同济大学计算机系
计算机组成原理
54条指令单周期CPU设计
实验报告

学　　号 2252552

姓　　名 胡译文

专　　业 信息安全

授课老师 张冬冬

目录

一、实验内容

1. 实验介绍

在本次实验中，我们将使用Verilog HDL语言实现 54 条 MIPS 指令的单周期 CPU 的设计和仿真。

2. 实验目标

- 深入了解 CPU 的原理。
- 画出实现 54 条指令的 CPU 的通路图。
- 学习使用 Verilog HDL 语言设计实现 54 条指令的 CPU。

3. 实验原理

实现54条MIPS指令CPU，在31条指令CPU基础上，需要添加的指令如下图， 其中中断相关指令需要用到CP0协处理器。

1） 已经实现的 31 条MIPS 指令，见图

MIPS 指令集（共 31 条）									
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 + (sign-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 <(sign-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

2） 需要添加的23条指令

Mips 指令集（共 23 条）								
指令	指令说明	指令格式	OP31-26	RS25-21	RT20-16	RD15-11	SA10-6	FUNCT5-0
div	除	DIV rs, rt	000000			00000	00000	011010
divu	除（无符号）	DIVU rs, rt	000000			00000		011011
mult	乘	MULT rs, rt	000000			00000		011000
multu	乘（无符号）	MULTU rs, rt	000000			00000		011001
bgez	大于等于 0 时分支	BGEZ rs, offset	000001		00001			
jalr	跳转至寄存器所指地址，返回地址保存	JALR rs	000000		00000			001001
lbu	取字节(无符号)	LBU rt, offset(base)	100100					

lhu	取半字(无符号)	LHU rt, offset(base)	100101					
lb	取字节	LBU rt, offset(base)	100000					
lh	取半字	LHU rt, offset(base)	100001					
sb	存字节	SB rt, offset(base)	101000					
sh	存半字	SH rt, offset(base)	101001					
break	断点	BREAK	000000					001101
syscall	系统调用	SYSCALL	000000					001100
eret	异常返回	ERET	010000	**10000**	00000	00000	00000	011000
mfhi	读Hi寄存器	MFHI rd	000000	**00000**	00000		00000	010000
mflo	读Lo寄存器	MFLO rd	000000	**00000**	00000		00000	010010
mthi	写Hi寄存器	MTHI rd	000000		00000	00000	00000	010001
mtlo	写Lo寄存器	MTLO rd	000000		00000	00000	00000	010011
mfc0	读CP0寄存器	MFC0 rt, rd	010000	**00000**			00000	00000
mtc0	写CP0寄存器	MTC0 rt, rd	010000	**00100**			00000	00000
clz	前导零计数	CLZ rd, rs	011100				00000	100000
teq	相等异常	TEQ rs, rt	000000					110100

3） 单周期数据通路设计的一般性方法

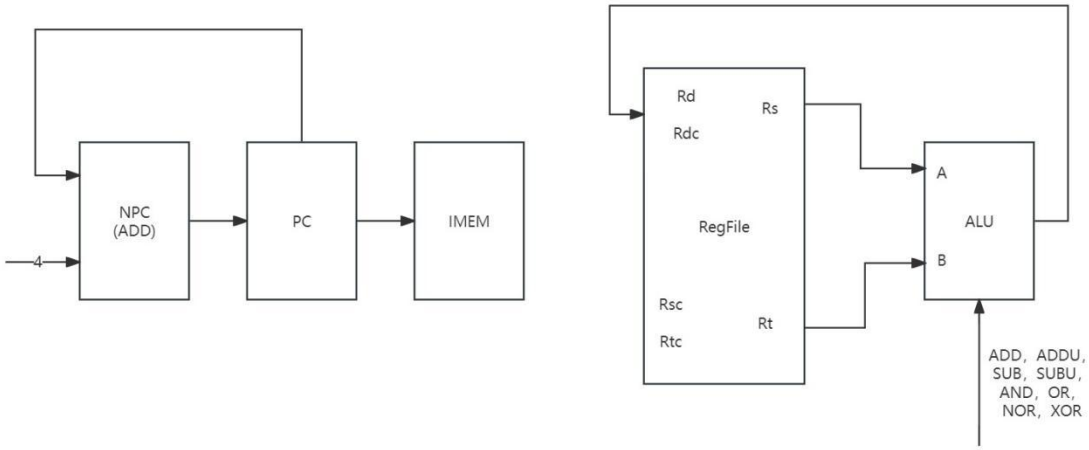
- 阅读每条指令，对每条指令所需执行的功能与过程都有充分的了解
- 确定每条指令在执行过程中所用到的部件
- 使用表格列出指令所用部件，并在表格中填入每个部件的数据输入来源
- 根据表格所涉及部件和部件的数据输入来源，画出整个数据通路

二、数据通路图

1. 单指令数据通路图

*为了节省篇幅，我将类似的指令数据通路图都合并在一起

1.1 8条计算指令



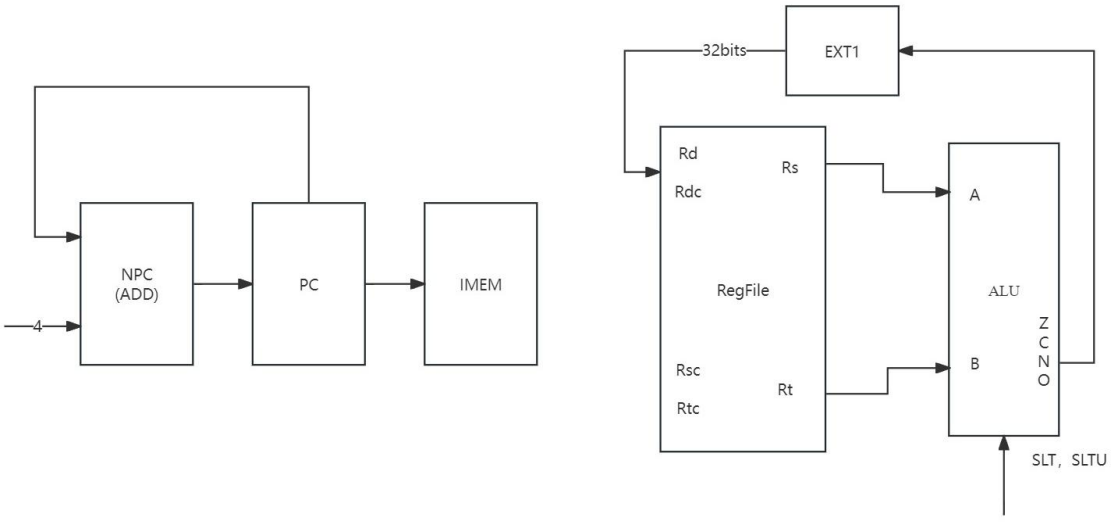
格式： ADD(ADDU/SUB/SUBU/AND/OR/NOR/XOR)　　rd, rs, rt

操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> PC

Rs -> A , Rt -> B
//ALU内部操作，8条指令只有这一步不同
//(A + B -> RES) (A - B -> RES)
//(A & B -> RES) (A | B -> RES)
//(A ⊕ B -> RES) (A ⊙ B -> RES)
RES -> Rd
```

所用部件：PC 寄存器(PC、NPC)，指令存储器(IMEM)，寄存器文件(RegFile)，ALU

1.2 SLT\SLTU



格式：SLT(SLTU) rd, rs, rt

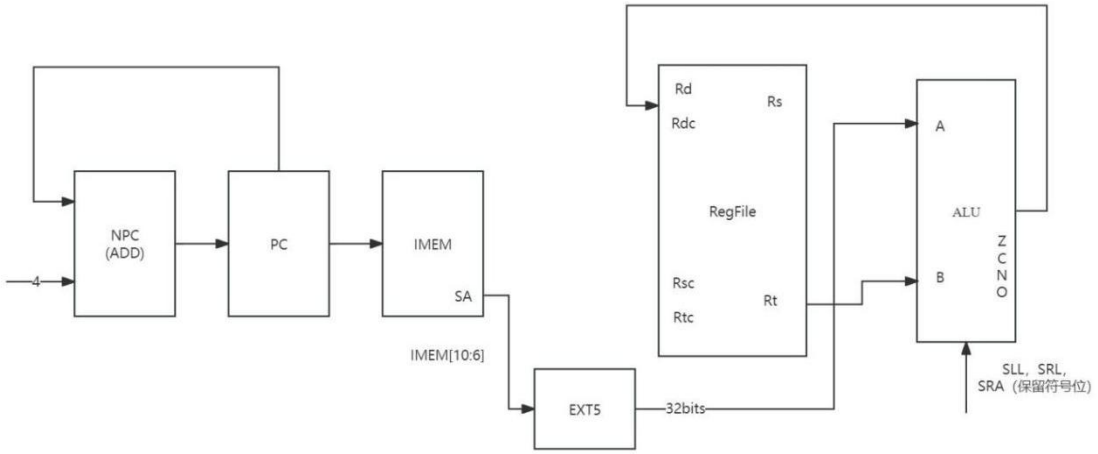
操作：

```
PC -> IMEM
```

```
PC + 4 -> NPC
NPC -> PC

Rs -> A , Rt -> B
// ALU内部操作,相减判断，负数则为Rs中数小
//(A - B -> RES)
SF -> EXT1
EXT1_OUT -> Rd
```

所用部件：PC寄存器(PC、NPC),指令存储器(IMEM),寄存器文件(RegFile),ALU, EXT1

1.3 SLL\SRL\SRA



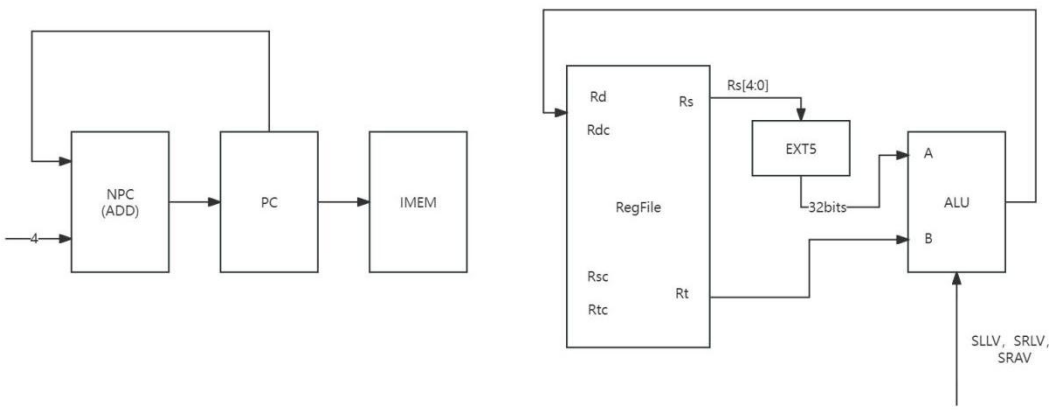
格式：SLL(SRL\SRA) rd, rt, sa

操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC > PC

IMEM[10:6] -> EXT5
EXT5_OUT -> A
Rt -> B
// ALU内部操作
//(B << A -> RES)
//(B >> A -> RES)
RES -> Rd
```

所用部件：PC 寄存器(PC、NPC)，指令存储器(IMEM)，寄存器文件(RegFile)，ALU,EXT5(零扩展)

1.4 SLLV\SRLV\SRAV



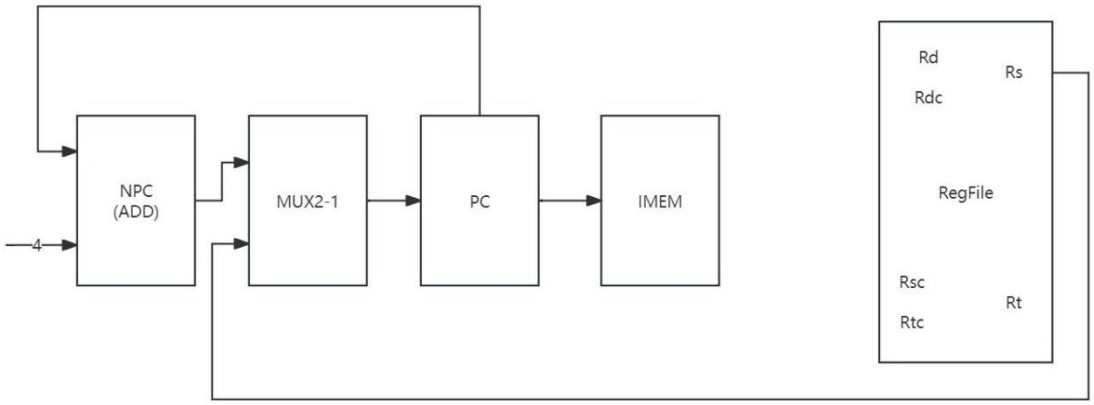
格式：SLLV(SRLV\SRAV) rd, rt, rs

操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> PC

Rs[4:0] -> EXT5
EXT5_OUT -> A
Rt -> B
//(B << A -> RES)
//(B >> A -> RES)
RES -> Rd
```

所用部件：PC 寄存器(PC、NPC)，指令存储器(IMEM)，寄存器文件(RegFile)，ALU,EXT5(零扩展)

1.5 JR



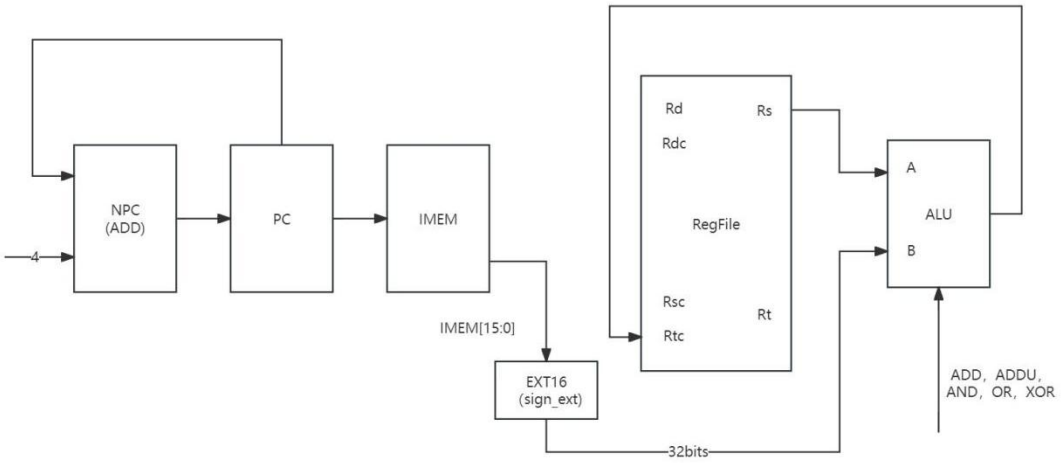
格式：JR rs

操作：

```
PC -> IMEM
PC + 4 -> NPC //其他指令

Rs -> MUX
MUX_OUT -> PC
NPC -> MUX    //其他指令
```

所用部件：PC 寄存器(PC、NPC)，指令存储器(IMEM)，寄存器文件(RegFile)，数据选择器(MUX)

1.6 5条立即数计算指令

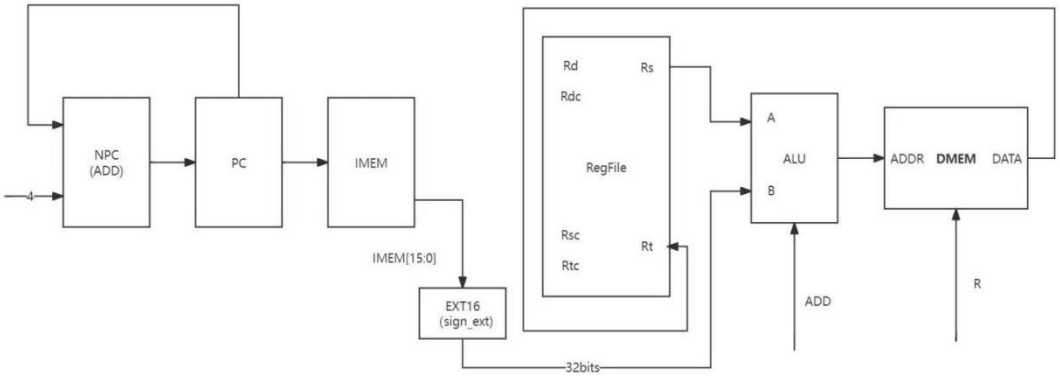


格式：ADDI(ADDIU/ANDI/ORI/XORI) rt, rs, immediate
操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC > PC

IMEM[15:0] -> EXT16
EXT16_OUT -> B
Rs -> A
//ALU内部操作，5条指令只有这一步不同
//(A + B -> RES) (A ⊕ B -> RES)
//(A & B -> RES) (A | B -> RES)
RES -> Rd
```

所用部件：PC寄存器(PC、NPC),指令存储器(IMEM),寄存器文件(RegFile),ALU,EXT16(符号扩展)

