同济大学计算机系

计算机组成原理实验报告

学　　号 2253214

姓　　名 李闯

专　　业 计算机科学与技术

授课老师 郝泳涛

一、实验内容

在本次实验中，我们将使用 Verilog 语言实现 31 条 MIPS 指令（如图 1 所示）的 CPU 的设计和仿真。

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

图 1. 31 条 MIPS 31 条指令

二、数据通路设计

（1） ADD

指令格式：ADD rd, rs, rt

指令功能：将通用寄存器中的 32 位有符号数 rs 与 rt 相加产生一个 32 位有符号数存入目标寄存器 rd。

所需操作：

1. IMEM←PC （取指令）
2. Rd←Rs+Rt （执行指令）
3. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU	
				Rd	A	B
ADD	NPC	PC	PC	ALU	RS	Rt

数据通路：

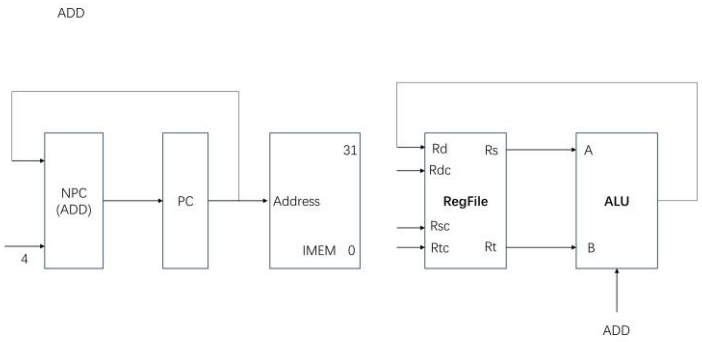


图 2. ADD 指令数据通路图

（2） ADDU

指令格式: ADDU rd, rs, rt

指令功能:将通用寄存器中的 32 位无符号数 rs 与 rt 相加产生一个 32 位无符号数存入目标寄存器 rd。

所需操作：

1. IMEM←PC （取指令）
2. Rd←Rs+Rt （执行指令）
3. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU	
				Rd	A	B
ADDU	NPC	PC	PC	ALU	Rs	Rt

数据通路：

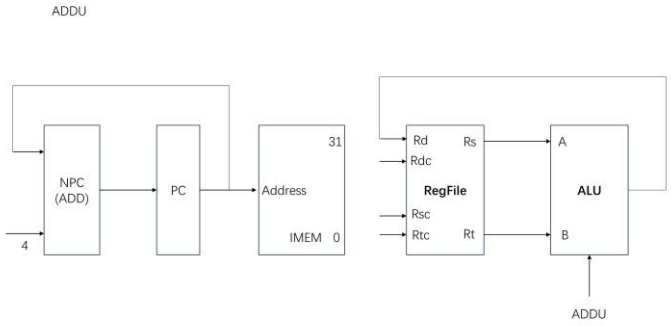


图 3. ADDU 指令数据通路图

（3） SUB

指令格式：SUB rd, rs, rt

指令功能：将通用寄存器中的 32 位有符号数 rs 与 rt 相减产生一个 32 位有符号数存入目标寄存器 rd。

所需操作：

1. IMEM←PC （取指令）
2. Rd←Rs-Rt （执行指令）
3. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU	
				Rd	A	B
SUB	NPC	PC	PC	ALU	Rs	Rt

数据通路：

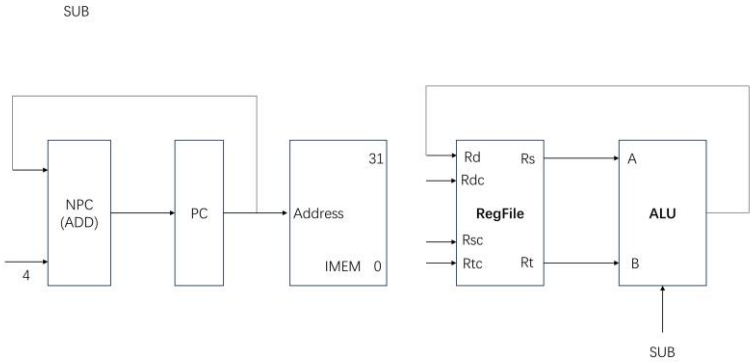


图 4. SUB **指令数据通路图**

（4） SUBU

指令格式：SUBU rd, rs, rt

指令功能：将通用寄存器中的 32 位无符号数 rs 与 rt 相减产生一个 32 位无符号数存入目标寄存器 rd。

所需操作：

1. IMEM←PC （取指令）
2. Rd←Rs-Rt （执行指令）
3. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU	
				Rd	A	B
SUBU	NPC	PC	PC	ALU	Rs	Rt

数据通路：

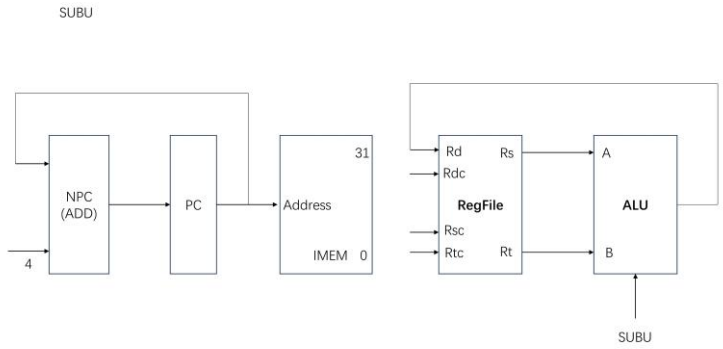


图 5. SUBU 指令数据通路图

（5） AND

指令格式：AND rd, rs, rt

指令功能：将通用寄存器中的 32 位数据 rs 与 rt 进行位与运算产生一个 32 位数据存入目标寄存器 rd。

所需操作：

1. IMEM←PC （取指令）
2. Rd←Rs AND Rt （执行指令）
3. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU	
				Rd	A	B
AND	NPC	PC	PC	ALU	Rs	Rt

数据通路：

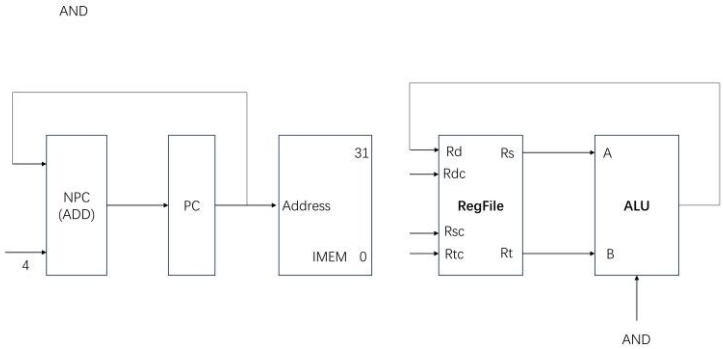


图 6. AND 指令数据通路图

（6） OR

指令格式：OR rd, rs, rt

指令功能:将通用寄存器中的 32 位数据 rs 与 rt 进行位或运算产生一个 32 位数据存入目标寄存器 rd。

所需操作：

1. IMEM←PC （取指令）
2. Rd←Rs OR Rt （执行指令）
3. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU	
				Rd	A	B
OR	NPC	PC	PC	ALU	Rs	Rt

数据通路：

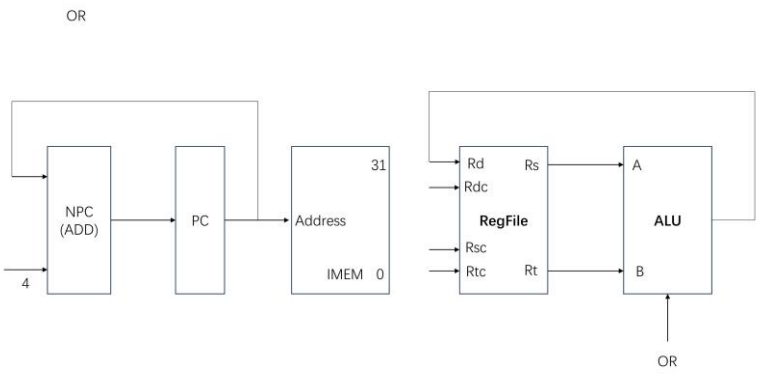


图 7. OR 指令数据通路图

（7） XOR

指令格式: XOR rd, rs, rt

指令功能:将通用寄存器中的 32 位数据 rs 与 rt 进行位异或运算产生一个 32 位数据存入目标寄存器 rd。

所需操作：

1. IMEM←PC （取指令）
2. Rd←Rs XOR Rt （执行指令）
3. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU	
				Rd	A	B
XOR	NPC	PC	PC	ALU	Rs	Rt

数据通路：

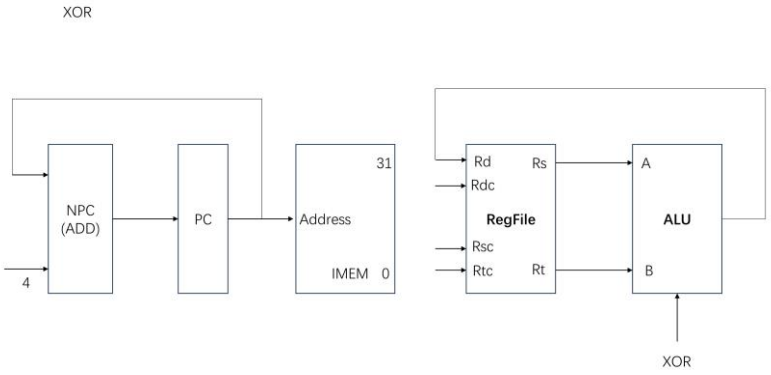


图 8. XOR 指令数据通路图

（8） NOR

指令格式：XOR rd, rs, rt

指令功能：将通用寄存器中的 32 位数据 rs 与 rt 进行位或非运算产生一个 32 位数据存入目标寄存器 rd。

所需操作：

1. IMEM←PC （取指令）
2. Rd←Rs NOR Rt （执行指令）
3. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU	
				Rd	A	B
NOR	NPC	PC	PC	ALU	Rs	Rt

数据通路：

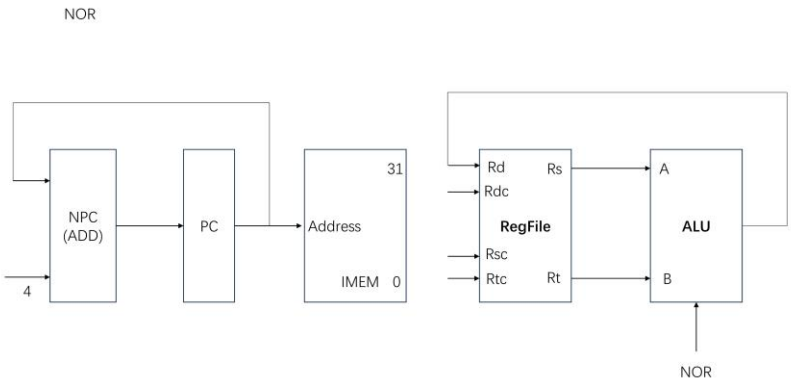


图 9. NOR 指令数据通路图

（9） SLT

指令格式：SLT rd，rs，rt

指令功能：将通用寄存器中的 32 位有符号数 rs 与 rt 进行大小比较，将比较结果存入目标寄存器 rd，如果 rs 小于 rt，则结果为 1，反之结果为 0。

所需操作：

1. IMEM←PC （取指令）
2. Rd←Rs < Rt （执行指令）
3. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU	
				Rd	A	B
SLT	NPC	PC	PC	ALU	RS	Rt

数据通路：

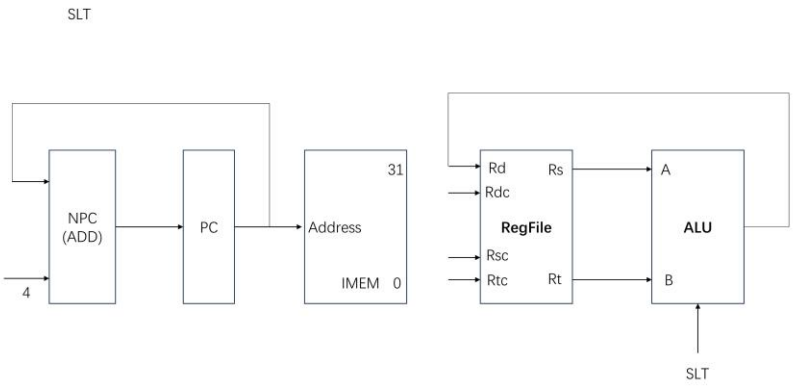


图 10. SLT 指令数据通路图

（10） SLTU

指令格式：SLTU rd，rs，rt

指令功能：将通用寄存器中的 32 位无符号数 rs 与 rt 进行大小比较，将比较结果存入目标寄存器 rd，如果 rs 小于 rt，则结果为 1，反之结果为 0。

