台湾友晶公司 DE2-70 实验平台

FPGA/SOPC 入门级实验指导书

适用 Altera 公司 FPGA 芯片 EP2C70F896C6
适用 Altera 公司软件工具 Quartus II　V7.2 / V8.0

2.90 版

教师和助教使用版

南京大学计算机系整理改编

2009 年 9 月 21 日

目　录

第 1 章　DE2-70 开发板驱动安装

1.1 DE2-70 介绍

Altera DE2-70（以下简称 DE2-70）实验/开发两用板是台湾友晶公司为高等院校学生学习 FPGA 编程和学习基于 Nios 软核处理器应用项目编程而开发的实验平台。DE2-70 多媒体开发平台安装了一片拥有 70，000 个逻辑单元的 Cyclone® II 系列 2C70 型 FPGA 芯片，该芯片是 Altera 公司出产的。它的全称是：EP2C70F896C6。

DE2-70 与 Altera DE2-70（以下简称 DE2-70）相兼容，也具有 DE2-70 多媒体平台丰富的多媒体、储存及网络等应用接口的优点。以下是 DE2-70 实验开发板的平面视图。参看图 1-1。

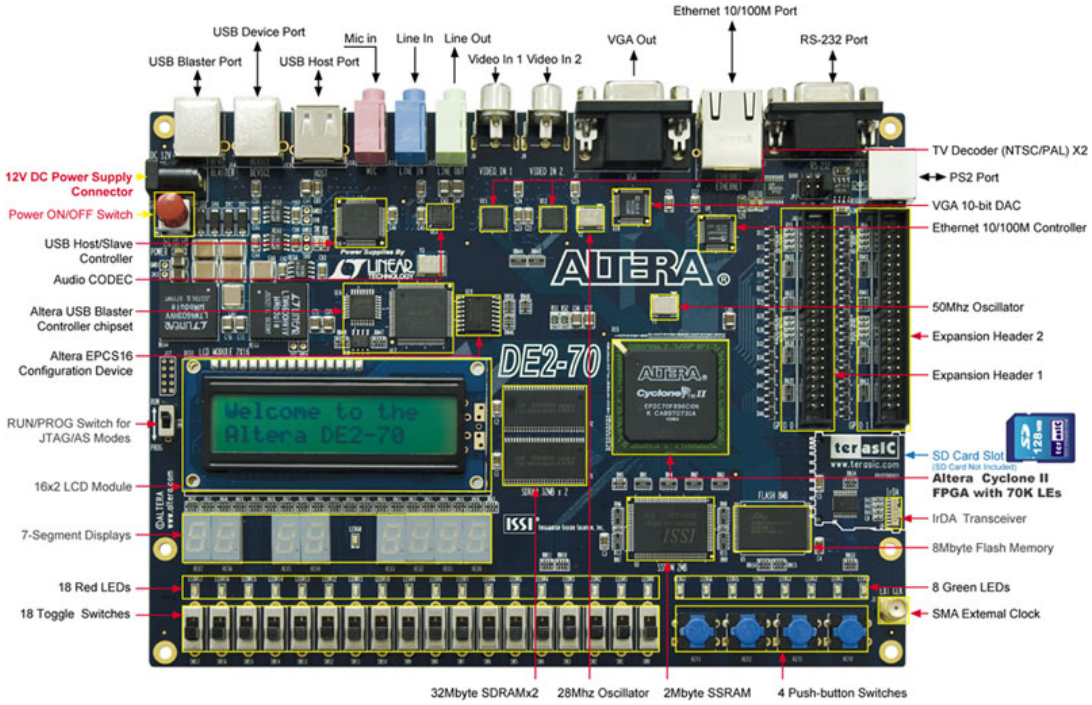


图 1-1　DE2-70 开发板/教学实验板俯视图

Altera 公司网站提供了 DE2-70 开发板/教学实验板的教学资源。其网址如下列出。

http://www.altera.com/education/univ/materials/unv-overview.html

该网站的主要教学内容有五大类，它们是：Tutorials and Lab Exercises, Development and Education Boards, Design Software, University Program Intellectual Property (IP) Cores, Textbooks。

表 1-1　DE2-70 实验平台的演示程序

DE2-70 控制面板工具程序	卡拉 OK 机功能展示
DE2-70 影像工具程序	以太网络封包传递功能展示
电视盒	SD 卡音乐播放程序功能演示
电视盒 Picture in Picture 子母画面	音乐合成器功能演示
USB Paintbrush 展示	音乐录制及播放功能
USB 装置功能展示	

DE2-70 实验平台具有较强的多媒体功能。特别是它具有 VGA 输出功能，可以把视频数据输出到 VGA 视频显示器上。用户可以通过演示程序来体验 DE2-70 实验平台多媒体功能。表 1-1 给出了这些演示程序的列表。

DE2-70 开发板/教学实验板的主要器件技术特点如表 1-2 所示。

表 1-2　DE2-70 开发板/实验板的器件技术特点

器件	描　　述
FPGA	Cyclone II EP2C70F896C6 with EPCS16 16-Mbit serial configuration device
I/O Devices	Built-in USB-Blaster cable for FPGA configuration
	10/100 Ethernet
	RS232
	Video Out (VGA 10-bit DAC)
	Video In (NTSC/PAL/Multi-format)
	USB 2.0 (type A and type B)
	PS/2 mouse or keyboard port
	Line In/Out， Microphone In (24-bit Audio CODEC)
	Expansion headers (76 signal pins)
	Infrared port
Memory	8-MBytes SDRAM， 512K SRAM， 4-MBytes Flash
	SD memory card slot
Displays	16 x 2 LCD display
	Eight 7-segment displays
Switches and LEDs	18 toggle switches
	18 red LEDs
	9 green LEDs
	Four debounced pushbutton switches
Clocks	50 MHz crystal for FPGA clock input
	27 MHz crystal for video applications
	External SMA clock input

1.2 USB-Blaster 的驱动安装

除了加电之外，为了让 DE2-70 开发板/教学实验板正常进行开发工作，还需要在主机上安装 USB-Blaster 电缆的驱动程序以支持 PC 端的开发软件，如 Quartus II、Nios II IDE 等。

安装环境相关说明：

开发软件：Quartus II 7.2/8.0

开发平台：Windows XP SP3

以下是 USB-Blaster 驱动程序在 Windows XP 下的安装步骤。

1.　将 DE2-70 实验平台的 Blaster 接口(开发板上部最左边)接好 USB 连接线，插头插入主机的 USB 接口，Windows XP 发现新硬件后会弹出一个对话框。参看图 1-2。

