南京理工大学

NANJING UNIVERSITY OF SCIENCE & TECHNOLOGY

南京理工大学计算机科学与工程学院

硬件课程设计（I）报告

班　　级 9191069501

学生姓名 王子辉

学　　号 9191160D0236

指导教师 杜姗姗

一、 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。

二、 实验设备

1、系统环境：Windows10
2、编程语言：Verilog
3、IDE：Vivado 2019.2

三、实验任务

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。

1、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。

2、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。

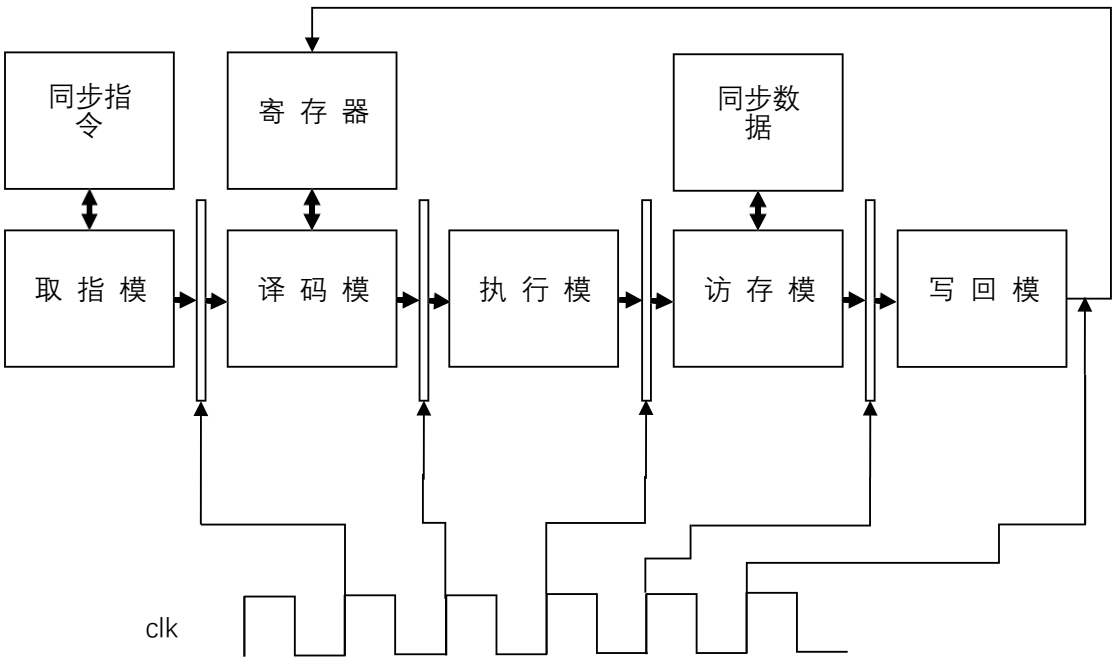
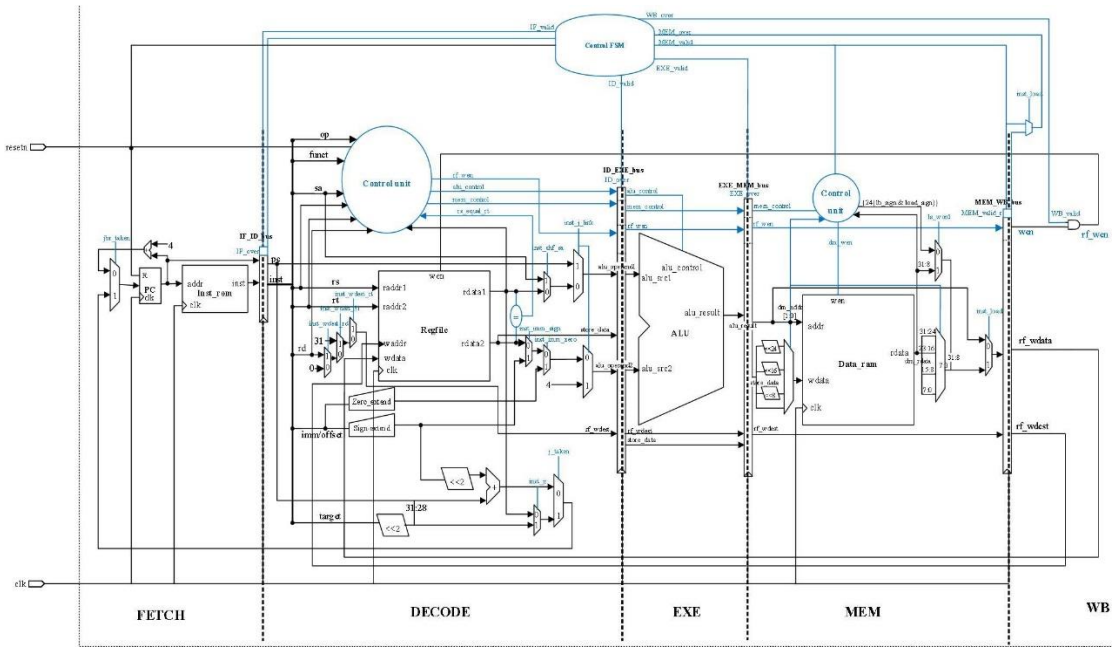


图 1　多周期CPU 的大致框图

3、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。

4、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。

四、实验整体设计框图



五、实现的指令归纳表

```
//------------- 指令编码 ---------|指令地址|----------- 汇编指令 -----------|- 指令结果 -----//
assign inst_rom[0] = 32'h3c010000;  //(00) | main:    lui $1, #0            | $1 = 0000_0000H
assign inst_rom[1] = 32'h34240000;  //(04) |          ori $4, $1, #0x00     | $4 = 0000_0000H
assign inst_rom[2] = 32'h24050004;  //(08) |          addiu $5, $0, #4      | $5 = 0000_0004H
assign inst_rom[3] = 32'h0c000018;  //(0C) | call:    jal sum               | pc = 0000_0060H

assign inst_rom[4] = 32'hac820000;  //(10) |          sw $2, #0($4)         | Mem[10H] = 0000_000AH
assign inst_rom[5] = 32'h8c890000;  //(14) |          lw $9, #0($4)         | $9 = 0000_000AH
assign inst_rom[6] = 32'h01244023;  //(18) |          subu $8, $9, $4       | $8 =  FFFF_FFFAH (-6D)
assign inst_rom[7] = 32'h24050003;  //(1C) |          addiu $5, $0, #3      | $5 = 0000_0003H
assign inst_rom[8] = 32'h24a5ffff;  //(20) | loop2:   addiu $5, $5, #-1     | $5 = $5 - 1
assign inst_rom[9]  = 32'h34a8ffff; //(24) |          ori $8, $5, #0xffff   | $8 = 0000_FFFFH
assign inst_rom[10] = 32'h39085555; //(28) |          xori $8, $8, #0x5555  | $8 = 0000_AAAAH
assign inst_rom[11] = 32'h2409ffff; //(2C) |          addiu $9, $0, #-1     | $9 = FFFF_FFFFH
assign inst_rom[12] = 32'h312affff; //(30) |          andi $10, $9, #0xffff | $10 = 0000_FFFFH
assign inst_rom[13] = 32'h01493025; //(34) |          or $6, $10, $9        | $6 = FFFF_FFFFH
assign inst_rom[14] = 32'h01494026; //(38) |          xor $8, $10, $9       | $8 = FFFF_0000H
assign inst_rom[15] = 32'h01463824; //(3C) |          and $7, $10, $6       | $7 = 0000_FFFFH
assign inst_rom[16] = 32'h10a00002; //(40) |          beq $5, $0, shift     | if $5 == 0: pc = 0000_0048H
assign inst_rom[17] = 32'h08000008; //(44) |          j loop2               | pc = 0000_0020H
assign inst_rom[18] = 32'h2405ffff; //(48) | shift:   addiu $5, $0, #-1     | $5 = FFFF_FFFFH
assign inst_rom[19] = 32'h000543c0; //(4C) |          sll $8, $5, #15       | $8 = FFFF_8000H
assign inst_rom[20] = 32'h00084400; //(50) |          sll $8, $8, #16       | $8 = 8000_0000H
assign inst_rom[21] = 32'h00084403; //(54) |          sra $8, $8, #16       | $8 = FFFF_8000H
assign inst_rom[22] = 32'h000843c2; //(58) |          srl $8, $8, #15       | $8 = 0001_FFFFH
assign inst_rom[23] = 32'h08000017; //(5C) | finish:  j finish              | pc = 0000_005CH

assign inst_rom[24] = 32'h00004021; //(60) | sum:     addu $8, $0, $0       | $8 = 0000_0000H
assign inst_rom[25] = 32'h8c890000; //(64) | loop1:   lw $9, #0($4)         | $9 = Mem[00H] = 0000_0001H
assign inst_rom[26] = 32'h24840004; //(68) |          addiu $4, $4, #4      | $4 = $4 + 4
assign inst_rom[27] = 32'h01094021; //(6C) |          addu $8, $8, $9       | $8 = $8 + $9
assign inst_rom[28] = 32'h24a5ffff; //(70) |          addiu $5, $5, #-1     | $5 = $5 - 1
assign inst_rom[29] = 32'h14a0fffc; //(74) |          bne $5, $0, loop1     | if $5 != 0: pc = 0000_0064H
assign inst_rom[30] = 32'h00081000; //(78) |          sll $2, $8, #0        | $2 = 0000_000AH
assign inst_rom[31] = 32'h03e00008; //(7c) |          jr $31                | pc = 0000_0010H
```

