东北大学秦皇岛分校

计算机与通信工程学院

计算机组成原理课程设计

学院	计算机通信工程学院
专业名称	计算机科学与技术
班级	1701 班
班级序号	24 号
学号	20178859
学生姓名	**朱子权**
指导教师	张冬丽
设计时间	2019.12.16~2019.12.27

课程设计任务书

专业班级：计科 1701 **学号：**20178859 **学生姓名：**

设计题目：MIPS32 五级流水及中断控制器设计。

一、设计实验条件

计算机组成原理实验室

二、设计任务及要求

1. 设计并实现指令 SUBU rd. rs, rt;
2. 32 位 ALU;
3. RS 触发器;

三、设计报告的内容

1. 前言

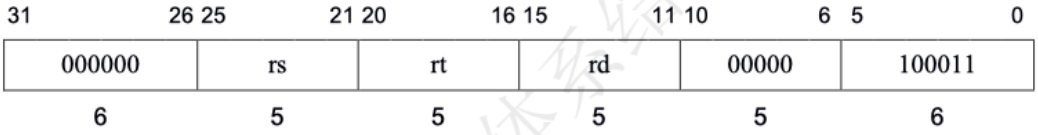
MIPS 架构 20 多年前由斯坦福大学开发，是一种简洁、优化、具有高度扩展性的 RISC 架构。它的基本特点是：包含大量的寄存器、指令数和字符、可视的管道延时时隙，这些特性使 MIPS 架构能够提供最高的每平方毫米性能和当今 SoC 设计中最低的能耗。

我们知道，RISC 的一大特点是：大量使用寄存器。这是因为寄存器的存取可以在一个时钟周期内完成，同时使用寄存器也大大简化了寻址方式。与其相应的，MIPS32 的指令中除加载（LDA）和存储（STO）指令外，都是使用寄存器或立即数作为操作数的。MIPS32 中的寄存器分为两类：通用寄存器（General Purpose Register, GPR）、特殊寄存器。

MIPS32 架构定义了 32 个通用寄存器，使用 $0、$1…$31 表示，都是 32 位。其中 $0 般用做常量 0。

2. 系统设计

2.1 五级流水 CPU 设计



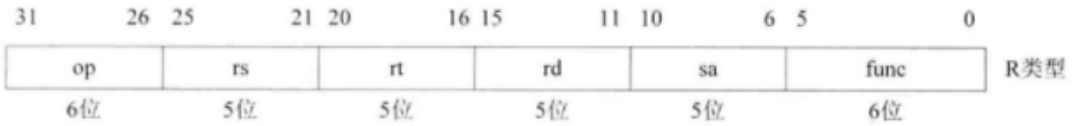
汇编格式：SUBU rd, rs, rt

功能描述：将寄存器 rs 的值与寄存器 rt 的值相减，结果写入 rd 寄存器中。

操作定义：GPR[rd] ← GPR[rs] – GPR[rt]

例 外：无

在这里，我们要实现的是 SUBU 指令的五级流水设计。其中，SUBU 指令的特点是：具体操作由 op、func 结合指定，其中 op 决定 SUBU 是 R 型、I 型还是 J 型，rs 和 rt 是源寄存器的编号，rd 是目的寄存器的编号，MIPS32 架构中有 32 个通用寄存器，使用 5 位编码就可以全部表示，所以 rs、rt、rd 的宽度都是 5 位。sa 只有在移位指令中使用，用来指定移位位数，在这里我们可以直接忽略。



2.1.1 指令数据通路的建立

根据设计需求，我们可以建立指令数据通路如下。其中 PC 指向与执行的指令，然后每次执行完后，PC 自增 4（因为 MIPS 一条指令为 32 位，即 4 个字节）。然后对于指令，高 6 位是 opcode，用来控制指令的类型。25 到 21 位是 rs，指明第一个源操作数的寄存器。同样地，20 到 16 位是 rt，指明第二个源操作数的寄存器，15 到 11 位是 rd，指明目的源操作数的寄存器。低六位是 func，用来指明需要执行的操作。然后从源寄存器读出数据送入 ALU 进行运算，再送入目的寄存器。

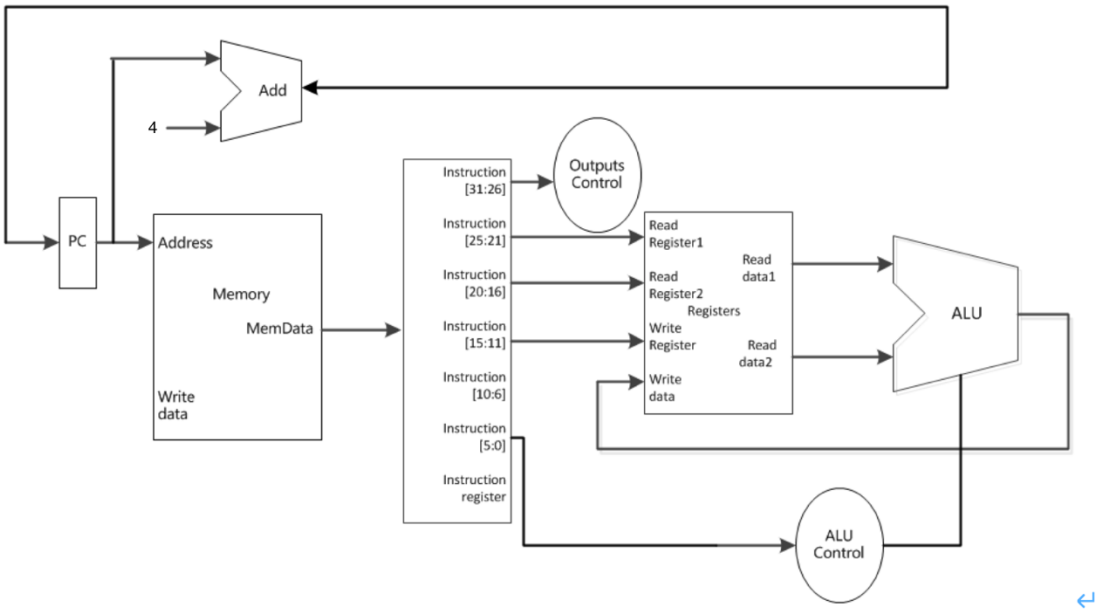


图 1 数据通路图

2.1.2 五级流水数据流图

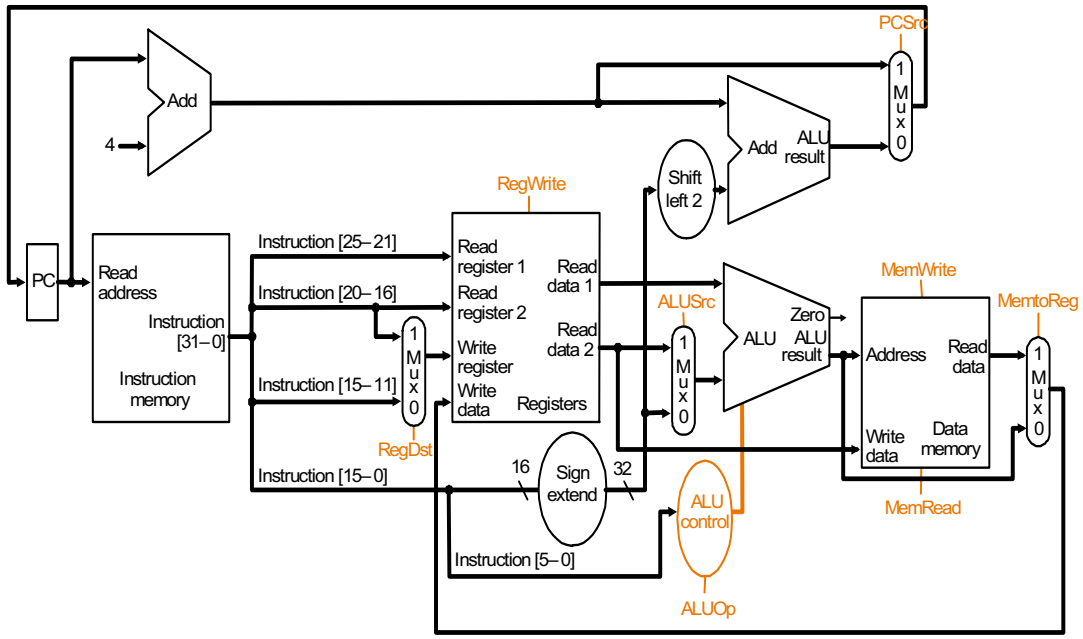


图 2 数据流图

2.1.3 五级流水系统结构图

可以看到，下面一共分为 5 个阶段，各个阶段完成的主要工作如下。

- 取指：取出指令存储器中的指令，PC 值递增，准备取下一条指令。
- 译码：对指令进行译码，依据译码结果 ，从 32 个通用寄存器中取出源操作数，有的指令要求两个源操作数都是寄存器的值。
- 执行阶段：依据译码阶段送入的源操作数、操作码，进行运算。
- 回写阶段：将运算结果保存到目的寄存器。

