<<23;//LW
    12'b101011??????:instruction_type=1<<24;//SW
    12'b000100??????:instruction_type=1<<25;//BEQ
    12'b000101??????:instruction_type=1<<26;//BNE
    12'b001010??????:instruction_type=1<<27;//SLTI
    12'b001011??????:instruction_type=1<<28;//SLTIU
    12'b000010??????:instruction_type=1<<29;//JL
    12'b000011??????:instruction_type=1<<30;//JAL
    default:;
    endcase
end

endmodule
```

8.微操作序列产生器

根据指令译码器送来的 32 位信号中第几位是高电平来确定将要执行的指令是哪一种，并从而产生上文所给出表格中的控制信号。

控制信号通过组合逻辑电路产生，其逻辑为：

①PC_CLK=1

②IM_R=1

③M1=~(jr+j+jal)，只有在执行 j、jal 指令时需要从拼接元件接受信号，只有在执行 jr 指令时 Mux1 需要直接从寄存器堆接受信号，其余情况均从 Mux5 接受信号

④M2=~lw，只有 lw 指令能够使 CPU 通过 Mux2 接受从数据存储器传来的数据

⑤M3=~(sll+srl+sra)，执行 sll、srl、sra 指令时 ALU 的第一个操作数为 shamt 立即数，

Mux3 需要接受立即数，否则 ALU 的第一个操作数为寄存器堆中的数据，Mux3 需要接受寄存器堆中数据

⑥ M4_1=addiu+addi+slti+sltiu+lw+sw

⑦ M4_2=ori+xori+andi+lui，M4(包括 M4_1 和 M4_2)是 2-4 选择器的控制信号，用以区分 ALU 的第二个操作数的来源，这个来源根据不同指令，可能是寄存器堆、无符号扩展部件 ext16、有符号扩展部件 s_ext16，通过列写真值表确定 M4 的逻辑表达式

⑧ M6_2=jal，只有执行 jal 指令时 Mux6 才接受常数 31，表示向 31 号寄存器写数据

⑨ M7=jal，只有执行 jal 指令时 Mux7 才接受 PC+4，否则均从 Mux2 接收信号

⑩ M8=jr，只有执行 jr 指令时 Mux8 才从寄存器堆直接接受信号，否则均从拼接元件接受信号

⑪ RF_W=~(jr+sw+beq+bne+j)，除 jr、sw、beq、bne、j 指令外，其余指令均需要向寄存器堆写数据

⑫ RF_CLK=~(jr+sw+beq+bne+j)clk，除 jr、sw、beq、bne、j 指令外，其余指令均需要向寄存器堆写数据

⑬ A0=sub+subu+or+nor+slt+srl+srlv+ori+beq+bne+slti

⑭ A1=add+sub+xor+nor+slt+sltu+sll+sllv+addi+slti+sltiu+xori+sll

⑮ A2=and+or+xor+nor+sll+srl+sra+sllv+srlv+srav+andi+ori+xori

⑯ A3=slt+sltu+sll+srl+sra+sllv+srlv+srav+slti+sltiu+lui，根据上文 ALUC 控制信号与所执行运算的关系确定 ALUC 的四位信号

⑰ DM_w=sw，只有 sw 指令需要向数据存储器写数据

⑱ DM_r=lw，只有 lw 指令需要从数据存储器读数据

⑲ DM_cs=lw+sw，只有 DM_cs 有效，才能对数据存储器执行读写操作，因此执行 sw 或 lw 指令时，该信号均应该有效

⑳ B1=beq，为 beq 专门设置的信号，有效时代表所执行的是 beq 指令

㉑ B2=bne，为 bne 专门设置的信号，有效时代表所执行的是 bne 指令

㉒ M5=(B1&~B2&Z)|(~B1&B2&~Z)，只有执行 beq 指令且 ALU 的 zero 标志位为 1，

或者执行 bne 指令且 ALU 的 zero 标志位为 0 时 Mux5 才从加法器接受信号，否则均从 NPC 接受信号

源代码如下：

```
module micro_operation_producer(
input clk,
input z,
input [31:0] instruction_code,
input [31:0] instruction_type,
output PC_CLK,
output IM_R,
output M1,
output M2,
output M3,
output [1:0]M4,
output [1:0]M6,
output M7,
output M8,
output [3:0]ALUC,
output RF_W,
output RF_CLK,
output DM_W,
output DM_R,
output DM_CS,
output [4:0]Rsc,
output [4:0]Rtc,
output [4:0]Rdc,
output B1,
output B2
);
//微操作序列产生部件，根据指令类型，通过组合逻辑电路生成控制信号

assign PC_CLK=clk;//PC_CLK=1
assign IM_R=1;//IM_R=1
assign M1=~(instruction_type[16]|instruction_type[29]|instruction_type[30]);//M1=~(jr+j+jal)
assign M2=~instruction_type[23];//M2=~lw
assign M3=~(instruction_type[10]|instruction_type[11]|instruction_type[12]);//M3=~(sll+srl+sra)
assign
M4[0]=instruction_type[17]|instruction_type[18]|instruction_type[27]|instruction_type[28]|instruction_type[23]|instruction_type[24];//M4_1=addiu+addi+slti+sltiu+lw+sw
assign M4[1]=instruction_type[19]|instruction_type[20]|instruction_type[21]|instruction_type[22];//M4_2=ori+xori+andi+lui
assign
M6[0]=~(instruction_type[17]|instruction_type[18]|instruction_type[19]|instruction_type[20]|instruction_type[21]|instruction_type[22]|instruction_type[23]|instruction_type[27]|instruction_type[28]|instruction_type[30]);//M6_1=~addi+addiu+andi+ori+xori+lw+slti+sltiu+lui
```