所需操作：

1. IMEM←PC （取指令）
2. Rd←Rs < Rt （执行指令）
3. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU	
				Rd	A	B
SLTU	NPC	PC	PC	ALU	RS	Rt

数据通路：

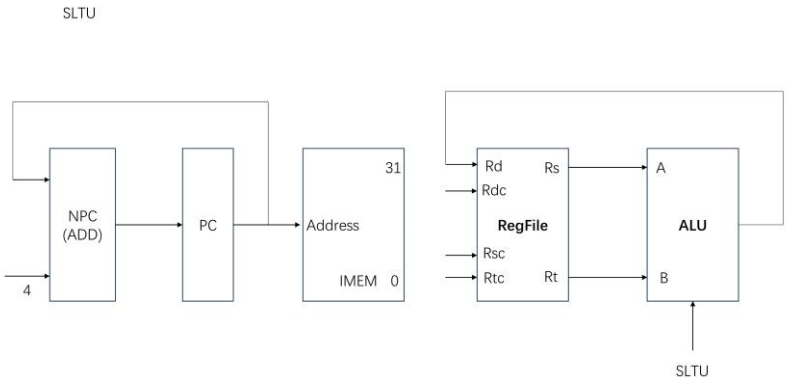


图 11. SLTU 指令数据通路图

（11） SLL

指令格式：SLL rd, rt, sa

指令功能：将通用寄存器 rt 的内容算数/逻辑左移 sa 位，空余出来的位置用 0 来填充，把结果存入 rd 寄存器。

所需操作：

1. IMEM←PC （取指令）
2. Ext_5←sa （数位扩展）
3. Rd←Rt 算数/逻辑左移 Ext_5 位（执行指令）
4. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU		Ext5
				Rd	A	B	
SLL	NPC	PC	PC	ALU	Ext5	Rt	shamt

数据通路：

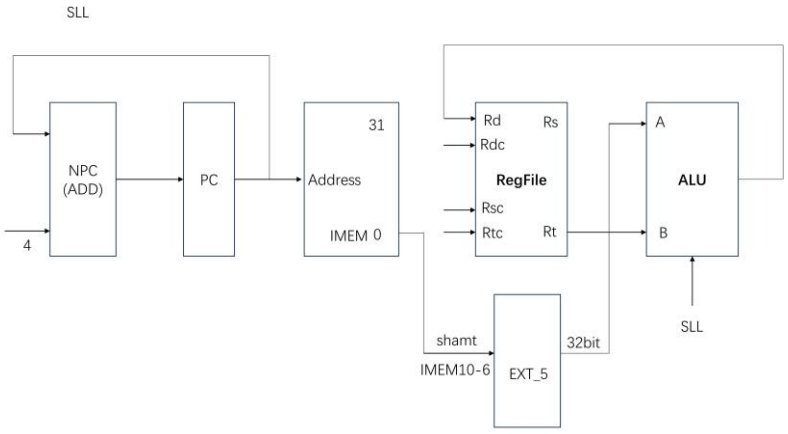


图 12. SLL 指令数据通路图

（12） SRL

指令格式：SRL rd, rt, sa

指令功能：将通用寄存器 rt 的内容逻辑右移 sa 位，空余出来的位置用 0 来填充，把结果存入 rd 寄存器。

所需操作：

1. IMEM←PC （取指令）
2. Ext_5←sa （数位扩展）
3. Rd←Rt 逻辑右移 Ext_5 位 （执行指令）
4. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU		Ext5
				Rd	A	B	
SRL	NPC	PC	PC	ALU	Ext5	Rt	shamt

数据通路：

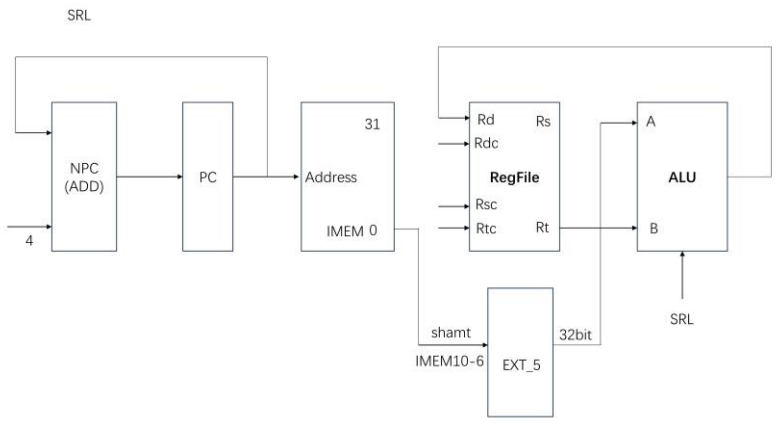


图 13. SRL 指令数据通路图

（13） SRA

指令格式：SRA rd, rt, sa

指令功能：将通用寄存器 rt 的内容算数右移 sa 位，空余出来的位置用符号位来填充，把结果存入 rd 寄存器。

所需操作：

1. IMEM←PC （取指令）
2. Ext_5←sa （数位扩展）
3. Rd←Rt 算数右移 Ext_5 位 （执行指令）
4. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU		Ext5
				Rd	A	B	
SRA	NPC	PC	PC	ALU	Ext5	Rt	shamt

数据通路：

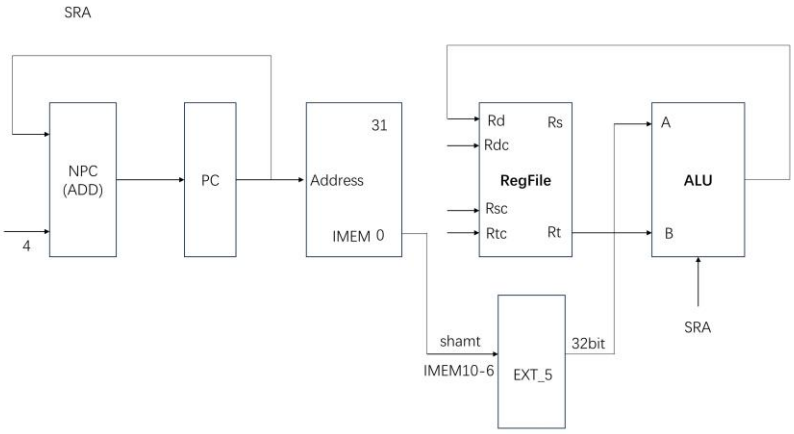


图 14. SRA 指令数据通路图

（14） SLLV

指令格式：SLLV rd, rt, rs

指令功能：将通用寄存器 rt 的内容算数/逻辑左移,移动的位数存储在 rs 寄存器中，空余出来的位置用 0 来填充，把结果存入 rd 寄存器。

所需操作：

1. IMEM←PC （取指令）
2. Rd←Rt 算数/逻辑左移 rs（执行指令）
3. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU	
				Rd	A	B
SLLV	NPC	PC	PC	ALU	Rs	Rt

数据通路：

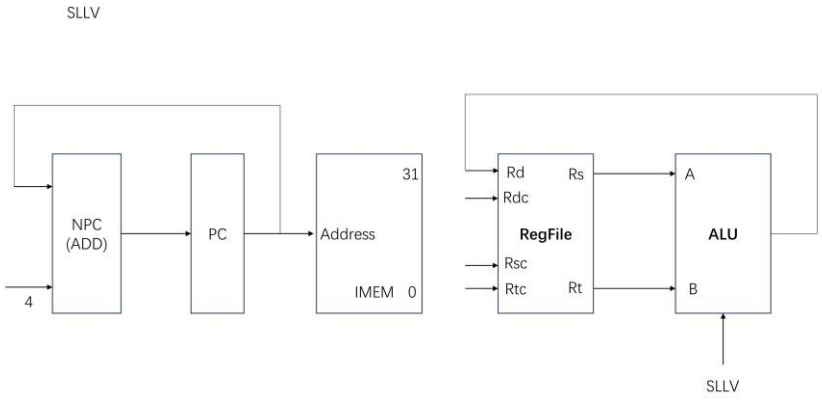


图 15. SLLV 指令数据通路图

（15） SRLV

指令格式：SRLV rd, rt, rs

指令功能：将通用寄存器 rt 的内容逻辑右移,移动的位置存储在 rs 寄存器中，空余出来的位置用 0 来填充，把结果存入 rd 寄存器。

所需操作：

1. IMEM←PC （取指令）
2. Rd←Rt 逻辑右移 rs（执行指令）
3. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU	
				Rd	A	B
SRLV	NPC	PC	PC	ALU	Rs	Rt

数据通路：

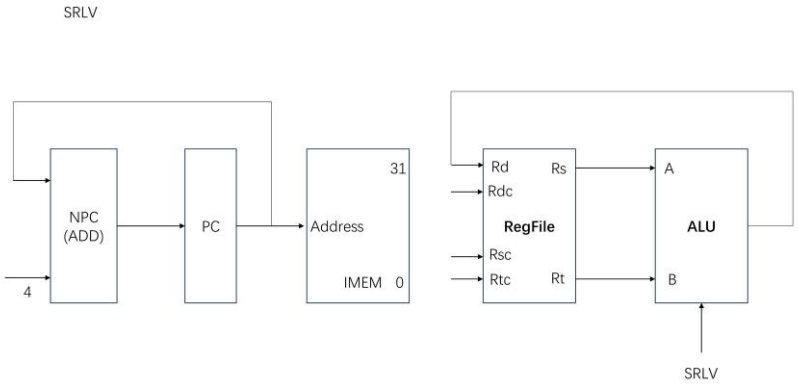


图 16. SRLV **指令数据通路图**

（16） SRAV

指令格式：SRAV rd, rt, rs

指令功能：将通用寄存器 rt 的内容算数右移,移动的位置存储在 rs 寄存器中，空余出来的位置符号位来填充，把结果存入 rd 寄存器。

所需操作：

1. IMEM←PC （取指令）
2. Rd←Rt 算数右移 rs（执行指令）
3. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU	
				Rd	A	B
SRAV	NPC	PC	PC	ALU	Rs	Rt

数据通路：

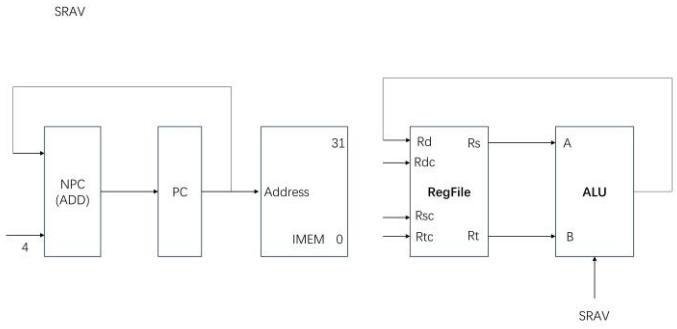


图 17. SRAV 指令数据通路图

（17） JR

指令格式：JR rs

指令功能：PC 寄存器跳转到 rs 寄存器中存储的地址。

所需操作：

1. IMEM←PC （取指令）
2. PC←rs （PC 转移）

所需部件与数据输入：

指令	PC	NPC	IMEM
JR	Rs	PC	PC

数据通路：

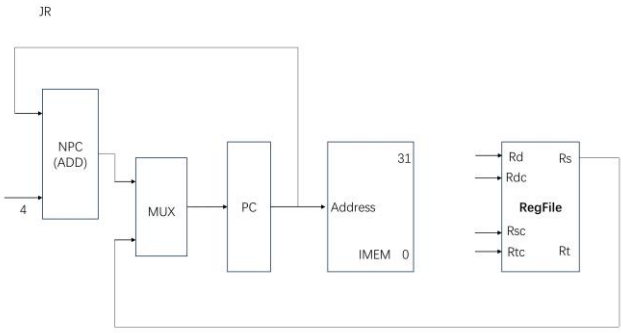


图 18. JR 指令数据通路图

（18） ADDI

指令格式：ADDI rt, rs, immediate

指令功能：16 位有符号立即数与通用寄存器 rs 中的 32 位数相加产生一个 32 位的数存入目标寄存器 rt。

所需操作：

1. IMEM←PC （取指令）

2. S_Ext_16←imme16 （数位扩展）
3. Rt←Rs+S_Ext_16 （执行指令）
4. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU		S_Ext16
				Rd	A	B	
ADDI	NPC	PC	PC	ALU	Rs	S_Ext16	imme16

