同济大学计算机系

计算机组成原理实验报告

学　　号 1952650

姓　　名 陈子翔

专　　业 信息安全

授课老师 郝老师

一、实验内容

使用 Verilog HDL 语言实现 31 条 MIPS 指令的 CPU 的设计和仿真，使其实现以下指令。

助记符	指令格式						示例	示例含义	操作及其解释
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	00000	100000	add $1,$2,$3	$1=$2+$3	rd <- rs + rt ； 其中rs = $2， rt=$3, rd=$1
addu	000000	rs	rt	rd	00000	100001	addu $1,$2,$3	$1=$2+$3	rd <- rs + rt ； 其中rs = $2， rt=$3, rd=$1,无符号数
sub	000000	rs	rt	rd	00000	100010	sub $1,$2,$3	$1=$2-$3	rd <- rs - rt ； 其中rs = $2， rt=$3, rd=$1
subu	000000	rs	rt	rd	00000	100011	subu $1,$2,$3	$1=$2-$3	rd <- rs - rt ； 其中rs = $2， rt=$3, rd=$1,无符号数
and	000000	rs	rt	rd	00000	100100	and $1,$2,$3	$1=$2 & $3	rd <- rs & rt ； 其中rs = $2， rt=$3, rd=$1
or	000000	rs	rt	rd	00000	100101	or $1,$2,$3	$1=$2 | $3	rd <- rs | rt ； 其中rs = $2， rt=$3, rd=$1
xor	000000	rs	rt	rd	00000	100110	xor $1,$2,$3	$1=$2 ^ $3	rd <- rs xor rt ； 其中rs = $2， rt=$3, rd=$1(异或)
nor	000000	rs	rt	rd	00000	100111	nor $1,$2,$3	$1=~($2 | $3)	rd <- not(rs | rt) ； 其中rs = $2， rt=$3, rd=$1(或非)
slt	000000	rs	rt	rd	00000	101010	slt $1,$2,$3	if($2<$3) $1=1 else $1=0	if (rs < rt) rd=1 else rd=0 ；其中rs = $2， rt=$3, rd=$1
sltu	000000	rs	rt	rd	00000	101011	sltu $1,$2,$3	if($2<$3) $1=1 else $1=0	if (rs < rt) rd=1 else rd=0 ；其中rs = $2， rt=$3, rd=$1 (无符号数)
sll	000000	00000	rt	rd	shamt	000000	sll $1,$2,10	$1=$2<<10	rd <- rt << shamt ； shamt存放移位的位数， 也就是指令中的立即数， 其中rt=$2, rd=$1
srl	000000	00000	rt	rd	shamt	000010	srl $1,$2,10	$1=$2>>10	rd <- rt >> shamt ； (logical)， 其中rt=$2, rd=$1
sra	000000	00000	rt	rd	shamt	000011	sra $1,$2,10	$1=$2>>10	rd <- rt >> shamt ； (arithmetic) 注意符号位保留 其中rt=$2, rd=$1
sllv	000000	rs	rt	rd	00000	000100	sllv $1,$2,$3	$1=$2<<$3	rd <- rt << rs ； 其中rs = $3， rt=$2, rd=$1
srlv	000000	rs	rt	rd	00000	000110	srlv $1,$2,$3	$1=$2>>$3	rd <- rt >> rs ； (logical)其中rs = $3， rt=$2, rd=$1
srav	000000	rs	rt	rd	00000	000111	srav $1,$2,$3	$1=$2>>$3	rd <- rt >> rs ； (arithmetic) 注意符号位保留 其中rs = $3， rt=$2, rd=$1
jr	000000	rs	00000	00000	00000	001000	jr $31	goto $31	PC <- rs

I-type	op	rs	rt	immediate			
addi	001000	rs	rt	immediate	addi $1,$2,100	$1=$2+100	rt <- rs + (sign-extend)immediate ； 其中rt=$1,rs=$2
addiu	001001	rs	rt	immediate	addiu $1,$2,100	$1=$2+100	rt <- rs + (zero-extend)immediate ； 其中rt=$1,rs=$2
andi	001100	rs	rt	immediate	andi $1,$2,10	$1=$2 & 10	rt <- rs & (zero-extend)immediate ； 其中rt=$1,rs=$2
ori	001101	rs	rt	immediate	andi $1,$2,10	$1=$2 | 10	rt <- rs | (zero-extend)immediate ； 其中rt=$1,rs=$2
xori	001110	rs	rt	immediate	andi $1,$2,10	$1=$2 ^ 10	rt <- rs xor (zero-extend)immediate ； 其中rt=$1,rs=$2
lui	001111	00000	rt	immediate	lui $1,100	$1=100*65536	rt <- immediate*65536 ； 将16位立即数放到目标寄存器高16 位， 目标寄存器的低16位填0
lw	100011	rs	rt	immediate	lw $1,10($2)	$1=memory[$2+10]	rt <- memory[rs + (sign-extend)immediate] ； rt=$1,rs=$2
sw	101011	rs	rt	immediate	sw $1,10($2)	memory[$2+10]=$1	memory[rs + (sign-extend)immediate] <- rt ； rt=$1,rs=$2
beq	000100	rs	rt	immediate	beq $1,$2,10	if($1==$2) goto PC+4+40	if (rs == rt) PC <- PC+4 + (sign-extend)immediate<<2
bne	000101	rs	rt	immediate	bne $1,$2,10	if($1!=$2) goto PC+4+40	if (rs != rt) PC <- PC+4 + (sign-extend)immediate<<2
slti	001010	rs	rt	immediate	slti $1,$2,10	if($2<10) $1=1 else $1=0	if (rs <(sign-extend)immediate) rt=1 else rt=0 ； 其中rs = $2， rt=$1
sltiu	001011	rs	rt	immediate	sltiu $1,$2,10	if($2<10) $1=1 else $1=0	if (rs <(zero-extend)immediate) rt=1 else rt=0 ； 其中rs = $2， rt=$1
J-type	op	address					
j	000010	address			j 10000	goto 10000	PC <- (PC+4)[31..28],address,0,0 ； address=10000/4
jal	000011	address			jal 10000	$31<-PC+4; goto 10000	$31<-PC+4； PC <- (PC+4)[31..28],address,0,0 ； address=10000/4

二、实验目的

1. 深入掌握 CPU 的构成及工作原理。
2. 设计 31 条指令的 CPU 的数据通路及控制器。
3. 使用 Verilog HDL 设计实现 31 条指令的 CPU 下板运行。

三、实验原理

3.1 实验原理

中央处理器（CPU）由运算器和控制器组成。其中，控制器的功能是负责协调并控制计算机各部件执行程序的指令序列，包括取指令、分析指令和执行指令；运算器的功能是对数据进行加工。CPU 的具体功能包括：

1）指令控制。完成取指令、分析指令和执行指令的操作，即程序的顺序控制。

2）操作控制。一条指令的功能往往由若干操作信号的组合来实现。CPU 管理并产生由内存取出的每条指令的操作信号，把各种操作信号送往相应的部件，

从而控制这些部件按指令的要求进行动作。

3）时间控制。对各种操作加以时间上的控制。时间控制要为每条指令按时间顺序提供应有的控制信号。

4）数据加工。对数据进行算术和逻辑运算。

5）中断处理。对计算机运行过程中出现的异常情况和特殊请求进行处理。

单周期 CPU 是对所有指令都选用相同的执行时间来完成，称为单指令周期方案。此时每条指令都在固定的时钟周期内完成，指令之间串行执行，即下一条指令只能在前一条指令执行结束后才能启动。在单周期处理器中，一条指令执行过程中数据通路的任何资源都不能被重复使用，因此，任何需要被多次使用的资源都需要设置多个。

CPU 在执行指令的不同阶段所涉及的数据流有所不同，可分为以下几个步骤：

（1） 取指周期：根据程序计数器 PC 中的内容从主存中取出一条指令并存放在指令寄存器 IR 中。PC 存放的是指令的地址，根据此地址从内存单元中取出的是指令，取指令的同时，PC 通过自增到存储下一条指令的地址。

（2） 指令译码：对在取指周期中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。

（3） 间址周期：取操作数有效地址，将指令中的地址码送到地址寄存器 MAR 并送至地址总线，此后控制单元 CU 向存储器发读命令，以获取有效地址并存至数据寄存器 MDR。

（4） 执行周期：根据指令寄存器 IR 中的指令字的操作码核操作数通过算数逻辑单元 ALU 操作产生执行结果。最后根据指令要求将执行结果或访问存储器得到的数据写回相应的寄存器。

