合肥工业大学计算机与信息学院

计算机组成原理实验教程

即

基于 Verilog HDL 的 CPU 设计实验教程

（含基础代码版）

计算机组成原理课程组

2023.5

目录

实验一　Vivado 平台入门

1.1 实验内容

完成 Vivado 的安装

1.2 实验目标

1）安装 Xilinx 的 Vivado 开发套件。
2）熟悉 Vivado 仿真，原理图，下载方法。

1.3 实验步骤

登录 http://china.xilinx.com/products/design-tools/vivado.html
选择合适的版本下载。（下载需要 Xilinx 账户，请自行申请，后续授权也要用到。）
安装过程不再赘述。

Vivado 平台在工程文件完成编码后一般沿着综合、仿真、链接端口、生成 Bit 文件和下载至开发板的过程完成工程的实践。如图 1.3.1 所示，展现了打开一个 Vivado 工程的示例。

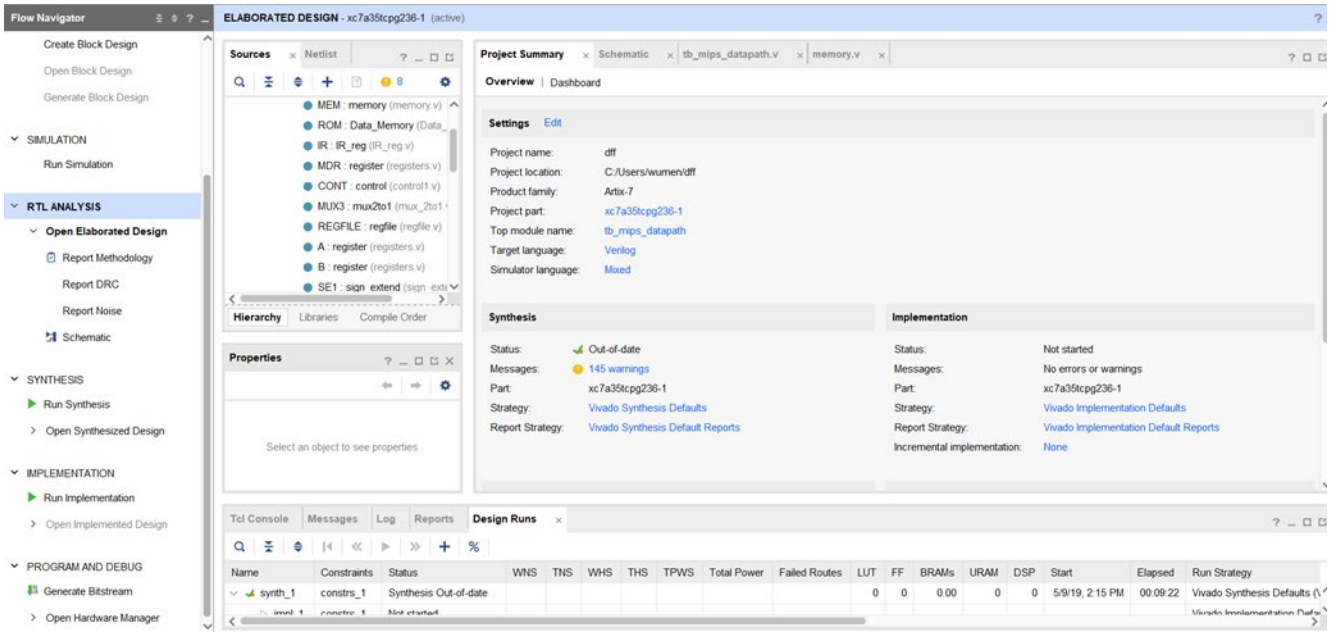


图 1.3.1Vivado 主界面

Flow Navigator：设计流程导航；

Simulation 对顶层文件仿真，可以看到时序波形的变化；

RTL Analysis：进行设计规则检测（DRCs），其下的 Schematic 可以生成器件连线的 RTL 原理图；

Synthesis 是对指定的顶层文件进行层次化综合；

Implementation 是对设计进行实现，链接相关端口；

Program and Debug：对 bitstream 进行设置，Generate Bitstream 生成 bit 流文件，可以打开硬件管理器，将 bit 流文件烧写到指定设备。如果设计中加入了 Debug 核，这里会打开 Vivado 逻辑分析仪，对系统进行 Debug。

例如：点击红框位置进行 Run Implementation 操作。

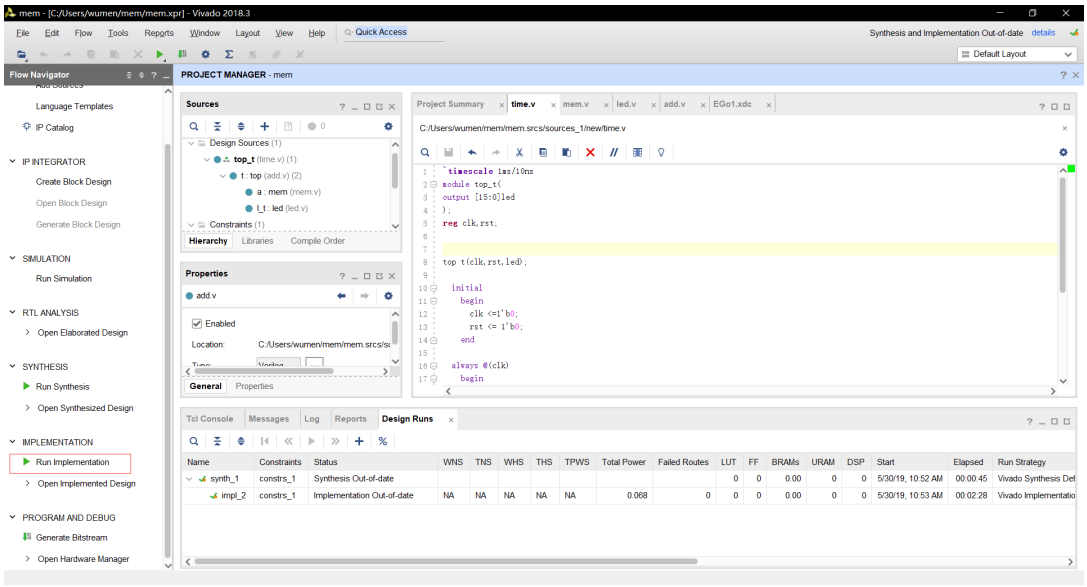


图 1.3.2 Run Implementation 操作

在这两部分设计验证都没有问题的情况下，可以生成 bit 文件，该文件就是用于写入开发板的部分。

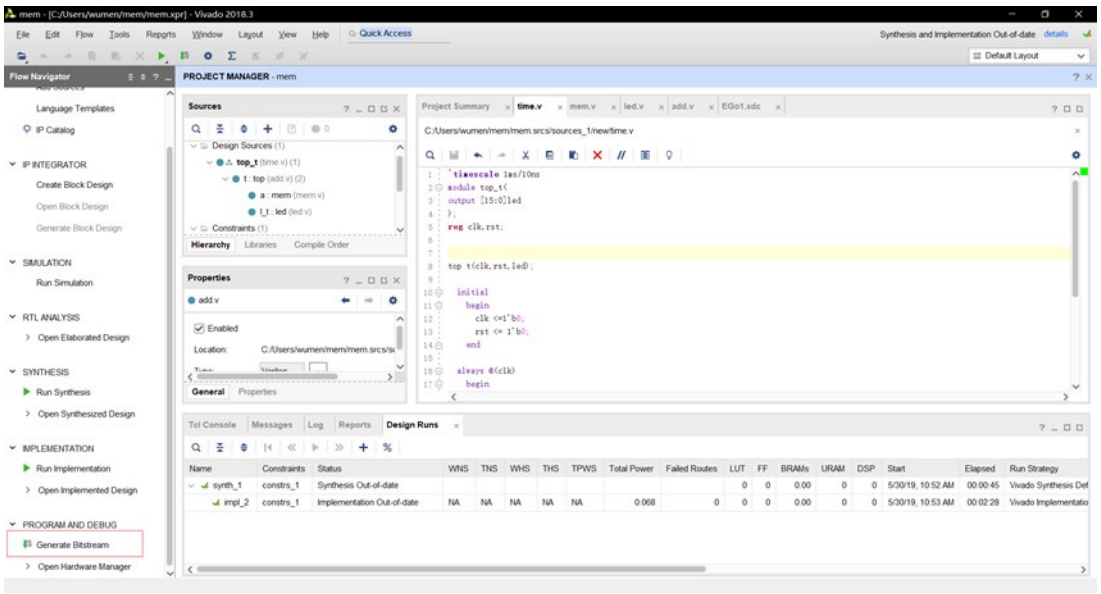


图 1.3.3Generate Bitstream

生成 bit 文件后就能打开硬件向导，连接开发板后就可以进行下载运行。

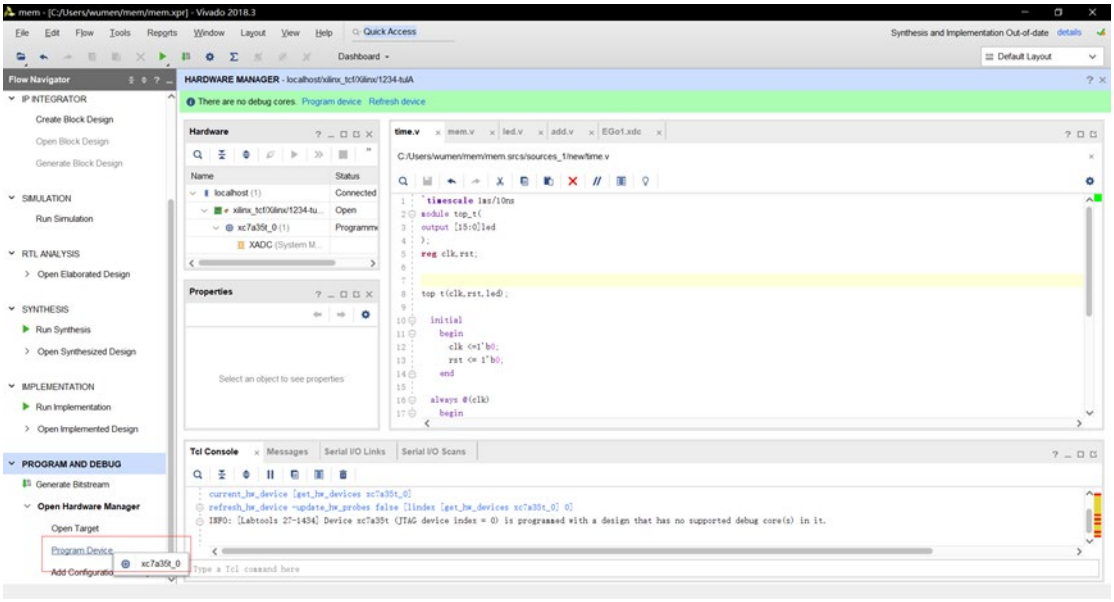


图 1.3.4Program Device

实验二　　多路选择器的实现

2.1 实验内容

这个实验主要使用 Vivado 软件进行多路数据选择器的设计与实现。多路选择器是现代 RISC CPU 内部数据通用选择的重要部件，是基于总线的数据通路的一部分。

2.2 实验目标

1. 了解 Vivado 平台开发环境，学习 Verilog HDL 语言。
2. 掌握多路选择器的工作原理和逻辑功能。
3. 了解计算机条件指令的实现。

2.3 实验步骤

1. 建立新工程；
2. 设计代码与输入；
3. 代码综合
4. 软件仿真
5. 硬件约束与实现
6. 生成流代码与下载

2.4 实验原理

数据选择器是计算机逻辑电路设计中最重要的基本逻辑电路之一，也是基于数据选择数据通路 CPU 的重要部件。在 Verilog 硬件描述语言中可以用 if…else 或 case 语句来生成一个数据选择器。

以 2 选 1 和 4 选 1 选择器为例，图图 2.4.1 和图图 2.4.2 为多路选择器的原理图：

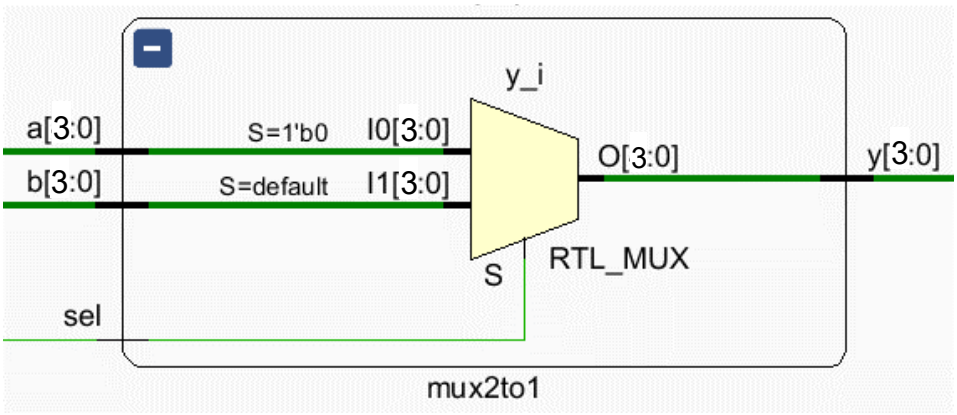


图 2.4.1 2 选 1

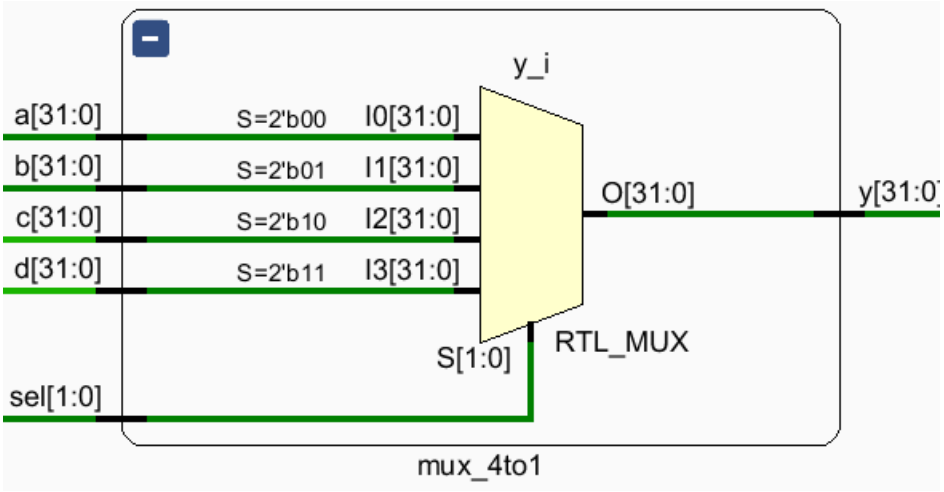


图 2.4.2 4 选 1

2.5 实验步骤

2.5.1 创建工程

（1）双击桌面 Vivado 快捷图标，启动 Vivado。

（2）Create Project 进入 New Project 向导，如图图 2.5.1 所示。

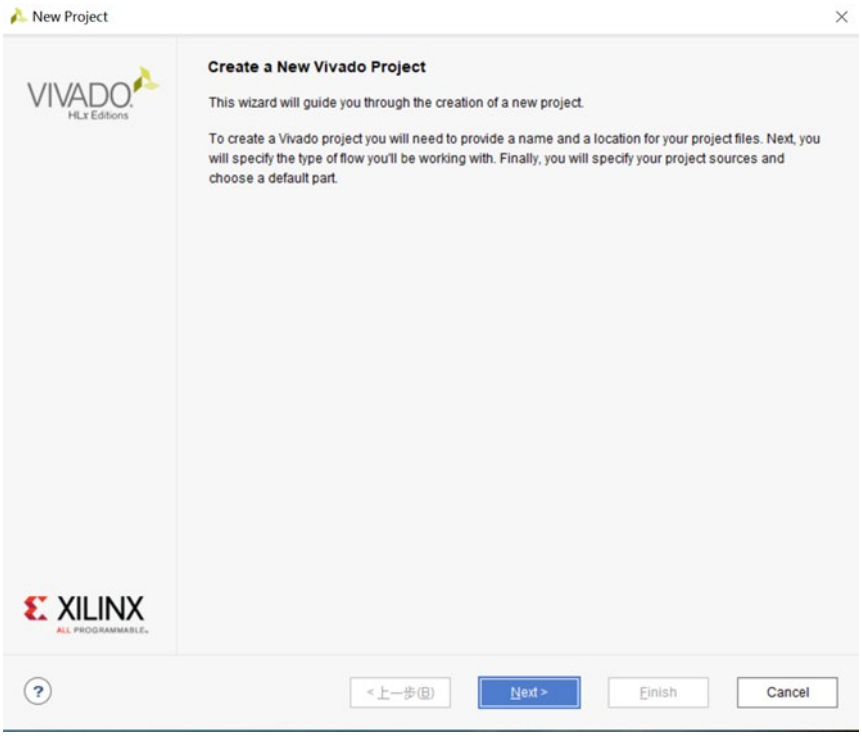


图 2.5.1 New Project

在 Project Name 中勾选  ，如图图 2.5.2 所示：

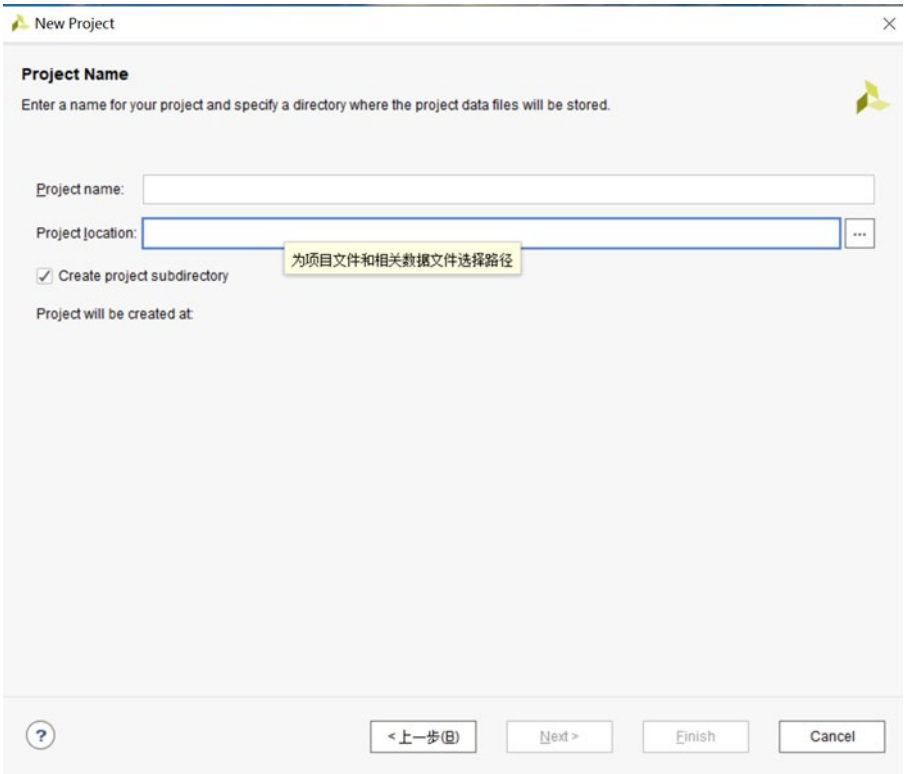


图 2.5.2 Project Name

在 Project Type 中选 RTL Project，如图图 2.5.3

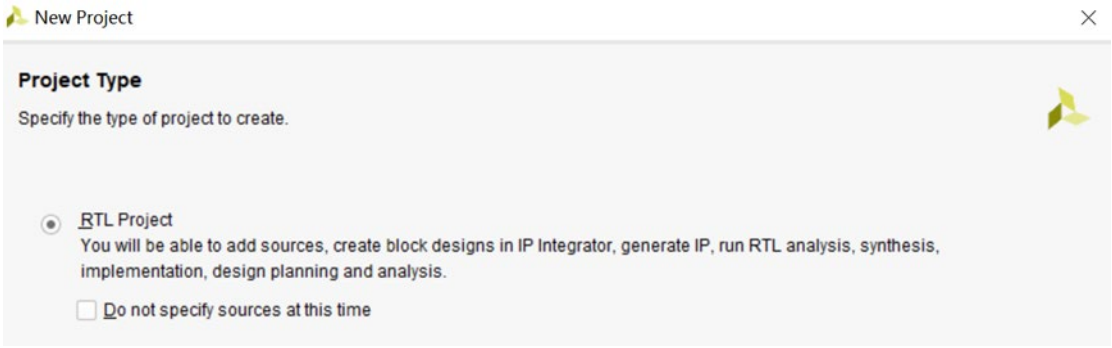


图 2.5.3 Project Type

在 Add Sources 中选 Verilog 作为目标语言，也可以选自己熟悉的其他语言，如图 2.5.4。如果有源文件，选 Add File，我们没有，所以选 Create File，如图 2.5.5。

