实验报告——寄存器指令初步

一、实验设计

本次实验相较前两次比较复杂一些，在本次实验中，我一共建了 4 个 module，下面分别解释每个 module 的功能和实现，并在截图中给出部分代码实现：

1. **module ALU**，这个模块实现的是要求中的四种操作，+/-/*/取反:、
   a) **输入**：操作符 op（2-bit），和两个操作数 q1 和 q2（8-bit）
   b) **输出**：两个操作数（或其中之一）在操作符下的运算结果，也是 8-bit
   c) 因为共有四种运算，所以此处 op 用 2 位表示，0 表示 q1+q2，1 表示 q1-q2，2 表示对 q1 取反，3 表示 q1*q2
   d) 因为具体实现在第一次 lab 中已经基本完成，因此直接上代码：

```
module ALU(
    input [1:0]op,
    input [7:0]q1, q2,
    output  reg [7:0]out
    );

    always @(*) begin
        case(op)
            2'b0: out = q1 + q2;
            2'b01: out = q1 - q2;
            2'b10: out = ~q1;
            default: out = q1 * q2;
        endcase
    end

endmodule
```

2. **module regFile**，这个 module 对应的是文档中的 RegFile，主要实现对 32 个寄存器内容的存储，读取 n1 和 n2 表示寄存器（以下简称 n1 寄存器和 n2 寄存器）中的值，将 DI 写入 n1 对应的寄存器
   a) **输入**：表示复位的 reset(1-bit)，表示是否写入的 WE(1-bit)，表示时钟的 clk(1-bit),两个寄存器的地址 n1 和 n2(5-bit)，要写入 n1 表示的寄存器的值 DI(8-bit)
   b) **输出**：当对 n1 和 n2 寄存器读的时候，返回两个寄存器中的值 q1 和 q2(8-bit)
   c) 寄存器中的值用一个包含 32 个 8-bit 的数的**二维数组 data** 表示

```
reg [7:0] data[0:31];
```

   d) **分频处理**：

      为了能够看到 q 值的变化，我们对 clk 做了分频处理，降低 clk 的频率到能够接受的范围（2-3s）；

      像文档中给出的那样，借助了一个新的时钟 clk2 和一个计数器

count，初始值都为 0，在每次 clk 的上升沿时，都会计数器增加一，当 count 加到 2e8 时，对 clk2 取反，count 复位（变为 0）；

这样的过程表示 count 个 clk 对应的时钟周期，clk2 变化一次，之后利用 clk2 的上升沿和下降沿执行相应的读写操作，相当于将时钟周期降低为原来的 1/count

实现代码如下：

```
reg clk2 = 0;
always @(posedge clk) begin
    count = count + 1;
        if(count > 200000000) begin
            clk2 = ~clk2;
            count = 0;
        end
end
```

e) 时钟上升沿读出 n1 寄存器和 n2 寄存器的值，借助分频处理中产生的 clk2，当 clk2 处于上升沿时，读出数据：

```
always @(posedge clk2) begin
      q1 = data[n1];
      q2 = data[n2];
end
```

f) 时钟下降沿时将 DI 值写入 n1 寄存器，若 reset=1，将所有寄存器的值复位（变为 1）：

```
always @(negedge clk2) begin
   if(WE) begin
         if(reset)
            data[n1] = 1;
         else
            data[n1] = DI;
      end
   if(reset) begin
      for(i = 0; i < 32; i = i + 1) begin
         data[i] = 1;
      end
   end
end
```

g) **注**：此处之所以将 reset 和写入操作放在同一个 always 块中，是因为 verilog 不允许在多个 always 块中对同一个数据对象做赋值；因为 reset 也是一种特殊的写操作（所有寄存器都写入 1），所以此处和 DI 的写操作放在一起，这不一定是解决信号多驱动问题的最佳解决方案，但是这是我目前想到的最好的实践

3. **module display**，该模块实现了用七段显示管显示 Q1 和 Q2

a) **输入**：时钟 clk(1-bit)，两个要显示的数值 Q1 和 Q2(8-bit)

b) **输出**：是否显示小数点 DP(1-bit)，两个 8 位的数用四个七段显示管显示 AN(4-bit)，每一位表示对应七段显示管是否被选择，七段显示管对应的七个管脚 A2G(7-bit,a,b,c,d,e,f,g)

c) **分时显示**：根据七段显示管的原理，如果要是每一个七段显示管显示不同的信息，需要做分时处理，借助一个 20 位的 clkdiv，在每一次时钟上升沿，都对 clkdiv 增加一，那么，因为权重不同，对应位变化的频率也不同，所以可以用不同的位表示不同的频率，也就是不同的周期

