
hit-cdp-demo

发行版本 1.0.0

处理器设计与实践实验指导书

2023 年 08 月 27 日

目录

1	Welcome to hit-cdp-demo's documentation!	1
1.1	课程介绍	1
1.2	Verilog 简介	2
1.3	Vivado 安装说明	15
1.4	Vivado 使用说明	23
1.5	Lab1: 硬件描述语言基础	72
1.6	Lab2: 基本组合逻辑设计	74
1.7	Lab3: 内存与寄存器堆	78
1.8	Lab4: 给定指令系统的处理器设计	84
1.9	龙芯 Artix-7 实验板的使用	92
1.10	远程实验平台	93
1.11	实验四测试环境说明	97

CHAPTER 1

Welcome to hit-cdp-demo's documentation!

1.1 课程介绍

1.1.1 处理器设计与实践

本课程是计算机科学与工程专业的专业实践课，着重引导学生以整体观念看待 CPU 和设计 CPU，有助于计算机整体结构的理解，培养系统设计能力和理论与实践结合的能力。

培养目标

- 培养学生使用芯片开发工具进行芯片设计、测试及综合优化的能力
- 培养学生设计及实现较大规模硬件工程的能力
- 使学生深入理解处理器工作原理，了解不同架构下 CPU 的指令集架构差异并完成简单 RISC CPU 的设计与实现

1.1.2 课程安排

本课程以实验为主，共设置如下四个实验题目：

- [4 学时] 实验一：硬件描述语言基础
- [4 学时] 实验二：基本组合逻辑设计
- [8 学时] 实验三：内存与寄存器文件
- [16 学时] 实验四：给定指令集的 CPU 设计

1.1.3 考核方式

实验给分规则

每个实验按 100 分计算，其中每次实验的评分方式如下：

- 实验出勤 20 分
- 课堂操作/验收 40 分，其中当堂完成实验题目并完成验收，最高得分 40；延后 1 次课完成课堂验收，最高得分 36；延后 2 次课完成课堂验收，最高得分 32；延后 1 周完成课堂验收，最高得分 28
- 实验报告 30 分，按实验报告模板完成报告的撰写，并依据报告模板中的评分标准评分
- 讲解视频 10 分，录制实验程序的主要内容讲解，包括代码编写、调试遇到的主要问题、解决方法、体会等

整理评分规则

所有 4 次实验的得分进行加权求和，四次实验的权值分别为：0.1、0.15、0.25 和 0.5，即为课程总得分。

1.1.4 参考资料

- [1] 王志英等，《计算机体系结构(第2版)》
- [2] 舒燕君等，《处理器设计与实践实验指导》

1.2 Verilog 简介

Verilog 是一种用于描述、设计电子系统（特别是数字电路）的硬件描述语言。

然而在硬件设计中，测试/验证所设计硬件的正确性也是很重要的环节，因此 Verilog 有一部分语法是方便做验证的，不能进行综合实现到硬件上。

所以学习 Verilog，主要学习 2 个部分：

- 可综合 (Synthesizable) 代码编写，这部分的代码能映射到硬件上
- 测试 (Testbench) 代码编写，可以使用 Verilog 全部的语法，方便验证设计的正确性

具体什么代码可以综合，请参考 Vivado Synthesis Guide。

1.2.1 可综合代码设计

Verilog 四值逻辑系统

Verilog 的逻辑系统中有四种值，也即四种状态。

- 逻辑 0：表示低电平；
- 逻辑 1：表示高电平；
- 逻辑 X：表示未知，有可能是高电平，也有可能是低电平；
- 逻辑 Z：表示高阻态，通常是没有激励信号造成的；

在仿真时，如果观察到一个信号为 X，这常常是驱动它的信号中有 Z 或者 X 造成的。

Verilog 变量类型

Verilog 所用到的所有变量都属于两个基本的类型：线网 (wire) 类型和寄存器 (reg) 类型。

线网 (wire)

线网与我们实际使用的电线类似，它的数值一般只能通过连续赋值，由赋值符右侧连接的驱动源决定。

例如：

```
wire a, b, c;           // 定义3个wire类型变量
assign a = b & c;       // a被(b & c)持续驱动，当(b &
→c)变化时，(不考虑延迟)a会同步变化
                           // 当然b和c作为wire类型，也需要被驱动
wire [1:0] d = {b, c}; // 定义1个宽度为2的wire类型变量并同时赋值
```

寄存器 (reg)

寄存器与之不同，它可以保存当前的数值，直到某个时机另一个数值被赋值给它。如果未对寄存器变量赋值，它的初始值则为 X。

Verilog 中的寄存器 (reg) 和硬件上的寄存器不能完全等价，具体区别在之后说明。

例如：

```
reg      valid;        // 宽度为1的reg变量
reg [1 :0] data_size; // 宽度为2的reg变量
reg [31:0] data;     // 宽度为64的reg变量
```

变量类型的选择

wire 类型还是 reg 类型，主要是根据自己需求选择。但需要符合一定的规定：

- 模块的输入端口只能是 wire，但可以被 wire/reg 驱动
- 模块的输出端口可以是 wire/reg，但是只能驱动外界的 wire。

数字的表示

对变量赋值时，需要使用下述数字表示方法。

基本格式：<位宽>'<数制的符号><数值>

- 位宽是与数据大小相等的对应二进制数的位数加上占位所用 0 的位数，这个位数需要使用十进制来表示。
- 如果省略位宽，那么实际宽度与综合工具实现有关。
- 数制需要用字母来表示，h 对应十六进制，d 对应十进制，o 对应八进制，b 对应二进制。

例如：

例子	说明
233	10 进制数 233
8'hff	8 位 16 进制数 FF
32'd1234	32 位 10 进制数 1234
3'b010	3 位 2 进制数 10
32'habcd_1234	32 位 16 进制 abcd1234
8'hz	8 位高阻态，可以赋给 wire 类型

向量

向量形式的数据是 Verilog 相对 C 语言较为特殊的一种数据。

向量和 C 语言的数组不能等价，向量可以直接进行加减等运算。

```
reg [7:0] data;
wire [7:0] a, b, c; // 声明 3 个向量 a, b, c
assign a = 8'd1;
assign b = 8'd2;
assign c = a + b; // c == 8'd3

wire [3:0] a_high = a[7:4]; // a 的高 4 位
wire [3:0] a_low = a[3:0]; // a 的低 4 位
```

数组

声明数组时，和声明向量不同，方括号位于数组名的后面，括号内的第一个数字为第一个元素的序号，第二个数字为最后一个元素的序号，中间用冒号隔开。

如果数组是由向量构成的，则数组的其中某个元素是向量。

```
wire      array [15:0]; // 一维数组
wire [7:0] vec_array [15:0]; // 一维数组
wire [7:0] mat      [15:0][15:0]; // 二维数组
reg       reg_array [15:0]; // reg 构成的一维数组

assign array[0]     = 1'b1;
assign vec_array[0] = 16'd2;
assign mat[0][9]   = 8'd3;

always @ (posedge clk) begin
  ...
  reg_array <= 16'hffff;
  ...
end
```

参数

可以通过 `parameter` 关键字声明参数以增强模块可拓展性和可读性。参数与常数的意义类似，不能够通过赋值运算改变它的数值。

在模块实例化时，可以使用 `#()` 将所需的实例参数覆盖模块的默认参数。

局部参数可以用 `localparam` 关键字声明，它不能够进行参数重载。

例如，我们声明一个宽度可变的加法器：

```
module adder #( // 默认宽度为 4
    parameter WIDTH = 4
) (
    input [WIDTH - 1 : 0] a,
    input [WIDTH - 1 : 0] b,
    output [WIDTH - 1 : 0] c
);
    assign c = a + b;
endmodule
```

在我们例化这个模块时，可以进行如下操作：

```
// 覆盖宽度，修改为 6
adder #( .WIDTH(6) ) U_adder_0(
    .a(a),
    .b(b),
    .c(c)
);

// 使用默认的宽度 4
adder U_adder_1(
    .a(a),
    .b(b),
    .c(c)
);
```

局部参数 `localparam` 和参数类似，但是不能在例化时被覆盖。

局部参数样例：

```
// 声明一个乘法器
module mult (
    [port list], ...
)

reg [1:0] state; // 乘法器有3个状态

localparam IDLE = 2'd0;
localparam CALC = 2'd1;
localparam DONE = 2'd2;

...
// 参数在模块内可以直接引用
wire mult_ready = (state == IDLE);
...

endmodule
```

运算

Verilog 的许多运算符和 C 语言类似，但是有一部分运算符是特有的，例如拼接运算符、缩减运算符、带有无关位的相等运算符等。

按位运算

- 按位取反 `~`: 1 个多位操作数按位取反。例如: `a = 4'b1011`, 则 `~a` 的结果为 `4'b0100`
- 按位与 `&`: 2 个多位操作数按位进行与运算, 各位的结果按顺序组成一个新的多位数。例如: `a = 2'b10, b = 2'b11`, 则 `a & b` 的结果为 `2'b10`
- 按位或 `|`: 2 个多位操作数按位进行或运算, 各位的结果按顺序组成一个新的多位数。例如: `a = 2'b10, b = 2'b11`, 则 `a | b` 的结果为 `2'b11`
- 按位异或 `^`: 2 个多位操作数按位进行异或运算, 各位的结果按顺序组成一个新的多位数。例如: `a = 2'b10, b = 2'b11`, 则 `a ^ b` 的结果为 `2'b01`

逻辑运算

- 逻辑取反 `!`: 对 1 个操作数进行逻辑取反。
- 逻辑与 `&&`: 对 2 个操作数进行逻辑与。
- 逻辑或 `||`: 对 2 个操作数进行逻辑或。

缩减运算

- 缩减与 `&`: 对一个多位操作数进行缩减与操作, 计算所有位之间的与操作结果, 例如: `&(4'b1011)` 的结果为 0
- 缩减或 `|`: 对一个多位操作数进行缩减或操作, 计算所有位之间的或操作结果。例如: `|(4'b1011)` 的结果为 1
- 缩减异或 `^`: 对一个多位操作数进行缩减异或操作, 计算所有位之间的异或操作结果。例如: `^(4'b1011)` 的结果为 1

缩减或非、缩减异或、缩减同或是类似的。

算术运算

- 加 `+`: 2 个操作数相加
- 减 `-`: 2 个操作数相减或取 1 个操作数的负数
- 乘 `*`: 2 个操作数相乘
- 除 `/`: 2 个操作数相除
- 求余 `%`: 2 个操作数求余

关系运算

- 大于`>`: 比较 2 个操作数，如果前者大于后者，结果为真
- 小于`<`: 比较 2 个操作数，如果前者小于后者，结果为真
- 大于或等于`>=`: 比较 2 个操作数，如果前者大于或等于后者，结果为真
- 小于或等于`<=`: 比较 2 个操作数，如果前者小于或等于后者，结果为真
- 逻辑相等 `==`: 2 个操作数比较，如果各位均相等，结果为真。如果其中任何一个操作数中含有 x 或 z，则结果为 x
- 逻辑不等`!=`: 2 个操作数比较，如果各位不完全相等，结果为真。如果其中任何一个操作数中含有 x 或 z，则结果为 x

移位运算

- 逻辑右移`>>`: 1 个操作数向右移位，产生的空位用 0 填充
- 逻辑左移`<<`: 1 个操作数向左移位，产生的空位用 0 填充
- 算术右移`>>>`: 1 个操作数向右移位。如果是无符号数，则产生的空位用 0 填充；有符号数则用其符号位填充
- 算术左移`<<<`: 1 个操作数向左移位，产生的空位用 0 填充

其他运算

- 拼接 `{,}`: 2 个操作数分别作为高低位进行拼接，例如：`{2'b10, 2'b11}` 的结果是 `4'b1011`
- 重复 `{n{m}}`: 将操作数 m 重复 n 次，拼接成一个多位的数。例如：`a=2'b01`，则 `{2{a}}` 的结果是 `4'b0101`
- 条件`? :`: 根据? 前的表达式是否为真，选择执行后面位于：左右两个语句。例如：`assign c = (a > b) ? a : b`，如果 a 大于 b，则将 a 的值赋给 c，否则将 b 的值赋给 c

备注:

1. 使用拼接可以实现循环左移或者右移，如：`assign c = {a[0], a[7:1]}` 是一个循环右移
2. 注意优先级，与运算优先级总是大于或运算
3. 有两个操作符的运算尽量保证二者宽度相等，不满足条件时可以用拼接运算手动添 0 补齐
4. 嵌套的条件运算可以容易地实现组合逻辑，但是要注意优先级
5. 可以用拼接和重复实现符号拓展：如拓展一个 8 位有符号数 imm 至 16 位：`{8{imm[7]}, imm}`

组合逻辑

组合逻辑可由 2 种方法表示：assign 语句和 always 块。

assign 语句

assign 语句是对线网类型变量的赋值。线网不能够像寄存器那样储存当前数值，它需要驱动源提供信号，这种驱动是连续不断的，因此线网变量的赋值称为连续赋值。

对同一个 wire 变量赋值的 assign 语句只能有一条。

样例：

```
wire [1:0] a;
wire [1:0] b;
wire [3:0] c = 4'b1011;
wire       d;
wire       e;

assign {a, b} = c; // a = 2'b10, b = 2'b11
assign d = c[0] ? a[0] :
               c[1] ? a[1] : 1'b0;
assign e = a[0] & b[0];
```

always 块

不推荐使用 always 块来表示组合逻辑。不正确的使用会生成大量锁存器。

下面的例子是一个 32-5 优先编码器，如果 tlb_hit_array 全为 0，那么 tlb_hit_index 也为 0。

always 块中被赋值的变量只能是 reg 类型，但是该编码器并不会综合出寄存器或者锁存器，这是因为 Verilog 中的寄存器 (reg) 和硬件上的寄存器不能完全等价。

如果去掉 tlb_hit_index = 5'd0；一句，则会综合出锁存器。

```
reg [4 :0] tlb_hit_index;
wire [31:0] tlb_hit_array;

integer i;
always @(*) begin
    tlb_hit_index = 5'd0;
    for (i = 0; i < 32; i = i + 1) begin
        if (tlb_hit_array[i]) begin
            tlb_hit_index = i;
        end
    end
end
end
```

警告：

- 建议大家使用 assign 表示组合逻辑，这种做可以保证不会综合出锁存器
- 要避免写出组合逻辑环，也就是 assign a = b; assign b = ~a 这种代码

备注: 思考: 为什么不要让代码综合出不必要的锁存器? (时序分析、资源占用、需求等等)

时序逻辑

时序逻辑是在 `always` 块中描述的，被赋值的变量一定要是 `reg` 类型。

下面的例子可以在时钟上升沿交换 `a, b` 的值：

```
reg a, b;

always @ (posedge clk) begin
    a <= b;
    b <= a;
end
```

上个例子中，寄存器没有做复位，这样实现的时候寄存器初值是不固定的。

```
reg a, b;

always @ (posedge clk) begin
    if (~rst_n) begin // 低电平有效
        a <= 1'b0;
        b <= 1'b0;
    end else if (...) begin
        ...
    end
    else begin
        a <= b;
        b <= a;
    end
end
```

需要注意，对一个寄存器变量的赋值一定要在一个 `always` 块中，不要写出多驱动代码。否则就算仿真时能波形正常，但是综合时也会失败。

在 Verilog 中，有两种赋值运算，一种叫做阻塞赋值（blocking assignment），其运算符为 `=`；另一种叫做非阻塞赋值（non-blocking assignment），其运算符为 `<=`。

在顺序代码块中使用阻塞赋值语句，如果这一句没有执行完成，那么后面的语句不会执行；如果在顺序代码块中使用非阻塞赋值，则执行这一句的同时，并不会阻碍下一句代码的执行。在时序逻辑中应使用非阻塞赋值。

警告:

1. 我们建议使用同步复位，也就是敏感列表中只有 `posedge clk`，不要把 `rst` 信号添加到敏感列表，更不要添加别的信号
2. 在时序逻辑中应使用非阻塞赋值
3. 避免写出多驱动的代码，对一个寄存器变量的赋值一定要在一个 `always` 块中
4. 建议使用复位信号实现对寄存器的初始化，对大块内存（指令内存或数据内存）的初始化参考 Vivado/ISE 初始化 Block Memory 教程

条件语句

表示条件可以以下 3 种方法：

条件运算符：

使用条件运算符表示组合逻辑。例如：assign data_out = source_a ? data_a : data_b;

条件运算符可以嵌套使用：例如：

```
// 有优先级的 4 选 1 MUX
assign data_out = source_a ? data_a :
                  source_b ? data_b :
                  source_c ? data_c : data_d;
```

if-else 语句

if—else 语句常用于描述时序逻辑中。例如这个例子：

```
always @ (posedge clk) begin
    if (~rst_n) begin
        a <= 1'b0;
    end else if (src_b) begin
        a <= b;
    end else if (src_c) begin
        a <= c;
    end
end
```

时钟上升沿 src_b 和 src_c 都为 0 时，a 会保持自己的值。

case 语句

常用于描述时序逻辑，语法和 C 语言类似，例如如下的自动机：

```
reg [2:0] state;

always @ (posedge clk) begin
    if (!rstn) begin
        state <= RESET;
    end
    else begin
        case (state)
            IDLE      : state <= RUN;
            RUN       : state <= (req_reg && !hit_data) ? MISS : RUN;
            MISS      : state <= already ? FILL : MISS;
            FILL      : state <= (axi_datacom && rlast) ? FINISH : FILL;
            FINISH    : state <= RUN;
            RESET     : state <= (reset_cnt == 7'd127) ? IDLE : RESET;
            default   : state <= RESET;
        endcase
    end
end
```

使用 verilog attribute (* full_case, parallel_case *) 可以让 case 语句变成并行的而且不需要 default 语句。具体可见 ISE/Vivado 手册。

备注:

1. 这三种语句都是有优先级的
2. if-else 和 case 推荐只在时序逻辑中使用

顺序代码块

begin-end 组合代表了这个代码块的各行代码是顺序执行的，这种代码块称为顺序代码块。

```
always @ (posedge clk) begin
    if (src_b) begin
        a <= b;
    if (src_c) begin
        a <= c;
    end
end
```

在时钟上升沿，如果 src_b, src_c 同时为 1，那么 a 会被赋值为 c，这是因为顺序代码块的各行代码是顺序执行的。这样 src_c 的优先级更高。如果想要 src_b 优先，只要像前面的例子使用嵌套 if-else 语句就行了。

模块声明和例化

模块被包含在关键字 module、endmodule 之内。

```
module adder (           // 模块名称声明
    input [31:0] a,      // 输入输出声明
    input [31:0] b,
    output [31:0] c
);
    assign c = a + b;   // 变量声明、always语句、assign语句等
endmodule             // 模块结束
```

在实现 CPU 的顶层代码时，需要进行大量的模块例化，这里给出了一个例子进行说明。

当进行模块的端口声明时，如果没有明确指出其类型，那么这个端口会被隐含地声明为 wire 类型。

如定义一个模块 uart_tx:

```
module uart_tx (
    input          clk,
    input          valid,
    ...
    output        ready,
    output reg txd
)
    ...
endmodule
```

我们在模块 `uart_top` 中例化这个模块，并命名为 `U_uart_tx`:

```
module uart_top (
    input      clk,
    input [7:0] tx_data,
    input      tx_valid,
    ...
    input      rxd,
    output     txd,
    ...
)

wire tx_start, tx_ready;

// uart_top 调用 uart_tx, 将其实例化为 U_uart_tx
uart_tx U_uart_tx (
    .clk      (clk),
    .valid    (tx_start),
    ...
    .ready    (tx_ready),
    .txd      (txd)
)

...
endmodule
```

Generate 块

这里只介绍 `generate for` 块。

`generate for` 的主要功能就是对模块或组件以及 `always` 块、`assign` 语句进行复制。

使用 `generate for` 的时候，必须要注意以下几点要求

- 在使用 `generate for` 的时候必须先声明一个 `genvar` 变量，用作 `for` 的循环变量。`genvar` 是 `generate` 语句中的一种变量类型，用于在 `generate for` 语句中声明一个正整数的索引变量。
- `for` 里面的内嵌语句，必须写在 `begin-end` 里
- 尽量对 `begin-end` 顺序块进行命名

`generate for` 的语法示例如下：

```
genvar i;
generate for (i = 0; i < 4; i = i + 1) begin: gen_assign_temp
    assign temp[i] = indata[2 * i + 1 : 2 * i];
end
endgenerate
```

编译指令

Verilog 具有一些编译指令，它们的基本格式为`<keyword>`，注意第一个符号不是单引号，而是键盘上数字 1 左边那个键对应的撇号。

常用的编译指令有文本宏预定义`define`，`include`，它们的功能与 C 语言中类似，分别提供文本替换、文件包含的功能。

Verilog 还提供了`ifdef`，`ifndef`等一系列条件编译指令，设计人员可以使得代码在满足一定条件的情况下才进行编译。

此外，`timescale` 指令可以对时间单位进行定义。例如：`timescale 1ns / 1ps`。

Attribute

Verilog 的 Attribute 通常用于对综合工具进行更细粒度的控制。比如说实现乘法，既可以用 FPGA 上的通用资源（查找表，触发器等）实现，也可以用专用的 DSP 模块实现，使用 Attribute 可以指导综合工具选择我们希望的方案。

Attribute 在变量、模块声明前，修饰该变量或模块。

举个常用的例子：`(* keep = "true" *) wire sig1;`，该语句告诉综合工具，在综合时不要将其优化掉。

1.2.2 测试代码设计

测试编写的 Verilog 模块 `xxx`，通常需要编写一个测试模块 `tb_xxx` 调用它并测试，该模块不会参与综合，可以使用些不可综合代码以方便测试编写。

时延语句

仿真中还经常使用时延来构造合适的仿真激励。它是不可综合的，仅能够在仿真中使用。时延分两类，一是语句内部时延，二是语句间时延，其示例如下所示：

```
// 语句内部时延
A = #5 1'b1;

// 语句间时延
begin
    Temp = 1'b1;
    #5 A = Temp;
end
```

两种方式都表达在五个时间单位后，将 A 的值赋为 1。

initial 语句

一般用来生成复位信号和激励。只在仿真开始时执行一次。

例如：

```
// 测试代码中
// 假设`timescale 1ns / 1ps
reg rst_n, clk;

always #5 clk = ~clk;

initial begin
    rst_n = 1'b0;
    clk   = 1'b0;
#50 rst_n = 1'b1;
end
```

会生成一个 100MHz 的时钟，一个 50ns 有效的复位信号。

警告：

- 虽然 initial 在 FPGA 上有时可以综合，但是在 ASIC 上不可综合。因此尽量减少在设计代码中使用 initial 语句，一般寄存器初始化应使用复位信号。
- 为了方便，FPGA 上如果要初始化大块寄存器堆，可以用 initial + \$readmemh，更好的方法是使用 Block RAM IP 核。

系统任务

系统任务可以被用来执行一些系统设计所需的输入、输出、时序检查、仿真控制操作。所有的系统任务名称前都带有符号 \$ 使之与用户定义的任务和函数相区分。

常见的系统任务：

- \$display：用于显示指定的字符串，然后自动换行（用法类似 C 语言中的 printf 函数）
- \$time：可以提取当前的仿真时间
- \$stop：暂停仿真
- \$finish：终止仿真
- \$random：生成随机数
- \$readmemh：读入一个 16 进制数的文件以初始化 reg

测试设计

测试最基本的结构包括信号声明、激励和模块例化。

测试模块声明时，一般不需要声明端口。因为激励信号一般都在测试模块内部，没有外部信号。

声明的变量应该能全部对应被测试模块的端口。当然，变量不一定要与被测试模块端口名字一样。但是被测试模块输入端对应的变量应该声明为 reg 型，输出端对应的变量应该声明为 wire 型。

仿真过程中可以使用 \$display 显示当前仿真进度或者测试结果。

在测试完成或者发现错误时可以使用 \$finish；或者 \$stop；来停止仿真。

1.2.3 推荐的编码风格

- 使用代码对齐、宏定义、参数增强可读性
- module, wire, reg 等变量小写, parameter, `define 一般大写
- 低有效的信号加上 _n 后缀, 例如 rst_n
- 模块例化名用 U_name[_number] 标识, 多次例化加上序号 0, 1, 2 等。如: U_icache, U_tag_0

1.2.4 参考资料

1. Verilog. 维基百科, 自由的百科全书: <https://zh.wikipedia.org/w/index.php?title=Verilog>
2. Verilog 教程. 菜鸟教程: <https://www.runoob.com/w3cnote/verilog-tutorial.html>
3. Synthesis and Simulation Design Guide(UG626). Xilinx: https://www.xilinx.com/support/documentation/sw-manuals/xilinx14_7/sim.pdf

1.3 Vivado 安装说明

在学习并尝试本章节前, 你需要具有以下环境:

- Windows 或 Linux 操作系统的电脑一台。
- 连通的网络。

通过本章节的学习, 你将获得:

- Vivado 的下载方法。
- Vivado 的在线安装方法。

本地安装与在线安装的优劣势对比如下。

安装方法	优势	劣势
本地安装	1. 安装过程不需要联网。	1. 本地安装包有 26G。提前下载或拷贝较慢。2. 下载过程中需要注册 Xilinx 账号, 以后会不停收到一堆推荐邮件。
在线安装	1. 安装包小, 只有 50M 左右。2. 安装可根据需求选择版本和支持的器件。	1. 安装过程中需要联网, 且不能断网。2. 下载安装包和在线安装时, 需要 Xilinx 账号。

1.3.1 Vivado 简介

Vivado 是 Xilinx 的一款开发套件 EDA 工具。下面以 Vivado 2019.2 的 WebPACK 版本在 Windows 上安装为例, WebPACK 版本为免 license 的 Vivado 免费版, 支持的器件受限, 该版本支持 Artix-7 器件的开发, 足够完成本实验, 默认安装要求至少 16GB 的硬盘空间。Vivado 2019.2 支持以下操作系统:

- Windows 7.1: 64-bit
- Windows 10 Professional versions 1809 and 1903: 64-bit
- Red Hat Enterprise Linux 7.4-7.6: 64-bit
- CentOS Linux 7.4-7.6: 64-bit

- SUSE Enterprise Linux 12.4: 64-bit
- Amazon Linux 2 AL2 LTS: 64-bit
- Ubuntu Linux 16.04.5, 16.04.6, 18.04.1 and 18.04.2 LTS: 64-bit
- Additional library installation required

其他不同版本的安装和使用具体参考相应版本的用户指导手册。

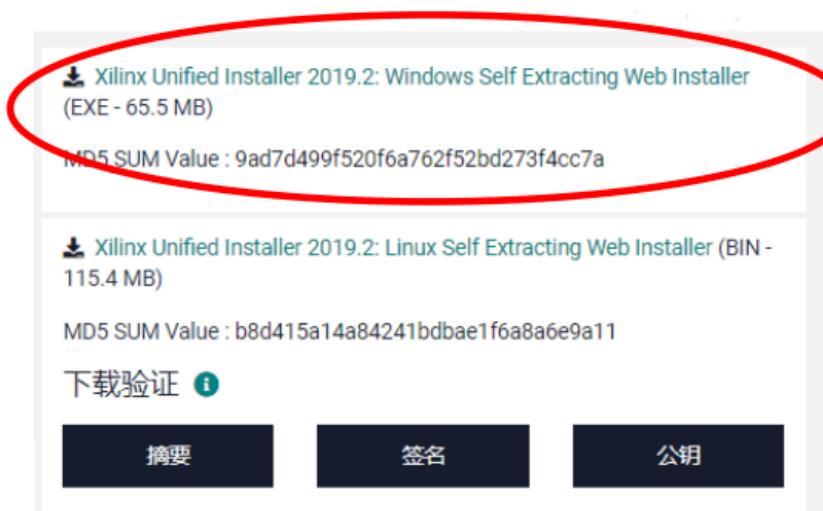
1.3.2 Vivado 安装文件的下载

在 Xilinx 官网 下载所需的 Vivado 版本。可以选择先下载安装 Web Installer，再通过安装器下载安装，可以减小下载时间和下载安装包大小；也可以直接下载安装包文件来安装，文件大小约 26GB。

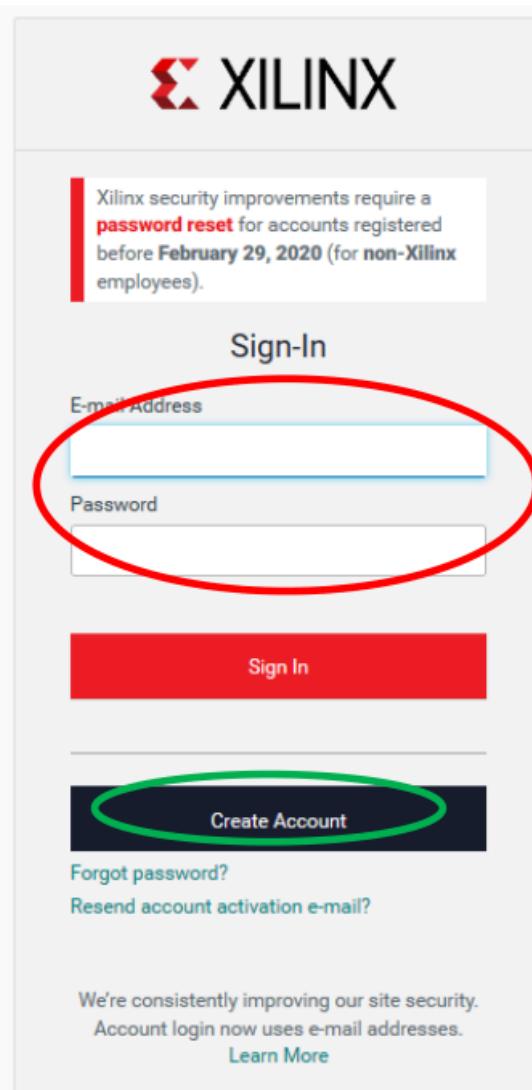
Vivado Design Suite - HLx 版本 - 2019.2

重要信息	下载内容包	Vivado Design Suite HLx Editions (All Editions)
Vivado Design Suite 2019.2 现已推出。	Last Updated	2019-11-12
• 在 Vivado Simulator (XSIM) 中引入 UVM 1.2 支持 • 提升 IP Integrator 中的层可见性 • 物理优化及其他 QoR 增强功能 • 设计编译运行时间减少 10% • 用于增强 Dynamic Function eXchange 的全新高带宽 ICAP IP	答案	2019.x - Vivado 已知问题
强烈建议您使用 web installer，它可缩短下载时间，还可节省大量的磁盘空间。 请查看 安装程序信息 ，了解详情。 注：只有 Google Chrome 和 Microsoft Internet Explorer 网络浏览器支持下载验证。	技术文档	发布说明 OS 支持更新 Vivado 的最新信息
	技术支持论坛	安装与许可

Vivado 设计套件提供支持 windows 系统、Linux 系统的在线安装器和支持双系统的本地安装包下载，选择相应的版本下载。这里选择 Windows 环境下的在线安装器（Windows Self Extracting Web Installer）。



下载需要登陆 Xilinx。如果已有 Xilinx 账户直接填写用户名和密码登陆，如果没有账户则点“Create account”免费创建一个新账户。



进行姓名和地址验证。输入信息后点网页底部的“下载”。

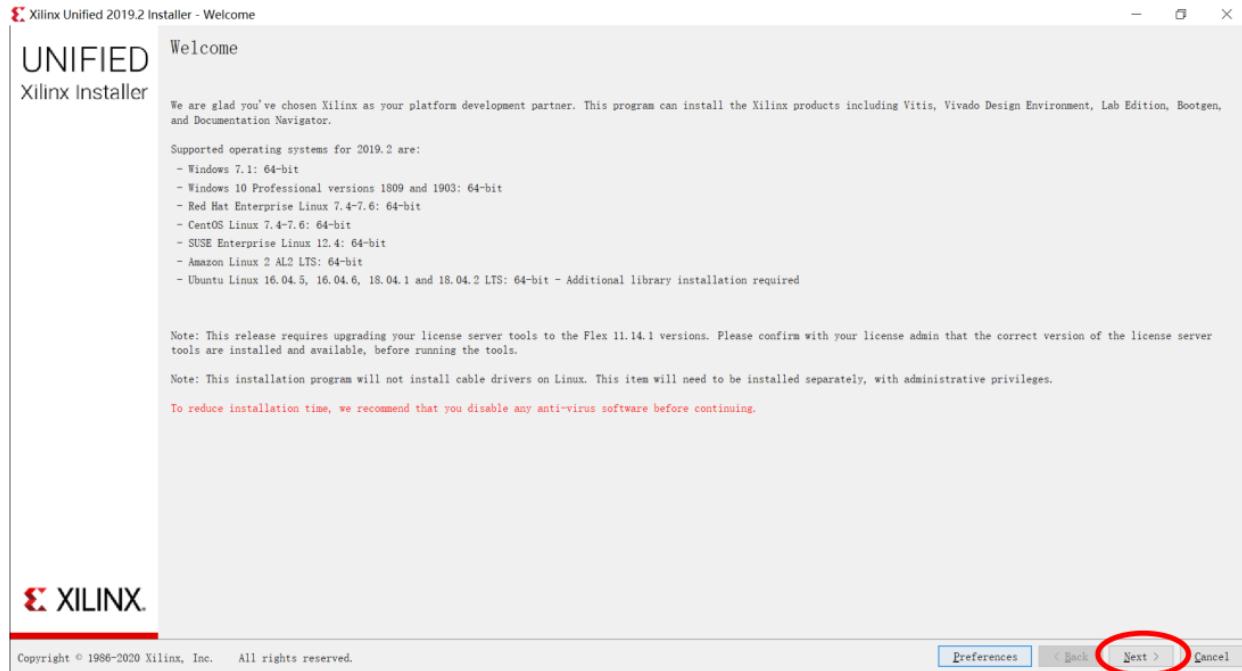
A screenshot of a download approval form. It features a section titled "美国政府出口批准" (U.S. Government Export Approval) containing two bullet points: "美国政府出口法律法规规定在满足您的下载需求前须验证您的名、姓、公司名称和收货地址。请提供正确完整的信息。" and "若地址包含带有非罗马字符(如如沉音符、波浪符或冒号)的邮政信箱和姓名，将无法通过美国出口合规系统。". Below this, there are two input fields labeled "姓 (中文)" and "名 (中文)" with their respective text boxes.

