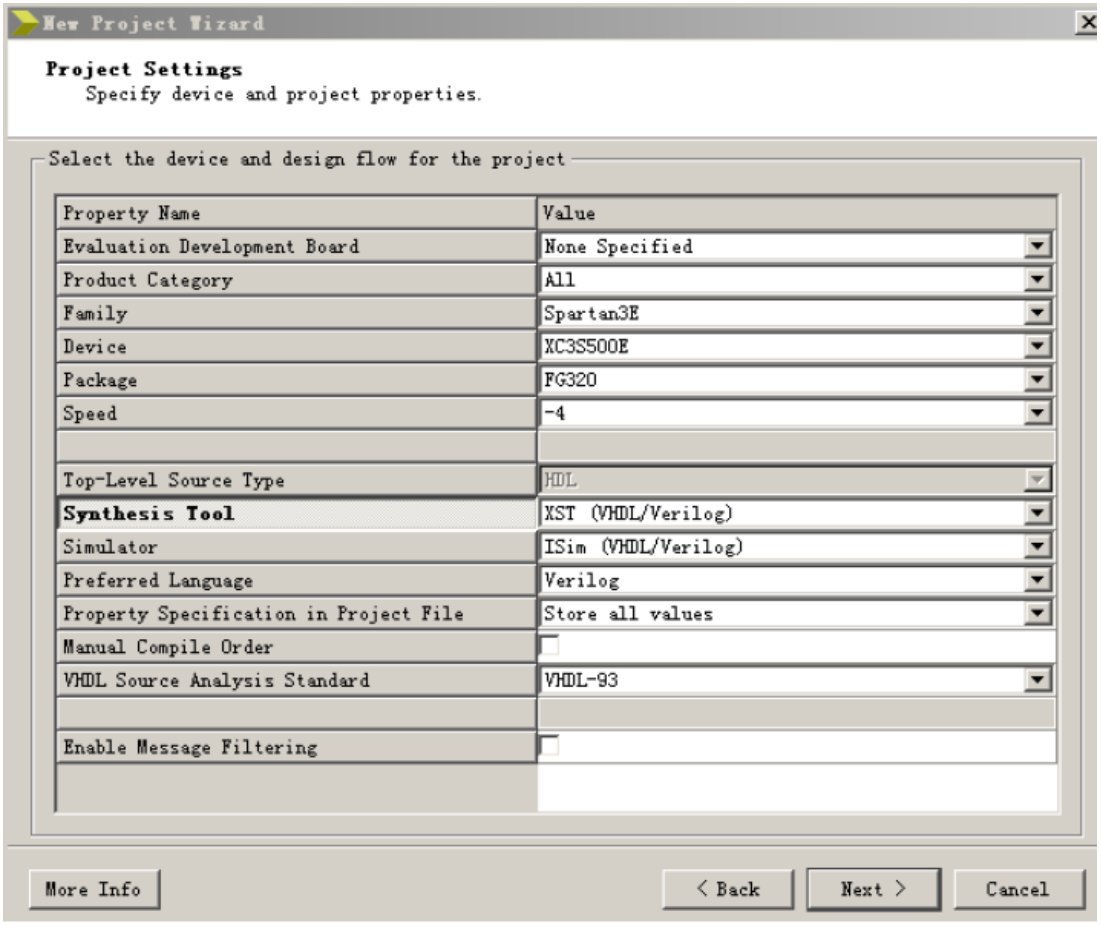
实验六实验报告

姓名：申继媛
学号：5130309194

【实验环境】



【仿真】

实验描述——

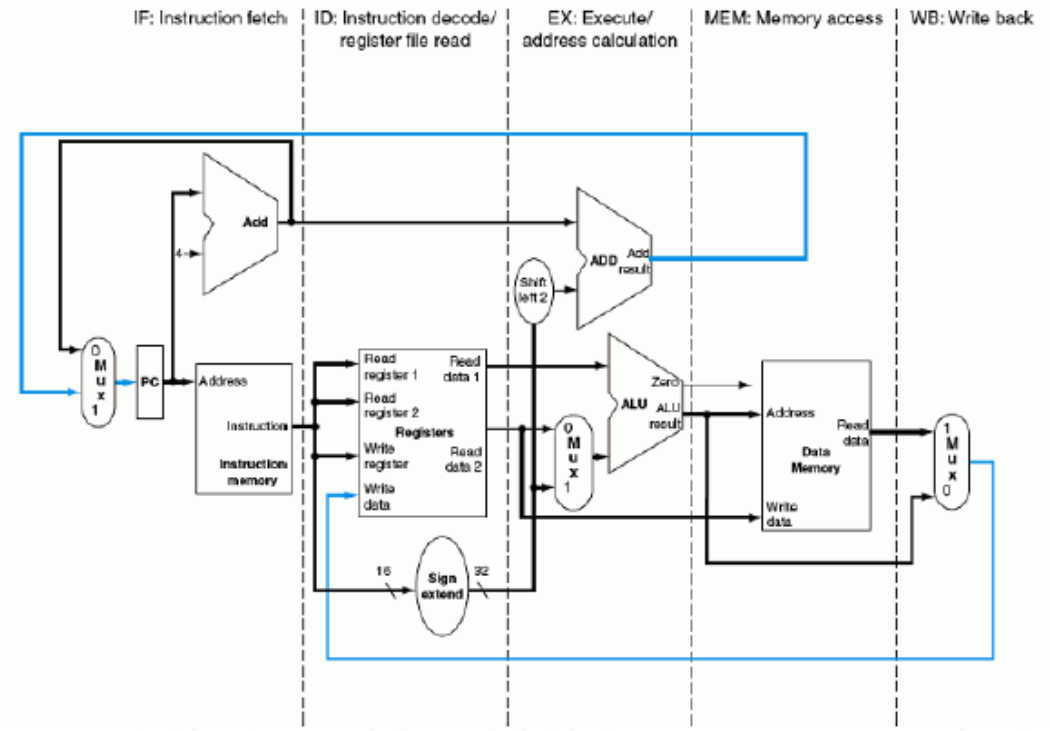
实验三和实验四都是把各个模块单独实现为.v 文件，在实验五十实现一个完整的单周期处理器。而在实验六中，需要实现类 MIPS 多周期流水线处理器。

