

数字逻辑实验手册

(基于 DIGILENT BASYS3 口袋实验室)

吴志红 潘薇 卢晓春 主编

四川大学计算机学院

2019 年

目录

第一部分 实验平台介绍	2
1.熟悉 Basys3 实验板	2
1.1 注意事项	2
1.2 使用介绍	3
2.熟悉 Vivado 平台	7
2.1 开发环境介绍	7
2.2 基于模块化的设计流程	7
第二部分数字逻辑实验	20
3.实现布尔表达式	20
3.1 实验要求	20
3.2 实验步骤	20
4.加法器	23
4.1 实验要求	23
4.2 实验步骤	23
5.译码器和编码器	26
5.1 实验要求	26
5.2 实验步骤	26
6.多路选择器	28
6.1 实验要求	28
6.2 实验步骤	28
7.寄存器	30
7.1 实验要求	30
7.2 实验步骤	30
8.计数器	36
8.1 实验要求	36
8.2 实验步骤	36
9.流水灯	38
9.1 实验要求	38
9.2 实验步骤	38
附录 1: 关于 Vivado 中的引脚约束	40
附录 2: 了解 Verilog 语言	41

第一部分 实验平台介绍

1.熟悉 Bsys3 实验板

Bsys3 是围绕着一个 Xilinx Artix®-7 FPGA 芯片 XC7A35T-1CPG236C 搭建的, 它为学习 FPGA 和数字电路设计的用户提供了一个理想、随时可以使用的电子设计硬件平台。

Bsys3 的关键特性:

- 33280 个逻辑单元, 六输入 LUT 结构
- 1800Kbits 快速 RAM 块
- 5 个时钟管理单元, 均各含一个锁相环 (PLL)
- 90 个 DSP slices
- 内部时钟最高可达 450MHz
- 1 个片上模数转换器 (XADC)

Bsys3 板提供完整的硬件存取电路, 可以完成从基本逻辑到复杂控制器的设计。四个标准扩展连接器配合用户设计的电路板, 或 Pmods (Digilent 设计的 A/D 和 D/A 转换, 电机驱动器, 传感器输入等) 其他功能, 扩展信号的 8 针接口均采用 ESD 保护。附带的 USB 电缆, 提供电源和编程接口, 不需要额外的配置电源或其他编程电缆, 是一个入门或复杂数字电路系统设计的完美低成本平台。

1.1 注意事项

- 1) Bsys 3 只接受 5V 直流输入, 通过 USB 供电 (可以通过 EXT 供电, 但实验时不用次方式供电。EXT 供电方式后面会介绍);
- 2) Bsys 3 上电后, 不可带电插拔 USB 线, 不得动跳线开关;
- 3) Bsys 3 须远离水源;
- 4) 取用实验板时, 请卡住实验板边缘, 尽量不要接触芯片管脚及导线, 尤其是在上电时;
- 5) 归还实验板时, 请检查 USB 电缆是否放入盒子内。

1.2 使用介绍

1.2.1 概述

Bsys3 板上集成了大量的 I/O 设备和 FPGA 所需的支持电路,让您能够构建无数的设计而不需要其他器件。

实验板正面如图 1-1 所示。

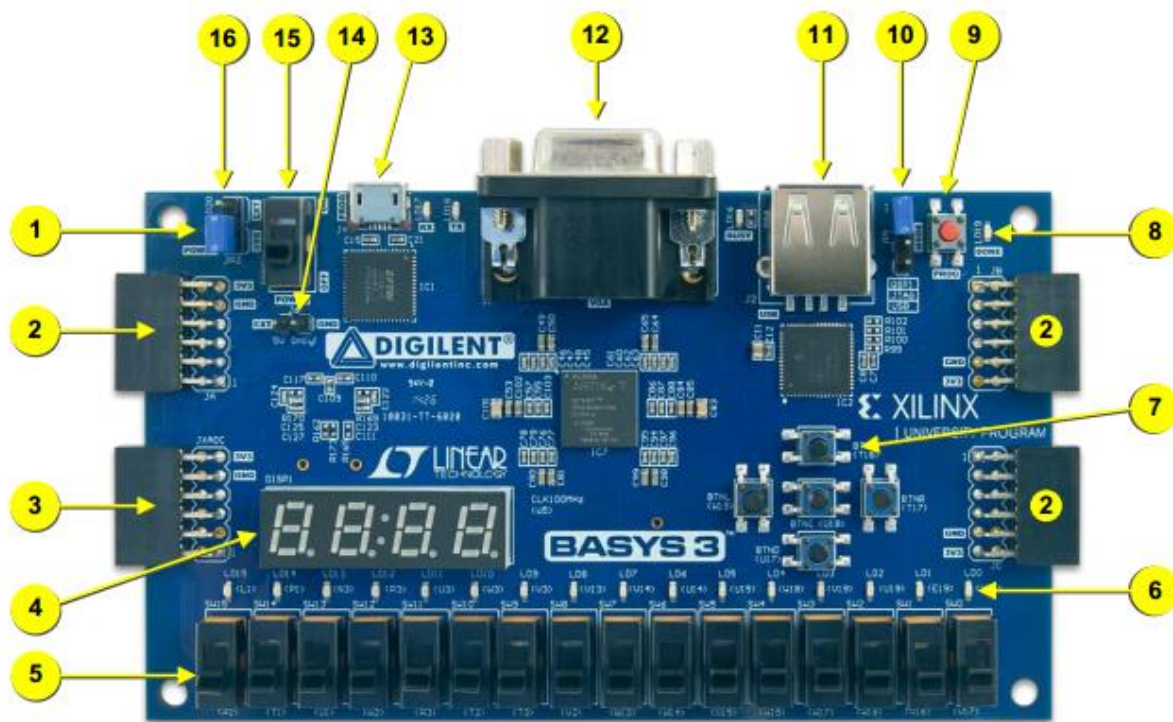


图 1-1 实验板顶视图

图 1-1 的各标号对应的 I/O 设备如表 1-1 所示。

表 1-1 实验板外设标号

序号	描述	序号	描述
1	电源指示灯	9	FPGA 配置复位按键
2	Pmod 数字信号接口	10	编程模式跳线柱
3	Pmod 模拟信号接口	11	USB 接口
4	4 位 7 段数码管	12	VGA 显示接口
5	16 个拨码开关	13	UART/JTAG 共用 USB 接口
6	16 个 LED 灯	14	外部电源接口

7	5 个按键开关	15	电源开关
8	FPGA 编程指示灯	16	电源选择跳线柱

1.2.2 供电方式

Basys3 板可以通过 2 种方式进行供电，一种是通过 J4 的 USB 端口供电，另一种是通过 J6 的接线柱进行外部供电（5V）。通过 JP2 跳线帽（图 1-1 的 16，表 1-1 进行了说明）的不同选择进行供电方式的选择。电源开关通过 SW16 进行控制，LD20 为电源开关的指示灯。如图 1-2 所示。

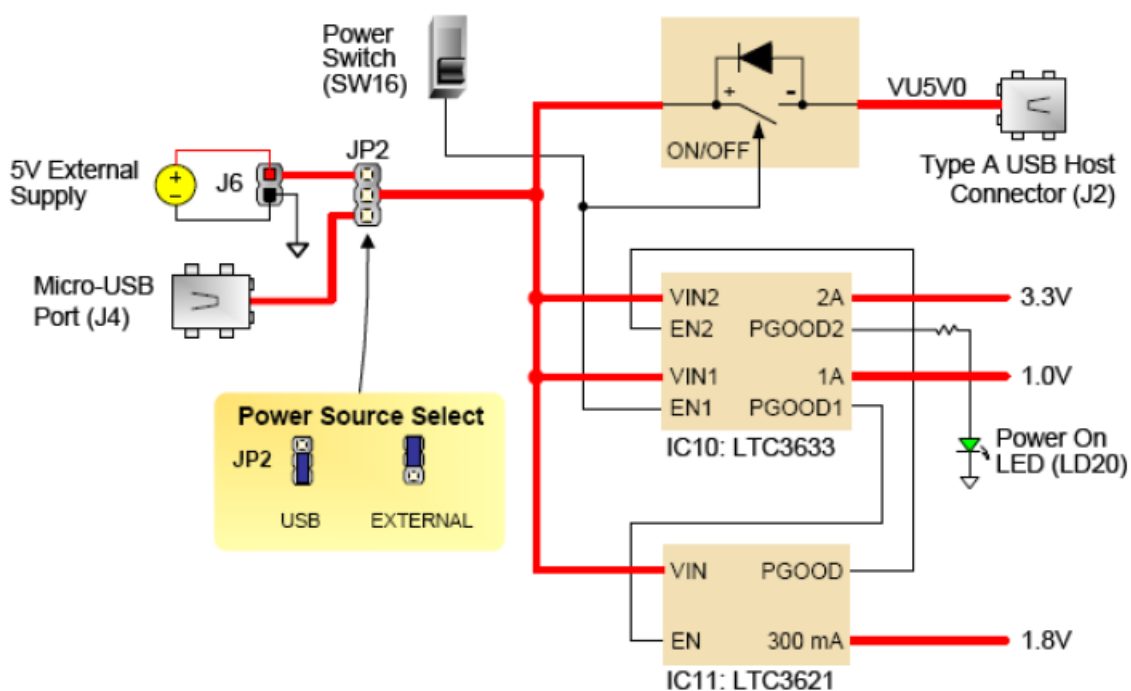


图 1-2 USB 接口电路

在本实验中，实验板采用 USB 供电。

1.2.3 LED 灯电路

LED 灯部分的电路如图 1-3 所示。当 FPGA 的输出为高电平时，相连接的 LED 灯点亮；否则，LED 灯熄灭。板上配有 16 个 LED 灯，可用作标志显示或结果显示。

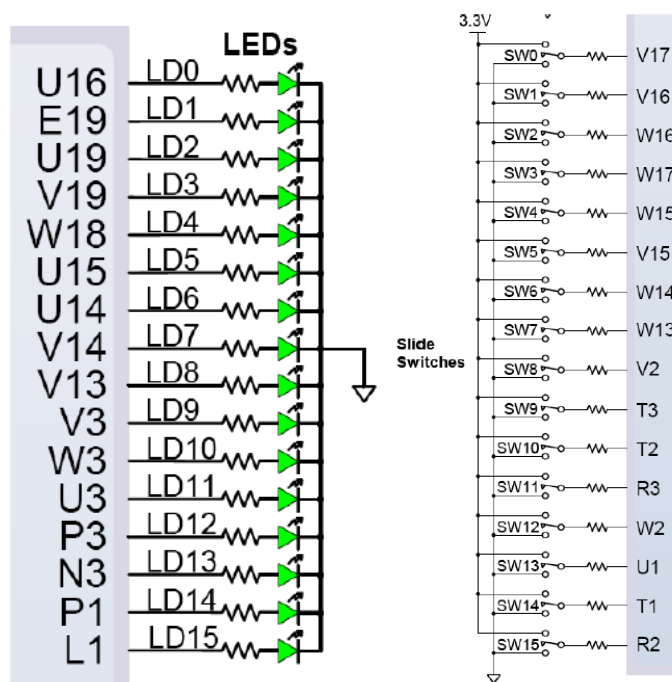


图 1-3LED 灯电路图 1-4 拨码开关电路

1.2.4 拨码开关电路

拨码开关电路如图 1-4 所示。开关打到下档时，表示所连接的 FPGA 输入为低电平。

1.2.5 按键电路

按键部分的电路如图 1-5 所示。当按键按下时，对应的 FPGA 输入为高电平。

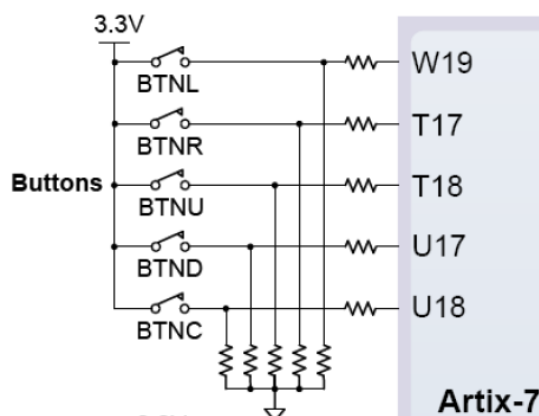


图 1-5 按键电路

1.2.6 数码管电路

数码管电路如图 1-6 所示。4 个带小数点的七段共阳极数码管，当连接的 FPGA 输出为低电平时，该段位的 LED 点亮。位选通为低电平选通。

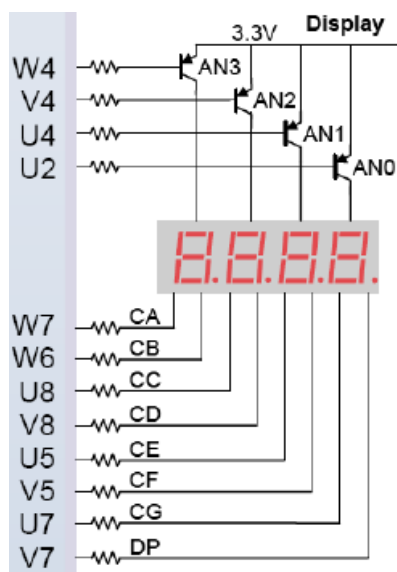


图 1-6 数码管电路

1.2.7 上电加载方式

上电后, Basys3 板上必须配置 FPGA 芯片, 然后才能执行相关功能。在配置过程中, 一个“bit”文件(程序经过编译后的二进制代码)会通过电缆下载到 FPGA 内存单元中, 实现逻辑功能和电路互联。借助赛灵思免费的 Vivado 软件, 可以通过 VHDL 或 Verilog 语言源文件, 或基于原理图输入, 创建.bit 文件。

编程加载模式有三种方式, 通过 JP1 的跳线帽进行选择。

- 用 Vivado 通过 QSPI 方式下载.bit 文件到 Flash 芯片, 实现掉电不易失。板子上电后, FPGA 首先从 Flash 芯片里读.bit 文件。
- 用 Vivado 通过 JTAG 方式从⑬下载.bit 文件到 Flash 芯片。
- 用 U 盘或移动硬盘通过 J2 的 USB 端口⑪下载.bit 文件到 FPGA 芯片(建议将.bit 文件放在 U 盘根目录下, 且只放 1 个), 该 U 盘应该是 FAT32 文件系统。

在实验中, 始终按图 1-7 选一配置, 如下所示。

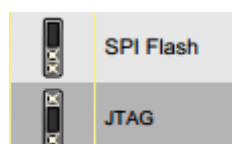


图 1-7 上电加载方式选择

1.2.8 其他

表 1-2 部分引脚对照

LED 灯	PIN	开关	PIN	7 段码管	PIN	按键	PIN
LD0	U16	SW0	V17	AN0	U2	BTNU	T18
LD1	E19	SW1	V16	AN0	U4	BTNR	T18
LD2	U19	SW2	W16	AN0	V4	BTND	U17
LD3	V19	SW3	W17	AN0	W4	BTNL	W19
LD4	W18	SW4	W15	CA	W7	BTNC	U18
LD5	U15	SW5	V15	CB	W6		
LD6	U14	SW6	W14	CC	U8	时钟	PIN
LD7	V14	SW7	W13	CD	V8	MRCC	W5
LD8	V13	SW8	V2	CE	U5		
LD9	V3	SW9	T3	CF	V5	USB(J2)	PIN
LD10	W3	SW10	T2	CG	U7	PS2_CLK	C17
LD11	U3	SW11	R3	DP	V7	PS2_DAT	B17
LD12	P3	SW12	W2				
LD13	N3	SW13	U1				
LD14	P1	SW14	T1				
LD15	L1	SW15	R2				

VGA 显示电路、I/O 扩展电路等，需要时查询 Basys3 的用户手册。

2.熟悉 Vivado 平台

2.1 开发环境介绍

Basys3 板的 FPGA 芯片属于 Xilinx 公司的 ATRIX 7 系列,其对应的开发环境是 Xilinx 推出的 Vivado 设计套件。

2.2 基于模块化的设计流程

采用模块化的设计流程，通过调用已经写好的模块设计数字电路，流程如下：

1、创建新工程

1) 打开 Vivado 设计开发软件，如图 2-1，选择 Create New Project.