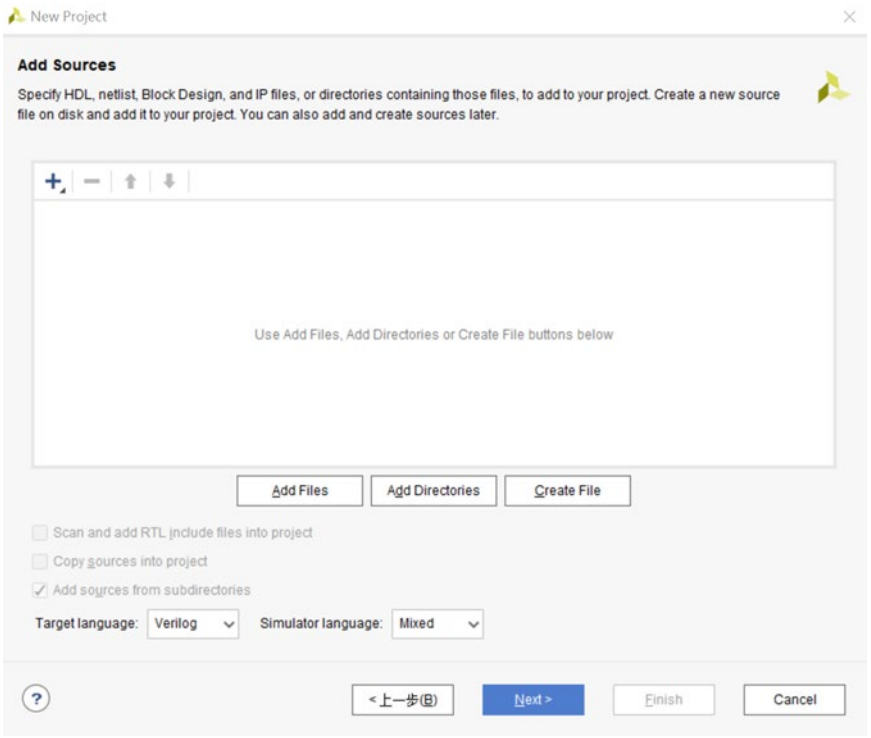


图 2.5.4 Add Sources

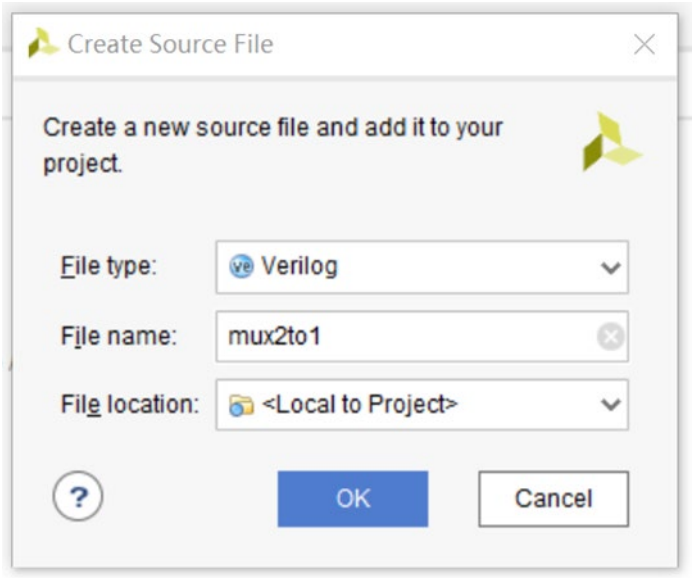


图 2.5.5 Create File

在 Add Constraints 里单击 Next。

在 Default Part 对话框中选择相应的芯片型号或者硬件平台。我们假设使用 EGO1 开发平台，如图 2.5.6

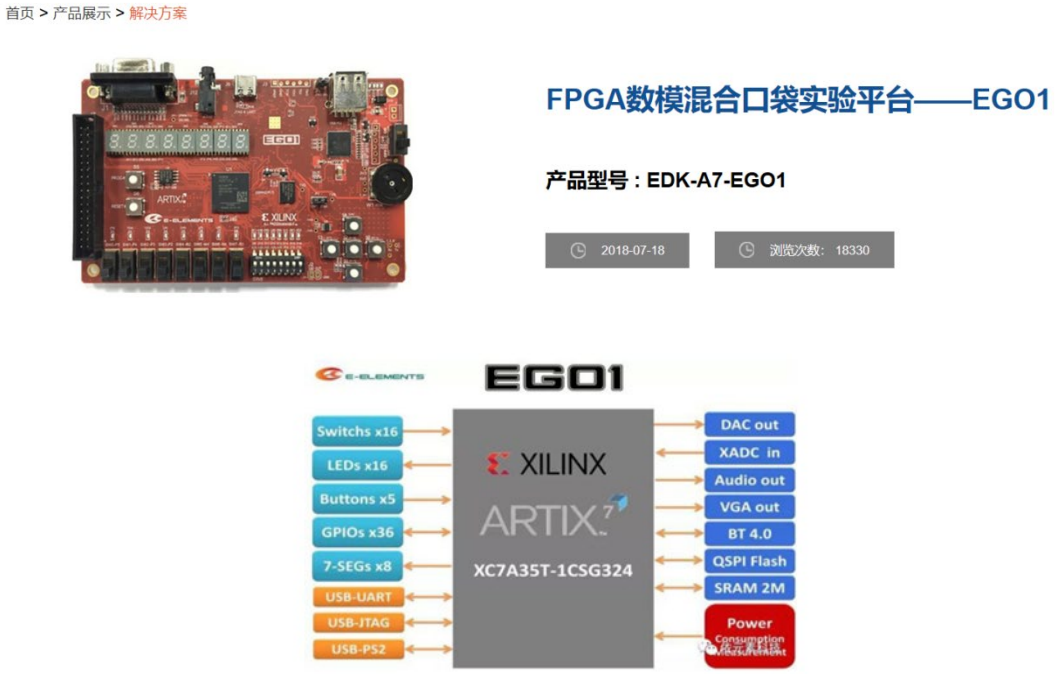


图 2.5.6 EGO1 开发板及芯片规格

具体设置如图图 2.5.7 所示。

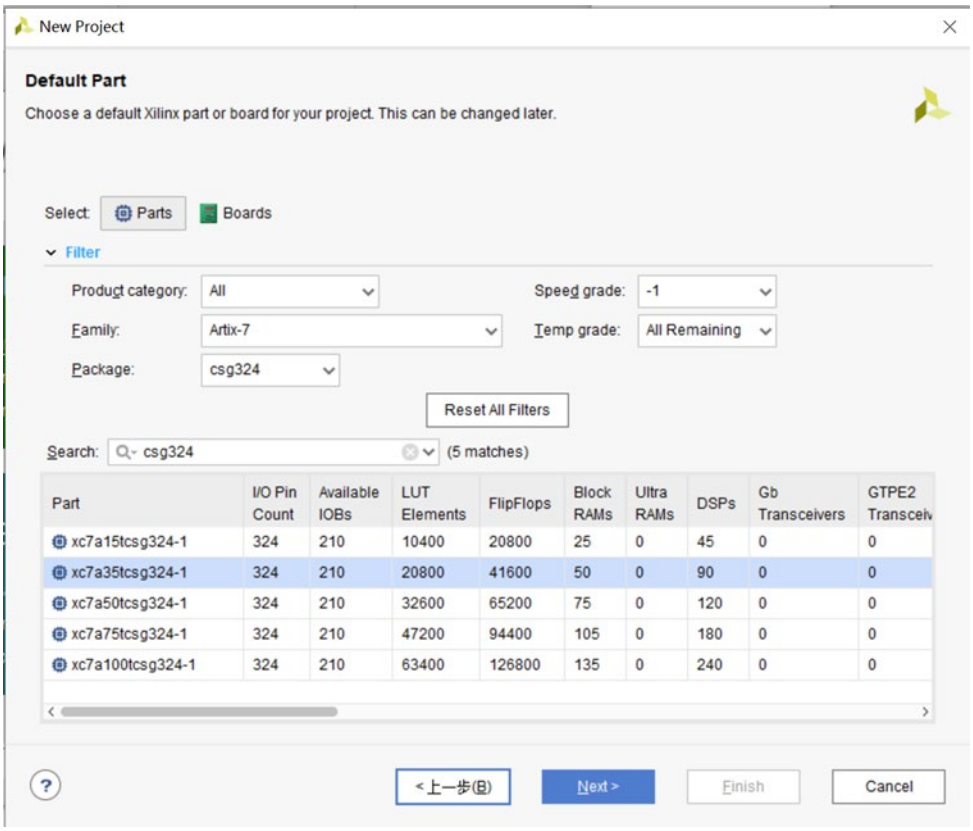


图 2.5.7 Default Part

Define Module 中 Module Name 栏用于输入模块名，下面的列表框用于端口的定义。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，对于单信号的 MSB 和 LSB 不用填写。当然，端口定义这一步也可以略过，在源程序中再行添加。如图 2.5.8

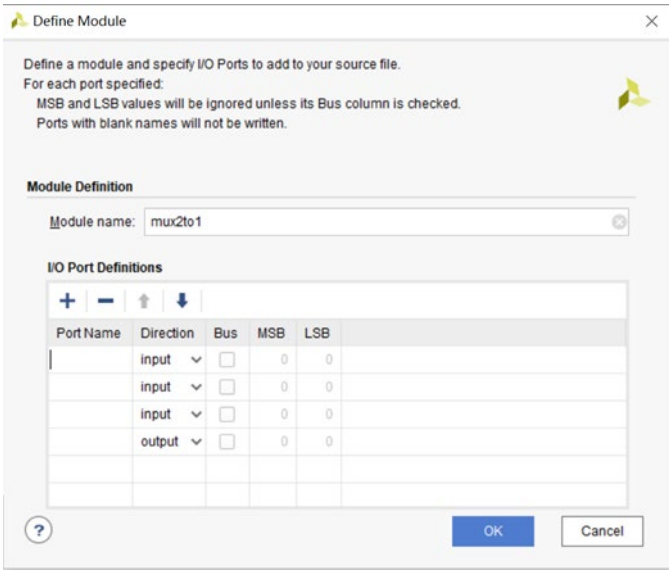


图 2.5.8 Define Module

定义了模块的端口后，单击 Next 进入下一步，点击 Finish 完成创建，进入

主界面。

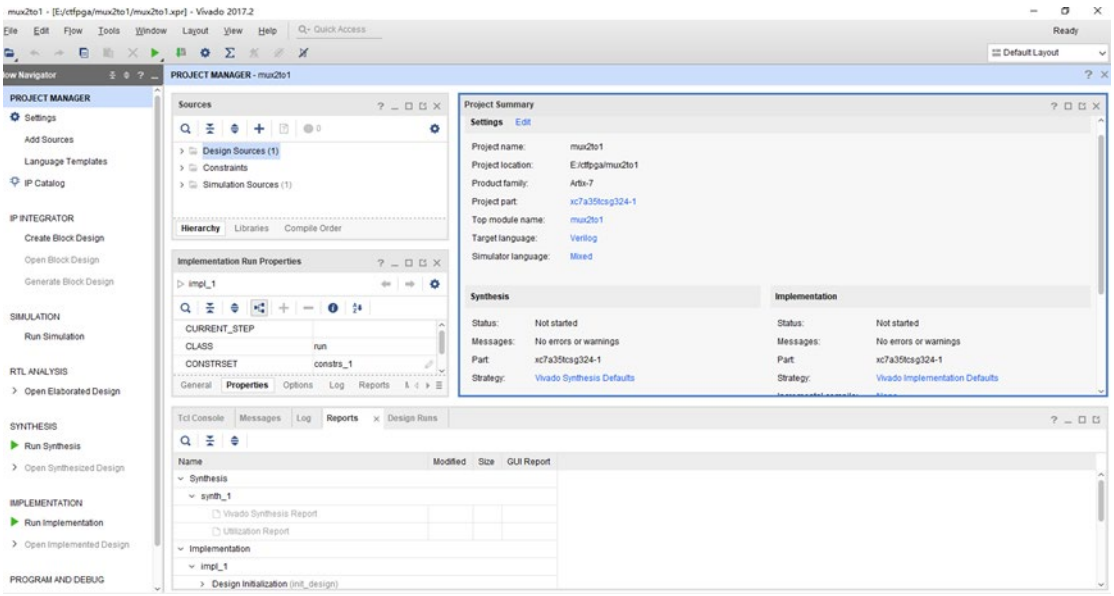


图 2.5.9 Vivado 工程界面

Vivado IDE 主要工作环境包括菜单栏、工具栏、设计流程导航、数据窗口区、工作空间、结果显示区。

(3) 添加或者创建设计文件，如图图 2.5.10 所示

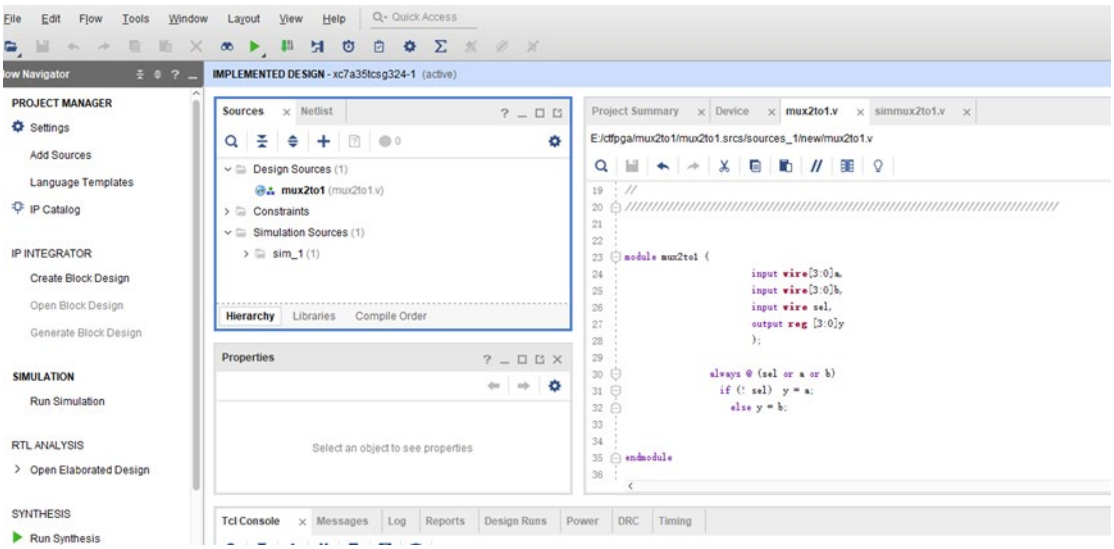


图 2.5.10 工程设计文件输入

2 选 1 的基本代码：

```
module mux2to1 (
                input wire[3:0]a,
                input wire[3:0]b,
                input wire sel,
                output reg [3:0]y
```

```
);

always @ (sel or a or b)
        if (! sel)    y = a;
          else y = b;
endmodule
```

请仿照以上例子自己设计 4 选 1 或 8 选 1 多路选择器。

2.5.2 添加约束文件

对于工程中输入的源代码，需要给设计添加管脚和时序约束。管脚约束是将设计文件的输入输出信号设置到器件的某个管脚，（包括设置此管角的电平标准、电流标准、上下拉特性）。时序约束的作用是为了提高设计的工作频率和获得正确的时序分析报告。

在这个实验中，我们将这一步略去。

2.5.3 综合与实现

综合就是针对输入设计以及约束条件，按照一定的优化算法进行优化处理，获得一个能满足预期功能的电路设计方案。在 FPGA 设计时，可以用硬件描述语言或者是原理图形式来表示电路的功能。综合工具将这些输入文件翻译成由 FPGA 内部逻辑资源（逻辑单元，RAM 存储单元，时钟单元等）按照某种连接方式组成的逻辑连接（网表），并根据用户的要求生成网表文件，这一过程称为综合过程。方法：Flow Navigator→Synthesis→Run Synthesis 进行工程综合。如图 2.5.11

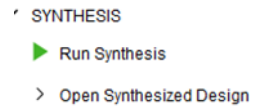


图 2.5.11 综合选项

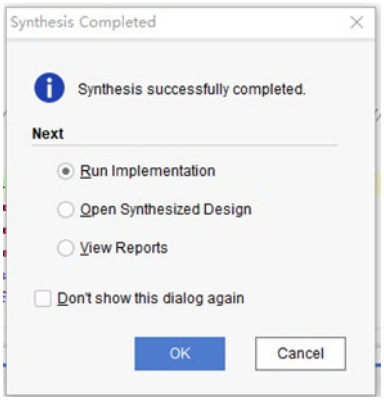


图 2.5.12 综合后的弹出框

实现就是通过翻译、映射、布局布线等过程来完成设计的固化。实现过程首先将综合生成的网表（Netlist）文件,通过翻译变成所选器件的内部资源和硬件单元，如可配置逻辑块（CLB），数字时钟单元（DCM），存储单元（RAM）等，这个步骤称为翻译过程；然后找到对应的硬件关系，将设计与这些硬件资源关系一一对应起来，这又称为映射过程（Map）；最后进行布局布线(Place&Route)，这样设计基本上就可以完全固化到 fpga 当中了。选择 Run Implementation，执行实现过程。

图 2.5.13 实现选项

2.5.4 仿真并查看波形

（1） 添加仿真文件，点图 2.5.14 的“+”

