电子科技大学计算机科学与工程学院

标 准 实 验 报 告

（实验）课程名称 计算机系统结构综合实验

电子科技大学教务处制表

电 子 科 技 大 学

实 验 报 告

学生姓名：卢晓雅　　学 号：2020080904026　　指导教师：王华

实验地点： 清水河校区主楼 A2 区　　实验时间：4.21 下午

一、实验室名称：国家级计算机实验教学示范中心

二、实验项目名称：单周期 CPU 代码分析

三、实验学时：4

四、实验原理：

一）单周期 CPU 的特点是：在一条指令的所有操作全部完成后，才开始执行下一条指令。它执行一条指令需要的硬件部件有：

1. 与取指令有关的电路：指令存储器、程序计数器 PC、修改 PC 值的加法器、选择不同 PC 值的多路选择器等；

2. 与数据处理有关的电路，具体说来包括下述电路：

1）算术逻辑运算部件，如 ALU；

2）与寄存器有关的部件，如寄存器堆；

3）与存储器有关的部件，如数据存储器；

4）与处理方式有关的部件，如多路选择器（选择器的具体规格视具体情况而定）；

5）与立即数处理有关的部件，如数据扩展器、移位器等。

二）硬件描述语言（Hardware Description Languages），是设计硬件时

使用的语言，常用的有 Verilog HDL 和 VHDL。两者的异同点包括：

1. 两者均为常用硬件描述语言，都有各自的 IEEE 标准；

2. 两者是不同的硬件描述语言；

3. Verilog HDL 带有 C 语言的风格，是工业界常用的 HDL；

4. VHDL 带有 C++的风格；

【说明】本课程只使用 Verilog HDL 语言，因此报告中均采用 Verilog HDL 语言进行器件设计。

三）本次课程的软件环境介绍：

1. 操作系统：Windows 7（实验室软件安装在 32 位操作系统下）；

2. 开发平台：Xilinx ISE Design Suite 14.7 集成开发系统

四）本实验课所有设计的 CPU 支持的指令集（32 位）如表 1 所示，其中：

- Op 和 func 为操作码；
- shift 保存要移位的位数；
- rd、rs、rt 表示寄存器号；
- immediate 保存立即数的低 16 位；
- offset 为偏移量；
- address 为转移地址的一部分。

表 1

指令	指令意义	Op[31:26]	func[25:20]	[19:15]	[14:10]	[9:5]	[4:0]
add	寄存器加法	000000	000001	00000	rd	rs	rt
and	寄存器与	000001	000001	00000	rd	rs	rt
or	寄存器或	000001	000010	00000	rd	rs	rt
xor	寄存器异或	000001	000100	00000	rd	rs	rt
srl	逻辑右移	000010	000010	shift	rd	00000	rt
sll	逻辑左移	000010	000011	shift	rd	00000	rt
addi	立即数加法	000101	16 位 immediate			rs	rt
andi	立即数与	001001	16 位 immediate			rs	rt
ori	立即数或	001010	16 位 immediate			rs	rt
xori	立即数异或	001100	16 位 immediate			rs	rt
load	取整数数据字	001101	16 位 offset			rs	rt
store	存整数数据字	001110	16 位 offset			rs	rt
beq	相等则跳转	001111	16 位 offset			rs	rt
bne	不相等则跳转	010000	16 位 offset			rs	rt
jump	无条件跳转	010010	26 位 address				

指令功能说明如下：

1、对于 add/and/or/xor　rd,rs,rt 指令　//rd←rs op rt　其中 rs 和 rt 是两个源操作数的寄存器号，rd 是目的寄存器号。

2、对于 sll/srl　rd,rt,shift 指令　//rd←rt 移动 shift 位

3、对于 addi　rt,rs,imm 指令　//rt←rs+imm(符号拓展)　rt 是目的寄存器号，立即数要做符号拓展到 32 位。

4、对于 andi/ori/xori　rt,rs,imm 指令　//rt←rs op imm(零拓展) 因为是逻辑指令，所以是零拓展（32 位）。

5、对于 load　rt,offset(rs)　指令　//rt←memory[rs+offset]　load 是一条取存储器字的指令。寄存器 rs 的内容与符号拓展的 offset 相加得到存储器地址。从存储器取来的数据存入 rt 寄存器。

6、对于 store　rt,offset(rs)　指令　// memory[rs+offset]←rt　store 是一条存字指令。存储器地址的计算方法与 load 相同。

7、对于 beq　rs,rt,label 指令　//if(rs==rt)　PC←label

beq 是一条条件转移指令。当寄存器 rs 内容与 rt 相等时，转移到 label。如果程序计数器 PC 是 beq 的指令地址，则 label=PC+4+offset<<2。offset 左移两位导致 PC 的最低两位永远是 0，这是因为 PC 是字节地址，而一条指令要占 4 个字节。offset 要进行符号拓展，因为 beq 能实现向前和向后两种转移。

8、bne 指令与 beq 类似，当寄存器 rs 内容与 rt 不相等时，转移到 label。

9、对于 jump　target 指令　//PC←target　jump 是一条跳转指令。target 是转移的目标地址，32 位，由 3 部分组成：最高 4 位来自于 PC+4 的高 4 位，中间 26 位是指令中的 address，最低两位为 0。

五、实验目的：

1. 掌握单周期 CPU 的特点；

2. 熟悉 Verilog HDL 硬件设计语言设计方法及其相关语法等；

3. 熟悉 Xilinx ISE Design Suite 14.7 集成开发环境的操作方法和仿真方法。

六、实验内容：

1. 认真阅读并分析所给的单周期 CPU 代码，掌握单周期 CPU 电路结构中各模块的工作原理；

2. 对单周期 CPU 中两个模块进行仿真，分析并理解仿真结果，验证模块逻辑功能：

1）IF_STAGE（取指阶段）；

2）Control_Unit（控制单元）；

3. 自行设计一个指令序列，要求尽可能涵盖 CPU 指令集中不同类型的指令。将指令序列写入指令存储器 inst_mem 中，使用该指令序列对指定的 SCCPU.v（单周期 CPU 完整电路模块）进行仿真，记录仿真结果并进行分析，直至仿真结果与预期相符。

七、实验器材：

联想 XiaoXin13Pro 笔记本电脑

处理器　AMD Ryzen 7 4800U with Radeon Graphics（1.80 GHz）

机带 RAM　16.0 GB

操作系统, 64 位 Windows10 操作系统，基于 x64 的处理器

八、实验步骤：

step1：创建工程（Project）

1）启动 ISE 软件，然后选择菜单 File→New Project，弹出 New Project Wizard 对话框，在对话框中输入，并指定工作路径，如图 1 所示。