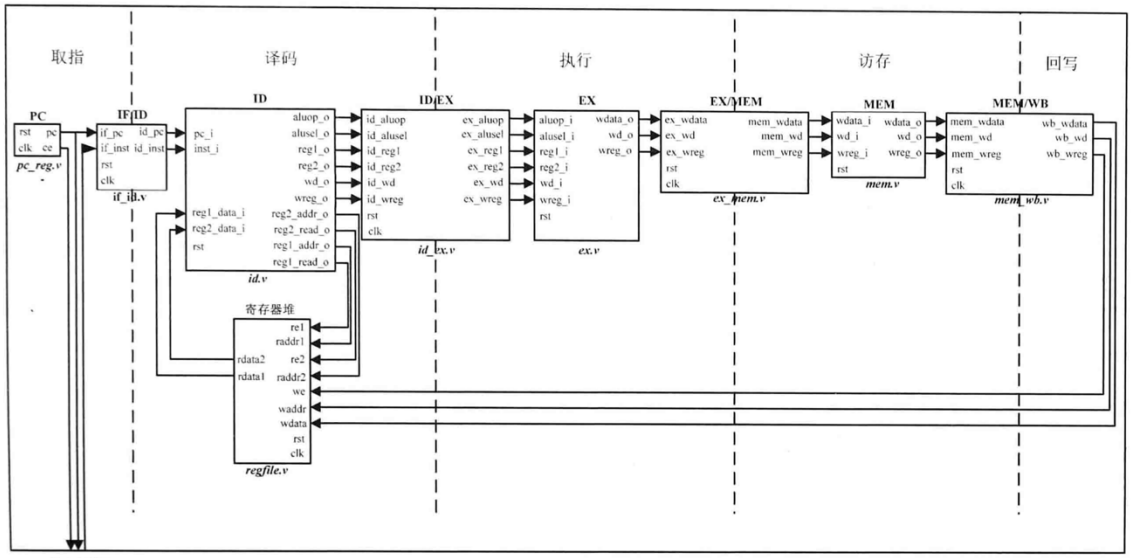


图 3 系统结构图

2.2　32 位 ALU

其中 ALU 的模块接口图如下，其中 A 和 W 是源操作数，Cin 是输入进位，S2 到 S0 代表的是所执行的运算，D 是运算结果，Cout 是输出进位。

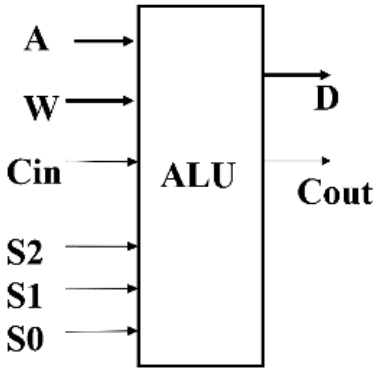


图 4 ALU 模块接口图

其中，S2-S0 对应的功能表如下：

S2	S1	S0	功能
0	0	0	A + W 加
0	0	1	A- W 减
0	1	0	A|W 或
0	1	1	A＆W 与
1	0	0	A + W + Cin 带进位加
1	0	1	A – W - Cin 带进位减
1	1	0	~A ， A 取反
1	1	1	A ， 输出 A

2.3　RS 触发器

下图是带时序信号 RS 触发器的原理图

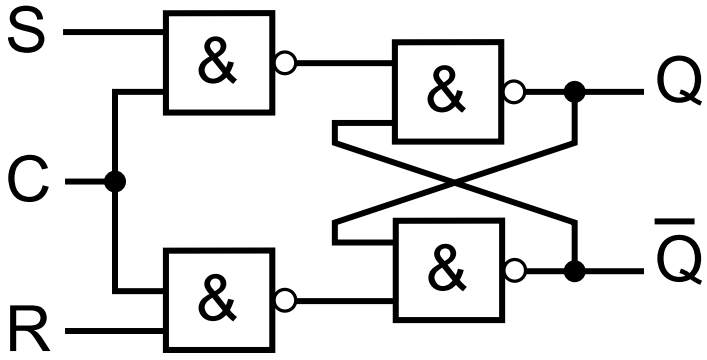


图 5 RS 触发器原理图

根据原理图，我们可以得到 RS 触发器的真值表。

RD	SD	Q	y
0	1	0	1
1	0	1	0
0	0	不定	
1	1	不变	

根据真值表，可以得到时序逻辑电路的特征方程为：

$$Q_{n+1} = S + \bar{R} \cdot Q_n$$

3. 系统实现

一、五级流水 CPU 实现，并实现指令 SUBU rd. rs, rt。

1.模块接口描述。

PC 模块

PC 模块的作用是给出指令地址。使用的是时序电路。接口描述如下。

序 号	接口名	宽度(bit)	输入/输出	作 用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	pc	32	输出	要读取的指令地址
4	ce	1	输出	指令存储器使能信号

IF/ID 模块

IF/ID 模块的作用是暂时保存取指阶段取得的指令，以及对应的指令地址，并在下一个时钟传递到译码阶段。使用的是时序电路。其接口描述如表所示。

序 号	接口名	宽度(bit)	输入/输出	作 用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	lf_pc	32	输入	取指阶段取得的指令对应的地址
4	if_inst	32	输入	取指阶段取得的指令
5	id_pc	32	输出	译码阶段的指令对应的地址
6	ld_inst	32	输出	译码阶段的指令

Regfile 模块

Regfile 模块实现了 32 个 32 位通用整数寄存器，可以同时进行两个寄存器的读操作和个寄存器的写操作。使用的是时序电路。其接口描述如表所示。

序 号	接口名	宽度(bit)	输入/输出	作 用
1	rst	1	输入	复位信号,商电平有效
2	clk	1	输入	时钟信号
3	waddr	5	输入	要写入的寄存器地址
4	wdata	32	输入	要写入的数据
5	we	1	输入	写使能信号
6	raddr1	5	输入	第一个读寄存器端口要读取的寄存器的地址
7	re1	1	输入	第一个读寄存器端口读使能信号
8	rdata1	32	输出	第一个读寄存器端口输出的寄存器值
]9	raddr2	5	输入	第二个读寄存器端口要读取的寄存器的地址
10	re2	1	输入	第二个读寄存器端口读使能信号
11	rdata2	32	输出	第二个读寄存器端口输出的寄存器值

ID 模块

ID 模块的作用是对指令进行译码，得到最终运算的类型、子类型、源操作数 1、源操作数 2、要写入的目的寄存器地址等信息，运算等。使用的是组合电路。ID 模块的接口描述如表所示。

序 号	接口名	宽度(bit)	输入/输出	作 用
1	rst	1	输入	复位信号
2	pc_i	32	输入	译码阶段的指令对应的地址
3	inst_i	32	输入	译码阶段的指令
4	reg1_data_i	32	输入	从 Regfile 输入的第一个读寄存器端口的输入
5	reg2_data_i	32	输入	从 Regfile 输入的第二个读寄存器端口的输入
6	reg1_read_o	1	输出	Regfile 模块的第一个读寄存器端口的读使能信号
7	Reg2_read_o	1	输出	Regfile 模块的第二个读寄存器端口的读使能信号
8	reg1_ addr_o	5	输出	Regfile 模块的第一个读寄存器端口的读地址信号
9	reg2_ addr_o	5	输出	Regfile 模块的第二个读寄存器端口的读地址信号
10	aluop_o	8	输出	译码阶段的指令要进行的运算的子类型
11	alusel_o	3	输出	译码阶段的指令要进行的运算的类型
12	reg1_o	32	输出	译码阶段的指令要进行的运算的源操作数 1
13	reg2_o	32	输出	译码阶段的指令要进行的运算的源操作数 2
14	wd_o	5	输出	译码阶段的指令要写入的目的寄存器地址
15	wreg_o	1	输出	译码阶段的指令是否有要写入的目的寄存器

ID/EX 模块

ID 模块的输出连接到 ID/EX 模块，后者的作用是将译码阶段取得的运算类型、源操作数、要写的目的寄存器地址等结果，在下一个时钟传递到流水线执行阶段。使用的是时序电路。其接口描述如表所示 。

序 号	接口名	宽度(bit)	输入输出	作 用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	id_alusel	3	输入	译码阶段的指令要进行的运算的类型
4	id_aluop	8	输入	译码阶段的指令要进行的运算的子类型
5	id_reg1	32	输入	译码阶段的指令要进行的运算的源操作数 1
6	id_reg2	32	输入	译码阶段的指令要进行的运算的源操作数 2
7	id_wd	5	输入	译码阶段的指令要写入的目的寄存器地址
8	id_wreg	1	输入	译码阶段的指令是否有要写入的目的寄存器
9	ex_alusel	3	输出	执行阶段的指令要进行的运算的类型
10	ex_aluop	8	输出	执行阶段的指令要进行的运算的子类型
11	ex_reg1	32	输出	执行阶段的指令要进行的运算的源操作数 1
12	ex_reg2	32	输出	执行阶段的指令要进行的运算的源操作数 2
13	ex_wd	5	输出	执行阶段的指令要写入的目的寄存器地址
14	ex_wreg	1	输出	执行阶段的指令是否有要写入的目的寄存器

EX 模块

EX 模块会从 ID/EX 模块得到运算类型 alusel_i、运算子类型 aluop_i、源操作数 reg1_i、源操作数 reg2_i、 要写的目的寄存器地址 wd_i。EX 模块会依据这些数据进行运算。使用的是组合电路。其接口描述如表所示。

序 号	接口名	宽度(bit)	输入/输出	作 用
1	rst	1	输入	复位信号
2	aluse1_i	3	输入	执行阶段要进行的运算的类型
3	aluop_i	8	输入	执行阶段要进行的运算的子类型
4	reg1_i	32	输入	参与运算的源操作数 1
5	reg2_i	32	输入	参与运算的源操作数 2
6	wd_i	5	输入	指令执行要写入的目的寄存器地址
7	wreg_i	1	输入	是否有要写入的目的寄存器
8	wd_o	5	输出	执行阶段的指令最终要写入的目的寄存器地址
9	wreg_o	1	输出	执行阶段的指令最终是否有要写入的目的寄存器
10	wdata_o	32	输出	执行阶段的指令最终要写入目的寄存器的值

EX/MEM 模块

EX 模块的输出连接到 EX/MEM 模块，后者的作用是将执行阶段取得的运算结果，在下一个时钟传递到流水线访存阶段。使用的是时序电路。其接口描述如表所示 。

序 号	接口名	宽度(bit)	输入/输出	作 用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	ex_wd	5	输入	执行阶段的指令执行后要写入的目的寄存器地址
4	ex_wreg	1	输入	执行阶段的指令执行后是否有要写入的目的寄存器
5	ex_wdata	32	输入	执行阶段的指令执行后要写入目的寄存器的值
6	mem_wd	5	输出	访存阶段的指令要写入的目的寄存器地址
7	mem_wreg	1	输出	访存阶段的指令是否有要写入的目的寄存器
8	mem_wdata	32	输出	访存阶段的指令要写入目的寄存器的值

MEM 模块

MEM 模块实现了内存，可以进行通过接口进行读写。由于这里我们没有访存的指令，所以我们只有一个组合逻辑电路，将输入的执行阶段的结果直接作为输出。使用的是组合电路。接口描述如下。

序 号	接口名	宽度(bit)	输入/输出	作 用
1	rst	1	输入	复位信号
2	wd_i	5	输入	访存阶段的指令要写入的目的寄存器地址
3	wreg_i	1	输入	访存阶段的指令是否有要写入的目的寄存器
4	wdata_i	32	输入	访存阶段的指令要写入目的寄存器的值
5	wd_o	5	输出	访存阶段的指令最终要写入的目的寄存器地址
6	wreg_o	1	输出	访存阶段的指令最终是否有要写入的目的寄存器
7	wdata	32	输出	访存阶段的指令最终要写入目的寄存器的值

