



哈尔滨工业大学 (深圳)
HARBIN INSTITUTE OF TECHNOLOGY

实验报告

开课学期: 2020 夏季

课程名称: 计算机设计与实践

实验名称: 多周期 CPU 设计

实验性质: 综合设计型

实验学时: 42 地点: 线上

学生班级: 5

学生学号: 18011511

学生姓名: 王家睿

评阅教师:

报告成绩:

实验与创新实践教育中心制

2020 年 5 月

注：

本设计报告中各个部分如果页数不够，请大家自行扩页，原则是一定要把报告写详细，能说明设计的成果和特色。报告中应该叙述设计中的每个模块。设计报告将是评定每个人成绩的重要组成部分（**设计内容及报告写作**都作为评分依据）。

完成单周期 **CPU** 设计与实现的同学根据单周期设计的内容完成实验报告。完成多周期 **CPU** 设计与实现的同学根据多周期设计的内容完成实验报告。

设计的功能描述（含所有实现的指令描述及模块的功能）

```

v ● minisys (minisys.v) (11)
  > ● u_clk : CLK (CLK.v) (1)
    ● u_pc : PC (PC.v)
  > ● u_npc : NPC (NPC.v) (1)
  > ● u_rom : programrom (programrom.v) (1)
    ● u_ext : S_EXT (S_EXT.v)
    ● u_rf : RF (RF.v)
  > ● u_ram : dmemory32 (dmemory32.v) (1)
    ● u_alu : ALU (ALU.v)
    ● u_cu : CU (CU.v)
    ● u_AR : AR_reg (AR_reg.v)
    ● u_IR : IR_reg (IR_reg.v)

```

整个设计的模块结构：

Minisys 是最上层模块：负责各个模块的实例化和数据通路的连线

外部的时钟信号和复位信号也从 minisys 中输入

Clk 是分频模块，调整 100MHz 时钟频率的大小

PC 传出下一条指令的地址

NPC 计算下一条 PC 值送到 PC 中

Programrom 是指令存储器

S_EXT 是拓展单元，将数据拓展到 32 位

RF 是寄存器堆，用于保存 CPU 执行过程中的各种数据，并完成指令的译码

Dmemory32 是数据存储器

ALU 是运算单元，完成 CPU 内部的逻辑运算

CU 是控制单元，发出控制信号

AR_reg 保存 ALU 的运算结果

IR_reg 保存 Programrom 中传出的指令

R 型指令：

以 add 为例：

假设 PC 传给 Programrom 的地址取出的指令为 add

```

programrom u_rom(
    .PC(debug_wb_pc),
    .clock(clock),
    .Instruction(Instruction)
);

S_EXT u_ext(
    .IMM(ins_out[15:0]),
    .EXTOp(EXTOp),
    .clock(~clock),
    .IMM_EX(imm_ex)
);

RF u_rf(
    .RD1(RF_RD1),
    .RD2(RF_RD2),
    .A3(debug_wb_rf_wnum),
    .Wdata(debug_wb_rf_wdata),
    .op(op),
    .func(func),
    .instruction(ins_out),

```

指令会被送入 RF 中，输出 op 和 func，即指令的高 6 位和低 6 位 RD1 和 RD2 会被送入 ALU 中，在 ALU 中进行相应的计算

```

CU u_cu(
    .reset(rst),
    .clock(clock),
    .op(op),
    .func(func),
    .Zero(Zero),
    .NPCOp(NPCOp),
    .PCWr(PCWr),
    .IRWr(IRWr),

```

Op 和 func 会被送入 CU 控制单元中（不直接向控制单元传送指令是为了确保控制信号的发出比译码操作晚）

```

        .ALUOp(ALUOp),
        .DMWr(DMWr),
        .WRSel(WRSel),
        .WDSel(WDSel),
        .BSel(BSel),
        .ZeroG(ZeroG),
        .EXTOp(EXTOp),
        .backflag(backsignal)
    ).

```

CU 中有状态机，在不同周期传出不同控制信号给各个部件进行相应的操作在 ALU 中完成相应的逻辑操作，现在已在 ALU 完成的操作有

```

always@(negedge clock)
begin
    case(ALUOp)
        `ADD:
        begin
            result = ALU_A + ALU_B ;
        end

        `SUB:
        begin
            result = ALU_A - ALU_B ;
        end

        `SLTI:
        begin
            result = ((($signed(ALU_A - ALU_B)) < 0)? 32'b01: 32'b00) ;
        end

        `BGTZ:
        begin
            result = ( ($signed(ALU_A)) > 0 ) ? 32'b01: 32'b0;
        end

        `AND:
        begin
            result = ALU_A & ALU_B;
        end

        `OR:
        begin
            result = ALU_A | ALU_B;
        end
    end
end

```

```
`XOR:
begin
    result = ALU_A ^ ALU_B;
end

`NOR:
begin
    result = ~(ALU_A | ALU_B);
end

`SLL:
begin
    result = ALU_B << ALU_C;
end

`SRL:
begin
    result = ALU_B >> ALU_C;
end

`SRA:
begin
    result = ($signed(ALU_B)) >>> ALU_C;
end

`SLLV:
begin
    result = ALU_B << ALU_A;
end

`SRLV:
begin
    result = ALU_B >> ALU_A;
end
```

```

`SRAV:
begin
    result = ($signed(ALU_B)) >>> ALU_A;
end
`X:
begin

end
`LUI:
begin
    result = ( IMM_EX<<16 ) & 32'b11111111111111110000000000000000;
end
`J:
begin

end
`JAL:
begin

end
endcase

```

(lw、sw 是进行 add 操作，beq, bne 是进行 sub 操作，`X 操作是 Jr 指令)

在 ALU 的操作完成后，结果送入 AR 寄存器保存，在送入到要保存的部件中

```

AR_reg u_AR(
    .clock(~clock),
    .ALU_C(ALU_C),
    .AR(AR)
);

```

R 型指令(除了 Jr)都是在 4 个时钟周期完成整个指令，Jr 由于不需要 ALU 的运算，只需要 2 个时钟周期即可完成结果，具体的信号控制可看报告后面的控制信号取值表。

对于 I 型指令

送入 ALU 中会多一条 imm_ex 的拓展数据，ALU 会根据 Bsel 控制 RF_RD2 和 imm_ex 哪一个作为第二个参与 ALU 运算的数据，从而区别 I 型指令和 R 型指令

```

ALU u_alu(
    .ALUOp(ALUOp),
    .IMM_EX(imm_ex),
    .RF_RD2(RF_RD2),
    .RF_RD1(RF_RD1),
    .IM_D(ins_out[10:6]),
    .BSel(BSel),
    .ZeroG(ZeroG),
    .clock(~clock),
    .Zero(Zero),
    .result(ALU_C)
);

```

I 型指令会在第二周期多出一个穿给 S_EXT 拓展单元的控制信号，除 beq 等跳转指令和 lw 指令，I 型指令也都是在 4 个周期就完成了整条指令，lw 由于需要从存储器读出数据，需要 5 个周期来完成，而 beq 等指令不需要写回 RF 中，只需要 3 个周期即可完成

最后两条 J 型指令，都只需要 2 个周期即可完成

第一周期取指，J 在第二周期由 NPC 控制直接跳转到对应地址

Jal 不仅在第二周期由 NPC 控制跳转到对应地址，还需要当前地址+4 存储到 RF 中。

设计的主要特色

① 省略了对 NPC 传入 Zero 信号

```

assign BEQ_PC = (PC4 + {IMM_EX[29:0], 2'b00});
assign NPCtoPC_reg = (NPCOP==2'b10) ? RA : ( (NPCOP==2'b11) ? {PC[31:28], IMM, 2'b00} : ((NPCOP == 2'b01) ? (PC4 + {14'b0, IMM[15:0], 2'b00}) : (PC4 + {14'b0, IMM[15:0], 2'b00}));
always@(posedge clock)
begin
    if(!reset && PC_flag == 0)
        if(!reset)
        begin
            PC_flag = 1;
        end
    else if(reset)
        NPCtoPC = 32'b0;
    else if(PC_Update)
    begin
        NPCtoPC = NPCtoPC_reg;
    end
    else
    begin
        end
end

```

下一条 PC 的取指完全由 NPCOP 决定，我将 ALU 计算结果的 Zero 标志传入了 CU 中，由 CU 中的控制信号结合指令和 Zero 的结果，传出正确的 NPCOP

例：

对于 beq 指令

```

if( Zero == 1)
begin
    NPCOp <= 2'b01;
end
else
begin
    NPCOp <= `PC4;
end

```

若标志位为 0，则相等，发生跳转，否则仍执行 PC+4

对于 bne 指令，则刚好相反

```

if( Zero == 0)
begin
    NPCOp <= 2'b01;
end
else
begin
    NPCOp <= `PC4;
end