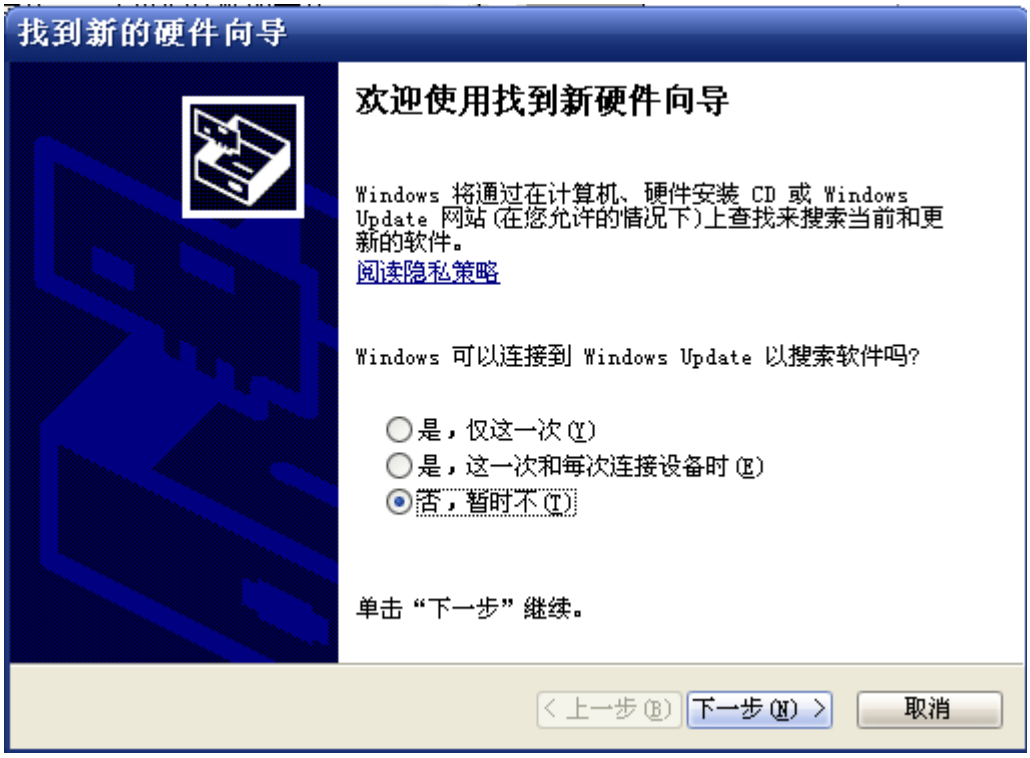


图 1-2　找到新的硬件向导（未识别）

注意：只有计算机没有识别出 USB-Blaster 才会出现图 1-2，如果是卸载后重新安装或者预装载一些驱动信息，则会从图 1-3 开始。

2.　硬件向导

图 1-3　找到新的硬件向导（识别）

3.　选择搜索和安装选项，如图 1-4。

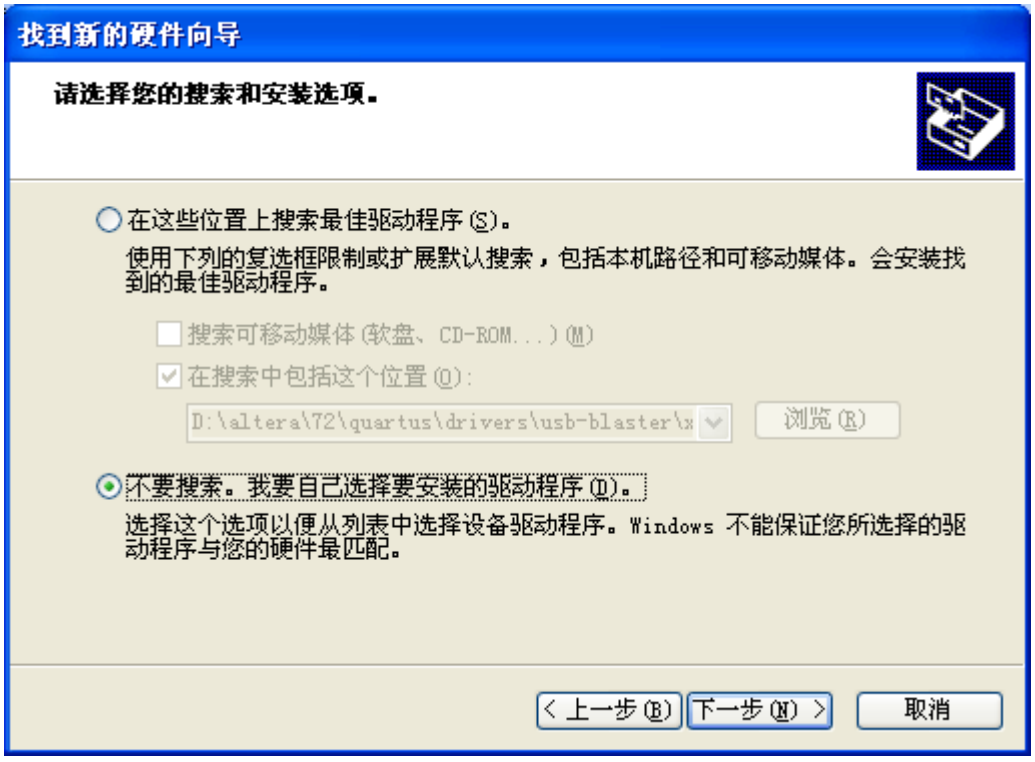


图 1-4　选择搜索和安装选项

4.　选择硬件类型

图 1-5　选择硬件类型

注意：只有从来没有安装过 USB-Blaster 才会出现图 1-5，如果是卸载后重新安装，则会直接转到图 1-6。

5.　选择从磁盘安装

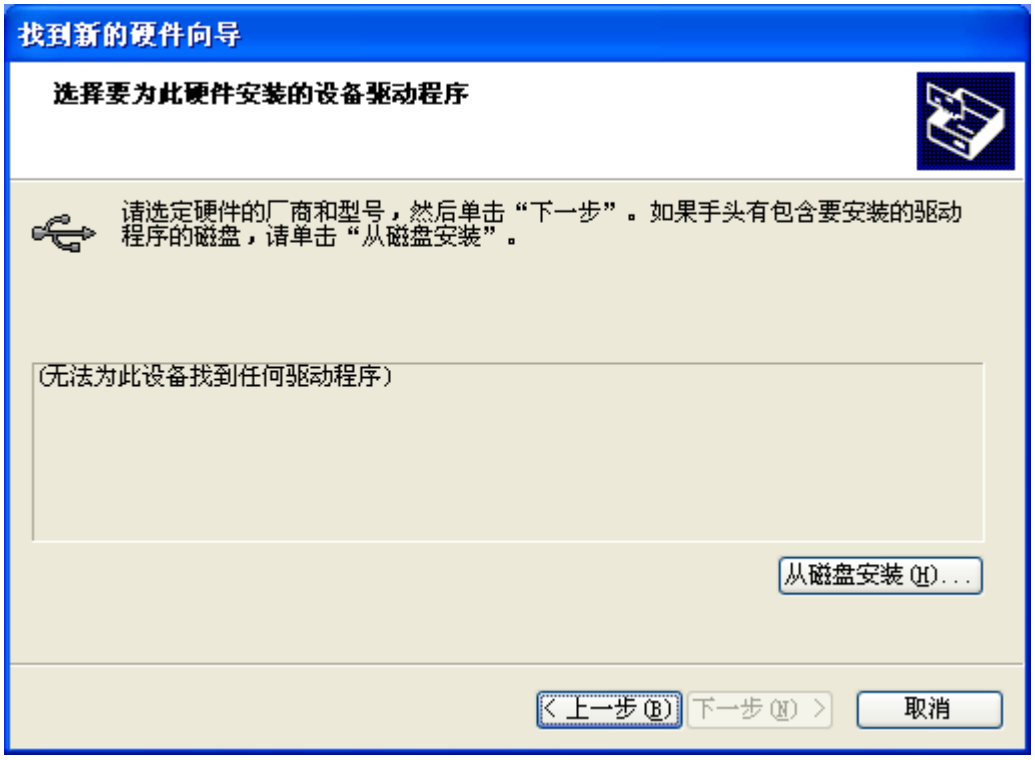


图 1-6A　选择驱动程序（未发现）

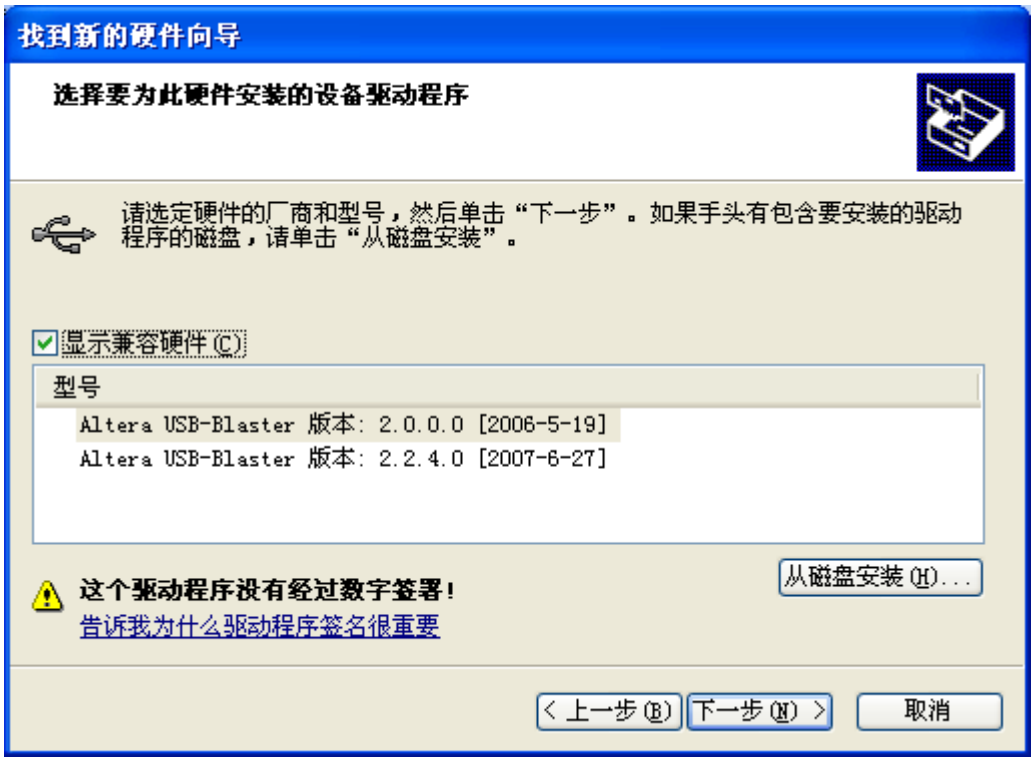


图 1-6B　选择驱动程序（发现）

如果从没装过，则如图 1-6A 所示；如果以前安装过 USB-Blaster 驱动，这时会显示兼容列表，如图 1-6B。选择从磁盘安装。

6.　选择路径及文件，如图 1-7。

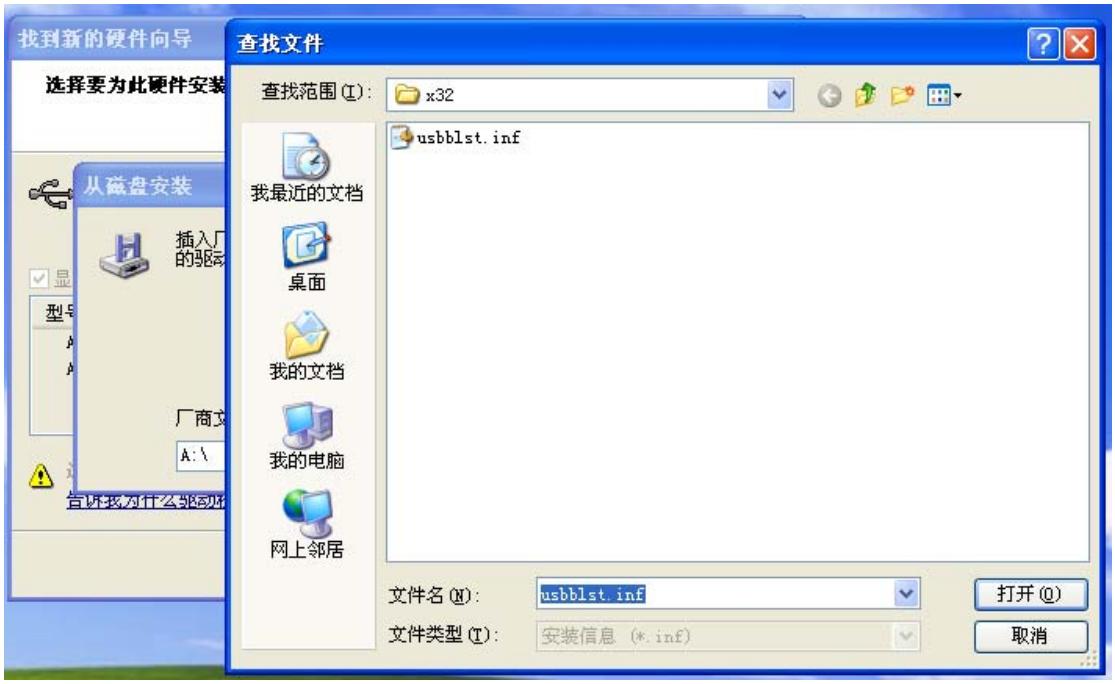


图 1-7　选择驱动程序文件路径

注意：7.2 的驱动文件路径在 D:\altera\72\quartus\drivers\usb-blaster\x32\usbblst.inf

8.0 的驱动文件路径在 D:\altera\80\quartus\drivers\usb-blaster\usbblst.inf

8.0 的 x32 与 x64 共用一个 inf 文件请注意！！

7.　选择驱动，如图 1-8。

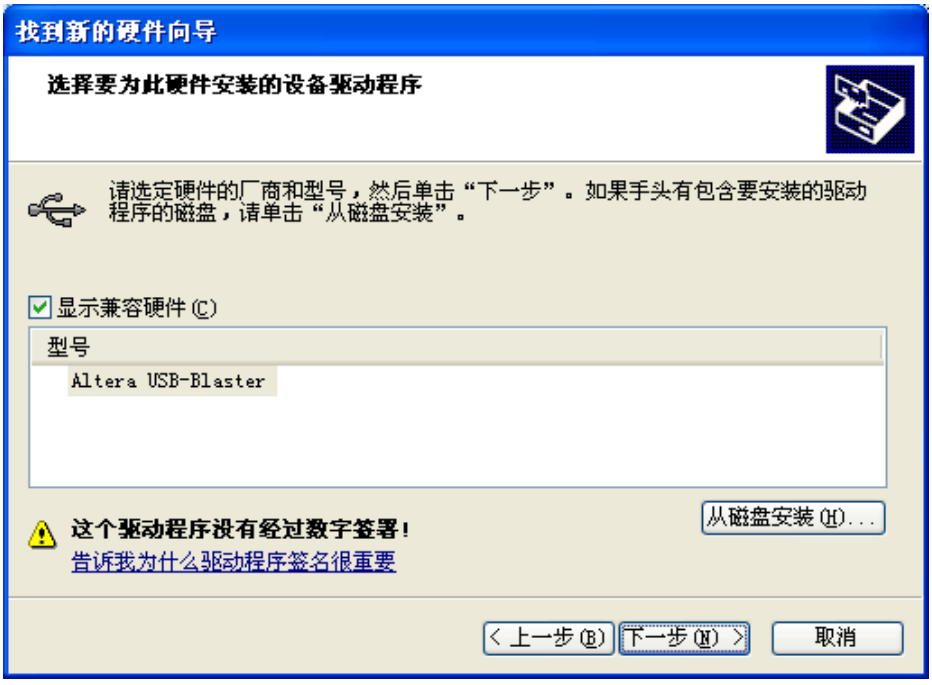


图 1-8　选择驱动程序(手动指定完毕)

8.　单击“下一步”及“仍然继续”，如图 1-9。

图 1-9　驱动程序未签名提示

9.　安装进程，如图 1-10。

图 1-10　驱动程序安装中

10.　完成安装的提示对话框如图 1-11。

图 1-11　驱动程序安装完成

11. 右键单击我的电脑，进入属性页，再进入“硬件”标签页，单击“设备管理器”对话框，单击“通用串行总线控制器”图标，查看安装是否成功。如图 1-12 所示。

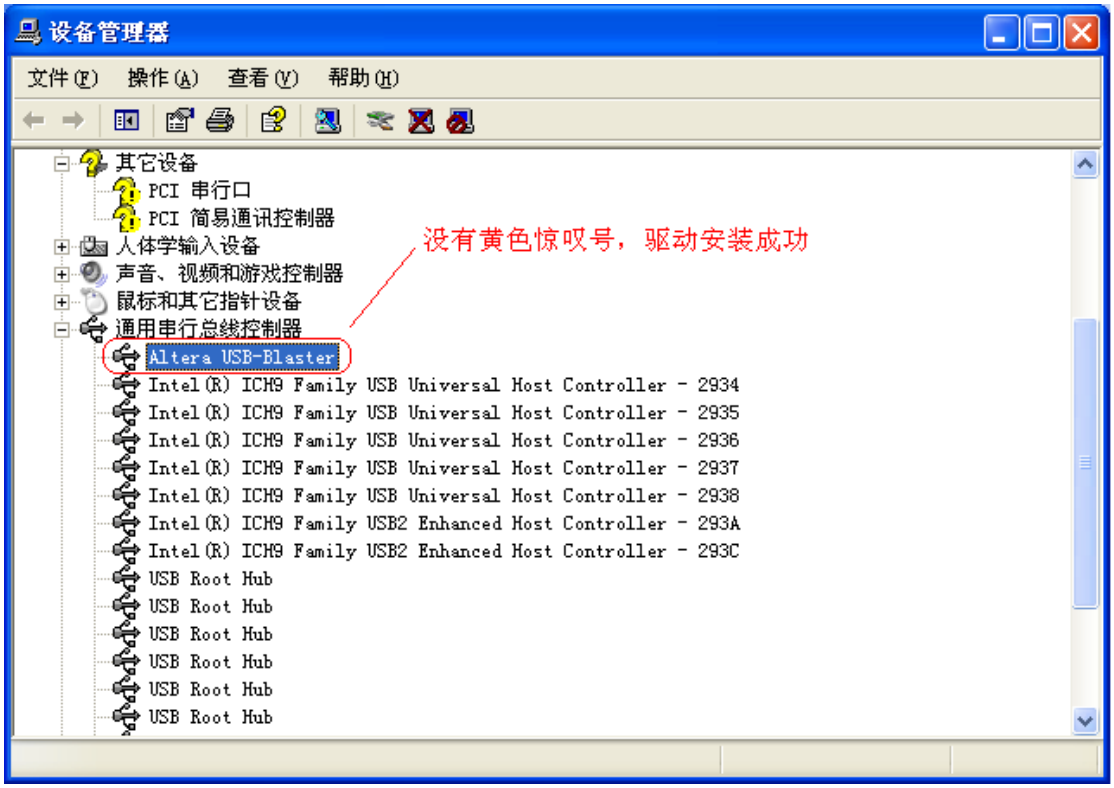


图 1-12　驱动程序安装成功与否检查

以上就是 DE2-70 实验平台的驱动安装过程。

1.3 USB-Blaster 驱动之疑难解答

问 1：USB-Blaster 8.0 的驱动在 Quartus II 7.2 下无法正常工作？
答：是的，如果想同时安装 Quartus II 7.2 与 8.0，必须使用 7.2 的 Altera USB-Blaster 驱动。