MEM/WB 模块

MEM/WB 模块的作用是将访存阶段的运算结果，在下一个时钟传递到回写阶段。使用的是时序电路。接口描述如下。

序 号	接口名	宽度(bit)	输入/输出	作 用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	mem_wd	5	输入	访存阶段的指令最终要写入的目的寄存器地址
4	mem_wreg	1	输入	访存阶段的指令最终是否有要写入的目的寄存器
5	mem_wdata	32	输入	访存阶段的指令最终要写入目的寄存器的值
6	wb_wd	5	输出	回?阶段的指令要写入的目的寄存器地址
7	wb_wreg	1	输出	回写阶段的指令是否有要写入的目的寄存器
8	wb_wdata	32	输出	回写阶段的指令要写入目的寄存器的值

2.RTL ANALYSIS 生成图如下（大图见附页）

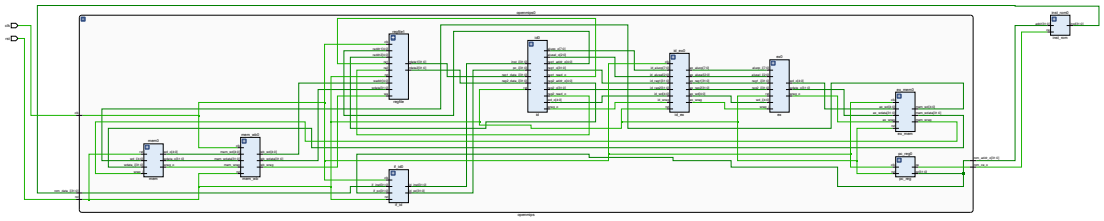


图 6　SUBU 指令的五级流水的 RTL ANALYSIS 生成图

二、实现一个 32 位的 ALU

ALU 模块

算术逻辑单元。使用的是时序电路。接口描述如下。

序 号	接口名	宽度(bit)	输入/输出	作 用
1	clk	1	输入	时钟信号
2	a	32	输入	源操作数 1
3	w	32	输入	源操作数 2
4	cin	32	输入	输入的进位
5	s	3	输入	对源操作数所执行的操作的操作码
6	d	32	输出	目的操作数（结果）
7	cout	1	输出	输出结果的进位

ADD/ADC/AND/NOT/OR/PRINT/SUB/SBB 模块

进行算术运算或者运算。使用的是组合电路。接口描述如下。

序 号	接口名	宽度(bit)	输入/输出	作 用
1	a	32	输入	源操作数 1
2	w	32	输入	源操作数 2
3	cin	32	输入	输入的进位

4	d	32	输出	目的操作数（结果）
5	cout	1	输出	输出结果的进位

OUT 模块

用于根据 S 的值，进行 ADD/ADC/AND/NOT/OR/PRINT/SUB/SBB 模块的结果的选择。使用的是组合电路。

序 号	接口名	宽度(bit)	输入/输出	作 用
1~8	output_cout<x>	1	输入	用于接收来自每个模块的输出的进位
9-16	output_d<x>	1	输入	用于接收来自每个模块的输出的结果
17	s	3	输入	用于选择输出的进位和结果
18	d	32	输出	目的操作数（结果）
19	cout	1	输出	输出结果的进位

2.RTL ANALYSIS 生成图如下

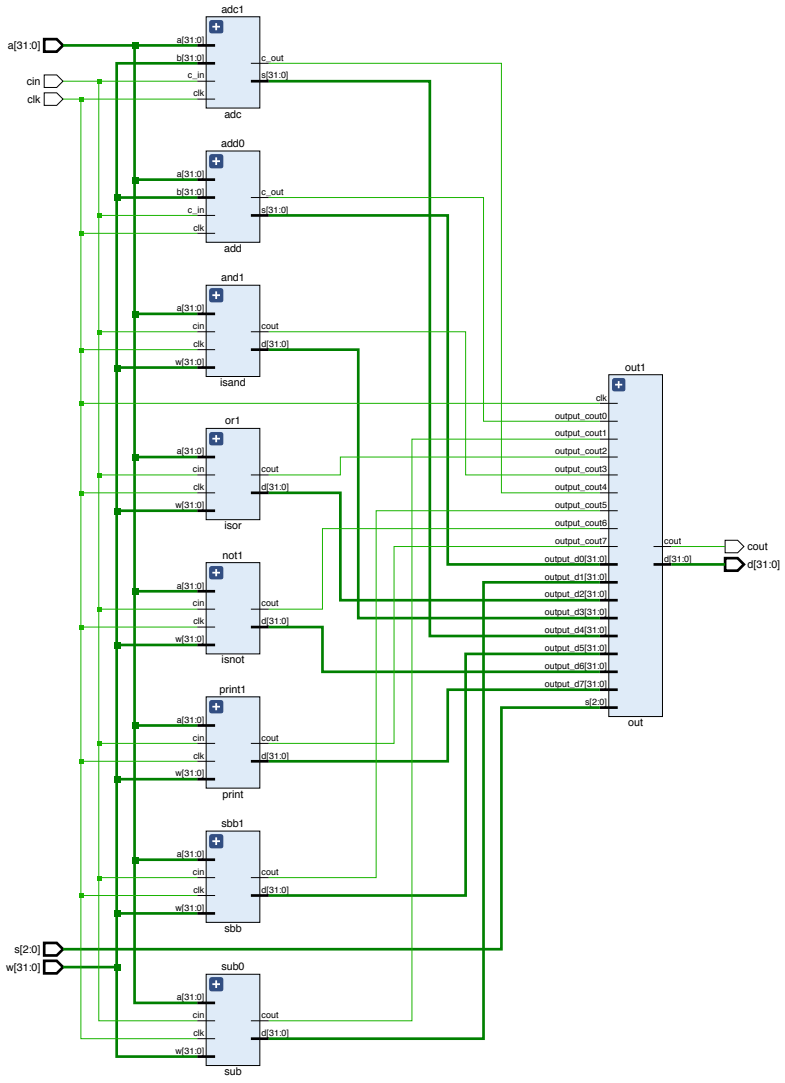


图 7 ALU 的 RTL ANALYSIS 生成图

三、实现 RS 触发器

1.模块接口描述。

RS 模块

实现了一个 RS 触发器，输入端分别为设置和清零端。使用的是时序电路。

序 号	接口名	宽度(bit)	输入/输出	作 用
1	clk	1	输入	时钟信号
2	s	1	输入	置位端
3	r	1	输入	清零端
4	q	1	输出	输出结果
5	qb	1	输出	q 的取反

ISNAND 模块

实现了一个 NAND 门，输入端分别为设置和清零端。使用的是组合电路。

序 号	接口名	宽度(bit)	输入/输出	作 用
1	a	1	输入	需要进行与或的操作数 1
2	b	1	输入	需要进行与或的操作数 2
3	c	1	输出	进行与或的结果

2.RTL ANALYSIS 生成图如下

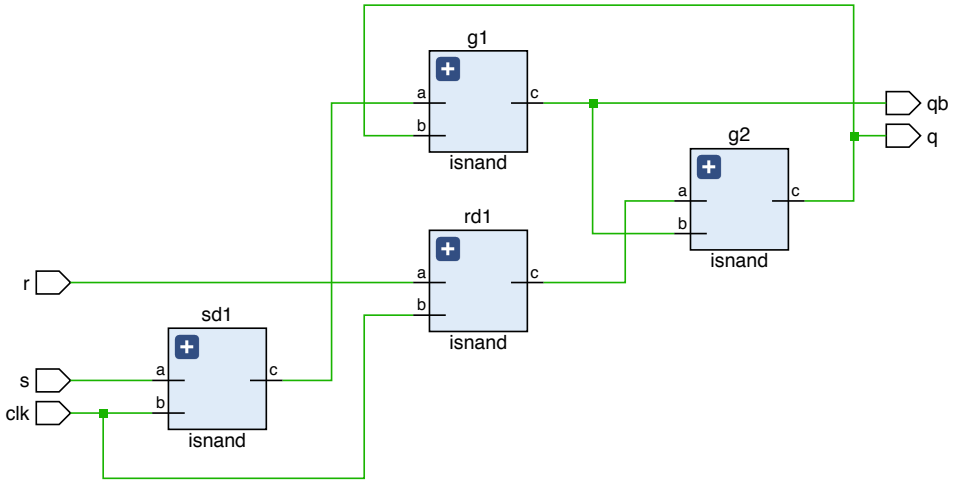


图 8 RS 触发器的 RTL ANALYSIS 生成图

4. 系统测试

一、五级流水 CPU 测试

1.测试的核心代码如下

```
module openmips_min_sopc_tb();

    reg     CLOCK_50;
    reg     rst;
```

```
  initial begin
    CLOCK_50 = 1'b0;
    forever #10 CLOCK_50 = ~CLOCK_50;
  end

  initial begin
    rst = `RstEnable;
    #195 rst= `RstDisable;
    #1000 $stop;
  end

  openmips_min_sopc openmips_min_sopc0(
     .clk(CLOCK_50),
     .rst(rst)
  );

endmodule
```

其中测试指令 .data 文件的数据如下，下面的每条指令分别代表其中代表寄存器 1 的值减去寄存器 2i 的值，然后回写到寄存器 2i+1（其中 i 的取值范围是 1 到 10）

inst_rom.data

```
1   00221823
2   00242823
3   00263823
4   00284823
5   002a5823
6   002c6823
7   002e7823
8   00308823
9   00329823
10  0034a823
```

图 9 测试指令 inst_rom.data

其中我们预定义寄存器的值始终如下：

```
always @ (posedge clk) begin
    regs[1] = 32'h11111111;
    regs[2] = 32'h10001000;
    regs[4] = 32'h00100010;
    regs[6] = 32'h01001000;
    regs[8] = 32'h11101110;
    regs[10] = 32'h10101010;
    regs[12] = 32'h01101110;
    regs[14] = 32'h11000110;
    regs[16] = 32'h10100000;
    regs[18] = 32'h00000fff;
    regs[20] = 32'h0fffffff;
end
```

图 10 定义寄存器的值

2. 测试的结果如下

其中前两排是源操作数的寄存器地址，第三排是目的操作数的寄存器的地址，注意，指令的目的操作数比源操作数晚三个时钟周期，这是因为在进行回写前还需要经过译码，执行，访存回写三个阶段。同理，下面三排是两个源操作数的值，和目的操作数的值。