```

② ALU 的赋值采用时钟控制，总是早于运算半个周期

```

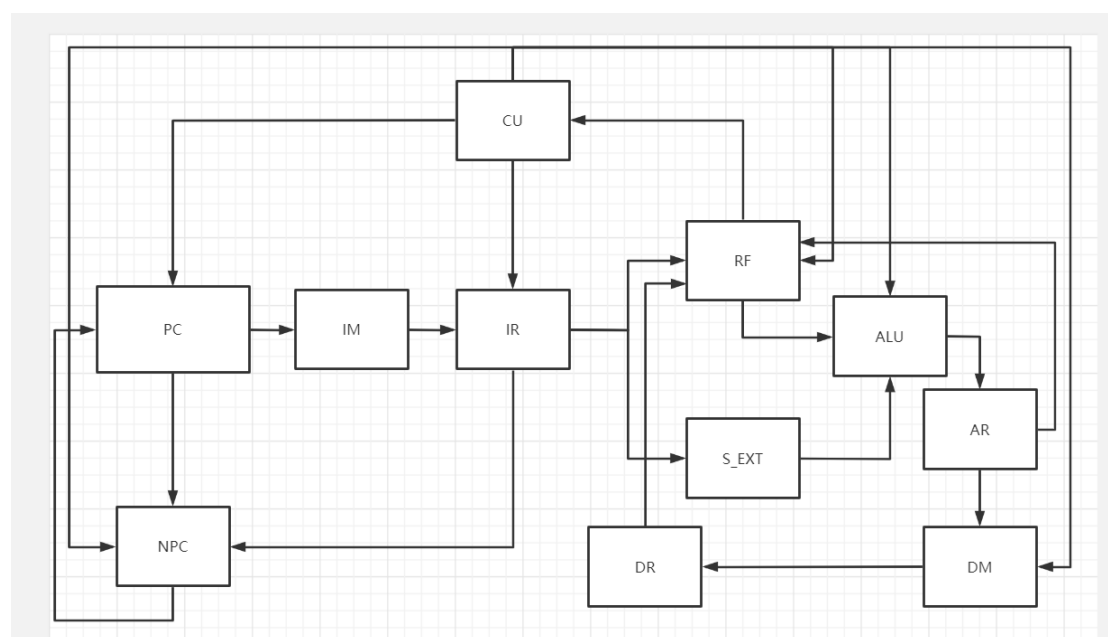
12 :
13 : always@(posedge clock)
14 : begin
15 :     ALU_A <= RF_RD1;
16 :     ALU_B <= BSel? IMM_EX:RF_RD2;
17 : end
18 : //      end
19 : always@(negedge clock)
20 : begin
21 :     case(ALUOp)
22 :     `ADD:
23 :     begin
24 :         result = ALU_A + ALU_B ;
25 :     end

```

若和单周期设计一样使用组合逻辑，ALU_A 和 ALU_B 的变化有时会在一个周期内发生变化，为保证运算的顺利进行，我保持参与运算的两个数据一个周期不变化，从而顺利的执行运算。

设计的体系结构

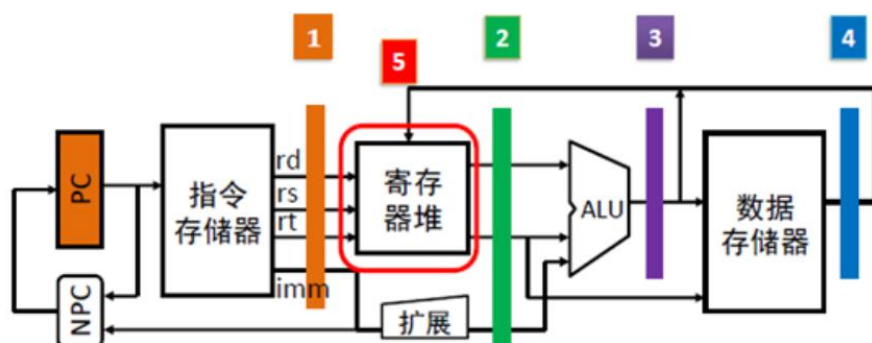
(包括体系结构框图和对结构图的简要解释)



PC 将指令的地址传入 IM 和 NPC 中，IM 根据地址找到对应的指令，将其传入 IR 寄存器中，之后所有部件需要的指令都来自于 IR 寄存器中，RF 将 IR 的指令进行译码，传给 CU，CU 受到译码信息，在每个时钟周期都发出对应的控制信号，NPC 根据 CU 给出的控制信号计算出下一个 PC 值，在 PCWr=1 的情况下就可将该值传给 PC，ALU 接受了 RF、S_EXT 的数据和 CU 的控制信号，进行相应的运算操作，运算结果送入 AR 中，运算结果可以写回 RF 中，也可作为地址进入 DM 中，取出存储器中的数据，存储器的数据进入 DR，再时钟信号的作用下写回 RF。一个指令执行完，CU 会发出进入下一个指令周期的信号，PC 获取 NPC 的值后重新开始取指周期，以此往复。

数据通路设计

(包含数据通路主要部件说明及数据通路表)



PC: 储存下一条指令的地址, 通过改变 PC 来决定下一条指令所要执行的操作

NPC: 接受从 PC 传出的地址值, 通过控制信号计算出下一条指令的地址, 送入 PC 中

IM: 指令存储器, 存储要执行的指令, 接受 PC 传来的地址, 取出指令送到指令寄存器 IR 中

IR: 指令寄存器, 在控制信号 IRWr 的作用下, 接受 IM 取出的指令值, 并将指令传送到中译码模块中进行指令的译码

RF: 寄存器堆, 保存各种指令操作所需要的数据, 可根据指令读出对应寄存器中的数据

S_EXT: 拓展单元, 将输入的 16 位数据拓展成 32 位, 可进行无符号拓展和带符号拓展

A、B、E: 中间寄存器, 在送入 ALU 之前, 暂时保存从 RF 取出的数据和 S_EXT 拓展后的数据

ALU: 运算器, 执行加法、移位等运算

ALU_out: 暂时存储 ALU 的计算结果

DM: 数据存储器, 即内存

DR: 数据寄存器, 暂时保存 DM 中取出的数据

	所属单元	取指单元					
	部件	PC	NPC			IM (指令存储器)	IR
	输入信号	DI	PC	Imm	RA (寄存器)	A	
R型指令	add	NPC. NPC	PC. DO			PC. DO	IM
	addu	NPC. NPC	PC. DO			PC. DO	IM
	sub	NPC. NPC	PC. DO			PC. DO	IM
	subu	NPC. NPC	PC. DO			PC. DO	IM
	and	NPC. NPC	PC. DO			PC. DO	IM
	or	NPC. NPC	PC. DO			PC. DO	IM
	xor	NPC. NPC	PC. DO			PC. DO	IM
	nor	NPC. NPC	PC. DO			PC. DO	IM
	slt	NPC. NPC	PC. DO			PC. DO	IM
	sltu	NPC. NPC	PC. DO			PC. DO	IM
	sll	NPC. NPC	PC. DO			PC. DO	IM
	srl	NPC. NPC	PC. DO			PC. DO	IM
	sra	NPC. NPC	PC. DO			PC. DO	IM
	sllv	NPC. NPC	PC. DO			PC. DO	IM
	srlv	NPC. NPC	PC. DO			PC. DO	IM
	srav	NPC. NPC	PC. DO			PC. DO	IM
I型指令	jr	NPC. NPC	PC. DO		A	PC. DO	IM
	addi	NPC. NPC	PC. DO			PC. DO	IM
	addiu	NPC. NPC	PC. DO			PC. DO	IM
	andi	NPC. NPC	PC. DO			PC. DO	IM
	ori	NPC. NPC	PC. DO			PC. DO	IM
	xori	NPC. NPC	PC. DO			PC. DO	IM
	sltiu	NPC. NPC	PC. DO			PC. DO	IM
	lui	NPC. NPC	PC. DO			PC. DO	IM
	lw	NPC. NPC	PC. DO			PC. DO	IM
	sw	NPC. NPC	PC. DO			PC. DO	IM
	beq	NPC. NPC	PC. DO	IR[15:0]		PC. DO	IM
	bne	NPC. NPC	PC. DO	IR[15:0]		PC. DO	IM
	bgtz	NPC. NPC	PC. DO	IR[15:0]		PC. DO	IM
J型指令	j	NPC. NPC	PC. DO	IR[25:0]		PC. DO	IM
	jal	NPC. NPC	PC. DO	IR[25:0]		PC. DO	IM
综合		NPC. NPC	PC. DO	IR[25:0]	A	PC. DO	IM

译码单元								执行单元			数据存储器		
RF（寄存器堆）				A	B	S_EXT	E	ALU		ALUOut	DM（数据存储器）	DR	
A1（读出1）	A2（读出2）	A3（写入）	WD（写回值）					A	B		A		
IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF, RD1	RF, RD2			A	B	ALU, C			
IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF, RD1	RF, RD2			A	B	ALU, C			
IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF, RD1	RF, RD2			A	B	ALU, C			
IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF, RD1	RF, RD2			A	B	ALU, C			
IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF, RD1	RF, RD2			A	B	ALU, C			
IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF, RD1	RF, RD2			A	B	ALU, C			
IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF, RD1	RF, RD2			A	B	ALU, C			
IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF, RD1	RF, RD2			A	B	ALU, C			
	IR[20:16]	IR[15:11]	ALUOut		RF, RD2				B	ALU, C			
	IR[20:16]	IR[15:11]	ALUOut		RF, RD2				B	ALU, C			
	IR[20:16]	IR[15:11]	ALUOut		RF, RD2				B	ALU, C			
IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF, RD1	RF, RD2			A	B	ALU, C			
IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF, RD1	RF, RD2			A	B	ALU, C			
IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF, RD1	RF, RD2			A	B	ALU, C			
IR[25:21]													
IR[25:21]		IR[20:16]	ALUOut	RF, RD1		IR[15:0]	EXT, Ext	A	E	ALU, C			
IR[25:21]		IR[20:16]	ALUOut	RF, RD1		IR[15:0]	EXT, Ext	A	E	ALU, C			
IR[25:21]		IR[20:16]	ALUOut	RF, RD1		IR[15:0]	EXT, Ext	A	E	ALU, C			
IR[25:21]		IR[20:16]	ALUOut	RF, RD1		IR[15:0]	EXT, Ext	A	E	ALU, C			
IR[25:21]		IR[20:16]	ALUOut	RF, RD1		IR[15:0]	EXT, Ext	A	E	ALU, C			
		IR[20:16]	ALUOut			IR[15:0]	EXT, Ext		E	ALU, C			
IR[25:21]		IR[20:16]	DR	RF, RD1		IR[15:0]	EXT, Ext	A	E	ALU, C	ALUOut	DM, RD	
IR[25:21]				RF, RD1		IR[15:0]	EXT, Ext	A	E	ALU, C	ALUOut		
IR[25:21]	IR[20:16]			RF, RD1	RF, RD2			A	B				
IR[25:21]	IR[20:16]			RF, RD1	RF, RD2			A	B				
IR[25:21]				RF, RD1				A					
		0x1F	PC										
IR[25:21]	IR[20:16]	IM, D[15:11]	ALUOut	RF, RD1	RF, RD2	IR[15:0]	EXT, Ext	A	B	ALU, C	ALUOut	DM, RD	
		IM, D[20:16]	DR										
		0x1F	PC						E				