1.7 LW

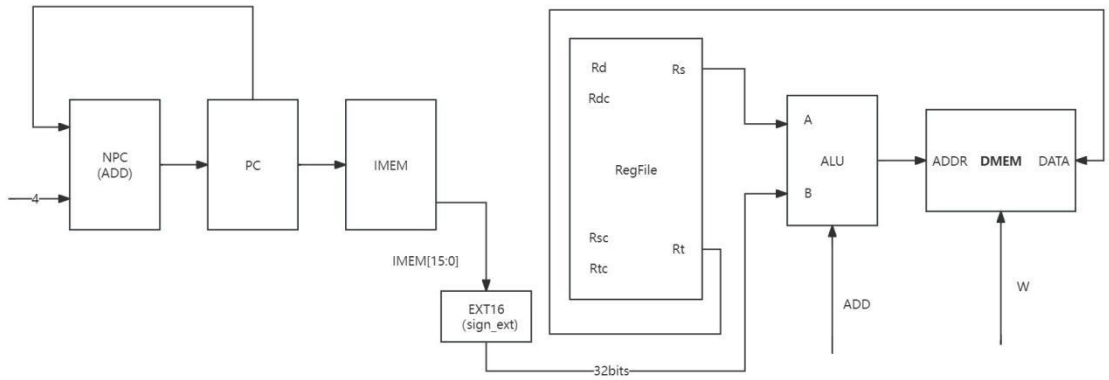


格式：LW rt, offset(base)
操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> PC
IMEM[15:0] -> EXT16
EXT16_OUT -> B
Rs -> A
//(A + B -> RES)
RES -> DMEM_ADDR
DMEM_OUT -> Rd
```

所用部件：PC 寄存器(PC、NPC)，指令存储器(IMEM)，寄存器文件(RegFile)，
ALU，EXT16(符号扩展)，数据存储器(DMEM)

1.8 SW



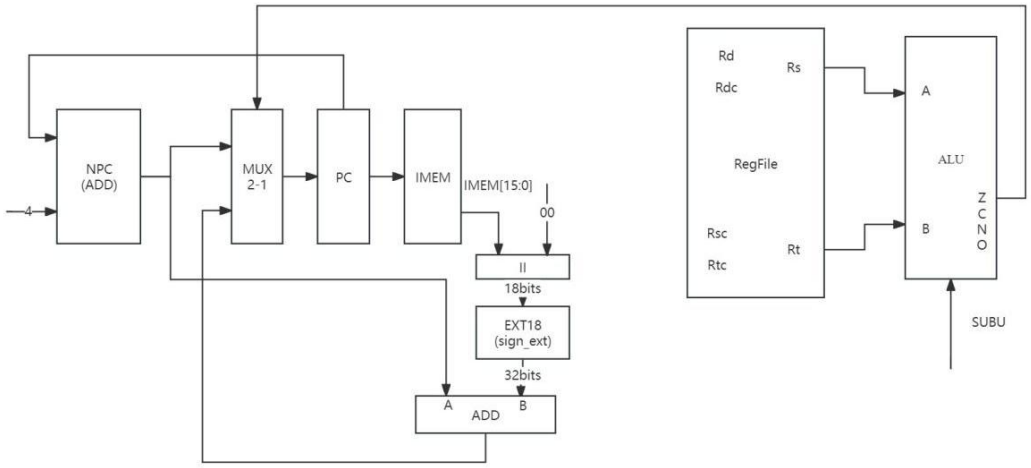
格式：SW rt, offset(base)

操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> PC

IMEM[15:0] -> EXT16
EXT16_OUT -> B
Rs -> A
//(A + B -> RES)
Rt -> DMEM
RES -> DMEM_ADDR
```

所用部件：PC 寄存器(PC、NPC)，指令存储器(IMEM)，寄存器文件(RegFile)，ALU,EXT16(符号扩展)，数据存储器(DMEM)

1.9 BEQ/BNE



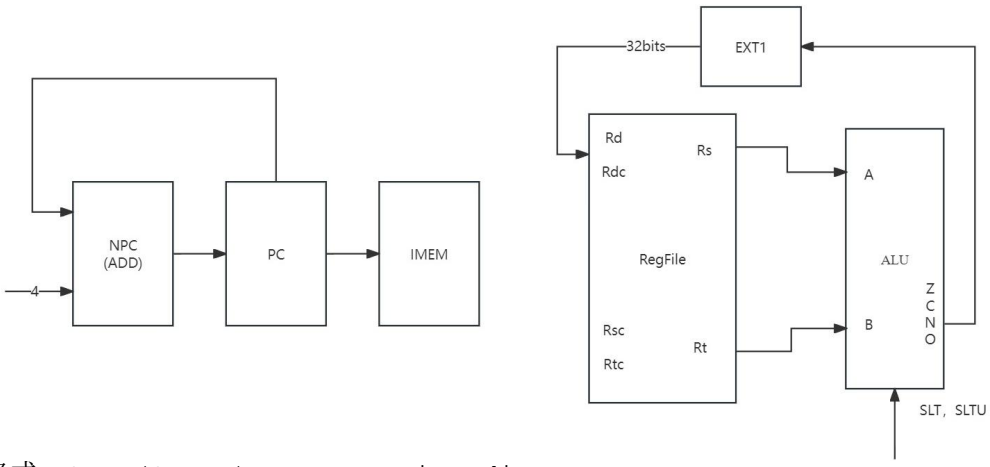
格式：BEQ(BNE) rs, rt, offset

操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> MUX

IMEM[15:0] | 00 -> EXT18
EXT18_OUT -> ADD_A
NPC -> ADD_B
//(ADD_A + ADD_B -> ADD_OUT)
ADD_OUT -> MUX

Rs -> A
Rt -> B
//(A + B -> RES)
Z -> MUX
MUX -> PC
```

所用部件：PC 寄存器(PC、NPC)，指令存储器(IMEM)，寄存器文件(RegFile)，ALU，EXT18(符号扩展)，数据选择器(MUX)，连接器||，加法器

1.10 SLTI/SLTIU



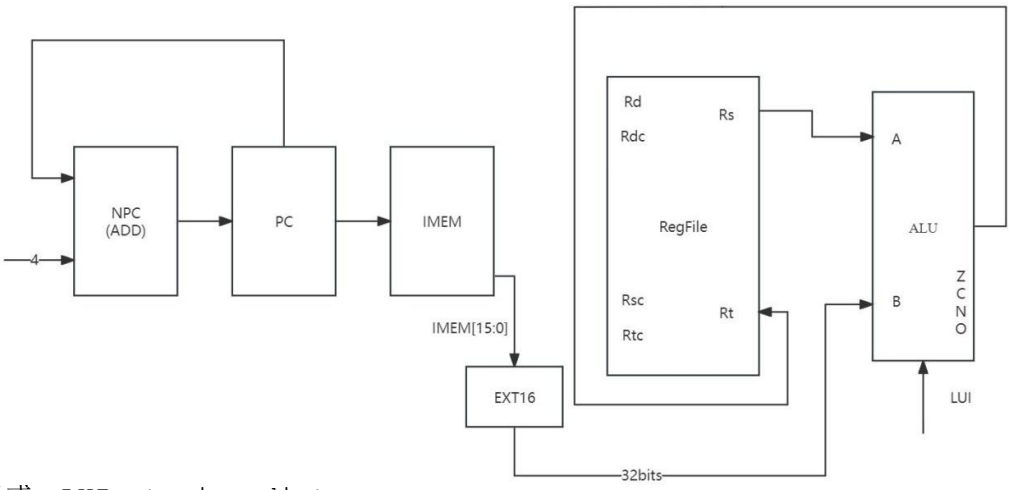
格式：SLTI(SLTIU) rt, rs, immediate

操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> PC
IMEM[15:0] -> EXT16
EXT16_OUT -> B
Rs -> A
//(A - B -> RES)
N -> EXT1
EXT1_OUT -> Rd
```

所用部件：PC 寄存器(PC、NPC)，指令存储器(IMEM)，寄存器文件(RegFile)，ALU，EXT16(符号扩展)

1.11 LUI



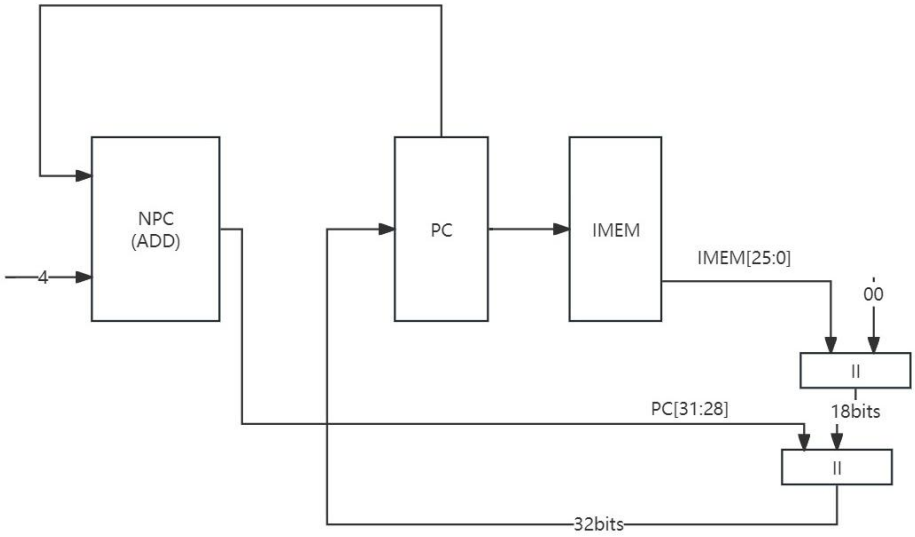
格式：LUI rt, immediate

操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> PC

IMEM[15:0] -> EXT16
EXT16_OUT -> B
RES -> Rd
```

所用部件：PC 寄存器(PC、NPC)，指令存储器(IMEM)，寄存器文件(RegFile)，ALU，EXT16(零扩展)

1.12 J



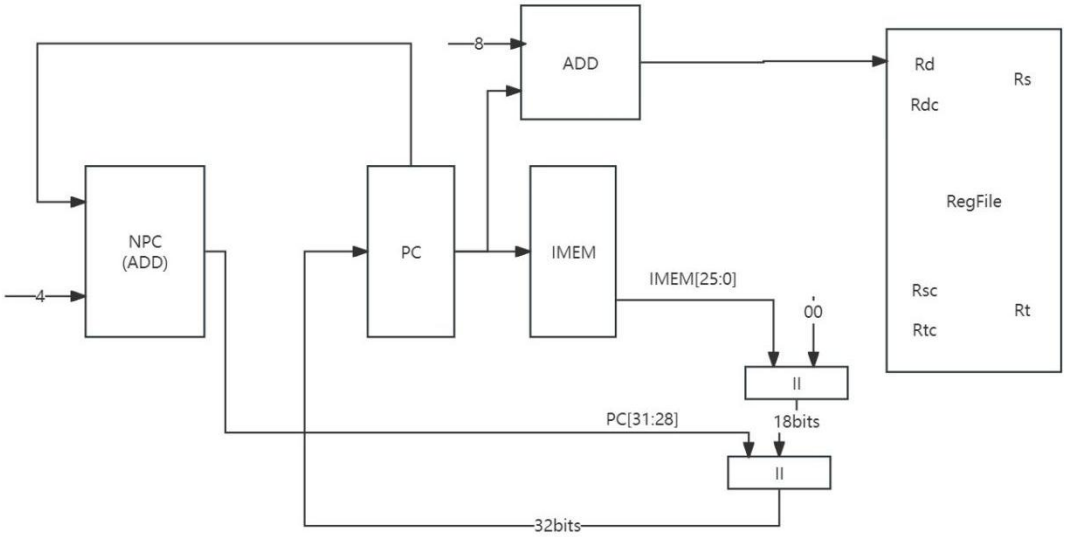
格式：J target

操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> MUX

PC[31:28] -> || _A
IMEM[25,0] || 00 -> || _B
||_OUT -> MUX
MUX -> PC
```

所用部件：PC 寄存器(PC、NPC)，指令存储器(IMEM)，连接器||

1.13 JAL



格式：JAL target

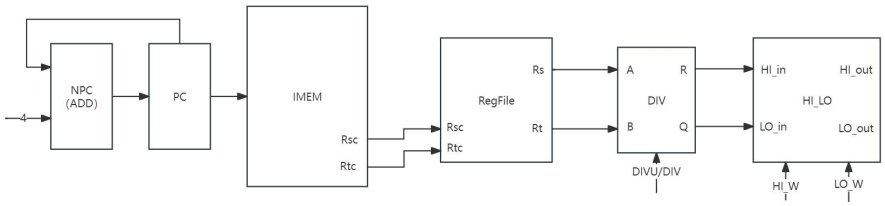
操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> MUX

8 -> ADD_A
```

```
PC -> ADD_B
//(ADD_A + ADD_B -> ADD_OUT)
ADD_OUT -> Rd

PC[31:28] -> || _A
IMEM[25,0] || 00 -> || _B
||_OUT -> MUX
MUX -> PC
```

所用部件：PC 寄存器(PC、NPC)，指令存储器(IMEM)，寄存器文件(RegFile)，连接器||，加法器(ADD)

1.14 DIV，DIVU



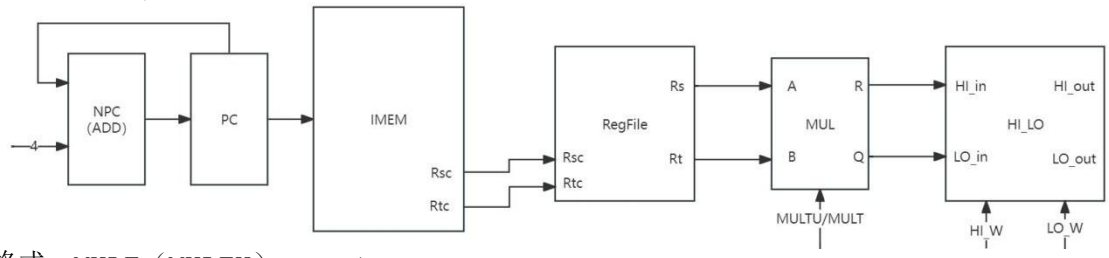
格式：DIV（DIVU） rs,rt

操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> PC


Rs -> A 被除数
Rt -> B 除数
A/B -> Q
A%B -> R
Q -> HI_in
R -> LO_in
```

所用部件：PC寄存器(PC、NPC),指令存储器(IMEM),寄存器文件(RegFile),除法器(DIV),加法器(ADD),HI_LO寄存器

1.15 MULT，MULTU



格式：MULT（MULTU） rs,rt

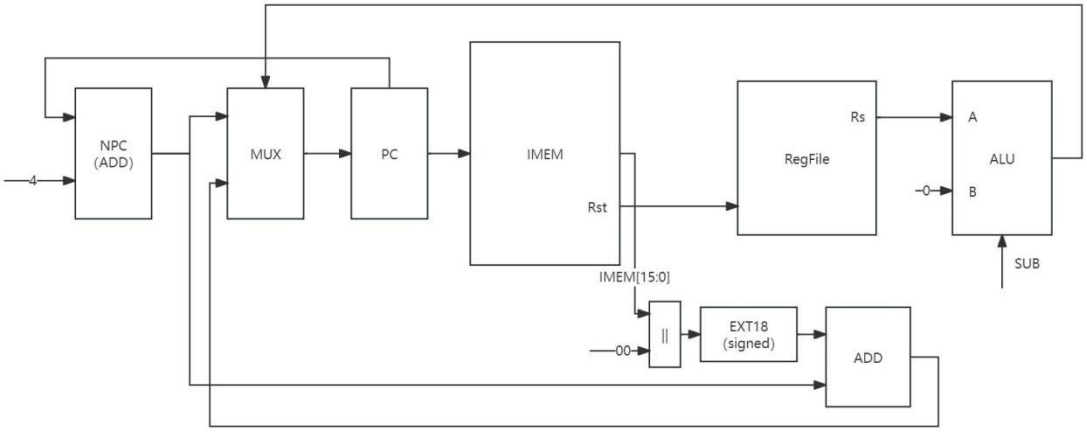
操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> PC
```

```
Rs -> A 乘数
Rt -> B 乘数
A*B -> rst
rst[63:32] -> HI_in
rst[31:0] -> LO_in
```

所用部件：PC寄存器(PC、NPC),指令存储器(IMEM),寄存器文件(RegFile),乘法器(MUL),加法器(ADD),HI_LO寄存器

1.16 BGEZ



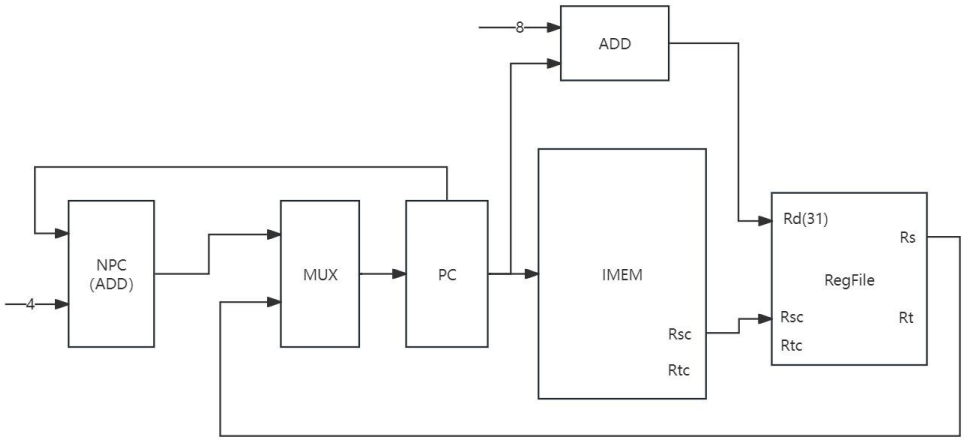
格式：BGEZ rs, rt, offset

操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> MUX

IMEM[15:0] || 00 -> EXT18
EXT18_OUT -> ADD_A
NPC -> ADD_B
//(ADD_A + ADD_B -> ADD_OUT)
ADD_OUT -> MUX

Rs -> A
0 -> B
//(A - B -> RES)
S -> MUX

if S < 0
MUX(NPC) -> PC
else
MUX(ADD_out) -> PC
```