3.2 指令格式

一条指令就是机器语言的一个语句，它是一组有意义的二进制代码。一条指令通常包括操作码字段和地址码字段两部分：

操作码字段	地址码字段

其中，操作码字段指出指令中该指令应该执行什么性质的操作和具有何种功能。操作码是识别指令、了解指令功能及区分操作数地址内容的组成和使用方法等的关键信息。

MIPS 框架下的 CPU 具有以下三种指令类型：

1．R-type

Bit #	31..26	25..21	20..16	15..11	10..6	5..0	
R-type	op	rs	rt	rd	shamt	func	

2．I-type

Bit #	31..26	25..21	20..16	15..0	
I-type	op	rs	rt	immediate	

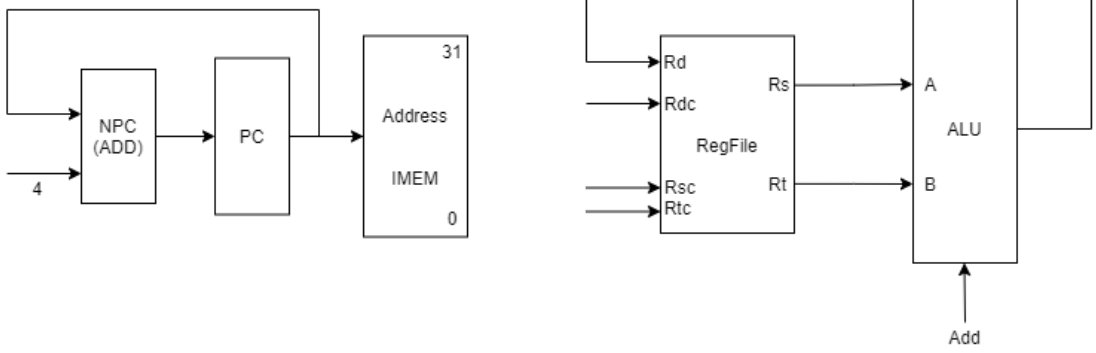
3．J-type

Bit #	31..26	25..0	
J-type	op	Index	

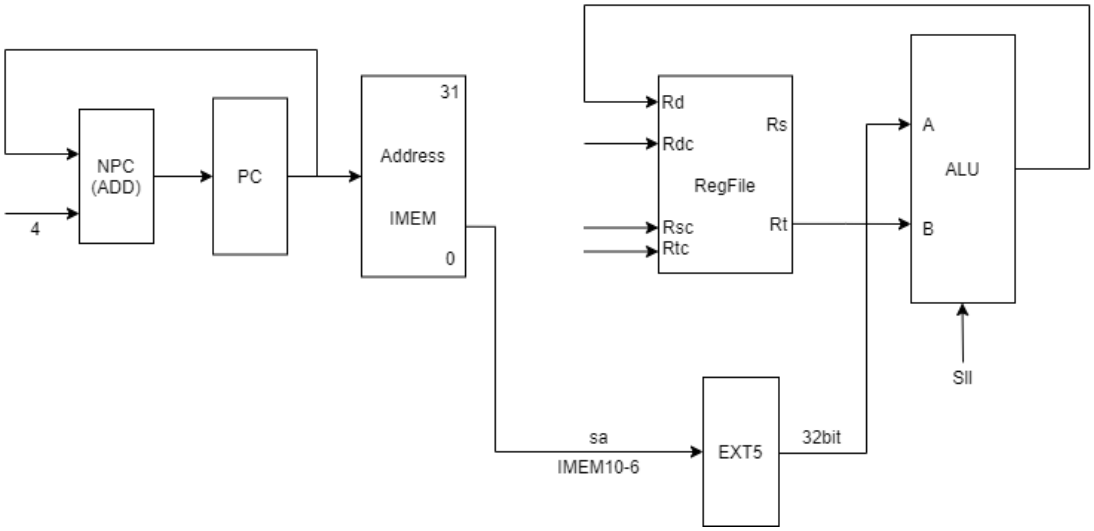
其中：
op:31 位~26 位,6 位操作码,决定执行何种指令,R 型指令的操作码均为 000000，其余指令操作码各不相同。
rs：5 位地址码，代表第一个源操作数地址，只读。
rt：5 位地址码，代表第二个源操作数地址，可读可写。
rd：5 位地址码，代表目的操作数地址，只写。
shamt：5 位数据，专用于移位指令中代表位移量。
func：6 位功能码，与操作码配合使用，在 R 类指令中，操作码均为 0，用不同的功能码区分执行何种指令。
immediate：16 位立即数，用作无符号的逻辑操作数、有符号的算数操作数、数据加载/数据保存指令的数据地址字节偏移量和分支指令中相对程序计数器的有符号偏移量。
index：26 位地址码，用于 J 类指令中作 pc 跳转的地址。

3.3 数据通路设计

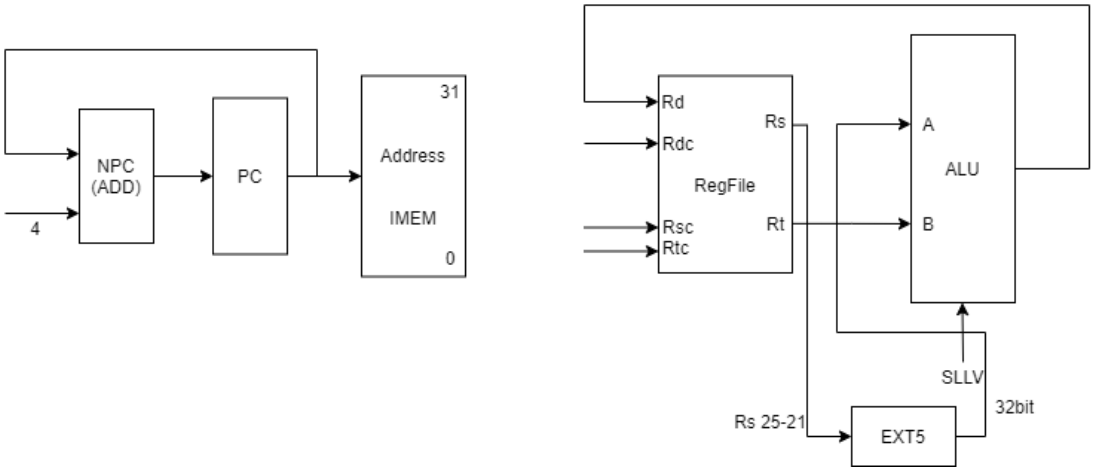
对于相似的指令，相同的数据通路图不再重复画出。
1. ADD/ADDU/SUB/SUBU/AND/OR/XOR/NOR/SLT/SLTU