此时弹出保存 Vivado 安装包的窗口，将安装包保存在合适的地方。

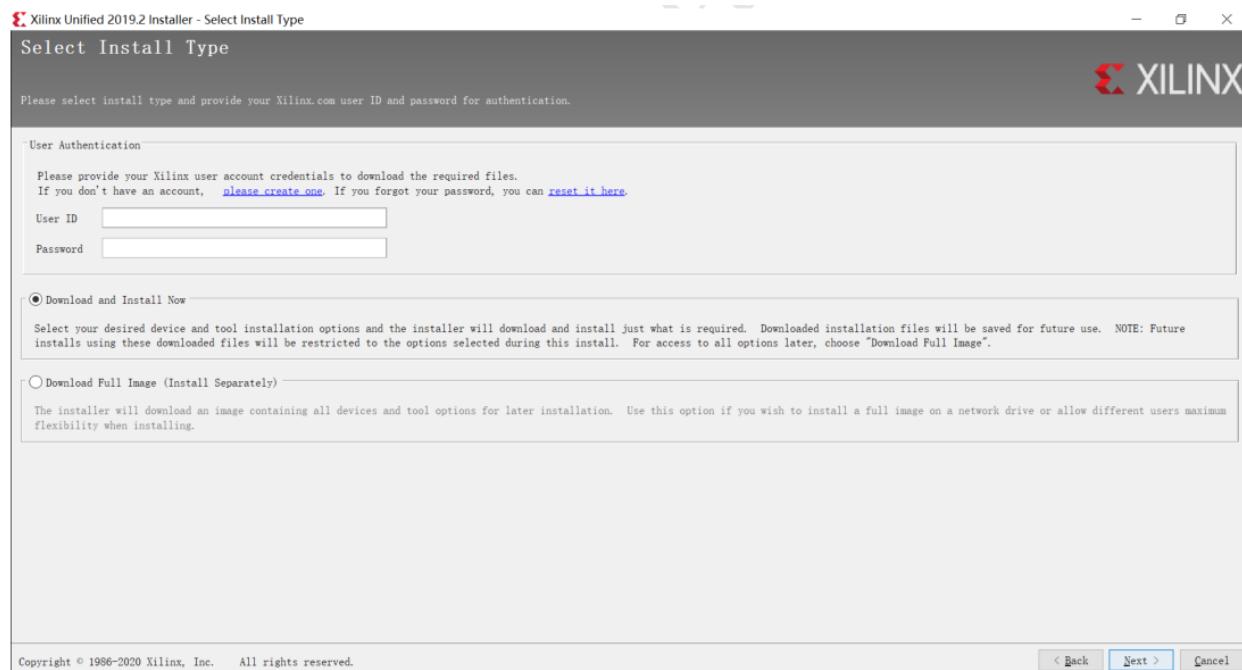
1.3.3 Vivado 在线安装

下载后，双击运行已下载的可执行文件 Xilinx_Unified_2019.2_1106_2127_Win64.exe。

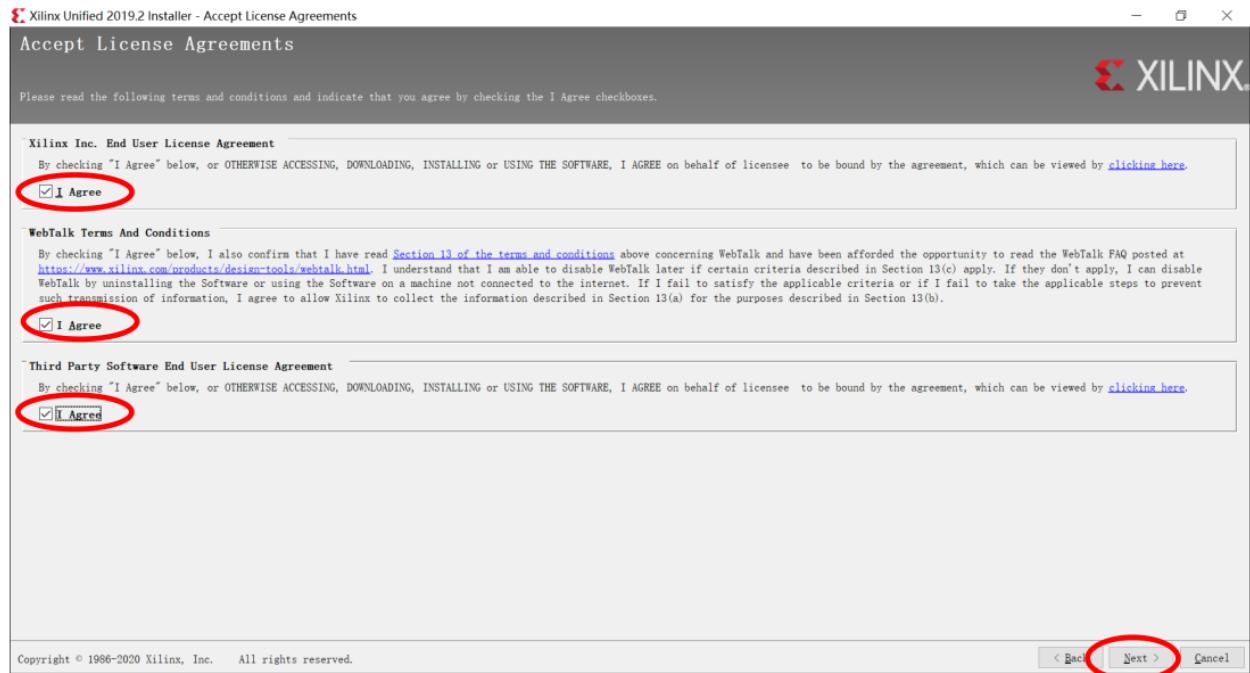
欢迎界面，点击“Next”。



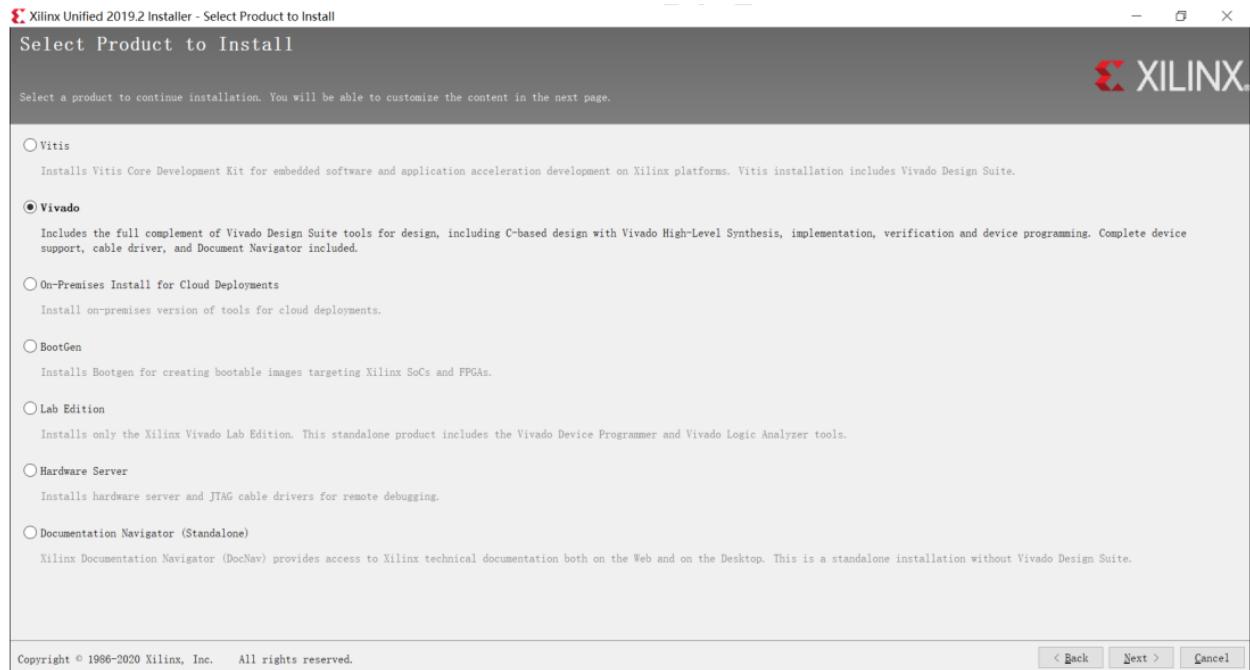
输入 Xilinx 账户、密码，选择“Download and Install Now”，点击“Next”。



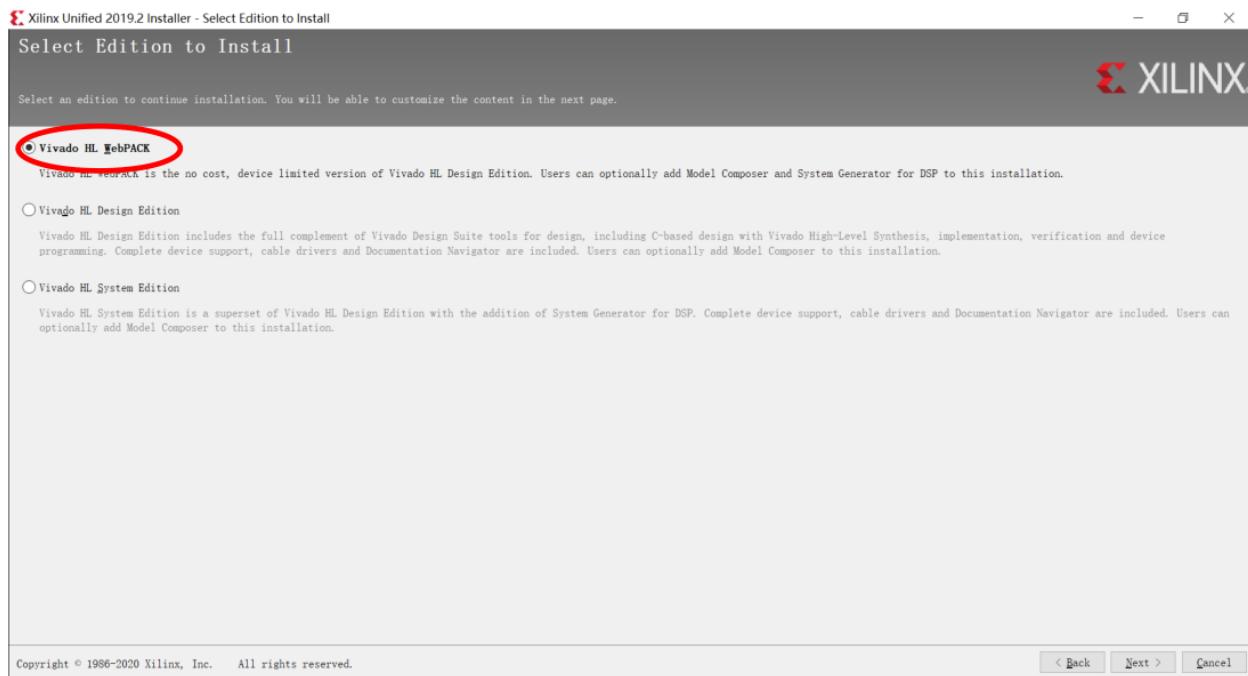
勾选所有“*I Agree*”选项，点击“Next”。



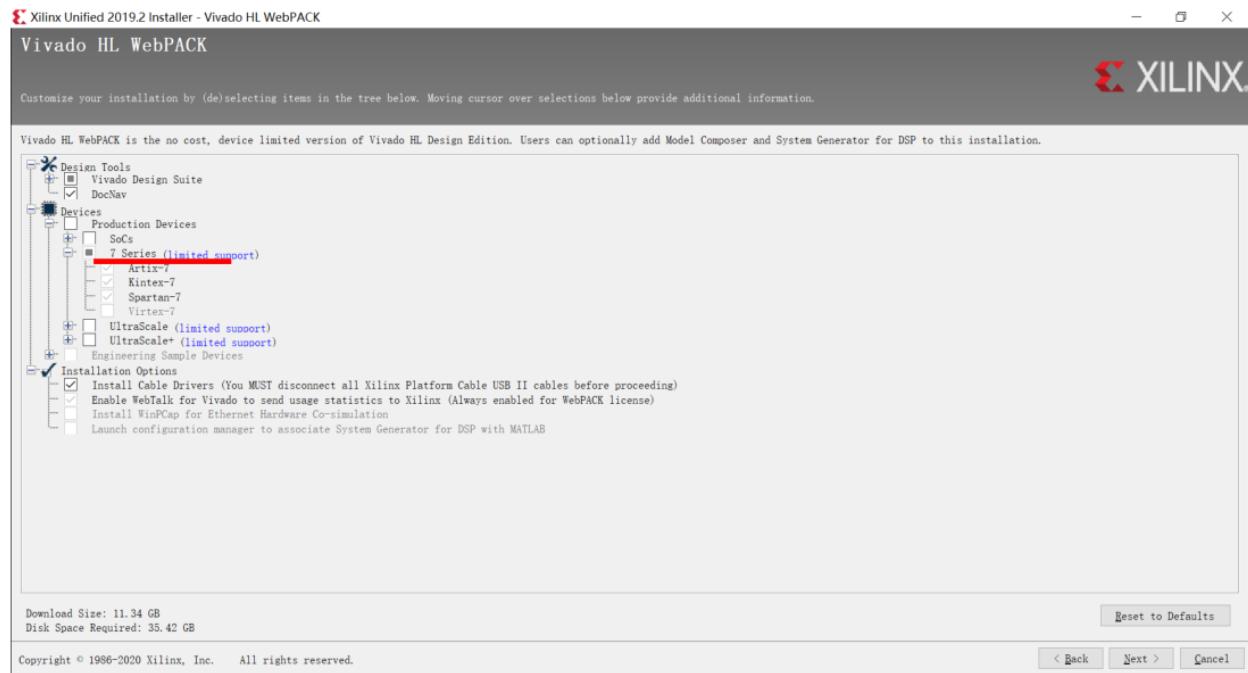
选择 Xilinx 的产品，这里勾选 Vivado，点击“Next”。



选择 Vivado 安装版本，这里勾选 Vivado HL WebPACK 版本，点击“Next”。

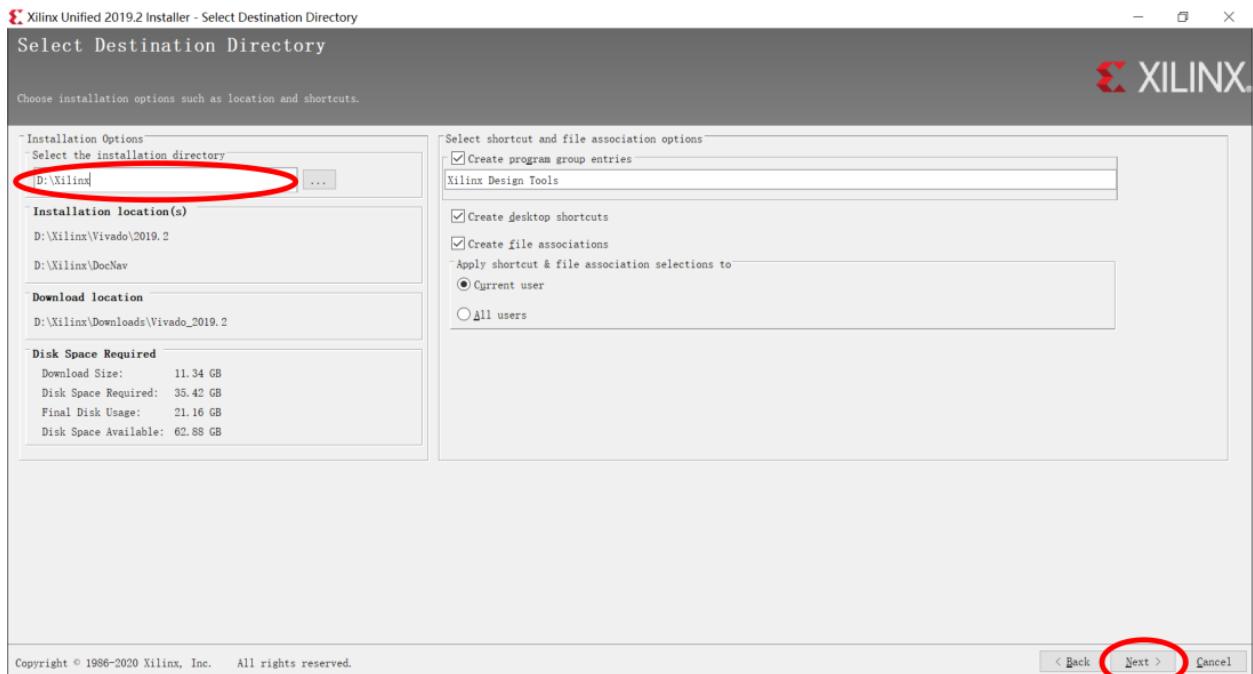


选择设计工具、支持的器件。“Design Tools”默认已选择“Design Vivado Suite”和“DocNav”；“Device”默认选择“Artix-7”，正好开发板搭载的FPGA是Artix-7，其他器件可以根据需要进行选择；“Installation Options”按照默认即可。点击“Next”。

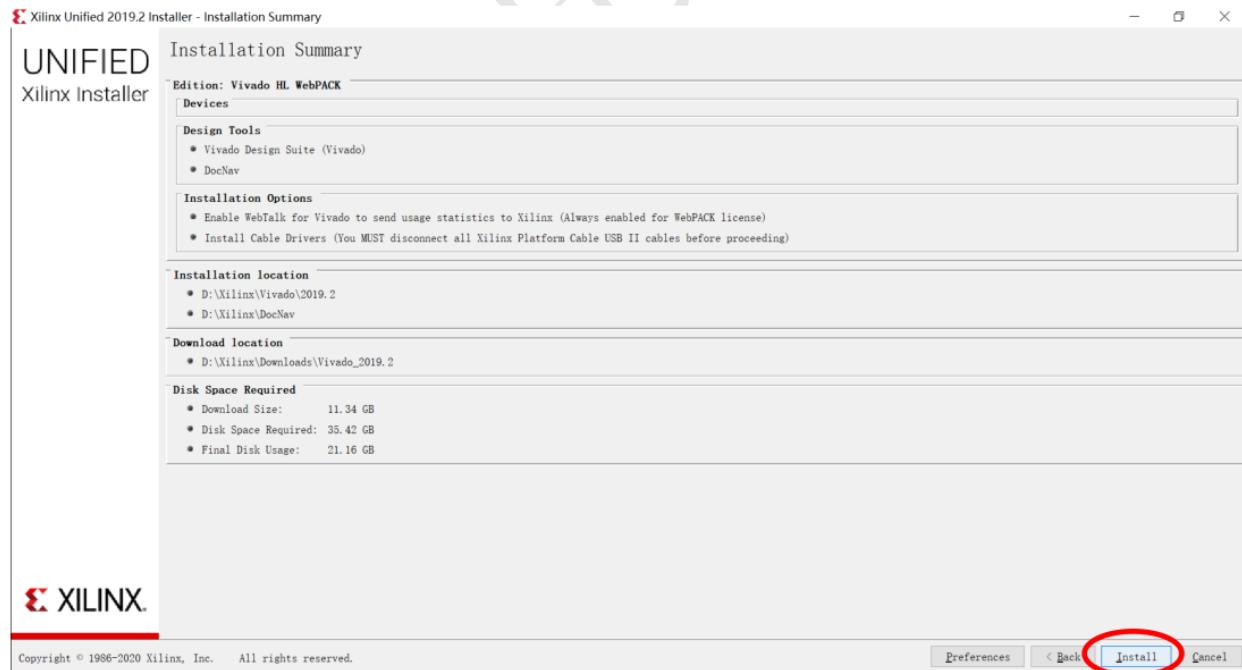


选择 Vivado 安装目录，默认安装在“C:\Xilinx”下，可以点击浏览或者直接更改路径。点击“Next”。

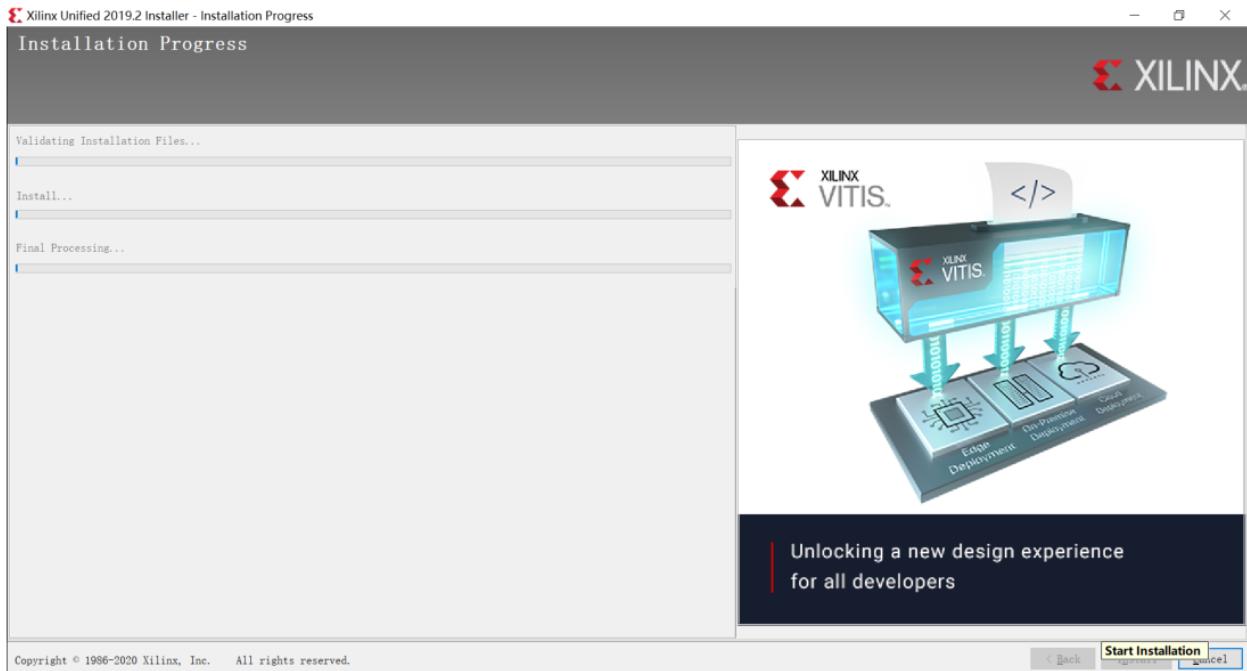
备注：安装路径中不能出现中文和空格。



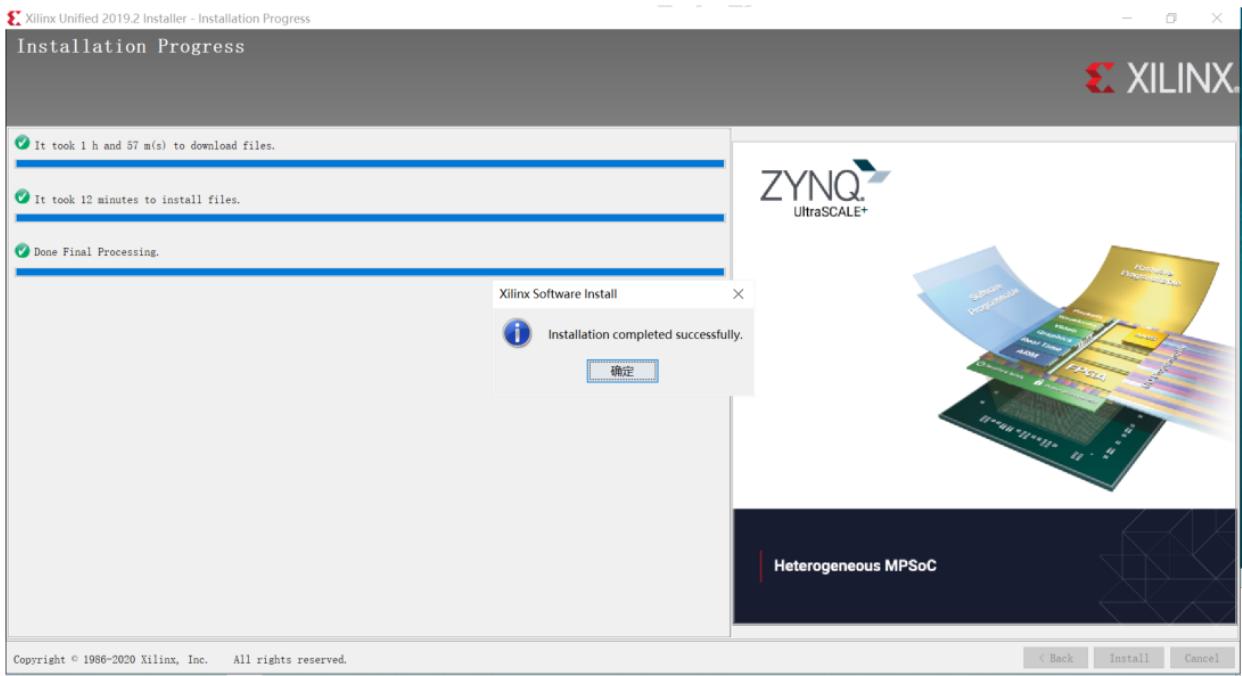
确认无误，点击“Install”开始安装；如果要修改安装设置，点击“Back”返回到相应的界面修改。



安装进度窗口，等待安装完成。



安装成功后会出提示窗口，点击“确定”。



1.3.4 辅助工具

我们推荐使用以下工具：

- Visual Studio Code, 现代化的代码编辑工具
- VS Code 插件: [Verilog-HDL](#)
- [Icarus Verilog](#)或者 Verilator on WSL, 除了仿真外, 二者都可以配合插件进行语法检查
- Ctags, 配合插件可以一键例化, 减少工作量

这些工具能够帮助同学们更加高效地使用 vivado, 同学们可以选择性地进行安装。当然, 仅使用 vivado 足以完成实验的所有内容。

1.4 Vivado 使用说明

通过本章节的学习, 你将获得:

- 初步熟悉龙芯 Artix-7 实验箱。
- 基本的 Vivado 使用方法。
- 初步了解 Vivado 的 XDC 约束文件。

Vivado 有两种开发模式, Project Mode 和 Non-Project Mode。两者的主要区别是, Project Mode 使用 Flow Navigation 更加自动化, Non-Project Mode 使用 Tcl 命令直接控制操作, 更手动但更不受限制。实际中根据开发需求选择这两种模式, 一般简单设计中采用 Project Mode。

下面以一个简单的流水灯实验为例介绍如何使用 Vivado 新建一个工程。

1.4.1 FPGA 设计流程简介

FPGA 即现场可编程门阵列, 不过 FPGA 内部最小单元其实并不是一个个与或非门, 而是一些更高层的模块(查找表、触发器、进位链等)。

Vivado 的作用是, 将我们的 Verilog 源码映射到 FPGA 芯片上, 用一种类似搭积木的方式实现我们的设计。

和编译一个 C 程序需要编译链接一样, 编译一个 Verilog 设计也需要若干步骤:

- 第一步: 综合 (Synthesize), 将 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 (网表), 并根据目标和要求 (约束条件) 优化所生成的逻辑连接。
- 第二步: 实现 (Implementation)。将综合输出的逻辑网表翻译成所选器件的底层模块与硬件原语, 将设计映射到器件结构上, 进行布局布线, 达到在选定器件上实现设计的目的。

其中实现又可分为 3 个步骤: 翻译 (Translate) 逻辑网表, 映射 (Map) 到器件单元与布局布线 (Place & Route)。

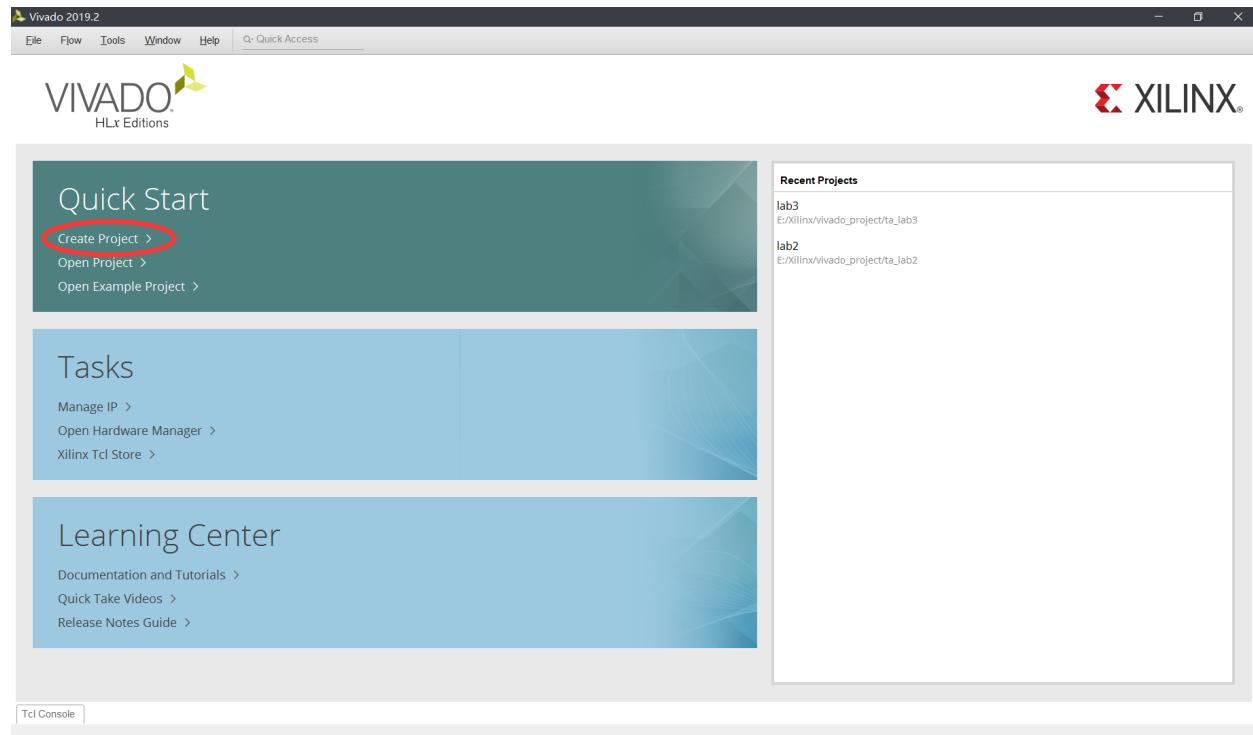
- 翻译: 将综合输出的逻辑网表翻译为 Xilinx 特定器件的底层结构和硬件原语
- 映射: 将设计映射到具体型号的器件上 (LUT、FF、Carry 等)。
- 布局布线: 根据用户约束和物理约束, 对设计模块进行实际的布局, 并根据设计连接, 对布局后的模块进行布线, 产生比特流文件。

目前多数 FPGA 的逻辑块都成二维阵列状排列, 因此逻辑块布局问题可以被视为二次分配问题, 是一种 NP 问题, 因此只能使用模拟退火算法等方法得到符合要求的近似解 (就算如此耗时也很长)。这就是为什么我们将管脚分配、时钟定义等称为约束。

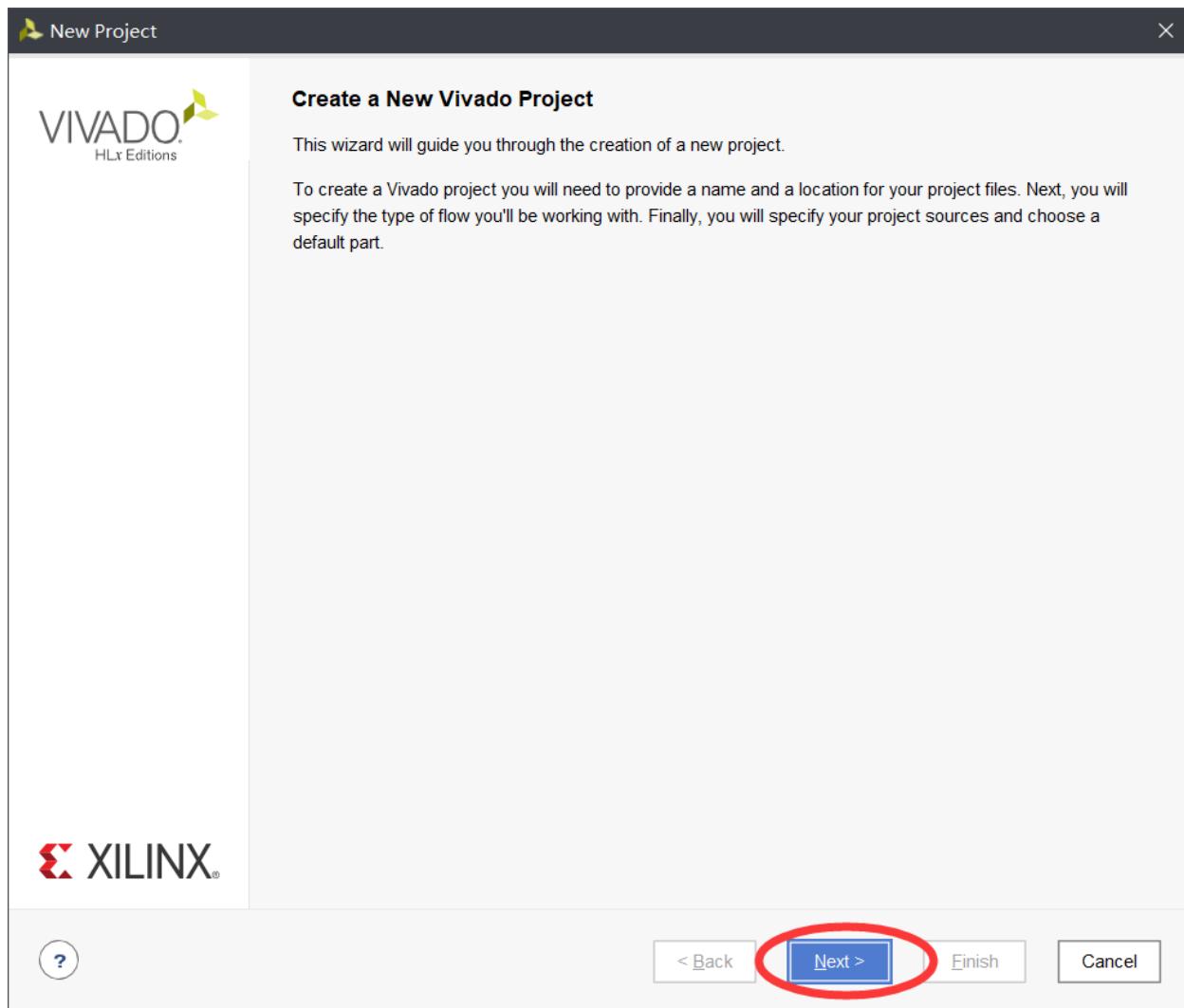
1.4.2 使用 Vivado 进行 FPGA 设计流程

新建工程

打开 Vivado 2019.2，在界面上“Quick Start”下选择“Create Project”。

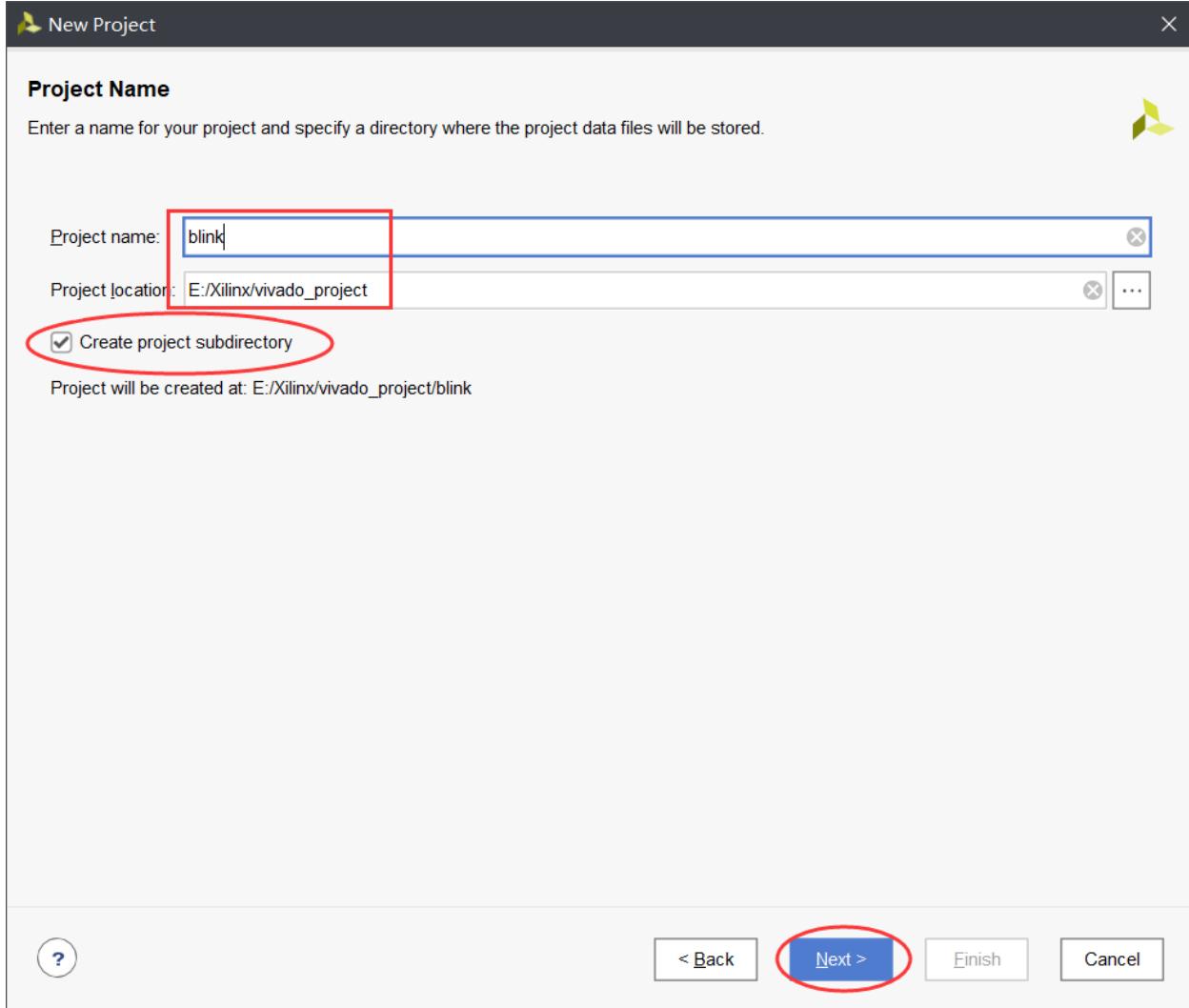


新建工程向导，点击“Next”。

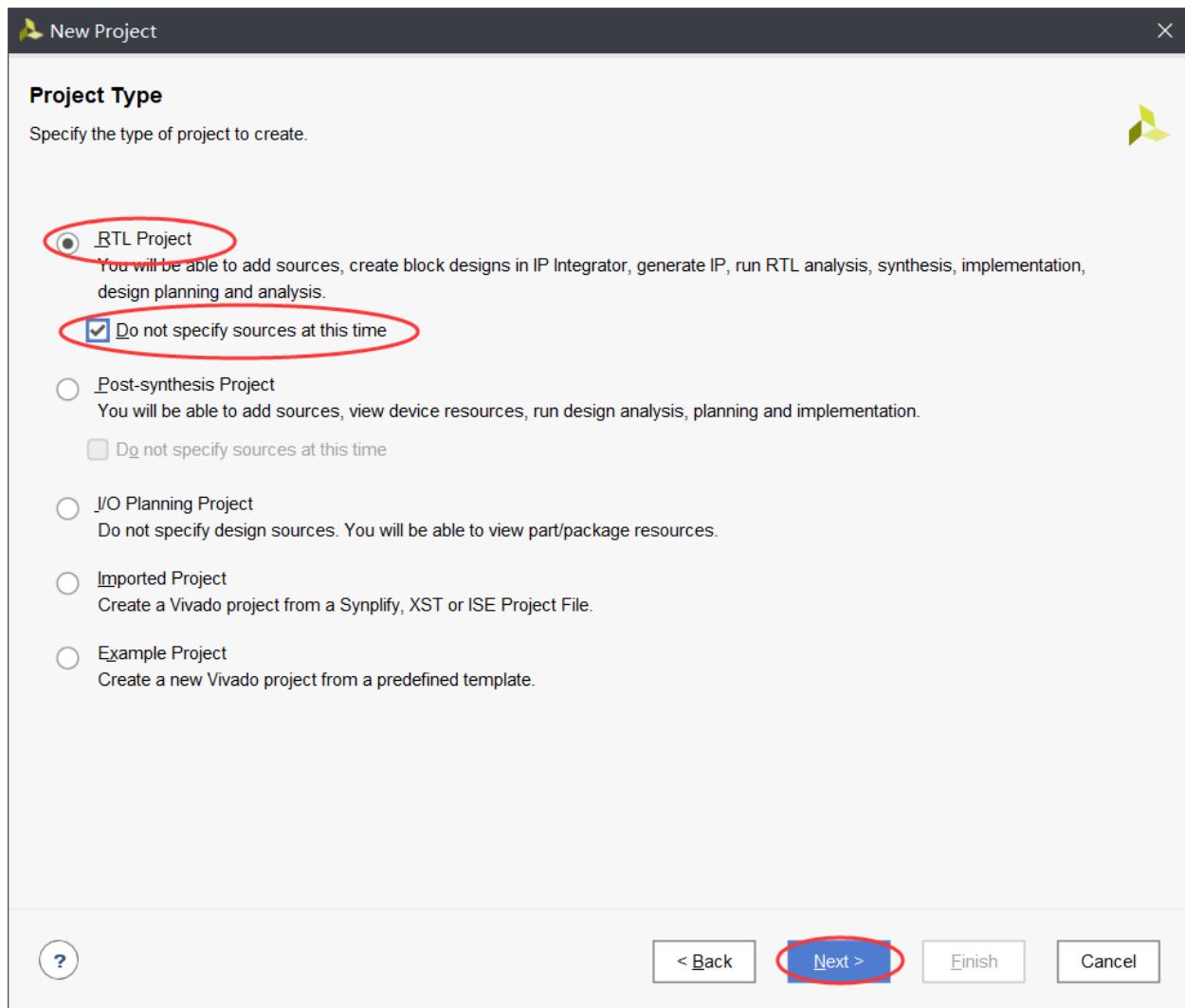


输入工程名称并选择工程的文件位置，并勾选“Create project subdirectory”选项，为工程在指定存储路径下建立独立的文件夹。设置完成后，点击“Next”。

