

---

# FPGA 在中低能实验核物理中的应用

发布 *1.0 alpha*

Hongyi Wu(吴鸿毅)

2020 年 01 月 03 日



---

## Contents:

---

<b>1</b>	<b>README</b>	<b>1</b>
<b>2</b>	<b>ISE/Altera/Vivado 软件</b>	<b>3</b>
2.1	软件安装 . . . . .	3
<b>3</b>	<b>语法</b>	<b>29</b>
3.1	VHDL . . . . .	29
3.2	verilog . . . . .	42
<b>4</b>	<b>硬件介绍</b>	<b>43</b>
4.1	LUPO . . . . .	43
4.2	DT5495 . . . . .	43
4.3	MZTIO . . . . .	43
<b>5</b>	<b>计数器</b>	<b>45</b>
5.1	计数器 . . . . .	45
<b>6</b>	<b>状态机</b>	<b>51</b>
6.1	状态机 . . . . .	51
<b>7</b>	<b>FIFO</b>	<b>53</b>
7.1	FIFO . . . . .	53
<b>8</b>	<b>经验总结</b>	<b>55</b>
8.1	经验总结 . . . . .	55
8.2	临时存放 . . . . .	56



国际上最大的两个 FPGA 厂家为 Xilinx 和 Altea (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 刚好对应三个软件。

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

- 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 PCIE 的在线监视 (将获取的数据直接发送到可编程 PCIE 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。。。这个难度有点大)

- FIFO

- \* 信号延迟 (简单)

- 脉冲发生器

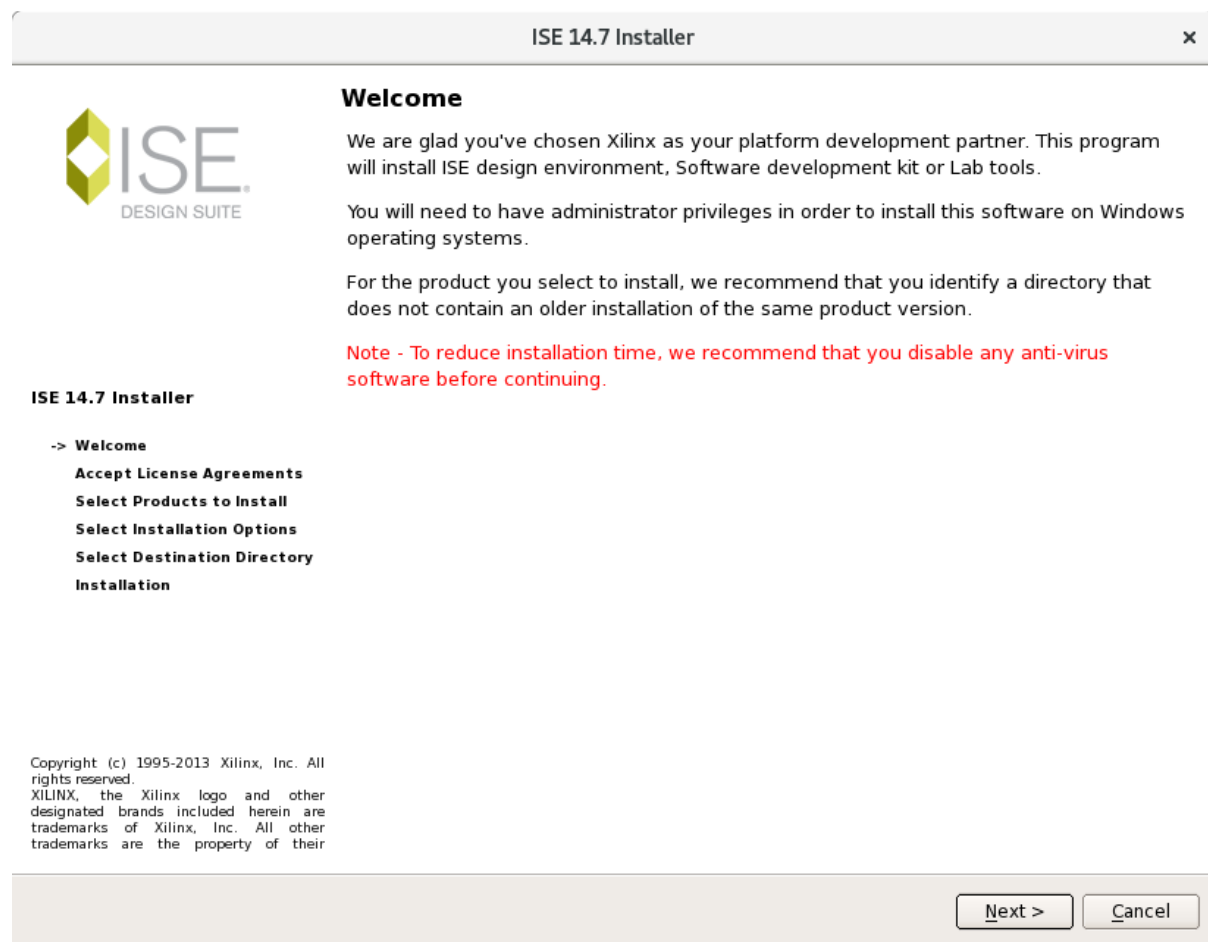
- \* 基本波形的脉冲产生 (简单)
- \* 任意波形的脉冲产生 (需要借助外部输入进行控制, 因此需要利用 UART/SPI/IIC 等通讯技术)