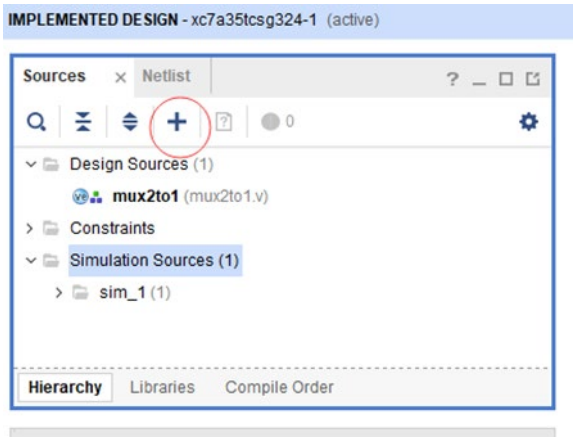


图 2.5.14 添加仿真文件

（2） 设置仿真文件

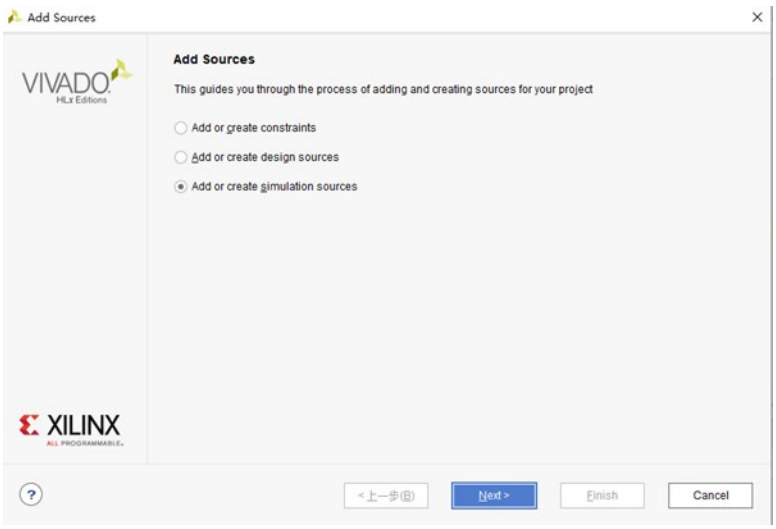


图 2.5.15 Add Source

（3） 建立仿真文件

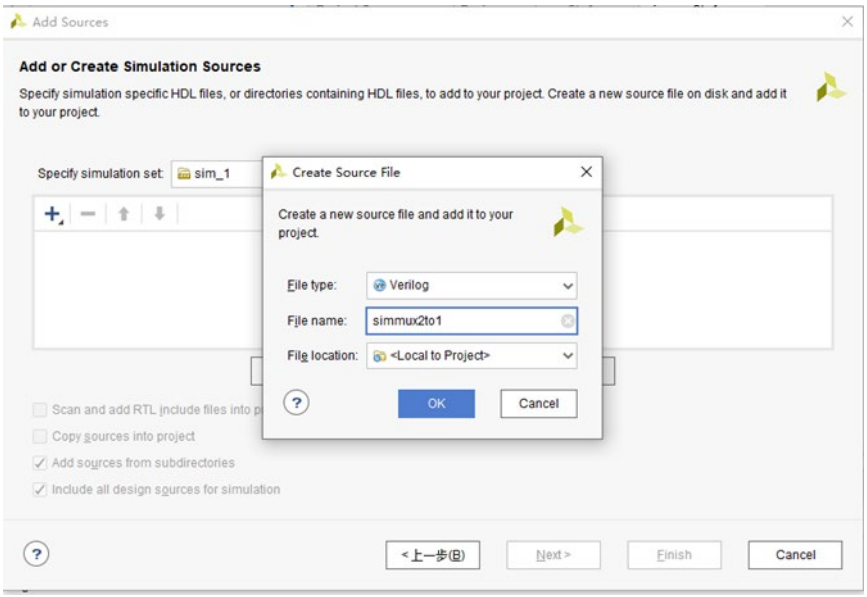


图 2.5.16 建立仿真文件

（4） 添加用于仿真的测试程序

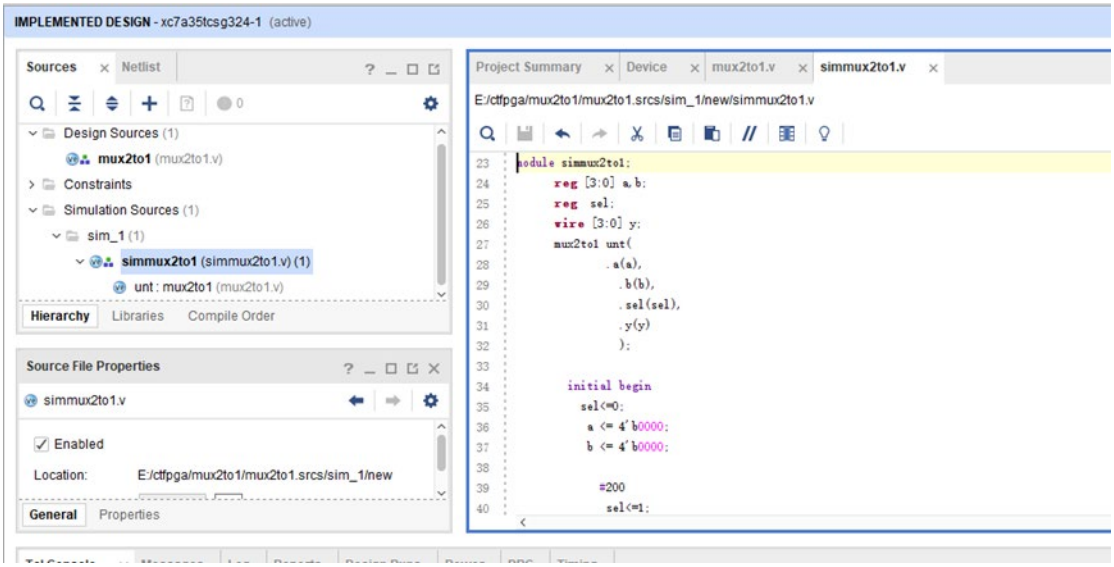


图 2.5.17 编写测试程序

测试代码如下：

```
module simmux2to1;
        reg [3:0] a,b;
        reg    sel;
        wire [3:0] y;
        mux2to1 unt(
                    .a(a),
                       .b(b),
                       .sel(sel),
                       .y(y)
                       );

         initial    begin
              sel<=0;
                a <= 4'b0000;
                b <= 4'b0000;

                   #200
                     sel<=1;
                     a <= 4'b0001;
                     b <= 4'b1000;

                      #200
                      sel<=0;
                      a <= 4'b0010;
```

```
        b <= 4'b0100;

        #200
          sel<=1;
          a <= 4'b0011;
          b <= 4'b1100;

    end

endmodule
```

（5） 仿真

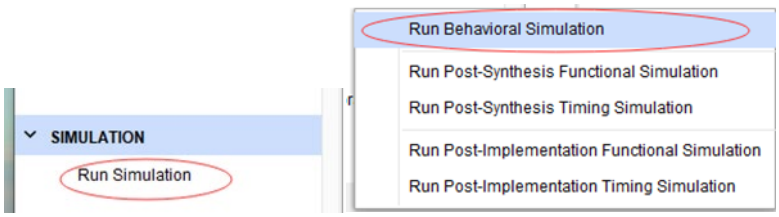
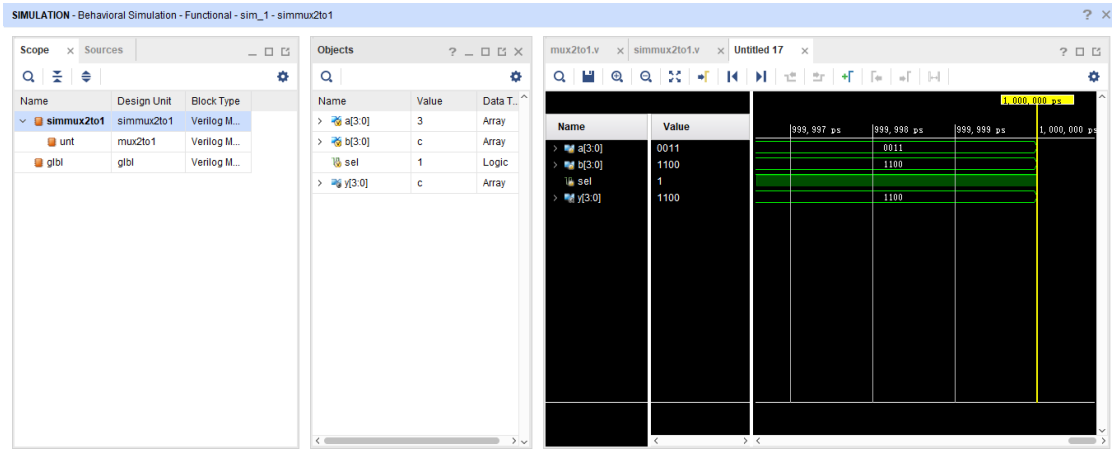


图 2.5.18 仿真

（6）分析波形

根据波形中的逻辑值分析是否符合逻辑设计的功能。



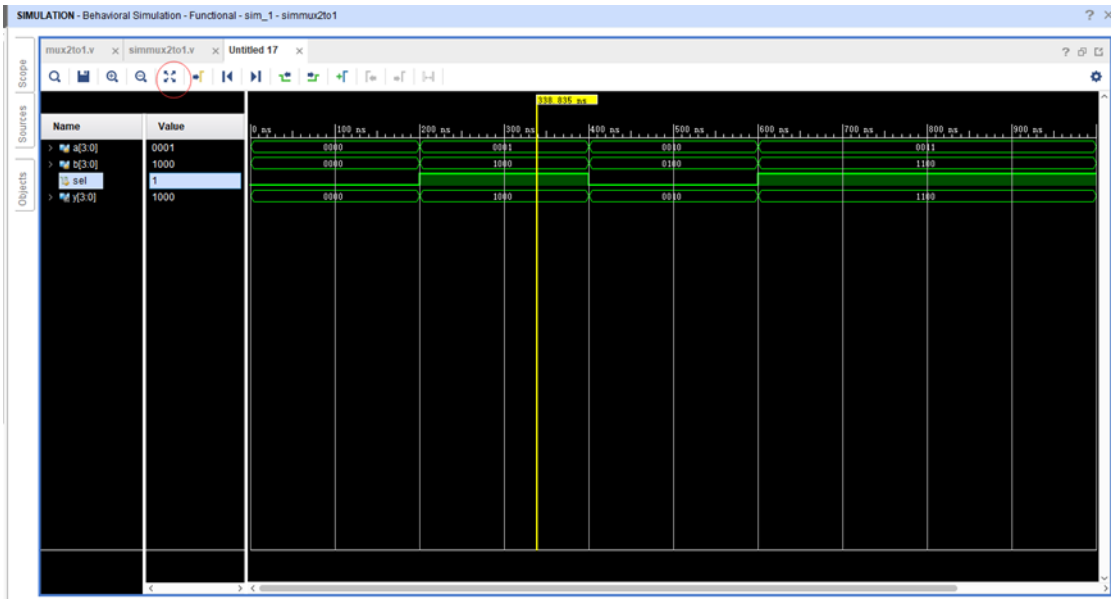


图 2.5.19 分析波形

查看原理图，如图图 2.5.20 所示。

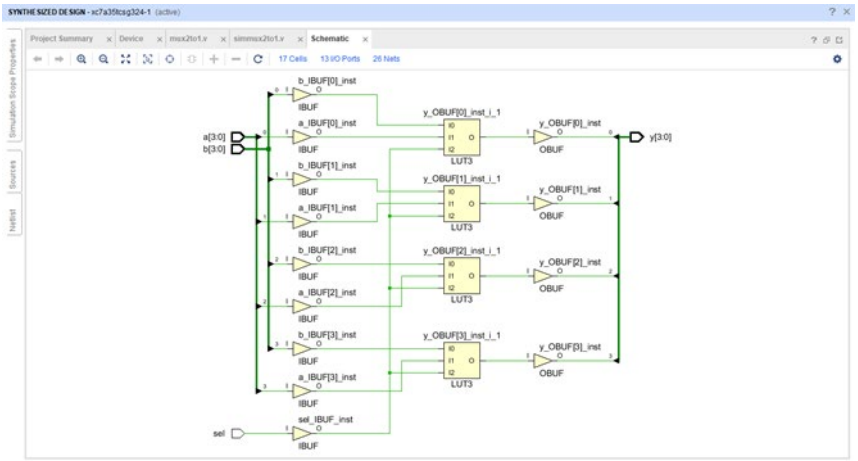


图 2.5.20 原理图

2.5.5 编译文件及下载

只有编译文件才能配置 FPGA，因此在完成综合之后，还要将生成的网表文件转换成可配置的 FPGA 文件。在实现过程完成后，用户可以查看设计实现的结果或者实现报告。如果需要继续进行生成编译时可选择生成 bitstream，点击 OK，执行编译过程。（在这个实验中因为没有做约束，也没有开发板用于下载，因此这一步可以跳过。）

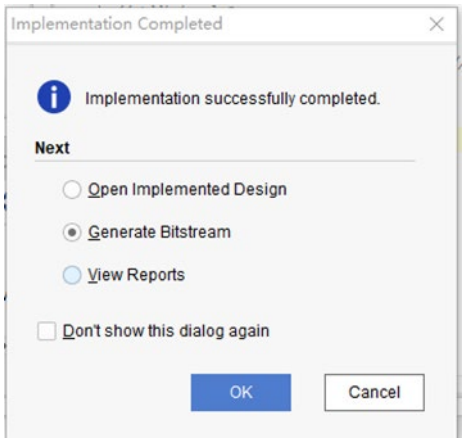

Implementation Completed
Implementation successfully completed.
Next
Open Implemented Design
Generate Bitstream
View Reports
Don't show this dialog again
OK
Cancel

图 2.5.21 弹窗

实验三　CPU 部件实现之 ALU 和寄存器堆

3.1 实验内容

理解和掌握 CPU 中的算术逻辑运算部件（ALU）和寄存器堆（Register File）的工作原理，并使用 Verilog 和 Vivado 进行设计和仿真。

3.2 实验目标

1. 使用 Verilog 完成 ALU 的设计,并编写测试仿真文件验证其正确性。要求：

- ALU 支持 16 位的加、减、与、或以及移位运算。

2. 使用 Verilog 完成通用寄存器堆的设计，并编写测试仿真文件验证其正确性。要求

- 寄存器堆包含 8 个 16 位的寄存器；
- 寄存器堆有两个读端口和一个写端口。

3.3 实验原理

（请同学们自行编写实验原理）

3.4 实验步骤

（请同学们自行补充实验步骤）

3.4.1 编写 ALU 源文件

(1) 建立 Verilog PC 源文件。点击“+”号（图 3.4.1），选择“Add or create design sources”（图 3.4.2）。

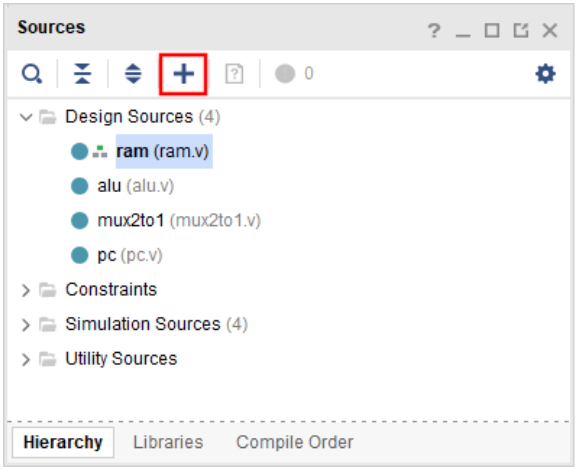


图 3.4.1 创建 alu 文件

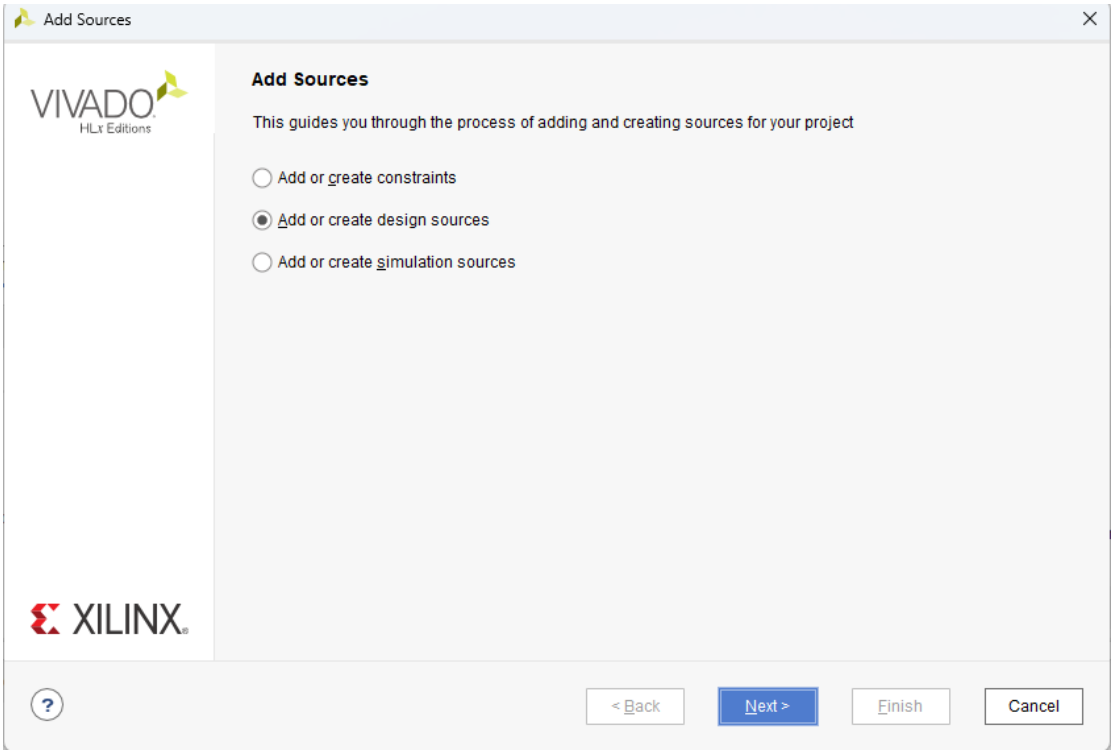


图 3.4.2 选择创建的文件类型

(2) 点击“Create File”（图 3.4.3），file name 命名为“alu”并选择“OK”（图 3.4.4）。然后选择 Finish，创建文件结束。