问 2：驱动无法在 X64 环境下使用?
答：这是因为 Altera 公司的驱动未加数字签名，而 Windows Vista x64 不允许加载未加数字签名的驱动；如果使用的是 Windows XP x64 环境，可以尝试使用 USB-Blaster 7.2 的驱动。

问 3：更换 Quartus 版本无法安装 USB-Blaster Driver 驱动，提示“名称已用作服务名或服务显示名”?
答：windows 设备管理器中的驱动卸载操作卸不干净，必须手动在注册表中删除服务，之后重新安装。

问 4：更换 Quartus 版本无法安装 USB-Blaster Driver 驱动，提示“找不到文件”?
答：很可能是原有驱动文件被破坏但注册表中信息未同步删除。手工 copy 驱动文件到如下位置，重启再次安装驱动。
windows\system32\drivers\ftdibus.sys
windows\system32\ftbussui.dll
windows\system32\ftd2xx.dll
windows\system32\drivers\usbblstr.sys (8.0 专有)
windows\system32\usbblstr32.dll(8.0 专有)
windows\system32\usbblstrui.dll(8.0 专有)

1.4 DE2-70 引脚分配的一般性指导

有三种为 DE2-70 实验程序分配器件引脚的方法。它们是**逐个手工指定**（参看第 2 章的操作步骤 14），**导入 DE2-70 的.csv 文件**（参看第 4 章的操作步骤 8）和**修改工程的.qsf 文件**（参看第 4 章的操作步骤 11）。作为初学者，这三种方式都要循序渐进地学习和练习，在学期结束时做到熟练掌握。

无论采用哪种方法定义 DE2-70 上的引脚信号，实验者可以参照 Terasic Technologies 公司出版的 DE2-70 Development and Education Board User Manual 英文文档中的引脚描述，找到引脚定义的根据。

1.5 DE2-70 实验板基本输入输出引脚信号

1. **LED** 灯：有两组，**LEDR[17:0]**和 **LEDG[7:0]**

   这两组 LED 灯用于简单输出。一般用于二进制结果输出，如果是较大的十进制数，采用 HEX 或者 LCD 输出较好。**oLEDR** 与 **oLEDG** 除了数量与颜色不同外，用法基本一致。
2. **HEX** 发光管 **HEX[7:0]**，用于数值的输出。

   一般用于十进制或十六进制结果的输出，有时也可用来显示英文字符。
3. 开关 **SW[17:0]**：用于简单的输入。

   拥有输入并保持同一电平信号的优势，一般用于数据信号或者功能控制信号。相对于按钮来说，可以用开关手工模拟低速的方波信号。
4. 按钮 **KEY[3:0]**：用于简单的输入。

平时状态是高电平，按下时低电平，与开关各有优劣，一般用于复位信号与单步调试时的时钟信号。

5. 外部时钟 **EXT_CLOCK**（PIN_R29）：外部时钟输入。

位于开发板的右下角，直径为 8mm 左右，一个带螺纹的黄铜接口。当实验工程中含有外部时钟输入信号发生器的时候，可以用引线外接到 **DE2-70** 上进行时钟输入。使用场合较少。

第 2 章　实验一　3-8 译码器实验

● 实验说明

Quartus II 设计工具支持多种设计输入模型，本次实验使用 Verilog 硬件描述语言在 DE2-70 开发平台上设计一个基本组合逻辑电路——3-8 译码器。通过这个实验，读者可以了解使用 Quartus 工具设计硬件的基本流程。

● 实验步骤

2.1 建立 Quartus 工程

1. 打开 Quartus II 工作环境，如图 2-1 所示。

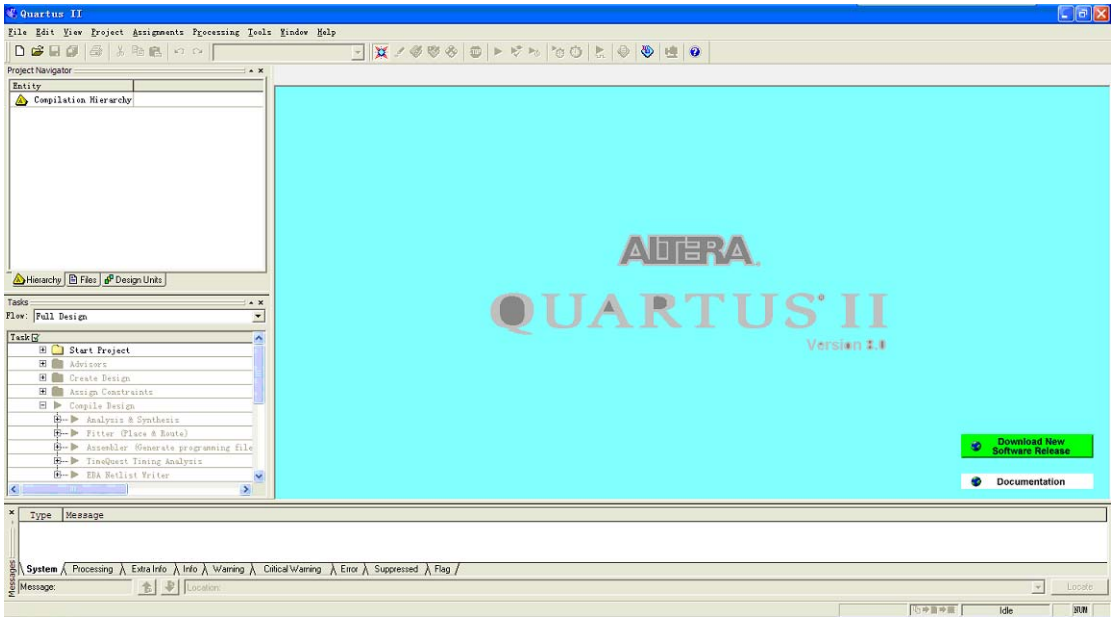


图 2-1　Quartus II 工作环境界面

2. 点击菜单项 File->New Project Wizard 帮助新建工程。参看图 2-2。

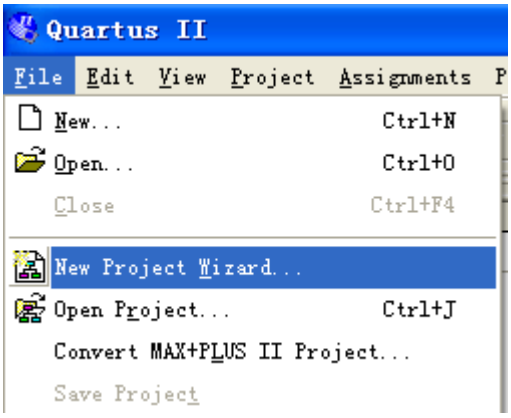


图 2-2　选择 New Project Wizard

打开 Wizard 之后，界面如图 2-3 所示。点击 Next，

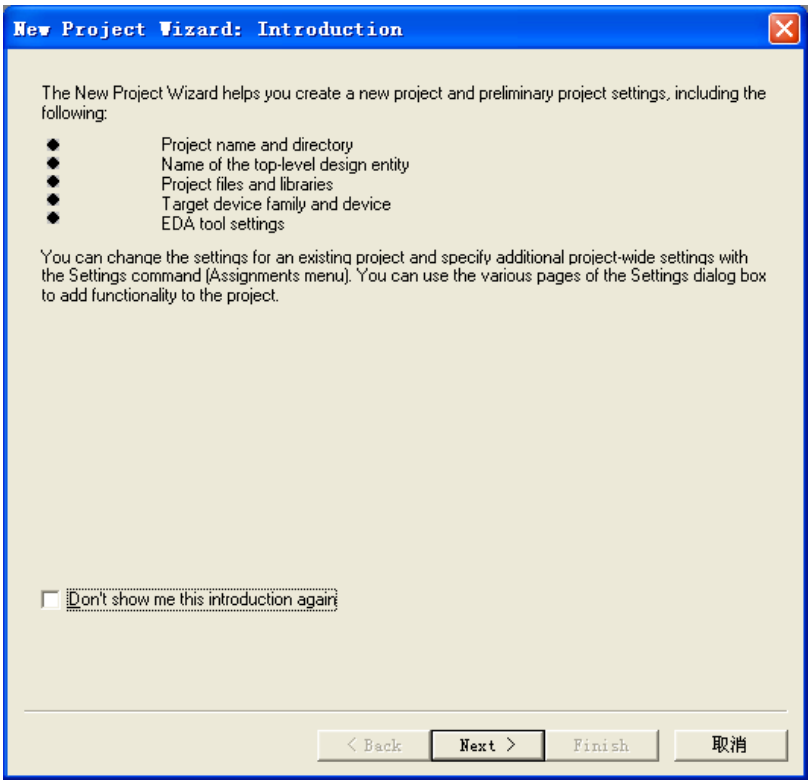


图 2-3　New Project Wizard 界面

3.　输入工程工作路径、工程文件名以及顶层实体名。

注意：这里输入的顶层实体名必须与之后设计文件的顶层实体名相同，默认的顶层实体名与工程文件名相同，本次实验采用这种命名方法。用户也可以根据需要输入不同的顶层实体名。输入结束后，如图 2-4 所示。点击 Next。

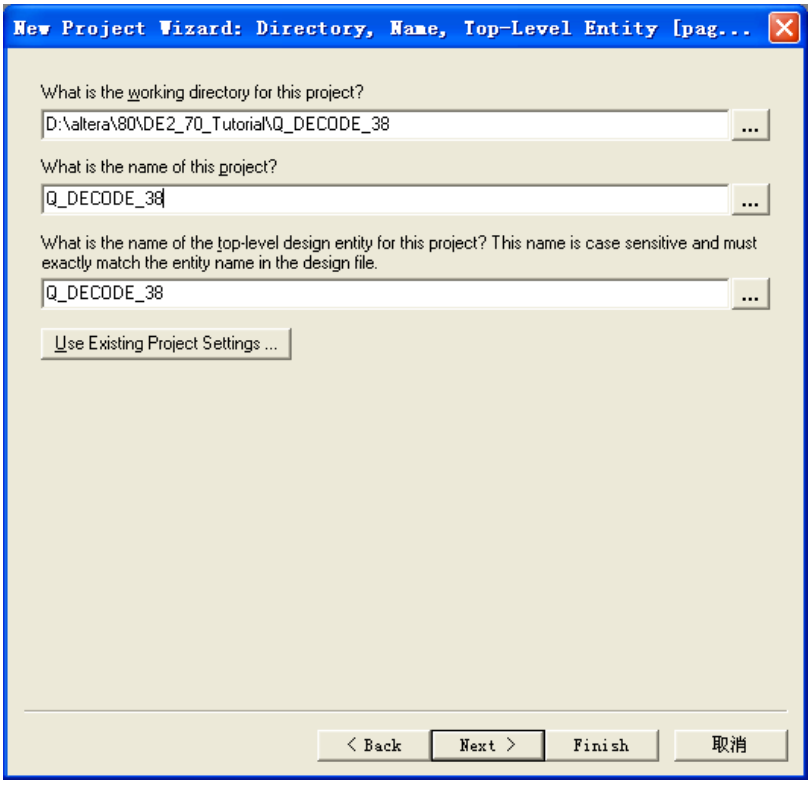


图 2-4　输入设计工程信息

4.　添加设计文件。界面如图 2-5 所示。如果用户之前已经有设计文件（比如.v 文件）。那么再次添加相应文件，如果没有完成的设计文件，点击 Next 之后添加并且编辑设计文件。