-

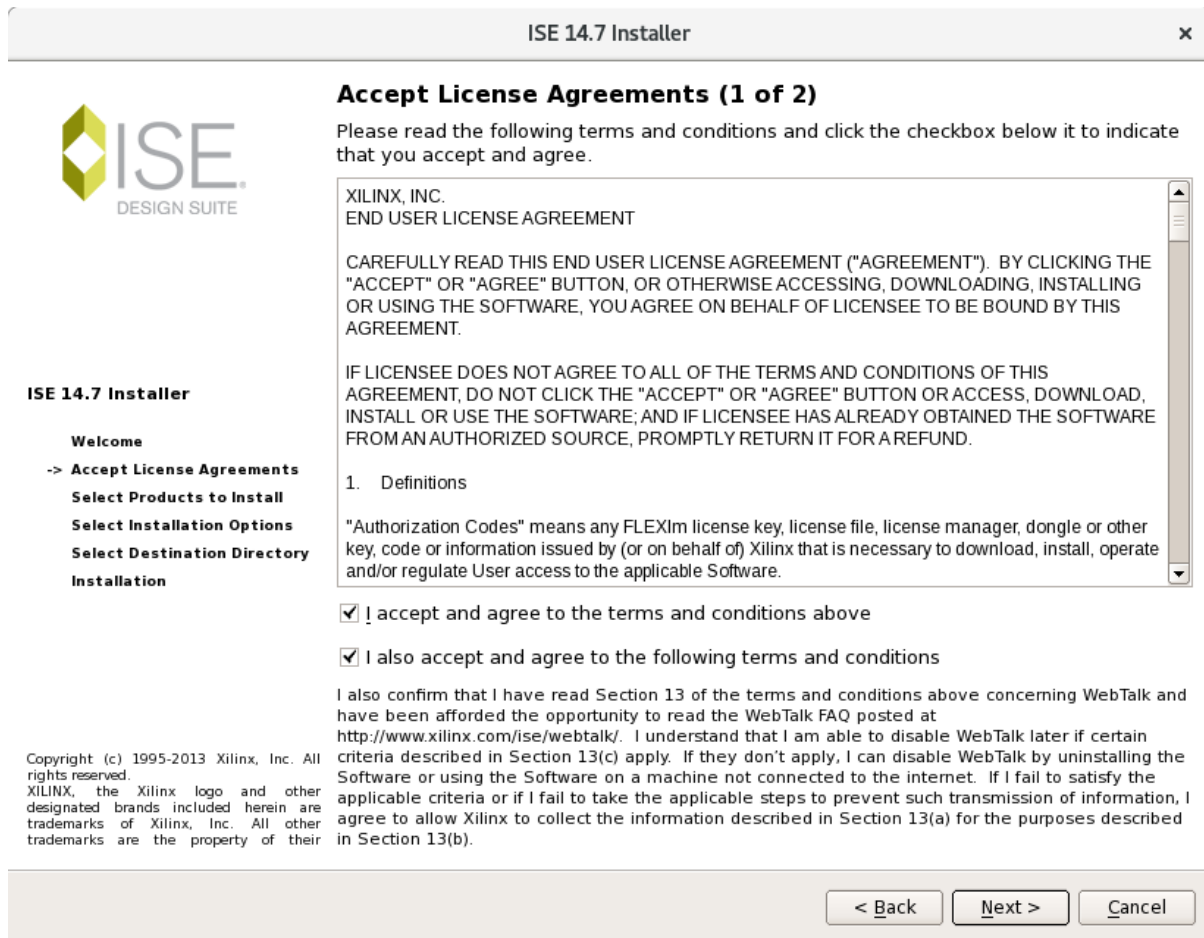
## 2.1 软件安装

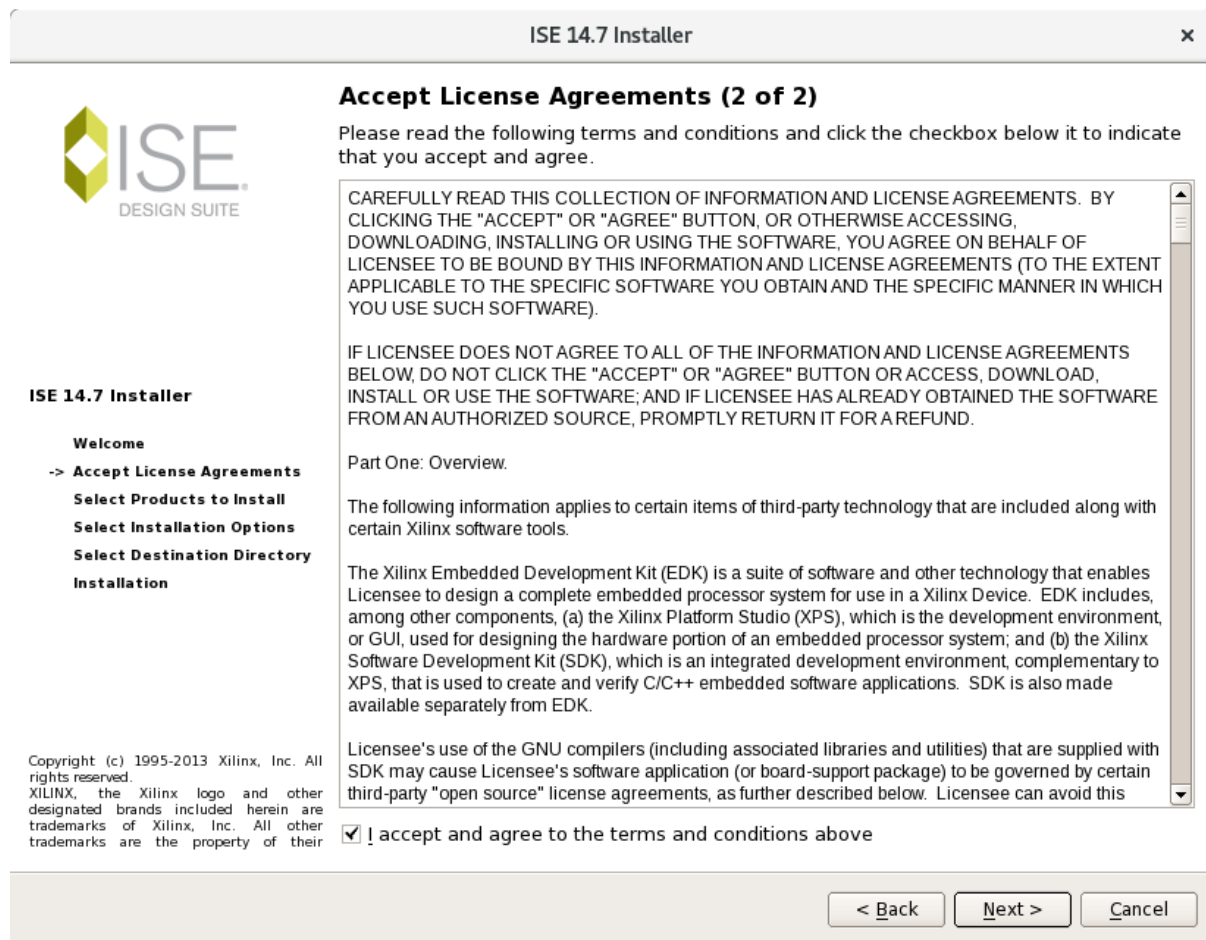
### 2.1.1 ISE 安装

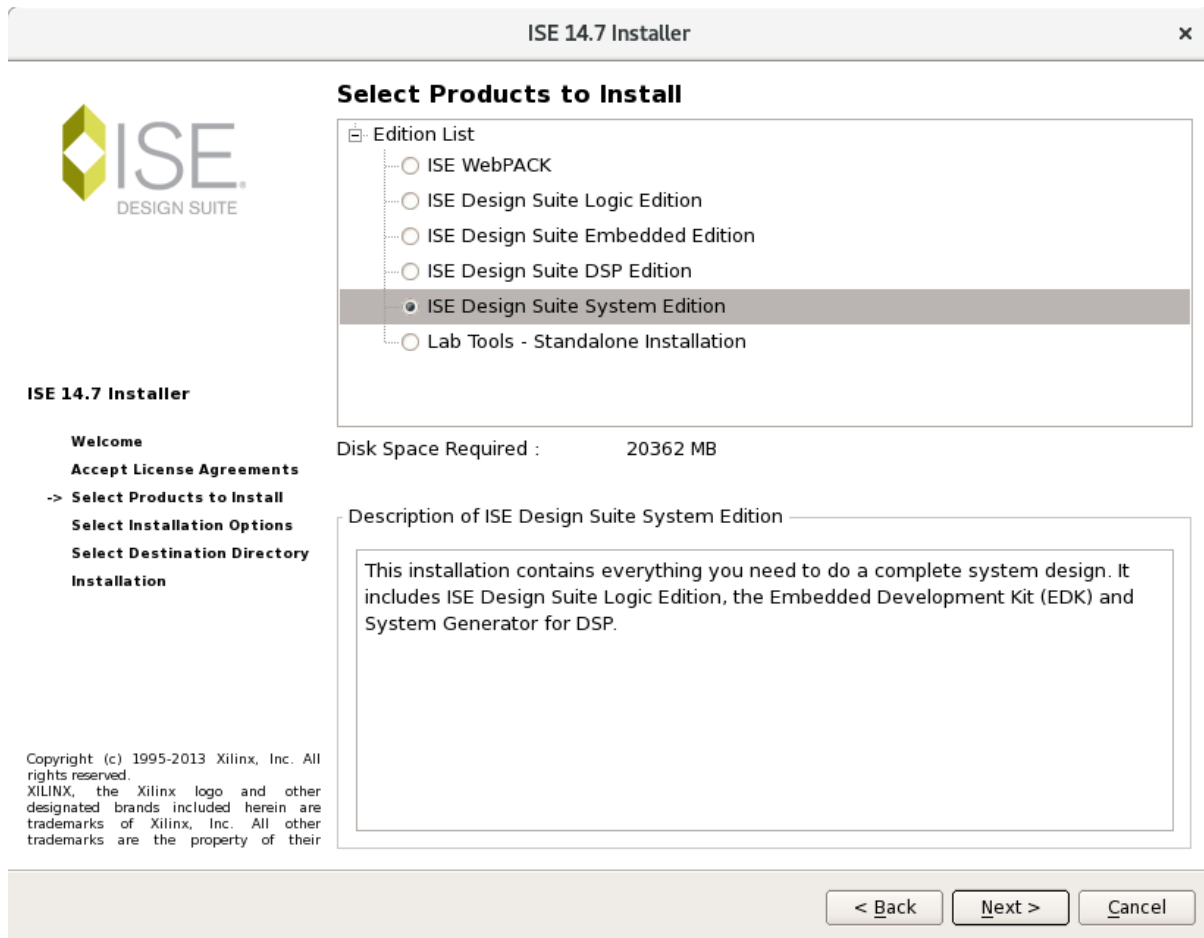
```
tar -xvf Xilinx_ISE_DS_Lin_14.7_1015_1.tar
cd Xilinx_ISE_DS_Lin_14.7_1015_1
./xsetup
```