图 11 SUBU 测试结果波形图

根据波形图，可以知道我们设计的 SUBU 指令的五级流水是正确的。

二、32 位 ALU 测试

1. 测试的核心代码如下

我们的两个源操作数 a 和 w 始终为 hfffffff0 和 h000000f0，输入的进位始终为 1，然后只改变 s 的值进行测试。

```
module alu_tb();

    reg     CLOCK_50;
    reg     rst;

    wire[`WIDTH] a;
    wire[`WIDTH] w;
    wire c_in;
    reg[`SELECT_WIDTH] s;

    wire[`WIDTH] d;
    wire c_out;
```

```
    integer k = 3'b0;

    assign a = 32'hfffffff0;
    assign w = 32'h000000f0;
    assign c_in = 1'b1;

    always @(posedge CLOCK_50) begin
      for (k=3'b0; k<=3'b111; k=k+3'b1)
        #50 s <= k;
    end

    initial begin
      CLOCK_50 = 1'b0;
      forever #10 CLOCK_50 = ~CLOCK_50;
    end

    initial begin
      #1000 $stop;
    end

    alu alu0(
      .clk(CLOCK_50),
      .a(a), .w(w), .cin(c_in),
      .s(s), .d(d), .cout(c_out)
    );
endmodule
```

2.测试的结果如下

其中前两行是源操作数，在这个测试中始终为 hfffffff0 和 h000000f0，第三行是要执行的操作 s， 第四行是输入的进位，始终为 1，第五行是进行运算后的结果，最后一行是输出的进位。

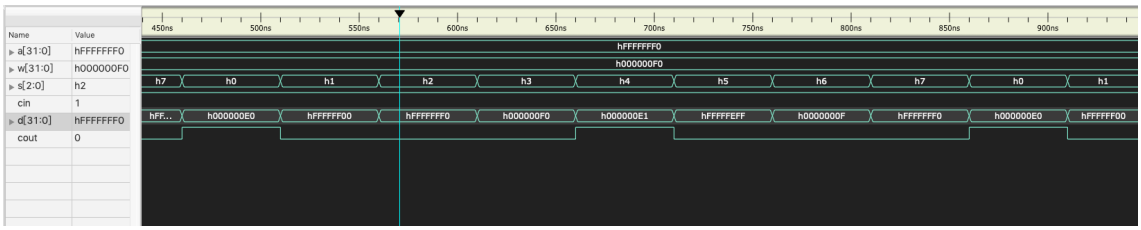


图 12 ALU 测试波形图

h0 到 h7 分别是，加法，减法，或，与，带进位的加法和减法，A 取反，以及直接输出 A。比较结果，发现结果没有错误。根据波形图，可以知道我们设计的 ALU 是正确的。

三、RS 触发器测试

1.测试的核心代码如下

```
  module rs_tb();
```

```
    reg     CLOCK_50;

    reg s;
    reg r;
    wire q;
    wire qb;

    integer high = 1'b1;
    integer low = 1'b0;

    initial @(posedge CLOCK_50) begin
      #10 s <= low; r <= high;
      #10 s <= low; r <= low;
      #10 s <= high; r <= low;
      #10 s <= high; r <= high;

      #10 s <= low; r <= high;
      #10 s <= low; r <= low;
      #10 s <= high; r <= low;
      #10 s <= high; r <= high;

      #10 s <= low; r <= high;
      #10 s <= high; r <= high;

      #10 s <= low; r <= low;
      #10 s <= low; r <= low;
    end

    initial begin
      CLOCK_50 = 1'b0;
      forever #10 CLOCK_50 = ~CLOCK_50;
    end

    initial begin
      #130 $stop;
    end

    rs rs0(
      CLOCK_50, s, r, q, qb
    );
  endmodule
```

2.测试的结果如下

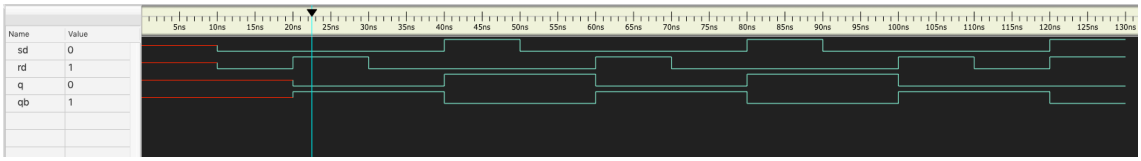


图 13 RS 触发器波形图

与真值表比较可以知道，当 SD 为 1 的时候，RD 为 0，Q 置 1；SD 为 0 的时候，RD 为 1，Q 置 0；SD，RD 都为 1 的时候，Q 置保持。QB 始终与 SD 相反。

RD	SD	Q	y
0	1	0	1
1	0	1	0
0	0	不定	
1	1	不变	

5. 总结

这次课程设计我完成了指令的五级流水，ALU 的设计和 RS 触发器的设计，在这次设计中主要遇到了以下问题，以及我的解决方案。

i. 关于 wire 在过程中利用 <= 赋值报错问题。后来改成 reg 即解决了问题。

ii. 关于 ALU 中加法进位的问题。Verilog 的 + 运算符并不会有返回进位，需要通过自己实现一个进位加法器。这里我实现了一个串行进位加法器。

iii. RS 触发器的结果不符合预期。后来在没必要的部件去除了时钟信号 clk，即解决了问题。

iv. macOS 上无法使用 vivado。使用的是 iverlog 指令 + Scansion 的解决方案。

后续，我打算完善指令的五级流水，加入更多的算术运算和逻辑指令，同时还可以增加流水深度。

6. 心得体会

通过一周的计算机组成原理课程设计，我自豪地说我设计一个简单有五级流水的 MIPS 的 CPU，这是一次有趣而又收益匪浅的经历。

在刚开始做课设的时候，其实我是有些畏惧的，因为以前没有接触过这种计算机硬件设计，还有一个原因是当时数电课设的时候连线连到自闭，然后对硬件设计总有种灰心。但不过，通过学习老师推荐的雷思磊大神的《自己动手写 CPU》，我很快地上手了 Verilog ，然后模仿书本实现了一个简单的五级流水的 CPU。虽然只实现了一条 SUBU 指令， 但不过我对整个五级流水线，以及如何实现一个 CPU 有了一个较为深入的了解，相信即使是其它的指令，我也更有信心能够实现。

虽然其实这些东西在数电和计算机组成课程中我们都是学过的，但是动手起来根本不是一回事。有时候出了 bug，就要对着波形图和代码一遍一遍找问题的所在，虽然看的眼睛都花了，但是还是还是很开心的，尤其是解决 bug 的时候，很有成就感的。

通过设计一个简单有五级流水的 MIPS 的 CPU，不仅使我了解了 RISC 和 MIPS 的特点和设计哲学，同时也是对我综合能力的一个提升，也是对数电和计组课程的温习。

这次计算机组成课程设计，张旭老师以及所有辛勤付出的老师，感谢你们用知识和耐心帮助我克服各种困难完成本次课程设计。

7. 参考文献

[1] 唐硕飞.计算机组成原理：高等教育出版社，2008-1.
[2] 雷思磊.自己动手写 CPU：电子工业出版社，2014-9.
[3] 张冬.大话计算机：清华大学出版社，2019-5-1.
[4] 阎石.数字电子技术基础：清华大学出版社，2006.
[5] 夏宇闻. Verilog 数字系统设计教程：北京航空航天大学出版社，2003-7.
[6] M.S. Schmalz. Organization of Computer Systems:§4: Processors：https://www.cise.ufl.edu/~mssz/CompOrg/CDA-proc.html，2001-4.

8. 附录

./Code-ALU/adc.v

```
module adc_1(
  input wire a,
  input wire b,
  input wire c,

  output wire s,
  output wire c_out
  );

  assign s = a^b^c;
        assign c_out = (a&b)|(a&c)|(b&c);

endmodule

module adc_2(
  input wire[1:0] a,
  input wire[1:0] b,
  input wire c_in,

  output wire[1:0] s,
  output wire c_out
);

  adc_1 a0(a[0], b[0], c_in, s[0], c_tmp);
  adc_1 a1(a[1], b[1], c_tmp, s[1], c_out);
endmodule
```

```
module adc_4(
  input wire[3:0] a,
  input wire[3:0] b,
  input wire c_in,

  output wire[3:0] s,
  output wire c_out
);

  adc_2 a0(a[1:0], b[1:0], c_in, s[1:0], c_tmp);
  adc_2 a1(a[3:2], b[3:2], c_tmp, s[3:2], c_out);
endmodule

module adc_8(
  input wire[7:0] a,
  input wire[7:0] b,
  input wire c_in,

  output wire[7:0] s,
  output wire c_out
);

  adc_4 a0(a[3:0], b[3:0], c_in, s[3:0], c_tmp);
  adc_4 a1(a[7:4], b[7:4], c_tmp, s[7:4], c_out);
endmodule

module adc_16(
  input wire[15:0] a,
  input wire[15:0] b,
  input wire c_in,

  output wire[15:0] s,
  output wire c_out
);

  adc_8 a0(a[7:0], b[7:0], c_in, s[7:0], c_tmp);
  adc_8 a1(a[15:8], b[15:8], c_tmp, s[15:8], c_out);
endmodule

module adc(
  input wire clk,

  input wire[31:0] a,
  input wire[31:0] b,
```

```
  input wire c_in,

  output wire[31:0] s,
  output wire c_out
);

  adc_16 a0(a[15:0], b[15:0], c_in, s[15:0], c_tmp);
  adc_16 a1(a[31:16], b[31:16], c_tmp, s[31:16], c_out);

endmodule
```

./Code-ALU/add.v

```
module add_1(
  input wire a,
  input wire b,
  input wire c,

  output wire s,
  output wire c_out
  );

  assign s = a^b^c;
        assign c_out = (a&b)|(a&c)|(b&c);

endmodule

module add_2(
  input wire[1:0] a,
  input wire[1:0] b,
  input wire c_in,

  output wire[1:0] s,
  output wire c_out
);

  add_1 a0(a[0], b[0], c_in, s[0], c_tmp);
  add_1 a1(a[1], b[1], c_tmp, s[1], c_out);
endmodule

module add_4(
  input wire[3:0] a,
  input wire[3:0] b,
  input wire c_in,

  output wire[3:0] s,
```

```
  output wire c_out
);

  add_2 a0(a[1:0], b[1:0], c_in, s[1:0], c_tmp);
  add_2 a1(a[3:2], b[3:2], c_tmp, s[3:2], c_out);
endmodule

module add_8(
  input wire[7:0] a,
  input wire[7:0] b,
  input wire c_in,

  output wire[7:0] s,
  output wire c_out
);

  add_4 a0(a[3:0], b[3:0], c_in, s[3:0], c_tmp);
  add_4 a1(a[7:4], b[7:4], c_tmp, s[7:4], c_out);
endmodule

module add_16(
  input wire[15:0] a,
  input wire[15:0] b,
  input wire c_in,

  output wire[15:0] s,
  output wire c_out
);

  add_8 a0(a[7:0], b[7:0], c_in, s[7:0], c_tmp);
  add_8 a1(a[15:8], b[15:8], c_tmp, s[15:8], c_out);
endmodule

module add(
  input wire clk,

  input wire[31:0] a,
  input wire[31:0] b,
  input wire c_in,

  output wire[31:0] s,
  output wire c_out
);

  add_16 a0(a[15:0], b[15:0], 1'b0, s[15:0], c_tmp);
```

```
  add_16 a1(a[31:16], b[31:16], c_tmp, s[31:16], c_out);

endmodule
```

