南京理工大学计算机科学与技术学院
硬件课程设计(Ⅰ)报告

班　级	9191062302
学　号	
姓　名	
教　师	杜姗姗

目录

1 实验目的 3
2 实验设备 3
3 实验任务 3
4 实验整体设计框图 6
5 实现的指令归纳表 8
6 验证程序 9
7 Verilog 程序代码 11
7.1 adder.v 11
7.2 alu.v 11
7.3 data_ram.v 16
7.4 decode.v 19
7.5 exe.v 25
7.6 fetch.v 27
7.7 inst_rom.v 29
7.8 mem.v 31
7.9 wb.v 34
7.10 regfile.v 36
7.11 multi_cycle_cpu.v 38
7.12 multi_cycle_cpu_display.v 46
7.13 tb.v 51
7.14 multi_cycle_cpu.xdc 53
8 仿真波形图 55
9 上板验证图 55
10 心得体会 56

1 实验目的

1.在单周期 CPU 实验完成的提前下，理解多周期的概念。
2.熟悉并掌握多周期 CPU 的原理和设计。
3.进一步提升运用 verilog 语言进行电路设计的能力。

2 实验设备

1.装有 XilinxVivado 的计算机一台。
2.LS-CPU-EXB-002 教学系统实验箱一套。

3 实验任务

1. 本次实验是对单周期 CPU 实验的拔高，前期的实验准备同单周期 CPU 的实验，在单周期 CPU 中只要求实现了五条指令，但此处要求扩展到 30 条以上指令。

   多周期 CPU 是指，一条指令需要花费多个周期才能完成所有操作，在每个周期内只做一部分操作，比如：取指、译码、执行、访存、写回，此时，一条指令执行完，共需 5 个周期，每个周期只做一部分操作。

   将 CPU 划分为多周期的优势在于，每个时钟周期内 CPU 需要做的工作就变少，因此频率可以更高，且每个部件做的事情单一了，比如取指部件只负责从指令存储器中取出指令，因此 CPU 可以进行流水工作，也相当于一个时钟周期完成一条指令。频率更高，依然相当于是一个周期完成一条指令，因此 CPU 可以运行的更快。

   本次实验就是将组成原理实验中实现的单周期 CPU 升级为多周期的，并扩展指令到 30 条以上。
2. 依据单周期实验的设计框图，将其划分为多个功能块，每个功能块占用一个周期完成，即每个功能块从上一个功能块获取信息，做相关动作，完成后将结果锁存到寄存器中，作为下一个功能块的输入。建议划分为 5 个功能块：取指、译码、执行、访存、写回，将理论与实践相结合。

3. 画出升级后的多周期 CPU 的框图，大致框图如图 1。从图中可以看出指令每个周期走完一个功能块，进入下一个功能块。标注的 clk 箭头是去往相邻模块的中间锁存器，因为每个模块的输出需要锁存到寄存器中，下一个模块会从该寄存器中读出数据作为自己的输入，寄存器的锁存是需要时钟控制的。值得注意的是，写回模块所做的就是从访存模块获取要写入寄存器堆的数据和目的寄存器，送往寄存器堆，其所需的 clk 信号是最终发生在寄存器堆的写操作上的。自己设计的框图中要力求精细。

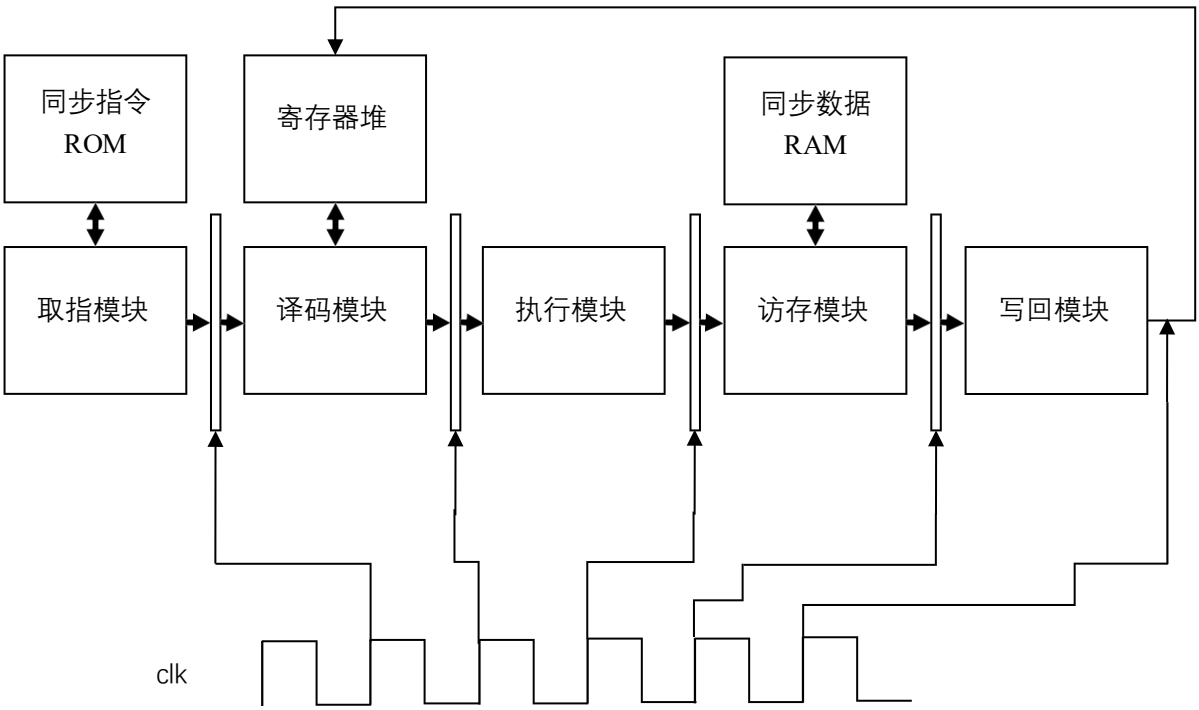


图 1 多周期 CPU 的大致框图

4. 本次课程设计是需要用到之前组成原理实验的成果的，比如 ALU 模块、寄存器堆模块、指令 ROM 模块和数据 RAM 模块，其中 ROM 和 RAM 建议使用调用库 IP 实例化的同步存储器，因为存储器在实际应用中基本都是同步读写的，为了更贴近真实情况，此处建议使用同步 RAM 和 ROM。
5. 在存储器实验中生成的同步 RAM 和 ROM，都是在发送地址后的下一拍才能获得对应数据的。故而在使用同步存储器时，从指令和数据存储器中读取数据就需要等待一拍时钟了，即取指令需要两拍时间，load 操作也需要两拍时间。在真实的处理器系统中，取指令和访存其实都是需要多拍时钟的。
6. 本次实验，需要完成表 1 和表 2 的填写。
7. 根据设计的实验方案，使用 verilog 编写相应代码。
8. 对编写的代码进行仿真，得到正确的波形图。
9. 将以上设计作为一个单独的模块，另外再设计一个外围模块去调用该模块，见图 2。外围模块中需调用封装好的 LCD 触摸屏模块，观察多周期 CPU 的内部状态，比如 32 个寄存器的值，各模块 PC 的值等。并且需要利用触摸功能输入特定数据 RAM 地址，从该 RAM 的调试端口读出数据显示在屏上，以达到实时观察数据存储器内部数据变化的效果。通过这些手段，可以在板上充分验证 CPU 的正确性。

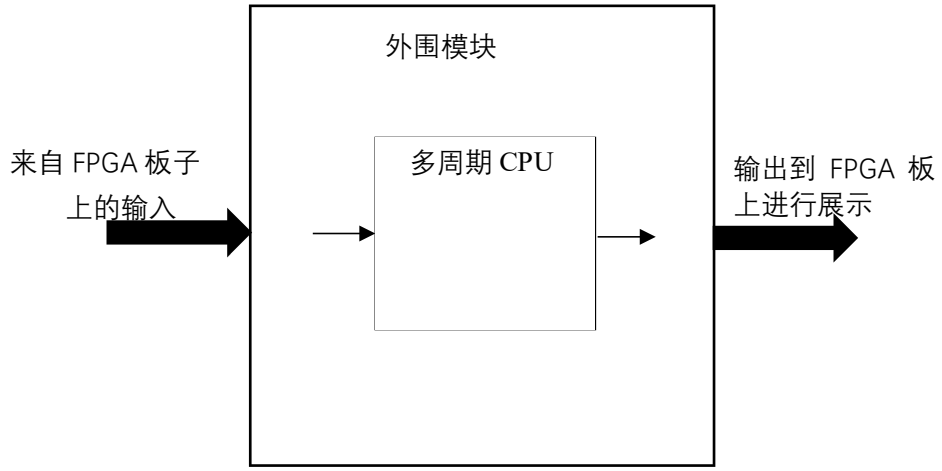


图 2 多周期 CPU 设计实验的顶层模块大致框图

10. 将编写的代码进行综合布局布线，并下载到实验箱中的 FPGA 板上进行演示。

注意：

①MIPS 架构中有延迟槽的设定，其本意是加快流水 CPU 的执行速度，故在多周期 CPU 中该设定无意义，反而带来了 CPU 实现上的麻烦，故建议在多周期 CPU 中不考虑延迟槽技术。

②MIPS 架构中分支和跳转指令参与计算的 PC 值均为延迟槽指令对应的 PC(即分支跳转指令的 PC+4),而由于多周期不考虑延迟槽，故在实验中分支跳转指令参与计算使用本指令的 PC 值，故跳转的偏移量设置需要注意下。比如一条指令“beq,r0,r0,#2”在不考虑延迟槽的多周期 CPU 中，其跳转的目标地址为 beq 指令后面的第 2 条。而在考虑延迟槽的流水 CPU 中，其跳转的目标地址为 beq 指令后面的第 3 条（即延迟槽指令后面的第 2 条）。这里需要理解清楚。

③一般而言控制 CPU 运转的时钟是由 FPGA 板上的时钟输出提供的，但为了方便演示，我们需要在每一个时钟里查看一条指令的运算结果，故演示时理想的时钟是手动输入的，可以使用 FPGA 板上的脉冲开关代替时钟。

4 实验整体设计框图

多周期 CPU 设计在单周期 CPU 基础上，主要做两部分改进。第一部分是控制单元，增加控制电路使每一个时钟只有一个阶段的电路产生的结果有效，并锁存上一阶段的结果用于后续阶段的运行；第二部分是数据通路，增加实现新增指令的电路。

第一部分的改进主要是增加状态机控制及增加各阶段之间的用于锁存的寄存器。由于有 5 个阶段，状态机共有 6 个状态：空闲（IDLE）、取指（FETCH）、译码（DECODE）、执行（EXE）、访存（MEM）、写回（WB），如下图：

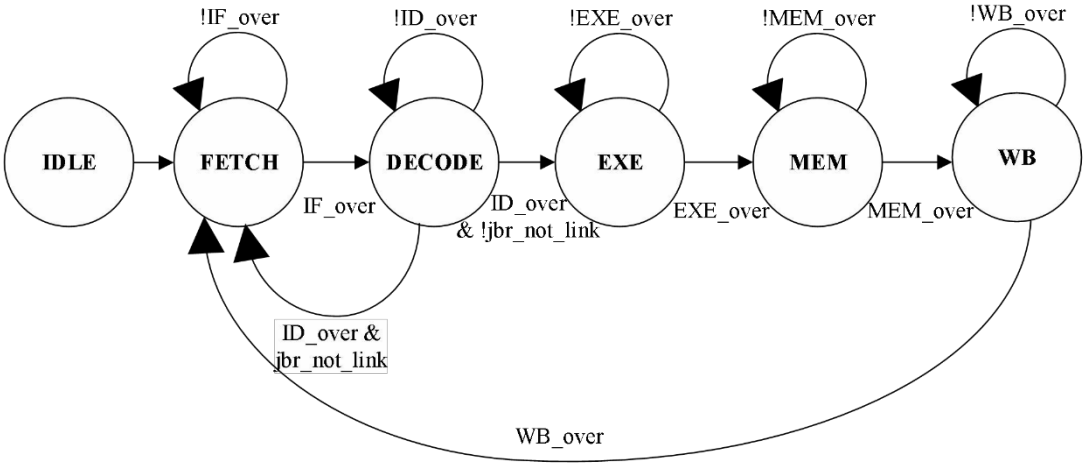


图 8.3 多周期 CPU 的状态机

空闲（IDLE）状态：CPU 在复位时，所有阶段电路都无效，CPU 等待复位结束开始下一状态——取指。

取指（FETCH）状态：在状态机进入取指状态的同时，PC 更新为下一 PC 值。故取指状态下，将 PC 值作为指令存储器的地址去取指令。由于同步指令存储器在下一时钟周期返回指令，因此取指需要两个时钟周期的时间。当取指结束后锁存取指阶段产生的结果——当前 PC 值和指令。

译码（DECODE）状态：类似于单周期 CPU 的译码阶段，主要完成指令译码、读寄存器、判断跳转等。控制单元区分各条指令并产生用于译码、执行、访存、写回的控制信号。当译码结束后锁存译码阶段产生的结果用于下一状态执行：分别用于执行、访存、写回的控制信号、用于执行阶段的两个源操作数、用于访存阶段的写入内存数据、用于写回阶段的写寄存器地址。

执行（EXE）状态：ALU 模块完成操作。当执行结束后锁存执行阶段产生的结果及前级传递的结果：用于访存、写回的控制信号、ALU 结果、内存写入数据、寄存器写地址。

访存（MEM）状态：完成对数据存储器的读或写，并选择出将要写回寄存器的值。当访存结束后锁存访存阶段产生的结果及前级传递的结果：用于写回的控制信号、写回数据、写回地址。

