
FPGA 在中低能实验核物理中的应用

发布 编写中

Hongyi Wu(吴鸿毅)

2021 年 06 月 09 日

Contents:

1	README	1
2	ISE/Altera/Vivado 软件	3
2.1	软件安装	3
2.2	ISE 软件	27
2.3	Altera 软件	37
2.4	Vivado 软件	65
3	语法	67
3.1	VHDL	67
3.2	verilog	83
3.3	时序约束	96
3.4	原语	98
4	硬件介绍	107
4.1	Lupo	107
4.2	DT5495	107
4.3	MZTIO	108
4.4	DT5550	109
4.5	R5560	109
5	计数器	111
5.1	计数器	111
6	状态机	117
6.1	状态机	117
7	FIFO	121
7.1	FIFO	121
8	高层次综合	123
8.1	高层次综合	123
9	经验总结	131
9.1	经验总结	131
9.2	verilog 临时存放	139
9.3	VHDL temp	142
9.4	LPM (library of parameterized modules)	147
9.5	attribute	152

CHAPTER 1

README

国际上最大的两个 FPGA 厂家为 Xilinx 和 Altera（已被 Intel 收购），Xilinx 公司的软件有 ISE(已停止更新，适用于早年推出的 FPGA) 和 Vivado (适用于新推出的 FPGA)，Altera 公司的软件为 Quartus。我们已有的 LUPO/DT5495/MZTIO 可编程逻辑刚好对应三个软件。对于数字信号处理模块，已有的 DT5550/R5560SE 已经在测试中。

编程语言 VHDL/verilog 两种，本质上没有多大的区别。我已经写好两种语言的关键模块的模版，直接套用就可以。整个编程的核心就是熟练掌握计数器/状态机以及 FIFO 的使用。

- ISE/Altera/Vivado 软件

- 软件安装

- * ISE
 - * Vivado
 - * Quartus

- 刷固件
 - 工程使用案例
 - 仿真
 - 常用 IP 核介绍

- 语法

- VHDL
 - verilog

- LUPO/DT5495/MZTIO 模块介绍

- LUPO
 - DT5495
 - MZTIO
 - DT5550
 - R5560SE

- 计数器
- 状态机

- FIFO

- 项目实践

- 计数器

- * 时钟降频 (很简单)
 - * LED 灯 (很简单)
 - * scaler (很简单)
 - * 信号展宽 (很简单)
 - * UART(有个不错的 verilog 模版, 需要进一步优化, 提高普适性)
 - * IIC
 - * 显示器实现示波器功能

- 状态机

- * SPI(CAEN DT5495 有个 VHDL 模版, 需要按照我们的规范重新优化)
 - * 基于 FPGA 的在线监视 (将获取的数据直接发送到可编程板卡进行处理, 可进行初步的处理后, 再通过网络发送到下游的服务器存储)

- FIFO

- * 信号延迟 (简单)

- 脉冲发生器

- * 基本波形的脉冲产生 (简单)
 - * 任意波形的脉冲产生

-

- 数字信号触发算法

- * XIA fast filter
 - * RC-CR2
 - * 前沿甄别
 - *

- 数字能量算法

- * XIA 梯形算法
 - * GAMMASPHERE 梯形算法
 - * Semi-Gaussian
 - * QDC 算法

- 数字时间算法

- * CFD 算法
 - * 高精度时间算法

参考文献

- 手把手教你学 FPGA 设计-基于大道至简的至简设计法
- VHDL 入门-解惑-经典实例-经验总结
- Verilog 数字系统设计教程

CHAPTER 2

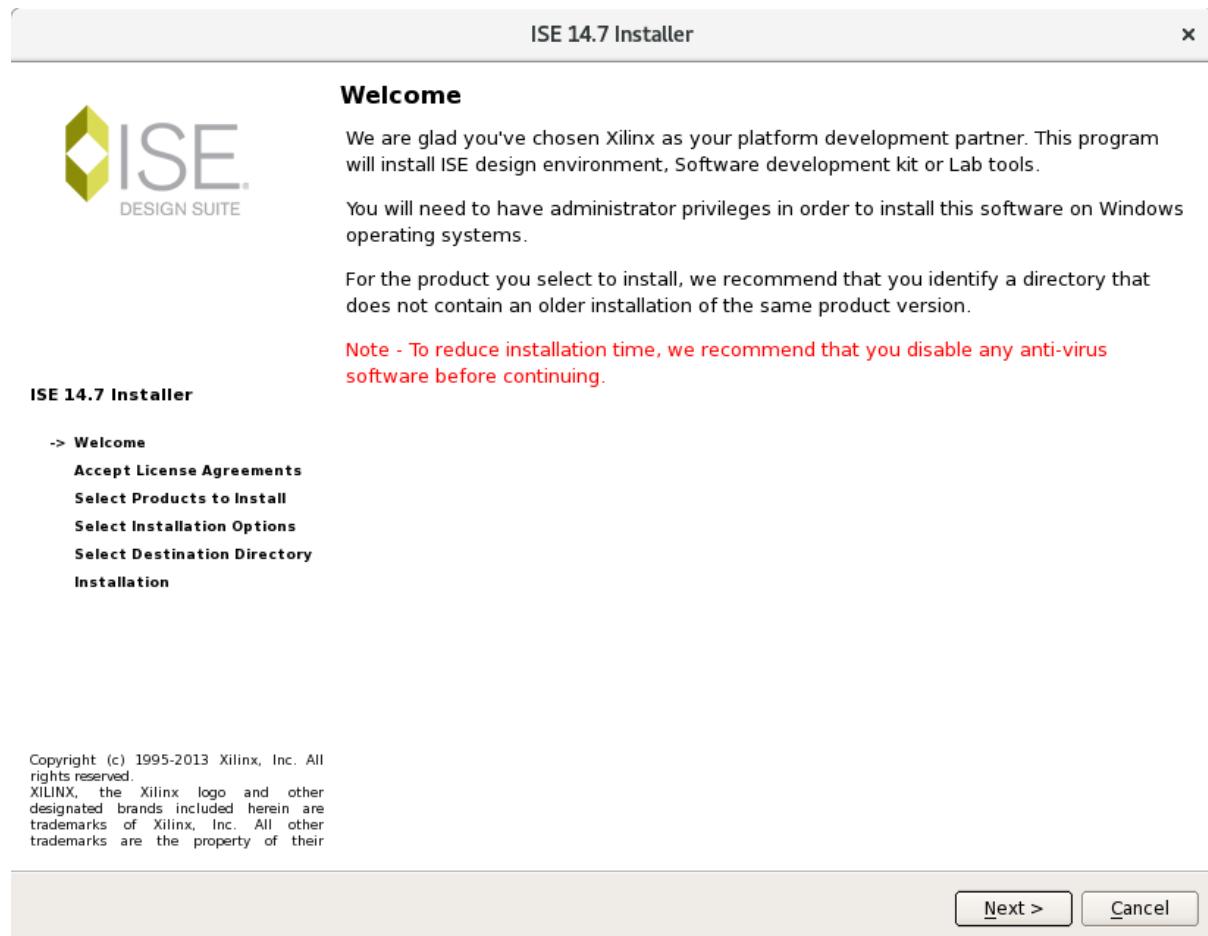
ISE/Altera/Vivado 软件

2.1 软件安装

2.1.1 ISE 安装

ISE 最后一个版本是 14.7，该版本在 LINUX 系统中只能支持 CentOS 6，在 CentOS 7 中存在问题。

```
tar -xvf Xilinx_ISE_DS_Lin_14.7_1015_1.tar  
cd Xilinx_ISE_DS_Lin_14.7_1015_1  
./xsetup
```



Copyright (c) 1995-2013 Xilinx, Inc. All rights reserved.
XILINX, the Xilinx logo and other designated brands included herein are trademarks of Xilinx, Inc. All other trademarks are the property of their

ISE 14.7 Installer

Accept License Agreements (1 of 2)

Please read the following terms and conditions and click the checkbox below it to indicate that you accept and agree.

XILINX, INC.
END USER LICENSE AGREEMENT

CAREFULLY READ THIS END USER LICENSE AGREEMENT ("AGREEMENT"). BY CLICKING THE "ACCEPT" OR "AGREE" BUTTON, OR OTHERWISE ACCESSING, DOWNLOADING, INSTALLING OR USING THE SOFTWARE, YOU AGREE ON BEHALF OF LICENSEE TO BE BOUND BY THIS AGREEMENT.

IF LICENSEE DOES NOT AGREE TO ALL OF THE TERMS AND CONDITIONS OF THIS AGREEMENT, DO NOT CLICK THE "ACCEPT" OR "AGREE" BUTTON OR ACCESS, DOWNLOAD, INSTALL OR USE THE SOFTWARE; AND IF LICENSEE HAS ALREADY OBTAINED THE SOFTWARE FROM AN AUTHORIZED SOURCE, PROMPTLY RETURN IT FOR A REFUND.

1. Definitions

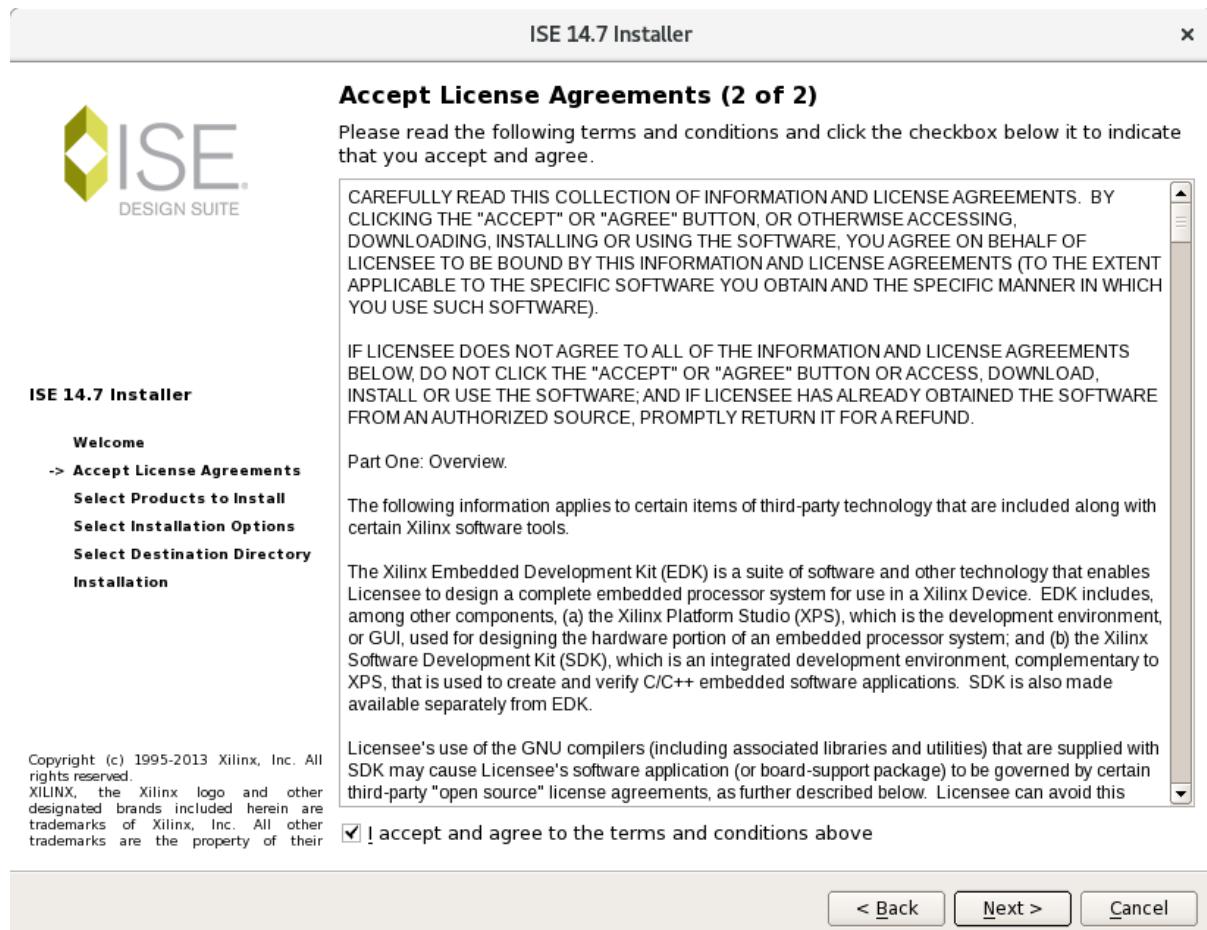
"Authorization Codes" means any FLEXIm license key, license file, license manager, dongle or other key, code or information issued by (or on behalf of) Xilinx that is necessary to download, install, operate and/or regulate User access to the applicable Software.

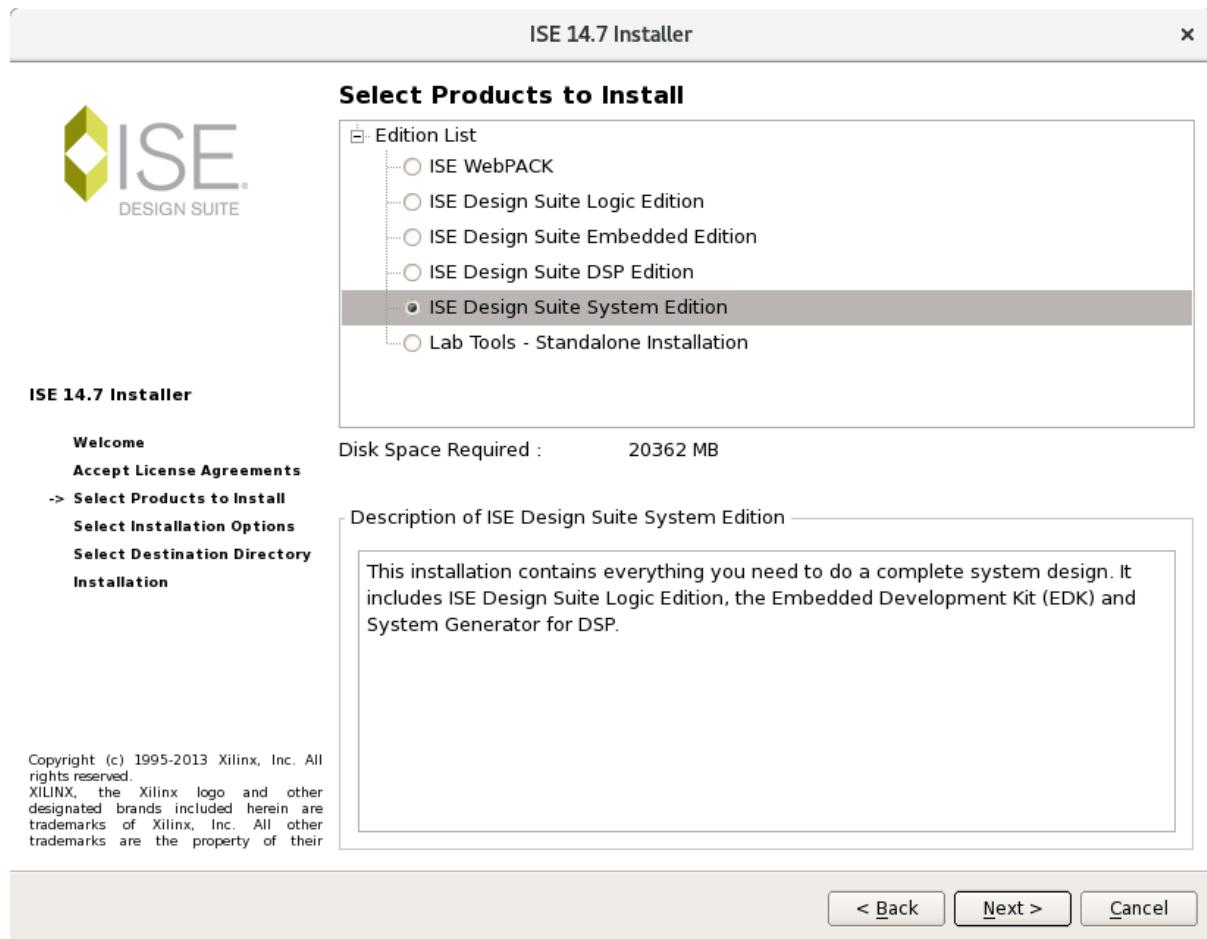
I accept and agree to the terms and conditions above

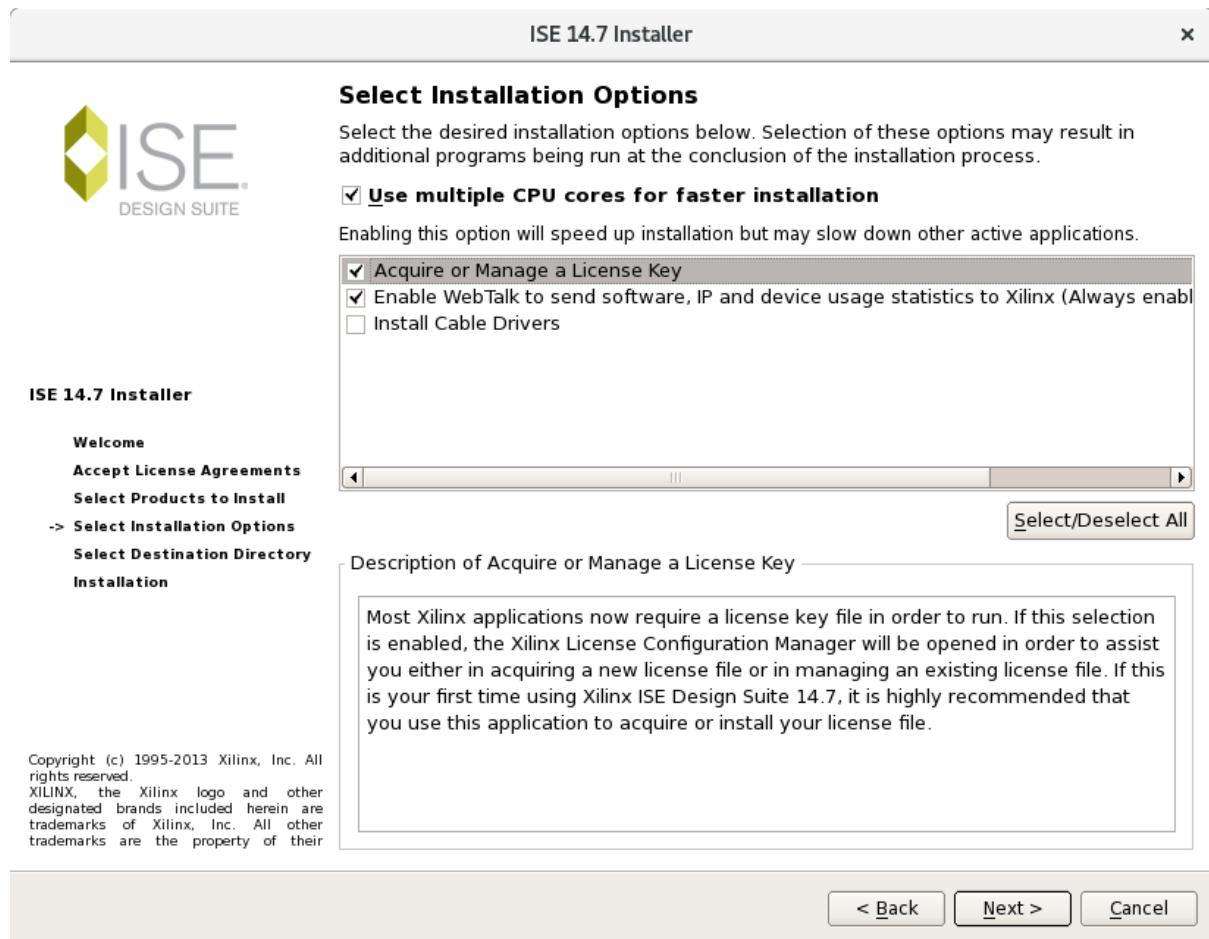
I also accept and agree to the following terms and conditions

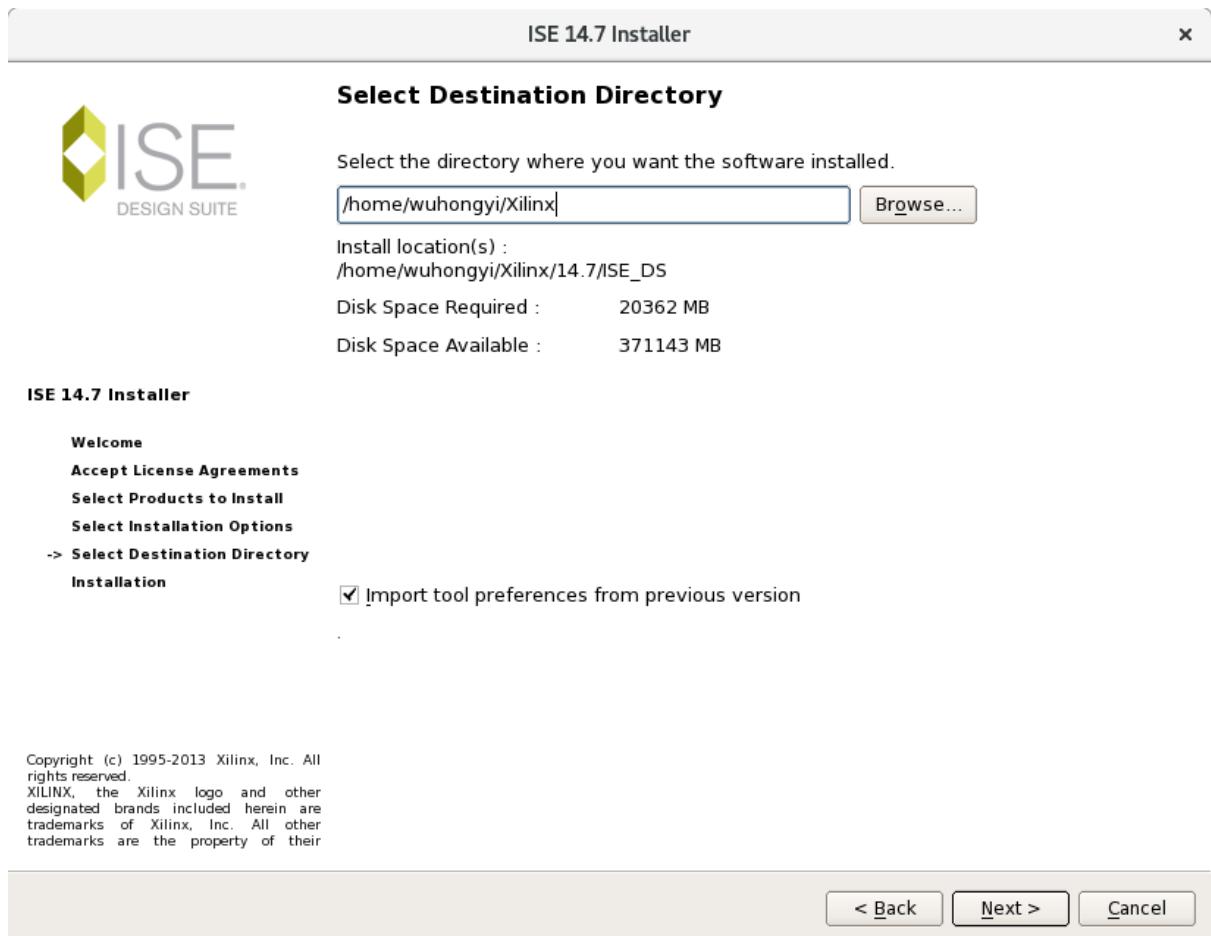
I also confirm that I have read Section 13 of the terms and conditions above concerning WebTalk and have been afforded the opportunity to read the WebTalk FAQ posted at <http://www.xilinx.com/ise/webtalk/>. I understand that I am able to disable WebTalk later if certain criteria described in Section 13(c) apply. If they don't apply, I can disable WebTalk by uninstalling the Software or using the Software on a machine not connected to the internet. If I fail to satisfy the applicable criteria or if I fail to take the applicable steps to prevent such transmission of information, I agree to allow Xilinx to collect the information described in Section 13(a) for the purposes described in Section 13(b).

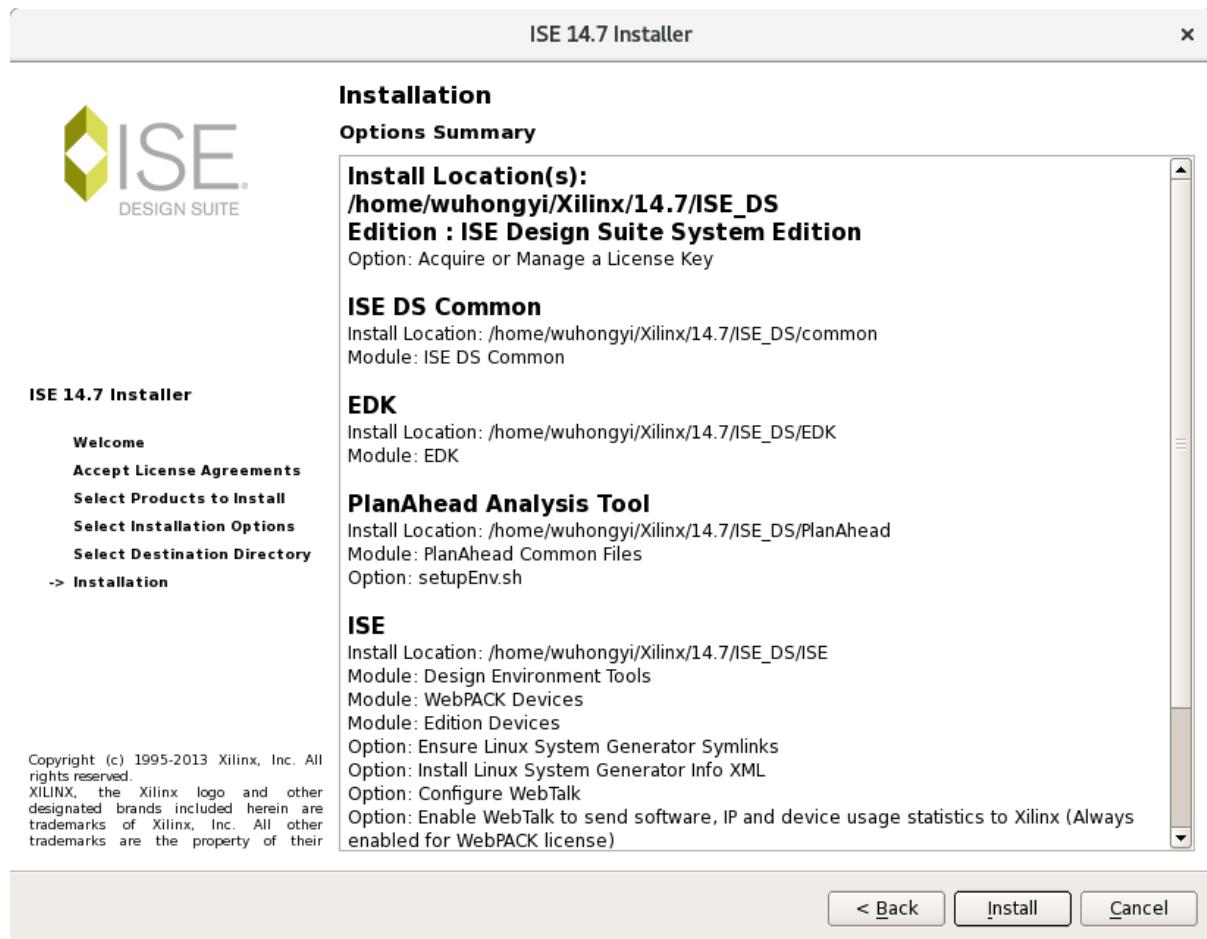
[**< Back**](#) [**Next >**](#) [**Cancel**](#)

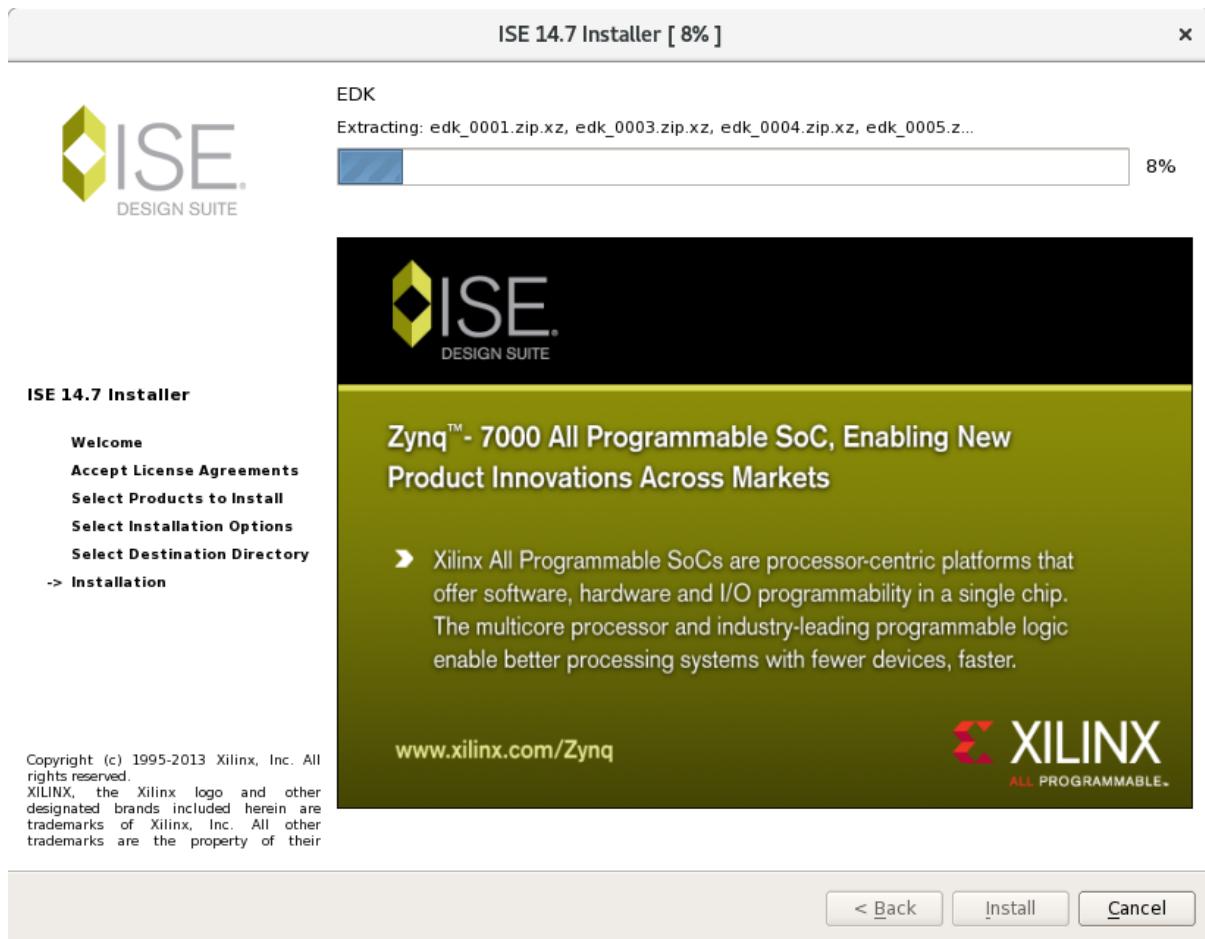










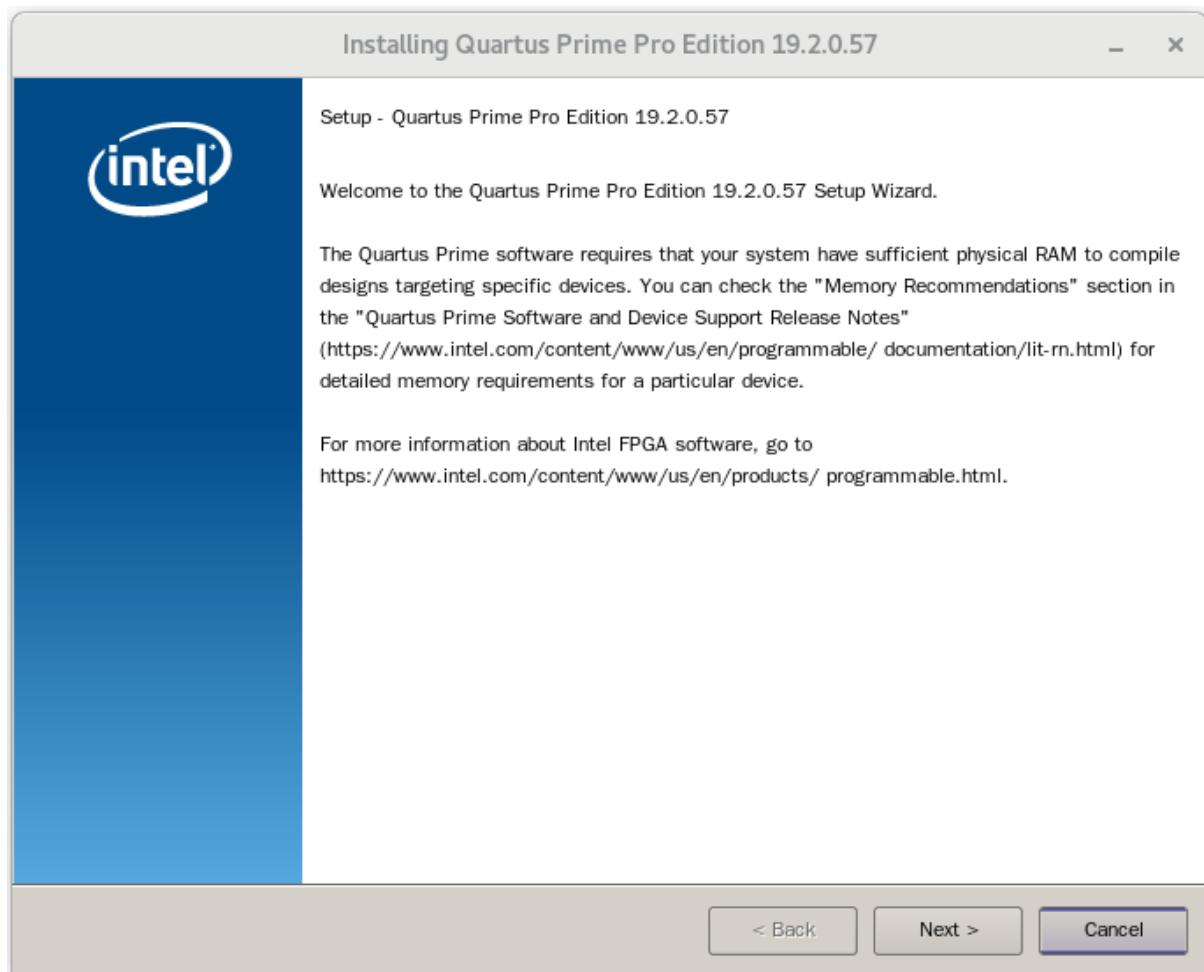


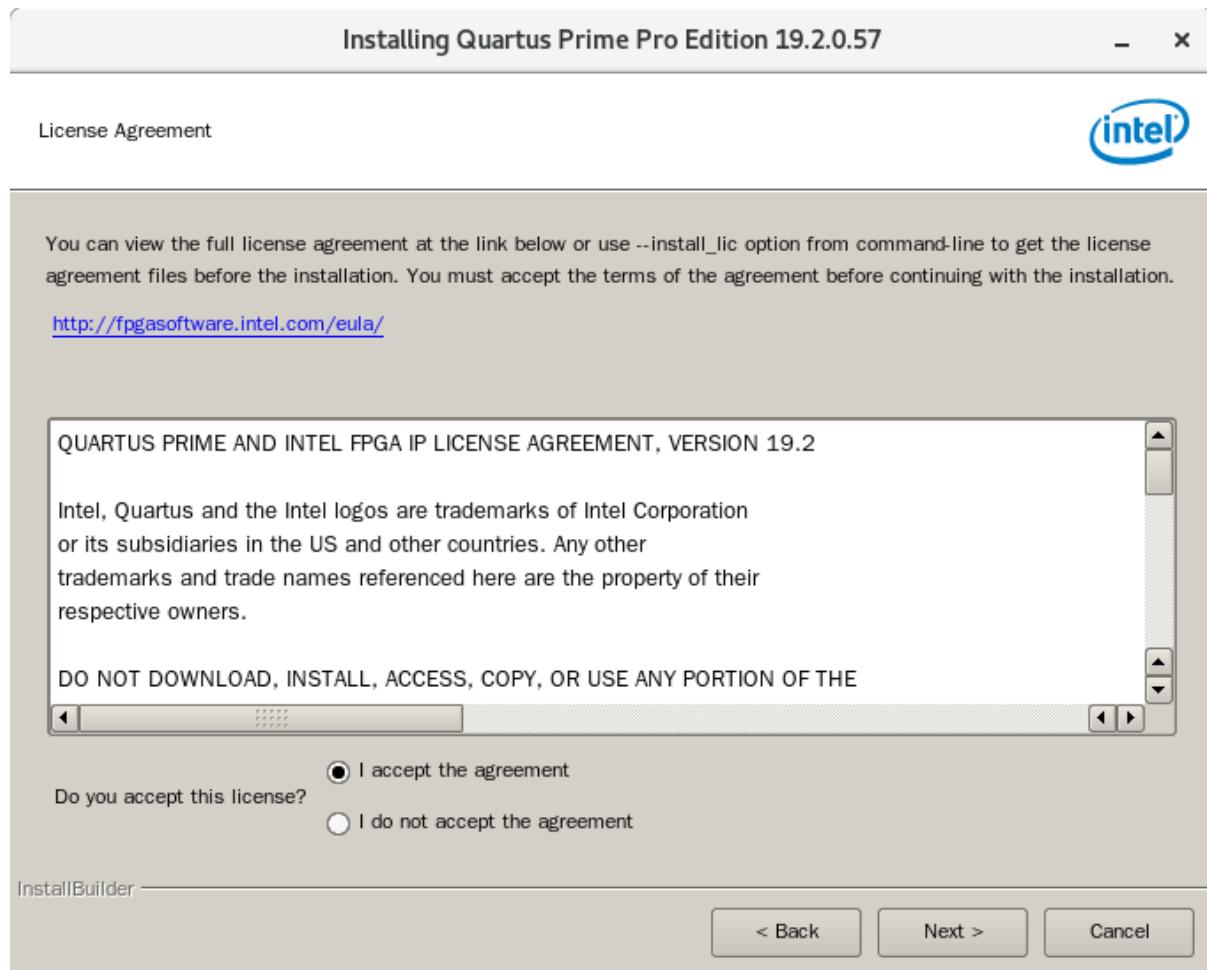
2.1.2 Altera 安装

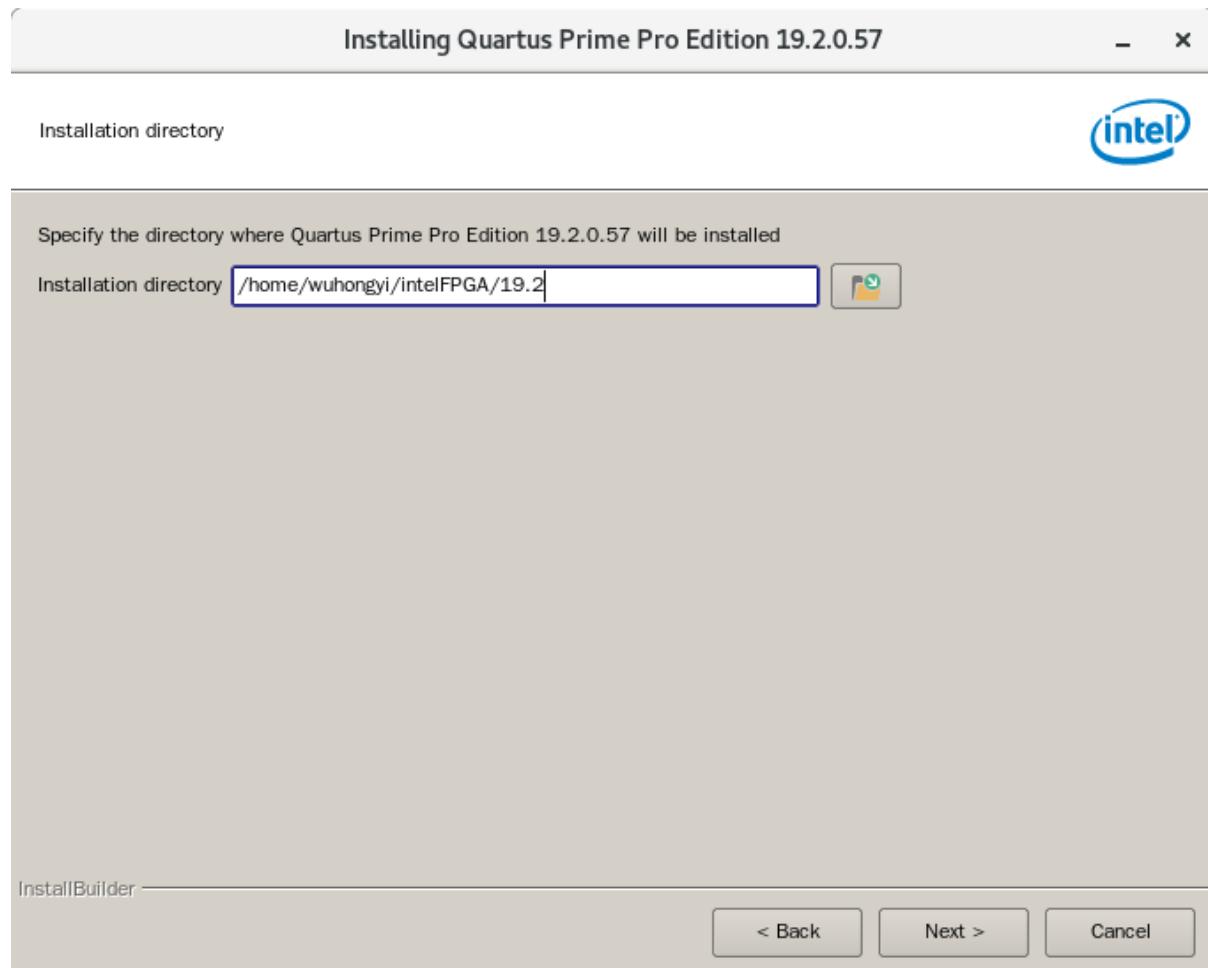
下载需要的所有文件，放在一个目录下

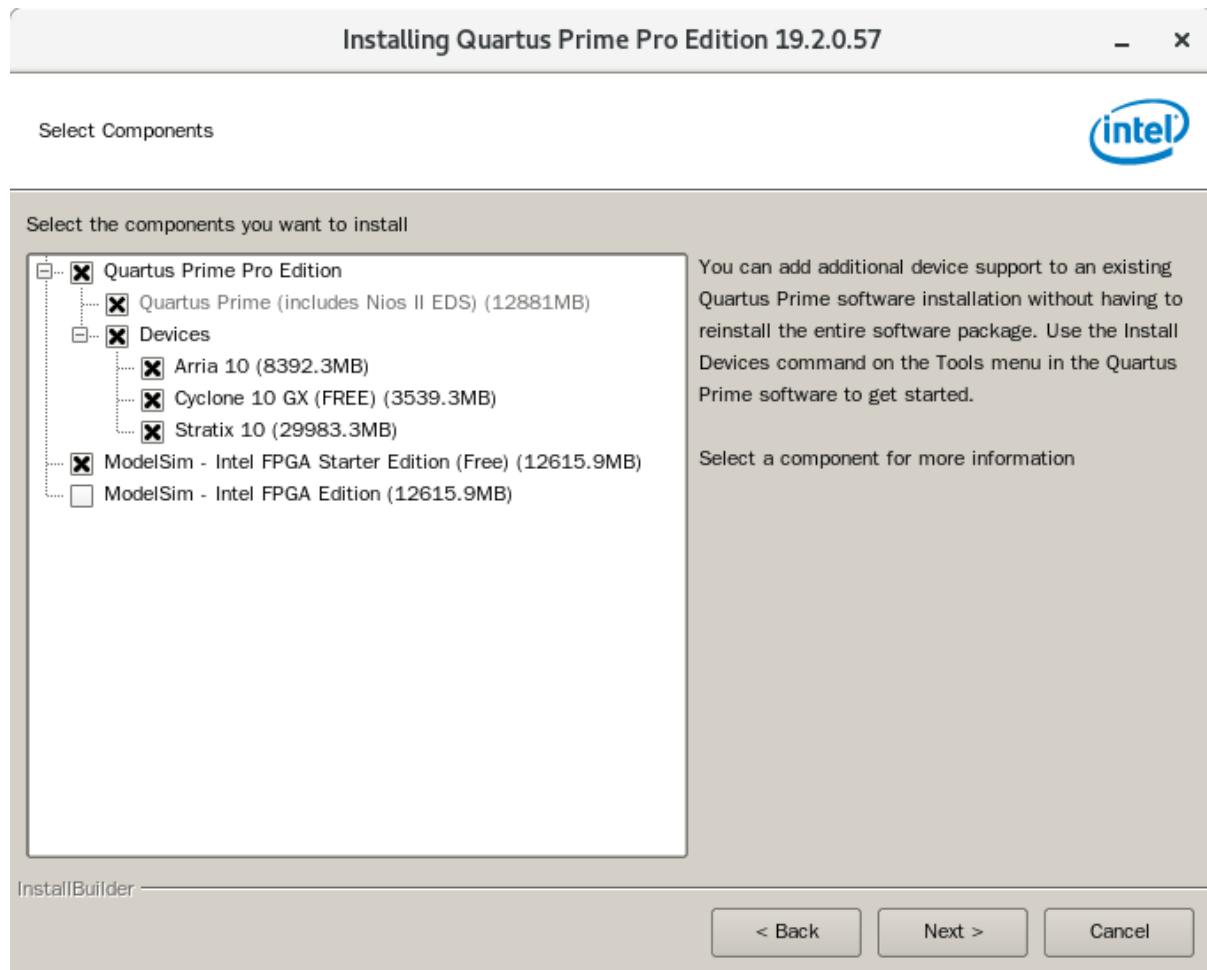
arria10-19.2.0.57.qdz	ModelSimProSetup-19.2.0.57-linux.run
cyclone10gx-19.2.0.57.qdz	QuartusProSetup-19.2.0.57-linux.run
modelsim-part2-19.2.0.57-linux.qdz	stratix10-19.2.0.57.qdz

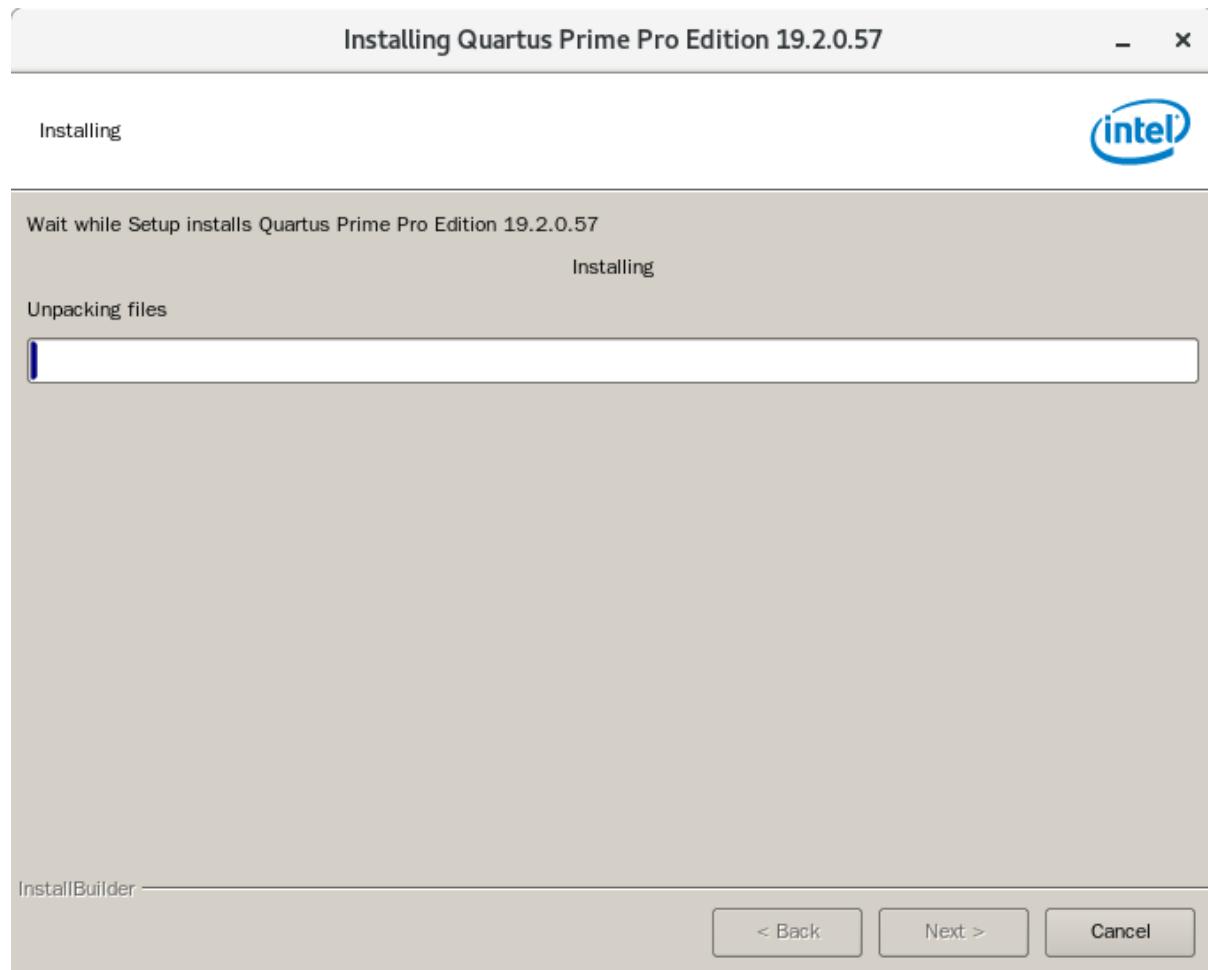
```
chmod +x QuartusProSetup-19.2.0.57-linux.run
./QuartusProSetup-19.2.0.57-linux.run
```

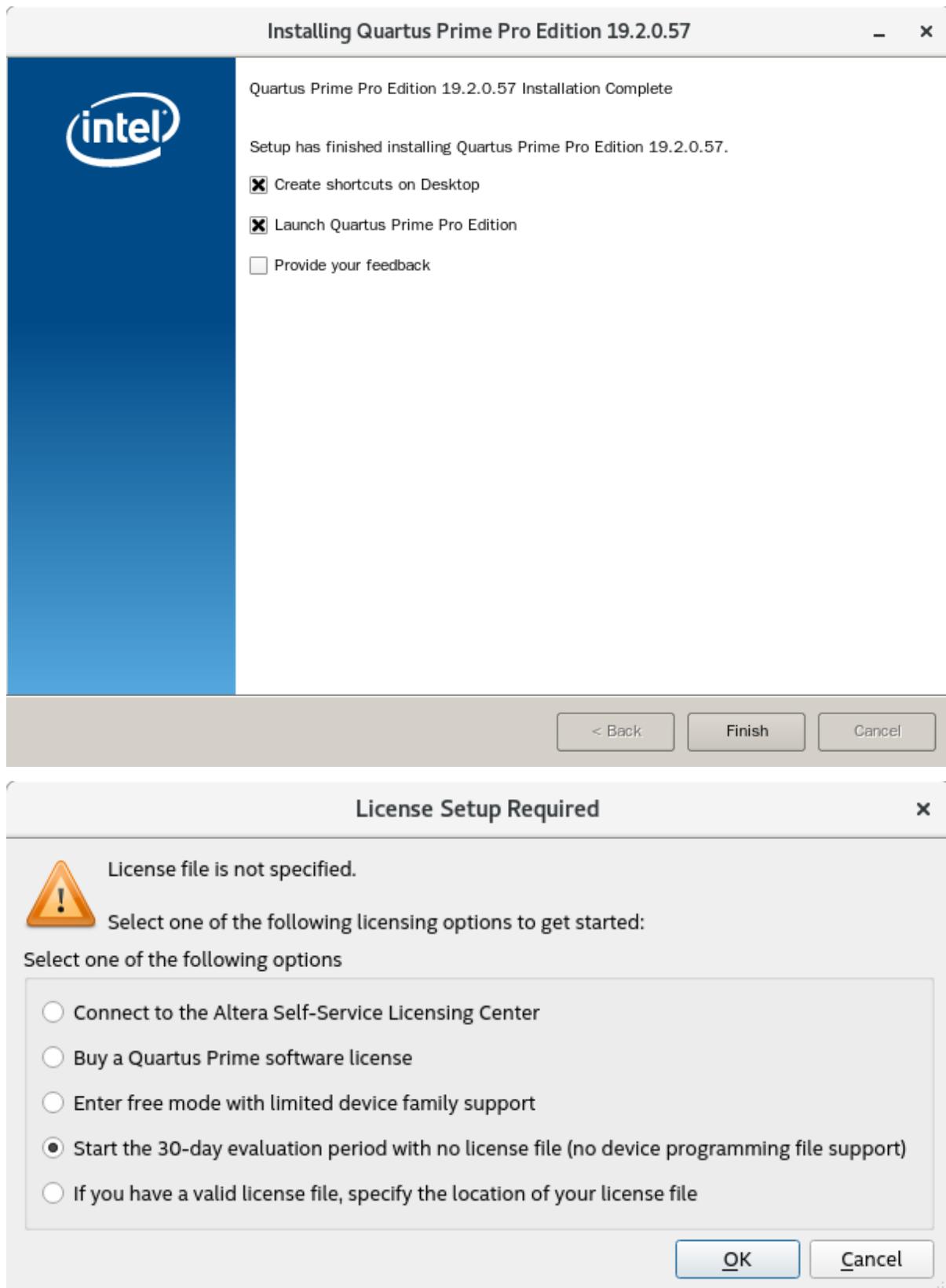












在安装路径下有以下文件

devdata	licenses	modelsim_ase	qsys	syscon
ip	logs	nios2eds		quartus uninstall

quartus/bin 文件夹内存放 quartus 启动的脚本

```
./quartus
```

modelsim_ase/bin 文件夹内存放 modelsim 启动的脚本

```
./vsim
```

CentOS 7 中 ModelSim 对依赖软件 freetype 版本有一定的要求。经过测试表明 2.4.12 版本可以支持。通常采用 modulefile 来对系统中的软件进行多版本控制，以下是该软件的配置示例。

```
#%Module 1.0
# 这一行一般 module file 都有

set _module_name [module-info name]
set is_module_rm [module-info mode remove]
set sys [uname sysname]
set os [uname release]

# 冲突标识符 gcc
conflict freetype

set FREETYPE_CURPATH /opt/freetype/2.4.12
set FREETYPE_LEVEL 2.4.12
set FREETYPE_MAJLEVEL 2.4

# 所需路径
prepend-path PATH $FREETYPE_CURPATH/bin
prepend-path LD_LIBRARY_PATH $FREETYPE_CURPATH/lib
prepend-path LIBRARY_PATH $FREETYPE_CURPATH/lib
```

linux usb blaster 权限的设置

对于错误 error (209053): unexpected error in jtag server – error code 89，它产生的原因在于，在 linux 系统下，Quartus ii 的驱动 USB-Blaster 只能有 root 用户使用，而普通用户是无权使用的。解决思路是更改 USB-Blaster 的使用权限，使得普通用户也能使用。