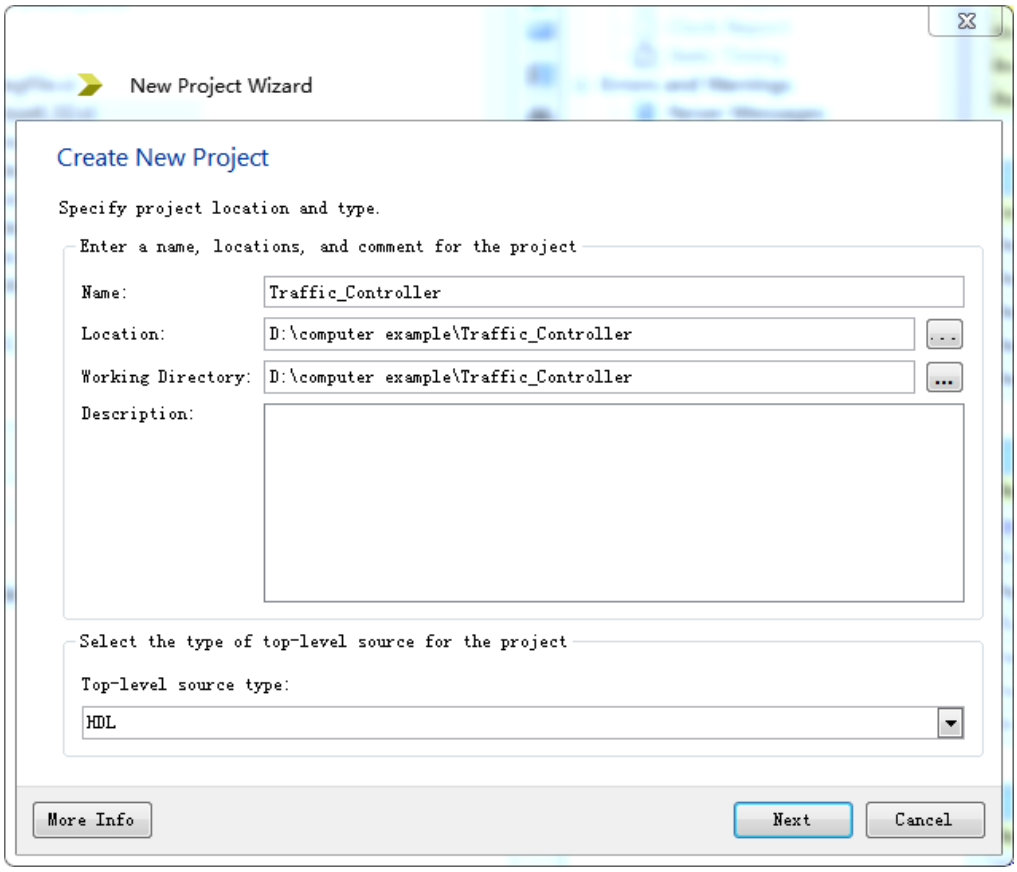


图 1

图 1 中指定的工程名为 Traffic_Controller，指定的工作路径为：D:\Computer_Example\Traffic_Controller

2）在上图中输入完工程名和工作路径后，点击 Next 进入下一页：Project Settings。如图 2 所示：

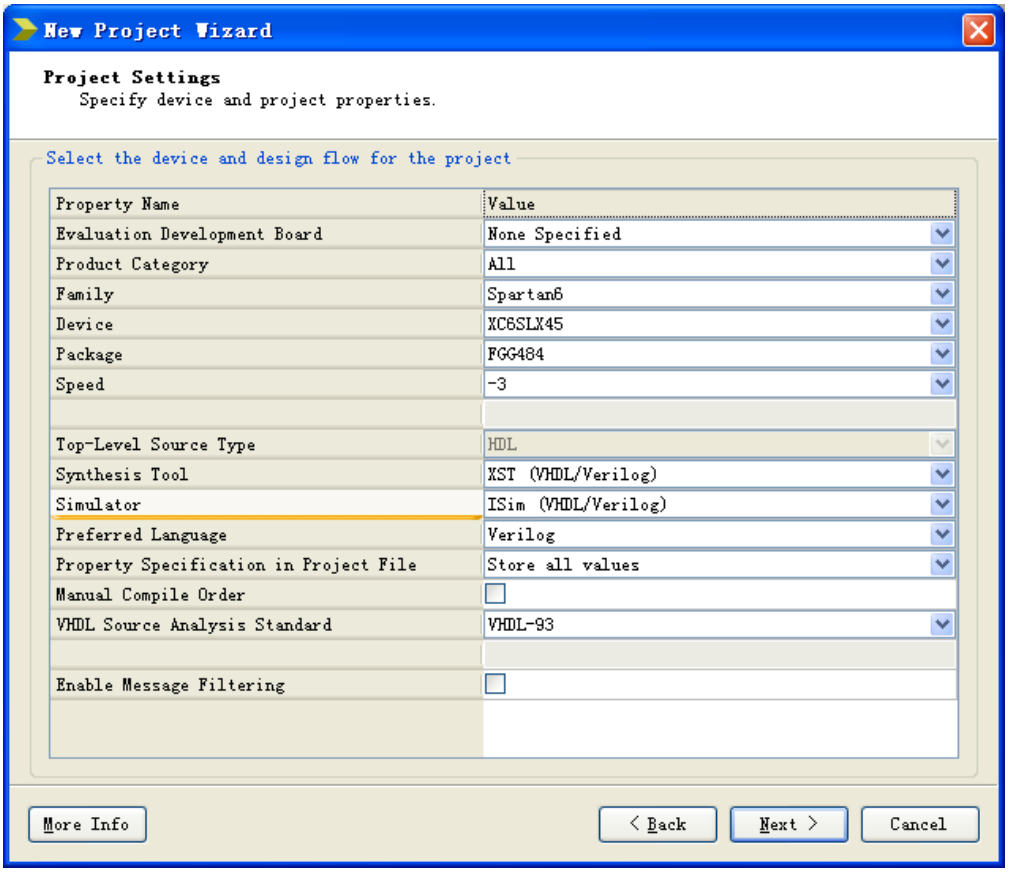


图 2

3）设置好后，在上图中点击 Next 进入下一页：Project Summary，如图 3 所示，这里显示了新建工程的信息，确认无误后，点击 Finish 按钮就可以建立一个新的工程了。