数据通路：

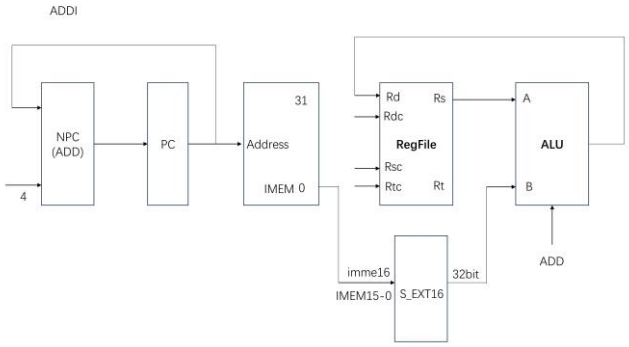


图 19. ADDI **指令数据通路图**

（19） ADDIU

指令格式: ADDIU rt, rs, immediate

指令功能：16 位有符号立即数与通用寄存器 rs 中的 32 位数相加产生一个 32 位的数存入目标寄存器 rt。

所需操作：

1. IMEM←PC （取指令）
2. S_Ext_16←imme16 （数位扩展）
3. Rt←Rs+S_Ext_16 （执行指令）
4. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU		S_Ext16
				Rd	A	B	
ADDIU	NPC	PC	PC	ALU	Rs	S_Ext16	imme16

数据通路：

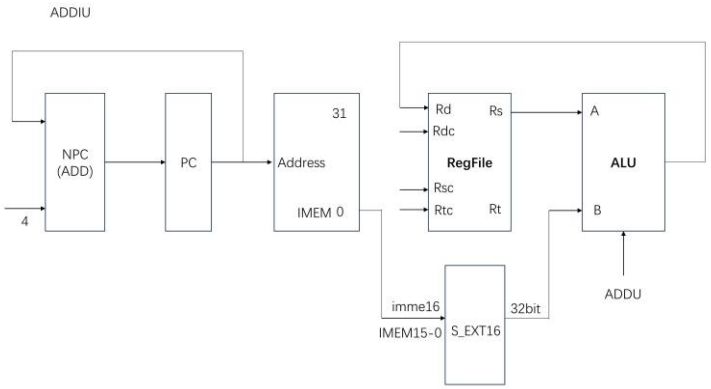


图 20. ADDIU 指令数据通路图

（20） ANDI

指令格式：ANDI rt, rs, immediate

指令功能：将 16 位立即数做 0 扩展后与通用寄存器 rs 中的 32 位数做位与运算，将结果存入目标寄存器 rt。

所需操作：

1. IMEM←PC （取指令）
2. U_Ext_16←imme16 （数位扩展）
3. Rt←Rs AND U_Ext_16 （执行指令）
4. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU		U_Ext16
				Rd	A	B	
ANDI	NPC	PC	PC	ALU	Rs	U_Ext16	imme16

数据通路：

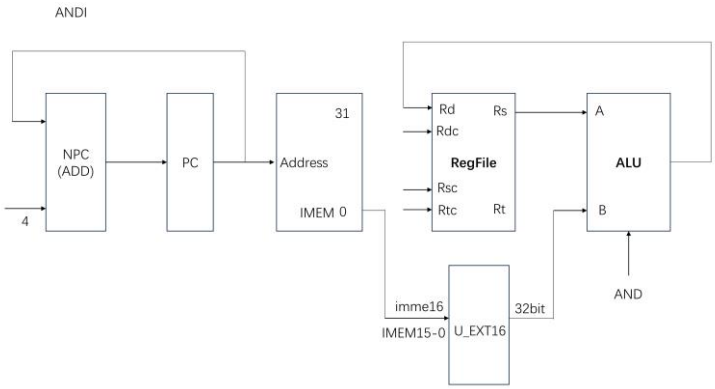


图 21. ANDI 指令数据通路图

（21） ORI

指令格式：ORI rt, rs, immediate

指令功能：将 16 位立即数做 0 扩展后与通用寄存器 rs 中的 32 位数做位或运算，将结果存入目标寄存器 rt。

所需操作：

1. IMEM←PC （取指令）
2. U_Ext_16←imme16 （数位扩展）
3. Rt←Rs OR U_Ext_16 （执行指令）
4. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU		U_Ext16
				Rd	A	B	
ORI	NPC	PC	PC	ALU	Rs	U_Ext16	imme16

数据通路：

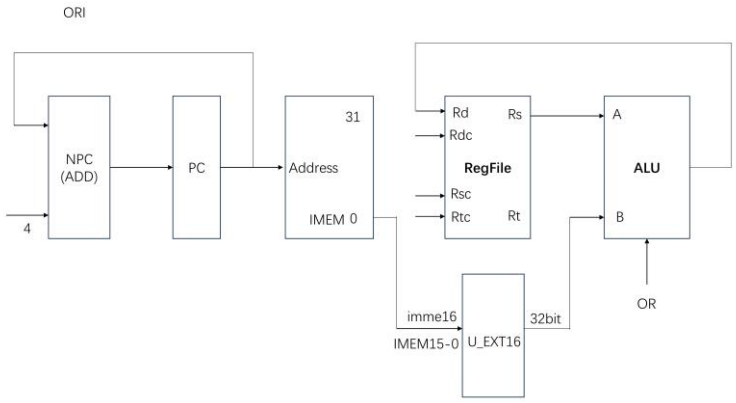


图 22. ORI **指令数据通路图**

（22） XORI

指令格式：XORI rt, rs, immediate

指令功能：将 16 位立即数做 0 扩展后与通用寄存器 rs 中的 32 位数做位异或运算，将结果存入目标寄存器 rt。

所需操作：

1. IMEM←PC （取指令）
2. U_Ext_16←imme16 （数位扩展）
3. Rt←Rs XOR U_Ext_16 （执行指令）
4. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU		U_Ext16
				Rd	A	B	
XORI	NPC	PC	PC	ALU	Rs	U_Ext16	imme16

数据通路：

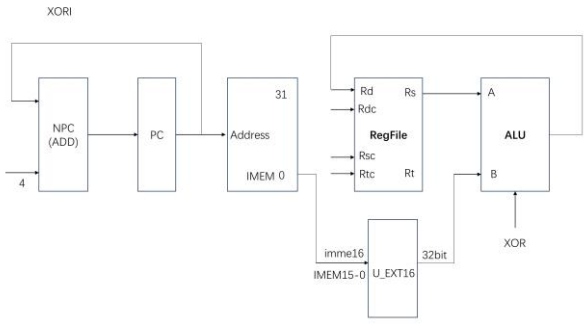


图 23. XORI 指令数据通路图

（23） LW

指令格式：LW rt, offset(base)

指令功能：将基地址（base）加偏移量（offset）所得到的有效数据存储器地址中的内容加载到通用寄存器 rt 中。

所需操作：

1. IMEM←PC （取指令）
2. S_Ext_16←imme16 （数位扩展）
3. Rt←[base + S_Ext_16]（执行指令）
4. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU		S_Ext16	DMEM	
				Rd	A	B		Addr	Data
LW	NPC	PC	PC	Data	Rs	S_Ext16	imme16	ALU	

数据通路：

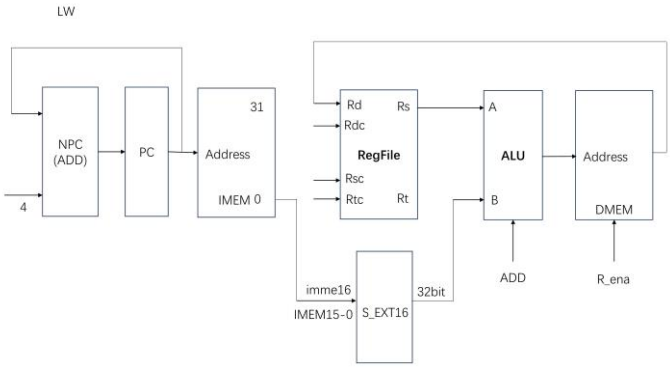


图 24. LW 指令数据通路图

（24） SW

指令格式：LW rt, offset(base)

指令功能：将通用寄存器 rt 中的 32 位数据存入数据存储器中的有效地址，有效地址由基地

址（base）和经过符号扩展的 16 位偏移量（offset）相加所得。

所需操作：

1. IMEM←PC （取指令）
2. S_Ext_16←imme16 （数位扩展）
3. [base + S_Ext_16]←Rt（执行指令）
4. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU		S_Ext16	DMEM	
				Rd	A	B		Addr	Data
LW	NPC	PC	PC		Rs	S_Ext16	imme16	ALU	Rt

数据通路：

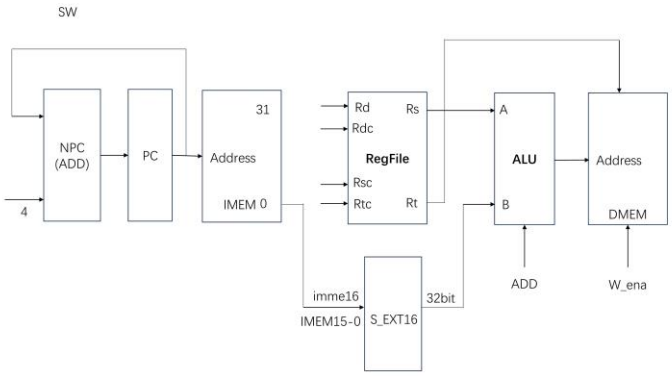


图 25. SW **指令数据通路图**

（25） BEQ

指令格式：BEQ rs，rt，offset

指令功能：比较通用寄存器的值，然后做 pc 相关的分支跳转。如果 rs = rt ，那么将 offset 左移两位，再进行符号扩展到 32 位与当前 pc 相加，形成有效转移地址，转到该地址。如果 rs != rt，则继续执行下条指令。

所需操作：

1. IMEM←PC （取指令）
2. S_Ext_18←imme16 || 02 （数位扩展）
3. PC←（rs==rt）? PC+S_Ext_18 : NPC （PC 转移）

所需部件与数据输入：

指令	PC	NPC	IMEM	ALU		Ext18	ADD	
				A	B		A	B
BEQ	NPC;ADD	PC	PC	Rs	Rt	Imm16||02	Ext18	NPC

数据通路：

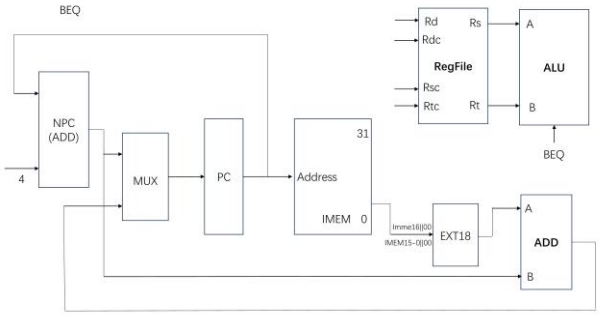


图 26. BEQ 指令数据通路图

（26） BNE

指令格式：BNE rs, rt, offset

指令功能：比较通用寄存器的值，然后做 pc 相关的分支跳转。如果 rs != rt ，那么将 offset 左移两位，再进行符号扩展到 32 位与当前 pc 相加，形成有效转移地址，转到该地址。如果 rs == rt，则继续执行下条指令。

所需操作：

1. IMEM←PC （取指令）
2. S_Ext_18←imme16 || 0^2 （数位扩展）
3. PC←（rs!=rt）? PC+S_Ext_18 : NPC （PC 转移）

所需部件与数据输入：

指令	PC	NPC	IMEM	ALU		Ext18	ADD	
				A	B		A	B
BNE	NPC;ADD	PC	PC	Rs	Rt	Imm16||02	Ext18	NPC

数据通路：

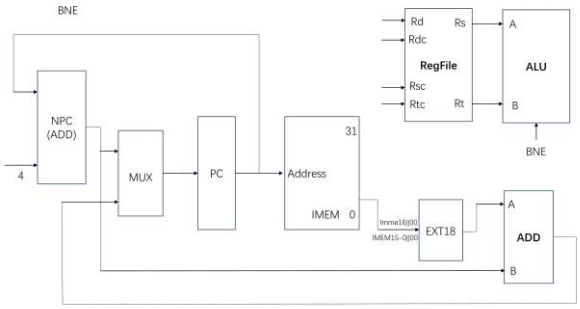


图 27. BNE 指令数据通路图

（27） SLTI

指令格式：SLTI rt，rs，immediate

指令功能：将通用寄存器中的 32 位有符号数 rs 与经符号扩展的 16 位立即数相比较，将比较结果存入目标寄存器 rd，如果 rs 小于 rt，则结果为 1，反之结果为 0。

所需操作：

1. IMEM←PC （取指令）
2. S_Ext_16←imme16 （数位扩展）
3. Rt←Rs < S_Ext_16 （执行指令）
4. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU		S_Ext16
				Rd	A	B	
SLTI	NPC	PC	PC	ALU	Rs	S_Ext16	imme16