因为 usb 默认只有 root 才有权限访问，所以只要把权限修改一下即可，usb blaster 链接上电脑

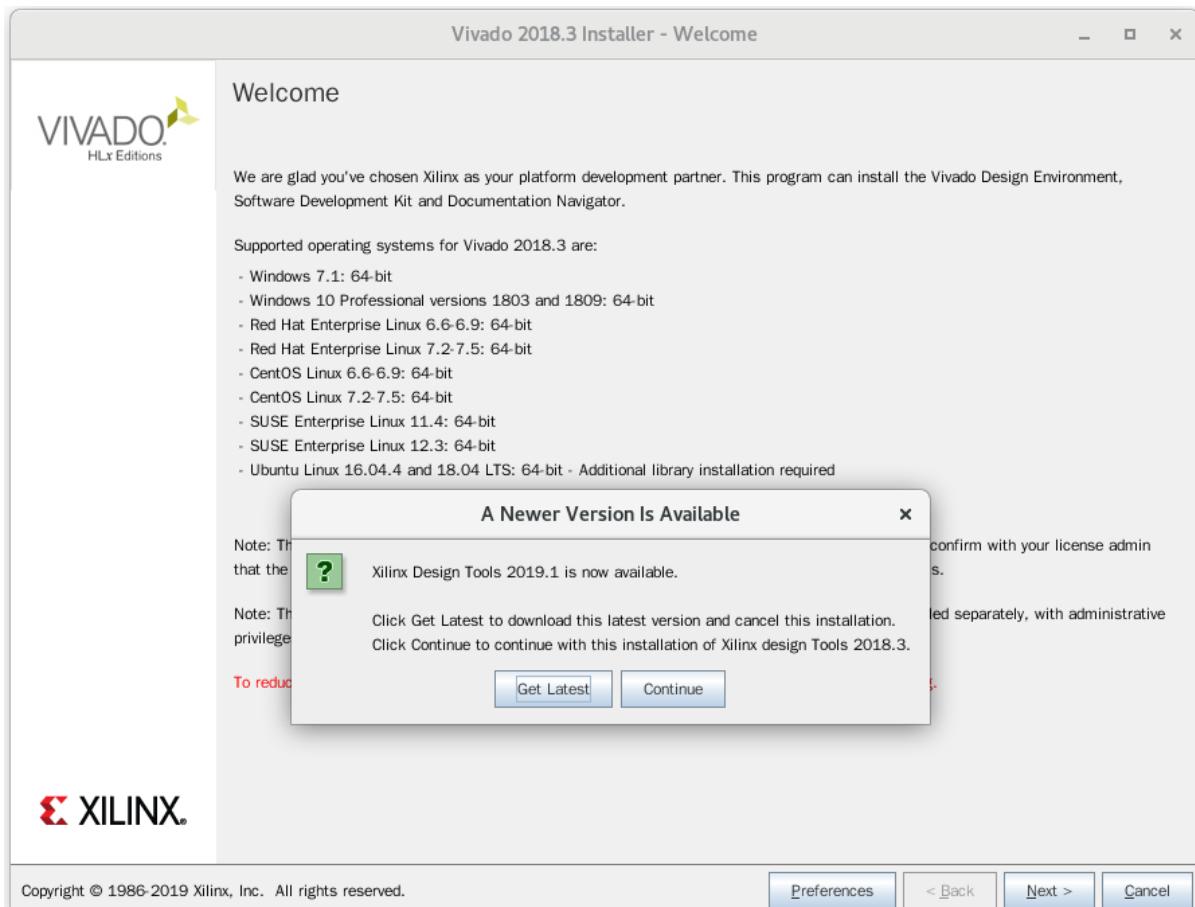
```
[root@localhost 003]# lsusb
Bus 002 Device 002: ID 8087:8000 Intel Corp.
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 002: ID 8087:8008 Intel Corp.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 004 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 003 Device 004: ID 0bda:0184 Realtek Semiconductor Corp. RTS5182 Card Reader
Bus 003 Device 013: ID 09fb:6001 Altera Blaster
Bus 003 Device 003: ID 046d:c077 Logitech, Inc. M105 Optical Mouse
Bus 003 Device 002: ID 413c:2107 Dell Computer Corp.
Bus 003 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

说明 /dev/bus/usb/003/013 这个文件现在就是我们的 Altera Blaster 设备

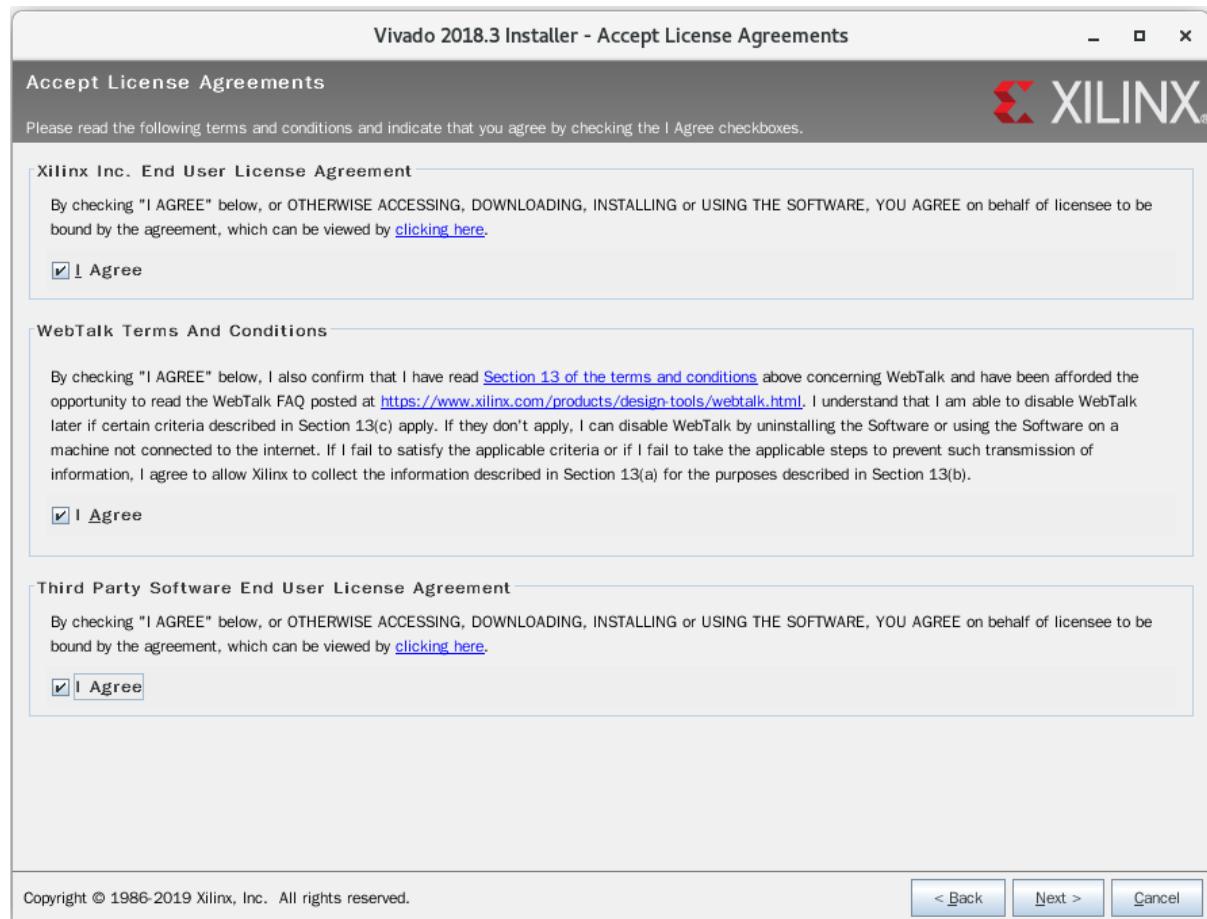
```
cd /dev/bus/usb/003
chmod 666 013
```

2.1.3 Vivado 安装