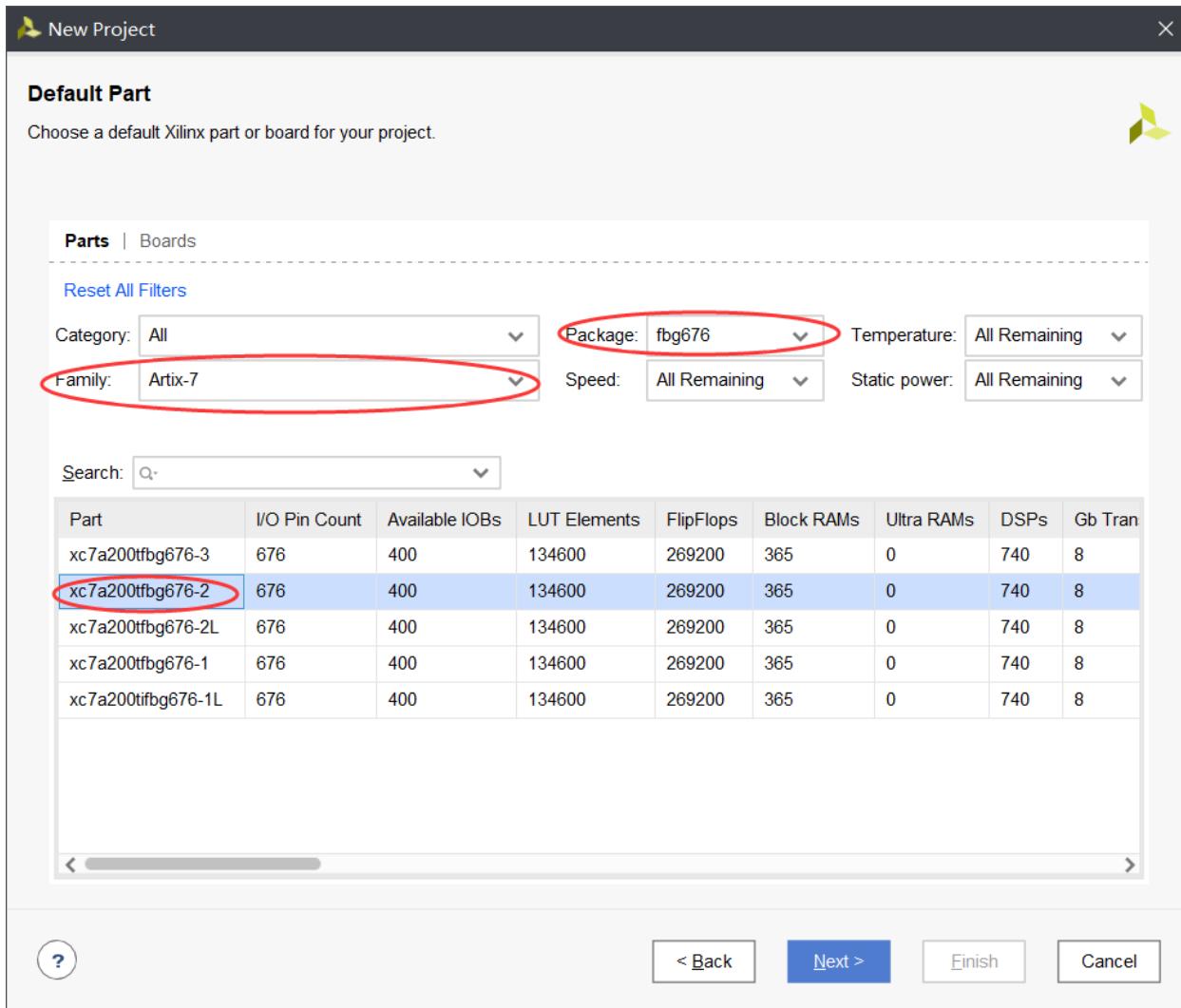
备注：工程名称和存储路径中不能出现中文和空格，建议工程名称以字母、数字、下划线来组成。



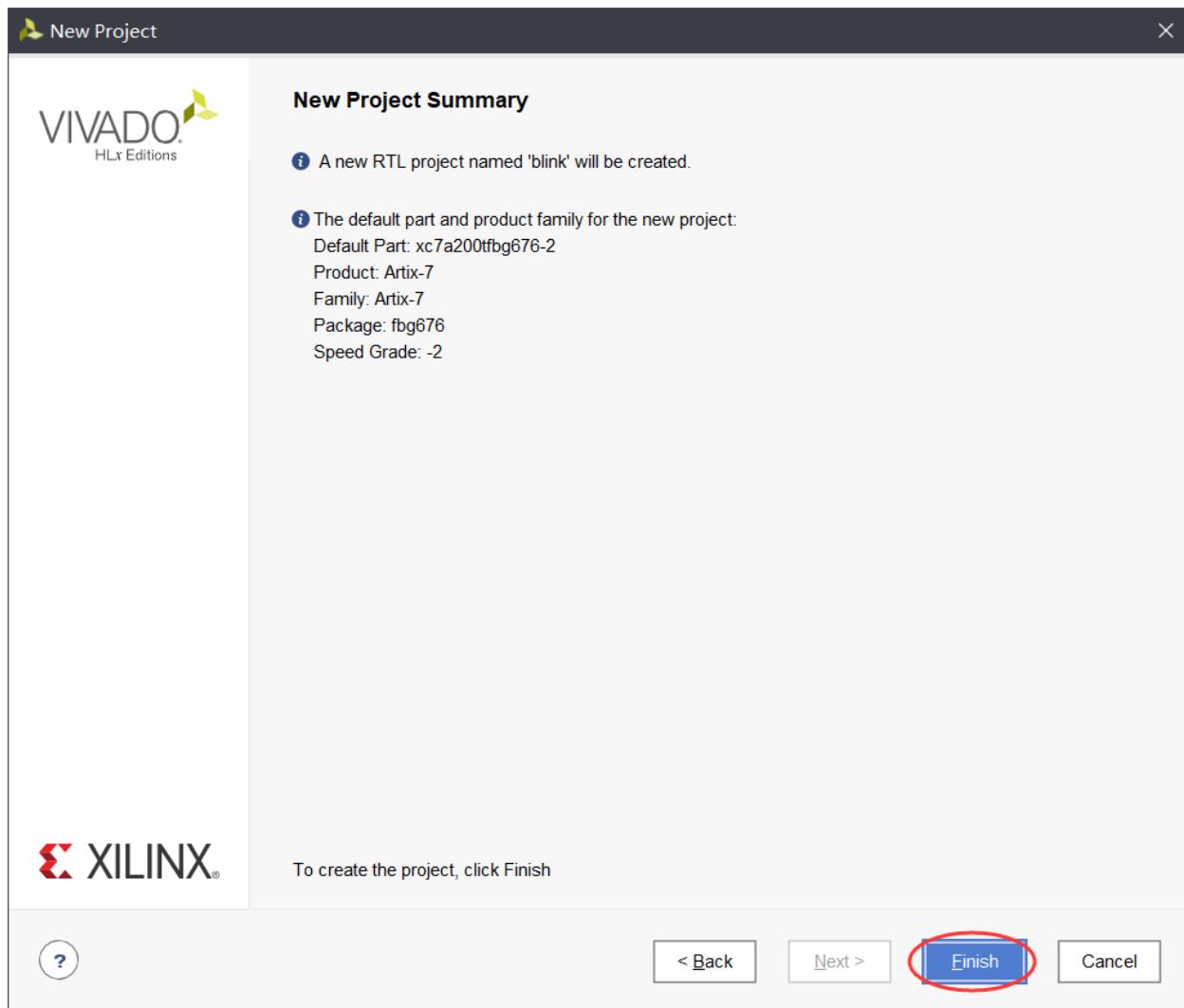
选择“RTL Project”一项，并勾选“Do not specify sources at this time”，勾选该选项是为了跳过在新建工程的过程中添加设计源文件，如果要在新建工程时添加源文件则不勾选。点击“Next”。



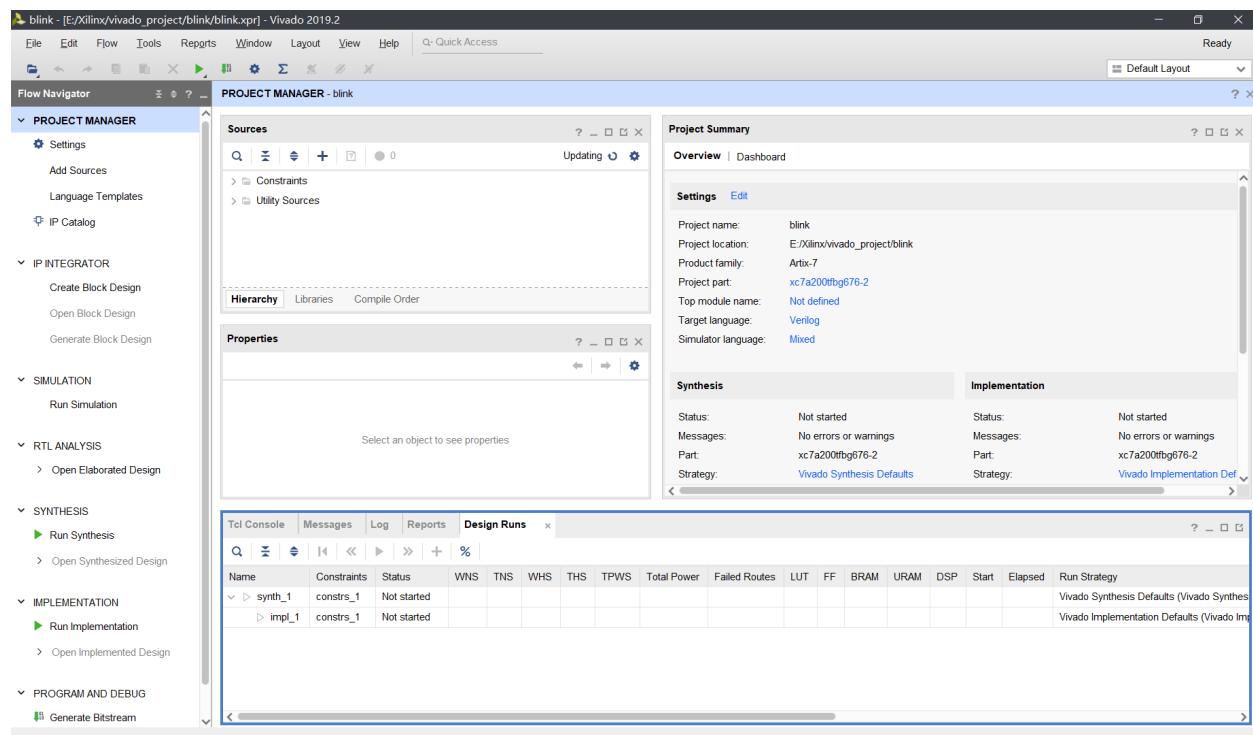
根据使用的 FPGA 开发平台，选择对应的 FPGA 目标器件。根据实验平台搭载的 FPGA，在筛选器的“Family”选择 Artix 7，“Package”选择 fbg676，在筛选得到的型号里面选择 xc7a200tfbg676-2。点击“Next”。



确认工程设置信息是否正确。正确点击“Finish”，不正确则点击“上一步”返回相应步骤修改。



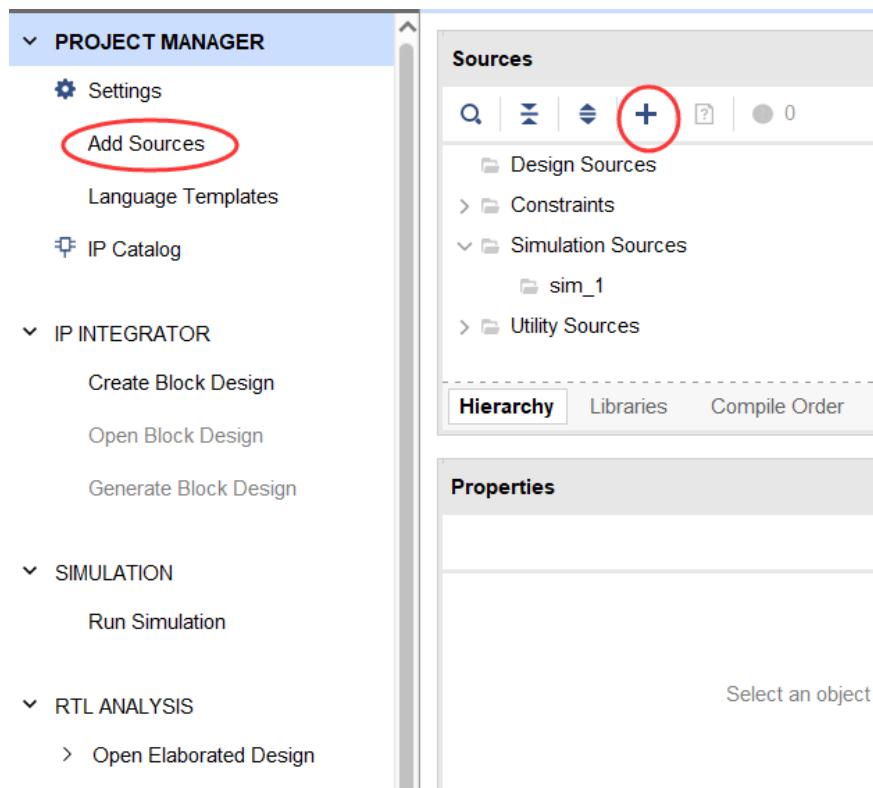
完成工程新建。



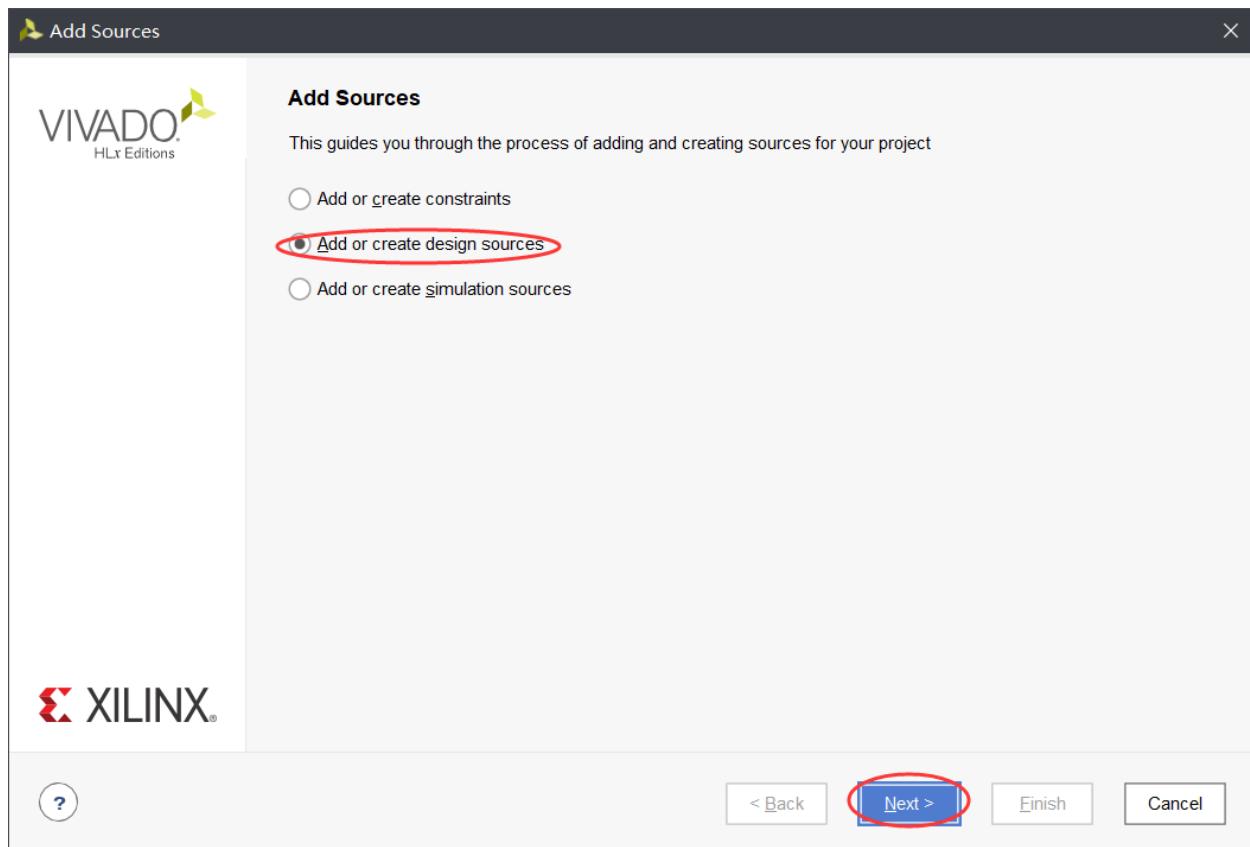
RTL 设计输入

以使用 Verilog 完成 RTL 设计为例。Verilog 代码都是以 “.v” 为后缀名的文件，可以在其他文件编辑器里写好，再添加到新建的工程中，也可以在工程中新建一个再编辑。

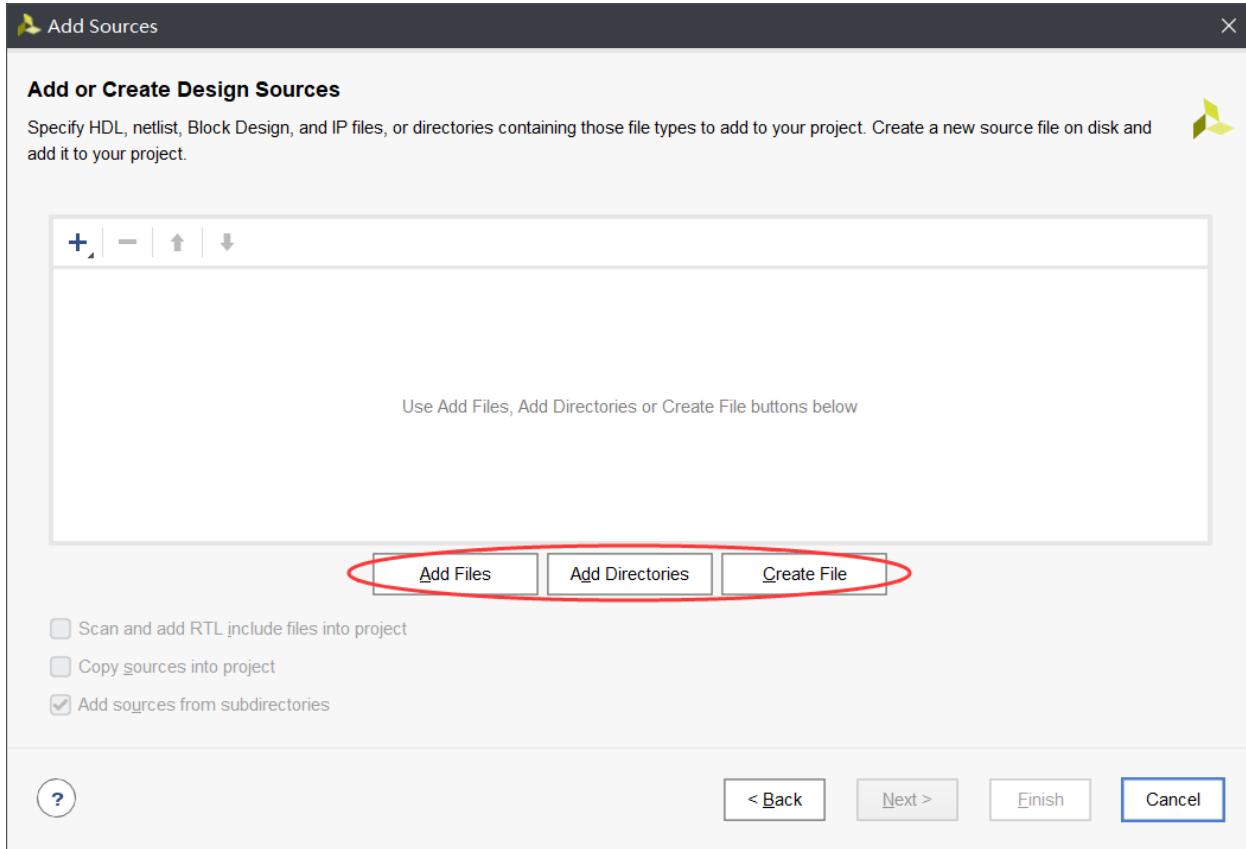
添加源文件。在 “Flow Navigator” 窗口下的 “Project Manager” 下点击 “Add sources”，或者点击 “Source” 窗口下 “Add Sources” 按钮，或者使用快捷键 “Alt + A”。



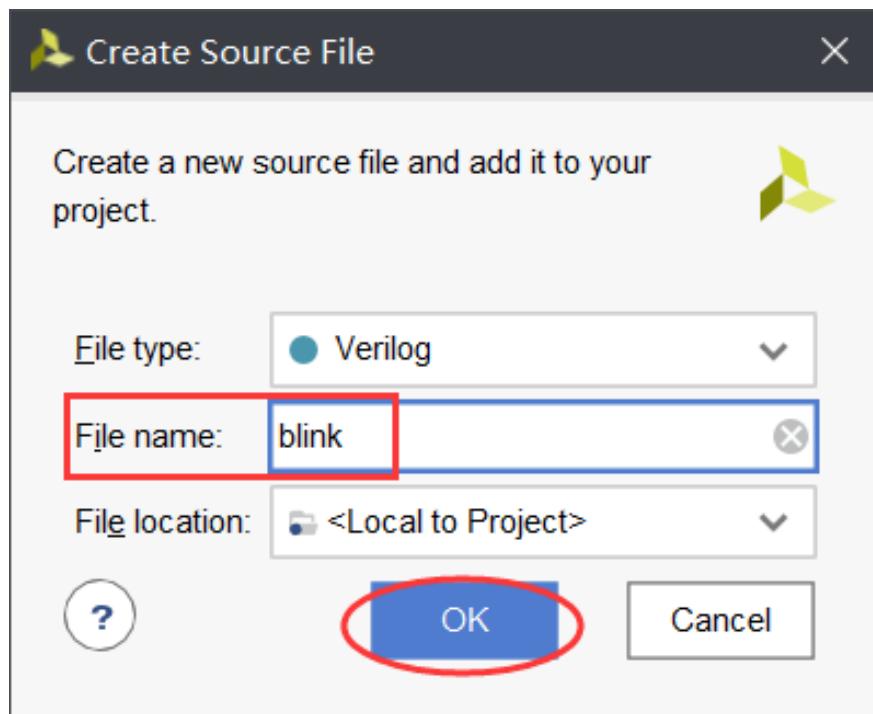
添加设计文件。选择“Add or create design sources”来添加或新建 Verilog 或 VHDL 源文件，点击“Next”。



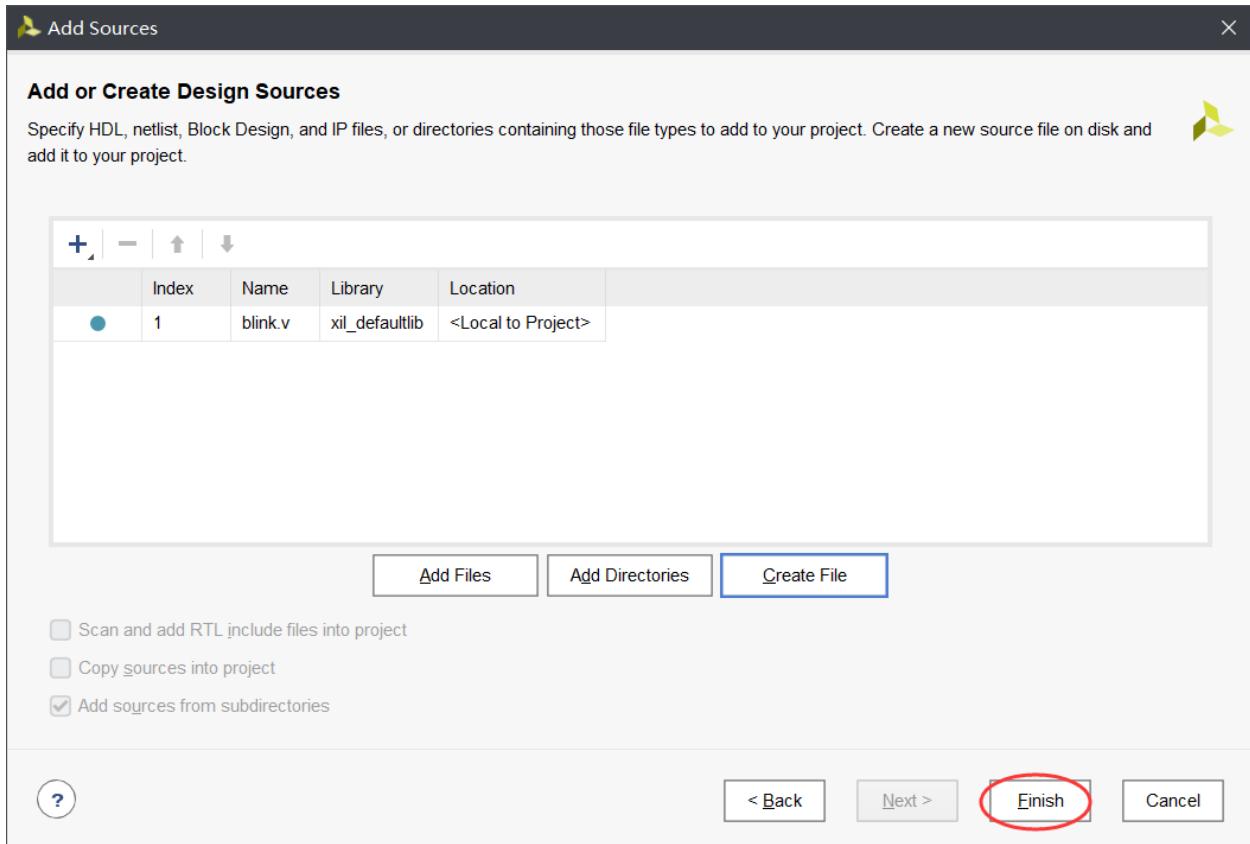
添加或者新建设计文件。如添加已有设计文件或者添加包含已有设计文件的文件夹，选择“Add Files”或者“Add Directories”，然后在文件浏览窗口选择已有的设计文件完成添加。如创建新的设计文件，则选择“Create File”。这里新建文件。



设置新创建文件的类型、名称和文件位置。注意：文件名称和位置路径中不能出现中文和空格。

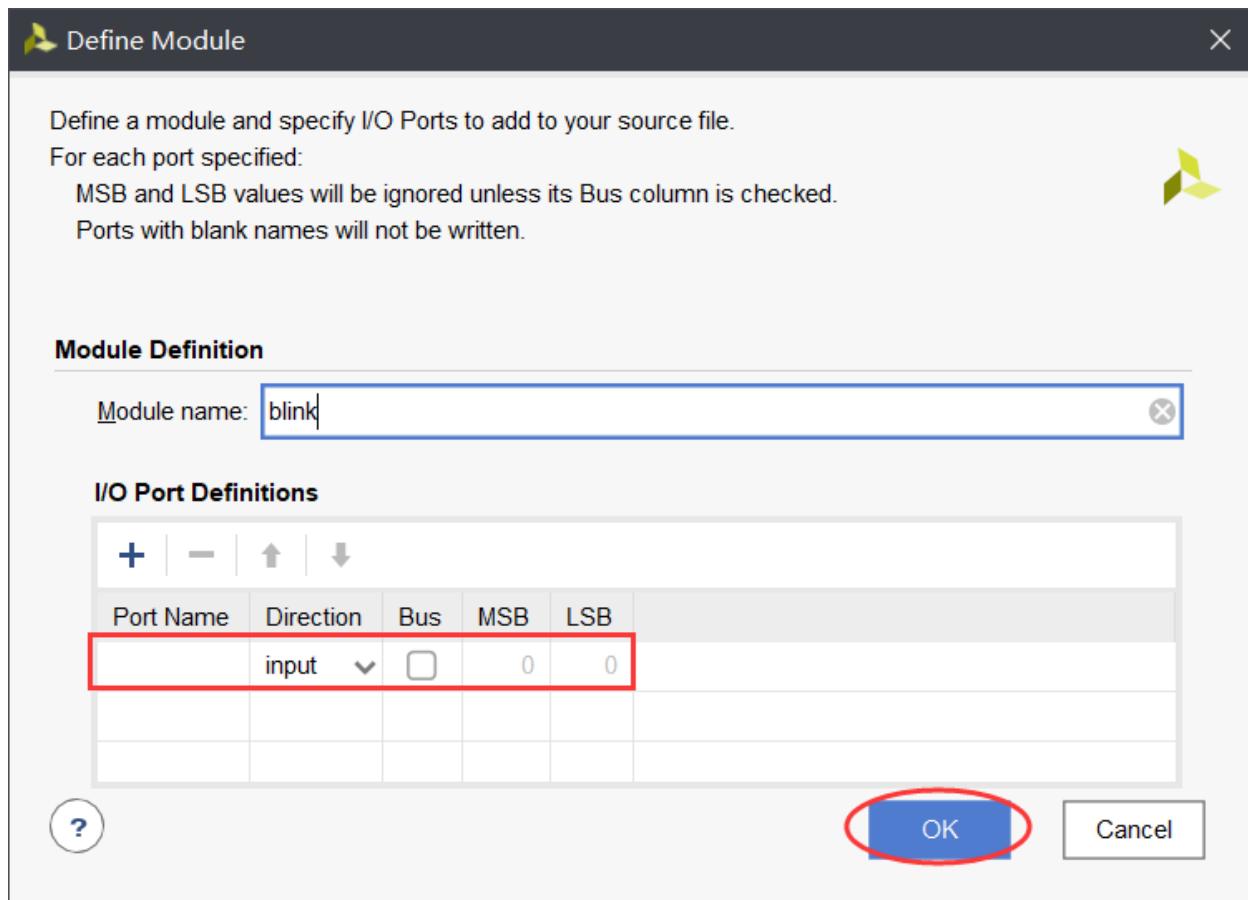


继续添加设计文件或者修改已添加设计文件设置。点击“Finish”。

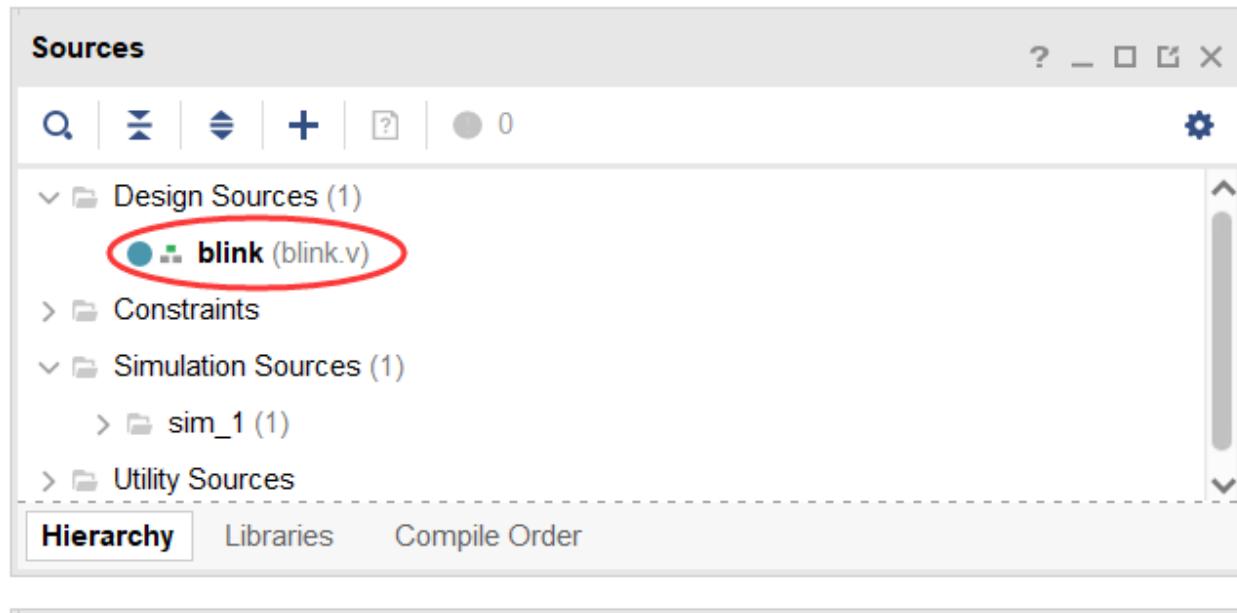


模块端口设置。在“Module Definition”中的“I/O Port Definitions”，输入设计模块所需的端口，并设置端口方向，如果端口为总线型，勾选“Bus”选项，并通过“MSB”和“LSB”确定总线宽度。完成后点击“OK”。

端口设置也可以在编辑源文件时完成，即可以在这一步直接点“OK”跳过。



双击“Sources”中的“Design Sources”下的“blink.v”中打开该文件，输入相应的设计代码。如果设置时文件位置按默认的，则设计文件位于工程目录下的“\blink.srsc\sources_1\new”中。完成的设计文件如下图所示。



以下给出 blink.v 示例文件。

```
// blink.v
`timescale 1ns / 1ps

module blink (
    input      clk ,
    input      reset,
    output [7:0] leds
);

    reg [7 :0] leds_reg;
    reg [26:0] cnt;

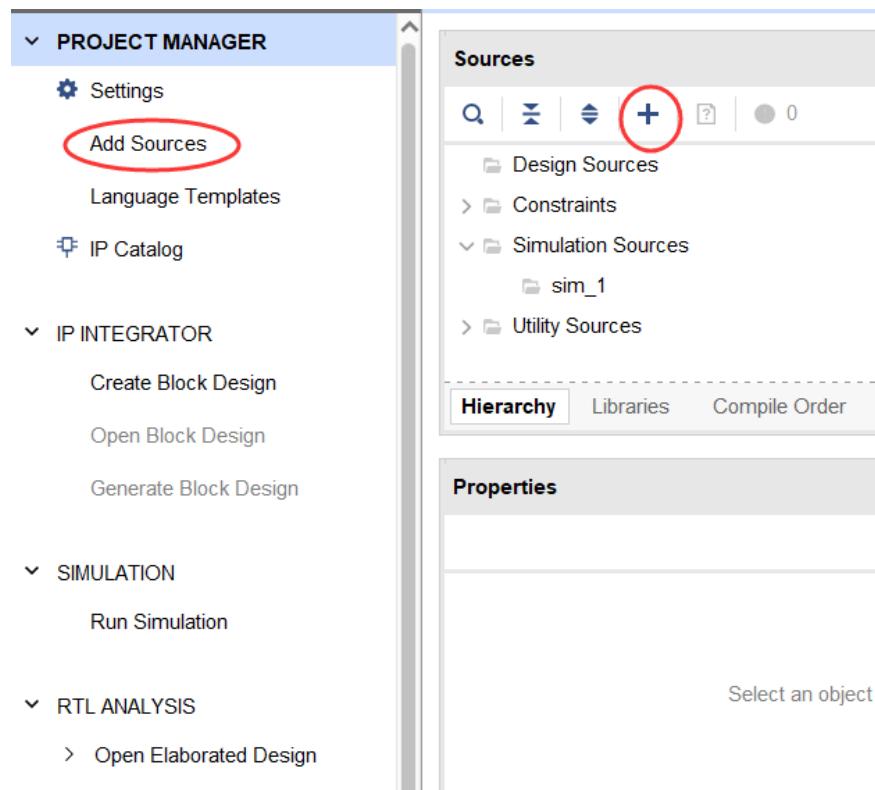
    assign leds = leds_reg;

    // clock freq = 100MHz
    always @ (posedge clk) begin
        if (!reset) begin
            leds_reg <= 8'b11111110;
            cnt      <= 27'd0;
        end else if (cnt == 27'd100000000) begin
            cnt      <= 27'd0;
            leds_reg <= {leds_reg[6:0], leds_reg[7]};
        end else begin
            cnt      <= cnt + 27'd1;
        end
    end
endmodule
```

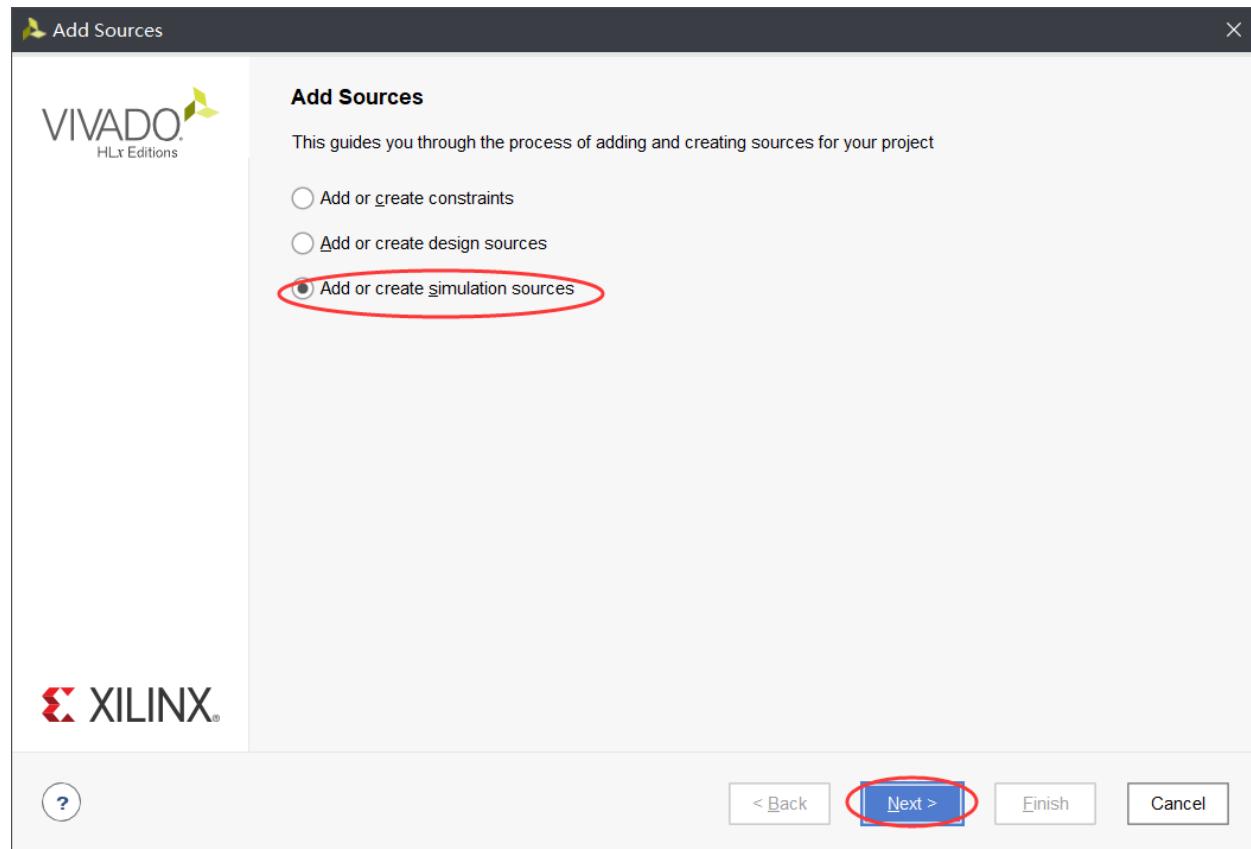
功能仿真

Vivado 集成了仿真器 Vivado Simulator，这里介绍使用 Vivado 仿真器仿真的方法。也可以使用 ModelSim 进行仿真。

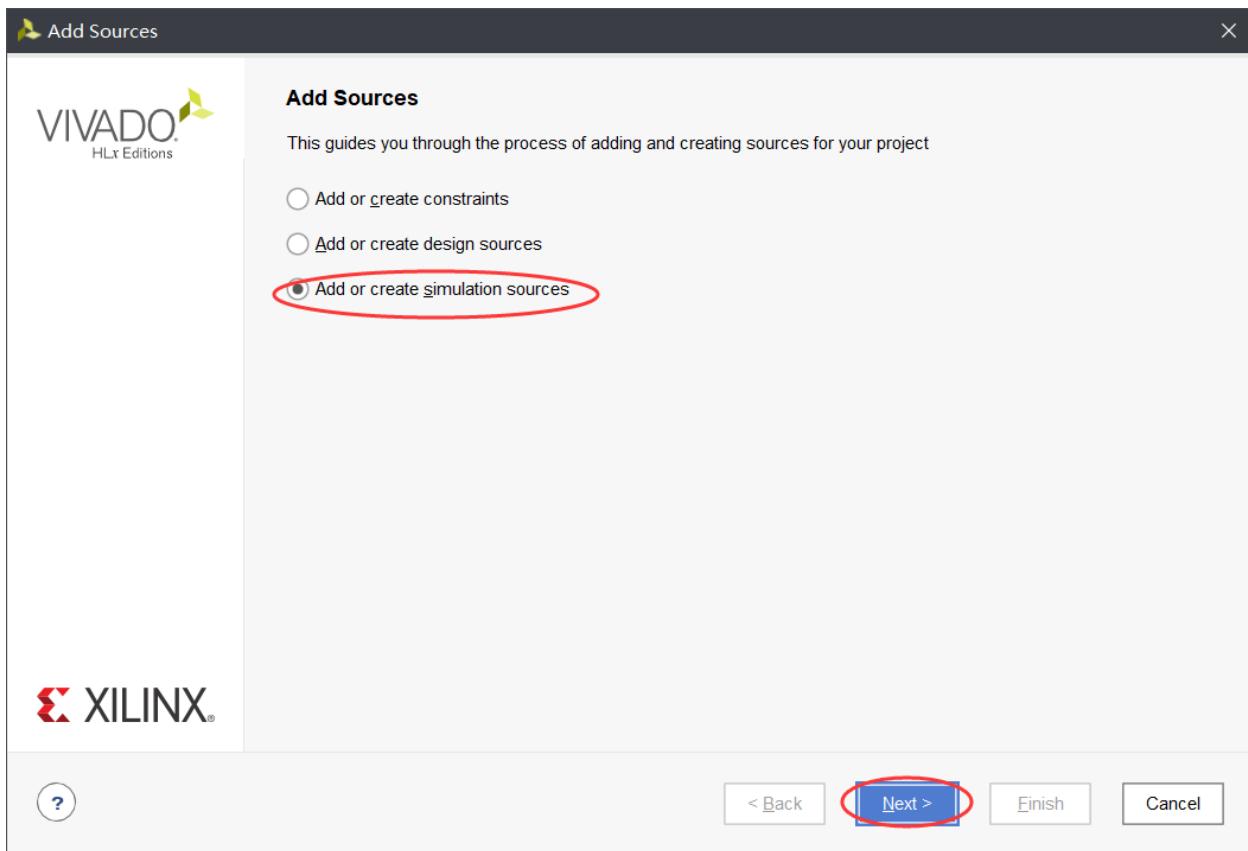
首先添加测试激励文件。该步骤与 3.2.2 RTL 设计输入中的第一步添加源文件类似。



