浙江大学

本科实验报告

课程名称：	数字逻辑设计
姓　　名：	
学　　院：	计算机科学与技术学院
专　　业：	计算机科学与技术
邮　　箱：	
QQ　　号：	
电　　话：	
指导教师：	蔡铭
报告日期：	2023 年 12 月 17 日

浙江大学实验报告

课程名称：数字逻辑设计 实验类型：综合

实验项目名称：寄存器和寄存器传输设计

学生姓名： 学号： 同组学生姓名：

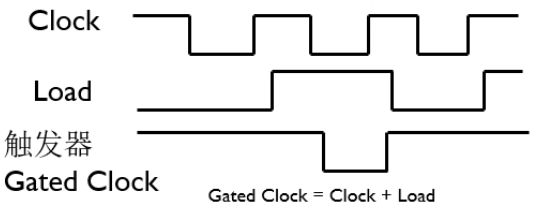
实验地点：紫金港东四 509 室 实验日期：2023 年 12 月 13 日

一、操作方法和实验步骤

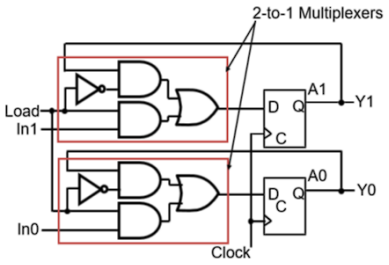
实验背景

- **寄存器**是一组二进制存储单元，一个寄存器可以用于存储多位二进制值，通常用于进行简单数据存储、移动和处理等操作。最基本的操作为**读和写**，其中写操作通常在时钟边沿时将接受的输入值存储在触发器中。

- **采用门控时钟的寄存器**指通过一个或多个控制信号控制门电路的打开和关闭，只有在门控打开时才能对寄存器的值进行修改，一个简单的实现是使用一个信号 Load 作为门控信号，Load 与时钟信号的与为门控时钟信号，即如果 Load 信号为 1，则允许时钟信号通过，如果为 0 则阻止时钟信号通过。



- 一个更稳定的方式是“采用 Load 控制反馈的寄存器”，它不对时钟信号进行处理，而是在触发器的输入端使用 2-1 多路选择器从**触发器的值**与**输入的值**之间进行选择。以下是存储两位数据的寄存器的原理图。



采用 Load 控制反馈的寄存器

- 我们用 verilog 代码实现有选择地加载寄存器

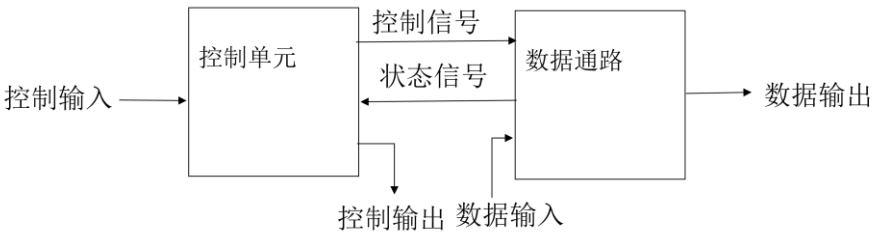
```
`timescale 1ns / 1ps

module MyRegister4b(
        input wire clk,
        input wire [3:0] IN,
        input wire Load_A,
        output reg [3:0] A
    );

    initial A = 4'b0000;

    // only can Load = 1'b1,
    // and the posdge clk
    // the register can be passed
    always @ (posedge clk) begin
        if (Load_A) A <= IN;
    end
endmodule
```

- 可以看到，只有当 Load 信号为 1 时，我们才将 In 的值传递给 A，否则就保持 A 不变。

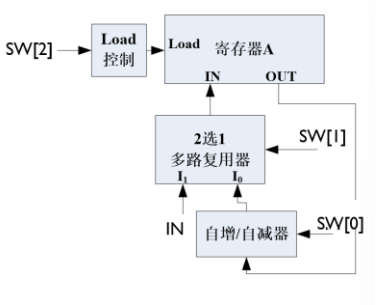
寄存器传输

- 寄存器传输，顾名思义就是将寄存器中的数据进行传输和梳理。它的控制输入用于控制基本的寄存器操作（例如加载、计数、移位、加法、按位），接下来在数据通路中寄存器完成对应的操作并将数据进行输出。

