中国科学院大学网络空间安全学院

计算机组成与结构研讨课

实验报告

实验序号：3　　实验名称：基本功能部件设计

1　实验 verilog 代码

1.1　实验代码

1.1.1　寄存器堆 Register 设计

实验目的

> 设计 MIPS 功能型处理器中使用的 32-bit 通用寄存器堆 (Register File)，支持 2 读 1 写端口。

代码实现

```
`timescale 10 ns / 1 ns

`define DATA_WIDTH 32
`define ADDR_WIDTH 5

module top_module(
    input clk,
    input rst,
    input [`ADDR_WIDTH - 1:0] waddr,
    input [`ADDR_WIDTH - 1:0] raddr1,
    input [`ADDR_WIDTH - 1:0] raddr2,
    input wen,
    input [`DATA_WIDTH - 1:0] wdata,
```

```
    output [`DATA_WIDTH - 1:0] rdata1,
    output [`DATA_WIDTH - 1:0] rdata2
);
    //声明通用寄存器
    reg [`DATA_WIDTH-1:0] reg_file[0:(1<<`ADDR_WIDTH)-1];

    //读操作
    assign rdata1=(raddr1==0)?32'b0:reg_file[raddr1];
    assign rdata2=(raddr2==0)?32'b0:reg_file[raddr2];
    //写操作
    always @(posedge clk) begin
        if (wen==1&&(waddr!=0)) begin
            reg_file[waddr]<=wdata;
        end
    end


endmodule
```

1.1.2 运算器 ALU 设计

实验目的

设计 MIPS 功能型处理器中使用的算术逻辑单元 ALU。要求支持逻辑按位与（And）、逻辑按位或（Or）、算术加法（Add）、算术减法（Subtract）、有符号整数比较（Set on less than,slt）等五种基本 ALU 运算功能。

代码实现

```
`define DATA_WIDTH 32

module top_module(
    input [`DATA_WIDTH - 1:0] A,
    input [`DATA_WIDTH - 1:0] B,
    input [2:0] ALUop,
    output Overflow,
    output CarryOut,
    output Zero,
    output [`DATA_WIDTH - 1:0] Result
);
```

```
12
13      wire [`DATA_WIDTH - 1:0] and_result, or_result, add_result, sub_result,
            slt_result;
14      wire add_carryout, sub_carryout;
15      wire add_overflow, sub_overflow;
16
17      // 基本逻辑运算
18      assign and_result = A & B;
19      assign or_result = A | B;
20
21      // 加减法运算
22      assign {add_carryout, add_result} = A + B;
23      assign {sub_carryout, sub_result} = A + (~B + 1'b1);
24
25      // 溢出检测
26      assign add_overflow = (~A[`DATA_WIDTH-1] & ~B[`DATA_WIDTH-1] &
            add_result[`DATA_WIDTH-1]) |
27                            (A[`DATA_WIDTH-1] & B[`DATA_WIDTH-1] &
                                 ~add_result[`DATA_WIDTH-1]);
28      assign sub_overflow = (~A[`DATA_WIDTH-1] & B[`DATA_WIDTH-1] &
            sub_result[`DATA_WIDTH-1]) |
29                            (A[`DATA_WIDTH-1] & ~B[`DATA_WIDTH-1] &
                                 ~sub_result[`DATA_WIDTH-1]);
30
31      // SLT实现
32      wire A_neg = A[`DATA_WIDTH-1];
33      wire B_neg = B[`DATA_WIDTH-1];
34      wire diff_sign = A_neg ^ B_neg;
35
36      // 符号不同时直接比较符号位
37      wire slt_diff_sign = A_neg & ~B_neg;
38
39      // 符号相同时使用减法结果
40      wire [`DATA_WIDTH-1:0] sub_for_slt = A + (~B + 1'b1);
41      wire slt_same_sign = sub_for_slt[`DATA_WIDTH-1];
42
43      assign slt_result = diff_sign ? (slt_diff_sign ? 32'b1 : 32'b0)
44                                    : (slt_same_sign ? 32'b1 : 32'b0);
45
46      // 结果选择
```

```
    assign Result = (ALUop == 3'b000) ? and_result :
                    (ALUop == 3'b001) ? or_result :
                    (ALUop == 3'b010) ? add_result :
                    (ALUop == 3'b110) ? sub_result :
                    (ALUop == 3'b111) ? slt_result :
                    32'b0;

    // 标志输出
    assign Overflow = (ALUop == 3'b010) ? add_overflow :
                      (ALUop == 3'b110) ? sub_overflow :
                      1'b0;

    assign CarryOut = (ALUop == 3'b010) ? add_carryout :
                      (ALUop == 3'b110) ? sub_carryout :
                      1'b0;

    assign Zero = (Result == 32'b0) ? 1'b1 : 1'b0;

endmodule
```