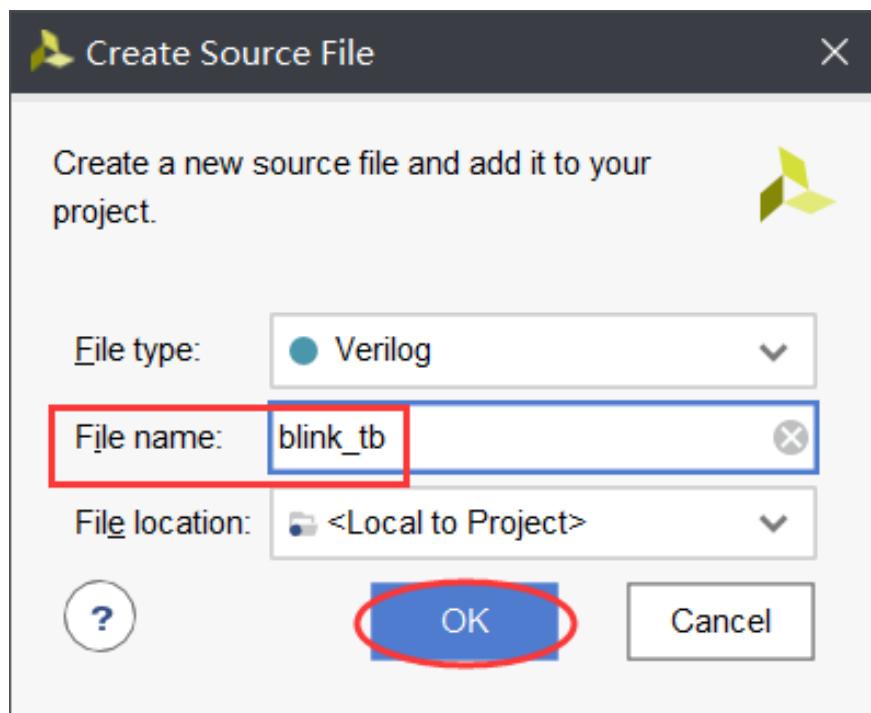
在“Add Source”界面中选择“Add or Create Simulation Sources”，点击“Next”。



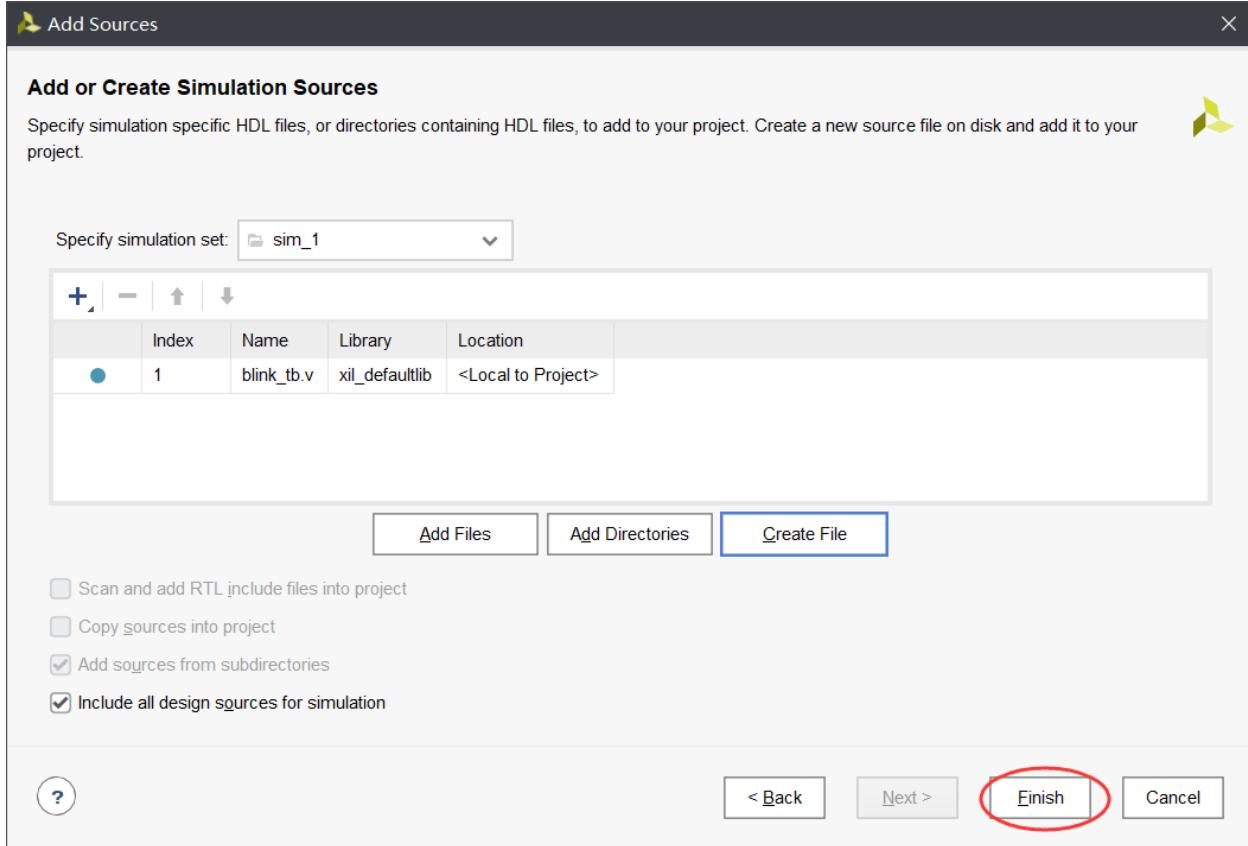
选择“Create File”，新建一个激励测试文件。



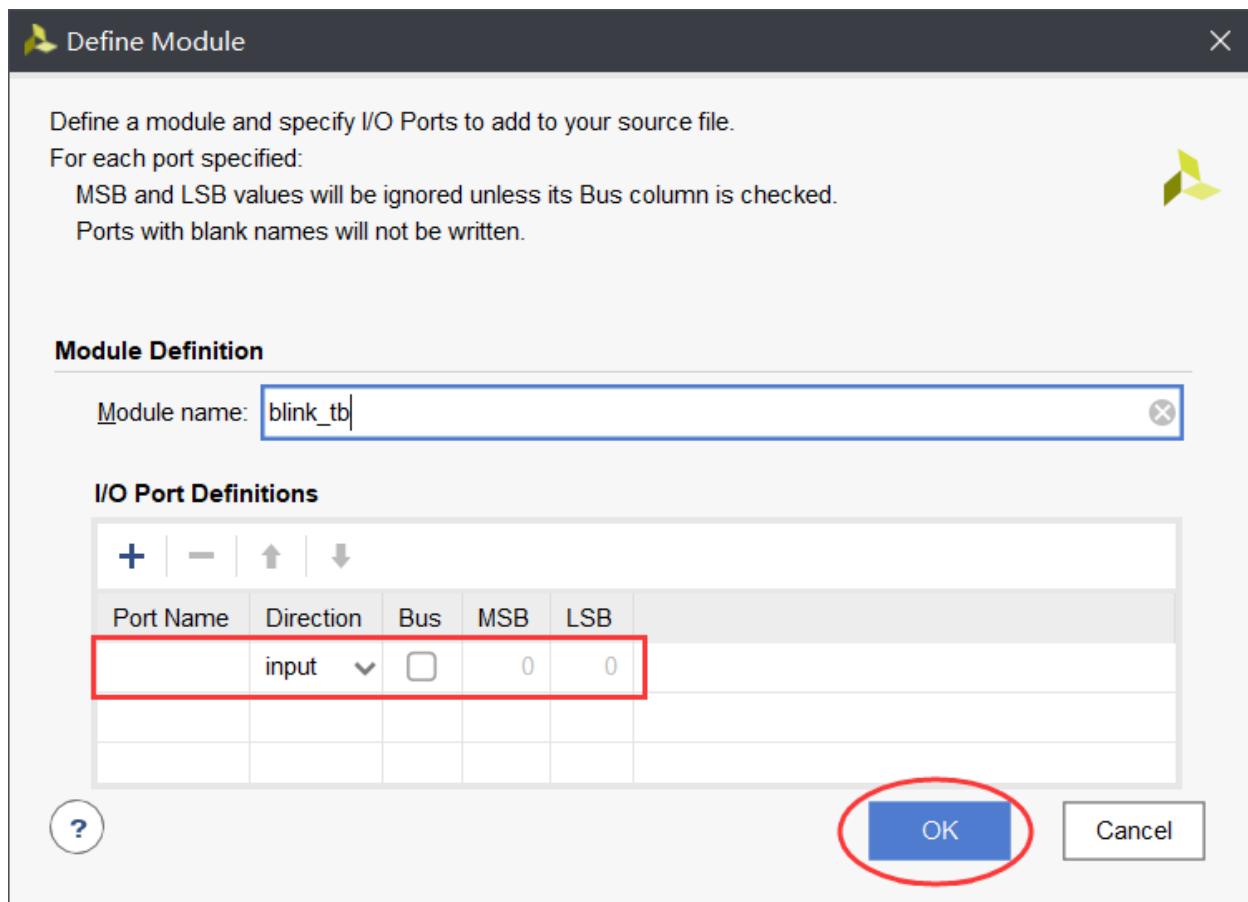
输入激励测试文件名，点击“OK”。



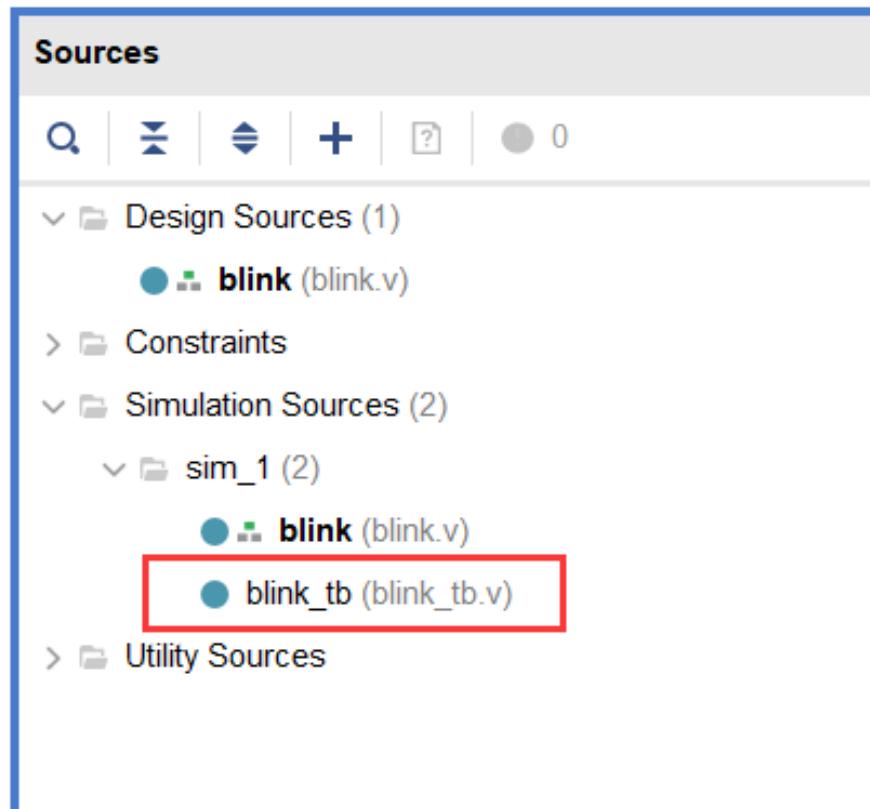
完成新建测试文件，点击“Finish”。



对测试激励文件进行 module 端口定义，由于激励测试文件不需要有对外的接口，所以不进行 I/O 端口设置直接点击“OK”，完成空白的激励测试文件创建。



在“Source”窗口下双击打开空白的激励测试文件。测试文件 blink_tb.v 位于工程目录下“\blink.srsc\sim_1\new”文件夹下。



完成对将要仿真的 module 的实例化和激励代码的编写，如下述代码所示。

```
// blink_tb.v
`timescale 1ns / 1ps

module blink_tb;

    reg clk, reset;
    wire [7:0] leds;

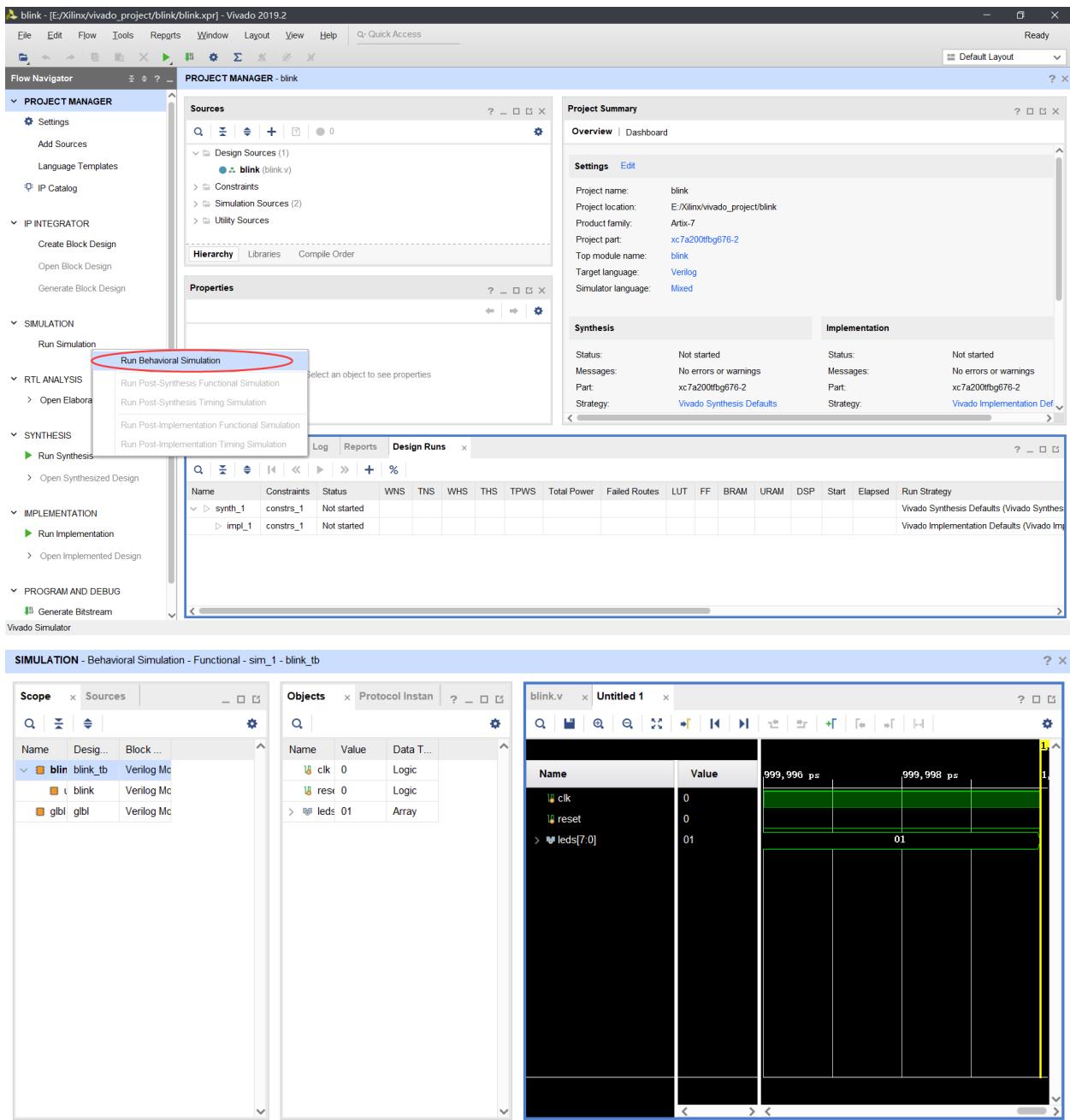
    initial begin
        clk = 0;
        reset = 0;
#50 reset = 1;
    end

    always #5 clk = ~clk;

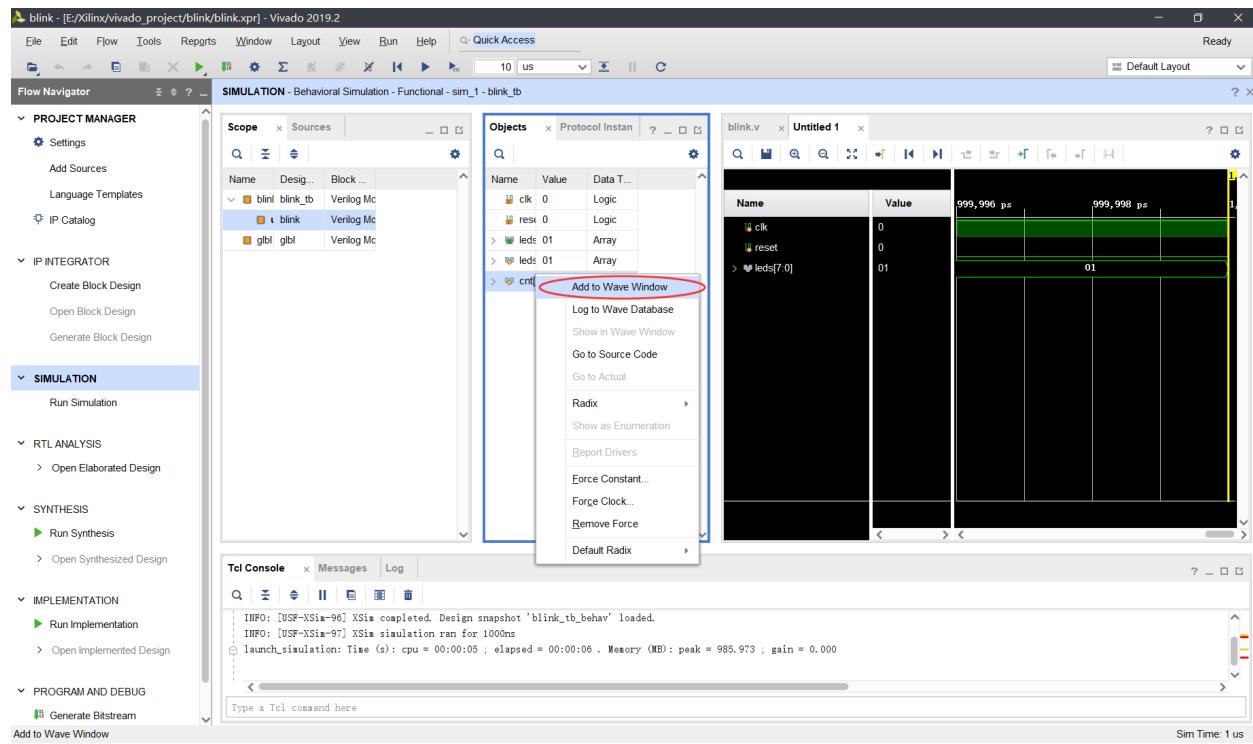
    blink u_blink(
        .clk    (clk    ),
        .reset (reset  ),
        .leds   (leds   )
    );

endmodule
```

进入仿真。在左侧“Flow Navigator”窗口中点击“Simulation”下的“Run Simulation”选项，选择“Run Behavioral Simulation”，进入仿真界面。



可通过左侧“Scope”一栏中的目录结构定位到想要查看的 module 内部信号，在“Objects”对应的信号名称上右击选择“Add To Wave Window”，将信号加入波形图中，也可以利用鼠标将信号直接拖拽入 Wave Window 中。仿真器默认显示 I/O 信号，由于这个示例不存在内部信号，因此不需要添加观察信号。



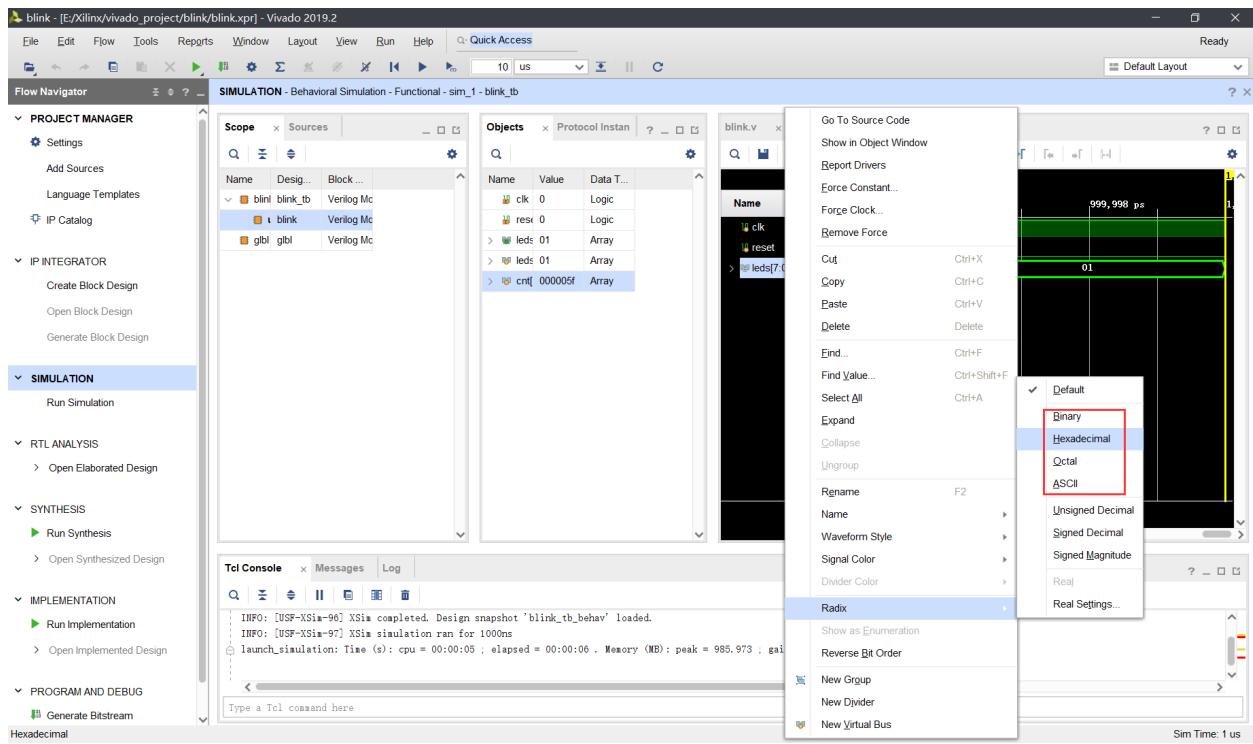
可通过选择工具栏中的选项来进行波形的仿真时间控制。如下图工具条，分别是复位波形（即清空现有波形）、运行仿真、运行特定时长的仿真、仿真时长设置、仿真时长单位、单步运行、暂停、重启动。



观察仿真波形是否符合预期功能。在波形显示窗口上侧是波形图控制工具，由左到右分别是：查找、保存波形配置、放大、缩小、缩放到全显示、缩放到光标、转到时间 0、转到时间的最后、前一个跳变、下一次跳变、添加标记、前标记、下一个标记、交换光标。



可通过右键选中信号来改变信号的显示形态。如下图将信号改为二进制显示。

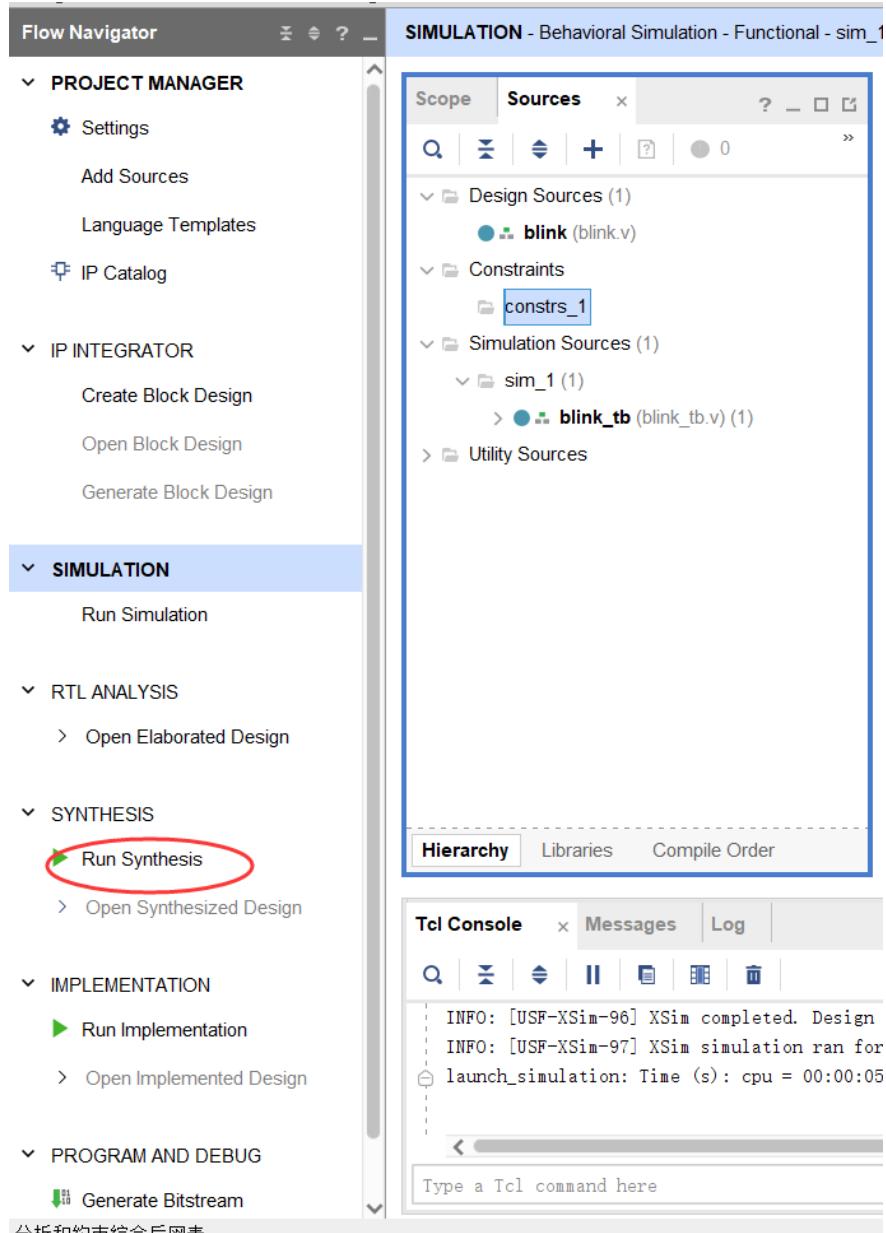


添加约束文件

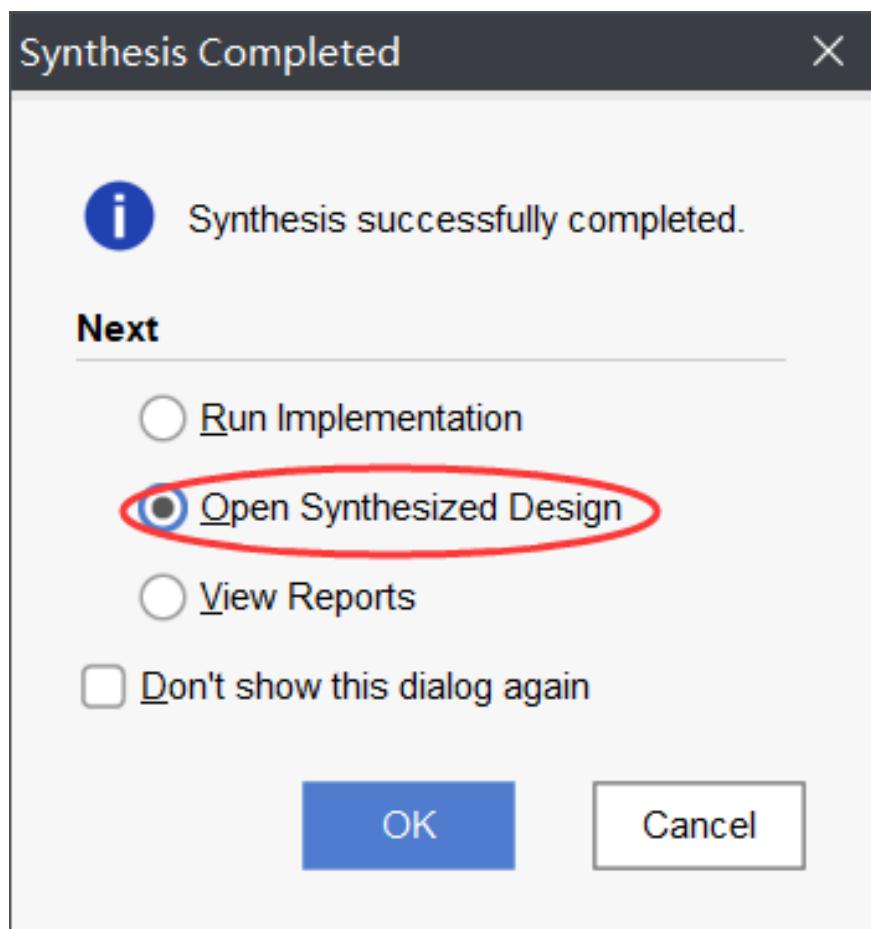
添加约束文件有两种方法，一是利用 Vivado 中 I/O Planning 功能，二是直接新建 XDC 的约束文件，手动输入约束命令。下面分别介绍这两种方法。

利用 I/O Planning 生成约束文件

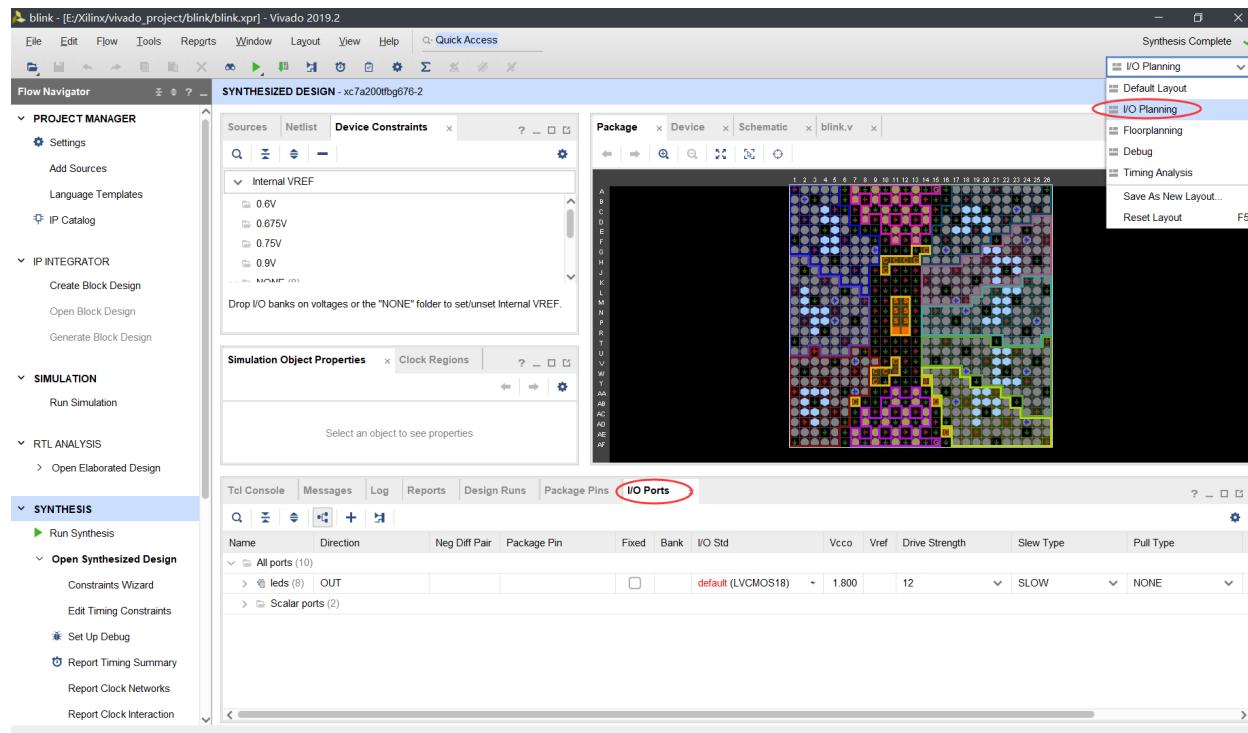
点击“Flow Navigator”中“Synthesis”下的“Run Synthesis”，在弹出的“Launch Runs”窗口点“OK”，先对工程进行综合。



综合完成之后，选择“Open Synthesized Design”，打开综合后的网表。



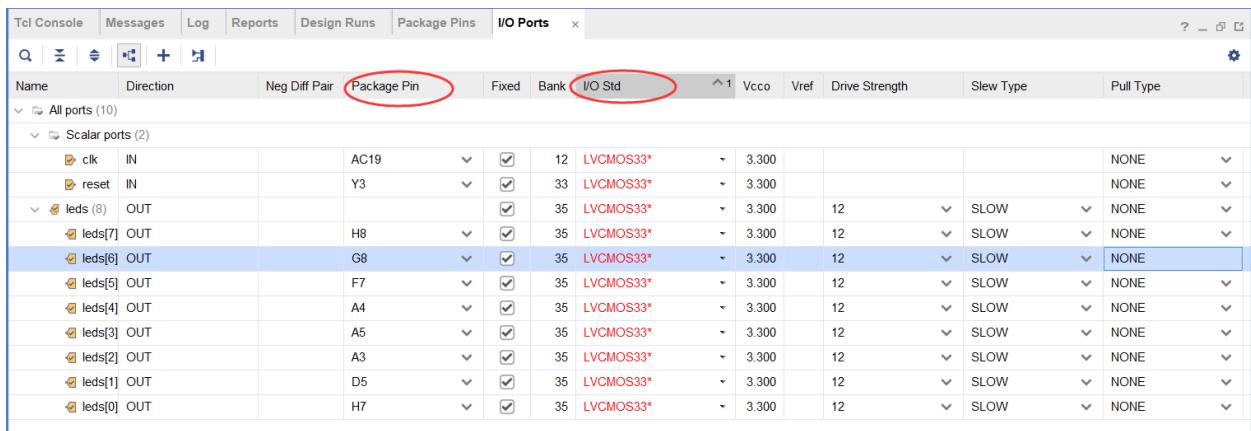
此时应看到如下界面，如果没出现如下界面，在图示位置的“layout”中选择“I/O Planning”一项。



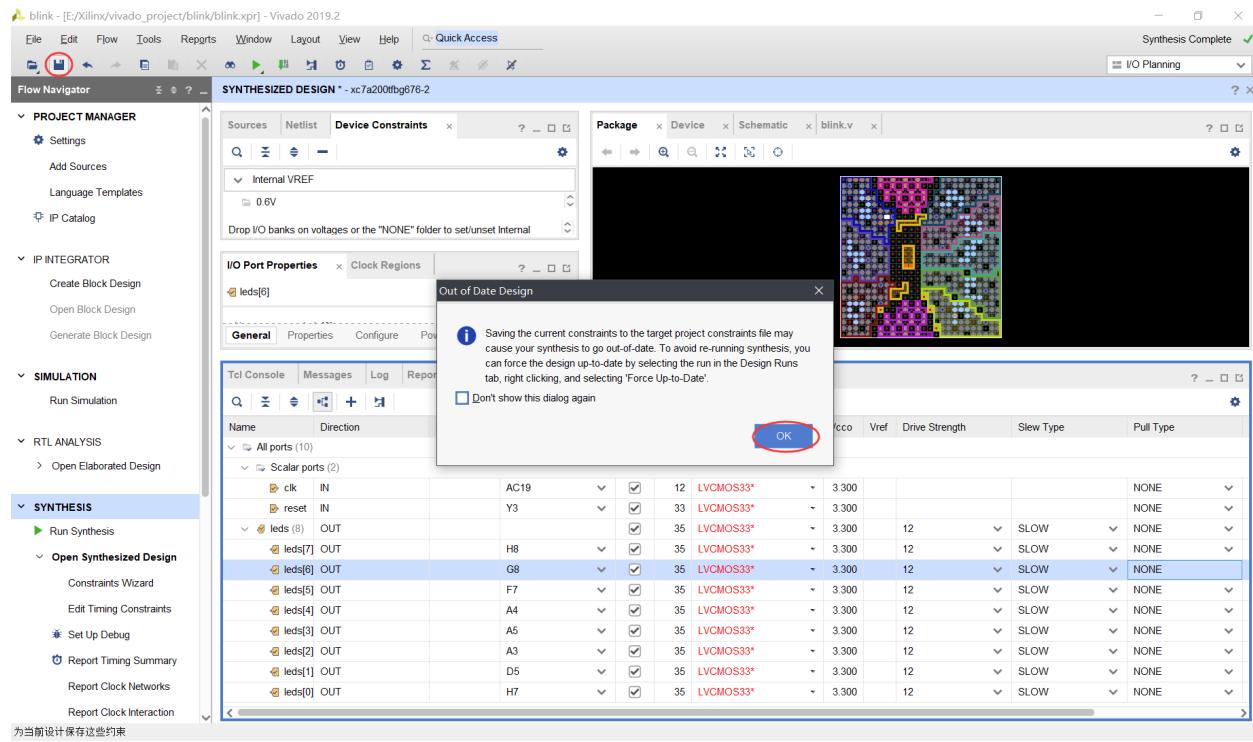
在界面右下方的选项卡中切换到“**I/O Ports**”一栏，并在对应的信号后，在“Site”一列输入对应的FPGA管脚标号（或将信号拖拽到右上方“**Package**”图中对应的管脚上），并指定“**I/O std**”。具体的FPGA约束管脚和I/O电平标准，可参考对应板卡的原理图或提供的常用的引脚对应关系表（参见龙芯Artix-7实验箱-原理图与引脚列表）。