- 以下是寄存器传输的基本原理图



- 以下是我们要实现的寄存器传输模块：



- 这个模块由以下几个部分组成：

    1. Load 控制模块，在 SW[2] 的上升沿产生一个时钟周期宽度的 Load 信号
    2. 自增/自减器，作为控制器进行数据输入
    3. 2-1 多路复用器，选择对应的输入信号输入寄存器中进行加载

- 其中，Load 模块的任务我们将在 Load_Gen 模块中完成，以下是该模块的 verilog 代码：

```
`timescale 1ns / 1ps

module Load_Gen(
    input wire clk,
    input wire btn_in,
    output reg Load_out
    );

     initial Load_out = 0;
     reg old_btn;

     always@(posedge clk) begin

        // old_btn to store the oldder state
        old_btn <= btn_in;

        // only if the btn is changed to 1'b1
        // within 1 clk, the Load_out = 1'b1
        if ( btn_in == 1'b1 && old_btn == 1'b0)
            Load_out <= 1'b1;
        else
            Load_out <= 1'b0;
     end

endmodule
```

- 可以看到，该模块只在时钟的上升沿触发一次，old_btn 用于存储上一次的输入信息，Load_out 用于输出控制信号。只有当控制信号由 0 变 1 时，控制信号才起作用，其他时刻都无效。

- 多路复用器的实现我们可以用以下代码来简化：

```
assign A_IN = (SW[1] == 1'b0)? Al : In;
```

- 接下来，我们完成 top 模块就可以实现寄存器传输的基本功能：

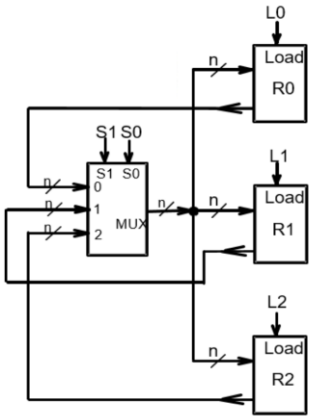
```
`timescale 1ns / 1ps

module TopModule(
        input clk,
        input [2:0] SW,
        input [3:0] In,
        output [3:0] A
    );

    wire [3:0] Al;
    wire Load_A;
    wire [3:0] A_IN;
    wire Co;

    MyRegister4b RegA(.clk(clk), .IN(A_IN), .Load_A(Load_A), .A(A));

    Load_Gen m0(.clk(clk), .btn_in(SW[2]), .Load_out(Load_A));

    AddSub4b m1( .Ctrl(SW[0]), .A(A), .B(4'b0001), .S(Al));

    // mux 2 to 1
    assign A_IN = (SW[1] == 1'b0)? Al : In;

endmodule
```

- 其中，各引脚的功能如下：

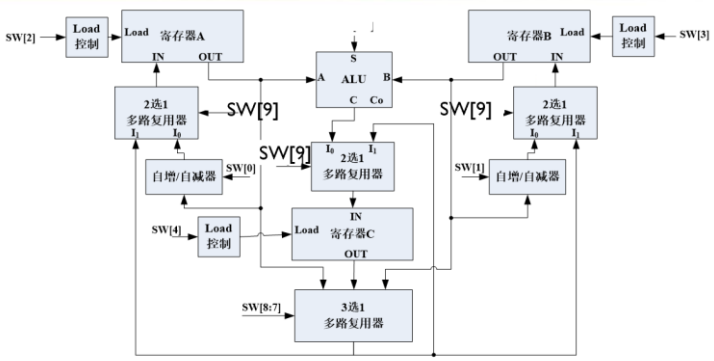
    1. SW[2]， 寄存器加载
    2. SW[1]， 寄存器清零
    3. SW[0]， 向上/下计数

基于 ALU 的数据传输应用设计

- 本次实验使用多路选择器总线进行寄存器传输，寄存器的值“汇总”到一个多路选择器并通过选择信号选择一个值作为三个寄存器的输入，当 Load 信号升起时对对应寄存器的值进行修改。



- 最终，我们将 ALU 和 寄存器传输模块进行组合之后就可以得到以下原理图。



- 我们将需要用到的模块导入之后完成 top 模块的书写。

```
`timescale 1ns / 1ps

module Trans(
        input clk,
        input [9:0] SW,
        output [3:0] A,
        output [3:0] B,
        output [3:0] C,
        output [3:0] Out
    );

     wire [3:0] I0;
     wire Co;
     wire Load_C;
     wire [3:0] In;

     TopModule t1(.clk(clk), .SW({SW[2], SW[9], SW[0]}), .In(Out), .A
(A));
     TopModule t2(.clk(clk), .SW({SW[3], SW[9], SW[1]}), .In(Out), .A
(B));

     ALU a(.A(A), .B(B), .res(I0), .op(SW[6:5]),  .Cout(Co));

     // mux 2 to 1
     assign In = (SW[9] == 0)? I0 : Out;

     MyRegister4b c(.clk(clk), .IN(In), .Load_A(Load_C), .A(C));
     Load_Gen l(.clk(clk), .btn_in(SW[4]), .Load_out(Load_C));

     Mux4to1b4 m(.s(SW[8:7]), .I0(A), .I1(B), .I2(C), .I3(4'b0000), .O
(Out));

endmodule
```