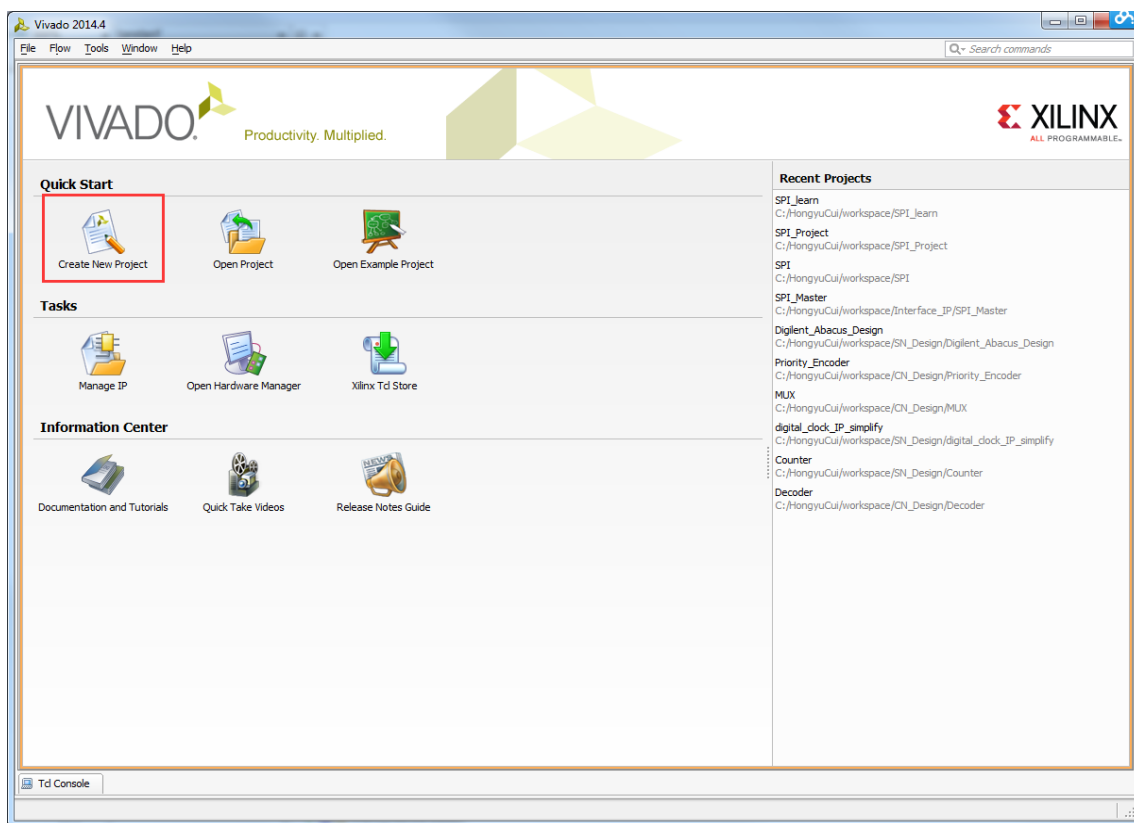


图 2-1 Vivado 初始界面

2) 在弹出的创建新工程的界面中，如图 2-2，点击 Next，开始创建新工程。

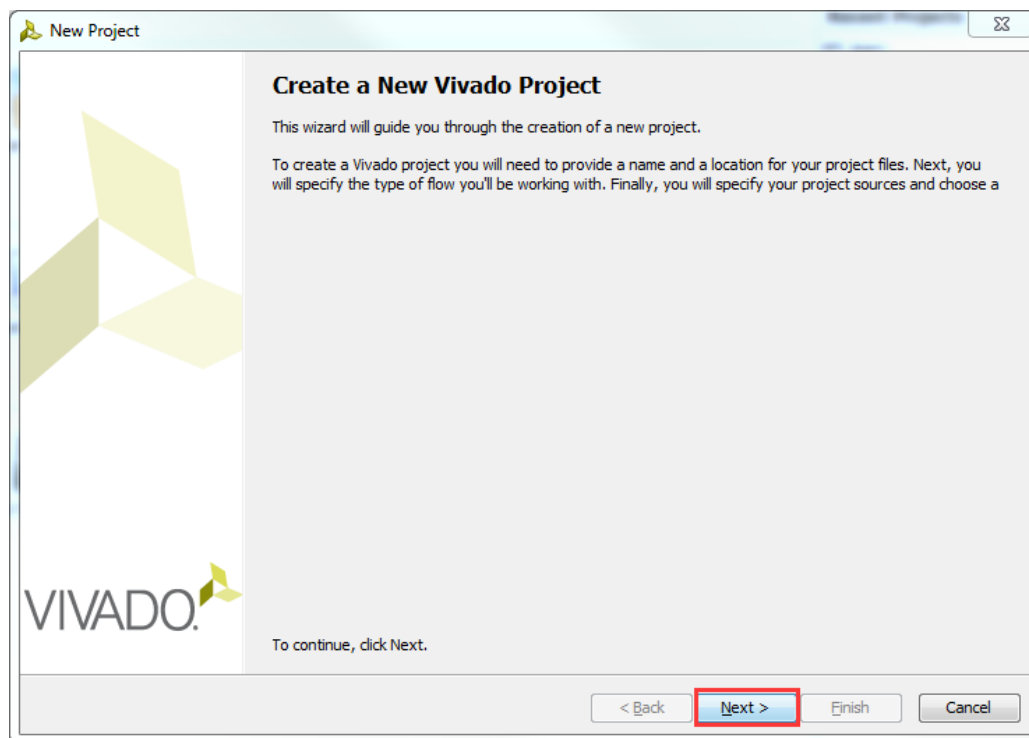


图 2-2 创建新工程

3) 在 Project Name 界面中, 工程名称预置为 project_1 (可根据个人需要进行修改), 设置好工程存放路径为 E:/demo (或其它拟存放工程的路径), 勾选创建工程子目录的选项 (默认项), 如图 2-3。这样, 整个工程文件都将存放在创建的 demo 子目录中。点击 Next。(注意: 路径以及工程名必须是英文!)

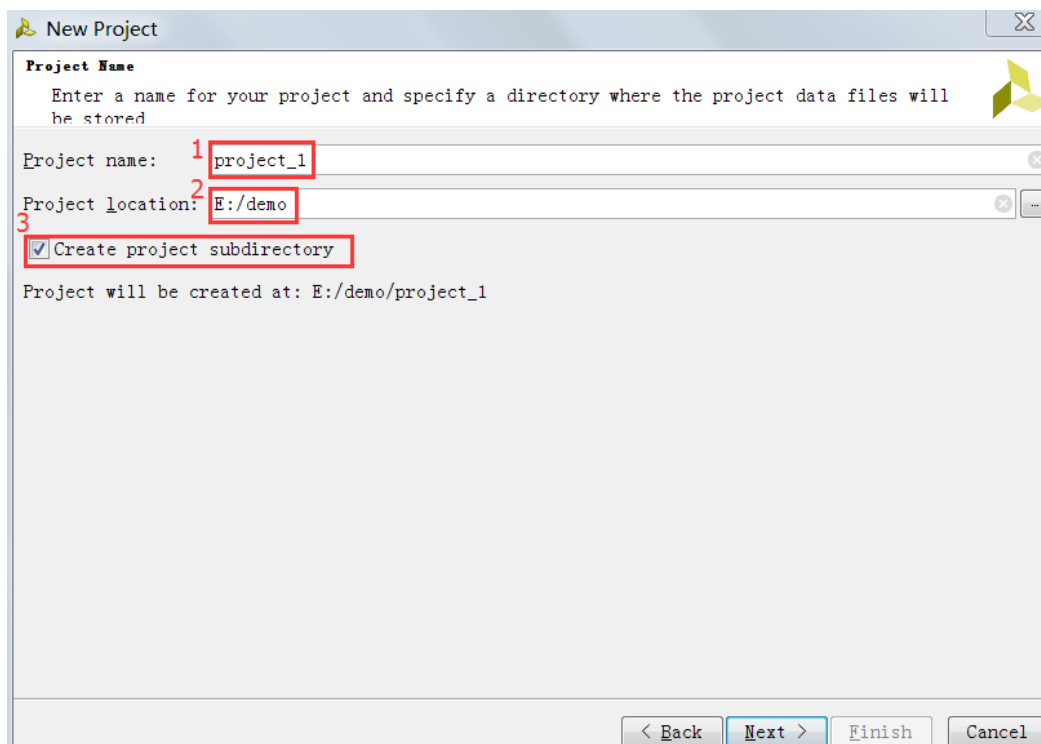


图 2-3 命名及设置路径

4) 在选择工程类型的界面中, 如图 2-4, 选择 RTL 工程。由于本工程无需创建源文件, 故将 Do not specify sources at this time (不指定添加源文件) 勾选上 (默认项)。点击 Next。

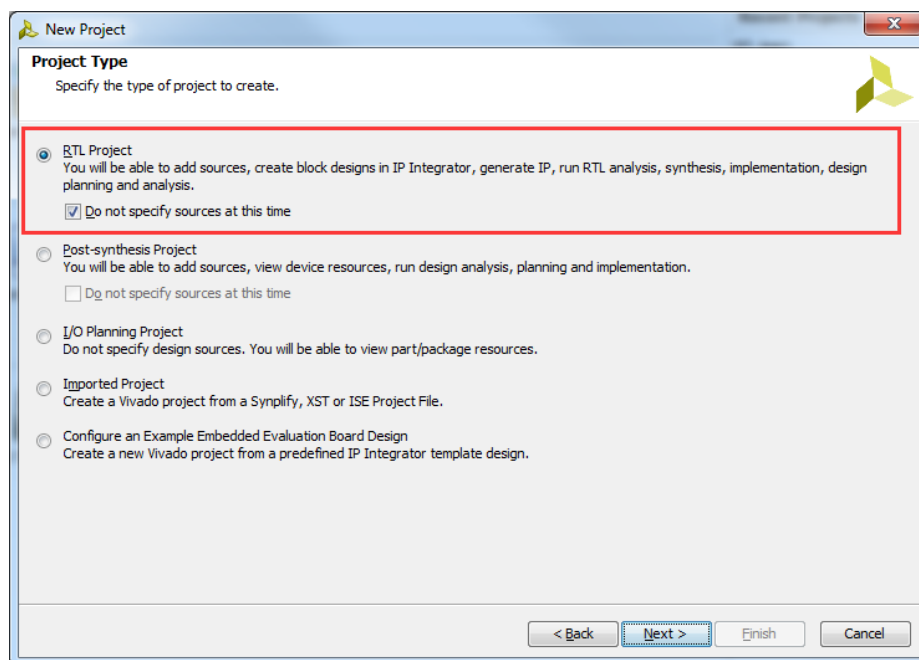


图 2-4 选择 RTL 工程

5) 在器件板卡选型界面中，如图 2-5，在 Search 栏中输入 xc7a35tcpg236 搜索本次实验所使用的 Basys3 板卡上的 FPGA 芯片。并选择 xc7a35tcpg236-1 器件。（器件命名规则详见 xilinx 官方文档）点击 Next。

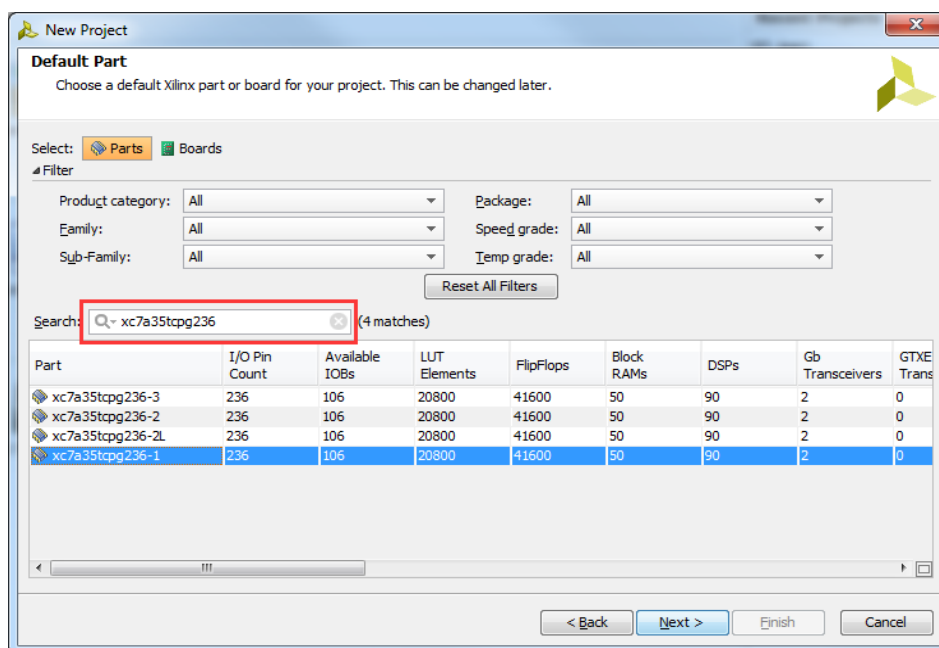


图 2-5 板卡选型

6) 最后在新工程总结中，检查工程创建是否有误。没有问题，则点击 Finish，完成新工程的创建。

2、添加已设计好的 IP 核。

工程建立完毕，我们需要将 project_1 这个工程所需的 IP 目录文件夹复制到本工程文件夹下。本书实验需要的 IP 目录为 Lab_IP 及 74LSXX_LIB(注：其中 Lab_IP 是封装好的单个门电路 IP 核，74LSXX_LIB 是按 74 系列集成电路封装的 IP 核，比较复杂的电路选后者能更简洁一些，可以根据工程需求自行选择)。

1) 在 Vivado 设计界面的左侧设计向导栏中，如图 2-6，点击 Project Manager 目录下的 Project Setting。

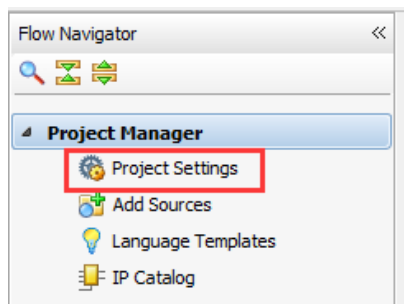


图 2-6 设计向导栏选项

2) 在 Project Setting 界面中，如图 2-7，选择 IP 选项，进入 IP 设置界面。点击 Add Repository..., 添加本工程文件夹下的 IP_Catalog 目录：

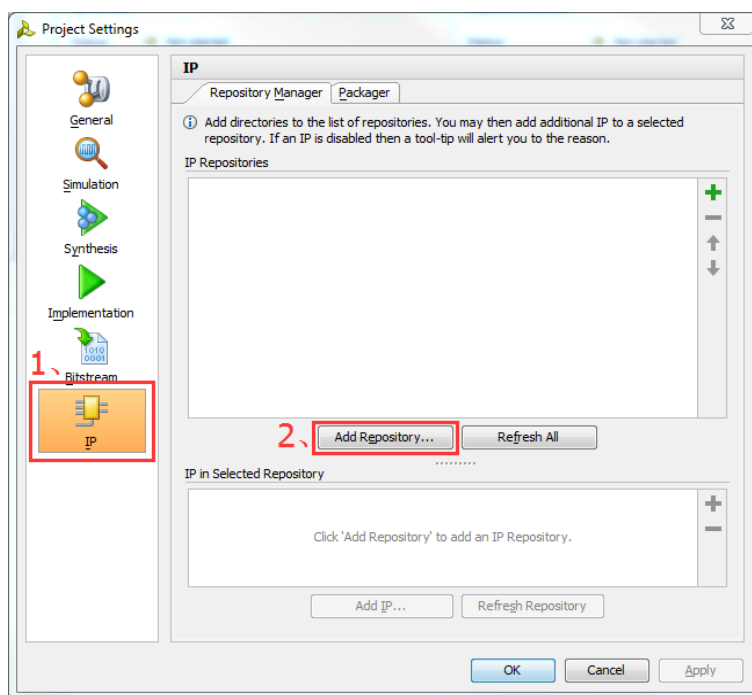


图 2-7IP 设置

3) 完成目录添加后，可以看到所需 IP 核已经自动添加。点击 OK 完成 IP 添加。如图 2-8。

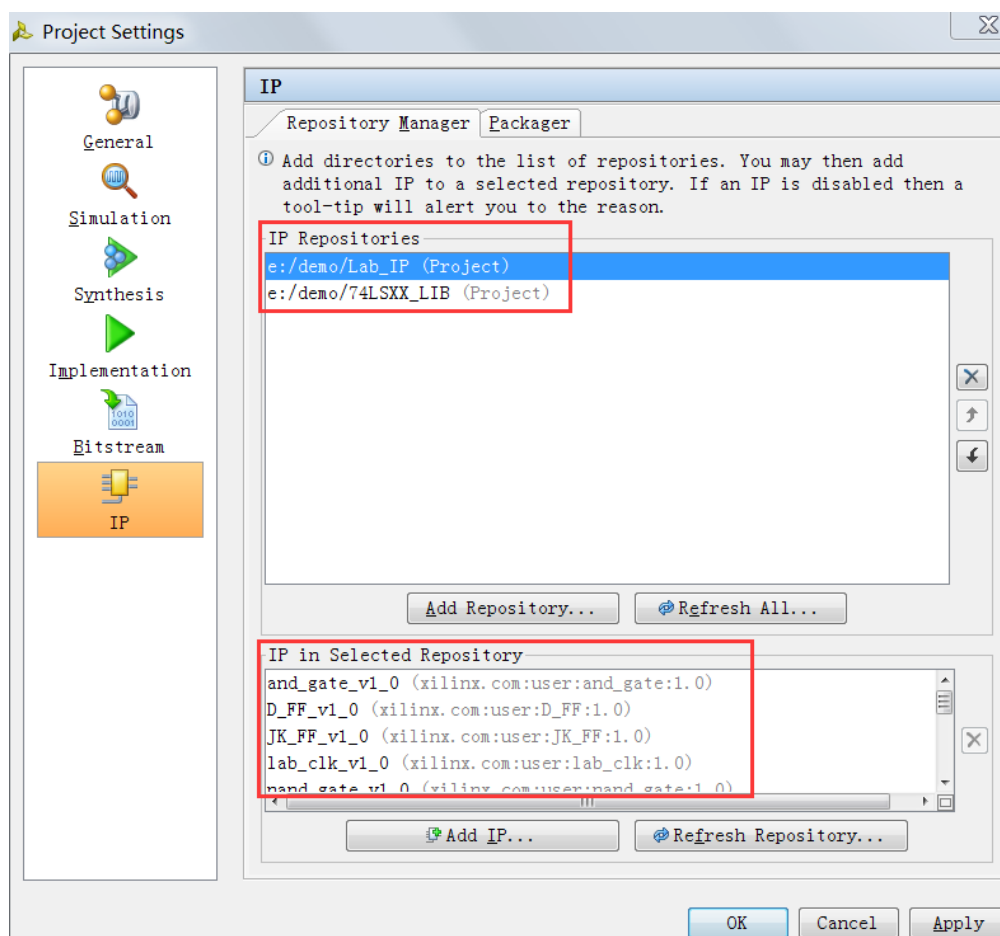


图 2-8 完成 IP 添加

3、创建原理图，添加 IP，进行原理图设计。

1) 在 Project Navigator 下的 IP Integrator 目录下，点击 Create Block Design，创建原理图，如图 2-9。

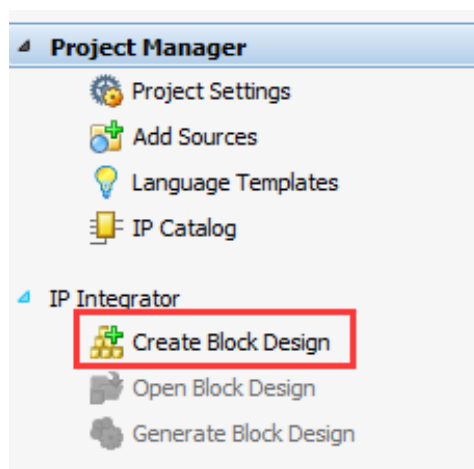


图 2-9 创建原理图

2) 在弹出的创建原理图界面中，如图 2-10，保持默认。点击 OK 完成创建。

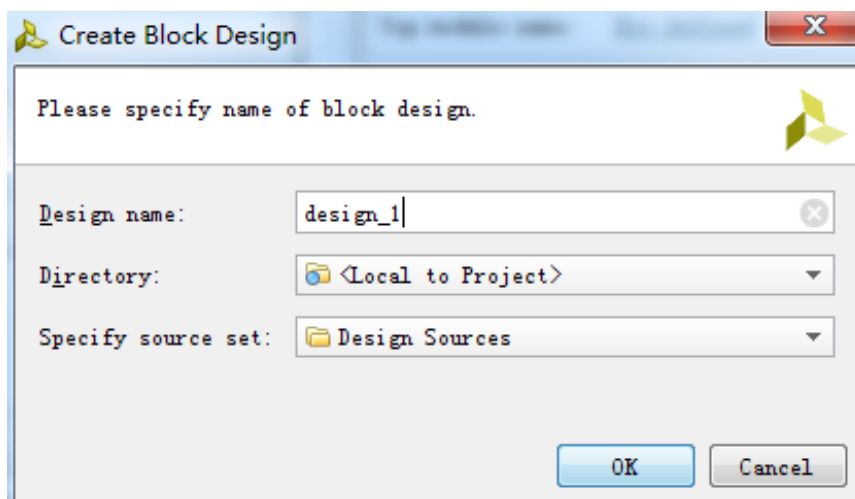



图 2-10 原理图文件名路径界面

3) 在原理图设计界面中，如图 2-11，添加 IP 的方式有 3 种：

- 在设计刚开始时，原理图界面的最上方有相关提示，可以点击 Add IP，进行添加 IP。
- 在原理图设计界面的左侧，有相应快捷键 。
- 在原理图界面中，鼠标右击选择 Add IP。

