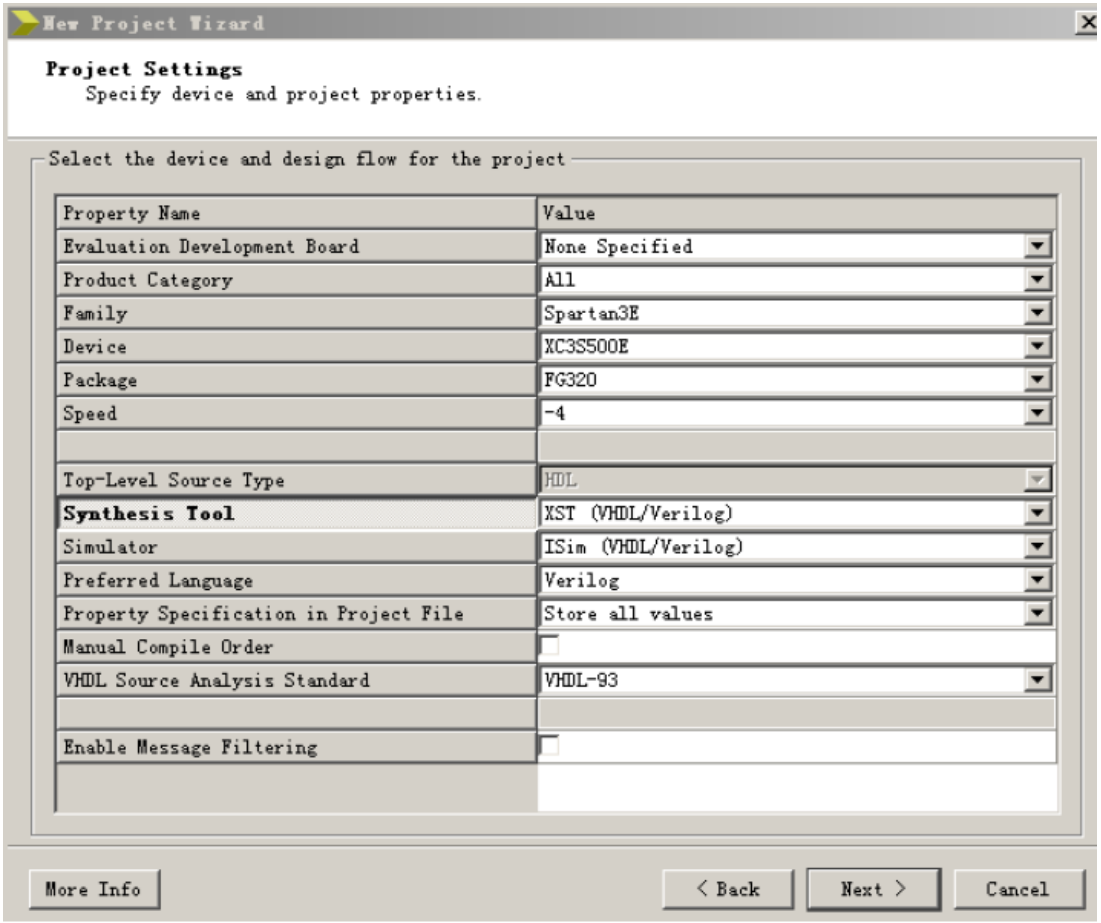
实验五实验报告

姓名：申继媛
学号：5130309194

【实验环境】



【仿真】

实验描述——

实验三和实验四都是把各个模块单独实现为.v 文件，在实验五中需要把这些单独实现的模块连接起来，实现一个完整的单周期处理器。单周期处理器的设计，关键是确定数据通路（信号和数据）以及确定那些操作需要时钟，哪些操作不需要时钟，要分析时序并约束。