二、实验结果和分析

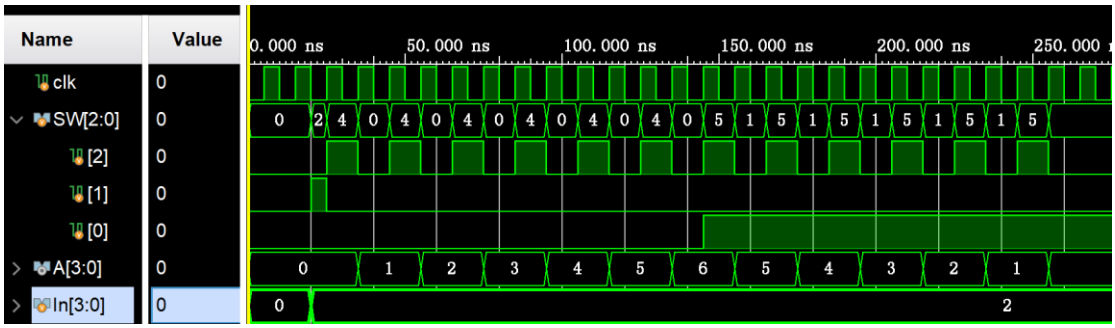
采用寄存器传输原理设计的计数器仿真激励

- 我们利用助教给出的仿真激励代码进行仿真激励

```
`timescale 1ns / 1ps

module TopModule_sim;

    // Inputs
    reg clk;
    reg [2:0] SW;
    reg [3:0] In;

    // Outputs
    wire [3:0] A;

    // Instantiate the Unit Under Test (UUT)
    TopModule uut (
        .clk(clk),
        .SW(SW),
        .In(In),
        .A(A)
    );

    // clock
    always begin
        clk = 0;#5;
        clk = 1;#5;
    end
    integer i;
    initial begin
        // Initialize Inputs
        clk = 0;
        SW = 0;
        In = 0;

        // Wait 100 ns for global reset to finish
        #20;

        // Initial
        In = 4'b0010;
        SW[1] = 1'b1;#5;

        // begin
        SW[1] = 1'b0;
        SW[0] = 1'b0;
        for(i=0;i<=5;i=i+1)begin
            SW[2] = 1;#10;
            SW[2] = 0;#10;
        end

        SW[0] = 1'b1;
        for(i=0;i<=5;i=i+1)begin
            SW[2] = 1;#10;
```

```
            SW[2] = 0;#10;
        end
    end

endmodule
```

- 以下是仿真激励的结果



- 可以看到，刚开始的初始化状态 A 的输出为 1。接下来我们将 SW[1] 置 1，则 A 输出 In 的结果（此时为 0）。接下来 SW[2] 交替置 1，多路选择器选择自增自减器，寄存器实现自增计数的功能。而后我们将 SW[0] 置 0，则自增自减器进行减法操作。仿真验证的结果和我们的预期符合得比较好。