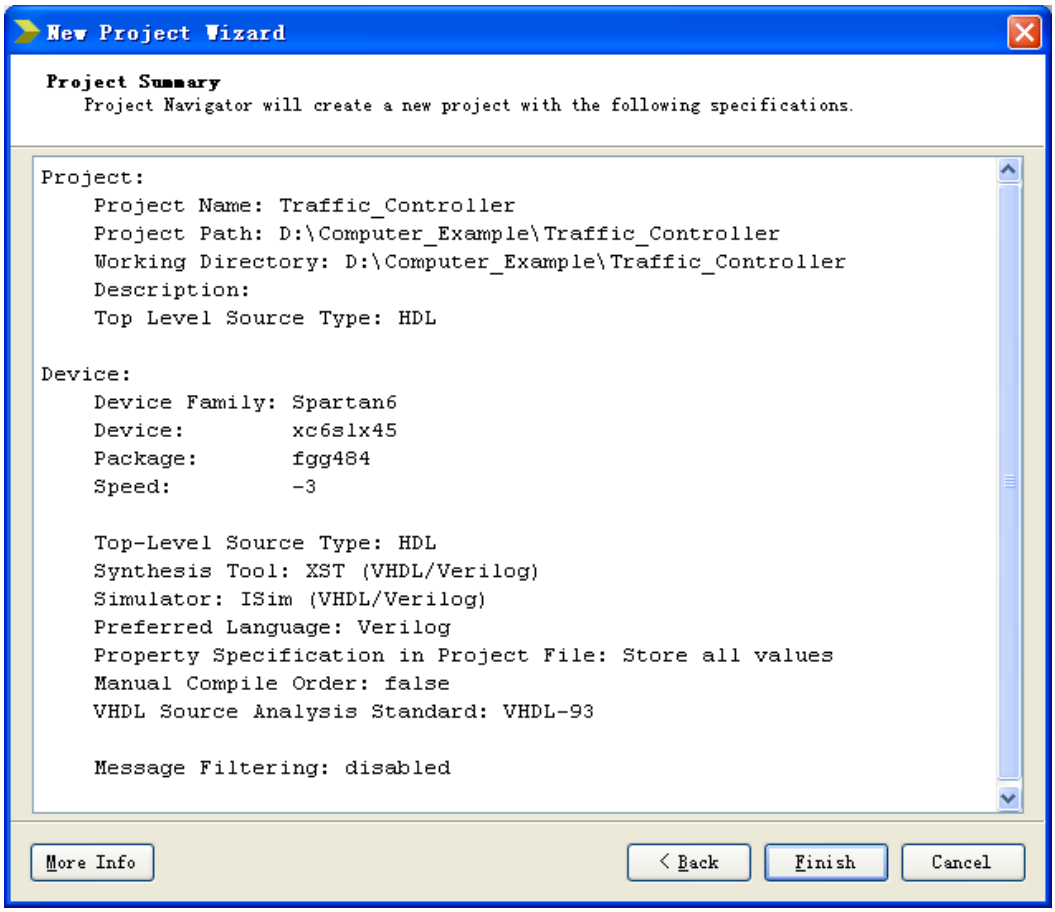


图 3

step2：创建模块（Module）

1）ISE 集成开发环境如图 4 所示，主要分为 4 个区：工程管理区、过程管理区、源代码编辑区和信息显示区。

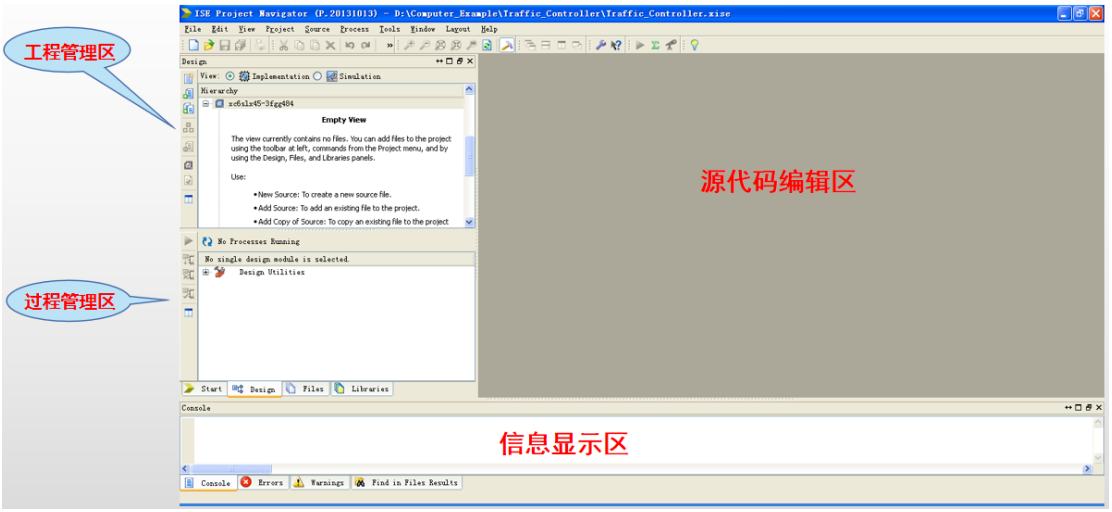


图 4

2）在工程管理区任意位置单击鼠标右键，在弹出的菜单中选择 New Source，会弹出如图 5 所示的 New Source Wizard 对话框：Select Source Type。

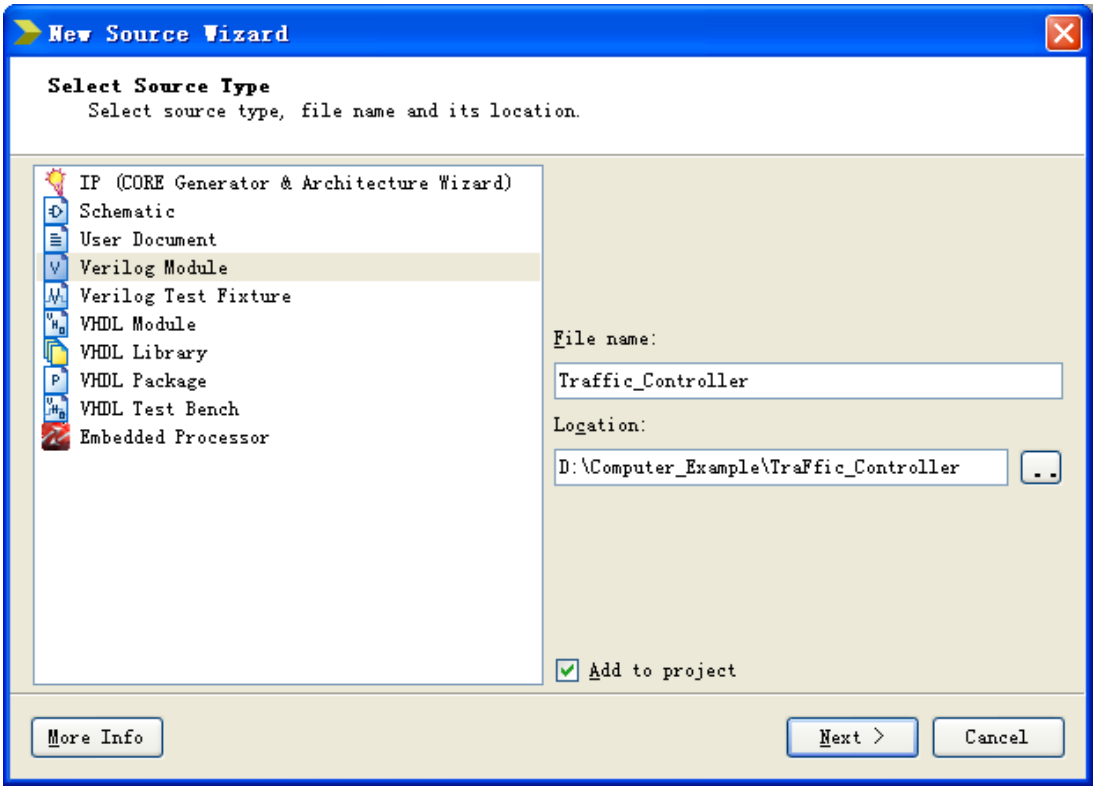


图 5

① 在图中选择 Verilog Module；

② 输入 Verilog 文件名，如： Traffic_Controller；

③ 输入代码存放位置（Location），如

D:\Computer_Example\Traffic_Controller；

④ 点击 Next；

3）接下来进入端口定义对话框：Define Module，如图 6 所示。

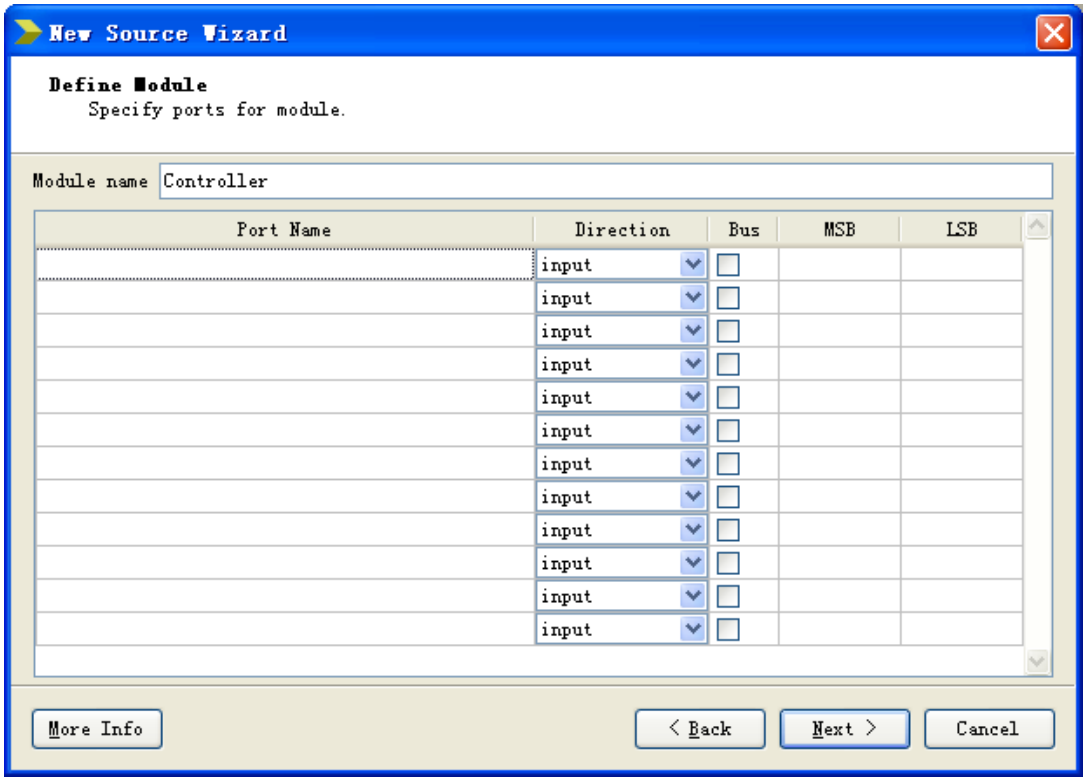


图 6

① Module name 栏用于输入模块名，比如 Controller。

② 下面的列表框用于端口的定义，端口定义这一步略过，在源程序中自行添加。其中：

Port Name 表示端口名称；

Direction 表示端口方向(可选择为 input、output 或 inout)；

MSB 表示信号最高位；

LSB 表示信号最低位；

③ 定义完端口后，点击 Next 按钮进入下一步，然后点击 Finish 按钮完成模块创建。

4）开始进行模块设计，代码输入完成后，首先检查 Verilog HDL 语法：在工程管理区选中要检查的模块，在过程管理区双击 Synthesize – XST→Check Syntax。

① 如果有语法错误，会在信息显示区给出指示，请检查调试。

② 如果没有语法错误，在模拟仿真前要进行 Verilog HDL 代码综合。

5）所谓综合，就是将 Verilog HDL 语言、原理图等设计输入翻译成由与、或、非门和 RAM、触发器等基本逻辑单元的逻辑连接，并根据目标和要求（约束条件）优化生成的 RTL（Register-Transfer-Level）层连接。在工程管理区的 View 中选择 Implementation，并选中要综合的模块 Controller，然后在过程管理区双击 Synthesize-XST，就开始综合过程，如图 7 所示。

