**实验报告**

2021年06月15日 成绩：---------

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 姓名 | 刘爱兵 | 学号 | 19114801 | 班级 | 19计科2班 |
| 专业 | 计算机科学与技术 | 课程名称 | 计算机组成原理实验 | 任课教师 | 吴云志 |
| 实验序号 | 07 | 实验名称 | 实现R型指令的CPU设计实验 | | |
| 实验时间 | 2021.06.15 | 机位号 | 18 | 实验设备号 | 21 |

1. **实验目的与要求**

1、实验目的

掌握MIPS R型指令的数据通路设计，掌握指令流和数据流的控制方法；

掌握完整的单周期CPU顶层模块的设计方法；

实现MIPS R型指令的功能；

2、实验内容与原理

设计实现一个单周期CPU，实现8条R型指令；

（1）建立R型指令的数据通路；

（2）构造顶层模块，含部件：

指令存储器（实验四）

PC及自增电路（实验五）

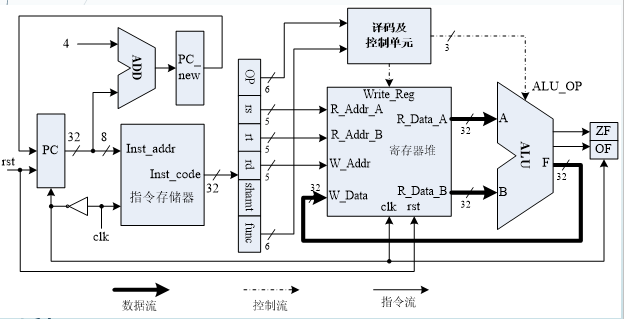
寄存器堆模块（实验三）

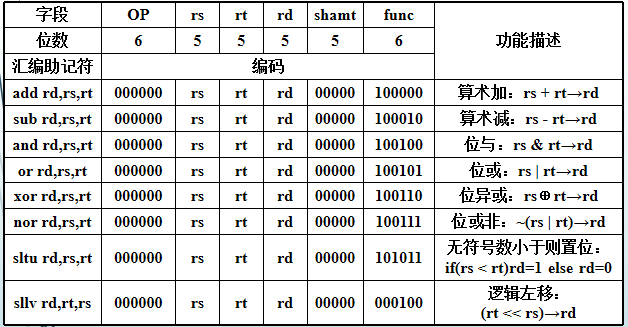
ALU模块（实验二）

指令译码与控制单元：新增

根据指令码和功能码，为数据通路上各部件发送控制信号（置位或复位）；

（1）R型指令数据通路：

（2）R型指令集：



由上表可知,R型指令的共同特征：

操作码字段OP=000000b

指令的功能则由功能码字段func指出

R型指令的操作数有3个

两个源操作数在rs和rt字段所指定的寄存器中

目的操作数是rt字段所指定的寄存器

汇编助记符中，紧跟指令符号右边的寄存器是目的寄存器，这和指令机器码的排列顺序有差异

（3）构造CPU顶层模块：

①新建一个工程；

②新建一个Veilog Module作为CPU顶层模块；

③拷贝各个模块的\*.v文件到当前工程目录下；

④将各个\*.v文件加到工程中来；

⑤对于指令存储器的\*.xco，处理较复杂（可重新在工程中生成ROM的IP核）

⑥在顶层模块中，引用各个模块的实例；

定义一组信号做为各个模块间的信号连接；

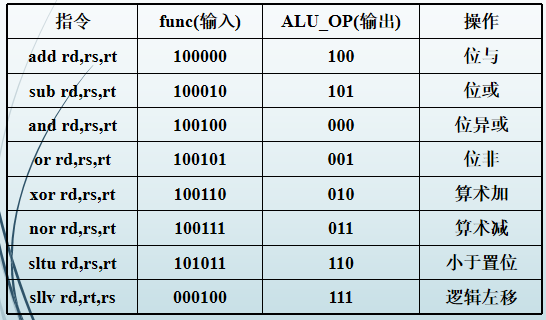
⑦在顶层模块中，编写有关指令译码和控制单元的程序；

注意：

寄存器模块：$0要始终置为0，且不允许写操作；

指令译码：OP=000000为R型指令，由func字段识别具体R型指令；

控制信号赋值：依据指令译码，见下表



指令译码与控制：OP=6‘b000000

时序设计

时钟源为CPU的主频脉冲clk

clk的上升沿，启动指令存储器依据PC读指令

clk高电平持续期间，完成PC值的自增、指令译码、寄存器读操作，随后完成ALU运算

clk的下降沿则完成目的寄存器的写入、PC值的更新和标志寄存器的更新

时序实现时：

指令存储器的clk：clk

PC寄存器的打入clk：~clk

寄存器堆的写入clk：~clk

指令对标志位的影响

传送类指令和跳转类指令不影响标志位

有符号算术运算类指令（包括slt和算术移位指令）影响ZF和OF

无符号算术运算类指令和逻辑运算类指令影响ZF，不影响OF

条件转移类指令一般会使用标志位ZF

标志位的赋值

算术运算类指令：影响标志位ZF和OF

逻辑运算类指令：只影响标志位ZF

存储器访问指令：不影响标志位

无条件跳转指令：不影响标志位

有条件分支指令：影响标志位ZF

影响→赋值

不影响→不赋值

（4）指令测试

实现的CPU能够支持8条R型指令子集吗？

用8条R型指令编写一段程序，用于测试CPU的功能：

测试程序：汇编

1. **nor $1, $0,$0; #$1=FFFF\_FFFF**
2. **sltu $2, $0, $1; #$2=0000\_0001**
3. **add $3, $2, $2; #$3=0000\_0002**
4. **add $4, $3, $2; #$4=0000\_0003**
5. **add $5, $4, $3; #$5=0000\_0005**
6. **add $6, $5, $3; #$6=0000\_0007**
7. **sllv $7, $6, $2; #$7=0000\_000E**
8. **add $9, $5, $6; #$9=0000\_000C**
9. **sllv $8, $6, $9; #$8=0000\_7000**
10. **xor $9, $1, $8; #$9=FFFF\_8FFF**
11. **add $10, $9, $1; #$10=FFFF\_8FFE**
12. **sub $11, $8, $7; #$11=0000\_6FF2**
13. **sub $12, $7, $8; #$12=FFFF\_900E**
14. **and $13, $9, $12; #$13=FFFF\_800E**
15. **or $14, $9, $12; #$14=FFFF\_9FFF**
16. **or $15, $6, $7; #$15=0000\_000F**
17. **nor $16, $6, $7; #$16=FFFF\_FFF0**
18. **add $17, $7, $3; #$17=0000\_0010**
19. **sllv $18, $8, $17; #$18=7000\_0000**
20. **sllv $19, $3, $17; #$19=0002\_0000**
21. **sllv $20, $19, $7; #$20=8000\_0000**
22. **add $21, $20, $1; #$21=7FFF\_FFFF**
23. **or $22, $18, $21; #$22=7FFF\_FFFF**
24. **add $23, $20, $22; #$23=FFFF\_FFFF**
25. **sub $24, $20, $22; #$24=0000\_0001**
26. **sub $25, $22, $20; #$25=FFFF\_FFFF**
27. **xor $26, $18, $1; #$26=8FFF\_FFFF**
28. **sltu $27, $22, $20; #$27=0000\_0001**
29. **sltu $28, $26, $20; #$28=0000\_0000**
30. **add $29, $22, $2; #$29=8000\_0000**
31. **sub $30, $20, $2; #$30=7FFF\_FFFF**
32. **add $31, $11, $26; #$30=9000\_6FF1**

测试程序：机器指令编码

00000827, 0001102b, 00421820, 00622020, 00832820, 00a33020, 00463804, 00a64820, 01264004, 00284826, 01215020, 01075822, 00e86022, 012c6824, 012c7025, 00c77825, 00c78027, 00e38820, 02289004, 02239804, 00f3a004, 0281a820, 0255b025, 0296b820, 0296c022, 02d4c822, 0241d026, 02d4d82b, 0354e02b, 02c2e820, 0282f022, 017af820

