计算机组成原理 · 实验3报告

计01　容逸朗　2020010869

实验过程

控制器

接口设计

```
module controller (
    input wire clk,
    input wire reset,

    // 连接寄存器堆模块的信号
    output reg  [4:0]  rf_raddr_a,
    input  wire [15:0] rf_rdata_a,
    output reg  [4:0]  rf_raddr_b,
    input  wire [15:0] rf_rdata_b,
    output reg  [4:0]  rf_waddr,
    output reg  [15:0] rf_wdata,
    output reg  rf_wen,

    // 连接 ALU 模块的信号
    output reg  [15:0] alu_a,
    output reg  [15:0] alu_b,
    output reg  [ 3:0] alu_op,
    input  wire [15:0] alu_y,

    // 控制信号
    input  wire        step,     // 用户按键状态脉冲
    input  wire [31:0] dip_sw,   // 32 位拨码开关状态
    output reg  [15:0] leds
);
```

思路

本次实验中，我采用三段式状态机实现控制器，第一段的状态机用于检查重置信号和状态跳转：

```
always_ff @(posedge clk) begin
  if (reset) begin
    cur_state <= ST_INIT;
  end else begin
    cur_state <= next_state;
  end
end
```

第二段是组合逻辑，用于设置下一时钟跳变时状态机的状态：

```
always_comb begin
    next_state = ST_INIT;
    case (cur_state)
      ST_INIT: begin
        if (step) begin
          next_state = ST_DECODE;
        end else begin
          next_state = ST_INIT;
        end
      end

      ST_DECODE: begin
        if (is_rtype) begin
          next_state = ST_CALC;
        end else if (is_peek) begin
          next_state = ST_READ_REG;
        end else if (is_poke) begin
          next_state = ST_WRITE_REG;
        end else begin
          next_state = ST_INIT;
        end
      end

      // 其他状态同理，此处省略之。

    endcase
  end
```

最后一段状态机是时序逻辑，主要负责寄存器与 ALU 的数据交换：

```
always_ff @(posedge clk) begin
    if (reset) begin
      inst_reg <= 32'b0;
      leds <= 32'b0;
      rf_wen <= 1'b0;
    end else begin
      case (cur_state)

        ST_DECODE: begin
          if (is_rtype) begin
```

```
            rf_raddr_a <= rs1;
            rf_raddr_b <= rs2;
            rf_wen <= 1'b0;
          end else if (is_peek) begin
            rf_raddr_a <= rd;
            rf_wen <= 1'b0;
          end else if (is_poke) begin
            rf_waddr <= rd;
            rf_wdata <= imm;
            rf_wen <= 1'b1;
          end
        end

        ST_CALC: begin
          // TODO: 将数据交给 ALU，并从 ALU 获取结果
          alu_a <= rf_rdata_a;
          alu_b <= rf_rdata_b;
          alu_op <= opcode;
        end

        ST_WRITE_REG: begin
          // TODO: 将结果存入寄存器
          rf_wen <= 1'b1;
          rf_waddr <= rd;
          if (is_rtype) begin
            rf_wdata <= alu_y;
          end else begin
            rf_wdata <= imm;
          end
        end

        // 其他状态同理，此处省略之。

      endcase
    end
  end
```

ALU

接口设计

```
module alu(
    input   reg  [15:0] a,
    input   reg  [15:0] b,
    input   reg  [ 3:0] op,
    output  wire [15:0] y
);
```

思路

由于 ALU 不涉及寄存器堆的操作，因此计算完成后可以立刻反回值，故利用组合逻辑完成此模块的设计：

```
reg [15:0] result;

always_comb begin
  case (op)
    4'b0001: result = a + b;
    4'b0010: result = a - b;
    4'b0011: result = a & b;
    4'b0100: result = a | b;
    4'b0101: result = a ^ b;
    4'b0110: result = ~a;
    4'b0111: result = a << (b & 15);
    4'b1000: result = a >> (b & 15);
    4'b1001: result = $signed(a) >>> (b & 15);
    4'b1010: result = (a << (b & 15)) + (a >> (16 - (b & 15)));
    default: result = 16'b0;
  endcase
end

assign y = result;
```