```
assign M6[1]=instruction_type[30];//M6_2=jal
assign M7=instruction_type[30];//M7=jal
assign M8=instruction_type[16];//M8=jr
assign
RF_W=~(instruction_type[16]|instruction_type[24]|instruction_type[25]|instruction_type[26]|instruction_type[29]);//RF_W=~(jr+sw+beq
+bne+j)
assign
RF_CLK=(~(instruction_type[16]|instruction_type[24]|instruction_type[25]|instruction_type[26]|instruction_type[29]))&clk;//RF_CLK=
~(jr+sw+beq+bne+j)clk
assign
ALUC[0]=instruction_type[2]|instruction_type[3]|instruction_type[5]|instruction_type[7]|instruction_type[8]|instruction_type[11]|instruct
ion_type[14]|instruction_type[20]|instruction_type[25]|instruction_type[26]|instruction_type[27];//A0=sub+subu+or+nor+slt+srl+srlv+ori
+beq+bne+slti
assign
ALUC[1]=instruction_type[0]|instruction_type[2]|instruction_type[6]|instruction_type[7]|instruction_type[8]|instruction_type[9]|instructi
on_type[10]|instruction_type[13]|instruction_type[17]|instruction_type[21]|instruction_type[27]|instruction_type[28];//A1=add+sub+xor+
nor+slt+sltu+sll+sllv+addi+slti+sltiu+xori+sll
assign
ALUC[2]=instruction_type[4]|instruction_type[5]|instruction_type[6]|instruction_type[7]|instruction_type[10]|instruction_type[11]|instruc
tion_type[12]|instruction_type[13]|instruction_type[14]|instruction_type[15]|instruction_type[19]|instruction_type[20]|instruction_type[21]
;//A2=and+or+xor+nor+sll+srl+sra+sllv+srlv+srav+andi+ori+xori
assign
ALUC[3]=instruction_type[8]|instruction_type[9]|instruction_type[10]|instruction_type[11]|instruction_type[12]|instruction_type[13]|inst
ruction_type[14]|instruction_type[15]|instruction_type[22]|instruction_type[27]|instruction_type[28];//A3=slt+sltu+sll+srl+sra+sllv+srlv+
srav+slti+sltiu+lui
assign DM_W=instruction_type[24];//DM_w=sw
assign DM_R=instruction_type[23];//DM_r=lw
assign DM_CS=instruction_type[23]|instruction_type[24];//DM_cs=lw+sw
assign Rsc=instruction_code[25:21];
assign Rtc=instruction_code[20:16];
assign
Rdc=(instruction_type[17]|instruction_type[18]|instruction_type[19]|instruction_type[20]|instruction_type[21]|instruction_type[22]|instru
ction_type[27]|instruction_type[28])?instruction_code[20:16]:instruction_code[15:11];
assign B1=instruction_type[25];
assign B2=instruction_type[26];
endmodule
```

9.加 4 加法器

将 PC 寄存器的内容+4 并送给 Mux7。该模块仅仅配合 jr 指令使用，将 PC 寄存器的内容+4 后写给 31 号寄存器。

源代码如下：

```
module add4(
```

```
input [31:0] a,
output [31:0] r
);
assign r=a+4;//(npc，实质是+4 加法器)
endmodule
```

10.加法器

将 NPC 中的内容(PC+4)与扩展后的偏移地址相加得到跳转地址所用的部件。该模块配合 beq、bne 指令使用。

源代码如下：

```
module add(
input [31:0] a,
input [31:0] b,
output [31:0] r,
output o
);
//无符号数加法器，用于计算地址
wire[32:0] c;
assign c={a[31],a}+{b[31],b};//使用双符号位方便判断溢出
assign r=a+b;
assign o=(c[32]==c[31])?0:1;//判断是否溢出

endmodule
```

11.有符号扩展部件 s_ext16

将指令中的 16 位立即数左侧填充 0 扩展成 32 位数据。

源代码如下：

```
module s_ext16(input [15:0] a,output [31:0] b);
assign b = {{16{a[15]}},a};//将 16 位数据高位填符号位扩展成 32 位
endmodule
```

12.无符号扩展部件 ext16

将指令中的 16 位立即数左侧填充最高位成 32 位数据。

源代码如下：

```
module ext16(input [15:0] a,output [31:0] b);
assign b = {{16{1'b0}},a};//将 16 位数据高位填 0 扩展成 32 位
endmodule
```

13.无符号扩展部件 ext5

将指令中的 5 位立即数左侧填充 0 成 32 位数据。

源代码如下：

```
module ext5(input [4:0] a,output [31:0] b);
assign b = {{27{1'b0}},a};//将 5 位数据高位填 0 扩展成 32 位
endmodule
```

14.有符号扩展部件 ext18

将指令中的 16 位立即数左移两位后得到的 18 位数据左侧填充最高位成 32 位数据。用于 beq 和 bne 指令。

源代码如下：

```
module ext18(input [17:0] a,output [31:0] b);
assign b = {{14{a[17]}},a};//将 18 位数据高位填符号位扩展成 32 位
endmodule
```

15.拼接部件

将 PC 寄存器中内容的 31~28 位和 26 位立即数左移得到的 28 位数据拼接起来得到 32 位数据。用于 jr、jal 指令。

源代码如下：

```
module bing(
input [3:0] a,
input [25:0] b,
output [31:0] c
);

assign c = {a, b,2'b00};//用于将 PC 的 31~28 位和左移两位后的 instruct_index 合并成 32 位地址

endmodule
```

16.selector 部件

进行 M5=(B1&~B2&Z)|(~B1&B2&~Z)的逻辑运算，由于需要使用 ALU 的 zero 位，所以不能在微操作序列产生器中得到。

源代码如下：

```
module selector(
input b1,
input b2,
```

```
input z,
output o
);

assign o=(b1&(~b2)&z)|((~b1)&b2&(~z));

endmodule
```

17.数据选择器

包括 1-2 选择器和 2-4 选择器，用于在 M1~M8 信号控制下对数据通路中的数据进行选择性传送。

源代码如下：