所用部件：PC 寄存器(PC、NPC)，指令存储器(IMEM)，寄存器文件(RegFile)，ALU，EXT18(符号扩展)，数据选择器(MUX)，连接器||,加法器(ADD)

1.17 JALR



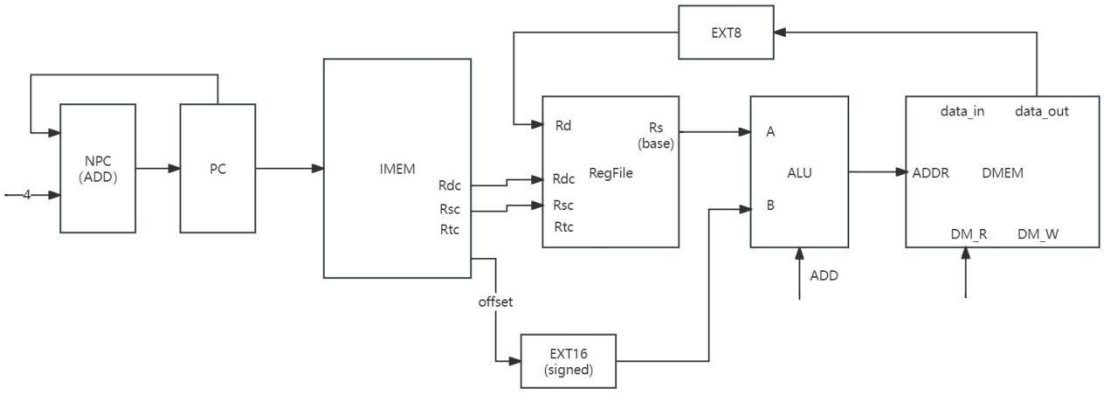
格式：JALR rs

操作：

```
PC -> IMEM
PC -> ADD_A
ADD_A + ADD_B -> ADD_out
ADD_out -> Rd(31)

PC + 4 -> NPC
NPC -> MUX
Rs -> MUX
MUX(Rs) -> PC
```

所用部件：PC 寄存器(PC、NPC)，指令存储器(IMEM)，数据选择器(MUX)，寄存器文件(RegFile)，加法器(ADD)

1.18 LBU



格式：LBU rt, offset(base)

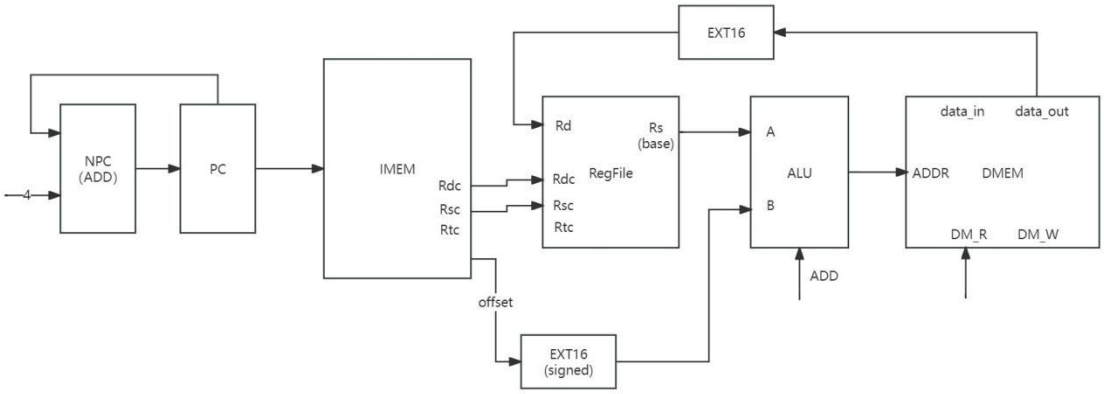
操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> PC

IMEM[15:0] -> EXT16
Rs -> A
EXT16_OUT -> B
//(A + B -> RES)

RES -> DMEM_ADDR
```

```
DMEM_OUT -> EXT8_in
EXT8_out -> Rd
```

所用部件：PC 寄存器(PC、NPC),指令存储器(IMEM),寄存器文件(RegFile), ALU, EXT8(无符号扩展),EXT16(符号扩展),数据存储器(DMEM)

1.19 LHU



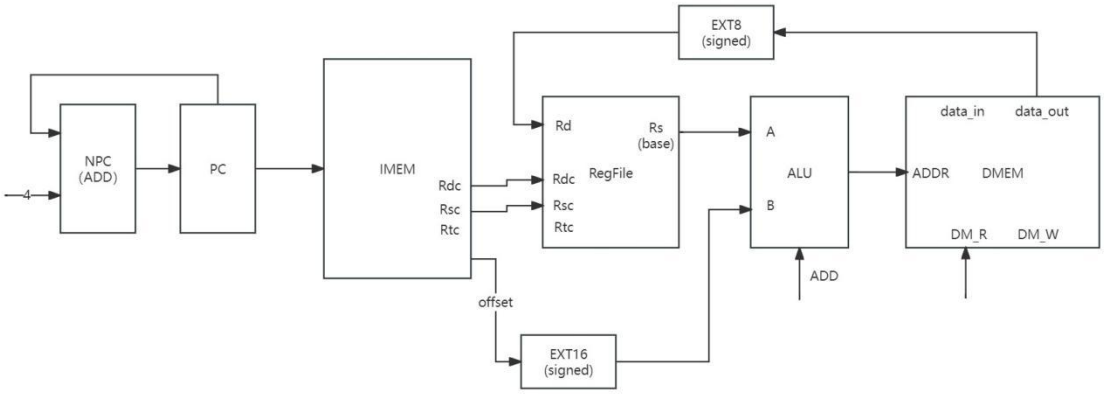
格式：LHU rt, offset(base)

操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> PC

IMEM[15:0] -> EXT16
Rs -> A
EXT16_OUT -> B
//(A + B -> RES)

RES -> DMEM_ADDR
DMEM_OUT -> EXT16_in
EXT16_out -> Rd
```

所用部件：PC 寄存器(PC、NPC),指令存储器(IMEM),寄存器文件(RegFile), ALU,EXT16(无符号扩展),EXT16(符号扩展),数据存储器(DMEM)

1.20 LB



格式：LB rt, offset(base)

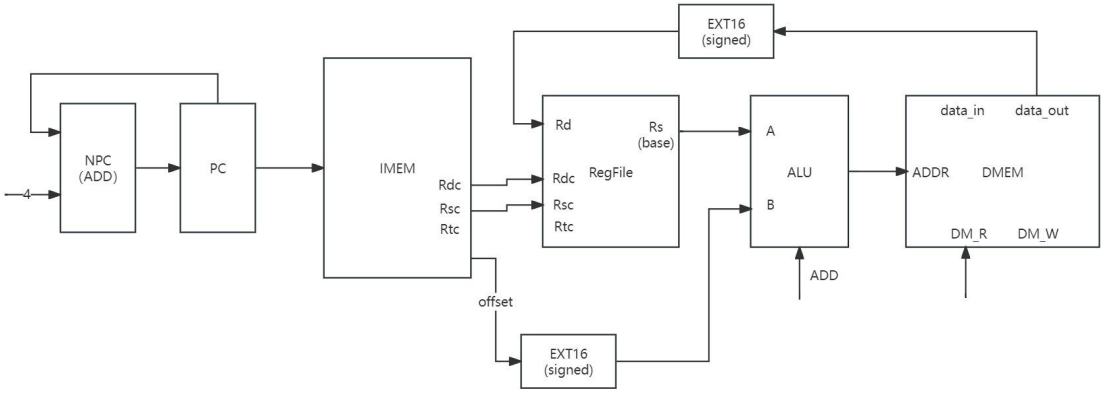
操作：

```
PC -> IMEM
```

```
PC + 4 -> NPC
NPC -> PC

IMEM[15:0] -> EXT16
Rs -> A
EXT16_OUT -> B
//(A + B -> RES)

RES -> DMEM_ADDR
DMEM_OUT -> EXT8_in
EXT8_out -> Rd
```

所用部件：PC 寄存器(PC、NPC),指令存储器(IMEM),寄存器文件(RegFile), ALU, EXT8(符号扩展),EXT16(符号扩展),数据存储器(DMEM)

1.21 LH



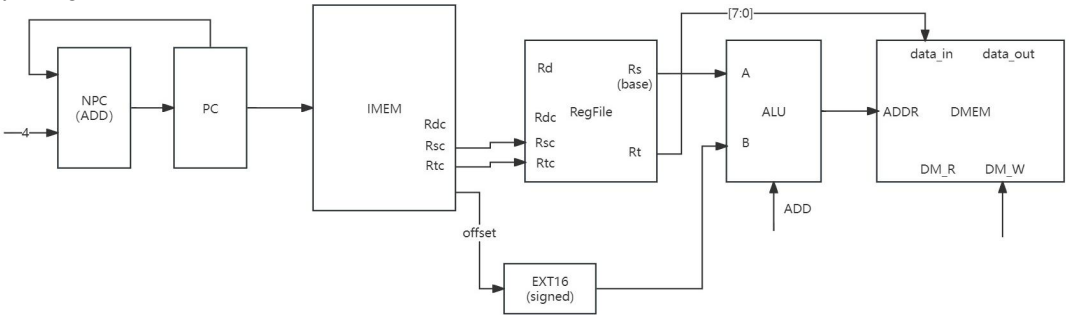
格式：LH rt, offset(base)

操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> PC

IMEM[15:0] -> EXT16
Rs -> A
EXT16_OUT -> B
//(A + B -> RES)

RES -> DMEM_ADDR
DMEM_OUT -> EXT16_in
EXT16_out -> Rd
```

所用部件：PC 寄存器(PC、NPC),指令存储器(IMEM),寄存器文件(RegFile), ALU, EXT16(符号扩展),数据存储器(DMEM)

1.22 SB



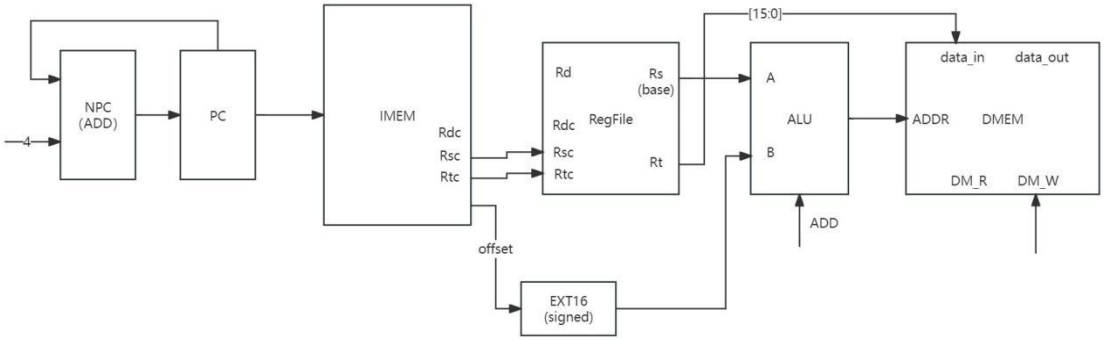
格式：SB rt, offset(base)

操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> PC

IMEM[15:0] -> EXT16
EXT16_OUT -> B
Rs -> A
//(A + B -> RES)
Rt[7:0] -> Data_in
RES -> DMEM_ADDR
```

所用部件：PC 寄存器(PC、NPC)，指令存储器(IMEM)，寄存器文件(RegFile)，ALU,EXT16(符号扩展)，数据存储器(DMEM)

1.23 SH



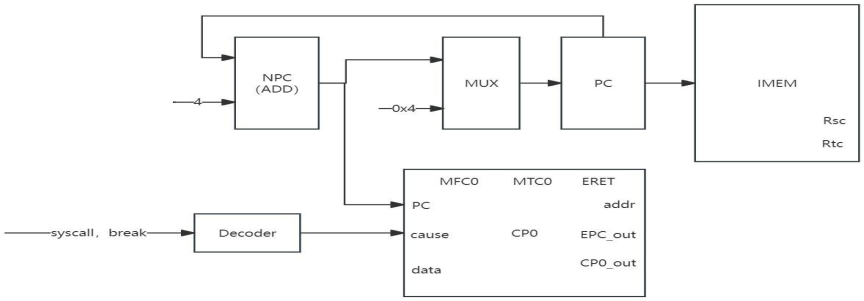
格式：SH rt, offset(base)

操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> PC

IMEM[15:0] -> EXT16
EXT16_OUT -> B
Rs -> A
//(A + B -> RES)
Rt[15:0] -> Data_in
RES -> DMEM_ADDR
```

所用部件：PC 寄存器(PC、NPC)，指令存储器(IMEM)，寄存器文件(RegFile)，ALU,EXT16(符号扩展)，数据存储器(DMEM)

1.25 BREAK,SYSCALL



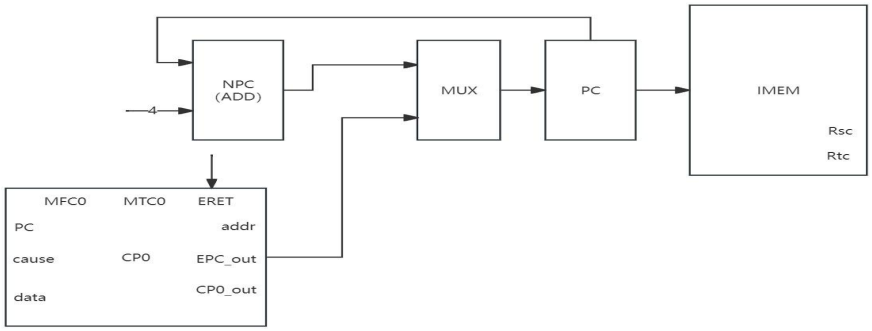
格式：BREAK(SYSCALL)

操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> MUX

NPC -> CP0
DECODER_out -> CP0_CAUSE
(STATUS << 5 -> STATUS)
0x4 -> MUX
MUX(0x4) -> PC
```

所用部件：PC 寄存器(PC、NPC)，指令存储器(IMEM)，译码器(DECODER)，多路选择器(MUX)，CP0

1.24 ERET



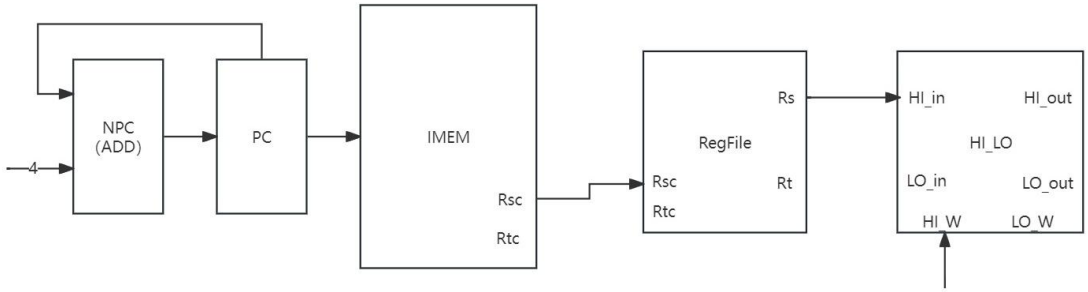
格式：ERET

操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> MUX

(STAUTS >> 5 -> STATUS)
EPC_OUT -> MUX
MUX(EPC_OUT) -> PC
```

所用部件：PC 寄存器(PC、NPC)，指令存储器(IMEM)，多路选择器(MUX)， CP0

1.26 MTHI



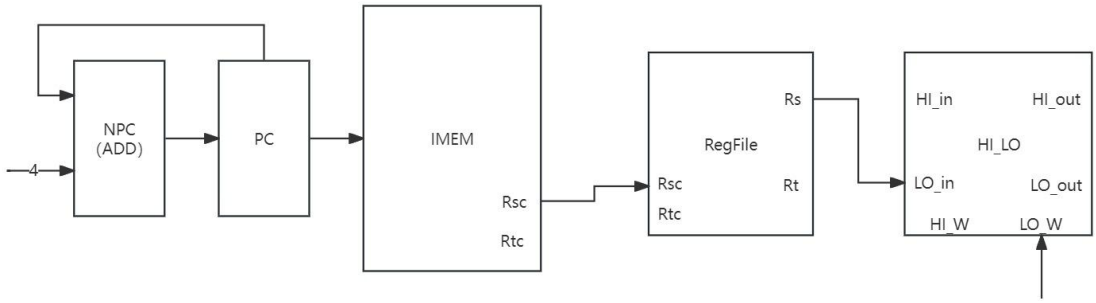
格式：MTHI rs

操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> MUX

Rs -> HI_in
```