各部件的设计与实现

(含模块功能, 输入、输出信号及变量定义, 关键代码等。控制器设计包含控制信号表) 控制信号表:

控制信号取值矩阵 (填写)										
指令	NPCOp	PCWr	IRWr	RFWr	EXTOp	ALUOp	DMWr	MRFA3Sel	MRFWDSe1	MALUBSe1
add	T1:+4	T1:1	T1:1	T4:1		T3:ADD		T4:RD	T4:AR	T3:B
addu	T1:+4	T1:1	T1:1	T4:1		T3:ADD		T4:RD	T4:AR	T3:B
sub	T1:+4	T1:1	T1:1	T4:1		T3:SUB		T4:RD	T4:AR	T3:B
subu	T1:+4	T1:1	T1:1	T4:1		T3:SUB		T4:RD	T4:AR	T3:B
and	T1:+4	T1:1	T1:1	T4:1		T3:AND		T4:RD	T4:AR	T3:B
or	T1:+4	T1:1	T1:1	T4:1		T3:OR		T4:RD	T4:AR	T3:B
xor	T1:+4	T1:1	T1:1	T4:1		T3:XOR		T4:RD	T4:AR	T3:B
nor	T1:+4	T1:1	T1:1	T4:1		T3:NOR		T4:RD	T4:AR	T3:B
slt	T1:+4	T1:1	T1:1	T4:1		T3:SUB		T4:RD	T4:AR	T3:B
sltu	T1:+4	T1:1	T1:1	T4:1		T3:SUB		T4:RD	T4:AR	T3:B
sll	T1:+4	T1:1	T1:1	T4:1		T3:SLL		T4:RD	T4:AR	T3:B
srl	T1:+4	T1:1	T1:1	T4:1		T3:SRL		T4:RD	T4:AR	T3:B
sra	T1:+4	T1:1	T1:1	T4:1		T3:SRA		T4:RD	T4:AR	T3:B
sllv	T1:+4	T1:1	T1:1	T4:1		T3:SLLV		T4:RD	T4:AR	T3:B
srlv	T1:+4	T1:1	T1:1	T4:1		T3:SRLV		T4:RD	T4:AR	T3:B
srav	T1:+4	T1:1	T1:1	T4:1		T3:SRAV		T4:RD	T4:AR	T3:B
jr	T1:+4 T2:JrNPC	T1:1 T2:1	T1:1							
addi	T1:+4	T1:1	T1:1	T4:1	T2:SE	T3:ADD		T4:RT	T4:AR	T3:E32
addiu	T1:+4	T1:1	T1:1	T4:1	T2:SE	T3:ADD		T4:RT	T4:AR	T3:E32
andi	T1:+4	T1:1	T1:1	T4:1	T2:UE	T3:AND		T4:RT	T4:AR	T3:E32
ori	T1:+4	T1:1	T1:1	T4:1	T2:UE	T3:OR		T4:RT	T4:AR	T3:E32
xori	T1:+4	T1:1	T1:1	T4:1	T2:UE	T3:XOR		T4:RT	T4:AR	T3:E32
sltiu	T1:+4	T1:1	T1:1	T4:1	T2:SE	T3:SUB		T4:RT	T4:AR	T3:E32
lui	T1:+4	T1:1	T1:1	T4:1	T2:HE	T3:ADD		T4:RT	T4:AR	T3:E32
lw	T1:+4	T1:1	T1:1	T5:1	T2:SE	T3:ADD		T5:RT	T5:DR	T3:E32
sw	T1:+4	T1:1	T1:1		T2:SE	T3:ADD	T4:1			T3:E32
beq	T1:+4 T3:BNPC	T1:1 T3:Zero	T1:1			T3:SUB				T3:B
bne	T1:+4 T3:BNPC	T1:1 T3:Zero	T1:1			T3:SUB				T3:B
bgtz	T1:+4 T3:BNPC	T1:1 T3:Zero	T1:1							
j	T1:+4 T2:JNPC	T1:1 T2:1	T1:1							
jal	T1:+4 T2:JNPC	T1:1 T2:1	T1:1	T2:1				T2:+31	T2:PC4	

顶层模块 minisys:

实例化各模块, 完成数据通路的连接, 并对关键信号进行监测

```

module minisys(
    input rst,          //Reset信号, 高电平复位
    input clk,          //板上的100MHz时钟信号
    output [31:0] imm_ex //作为输出, 完成Implemented 无实际意义
);

    wire clock;         //分频后的23MHz时钟信号
    wire [31:0] debug_wb_pc; //查看PC的值, 连接PC寄存器
    wire debug_wb_rf_wen;   //查看寄存器堆的写使能, 连接RFWr
    wire [4:0] debug_wb_rf_wnum; //查看寄存器堆的目的寄存器号, 连接目的寄存器A3
    wire [31:0] debug_wb_rf_wdata; //查看寄存器堆的写数据, 连接WD

```

定义的变量：

```

wire [31:0] NPCtoPC;    //NPC传给PC的值
wire [31:0] PCtoNPC;    //pc传给NPC的值
//   wire [31:0] imm_ex;
wire [31:0] RF_RD1;     //RF的第一个输出数据
wire [31:0] RF_RD2;     //RF的第二个输出数据
wire [31:0] PC4;        // PC + 4
wire [31:0] Instruction; //当前读出的指令
wire [31:0] read_data;  //内存中读出的数据 要送入RF中
wire [31:0] ALU_C;      //ALU的计算结果
wire [4:0] ALUOp;       //ALU要执行的操作
wire [31:0] AR;        //保存ALU运算结果的寄存器

//通过指令的高6位和低6位完成译码操作
wire [5:0] op;          //指令的高6位
wire [5:0] func;        //指令的低6位
wire [31:0] ins_out;    //保存指令的寄存器 IR

//控制信号
wire [1:0] NPCOP;       //控制NPC下条指令地址的计算方式的信号
wire EXTOp;             //数据拓展信号
wire [1:0] WRSel;       //RF写入数据寄存器号控制信号
wire [1:0] WDSel;       //RF写入数据来源控制信号
wire DMWr;              //存储器写信号
wire BSel;              //控制ALU_B取值的信号
wire ZeroG;             //控制是否要进行计算结果的判零
wire Zero;              //计算结果零标志
wire PCWr;              //PCWr写信号
wire IRWr;              //IRWr IR写信号
wire backsignal;        //一条指令完成后，控制PC的更新

```

各个模块的实例化：


```

CLK u_clk(.clk_in1(clk),.clk_out1(clock));

PC u_pc(.clk(~clock),.reset(rst),.DI(debug_wb_pc),
.PCWr(PCWr),
.DO(PCtoNPC));

NPC u_npc(
.PC(PCtoNPC),
.IMM(ins_out[25:0]),
.NPCOP(NPCOP),
.RA(RF_RD1),
.clock(~clock),
.reset(rst),
// .PCWr(PCWr),
.PC_Update(backsignal),
.PC4(PC4),
.NPCtoPC(debug_wb_pc)
);

programrom u_rom(
.PC(debug_wb_pc),
.clock(clock),
.Instruction(Instruction)
);

```

```

programrom u_rom(
    .PC(debug_wb_pc),
    .clock(clock),
    .Instruction(Instruction)
);

S_EXT u_ext(
    .IMM(ins_out[15:0]),
    .EXTOp(EXTOp),
    .clock(~clock),
    .IMM_EX(imm_ex)
);

RF u_rf(
    .RD1(RF_RD1),
    .RD2(RF_RD2),
    .A3(debug_wb_rf_wnum),
    .Wdata(debug_wb_rf_wdata),
    .op(op),
    .func(func),
    .instruction(ins_out),
    .clock(~clock),
    .ALU_C(ALU_C),
    .DM_RD(read_data),
    //    .NPC_PC(debug_wb_pc),
    .NPC_PC(PCtoNPC),
    .WRSel(WRSel),
    .RFWr(debug_wb_rf_wen),
    .WDSel(WDSel)

```

```

dmemory32 u_ram(
    .read_data(read_data),
    .address(AR),
    .write_data(RF_RD2),
    .Memwrite(DMWr),
    .clock(~clock)
);

```

```

ALU u_alu(
    .ALUOp(ALUOp),
    .IMM_EX(imm_ex),
    .RF_RD2(RF_RD2),
    .RF_RD1(RF_RD1),
    .IM_D(ins_out[10:6]),
    .BSel(BSel),
    .ZeroG(ZeroG),
    .clock(~clock),
    .Zero(Zero),
    .result(ALU_C)
);