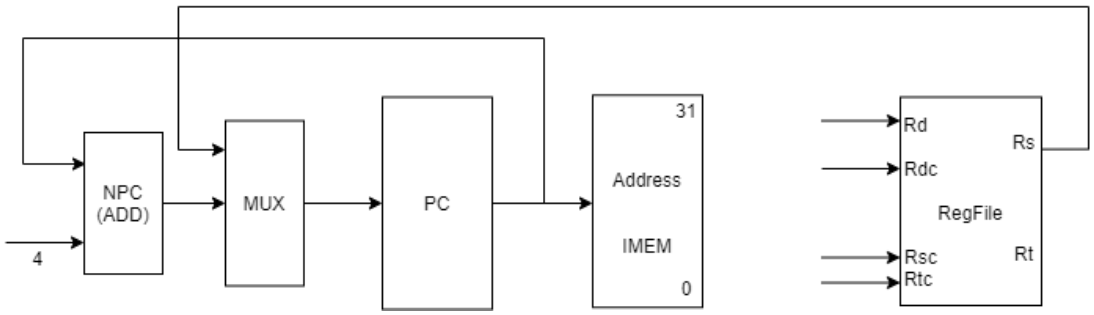
2. SLL/SRL/SRA



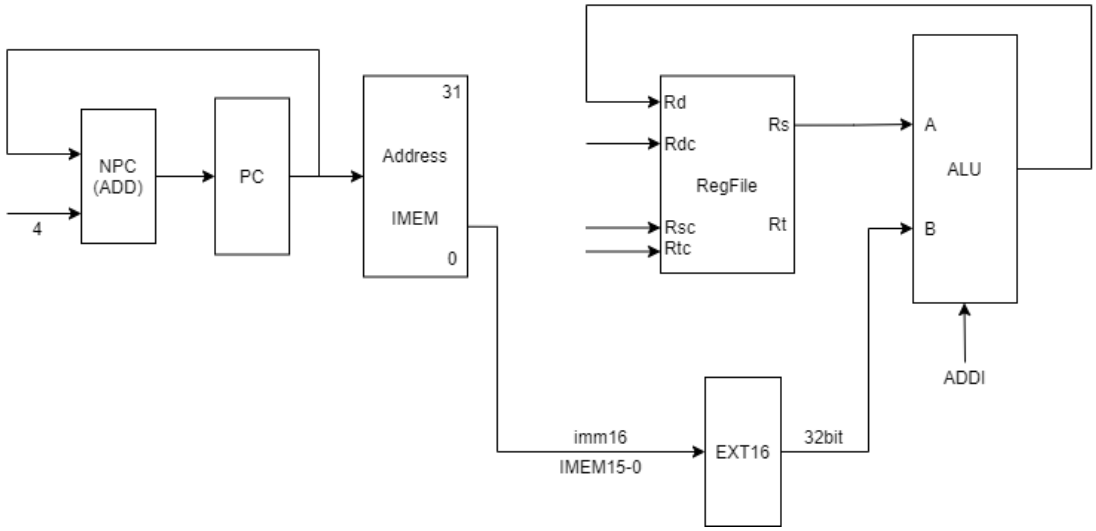
3. SLLV/SRLV/SRAV



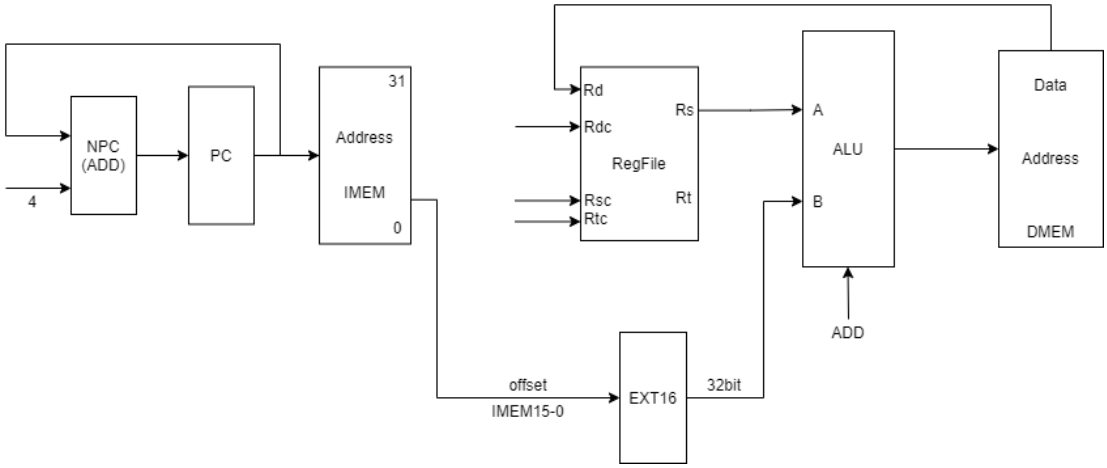
4. JR



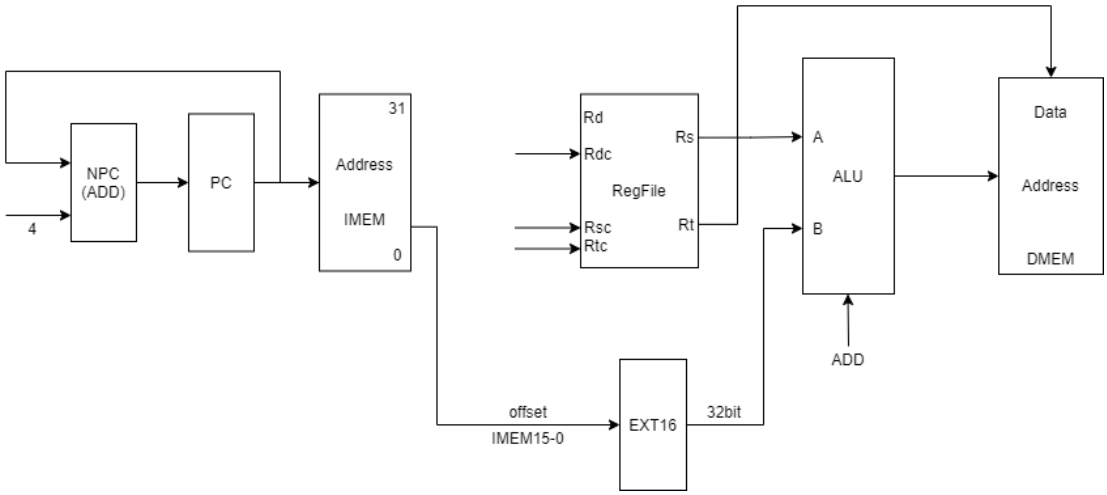
5. ADDI/ADDIU/ANDI/ORI/XORI



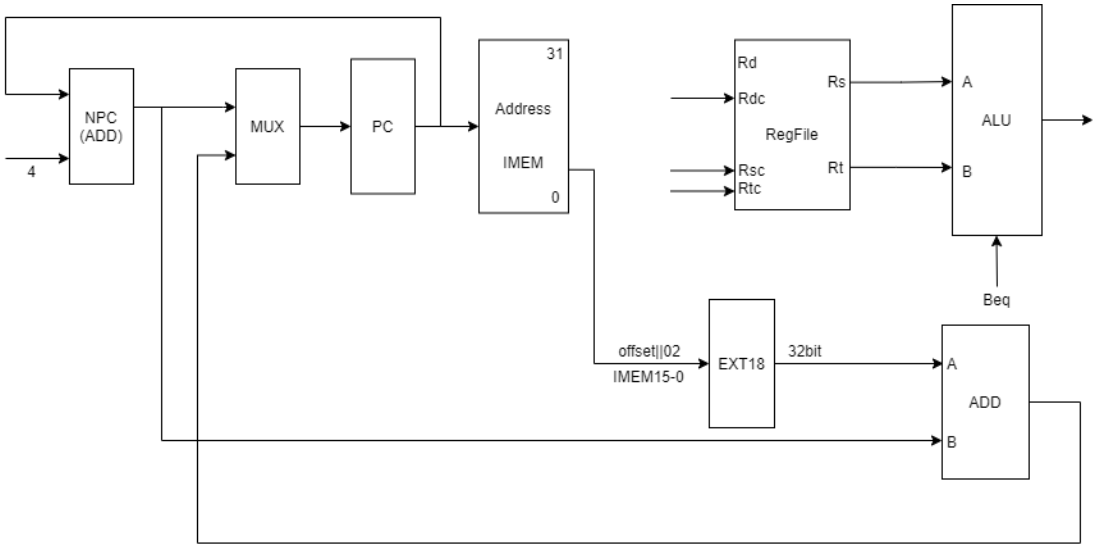
6. LW



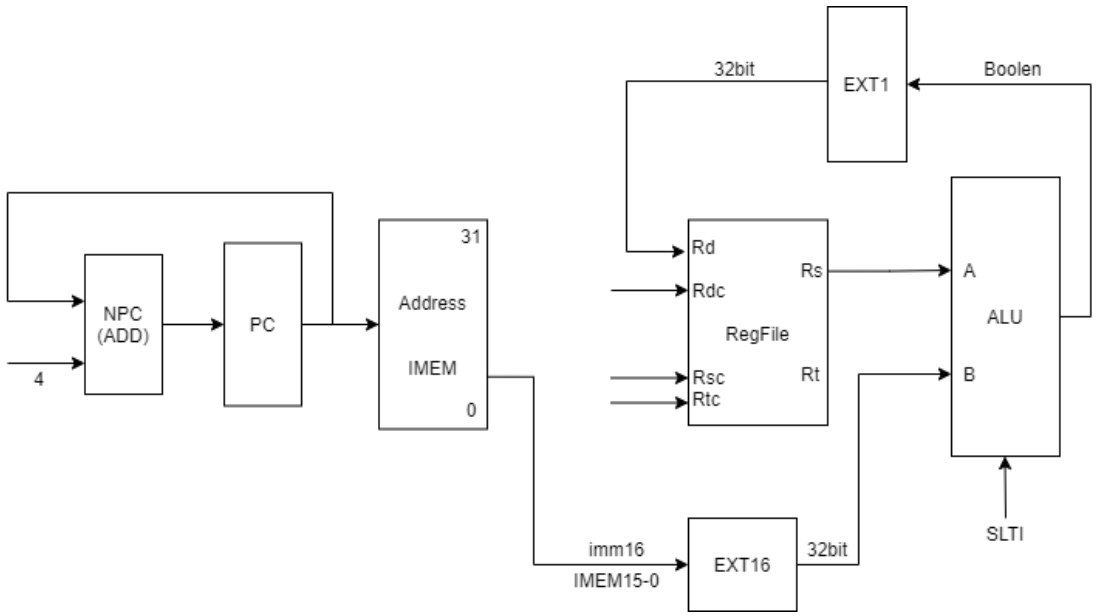
7. SW



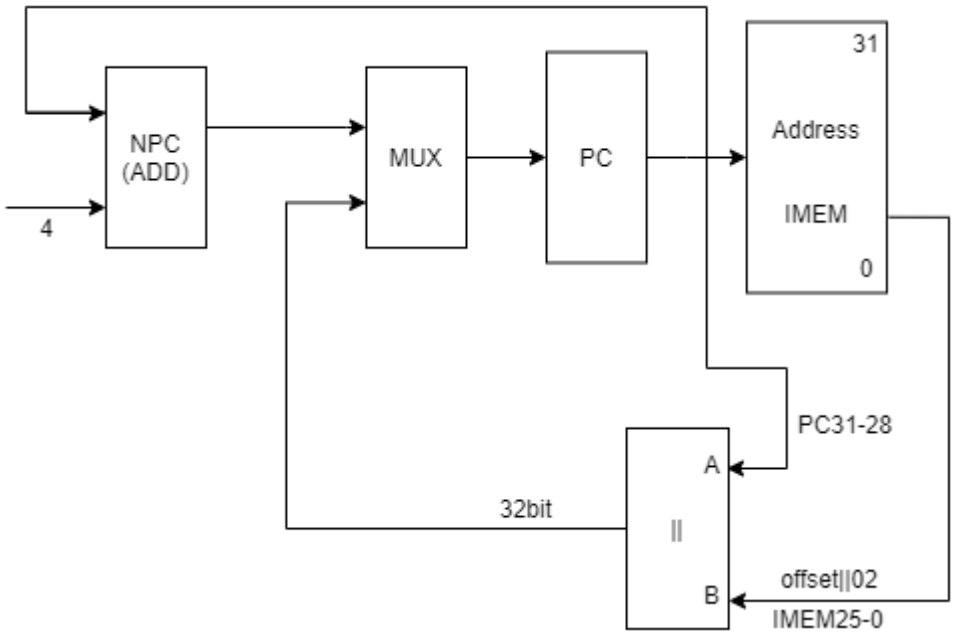
8. BEQ/BNE



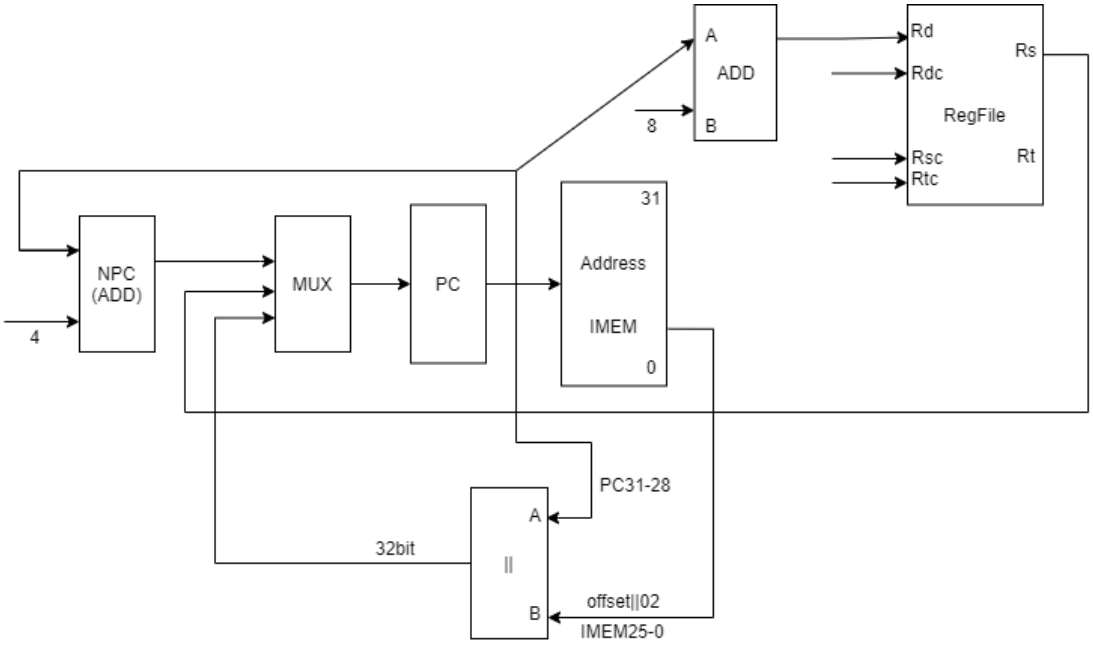
9. SLTI/SLTIU/LUI



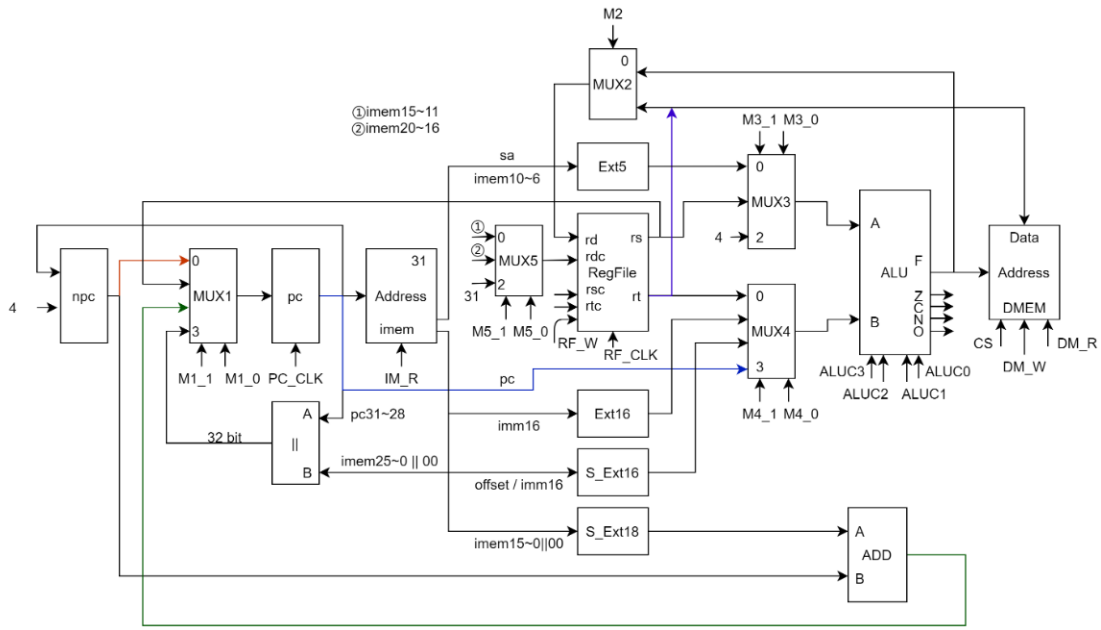
10. J


NPC
(ADD)
4
MUX
PC
31
Address
IMEM
0
PC31-28
A
||
B
32bit
offset||02
IMEM25-0

11.JAL


A
ADD
8
B
Rd
Rdc
RegFile
Rsc
Rtc
Rs
Rt
NPC
(ADD)
4
MUX
PC
31
Address
IMEM
0
PC31-28
A
||
B
32bit
offset||02
IMEM25-0

总数据通路：



各部件输入输出关系如下图：

指令	pc	npc	imem	RegFile	ALU		Ext5	Ext16	DMEM		S_Ext16	Ext18	S_Ext18	ADD			
				rd	A	B			Addr	Data				A	B	A	B
ADD	npc	pc	pc	ALU	rs	rt											
ADDU	npc	pc	pc	ALU	rs	rt											
SUB	npc	pc	pc	ALU	rs	rt											
SUBU	npc	pc	pc	ALU	rs	rt											
AND	npc	pc	pc	ALU	rs	rt											
OR	npc	pc	pc	ALU	rs	rt											
XOR	npc	pc	pc	ALU	rs	rt											
NOR	npc	pc	pc	ALU	rs	rt											
SLT	npc	pc	pc	ALU	rs	rt											
SLTU	npc	pc	pc	ALU	rs	rt											
SLL	npc	pc	pc	ALU	Ext5	rt	sa										
SRL	npc	pc	pc	ALU	Ext5	rt	sa										
SRA	npc	pc	pc	ALU	Ext5	rt	sa										
SLLV	npc	pc	pc	ALU	rs	rt											
SRLV	npc	pc	pc	ALU	rs	rt											
SRAV	npc	pc	pc	ALU	rs	rt											
JR	rs	pc	pc														
ADDI	npc	pc	pc	ALU	rs	Ext16					imm16						
ADDIU	npc	pc	pc	ALU	rs	Ext16		imm16									
ANDI	npc	pc	pc	ALU	rs	Ext16		imm16									
ORI	npc	pc	pc	ALU	rs	Ext16		imm16									
XORI	npc	pc	pc	ALU	rs	Ext16		imm16									
LW	npc	pc	pc	Data	rs	S_Ext16			alu		offset						
SW	npc	pc	pc		rs	S_Ext16			alu	rt	offset						
BEQ	ADD	pc	pc		rs	rt							offset	npc	S_Ext18		
BNE	ADD	pc	pc		rs	rt							offset	npc	S_Ext18		
SLTI	npc	pc	pc	ALU	rs	S_Ext16					imm16						
SLTIU	npc	pc	pc	ALU	rs	Ext16		imm16									
LUI	npc	pc	pc	ALU	16	Ext16		imm16									
J	||	pc	pc													pc31-28	imem25-0
JAL	||	pc	pc	pc										32'h4	npc	pc31-28	imem25-0

部件功能说明：

NPC：即 PC+4，可用简单加法实现

PC：指令计数器

IMEM：指令存储器

DMEM：数据存储器

RegFile：寄存器堆

ALU：算术逻辑单元，实现算数运算与逻辑运算

Ext5：将 5 位数据无符号扩展为 32 位

Ext16：将 16 位数据无符号扩展为 32 位

S_Ext16：将 16 位数据有符号扩展为 32 位

ADD：加法器