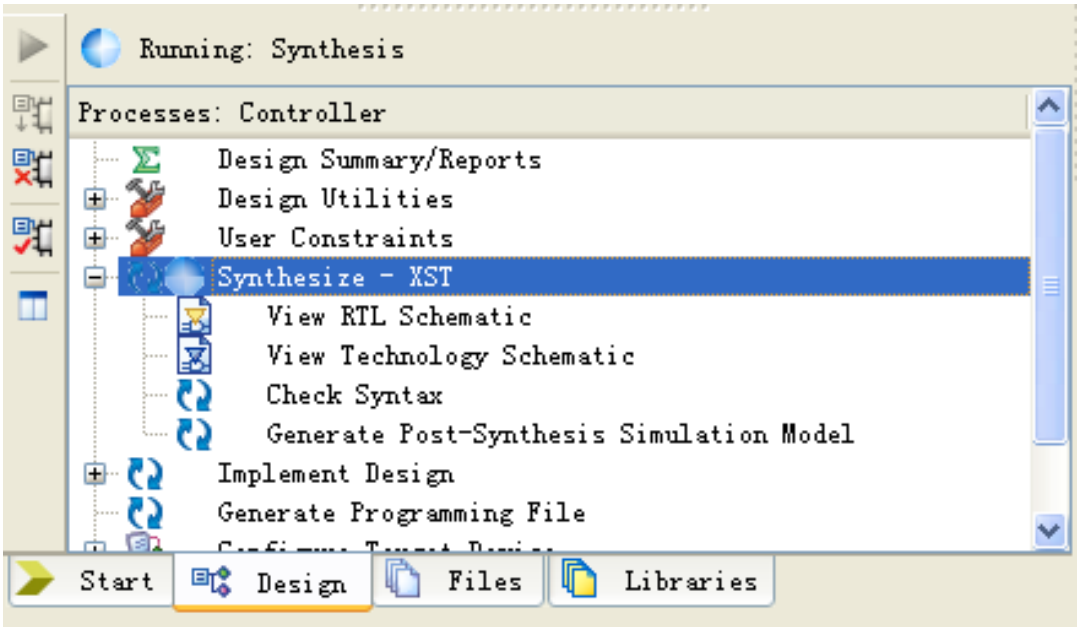


图 7

step3：仿真（Simulation）

仿真并不是设计过程必须的步骤。但是为了尽量减少设计中的错误，在将所做设计下载到开发板上进行板级验证之前，对所做设计进行仿真是必要的。

在工程管理区任意位置单击鼠标右键，在弹出的菜单中选择 New

Source，会弹出如图 8 所示的 New Source Wizard 对话框：Select Source Type。

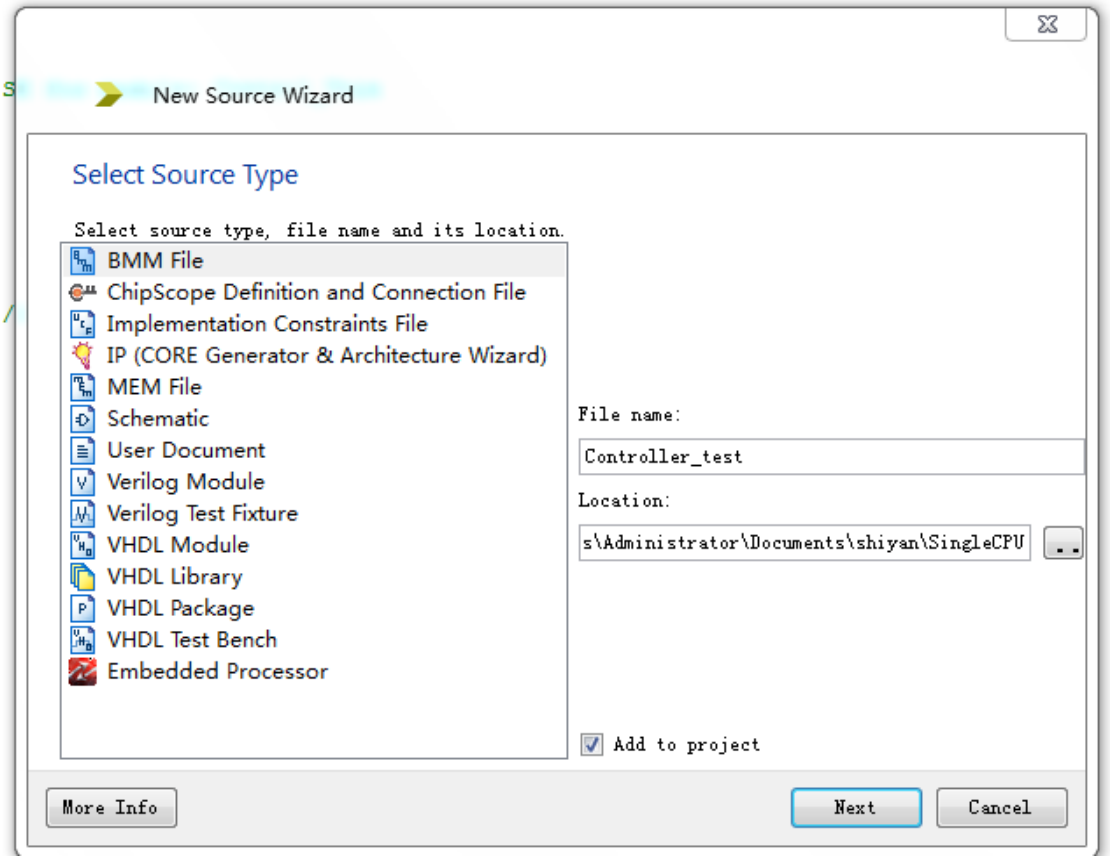


图 8

在图 8 中选择 Verilog Test Fixture，输入测试文件名：Controller_test，单击 Next 按钮，进入下一个对话框，如图 9 所示。