写回（WB）状态：完成寄存器写入。

CPU 复位结束，状态机由 IDLE 进入取指状态，其后在每次上一级结束信号有效的时进入下一状态，写回级结束后返回取指级。当然也有例外，当指令是跳转而非链接跳转指令时，在译码状态后直接返回取下一指令不需要经过执行等后续阶段。

多周期 CPU 的实现框图如下：

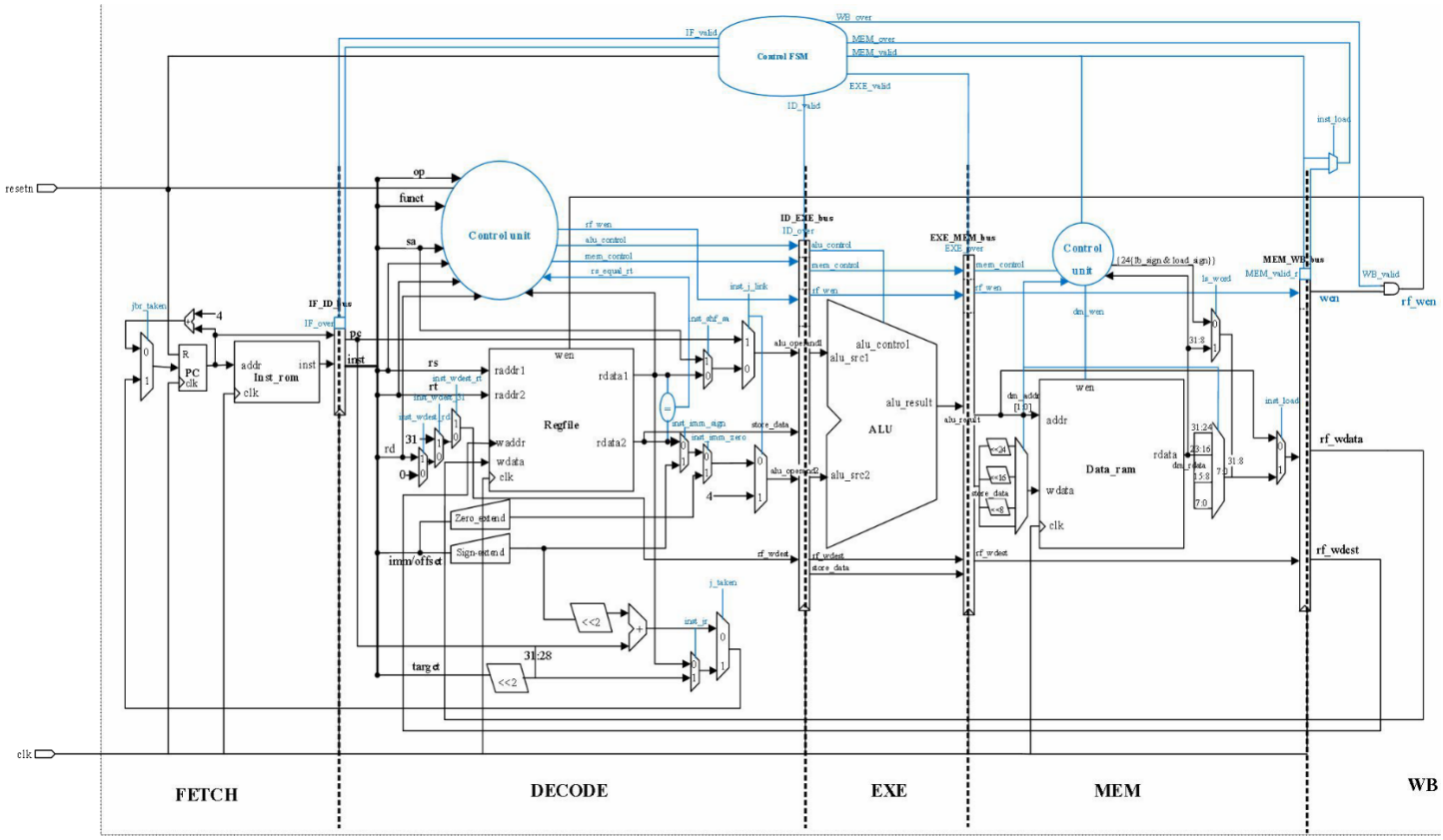


图 8.4　多周期 CPU 的实现框图

5 实现的指令归纳表

表 1mips 基础指令特性归纳表

指令类型	汇编指令	指令码	源操作数 1	源操作数 2	源操作数 3	目的寄存器	功能描述
R型指令	and rd,rs,rt	000000|rs|rt|rd|00000|100100	[rs]	[rt]		rd	GPR[rd]=GPR[rs]&GPR[rt]
	nor rd,rs,rt	000000|rs|rt|rd|00000|100111	[rs]	[rt]		rd	GPR[rd]=~(GPR[rs]|GPR[rt])
	or rd,rs,rt	000000|rs|rt|rd|00000|100101	[rs]	[rt]		rd	GPR[rd]=GPR[rs]|GPR[rt]
	xor rd,rs,rt	000000|rs|rt|rd|00000|100110	[rs]	[rt]		rd	GPR[rd]=GPR[rs]^GPR[rt]
	sll rd,rt,shf	000000|00000|rt|rd|shf|000000		[rt]		rd	GPR[rd]=zero(GPR[rt])<<shf
	sllv rd,rt,rs	000000|rs|rt|rd|00000|000100	[rs]	[rt]		rd	GPR[rd]=zero(GPR[rt])<<(GPR[rs]%32)
	sra rd,rt,shf	000000|00000|rt|rd|shf|000011		[rt]		rd	GPR[rd]=sign(GPR[rt])>>shf
	srav rd,rt,rs	000000|rs|rt|rd|00000|000111	[rs]	[rt]		rd	GPR[rd]=sign(GPR[rt])>>(GPR[rs]%32)
	srl rd,rt,shf	000000|00000|rt|rd|shf|000010		[rt]		rd	GPR[rd]=zero(GPR[rt])>>shf
	srlv rd,rt,rs	000000|rs|rt|rd|00000|000110	[rs]	[rt]		rd	GPR[rd]=zero(GPR[rt])>>GPR[rs]
	jr rs	000000|rs|000000000000|001000	[rs]				PC=GPR[rs]
	addiu rt,rs,imm	001001|rs|rt|imm	[rs]	sign_ext (imm)		rt	GPR[rt]=GPR[rs]+ sign_ext (imm)
	slti rt,rs,imm	001010|rs|rt|imm	[rs]	sign_ext (imm)		rt	GPR[rt]=(sign(GPR[rs])< sign_ext (imm))
	sltiu rt,rs,imm	001011|rs|rt|imm	[rs]	sign_ext (imm)		rt	GPR[rt]=(zero(GPR[rs])< sign_ext (imm))
	beq rs,rt,offset	000100|rs|rt|offs	[rs]	[rt]			if GPR[rs]=GPR[rt] then PC=PC+ sign_ext (offset)<<2

		et					
	bgez rs,offset	000001|rs|00001|offset	[rs]				if GPR[rs]≥0 then PC=PC+ sign_ext (offset)<<2
	bgtz rs,offset	000111|rs|00000|offset	[rs]				if GPR[rs]>0 then PC=PC+ sign_ext (offset)<<2
	blez rs,offset	000110|rs|00000|offset	[rs]				if GPR[rs]≤0 then PC=PC+ sign_ext (offset)<<2
	bltz rs,offset	000001|rs|00000|offset	[rs]				if GPR[rs]<0 thenPC=PC+ sign_ext (offset)<<2
I 型指令	bne rs,rt,offset	000101|rs|rt|offset	[rs]	[rt]			if GPR[rs]≠GPR[rt] then PC=PC+ sign_ext (offset)<<2
	lw rt,offset(b)	100011|b|rt|offset	[b]	sign_ext (offset)		rt	GPR[rt]=Mem[GPR[b]+sign_ext (offset)]
	sw rt,offset(b)	101011|b|rt|offset	[b]	sign_ext (offset)	[rt]		Mem[GPR[b]+sign_ext (offset)]=GPR[rt]
	lb rt,offset(b)	100000|b|rt|offset	[b]	sign_ext (offset)		rt	GPR[rt]=sign(Mem[GPR[b]+sign_ext (offset)])
	lbu rt,offset(b)	100100|b|rt|offset	[b]	sign_ext (offset)		rt	GPR[rt]=zero(Mem[GPR[b]+sign_ext (offset)])
	sb rt,offset(b)	101000|b|rt|offset	[b]	sign_ext (offset)	[rt]		Mem[GPR[b]+sign_ext(offset)]=GPR[rt]
	xori rt,rs,imm	001110|rs|rt|imm	[rs]	zero_ext (imm)		rt	GPR[rt]=GPR[rs]^zero_ext (imm)
J 型指令	j target	000010|target					PC={PC[31:28],target<<2}

6 验证程序

表.2 测试所用汇编程序详述

指令	汇编指令	结果描述	机器指令的机器码	
			16进制	二进制
00H	addiu $1,$0,#1	[$1]=0000_0001H	24010001	0010_0100_0000_00010000_0000_0000_0001
04H	sll $2,$1,#4	[$2]=0000_0010H	00011100	0000_0000_0000_0001_0001_0001_0000_0000
08H	addu $3,$2,$1	[$3]=0000_0011H	00411821	0000_0000_0100_0001_0001_1000_0010_0001
0CH	srl $4,$2,#2	[$4]=0000_0004H	00022082	0000_0000_0000_0010_0010_0000_1000_0010
10H	subu $5,$3,$4	[$5]=0000_000DH	00642823	0000_0000_0110_0100_0010_1000_0010_0011
14H	sw $5,#19($1)	Mem[0000_0014H]=0000_000DH	AC250013	1010_1100_0010_0101_0000_0000_0001_0011
18H	nor $6,$5,$2	[$6]=FFFF_FFE2H	00A23027	0000_0000_1010_0010_0011_0000_0010_0111
1CH	or $7,$6,$3	[$7]=FFFF_FFF3H	00C33825	0000_0000_1100_0011_0011_1000_0010_0101
20H	xor $8,$7,$6	[$8]=0000_0011H	00E64026	0000_0000_1110_0110_0100_0000_0010_0110
24H	sw $8,#28($0)	Mem[0000_001CH]=0000_0011H	AC08001C	1010_1100_0000_1000_0000_0000_0001_1100
28H	slt $9,$6,$7	[$9]=0000_0001H	00C7482A	0000_0000_1100_0111_0100_1000_0010_1010
2CH	beq $9,$1,#2	相等则跳转到指令34H	11210002	0001_0001_0010_0001_0000_0000_0000_0010
30H	addiu $1,$0,#4	不执行	24010004	0010_0100_0000_0001_0000_0000_0000_0100
34H	sllv $10,$4,$4	[$10]=0000_0040H	00845004	0000_0000_1000_0100_0101_0000_0000_0100
38H	jr $10	跳转到指令 40H	01400008	0000_0001_0100_0000_0000_0000_0000_1000
3CH	addiu $1,$0,#2	不执行	24010002	0010_0100_0000_0001_0000_0000_0000_0010
40H	or $11,$2,$1	[$11]=0010_0001H	00415825	0000_0000_0100_0001_0101_1000_0010_0101
44H	sw $7,#28($0)	Men[0000_001CH]=FFFF_FFF3	AC07001C	1010_1100_0000_0111_0000_0000_0001_1100
48H	srav $11,$6,$1	[$11]=FFFF_FFF1H	00265807	0000_0000_0010_0110_0101_1000_0000_0111
4CH	lui $12,#12	[$12]=000C_0000H	3C0C000C	0011_1100_0000_1100_0000_0000_0000_1100
50H	srlv $11,$6,$1	[$11]=7FFF_FFF1H	00265806	0000_0000_0010_0110_0101_1000_0000_0110
54H	addiu $1,$0,#2	[$1]=0000_0010H	24010002	0010_0100_0000_0001_0000_0000_0000_0010
58H	and $2,$1,$1	[$2]=0000_000	00211024	0000_0000_0010_0001_0001_0000_0010_0100
5CH	ori $3,$0,#10	[$3]=0000_1010H	3403000A	0011_0100_0000_0011_0000_0000_0000_1010
60H	beq $2,$3,#4	相等则跳转	10430004	0001_0000_0100_0011_0000_0000_0000_0100
64H	sll $1,$1,#1	[$1]=[$1]*2	00010840	0000_0000_0000_0001_0000_1000_0100_0000
68H	addiu $2,$2,#1	[$2]=[$2]+1	24420001	0010_0100_0100_0010_0000_0000_0000_0001
6CH	j 60H	跳转指令 60H	08000018	0000_1000_0000_0000_0000_0000_0001_1000
70H	sw $1,#28($0)	Men[0000_001CH]=0000_0200H	AC01001C	1010_1100_0000_0001_0000_0000_0001_1100
74H	j 00H	跳转指令 00H	08000000	0000_1000_0000_0000_0000_0000_0000_0000

7 Verilog 程序代码

7.1 adder.v

```
`timescale 1ns / 1ps
//************************************************************************
//    > 文件名: adder.v
//    > 描述   : 加法器，直接使用"+"，会自动调用库里的加法器
//************************************************************************
module adder(
    input  [31:0] operand1,
    input  [31:0] operand2,
    input         cin,
    output [31:0] result,
    output        cout
    );
    assign {cout,result} = operand1 + operand2 + cin;

endmodule
```