```
tar -zvxf Xilinx_Vivado_SDK_2018.3_1207_2324.tar.gz
cd Xilinx_Vivado_SDK_2018.3_1207_2324
./xsetup
```



点击 continue 选择不下载最新版本，然后点击 Next 进入下一步



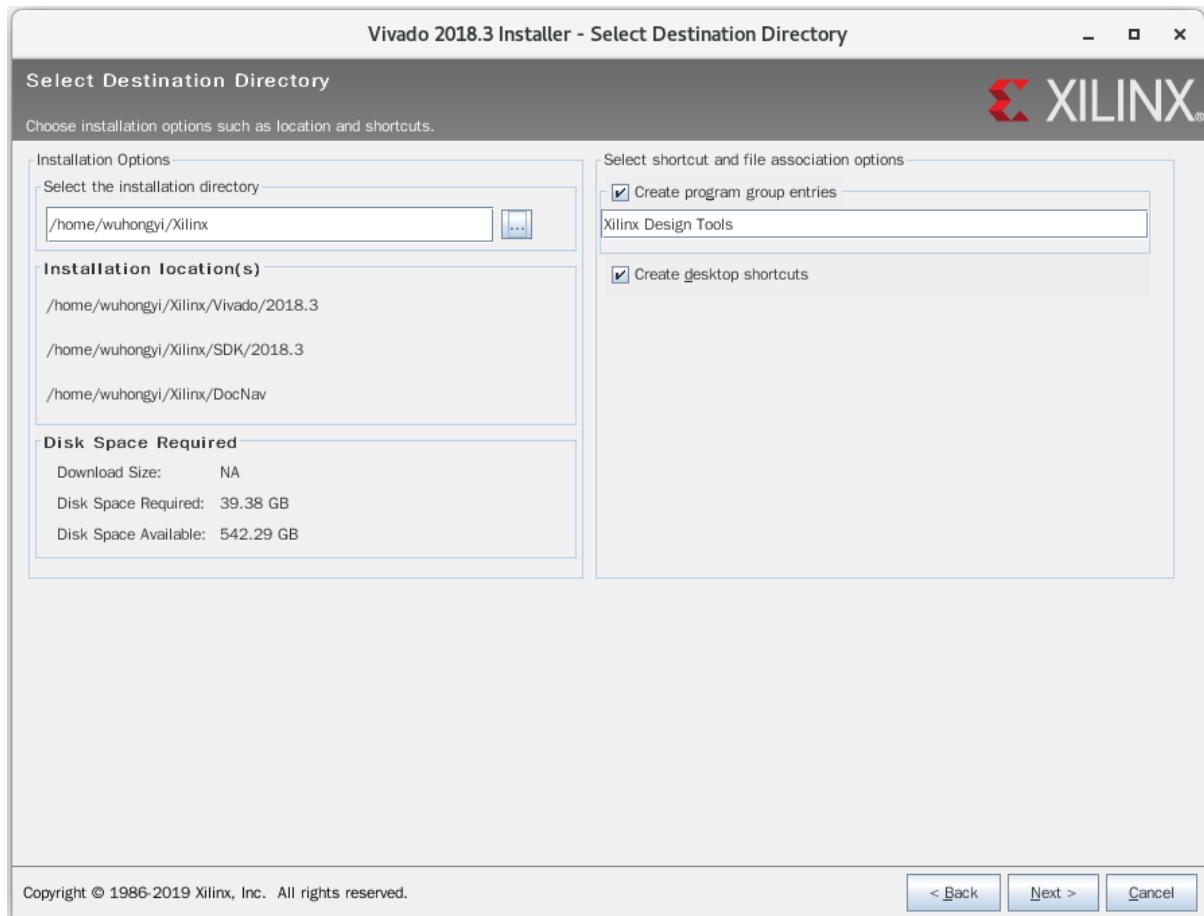
点击三个可选框，然后点击 Next 进入下一步



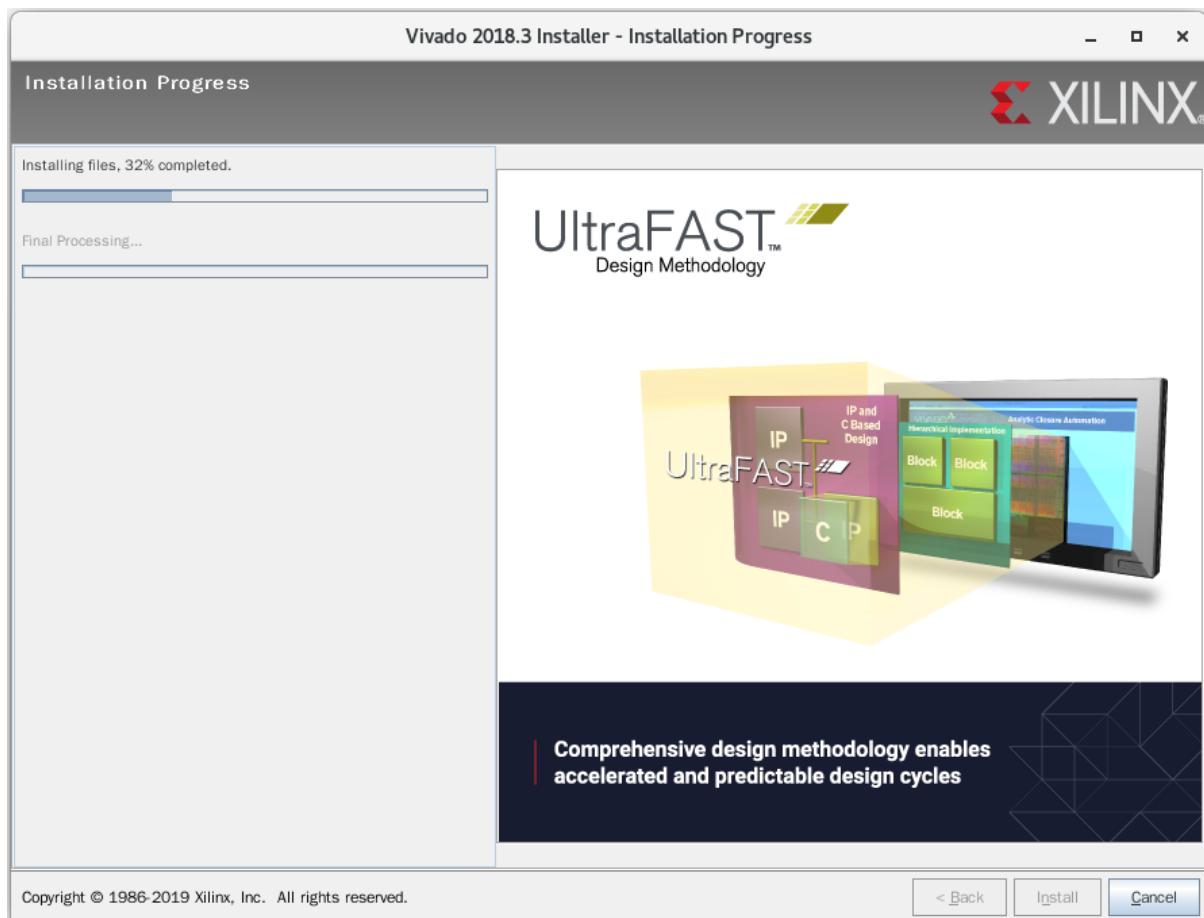
选择 Vinado HL Design Edition, 然后点击 Next 进入下一步



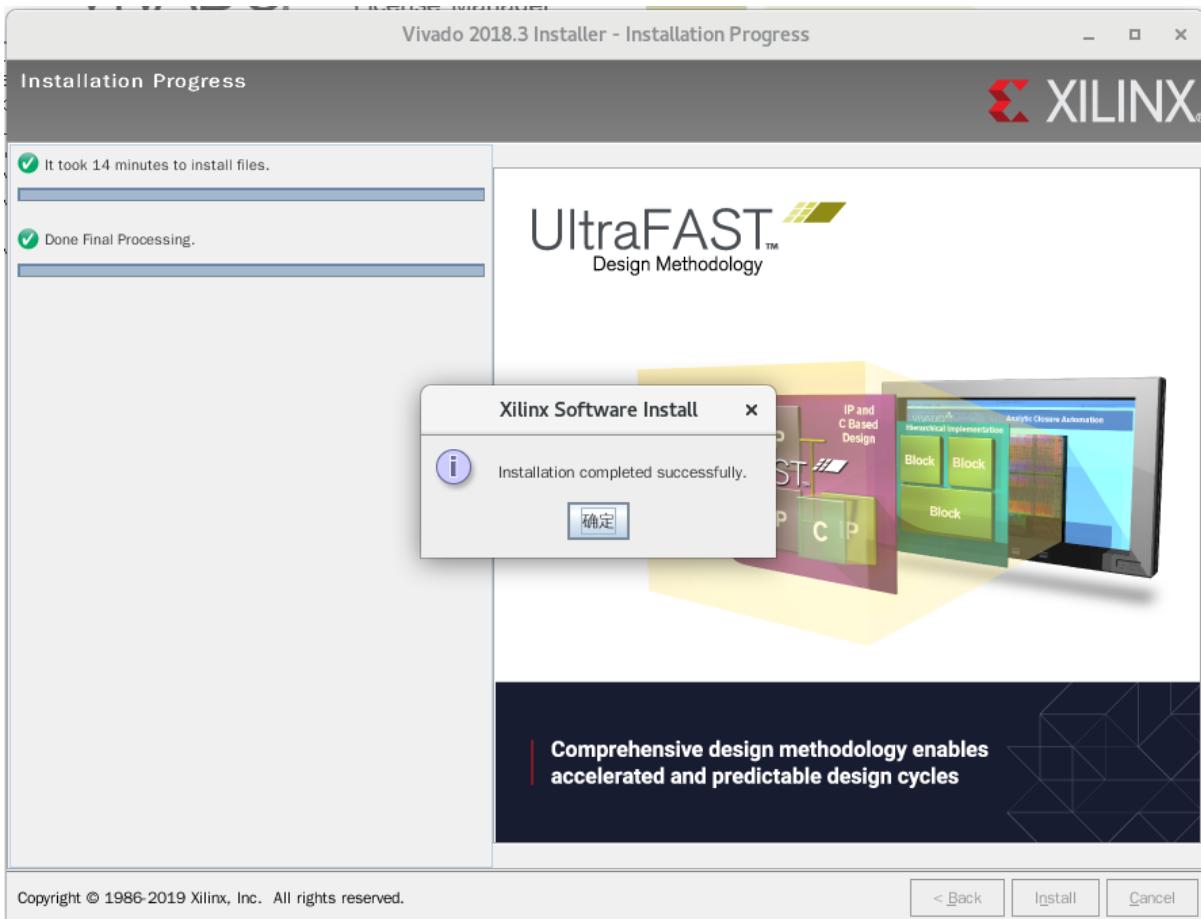
直接点击 Next 进入下一步



选择安装目录，这里我选择安装到 /home/wuhongyi/Xilinx，然后点击 Next 进入下一步

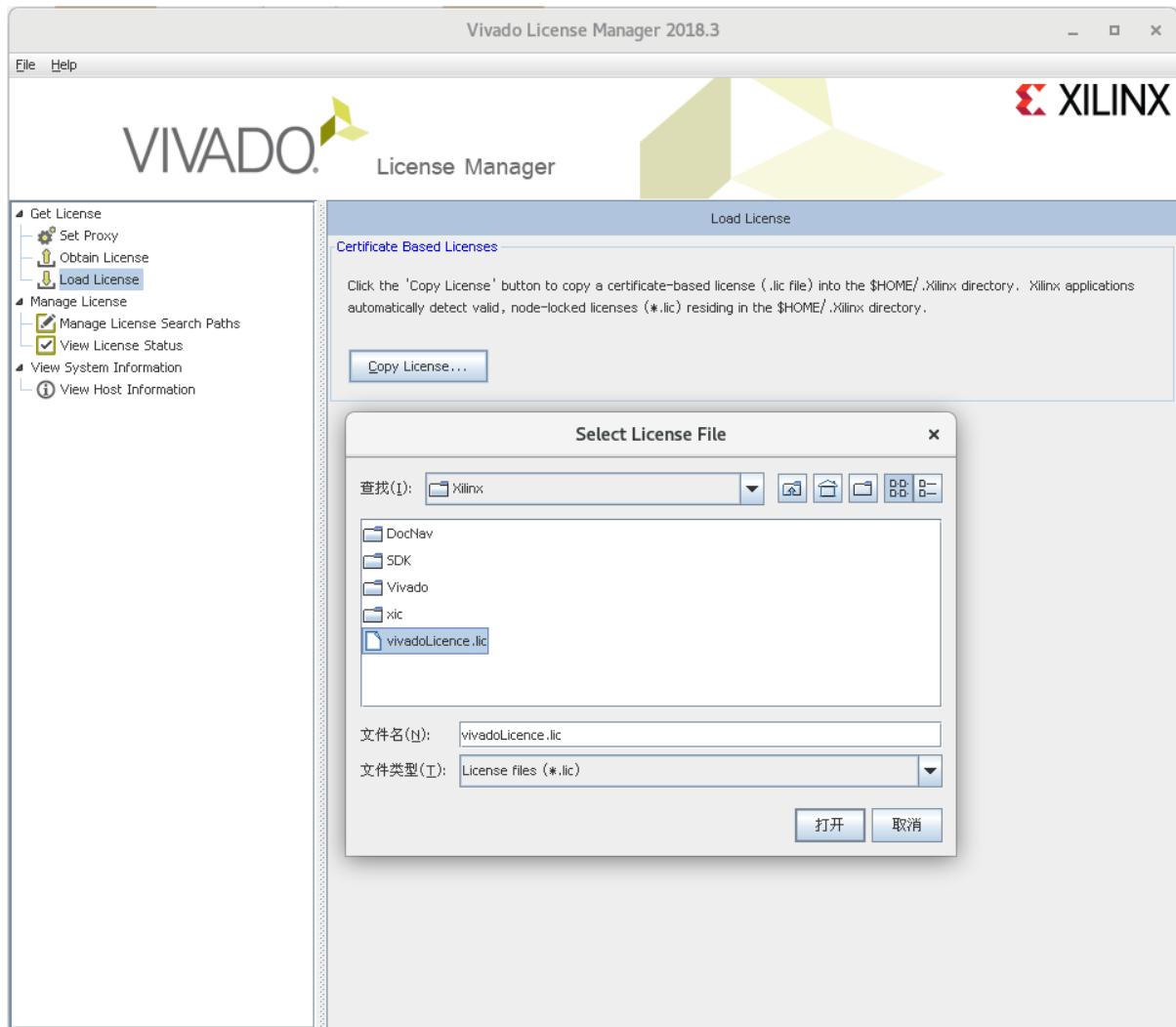


等待安装完成



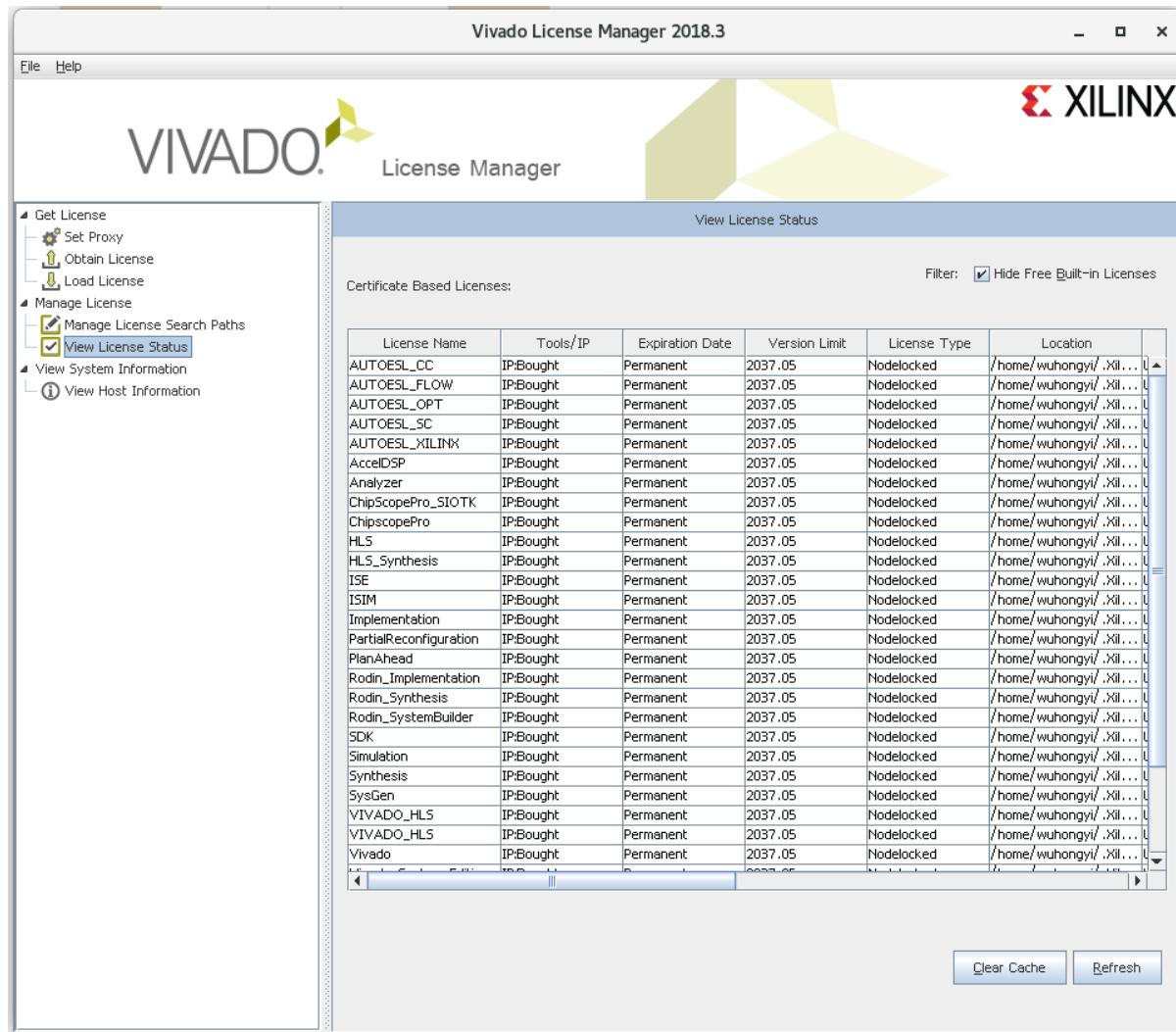
将 vivadoLicence.lic 文件复制到安装目录，这里为 /home/wuhongyi/Xilinx

安装完成之后会弹出以下界面



点击左上方的 Load License，选择我们的 vivadoLicence.lic 文件

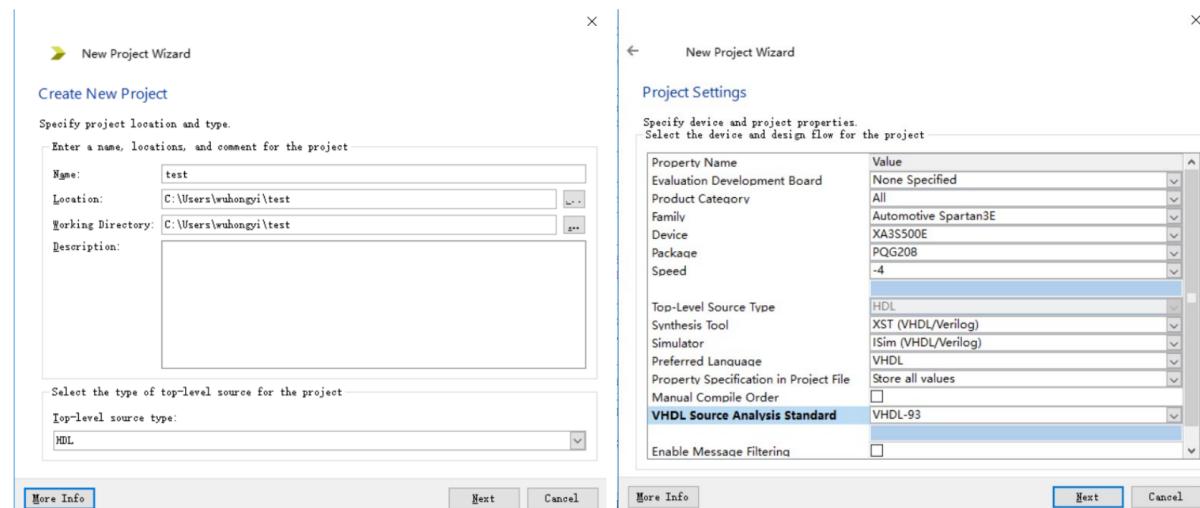
然后点击左上方的 View License Status 可查看破解的 IP 核



2.2 ISE 软件

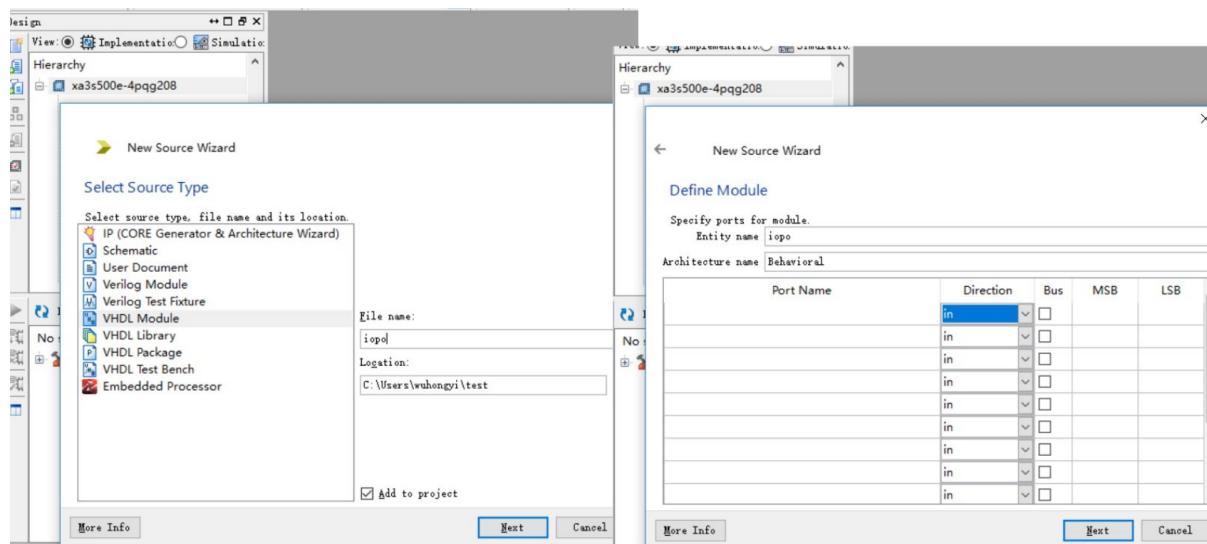
2.2.1 ISE

新建工程

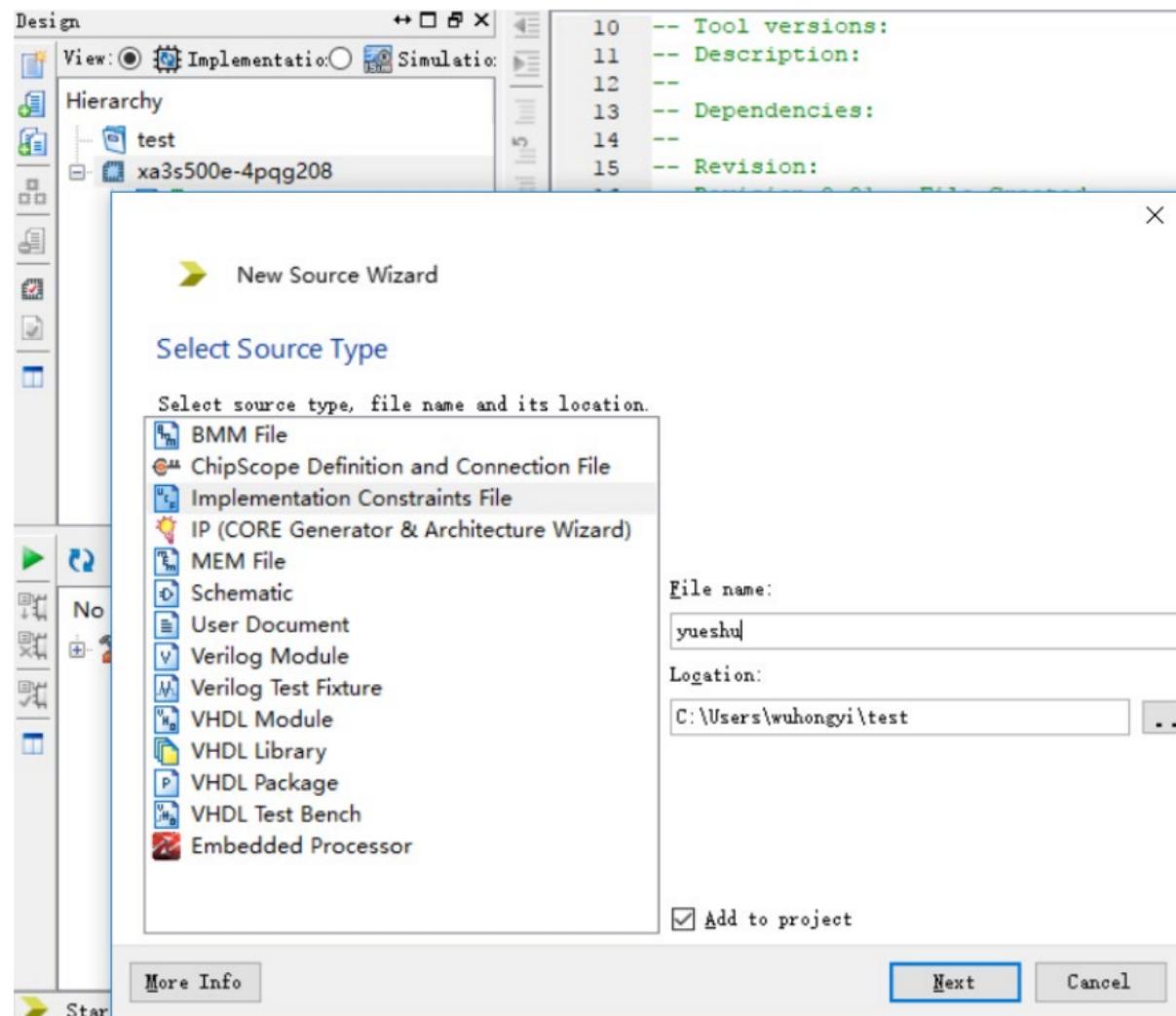


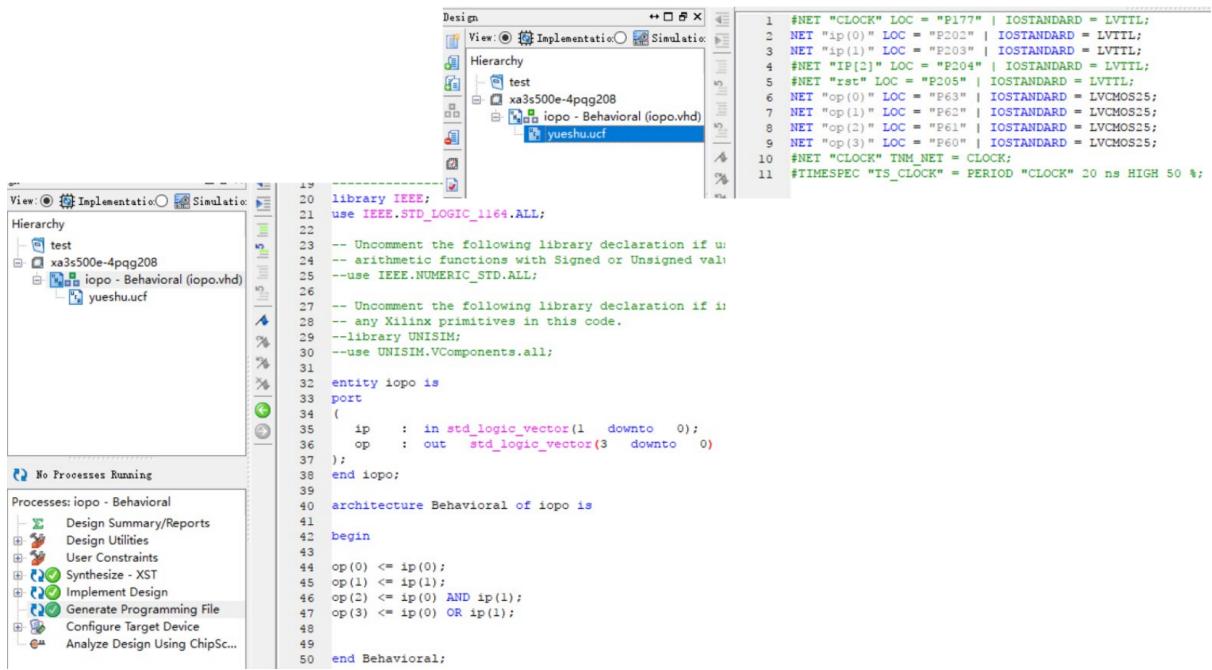
2.2. ISE 软件

右键 New Source 建立 Top Module



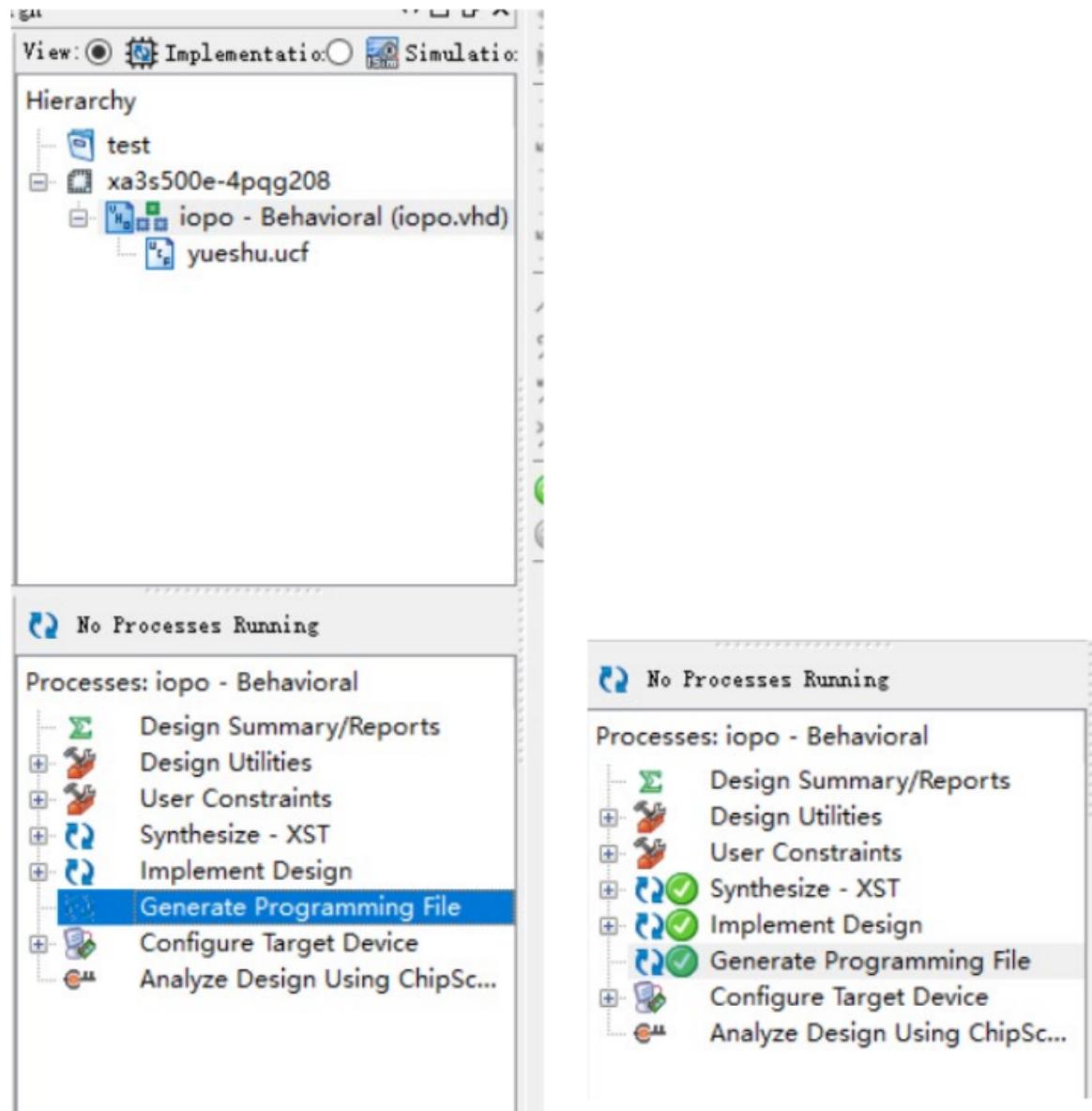
右键 New Source 建立约束文件



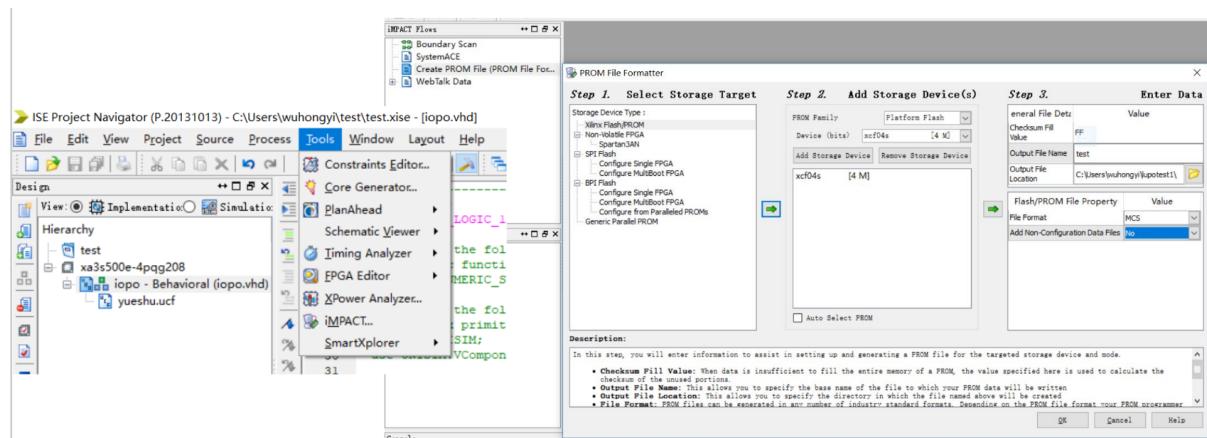


编译生成 xxx.bit 文件

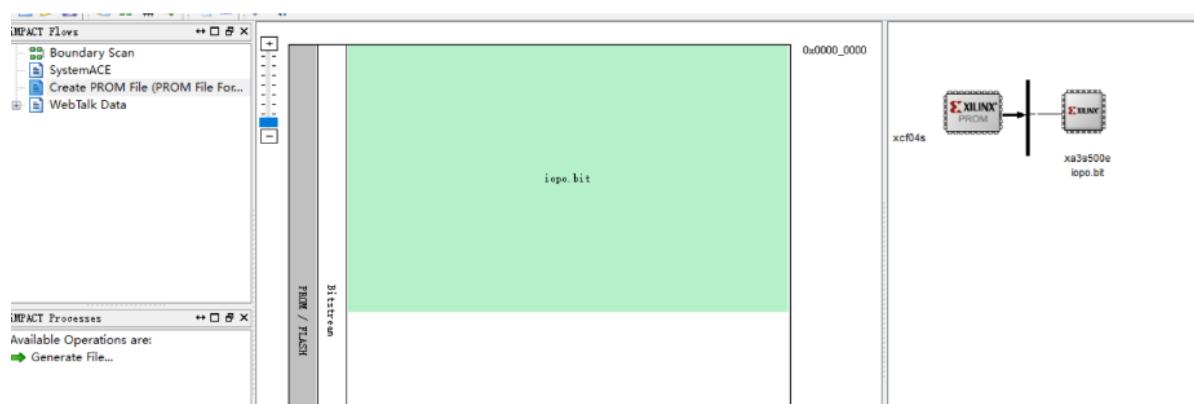
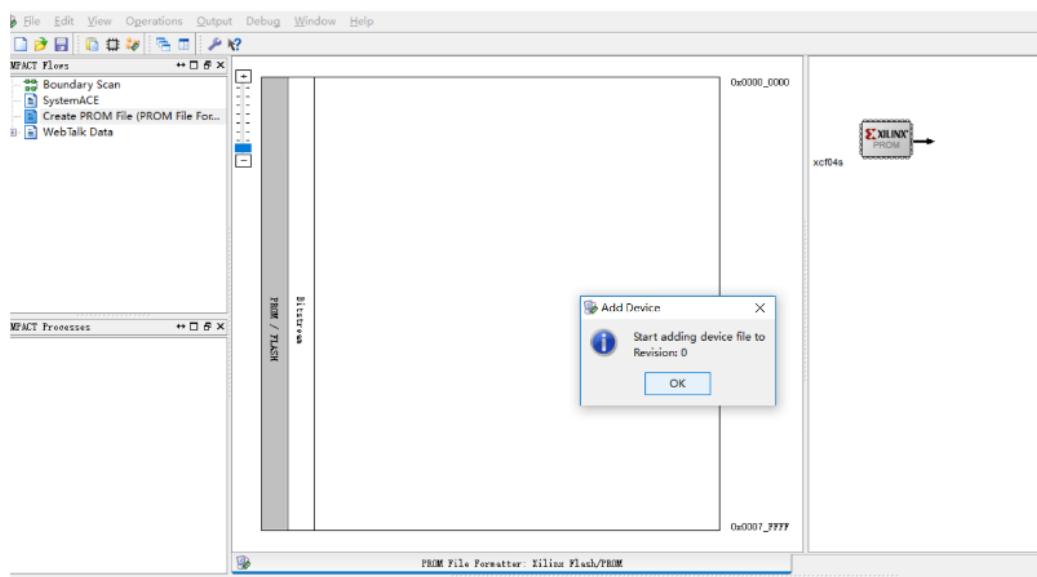
- 选中主程序
- 在下方的 Generate Programming File
- 右键选择 Rerun All
- 等待完成或者报错



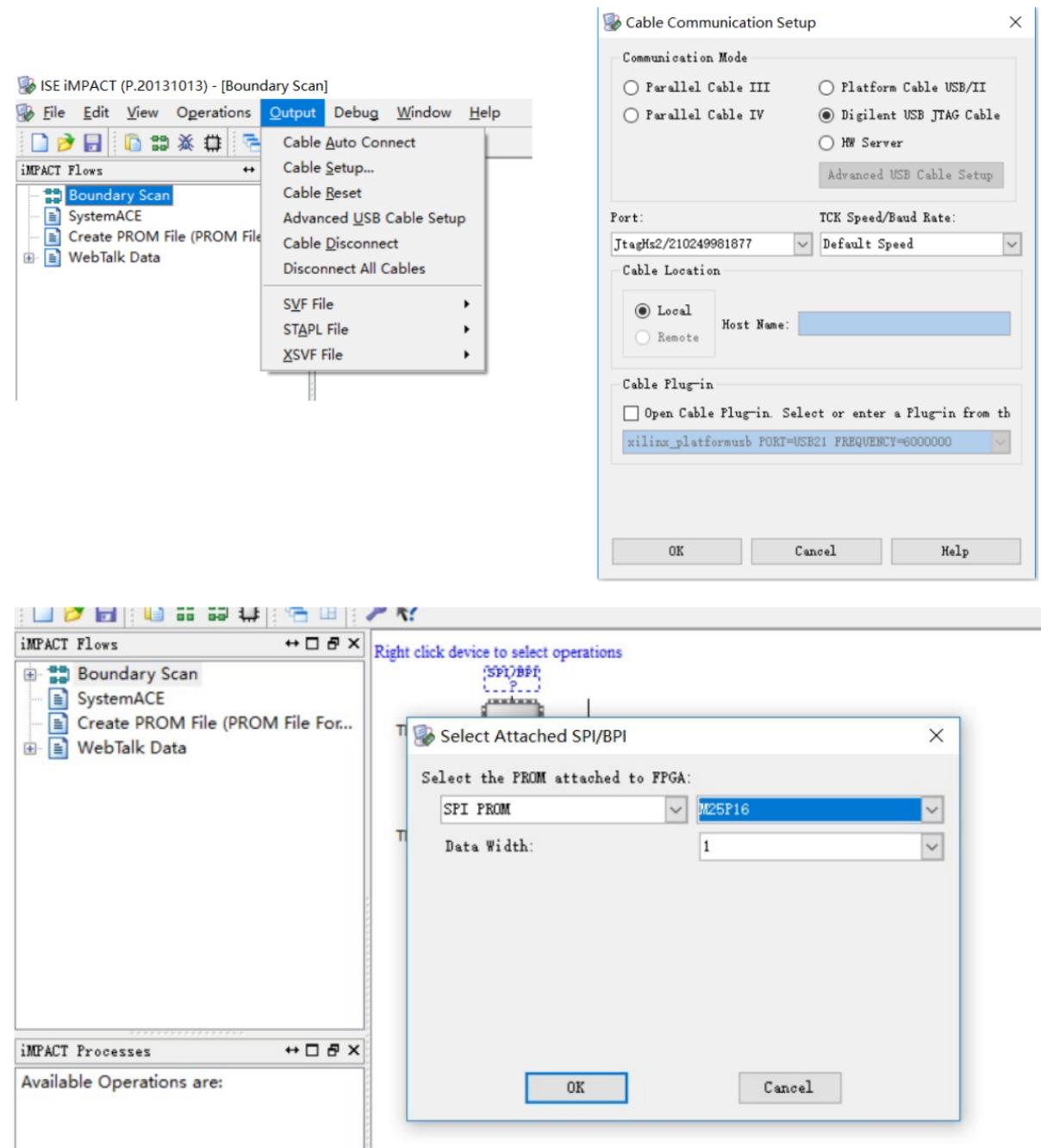
刷新固件

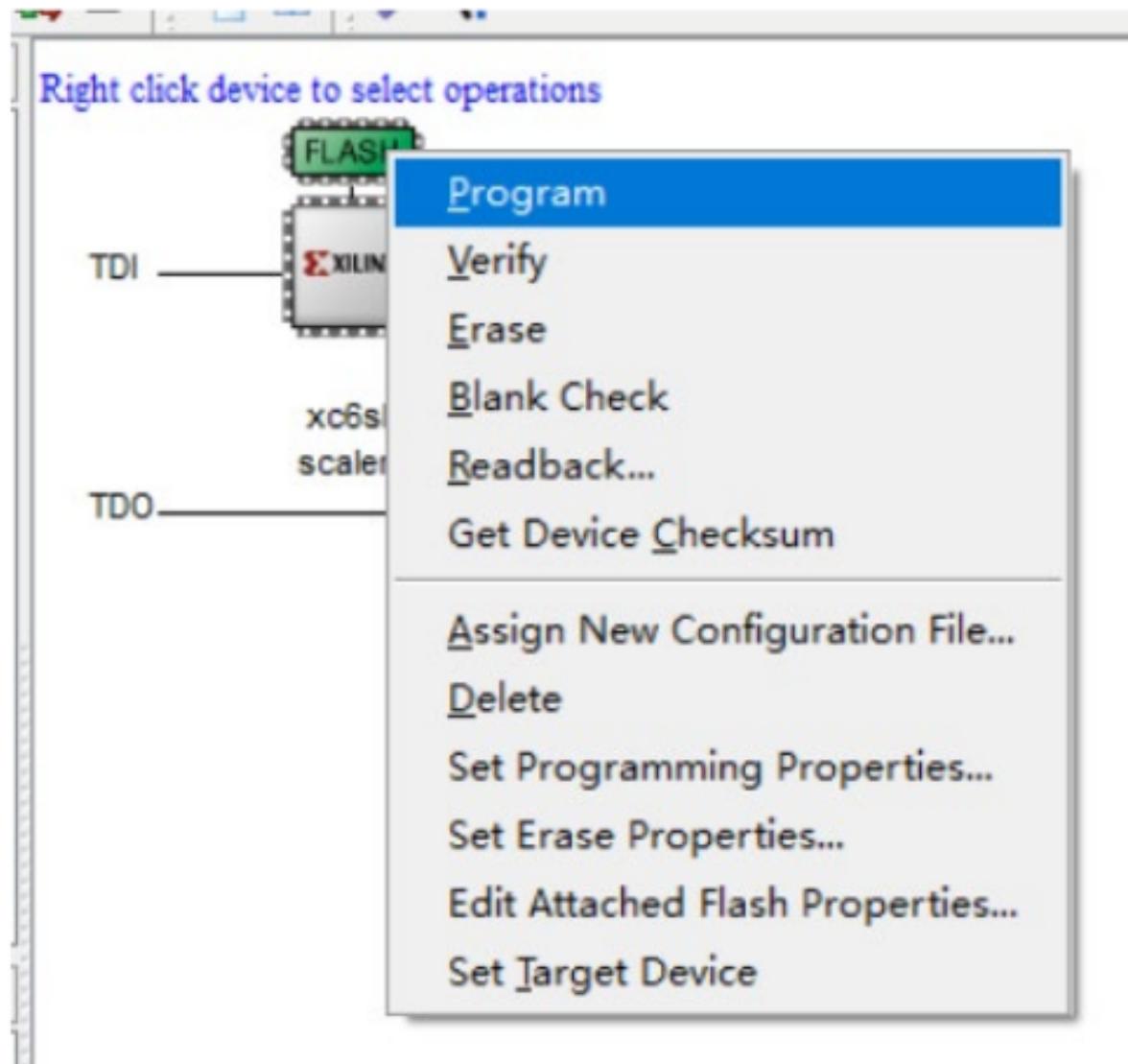


- Add a device? -> OK
- 选择之前生成的 xxx.bit 文件
- Any other device? -> No



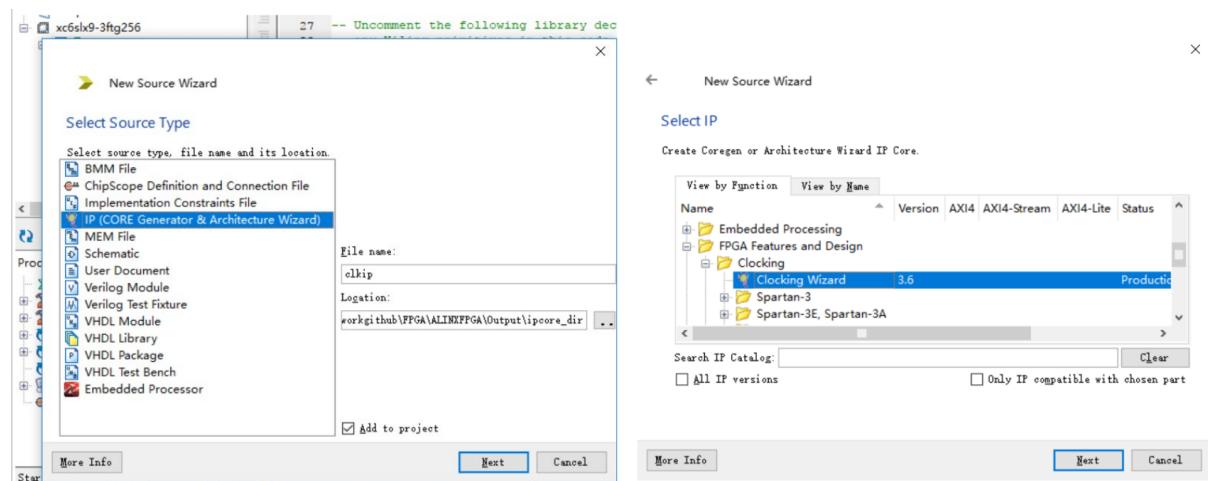
MSC

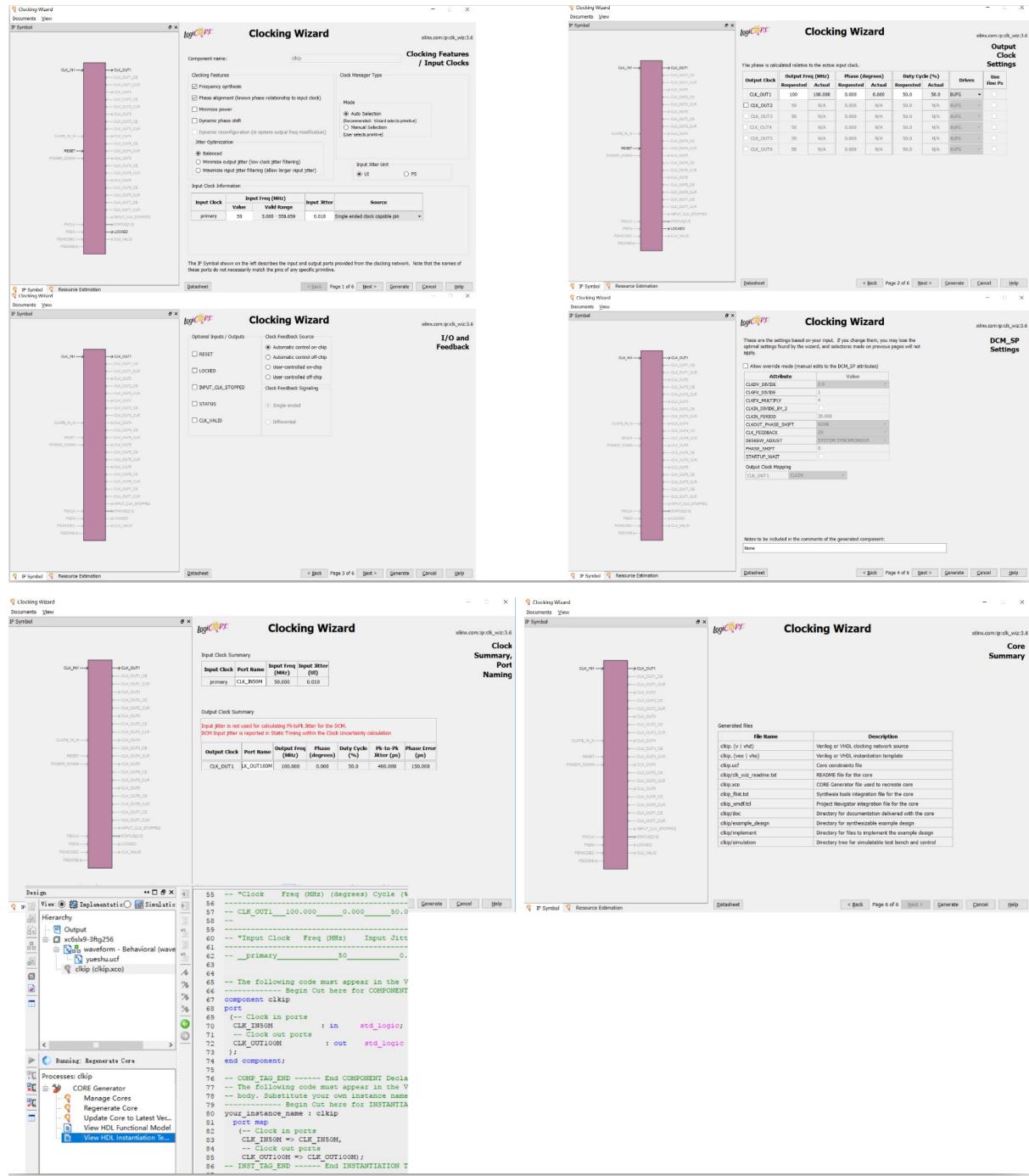




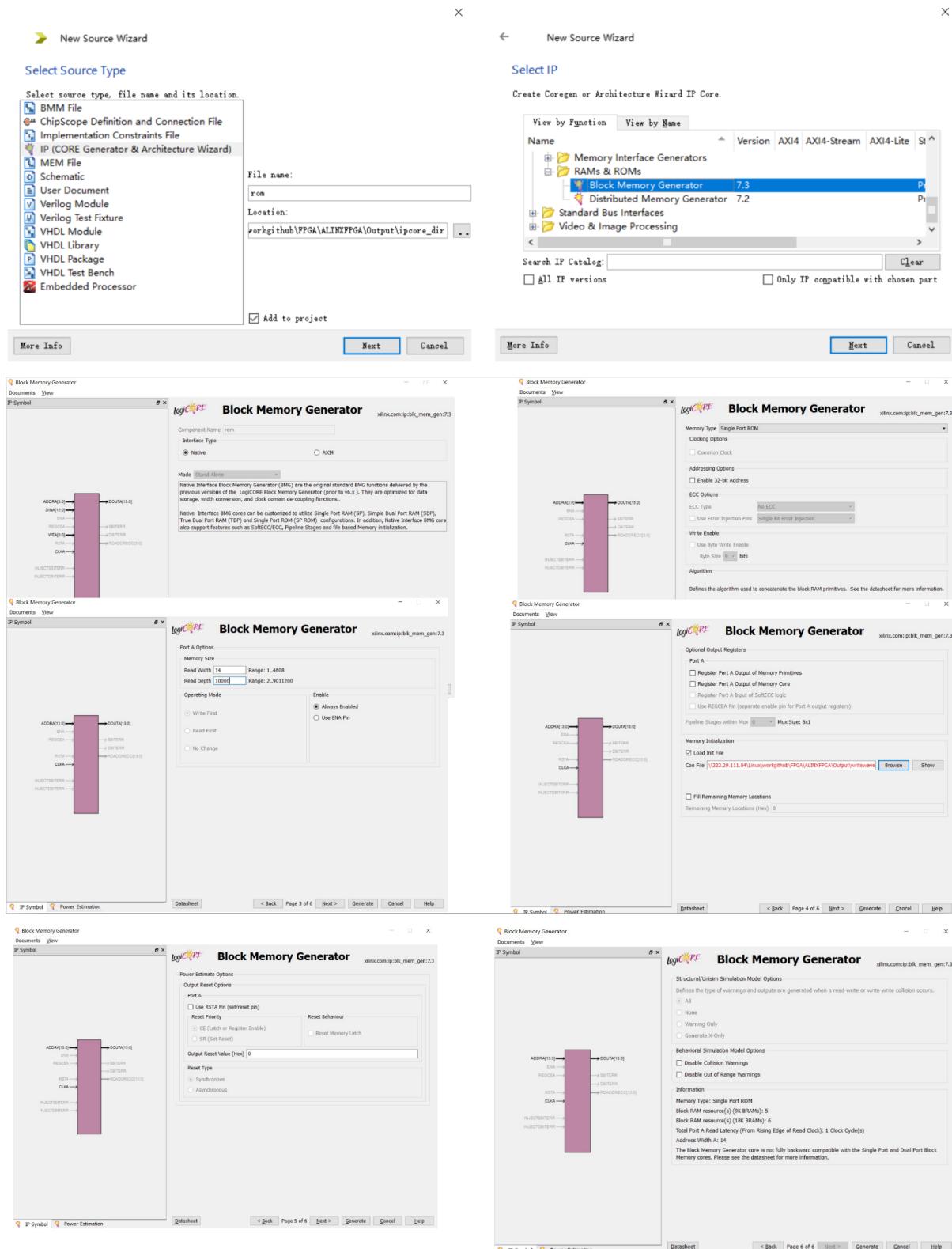
IP 核

CLOCK

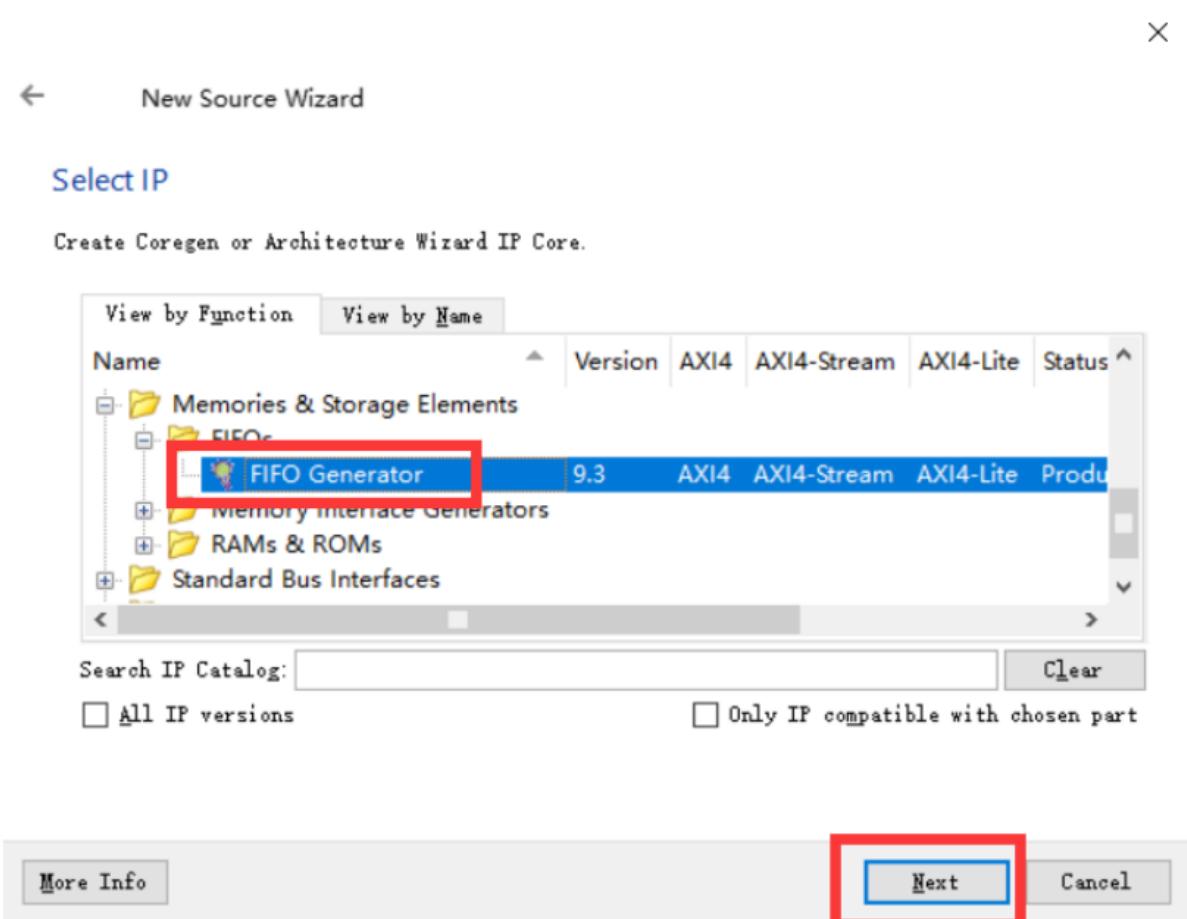




ROM



FIFO



http://www.eefocus.com/guoke1993102/blog/15-06/313183_36284.html

2.2.2 ISim

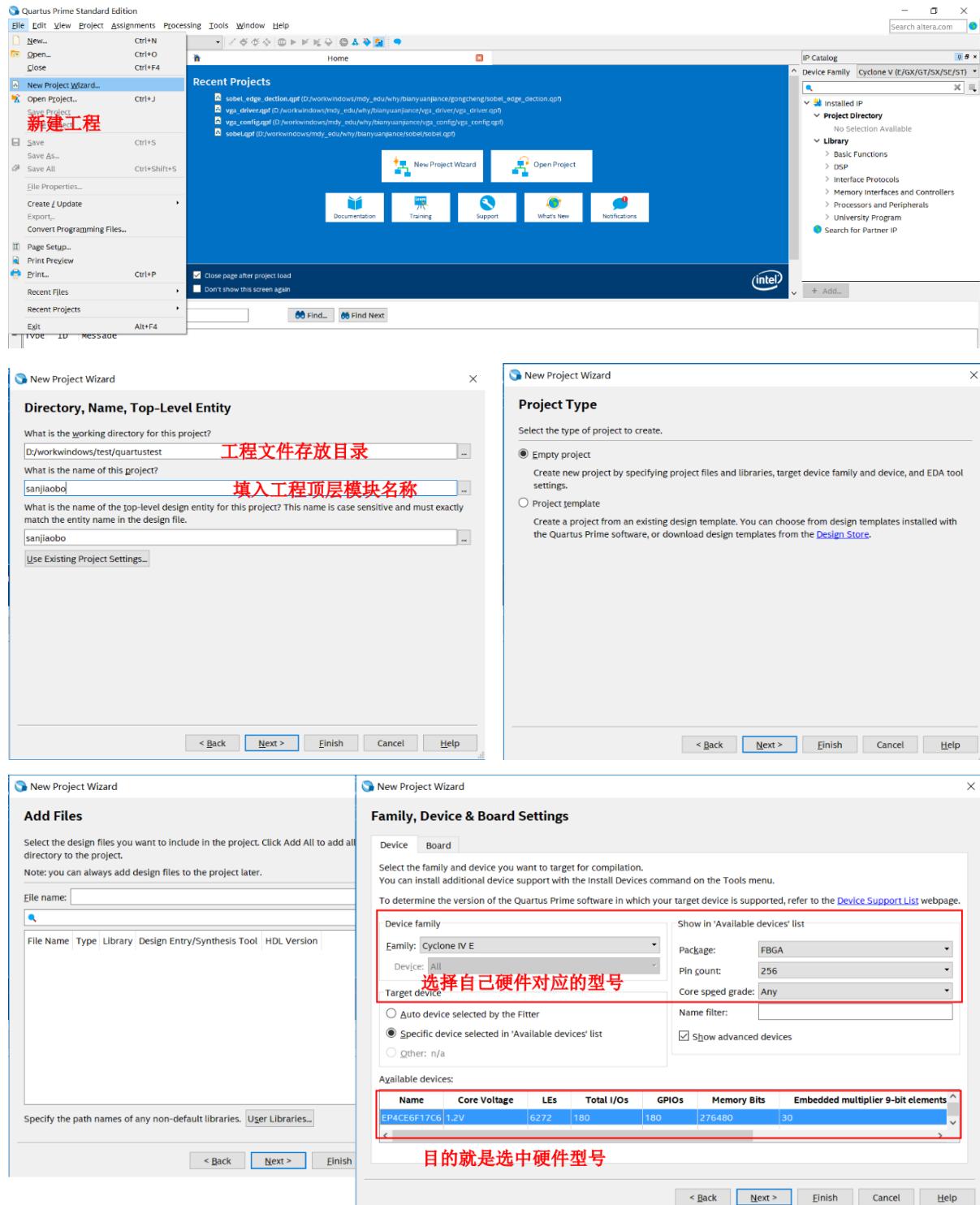
TODO

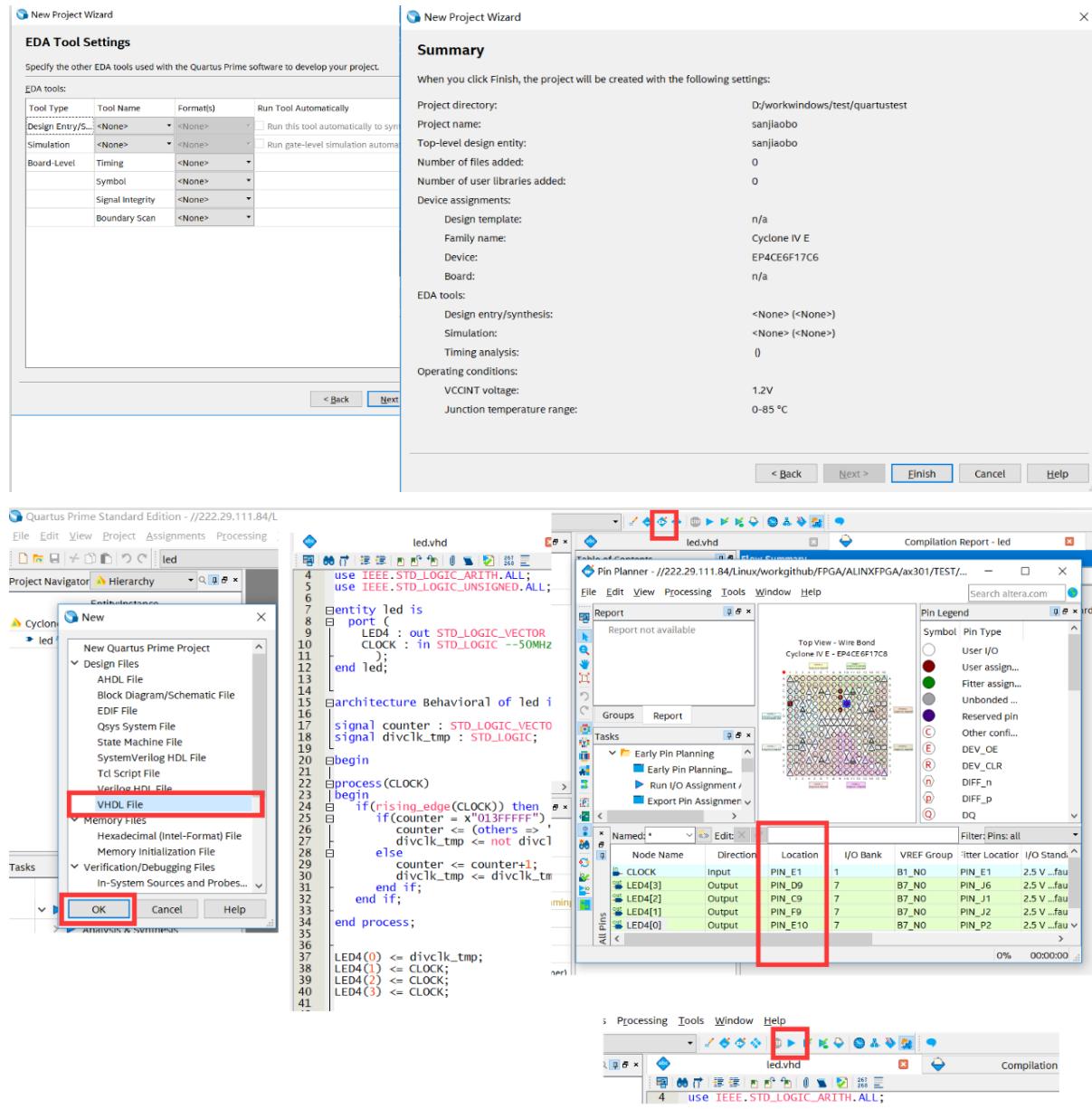
2.2.3 chipscope

TODO

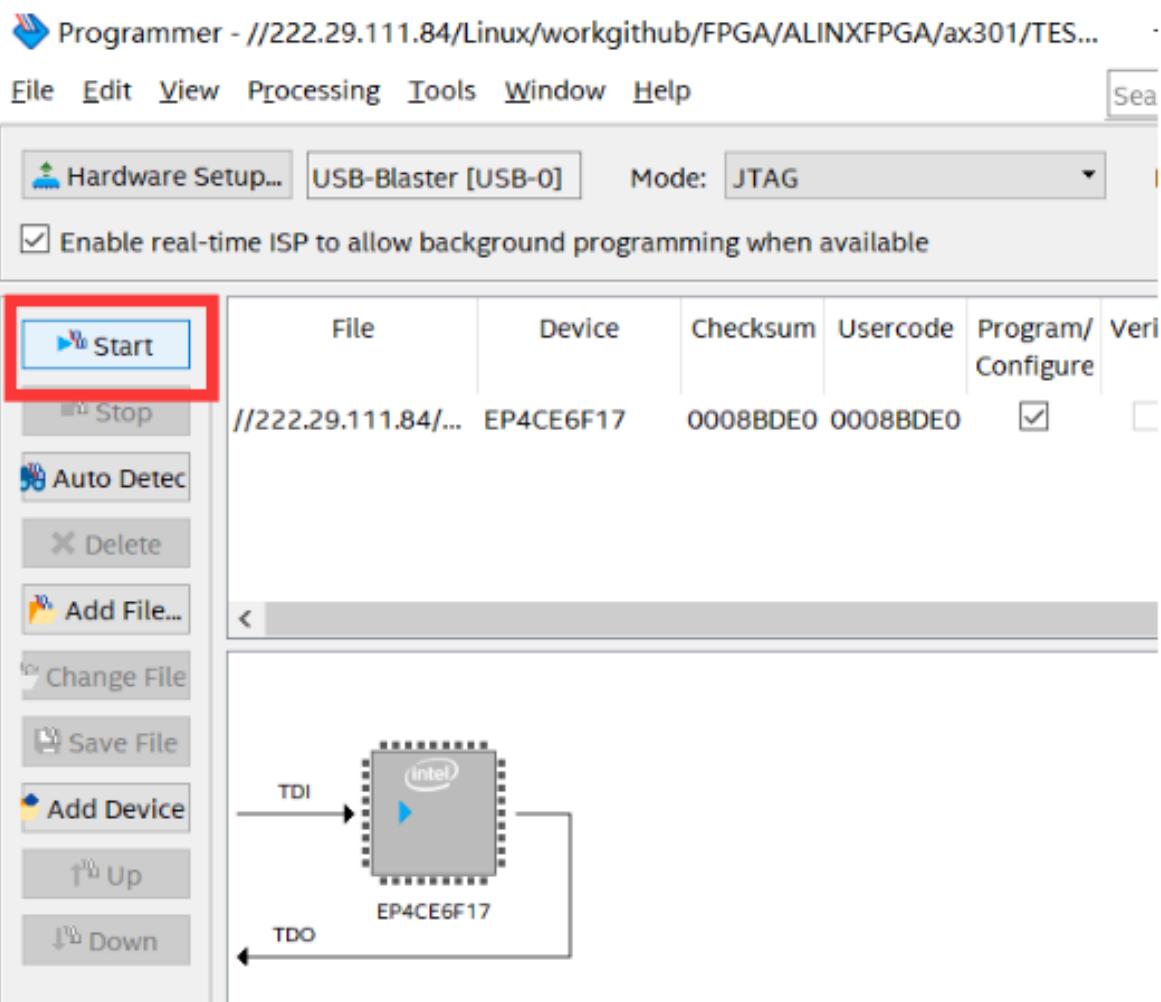
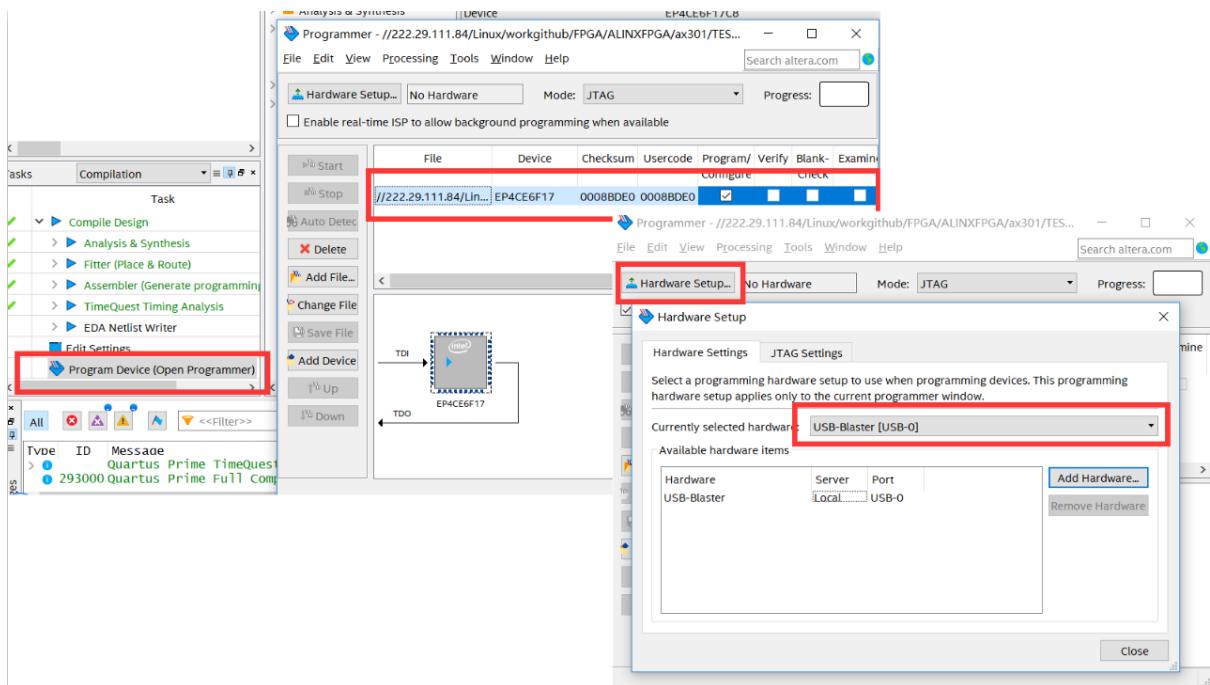
2.3 Altera 软件

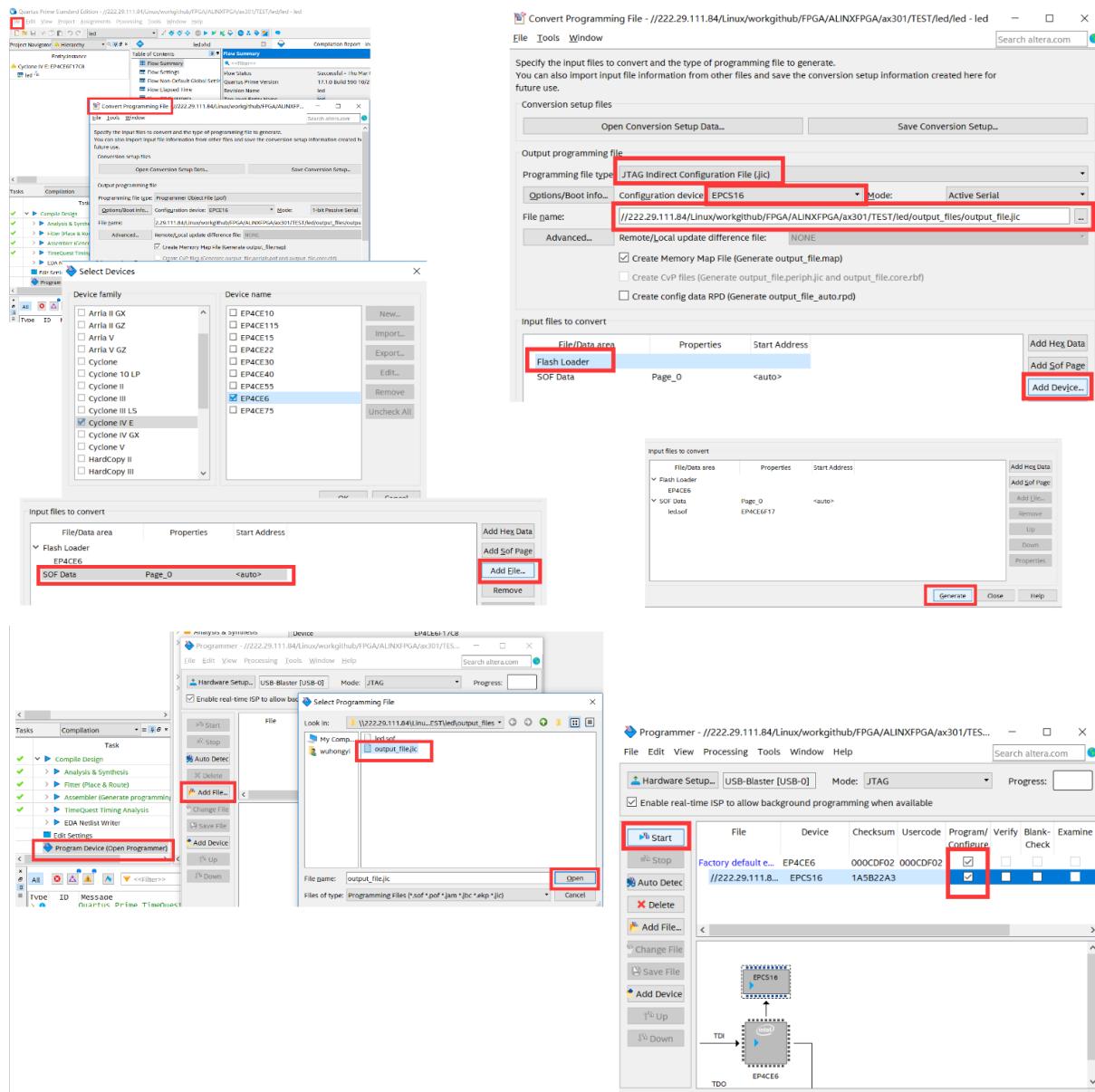
2.3.1 Quartus





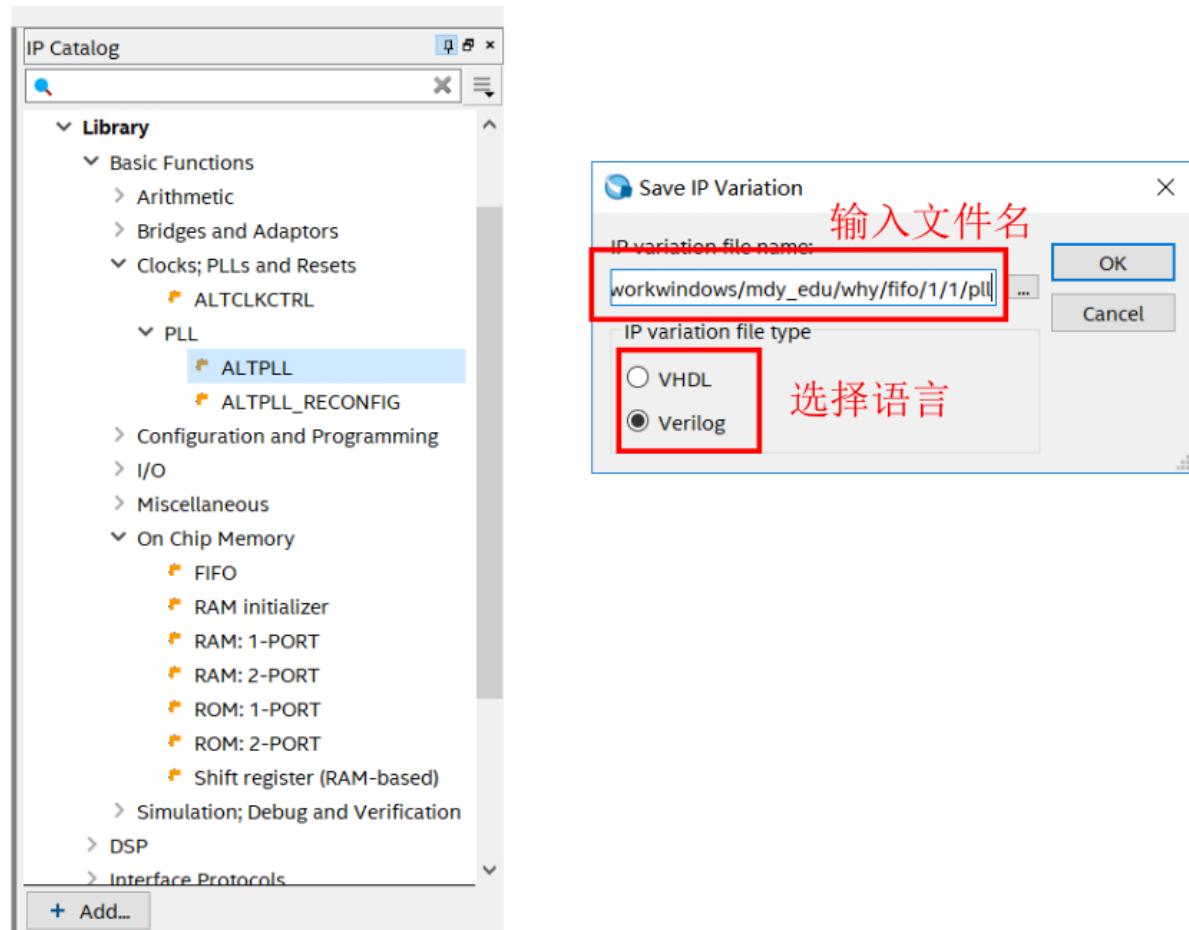
刷固件方式

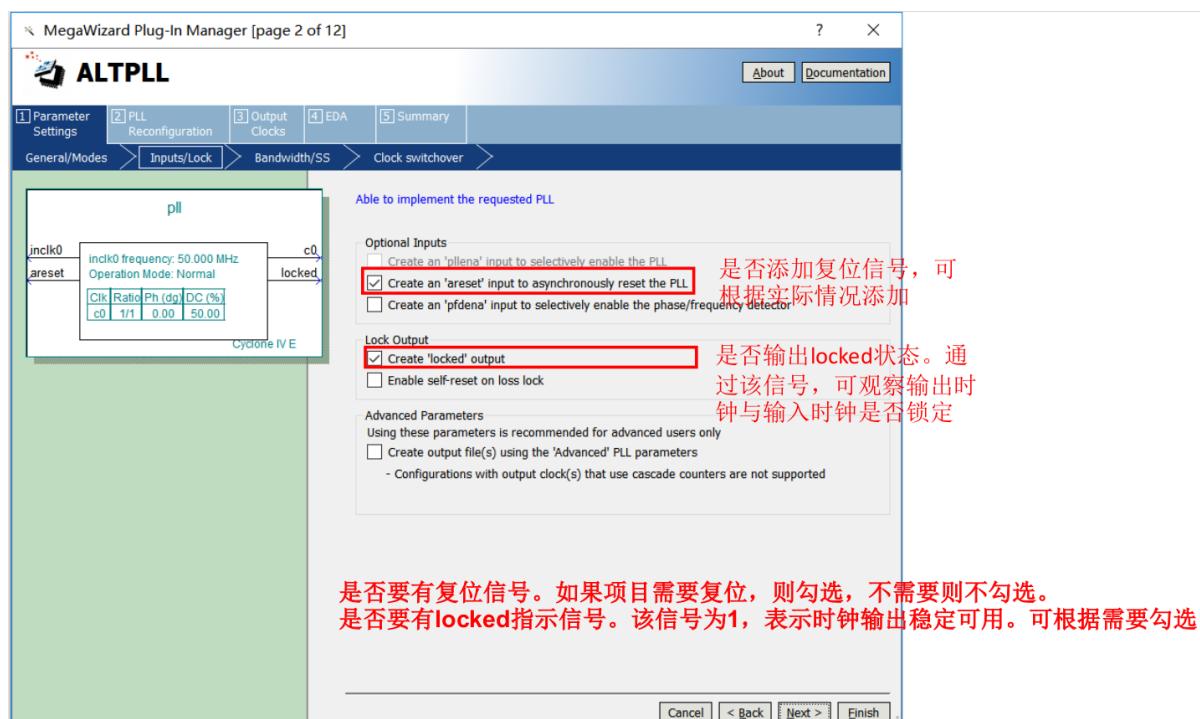
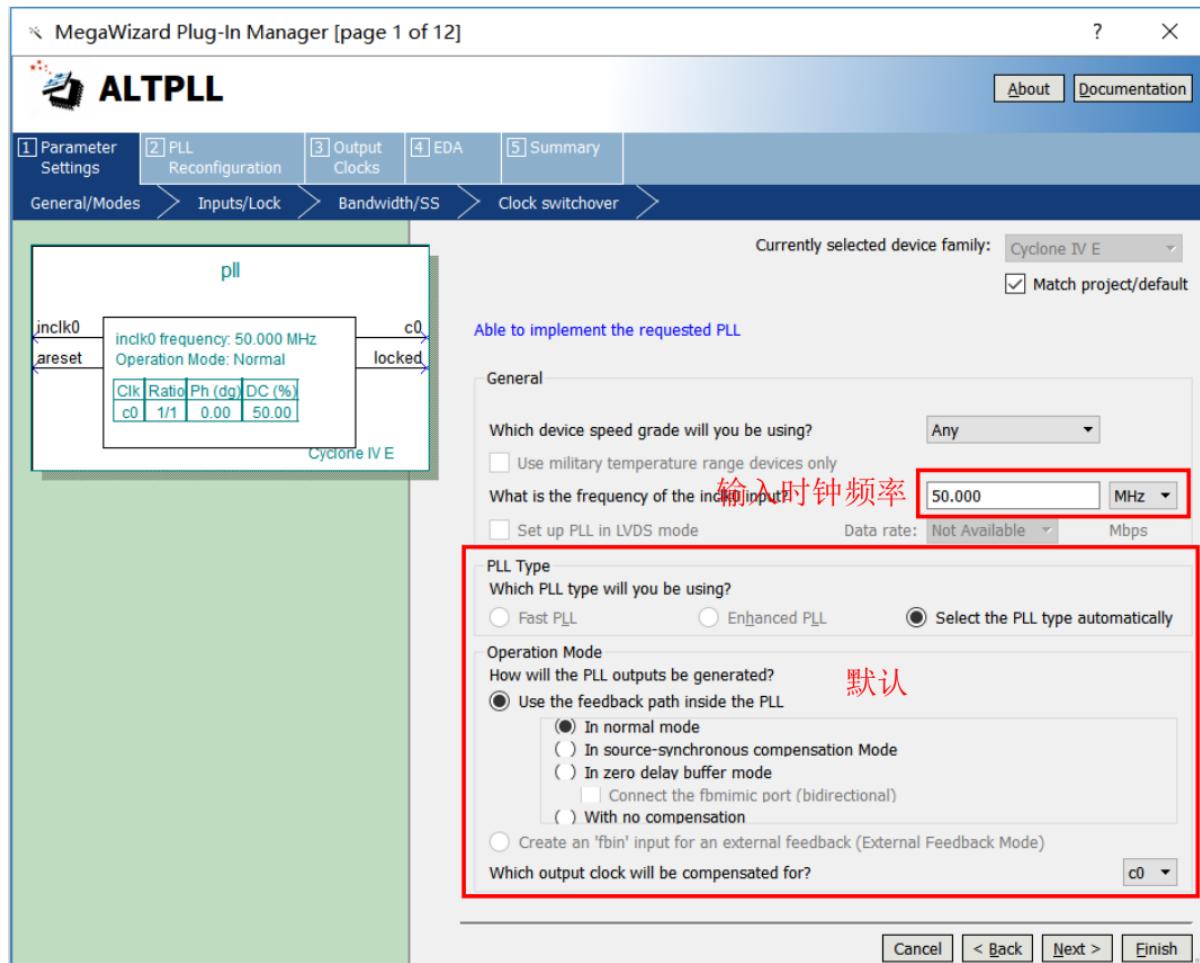


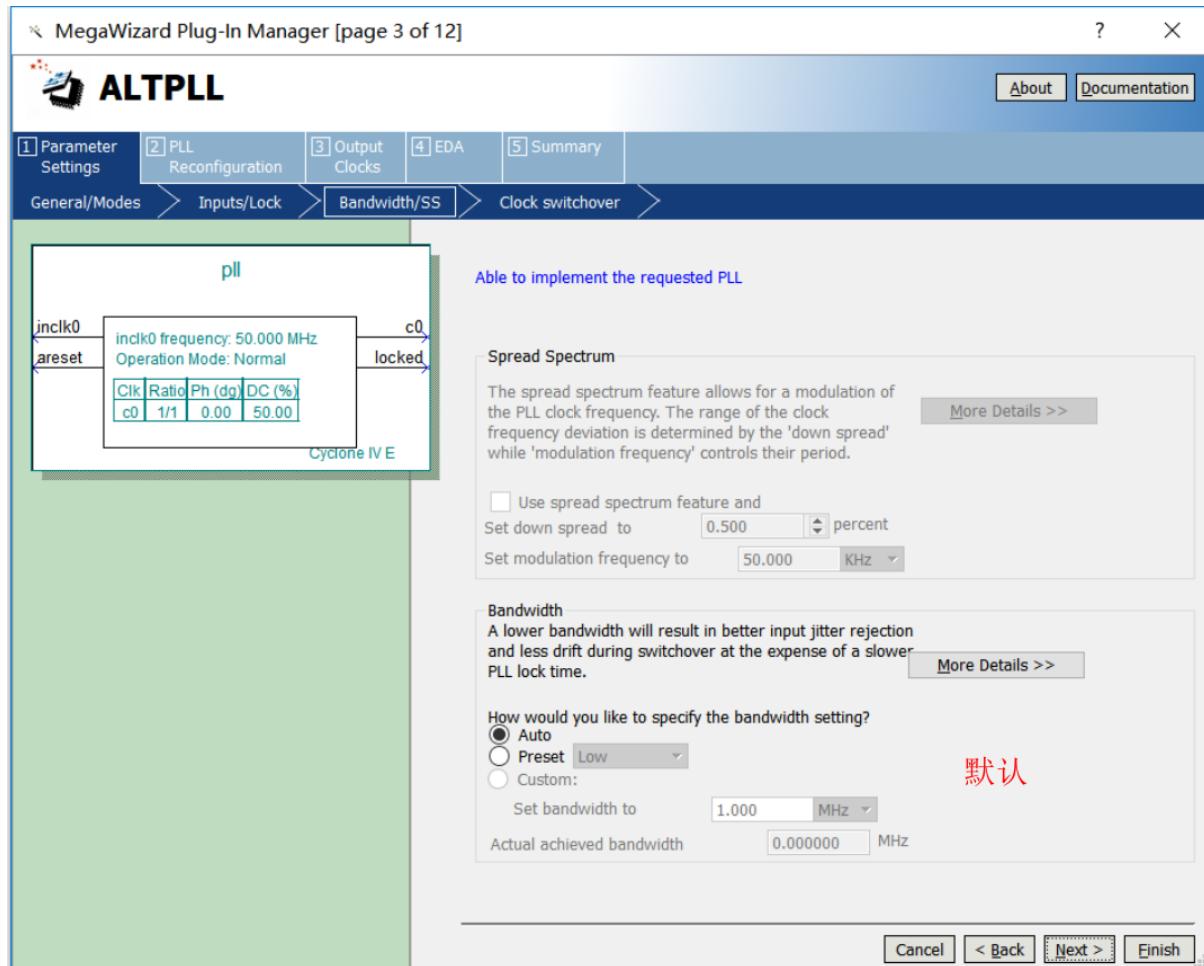


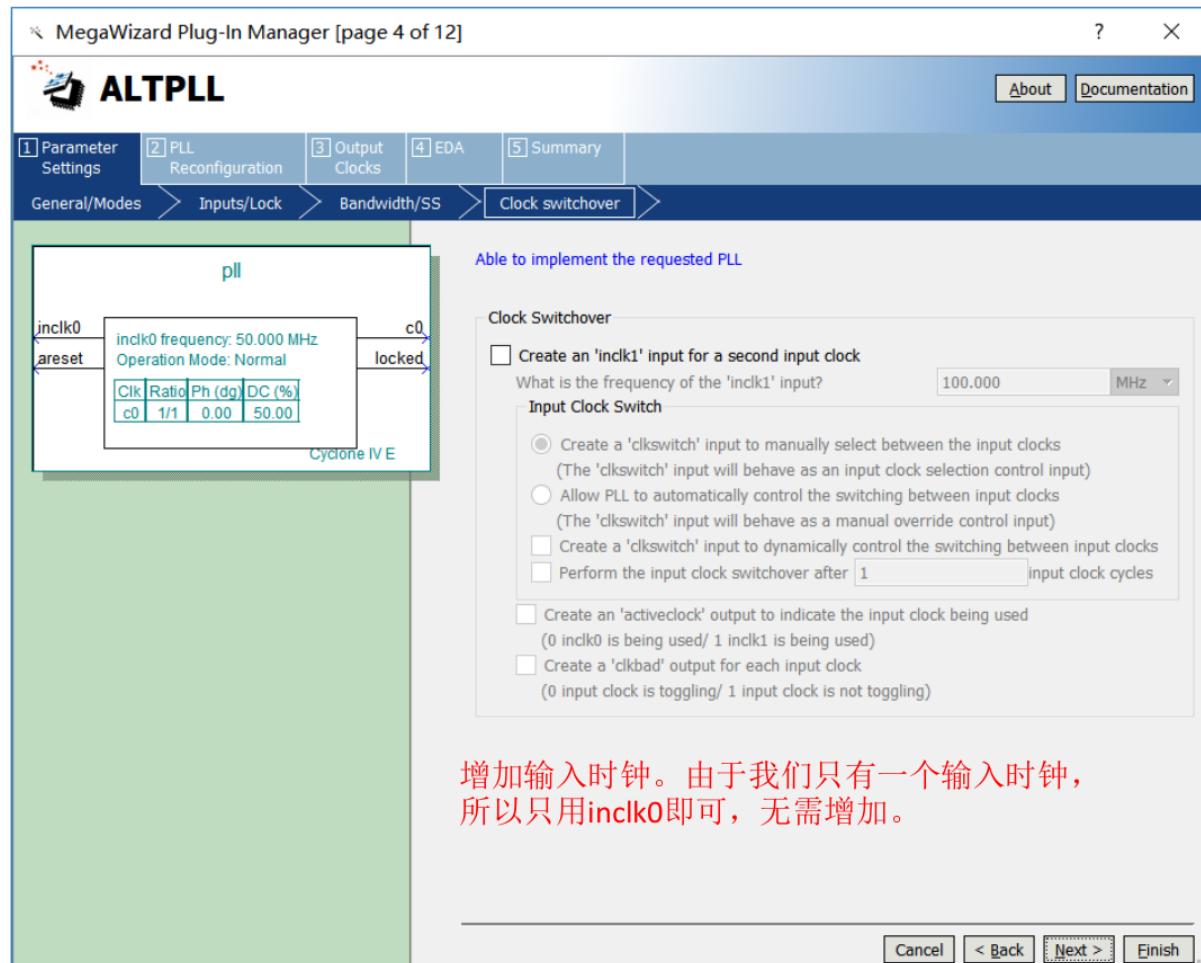
IP 核

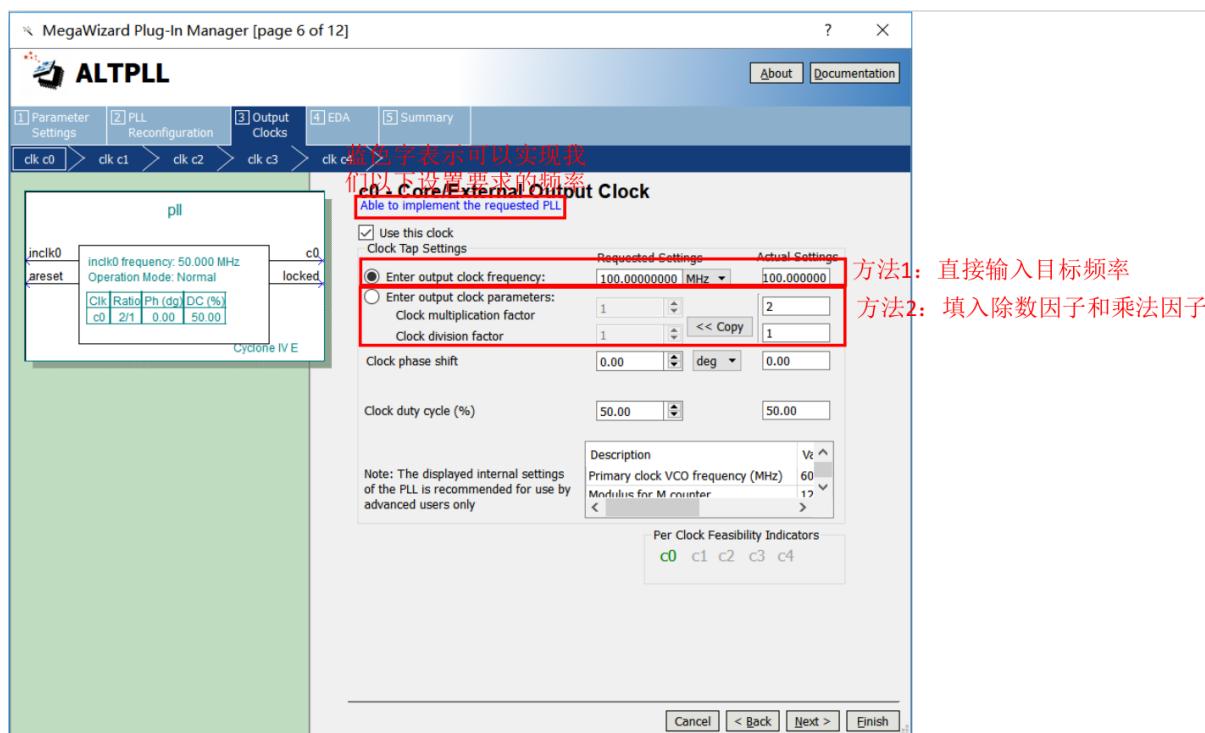
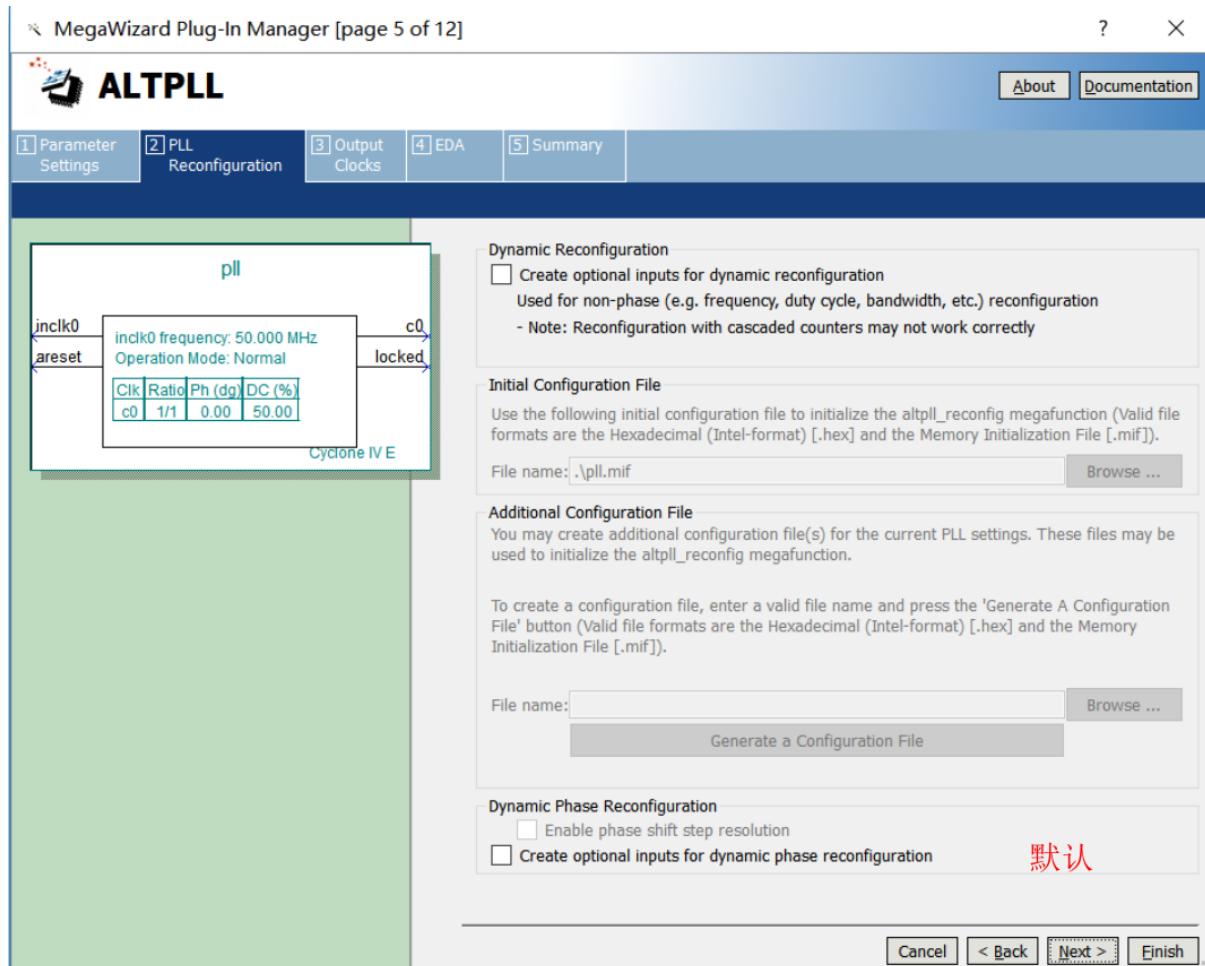
PLL

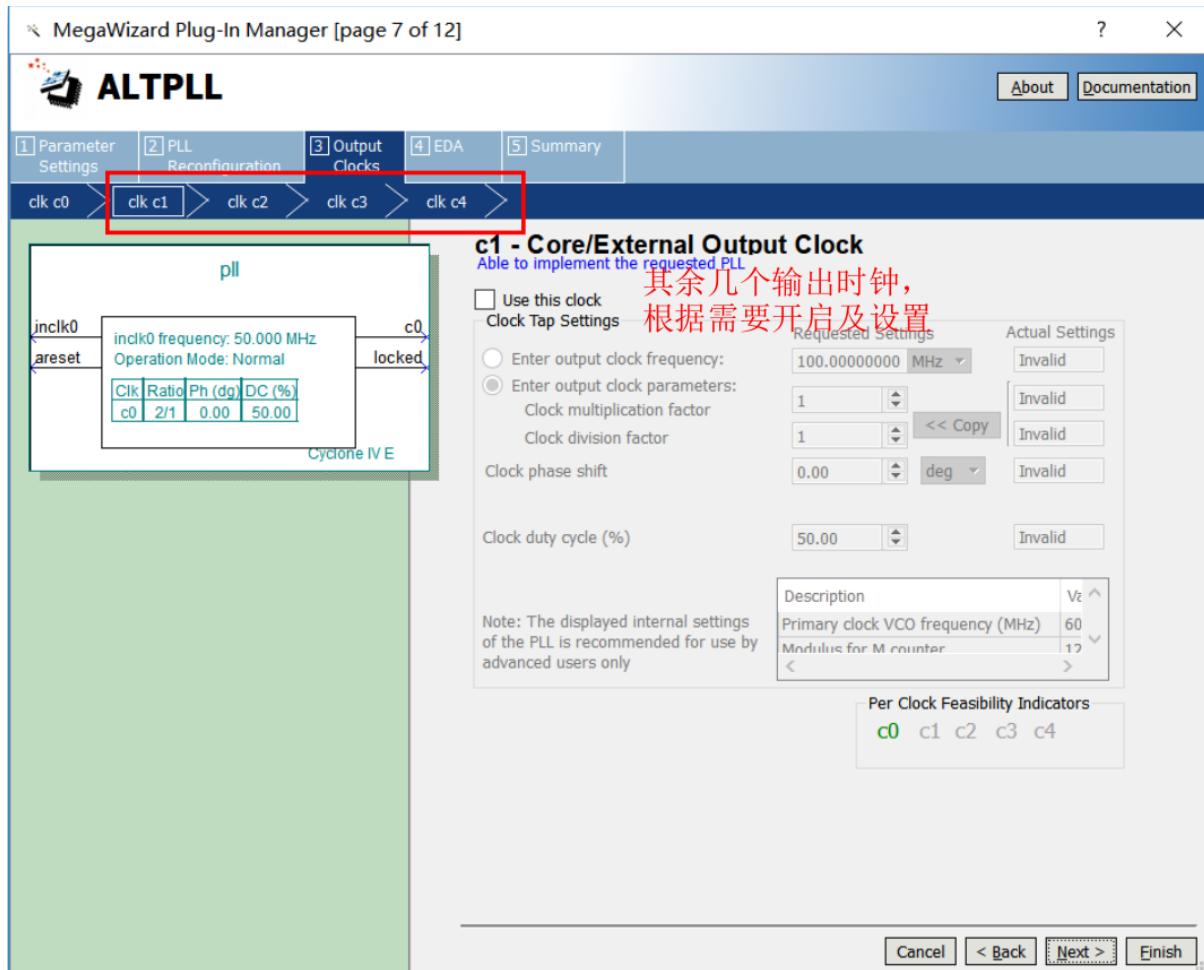


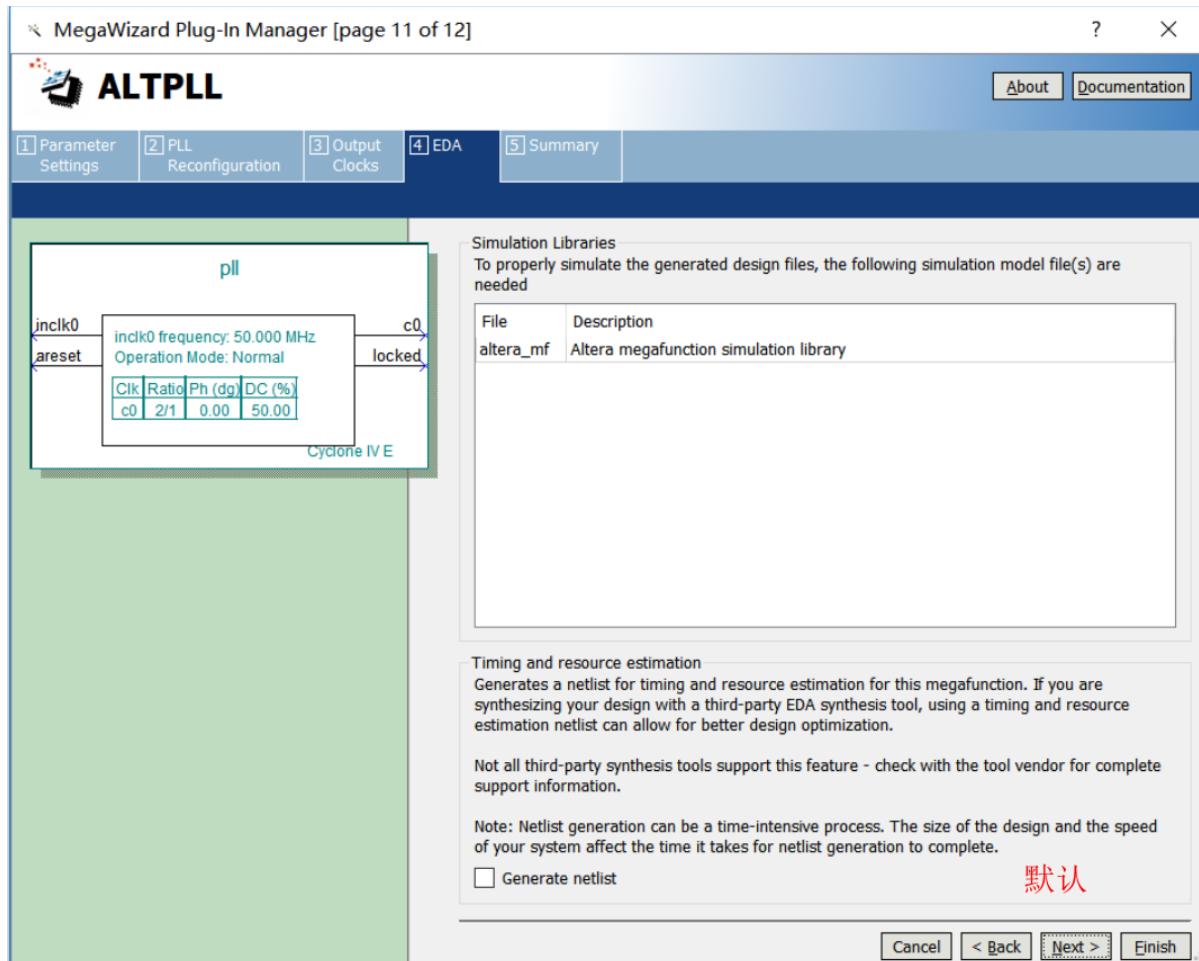


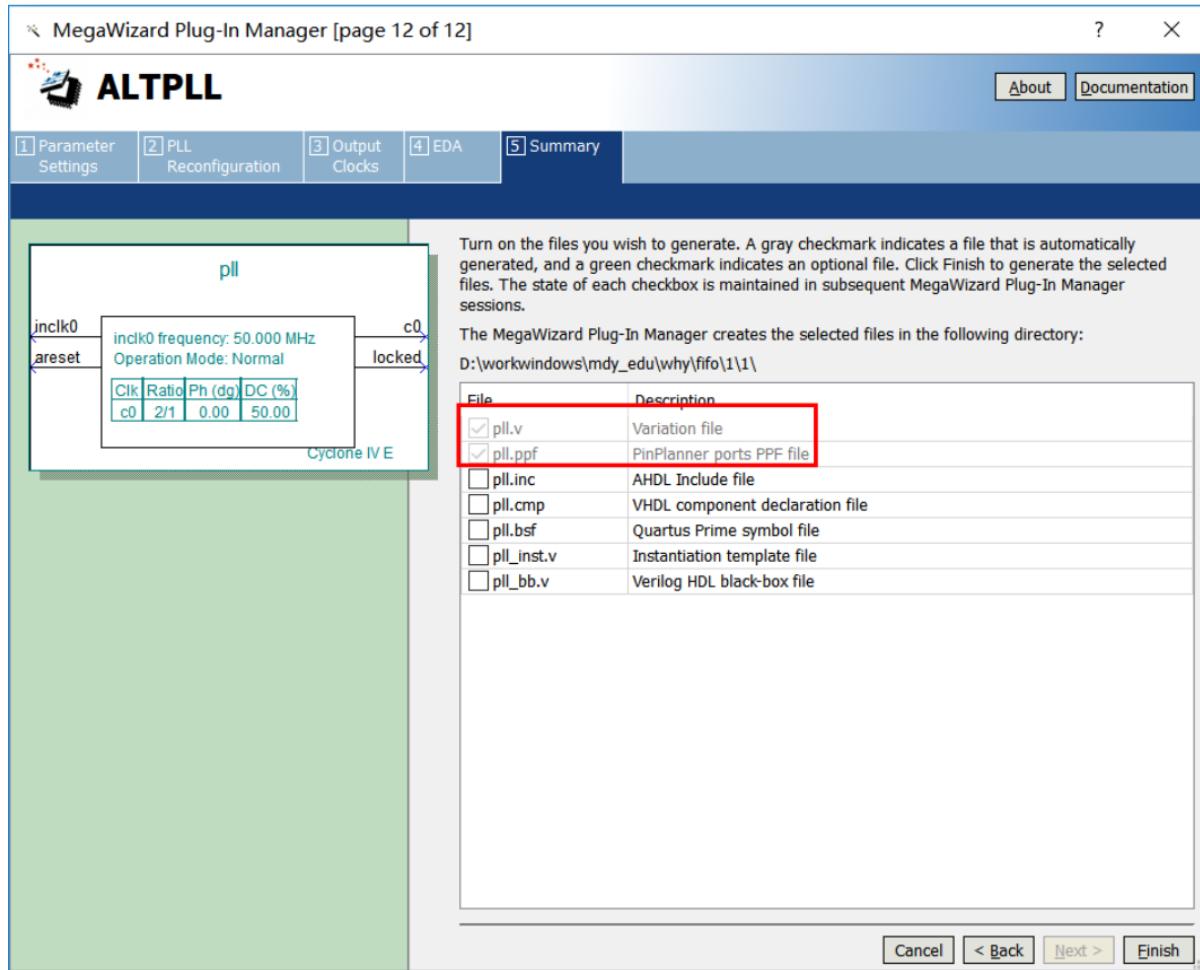




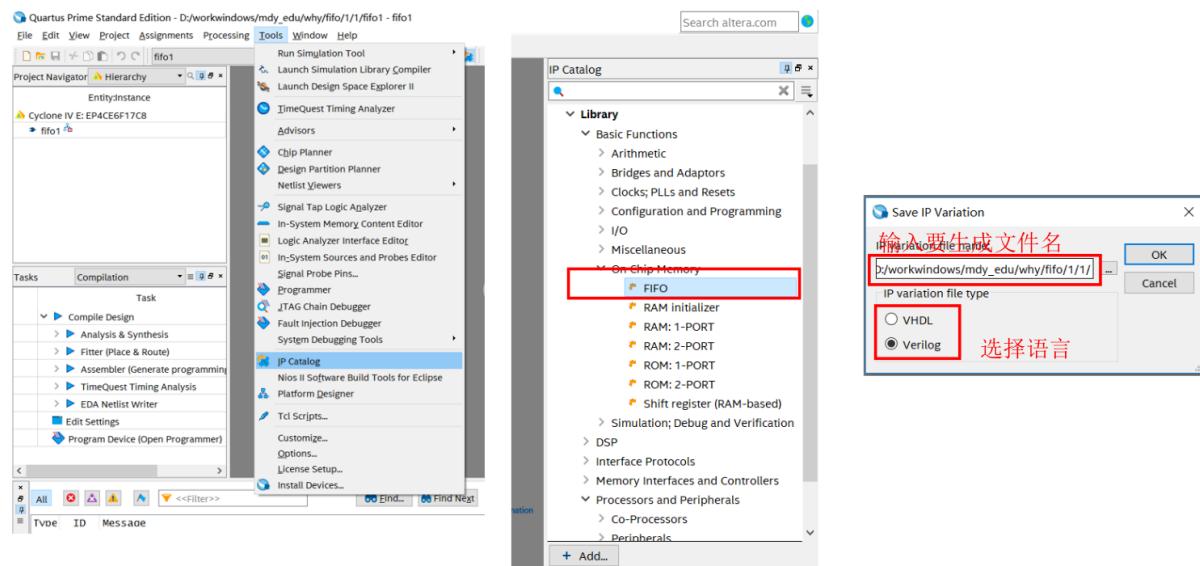


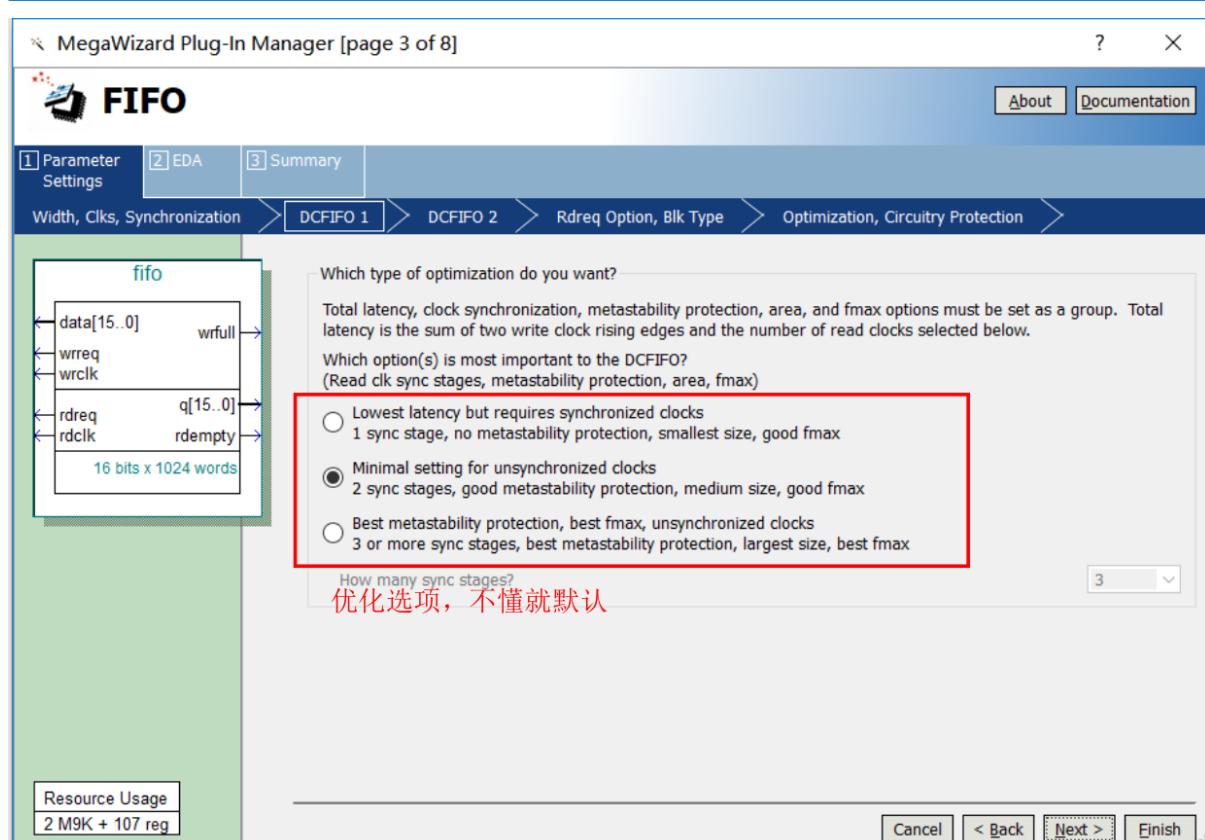
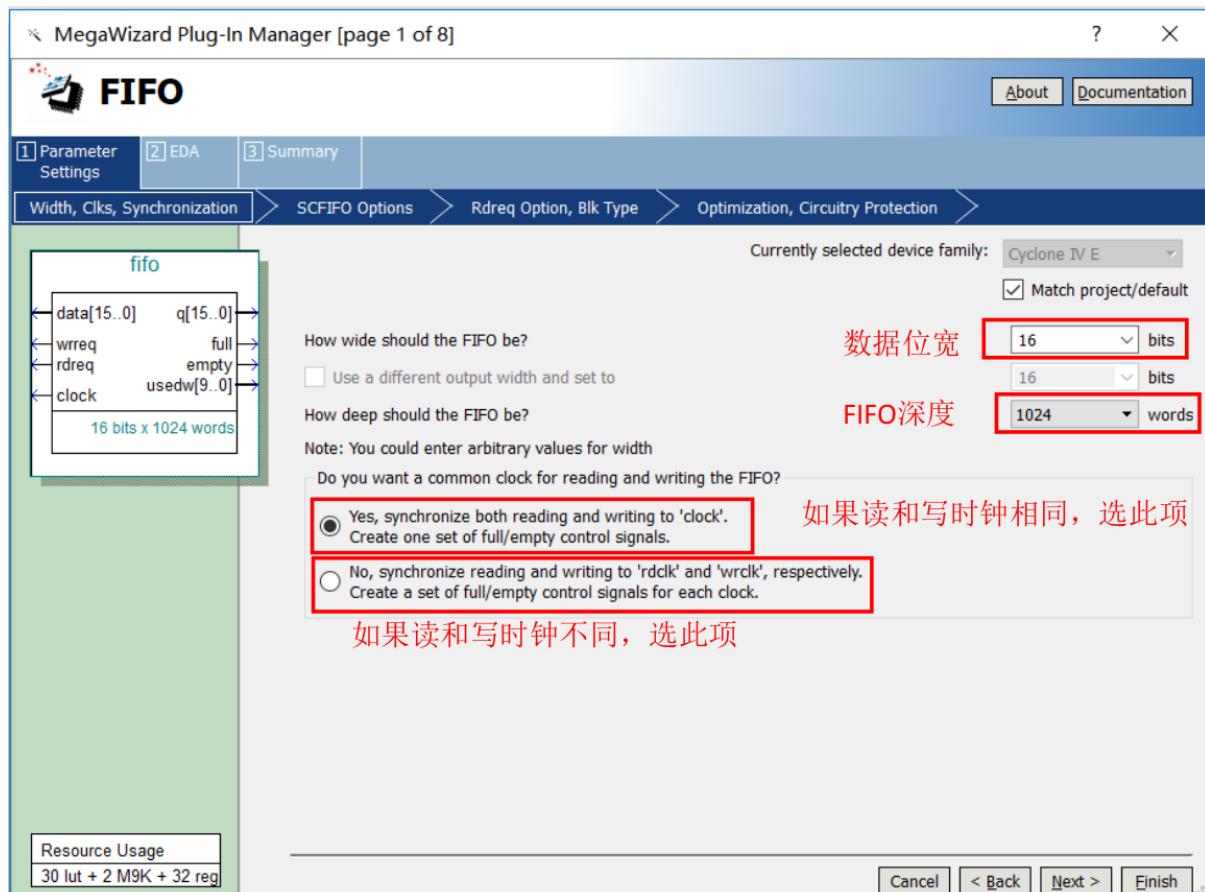


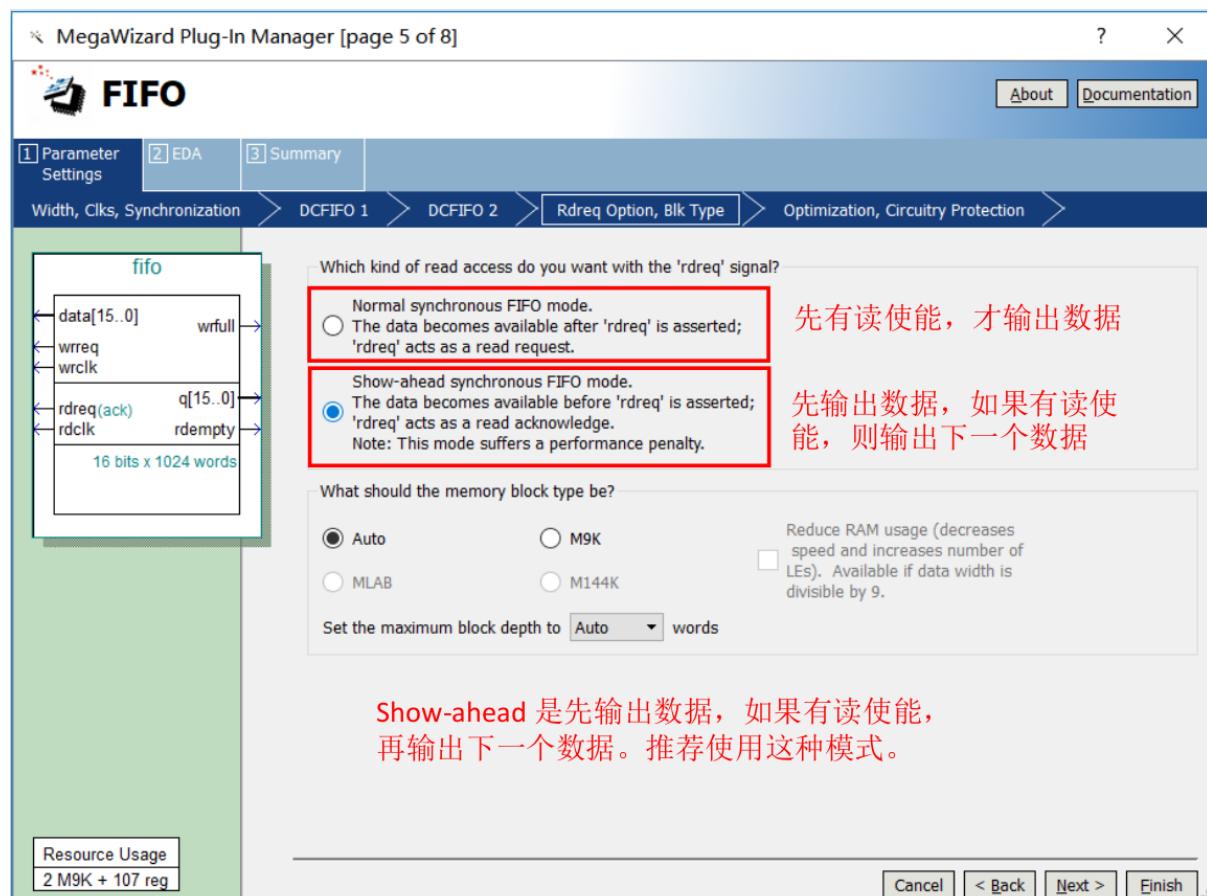
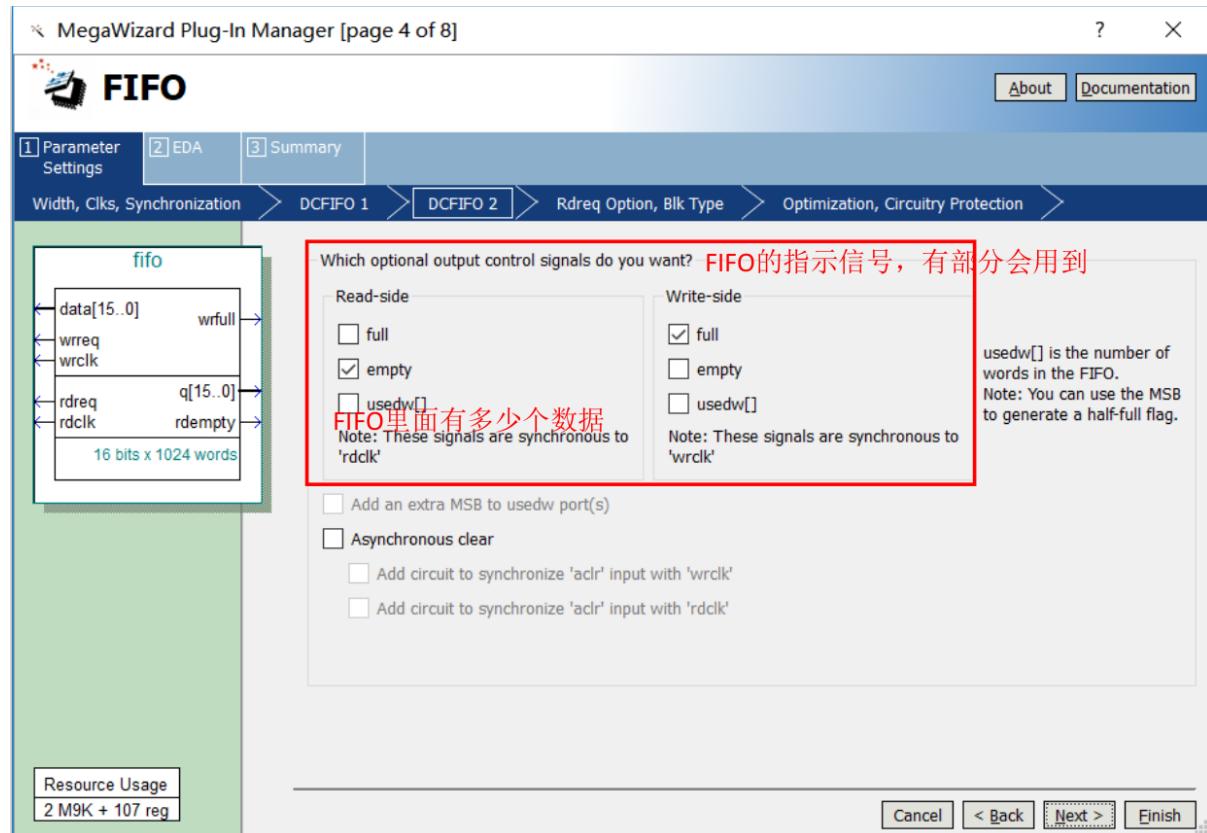




FIFO







※ MegaWizard Plug-In Manager [page 6 of 8]

FIFO

1 Parameter Settings **2 EDA** **3 Summary**

Width, Clks, Synchronization > DCFIFO 1 > DCFIFO 2 > Rdreq Option, Blk Type > Optimization, Circuitry Protection >

Would you like to disable any circuitry protection?

If not required, overflow and underflow checking can be disabled to improve performance.

Disable overflow checking. Writing to a full FIFO will corrupt contents.
 Disable underflow checking. Reading from an empty FIFO will corrupt contents.
 Implement FIFO storage with logic cells only, even if the device contains memory blocks

Resource Usage
2 M9K + 107 reg

默认

Cancel < Back Next > Finish

※ MegaWizard Plug-In Manager [page 7 of 8]

FIFO

1 Parameter Settings **2 EDA** **3 Summary**

Simulation Libraries
To properly simulate the generated design files, the following simulation model file(s) are needed

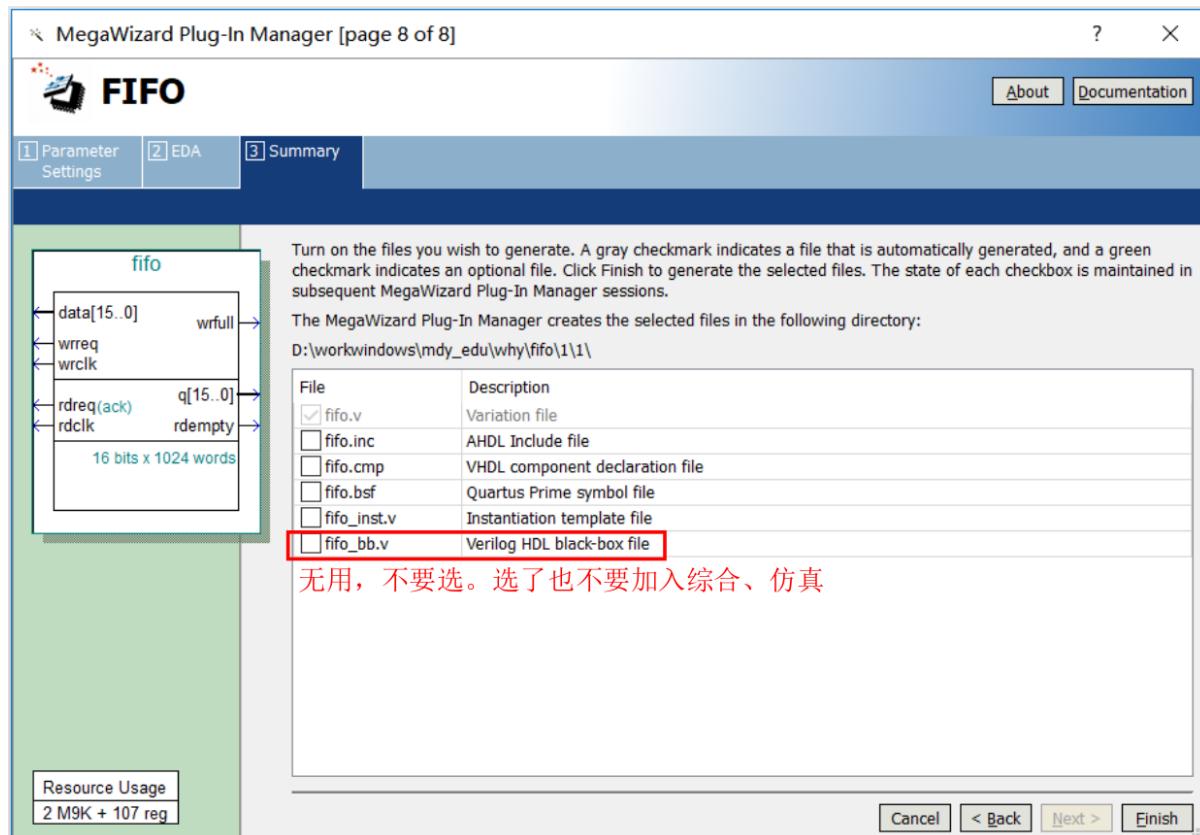
File	Description
altera_mf	Altera megafunction simulation library

Timing and resource estimation
Generates a netlist for timing and resource estimation for this megafunction. If you are synthesizing your design with a third-party EDA synthesis tool, using a timing and resource estimation netlist can allow for better design optimization.
Not all third-party synthesis tools support this feature - check with the tool vendor for complete support information.
Note: Netlist generation can be a time-intensive process. The size of the design and the speed of your system affect the time it takes for netlist generation to complete.
 Generate netlist

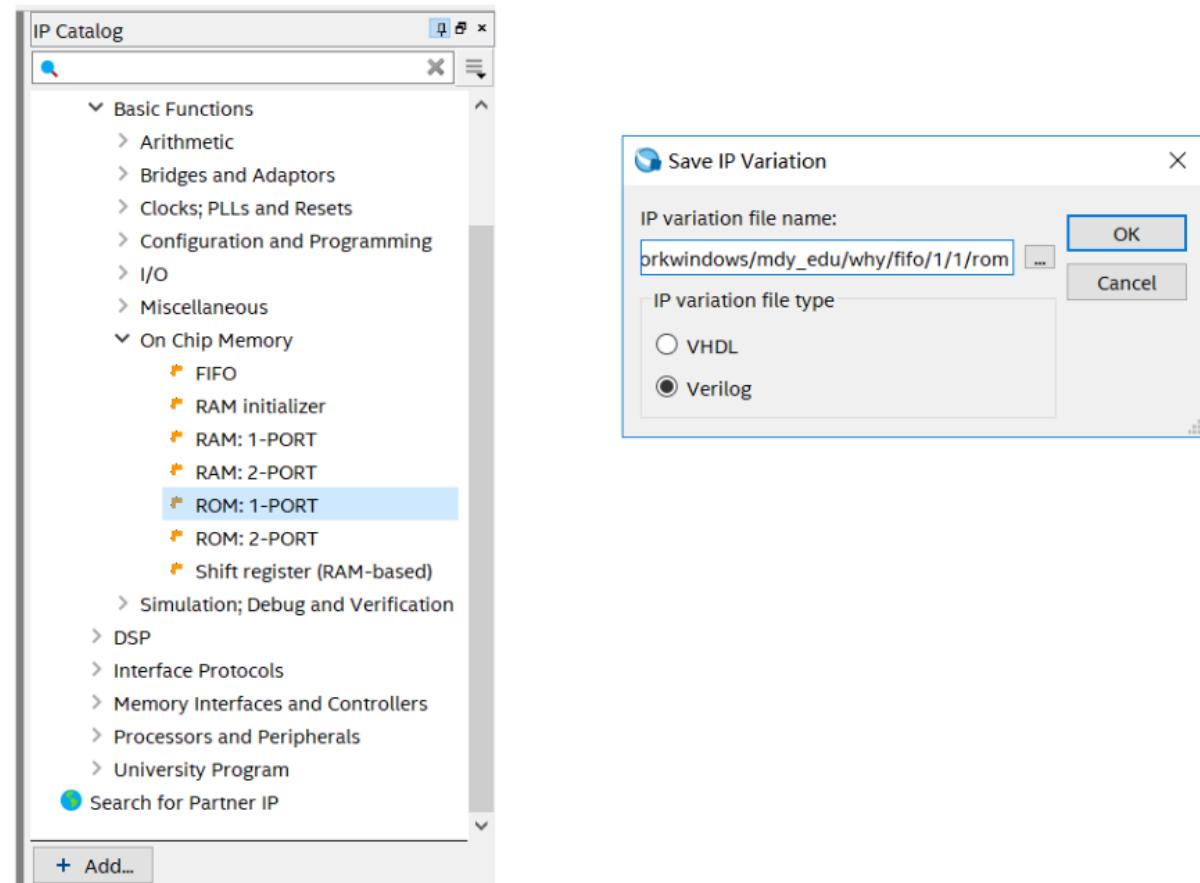
Resource Usage
2 M9K + 107 reg

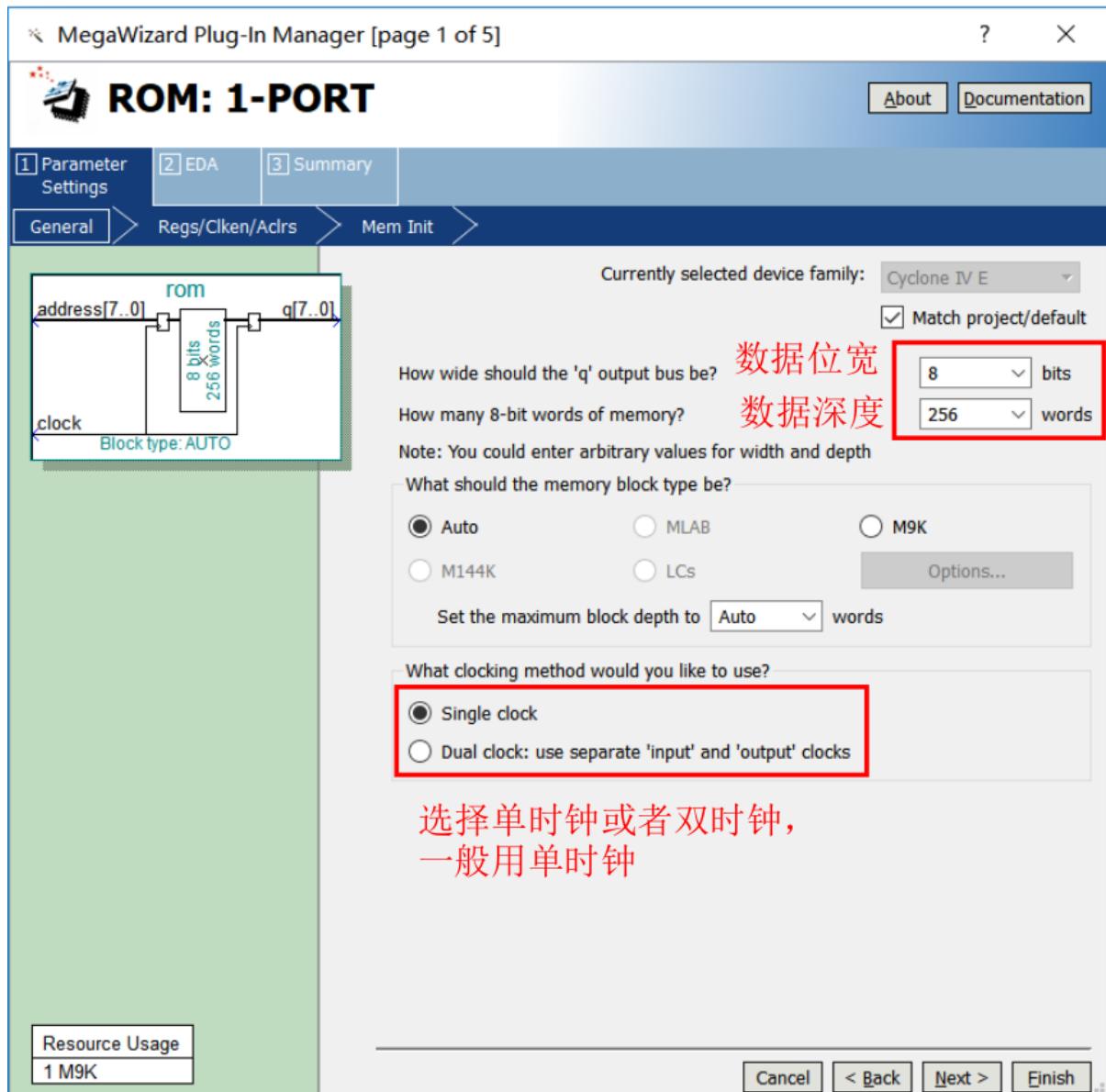
默认

Cancel < Back Next > Finish

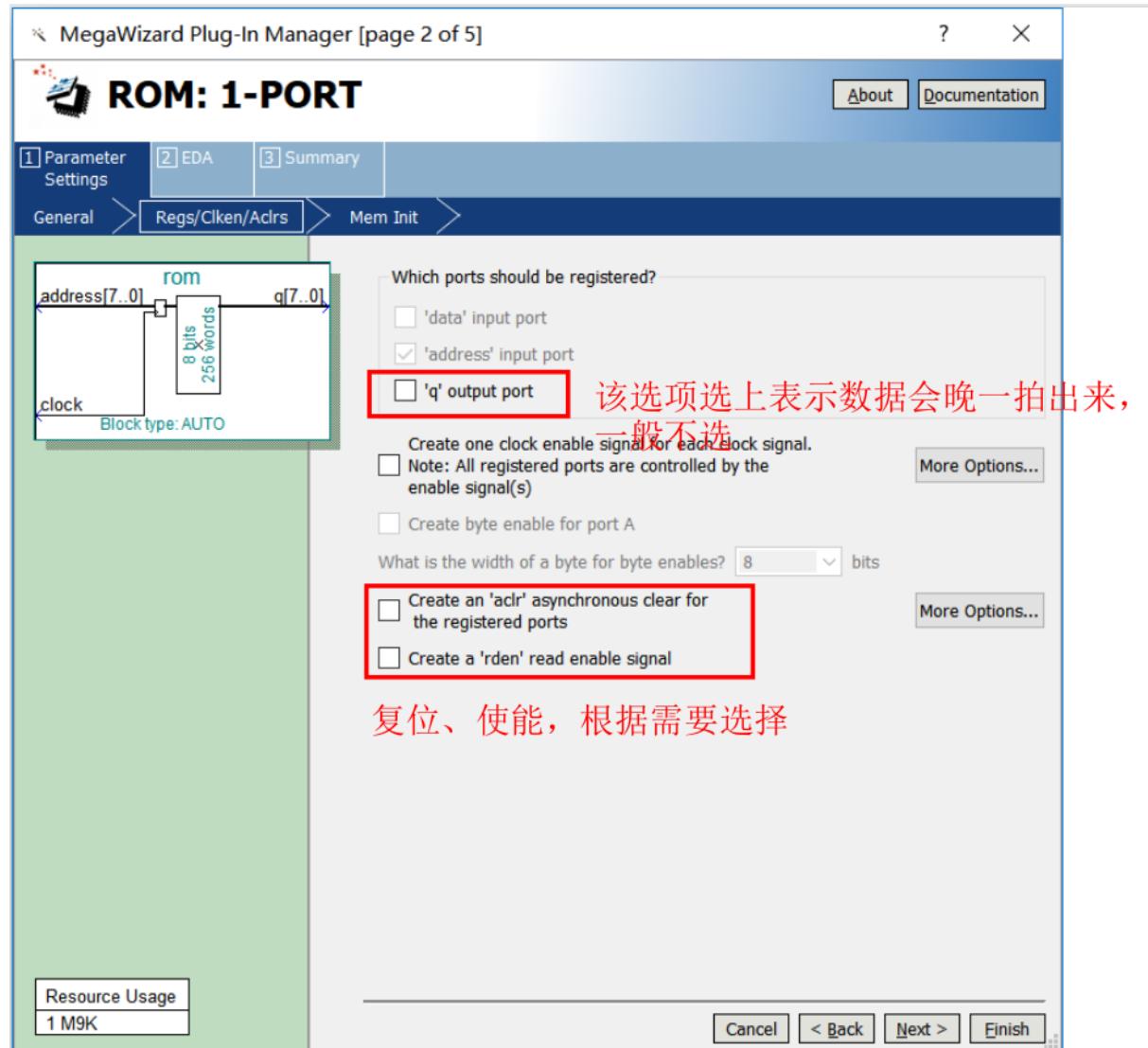


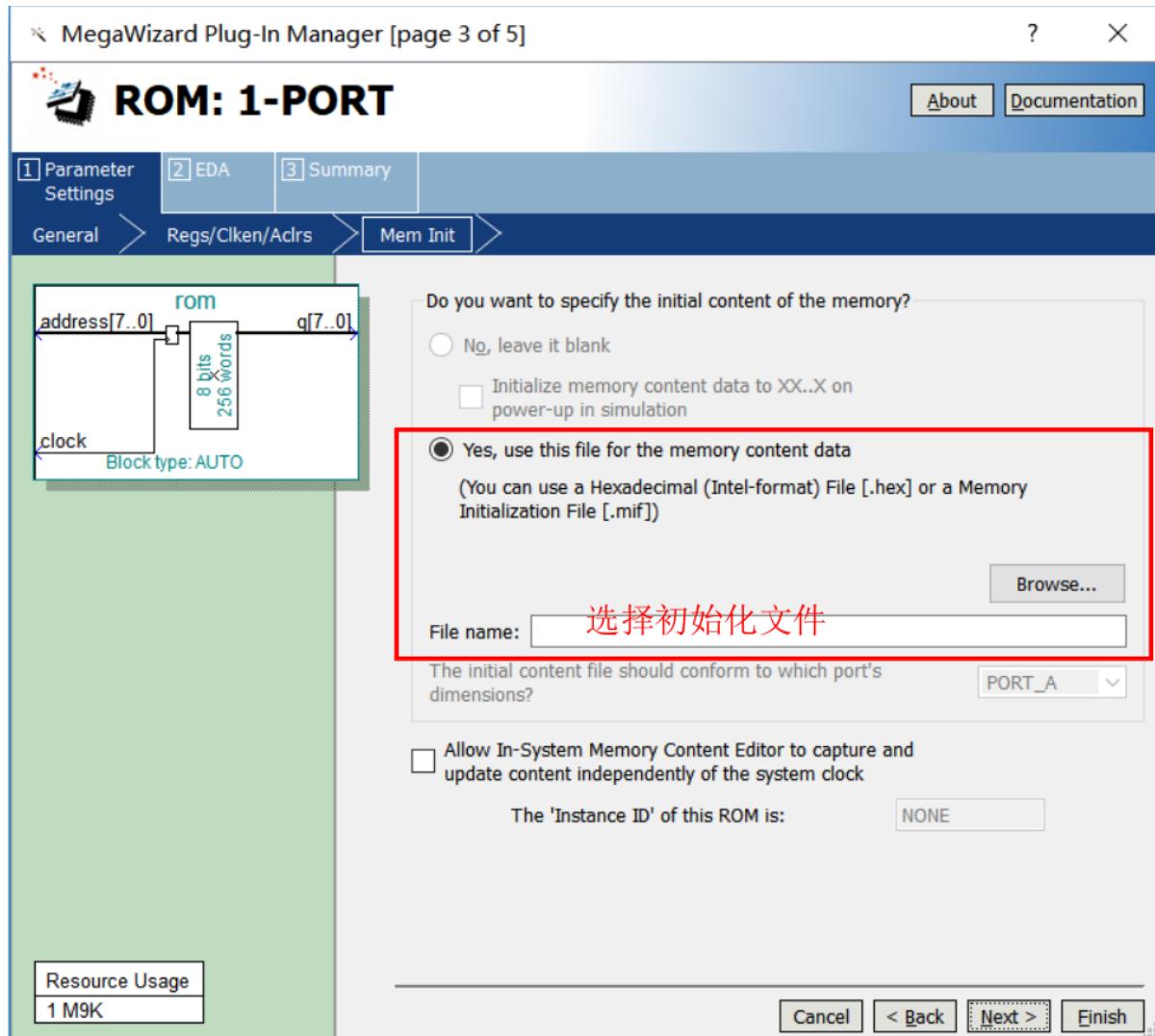
RAMROM

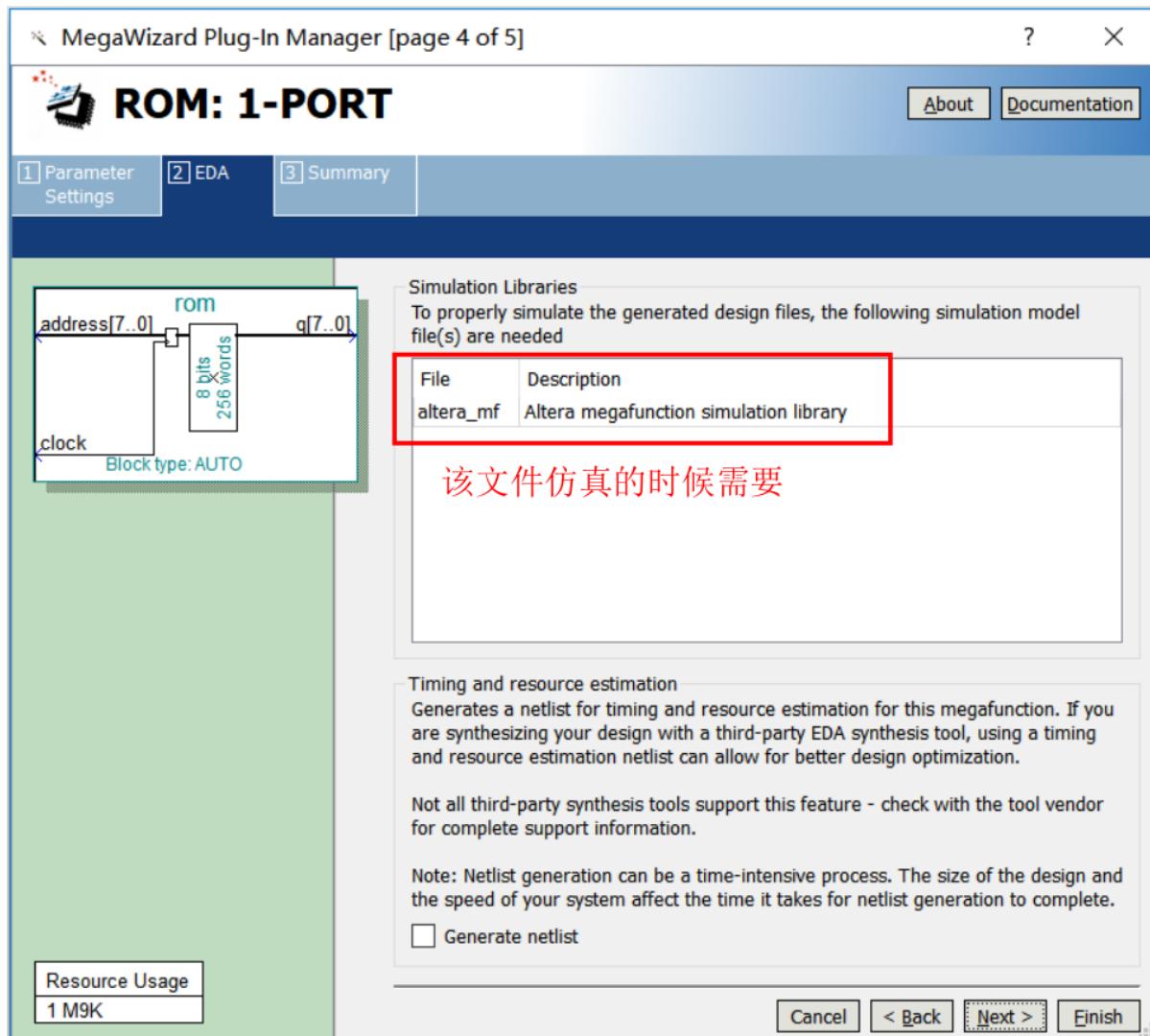


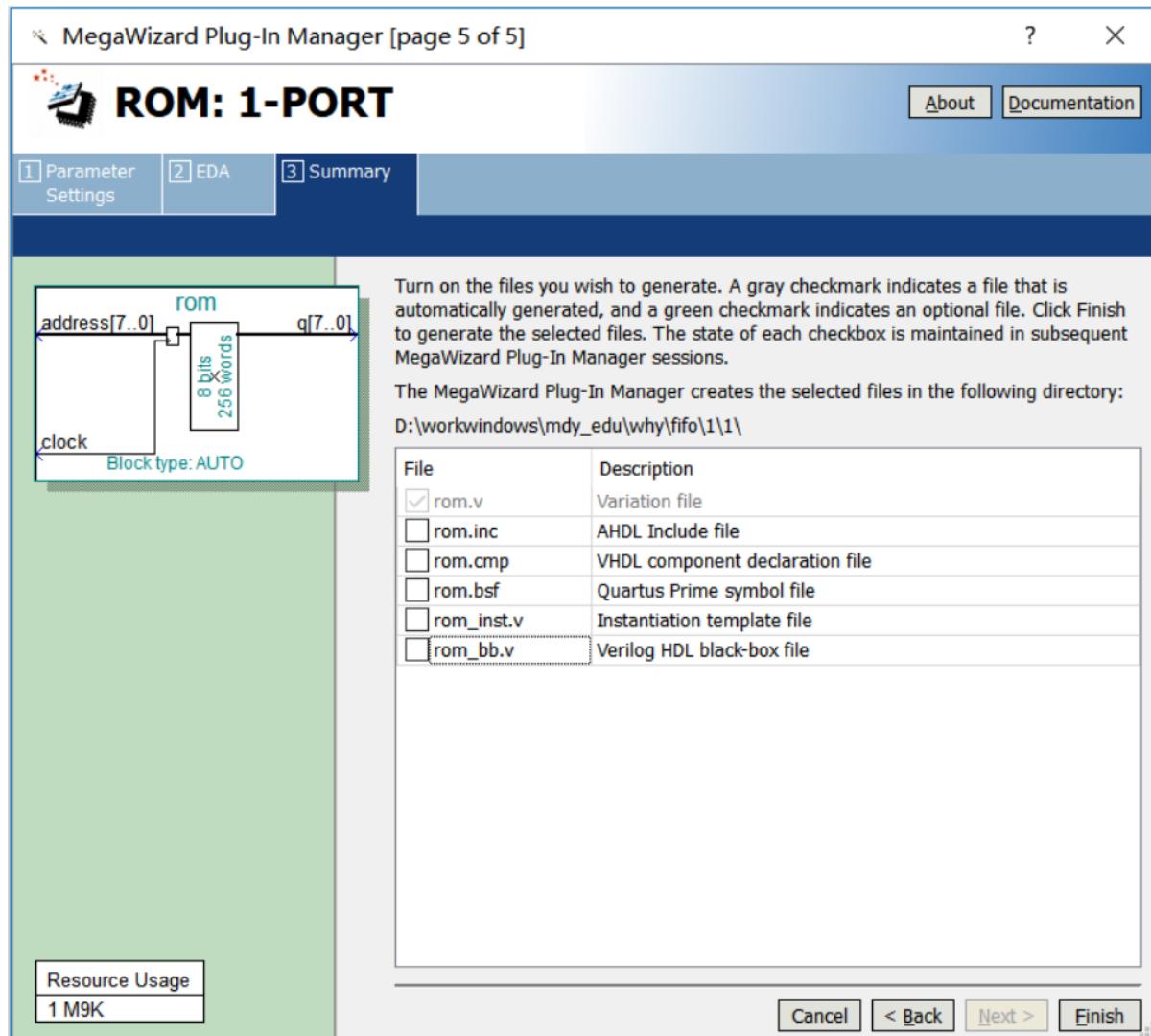


选择单时钟或者双时钟，
一般用单时钟

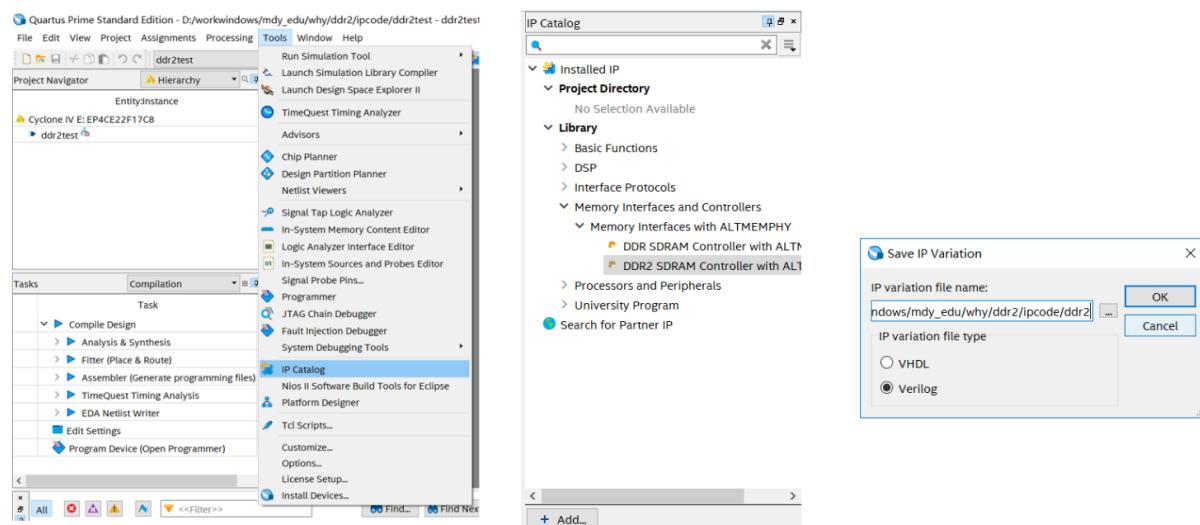








DDRSDRAM



MegaWizard Plug-In Manager - DDR2 SDRAM Controller with ALTMEMPHY

DDR2 SDRAM Controller with ALTMEMPHY

Memory Settings > PHY Settings > Board Settings > Controller Settings >

General Settings

Device family: Cyclone IV E
选择芯片型号
Speed grade: 6
PLL reference clock frequency: 50 MHz (2000 ps)
Memory clock frequency: 133 MHz (751 ps) DDR2时钟
Controller data rate: Half
Local interface clock frequency: 66.5 MHz
Local interface width: 64 bits

Show in 'Memory Presets' List:

Parameter	Value
Memory vendor	(All)
Memory format	(All)
Maximum memory frequency	(All)

Selected memory preset: Micron MT47H32M16-5E
Description: DDR2 SDRAM, 200.0MHz, 1024MB, 16 bits wide, Discrete Device, CAS 3.0, 1 Chip Selected

**选择DDR2型号
如果不在列表, 选相似的
然后根据datasheet修改参数**

也可以外部导入设置 **Load Preset...**
进入修改参数界面 **Modify parameters...**

然后根据datasheet修改参数

修改之后另存为 **Save As...** **OK** **Cancel**

Advanced PHY Settings

Address/Command Clock Settings: Clock phase: 90 时钟偏移, 默认
Auto-Calibration Simulation Options: Quick Calibration 仿真精度选择, 默认

Board Timing Parameters

The time difference between the longest and shortest traces, which connect the FPGA to the memory device.
Board skew: 20 ps

Latency Settings

Fix read latency at: 0 cycles (0 cycles=minimum latency, non-deterministic)

Number of Slots/Discrete Devices 1

Restore Altera Board Defaults

Slew Rates

The slew rate of the output signals affects the setup and hold times of the memory device.
Either enter the slew rates to obtain calculated setup and hold times, or enter the setup and hold times directly.

	Calculated	Applied
CK/CKN slew rate (Differential)	2 V/ns	tS 0.600 0.6 ns
Addr/Command slew rate	1 V/ns	tH 0.600 0.6 ns
DQ/DQS/DQS# slew rate (Differential)	2 V/ns	tDS 0.400 0.4 ns
DQ slew rate	1 V/ns	tDH 0.400 0.4 ns

Intersymbol Interference

Addr/Command eye reduction (setup): 0 ns
Addr/Command eye reduction (hold): 0 ns
DQ eye reduction: 0 ns
DQS/DQS# arrival time: 0 ns

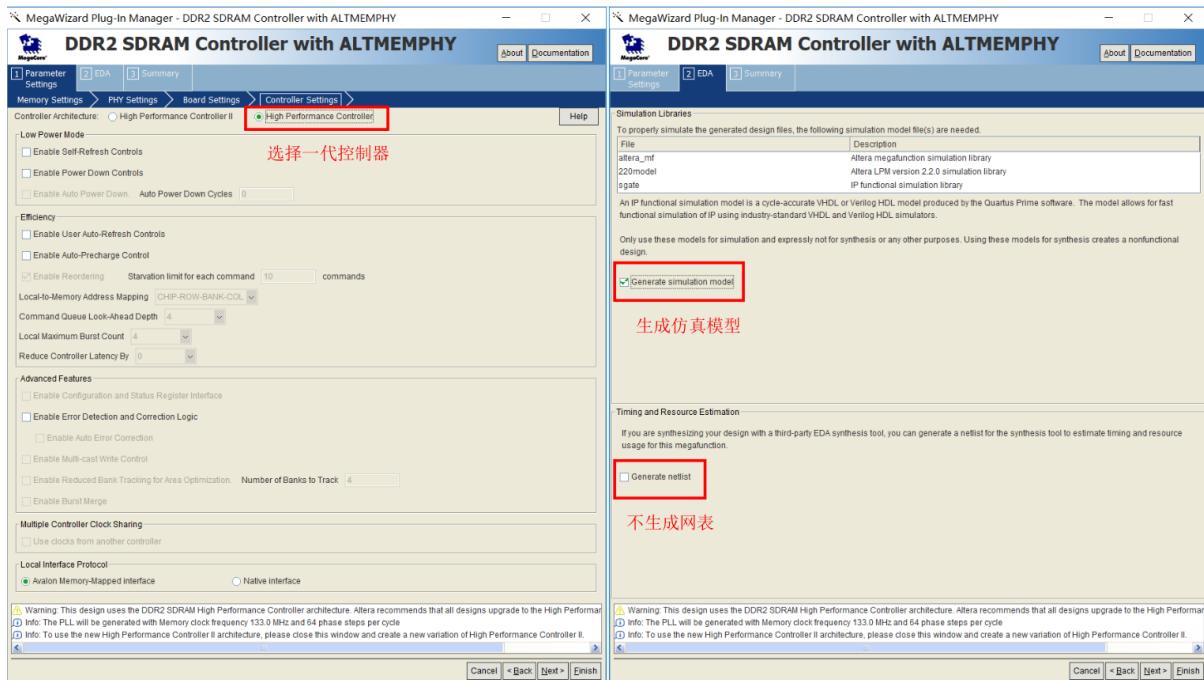
Board Skews

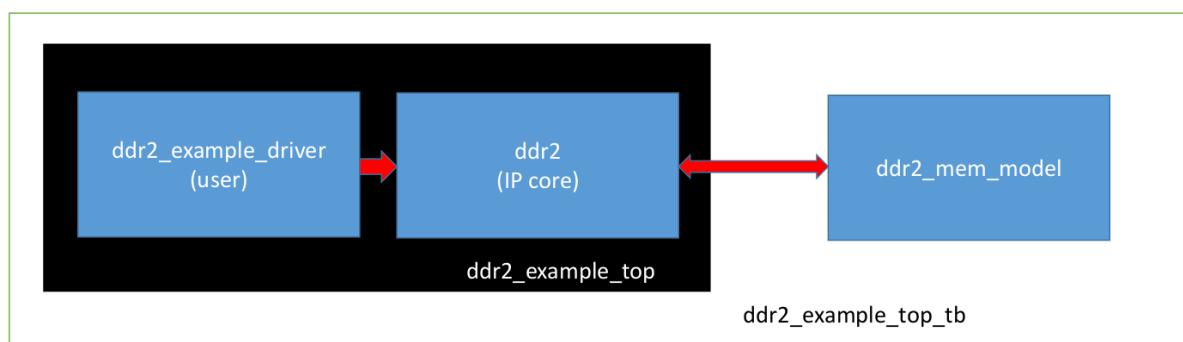
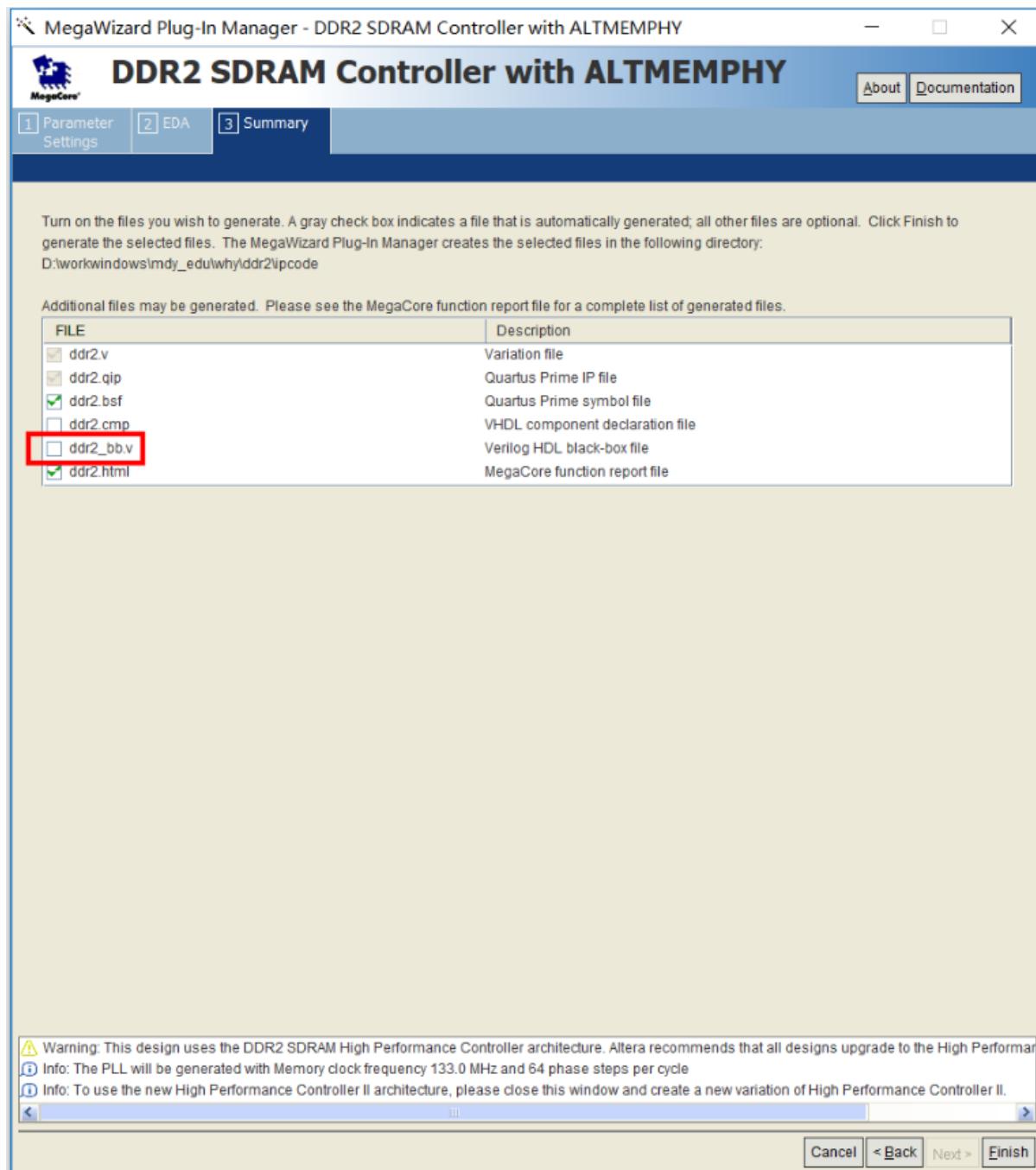
Min CK/DQS skew to DIMM: -0.01 ns
Max CK/DQS skew to DIMM: 0.05 ns
Max skew between DIMMs/devices: 0.05 ns
Max skew within DQS group: 0.02 ns
Max skew between DQS groups: 0.02 ns
Addr/Command to CK skew: 0 ns

默认

**Warning: Cannot meet IRTP requirement of 7.5 ns. For a Memory interface clock frequency of 133.0 MHz, the minimum is 8.2 ns. Click "Modify Parameters".
Info: The PLL will be generated with Memory clock frequency 133.0 MHz and 64 phase steps per cycle**

Cancel **Back** **Next >** **Finish**





用户侧 信号	local_address,	写地址
	local_write_req,	写使能
	local_read_req,	读使能
	local_burstbegin,	无用
	local_wdata,	写数据
	local_be,	写数据位有效标记, 1bit=>8bit
	local_size,	I代控制器为1, II代看说明书
	global_reset_n,	复位信号, 0有效
	pll_ref_clk,	输入时钟
	soft_reset_n,	软件复位信号, 0有效
DDR2 信号	local_ready,	本地准备好信号, 1 && req 读写
	local_rdata,	读数据
	local_rdata_valid,	读数据有效, req之后一段时间才返回
	local_refresh_ack,	
	local_wdata_req,	
	local_init_done,	
	reset_phy_clk_n,	
	mem_odt,	
	mem_cs_n,	
	mem_cke,	
	mem_addr,	
	mem_ba,	
	mem_ras_n,	
	mem_cas_n,	
	mem_we_n,	
	mem_dm,	
	phy_clk,	用户侧信号时钟
	aux_full_rate_clk,	
	aux_half_rate_clk,	
	reset_request_n,	
	mem_clk,	
	mem_clk_n,	
	mem_dq,	
	mem_dqs);	

IP 核生成后会有一个测试用的顶层文件, 该文件可以用来测试硬件是否有问题

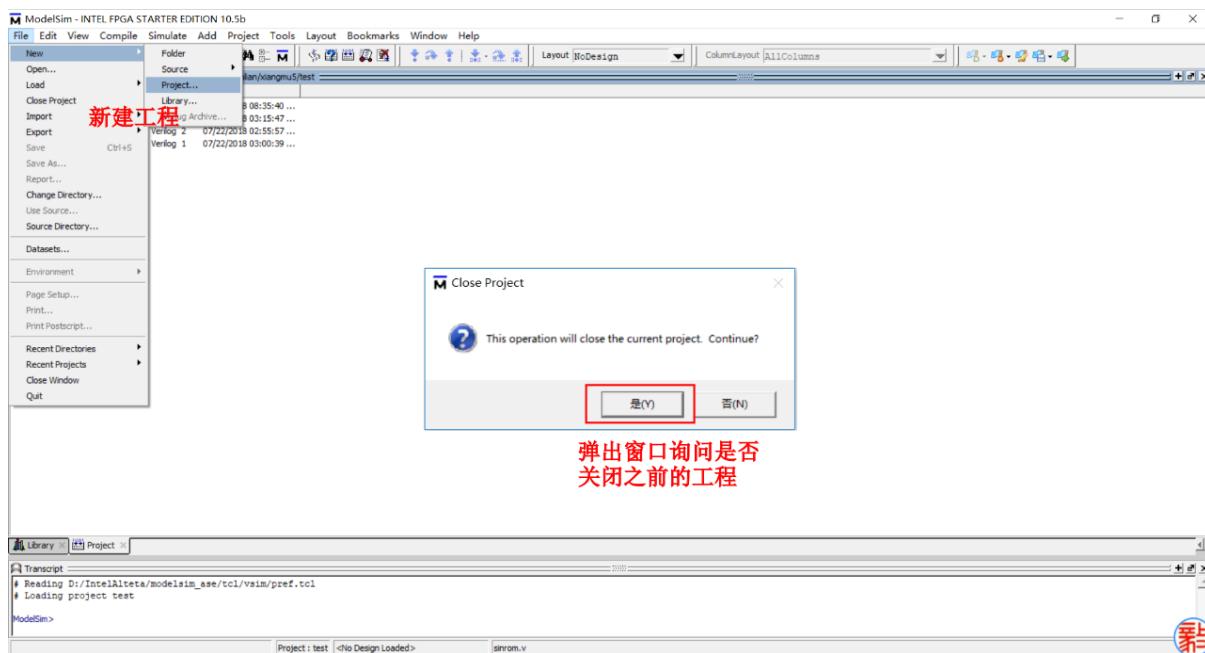
```