六、Verilog 程序代码

1、取指 Fetch：st1_fetch.v

```
`timescale 1ps / 1ps
`define STARTADDR 32'd0
module st1_fetch (    // 取指
        input clk,    // 时钟信号
        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 {jbr_taken, jbr_target}
        output [31:0] inst_addr,    // 发往 inst_rom 的取值地址
        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
```

```
);
   //pc begin
   wire [31:0] next_pc;   // 下一指令地址
   wire [31:0] seq_pc;   // 非跳转（顺序执行）的下一指令地址
   reg [31:0] pc;   // 程序计数器 pc
   // 跳转 pc
   wire jbr_taken;   // 跳转信号
   wire [31:0] jbr_target;   // 跳转地址
   assign {jbr_taken, jbr_target} = jbr_bus;   // 跳转总线

   assign seq_pc[31:2] = pc[31:2] + 1'b1;   // 顺序执行的下一指令地址：
b<PC>=b<PC>+b100
   assign seq_pc[1:0] = pc[1:0];

   // 新指令：若指令跳转，为跳转地址；否则为下一条指令
   assign next_pc = jbr_taken ? jbr_target : seq_pc;

   always @(posedge clk) begin    // pc 程序计数器
      if (!resetn) begin
         pc <= `STARTADDR;   // 复位，取程序起始地址
      end else if (next_fetch) begin   // 锁存 pc 值
         pc <= next_pc;   // 不复位，取新指令
      end
   end
   // pc end

   // to instrom begin
   assign inst_addr = pc;
   // to instrom end

   // IF 执行完成  begin
   // 由于 inst_rom 为同步读的
   // 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
   // 故取值模块需要两拍时间
   // 将 IF_valid 锁存一拍即是 IF_over 信号
   always @(posedge clk) begin   // 同步读
      // always @(*) begin   // 异步读
      IF_over <= IF_valid;   // 非阻塞赋值  此时 IF_valid 还是上一时刻值 0
   end
   // 如果 inst_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

    // display IF_pc、IF_inst
    assign IF_pc = pc;
    assign IF_inst = inst;
    // display end
endmodule
```

2、译码 Decode：st2_decode.v

```
`timescale 1ps / 1ps
module st2_decode (    // 译码
        input ID_valid,    // 译码级有效信号
        input [63:0] IF_ID_bus_r,    // IF->ID 总线
        input [31:0] rs_value,    // 源操作数 1 数值
        input [31:0] rt_value,    // 源操作数 2 数值
        output [4:0] rs,    // 源操作数 1 地址
        output [4:0] rt,    // 源操作数 2 地址
        output [32:0] jbr_bus,    // 跳转总线  {jbr_taken, jbr_target}
        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 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 end

    //inst decode 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 信号量
    wire sa_zero;   // sa 域 全 0 信号量
    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

    // 指令分类：
    // 1、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;   // 两种寄存器跳转指令
    assign inst_j_link = inst_JAL | inst_JALR;   // 两种链接跳转指令
    // 全部非 link 类跳转指令
    assign jbr_not_link = inst_J | inst_JR | inst_BEQ | inst_BNE | inst_BGEZ | inst_BGTZ | inst_BLEZ | inst_BLTZ;

    // 2、load / store
    wire inst_load;
    wire inst_store;
    assign inst_load  = inst_LW | inst_LB | inst_LBU;   // load 型指令
    assign inst_store = inst_SW | inst_SB;   // store 型指令

    // 3、alu 操作分类（12 个操作）
    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;//做加法的指令
    assign inst_sub = inst_SUBU;   // 做减法的指令
    assign inst_slt = inst_SLT | inst_SLTI;   // 有符号小于置位
    assign inst_sltu = inst_SLTIU | inst_SLTU;   // 无符号小于置位
    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

    // 4、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;

    // 5、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;

    // 6、依据目的寄存器号分类
    wire inst_wdset_rt;   // 寄存器堆写入地址为 rt 的指令
    wire inst_wdset_31;   // 寄存器堆写入地址为 31 的指令
    wire inst_wdset_rd;   // 寄存器堆写入地址为 rd 的指令
    assign inst_wdset_rt = inst_imm_zero | inst_ADDIU | inst_SLTI | inst_SLTIU | inst_load;
    assign inst_wdset_31 = inst_JAL;
    assign inst_wdset_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;
    // inst decode 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 寄存器值等于 0
assign rs_ltz = rs_value[31];   // rs 寄存器值小于 0
wire br_taken;
wire [31:0] br_target;
// 相等跳转 | 不等跳转 | 大于等于 0 跳转 | 大于 0 跳转 | 小于等于 0 跳转 | 小于 0 跳转
assign br_taken= (inst_BEQ & rs_equql_rt) | (inst_BNE & ~rs_equql_rt) | (inst_BGEZ & ~rs_ltz) | (inst_BGTZ & ~rs_ltz & ~rs_ez) | (inst_BGTZ & (rs_ltz | rs_ez)) | (inst_BLTZ & rs_ltz);

// 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];

// 判断 jump or 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;

// IF->ID 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 = {   // ALU 操作码，独热编码
            inst_add,
            inst_sub,
            inst_slt,
            inst_sltu,
            inst_and,
            inst_nor,
            inst_or,
            inst_xor,
            inst_sll,
            inst_srl,
            inst_sra,
            inst_lui
        };
  // 仿存需要用到的 load / store 信息
  wire lb_sign;   // load 一字节为有符号 load
  wire Is_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 Is_word = inst_LW | inst_SW;
  assign mem_control = {inst_load, inst_store, Is_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_wdset_rt | inst_wdset_31 | inst_wdset_rd;
  assign rf_wdest = inst_wdset_rt ? rt : (inst_wdset_31 ? 5'd31 : (inst_wdset_rd ? rd : 5'd0));// 在不写寄存器堆时，设置为 0
  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

    // display ID_pc begin
    assign ID_pc = pc;
    // display ID_pc end
endmodule
```

3、执行 Execute：st3_exe.v

```
`timescale 1ps / 1ps
`include "alu.v"
module st3_exe (    //  执行
        input EXE_valid,    //  执行级有效信号
        input [149:0] ID_EXE_bus_r,    //ID->EXE 总线
        output EXE_over,
        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;

    //  仿存需要用到的  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;
    wire [4:0] rf_wdest;

    // 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 end

    // ALU begin
    wire [31:0] alu_result;
```

```
    // 调用 alu 模块
    alu alu_module (
            .alu_control(alu_control),    // input, 12, ALU 控制信号
            .alu_src1(alu_operand1),    // input, 32, ALU 操作数 1
            .alu_src2(alu_operand2),    // input, 32, ALU 操作数 2
            .alu_result(alu_result)    // output, 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
    // 且所以 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 值
    };
    // EXE->MEM 总线 end

    // display EXE_pc begin
    assign EXE_pc = pc;
    // display EXE_pc end
endmodule
```

4、访存 Memory：st4_mem.v

```
`timescale 1ps / 1ps
module st4_mem (    // 仿存
        input clk,    // 时钟
        input MEM_valid,    // 仿存级有效信号
        input [105:0] EXE_MEM_bus_r,    //EXE->MEM 总线
        input [31:0] dm_rdata,    // 仿存读数据
        output [31:0] dm_addr,    // 仿存读写数据
        output reg [3:0] dm_wen,    // 仿存写使能
        output reg [31:0] dm_wdata,    // 仿存写数据
```

```
    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
  //  仿存需要用到的  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  仿存  begin
  wire inst_load;    // load 操作
  wire inst_store;    // store 操作
  wire Is_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, Is_word, lb_sign} = mem_control;

  //  仿存读写地址
  assign dm_addr = alu_result;

  // store 操作的写使能
  always @(*) begin
    if (MEM_valid && inst_store) begin    //仿存级有效时  且为 store 操作
      if (Is_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 @(*) begin// 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
    if (!Is_word) 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 else begin
        dm_wdata <= store_data;
    end
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] = Is_word ? dm_rdata : {24{lb_sign & load_sign}};
//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] = Is_word ? dm_rdata[31:8] : {24{lb_sign & load_sign}};
    // load / store 仿存 end

    // MEM 执行完成 begin
    // 由于 data_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    // 同步读
        // always @(*) begin    // 异步读
        MEM_valid_r <= MEM_valid;
    end
    assign MEM_over = inst_load ? MEM_valid_r : MEM_valid;
    // 如果 data_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 end

    // display MEM_pc begin
    assign MEM_pc = pc;
    //display MEM_pc end
endmodule
```

5、写回 Write Back：st5_wb.v

```
`timescale 1ps / 1ps
module st5_wb (    // 写回
        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 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 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

  // display WB_pc begin
  assign WB_pc = pc;
  // display WB_pc end
endmodule
```

6、加法器 Adder：adder.v

```
`timescale 1ps / 1ps
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、运算器 ALU：alu.v

```
`timescale 1ps / 1ps
`include "adder.v"
module alu (    // ALU 模块，可进行 12 种操作
    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;
    assign or_result    = alu_src1 | alu_src2;
    assign nor_result = ~or_result;
    assign xor_result = alu_src1 ^ alu_src2;
    assign lui_result = {alu_src2[15:0], 16'd0};    // src2 低 16 位装载至高 16 位

    //  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;    //  判断做加法还是减法
    assign adder_cin = ~alu_add;    //  减法需要 cin，因为负数的补码是取反加 1，前面已将 alu_src2 按位取反

    //  调用加法器模快
    adder adder_module (
        .operand1(adder_operand1),    // input, 32
        .operand2(adder_operand2),    // input, 32
        .cin(adder_cin),    // input, 1
        .result(adder_result),    // output, 32
        .cout(adder_cout)    // output, 1
    );

    //  加减结果
    assign add_sub_result = adder_result;

    // slt 结果
    assign slt_result = adder_result[31] ? 1'b1 : 1'b0;
```

```
    // sltu 结果
    assign sltu_result = adder_cout ? 1'b0 : 1'b1;   //无符号数小于置位


    // 逻辑左移
    assign sll_result = alu_src2 << alu_src1;

    // 逻辑右移
    assign srl_result = alu_src2 >> alu_src1;

    // 算术右移
    wire signed [31:0] temp_src2;   //带符号数的临时变量
    assign temp_src2    = alu_src2;
    assign sra_result = temp_src2 >>> alu_src1;

    // 选择相应结果输出
    reg [31:0] alu_result_r;
    always @(*) begin
        if (alu_add | alu_sub) alu_result_r <= add_sub_result;
        else if (alu_slt) alu_result_r <= slt_result;
        else if (alu_sltu) alu_result_r <= sltu_result;
        else if (alu_and) alu_result_r <= and_result;
        else if (alu_nor) alu_result_r <= nor_result;
        else if (alu_or) alu_result_r <= or_result;
        else if (alu_xor) alu_result_r <= xor_result;
        else if (alu_sll) alu_result_r <= sll_result;
        else if (alu_srl) alu_result_r <= srl_result;
        else if (alu_sra) alu_result_r <= sra_result;
        else if (alu_lui) alu_result_r <= lui_result;
    end
    assign alu_result = alu_result_r;
endmodule
```

8、寄存器堆 Regfile：regfile.v