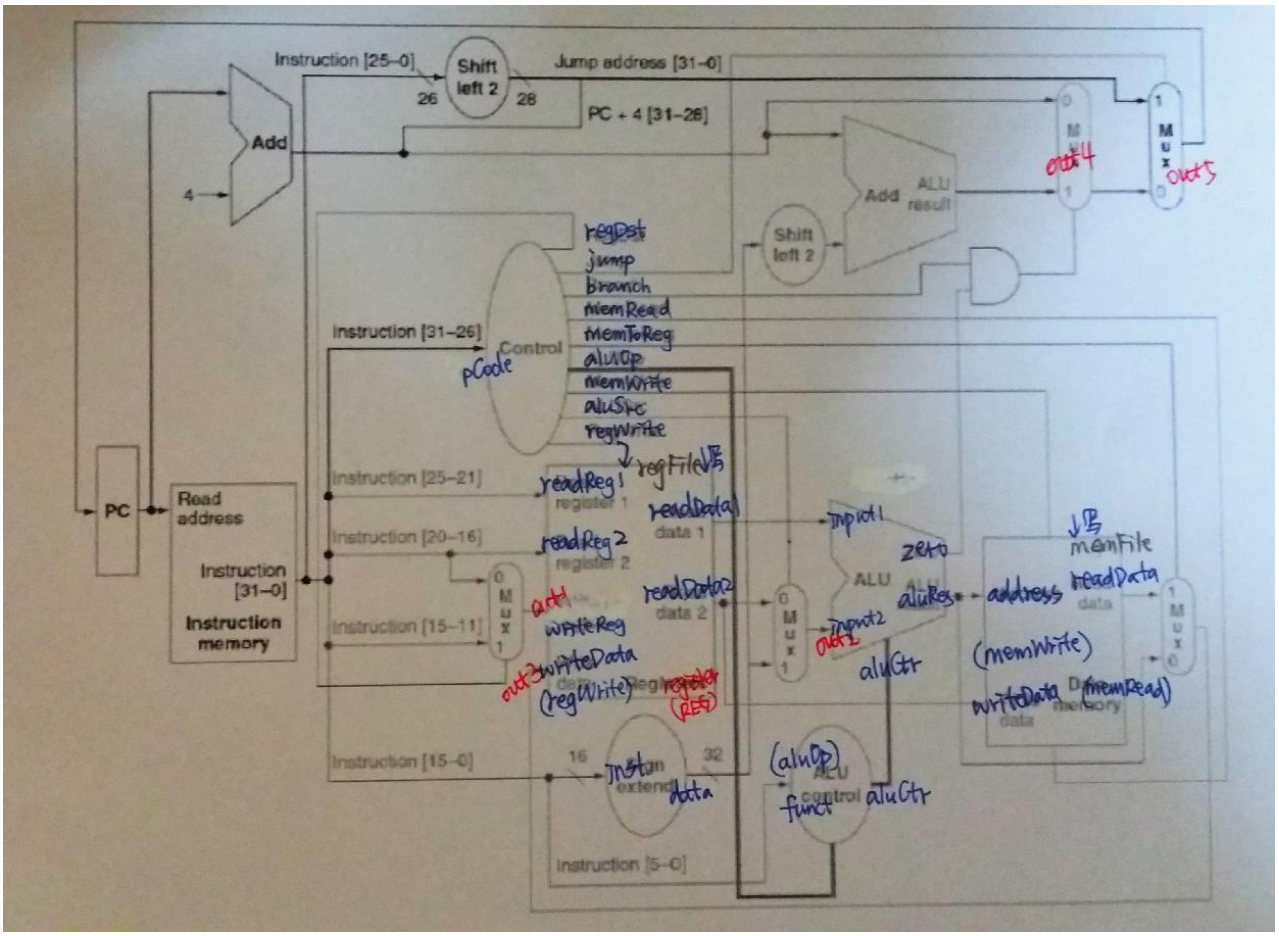
实验准备——

在 ISE14.7 中创建一个新的 project，命名为 lab5。按照实验指导书上，右键点击 Hierarchy 窗口，点击 Add Copy of Source，将实验三和实验四共 6 个.v 文件拷贝到 lab5 中。

实验过程——

（1） 新建模块源文件 top，在 top 模块内为每一根连接的信号线命名，并在 top 模块中声明它们。

1>草图：



2>代码截图：

```
module top(
    input mainClock,
    input reset,
    input [3:0] switch,
    output reg [7:0] LED
    );

    reg [31:0] PC;

    wire clk,  //reset,
         REG_DST,
         JUMP,
         BRANCH,
         MEM_READ,
         MEM_TO_REG,
         MEM_WRITE,
         ALU_SRC,
         REG_WRITE,
         ZERO;
    wire [1:0] ALU_OP;
    wire [3:0] ALU_CTR;
    wire [31:0] INST;
    wire [31:0] ALU_RES;
    wire [31:0] DATA;
```

```
wire [31:0] READ_DATA;
wire [31:0] READ_DATA1;
wire [31:0] READ_DATA2;

wire [4:0] OUT1;
wire [31:0] OUT2;
wire [31:0] OUT3;
wire [31:0] OUT4;
wire [31:0] OUT5;

wire [7:0] REG1;
wire [7:0] REG2;
wire [7:0] REG3;
wire [7:0] REG4;
```