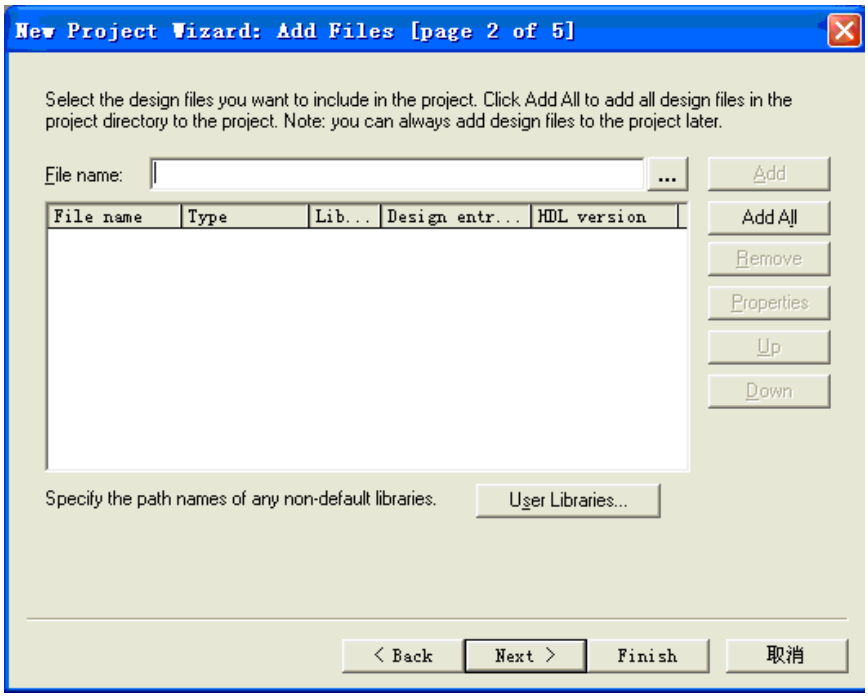


图 2-5　添加设计文件

5.　选择设计所用器件。由于本次实验使用 Altera 公司提供的 DE2-70 开发板，用户必须选择与 DE2-70 开发板相对应的 FPGA 器件型号。

在 Family 菜单中选择 Cyclone II，Package 选 FBGA，Pin Count 选 896，Speed grade 选 6，确认 Available devices 中选中 EP2C70F896C6，如图 2-6 所示。

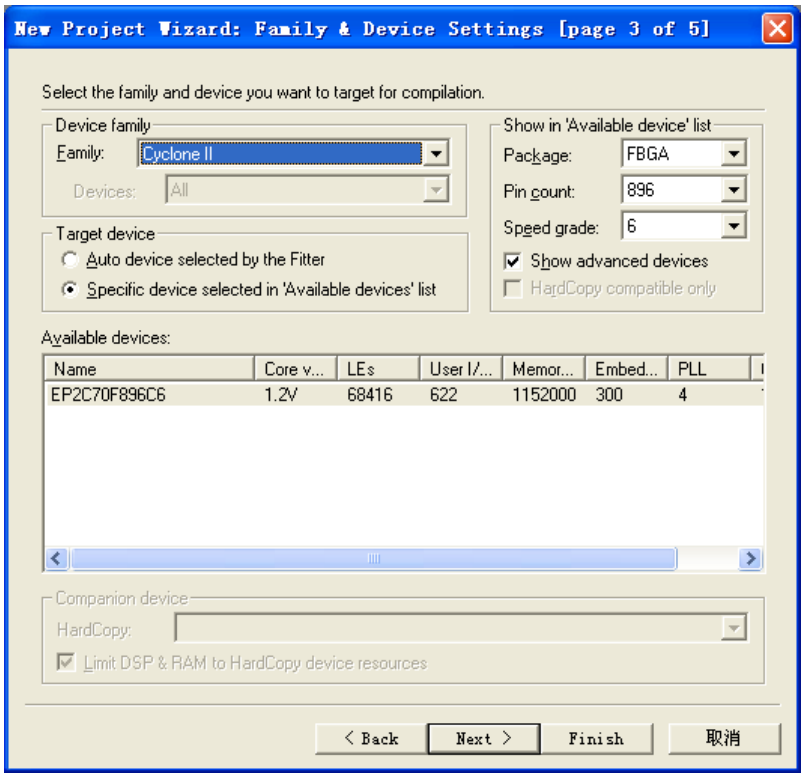


图 2-6　选择相应器件

6.　设置 EDA 工具。设计中可能会用到的 EDA 工具有综合工具、仿真工具以及时序分析工具。本次实验中不使用这些工具，因此点击 Next 直接跳过设置，如图 2-7 所示。

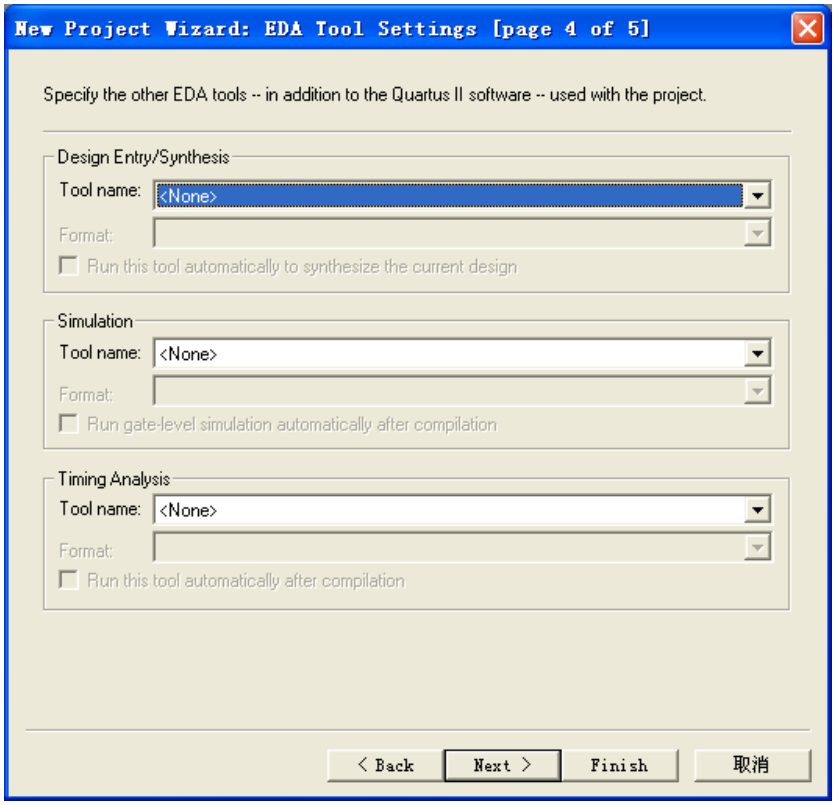


图 2-7　设置 EDA 工具

7.　查看新建工程总结。在基本设计完成后，Quartus II 会自动生成一个总结让用户核对之前的设计，如图 2-8 所示，确认后点击 Finish 完成新建。

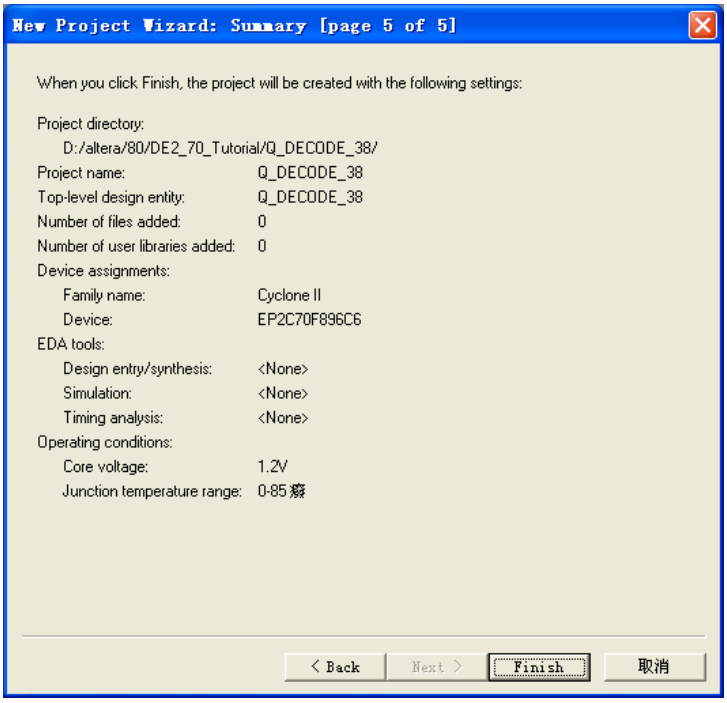


图 2-8　新建工程总结

在完成新建后，Quartus II 界面中 Project Navigator 的 Hierarchy 标签栏中会出现用户正

在设计的工程名以及所选用的器件型号，如图 2-9 所示。

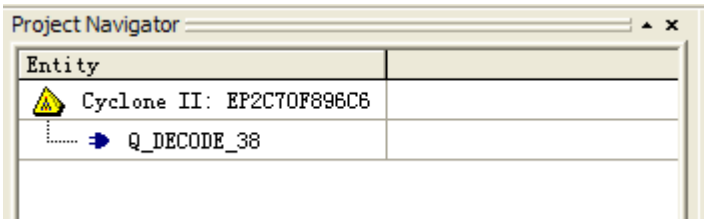


图 2-9　观察正在设计的工程

8.　培养良好的文件布局。Quartus II 默认把所有编译结果放在工程根目录，为了让 Quartus II 像 Visual Studio 等 IDE 一样把编译结果放在一个单独的目录中，需要指定编译结果输出路径。

点击菜单项 Assignments->Device，选中 Compilation Process Settings 选项卡，勾上右边的 Save Project output files in specified directory，输入路径(一般为 debug 或者 release)，如图 2-10 所示。