```
`timescale 1ns / 1ps
module regfile (
        input clk,    // 时钟
        input wen,    // 使能信号  1: 写; 0: 读
        input [4:0] raddr1,    // 读端口 1 地址
        input [4:0] raddr2,    // 读端口 2 地址
        input [4:0] waddr,    // 写端口地址
        input [31:0] wdata,    // 写数据
        output reg [31:0] rdata1,    // 读端口 1 数据
```

```
    output reg [31:0] rdata2,   // 读端口 2 数据

    // display rf
    input   [ 4:0] rf_addr,
    output [31:0] rf_data
);


  reg [31:0] regfile[31:0];
  initial begin
      regfile[0] <= 0;
  end


  always @(posedge clk) begin   // 写数据 同步
    if (wen) begin
      regfile[waddr] <= wdata;
    end
  end

  always @(*) begin   // 读数据 异步
  // always @(posedge clk) begin   // 读数据 同步
    if (raddr1 >= 0 && raddr1 < 32) begin
      rdata1 <= regfile[raddr1];
    end else begin
      rdata1 <= 32'd0;
    end

    if (raddr2 >= 0 && raddr2 < 32) begin
      rdata2 <= regfile[raddr2];
    end else begin
      rdata2 <= 32'd0;
    end
  end

  // display rf begin
  assign rf_data = regfile[rf_addr];
  // display rf end
endmodule
```

9、数据存储器 Data RAM：data_ram.v

```
`timescale 1ns / 1ps
module data_ram (   // 数据存储模块 同步读写
    input clk,   // 时钟
```

```
    input [3:0] wen,   // 字节写使能
    input [4:0] addr,   // 地址
    input [31:0] wdata,   // 写数据
    output reg [31:0] rdata,   // 读数据

    //调试端口，用于读出数据显示
    input clka,
    input [3:0] wea,
    input [7:0] addra,
    output reg [31:0] rdataa,
    input [31:0] wdataa
    // input [4:0] test_addr,
    // output reg [31:0] test_data
);
  reg [31:0] DM[31:0];   //数据存储器，字节地址  7'b000_0000~7'b111_1111

  integer i;
  initial begin
    for (i = 0; i < 32; i = i + 1) DM[i] <= i;
  end

  //写数据  begin
  always @(posedge clk) begin   // 当写控制信号为 1  数据写入内存
    if (wen[3]) begin
      DM[addr][31:24] <= wdata[31:24];
    end
  end
  always @(posedge clk) begin
    if (wen[2]) begin
      DM[addr][23:16] <= wdata[23:16];
    end
  end
  always @(posedge clk) begin
    if (wen[1]) begin
      DM[addr][15:8] <= wdata[15:8];
    end
  end
  always @(posedge clk) begin
    if (wen[0]) begin
      DM[addr][7:0] <= wdata[7:0];
    end
  end
  // 写数据  end
```

```
// 读数据 取 4 字节
always @(posedge clk) begin   // 当写控制信号为 0 数据读出内存
   if (wen == 4'b0) begin
      case (addr)
         5'd0:    rdata <= DM[0];
         5'd1:    rdata <= DM[1];
         5'd2:    rdata <= DM[2];
         5'd3:    rdata <= DM[3];
         5'd4:    rdata <= DM[4];
         5'd5:    rdata <= DM[5];
         5'd6:    rdata <= DM[6];
         5'd7:    rdata <= DM[7];
         5'd8:    rdata <= DM[8];
         5'd9:    rdata <= DM[9];
         5'd10: rdata <= DM[10];
         5'd11: rdata <= DM[11];
         5'd12: rdata <= DM[12];
         5'd13: rdata <= DM[13];
         5'd14: rdata <= DM[14];
         5'd15: rdata <= DM[15];
         5'd16: rdata <= DM[16];
         5'd17: rdata <= DM[17];
         5'd18: rdata <= DM[18];
         5'd19: rdata <= DM[19];
         5'd20: rdata <= DM[20];
         5'd21: rdata <= DM[21];
         5'd22: rdata <= DM[22];
         5'd23: rdata <= DM[23];
         5'd24: rdata <= DM[24];
         5'd25: rdata <= DM[25];
         5'd26: rdata <= DM[26];
         5'd27: rdata <= DM[27];
         5'd28: rdata <= DM[28];
         5'd29: rdata <= DM[29];
         5'd30: rdata <= DM[30];
         5'd31: rdata <= DM[31];
      endcase
   end
end


//调试端口 读出特定内存的数据
always @(posedge clka) begin
```

```
        if (wea == 4'b0) begin
            case (addra)
                5'd0:    rdataa <= DM[0];
                5'd1:    rdataa <= DM[1];
                5'd2:    rdataa <= DM[2];
                5'd3:    rdataa <= DM[3];
                5'd4:    rdataa <= DM[4];
                5'd5:    rdataa <= DM[5];
                5'd6:    rdataa <= DM[6];
                5'd7:    rdataa <= DM[7];
                5'd8:    rdataa <= DM[8];
                5'd9:    rdataa <= DM[9];
                5'd10: rdataa <= DM[10];
                5'd11: rdataa <= DM[11];
                5'd12: rdataa <= DM[12];
                5'd13: rdataa <= DM[13];
                5'd14: rdataa <= DM[14];
                5'd15: rdataa <= DM[15];
                5'd16: rdataa <= DM[16];
                5'd17: rdataa <= DM[17];
                5'd18: rdataa <= DM[18];
                5'd19: rdataa <= DM[19];
                5'd20: rdataa <= DM[20];
                5'd21: rdataa <= DM[21];
                5'd22: rdataa <= DM[22];
                5'd23: rdataa <= DM[23];
                5'd24: rdataa <= DM[24];
                5'd25: rdataa <= DM[25];
                5'd26: rdataa <= DM[26];
                5'd27: rdataa <= DM[27];
                5'd28: rdataa <= DM[28];
                5'd29: rdataa <= DM[29];
                5'd30: rdataa <= DM[30];
                5'd31: rdataa <= DM[31];
            endcase
        end
    end
endmodule
```

10、指令寄存器 Instruction ROM：inst_rom.v

```
`timescale 1ps / 1ps
module inst_rom (    // 指令寄存器模块  同步读
        input clk,
        input [7:0] addr,
```

```
    output [31:0] inst
);
  wire [31:0] inst_rom[49:0];      // 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'h3c010000;   //(00) | main:      lui $1, #0                | $1 = 0000_0000H
  assign inst_rom[1] = 32'h34240000;   //(04) |             ori $4, $1, #0x00        | $4 = 0000_0000H
  assign inst_rom[2] = 32'h24050004;   //(08) |             addiu $5, $0, #4         | $5 = 0000_0004H
  assign inst_rom[3] = 32'h0c000018;   //(0C) | call:      jal sum                   | pc = 0000_0060H

  assign inst_rom[4] = 32'hac820000;   //(10) |                  sw $2, #0($4)      | Mem[10H] = 0000_000AH
  assign inst_rom[5] = 32'h8c890000;   //(14) |                  lw $9, #0($4)      | $9 = 0000_000AH
  assign inst_rom[6] = 32'h01244023;   //(18) |             subu $8, $9, $4          | $8 =   FFFF_FFFAH (-6D)
  assign inst_rom[7] = 32'h24050003;   //(1C) |             addiu $5, $0, #3         | $5 = 0000_0003H
  assign inst_rom[8] = 32'h24a5ffff;   //(20) | loop2:     addiu $5, $5, #-1        | $5 = $5 - 1
  assign inst_rom[9]  = 32'h34a8ffff; //(24) |             ori $8, $5, #0xffff     | $8 = 0000_FFFFH
  assign inst_rom[10] = 32'h39085555; //(28) |             xori $8, $8, #0x5555    | $8 = 0000_AAAAH
  assign inst_rom[11] = 32'h2409ffff; //(2C) |             addiu $9, $0, #-1        | $9 = FFFF_FFFFH
  assign inst_rom[12] = 32'h312affff; //(30) |             andi $10, $9, #0xffff | $10 = 0000_FFFFH
  assign inst_rom[13] = 32'h01493025; //(34) |             or $6, $10, $9           | $6 = FFFF_FFFFH
  assign inst_rom[14] = 32'h01494026; //(38) |             xor $8, $10, $9          | $8 = FFFF_0000H
  assign inst_rom[15] = 32'h01463824; //(3C) |             and $7, $10, $6          | $7 = 0000_FFFFH
  assign inst_rom[16] = 32'h10a00002; //(40) |             beq $5, $0, shift        | if $5 == 0: pc = 0000_0048H
  assign inst_rom[17] = 32'h08000008;  //(44) |                  j loop2           | pc = 0000_0020H
  assign inst_rom[18] = 32'h2405ffff; //(48) | shift:     addiu $5, $0, #-1        | $5 =
```

```
FFFF_FFFFH
   assign inst_rom[19] = 32'h000543c0; //(4C) |                sll $8, $5, #15          |
$8 = FFFF_8000H
   assign inst_rom[20] = 32'h00084400; //(50) |                sll $8, $8, #16          |
$8 = 8000_0000H
   assign inst_rom[21] = 32'h00084403; //(54) |                sra $8, $8, #16          |
$8 = FFFF_8000H
   assign inst_rom[22] = 32'h000843c2; //(58) |                srl $8, $8, #15          |
$8 = 0001_FFFFH
   assign inst_rom[23] = 32'h08000017; //(5C) | finish:    j finish                    | pc
= 0000_005CH

   assign inst_rom[24] = 32'h00004021; //(60) | sum:         addu $8, $0, $0          |
$8 = 0000_0000H
   assign inst_rom[25] = 32'h8c890000; //(64) | loop1:      lw $9, #0($4)            |
$9 = Mem[00H] = 0000_0001H
   assign inst_rom[26] = 32'h24840004; //(68) |                addiu $4, $4, #4         |
$4 = $4 + 4
   assign inst_rom[27] = 32'h01094021; //(6C) |                addu $8, $8, $9          |
$8 = $8 + $9
   assign inst_rom[28] = 32'h24a5ffff; //(70) |               addiu $5, $5, #-1        | $5
= $5 - 1
   assign inst_rom[29] = 32'h14a0fffc; //(74) |                bne $5, $0, loop1        | if
$5 != 0: pc = 0000_0064H
   assign inst_rom[30] = 32'h00081000; //(78) |                sll $2, $8, #0           |
$2 = 0000_000AH
   assign   inst_rom[31]   =   32'h03e00008;   //(7c)   |                          jr   $31
| pc = 0000_0010H

   reg [31:0] inst_r;
   always @(posedge clk) begin    // 同步读
      inst_r <= inst_rom[addr];
   end
   assign inst = inst_r;
endmodule
```

11、多周期 CPU：multi_cycle_cpu.v