所用部件：PC 寄存器(PC、NPC),指令存储器(IMEM),寄存器文件(RegFile),HI_LO寄存器

1.27 MTLO



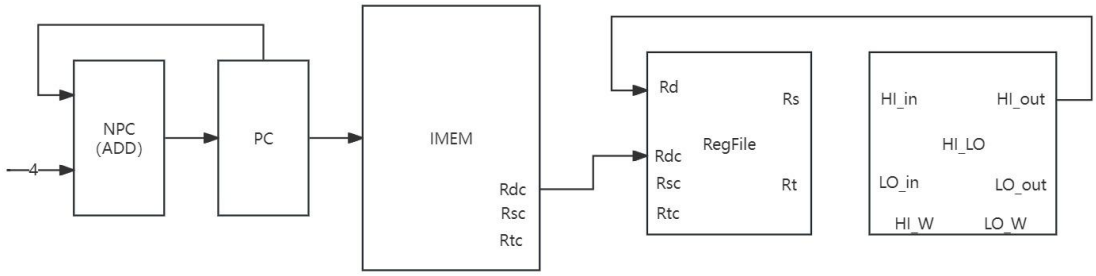
格式：MTLO rs

操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> MUX

Rs -> LO_in
```

所用部件：PC 寄存器(PC、NPC),指令存储器(IMEM),寄存器文件(RegFile),HI_LO寄存器

1.28 MFHI



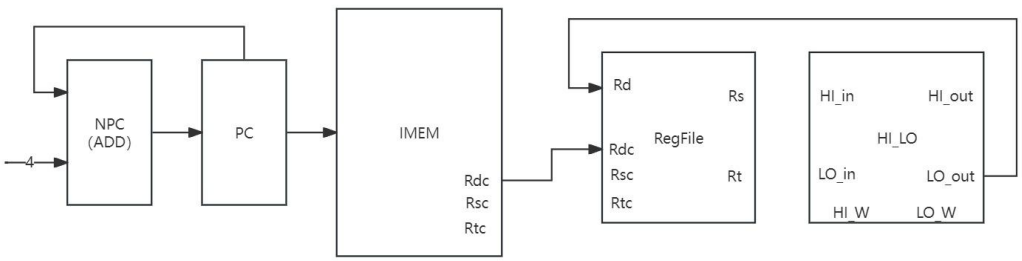
格式：MFHI rd

操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> MUX

HI_out  -> Rd
```

所用部件：PC 寄存器(PC、NPC),指令存储器(IMEM),寄存器文件(RegFile),HI_LO寄存器

1.29 MFLO



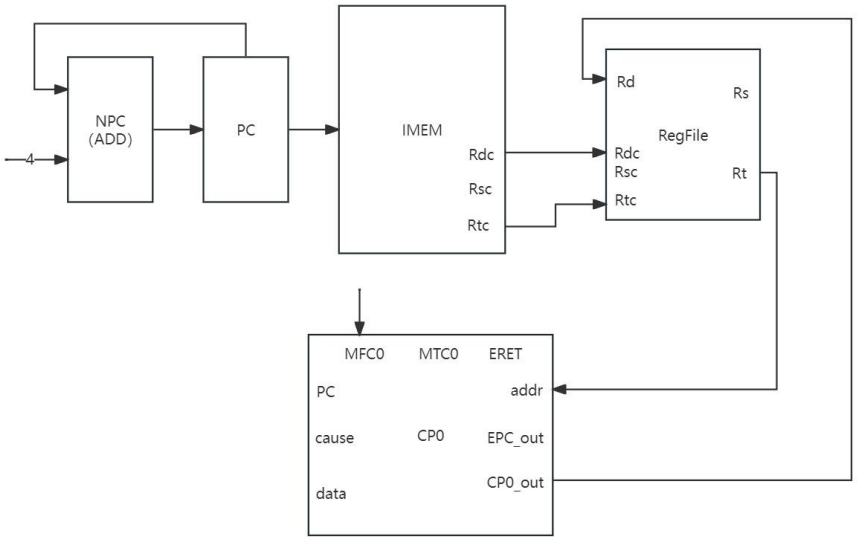
格式：MFLO rd

操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> MUX
LO_out  -> Rd
```

所用部件：PC 寄存器(PC、NPC),指令存储器(IMEM),寄存器文件(RegFile),HI_LO寄存器

1.30 MFC0



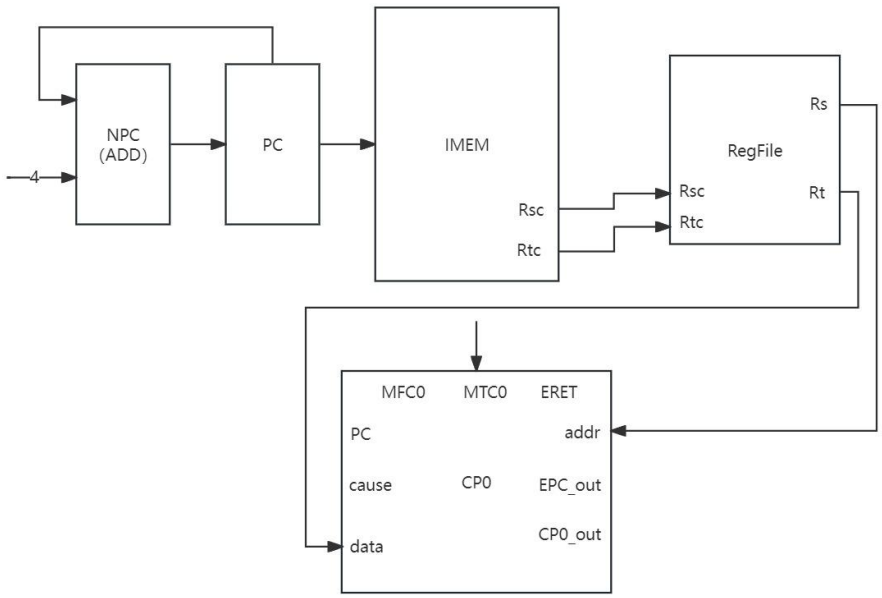
格式：MFCO rt,rd

操作：注：做了调整，互换了Rt和Rd，改为Rt输出地址，Rd写入

```
PC -> IMEM
PC + 4 -> NPC
NPC -> MUX
Rt -> ADDR
CP0_OUT -> Rd
```

所用部件：PC 寄存器(PC、NPC),指令存储器(IMEM),寄存器文件(RegFile),CP0

1.31 MTC0



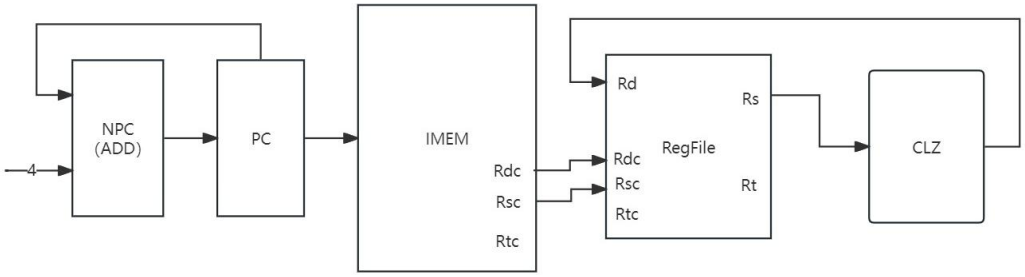
格式：MTCO rt,rs

操作：注：做了调整，互换了Rt和Rs，改为Rt输出地址，Rs写入的数据

```
PC -> IMEM
PC + 4 -> NPC
NPC -> MUX

Rt -> ADDR
Rs -> CP0_DATA
```

所用部件：PC 寄存器(PC、NPC),指令存储器(IMEM),寄存器文件(RegFile),CP0

1.32 CLZ



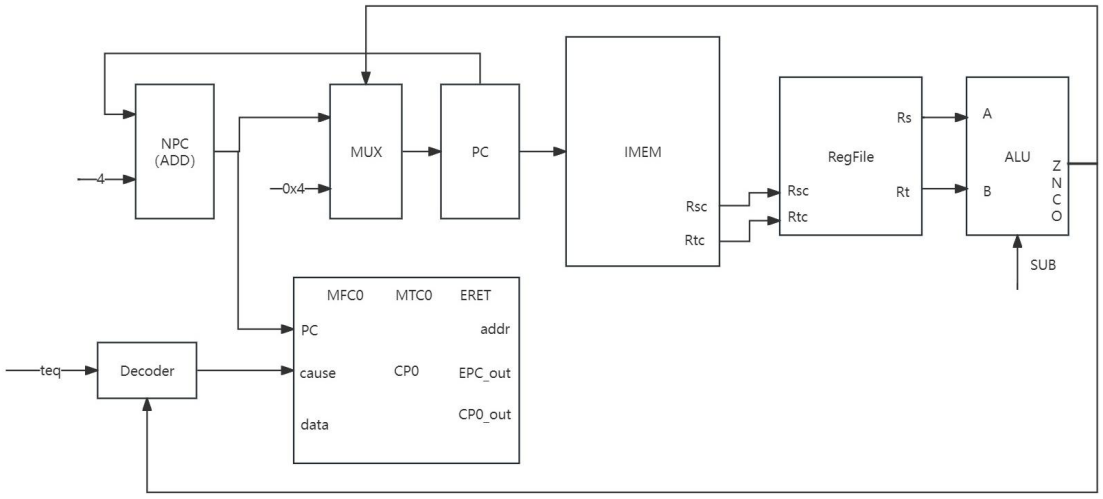
格式：CLZ rs,rt

操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> MUX

Rs -> CLZ_in
CLZ_out -> Rd
```

所用部件：PC 寄存器(PC、NPC),指令存储器(IMEM),寄存器文件(RegFile),CP0

1.33 TEQ



格式：TEQ rs,rt

操作：

```
PC -> IMEM
PC + 4 -> NPC
NPC -> MUX
0x4 -> MUX
```

```
Rs -> ALU_A
Rt -> ALU_B
ZERO -> Rs - Rt = 0
if ZERO:
PC -> CP0
```

所用部件：PC 寄存器(PC、NPC)，指令存储器(IMEM)，寄存器文件(RegFile)，译码器(DECODER)，多路选择器(MUX)，ALU， CP0

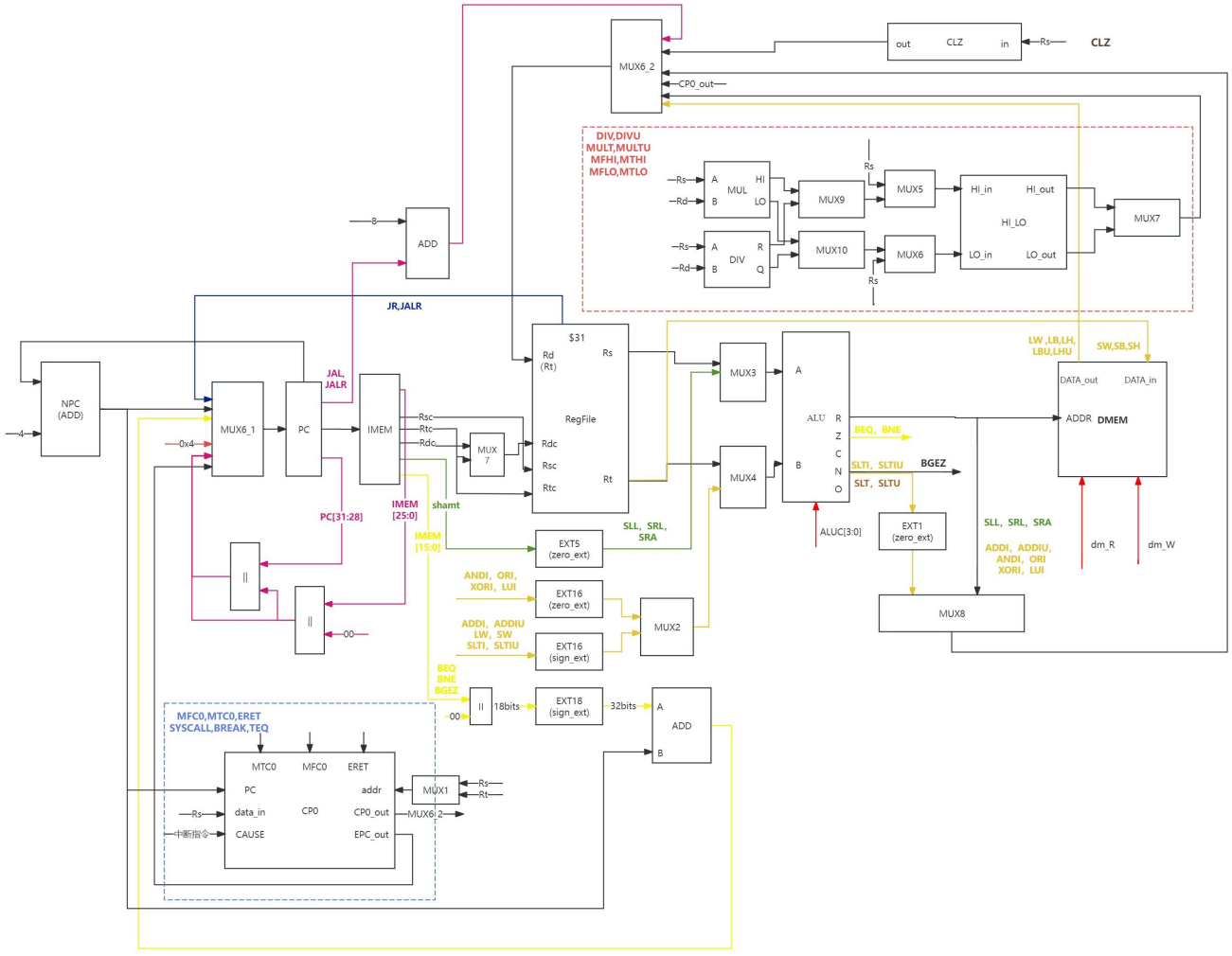
2. 总体数据通路图

2.1 54 条指令 CPU 所用部件

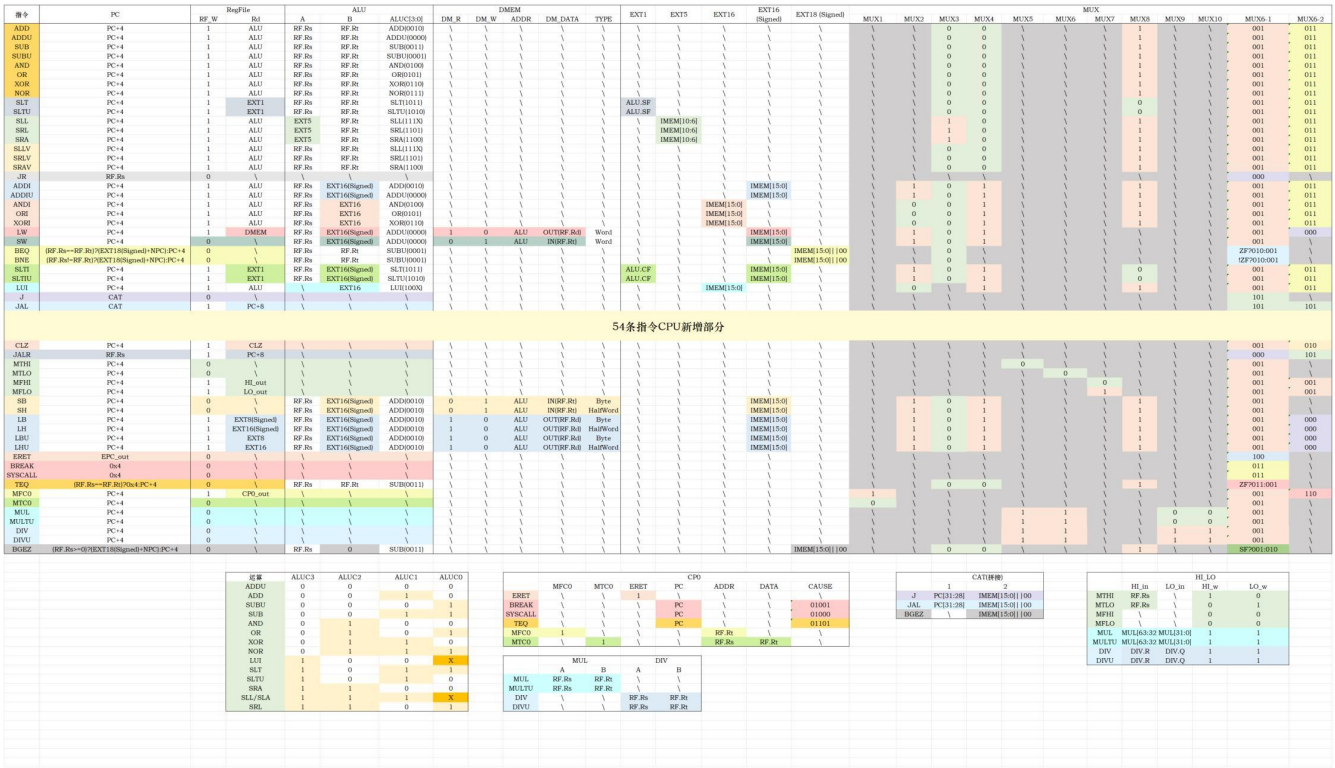
PC,NPC,IMEM,RegFile,ALU,DMEM,||,ADD,EXT1,EXT5,EXT16,EXT18,MUX