1.2 关键代码解释

1.2.1 寄存器堆 Register 设计

声明 32x32bit 通用寄存器，[‘DATA_WIDTH-1:0] 代表每个寄存器位宽为 32 位宽;(1«‘ADDR_WIDTH) 为通过地址位宽计算地址空间大小 ($2^5 = 32$)，[0:(1«‘ADDR_WIDTH)-1] 即索引为 0 到 31 的数组。

```
    reg [`DATA_WIDTH-1:0] reg_file[0:(1<<`ADDR_WIDTH)-1];
```

读操作（组合逻辑）。按要求，从 0 号地址读出的数据应为常量 $32'b0$，因此首先判断地址是否为 0 号地址，若不为 0，则读出地址为 raddr 的寄存器中的数据。

```
    assign rdata1=(raddr1==0)?32'b0:reg_file[raddr1];
    assign rdata2=(raddr2==0)?32'b0:reg_file[raddr2];
```

写操作（时序逻辑）。仅当 wen=1（有效）且 waddr 不等于 0 时，才向 waddr 对应的寄存器写入 wdata。

```
    always @(posedge clk) begin
        if (wen==1&&(waddr!=0)) begin
            reg_file[waddr]<=wdata;
        end
```

```
    end
```

1.2.2 运算器 ALU 设计

减运算。按要求需要通过加运算实现。$A-B=A+(-B)$, 通过 $\sim B+1$ 计算得 $-B$ 的补码。由 $A+(\sim B+1)$ 即可实现加法运算下的减运算。

```
    assign {sub_carryout, sub_result} = A + (~B + 1'b1);
```

溢出检测。对于加运算，**正数 + 正数 = 负数**或**负数 + 负数 = 正数**都代表发生了溢出；对于减运算，**负数-正数 = 正数**或**正数-负数 = 负数**都代表发生了溢出。

```
    assign add_overflow = (~A[`DATA_WIDTH-1] & ~B[`DATA_WIDTH-1] &
       add_result[`DATA_WIDTH-1]) | (A[`DATA_WIDTH-1] & B[`DATA_WIDTH-1] &
       ~add_result[`DATA_WIDTH-1]);
    assign sub_overflow = (~A[`DATA_WIDTH-1] & B[`DATA_WIDTH-1] &
       sub_result[`DATA_WIDTH-1]) | (A[`DATA_WIDTH-1] & ~B[`DATA_WIDTH-1] &
       ~sub_result[`DATA_WIDTH-1]);
```

基于加运算的有符号整数比较。首先单独比较符号位，如果 A、B 符号位不同，则符号位为 0 的数大，直接输出结果；如果符号位相同，两数相减，再比较结果 sub_for_slt 的符号位 slt_same_sign，如果为 1 则 $A<B$; 如果为 0 则 $A\geq B$。

```
    wire A_neg = A[`DATA_WIDTH-1];
    wire B_neg = B[`DATA_WIDTH-1];
    //比较符号位是否相同
    wire diff_sign = A_neg ^ B_neg;

    // 符号不同时直接比较符号位
    wire slt_diff_sign = A_neg & ~B_neg;

    // 符号相同时使用减法结果
    wire [`DATA_WIDTH-1:0] sub_for_slt = A + (~B + 1'b1);
    wire slt_same_sign = sub_for_slt[`DATA_WIDTH-1];

    assign slt_result = diff_sign ? (slt_diff_sign ? 32'b1 : 32'b0)
                                  : (slt_same_sign ? 32'b1 : 32'b0);
```