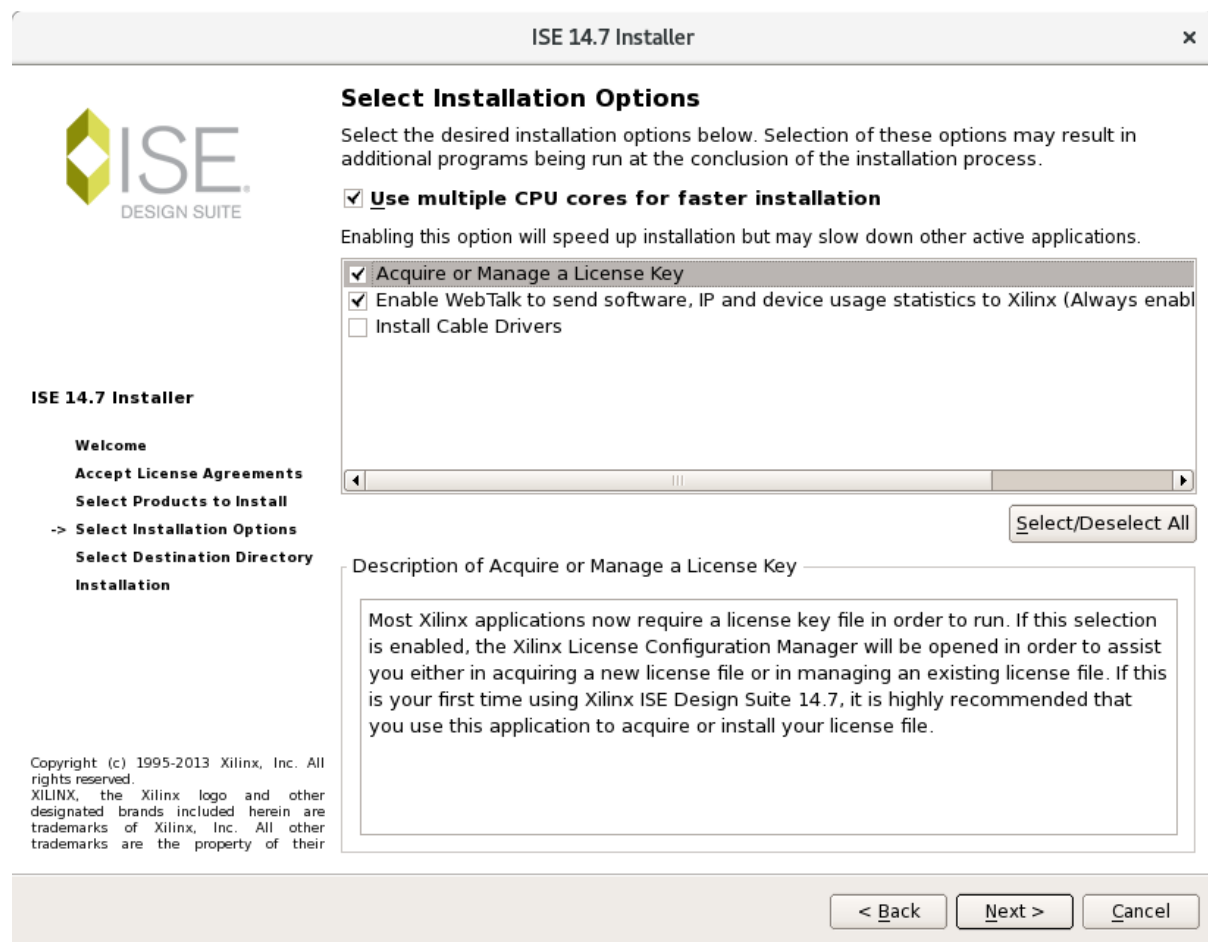


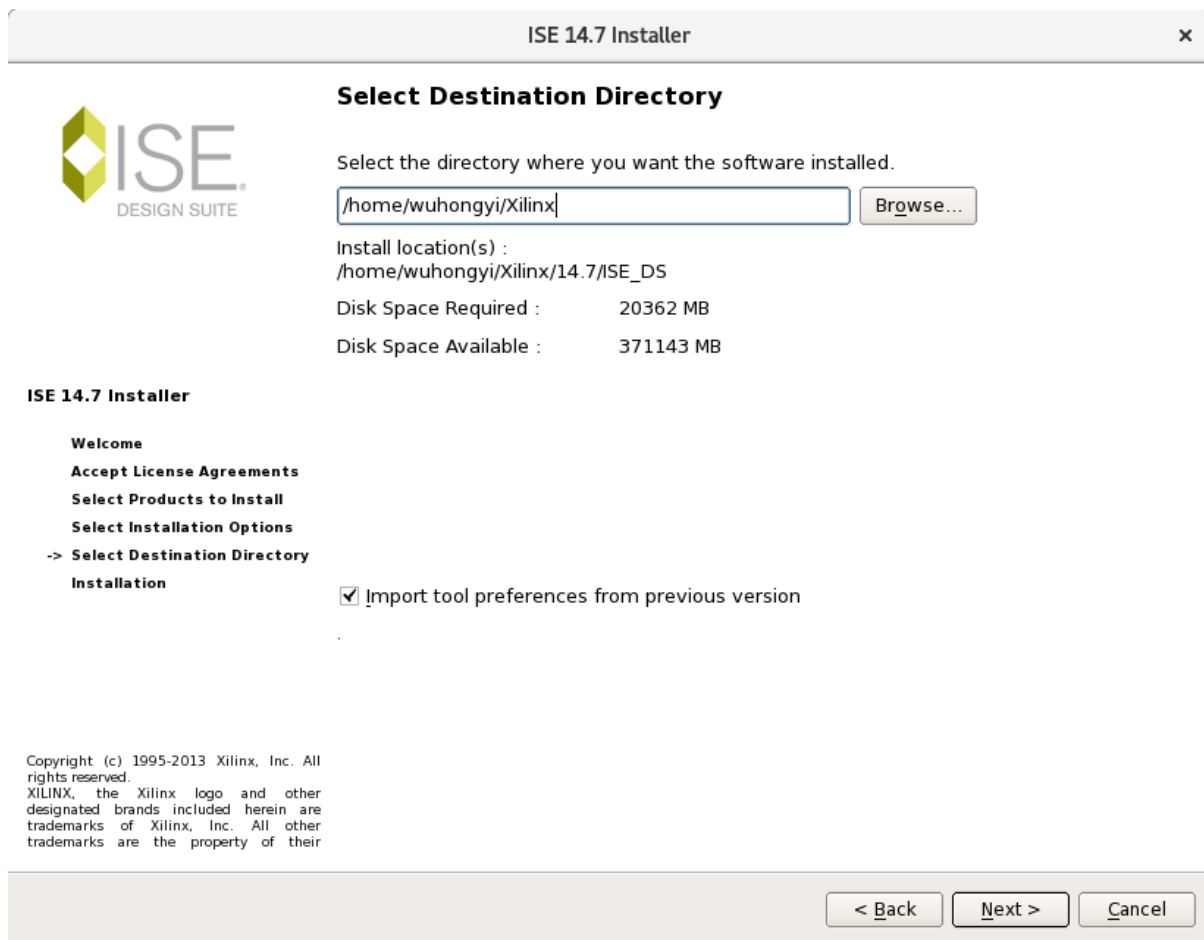


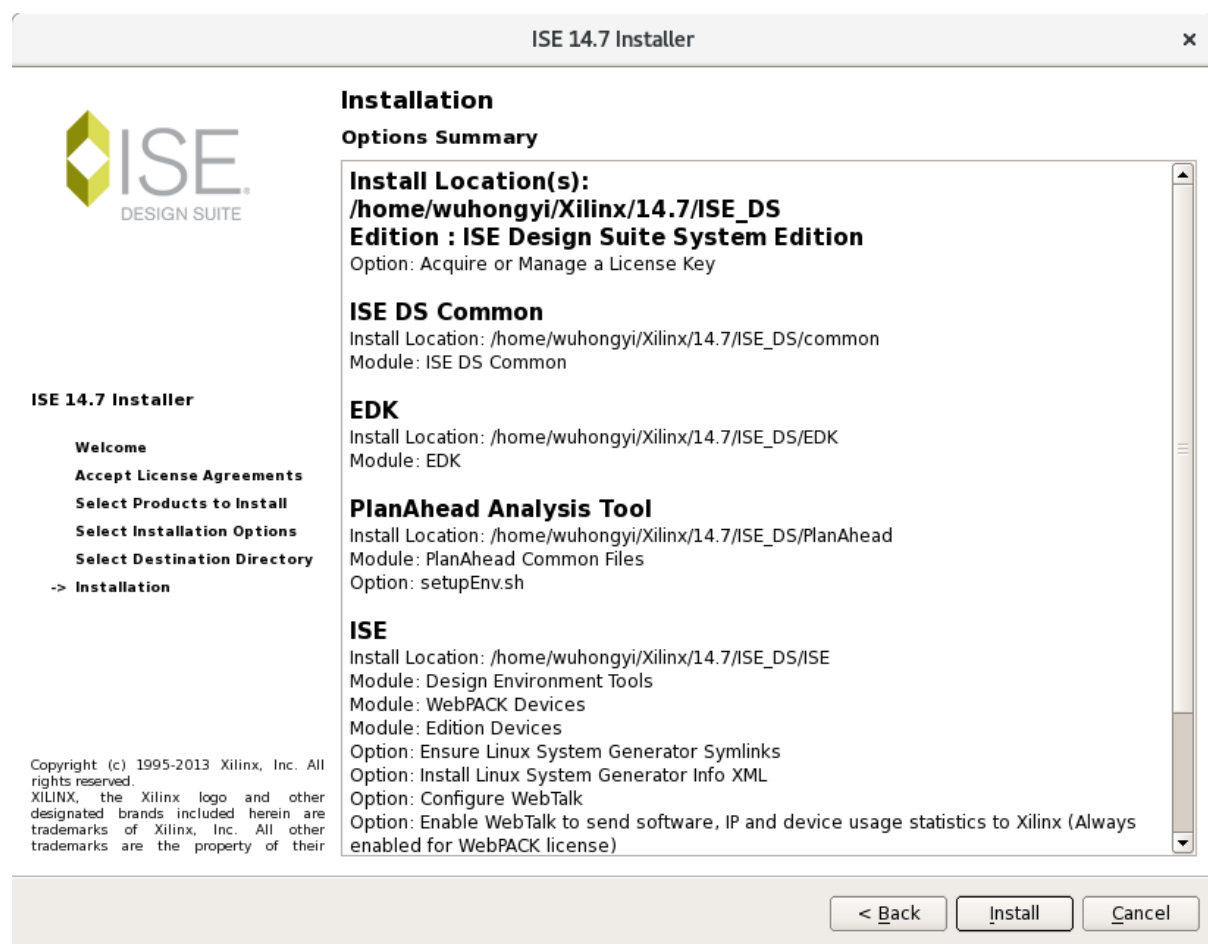


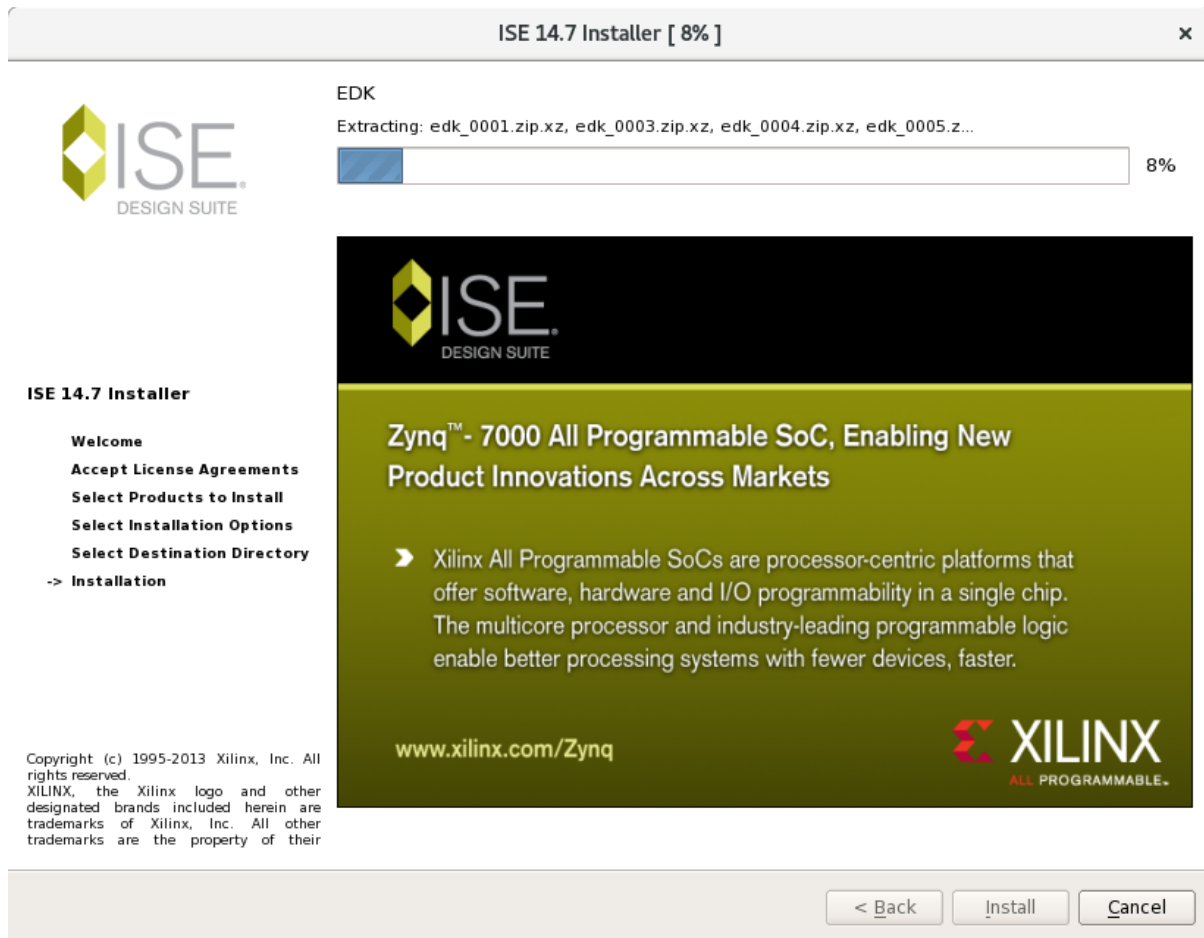










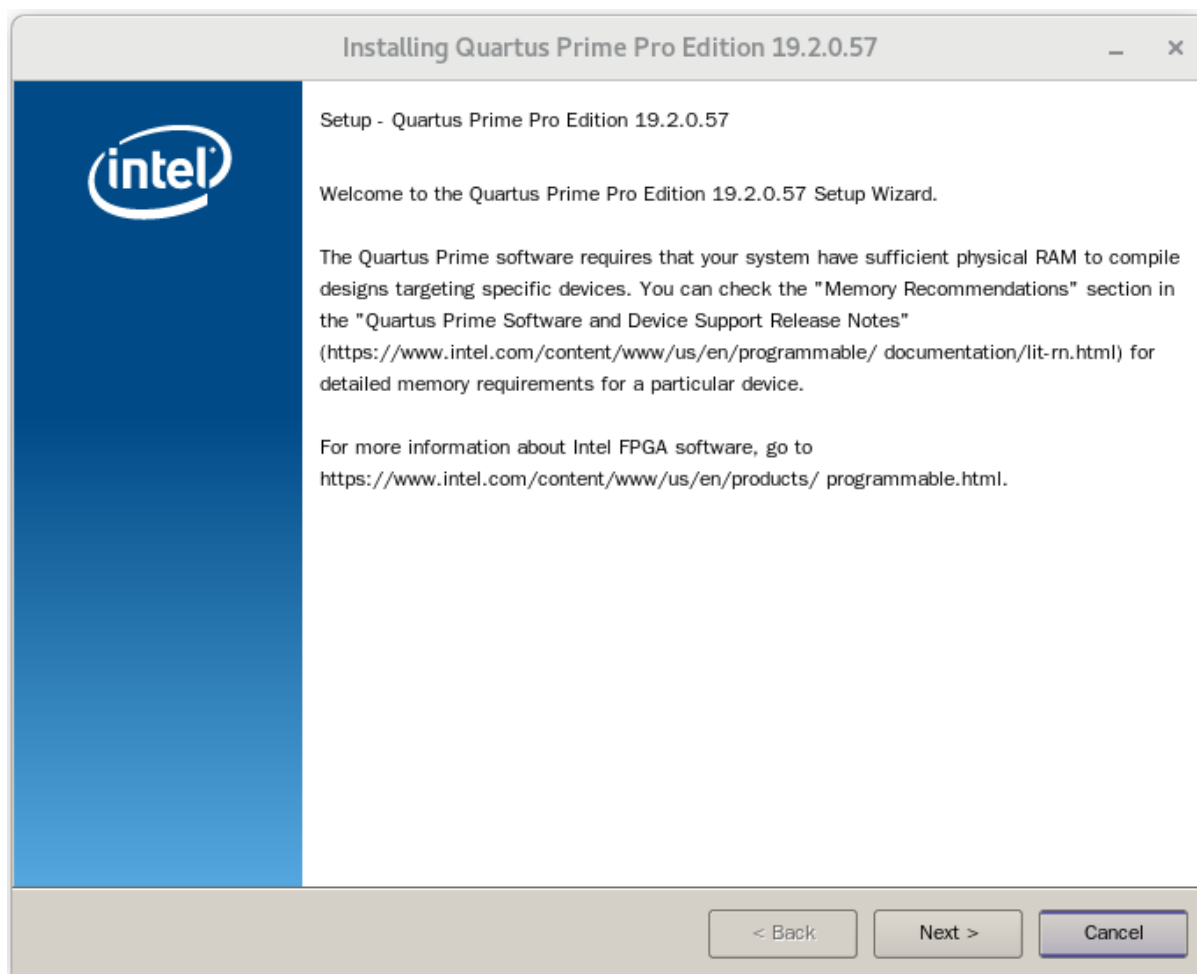


## 2.1.2 Altera 安装

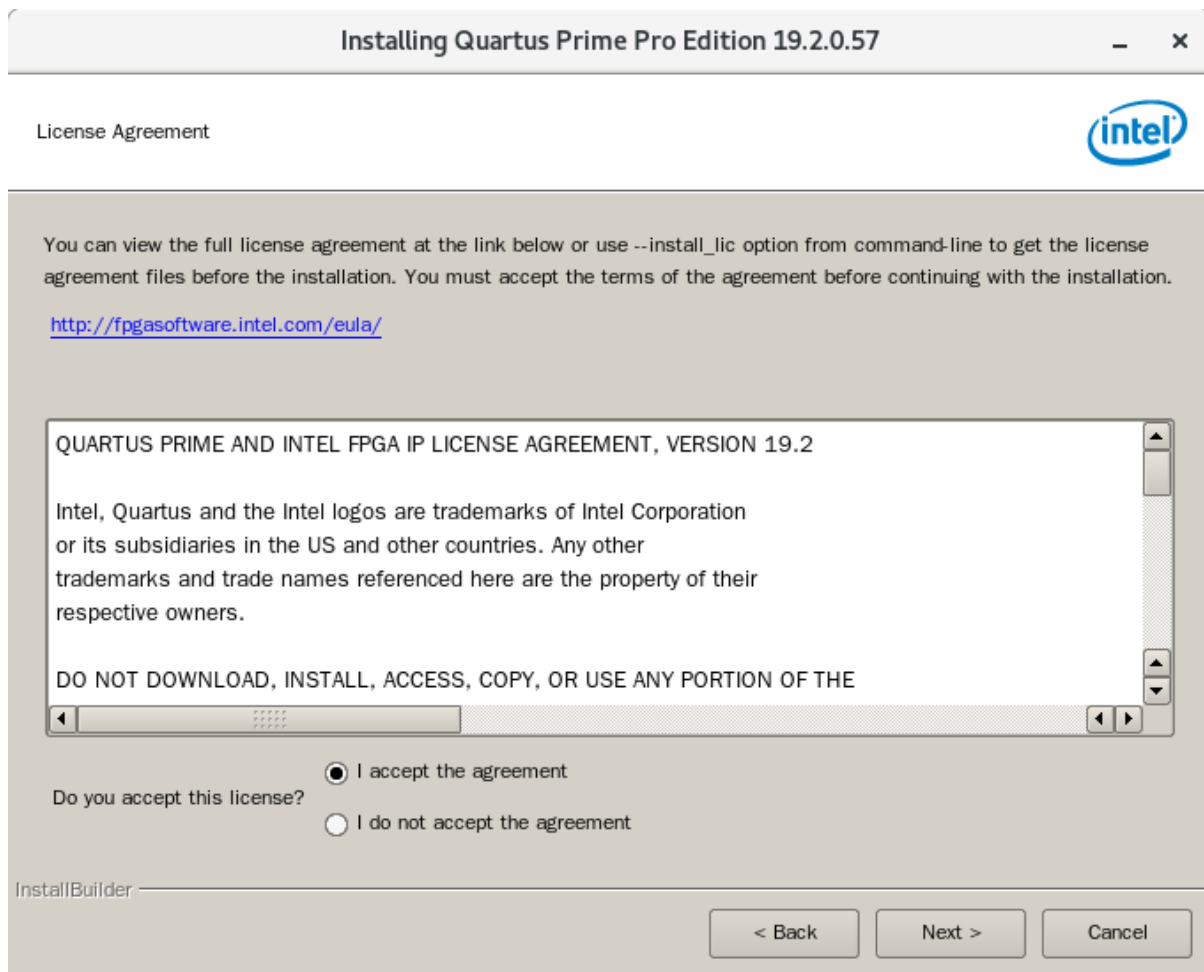
下载需要的所有文件，放在一个目录下

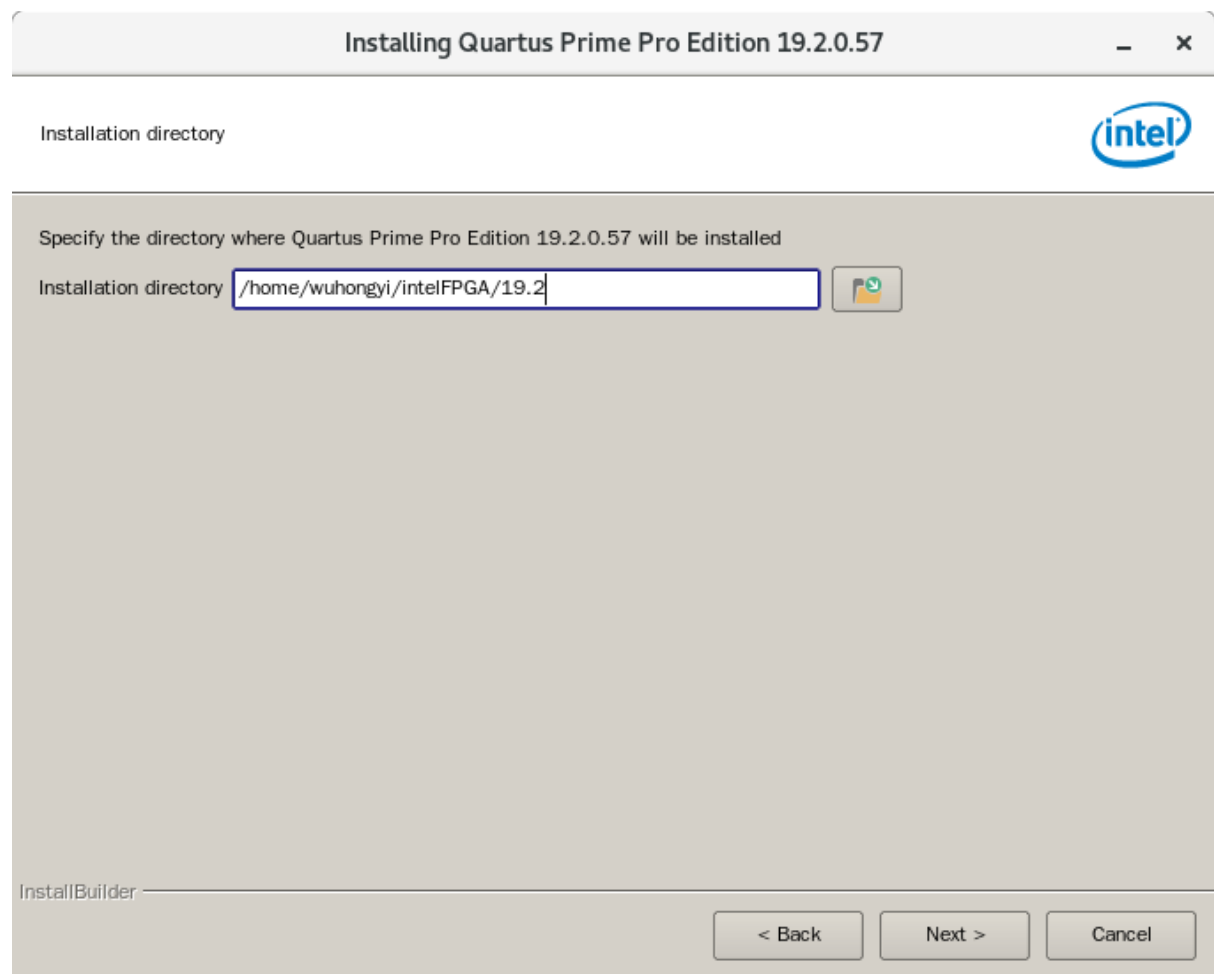
