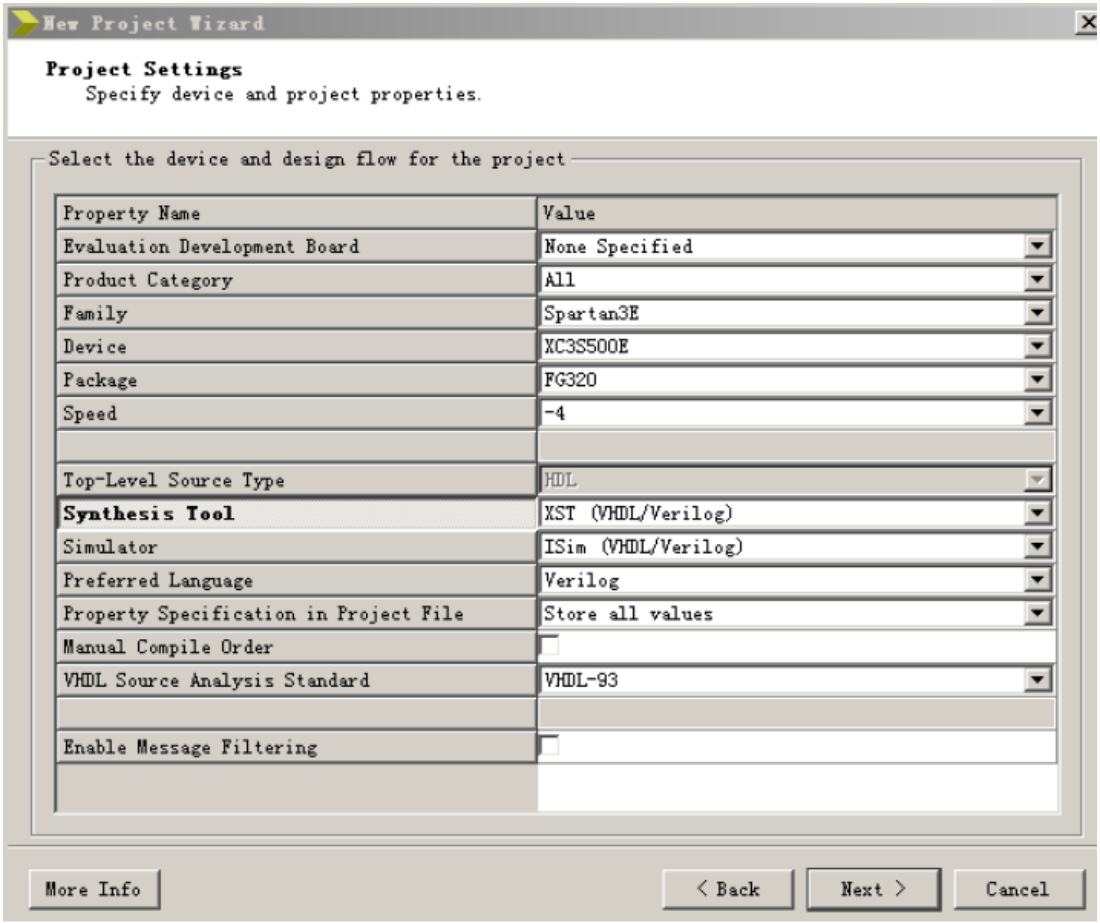
实验四实验报告

姓名：申继媛
学号：5130309194

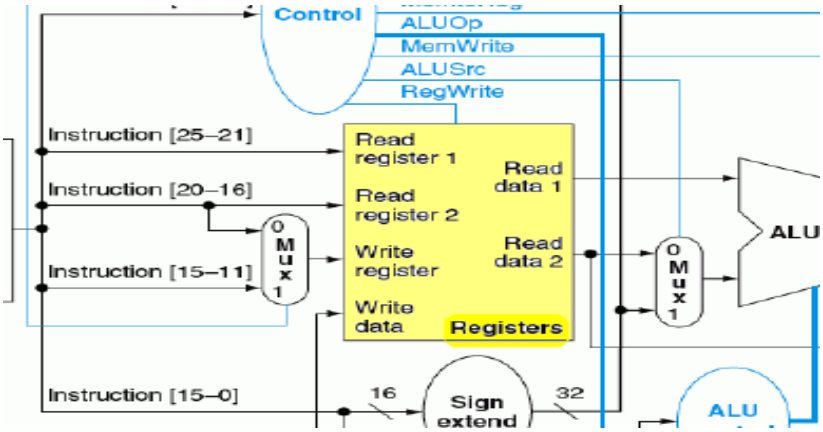
【实验环境】



【寄存器单元模块 REGISTER】

模块描述——

依照实验指导书给出如下原理图，我们可以发现 register 模块主要有一个时钟输入（clock_in，因为写操作），四个数据输入（readReg1、readReg2、writeReg、writeData），一个写信号输入（regWrite）以及两个读数据输出（readData1、readData2）。因为这是一个指令 regFile，还要定义一个 32bit一寄存器变量 regFile[31:0]。



实验过程——

（1） 在 register 模块中实现在任何时候都读取 data 和在时钟下降沿写入 data，这就需要用到 always@语句。

（2） register 代码截图如下：

```
module register(
    input clock_in,
    input [25:21]readReg1,
    input [20:16]readReg2,
    input [4:0]writeReg,
    input [31:0]writeData,
    input regWrite,
    output [31:0]readData1,
    output [31:0]readData2
    );
    reg [31:0] regFile[31:0];
    reg [31:0] readData1;
    reg [31:0] readData2;

always @ (readReg1 or readReg2 or readData1 or readData2 or regFile)
begin
   readData1 = regFile [ readReg1 ];
   readData2 = regFile [ readReg2 ];
end

always @(negedge clock_in)
begin
   if(regWrite==1)
      regFile[writeReg] = writeData;
end
endmodule
```

（3） test_for_register 代码截图如下：

```
    initial begin
        // Initialize Inputs
        clock_in = 0;
        readReg1 = 0;