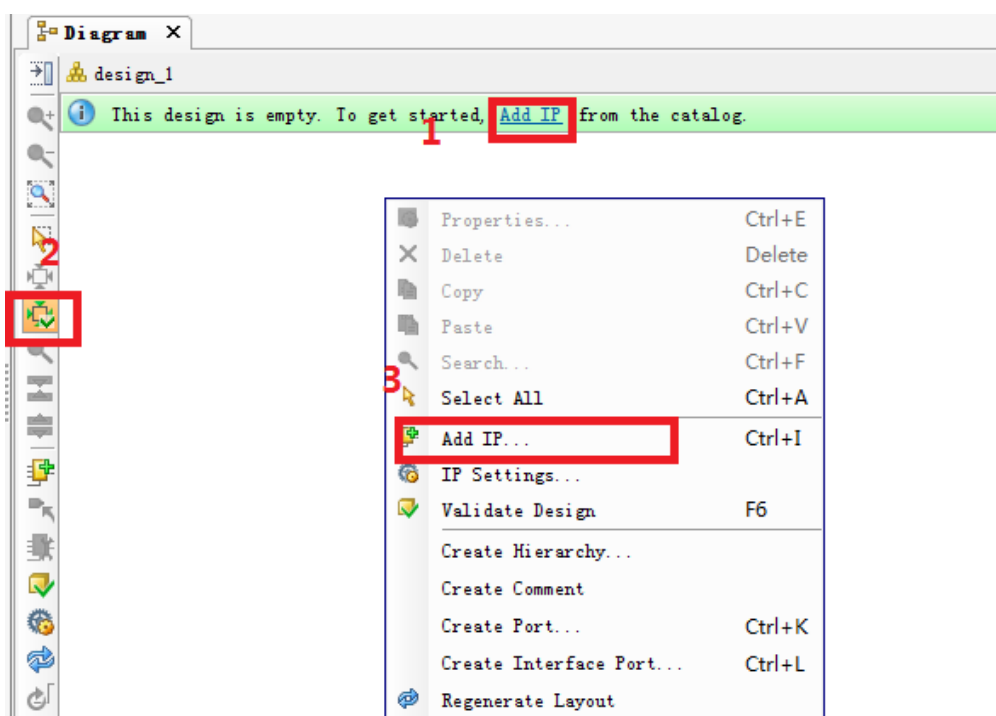


图 2-11 原理图添加 IP

4) 在 IP 选择框中，输入 gate，搜索本实验所需要的 IP。

5) 按 Enter 键，或者鼠标双击该 IP，可以完成添加。如添加 1 个 nand_gate 如图 2-12。

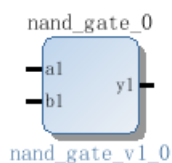


图 2-12 模块图及引脚命名

注意：Vivado 要求模块的输入引脚（左侧）必须都有一个端口，而输出引脚则不必。

6) 添加完 IP 后，进行端口设置和连线操作。连线时，将鼠标移至 IP 引脚附近，鼠标图案变成铅笔状。此时，点击鼠标左键进行拖拽。Vivado 会提醒用户可以与该引脚相连的引脚或端口。

7) 创建与外界相连的端口有两种方式：

- 点击选中 IP 的某一引脚，点击右键在菜单中选择 Make External...自动创建与 IP 引脚同名、同方向的端口；
- 点击选中 IP 的某一引脚，在右键菜单中选择 Create Port，然后在弹出的窗口中设置端口名称，方向以及类型；

8) 通过点击端口，可以在原理图窗口的左侧 external port properties 窗口中修改端口名称。如将上一步中自动创建的端口名 a1 改为 a1_in，b1 改为 b1_in。

9) 完成原理图设计后，生成顶层文件。在 Source 界面中右击 design_1，选择 Generate Output Products，如图 2-13。

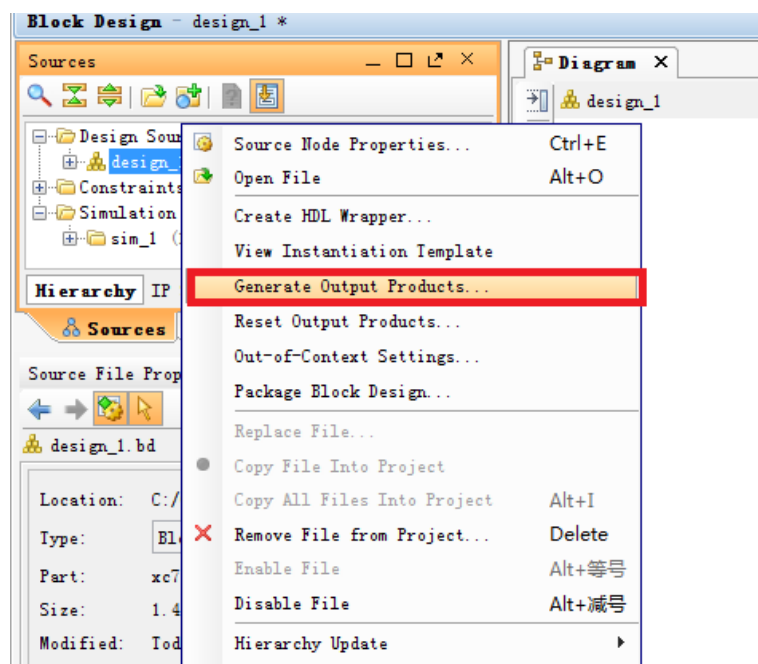


图 2-13 生成顶层文件

10) 在生成输出文件的界面中点击 **Generate**，如图 2-14。生成完输出文件后，再次右击 **design_1**，选择 **Create HDL Wrapper**，创建 HDL 代码文件。对原理图文件进行实例化。在创建 HDL 文件的界面中，保持默认选项，点击 **OK**，完成 HDL 文件的创建。如图 2-15。至此，原理图设计已经完成。

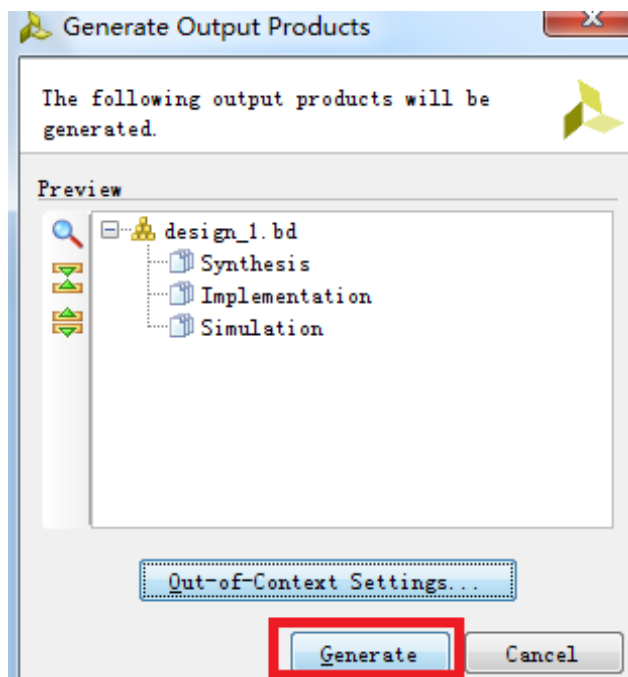


图 2-14 生成顶层文件界面

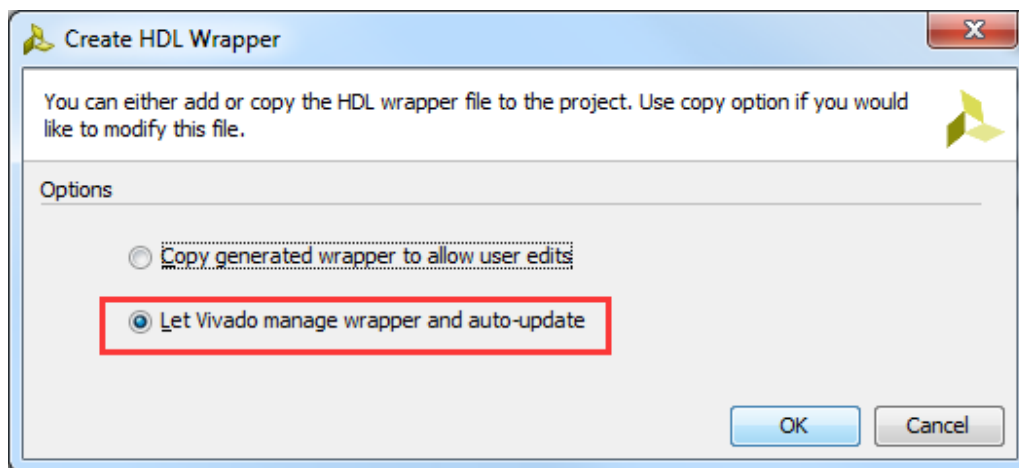


图 2-15 创建 HDL 文件界面

4. 添加管脚约束、综合、实现、生成 bit 流并下载到实验板

- 1) 在导航栏的 RTL Analysis 里选择打开 Elaborated Design。
- 2) 在 Vivado 标签栏中，将界面调整到 I/O PLANNING，如图 2-16。



图 2-16 Vivado 标签栏

3) 在屏幕下方 I/O ports 窗口中将设计端口与 FPGA 引脚进行关联, 如图 2-17。

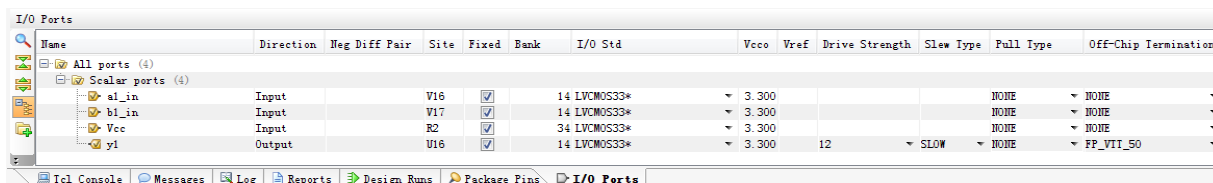


图 2-17 引脚约束

根据 1.2 节的介绍, 实验板上 FPGA 的 V16、V17 引脚分别与拨码开关 SW1 和 SW2 相连, 拨码开关朝上拨为输入高电平。U16 引脚与 LED 灯 LD0 连接, 当有高电平输出时, 灯被点亮 (引脚与拨码开关和 LED 灯的对应关系见表 1-2)。

因此, 在 Site 栏, 三个引脚 a1_in、b1_in、y1 分别设置为 V16、V17、U16; Fixd 栏会自动勾选; I/O std 栏均设为 LVCMOS33 即可。

完成后保存, 将提示为存储的文件取名, 在此用同样英文命名, 如图 2-18。

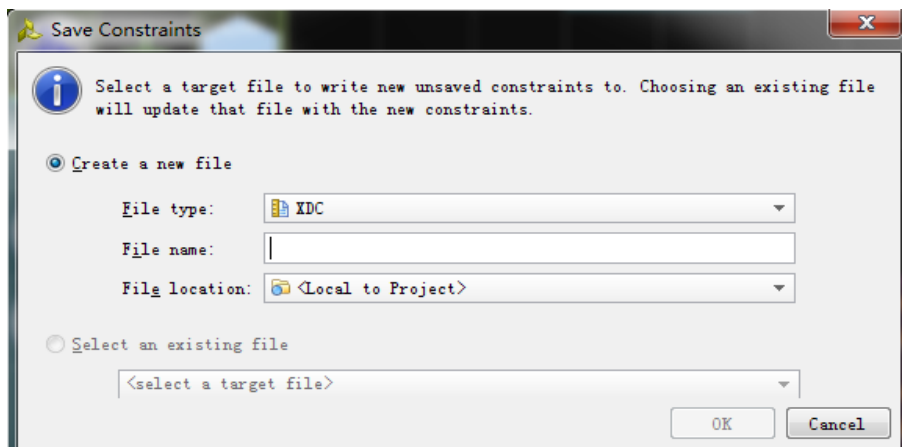


图 2-18 保存约束文件

4) 综合验证。

在导航栏的 Synthesis 里选择打开 Run synthesis。

5) 完成综合验证后, 选择 Run Implementation, 进行工程实现, 如图 2-19。

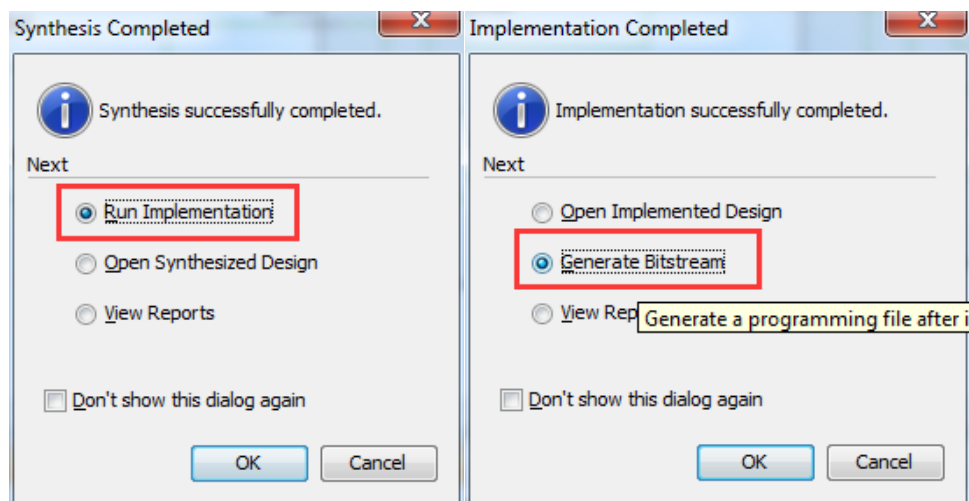


图 2-19 进行工程实现图 2-20 生成编译文件

6) 工程实现完成后, 选择 **Generate Bitstream**, 生成编译文件, 如图 2-20。

7) 生成编译文件后, 选择 **Open Hardware Manager**, 打开硬件管理器, 进行板级验证, 如图 2-21。

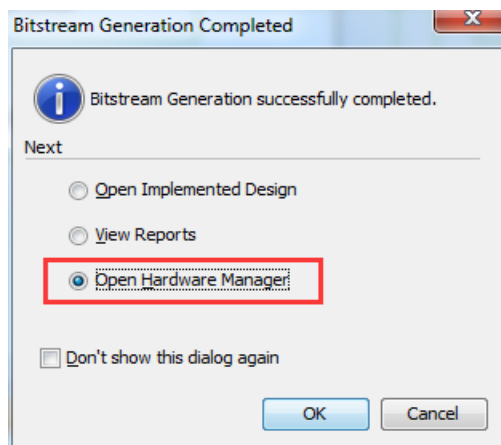


图 2-21 打开 Hardware Manager

8) 打开目标器件, 点击 **Open target**。如果初次连接板卡, 选择 **Open a New Hardware**。如果之前连接过板卡, 可以选择 **Open Recent Target**, 在其列表中选择相应板卡。