在图 9 中会显示工程中所有的模块名，这里只有一个 Control_Unit。

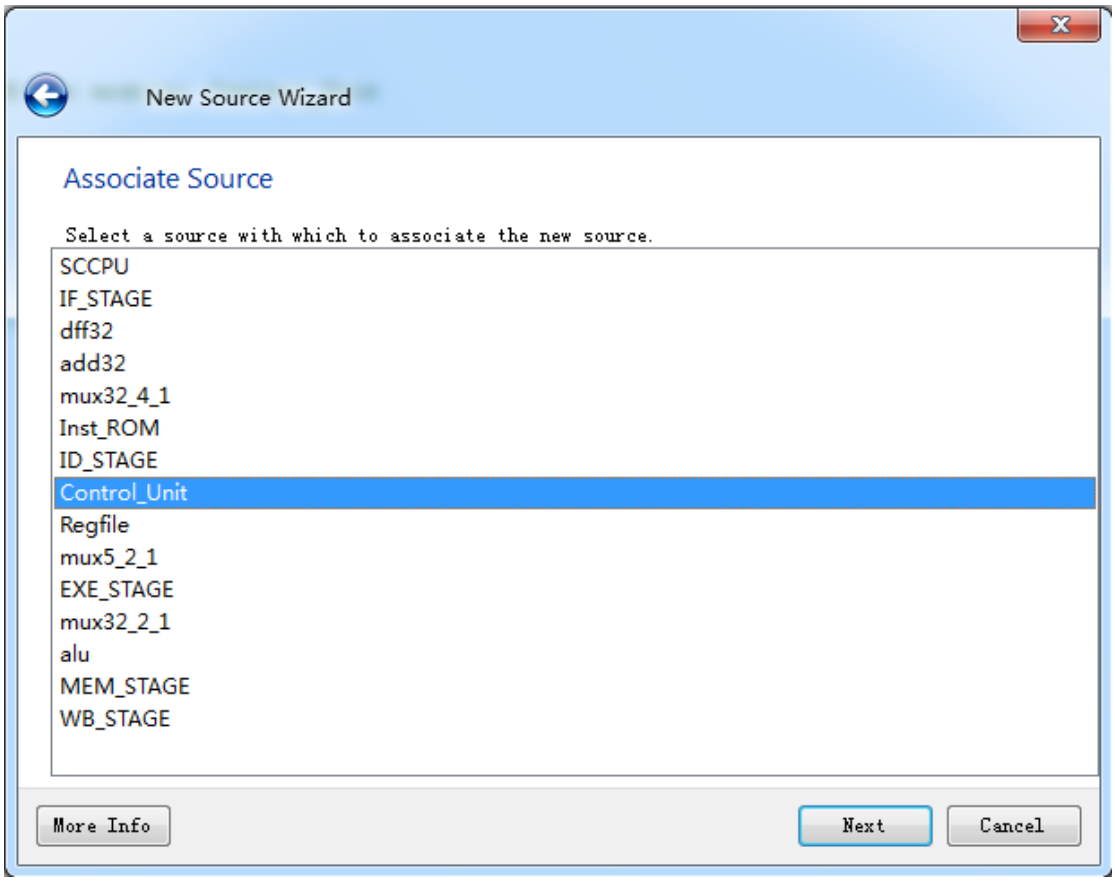


图 9

选择要测试的模块 Control_Unit 按钮，再点击 Finish 按钮，ISE 会在源代码编辑区自动生成测试模块的代码，如图 10 所示。

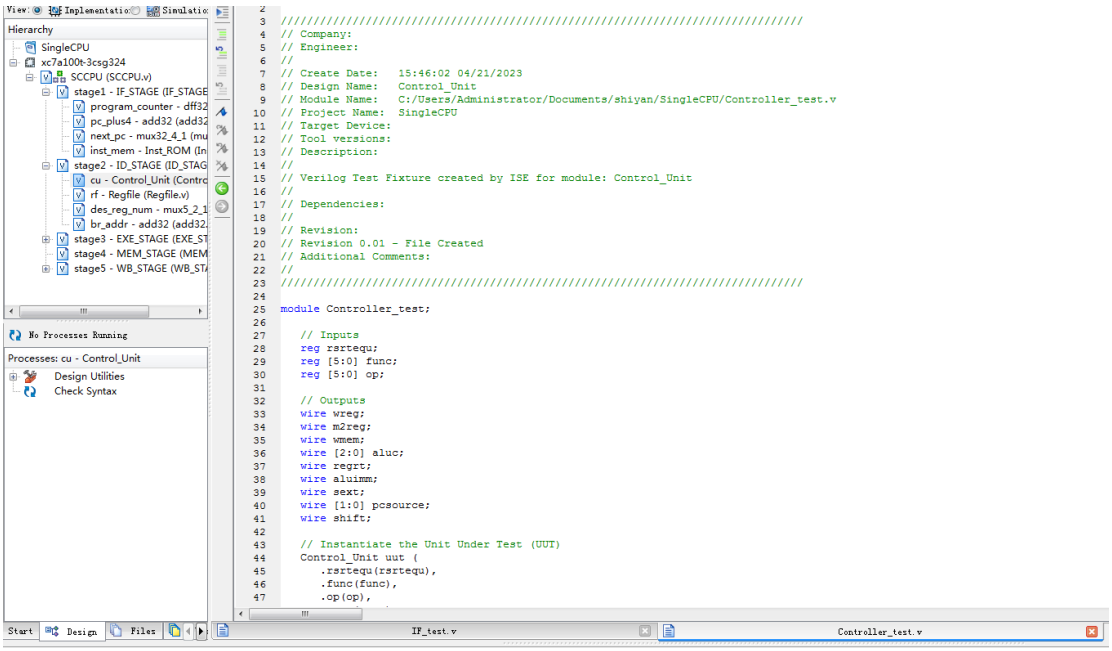


图 10

完成测试文件编辑后，确认工程管理区中 View 选项设置为

Simulation，并选中 Controller_test 这时在过程管理区会显示与仿真有关的进程，如图 11 所示。

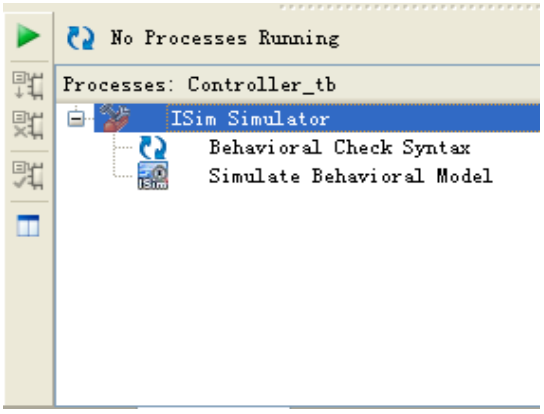


图 11

单击其中的 Simulate Behavioral Model 项，选择弹出菜单中的 Process Properties 项，会弹出如图 12 所示的属性设置对话框。

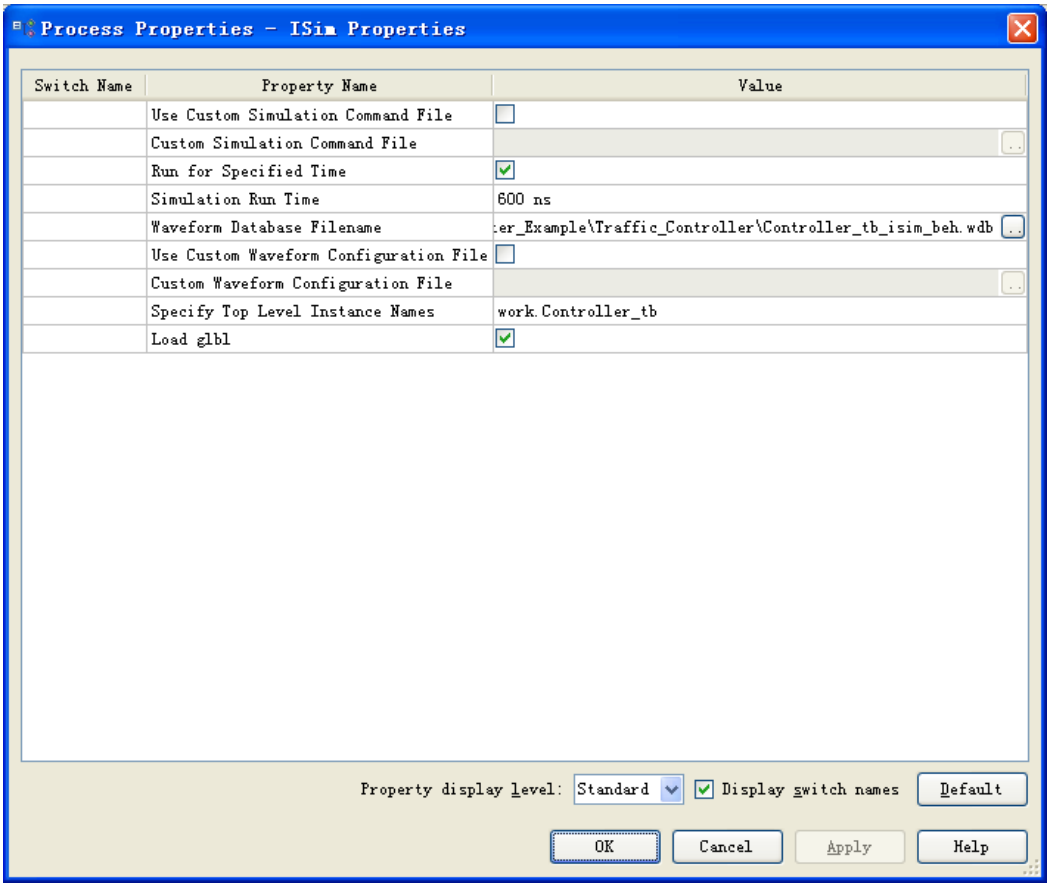


图 12

其中 Simulation Run Time 是仿真时间的设置，可将其修改为任意

时长，由于测试模块 Controller_test 中测试时间定义为 300ns，故在图 12 的仿真测试设置为 600ns。

仿真参数设置完后，就可以进行仿真。首先在工程管理区选中测试代码，然后在过程管理区双击 Simulate Behavioral Model，ISE 将启动 ISE Simulator，可以得到仿真结果，如图 13 所示。