①将机器代码编辑成\*.coe文件（格式化）；

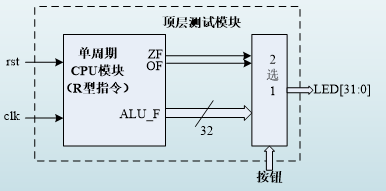
②与指令存储器关联；

③重新生成ROM核：Regenerate Core操作

（5）实验验证

在板卡上如何看到测试程序的每条指令的执行结果？如何验证功能是否正确？

在CPU模块之上，再构造一个实验验证模块，其中引用改造过的CPU模块（引出ALU的运算结果）



3、实验要求

在实验二~实验五的基础上，编写一个CPU模块，能够实现8条指定的R型指令。

编写一段测试8条指令的汇编程序，使用实验九的汇编器，将其翻译成二进制机器码，并通过关联文件初始化指令存储器。

编写一个实验验证的顶层模块，用于验证实验。

实验室任务：

配置管脚；



生成\*.bit文件并下载。

完成板级验证。

撰写实验报告。

4、实验步骤

在Xilinx ISE/Vivado中创建工程，编码，然后编译、综合

编写激励代码，观察仿真波形，直至验证正确

实验准备；

在PC机上打开工程文件，进行管脚配置。

生成编程文件\*.bit，下载到板卡中。

实验十一实现R型指令的CPU设计实验

实验：

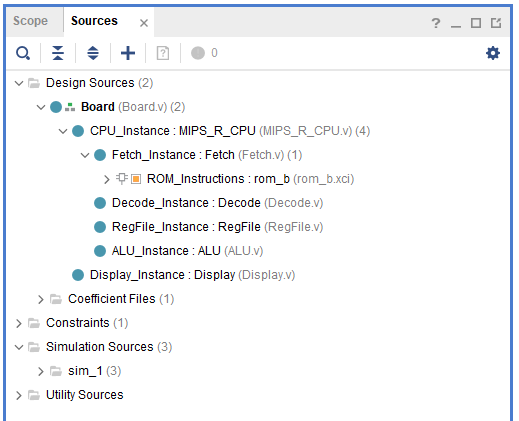
按动Rst按钮使PC、寄存器堆、ZF、OF清零；

每按动Clk按钮一次，则执行一条指令；

1个按钮选择显示ALU的32位结果字或者标志，并记录

1. **实验设计与程序代码**
2. 模块设计说明

（描述这个实验的设计方案，分几个模块，各模块的功能，各模块之间的连接关系，可附图）



1. 实验程序源代码及注释等

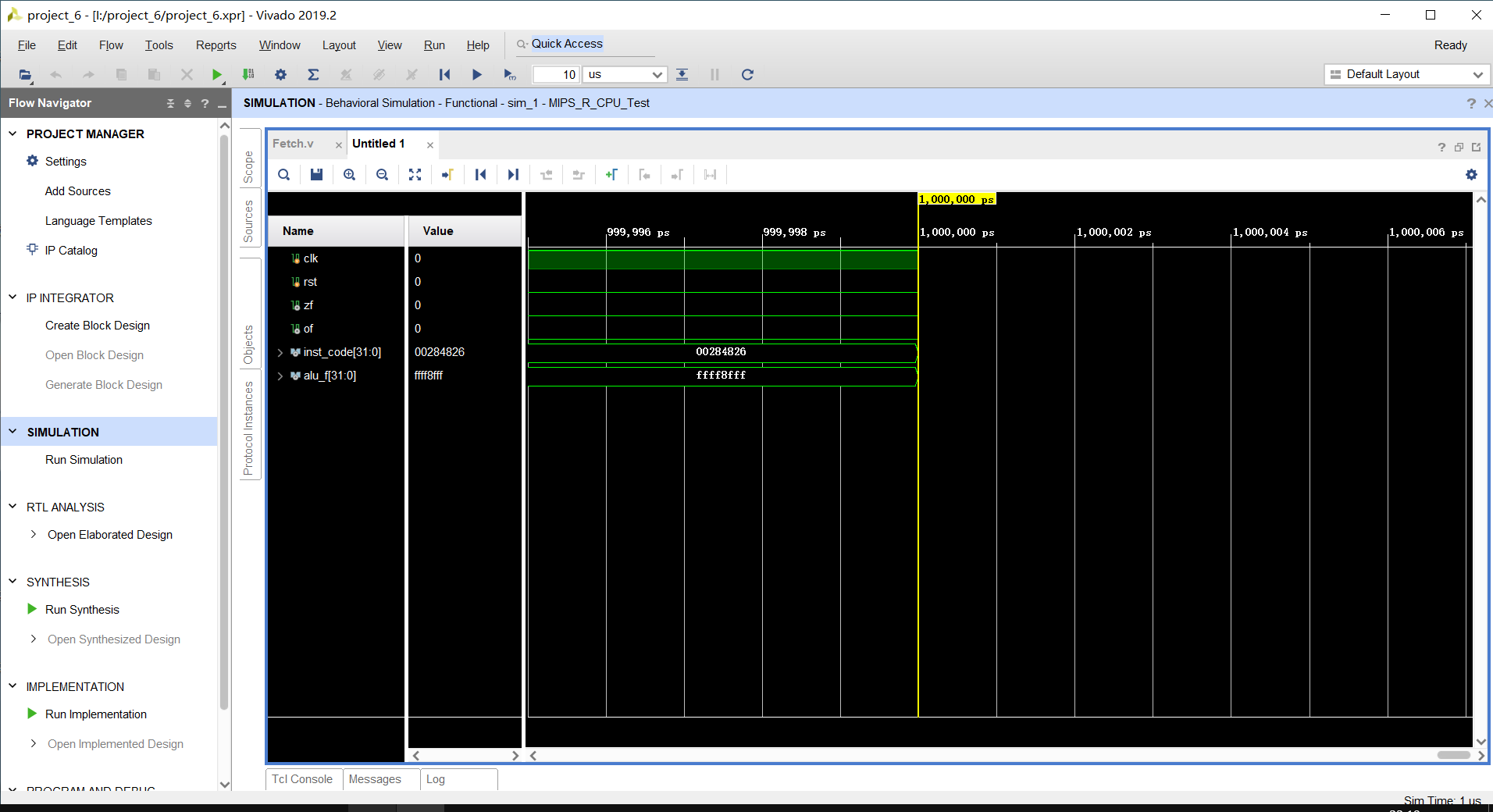
（实验各个模块的代码，包含功能注释）

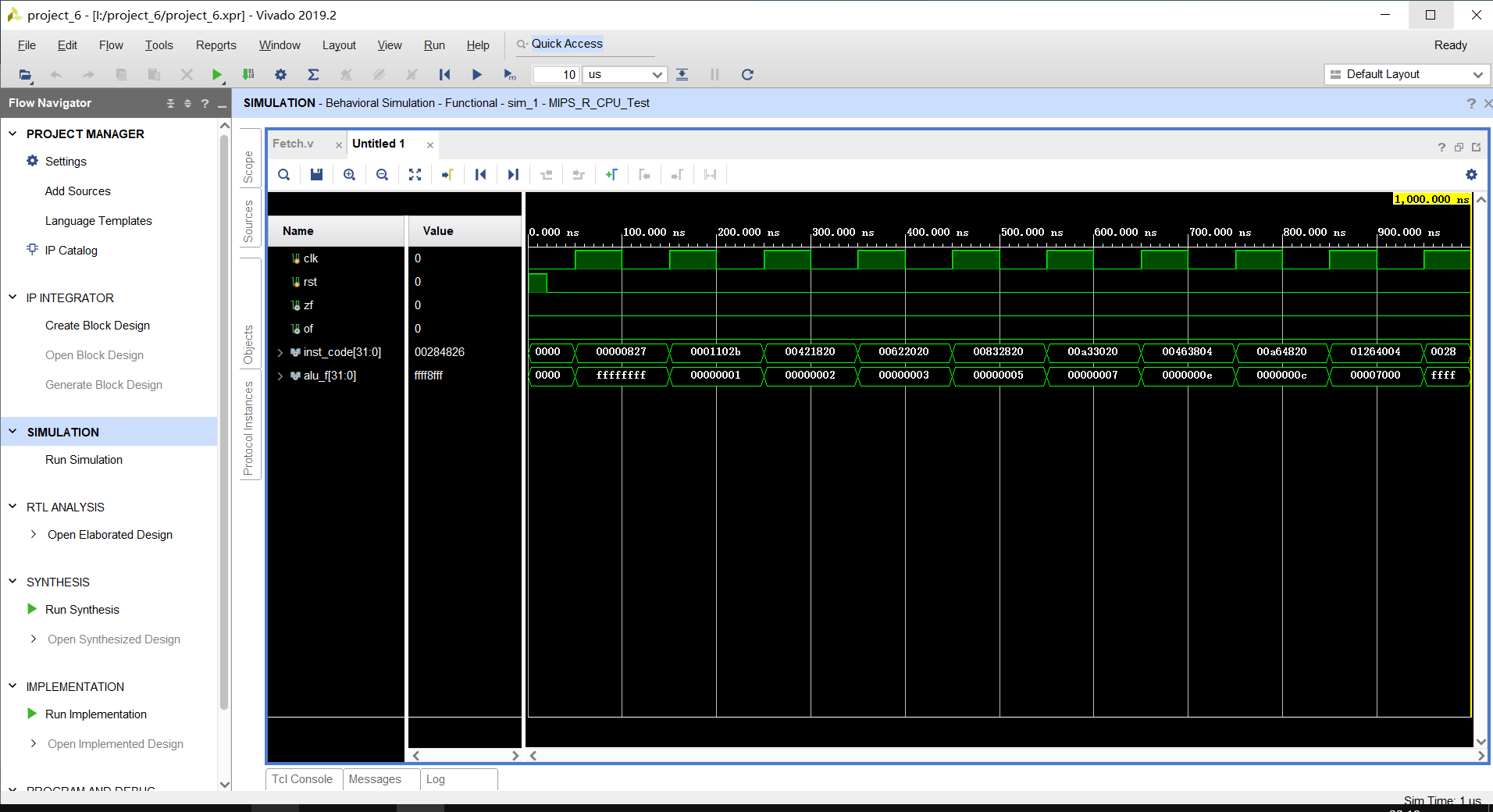
|  |
| --- |
| 1. **ALU.v**   `timescale 1ns / 1ps  // Arithmetic Logic Unit，算数逻辑单元  module ALU(op, a, b, f, zero, ovf);  parameter N = 32; // bit witdh，数据位宽  input [3:0] op; // operator，运算符编码  input [N:1] a, b; // operands，左右操作数  output reg [N:1] f; // result，运算结果  reg cout = 0; // carry-out，最高位进位  always @\* case (op) // 判断运算符并进行运算  0: {cout, f} <= a & b; // bitwise logical and，按位逻辑与  1: {cout, f} <= a | b; // bitwise logical or，按位逻辑或  2: {cout, f} <= a ^ b; // bitwise logical exclusive or，按位逻辑异或  3: {cout, f} <= ~(a | b); // bitwise logical nor，按位逻辑或非  4: {cout, f} <= a + b; // arithmetic signed add，算数有符号加法  5: {cout, f} <= a - b; // arithmetic signed subtract，算数有符号减法  6: {cout, f} <= a < b; // set on less than (unsigned)，无符号数小于置位（a < b 则结果为 1，否则为 0）  7: {cout, f} <= b << a; // shift left logical variable (shift in zeros)，将 b 逻辑左移 a 位，右端补零  default: {cout, f} <= 0; // 未定义运算，结果置零  endcase  output zero, ovf; // zero, overflow，零、溢出  assign zero = ~(|f); // 结果归约或、取反得到，即结果全零则为 1，否则为 0  assign ovf = a[N] ^ b[N] ^ f[N] ^ cout; // 由判溢公式计算（只对有符号数有意义）  // output neg, even; // negative, even(parity)，负数、偶数  // assign neg = f[N]; // 取结果最高位，补码符号位 0正 1负（只对有符号数有意义）  // assign even = ~^F; // 结果中 1 为偶数个则置 1，否则置 0  endmodule // ALU   1. **Board.v**   `timescale 1ns / 1ps  // MIPS R型指令 CPU 板极调试  module Board(swb, led, clk, which, seg, enable);  input [1:6] swb;  output [1:32] led;  wire zf, of;  wire [31:0] alu\_f;  MIPS\_R\_CPU CPU\_Instance(.clk(swb[1]), .rst(swb[2]),  .inst\_code(led), // LED：指令机器码  .alu\_f(alu\_f),  .zf(zf), .of(of));  input clk; // 数码管相关  output [2:0] which;  output [7:0] seg;  output reg enable = 1; // 默认开启数码管使能  reg [31:0] data;  Display Display\_Instance(.clk(clk), .data(data), .which(which), .seg(seg));  // SWB3：按住时，数码管显示 ALU 运算结果；松开时，最低两位显示标志位  always @\* data <= swb[3] ? alu\_f : {zf, 3'b0, of};  endmodule // Board   1. **Decode.v**   `timescale 1ns / 1ps  module Decode( // R-Type Instruction Decode，R型指令解码  output [5:0] op, // opcode (R-I)  output [4:0] rs, // register source (R-I)  output [4:0] rt, // register target (R-I)  output [4:0] rd, // register destination (R)  output [4:0] shift, // shamt (R)  output [5:0] funct, // function (R)  input [31:0] inst\_code); // instruction machine code  assign op = inst\_code[31:26];  assign rs = inst\_code[25:21];  assign rt = inst\_code[20:16];  assign rd = inst\_code[15:11];  assign shift = inst\_code[10: 6];  assign funct = inst\_code[ 5: 0];  endmodule // Decode   1. **Display.v**   `timescale 1ns / 1ps  // 8 位七段数码管扫描显示模块  module Display(clk, data, which, seg,  count, digit); // 调试接口  input clk; // 接入系统时钟  input [32:1] data; // 32 位显示数据  output reg [2:0] which = 0; // 片选编码（驱动哪一位数码管），低电平有效  output reg [7:0] seg; // 段选信号（点亮哪些笔划），低电平有效  output reg [10:0] count = 0; // 分频扫描，从左至右循环驱动每一位数码管  always @(posedge clk) count <= count + 1'b1;  always @(negedge clk) if (&count) which <= which + 1'b1;  output reg [3:0] digit; // 显示数据 片选得到 十六进制数码  always @\* case (which)  0: digit <= data[32:29]; // 最高位  1: digit <= data[28:25];  2: digit <= data[24:21];  3: digit <= data[20:17];  4: digit <= data[16:13];  5: digit <= data[12:09];  6: digit <= data[08:05];  7: digit <= data[04:01]; // 最低位  endcase  always @\* case (digit) // 十六进制数码 转换为 段选信号（a,b,c,...g,dp）  4'h0: seg <= 8'b0000\_0011; // 