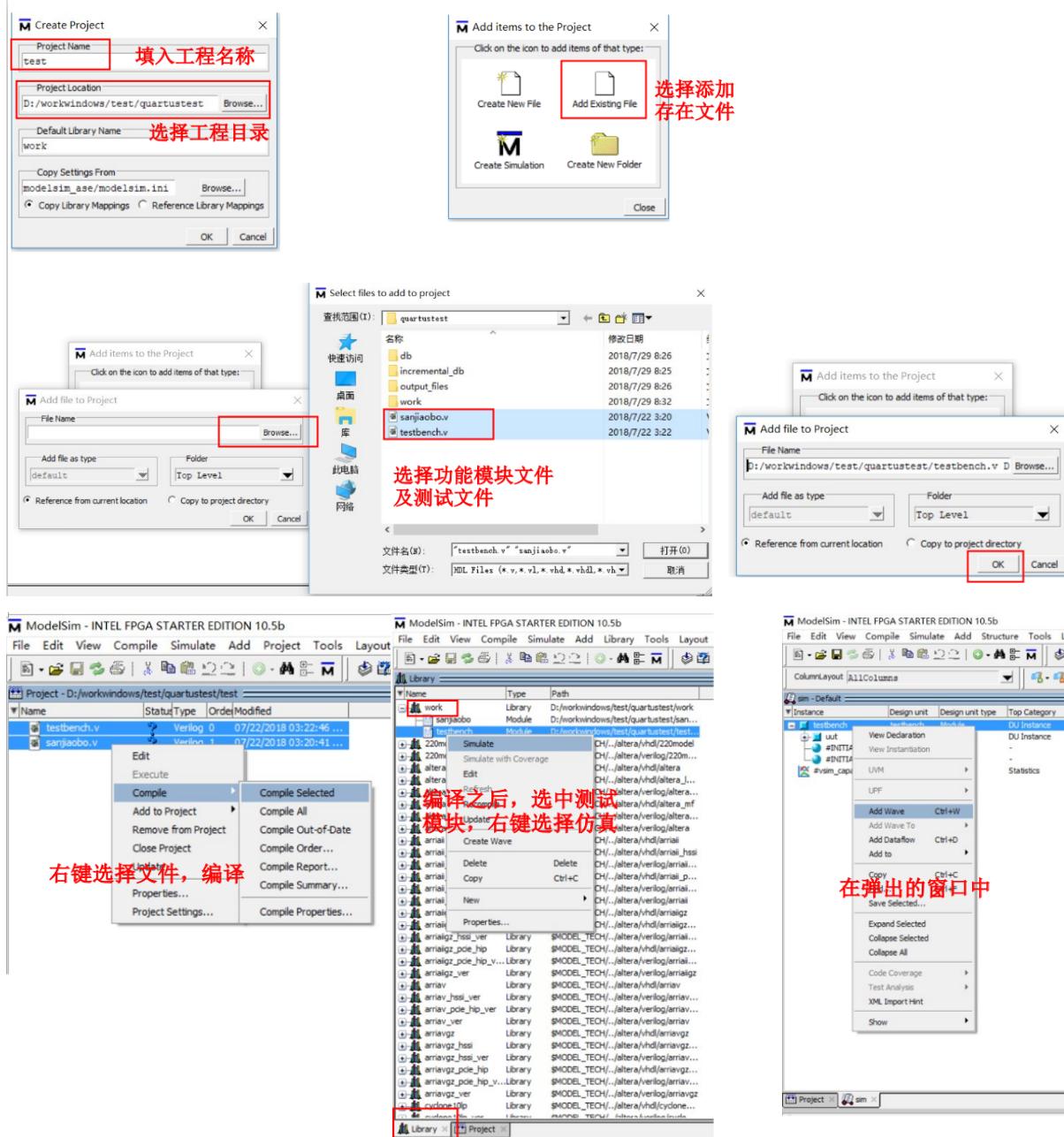
module ddr2_example_top (
    // inputs:
    clock_source,
    global_reset_n, 时钟重置