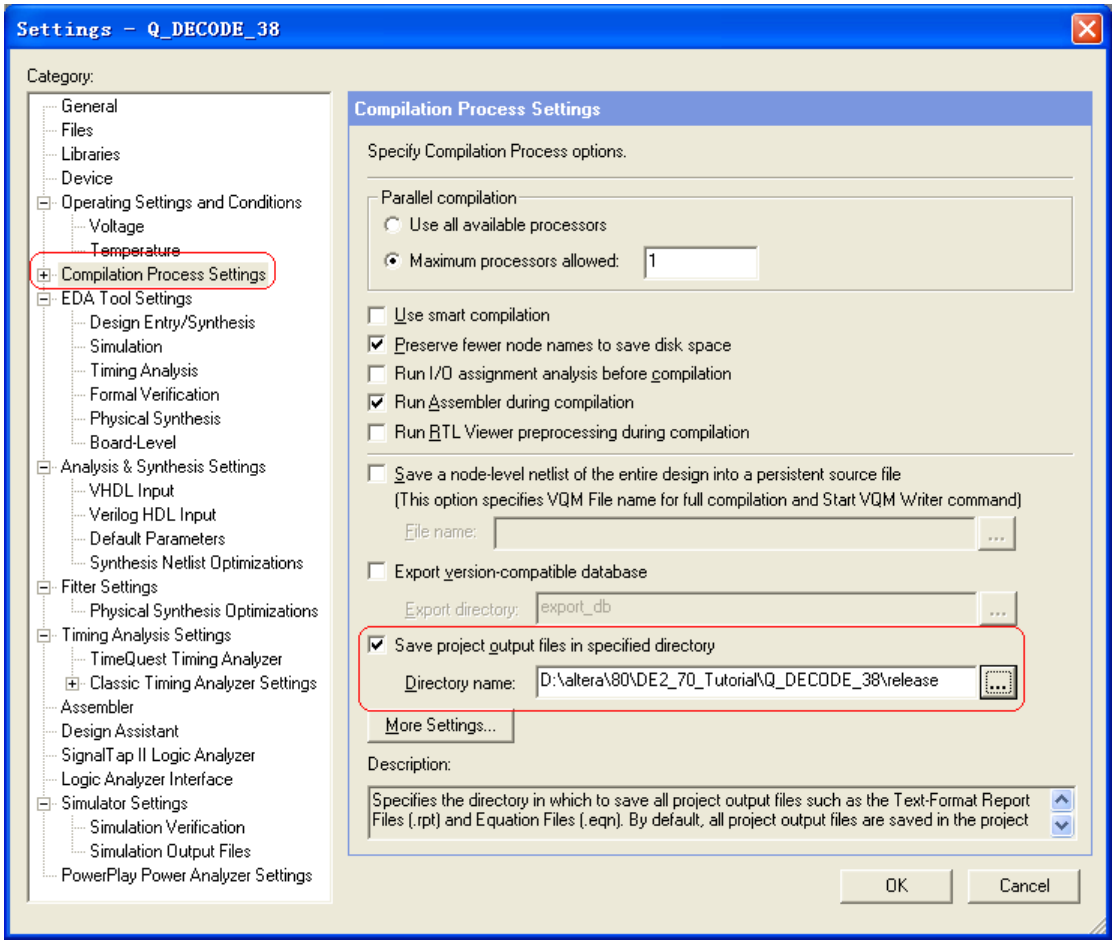


图 2-10　指定单独的编译结果文件目录

2.2 使用 Verilog HDL 完成硬件设计

9.　添加所需设计文件。本次实验通过 Verilog HDL 来描述所设计的硬件，因此要添加 Verilog 设计文件到工程文件中去。

点击菜单项 File->New、点击图标或者使用快捷键 Ctrl+N 新建一个设计文件，选

择 Verilog HDL File，如图 2-11 所示，点击 OK。

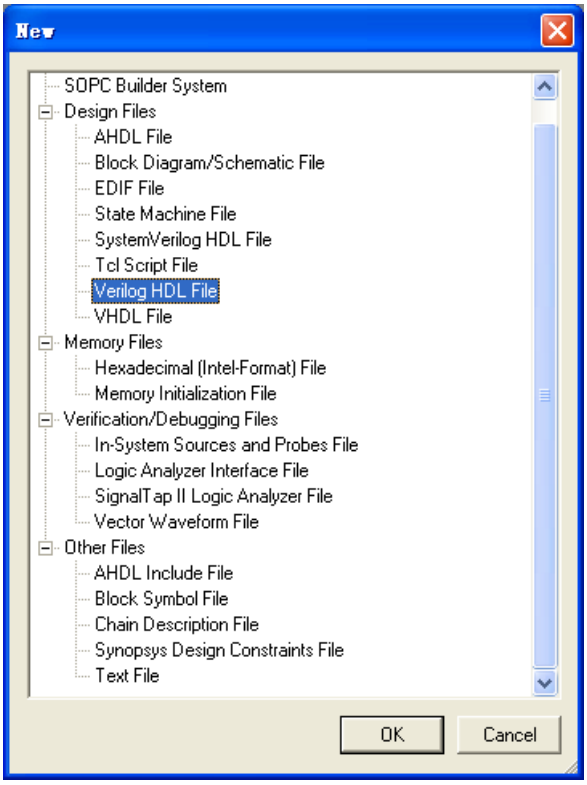


图 2-11　选择设计文件类型

10. 输入硬件描述。在 Quartus II 环境提供的文本编辑器中输入用户设计的硬件描述语言，在本次实验设计的是一个 3-8 译码器，输入代码如图 2-12 所示。

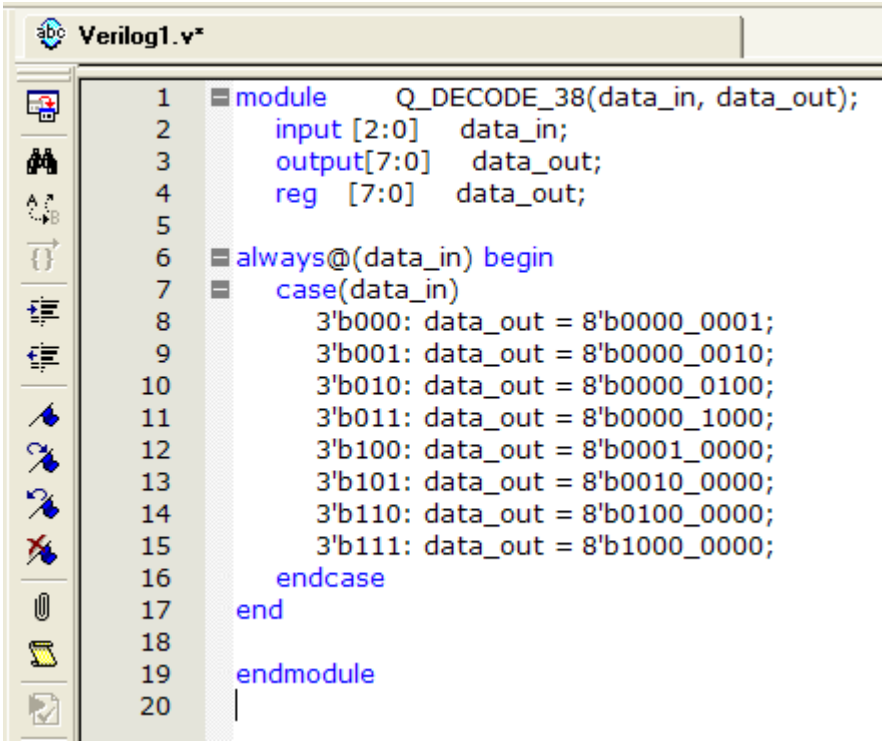


图 2-12　输入设计代码

11. 保存设计。点击菜单项 File->Save、点击图标或者使用快捷键 Ctrl+S 保存设计，

如图 2-13 所示。给设计文件命名 Q_DECODE_38，与 3-8 译码器的模块名相同，点击保存。

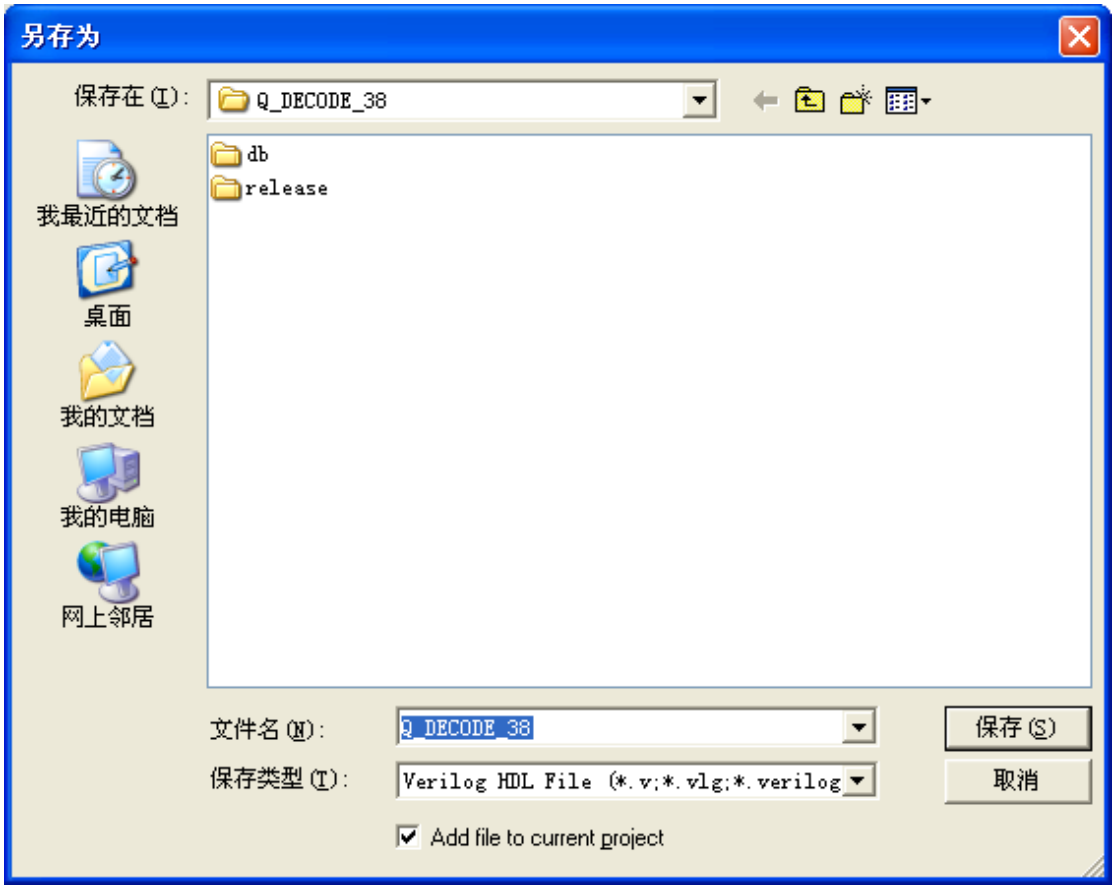


图 2-13　保存设计文件

12. 分析与综合。点击菜单项 Processing->start->Start Analysis & Synthesis、点击图标 或者使用快捷键 Ctrl+K 执行分析与综合。参看图 2-14。

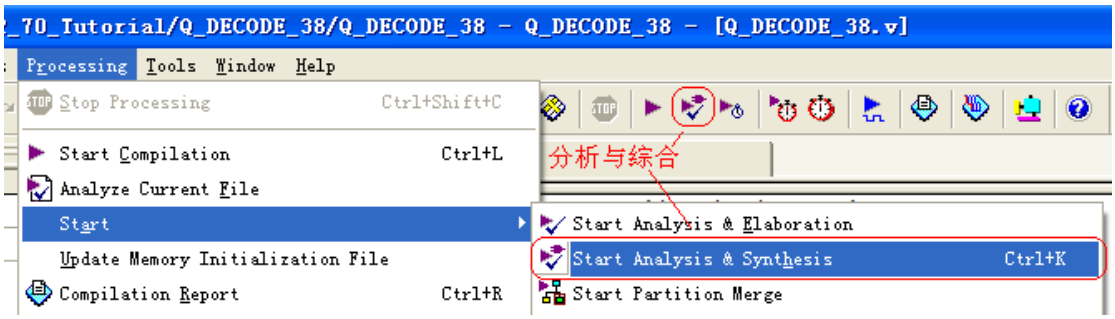


图 2-14　执行 start Analysis & Synthesis（开始分析与综合）

注意：Start Analysis & Synthesis(分析与综合) = Start Analysis & Elaboration(分析与解析)+ Mapping(映射)。

如果仅仅需要检查语法，那么执行 Analysis & Elaboration 即可，但是这一步生成的数据库并不对应 FPGA 器件的物理结构，生成的网表中结点的名称也不与 FPGA 器件的 Cell 名称对应。而且这一操作没有快捷键支持，更多的情况下直接执行 Start Analysis & Synthesis。

Start Analysis & Synthesis 后，生成的数据库已经对应了 FPGA 器件的物理结构，“映射”后的数据库包含了 FPGA 底层 Cell 的位置信息和 Cell 本身的时序信息。

分析与综合完成后，状态窗口如图 2-15 所示。

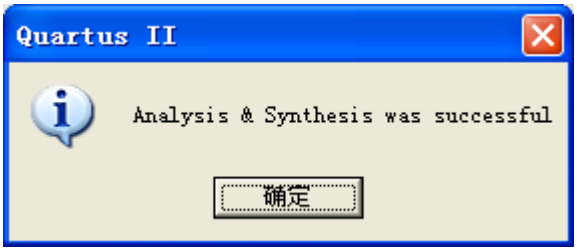


图 2-15　执行 start Analysis & Synthesis 后

13. 全编译文件。点击菜单项 Processing->start compilation、点击图标 或使用 CTRL+L 执行全编译，如图 2-16。

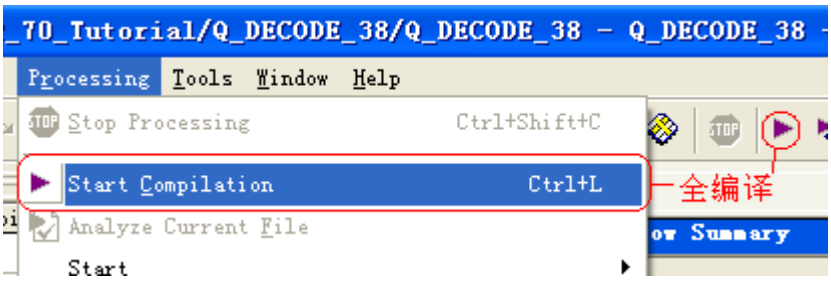


图 2-16　执行 start compilation

编译结果如图 2-17 所示。

图 2-17　全编译结果显示

注意：进行这次全编译仅仅是为了利用 Assignments->Pins 来手工分配引脚，分配完成后需要再次全编译。如使用 qsf 文件分配引脚则只需全编译一次即可。