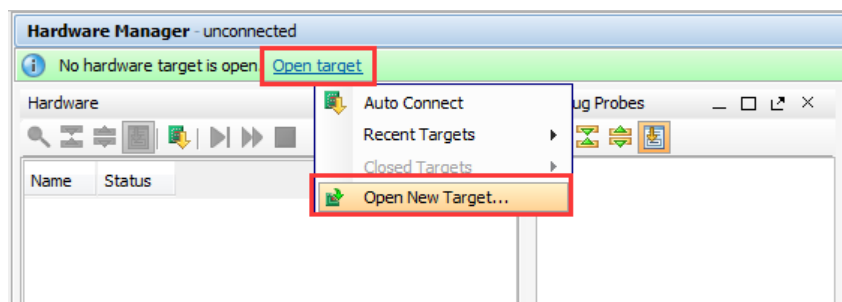


图 2-22 打开目标器件

在打开新硬件目标界面中，点击 Next 进行创建。选择 Local server，点击 Next。

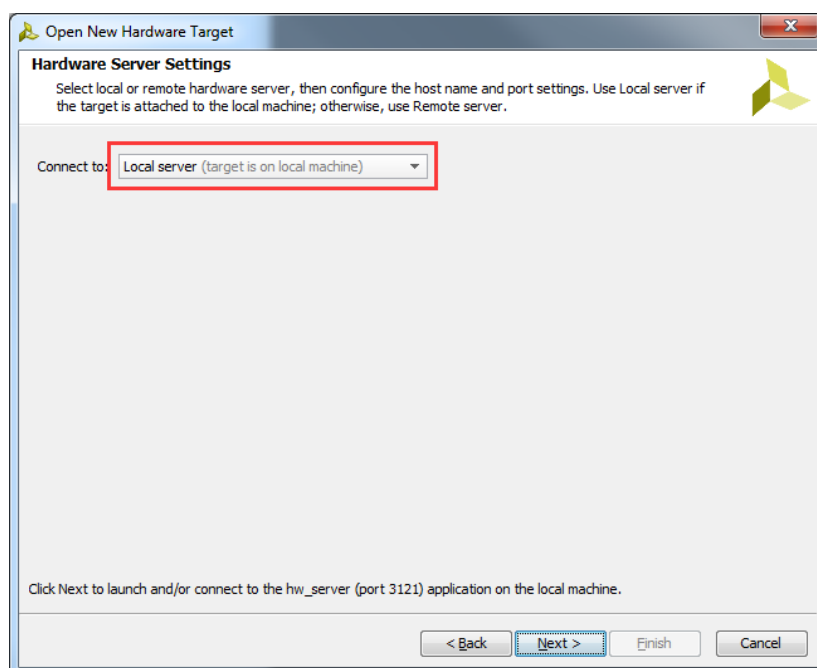


图 2-23 打开新硬件目标界面

点击 Next，再点击 Finish，完成创建。

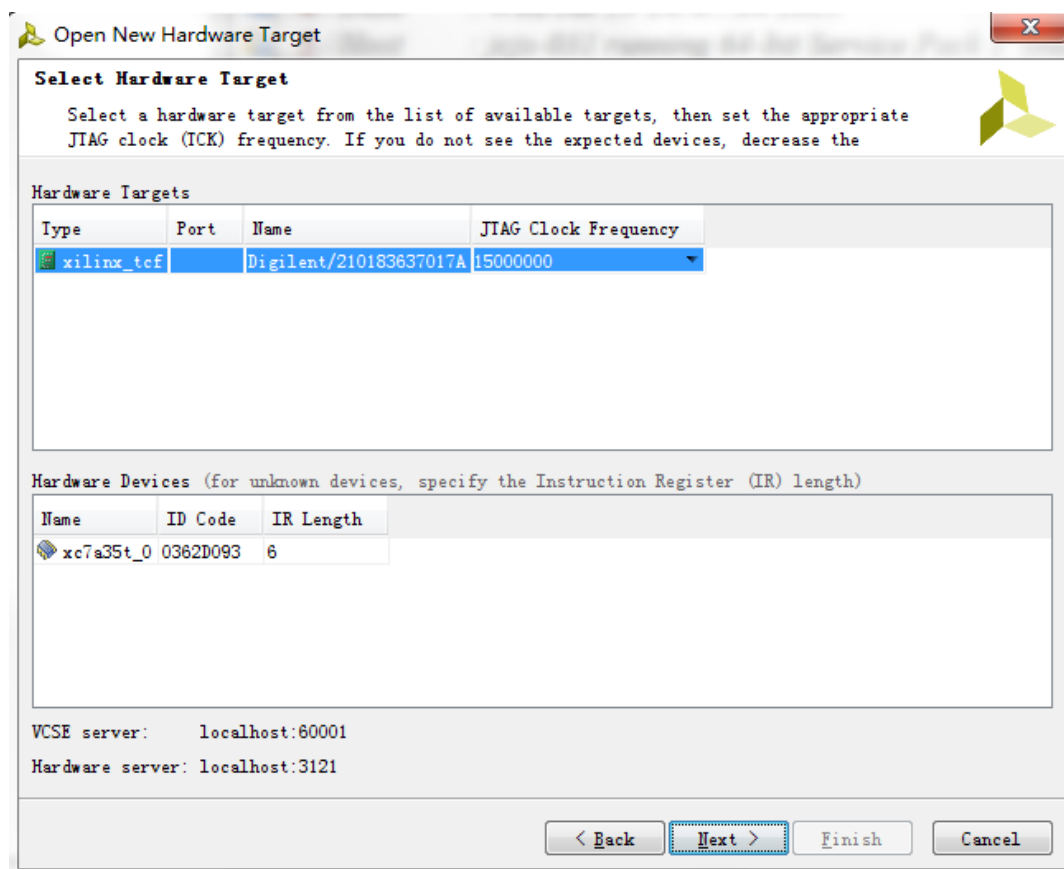


图 2-24 完成创建

9) 下载 bit 文件。

点击 Hardware Manager 上方提示语句中的 Program device。选择目标器件。检查弹出框中所选中的 bit 文件，然后点击 Program 进行下载。进行板级验证。

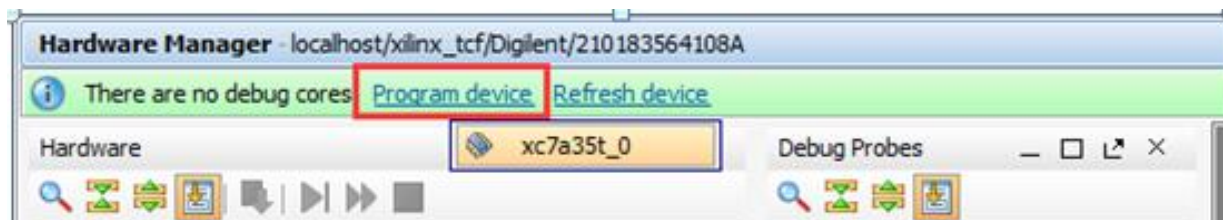


图 2-25 下载 bit 文件

拨动最右端的两个拨码开关 SW1 和 SW0，观察 LED 灯 LD0 的明亮情况，查看开关与灯组成的逻辑是否实现了一个与非门的功能。

第二部分数字逻辑实验

3.实现布尔表达式

3.1 实验要求

目的：通过实验，使学生学会根据布尔表达式实现与其对应的数字电路。

内容：使用第一部分介绍的基本门电路 IP 核，利用 Vivado 设计一个组合逻辑电路，实现布尔表达式 $Y = AB + AC + B'$ 的逻辑功能。

报告：按要求填写实验报告。

扩展：使用附录 2 中 Verilog 语言的结构化描述进行设计及实现。

3.2 实验步骤

3.2.1 设计示例

1. 分析表达式， A 、 B 、 C 作为电路的输入信号。 Y 是电路的输出信号。根据表达式画卡诺图，观察是否可以化简，得到化简后的方程进行电路设计。
2. 化简后 $Y = A + B'$ 。
3. 根据方程可知，仅需要 IP 核中的 or_gate（或门）和 not_gate（非门）。
4. 创建新的工程(注：不要使用中文路径，中文名字！)，加载 IP 核。
5. 创建原理图，进行原理设计。
 - a) 在 Project Navigator 下的 IP Integrator 目录下，点击 Create Block Design，创建原理图。
 - b) 保持默认，点击 OK 完成创建。
 - c) 在 Diagram 中通过添加 IP 来添加逻辑门电路。



图 3-2 创建原理图

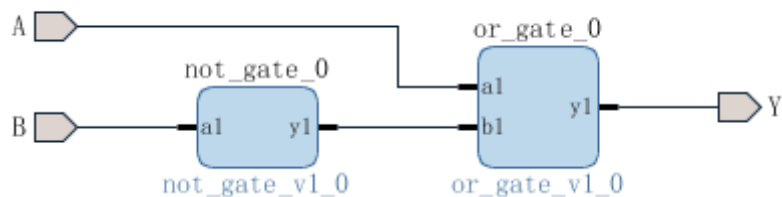


图 3-3 表达式的 BD 设计图示例

6. 右击 Block_Designs 中的 design_1，先点击“Generate Output Products”，然后点击“Create HDL Wrapper”。

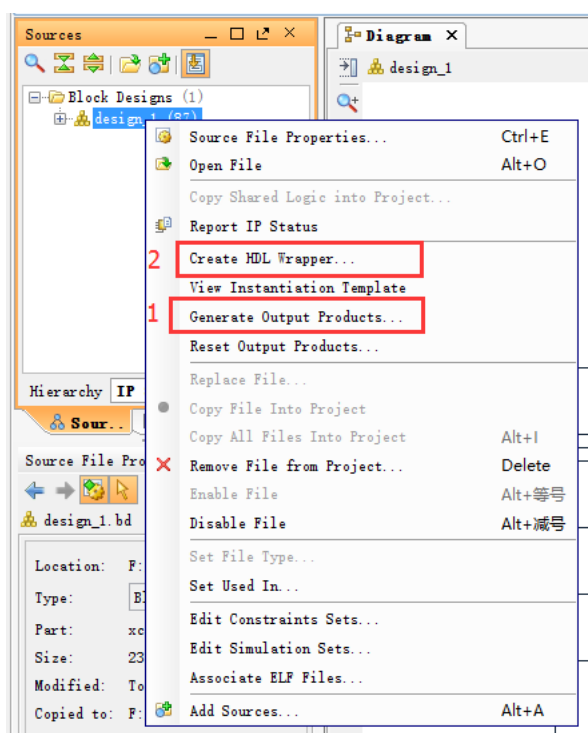


图 3-4 生成顶层文件

7. 打开 Elaborated Design，配置管脚约束（I/O PLANNING），为输入指定相应的拨码开关，管脚可以查看 1.2.6。
8. 综合、实现、生成 bitstream。
9. 完成板级验证。

3.2.2 设计电路实现布尔表达式

根据表达式 $Y = (A' + C')(B + C')(B + D)$ 进行化简。根据化简的结果，仅使用与非门，设计一个组合逻辑电路实现 4 变量的布尔表达式。

要求：在实验报告中，参考表 4-1 列出 4 位二进制数不同取值，填写对应的 Y 值、表达式的化简结果，以及表达式对应的电路图。

表 3-1 记录表

A (SW)	B (SW)	C (SW)	D (SW)	Y (LED)
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
...	
1	1	1	1	
化简结果				

4. 加法器

4.1 实验要求

目的：通过实验，使学生学会设计加法器。

内容：利用 Vivado 设计一个 2 位并行加法器，并利用半加器和全加器实现一个 2 位并行加法器。

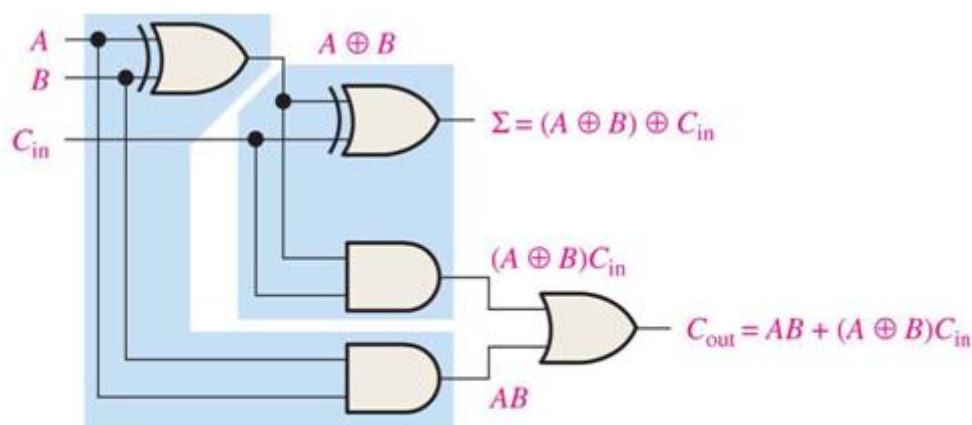
报告：按要求填写实验报告。

扩展：使用附录 2 中 Verilog 语言的结构化描述进行设计及实现。

4.2 实验步骤

4.2.1 全加器的设计

根据图 4-1（教材 P198）介绍的原理用 Block Design 设计一个全加器：



(b) Complete logic circuit for a full-adder (each half-adder is enclosed by a shaded area)

图 4-1 全加器原理图

- 1) 分析输入（加数 A、被加数 B 及低位进位 C_{in} ）、输出（和 S 及进位位 C_{out} ）；根据方程和 IP 核库判断需要使用的门电路以及个数（异或门、与门、或门）。
- 2) 创建新的工程(注：不要使用中文路径，中文名字！)，加载 IP 核。
- 3) 在 Project Navigator 下的 IP Integrator 下，点击 Create Block Design，创建新的 Block Design，根据上面的分析添加基本门电路、输入输出端口并连线（注意未使用的输入管脚不能为空），将该设计保存时命名为 fulladder(注意名字！)。

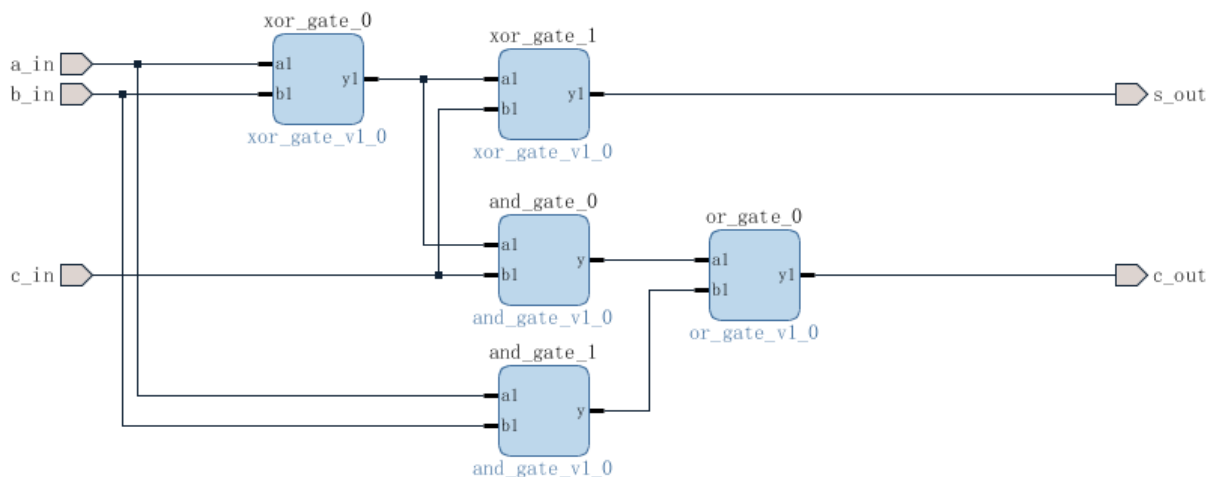


图 4-2 全加器 BD 设计图

- 4) 完成原理图设计后，生成顶层文件（Generate Output Products）和 HDL 代码文件（Create HDL Wrapper）。
- 5) 配置管脚约束（I/O PLANNING），为 3 个输入指定相应的拨码开关，为 2 个输出指定相应的 led 灯显示。

I/O Ports									
Name	Direction	Neg Diff Pair	Site	Fixed	Bank	I/O Std	Vcco	Vre	
All ports (5)									
Scalar ports (5)									
a_in	Input		R2	<input checked="" type="checkbox"/>	34	LVC MOS33*	3.300		
b_in	Input		T1	<input checked="" type="checkbox"/>	34	LVC MOS33*	3.300		
c_in	Input		U1	<input checked="" type="checkbox"/>	34	LVC MOS33*	3.300		
c_out	Output		L1	<input checked="" type="checkbox"/>	35	LVC MOS33*	3.300		
s_out	Output		P1	<input checked="" type="checkbox"/>	35	LVC MOS33*	3.300		

图 4-3 管脚约束示例

- 6) 综合、实现、生成 bitstream。
- 7) 板级验证。

4.2.2 利用逻辑门构成半加器和全加器，并设计一个 2 位并行加法器

参照下图（教材 P202 图 6.9）用 Block Design 设计一个 2 位并行加法器：使用逻辑门实现一个半加器级联一个全加器构成两位并行加法器，完成设计，将设计下载到实验板进行验证。