数据通路：

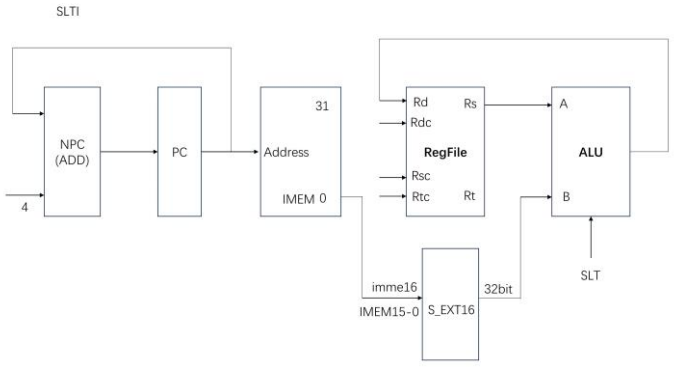


图 28. SLTI 指令数据通路图

（28） SLTIU

指令格式：SLTIU rt，rs，immediate

指令功能：将通用寄存器中的 32 位有符号数 rs 与经 0 扩展的 16 位立即数相比较，将比较结果存入目标寄存器 rd，如果 rs 小于 rt，则结果为 1，反之结果为 0。

所需操作：

1. IMEM←PC （取指令）
2. U_Ext_16←imme16 （数位扩展）
3. Rt←Rs < U_Ext_16 （执行指令）
4. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU		U_Ext16
				Rd	A	B	
SLTI	NPC	PC	PC	ALU	Rs	U_Ext16	imme16

数据通路：

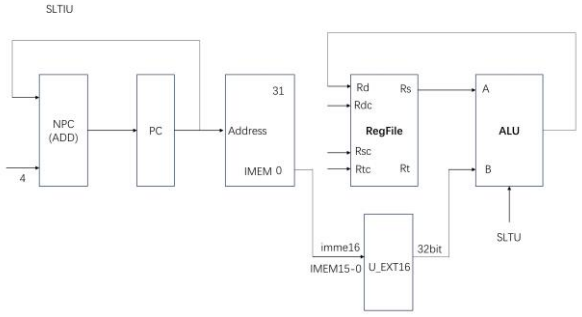


图 29. SLTI 指令数据通路图

（29） LUI

指令格式：LUI rt,immediate

指令功能：将一个 16 位的立即数载入到通用寄存器 rt 的高位，低 16 位补 0。

所需操作：

1. IMEM←PC （取指令）
2. U_Ext_16←imme16 （数位扩展）
3. Rt←LUI U_Ext_16 （执行指令）
4. PC←NPC （NPC 完成 PC+4 增值）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ALU		U_Ext16
				Rd	A	B	
LUI	NPC	PC	PC	ALU		U_Ext16	imme16

数据通路：

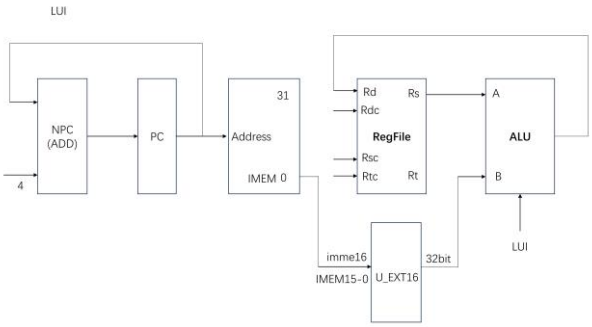


图 30. LUI 指令数据通路图

（30） J

指令格式：J target

指令功能：无条件跳转到一个绝对地址。

所需操作：

1. IMEM←PC （取指令）

2. PC←PC_{31-28} || instr_index || 0^2（PC 转移）

所需部件与数据输入：

指令	PC	NPC	IMEM	||	
				A	B
J	||	PC	PC	PC31-28	IMEM25-0<<2

数据通路：

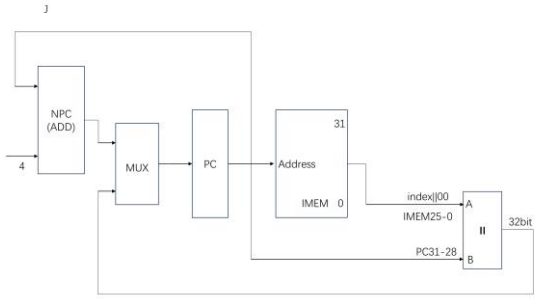


图 31. J 指令数据通路图

（31） JAL

指令格式：JAL target

指令功能：在跳转到指定地址执行子程序调用的同时，在 31 号寄存器中存放返回地址（当前地址后的第二条指令地址）。

所需操作：

1. IMEM←PC （取指令）
2. PC←PC_{31-28} || instr_index || 0^2（PC 转移）
3. Reg31←PC+8 （存放返回地址）

所需部件与数据输入：

指令	PC	NPC	IMEM	RegFile	ADD		||	
				Rd	A	B	A	B
JAL	||	PC	PC	ADD	PC	8	PC31-28	IMEM25-0

数据通路：

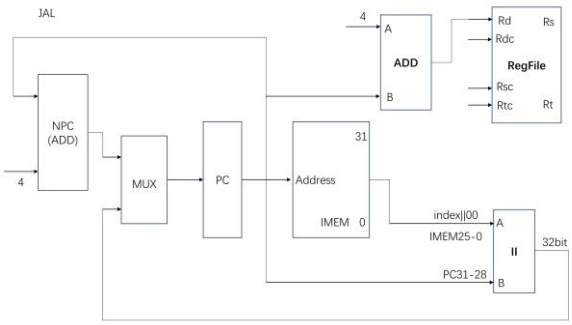


图 32. JAL 指令数据通路图

（32） 数据通路总图

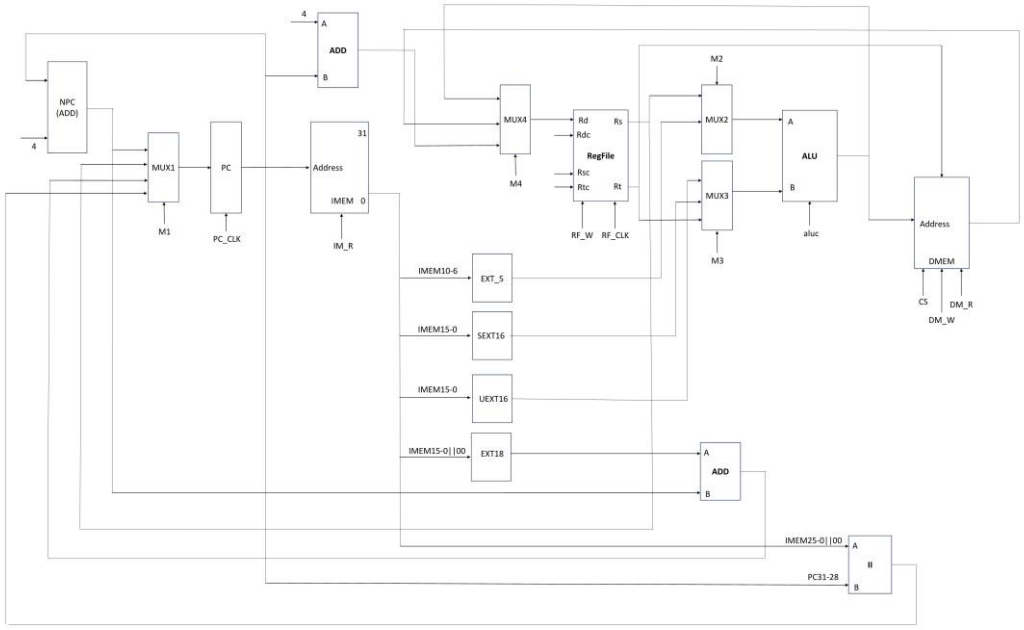


图 33. 数据通路总图

指令	PC	NPC	IMEM	RegFile	ALU		Ext5	U_Ext16	S_Ext16	Ext18	DMEM		ADD			||
				Rd	A	B					Addr	Data	A	B	A	B
R-type 寄存器类型算数运算																	
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											
位运算																	
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											
比较运算																	
SLT	NPC	PC	PC	ALU	RS	Rt											
SLTU	NPC	PC	PC	ALU	RS	Rt											
移位运算																	
SLL	NPC	PC	PC	ALU	Ext5	Rt	shamt										
SRL	NPC	PC	PC	ALU	Ext5	Rt	shamt										
SRA	NPC	PC	PC	ALU	Ext5	Rt	shamt										
SLLV	NPC	PC	PC	ALU	Rs	Rt											
SRLV	NPC	PC	PC	ALU	RS	Rt											
SRAV	NPC	PC	PC	ALU	RS	Rt											
跳转运算																	
JR	Rs	PC	PC														
I-type 立即数类型算数运算																	
ADDI	NPC	PC	PC	ALU	Rs	S_Ext16			imme16								
ADDIU	NPC	PC	PC	ALU	Rs	S_Ext16			imme16								
位运算																	
ANDI	NPC	PC	PC	ALU	Rs	U_Ext16		imme16									
ORI	NPC	PC	PC	ALU	Rs	U_Ext16		imme16									
XORI	NPC	PC	PC	ALU	Rs	U_Ext16		imme16									
存取																	
LW	NPC	PC	PC	Data	Rs	S_Ext16			imme16		ALU						
SW	NPC	PC	PC		Rs	S_Ext16			imme16		ALU	Rt					
条件跳转																	
BEQ	NPC:ADD	PC	PC		Rs	Rt				Imm16||02			Ext18	NPC			
BNE	NPC:ADD	PC	PC		Rs	Rt				Imm16||02			Ext18	NPC			
比较运算																	
SLTI	NPC	PC	PC	ALU	Rs	S_Ext16			imme16								
SLTIU	NPC	PC	PC	ALU	Rs	U_Ext16		imme16									
立即数载入寄存器																	
LUI	NPC	PC	PC	ALU		U_Ext16		imme16									
J-type 跳转类型																	
J	||	PC	PC												PC31-28	IMEM25-0	
JAL	||	PC	PC	ADD									PC	8	PC31-28	IMEM25-0	

图 34. 31 条 MIPS 指令部件使用与数据输入总图

三、模块建模

Top 模块

```
// CPU31 顶层
// 顶层模块 sccomp_dataflow(原 top 模块)
module sccomp_dataflow (
    input clk_in,
    input reset,
    output [31:0] inst,
    output [31:0] pc
    // 用于七段数码管下板展示
//    output [7:0] o_seg,
//    output [7:0] o_sel
);

wire get;              // 获取数据使能信号
wire pull;             // 存储数据使能信号
wire [31:0] data_get;  // 从 DMEM 中获取的数据
wire [31:0] data_pull; // 存入 DMDM 中的数据

wire [31:0] dm_addr;   // DMEM 的访问地址
wire [31:0] im_addr;   // IMEM 的访问地址

wire [31:0] inst_code; // 从 IMEN 中获取的指令代码
wire [31:0] PCReg;     // PC 寄存器中的值
wire [31:0] alures;    // ALU 的计算结果

assign inst = inst_code;
assign pc = PCReg;

assign im_addr = (PCReg - 32'h0040_0000) / 4;

//// 核心部件
IMEM imemory(clk_in, 1'b1, im_addr[10:0], inst_code);
DMEM dmemory(clk_in, 1'b1, get, pull, dm_addr, data_pull, data_get);
CPU sccpu(clk_in, reset, 1'b1, inst_code, dm_addr, data_get,
data_pull, get, pull, PCReg, alures);

//// 下板测试需要进行分频
//// 下板测试时将 IMEM\DMEM\CPU 的时钟输入改为 clk_test
//// 以下代码用于下板测试
//wire clk_test;
//clk_divider divider(clk_in, reset, clk_test);
//IMEM imemory(clk_test, 1'b1, im_addr[10:0], inst_code);
//DMEM dmemory(clk_test, 1'b1, get, pull, dm_addr, data_pull,
data_get);
```

```
//CPU sccpu(clk_test, reset, 1'b1, inst_code, dm_addr, data_get,
data_pull, get, pull, PCReg, alures);
//seg7x16 show_pc(clk_in, reset, 1'b1, PCReg, o_seg, o_sel); // 数
码管展示

endmodule
```

IEME 模块

```
// 指令存储器
module IMEM (
    input clk,
    input ena,
    input [10:0] addr,
    output wire [31:0] inst_code
);

// 根据 pc 的值取指令
// 地址发生变化时改变 instruction_code

// 未使用 IP 核
//reg    [31:0] memory [1023:0];
//assign inst_code = memory[addr];

//initial begin
//  $readmemh("D:/test/h_02_addiu.txt", memory);
//end

// 使用 IP 核
dist_mem_gen_0 imemory (
  .a(addr),      // input wire [10 : 0] a
  .spo(inst_code)  // output wire [31 : 0] spo
);

endmodule
```