7.2 alu.v

```
`timescale 1ns / 1ps
//************************************************************************
//    > 文件名: alu.v
//    > 描述   : ALU模块，可做12种操作
//************************************************************************
module alu(
    input  [11:0] alu_control,  // ALU控制信号
    input  [31:0] alu_src1,     // ALU操作数1,为补码
    input  [31:0] alu_src2,     // ALU操作数2，为补码
    output [31:0] alu_result    // ALU结果
    );

    // ALU控制信号，独热码
    wire alu_add;   //加法操作
    wire alu_sub;   //减法操作
    wire alu_slt;   //有符号比较，小于置位，复用加法器做减法
    wire alu_sltu;  //无符号比较，小于置位，复用加法器做减法
    wire alu_and;   //按位与
    wire alu_nor;   //按位或非
```

```
    wire alu_or;     //按位或
    wire alu_xor;    //按位异或
    wire alu_sll;    //逻辑左移
    wire alu_srl;    //逻辑右移
    wire alu_sra;    //算术右移
    wire alu_lui;    //高位加载

    assign alu_add  = alu_control[11];
    assign alu_sub  = alu_control[10];
    assign alu_slt  = alu_control[ 9];
    assign alu_sltu = alu_control[ 8];
    assign alu_and  = alu_control[ 7];
    assign alu_nor  = alu_control[ 6];
    assign alu_or   = alu_control[ 5];
    assign alu_xor  = alu_control[ 4];
    assign alu_sll  = alu_control[ 3];
    assign alu_srl  = alu_control[ 2];
    assign alu_sra  = alu_control[ 1];
    assign alu_lui  = alu_control[ 0];

    wire [31:0] add_sub_result;
    wire [31:0] slt_result;
    wire [31:0] sltu_result;
    wire [31:0] and_result;
    wire [31:0] nor_result;
    wire [31:0] or_result;
    wire [31:0] xor_result;
    wire [31:0] sll_result;
    wire [31:0] srl_result;
    wire [31:0] sra_result;
    wire [31:0] lui_result;

    assign and_result = alu_src1 & alu_src2;       // 与结果为两数按位与
    assign or_result  = alu_src1 | alu_src2;       // 或结果为两数按位或
    assign nor_result = ~or_result;                // 或非结果为或结果按位取反
    assign xor_result = alu_src1 ^ alu_src2;       // 异或结果为两数按位异或
    assign lui_result = {alu_src2[15:0], 16'd0};   // 立即数装载结果为立即数移位至高半字节

//-----{加法器}begin
//add,sub,slt,sltu 均使用该模块
    wire [31:0] adder_operand1;
    wire [31:0] adder_operand2;
    wire        adder_cin    ;
    wire [31:0] adder_result ;
    wire        adder_cout   ;
    assign adder_operand1 = alu_src1;
```

```
    assign adder_operand2 = alu_add ? alu_src2 : ~alu_src2;
    assign adder_cin      = ~alu_add; //减法需要 cin
    adder adder_module(
    .operand1(adder_operand1),
    .operand2(adder_operand2),
    .cin      (adder_cin      ),
    .result   (adder_result   ),
    .cout     (adder_cout     )
    );

    //加减结果
    assign add_sub_result = adder_result;

    //slt 结果
    //adder_src1[31] adder_src2[31] adder_result[31]
    //        0              1            X(0 或 1)        "正-负"，显然小于不成立
    //        0              0              1              相减为负，说明小于
    //        0              0              0              相减为正，说明不小于
    //        1              1              1              相减为负，说明小于
    //        1              1              0              相减为正，说明不小于
    //        1              0            X(0 或 1)        "负-正"，显然小于成立
    assign slt_result[31:1] = 31'd0;
    assign slt_result[0]    = (alu_src1[31] & ~alu_src2[31]) | (~(alu_src1[31]^alu_src2[31]) &
adder_result[31]);

    //sltu 结果
    //对于 32 位无符号数比较，相当于 33 位有符号数（{1'b0,src1}和{1'b0,src2}）的比较，最
高位 0 为符号位
    //故，可以用 33 位加法器来比较大小，需要对{1'b0,src2}取反,即需要
{1'b0,src1}+{1'b1,~src2}+cin
    //但此处用的为 32 位加法器，只做了运算:                    src1    +    ~src2
+cin
    //32 位加法的结果为{adder_cout,adder_result},则 33 位加法结果应该为
{adder_cout+1'b1,adder_result}
    //对比 slt 结果注释，知道，此时判断大小属于第二三种情况，即源操作数 1 符号位为 0，
源操作数 2 符号位为 0
    //结果的符号位为 1，说明小于，即 adder_cout+1'b1 为 2'b01，即 adder_cout 为 0
    assign sltu_result = {31'd0, ~adder_cout};
//-----{加法器}end

//-----{移位器}begin
    // 移位分三步进行，
    // 第一步根据移位量低 2 位即[1:0]位做第一次移位，
    // 第二步在第一次移位基础上根据移位量[3:2]位做第二次移位，
```

```
    // 第三步在第二次移位基础上根据移位量[4]位做第三次移位。
    wire [4:0] shf;
    assign shf = alu_src1[4:0];
    wire [1:0] shf_1_0;
    wire [1:0] shf_3_2;
    assign shf_1_0 = shf[1:0];
    assign shf_3_2 = shf[3:2];

     // 逻辑左移
    wire [31:0] sll_step1;
    wire [31:0] sll_step2;
    assign sll_step1 = {32{shf_1_0 == 2'b00}} & alu_src2                    // 若 shf[1:0]="00",不移位
                     | {32{shf_1_0 == 2'b01}} & {alu_src2[30:0], 1'd0}      // 若 shf[1:0]="01",左移 1 位
                     | {32{shf_1_0 == 2'b10}} & {alu_src2[29:0], 2'd0}      // 若 shf[1:0]="10",左移 2 位
                     | {32{shf_1_0 == 2'b11}} & {alu_src2[28:0], 3'd0};     // 若 shf[1:0]="11",左移 3 位
    assign sll_step2 = {32{shf_3_2 == 2'b00}} & sll_step1                   // 若 shf[3:2]="00",不移位
                     | {32{shf_3_2 == 2'b01}} & {sll_step1[27:0], 4'd0}     // 若 shf[3:2]="01",第一次移位结果左移 4 位
                     | {32{shf_3_2 == 2'b10}} & {sll_step1[23:0], 8'd0}     // 若 shf[3:2]="10",第一次移位结果左移 8 位
                     | {32{shf_3_2 == 2'b11}} & {sll_step1[19:0], 12'd0};   // 若 shf[3:2]="11",第一次移位结果左移 12 位
    assign sll_result = shf[4] ? {sll_step2[15:0], 16'd0} : sll_step2;      // 若 shf[4]="1",第二次移位结果左移 16 位

    // 逻辑右移
    wire [31:0] srl_step1;
    wire [31:0] srl_step2;
    assign srl_step1 = {32{shf_1_0 == 2'b00}} & alu_src2                    // 若 shf[1:0]="00",不移位
                     | {32{shf_1_0 == 2'b01}} & {1'd0, alu_src2[31:1]}      // 若 shf[1:0]="01",右移 1 位,高位补 0
                     | {32{shf_1_0 == 2'b10}} & {2'd0, alu_src2[31:2]}      // 若 shf[1:0]="10",右移 2 位,高位补 0
                     | {32{shf_1_0 == 2'b11}} & {3'd0, alu_src2[31:3]};     // 若 shf[1:0]="11",右移 3 位,高位补 0
    assign srl_step2 = {32{shf_3_2 == 2'b00}} & srl_step1                   // 若 shf[3:2]="00",不移位
```

```
                        | {32{shf_3_2 == 2'b01}} & {4'd0, srl_step1[31:4]}      // 若 shf[3:2]="01",第一次移位结果右移 4 位,高位补 0
                        | {32{shf_3_2 == 2'b10}} & {8'd0, srl_step1[31:8]}      // 若 shf[3:2]="10",第一次移位结果右移 8 位,高位补 0
                        | {32{shf_3_2 == 2'b11}} & {12'd0, srl_step1[31:12]}; // 若 shf[3:2]="11",第一次移位结果右移 12 位,高位补 0
    assign srl_result = shf[4] ? {16'd0, srl_step2[31:16]} : srl_step2;      // 若 shf[4]="1",第二次移位结果右移 16 位,高位补 0

    // 算术右移
    wire [31:0] sra_step1;
    wire [31:0] sra_step2;
    assign sra_step1 = {32{shf_1_0 == 2'b00}} & alu_src2                                    // 若 shf[1:0]="00",不移位
                        | {32{shf_1_0 == 2'b01}} & {alu_src2[31], alu_src2[31:1]}           // 若 shf[1:0]="01",右移 1 位,高位补符号位
                        | {32{shf_1_0 == 2'b10}} & {{2{alu_src2[31]}}, alu_src2[31:2]}      // 若 shf[1:0]="10",右移 2 位,高位补符号位
                        | {32{shf_1_0 == 2'b11}} & {{3{alu_src2[31]}}, alu_src2[31:3]};     // 若 shf[1:0]="11",右移 3 位,高位补符号位
    assign sra_step2 = {32{shf_3_2 == 2'b00}} & sra_step1                                   // 若 shf[3:2]="00",不移位
                        | {32{shf_3_2 == 2'b01}} & {{4{sra_step1[31]}}, sra_step1[31:4]}    // 若 shf[3:2]="01",第一次移位结果右移 4 位,高位补符号位
                        | {32{shf_3_2 == 2'b10}} & {{8{sra_step1[31]}}, sra_step1[31:8]}    // 若 shf[3:2]="10",第一次移位结果右移 8 位,高位补符号位
                        | {32{shf_3_2 == 2'b11}} & {{12{sra_step1[31]}}, sra_step1[31:12]}; // 若 shf[3:2]="11",第一次移位结果右移 12 位,高位补符号位
    assign sra_result = shf[4] ? {{16{sra_step2[31]}}, sra_step2[31:16]} : sra_step2;    // 若 shf[4]="1",第二次移位结果右移 16 位,高位补符号位
//-----{移位器}end

    // 选择相应结果输出
    assign alu_result = (alu_add|alu_sub) ? add_sub_result[31:0] :
                        alu_slt             ? slt_result :
                        alu_sltu            ? sltu_result :
                        alu_and             ? and_result :
                        alu_nor             ? nor_result :
                        alu_or              ? or_result  :
                        alu_xor             ? xor_result :
                        alu_sll             ? sll_result :
                        alu_srl             ? srl_result :
                        alu_sra             ? sra_result :
                        alu_lui             ? lui_result :
```

```
                            32'd0;
endmodule
```

7.3 data_ram.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2022/09/08 18:04:09
// Design Name:
// Module Name: data_ram
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

//************************************************************************
//    > 文件名: data_ram.v
//    > 描述   ：异步数据存储器模块，采用寄存器搭建而成，类似寄存器堆
//    >           同步写，异步读
//************************************************************************
module data_ram(
    input           clka,           // 时钟
    input   [3:0]   wea,            // 字节写使能
    input   [7:0] addra,            // 地址
    input   [31:0] dina,            // 写数据
    output reg [31:0] douta,        // 读数据

    //调试端口，用于读出数据显示
    input           clkb,           // 时钟
    input   [3:0]   web,            // 字节写使能
    input   [7:0] addrb,            // 地址
    input   [31:0] dinb,            // 写数据
    output reg [31:0] doutb         // 读数据
);
```

```
reg [31:0] DM[31:0];   //数据存储器，字节地址 7'b000_0000~7'b111_1111

//写数据
always @(posedge clka)      // 当写控制信号为 1，数据写入内存
begin
    if (wea[3])
    begin
        DM[addra][31:24] <= dina[31:24];
    end
end
always @(posedge clka)
begin
    if (wea[2])
    begin
        DM[addra][23:16] <= dina[23:16];
    end
end
always @(posedge clka)
begin
    if (wea[1])
    begin
        DM[addra][15: 8] <= dina[15: 8];
    end
end
always @(posedge clka)
begin
    if (wea[0])
    begin
        DM[addra][7 : 0] <= dina[7 : 0];
    end
end

//读数据,取 4 字节
always @(posedge clka)
begin
    case (addra)
        5'd0 : douta <= DM[0 ];
        5'd1 : douta <= DM[1 ];
        5'd2 : douta <= DM[2 ];
        5'd3 : douta <= DM[3 ];
        5'd4 : douta <= DM[4 ];
        5'd5 : douta <= DM[5 ];
        5'd6 : douta <= DM[6 ];
        5'd7 : douta <= DM[7 ];
        5'd8 : douta <= DM[8 ];
        5'd9 : douta <= DM[9 ];
        5'd10: douta <= DM[10];
        5'd11: douta <= DM[11];
        5'd12: douta <= DM[12];
```

```
            5'd13: douta <= DM[13];
            5'd14: douta <= DM[14];
            5'd15: douta <= DM[15];
            5'd16: douta <= DM[16];
            5'd17: douta <= DM[17];
            5'd18: douta <= DM[18];
            5'd19: douta <= DM[19];
            5'd20: douta <= DM[20];
            5'd21: douta <= DM[21];
            5'd22: douta <= DM[22];
            5'd23: douta <= DM[23];
            5'd24: douta <= DM[24];
            5'd25: douta <= DM[25];
            5'd26: douta <= DM[26];
            5'd27: douta <= DM[27];
            5'd28: douta <= DM[28];
            5'd29: douta <= DM[29];
            5'd30: douta <= DM[30];
            5'd31: douta <= DM[31];
        endcase
    end
    //调试端口，读出特定内存的数据
    always @(posedge clkb)
    begin
        case (addrb)
            5'd0 : doutb <= DM[0 ];
            5'd1 : doutb <= DM[1 ];
            5'd2 : doutb <= DM[2 ];
            5'd3 : doutb <= DM[3 ];
            5'd4 : doutb <= DM[4 ];
            5'd5 : doutb <= DM[5 ];
            5'd6 : doutb <= DM[6 ];
            5'd7 : doutb <= DM[7 ];
            5'd8 : doutb <= DM[8 ];
            5'd9 : doutb <= DM[9 ];
            5'd10: doutb <= DM[10];
            5'd11: doutb <= DM[11];
            5'd12: doutb <= DM[12];
            5'd13: doutb <= DM[13];
            5'd14: doutb <= DM[14];
            5'd15: doutb <= DM[15];
            5'd16: doutb <= DM[16];
            5'd17: doutb <= DM[17];
            5'd18: doutb <= DM[18];
            5'd19: doutb <= DM[19];
            5'd20: doutb <= DM[20];
            5'd21: doutb <= DM[21];
            5'd22: doutb <= DM[22];
            5'd23: doutb <= DM[23];
```