寄存器堆

接口设计

```
module register_file(
    input wire clk,
    input wire reset,

    // 连接寄存器堆模块的信号
    input  reg  [ 4:0] raddr_a,
    output wire [15:0] rdata_a,
    input  reg  [ 4:0] raddr_b,
    output wire [15:0] rdata_b,
    input  reg  [ 4:0] waddr,
    input  reg  [15:0] wdata,
    input  reg         wen
);
```

思路

为了避免寄存器数据在计算时被复盖，因此写入寄存器时需要等待时钟正跳变时才能写入。由于数据不会突然跳变，所以我们可以直接读取寄存器而不需等待时钟跳变，即：

```
logic [15:0] register [31:0];

```

```
  always_ff @(posedge clk) begin
    if (reset) begin
      register <= '{default:0};
    end else begin
      if (wen && waddr =/= 0) begin
        register[waddr] <= wdata;
      end
    end
  end

  assign rdata_a = register[raddr_a];
  assign rdata_b = register[raddr_b];
```

仿真

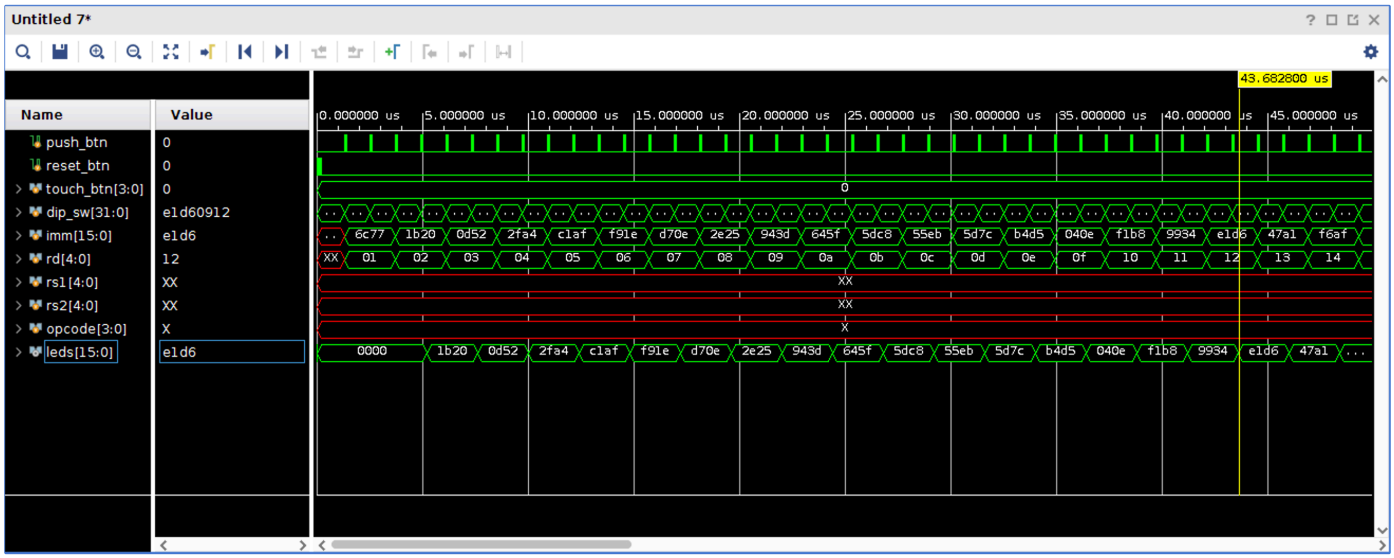
Type-I

首先把 32 个寄存器随意放入值并检查之：

```
for (int i = 1; i < 32; i = i + 1) begin
  #100;
  rd = i;   // only lower 5 bits
  imm = $urandom_range(0, 65536);
  dip_sw = `inst_poke(rd, imm);
  push_btn = 1;
  #100;
  push_btn = 0;
  #1000;

  #100;
  dip_sw = `inst_peek(rd, imm);
  push_btn = 1;
  #100;
  push_btn = 0;
  #1000;
end
```

