浙江大学

本科实验报告

课程名称:	计算机组成
实验名称:	实验一：ALU、Regfile 与有限状态机设计
姓　　名:	李宇晗
学　　号:	3240106155
专　　业:	计算机科学与技术
实验地点:	东教 509
指导教师:	林芃
报告日期:	2025/09/23

ALU 设计

一、 实验目的及任务

本次实验旨在设计并实现一个 32 位的算术逻辑单元 (ALU)，它是 CPU 数据通路中的核心计算部件。通过本实验，需要掌握数据通路中核心功能部件的设计方法，并熟练运用两种不同的硬件描述方法——结构化描述与功能性描述——来构建数字逻辑系统。

任务是设计一个 32 位 ALU，能够根据 3 位控制信号 `ALU_operation` 执行 8 种不同的操作。本实验将探索并实现两种设计方法：

1. 结构化描述：采用原理图或模块实例化的方法，将预先设计好的基本运算单元（如加法器、与门等）连接起来，构成完整的 ALU。
2. 功能性描述：采用 Verilog 的行为级描述，在 `always` 块中使用 `case` 语句，根据控制信号直接描述 ALU 应执行的数学或逻辑运算功能。

二、 实验原理

ALU 的核心是一个 8-1 多路选择器 (MUX)，它根据控制信号 `ALU_operation` 从多个并行运算单元的输出中选择一个作为最终结果 `res`。

1. 减法实现：减法 `A - B` 通过二进制补码加法 `A + (~B) + 1` 来实现。设计中，一个异或门用于根据减法控制信号对操作数 B 进行按位取反，同时将加法器的最低位进位输入 `Cin` 置为 1。
2. 小于则置位 **(SLT)**：无符号数的小于比较 `A < B` 可通过判断 `A - B` 是否产生借位来实现。在 `A + (~B) + 1` 的运算中，这等价于最高位的进位输出 `add_sub_cout` 为 0。
3. 零标志位 **(Zero Flag)**：`zero` 信号用于指示运算结果 `res` 是否为零。该信号通过一个 32 输入的“或”逻辑将 `res` 的所有位进行运算，然后将结果取反得到。只有当 `res` 所有位都为 0 时，`zero` 信号才为 1。

三、 实验过程及记录

3.1) 结构化描述

1. 创建工程：新建 Vivado 工程，并创建一个名为 “ALU” 的 Block Design。
2. 添加模块：将实验提供的 `and32`、`or32`、`addc_32` 等基础模块的 Verilog 文件添加为工程的设计源文件。
3. 编写顶层代码：编写 ALU.v，连接各个模块
4. 仿真验证：编写 Testbench 文件 (`ALU_tb.v`)，对 ALU 的所有功能进行测试。

实验记录 (结构化描述 **Verilog** 代码)**:**

```
module ALU(
    input  [31:0] A,
    input  [2:0]  ALU_operation,
    input  [31:0]  B,
    output [31:0] res,
```

```
    output        zero
);
// --- 中间信号线 ---
    wire [31:0] and_res;
    wire [31:0] or_res;
    wire [31:0] main_xor_res;
    wire [31:0] nor_res;
    wire [31:0] srl_res;

    // 减法和比较操作所需的信号
    wire        sub_ctrl; // 减法控制信号
    wire [31:0] sub_ext_bus;    // 扩展后的减法控制总线
    wire [31:0] b_operand; // 送入加法器的B操作数 (减法时为~B)
    wire [31:0] add_sub_sum;    // 加法/减法的结果
    wire        add_sub_cout; // 加法/减法的进位输出

    // 比较结果信号
    wire [31:0] slt_res;

    // 控制信号：当操作为 SUB (110) 或 SLT (111) 时，启动减法模式
    assign sub_ctrl = (ALU_operation == 3'b110) || (ALU_operation == 3'b111);

    // --- 子模块实例化 ---
    SignalExt_32_0 SignalExt_32_0 ( .sel(sub_ctrl), .out(sub_ext_bus) );
    xor32_0 xor32_1 ( .A(B), .B(sub_ext_bus), .res(b_operand) );
    addc_32_0 addc_32_0
( .A(A), .B(b_operand), .Cin(sub_ctrl), .S(add_sub_sum), .Cout(add_sub_cout) );
    and32_0 and32_0 (.A(A), .B(B), .res(and_res));
    nor32_0 nor32_0 (.A(A), .B(B), .res(nor_res));
    or32_0 or32_0 (.A(A), .B(B), .res(or_res));
    srl32_0 srl32_0 (.A(A), .B(B), .res(srl_res));
    xor32_0 xor32_0 (.A(A), .B(B), .res(main_xor_res));

    assign slt_res = {31'b0, ~add_sub_cout};

    MUX8T1_32_0 MUX8T1_32_0 (
        .I0(and_res), .I1(or_res), .I2(add_sub_sum), .I3(main_xor_res),
        .I4(nor_res), .I5(srl_res), .I6(add_sub_sum), .I7(slt_res),
        .s(ALU_operation), .o(res)
    );

    wire zero_flag_intermediate;
    or_bit_32_0 or_bit_32_0 ( .in(res), .out(zero_flag_intermediate) );
    assign zero = ~zero_flag_intermediate;

endmodule
```