```
`timescale 1ps / 1ps
`include "st1_fetch.v"
`include "st2_decode.v"
`include "st3_exe.v"
`include "st4_mem.v"
`include "st5_wb.v"
`include "inst_rom.v"
```

```
`include "data_ram.v"
`include "regfile.v"
module multi_cycle_cpu (    //  多周期 cpu
        //  时钟与复位信号
        input clk,
        input resetn, //  后缀"n"代表低电平有效

        // display data
        input    [ 4:0] rf_addr,
        input    [31:0] mem_addr,
        output [31:0] rf_data,
        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
    assign display_state = {29'd0, state};    //展示当前处理机正在处理哪个模块
    //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;    //  仿存
    parameter WB = 3'd5;    //  写回

    always @(posedge clk) begin    //  当前状态
        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 @(*) begin
   case (state)
      IDLE: begin
         next_state = FETCH;   // IDLE->IF
      end
      FETCH: begin
         if (IF_over) begin
            next_state = DECODE;   // IF->ID
         end else begin
            next_state = FETCH;   // continue IF
         end
      end
      DECODE: begin
         if (ID_over) begin   // 如果是非 link 类的分支指令则写回，否则执行
            next_state = jbr_not_link ? FETCH : EXE;   // ID->WB/EXE
         end else begin
            next_state = DECODE;   // continue ID
         end
      end
      EXE: begin
         if (EXE_over) begin
            next_state = MEM;   // EXE->MEM
         end else begin
            next_state = EXE;   // continue EXE
         end
      end
      MEM: begin
         if (MEM_over) begin
            next_state = WB;   // MEM->WB
         end else begin
            next_state = MEM;   // continue MEM
         end
      end
      WB: begin
         if (WB_over) begin
            next_state = FETCH;   // WB->IF
         end else begin
            next_state = WB;   // continue WB
         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->ID 的锁存信号
always @(posedge clk) begin
    if (IF_over) begin
        IF_ID_bus_r <= IF_ID_bus;
    end
end

//ID->EXE 的锁存信号
always @(posedge clk) begin
    if (ID_over) begin
        ID_EXE_bus_r <= ID_EXE_bus;
    end
end

//EXE->MEM 的锁存信号
always @(posedge clk) begin
    if (EXE_over) begin
```

```
        EXE_MEM_bus_r <= EXE_MEM_bus;
      end
    end

    //MEM->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;    //透
    //  其他交互信号  end

    //  各模块实例化  begin
    wire next_fetch;    // next_state_is_fetch  即将运行取值模块，需要先锁存 pc 值
    //  当前状态为 ID，且指令为跳转分支指令（非 link 类），且 ID 执行完成
    //  或者，当前状态为 WB，且 WB 执行完成，则即将进入 IF 状态
    assign next_fetch = (state == DECODE & ID_over & jbr_not_link) | (state == WB
& WB_over);
```

```
st1_fetch IF_module (    // 取指
        .clk(clk),    // input, 1
        .resetn(resetn),    // input, 1
        .IF_valid(IF_valid),    // input, 1
        .next_fetch(next_fetch),    // input, 1
        .inst(inst),    // input, 32
        .jbr_bus(jbr_bus),    // input, 33
        .inst_addr(inst_addr),    // optput, 32
        .IF_over(IF_over),    // output, 1
        .IF_ID_bus(IF_ID_bus),    // output, 64

        // 展示 pc 和取出的指令
        .IF_pc    (IF_pc),
        .IF_inst(IF_inst)
);

st2_decode ID_module (    // 译码
        .ID_valid(ID_valid),    // input, 1
        .IF_ID_bus_r(IF_ID_bus_r),    // input, 64
        .rs_value(rs_value),    // input, 32
        .rt_value(rt_value),    // input, 32
        .rs(rs),    // output, 5
        .rt(rt),    // output, 5
        .jbr_bus(jbr_bus),    // output, 33
        .jbr_not_link(jbr_not_link),    // output, 1
        .ID_over(ID_over),    // output, 1
        .ID_EXE_bus(ID_EXE_bus),    // output, 150

        // 展示 pc
        .ID_pc(ID_pc)
);

st3_exe EXE_module (    // 执行级
        .EXE_valid(EXE_valid),    // input, 1
        .ID_EXE_bus_r(ID_EXE_bus_r),    // input, 150
        .EXE_over(EXE_over),    // output, 1
        .EXE_MEM_bus(EXE_MEM_bus),    // output, 106

        //展示 pc
        .EXE_pc(EXE_pc)
);

st4_mem MEM_module (    // 访存级
```

```
        .clk(clk),    // input, 1
        .MEM_valid(MEM_valid),    // input, 1
        .EXE_MEM_bus_r(EXE_MEM_bus_r),    // input, 106
        .dm_rdata(dm_rdata),    // input, 32
        .dm_addr(dm_addr),    // output, 32
        .dm_wen(dm_wen),    // output, 4
        .dm_wdata(dm_wdata),    // output, 32
        .MEM_over(MEM_over),    // output, 1
        .MEM_WB_bus(MEM_WB_bus),    // output, 70

        //展示 pc
        .MEM_pc(MEM_pc)
    );

    st5_wb WB_module (    //  写回级
        .WB_valid(WB_valid),    // input, 1
        .MEM_WB_bus_r(MEM_WB_bus_r),    // input, 70
        .rf_wen(rf_wen),    // output, 1
        .rf_wdest(rf_wdest),    // output, 5
        .rf_wdata(rf_wdata),    // output, 32
        .WB_over(WB_over),    // output, 1

        //  展示 pc
        .WB_pc(WB_pc)
    );

    inst_rom inst_rom_module (    //  指令存储器
        .clk(clk),    // input, 1 ,时钟
        .addr(inst_addr[9:2]),    // input, 8,  指令地址：pc[9:2]
        .inst(inst)    // output, 32,  指令
    );

    regfile rf_module (    //  寄存器堆模块
        .clk(clk),    // input, 1
        .wen(rf_wen),    // input, 1
        .raddr1(rs),    // input, 5
        .raddr2(rt),    // input, 5
        .waddr(rf_wdest),    // input, 5
        .wdata(rf_wdata),    // input, 32
        .rdata1(rs_value),    // output, 32
        .rdata2(rt_value),    // output, 32

        //display rf
        .rf_addr(rf_addr),    // input, 4
```

```
        .rf_data(rf_data)      // output, 32
    );

    data_ram data_ram_module (    //  数据存储模块
        .clk(clk),    // input, 1,  时钟
        .wen(dm_wen),    // input, 4,  写使能
        .addr(dm_addr[9:2]),    // input, 8,  读地址
        .wdata(dm_wdata),    // input, 32,  写数据
        .rdata(dm_rdata),    // output, 32,  读数据

        //display mem
        .clka(clk),
        .wea(4'd0),
        .addra(mem_addr[9:2]),    // input, 8
        .rdataa(mem_data),    // output, 32
        .wdataa(32'd0)
    );

    //各模块实例化  end
endmodule
```

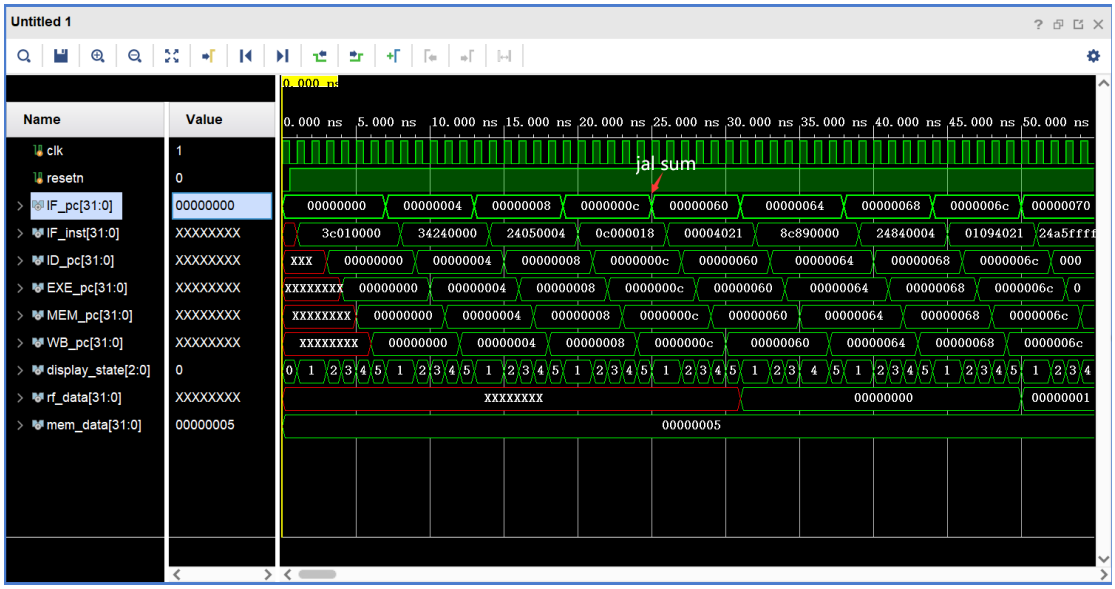
12、验证程序：test_multi_cycle_cpu.v

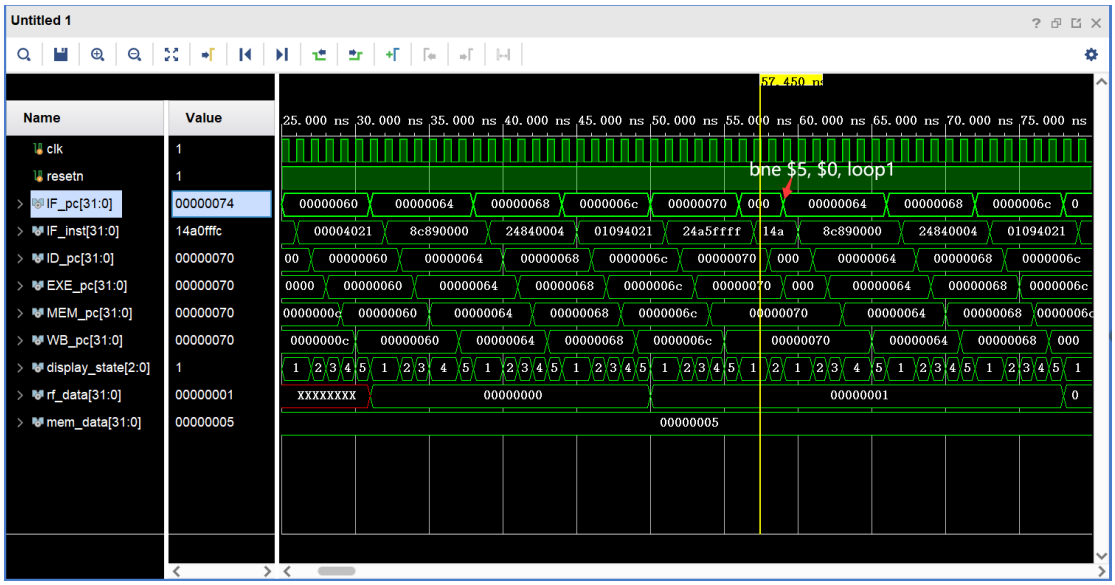
```
`timescale 1ns / 1ps
module test_multi_cycle_cpu ();
    reg clk;
    reg resetn;