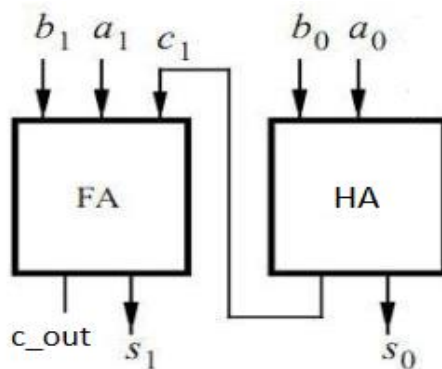


图 4-4 2 位并行加法器原理图

- 1) 创建新的工程，加入 lab_IP 核，按上图原理设计一个 2 位二进制加法器，添加管脚约束、综合、实现、生成 bitstream 进行板级验证。

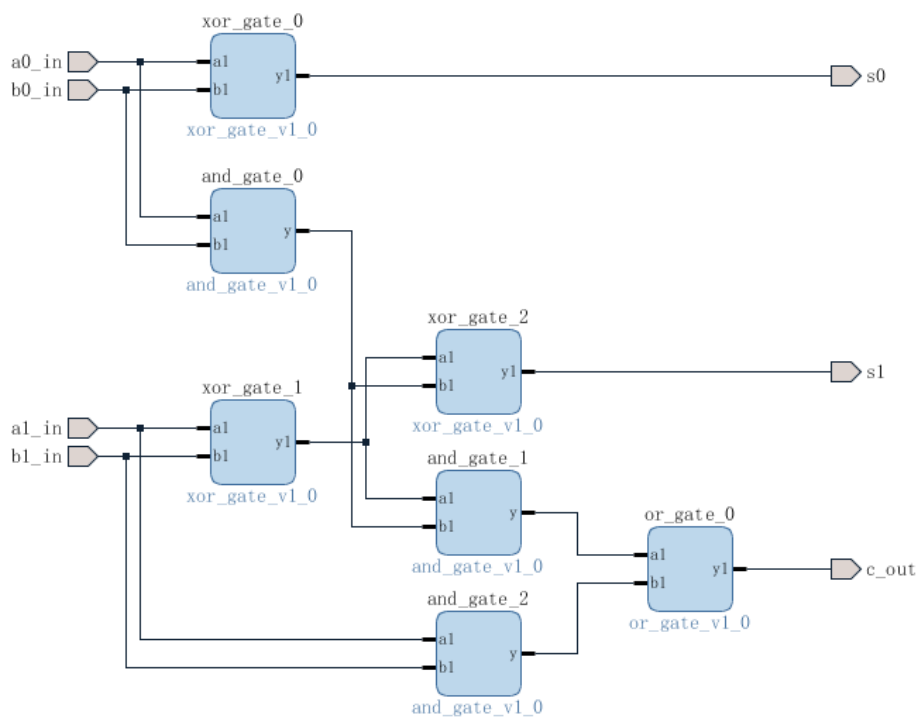


图 4-5 2 位并行加法器的 BD 设计图

要求: 在实验报告中参考表 3-1 列出两个 2 位二进制数 3 种不同取值对应的 LED 灯显示。

表 4-1 记录表

A1	A0	B1	B0	LED 灯显示
0	1	1	0	
1	0	0	1	
0	1	1	1	

5. 译码器和编码器

5.1 实验要求

目的：通过实验，使学生学会设计译码器和优先编码器。

内容：使用基本门电路 IP 核设计一个 3-8 译码器、一个 4-2 优先编码器。

报告：按要求填写实验报告。

扩展：1. 封装 IP 核

2. 使用附录 2 中 Verilog 语言的结构化描述进行设计及实现。

5.2 实验步骤

5.2.1 设计译码器

图 5-1 是 3-8 译码器（74LS138）的电路图。我们将依据此图来进行设计。

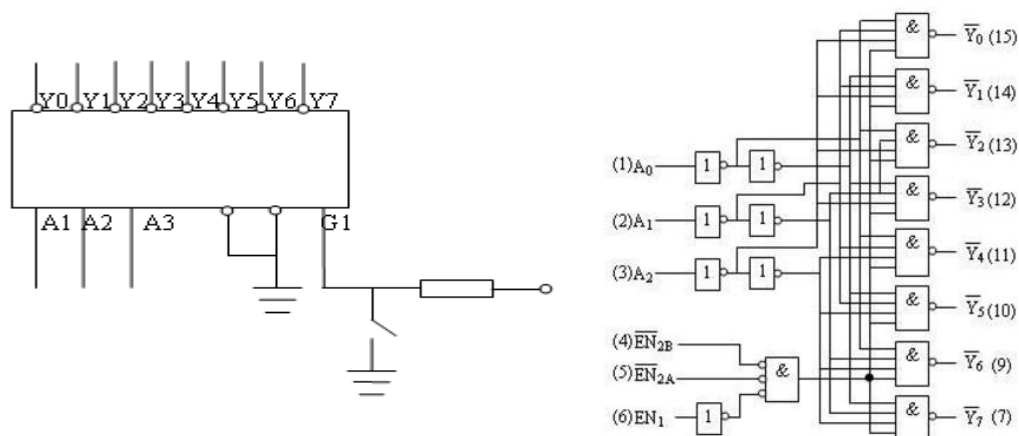


图 5-1 电路及原理图

- 1) 分析输入、输出；根据方程和 IP 核库判断需要使用的门电路以及个数。
- 2) 创建新的工程，加载需要使用的 IP 核。
- 3) 创建 BD 设计文件，添加你所需要的 IP 核，根据图 5-1 进行端口设置和连线操作。
- 4) 完成原理图设计后，生成顶层文件（Generate Output Products）和 HDL 代码文件（Create HDL Wrapper）。
- 5) 配置管脚约束（I/O PLANNING），为输入指定相应的拨码开关，为输出指定相应的 led 灯显示。

6) 综合、实现、生成 bitstream。

7) 仿真验证，依据 3-8 译码器输入输出真值表，在实验板验证试验结果。

输入						输出							
G1	G2	G3	A2	A1	A0	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
x	1	x	x	x	x	1	1	1	1	1	1	1	1
x	x	1	x	x	x	1	1	1	1	1	1	1	1
0	x	x	x	x	x	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	1	1	1	0
1	0	0	0	0	1	1	1	1	1	1	1	0	1
1	0	0	0	1	0	1	1	1	1	1	0	1	1
1	0	0	0	1	1	1	1	1	1	0	1	1	1
1	0	0	1	0	0	1	1	1	0	1	1	1	1
1	0	0	1	0	1	1	1	0	1	1	1	1	1
1	0	0	1	1	0	1	0	1	1	1	1	1	1
1	0	0	1	1	1	0	1	1	1	1	1	1	1

5.2.2 设计 4-2 优先编码器

图 5-2 是一个 4-2 优先编码器的真值表。我们将依据此图来进行设计和验证。

Inputs				Outputs		
D ₃	D ₂	D ₁	D ₀	A ₁	A ₀	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

- 1) 分析输入、输出，列出方程。根据方程和 IP 核库判断需要使用的门电路以及个数。
- 2) 创建新的工程，加载需要使用的 IP 核。
- 3) 创建 BD 设计文件，添加你所需要的 IP 核，进行端口设置和连线操作。
- 4) 完成原理图设计后，生成顶层文件（Generate Output Products）和 HDL 代码文件（Create HDL Wrapper）。
- 5) 配置管脚约束（I/O PLANNING），为输入指定相应的拨码开关，为输出指定相应的 led 灯显示。
- 6) 综合、实现、生成 bitstream。
- 7) 仿真验证，依据真值表，在实验板验证试验结果。

6. 多路选择器

6.1 实验要求

目的：通过实验，使学生学会设计多路选择器。

内容：利用 Vivado 设计一个 2 选 1 多路选择器和一个 4 选 1 多路选择器。

报告：按要求填写实验报告。

扩展：使用附录 2 中 Verilog 语言的结构化描述进行设计及实现。

6.2 实验步骤

6.2.1 设计 4 选 1 多路选择器并封装为 IP 核

图 6-1 是一位 4 选 1 多路选择器的电路图，我们将依据此图来进行设计。

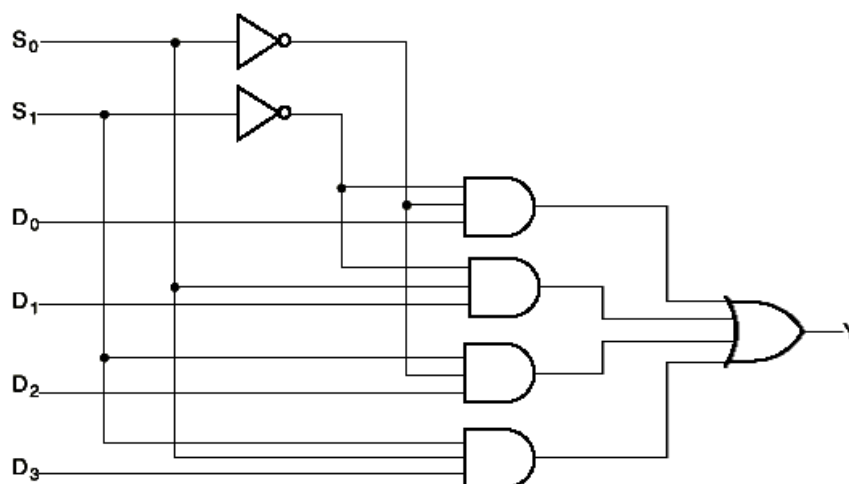


图 6-1 4 选 1 多路选择器电路图

- 1) 分析输入、输出；根据方程和 IP 核库判断需要使用的门电路以及个数。
- 2) 创建新的工程，加载需要使用的 IP 核。
- 3) 创建 BD 设计文件，添加你所需要的 IP 核，进行端口设置和连线操作。
- 4) 完成原理图设计后，生成顶层文件（Generate Output Products）和 HDL 代码文件（Create HDL Wrapper）。
- 5) 配置管脚约束（I/O PLANNING），为输入指定相应的拨码开关，为输出指定相应的 led 灯显示。
- 6) 综合、实现、生成 bitstream。
- 7) 仿真验证

4-1 多路器的逻辑表达式为 $Y = S_1'S_0'D_0 + S_1'S_0D_1 + S_1S_0'D_2 + S_1S_0D_3$ ，对应功能如下表，请仿真验证所设计的多路器电路是否正确。

表 6-1 4-1 多路器功能表

S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

6.2.2 利用自己设计的 4 选 1 多路选择器实现逻辑函数

请利用自己设计的多路器实现以下函数并上板验证：

1) $y = ab + a'b + ab'$

2) $y = abc + ab' + a'bc$

要求：实验报告给出详细设计过程。

7. 寄存器

7.1 实验要求

目的：通过实验，使学生学会设计寄存器。

内容：设计一个 4 位并行输入并行输出寄存器和一个 4 位循环移位寄存器（IP 核 D_FF（带低有效置位复位的下降沿 D 触发器）和 lab_clk（时钟分频））。

报告：按要求填写实验报告。

扩展：使用附录 2 中 Verilog 语言的结构化描述进行设计及实现。

7.2 实验步骤

7.2.1 4 位并行输入并行输出寄存器

图 7-1 是 4 位并行输入并行输出寄存器的原理图。我们将依据此图来进行设计。

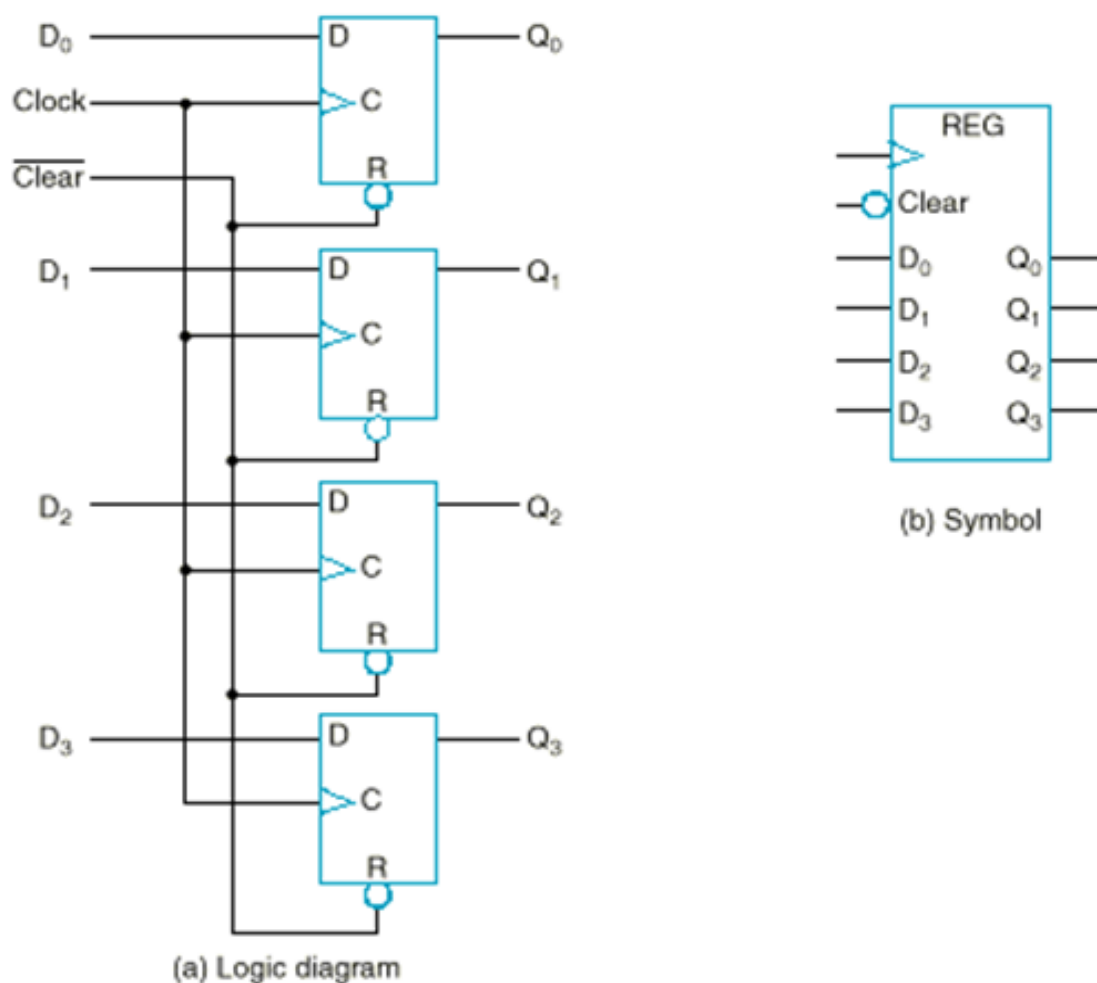


图 7-1 4 位并行输入并行输出寄存器原理图

实验步骤：

- (1) 分析输入、输出；根据方程和 IP 核库判断需要使用的门电路以及个数。
- (2) 按照第二章模块化设计流程新建工程，并且导入正确的 IP 核，准备新建 BlockDesign 文件。
- (3) 参考图 7-2，在导入的 IP 核中选择 D 触发器进行 BlockDesign 设计，并且加入清零和置位功能接口，清零端和置位端都是低有效。

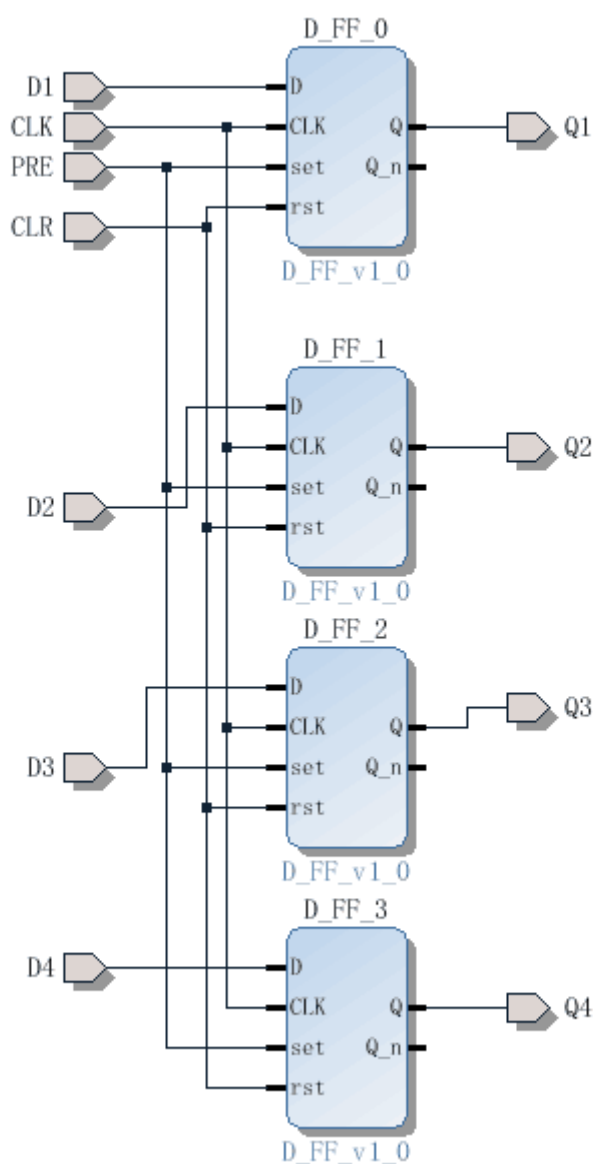


图 7-2 4 位并行输入并行输出的寄存器 BD 设计图

- (4) BD 文件设计好之后，点击 GenerateOutputProducts 和 CreateHDLWrapper，之后进入 RTL 分析，进行引脚分配如图 7-3。

I/O Ports									
Name	Dir...	Ne...	Site	Fixed	Bank	I/O Std	Vcco	Vre	
All ports (11)									
Scalar ports (11)									
CLK	Input		W5	<input checked="" type="checkbox"/>		34 LVC MOS33*	3.300		
CLR	Input		R2	<input checked="" type="checkbox"/>		34 LVC MOS33*	3.300		
D1	Input		V17	<input checked="" type="checkbox"/>		14 LVC MOS33*	3.300		
D2	Input		V16	<input checked="" type="checkbox"/>		14 LVC MOS33*	3.300		
D3	Input		W16	<input checked="" type="checkbox"/>		14 LVC MOS33*	3.300		
D4	Input		W17	<input checked="" type="checkbox"/>		14 LVC MOS33*	3.300		
PRE	Input		T1	<input checked="" type="checkbox"/>		34 LVC MOS33*	3.300		
Q1	Output		U16	<input checked="" type="checkbox"/>		14 LVC MOS33*	3.300		
Q2	Output		E19	<input checked="" type="checkbox"/>		14 LVC MOS33*	3.300		
Q3	Output		U19	<input checked="" type="checkbox"/>		14 LVC MOS33*	3.300		
Q4	Output		V19	<input checked="" type="checkbox"/>		14 LVC MOS33*	3.300		