实验准备——

在 ISE14.7 中创建一个新的 project，命名为 lab6。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 lab6 中。（需要把 mem 的.v 文件分成 datamem 和 instmem 两种）

实验过程——

（1） 观察到流水线的主要结构，有四级寄存器（IF/ID , ID/EX , EX/MEM , MEM/WB）。这些寄存器用来把控制信号一级级保存下来，以供后续每级流水使用。



（2）新建模块源文件 top，在 top 模块内为每一级寄存器定义好它们各自应该包含的信号和变量，参照实验指导书上给出的部分代码，代码截图如下：

```
//1.0 for stage IF to ID
reg [31:0] IF_ID_PcAdd4;
reg [31:0] IF_ID_Instruction;

//2.0 for stage ID to EX
reg [31:0] ID_EX_PcAdd4;
reg [31:0] ID_EX_ReadData1;
reg [31:0] ID_EX_ReadData2;
reg [31:0] ID_EX_SignExt;
reg [20:16] ID_EX_InstHigh;
reg [15:11] ID_EX_InstLow;
//2.1 to EX
reg ID_EX_RegDst;
reg [1:0] ID_EX_ALUOp;
reg ID_EX_ALUSrc;
//2.2 to MEM;
reg ID_EX_Branch;
reg ID_EX_MemRead;
reg ID_EX_MemWrite;
//2.3 to WB
reg ID_EX_MemToReg;
reg ID_EX_RegWrite;
```

```
//3.0 for stage EX to MEM
reg [31:0] EX_MEM_PcNew;
reg EX_MEM_Zero;
reg [31:0] EX_MEM_ALUOut;
reg [31:0] EX_MEM_ReadData2;
reg [4:0] EX_MEM_WriteReg;
//3.1 to MEM;
reg EX_MEM_Branch;
reg EX_MEM_MemRead;
reg EX_MEM_MemWrite;
//3.2 to WB
reg EX_MEM_MemToReg;
reg EX_MEM_RegWrite;

//4.0 for stage MEM to WB
reg [31:0] MEM_WB_ALUOut;
reg [31:0] MEM_WB_ReadData;
reg [4:0] MEM_WB_WriteReg;
//4.1 to WB
reg MEM_WB_MemToReg;
reg MEM_WB_RegWrite;
```

（3）定义变量、输入、输出，并初始化 PC 的地址。

（定义变量）

```
module Top(
    input CLOCK_IN,
    input RESET
    );

    //for io
    wire CLK;
    assign CLK=CLOCK_IN;

    //Regs
    //PC
    reg [31:0] PC;
```

```
//wire
 wire REG_DST,
    JUMP,
    BRANCH,
    MEM_READ,
    MEM_TO_REG,
    MEM_WRITE;
wire [1:0] ALU_OP;
wire ALU_SRC,
    REG_WRITE;
wire [31:0] REG_READ1,REG_READ2,SGE_OUT;
wire ZERO;
wire [31:0] ALU_OUT,MEM_DATA,READ_ADD,INST;
wire [3:0] ALU_CTR;

assign READ_ADD=PC;
```

（初始化 PC）

```
initial
begin
PC<=32'b0;
end
```

（4）将所有定义的变量与每个独立的模块连接起来：

```
Ctr mainCtr(
        .opCode(IF_ID_Instruction[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));
signext se(
        .inst(IF_ID_Instruction[15:0]),
        .data(SGE_OUT));
AluCtr aluCtr(
        .aluOp(ID_EX_ALUOp),
        .funct(ID_EX_SignExt[5:0]),
        .aluCtr(ALU_CTR));
Alu alu(
        .input1(ID_EX_ReadData1),
        .input2(ID_EX_ALUSrc? ID_EX_SignExt:ID_EX_ReadData2),
        .zero(ZERO),
        .aluRes(ALU_OUT),
        .aluCtr(ALU_CTR));
```

```
register regs(
        .clock_in(CLK),
        .res(RESET),
        .readReg1(IF_ID_Instruction[25:21]),
        .readReg2(IF_ID_Instruction[20:16]),
        .writeReg(MEM_WB_WriteReg),
        .writeData(MEM_WB_MemToReg? MEM_WB_ReadData:MEM_WB_ALUOut),
        .regWrite(MEM_WB_RegWrite),
        .readData1(REG_READ1),
        .readData2(REG_READ2));
data_memory dm(
        .clock_in(CLK),
        .writeData(EX_MEM_ReadData2),
        .address(EX_MEM_ALUOut),
        .memRead(EX_MEM_MemRead),
        .memWrite(EX_MEM_MemWrite),
        .readData(MEM_DATA));
inst_memory im(
        .address(READ_ADD/4),
        .readData(INST));
```