14. 为 DE2-70 运行 3-8 译码器配置引脚。配置引脚有 3 种方法，分别是手工指定、使用 csv 文件导入与使用 qsf 文件，这里使用第一种，剩下两种后面的实验中会提到。点击菜单项 Assignments->Pins，如图 2-18。

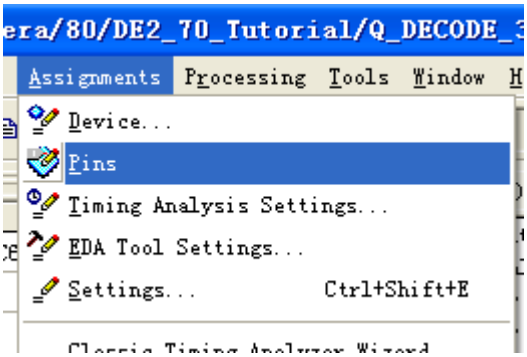


图 2-18　配置 DE2-70 的引脚操作

15. Pins 菜单项执行之后，会出现一个引脚配置窗口。参看图 2-19。我们只用该窗口的下部的列表进行具体信号的引脚指定，参看图 2-20。

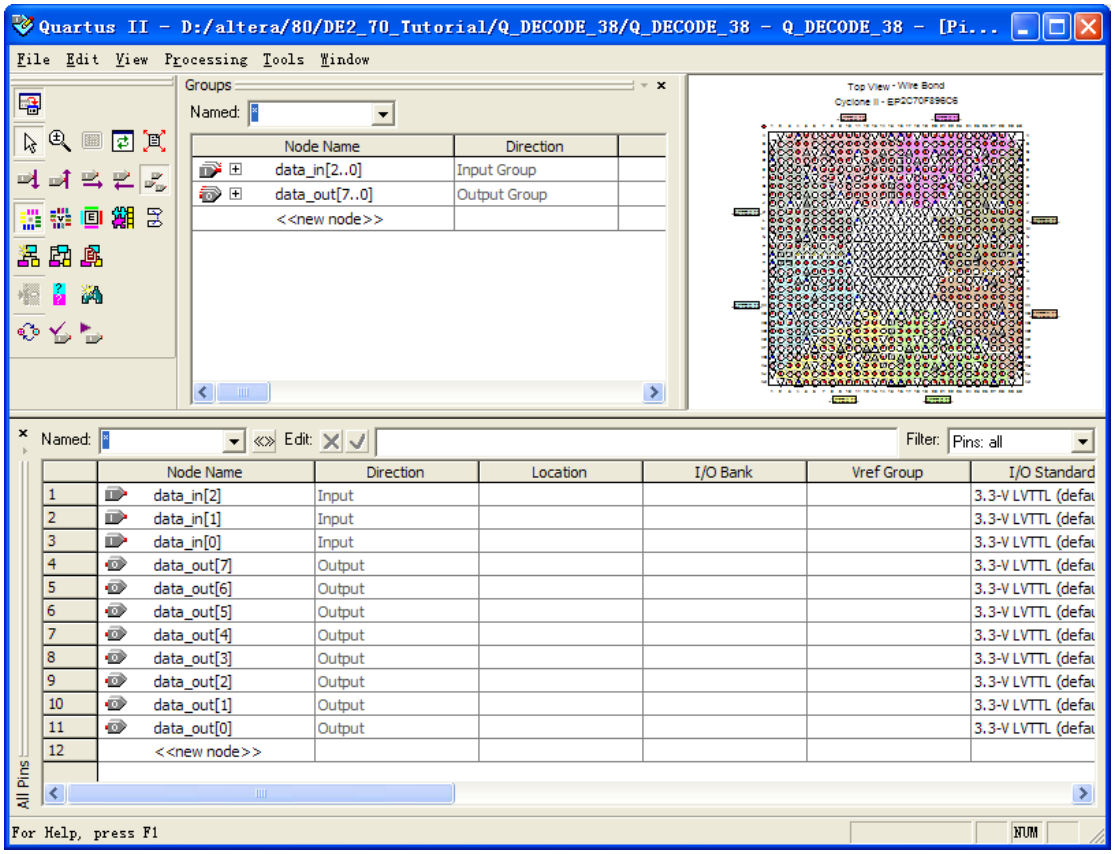


图 2-19　引脚配置窗口

为了将逻辑分配到 FPGA 外围引脚上，必须根据所用的 FPGA 型号配置输出引脚。根据所提供的 DE2-70 用户指导手册，将 3-8 译码器的输入与输出分别配置到 DE2-70 开发板的 3 个选择开关（SW2，SW1，SW0）以及 8 个 LED（LEDR7-LEDR0）上。

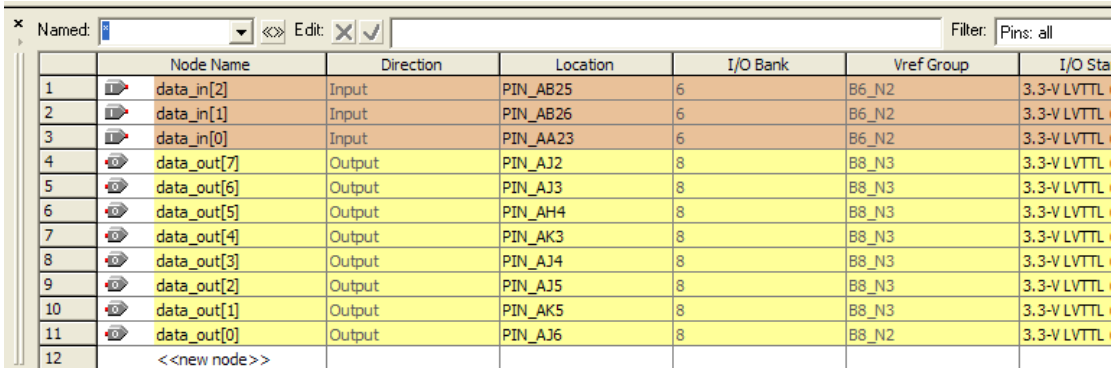


图 2-20　输入引脚编号

16. 全编译文件。完成分配引脚后，点击菜单项 Processing->start compilation、点击图标  或使用 CTRL+L 执行全编译，生成 sof 目标文件。结果如图 2-21 所示，可见引脚未指定的 warning 已经去掉。