* Decoder,Controler未在图中画出,MUX6_1与MUX6_2均为六选一多路选择器

2.2 CPU 总数据通路图



2.3 指令操作表



三、模块建模

1. sccomp_dataflow 模块

用于连接 CPU、DMEM 和 IMEM，是哈佛结构 CPU 的顶层模块。

```
module sccomp_dataflow(
input  clk_in,
input  reset,
output [31:0] inst,
output [31:0] pc
);
wire [31:0] pc_out;
wire [31:0] dm_addr_temp;
wire [31:0] im_addr_in;
wire [31:0] im_instr_out;
assign im_addr_in = pc_out -
32'h00400000;

wire dm_ena;
wire dm_r, dm_w;
wire [31:0] dm_addr;
wire [31:0] dm_data_out;
wire [31:0] dm_data_w;
wire sb_flag;
wire sh_flag;
wire sw_flag;
wire lb_flag;
wire lh_flag;
wire lbu_flag;
wire lhu_flag;
wire lw_flag;

assign dm_addr = dm_addr_temp -
32'h10010000;
assign pc = pc_out;
assign inst = im_instr_out;
```

```
IMEM imem(
.im_addr_in(im_addr_in[12:2]),
.im_instr_out(im_instr_out) );

DMEM dmem(
    .dm_clk(clk_in),
    .dm_ena(dm_ena),
    .dm_r(dm_r),
    .dm_w(dm_w),
    .sb_flag(sb_flag),
    .sh_flag(sh_flag),
    .sw_flag(sw_flag),
    .lb_flag(lb_flag),
    .lh_flag(lh_flag),
    .lbu_flag(lbu_flag),
    .lhu_flag(lhu_flag),
    .lw_flag(lw_flag),
    .dm_addr(dm_addr[6:0]),
    .dm_data_in(dm_data_w),
    .dm_data_out(dm_data_out)
    );

cpu sccpu(
    .clk(clk_in),
    .ena(1'b1),
    .rst(reset),
    .instr_in(im_instr_out),
    .dm_data(dm_data_out),
    .dm_ena(dm_ena),
    .dm_w(dm_w),
```

```
    .dm_r(dm_r),
    .pc_out(pc_out),
    .dm_addr(dm_addr_temp),
    .dm_data_w(dm_data_w),
    .sb_flag(sb_flag),
    .sh_flag(sh_flag),
    .sw_flag(sw_flag),
    .lb_flag(lb_flag),
    .lh_flag(lh_flag),
    .lbu_flag(lbu_flag),
    .lhu_flag(lhu_flag),
    .lw_flag(lw_flag)
    );

endmodule
```

2. cpu 模块

调用 ALU 模块、PC 模块和 RegFile 模块，并通过Decoder模块和Controler模块对信号进行处理，是本次实验的核心部分。

```
module cpu(
    input clk,
    input ena,
    input rst,
    input [31:0] instr_in,
    input [31:0] dm_data,
    output dm_ena,
    output dm_w,
    output dm_r,
    output [31:0] pc_out,
    output [31:0] dm_addr,
    output [31:0] dm_data_w,
    output sb_flag,
    output sh_flag,
    output sw_flag,
    output lb_flag,
    output lh_flag,
    output lbu_flag,
    output lhu_flag,
    output lw_flag
    );
parameter ADD   = 6'd0;
parameter ADDU  = 6'd1;
parameter SUB   = 6'd2;
parameter SUBU  = 6'd3;
parameter AND   = 6'd4;
parameter OR    = 6'd5;
parameter XOR   = 6'd6;
parameter NOR   = 6'd7;
parameter SLT   = 6'd8;
parameter SLTU  = 6'd9;
parameter SLL   = 6'd10;
parameter SRL   = 6'd11;
parameter SRA   = 6'd12;
parameter SLLV  = 6'd13;
parameter SRLV  = 6'd14;
parameter SRAV  = 6'd15;
parameter JR    = 6'd16;
parameter ADDI  = 6'd17;
parameter ADDIU = 6'd18;
parameter ANDI  = 6'd19;
parameter ORI   = 6'd20;
parameter XORI  = 6'd21;
parameter LW    = 6'd22;
parameter SW    = 6'd23;
parameter BEQ   = 6'd24;
parameter BNE   = 6'd25;
parameter SLTI  = 6'd26;
parameter SLTIU = 6'd27;
parameter LUI   = 6'd28;
parameter J     = 6'd29;
parameter JAL   = 6'd30;
parameter CLZ     = 6'd31;
parameter JALR    = 6'd32;
parameter MTHI    = 6'd33;
parameter MTLO    = 6'd34;
parameter MFHI    = 6'd35;
parameter MFLO    = 6'd36;
parameter SB      = 6'd37;
parameter SH      = 6'd38;
parameter LB      = 6'd39;
parameter LH      = 6'd40;
parameter LBU     = 6'd41;
parameter LHU     = 6'd42;
parameter ERET    = 6'd43;
parameter BREAK   = 6'd44;
parameter SYSCALL = 6'd45;
parameter TEQ     = 6'd46;
parameter MFC0    = 6'd47;
parameter MTC0    = 6'd48;
parameter MUL     = 6'd49;
parameter MULTU   = 6'd50;
parameter DIV     = 6'd51;
parameter DIVU    = 6'd52;
parameter BGEZ    = 6'd53;

/* Decoder */
wire [53:0] op_flags;
wire [4:0] RsC;
wire [4:0] RtC;
wire [4:0] RdC;
wire [4:0] shamt;
wire [15:0] immediate;
wire [25:0] address;

/* Controler */
wire reg_w;
wire [4:0] cause;
wire [10:0] mux;
wire [2:0]  mux6_1;
wire [2:0]  mux6_2;
wire [4:0] ext_ena;
wire cat_ena;

/* ALU */
wire [31:0] a, b;
wire [3:0]  aluc;
wire [31:0] alu_data_out;
wire zero, carry, negative, overflow;

/* RegFile */
wire [31:0] Rd_data_in;
wire [31:0] Rs_data_out;
wire [31:0] Rt_data_out;
```

```
/* PC寄存器 */
wire [31:0] pc_addr_in;
wire [31:0] pc_addr_out;

/* HI_LO寄存器 */
wire HI_w;
wire LO_w;
wire [31:0] HI_out;
wire [31:0] LO_out;

/* DIV模块用 */
wire sign_flag;
wire [31:0] DIV_A;
wire [31:0] DIV_B;
wire [31:0] DIV_R;
wire [31:0] DIV_Q;

/* MUL模块用 */
wire [31:0] MUL_A;
wire [31:0] MUL_B;
wire [31:0] MUL_HI;
wire [31:0] MUL_LO;

/* CP0模块用 */
wire [31:0] cp0_out;
wire [31:0] epc_out;
/* CLZ模块用 */
wire [31:0] CLZ_out;

/* 符号、数据扩展器线路 */
wire [31:0] ext1_out;
wire [31:0] ext5_out;
wire [31:0] ext16_out;
wire signed [31:0] ext16_out_signed;
wire signed [31:0] ext18_out_signed;

assign ext1_out         =
(op_flags[SLT]  || op_flags[SLTU]) ?
negative :

(op_flags[SLTI] || op_flags[SLTIU]) ?
carry : 32'hz;
assign ext5_out         =
(op_flags[SLL]  || op_flags[SRL]   ||
op_flags[SRA]) ? shamt : 32'hz;
assign ext16_out        =
(op_flags[ANDI] || op_flags[ORI]   ||
op_flags[XORI] || op_flags[LUI]) ?
{ 16'h0 , immediate[15:0] } : 32'hz;
assign ext16_out_signed =
(op_flags[ADDI] || op_flags[ADDIU] ||
op_flags[LW]   || op_flags[SW] ||

op_flags[SLTI] || op_flags[SLTIU] ||
op_flags[SB]   || op_flags[SH] ||

op_flags[LB]   || op_flags[LH]    ||
op_flags[LBU]  || op_flags[LHU]) ?
{ {16{immediate[15]}} ,
immediate[15:0] } : 32'hz;
assign ext18_out_signed =
(op_flags[BEQ]  || op_flags[BNE]   ||
op_flags[BGEZ]) ?
{{14{immediate[15]}}, immediate[15:0],
2'b0} : 32'hz;

/* ||拼接器线路 */
wire [31:0] cat_out;
assign cat_out = cat_ena ?
{pc_out[31:28], address[25:0], 2'h0} :
32'hz;

/* NPC线路 */
wire [31:0] npc;
wire [31:0] add_out_61;    //加法器，对
应6选1通路
wire [31:0] add_out_62;    //加法器，对
应第二个6选1通路
assign npc = pc_addr_out + 4;

/* 多路选择器线路 */
wire [31:0] mux1_out;
wire [31:0] mux2_out;
wire [31:0] mux3_out;
wire [31:0] mux4_out;
wire [31:0] mux5_out;
wire [31:0] mux6_out;
wire [31:0] mux7_out;
wire [31:0] mux8_out;
wire [31:0] mux9_out;
wire [31:0] mux10_out;
wire [31:0] mux6_1_out;
wire [31:0] mux6_2_out;

assign mux1_out  = mux[1]  ?
Rt_data_out : Rs_data_out;
assign mux2_out  = mux[2]  ?
ext16_out_signed : ext16_out;
assign mux3_out  = mux[3]  ?
ext5_out : Rs_data_out;
assign mux4_out  = mux[4]  ?
mux2_out : Rt_data_out;
assign mux5_out  = mux[5]  ?
mux9_out : Rs_data_out;
assign mux6_out  = mux[6]  ?
mux10_out: Rs_data_out;
assign mux7_out  = mux[7]  ?
LO_out   : HI_out;
assign mux8_out  = mux[8]  ?
alu_data_out : ext1_out;
assign mux9_out  = mux[9]  ? DIV_R :
MUL_HI;
assign mux10_out = mux[10] ? DIV_Q :
MUL_LO;

/* PC线路 */
assign pc_addr_in = mux6_1_out;
assign add_out_61 = ext18_out_signed
+ npc;
assign add_out_62 = pc_addr_out + 4;

/* ALU 接线口 */
assign a = mux3_out;
assign b = op_flags[BGEZ] ? 32'd0 :
mux4_out;

/* IMEM接口 */
assign pc_out = pc_addr_out;

/* DMEM接口 */
assign dm_ena    = (dm_r || dm_w) ?
1'b1 : 1'b0;
```

```
assign dm_addr   = alu_data_out;
assign dm_data_w = Rt_data_out;
assign sb_flag   = op_flags[SB];
assign sh_flag   = op_flags[SH];
assign sw_flag   = op_flags[SW];
assign lb_flag   = op_flags[LB];
assign lh_flag   = op_flags[LH];
assign lbu_flag  = op_flags[LBU];
assign lhu_flag  = op_flags[LHU];
assign lw_flag   = op_flags[LW];

/* 寄存器堆线路 */
assign Rd_data_in = mux6_2_out;

/* DIV MUL用 */
assign sign_flag = op_flags[MUL] ||
op_flags[DIV] ? 1'b1 : 1'b0;
assign MUL_A = op_flags[MUL] ||
op_flags[MULTU] ? Rs_data_out : 32'hz;
assign MUL_B = op_flags[MUL] ||
op_flags[MULTU] ? Rt_data_out : 32'hz;
assign DIV_A = op_flags[DIV] ||
op_flags[DIVU]  ? Rs_data_out : 32'hz;
assign DIV_B = op_flags[DIV] ||
op_flags[DIVU] ? Rt_data_out : 32'hz;

/* 实例化译码器 */
Decoder Decoder_inst(
    .instr_in(instr_in),
    .op_flags(op_flags),
    .RsC(RsC),
    .RtC(RtC),
    .RdC(RdC),
    .shamt(shamt),
    .immediate(immediate),
    .address(address)
    );

/* 实例化控制器 */
Controler Controler_inst(
    .op_flags(op_flags),
    .zero_flag(zero),
    .sign_flag(negative),
    .reg_w(reg_w),
    .aluc(aluc),
    .dm_r(dm_r),
    .dm_w(dm_w),
    .HI_w(HI_w),
    .LO_w(LO_w),
    .cause(cause),
    .ext_ena(ext_ena),
    .cat_ena(cat_ena),
    .mux(mux),
    .mux6_1(mux6_1),
    .mux6_2(mux6_2)
    );

/* 实例化ALU */
ALU ALU_inst(
    .A(a),                          //对应
A接口
    .B(b),                          //对应
B接口
    .ALUC(aluc),
//ALUC四位操作指令
    .alu_data_out(alu_data_out),//输出
数据
    .zero(zero),                    //ZF标
志位，BEQ/BNE使用
    .carry(carry),                  //CF标
志位，SLTI/SLTIU使用
    .negative(negative),
//NF(SF)标志位，SLT/SLTU使用
    .overflow(overflow)             //OF标
志位，其实没有用到
    );

/* 实例化寄存器堆 */
regfile cpu_ref(
    .reg_clk(clk),
    .reg_ena(ena),
    .rst(rst),
    .reg_w(reg_w),
    .RdC(RdC),
    .RtC(RtC),
    .RsC(RsC),
    .Rd_data_in(Rd_data_in),
    .Rs_data_out(Rs_data_out),
    .Rt_data_out(Rt_data_out)
    );

/* 实例化PC寄存器 */
PC PC_inst(
    .pc_clk(clk),
    .pc_ena(ena),
    .rst(rst),
    .pc_addr_in(pc_addr_in),
    .pc_addr_out(pc_addr_out)
    );

/* 实例化HI_LO寄存器 */
HI_LO HI_LO_inst(
    .HI_LO_clk(clk),
    .HI_LO_ena(ena),
    .HI_LO_rst(rst),
    .HI_in(mux5_out),
    .LO_in(mux6_out),
    .HI_w(HI_w),
    .LO_w(LO_w),
    .HI_out(HI_out),
    .LO_out(LO_out)
    );

MUX MUX6_1(                         //选PC
    .chosen(mux6_1),
    .line0(Rs_data_out),
    .line1(npc),
    .line2(add_out_61),
    .line3(32'h00400004),           //出口
    .line4(epc_out),
    .line5(cat_out),
    .line6(32'bz),
    .line7(32'bz),
    .MUX_out(mux6_1_out)
    );

MUX MUX6_2(
    .chosen(mux6_2),
    .line0(dm_data),
    .line1(mux7_out),
    .line2(CLZ_out),
```

```
    .line3(mux8_out),
    .line4(32'bz),
    .line5(add_out_62),
    .line6(cp0_out),
    .line7(32'bz),
    .MUX_out(mux6_2_out)
    );

DIV DIV_inst(
    .sign_flag(sign_flag),
    .A(DIV_A),
    .B(DIV_B),
    .R(DIV_R),
    .Q(DIV_Q)
    );

MUL MUL_inst(
    .sign_flag(sign_flag),
    .A(MUL_A),
    .B(MUL_B),
    .HI(MUL_HI),
    .LO(MUL_LO)
    );

CP0 CP0_inst(
    .cp0_clk(clk),
    .cp0_rst(rst),
    .cp0_ena(ena),
    .MFC0(op_flags[MFC0]),
    .MTC0(op_flags[MTC0]),
    .ERET(op_flags[ERET]),
    .PC(npc),
    .addr(mux1_out),
    .cause(cause),
    .data_in(Rt_data_out),
    .CP0_out(cp0_out),
    .EPC_out(epc_out)
    );

CLZ CLZ_inst(
    .CLZ_in(Rs_data_out),
    .CLZ_out(CLZ_out)
    );

endmodule
```

3. IMEM 模块

用于封装 ROM 的 IP 核，进行读取指令

```
module IMEM(
    input [10:0] im_addr_in,
    output [31:0] im_instr_out
    );

simulate imem(
    .a(im_addr_in),
    .spo(im_instr_out)
    );
Endmodule
```

4. DMEM 模块

用于读写数据到内存

```
module DMEM(
    input dm_clk,
    input dm_ena,
    input dm_r,
    input dm_w,
    input sb_flag,
    input sh_flag,
    input sw_flag,
    input lb_flag,
    input lh_flag,
    input lbu_flag,
    input lhu_flag,
    input lw_flag,
    input [6:0] dm_addr,
    input [31:0] dm_data_in,
    output [31:0] dm_data_out
    );

reg [31:0] dmem [31:0];
assign dm_data_out = (dm_ena && dm_r && !dm_w) ?
(lb_flag ? { {24{dmem[dm_addr][7]}} , dmem[dm_addr][7:0] } :
(lbu_flag ? { 24'h0 , dmem[dm_addr][7:0] } :
(lh_flag ? { {16{dmem[dm_addr >> 1][15]}} , dmem[dm_addr >> 1][15:0] } :
(lhu_flag ? { 16'h0 , dmem[dm_addr >> 1][15:0] } :
(lw_flag ? dmem[dm_addr >> 2]: 32'bz))))) : 32'bz;
```

```
always @(negedge dm_clk)
begin
    if(dm_ena && dm_w &&!dm_r) begin
        if(sb_flag)
            dmem[dm_addr][7:0] <= dm_data_in[7:0];
        else if(sh_flag)
            dmem[dm_addr >> 1][15:0] <= dm_data_in[15:0];
        else if(sw_flag)
            dmem[dm_addr >> 2] <= dm_data_in;
    end
end

endmodule
```