```
module mux_1_2(
input [31:0] a,
input [31:0] b,
input m,
output [31:0] c
);

assign c=(m==1)?b:a;//1-2 路选择器

endmodule

module mux_2_4(
input [31:0] a,
input [31:0] b,
input [31:0] c,
input [1:0] m,
output reg [31:0] e
);

always @(m or a or b or c)
begin
   case (m)//2-4 路选择器
      2'b00:e<=a;
      2'b01:e<=b;
      2'b10:e<=c;
   endcase
end

endmodule
```

18.CPU 整体

将所用到的 CPU 内部部件按照数据通路图和信号关系连接起来构成除指令存储器和数据存储器外的 CPU 整体，用以进行指令的主体执行部分。

源代码如下:

```
module cpu(
input clk,
input reset,
input [31:0]instruction,
input [31:0]rdata,
output [31:0]pc,
output [31:0]addr,
output [31:0]wdata,
output IM_R,
output DM_CS,
output DM_R,
output DM_W
);

wire PC_CLK,PC_ENA,M1,M2,M3,M5,M7,M8,RF_W,RF_CLK,B1,B2;
wire [1:0] M4;
wire [1:0] M6;
wire [3:0] ALUC;
wire z,c,n,o,a;
wire [31:0] INS;
wire [31:0] D_ALU;
wire [31:0] D_PC;
wire [31:0] D_NPC;
wire [31:0] D_RF;
wire [31:0] D_Rs;
wire [31:0] D_Rt;
wire [31:0] D_IM;
wire [31:0] D_DM;
wire [31:0] D_Mux1;
wire [31:0] D_Mux2;
wire [31:0] D_Mux3;
wire [31:0] D_Mux4;
wire [31:0] D_Mux5;
wire [4:0] D_Mux6;
wire [31:0] D_Mux7;
wire [31:0] D_Mux8;
wire [31:0] D_EXT5;
wire [31:0] D_EXT16;
wire [31:0] D_S_EXT16;
```

```
wire [31:0] D_EXT18;
wire [31:0] D_ADD;
wire [31:0] D_ADD8;
wire [31:0] D_ADD4;
wire [31:0] D_BING;
wire [4:0] D_Rsc;
wire [4:0] D_Rtc;
wire [4:0] D_Rdc;

assign PC_ENA = 1;
assign pc=D_PC;
assign addr=D_ALU;
assign wdata=D_Rt;

instruction_decoder cpu_ins(instruction,INS);
micro_operation_producer
cpu_opcode(clk,z,instruction,INS,PC_CLK,IM_R,M1,M2,M3,M4,M6,M7,M8,ALUC,RF_W,RF_CLK,DM_W,DM_R,DM_CS,D_Rsc,D
_Rtc,D_Rdc,B1,B2);
pcreg2 pc_out(PC_CLK,reset,PC_ENA,D_Mux1,D_PC);
alu cpu_alu(D_Mux3,D_Mux4,ALUC[3:0],D_ALU,z,c,n,o);
regfile cpu_ref(~RF_CLK,reset,RF_W,D_Rtc,D_Rsc,D_Mux6,D_Mux7,D_Rt,D_Rs);
mux_1_2 cpu_mux1(D_Mux8,D_Mux5,M1,D_Mux1);
mux_1_2 cpu_mux2(rdata,D_ALU,M2,D_Mux2);
mux_1_2 cpu_mux3(D_EXT5,D_Rs,M3,D_Mux3);
mux_2_4 cpu_mux4(D_Rt,D_S_EXT16,D_EXT16,M4,D_Mux4);
selector sel(B1,B2,cpu_alu.z,M5);
mux_1_2 cpu_mux5(D_NPC,D_ADD,M5,D_Mux5);
mux_2_4 cpu_mux6(D_Rtc,D_Rdc,5'b11111,M6,D_Mux6);
mux_1_2 cpu_mux7(D_Mux2,D_ADD8,M7,D_Mux7);
mux_1_2 cpu_mux8(D_BING,D_Rs,M8,D_Mux8);
ext5 cpu_ext5(instruction[10:6],D_EXT5);
ext16 cpu_ext16(instruction[15:0],D_EXT16);
s_ext16 cpu_s_ext16(instruction[15:0],D_S_EXT16);
ext18 cpu_ext18(instruction[15:0]<<2,D_EXT18);
add cpu_add(D_EXT18,D_NPC,D_ADD,ao);
add8 cpu_add8(D_PC,D_ADD8);
add4 cpu_npc(D_PC,D_NPC);
bing cpu_bing(D_PC[31:28],instruction[25:0],D_BING);

endmodule
```

19.顶层模块

根据 CPU 整体以及指令存储器和数据存储器之间的数据流通关系，将 CPU 主体和指令存储器、数据存储器连接起来，构成完整的 CPU 通路。

源代码如下：

```
module sccomp_dataflow(
input clk_in,
input reset,
output [31:0] inst,
output [31:0] pc
);

wire [10:0] addr;
wire [31:0] wdata;
wire [31:0] rdata;
wire [31:0] ip_in;

assign   ip_in=pc-32'h00400000;
cpu sccpu(clk_in,reset,inst,rdata,pc,addr,wdata,IM_R,DM_CS,DM_R,DM_W);
imem im(ip_in[12:2],inst);
//IMEM im(clk_in,pc[12:2],IM_R,inst);
DMEM dm(clk_in,addr-32'h10010000,DM_CS,DM_R,DM_W,wdata,rdata);