    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 [2:0] display_state;
    wire [31:0] rf_data;
    wire [31:0] mem_data;

    multi_cycle_cpu multi_cycle_cpu (
        .clk(clk),
        .resetn(resetn),

        // display data
        .rf_addr(5'd8),    // input
        .mem_addr(32'h10),    // input
```

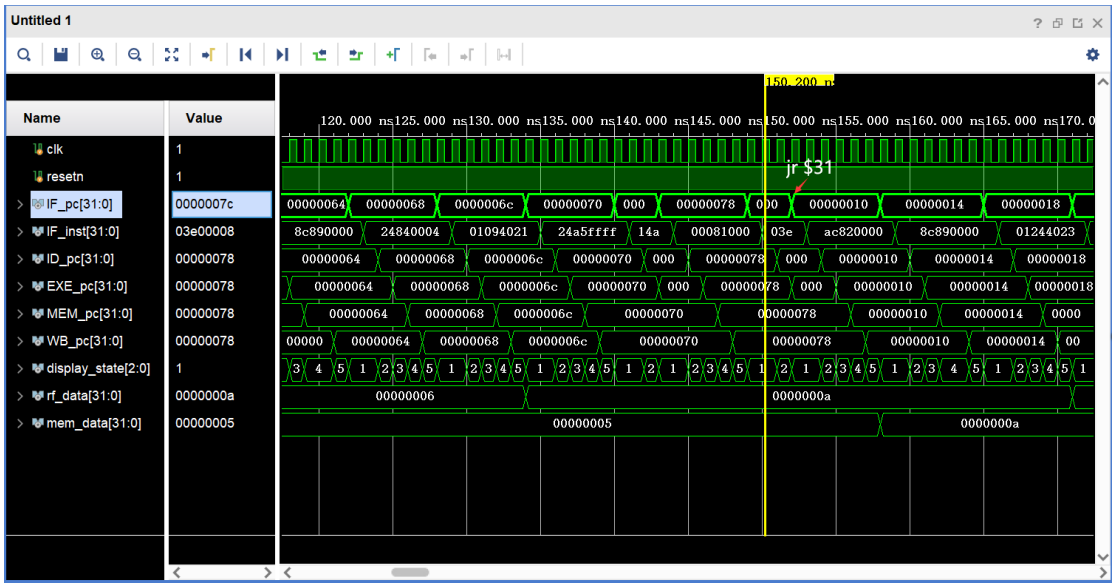
```
        .rf_data(rf_data),    // output...
        .mem_data(mem_data),
        .IF_pc(IF_pc),
        .IF_inst(IF_inst),
        .ID_pc(ID_pc),
        .EXE_pc(EXE_pc),
        .MEM_pc(MEM_pc),
        .WB_pc(WB_pc),
        .display_state(display_state)
    );