||：数据拼接

3.4 CPU 控制器设计

各指令控制信号表：

指令	PC_CLK	IM_R	Rsc4~0	Rtc4~0	Rdc4~0	RF_W	RF_CLK	CS	DM_R	DM_W	ALU3	ALU2	ALU1	ALU0
ADD	1	1	IM25~21	IM20~16	IM15~11	1	1	0	0	0	0	0	1	0
ADDU	1	1	IM25~21	IM20~16	IM15~11	1	1	0	0	0	0	0	0	0
SUB	1	1	IM25~21	IM20~16	IM15~11	1	1	0	0	0	0	0	1	1
SUBU	1	1	IM25~21	IM20~16	IM15~11	1	1	0	0	0	0	0	0	1
AND	1	1	IM25~21	IM20~16	IM15~11	1	1	0	0	0	0	1	0	0
OR	1	1	IM25~21	IM20~16	IM15~11	1	1	0	0	0	0	1	0	1
XOR	1	1	IM25~21	IM20~16	IM15~11	1	1	0	0	0	0	1	1	0
NOR	1	1	IM25~21	IM20~16	IM15~11	1	1	0	0	0	0	1	1	1
SLT	1	1	IM25~21	IM20~16	IM15~11	1	1	0	0	0	1	0	1	1
SLTU	1	1	IM25~21	IM20~16	IM15~11	1	1	0	0	0	1	0	1	0
SLL	1	1	0	IM20~16	IM15~11	1	1	0	0	0	1	1	1	X
SRL	1	1	0	IM20~16	IM15~11	1	1	0	0	0	1	1	0	1
SRA	1	1	0	IM20~16	IM15~11	1	1	0	0	0	1	1	0	0
SLLV	1	1	IM25~21	IM20~16	IM15~11	1	1	0	0	0	1	1	1	X
SRLV	1	1	IM25~21	IM20~16	IM15~11	1	1	0	0	0	1	1	0	1
SRAV	1	1	IM25~21	IM20~16	IM15~11	1	1	0	0	0	1	1	0	0
JR	1	1	IM25~21	0	0	0	0	0	0	0	0	0	0	0
ADDI	1	1	IM25~21	0	IM20~16	1	1	0	0	0	0	0	1	0
ADDIU	1	1	IM25~21	0	IM20~16	1	1	0	0	0	0	0	0	0
ANDI	1	1	IM25~21	0	IM20~16	1	1	0	0	0	0	1	0	0
ORI	1	1	IM25~21	0	IM20~16	1	1	0	0	0	0	1	0	1
XORI	1	1	IM25~21	0	IM20~16	1	1	0	0	0	0	1	1	0
LW	1	1	IM25~21	0	IM20~16	1	1	1	1	0	0	0	0	0
SW	1	1	IM25~21	IM20~16	0	0	0	1	0	1	0	0	0	0
BEQ	1	1	IM25~21	IM20~16	0	0	0	0	0	0	0	0	0	1
BNE	1	1	IM25~21	IM20~16	0	0	0	0	0	0	0	0	0	1
SLTI	1	1	IM25~21	0	IM20~16	1	1	0	0	0	1	0	1	1
SLTIU	1	1	IM25~21	0	IM20~16	1	1	0	0	0	1	0	1	0
LUI	1	1	0	0	IM20~16	1	1	0	0	0	1	0	0	X
J	1	1	0	0	0	0	0	0	0	0	0	0	0	0
JAL	1	1	0	0	31	1	1	0	0	0	0	0	0	0

指令	M1_1	M1_0	M2	M3_1	M3_0	M4_1	M4_0	M5_1	M5_0
ADD	0	0	0	0	1	0	0	0	0
ADDU	0	0	0	0	1	0	0	0	0
SUB	0	0	0	0	1	0	0	0	0
SUBU	0	0	0	0	1	0	0	0	0
AND	0	0	0	0	1	0	0	0	0
OR	0	0	0	0	1	0	0	0	0
XOR	0	0	0	0	1	0	0	0	0
NOR	0	0	0	0	1	0	0	0	0
SLT	0	0	0	0	1	0	0	0	0
SLTU	0	0	0	0	1	0	0	0	0
SLL	0	0	0	0	0	0	0	0	0
SRL	0	0	0	0	0	0	0	0	0
SRA	0	0	0	0	0	0	0	0	0
SLLV	0	0	0	0	1	0	0	0	0
SRLV	0	0	0	0	1	0	0	0	0
SRAV	0	0	0	0	1	0	0	0	0
JR	0	1	0	0	0	0	0	0	0
ADDI	0	0	0	0	1	1	0	0	1
ADDIU	0	0	0	0	1	1	0	0	1
ANDI	0	0	0	0	1	0	1	0	1
ORI	0	0	0	0	1	0	1	0	1
XORI	0	0	0	0	1	0	1	0	1
LW	0	0	1	0	1	1	0	0	1
SW	0	0	0	0	1	1	0	0	0
BEQ	Z==1	0	0	0	1	0	0	0	0
BNE	Z==0	0	0	0	1	0	0	0	0
SLTI	0	0	0	0	1	1	0	0	1
SLTIU	0	0	0	0	1	0	1	0	1
LUI	0	0	0	0	0	0	1	0	1
J	1	1	0	0	0	0	0	0	0
JAL	1	1	0	1	0	1	1	1	0