DMEM 模块

```
// 数据存储器
module DMEM (
    input clk,
    input ena,
    input data_get_ena,
    input data_pull_ena,
    input [31:0] dm_addr,
    input [31:0] data_in,
```

```
    output [31:0] data_out
);

// 有数据存储器参与的指令
parameter LW = 23;
parameter SW = 24;

reg [31:0] mem [0:31];

always@(posedge clk) begin
    if(ena==1 && data_pull_ena==1) begin
        mem[dm_addr] <= data_in;
    end
end

assign data_out = (ena==1 && data_get_ena==1) ? mem[dm_addr] : 32'bz;

endmodule
```

CPU 模块

```
module CPU (
    input clk,
    input reset,
    input ena,
    input [31:0] inst_code,
    output wire [31:0] dm_addr,
    input [31:0] data_get,
    output wire [31:0] data_pull,
    output wire data_get_ena,
    output wire data_pull_ena,
    output reg [31:0] PC,
    output [31:0] alures
);
parameter UNDEFINED = 0;
// R-type 类型指令
parameter ADD = 1;
parameter ADDU = 2;
parameter SUB = 3;
parameter SUBU = 4;
parameter AND = 5;
parameter OR = 6;
parameter XOR = 7;
parameter NOR = 8;
parameter SLT = 9;
```

```
parameter SLTU = 10;
parameter SLL = 11;
parameter SRL = 12;
parameter SRA = 13;
parameter SLLV = 14;
parameter SRLV = 15;
parameter SRAV = 16;
parameter JR = 17;
// I-type 类型指令
parameter ADDI = 18;
parameter ADDIU = 19;
parameter ANDI = 20;
parameter ORI = 21;
parameter XORI = 22;
parameter LW = 23;
parameter SW = 24;
parameter BEQ = 25;
parameter BNE = 26;
parameter SLTI = 27;
parameter SLTIU = 28;
parameter LUI = 29;
// J-type 类型指令
parameter J = 30;
parameter JAL = 31;

wire [5:0] inst_type;
wire [4:0] shamt;
wire [15:0] imme16;
wire [25:0] index;
wire [4:0] Rdc, Rsc, Rtc; // 寄存器地址
wire [31:0] Rd, Rs, Rt;   // 寄存器数值
wire RF_R = 1'b1;
wire RF_W = ~(inst_type == JR || inst_type == SW || inst_type == BEQ
|| inst_type == BNE || inst_type == J);
wire [31:0] alu_A, alu_B, alu_res;
wire ZF, CF, SF, OF; // ALU 操作标志位
wire [3:0] aluc;     // ALU 操作控制位
wire [31:0] NPC;
assign aluc[0] = (inst_type == SUB || inst_type == SUBU ||
                  inst_type == OR || inst_type == ORI ||
                  inst_type == NOR || inst_type == SLT ||
                  inst_type == SLTI || inst_type == SRL ||
                  inst_type == SRLV || inst_type == BNE ||
                  inst_type == BEQ);
```

```
assign aluc[1] = (inst_type == ADD || inst_type == ADDI ||
                  inst_type == SUBU || inst_type == XOR ||
                  inst_type == XORI || inst_type == NOR ||
                  inst_type == SLT || inst_type == SLTI ||
                  inst_type == SLTU || inst_type == SLTIU ||
                  inst_type == SLL || inst_type == SLLV ||
                  inst_type == LW || inst_type == SW ||
                  inst_type == BNE || inst_type == BEQ);
assign aluc[2] = (inst_type == AND || inst_type == ANDI ||
                  inst_type == OR || inst_type ==ORI ||
                  inst_type == XOR || inst_type == XORI ||
                  inst_type == NOR || inst_type == SRA ||
                  inst_type == SRAV || inst_type == SLL ||
                  inst_type == SLLV || inst_type == SRL ||
                  inst_type == SRLV);
assign aluc[3] = (inst_type == LUI || inst_type == SLT ||
                  inst_type == SLTI ||inst_type == SLTU ||
                  inst_type == SLTIU || inst_type == SRA ||
                  inst_type == SRAV || inst_type == SLL ||
                  inst_type == SLLV || inst_type == SRL ||
                  inst_type == SRLV);
assign Rd = (inst_type == LW) ? data_get :
            (inst_type == JAL) ? NPC :
            alu_res;
InstDecoder decoder(inst_code, inst_type, Rsc, Rtc, Rdc, shamt,
imme16, index);
RegFile cpu_ref(clk, reset, RF_R, RF_W, Rdc, Rsc, Rtc, Rd, Rs, Rt);



/*****************************************************************
                数位扩展器（指令类型发生变化时改变）
*****************************************************************/
wire [31:0] Ext_5;
wire Ext_5_ena;
assign Ext_5_ena = (inst_type == SLL || inst_type == SRL || inst_type
== SRA);
assign Ext_5 = (Ext_5_ena == 1) ? {27'b0, shamt} : 32'bz;
wire [31:0] signed_Ext_16;
wire signed_Ext_16_ena;
assign signed_Ext_16_ena = (inst_type == LW || inst_type == SW ||
inst_type == ADDIU || inst_type == ADDI || inst_type == LW || inst_type
== SW || inst_type == SLTI);
assign signed_Ext_16 = (signed_Ext_16_ena == 1) ? { { 16{imme16[15]} },
imme16} : 32'bz;
```

```
wire [31:0] zero_Ext_16;
wire zero_Ext_16_ena;
assign zero_Ext_16_ena = (inst_type == ANDI || inst_type == ORI ||
inst_type == XORI || inst_type == SLTIU || inst_type == LUI);
assign zero_Ext_16 = (zero_Ext_16_ena == 1) ? { 16'b0, imme16 } : 32'bz;
wire [31:0] Ext_18;
wire Ext_18_ena;
assign Ext_18_ena = (inst_type == BNE || inst_type == BEQ);
assign Ext_18 = (Ext_18_ena == 1) ? { { 14{imme16[15]} }, imme16, 2'b0} :
32'bz;

/*****************************************************************
                    PC 寄存器部分（时钟下降沿改变）
*****************************************************************/
// PC 的数据来源有 4 个:1.NPC;2.JR;3.J\JAL;3.BNE\BEQ(Ext_18)
wire PC1, PC2;
assign NPC = PC + 4;
assign PC1 = (inst_type == JR);
assign PC2 = (inst_type == J || inst_type == JAL);
assign PC3 = ((inst_type == BNE && alu_res != 0) || (inst_type == BEQ
&& alu_res == 0));
always@(negedge clk or posedge reset) begin
    if(reset) begin
        PC <= 32'h0040_0000;
    end
    else if(PC1) begin
        PC <= Rs;
    end
    else if(PC2) begin
        PC <= {PC[31:28], index, 2'b0};
    end
    else if(PC3) begin
        PC <= PC + 4 + Ext_18;
    end
    else begin
        PC <= NPC;
    end
end

/*****************************************************************
                        LW 和 SW 指令数据处理
*****************************************************************/
// LW 和 SW 指令
// 该指令由基址和偏移量组成：基址存储在寄存器中，偏移量是符号扩展的立即数
```

```
// lw 指令 Rs<-mem[addr]
// sw 指令 mem[addr]<-Rs
assign data_get_ena = (inst_type == LW);
assign data_pull_ena = (inst_type == SW);
assign data_pull = (data_pull_ena == 1) ? Rt : 32'bz;
assign dm_addr = (data_get_ena==1 || data_pull_ena==1) ? (alu_res -
32'h1001_0000) / 4 : 32'bz;

/*****************************************************************
            ALU 组件（操作数或者控制信号改变时进行计算）
*****************************************************************/
assign alu_A = Ext_5_ena ? Ext_5 : Rs;
assign alu_B = (signed_Ext_16_ena == 1) ? signed_Ext_16 :
(zero_Ext_16_ena == 1) ? zero_Ext_16 : Rt;
ALU alu(alu_A, alu_B, aluc, alu_res, ZF, CF, SF, OF);
assign alures = alu_res;

endmodule
```

InstDecoder 模块

```
// 指令译码器
module InstDecoder (
    input [31:0] inst_code,
    output reg [5:0] inst_type,
    output reg [4:0] Rsc,
    output reg [4:0] Rtc,
    output reg [4:0] Rdc,
    output reg [4:0] shamt,
    output reg [15:0] imme16,
    output reg [25:0] index
);

parameter UNDEFINED = 0;
// R-type 指令
parameter ADD = 1;
parameter ADDU = 2;
parameter SUB = 3;
parameter SUBU = 4;
parameter AND = 5;
parameter OR = 6;
parameter XOR = 7;
parameter NOR = 8;
parameter SLT = 9;
parameter SLTU = 10;
```

```
parameter SLL = 11;
parameter SRL = 12;
parameter SRA = 13;
parameter SLLV = 14;
parameter SRLV = 15;
parameter SRAV = 16;
parameter JR = 17;
// I-type 指令
parameter ADDI = 18;
parameter ADDIU = 19;
parameter ANDI = 20;
parameter ORI = 21;
parameter XORI = 22;
parameter LW = 23;
parameter SW = 24;
parameter BEQ = 25;
parameter BNE = 26;
parameter SLTI = 27;
parameter SLTIU = 28;
parameter LUI = 29;
// J-type 指令
parameter J = 30;
parameter JAL = 31;

// 当 instruction_code 发生变化时开始指令译码
always@(*) begin
    // R-type 类型指令
    if(inst_code[31:26] == 6'b000000) begin
        case(inst_code[5:0])
            6'b100_000: inst_type <= ADD;// 有符号加法
            6'b100_001: inst_type <= ADDU;// 无符号加法
            6'b100_010: inst_type <= SUB;// 有符号减法
            6'b100_011: inst_type <= SUBU;// 无符号减法
            6'b100_100: inst_type <= AND;// and
            6'b100_101: inst_type <= OR;// or
            6'b100_110: inst_type <= XOR;// xor
            6'b100_111: inst_type <= NOR;// nor
            6'b101_010: inst_type <= SLT;// slt
            6'b101_011: inst_type <= SLTU;// sltu
            6'b000_000: inst_type <= SLL;// sll
            6'b000_010: inst_type <= SRL;// srl
            6'b000_011: inst_type <= SRA;// sra
            6'b000_100: inst_type <= SLLV;// sllv
            6'b000_110: inst_type <= SRLV;// srlv
```

```
            6'b000_111: inst_type <= SRAV;// srav
            6'b001_000: inst_type <= JR;// jr
            default:    inst_type <= UNDEFINED;// 其他情况未定义
        endcase
        Rsc <= inst_code[25:21];
        Rtc <= inst_code[20:16];
        Rdc <= inst_code[15:11];
        shamt <= inst_code[10:6];
    end
    else if(inst_code[31:28] == 4'b0000) begin
        case(inst_code[27:26])
            // J-type 指令
            2'b10: inst_type <= J; // j
            2'b11: begin
                inst_type <= JAL; // jal
                Rdc <= 31;
            end
            default: inst_type <= UNDEFINED; // 其他类型未定义
        endcase
        index <= inst_code[25:0];
    end
    else begin
        case(inst_code[31:26])
        // I-type 指令
            6'b001_000: inst_type <= ADDI;// addi
            6'b001_001: inst_type <= ADDIU;// addiu
            6'b001_100: inst_type <= ANDI;// andi
            6'b001_101: inst_type <= ORI;// ori
            6'b001_110: inst_type <= XORI;// xori
            6'b100_011: inst_type <= LW;// lw
            6'b101_011: inst_type <= SW;// sw
            6'b000_100: inst_type <= BEQ;// beq
            6'b000_101: inst_type <= BNE;// bne
            6'b001_010: inst_type <= SLTI;// slti
            6'b001_011: inst_type <= SLTIU; // sltiu
            6'b001_111: inst_type <= LUI; // lui
            default: inst_type <= UNDEFINED; // 其他类型未定义
        endcase
        if(inst_code[31:26] == 6'b101_011 || inst_code[31:26] ==
6'b000_100 || inst_code[31:26] == 6'b000_101) begin
        // sw 指令与其他的 I-type 指令的寄存器使用情况有些不同：
        // BNE 和 BEQ 也不同
            Rsc <= inst_code[25:21];
            Rtc <= inst_code[20:16];
```

```
            imme16 <= inst_code[15:0];
        end
        else begin
            Rsc <= inst_code[25:21];
            Rdc <= inst_code[20:16];
            imme16 <= inst_code[15:0];
        end

    end
end



endmodule
```