    initial begin
        clk    = 1;
        resetn = 0;
    end

    always #0.5 begin
        resetn = 1;
        clk    = ~clk;
    end
    // 测试多周期 cpu 模块结束
endmodule
```

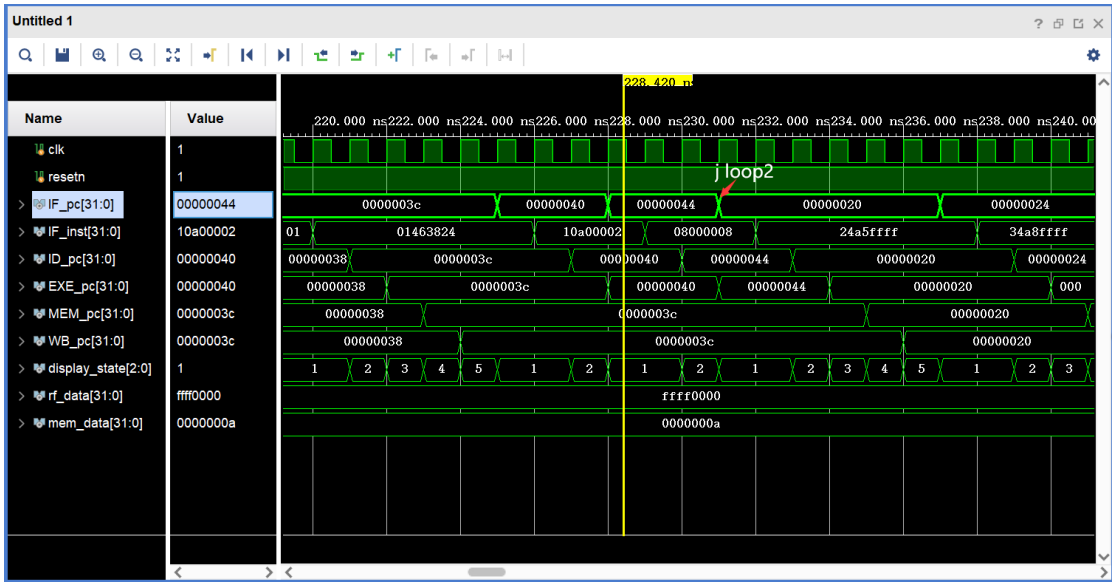
七、仿真波形图

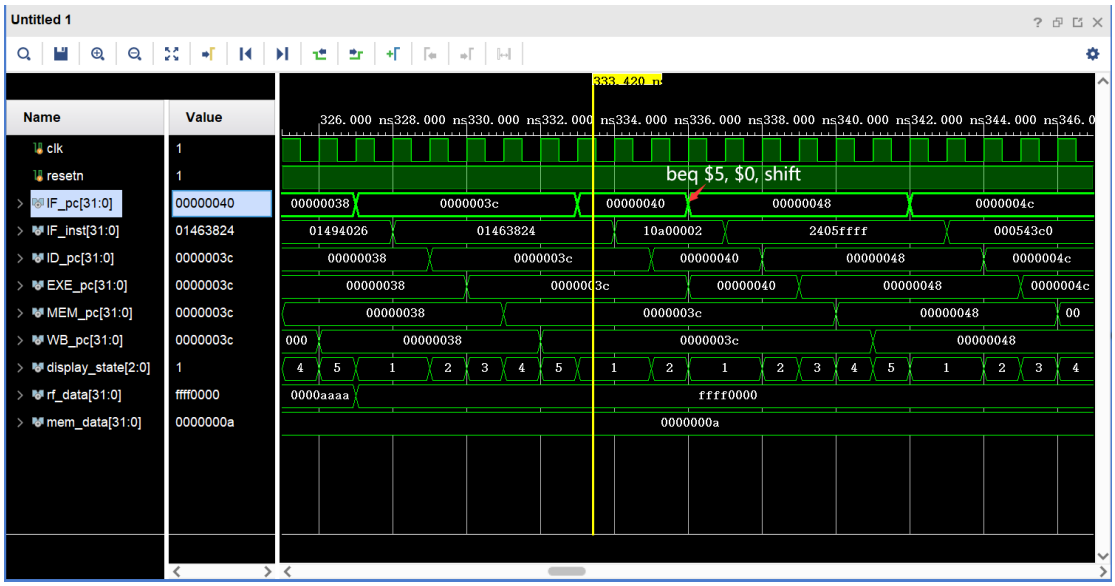
1、函数调用与跳转指令

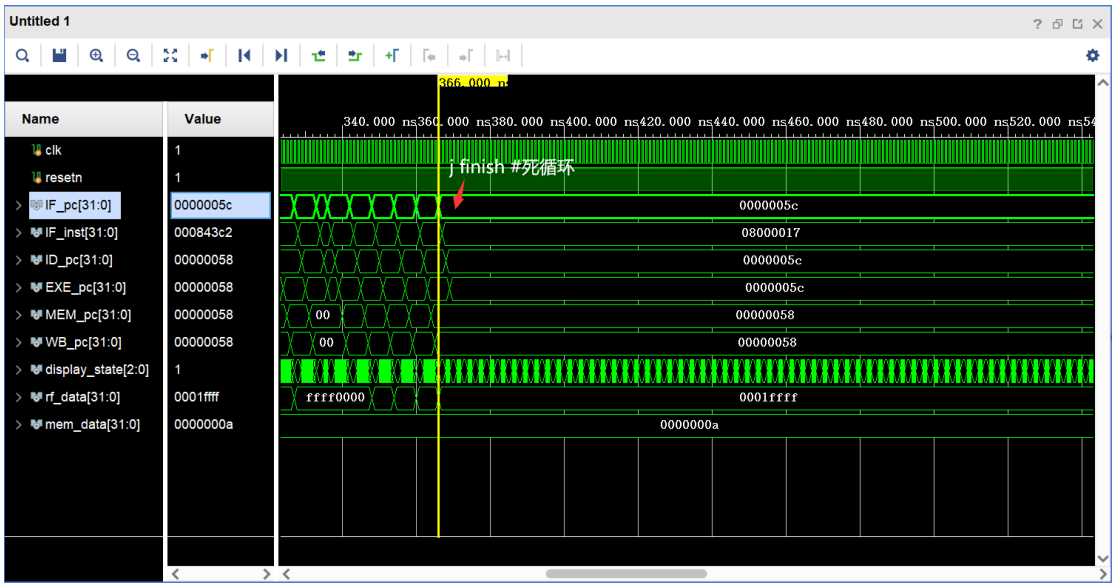



Untitled 1
Name
Value
clk
resetn
IF_pc[31:0]
IF_inst[31:0]
ID_pc[31:0]
EXE_pc[31:0]
MEM_pc[31:0]
WB_pc[31:0]
display_state[2:0]
rf_data[31:0]
mem_data[31:0]
00000074
14a0fffc
00000070
00000070
00000070
00000070
1
00000001
00000005
bne $5, $0, loop1

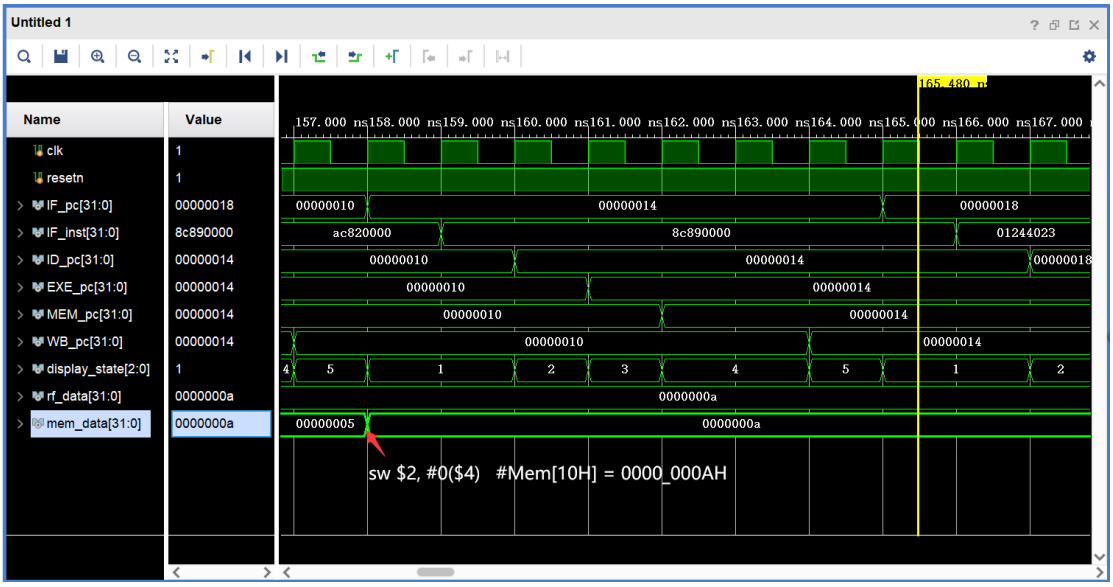

Untitled 1
Name
Value
clk
resetn
IF_pc[31:0]
IF_inst[31:0]
ID_pc[31:0]
EXE_pc[31:0]
MEM_pc[31:0]
WB_pc[31:0]
display_state[2:0]
rf_data[31:0]
mem_data[31:0]
0000007c
03e00008
00000078
00000078
00000078
00000078
1
0000000a
00000005
jr $31


Untitled 1
Name
Value
clk
resetn
IF_pc[31:0]
IF_inst[31:0]
ID_pc[31:0]
EXE_pc[31:0]
MEM_pc[31:0]
WB_pc[31:0]
display_state[2:0]
rf_data[31:0]
mem_data[31:0]
00000044
10a00002
00000040
00000040
0000003c
0000003c
1
ffff0000
0000000a
j loop2

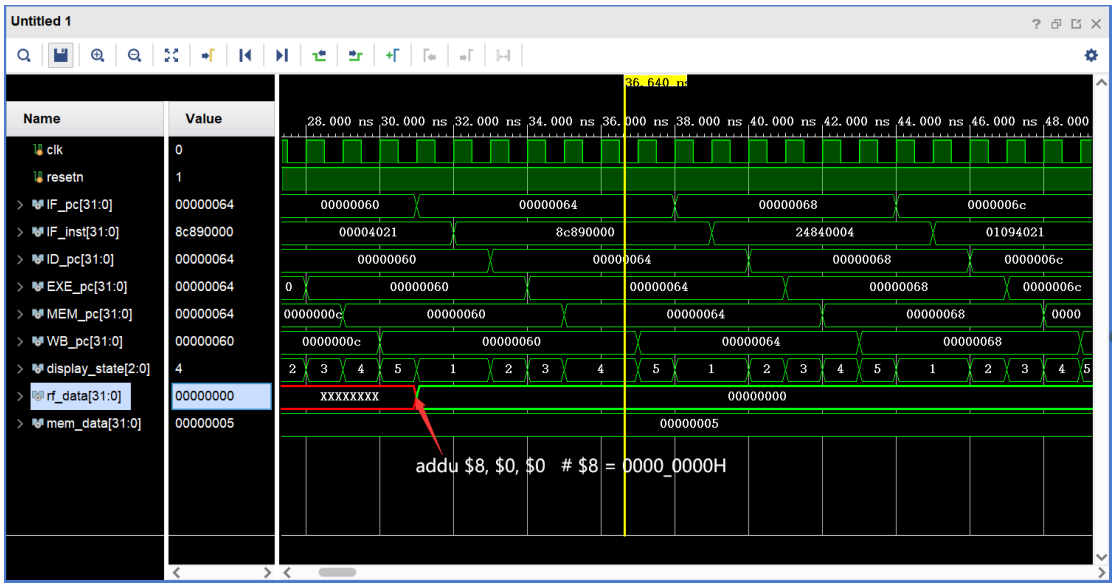

Untitled 1
Name
Value
clk
resetn
IF_pc[31:0]
IF_inst[31:0]
ID_pc[31:0]
EXE_pc[31:0]
MEM_pc[31:0]
WB_pc[31:0]
display_state[2:0]
rf_data[31:0]
mem_data[31:0]
00000040
01463824
0000003c
0000003c
0000003c
0000003c
1
ffff0000
0000000a
beq $5, $0, shift

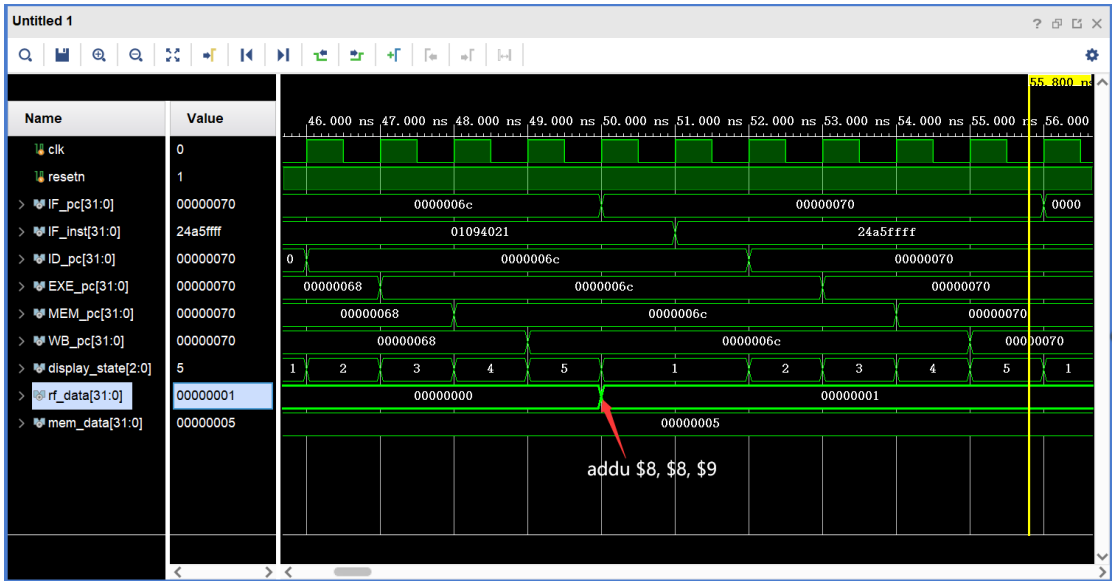

Untitled 1
Name
Value
clk
resetn
IF_pc[31:0]
IF_inst[31:0]
ID_pc[31:0]
EXE_pc[31:0]
MEM_pc[31:0]
WB_pc[31:0]
display_state[2:0]
rf_data[31:0]
mem_data[31:0]
0000005c
000843c2
00000058
00000058
00000058
00000058
1
0001ffff
0000000a
j finish #死循环

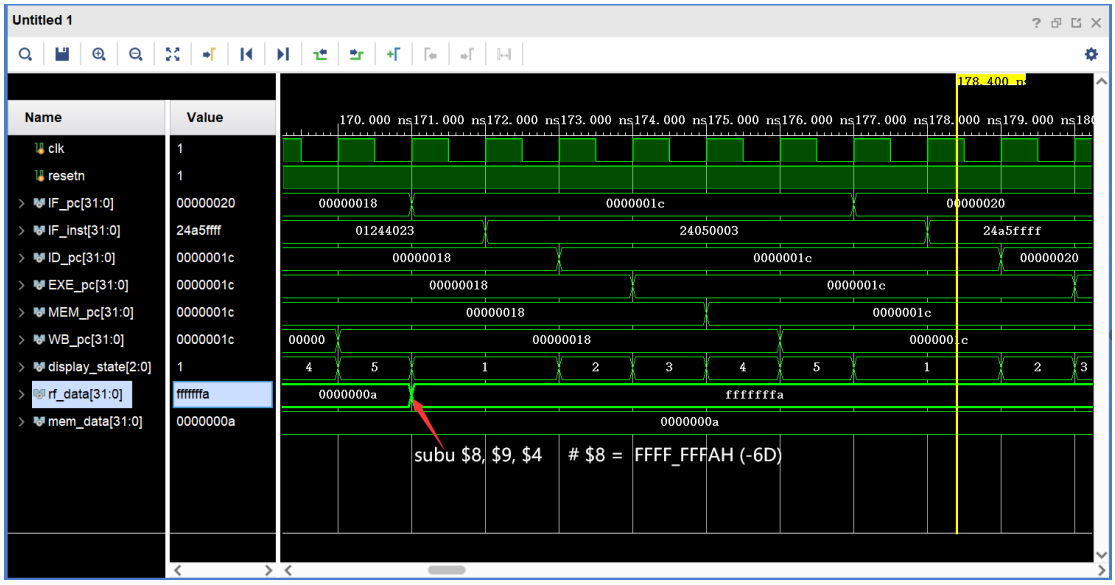
2、访存指令

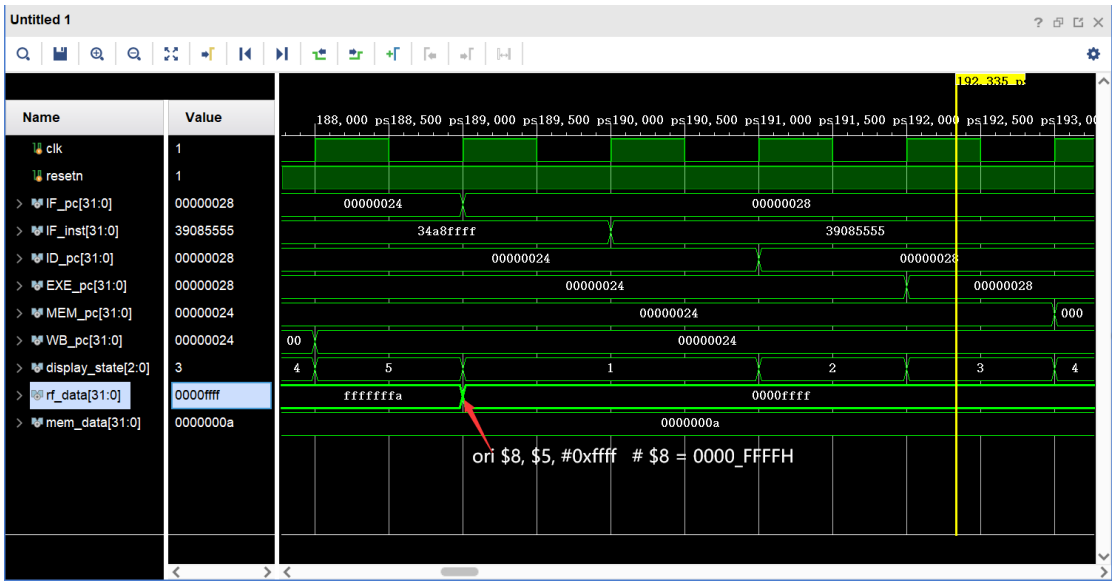

Untitled 1
Name
Value
clk
resetn
IF_pc[31:0]
IF_inst[31:0]
ID_pc[31:0]
EXE_pc[31:0]
MEM_pc[31:0]