控制信号说明：
PC_CLK：pc 寄存器时钟
IM_R：指令寄存器 IMEM 读有效信号
Rsc4~0：rs 寄存器选择输入控制端
Rtc4~0：rt 寄存器选择输入控制端
Rdc4~0：rd 寄存器选择输入控制端
RF_W：RegFile 写信号
RF_CLK：RegFile 时钟
CS：数据存储器片选信号
DM_R：数据存储器读信号
DM_W：数据存储器写信号
ALU3：ALU 控制端 3
ALU2：ALU 控制端 2
ALU1：ALU 控制端 1
ALU0：ALU 控制端 0
M1_1：MUX1 选择器控制端 1
M1_0：MUX1 选择器控制端 0
M2：MUX2 选择器控制端
M3_1：MUX3 选择器控制端 1
M3_0：MUX3 选择器控制端 0
M4_1：MUX4 选择器控制端 1
M4_0：MUX4 选择器控制端 0
M5_1：MUX5 选择器控制端 1
M5_0：MUX5 选择器控制端 0

各微指令逻辑表达式如下图：

信号	逻辑表达式
PC_CLK	CLK（上升沿，后同）
IM_R	1
Rsc4~0	IM25~21
Rtc4~0	IM20~16
Rdc4~0	IM15~11(ADD+ADDU+SUB+SUBU+AND+OR+XOR+NOR+SLT+SLTU+SLL+SRL+SRA+SLLV+SRLV+SRAV)+IM20~16(ADDI+ADDIU+ANDI+ORI+XORI+LW+SLTI+SLTIU+LUI)+31(JAL)
RF_W	ADD+ADDU+SUB+SUBU+AND+OR+XOR+NOR+SLT+SLTU+SLL+SRL+SRA+SLLV+SRLV+SRAV+ADDI+ADDIU+ANDI+ORI+XORI+LW+SLTI+SLTIU+LUI+JAL
RF_CLK	CLK
CS	LW+SW
DM_R	LW
DM_W	SW
ALUC3	SLT+SLTU+SLL+SRL+SRA+SLLV+SRLV+SRAV+SLTI+SLTIU+LUI
ALUC2	AND+OR+XOR+NOR+SLL+SRL+SRA+SLLV+SRLV+SRAV+ANDI+ORI+XORI
ALUC1	ADD+SUB+XOR+NOR+SLT+SLTU+SLL+SLLV+ADDI+XORI+SLTI+SLTIU
ALUC0	SUB+SUBU+OR+NOR+SLT+SRL+SRLV+ORI+BEQ+BNE+SLTI
M1_1	BEQ(Z==0)+BNE(Z==1)+J+JAL
M1_0	JR+J+JAL
M2	LW
M3_1	JAL
M3_0	ADD+ADDU+SUB+SUBU+AND+OR+XOR+NOR+SLT+SLTU+SLLV+SRLV+SRAV+ADDI+ ADDIU+ANDI+ORI+XORI+LW+SW+BEQ+BNE+SLTI+SLTIU
M4_1	ADDI+ADDIU+LW+SW+SLTI+JAL
M4_0	ANDI+ORI+XORI+SLTIU+LUI+JAL
M5_1	JAL
M5_0	ADDI+ADDIU+ANDI+ORI+XORI+LW+SLTI+SLTIU+LUI

四、模块建模

4.1 sccomp_dataflow //顶层模块，调用 cpu、imem、dram 三个模块

```
module sccomp_dataflow(
    input clk_in,
    input reset,
    output [31:0] inst,
    output [31:0] pc
    );
    //DM 信号
    wire CS,DM_W,DM_R;
    wire [31:0] DM_addr;
    wire [31:0] DM_WData,DM_RData;

    wire [31:0] instr_addr;
```

```
    wire [31:0] dm_addr;
    assign instr_addr = pc - 32'h0040_0000;

    assign dm_addr = (DM_addr - 32'h1001_0000) / 4;

    //调用 IP 核
    imem IM(instr_addr[12:2],inst);

    Dram Dram(clk_in,CS,DM_W,DM_R,dm_addr,DM_WData,DM_RData);

    cpu sccpu(clk_in,reset,inst,DM_addr,DM_RData,DM_WData,CS,DM_W,DM_R,
pc);

endmodule
```

4.2 cpu //CPU 模块，实现指令译码、数据通路设计、控制器设计等核心功能

```
module cpu(
    input clk,
    input rst,
    //data(instruction) from iram
    input [31:0] IM,
    //data exchanged with dram
    output [31:0] DM_addr,
    input [31:0] DM_RData,
    output [31:0] DM_WData,
    //control of dram
    output CS,
    output DM_W,
    output DM_R,
    //output of pcre
    output [31:0] PC_out
    );

    //pcreg
    wire PC_CLK;
    wire ena;
    wire [31:0] PC_in;
    //wire [31:0] PC_out;

    //NPC
    wire [31:0] NPC;

    //regfile
    wire [31:0] rs;
```

```
wire [31:0] rt;
wire [31:0] rd;
wire [4:0] rdc;
wire [4:0] rsc;
wire [4:0] rtc;
wire RF_CLK;
wire RF_W;

//alu
wire [3:0] aluc;
wire [31:0] alu_a;
wire [31:0] alu_b;
wire [31:0] alu_c;
wire zero;
wire carry;
wire negative;
wire overflow;

//Iram
//wire [31:0] IM;

//EXT
wire [31:0] Ext5;
wire [31:0] Ext16;
wire [31:0] S_Ext16;
wire [31:0] S_Ext18;

//MUX
wire M1_1;
wire M1_0;
wire [31:0] M1_out;
wire M2;
wire [31:0] M2_out;
wire M3_1;
wire M3_0;
wire [31:0] M3_out;
wire M4_1;
wire M4_0;
wire [31:0] M4_out;
wire M5_1;
wire M5_0;
wire [4:0] M5_out;

//ADD
```

```
wire [31:0] ADD_A;
wire [31:0] ADD_B;
wire [31:0] ADD_C;

//Connect||
wire [3:0] Connect_A;
wire [27:0] Connect_B;
wire [31:0] Connect_C;

//指令
wire ADD = (IM[31:26]==6'b0) && (IM[5:0]==6'b100000);
wire ADDU = (IM[31:26]==6'b0) && (IM[5:0]==6'b100001);
wire SUB = (IM[31:26]==6'b0) && (IM[5:0]==6'b100010);
wire SUBU = (IM[31:26]==6'b0) && (IM[5:0]==6'b100011);
wire AND = (IM[31:26]==6'b0) && (IM[5:0]==6'b100100);
wire OR = (IM[31:26]==6'b0) && (IM[5:0]==6'b100101);
wire XOR = (IM[31:26]==6'b0) && (IM[5:0]==6'b100110);
wire NOR = (IM[31:26]==6'b0) && (IM[5:0]==6'b100111);
wire SLT = (IM[31:26]==6'b0) && (IM[5:0]==6'b101010);
wire SLTU = (IM[31:26]==6'b0) && (IM[5:0]==6'b101011);
wire SLL = (IM[31:26]==6'b0) && (IM[5:0]==6'b000000);
wire SRL = (IM[31:26]==6'b0) && (IM[5:0]==6'b000010);
wire SRA = (IM[31:26]==6'b0) && (IM[5:0]==6'b000011);
wire SLLV = (IM[31:26]==6'b0) && (IM[5:0]==6'b000100);
wire SRLV = (IM[31:26]==6'b0) && (IM[5:0]==6'b000110);
wire SRAV = (IM[31:26]==6'b0) && (IM[5:0]==6'b000111);
wire JR = (IM[31:26]==6'b0) && (IM[5:0]==6'b001000);

wire ADDI = (IM[31:26]==6'b001000);
wire ADDIU = (IM[31:26]==6'b001001);
wire ANDI = (IM[31:26]==6'b001100);
wire ORI = (IM[31:26]==6'b001101);
wire XORI = (IM[31:26]==6'b001110);
wire LW = (IM[31:26]==6'b100011);
wire SW = (IM[31:26]==6'b101011);
wire BEQ = (IM[31:26]==6'b000100);
wire BNE = (IM[31:26]==6'b000101);
wire SLTI = (IM[31:26]==6'b001010);
wire SLTIU = (IM[31:26]==6'b001011);
wire LUI = (IM[31:26]==6'b001111);

wire J = (IM[31:26]==6'b000010);
wire JAL = (IM[31:26]==6'b000011);
```

```
    //pcreg
    assign PC_CLK = clk;
    assign ena = 1;
    assign PC_in = M1_out;

    //Regfiles
    assign rd = M2_out;
    assign rdc = M5_out;
    assign rsc = IM[25:21];
    assign rtc = IM[20:16];
    assign RF_W = ADD||ADDU||SUB||SUBU||AND||OR||XOR||NOR||SLT||SLTU||S
LL||SRL||SRA||SLLV||SRLV||
                  SRAV||ADDI||ADDIU||ANDI||ORI||XORI||LW||SLTI||SLTIU||
LUI||JAL;
    assign RF_CLK = clk;

    //alu
    assign aluc[3] = SLT||SLTU||SLL||SRL||SRA||SLLV||SRLV||SRAV||SLTI||
SLTIU||LUI;
    assign aluc[2] = AND||OR||XOR||NOR||SLL||SRL||SRA||SLLV||SRLV||SRAV
||ANDI||ORI||XORI;
    assign aluc[1] = ADD||SUB||XOR||NOR||SLT||SLTU||SLL||SLLV||ADDI||XO
RI||SLTI||SLTIU||JAL;
    assign aluc[0] = SUB||SUBU||OR||NOR||SLT||SRL||SRLV||ORI||BEQ||BNE|
|SLTI;
    assign alu_a = M3_out;
    assign alu_b = M4_out;

    //MUX
    assign M1_1 = (BEQ && (zero == 1))||(BNE && (zero == 0))||J||JAL;
    assign M1_0 = JR||J||JAL;
    assign M2 = LW;
    assign M3_1 = JAL;
    assign M3_0 = ADD||ADDU||SUB||SUBU||AND||OR||XOR||NOR||SLT||SLTU||S
LLV||SRLV||SRAV||ADDI||
                ADDIU||ANDI||ORI||XORI||LW||SW||BEQ||BNE||SLTI||SLTIU;
    assign M4_1 = ADDI||ADDIU||LW||SW||SLTI||JAL;
    assign M4_0 = ANDI||ORI||XORI||SLTIU||LUI||JAL;
    assign M5_1 = JAL;
    assign M5_0 = ADDI||ADDIU||ANDI||ORI||XORI||LW||SLTI||SLTIU||LUI;

    //Dram
    assign CS = LW||SW;
    assign DM_R = LW;
```

```
    assign DM_W = SW;
    assign DM_addr = alu_c;
    assign DM_WData = rt;

    //Connect||
    assign Connect_A = PC_out[31:28];
    assign Connect_B = {IM[25:0],2'b00};
    assign Connect_C = {Connect_A,Connect_B};

    //ADD
    assign ADD_A = S_Ext18;
    assign ADD_B = NPC;
    assign ADD_C = ADD_A + ADD_B;

    assign NPC = PC_out + 4;

    Ext5 E5(IM[10:6],Ext5);
    Ext16 E16(IM[15:0],Ext16);
    S_Ext16 S_E16(IM[15:0],S_Ext16);
    S_Ext18 S_E18({IM[15:0],2'b00},S_Ext18);

    MUX MUX1(NPC,rs,ADD_C,Connect_C,M1_1,M1_0,M1_out);
    assign M2_out = M2==0?alu_c:DM_RData;
    MUX MUX3(Ext5,rs,4,0,M3_1,M3_0,M3_out);
    MUX MUX4(rt,Ext16,S_Ext16,PC_out,M4_1,M4_0,M4_out);
    MUX_5bit MUX5(IM[15:11],IM[20:16],31,0,M5_1,M5_0,M5_out);

    pcreg pcreg(PC_CLK,rst,ena,PC_in,PC_out);

    alu alu(alu_a,alu_b,aluc,alu_c,zero,carry,negative,overflow);

    regfile cpu_ref(RF_CLK,rst,1,RF_W,rsc,rtc,rdc,rd,rs,rt);


endmodule
```