endmodule
```

四、测试/调试过程

1.测试方法

①依次将含有要测试的 31 条指令的汇编程序使用 Mars 导出 16 进制的指令序列文件

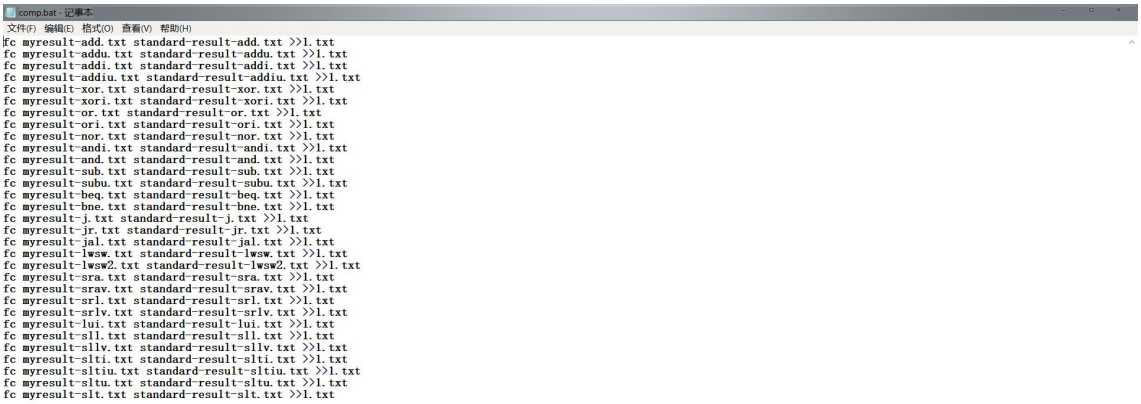
②使用 Mars 依次运行含有要测试的 31 条指令的汇编程序，得到 result.txt 文件，重命名为 standard-result-xxx.txt，其中记录了每条指令执行完毕后 PC 寄存器、所执行指令、寄存器堆中 32 个寄存器的内容

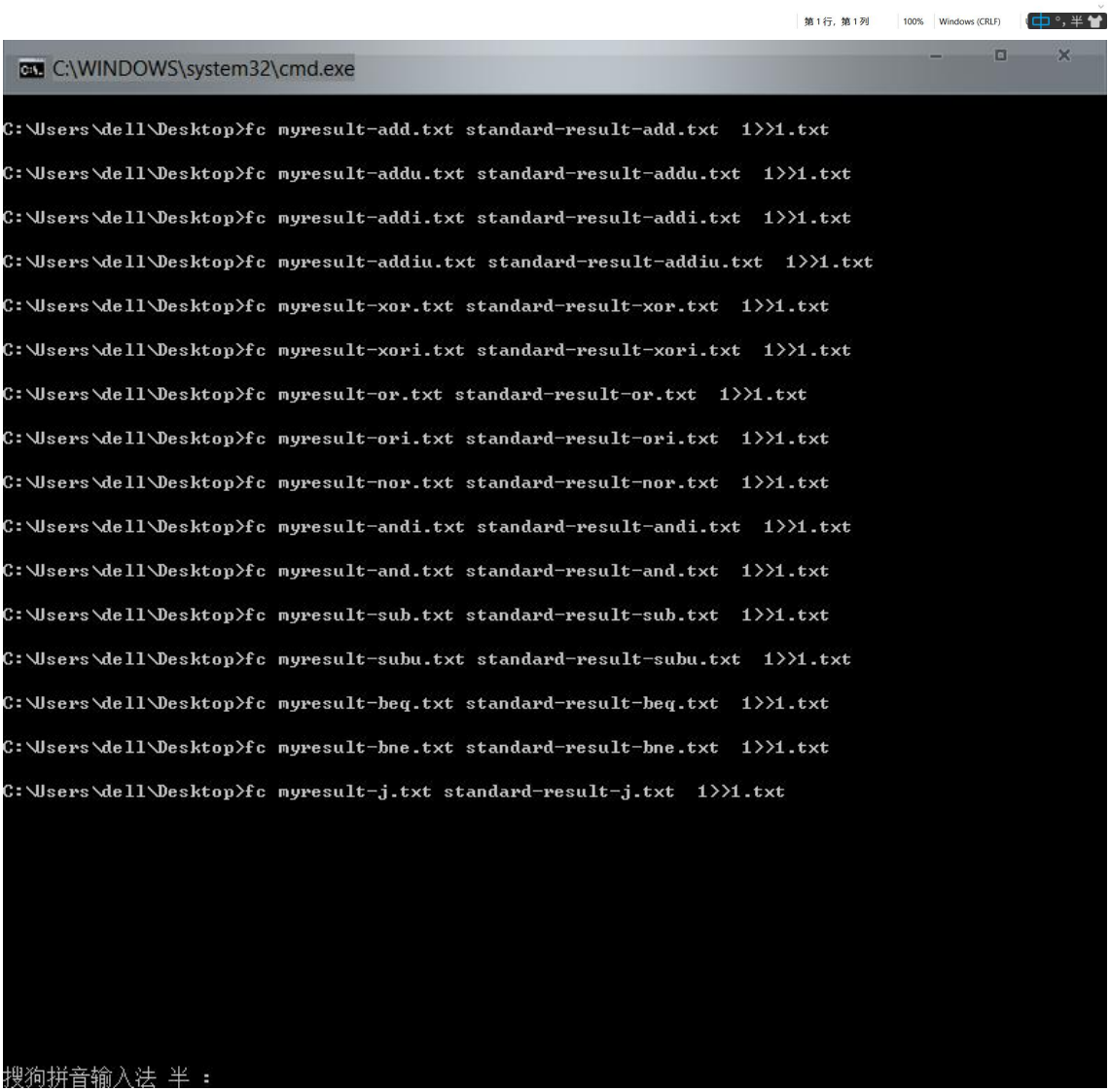
③使用 Mars 导出的 16 进制指令序列文件初始化指令存储器，在 vivado 环境下进行前仿真，每执行完一条指令将 PC 寄存器、所执行指令、寄存器堆中 32 个寄存器的内容输出到 myresult-xxx.txt 文件中(xxx 为测试的指令)

④将 myresult-xxx.txt 和 result.txt 依次对比，观察运行结果是否正确

通过对比 myresult-xxx.txt 和对应的 standard-result-xxx.txt 来检查并分析实验结果：

使用 windows10 的 fc 命令，编写 bat 文件，将每次比较的结果输出到 1.txt 中，从而一次性得知所有比较结果。


comp.bat - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
fc myresult-add.txt standard-result-add.txt >>1.txt
fc myresult-addu.txt standard-result-addu.txt >>1.txt
fc myresult-addi.txt standard-result-addi.txt >>1.txt
fc myresult-addiu.txt standard-result-addiu.txt >>1.txt
fc myresult-xor.txt standard-result-xor.txt >>1.txt
fc myresult-xori.txt standard-result-xori.txt >>1.txt
