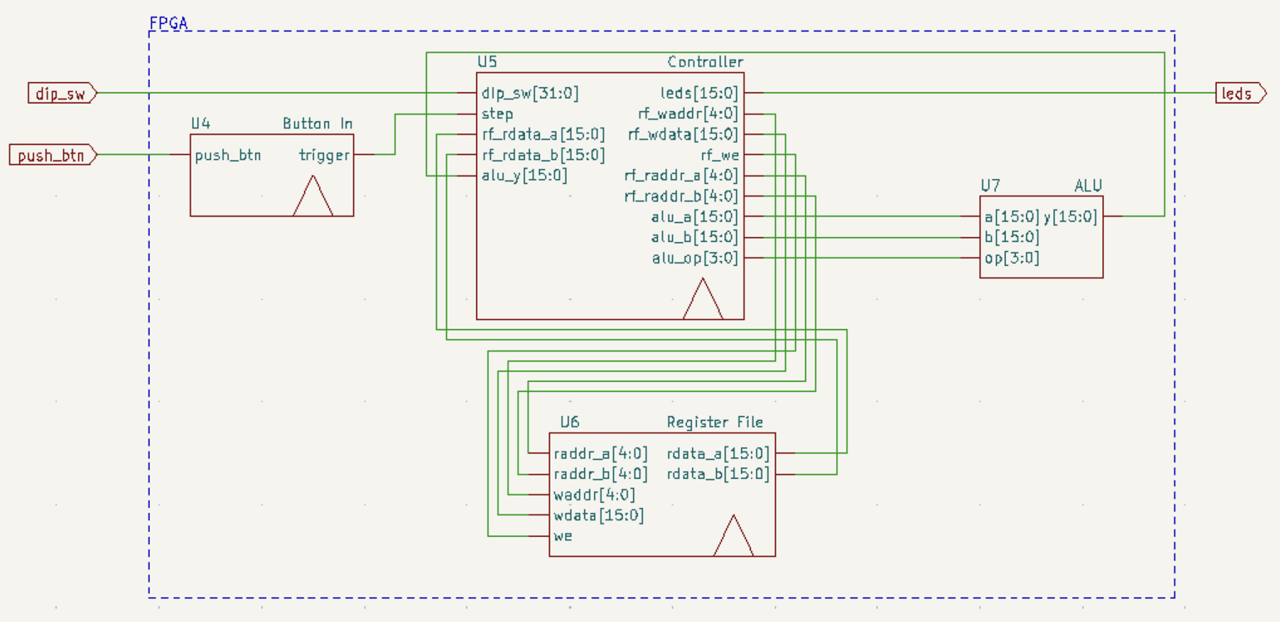
# 实验三：ALU 与寄存器堆实验 实验报告

(邢竞择 2020012890)

## 1 模块设计

实验共需设计三个模块：控制器、ALU、寄存器堆，其中 ALU 完全为组合逻辑，寄存器堆的数据读取是组合逻辑、数据写入是时序逻辑，控制器总体为三段式的 FSM，其中状态转移为时序逻辑、计算下一状态、解码等为组合逻辑、其他操作（LED 驱动、指令读取）为时序逻辑。

附实验指导书上的原理图：




### 1.1 Controller

FSM 第一段，在上升沿转移到下一个状态

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

FSM 第二段，推导下一状态

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

```
        if (is_rtype) // R-Type operation
          next_state = ST_CALC;
        else if (is_peek)
          next_state = ST_READ_REG;
        else if (is_poke)
          next_state = ST_WRITE_REG;
        else next_state = ST_INIT;  // illegal operation
      end
      ST_CALC: next_state = ST_WRITE_REG;
      ST_READ_REG: next_state = ST_INIT;
      ST_WRITE_REG: next_state = ST_INIT;
      default: next_state = ST_INIT;
    endcase
  end
```