```

```

CU u_cu(
    .reset(rst),
    .clock(clock),
    .op(op),
    .func(func),
    .Zero(Zero),
    .NPCOp(NPCOP),
    .PCWr(PCWr),
    .IRWr(IRWr),
    .RFWr(debug_wb_rf_wen),
    .ALUOp(ALUOp),
    .DMWr(DMWr),
    .WRSel(WRSel),
    .WDSel(WDSel),
    .BSel(BSel),
    .ZeroG(ZeroG),
    .EXTOp(EXTOp),
    .backflag(backsignal)
);

```

```

AR_reg u_AR(
    .clock(~clock),
    .ALU_C(ALU_C),
    .AR(AR)
);

IR_reg u_IR(
    .clock(~clock),
    .IRWr(IRWr),
    .ins_in(Instruction),
    .ins_out(ins_out)
);

```

CLK: 时钟分频模块

通过时钟 IP 核完成对输入的 100MHz 时钟信号的分频，使其输出为 23MHz 的时钟信号

```

module CLK(
    input clk_in1,
    output clk_out1
);

    cpucclk cc(
        .clk_in1(clk_in1),
        .clk_out1(clk_out1)
    );
endmodule

```

PC: PC 模块，输出指令的地址

clk:时钟信号

reset:复位信号

DI: 从 NPC 传来的下一条指令的地址

PCWr: PC 写信号

DO: PC 输出信号

若 reset 信号为低电平且 PCWr 被拉高，则将 DI 中的地址写入到寄存器 D 中，通过 DO 线输出，否则 D 信号保持不变，确保输出的 PC 值不会改变，从而当前执行的指令也不会发生改变。

```

module PC(
    input clk,
    input reset,
    input [31:0] DI,
    input PCWr,
    output [31:0] D0
);
    reg [31:0] D = 32'b0;
    assign D0 = D;

    always@(posedge clk)
    begin
        if(!reset && PCWr )
            begin
                D <= DI;
            end
        else
            D <= D;
        end
    end
end

```

NPC: NPC 模块

PC: 从 PC 传来的地址值

IMM: 指令的低 25 位, 用于指令的跳转

NPCOP: 计算下一条指令的控制信号

RA: 寄存器, 保存 JAL 跳转后下一条 PC 值

Clock: 时钟信号

Reset: 复位信号

PC_Update: PC 更新信号, 控制每条指令执行完之后再更新传入 PC 的值

PC4: $PC + 4$

NPCtoPC: NPC 传给 PC 的地址值

```

module NPC(
    input [31:0] PC,
    input [25:0] IMM,
    input [1:0] NPCOP,
    input [31:0] RA,
    input clock,
    input reset,
    input PC_Update,
    output [31:0] PC4,
    output reg [31:0] NPCtoPC
);

```

定义的变量:

```

wire[31:0] BEQ_PC;
wire [31:0] IMM_EX;
wire [31:0] NPCtoPC_reg;
reg PC_flag = 0; //reset降下去之后, 保持PC=32'b0一个指令周期, 执行第一条指令
|

```

BEQ_PC, IMM_EX 保存 NPC 可能的取值

NPCtoPC_reg 保存 NPC 传给 PC 的值, 需通过控制信号才能送到 NPCtoPC 上, 传递给 PC

在执行仿真时, 要在 reset 降下去之前就要给 PC 赋值, 使其读出第一条指令, 否则会造成 reset 置 0 之后, 由于指令未取出, 无控制信号输出, 导致 PC 不再更新的情况; 而在 reset 下降之前, 第一条指令已经开始执行, 这样 reset 下降之后无法保持第一个指令完整的执行一个周期 (会缺少第一个取值阶段的时钟周期), 加入 PC_flag 信号控制 NPC 将下一条指令的地址值延迟一个周期传入 PC 中, 达到第一条指令完整的完成一个指令周期的目的。

具体操作:

```

EXT_4_NPC U_S_EXT_4_NPC(
    .IMM(IMM[15:0]),
    .EXTOP(1'b1),
    .IMM_EX(IMM_EX)
);
assign PC4 = PC + 4;
assign BEQ_PC = (PC4 + {IMM_EX[29:0], 2'b00});
assign NPCtoPC_reg = (NPCOP==2'b10) ? RA : ( (NPCOP==2'b11) ? {PC[31:28], IMM, 2'b00} : ((NPCOP == 2'b01) ? {PC4+[14'b0, IMM[15:0], 2'b00}] : PC4 ));
always@(posedge clock)
begin
    if(!reset && PC_flag == 0)
    //
        if(!reset)
        begin
            PC_flag = 1;
        end
    else if(reset)
        NPCtoPC = 32'b0;
    else if(PC_Update)
        begin
            NPCtoPC = NPCtoPC_reg;
        end
    else
        begin
            NPCtoPC = NPCtoPC_reg;
        end
end
end
endmodule

```

传入 PC 值和 NPCOP 时就已经完成下一条 PC 值的计算, 存在 NPCtoPC_reg 中, 只有 PC_Update 信号置高电平, 才能更新 NPCtoPC 的值, PC_flag 的延迟效果只在 reset 降低后第一个时钟周期有效。

指令存储器:

输入 PC 值, 输出指令值, 采用 block memory 实现存储器的功能

```

module programrom(
    input [31:0]PC,
    input clock,
    output [31:0] Instruction
);

    //分配 64KB ROM,
    //分配 64KB ROM,
    prgrom instmem(
        .clka(clock), // input wire clka
        .addra(PC[15:2]), // input wire [13 : 0] addra
        .douta(Instruction) // output wire [31 : 0] douta
    );
endmodule

```

S_EXT:拓展单元

IMM: 输入的 16 位数据

EXTOp: 控制是否进行符号拓展

Clock: 时钟信号

IMM_EX: 输出的 32 位拓展数据

```

module S_EXT(
    input [15:0] IMM,
    input EXTOp,
    input clock,
    output [31:0] IMM_EX
);

    reg [31:0] imm;

    assign IMM_EX = imm;
    always@(negedge clock)
    begin
        case(EXTOp)
            1'b0: imm <= {16'b0, IMM};
            1'b1: begin
                if( IMM[15] == 0 )
                    imm <= {16'b0, IMM};
                else
                    imm <= {16'b1111111111111111, IMM};
                end
            default: imm <= 32'b0;
        endcase
    end
end

```

EXTOp 为 0 时，进行无符号拓展 高位全部添 0

EXTOp 为 1 时, 进行符号拓展, 根据 16 位数据的最高位判断添 0 还是添 1, 用时钟信号控制是为了满足多周期 CPU 的时序关系。

RF: 寄存器堆

```

module RF(
    output [31:0] RD1,
    output [31:0] RD2,
    output [4:0] A3,
    output [31:0] Wdata,
    output [5:0] op,
    output [5:0] func,
    input [31:0] instruction,
    input clock,
    input [31:0] ALU_C,
    input [31:0] DM_RD,
    input [31:0] NPC_PC,
    input [1:0] WRSel,
    input RFWr,
    input [1:0] WDSel
);

```

输入信号:

instruction: 输入的指令

clock: 时钟信号

ALU_C: ALU 的计算结果

DM_RD: 从存储器中取出的数据

NPC_PC: PC 传给 NPC 的数据, 即当前指令的地址

WRSel: 寄存器堆的目的寄存器号控制信号

RFWr: 寄存器写使能

WDSel: RF 写入数据来源控制信号

输出信号:

RD1: 寄存器堆的第一个输出端口

RD2: 寄存器堆的第二个输出端口

A3: 寄存器堆的目的寄存器号

Wdata: 寄存器堆的写数据

op: 指令的高 6 位

func: 指令的低 6 位


```

wire [4:0] A1;
wire [4:0] A2;
wire [31:0] NPC_PC4;
reg [31:0] regs[0:31];
wire [31:0] WD ;
wire [4:0] A3_wire;
reg init = 0; //初始化信号
assign NPC_PC4 = NPC_PC + 4;
assign op = instruction[31:26];
assign func = instruction[5:0];
assign Wdata = WD;
assign A1 = instruction[25:21];
assign A2 = instruction[20:16];
assign A3_wire = (WRSel == 2'b00)? instruction[20:16]: ((WRSel == 2'b01)? instruction[15:11]: ((WRSel == 2'b10)? 5'b1111:A3_wire));
assign A3 = A3_wire;
assign RD1 = regs[A1];
assign RD2 = regs[A2];
assign WD = RFWr? ((WDSel == 2'b00)? ALU_C : ((WDSel == 2'b01)? DM_RD: ((WDSel == 2'b10)? NPC_PC4:WD))) : WD;

```

A1 是第一个输出端口的寄存器编号

A2 是第二个输出端口的寄存器编号

$NPC_PC4 = PC + 4$

regs 是 32 个 32 位寄存器

WD 是寄存器的写入数据

A3_wire 就是 A3

Init 用于寄存器的输出化

A3 的值由 WRSel 决定

WD 的值在 RDWr 拉高的情况下由 WDSel 决定

```

1. always@(negedge clock)
2.     begin
3.         if( init == 0 )
4.             begin
5.                 regs[0] = 32'b0;
6.                 regs[1] = 32'b0;
7.                 regs[2] = 32'b0;
8.                 regs[3] = 32'b0;
9.                 regs[4] = 32'b0;
10.                regs[5] = 32'b0;
11.                regs[6] = 32'b0;
12.                regs[7] = 32'b0;
13.                regs[8] = 32'b0;
14.                regs[9] = 32'b0;
15.                regs[10] = 32'b0;
16.                regs[11] = 32'b0;
17.                regs[12] = 32'b0;
18.                regs[13] = 32'b0;
19.                regs[14] = 32'b0;
20.                regs[15] = 32'b0;
21.                regs[16] = 32'b0;
22.                regs[17] = 32'b0;
23.                regs[18] = 32'b0;