RegFile 模块

```
module RegFile (
    input clk,
    input reset,
    input RF_R,
    input RF_W,
    input [4:0] Rdc,
    input [4:0] Rsc,
    input [4:0] Rtc,
    input [31:0] Rd,
    output [31:0] Rs,
    output [31:0] Rt
);
reg [31:0] registers [31:0]; // 32 个寄存器

/// 时钟上升沿将获得的数据写入寄存器中
always @(posedge clk or posedge reset) begin
    if(reset) begin
        // 寄存器初始化
        registers[0] <= 32'b0;
        registers[1] <= 32'b0;
        registers[2] <= 32'b0;
        registers[3] <= 32'b0;
        registers[4] <= 32'b0;
        registers[5] <= 32'b0;
        registers[6] <= 32'b0;
        registers[7] <= 32'b0;
        registers[8] <= 32'b0;
        registers[9] <= 32'b0;
        registers[10] <= 32'b0;
```

```
        registers[11] <= 32'b0;
        registers[12] <= 32'b0;
        registers[13] <= 32'b0;
        registers[14] <= 32'b0;
        registers[15] <= 32'b0;
        registers[16] <= 32'b0;
        registers[17] <= 32'b0;
        registers[18] <= 32'b0;
        registers[19] <= 32'b0;
        registers[20] <= 32'b0;
        registers[21] <= 32'b0;
        registers[22] <= 32'b0;
        registers[23] <= 32'b0;
        registers[24] <= 32'b0;
        registers[25] <= 32'b0;
        registers[26] <= 32'b0;
        registers[27] <= 32'b0;
        registers[28] <= 32'b0;
        registers[29] <= 32'b0;
        registers[30] <= 32'b0;
        registers[31] <= 32'b0;
    end
    else begin
        // 将结果写入寄存器 Rd
        if(RF_W && Rdc != 5'b0)
            registers[Rdc] <= Rd;
    end
end

assign Rs = RF_R ? registers[Rsc] : 32'bz;
assign Rt = RF_R ? registers[Rtc] : 32'bz;

endmodule
```

ClkDivider 模块

```
// 时钟信号分频器,用于下板测试
module clk_divider(I_CLK, rst, O_CLK);
input I_CLK;         //输入时钟信号，上升沿有效
input rst;           //复位信号，高电位有效
output reg O_CLK;    //输出时钟
//分频器的默认分频倍数是 20
parameter Magnification = 99_999_999;
reg [31:0]counter = 0;
initial begin
```

```
O_CLK = 0;
end

always@(posedge I_CLK) begin
    //复位信号
    if(rst == 1) begin
        counter = 0;
        O_CLK = 0;
    end
    //计数 CLk
    else begin
        counter = counter + 1'b1;
        if(counter == Magnification / 2 || counter == Magnification)
begin
            if(O_CLK == 0)
                O_CLK = 1;
            else if(O_CLK == 1)
                O_CLK = 0;
            if(counter == Magnification) begin
                counter = 0;
            end
        end
    end
end
endmodule
```

Seg7x16 模块

```
`timescale 1ns / 1ps
// 七段数码管,用于下板展示
module seg7x16(
    input clk,
    input reset,
    input cs,
    input [31:0] i_data,
    output [7:0] o_seg,
    output [7:0] o_sel
);

    reg [14:0] cnt;
    always @ (posedge clk, posedge reset)
        if (reset)
            cnt <= 0;
        else
            cnt <= cnt + 1'b1;
```

```
    wire seg7_clk = cnt[14];

    reg [2:0] seg7_addr;

always @ (posedge seg7_clk, posedge reset)
    if(reset)
        seg7_addr <= 0;
    else
        seg7_addr <= seg7_addr + 1'b1;

    reg [7:0] o_sel_r;

always @ (*)
    case(seg7_addr)
        7 : o_sel_r = 8'b01111111;
        6 : o_sel_r = 8'b10111111;
        5 : o_sel_r = 8'b11011111;
        4 : o_sel_r = 8'b11101111;
        3 : o_sel_r = 8'b11110111;
        2 : o_sel_r = 8'b11111011;
        1 : o_sel_r = 8'b11111101;
        0 : o_sel_r = 8'b11111110;
    endcase

reg [31:0] i_data_store;
always @ (posedge clk, posedge reset)
    if(reset)
        i_data_store <= 0;
    else if(cs)
        i_data_store <= i_data;

reg [7:0] seg_data_r;
always @ (*)
    case(seg7_addr)
        0 : seg_data_r = i_data_store[3:0];
        1 : seg_data_r = i_data_store[7:4];
        2 : seg_data_r = i_data_store[11:8];
        3 : seg_data_r = i_data_store[15:12];
        4 : seg_data_r = i_data_store[19:16];
        5 : seg_data_r = i_data_store[23:20];
        6 : seg_data_r = i_data_store[27:24];
        7 : seg_data_r = i_data_store[31:28];
    endcase
```

```
    reg [7:0] o_seg_r;
    always @ (posedge clk, posedge reset)
        if(reset)
            o_seg_r <= 8'hff;
        else
            case(seg_data_r)
                4'h0 : o_seg_r <= 8'hC0;
                4'h1 : o_seg_r <= 8'hF9;
                4'h2 : o_seg_r <= 8'hA4;
                4'h3 : o_seg_r <= 8'hB0;
                4'h4 : o_seg_r <= 8'h99;
                4'h5 : o_seg_r <= 8'h92;
                4'h6 : o_seg_r <= 8'h82;
                4'h7 : o_seg_r <= 8'hF8;
                4'h8 : o_seg_r <= 8'h80;
                4'h9 : o_seg_r <= 8'h90;
                4'hA : o_seg_r <= 8'h88;
                4'hB : o_seg_r <= 8'h83;
                4'hC : o_seg_r <= 8'hC6;
                4'hD : o_seg_r <= 8'hA1;
                4'hE : o_seg_r <= 8'h86;
                4'hF : o_seg_r <= 8'h8E;
            endcase

    assign o_sel = o_sel_r;
    assign o_seg = o_seg_r;

endmodule
```

ALU 模块

```
// 算数逻辑运算部件 ALU
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 [63:0] a_ext, b_ext, r_ext;
```

```
always @ (a or b or aluc) begin
    a_ext = {32'b0, a[31:0]};
    b_ext = {32'b0, b[31:0]};
    case(aluc)
        4'b0000: begin
        // 1. Addu 无符号加法 r=a+b
            r_ext = a_ext + b_ext;
            if( r_ext[63:32] > 0 )
                carry = 1;
            else
                carry = 0;
            r = r_ext[31:0];
            if(r == 0)
                zero = 1;
            else
                zero = 0;
            if(r[31] == 0)
                negative = 0;
            else if(r[31] == 1)
                negative = 1;
        end

        4'b0010: begin
        // 2. Add 有符号加法 r=a+b
            r = a + b;
            if(a[31]==0 && b[31]==0) begin //两个正数相加:
                if(r[31] == 1)
                    overflow = 1;
                else
                    overflow = 0;
            end
            else if(a[31]==1 && b[31]==1) begin //两个负数相加
                if(r[31] == 0)
                    overflow = 1;
                else
                    overflow = 0;
            end
            else
                overflow = 0;
            if(r == 0)
                zero = 1;
            else
                zero = 0;
            if(r[31] == 0)
```

```
            negative = 0;
        else if(r[31] == 1)
            negative = 1;
    end

    4'b0001: begin
    // 3. Subu 无符号减法 r=a-b
        if(a < b)
            carry = 1;
        else
            carry = 0;
        r = a - b;
        if(r == 0)
            zero = 1;
        else
            zero = 0;
        if(r[31] == 0)
            negative = 0;
        else if(r[31] == 1)
            negative = 1;
    end

    4'b0011: begin
    // 4. Sub 有符号减法 r=a-b
        r = a - b;
        if(a[31]== 0 && b[31]== 1) begin //正数减负数得到负数
            if(r[31] == 1)
                overflow = 1;
            else
                overflow = 0;
        end
        else if(a[31]== 1 && b[31]== 0) begin //负数减正数得到正数
            if(r[31] == 0)
                overflow = 1;
            else
                overflow = 0;
        end
        else
            overflow = 0;
        if(r == 0)
            zero = 1;
        else
            zero = 0;
        if(r[31] == 0)
```

```
            negative = 0;
        else if(r[31] == 1)
            negative = 1;
    end

    4'b0100: begin
    // 5. And 与运算 r=a&b
        r = a & b;
        if(r == 0)
            zero = 1;
        else
            zero = 0;
        if(r[31] == 0)
            negative = 0;
        else if(r[31] == 1)
            negative = 1;
    end

    4'b0101: begin
    // 6. Or 或运算 r=a|b
        r = a| b;
        if(r == 0)
            zero = 1;
        else
            zero = 0;
        if(r[31] == 0)
            negative = 0;
        else if(r[31] == 1)
            negative = 1;
    end

    4'b0110: begin
    // 7. Xor 异或运算 r=a^b
        r = a ^ b;
        if(r == 0)
            zero = 1;
        else
            zero = 0;
        if(r[31] == 0)
            negative = 0;
        else if(r[31] == 1)
            negative = 1;
    end
```

```
        4'b0111: begin
        // 8. Nor 或非运算 r=~(a|b)
            r = ~(a | b);
            if(r == 0)
                zero = 1;
            else
                zero = 0;
            if(r[31] == 0)
                negative = 0;
            else if(r[31] == 1)
                negative = 1;
        end

        4'b1000, 4'b1001: begin
        // 9. Lui 扩展运算 r={b[15:0],16'b0}
            r = { b[15:0], 16'b0 };
            if(r == 0)
                zero = 1;
            else
                zero = 0;
            if(r[31] == 0)
                negative = 0;
            else if(r[31] == 1)
                negative = 1;
        end

        4'b1011: begin
        // 10. Slt 有符号数比较 r=(a<b)?1:0
            if(a[31]==0 && b[31]==0) begin //两个正数相比较
                if(a < b) begin
                    r = 1;
                    negative = 1;
                    zero = 0;
                end
                else if(a > b)begin
                    r = 0;
                    negative = 0;
                    zero = 0;
                end
                else if(a == b) begin
                    r = 0;
                    negative = 0;
                    zero = 1;
                end
```

```
            end
            else if(a[31]==1 && b[31]==1) begin //两个负数相比较
                if(a < b) begin
                    r = 1;
                    negative = 1;
                    zero = 0;
                end
                else if(a > b) begin
                    r = 0;
                    negative = 0;
                    zero = 0;
                end
                else if(a == b) begin
                    r = 0;
                    negative = 0;
                    zero = 1;
                end
            end
            else begin //一个正数和一个负数比较
                if(a[31] == 1 && b[31] == 0) begin
                    r = 1;
                    negative = 1;
                    zero = 0;
                end
                else if(a[31] == 0 && b[31] == 1) begin
                    r = 0;
                    negative = 0;
                    zero = 0;
                end
            end
        end

        4'b1010: begin
        // 11. Sltu 无符号数比较 r=(a<b)?1:0
            if(a < b) begin
                r = 1;
                carry = 1;
                zero = 0;
            end
            else if(a > b)begin
                r = 0;
                carry = 0;
                zero = 0;
            end
```

```
        else if(a == b) begin
            r = 0;
            carry = 0;
            zero = 1;
        end
        if(r[31] == 0)
            negative = 0;
        else if(r[31] == 1)
            negative = 1;
    end

    4'b1100: begin
    // 12. Sra 算数右移 r=b>>>a
        r = b >> a - 1;
        if(r[0] == 1)
            carry=1;
        else if(r[0] == 0)
            carry=0;
        r = ({ 32{b[31]} } << ( 6'd32 - a ) ) | ( b >> a );
        if(r == 0)
            zero = 1;
        else
            zero = 0;
        if(r[31] == 0)
            negative = 0;
        else if(r[31] == 1)
            negative = 1;
    end

    4'b1110, 4'b1111: begin
    // 13. Sll\Sla 逻辑左移和算数左移 r=b<<a
        r = b << a - 1;
        if(r[31] == 1)
            carry = 1;
        else if(r[31] == 0)
            carry = 0;
        r = b << a;
        if(r == 0)
            zero = 1;
        else
            zero = 0;
        if(r[31] == 0)
            negative = 0;
        else if(r[31] == 1)
```

```
                negative = 1;
        end

        4'b1101: begin
        // 14. Srl 逻辑右移 r=b>>a
            r = b >> a - 1;
            if(r[0] == 1)
                carry=1;
            else if(r[0] == 0)
                carry=0;
            r = b >> a;
            if(r == 0)
                zero = 1;
            else
                zero = 0;
            if(r[31] == 0)
                negative = 0;
            else if(r[31] == 1)
                negative = 1;
        end

    endcase

end
endmodule
```