FSM 第三段，完成其他操作

```
  always_ff @(posedge clk) begin
    case(cur_state)
      ST_INIT: begin
        inst_reg <= dip_sw;
      end
      ST_DECODE: begin
      end
      ST_CALC: begin
      end
      ST_READ_REG: begin
        leds <= rf_rdata_a;
      end
      ST_WRITE_REG: begin
      end
      default: begin
      end
    endcase
  end
```

为了尽量提高效率，将解码用组合逻辑完成， `rf_raddr_a` 等信号随着 `inst_reg` 的变化立即变化，不需要慢一拍，而读寄存器也是组合逻辑，从而避免了等待。 `rf_we` 仅在写操作前一周期被设置为 1，从而保证在且仅在写操作阶段的时钟上升沿寄存器堆读到的 `we` 是 1.

```
  always_comb begin
    is_rtype = (inst_reg[2:0] == 3'b001);
    is_itype = (inst_reg[2:0] == 3'b010);
    is_peek = is_itype && (inst_reg[6:3] == 4'b0010);
    is_poke = is_itype && (inst_reg[6:3] == 4'b0001);

    imm = inst_reg[31:16];
    rd = inst_reg[11:7];
    rs1 = inst_reg[19:15];
```

```
    rs2 = inst_reg[24:20];
    opcode = inst_reg[6:3];

    rf_raddr_a = is_peek ? rd : rs1;
    rf_raddr_b = rs2;
    alu_op = opcode;
    alu_a = rf_rdata_a;
    alu_b = rf_rdata_b;
    rf_waddr = rd;
    rf_wdata = is_rtype ? alu_y : imm; // will be peek if not calc
    rf_we = next_state == ST_WRITE_REG ? 1'b1 : 1'b0;
  end
```

## 1.2 ALU

实现算术右移时，需注意由于默认当作无符号数，所以必须加入 `$signed()` 。

```
always_comb begin
  case(op)
    4'd1: data_y = data_a + data_b;
    4'd2: data_y = data_a - data_b;
    4'd3: data_y = data_a & data_b;
    4'd4: data_y = data_a | data_b;
    4'd5: data_y = data_a ^ data_b;
    4'd6: data_y = ~data_a;
    4'd7: data_y = data_a << (data_b & 16'b1111);
    4'd8: data_y = data_a >> (data_b & 16'b1111);
    4'd9: data_y = $signed(data_a) >>> (data_b & 16'b1111);
    4'd10: data_y = (data_a >> (16 - (data_b & 16'b1111))) + (data_a << (data_b &
16'b1111));
    default: data_y = 16'b0;
  endcase
end
```

## 1.3 RegFiles

```
  reg [15:0] files [31:0];
  always_comb begin
    if (raddr_a == 5'b0)
      rdata_a = 16'b0;
    else
      rdata_a = files[raddr_a];
    if (raddr_b == 5'b0)
      rdata_b = 16'b0;
    else
      rdata_b = files[raddr_b];
  end
  always_ff @(posedge clk) begin
    if (we == 1'b1)
```

```
14          files[waddr] <= wdata;
15      end
```

## 2 讨论

如果寄存器堆的读取、控制器的解码用时序逻辑实现，应该如何实现？我在该实验的讲解之前就提前完成了一个版本 e4bd2a8f ，它使用了 7 个状态来完成这个任务，它与我现在的实现的差别在于

- rf_raddr_a 在 ST_DECODE 阶段才被更新，在下一时钟周期寄存器堆拿到地址，再过一个时钟周期读取的数据才被返回
- 在 ST_READ_REG 前增加一个 PENDING 状态，等待寄存器堆返回数据
- 在 ST_CALC 前增加一个 PENDING 状态，等待寄存器堆返回数据
- 在 ST_INIT 要将 rf_we 置 0

这种实现比目前的实现执行效率低很多，而且现在的实现还可以进一步通过状态压缩来缩短周期。