```
                5'd24: doutb <= DM[24];
                5'd25: doutb <= DM[25];
                5'd26: doutb <= DM[26];
                5'd27: doutb <= DM[27];
                5'd28: doutb <= DM[28];
                5'd29: doutb <= DM[29];
                5'd30: doutb <= DM[30];
                5'd31: doutb <= DM[31];
            endcase
    end
endmodule
```

7.4 decode.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////
//  > 文件名: decode.v
//  > 描述    :多周期 CPU 的译码模块
//////////////////////////////////////////////////////////////////
module decode(      // 译码级
    inputID_valid,   // 译码级有效信号
    input[ 63:0] IF_ID_bus_r, // IF->ID 总线
    input[ 31:0] rs_value,  // 第一源操作数值
    input[ 31:0] rt_value,  // 第二源操作数值
    output      [      4:0] rs,      // 第一源操作数地址
    output      [      4:0] rt,      // 第二源操作数地址
    output      [ 32:0] jbr_bus,   // 跳转总线
    output      jbr_not_link,// 指令为跳转分支指令,且非 link 类指令
    output      ID_over,    // ID 模块执行完成
    output      [149:0]     ID_EXE_bus,     // ID->EXE 总线
    //展示 PC
    output      [ 31:0] ID_pc
);
//  {IF->ID 总线}begin
    wire [31:0] pc;
    wire [31:0] inst;
    assign {pc, inst} = IF_ID_bus_r;  // IF->ID 总线传 PC 和指令


    //      {IF->ID 总线}end
    //      {指令译码}begin
    wire [5:0] op;
    wire [4:0] rd;
```

```
    wire [4:0] sa;
    wire [5:0] funct;
    wire [15:0] imm;
    wire [15:0] offset;
    wire [25:0] target;
    assign op   = inst[31:26];      // 操作码
    assign rs   = inst[25:21];      // 源操作数 1
    assign rt   = inst[20:16];      // 源操作数 2
    assign rd   = inst[15:11];      // 目标操作数
    assign sa   = inst[10:6];       // 特殊域，可能存放偏移量
    assign funct      = inst[5:0]; // 功能码
    assign imm        = inst[15:0];       // 立即数
    assign offset = inst[15:0];     // 地址偏移量
    assign target = inst[25:0];     // 目标地址
    // 实现指令列表
    wire inst_ADDU, inst_SUBU , inst_SLT , inst_AND;
    wire inst_NOR , inst_OR     , inst_XOR , inst_SLL;
    wire inst_SRL , inst_ADDIU, inst_BEQ , inst_BNE;
    wire inst_LW    , inst_SW   , inst_LUI , inst_J;
    wire inst_SLTU, inst_JALR , inst_JR    , inst_SLLV;
    wire inst_SRA , inst_SRAV , inst_SRLV, inst_SLTIU;
    wire inst_SLTI, inst_BGEZ , inst_BGTZ, inst_BLEZ;
    wire inst_BLTZ, inst_LB     , inst_LBU , inst_SB;
    wire inst_ANDI, inst_ORI    , inst_XORI, inst_JAL;
    wire op_zero;    // 操作码全 0
    wire sa_zero;    // sa 域全 0
    assign op_zero = ~(|op);
    assign sa_zero = ~(|sa);
    assign inst_ADDU   = op_zero & sa_zero      & (funct == 6'b100001);//无符号加法
    assign inst_SUBU   = op_zero & sa_zero      & (funct == 6'b100011);//无符号减法
    assign inst_SLT = op_zero & sa_zero    & (funct == 6'b101010);//小于则置位
    assign inst_SLTU   = op_zero & sa_zero      & (funct == 6'b101011);//无符号小则置
    assign inst_JALR = op_zero & (rt==5'd0) & (rd==5'd31)& sa_zero & (funct == 6'b001001);//跳转寄存器并链接
    assign inst_JR    = op_zero & (rt==5'd0) & (rd==5'd0 )& sa_zero & (funct == 6'b001000);    //跳转寄存器
    assign      inst_AND  = op_zero  & sa_zero  & (funct == 6'b100100);//与运算
    assign      inst_NOR  = op_zero  & sa_zero  & (funct == 6'b100111);//或非运算
    assign      inst_OR     = op_zero  & sa_zero  & (funct == 6'b100101);//或运算
    assign      inst_XOR  = op_zero  & sa_zero  & (funct == 6'b100110);//异或运算
    assign      inst_SLL    = op_zero  & (rs==5'd0)      & (funct == 6'b000000);//逻辑左移
    assign      inst_SLLV = op_zero  & sa_zero  & (funct == 6'b000100);//变量逻辑左移
    assign      inst_SRA  = op_zero  & (rs==5'd0)      & (funct == 6'b000011);//算术右移
```

```
assign      inst_SRAV= op_zero  & sa_zero  & (funct == 6'b000111);//变量算术右移
assign      inst_SRL   =      op_zero & (rs==5'd0) & (funct == 6'b000010);//逻辑右移
assign      inst_SRLV=      op_zero & sa_zero      & (funct == 6'b000110);//变量逻辑右移
assign      inst_ADDIU      =      (op == 6'b001001);      //立即数无符号加法
assign      inst_SLTI   =      (op == 6'b001010);      //小于立即数则置位
assign      inst_SLTIU      =      (op == 6'b001011);      //无符号小于立即数则置位
assign      inst_BEQ   =      (op == 6'b000100);      //判断相等跳转
assign      inst_BGEZ=      (op == 6'b000001) & (rt==5'd1); //大于等于 0 跳转
assign      inst_BGTZ=      (op == 6'b000111) & (rt==5'd0); //大于 0 跳转
assign      inst_BLEZ=      (op == 6'b000110) & (rt==5'd0); //小于等于 0 跳转
assign      inst_BLTZ=      (op == 6'b000001) & (rt==5'd0); //小于 0 跳转
assign      inst_BNE   =      (op == 6'b000101);      //判断不等跳转
assign      inst_LW    =      (op == 6'b100011);      //从内存装载字
assign      inst_SW    =      (op == 6'b101011);      //向内存存储字
assign      inst_LB    =      (op == 6'b100000);      //load 字节（符号扩展）
assign      inst_LBU   =      (op == 6'b100100);      //load 字节（无符号扩展）
assign      inst_SB    =      (op == 6'b101000);      //向内存存储字节
assign      inst_ANDI=      (op == 6'b001100);      //立即数与
assign      inst_LUI   =      (op == 6'b001111) & (rs==5'd0); //立即数装载高半字节
assign      inst_ORI   =      (op == 6'b001101);      //立即数或
assign      inst_XORI=      (op == 6'b001110);      //立即数异或
assign      inst_J     =      (op == 6'b000010);      //跳转
assign      inst_JAL   =      (op == 6'b000011);      //跳转和链接

//跳转分支指令
wire inst_jr;      //寄存器跳转指令
wire inst_j_link;//链接跳转指令
assign inst_jr     = inst_JALR | inst_JR;
assign inst_j_link = inst_JAL      | inst_JALR;
assign jbr_not_link= inst_J  | inst_JR    //全部非 link 类跳转指令
| inst_BEQ        | inst_BNE | inst_BGEZ
| inst_BGTZ | inst_BLEZ | inst_BLTZ;
//load store
wire inst_load;
wire inst_store;
assign inst_load = inst_LW | inst_LB | inst_LBU;   // load 指令
assign inst_store = inst_SW | inst_SB;   // store 指令
//alu 操作分类
wire inst_add, inst_sub, inst_slt,inst_sltu;
wire inst_and, inst_nor, inst_or, inst_xor;
wire inst_sll, inst_srl, inst_sra,inst_lui;
```

```
    assign inst_add = inst_ADDU     | inst_ADDIU | inst_load | inst_store | inst_j_link;    // 做加法位
    assign inst_sub = inst_SUBU;     // 减法位
    assign inst_slt = inst_SLT | inst_SLTI;  // 有符号小于置
    assign inst_sltu= inst_SLTIU | inst_SLTU;    // 无符号小于置
    assign      inst_and    =      inst_AND |inst_ANDI;       //      逻辑与
    assign      inst_nor    =      inst_NOR;         //      逻辑或非
    assign      inst_or     =      inst_OR    |       inst_ORI;  //      逻辑或
    assign      inst_xor    =      inst_XOR |inst_XORI;       //      逻辑或非
    assign      inst_sll    =      inst_SLL |  inst_SLLV;       //      逻辑左移
    assign      inst_srl    =      inst_SRL |  inst_SRLV;       //      逻辑右移
    assign      inst_sra    =      inst_SRA | inst_SRAV;       //      算术右移
    assign inst_lui = inst_LUI;  // 立即数装载高位

    //使用 sa 域作为偏移量的移位指令
    wire inst_shf_sa;
    assign inst_shf_sa =    inst_SLL | inst_SRL | inst_SRA;
    //依据立即数扩展方式分类
    wire inst_imm_zero; //立即数 0 扩展
    wire inst_imm_sign; //立即数符号扩展
    assign inst_imm_zero = inst_ANDI      | inst_LUI  | inst_ORI | inst_XORI;
    assign inst_imm_sign = inst_ADDIU | inst_SLTI | inst_SLTIU
                                | inst_load       | inst_store;

    //依据目的寄存器号分类
    wire inst_wdest_rt;    // 寄存器堆写入地址为 rt 的指令
    wire inst_wdest_31;    // 寄存器堆写入地址为 31 的指令
    wire inst_wdest_rd;    // 寄存器堆写入地址为 rd 的指令
   assign inst_wdest_rt = inst_imm_zero | inst_ADDIU | inst_SLTI |     inst_SLTIU       | inst_load;
    assign      inst_wdest_31    =      inst_JAL;
    assign inst_wdest_rd = inst_ADDU | inst_SUBU | inst_SLT     | inst_SLTU
    | inst_JALR | inst_AND| inst_NOR       | inst_OR
    | inst_XOR       | inst_SLL| inst_SLLV | inst_SRA
    | inst_SRAV | inst_SRL| inst_SRLV;
//  {指令译码}end
//  {分支指令执行}begin
    //无条件跳转
    wire j_taken;
    wire [31:0] j_target;
    assign j_taken = inst_J | inst_JAL | inst_jr;
    //寄存器跳转地址为 rs_value,其他跳转为{pc[31:28],target,2'b00}
    assign j_target = inst_jr ? rs_value :{pc[31:28],target,2'b00};
```

```
//branch 指令
wire rs_equql_rt; wire rs_ez;
wire rs_ltz;
assign rs_equql_rt = (rs_value == rt_value);// GPR[rs]==GPR[rt]
assign rs_ez = ~(|rs_value);// rs 寄存器值为 0
assign rs_ltz = rs_value[31];// rs 寄存器值小于 0
wire br_taken;
wire [31:0] br_target;
assign br_taken = inst_BEQ &     rs_equql_rt //     相等跳转
                | inst_BNE  &     ~rs_equql_rt     //     不等跳转
                | inst_BGEZ & ~rs_ltz  // 大于等于 0 跳转
                | inst_BGTZ & ~rs_ltz & ~rs_ez // 大于 0 跳转
                | inst_BLEZ & (rs_ltz | rs_ez) // 小于等于 0 跳转
                | inst_BLTZ & rs_ltz;   // 小于 0 跳转
// 分支跳转目标地址：PC=PC+offset<<2
assign br_target[31:2] = pc[31:2] + {{14{offset[15]}},   offset};
assign br_target[1:0]  = pc[1:0];

//jump and branch 指令
wire jbr_taken;
wire [31:0] jbr_target;
assign jbr_taken = j_taken | br_taken;
assign jbr_target = j_taken ? j_target : br_target;
//ID 到 IF 的跳转总线
assign jbr_bus = {jbr_taken, jbr_target};
//    {分支指令执行}end

//    {ID 执行完成}begin
//由于是多周期的，不存在数据相关
//故 ID 模块一拍就能完成所有操作
//故 ID_valid 即是 ID_over 信号
assign ID_over = ID_valid;
//    {ID 执行完成}end

//    {ID->EXE 总线}begin
//EXE 需要用到的信息
//ALU 两个源操作数和控制信号
wire [11:0] alu_control;
wire [31:0] alu_operand1;
wire [31:0] alu_operand2;

//所谓链接跳转是将跳转返回的 PC 值存放到 31 号寄存器里
//在多周期 CPU 里，不考虑延迟槽，故链接跳转需要计算 PC+4，存放到 31 号寄存器里
assign alu_operand1 = inst_j_link ? pc :
```

```
                inst_shf_sa ? {27'd0,sa} : rs_value;
    assign alu_operand2 = inst_j_link ? 32'd4 :
                inst_imm_zero ? {16'd0, imm} :
                inst_imm_sign ? {{16{imm[15]}}, imm} : rt_value;
assign    alu_control=      {inst_add,  // ALU 操作码，独热编码
                             inst_sub,
                             inst_slt,
                             inst_sltu,
                             inst_and,
                             inst_nor,
                             inst_or,
                             inst_xor,
                             inst_sll,
                             inst_srl,
                             inst_sra,
                             inst_lui};
//访存需要用到的 load/store 信息
    wire lb_sign;     //load 一字节为有符号 load
    wire ls_word;     //load/store 为字节还是字,0:byte;1:word
    wire [3:0] mem_control;      //MEM 需要使用的控制信号
    wire [31:0] store_data;        //store 操作的存的数据
    assign lb_sign = inst_LB;
    assign ls_word = inst_LW | inst_SW;
    assign mem_control = {inst_load,
                          inst_store,
                          ls_word,
                          lb_sign };

    //写回需要用到的信息
    wire rf_wen;     //写回的寄存器写使能
    wire [4:0] rf_wdest;    //写回的目的寄存器
    assign rf_wen     = inst_wdest_rt | inst_wdest_31 | inst_wdest_rd;
    assign rf_wdest = inst_wdest_rt ? rt ://在不写寄存器堆时，设置为 0
    inst_wdest_31 ? 5'd31 :
    inst_wdest_rd ? rd : 5'd0;
    assign store_data = rt_value;
    assign ID_EXE_bus = {alu_control,alu_operand1,alu_operand2,//EXE 需要使用的信息
                         mem_control,store_data, //MEM 需要使用的信号
                         rf_wen, rf_wdest,    //WB 需要使用的信号
                         pc};      //PC 值
    //     {ID->EXE 总线}end
    //     {展示 ID 模块的 PC 值}begin
    assign ID_pc = pc;
    //     {展示 ID 模块的 PC 值}end
endmodule
```