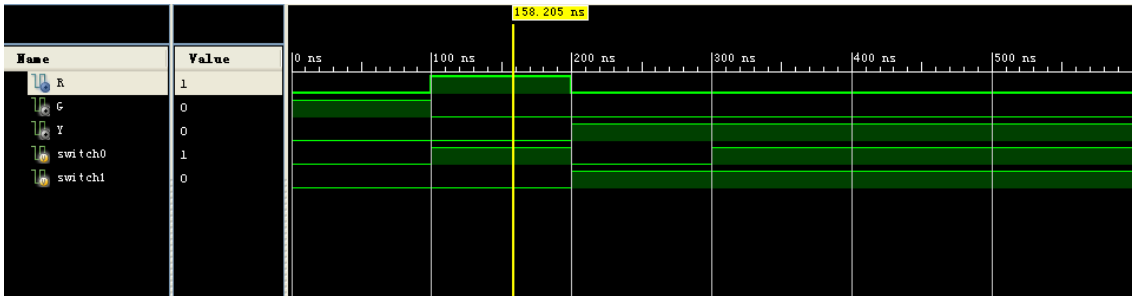
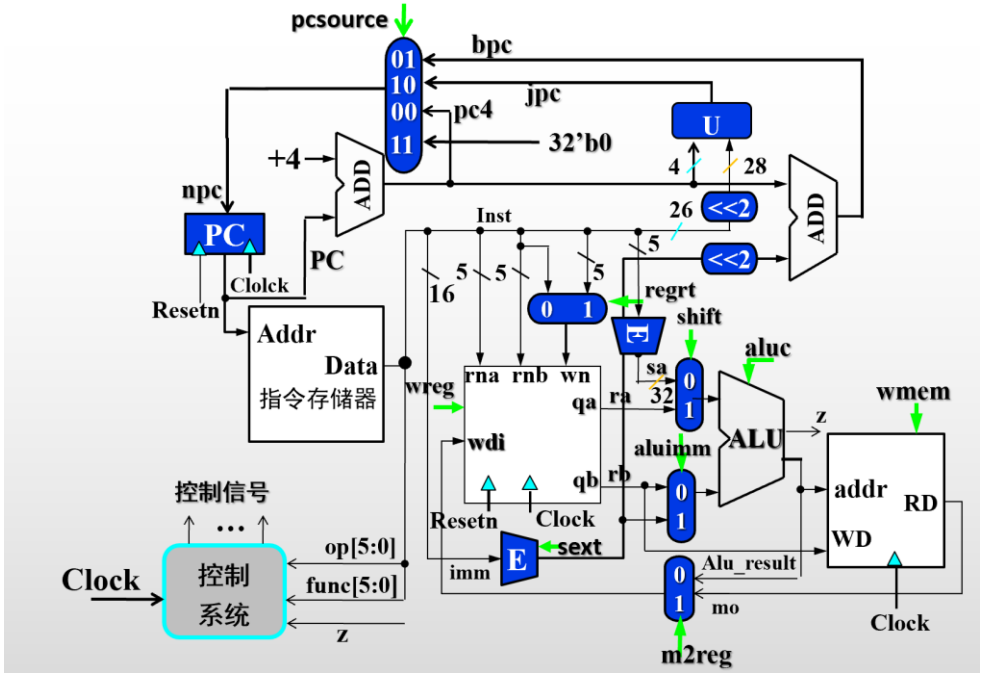


图 13

九、实验数据及结果分析：（实验运行结果介绍或者截图，对不同的结果进行分析）

1. 补充完整单周期 CPU 微架构图（包括统一信号名），并进行必要的文字说明。

补充完整的单周期 CPU 微架构图如下：



相应的文字说明（重点就补充的部分进行说明）：

xor	寄存器异或	000001	000100	00000	rd	rs	rt
srl	逻辑右移	000010	000010	shift	rd	00000	rt

从指令格式可以看出，移位指令中，ALU 的第一个操作数不是 rs 的内容，而是 shift（Inst[19:15]），所以需要在 ALU 之前加一个 MUX 来选择 R[rs]还是 shift，并由信号 shift 控制。由于 R[rs]为 32 位，shift 为 5 位，所以要先扩展 shift 至 32 位，再使用 32 位的 MUX。参考 EXE_STAGE 中的代码，即可将相应部分补充完整。

```
assign sa={27'b0,eimm[9:5]};//移位位数的生成

mux32_2_1 alu_ina (ea,sa,eshift,alua);//选择ALU a端的数据来源
mux32_2_1 alu_inb (eb,eimm,ealuimm,alub);//选择ALU b端的数据来源
alu al_unit (alua,alub,ealuc,ealu,z);//ALU
```

参考 RegFile 部分代码可知，图中缺少了控制寄存器堆复位的信号 Resetn，将其补充完整，并修改信号名与顶层单元的代码一致。

需要注意 CPU 架构中所有的寄存器都被复位信号控制，Resetn=0 时寄存器内容清零，代码中虽然使用寄存器堆模拟内存，但实际上内存的内容不应该被单条指令控制清零，所以图中的 Mem 不应该画 Resetn 信号。

2. 分析 IF_STAGE、Control_Unit 两个模块的代码

1）IF_STAGE

输入信号	**信号说明**
clk	时钟信号
clrn	复位信号
pcsource	选择下一条指令的 PC 的来源，共有 bpc、jpc、pc4、

	32'b0 四种来源，使用 2bit 编码
bpc	条件分支指令的 PC
jpc	直接跳转指令的 PC
输出信号	**信号说明**
inst	表示指令内容
pc4	表示 PC+4
PC	程序计数器的值，指向下一条要读取的指令

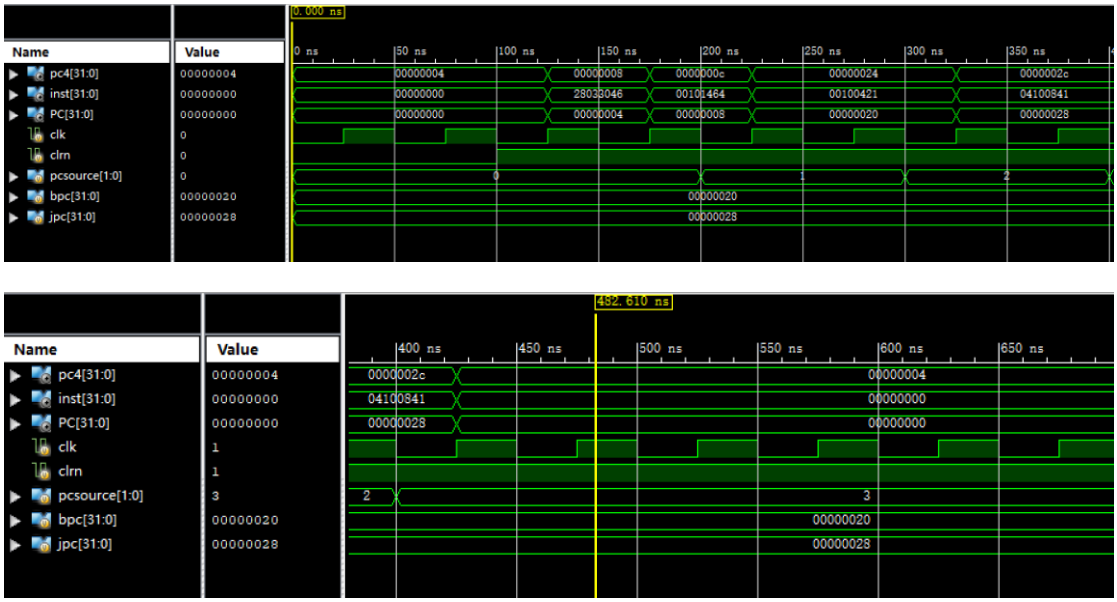
仿真测试代码如下：

使用 5 组测试数据，第一组数据测试复位信号是否有效，后四组数据覆盖了 PC 的四种选择。

```
51      initial begin
52          // Initialize Inputs
53          // 测试复位信号是否有效
54          clk = 0;
55          clrn = 0;
56          pcsource = 0;
57          bpc = 32'h00000020; // 跳转到第8条指令
58          jpc = 32'h00000028; // 跳转到第10条指令
59
60          // Wait 100 ns for global reset to finish
61          #100;
62          clrn = 1;
63          pcsource = 0;
64          bpc = 32'h00000020;
65          jpc = 32'h00000028;
66
67          #100;
68          clrn = 1;
69          pcsource = 2'b01;
70          bpc = 32'h00000020;
71          jpc = 32'h00000028;
72
73          #100;
74          clrn = 1;
75          pcsource = 2'b10;
76          bpc = 32'h00000020;
77          jpc = 32'h00000028;
```

```
78
79          #100;
80          clrn = 1;
81          pcsource = 2'b11;
82          bpc = 32'h00000020;
83          jpc = 32'h00000028;
84
85      end
86  always #25 clk=~clk; // 周期为50nm的方波信号
```