（5）操作实现：在时钟上升沿时 reset，即把所有变量置零。

```
always@(posedge CLK)
begin
   if (RESET == 1)
   begin
      PC<=32'b0;

      IF_ID_PcAdd4 <= 0;
      IF_ID_Instruction <= 0;

      ID_EX_PcAdd4 <= 0;
      ID_EX_ReadData1 <= 0;
      ID_EX_ReadData2 <= 0;
      ID_EX_SignExt <= 0;
      ID_EX_InstHigh <= 0;
      ID_EX_InstLow <= 0;
      ID_EX_RegDst <= 0;
      ID_EX_ALUOp <= 0;
      ID_EX_ALUSrc <= 0;
      ID_EX_Branch <= 0;
      ID_EX_MemWrite <= 0;
      ID_EX_MemRead <= 0;
      ID_EX_RegWrite <= 0;
      ID_EX_MemToReg <= 0;

      EX_MEM_PcNew <= 0;
      EX_MEM_Zero <= 0;
      EX_MEM_ALUOut <= 0;
      EX_MEM_ReadData2 <= 0;
      EX_MEM_WriteReg <= 0;
      EX_MEM_Branch <= 0;
      EX_MEM_MemWrite <= 0;
      EX_MEM_MemRead <= 0;
      EX_MEM_RegWrite <= 0;
      EX_MEM_MemToReg <= 0;

      MEM_WB_ALUOut <= 0;
      MEM_WB_ReadData <= 0;
      MEM_WB_WriteReg <= 0;
      MEM_WB_RegWrite <= 0;
      MEM_WB_MemToReg <= 0;
   end
```

（6）操作实现：在时钟上升沿的 always@中，实现 general operations 状态的值。

```
else
begin
    IF_ID_PcAdd4 <= READ_ADD+4;
    IF_ID_Instruction <= INST;

    ID_EX_PcAdd4 <= IF_ID_PcAdd4;
    ID_EX_ReadData1 <= REG_READ1;
    ID_EX_ReadData2 <= REG_READ2;
    ID_EX_SignExt <= SGE_OUT;
    ID_EX_InstHigh <= IF_ID_Instruction[20:16];
    ID_EX_InstLow <= IF_ID_Instruction[15:11];
    ID_EX_RegDst <= REG_DST;
    ID_EX_ALUOp <= ALU_OP;
    ID_EX_ALUSrc <= ALU_SRC;
    ID_EX_Branch <= BRANCH;
    ID_EX_MemWrite <= MEM_WRITE;
    ID_EX_MemRead <= MEM_READ;
    ID_EX_RegWrite <= REG_WRITE;
    ID_EX_MemToReg <= MEM_TO_REG;

    EX_MEM_PcNew <= ID_EX_PcAdd4 + (ID_EX_SignExt << 2);
    EX_MEM_Zero <= ZERO;
    EX_MEM_ALUOut <= ALU_OUT;
    EX_MEM_ReadData2 <= ID_EX_ReadData2;
    EX_MEM_WriteReg <= ID_EX_RegDst ? ID_EX_InstLow : ID_EX_InstHigh;
    EX_MEM_Branch <= ID_EX_Branch;
    EX_MEM_MemWrite <= ID_EX_MemWrite;
    EX_MEM_MemRead <= ID_EX_MemRead;
    EX_MEM_RegWrite <= ID_EX_RegWrite;
    EX_MEM_MemToReg <= ID_EX_MemToReg;

    MEM_WB_ALUOut <= EX_MEM_ALUOut;
    MEM_WB_ReadData <= MEM_DATA;
    MEM_WB_WriteReg <= EX_MEM_WriteReg;
    MEM_WB_RegWrite <= EX_MEM_RegWrite;
    MEM_WB_MemToReg <= EX_MEM_MemToReg;

    PC <= (EX_MEM_Zero & EX_MEM_Branch) ?  EX_MEM_PcNew : (PC+4);
```