3.2) 功能性描述

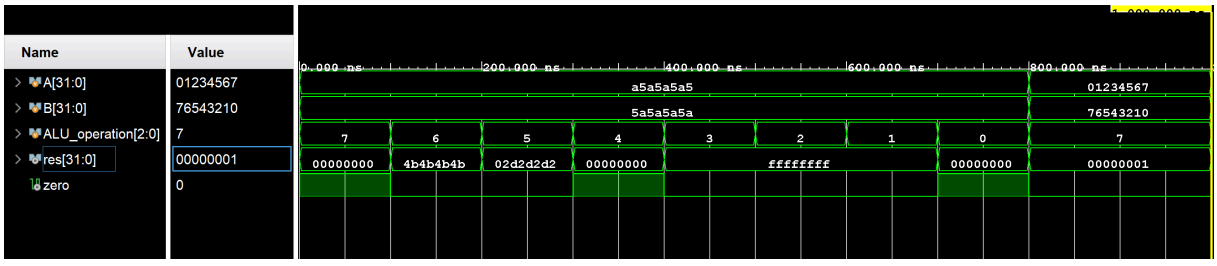
采用 Verilog 行为级描述，在一个 `always` 块中使用 `case` 语句，根据控制信号直接描述 ALU 应执行的数学或逻辑运算功能。

实验记录 **(**功能性描述 **Verilog** 代码**):**

```
`timescale 1ns / 1ps

// 32位ALU
// ALU_op: 000=AND, 001=OR, 010=ADD, 011=XOR
//         100=NOR, 101=SRL, 110=SUB, 111=SLTU
module ALU(
    input  [31:0] A,
    input  [2:0]  ALU_operation,
    input  [31:0] B,
    output reg [31:0] res,
    output            zero
);

    always @(*) begin
        case (ALU_operation)
            3'b000: res = A & B;
            3'b001: res = A | B;
            3'b010: res = A + B;
            3'b011: res = A ^ B;
            3'b100: res = ~(A | B);
            3'b101: res = B >> A[4:0];
            3'b110: res = A - B;
            3'b111: res = (A < B) ? 32'd1 : 32'd0;
            default: res = 32'hxxxxxxxx;
        endcase
    end

    assign zero = (res == 32'd0);

endmodule
```

四、 实验结果分析

4.1) ALU 仿真波形图:



4.2) 结果分析:

从 ALU 的仿真波形图可以看出，测试平台在实验开始后，依次将 `ALU_operation` 控制信号从 7 (`111`) 遍历到 0 (`000`)，从而对 ALU 的全部八种功能进行了测试。

1. 逻辑与算术运算验证：在 `t=400ns` 时，`ALU_operation` 为 `4` (NOR)，输入 A=`a5a5a5a5`，B=`5a5a5a5a`，此时 `A | B` 的结果为 `ffffffff`，取反后得到 `res` 为 `00000000`，同时 `zero` 标志位被正确置为 1。在 `t=200ns` 时，`ALU_operation` 为 `6` (SUB)，`res` 显示为 `4b4b4b4b`，

这正是 0xa5a5a5a5 - 0x5a5a5a5a 的正确结果。其他如加法（op=2）、异或（op=3） 等运算结果也均与预期理论值相符。

2. 移位与比较运算验证：在 t=300ns 时，ALU_operation 为 5 (SRL)，res 显示为 02d2d2d2，这是将操作数 B (5a5a5a5a) 逻辑右移 A 的低 5 位(即 5 位)的结果，验证了移位功能的正确性。在 t=100ns 之前，ALU_operation 为 7 (SLTU)，输入 A=01234567, B=76543210，由于 A < B，res 的值为 00000001，正确实现了无符号数小于则置位的功能。