在本次实验中，对于显示管的刷新频率，我取了 190Hz（高于人眼所能接受的最低频率，24Hz）,对应于 clkdiv 也就是第 18 位，剩余两位 s(2-bit)用来表示显示 Q1 和 Q2 的那一部分，显示在哪一个七段显示管上，s 与 Q1 和 Q2 以及与四个七段显示管的对应关系如下：

```
always @(s) begin
    case(s)
        1:digit = Q1[3:0];
        2:digit = Q2[7:4];
        3:digit = Q2[3:0];
        default:digit = Q1[7:4];
    endcase
end

always @(*) begin
    AN = 4'b1111;
    AN[s] = 0;
end
```

d) 根据 s 得到对应要显示的数值 digit 之后，可以通过 case 语句选择{a,b,c,d,e,f,g}的显示方式：

```
always @(*) begin
    case(digit)
        4'h1: A2G = 7'b1111001;
        4'h2: A2G = 7'b0100100;
        4'h3: A2G = 7'b0110000;
        4'h4: A2G = 7'b0011001;
        4'h5: A2G = 7'b0010010;
        4'h6: A2G = 7'b0000010;
        4'h7: A2G = 7'b1111000;
        4'h8: A2G = 7'b0000000;
        4'h9: A2G = 7'b0010000;
        4'hA: A2G = 7'b0001000;
        4'hB: A2G = 7'b0000011;
        4'hC: A2G = 7'b1000110;
        4'hD: A2G = 7'b0100001;
        4'hE: A2G = 7'b0000110;
        4'hF: A2G = 7'b0001110;
        default: A2G = 7'b1000000;
    endcase
end
```

e） **注**：我们实验中用到的电路板是低电平有效

4. module main：这一模块实现的是顶层的控制，输入就是总输入，输出就是总输出，不再赘述，直接上代码：

```
module main(
    input reset, we, clk,
    input [1:0]op,
    input [4:0]n1, n2,
    output DP,
    output [3:0] AN,
    output  [6:0] A2G
    );

    wire [7:0]q1, q2;
    wire [7:0] di;

    regFile regfile(reset, we, clk, n1, n2, di, q1, q2);
    ALU alu(op, q1, q2, di);
    display displayt(clk, q1, q2, DP, AN, A2G);
endmodule
```

二、仿真波形

1. module ALU：

①**测试代码：**

该模块比较简单，在 LAB1 中已经实现过了，因此在此处只做了简单的测试，取了 Q1=0x80，Q2=0x7F，分别做加减乘和取反运算

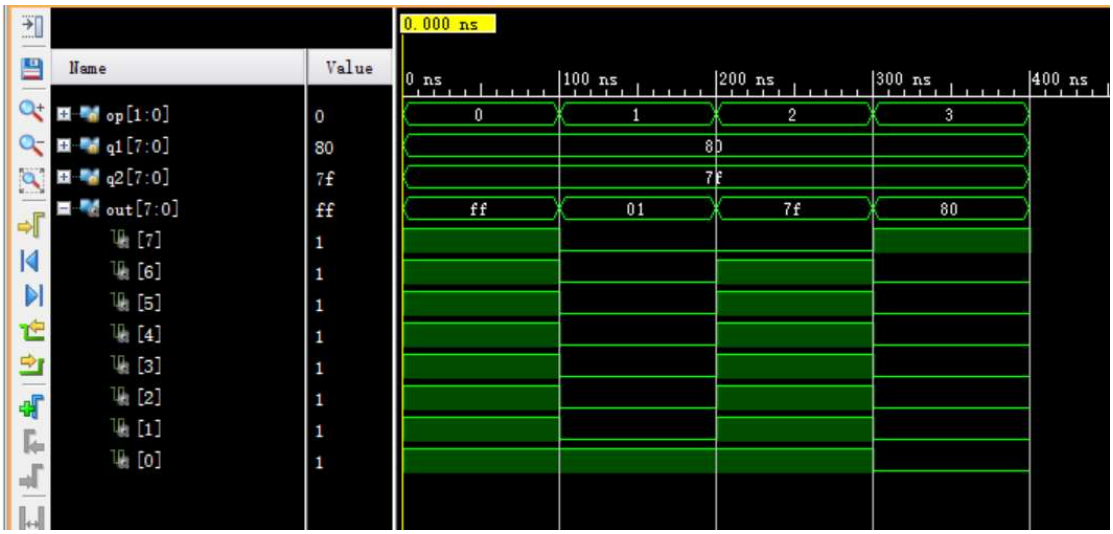
```
module test_ALU();
    reg [1:0]op;
    reg [7:0]q1, q2;
    wire [7:0]out;
    ALU test(op, q1, q2, out);
    initial begin
        op = 2'b00;
        q1 = 8'b10000000;
        q2 = 8'b01111111;

        #100 op = 2'b01;

        #100 op = 2'b10;

        #100 op = 2'b11;
    end
    always begin
        #400 $finish;
    end
endmodule
```