./Code-ALU/alu.v

```
`include "define.v"
`include "add.v"
`include "sub.v"
`include "or.v"
`include "and.v"
`include "adc.v"
`include "sbb.v"
`include "not.v"
`include "print.v"
`include "out.v"


module alu(
  input wire clk,

  input wire[`WIDTH] a,
  input wire[`WIDTH] w,
  input wire cin,
  input wire[`SELECT_WIDTH] s,

  output wire[`WIDTH] d,
  output wire cout
);

  wire[`WIDTH] output_d0, output_d1, output_d2, output_d3, output_d4,
output_d5, output_d6, output_d7;
  wire output_cout0, output_cout1, output_cout2, output_cout3, output_cout4,
output_cout5, output_cout6, output_cout7;

  add add0(clk, a, w, cin, output_d0, output_cout0);
  sub sub0(clk, a, w, cin, output_d1, output_cout1);
  isor or1(clk, a, w, cin, output_d2, output_cout2);
  isand and1(clk, a, w, cin, output_d3, output_cout3);
  adc adc1(clk, a, w, cin, output_d4, output_cout4);
  sbb sbb1(clk, a, w, cin, output_d5, output_cout5);
  isnot not1(clk, a, w, cin, output_d6, output_cout6);
  print print1(clk, a, w, cin, output_d7, output_cout7);
  out out1(clk,
          output_cout0, output_cout1, output_cout2, output_cout3,
          output_cout4, output_cout5, output_cout6, output_cout7,
```

```
        output_d0, output_d1, output_d2, output_d3,
        output_d4, output_d5, output_d6, output_d7,
        s, d, cout);

endmodule // alu
```

./Code-ALU/alu_tb.v

```
`include "define.v"
`include "alu.v"
`timescale 1ns/1ps

module alu_tb();

  reg     CLOCK_50;
  reg     rst;

  wire[`WIDTH] a;
  wire[`WIDTH] w;
  wire c_in;
  reg[`SELECT_WIDTH] s;

  wire[`WIDTH] d;
  wire c_out;

  integer k = 3'b0;

  assign a = 32'hfffffff0;
  assign w = 32'h000000f0;
  assign c_in = 1'b1;

  always @(posedge CLOCK_50) begin
    for (k=3'b0; k<=3'b111; k=k+3'b1)
      #50 s <= k;
  end

  initial begin

    $dumpfile("test.vcd");
    $dumpvars(0, alu_tb);

    CLOCK_50 = 1'b0;
    forever #10 CLOCK_50 = ~CLOCK_50;
  end

  initial begin
```

```
    #1000 $stop;
  end

  alu alu0(
    .clk(CLOCK_50),
    .a(a), .w(w), .cin(c_in),
    .s(s), .d(d), .cout(c_out)
  );
endmodule
```

./Code-ALU/and.v

```
`include "define.v"

module isand(
  input wire clk,

  input wire[`WIDTH] a,
  input wire[`WIDTH] w,
  input wire cin,

  output reg[`WIDTH] d,
  output reg cout
);

  always @(*) begin
    d <= a & w;
    cout <= 0;
  end

endmodule
```

./Code-ALU/define.v

```
//指令
`define ADD 3'b000
`define SUB 3'b001
`define OR 3'b010
`define AND 3'b011
`define ADC 3'b100
`define SBB 3'b101
`define NOT 3'b110
`define PRINT 3'b111

//参数
`define WIDTH 31:0
`define SELECT_WIDTH 2:0
```

```
`define RstEnable 1'b1   //复位有效
`define RstDisable 1'b0  //复位无效
```

./Code-ALU/not.v

```
`include "define.v"

module isnot(
  input wire clk,

  input wire[`WIDTH] a,
  input wire[`WIDTH] w,
  input wire cin,

  output reg[`WIDTH] d,
  output reg cout
);

  always @(*) begin
    d <= ~a;
    cout <= 0;
  end

endmodule
```

./Code-ALU/or.v

```
`include "define.v"

module isor(
  input wire clk,

  input wire[`WIDTH] a,
  input wire[`WIDTH] w,
  input wire cin,

  output reg[`WIDTH] d,
  output reg cout
);

  always @(*) begin
    d <= a | w;
    cout <= 0;
  end

endmodule
```

./Code-ALU/out.v

```
`include "define.v"

module out(
  input wire clk,
  input wire output_cout0,
  input wire output_cout1,
  input wire output_cout2,
  input wire output_cout3,
  input wire output_cout4,
  input wire output_cout5,
  input wire output_cout6,
  input wire output_cout7,
  input wire[`WIDTH] output_d0,
  input wire[`WIDTH] output_d1,
  input wire[`WIDTH] output_d2,
  input wire[`WIDTH] output_d3,
  input wire[`WIDTH] output_d4,
  input wire[`WIDTH] output_d5,
  input wire[`WIDTH] output_d6,
  input wire[`WIDTH] output_d7,

  input wire[`SELECT_WIDTH] s,

  output reg[`WIDTH] d,
  output reg cout
);

  always @(posedge clk) begin
    case (s)
      `ADD: begin
        d <= output_d0;
        cout <= output_cout0;
      end
      `SUB: begin
        d <= output_d1;
        cout <= output_cout1;
      end
      `OR: begin
        d <= output_d2;
        cout <= output_cout2;
      end
      `AND: begin
        d <= output_d3;
        cout <= output_cout3;
      end
```

```
      `ADC: begin
        d <= output_d4;
        cout <= output_cout4;
      end
      `SBB: begin
        d <= output_d5;
        cout <= output_cout5;
      end
      `NOT: begin
        d <= output_d6;
        cout <= output_cout6;
      end
      `PRINT: begin
        d <= output_d7;
        cout <= output_cout7;
      end
      default: begin
      end
    endcase
  end
endmodule
```

./Code-ALU/print.v

```
`include "define.v"

module print(
  input wire clk,

  input wire[`WIDTH] a,
  input wire[`WIDTH] w,
  input wire cin,

  output reg[`WIDTH] d,
  output reg cout
);

  always @(*) begin
    d <= a;
    cout <= 0;
  end

endmodule
```

./Code-ALU/sbb.v

```
`include "define.v"
```

```
module sbb(
  input wire clk,

  input wire[`WIDTH] a,
  input wire[`WIDTH] w,
  input wire cin,

  output reg[`WIDTH] d,
  output reg cout
);

  always @(*) begin
    d <= a - w - cin;
    cout <= 0;
  end

endmodule
```

./Code-ALU/sub.v

```
`include "define.v"

module sub(
  input wire clk,

  input wire[`WIDTH] a,
  input wire[`WIDTH] w,
  input wire cin,

  output reg[`WIDTH] d,
  output reg cout
);

  always @(*) begin
    d <= a - w;
    cout <= 0;
  end

endmodule
```

./Code-RS 触发器/nand.v

```
module isnand(
  input wire a,
  input wire b,
```

```
  output reg c
);

  always @(*) begin
    c <= ~(a | b);
  end

endmodule
```

./Code-RS 触发器/rs.v

```
`include "nand.v"

module rs(
  input wire clk,

  input wire r,
  input wire s,

  output wire q,
  output wire qb
);

  isnand sd1(s, clk, sd);
  isnand rd1(r, clk, rd);
  isnand g1(sd, q, qb);
  isnand g2(rd, qb, q);

endmodule
```

./Code-RS 触发器/rs_tb.v

```
`timescale 1ns/1ps
`include "rs.v"

module rs_tb();

  reg     CLOCK_50;

  reg s;
  reg r;
  wire q;
  wire qb;

  integer high = 1'b1;
  integer low = 1'b0;
```

```
  initial @(posedge CLOCK_50) begin
    #10 s <= low; r <= high;
    #10 s <= low; r <= low;
    #10 s <= high; r <= low;
    #10 s <= high; r <= high;

    #10 s <= low; r <= high;
    #10 s <= low; r <= low;
    #10 s <= high; r <= low;
    #10 s <= high; r <= high;

    #10 s <= low; r <= high;
    #10 s <= high; r <= high;

    #10 s <= low; r <= low;
    #10 s <= low; r <= low;
  end

  initial begin

    $dumpfile("test.vcd");
    $dumpvars(0, rs_tb);

    CLOCK_50 = 1'b0;
    forever #10 CLOCK_50 = ~CLOCK_50;
  end

  initial begin
    #130 $stop;
  end

  rs rs0(
    CLOCK_50, s, r, q, qb
  );
endmodule
```

./Code-SUB 指令/defines.v

```
//全局
`define RstEnable 1'b1   //复位有效
`define RstDisable 1'b0  //复位无效
`define ZeroWord 32'h00000000    //32 位的数值 0
`define WriteEnable 1'b1 //使能写
`define WriteDisable 1'b0        //禁止写
`define ReadEnable 1'b1  //使能读
`define ReadDisable 1'b0 //禁止读
```

```
`define AluOpBus 7:0     //译码输出运算子类型的长度
`define AluSelBus 2:0    //译码输出运算类型的长度
`define InstValid 1'b0   //指令有效
`define InstInvalid 1'b1 //指令无效
`define Stop 1'b1
`define NoStop 1'b0
`define InDelaySlot 1'b1
`define NotInDelaySlot 1'b0
`define Branch 1'b1
`define NotBranch 1'b0
`define InterruptAssert 1'b1
`define InterruptNotAssert 1'b0
`define TrapAssert 1'b1
`define TrapNotAssert 1'b0
`define True_v 1'b1      //逻辑"真"
`define False_v 1'b0     //逻辑"假"
`define ChipEnable 1'b1  //芯片使能
`define ChipDisable 1'b0 //芯片禁止


//指令
//`define EXE_ORI 6'b001101      //指令 ori 的指令码
`define EXE_SUBU  6'b100011
`define EXE_NOP 6'b000000
`define EXE_SPECIAL_INST 6'b000000


//AluOp
// `define EXE_OR_OP   8'b00100101
// `define EXE_ORI_OP  8'b01011010
`define EXE_SUBU_OP  8'b00010001


`define EXE_NOP_OP    8'b00000000

//AluSel
// `define EXE_RES_LOGIC 3'b001
`define EXE_RES_ARITHMETIC 3'b100
`define EXE_RES_NOP 3'b000


//指令存储器inst_rom
`define InstAddrBus 31:0 //ROM 的地址总线宽度
`define InstBus 31:0     //ROM 的数据总线宽度
```

```
`define InstMemNum 131071          //ROM 实际大小
`define InstMemNumLog2 17          //ROM 实际使用的地址总线


//通用寄存器regfile
`define RegAddrBus 4:0    //寄存器地址线宽度
`define RegBus 31:0       //寄存器数据宽度
`define RegWidth 32       //寄存器的宽度
`define DoubleRegWidth 64
`define DoubleRegBus 63:0
`define RegNum 32
`define RegNumLog2 5      //寻址寄存器位数
`define NOPRegAddr 5'b00000
```