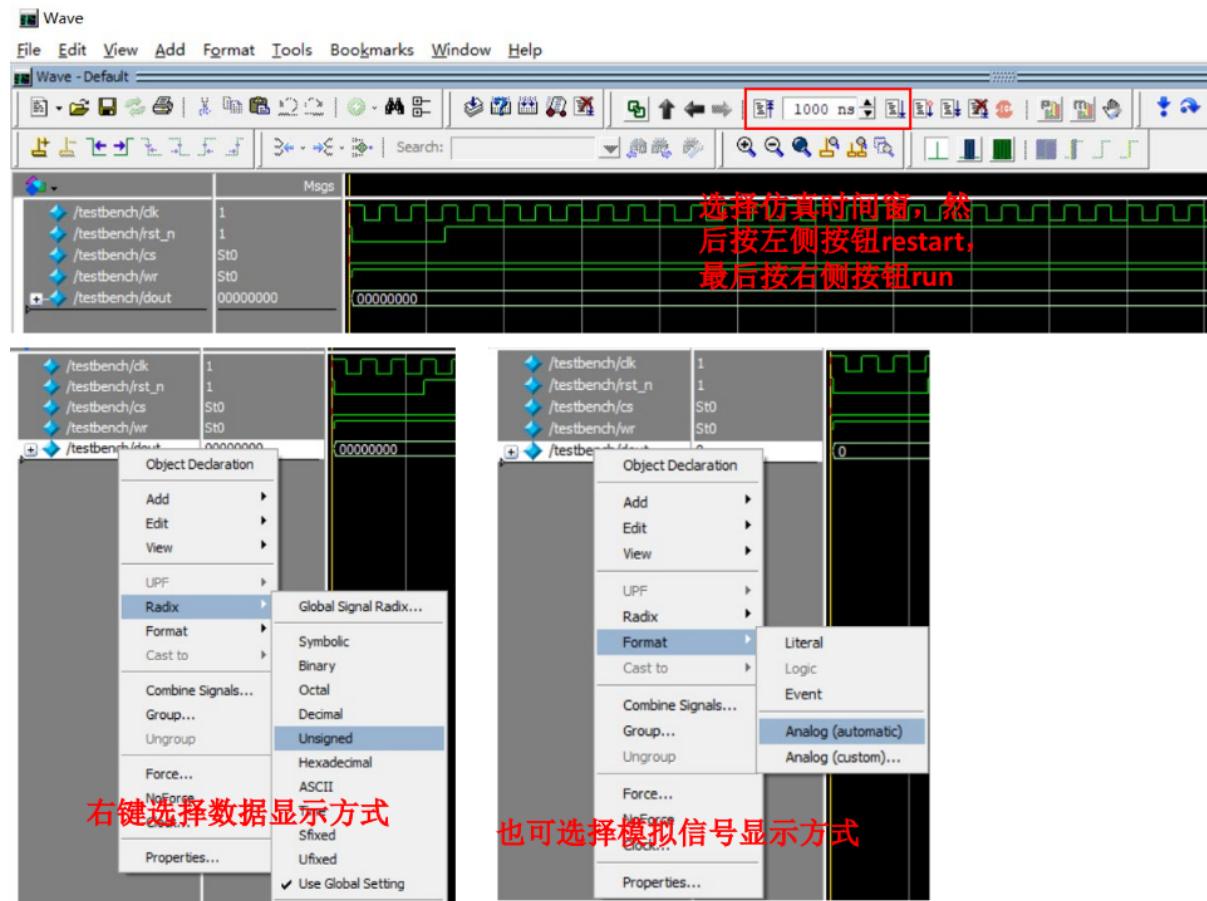
    // outputs:
    mem_addr,
    mem_ba,
    mem_cas_n,
    mem_cke,
    mem_clk,
    mem_clk_n,
    mem_cs_n,
    mem_dm,
    mem_dq,
    mem_dqs,
    mem_odt,
    mem_ras_n,
    mem_we_n,
    pnf,          如果 pnf 保持为 1,
    pnf_per_byte, 则表示正确
    test_complete,
    test_status
);

```

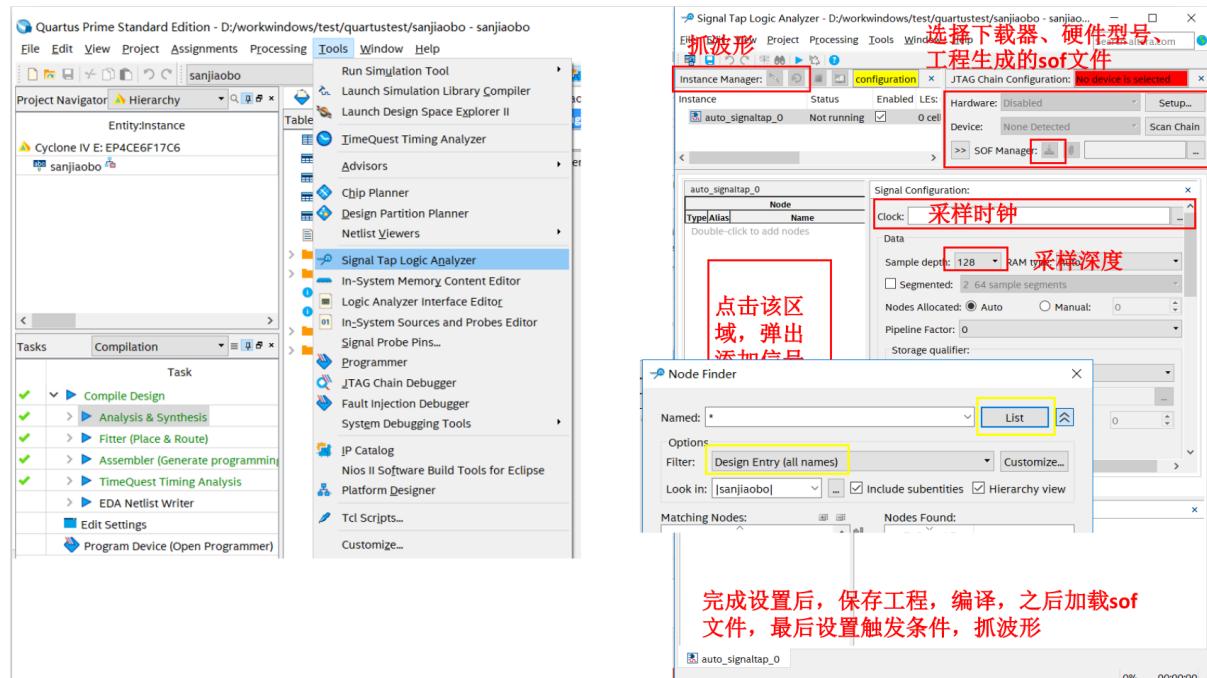
2.3.2 ModelSim







2.3.3 SignalTapII



2.4 Vivado 软件

2.4.1 IP 核

PLL

FIFO

RAMROM

2.4.2 仿真

DCP 文件不能直接用于仿真，需要转成可仿真的文件

转换的方法 (在 tcl console 命令行输入转换命令)

```
open_checkpoint XXX.dcp
write_vhdl -mode funcsim XXX.vhd
write_verilog -mode funcsim XXX.v
```


CHAPTER 3

语法

3.1 VHDL

3.1.1 程序结构

一个完整的 VHDL 程序的以及各部分说明如下：

- **库 (LIBRARY)**

- 存放已经编译的包集合、实体、结构体和配置等。库的好处在于使设计者可共享已经编译过的设计结果

- **包 (PACKAGE)**

- 声明在实体中将用到的信号定义、常数定义、数据类型、元件语句、函数定义和过程定义等

- **实体 (ENTITY)**

- 定义电路的输入/输出接口

- **结构体 (ARCHITECTURE)**

- 描述电路内部的功能。一个实体可以对应很多个结构体，但在同一时间，只有一个结构体被使用

- **配置 (CONFIGURATION)**

- 决定哪一个结构体被使用

实体

实体用于定义电路的输入/输出引脚，但并不描述电路的具体构造和实现的功能。

实体声明的格式是：

```
ENTITY 实体名 IS
[GENERIC (常数名：数据类型：设定值)] -- 类属参数说明，"[]" 中内容为可选项
PORT
()
```

(下页继续)

(续上页)

```

端口名 1: 端口方向 端口类型; -- 端口声明语句用分号隔开
端口名 1: 端口方向 端口类型;

.
.

端口名 n: 端口方向 端口类型;
);
END [实体名]; -- 可以只用 END 结束实体声明, 不一定加实体名

```

格式说明:

- **实体名**
 - 实体名必须与文件名相同, 否则编译时会出错。
- **类属参数**
 - 类属参数为实体声明中的可选项, 常用来规定端口的大小、信号的定时特征等。
- **端口名**
 - 端口名时设计者赋予每个外部引脚的名称。
- **端口方向**
 - 端口方向用来定义外部引脚的信号方向时输入还是输出 (或者同时可作为输入与输出)
- **端口类型**
 - 定义端口的数据类型。VHDL 是一种强类型语言, 即对语句中的所有端口信号、内部信号和操作数的数据类型有严格规定, 只有相同数据类型的端口信号和操作数才能相互作用。

结构体

结构体描述实体内部的结构或功能。一个实体可对应多个结构体。每个结构体分别代表该实体功能的不同实现方案或不同描述方式。在同一时刻, 只有一个结构体起作用, 可以通过配置来决定使用哪一个结构体进行综合或仿真。

结构体的语法格式如下:

```

ARCHITECTURE 结构体名 OF 实体名 IS
[声明语句]
BEGIN
功能描述语句
END [结构体名]

```

实体名必须与实体声明部分所取的名字相同, 而结构体名则可有设计者自由选择, 但当一个实体具有多个结构体时, 各结构体的取名不可相同。

声明语句用于声明该结构体将用到的信号、数据类型、常数、子程序和元件等。需要注意的是, 在一个结构体内声明的数据类型、常数、子程序 (包括函数和过程) 和元件只能用于该结构体中。如果希望在其它的实体或结构体中引用这些定义, 那么需要将其作为包来处理。

功能描述语句具体描述了结构体的功能和行为。功能描述语句可能包含有 5 种不同类型的以并行方式工作的语句结构, 这几个语句结构又被称为结构体的子结构。

- **块语句 (BLOCK):** 由一系列并行语句 (concurrent statement) 组成, 从形式上划分出模块, 改善程序的可读性, 对综合无影响。
- **进程语句 (PROCESS):** 进程内部为顺序语句, 而不同进程间则是并行执行的。进程只有在某个敏感信号发生变化时才会触发。
- **信号赋值语句:** 将实体内的处理结果向定义的信号或端口进行赋值。
- **子程序调用:** 调用过程 (PROCEDURE) 或函数 (FUNCTION), 并将获得的结果赋值给信号。

- 元件例化语句：调用其它设计实体描述的电路，将其作为本设计实体的一个元件。元件例化时实现层次化设计的重要语句。

库与包的调用

当要引用一个库时，首先要对库名进行说明，其格式为：

```
LIBRARY 库名; -- 如 LIBRARY IEEE; 即调用 IEEE 标准库
```

对库名进行说明后，就可以使用库中已编译好的设计。而对库中程序包的访问，则必须通过 USE 语句实现，其格式为：

```
USE 库名. 程序包名. 项目名; -- 如 USE IEEE.Std_logic_1164.ALL;
```

其中，关键字 ALL 表示本设计实体可以引用次程序包中的所有资源。

虽然 NUMERIC_STD 有时候操作有点繁琐，但是更加规矩，并且可以有效避免一些错误，所以应该首选使用该库文件。一般来说，以下三行代码足以应付大部分的 VHDL 程序设计了。调用库和程序的语句本身在综合时并不消耗更多的资源。

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

3.1.2 基本数据类型

预定义数据类型

常见的预定义数据类型及其简要说明如下：

- 布尔量 (boolean)：取值位 false 和 true，用于逻辑运算
- 位 (bit)：取值为 0 和 1，用于逻辑运算
- 位矢量 (bit_vector)：基于 bit 类型的数组，用于逻辑运算
- 整数 (integer)：整数的取值范围是 -(2^31 -1) ~ (2^31 -1)，可用 32 位有符号的二进制数表示，用于数值运算
- 实数 (real)：实数的取值范围是 -1.0E38 ~ +1.0E38，仅用于仿真，不可综合
- 时间 (time)：完整的时间类型包括整数和物理量单位两部分，整数与单位之间至少留 1 个空格，如 20 ms、30 us 等。整数部分取值范围与 integer 相同。此类型仅用于仿真，不可综合

布尔数据类型

布尔数据类型实际上是一个二值枚举型数据类型，取值为 false 和 true。

位数据类型

位与布尔一样，同属二值枚举型数据类型。取值为 0 或者 1。对应于实际电路中的低电平与高电平。bit 类型的数据对象可以进行“与”、“或”、“非”等逻辑运算，结果仍为 bit 类型。

位矢量数据类型

位矢量是基于位类型的数组。使用 bit_vector 时，必须注明数组中的元素个数和排列方向。例如：

```
signal a: bit_vector(0 to 7);
```

信号 a 被定义成一个具有 8 个元素的数组，而且它的最高位为 a(0)，而最低位为 a(7)。

若希望这个数组的排列符合日常使用的顺序，即最高位为 a(7)，而最低位为 a(0)，则应将该信号声明语句改写成：

```
signal a : bit_vector(7 downto 0);
```

关键字 to 表示数组从左到右是生序排列，而 downto 则是降序排列。

整数数据类型

整数类型的数包括正整数、负整数和零。在 VHDL 中，整数的取值范围为 $-(2^{31}-1) \sim (2^{31}-1)$ 。整数类型的数常用于加、减、乘、除四则运算。在使用整数时，必须用 range … to … 限定整数的范围，综合器将根据所限定的范围来决定此信号或变量的二进制数的位数。若所设计的整数范围包括负数，则该数将以二进制补码的形式出现。

IEEE 预定义标准逻辑位与矢量

在 IEEE 库的程序包 std_logic_1164 中，定义了两个十分重要的数据类型，即标准逻辑位 std_logic 和标准逻辑矢量 std_logic_vector。

标准逻辑位数据类型

标准逻辑位数据类型共定义了 9 种信号状态。

- U：未初始化的
- X：强未知的
- 0：强 0
- 1：强 1
- Z：高阻态
- W：弱未知的
- L：弱 0
- H：弱 1
- -：忽略

std_logic 的信号定义比 bit 类型对数字电路的逻辑特性描述更完整，更真实。std_logic 中的 X 态和 Z 态可以使设计者模拟一些未知的和高阻态的线路情况，“-”态常用于一些 boolean 表达式的化简。但就综合而言，只有 4 种状态可被综合，即 0、1、“-”和 Z。其它态虽然不可综合，但对行为仿真仍有十分重要的意义。

标准逻辑位矢量数据类型

标准逻辑位矢量是基于 std_logic 类型的数组。简而言之，std_logic_vector 和 std_logic 的关系就像 bit_vector 与 bit 的关系。

需要强调的是，使用 std_logic 和 std_logic_vector 时，一定要调用 IEEE 库中的 std_logic_1164 的程序包。

用户自定义的数据类型

用户自定义的数据类型主要有枚举类型（enumerated types）和数组类型（array types）等，前者常用于状态机描述，而后者常用于 ROM 和 RAM 的描述等。

枚举类型

枚举类型的语法格式如下：

```
type 数据类型名 is (元素 1, 元素 2, ...);
```

在状态机描述中，常常使用枚举类型为每一状态命名，使程序更具有可读性。例如：

```
type state_type is (start, step1, step2, stop);
signal state : state_type;
```

上面这个例子为状态机定义了 4 个状态: start、step1、step2、stop。表征当前状态的信号 state 就在这 4 个状态中取值。

数组类型

数组类型常用于组合同样数据类型的元素，其语法格式如下：

```
type 数组名 is array (范围) of 数据类型;
```

下面是几个数组定义的例子：

```
type byte is array (7 downto 0) of bit; -- 1 byte=8 bits
type word is array (31 downto 0) of bit; -- 1 word= 32 bits
```

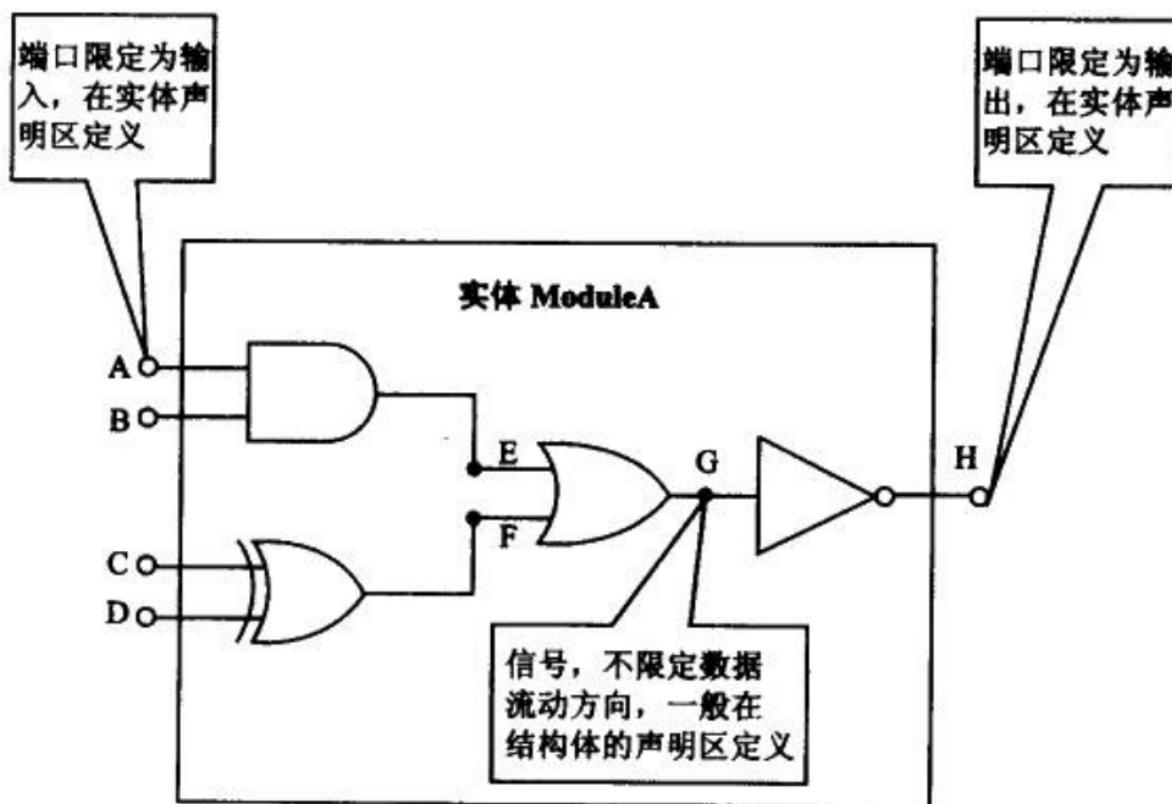
3.1.3 数据对象

在 VHDL 中，数据对象 (data object) 有 3 类：信号 (signal)、变量 (variable) 和常量 (constant)。

VHDL 中的变量和常量与软件高级语言中的变量和常量相似，而信号则具有更多的硬件特征，是硬件描述语言所特有的数据对象。

信号

信号是用来描述实体内部节点的重要数据类型。



从图中可以看出，信号 (signal) 与端口 (port) 之间的相似之处和差异点。信号与端口都描述了电路中实际存在的节点 (node)，只是信号描述的是实体内部的节点，而端口则描述了实体与外界的接口。在语法上，信号的声明与端口的声明很相似，下面是信号声明的语法格式：

```
signal 信号名: 数据类型 [:= 初始值]; -- 初始值仅在仿真时有意义，综合时将忽略此值
```

对比信号声明与端口声明的格式可以发现，除了端口声明中规定的方向之外，二者无任何差别（虽然信号声明比端口声明多了初始值的赋值，但是这一赋值仅在仿真时有意义，综合器会忽略这一赋值。因此在实际应用中，基本不使用初始值赋值语句）。换句话说，可以将信号理解为“实体内部不限定数据流动方向的端口”，或者将端口理解为“限定数据流动方向的信号”。因此，信号赋值语句同样适用于端口。

信号赋值语句的格式如下：

```
目标信号名 <= 表达式;
```

信号赋值语句同样适用于位矢量和标准逻辑矢量，只要赋值符号左、右两边的位数相同即可。

需要特别强调的是，信号的赋值具有“非立即性”，即会有延时。这与实际硬件的传播延迟特性十分吻合。

变量

变量只能在进程和子程序中使用，主要用于描述算法和方便程序中的数值计算。

定义变量的语法格式如下：

```
variable 变量名: 数据类型 [:= 初始值]; --初始值仅在仿真时有意义，综合时将忽略此值
```

定义变量与定义信号的语法格式十分相似，只是将关键字 signal 变成 variable。与信号一样，变量的初始值赋值只在仿真中有用，综合时将被忽略，因此在实际应用中很少对变量赋初值。虽然二者语法格式十分相似，但在程序中的位置却不同。

变量赋值语句的格式如下：

```
目标变量名 := 表达式;
```

表达式可以是一个数值，也可以是一个与目标变量数据类型相同的变量，或者是运算表达式。

变量与信号的区别不仅仅再与声明与赋值语句的格式，最重要的区别在于信号与实际电路的某个节点或信号线对应，因此硬件具有传播延迟特征，所有信号的赋值具有延时特性；而变量是一个抽象值，它不与任何实际电路连线对应，因此它的赋值是立即生效的。

在实际应用中，信号的行为更接近硬件的实际情况，因此将更多时用信号进行电路内部数据传递。只有在描述一些算法时，才用到变量。当然，有些情况下（如作矢量的索引值等）只能时用变量。

常数

VHDL 中的常数与软件高级语言中的常数十分相似，作用如下：

- 保证该常数描述的那部分数据在程序中不会因操作被改变；
- 对程序中的某些关键数值进行命名，可以提高程序的可读性；
- 将出现次数较多的关键数值用常数表示，可以使程序易于修改：只需修改常数就可以替换所有相关数值。

定义常数的语法格式如下：

```
constant 常数名 : 数据类型 := 设置值
```

3.1.4 运算符

VHDL 的运算符主要有 4 种：算术运算符、并置运算符、关系运算符和逻辑运算符。

算术运算符

表 1: 常见算术运算符及其说明

运算符	含义	备注
+	加	一般情况下, + 号两边只能是整形信号(变量)。但若事先调用了 IEEE 库中的 std_logic_1164 和 std_logic_unsigned(或 std_logic_signed) 程序包, 则 + 号两边可以是: 1) std_logic_vector+std_logic_vector; 2) std_logic_vector+integer; 3) integer+std_logic_vector; 4) integer+integer
-	减	同上
*	乘	一般情况下, * 号两边只能是整形信号(变量)。但若事先调用了 IEEE 库中的 std_logic_1164 和 std_logic_unsigned(或 std_logic_signed) 程序包, 则 * 号两边可以是: 1) std_logic_vector * std_logic_vector; 2) integer * integer
/	除	
**	乘方	
MOD	求模	
REM	求余	
ABS	求绝对值	

并置运算符

并置运算符“&”用于将多个元素或矢量连接成新的矢量。例如:

```
signal A : std_logic_vector(3 downto 0);
signal B : std_logic_vector(1 downto 0);
signal C : std_logic_vector(5 downto 0);
signal D : std_logic_vector(4 downto 0);
signal E : std_logic_vector(2 downto 0);
.
.
.
C<=A&B; --矢量与矢量并置
D<=A(1 downto 0)&B(1 downto 0)&'1'; --矢量与元素并置
E<=B(0)&A(1)&'0'; --元素与元素并置
```

关系运算符

VHDL 预定义的关系运算符如下所列:

- = 等于
- /= 不等于
- < 小于
- <= 小于或等于
- > 大于
- >= 大于或等于

关系运算符的作用是将相同数据类型的数据对象进行数值比较或关系排序判断, 并将结果以 boolean 类型的数据表示, 即 true 或 false。

VHDL 规定, “=” 和 “/=” 的操作对象可以是 VHDL 种任何数据类型构成的操作数; 其余关系运算符的操作对象, 则仅限于整数数据类型、枚举数据类型以及由整数型或枚举型数据类型元素构成的一维数组。

需要注意的是，“小于或等于”关系运算符“`<=`”的形式与信号赋值操作符一模一样。判别二者的关键在于其使用环境：在条件语句（如 `if_then_else`、`when` 等）中的条件式（即条件判断语句）中出现的“`<=`”是关系运算符，其它情况则是信号赋值操作符。

逻辑运算符

VHDL 共定义了 7 种逻辑运算符。

- AND 与
- OR 或
- NOT 非
- NAND 与非
- NOR 或非
- XOR 异或
- XNOR 同或

逻辑操作符的操作对象一般为以下 5 种数据类型之一：`boolean`、`bit`、`bit_vector`、`std_logic` 和 `std_logic_vector`。

虽然 `NOT` 比其它逻辑运算符的优先级高，但为了避免犯错，在写程序时仍应用括号将 `NOT` 与其对应的操作数括起来。其它逻辑运算符也应照此处理。例如：

```
A <= B AND (NOT C);
A <= (B AND C) XOR (C AND D);
A <= (NOT (B AND C)) NAND (C XOR D);
```

这样可使整个逻辑表达式层次清除，提高程序的可读性，同时方便查错。

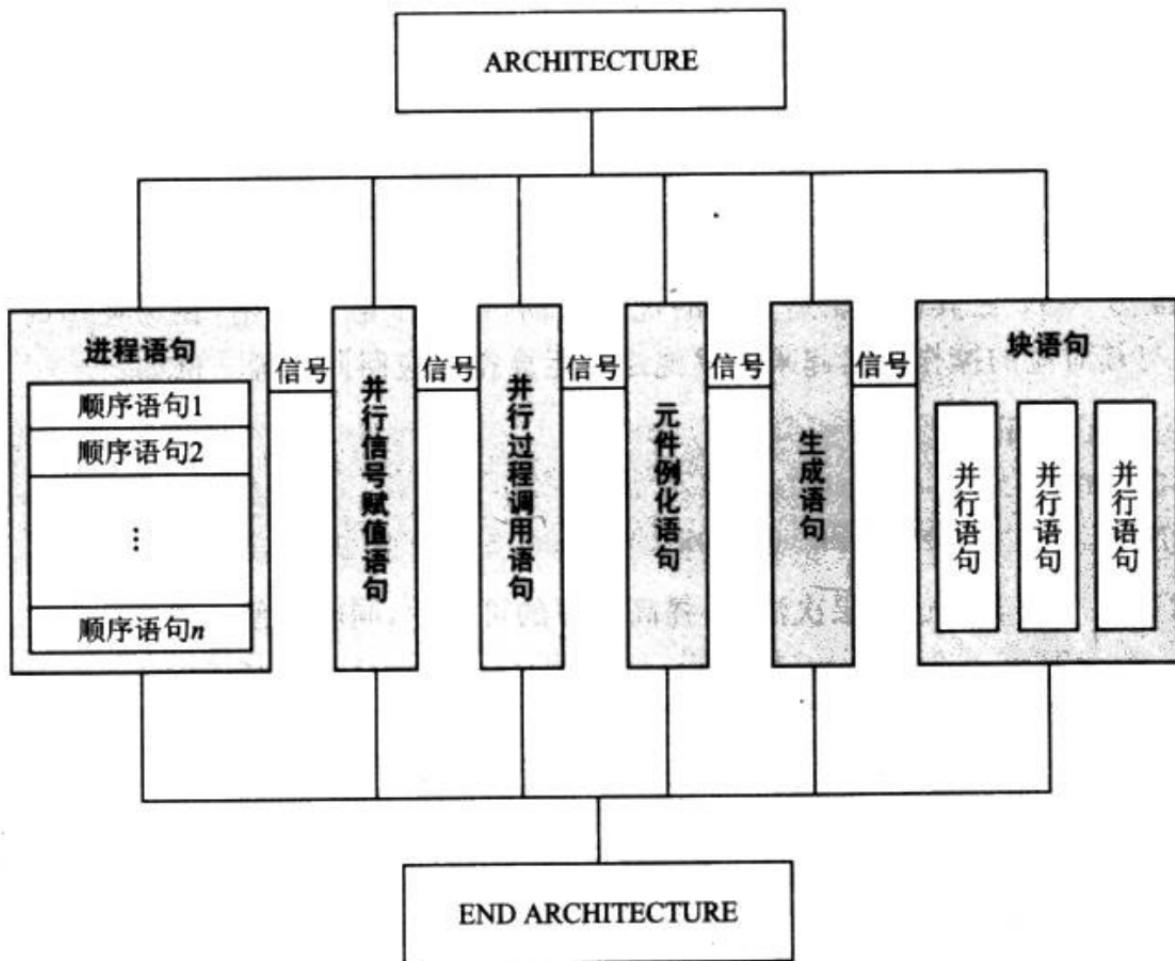
3.1.5 并行语句

并行语句（concurrent statements）是硬件描述语言区别于一般软件程序语言的最显著的特点之一。所有并行语句在结构体中的执行都是同时进行的，即它们的执行顺序与语句书写的顺序无关。

所谓“并行”，指的是这些并行语句之间没有执行顺序的先、后之分，但并不意味着并行语句内部也一定是并行方式运行的。事实上，并行语句内部的语句运行可以是并行的（如块语句），也可以是顺序的（如进程）。

VHDL 的并行语句主要有以下 6 种：

- 进程语句
- 并行信号赋值语句
- 并行过程调用语句
- 元件例化语句
- 生成语句
- 块语句



并行信号赋值语句

并行信号赋值语句又分为以下 3 种类型：

- 简单信号赋值语句
 - 选择信号赋值语句
 - 条件信号赋值语句

这 3 种信号赋值语句的赋值目标都必须是信号。下面分别介绍这几种并行信号赋值语句。

简单信号赋值语句

简单信号赋值语句它的语句格式如下：

目标信号名 <= 表达式;

因为 VHDL 是强类

选择信号赋值语句

```
with 选择表达式 select  
    赋值目标信号 <= 表达式 1 when 选择值 1,  
                  表达式 2 when 选择值 2,  
                  .
```

(下页继续)

(续上页)

```
表达式 n when others;
```

从 with_select 语句的格式不难猜出它的用法：当“选择表达式”等于某一个“选择值”时，就将其对应的表达式的值赋给目标信号；若“选择表达式”与任何一个“选择值”均不相等，则 when others 前的表达式的值赋给目标信号。

使用 with_select 语句的注意事项：

- “选择值”要覆盖所有可能的情况，若不可能一一指定，则要借助 others 为其它情况找一个“出口”；
- “选择值”必须互斥，不能出现条件重复或重叠的情况。

条件信号赋值语句

选择信号赋值语句简单、易用，但它仅对某一特定信号进行选择值的判断（所以叫“选择信号赋值语句”），当粗要对较多信号条件进行判断时，它就无能为力了。这时，则需要用到条件信号赋值语句。

条件信号赋值语句的格式如下：

```
赋值目标信号 <= 表达式 1 when 赋值条件 1 else
    表达式 2 when 赋值条件 2 else
    .
    .
    .
    表达式 n-1 when 赋值条件 n-1 else
    表达式 n;
```

在执行 when_else 语句时，赋值条件按书写的先后顺序逐项测试，一旦发现某一项赋值条件得到满足，即刻将相应表达式的值赋给目标信号，并不再测试下面的赋值条件。换言之，各赋值子句有优先级的差别，按书写先后顺序从高到低排列。

进程语句

进程语句 process 可以说是 VHDL 语言中最重要的语句之一，它的特点如下：

- 进程本身是并行语句，但其内部则为顺序语句
- 进程只有在特定的时刻（敏感信号发生变化）才会被激活

进程语句的语法格式

process 语句的语法格式有如下两种：

process 语句格式 1：

```
[进程标号:] process (敏感信号参数表)
[声明区];
begin
    顺序语句
end process [进程标号];
```

process 语句格式 2：

```
[进程标号:] process
[声明区];
begin
    wait until (激活进程的条件);
    顺序语句
end process [进程标号];
```

上面这两种语法格式是等价的，但一般只采用第 1 种语法格式，而避免使用 wait 语句。

下面对语句格式 1 种的各项进行说明：

- **进程标号**

- 简单地说，就是给进程起名。这个标号不是必须的，在大型的多个进程并存的程序中，标号可提高程序的可读性。

- **敏感信号参数列表**

- 如前所述，进程只在敏感信号发生变化的情况下被激活，而这些敏感信号就包括在敏感信号参数表中。
 - 注意：一个进程可有多个敏感信号，任一敏感信号发生变化都会激活进程，各敏感信号间以逗号隔开。

- **声明区**

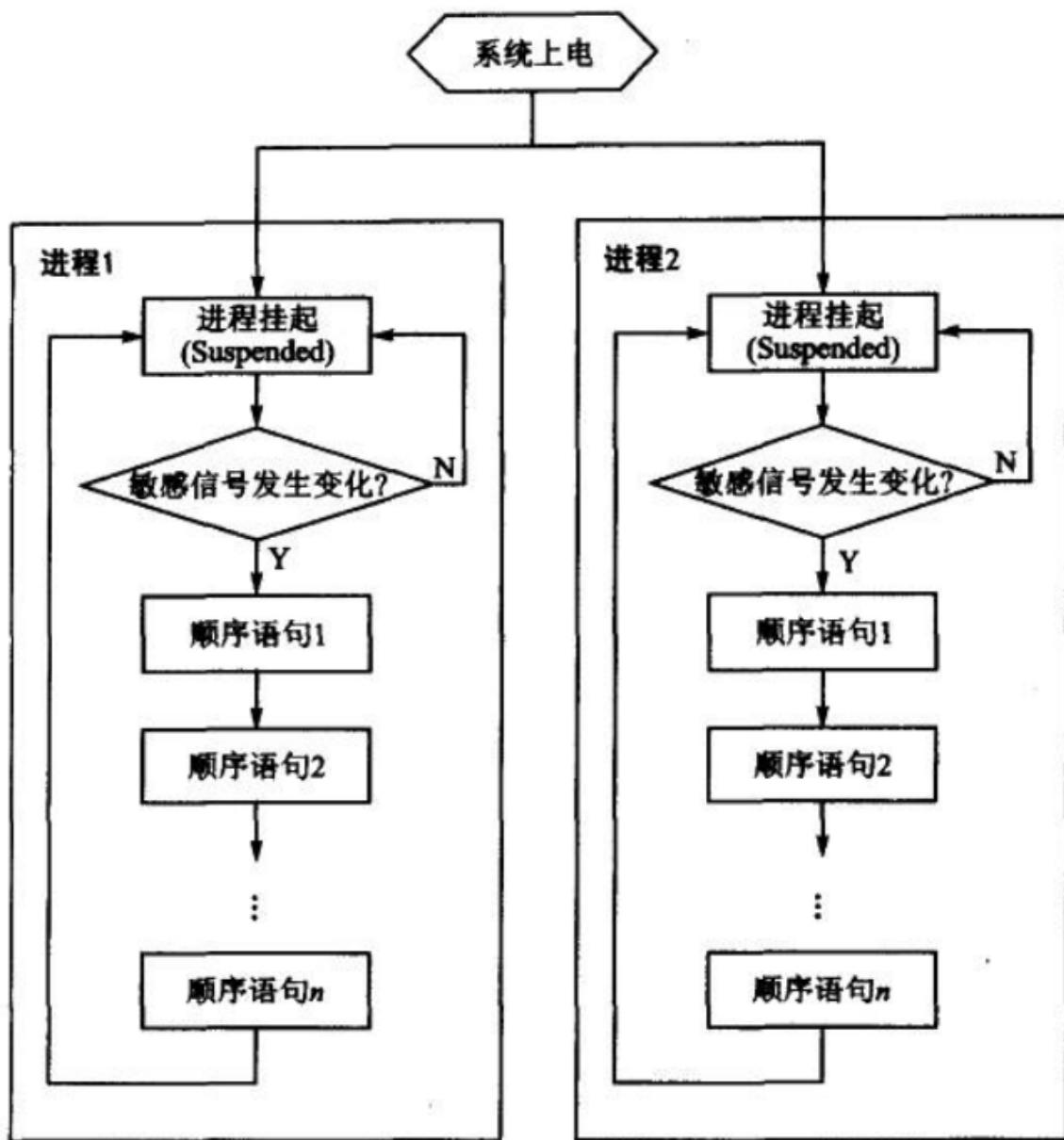
- 定义一些仅在本进程中起作用的局部量，最常在此处定义的是变量。
 - 注意：信号是全局变量，不可在此处声明。

- **顺序语句**

- 按书写顺序执行的语句，如 if_then_else 和 case 语句。
 - 注意：所谓“顺序执行”是指在仿真意义上具有一定的顺序性（并不意味着这些语句对应的硬件结构也有相同的顺序性）

进程的工作原理

下图说明了进程工作的基本原理.



- 当进程的敏感信号参数表中的任一敏感信号发生变化时，进程被激活，开始从上而下按顺序执行进程中的顺序语句；当最后一个语句执行完毕，进程刮起，等待下一次敏感信号的变化。从系统上电开始，这个过程就周而复始地进行，就像软件中的死循环。
- 虽然进程内部的语句是顺序执行的，但进程与进程之间则是并行的关系。例如，一个 architecture 中有若干个 process，颠倒各 process 在程序中的顺序并不会造成仿真与综合结果的改变。

进程与时钟

虽然进程可用来描述组合逻辑电路，但最重要的还是用它来设计时序电路（或是时序电路与组合逻辑电路的综合电路）。对于组合逻辑电路的设计，用前面所提到的一些语句和关系符就可以实现，而时序电路的设计则必须借助 process 的力量。

如果设计时序电路，就一定要对时钟有所了解，因为大多数时序电路的正常工作依赖于时钟。这里仅讨论以下两个问题：1) 时钟与进程的关系；2) 时钟沿在 VHDL 中的描述方法。

• 时钟与进程的关系

- 进程是由敏感信号的变化来启动的，因此可将时钟作为敏感信号，用时钟的上升沿或下降沿来驱动进程语句的执行。

- 在每个时钟上升沿，进程都被激活，进程中的语句被执行。有一点要特别强调：是在每个上升沿启动一次进程（执行进程内所有语句），而不是在每个上升沿执行一条语句。

• 时钟沿的 VHDL 描述方法

- 假设时钟信号的为 `clock`，且数据类型为 `std_logic`，则时钟沿在 VHDL 中的描述方法如下：
上升沿描述：`clock' event and clock= '1'`，下降沿描述：`clock' event and clock= '0'`
- 其中，`clock' event` 表示在 `clock` 信号上有事件发生（即信号发生变化）。若变化后 `clock` 值为 ‘1’，则表示时钟从 ‘0’ 变成 ‘1’，因此 `clock' event and clock= '1'` 就表示时钟上升沿。
- 除了前面的表示方法外，还有两个预定义的函数来表示时钟沿：上升沿描述 `rising_edge(clock)`，下降沿描述 `falling_edge(clock)`。

进程要点

进程有如下要点：

- 进程语句本身是并行语句，但其内部为顺序语句。
- 进程在敏感信号发生变化时被激活。
- 在同一进程中对同一信号多次赋值，只有最后一次生效。
- 在不同进程中，不可对同一信号进行赋值。
- 一个进程不可同时对时钟上、下沿敏感。
- 进程中的信号赋值是在进程挂起时生效的，而变量赋值则是即时生效的。

3.1.6 顺序语句

顺序语句是与并行语句相对而言的，其特点是：每一条顺序语句的执行顺序与其书写顺序对应，改变顺序语句的书写顺序有可能改变综合的结果。顺序语句只能出现在进程和子程序中。

前面介绍进程时提到过，所谓“顺序执行”是指仿真意义上具有一定的顺序性（或者说在逻辑上有先、后之分），并不意味着这些语句对应的硬件结构也有相同的顺序性。当顺序语句综合后，映射为实际的门电路，系统一上电，这些门电路就同时开始工作。电路可实现逻辑上的顺序执行，实际上所有门电路是并行地工作，并没有先、后之分。这种以并行的工作方式实现顺序的逻辑是硬件描述语言的一大特点，也是进程可以在被激活的瞬间执行完进程中所有语句的原因。

VHDL 中主要的顺序语句有 6 种：赋值语句、流程控制语句、空操作语句、等待语句、子程序调用语句和返回语句。

赋值语句

赋值语句包括信号赋值语句和变量赋值语句。信号赋值语句在进程与子程序之外是并行语句，在进程与子程序之内则为顺序语句；而变量赋值语句只存在于进程和子程序中。

信号和变量的一些不同之处：

- 声明形式与赋值符号不同。变量声明为 `variable`，赋值符号为 “`:=`”；而信号声明为 `signal`，带入语句采用 “`<=`” 带入符。
- 信号在结构体中、进程外定义，而变量在进程内定义。换句话说，信号的有效域为整个结构体，可在不同进程间传递数值；变量的有效域只是定义该变量的进程，不能为多个进程所用。
- 操作过程不同。在进程中，便来嗯赋值语句一旦被执行，目标变量立即被赋予新值，在执行下一条语句时，该变量的值为上一句新赋的值；而信号的赋值语句即使被执行，也不会使信号立即发生代入，下一条语句执行时，仍使用原来的信号值（信号是在进程挂起时才发生代入的）。

流程控制语句

常用的流程控制语句有 3 个：if 语句、case 语句和 loop 语句。

if 语句

if 语法格式有 3 种。

if 语法格式 1：