2 实验中遇到的问题

1. 有符号整数比较的实现。

对于该部分的设计主要经历了三次改动。

(a) 初次设计

```
1            assign {compare_carryout,slt_result}=A+(~B+32'b1);
2         assign slt_result=compare_carryout?32'b1:32'b0;
```

主要思路是通过 A-B 结果的符号位来判断是否发生了借位，若发生了借位则 $A+(-B)<0, A<B$，输出结果为 $result=1$; 反之输出结果为 $result=0$。但这种设计只检查了进位标志，没有正确实现有符号比较，应该检查结果的符号位和溢出标志。

(b) 第二次设计

```
1     assign sub_result = A + (~ B + 1)
2     wire slt_overflow = (~A[`DATA_WIDTH-1] & B[`DATA_WIDTH-1]) |
         (A[`DATA_WIDTH-1] & ~B[`DATA_WIDTH-1] &
         sub_result[`DATA_WIDTH-1]) |(~A[`DATA_WIDTH-1] &
         B[`DATA_WIDTH-1] & ~sub_result[`DATA_WIDTH-1]);
3     assign slt_result = (sub_result[`DATA_WIDTH-1] ^ slt_overflow) ?
         32'b1 : 32'b0;
```

在第二次的设计中,slt_overflow 为溢出标志，共有三种溢出情况：

- A 正 B 负：正常结果应为负，若为正则溢出。
- A 负 B 正：正常结果应为正，若为负则溢出。
- A 正 B 负但结果为正：明确溢出。

最终比较结果 slt_result 与溢出标志的异或结果决定大小：

- 若符号位与溢出标志不同，说明 $A<B$。
- 若相同（0），说明 $A\geq B$。

但这种设计在测试时出现了问题：

```
1     ERROR: A = 80000000, B = ffffffff, ALUop = 7, Result = X0000001,
         Reference = 00000001.
```

这是因为当 A=0x80000000（最小负整数），在 Verilog 中，当运算结果超出 32 位表示范围时，最高位可能变为不定态（X）。

(c) 第三次设计

```
1     // 符号不同时直接比较符号位
2     wire slt_diff_sign = A_neg & ~B_neg;
3
4     // 符号相同时使用减法结果
5     wire [`DATA_WIDTH-1:0] sub_for_slt = A + (~B + 1'b1);
```

```
6     wire slt_same_sign = sub_for_slt[`DATA_WIDTH-1];
7     assign slt_result = diff_sign ? (slt_diff_sign ? 32'b1 : 32'b0)
         : (slt_same_sign ? 32'b1 : 32'b0);
```

当 A 是最小负整数 (0x80000000) 时，减去任何正数都会导致算术溢出。在第三次的设计中，首先检查操作数的符号位；如果符号不同，直接比较符号位（负数小于正数）；如果符号相同，才使用减法结果进行比较；完全避免了减法溢出导致的问题。

3 实验心得

在本次基于 Verilog 的矩阵密码加解密实验过程中，我深刻体会到硬件描述语言设计与传统软件编程的显著差异。实验初期，面对矩阵转置和列置换的算法实现，我过于依赖软件编程思维，导致在变量命名和时序控制上屡屡碰壁。特别是在实现补码减法运算时，最初未能理解硬件层面” 取反加一” 的本质意义，仅将其视为数学公式的简单转换，直到通过 ModelSim 波形仿真观察到实际运算过程，才真正领悟到这种设计在数字电路中的精妙之处。

在实现有符号数比较模块时，我遭遇了更为复杂的技术挑战。当测试用例中出现 32 位最小负数的边界情况时，模块输出结果与预期不符。经过细致的波形分析和理论推演，发现问题源于补码表示范围的非对称性导致的溢出判断失误。这一问题的解决过程让我深刻认识到，硬件设计必须严格考虑所有可能的极端情况，任何逻辑疏漏都会在实际电路中表现为功能异常。通过重构比较逻辑，将符号位判断与数值比较分离，最终实现了稳定可靠的比较功能。