./Code-SUB 指令/ex.v

```
`include "defines.v"

module ex(

        input wire
                        rst,

        //送到执行阶段的信息
        input wire[`AluOpBus]         aluop_i,
        input wire[`AluSelBus]        alusel_i,
        input wire[`RegBus]           reg1_i,
        input wire[`RegBus]           reg2_i,
        input wire[`RegAddrBus]       wd_i,
        input wire                    wreg_i,


        output reg[`RegAddrBus]       wd_o,
        output reg                    wreg_o,
        output reg[`RegBus]
        wdata_o

);

  //保存逻辑运算的结果
  reg[`RegBus] arithmeticout;

  //依据 aluop_i 指示的运算子类型进行运算(这里只实现了SUBU)
  always @ (*) begin
    if(rst == `RstEnable) begin
      arithmeticout <= `ZeroWord;
```

```
    end else begin
      case (aluop_i)
        `EXE_SUBU_OP:     begin
          //进行减法运算
          arithmeticout <= reg1_i - reg2_i;
        end
        default: begin
          arithmeticout <= `ZeroWord;
        end
      endcase
    end    //if
  end      //always

  //依据 alusel_i，选择一个运算结果作为最终结果（这里只有算术运算）
  always @ (*) begin
    wd_o <= wd_i;
    wreg_o <= wreg_i;
    case ( alusel_i )
      `EXE_RES_ARITHMETIC:        begin
        //把逻辑运算结果输出
        wdata_o <= arithmeticout;
      end
      default:    begin
        wdata_o <= `ZeroWord;
      end
    endcase
  end
endmodule
```

./Code-SUB 指令/ex_mem.v

```
`include "defines.v"

module ex_mem(

        input    wire
                        clk,
        input wire
                        rst,


        //来自执行阶段的信息
        input wire[`RegAddrBus]       ex_wd,
        input wire                    ex_wreg,
        input wire[`RegBus]                                     ex_wdata,
```

```
        //送到访存阶段的信息
        output reg[`RegAddrBus]      mem_wd,
        output reg                   mem_wreg,
        output reg[`RegBus]                                    mem_wdata


);


        always @ (posedge clk) begin
                if(rst == `RstEnable) begin
                        mem_wd <= `NOPRegAddr;
                        mem_wreg <= `WriteDisable;
                  mem_wdata <= `ZeroWord;
                end else begin
                        mem_wd <= ex_wd;
                        mem_wreg <= ex_wreg;
                        mem_wdata <= ex_wdata;
                end    //if
        end      //always


endmodule
```

./Code-SUB 指令/id.v

```
`include "defines.v"

module id(

        input wire
                        rst,
        input wire[`InstAddrBus]                pc_i,
        input wire[`InstBus]         inst_i,

        input wire[`RegBus]          reg1_data_i,
        input wire[`RegBus]          reg2_data_i,

        //送到regfile的信息
        output reg                   reg1_read_o,
        output reg                   reg2_read_o,
        output reg[`RegAddrBus]      reg1_addr_o,
        output reg[`RegAddrBus]      reg2_addr_o,

        //送到执行阶段的信息
```

```
        output reg[`AluOpBus]         aluop_o,
        output reg[`AluSelBus]        alusel_o,
        output reg[`RegBus]           reg1_o,
        output reg[`RegBus]           reg2_o,
        output reg[`RegAddrBus]       wd_o,
        output reg                    wreg_o
);

  //R 型： 31~26:  opcode，和 function 一起决定指令
  //     25~21:   rs，源寄存器 1
  //     20~16:   rt，源寄存器 2
  //     15~11:   rd，目的寄存器
  //     10~6 :   sa，移位操作中指定移位长度（这里没用）
  //     5~0  :   function，（比如 SUB 和 SUBU 的 opcode 相同，func 不同）
  wire[5:0] op = inst_i[31:26];
  wire[4:0] op2 = inst_i[10:6];
  wire[5:0] op3 = inst_i[5:0];
  wire[4:0] op4 = inst_i[20:16];

  //保存立即数
  reg[`RegBus]   imm;

  //指示指令是否有效
  reg instvalid;

  //进行指令的译码
  always @ (*) begin
    if (rst == `RstEnable) begin
      //置位操作
      aluop_o <= `EXE_NOP_OP;
      alusel_o <= `EXE_RES_NOP;
      wd_o <= `NOPRegAddr;
      wreg_o <= `WriteDisable;
      instvalid <= `InstValid;
      reg1_read_o <= 1'b0;
      reg2_read_o <= 1'b0;
      reg1_addr_o <= `NOPRegAddr;
      reg2_addr_o <= `NOPRegAddr;
      imm <= 32'h0;
    end else begin
      //相当于初始化
      aluop_o <= `EXE_NOP_OP;
      alusel_o <= `EXE_RES_NOP;
      wd_o <= inst_i[15:11];
      wreg_o <= `WriteDisable;
```

```
    instvalid <= `InstInvalid;
    reg1_read_o <= 1'b0;
    reg2_read_o <= 1'b0;
    reg1_addr_o <= inst_i[25:21];
    reg2_addr_o <= inst_i[20:16];
    imm <= `ZeroWord;
    case (op3)
      `EXE_SUBU: begin  //根据 op 的值判断是否是 ori 指令
        //需要写入目的寄存器，所以为WriteEnable
        wreg_o <= `WriteEnable;
        //子类型为SUBU，运算类型为算术运算
        aluop_o <= `EXE_SUBU_OP;
        alusel_o <= `EXE_RES_ARITHMETIC;
        //需要从端口1 和2 读，所以为1
        reg1_read_o <= 1'b1;
        reg2_read_o <= 1'b1;
        instvalid <= `InstValid;
      end
      default: begin
      end
    endcase              //case op
  end       //if
end         //always

//确定源操作数1，reg1_read_o 用于判断比特串是寄存器地址还是立即数
always @ (*) begin
  if(rst == `RstEnable) begin
    reg1_o <= `ZeroWord;
  end else if(reg1_read_o == 1'b1) begin
    reg1_o <= reg1_data_i;
  end else if(reg1_read_o == 1'b0) begin
    reg1_o <= imm;
  end else begin
    reg1_o <= `ZeroWord;
  end
end

//确定源操作数2
always @ (*) begin
  if(rst == `RstEnable) begin
    reg2_o <= `ZeroWord;
  end else if(reg2_read_o == 1'b1) begin
    reg2_o <= reg2_data_i;
  end else if(reg2_read_o == 1'b0) begin
    reg2_o <= imm;
```

```
    end else begin
      reg2_o <= `ZeroWord;
    end
  end

endmodule
```

./Code-SUB 指令/id_ex.v

```
`include "defines.v"

module id_ex(

        input    wire
                        clk,
        input wire
                        rst,


        //从译码阶段传递的信息
        input wire[`AluOpBus]         id_aluop,
        input wire[`AluSelBus]        id_alusel,
        input wire[`RegBus]           id_reg1,
        input wire[`RegBus]           id_reg2,
        input wire[`RegAddrBus]       id_wd,
        input wire                    id_wreg,

        //传递到执行阶段的信息
        output reg[`AluOpBus]         ex_aluop,
        output reg[`AluSelBus]        ex_alusel,
        output reg[`RegBus]           ex_reg1,
        output reg[`RegBus]           ex_reg2,
        output reg[`RegAddrBus]       ex_wd,
        output reg                    ex_wreg

);

        always @ (posedge clk) begin
                if (rst == `RstEnable) begin
                        ex_aluop <= `EXE_NOP_OP;
                        ex_alusel <= `EXE_RES_NOP;
                        ex_reg1 <= `ZeroWord;
                        ex_reg2 <= `ZeroWord;
                        ex_wd <= `NOPRegAddr;
                        ex_wreg <= `WriteDisable;
                end else begin
```

```
                        ex_aluop <= id_aluop;
                        ex_alusel <= id_alusel;
                        ex_reg1 <= id_reg1;
                        ex_reg2 <= id_reg2;
                        ex_wd <= id_wd;
                        ex_wreg <= id_wreg;
                end
        end

endmodule
```

./Code-SUB 指令/if_id.v

```
`include "defines.v"

module if_id(

        input    wire
                        clk,
        input wire
                        rst,


        input wire[`InstAddrBus]                if_pc,
        input wire[`InstBus]          if_inst,
        output reg[`InstAddrBus]      id_pc,
        output reg[`InstBus]          id_inst

);

  always @ (posedge clk) begin
    if (rst == `RstEnable) begin
      id_pc <= `ZeroWord; //复位的时候 pc 为 0
      id_inst <= `ZeroWord;       //复位的时候指令也为 0,实际就是空指令
    end else begin
      id_pc <= if_pc;     //其余时刻向下传递取指阶段的值
      id_inst <= if_inst;
    end
  end
endmodule
```

./Code-SUB 指令/inst_rom.v

```
`include "defines.v"

module inst_rom(
```

```
//      input   wire
                    clk,
        input wire                  ce,
        input wire[`InstAddrBus]                addr,
        output reg[`InstBus]                                    inst

);

        reg[`InstBus]  inst_mem[0:`InstMemNum-1];

        initial $readmemh ( "inst_rom.data", inst_mem );

        always @ (*) begin
                if (ce == `ChipDisable) begin
                        inst <= `ZeroWord;
          end else begin
                  inst <= inst_mem[addr[`InstMemNumLog2+1:2]];
                end
        end

endmodule
```