基于 ALU 的数据传输应用仿真激励

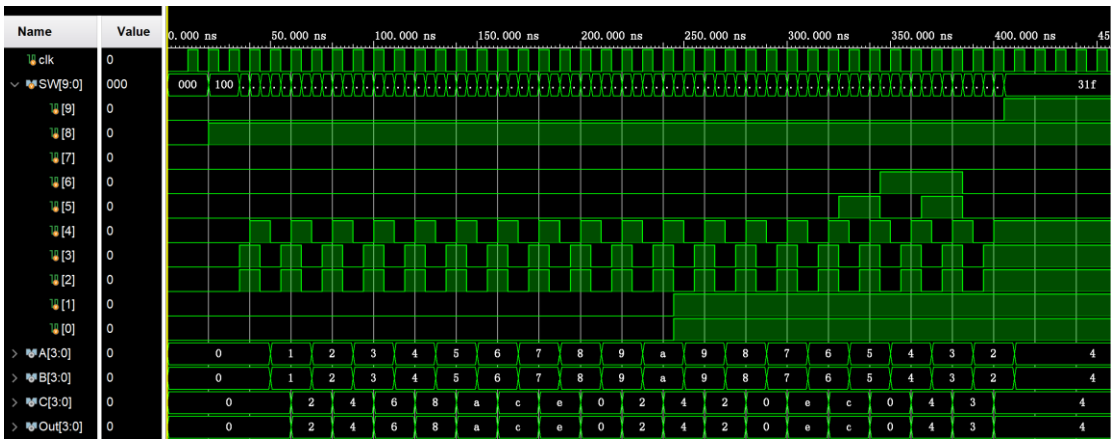
- 以下是助教提供的仿真激励的代码

```
`timescale 1ns / 1ps

module Trans_sim;
    // Inputs
    reg clk;
    reg [9:0] SW;

    // Outputs
    wire [3:0] A;
    wire [3:0] B;
    wire [3:0] C;
    wire [3:0] Out;

    // Instantiate the Unit Under Test (UUT)
    Trans uut (
        .clk(clk),
        .SW(SW),
        .A(A),
        .B(B),
        .C(C),
        .Out(Out)
    );

    integer i;
    // clock
    always begin
        clk = 1; #5;
        clk = 0; #5;
```

```
end
initial begin
    // Initialize Inputs
    clk = 0;
    SW = 0;

    // Wait 100 ns for global reset to finish
    #20;

    // initial
    // ALU control
    // 自增
    SW[8:7] = 2'b10;
    SW[9] = 1'b0;#5;
    for( i=0; i<=10; i=i+1)begin
        SW[3:2] <= 2'b00;#5;
        SW[4] <= 1'b0;#5;
        SW[3:2] <= 2'b11;#5;
        SW[4] <= 1'b1;#5;
    end

    // 自减
    SW[1:0] = 2'b11;
    for( i=0; i<=3; i=i+1)begin
        SW[3:2] <= 2'b00;#5;
        SW[4] <= 1'b0;#5;
        SW[3:2] <= 2'b11;#5;
        SW[4] <= 1'b1;#5;
    end

    // ALU 的其他算法
    SW[6:5] = 2'b01;
    SW[3:2] <= 2'b00;#5;
    SW[4] <= 1'b0;#5;
    SW[3:2] <= 2'b11;#5;
    SW[4] <= 1'b1;#5;
    SW[6:5] = 2'b10;
    SW[3:2] <= 2'b00;#5;
    SW[4] <= 1'b0;#5;
    SW[3:2] <= 2'b11;#5;
    SW[4] <= 1'b1;#5;
    SW[6:5] = 2'b11;
    SW[3:2] <= 2'b00;#5;
    SW[4] <= 1'b0;#5;
    SW[3:2] <= 2'b11;#5;
    SW[4] <= 1'b1;#5;
    SW[6:5] = 2'b00;
    SW[3:2] <= 2'b00;#5;

    // 换回加法
    SW[4] <= 1'b0;#5;
    SW[3:2] <= 2'b11;#5;
    SW[4] <= 1'b1;#5;

    // 寄存器之间交流
    SW[9] = 1'b1;#5
    SW[3:2] <= 2'b11;#5;
    SW[4] <= 1'b1;#5;
```

```
    end
endmodule
```

- 以下是仿真激励的结果



- 可以看到，刚开始的初始化状态 `A, B, C, Out` 均置 0。接下来 `SW[2], SW[3], SW[4]` 交替置 1，更新 `A, B, C` 的寄存器数据。其中，`ALU` 完成加法操作（`A + B = C`）。
- 在 `250ns` 后，`SW[0], SW[1]` 置 1，自增器进入减法状态，`A, B` 在每次 `Load` 信号中都更新其寄存器数据为自减 1。可以看到仿真结果和预期相同。
- 接下来，我们验证 `ALU` 的运算功能。在 `300ns` 之后， `ALU` 分别进行了减法、按位与、按位或操作，分别对应 `5 - 5 = 0`， `4 & 4 = 4`，`3 | 3 = 3`。
- 最后，我们实现寄存器之间的数据通信，将 `SW[9]` 置 1，让 `C` 同步 `A, B` 的数据。
- 值得注意的是，在这期间，`SW[8:7] = 10`，即输出选择 `C` 的数据信号。

三、讨论、心得

- 这次的实验相比之前的几次略微有些复杂。由于我使用 `Vivado` 完成实验，因此刚开始为了理清助教给的工程文件中的模块逻辑就花了不少的时间。
- 接下来，在书写代码的过程中，也遇到了一些问题。例如考虑清楚 `Load` 模块的时序顺序问题，后面是在仿真激励的过程中得到验证。
- 在仿真激励的过程中，可能是由于激励代码的设计还不够好，导致 `Load` 信号的加载与 `clk` 的振荡频率和起点相同，因此数据的加载时刻来到了 `Load` 信号的末尾、下一个 `clk` 信号的开头。
- 这也是最后一个小实验了。希望接下来的大程能够顺利完成。