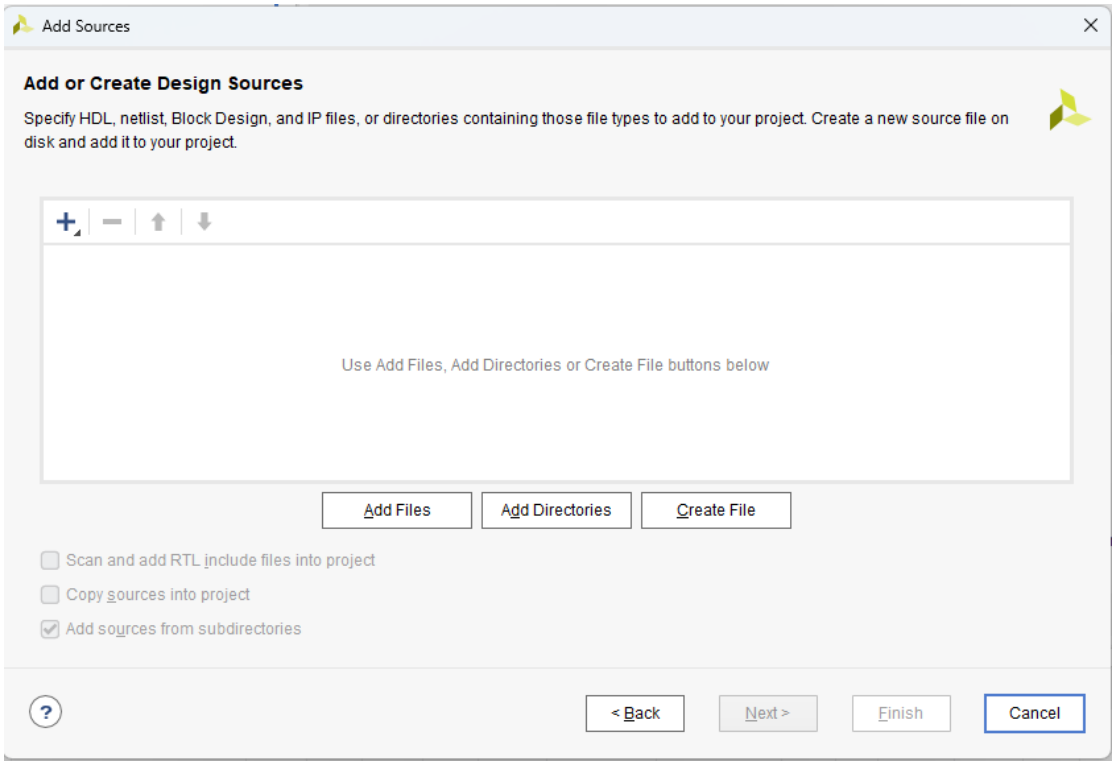


图 3.4.3 Create File 界面

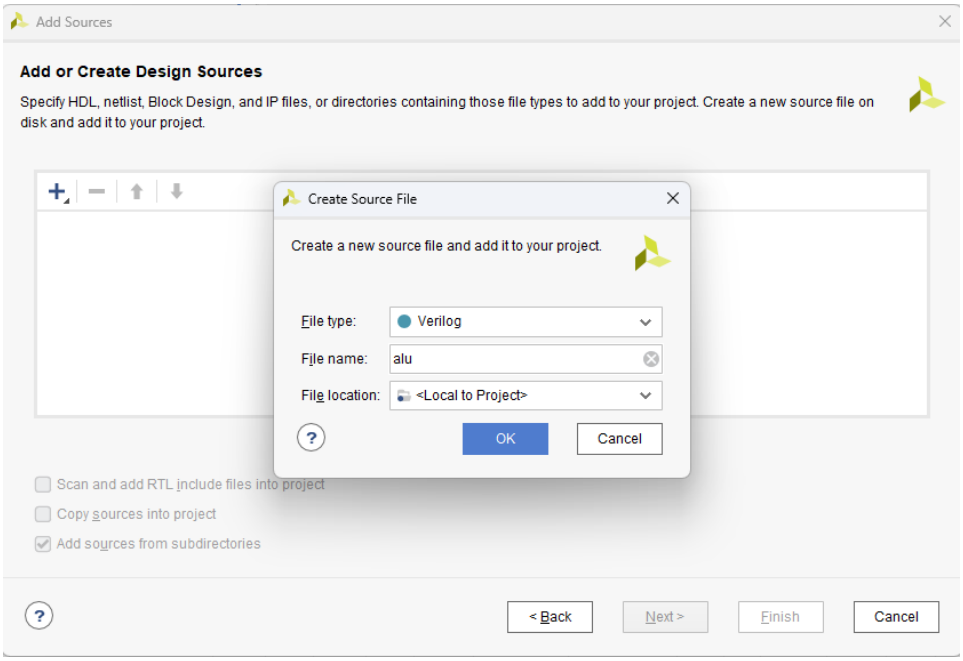


图 3.4.4 Create alu source file

(3) 在弹出的“Define Module”中（图 3.4.5），创建三个 input，一个 output，然后选择 ok（此步也可以不做）。

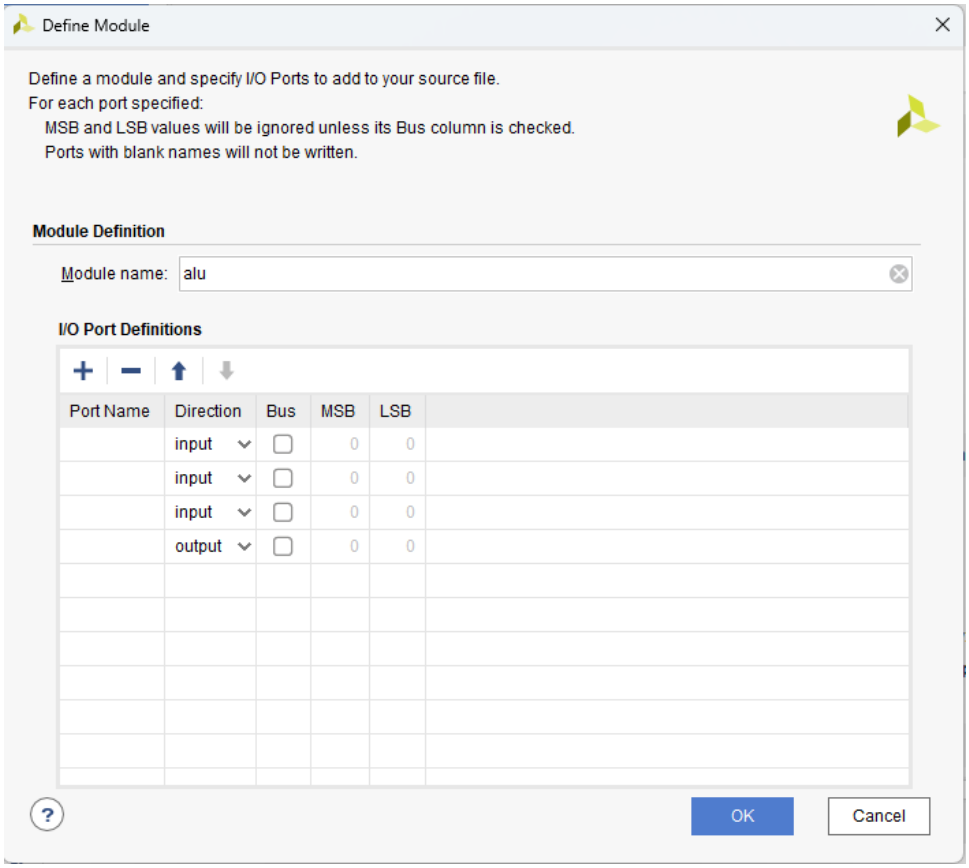


图 3.4.5 Define module

(4) 在 Design Sources 下选中并双击刚才建立的“alu”源文件，然后编写 PC 的 Verilog 源代码。代码如下：

```
module alu(
    input wire [15:0] in1, //输入的数据1
    input wire [15:0] in2, //输入的数据2
    input wire [2:0] alu_op, //操作码
    output reg [15:0] result //输出
    );

    parameter //定义操作码
    ADD = 3'b000, // 加法
    SUB = 3'b001, //减法
    AND = 3'b010, //逻辑与
    OR  = 3'b011, //逻辑或
    SLL = 3'b100, //逻辑左移
    SRL = 3'b101, //逻辑右移
    SLA = 3'b110, //算数左移
    SRA = 3'b111; //算数右移

    always @(*)
    begin
        case(alu_op)
            ADD: result = in1 + in2;
            SUB: result = in1 - in2;
            AND: result = in1 && in2;
            OR:  result = in1 || in2;
            SLL: result = in1 << in2;
            SRL: result = in1 >> in2;
            SLA: result = in1 <<< in2;
            SRA: result = in1 >>> in2;
            default: result = 5;
        endcase
    end

endmodule
```

3.4.2 仿真 ALU 并查看波形

（1） 点击“+”添加仿真文件（图 3.4.6）。

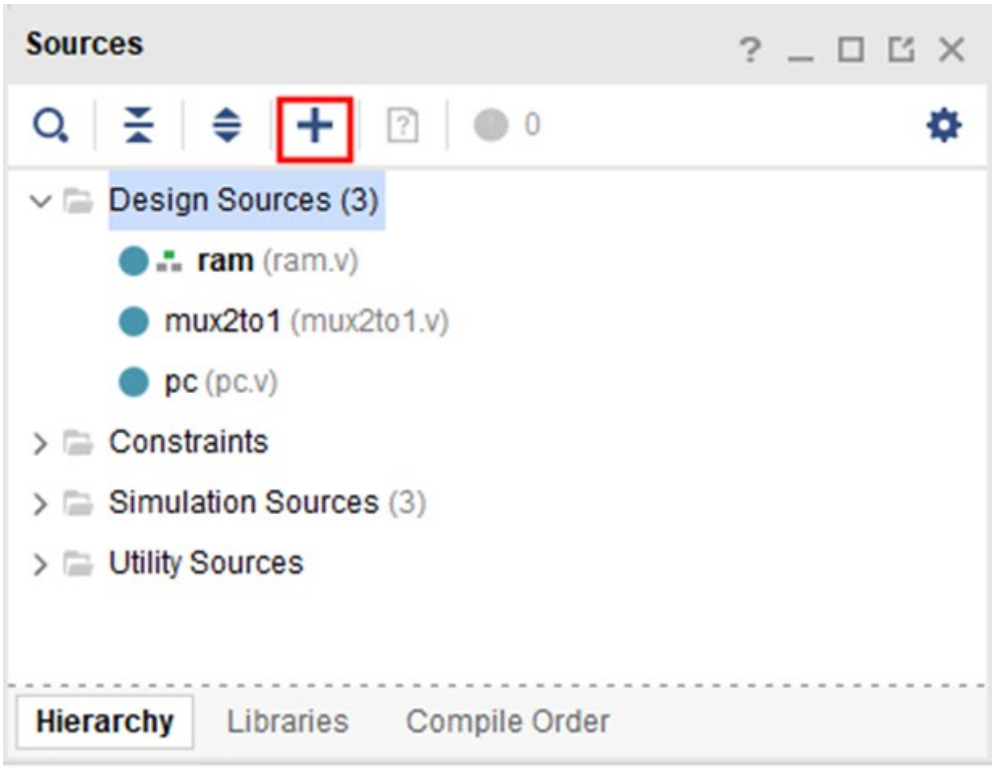


图 3.4.6 添加 alu 仿真文件

（2） 选择生成或添加仿真文件（见图 3.4.7）。

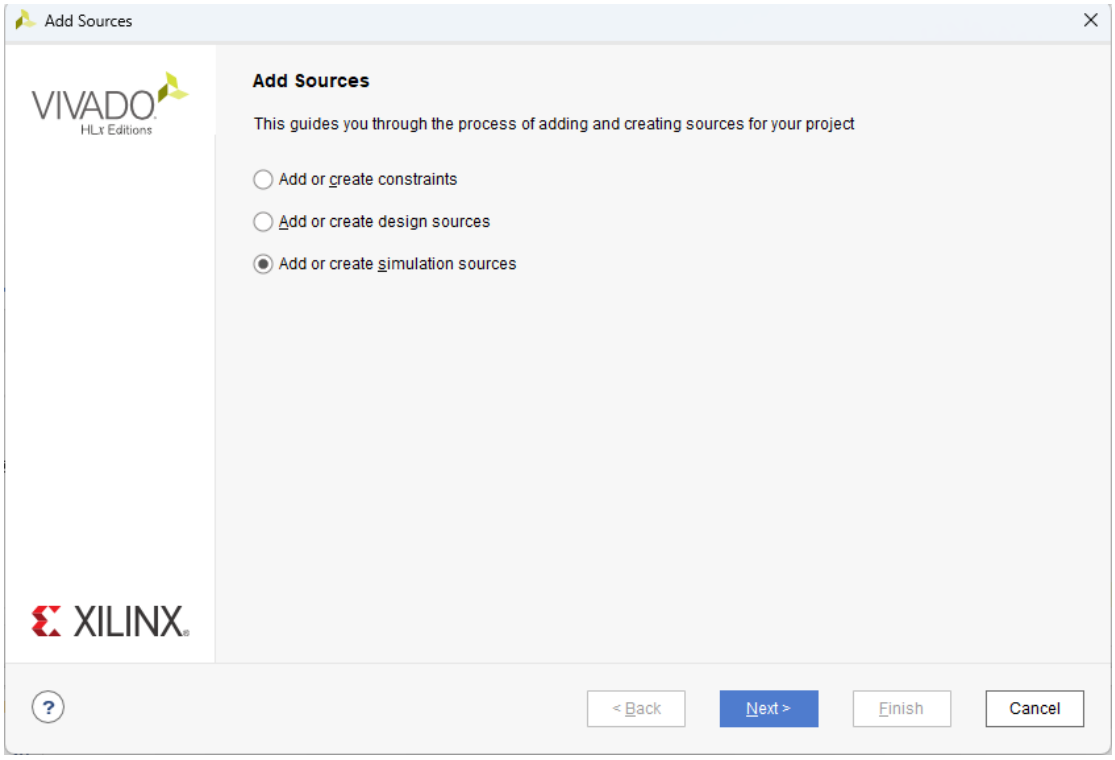


图 3.4.7 设置 alu 仿真文件

（3） 建立仿真文件（图 3.4.8）。

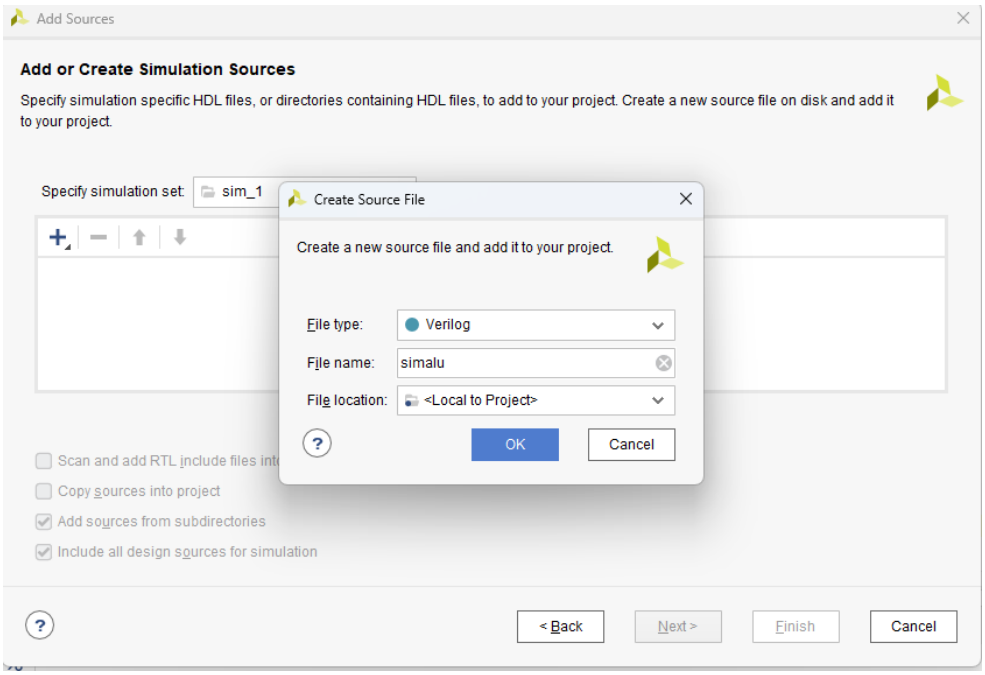


图 3.4.8 建立 simalu 仿真文件

（4） 添加用于仿真的测试程序，代码如下：

```
module simalu(
    );
    reg [15:0] in1, in2;
    reg [2:0] alu_op;
    wire [15:0] result;

    alu dut(
        .in1(in1),
        .in2(in2),
        .alu_op(alu_op),
        .result(result)
    );

    parameter
    ADD = 3'b000, //加法
    SUB = 3'b001, //减法
    AND = 3'b010, //逻辑与
    OR  = 3'b011, //逻辑或
    SLL = 3'b100, //逻辑左移
    SRL = 3'b101, //逻辑右移
    SLA = 3'b110, //算数左移
    SRA = 3'b111; //算数右移
```

```
    initial begin
        in1 = 16'h0020; //二进制0000_0000_0010_0000
        in2 = 16'h0001; //二进制0000_0000_0000_0001

        //测试加法
        alu_op = ADD;
        #10 $display("in1 + in2 = %b",result);

        //测试减法
        alu_op = SUB;
        #10$display("in1 - in2 = %b", result);

        //测试逻辑与
        alu_op = AND;
        #10$display("in1 && in2 = %b", result); //如果两个数都不为0，则结果为1

        //测试逻辑或
        alu_op = OR;
        #10$display("in1 || in2 = %b", result);

        //测试逻辑左移
        alu_op = SLL;
        #10$display("in1 << in2 = %b", result);

        //测试逻辑右移
        alu_op = SRL;
        #10$display("in1 >> in2 = %b", result);

        //测试算数左移
        alu_op = SLA;
        #10$display("in1 <<< in2 = %b", result);

        //测试算数右移
        alu_op = SRA;
        #10$display("in1 >>》  in2 = %b", result);

        $finish;
    end
endmodule
```

（5） 对 simalu 进行仿真（图 3.4.9）。

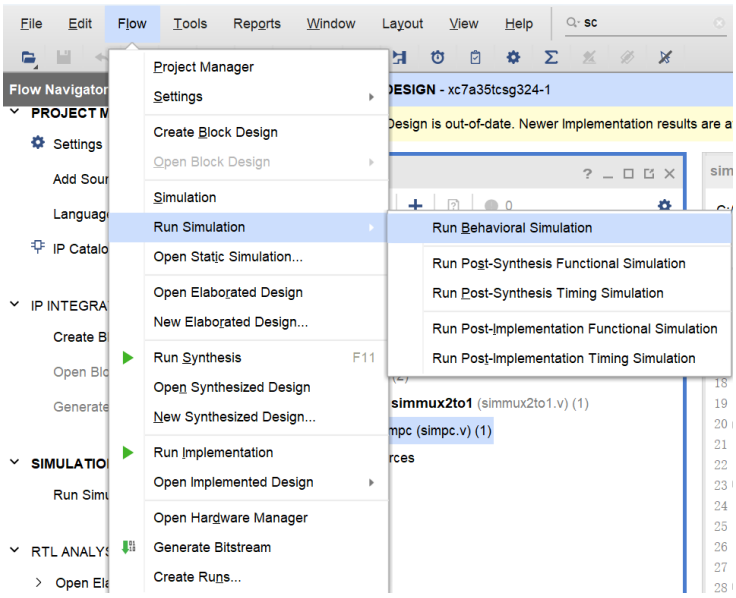


图 3.4.9 对 simalu 进行仿真

（6） 分析波形（图 3.4.10）。

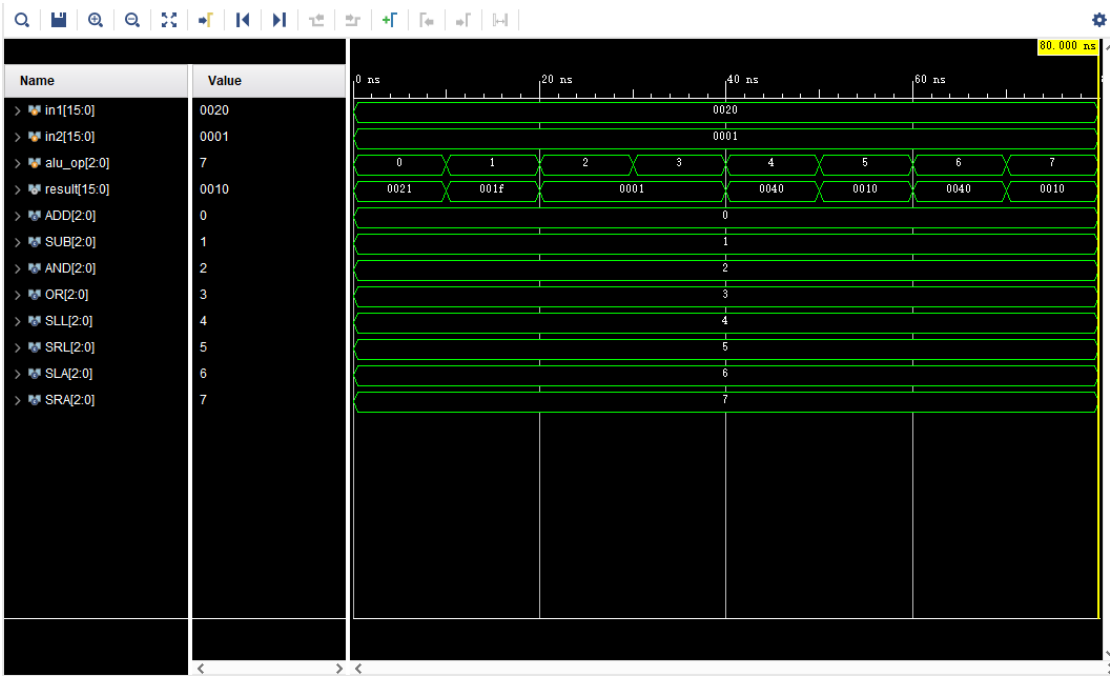


图 3.4.10 simalu 的仿真波形图

3.4.3 编写寄存器堆源文件

(1) 建立 Verilog 寄存器源文件。点击“+”号（图 3.4.11），选择“Add or create design sources”（图 3.4.12）。