```

```

24.         regs[19] = 32'b0;
25.         regs[20] = 32'b0;
26.         regs[21] = 32'b0;
27.         regs[22] = 32'b0;
28.         regs[23] = 32'b0;
29.         regs[24] = 32'b0;
30.         regs[25] = 32'b0;
31.         regs[26] = 32'b0;
32.         regs[27] = 32'b0;
33.         regs[28] = 32'b0;
34.         regs[29] = 32'b0;
35.         regs[30] = 32'b0;
36.         regs[31] = 32'b0;
37.         init <= 1;
38.     end
39.
40.     else if( RFWr )
41.     begin
42.         regs[ A3 ] <= WD;
43.     end
44. end

```

regs 的初始化和写入在 clock 的下降沿, 由于 clock 会在 reset 下降之前会进行一次信号的突变, 故在真正执行指令之前就已完成了初始化, 不会影响指令执行时候 regs 的写入操作。

数据存储器:

采用 blockmemory 实现数据存储器的功能, 利用反向时钟, 在地址准备好之后半个周期读出数据, 避免了读出数据时一个周期的延迟

```

module dmemory32(read_data,address,write_data,Memwrite,clock);
    output [31:0] read_data; // 从存储器中获得的数据
    input[31:0] address; //来自 memorio 模块, 源头是来自执行单元算出的
    //alu_result
    input[31:0] write_data; //来自译码单元的 read_data2
    input Memwrite; //来自控制单元
    input clock;
    wire clk;
    // wire [31:0] DR;
    // assign clk = !clock; // 因为使用芯片的固有延迟, RAM 的地址
    assign clk = !clock;
    //线来不及在时钟上升沿准备好, 使得时钟上升沿数据读出有误,
    //所以采用反相时钟, 使得读出数据比地址准备好要晚大约半个时钟, 从而得到正确地址。

```

```
// 分配 64KB RAM
```

```
RAM ram (
    .clka(clk), // input wire clka
    .wea(Memwrite), // input wire [0 : 0] wea
    .addra(address[15:2]), // input wire [13 : 0] addra
    .dina(write_data), // input wire [31 : 0] dina
    .douta(read_data) // output wire [31 : 0] douta
);
```

Read_data: 根据 address 的地址取出存储器中的数据

Address: 读出数据的地址

Write_data: 写入存储器的数据

Memwrite: 存储器写使能

Clock: 时钟信号

ALU: 运算器, 完成 CPU 所需的各种运算操作

```
module ALU(
    input [4:0] ALUOp,
    input [31:0] IMM_EX,
    input [31:0] RF_RD2,
    input [31:0] RF_RD1,
    input [4:0] IM_D,
    input BSel,
    input ZeroG,
    input clock,
    output Zero,
    output reg[31:0] result
);
```

ALUOp: 运算器所要进行的运算操作

IMM_EX: 拓展单元的输出数据

RF_RD2: RF 第二个端口的输出数据

RF_RD1: RF 第一个端口的输出数据

IM_D: 移位指令的移位位数, 给到 ALU 的第三个输入端口

BSel: 控制 ALU 的第二个输入端口的信号

ZeroG: 运算是否需要判零

Clock: 时钟信号

Zero: 零标志位

Result: 运算结果

```
always@(posedge clock)
begin
    ALU_A <= RF_RD1;
    ALU_B <= BSel? IMM_EX:RF_RD2;
end
```

每个时钟上升沿, 准备好 ALU 两个输入端口的数据,

```

always@(negedge clock)
begin
    case(ALUOp)
    `ADD:
    begin
        result = ALU_A + ALU_B ;
    end
    `ADDI:
    begin
        result <= ALU_A + ALU_B ;
    end
    `LW:
    begin
        result <= ALU_A + ALU_B ;
    end
    `SW:
    begin
        result <= ALU_A + ALU_B ;
    end
    `SUB:
    begin
        result = ALU_A - ALU_B ;
    end
    `SLTI:
    begin
        result = ((($signed(ALU_A - ALU_B)) < 0)? 32'b01: 32'b00)
    end
    `BEQ:

```

在时钟下降沿进行 ALU 的运算，得到运算结果，通过宏定义判断所要进行的运算操作部分运算举例：

```
`NOR:
begin
    result = ~(ALU_A | ALU_B);
end
`SLL:
begin
    result = ALU_B << ALU_C;
end
`SRL:
begin
    result = ALU_B >> ALU_C;
end
`SRA:
begin
    result = ($signed(ALU_B)) >>> ALU_C;
end
`SLLV:
begin
    result = ALU_B << ALU_A;
end
`SRLV:
begin
    result = ALU_B >> ALU_A;
end
`SRAV:
begin
    result = ($signed(ALU_B)) >>> ALU_A;
end
```

CU: 控制单元, 根据指令发出相应的控制信号, 控制各部件执行相应的工作

```

module CU(
    input reset,
    input clock,
    input [5:0] op,
    input [5:0] func,
    input Zero,
    output backflag,
    output reg [1:0] NPCOp,
    output reg PCWr,
    output reg IRWr,
    output reg RFWr,
    output reg [4:0] ALUOp,
    output reg DMWr,
    output reg [1:0] WRSel,
    output reg [1:0] WDSel,
    output reg BSel,
    output reg ZeroG,
    output reg EXTOp
    //output M1
);

```

Reset: 复位信号

Clock: 时钟信号

Op: 指令的高 6 位

Func 指令的低 6 位

Zero: 零标志位

Backflag: 一个指令结束后, 状态机返回初始状态的信号

NPCOp: NPC 控制信号

PCWr: PC 写信号

IRWr: IR 写信号

RFWr: RF 写信号

ALUOp: ALU 控制信号

DMWr: 存储器写信号

WRSel: 写入寄存器号控制信号

WDSel: 写入寄存器数据控制信号

BSel: ALU_B 端口数据选择信号

ZeroG: 判零信号

EXTOp: 数据拓展控制信号

为将信号的发出与当前状态时刻对应, 我将状态机的转换放在了控制信号模块中

```

always@(posedge clock)
begin
    if(reset==1)
    begin
        Current_State <= 3'b0;
        NPCOp <= 2'b0;
        PCWr <= 1;
        IRWr <= 1;
        RFWr <= 1;
        WRSel <= 2'b00;
    end
    else
    begin
        if( Current_State <= 3'b0 || backsign == 1 || backsign2 == 1)
            if(backsign == 1 || backsign2 == 1)
                Current_State <= `S1;
            else if(Current_State <= `S1)
                Current_State <= `S2;
            else if(Current_State <= `S2)
                Current_State <= `S3;
            else if(Current_State <= `S3)
                Current_State <= `S4;
            else if(Current_State <= `S4)
                Current_State <= `S5;
            else if(Current_State <= `S5)
                Current_State <= `S1;
        end
    end
end

```

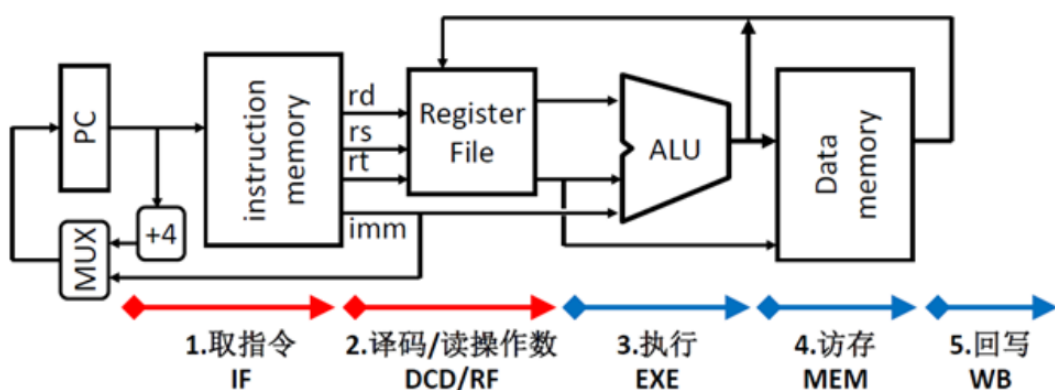
```

`define S1 3'b001
`define S2 3'b010
`define S3 3'b011
`define S4 3'b100
`define S5 3'b101

```

在 reset 高电平时，控制信号不进入状态机中，当 reset 降下去之后，开始状态机的状态转换，五个周期分别对应数据通路的五个阶段

3.1 数据通路



一个指令周期结束后，发出返回信号 `backsign=1`，开始进行下一条指令的取指，实现多周期 CPU 的状态机转换

特别地，在 reset 低电平之前，要把 `PCWr`、`IRWr`、`RFWr`、`WDSel`、`WRSel` 等信号持续拉高

```

always@(*)
begin
    if(reset==1)
    begin
        NPCOp <= 2'b0;
        PCWr <= 1;
        IRWr <= 1;
        //        backsign <= 0;
        RFWr <= 1;
        WDSel <= 2'b00;
        WRSel <= 2'b00;
    end
end

```

确保在进行取指周期时，PC 和指令能够有确定值，RF 中的写入数据能够及时出现，否则会与参考的写入数据信号存在一个周期的误差（即是要完成第一条指令执行之前的初始化）