fc myresult-or.txt standard-result-or.txt >>1.txt
fc myresult-ori.txt standard-result-ori.txt >>1.txt
fc myresult-nor.txt standard-result-nor.txt >>1.txt
fc myresult-andi.txt standard-result-andi.txt >>1.txt
fc myresult-and.txt standard-result-and.txt >>1.txt
fc myresult-sub.txt standard-result-sub.txt >>1.txt
fc myresult-subu.txt standard-result-subu.txt >>1.txt
fc myresult-beq.txt standard-result-beq.txt >>1.txt
fc myresult-bne.txt standard-result-bne.txt >>1.txt
fc myresult-j.txt standard-result-j.txt >>1.txt
fc myresult-jr.txt standard-result-jr.txt >>1.txt
fc myresult-jal.txt standard-result-jal.txt >>1.txt
fc myresult-lwsw.txt standard-result-lwsw.txt >>1.txt
fc myresult-lwsw2.txt standard-result-lwsw2.txt >>1.txt
fc myresult-sra.txt standard-result-sra.txt >>1.txt
fc myresult-srav.txt standard-result-srav.txt >>1.txt
fc myresult-srl.txt standard-result-srl.txt >>1.txt
fc myresult-srlv.txt standard-result-srlv.txt >>1.txt
fc myresult-lui.txt standard-result-lui.txt >>1.txt
fc myresult-sll.txt standard-result-sll.txt >>1.txt