	A	B	C
1	引脚对应关系		
2	引脚名称	FPGA接口编号	描述
3	LED灯		
4	FPGA_LED1	H7	实验板放正，一排LED灯左起第一个。
5	FPGA_LED2	D5	依次类推
6	FPGA_LED3	A3	
7	FPGA_LED4	A5	
8	FPGA_LED5	A4	
9	FPGA_LED6	F7	
10	FPGA_LED7	G8	
11	FPGA_LED8	H8	
12	FPGA_LED9	J8	
13	FPGA_LED10	J23	
14	FPGA_LED11	J26	
15	FPGA_LED12	G9	
16	FPGA_LED13	J19	
17	FPGA_LED14	H23	
18	FPGA_LED15	J21	
19	FPGA_LED16	K23	实验板放正，一排LED灯左起第八个。

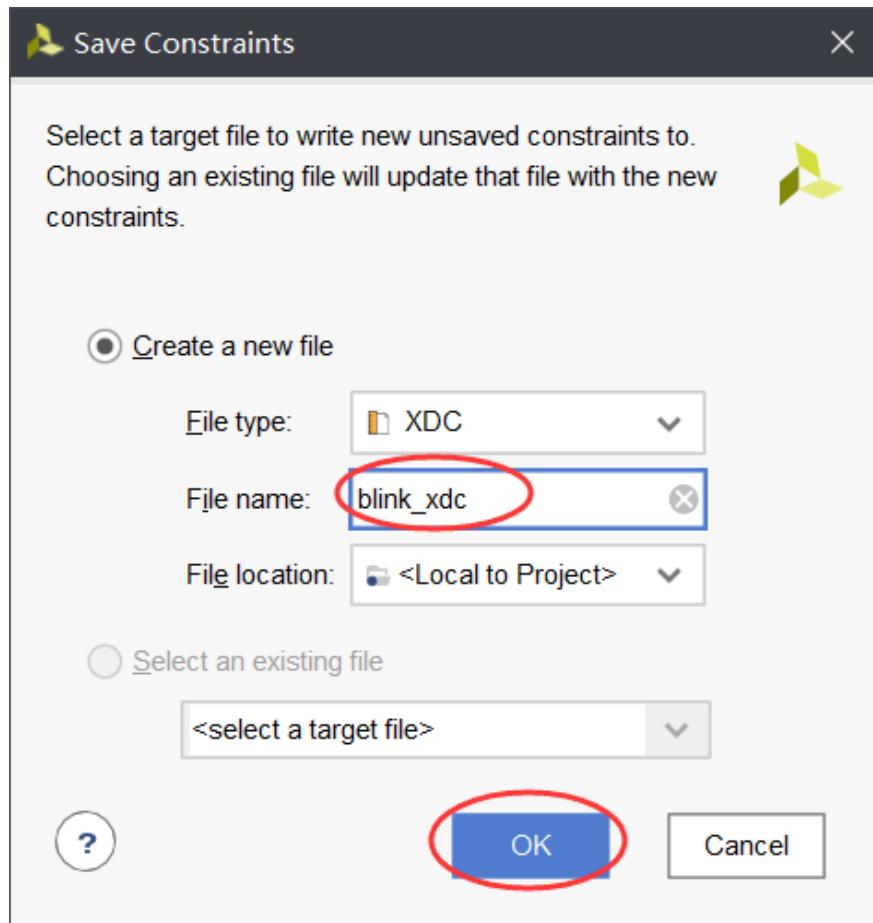
完成 I/O Ports 设置如下图，“I/O Std” 设置为 “LVC MOS33*” 表明 I/O 标准是 3.3V 的 LVC MOS。



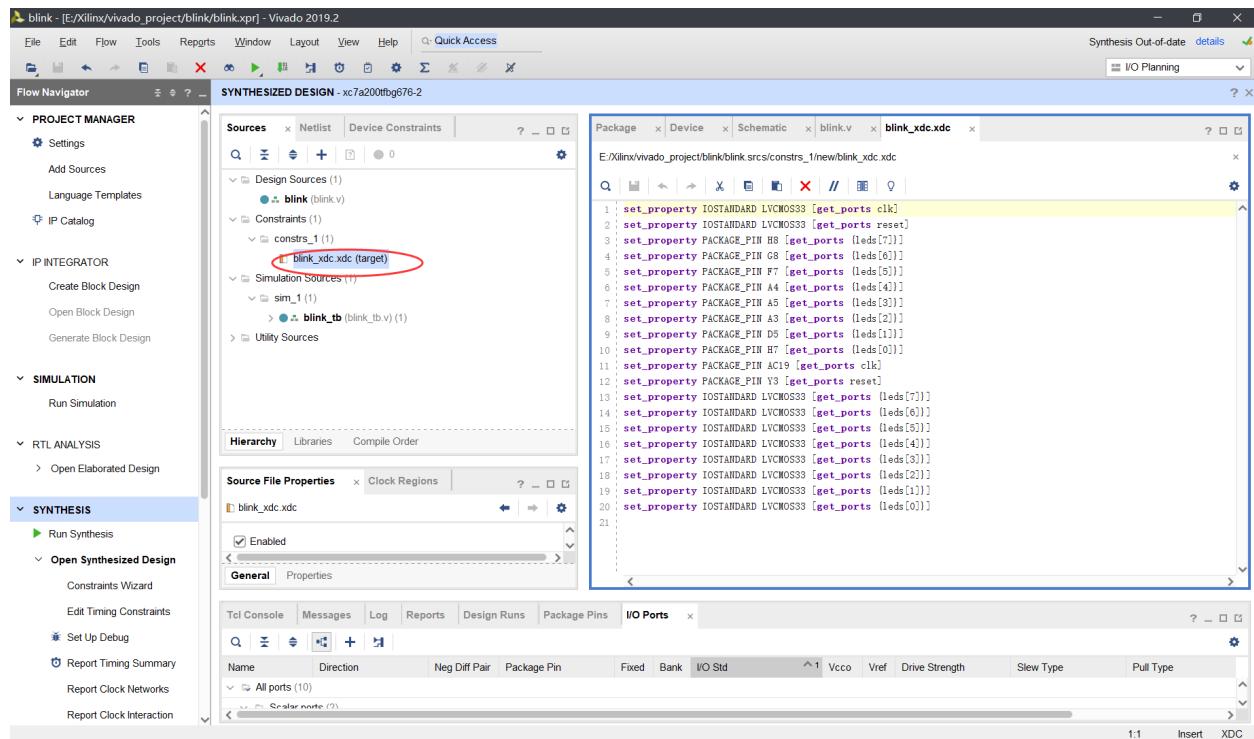
点击左上方工具栏中的保存按钮，提示添加新的约束文件使综合过时失效。点击“OK”。



新建 XDC 文件或选择工程中已有的 XDC 文件。选择“Create a new file”，输入“File name”，点击“OK”完成约束过程。

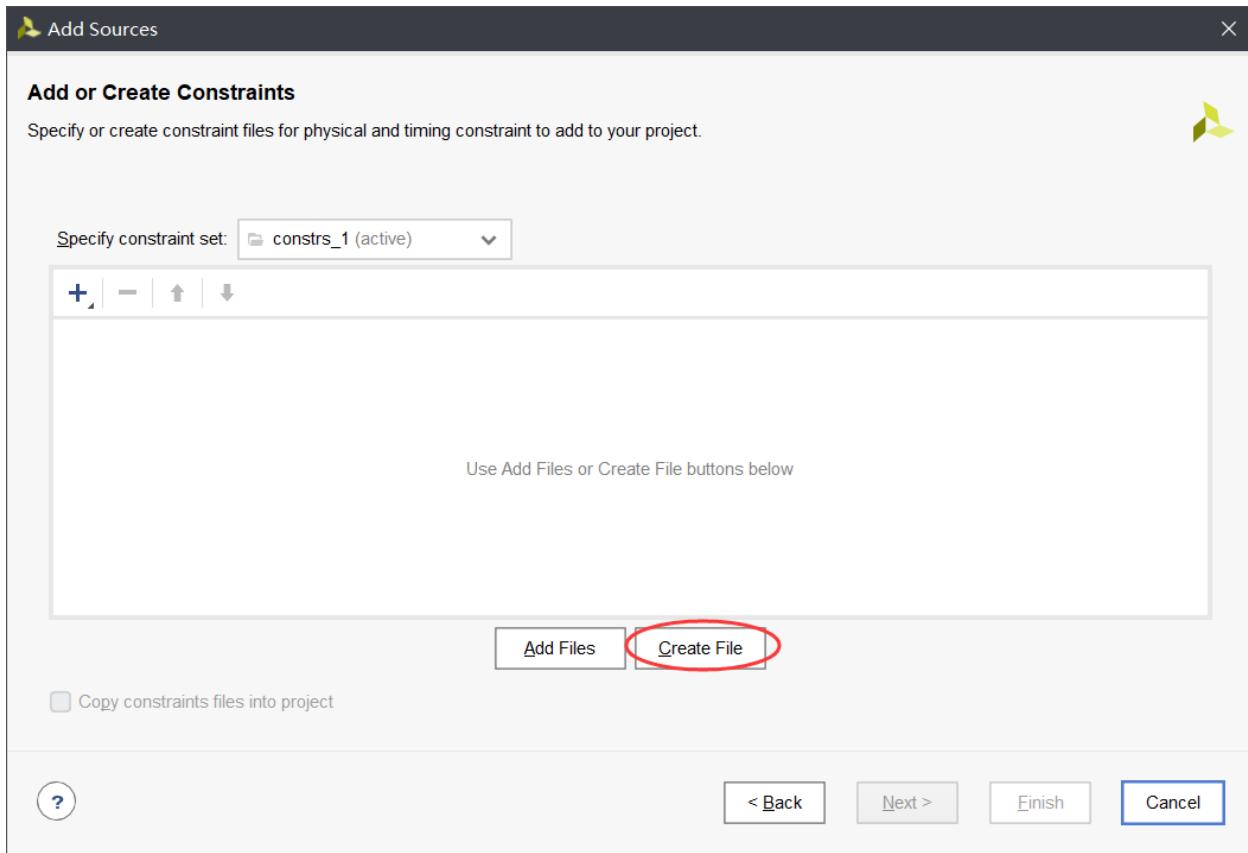


在“Sources”窗口“Constraints”层次下可以查看新建的 XDC 文件。

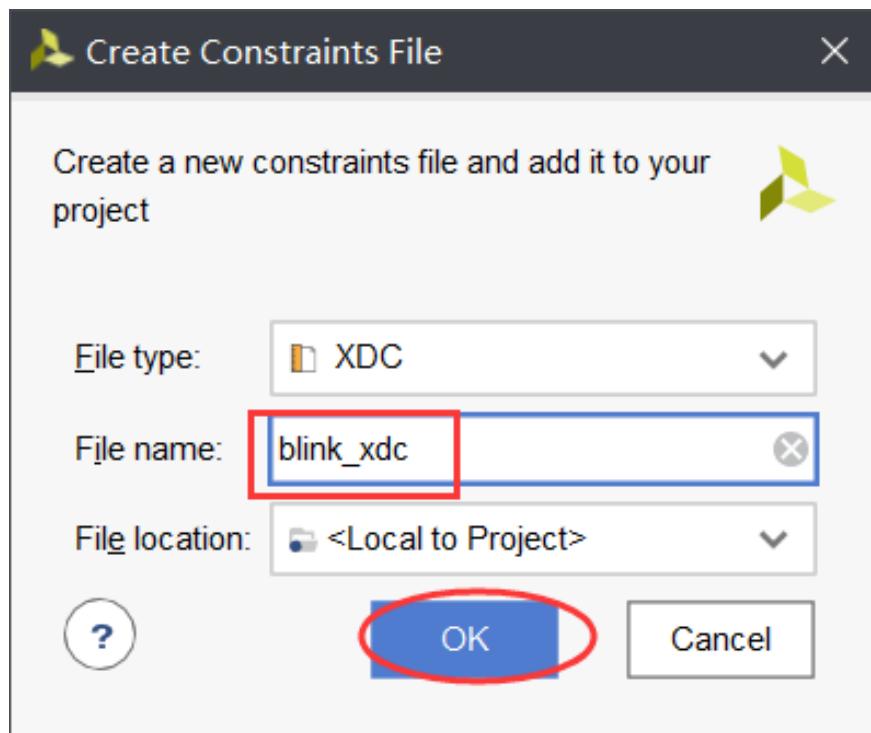


新建 XDC 文件并手动输入约束

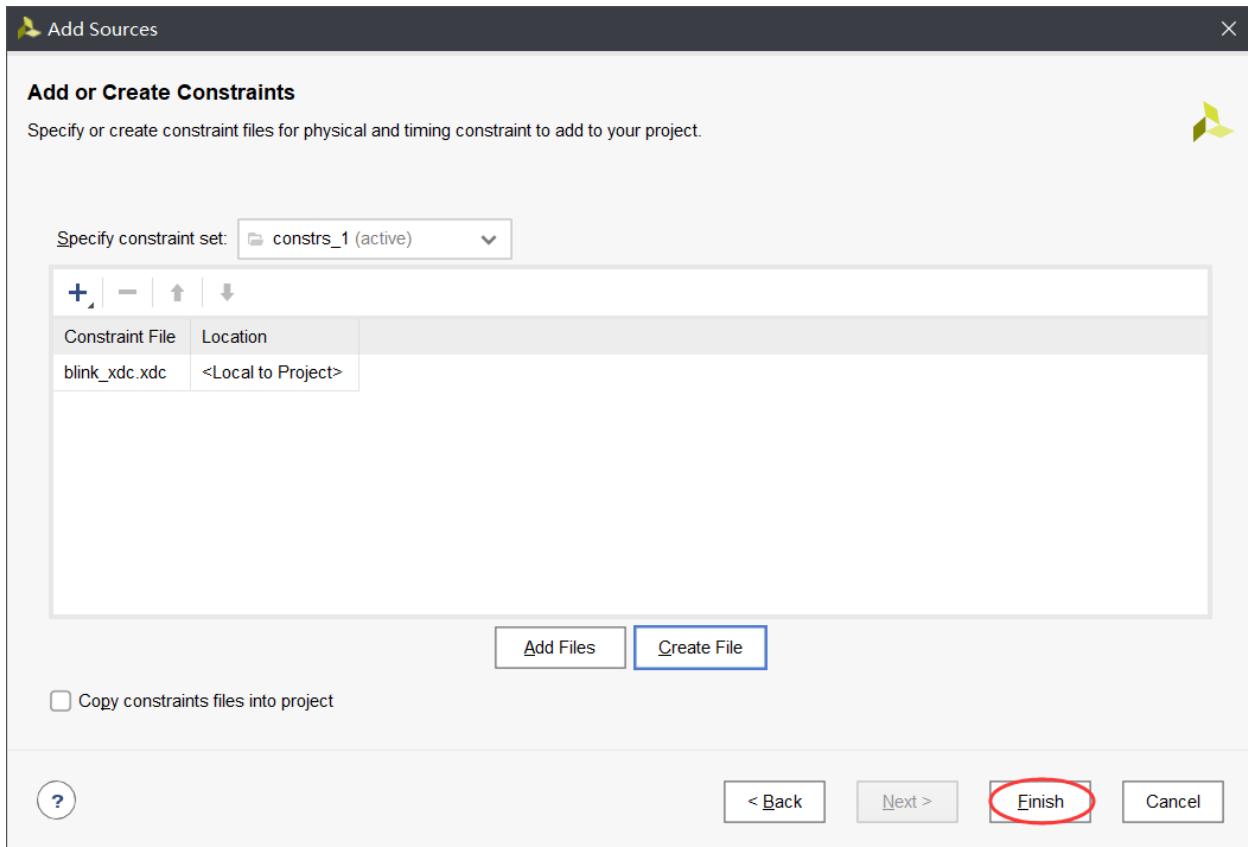
新建约束文件。点击“Add Sources”，选择“Add or Create Constraints”，点击“Next”。



设置新建的 XDC 文件，输入 XDC 文件名，点击“OK”。默认文件在工程目录下的“\blink.srcc\constrs_1\new”中。



完成新建 XDC 文件，点击“Finish”。



在“Sources”窗口“Constraints”下双击打开新建好的 XDC 文件“blink_xdc.xdc”，并参照下面的约束文件格式，输入相应的 FPGA 管脚约束信息和电平标准。



约束文件参考如下。

```
set_property PACKAGE_PIN AC19 [get_ports clk]
set_property PACKAGE_PIN Y3 [get_ports reset]

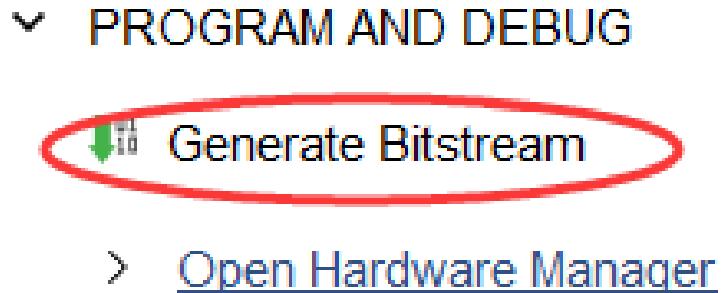
set_property IOSTANDARD LVCMOS33 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports reset]

set_property PACKAGE_PIN H8 [get_ports {leds[7]}]
set_property PACKAGE_PIN G8 [get_ports {leds[6]}]
set_property PACKAGE_PIN F7 [get_ports {leds[5]}]
set_property PACKAGE_PIN A4 [get_ports {leds[4]}]
set_property PACKAGE_PIN A5 [get_ports {leds[3]}]
set_property PACKAGE_PIN A3 [get_ports {leds[2]}]
set_property PACKAGE_PIN D5 [get_ports {leds[1]}]
set_property PACKAGE_PIN H7 [get_ports {leds[0]}]

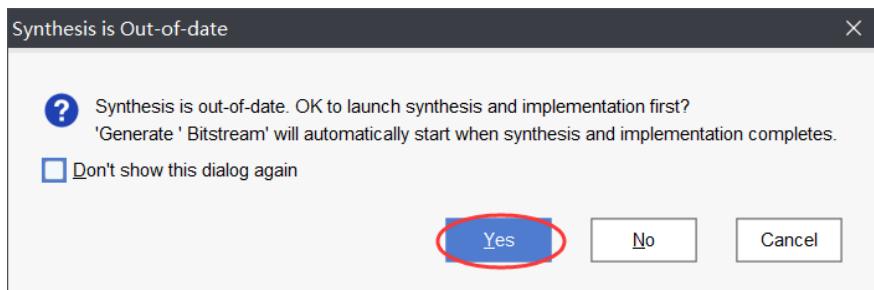
set_property IOSTANDARD LVCMOS33 [get_ports {leds[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds[0]}]
```

工程实现

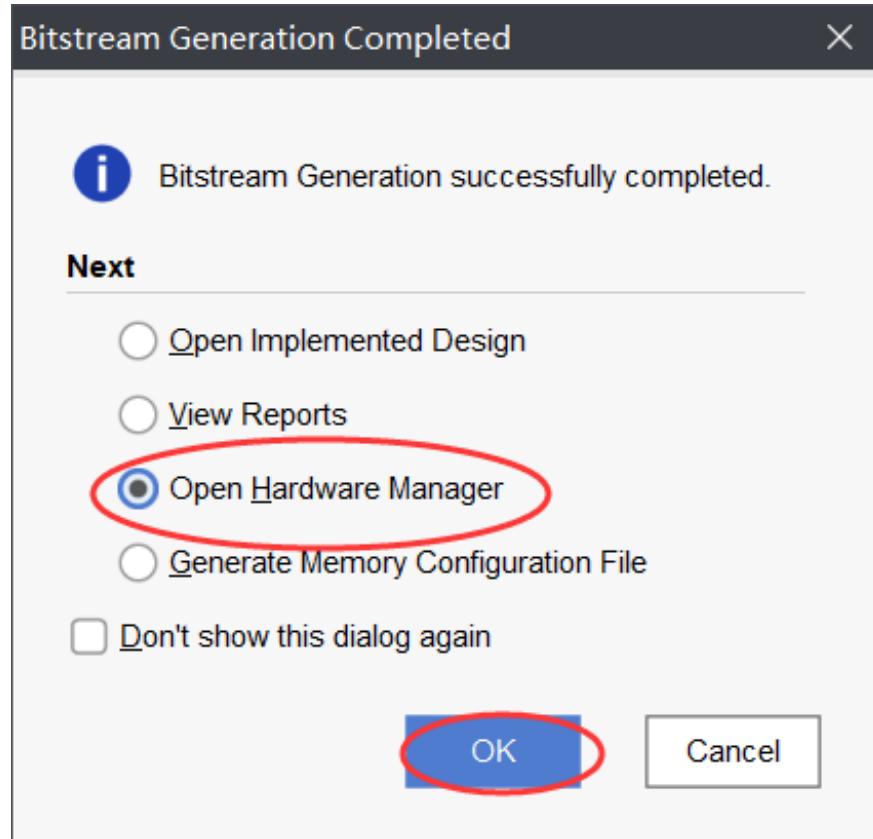
在“Flow Navigator”中点击“Program and Debug”下的“Generate Bitstream”选项，工程会自动完成综合、布局布线、Bit 文件生成过程，完成之后，可点击“Open Implemented Design”来查看工程实现结果。



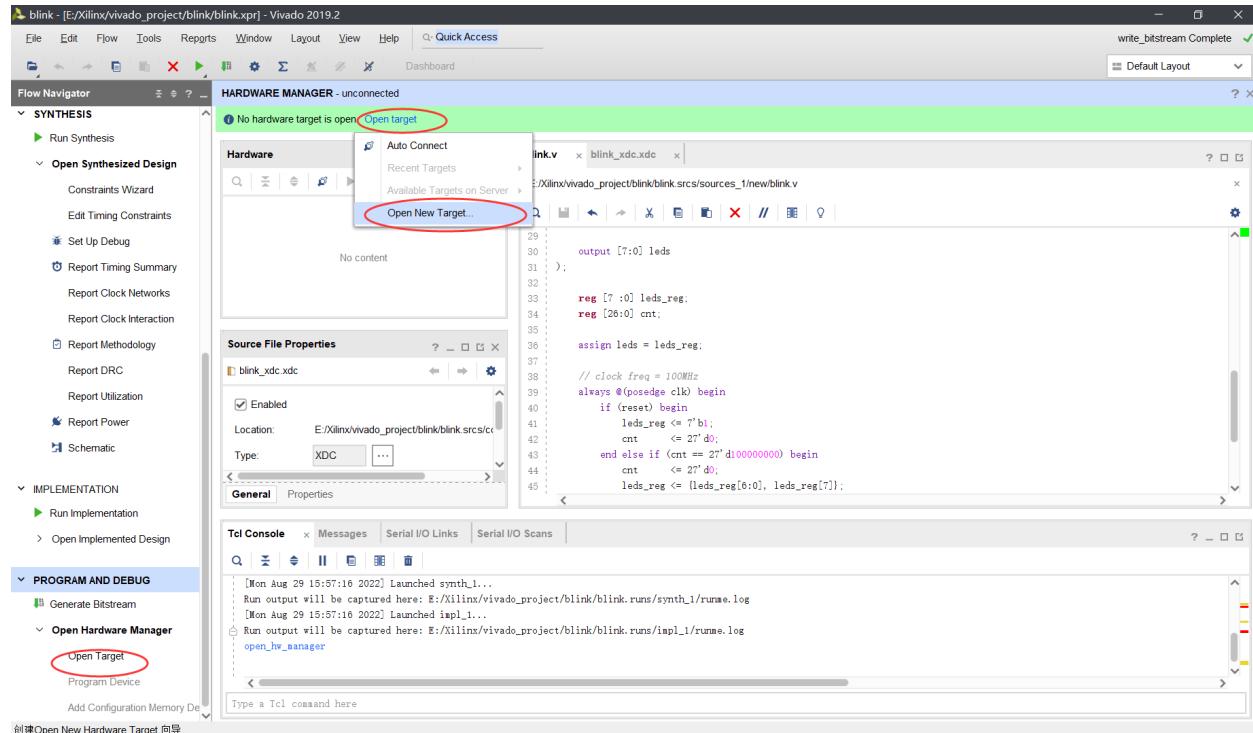
提示综合过时重新运行综合和实现，点击“Yes”。在弹出的“Launch Runs”窗口点“OK”。



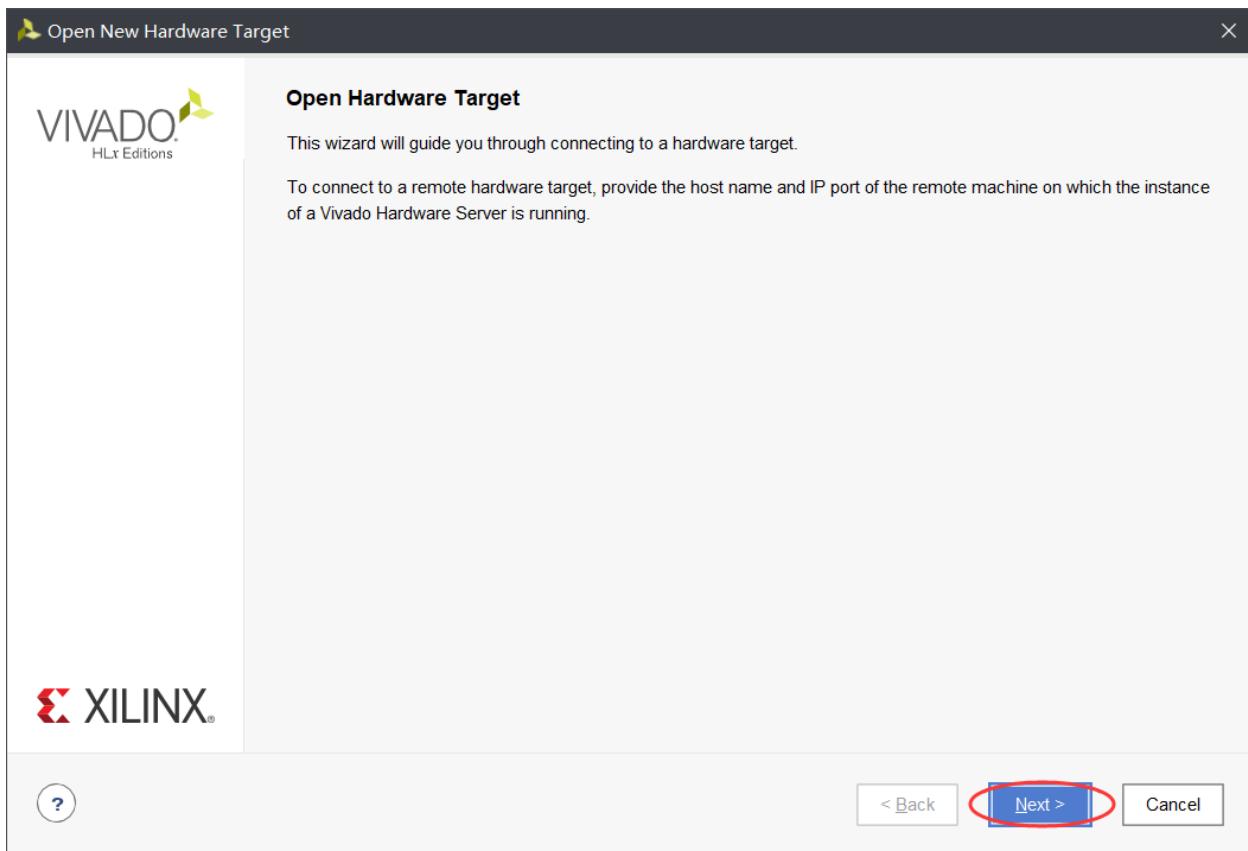
在比特流文件生成完成的窗口选择“Open Hardware Manager”，进入硬件管理界面。连接 FPGA 开发板的电源线和与电脑的下载线，打开 FPGA 电源。



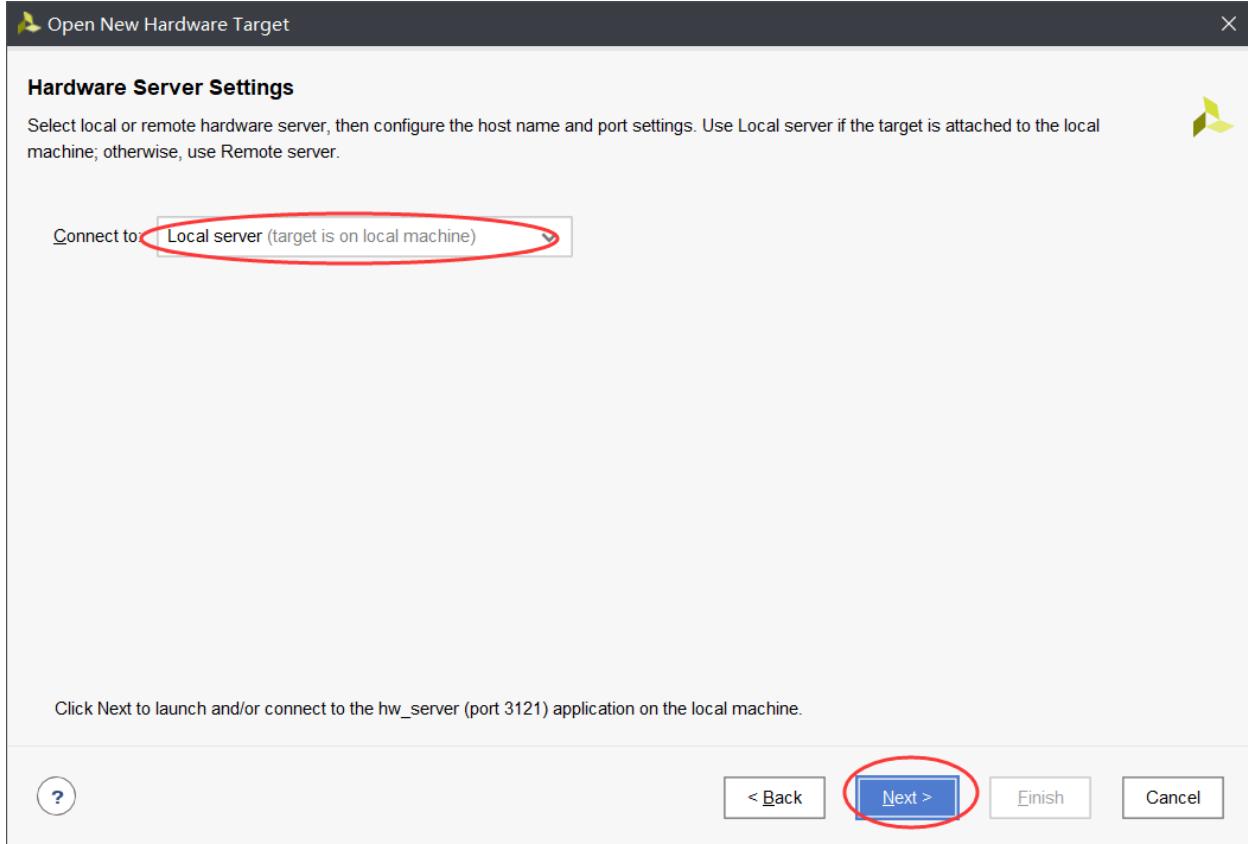
在“Hardware Manager”窗口的提示信息中，点击“Open target”的下拉菜单的“Open New Target”（或在“Flow Navigator”下“Program and Debug”中展开“Open Hardware Manager”，点击“Open Target”->“Open New Target”）。也可以选择“Auto Connect”自动连接器件。



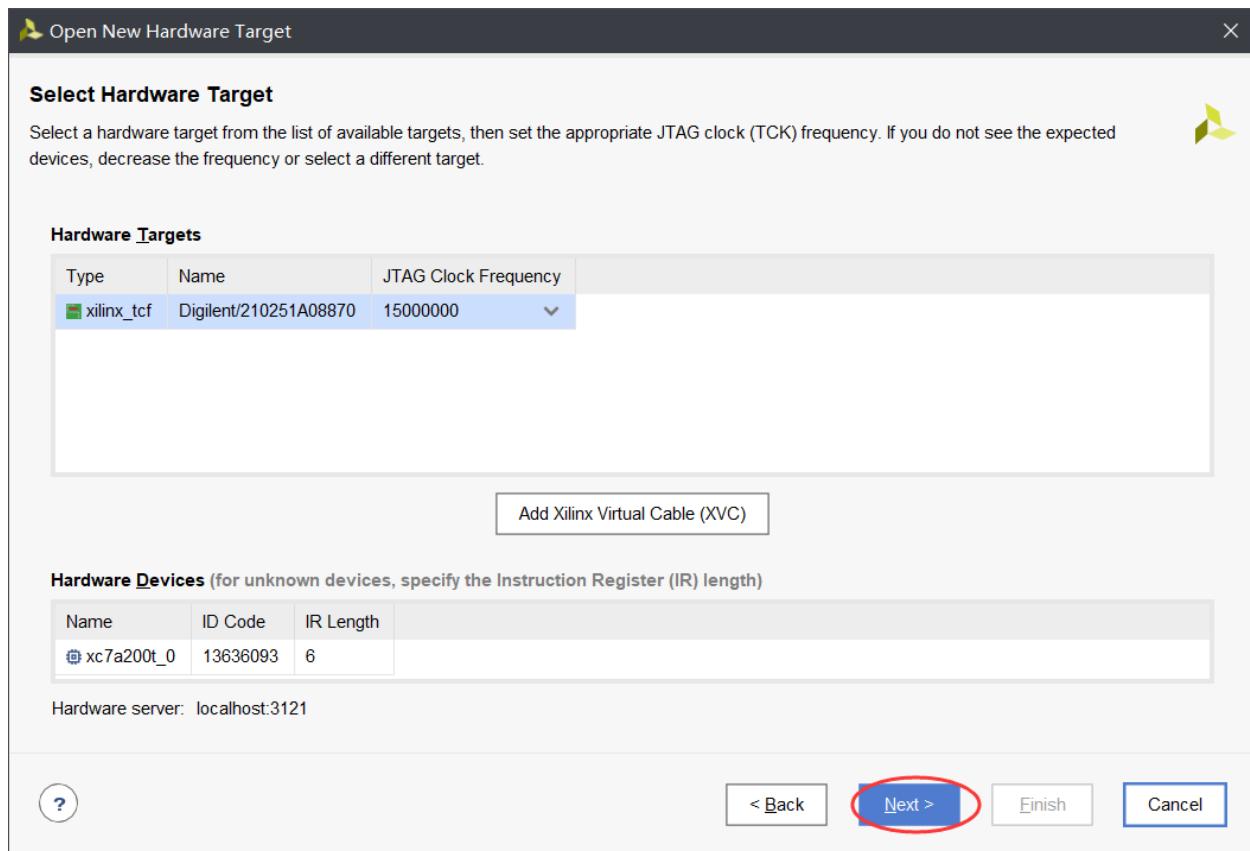
在“Open Hardware Target”向导中，先点击“Next”，进入 Server 选择向导。