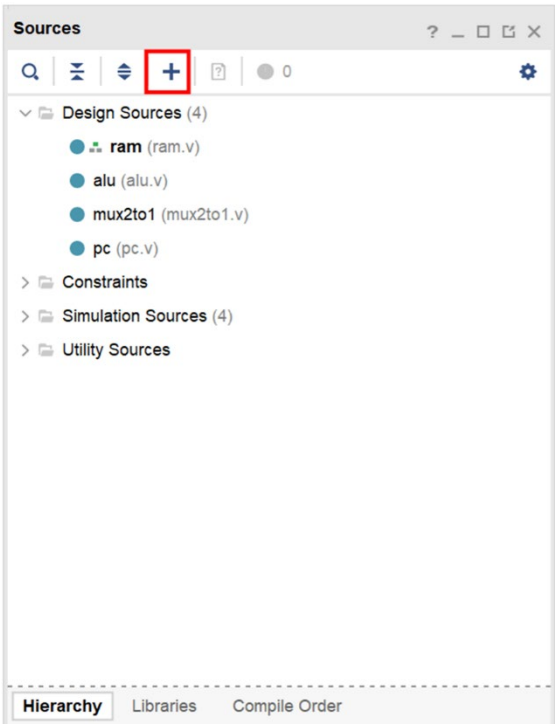


图 3.4.11 创建 register 文件

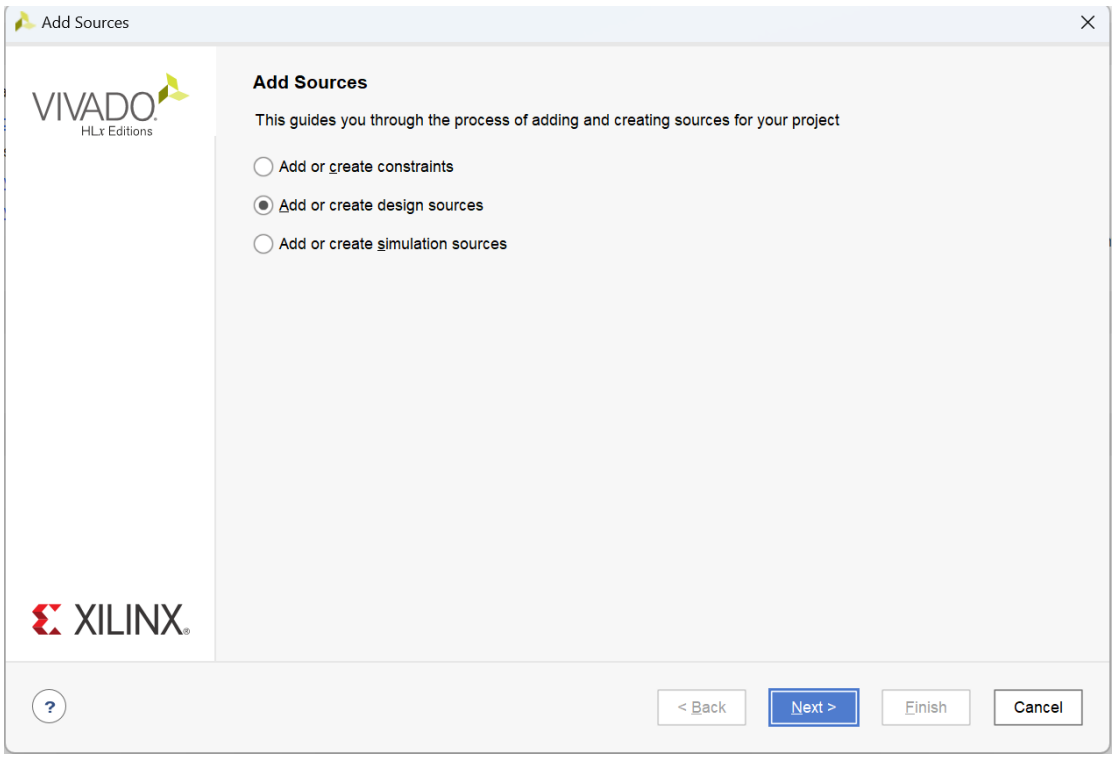


图 3.4.12 Add or create design sources

(2) 选择“Create File”（图 3.4.13），file name 命名为“register”并选择“OK”（图 3.4.14）。然后选择 Finish，创建文件结束。

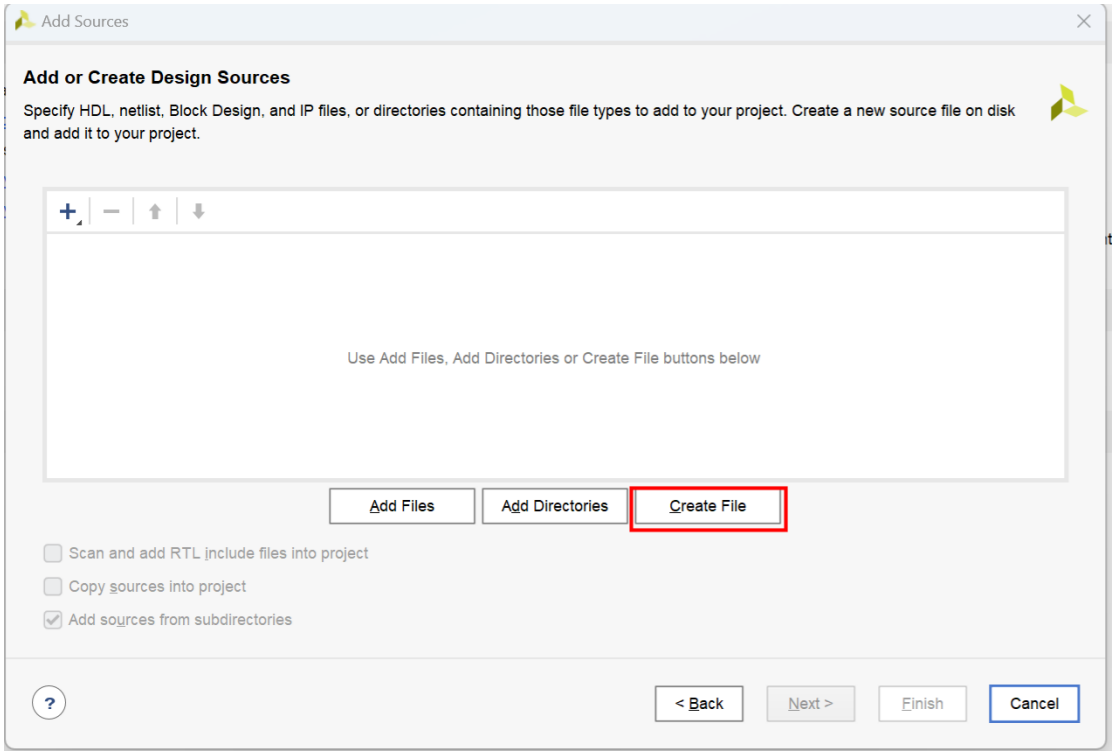


图 3.4.13 点击 Create File

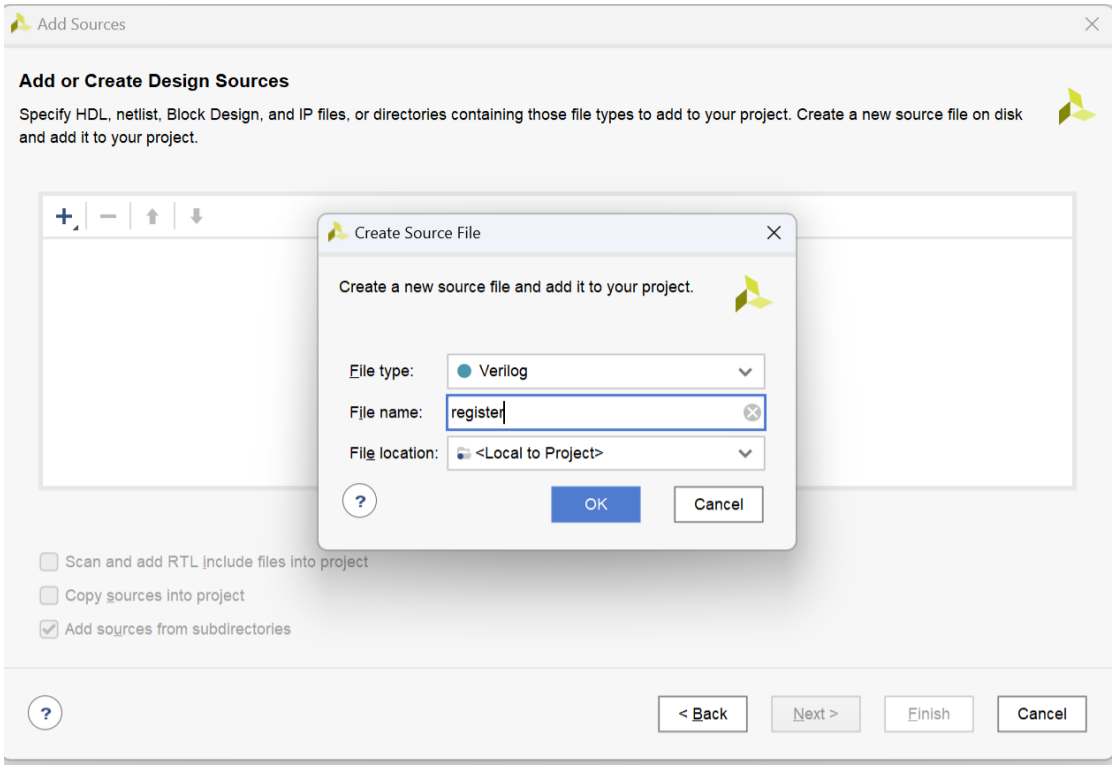


图 3.4.14 Create register source file

(3) 在弹出的“Define Module”中（图 3.4.15），创建三个 input，一个 output，然后选择 ok（此步也可以不做）。

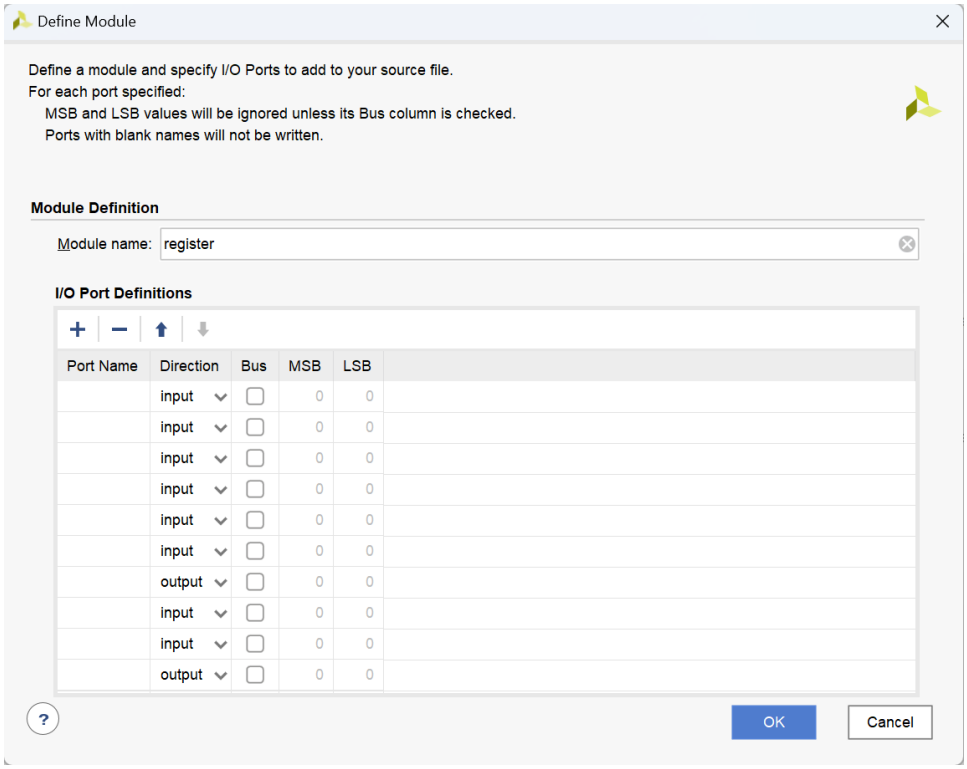


图 3.4.15 Define Module

(4) 在 Design Sources 下选中并双击刚才建立的“register”源文件，然后编写 register 的 Verilog 源代码。代码如下

```
module register(
    input wire clk,

    //写端口
    input wire we,
    input wire[2:0] waddr,
    input wire[15:0] wdata,

    //读端口1
    input wire  re1,
    input wire[2:0] raddr1,
    output reg[15:0] rdata1,

    //读端口2
    input wire re2,
    input wire[2:0]  raddr2,
    output reg[15:0] rdata2
    );
```

```
    reg[15:0]  regs[7:0]; //8个16位寄存器

    always @ (posedge clk) begin
        if(we == 0) begin
            regs[waddr] <= wdata;
        end
        if ((raddr1 == waddr) && (re1 ==0)) begin
           //如果第一个读寄存器端口要读取的目标寄存器
//            与要写入的目的寄存器是同一个寄存器，
//            那么直接将要写入的值作为第一个读寄存器端口的输出
           rdata1 <= wdata;
        end
        if((raddr2 == waddr) && (re2 == 0)) begin
          //如果第二个读寄存器端口要读取的目标寄存器
//        与要写入的目的寄存器是同一个寄存器，那么
          //直接将要写入的值作为第二个读寄存器端口的输出
          rdata2 <= wdata;
        end
    end

    always @ (*) begin
          if(re1 == 0) begin
          rdata1 <= regs[raddr1];
      end else begin
          //如果第一个读寄存器端口不能使用，直接输出0
          rdata1 <= 16'h0000;
      end
    end

    always @ (*) begin
          if(re2 == 0) begin
          rdata2 <= regs[raddr2];
      end else begin
          //如果第二个读寄存器端口不能使用，直接输出0
          rdata2 <= 16'h0000;
      end
    end

endmodule
```

5.1 仿真寄存器堆并查看波形

（1） 点击“+”添加“register”的仿真文件（图 3.4.16）。

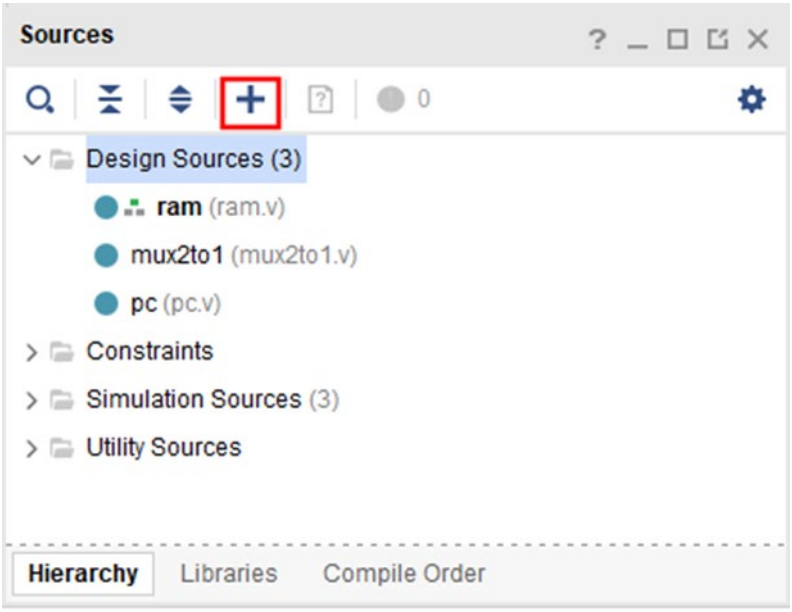


图 3.4.16 添加 register 的仿真文件

（2） 设置仿真文件（图 3.4.17）。

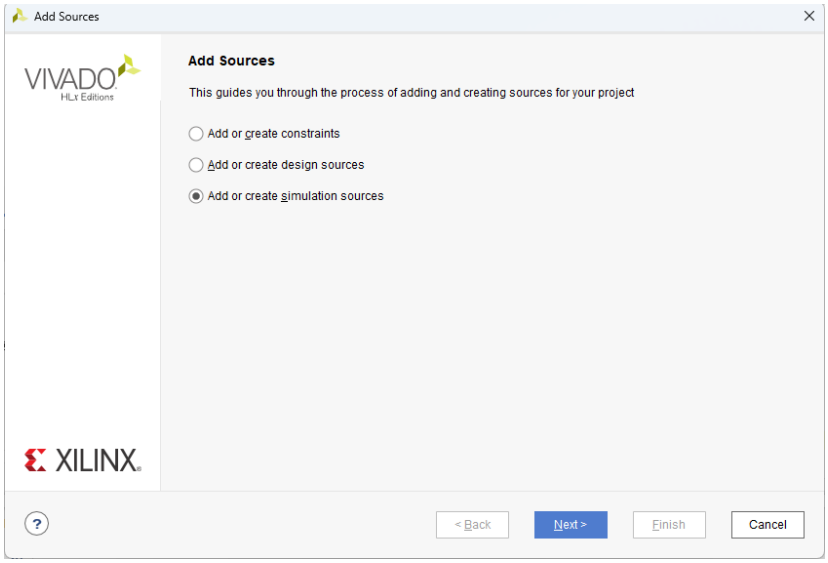


图 3.4.17 设置 register 的仿真文件

（3） 建立“register”仿真文件（图 3.4.18）。

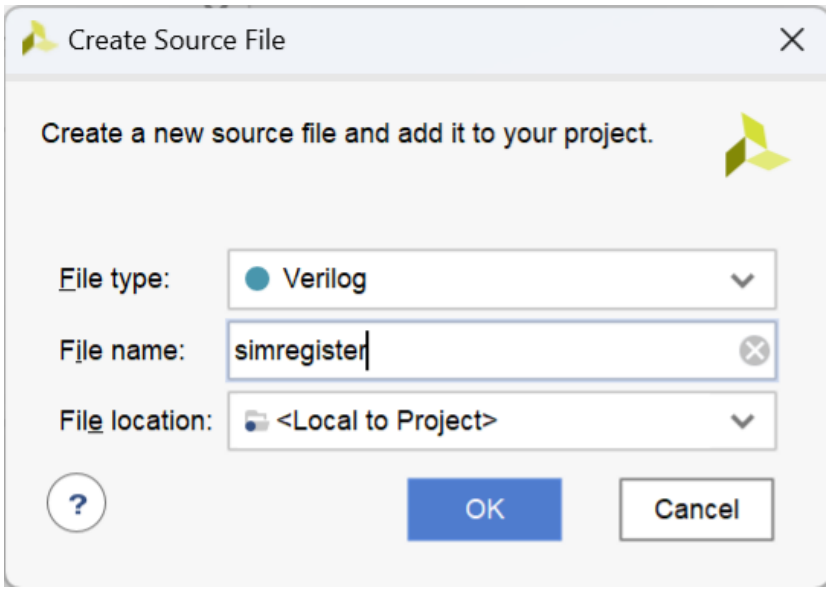


图 3.4.18 建立 register 的仿真文件

（4） 添加“register”用于仿真的测试程序，代码如下：

```
module simregister( );
    reg clk;

    reg we;
    reg[2:0] waddr;
    reg[15:0] wdata;

    reg re1;
    reg[2:0] raddr1;
    wire[15:0] rdata1;

    reg re2;
    reg[2:0] raddr2;
    wire[15:0] rdata2;

    register dut(
       .clk(clk),
       .we(we),
       .waddr(waddr),
       .wdata(wdata),
       .re1(re1),
       .raddr1(raddr1),
       .rdata1(rdata1),
       .re2(re2),
       .raddr2(raddr2),
       .rdata2(rdata2)
    );
```

```
    initial begin
        clk = 1; //初始时时钟上升沿
        //时间0向寄存器0写入数据2222H，
        //并从读端口1读出此数据
        we = 0;
        re1 = 0;
        re2 = 1;
        waddr = 3'b000;
        wdata = 16'h2222;
        raddr1= 3'b000;
        #5;
        //时间5试图向寄存器1中写入数据1111H，
        //并从读端口2读出此数据
        //由于时刻5时钟是下降沿，
        //即使we有效（we=0）也不能写入，
        //源代码里面貌似有问题，没有检查时钟上升沿？
        we=0;
        re1=1;
        re2 = 0;
        waddr = 3'b001;
        wdata = 16'h1111;
        raddr2 = 3'b001;
        #5;
        //在时间10向寄存器1中写入数据1111H，
        //由于时间10时钟是上升沿，可以写入，
        //并可以从读端口2读出此数据
        we=0;
        re1=1;
        re2 = 0;
        waddr = 3'b001;
        wdata = 16'h1111;
        raddr2 = 3'b001;
        #5;
        //在时间15从读端口1读出寄存器0中的数据，
        //并同时从读端口2读出寄存器1中的数据
        we=1;
        re1=0;
        re2 = 0;
        raddr1 = 3'b000;
        raddr2 = 3'b001;
        #5;$finish;
    end
    always #5 clk = ~clk; // 时钟信号产生器
endmodule
```