fc myresult-sllv.txt standard-result-sllv.txt >>1.txt
fc myresult-slti.txt standard-result-slti.txt >>1.txt
fc myresult-sltiu.txt standard-result-sltiu.txt >>1.txt
fc myresult-sltu.txt standard-result-sltu.txt >>1.txt
fc myresult-slt.txt standard-result-slt.txt >>1.txt

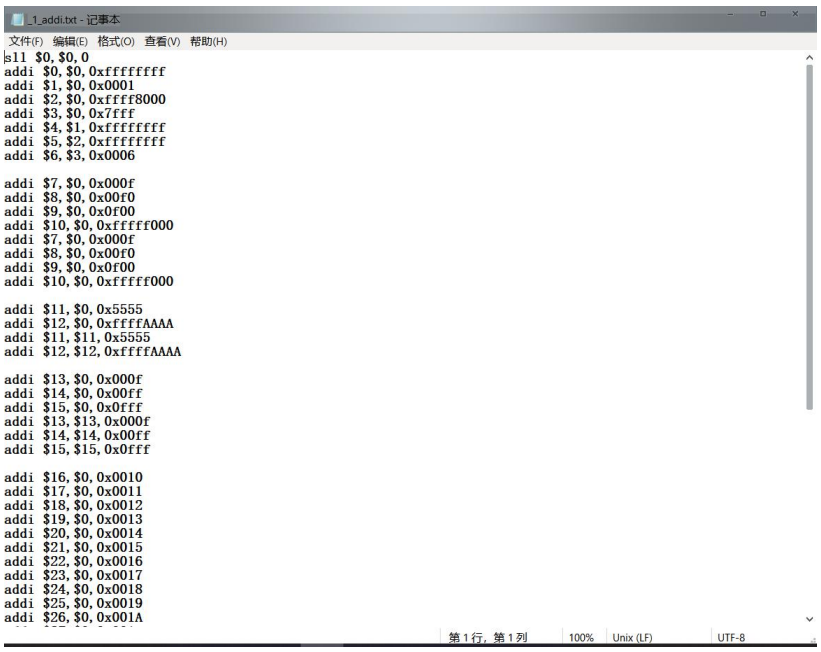

第 1 行，第 1 列
100%
Windows (CRLF)
C:\WINDOWS\system32\cmd.exe
C:\Users\dell\Desktop>fc myresult-add.txt standard-result-add.txt 1>>1.txt
C:\Users\dell\Desktop>fc myresult-addu.txt standard-result-addu.txt 1>>1.txt
C:\Users\dell\Desktop>fc myresult-addi.txt standard-result-addi.txt 1>>1.txt
C:\Users\dell\Desktop>fc myresult-addiu.txt standard-result-addiu.txt 1>>1.txt
C:\Users\dell\Desktop>fc myresult-xor.txt standard-result-xor.txt 1>>1.txt
C:\Users\dell\Desktop>fc myresult-xori.txt standard-result-xori.txt 1>>1.txt
C:\Users\dell\Desktop>fc myresult-or.txt standard-result-or.txt 1>>1.txt
C:\Users\dell\Desktop>fc myresult-ori.txt standard-result-ori.txt 1>>1.txt