以 addi 为例：

```

1. case(op)
2.             6'b001000:
3.             begin
4.                 if(T1)
5.                 begin
6.                     backsign2 <= 0;
7.                     NPCOp <= `PC4;
8.                     PCWr <= 1;
9.                     IRWr <= 1;
10.                    backsign <= 0;
11.                    RFWr <= 0;
12.                    DMWr <= 0;
13.                end
14.            else if(T2)
15.            begin
16.                PCWr <= 0;
17.                //                IRWr <= 0;
18.                EXTOp <= 1'b1;
19.            end
20.            else if(T3)
21.            begin
22.                ALUOp <= `ADD;
23.                ZeroG <= 0;
24.                BSel <= 1'b1;
25.            end
26.            else if(T4)
27.            begin
28.                WRSel <= 2'b00;

```



```

29.                WDSel <= 2'b00;
30.                RFWr <= 1;
31.                backsign <= 1;
32.                IRWr <= 0;
33.                end
34.
35.                end

```

第一个周期要把 PCWr、IRWr 等取指信号拉高，上一条指令可能置 1 的 PCWr、DMWr 等信号拉低，同时 backsign 清零，否则状态机不会进行状态转换，第二个周期将 PCWr 置 0，同时在这个周期中完成数据拓展（addi 为带符号拓展，EXTOp=1），第三个周期进行 ALU 的工作，给出 ALU 的操作信号，数据选择信号和判零信号，最后一个周期写回 RF，同时拉高 backsign，回到下一个取指周期

整个取指周期中我拉高了 IRWr 信号，是确保 IR 寄存器始终输出当前所要执行的指令，控制信号的发出依赖指令的高 6 位和低 6 位进行判断，若在第二个周期就将 IRWr 置 0，则指令的输出会中断，后续的控制信号将无法再判断。（其实好像也不需要）

AR 寄存器:保存 ALU 的计算结果

```

module AR_reg(
    input clock,
    input [31:0] ALU_C,
    output reg [31:0] AR
);

always@(posedge clock)
begin
    AR = ALU_C;
end

```

IR 寄存器：保存 IM 中取出的指令

```
module IR_reg(  
    input clock,  
    input IRWr,  
    input [31:0] ins_in,  
    output reg [31:0] ins_out  
);  
  
always@(posedge clock)  
begin  
    if(IRWr==1)  
    begin  
        ins_out <= ins_in ;  
    end  
    else  
    begin  
        ins_out <= ins_out ;  
    end  
end
```

设计主要测试结果（仿真截图或下载照片）

（自己实现的仿真部分代码及截图（至少包括除时钟和存储器外的 2 个模块），贴主要说明问题的时序图并对时序进行分析，可以竖贴）

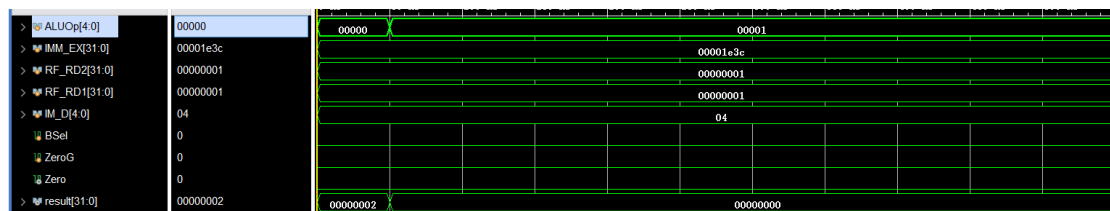
```

`define ADD 5'b00000
`define SUB 5'b00001
`define AND 5'b00011
`define OR 5'b00100
`define XOR 5'b00101
`define NOR 5'b00110
`define SLL 5'b00111
`define SRL 5'b01000
`define SRA 5'b01001
`define SLLV 5'b01010
`define SRLV 5'b01011
`define SRAV 5'b01100
`define X 5'b01101
`define ADDI 5'b01110
`define ANDI 5'b01111
`define ORI 5'b10000
`define XORI 5'b10001
`define SLTI 5'b10010
`define LUI 5'b10011
`define LW 5'b10100
`define SW 5'b10101
`define BEQ 5'b10110
`define BNE 5'b10111
`define BGTZ 5'b11000
`define J 5'b11001
`define JAL 5'b11010

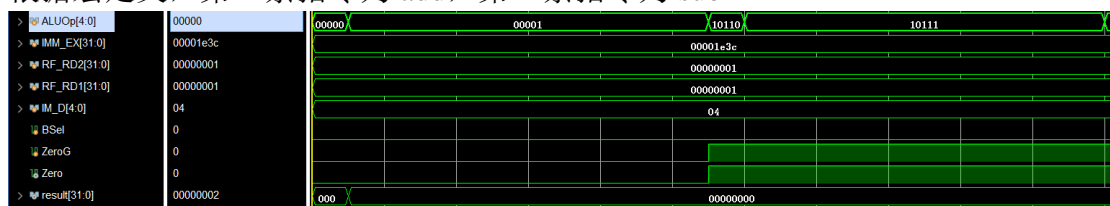
```

多周期实验是在单周期实验的基础上进行修改的，但我实现多周期功能时未进行模块分组仿真，而是直接在测试文件中进行查错的，故部分仿真文件对多周期的设计并不适用，对这些仿真我贴上单周期测试的图

ALU 的仿真只测试了 add、sub、beq、bne 的功能，确保控制信号正确即可



根据宏定义，第一条指令为 add，第二条指令为 sub



第三条指令为 **beq**，第四条指令为 **bne**，这两条指令执行的是 **sub** 操作
可以看到计算结果均正确，且后两条指令 **ZeroG** 判零信号拉高，结果为 0 是 **Zero** 信号也拉高。

```

1. `timescale 1ns / 1ps
2. //////////////////////////////////////
3. // Company:
4. // Engineer:
5. //
6. // Create Date: 2020/06/12 16:07:04
7. // Design Name:
8. // Module Name: ALU_sim
9. // Project Name:
10. // Target Devices:
11. // Tool Versions:
12. // Description:
13. //
14. // Dependencies:
15. //
16. // Revision:
17. // Revision 0.01 - File Created
18. // Additional Comments:
19. //
20. //////////////////////////////////////
21.
22.
23. module ALU_sim(
24.
25. );
26.     reg [4:0]ALUOp;
27.     reg [31:0] IMM_EX;
28.     reg [31:0] RF_RD2;
29.     reg [31:0] RF_RD1;
30.     reg [4:0] IM_D;
31.     reg BSel;
32.     reg ZeroG;
33.     wire Zero;
34.     wire [31:0] result;
35.
36.     ALU aaa(
37.         .ALUOp(ALUOp),
38.         .IMM_EX(IMM_EX),
39.         .RF_RD2(RF_RD2),
40.         .RF_RD1(RF_RD1),

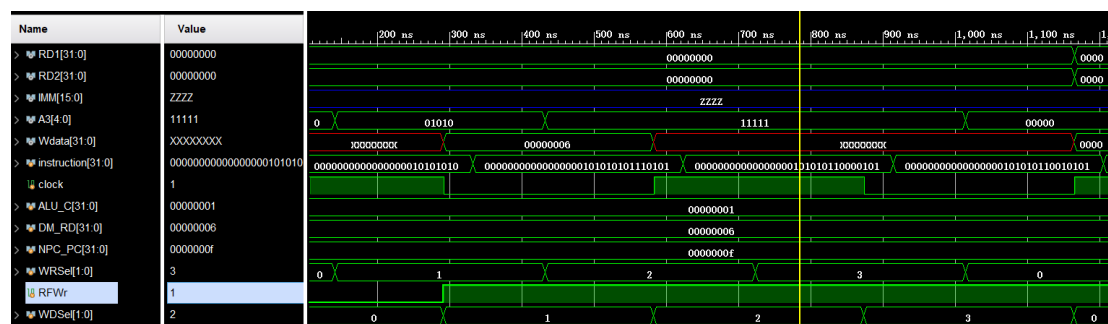
```

```

41.      .IM_D(IM_D),
42.      .BSel(BSel),
43.      .ZeroG(ZeroG),
44.      .Zero(Zero),
45.      .result(result)
46.  );
47.
48.  initial
49.  begin
50.      ALUOp = 5'b00000;
51.      IMM_EX = 32'b001111000111100;
52.      RF_RD1 = 32'b01;
53.      RF_RD2 = 32'b01;
54.      IM_D = 5'b00100;
55.      BSel = 0;
56.      ZeroG = 0;
57.  end
58.
59.  always
60.  begin
61.      #50 ALUOp = ALUOp + 1;
62.      #500 ALUOp = 5'b10110 ;
63.      ZeroG = 1;
64.      BSel = 0;
65.  end
66. endmodule

```

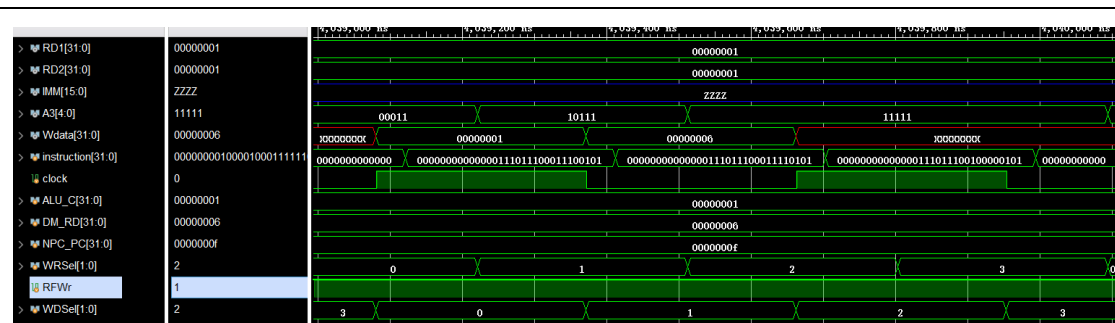
RF 的仿真:



在 RFWr 拉高之前, Wdata (写回值) 不确定, RFWr 拉高之后, Wdata 出现确定值,

```
assign WD = RFWr ? ((WDSel == 2'b00) ? ALU C : ((WDSel == 2'b01) ? DM RD : ((WDSel == 2'b10) ? NPC PC4:WD))) : WD;
```

但由于仿真中 NPC_PC4 无值，故每次 WDSel 变为 2 时，Wdata 又变成了不确定的值。时钟下降沿改变 WRSel 的值，时钟上升沿改变 WDSel 的值。为的是先确定写入的寄存器号，再输入写入寄存器的数据。



```

1. `timescale 1ns / 1ps
2. //////////////////////////////////////
3. // Company:
4. // Engineer:
5. //
6. // Create Date: 2020/06/12 11:20:35
7. // Design Name:
8. // Module Name: RF_sim
9. // Project Name:
10. // Target Devices:
11. // Tool Versions:
12. // Description:
13. //
14. // Dependencies:
15. //
16. // Revision:
17. // Revision 0.01 - File Created
18. // Additional Comments:
19. //
20. //////////////////////////////////////
21.
22.
23. module RF_sim(
24.
25. );
26.   wire [31:0] RD1;
27.   wire [31:0] RD2;
28.   wire [15:0] IMM;
29.   reg [31:0] instruction;
30.   reg clock;
31.   reg [31:0] ALU_C;
32.   reg [31:0] DM_RD;
33.   reg [31:0] NPC_PC4;
34.   reg [1:0] WRSe;
35.   reg RFWr;

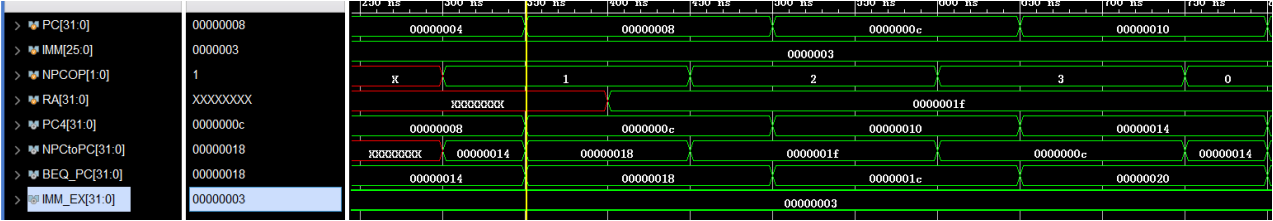
```

```

36.     reg [1:0] WDSel;
37.
38.
39.     RF rrr(
40.         .RD1(RD1),
41.         .RD2(RD2),
42.         .IMM(IMM),
43.         .instruction(instruction),
44.         .clock(clock),
45.         .ALU_C(ALU_C),
46.         .DM_RD(DM_RD),
47.         .NPC_PC4(NPC_PC4),
48.         .WRSel(WRSel),
49.         .RFWr(RFWr),
50.         .WDSel(WDSel)
51.     );
52.
53.     initial
54.     begin
55.         instruction = 32'b0101010101010101;
56.         clock = 0;
57.         ALU_C = 32'b01;
58.         DM_RD = 32'b0110;
59.         NPC_PC4 = 32'b01111;
60.         WRSel = 2'b00;
61.         RFWr = 0;
62.         WDSel = 2'b00;
63.     end
64.
65.     always
66.     begin
67.         #1 clock = ~clock;
68.         #40 instruction = instruction + 16;
69.         #100 WRSel = WRSel + 1;
70.         #150 WDSel = WDSel + 1;
71.         RFWr = 1;
72.     end
73. endmodule

```

NPC 仿真:



```

> PC[31:0] 00000008
> IMM[25:0] 00000003
> NPCOP[1:0] 1
> RA[31:0] XXXXXXXX
> PC4[31:0] 0000000c
> NPCtoPC[31:0] 00000018
> BEQ_PC[31:0] 00000018
> IMM_EX[31:0] 00000003

always@(*)
begin
    case(NPCOP)
        2'b10: NPCtoPC_reg <= RA;
        2'b11: NPCtoPC_reg <= {PC[31:28], IMM, 2'b00};
        2'b01: NPCtoPC_reg <= BEQ_PC;
        2'b00: NPCtoPC_reg <= PC4;
        default: begin end
    endcase
end

```

根据 NPCOP 决定 NPC 的值，

```

1. `timescale 1ns / 1ps
2. //////////////////////////////////////
3. // Company:
4. // Engineer:
5. //
6. // Create Date: 2020/06/12 10:24:13
7. // Design Name:
8. // Module Name: NPC_sim
9. // Project Name:
10. // Target Devices:
11. // Tool Versions:
12. // Description:
13. //
14. // Dependencies:
15. //
16. // Revision:
17. // Revision 0.01 - File Created
18. // Additional Comments:
19. //
20. //////////////////////////////////////
21.
22.
23. module NPC_sim();
24.
25.     reg [31:0] PC;
26.     reg [25:0] IMM;
27.     reg [1:0] NPCOP;

```



```
28.    reg [31:0] RA;
29.    reg clock;
30.    wire [31:0] PC4;
31.    wire [31:0] NPCtoPC;
32.
33.
34.    NPC nnn(
35.
36.        .PC(PC),
37.        .IMM(IMM),
38.        .NPCOP(NPCOP),
39.        .RA(RA),
40.
41.        .PC4(PC4),
42.        .NPCtoPC(NPCtoPC)
43.    );
44.
45.
46.    initial
47.    begin
48.        #100 PC = 32'b0;
49.        #100 IMM = 26'b011;
50.        #100 NPCOP = 2'b00;
51.        #100 RA = 32'b0111111;
52.    end
53.
54.    always
55.    begin
56.        #5 clock = ~clock;
57.        #50 PC = PC + 4;
58.        #100 NPCOP = NPCOP + 1;
59.    end
60. endmodule
```

设计过程中遇到的问题及解决方法

(包括设计过程中的错误及测试过程中遇到的问题)

① 将无符号数转换成带符号数

```
assign result = (ALUOp == `ADD)? (ALU_A + ALU_B):
                ((ALUOp == `SUB)? (ALU_A - ALU_B):
                ((ALUOp == `AND)? (ALU_A & ALU_B):
                ((ALUOp == `BGTZ)? ((($signed(ALU_A)) > 0) ? 32'b01:32'b0):
                ((ALUOp == `OR)? (ALU_A | ALU_B):
                ((ALUOp == `XOR)? (ALU_A ^ ALU_B):
                ((ALUOp == `NOR)? (~ (ALU_A | ALU_B)):
                ((ALUOp == `SLL)? (ALU_B << ALU_C):
                ((ALUOp == `SRL)? (ALU_B >> ALU_C):
                ((ALUOp == `SRA)? (($signed(ALU_B)) >>> ALU_C):
                ((ALUOp == `SLLV)? (ALU_B << ALU_A):
                ((ALUOp == `SRLV)? (ALU_B >> ALU_A):
                ((ALUOp == `SRAV)? (($signed(ALU_B)) >>> ALU_A):
                //((ALUOp == `SRAV)? (ALU_B >>> ALU_A):
                ((ALUOp == `X)? 32'bx:
                ((ALUOp == `LUI)? ((IMM_EX<<16) & 32'b11111111111111110000000000000000):
                ((ALUOp == `J)? 32'bx:
                ((ALUOp == `JAL)? 32'bx:
                ((ALUOp == `SLTI)? ((($signed(ALU_A - ALU_B)) < 0)? 32'b01: 32'b00):
                32'bx
                ))))))))))))));
```

在 ALU 中，针对 SRA 指令进行算术右移时，要将无符号数转换成带符号数，但 wire 类型的数据使用 signed() 后，进行算术右移仍是高位补 0，

```
`SRA:
begin
    result = ($signed(ALU_B)) >>> ALU_C;
end
```

经过尝试，将 result 改为 reg 类型，再在 always 块中进行符号化，即可完成高位补 1 的算术右移

② 在多周期实验测试时，将 DR 寄存器放到与 RAM 统一模块下，导致 Lw 指令在最后一个写回周期时，会多延迟半个周期。

```

    wire clk;
| //    wire [31:0] DR;
|    // assign clk = !clock; // 因为使用芯片的固有延迟, RAM 的地址
    assign clk = !clock;
| //线来不及在时钟上升沿准备好, 使得时钟上升沿数据读出有误,
    //所以采用反相时钟, 使得读出数据比地址准备好要晚大约半个时钟, 从而得到正确地址。
    //    always@(negedge clock)
    //    begin
    //        read_data = DR;
    //    end
| // 分配 64KB RAM
    RAM ram (
        .clk(clk), // input wire clka
        .wea(Memwrite), // input wire [0 : 0] wea
        .addra(address[15:2]), // input wire [13 : 0] addra
        .dina(write_data), // input wire [31 : 0] dina
        .douta(read_data) // output wire [31 : 0] douta
    );

```

后发现是在单周期实验时, 将 `clk` 时钟取反, RAM 的时钟周期与 CPU 的时钟周期刚好相反, 导致 DR 传给 RF 会延迟半个周期。应将 DR 独立写进一个模块中, 用 CPU 的时钟周期作为触发沿, 消除半个周期的延迟。