```
if 条件式 then
    顺序语句
end if;
```

if 语法格式 2：

```
if 条件式 then
    顺序语句
else
    顺序语句
end if;
```

if 语法格式 3：

```
if 条件式 1 then
    顺序语句
elsif 条件式 2 then
    顺序语句
    .
    .
    .
else
    顺序语句
end if;
```

以上 3 种格式的 if 语句的执行流程，与软件编程语言中的 if 语句相差无几。

- 格式 1：判断条件式是否成立。若条件成立，则执行 then 与 end if 之间的顺序语句；若条件不成立，则跳过不执行，if 语句结束。
- 格式 2：判断条件式是否成立。若条件成立，则执行 then 与 else 之间的顺序语句；若条件不成立，则执行 else 与 end if 之间额的顺序语句。
- 格式 3：自上而下逐一判断条件式是否成立。若条件成立，则执行相应的顺序语句，并不再判断其它条件式，直接结束 if 语句的执行。其执行流程与 when_else 相似。

使用 if 语句时的要点：

- if 语句可以嵌套，但层数不宜过多。
- 前面的 if 语句格式 3 和 when_case 一样，用于有优先级的条件判断，因此各条件式中的条件可以重叠。如果所需判断的条件没有优先级的差别，且条件之间没有重叠的情况，那么建议使用 case 语句。
- 用 if 语句描述异步复位信号和时钟沿时，只能用 if_elsif_end if 的格式，不能不出现 else。
- 在进程中用 if 语句描述组合逻辑电路时，务必覆盖所有的情况，否则综合后将引入所存其，违背设计初衷。

设计组合进程的要点：

- 所有在该进程中被读取的信号都必须加入敏感信号变量表；否则，当没有被包括在敏感变量表中的信号发生变化时，进程不能即时生成新的输出，综合器可能会误以为设计者希望存储数据而引入锁存器。这样就违背了组合逻辑的原则：当前输出只与当前输入有关系。
- 必须在所有条件下指定输出值。在没有指定输出值的条件下，综合器同样可能误以为设计者希望存储数据而引入锁存器。

case 语句

case 语句根据满足的条件直接选择多项顺序语句中的一项执行，其语法格式如下：

```
case 表达式 is
  when 选择值 [ | 选择值] => 顺序语句;--若多个选择值，则用“|”间隔
  when 选择值 [ | 选择值] => 顺序语句;--“=>”相当于 if 语句中的 then
  .
  .
  .
  when others => 顺序语句;
end case;
```

case 语句的使用要点：

- 选择值不可重复或重叠。例如不可同时出现两次 when “00”，也不可同时出现 when “00” | “01” 和 when “00” | “11”
- 当 case 语句的选择值无法覆盖所有情况时，要用 others 指定未能列出的其它所有情况的输出值。
- 用 case 语句设计组合进程的要点与用 if 语句设计组合进程的要点相同。

loop 语句

loop 语句是循环语句，它有 3 种常见格式 (loop_exit、for_loop 和 while_loop)，这里介绍最常用的 for_loop 语句，其语法格式如下：

```
[loop 标号:] for 循环变量 in 循环次数范围 loop
  顺序语句
end loop [loop 标号];
```

- 循环变量：这是一个临时的变量，仅在此 loop 语句种有效，因此不需要事先定义。但要注意，在 process 的声明区不要定义与此同名的变量。
- 循环次数范围：主要有两种格式，即“…to …”和“…downto …”。循环变量从循环次数范围的初值开始，每执行完一次顺序语句后递增或递减 1，直达到循环次数范围的终值为止。
- loop 标号：不是必需的，可以省略。

使用 loop 语句时，要注意循环次数范围只能用具体数值表示。

null 语句

null 语句即空操作语句。它不执行任何操作，只是让程序接着往下执行。null 语句一般用在 case 语句中，用于表示在某些情况下对输出不作任何改变。所谓的“不作任何改变”，实质上隐含了锁存信号的意思。因此在设计纯组合逻辑电路，就不要使用 null 语句。

3.1.7 VHDL 中数据类型转换与移位

signed、unsigned 以及 std_logic_vector 之间的区别

首先就是 signed 与 unsigned 这两种数据类型。他们的定义为：

```
type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
type SIGNED is array (NATURAL range <>) of STD_LOGIC;
```

与 std_logic_vector 的定义完全相同，所不同的是表示的意义不同。举例来说：

“1001”的含义对这三者而言是不同的：

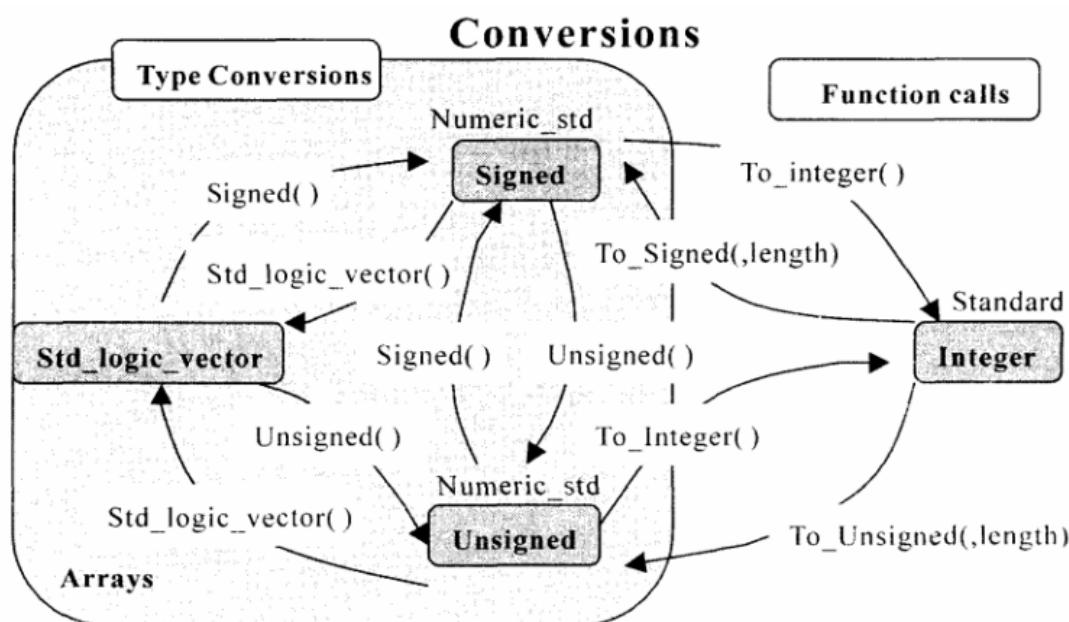
- std_logic_vector：简单的四个二进制位；
- unsigned：代表数字 9；
- signed：代表数字 -7（补码表示的）；

NUMERIC_STD

使用 NUMERIC_STD 可以完全替代 std_logic_arith、std_logic_unsigned、std_logic_signed 这三个库文件！

- 首先，NUMERIC_STD 这个库文件才是血统最正的 IEEE 库文件！！上述的其他三个其实都是 Synopsis 这个公司的，但是由于这个公司抢先了一步，所以占据了大量的用户资源。
- std_logic_arith、std_logic_unsigned、std_logic_signed 的问题在于当在同一文件中同时使用 signed 和 unsigned 时，会出现函数重载的冲突，导致错误。
- 其次，NUMERIC_STD 是完全基于 signed 和 unsigned 所写的算术重载函数和数据类型转换函数。不管是 INTEGER 还是 STD_LOGIC_VECTOR 要进行算术运算，都必须转换为 signed 和 unsigned 两种数据类型。

转换函数	功能
TO_INTEGER (ARG: UNSIGNED)	把无符号数 UNSIGNED 转换为自然数 NATURAL
TO_INTEGER (ARG:SIGNED)	把有符号数 SIGNED 转换为整数 INTEGER
TO_UNSIGNED(ARG, SIZE)	把自然数 NATURAL 转换为无符号数 UNSIGNED
TO_SIGNED (ARG, SIZE)	把整数 INTEGER 转换为有符号数 SIGNED



下面举个例子来说明 NUMERIC_STD 库的使用。

```
DOUT <= std_logic_vector(to_unsigned(0, 64));
DOUT(to_integer(unsigned(DIN))) <= '1';
```

shift_left() and shift_right()

```
r_Unsigned_L <= shift_left(unsigned(r_Shift1), 1);
r_Signed_L   <= shift_left(signed(r_Shift1), 1);

r_Unsigned_R <= shift_right(unsigned(r_Shift1), 2);
r_Signed_R   <= shift_right(signed(r_Shift1), 2);
```

3.2 verilog

3.2.1 程序结构

Verilog 的基本设计单元是模块 (module)。一个模块是由两部分组成的，一部分描述接口，一部分描述逻辑功能，用来定义怎么由输入到输出的。

Verilog 程序由三部分构成：I/O 端口声明、信号声明、功能描述。

端口声明

模块的端口声明了模块的输入输出接口。其格式如下：

```
module 模块名 (口 1, 口 2, 口 3, ...);
```

模块的端口表示的是模块的输入和输出口名，也就是说，它与别的模块联系端口的标识。在模块被引用时，在被引用的模块中，有些信号要输入到被引用的模块中，有的信号需要从被引用的模块中取出来。在引用模块时其端口可以用两种方法连接：- 在引用时，严格按照模块定义的端口顺序来连接，不用标明原模块定义的规定的端口名，例如：模块名(连接端口 1 信号名, 连接端口 2 信号名, 连接端口 3 信号名, …); - 在引用时用“.”符号，标明原模块时定义时规定的端口名，例如：模块名(.端口 1 名(连接信号 1 名),.端口 2 名(连接信号 2 名), …);。这样表示的好处在于可以用端口名与被引用模块的端口相对应，而不必严格按端口顺序对应，提高了程序的可读性和可移植性。

模块内容

模块的内容包括 I/O 说明、内部信号声明和功能定义。

I/O 说明的格式如下：

```
// 输入口
input [信号位宽-1:0] 端口名 1;
input [信号位宽-1:0] 端口名 2;
// ...
input [信号位宽-1:0] 端口名 i;//共有 i 个输入口

// 输出口
output [信号位宽-1:0] 端口名 1;
output [信号位宽-1:0] 端口名 2;
// ...
output [信号位宽-1:0] 端口名 j;//共有 j 个输出口

// 输入/输出口
inout [信号位宽-1:0] 端口名 1;
inout [信号位宽-1:0] 端口名 2;
// ...
inout [信号位宽-1:0] 端口名 k;//共有 k 个双向总线端口
```

I/O 说明也可以写在端口声明语句中。其格式如下：

```
module module_name(input port1, input port2, ..., output port1, output port2, ...);
```

在模块内部用到的与端口有关的 wire 和 reg 类型变量的声明。例如：

```
reg [width-1: 0] R 变量 1, R 变量 2;
wire [width-1: 0] W 变量 1, W 变量 2;
// ...
```

模块中最重要的部分是逻辑功能定义部分。有 3 种方法可在模块中产生逻辑。

- 用“assign”声明语句，如 assign a = b&c;

- 这种方法的句法很简单，只需写一个“assgin”，后面再加一个方程式即可。示例中的方程式描述了一个有两个输入的与门
- 用实例原件，如 **and #2 ul(q,a,b);**
 - 采用实例元件的方法像在电路图输入方式下调入库元件一样，键入元件的名字和相连的引脚即可。
 - 这表示在设计中用到一个跟与门（and）一样的名为 ul 的与门，其输入端为 a、b，输出端为 q。输出延迟为 2 个单位时间。
 - 要求每个实例元件的名字必须是唯一的，以避免与其它调用与门（and）的实例混淆。
- 用“always”块，如 **always@(posedge clk or posedge clr) begin if(clr) q<=0; else if(en) q<= d; end**

采用“assgin”语句是描述组合逻辑最常用的方法之一。而“always”块既可用于描述组合逻辑，也可描述时序逻辑。用“always”块的例子生成了一个带有异步清除端的 D 触发器。“always”块可用很多描述手段来表达逻辑，例如上例就用了 if…else 语句来表达逻辑关系。如按一定的风格来编写“always”块，可以通过综合工具把源代码自动综合成用门级结构表示的组合或时序逻辑电路。

3.2.2 数据类型及其常量和变量

verilog 中总共有 19 种数据类型。数据类型是用来表示数字电路硬件中的数据存储和传送元素的。这里介绍 4 种最基本的数据类型，它们是：reg 型、wire 型、integer 型和 parameter 型。

verilog 语言中也有常量和变量之分，它们分别属于 19 种数据类型。下面就最常用的几种进行介绍。

常量

在程序运行中，其值不能被改变的量称为常量。下面首先对在 verilog 语言中使用的数字及其表示方式进行介绍。

数字

整数

在 verilog 中，整型常量即整常数有以下 4 种进制表示形式：- 二进制整数（b 或 B）- 十进制整数（d 或 D）- 十六进制整数（h 或 H）- 八进制整数（o 或 O）

数字表达方式有以下三种：

- <位宽><进制><数字>，这是一种全面的描述方式。
- 在 <进制><数字> 这种描述方式中，数字的位宽采用默认位宽。
- 在 <数字> 这种描述方式中，采用默认进制（十进制）。

在表达式中，位宽指明了数字的精确位数。例如：一个 4 位二进制数的数字的位宽为 4，一个 4 位十六进制数字的位宽为 16（因为每单个十六进制数就要用 4 位二进制数来表示）。

例如：

```
8'b10101100 // 位宽位 8 的数的二进制表示, 'b 表示二进制
8'ha2       // 位宽为 8 的数的十六进制表示, 'h 表示十六进制
```

x 和 z 值

在数字电路中，x 代表不定值，z 代表高阻值。一个 x 可以用来定义十六进制数的 4 位二进制数的状态，八进制数的 3 位，二进制数的 1 位。z 的表示方式同 x 类似。z 还有一种表达方式是可以写作“?”。在使用 case 表达式时建议使用这种写法，以提高程序的可读性。

例如：

```
4'b10x0 // 位宽为 4 的二进制数从低位数起第 2 位为不定值
4'b101z // 位宽为 4 的二进制数从低位数起第 1 位位高阻值
12'dz   // 位宽为 12 的十进制数, 其值为高阻值 (第 1 种表达方式)
12'd?   // 位宽为 12 的十进制数, 其值为高阻值 (第 2 种表达方式)
8'h4x   // 位宽为 8 的十六进制数, 其低 4 位值为不定值
```

负数

一个数字可以被定义为负数, 只需在位宽表达式前加一个减号, 减号必须写在数字定义表达式的最前面。减号不可以放在位宽和进制之间, 也不可以放在进制和具体的数之间。

例如:

```
-8'd5 // 这个表达式代表 5 的补数 (用八位二进制数表示)
8'd-5 // 非法格式
```

下划线

下划线可以用来分隔开数的表达以提高程序的可读性。它不可以用在位宽和进制处, 只能用在具体的数字之间。

例如:

```
16'b1010_1011_1111_1010 // 合法格式
8'b_0011_1010           // 非法格式
```

当常量不说明位数时, 默认值是 32 位, 每个字母用 8 位的 ASCII 值表示。例如:

```
10=32'd10=32'b1010
1=32'd1=32'b1
-1=-32'd1=32'hFFFFFF
'BX=32'BX=32'BXXXXXX...X
"AB"=16'B01000001_01000010 // 字符串 AB, 为十六进制数 16'h4142
```

参数 (parameter) 型

在 verilog 中用 parameter 来定义常量, 即用 parameter 来定义一个标识符代表一个常量, 称为符号常量, 即标识符形式的常量, 采用标识符代表一个常量可提高程序的可读性和可维护性。parameter 型数据是一种常数型的数据, 其说明格式如下:

```
parameter 参数名 1= 表达式, 参数名 2= 表达式, ..., 参数名 n= 表达式;
```

parameter 是参数型数据的确认符。确认符后跟着一个用逗号分隔开的赋值语句表。在每一个赋值语句的右边必须是一个常数表达式。也就是说, 该表达式只能包含数字或先前已经定义过的参数。例如:

```
parameter msb = 7;      // 定义参数 msb 为常量 7
parameter e = 25, f = 29; // 定义两个常数参数
parameter r = 5.7;       // 声明 r 为一个实型参数
parameter byte_size = 8, byte_msb=byte_size-1; // 用常数表达式赋值
parameter average_delay = (r+f)/2; // 用常数表达式赋值
```

参数型常量经常用于定义延迟时间和变量宽度。在模块或实例引用时, 可通过参数传递改变在被引用模块或实例中已定义的参数。

变量

变量是一种在程序运行过程中可以改变的量, 在 verilog 中变量的数据类型有多种, 这里介绍几种常用的。网络数据类型表示结构实体之间的物理连接。网络类型的变量不能储存值, 而且它必须受到驱动器 (例如门或连续赋值语句, assign) 的驱动。如果没有驱动器连接到网络类型的变量上, 则该变量就是高阻值, 即其值为 z。

wire 型

wire 型数据常用来表示用以 assign 关键字指定的组合逻辑信号。verilog 程序模块中输入、输出信号类型默认时自动定义为 wire 型。wire 型信号可以用做任何方程式的输入，也可以用做“assign”语句或实例元件的输出。

wire 型信号的格式同 reg 型信号的格式很类似。其格式如下：

```
wire [n-1:0] 数据名 1, 数据名 2, ..., 数据名 i; // 共有 i 条总线, 每条总线内有 n 条线路
wire [n:1] 数据名 1, 数据名 2, ..., 数据名 i;
// wire 是 wire 型数据的确认符; [n-1:0] 和 [n:1] 代表该数据的位宽, 即该数据有几位; 最后跟着的是数据的名字。如果一次定义多个数据, 数据名之间之间用逗号隔开。声明语句的最后要用分号表示语句结束。例如:
```

```
wire a; // 定义了一个 1 位的 wire 型数据
wire [7:0] b; // 定义了一个 8 位的 wire 型数据
wire [4:1] c,d; // 定义了二个 4 位的 wire 型数据
```

reg 型

寄存器是数据储存单元的抽象。寄存器数据类型的关键字是 reg。通过赋值语句可以改变寄存器储存的值，其作用与改变触发器储存的值相当。verilog 语言提供了功能强大的结构语句，使设计者能有效地控制是否执行这些赋值语句。这些控制结构用来描述硬件触发条件，例如时钟的上升沿和多路器的选通信号。reg 类型数据的默认初始值位不定值 x。

reg 型数据常用来表示“always”模块内的指定，常代表触发器。通常，在设计中要由“always”模块通过使用行为描述语句来表达逻辑关系。在“always”模块内被赋值的每一个信号都必须定义成 reg 型。

reg 型数据的格式如下：

```
reg [n-1:0] 数据名 1, 数据名 2, ..., 数据名 i;
reg [n:1] 数据名 1, 数据名 2, ..., 数据名 i;
// reg 是 reg 型数据的确认标识符; [n-1:0] 和 [n:1] 代表该数据的位宽, 即该数据有几位; 最后跟着的是数据的名字。如果一次定义多个数据, 数据名之间之间用逗号隔开。声明语句的最后要用分号表示语句结束。例如:
```

```
reg rega; // 定义了一个 1 位的名为 rega 的 reg 型数据
reg [3:0] regb; // 定义了一个 4 位的名为 regb 的 reg 型数据
reg [4:1] regc, regd; // 定义了二个 4 位的名为 regc 和 regd 的 reg 型数据
```

对于 reg 型数据，其赋值语句的作用就如同改变一组触发器的存储单元的值。在 verilog 中有许多构造(construct)用来控制何时或是否执行这些赋值语句。这些控制构造可用来描述硬件触发器的各种具体情况，如触发条件时用时间的上升沿，或用来描述判断逻辑的细节，如各种多路选择器。

reg 型数据的默认初始值时不定值。reg 型数据可以赋正值，也可以赋负值。但当一个 reg 型数据时一个表达式中的操作数时，它的值被当作时无符号值，即正值。例如，当一个 4 位的寄存器用做表达式中的操作数时，如果开始寄存器被赋以值 -1，则在表达式中进行运算时，其值被认为是 +15。

memory 型

verilog 通过对 reg 型变量建立数组来对存储器建模，可以描述 RAM 型存储器、ROM 存储器和 reg 文件。数组中的每一个单元通过一个数组索引进行寻址。在 verilog 语言中没有多维数组存在。memory 型数据是通过扩展 reg 型数据的地址范围来生成的。其格式如下：

```
reg [n-1:0] 存储器名 [m-1:0];
reg [n-1:0] 存储器名 [m:1];
// reg[n-1:0] 定义了存储器中每一个存储单元的大小, 即该存储单元是一个 n 位的寄存器; 存储器名后面的 [m-1:0] 或 [m:1] 则定义了该存储器中有多少个这样的寄存器; 最后用分号结束定义语句。
```

这里通过一个例子来说明：

```
reg [7:0] mema [255:0];
// 定义了一个名为 mema 的存储器，该存储器有 256 个 8 位的存储器。该存储器的地址范围是 0 到 255。
```

3.2.3 运算符及表达式

verilog 语言的运算符范围很广，其运算符按其功能可分为以下几类：

- 算术运算符 (+,-,*,/,%)
- 赋值运算符 (=,<=)
- 关系运算符 (>,<,>=,<=)
- 逻辑运算符 (&&,||,!)
- 条件运算符 (?:)
- 位运算符 (~,|,^,&,^~)
- 移位运算符 (<<,>>)
- 拼接运算符 ({ })
- 其它

算术运算符

在 verilog 语言中，算术运算符又称为二进制运算符，共有下面几种：

- + (加法运算符，或正值运算符)
- - (减法运算符，或负值运算符)
- * (乘法运算符)
- / (除法运算符)
- % (模运算符，或称为求余运算符，要求%两侧均为整型数据。)

在进行整数除法运算时，结果值要略去小数部分，只取整数部分；而进行取模运算时，结果值的符号位采用模运算式里第一个操作数的符号位。

位运算符

verilog 作为一种硬件描述语言，是针对硬件电路而言的。在硬件电路中信号有 4 种状态值，即 1, 0, x, z。在电路中信号进行与、或、非时，反映在 verilog 中则是相应的操作数的位运算。verilog 提供了以下 5 种位运算符：

- ~ 取反
- & 按位与
- | 按位或
- ^ 按位异或
- ^~ 按位同或（异或非）

逻辑运算符

在 verilog 语言中存在 3 种逻辑运算符：

- $\&\&$ 逻辑与
- $\|$ 逻辑或
- $!$ 逻辑非

逻辑运算符中 “ $\&\&$ ” 和 “ $\|$ ” 的优先级别低于关系运算符， “ $!$ ” 高于算术运算符。

为了提高程序的可读性，明确表达各运算符元间的优先关系，建议使用括号。

关系运算符

关系运算符共有以下 4 种：

- $a < b$, 读作 a 小于 b
- $a > b$, 读作 a 大于 b
- $a \leq b$, 读作 a 小于或等于 b
- $a \geq b$, 读作 a 大于或等于 b

在进行关系运算时，如果声明的关系是假的（false），则返回值是 0；如果声明的关系是真的（true），则返回值是 1；如果某个操作数的值不定，则关系是模糊的，返回值是不定值。

所有的关系运算符有着相同的优先级别。关系运算符的优先级别低于算术运算符的优先级别。

等式运算符

在 verilog 语言中存在 4 种等式运算符：

- $==$ (等于)
- $!=$ (不等于)
- $== ==$ (等于)
- $!==$ (不等于)

“ $==$ ” 和 “ $!=$ ” 又称为逻辑等式运算符，其结果由两个操作数的值决定。由于操作数中某些位可能是不定值 x 和高阻值 z，结果可能为不定值 x。而 “ $== ==$ ” 和 “ $!==$ ” 运算符则不同，它在对操作数进行比较时对某些位的不定值 x 和高阻值 z 也进行比较，两个操作数必须完全一致，其结果才是 1，否则为 0。“ $== ==$ ” 和 “ $!==$ ” 运算符常用于 case 表达式的判别，所以又称为 “case 等式运算符”。这 4 个等式运算符的优先级别是相同的。

移位运算符

在 verilog 中有两种移位运算符：“ $<<$ ”（左移位运算符）和 “ $>>$ ”（右移位运算符）。其使用方法如下：

```
a >> n
a << n
// a 代表要进行移位的操作数, n 代表要移几位。这两种移位运算都用 0 来填补移出的空位。
```

位拼接运算符

在 verilog 语言中有一个特殊的运算符：位拼接运算符。用这个运算符可以把两个或多个信号的某些位拼接起来进行运算操作。其使用方法如下：

```
{信号 1 的某几位, 信号 2 的某几位, ..., ..., 信号 n 的某几位}
// 即把某些信号的某些位详细地列出来, 中间用逗号分开, 最后用大括号括起来表示一个整体信号。
// 例如
{a, b[3:0], w, 3'b101}
```

在位拼接表达式中不允许存在没有指明位数的信号。这是因为在计算拼接信号位宽的大小时必须知道其中每个信号的位宽。

3.2.4 赋值语句和块语句

赋值语句

在 verilog 中, 信号有两种赋值方式:

- 非阻塞赋值方式

- 在语句块中, 上面语句所赋的变量值不能立即就为下面的语句所用
- 块结束后才能完成这次赋值操作, 而所赋值的变量值是上一次赋值得到的
- 在编写可综合的时序逻辑模块时, 这是最常用的赋值方法

- 阻塞赋值方式

- 赋值语句执行完后, 块才结束
- 值在赋值语句执行完后立刻就改变的
- 在时序逻辑中使用时, 可能会产生意想不到的结果

块语句

块语句用来将两条或多条语句组合在一起, 使其在格式上看更像一条语句。通常用 begin_end 来标识顺序执行的语句, 用它来标识的块称为顺序块。块内的语句时按顺序执行的, 即只有上面一条语句执行完后下面的语句才能执行

顺序块的格式如下:

```
begin
    语句 1;
    语句 2;
    // ...
    语句 n;
end

// 或者

begin: 块名
    块内声明语句
    语句 1;
    语句 2;
    // ...
    语句 n;
end

// 块名即该块的名字, 一个标识名。块内声明语句可以是参数声明语句、reg 型变量声明语句、integer 型变量声明语句和 reg 型变量声明语句。
```

3.2.5 条件语句

if_else 语句

if 语句是用来判断所给定的条件是否满足，根据判定的结果（真或假）决定执行给出的两种操作之一。verilog 语言提供了 3 种形式的 if 语句。

```
// if(表达式) 语句
// 例如
if(a > b)
    out1 = int1;

// if(表达式)
//   语句 1
// else
//   语句 2
// 例如
if(a > b)
    out1 = int1;
else
    out1 = int2;

// if(表达式)
// else if(表达式 2) 语句 2;
// else if(表达式 3) 语句 3;
//
// ...
// else if(表达式 m) 语句 m;
// else
//   语句 n;
```

条件语句必须在过程块语句中使用。所谓的过程块是指由 initial 和 always 语句引导的执行语句集合。除这两种块语句引导的 begin end 块中可以编写条件语句外，模块汇总的其它地方都不能编写。

case 语句

case 语句是一种多分支选择语句，if 语句只有两个分支可供选择，而实际问题中常常需要用到多分支选择。verilog 语言提供的 case 语句直接处理多分支选择。它的一般形式如下：

```
case(表达式) <case 分支项> endcase
// case 分支项的一般格式如下
// 分支表达式: 语句;
// 默认项 (default 项): 语句;
```

- case 括弧内的表达式称为控制表达式，case 分支项中的表达式称为分支表达式。控制表达式通常表示为控制信号的某些位，分支表达式则用这些控制信号的具体状态值来表示，因此分支表达式又可以称为常量表达式。
- 当控制表达式的值与分支表达式的值相等时，就执行分支表达式后面的语句。如果所有的分支表达式的值都没有与控制表达式的值相匹配，就执行 default 后面的语句。
- default 项可有可无，一个 case 语句里只准有一个 default 项。
- 每一个 case 分项的分支表达式的值必须互不相同，否则就会出现问题，即对表达式的同一个值，将出现多种执行方案，产生矛盾。
- 执行完 case 分项后的语句，则跳出该 case 语句结构，终止 case 语句的执行。
- 在用 case 语句表达式进行比较的过程中，只有当信号的对应位的值能明确进行比较时，比较才能成功。因此，要注意详细说明 case 分项的分支表达式的值。
- case 语句的所有表达式的位宽必须相等，只有这样，控制表达式和分支表达式才能进行对应位的比较。

case 语句与 if_else_if 的区别主要有两点：

- 与 case 语句中的控制表达式和多分支表达式这种比较结构相比，if_else_if 结构中的条件表达式更为直观一些。

- 对于那些分支式中存在不定值 x 和高阻值 z 的位时, case 语句提供了处理这种情况的手段。

条件语句的语法

条件语句用于根据某个条件来确定是否执行其后的语句, 关键字 if 和 else 用于表示条件语句。verilog 语言共有 3 种类型的条件语句, 条件语句的用法如下所示。

```
// 第一类条件语句: 没有 else 语句
// 其后的语句执行或不执行
if( <expression> ) true_statement;

// 第二类条件语句: 有一条 else 语句
// 根据表达式的值, 决定执行 true_statement 或者 false_statement
if( <expression> ) true_statement; else false_statement;

// 第三类条件语句: 嵌套的 if_else_if 语句
// 可供选择的语句有许多条, 只有一条被执行
if( <expression1> ) true_statement1;
else if( <expression2> ) true_statement2;
else if( <expression3> ) true_statement3;
else default_statement;
```

条件语句的执行过程为: 计算条件表达式 <expression>, 如果结果为真 (1 或非零值), 则执行 true_statement 语句; 如果条件为假 (0 或者不确定值 x), 则执行 false_statement 语句。条件表达式中可以包含任何操作符。true_statement 和 false_statement 可以时一条语句, 也可以时一组语句。如果时一组语句, 则通常使用 begin、end 关键字将它们组成一个块语句。

多路分支语句

前面讲述的条件语句 if_else_if 的形式从多个选项中确定一个结果。如果选项的数目很多, 那么使用起来很不方便。而使用 case 语句来描述这种情况时非常简便的。

case 语句使用关键字 case、endcase、和 default 来表示。

```
case (expression)
    alternative1 : statement1;
    alternative2 : statement2;
    alternative3 : statement3;
    ...
    default: default_statement;
endcase
```

case 语句中的每一条分支语句都可以时一条语句或一组语句。多条语句需要使用关键字 begin_end 组合为一个块语句。在执行时, 首先计算条件表达式的值, 然后按顺序将它和各个候选项进行比较; 如果等于第一个候选项, 则执行对应的语句 statement1; 如果和全部候选项都不相等, 则执行 default_statement 语句。

3.2.6 循环语句

在 verilog 中存在着 4 种类型的循环语句, 用来控制执行语句的执行次数。

- forever: 连续的执行语句
- repeat 语句: 连续执行一条语句 n 次
- while 语句: 执行一条语句直到某个条件不满足。如果一开始条件即不满足 (为假), 则语句一次也不能被执行
- for 语句: 通过以下 3 个步骤来决定语句的循环执行
 - 先给控制循环次数的变量赋初值

- 判定控制循环的表达式的值，如为假，则跳出循环语句，如为真，则执行指定的语句后，转到第三步
- 执行一条赋值语句来修正控制循环变量次数的变量的值，然后返回第二步

forever 语句

forever 语句的格式如下：

```
forever 语句;
// 或者
forever begin 多条语句 end
```

forever 循环语句常用于产生周期性的波形，用来作为仿真测试信号。它与 always 语句不同之处在于不能独立写在程序中，而必须写在 initial 块中。

repeat 语句

repeat 语句的格式如下：

```
repeat(表达式) 语句;
// 或者
repeat(表达式) begin 多条语句 end
```

在 repeat 语句中，其表达式通常为常量表达式。

while 语句

while 语句的格式如下：

```
while(表达式) 语句;
// 或者
while(表达式) begin 多条语句 end
```

for 语句

for 语句的一般形式为：

```
for(表达式 1; 表达式 2; 表达式 3) 语句;
```

它的执行过程如下：

- 先求解表达式 1
- 求解表达式 2，若其值为真（非 0），则执行 for 语句中指定的内嵌语句，然后执行下面的第三步。若为假（0），则结束循环，转到第五步。
- 若表达式为真，在执行指定的语句后，求解表达式 3
- 转回上面的第二步骤继续执行
- 执行 for 语句下面的语句

for 语句最简单的应用形式时很容易理解的，其形式如下：

```
for(循环变量赋初值; 循环结束条件; 循环变量增值)
    执行语句;
```

for 循环语句实际上相当于采用 while 循环语句建立以下的循环结构：

```

begin
    循环变量赋初值;
    while(循环结束条件)
        begin
            执行语句;
            循环变量增值;
        end
    end

```

这样对于需要 8 条语句才能完成的一个循环控制, for 循环语句只需要两条即可。

3.2.7 生成块

生成语句可以动态地生成 verilog 代码。这一声明语句方便了参数化模块的生成。当对矢量中的多个位进行重复操作时, 或者当进行多个模块的实例引用的重复操作时, 或者在根据参数的定义来确定程序中是否应该包括某段 verilg 代码的时候, 使用生成语句能够大大简化程序的编写过程。

生成语句能够控制变量的声明、任务或函数的调用, 还能对实例引用进行全面的控制。编写代码时必须在模块中说明生成的实例范围, 关键字 generate-endgenerate 用来指定该范围。

在 verilog 中有 3 种创建生成语句的方法, 它们是:

- 循环语句
- 条件语句
- case 生成

循环生成语句

循环生成语句允许使用者对下面的模块或模块项进行多次实例引用:

- 变量声明
- 模块
- 用户定义原语、门级原语
- 连续赋值语句
- initial 和 always 块

这里用一个例子来说明如何使用生成语句对其中两个 N 位的总线用门级原语进行按位异或。在这里其目的在于说明循环生成语句的使用方法。

```

// 本模块生成两条 N 位总线变量进行按位异或
module bitwise_xor(out, i0, i1);

// 参数声明语句, 参数可以重新定义
parameter N = 32;

// 端口声明语句
output [N-1:0] out;
input [N-1:0] i0,i1;

// 声明一个临时循环变量, 该变量只用于生成块的循环计算
genvar j;

// 用一个单循环生成按位异或的异或门 (xor)
generate
    for(j=0; j<N; j=j+1)
        begin: xor_loop //xor_loop 是赋予循环生成语句的名字, 目的在于通过它对循环生成语句之中的变
量进行层次化引用。

```

(下页继续)

(续上页)

```

    xor g1(out[j], i0[j], i1[j]);
  end // 在生成块内部结束循环
endgenerate //结束生成块

// 另外一种编写形式

reg [N-1:0] out;
generate
  for(j=0; j<N; j=j+1)
    begin: bit
      always @ (i0[j] or i1[j]) out[j] = i0[j]^i1[j];
    end
endgenerate

endmodule

```

条件生成语句

条件生成语句类似于 if_else_if 的生成构造，该结构可以在设计模块中根据经过仔细推敲并确定表达式，有条件地调用（实例引用）以下这些 verilog 结构：

- 模块
- 用户定义原语、门级原语
- 连续赋值语句
- initial 或 always 块

这里用一个例子来说明如何使用条件生成语句实现参数化乘法器。如果参数 a0_width 或 a1_width 小于 8（生成实例的条件），则调用（实例引用）超前进位乘法器；否则调用（实例引用）树形乘法器。

```

// 本模块实现一个参数化乘法器
module multiplier(product, a0, a1);

// 参数声明
parameter a0_width = 8;
parameter a1_width = 8;

// 本地参数声明
// 本地参数不能用参数重新定义修改，也不能在实例引用时通过传递参数语句的方法修改
localparam product_width = a0_width + a1_width;

// 端口声明语句
output [product_width-1:0] product;
input [a0_width-1:0] a0;
input [a1_width-1:0] a1;

// 有条件地调用（实例引用）不同类型的乘法器
// 根据参数 a0_width 和 a1_width 的值，在调用时引用相对应的乘法器实例
generate
  if(a0_width<8 || a1_width<8)
    cal_multiplier #(a0_width, a1_width) m0(product, a0, a1);
  else
    tree_multiplier #(a0_width, a1_width) m0(product, a0, a1);
endgenerate

endmodule

```

case 生成语句

case 生成语句可以在设计模块中，根据仔细推敲确定多选一 case 构造，有条件地调用（实例引用）下面这些 verilog 结构：

- 模块
- 用户定义原语、门级原语
- 连续赋值语句
- initial 或 always 块

这里用一个例子来说明如何使用 case 生成语句实现 N 位加法器

```
// 本模块生成 N 位的加法器
module adder(co, sum, a0, a1, ci);
// 参数声明
parameter N = 4;

// 端口声明
output [N-1:0] sum;
output co;
input [N-1:0] a0,a1;
input ci;

// 根据总线的位宽，调用（实例引用）相应的加法器
// 参数 N 在调用（实例引用）时可以重新定义，调用（实例引用）不同位宽的加法器是根据不同的 N 来决定的
generate
  case (N)
    // 当 N=1 或 2 时分别选用位宽为 1 位或 2 位的加法器
    1: adder_1bit adder1(co, sum, a0, a1, ci);
    2: adder_2bit adder2(co, sum, a0, a1, ci);
    default: adder_cla #(N) adder3(co, sum, a0, a1, ci);
  endcase
endgenerate

endmodule
```

3.3 时序约束



- 按前面的顺序去索引，找到对应情况，按要求约束
- 开始时至配置时钟，不配置 input delays、output delays 和时序例外
- 待内容时钟完全通过后，再配置 input delays、output delays
- 时序例外是最后差不多要完工了再配置

3.3.1 create clocks

- 建立/保持时间是 D 触发器的一个固有属性
- 在时钟跳变沿到来前、后的一段时间内输入数据要稳定不变，“前”称之为建立时间，“后”称之为保持时间
- D 触发器的数据输入端必须在建立时间前到达，而且要保持到持续时间之后
- $T_{min} = T_{co} + T_{data} + T_{us}$ 寄存器传输延时、组合逻辑延时、建立时间

建立/保持时间	建立时间是电路延时的一部分
电路的延时	寄存器间最长的延时
关键路径	决定最高的时钟频率
时钟频率	提高一个电路工作频率的方法
流水线设计	

时钟定义的先后顺序

- 时钟的定义也遵从 XDC/Tcl 的一般优先级
- 同一个点上，用户定义的时钟会覆盖工具自动推导的时钟
- 后定义的时钟会覆盖先定义的时钟
- 若要二者并存，必须使用 `-add` 选项

输入时钟

输入管脚 CLK

```
create_clock -name SysClk -period 10 -waveform {0 5} [get_ports Clk]
```

```
create_clock -name SysClk -period 10 -waveform {0 5} [get_ports Clk]
```

差分时钟

使用 `create_clock` 来约束 P 端即可 (XILINX)

```
create_clock -name clk_200 -period 5 [get_ports clk_200_p]
```

GT 或恢复的时钟

```
create_clock -name txclk -period 6.667 [get_pins GT/TXOUTCLK]
```

约束 GT IP 核的输出 TXCLK

PLL 等衍生时钟

- 工具自动推导，一般无需约束
- 但建议用下面约束，可以确定时钟名，方便用其来生成其它约束

```
create_clock -name clk_200 -period 5 [get_ports clk_200_p]
```

```
Create_generated_clock -name my_clk_name [get_pins mmcm0/CLKOUT] -source [get_pins mmcm0/CLKIN] -master_clock clk_200
```

ALTERA 如想让工具自动推导，只需要下面约束

```
create_clock -name clk_200 -period 5 [get_ports clk_200_p]
```

```
derive_pll_clocks
```

自己分频时钟

```
create_clock -name CLK1 -period 5 [get_ports CKP1]
```

```
create_generated_clock -name CLK2 [get_pins REGA/Q] -source [get_ports CKP1] -divide_by 2
```

3.3.2 input delays

zheli

3.3.3 output delays

zheli

3.3.4 set timing exceptions

zheli

3.4 原语

3.4.1 Altera

TODO

3.4.2 Xilinx

Xilinx 公司的原语按照功能分为 10 类，包括：计算组件、I/O 端口组件、寄存器和锁存器、时钟组件、处理器组件、移位寄存器、配置和检测组件、RAM/ROM 组件、Slice/CLB 组件以及 G 比特收发器组件。

计算组件

计算组件指的就是 DSP48 核，也有人将其称为硬件乘法器

I/O 端口组件

I/O 组件提供了本地时钟缓存、标准单端 I/O 缓存、差分 I/O 信号缓存、DDR 专用 I/O 信号缓存、可变抽头延迟链、上拉、下拉以及单端信号和差分信号之间的相互转换

- BUFIN I/O 的本地时钟缓存
- DCIRESER FPGA 配置成功后 DCI 状态机的复位信号
- IBUF 标准容量可选择 I/O 单端口输入缓存
- IBUFDS 带可选择端口的差分信号输入缓存
- IBUFG 带可选择端口的专用输入缓存
- IBUFGDS 带可选择端口的专用差分信号输入缓存
- IDDR 用于接收外部 DDR 输入信号的专用输入寄存器
- IDELAY 专用的可变抽头输入延迟链
- IDELAYCTRL IDELAY 抽头数的控制模块
- IOBUF 带可选择端口的双向缓存
- IOBUFDS 低有效输出的三态差分信号 I/O 缓存
- ISERDES 专用 I/O 缓存的输入分解器
- KEEPER KEEPER 符号
- OBUF 单端输出端口缓存

- OBUFT 带可选择端口的低有效输出的三态输出缓冲
- OBUFDS 带可选择端口的差分信号输出缓冲
- OBUFTDS 带可选择端口的低有效输出的三态差分输出缓冲
- ODDR 用于向外部 DDR 发送信号的专用输出寄存器
- OSERDES 用于快速实现输入源同步接口
- PULLDOWN 输入端寄存器下拉至 0
- PULLUP 输入端寄存器、开路以及三态输出端口上拉至 VCC

IBUFDS

IBUFDS 原语用于将差分输入信号转化成标准单端信号，且可加入可选延迟。在 IBUFDS 原语中，输入信号为 I、IB，一个为主，一个为从，二者相位相反。

```
// IBUFDS: 差分输入缓冲器 (Differential Input Buffer)
// 适用芯片: Virtex-II/II-Pro/4, Spartan-3/3E
IBUFDS #(
  .DIFF_TERM("FALSE"),
  // 差分终端，只有 Virtex-4 系列芯片才有，可设置为 True/False
  .IOSTANDARD("DEFAULT")
  // 指定输入端口的电平标准，如果不确定，可设为 DEFAULT
) IBUFDS_inst (
  .O(O), // 时钟缓冲输出
  .I(I), // 差分时钟的正端输入，需要和顶层模块的端口直接连接
  .IB(IB) // 差分时钟的负端输入，需要和顶层模块的端口直接连接
);
// 结束 IBUFDS 模块的例化过程
```

OBUFDS

OBUFDS 将标准单端信号转换成差分信号，输出端口需要直接对应到顶层模块的输出信号，和 IBUFDS 为一对互逆操作。

```
// OBUFDS: 差分输出缓冲器 (Differential Output Buffer)
// 适用芯片: Virtex-II/II-Pro/4, Spartan-3/3E
OBUFDS #(
  .IOSTANDARD("DEFAULT")
  // 指名输出端口的电平标准
) OBUFDS_inst (
  .O(O), // 差分正端输出，直接连接到顶层模块端口
  .OB(OB), // 差分负端输出，直接连接到顶层模块端口
  .I(I) // 缓冲器输入
);
// 结束 OBUFDS 模块的例化过程
```

IOBUF

IOBUF 原语是单端双向缓冲器，其 I/O 接口必须和指定的电平标准相对应，支持 LVTTL、LVCMS15、LVCMS18、LVCMS25 以及 LVCMS33 等信号标准，同时还可通过 DRIVE、FAST 以及 SLOW 等约束来满足的不同驱动和抖动速率的需求。默认的驱动能力为 12mA，低抖动。IOBUF 由 IBUF 和 OBUFT 两个基本组件构成，当 I/O 端口为高阻时，其输出端口 O 为不定态。IOBUF 原语的功能也可以通过其组成组件的互联来实现。

```
// IOBUF: 单端双向缓冲器 (Single-ended Bi-directional Buffer)
// 适用芯片: 所有芯片
IOBUF #(
    .DRIVE(12),
    // 指定输出驱动的强度
    .IOSTANDARD("DEFAULT"),
    // 指定 I/O 电平的标准, 不同的芯片支持的接口电平可能会有所不同
    .SLEW("SLOW")
    // 制定输出抖动速率
) IOBUF_inst (
    .O(O), // 缓冲器的单元输出
    .IO(IO), // 缓冲器的双向输出
    .I(I), // 缓冲器的输入
    .T(T) // 3 态使能输入信号
);
// 结束 IOBUF 模块的例化过程
```

PULLDOWN 和 PULLUP

数字电路有三种状态：高电平、低电平、和高阻状态。有些应用场合不希望出现高阻状态，可以通过上拉电阻或下拉电阻的方式使其处于稳定状态。FPGA 的 I/O 端口，可以通过外接电阻上下拉，也可以在芯片内部，通过配置完成上下拉。上拉电阻是用来解决总线驱动能力不足时提供电流的，而下拉电阻是用来吸收电流。通过 FPGA 内部配置完成上下拉，能有效节约电路板面积，是设计的首选方案。上、下拉的原语分别为 PULLUP 和 PULLDOWN。

```
// PULLUP: 上拉原语 (I/O Buffer Weak Pull-up)
// 适用芯片: 所有芯片
PULLUP PULLUP_inst (
    .O(O),
    // 上拉输出, 需要直接连接到设计的顶层模块端口上;
);
// 结束 PULLUP 模块的例化过程

// PULLDOWN: 下拉原语 (I/O Buffer Weak Pull-down)
// 适用芯片: 所有芯片
PULLDOWN PULLDOWN_inst (
    .O(O),
    // 下拉输出, 需要直接连接到设计的顶层模块端口上
);
// 结束 PULLDOWN 模块的例化过程
```

寄存器和锁存器

寄存器和锁存器是时序电路中常用的基本元件。

- FDCPE 带有时钟使能、异步预配置和清空信号的 D 触发器
- FDRSE 带有同步时钟使能、同步预配置和清空信号的 RS 触发器
- LCDPE 带有时钟使能、异步预配置和清空信号的透明数据锁存器

FDCPE

FDCPE 指带单数据输入信号 D、单输出 O、时钟使能信号 CE、异步复位 PRE 和异步清空信号 CLR 的 D 触发器。当 PRE 信号为高时，输出端 O 为高；当 CLR 为高时，输出端 O 为低；当 PRE 和 CLR 都为低、CE 信号为高时，输入信号 D 在时钟上升沿被加载到触发器中，并被送到输出端；当 PRE 和 CLR 都为低、CE 信号为低时，输出端保持不变。

```
// FDCPE: D 触发器 (Single Data Rate D Flip-Flop )
// 适用芯片: 所有 FPGA 芯片
FDCPE #((
    .INIT(1'b0)
)) FDCPE_inst (
    .Q(Q), // 数据输出端口
    .C(C), // 时钟输入端口
    .CE(CE), // 时钟使能输入信号
    .CLR(CLR), // 异步清空输入信号
    .D(D), // 数据输入信号
    .PRE(PRE) // 异步复位输入信号
);
// 结束 FDCPE 模块的例化过程
```

时钟组件

时钟组件包括各种全局时钟缓冲器、全局时钟复用器、普通 I/O 本地的时钟缓冲器以及高级数字时钟管理模块

- BUFG 全局时钟缓冲器
- BUFGCE 全局时钟复用器，附带时钟使能信号和 0 状态输出
- BUFGCE_1 全局时钟复用缓冲器，附带时钟使能信号和 1 状态输出
- BUFGCTRL 全局时钟复用缓冲器
- BUFGMUX 全局时钟复用缓冲器，附带时钟使能信号和 0 状态输出
- BUFGMUX_1 全局时钟复用器，附带 0 状态输出
- BUFGMUX_VIRTEX4 Virtex-4 器件特有的全局时钟复用缓冲器
- BUFI0 I/O 端口本地时钟缓冲器
- BUFR I/O 端口和 CLB 的本地时钟缓冲器
- DCM_ADV 带有高级特征的数字时钟管理模块
- DCM_BASE 带有基本特征的数字时钟管理模块
- DCM_PS 带有基本特征和移相特征的数字时钟管理模块
- PMCD 匹配相位时钟分频器

BUFG

BUFG 是具有高扇出的全局时钟缓冲器，一般由综合器自动推断并使用。

全局时钟是具有高扇出驱动能力的缓冲器，可以将信号连到时钟抖动可以忽略不计的全局时钟网络，BUFG 组件还可应用于典型的高扇出信号和网络，如复位信号和时钟使能信号。如果要对全局时钟实现 PLL 或 DCM 等时钟管理，则需要手动例化该缓冲器。

```
// BUFG: 全局时钟缓存 (Global Clock Buffer)，只能以内部信号驱动
BUFG BUFG_inst (
    .O(O), // 时钟缓存输出信号
    .I(I) // 时钟缓存输入信号
);
// 结束 BUFG_ins 模块的例化过程
```

BUFMUX

BUFMUX 是全局时钟复用器，选择两个输入时钟 I0 或 I1 中的一个作为全局时钟。

当选择信号 S 为低时，选择 I0；否则输出 I1，其真值表如表 M 所示。BUFMUX 原语和 BUFMUX1 原语的功能一样，只是选择逻辑不同，对于 BUFMUX1，当选择信号 S 为低时，选择 I1；否则输出 I0。

```
// BUFGMUX: 全局时钟的 2 到 1 复用器 (Global Clock Buffer 2-to-1 MUX)
// 适用芯片: Virtex-II/II-Pro/4/5, Spartan-3/3E/3A
BUFGMUX BUFGMUX_inst (
    .O(O), // 时钟复用器的输出信号
    .I0(I0), // 0 时钟输入信号
    .I1(I1), // 1 时钟输入信号
    .S(S) // 时钟选择信号
);
// 结束 BUFGMUX_inst 模块的例化过程
```

需要注意的是：该原语只用用全局时钟处理，不能作为接口使用。

BUFIO

BUFIO 是本地 I/O 时钟缓冲器，只有一个输入与输出，非常简单。BUFIO 使用独立于全局时钟网络的专用时钟网络来驱动纵向 I/O 管脚，所以非常适合同步数据采集。BUFIO 要求时钟和相应的 I/O 必须在同一时钟区域，而不同时钟网络的驱动需要 BUFR 原语来实现。需要注意的是，由于 BUFIO 引出的时钟只到达了 I/O 列，所以不能来驱动逻辑资源，如 CLB 和块 RAM。