综上所述，仿真波形验证了所设计的 ALU 能够根据 ALU_operation 控制信号，对输入的 32 位操作数 A 和 B 正确执行 AND, OR, ADD, XOR, NOR, SRL, SUB, SLTU 共八种预设的运算，且 zero 标志位也能准确反映运算结果是否为零。

五、 讨论与心得

本次实验让我对 ALU 的内部结构有了直观的认识。结构化描述方法虽然前期需要准备好各个基础模块，但逻辑清晰，易于调试；而功能性描述则更为简洁，代码量少，可读性高，更侧重于算法行为本身。关于“如何给 ALU 增加溢出功能”，对于带符号数的加法，当两个正数相加得到负数，或两个负数相加得到正数时，即发生溢出。这可以通过检测操作数 A、B 和结果 res 的最高位（符号位）来实现。

Regfile 设计

一、 实验目的及任务

本次实验旨在设计并实现一个 32x32 位的寄存器文件 (Register Files)，掌握其异步读、同步写的工作原理。任务要求采用 Verilog HDL 行为级描述方法进行设计，并学习在 IP 封装过程中处理时钟与复位信号的属性约束问题，最终封装为一个可复用的 IP 核。

二、 实验原理

寄存器文件的核心是一个由 31 个 32 位寄存器构成的存储阵列（`reg [31:0] register [1:31]`）。设计遵循 MIPS 架构的约定，即 0 号寄存器是只读的，且其值恒为 0。

1. 读操作：读操作是异步的组合逻辑。通过 `assign` 语句实现，输出数据会立即根据输入的读地址变化而更新。
2. 写操作：写操作是同步的时序逻辑。只有当写使能信号 `RegWrite` 为 1 且写地址 `Wt_addr` 不为 0 时，在时钟的上升沿，`Wt_data` 的数据才会被写入指定的寄存器中。
3. 复位操作：当复位信号 `rst` 为高电平时，会将所有寄存器（r1-r31）异步清零。

三、 实验过程及记录

1. 代码实现：使用 Verilog HDL 编写 `Regs.v` 模块，描述寄存器文件的行为。
2. 仿真验证：编写 Testbench，模拟一系列读写操作，验证其功能的正确性。
3. **IP** 封装与调试：将仿真通过的模块封装为 IP 核。为解决 Vivado 产生的端口警告和复位信号逻辑反向问题，需在 IP Packager 中为 `clk` 端口添加 `ASSOCIATED_BUSIF` 属性，并为 `rst` 端口添加 `POLARITY` 属性，将其值设置为 `ACTIVE_HIGH`。

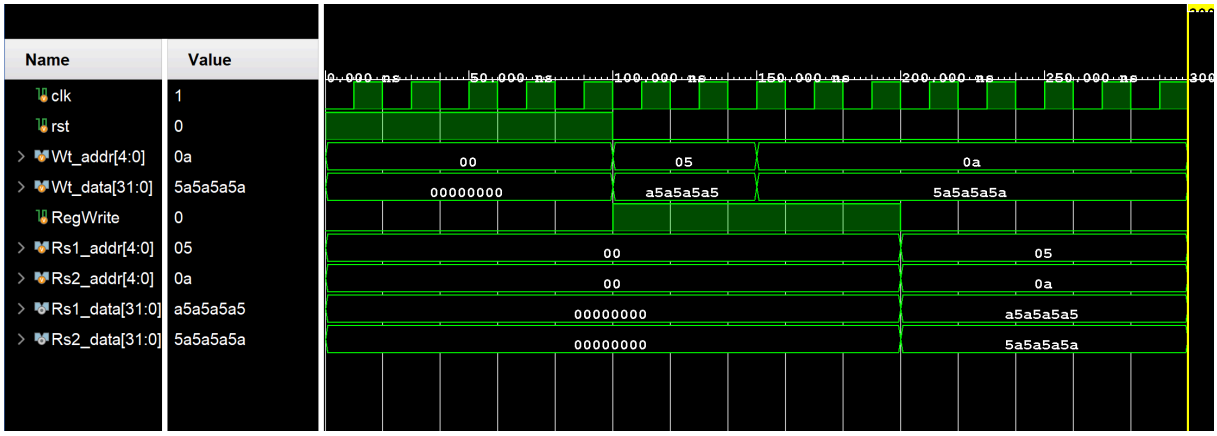
实验记录 **(Regs.v):**

```
module Regs(
    input clk,
    input rst,
    input RegWrite,
    input [4:0] Rs1_addr,
    input [4:0] Rs2_addr,
    input [4:0] Wt_addr,
    input [31:0] Wt_data,
    output [31:0] Rs1_data,
    output [31:0] Rs2_data
);
    reg [31:0] register;
    integer i;

    assign Rs1_data = (Rs1_addr == 5'b0) ? 32'b0 : register[Rs1_addr];
    assign Rs2_data = (Rs2_addr == 5'b0) ? 32'b0 : register[Rs2_addr];

    always @(posedge clk or posedge rst)
    begin
        if (rst == 1'b1) begin
```

```
            for (i = 1; i < 32; i = i + 1) begin
                register[i] <= 32'b0;
            end
        end
        else if ((Wt_addr != 5'b0) && (RegWrite == 1'b1)) begin
            register[Wt_addr] <= Wt_data;
        end
    end
endmodule
```