arria10-19.2.0.57.qdz	ModelSimProSetup-19.2.0.57-linux.run
cyclone10gx-19.2.0.57.qdz	QuartusProSetup-19.2.0.57-linux.run
modelsim-part2-19.2.0.57-linux.qdz	stratix10-19.2.0.57.qdz

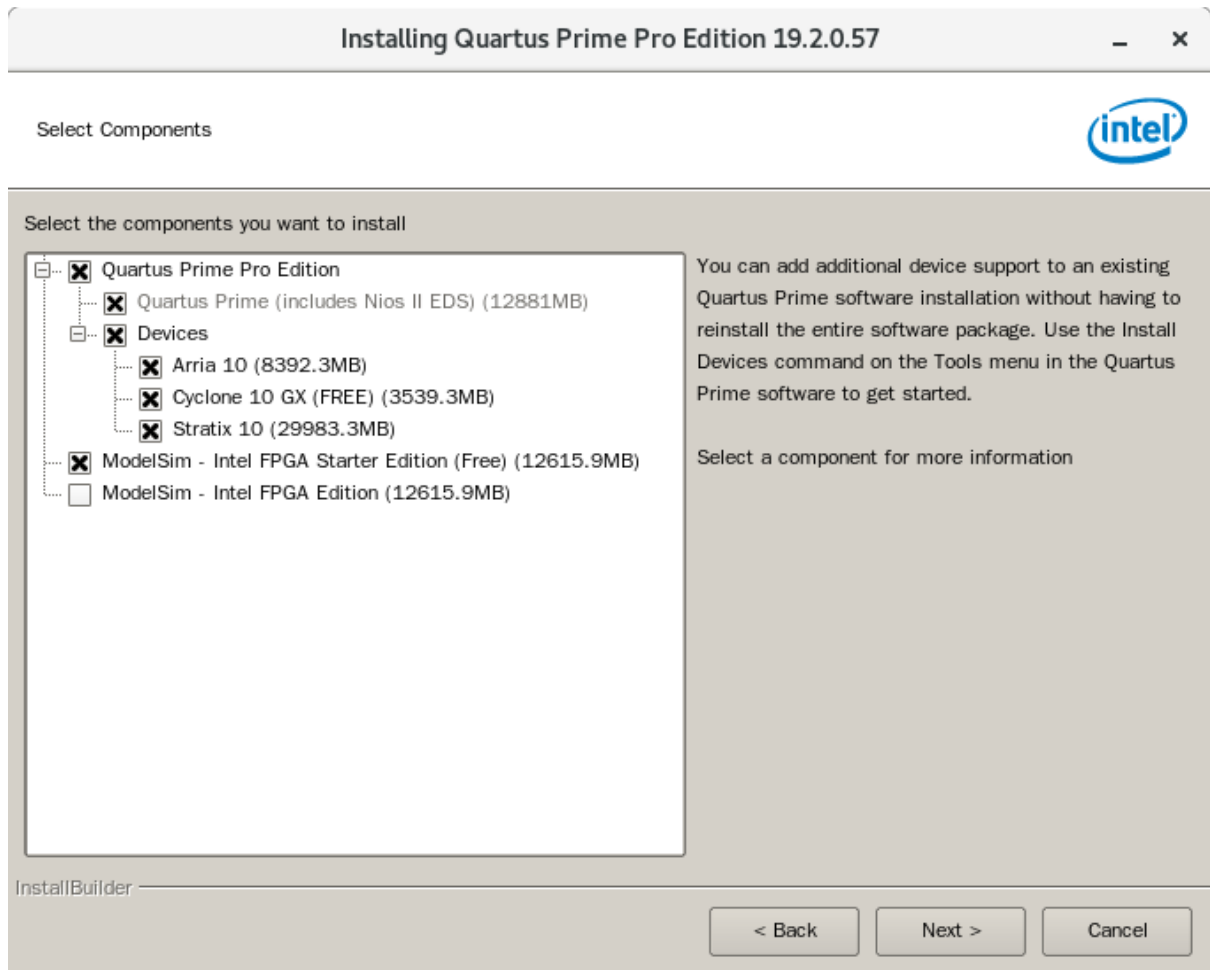
```
chmod +x QuartusProSetup-19.2.0.57-linux.run
./QuartusProSetup-19.2.0.57-linux.run
```