②**仿真波形：**



2. module regFile：

①**测试代码：**

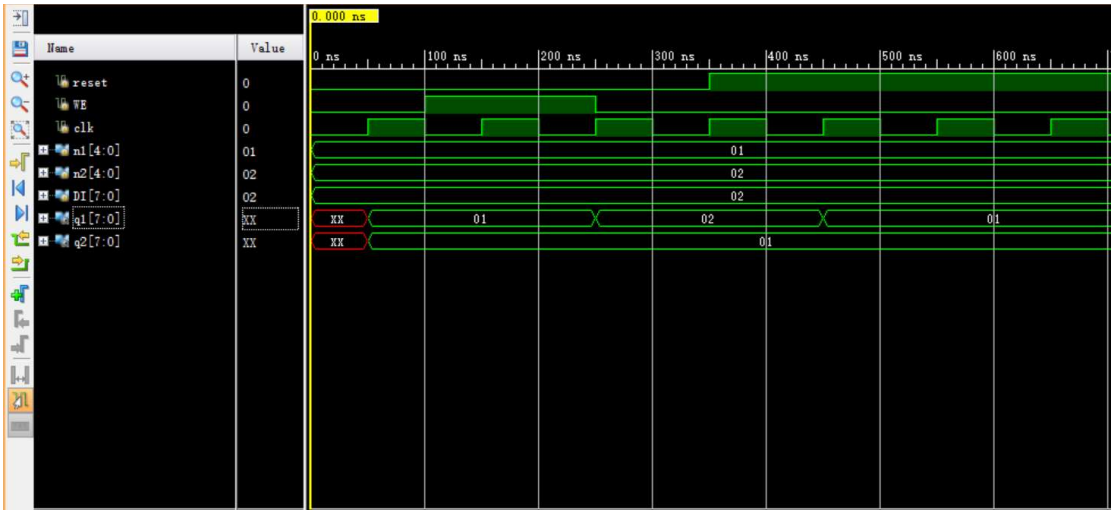
对该模块的测试，我测试了初始状态写的地址为 1 和 2 两个寄存器的值，WE 设为 1 后，将值 2 写入地址为 1 的寄存器之后，对应寄存器的值变化，以及 reset 之后，两个寄存器的值

```
module test_regFile();
    reg reset, WE, clk;
    reg [4:0]n1,n2;
    reg [7:0]DI;
    wire [7:0]q1,q2;
    regFile regfile(reset, WE, clk, n1, n2, DI, q1, q2);
    initial begin
        clk = 0;
        WE = 0;
        reset = 0;

        n1 = 4'b0001;
        n2 = 4'b0010;
        DI = 8'b00000010;

        #100 WE = 1;
        #150 WE = 0;
        #100 reset = 1;
    end
    always #50 clk = ~clk;
endmodule
```