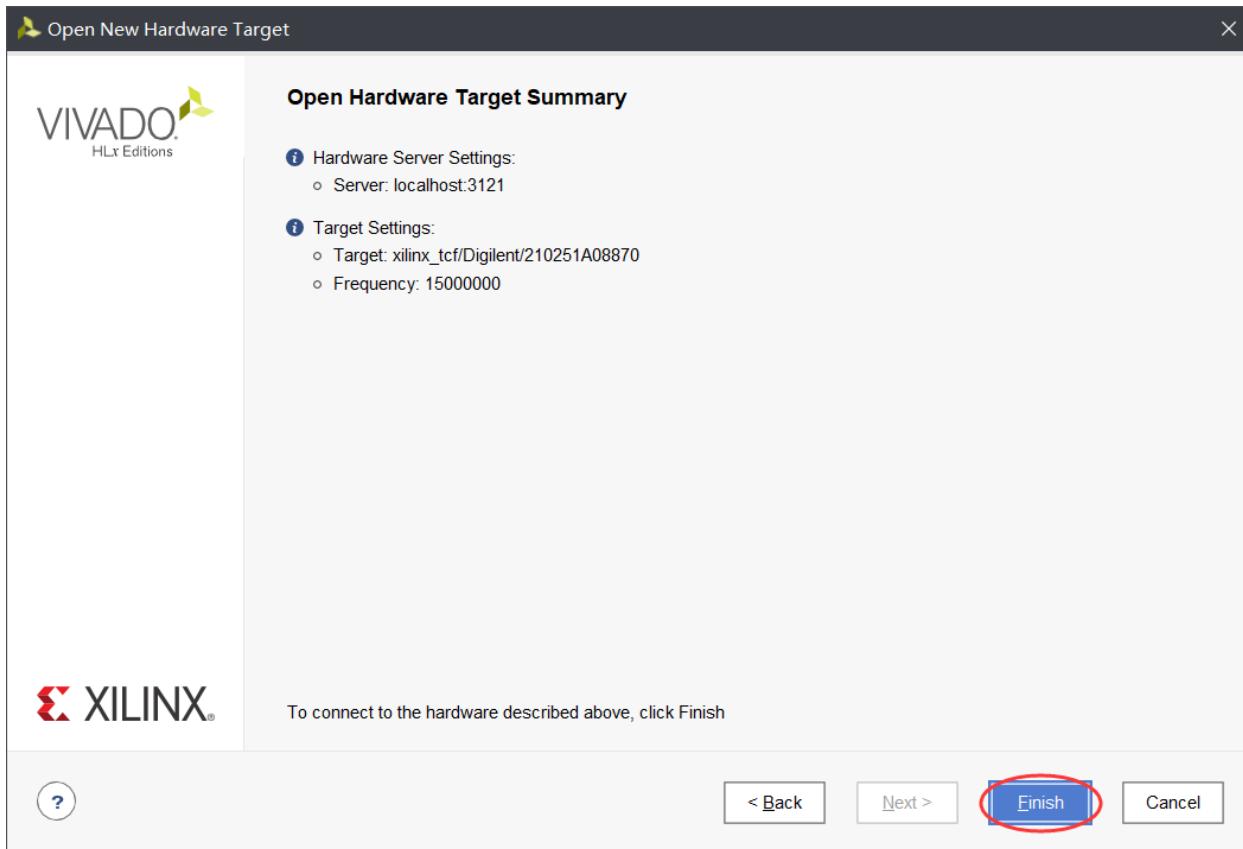
选择连接到“Local server”，点击“Next”。



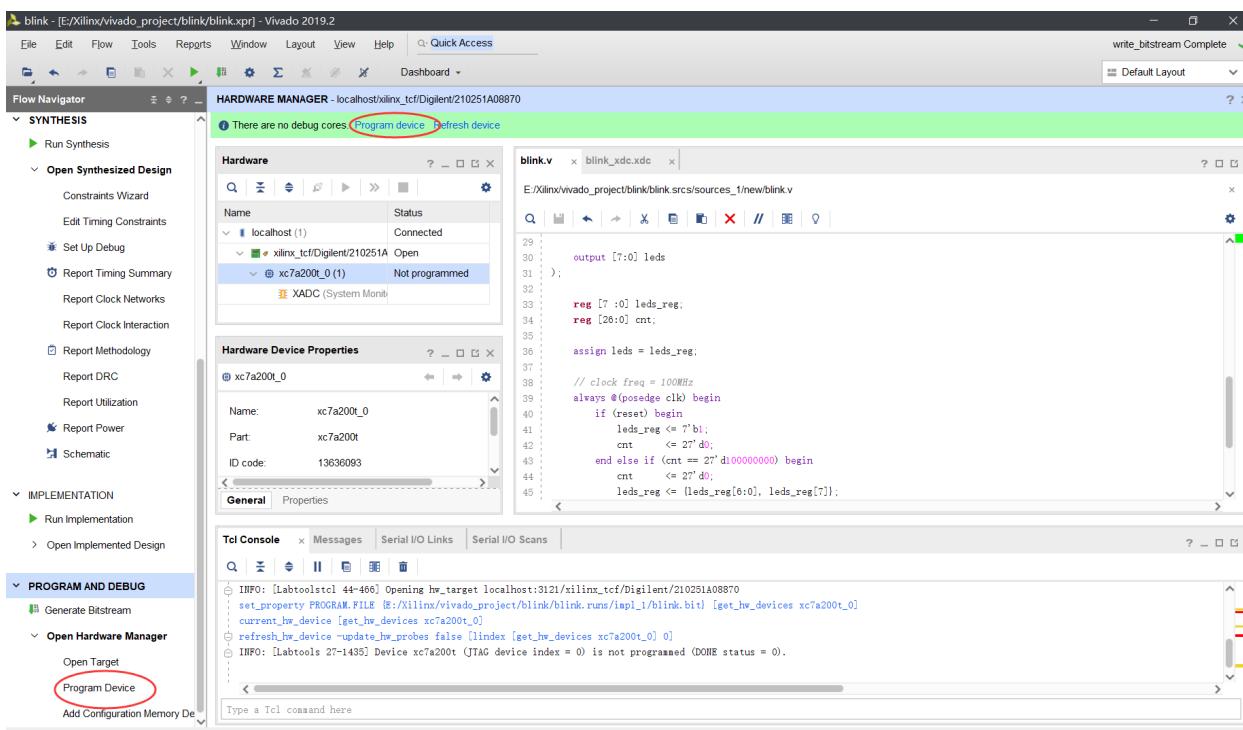
选择目标硬件，点击“Next”。



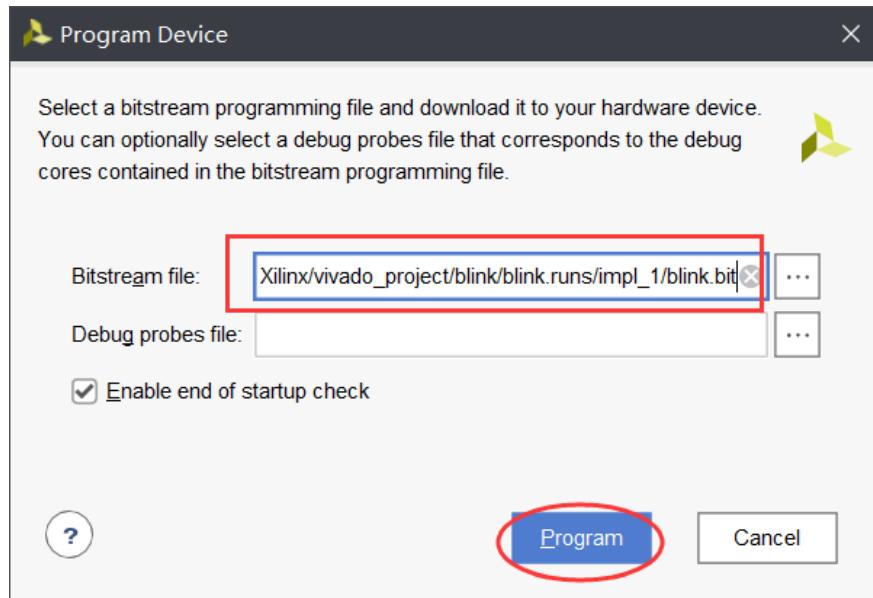
完成目标硬件打开。点击“Finish”。



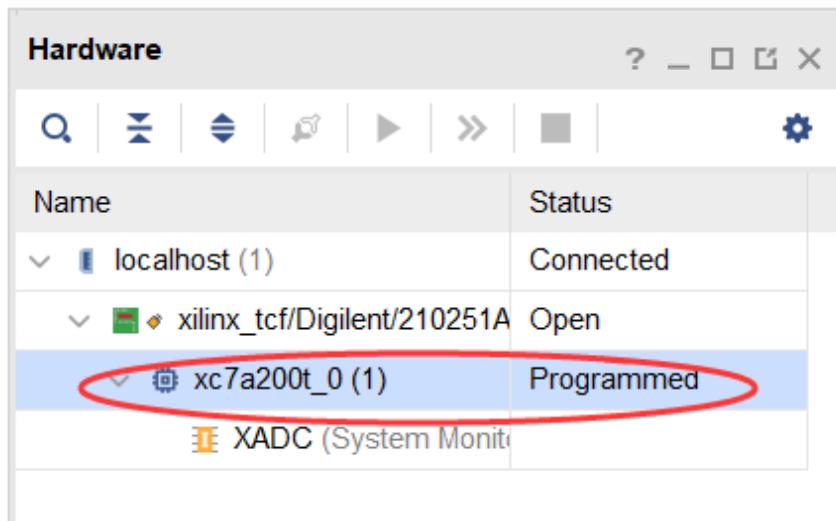
对目标硬件编程。在“Hardware”窗口右键单击目标器件“xc7a200t_0”，选择“Program Device…”。或者“Flow Navigator”窗口中“Program and Debug”->“Hardware Manager”->“Program Device”。



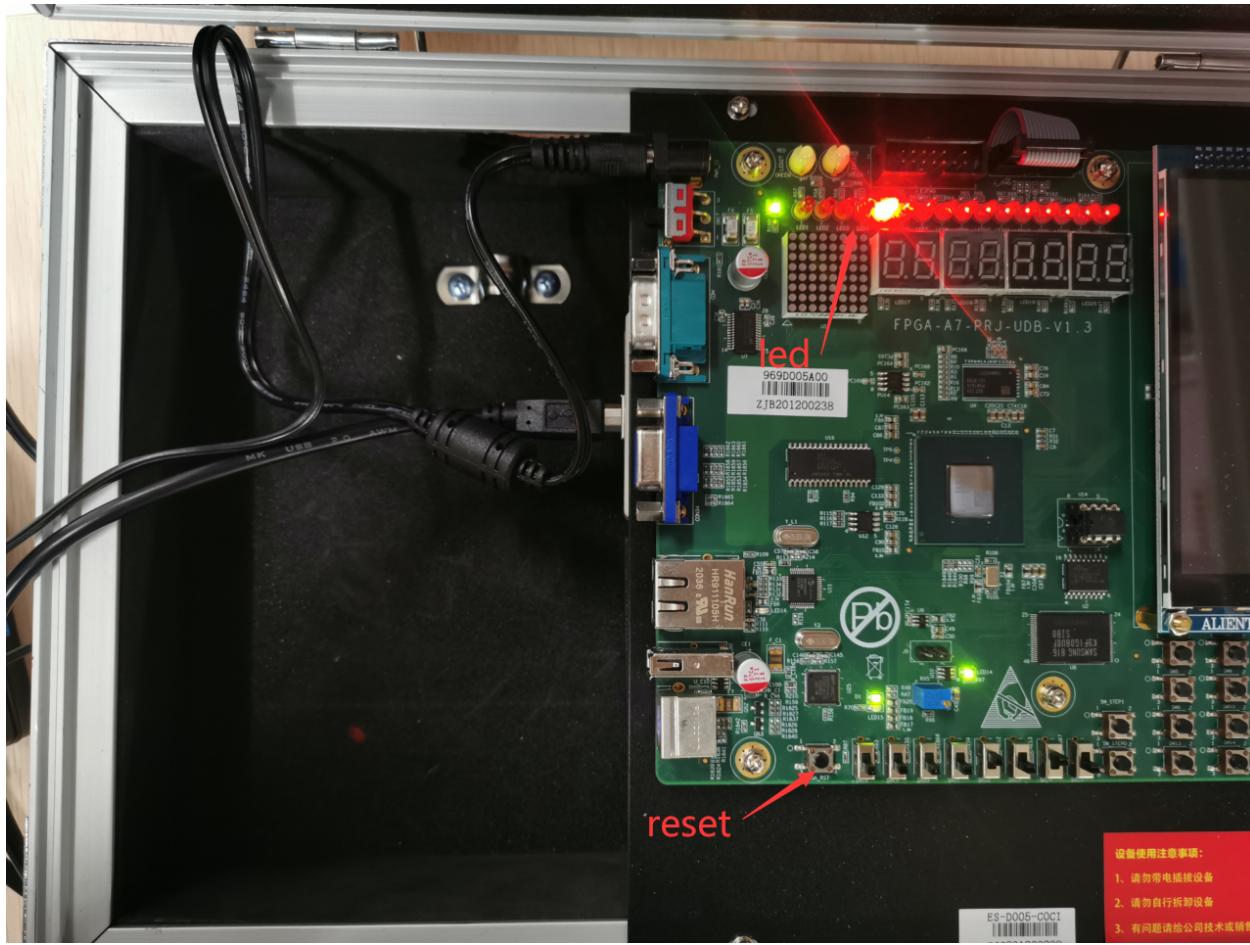
选择下载的 bit 流文件，点击“Program”。



完成下载后，“Hardware”窗口下的“xc7a200t_0”的状态变成“Programmed”。



FPGA 开发板上，可以看到 8 个 led 灯 LED1~LED8 实现了流水灯的效果。设计完成。



FPGA 在线调试说明

在 FPGA 开发过程中，经常碰到仿真上板行为不一致，更多时候是仿真正常，上板异常。由于上板调试手段薄弱，导致很难定位错误。这时候可以借助 Xilinx 的下载线进行在线调试，在线调试是在 FPGA 上运行过程中探测预定好的信号，然后通过 USB 编程线缆显示到上位机上。

本节给出简单使用在线调试的方法：RTL 里设定需探测的信号，综合并建立 Debug core，实现并生产 bit 流文件，下载 bit 流和 debug 文件，上板观察。

抓取需探测的信号

在 RTL 源码中，给想要抓取的探测信号声明前增加 (`mark_debug = "true"`)。

仍然以流水灯实验为例，我们想要在 FPGA 板上观察 clk 信号、reset 信号、led 寄存器以及 cnt 计数器，需要在代码里进行如下修改。

```
// blink_tb.v
`timescale 1ns / 1ps

module blink (
    (*mark_debug = "true") input      clk ,
    (*mark_debug = "true") input      reset,
```

(续下页)

(接上页)

```

    output [7:0] leds
);

(*mark_debug = "true") reg [7 :0] leds_reg;
(*mark_debug = "true") reg [26:0] cnt;

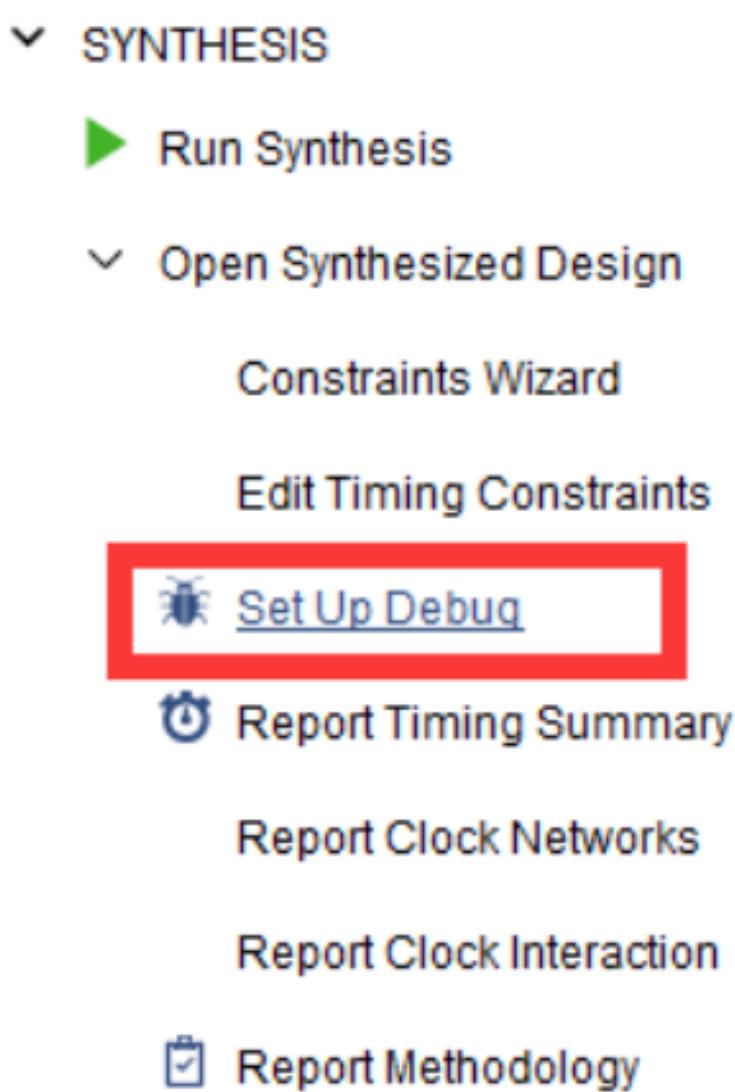
...
endmodule

```

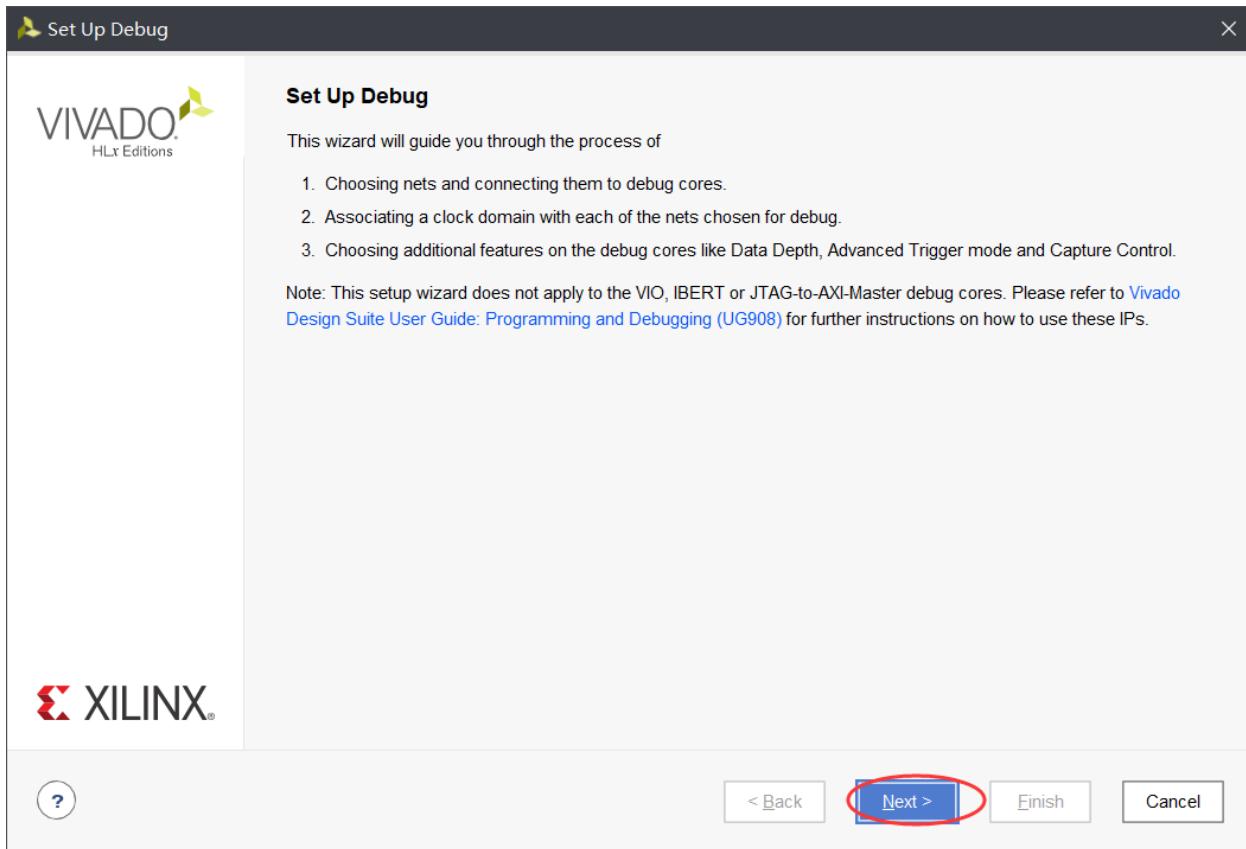
综合并建立 debug

在综合完成后，需建立 debug。

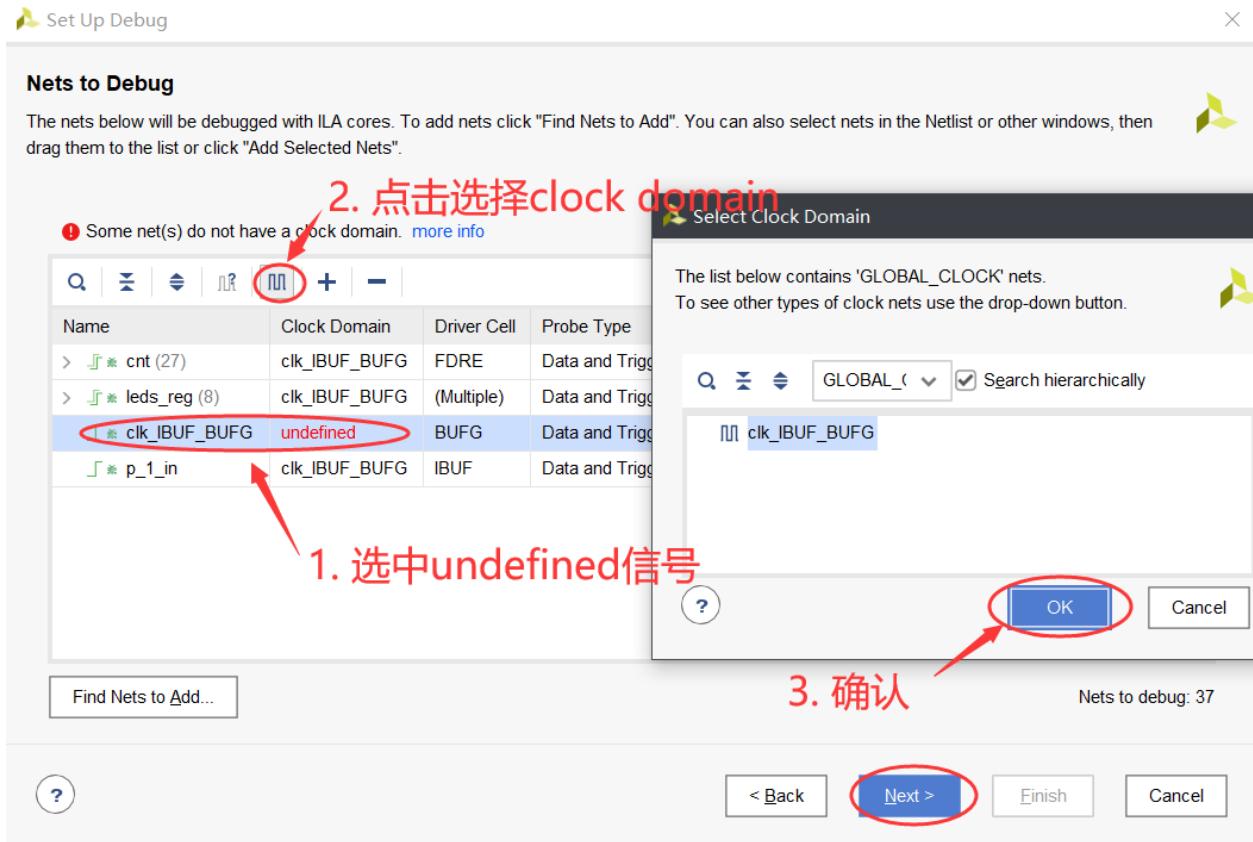
点击工程左侧 synthesis->Open Synthesized Desgin->Set Up Debug。



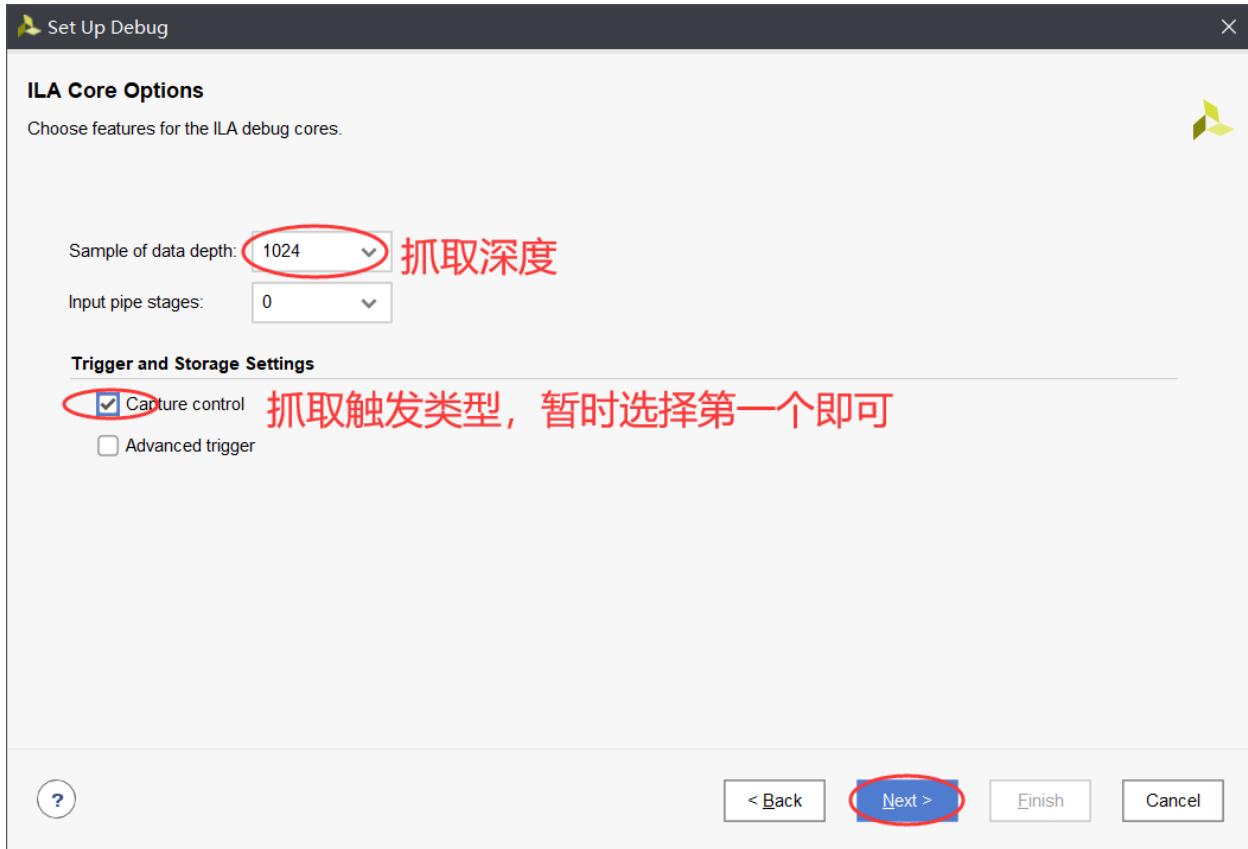
随后会出现如下界面，点击 Next。



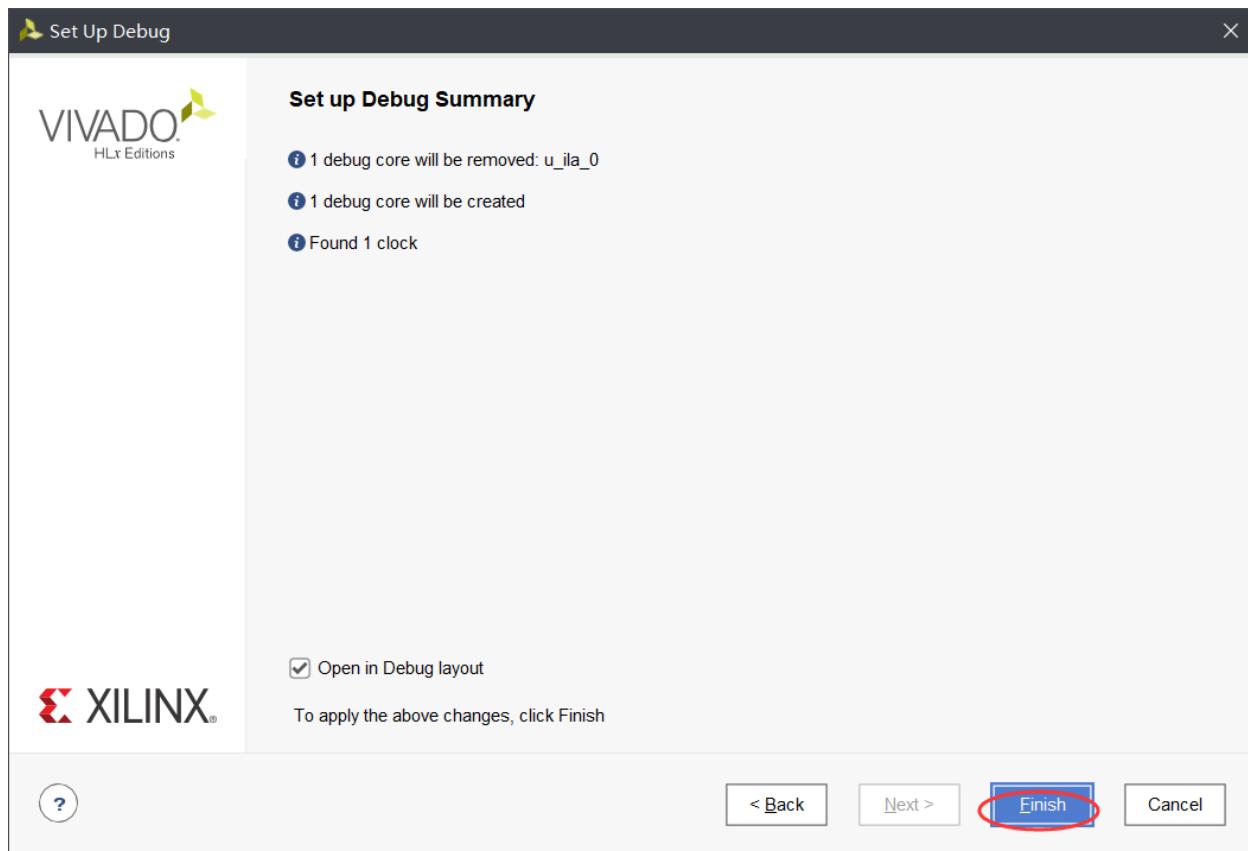
随后会列出抓取的 Debug 信息，点击 Next。若出现如下图所示的 Clock Domain 为 undefined 的情况，则需要按图中所示手动进行配置。



选择抓取的深度和触发控制类型，点击 Next（更高级的调试可以勾选“Advanced trigger”）。

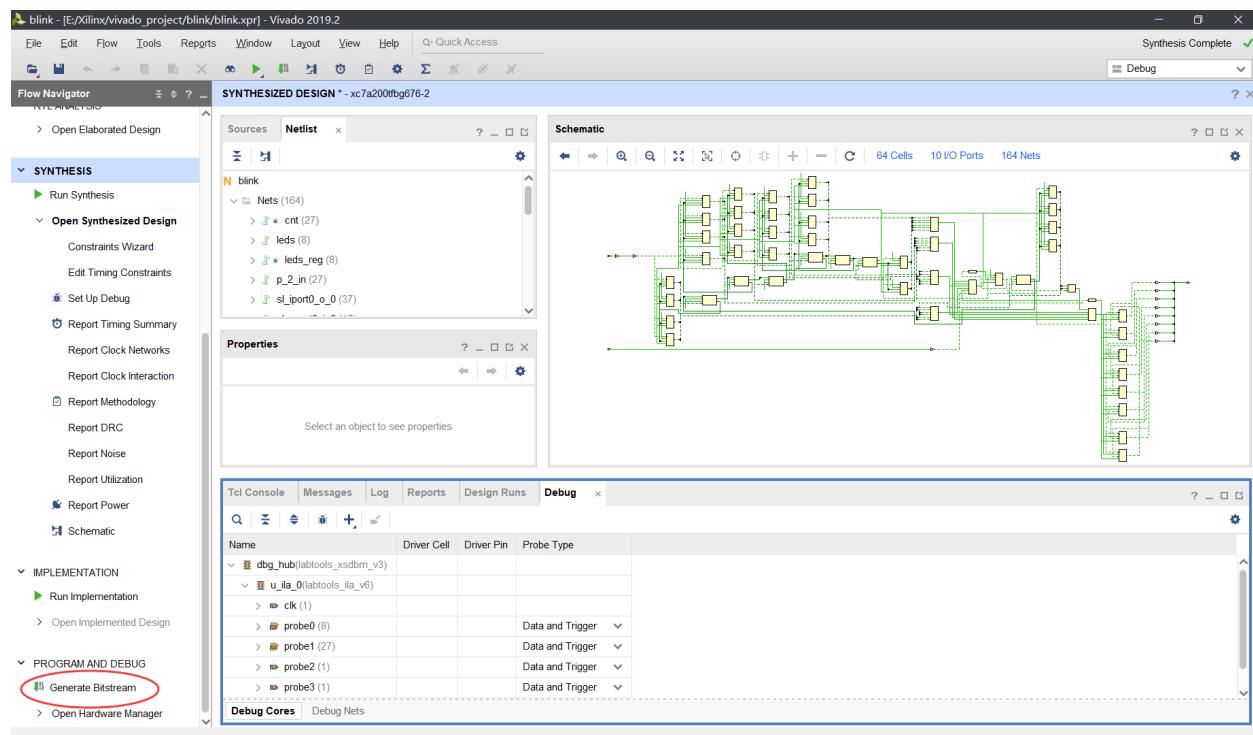


最后, 点击 Finish。

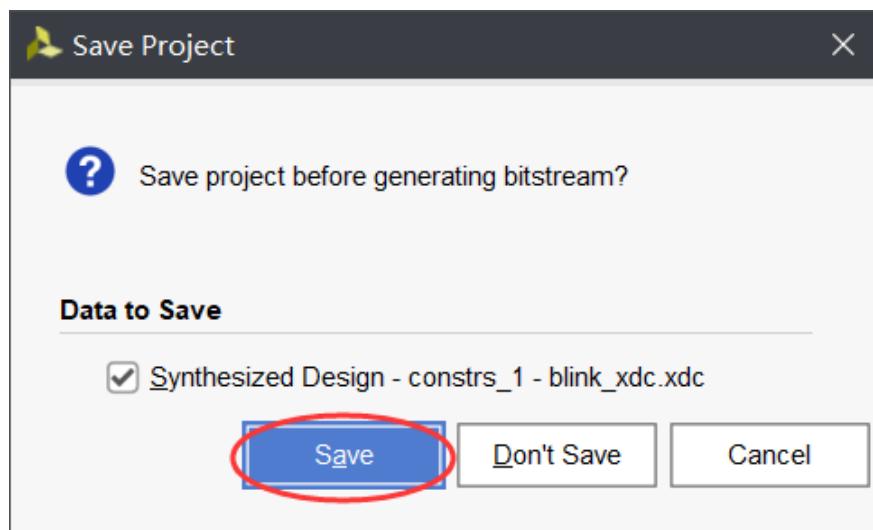


实现并生产 bit 流文件

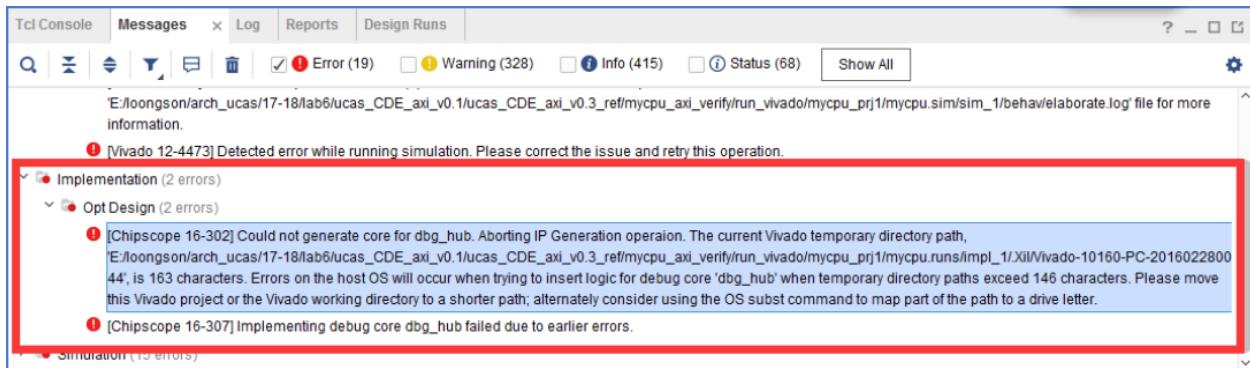
完成上一节后会出现类似下图界面，直接点击 Generate Bitstream。



弹出如下界面，点击 Save。

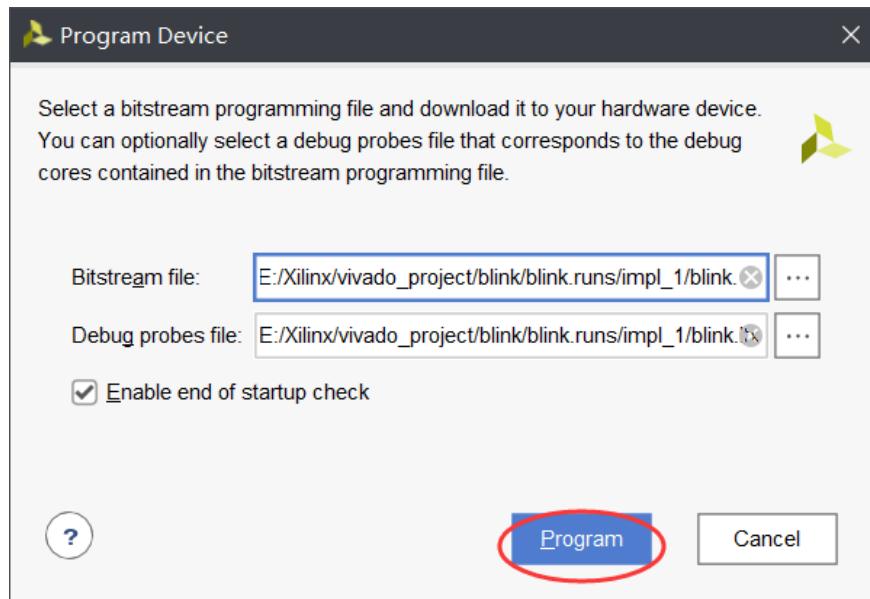


如果有后续弹出界面，继续点击 OK 或 Yes 即可。这时就进入后续生产 bit 文件的流程了，此时 Vivado 界面里的 synthesis design 界面就可以关闭了。如果发现以下错误，则是因为路径太深，引用起来名字太长，降低工程目录深度即可。



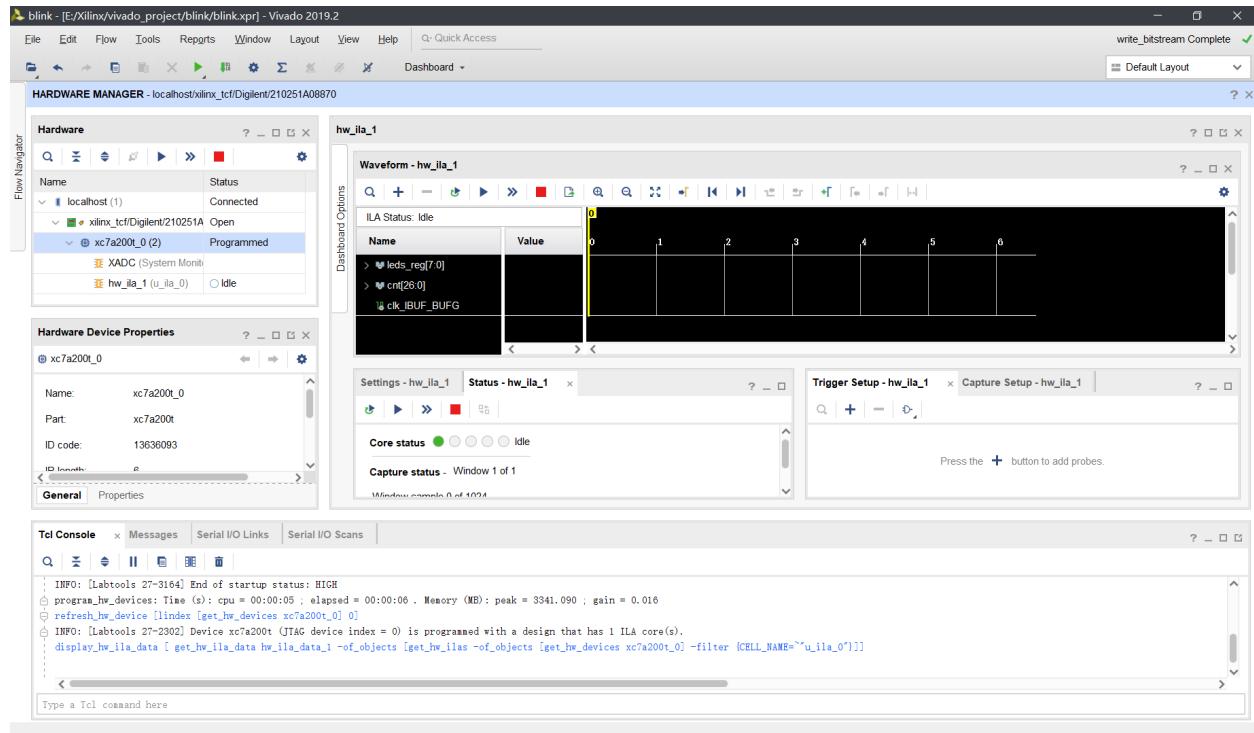
下载 bit 流和 debug 文件

在完成上一节后，会生成 bit 流文件和调试使用 ltx 文件。这里，打开 Open Hardware Manager，连接好 FPGA 开发板后，选择 Program device，如下图。自动加载了 bit 流文件和调试的 ltx 文件。选择 Program，等待下载完成。