（5） 仿真（图 3.4.19）。

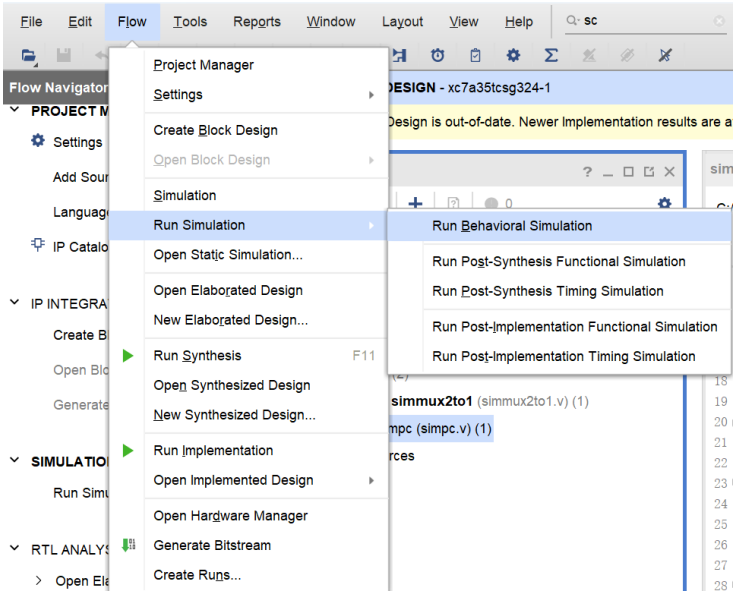


图 3.4.19 register 仿真

（6） 分析波形（图 3.4.20）

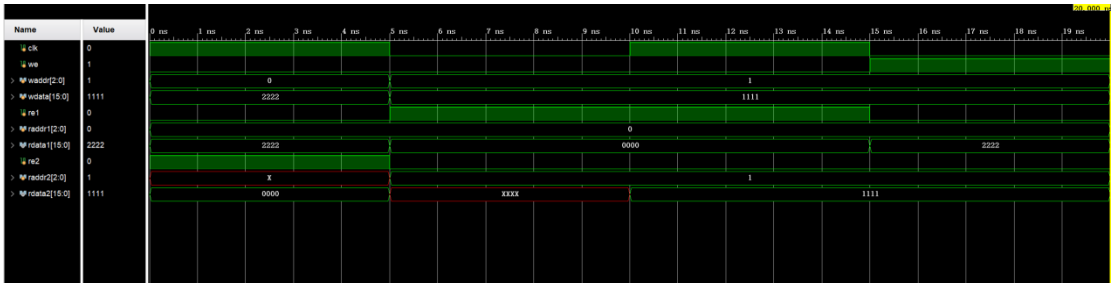


图 3.4.20 register 的仿真波形图

这里要注意一下，如果同一个项目里有多个模块，又没有设置顶层文件，在做仿真的时候，需要把对应的仿真文件设置为顶层文件再仿真。

实验四　实验四 CPU 部件实现之 PC 和半导体存储器

RAM

4.1 实验内容

理解和掌握 CPU 中程序计数器 PC 和半导体存储器 RAM 的工作原理，并使用 Verilog 和 Vivado 进行设计和仿真。

4.2 实验目标

1. 使用Verilog完成程序计数器PC的设计，要求：
   - PC为8位计数器。
2. 使用Verilog 完成数据存储器的设计，并编写测试仿真文件验证其正确性。要求：
   - 存储字长为16位，存储容量为1K字节；
   - 一根读写控制信号线控制读写，低电平有效。

4.3 实验原理

（请同学们自行编写实验原理）

4.4 实验步骤

（请同学们自行补充实验步骤）

4.4.1 编写 PC 源文件

(1) 建立 Verilog PC 源文件。点击“+”号（图 4.4.1），选择“Add or create

design sources”（图 4.4.2）。

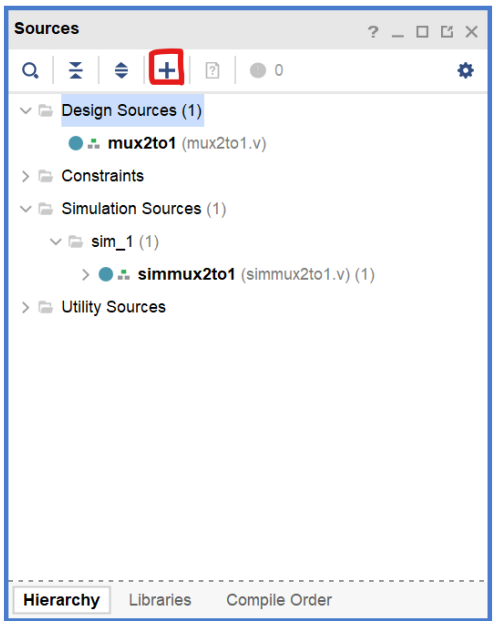


图 4.4.1 创建源文件

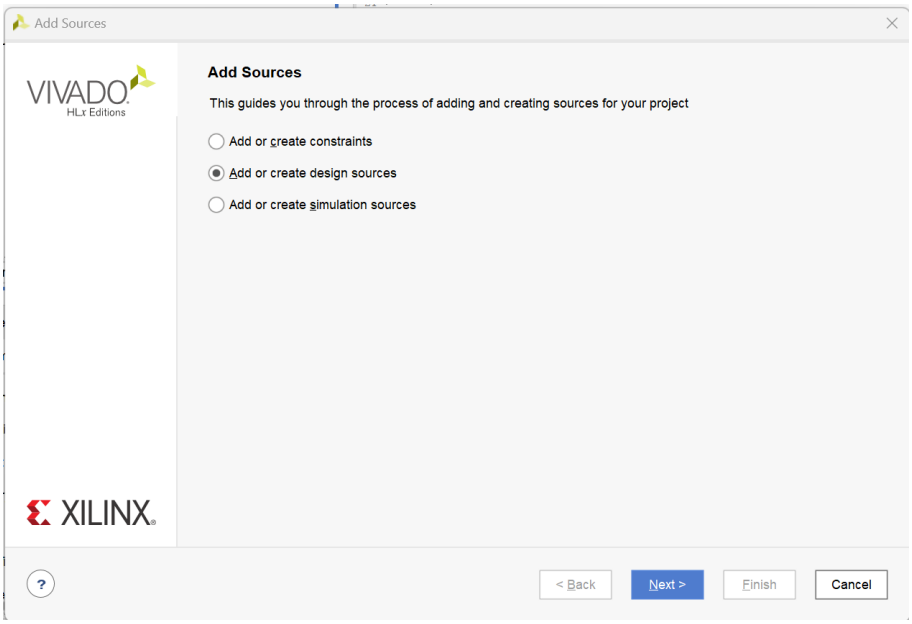


图 4.4.2 Add or create design sources

(2) 选择“Create File”（图 4.4.3），file name 命名为“pc”并选择“OK”（图 4.4.4）。然后选择 Finish，创建文件结束。

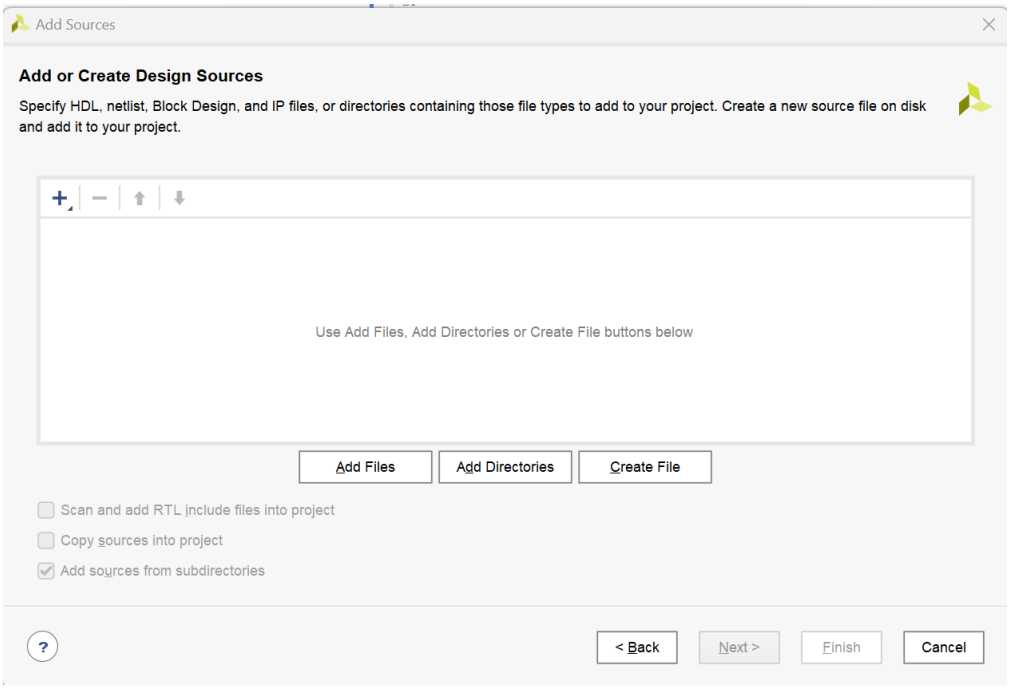


图 4.4.3 Create file

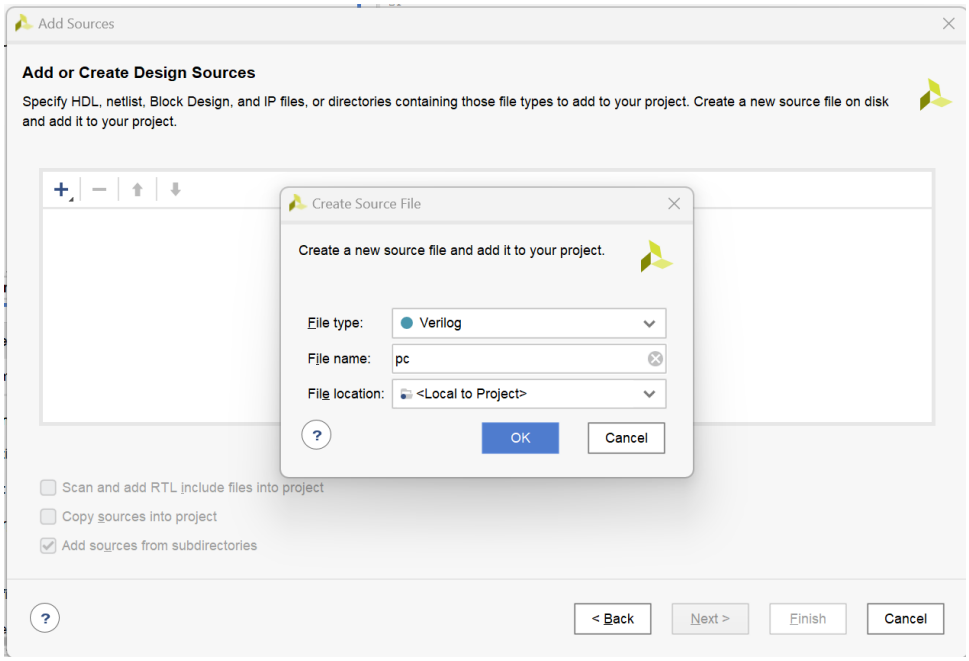


图 4.4.4 创建 pc 源文件

(3) 在弹出的“Define Module”中（图 4.4.5），创建两个 input，一个 output，然后选择 ok（此步也可以不做）。

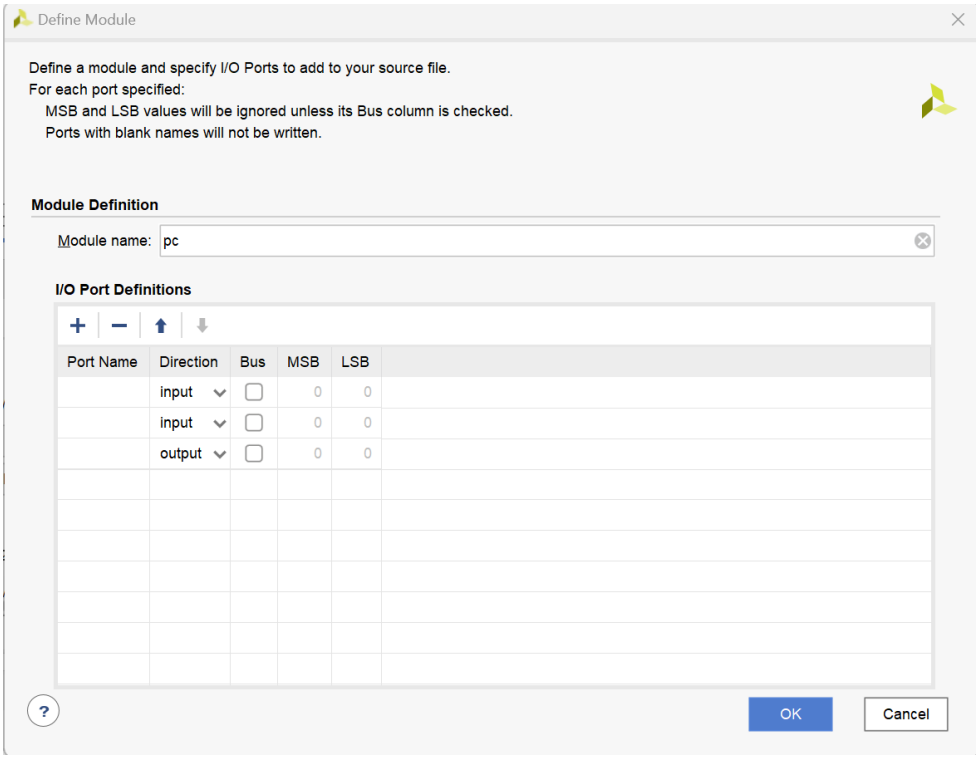


图 4.4.5 Define Module

(4) 在 Design Sources 下选中并双击刚才建立的“pc”源文件，然后编写 pc 的 Verilog 源代码。代码如下

```
module pc(clk, rst, pc);
    input wire clk; //时钟信号
    input wire rst; //重置信号
    output reg[7:0] pc; //PC寄存器

    always @(posedge clk) begin
        if (rst) //rst信号为1时重置
            pc <= 0;
        else //否则在时钟信号上升沿PC+1
            pc <= pc + 1;
    end
endmodule
```

4.4.2 仿真 PC 并查看波形

（1） 点击“+”添加仿真文件（图 4.4.6）。

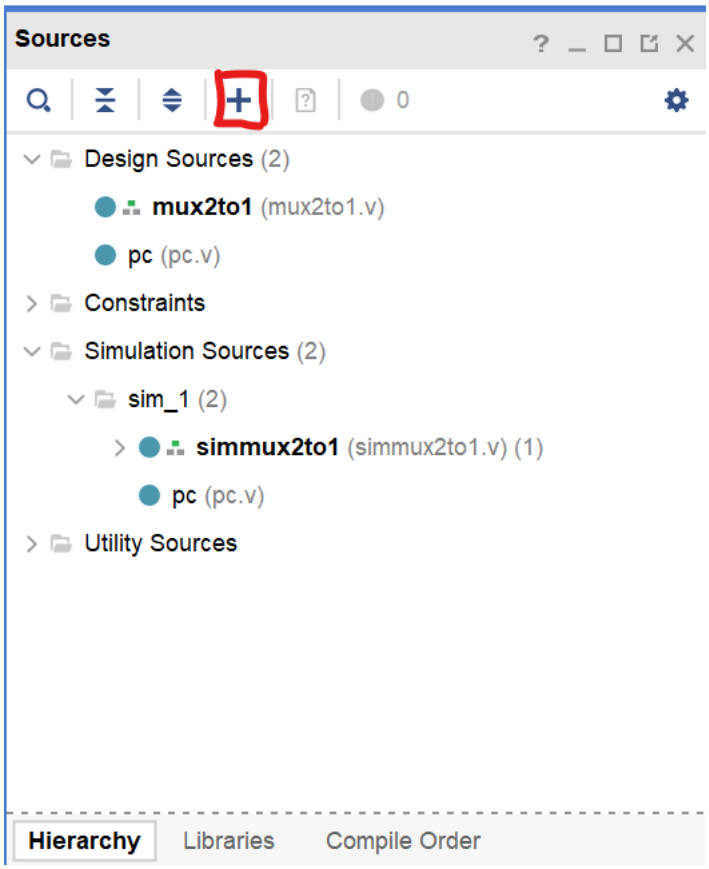


图 4.4.6 添加 pc 的仿真文件

（2） 设置 pc 的仿真文件（图 4.4.7）。

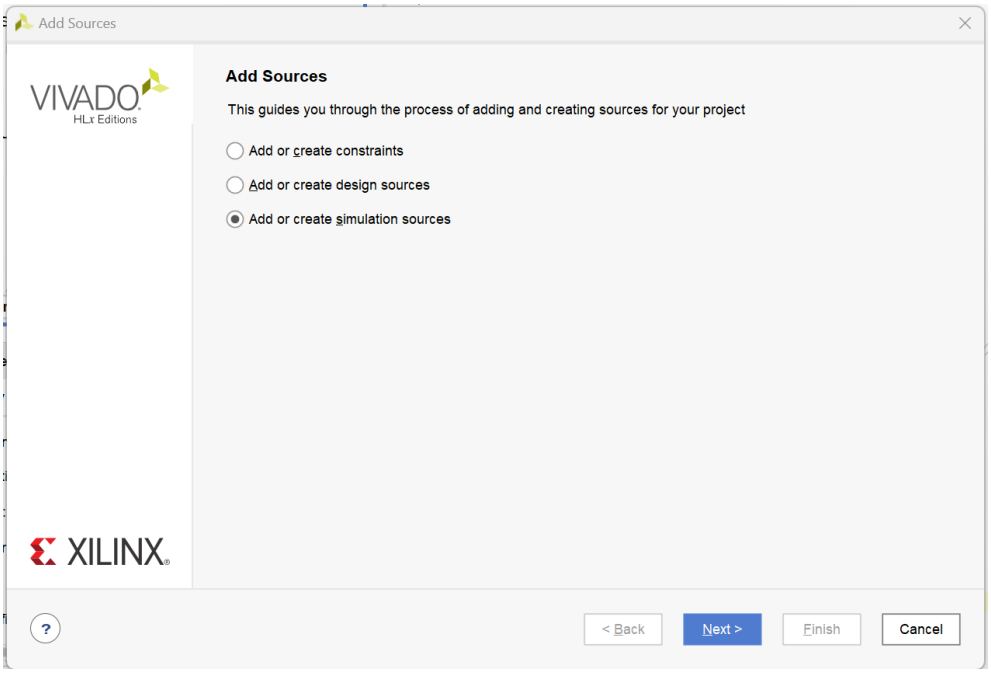


图 4.4.7 设置 pc 的仿真文件

（3） 建立 simpc 仿真文件（图 4.4.8）。

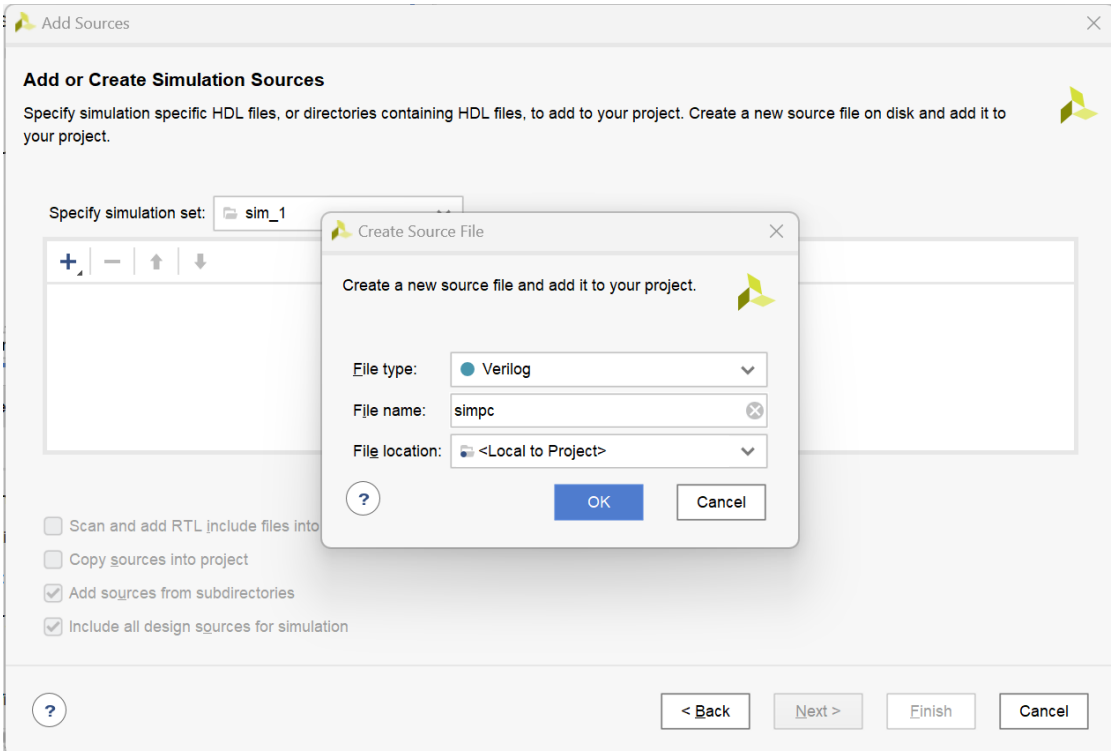


图 4.4.8 建立 simpc 仿真文件

（4） 添加用于仿真的测试程序，代码如下：

```
module simpc;
    reg clk;
    reg rst;
    wire [7:0] pc;

    initial begin
        clk = 1; //初始时将时钟信号置1

        forever begin
            if($time == 0 || $time == 80)
                rst = 1; //在开始和第80ns时将rst置0
            else
                rst = 0;
            if($time >= 160) $stop; //160ns后停止仿真
            #5 clk = ~clk; //每间隔5ns翻转时钟信号
        end
    end

    pc sim_pc(
        .clk(clk),
        .rst(rst),
        .pc(pc)
    );
endmodule
```