7.5 exe.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2022/09/08 17:58:05
// Design Name:
// Module Name: exe
// Project Name:
// Target Devices:
// Tool Versions:
// Description: 多周期 CPU 的执行模块
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


    module exe(                 // 执行级
        input   EXE_valid,      // 执行级有效信号
        input   [149:0] ID_EXE_bus_r,// ID->EXE 总线
        output EXE_over, // EXE 模块执行完成
        output [105:0] EXE_MEM_bus, // EXE->MEM 总线
        //展示 PC
        output [ 31:0] EXE_pc
    );
    //     {ID->EXE 总线}begin
    //EXE 需要用到的信息
    //ALU 两个源操作数和控制信号
    wire [11:0] alu_control;
    wire [31:0] alu_operand1;
    wire [31:0] alu_operand2;
    //访存需要用到的 load/store 信息
    wire [3:0] mem_control;     //MEM 需要使用的控制信号
    wire [31:0] store_data;     //store 操作的存的数据

    //写回需要用到的信息
    wire rf_wen;     //写回的寄存器写使能
```

```
    wire [4:0] rf_wdest;    //写回的目的寄存器 31
    //pc
    wire [31:0] pc;
    assign {alu_control,
            alu_operand1,
            alu_operand2,
            mem_control,
            store_data,
            rf_wen,
            rf_wdest,
            pc } = ID_EXE_bus_r;
    //      {ID->EXE 总线}end

    //      {ALU}begin
    wire [31:0] alu_result;
    alu alu_module(
        .alu_control (alu_control ),     // I, 12, ALU 控制信号
        .alu_src1     (alu_operand1),  // I, 32, ALU 操作数 1
        .alu_src2     (alu_operand2),  // I, 32, ALU 操作数 2
        .alu_result   (alu_result )      // O, 32, ALU 结果
    );
    //      {ALU}end

    //      {EXE 执行完成}begin
    //由于是多周期的，不存在数据相关
    //且所有 ALU 运算都可在一拍内完成
    //故 EXE 模块一拍就能完成所有操作
    //故 EXE_valid 即是 EXE_over 信号
    assign EXE_over = EXE_valid;
 // {EXE 执行完成}end
//   {EXE->MEM 总线}begin
    assign EXE_MEM_bus = {mem_control,store_data,//load/store 信息和 store 数据
    alu_result,//alu 运算结果
    rf_wen,rf_wdest, // WB 需要使用的信号
    pc};// PC
//   {EXE->MEM 总线}end
//   {展示 EXE 模块的 PC 值}begin
    assign EXE_pc = pc;
//   {展示 EXE 模块的 PC 值}end
endmodule
```

7.6 fetch.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2022/09/08 17:31:16
// Design Name:
// Module Name: fetch
// Project Name:
// Target Devices:
// Tool Versions:
// Description: 多周期 CPU 的取指模块
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


    `define STARTADDR 32'd0        // 程序起始地址为 0
    module fetch(    // 取指级
        input   clk,   // 时钟
        input   resetn,       // 复位信号，低电平有效
        input   IF_valid,    // 取指级有效信号
        input   next_fetch,// 取下一条指令，用来锁存 PC 值
        input   [31:0] inst, // inst_rom 取出的指令
        input   [32:0] jbr_bus,    // 跳转总线
        output [31:0] inst_addr, // 发往 inst_rom 的取指地址
        output reg    IF_over,     // IF 模块执行完成
        output [63:0] IF_ID_bus, // IF->ID 总线

        //展示 PC 和取出的指令
        output [31:0] IF_pc,
        output [31:0] IF_inst
    );

    //     {程序计数器 PC}begin
    wire [31:0] next_pc;
    wire [31:0] seq_pc;
    reg   [31:0] pc;
```

```
    //跳转 pc
    wire jbr_taken;
    wire [31:0] jbr_target;
    assign {jbr_taken, jbr_target} = jbr_bus; //跳转总线 33
    assign seq_pc[31:2]    = pc[31:2] + 1'b1;       //下一指令地址：  PC=PC+4
    assign seq_pc[1 :0]    = pc[1:0];
    // 新指令：若指令跳转，为跳转地址；否则为下一指令
    assign next_pc = jbr_taken ? jbr_target : seq_pc;
    always @(posedge clk)       // PC 程序计数器
    begin
        if (!resetn)
        begin
            pc <= `STARTADDR; // 复位，取程序起始地址
        end
        else if (next_fetch)
        begin
            pc <= next_pc;       // 不复位，取新指令
        end
    end
//  {程序计数器 PC}end

//  {发往 inst_rom 的取指地址}begin
    assign inst_addr = pc;
//  {发往 inst_rom 的取指地址}end

//  {IF 执行完成}begin
    //由于指令 rom 为同步读写的,
    //取数据时，有一拍延时
    //即发地址的下一拍时钟才能得到对应的指令
    //故取指模块需要两拍时间
    //将 IF_valid 锁存一拍即是 IF_over 信号
    always @(posedge clk)
    begin
        IF_over <= IF_valid;
    end
    //如果指令 rom 为异步读的，则 IF_valid 即是 IF_over 信号，
    //即取指一拍完成
//  {IF 执行完成}end
//  {IF->ID 总线}begin
    assign IF_ID_bus = {pc, inst};
//  {IF->ID 总线}end
//  {展示 IF 模块的 PC 值和指令}begin
    assign IF_pc = pc;
    assign IF_inst = inst;
```

```
//   {展示 IF 模块的 PC 值和指令}end
endmodule
```

7.7 inst_rom.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2022/09/08 18:04:09
// Design Name:
// Module Name: inst_rom
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


//*************************************************************************
//   > 文件名: inst_rom.v
//   > 描述   : 同步指令存储器模块，采用寄存器搭建而成，类似寄存器堆
//   >          内嵌好指令，只读，同步读
//   > 作者   : KY
//   > 日期   : 2022-06-07
//*************************************************************************
module inst_rom(
    input       clka,
    input       [7:0] addra, // 指令地址
    output reg [31:0] douta          // 指令
    );

    wire [31:0] inst_rom[29:0];   // 指令存储器，字节地址 7'b000_0000~7'b111_1111
    //------------ 指令编码 ---------|指令地址|--- 汇编指令 -----|- 指令结果 -----//
    assign inst_rom[ 0] = 32'h24010001; // 00H
    assign inst_rom[ 1] = 32'h00011100; // 04H
    assign inst_rom[ 2] = 32'h00411821; // 08H
```

```
assign inst_rom[ 3] = 32'h00022082; // 0CH
assign inst_rom[ 4] = 32'h00642823; // 10H
assign inst_rom[ 5] = 32'hAC250013; // 14H
assign inst_rom[ 6] = 32'h00A23027; // 18H
assign inst_rom[ 7] = 32'h00C33825; // 1CH
assign inst_rom[ 8] = 32'h00E64026; // 20H
assign inst_rom[ 9] = 32'hAC08001C; // 24H
assign inst_rom[10] = 32'h00C7482A; // 28H
assign inst_rom[11] = 32'h11210002; // 2CH
assign inst_rom[12] = 32'h24010004; // 30H
assign inst_rom[14] = 32'h01400008; // 38H
assign inst_rom[16] = 32'h00415825; // 40H
assign inst_rom[17] = 32'hAC07001C; // 44H
assign inst_rom[18] = 32'h00265807; // 48H
assign inst_rom[19] = 32'h3C0C000C; // 4CH
assign inst_rom[20] = 32'h00265806; // 50H
assign inst_rom[21] = 32'h24010002; // 54H
assign inst_rom[22] = 32'h00211024; // 58H
assign inst_rom[23] = 32'h3403000A; // 5CH
assign inst_rom[24] = 32'h10430004; // 60H
assign inst_rom[25] = 32'h00010840; // 64H
assign inst_rom[26] = 32'h24220001; // 68H
assign inst_rom[27] = 32'h08000060; // 6CH
assign inst_rom[28] = 32'hAC01001C; // 70H
assign inst_rom[29] = 32'h08000000; // 74H

//读指令,取 4 字节
always @(posedge clka)
begin
    case (addra)
        5'd0 : douta <= inst_rom[0 ];
        5'd1 : douta <= inst_rom[1 ];
        5'd2 : douta <= inst_rom[2 ];
        5'd3 : douta <= inst_rom[3 ];
        5'd4 : douta <= inst_rom[4 ];
        5'd5 : douta <= inst_rom[5 ];
        5'd6 : douta <= inst_rom[6 ];
        5'd7 : douta <= inst_rom[7 ];
        5'd8 : douta <= inst_rom[8 ];
        5'd9 : douta <= inst_rom[9 ];
        5'd10: douta <= inst_rom[10];
        5'd11: douta <= inst_rom[11];
        5'd12: douta <= inst_rom[12];
        5'd13: douta <= inst_rom[13];
        5'd14: douta <= inst_rom[14];
        5'd15: douta <= inst_rom[15];
        5'd16: douta <= inst_rom[16];
        5'd17: douta <= inst_rom[17];
        5'd18: douta <= inst_rom[18];
```