（2）定义好 OUT1、OUT2、OUT3、OUT4、OUT5 变量。

如（1）中给出的实验草图，OUT 系列定义的变量分别是 5 个 MUX 之后的选择值。

```
assign OUT1 = REG_DST ? INST[15:11] : INST[20:16];
assign OUT2 = ALU_SRC ? DATA :READ_DATA2;
assign OUT3 = MEM_TO_REG ? READ_DATA :ALU_RES;
assign OUT4 = (ZERO && BRANCH) ? ((DATA << 2)+PC+4) : (PC +4 );
assign OUT5 = JUMP ? (PC+4) &32'b11110000000000000000000000000000|(INST[25:0] <<2) :OUT4;
```

（定义为变量的形式能让代码更加简洁，也更方便编程。）

（3）设置 PC 的值。

注意到 PC 受到 reset、branch&zero、jump 的异步控制，而它本身会同步在时钟上升沿加 4。

①首先把 PC 初始化为 0：

```
initial
begin
PC = 0;
end
```

②根据原理图的逻辑设置异步 PC 逻辑：

```
always @ (posedge clk)
begin
if(reset)
  PC<=0;
else if(ZERO & BRANCH)
  PC<=OUT4;
else if(JUMP)
  PC<=OUT5;
else
  PC <= PC+4;
end
```

（4）模块实例化，连接 top 和各个单独的模块。

按照指导书上的方法，可以将模块连接起来，代码截图如下：

```
Ctr mainCtr(
  .reset(reset),
  .opCode(INST[31:26]),
  .regDst(REG_DST),
  .jump(JUMP),
  .branch(BRANCH),
  .memRead(MEM_READ),
  .memToReg(MEM_TO_REG),
  .aluOp(ALU_OP),
  .memWrite(MEM_WRITE),
  .aluSrc(ALU_SRC),
  .regWrite(REG_WRITE));
```

```
Alu mainAlu(
   .reset(reset),
   .input1(READ_DATA1),
   .input2(OUT2),
   .aluCtr(ALU_CTR),
   .zero(ZERO),
   .aluRes(ALU_RES));

aluCtr mainAluCtr(
   .reset(reset),
   .aluOp(ALU_OP),
   .funct(INST[5:0]),
   .aluCtr(ALU_CTR));
```

```
data_memory mainDataMemory(
    .clock_in(clk),
    //.clock_in(mainClock),
    .reset(reset),
    .address(ALU_RES),
    .writeData(READ_DATA2),
    .memWrite(MEM_WRITE),
    .memRead(MEM_READ),
    .readData(READ_DATA));

register mainRegister(
    .clock_in(clk),
    //.clock_in(mainClock),
    .reset(reset),
    .readReg1(INST[25:21]),
    .readReg2(INST[20:16]),
    .writeReg(OUT1),
    .writeData(OUT3),
    .regWrite(REG_WRITE),
    .readData1(READ_DATA1),
    .readData2(READ_DATA2),
    .register1(REG1),
    .register2(REG2),
    .register3(REG3),
    .register4(REG4));
```

```
signext mainSignExt(
  .inst(INST[15:0]),
  .data(DATA),
  .reset(reset));

inst_memory mainInstMemory(
  .reset(reset),
  .clock_in(clk),
  .addr(PC),
  .inst(INST));
```

（5）用 timeDivider 分频控制

使用如下代码达到这样的目的（输入为 mainclock，而真正控制处理器的时钟为 clk，使始终得到缓冲）。

```
module timeDivider(
    input clockIn,
    output reg clockOut
    );

   reg [22:0] buffer;
   initial begin
   buffer = 0;
   clockOut = 0;
   end
   always @ (posedge clockIn)
   begin
      buffer <= buffer + 1;
      clockOut <= buffer[22];
   end

endmodule
```

将 timeDivider 连接到 top 模块中如下：

```
timeDivider mainTimeDivider(
  .clockIn(mainClock),.clockOut(clk));
```