其中，当 branch 发生时，亦即(EX_MEM_Zero & EX_MEM_Branch)==TRUE 时，需要把一些信号和数据置零如下：

```
if (EX_MEM_Zero & EX_MEM_Branch)
begin
IF_ID_PcAdd4 <= 0;
IF_ID_Instruction <= 0;

ID_EX_PcAdd4 <= 0;
ID_EX_ReadData1 <= 0;
ID_EX_ReadData2 <= 0;
ID_EX_SignExt <= 0;
ID_EX_InstHigh <= 0;
ID_EX_InstLow <= 0;                EX_MEM_PcNew <= 0;
ID_EX_RegDst <= 0;                 EX_MEM_Zero <= 0;
ID_EX_ALUOp <= 0;                  EX_MEM_ALUOut <= 0;
ID_EX_ALUSrc <= 0;                 EX_MEM_ReadData2 <= 0;
ID_EX_Branch <= 0;                 EX_MEM_WriteReg <= 0;
ID_EX_MemWrite <= 0;               EX_MEM_Branch <= 0;
ID_EX_MemRead <= 0;                EX_MEM_MemWrite <= 0;
ID_EX_RegWrite <= 0;               EX_MEM_MemRead <= 0;
ID_EX_MemToReg <= 0;               EX_MEM_RegWrite <= 0;
                                   EX_MEM_MemToReg <= 0;          end
```

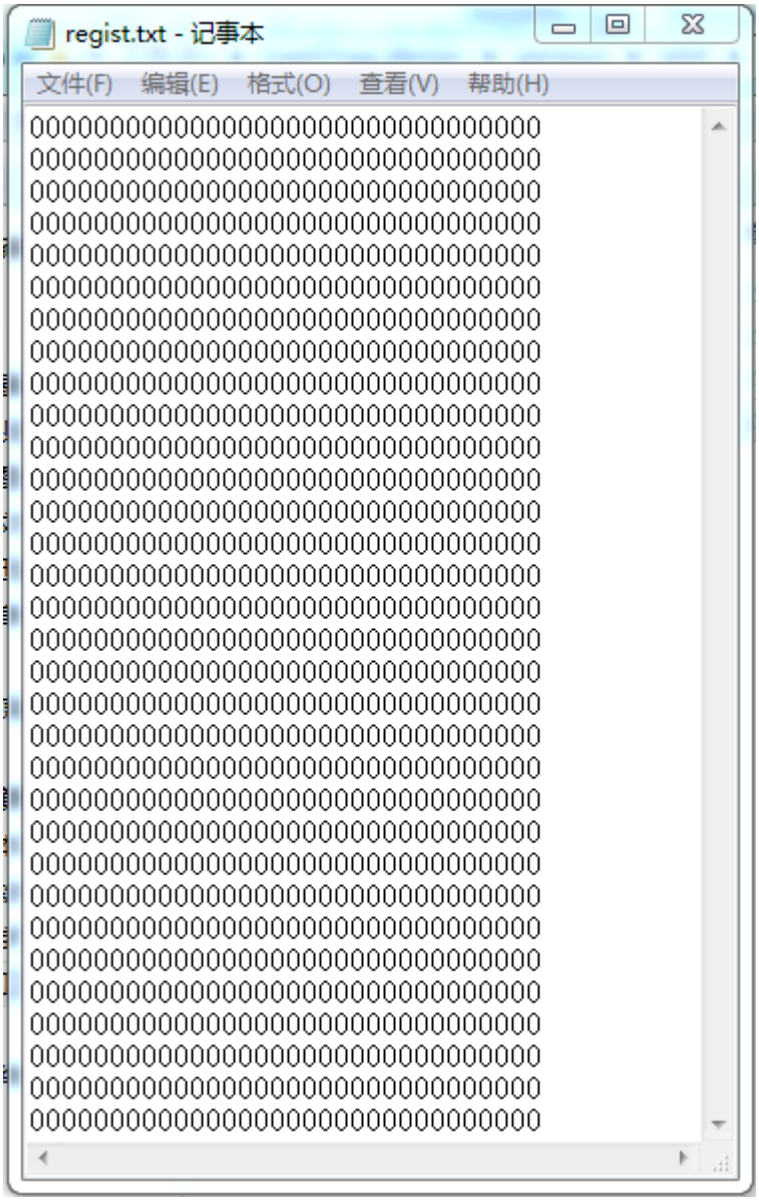
（7）仿真测试：

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。

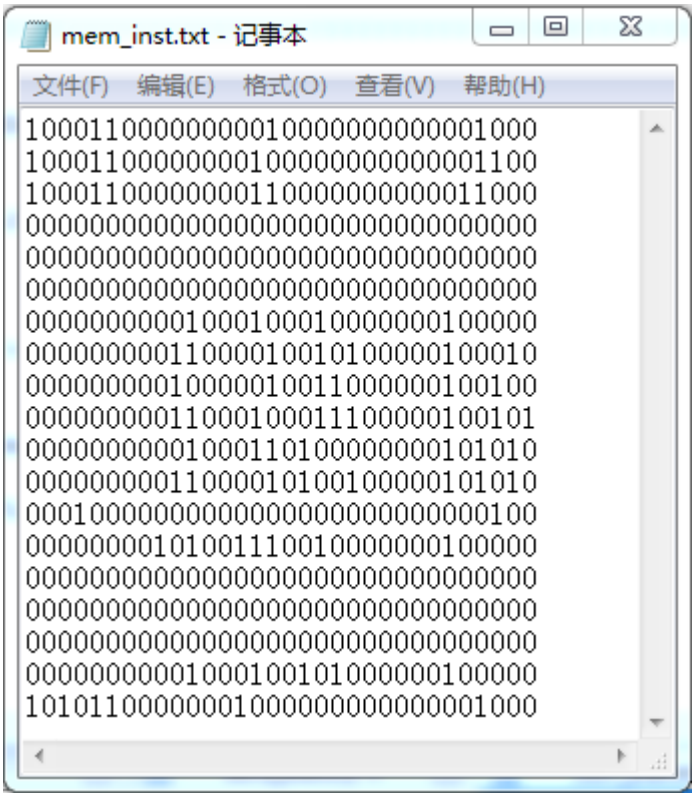
①对于 register。定义 regFile 中有 32 个 register（regFile[0]~regFile[31]），均赋初始值为 0，（我在实验五和实验六中用到的赋值方法是不同的，实验五是直接写到.v 文件里面，而实验六是写到一个文本文件里面，再有.v 文件读进来）如下：

```
initial
begin
    $readmemh("./Src/regist.txt",regFile,32'h0);
end
```