四、测试模块建模

前仿真功能测试模块代码:

```
`timescale 1ns / 1ps
module testbench();
reg clk, reset;
integer file_output;
integer counter = 0;

wire [31:0] pc, inst;
assign pc = cputb.PCReg;
assign inst = cputb.inst_code;

wire [7:0] o_seg, o_sel;

initial begin
```

```
    file_output = $fopen("D:/test/CPU31res.txt", "w");
    clk = 0;
    reset = 1;
    # 50;
    reset = 0;
end
always begin
    #50;
    clk = ~clk;
    // 时钟下降沿将数据打印
    if(clk == 1'b0) begin
        counter = counter + 1;
        $fdisplay(file_output, "pc: %h", pc);
        $fdisplay(file_output, "instr: %h", inst);
        $fdisplay(file_output, "regfile0: %h", cputb.cpu.Rf.registers[0]);
        $fdisplay(file_output, "regfile1: %h", cputb.cpu.Rf.registers[1]);
        $fdisplay(file_output, "regfile2: %h", cputb.cpu.Rf.registers[2]);
        $fdisplay(file_output, "regfile3: %h", cputb.cpu.Rf.registers[3]);
        $fdisplay(file_output, "regfile4: %h", cputb.cpu.Rf.registers[4]);
        $fdisplay(file_output, "regfile5: %h", cputb.cpu.Rf.registers[5]);
        $fdisplay(file_output, "regfile6: %h", cputb.cpu.Rf.registers[6]);
        $fdisplay(file_output, "regfile7: %h", cputb.cpu.Rf.registers[7]);
        $fdisplay(file_output, "regfile8: %h", cputb.cpu.Rf.registers[8]);
        $fdisplay(file_output, "regfile9: %h", cputb.cpu.Rf.registers[9]);
        $fdisplay(file_output, "regfile10: %h", cputb.cpu.Rf.registers[10]);
        $fdisplay(file_output, "regfile11: %h", cputb.cpu.Rf.registers[11]);
        $fdisplay(file_output, "regfile12: %h", cputb.cpu.Rf.registers[12]);
        $fdisplay(file_output, "regfile13: %h", cputb.cpu.Rf.registers[13]);
        $fdisplay(file_output, "regfile14: %h", cputb.cpu.Rf.registers[14]);
        $fdisplay(file_output, "regfile15: %h", cputb.cpu.Rf.registers[15]);
        $fdisplay(file_output, "regfile16: %h", cputb.cpu.Rf.registers[16]);
        $fdisplay(file_output, "regfile17: %h", cputb.cpu.Rf.registers[17]);
        $fdisplay(file_output, "regfile18: %h", cputb.cpu.Rf.registers[18]);
        $fdisplay(file_output, "regfile19: %h", cputb.cpu.Rf.registers[19]);
        $fdisplay(file_output, "regfile20: %h", cputb.cpu.Rf.registers[20]);
        $fdisplay(file_output, "regfile21: %h", cputb.cpu.Rf.registers[21]);
        $fdisplay(file_output, "regfile22: %h", cputb.cpu.Rf.registers[22]);
        $fdisplay(file_output, "regfile23: %h", cputb.cpu.Rf.registers[23]);
        $fdisplay(file_output, "regfile24: %h", cputb.cpu.Rf.registers[24]);
        $fdisplay(file_output, "regfile25: %h", cputb.cpu.Rf.registers[25]);
        $fdisplay(file_output, "regfile26: %h", cputb.cpu.Rf.registers[26]);
        $fdisplay(file_output, "regfile27: %h", cputb.cpu.Rf.registers[27]);
        $fdisplay(file_output, "regfile28: %h", cputb.cpu.Rf.registers[28]);
        $fdisplay(file_output, "regfile29: %h", cputb.cpu.Rf.registers[29]);
```

```
        $fdisplay(file_output, "regfile30: %h", cputb.cpu.Rf.registers[30]);
        $fdisplay(file_output, "regfile31: %h", cputb.cpu.Rf.registers[31]);
    end
end

Top cputb(clk, reset, o_seg, o_sel);

endmodule
```

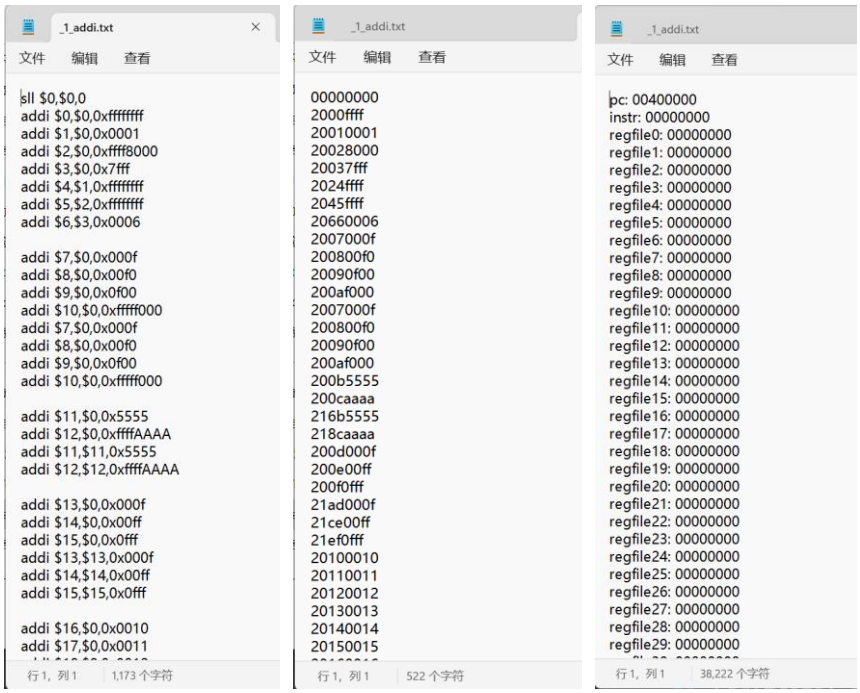
IMEM 测试模块代码:

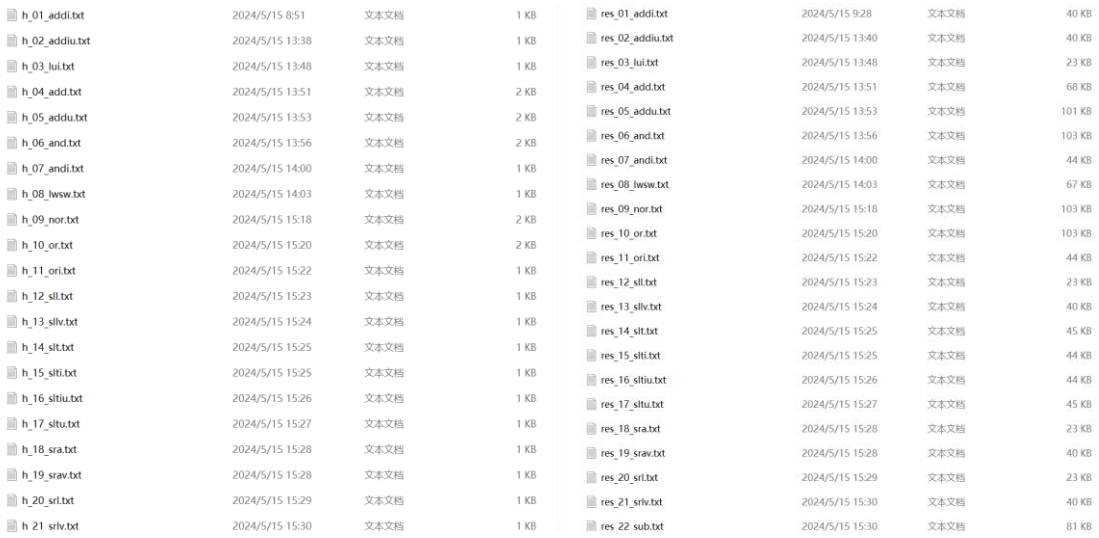
```
`timescale 1ns / 1ps
module IMEM_tb();
reg [31:0] addr;
wire [31:0] inst_code;
initial begin
    addr <= 0;
end
always #5 begin
    addr <= addr + 1;
end
IMEM imem(1'b1, addr, inst_code);
endmodule
```

五、实验结果

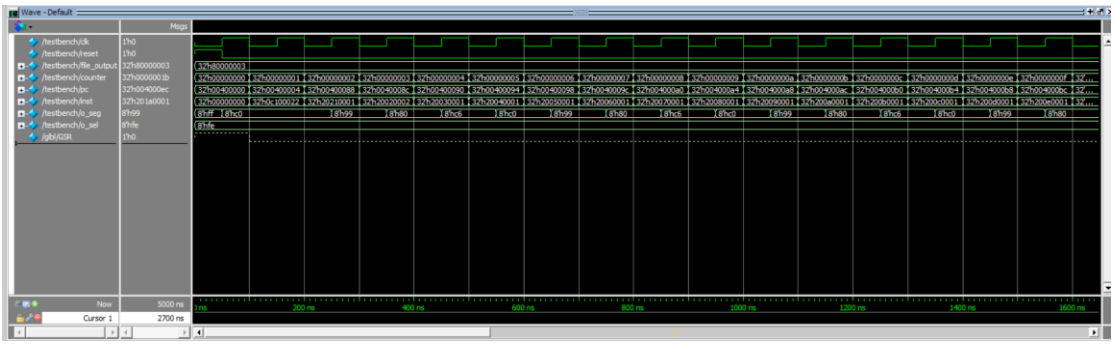
前仿真功能测试:

1. 首先将“31 条 CPUtest 指令实例核使用说明”中的汇编示例代码使用 Mars 进行编译得到相应的 16 进制机器码，随后运行程序得到运行过程中 PC、Instr、registers 值。

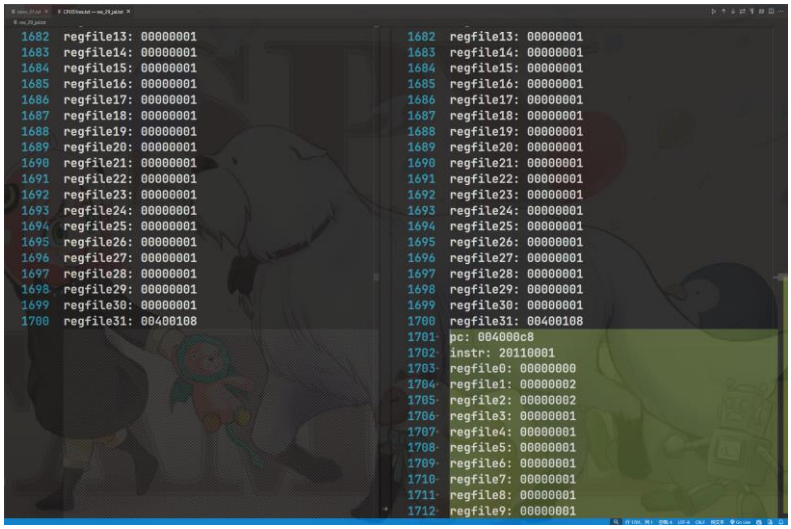




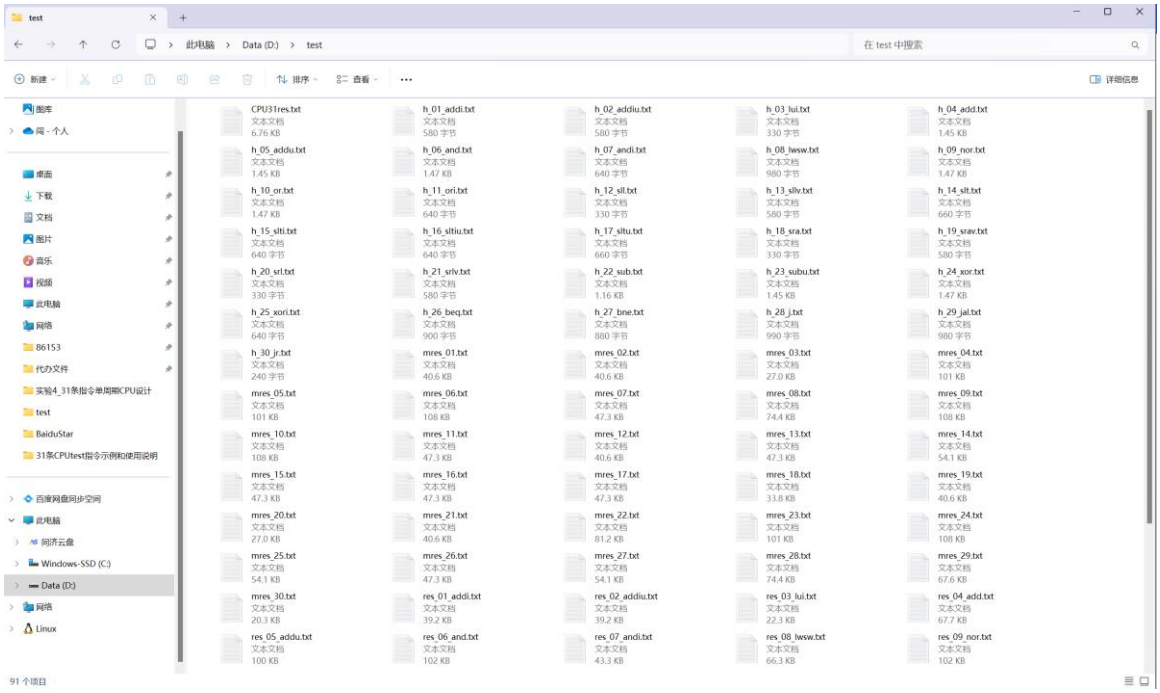
2. 使用编译得到的 16 进制机器码初始化 IMEM 模块，运行 testbech 模块进行前仿真测试，同时将运行过程中的 PC、Instr、registers 值按照 Mars 生成文件的格式输出。