四、 实验结果分析

4.1) Regfile 仿真波形图:



4.2) 结果分析:

该仿真波形清晰地展示了寄存器文件异步读、同步写的工作特性。

1. 同步写操作验证：在 `t=100ns` 附近第一个时钟上升沿，`RegWrite` 为高电平，`Wt_addr` 为 `0a`，`Wt_data` 为 `5a5a5a5a`，数据被成功写入 10 号寄存器。同样，在 `t=150ns` 附近的时钟上升沿，数据 `a5a5a5a5` 被写入 `05` 号寄存器。数据写入操作严格发生在时钟的上升沿，且写使能 `RegWrite` 为高电平、写地址不为 0 时才执行，验证了同步写的时序逻辑。
2. 异步读操作验证：在第一次写操作完成后，读地址 `Rs2_addr` 立即变为 `0a`，输出端口 `Rs2_data` 几乎在同一时刻就更新为刚写入的数据 `5a5a5a5a`,没有等待下一个时钟周期。同样，当 `Rs1_addr` 更新为 `05` 后，`Rs1_data` 也立即更新为 `a5a5a5a5`。这表明读操作是组合逻辑，实现了异步读取。
3. 零号寄存器特性验证：在仿真开始阶段，读地址 `Rs1_addr` 和 `Rs2_addr` 均为 `00`，输出数据 `Rs1_data` 和 `Rs2_data` 始终为 `00000000`，这与代码中对 0 地址的特殊判断逻辑相符，验证了 0 号寄存器恒为 0 的设计要求。

仿真结果表明，所设计的寄存器文件功能正确。

五、 讨论与心得

本次实验让我掌握了使用 Verilog 设计时序与组合逻辑混合电路的方法。特别是在 IP 封装阶段遇到的 `clk` 和 `rst` 信号属性约束问题，让我认识到在模块化设计中，提供清晰的

接口属性定义对于后续的系统集成至关重要。通过对比异步读和同步写，我更深刻地理解了时序电路中数据读写的控制方法。

三段式有限状态机 (FSM) 设计

一、 实验目的及任务

本次实验旨在设计一个序列检测器，通过实践加深对有限状态机 (FSM) 基本原理的理解。任务要求设计一个能够检测特定二进制序列 “1110010” 的 Moore 型状态机，并重点学习和应用 “三段式” Verilog 描述风格来设计该时序逻辑电路。

二、 实验原理

Moore 型状态机的输出只与当前所处的状态有关，与当前的输入无关。本实验采用的“三段式”设计风格是一种结构清晰、易于维护和时序优化的 FSM 编码风格，它将 FSM 的逻辑划分为三个独立的部分：

1. 第一段 (状态转移)：一个时序逻辑 `always` 块，负责在时钟 `clk` 驱动下更新状态寄存器 (`curr_state <= next_state`)，并处理复位逻辑。
2. 第二段 (次态逻辑)：一个组合逻辑 `always` 块，根据当前状态 `curr_state` 和输入 `in` 来判断下一个状态 `next_state`。
3. 第三段 (输出逻辑)：一个组合逻辑部分 (`assign` 语句)，根据当前状态 `curr_state` 产生输出 `out`。