（5） 对 simpc 仿真（图 4.4.9）。

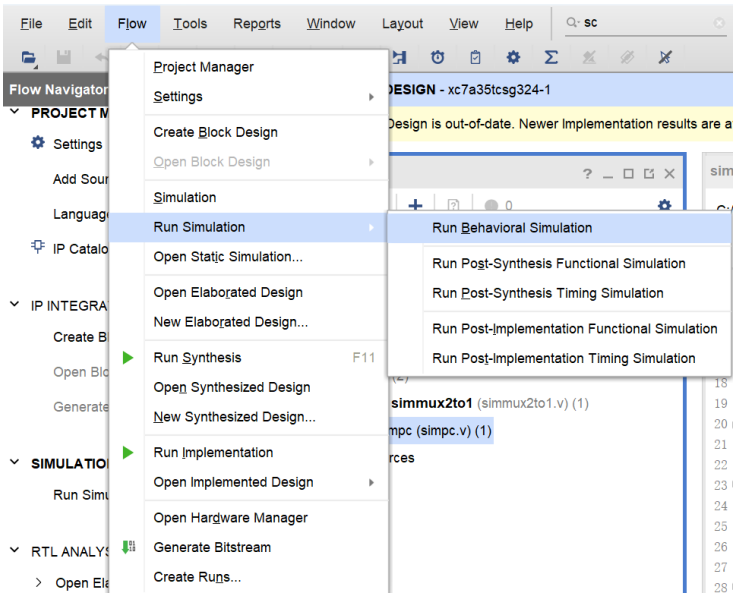


图 4.4.9 对 simpc 仿真

（6） 分析波形（图 4.4.10）

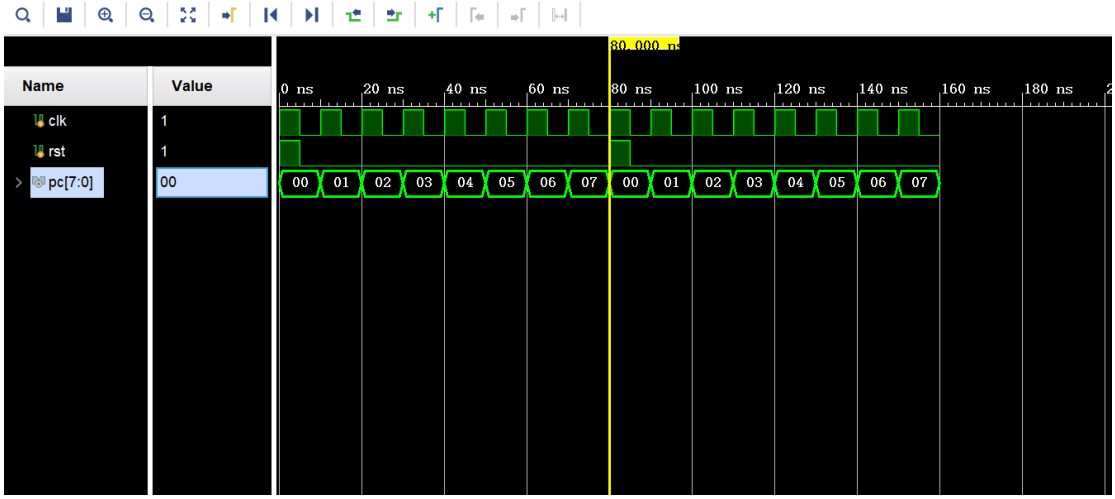


图 4.4.10 分析波形图

4.4.3 编写 RAM 源文件

(1) 建立 Verilog RAM 源文件。点击“+”号（图 4.4.11），选择“Add or create design sources”（图 4.4.12）。

图 4.4.11 创建 ram 源文件

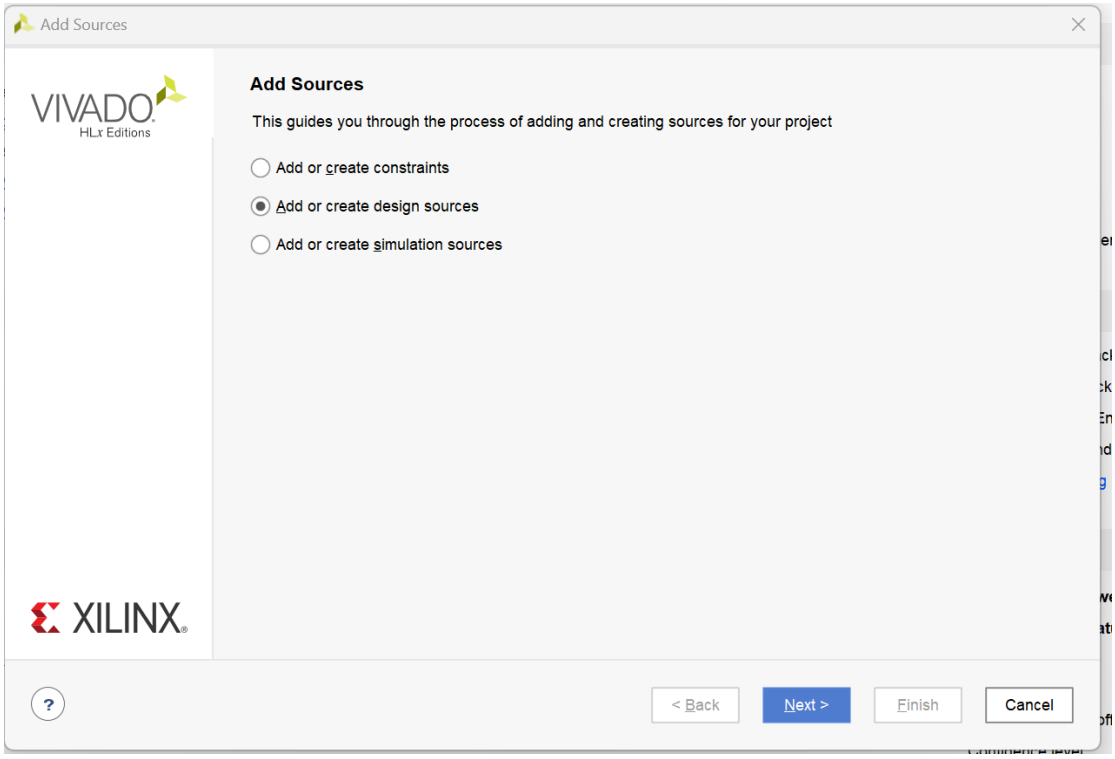


图 4.4.12 Add or create sources

(2) 选择“Create File”（图 4.4.13），file name 命名为“ram”并选择“OK”（图 4.4.14）。然后选择 Finish，创建文件结束。

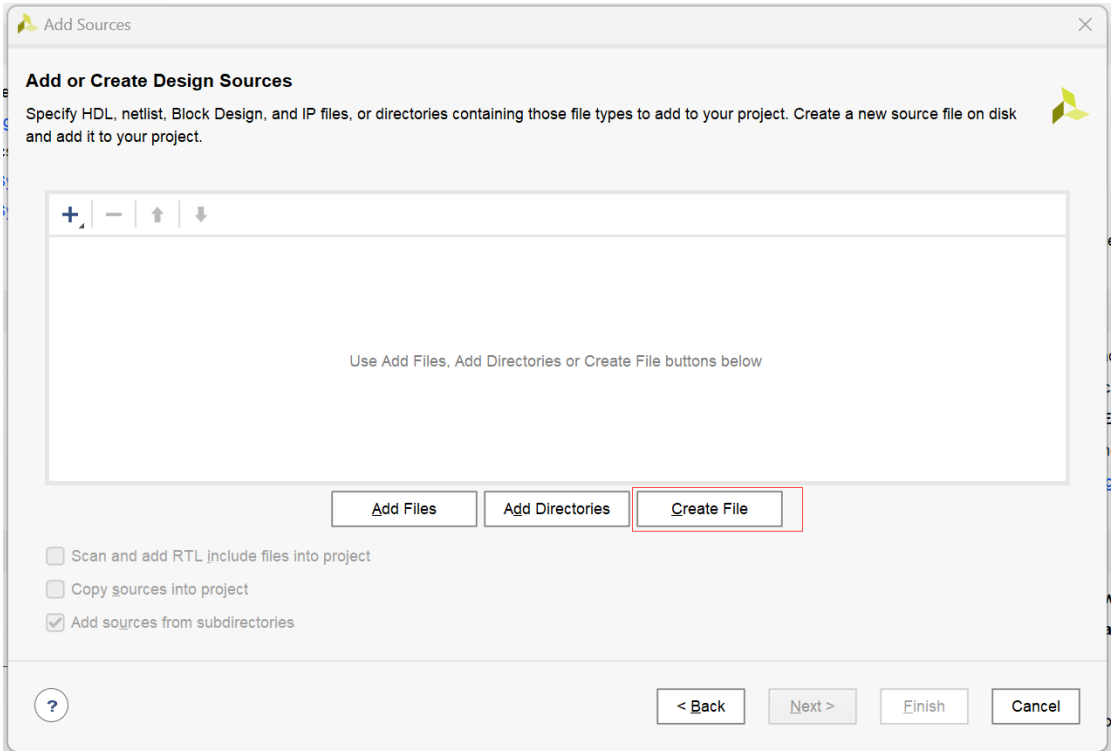


图 4.4.13 Create file

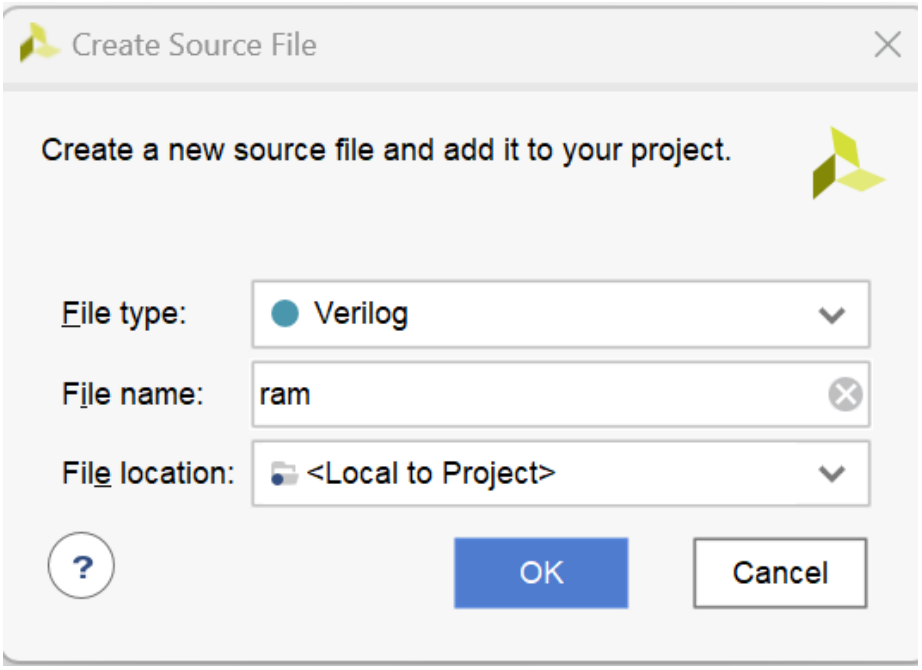


图 4.4.14 Create ram source file

(3) 在弹出的“Define Module”中（图 4.4.15），创建四个 input，一个 output，然后选择 ok（此步也可以不做）。

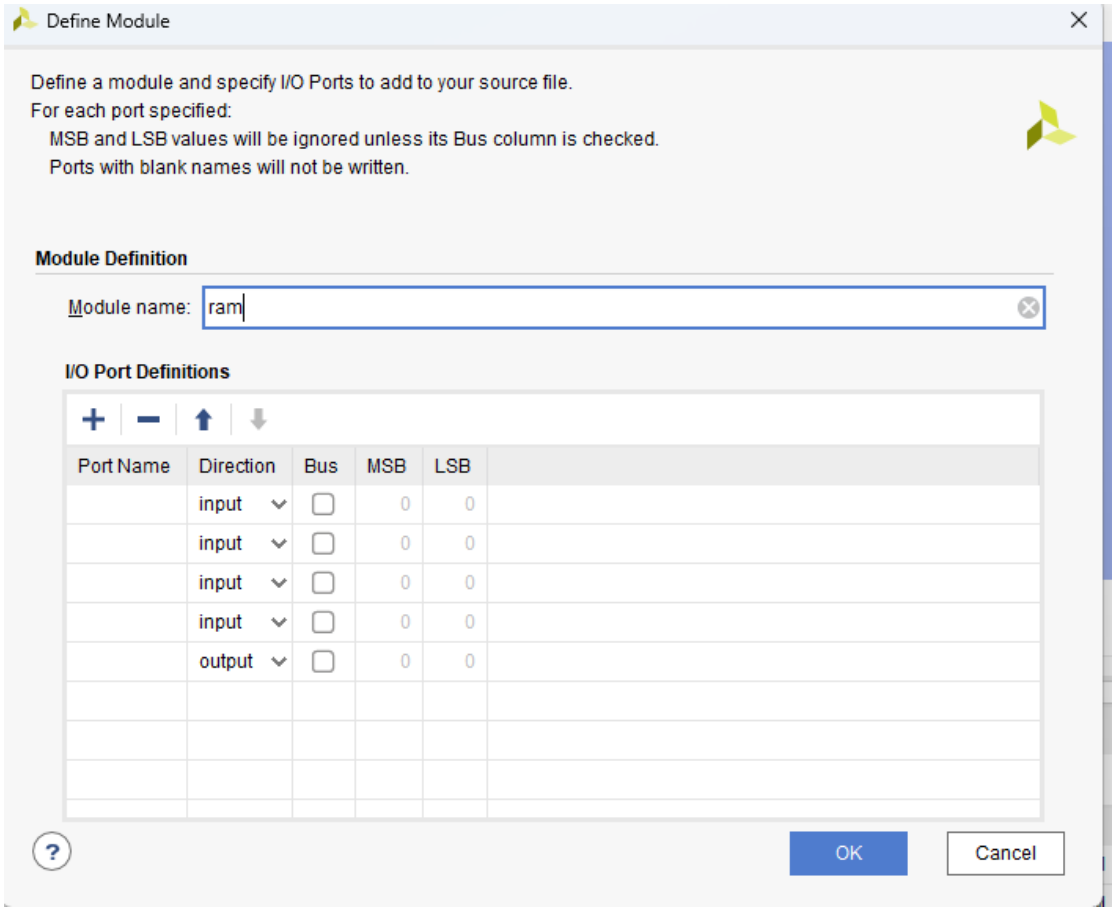


图 4.4.15 Define module

(4) 在 Design Sources 下选中并双击刚才建立的“register”源文件，然后编写 register 的 Verilog 源代码。代码如下：

```
module ram(
    input wire clk,              // 时钟信号
    input wire wr_en,            // 写使能信号
    input wire [9:0] addr,       // 存储器地址
    input wire [15:0] data_in,   // 写入数据
    output reg [15:0] data_out   // 输出数据
    );

    reg [15:0] ram[511:0];
    //RAM单元，每个存储字16位，一共可存512个存储字

    always @(posedge clk) begin
        if (wr_en) begin
          ram[addr] <= data_in; // 写入数据
        end
        else begin
          data_out <= ram[addr]; // 读取数据
        end
    end
endmodule
```