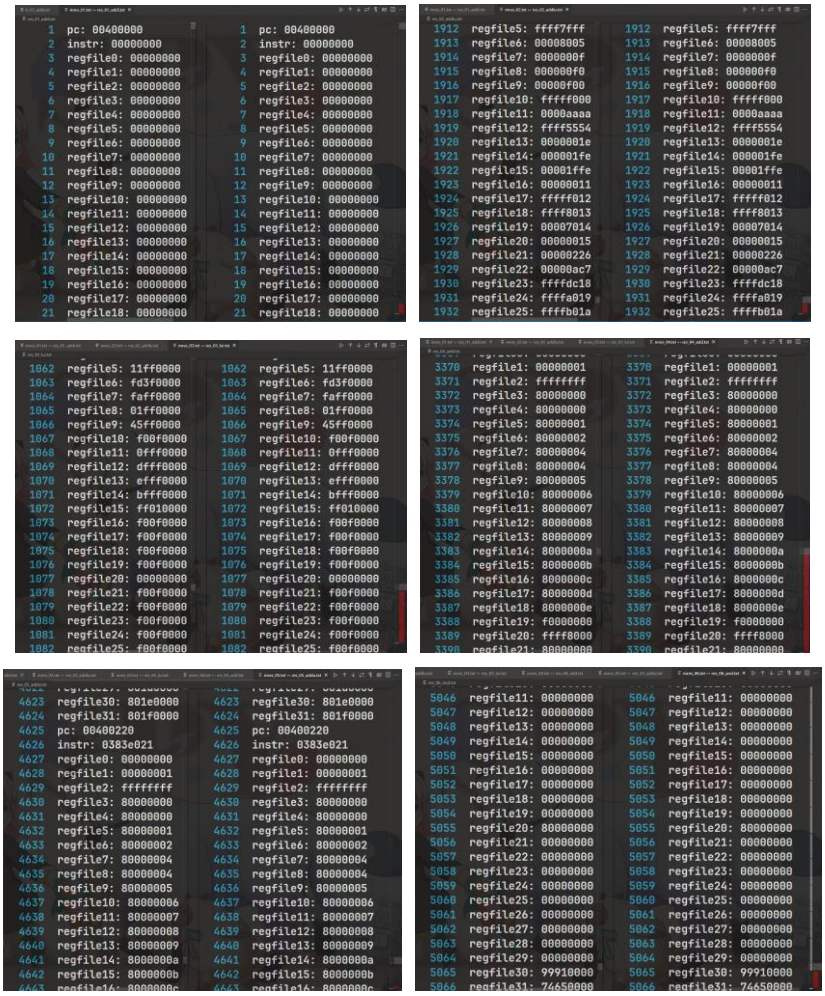
3. 将 testbech 生成的文件与 Mars 生成的文件进行文本对比，查看相应的 MIPS 指令是否运行正确。文本对比中后面黄色部分出现的原因是仿真时运行的时间较短，未全部运行完所有的指令。

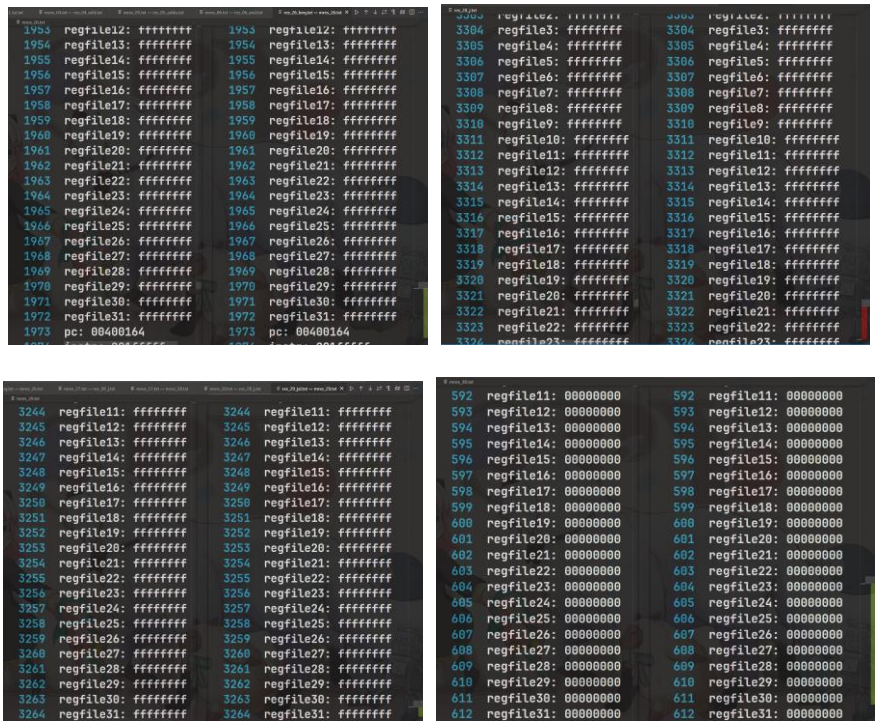


为了方便测试将生成的所有文件存放在一个“测试”文件中。



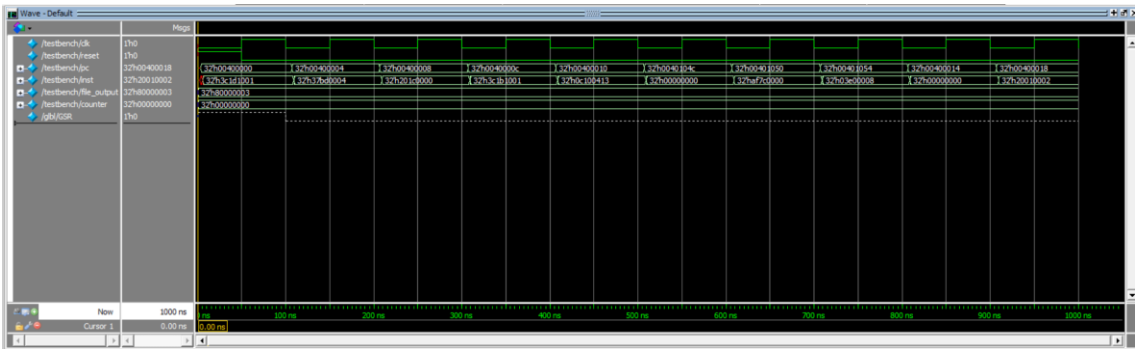
4. 部分测试对比结果展示：





后仿真时序测试：

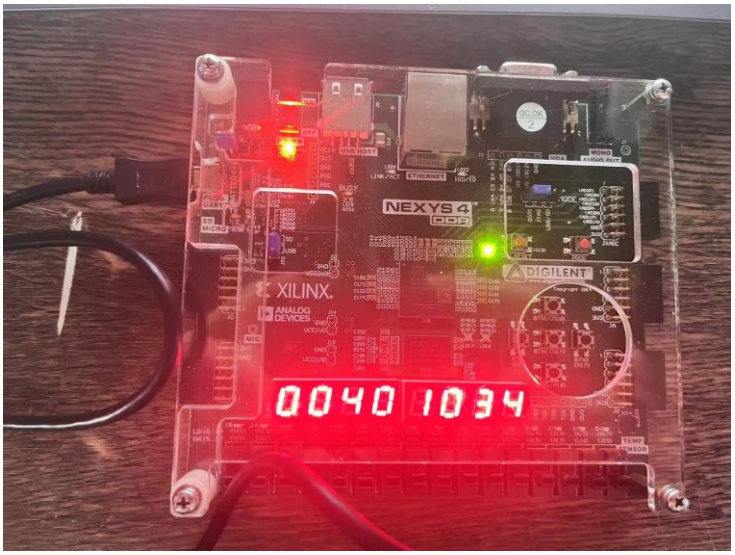
与前仿真功能测试相比，后仿真时序测试考虑了电路路径延迟和门延迟的影响，所以存在一定的偏移。

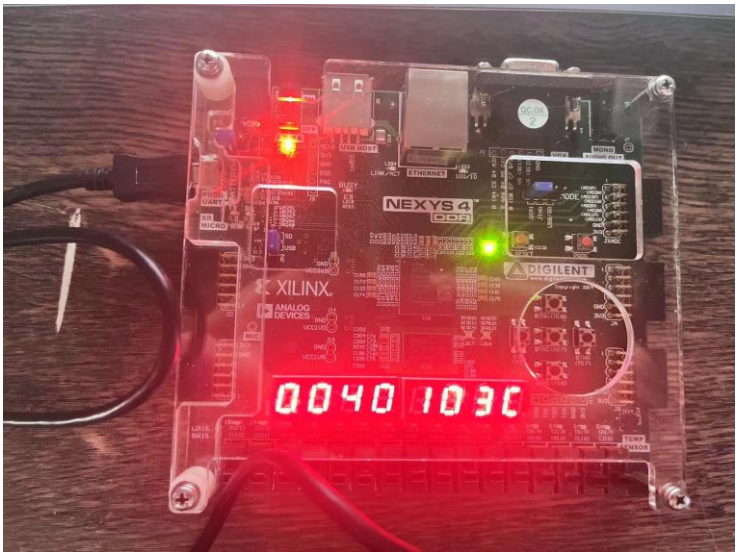


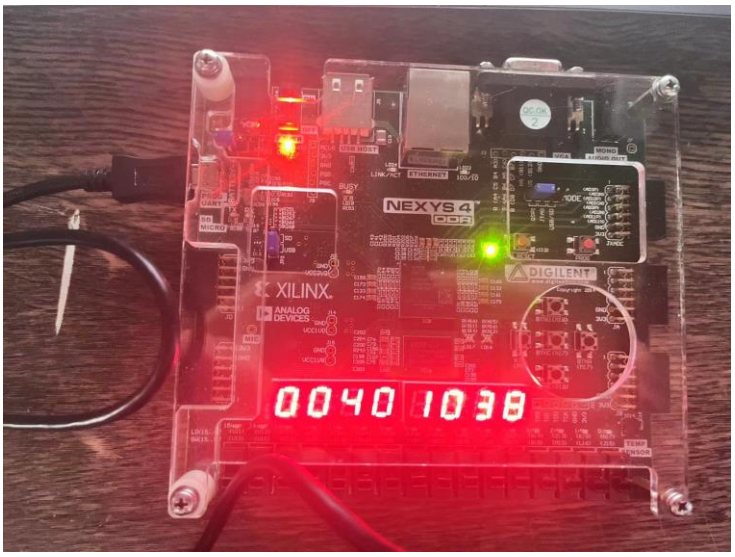
下板测试：

在下板测试之前需要对源代码进行一定的修改，在 sccomp_dataflow 模块中，需要删除[31:0] inst 和[31:0] pc 两个接口，接入 o_seg 和 o_sel 接口，同时需要实例化一个 seg7x16 模块用于数码管展示，在本次实验中我将 pc 的值传入其中，为了更好地观察 pc 值得变化，还需要对工作脉冲进行分频，使得其运行速率不会太快以导致人眼观察不出来。

使用“CPU31 验收说明”文件中的“test2019_for_cpu31.coe”文件初始化 ROM IP 核，然后进行综合并生成 bit 流进行下板测试，在下板测试之前先将时钟进行分频，使得其可以 1S 运行一条指令，同时将 PC 值输出到七段数码管上进行检测。


NEXYS 4
DDR
XILINX
DIGILENT
0040 1034


NEXYS 4
DDR
XILINX
DIGILENT
0040 103C


NEXYS 4
DDR
XILINX
DIGILENT
0040 1038