C:\Users\dell\Desktop>fc myresult-nor.txt standard-result-nor.txt 1>>1.txt
C:\Users\dell\Desktop>fc myresult-andi.txt standard-result-andi.txt 1>>1.txt
C:\Users\dell\Desktop>fc myresult-and.txt standard-result-and.txt 1>>1.txt
C:\Users\dell\Desktop>fc myresult-sub.txt standard-result-sub.txt 1>>1.txt
C:\Users\dell\Desktop>fc myresult-subu.txt standard-result-subu.txt 1>>1.txt
C:\Users\dell\Desktop>fc myresult-beq.txt standard-result-beq.txt 1>>1.txt
C:\Users\dell\Desktop>fc myresult-bne.txt standard-result-bne.txt 1>>1.txt
C:\Users\dell\Desktop>fc myresult-j.txt standard-result-j.txt 1>>1.txt
搜狗拼音输入法 半 ：

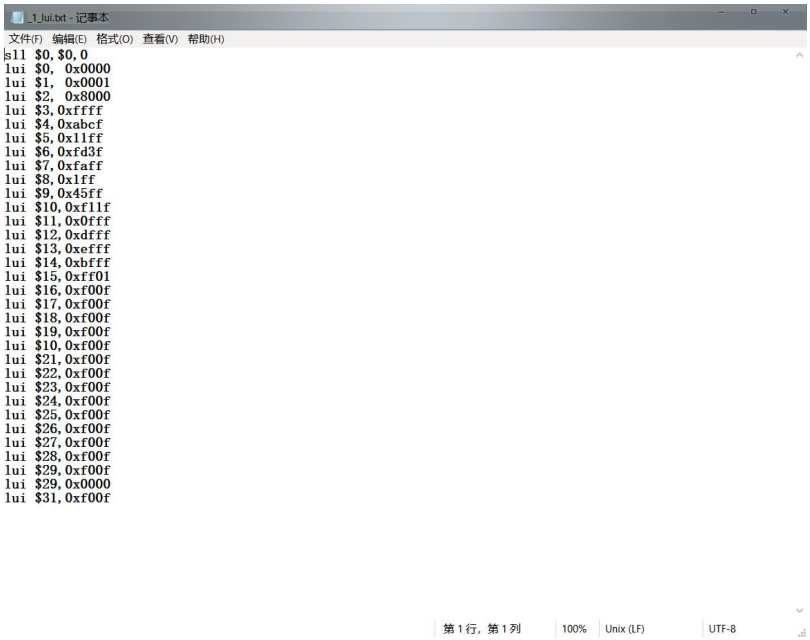
(运行过程中的截图)

2.测试种类：

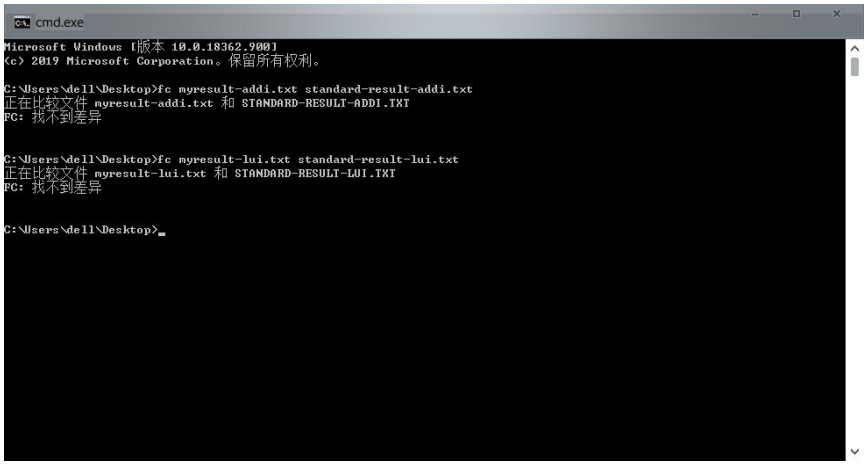
①单条指令的测试，依次测试 31 条指令执行的正确性，从而测试所设计的 CPU 的正确性

单条指令包括 addi、lui 等单一指令构成的程序，通过测试只含有对应种指令的程序来检测 CPU 执行该指令时的正确性，程序如下所示：



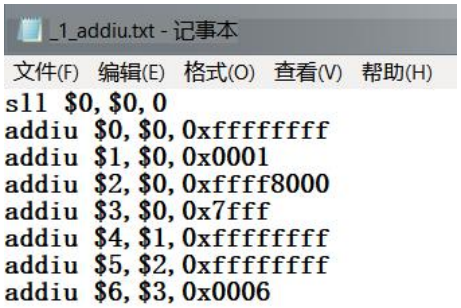


通过 Mars 将汇编代码转换成机器码，用设计的 CPU 运行机器码，将输出结果与 Mars 运行汇编代码的结果比较，发现均一致，说明所设计的 CPU 在执行单条指令时完全正确。



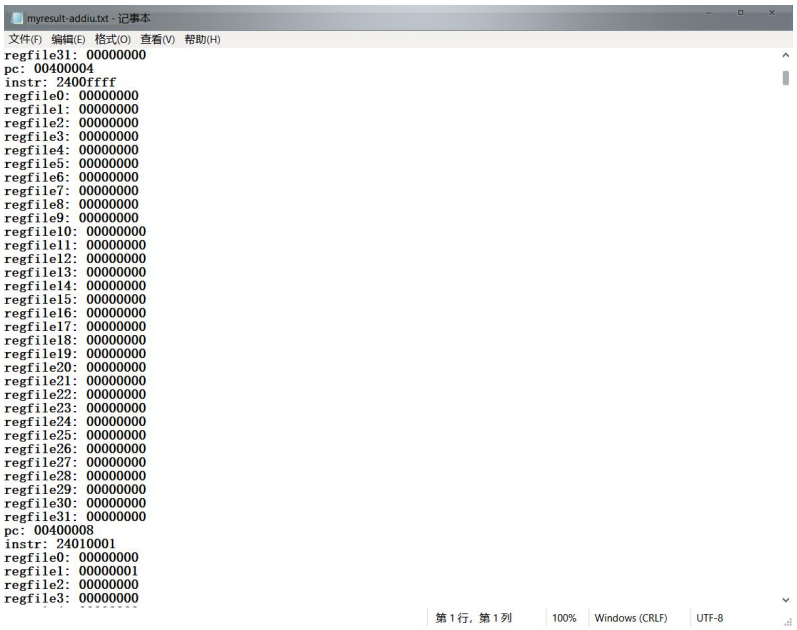