②对于 inst_memory 模块。定义在 INSTMemFile 中有 memFile，分别写入想要测试的语句指令，并读进来如下：

```
initial
begin
    $readmemb("./Src/mem_inst.txt",memFile,8'h0);
end
```

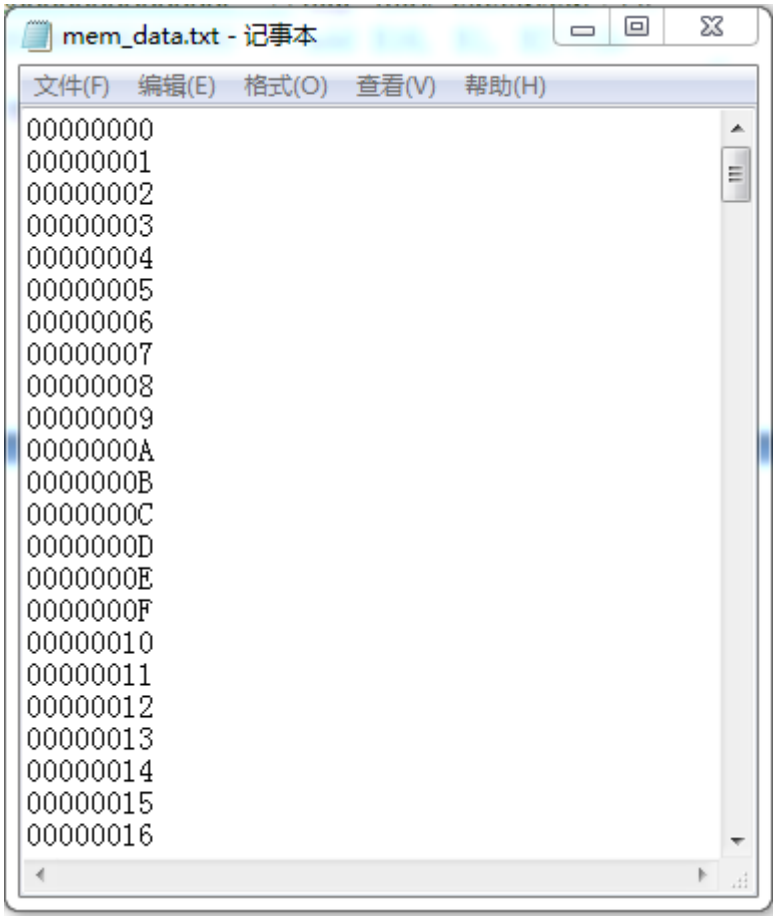


指令解释：在./Src 文件夹目录中有一个 word 文档是对所写测试指令的解释，截图如下：

```
\verb|10001100000000010000000000001000  // lw $1, 8($0)|\\2
\verb|10001100000000010000000000001100  // lw $2, 12($0)|\\3
\verb|10001100000000011000000000011000  // lw $3, 24($0)|\\6
\verb|00000000000000000000000000000000  //nop (data hazard!)|\\
\verb|00000000000000000000000000000000  //nop|\\
\verb|00000000000000000000000000000000  //nop|\\
\verb|00000000001000100010000000100000  //add $4, $1, $2|\\5
\verb|00000000011000010010100000100010  //sub $5, $3, $1|\\4
\verb|00000000010000010011000000100100  //and $6, $2, $1|\\2
\verb|00000000011000100011100000100101  //or $7, $3, $2|\\7
\verb|00000000001000110100000000101010  //slt $8, $1, $3|\\1
\verb|00000000011000010100100000101010  //slt $9, $3, $1|\\1
\verb|00010000000000000000000000000100  //beq $0, $0, j4 (branch
hazard!)|\\
\verb|00000000101001110010000000100000  //add $4, $5, $7 (not
executed)|\\

\verb|00000000000000000000000000000000  //nop (not executed)|\\
\verb|00000000000000000000000000000000  //nop (not executed)|\\
\verb|00000000000000000000000000000000  //nop (not executed)|\\
\verb|00000000001000100101000000100000  //add $10, $1, $2|\\5
```