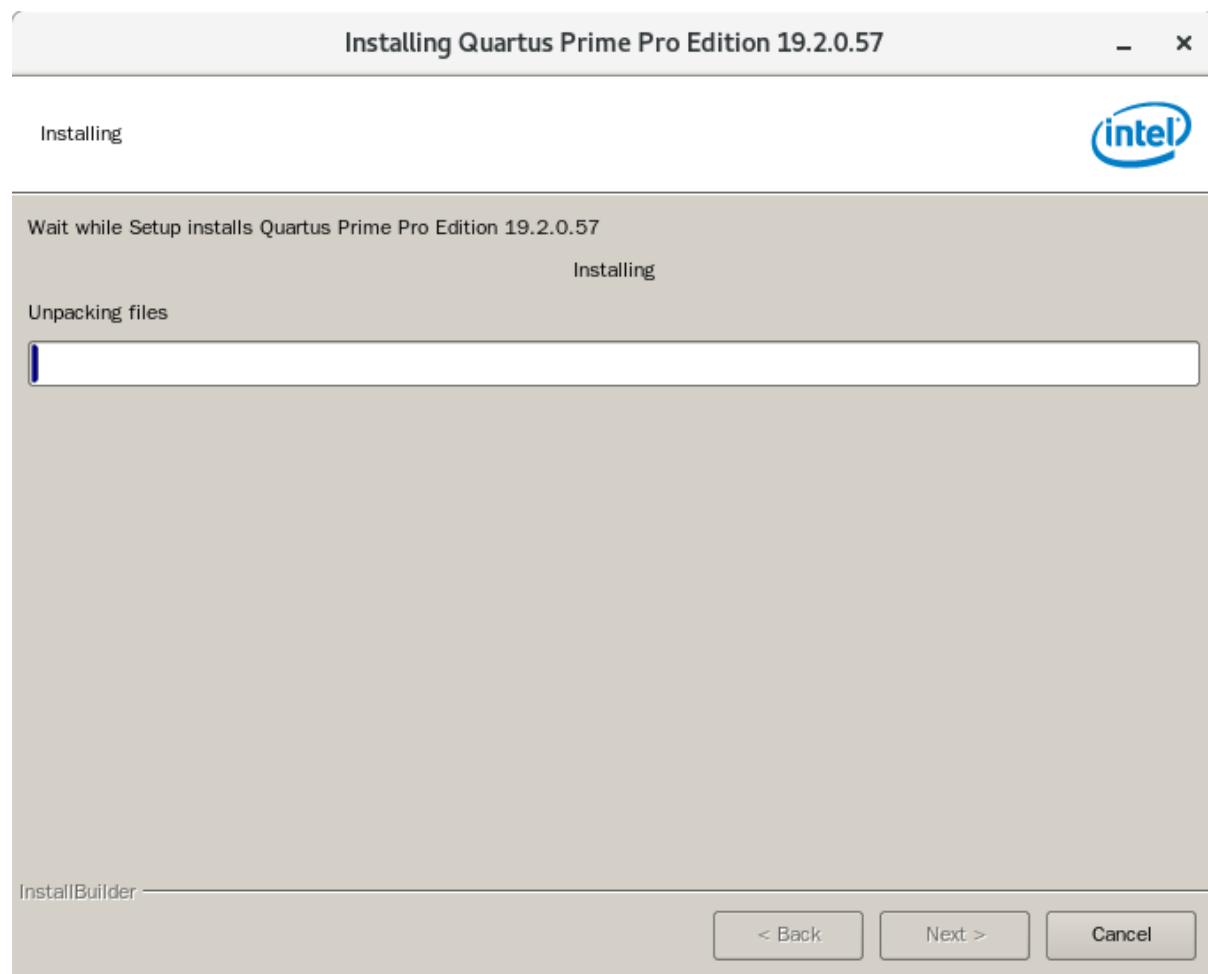


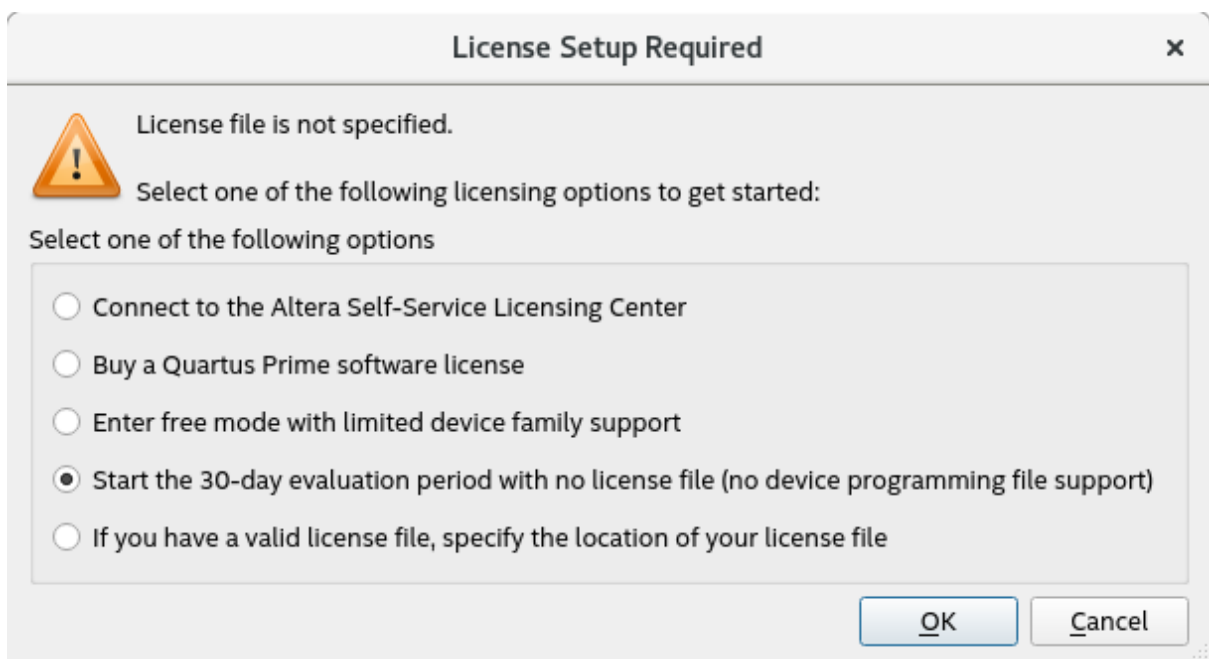
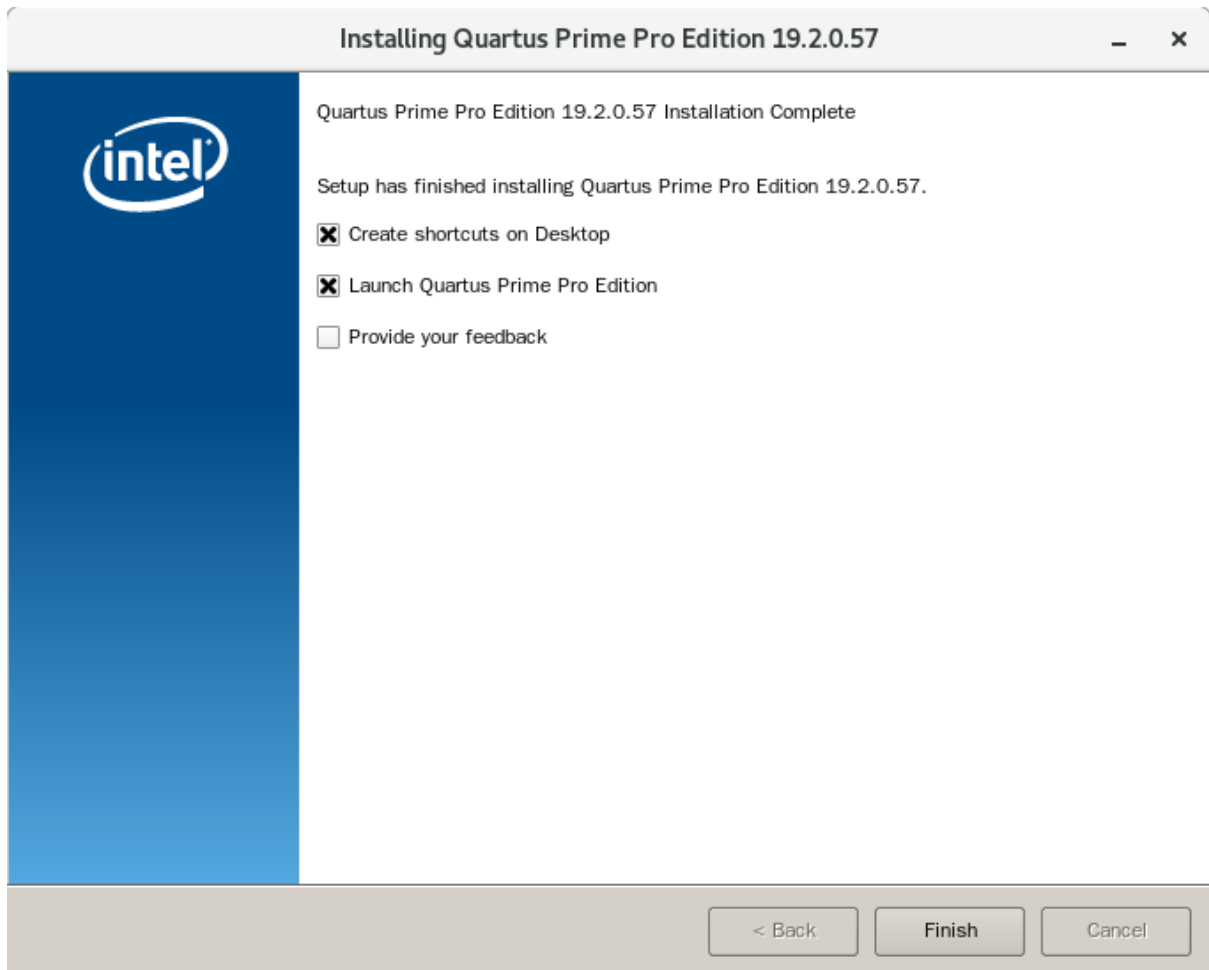












在安装路径下有以下文件

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
```

### 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

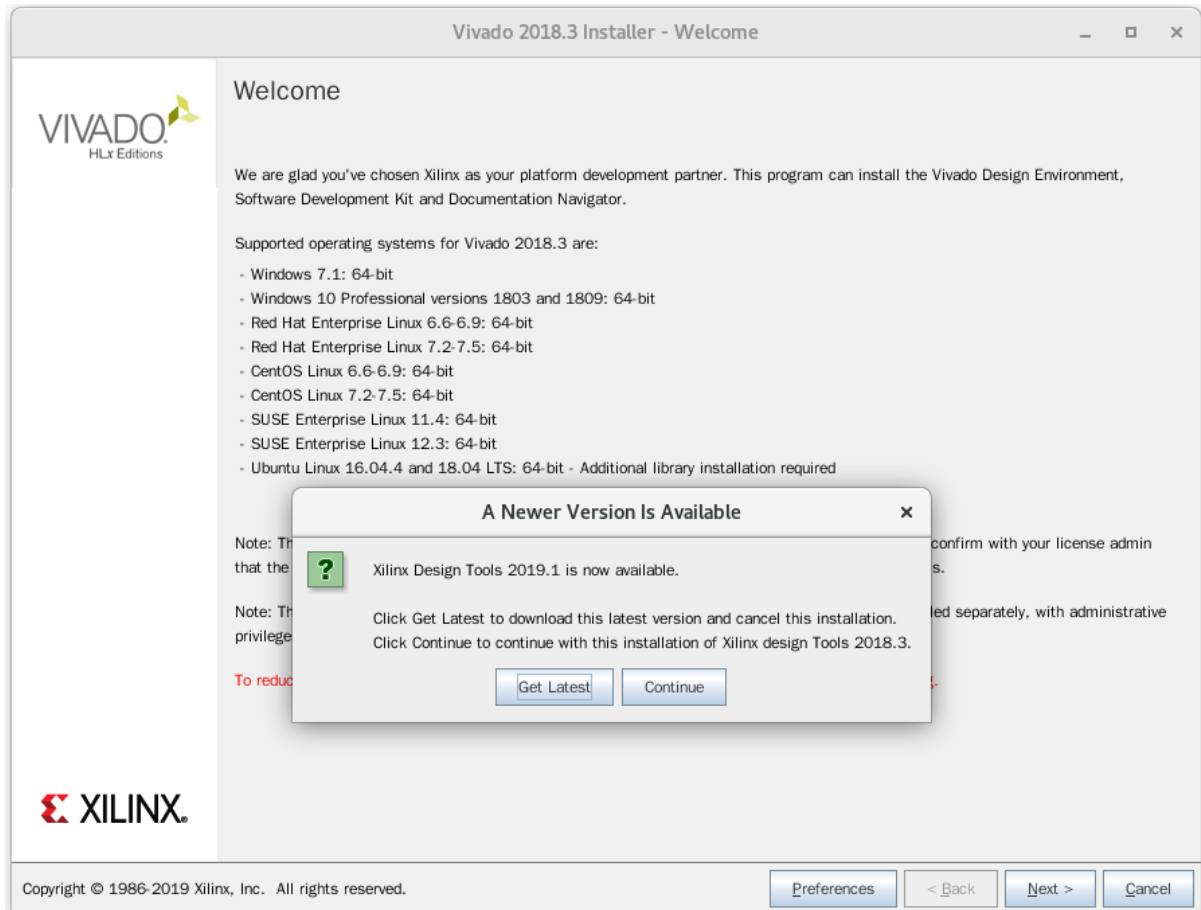
```
[root@localhost 003]# lsusb
Bus 002 Device 002: ID 8087:8000 Intel Corp.
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 002: ID 8087:8008 Intel Corp.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 004 Device 001: ID 1d6b:0003 Linux Foundation 3.0 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

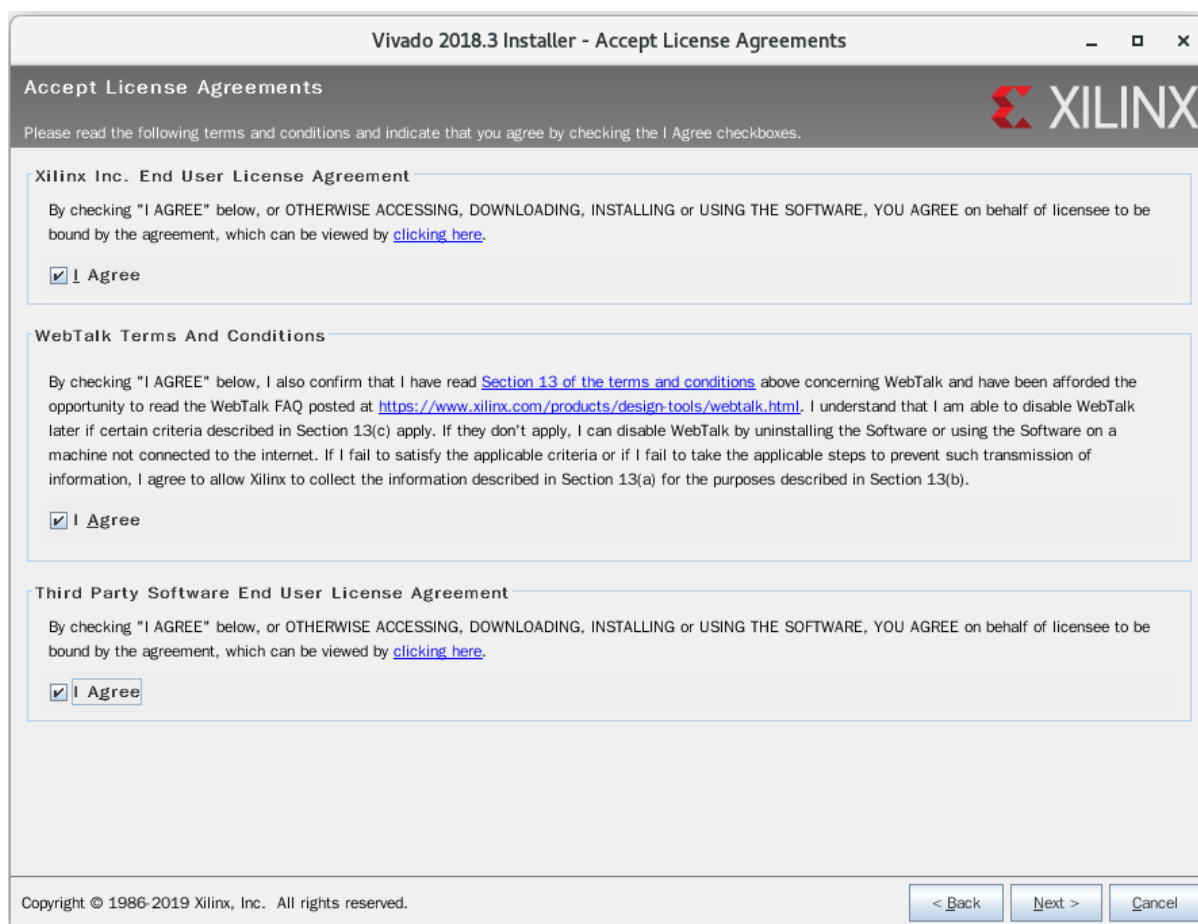
```
cd /dev/bus/usb/003
chmod 666 013
```