- ③ 多周期中每个指令在不同时钟周期控制信号的复位, 如上述提过的 `IRWr` 应在指令执行的最后一个周期置 0

```

    else if (T4)
    begin
        RFWr <= 1;
        WRSel <= 2'b01;
        WDSel <= 2'b00;
        backsign <= 1;
        IRWr <= 0;
    end

```

之前设计时。我将 `IRWr` 和 `PCWr` 在取指阶段后就置为 0, 导致置 0 之后的周期无法经过 case 语句进入对应指令的控制信号块中, 要保证指令在整个执行周期不中断, 就要 `IRWr` 一直处于高电平, 但 `PCWr` 又不能置 0, 否则指令存储器会读出下一条指令, 改变当前指令的值, 从而进入错误的控制信号块中, 导致仿真出错。

设计的性能分析

(资源使用情况、主频、功耗数据和自我分析)

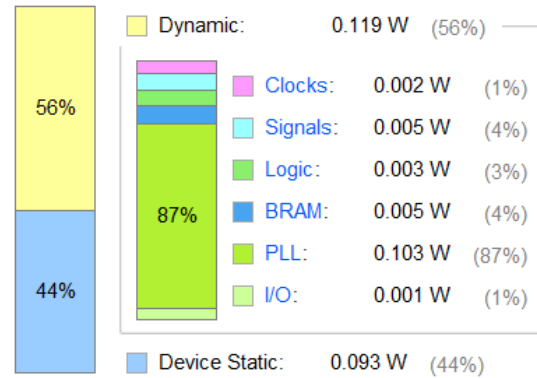
23MHz:

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power: 0.211 W
Design Power Budget: Not Specified
Power Budget Margin: N/A
Junction Temperature: 25.6°C
 Thermal Margin: 59.4°C (22.1 W)
 Effective θ_{JA} : 2.7°C/W
 Power supplied to off-chip devices: 0 W
 Confidence level: Medium

[Launch Power Constraint Advisor](#) to find and fix invalid switching activity

On-Chip Power



Name	^ 1	Slice LUTs (63400)	Block RAM Tile (135)	Bonded IOB (285)	BUFGCTRL (32)	PLLE2_ADV (6)	Slice Registers (126800)	F7 Muxes (31700)	Slice (15850)	LUT as Logic (63400)
minisys		1867	29	34	4	1	1344	256	789	1867
u_alu (ALU)		602	0	0	0	0		0	205	602
u_AR (AR_reg)		0	0	0	0	0		0	6	0
u_clk (CLK)		0	0	0	2	1		0	0	0
cc (cpucclk)		0	0	0	2	1		0	0	0
u_cu (CU)		467	0	0	0	0		0	206	467
u_ext (S_EXT)		9	0	0	0	0		0	34	9
u_IR (IR_reg)		121	0	0	0	0		0	69	121
u_npc (NPC)		2	0	0	0	0		0	28	2
u_pc (PC)		2	0	0	0	0		0	20	2
u_ram (dmemory32)		31	14.5	0	0	0		0	15	31
ram (RAM)		31	14.5	0	0	0		0	15	31
u_rf (RF)		601	0	0	0	0		256	493	601
u_rom (programrom)		31	14.5	0	0	0		0	16	31
instmem (prgrom)		31	14.5	0	0	0		0	16	31

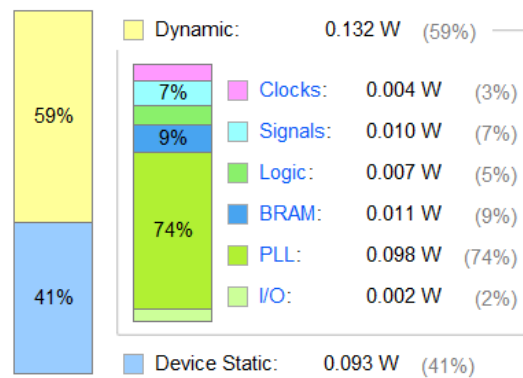
55MHz:

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power: 0.224 W
Design Power Budget: Not Specified
Power Budget Margin: N/A
Junction Temperature: 25.6°C
 Thermal Margin: 59.4°C (22.1 W)
 Effective θ_{JA} : 2.7°C/W
 Power supplied to off-chip devices: 0 W
 Confidence level: Medium

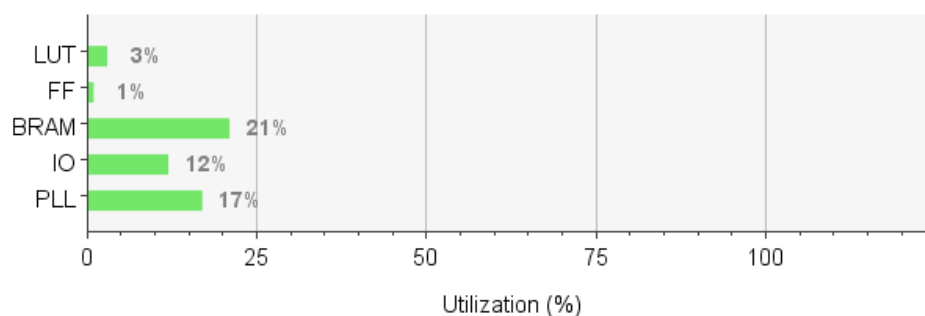
[Launch Power Constraint Advisor](#) to find and fix invalid switching activity

On-Chip Power



Name	Slice LUTs (63400)	Block RAM Tile (135)	Bonded IOB (285)	BUFCTRL (32)	PLLE2_ADV (6)	Slice Registers (126800)	F7 Muxes (31700)	Slice (15850)	LUT as Logic (63400)
minisys	1869	29	34	4	1	1344	256	808	1869
u_alu (ALU)	604	0	0	0	0		0	215	604
u_AR (AR_reg)	0	0	0	0	0		0	8	0
u_clk (CLK)	0	0	0	2	1		0	0	0
u_cu (CU)	467	0	0	0	0		0	200	467
u_ext (S_EXT)	9	0	0	0	0		0	29	9
u_IR (IR_reg)	121	0	0	0	0		0	67	121
u_npc (NPC)	2	0	0	0	0		0	28	2
u_pc (PC)	2	0	0	0	0		0	16	2
u_ram (dmemory32)	31	14.5	0	0	0		0	16	31
u_rf (RF)	601	0	0	0	0		256	494	601
u_rom (programrom)	31	14.5	0	0	0		0	15	31

Resource	Utilization	Available	Utilization %
LUT	1869	63400	2.95
FF	1344	126800	1.06
BRAM	29	135	21.48
IO	34	285	11.93
PLL	1	6	16.67



60MHz:

Worst Negative Slack (WNS):	-0.008 ns	Worst Hold Slack (WHS):	0.173 ns	Worst Pulse Width Slack (WPWS):	2.633 ns
Total Negative Slack (TNS):	-0.008 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	1	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	2902	Total Number of Endpoints:	2902	Total Number of Endpoints:	1360

55MHz:

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0.406 ns	Worst Hold Slack (WHS): 0.144 ns	Worst Pulse Width Slack (WPWS): 3.000 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 2902	Total Number of Endpoints: 2902	Total Number of Endpoints: 1360

All user specified timing constraints are met.

CPU 能跑的最大时钟频率在 55-60MHz 之间

项目总结

（包括设计的总结和还需改进的内容以及收获）

单周期实验的设计过程花了 20 个小时左右，多周期的设计也花费近相同的时间，我认为该实验主要的难点就是时序关系，由于数据通路中不存在控制器模块，故控制信号在什么时刻传入对应模块，又应该在什么时刻重置是应当弄清楚的一个点。在单周期的基础上，对多周期进行最多修改的就是 CU 控制单元模块，要将一个无时序逻辑的通路转化为一个状态转换表，并根据时序关系输出不同的控制信号，是一个比较复杂的点，中间经过了多次错误才找到正确关系。

另一个比较麻烦的点就是整个数据通路的初始化，在 rst 降低到低电平之前就要输出第一条指令和对应的控制信号，但又要保证状态转换不能开始，这中间的一点小错误就可能会导致死循环或者无信号，仿真在第一条指令就终止，有时会导致监测值的实际值与参考值不一致或者错位，只得又回到第一个麻烦的点去调整时序逻辑，这是一个很繁琐的过程。

但是当程序克服了以上两个困难，让仿真能够跑起来之后，后面再出现的问题就很好解决了，如果是写入的寄存器号或者写入的寄存器数据不正确，这种代码的设计错误可能就是一个赋值，一个数据类型的错误，并不需要考虑整体流程的关系，只需要根据控制信号去找到出错的数据就可以很快解决了。

总的来说，这门课在中期还是很折磨人的，其他课程的作业也都大量集中在 5、6 月之间，课业很繁重，但万幸还是扛过来了，顺利的完成了多周期 CPU 的设计。要感谢老师和助教，指导书设计的很详细，答疑也很全面，尽管有的同学十分暴躁，但仍十分耐心。在动手之前看着这么多条指令，这么多个模块，还没有模板参考，我的内心是有一段十分煎熬的心理活动的，但就两周时间，躲也躲不过，就硬着头皮一个个模块写，最后能看到 PASS 的那一瞬间，感觉整个人都升华了，想着这个实验还是很有意思的。

至于实验的改进方面，说实话对于状态机的设计和控制信号的设置我还是有其他想法的，但已改过多次，再加上已经完成了实验，也就没有再去做更多的设计了，所以暂时也没有发现有什么可以改进的地方。