```
// BUFI0: 本地 I/O 时钟缓冲器 (Local Clock Buffer)
// 适用芯片: Virtex-4/5
BUFI0 BUFI0_inst (
    .O(O), // 本地 I/O 时钟缓冲器的输出信号
    .I(I) // 本地 I/O 时钟缓冲器的输入信号
);
// 结束 BUFI0 模块的例化过程
```

BUFR

BUFR 是本地 I/O 时钟、逻辑缓冲器。BUFR 和 BUFIO 都是将驱动时钟引入某一时钟区域的专用时钟网络，而独立于全局时钟网络；不同的是，BUFR 不仅可以跨越不同的时钟区域（最多 3 个），还能够驱动 I/O 逻辑以及自身或邻近时钟区域的逻辑资源。BUFIO 的输出和本地内部互联都能驱动 BUFR 组件。此外，BUFR 能完成输入时钟 1~8 的整数分频。因此，BUFR 是同步设计中实现跨时钟域以及串并转换的最佳方式。

```
// BUFR: 本地 I/O 时钟、逻辑缓冲器 (Regional Clock Buffer)
// 适用芯片: Virtex-4/5
BUFR #(
    .BUFR_DIVIDE ("BYPASS"),
    // 分频比，可选择 "BYPASS", "1", "2", "3", "4", "5", "6", "7", "8"。
    .SIM_DEVICE ("VIRTEX4")
    // 指定目标芯片, "VIRTEX4" 或者 "VIRTEX5"
) BUFR_inst (
    .O(O), // 时钟缓存输出信号
    .CE(CE), // 时钟使能信号，输入信号
    .CLR(CLR), // 时钟缓存清空信号
    .I(I) // 时钟缓存输入信号
);
// 结束 BUFR 模块的例化过程
```

需要注意的是：BUFIO 和 BUFR 只能在 Virtex-4 系列以及更高系列芯片中使用。

处理器组件

处理器组件主要包括高速以太网 MAC 控制器和 PowerPC 硬核

移位寄存器

移位寄存器组件为 Xilinx 芯片所独有，由于属性的不同，具体有 8 个。各个移位寄存器原语都是在 SRL16 的基础上发展起来的。

- SRL16 16 比特移位寄存器查找表
- SRL16_1 时钟下降作用的 16 比特移位寄存器查找表
- SRL16E 带有时钟使能信号的 16 比特移位寄存器查找表
- SRL16E_1 带有时钟使能信号且在时钟下降沿作用的 16 比特移位寄存器查找表
- SRLC16 带有进位的 16 比特移位寄存器查找表
- SRLC616_1 带有进位且在时钟下降沿作用的 16 比特移位寄存器查找表
- SRLC6E 带有时钟使能和进位信号的 16 比特移位寄存器查找表
- SRLC6E_1 带有时钟使能和进位信号，且在时钟下降沿作用的 16 比特移位寄存器查找表

SRL16

SRL16 是基于查找表 (LUT) 的移位寄存器，在实际应用中既能节省资源，还能保证时序。其输入信号 A3、A2、A1 以及 A0 选择移位寄存器的读取地址，当写使能信号高有效时，输入信号将被加载到移位寄存器中。单个移位寄存器的深度可以是固定的，也可以动态调整，最大不能超过 16。需要更大深度的移位寄存器时，则需要将多个 SRL16 拼接起来。

```
// SRL16: 16 位查找表移位寄存器 (16-bit shift register LUT operating on posedge of
// clock)
// 适用芯片: 所有 FPGA 芯片
SRL16 #(
    .INIT(16'h0000)
    // 初始化移位寄存器的值, 可以为 16 比特任意整数
) SRL16_inst (
    .Q(Q), // SRL16 的数据输出信号
    .A0(A0), // 选择 [0] 输入
    .A1(A1), // 选择 [1] 输入
    .A2(A2), // 选择 [2] 输入
    .A3(A3), // 选择 [3] 输入
    .CLK(CLK), // 时钟输入信号
    .D(D) // SRL16 的数据输入信号
);
// 结束 SRL16 模块的例化过程
```

RAM/ROM 组件

RAM/ROM 组件可用于例化 FIFO、分布式 RAM、分布式 ROM、块 RAM 以及块 ROM

- FIFO16 基于 Virtex-4 块 RAM 的内嵌 FIFO
- RAM16X1D 深度为 16，位宽为 1 的静态同步双口 RAM
- RAM16X1S 深度为 16，位宽为 1 的静态同步 RAM
- RAM32X1S 深度为 32，位宽为 1 的静态同步 RAM
- RAM64X1S 深度为 64，位宽为 1 的静态同步 RAM

- RAMB16 单口块 RAM, 位宽可配置成 1/2/4/9/18/36, 其大小可配置成 16384 比特的数据存储器, 或者 2048 的奇偶存储器
- RAMB32_S64_ECC 带有差错处理的深度为 64 位, 位宽为 64 位的同步双口块 RAM
- ROM16X1 深度为 16, 位宽为 1 的静态同步 ROM
- ROM32X1 深度为 32, 位宽为 1 的静态同步 ROM
- ROM64X1 深度为 64, 位宽为 1 的静态同步 ROM
- ROM128X1 深度为 128, 位宽为 1 的静态同步 ROM
- ROM256X1 深度为 256, 位宽为 1 的静态同步 ROM

RAM16X1S

RAM16X1S 是深度为 16 比特, 位宽为 1 的同步 RAM。当写使能信号 WE 为低时, 写端口的数据操作无效, RAM 内部的数据不会改变; 当 WE 为高时, 可以在任意地址中写入比特。为了保证数据的稳定性, 地址和数据应该在 WCLK 的上升沿前保持稳定。输出信号 O 为 RAM 中由读地址信号所确定的地址中所存数据的值。此外, 还可通过属性指定 RAM 的初始值。

```
// RAM16X1S: 16 比特 1 深度同步 RAM (16 x 1 posedge write distributed (LUT)
RAM)
// 适用芯片: 所有芯片
RAM16X1S #(
.INIT(16'h0000)
//对 RAM 的内容进行初始化, 这里初始化为全 1
) RAM16X1S_inst (
.O(O), // RAM output
.A0(A0), // RAM address[0] input
.A1(A1), // RAM address[1] input
.A2(A2), // RAM address[2] input
.A3(A3), // RAM address[3] input
.D(D), // RAM data input
.WCLK(WCLK), // Write clock input
.WE(WE) // Write enable input
);
// 结束 RAM16X1S 模块的例化过程
```

需要注意的是, RAM16X1S 原语是 Xilinx 独有的一类结构, 在小数据量存储方面非常节省资源。

Slice/CLB 组件

Slice/CLB 组件涵盖了 Xilinx FPGA 中所有的逻辑单元, 包括各种查找表、复用器以及逻辑操作等 21 个原语

- BUFCF 快速连接缓冲
- LUT1 带通用输出的 1 比特查找表
- LUT2 带通用输出的 2 比特查找表
- LUT3 带通用输出的 3 比特查找表
- LUT4 带通用输出的 4 比特查找表
- LUT1_D 带两个输出的 1 比特查找表
- LUT2_D 带两个输出的 2 比特查找表
- LUT3_D 带两个输出的 3 比特查找表
- LUT4_D 带两个输出的 4 比特查找表
- LUT1_L 带本地输出的 1 比特查找表

- LUT2_L 带本地输出的 2 比特查找表
- LUT3_L 带本地输出的 3 比特查找表
- LUT4_L 带本地输出的 4 比特查找表
- MULT_AND 用于乘法的快速与门
- MUXCY 带有进位和通用输出的 2 到 1 复用器
- MUXCY_D 带有进位和双输出的 2 到 1 复用器
- MUXCY_L 带有进位和本地输出的 2 到 1 复用器
- MUXF5 基于查找表的 2 到 1 复用器, 带通用输出
- MUXF5_D 基于查找表的 2 到 1 复用器, 带双输出
- MUXF5_L 基于查找表的 2 到 1 复用器, 带本地输出
- MUXF6 基于查找表的 2 到 1 复用器, 带通用输出
- MUXF6_D 基于查找表的 2 到 1 复用器, 带本地输出
- MUXF6_L 基于查找表的 2 到 1 复用器, 带本地输出
- MUXF7 基于查找表的 2 到 1 复用器, 带通用输出
- MUXF7_D 基于查找表的 2 到 1 复用器, 带本地输出
- MUXF7_L 基于查找表的 2 到 1 复用器, 带本地输出
- MUXF8 基于查找表的 2 到 1 复用器, 带通用输出
- MUXF8_D 基于查找表的 2 到 1 复用器, 带本地输出
- MUXF8_L 基于查找表的 2 到 1 复用器, 带本地输出
- XORCY 带通用输出进位逻辑的 XOR
- XORCY_D 带两个输出进位逻辑的 XOR
- XORCY_L 带本地输出进位逻辑的 XOR

CHAPTER 4

硬件介绍

4.1 LUPO

这里应该有图片。

4.2 DT5495

<https://www.caen.it/products/dt5495/>



更多中文详细信息, 请访问 <http://wuhongyi.cn/CAENx495/index.html>

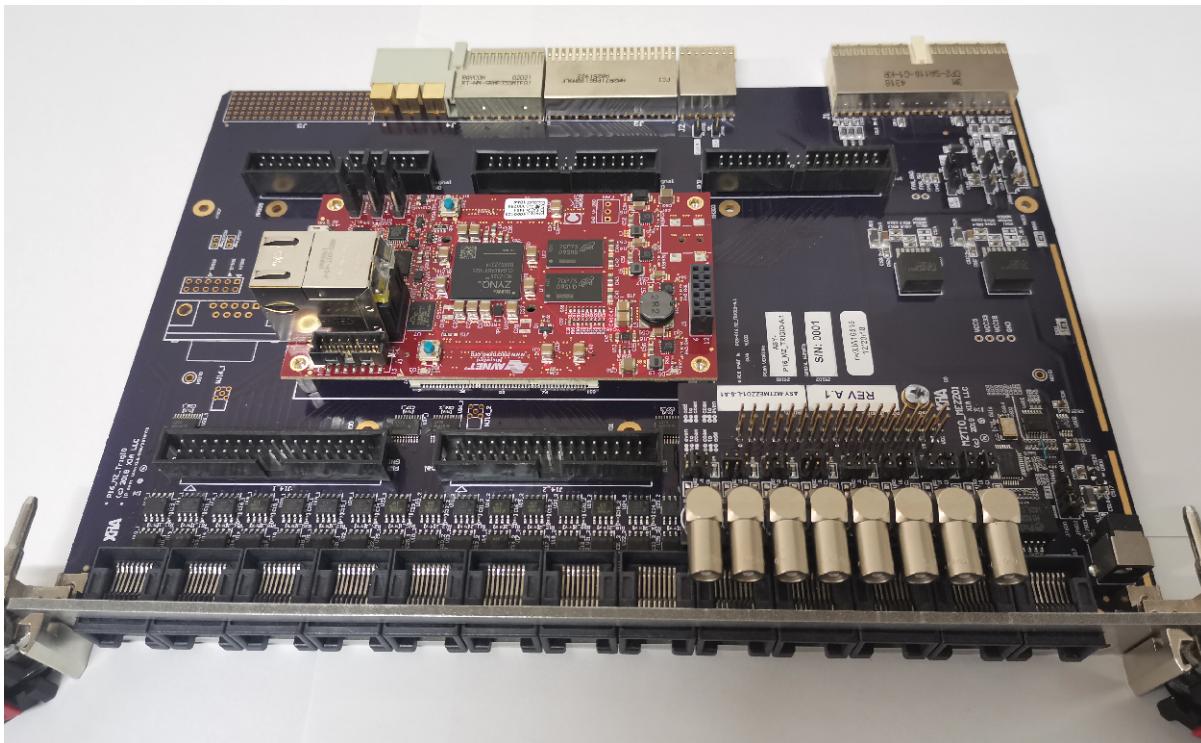
4.3 MZTIO

Pixie-16 MZ-TrigIO 设计用于将信号从背板（后连接器）连接到前面板（前连接器），并在 FPGA 架构中实现逻辑组合。它具有以下功能和特性：

- 用于 Pixie-16 的以太网可编程触发/符合控制模块
- 48+ Pixie-16 背板触发连接到本地 Zynq 处理器
- 48 个前面板 LVDS 连接到本地 Zynq 处理器
- 带嵌入式 Linux 的 MicroZed Zynq 处理器，作为独立 PC，内置 SD 卡驱动器，USB 主机，10/100 以太网，网络服务器等
- 1588 PTP 和 SyncE 时钟同步
- 开源用户访问软件和固件
- 用作独立桌面设备或 6U PXI 机箱

- 通过子卡自定义 I/O 标准

更多详细信息, 请访问 <http://wuhongyi.cn/MZTIO/>



4.4 DT5550

<https://www.caen.it/products/dt5550/>

- 80MS/s, 14bit ADC

4.5 R5560

<https://www.caen.it/products/r5560se/>

- 128 channels, 14-bit @125 MS/s Digitizer

计数器

5.1 计数器

5.1.1 计数器规则

计数器规则 1: 计数器逐一考虑三要素-初值、加 1 条件、结束条件

任何计数器都有三个要素：初值、加 1 条件、结束值

- 初值：计数器的默认值或者开始计数的值
- 加 1 条件：计数器执行加 1 条件
- 结束值：计数器计数周期的最后一个值设计计数器，要逐一考虑这三个要素，一般是先考虑初值，再考虑加 1 条件，最后再考虑结束值。

计数器规则 2: 计数初值必须为 0

计数器的默认值和开始值一定要为 0。这是我们规范的统一要求。我们知道一般编程语言计数都是从 0 开始的，0 表示第 1 个，1 表示第 2 个，。。。这里我们也参考这种做法，计数器都从 0 开始计数。

所有计数器都统一从 0 开始计数，有助于我们阅读理解、方便使用，从而不用从头看具体代码，就能清楚这个数值的含义。

计数器规则 3: 使用某一计数值，必须同时满足加 1 条件

计数器从 0 开始计数，计数器的默认值，也就是复位值也是 0。当计数器值为 0 时，如何判断这是计数器的第一个值还是还没开始计数的默认值呢？

可以通过加 1 条件来判断。当加 1 条件无效时，计数器值为 0 表示未开始计数的默认值；当加 1 条件有效时，计数器值为 0 表示计数器的第 1 个值。以此类推，当 $cnt==x-1$ 时，不表示数到 x ；只有当 $cnt==x-1$ 时，并且加 1 条件有效时，才表示数到 x 。

例如：当加 1 条件为 `add_cnt`，且 `add_cnt && cnt==4` 时，表示计数到 5 个；而当 `add_cnt==0 && cnt==4` 时，不表示计数到 5 个。

计数器规则 4：结束条件必须同时满足加 1 条件，且结束值必须是 x-1 的形式。

计数器的结束条件必须同时满足加 1 条件。例如假设要计数 5 个，那么结束值是 4，但是结束条件不是 `cnt==4` 而是 `add_cnt && cnt==4`。因为 `cnt==4` 不表示计数到 5 个，只有 `add_cnt && cnt==4` 时，才表示计数到 5 个。

为了更好地阅读代码，我们这里规定结束值必须是 $x-1$ 的形式，即 `add_cnt && cnt==4` 要写成 `add_cnt && cnt==5-1`。这里的“5”表示希望计算的个数，“-1”则是固定格式。有了这个约定后，计数的边界就很明确了。

计数器规则 5：当取某个数时，assign 形式必须为：(加 1 条件) && (cnt== 计数值-1)

当要从计数器取某个数时，例如要取计数器的第 5 个点，就很容易写成 `cnt==5-1`，这是不正确的。正确的写法时：(加 1 条件) && (cnt== 计数值-1)，如 `add_cnt && cnt==5-1`。

计数器规则 6：结束后必须回到 0

每轮计数周期结束后，计数器变回 0，这是为了使计数器能够循环重复计数。

计数器规则 7：若需要限定范围，则推荐使用 “>=” 和 “<” 两种符号

设计时，考虑边界值通常要花费一些心思，而且容易出错。为此，我们约定：若需要限定范围，则推荐使用“ \geq ”和“ $<$ ”两种符号。例如要取前 8 个数，那么就取 `cnt>=0 && cnt<8`。注意，一定是“大与或等于”和“小于”符号，而不使用“大与”和“小于或等于”符号。

该规则参考编程里的 for 循环语句。假如要循环 8 次，for 循环的条件通常会写成“`i==0; i<8; i++`”，前面的 0 表示开始值，后面的 8 表示循环次数。当然，也可以写成“`i==0; i<=7; i++`”，但是这数字的意义就实在令人费解了，虽然知道 7 是从 8-1 得来的，但多一个“-1”的思考，就纯属画蛇添足了。

计数器规则 8：设计步骤

设计步骤：先写计数器的 always 段，条件用名字代替；然后用 assign 写出加 1 条件；最后用 assign 写出结束条件

我们的计数器模版代码包括三段。

第一段，写出计数器的 process/always 段

VHDL

```
process(clk, rst_n)
begin
    if(rst_n = '0')then
        cnt <= 0;
    elsif(clk'event and clk = '1')then
        if(加 1 条件)then
            if(结束条件)then
                cnt <= 0;
            else
                cnt <= cnt + 1;
            end if;
        end if;
    end if;
end process;
```

verilog

```
always @ (posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        cnt <= 0;
    end
    else if (加 1 条件) begin
        if (结束条件)
            cnt <= 0;
        else
            cnt <= cnt + 1;
    end
end
```

大家有没有发现上述模版的特点？这个模版只需要填两项内容：加一条件和结束条件。如果为加 1 条件和结束条件定义一个信号名，例如 add_cnt 和 end_cnt，则代码变成：

VHDL

```
process (clk, rst_n)
begin
    if (rst_n = '0') then
        cnt <= 0;
    elsif (clk'event and clk = '1') then
        if (add_cnt) then
            if (end_cnt) then
                cnt <= 0;
            else
                cnt <= cnt + 1;
            end if;
        end if;
    end if;
end process;
```

verilog

```
always @ (posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        cnt <= 0;
    end
    else if (add_cnt) begin
        if (end_cnt)
            cnt <= 0;
        else
            cnt <= cnt + 1;
    end
end
```

第二段，用组合逻辑写出加 1 条件

在此阶段，只需要想好一个点，就是计数器的加 1 条件。假设计数器的加 1 条件为 $a==2$ ，则代码如下：

VHDL

```
add_cnt <= a=2; --add 1
```

verilog

```
assign add_cnt = a==2; //add 1
```

第三段，用组合逻辑写出结束条件

在此阶段，只需要想好一个点，就是计数器的结束值。参考计数器规则 5 的要求，结束条件的形式一定是：(加 1 条件) && (cnt== 计数值-1)。假设计数器要计数 10 个，则代码如下：

VHDL

```
end_cnt <= add_cnt and (cnt = 10-1); --end
```

verilog

```
assign end_cnt = add_cnt && cnt == 10-1; //end
```

至此，就完成了计数器代码的设计。总结一下这段代码的特点：每次值考虑一件事，按这要求去做，就非常容易完成代码设计。

以下是我们完整的模版：

VHDL

```
signal cnt : integer range 0 to ;--max number
signal add_cnt : boolean;
signal end_cnt : boolean;

process(clk,rst_n)
begin
    if(rst_n = '0')then
        cnt <= 0;
    elsif(clk'event and clk = '1')then
        if(add_cnt )then
            if(end_cnt)then
                cnt <= 0;
            else
                cnt <= cnt + 1;
            end if;
        end if;
    end if;
end processand (cnt = -1);--end
```

以上模版中，只需要补充三个地方，

```
signal cnt : integer range 0 to ;--在 to 后面补充计数器的最大计数范围
add_cnt <= ;--补充加 1 条件
end_cnt <= add_cnt and (cnt = -1);--补充计数器数多少数
```

verilog

```
reg [ :0]      cnt      ;
wire          add_cnt;
wire          end_cnt;

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        cnt <= 0;
    end
    else if(add_cnt) begin
        if(end_cnt)
            cnt <= 0;
        else
            cnt <= cnt + 1;
    end
end

assign add_cnt = ;//condition: add 1
assign end_cnt = add_cnt && cnt == -1; //End condition, last value
```

以上模版中，只需要补充三个地方，

```

reg [ :0] cnt ; //补充计数器的最大计数范围
assign add_cnt = ; //补充加 1 条件
assign end_cnt = add_cnt && cnt == -1; //补充计数器数多少数

```

计数器规则 9：加 1 条件必须与计数器严格对齐，其它信号一律向计数器对齐

我们设计出计数器，但一般计数器不是最终的目的，最终的目的是输出各种信号。设计计数器是为了方便产生这些输出信号（包括中间信号），并能从计数器获取变化条件。例如：信号 dout 在计数到 6 时拉高，则其变 1 的条件是：add_cnt && cnt==6-1。

假设有两个信号：dout0 在计数到 6 时拉高；dout1 在计数到 7 时拉高。一种做法是 dout0 变 1 的条件是 add_cnt && cnt==6-1，dout1 变 1 的条件是 dout0==1。这个 dout1 就是间接与计数器对齐。这是非常不好的方法。这里我们建议一律向计数器对齐，dout1 变 1 的条件应该为 add_cnt && cnt==7-1。

计数器规则 10：命名必须符合规范

比如：add_cnt 表示加 1 条件；end_cnt 表示结束条件

如无特别说明，计数器的命名都要符合规范，加 1 条件的前缀为 “add_”，结束条件的前缀为 “end_”。

CHAPTER 6

状态机

6.1 状态机

6.1.1 状态机规则

状态机规则 1：使用四段式写法

四段式不是指四个 always 代码，而是四段代码。另外需要注意的是，四段式状态机并非固定不变。如果没有输出信号就只有三段代码（两个 always）；如果有多个输出信号，那么就会有多个 always。

第一段，同步时序的 always 模块，格式化描述次态迁移到现态寄存器。

VHDL

```
-- code here  
xxxxx;
```

verilog

```
always@(posedge clk or negedge rst_n)begin  
    if(!rst_n)begin  
        state_c<= IDLE;  
    end  
    else begin  
        state_c<= state_n;  
    end  
end
```

第二段，组合逻辑的 always 模块，描述状态转移判断条件。注意，转移条件用信号来表示，信号名要按规则来命名。

VHDL

```
-- code here  
xxxxx;
```

verilog

```

always@(*)begin
    case(state_c)
        IDLE:begin
            if(idle2s1_start)begin
                state_n = S1;
            end
            else begin
                state_n = state_c;
            end
        end
        S1:begin
            if(s12s2_start)begin
                state_n = S2;
            end
            else begin
                state_n = state_c;
            end
        end
        S2:begin
            if(s22idl)begin
                state_n = IDLE;
            end
            else begin
                state_n = state_c;
            end
        end
        default:begin
            state_n = IDLE;
        end
    endcase
end

```

第三段, 用 assign 定义转移条件。注意, 条件一定要加上现态。

VHDL

```
-- code here
xxxxx;
```

verilog

```

assign idle2s1_start = state_c==IDLE && ;
assign s12s2_start = state_c==S1 && ;
assign s22idl_start = state_c==S2 && ;

```

第四段, 设计输出信号。规范要求一段 always 代码设计一个信号, 因此有多少个输出信号就有多少段 always 代码。

VHDL

```
-- code here
xxxxx;
```

verilog

```

always @(posedge clk or negedge rst_n)begin
    if(!rst_n)begin
        out1 <= 1'b0
    end
    else if(state_c==S1)begin
        out1 <= 1'b1;
    end

```

(下页继续)

(续上页)

```

else begin
    out1 <= 1'b0;
end
end

```

状态机规则 2：四段式状态机第一段写法不变

设计状态机时所有四段式状态机模板的第一段除了名字外的代码都可以直接用，不需要进行改动。

状态机规则 3：第二段的状态转移条件用信号来表示

设计状态机时，要求四段式状态机的第二段中用信号名来表示转移条件，而无须直接写出具体的转移条件。

用信号名表示的好处是后续修改时只需改动信号的名字，并且方便根据状态机的命名修改对应的跳转条件。

状态机规则 4：用 assign 将状态转移条件写成 XX2XX_start 的形式

状态机规则 3 要求转移条件用信号名来表示，这样一来设计就要编写很多信号名称，这也是设计工作中的一大困扰。因此制定此规则：将状态转移的条件信号用 xx2xx_start 的形式表示。

例如有三个状态 IDLE、READ、WRITE，若从 IDLE 跳转到 READ 状态，其跳转条件可以命名为：idle2read_start；若从 IDLE 跳转到 WRITE，其跳转条件可以命名为：idle2write_start。

这个命名方式既能够解决命名的困扰，又能直接从信号名看出信号的作用。

状态机规则 5：assign 定义状态转移条件信号时，必须加上当前状态

状态机的第二段代码中使用信号名来表示转移条件，在代码后则需用 assign 对相应信号进行定义。注意，定义这个转移条件信号时必须加上当前状态，以避免因两个不同状态由同一种变化条件发生转移而导致错误。

VHDL

```
-- code here
xxxxx;
```

verilog

```

assign idle2s1_start = state_c==IDLE && XX;
assign s12s2_start = state_c==S1 && XX;
assign s22idl_start = state_c==S2 && XX;

```

状态机规则 6：状态不变时使用 state_n = state_c

编写状态机代码时有很大一部分错误是复制粘贴过程出错造成的，很多会出现复制其它状态的代码时忘记修改状态的错误。此外，也有一部分写第二段状态机时，容易把状态保持不变写成 state_n=state_n。这个写法是错误的，因为组合逻辑只有锁存器才能有保持电路，而数字电路中通常不希望出现锁存器。

为此，这里规定，其四段式状态机的第二段，状态不变时使用 state_n=state_c。如下所示，可以自行对比。这样写不但可以减少出错的可能，还可以减少调试的时间。

VHDL

```
-- code here  
xxxxx;
```

verilog

```
IDLE:begin  
    if(idle2s1_start)begin  
        state_n = S1;  
    end  
    else begin  
        state_n = state_c;  
    end  
end  
  
IDLE:begin  
    if(idle2s1_start)begin  
        state_n = S1;  
    end  
    else begin  
        state_n = IDLE;  
    end  
end
```

7.1 FIFO

FIFO (First Input First Output)，即先入先出队列。在数字电路设计中所说的 FIFO 实际上是指 FIFO 存储器，主要用于数据缓存和异步处理，当然 FIFO 存储器缓存数据也遵循先入先出的原则。

FIFO 本质上是一个 RAM，它与普通存储器的区别是没有外部读/写地址线，这样使用起来非常简单，但缺点是只能顺序写入数据，顺序读出数据，其数据地址由内部读/写指针自动加 1 完成，不能像普通存储器那样可以由地址线决定读取或写入某个指定的地址。

7.1.1 FIFO 规则

FIFO 规则 1：使用 Show-ahead 都模式

根据 FIFO 的读模式，一般有两种使用模式：Show-ahead 和 Normal 模式。这两种模式的区别在生成 FIFO IP 核步骤中说明。

其中 Normal 模式是读使能有效后的下一拍读出相应数据。而 Show-ahead 模式是先进行数据输出，在读使能有效时对 FIFO 输出数据进行更新。即 FIFO 中的第一个数据输出在总线上，在读使能信号到来的下一拍直接输出第二个数据。

推荐使用 Show-ahead 模式，因为在这种模式下可以将读使能信号与读出数据当做有效信号和数据来使用，只要读使能有效则对应的数据就始终有效。

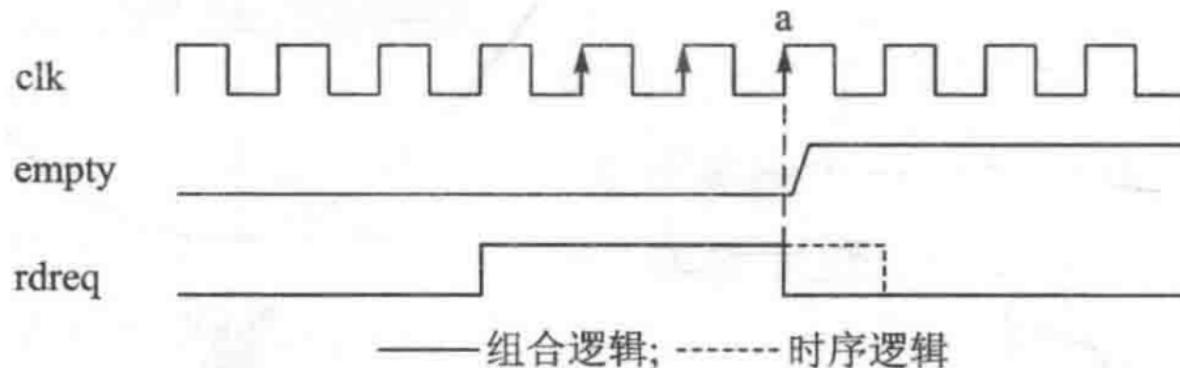
FIFO 规则 2：读、写隔离规则

读、写隔离规则是指：读控制和写控制是独立的，它们之间除了用 FIFO 交流信息外，不能有任何信息传递。因此，既不能根据 FIFO 的读状态或者读出的数据来决定写行为，也不能根据 FIFO 的写状态和写入的数据来决定读行为。

FIFO 规则 3：读使能必须判断空状态，并且用组合逻辑产生

rdreq 必须由组合逻辑产生，原因与 empty 有关。

通过下图来说明 rdreq 与 empty 的关系。使用 Show-ahead 模式，假设 FIFO 中存有 3 字节数据，现对 FIFO 进行读操作，要把 FIFO 中存的所有数据读出来。



从图中可以看出, 由于电路存在一定延时, 在 a 时刻 FIFO 已经为空, empty 变为高电平需要经历一段时间。如果用时序逻辑产生 rereq, 在 a 时刻 empty 为 0, 则表示 FIFO 中还有数据 (实际上已经为空), 因此 rereq 还要保持一个时钟周期。在 FIFO 为空的情况下要再读取一个数据, 读操作会出错。如果用组合逻辑产生 rereq, 当 empty 为 1 时, rdreq 马上拉低, 图中的实线部分波形, 就不会出现读取空 FIFO 的错误。

FIFO 规则 4: 处理报文时, 把指示信号与数据一起存入 FIFO

FIFO 不仅能保存 “数据”, 也能保存 “指示信号”, 因此可以将数据和对应的 “指示信号值” 一起写入 FIFO。

假设一个 8 位宽度的数据, 如生成一个 10 位宽度的 FIFO, 保存到 FIFO 的数据是 {din, din_sop, din_eop}。通过这种方式保存后, 就很容易产生 dout_sop 和 dout_eop 信号了, 从 FIFO 中读到的数据, 就可以用来判断报文的开始和结束, 从而用于其它判断等。

FIFO 规则 5: 读、写时钟不同时, 必须用异步 FIFO

FIFO 按时钟分可以分为同步 FIFO 和异步 FIFO。

同步 FIFO: 指读时钟和写时钟都相同的 FIFO。同步 FIFO 内部没有异步处理, 因此结构简单, 资源占用较少。

异步 FIFO: 指读时钟和写时钟可以不同的 FIFO。异步 FIFO 内部有专门的异步处理电路, 处理读、写信号的交互, 因此异步 FIFO 结构复杂, 占用资源较大。

高层次综合

8.1 高层次综合

利用 FPGA 进行算法实现已经被广泛认知,但对于很多没有 FPGA 和 HDL 设计经验的开发者而言,往往又觉得开发门槛较高,因此全球相关的科研和工程人员都在致力于如何将 FPGA 技术介绍给更多的开发者,使更多人从 FPGA 的并行性、高性能、低功耗、灵活配置中获益。其中,Vivado HLS(高层次综合)就是一个成功的代表。通过 Vivado HLS 工具中,开发者可利用 C/C++ 语言对 FPGA 进行编程,这项技术已经趋于成熟,在实验核物理中也已广泛采用。

根据 Vivado HLS 的使用指南,我们将对我们的输入程序作出以下规范:

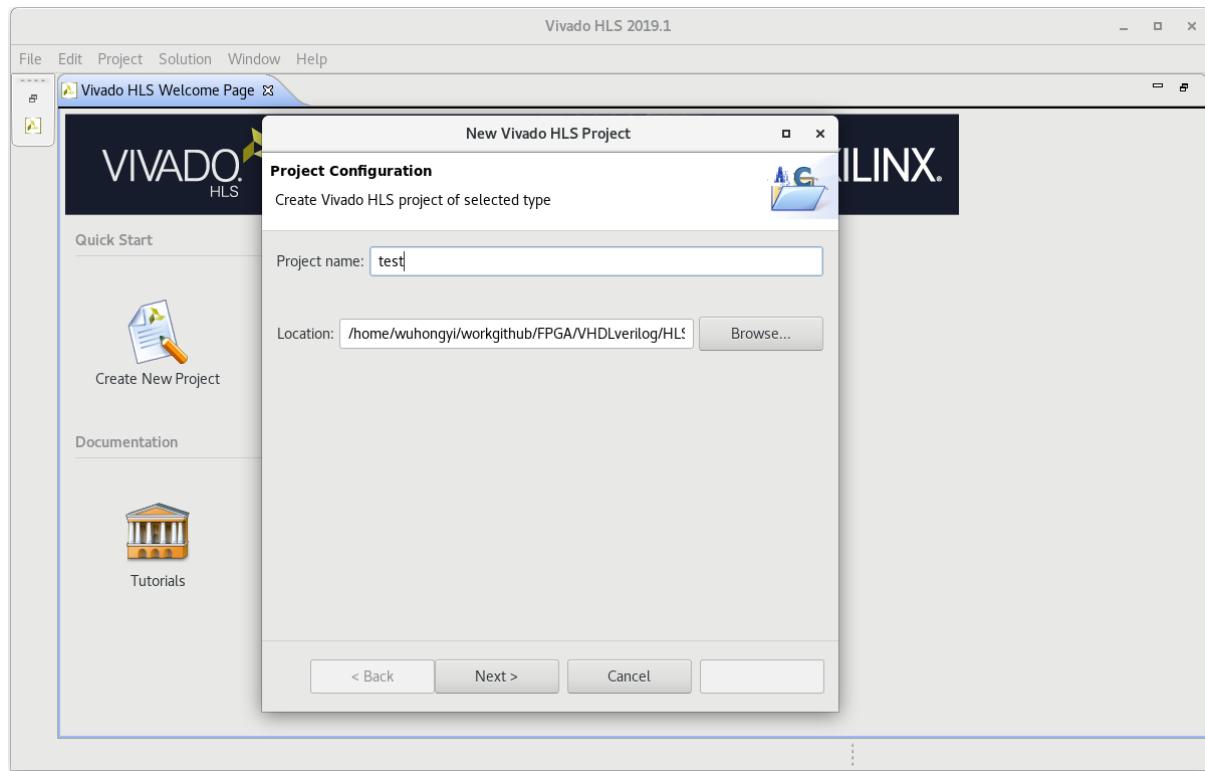
- 不使用动态内存分配(不使用 malloc(),free(),new 和 delete())
- 减少使用指针对指针的操作
- 不使用系统调用(例如 abort(),exit(),printf()), 我们可以在其它代码例如测试平台上使用这些指令,但是综合的时候这些指令会被无视(或直接删掉)
- 减少使用其它标准库里的内容(支持 math.h 里常用的内容,但还是有一些不兼容)
- 减少使用 C++ 中的函数指针和虚拟函数
- 不使用递归方程
- 精准的表达我们的交互接口

8.1.1 创建一个高级合成项目

打开 Vivado HLS 图形用户界面(GUI), 在欢迎页面中, 选择创建新项目以打开项目向导。

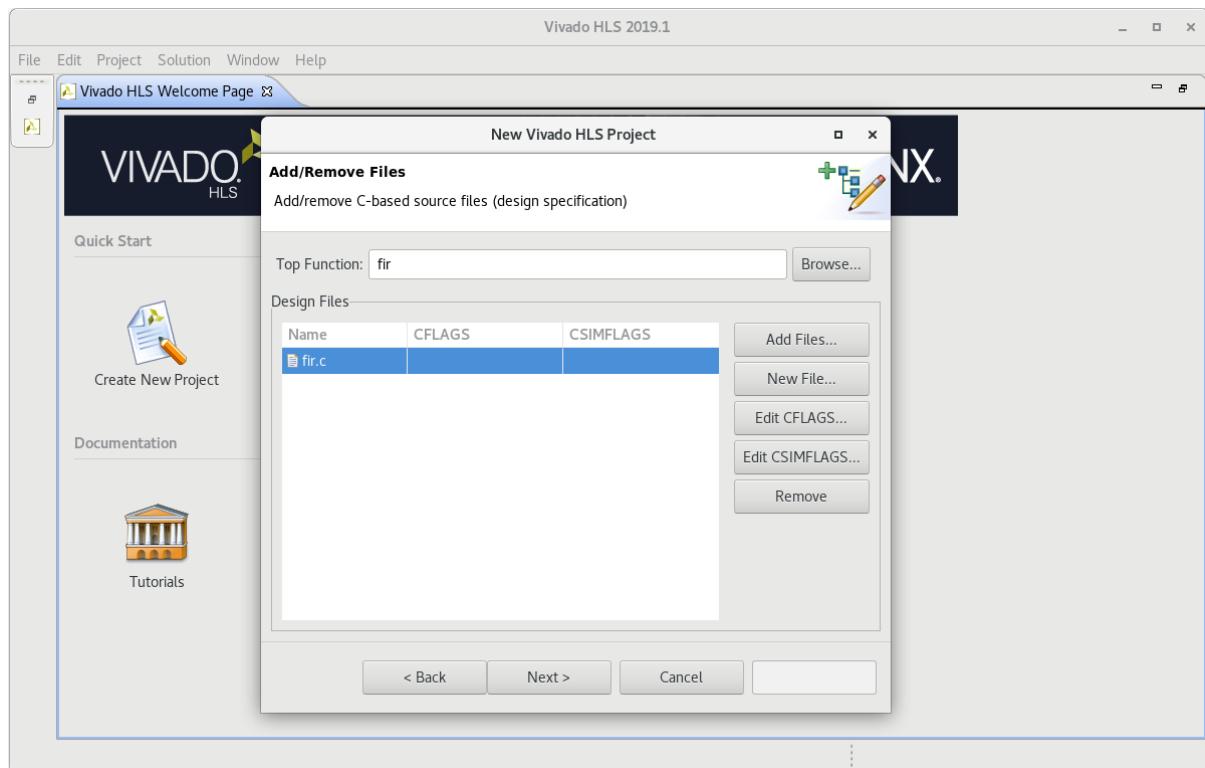


输入项目名称；单击 Browse 导航到目录的位置；选择目录并单击 OK；单击 Next。



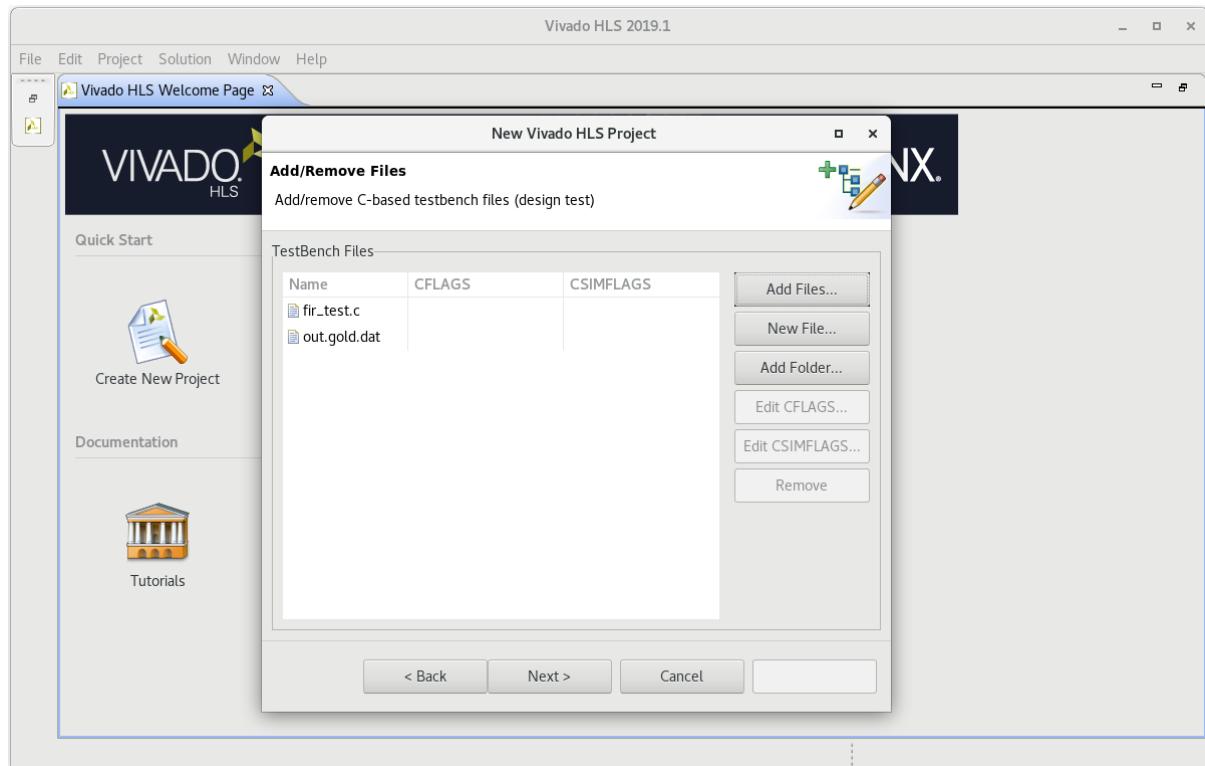
输入以下信息来指定 C 设计文件: 点击添加文件；选择 (fir.c) 并单击 OK；使用 Browse 按钮指定顶级函数 (fir.c) 功能；单击 Next。

在这个项目里只有一个 C 设计文件。当有多个 C 文件要合成时，您必须在此阶段将它们全部添加到项目中。存在于本地目录中的任何头文件都会自动包含在项目中。



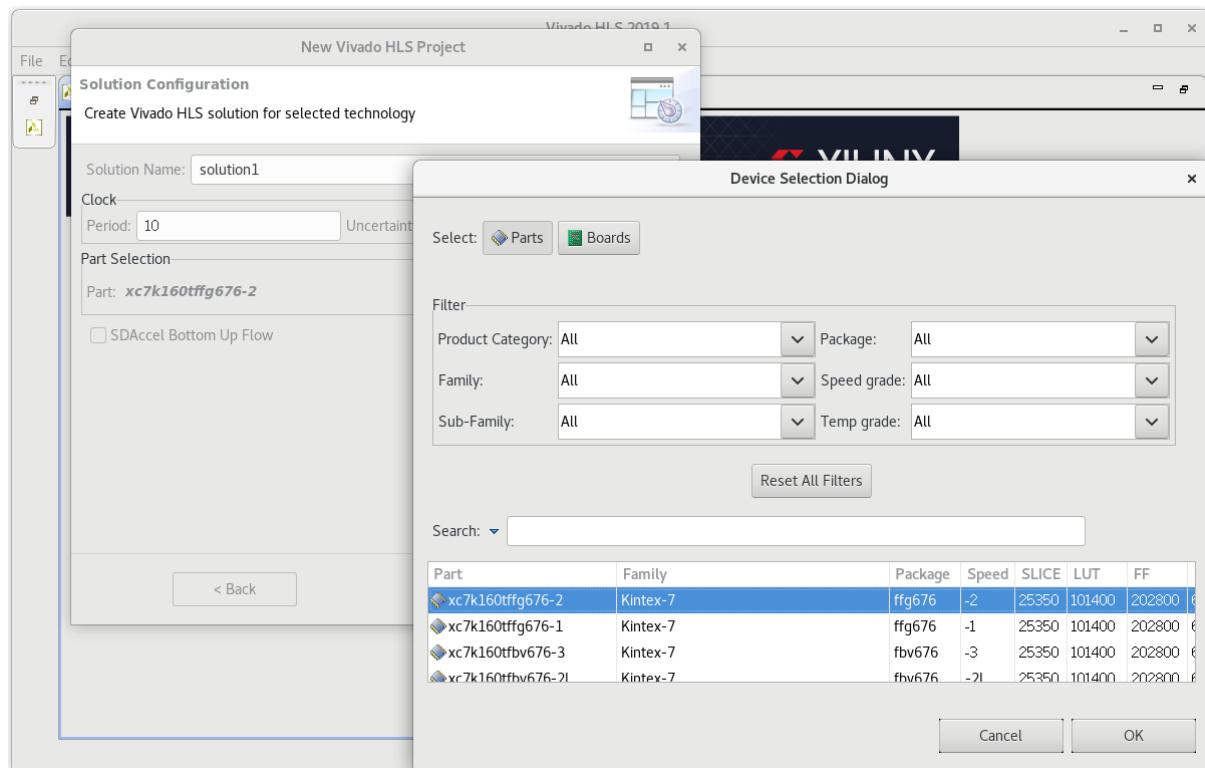
单击 Add Files 按钮来包含测试工作台文件 (fir_test.c) 和 out.gold.dat。单击 Next。

测试工作台和测试工作台使用的所有文件(头文件除外)必须包括在内。如果您没有包含测试工作台使用的所有文件(例如,由测试工作台读取的数据文件),可能会因为无法找到数据文件而失败。

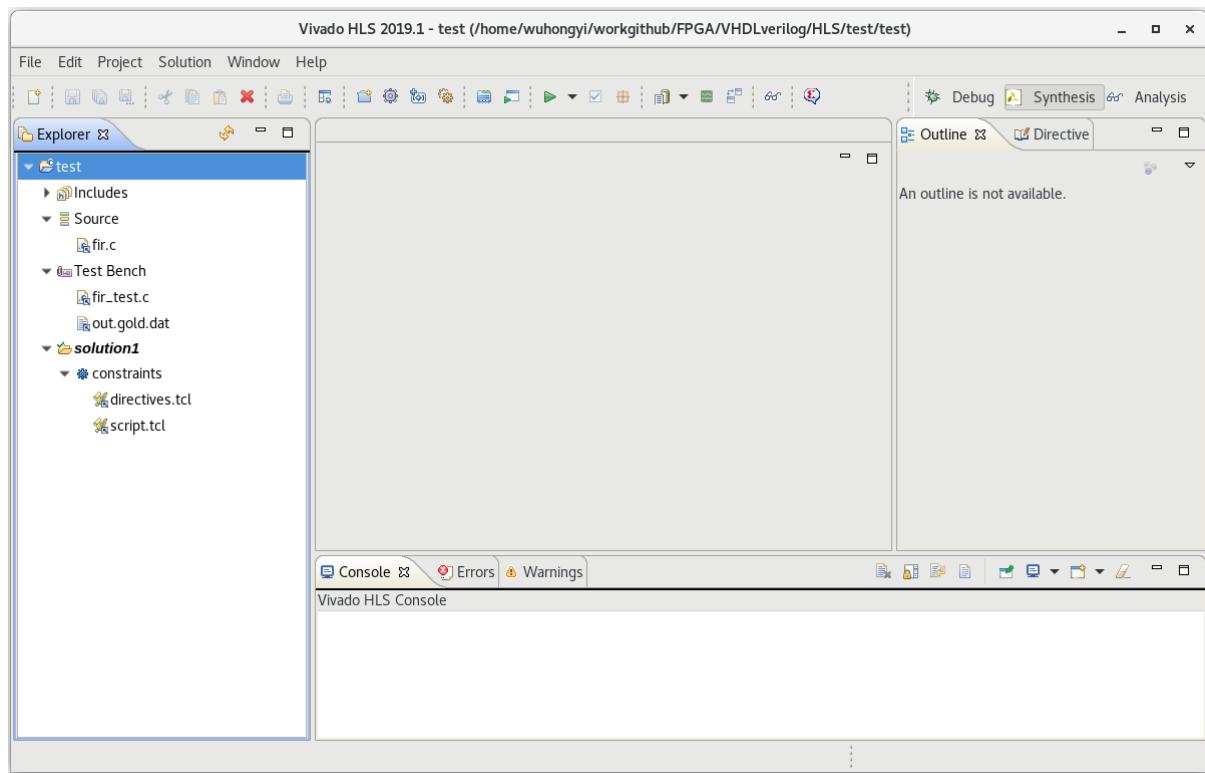


解决方案配置窗口指定第一个解决方案的技术规范。一个项目可以有多个解决方案,每个解决方案使用不同的目标技术、包、约束和/or综合指令。

接受默认的解决方案名称 (solution1)、时钟周期 (10ns) 和时钟不确定性(默认为 12.5% 的时钟周期,留空/未定义)。单击器件选择按钮,打开器件选择窗口。从可用设备列表中选择设备。单击 OK。

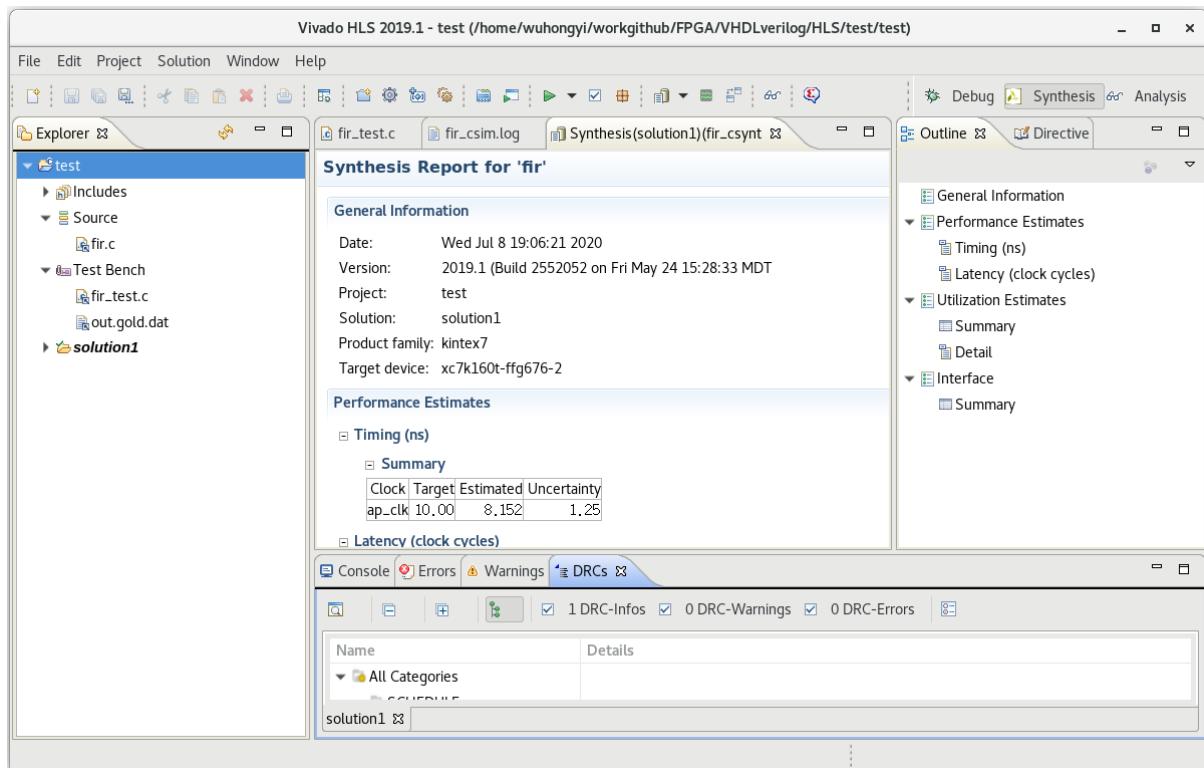


单击 Finish 打开 Vivado HLS 项目



HLS 项目的第一步是确认 C 代码是正确的。这个过程称为 C 验证或 C 仿真。在这个项目中, 测试台将函数的输出数据与已知的好值进行比较。

高级合成步骤中, 您将 C 设计合成为 RTL 设计, 并查看合成报告。单击运行 C 合成工具栏按钮。合成完成后, 报告文件将自动打开。



8.1.2 综合创建的端口

输入的原始程序如下：

```
void fir (data_t *y, coef_t c[N], data_t x)
{
    static data_t shift_reg[N];
    acc_t acc;
    data_t data;
    int i;

    acc=0;
Shift_Accum_Loop:
    for (i=N-1;i>=0;i--)
    {
        if (i==0)
        {
            shift_reg[0]=x;
            data = x;
        }
        else
        {
            shift_reg[i]=shift_reg[i-1];
            data = shift_reg[i];
        }
        acc+=data*c[i];
    }
    *y=acc;
}
```

综合生成的 verilog 代码接口如下：

```
module fir (
    ap_clk,
```

(下页继续)

(续上页)

```

    ap_rst,
    ap_start,
    ap_done,
    ap_idle,
    ap_ready,
    y,
    y_ap_vld,
    c_address0,
    c_ce0,
    c_q0,
    x
);














```

综合生成的 VHDL 代码接口如下：