### 2.1.3 Vivado 安装

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

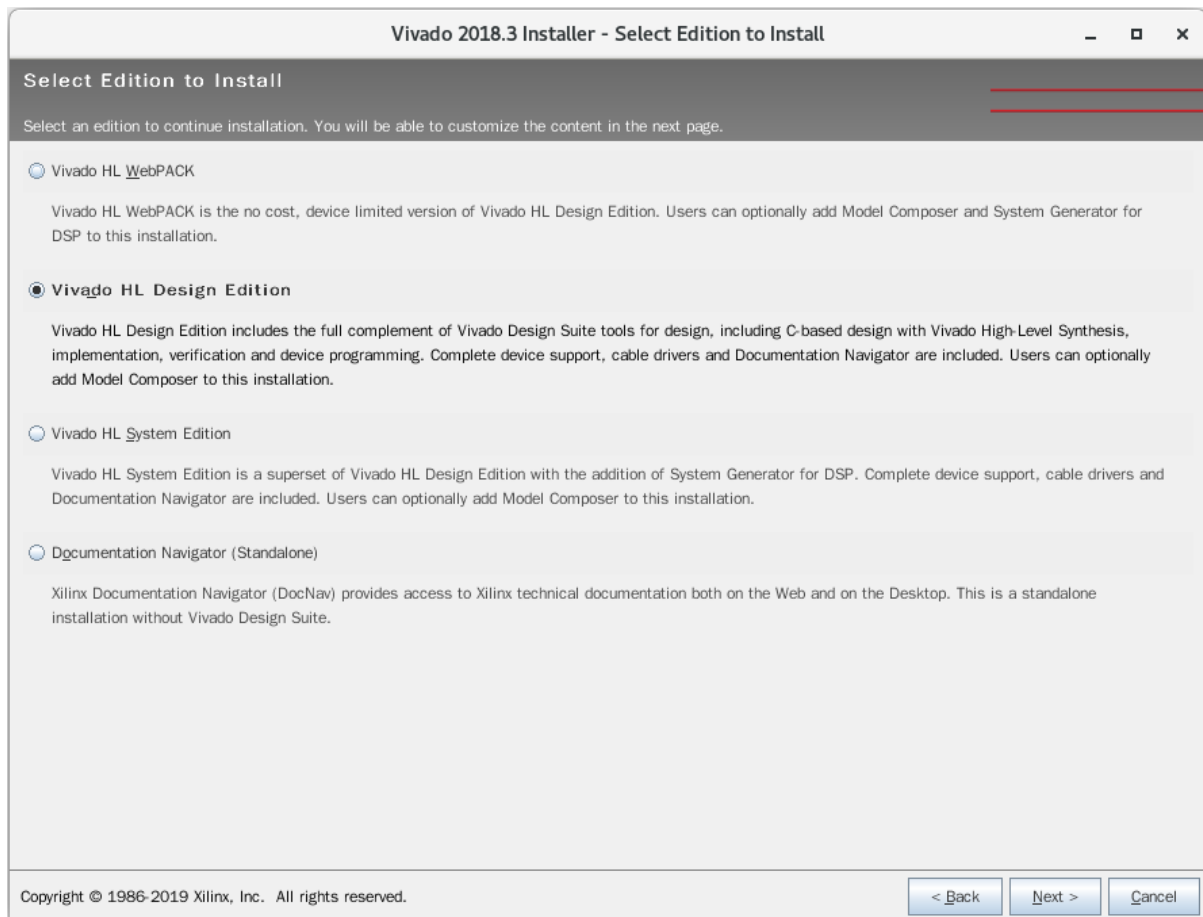


点击 continue 选择不下载最新版本，然后点击 Next 进入下一步

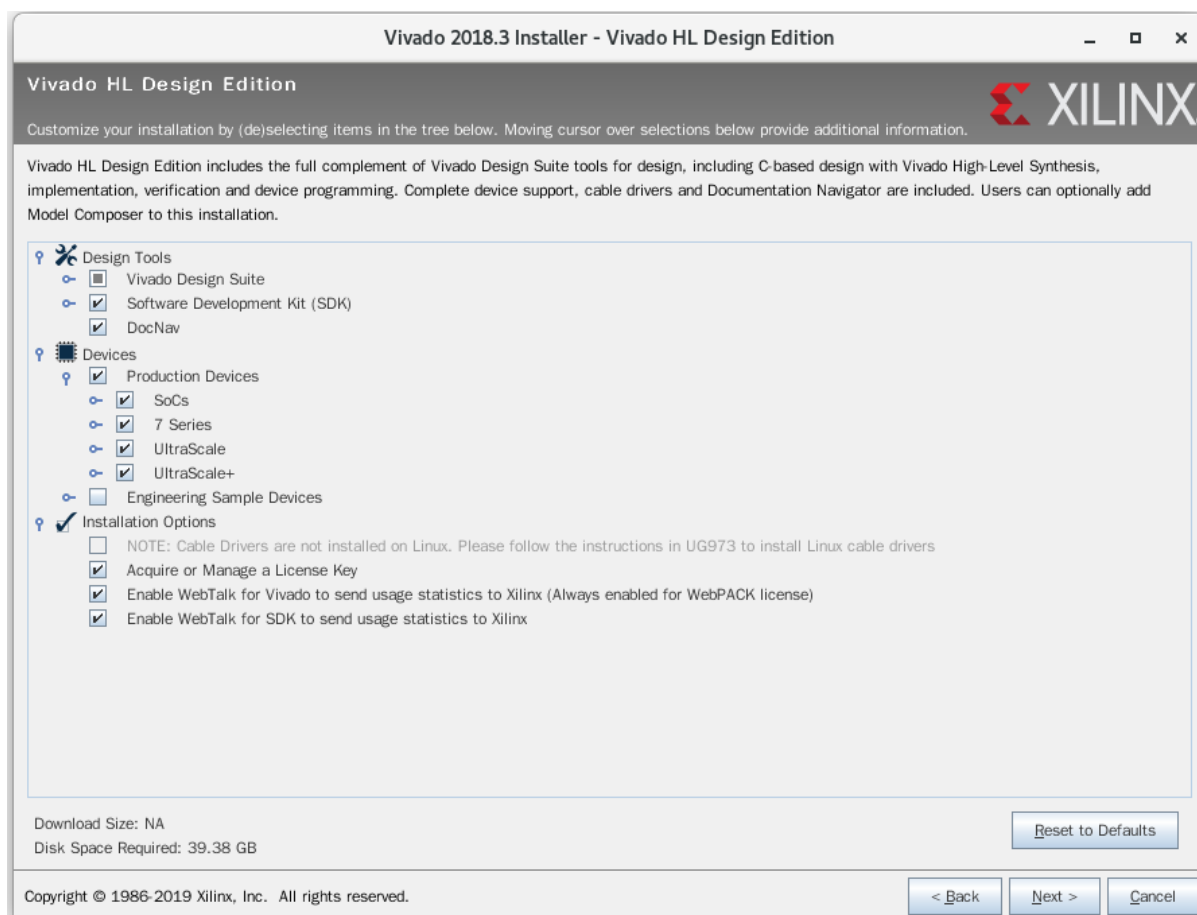


点击三个可选框，然后点击 Next 进入下一步

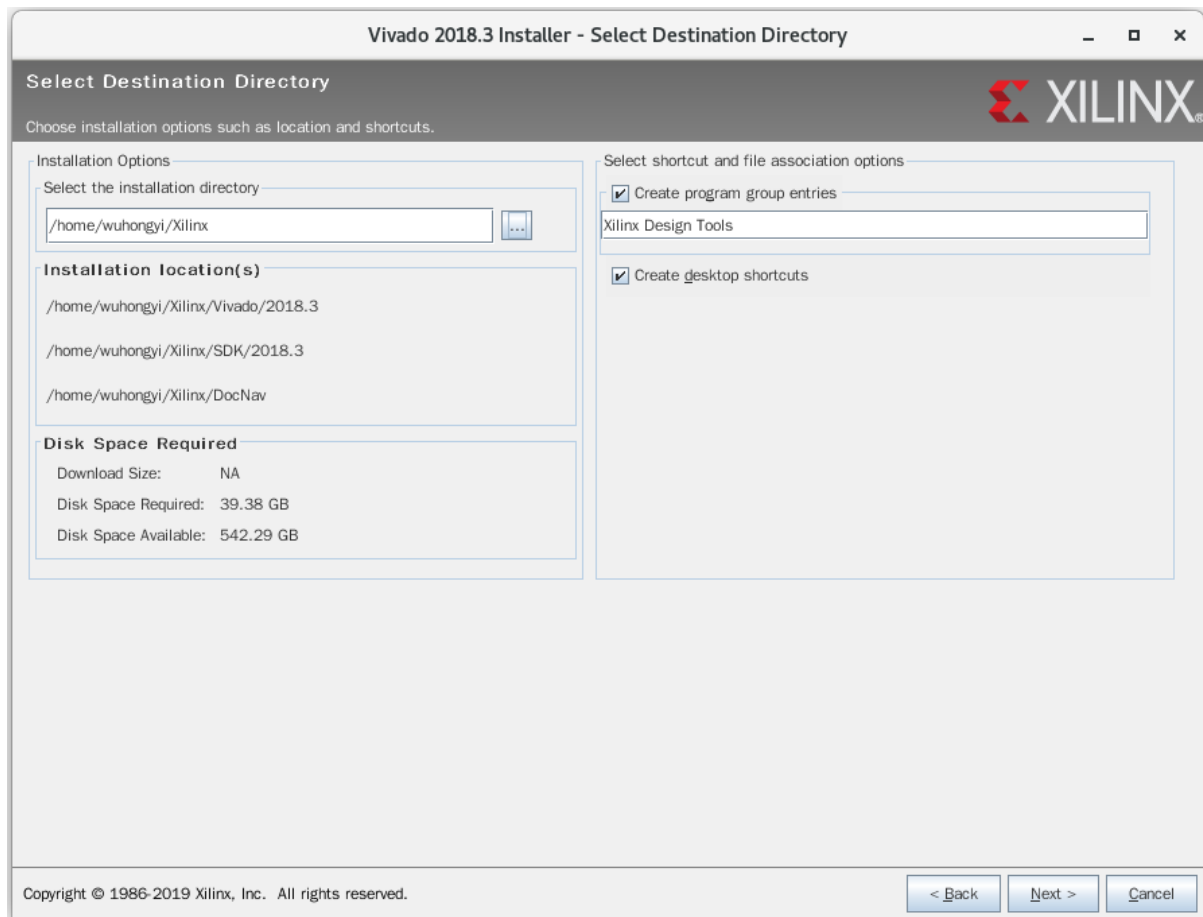




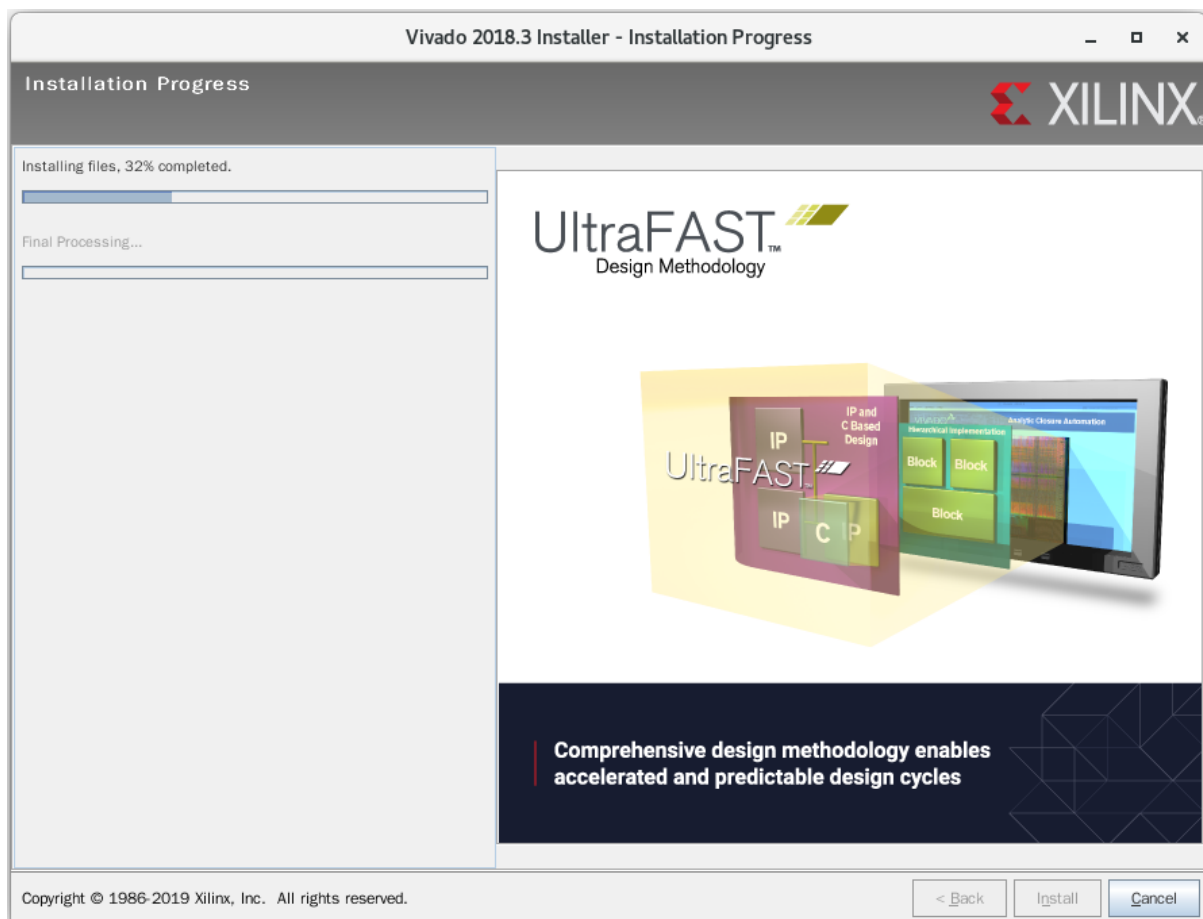
选择 Vivado HL Design Edition，然后点击 Next 进入下一步