③对于 data_memory。因设置的 memFile（memFile[0]~memFile[255]），故可以赋初始值为 00~FF，即部分截图如下：



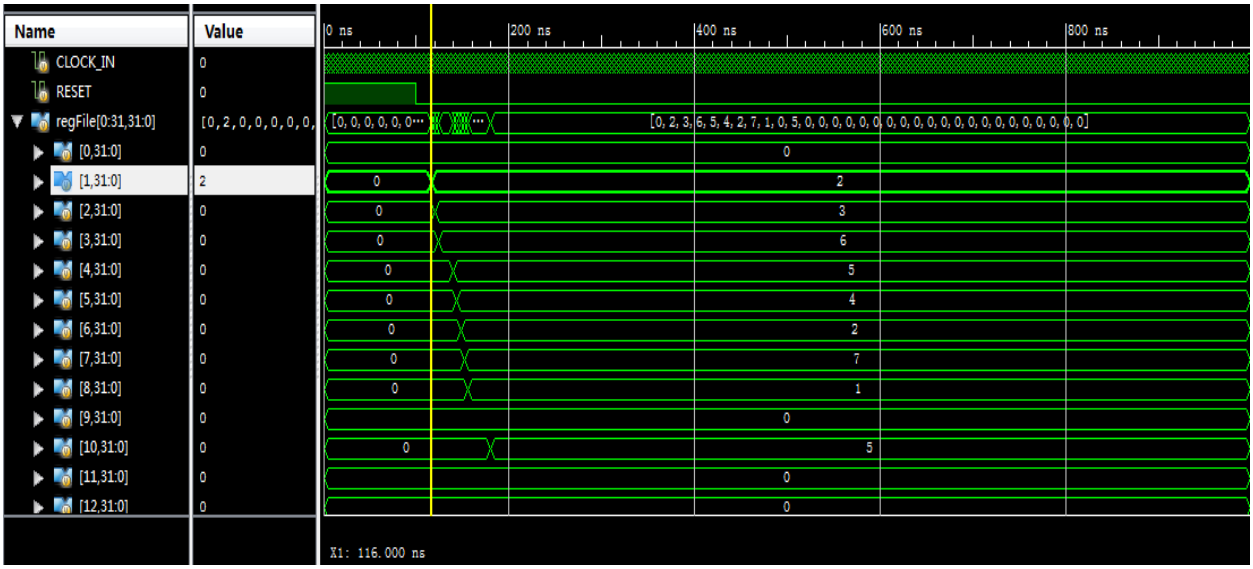
2>生成 testbench 文件 test_for_Top:

```
initial begin
   // Initialize Inputs
   CLOCK_IN = 0;
   RESET = 1;

   // Wait 100 ns for global reset to finish
   #100;
     RESET=0;
   // Add stimulus here

end
always
#2 CLOCK_IN=!CLOCK_IN;
```

3>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：



分析仿真波形：仿真波形是按照 inst_mem 中的代码执行的，图示给出了各阶段 register 的状态值。因初始化 registers 都是 0，故所有的 register 在仿真波形的开始那段都是 0。

执行第一条指令（lw $1, 8($0)）后，把 data_mem 中$2 的值（2）load 出来到 register$1 中，故 register$1 最先变化，由 0 变为 2；

执行第二条指令（lw $2, 12($0)），同第一条指令类似，是把 data_mem 中$3 的值（3）load 出来到 register$2 中，故 register$2 变化，由 0 变为 3；

执行第三条指令（lw $3, 24($0)），同第一二条指令类似，是把 data_mem 中$6 的值（6）load 出来到 register$3 中，故 register$3 变化，由 0 变为 6；

执行第四、五、六条指令（nop），即避免 data hazard，可在仿真波形上明显观察到 register$3 和 register$4 变化的时间间隔较之前的时间变化间隔长；

执行第七条指令（add $4, $1, $2），是把 register$1 的值（2）和 register$2 的值（3）做加法，将得到的和结果写入 register$4，故 register$4 变化，由 0 到（2+3=）5；

执行第八条指令（sub $5, $3, $1），是把 register$3 的值（6）减去 register$1 的值（2），将得到的差结果写入 register$5，故 register$5 变化，由 0 到（6-2=）4；

执行第九条指令（and $6, $2, $1），是把 register$2 的值（0011）和 register$1 的值（0010）按位与，将得到的结果写入 regster$6，故 register$6 变化，由 0 到（0010）2；

执行第十条指令（or $7, $3, $2），是把 register$3 的值（0110）和 register$2 的值（0011）按位或，将得到的结果写入 register$7，故 register$7 变化，由 0 到（0111）7；

执行第十一条指令（slt $8, $1, $3），因 register$1 值为 2，register$3 中值为 6，故（$1<$3）条件成立，即 register$8 会变化，由 0 到 1；

执行第十二条指令（slt $9, $3, $1），因 register$3 中值为 6，register$1 值为 2，故（$3<$1）条件不成立，即 register$9 不会变化，即 register9 会一直为 0；