```

entity fir is
port (
    ap_clk : IN STD_LOGIC;
    ap_rst : IN STD_LOGIC;
    ap_start : IN STD_LOGIC;
    ap_done : OUT STD_LOGIC;
    ap_idle : OUT STD_LOGIC;
    ap_ready : OUT STD_LOGIC;
    y : OUT STD_LOGIC_VECTOR (31 downto 0);
    y_ap_vld : OUT STD_LOGIC;
    c_address0 : OUT STD_LOGIC_VECTOR (3 downto 0);
    c_ce0 : OUT STD_LOGIC;
    c_q0 : IN STD_LOGIC_VECTOR (31 downto 0);
    x : IN STD_LOGIC_VECTOR (31 downto 0));
end;

```

- 设计有一个时钟和复位端口 (ap_clk, ap_reset)，这些都与设计本身的源对象相关联。
- 有与设计相关的额外端口。自动添加了一些块级控制端口: ap_start、ap_done、ap_idle 和 ap_ready。
- 变量 output y 现在是一个 32 位的数据端口，带有一个相关的输出有效信号指示器 y_ap_vld。
- 输入参数 c(一个数组)已经被实现为一个块 RAM 接口。具有 4 位输出地址端口、一个输出 CE 端口和一个 32 位输入数据端口。
- 标量输入参数 x 被实现为一个没有 I/O 协议 (ap_none) 的数据端口。

高级综合可以重用 C 测试平台, 通过仿真验证 RTL。

- 单击 Run C/RTL CoSimulation 工具栏按钮
- 在 C/RTL 协同仿真对话框中单击 OK 以执行 RTL 仿真。RTL 联合仿真的默认选项是使用 Vivado 模拟器和 Verilog RTL 执行仿真。要使用不同的模拟器或语言执行验证, 请使用 C/RTL 联合模拟对话框中的选项。

- 当 RTL 联合模拟完成时, 报告将在信息窗格中自动打开。这是在 C 仿真结束时产生的相同消息。
- C 测试平台为 RTL 设计生成输入向量。
- 对 RTL 设计进行了仿真。
- 将 RTL 的输出向量应用回 C 测试台中, 测试台中的结果检查验证结果是否正确。
- Vivado HLS 表明, 如果测试工作台返回的值为 0, 模拟就通过。它是测试台中返回变量的值, 仅这一点就表示模拟是否成功。重要的是, 测试工作台仅在结果正确时才返回值 0。

高级综合流程的最后一步是将设计打包为 IP 块, 以便与 Vivado 设计套件中的其他工具一起使用。

- 单击 Export RTL 工具栏按钮
- 确保格式选择下拉菜单显示 IP 目录。
- 单击 OK。IP 打包程序为 Vivado IP 目录创建一个包。
- 在浏览器中扩展解决方案 1。
- 展开 Export RTL 命令创建的 impl 文件夹。
- 展开文件夹, 找到打包成 zip 文件的 IP, 准备添加到 Vivado IP 目录

经验总结

9.1 经验总结

9.1.1 计数器

- 画出输入、输出波形（根据功能要求、画出输入和输出的波形）
- 画出计数器结构
- 确定加 1 条件（计数器数什么，加 1 条件不足则加 flag_add）
- 确定计数器结束条件（数多少个，个数不同时，用变量法，即用 x 替代；x 不足以区分时则加 flag_sel）
- 如果有更新波形
- 其它信号变化点条件（其它信号：即输出或内部信号；变化点：0 变 1、1 变 0）
- 写出计数器代码
- 写出其它信号代码

9.1.2 状态机

- 明确功能
- 输出分析
- 状态合并
- 状态转移条件
- 转移条件
- 完整性检查
- 状态机代码
- 功能代码

9.1.3 FIFO

- FIFO 的写使能写数据，同时用组合逻辑或者同时用时序逻辑。
- FIFO 的读使能，用组合逻辑。
- 数据的输出用组合逻辑。

9.1.4 波形

- 时序逻辑的波形观看方法：时钟上升沿前看输入，时钟上升沿后看输出。
- 组合逻辑的波形观看方法：输入变输出即刻变。

9.1.5 抄书

与门

与门是具有两个或多个输入端和一个输出端的逻辑门。当所有的输入都是逻辑 1 时，与门输出为逻辑 1。换句话说，如果任何输出为 0 则输出为 0。

从原理上说，我们可以定义具有许多输入输入端口的与门，但在具体实现时还是有实际限制的。一般来说与门的输入端最多为 4 个或 5 个，如果需要更多的输入端口，那么可以将多个与门级联起来。另外，在实际应用中，通常更倾向于使用与非门，其逻辑功能端进行逻辑与后再取反输出，主要原因是采用 CMOS 工艺实现的与非门工作速度比与门快得多。

扇入扇出

逻辑门的扇入指的是一个逻辑门正常工作时输入端的数量，例如，理论上一个与门可以有 20 个输入端，但是包含 20 个输入端的与门在工作时可能会因为输入负荷过大而出现逻辑错误或者速度下降的情况，此时扇出就不能选为 20。门电路的扇出与具体的电路制造工艺和电路结构有密切关系，进行集成电路设计时，电路单元库中会给出扇入的具体参数。

扇出是在不降低输出电平的情况下逻辑门能够驱动的负载的数量。例如，从理论上讲一个与门可以驱动 20 个以上的输入端，但此时门电路的输出电容负载非常大，电路的工作速度会严重下降。集成电路单元库中会给出不同类型门电路的扇出。综合工具进行 RTL 代码综合时，会从单元库中读取扇入和扇出参数，以确保不超过最大值要求。

通用 D 触发器

D 触发器有数据、时钟、和 RST# 输入端以及 Q 和 !Q 两个输出端。在每一个时钟的上升沿，输出 Q 将输入 D 的值锁存，直到下一个时钟上升沿出现时才继续锁存当前 D 端的值。!Q 输出的值与 Q 输出的值相反。在时钟上升沿进行输出数据更新时，D 端的输入数据必须满足称为建立时间的定时要求，否则输出端 Q 可能会出现不确定值。

建立时间

在时钟上升沿值之前 D 需要保持稳定的最短时间称为建立时间。如果在建立时间内 D 的值发生了变化，那么将无法确定 Q 的电平，其可能为一个不确定的电平值。

保持时间

在时钟的上升沿之后的一段时间内，D 的输入值也不允许改变，否则也会造成 Q 输出的不稳定，这个窗口被称为保持时间。

亚稳态

当输入 D 在建立时间和保持时间窗口内发生变化时，在此后的几乎一个时钟周期内。输出电平既不是 0 也不是 1，处于不确定值。这种不稳定的状态也被称为亚稳态。亚稳态的输出将在下一个时钟的上升沿之前稳定为 0 或 1。如果亚稳态输出被用于其它逻辑门的输入，那么将会造成难以预计的不良影响，可能会造成连锁反应，使整个数字系统工作不稳定。因此，必须采取一定的设计手段避免 D 触发器进入亚稳态，或者避免亚稳态被传递，影响整个系统的稳定性。

当触发器的输入不满足建立时间和保持时间要求时，输出为亚稳态。为了使系统正常工作，必须采取一定的手段避免或消除其影响。在只有一个时钟的数字系统中（称为单时钟域数字系统），通过控制一个D触发器的输出到另一个D触发器输入之间组合逻辑门的数量。可以减少其带来的延迟从而避免D触发器的输入在建立时间和保持时间窗口内发生波动。但是，当一个数字系统中有两个或两个以上时钟时（称为多时钟域系统），会出现一个时钟域的D触发器的输出作为另一个时钟域的D触发器输入的情况，当两个时钟之间没有任何关联时，亚稳态的出现时无法避免的。

信号同步规则

当信号从一个时钟域进入另一个时钟域时，为了使信号正确传递同时保持系统工作稳定，必须遵循以下几条设计规则。

- 跨时钟域的信号必须直接来自源时钟域的寄存器输出。
 - 如果信号1来自组合逻辑，而不直接来自触发器，可能会造成信号在目标时钟域中出现难以预料的不稳定情况，从而造成整个系统出现无法预测的问题。
- 使用逻辑单元库中的专用触发器实现两级同步器。
 - 这里所说的专用触发器与普通触发器有所不同，它们具有高驱动能力和高增益，这会使它们比常规的触发器更快地进入稳定状态。根据前面的分析，将两个或多个触发器级联起来可以减少亚稳态出现的概率，那么采用这种专用触发器，可以大大提高电路从亚稳态中摆脱出来的速度。
- 在一个点而不是在多个点上进行跨时钟域信号的同步。
 - 同步电路可以消除亚稳态及其传递，但得到的结果可能是1也可能是0，当只有一个连接点时，最多是信号延迟不同的问题，如果是多个点，那么这些信号组合之后的结果可能性非常多，这会造成信号传递的错误，可能会导致下游系统出错。

事件/边沿检测

同步上升沿检测

```
input sig_a;
reg sig_a_d1;
wire sig_a_risedge;

always @(posedge clk or negedge rstb)
begin
  if(!rstb) sig_a_d1 <= 1'b0;
  else sig_a_d1 <= sig_a;
end

assign sig_a_risedge = sig_a & !sig_a_d1;
```

同步下降沿检测

```
input sig_a;
reg sig_a_d1;
wire sig_a_faledge;

always @(posedge clk or negedge rstb)
begin
  if(!rstb) sig_a_d1 <= 1'b0;
  else sig_a_d1 <= sig_a;
end

assign sig_a_faledge = !sig_a & sig_a_d1;
```

同步上升/下降沿检测

```
input sig_a;
reg sig_a_d1;
wire sig_a_anymedge;
```

(下页继续)

(续上页)

```

always @(posedge clk or negedge rstb)
  begin
    if(!rstb) sig_a_d1 <= 1'b0;
    else sig_a_d1 <= sig_a;
  end

assign sig_a_anymedge = (!sig_a & sig_a_d1) | (sig_a & !sig_a_d1);

```

异步输入上升沿检测

```

input sig_a; //domain clka
input clk_b;
input rstb;
reg sig_a_d1, sig_a_d2, sig_a_d3;
wire sig_a_posedge;

assign sig_a_posedge = sig_a_d2 & !sig_a_d3;

always @(posedge clk_b or negedge rstb)
  begin
    if(!rstb)
      begin
        sig_a_d1 <= 1'b0;
        sig_a_d2 <= 1'b0;
        sig_a_d3 <= 1'b0;
      end
    else
      begin
        sig_a_d1 <= sig_a;
        sig_a_d2 <= sig_a_d1;
        sig_a_d3 <= sig_a_d2;
      end
  end

```

同步技术

数据同步和在不同时钟域之间进行数据传输会经常出现。为避免任何差错、系统故障和数据破坏，正确的同步和数据传输就显得格外重要。这些问题的出现往往比较隐蔽，不易被发现，因此正确进行跨时钟域处理就显得极为重要。实现数据同步有许多种方式，在不同的情况下进行恰当的同步方式选择非常重要。这里简单介绍目前常用的两种同步技术。

使用 FIFO 进行的数据同步

当存在两个异步时钟域并且二者之间进行数据包传输时，双端口 FIFO 最为合适。FIFO 有两个端口，一个端口写入输入数据，另一个端口读出数据。两个端口工作在相互独立的时钟域内，通过各自的指针（地址）来读写数据。由于每个端口工作在相互独立的时钟域内，因此读写操作可以独立实现并且不会出现任何差错。当 FIFO 变满时，应停止写操作，直到 FIFO 中出现空闲空间，同样，当 FIFO 为空时，应停止读操作，直到有新的数据被写入 FIFO 中。FIFO 有满标记和空标记，有关 FIFO 操作的详细描述在相应章节给出。

握手同步方式

FIFO 可用于在不同的时钟域之间进行数据包的传输，但是在一些应用中需要在不同时钟域之间进行少量数据传输。FIFO 占用的硬件资源较大，此时可以考虑使用握手同步机制。

以下是握手同步机制的工作步骤：

- 用后缀 **_t** 表示发送端，用后缀 **_r** 表示接收端。发送时钟用 **tclk** 表示，接收时钟用 **rclk** 表示。数据从 **tclk** 域向 **rclk** 域传输；
- 当需要发送的数据准备好后，发送端将 **t_rdy** 信号置为有效，该信号必须在 **tclk** 下降沿时采样输出；
- 在 **t_rdy** 有效期间，**t_data** 必须保持稳定；

- 接收端在 rclk 域中采用双同步器同步 t_rdy 控制信号，并把同步后的信号命名为 t_rdy_rclk；
- 接收端在发现 t_rdy_rclk 信号有效时，t_data 已经安全地进入了 rclk 域，使用 rclk 对其进行采样，可以得到 t_data_rclk。由于数据已经在 rclk 域进行了正确采样，所以此后在 rclk 域使用该数据是安全的；
- 接收端将 r_ack 信号置为 1，信号必须在 rclk 下降沿输出；
- 发送端通过双同步器在 tclk 域内同步 r_ack 信号，同步后的信号称为 r_ack_tclk；
- 以上所有步骤称为“半握手”。这是因为发送端在输出下一数据之前，不会等到 r_ack_rclk 被置为 0；
- 半握手机制工作速度快，但是，使用半握手机制时需要谨慎，一旦使用不当，会导致操作错误；
- 从低频时钟域向高频时钟域传数据时，半握手机制较为适用，这是由于接收端可以更快地完成操作。然而，如果从高频时钟域向低频时钟域传输数据，则需要采用全握手机制；
- 当 r_ack_tclk 为高电平时，发送端将 t_rdy 置为 0；
- 当 t_rdy_rclk 为低电平时，接收端将 r_ack 置为 0；
- 当发送端发现 r_ack_tclk 为低电平后，全握手过程结束，传输端可以发送新的数据；
- 显然，权握手过程耗时较长，数据传输速度较慢。然而，全握手机制稳定可靠，可以在两个任意频率的时钟域内安全地进行数据传输。

全握手机制代码如下：

```
// Verilog RTL for Full Handshake -Transmit
module handshake_tclk
(
    tclk,
    resetb_tclk,
    t_rdy,
    data_avail,
    transmit_data,
    t_data,
    r_ack
);

input tclk;
input resetb_tclk;

input r_ack;
input data_avail;
input [31:0] transmit_data;
output t_rdy;
output [31:0] t_data;

localparam IDLE_T = 2'd0, ASSERT_TRDY = 2'd1, DEASSERT_TRDY = 2'd2;

reg [1:0] t_hndshk_state, t_hndshk_state_nxt;
reg t_rdy, t_rdy_nxt;
reg [31:0] t_data, t_data_nxt;
reg r_ack_d1, r_ack_tclk;

always @(*)
begin
    t_hndshk_state_nxt = t_hndshk_state;
    t_rdy_nxt = 1'b0;
    t_data_nxt = t_data;

    case(t_hndshk_state)
        IDLE_T:
            begin
                if(r_ack == 1)
                    t_rdy_nxt = 1'b1;
                else
                    t_rdy_nxt = 1'b0;
            end
        ASSERT_TRDY:
            begin
                t_rdy_nxt = 1'b1;
            end
        DEASSERT_TRDY:
            begin
                t_rdy_nxt = 1'b0;
            end
    endcase
end

assign t_rdy = t_rdy_nxt;
assign t_data = t_data_nxt;
assign r_ack_tclk = r_ack_d1;
```

(下页继续)

(续上页)

```

if(data_avail) // if the data is available in transmit side
begin
    t_hndshk_state_nxt = ASSERT_TRDY;
    t_rdy_nxt = 1'b1;
    t_data_nxt = transmit_data;// data to be transferred
end
end
ASSERT_TRDY:
begin
    if(r_ack_tclk)
        begin
            t_rdy_nxt = 1'b0;
            t_hndshk_state_nxt = DEASSERT_TRDY;
            t_data_nxt = 'd0;
        end
    else
        begin
            t_rdy_nxt = 1'b1;// keep driving until r_ack_tclk=1
            t_data_nxt = t_data;// keep supplying data
        end
    end
DEASSERT_TRDY:
begin
    if(!r_ack_tclk)
        begin
            if(data_avail)
                begin
                    t_hndshk_state_nxt = ASSERT_TRDY;
                    t_rdy_nxt = 1'b1;
                    t_data_nxt = transmit_data;
                end
            else
                t_hndshk_state_nxt = IDLE_T;
        end
    end
default:
begin
end
endcase
end

always @ (posedge tclk or negedge resetb_tclk)
begin
if(!resetb_tclk)
begin
    t_hndshk_state <= IDLE_T;
    t_rdy <= 1'b0;
    t_data <= 'd0;
    r_ack_d1 <= 1'b0;
    r_ack_tclk <= 1'b0;
end
else
begin
    t_hndshk_state <= t_hndshk_state_nxt;
    t_rdy <= t_rdy_nxt;
    t_data <= t_data_nxt;
    r_ack_d1 <= r_ack;
    r_ack_tclk <= r_ack_d1;
end
end
end

```

(下页继续)

(续上页)

endmodule

```
// Verilog RTL for Full Handshake - Receive
module handshake_rclk
(
    rclk,
    resetb_rclk,
    t_rdy,
    t_data,
    r_ack
);

input rclk;
input resetb_rclk;
input t_rdy;
input [31:0] t_data;
output r_ack;

localparam IDLE_R = 1'b0, ASSERT_ACK = 1'b1;
reg r_hndshk_state, r_hndshk_state_nxt;
reg r_ack, r_ack_nxt;
reg [31:0] t_data_rclk, t_data_rclk_nxt;
reg t_rdy_d1, t_rdy_rclk;

always @(*)
    begin
        r_hndshk_state_nxt = r_hndshk_state;
        r_ack_nxt = 1'b0;
        t_data_rclk_nxt = t_data_rclk;

        case(r_hndshk_state)
            IDLE_R:
                begin
                    if(t_rdy_rclk)
                        begin
                            r_hndshk_state_nxt = ASSERT_ACK;
                            r_ack_nxt = 1'b1;
                            t_data_rclk_nxt = t_data;
                        end
                end
            ASSERT_ACK:
                begin
                    if(!t_rdy_rclk)
                        begin
                            r_ack_nxt = 1'b0;
                            r_hndshk_state_nxt = IDLE_R;
                        end
                    else
                        r_ack_nxt = 1'b1;
                end
            default:
                begin
                end
        endcase
    end

    always @(posedge rclk or negedge resetb_rclk)
    begin
        if(!resetb_rclk)
            begin
                r_hndshk_state <= IDLE_R;
            end
    end

```

(下页继续)

(续上页)

```

    r_ack <= 1'b0;
    t_data_rclk <= 'd0;
    t_rdy_d1 <= 1'b0;
    t_rdy_rclk <= 1'b0;
  end
else
begin
  r_hndshk_state <= r_hndshk_state_nxt;
  r_ack <= r_ack_nxt;
  t_data_rclk <= t_data_rclk_nxt;
  t_rdy_d1 <= t_rdy;
  t_rdy_rclk <= t_rdy_d1;
end
end
endmodule

```

复位方法

复位被用来将数字电路中的触发器强制设置到一个确定的初始值上，从而使状态机和其它的控制电路可以从一个已知的初始状态开始工作。带有复位引脚的触发器所占用的芯片面积比没有复位引脚的触发器略微大一些。在某些情况下，处于数据处理路径上的触发器的初始值无关紧要，此时可以使用不带复位引脚的触发器，以将低芯片的总面积。电路中有两种典型的复位方法，非同步复位和同步复位。

非同步复位（异步复位）

采用异步复位时，触发器中存在一个复位端。一般情况下，复位时低电平有效的，通常用 `reset#` 来表示。当 `reset#` 为低电平时，触发器输出立刻变成 0 或 1。`reset#` 可以在任何时刻被置为低电平，它与时钟边沿之间可以没有任何关系，因此这种复位方式被称为异步复位。

需要注意的是，当 `reset#` 被置为高电平时，它必须与时钟的上升沿同步。`reset#` 从 0 到 1 翻转时，不能离时钟的上升沿太近，不然会产生与违反建立时间/保持时间类似的输出不稳定问题，此时它被称为复位恢复错误。有时候一个复位信号会经过不同的时钟域，此时不能直接使用该复位信号。此时它的上升沿必须与新时钟域进行同步以免产生复位恢复错误。针对跨时钟域的同步，可以使用专门设计的复位同步电路。

复位同步电路

这里给出一种复位同步电路的代码：

```

module reset_synchronizer
(
  clk,
  rstb_in,
  rstb_sync
);

  input clk;
  input rstb_in;
  output rstb_sync;

  reg      rstb_in_pre, rstb_sync;

  always @ (posedge clk or negedge rstb_in)
  begin
    if (!rstb_in)
      begin
        rstb_in_pre <= 1'b0;
        rstb_sync <= 1'b0;
      end
    else
      begin

```

(下页继续)

(续上页)

```

    rstb_in_pre <= 1'b1;
    rstb_sync <= rstb_in_pre;
end
end

endmodule

```

同步复位

采用同步复位时，没有专用的复位端。复位信号时决定触发器输入信号值的变量之一。由于复位信号被当成输入信号的一部分，因此它必须满足和一般数据输入一样的建立时间和保持时间要求。

异步复位和同步复位的选择

异步复位和同步复位都可以用于数字系统设计之中。异步复位信号必须直接来自触发器，并且复位信号中不能有毛刺。对于同步复位来说，复位信号可以容忍毛刺，前提是毛刺不能不在建立时间和保持时间窗口内。由于异步复位不依赖于时钟，因此在时钟没有运行根本没有时钟的情况下可以使用。对于同步复位方法，必须在有时钟的情况下才能进行复位。另外，同步复位信号会出现在数据延迟路径中，可能会带来路径延迟的增加，对于告诉设计来说需要仔细考虑，可能会产生不良影响。

9.2 verilog 临时存放

9.2.1 Verilog \$random 用法随机数

\$random 函数调用时返回一个 32 位的随机数，它是一个带符号的整形数。

```

reg[23:0] rand;
//产生一个在 -59 — 59 范围的随机数
rand=$random%60;

// 产生 0~59 之间的随机数的例子
rand={$random} %60; //通过位拼接操作 {}

// 产生一个在 min, max 之间随机数
rand = min+{$random}%(max-min+1);

```

9.2.2 verilog 数组定义及其初始化

这里的内存模型指的是内存的行为模型。Verilog 中提供了两维数组来帮助我们建立内存的行为模型。具体来说，就是可以将内存宣称为一个 reg 类型的数组，这个数组中的任何一个单元都可以通过一个下标去访问。这样的数组的定义方式如下：

```

reg [wordsize : 0] array_name [0 : arraysize];

// 例如：
reg [7:0] my_memory [0:255];
// 其中 [7:0] 是内存的宽度，而 [0:255] 则是内存的深度（也就是有多少存储单元），其中宽度为 8 位，深度为 256。地址 0 对应着数组中的 0 存储单元。

// 如果要存储一个值到某个单元中去，可以这样做：
my_memory [address] = data_in;

// 而如果要从某个单元读出值，可以这么做：
data_out = my_memory [address];

// 但要是只需要读一位或者多个位，就要麻烦一点，因为 Verilog 不允许读/写一个位。这时，就需要使用一个变量转换一下：

```

(下页继续)

(续上页)

```
// 例如:
data_out = my_memory[address];
data_out_it_0 = data_out[0];
// 这里首先从一个单元里面读出数据, 然后再取出读出的数据的某一位的值。
```

初始化内存

初始化内存有多种方式, 这里介绍的是使用 \$readmemb 和 \$readmemh 系统任务来将保存在文件中的数据填充到内存单元中去。\$readmemb 和 \$readmemh 是类似的, 只不过 \$readmemb 用于内存的二进制表示, 而 \$readmemh 则用于内存内容的 16 进制表示。这里以 \$readmemh 系统任务来介绍。

语法

```
$readmemh("file_name", mem_array, start_addr, stop_addr);

// 注意的是:
// file_name 是包含数据的文本文件名, mem_array 是要初始化的内存单元数组名, start_addr 和
// stop_addr 是可选的, 指示要初始化单元的起始地址和结束地址。
```

下面是一个简单的例子:

```
module memory ();
reg [7:0] my_memory [0:255];

initial begin
$readmemh("memory.list", my_memory);
end
endmodule

// 这里使用内存文件 memory.list 来初始化 my_memory 数组。
```

9.2.3 组合逻辑 for 循环和 generate 生成块 for 循环

例 1: 给一个 100 位的输入向量, 颠倒它的位顺序输出

只需要将 in[0] 赋值给 out[99]、in[1] 赋值给 out[98]……也可以直接用 for 循环, 其规范格式如下:

```
for (循环变量赋初值; 循环执行条件; 循环变量增值) 循环体语句块;
```

通过 for 循环赋值很方便:

```
module top_module (
    input [99:0] in,
    output reg [99:0] out
);

    always @(*) begin
        for (int i=0;i<$bits(out);i++)           // $bits() is a system function that
// returns the width of a signal.
            out[i] = in[$bits(out)-i-1];          // $bits(out) is 100 because out is
// 100 bits wide.
    end

endmodule
```

例 2: 建立一个“人口计数器”来统计一个 256 位输入向量中 1 的数量

统计 1 的个数可以直接将每一 bit 位加起来, 得到的数值即为 1 的个数。缩减运算符只有与或非, 由于加法不是一个简单地逻辑门就可以计算, 所以只能一位一位的提取出来相加, 因此用 for 语句

```

module top_module (
    input [254:0] in,
    output reg [7:0] out
);

    always @(*) begin          // Combinational always block
        out = 0;           // if don't assign initial value zero, simulate
        ← errors will emerge
        for (int i=0;i<255;i++)
            out = out + in[i];
    end

endmodule

```

例 3：通过实例化 100 个一位全加器制造一个 100 位的脉冲进位加法器

这个加法器将两个 100 位的输入信号和一个进位进位加起来产生一个 100 位的输出信号和进位信号。我们依旧用 for 循环语句，只是这次循环内容是另一个模块，在这里就要引入一个新的概念 generate 生成块。

Verilog-2001 添加了 generate 循环，允许产生 module 和 primitive 的多个实例化，同时也可以产生多个 variable, net, task, function, continuous assignment, initial 和 always。在 generate 语句中可以引入 if-else 和 case 语句，根据条件不同产生不同的实例化。

用法：

1. generate 语法有 generate for, genreate if 和 generate case 三种
2. generate for 语句必须有 genvar 关键字定义 for 的变量
3. for 的内容必须加 begin 和 end
4. 必须给 for 语段起个名字，这个名字会作为 generate 循环的实例名称。

标准格式：

```

generate
genvar i; //定义变量
for(循环变量赋初值; 循环执行条件; 循环变量增值) begin: gfor //生成后的例化名, gfor[1].ui(实例化)、gfor[2].ui(实例化).....
//需要循环的实例模块
end
endgenerate

```

因为第一个实例的输入是 cin，其他的都是上一级的 cout，因此把第一个单独例化。

```

module top_module(
    input [99:0] a, b,
    input cin,
    output [99:0] cout,
    output [99:0] sum );
    fadd u0(.a(a[0]),
        .b(b[0]),
        .cin(cin),
        .cout(cout[0]),
        .sum(sum[0])
    );
    generate
        genvar i;
        for(i=1;i<100;i++) begin: gfor
            fadd ui(.a(a[i]),           //this i of ui won't be replaced
                .b(b[i]),
                .cin(cout[i-1]),
                .cout(cout[i]),
                .sum(sum[i])
            );
    end

```

(下页继续)

(续上页)

```

    endgenerate
endmodule
module fadd(
    input a, b, cin,
    output cout, sum );
    assign {cout,sum} = a+b+cin;
endmodule

```

https://blog.csdn.net/weixin_38197667/article/details/90401400

9.3 VHDL temp

9.3.1 缩位与/缩位或运算

缩位运算符，即“reduction operator”。对于VHDL来说，很少有人知道其缩位运算符是什么。首先缩位运算的意思是把一个vector合并成一位，例如缩位与运算符：对于一个std_logic_vector名为example的变量，完成example[0] and example[1] and … and example[22]这样的运算的运算符。

- 对于VHDL-2008，直接用and就可以完成：and(example);
- 用组合逻辑自己写一个函数，按位或/与即可。
- or_reduce 和 and_reduce 也可以完成上面的内容。要主要包含头文件 std_logic_misc。

```

function and_reduct(slv : in std_logic_vector) return std_logic is
    variable res_v : std_logic := '1'; -- Null slv vector will also return '1'
begin
    for i in slv'range loop
        res_v := res_v and slv(i);
    end loop;
    return res_v;
end function;
-- You can then use the function both inside and outside functions with:

signal arg : std_logic_vector(7 downto 0);
signal res : std_logic;
-- ...
res <= and_reduct(arg);

```

9.3.2 循环

```

for i in 0 to N_HISTOGRAM_REGS-1 loop
    tmp_histogram(i) <= std_logic_vector(to_unsigned(0,N_HISTOGRAM_BIT));
end loop;

```

```

detrigA: for i in 0 to 31 generate
    process(clk100)
    begin
        if(clk100'event and clk100 = '1')then
            if(A0(i)='1' and A1(i)='0')then
                triggerA(i) <= '1';
            else
                triggerA(i) <= '0';
            end if;
        end if;
    end process;
end generate;

```

9.3.3 信号处理

HLS 生成模块

```
entity BaselineRestorer is
port (
    CLK : IN STD_LOGIC;
    TRIGGER: IN STD_LOGIC;--原始波形触发信号
    DATA_IN: IN STD_LOGIC_VECTOR(15 DOWNTO 0);--输入原始波形
    M_LENGTH: IN INTEGER;-- $2^n$  参与基线计算的时间窗长度
    BL_HOLD: IN INTEGER;--触发之后禁止基线计算的点的个数
    FLUSH: IN STD_LOGIC;
    BASELINE: OUT STD_LOGIC_VECTOR(15 DOWNTO 0);--输出基线
    BASELINE_VALID: OUT STD_LOGIC;--基线有效标记
    HOLD_TIME: OUT STD_LOGIC_VECTOR(31 DOWNTO 0);--不计算基线的时间
    RUNNING_NOT_HOLD: OUT STD_LOGIC--正在计算基线的标记
);
end;
```

```
entity moving_average is
port (
    ap_clk : IN STD_LOGIC;
    ap_rst : IN STD_LOGIC;
    adc_data_V : IN STD_LOGIC_VECTOR (15 downto 0);
    adc_data_V_ap_vld : IN STD_LOGIC;
    hold : IN STD_LOGIC;
    length_V : IN STD_LOGIC_VECTOR (15 downto 0);
    dataout_V : OUT STD_LOGIC_VECTOR (15 downto 0);
    dataout_V_ap_vld : OUT STD_LOGIC );
end;
```

```
entity charge_integration is
port (
    ap_clk : IN STD_LOGIC;
    ap_rst : IN STD_LOGIC;
    in1_V : IN STD_LOGIC_VECTOR (15 downto 0);--输入波形
    in1_V_ap_vld : IN STD_LOGIC;-- '1'
    base_line_V : IN STD_LOGIC_VECTOR (15 downto 0);--输入基线
    trigger_signal : IN STD_LOGIC;--输入触发
    p_int_length_V : IN STD_LOGIC_VECTOR (15 downto 0);--积分门宽
    p_pre_length_V : IN STD_LOGIC_VECTOR (15 downto 0);--积分起始点, 触发前
    p_gain_V : IN STD_LOGIC_VECTOR (15 downto 0);--增益
    p_offset_V : IN STD_LOGIC_VECTOR (15 downto 0);--偏置
    p_pileup_inib_V : IN STD_LOGIC_VECTOR (15 downto 0);--堆积拒绝的事件间隔
    enable : IN STD_LOGIC;-- '1'
    energy_out_V : OUT STD_LOGIC_VECTOR (15 downto 0);--输出积分结果
    energy_trigger : OUT STD_LOGIC;-- 标记输出积分有效
    energy_trigger_ap_vld : OUT STD_LOGIC;-- open
    p_integrate : OUT STD_LOGIC;-- 积分门
    p_pileup : OUT STD_LOGIC;-- 堆积标记
    p_busy : OUT STD_LOGIC );
end;
```

```
entity trapezio is
port (
    ap_clk : in STD_LOGIC;
    ap_rst : in STD_LOGIC;
    adc_data_V : in STD_LOGIC_VECTOR ( 15 downto 0 );
    adc_data_V_ap_vld : in STD_LOGIC;
    baseline_V : in STD_LOGIC_VECTOR ( 15 downto 0 );
```

(下页继续)

(续上页)

```

k_V : in STD_LOGIC_VECTOR ( 15 downto 0 );
m_V : in STD_LOGIC_VECTOR ( 15 downto 0 );
M_V_r : in STD_LOGIC_VECTOR ( 31 downto 0 );
G_V : in STD_LOGIC_VECTOR ( 31 downto 0 );
dataout_V : out STD_LOGIC_VECTOR ( 15 downto 0 );
dataout_V_ap_vld : out STD_LOGIC
);
attribute NotValidForBitStream : boolean;
attribute NotValidForBitStream of trapezio : entity is true;
end trapezio;

```

```

entity MCAHP_512 is
port (
    ap_clk : IN STD_LOGIC;
    ap_rst : IN STD_LOGIC;
    adc_data_V : IN STD_LOGIC_VECTOR (15 downto 0);
    positive_r : IN STD_LOGIC;
    digital_offset_V : IN STD_LOGIC_VECTOR (15 downto 0);
    threshold_V : IN STD_LOGIC_VECTOR (31 downto 0);
    trig_k_V : IN STD_LOGIC_VECTOR (15 downto 0);--快速滤波梯形上升时间
    trig_m_V : IN STD_LOGIC_VECTOR (15 downto 0);--快速滤波梯形上升 + 平台时间
    e_k_V : IN STD_LOGIC_VECTOR (15 downto 0);--梯形上升时间
    e_m_V : IN STD_LOGIC_VECTOR (15 downto 0);--梯形上升 + 平台时间
    e_MDec_V : IN STD_LOGIC_VECTOR (23 downto 0);-- int(256/(exp(clock_sampling_
→time/tau)-1))
    e_G_V : IN STD_LOGIC_VECTOR (23 downto 0);--int(gain*0x10000) gain 为浮点数
    e_MDec2_V : IN STD_LOGIC_VECTOR (15 downto 0);
    e_G2_V : IN STD_LOGIC_VECTOR (15 downto 0);
    e_sample_delay_V : IN STD_LOGIC_VECTOR (15 downto 0);
    baseline_len_V : IN STD_LOGIC_VECTOR (3 downto 0);--2^n      5->32   6->64   7->
→128 8->256 9->512 10->1024 11->2048
    baseline_inib_V : IN STD_LOGIC_VECTOR (15 downto 0);--触发之后抑制基线计算的时间, 需
要大于两倍能量梯形的上升和平台之和
    run_cfg : IN STD_LOGIC;-- '1'
    deconv2_sig_V : OUT STD_LOGIC_VECTOR (31 downto 0);--open
    deconv2_sig_V_ap_vld : OUT STD_LOGIC;--open
    trigger_delta_monitor_V : OUT STD_LOGIC_VECTOR (31 downto 0);--open
    trigger_delta_monitor_V_ap_vld : OUT STD_LOGIC;--open
    trigger_trap_monitor_V : OUT STD_LOGIC_VECTOR (31 downto 0);--触发滤波
    trigger_trap_monitor_V_ap_vld : OUT STD_LOGIC;
    trap_V : OUT STD_LOGIC_VECTOR (31 downto 0);--梯形滤波
    trap_V_ap_vld : OUT STD_LOGIC;
    trap_minus_baseline_V : OUT STD_LOGIC_VECTOR (31 downto 0);--梯形滤波减基线滤波
    trap_minus_baseline_V_ap_vld : OUT STD_LOGIC;
    baseline_out_V : OUT STD_LOGIC_VECTOR (31 downto 0);--基线滤波
    baseline_out_V_ap_vld : OUT STD_LOGIC;
    energy_V : OUT STD_LOGIC_VECTOR (31 downto 0);--能量滤波
    energy_V_ap_vld : OUT STD_LOGIC;
    energy_strobe : OUT STD_LOGIC;--能量采集标记
    trigger_sig : OUT STD_LOGIC;--触发信号
    baseline_hold : OUT STD_LOGIC;
    GIN_SELECT_V : IN STD_LOGIC_VECTOR (3 downto 0);
    gin : IN STD_LOGIC );
end;

```

```

entity gated_integrator is
port (
    ap_clk : IN STD_LOGIC;--时钟
    ap_rst : IN STD_LOGIC;--重置, 高电平有效
    data_in_V : IN STD_LOGIC_VECTOR (31 downto 0);--输入数据

```

(下页继续)

(续上页)

```

data_in_V_ap_vld : IN STD_LOGIC;--输入数据有效标记
gate_len_V : IN STD_LOGIC_VECTOR (15 downto 0);--积分门宽, 最大到 1024
gain_V : IN STD_LOGIC_VECTOR (31 downto 0);--增益, 65536 表示增益为 1。算法为 gain/
→65536
clear : IN STD_LOGIC;--从高电平到低电平时, 重新初始化。初始化时间与积分门宽成正比。这期间_
→data_out_V_ap_vld/ready 为低电平
data_out_V : OUT STD_LOGIC_VECTOR (31 downto 0);--数据输出, 当前时钟之前的积分门结果输出
延迟 4 个时钟
data_out_V_ap_vld : OUT STD_LOGIC;--输出数据有效标记
ready : OUT STD_LOGIC --高电平表示输出有效, 低电平表示在初始化
);
end;

```

```

-- 数值导数/微分
-- OUT[n] = DATA_IN[n] - DATA_IN[n-WINDOW])
entity differenziator is
port (
    ap_clk : IN STD_LOGIC;--时钟
    ap_rst : IN STD_LOGIC;--重置, 高电平有效
    data_in_V : IN STD_LOGIC_VECTOR (31 downto 0);--输入数据
    data_in_V_ap_vld : IN STD_LOGIC;--输入数据有效标记
    diff_len_V : IN STD_LOGIC_VECTOR (15 downto 0);--前后做差两个点的间隔, 最大为 999
    clear : IN STD_LOGIC;----从高电平到低电平时, 重新初始化。初始化时间最小为 WINDOW。这期间_
→data_out_V_ap_vld/ready 为低电平
    data_out_V : OUT STD_LOGIC_VECTOR (31 downto 0);--当前时钟输入数据与之前数据的差输出
延迟两个时钟
    data_out_V_ap_vld : OUT STD_LOGIC;--数据输出, 当前时钟输入数据的输出延迟 2 个时钟
    ready : OUT STD_LOGIC;--输出数据有效标记
    ready_ap_vld : OUT STD_LOGIC --输出常为 1, open
);
end;

```

```

entity PSD_INTDUAL is
port (
    ap_clk : IN STD_LOGIC;
    ap_rst : IN STD_LOGIC;
    data_in_V : IN STD_LOGIC_VECTOR (15 downto 0);
    data_in_V_ap_vld : IN STD_LOGIC;
    trigger : IN STD_LOGIC;
    int_short_V : IN STD_LOGIC_VECTOR (15 downto 0);
    int_long_V : IN STD_LOGIC_VECTOR (15 downto 0);
    pileup_inib_V : IN STD_LOGIC_VECTOR (15 downto 0);
    psd_out_V : OUT STD_LOGIC_VECTOR (31 downto 0);
    psd_out_V_ap_vld : OUT STD_LOGIC;
    q_long_out_V : OUT STD_LOGIC_VECTOR (31 downto 0);
    q_long_out_V_ap_vld : OUT STD_LOGIC;
    q_short_out_V : OUT STD_LOGIC_VECTOR (31 downto 0);
    q_short_out_V_ap_vld : OUT STD_LOGIC );
end;

```

手写模块

```

-- 1,2,4,8 个点平均
entity NoiseFilter is
Generic (data_bit : integer := 16);
port (
    CLK : IN STD_LOGIC;
    MODE : IN STD_LOGIC_VECTOR (1 downto 0);-- "00"=>1 "01"=>2 "10"=>4 "11"=>8
);

```

(下页继续)

(续上页)

```

DATA_IN : IN STD_LOGIC_VECTOR (data_bit-1 DOWNTO 0);
DATA_OUT : OUT STD_LOGIC_VECTOR (data_bit-1 DOWNTO 0)
);
end;

```

```

entity TriggerDerivative is
Generic (
  data_bit : integer := 16;--输入波形位数
  noise_filter : integer := 2;--计算两个间隔为 n 的点的差值为滤波结果
  data_delay : integer := 3--输入波形的延迟输出
);
port (
  CLK : IN STD_LOGIC;
  POLARITY: IN STD_LOGIC;-- 1=>pos 0=>neg
  DATA_IN: IN STD_LOGIC_VECTOR(data_bit-1 DOWNTO 0);
  THRESHOLD: IN STD_LOGIC_VECTOR(data_bit-1 DOWNTO 0);
  DELAYED_DATA: OUT STD_LOGIC_VECTOR(data_bit-1 DOWNTO 0);
  TRIGGER: OUT STD_LOGIC
);
end;

```

```

entity TriggerLeading is
Generic (
  data_bit : integer := 16;--输入波形位数
  data_delay : integer := 3--输入波形的延迟输出
);
port (
  CLK : IN STD_LOGIC;
  POLARITY: IN STD_LOGIC;--1=>pos 0=>neg
  DATA_IN: IN STD_LOGIC_VECTOR(data_bit-1 DOWNTO 0);
  THRESHOLD: IN STD_LOGIC_VECTOR(data_bit-1 DOWNTO 0);
  DELAYED_DATA: OUT STD_LOGIC_VECTOR(data_bit-1 DOWNTO 0);
  TRIGGER: OUT STD_LOGIC
);
end;

```

```

entity LOGIC_ANALYZER is
Generic (
  bitsize : integer := 32;--通道数, 最大 256
  fifolength : integer := 16);
port (
  RESET : IN STD_LOGIC_VECTOR (0 DOWNTO 0);--重置
  CLK_READ : IN STD_LOGIC_VECTOR (0 DOWNTO 0);--时钟
  CLK_WRITE : IN STD_LOGIC_VECTOR (0 DOWNTO 0);--时钟
  DATAIN : IN STD_LOGIC_VECTOR (bitsize-1 DOWNTO 0);--数据输入
  TRIGGER : IN STD_LOGIC_VECTOR (0 DOWNTO 0);--外部输入触发
  FULL: OUT STD_LOGIC_VECTOR (0 downto 0);
  READ_DATA : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
  READ_DATALID : OUT STD_LOGIC_VECTOR (0 DOWNTO 0);
  READ_NEXT : IN STD_LOGIC_VECTOR (0 DOWNTO 0);
  STATUS : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);--[0]1 表示数据可以读取 [1]1 表示已经初始化, 0 表示重置或者等待触发中 [2]1 表示已经初始化, 0 表示重置或者数据已经写入 FIFO 完成
  CONFIG : IN STD_LOGIC_VECTOR (31 DOWNTO 0);--[0]1 表示 enable [1]1 表示 reset--[2]1 表示外部输入 TRIGGER 触发 [3]1 表示上升沿或者下降沿触发 [4]1 表示软件触发
  CONFIG0 : IN STD_LOGIC_VECTOR (31 DOWNTO 0);--0-31 通道是否开启上升沿触发
  CONFIG1 : IN STD_LOGIC_VECTOR (31 DOWNTO 0);--32-63 通道是否开启上升沿触发
  CONFIG2 : IN STD_LOGIC_VECTOR (31 DOWNTO 0);--后面以此类推
  CONFIG3 : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
  CONFIG4 : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
  CONFIG5 : IN STD_LOGIC_VECTOR (31 DOWNTO 0);

```

(下页继续)

(续上页)

```

CONFIG6 : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
CONFIG7 : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
CONFIG8 : IN STD_LOGIC_VECTOR (31 DOWNTO 0);--0-31 通道是否开启下降沿触发
CONFIG9 : IN STD_LOGIC_VECTOR (31 DOWNTO 0);--32-63 通道是否开启下降沿触发
CONFIGA : IN STD_LOGIC_VECTOR (31 DOWNTO 0);--后面以此类推
CONFIGB : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
CONFIGC : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
CONFIGD : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
CONFIGE : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
CONFIGF : IN STD_LOGIC_VECTOR (31 DOWNTO 0)
);
end;

```

9.4 LPM (library of parameterized mudules)

- https://blog.csdn.net/next_fse/article/details/73864596
- http://blog.sina.com.cn/s/blog_6e350d8801011hfx.html

文件位置，安装目录下 quartus/eda/fv_lib/verilog

ALTERA 在 LPM (library of parameterized mudules) 库中提供了参数可配置的单时钟 FIFO (SCFIFO) 和 双时钟 FIFO (DCFIFO)。FIFO 主要应用在需要数据缓冲且数据符合先进先出规律的同步或异步场合。LPM 中的 FIFO 包含以下几种：

- SCFIFO: 单时钟 FIFO；
- DCFIFO: 双时钟 FIFO，数据输入和输出的宽度相同；
- DCFIFO_MIXED_WIDTHS: 双时钟 FIFO，输入输出数据位宽可以不同。

<https://www.cnblogs.com/rouwawa/p/7066635.html>

```

-- 以下模块需要研究明白参数
in_delay : altshift_taps
generic map (
  intended_device_family => "unused",
  number_of_taps => 64,
  power_up_state => "CLEARED",
  tap_distance => 63,
  width => bitsize,
  lpm_hint => "UNUSED",
  lpm_type => "altshift_taps"
)
port map(
  aclr => RESET(0),
  clken => '1',
  clock => CLK_WRITE(0),
  shiftin => DATAIN,
  shiftout => DATA_DELAY,
  taps => open
);

dcfifo_mixed_widths_component : dcfifo_mixed_widths
GENERIC MAP (
  intended_device_family => "Cyclone V",
  lpm_numwords => fifolength,
  lpm_showahead => "ON",
  lpm_type => "dcfifo_mixed_widths",
  lpm_width => bitsize,
  lpm_widthu => wBits,
)

```

(下页继续)

(续上页)

```

lpm_widthu_r => rBits,
lpm_width_r => 32,
overflow_checking => "ON",
rdsync_delaypipe => 4,
read_aclr_synch => "OFF",
underflow_checking => "ON",
use_eab => "ON",
write_aclr_synch => "OFF",
wrsync_delaypipe => 4
)
PORT MAP (
    aclr => FifoClear,
    data => DATA_DELAY,
    rdclk => CLK_READ(0),
    rdreq => iREAD_NEXT,
    wrclk => CLK_WRITE(0),
    wrreq => EnableWrite,
    q => READ_DATA,
    rdempty => iEMPTY,
    wrfull => iFULL
);

```

```

GEN_INPUT_FIFO:
for I in 0 to InputCount-1 generate
    xpm_fifo_async_inst : xpm_fifo_async
        generic map (
            FIFO_MEMORY_TYPE => "auto", --string; "auto", "block", or "distributed"
            --;
            ECC_MODE => "no_ecc", --string; "no_ecc" or "en_ecc";
            RELATED_CLOCKS => 0, --positive integer; 0 or 1
            FIFO_WRITE_DEPTH => InputFifoSize, --positive integer
            WRITE_DATA_WIDTH => InputWordSize, --positive integer
            WR_DATA_COUNT_WIDTH => wBits, --positive integer
            PROG_FULL_THRESH => InputFifoSize-5, --positive integer
            FULL_RESET_VALUE => 0, --positive integer; 0 or 1;
            READ_MODE => "fwft", --string; "std" or "fwft";
            FIFO_READ_LATENCY => 1, --positive integer;
            READ_DATA_WIDTH => InputWordSize, --positive integer
            RD_DATA_COUNT_WIDTH => rBits, --positive integer
            PROG_EMPTY_THRESH => 5, --positive integer
            DOUT_RESET_VALUE => "0", --string
            CDC_SYNC_STAGES => 2, --positive integer
            WAKEUP_TIME => 0 --positive integer; 0 or 2;
        )
        port map (
            sleep => '0',
            rst => reset,
            wr_clk => clk,
            wr_en => DV_IN(I),
            din => DATA_IN(((I+1)*InputWordSize)-1 downto (I*InputWordSize)),
            full => BUSY_IN(I),
            overflow => open,
            wr_rst_busy => open,
            rd_clk => clk,
            rd_en => RDEN_FIFO(I),
            dout => DATA_IN_FIFO(((I+1)*InputWordSize)-1 downto
            --(I*InputWordSize)),
            empty => EMPTY_FIFO(I),
            underflow => open,
            rd_rst_busy => open,
            prog_full => open,

```

(下页继续)

(续上页)

```

        wr_data_count => open,
        prog_empty => open,
        rd_data_count => open,
        injectsbiterr => '0',
        injectdbiterr => '0',
        sbiterr => open,
        dbiterr => open
    );
RDEN_FIFO(I) <= (not EMPTY_FIFO(I)) and (not BUSY_IN_FIFO(I));
DV_IN_FIFO(I) <= RDEN_FIFO(I);
end generate GEN_INPUT_FIFO;

```

```

COMPONENT dcfifo_mixed_widths
  GENERIC (
    intended_device_family : STRING;
    lpm_numwords          : NATURAL;
    lpm_showahead         : STRING;
    lpm_type               : STRING;
    lpm_width              : NATURAL;
    lpm_widthhu            : NATURAL;
    lpm_widthhu_r          : NATURAL;
    lpm_width_r             : NATURAL;
    overflow_checking      : STRING;
    rdsync_delaypipe       : NATURAL;
    read_aclr_synch        : STRING;
    underflow_checking     : STRING;
    use_eab                : STRING;
    write_aclr_synch       : STRING;
    wrsync_delaypipe       : NATURAL
  );
  PORT (
    aclr      : IN STD_LOGIC ;
    data      : IN STD_LOGIC_VECTOR (bitsize-1 DOWNTO 0);
    rdclk     : IN STD_LOGIC ;
    rdreq     : IN STD_LOGIC ;
    wrclk     : IN STD_LOGIC ;
    wrreq     : IN STD_LOGIC ;
    q         : OUT STD_LOGIC_VECTOR (32-1 DOWNTO 0);
    rdempty   : OUT STD_LOGIC ;
    wrfull   : OUT STD_LOGIC
  );
END COMPONENT;

```

```

xpm_fifo_async_inst : xpm_fifo_async
generic map (
  FIFO_MEMORY_TYPE => "auto", --string; "auto", "block", or "distributed";
  ECC_MODE => "no_ecc", --string; "no_ecc" or "en_ecc";
  RELATED_CLOCKS => 0, --positive integer; 0 or 1
  FIFO_WRITE_DEPTH => fifolength, --positive integer
  WRITE_DATA_WIDTH => bitsize, --positive integer
  WR_DATA_COUNT_WIDTH => wBits, --positive integer
  PROG_FULL_THRESH => 5, --positive integer
  FULL_RESET_VALUE => 0, --positive integer; 0 or 1;
  READ_MODE => "std", --string; "std" or "fwft";
  FIFO_READ_LATENCY => 1, --positive integer;
  READ_DATA_WIDTH => 32, --positive integer
  RD_DATA_COUNT_WIDTH => rBits, --positive integer
  PROG_EMPTY_THRESH => 3, --positive integer
  DOUT_RESET_VALUE => "0", --string
  CDC_SYNC_STAGES => 2, --positive integer

```

(下页继续)

(续上页)

```

WAKEUP_TIME => 0 --positive integer; 0 or 2;
)
port map (
    sleep => '0',
    rst => RESET(0),
    wr_clk => CLK_WRITE(0),
    wr_en => iWRITE,
    din => DATAIN,
    full => iFULL,
    overflow => open,
    wr_rst_busy => open,
    rd_clk => CLK_READ(0),
    rd_en => iREAD_NEXT,
    dout => READ_DATA,
    empty => iEMPTY,
    underflow => open,
    rd_rst_busy => open,
    prog_full => open,
    wr_data_count => open,
    prog_empty => open,
    rd_data_count => open,
    injectsbiterr => '0',
    injectdbiterr => '0',
    sbiterr => open,
    dbiterr => open
);

```

```

xpm_memory_sdpram_inst : xpm_memory_sdpram
generic map (
    -- Common module generics
    MEMORY_SIZE => maxDelay, --positive integer
    MEMORY_PRIMITIVE => "auto", --string; "auto", "distributed", "block" or
    -->"ultra" ;
    CLOCKING_MODE => "common_clock",--string; "common_clock", "independent_clock"
    MEMORY_INIT_FILE => "none", --string; "none" or "<filename>.mem"
    MEMORY_INIT_PARAM => "", --string;
    USE_MEM_INIT => 1, --integer; 0,1
    WAKEUP_TIME => "disable_sleep",--string; "disable_sleep" or "use_sleep_pin"
    MESSAGE_CONTROL => 0, --integer; 0,1
    -- Port A module generics
    WRITE_DATA_WIDTH_A => busWidth, --positive integer
    BYTE_WRITE_WIDTH_A => busWidth, --integer; 8, 9, or WRITE_DATA_WIDTH_A value
    ADDR_WIDTH_A => maxDelayBits, --positive integer
    -- Port B module generics
    READ_DATA_WIDTH_B => busWidth, --positive integer
    ADDR_WIDTH_B => maxDelayBits, --positive integer
    READ_RESET_VALUE_B => "0", --string
    READ_LATENCY_B => 2, --non-negative integer
    WRITE_MODE_B => "no_change" --string; "write_first", "read_first", "no_change
    --"
)
port map (
    -- Common module ports
    sleep => '0',
    -- Port A module ports
    clka => CLK(0),
    ena => '1',
    wea => "1",
    addra => WP,
    dina => iIN,
    injectsbiterra => '0', --do not change

```

(下页继续)

(续上页)

```

injectdbiterra => '0', --do not change
-- Port B module ports
clkb => CLK(0),
rstb => RESET(0),
enb => '1',
regceb => '1',
addrb => RP,
doutb => memOut,
sbiterrb => open, --do not change
dbiterrb => open --do not change
);
end generate;

```

```

xpm_memory_tdpram_inst : xpm_memory_tdpram
generic map (
    -- Common module generics
    MEMORY_SIZE => memLength*wordWidth, --positive integer
    MEMORY_PRIMITIVE => "auto", --string; "auto", "distributed", "block" or "ultra
    -- ;
    CLOCKING_MODE => "independent_clock", --string; "common_clock", "independent_
    --clock"
    MEMORY_INIT_FILE => "none", --string; "none" or "<filename>.mem"
    MEMORY_INIT_PARAM => "", --string;
    USE_MEM_INIT => 0, --integer; 0,1
    WAKEUP_TIME => "disable_sleep", --string; "disable_sleep" or "use_sleep_pin"
    MESSAGE_CONTROL => 0, --integer;
    ECC_MODE => "no_ecc", --string; "no_ecc", "encode_only",
    --"decode_only" or "both_encode_and_decode"
    AUTO_SLEEP_TIME => 0, --Do not Change
    -- USE_EMBEDDED_CONSTRAINT => 0, --integer: 0,1
    -- MEMORY_OPTIMIZATION => "true", --string; "true", "false"

    -- Port A module generics
    WRITE_DATA_WIDTH_A => wordWidth, --positive integer
    READ_DATA_WIDTH_A => wordWidth, --positive integer
    BYTE_WRITE_WIDTH_A => wordWidth, --integer; 8, 9, or WRITE_DATA_WIDTH_A value
    ADDR_WIDTH_A => addressBits, --positive integer
    READ_RESET_VALUE_A => "0", --string
    READ_LATENCY_A => 1, --non-negative integer
    WRITE_MODE_A => "no_change", --string; "write_first", "read_first", "no_change"
    -- Port B module generics
    WRITE_DATA_WIDTH_B => wordWidth, --positive integer
    READ_DATA_WIDTH_B => wordWidth, --positive integer
    BYTE_WRITE_WIDTH_B => wordWidth, --integer; 8, 9, or WRITE_DATA_WIDTH_B value
    ADDR_WIDTH_B => addressBits, --positive integer
    READ_RESET_VALUE_B => "0", --string
    READ_LATENCY_B => 1, --non-negative integer
    WRITE_MODE_B => "read_first" --string; "write_first", "read_first", "no_change"
)
port map (
    -- Common module ports
    sleep => '0',
    -- Port A module ports
    clka => CLK_WRITE(0),
    rsta => RESET(0),
    ena => '1',
    regcea => '1',
    wea => wea,
    addra => addra,
    dina => dina,
    injectsbiterra => '0', --do not change

```

(下页继续)

(续上页)

```
injectdbiterra => '0', --do not change
douta => douta,
sbiterra => open, --do not change
dbiterra => open, --do not change
-- Port B module ports
clkb => CLK_READ(0),
rstb => RESET(0),
enb => '1',
regceb => '1',
web => web,
addrb => READ_ADDRESS(addressBits-1 downto 0),
dinb => dinb,
injectsbiterrb => '0', --do not change
injectdbiterrb => '0', --do not change
doutb => doutb,
sbiterrb => open, --do not change
dbiterrb => open --do not change
);
```

9.5 attribute

<https://www.cnblogs.com/christsong/p/5793213.html>

<https://www.jianshu.com/p/ea785a8c8240>

<https://wenku.baidu.com/view/dd2f122cb4daa58da0114a0a.html>