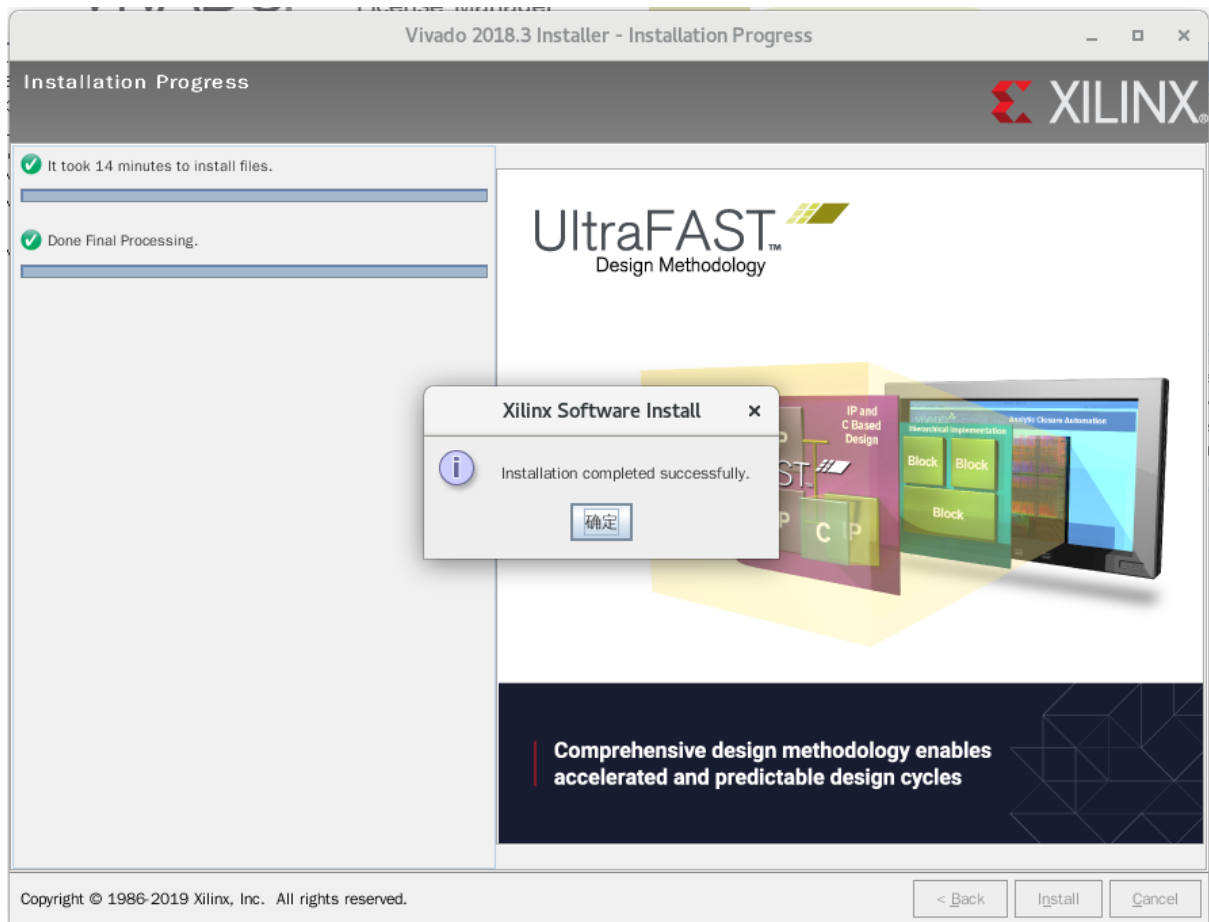
直接点击 Next 进入下一步



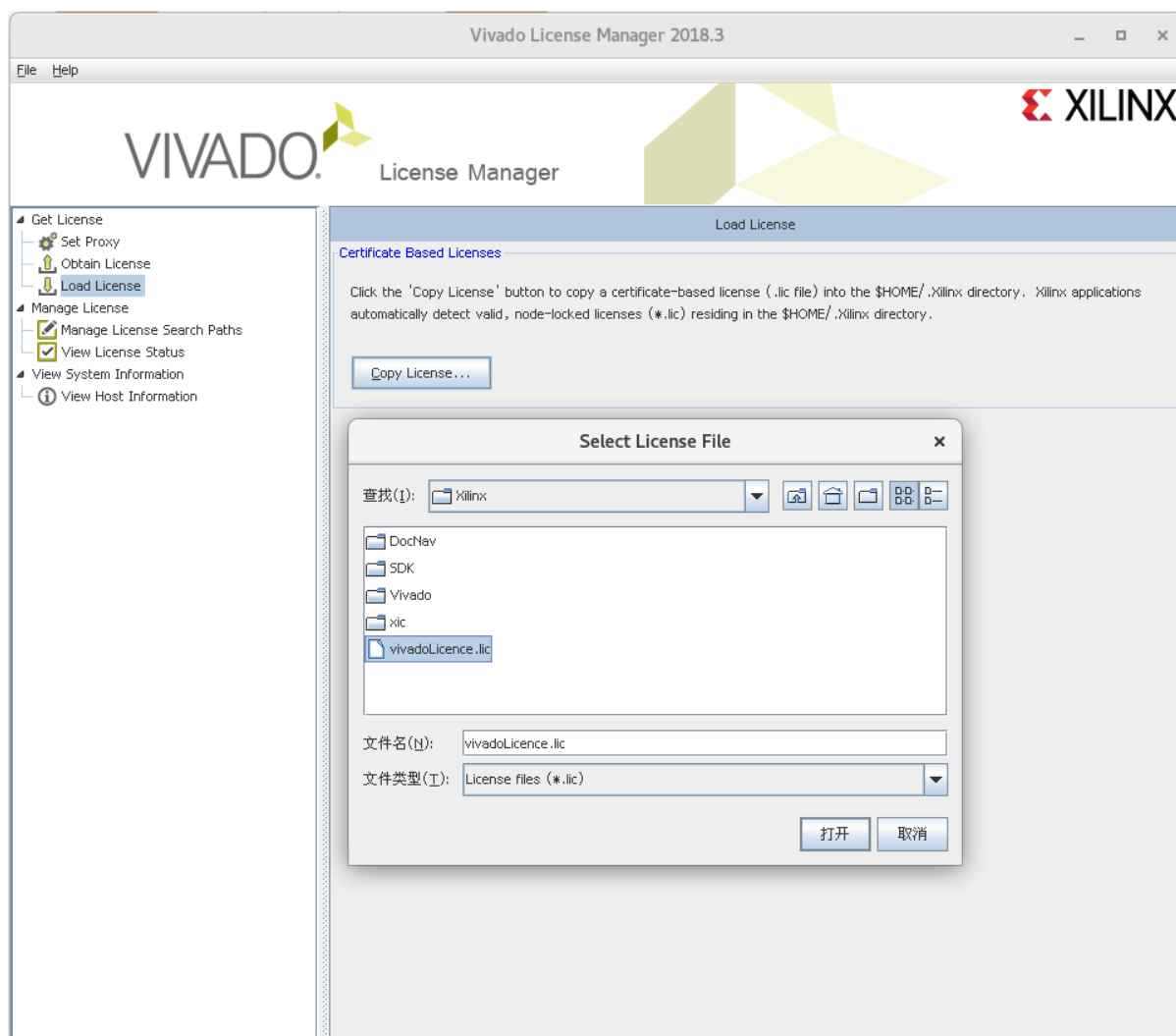
选择安装目录，这里我选择安装到 /home/wuhongyi/Xilinx，然后点击 Next 进入下一步