        readReg2 = 0;
        writeReg = 0;
        writeData = 0;
        regWrite = 0;

        // Wait 100 ns for global reset to finish
        #180;
        regWrite = 1'b1;
        writeReg = 5'b10101;
        writeData = 32'b11111111111111110000000000000000;
        readReg1 = 5'b00000;
        readReg2 = 5'b00000;

        #200;
        regWrite = 1'b1;
        writeReg = 5'b01010;
        writeData = 32'b00000000000000001111111111111111;
        readReg1 = 5'b00000;
        readReg2 = 5'b00000;
```

```
        #200;
        regWrite = 1'b0;
        writeReg = 5'b00000;
        writeData = 32'b00000000000000000000000000000000;
        readReg1 = 5'b00000;
        readReg2 = 5'b00000;

        #200;
        readReg1 = 5'b10101;
        readReg2 = 5'b01010;
        // Add stimulus here

    end
    always
        begin
            #100;
            clock_in = ~clock_in;
        end;
```

测试代码解释：

运用 always 语句设置 clock_in 周期为 200ns，duty-cycle 为 50%。

先将所有信号和数据初始化为 0；

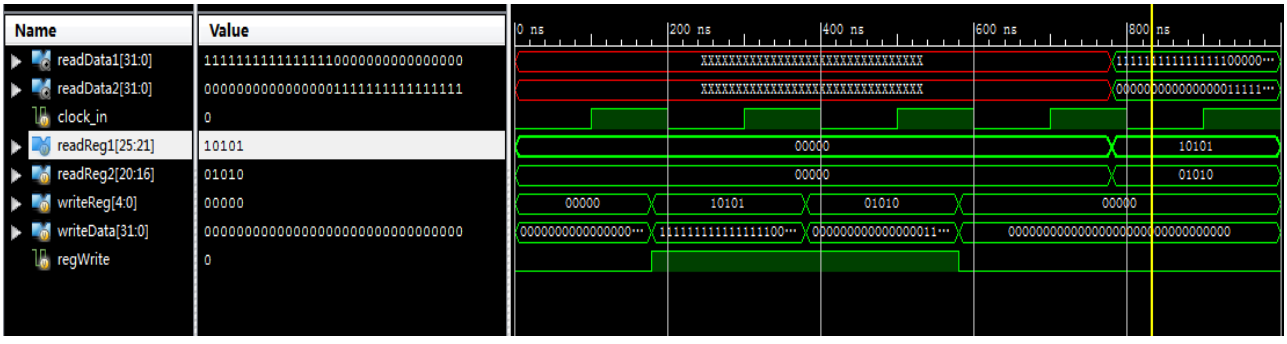
在 180ns 后（为了不与 clock 沿同时发生信号），将数据 32’b11111111111111110000000000000000 写入 regFile 的 5’b10101 处，并将两个 readReg 均置为 0。