（6）仿真测试：

1>初始化 inst_memory , register , data_memory：

因测试需要指 instruction，则 inst_memory 需要初始化；涉及的指令比如加减等是对

register 内容进行更改，则需要对 register 初始化；涉及的指令如 lw 和 sw 是对 data_memory 进行更改，则需要对 data_memory 初始化。

①对于 register。定义 regFile 中有 16 个 register（regFile[0]~regFile[15]），分别赋初始值为 0 到 15，如下：

```
reg [31:0] regFile[15:0];
initial begin
  regFile[0] = 32'b00000000000000000000000000000000;
  regFile[1] = 32'b00000000000000000000000000000001;
  regFile[2] = 32'b00000000000000000000000000000010;
  regFile[3] = 32'b00000000000000000000000000000011;
  regFile[4] = 32'b00000000000000000000000000000100;
  regFile[5] = 32'b00000000000000000000000000000101;
  regFile[6] = 32'b00000000000000000000000000000110;
  regFile[7] = 32'b00000000000000000000000000000111;
  regFile[8] = 32'b00000000000000000000000000001000;
  regFile[9] = 32'b00000000000000000000000000001001;
  regFile[10] = 32'b00000000000000000000000000001010;
  regFile[11] = 32'b00000000000000000000000000001011;
  regFile[12] = 32'b00000000000000000000000000001100;
  regFile[13] = 32'b00000000000000000000000000001101;
  regFile[14] = 32'b00000000000000000000000000001110;
  regFile[15] = 32'b00000000000000000000000000001111;
end
```

在测试过程中，测试中用到的 register 的值会发生变化，但当 reset 发生时所有的 register 又恢复初始值，如下：

```
always @ (negedge clock_in)
begin
     if (reset)
     begin
     regFile[0] <= 32'b00000000000000000000000000000000;
     regFile[1] <= 32'b00000000000000000000000000000001;
     regFile[2] <= 32'b00000000000000000000000000000010;
     regFile[3] <= 32'b00000000000000000000000000000011;
     regFile[4] <= 32'b00000000000000000000000000000100;
     regFile[5] <= 32'b00000000000000000000000000000101;
     regFile[6] <= 32'b00000000000000000000000000000110;
     regFile[7] <= 32'b00000000000000000000000000000111;
     regFile[8] <= 32'b00000000000000000000000000001000;
     regFile[9] <= 32'b00000000000000000000000000001001;
     regFile[10] <= 32'b00000000000000000000000000001010;
     regFile[11] <= 32'b00000000000000000000000000001011;
     regFile[12] <= 32'b00000000000000000000000000001100;
     regFile[13] <= 32'b00000000000000000000000000001101;
     regFile[14] <= 32'b00000000000000000000000000001110;
     regFile[15] <= 32'b00000000000000000000000000001111;
     end
     else if (regWrite) regFile[writeReg]<=writeData;
end
```

②对于 inst_memory。定义在 INSTMemFile 中有 16 个 registers（INSTMemFile[0]~INSTMemFile[15]），分别写入想要测试的语句指令，如下：

```
reg [31:0] INSTMemFile [0:15];
initial
begin
     INSTMemFile[0] = 32'b00001000000000000000000000000100; // j a
     INSTMemFile[1] = 32'b00000000000000000000000000000000;
     INSTMemFile[2] = 32'b00000000000000000000000000000000;
     INSTMemFile[3] = 32'b00000000000000000000000000000000;
     INSTMemFile[4] = 32'b101011_00000_00100_00000_00000_000000; // sw $4 0($0)
     INSTMemFile[5] = 32'b101011_00000_00011_00000_00000_000100; // sw $3 4($0)
     INSTMemFile[6] = 32'b100011_00000_00001_00000_00000_000100; // lw $1 4($0)
     INSTMemFile[7] = 32'b100011_00000_00010_00000_00000_000000; // lw $2 0($0)
     INSTMemFile[8] = 32'b00000000001000100001100000100000; // add $3,$1,$2
     INSTMemFile[9] = 32'b00000000001000100010000000100010; // sub $4,$1,$2
     //INSTMemFile[6] = 32'b00000000000000000000000000000000;
     //INSTMemFile[7] = 32'b00000000000000000000000000000000;
     //INSTMemFile[8] = 32'b00000000000000000000000000000000;
     //INSTMemFile[9] = 32'b00000000000000000000000000000000;
     INSTMemFile[10] = 32'b00001000000000000000000000000100; // j a
     INSTMemFile[11] = 32'b00000000000000000000000000000000;
     INSTMemFile[12] = 32'b00000000000000000000000000000000;
     INSTMemFile[13] = 32'b00000000000000000000000000000000;
     INSTMemFile[14] = 32'b00000000000000000000000000000000;
     INSTMemFile[15] = 32'b00000000000000000000000000000000;
end
```