执行第十三条指令（beq $0, $0, j4），因 register$0 和自己的值肯定相等，满足 branch 条件，故会跳到该指令之后的第四个指令再开始继续执行，因此第十四条 ADD 指令是不会执行的（如果执行，那么会把 register$4 变为 11，然而仿真波形中 register$4 没有变化，故说明 branch 成功）；

执行第十八条指令（add $10, $1, $2），把 register$1 的值（2）和 register$2 的值（3）相加，并把和结果写入 register$10，即 register$10 变化，由 0 到（2+3=）5。

【上板】

实验描述——

将 PC 的[3:0]显示到 LED 的[3:0]；根据 index 的选择将对应的 register 或 memFile reg 的值显示到 LED 的[7:4]。有 reset，pause，memreg，index[1:0]开关控制。其中 reset 对应 K17，pause 对应 N17，memreg 对应 H18，index[1]对应 L14，index[0]对应 L13。

实验过程——

（1)将时钟分频建一个 module 实现分频，使得上板的 LED 显示灯能够有合适的闪烁频率：

```
module timerDivider(
       clockIn,
       clockOut,
       pau
       );
       input clockIn;
       input pau;
       output clockOut;
       reg clockOut;

       reg[24:0] buffer;
       initial
       buffer=0;
       always@(posedge clockIn)
       begin
       if(pau==0)
       begin
          buffer <= buffer+1;
          clockOut <= &buffer;
       end
       end

endmodule
```

```
    wire clk;
 timerDivider td(.clockIn(mainclk), .clockOut(clk), .pau(pause));
```

（2）新设置一个暂停输入 pause；一个 PC 输出 reg[3:0]currentPC；一个 register/memory value 的输出 wire[3:0]value；一个控制 value 值的输入 memreg（若 memreg==1，则输出的 value 是 data_memory 值；若 memreg==0，则输出的 value 是 register 的值）；一个输入 index 选择查看的是哪一个 register/memory reg，即[1:0]index（可以查看 00,01,10,11 四个 register 值或 00,01,10,11 四个 data_memory 值）。

```
input mainclk,
input reset,
   input memreg,
input[1:0]index,
input pause,
output reg[3:0]currentPC,
output wire[3:0]value

assign value=(memreg)? MEM_VALUE:REG_VALUE;
```

（3）初始化 inst_mem、data_mem、register 如下：

①对于 register。（32 个 register，每个 register 有 8’bits）

```
reg [31:0] regFile[7:0];
initial
begin
     regFile[0]=32'h0;
     regFile[1]=32'h1;
     regFile[2]=32'h2;
     regFile[3]=32'h3;
     regFile[4]=32'h4;
     regFile[5]=32'h5;
     regFile[6]=32'h6;
     regFile[7]=32'h7;
end
```

②对于 data_memory。（memFile 里有 32 个 register，每个 register 有 8’bits）

```
reg [31:0] memFile[0:7];
initial
        begin
     memFile[0]=32'h0;
     memFile[1]=32'h1;
     memFile[2]=32'h2;
     memFile[3]=32'h3;
     memFile[4]=32'h4;
     memFile[5]=32'h5;
     memFile[6]=32'h6;
     memFile[7]=32'h7;
     end
```

③对于 inst_memory。在.v 文件中写下测试指令。

```
begin
     memFile[0]=32'b10101100000001000000000000001000; // sw $4, 8($0)
     memFile[1]=32'b10001100000000100000000000001100; // lw $2, 12($0):3
     memFile[2]=32'b10001100000000110000000000011000; // lw $3, 24($0):6
     memFile[3]=32'b00000000000000000000000000000000; // nop(data hazard!)
     memFile[4]=32'b00000000000000000000000000000000; //nop
     memFile[5]=32'b00000000000000000000000000000000; //nop
     memFile[6]=32'b00000000010000110000100000100000; //add $1, $2, $3:9
     memFile[7]=32'b00000000011000100000100000100010; //sub $1, $3, $2:3
     memFile[8]=32'b00000000010000110000100000100100; //and $1, $2, $3:2
     memFile[9]=32'b00000000010000110000100000100101; //or $1, $2, $3:7
     memFile[10]=32'b00000000010000110000100000101010; //slt $1, $2, $3:1
     memFile[11]=32'b00000000011000100000100000101010; //slt $1, $3, $2:0
     memFile[12]=32'b00010000001000010000000000000010; //beq $1, $1, j2 (branch hazard!)
     memFile[13]=32'b00000000010000110000100000100000; //add $1, $2, $3 (not executed):9
     memFile[14]=32'b00000000010000110000100000100000; //add $1, $2, $3 (not executed):9
     memFile[15]=32'b00000000011000100000100000100010; //sub $1, $3, $2:3
end
```

指令解释：

第一条指令（sw $4, 8($0)），把 register$4 中的值（4）存到 memFile[2]中，即 memFile[2]的值由 2 变为 4；

第二条指令（lw $2, 12($0)），把 memFile[3]的值（3）load 到 register$2 中，即 register$2 的值由 2 变为 3；

第三条指令（lw $3, 24($0)），把 memFile[6]的值（6）load 到 register[3]中，即 register$3 的值由 3 变为 6；