5. regfile 模块

寄存器堆，用于配合指令的输入输出。

```
module regfile(
input  reg_clk,             //时钟信号
input  reg_ena,             //使能信号
input  rst_n,               //复位信号，高电平有效（检测上升沿）
input  reg_w,               //写信号
input  [4:0] RdC,           //Rd 地址（写入端）
input  [4:0] RtC,           //Rt 地址（输出端）
input  [4:0] RsC,           //Rs 地址（输出端）
input  [31:0] Rd_data_in,   //输入数据（需拉高reg_w）
output [31:0] Rs_data_out,  //Rs对应的输出数据
output [31:0] Rt_data_out   //Rt对应的输出数据
);
reg [31:0] array_reg [31:0];
always @(negedge reg_clk or posedge
rst_n)
begin
    if(rst_n && reg_ena) begin
        array_reg[0]  = 32'h0;
        array_reg[1]  = 32'h0;
        array_reg[2]  = 32'h0;
        array_reg[3]  = 32'h0;
        array_reg[4]  = 32'h0;
        array_reg[5]  = 32'h0;
        array_reg[6]  = 32'h0;
        array_reg[7]  = 32'h0;
        array_reg[8]  = 32'h0;
        array_reg[9]  = 32'h0;
        array_reg[10] = 32'h0;
        array_reg[11] = 32'h0;
        array_reg[12] = 32'h0;
        array_reg[13] = 32'h0;
        array_reg[14] = 32'h0;
        array_reg[15] = 32'h0;
        array_reg[16] = 32'h0;
        array_reg[17] = 32'h0;
        array_reg[18] = 32'h0;
        array_reg[19] = 32'h0;
        array_reg[20] = 32'h0;
        array_reg[21] = 32'h0;
        array_reg[22] = 32'h0;
        array_reg[23] = 32'h0;
        array_reg[24] = 32'h0;
        array_reg[25] = 32'h0;
        array_reg[26] = 32'h0;
        array_reg[27] = 32'h0;
        array_reg[28] = 32'h0;
        array_reg[29] = 32'h0;
        array_reg[30] = 32'h0;
        array_reg[31] = 32'h0;
    end
    else if(reg_ena&&reg_w&&(RdC!=
5'h0))
    array_reg[RdC] = Rd_data_in;
end

assign Rs_data_out = reg_ena ?
array_reg[RsC] : 32'bz;
assign Rt_data_out = reg_ena ?
array_reg[RtC] : 32'bz;
endmodule
```

6. ALU 模块

CPU 中的运算器，是 CPU 的计算核心。

```
module alu(
input  [31:0] a,b,         //ALU 输入a,b
input  [3:0]  aluc, //ALU 控制信号
output reg [31:0] r,  //ALU 运算结果
output reg zero,          //zero 标志位
output reg carry,         //carry 标志位
output reg negative,  //negative 标志位
```

```
output reg  overflow   //overflow 标志位
);
module ALU(
    input  [31:0] A,
    input  [31:0] B,
    input  [3:0] ALUC,
    output [31:0] alu_data_out,
    output zero,
    output carry,
    output negative,
    output overflow
    );
parameter ADDU = 4'b0000;
parameter ADD  = 4'b0010;
parameter SUBU = 4'b0001;
parameter SUB  = 4'b0011;
parameter AND  = 4'b0100;
parameter OR   = 4'b0101;
parameter XOR  = 4'b0110;
parameter NOR  = 4'b0111;
parameter LUI1 = 4'b1000;
parameter LUI2 = 4'b1001;
parameter SLT  = 4'b1011;
parameter SLTU = 4'b1010;
parameter SRA  = 4'b1100;
parameter SLL  = 4'b1110;
parameter SLA  = 4'b1111;
parameter SRL  = 4'b1101;

reg [32:0] result;
wire signed [31:0] signedA,signedB;
assign signedA = A;
assign signedB = B;

always @(*)
begin
case(ALUC)
ADDU: begin result <= A + B;
end
ADD:  begin result <= signedA+signedB;
end
SUBU: begin result <= A - B;
end
SUB:  begin result <= signedA-signedB;
end
AND:  begin result <= A & B;
end
OR:   begin result <= A | B;
end
XOR:  begin result <= A ^ B;
end
NOR:  begin result <= ~(A | B);
end
LUI1,LUI2:  begin result <=
{ B[15:0] , 16'b0 };            end
SLT:  begin result <= signedA-signedB;
end
SLTU: begin result <= A - B;
end
SRA:  begin result <= signedB >>>
signedA;            end
SLL,SLA: begin result <= B << A;
end
SRL:  begin result <= B >> A;
end
    endcase
end

assign alu_data_out = result[31:0];
assign zero = (result == 32'b0) ? 1:0;
assign carry = result[32];
assign negative =  (ALUC == SLT ?
(signedA < signedB) : ((ALUC ==
SLTU) ? (A < B) : 1'b0));
assign overflow = result[32];

Endmodule
```

7. Decoder 模块

指令的译码器，根据指令置31个标识符中某一个，并将信息分为Rsc，Rdc，Rtc，shamt，immediate，address。

```
module Decoder(
    input  [31:0] instr_in,
    output [53:0] op_flags,
    output [4:0]  RsC,
    output [4:0]  RtC,
    output [4:0]  RdC,
    output [4:0]  shamt,
    output [15:0] immediate,
    output [25:0] address
    );
parameter ADD_OPE   = 6'b100000;
parameter ADDU_OPE  = 6'b100001;
parameter SUB_OPE   = 6'b100010;
parameter SUBU_OPE  = 6'b100011;
parameter AND_OPE   = 6'b100100;
parameter OR_OPE    = 6'b100101;
parameter XOR_OPE   = 6'b100110;
parameter NOR_OPE   = 6'b100111;
parameter SLT_OPE   = 6'b101010;
parameter SLTU_OPE  = 6'b101011;
parameter SLL_OPE   = 6'b000000;
parameter SRL_OPE   = 6'b000010;
parameter SRA_OPE   = 6'b000011;
parameter SLLV_OPE  = 6'b000100;
parameter SRLV_OPE  = 6'b000110;
parameter SRAV_OPE  = 6'b000111;
parameter JR_OPE    = 6'b001000;
parameter ADDI_OPE  = 6'b001000;
```

```
parameter ADDIU_OPE = 6'b001001;
parameter ANDI_OPE  = 6'b001100;
parameter ORI_OPE   = 6'b001101;
parameter XORI_OPE  = 6'b001110;
parameter LW_OPE    = 6'b100011;
parameter SW_OPE    = 6'b101011;
parameter BEQ_OPE   = 6'b000100;
parameter BNE_OPE   = 6'b000101;
parameter SLTI_OPE  = 6'b001010;
parameter SLTIU_OPE = 6'b001011;
parameter LUI_OPE   = 6'b001111;
parameter J_OPE     = 6'b000010;
parameter JAL_OPE   = 6'b000011;

parameter CLZ_OPE     = 6'b100000;
parameter JALR_OPE    = 6'b001001;
parameter MTHI_OPE    = 6'b010001;
parameter MFHI_OPE    = 6'b010000;
parameter MTLO_OPE    = 6'b010011;
parameter MFLO_OPE    = 6'b010010;
parameter SB_OPE      = 6'b101000;
parameter SH_OPE      = 6'b101001;
parameter LB_OPE      = 6'b100000;
parameter LH_OPE      = 6'b100001;
parameter LBU_OPE     = 6'b100100;
parameter LHU_OPE     = 6'b100101;
parameter ERET_OPE    = 6'b011000;
parameter BREAK_OPE   = 6'b001101;
parameter SYSCALL_OPE = 6'b001100;
parameter TEQ_OPE     = 6'b110100;
parameter MFC0_OPE    = 5'b00000;
parameter MTC0_OPE    = 5'b00100;
parameter MUL_OPE     = 6'b011000;
parameter MULTU_OPE   = 6'b011001;
parameter DIV_OPE     = 6'b011010;
parameter DIVU_OPE    = 6'b011011;
parameter BGEZ_OPE    = 6'b000001;

parameter ADD   = 6'd0;
parameter ADDU  = 6'd1;
parameter SUB   = 6'd2;
parameter SUBU  = 6'd3;
parameter AND   = 6'd4;
parameter OR    = 6'd5;
parameter XOR   = 6'd6;
parameter NOR   = 6'd7;
parameter SLT   = 6'd8;
parameter SLTU  = 6'd9;
parameter SLL   = 6'd10;
parameter SRL   = 6'd11;
parameter SRA   = 6'd12;
parameter SLLV  = 6'd13;
parameter SRLV  = 6'd14;
parameter SRAV  = 6'd15;
parameter JR    = 6'd16;
parameter ADDI  = 6'd17;
parameter ADDIU = 6'd18;
parameter ANDI  = 6'd19;
parameter ORI   = 6'd20;
parameter XORI  = 6'd21;
parameter LW    = 6'd22;
parameter SW    = 6'd23;
parameter BEQ   = 6'd24;
parameter BNE   = 6'd25;
parameter SLTI  = 6'd26;
parameter SLTIU = 6'd27;
parameter LUI   = 6'd28;
parameter J     = 6'd29;
parameter JAL   = 6'd30;

parameter CLZ     = 6'd31;
parameter JALR    = 6'd32;
parameter MTHI    = 6'd33;
parameter MTLO    = 6'd34;
parameter MFHI    = 6'd35;
parameter MFLO    = 6'd36;
parameter SB      = 6'd37;
parameter SH      = 6'd38;
parameter LB      = 6'd39;
parameter LH      = 6'd40;
parameter LBU     = 6'd41;
parameter LHU     = 6'd42;
parameter ERET    = 6'd43;
parameter BREAK   = 6'd44;
parameter SYSCALL = 6'd45;
parameter TEQ     = 6'd46;
parameter MFC0    = 6'd47;
parameter MTC0    = 6'd48;
parameter MULT     = 6'd49;
parameter MULTU   = 6'd50;
parameter DIV     = 6'd51;
parameter DIVU    = 6'd52;
parameter BGEZ    = 6'd53;

assign op_flags[ADD]   = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
ADD_OPE )) ? 1'b1 : 1'b0;
assign op_flags[ADDU]  = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
ADDU_OPE)) ? 1'b1 : 1'b0;
assign op_flags[SUB]   = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
SUB_OPE )) ? 1'b1 : 1'b0;
assign op_flags[SUBU]  = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
SUBU_OPE)) ? 1'b1 : 1'b0;
assign op_flags[AND]   = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
AND_OPE )) ? 1'b1 : 1'b0;
assign op_flags[OR]    = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
OR_OPE  )) ? 1'b1 : 1'b0;
assign op_flags[XOR]   = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
XOR_OPE )) ? 1'b1 : 1'b0;
assign op_flags[NOR]   = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
NOR_OPE )) ? 1'b1 : 1'b0;
assign op_flags[SLT]   = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
SLT_OPE )) ? 1'b1 : 1'b0;
assign op_flags[SLTU]  = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
SLTU_OPE)) ? 1'b1 : 1'b0;
```

```
assign op_flags[SLL]   = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
SLL_OPE )) ? 1'b1 : 1'b0;
assign op_flags[SRL]   = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
SRL_OPE )) ? 1'b1 : 1'b0;
assign op_flags[SRA]   = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
SRA_OPE )) ? 1'b1 : 1'b0;
assign op_flags[SLLV]  = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
SLLV_OPE)) ? 1'b1 : 1'b0;
assign op_flags[SRLV]  = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
SRLV_OPE)) ? 1'b1 : 1'b0;
assign op_flags[SRAV]  = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
SRAV_OPE)) ? 1'b1 : 1'b0;
assign op_flags[JR]    = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
JR_OPE  )) ? 1'b1 : 1'b0;
assign op_flags[ADDI]  = (instr_in[31:26] == ADDI_OPE ) ? 1'b1 : 1'b0;
assign op_flags[ADDIU] = (instr_in[31:26] == ADDIU_OPE) ? 1'b1 : 1'b0;
assign op_flags[ANDI]  = (instr_in[31:26] == ANDI_OPE ) ? 1'b1 : 1'b0;
assign op_flags[ORI]   = (instr_in[31:26] == ORI_OPE  ) ? 1'b1 : 1'b0;
assign op_flags[XORI]  = (instr_in[31:26] == XORI_OPE ) ? 1'b1 : 1'b0;
assign op_flags[LW]    = (instr_in[31:26] == LW_OPE   ) ? 1'b1 : 1'b0;
assign op_flags[SW]    = (instr_in[31:26] == SW_OPE   ) ? 1'b1 : 1'b0;
assign op_flags[BEQ]   = (instr_in[31:26] == BEQ_OPE  ) ? 1'b1 : 1'b0;
assign op_flags[BNE]   = (instr_in[31:26] == BNE_OPE  ) ? 1'b1 : 1'b0;
assign op_flags[SLTI]  = (instr_in[31:26] == SLTI_OPE ) ? 1'b1 : 1'b0;
assign op_flags[SLTIU] = (instr_in[31:26] == SLTIU_OPE) ? 1'b1 : 1'b0;
assign op_flags[LUI]   = (instr_in[31:26] == LUI_OPE  ) ? 1'b1 : 1'b0;
assign op_flags[J]     = (instr_in[31:26] == J_OPE    ) ? 1'b1 : 1'b0;
assign op_flags[JAL]   = (instr_in[31:26] == JAL_OPE  ) ? 1'b1 : 1'b0;

assign op_flags[CLZ]     = ((instr_in[31:26] == 6'b011100) && (instr_in[5:0] ==
CLZ_OPE )) ? 1'b1 : 1'b0;
assign op_flags[JALR]    = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
JALR_OPE )) ? 1'b1 : 1'b0;
assign op_flags[MTHI]    = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
MTHI_OPE )) ? 1'b1 : 1'b0;
assign op_flags[MTLO]    = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
MTLO_OPE )) ? 1'b1 : 1'b0;
assign op_flags[MFHI]    = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
MFHI_OPE )) ? 1'b1 : 1'b0;
assign op_flags[MFLO]    = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
MFLO_OPE )) ? 1'b1 : 1'b0;
assign op_flags[SB]      = (instr_in[31:26] == SB_OPE  ) ? 1'b1 : 1'b0;
assign op_flags[SH]      = (instr_in[31:26] == SH_OPE  ) ? 1'b1 : 1'b0;
assign op_flags[LB]      = (instr_in[31:26] == LB_OPE  ) ? 1'b1 : 1'b0;
assign op_flags[LH]      = (instr_in[31:26] == LH_OPE  ) ? 1'b1 : 1'b0;
assign op_flags[LBU]     = (instr_in[31:26] == LBU_OPE ) ? 1'b1 : 1'b0;
assign op_flags[LHU]     = (instr_in[31:26] == LHU_OPE ) ? 1'b1 : 1'b0;
assign op_flags[ERET]    = ((instr_in[31:26] == 6'b010000) && (instr_in[5:0] ==
ERET_OPE) && (instr_in[25:21] == 5'b10000)) ? 1'b1 : 1'b0;
assign op_flags[BREAK]   = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
BREAK_OPE   )) ? 1'b1 : 1'b0;
assign op_flags[SYSCALL] = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
SYSCALL_OPE )) ? 1'b1 : 1'b0;
assign op_flags[TEQ]     = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
TEQ_OPE     )) ? 1'b1 : 1'b0;
assign op_flags[MFC0]    = ((instr_in[31:26] == 6'b010000) && (instr_in[5:0] ==
6'h0) && (instr_in[25:21] == MFC0_OPE)) ? 1'b1 : 1'b0;
assign op_flags[MTC0]    = ((instr_in[31:26] == 6'b010000) && (instr_in[5:0] ==
6'h0) && (instr_in[25:21] == MTC0_OPE)) ? 1'b1 : 1'b0;
assign op_flags[MULT]     = ((instr_in[31:26] == 6'b0) && (instr_in[5:0] ==
MUL_OPE     )) ? 1'b1 : 1'b0;
assign op_flags[MULTU]   = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
MULTU_OPE   )) ? 1'b1 : 1'b0;
assign op_flags[DIV]     = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
DIV_OPE     )) ? 1'b1 : 1'b0;
assign op_flags[DIVU]    = ((instr_in[31:26] == 6'h0) && (instr_in[5:0] ==
DIVU_OPE    )) ? 1'b1 : 1'b0;
assign op_flags[BGEZ]    = ((instr_in[31:26] == BGEZ_OPE) && (instr_in[20:16]
```

```
== 5'b00001) ) ? 1'b1 : 1'b0;


assign RsC =
(op_flags[ADD]  || op_flags[ADDU] || op_flags[SUB]  || op_flags[SUBU]  ||
op_flags[AND]  || op_flags[OR]   || op_flags[XOR]  || op_flags[NOR]   ||
op_flags[SLT]  || op_flags[SLTU] || op_flags[SLLV] || op_flags[SRLV]  ||
op_flags[SRAV] || op_flags[JR]   || op_flags[ADDI] || op_flags[ADDIU] ||
op_flags[ANDI] || op_flags[ORI]  || op_flags[XORI] || op_flags[LW]    ||
op_flags[SW]   || op_flags[BEQ]  || op_flags[BNE]  || op_flags[SLTI]  ||
op_flags[SLTIU]|| op_flags[CLZ]  || op_flags[JALR] || op_flags[MTHI]  ||
op_flags[MTLO] || op_flags[SB]   || op_flags[SH]   || op_flags[LB]    ||
op_flags[LH]   || op_flags[LBU]  || op_flags[LHU]  || op_flags[TEQ]   ||
op_flags[MULT] || op_flags[MULTU]|| op_flags[DIV]  || op_flags[DIVU]  ||
op_flags[BGEZ]) ? instr_in[25:21] :
(op_flags[MTC0] ? instr_in[15:11] : 5'hz);

assign RtC =
(op_flags[ADD]  || op_flags[ADDU]  || op_flags[SUB]   || op_flags[SUBU] ||
op_flags[AND]  || op_flags[OR]    || op_flags[XOR]   || op_flags[NOR]  ||
op_flags[SLT]  || op_flags[SLTU]  || op_flags[SLL]   || op_flags[SRL]  ||
op_flags[SRA]  || op_flags[SLLV]  || op_flags[SRLV]  || op_flags[SRAV] ||
op_flags[SW]   || op_flags[BEQ]   || op_flags[BNE]   || op_flags[SB]   ||
op_flags[SH]   || op_flags[TEQ]   || op_flags[MTC0]  || op_flags[MULT] ||
op_flags[MULTU]|| op_flags[DIV]   || op_flags[DIVU]) ? instr_in[20:16] :
(op_flags[MFC0] ? instr_in[15:11] : 5'hz);

assign RdC =
(op_flags[ADD]  || op_flags[ADDU]  || op_flags[SUB]  || op_flags[SUBU]  ||
op_flags[AND]  || op_flags[OR]    || op_flags[XOR]  || op_flags[NOR]   ||
op_flags[SLT]  || op_flags[SLTU]  || op_flags[SLL]  || op_flags[SRL]   ||
op_flags[SRA]  || op_flags[SLLV]  || op_flags[SRLV] || op_flags[SRAV]  ||
op_flags[CLZ]  || op_flags[JALR]  || op_flags[MFHI] || op_flags[MFLO]  ||
op_flags[MULT]) ? instr_in[15:11] : ((
op_flags[ADDI] || op_flags[ADDIU] || op_flags[ANDI] || op_flags[ORI]   ||
op_flags[XORI] || op_flags[LW]    || op_flags[SLTI] || op_flags[SLTIU] ||
op_flags[LUI]  || op_flags[MFC0]  || op_flags[LB]   || op_flags[LH]    ||
op_flags[LBU]  || op_flags[LHU]   ) ? instr_in[20:16] : (op_flags[JAL] ? 5'd31 :
5'hz));

assign shamt =
(op_flags[SLL] || op_flags[SRL] || op_flags[SRA]) ? instr_in[10:6] : 5'hz;

assign immediate =
(op_flags[ADDI] || op_flags[ADDIU] || op_flags[ANDI]  || op_flags[ORI] ||
op_flags[XORI] || op_flags[LW]    || op_flags[SW]    || op_flags[BEQ] ||
op_flags[BNE]  || op_flags[SLTI]  || op_flags[SLTIU] || op_flags[LUI] ||
op_flags[SB]   || op_flags[SH]    || op_flags[LB]    || op_flags[LH]  ||
op_flags[LBU]  || op_flags[LHU]   || op_flags[BGEZ]) ? instr_in[15:0] : 16'hz;

assign address = (op_flags[J] || op_flags[JAL]) ? instr_in[25:0] : 26'hz;

endmodule
```