指令解释：INSTMemFile[0]是一个 jump 指令，执行之后会跳到 INSTMemFile[4]执行，而在 INSTMemFile[1]到 INSTMemFile[3]中即使有指令也不会执行，这里我的测试代码中没有写指令。

INSTMemFile[4]是把 register$4 中值存到 data_mem$0 中，即 data_mem$0 由 0 变成 4 等等。

INSTMemFile[5]是把 register$3 中值存到 data_mem$1 中，即 data_mem$1 由 1 变成 3 等等。

INSTMemFile[6]是从 data_mem$1 中把值 load 到 register$1 中，即 register$1 由 1 变成 3 等等。

INSTMemFile[7]是从 data_mem$0 中把值 load 到 register$2 中，即 register$2 由 2 变成 4 等等。

（以上四个指令完成将 reg$3 的值放到 reg$1，将 reg$4 的值放到 reg$2）

INSTMemFile[8]是将 register$1 的值加上 register$2 中的值，将差结果放到 register$3 中。

INSTMemFile[9]是将 register$1 的值减去 register$2 中的值，将和结果放到 register$4 中。

INSTMemFile[10]是一个 jump 指令，执行之后会跳到 INSTMemFile[4]执行。（相当于一个循环过程）

仿真预测： 根据指令分析以及 register 和 data_mem 的初始值，若指令运行成功，应该出现如下序列：

register$1: 1—3—7—15—14—30……

register$2: 2—4—-1—8—-1—16……

register$3: 3—7—6—15—14—30……

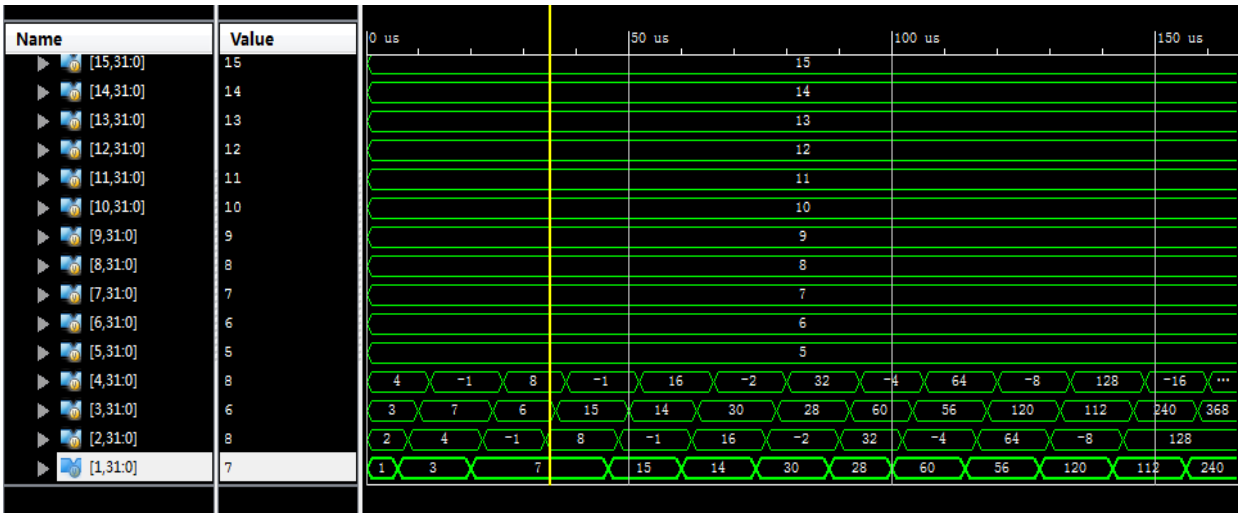
register$4: 4—-1—8—-1—16—-2……

③对于 data_memory。对于 memFile[0]到 memFile[15]分别赋初始值 0 到 15。

```
initial
begin
  memFile[0] = 32'b00000000000000000000000000000000;
  memFile[1] = 32'b00000000000000000000000000000001;
  memFile[2] = 32'b00000000000000000000000000000010;
  memFile[3] = 32'b00000000000000000000000000000011;
  memFile[4] = 32'b00000000000000000000000000000100;
  memFile[5] = 32'b00000000000000000000000000000101;
  memFile[6] = 32'b00000000000000000000000000000110;
  memFile[7] = 32'b00000000000000000000000000000111;
  memFile[8] = 32'b00000000000000000000000000001000;
  memFile[9] = 32'b00000000000000000000000000001001;
  memFile[10] = 32'b00000000000000000000000000001010;
  memFile[11] = 32'b00000000000000000000000000001011;
  memFile[12] = 32'b00000000000000000000000000001100;
  memFile[13] = 32'b00000000000000000000000000001101;
  memFile[14] = 32'b00000000000000000000000000001110;
  memFile[15] = 32'b00000000000000000000000000001111;

  //$readmemh(",/src/mem_data.txt",memFile,10'h0);
end
```

2>在 simulation 的 hierarchy 窗口中选中 test_for_Top 文件，并双击 Simulate Behavioral Model，得到仿真波形，并将 regFile 拖到仿真波形区域，单击 re—launch 如下截图：