仿真结果（截图）：



对结果进行简要的说明：

t=0ns 时，clrn=1，PC 被复位，PC4=PC+4=4,0 地址存储的指令为空；

t=100ns 时，clrn=1，pcsource=0，时钟上升沿更新 PC=PC4=4，取得的指令为 0x28033046;下一个上升沿更新 PC=PC4=8，取得指令 0x00101464；

t=200ns 时，clrn=1，pcsource=2’b01，bpc=0x00000020，时钟上升沿更新 PC=bpc，取得第 8 条指令，即 0x00100421;

t=300ns 时，clrn=1，pcsource=2’b10，jpc=0x00000028，时钟上升沿

更新 PC=jpc，取得第 10 条指令，即 0x04100841;

t=400ns 时，clrn=1，pcsource=2’b11，时钟上升沿更新 PC=32’b0，即被复位。

```
assign rom[6'h00]=32'h00000000;     //0地址为空，从1地址开始执行；
assign rom[6'h01]=32'h28033046;     //ori r6,r2,0x00cc   r6=0x000000ce
assign rom[6'h02]=32'h00101464;     //add r5,r3,r4   r5=0x00000007

assign rom[6'h08]=32'h00100421;         //add r1,r1,r1
assign rom[6'h09]=32'h00100421;         //add r1,r1,r1
assign rom[6'h0A]=32'h04100841;         //and r2,r2,r1
```

结论：

结果符合预期。

2）Control_Unit

输入信号	**信号说明**
rsrtequ	判断 ALU 输出结果是否为 0：if(r=0)rsrtequ=1
func	R 类型指令中控制码字段
op	指令中相应控制码字段
输出信号	**信号说明**
wreg	寄存器堆的写使能信号
m2reg	选择回写的内容来自 mem 还是 ALU 计算结果
wmem	内存的写使能信号
aluc	ALU 控制码
regrt	选择写入的目的寄存器的编号
aluimm	选择 ALU 的第二个操作数是 R[rt]还是 imm16
sext	控制选择符号扩展还是逻辑扩展
pcsource	PC 多路选择器控制码

shift	为移位运算选择操作数

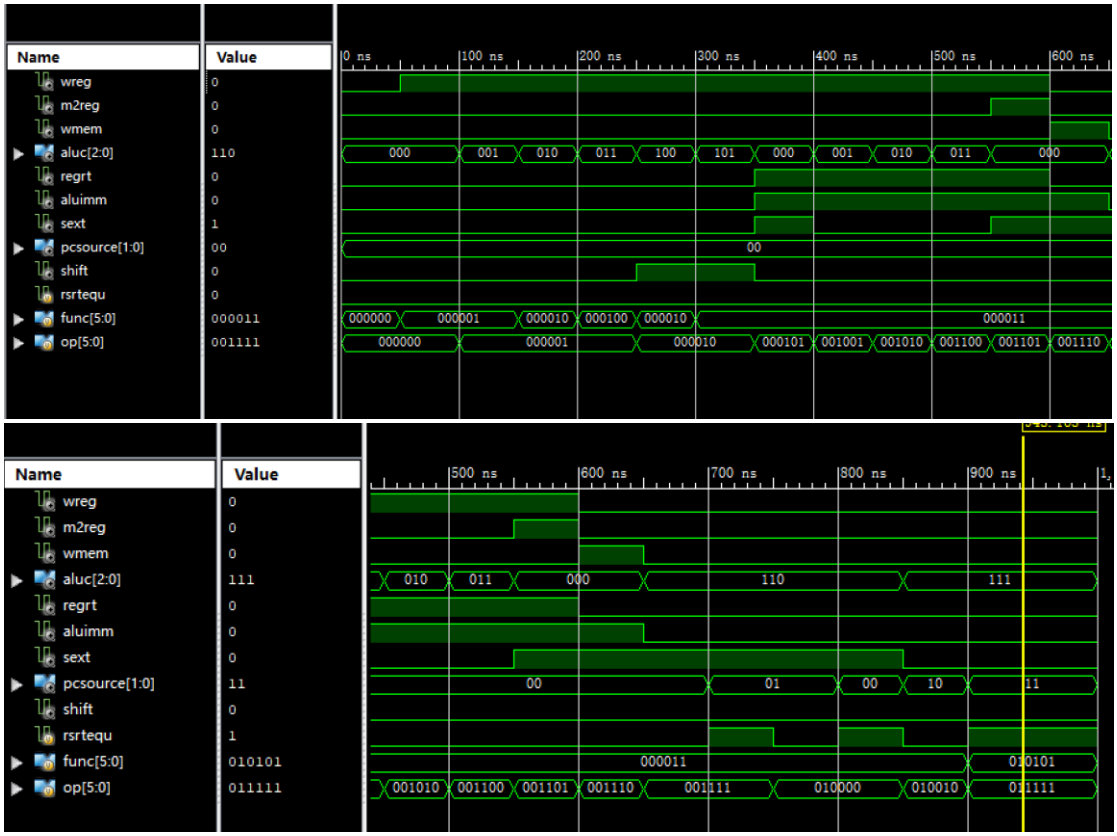
仿真测试代码如下：

```
59      initial begin
60          // Initialize Inputs
61          rsrtequ = 0; //判断ALU输出结果是否为0，转移条件是否成立
62          func = 0;
63          op = 0;
64
65          // Wait 50 ns for global reset to finish
66          // add
67          #50;
68          rsrtequ = 0;
69          op=0;
70          func=5'b00001;
71
72          // and
73          #50;
74          rsrtequ = 0;
75          op=6'b000001;
76          func=5'b00001;
77
78          // or
79          #50;
80          rsrtequ = 0;
81          op=6'b000001;
82          func=5'b00010;
83
84          // xor
85          #50;
86          rsrtequ = 0;
87          op=6'b000001;
88          func=5'b00100;
89
90          // srl
91          #50;
92          rsrtequ = 0;
93          op=6'b000010;
94          func=5'b00010;
95
96          //sll
97          #50;
98          rsrtequ = 0;
99          op=6'b000010;
100         func=5'b00011;
```

```
101
102         // addi
103         #50;
104         rsrtequ = 0;
105         op=6'b000101;
106
107         // andi
108         #50;
109         rsrtequ = 0;
110         op=6'b001001;
111
112         // ori
113         #50;
114         rsrtequ = 0;
115         op=6'b001010;
116
117         // xori
118         #50;
119         rsrtequ = 0;
120         op=6'b001100;
121
122         // load
123         #50;
124         rsrtequ = 0;
125         op=6'b001101;
126
127         // store
128         #50;
129         rsrtequ = 0;
130         op=6'b001110;
131
132         // beq
133         #50;
134         rsrtequ = 0;
135         op=6'b001111;
136
137         #50;
138         rsrtequ = 1;
139         op=6'b001111;
140
```

```
141         //bneq
142         #50;
143         rsrtequ = 0;
144         op=6'b010000;
145 
146         #50;
147         rsrtequ = 1;
148         op=6'b010000;
149 
150         // jump
151         #50;
152         rsrtequ = 0;
153         op=6'b010010;
154 
155         // 无匹配项
156         #50;
157         rsrtequ = 1;
158         op=6'b011111;
159         func=5'b10101;
```

仿真结果（截图）：



【对结果进行说明】

各类指令对应产生的控制信号如下：

指令	rsrt equ	op	func	wreg	m2reg	wmem	aluc	regrt	aluimm	sext	pcsource	shift
Null	0	0	0	0	0	0	000	0	0	0	00	0
Add	\	0	1	1	0	0	000	0	0	\	00	0
And	\	1	1	1	0	0	001	0	0	\	00	0
Or	\	1	10	1	0	0	010	0	0	\	00	0
Xor	\	1	100	1	0	0	011	0	0	\	00	0
Srl	\	10	10	1	0	0	100	0	0	\	00	1
Sll	\	10	11	1	0	0	101	0	0	\	00	1
Addi	\	101	\	1	0	0	000	1	1	1	00	0
Andi	\	1001	\	1	0	0	001	1	1	0	00	0
Ori	\	1010	\	1	0	0	010	1	1	0	00	0
Xori	\	1100	\	1	0	0	011	1	1	0	00	0
Load	\	1101	\	1	0	0	000	1	1	1	00	0
Store	\	1110	\	0	1	1	000	\	1	1	00	0
Beq	0	1111	\	0	\	0	110	\	0	1	00	0
	1	1111	\	0	\	0	110	\	0	1	01	0
bne	0	10000	\	0	\	0	110	\	0	1	01	0
	1	10000	\	0	\	0	110	\	0	1	0	0
Jump	\	10010	\	0	\	0	\	\	\	0	10	0
default	1	11111	10101	\	\	\	111	\	\	\	11	\