./Code-SUB 指令/mem.v

```
`include "defines.v"

module mem(

        input wire
                    rst,

        //来自执行阶段的信息
        input wire[`RegAddrBus]      wd_i,
        input wire                  wreg_i,
        input wire[`RegBus]                                     wdata_i,

        //送到回写阶段的信息
        output reg[`RegAddrBus]     wd_o,
        output reg                  wreg_o,
        output reg[`RegBus]                                     wdata_o

);


        always @ (*) begin
                if(rst == `RstEnable) begin
```

```
                        wd_o <= `NOPRegAddr;
                        wreg_o <= `WriteDisable;
                    wdata_o <= `ZeroWord;
                end else begin
                    wd_o <= wd_i;
                        wreg_o <= wreg_i;
                        wdata_o <= wdata_i;
                end    //if
        end      //always


endmodule
```

./Code-SUB 指令/mem_wb.v

```
`include "defines.v"

module mem_wb(

        input    wire
                        clk,
        input wire
                        rst,


        //来自访存阶段的信息
        input wire[`RegAddrBus]       mem_wd,
        input wire                    mem_wreg,
        input wire[`RegBus]
mem_wdata,

        //送到回写阶段的信息
        output reg[`RegAddrBus]      wb_wd,
        output reg                   wb_wreg,
        output reg[`RegBus]                                     wb_wdata


);


        always @ (posedge clk) begin
                if(rst == `RstEnable) begin
                        wb_wd <= `NOPRegAddr;
                        wb_wreg <= `WriteDisable;
                    wb_wdata <= `ZeroWord;
                end else begin
```

```
                        wb_wd <= mem_wd;
                        wb_wreg <= mem_wreg;
                        wb_wdata <= mem_wdata;
                end    //if
        end      //always


endmodule
```

./Code-SUB 指令/openmips.v

```
`include "defines.v"
`include "pc_reg.v"
`include "if_id.v"
`include "id.v"
`include "id_ex.v"
`include "ex.v"
`include "ex_mem.v"
`include "mem.v"
`include "mem_wb.v"
`include "reg_file.v"

module openmips(

        input    wire
                        clk,
        input wire
                        rst,


        input wire[`RegBus]           rom_data_i,
        output wire[`RegBus]          rom_addr_o,
        output wire                   rom_ce_o

);

        wire[`InstAddrBus] pc;
        wire[`InstAddrBus] id_pc_i;
        wire[`InstBus] id_inst_i;

        //连接译码阶段 ID 模块的输出与 ID/EX 模块的输入
        wire[`AluOpBus] id_aluop_o;
        wire[`AluSelBus] id_alusel_o;
        wire[`RegBus] id_reg1_o;
        wire[`RegBus] id_reg2_o;
        wire id_wreg_o;
```

```
        wire[`RegAddrBus] id_wd_o;

        //连接 ID/EX 模块的输出与执行阶段 EX 模块的输入
        wire[`AluOpBus] ex_aluop_i;
        wire[`AluSelBus] ex_alusel_i;
        wire[`RegBus] ex_reg1_i;
        wire[`RegBus] ex_reg2_i;
        wire ex_wreg_i;
        wire[`RegAddrBus] ex_wd_i;

        //连接执行阶段 EX 模块的输出与 EX/MEM 模块的输入
        wire ex_wreg_o;
        wire[`RegAddrBus] ex_wd_o;
        wire[`RegBus] ex_wdata_o;

        //连接 EX/MEM 模块的输出与访存阶段 MEM 模块的输入
        wire mem_wreg_i;
        wire[`RegAddrBus] mem_wd_i;
        wire[`RegBus] mem_wdata_i;

        //连接访存阶段 MEM 模块的输出与 MEM/WB 模块的输入
        wire mem_wreg_o;
        wire[`RegAddrBus] mem_wd_o;
        wire[`RegBus] mem_wdata_o;

        //连接 MEM/WB 模块的输出与回写阶段的输入
        wire wb_wreg_i;
        wire[`RegAddrBus] wb_wd_i;
        wire[`RegBus] wb_wdata_i;

        //连接译码阶段 ID 模块与通用寄存器 Regfile 模块
wire reg1_read;
wire reg2_read;
wire[`RegBus] reg1_data;
wire[`RegBus] reg2_data;
wire[`RegAddrBus] reg1_addr;
wire[`RegAddrBus] reg2_addr;

//pc_reg 例化
        pc_reg pc_reg0(
                .clk(clk),
                .rst(rst),
                .pc(pc),
                .ce(rom_ce_o)
```

```
        );

assign rom_addr_o = pc;

//IF/ID 模块例化
        if_id if_id0(
                .clk(clk),
                .rst(rst),
                .if_pc(pc),
                .if_inst(rom_data_i),
                .id_pc(id_pc_i),
                .id_inst(id_inst_i)
        );

        //译码阶段 ID 模块
        id id0(
                .rst(rst),
                .pc_i(id_pc_i),
                .inst_i(id_inst_i),

                .reg1_data_i(reg1_data),
                .reg2_data_i(reg2_data),

                //送到regfile 的信息
                .reg1_read_o(reg1_read),
                .reg2_read_o(reg2_read),

                .reg1_addr_o(reg1_addr),
                .reg2_addr_o(reg2_addr),

                //送到 ID/EX 模块的信息
                .aluop_o(id_aluop_o),
                .alusel_o(id_alusel_o),
                .reg1_o(id_reg1_o),
                .reg2_o(id_reg2_o),
                .wd_o(id_wd_o),
                .wreg_o(id_wreg_o)
        );

//通用寄存器 Regfile 例化
        regfile regfile1(
                .clk (clk),
                .rst (rst),
                .we     (wb_wreg_i),
                .waddr (wb_wd_i),
```

```
            .wdata (wb_wdata_i),
            .re1 (reg1_read),
            .raddr1 (reg1_addr),
            .rdata1 (reg1_data),
            .re2 (reg2_read),
            .raddr2 (reg2_addr),
            .rdata2 (reg2_data)
    );

    //ID/EX 模块
    id_ex id_ex0(
            .clk(clk),
            .rst(rst),

            //从译码阶段 ID 模块传递的信息
            .id_aluop(id_aluop_o),
            .id_alusel(id_alusel_o),
            .id_reg1(id_reg1_o),
            .id_reg2(id_reg2_o),
            .id_wd(id_wd_o),
            .id_wreg(id_wreg_o),

            //传递到执行阶段 EX 模块的信息
            .ex_aluop(ex_aluop_i),
            .ex_alusel(ex_alusel_i),
            .ex_reg1(ex_reg1_i),
            .ex_reg2(ex_reg2_i),
            .ex_wd(ex_wd_i),
            .ex_wreg(ex_wreg_i)
    );

    //EX 模块
    ex ex0(
            .rst(rst),

            //送到执行阶段 EX 模块的信息
            .aluop_i(ex_aluop_i),
            .alusel_i(ex_alusel_i),
            .reg1_i(ex_reg1_i),
            .reg2_i(ex_reg2_i),
            .wd_i(ex_wd_i),
            .wreg_i(ex_wreg_i),

      //EX 模块的输出到 EX/MEM 模块信息
            .wd_o(ex_wd_o),
```

```
                .wreg_o(ex_wreg_o),
                .wdata_o(ex_wdata_o)

        );

//EX/MEM 模块
ex_mem ex_mem0(
                .clk(clk),
                .rst(rst),

                //来自执行阶段 EX 模块的信息
                .ex_wd(ex_wd_o),
                .ex_wreg(ex_wreg_o),
                .ex_wdata(ex_wdata_o),


                //送到访存阶段 MEM 模块的信息
                .mem_wd(mem_wd_i),
                .mem_wreg(mem_wreg_i),
                .mem_wdata(mem_wdata_i)


        );

//MEM 模块例化
        mem mem0(
                .rst(rst),

                //来自 EX/MEM 模块的信息
                .wd_i(mem_wd_i),
                .wreg_i(mem_wreg_i),
                .wdata_i(mem_wdata_i),

                //送到MEM/WB 模块的信息
                .wd_o(mem_wd_o),
                .wreg_o(mem_wreg_o),
                .wdata_o(mem_wdata_o)
        );

//MEM/WB 模块
        mem_wb mem_wb0(
                .clk(clk),
                .rst(rst),
```

```
                //来自访存阶段MEM模块的信息
                .mem_wd(mem_wd_o),
                .mem_wreg(mem_wreg_o),
                .mem_wdata(mem_wdata_o),

                //送到回写阶段的信息
                .wb_wd(wb_wd_i),
                .wb_wreg(wb_wreg_i),
                .wb_wdata(wb_wdata_i)


        );

endmodule
```

./Code-SUB 指令/openmips_min_sopc_tb.v

```
`include "defines.v"
`include "openmips_mini_sopc.v"
`timescale 1ns/1ps

module openmips_min_sopc_tb();

  reg     CLOCK_50;
  reg     rst;


  initial begin

    $dumpfile("test.vcd");
    $dumpvars(0, openmips_min_sopc_tb);

    CLOCK_50 = 1'b0;
    forever #10 CLOCK_50 = ~CLOCK_50;
  end

  initial begin

    rst = `RstEnable;
    #195 rst= `RstDisable;
    #1000 $stop;
  end

  openmips_min_sopc openmips_min_sopc0(
                .clk(CLOCK_50),
                .rst(rst)
```

```
        );

endmodule
```

./Code-SUB 指令/openmips_mini_sopc.v

```
`include "defines.v"
`include "openmips.v"
`include "inst_rom.v"

module openmips_min_sopc(

        input    wire
                        clk,
        input wire
                        rst

);

  //连接指令存储器
  wire[`InstAddrBus] inst_addr;
  wire[`InstBus] inst;
  wire rom_ce;


 openmips openmips0(
                .clk(clk),
                .rst(rst),

                .rom_addr_o(inst_addr),
                .rom_data_i(inst),
                .rom_ce_o(rom_ce)
        );

        inst_rom inst_rom0(
                .addr(inst_addr),
                .inst(inst),
                .ce(rom_ce)
        );


endmodule
```