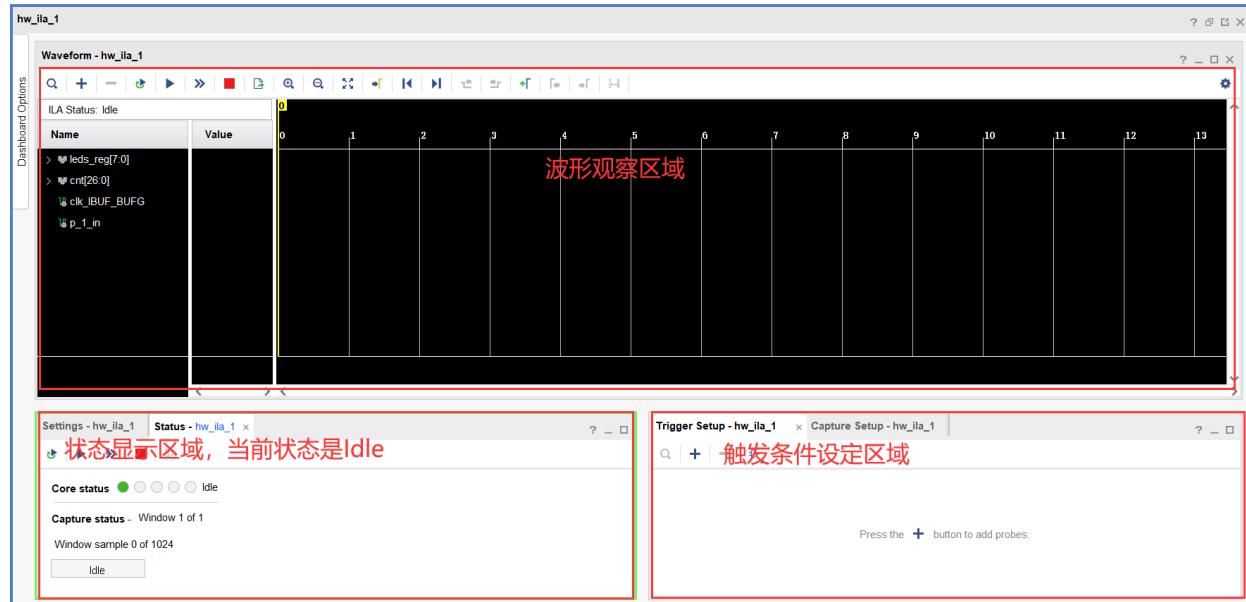


上板观察

在下载完成后, vivado 界面如下, 在线调试就是在 hw_ilia_1 界面里进行的。



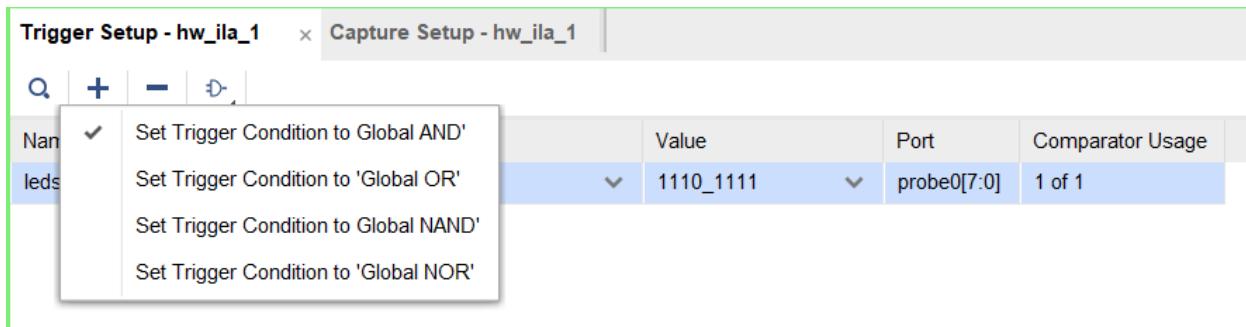
hw_ilia_1 界面主要分为 3 个界面, 分别如下。



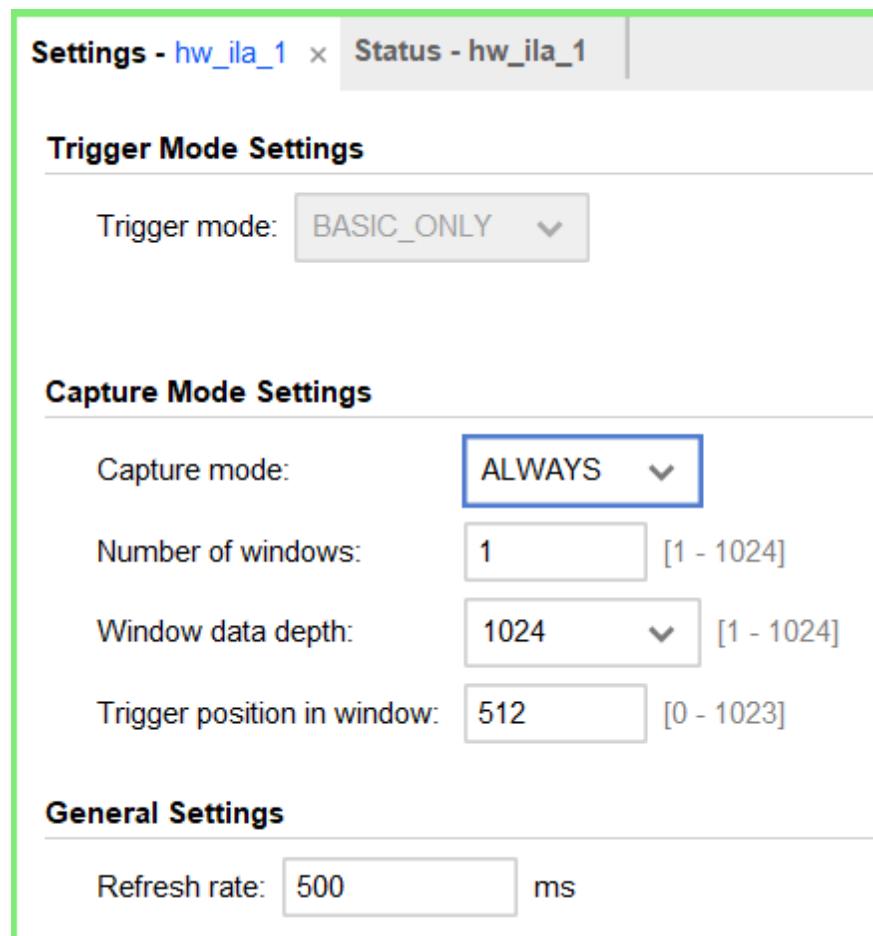
首先, 我们需要在右下角区域设定触发条件。所谓触发条件, 就是设定该条件满足时获取波形, 比如我先设定触发条件是 led 寄存器到达 8'b11101111。在下图中, 先点击 “+”, 在双击 leds_reg。然后如图设定触发条件。



此外可以设定多个触发条件，比如，再加一个除法条件是写回使能是 0xf，可以设定多个触发条件直接的关系，比如是任意一个满足、两个都是满足等等，如下图。

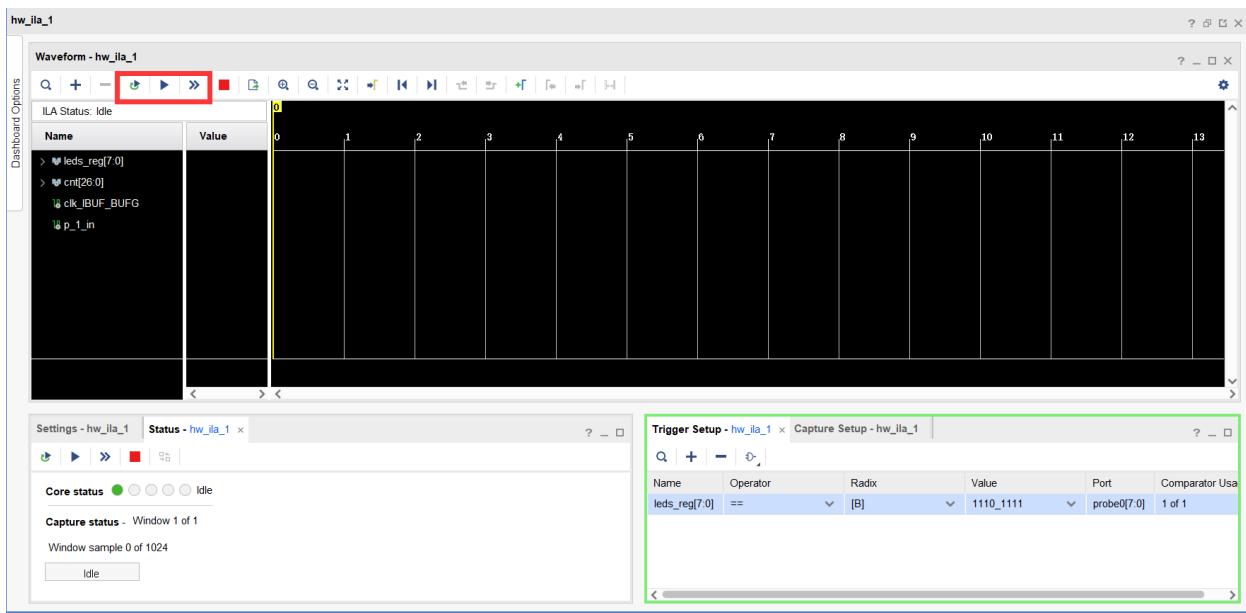


在左下角窗口，选择 settings，可以设定 Capture 选项，可能经常用到的是 Trigger position in window，用来设定触发条件满足的时刻在波形窗口的位置。比如，下图设定为 500，当触发条件满足时，波形窗口的第 500 个 clk 的位置是该条件，言下之意，将触发条件满足前的 500 个 clk 的信号值也抓出来了，这样可以看到触发条件之前的电路行为。Refresh rate 设定了波形窗口的刷新频率。



触发条件建立后，就可以启动波形抓取了，最关键的有三个触发按键，即下图圈出的 3 个按键：

- 左起第一个，设定触发模式，有两个选项：单触发；循环触发。当该按键按下时，表示循环检测触发，那么只要触发条件满足，波形窗口就会更新。当设置为单触发时，就是触发一次完成后，就不会再检测触发条件了。
- 左起第二个，等待触发条件被满足。点击该按键，就是等待除法条件被满足，展示出波形。
- 左起第三个，立即触发。点击该按键，表示不管触发条件，立即抓取一段波形展示到窗口中。



剩下的 debug 过程，就和仿真 debug 类似了，去观察波形。但在线调试时，你无法添加未被添加 debug mark 的信号。在线调试过程中，可能需要不停的更换触发条件，不停的按复位键。

注意事项

在添加要抓取的信号时，不要给太多信号标注 debug mark 了。在线调试时抓取波形是需要消耗电路资源和存储单元的，因而能抓取的波形大小是受限的。应当只给必要的 debug 信号添加 debug mark。

在设置抓取波形的深度时，不宜太深。如果设定得太深，那么会存在存储资源不够，导致最后生成 bit 流和 debug 文件失败。

抓取的信号数量和抓取的深度是一对矛盾的变量。如果抓取深度相对较低，抓取的信号数量就可以相对多些。相对仿真调试，在线调试对调试思想和技巧有更高的要求，请好好整理思路，多多总结技巧。

特别强调以下几点：

- 触发条件的设定有很多组合，请根据需求认真考虑，好好设计。
- 通常只需要使用单触发模式，但循环触发有时候也很有用，必要时好好利用。
- 在线调试界面里很多按键，请自行学习，可以网上搜索资料，xilinx 官网上搜索，查找官方文档等。

最后再提醒一点，仿真通过但上板失败时，请先重点排查其他问题，最后再使用在线调试的方法。也就是仿真通过，上板异常时，应按以下流程排查：

- 1) 复核生成、下载的 bit 文件是否正确。
- 2) 复核仿真结果是否正确。
- 3) 检查实现时的时序报告（Vivado 界面左侧 “Timing Summary”）。
- 4) 认真排查综合和实现时的 Warning
- 5) 排查 RTL 代码规范，避免多驱动、阻塞赋值乱用。
- 6) 使用 Vivado 的逻辑分析仪进行板上在线调试

1.5 Lab1：硬件描述语言基础

1.5.1 实验目的

- 使用 Verilog 编写 RTL 程序，描述硬件设计
- 使用 Vivado 完成 Verilog 程序的设计、调试及波形仿真

1.5.2 实验环境

- Vivado 集成开发环境

1.5.3 实验内容

二选一多路选择器

多路选择器是常用的组合逻辑，根据选择信号匹配输入与输出。请自行阅读，理解 MUX 的原理。实验要求实现 2 位 2 路选择器。

节拍发生器

构建由 4 个 DFF 构成的 4 位节拍脉冲发生器，练习编写时序逻辑。可以采用循环移位的方式实现，也可以采用计数器 + 译码器方式实现。实验要求输出一个独热码信号 ($4'b1000$, $4'b0100$, $4'b0010$, $4'b0000$)。

模块接口设计

2-to-1 MUX 的信号说明如下：

名称	宽度	方向	描述
d0	2	IN	输入数据
d1	2	IN	输入数据
select	1	IN	选择信号
out	2	OUT	select 结果

节拍发生器的信号说明如下：

名称	宽度	方向	描述
clk	1	IN	时钟信号
rst	1	IN	复位时置输出信号为 1000
T	4	OUT	输出节拍信号

1.5.4 实验要求

实验预习

- 掌握多路选择器的工作原理，熟悉 Verilog 语法，编写组合逻辑。
- 掌握节拍发生器的原理，任选一种方式实现电路，采用 Verilog 编写时序逻辑。
- 阅读前置内容，了解 Vivado 开发调试环境的搭建方法。

完成实验内容

- 按照实验内容完成 MUX 的编写，自行编写测试用例，给出模拟测试波形，记录运行过程和结果，完成实验报告。
- 按照实验内容完成 4 位节拍发生器的编写，自行编写测试用例，给出模拟测试波形，记录运行过程和结果，完成实验报告。

1.5.5 实验帮助信息

实现多路选择器

`mux_2_1.v` 中，你可以采用 `case`、`if` 甚至 `assign` 的方式实现选择逻辑。它们在实现中有有些不同，请自行查阅资料，或查看资源占用情况来理解。

实现节拍发生器

`counter.v` 中，你可能用到如下描述。

```
T <= {T[0], T[3:1]};
```

推荐采用同步复位。

1.5.6 选做：硬件实现

自行设计输入输出方式，如拨码开关 + 数码管等，编写顶层文件，展示你的设计。

添加约束文件，将编写的代码进行综合布局布线，并烧写比特流，上板验证设计。对于远程 FPGA 实验平台，只有最后一步烧写配置与本地 FPGA 实验平台不同，前面的各个步骤都是完全一样的。

备注：

- 在 Pre 和 Appendix 的相关章节中给出了 FPGA 上板实现的例子及远程实验平台的介绍与使用，请参阅。

1.6 Lab2：基本组合逻辑设计

1.6.1 实验目的

- 掌握 Verilog 语言基本知识及 Vivado 集成开发环境
- 掌握并理解算术逻辑单元（ALU）的原理和设计
- 掌握仿真测试方法

1.6.2 实验环境

- Vivado 集成开发环境

1.6.3 实验内容

加法器

请自行阅读，理解计算机中加法器的原理。实验要求设计有 2 个 32 位数输入和 1 个进位输入，产生 1 个 32 位的加法和结果和 1 个向高位的进位的加法器。

ALU

基于前面完成的加法器设计，设计一个简单的算术逻辑单元，能够执行以下 16 种算术逻辑运算操作。实验要求 ALU 支持如下 16 种运算操作。

序号	运算操作	序号	运算操作
1	$F=A$ 加 B	9	$F=/A$
2	$F=A$ 加 B 加 Cin	10	$F=/B$
3	$F=A$ 减 B	11	$F=A+B$
4	$F=A$ 减 B 减 Cin	12	$F=AB$
5	$F=B$ 减 A	13	$F=A\odot B$
6	$F=B$ 减 A 减 Cin	14	$F=A\oplus B$
7	$F=A$	15	$F=/(AB)$
8	$F=B$	16	$F=0$

备注：

- / 是逻辑非操作， \odot 是逻辑同或操作， \oplus 是逻辑异或操作。
-

模块接口设计

加法器的信号说明如下：

- 定义三个输入信号 A、B、Cin。其中，A、B 为 32 位运算数，Cin 为进位。
- 定义两个输出信号 F, Cout。其中 F 为运算结果，Cout 为结果进位。

ALU 的信号说明如下：

- 定义四个输入信号 A、B、Cin、Card。其中，A、B 为 32 位运算数，Card 为 5 位运算操作码，Cin 为进位。
- 定义三个输出信号 F, Cout, Zero，其中 F 为运算结果，Cout 为结果进位，Zero 为零标志。
- 要求根据 16 种运算操作对运算操作码 Card 进行编码，并实现这 16 种运算操作。

1.6.4 实验要求

实验预习

- 掌握定点加法的工作原理，了解实验开发所需的软硬件平台。
- 掌握 ALU（算术逻辑单元）的 16 种算术逻辑运算操作的意义，完成这 16 种算术逻辑运算操作的编码。

完成实验内容

- 按照实验内容完成加法器的设计，自行编写测试用例，给出模拟测试波形，记录运行过程和结果，完成实验报告。
- 按照实验内容完成 ALU（算术逻辑单元）的设计，自行编写测试用例，给出模拟测试波形，记录运行过程和结果，完成实验报告。

1.6.5 实验帮助信息

面向硬件电路的设计思维方式

明确 Verilog 是一门 **HDL (Hardware Description Language) ** 语言，也即描述一个真正可以物理上实现的电路，这与之前的软件设计是有很大的不同的。典型的例子如：非阻塞赋值 `=>` 时完全不需要顺序描述，寄存器的 `always` 块内不允许阻塞赋值以及多驱动等。

明确面向硬件电路的设计思维方式的核心实际上就是“**数据通路 (Datapath) + 控制逻辑 (Control Logic)**”。

推荐一些写法：

- 完全使用 `assign` 描述组合逻辑，简洁清晰。这是龙芯风格的描述，当然你也可以使用 `always @(*)`。
- 完全使用同步操作，即所有的寄存器变化都发生在 `always @ (posedge clk)` 内。

规范的代码将使得软件能够清晰理解你的意图，进而得到正确的电路实现。

ALU 的设计

ALU 作为 CPU 中的算逻运算单元，需要帮助完成 CPU 中加减运算指令，比较运算指令，移位运算指令、逻辑运算指令等操作。因此，ALU 的设计需要实现相应的多条数据通路。外部模块将参加运算的源操作数同时传输给这些数据通路，源操作数在数据通路中的流动过程就是运算过程。最后根据控制逻辑的信号，配合多路选择电路从多条数据通路中选择出合适的结果作为 ALU 的输出。

alu.v 的示例代码如下。

```

`define ADD      5'b00001
`define SUB      5'b00010
`define SLT      5'b00011
`define SLTU     5'b00100
.....
module alu (
    input [31:0]   A   ,
    input [31:0]   B   ,
    input          Cin ,
    input [4 :0]   Card,
    output [31:0]  F   ,
    output          Cout,
    output          Zero
);

    wire [31:0] add_result;
    wire [31:0] sub_result;
    wire [31:0] slt_result;
    wire [31:0] sltu_result;
    .....

    assign add_result = /*TODO*/;
    assign sub_result = /*TODO*/;
    assign slt_result = /*TODO*/;
    assign sltu_result = /*TODO*/;
    .....

    assign F = ({32{alusel == `ADD}} & add_result) |
               ({32{alusel == `SUB}} & sub_result) |
               ({32{alusel == `SLT}} & slt_result) |
               ({32{alusel == `SLTU}} & sltu_result) |
               .....

    assign Cout = .....
    assign Zero = .....

endmodule

```

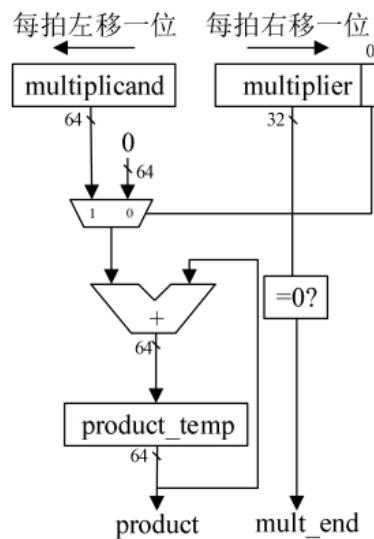
1.6.6 选做：定点乘法器

实验目的

- 理解定点乘法的不同实现算法的原理，掌握基本实现算法。
 - 熟悉并运用 Verilog 语言进行电路设计。
 - 为后续设计 CPU 的实验打下基础。

实验内容

请自行学习定点乘法器的实现算法及原理。定点乘法器有多种实现，实验要求实现迭代乘法器，其结构如图所示。



乘数每次右移一位，根据最低位，判断是加被乘数移位后的值还是加 0，不停地累加，最后就得到乘积了。

可以看到迭代乘法是用多次加法完成乘法操作的，故需要多拍时间，其结束标志为乘数移位后为 0，故对于 32 位乘法，最多需要 32 拍才能完成一次乘法。

实验要求

- 掌握定点乘法的多种实现算法的原理。
 - 确定定点乘法的输入输出端口设计。
 - 编写乘法器 Verilog 代码，自行编写测试用例，给出模拟测试波形，记录运行过程和结果，完成实验报告。

实验帮助信息

其他乘法器

备注:

- Booth 乘法器：更适合硬件实现的乘法器算法。
 - 华莱士树：通过面积换时间的方式实现并行加法。感兴趣的同学可以自行学习华莱士树的结构。
-

1.6.7 选做：ALU 硬件实现

确保仿真无误后，设计一个外围模块，并将你的 ALU 设计适配到其中，外围模块的输出可以为数码管，也可以为 LCD 显示屏等。根据输入的源操作数、操作码，显示正确的运算结果。

1.7 Lab3：内存与寄存器堆

1.7.1 实验目的

- 熟悉并掌握 MIPS 计算机中寄存器堆的原理和设计方法
- 理解源操作数/目的操作数的概念
- 理解 RAM 读取、写入数据的过程
- 掌握调用 xilinx 库 IP 实例化 RAM 的设计方法

1.7.2 实验环境

- vivado 集成开发环境

1.7.3 实验内容

寄存器堆

学习 MIPS 计算机中寄存器堆的设计原理。实验要求设计寄存器堆，包含 1 个写端口和 2 个读端口。

RAM

学习 RAM 的读写时序，并定制一块 RAM IP 核，在顶层文件中实例化。实验要求实现同步 RAM，定制深度为 65536，定制宽度为 32。

模块接口设计

寄存器堆的信号说明如下。

名称	宽度	方向	描述
clk	1	IN	时钟信号
raddr1	5	IN	寄存器堆读地址 1
rdata1	32	OUT	寄存器堆读返回数据 1
raddr2	5	IN	寄存器堆读地址 2
rdata2	32	OUT	寄存器堆读返回数据 2
we	1	IN	寄存器堆写使能
waddr	5	IN	寄存器堆写地址
wdata	32	IN	寄存器堆写数据

RAM 顶层模块的接口信号说明如下。

名称	宽度	方向	描述
clk	1	IN	时钟信号
ram_wen	1	IN	同步 RAM 写使能，置位时写入
ram_addr	16	IN	同步 RAM 地址信号，表示读/写地址
ram_wdata	32	IN	同步 RAM 写数据信号，表示写入数据
ram_rdata	32	OUT	同步 RAM 读数据信号，表示读出数据

1.7.4 实验要求

实验预习

- 掌握 MIPS 寄存器堆的工作原理
- 根据给定的输入输出端口设计寄存器堆
- 掌握存储器的工作原理，理解同步 RAM 与异步 RAM 的区别。
- 掌握使用 xilinx 库 IP 进行设计的方法。

完成实验内容

- 按照实验内容完成寄存器堆的设计，自行编写测试用例，给出模拟测试波形，记录运行过程和结果，完成实验报告。
- 按照实验内容实例化同步 RAM IP，使用给定的测试用例，给出模拟测试波形，观察时序特征，对比读写行为的异同，完成实验报告。

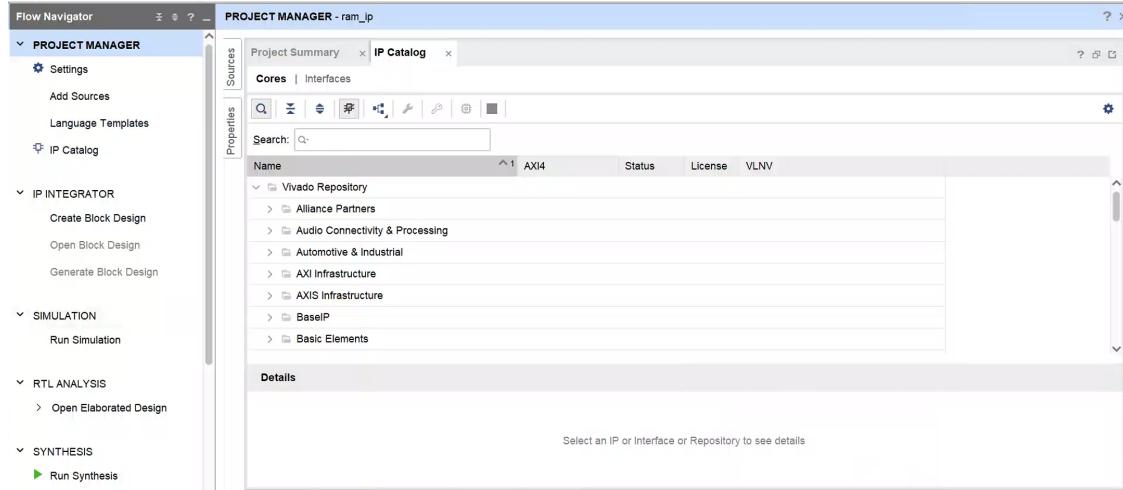
1.7.5 实验帮助信息

regfile

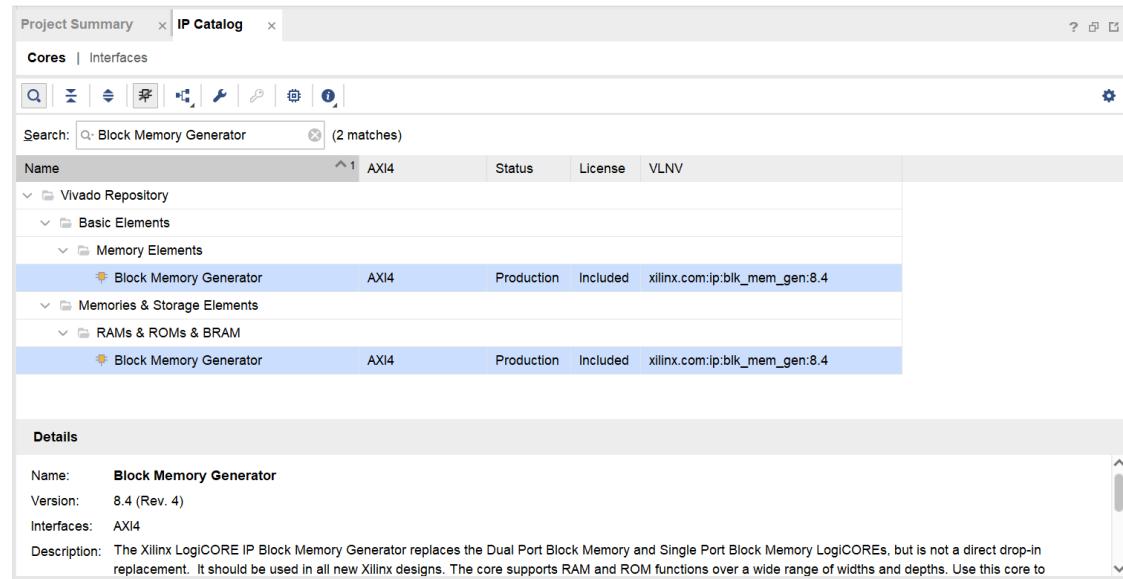
regfile.v 的正确实现对于后续实现 CPU 十分重要，力求简洁。实现中可以采用寄存器数组的形式进行描述。

RAM IP

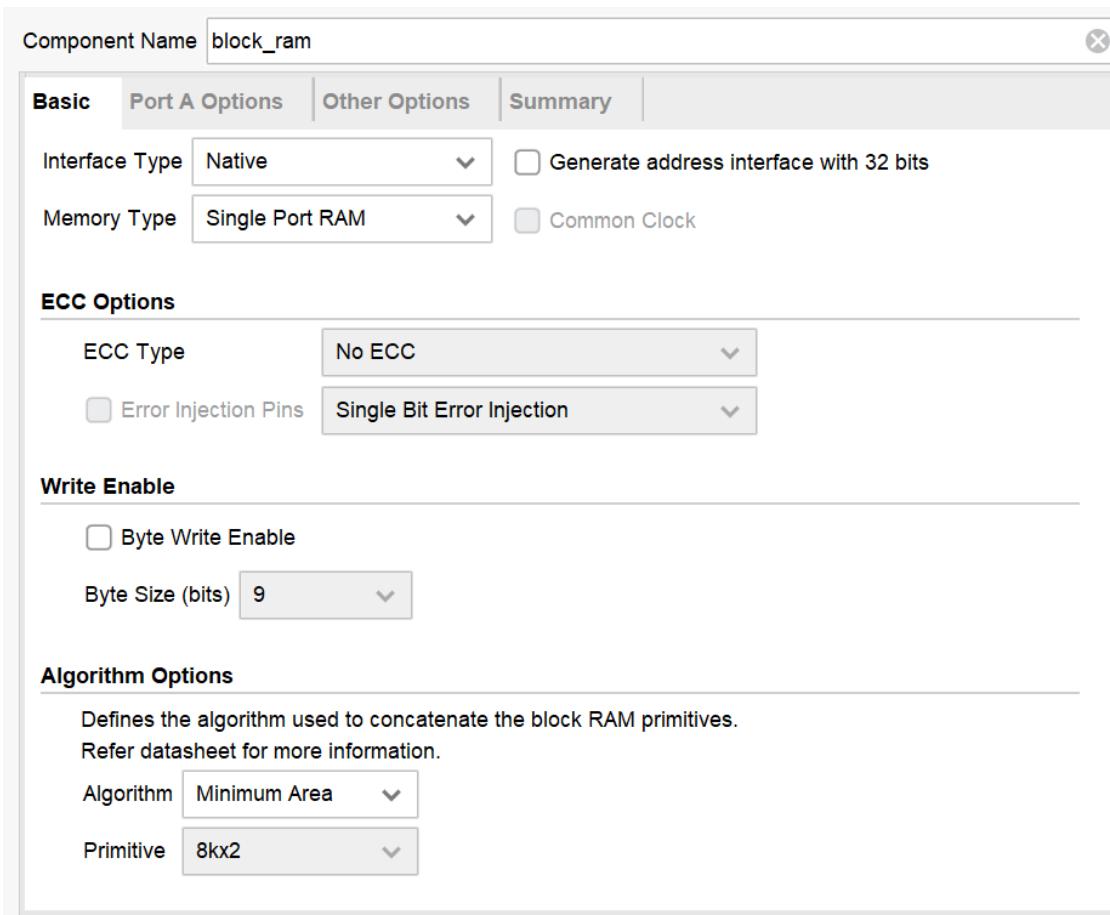
新建工程，在 PROJECT MANAGER 中点击 IP Catalog，打开 IP 目录。



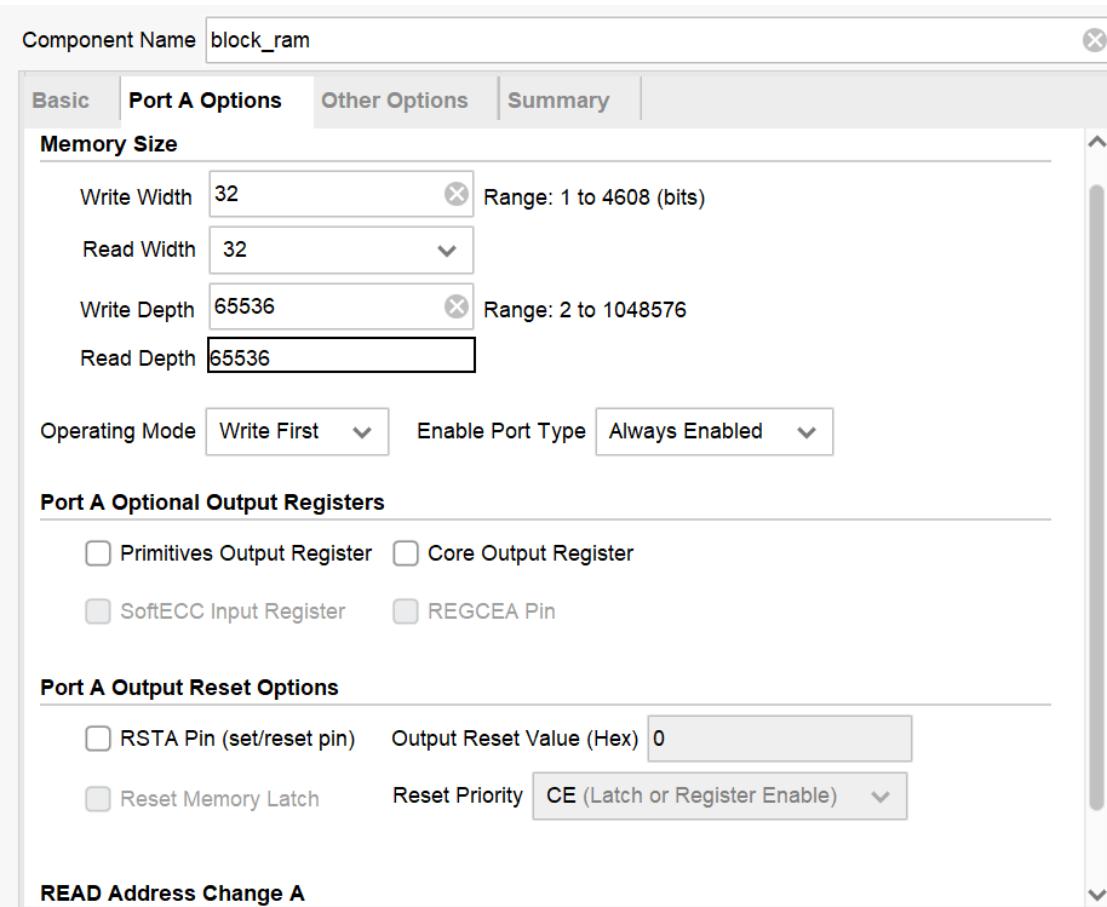
在搜索框中查找 Block Memory Generator。



在 Basic 选项卡下，将 IP 重命名为 block_ram，内存类型设置为 Single Port RAM。注意不要勾选 Byte Write Enable。



在 Port A Options 选项卡下，设置 RAM 深度及宽度。使能端口设置为 Always Enabled，不要勾选 Primitives Output Register，输出不需要打拍。



其他设置保持默认即可。

备注:

- vivado 提供两种类型的 RAM IP - BRAM 和 DRAM (Distributed RAM)。在综合实现后的资源占用报告中，能够观察到 LUT RAM 一项，它就是 DRAM/ROM 及移位寄存器占用资源的总和。
- 若实现异步 RAM，请自行尝试实例化 DRAM IP，并与 BRAM 比较，观察不同时序下的波形特征。

bram_top.v 作为同步 RAM 的源码顶层文件，其代码如下。

```
module ram_top (
    input      clk      ,
    input [15:0] ram_addr ,
    input [31:0] ram_wdata,
    input      ram_wen  ,
    output [31:0] ram_rdata
);

block_ram block_ram (
    .clka (clk      ),
    .wea (ram_wen  ),
    .addr (ram_addr ),
    .dina (ram_wdata),
    .douta(ram_rdata)
);
```

(续下页)

(接上页)

```
) ;
endmodule
```

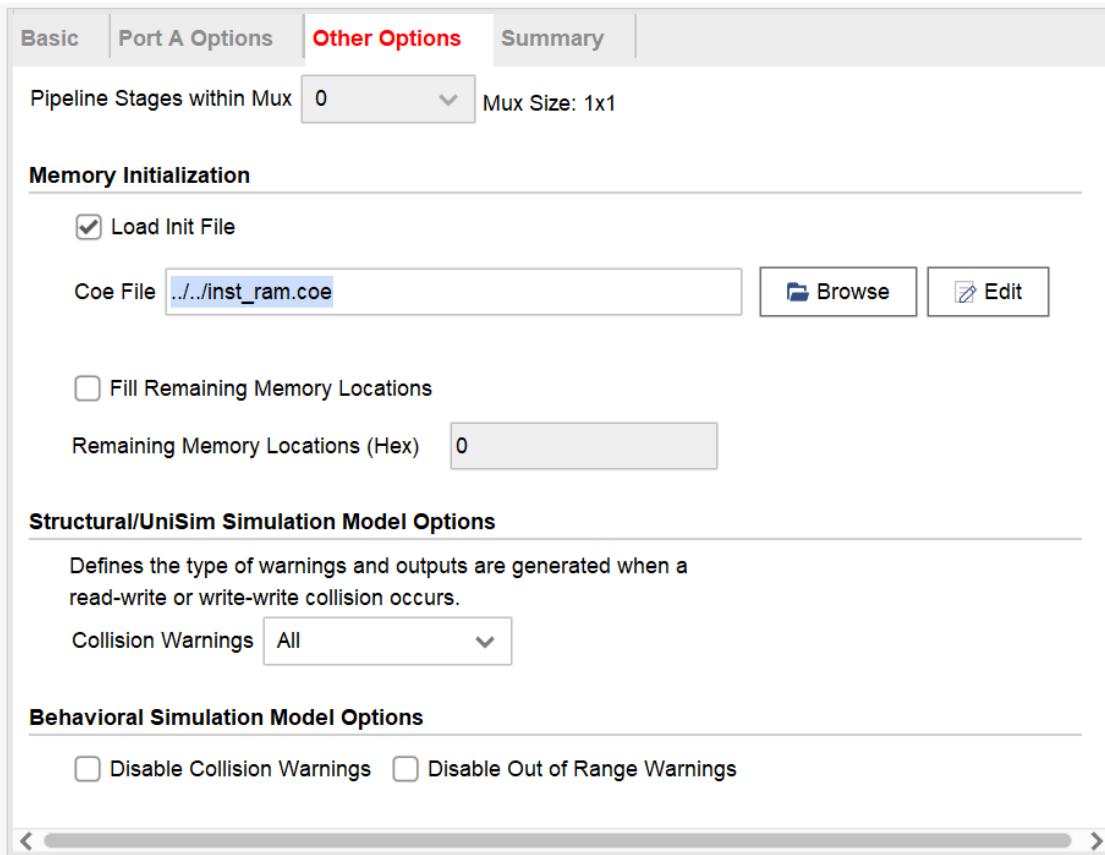
1.7.6 选做：指令 ROM

在后续的 CPU 实验中，我们将使用 BRAM 实现指令 ROM。仍然调用 ram_top，将写使能恒置为 0，并加载.coe 格式的初始化文件。

在 inst_ram.txt 前加入两行：

```
memory_initialization_radix = 16;
memory_initialization_vector =
```

并修改文件后缀为.coe，装载入 BRAM IP 核。在 Other Options 选项卡下，勾选 Load Init File，并填入.coe 文件路径。



编写合适的测试用例，给出模拟测试波形，记录运行过程和结果。

1.8 Lab4：给定指令系统的处理器设计

1.8.1 实验目的

- 掌握 Vivado 集成开发环境
- 掌握 Verilog 语言
- 掌握 FPGA 编程方法及硬件调试手段
- 深刻理解处理器结构和计算机系统的整体工作原理

1.8.2 实验环境

- Vivado 集成开发环境
- 龙芯 Artix-7 实验平台

1.8.3 实验内容

- 根据《计算机体系结构》第五章里 MIPS 指令集的基本实现，设计并实现一个符合实验指令的非流水处理器，包括 Verilog 语言的实现和 FPGA 芯片的编程实现，要求该处理器可以通过所提供的自动测试环境。