4.3 alu //算术逻辑单元模块，实现多种逻辑运算与算术运算

```
module alu(
    input [31:0] a,
    input [31:0] b,
    input [3:0] aluc,
    output reg [31:0] r,
```

```
output reg zero,
output reg carry,
output reg negative,
output reg overflow
);
reg [31:0] oTmpt;
always@(*)
    case(aluc)
        4'b0000:r=a+b;
        4'b0010:r=$signed(a)+$signed(b);
        4'b0001:r=a-b;
        4'b0011:r=a-b;
        4'b0100:r=a&b;
        4'b0101:r=a|b;
        4'b0110:r=a^b;
        4'b0111:r=~(a|b);
        4'b1000:r={b[15:0],16'b0};
        4'b1001:r={b[15:0],16'b0};
        4'b1011:r=($signed(a)<$signed(b))?1:0;
        4'b1010:r=(a<b)?1:0;
        4'b1100:r=$signed(b)>>>a;
        4'b1110:r=b<<a;
        4'b1111:r=b<<a;
        4'b1101:r=b>>a;
        default:r=32'h00000000;
    endcase

//zero
always@(*)
    if(aluc==4'b1011||aluc==4'b1010)
        if(a==b)
            zero=1'b1;
        else
            zero=1'b0;
    else if(r==0)
        zero=1'b1;
    else
        zero=1'b0;

//negative
always@(*)
    if(aluc==4'b0010||aluc==4'b0011)
        if($signed(r)<0)
            negative=1'b1;
```

```
        else
            negative=1'b0;
    else if(aluc==4'b1011)
        if(r==1'b1)
            negative=1'b1;
        else
            negative=1'b0;
    else
        if(r[31]==1'b1)
            negative=1'b1;
        else
            negative=1'b0;

//carry
always@(*)
    if(aluc==4'b0000)
        if(a[31]==1'b1&&b[31]==1'b1)
            carry=1'b1;
        else if(a[31]==1'b1&&b[31]==1'b0&&r[31]==1'b0)
            carry=1'b1;
        else if(a[31]==1'b0&&b[31]==1'b1&&r[31]==1'b0)
            carry=1'b1;
        else
            carry=1'b0;
    else if(aluc==4'b0001)
        if(a<b)
            carry=1'b1;
        else
            carry=1'b0;
    else if(aluc==4'b1010)
        if(a<b)
            carry=1'b1;
        else
            carry=1'b0;
    else if(aluc==4'b1100)
        begin
        oTmpt=$signed(b)>>>(a-1);
        carry=oTmpt[0];
        end
    else if(aluc==4'b1101)
        begin
        oTmpt=b>>>(a-1);
        carry=oTmpt[0];
        end
```

```
        else if(aluc==4'b1110||aluc==4'b1111)
            begin
            oTmpt=b<<(a-1);
            carry=oTmpt[31];
            end

    //overflow
    always@(*)
        if(aluc==4'b0010)
            if(a[31]==b[31]&&~r[31]==a[31])
                overflow=1'b1;
            else
                overflow=1'b0;
        else if(aluc==4'b0011)
            if(a[31]==0&&b[31]==1&&r[31]==1)
                overflow=1'b1;
            else if(a[31]==1&&b[31]==0&&r[31]==0)
                overflow=1'b1;
            else
                overflow=1'b0;

endmodule
```

4.4 Dram //数据存储器模块，用于向内存中读写数据

```
module Dram(
    input clk,
    input ena,
    input DM_W,
    input DM_R,
    input [31:0] addr,
    input [31:0] data_in,
    output [31:0] data_out
    );

    reg [31:0] RAM [31:0];
    assign data_out = (ena && DM_R) ? RAM[addr] : 32'hzzzzzzzz;

    always @ (posedge clk)
    begin
        if(ena && DM_W)
            RAM[addr] <= data_in;
    end

endmodule
```

4.5 pcreg //指令寄存器模块，每个时钟下降沿更新一次

```
module pcreg(
    input clk,
    input rst,
    input ena,

    input [31:0] data_in,
    output reg [31:0] data_out
    );
    always@(negedge clk or posedge rst)
        begin
            if(rst==1'b1)
                data_out=32'h00400000;
            else
                if(ena==1'b1)
                    data_out=data_in;
                else if(ena==1'b0)
                    data_out=data_out;
        end
endmodule
```

4.6 regfile //寄存器堆模块，包括 32 个寄存器，根据地址内容进行读写

```
module regfile(
    input clk,
    input rst,
    input en,
    input rf_write,
    input [4:0] rsc,
    input [4:0] rtc,
    input [4:0] rdc,
    input [31:0] rd,
    output [31:0] rs,
    output [31:0] rt
    );
    reg [31:0] array_reg[31:0];
    reg [5:0] i;
    assign rs = en ? array_reg[rsc] : 32'bz;
    assign rt = en ?array_reg[rtc] : 32'bz;
    always @(negedge clk or posedge rst) begin
      if (rst) begin
        for(i=0;i<32;i=i+1)
          array_reg[i] <= 0;
        end
    end
```

```
    always@(posedge clk) begin begin
      if (rf_write && en && (rdc != 0))
        array_reg[rdc] <= rd;
      end
    end
endmodule
```

4.7 MUX //四选一数据选择器

```
module MUX(
    input [31:0] iC0,
    input [31:0] iC1,
    input [31:0] iC2,
    input [31:0] iC3,
    input iS1,
    input iS0,
    output [31:0] oZ
    );
reg[31:0] tmpt;
always@(*)
begin
   if(iS1==0)
        begin
            if(iS0==0)
                tmpt = iC0;
            else
                tmpt = iC1;
        end
   else
        begin
            if(iS0==0)
                tmpt = iC2;
            else
                tmpt = iC3;
        end
end
assign oZ = tmpt;
endmodule
```

4.7 MUX_5bit //五位四选一数据选择器

```
module MUX_5bit(
    input [4:0] iC0,
    input [4:0] iC1,
    input [4:0] iC2,
```

```
    input [4:0] iC3,
    input iS1,
    input iS0,
    output [4:0] oZ
    );
reg[31:0] tmpt;
    always@(*)
    begin
       if(iS1==0)
            begin
                if(iS0==0)
                    tmpt = iC0;
                else
                    tmpt = iC1;
            end
       else
            begin
                if(iS0==0)
                    tmpt = iC2;
                else
                    tmpt = iC3;
            end
    end
    assign oZ = tmpt;
endmodule
```

4.8 Ext5 //将 5 位数据无符号扩展为 32 位

```
module Ext5(
    input [4:0] data_in,
    output [31:0] data_out
    );
    assign data_out = {27'b0,data_in};
endmodule
```

4.9 Ext16 //将 16 位数据无符号扩展为 32 位

```
module Ext16(
    input [15:0] data_in,
    output [31:0] data_out
    );
    assign data_out = {16'b0,data_in};
endmodule
```

4.10 S_Ext16 //将 16 位数据有符号扩展为 32 位

```
module S_Ext16(
    input [15:0] data_in,
    output [31:0] data_out
    );
    assign data_out = {{16{data_in[15]}},data_in};
endmodule
```

4.11 S_Ext18 //将 18 位数据有符号扩展为 32 位

```
module S_Ext18(
    input [17:0] data_in,
    output [31:0] data_out
    );
    assign data_out = {{14{data_in[17]}},data_in};
endmodule
```