三、 实验过程及记录

1. 状态分析与定义：根据目标序列 “1110010” 设计状态转换图，定义 S0 到 S7 共 8 个状态，并使用 3 位二进制码进行编码。
2. 代码实现：根据状态转换图，使用 Verilog 的三段式风格编写 FSM 模块。
3. 仿真验证：编写 Testbench，提供包含目标序列的输入激励，观察输出 `out` 是否在正确的时间点产生高电平脉冲。

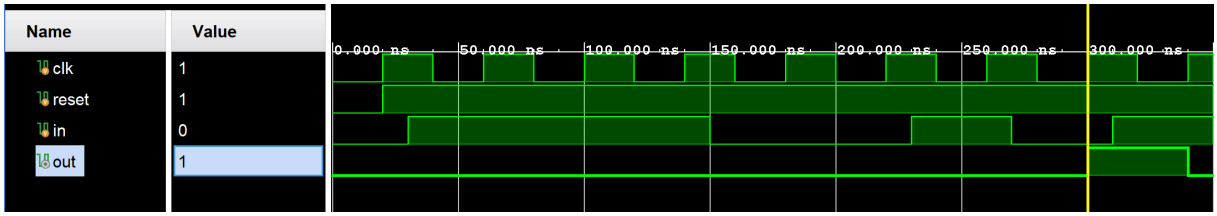
实验记录 **(seq.v):**

```
module seq(
    clk,
    reset,
    in,
    out
);
    input clk;
    input reset;
    input in;
    output out;

    parameter [2:0] S0 = 3'b000, S1 = 3'b001, S2 = 3'b010,
                    S3 = 3'b011, S4 = 3'b100, S5 = 3'b101,
                    S6 = 3'b110, S7 = 3'b111;

    reg [2:0] curr_state;
    reg [2:0] next_state;
```

```
    // 第一段：状态转移
    always @(posedge clk or negedge reset)
    begin
        if(!reset)
            curr_state <= S0;
        else
            curr_state <= next_state;
    end

    // 第二段：次态逻辑
    always @(curr_state or in)
    begin
        case(curr_state)
            S0: if(in == 0) next_state = S0; else next_state = S1;
            S1: if(in == 0) next_state = S0; else next_state = S2;
            S2: if(in == 0) next_state = S0; else next_state = S3;
            S3: if(in == 0) next_state = S4; else next_state = S2;
            S4: if(in == 0) next_state = S5; else next_state = S1;
            S5: if(in == 0) next_state = S0; else next_state = S6;
            S6: if(in == 0) next_state = S7; else next_state = S2;
            S7: if(in == 0) next_state = S0; else next_state = S1;
            default: next_state = S0;
        endcase
    end

    // 第三段：输出逻辑
    assign out = (curr_state == S7) ? 1:0;
endmodule
```

四、 实验结果分析

4.1) FSM 仿真波形图：



4.2) 结果分析：

仿真波形验证了所设计的序列检测器能够成功检测到目标序列 “1110010”。

1. 序列检测过程：从波形图可以看出，输入信号 `in` 在时钟 `clk` 的驱动下，依次输入了 `1` (持续 3 个周期)、`0` (持续 2 个周期)、`1` (持续 1 个周期)、`0` (持续 1 个周期) 的序列，这恰好构成了目标序列 “1110010”。
2. 状态转移与输出：根据 FSM 的状态转移设计，当最后一个 `0` 在 `t=275ns` 的时钟上升沿被采样后，状态机从 S6 状态转移到了最终的 S7 状态。

3. **Moore** 型输出验证：由于采用 Moore 型状态机设计，输出 `out` 仅与当前状态有关。在 `t=275ns` 到 `t=300ns` 这一个时钟周期内，当前状态 `curr_state` 为 S7。因此，输出信号 `out` 在此期间变为高电平 `1`，并在下一个时钟沿（`t=300ns`）随着状态的改变而恢复为低电平 `0`。

五、 讨论与心得

本次实验让我深刻体会到了三段式 FSM 设计的优越性。它将时序逻辑和组合逻辑完全分离，使得代码结构非常清晰，易于理解和调试。与一段式相比，三段式能更好地被综合工具优化，有助于获得更好的时序性能，并且可以有效避免因组合逻辑产生的毛刺被误采样为状态输出。通过这个实验，我不仅掌握了 FSM 的设计方法，也对如何将一个具体问题抽象为状态转换模型有了更深入的理解。