```
                5'd19: douta <= inst_rom[19];
                5'd20: douta <= inst_rom[20];
                5'd21: douta <= inst_rom[21];
                5'd22: douta <= inst_rom[22];
                5'd23: douta <= inst_rom[23];
                5'd24: douta <= inst_rom[24];
                5'd25: douta <= inst_rom[25];
                5'd26: douta <= inst_rom[26];
                5'd27: douta <= inst_rom[27];
                5'd28: douta <= inst_rom[28];
                5'd29: douta <= inst_rom[29];
                default: douta <= 32'd0;
            endcase
    end
endmodule
```

7.8 mem.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2022/09/08 18:04:09
// Design Name:
// Module Name: mem
// Project Name:
// Target Devices:
// Tool Versions:
// Description: 多周期 CPU 的访存模块
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module mem(// 访存级
    input clk,// 时钟
    input MEM_valid,// 访存级有效信号
    input [105:0] EXE_MEM_bus_r,// EXE->MEM 总线
    input [ 31:0] dm_rdata, //访存读数据
    output [ 31:0] dm_addr,//访存读写地址
```

```
    output reg [3:0] dm_wen,// 访存写使能
    output reg [ 31:0] dm_wdata,// 访存写数据
    output MEM_over,// MEM 模块执行完成
    output [ 69:0] MEM_WB_bus,// MEM->WB 总线
    //展示 PC
    output [ 31:0] MEM_pc
);
//  {EXE->MEM 总线}begin
    //访存需要用到的 load/store 信息
    wire [3 :0] mem_control;//MEM 需要使用的控制信号
    wire [31:0] store_data;//store 操作的存的数据
    //alu 运算结果
    wire [31:0] alu_result;

    //写回需要用到的信息
    wire rf_wen;//写回的寄存器写使能
    wire [4:0] rf_wdest;//写回的目的寄存器
    //pc
    wire [31:0] pc;

    assign {mem_control,
            store_data,
            alu_result,
            rf_wen,
            rf_wdest,
            pc } = EXE_MEM_bus_r;
    //     {EXE->MEM 总线}end
    //     {load/store 访存}begin
    wire inst_load;   //load 操作
    wire inst_store; //store 操作
    wire ls_word;    //load/store 为字节还是字,0:byte;1:word
    wire lb_sign;     //load 一字节为有符号 load
    assign {inst_load,inst_store,ls_word,lb_sign} = mem_control;
    //访存读写地址
    assign dm_addr = alu_result;
    //store 操作的写使能
    always @ (*)     // 内存写使能信号
    begin
        if (MEM_valid && inst_store) // 访存级有效时,且为 store 操作
        begin
            if (ls_word)
            begin
                dm_wen <= 4'b1111; // 存储字指令，写使能全 1
            end
```

```
            else
            begin // SB 指令，需要依据地址底两位，确定对应的写使能
                case (dm_addr[1:0])
                    2'b00  : dm_wen <= 4'b0001;
                    2'b01  : dm_wen <= 4'b0010;
                    2'b10  : dm_wen <= 4'b0100;
                    2'b11  : dm_wen <= 4'b1000;
                    default : dm_wen <= 4'b0000;
                endcase
            end
        end
        else
        begin
            dm_wen <= 4'b0000;
        end
    end

    //store 操作的写数据
    always @ (*)    // 对于 SB 指令，需要依据地址低两位，移动 store 的字节至对应位置
    begin
        case (dm_addr[1:0])
            2'b00    : dm_wdata <= store_data;
            2'b01    : dm_wdata <= {16'd0,store_data[7:0],8'd0};
            2'b10    : dm_wdata <= {8'd0, store_data[7:0],16'd0};
            2'b11    : dm_wdata <= {store_data[7:0], 24'd0};
            default : dm_wdata <= store_data;
        endcase
    end

    //load 读出的数据
    wire load_sign;
    wire [31:0] load_result;
    assign load_sign = (dm_addr[1:0]==2'd0) ? dm_rdata[ 7] : (dm_addr[1:0]==2'd1) ?
dm_rdata[15] :
                       (dm_addr[1:0]==2'd2) ? dm_rdata[23] : dm_rdata[31] ;
    assign load_result[7:0] = (dm_addr[1:0]==2'd0) ? dm_rdata[ 7:0 ] :
                              (dm_addr[1:0]==2'd1) ? dm_rdata[15:8 ] :
                              (dm_addr[1:0]==2'd2) ? dm_rdata[23:16] :
                              dm_rdata[31:24] ;
    assign load_result[31:8]= ls_word ? dm_rdata[31:8] :
                                        {24{lb_sign & load_sign}};
    //    {load/store 访存}end

    //    {MEM 执行完成}begin
    //由于数据 RAM 为同步读写的,
    //故对 load 指令，取数据时，有一拍延时
    //即发地址的下一拍时钟才能得到 load 的数据
```

```
    //故 mem 在进行 load 操作时有需要两拍时间才能取到数据
    //而对其他操作，则只需要一拍时间
    reg MEM_valid_r;
    always @(posedge clk)
    begin
        MEM_valid_r <= MEM_valid;
    end
    assign MEM_over = inst_load ? MEM_valid_r :MEM_valid;
    //如果数据 ram 为异步读的，则 MEM_valid 即是 MEM_over 信号，
    //即 load 一拍完成
    //    {MEM 执行完成}end

    //    {MEM->WB 总线}begin
    wire [31:0] mem_result; //MEM 传到 WB 的 result 为 load 结果或 ALU 结果
    assign mem_result = inst_load ? load_result : alu_result;
    assign MEM_WB_bus = {rf_wen,rf_wdest, // WB 需要使用的信号
                         mem_result,  // 最终要写回寄存器的数据
                         pc};     // PC 值
    //    {MEM->WB 总线}begin

    //    {展示 MEM 模块的 PC 值}begin
    assign MEM_pc = pc;
    //    {展示 MEM 模块的 PC 值}end
endmodule
```

7.9 wb.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2022/09/08 18:04:09
// Design Name:
// Module Name: wb
// Project Name:
// Target Devices:
// Tool Versions:
// Description: 多周期 CPU 的写回模块
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
```

```
//
//////////////////////////////////////////////////////////////////////////////////

    module wb(      // 写回级
        input   WB_valid, // 写回级有效
        input   [69:0] MEM_WB_bus_r, // MEM->WB 总线
        output rf_wen,     // 寄存器写使能
        output [ 4:0] rf_wdest,   // 寄存器写地址
        output [31:0] rf_wdata,  // 寄存器写数据
        output WB_over,  // WB 模块执行完成

        //展示 PC
        output [ 31:0] WB_pc
    );
    //     {MEM->WB 总线}begin
    //寄存器堆写使能和写地址
    wire wen;
    wire [4:0] wdest;
    //MEM 传来的 result
    wire [31:0] mem_result;
    //pc
    wire [31:0] pc;
    assign {wen,wdest,mem_result,pc} = MEM_WB_bus_r;
    //     {MEM->WB 总线}end
    //     {WB 执行完成}begin
    //WB 模块只是传递寄存器堆的写使能/写地址和写数据
    //可在一拍内完成
    //故 WB_valid 即是 WB_over 信号
    assign WB_over = WB_valid;
    //     {WB 执行完成}end

    //     {WB->regfile 信号}begin
    assign rf_wen    = wen & WB_valid;
    assign rf_wdest = wdest;
    assign rf_wdata = mem_result;
    //     {WB->regfile 信号}end

    //     {展示 WB 模块的 PC 值}begin
    assign WB_pc = pc;
    //     {展示 WB 模块的 PC 值}end
endmodule
```

7.10 regfile.v

```
`timescale 1ns / 1ps
//*************************************************************************
//   > 文件名: regfile.v
//   > 描述   : 寄存器堆模块，同步写，异步读
//   > 作者   : LOONGSON
//   > 日期   : 2016-04-14
//*************************************************************************
module regfile(
    input          clk,
    input          wen,
    input  [4 :0] raddr1,
    input  [4 :0] raddr2,
    input  [4 :0] waddr,
    input  [31:0] wdata,
    input  [4 :0] test_addr,
    output reg [31:0] rdata1,
    output reg [31:0] rdata2,
    output reg [31:0] test_data1,
    output reg [31:0] test_data2,
    output reg [31:0] test_data3,
    output reg [31:0] test_data4,
    output reg [31:0] test_data5,
    output reg [31:0] test_data6,
    output reg [31:0] test_data7,
    output reg [31:0] test_data8,
    output reg [31:0] test_data9,
    output reg [31:0] test_data10,
    output reg [31:0] test_data11,
    output reg [31:0] test_data12
    );
    reg [31:0] rf[31:0];

    // three ported register file
    // read two ports combinationally
    // write third port on rising edge of clock
    // register 0 hardwired to 0

    always @(posedge clk)
    begin
        if (wen)
        begin
            rf[waddr] <= wdata;
        end
    end
```

```
//读端口 1
always @(*)
begin
    case (raddr1)
        5'd1 : rdata1 <= rf[1 ];
        5'd2 : rdata1 <= rf[2 ];
        5'd3 : rdata1 <= rf[3 ];
        5'd4 : rdata1 <= rf[4 ];
        5'd5 : rdata1 <= rf[5 ];
        5'd6 : rdata1 <= rf[6 ];
        5'd7 : rdata1 <= rf[7 ];
        5'd8 : rdata1 <= rf[8 ];
        5'd9 : rdata1 <= rf[9 ];
        5'd10: rdata1 <= rf[10];
        5'd11: rdata1 <= rf[11];
        5'd12: rdata1 <= rf[12];
        5'd13: rdata1 <= rf[13];
        5'd14: rdata1 <= rf[14];
        5'd15: rdata1 <= rf[15];
        5'd16: rdata1 <= rf[16];
        5'd17: rdata1 <= rf[17];
        5'd18: rdata1 <= rf[18];
        5'd19: rdata1 <= rf[19];
        5'd20: rdata1 <= rf[20];
        5'd21: rdata1 <= rf[21];
        5'd22: rdata1 <= rf[22];
        5'd23: rdata1 <= rf[23];
        5'd24: rdata1 <= rf[24];
        5'd25: rdata1 <= rf[25];
        5'd26: rdata1 <= rf[26];
        5'd27: rdata1 <= rf[27];
        5'd28: rdata1 <= rf[28];
        5'd29: rdata1 <= rf[29];
        5'd30: rdata1 <= rf[30];
        5'd31: rdata1 <= rf[31];
        default : rdata1 <= 32'd0;
    endcase
end
//读端口 2
always @(*)
begin
    case (raddr2)
        5'd1 : rdata2 <= rf[1 ];
        5'd2 : rdata2 <= rf[2 ];
        5'd3 : rdata2 <= rf[3 ];
        5'd4 : rdata2 <= rf[4 ];
        5'd5 : rdata2 <= rf[5 ];
        5'd6 : rdata2 <= rf[6 ];
        5'd7 : rdata2 <= rf[7 ];
```

```
                5'd8 : rdata2 <= rf[8 ];
                5'd9 : rdata2 <= rf[9 ];
                5'd10: rdata2 <= rf[10];
                5'd11: rdata2 <= rf[11];
                5'd12: rdata2 <= rf[12];
                5'd13: rdata2 <= rf[13];
                5'd14: rdata2 <= rf[14];
                5'd15: rdata2 <= rf[15];
                5'd16: rdata2 <= rf[16];
                5'd17: rdata2 <= rf[17];
                5'd18: rdata2 <= rf[18];
                5'd19: rdata2 <= rf[19];
                5'd20: rdata2 <= rf[20];
                5'd21: rdata2 <= rf[21];
                5'd22: rdata2 <= rf[22];
                5'd23: rdata2 <= rf[23];
                5'd24: rdata2 <= rf[24];
                5'd25: rdata2 <= rf[25];
                5'd26: rdata2 <= rf[26];
                5'd27: rdata2 <= rf[27];
                5'd28: rdata2 <= rf[28];
                5'd29: rdata2 <= rf[29];
                5'd30: rdata2 <= rf[30];
                5'd31: rdata2 <= rf[31];
                default : rdata2 <= 32'd0;
            endcase
        end
        //调试端口，读出寄存器值显示在触摸屏上
        always @(*)
        begin
            test_data1 <= rf[1 ];
            test_data2 <= rf[2 ];
            test_data3 <= rf[3 ];
            test_data4 <= rf[4 ];
            test_data5 <= rf[5 ];
            test_data6 <= rf[6 ];
            test_data7 <= rf[7 ];
            test_data8 <= rf[8 ];
            test_data9 <= rf[9 ];
            test_data10 <= rf[10];
            test_data11 <= rf[11];
            test_data12 <= rf[12];
        end
    endmodule
```

7.11 multi_cycle_cpu.v

```
`timescale 1ns / 1ps
```

```
//************************************************************************
//    > 文件名: multi_cycle_cpu.v
//    > 描述   :多周期 CPU 模块，共实现 30 条指令
//    >          指令 rom 和数据 ram 均实例化 xilinx IP 得到，为同步读写
//    > 作者   : KY
//    > 日期   : 2022-09-08
//************************************************************************
module multi_cycle_cpu(//多周期 cpu
    input clk,     // 时钟
    input resetn,  // 复位信号，低电平有效

    //display data
    input   [ 4:0] rf_addr,
    input   [31:0] mem_addr,
    output [31:0] rf_data1,
    output [31:0] rf_data2,
    output [31:0] rf_data3,
    output [31:0] rf_data4,
    output [31:0] rf_data5,
    output [31:0] rf_data6,
    output [31:0] rf_data7,
    output [31:0] rf_data8,
    output [31:0] rf_data9,
    output [31:0] rf_data10,
    output [31:0] rf_data11,
    output [31:0] rf_data12,
    output [31:0] mem_data,
    output [31:0] IF_pc,
    output [31:0] IF_inst,
    output [31:0] ID_pc,
    output [31:0] EXE_pc,
    output [31:0] MEM_pc,
    output [31:0] WB_pc,
    output [31:0] display_state
    );
//--------------------{控制多周期的状态机}begin--------------------
//
    reg [2:0] state;// 当前状态
    reg [2:0] next_state;// 下一状态
    //展示当前处理器正在执行哪个模块
    assign display_state = {29'd0,state};
    // 状态机状态
    parameter IDLE = 3'd0;// 开始
    parameter FETCH = 3'd1;   // 取指
    parameter DECODE = 3'd2;// 译码
```

```
parameter EXE  = 3'd3;      // 执行
parameter MEM= 3'd4;      // 访存
parameter WB   = 3'd5;      // 写回

always @ (posedge clk)      // 当前状态
begin
    if (!resetn) begin   // 如果复位信号有效
        state <= IDLE;       // 当前状态为 开始
    end
    else begin    // 否则
        state <= next_state; // 为下一状态
    end
end

wire IF_over;    // IF 模块已执行完
wire ID_over;    // ID 模块已执行完
wire EXE_over; // EXE 模块已执行完
wire MEM_over;       // MEM 模块已执行完
wire WB_over;  // WB 模块已执行完
wire jbr_not_link;//分支指令(非 link 类)，只走 IF 和 ID 级
always @ (*) //下一-状态
begin
    case (state)
        IDLE:
        begin
            next_state = FETCH;//开始->取指
        end
        FETCH:
        begin
            if (IF_over)
            begin
                next_state = DECODE ;//取指->译码
            end
            else
            begin
                next_state = FETCH;//取指->译码
            end
        end
        DECODE:
        begin
            if (ID_over)
            begin //译码->执行或写回
                next_state = jbr_not_link ? FETCH : EXE;
            end
            else
```

```
                begin
                    next_state = DECODE;//取指->译码
                end
            end
            EXE:
            begin
                if (EXE_over)
                begin
                    next_state = MEM;       // 执行->访存
                end
                else
                begin
                    next_state = EXE;       // 取指->译码
                end
            end
            MEM:
            begin
                    if (MEM_over)
                    begin
                        next_state = WB;    // 访存->写回
                    end
                    else
                    begin
                        next_state = MEM; // 取指->译码
                    end
            end
            WB:
            begin
                if (WB_over)
                begin
                    next_state = FETCH;    // 写回->取指
                end
                else
                begin
                    next_state = WB; // 取指->译码
                end
            end
            default : next_state = IDLE;
        endcase
    end
    //5 模块的 valid 信号
    wire IF_valid;
    wire ID_valid;
    wire EXE_valid;
    wire MEM_valid;
    wire WB_valid;
assign    IF_valid = (state == FETCH );      //当前状态为取指时，IF 级有效
```