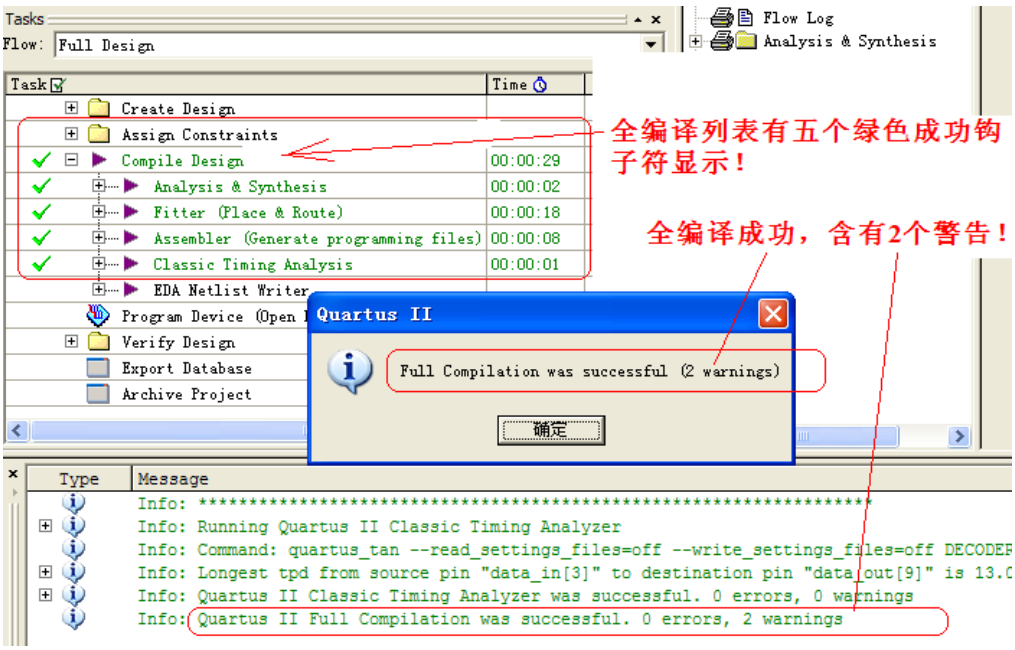


图 2-21　编译结果窗口

17. 将设计下载在 FPGA 中。点击菜单项 Tools->Programmer 或者点击图标  打开程序下载环境，如图 2-22 所示。

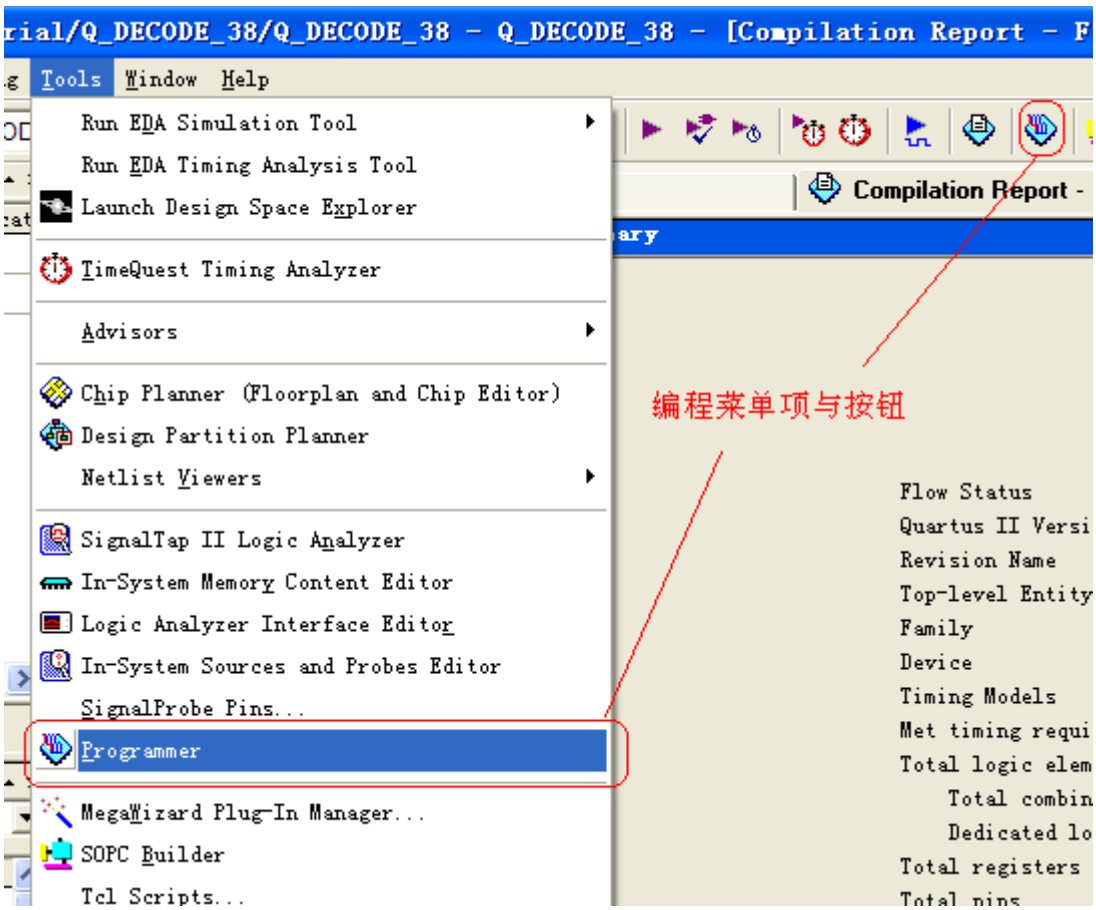


图 2-22　编程菜单项和编程按钮

18. 之后的输出画面如图 2-23 所示。

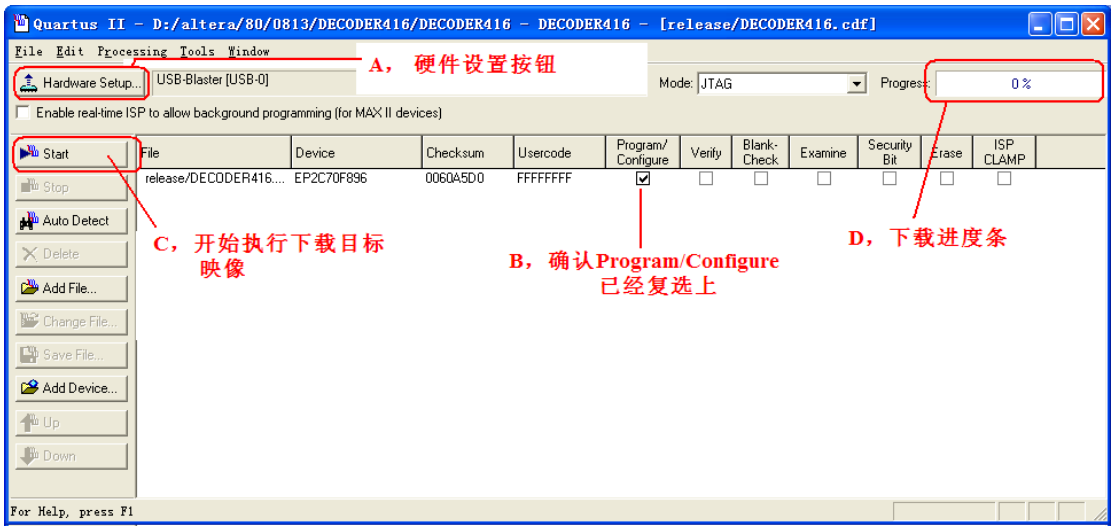


图 2-23　程序下载界面

点击 Hardware Setup 按钮下载时使用的硬件。如图 2-24 所示。

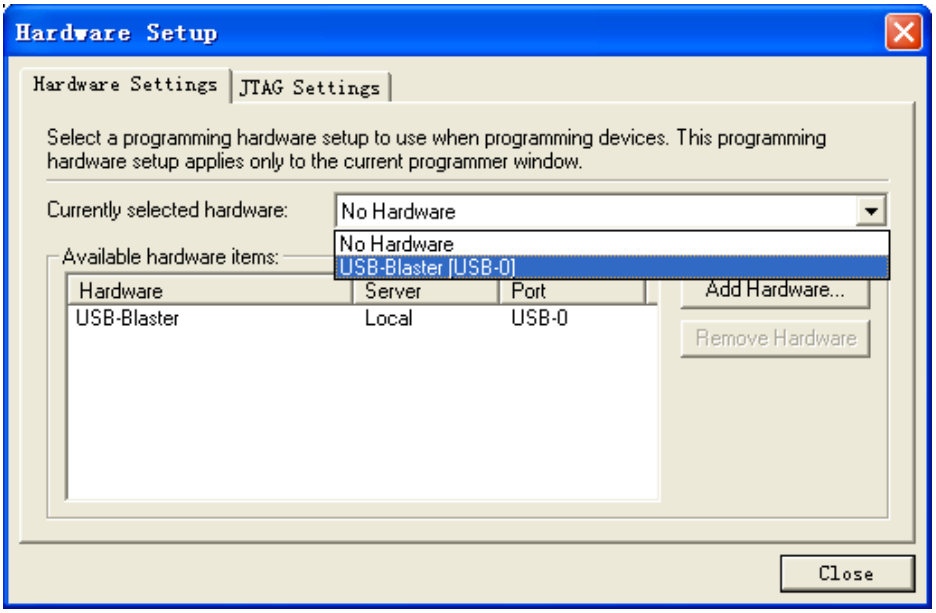


图 2-24　Hardware Setup 界面

点击 Close 确认设置。

19. 下载程序。在 Programmer 界面中，将 Q_DECODE_38.sof 文件列表中 Program/Configure 属性勾上，如图 2-23 和图 2-25 所示。

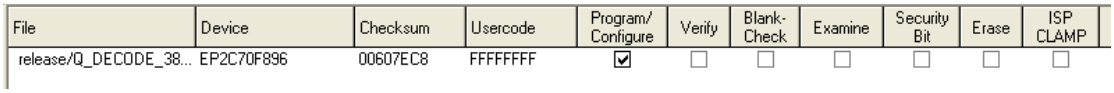


图 2-25　选择 Program/Configure 属性

再在图 2-23 点击 Start 按钮，开始下载程序。

完成后，下载程序显示为 100%，如图 2-26。

图 2-26 下载进度显示

20. 最终调试，在 DE2-70 实验板上，扳动 SW2，SW1 和 SW0 开关，可以看到译码之后的 LEDR7-LEDR0 红色 LED 发光输出。

2.3 替换练习

21. 改写本实验代码，新建立一个工程文件，实现一个 4-16 译码器（4 输入 16 输出译码器）。

参考解答：

```
module DECODER416(data_in, data_out);
    input[3:0] data_in;
    output[15:0] data_out;
    reg [15:0] data_out;
always@(data_in)begin
    case(data_in)
    4'b0000:data_out=16'h0001;   /* light LEDR[0] */
    4'b0001:data_out=16'h0002;   /* light LEDR[1] */
    4'b0010:data_out=16'h0004;   /* light LEDR[2] */
    4'b0011:data_out=16'h0008;   /* light LEDR[3] */
    4'b0100:data_out=16'h0010;   /* light LEDR[4] */
    4'b0101:data_out=16'h0020;   /* light LEDR[5] */
    4'b0110:data_out=16'h0040;   /* light LEDR[6] */
    4'b0111:data_out=16'h0080;   /* light LEDR[7] */
    4'b1000:data_out=16'h0100;   /* light LEDR[8] */
    4'b1001:data_out=16'h0200;   /* light LEDR[9] */
    4'b1010:data_out=16'h0400;   /* light LEDR[10] */
    4'b1011:data_out=16'h0800;   /* light LEDR[11] */
    4'b1100:data_out=16'h1000;   /* light LEDR[12] */
    4'b1101:data_out=16'h2000;   /* light LEDR[13] */
    4'b1110:data_out=16'h4000;   /* light LEDR[14] */
    4'b1111:data_out=16'h8000;   /* light LEDR[15] */
    endcase
end
endmodule
```

22. 建立一个名为 Q_74LS138 的工程文件，用输入电原理图的方法（参看图 2-27）完成 74LS138 译码器（商品 3-8 译码器芯片）的实验。

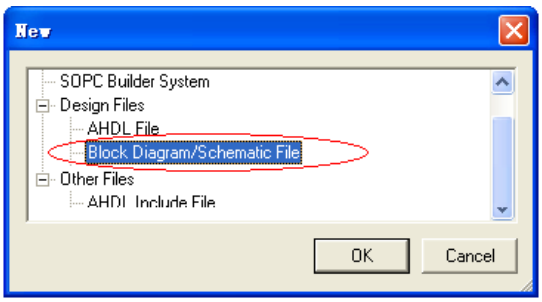


图 2-27　新建一个方框图/电原理图文件的对话框

◆ 本实验指导结束

第 3 章　实验二　十进制计数器实验

● 实验说明

该实验将使用 Verilog 硬件描述语言在 DE2-70 开发平台上设计一个基本时序逻辑电路——1 位十进制计数器。通过这个实验，读者可以了解使用 Quartus 工具设计硬件的基本流程以及使用 Quartus II 内置的工具进行仿真的基本方法和使用 SignalTap II 实际观察电路运行输出情况。SignalTap II 是 Quartus 工具的一个组件，是一个片上的逻辑分析仪，可以通过 JTAG 电缆将电路运行的实际输出传回 Quartus 进行观察，从而省去了外界逻辑分析仪时的很多麻烦。