8. Controler 模块

cpu的控制器，根据当前要执行的指令输出各个元器件的状态

```
module Controler(
    input [53:0] op_flags,
    input zero_flag,
    input sign_flag,
    output reg_w,
    output [3:0] aluc,
    output dm_r,
    output dm_w,
    output HI_w,
```

```
    output LO_w,
    output [4:0] cause,
    output [4:0] ext_ena,
    output cat_ena,
    output [10:0] mux,
    output [2:0]  mux6_1,
    output [2:0]  mux6_2
    );

parameter ADD   = 6'd0;                 parameter SLTIU = 6'd27;
parameter ADDU  = 6'd1;                 parameter LUI   = 6'd28;
parameter SUB   = 6'd2;                 parameter J     = 6'd29;
parameter SUBU  = 6'd3;                 parameter JAL   = 6'd30;
parameter AND   = 6'd4;                 parameter CLZ     = 6'd31;
parameter OR    = 6'd5;                 parameter JALR    = 6'd32;
parameter XOR   = 6'd6;                 parameter MTHI    = 6'd33;
parameter NOR   = 6'd7;                 parameter MTLO    = 6'd34;
parameter SLT   = 6'd8;                 parameter MFHI    = 6'd35;
parameter SLTU  = 6'd9;                 parameter MFLO    = 6'd36;
parameter SLL   = 6'd10;                parameter SB      = 6'd37;
parameter SRL   = 6'd11;                parameter SH      = 6'd38;
parameter SRA   = 6'd12;                parameter LB      = 6'd39;
parameter SLLV  = 6'd13;                parameter LH      = 6'd40;
parameter SRLV  = 6'd14;                parameter LBU     = 6'd41;
parameter SRAV  = 6'd15;                parameter LHU     = 6'd42;
parameter JR    = 6'd16;                parameter ERET    = 6'd43;
parameter ADDI  = 6'd17;                parameter BREAK   = 6'd44;
parameter ADDIU = 6'd18;                parameter SYSCALL = 6'd45;
parameter ANDI  = 6'd19;                parameter TEQ     = 6'd46;
parameter ORI   = 6'd20;                parameter MFC0    = 6'd47;
parameter XORI  = 6'd21;                parameter MTC0    = 6'd48;
parameter LW    = 6'd22;                parameter MUL     = 6'd49;
parameter SW    = 6'd23;                parameter MULTU   = 6'd50;
parameter BEQ   = 6'd24;                parameter DIV     = 6'd51;
parameter BNE   = 6'd25;                parameter DIVU    = 6'd52;
parameter SLTI  = 6'd26;                parameter BGEZ    = 6'd53;

assign reg_w =
(!op_flags[JR]  && !op_flags[SW]   && !op_flags[BEQ]  && !op_flags[BNE]
&&!op_flags[J]  && !op_flags[MTHI] && !op_flags[MTLO] && !op_flags[SB]
&&!op_flags[SH] && !op_flags[ERET] && !op_flags[BREAK]&& !op_flags[SYSCALL]
&&!op_flags[TEQ]&& !op_flags[MTC0] && !op_flags[MUL]  && !op_flags[MULTU]
&&!op_flags[DIV]&& !op_flags[DIVU] && !op_flags[BGEZ]) ? 1'b1 : 1'b0;

assign aluc[3] =
(op_flags[SLT]  || op_flags[SLTU]  || op_flags[SLLV] || op_flags[SRLV] ||
 op_flags[SRAV] || op_flags[SLL]   || op_flags[SRL]  || op_flags[SRA]  ||
 op_flags[SLTI] || op_flags[SLTIU] || op_flags[LUI]) ? 1'b1 : 1'b0;

assign aluc[2] =
(op_flags[AND]  || op_flags[OR]    || op_flags[XOR]  || op_flags[NOR]  ||
 op_flags[SLLV] || op_flags[SRLV]  || op_flags[SRAV] || op_flags[SLL]  ||
 op_flags[SRL]  || op_flags[SRA]   || op_flags[ANDI] || op_flags[ORI]  ||
 op_flags[XORI]) ? 1'b1 : 1'b0;

assign aluc[1] =
(op_flags[ADD]  || op_flags[SUB]   || op_flags[XOR]  || op_flags[NOR]  ||
 op_flags[SLT]  || op_flags[SLTU]  || op_flags[SLLV] || op_flags[SLL]  ||
 op_flags[ADDI] || op_flags[XORI]  || op_flags[SLTI] || op_flags[SLTIU]||
 op_flags[SB]   || op_flags[SH]    || op_flags[LB]   || op_flags[LH]   ||
 op_flags[LBU]  || op_flags[LHU]   || op_flags[TEQ]  || op_flags[BGEZ])
? 1'b1 : 1'b0;

assign aluc[0] =
(op_flags[SUB]  || op_flags[SUBU]  || op_flags[OR]   || op_flags[NOR]  ||
 op_flags[SLT]  || op_flags[SLLV]  || op_flags[SRLV] || op_flags[SLL]  ||
 op_flags[SRL]  || op_flags[ORI]   || op_flags[SLTI] || op_flags[LUI]  ||
 op_flags[BEQ]  || op_flags[BNE]   || op_flags[MFC0] || op_flags[TEQ]  ||
```

```
op_flags[BGEZ]) ? 1'b1 : 1'b0;

assign dm_r = op_flags[LW] || op_flags[LB] || op_flags[LH] || op_flags[LBU] ||
op_flags[LHU] ? 1'b1 : 1'b0;
assign dm_w = op_flags[SW] || op_flags[SB] || op_flags[SH] ? 1'b1 : 1'b0;

assign HI_w = (op_flags[MTHI] || op_flags[MUL] || op_flags[MULTU] ||
op_flags[DIV] || op_flags[DIVU]) ? 1'b1 : 1'b0;
assign LO_w = (op_flags[MTLO] || op_flags[MUL] || op_flags[MULTU] ||
op_flags[DIV] || op_flags[DIVU]) ? 1'b1 : 1'b0;

assign cause = op_flags[SYSCALL] ? 5'b01000 : (op_flags[BREAK] ? 5'b01001 :
(op_flags[TEQ] ? 5'b01101 : 5'bz));

assign ext_ena[4] =
(op_flags[BEQ]  || op_flags[BNE]   || op_flags[BGEZ]) ? 1'b1 : 1'b0;
assign ext_ena[3] =
(op_flags[ADDI] || op_flags[ADDIU] || op_flags[LW]   || op_flags[SW] ||
 op_flags[SLTI] || op_flags[SLTIU] || op_flags[SB]   || op_flags[SH] ||
 op_flags[LB]   || op_flags[LH]    || op_flags[LBU]  || op_flags[LHU])
? 1'b1 : 1'b0;
assign ext_ena[2] =
(op_flags[ANDI] || op_flags[ORI]   || op_flags[XORI] || op_flags[LUI])
? 1'b1 : 1'b0;
assign ext_ena[1] =
(op_flags[SLL]  || op_flags[SRL]   || op_flags[SRA]) ? 1'b1 : 1'b0;
assign ext_ena[0] =
(op_flags[SLT]  || op_flags[SLTU]  || op_flags[SLTI] || op_flags[SLTIU])
? 1'b1 : 1'b0;

assign cat_ena = (op_flags[J] || op_flags[JAL]) ? 1'b1 : 1'b0;

assign mux[1]  = op_flags[MFC0] ? 1'b1 : 1'b0;
assign mux[2]  =
(op_flags[ANDI] || op_flags[ORI]   || op_flags[XORI] || op_flags[LUI])
? 1'b0 : 1'b1;
assign mux[3]  =
(op_flags[SLL]  || op_flags[SRL]   || op_flags[SRA]) ? 1'b1 : 1'b0;
assign mux[4]  =
(op_flags[ADDI] || op_flags[ADDIU] || op_flags[ANDI] || op_flags[ORI]  ||
 op_flags[XORI] || op_flags[LW]    || op_flags[SW]   || op_flags[SLTI] ||
 op_flags[SLTIU]|| op_flags[LUI]   || op_flags[SB]   || op_flags[SH]   ||
 op_flags[LB]   || op_flags[LH]    || op_flags[LBU]  || op_flags[LHU])
? 1'b1 : 1'b0;
assign mux[5]  = op_flags[MTHI] ? 1'b0 : 1'b1;
assign mux[6]  = op_flags[MTLO] ? 1'b0 : 1'b1;
assign mux[7]  = op_flags[MFHI] ? 1'b0 : 1'b1;
assign mux[8]  =
(op_flags[SLT]  || op_flags[SLTU]  || op_flags[SLTI] || op_flags[SLTIU])
? 1'b0 : 1'b1;
assign mux[9]  = (op_flags[MUL]  || op_flags[MULTU]) ? 1'b0 : 1'b1;
assign mux[10] = (op_flags[MUL]  || op_flags[MULTU]) ? 1'b0 : 1'b1;

assign mux6_1  =
(op_flags[JR]   || op_flags[JALR])            ? 3'b000 :
((op_flags[BEQ] && zero_flag) || (op_flags[BNE]&&!zero_flag)||
(op_flags[BGEZ] && !sign_flag)          ? 3'b010 :
(op_flags[BREAK]|| op_flags[SYSCALL] || (op_flags[TEQ] && zero_flag)   3'b011 :
(op_flags[ERET]                         ? 3'b100 :
(op_flags[J]    || op_flags[JAL]        ? 3'b101 : 3'b001))));

assign mux6_2  =( op_flags[LW] ||op_flags[LB]||
op_flags[LH]||op_flags[LBU]||op_flags[LHU]) ? 3'b000 :
(op_flags[MFHI]  || op_flags[MFLO]          ? 3'b001 :
(op_flags[CLZ]                              ? 3'b010 :
(op_flags[JALR]  || op_flags[JAL]           ? 3'b101 :
(op_flags[MFC0]                             ? 3'b110 : 3'b011))));
Endmodule
```

9. PC 模块

Pc寄存器，按时钟信号传递指令。

```
module PC(
    input  pc_clk,
    input  pc_ena,
    input  rst,
    input  [31:0] pc_addr_in,
    output [31:0] pc_addr_out
    );
reg [31:0] pc_reg = 32'h00400000;
assign pc_addr_out = pc_ena ? pc_reg :
32'hz;

always @(negedge pc_clk or posedge
rst)
begin
    if(rst && pc_ena)
pc_reg = 32'h00400000;
    else if(pc_ena)
        pc_reg = pc_addr_in;
end
```

10. **HI_LO寄存器模块**

HI_LO寄存器，用来存储MUL、DIV的运算结果

```
module HI_LO(
    input HI_LO_clk,         //时钟信号
    input HI_LO_ena,         //使能信号
    input HI_LO_rst,         //复位信号
    input [31:0] HI_in,      //向HI输入的数
    input [31:0] LO_in,      //向LO输入的数
    input HI_w,              //使能信号，是否是向HI中写入数
    input LO_w,              //使能信号，是否是向LO中写入数
    output [31:0] HI_out,    //从HI传出的数
    output [31:0] LO_out     //从LO传出的数
    );

reg [31:0] HI = 32'd0;       //存储高32位数
reg [31:0] LO = 32'd0;       //存储低32位数

assign HI_out = HI_LO_ena ? HI : 32'bz;
assign LO_out = HI_LO_ena ? LO : 32'bz;

always @(posedge HI_LO_rst or negedge HI_LO_clk) begin
    if (HI_LO_ena && HI_LO_rst) begin
        HI = 32'd0;
        LO = 32'd0;
    end
    else if(HI_LO_ena)
    begin
        if(HI_w)
            HI = HI_in;
        if(LO_w)
            LO = LO_in;
    end
end

endmodule
```