处理器功能

本实验的任务是设计一个简单的 RISC 处理器，该处理器是在给定的指令集（与 MIPS32 类似）下构建的，支持 12 条指令。假定存储器分为数据缓冲存储器和指令缓冲存储器，且都可以在一个时钟周期内完成一次同步存取操作，时钟信号和 CPU 相同。处理器的指令字长为 32 位，包含 32 个 32 位通用寄存器 R0~R31，1 个 32 位的指令寄存器 IR 和 1 个 32 位的程序计数器 PC，1 个 256×32 位指令缓冲存储器，1 个 256×32 位的数据缓冲存储器。

指令系统定义

处理器所支持的指令包括 LW, SW, ADD, SUB, SLL, AND, OR, XOR, SLT, MOVZ, BNE, J。

其中仅有 LW 和 SW 是字访存指令，所有的存储器访问都通过这两条指令完成；ADD、SUB、SLL、AND、OR、XOR、SLT、MOVZ 是运算指令，它们都在处理器内部完成；BNE 是分支跳转指令，根据寄存器的内容进行相对跳转；J 是无条件转移指令。

指令说明

关于汇编指令的说明：

- rs 意味着 source register，rd 意味着 destination register；
- rt 有时（如运算指令）意味着第二个 source register，有时（如访存指令）意味着 target (source/destination) register；
- 我们使用 [rs] 表示寄存器 rs 的内容；

另外，所有算数运算指令都执行有符号运算。

运算指令

(1) 加法指令 ADD rd, rs, rt

31	26 25	21 20	16 15	11 10	0
6	5	5	5	11	
000000	rs	rt	rd	00000_100000	

该指令将两个源寄存器内容相加，结果送回目的寄存器的操作。

具体为: [rd] <- [rs] + [rt]

(2) 减法指令 SUB rd, rs, rt

31	26 25	21 20	16 15	11 10	0
6	5	5	5	11	
000000	rs	rt	rd	00000_100010	

该指令将两个源寄存器内容相减，结果送回目的寄存器的操作。

具体为: [rd] <- [rs] - [rt]

(3) 与运算指令 AND rd, rs, rt

31	26 25	21 20	16 15	11 10	0
6	5	5	5	11	
000000	rs	rt	rd	00000_100100	

该指令将两个源寄存器内容相与，结果送回目的寄存器的操作。

具体为: [rd] <- [rs] & [rt]

(4) 或运算指令 OR rd, rs, rt

31	26 25	21 20	16 15	11 10	0
6	5	5	5	11	
000000	rs	rt	rd	00000_100101	

该指令将两个源寄存器内容相或，结果送回目的寄存器的操作。

具体为: [rd] <- [rs] | [rt]

(5) 异或指令 XOR rd, rs, rt

31	26 25	21 20	16 15	11 10	0
6	5	5	5	11	
000000	rs	rt	rd	00000_100110	

该指令将两个源寄存器内容相异或，结果送回目的寄存器的操作。

具体为: [rd] <- [rs] ⊕ [rt]

(6) 小于指令 SLT rd, rs, rt

31	26 25	21 20	16 15	11 10	0
000000	rs	rt	rd	00000_101010	

6 5 5 5 11

该指令将两个源寄存器内容相比较，结果决定目的寄存器的值。

具体为：[rd] <- [rs] < [rt] ? 1 : 0

(7) 条件移动指令 MOVZ rd, rs, rt

31	26 25	21 20	16 15	11 10	0
000000	rs	rt	rd	00000_001010	

6 5 5 5 11

该指令根据其中一个源寄存器内容，决定另一个源寄存器的值是否写回目的寄存器。

具体为：if ([rt] == 0) then [rd] <- [rs]

备注：注意：当 [rt] != 0 时，不对寄存器 rd 进行任何写回操作。

(8) 移位指令 SLL rd, rt, sa

31	26 25	21 20	16 15	11 10	6 5	0
000000	00000	rt	rd	sa	00000	

6 5 5 5 5 6

该指令根据 sa 字段指定的位移量，对寄存器 rt 的值进行逻辑左移，结果决定目的寄存器的值。

具体为：[rd] <- [rt] « sa

访存指令

警告：使用访存指令时，计算后的目标地址必须是访问数据单元大小的整数倍。对于 LW 和 SW 指令， $([\text{base}] + \text{offset}) \% 4 == 0$

(1) 存数指令 SW rt, offset(base)

31	26 25	21 20	16 15	0
101011	base	rt	offset	

6 5 5 16

该指令将寄存器 rt 的内容存于主存单元中，对应的地址由 16 位偏移地址 offset 经符号拓展加上 base 的内容生成。

具体操作为：Mem[[base] + offset] <- [rt]

(2) 取数指令 LW rt, offset(base)

31	26 25	21 20	16 15	0
6	5	5	16	
100011	base	rt	offset	

该指令将主存单元中的内容存于寄存器 rt，对应的地址由 16 位偏移地址 offset 经符号拓展加上 base 内容生成。

具体操作为：[rt] <- Mem[[base] + offset]

转移类指令**警告：**

- 和 MIPS32 指令集不同，我们给出的跳转指令没有延迟槽。
- 跳转指令使用 NPC（也就是 PC+4）参与跳转地址运算。

(1) 条件转移（不相等跳转）指令 BNE rs, rt, offset

31	26 25	21 20	16 15	0
6	5	5	16	
000101	rs	rt	offset	

该指令根据寄存器 rs 和 rt 的内容决定下一条指令的地址，若两个寄存器内容不相等，则 16 位偏移 offset 扩充为 32 位，左移 2 位后与 NPC 相加，作为下一条指令的地址，否则程序按原顺序执行。

具体操作为：PC <- ([rs] != [rt]) ? [sign_extend(offset) << 2 + NPC] : NPC

(2) 无条件转移指令 J target

31	26 25	0
6	26	
000010	instr_index	

该指令改变下一条指令的地址，地址由指令中的 26 位形式地址 instr_index 左移 2 位作为低 28 位，和 NPC 的高 4 位拼接而成。

具体操作为：PC <- (NPC[31:28]) ## (instr_index << 2)

1.8.4 实验要求

要求根据以上给定的指令系统设计处理器，包括指令格式设计、操作的定义、Verilog 语言的实现及 FPGA 编程实现。处理器设计实验要求按指定阶段进行。

实验预习

在实验开始前给出处理器的**设计方案**，设计方案要求包括：

- 指令格式设计
- 处理器结构设计框图（要求精确到信号以及信号的位宽）及功能描述
- 各功能模块结构设计框图及功能描述
- 各模块输入输出接口信号定义（以表格形式给出）

完成实验内容

Verilog 语言实现：要求采用结构化设计方法，用 Verilog 语言实现处理器的设计。设计包括：

- 各模块的详细设计（包括各模块功能详述，设计方法，Verilog 语言实现等）
- 各模块的功能测试（每个模块作为一个部分，包括测试方案、测试过程和测试波形等）
- 系统的详细设计（包括系统功能详述，设计方法，Verilog 语言实现等）
- 系统的功能测试（包括系统整体功能的测试方案、测试过程和测试波形等）

备注：

- 要求所有同学都通过基础验收指令测试（完成仿真即可），同一小组中仅需一人通过所有附加验收指令测试。
-

1.8.5 处理器测试环境

要求使用我们提供的处理器测试环境对 CPU 进行测试，包括仿真测试和板上测试两部分。

使用要求

要求每个同学的 CPU 对外暴露以下接口（信号名称与位数均不可改变）：

```
module cpu (
    input          clk      ,  // clock, 100MHz
    input          resetn   ,  // active low

    // debug signals
    output [31:0]  debug_wb_pc   ,  // 当前正在执行指令的PC
    output          debug_wb_rf_wen ,  // 当前通用寄存器组的写使能信号
    output [4 :0]   debug_wb_rf_addr,  // 当前通用寄存器组写回的寄存器编号
    output [31:0]   debug_wb_rf_wdata // 当前指令需要写回的数据
);

/*TODO: 完成非流水CPU的设计代码*/
```

(续下页)

(接上页)

endmodule**警告:**

- 只允许按照给定的接口格式去设计 CPU, 不允许更改接口格式
- 所有信号在时钟上升沿采样, 在写使能为 0 时, debug_rf_addr、debug_rf_wdata 可以为任意值
- 复位后, 尽量保证上述接口信号不出现 X 或 Z
- 仅需添加待完成的 CPU, 其他部分不要修改

测试用例

针对本实验中的处理器测试环境, 使用我们提供的一个简单的测试用例对 CPU 进行测试, 包含了要求的 12 种指令。

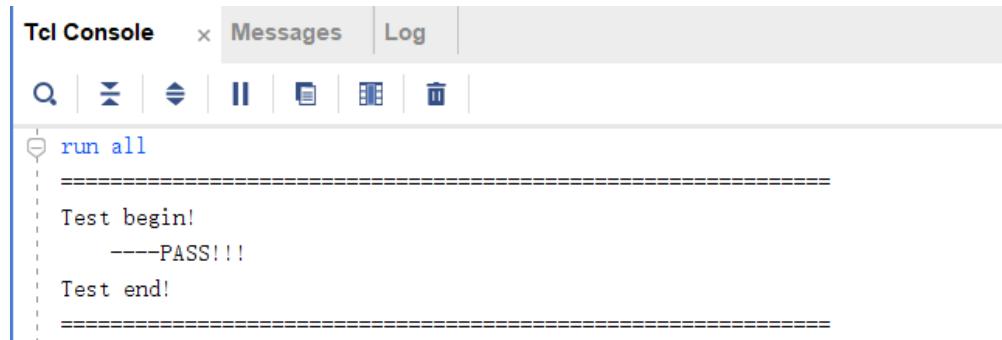
测试所用指令序列如下。

```
// 非流水线CPU实验测试

LW    $1,    4($0)
LW    $2,    8($0)
ADD   $3,    $1,    $2
SUB   $4,    $1,    $2
AND   $5,    $1,    $2
OR    $6,    $1,    $2
XOR   $7,    $1,    $2
SLT   $8,    $1,    $2
SLL   $9,    $1,    0x2 // $9 <- $1 << 2
SW    $1,    8($0)
SW    $2,    4($0)
BNE   $0,    $6,    0x2 // taken to 0x38
SW    $1,    0($0)
J     0          // back to start
J     0x10       // jumps to 0x40
SW    $1,    0($0)
LW    $0,    0($0)
NOP
J     0          // back to start
```

仿真结果

当所实现的 CPU 功能正确时，会在控制台打印 **PASS**，如下图所示。

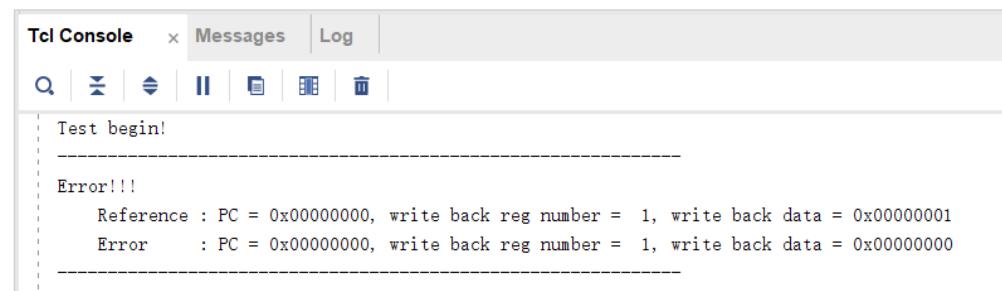


The screenshot shows a window titled "Tcl Console" with three tabs: "Messages" (selected), "Log", and "Tcl Console". Below the tabs is a toolbar with icons for search, zoom, and other functions. The main area displays the following text:

```
run all
=====
Test begin!
----PASS!!!
Test end!
=====
```

当所实现的 CPU 出现错误时，会在控制台打印错误信息，如下图所示。

其中 Reference 一行打印的是正确的 CPU 的输出信息，Error 一行打印的是你所实现的 CPU 的输出信息。



The screenshot shows a window titled "Tcl Console" with three tabs: "Messages" (selected), "Log", and "Tcl Console". Below the tabs is a toolbar with icons for search, zoom, and other functions. The main area displays the following text:

```
Test begin!
-----
Error!!!
Reference : PC = 0x00000000, write back reg number = 1, write back data = 0x00000001
Error      : PC = 0x00000000, write back reg number = 1, write back data = 0x00000000
-----
```

1.8.6 实验帮助信息

关于数据通路的复用

在进行 CPU 的总体设计时，需要着重考虑的一点是“**数据通路的复用**”。

毫无疑问，指令系统中的每条指令都需要一条数据通路来完成该指令的功能，不同指令的数据通路之间存在重合部分。如果设计出来的 CPU 的每条指令所需要的数据通路都不存在重合，那么该 CPU 浪费的资源多，性能也不好。

备注：

- MIPS 的 `nop` 编码为全 0，对应移位指令 `sll $0, $0, 0`，保证你的设计在执行这条指令时，不写寄存器，PC 正常 +4 即可。
-

1.8.7 选做：处理器下载

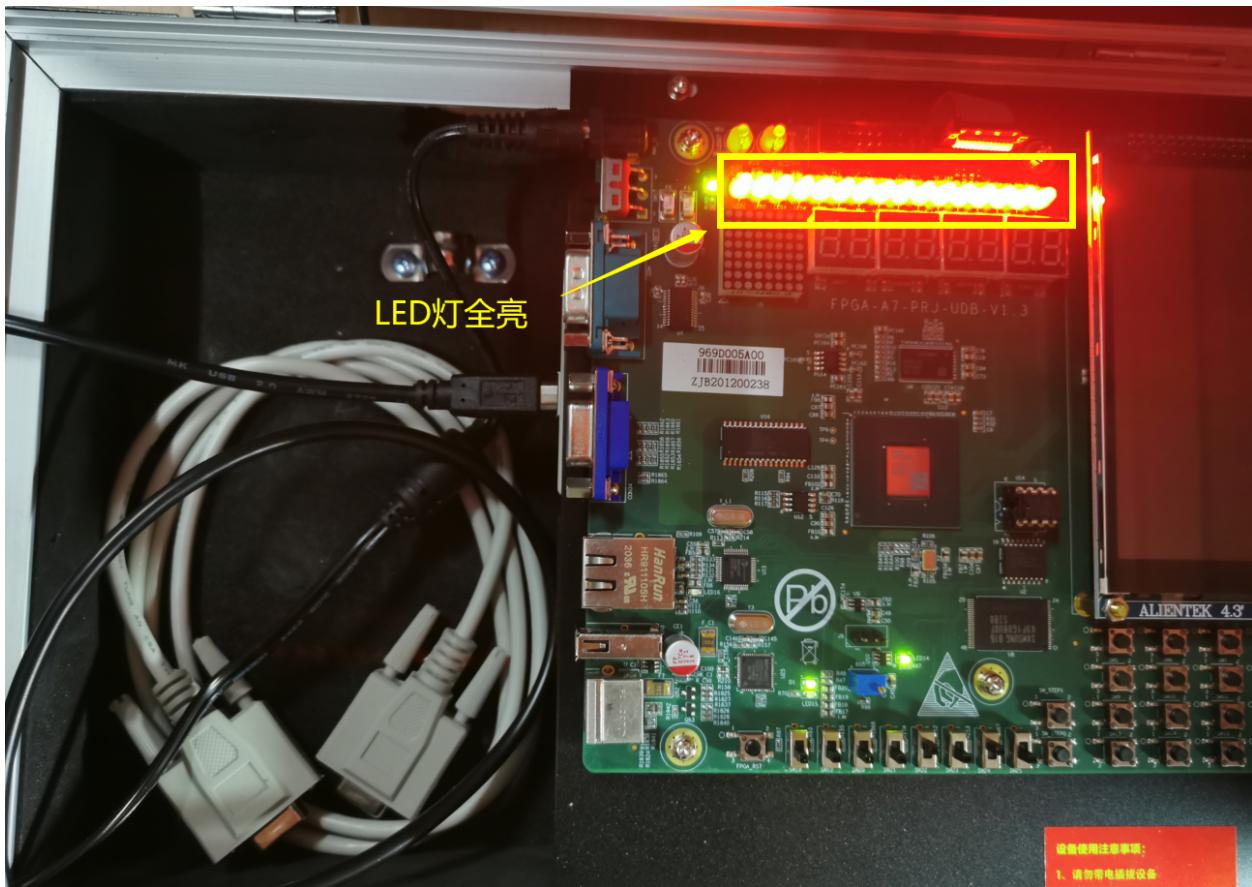
上板调试

FPGA 编程下载：将比特流下载到 Artix-7 实验板中。然后利用 Vivado 的上板调试功能观察 Artix-7 实验板的 FPGA 芯片中的实际运行，观察处理器内部运行状态，显式输出内部状态运行结果。

对处理器进行功能测试，记录运行过程和结果，完成实验报告：对处理器进行功能测试，使用提供的处理器功能测试环境完成处理器功能测试，并观察记录运行过程和结果，完成实验报告。

上板结果

CPU 通过仿真验证后，需要进行将设计下载到 Artix-7 开发板上观察设计正确性，当所实现的 CPU 功能正确时，开发板上的一排单色 LED 灯会同时亮起，如下图所示。

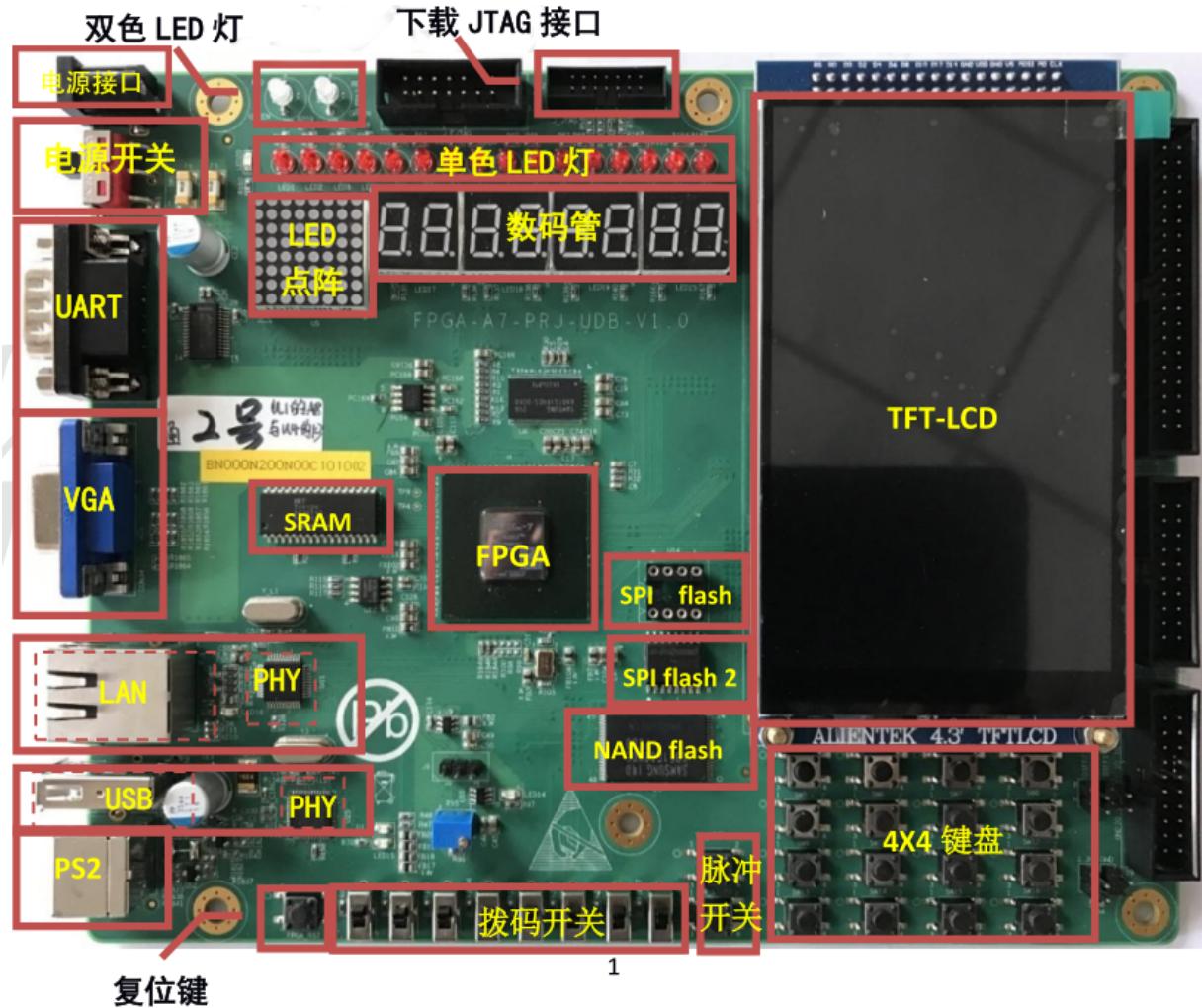


 此外，需要同学们抓取 debug_wb_pc、debug_wb_rf_addr、debug_wb_rf_wdata 以及 debug_wb_rf_wen 四个信号，观察上板时这些信号的波形图。

1.9 龙芯 Artix-7 实验板的使用

这里简单介绍一下本课程实验中如何使用提供的龙芯 Artix-7 实验板。

以下是实验板的实景图以及配套器件。





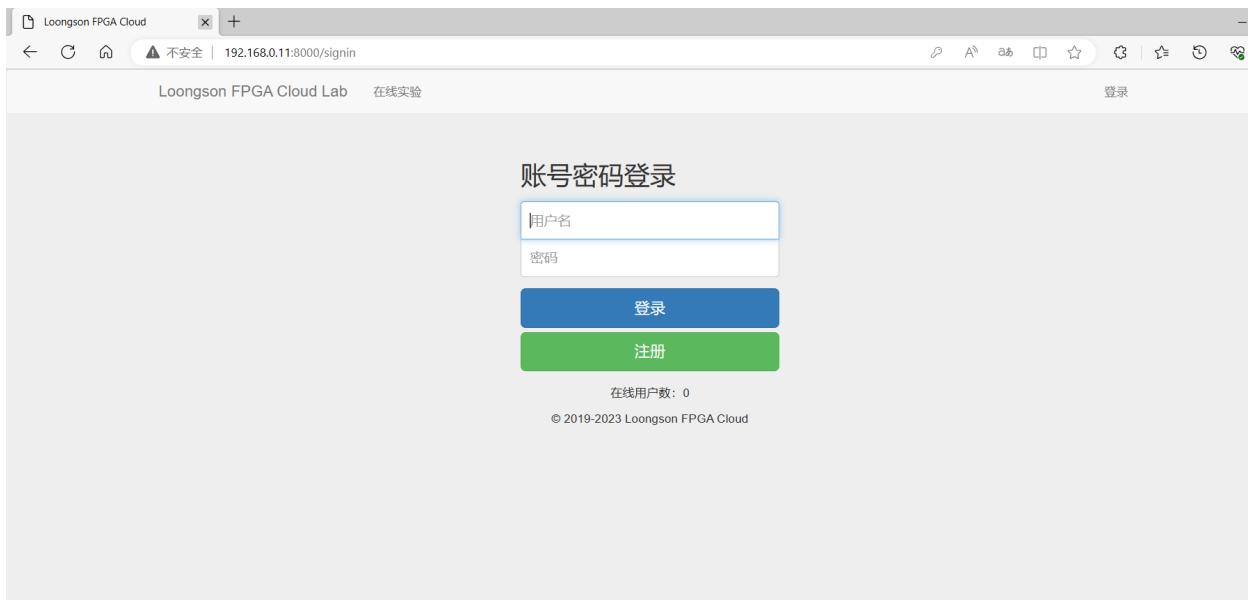
在本课程实验中，我们只需用到龙芯 Artix-7 实验板、USB 数据线以及电源适配器。在进行上板实验前，使用电源适配器将实验板与电源连接，并打开实验板电源开关，接着使用 USB 数据线将实验板与电脑连接，然后即可在 Vivado 中连接上实验板进行上板测试。

1.10 远程实验平台

- 远程实验平台的介绍和使用。

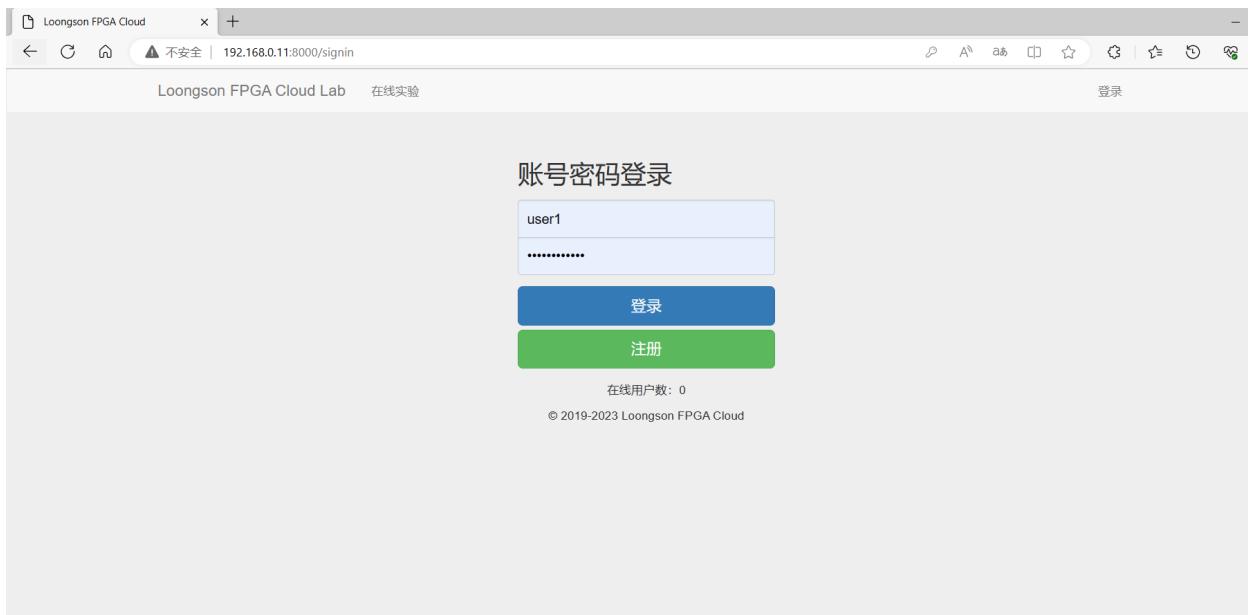
1.10.1 连接远程实验平台

- 连接实验室的无线网络。（密码为 **happyhacking**）
- 使用浏览器访问如下地址：
<http://192.168.0.11:8000/>
- 成功连接远程实验平台，看到如下图所示的登录界面。

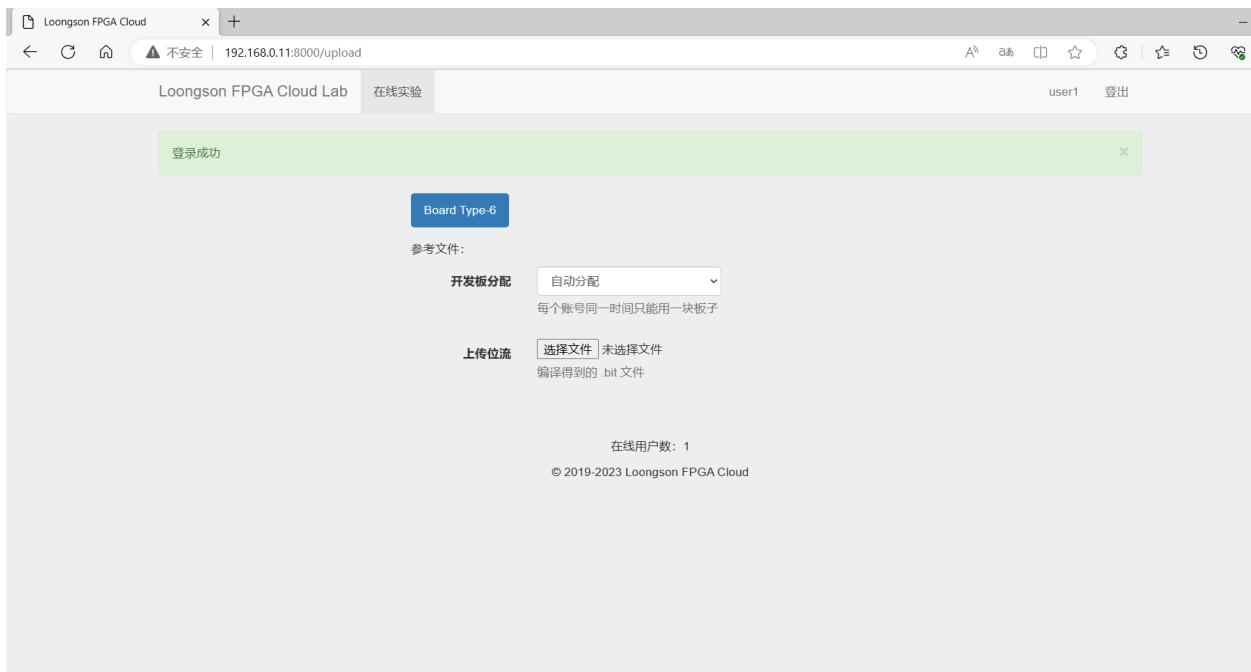


1.10.2 使用远程实验平台

- 登录远程实验平台，共计 20 组用户，用户名为 **user1** 到 **user20**。
- 所有用户密码均为 **happyhacking**。

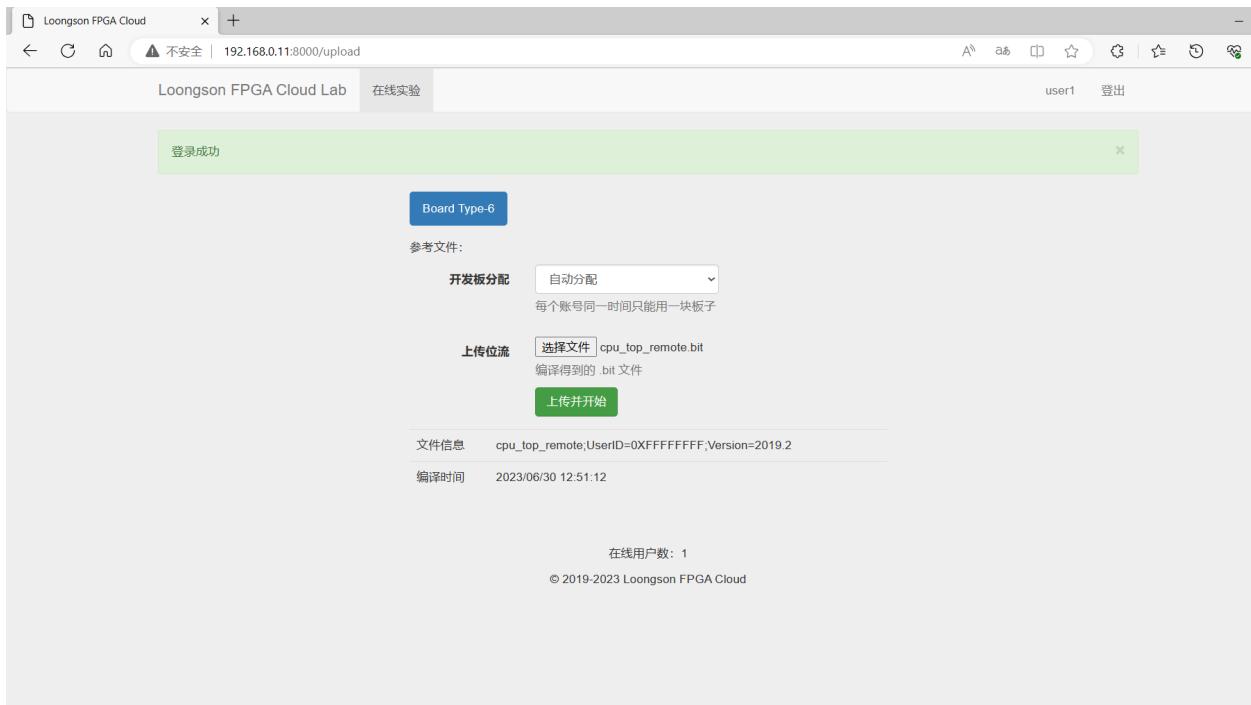


- 点击登录按钮后，可以看到如下图所示的界面：

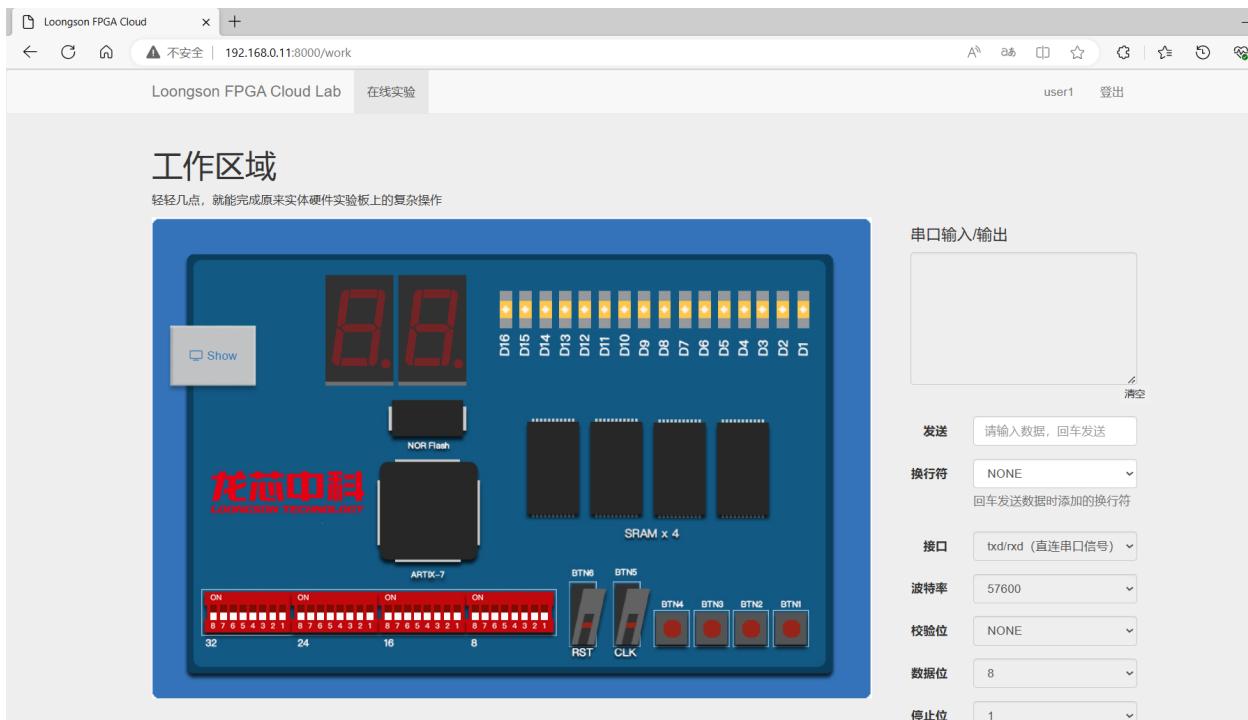


- 开发板分配项无需选择，点击“上传位流”选项右侧的“选择文件”按钮，选择 vivado 生成的 bit 流文件进行上传。

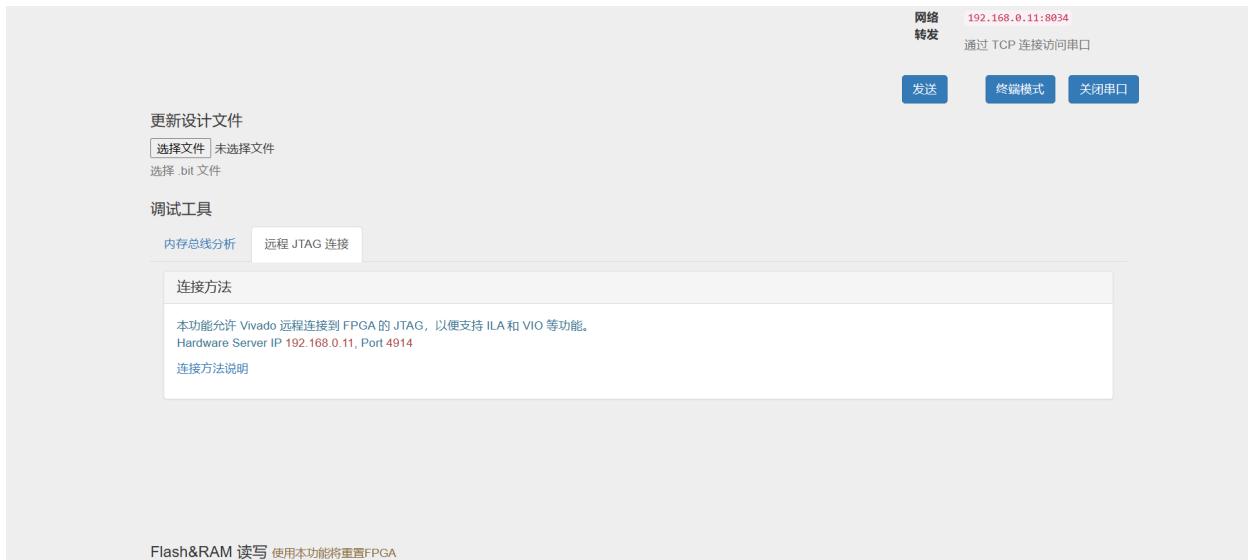
注:vivado 工程生成的 bit 流位置一般是 *your_project_name/your_project_name.runs/impl_1/your_top_name.bit*



- 点击上传并开始，即可进入远程实验板界面，在此界面可以模拟点击、拨动开关等操作进行实验。



- 向下滑动页面，可以利用 vivado 的远程 jtag 连接功能进行远程调试，具体连接方法见页面介绍。



1.10.3 注意事项

- 远程实验平台的引脚绑定、触发方式与实验箱均有所差异，具体来说需要使用新的 xdc 文件进行约束，并且需要对 rst、leds 信号等进行取反操作。

1.11 实验四测试环境说明

非流水线 CPU 实验测试环境。

1.11.1 测试样例

测试所用的完整指令序列如下。

```
// 非流水线 CPU 实验测试

0x0000: 8C 01 00 04    LW   $1,  4($0)
0x0004: 8C 02 00 08    LW   $2,  8($0)
0x0008: 00 22 18 20    ADD  $3,  $1,  $2
0x000c: 00 22 20 22    SUB  $4,  $1,  $2
0x0010: 00 22 28 24    AND  $5,  $1,  $2
0x0014: 00 22 30 25    OR   $6,  $1,  $2
0x0018: 00 22 38 26    XOR  $7,  $1,  $2
0x001c: 00 22 40 2A    SLT  $8,  $1,  $2
0x0020: 00 01 48 80    SLL  $9,  $1,  0x2 // $9 <- $1 << 2
0x0024: AC 01 00 08    SW   $1,  8($0)
0x0028: AC 02 00 04    SW   $2,  4($0)
0x002c: 14 06 00 02    BNE  $0,  $6,  0x2 // taken to 0x38
0x0030: AC 01 00 00    SW   $1,  0($0)
0x0034: 08 00 00 00    J    0           // back to start
0x0038: 08 00 00 10    J    0x10        // jumps to 0x40
0x003c: AC 01 00 00    SW   $1,  0($0)
0x0040: 8C 00 00 00    LW   $0,  0($0)
0x0044: 00 00 00 00    NOP
0x0048: 08 00 00 00    J    0           // back to start
```

数据 RAM 初始化内容如下。

```
00000000
00000001
00000002
00000003
00000004
00000005
00000006
00000007
```