WB_pc[31:0]
display_state[2:0]
rf_data[31:0]
mem_data[31:0]
00000018
8c890000
00000014
00000014
00000014
00000014
1
0000000a
0000000a
sw $2, #0($4) #Mem[10H] = 0000_000AH

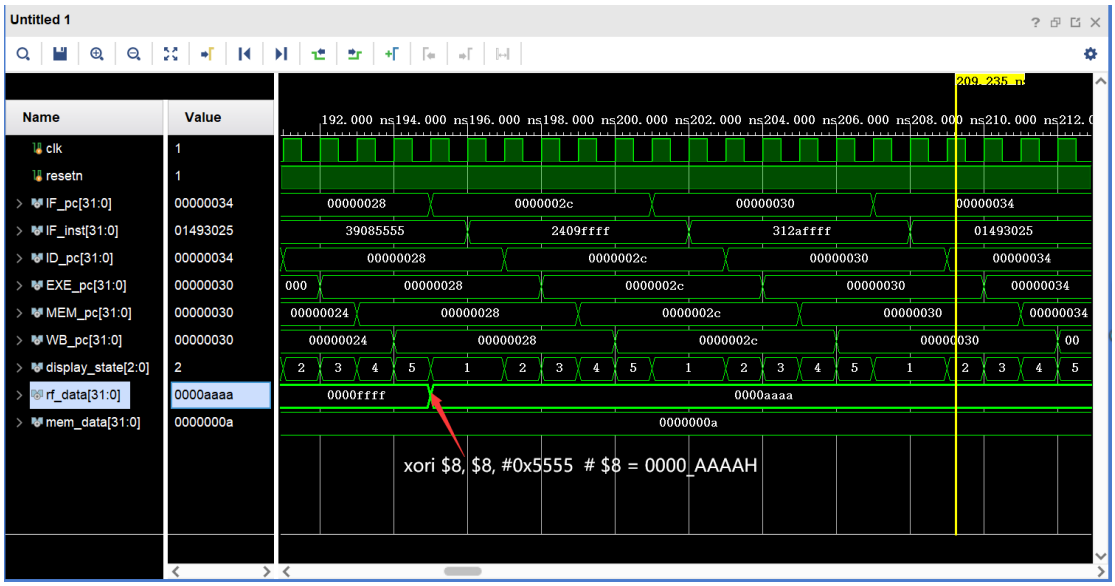
3、寄存器的变化

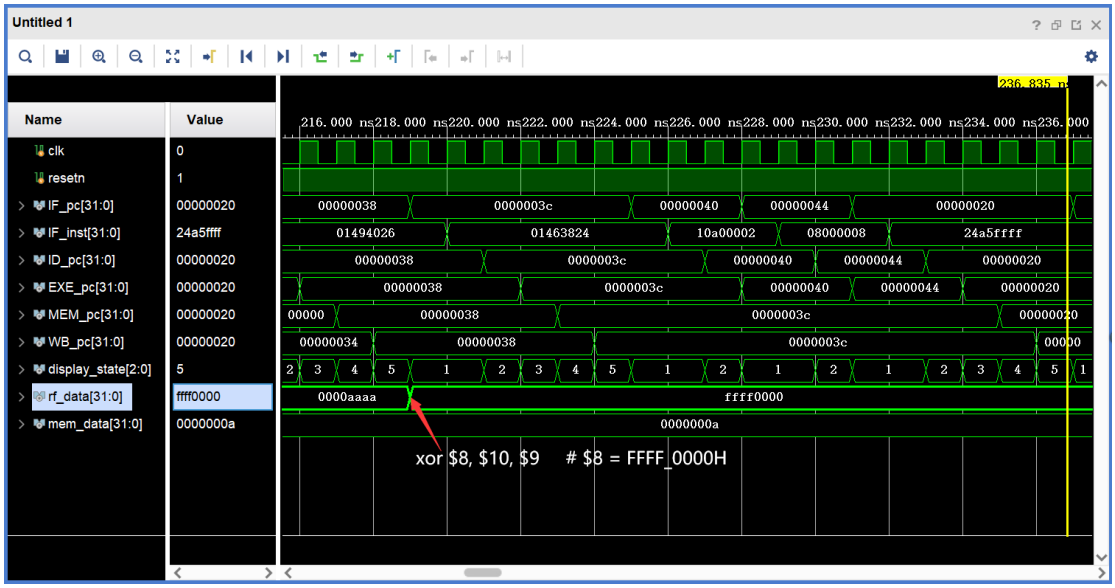

Untitled 1
Name
Value
clk
0
resetn
1
IF_pc[31:0]
00000064
IF_inst[31:0]
8c890000
ID_pc[31:0]
00000064
EXE_pc[31:0]
00000064
MEM_pc[31:0]
00000064
WB_pc[31:0]
00000060
display_state[2:0]
4
rf_data[31:0]
00000000
mem_data[31:0]
00000005
addu $8, $0, $0 # $8 = 0000_0000H

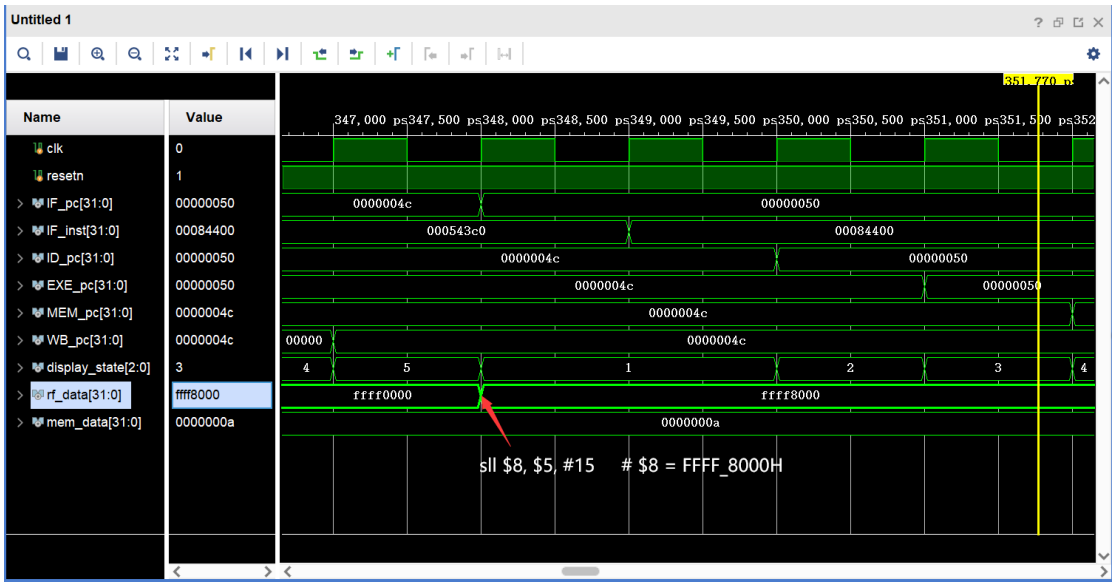

Untitled 1
Name
Value
clk
0
resetn
1
IF_pc[31:0]
00000070
IF_inst[31:0]
24a5ffff
ID_pc[31:0]
00000070
EXE_pc[31:0]
00000070
MEM_pc[31:0]
00000070
WB_pc[31:0]
00000070
display_state[2:0]
5
rf_data[31:0]
00000001
mem_data[31:0]
00000005
addu $8, $8, $9

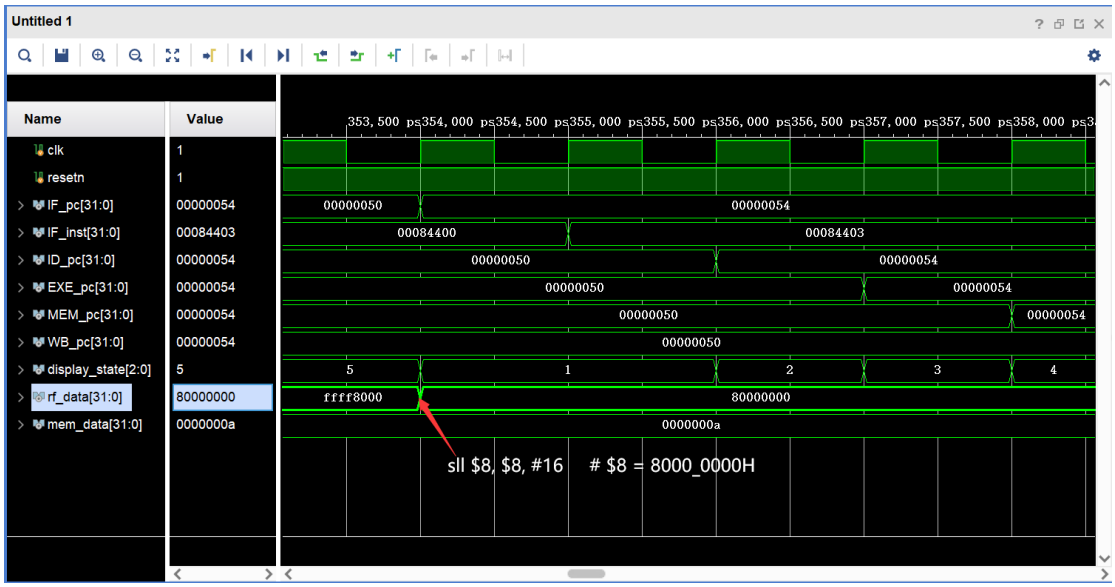

Untitled 1
Name
Value
clk
1
resetn
1
IF_pc[31:0]
00000020
IF_inst[31:0]
24a5ffff
ID_pc[31:0]
0000001c
EXE_pc[31:0]
0000001c
MEM_pc[31:0]
0000001c
WB_pc[31:0]
0000001c
display_state[2:0]
1
rf_data[31:0]
fffffffa
mem_data[31:0]
0000000a
subu $8, $9, $4 # $8 = FFFF_FFFAH (-6D)

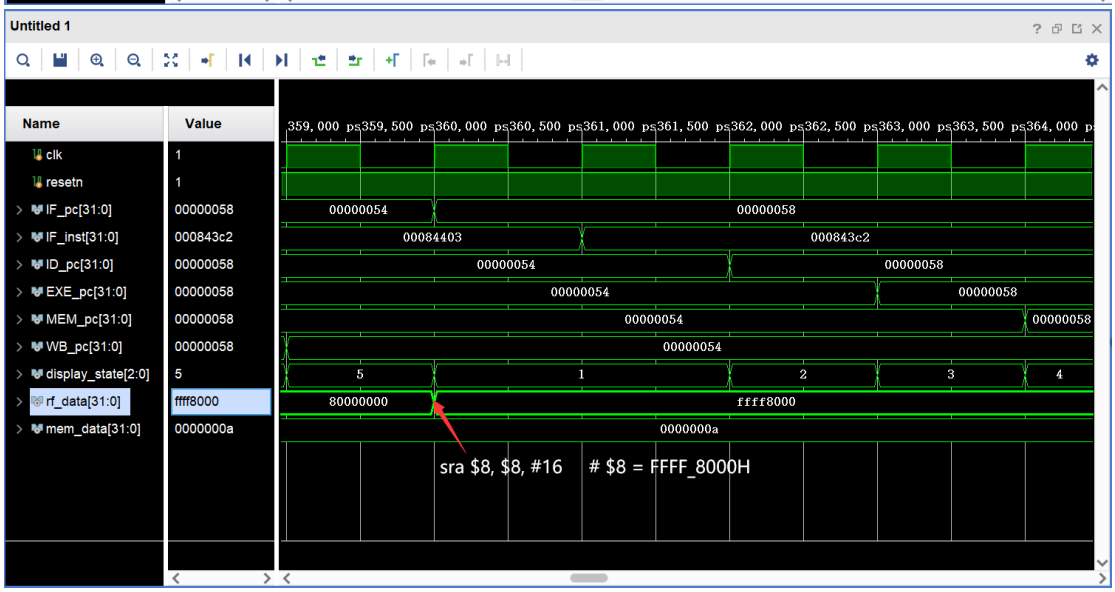

Untitled 1
Name
Value
clk
resetn
IF_pc[31:0]
IF_inst[31:0]
ID_pc[31:0]
EXE_pc[31:0]
MEM_pc[31:0]
WB_pc[31:0]
display_state[2:0]
rf_data[31:0]
mem_data[31:0]
00000028
39085555
00000028
00000028
00000024
00000024
00000024
3
0000ffff
0000000a
ori $8, $5, #0xffff # $8 = 0000_FFFFH

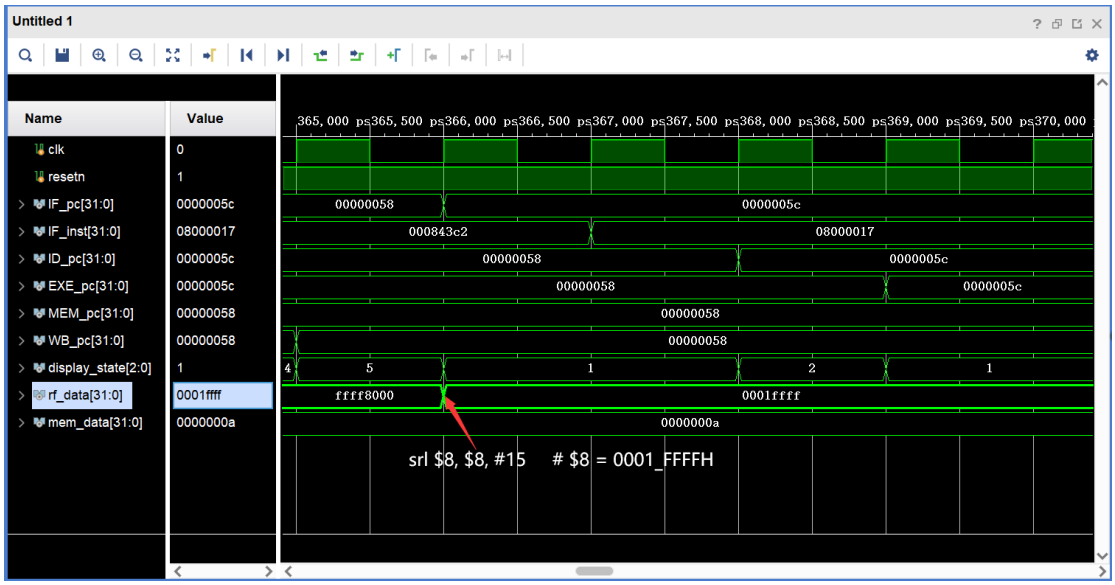

Untitled 1
Name
Value
clk
resetn
IF_pc[31:0]
IF_inst[31:0]
ID_pc[31:0]
EXE_pc[31:0]
MEM_pc[31:0]
WB_pc[31:0]
display_state[2:0]
rf_data[31:0]
mem_data[31:0]
00000034
01493025
00000034
00000030
00000030
00000030
2
0000aaaa
0000000a
xori $8, $8, #0x5555 # $8 = 0000_AAAAH


Untitled 1
Name
Value
clk
resetn
IF_pc[31:0]
IF_inst[31:0]
ID_pc[31:0]
EXE_pc[31:0]
MEM_pc[31:0]