./Code-SUB 指令/pc_reg.v

```
`include "defines.v"
```

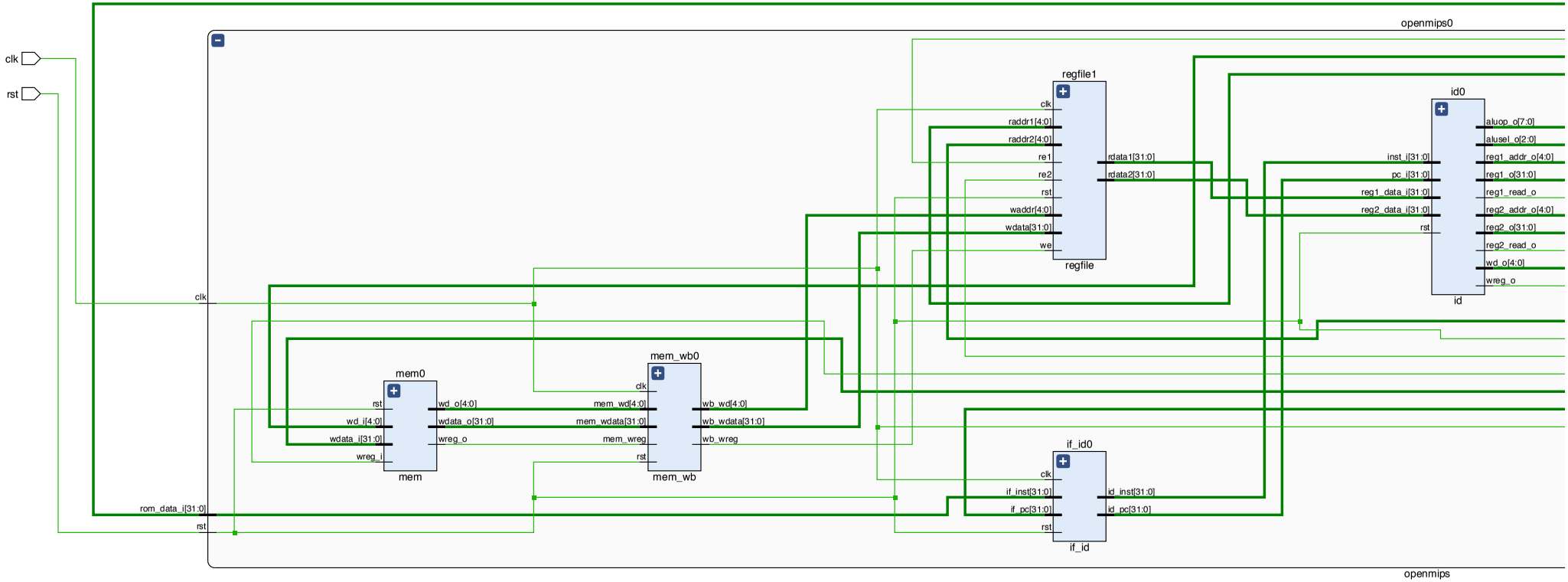
```
module pc_reg(

        input    wire
                        clk,
        input wire
                        rst,

        output reg[`InstAddrBus]                pc,
        output reg                    ce

);

  always @ (posedge clk) begin
    if (ce == `ChipDisable) begin
      pc <= 32'h00000000;
    end else begin
      pc <= pc + 4'h4;
    end
  end

  always @ (posedge clk) begin
    if (rst == `RstEnable) begin
      ce <= `ChipDisable;
    end else begin
      ce <= `ChipEnable;
    end
  end
endmodule
```

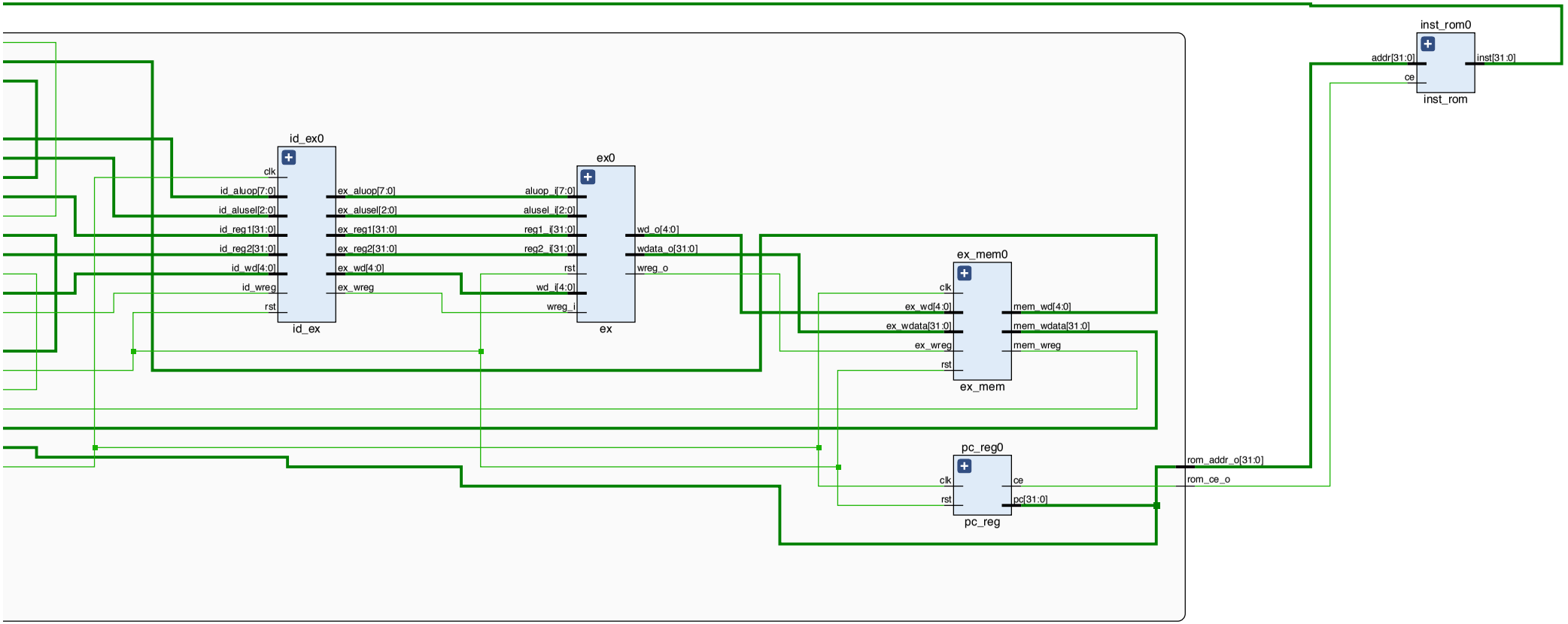
./Code-SUB 指令/reg_file.v

```
`include "defines.v"

module regfile(

        input    wire
                        clk,
        input wire
                        rst,

        //写端口
        input wire
                        we,
        input wire[`RegAddrBus]                         waddr,
        input wire[`RegBus]
        wdata,
```

```
        //读端口1
        input wire
                        re1,
        input wire[`RegAddrBus]                     raddr1,
        output reg[`RegBus]          rdata1,

        //读端口2
        input wire
                        re2,
        input wire[`RegAddrBus]                     raddr2,
        output reg[`RegBus]          rdata2

);

        reg[`RegBus]  regs[0:`RegNum-1];

  always @ (posedge clk) begin
    regs[1] = 32'h11111111;
    regs[2] = 32'h10001000;
    regs[4] = 32'h00100010;
    regs[6] = 32'h01001000;
    regs[8] = 32'h11101110;
    regs[10] = 32'h10101010;
    regs[12] = 32'h01101110;
    regs[14] = 32'h11000110;
    regs[16] = 32'h10100000;
    regs[18] = 32'h00000fff;
    regs[20] = 32'h0fffffff;
  end

        always @ (posedge clk) begin
                if (rst == `RstDisable) begin
                        if((we == `WriteEnable) && (waddr !=
`RegNumLog2'h0)) begin
                                regs[waddr] <= wdata;
                        end
                end
        end

        always @ (*) begin
                if(rst == `RstEnable) begin
                          rdata1 <= `ZeroWord;
          end else if(raddr1 == `RegNumLog2'h0) begin
                        rdata1 <= `ZeroWord;
```

```
          end else if((raddr1 == waddr) && (we == `WriteEnable)
                          && (re1 == `ReadEnable)) begin
                  rdata1 <= wdata;
          end else if(re1 == `ReadEnable) begin
              rdata1 <= regs[raddr1];
          end else begin
              rdata1 <= `ZeroWord;
          end
        end

        always @ (*) begin
                if(rst == `RstEnable) begin
                          rdata2 <= `ZeroWord;
          end else if(raddr2 == `RegNumLog2'h0) begin
                        rdata2 <= `ZeroWord;
          end else if((raddr2 == waddr) && (we == `WriteEnable)
                          && (re2 == `ReadEnable)) begin
                  rdata2 <= wdata;
          end else if(re2 == `ReadEnable) begin
              rdata2 <= regs[raddr2];
          end else begin
              rdata2 <= `ZeroWord;
          end
        end

endmodule
```


clk
rst
openmips0
clk
rom_data_i[31:0]
rst
mem0
rst
wd_i[4:0]
wdata_i[31:0]
wreg_i
wd_o[4:0]
wdata_o[31:0]
wreg_o
mem
mem_wb0
clk
mem_wd[4:0]
mem_wdata[31:0]
mem_wreg
rst
wb_wd[4:0]
wb_wdata[31:0]
wb_wreg
mem_wb
regfile1
clk
raddr1[4:0]
raddr2[4:0]
re1
re2
rst
waddr[4:0]
wdata[31:0]
we
rdata1[31:0]
rdata2[31:0]
regfile
if_id0
clk
if_inst[31:0]
if_pc[31:0]
rst
id_inst[31:0]
id_pc[31:0]
if_id
id0
inst_i[31:0]
pc_i[31:0]
reg1_data_i[31:0]
reg2_data_i[31:0]
rst
aluop_o[7:0]
alusel_o[2:0]
reg1_addr_o[4:0]
reg1_o[31:0]
reg1_read_o
reg2_addr_o[4:0]
reg2_o[31:0]
reg2_read_o
wd_o[4:0]
wreg_o
id
openmips


inst_rom0
addr[31:0]
ce
inst[31:0]
inst_rom
id_ex0
clk
id_aluop[7:0]
id_alusel[2:0]
id_reg1[31:0]
id_reg2[31:0]
id_wd[4:0]
id_wreg
rst
ex_aluop[7:0]
ex_alusel[2:0]
ex_reg1[31:0]
ex_reg2[31:0]
ex_wd[4:0]
ex_wreg
id_ex
ex0
aluop_i[7:0]
alusel_i[2:0]
reg1_i[31:0]
reg2_i[31:0]
rst
wd_i[4:0]
wreg_i
wd_o[4:0]
wdata_o[31:0]
wreg_o
ex
ex_mem0
clk
ex_wd[4:0]
ex_wdata[31:0]
ex_wreg
rst
mem_wd[4:0]
mem_wdata[31:0]
mem_wreg
ex_mem
pc_reg0
clk
rst
ce
pc[31:0]
pc_reg
rom_addr_o[31:0]
rom_ce_o