除 g、dp 外全亮，显示数码 0  4'h1: seg <= 8'b1001\_1111; // 仅 b、c 亮，显示数码 1  4'h2: seg <= 8'b0010\_0101;  4'h3: seg <= 8'b0000\_1101;  4'h4: seg <= 8'b1001\_1001;  4'h5: seg <= 8'b0100\_1001;  4'h6: seg <= 8'b0100\_0001;  4'h7: seg <= 8'b0001\_1111;  4'h8: seg <= 8'b0000\_0001;  4'h9: seg <= 8'b0000\_1001;  4'hA: seg <= 8'b0001\_0001;  4'hB: seg <= 8'b1100\_0001;  4'hC: seg <= 8'b0110\_0011;  4'hD: seg <= 8'b1000\_0101;  4'hE: seg <= 8'b0110\_0001;  4'hF: seg <= 8'b0111\_0001;  endcase  endmodule // Display   1. **Fetch.v**   `timescale 1ns / 1ps  // Instruction Fetching，取指令模块  module Fetch(clk, rst, inst\_code,  pc, pc\_new); // 调试输出  input clk, rst; // Clock and Reset，时钟、复位信号  output [31:0] inst\_code; // Instruction Code，指令机器码  output reg [31:0] pc; // Program Counter，程序计数器  output [31:0] pc\_new; // PC+4，用于更新 PC  always @(negedge clk) pc <= rst ? 32'b0 : pc\_new; // 同步清零、同步更新 PC  assign pc\_new = pc + 4; // 组合逻辑，暂存 PC+4  ROM ROM\_Instructions( // Instantiate a ROM storing instructions  .clka(clk), // input wire clka  .addra(pc[7:2]), // input wire [5 : 0] addra  .douta(inst\_code)); // output wire [31 : 0] douta  endmodule // Fetch   1. **MIPS\_R\_CPU.v**   `timescale 1ns / 1ps  // MIPS R型指令 CPU  module MIPS\_R\_CPU(clk, rst, zf, of,  inst\_code, alu\_f); // output for debug  input clk, rst;  output [31:0] inst\_code; // 取指令  Fetch Fetch\_Instance(.clk(clk), .rst(rst), .inst\_code(inst\_code));  wire [5:0] op, funct; // 指令解码  wire [4:0] rs, rt, rd;  Decode Decode\_Instance(.inst\_code(inst\_code),  .op(op), .rs(rs), .rt(rt), .rd(rd), .funct(funct));  reg rf\_write; // 寄存器堆  wire [31:0] rf\_a\_data, rf\_b\_data;  output [31:0] alu\_f;  RegFile RegFile\_Instance(.wrt(rf\_write), .rst(rst), .clk(~clk), // 反相时钟脉冲  .w\_addr(rd), .w\_data(alu\_f), // ALU 运算结果 写入 RegFile[rd]  .a\_data(rf\_a\_data), .b\_data(rf\_b\_data), .a\_addr(rs), .b\_addr(rt));  reg [3:0] alu\_op; // ALU  wire alu\_zero, alu\_ovf;  ALU ALU\_Instance(.op(alu\_op), // 运算符编码待定  .a(rf\_a\_data), .b(rf\_b\_data), // 从寄存器堆双端口 读取 左右操作数  .zero(alu\_zero), .ovf(alu\_ovf), .f(alu\_f));  output reg zf, of; // 标志寄存器（类 x86 状态寄存器）  reg set\_zf, set\_of; // 置位/更新信号  always @(negedge clk) zf <= rst ? 