第四到第六条指令都是 nop 避免 data hazard；

第七条指令（add $1, $2, $3），把 register$2 的值（3）和 register$3 的值（6）相加，并

把和结果（9）写到 register$1 中，即 register$1 的值由 1 变为 9；

第八条指令（sub $1, $3, $2），把 register$3 的值（6）和 register$2 的值（3）相减，差结果（3）写到 register$1 中，即 register$1 的值由 9 变为 3；

第九条指令（and $1, $2, $3），把 reister$2 的值（3——0011）和 register$3 的值（6——0110）按位与，将结果（2——0010）写到 register$1 中，即 register$1 由 3 变为 2；

第十条指令（or $1, $2, $3），把 register$2 的值（3——0011）和 register$3 的值（6——0110）按位或，将结果（7——0111）写到 register$1 中，即 register$1 由 2 变为 7；

第十一条指令(slt $1, $2, $3)，因($2<$3)条件成立，故把 1 写到 register$1 中，即 register$1 的值由 7 变为 1；

第十二条指令（slt $1, $3, $2），因（$3<$2）条件不成立，故把 0 写到 register$1 中，即 register$1 的值由 1 变为 0；

第十三条指令（beq $1, $1, j2），因（$1==$1）条件成立，故会跳过这之后的两条指令条指令，即第十四和第十五条指令没有执行；

第十六条指令（sub $1, $3, $2），把 register$3 的值（6）减去 register$2 的值（3），将差结果（3）写到 register$1 中，即 register$1 的值由 0 变为 3。

综上讨论：在上板之后的理想情况是——

memFile[0]是 0；

memFile[1]是 1；

memFile[2]是 2——4；

memFile[3]是 3；

register$0 是 0；

register$1 是 1——9——3——2——7——1——0——3；

register$2 是 2——3；

register$3 是 3——6；

（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 "mainclk" LOC = C9;
NET "mainclk" IOSTANDARD = LVCMOS33;
NET "reset" IOSTANDARD = LVTTL;
NET "reset" PULLDOWN;
NET "pause" LOC = N17;
NET "pause" IOSTANDARD = LVTTL;
NET "pause" PULLUP;
NET "memreg" LOC = H18;
NET "memreg" IOSTANDARD = LVTTL;
NET "memreg" PULLUP;
NET "index[1]" LOC = L14;
NET "index[1]" IOSTANDARD = LVTTL;
NET "index[1]" PULLUP;
NET "index[0]" LOC = L13;
NET "index[0]" IOSTANDARD = LVTTL;
NET "index[0]" PULLUP;
NET "currentPC[3]" LOC = F9;
NET "currentPC[3]" IOSTANDARD = LVTTL;
NET "currentPC[3]" DRIVE = 8;
NET "currentPC[3]" SLEW = SLOW;
NET "currentPC[2]" LOC = E9;
NET "currentPC[2]" IOSTANDARD = LVTTL;
NET "currentPC[2]" DRIVE = 8;
NET "currentPC[2]" SLEW = SLOW;
NET "currentPC[1]" LOC = D11;
NET "currentPC[1]" IOSTANDARD = LVTTL;
NET "currentPC[1]" DRIVE = 8;
NET "currentPC[1]" SLEW = SLOW;
NET "currentPC[0]" LOC = C11;
NET "currentPC[0]" IOSTANDARD = LVTTL;
NET "currentPC[0]" DRIVE = 8;
NET "currentPC[0]" SLEW = SLOW;
NET "value[3]" LOC = F11;
NET "value[3]" IOSTANDARD = LVTTL;
NET "value[3]" DRIVE = 8;
NET "value[3]" SLEW = SLOW;
NET "value[2]" LOC = E11;
NET "value[2]" IOSTANDARD = LVTTL;
NET "value[2]" DRIVE = 8;
NET "value[2]" SLEW = SLOW;
NET "value[1]" LOC = E12;
NET "value[1]" IOSTANDARD = LVTTL;
NET "value[1]" DRIVE = 8;
NET "value[1]" SLEW = SLOW;
NET "value[0]" LOC = F12;
NET "value[0]" IOSTANDARD = LVTTL;
NET "value[0]" DRIVE = 8;
NET "value[0]" SLEW = SLOW;

# PlanAhead Generated physical constraints

NET "reset" LOC = K17;
```

（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（3）中的预想的理想情况一致。经实验，已测得预想结果，实验成功。

实验感想——

依托在实验五的基础上，对整体设计有了初步的认识，在面对实验六的流水线设计要求时，感觉大致有了一个实验方向。在上板的时候有了实验五的注意，减少了不必要的错误。给老师检查的时候，我只在上板时写了一个 register$3 或 data_mem$3 的状态显示，回宿舍后我加了一个 index[1:0],从而能显示 register$0~register$3 或者 data_mem$0~data_mem$3 的状态。在所有的计组实验结束之后，看着所有的 lab projects，在宿舍时花了很多时间在计组实验上，实在不懂的地方和同学进行交流。这是一次很愉快的实验经历，让我对以后的实验更加期待。最后，在问老师问题时，总会给我一些提示，而不是直接告诉我应该怎么样做，我真心认为这种方法很好，锻炼了我自己的思维，在此特别感谢老师的悉心指导！