在 200ns 后，将数据 32’b00000000000000001111111111111111 写入 regFile 的 5’b01010 处，并将两个 readReg 均置为 0。

在 200ns 后，将写信号置为 0，并将两个 readReg 均置为 0。

在 200ns 后，（写信号不变仍为 0），将 readReg 分别置为刚才写入的那两个寄存器处，即 regFile 的 5’b10101 和 5’b01010 处，则将读出刚才写入的值，预计结果为 readData1=32’b11111111111111110000000000000000,readData2=32’b00000000000000001111111111111111。

（4）在 simulation 的 hierarchy 窗口中选中 test_for_register 文件，并双击 Simulate Behavioral Model，得到仿真波形如下截图：



将仿真波形与测试预期比较，得到正确的实验结果。

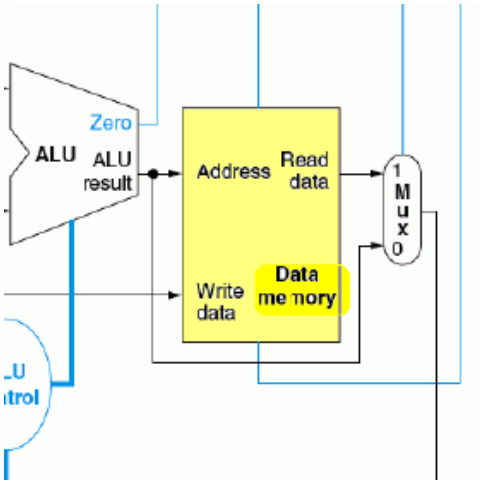
实验感想——

其实在做 register 的时候还是感受很深的。因为刚开始做的时候没有写 clock_in，因此仿真的波形最后的 readData 总是读不出来，其实是因为没有时钟的情况下，那两个寄存器根本就没有写入预期数据。这个错误是非常不应该的，也提醒了我要注意每一个细节，避免这种错误再次发生。

【内存单元模块 MEMORY】

模块描述——

data_memory 模块与 register 模块类似，依照实验指导书给出如下原理图，我们可以发现 data_memory 模块有一个时钟输入（clock_in），一个 32—bit 地址输入（address，既是读地址也是写地址），一个 32—bit 写数据（writeData），两个输入控制信号（memRead 和 memWrite）；有一个 32—bit 读数据的输出（readData）。因这是一个 dataFile，还需要一个 32—bit 的 memFile。



实验过程——

（1） 在 data_memory 模块中实现在任何时候都读取 data 和在时钟下降沿写入 data，这就需要用到 always@语句。

（2） data_memory 代码截图如下：

```
module data_memory(
    input clock_in,
    input [31:0] writeData,
    input [31:0]address,
    input memRead,
    input memWrite,
    output reg[31:0] readData
    );

    reg [31:0] memFile[0:127];

always @(address or readData or memRead or memFile)
begin
   if(memRead==1)
      readData=memFile[address];
end

always @(negedge clock_in)
begin
   if(memWrite==1)
      memFile[address]=writeData;
end


endmodule
```

（3） test_for_datamem 代码截图如下：

```
initial begin
    // Initialize Inputs
    clock_in = 0;
    writeData = 0;
    address = 0;
    memRead = 0;
    memWrite = 0;

    // Wait 100 ns for global reset to finish
    #185;
    memWrite = 1'b1;
    address = 32'b00000000000000000000000000001111;
    writeData = 32'b11111111111111110000000000000000;
    memRead=0;
    #250;
    memRead = 1'b1;
    memWrite = 1'b0;

    // Add stimulus here

end
always
    begin
        #100;
        clock_in = ~clock_in;
    end;
```