0 : (set\_zf & alu\_zero);  always @(negedge clk) of <= rst ? 0 : (set\_of & alu\_ovf);  always @\* if (op === 6'b00\_0000) begin // R-Type Instructions  rf\_write <= 1; // RegFile[rd] = alu\_f  case (funct)  6'b10\_0100: begin alu\_op <= 0; set\_zf <= 1; set\_of <= 0; end // AND  6'b10\_0101: begin alu\_op <= 1; set\_zf <= 1; set\_of <= 0; end // OR  6'b10\_0110: begin alu\_op <= 2; set\_zf <= 1; set\_of <= 0; end // XOR  6'b10\_0111: begin alu\_op <= 3; set\_zf <= 1; set\_of <= 0; end // NOR  6'b10\_0000: begin alu\_op <= 4; set\_zf <= 1; set\_of <= 1; end // ADD (Signed)  6'b10\_0010: begin alu\_op <= 5; set\_zf <= 1; set\_of <= 1; end // SUB (Signed)  6'b10\_1011: begin alu\_op <= 6; set\_zf <= 1; set\_of <= 0; end // SLTU: Set on Less Than Unsigned  6'b00\_0100: begin alu\_op <= 7; set\_zf <= 1; set\_of <= 0; end // SLLV: Shift Left Logical Variable (Shift in Zeros)  default: begin alu\_op <= 8; set\_zf <= 0; set\_of <= 0; end // 非算术逻辑运算类/未定义 R型指令  endcase  end  endmodule // MIPS\_R\_CPU   1. **RegFile.v**   `timescale 1ns / 1ps  // Register File，寄存器堆  module RegFile(clk, rst, wrt, w\_addr, w\_data, // 单端口写入  a\_addr, b\_addr, a\_data, b\_data); // A、B 双端口读取  parameter ADDR = 5; // 寄存器堆地址位宽，寻址能力决定最大字数  parameter NUMB = 1 << ADDR; // 寄存器堆字数，如 2^5 共 32 个字  parameter SIZE = 32; // 寄存器堆字长（字的位宽）  reg [SIZE:1] reg\_file [0:NUMB-1]; // NUMB x SIZE 位寄存器堆  integer i; // 用于遍历 NUMB 个字  input clk, rst, wrt; // clock, reset, write：时钟、清零信号、写控制信号  input [ADDR:1] w\_addr; // 写入地址  input [SIZE:1] w\_data; // 写入数据  always @(posedge clk) begin  if (rst) for (i = 0; i < NUMB; i = i+1) reg\_file[i] <= 0; // 同步清零  else if (wrt && w\_addr) reg\_file[w\_addr] <= w\_data;  end // 时钟上跳且写控制高电平时写入，且 reg\_file[0] 只读  input [ADDR:1] a\_addr, b\_addr; // A、B 两端口读取地址  output [SIZE:1] a\_data, b\_data; // A、B 两端口读出数据  assign a\_data = reg\_file[a\_addr];  assign b\_data = reg\_file[b\_addr];  endmodule // RegFile |