11. MUL 模块

乘法器，用来执行32位有符号或无符号乘法，结果写入HI_LO寄存器

```
module MUL(
    input sign_flag,
    input [31:0] A,
    input [31:0] B,
    output [31:0] HI,
    output [31:0] LO
    );
wire [31:0] aa=A[31]?(A^
32'hFFFFFFFF+1):A;
wire [31:0] bb=B[31]?(B^
32'hFFFFFFFF+1):B;

wire [31:0] a=sign_flag?aa:A;
wire [31:0] b=sign_flag?bb:B;

wire [63:0] result;

assign store0=b[0]?{32'b0,a}:64'b0;
assign store1=b[1]?{31'b0,a,1'b0}:64'b0;
assign store2=b[2]?{30'b0,a,2'b0}:64'b0;
assign store3=b[3]?{29'b0,a,3'b0}:64'b0;
assign store4=b[4]?{28'b0,a,4'b0}:64'b0;
assign store5=b[5]?{27'b0,a,5'b0}:64'b0;
assign store6=b[6]?{26'b0,a,6'b0}:64'b0;
```

```
assign store7=b[7]?{25'b0,a,7'b0}:64'b0;
assign store8=b[8]?{24'b0,a,8'b0}:64'b0;
assign store9=b[9]?{23'b0,a,9'b0}:64'b0;
assign store10=b[10]?{22'b0,a,10'b0}:64'b0;
assign store11=b[11]?{21'b0,a,11'b0}:64'b0;
assign store12=b[12]?{20'b0,a,12'b0}:64'b0;
assign store13=b[13]?{19'b0,a,13'b0}:64'b0;
assign store14=b[14]?{18'b0,a,14'b0}:64'b0;
assign store15=b[15]?{17'b0,a,15'b0}:64'b0;
assign store16=b[16]?{16'b0,a,16'b0}:64'b0;
assign store17=b[17]?{15'b0,a,17'b0}:64'b0;
assign store18=b[18]?{14'b0,a,18'b0}:64'b0;
assign store19=b[19]?{13'b0,a,19'b0}:64'b0;
assign store20=b[20]?{12'b0,a,20'b0}:64'b0;
assign store21=b[21]?{11'b0,a,21'b0}:64'b0;
assign store22=b[22]?{10'b0,a,22'b0}:64'b0;
assign store23=b[23]?{9'b0,a,23'b0}:64'b0;
assign store24=b[24]?{8'b0,a,24'b0}:64'b0;
assign store25=b[25]?{7'b0,a,25'b0}:64'b0;
assign store26=b[26]?{6'b0,a,26'b0}:64'b0;
assign store27=b[27]?{5'b0,a,27'b0}:64'b0;
assign store28=b[28]?{4'b0,a,28'b0}:64'b0;
assign store29=b[29]?{3'b0,a,29'b0}:64'b0;
assign store30=b[30]?{2'b0,a,30'b0}:64'b0;
assign store31=b[31]?{1'b0,a,31'b0}:64'b0;

assign store0_1=store0+store1;
assign store2_3=store2+store3;
assign store4_5=store4+store5;
assign store6_7=store6+store7;
assign store8_9=store8+store9;
assign store10_11=store10+store11;
assign store12_13=store12+store13;
assign store14_15=store14+store15;
assign store16_17=store16+store17;
assign store18_19=store18+store19;
assign store20_21=store20+store21;
assign store22_23=store22+store23;
assign store24_25=store24+store25;
assign store26_27=store26+store27;
assign store28_29=store28+store29;
assign store30_31=store30+store31;
assign store0_3=store0_1+store2_3;
assign store4_7=store4_5+store6_7;
assign store8_11=store8_9+store10_11;
assign store12_15=store12_13+store14_15;
assign store16_19=store16_17+store18_19;
assign store20_23=store20_21+store22_23;
assign store24_27=store24_25+store26_27;
assign store28_31=store28_29+store30_31;

assign store0_7=store0_3+store4_7;
assign store8_15=store8_11+store12_15;
assign store16_23=store16_19+store20_23;
assign store24_31=store24_27+store28_31;

assign store0_15=store0_7+store8_15;
assign store16_31=store16_23+store24_31;

assign result=store0_15+store16_31;
assign {HI,LO}=sign_flag&&(A[31]!=B[31])
?(result^ 32'hFFFFFFFF+1):result;
endmodule
```

12. DIV 模块

除法器，用来执行32位有符号或无符号除法，结果写入HI_LO寄存器

```
module DIV(
    input sign_flag,    //是否是有符号除法
    input [31:0] A,     //输入的被除数
    input [31:0] B,     //输入的除数
    output reg [31:0] R,    //余数
    output reg [31:0] Q     //商
    );

    always @(*) begin
    if(sign_flag)
    {R, Q} <= {$signed(A) % $signed(B), $signed(A) / $signed(B)};//DIV
    else
    {R, Q} <= {A % B, A / B};//DIVU
    end

endmodule
```

13. MUX 模块

结构上是8选1多路选择器，实际只用了六根线，在cpu中使用了两个MUX

```
module MUX(
    input [2:0] chosen,     //8选1，用三位
    input [31:0] line0,
    input [31:0] line1,
    input [31:0] line2,
    input [31:0] line3,
    input [31:0] line4,
    input [31:0] line5,
    input [31:0] line6,
    input [31:0] line7,
    output [31:0] MUX_out   //选择的线
    );
```

```
assign MUX_out =
(chosen == 3'b000 ? line0 : (chosen == 3'b001 ? line1 :
(chosen == 3'b010 ? line2 : (chosen == 3'b011 ? line3 :
(chosen == 3'b100 ? line4 : (chosen == 3'b101 ? line5 :
(chosen == 3'b110 ? line6 :  (chosen == 3'b111 ? line7 : 32'bz)))))));

endmodule
```

14. CLZ 模块

前导零计数器，用于计算32位数据前导零的个数

```
module CLZ(
    input [31:0] CLZ_in,     //要计算前导0的数值
    output [31:0] CLZ_out    //输出前导0的个数
    );

reg [31:0] cnt = 32'd0;      //前导0的数量
always @(*) begin
    if (CLZ_in[31] == 1'b1)     cnt<=0;
    else if(CLZ_in[30] == 1'b1)cnt<=1;
    else if(CLZ_in[29] == 1'b1)cnt<=2;
    else if(CLZ_in[28] == 1'b1)cnt<=3;
    else if(CLZ_in[27] == 1'b1)cnt<=4;
    else if(CLZ_in[26] == 1'b1)cnt<=5;
    else if(CLZ_in[25] == 1'b1)cnt<=6;
    else if(CLZ_in[24] == 1'b1)cnt<=7;
    else if(CLZ_in[23] == 1'b1)cnt<=8;
    else if(CLZ_in[22] == 1'b1)cnt<=9;
    else if(CLZ_in[21] == 1'b1)cnt<=10;
    else if(CLZ_in[20] == 1'b1)cnt<=11;
    else if(CLZ_in[19] == 1'b1)cnt<=12;
    else if(CLZ_in[18] == 1'b1)cnt<=13;
    else if(CLZ_in[17] == 1'b1)cnt<=14;
    else if(CLZ_in[16] == 1'b1)cnt<=15;
    else if(CLZ_in[15] == 1'b1)cnt<=16;
    else if(CLZ_in[14] == 1'b1)cnt<=17;
    else if(CLZ_in[13] == 1'b1)cnt<=18;
    else if(CLZ_in[12] == 1'b1)cnt<=19;
    else if(CLZ_in[11] == 1'b1)cnt<=20;
    else if(CLZ_in[10] == 1'b1)cnt<=21;
    else if(CLZ_in[ 9] == 1'b1)cnt<=22;
    else if(CLZ_in[ 8] == 1'b1)cnt<=23;
    else if(CLZ_in[ 7] == 1'b1)cnt<=24;
    else if(CLZ_in[ 6] == 1'b1)cnt<=25;
    else if(CLZ_in[ 5] == 1'b1)cnt<=26;
    else if(CLZ_in[ 4] == 1'b1)cnt<=27;
    else if(CLZ_in[ 3] == 1'b1)cnt<=28;
    else if(CLZ_in[ 2] == 1'b1)cnt<=29;
    else if(CLZ_in[ 1] == 1'b1)cnt<=30;
    else if(CLZ_in[ 0] == 1'b1)cnt<=31;
    else if(CLZ_in == 0)       cnt<=32;
end

assign CLZ_out = cnt;

endmodule
```

15. CP0 模块

用于处理中断指令，主要有读写功能、对异常发生时的跳转功能和异常返回功能，异常的入口地址为0x4，在MIPScpu中还要加上偏移地址32'h00400000

```
module CP0(
    input cp0_clk,
    input cp0_rst,
    input cp0_ena,
    input MFC0,
    input MTC0,
```

```
    input ERET,
    input [31:0] PC,
    input [31:0] addr,
    input [4:0] cause,
    input [31:0] data_in,
    output [31:0] CP0_out,
    output [31:0] EPC_out
    );

parameter SYSCALL = 5'b01000,  BREAK   = 5'b01001,  TEQ     = 5'b01101;
parameter STATUS = 4'd12,      CAUSE  = 4'd13,      EPC    = 4'd14;
reg [31:0] cp0_reg [31:0];

assign EPC_out = ERET && cp0_ena? cp0_reg [EPC] : 32'hz;
assign CP0_out = MFC0 && cp0_ena? cp0_reg [addr[4:0]] : 32'hz;

always @(negedge cp0_clk or posedge cp0_rst)
begin
    if(cp0_rst && cp0_ena)
    begin
        cp0_reg [0] <=0 ;                          cp0_reg [16] <=0 ;
        cp0_reg [1] <=0 ;                          cp0_reg [17] <=0 ;
        cp0_reg [2] <=0 ;                          cp0_reg [18] <=0 ;
        cp0_reg [3] <=0 ;                          cp0_reg [19] <=0 ;
        cp0_reg [4] <=0 ;                          cp0_reg [20] <=0 ;
        cp0_reg [5] <=0 ;                          cp0_reg [21] <=0 ;
        cp0_reg [6] <=0 ;                          cp0_reg [22] <=0 ;
        cp0_reg [7] <=0 ;                          cp0_reg [23] <=0 ;
        cp0_reg [8] <=0 ;                          cp0_reg [24] <=0 ;
        cp0_reg [9] <=0 ;                          cp0_reg [25] <=0 ;
        cp0_reg [10] <=0 ;                         cp0_reg [26] <=0 ;
        cp0_reg [11] <=0 ;                         cp0_reg [27] <=0 ;
        cp0_reg [12] <=0 ;                         cp0_reg [28] <=0 ;
        cp0_reg [13] <=0 ;                         cp0_reg [29] <=0 ;
        cp0_reg [14] <=0 ;                         cp0_reg [30] <=0 ;
        cp0_reg [15] <=0 ;                         cp0_reg [31] <=0 ;

    end
    else if(cp0_ena)
    begin
        if(MTC0)
            cp0_reg [addr[4:0]] <= data_in;
        else if (cause == SYSCALL || cause == BREAK || cause == TEQ)
        begin
            cp0_reg [STATUS] <= {cp0_reg [STATUS][26:0],5'd0}; //左移5位关中断
            cp0_reg [CAUSE]  <= {24'b0 , cause , 2'b0};
            cp0_reg [EPC]    <= PC;
        end
        else if(ERET)
            cp0_reg [STATUS] <= {5'd0,cp0_reg [STATUS][31:5]};//右移5位开中断
    end
end
endmodule
```

16. 测试模块

```
module cpu_tb;
reg clk;
reg rst;
wire [31:0] inst;
wire [31:0] pc;
initial
begin
    clk = 1'b0;
    rst = 1'b1;
    #50 rst = 1'b0;
end
always  #50 clk = ~clk;
sccomp_dataflow _inst(
    .clk_in(clk),
    .reset(rst),
    .inst(inst),
    .pc(pc)
);
endmodule
```

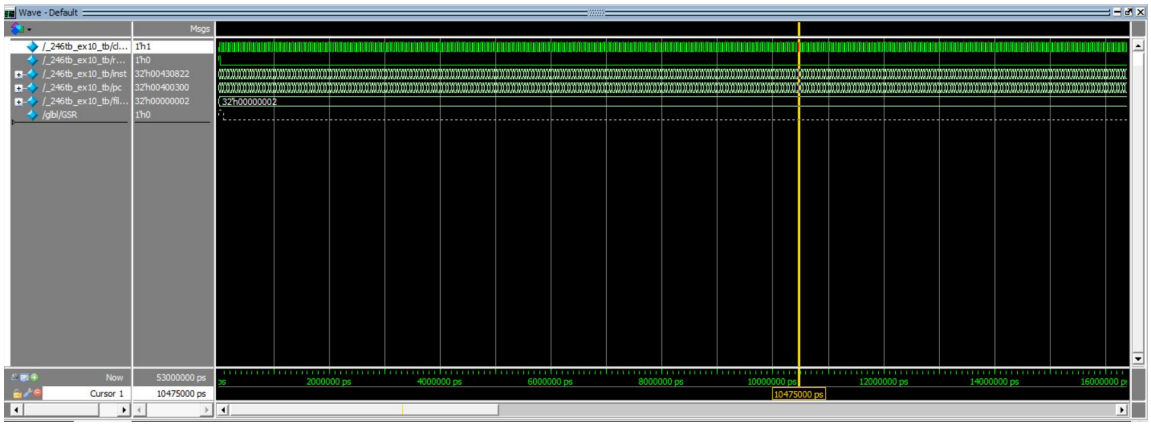
17. Divider 模块

```
module  Divider(
    input I_CLK,
    output reg  O_CLK_1M=0
    );
      parameter size=32'd50000000;
    integer i = 0;
    always@(posedge I_CLK) begin
    i=i+1;
    if(i >= size) begin
    i=0;
    O_CLK_1M=!O_CLK_1M;
    end
      end
endmodule
```

四、测试结果

1. 综合指令测试

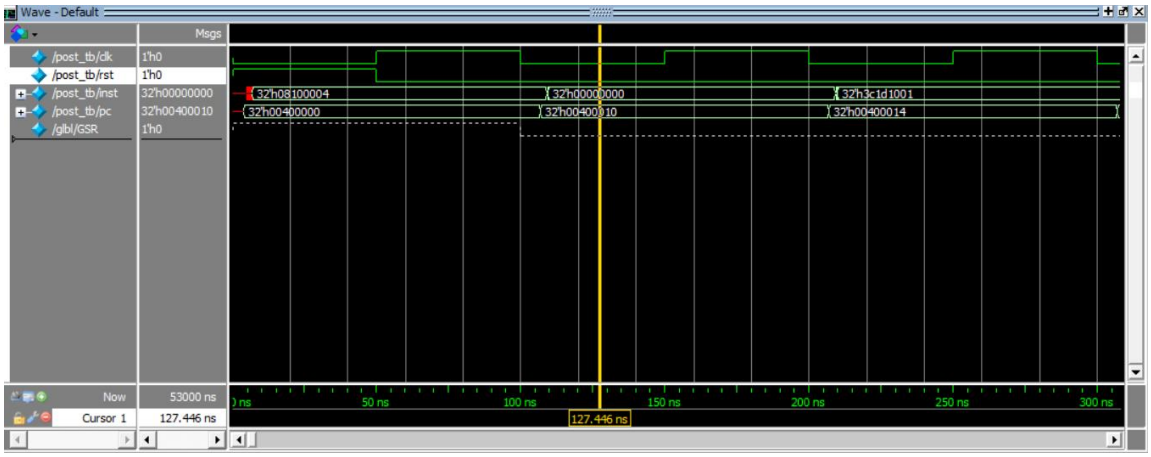
modelsim 仿真波形图：（前仿真）



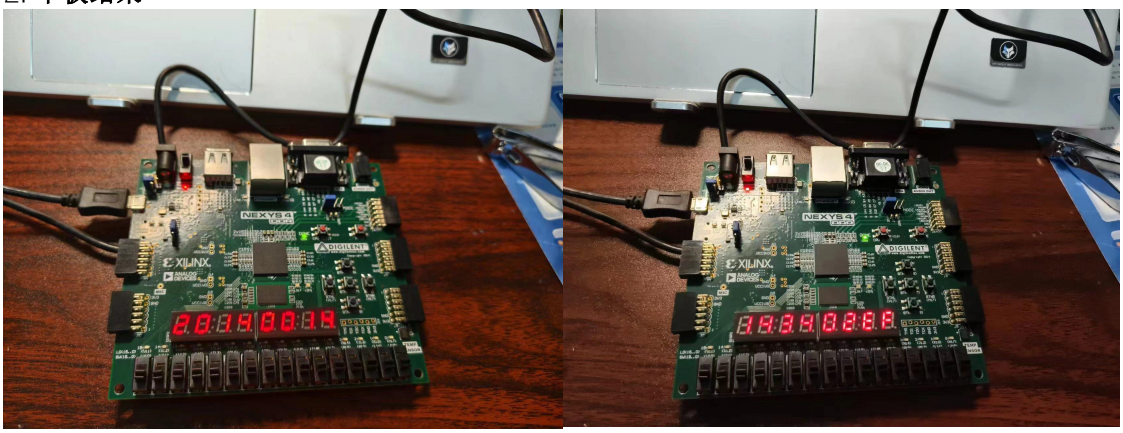
用之前课程老师提供的txt文件比较程序比较了输出结果与标准答案，如下图可知输出正确。

```
C:\Users\86138\Desktop>txt_compare --file1 rst0.txt --file2 rst1.txt --display normal
比较结果输出：
========================================
第[35837 / 35837]行 - 文件2的尾部有多余字符：
文件1 : <EOF>
文件2 : pc: 004013f4<CR>

========================================
在指定检查条件下共1行有差异.
阅读提示：
        1、空行用<EMPTY>标出
        2、文件结束用<EOF>标出
        3、两行相同列位置的差异字符用亮色标出
        4、每行中的CR/LF/VT/BS/BEL用X标出(方便看清隐含字符)
        5、每行尾的多余的字符用亮色标出，CR/LF/VT/BS/BEL用亮色X标出(方便看清隐含字符)
        6、每行最后用<CR>标出(方便看清隐含字符)
        7、中文因为编码问题，差异位置可能报在后半个汉字上，但整个汉字都亮色标出
        8、用--display detailed可以得到更详细的信息
========================================
```

modelsim 仿真波形图：（后仿真）



2. 下板结果



五、心得体会及建议

1. 心得体会

通过本次实验，我亲身参与了 CPU 结构的设计、数据通路图的绘制以及使用 Verilog 语言实现了支持 54 条指令的 CPU。这一过程让我获得了许多宝贵的经验和体会。

在设计阶段，需要全面考虑数据通路、指令结构以及指令执行的各种情况。一个良好的设计能够为后续的实现工作奠定坚实的基础。相比之下，实现阶段只需将设计转化为代码描述，如果设计不清晰，实现过程很可能会陷入混乱，最终导致 CPU 无法正常工作。因此，设计阶段的重要性不容忽视。

本次实验让我对计算机组成原理课程中学到的知识有了更深层次的理解与应用。通过将课本上的理论知识转化为实践中的设计与实现，我对 CPU 的工作原理有了更加清晰的认识。实践使我能够更好地理解课堂上学到的抽象概念，并将其应用于实际工程中。