图 7-3 引脚分配图

- (5) 把寄存器输入，清零端和置位端接到开关引脚上面，把输出接到 LED 灯引脚上面，把 CLK 接到时钟 W5 引脚上。
- (6) 引脚分配完成之后进行综合，分析，生成比特流。
- (7) 最后下载到实验板上面进行验证实验结果：当清零端为低电平时，四个 LED 灯都不亮代表寄存器输出为 0，当置位端为低电平时四个 LED 灯全部亮，代表寄存器输出为 1，当清零端和置位端为高电平时，LED 灯亮暗根据对应的四个开关电平的高低。

8.2.2 4 位循环移位寄存器

图 7-4 是 4 位循环移位寄存器的原理图。我们将依据此图来进行设计。

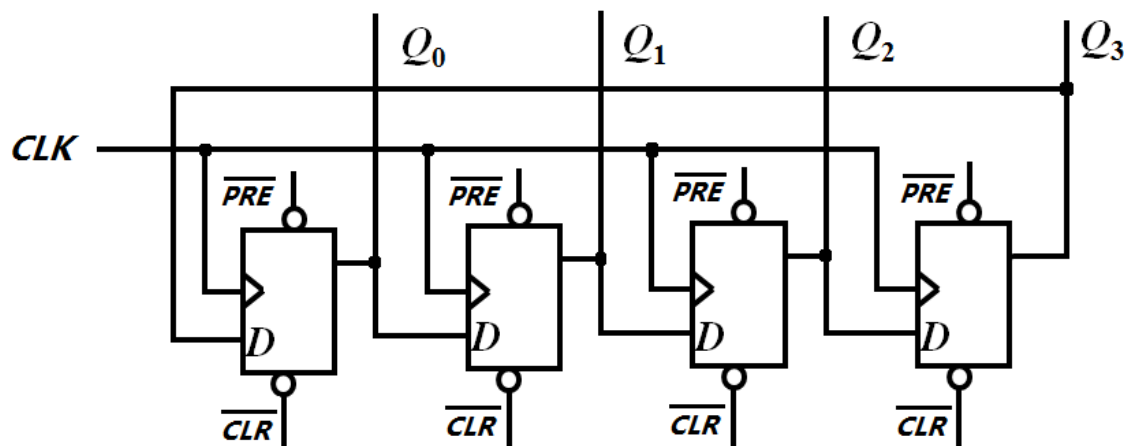


图 7-4 4 位循环移位寄存器原理图

实验步骤：

- (1) 分析输入、输出；根据原理图和 IP 核库判断需要使用的门电路以及个数。

- (2) 按照第二章模块化设计流程，新建工程，并且导入正确的 IP 核，准备新建 BlockDesign 文件。
- (3) 参考图 7-5，在导入的 IP 核中选择 D 触发器进行 BlockDesign 设计，并且加入清零和置位功能接口，清零端和置位端都是低有效，将上一个寄存器的输出接到下一个寄存器的输入端，最后一个寄存器的输出接到第一个寄存器的输入，实现循环移位。

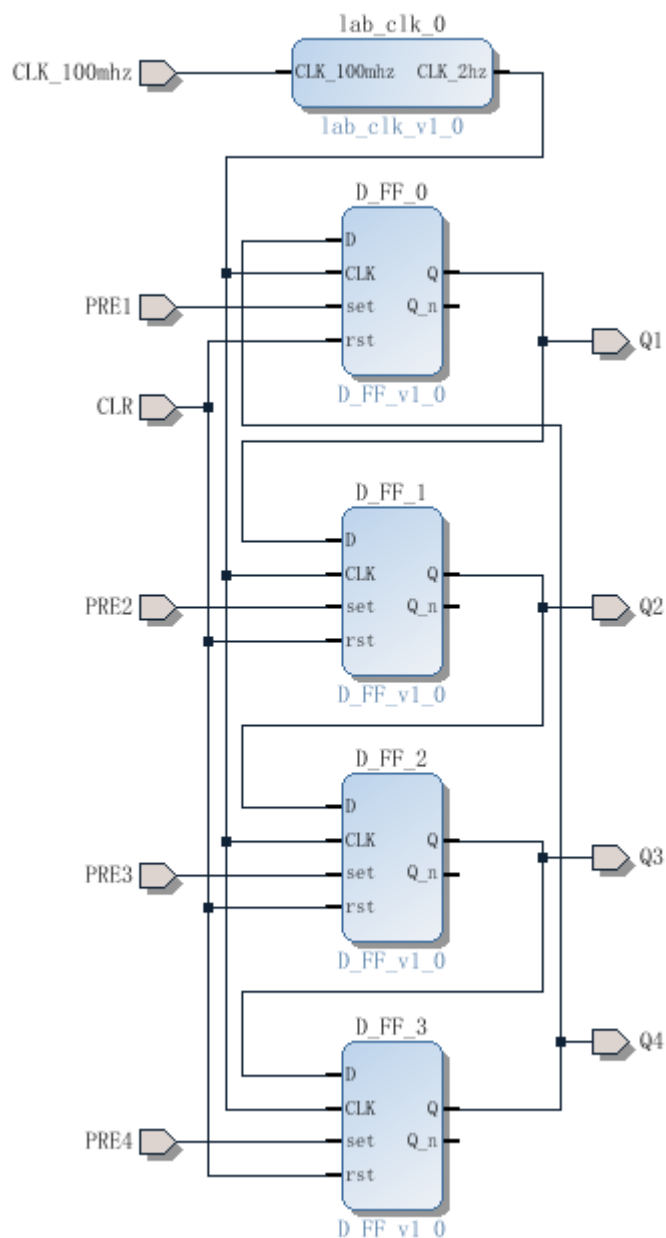
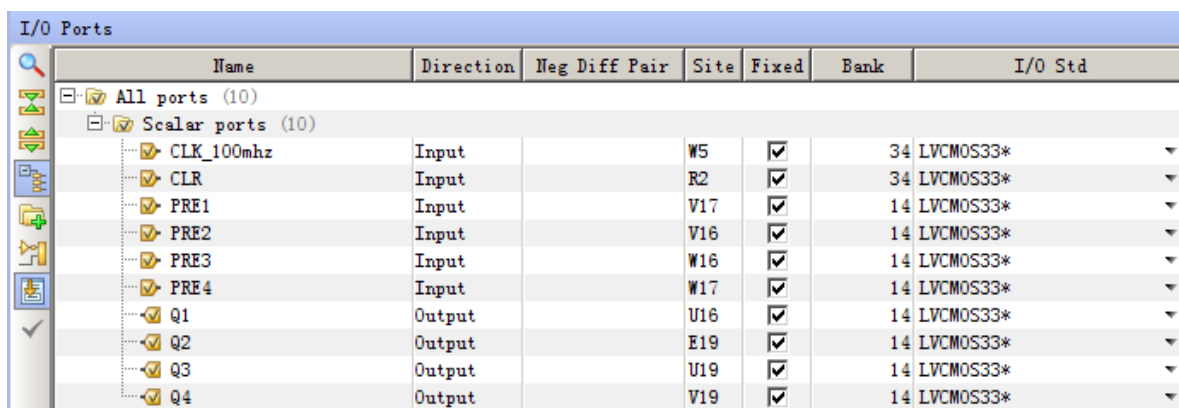


图 7-5 4 位循环移位寄存器 BD 设计图

- (4) BD 文件设计好之后，点击 Generate Output Products 和 Create HDL Wrapper，之后进入 RTL 分析，进行引脚分配如图 7-6。



Name	Direction	Neg Diff Pair	Site	Fixed	Bank	I/O Std
CLK_100mhz	Input		W5	<input checked="" type="checkbox"/>	34	LVC MOS33*
CLR	Input		R2	<input checked="" type="checkbox"/>	34	LVC MOS33*
PRE1	Input		V17	<input checked="" type="checkbox"/>	14	LVC MOS33*
PRE2	Input		V16	<input checked="" type="checkbox"/>	14	LVC MOS33*
PRE3	Input		W16	<input checked="" type="checkbox"/>	14	LVC MOS33*
PRE4	Input		W17	<input checked="" type="checkbox"/>	14	LVC MOS33*
Q1	Output		U16	<input checked="" type="checkbox"/>	14	LVC MOS33*
Q2	Output		E19	<input checked="" type="checkbox"/>	14	LVC MOS33*
Q3	Output		U19	<input checked="" type="checkbox"/>	14	LVC MOS33*
Q4	Output		V19	<input checked="" type="checkbox"/>	14	LVC MOS33*

图 7-6 4 位循环移位寄存器引脚分配图

- (5) 把寄存器输入，清零端和置位端接到开关引脚上面，把输出接到 LED 灯引脚上面，把 CLK 接到时钟 W5 引脚上。
- (6) 引脚分配完成之后进行综合，分析，生成比特流，最后下载到实验板上进行验证实验结果：当清零端为低电平时，四个 LED 灯都不亮代表寄存器输出为 0，当置位端为低电平时四个 LED 灯全部亮，代表寄存器输出为 1，当清零和置位端为高电平时，LED 会根据存储在寄存器中的内容进行循环位移。实际操作时应注意先使用 CLR 开关清零，然后使用 PRE 开关置位(如 0100)，然后将 5 个开关都同时置为高电平，这样才能观察到开始循环移位。

注意事项：

- (1) 实验板 W5 引脚的时钟频率为 100MHz。
- (2) IP 核中 lab_clk 为分频器输入频率为 100MHz，输出频率为 2Hz。
- (3) 正沿触发 D 触发器的清零端 clr 和置位端 pre 低电平有效，不能留有空脚，否则无法实现清零和置位功能。
- (4) 本实验引脚分配较多，在引脚分配时要注意，最好按照开关，LED 灯对应分配，方便观察实验现象。
- (5) 在设计 4 位并行输入并行输出寄存器时 CLK 直接接到 W5 引脚，因为时钟频率很快，在输入和输出之间几乎没有时间差；若输入后输出无变化，检查清零端是否有效。
- (6) 在设计 4 位循环移位寄存器时，CLK 经过 lab_clk 分频器频率降为 2Hz，在实验板上验证实验时，首先将置位端和清零端都设置为高电平，既开关处于闭合状态，然后置位端开始进行置数，但是要注意置位时间，时钟频率为 2Hz，每隔 0.5 秒就进行一次位移，如果置位时间过长 LED 灯被全部点亮就无法观察到循环过程。

实验问题分析：

- (1) 在并行输入输出寄存器中 LED 灯一直亮，检查输入端口引脚是否为空，若输入端口引脚为空，则输入和输出都为高电位 LED 灯会一直亮，应将输入引脚接到开关引脚上。
- (2) 若置位端和清零端失效，检查对应端口引脚是否为空，置位端和清零端都是低电平有效，空脚输入为高电平，置位和清零功能失效。
- (3) 若循环右移寄存器观察不到循环位移，检测时钟是否进行分频，若未进行分频，时钟频率过高循环过快，无法观察到 LED 灯闪烁，对时钟频率分频降低频率，方便观察现象。

8. 计数器

8.1 实验要求

目的：通过实验，使学生深入理解计数器的工作原理，学会设计和使用计数器。

内容：设计一个模 8 异步计数器和一个模 5 计数器。可使用的 IP 核包括：JK_FF（带低有效置位复位的下降沿触发 JK 触发器）和 lab_clk（时钟分频模块）。

报告：按要求填写实验报告。

扩展：使用附录 2 中 Verilog 语言的结构化描述进行设计及实现。

8.2 实验步骤

8.2.1 模 8 异步计数器

下图是模 8 异步计数器的原理图。我们将依据此图来进行设计。

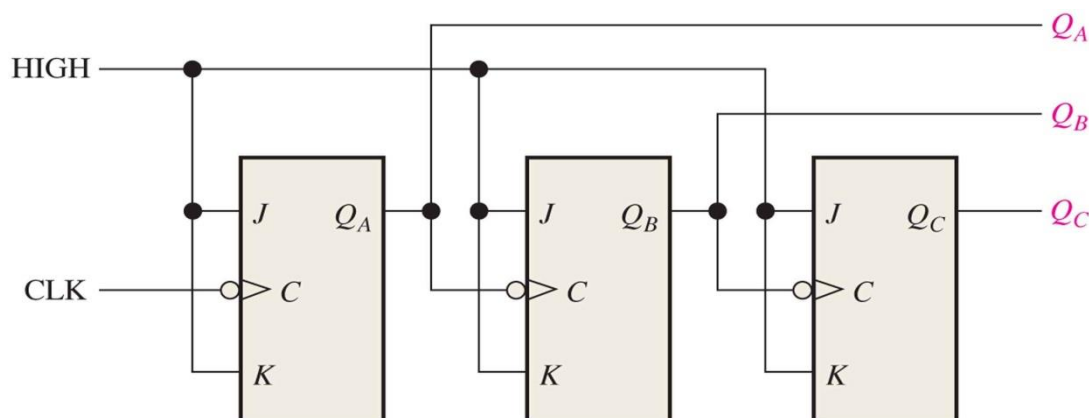


图 8-1 模 8 异步计数器原理图

实验步骤：

- （8） 分析输入、输出；根据方程和 IP 核库判断需要使用的门电路以及个数。
- （9） 按照第二章模块化设计流程新建工程，并且导入正确的 IP 核，准备新建 BlockDesign 文件。为了方便观察结果，使用时钟分频模块对时钟进行分频。
- （10） 参考图 7-2，在导入的 IP 核中选择 JK_FF 和 lab_clk 进行 BlockDesign 设计。

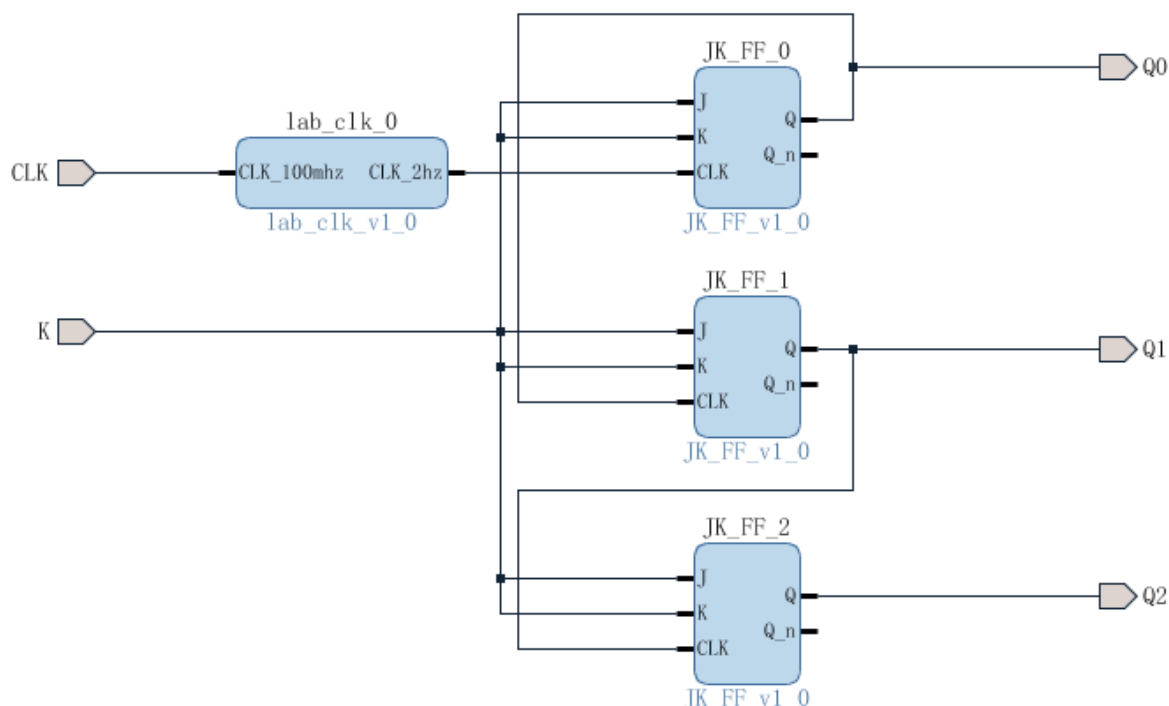


图 8-2 模 8 异步计数器 BD 设计图

(11) BD 文件设计好之后，点击 GenerateOutputProducts 和 CreateHDLWrapper，之后进入 RTL 分析，进行引脚分配如下图：

I/O Ports							
Name	Direction	Neg Diff Pair	Site	Fixed	Bank	I/O Std	
All ports (5)							
Scalar ports (5)							
CLK	Input		W5	<input checked="" type="checkbox"/>	34	LVC MOS33*	
K	Input		R2	<input checked="" type="checkbox"/>	34	LVC MOS33*	
Q0	Output		IF3	<input checked="" type="checkbox"/>	35	LVC MOS33*	
Q1	Output		P1	<input checked="" type="checkbox"/>	35	LVC MOS33*	
Q2	Output		L1	<input checked="" type="checkbox"/>	35	LVC MOS33*	

图 8-3 引脚分配图

(12) 把寄存器输入，清零端和置位端接到开关引脚上面，把输出接到 LED 灯引脚上面，把 CLK 接到时钟 W5 引脚上。

(13) 引脚分配完成之后进行综合，分析，生成比特流。

(14) 最后下载到实验板上面进行验证实验结果：将 K 端接入高电平后应观察到 Q2Q1Q0 呈现 000-001-010-011-100-101-110-111-000 的循环。