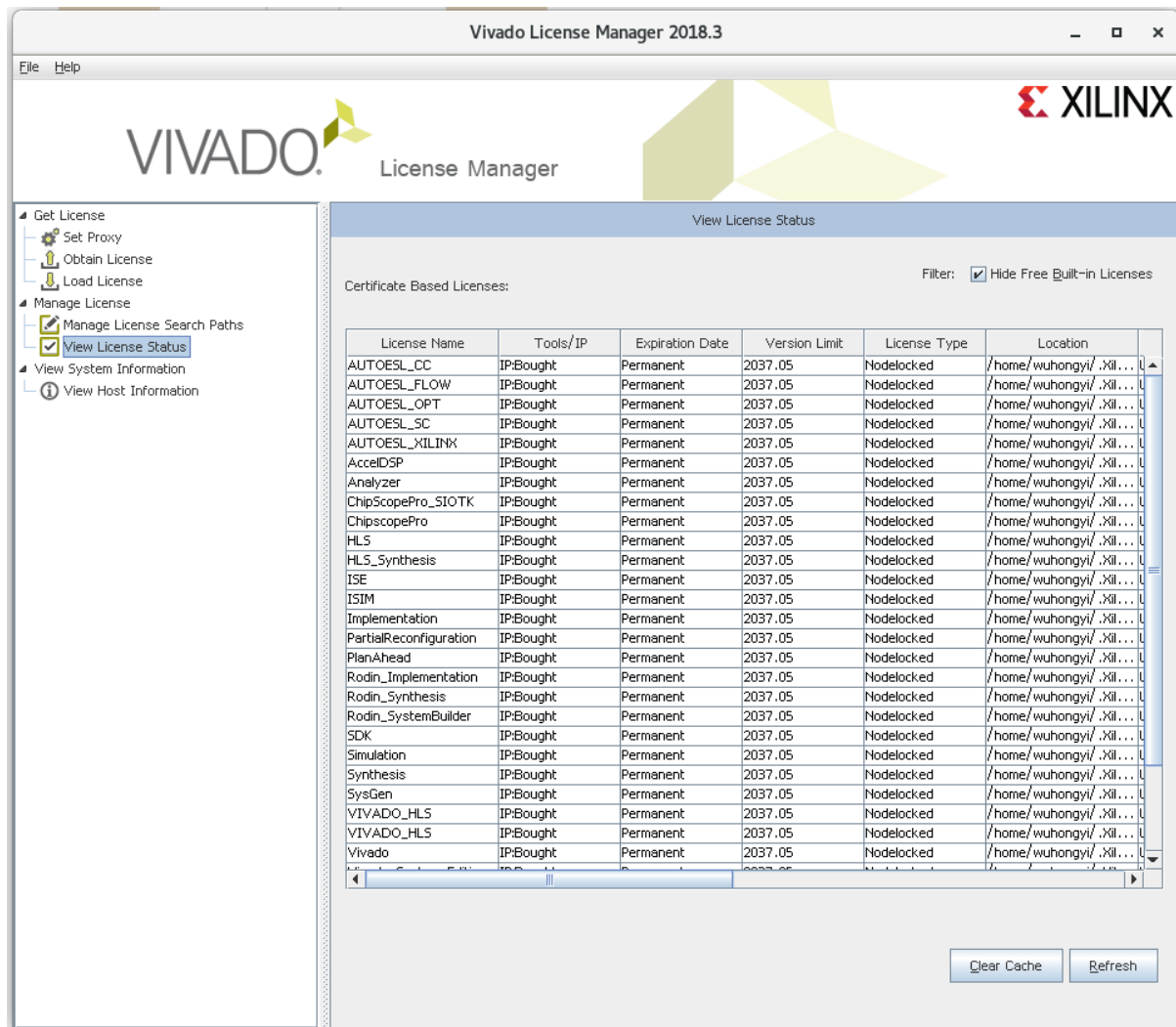
等待安装完成



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







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  $-(2^{31}-1) \sim (2^{31}-1)$ ，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  $-1.0E38 \sim +1.0E38$ ，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升序排列，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
```

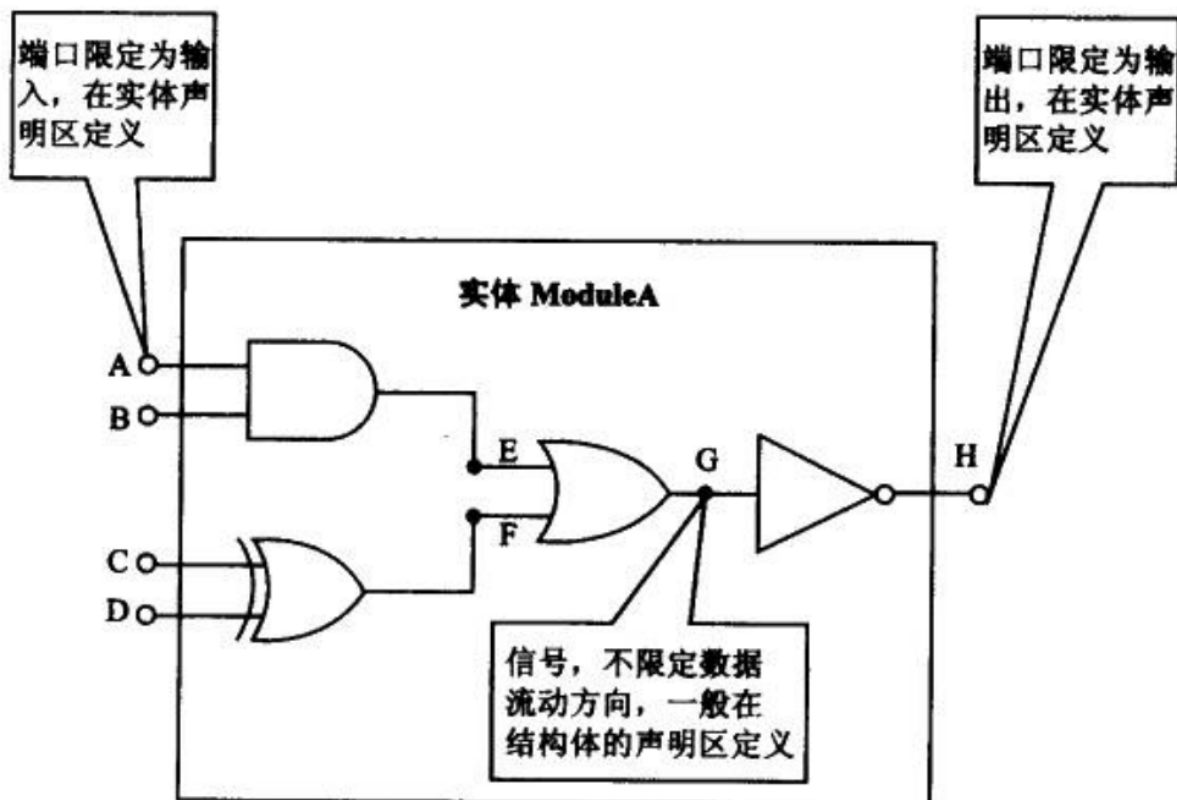
## 3.1.3 数据对象

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连接。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; --矢量于矢量并置
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

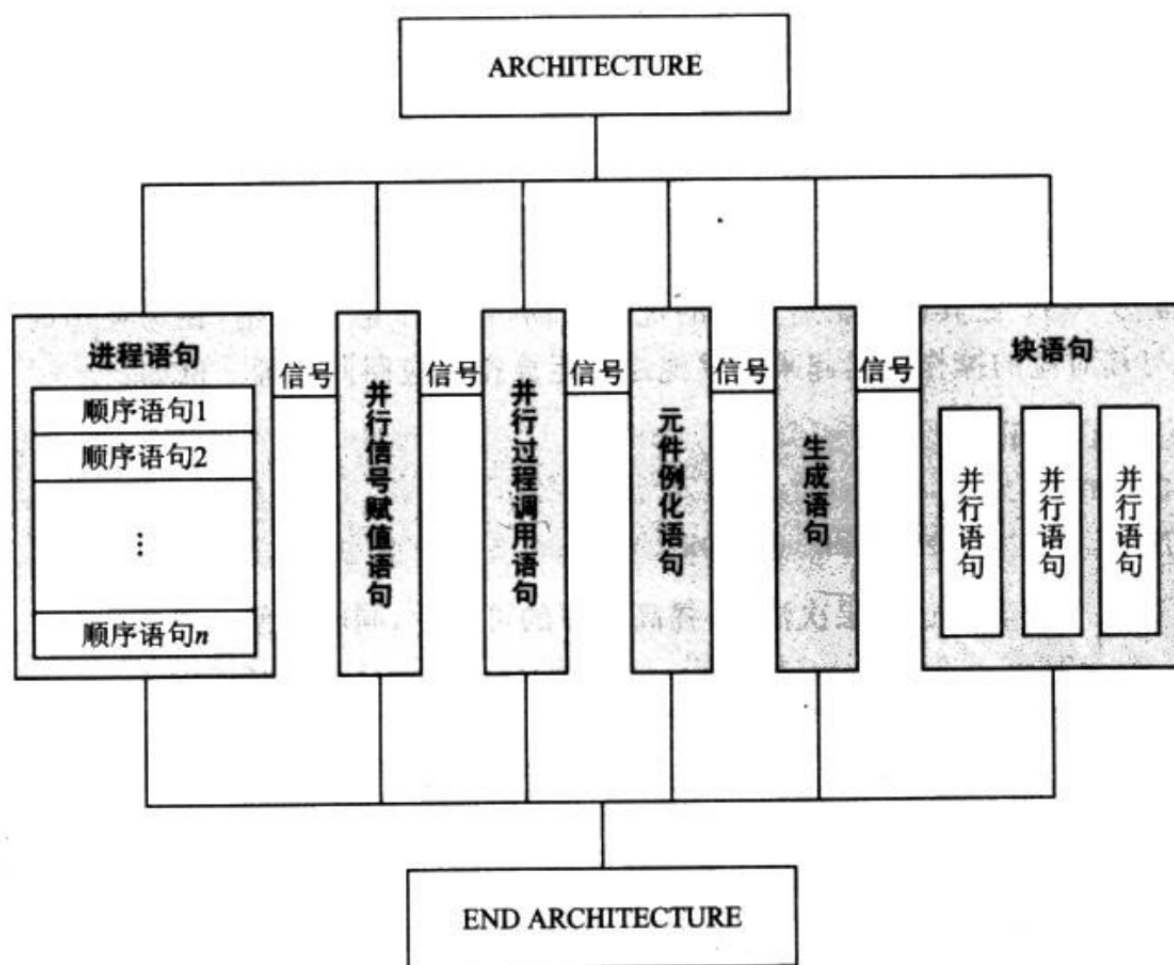
并行语句(concurrent statements)是硬件描述语言区别于一般软件程序语言的最显著的特点之一。所有并行语句在结构体中的执行都是同时进行的,即它们的执行顺序与语句书写的顺序无关。

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 是强类型语言，所以目标信号的数据类型必须与赋值符号“<=”右边表达式的数据类型一致。

#### 选择信号赋值语句

选择信号赋值语句的格式如下：

```
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` 语法格式 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` 语法格式 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。

下面对语法格式 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**

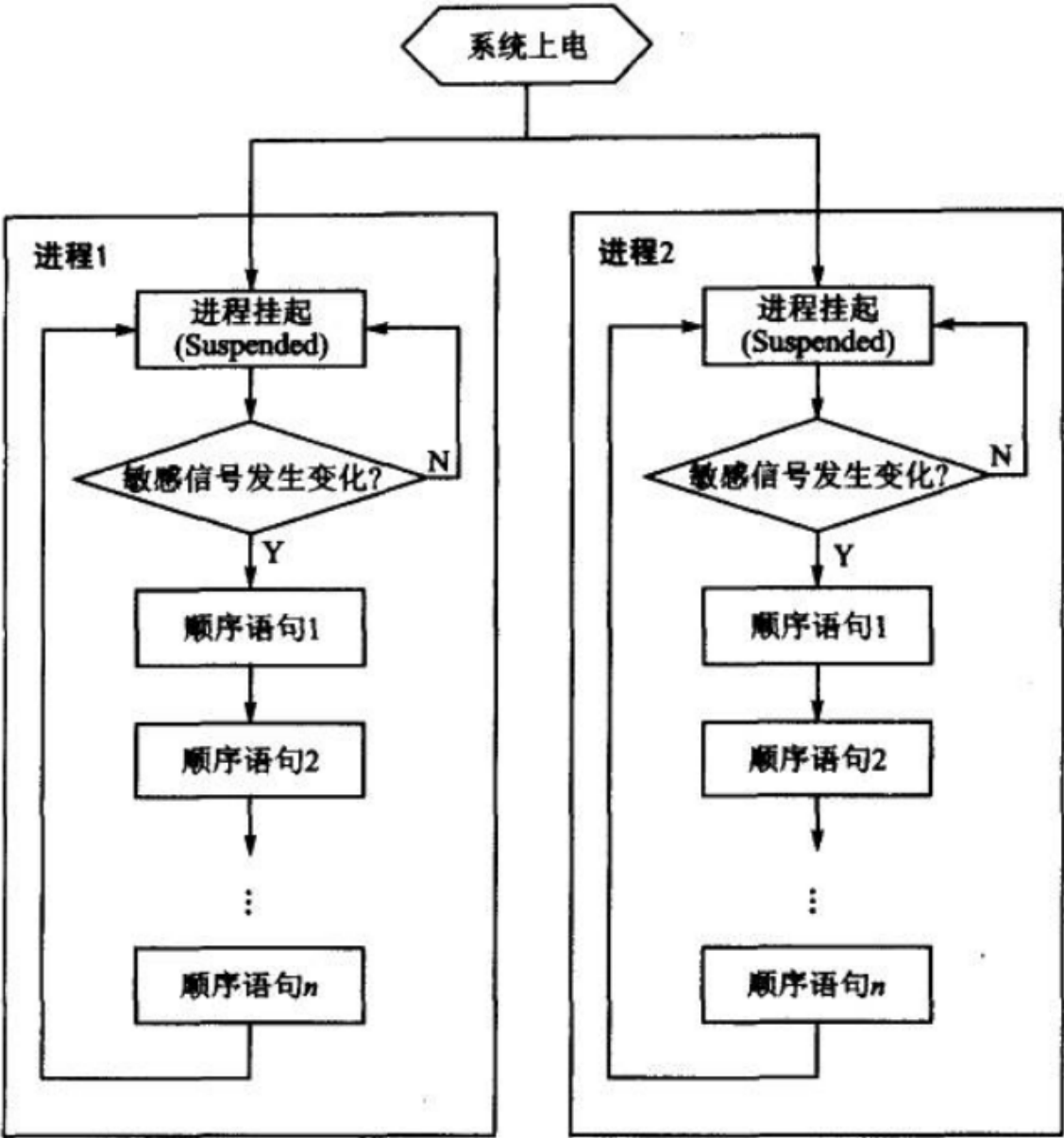
- 定义一些仅在本进程中起作用的局部量, 最常在此处定义的是变量。
- 注意: 信号是全局变量, 不可在此处声明。

- **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变化。

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。

...

VHDL 入门解惑经典实例经验总结 P60

3.1.6 顺序语句

...

## VHDL 入门解惑经典实例经验总结 P67

## 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：

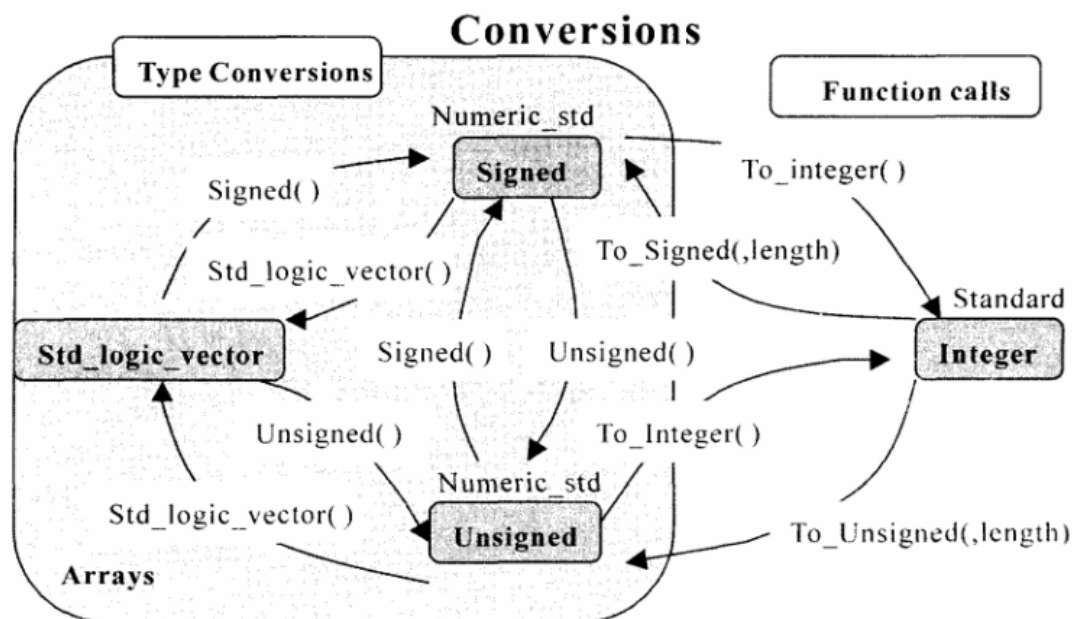
- std\_logic\_vector：简单的四个二进制位；
- unsigned：代表数字 9；
- signed：代表数字 -7（补码表示的）；

## 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

...

#### 4.1 LUPO

这里应该有图片。

#### 4.2 DT5495

这里应该有图片。

#### 4.3 MZTIO

这里应该有图片。





## 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 very 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 “>=” 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“大于或等于”和“小于”符号, 而不使用“大于”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
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 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
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 process;

add_cnt <= ;--add 1   dout_tmp='1'
end_cnt <= add_cnt and (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
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\_”。



## 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。





## 7.1 FIFO

### 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 Nornal 模式。这两种模式的区别在生成 FIFO IP 核额步骤中说明。

#### **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。

#### **FIFO 规则 4：处理报文时，把指示信号与数据一起存入 FIFO**

#### **FIFO 规则 5：读、写时钟不同时，必须用异步 FIFO**



### 8.1 经验总结

#### 8.1.1 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` 代替；`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

#### 8.1.2 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

### 8.1.3 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。

### 8.1.4 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。

## 8.2 临时存放

### 8.2.1 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);
```

### 8.2.2 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 中提供了二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;

// 而如果要读某个单元的值，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:
// 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。
```

## 8.2.3 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
```

(下页继续)

(续上页)

```

        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, generate 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
    endgenerate
endmodule

module fadd(
    input a, b, cin,
    output cout, sum );

```

(下页继续)

(续上页)

```
    assign {cout,sum} = a+b+cin;  
endmodule
```

[https://blog.csdn.net/weixin\\_38197667/article/details/90401400](https://blog.csdn.net/weixin_38197667/article/details/90401400)