4.12 imem //指令寄存器模块，通过调用 ip 核实现

```
`timescale 1ns/1ps

(* DowngradeIPIdentifiedWarnings = "yes" *)
module imem (
  a,
  spo
);

input wire [10 : 0] a;
output wire [31 : 0] spo;

  dist_mem_gen_v8_0_10 #(
    .C_FAMILY("artix7"),
    .C_ADDR_WIDTH(11),
    .C_DEFAULT_DATA("0"),
    .C_DEPTH(2048),
    .C_HAS_CLK(0),
    .C_HAS_D(0),
    .C_HAS_DPO(0),
    .C_HAS_DPRA(0),
    .C_HAS_I_CE(0),
    .C_HAS_QDPO(0),
    .C_HAS_QDPO_CE(0),
    .C_HAS_QDPO_CLK(0),
```

```
    .C_HAS_QDPO_RST(0),
    .C_HAS_QDPO_SRST(0),
    .C_HAS_QSPO(0),
    .C_HAS_QSPO_CE(0),
    .C_HAS_QSPO_RST(0),
    .C_HAS_QSPO_SRST(0),
    .C_HAS_SPO(1),
    .C_HAS_WE(0),
    .C_MEM_INIT_FILE("imem.mif"),
    .C_ELABORATION_DIR("./"),
    .C_MEM_TYPE(0),
    .C_PIPELINE_STAGES(0),
    .C_QCE_JOINED(0),
    .C_QUALIFY_WE(0),
    .C_READ_MIF(1),
    .C_REG_A_D_INPUTS(0),
    .C_REG_DPRA_INPUT(0),
    .C_SYNC_ENABLE(1),
    .C_WIDTH(32),
    .C_PARSER_TYPE(1)
  ) inst (
    .a(a),
    .d(32'B0),
    .dpra(11'B0),
    .clk(1'D0),
    .we(1'D0),
    .i_ce(1'D1),
    .qspo_ce(1'D1),
    .qdpo_ce(1'D1),
    .qdpo_clk(1'D0),
    .qspo_rst(1'D0),
    .qdpo_rst(1'D0),
    .qspo_srst(1'D0),
    .qdpo_srst(1'D0),
    .spo(spo),
    .dpo(),
    .qspo(),
    .qdpo()
  );
endmodule
```

五、测试模块建模

（要求列写各建模模块的 test bench 模块代码）

```
`timescale 1ns / 1ps
module cpu_tb(

    );
        reg clk_in;
        reg reset;
        reg start;
        wire[31:0] inst;
        wire[31:0] pc;


        sccomp_dataflow sc(clk_in, reset, inst, pc);

        wire [31:0] M1_out = sc.sccpu.M1_out;
        wire [31:0] NPC = sc.sccpu.NPC;
        wire [31:0] rd = sc.sccpu.rd;
        wire [31:0] rs = sc.sccpu.rs;
        wire [31:0] rt = sc.sccpu.rt;
        wire [31:0] alu_a = sc.sccpu.alu_a;
        wire [31:0] alu_b = sc.sccpu.alu_b;
        wire [31:0] alu_c = sc.sccpu.alu_c;
        wire [31:0] Ext5 = sc.sccpu.Ext5;
        wire [31:0] Ext16 = sc.sccpu.Ext16;
        wire [31:0] S_Ext16 = sc.sccpu.S_Ext16;
        wire [31:0] S_Ext18 = sc.sccpu.S_Ext18;
        wire [4:0] rsc = sc.sccpu.rsc;
        wire [4:0] rtc = sc.sccpu.rtc;
        wire M4_1 = sc.sccpu.M4_1;
        wire M4_0 = sc.sccpu.M4_0;

        wire [31:0] DM_addr = sc.DM_addr;
        wire [31:0] dm_addr = sc.dm_addr;

        wire CS = sc.CS;
        wire DM_R = sc.DM_R;


        integer file_output;
        integer counter = 0;

        initial
        begin
```

```
            file_output = $fopen("D:/computer_composition/MIPS31/result
.txt");
        end

        initial
        begin
            clk_in = 1;
            start  = 0;
            forever begin
                #50 clk_in = ~clk_in;
            end
        end

        initial begin
            reset     = 0;
            #6 reset  = 1;
            #50 reset = 0;
            start = 1;
        end

        always @(posedge clk_in) begin
            if (start)begin
                counter  = counter + 1;


                $fdisplay(file_output, "regfiles0 = %h", sc.sccpu.cpu_r
ef.array_reg[0]);
                $fdisplay(file_output, "regfiles1 = %h", sc.sccpu.cpu_r
ef.array_reg[1]);
                $fdisplay(file_output, "regfiles2 = %h", sc.sccpu.cpu_r
ef.array_reg[2]);
                $fdisplay(file_output, "regfiles3 = %h", sc.sccpu.cpu_r
ef.array_reg[3]);
                $fdisplay(file_output, "regfiles4 = %h", sc.sccpu.cpu_r
ef.array_reg[4]);
                $fdisplay(file_output, "regfiles5 = %h", sc.sccpu.cpu_r
ef.array_reg[5]);
                $fdisplay(file_output, "regfiles6 = %h", sc.sccpu.cpu_r
ef.array_reg[6]);
                $fdisplay(file_output, "regfiles7 = %h", sc.sccpu.cpu_r
ef.array_reg[7]);
                $fdisplay(file_output, "regfiles8 = %h", sc.sccpu.cpu_r
ef.array_reg[8]);
```

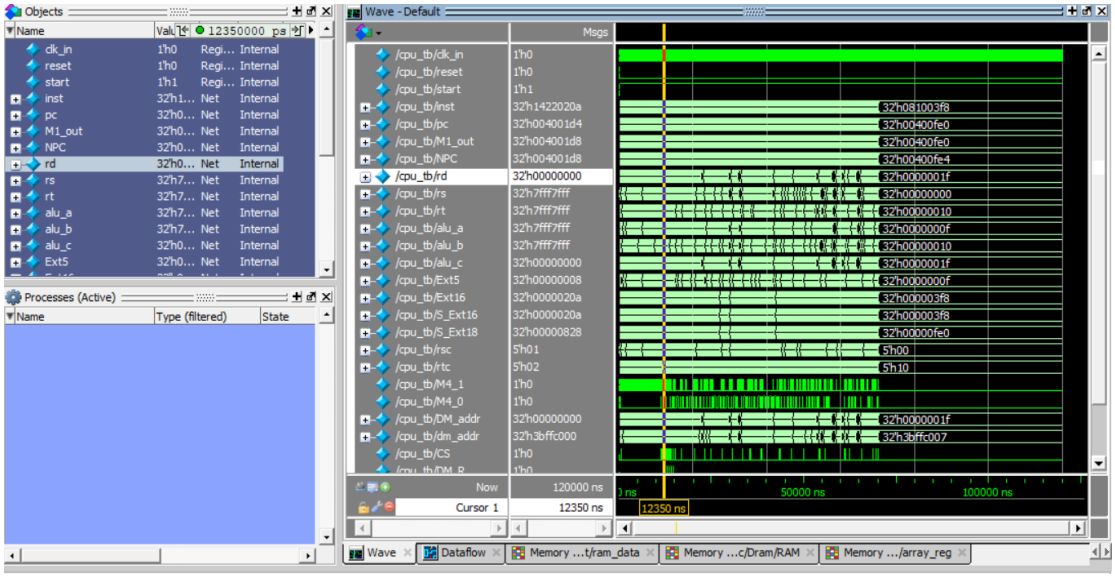
```
                $fdisplay(file_output, "regfiles9 = %h", sc.sccpu.cpu_r
ef.array_reg[9]);
                $fdisplay(file_output, "regfiles10 = %h", sc.sccpu.cpu_
ref.array_reg[10]);
                $fdisplay(file_output, "regfiles11 = %h", sc.sccpu.cpu_
ref.array_reg[11]);
                $fdisplay(file_output, "regfiles12 = %h", sc.sccpu.cpu_
ref.array_reg[12]);
                $fdisplay(file_output, "regfiles13 = %h", sc.sccpu.cpu_
ref.array_reg[13]);
                $fdisplay(file_output, "regfiles14 = %h", sc.sccpu.cpu_
ref.array_reg[14]);
                $fdisplay(file_output, "regfiles15 = %h", sc.sccpu.cpu_
ref.array_reg[15]);
                $fdisplay(file_output, "regfiles16 = %h", sc.sccpu.cpu_
ref.array_reg[16]);
                $fdisplay(file_output, "regfiles17 = %h", sc.sccpu.cpu_
ref.array_reg[17]);
                $fdisplay(file_output, "regfiles18 = %h", sc.sccpu.cpu_
ref.array_reg[18]);
                $fdisplay(file_output, "regfiles19 = %h", sc.sccpu.cpu_
ref.array_reg[19]);
                $fdisplay(file_output, "regfiles20 = %h", sc.sccpu.cpu_
ref.array_reg[20]);
                $fdisplay(file_output, "regfiles21 = %h", sc.sccpu.cpu_
ref.array_reg[21]);
                $fdisplay(file_output, "regfiles22 = %h", sc.sccpu.cpu_
ref.array_reg[22]);
                $fdisplay(file_output, "regfiles23 = %h", sc.sccpu.cpu_
ref.array_reg[23]);
                $fdisplay(file_output, "regfiles24 = %h", sc.sccpu.cpu_
ref.array_reg[24]);
                $fdisplay(file_output, "regfiles25 = %h", sc.sccpu.cpu_
ref.array_reg[25]);
                $fdisplay(file_output, "regfiles26 = %h", sc.sccpu.cpu_
ref.array_reg[26]);
                $fdisplay(file_output, "regfiles27 = %h", sc.sccpu.cpu_
ref.array_reg[27]);
                $fdisplay(file_output, "regfiles28 = %h", sc.sccpu.cpu_
ref.array_reg[28]);
                $fdisplay(file_output, "regfiles29 = %h", sc.sccpu.cpu_
ref.array_reg[29]);
                $fdisplay(file_output, "regfiles30 = %h", sc.sccpu.cpu_
ref.array_reg[30]);
```

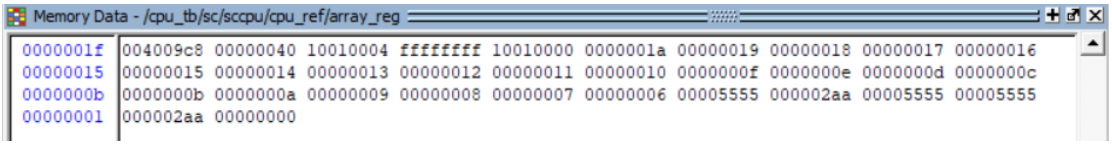
```
                $fdisplay(file_output, "regfiles31 = %h", sc.sccpu.cpu_
ref.array_reg[31]);

                $fdisplay(file_output, "instr = %h", inst);
                $fdisplay(file_output, "pc = %h", pc);
            end
        end
endmodule
```