②**仿真波形**：下图中 q1 的三个值对应未写入，写入和 reset 三个测试



3. module display：

①**测试代码**：

该测试比较简单，设定了 Q1=0x96，Q2=0x5A，将这两个数显示出来：

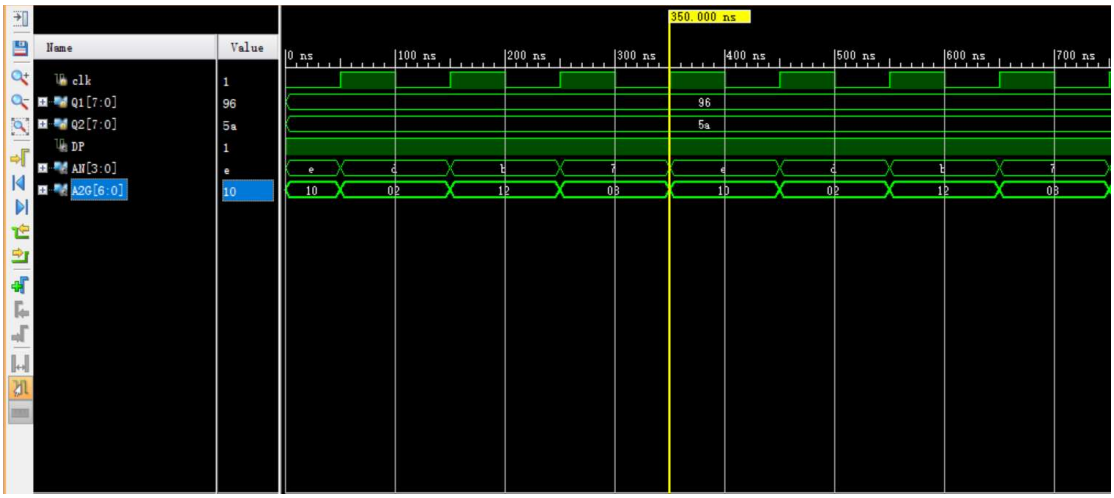
```
module test_display();
    reg clk;
    reg [7:0]Q1, Q2;
    wire DP;
    wire [3:0] AN;
    wire [6:0] A2G;

    display test(clk, Q1, Q2, DP, AN, A2G);

    initial begin
        clk = 0;
        Q1 = 8'b10010110;
        Q2 = 8'b01011010;
    end

    always #50 clk = ~clk;
endmodule
```

②仿真波形：



4. module main:

① **测试代码：**

因为涉及四种基本运算，所以我对这四种运算都进行了测试，每一个测试用到两个测试用例

以下列出的代码截图都在同一个 initial 块中，在模块中的顺序与下面列举顺序相同，整体的操作顺序就是（data 是 regfile 数组）

```
data[1]+data[1],
data[2]+data[1],
data[1]-data[1],
data[2]-data[1],
data[3]*data[1],
```

data[4]*data[2],
data[3]取反，
data[4]取反

a> **初始化：**

```
reset = 1;
clk = 0;
we = 0;
#10 reset = 0;
```

b> **加法：**第一种测试的是同一个寄存器上的加法；第二种测试的是不同寄存器上的加法

```
op = 0; n1 = 1; n2 = 1;
#15 we = 1;
#15 we = 0;

#35 op = 0; n1 = 2;
#10 we = 1;
#15 we = 0;
```

c> **减法：**第一个测的是同一个寄存器上的减法，第二个测的是不同寄存器上的减法

```
#55 op = 1; n1 =1; n2 = 1;
#10 we = 1;
#15 we = 0;

#55 op = 1; n1 = 2; n2 =1;
#10 we = 1;
#15 we = 0;
```

d> **乘法：**第一个测的是包含 0 的乘法，第二个是一般乘法

```
#55 op = 3; n1 = 3; n2 =1;
#10 we = 1;
#15 we = 0;

#55 op = 3; n1 = 4; n2 =2;
#10 we = 1;
#15 we = 0;
```

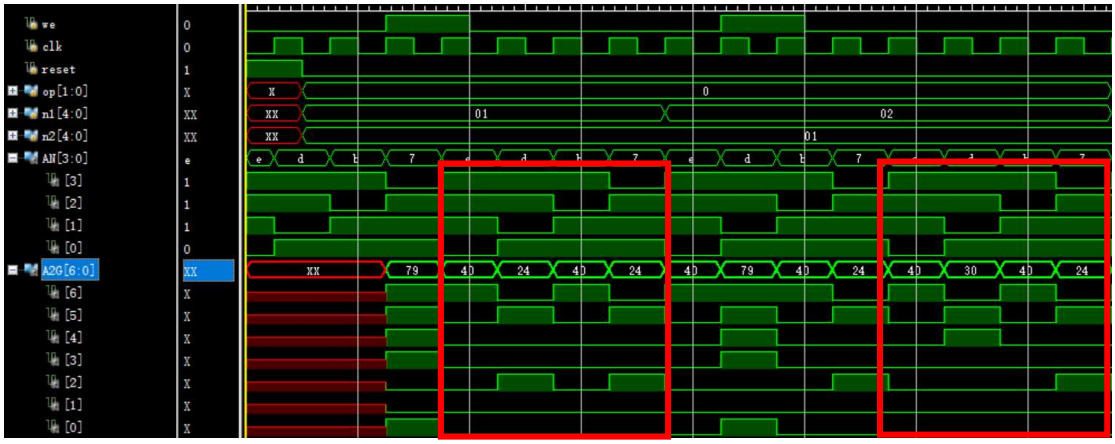
e> **取反：**第一个测的是对 0 的取反，第二个测试的对非 0 数的取反操作

```
#55 op = 2; n1 = 3; n2 = 2;
#10 we = 1;
#15 we = 0;

#55 op = 2; n1 = 4; n2 = 2;
#10 we = 1;
#15 we = 0;
```