```
assign    ID_valid = (state == DECODE);   //当前状态为译码时，ID 级有效
assign   EXE_valid = (state == EXE   );     //当前状态为执行时，EXE 级有效
assign   MEM_valid = (state == MEM       );     //当前状态为访存时，MEM 级有效
assign    WB_valid = (state == WB    );     //当前状态为写回时，WB 级有效
//--------------------{控制多周期的状态机}end--------------------
//----------------------{5 级间的总线}begin----------------------
//
    wire [ 63:0] IF_ID_bus;      // IF->ID 级总线
    wire [149:0] ID_EXE_bus;  // ID->EXE 级总线
    wire [105:0] EXE_MEM_bus; // EXE->MEM 级总线
    wire [ 69:0] MEM_WB_bus;       // MEM->WB 级总线
//锁存以上总线信号
reg [ 63:0] IF_ID_bus_r;
reg [149:0] ID_EXE_bus_r;
reg [105:0] EXE_MEM_bus_r;
reg [ 69:0] MEM_WB_bus_r;

    //IF 到 ID 的锁存信号
    always @(posedge clk)
    begin
        if(IF_over)
        begin
            IF_ID_bus_r <= IF_ID_bus;
        end
    end
    //ID 到 EXE 的锁存信号
    always @(posedge clk)
        begin
        if(ID_over)
        begin
            ID_EXE_bus_r <= ID_EXE_bus;
        end
    end
    //EXE 到 MEM 的锁存信号
    always @(posedge clk)
    begin
        if(EXE_over)
        begin
            EXE_MEM_bus_r <= EXE_MEM_bus;
        end
    end
    //MEM 到 WB 的锁存信号
    always @(posedge clk)
    begin
        if(MEM_over)
        begin
```

```
            MEM_WB_bus_r <= MEM_WB_bus;
        end
    end
//----------------------{5 级间的总线}end----------------------
//
//----------------------{其他交互信号}begin----------------------
    //跳转总线
    wire [ 32:0] jbr_bus;
    //IF 与 inst_rom 交互
    wire [31:0] inst_addr;
    wire [31:0] inst;
    //MEM 与 data_ram 交互
    wire [ 3:0] dm_wen;
    wire [31:0] dm_addr;
    wire [31:0] dm_wdata;
    wire [31:0] dm_rdata;
    //ID 与 regfile 交互
    wire [ 4:0] rs;
    wire [ 4:0] rt;
    wire [31:0] rs_value;
    wire [31:0] rt_value;
    //WB 与 regfile 交互
    wire rf_wen;
    wire [ 4:0] rf_wdest;
    wire [31:0] rf_wdata;
//----------------------{其他交互信号}end----------------------
//----------------------{各模块实例化}begin----------------------
//
    wire next_fetch; //即将运行取指模块，需要先锁存 PC 值
//当前状态为 decode，且指令为跳转分支指令(非 link 类)，且 decode 执行完成
//或者，当前状态为 wb，且 wb 执行完成，则即将进入 fetch 状态
    assign next_fetch = (state==DECODE & ID_over & jbr_not_link)|(state==WB & WB_over);
    fetch IF_module(        // 取指级
        .clk    (clk   ),      // I, 1
        .resetn (resetn        ),      // I, 1
        .IF_valid     (IF_valid    ),      // I, 1
        .next_fetch(next_fetch), // I, 1
        .inst    (inst  ),      // I, 32
        .jbr_bus       (jbr_bus     ),      // I, 33
        .inst_addr (inst_addr ),   // O, 32
        .IF_over       (IF_over     ),      // O, 1
        .IF_ID_bus (IF_ID_bus ),      // O, 64

        //展示 PC 和取出的指令
        .IF_pc (IF_pc        ),
        .IF_inst       (IF_inst     )
```

```
);
decode ID_module(    // 译码级
    .ID_valid     (ID_valid   ),      // I, 1
    .IF_ID_bus_r (IF_ID_bus_r ),// I, 64
    .rs_value     (rs_value   ),      // I, 32
    .rt_value     (rt_value   ),      // I, 32
    .rs       (rs     ),      // O, 5
    .rt       (rt     ),      // O, 5
    .jbr_bus      (jbr_bus    ),      // O, 33
    .jbr_not_link(jbr_not_link),    // O, 1
    .ID_over      (ID_over    ),      // O, 1
    .ID_EXE_bus       (ID_EXE_bus     ),      // O, 150
    //展示 PC
     .ID_pc       (ID_pc      )
);

exe EXE_module(      // 执行级
    .EXE_valid  (EXE_valid        ),      // I, 1
    .ID_EXE_bus_r(ID_EXE_bus_r),   // I, 150
    .EXE_over   (EXE_over ),      // O, 1
    .EXE_MEM_bus (EXE_MEM_bus ),      // O, 106 249
    //展示 PC
    .EXE_pc     (EXE_pc    )
);

mem MEM_module(  // 访存级
      .clk   (clk   ),      //      I,      1
      .MEM_valid       (MEM_valid       ),      //      I,      1
      .EXE_MEM_bus_r(EXE_MEM_bus_r),//      I,      106
      .dm_rdata  (dm_rdata  ),      //      I,      32
      .dm_addr   (dm_addr   ),      //      O,      32
      .dm_wen    (dm_wen    ),      //      O,      4
      .dm_wdata (dm_wdata ),      //      O,      32
      .MEM_over        (MEM_over        ),      //      O,      1
      .MEM_WB_bus (MEM_WB_bus ),      //      O,      70

      //展示 PC
      .MEM_pc  (MEM_pc  )
);

wb WB_module(        //      写回级
      .WB_valid (WB_valid ),      //      I, 1
      .MEM_WB_bus_r(MEM_WB_bus_r),   //      I, 70
      .rf_wen      (rf_wen      ),      //      O, 1
      .rf_wdest    (rf_wdest    ),      //      O, 5
      .rf_wdata    (rf_wdata    ),      //      O, 32
      .WB_over  (WB_over  ),      //      O, 1
```

```
        //展示 PC
        .WB_pc   (WB_pc)
    );
    inst_rom inst_rom_module(// 指令存储器
        .clka          (clk), //I,1,时钟
        .addra         (inst_addr[9:2]),//I，8，指令地址
        .douta         (inst)//O，32，指令
    );
    regfile rf_module( // 寄存器堆模块
        .clk   (clk),// I, 1
        .wen   (rf_wen),// I, 1
        .raddr1 (rs),// I, 5
        .raddr2 (rt),// I, 5
        .waddr (rf_wdest),// I, 5
        .wdata (rf_wdata),// I, 32
        .rdata1 (rs_value ),// O, 32
        .rdata2 (rt_value ),// O, 32
        //display rf
        .test_addr(rf_addr),
        .test_data1(rf_data1),
        .test_data2(rf_data2),
        .test_data3(rf_data3),
        .test_data4(rf_data4),
        .test_data5(rf_data5),
        .test_data6(rf_data6),
        .test_data7(rf_data7),
        .test_data8(rf_data8),
        .test_data9(rf_data9),
        .test_data10(rf_data10),
        .test_data11(rf_data11),
        .test_data12(rf_data12)
    );
    data_ram data_ram_module(   // 数据存储模块
        .clka (clk),// I, 1,时钟
        .wea (dm_wen),// I, 1,写使能
        .addra (dm_addr[9:2]),// I, 8,读地址
        .dina (dm_wdata),// I, 32, 写数据
        .douta (dm_rdata),// O, 32, 读数据
        //display mem
        .clkb (clk),
        .web (4'd0),
        .addrb (mem_addr[9:2]),
        .doutb(mem_data),
        .dinb(32'd0)
    );
//----------------------{各模块实例化}end-----------------------
```

```
//
endmodule
```

7.12 multi_cycle_cpu_display.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2022/09/08 15:48:00
// Design Name:
// Module Name: multi_cycle_cpu_display
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module multi_cycle_cpu_display(
    input clk,
    input resetn,

    input btn_clk,


    output lcd_rst,
    output lcd_cs,
    output lcd_rs,
    output lcd_wr,
    output lcd_rd,
    inout[15:0] lcd_data_io,
    output lcd_bl_ctr,
    inout ct_int,
    inout ct_sda,
    output ct_scl,
    output ct_rstn
    );
```

```
wire cpu_clk;
reg btn_clk_r1;
reg btn_clk_r2;
always @(posedge clk)
begin
    if (!resetn)
    begin
        btn_clk_r1<= 1'b0;
    end
    else
    begin
        btn_clk_r1 <= ~btn_clk;
    end

    btn_clk_r2 <= btn_clk_r1;
end

wire clk_en;
assign clk_en = !resetn || (!btn_clk_r1 && btn_clk_r2);
BUFGCE cpu_clk_cg(.I(clk),.CE(clk_en),.O(cpu_clk));

wire [ 4:0] rf_addr;
wire [31:0] rf_data1;
wire [31:0] rf_data2;
wire [31:0] rf_data3;
wire [31:0] rf_data4;
wire [31:0] rf_data5;
wire [31:0] rf_data6;
wire [31:0] rf_data7;
wire [31:0] rf_data8;
wire [31:0] rf_data9;
wire [31:0] rf_data10;
wire [31:0] rf_data11;
wire [31:0] rf_data12;
reg  [31:0] mem_addr;
wire [31:0] mem_data;
wire [31:0] IF_pc;
wire [31:0] IF_inst;
wire [31:0] ID_pc;
wire [31:0] EXE_pc;
wire [31:0] MEM_pc;
wire [31:0] WB_pc;
wire [31:0] display_state;
multi_cycle_cpu cpu(
    .clk      (cpu_clk ),
    .resetn (resetn ),
    .rf_addr (rf_addr ),
    .mem_addr(mem_addr),
    .rf_data1 (rf_data1),
```

```
        .rf_data2 (rf_data2),
        .rf_data3 (rf_data3),
        .rf_data4 (rf_data4),
        .rf_data5 (rf_data5),
        .rf_data6 (rf_data6),
        .rf_data7 (rf_data7),
        .rf_data8 (rf_data8),
        .rf_data9 (rf_data9),
        .rf_data10 (rf_data10),
        .rf_data11 (rf_data11),
        .rf_data12 (rf_data12),
        .mem_data(mem_data),
        .IF_pc   (IF_pc   ),
        .IF_inst (IF_inst ),
        .ID_pc   (ID_pc   ),
        .EXE_pc (EXE_pc ),
        .MEM_pc (MEM_pc ),
        .WB_pc   (WB_pc   ),
        .display_state (display_state)
    );

     reg display_valid;
     reg [39:0] display_name;
     reg [31:0] display_value;
     wire [5 :0] display_number;
     wire       input_valid;
     wire [31:0] input_value;
     lcd_module lcd_module(
        .clk      (clk      ),   //10Mhz
        .resetn (resetn ),
        .display_valid   (display_valid ),
        .display_name     (display_name     ),
        .display_value   (display_value ),
        .display_number (display_number),
        .input_valid      (input_valid      ),
        .input_value      (input_value      ),

        .lcd_rst      (lcd_rst      ),
        .lcd_cs (lcd_cs ),
        .lcd_rs (lcd_rs ),
        .lcd_wr (lcd_wr ),
        .lcd_rd (lcd_rd ),
        .lcd_data_io      (lcd_data_io      ),
        .lcd_bl_ctr (lcd_bl_ctr ),
        .ct_int (ct_int ),
        .ct_sda (ct_sda ),
        .ct_scl (ct_scl ),
        .ct_rstn      (ct_rstn      )
    );
```

```
always @(posedge clk)
begin
    if (!resetn)
    begin
        mem_addr <= 32'd0;
    end
    else if (input_valid)
    begin
        mem_addr <= input_value;
    end
end
assign rf_addr = display_number-6'd11;

always @(posedge clk)
begin
    if (display_number >6'd10 && display_number <6'd43 )
    begin
        display_valid <= 1'b1;
        display_name[39:16] <= "REG";
        display_name[15: 8] <= {4'b0011,3'b000,rf_addr[4]};
        display_name[7 : 0] <= {4'b0011,rf_addr[3:0]};
        case (rf_addr)
            5'd1 : display_value <= rf_data1 ;
            5'd2 : display_value <= rf_data2 ;
            5'd3 : display_value <= rf_data3 ;
            5'd4 : display_value <= rf_data4 ;
            5'd5 : display_value <= rf_data5 ;
            5'd6 : display_value <= rf_data6 ;
            5'd7 : display_value <= rf_data7 ;
            5'd8 : display_value <= rf_data8 ;
            5'd9 : display_value <= rf_data9 ;
            5'd10: display_value <= rf_data10;
            5'd11: display_value <= rf_data11;
            5'd12: display_value <= rf_data12;
            default: display_value <= 32'd0;
        endcase
    end
    else
    begin
        case(display_number)
            6'd1 :
            begin
                display_valid <= 1'b1;
                display_name      <= "IF_PC";
                display_value <= IF_pc;
            end
            6'd2 :
            begin
```

```
            display_valid <= 1'b1;
            display_name  <= "IF_IN";
            display_value <= IF_inst;
        end
        6'd3 :
        begin
            display_valid <= 1'b1;
            display_name  <= "ID_PC";
            display_value <= ID_pc;
        end
        6'd4 :
        begin
            display_valid <= 1'b1;
            display_name  <= "EXEPC";
            display_value <= EXE_pc;
        end
        6'd5 :
        begin
            display_valid <= 1'b1;
            display_name  <= "MEMPC";
            display_value <= MEM_pc;
        end
        6'd6 :
        begin
            display_valid <= 1'b1;
            display_name  <= "WB_PC";
            display_value <= WB_pc;
        end
        6'd7 :
        begin
            display_valid <= 1'b1;
            display_name  <= "MADDR";
            display_value <= mem_addr;
        end
        6'd8 :
        begin
            display_valid <= 1'b1;
            display_name  <= "MDATA";
            display_value <= mem_data;
        end
        6'd9 :
        begin
            display_valid <= 1'b1;
            display_name  <= "STATE";
            display_value <= display_state;
        end
        default :
        begin
            display_valid <= 1'b0;
```

```
                    display_name <= 40'd0;
                    display_value <= 32'd0;
                end
            endcase
        end
    end
endmodule
```