WB_pc[31:0]
display_state[2:0]
rf_data[31:0]
mem_data[31:0]
00000020
24a5ffff
00000020
00000020
00000020
00000020
5
ffff0000
0000000a
xor $8, $10, $9 # $8 = FFFF_0000H


Untitled 1
Name
Value
clk
0
resetn
1
IF_pc[31:0]
00000050
IF_inst[31:0]
00084400
ID_pc[31:0]
00000050
EXE_pc[31:0]
00000050
MEM_pc[31:0]
0000004c
WB_pc[31:0]
0000004c
display_state[2:0]
3
rf_data[31:0]
ffff8000
mem_data[31:0]
0000000a
sll $8, $5, #15 # $8 = FFFF_8000H


Untitled 1
Name
Value
clk
1
resetn
1
IF_pc[31:0]
00000054
IF_inst[31:0]
00084403
ID_pc[31:0]
00000054
EXE_pc[31:0]
00000054
MEM_pc[31:0]
00000054
WB_pc[31:0]
00000054
display_state[2:0]
5
rf_data[31:0]
80000000
mem_data[31:0]
0000000a
sll $8, $8, #16 # $8 = 8000_0000H


Untitled 1
Name
Value
clk
1
resetn
1
IF_pc[31:0]
00000058
IF_inst[31:0]
000843c2
ID_pc[31:0]
00000058
EXE_pc[31:0]
00000058
MEM_pc[31:0]
00000058
WB_pc[31:0]
00000058
display_state[2:0]
5
rf_data[31:0]
ffff8000
mem_data[31:0]
0000000a
sra $8, $8, #16 # $8 = FFFF_8000H



八、心得体会

这次的硬件课程设计，应该是最为简单的一次课设了。

以往的课程设计，要么像 C++课设、编译原理课设、机器学习大作业那样，根据课上学到的知识自己敲代码，算法什么的得将书本上的算法转换成实际的代码、工程文件等，要么像软件课设（Ⅰ）那样，基本得在一些视频网站上现看现学关于某某管理系统架构、前后端分离技术等，看一遍视频作者是怎么搭建一个管理系统，然后自己根据课程设计实际需要进行修改与完善。这次的硬件课设（Ⅰ），只需要照着参考书（我用的是《龙芯教学实验平台 LS-CPU-EXB-002 CPU 设计与体系结构实验指导手册》）打一遍代码，理解如何将指令执行的五个周期编写为五个模块，再将其通过总线串联起来，从而实现一个多周期的 CPU。虽然简单，但是代码量非常大，不过照着参考书打了一遍并按照课设实际要求改了改之后，感觉对指令执行的五个周期更熟悉了，也算是巩固了一遍计算机组成原理的知识，可惜我保研了，不需要考这个。

说实话，这次的课设，以及我们专业的毕业实习，让我对硬件行业有了一些了解，是一次很不错的经历。在编码过程中才想起来，我们去年的这个时候在学组原时，做的实验就是实现寄存器堆、数据存储器这样的，当时觉得那些代码看起来就跟天书一样，同学们你 copy 我的，我 copy 他的，就都完成实验了，现在看来，其实自己学一学 Verilog，再结合组原课上学到的知识，是完全可以自己独立完成那四个实验的。这时其实也不由得感叹学院排课的不合理，因为我个人觉得，这门课设应当安排在大三下学期，也就是刚上完组原的那个学期，这样的话既巩固了我们上学期组原课学到的知识，对考研的同学来说大有裨益，同时也将书本知识转化为代码、工程文件之类，不至于到今年这时候才开始做，对于我一保研的同学来说，组原的知识差不多都忘光了。说实话，如果以后我还干这一行，我在研究生阶段大概会重新精读一遍组原书；如果以后我去卷 AI，搞 CV、NLP 这些，那我可能跟组成原理这方面的工程基本无缘了

仅以此次课设，纪念我曾经学过的一门 4.5 学分的硬课——计算机组成原理。