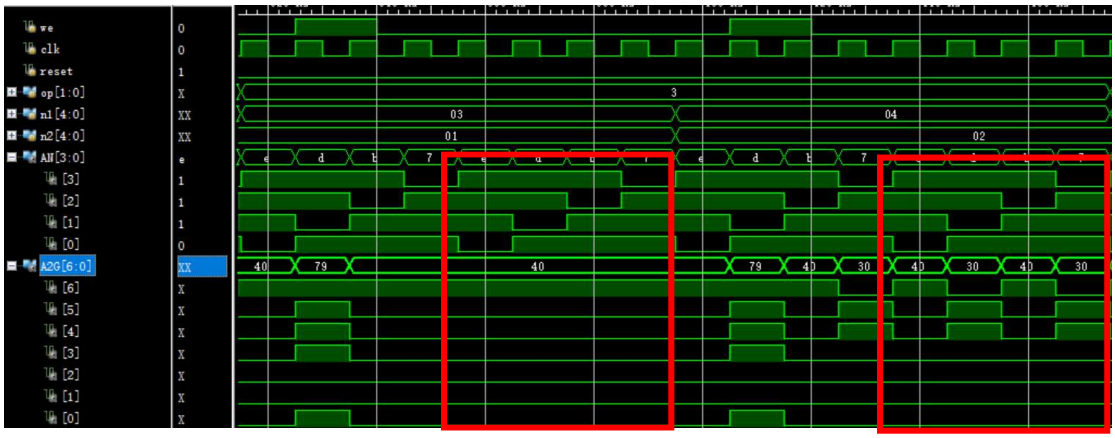
② **仿真波形**：（图中红色框框出的是每中测试用例下的 q1 和 q2 的值，前两位表示 q1，后两位表示 q2）

a> **加法**：



b> **减法**：

c> **乘法**：



d> **取反：**

三、实验中遇到的问题和解决方法

1.信号多驱动问题，指的是在不同的 always 块中对同一个数据对象进行写操作，这个问题出现在寄存器数组的两处写操作时，一处是 reset，另外一处是将 DI 的值写入 n1 寄存器，解决方法就是将两个写入同一个 always 块中，上面对 regFile 的说明中已经提到。

2.端口连接问题，因为之前的两个 lab 中都是单一的 module，不存在 module 的连接问题，但是在这个 LAB 中存在，去网上查了教程之后，概括如下：

将一个端口看成由相互链接的两个部分组成，一部分位于模块内部，另一部分位于模块外部。当在一个模块中调用另一个模块时：

输入端口：从模块内部来讲，输入端口必须为线网数据类型，从模块外部来看，输入端口可以连接到线网或者 reg 数据类型的变量。

输出端口：从模块内部来讲，输出端口可以是线网或者 reg 数据类型，从模块外部来看，输出必须连接到线网类型的变量，而不能连接到 reg 类型的变量。

3.我在写 regfile 时，为了解决信号多驱动问题，尝试将上升沿时间和下降沿事件合在一起写，结果出现了一些“怪异”的事情，查了资料了解到：如果在 always 的敏感列表中，同时包括一个信号的上升沿和它的下降沿的话，这两个事件会合并为一个电平事件。

四、实验的收获与感想

1. 掌握了分频处理的方法

2. 充分了解了七段显示管的原理，并可以用分时显示使七段显示管显示不同的数值

3. 了解了 verilog 中模块与模块连接时输入和输出的类型

4. 每一个模块都需要测试!!!

这次 LAB 相较前两次都比较复杂，尤其是七段显示管，了解了原理之后，具体实践仍有很多需要注意的地方。