1. **实验仿真**
2. 仿真代码（仿真源代码）

|  |
| --- |
| `timescale 1ns / 1ps  // MIPS R型指令 CPU 仿真测试  module MIPS\_R\_CPU\_Test();  reg clk, rst; // input  wire zf, of;  wire [31:0] inst\_code, alu\_f; // output  MIPS\_R\_CPU CPU\_Instance(.clk(clk), .rst(rst), .zf(zf), .of(of),  .inst\_code(inst\_code), .alu\_f(alu\_f));  always #50 clk = ~clk;  initial begin clk = 0; rst = 1; #20; rst = 0; end  endmodule // MIPS\_R\_CPU\_Test |

1. 仿真波形（运行仿真时波形截图）



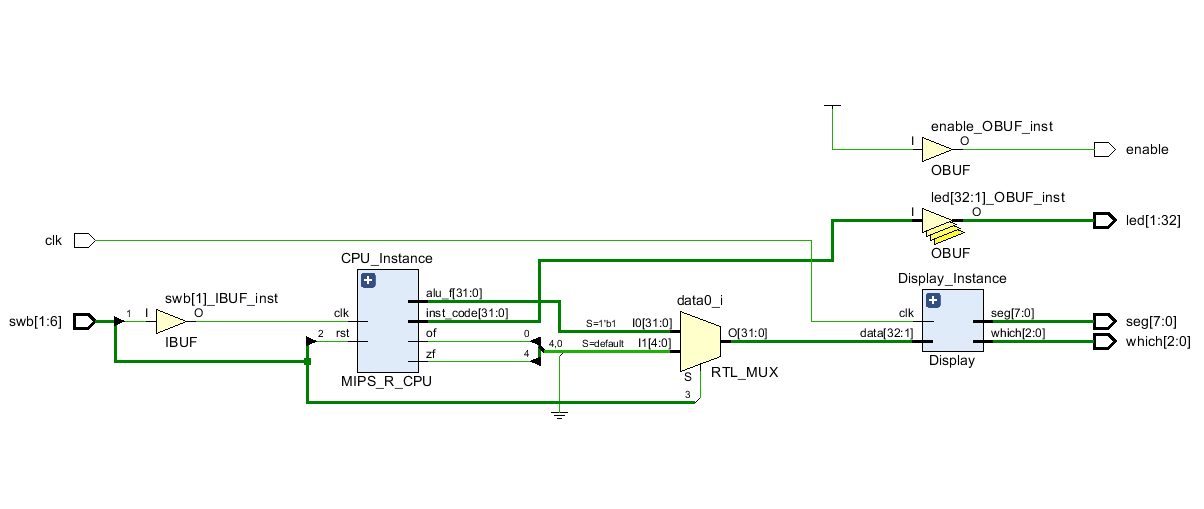


1. 仿真结果分析（对仿真波形进行分析）

这是一个单周期MIPS CPU，所有指令在一个周期内全部完成，并且可以实现8条R型指令。

1. **电路图**

（开发工具中显示的电路模块图）



1. **引脚配置**

（引脚约束文件的内容，描述主要配置情况）

|  |
| --- |
| # 开启比特流压缩，优化 .bit 文件大小  set\_property BITSTREAM.GENERAL.COMPRESS TRUE [current\_design]  # Switch Button，按钮  set\_property IOSTANDARD LVCMOS18 [get\_ports swb]  set\_property PACKAGE\_PIN R4 [get\_ports {swb[1]}]  set\_property PACKAGE\_PIN AA4 [get\_ports {swb[2]}]  set\_property PACKAGE\_PIN AB6 [get\_ports {swb[3]}]  set\_property PACKAGE\_PIN T5 [get\_ports {swb[4]}]  set\_property PACKAGE\_PIN V8 [get\_ports {swb[5]}]  set\_property PACKAGE\_PIN AA8 [get\_ports {swb[6]}]  # LED  set\_property IOSTANDARD LVCMOS18 [get\_ports led]  set\_property PACKAGE\_PIN R1 [get\_ports {led[1]}]  set\_property PACKAGE\_PIN P2 [get\_ports {led[2]}]  set\_property PACKAGE\_PIN P1 [get\_ports {led[3]}]  set\_property PACKAGE\_PIN N2 [get\_ports {led[4]}]  set\_property PACKAGE\_PIN M1 [get\_ports {led[5]}]  set\_property PACKAGE\_PIN M2 [get\_ports {led[6]}]  set\_property PACKAGE\_PIN L1 [get\_ports {led[7]}]  set\_property PACKAGE\_PIN J2 [get\_ports {led[8]}]  set\_property PACKAGE\_PIN G1 [get\_ports {led[9]}]  set\_property PACKAGE\_PIN E1 [get\_ports {led[10]}]  set\_property PACKAGE\_PIN D2 [get\_ports {led[11]}]  set\_property PACKAGE\_PIN A1 [get\_ports {led[12]}]  set\_property PACKAGE\_PIN L3 [get\_ports {led[13]}]  set\_property PACKAGE\_PIN G3 [get\_ports {led[14]}]  set\_property PACKAGE\_PIN K4 [get\_ports {led[15]}]  set\_property PACKAGE\_PIN G4 [get\_ports {led[16]}]  set\_property PACKAGE\_PIN K1 [get\_ports {led[17]}]  set\_property PACKAGE\_PIN J1 [get\_ports {led[18]}]  set\_property PACKAGE\_PIN H2 [get\_ports {led[19]}]  set\_property PACKAGE\_PIN G2 [get\_ports {led[20]}]  set\_property PACKAGE\_PIN F1 [get\_ports {led[21]}]  set\_property PACKAGE\_PIN E2 [get\_ports {led[22]}]  set\_property PACKAGE\_PIN D1 [get\_ports {led[23]}]  set\_property PACKAGE\_PIN B1 [get\_ports {led[24]}]  set\_property PACKAGE\_PIN B2 [get\_ports {led[25]}]  set\_property PACKAGE\_PIN N3 [get\_ports {led[26]}]  set\_property PACKAGE\_PIN M3 [get\_ports {led[27]}]  set\_property PACKAGE\_PIN K3 [get\_ports {led[28]}]  set\_property PACKAGE\_PIN H3 [get\_ports {led[29]}]  set\_property PACKAGE\_PIN N4 [get\_ports {led[30]}]  set\_property PACKAGE\_PIN L4 [get\_ports {led[31]}]  set\_property PACKAGE\_PIN J4 [get\_ports {led[32]}]  # 数码管相关  set\_property IOSTANDARD LVCMOS18 [get\_ports seg]  set\_property PACKAGE\_PIN H19 [get\_ports {seg[7]}]  set\_property PACKAGE\_PIN G20 [get\_ports {seg[6]}]  set\_property PACKAGE\_PIN J22 [get\_ports {seg[5]}]  set\_property PACKAGE\_PIN K22 [get\_ports {seg[4]}]  set\_property PACKAGE\_PIN K21 [get\_ports {seg[3]}]  set\_property PACKAGE\_PIN H20 [get\_ports {seg[2]}]  set\_property PACKAGE\_PIN H22 [get\_ports {seg[1]}]  set\_property PACKAGE\_PIN J21 [get\_ports {seg[0]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports which]  set\_property PACKAGE\_PIN N22 [get\_ports {which[0]}]  set\_property PACKAGE\_PIN M21 [get\_ports {which[1]}]  set\_property PACKAGE\_PIN M22 [get\_ports {which[2]}]  set\_property -dict {IOSTANDARD LVCMOS18 PACKAGE\_PIN L21} [get\_ports enable]  set\_property -dict {IOSTANDARD LVCMOS18 PACKAGE\_PIN H4} [get\_ports clk]  # [Place 30-574] Poor placement for routing between an IO pin and BUFG.If this  # sub optimal condition is acceptable for this design, you may use the  # CLOCK\_DEDICATED\_ROUTE constraint in the .xdc file to demote this message to a  # WARNING. However, the use of this override is highly discouraged.  set\_property CLOCK\_DEDICATED\_ROUTE FALSE [get\_nets clk\_IBUF]  set\_property CLOCK\_DEDICATED\_ROUTE FALSE [get\_nets swb\_IBUF[1]]  set\_property CLOCK\_DEDICATED\_ROUTE FALSE [get\_nets swb\_IBUF[2]]  set\_property CLOCK\_DEDICATED\_ROUTE FALSE [get\_nets swb\_IBUF[3]]  # set\_property CLOCK\_DEDICATED\_ROUTE FALSE [get\_nets swb\_IBUF[4]]  # set\_property CLOCK\_DEDICATED\_ROUTE FALSE [get\_nets swb\_IBUF[5]]  # set\_property CLOCK\_DEDICATED\_ROUTE FALSE [get\_nets swb\_IBUF[6]] |