8.2.2 模 5 计数器

请在上一节完成的模 8 计数器基础上构建一个模 5 的计数器，使得 LED 灯呈现 000-001-010-011-100-000 的循环。

9.流水灯

9.1 实验要求

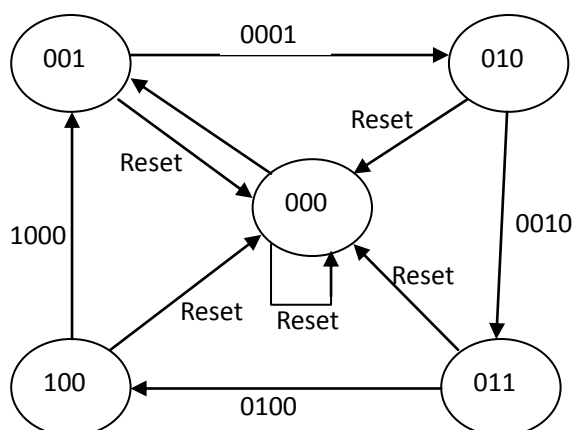
目的：通过实验，使学生学会简单的时序逻辑设计。

内容：使用尽可能少的 D 触发器设计一个 4 位流水灯。要求：4 盏灯一字排列，从右到左，灯依次点亮熄灭，前一盏灯熄灭后，后一盏灯点亮。用 1 表示点亮，0 表示熄灭的话，那么 LED 灯呈现 0001->0010->0100->1000->0001 的循环。该电路还应该有一个使能开关，当开关闭合时，电路开始工作，否则电路保持在 0000。

报告：按要求填写实验报告。

9.2 实验步骤

- 1) 分析输入、输出；列出状态图、状态表、状态方程。以下示例供参考，也可以自己重新设计。设状态 $Q_0Q_1Q_2$ ，状态转换图如下：



Q0	Q1	Q2	Reset	Q0*	Q1*	Q2*	Y0	Y1	Y2	Y3
0	0	0	0	0	0	1	0	0	0	0
0	0	1	0	0	1	0	0	0	0	1
0	1	0	0	0	1	1	0	0	1	0
0	1	1	0	1	0	0	0	1	0	0
1	0	0	0	0	0	1	1	0	0	0
0	0	0	1	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0	1
0	1	0	1	0	0	0	0	0	1	0
0	1	1	1	0	0	0	0	1	0	0
1	0	0	1	0	0	0	1	0	0	0

图 9-1 状态转换图和转换表示例

2) 输出方程: $Y0=Q0Q1'Q2'$

$$Y1=Q0'Q1Q2$$

$$Y2=Q0'Q1Q2'$$

$$Y3=Q0'Q1'Q2$$

状态方程: $Q0^*=Q0'Q1Q2Reset'$

$$Q1^*=(Q0'Q1'Q2+Q0'Q1Q2')Reset'$$

$$Q2^*=(Q0'Q1'Q2'+Q0'Q1Q2'+Q0Q1'Q2')Reset'$$

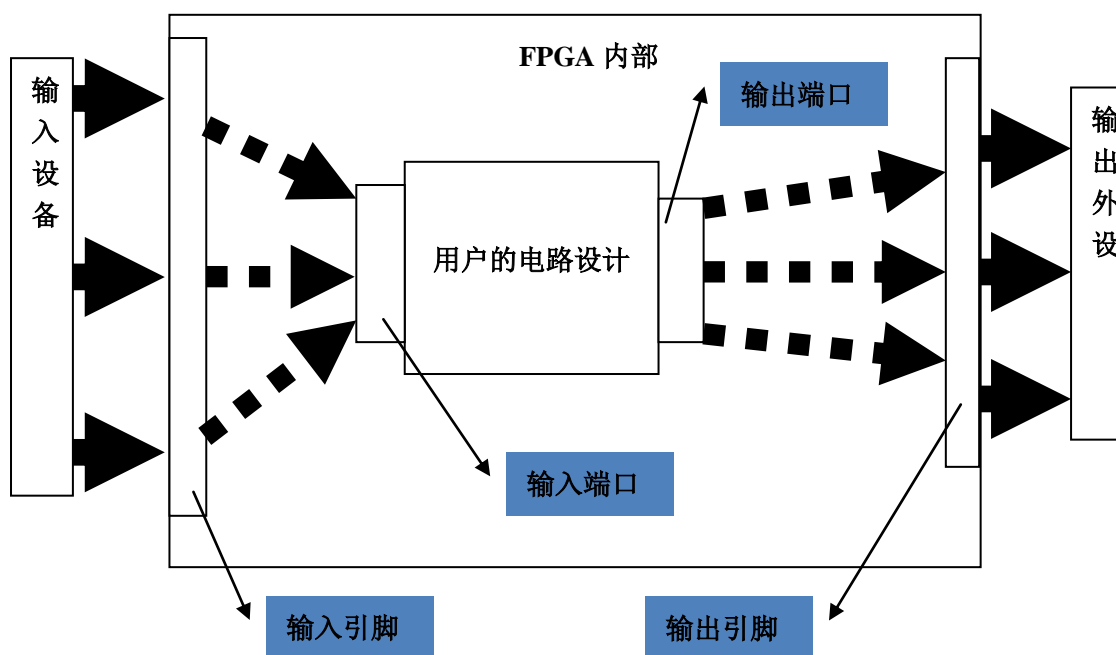
- 3) 根据方程和已有 IP 核判断需要使用的门电路以及个数。
- 4) 新建工程, 并且导入正确的 IP 核, 准备新建 BlockDesign 文件, 根据原理图连线。
- 5) 完成原理图设计后, 生成顶层文件 (Generate Output Products) 和 HDL 代码文件 (Create HDL Wrapper)。
- 6) 配置管脚约束 (I/O PLANNING)。
- 7) 综合、实现、生成 bitstream。
- 8) 下载程序到实验板, 进行板级功能验证。

附录 1：关于 Vivado 中的引脚约束

有些同学会对引脚约束感到困惑：在实验板上，拨码开关被命名为 SW，然而每个拨码开关又具有一个类似于 U17 的编号。

在此，列出以下几点以方便对引脚约束的理解。

1. 电路的实现场所在 FPGA 芯片内部，任何一个数字电路都会有输入输出端口；
2. 将 FPGA 的引脚的一端与设计的端口相对应，相连接，这一过程为引脚约束，这一过程的是可以通过 Vivado 来进行重配置的；
3. 在芯片外部，FPGA 的引脚的另一端与实验板上的外设相连，这种连接是通过焊接和实际的走线实现的，是物理的，不可更改的；



附录 2：了解 Verilog 语言

Verilog 是由 Gateway 设计自动化公司的工程师于 1983 年末创立的。当时 Gateway 设计自动化公司还叫做自动集成设计系统（Automated Integrated Design Systems），1985 年公司将名字改成了前者。该公司的菲尔·莫比（Phil Moorby）完成了 Verilog 的主要设计工作。1990 年，Gateway 设计自动化被 Cadence 公司收购。

1990 年代初，开放 Verilog 国际（Open Verilog International, OVI）组织（即现在的 Accellera）成立，Verilog 面向公有领域开放。1992 年，该组织寻求将 Verilog 纳入电气电子工程师学会标准。最终，Verilog 成为了电气电子工程师学会 1364-1995 标准，即通常所说的 Verilog-95。

设计人员在使用这个版本的 Verilog 的过程中发现了一些可改进之处。为了解决用户在使用此版本 Verilog 过程中反映的问题，Verilog 进行了修正和扩展，这部分内容后来再次被提交给电气电子工程师学会。这个扩展后的版本后来成为了电气电子工程师学会 1364-2001 标准，即通常所说的 Verilog-2001。Verilog-2001 是对 Verilog-95 的一个重大改进版本，它具备一些新的实用功能，例如敏感列表、多维数组、生成语句块、命名端口连接等。目前，Verilog-2001 是 Verilog 的最主流版本，被大多数商业电子设计自动化软件包支持。

2005 年，Verilog 再次进行了更新，即电气电子工程师学会 1364-2005 标准。该版本只是对上一版本的细微修正。这个版本还包括了一个相对独立的新部分，即 Verilog-AMS。这个扩展使得传统的 Verilog 可以对集成的模拟和混合信号系统进行建模。容易与电气电子工程师学会 1364-2005 标准混淆的是加强硬件验证语言特性的 SystemVerilog（电气电子工程师学会 1800-2005 标准），它是 Verilog-2005 的一个超集，它是硬件描述语言、硬件验证语言（针对验证的需求，特别加强了面向对象特性）的一个集成。

2009 年，IEEE 1364-2005 和 IEEE 1800-2005 两个部分合并为 IEEE 1800-2009，成为了一个新的、统一的 SystemVerilog 硬件描述验证语言（hardware description and verification language, HDVL）。

对于硬件设计，它具有以下优点：

- Verilog HDL 是一种通用的硬件描述语言，易学易用。由于它的语法与 C 语言类似，因此对于具有 C 语言编程经验的设计者来说，很容易学习和掌握。
- Verilog HDL 允许在同一个电路模型内进行不同抽象层次的描述。设计者可以从开关、门、RTL 或者行为等各个层次对电路模型进行定义。同时，设计者只需学习一种语言就能够使用它来描述电路的激励，进行层次化设计。
- 绝大多数流行的综合工具都支持 Verilog HDL，这是 Verilog HDL 成为设计者的首选语言的

重要原因之一。

- 所有的制造厂商都提供用于 Verilog HDL 综合之后的逻辑仿真的元件库，因此使用 Verilog HDL 进行设计，即可在更广泛的范围内选择委托制造的厂商。
- 编程语言接口（PLI）是 Verilog 语言最重要的特性之一，它使得设计者可以通过自己编写 C 代码来访问 Verilog 内部的数据结构。设计者可以使用 PLI 按照自己的需要来配置 Verilog HDL 仿真器。

1. Verilog HDL 中的数据类型

1) 线网变量

线网表示硬件单元之间的连接。就像在真实的电路中一样，线网由其连接器件的输出端连续驱动。线网一般使用关键字 `wire` 进行声明。如果没有显示地说明为向量，则默认线网的位宽为 1。线网的默认值为 `z`（高阻态）。线网的值由其驱动源决定，如果没有驱动源，则线网的值为 `z`。举例如下：

```
wire a; //声明 a 是 wire（连线）类型

wire b, c; //声明 b 和 c 也是 wire（连线）类型

assign a = b & c; //a 由 b & c 来驱动

wire d = 1'b0; //连线 d 在声明时，d 被赋值为逻辑值 0，

                // 1'b0 中 1 是数据位数为 1，b 代表二进制数，0 是该数的值，总的意思就是

                // 1 位二进制数 0
```

2) 寄存器变量

寄存器用来表示存储元件，它保持原有的数值，直到被改写。注意不要将这里的寄存器与实际电路中由边沿触发的触发器构成的硬件寄存器混淆。在 Verilog 中，术语 `register` 仅仅意味着一个保持数值的变量。与线网不同，寄存器不需要驱动源，而且也不像硬件寄存器那样需要始终信号。在仿真过程中的任意时刻，寄存器的值都可以通过赋值来改变。寄存器数据类型一般通过使用关键字 `reg` 来声明，默认值为 `x`（不确定状态）。下面给出了如何使用寄存器的例子。

```
reg reset; //声明能保持数值的变量 reset

initial begin //initial 是初始化模块，该模块中的语句只执行一次
```

```

reset = 1'b1; //把 reset 初始化为 1，使数字电路复位

#100 reset = 1'b0; //经过 100 个时间单位后，reset 置逻辑 0

end

```

3) 向量

线网和寄存器类型的数据均可以声明为向量（位宽大于 1）。如果在声明中没有指定位宽，则默认为标量（1 位）。举例如下：

```

wire a; // 标量线网变量，默认

wire [7:0] bus; // 8 位的总线

wire [31:0] busA, busB, busC; // 3 条 32 位宽的总线

reg clock; // 标量寄存器，默认

reg [0:40] virtual_addr; // 向量寄存器，41 位宽的虚拟地址

```

向量通过[high#:low#]或[low#:high#]进行说明，方括号中左边的数总是代表向量的最高有效位。在上面的例子中，向量 virtual_addr 的最高有效位是它的第 0 位。

向量域选择

对于上面的例子中的向量，我们可以指定它的某一位或若干个相邻位。举例如下：

```

busA[7] // 向量 busA 的第 7 位

bus[2:0] // 向量 bus 的最低 3 位

// 如果写成 bus[0:2]是非法的，因为最高位应该写在范围说明的左侧

Virtual_addr[0:1] // 向量 virtual_addr 的两个最高位

```

可变的向量域选择

除了用常量指定向量域以外，Verilog HDL 还允许指定可变的向量域选择。这样就使得设计者可以通过 for 循环来动态地选取向量的各个域。下面是动态选择的两个专用操作符：

```

[<starting_bit>+ : width] : 从起始位开始递增，位宽为 width。

[<starting_bit>- : width] : 从起始位开始递减，位宽为 width。

```

起始位可以是一个变量，但是位宽必须是一个常量。下面的例子说明了可变的向量域选择的使用方法：

```
reg [255:0] data1; // data1[255]是最高位
reg [0:255] data2; // data2[0]是最高位
reg [7:0] byte;

// 用变量选择向量的一部分
byte = data1 [31- : 8]; //从 31 位算起，宽度为 8 位，相当于 data1[31:24]
byte = data1 [24+ : 8]; //从 24 位算起，宽度为 8 位，相当于 data1[31:24]
byte = data2 [31- : 8]; //从 31 位算起，宽度为 8 位，相当于 data1[24:31]
byte = data2 [24+ : 8]; //从 24 位算起，宽度为 8 位，相当于 data1[24:31]

// 起始位可以是变量，但位宽必须是常数。因此可以通过可变域选择，
// 用循环语句选取一个很长的向量的所有位
for (j=0; j<=31; j=j+1)
    byte = data1[(j*8)+ : 8]; // 次序是[7:0], [15:8] ... [255:148]
// 用于初始化向量的一个域
data1[(byteNum*8)+ : 8] = 8'b0; //如果 byteNum = 1，共有 8 位被清零，[15 : 8]
```

4) 参数

Verilog 允许使用关键字 `parameter` 在模块内定义常数。参数代表常数，不能像变量那样赋值，但是每个模块实例的参数值可以在编译阶段重载。通过参数重载使得用户可以对模块实例进行定制。除此之外，还可以对参数的类型和范围进行定义。举例如下：

```
parameter port_id = 5; //定义常数 port_id 为 5
parameter cache_line_width = 256; // 定义高速缓冲器总线宽度为常数 256
parameter signed [15:0] WIDTH; //把参数 WIDTH 规定为有符号，宽度为 16 位
```

通过使用参数，用户可以更加灵活地对模块进行说明。用户不但可以根据参数来定义模块，还可以方便地通过参数重定义来改变模块的行为：通过模块实例化或使用 `defparam` 语句改变参数值。

Verilog 中的局部参数使用关键字 `localparam` 来定义，起作用等同于参数，区别在于它的值不能改变，不能通过参数重载语句（`defparam`）或通过有序参数列表或命名参数赋值来直接修改。例如，状态机的状态编码是不能被修改的，为了避免被意外的更改，应当将其定义为局部参数。举例如下：

```
localparam state1 = 4'b0001,
           state1 = 4'b0010,
           state1 = 4'b0100,
           state1 = 4'b1000;
```

除了以上几种数据类型外，Verilog 中还有 `integer`（整型）、`real`（实型）、`time`（时间型）等数据类型，同学们可以自行查阅相关资料。

2. 模块

Verilog 使用模块（`module`）的概念来代表一个基本的功能块。一个模块可以是一个元件，也可以是低层次的组合。常用的设计方法是使用元件构建在设计中的多个地方使用的功能块，以便进行代码重用。模块通过接口（输入和输出）被高层的模块调用，但隐藏了内部的实现细节。这样就使得设计者可以方便地对某个模块进行修改，而不影响设计的其他部分。

在 Verilog 中，模块声明由关键字 `module` 开始，关键字 `endmodule` 则必须出现在模块定义的结尾。每个模块具有一个模块名，由它唯一的标识这个模块。模块的端口列表则描述这个模块的输入和输出端口。

例 1 模块

```
module <模块名>(<端口列表>);
```

```
...
```

```
<模块的内容>
```

```
...
```

```
...
```

```
endmodule
```

1) 端口

端口是模块与外界环境交互的接口，例如 IC 芯片的输入、输出引脚就是它的端口。对于外部环

境来讲，模块内部是不可见的，对模块的调用（实例引用）只能通过端口进行。这种特点为设计者提供了很大的灵活性：只要接口保持不变，模块内部的修改并不会影响到外部环境。我们也常常将端口称为终端（terminal）。

在模块的定义中包括一个可选的端口列表。如果模块和外部环境没有交换任何信号，则可以没有端口列表。例如我们在进行功能仿真时编写的测试块，通常没有端口。

```
module fulladd4(sum, c_out, a, b, c_in); //有端口列表的模块
```

```
module tb_fulladd4; // 没有端口列表的模块，功能仿真用作测试块
```

端口列表中的所有端口必须在模块中进行声明，Verilog 中的端口具有以下三种类型：

input 输入端口

output 输出端口

inout 输入/输出双向端口

根据端口信号的方向，端口具有三种类型：输入、输出和输入/输出。因此模块 fulladd4 的端口声明如例 2 所示。

例 2 端口声明

```
module fulladd4(sum, c_out, a, b, c_in);
```

```
// 端口声明开始
```

```
output [3:0] sum;
```

```
reg [3:0] sum;
```

```
output c_out;
```

```
reg c_out;
```

```
input [3:0] a, b;
```

```
input c_in;
```

```
// 端口声明结束
```

```
...
```

```
<模块的内容>
```

```
...
```

```
endmodule
```