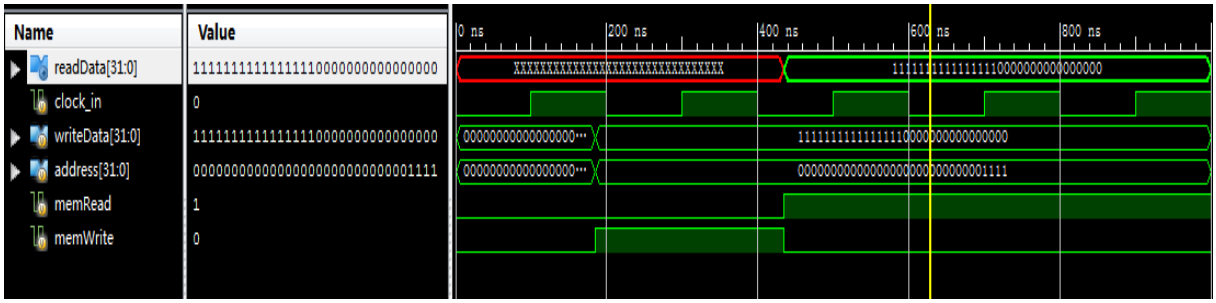
测试代码解释：

先把所有的输入数据和信号置为 0。

在 185ns 后，把数据 32’b11111111111111110000000000000000 写入 memFile 的 32’b00000000000000000000000000001111 处。

在 250ns 后，把 address（地址不变）数据读出来。

（4）在 simulation 的 hierarchy 窗口中选中 test_for_datamem 文件，并双击 Simulate Behavioral Model，得到仿真波形如下截图：



实验感想——

dataMem 模块和前面做的 register 模块非常相似，而且更为简单，按照指导书的步骤一步步来就可以做出正确的实验结果了。

【带符号扩展单元模块 SIGN_EXTENSION】

模块描述——

模块的目的是将 16 位的有符号数扩展为 32 位的有符号数。即若 16 位有符号数的最高位是 1，即 16'b1**********，就在它前面扩 16 个 1；若 16 位有符号数的最高位是 0，即 16'b0**********，就在它前面扩 16 个 0。

实验过程——

（1） signext.v 代码截图如下:

```
module signext(
    input [15:0] inst,
    output [31:0] data
    );
    assign data={16'b1111111111111111*inst[15],inst};

endmodule
```

(2)test_for_sgnext 代码截图如下：

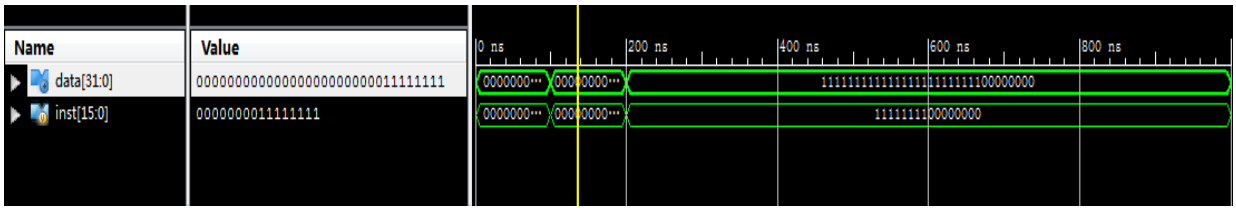
```
   initial begin
      // Initialize Inputs
      inst = 0;

      // Wait 100 ns for global reset to finish
      #100;
       inst=16'b0000000011111111;
       #100;
       inst=16'b1111111100000000;
      // Add stimulus here

   end
```

即测试输入数据一个为正数，一个为负数。

（3）在 simulation 的 hierarchy 窗口中选中 test_for_signext 文件，并双击 Simulate Behavioral Model，得到仿真波形如下截图：



实验感想——

总的来说，实验四比实验三要难一些，主要是要把 regFile 和 datamem 的信号数据过程弄清楚，才能把实验做好，加深了我对计组基础知识的认识和巩固。是一次愉快的实验经历。