从仿真的结果可知代码无误：



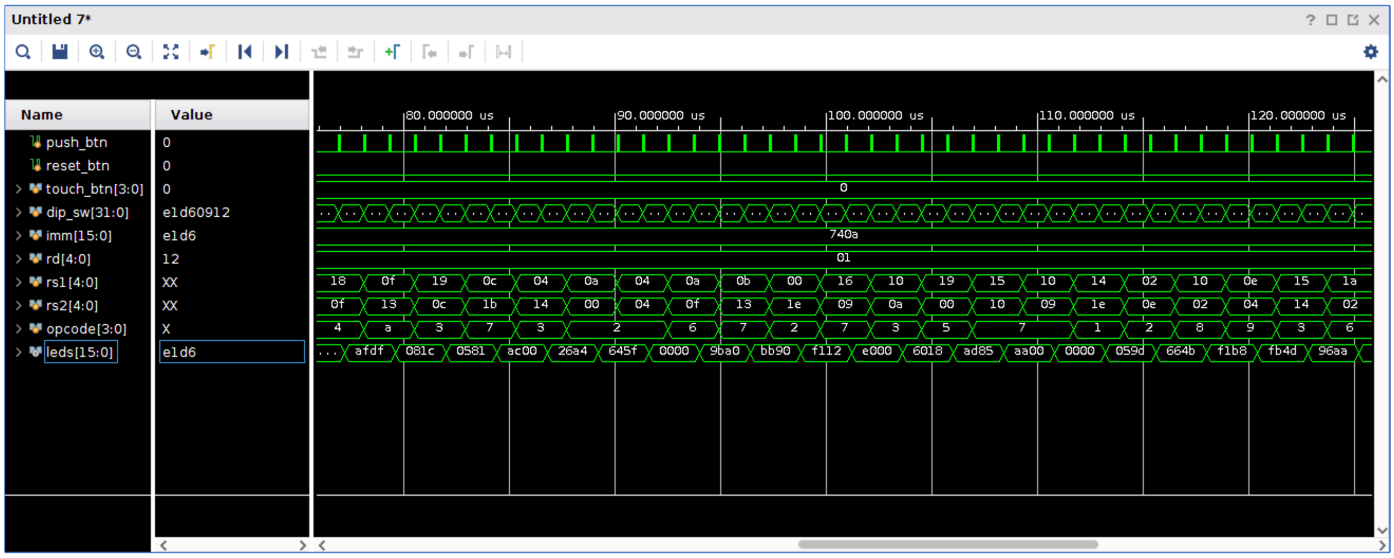
Type-R

随意从 32 个寄存器抽出两个（可重复），再从 10 个操作中抽作一个操作计算，再利用 peek 打印结果：

```
for (int i = 1; i < 32; i = i + 1) begin
  #100;
  rd = 1;   // only lower 5 bits
  rs1 = $urandom_range(0, 31);
  rs2 = $urandom_range(0, 31);
  opcode = $urandom_range(1, 10);
  dip_sw = `inst_rtype(rd, rs1, rs2, opcode);
  push_btn = 1;
  #100;
  push_btn = 0;
  #1000;

  #100;
  dip_sw = `inst_peek(rd, 0);
  push_btn = 1;
  #100;
  push_btn = 0;
  #1000;
end
```

从仿真的结果可知代码无误：



实验总结

本次实验中，我共提交了三次评测才通过本次实验，错误原因如下所述：

- 忽略了 `0` 号寄存器的值是固定为零的规则
- 移位指令只取寄存器 `B` 的低 `4` 位来计算

综上所述，我犯的错误主要是因为对 `RISC-V` 文档认识不全面所导致的，因此在后续实验中需要特别注意这一问题。