5、思考与探索：必做（1）

（1）将测试程序（≥16条）的各条指令执行的结果和标志记录到表中，分析结果正确与否？如果不正确，请分析原因。

答：见下表。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 指令编号 | 指令结果 | ZF | OF |
| 1 | 00000827 | FFFF\_FFFF | 0 | 0 |
| 2 | 0001102b | 0000\_0001 | 0 | 0 |
| 3 | 00421820 | 0000\_0002 | 0 | 0 |
| 4 | 00622020 | 0000\_0003 | 0 | 0 |
| 5 | 00832820 | 0000\_0005 | 0 | 0 |
| 6 | 00a33020 | 0000\_0007 | 0 | 0 |
| 7 | 00463804 | 0000\_000E | 0 | 0 |
| 9 | 00a64820 | 0000\_000C | 0 | 0 |
| 8 | 01264004 | 0000\_7000 | 0 | 0 |
| 9 | 00284826 | FFFF\_8FFF | 0 | 0 |
| 10 | 01215020 | FFFF\_8FFE | 0 | 0 |
| 11 | 01075822 | 0000\_6FF2 | 0 | 0 |
| 12 | 00e86022 | FFFF\_900E | 0 | 0 |
| 13 | 012c6824 | FFFF\_800E | 0 | 0 |
| 14 | 012c7025 | FFFF\_9FFF | 0 | 0 |
| 15 | 00c77825 | 0000\_000F | 0 | 0 |
| 16 | 00c78027 | FFFF\_FFF0 | 0 | 0 |
| 17 | 00e38820 | 0000\_0010 | 0 | 0 |
| 18 | 02289004 | 7000\_0000 | 0 | 0 |
| 19 | 02239804 | 0002\_0000 | 0 | 0 |
| 20 | 00f3a004 | 8000\_0000 | 0 | 0 |
| 21 | 0281a820 | 7FFF\_FFFF | 0 | 1 |
| 22 | 0255b025 | 7FFF\_FFFF | 0 | 0 |
| 23 | 0296b820 | FFFF\_FFFF | 0 | 0 |
| 24 | 0296c022 | 0000\_0001 | 0 | 1 |
| 25 | 02d4c822 | FFFF\_FFFF | 0 | 1 |
| 26 | 0241d026 | 8FFF\_FFFF | 0 | 0 |
| 27 | 0241d026 | 0000\_0001 | 0 | 0 |
| 28 | 02d4d82b | 0000\_0000 | 1 | 0 |
| 29 | 0354e02b | 8000\_0000 | 0 | 1 |
| 30 | 02c2e820 | 1. 7FFF\_FFFF | 0 | 1 |
| 31 | 017af820 | 9000\_6FF1 | 0 | 0 |

（2）sll rd, rt, shamt指令将rt寄存器的数据进行逻辑左移，左移的位数则是由字段shamt指定。试着实现该指令，谈谈你的实现方法。

（3）本实验实现的sltu指令是对无符号数的比较置位指令，如果需要实现有符号数的比较置位指令——slt指令，请问应该如何实现？

（4）srav是对（有符号）数据的算术右移指令，考虑如何实现它？

（5）说说你在实验中碰到了哪些问题，你是如何解决的？