②指令的边界数据测试，测试所设计的 CPU 的完备性

在 MIPS 架构的 CPU 中，0 号寄存器始终返回 0，其寄存器值不能被修改。通过测试与 0 号寄存器相关的边界数据，可以测试所设计的 CPU 的完备性。



如上的一小段汇编代码，其第二条指令 addiu $0 $0 0xffffffff 将 0 号寄存器的值与 0xffffffff 相加后再存回 0 号寄存器，但由于 0 号寄存器始终返回 0，其寄存器值不能被

修改，因此执行完该条指令后 0 号寄存器的值依然为 0。通过观察 CPU 运行该条指令之后输出的寄存器信息可以验证所设计的 CPU 的完备性。

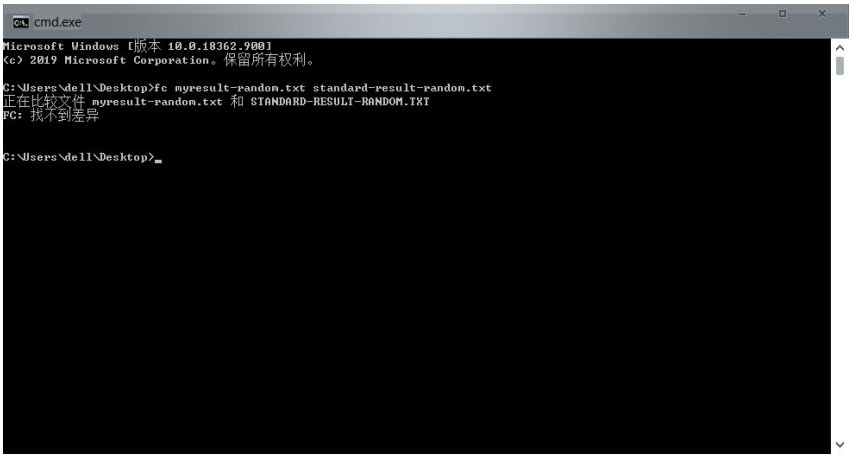


如上图，执行第二条指令 addiu $0 $0 0xffffffff 后 0 号寄存器的值仍然为 0，CPU 正确处理了边界数据，所设计的 CPU 完备。

③随机序列测试，自行编写汇编语句，使用上文所述测试方法，验证正确性

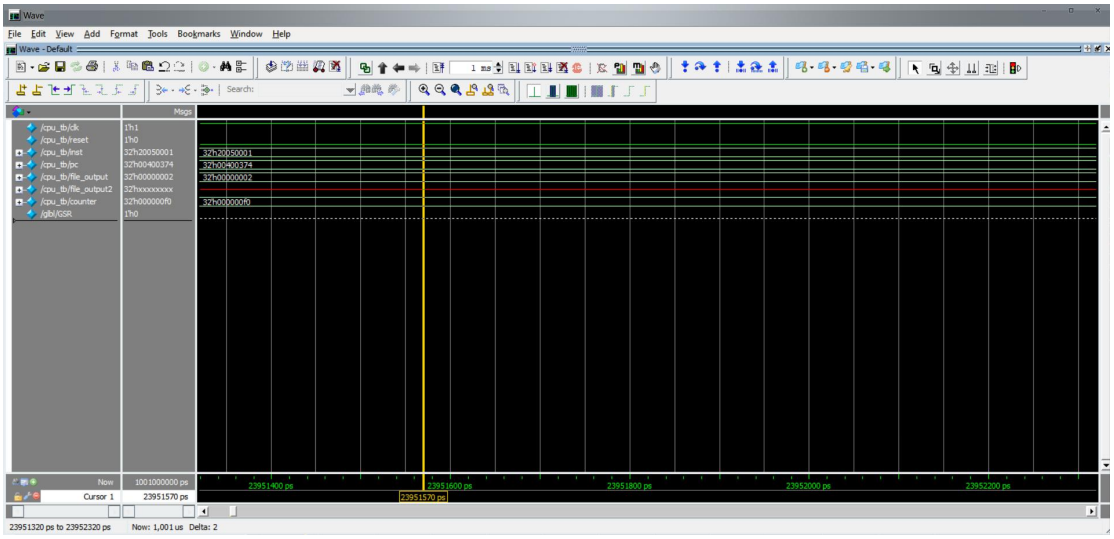
编写如下汇编语句：

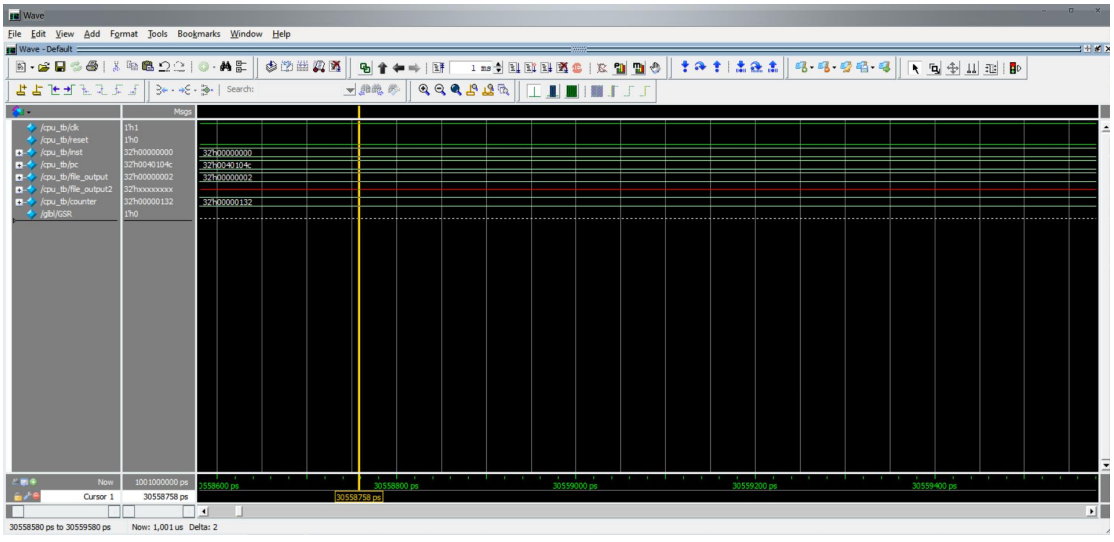
通过 Mars 将如上汇编代码转换成机器码，用设计的 CPU 运行机器码，将输出结果与 Mars 运行汇编代码的结果比较，发现一致，说明所设计的 CPU 在执行随机指令序列时正确。



五、实验结果

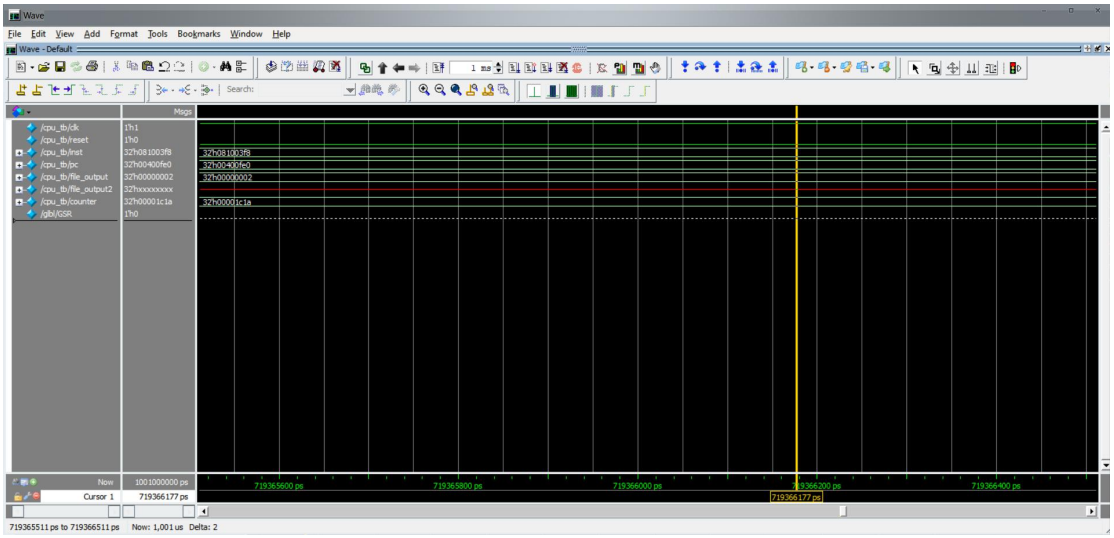
通过 mips_31_mars_simulate.coe 文件初始化使用 IP 核的指令寄存器，使用 Vivado 和 Modelsim 联合仿真 1000us，可以观察到如下的波形图(整个仿真过程较长，仅截取部分片段)。

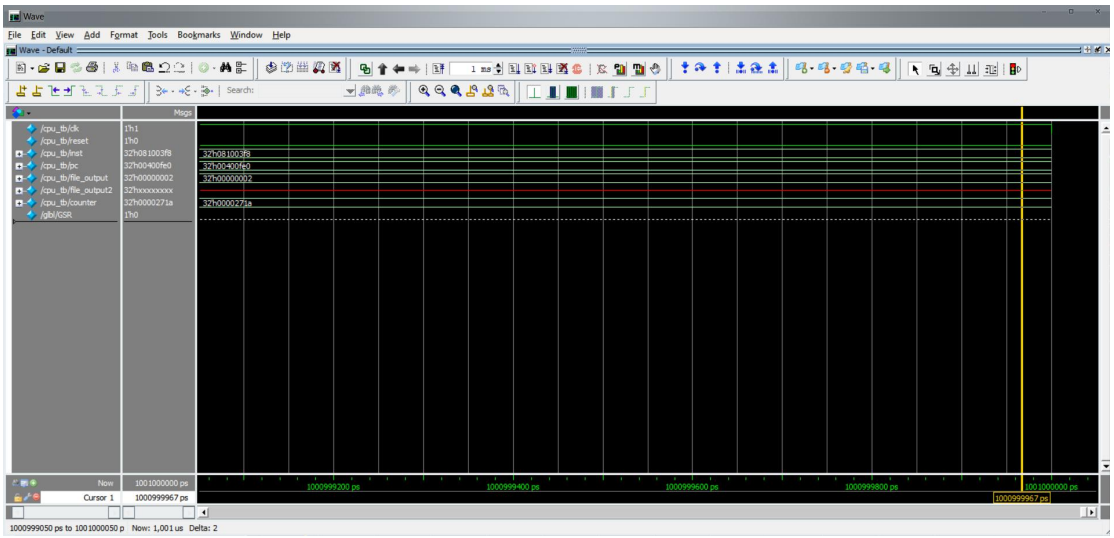




在仿真的最后会发现 PC 寄存器始终为 0x00400fe0，说明 CPU 所执行的程序陷入了死循环之中，这与 COE 文件的程序相一致，说明 CPU 正确执行 COE 文件所代表的的程序并输出了正确结果。

死循环部分的仿真图如下所示：





由上可知，所有指令均能正确执行，所设计的 CPU 正确无误。