六、实验结果

6.1 modelsim 仿真波形图

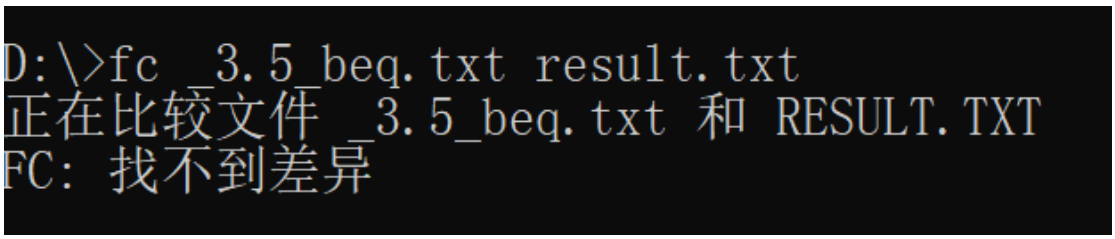


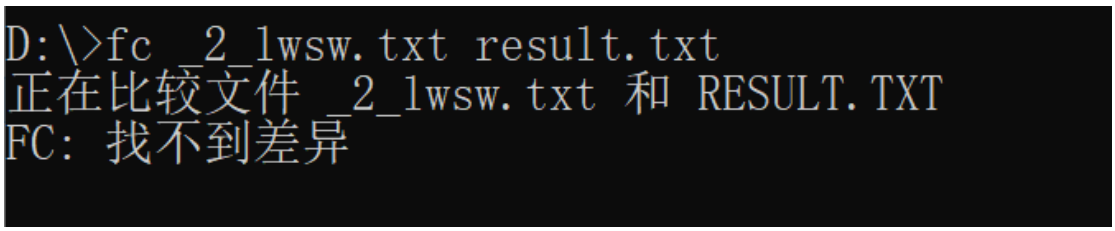


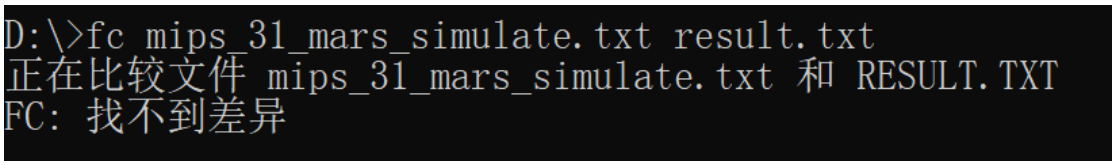
由上图我们可以看到，指令寄存器 IM 内存已读入相关指令译码后的结果，指令寄存器运行正常。同时寄存器堆 regfile 中已成功写入数据，寄存器堆也正常工作。

6.2 指令执行结果比对

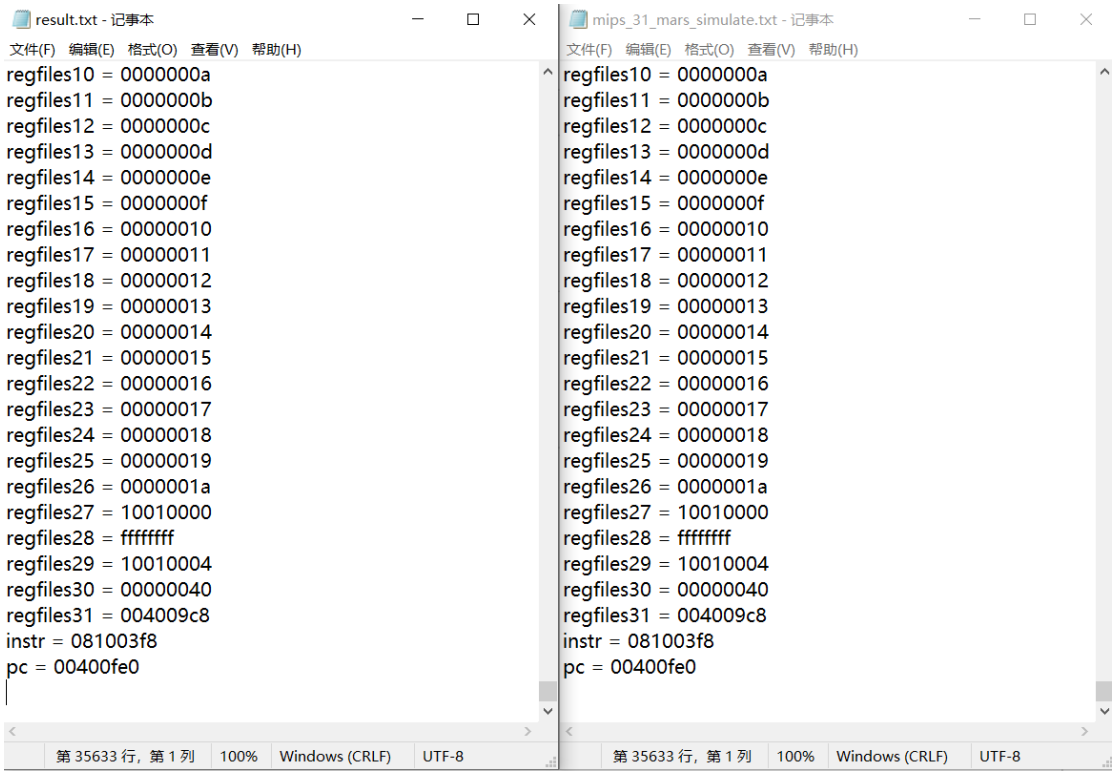
我通过采用 cmd 控制台的比较文件指令分别对 31 条指令以及最终版指令运行结果与标准结果进行比对，依次确认比对结果与标准结果并无差异以确认指令运行正确，CPU 设计正确。

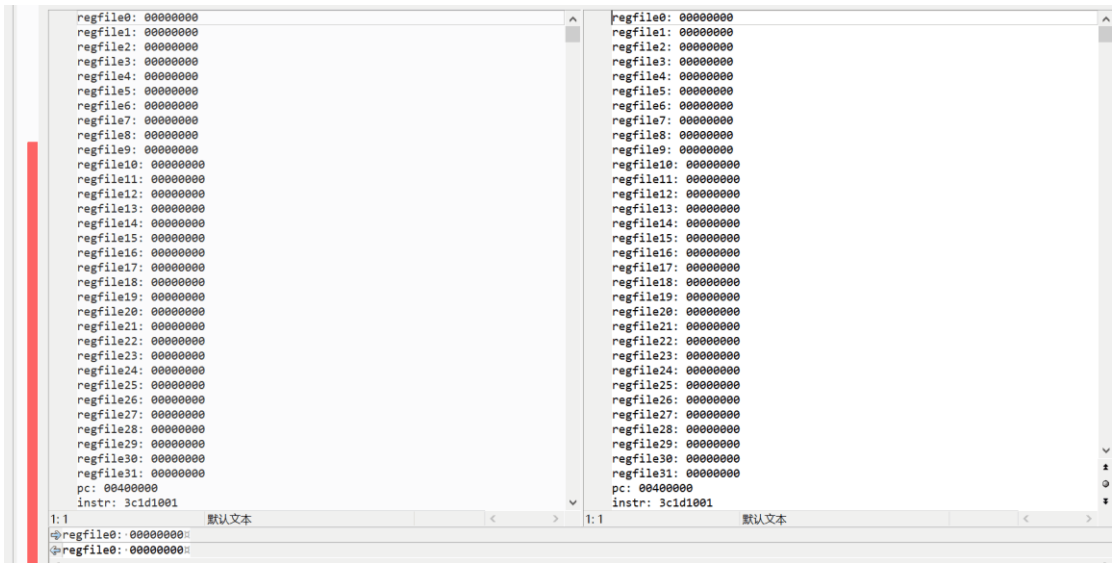






通过文件比对可以看出运行结果与标准并无差异，唯一一处不同在于运行结尾循环终止出，由此证明运行正常。





七、总结与体会

总体来说，在本次 31 条指令 CPU 的设计中，我从最初的对 CPU 构架处于一知半解的状态，只是在理论层面有所了解而理解不够深刻到通过实际操作完成本次作业实现一个能够运行 31 条指令的 CPU 后，对 CPU 的设计流程、数据通路、控制器等有了更深刻的理解，收获很大。

此次 CPU 设计过程，我是根据实验指导书上的顺序对不同部件依次进行设计的。首先是 CPU 数据通路的设计，我根据指令的功能，确定每条指令所用到的部件；其次根据各个指令所用的部件，表格列出，同时在表格中填入每个部件的数据输入来源，再根据数据输入来源画出每条指令的数据通路，最后将所有指

令数据通路合并成一个总的数据通路。合并过程涉及数据选择器 MUX，对于同一部件有不同输入来源的情况，根据数据选择器输入选择端决定在不同指令执行情况下将什么输入来源送到该部件。

数据通路设计完成后，我接着进行了 CPU 控制器的设计，具体流程为：首先绘制指令流程图，把每一条机器指令都分解为一系列微操作，编排指令操作时间表，再者进行微操作综合，将相同的微操作综合起来，通过不同指令间的或操作进行赋值得到每个微操作的逻辑表达式，记录各个微操作不同时刻的电平，若为高电平则执行此微操作。最后画出控制器逻辑电路，根据各微操作的逻辑表达式设计 CPU 控制单元。

数据通路以及控制器均设计完成后，前期准备工作告一段落，就进入了代码编写环节，通过顶层模块调用 cpu 模块、指令/数据寄存器模块，在 CPU 模块中调用诸如 ALU、regfile、pcreg、Ext 等子模块实现 CPU 中数据通路、控制器的设计。代码完成后，通过 IP 核读入指令，将寄存器结果写入文件与通过 MARS 编译后产生的结果进行比对，准确无误后，确实设计正确。

在设计过程中，我遇到了许多由于疏忽而产生的 bug，有些错误甚至极难发现修改，debug 过程还是比较艰难痛苦的。首先是在将数据通路与控制器通过代码呈现时无法正常读入诸如 IP 核中的指令数据等，导致 modelsim 仿真以及寄存器结果文件输出无有效值，均为未知值 x 或 z。我通过排查数据通路是否连接、命名是否正确，最终发现问题出在顶层模块调用子模块时命名出现了错误，instr 写成了 inst，导致无法正常写入。更多的错误出在依次对单条指令进行执行比对过程中寄存器结果、地址、程序计数器出现不一致的情况。具体的原因包括：控制信号逻辑表达式有误、数据选择器有误，经过仔细排查，重新设计更改指令数据通路，重新写不同微指令的逻辑表达式，最终解决了相关问题。