观察所得仿真波形中 register 的值，与上一步骤预测的出现顺序相同，故仿真成功。

【上板】

实验描述——

将 PC 的[3:0]显示到 LED 的[3:0]，将 register$1 的[7:4]显示到 LED 的[7:4]。有 reset 开关控制。其中 reset 对应 N17。

实验过程——

（1）新设置一个 LED 输出 `output reg [7:0] LED`

（2）将 PC 的[3:0]显示到 LED 的[3:0]，将 register$1 的[7:4]显示到 LED 的[7:4]。

```
always begin
LED[3:0]=PC[3:0];
LED[7:4]=REG1[7:4];
end
```

（3）本实验在上板后没有更改仿真的测试 instructions、register、data_mem，故上板

的 LED 灯显示结果应该和仿真波形的 register 值相同。

（4）创建 ucf 文件，配置好 XC3S500E 芯片与程序对应的输入和显示接口，如下：

```
NET "LED[5]" LOC = D11 | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "LED[4]" LOC = C11 | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "LED[7]" LOC = F9 | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "LED[3]" LOC = F11 | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "LED[1]" LOC = E12 | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "LED[6]" LOC = E9 | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "LED[2]" LOC = E11 | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "LED[0]" LOC = F12 | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;

# PlanAhead Generated physical constraints

#NET "switch[2]" LOC = H18 | IOSTANDARD = LVTTL | PULLUP;
#NET "switch[1]" LOC = L14 | IOSTANDARD = LVTTL | PULLUP;
#NET "switch[3]" LOC = N17 | IOSTANDARD = LVTTL | PULLUP;
#NET "switch[0]" LOC = L13 | IOSTANDARD = LVTTL | PULLUP;
NET "mainClock" LOC = C9 | IOSTANDARD = LVCMOS33;
NET "reset" LOC=N17 | IOSTANDARD = LVTTL | PULLUP;
```

（5）单击 top.v 文件之后，在 process 窗口中双击 Generate Programming File，再双击 Manage Configuration Project（iMPACT），等待一段时间后，在弹出来的 ISE iMPACT 窗口中，双击 Boundary Scan，在 iMPACT 右侧部分右键选择 Initialize Chain，在弹出的对话框中选择 yes，然后 open 刚刚生成的.bit 文件，再选择 no—bypass—bypass—ok。右键选中待烧写.bit 的在用 FPGA 芯片，运行 program。若出现 Program Succeeded，说明烧写成功。此时对芯片的 switch 进行控制，并且观察 LED 显示情况是否与仿真波形情况一致。经实验，已测得预想结果，实验成功。

实验感想——

实验五明显比前面的实验要难一些，而且指导书上的说明也没有把所有的细节都覆盖。因此做实验五的时候在宿舍写代码写了很长时间。另外这是第一次上板，刚开始调节的时候结果总是很奇怪，有时候 LED 都不会亮，最后尝试着改了 reset 和频率，就得到了预期的结果，这让我发现频率真的很重要。做完实验五之后有一种满足感，是进一步完成实验六的动力。