7.13 tb.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2022/09/09 18:20:46
// Design Name:
// Module Name: tb
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module tb;

    // Inputs
    reg clk;
    reg resetn;
    reg [4:0] rf_addr;
    reg [31:0] mem_addr;

    // Outputs
    wire [31:0] rf_data1;
    wire [31:0] rf_data2;
    wire [31:0] rf_data3;
    wire [31:0] rf_data4;
    wire [31:0] rf_data5;
    wire [31:0] rf_data6;
    wire [31:0] rf_data7;
```

```
wire [31:0] rf_data8;
wire [31:0] rf_data9;
wire [31:0] rf_data10;
wire [31:0] rf_data11;
wire [31:0] rf_data12;
wire [31:0] mem_data;        //内存地址对应的数据
wire [31:0] IF_pc;       //IF 模块的 PC
wire [31:0] IF_inst;     //IF 模块取出的指令
wire [31:0] ID_pc;       //ID 模块的 PC
wire [31:0] EXE_pc;   //EXE 模块的 PC
wire [31:0] MEM_pc; //MEM 模块的 PC
wire [31:0] WB_pc;    //WB 模块的 PC
wire [31:0] display_state; //展示 CPU 当前状态

multi_cycle_cpu cpu(
    .clk      (clk ),
    .resetn (resetn ),
    .rf_addr (rf_addr ),
    .mem_addr(mem_addr),
    .rf_data1 (rf_data1),
    .rf_data2 (rf_data2),
    .rf_data3 (rf_data3),
    .rf_data4 (rf_data4),
    .rf_data5 (rf_data5),
    .rf_data6 (rf_data6),
    .rf_data7 (rf_data7),
    .rf_data8 (rf_data8),
    .rf_data9 (rf_data9),
    .rf_data10 (rf_data10),
    .rf_data11 (rf_data11),
    .rf_data12 (rf_data12),
    .mem_data(mem_data),
    .IF_pc   (IF_pc   ),
    .IF_inst (IF_inst ),
    .ID_pc   (ID_pc   ),
    .EXE_pc (EXE_pc ),
    .MEM_pc (MEM_pc ),
    .WB_pc   (WB_pc   ),
    .display_state (display_state)
);
initial begin
    // Initialize Inputs
    clk = 0;
    resetn = 0;
    rf_addr = 'hB;
    mem_addr = 'h1C;

    // Wait 100 ns for global reset to finish
```

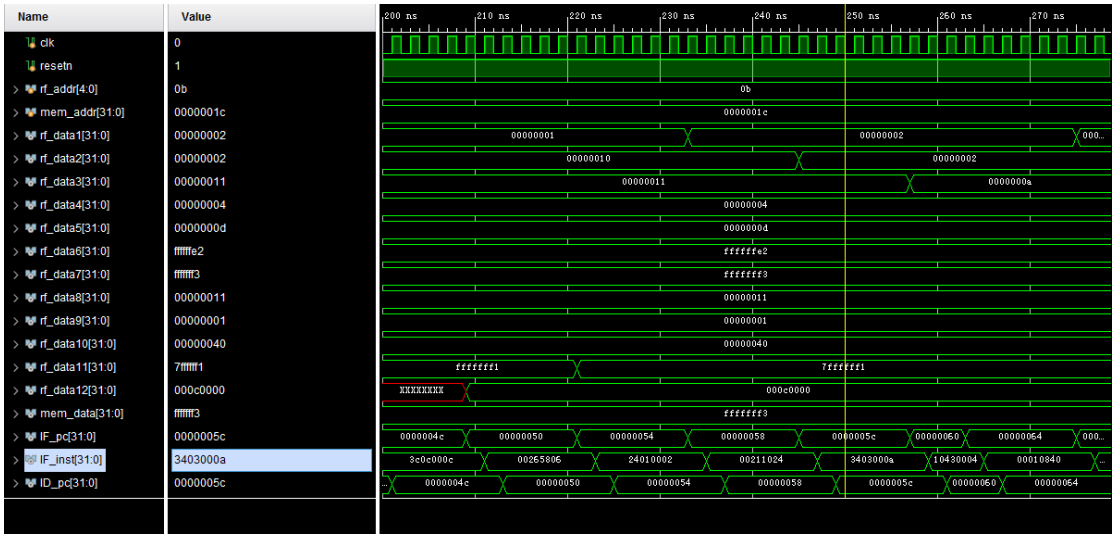
```
        #5;
      resetn = 1;
        // Add stimulus here
    end
    always #1 clk=~clk;
endmodule
```

7.14 multi_cycle_cpu.xdc

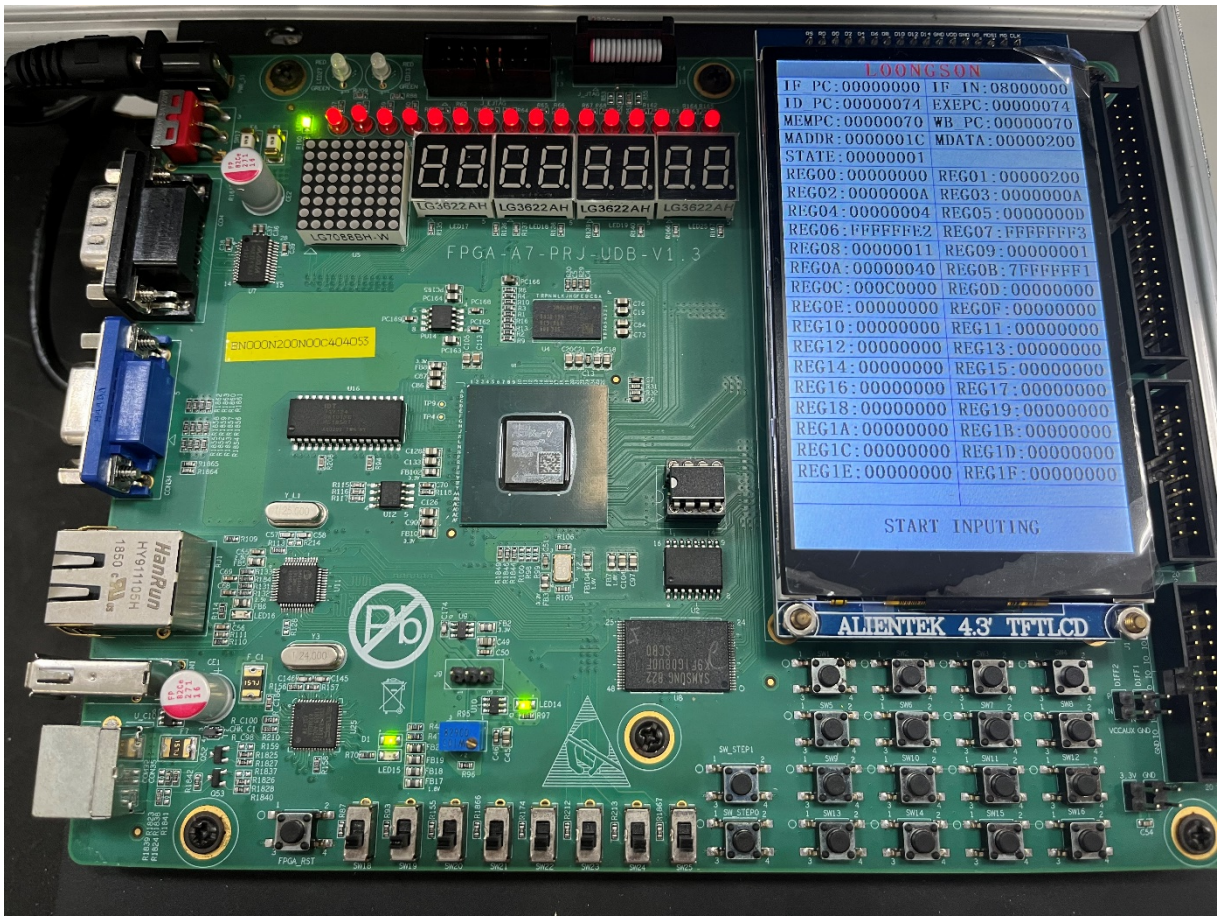
```
#时钟信号连接
set_property PACKAGE_PIN AC19 [get_ports clk]
#脉冲开关，用于输入作为复位信号，低电平有效
set_property PACKAGE_PIN Y3 [get_ports resetn]
#脉冲开关，用于输入作为单步执行的 clk
set_property PACKAGE_PIN Y5 [get_ports btn_clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports resetn]
set_property IOSTANDARD LVCMOS33 [get_ports btn_clk]
#触摸屏引脚连接
set_property PACKAGE_PIN J25 [get_ports lcd_rst]
set_property PACKAGE_PIN H18 [get_ports lcd_cs]
set_property PACKAGE_PIN K16 [get_ports lcd_rs]
set_property PACKAGE_PIN L8 [get_ports lcd_wr]
set_property PACKAGE_PIN K8 [get_ports lcd_rd]
set_property PACKAGE_PIN J15 [get_ports lcd_bl_ctr]
set_property PACKAGE_PIN H9 [get_ports {lcd_data_io[0]}]
set_property PACKAGE_PIN K17 [get_ports {lcd_data_io[1]}]
set_property PACKAGE_PIN J20 [get_ports {lcd_data_io[2]}]
set_property PACKAGE_PIN M17 [get_ports {lcd_data_io[3]}]
set_property PACKAGE_PIN L17 [get_ports {lcd_data_io[4]}]
set_property PACKAGE_PIN L18 [get_ports {lcd_data_io[5]}]
set_property PACKAGE_PIN L15 [get_ports {lcd_data_io[6]}]
set_property PACKAGE_PIN M15 [get_ports {lcd_data_io[7]}]
set_property PACKAGE_PIN M16 [get_ports {lcd_data_io[8]}]
set_property PACKAGE_PIN L14 [get_ports {lcd_data_io[9]}]
set_property PACKAGE_PIN M14 [get_ports {lcd_data_io[10]}]
set_property PACKAGE_PIN F22 [get_ports {lcd_data_io[11]}]
set_property PACKAGE_PIN G22 [get_ports {lcd_data_io[12]}]
set_property PACKAGE_PIN G21 [get_ports {lcd_data_io[13]}]
set_property PACKAGE_PIN H24 [get_ports {lcd_data_io[14]}]
set_property PACKAGE_PIN J16 [get_ports {lcd_data_io[15]}]
set_property PACKAGE_PIN L19 [get_ports ct_int]
set_property PACKAGE_PIN J24 [get_ports ct_sda]
set_property PACKAGE_PIN H21 [get_ports ct_scl]
set_property PACKAGE_PIN G24 [get_ports ct_rstn]
set_property IOSTANDARD LVCMOS33 [get_ports lcd_rst]
set_property IOSTANDARD LVCMOS33 [get_ports lcd_cs]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports lcd_rs]
set_property IOSTANDARD LVCMOS33 [get_ports lcd_wr]
set_property IOSTANDARD LVCMOS33 [get_ports lcd_rd]
set_property IOSTANDARD LVCMOS33 [get_ports lcd_bl_ctr]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[11]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[12]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[13]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[14]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[15]}]
set_property IOSTANDARD LVCMOS33 [get_ports ct_int]
set_property IOSTANDARD LVCMOS33 [get_ports ct_sda]
set_property IOSTANDARD LVCMOS33 [get_ports ct_scl]
set_property IOSTANDARD LVCMOS33 [get_ports ct_rstn]
set_property IOSTANDARD LVCMOS33 [get_ports btn_clk]
```

8 仿真波形图



9 上板验证图



上图可显示出，验证程序完整执行一个循环后各个寄存器的值。

[$1]=0000_0200H
[$2]=0000_000AH
[$3]=0000_000AH
[$4]=0000_0004H
[$5]=0000_000DH
[$6]=FFFF_FFE2H
[$7]=FFFF_FFF3H
[$8]=0000_0011H
[$9]=0000_0001H
[$10]=0000_0040H
[$11]=7FFF_FFF1H
[$12]=000C_0000H
Mem[0000_001C]=0200_0000H
由此可见，与验证程序计算结果相同。

10 心得体会

在本次课程实验中，代码编写的过程中难点在于控制单元，如何将一个指令分成多个周期执行，并且保证多周期执行也不会影响指令执行的正确性，设置一个通用的状态机是比较复杂的阶段。还要多处注意时钟沿的情况和寄存器读写的设置，多个繁琐的过程稍微出错就可能导致波形图和理想状态中的巨大差异。同时，需要设计同步的指令 rom、数据 ram,以及设计 mips 程序代码并将其转换为 16 进制，为此，我特意写了一个 python 脚本，将 mips 代码转换为 16 进制，节省了较多时间。

根据生成的波形图进行验证，经过多次调试，发现了许多个 bug，又经过多次修改之后，才完成了现在的作品。与软件课程设计不同，硬件课程设计需要更好的实验环境，也会更加有趣，当处于实验室中时，感觉自己的专注力显著提高了。

在这次硬件课程设计 1 中，我学会了很多，对 CPU 的原理也有了更深的认识，更是为了以后接触 PLC 设备奠定了基础。