在 Verilog 中，所有的端口隐含地声明为 wire 类型，因此如果希望端口具有 wire 数据类型，将其声明为三种类型之一即可；如果输出类型的端口需要保存数值，则必须将其显示地声明为 reg 数

据类型，就像例 2 中的 reg [3:0] sum 和 reg c_out。

2) 模块调用

模块声明类似于一个模板，使用这个模板就可以创建实际的对象。当一个模块调用的时候，Verilog 会根据模板创建一个唯一的模块对象，每个对象都有其各自的名字、变量、参数和输入/输出（I/O）接口。从模板创建对象的过程称为实例化（instantiation），创建的对象称为实例（instance）。以下给出模块调用的例子。

例 3 模块调用

//它引用了 4 个 T 触发器

//定义名为 ripple_carry_counter（脉动进位计数器）的模块

```
module repple_carry_counter(q, clk, reset);
```

```
output [3:0] q; // 输入/输出端口的信号和向量声明
```

```
input clk, reset; // 输入/输出端口的信号声明
```

```
// 生成了 4 个 T 触发器 T_FF 的实例，每个实例都有自己的名字，每个实例都传递一组信号
```

```
// 注意每个实例都是 T_FF 模块的副本
```

```
T_FF tff0(q[0], clk, reset);
```

```
T_FF tff1(q[1], q[0], reset);
```

```
T_FF tff2(q[2], q[1], reset);
```

```
T_FF tff3(q[3], q[2], reset);
```

```
endmodule
```

```
// 定义名为 T_FF（T 触发器）的模块，它引用了一个 D 触发器。
```

```
module T_FF(q, clk, reset);
```

```
output q;
```

```
input clk, reset;
```

```
wire d;
```

```
D_FF dff0(q, d, clk, reset); // 调用（实例引用）D_FF，取名为 dff0
```



```
not n1(d, q); //非门 (not) 是 Verilog 语言的内部原语部件 (primitive)

endmodule
```

// 定义名为 D_FF (D 触发器) 的模块。

```
module D_FF(q, d, clk, reset);
```

```
output q;
```

```
input d, clk, reset;
```

```
reg q;
```

// 可以有多种新结构, 不考虑这些结构的功能,

//只需要注意设计块是如何自顶向下的方式编写的

```
always @(posedge reset or negedge clk)
```

```
if (reset)
```

```
    q <= 1'b0;
```

```
else
```

```
    q <= d;
```

```
endmodule
```

3. 数据流级建模

1) 连续赋值语句

连续赋值语句是 Verilog 数据流建模的基本语句, 用于对线网进行赋值。它等价于门级描述, 但是从更高的抽象角度来对电路进行描述。连续赋值语句必须以关键字 `assign` 开始, 连续赋值语句具有以下特点:

- 连续赋值语句的左值必须是一个标量或向量线网, 或者是标量或线网的拼接 (使用 { , } 对信号进行拼接), 而不能是向量或向量寄存器;
- 连续赋值语句总是处于激活状态。只要任意一个操作数发生变化, 表达式就会被立即重新计算, 并且将结果赋给等号左边的线网;
- 操作数可以是标量或向量的线网或寄存器, 也可以是函数调用;

- 赋值延迟用于控制对线网赋予新值的时间，根据仿真时间单位进行说明。赋值延迟类似于门延迟，对于描述实际电路中的时序是非常有用的。

连续赋值语句的举例如下：

//连续赋值语句，out 是线网，i1 和 i2 也是线网

```
assign out = i1 & i2;
```

//向量线网的连续赋值语句，addr 是 16 位的向量线网

//addr1_bits 和 addr2_bits 是 16 位向量寄存器，两者相异或

```
assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];
```

// 拼接操作。赋值操作符左侧是标量线网和向量线网的拼接

```
assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
```

2) 数据流级建模举例

下面以二选一多路选择器对数据流级建模进行说明。

例 4 数据流级建模

```
module mux2to1(out, i0, i1, s);
```

```
// 端口声明
```

```
output out;
```

```
input i0, i1;
```

```
input s;
```

```
// 产生输出 out 的逻辑方程
```

```
assign out = (~s & i0) | (s & i1); //~是逻辑取反，&是逻辑与
```

```
//下面这种描述也是正确的
```

```
//assign out = s ? i1 : i0; // ?: 是条件操作符，如果 s 为真，结果为 i1，否则为 i0
```

```
endmodule
```

4. 行为级建模

在 Verilog 中有两种结构化的过程语句：initial 和 always 语句，它们是行为级建模的两种基本语句。其它所有的行为语句只能出现在这两种结构化过程语句里。

与 C 语言不同，Verilog 在本质上是并发而非顺序的。Verilog 中的各个执行流程（进程）并发执行，而不是顺序执行的。每个 `initial` 语句和 `always` 语句代表一个独立的执行过程，每个执行过程从仿真时间 0 时刻开始执行，并且这两种语句不能嵌套使用。

1) `initial` 语句

所有在 `initial` 语句内的语句构成了一个 `initial` 块。`Initial` 块从仿真 0 时刻开始执行，在整个仿真过程中只执行一次。如果一个模块中包括了若干个 `initial` 块，则这些 `initial` 块从仿真 0 时刻开始并发执行，且每个块的执行时各自独立的。如果在块内包含多条行为语句，那么需要将这些语句组成一组，一般使用关键字 `begin` 和 `end` 将它们组合为一个语句块；如果块内只有一条语句，则不必使用 `begin` 和 `end`。这一点类似于 C 语言中的 `{ }` 语句块，`initial` 块不可综合，一般只在测试块中使用。

下面举例说明 `initial` 语句

例 5 `initial` 语句

```
module stimulus;

reg x, y, a, b, m;

initial

    m = 1'b0; // 只有一条语句，不需要使用 begin-end

initial begin

    #5 a = 1'b1; // 多条语句，需要使用 begin-end

    #10 b = 1'b0;

end

initial begin

    #10 x = 1'b0;

    #25 y = 1'b1;

end

initial

    #50 $finish; //仿真结束
```

在上面的例子中，三条 `initial` 语句在仿真 0 时刻开始并发执行。如果某一条语句前面存在延迟 `#<delay>`，那么对这条 `initial` 语句的仿真将会停顿下来，在经过指定的延迟时间之后再继续执行。

由于 `initial` 块语句在整个仿真期间只执行一次，因此它一般被用于初始化、信号监视、生成仿真波形等目的。

2) always 语句

always 语句包括的所有行为语句构成了一个 always 语句块。该 always 语句块从仿真 0 时刻开始顺序执行其中的行为语句；在最后一句执行完成后，再次开始执行其中的第一条语句，如此循环往复，直到整个仿真结束。因此 always 语句通常用于对数字电路中一组反复执行的活动进行建模。例 6 说明了一种时钟发生器的建立模型的一种方法。

例 6 always 语句

```
module clock_gen (output reg clock);

//在 0 时刻把 clock 变量初始化

Initial

    clock = 1'b0;

// 每半个周期把 clock 信号的值翻转一次（周期 = 20）

always

    #10 clock = ~clock;

Initial

    #1000 $finish;

endmodule
```

3) 过程赋值语句

过程赋值语句的更新对象是寄存器、整数、实数或时间变量。这些类型在被赋值后，其值保持不变，直到被其它过程赋值语句赋予新值。Verilog 包括两种类型的过程赋值语句：阻塞赋值和非阻塞赋值语句。

阻塞赋值语句

串行语句块中的阻塞赋值语句按顺序依次执行。阻塞赋值语句使用“=”作为赋值符。在例 7 中，只有当语句 $x = 0$ 执行完成之后，才执行 $y = 1$ ，而语句 $count = count + 1$ 按顺序在最后执行。由于阻塞赋值语句是按顺序执行的，因此如果在一个 begin-end 块中使用了阻塞赋值语句，那么这个块语句表现的是串行行为。

例 7 阻塞赋值语句

```
reg x, y, z;

reg [15:0] reg_a, reg_b;

integer count;
```

//所有行为语句必须放在 initial 或 always 块内部

Initial begin

x = 0; y = 1; z = 1; //标量赋值

count = 0; //整型变量赋值

reg_a = 16'b0; reg_b = reg_a; // 向量的初始化

#15 reg_a[2] = 1'b1; //带延迟的位选赋值

#10 reg_b[15:13] = {x, y, z}; //把拼接操作的结果赋值给向量的部分位（域）

count = count + 1;

end

在例 7 中，begin-end 块中各条执行的仿真时间为：

- x = 0 到 reg_b = reg_a 之间的语句在仿真 0 时刻执行；
- 语句 reg_a[2] = 0 在仿真时刻 15 执行；
- 语句 reg_b[15 : 13] = {x, y, z} 在仿真时刻 25 执行；
- 语句 count = count + 1 在仿真时刻 25 执行；
- 由于前面的语句中分别包含了 15 和 10 个时间单位的延迟，因此语句 count = count + 1 将在第 25 时刻执行。

非阻塞赋值语句

非阻塞赋值语句允许赋值调度，但它不会阻塞位于同一个顺序块中其后语句的执行。非阻塞赋值使用“<=”作为赋值符。读者会注意到，它与“小于等于”关系操作符是同一个符号，但在表达式中它被解释为关系操作，而在非阻塞赋值的环境下被解释成非阻塞赋值。为了说明非阻塞赋值的意义以及与阻塞赋值的区别，让我们来考虑将例 7 中的部分阻塞赋值改为非阻塞赋值后的结果，例 8 给出了修改后的语句。

例 8 非阻塞赋值语句

reg x, y, z;

reg [15:0] reg_a, reg_b;

integer count;

//所有行为语句必须放在 initial 或 always 块内部

Initial begin

 x = 0; y = 1; z = 1; //标量赋值

 count = 0; //整型变量赋值

 reg_a = 16'b0; reg_b = reg_a; // 向量的初始化

 reg_a[2] <= #15 1'b1; //带延迟的位选赋值

 reg_b[15:13] <= #10 {x, y, z}; //把拼接操作的结果赋值给向量的部分位（域）

 count <= count + 1;

end

在这个例子中，从 x = 0 到 reg_b = reg_a 之间的语句是在仿真 0 时刻顺序执行的，之后的 3 条非阻塞赋值语句在 reg_b = reg_a 执行完成后并发执行。

- reg_a[2] = 1'b1 被调度到 15 个单位之后执行，即仿真时刻为 15；
- reg_b[15:13] = {x, y, z} 被调度到 10 个时间单位之后执行，即仿真时刻为 10；
- count <= count + 1 被调度到无任何延迟执行，即仿真时刻为 0。

从上面的分析中可以看到，仿真器将非阻塞赋值调度到相应的仿真时刻，然后继续执行后面的语句，而不是停下来等待赋值的完成。一般情况下，非阻塞赋值是在当前仿真时刻的最后一个时间同步，即阻塞只完成之后才执行。

在上面的例子中，我们把阻塞和非阻塞赋值语句混合在一起使用，目的是想清楚地比较和说明它们的行为。需要提醒大家注意的是，不要在同一个 always 块中混合使用阻塞和非阻塞赋值语句。通常情况下，组合逻辑中使用阻塞赋值语句，时序逻辑中使用非阻塞赋值语句。

此外，还有条件语句(if-else-if)、多路分支语句(case- default-endcase)及循环语句(for 和 while)，其语法与 C 语言类似，不再一一讲述。

4) 行为级建模举例

同样以二选一多路选择器为例，来说明行为级建模过程。

例 9 行为级建模

```
module mux2to1(out, i0, i1, s);
```

```
// 端口声明
```

```
output out;
```

```
input i0, i1;
```

```

input s;

//输出端口被声明为寄存器类型变量

reg out;

// 若输入信号改变，则重新计算输出信号 out
//造成输出信号 out 重新计算的所有输入信号必须写入 always @(...)的电平敏感列表中
always @(s, i0, i1) begin

    if(s) out = i1; //s 为 1 输出 i1，否则输出 i0

    else out = i0;

end

endmodule

```

在实际的工程应用各种建模方式不是独立使用的，通常是混合起来使用，使我们的设计更灵活。

5. 实际应用

下面给出了 74LS138 译码器、74LS148 优先编码器以及流水灯实现的 Verilog 代码，作为参考。

1) 74LS138 译码器 Verilog 代码:

功能模块:

```

`timescale 1ns / 1ps

module ls138( //模块名称

    input en,          //使能端

    input [2:0] sel,   //3 位输入端

    output [7:0] y     //8 位输出端

);

reg [7:0] y; //reg 型变量在 always 块中才能被赋值，所以将输出端 y 声明为 reg 型

always @(en, sel) begin //always 块开始,括号内是敏感信号，当信号变化时，执行 always 块

    if(!en) y = 8'b11111111; //使能信号 en 无效（为低）时，输出默认状态

    else begin

        case(sel) //case 块中是 8 种状态的译码

```

```
3'b000: y = 8'b1111_1110; //数字中的下划线没有实际意义，仅用作连接，
                               //使数据易读

3'b001: y = 8'b1111_1101;

3'b010: y = 8'b1111_1011;

3'b011: y = 8'b1111_0111;

3'b100: y = 8'b1110_1111;

3'b101: y = 8'b1101_1111;

3'b110: y = 8'b1011_1111;

3'b111: y = 8'b0111_1111;

default: y = 8'b1111_1111; //默认状态

    endcase

end

end //always 块结束

endmodule

测试模块:

`timescale 1ns / 1ps

module tb_ls138;

    // 输入

    reg en;

    reg [2:0] din;

    // 输出

    wire [7:0] dout;

    // 实例化被测试模块 ls138

    ls138 ls138_init (

        .en(en),

        .sel(din),

        .y(dout)

    );
```



```

initial begin

    // 初始化输入信号

    din = 0;

    en = 0;    //使能无效

    #100;    //延迟 100 个时钟

    en = 1;    //使能有效

end

always #30 din = din + 1;    //每 30 个时钟，输入加 1（改变一个状态）

endmodule

```

2) 74LS148 优先编码器 Verilog 代码:

功能模块:

```

`timescale 1ns / 1ps

module ls148(    //模块名称

    input [7:0] I,    //8 位输入端

    input EI,    //使能输入端

    output [2:0] A,    //3 位输出端

    output GS,    //输出端 GS

    output EO    //输出端 EO

);

reg [2:0] A;

reg GS;

reg EO;

always @ ( I, EI ) begin

    if ( EI ) begin

        A = 3'b111;

        GS = 1;

```

```
        EO = 1;

    end

    else if ( I[7] == 0 ) begin    //I[7]有最高优先级

        A = 3'b000;

        GS = 0;

        EO = 1;

    end

    else if ( I[6] == 0 ) begin

        A = 3'b001;

        GS = 0;

        EO = 1;

    end

    else if ( I[5] == 0 ) begin

        A = 3'b010;

        GS = 0;

        EO = 1;

    end

    else if ( I[4] == 0 ) begin

        A = 3'b011;

        GS = 0;

        EO = 1;

    end

    else if ( I[3] == 0 ) begin

        A = 3'b100;

        GS = 0;

        EO = 1;

    end

    else if ( I[2] == 0 ) begin

        A = 3'b101;

        GS = 0;
```

```
        EO = 1;

    end

    else if ( I[1] == 0 ) begin

        A = 3'b110;

        GS = 0;

        EO = 1;

    end

    else if ( I[0] == 0 ) begin

        A = 3'b111;

        GS = 0;

        EO = 1;

    end

    else begin

        A = 3'b111;

        GS = 1;

        EO = 0;

    end

end

end

endmodule
```

测试模块:

```
`timescale 1ns / 1ps

module tb_ls148;

    //输入

    reg EI;

    reg [7:0] I;

    //输出

    wire [2:0] A;

    wire GS;

    wire EO;

    //实例化被测测试模块
```

```

ls148 ls148_inst(
    .EI(EI),
    .I(I),
    .A(A),
    .GS(GS),
    .EO(EO)
);
//初始化输入信号
initial begin
    //初始化输入信号
    EI = 1;
    I = 8'b1111_1111;
    #100
    EI = 0;
end
always #5 I = I - 1;
endmodule

```

3) 流水灯 Verilog 代码:

```

`timescale 1ns / 1ps

module flowLED(
    input clk,    //100MHz 时钟
    input rst,    //复位
    input on_off, //控制启停, 0-off, 1-on
    input DIR,    //控制移动方向, 0-左移, 1-右移
    output [3:0] LED //4 位 LED
);
//10ms = 1/100MHz * 1000_000
parameter T100MS = 999_9999;
parameter cntWidth = 24;

```

```
reg [cntWidth - 1 : 0] cnt;    //计数

reg clken;    //时钟使能

//计数延迟

always @(posedge clk, negedge rst) begin

    if(!rst) begin    //rst 为低时，复位计数和时钟使能

        cnt <= 0;

        clken <= 0;

    end

    else if(cnt == T100MS) begin

        cnt <= 0;    //每延迟 10ms，重新计数

        clken <= 1;    //每延迟 10ms，时钟使能 1 次

    end

    else begin

        cnt <= cnt + 1'b1;

        clken <= 0;

    end

end

end

reg [3:0] LEDr;

always @(posedge clk, negedge rst) begin

    if(!rst) //rst 为低时，复位 LEDr

        LEDr <= 4'b0001;

    else if(on_off && clken) begin    //on_off = 1,使能有效时，开始移位

        if(!DIR)

            LEDr <= {LEDr[2:0], LEDr[3]};    //左移

        else if(DIR)

            LEDr <= {LEDr[0], LEDr[3:1]};    //右移

    end

end
```

```
    else

        LEDr <= LEDr;

    end

    assign LED = rst ? LEDr : 4'b0000;    //rst = 0 时，LED 复位
endmodule
```

如果想更全面的了解和掌握 Verilog HDL，请同学们自行阅读夏宇闻老师编写的《Verilog HDL 数字设计与综合》。