对比仿真图可知，当输入相应的 op 和 func 操作码时，各类指令产生的控制信号均与上表相符。

注意 bne 和 beq 也使用符号扩展将 16 位立即数扩充成 32 位的 PC

如果指令无法匹配任何一种类型，则说明出错，pcsourse=2’b11，将 PC 复位。

【结论】

结果符合预期。

3. 分析 SCCPU（单周期 CPU）代码

1）测试的指令序列如下所示，对应的初始化 inst_mem 部分 Verilog 代码如下：

以下指令包含 lw、sw、jump、R 型、beq、i 型

```
assign rom[6'h00]=32'h00000000;    //0 地址为空，从 1 地址开始执行；
assign rom[6'h01]=32'h28033046;    //ori r6,r2,0x00cc r6=0x000000ce
assign rom[6'h02]=32'h00101464;    //add r5,r3,r4    r5=0x00000007
assign rom[6'h03]=32'h38000866;    //store r6,0x0002(r3)
m5=0x000000ce
assign rom[6'h04]=32'h34000489;    //load r9,0x0001(r4)
r9=0x000000ce
assign rom[6'h06]=32'h3c000c21;    //beq   r1,r1,6'h0a      offset=0x0003
相等转移到 0ah 处
assign rom[6'h07]=32'h00100421;              //add r1,r1,r1
```

```
assign rom[6'h08]=32'h00100421;              //add r1,r1,r1
assign rom[6'h09]=32'h00100421;              //add r1,r1,r1
assign rom[6'h0A]=32'h04100841;              //and r2,r2,r1
assign rom[6'h0B]=32'h04200823;              //or r2,r1,r3
assign rom[6'h0C]=32'h044020e5;              //xor r8,r7,r5
assign rom[6'h0D]=32'h14000901;              //addi r1,r8,0x02
assign rom[6'h0E]=32'h0821a408;              //sr1 r9,r8,5'h03
assign rom[6'h0F]=32'h14002d29;              //addi r9,r9,0x000b
assign rom[6'h10]=32'h27ffc107;              //andi r7,r8,0xfff0
assign rom[6'h11]=32'h3003fd27;              //xori r7,r9,0x00ff
assign rom[6'h12]=32'h43ffbc21;              //bne r1,r1,6'h02
assign rom[6'h13]=32'h48000001;              //jump 0x0000001    无条件转移到 01h 处
assign rom[6'h14]=32'h08218804;      //srl r2, r4, 3
// 32’b 000010 000010 00011 00010 00000 00100
assign rom[6'h15]=32'h00102021;      //add r8 r1 r1
// 32’b 000000 000001 00000 01000 00001 00001
assign  rom[6'h16]=32'h48000002;    //jump  0x00000A0  无条件转移到 A0h 处
// 32’b 010010 0000 0000 0000 0000 0000 1010 0000
```

2）顶层模块的信号如下表所示：

输入信号	信号说明
Clock	时钟信号
Resetn	复位信号
输出信号	**信号说明**
PC	程序计数器的值，指向下一条要读取的指令
Inst	指令的内容
Alu_Result	记录 ALU 运算的结果

3）regfile 初始化部分数据：

```
register[5'h01]<=32'h00000001;
register[5'h02]<=32'h00000002;
register[5'h03]<=32'h00000003;
register[5'h04]<=32'h00000004;
register[5'h05]<=32'h00000005;
register[5'h06]<=32'h00000006;
register[5'h07]<=32'h00000007;
register[5'h08]<=32'h00000008;
register[5'h09]<=32'h00000009;
```

4）数据存储器初始化数据：

```
for(i=0;i<32;i=i+1)              //存储器清零
        ram[i]=0;
    ram[5'h01]=32'h0000000a;      //存储器对应地址初始化赋值
    ram[5'h02]=32'h0000000b;
```

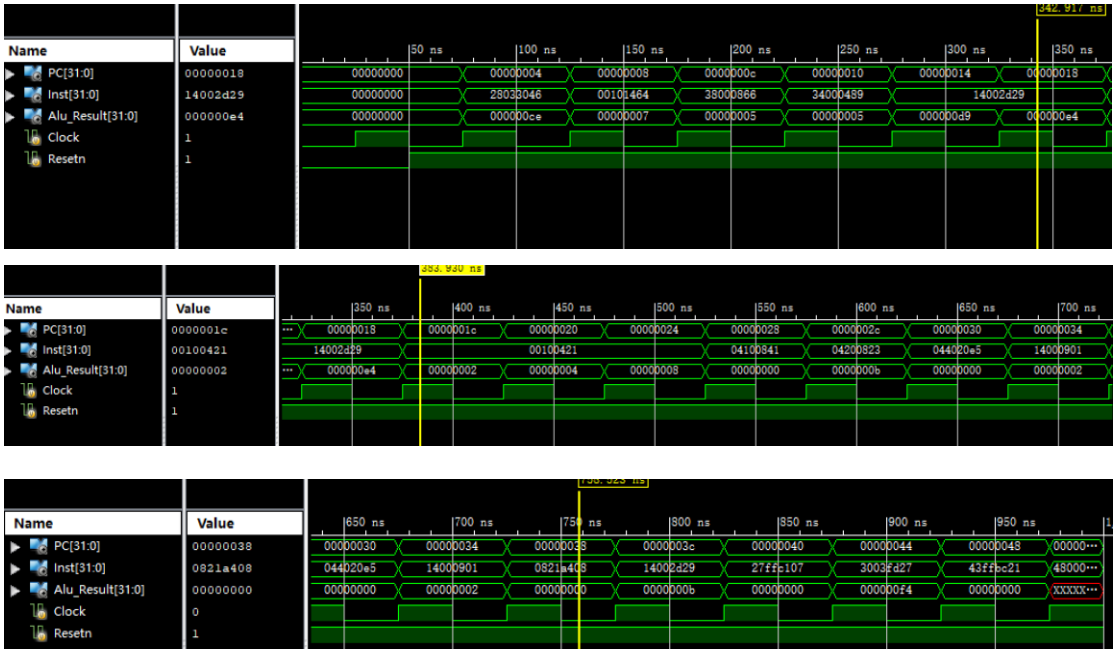
ram[5'h03]=32'h0000000c;

ram[5'h04]=32'h0000000d;

5）SCCPU 模块仿真测试代码如下：

```
45      initial begin
46          // Initialize Inputs
47          Clock = 0;
48          Resetn = 0;
49
50          # 50
51          Resetn = 1;
52
53      end
54
55      always #25 Clock=~Clock; // 周期为50nm的方波信号
```

6）仿真结果：



【对结果进行说明】

50ns 之前 Resetn=0，时钟上升沿到来时，PC 也没有更新，一直保持 0，t=50ns 时 Resetn=1，之后当时钟上升沿到来时，PC 更新，t=75ns 时取到第一条指令 0x28033046，前 11 条指令都不是跳转指令，第 12 条指令由于条件不成立没有跳转，

【结论】

结果符合预期。

十、实验结论：

本次实验中，我们详细阅读并理解了单周期 CPU 的代码，添加了激励文件对顶层模块、IF_STAGE 模块和 Control_Unit 模块进行测试，从而对单周期 CPU 的结构和工作流程有了大致的理解。

十一、总结及心得体会：

从数字电路实验到组成原理实验，再到这次的体系结构实验，我对 Verilog 语言的理解更加深入，对单周期 CPU 结构的理解也越来越全面。

十二、对本实验过程及方法、手段的改进建议：

本次实验中有以下几点需要注意：

1） 区分时序逻辑元件与组合逻辑元件，参考代码中的 EXE_STAGE 部件和 WB_STAGE 部件都是组合逻辑元件，所以不用传入 Clock 信号。测试时序逻辑元件时，不仅要对 input 信号进行赋值，还要设置一个周期性的时钟信号，比如本次实验中使用的周期为 100ns 的方波信号。

2） 区分阻塞赋值和非阻塞赋值。阻塞赋值用“=”表示，是串

行操作，即执行完上一条赋值语句后再执行下一条，非阻塞赋值用“<=”表示，是并行操作，即语句块中的所有赋值语句同时执行，如果有 q1<=d, q2<=q1，则 q2 被赋予的是 q1 原本的值，而不是更新后的 d，语句顺序改变会影响阻塞赋值的结果，而不影响非阻塞赋值的结果，单周期 CPU 的代码中，时序逻辑元件需要在时钟上升沿更新信号，更新的值是上一周期准备好的，所以要使用非阻塞语句进行赋值。

3） 要针对具体的指令格式设计数据通路，本次实验使用的指令格式与课堂所学不同，要注意灵活分析。

报告评分：

指导教师签字：