4.4.4 仿真 RAM 并查看波形

（1） 点击“+”添加仿真文件（图 4.4.16）。

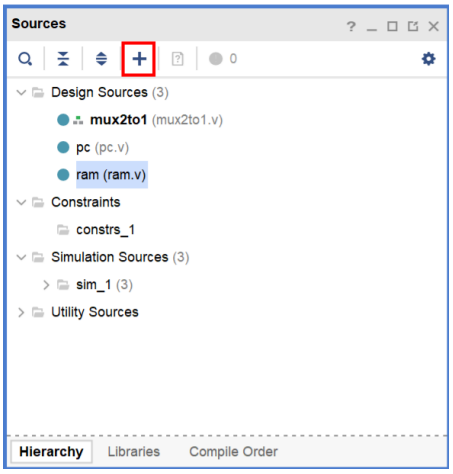


图 4.4.16 添加 ram 的仿真文件

（2） 设置 ram 仿真文件（图 4.4.17）。

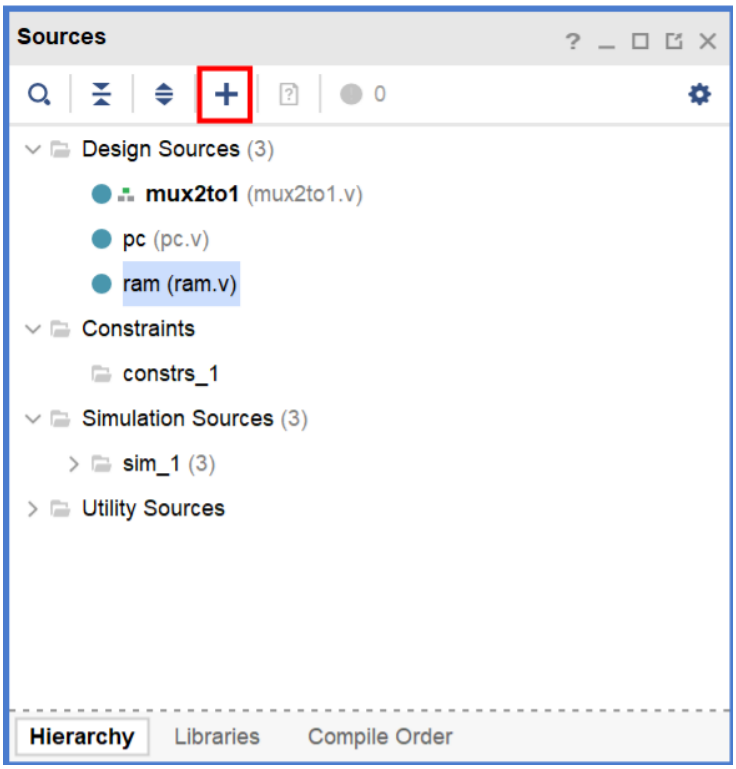


图 4.4.17 设置 ram 的仿真文件

（3） 建立 simram 仿真文件（图 4.4.18）。

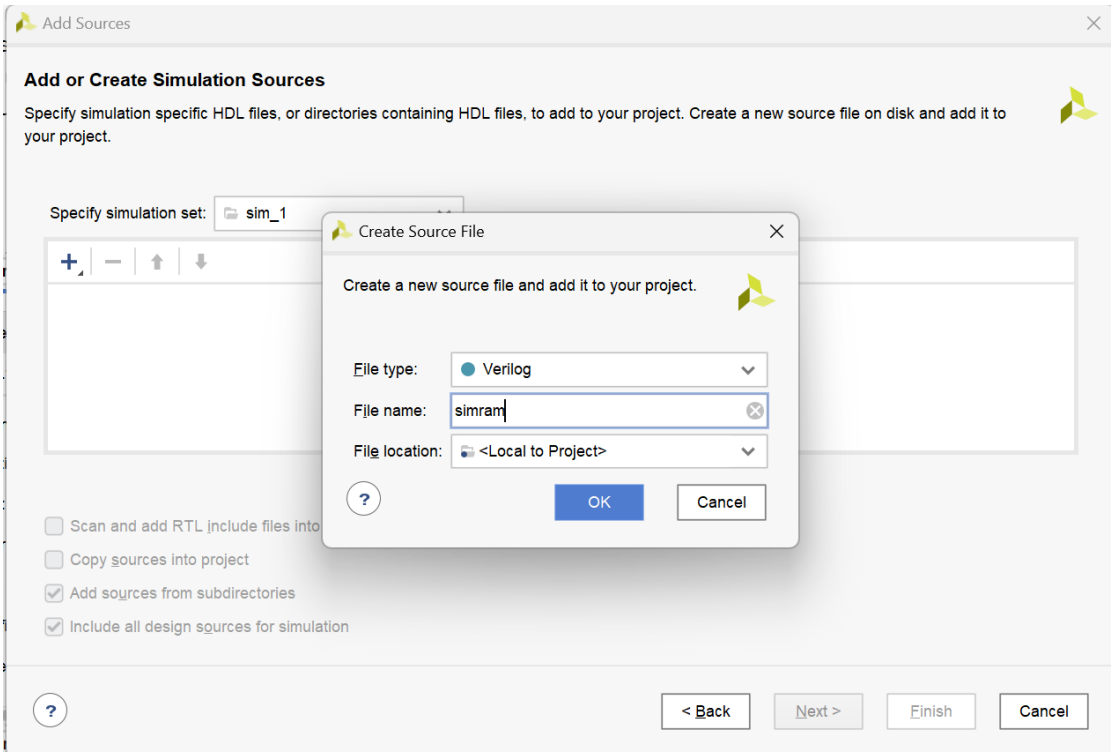


图 4.4.18 建立 simram 仿真文件

（4） 添加用于仿真的测试程序，代码如下：

```
module simram();
    reg clk;
    reg wr_en;
    reg [9:0] addr;
    reg [15:0] data_in;
    wire [15:0] data_out;

    ram dut(
       .clk(clk),
    .wr_en(wr_en),
    .addr(addr),
    .data_in(data_in),
    .data_out(data_out)
    );

    initial begin
        clk = 1;
        wr_en = 0;
        addr = 0;
        data_in = 0;
        #10;
        wr_en = 1; // 写入数据
        addr = 0;
        data_in = 16'h1234;
        #10;
        wr_en = 0; // 停止写入，开始读数
        addr = 0;
        #10;
        $display("Data out = %h", data_out);
        // 输出读取到的数据
        #10;
        $finish;
    end

  always #5 clk = ~clk; // 时钟信号产生器
endmodule
```

（5） 对 simram 仿真（图 4.4.19）。

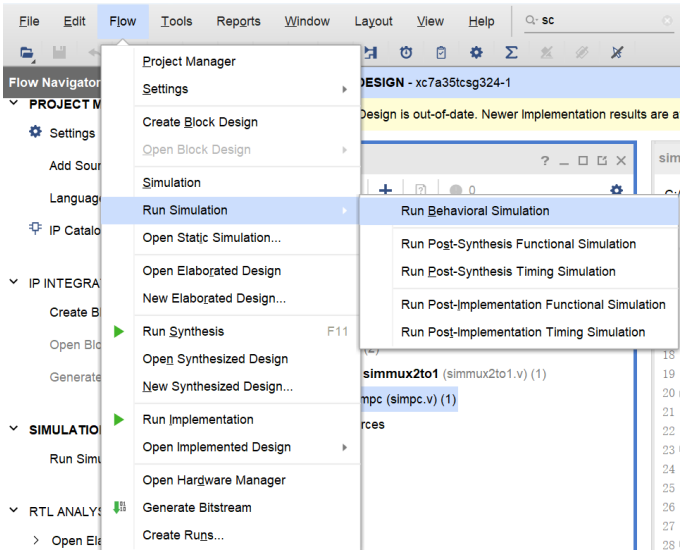


图 4.4.19 对 simram 进行仿真

（6） 分析波形（图 4.4.20）

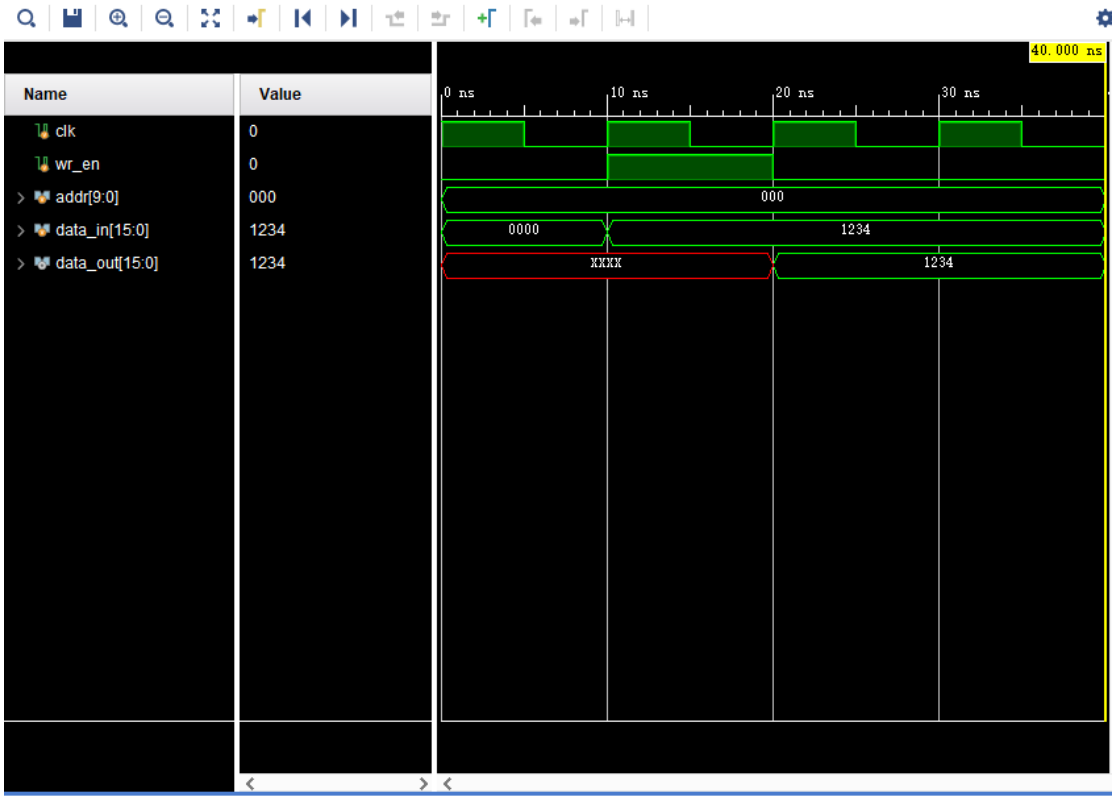


图 4.4.20 simram 分析波形图

实验五　单周期 CPU 设计与实现——单指令 CPU

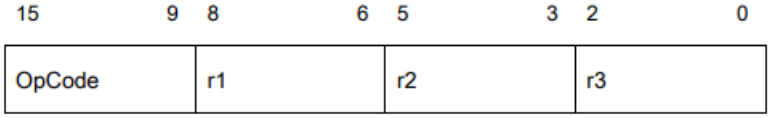
5.1 实验内容

设计和实现一个支持加法指令的 CPU，理解并掌握 CPU 设计的基本原理和过程。

5.2 实验目标

使用Verilog设计和实现一个支持加法指令的单周期CPU。要求该加法指令（表示为 add r1，r2，r3）格式约定如下：

- 采用寄存器寻址，r1，r2，r3 为寄存器编号，r1 和 r2 存放两个源操作数，r3为目标寄存器，其功能为[r1] + [r2] -> r3；
- 指令字长 16 位，操作码和地址码字段分配如下所示：



5.3 实验原理

（请同学们自行编写实验原理）

5.4 实验步骤

（请同学们自行补充实验步骤）

实验五将复用实验一到实验四编写的源代码，并且实现 insMem 模块，在 Vivado 中创建源文件和仿真文件的方法可以参照实验一到实验四，此处不再赘述。

5.4.1 编写 insMem 源文件

insMem 用于存放指令。代码如图 5.4.1 所示：

```
`timescale 1ns / 1ps

module insMem(
    input wire [7:0] addr, // 8位指令地址
    output reg [15:0] ins // 16位指令
    );

    reg [15:0] units [8:0]; //unit用来存放指令，每条指令16位，一共可访存8条指令

    initial begin
        units[1] = 16'b000000_000_001_010;
        units[2] = 16'b000000_001_010_011;
    end

    always@(addr) begin //当地址信号改变时， 输出相应的指令
        ins <= units[addr];
    end

endmodule
```

图 5.4.1 insMem 代码

在上述代码中，units 被用来存储指令。为了简单起见，这里的 units 只保存了 8 条指令，并且初始时有两条指令。需要注意的是，units[0]没有初始指令，这是因为我们假设当 rst=0 时，pc 的输出值为 0，此时不执行任何指令。

由于只有一条指令，因此指令中操作码部分没有实际意义，我们默认指令为加法，所以指令的操作码部分可以设为任何值（此处设置为 000000）。根据我们定义的指令格式，units[1]中的加法指令表示将 000 寄存器和 001 寄存器中的值相加，并把结果保存至 010 寄存器。units[2]中的加法指令表示将 001 寄存器和 010 寄存器中的值相加，并把结果保存至 011 寄存器。

5.4.2 编写 cu 源文件

控制单元 CU 的代码如所示：

```
`timescale 1ns / 1ps

module cu(
    input wire [6:0] opcode, //指令操作码
    output reg we, //寄存器堆写端口控制
    output reg re1,//寄存器堆读端口1控制
    output reg re2,//寄存器堆读端口2控制
    output reg [2:0] alu_op //ALU操作选择信号
    );

    always@(opcode) begin
        alu_op <= 3'b000; //此时只实现了一条加法指令
        we <= 1'b0; //低电平写
        re1 <= 1'b0; //低电平读
        re2 <= 1'b0; //低电平读
    end

endmodule
```

图 5.4.2 cu 代码

5.4.3 改写 register 源文件

为了简单起见，我们改写 register 源文件，在 0 号和 1 号寄存器中初始两个值用于加法计算。代码如图 5.4.3 所示：

```
`timescale 1ns / 1ps


module register(
    input wire clk,

    //写端口
    input wire we,
    input wire[2:0] waddr,
    input wire[15:0] wdata,

    //读端口1
    input wire re1,
    input wire[2:0]raddr1,
    output reg[15:0] rdata1,

    //读端口2
    input wire re2,
    input wire[2:0] raddr2,
    output reg[15:0] rdata2
    );



    reg[15:0]  regs[7:0]; //8个16位寄存器

    initial begin
        regs[0] <= 16'b0000_0000_0000_1111;
        regs[1] <= 16'b0000_0000_0010_0000;
    end


    always @ (posedge clk) begin
        if(we == 0) begin
            regs[waddr] <= wdata;
        end
        if ((raddr1 == waddr) && (re1 ==0)) begin
          //如果第一个读寄存器端口要读取的目标寄存器与要写入的目的寄存器是同一个寄存器，那么
          //直接将要写入的值作为第一个读寄存器端口的输出
          rdata1 <= wdata;
        end
        if((raddr2 == waddr) && (re2 == 0)) begin
          //如果第二个读寄存器端口要读取的目标寄存器与要写入的目的寄存器是同一个寄存器，那么
          //直接将要写入的值作为第二个读寄存器端口的输出
          rdata2 <= wdata;
        end
    end

    always @ (*) begin
          if(re1 == 0) begin
          rdata1 <= regs[raddr1];
      end else begin
          //如果第一个读寄存器端口不能使用，直接输出0
          rdata1 <= 16'h0000;
      end
    end

    always @ (*) begin
          if(re2 == 0) begin
          rdata2 <= regs[raddr2];
      end else begin
          //如果第二个读寄存器端口不能使用，直接输出0
          rdata2 <= 16'h0000;
      end
    end

endmodule

```

图 5.4.3 register 改写后的代码

5.4.4　编写 cpu 顶层封装文件

最后，编写 CPU 源文件。代码如图 5.4.4 所示：

```
`timescale 1ns / 1ps

module cpu(
    input wire clk,
    input wire rst
    );

    wire we;
    wire [2:0] alu_op;
    wire [7:0] addr;
    wire re1, re2;
    wire [15:0] result, ins, rdata1, rdata2;

    pc pc(
        .clk(clk),
        .rst(rst),
        .pc(addr)
    );

    insMem insMem(
        .addr(addr),
        .ins(ins)
    );

    register register(
       .clk(clk),
       .we(we),
       .waddr(ins[2:0]),
       .wdata(result),
       .re1(re1),
       .raddr1(ins[8:6]),
       .rdata1(rdata1),
       .re2(re2),
       .raddr2(ins[5:3]),
       .rdata2(rdata2)
    );

    alu alu(
       .in1(rdata1),
       .in2(rdata2),
       .alu_op(alu_op),
       .result(result)
    );

    cu cu(
       .opcode(ins[15:9]),
       .we(we),
       .re1(re1),
       .re2(re2),
       .alu_op(alu_op)
    );

endmodule
```

图 5.4.4cpu 代码

5.4.5 编写 cpu 仿真文件

CPU 仿真文件代码如图 5.4.5 所示：

```
1  `timescale 1ns / 1ps
2
3  module simcpu(
4      );
5
6      reg clk, rst;
7      initial begin
8          clk = 1;
9          rst = 1;
10         #10 rst = 0;
11     end
12
13     always begin
14         #5 clk = ~clk;
15         if ($time >= 50) $stop;
16     end
17
18     cpu cpu(
19         .clk(clk),
20         .rst(rst)
21     );
22 endmodule
23
```

图 5.4.5cpu 测试代码

编写完测试代码后可以仿真。注意，进行仿真后，由于在 simcpu 文件中仅定义了 clk 和 rst，因此如所示，在仿真后波形图界面可能只会显示出 clk 和 rst。若要在波形图中观察其他变量，可在“Scope”界面中选中相关模块，并在“Objects”界面中右键选择希望观察的变量，并点击“Add to Wave Window”（图 5.4.6）。然后依次点击“Restart”和“Run all”按钮重新仿真（图 5.4.6）。最后可以观察到添加变量后的波形图（图 5.4.7）。

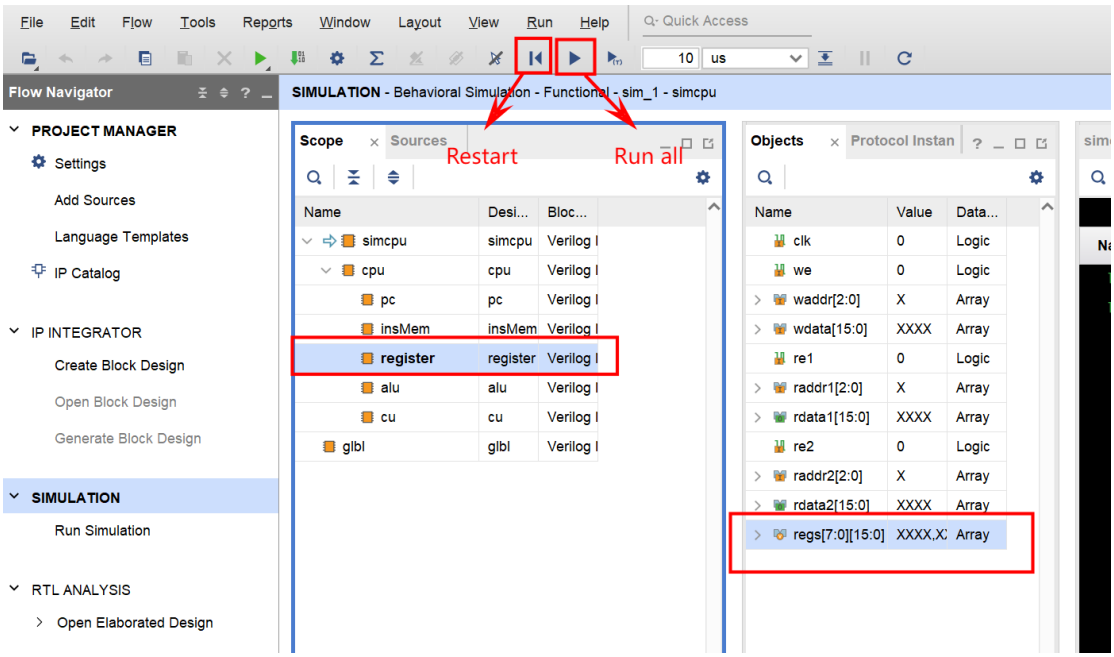


图 5.4.6 添加变量

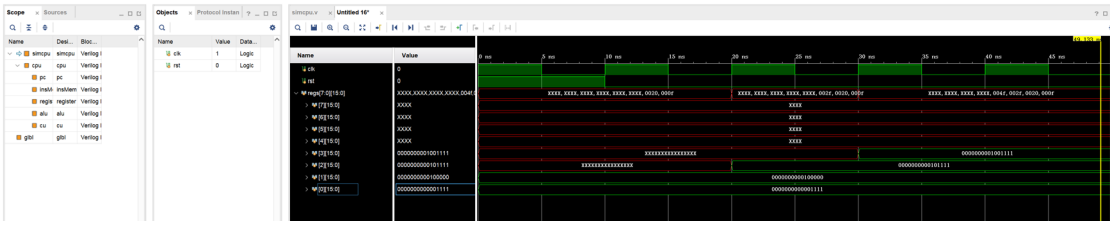


图 5.4.7CPU 仿真波形图

实验六　单周期 CPU 设计与实现——十条指令 CPU

6.1 实验内容

设计和实现一个支持至少十条指令的 CPU，进一步理解并掌握 CPU 设计的基本原理和过程。

6.2 实验目标

在实验五的基础上，设计和实现一个支持如下十条指令的单周期 CPU：

非访存指令	清除累加器指令 CLA
	累加器取反指令 COM
	算术右移一位指令 SHR：将累加器 ACC 中的数右移一位，结果放回 ACC
	循环左移一位指令 CSL：对累加器中的数据进行操作
	停机指令STP
访存指令	加法指令ADD X：[X] + [ACC]➞ACC，X 为存储器地址，直接寻址
	存数指令STA X，采用直接寻址方式
	取数指令LDA X，采用直接寻址
转移类指令	无条件转移指令 JMP imm：signExt(imm) ➞ PC
	有条件转移（负则转）指令 BANX: ACC 最高位为 1 则（PC）+ X ➞ PC,否则 PC 不变

在上述十条指令的基础上，也可以实现并设计更多的指令从而增加实验的复杂性。

6.3 实验原理

（请同学们自行编写实验原理，参照上次实验，依次给出指令格式定义、数据通路和控制单元的设计，并给出目标 CPU 的原理图。）

6.4 实验步骤

（对实验过程进行描述和分析，并给出相关代码和仿真结果。）