● 实验步骤

3.1 建立工程并完成硬件描述设计

1. 打开 Quartus II 工作环境，如图 3-1 所示。

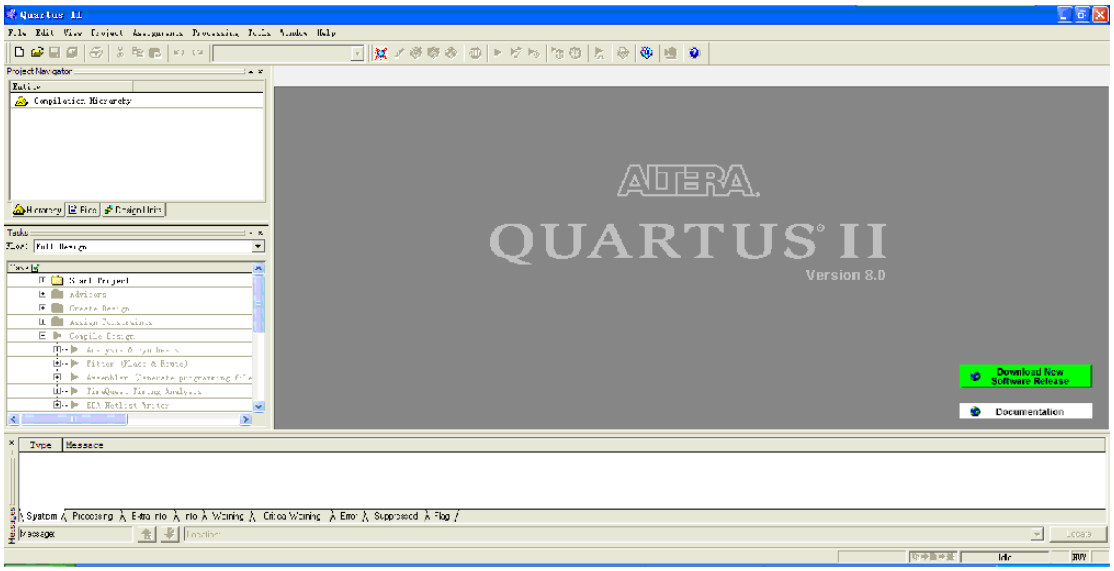


图 3-1　Quartus II 工作环境界面

2. 点击菜单项 File->New Project Wizard 帮助新建工程。参看图 3-2。

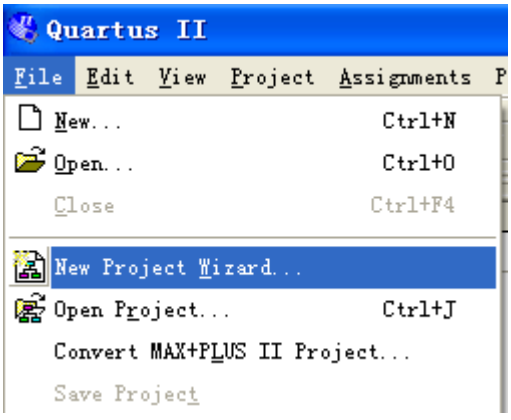


图 3-2　选择 New Project Wizard

打开 Wizard 之后，界面如图 3-3 所示。点击 Next，如图 3-3。

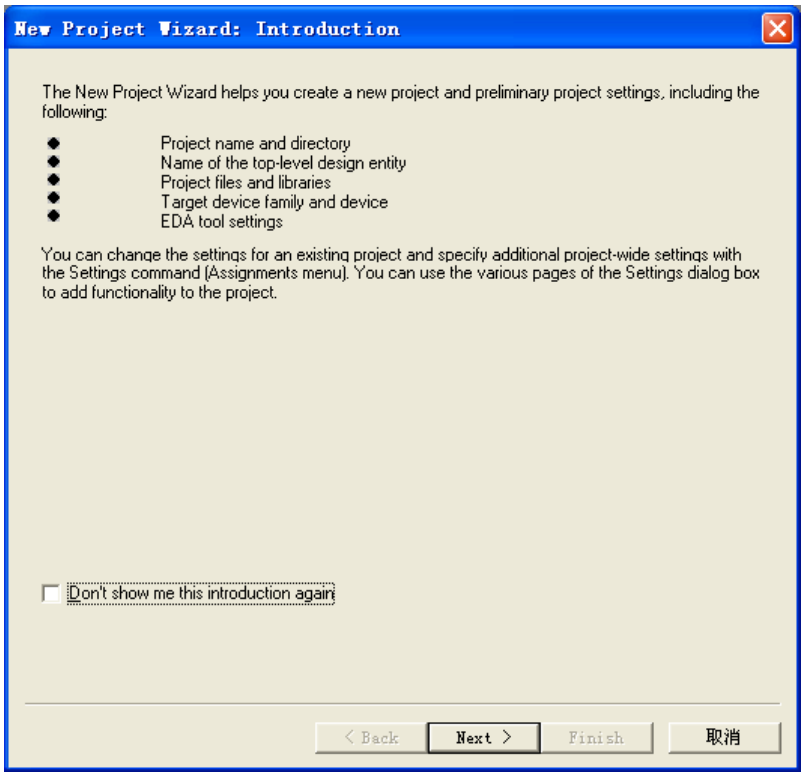


图 3-3 New Project Wizard 界面

3.　输入工程工作路径、工程文件名以及顶层实体名。

这次实验会帮助读者理解顶层实体名和工程名的关系，记住目前指定的工程名与顶层实体名都是 Counter10，输入结束后，如图 3-4 所示。点击 Next。

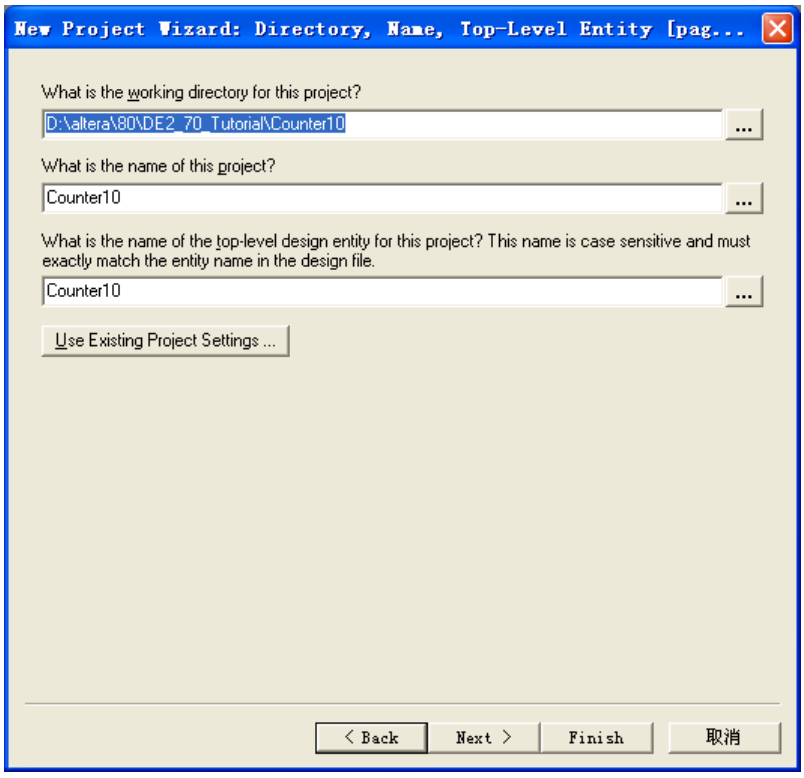


图 3-4 输入设计工程信息

4.　添加设计文件。界面如图 3-5 所示。如果用户之前已经有设计文件（比如.v 文件）。

那么再次添加相应文件，如果没有完成的设计文件，点击 Next 之后添加并且编辑设计文件。

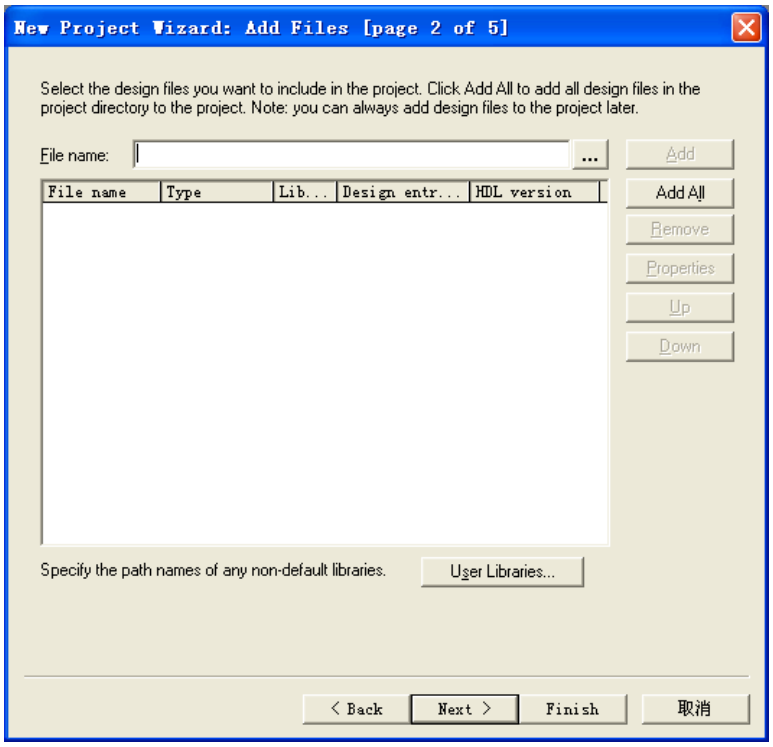


图 3-5 添加设计文件

5. 选择设计所用器件。由于本次实验使用 Altera 公司提供的 DE2-70 开发板，用户必须选择与 DE2-70 开发板相对应的 FPGA 器件型号。

在 Family 菜单中选择 Cyclone II，Package 选 FBGA，Pin Count 选 896，Speed grade 选 6，确认 Available devices 中选中 EP2C70F896C6，如图 3-6。

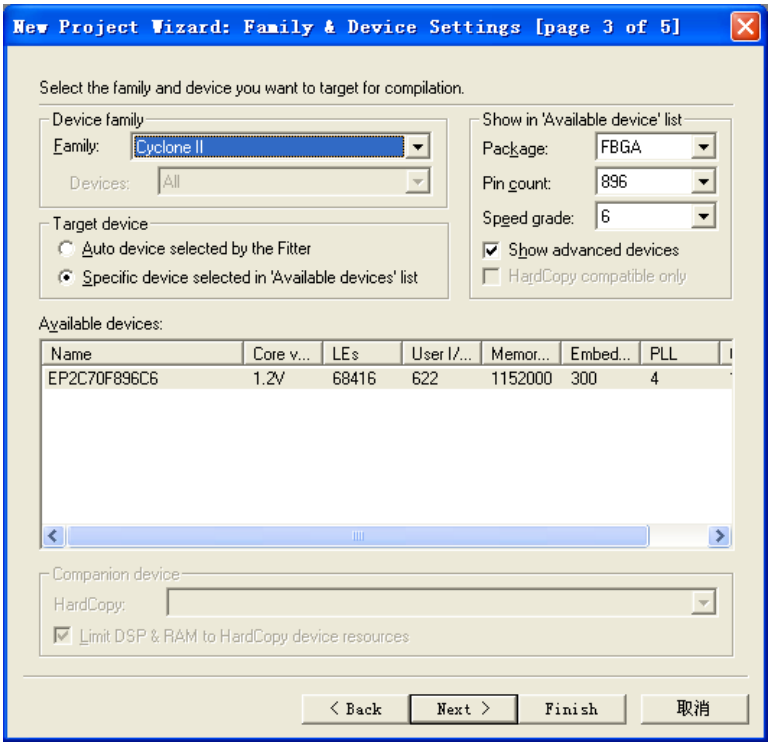


图 3-6 选择相应器件

6. 设置 EDA 工具。设计中可能会用到的 EDA 工具有综合工具、仿真工具以及时序

分析工具。本次实验中不使用这些工具，因此点击 Next 直接跳过设置。如图 3-7。

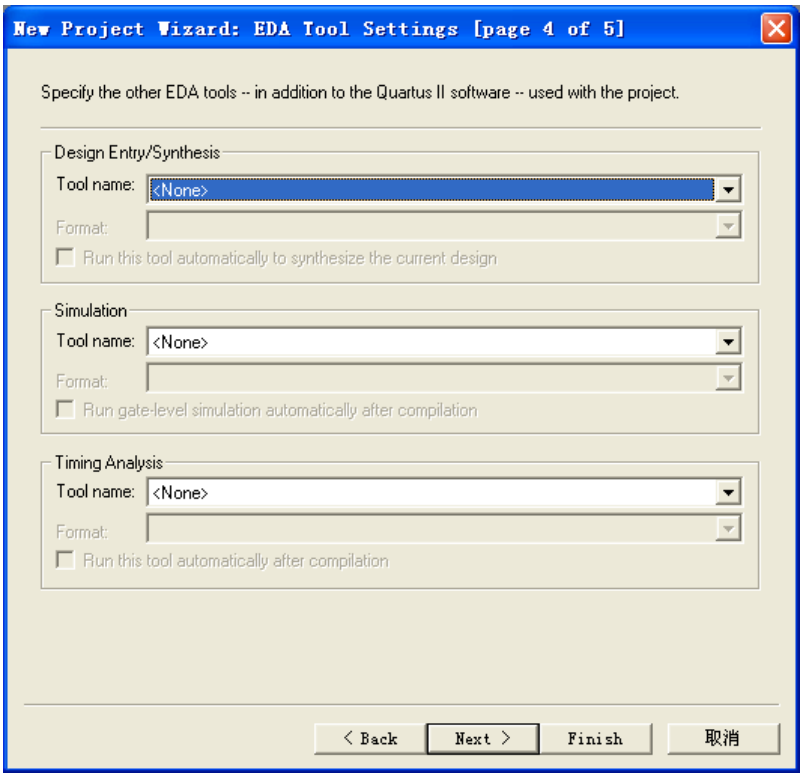


图 3-7 设置 EDA 工具

7. 查看新建工程总结。在基本设计完成后，Quartus II 会自动生成一个总结让用户核对之前的设计，如图 3-8 所示，确认后点击 Finish 完成新建。

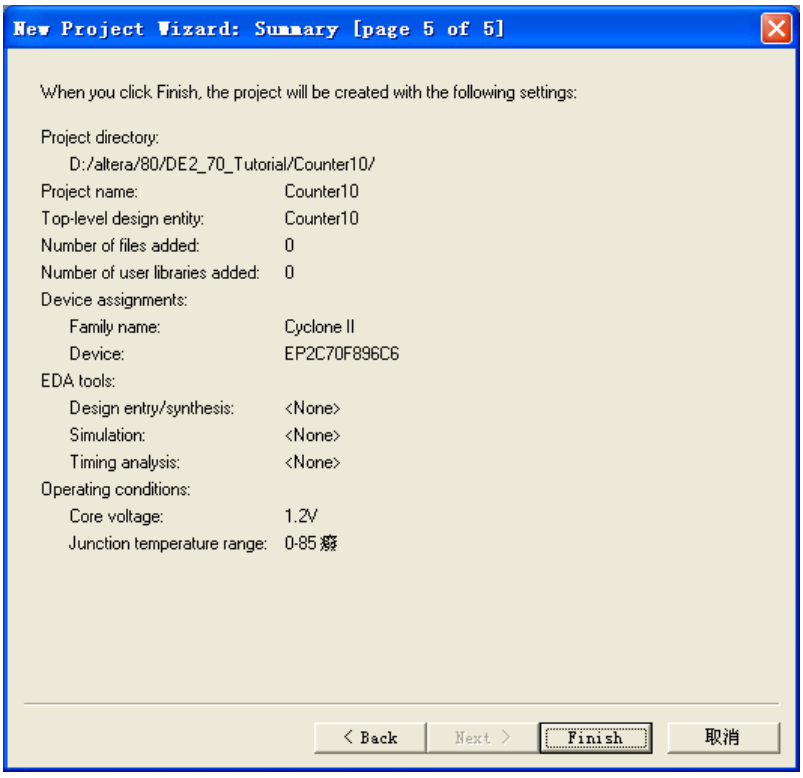


图 3-8 新建工程总结

在完成新建后，Quartus II 界面中 Project Navigator 的 Hierarchy 标签栏中会出现用户正

在设计的工程名以及所选用的器件型号，如图 3-9 所示。

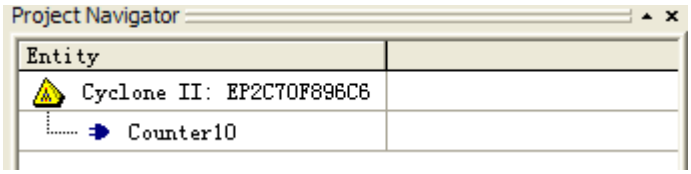


图 3-9 观察正在设计的工程

8. 培养良好的文件布局。

点击菜单项 Assignments->Device，选中 Compilation Process Settings 选项卡，勾上右边的 Save Project output files in specified directory，输入路径(一般为 debug 或者 release)，如图 3-10 所示。

注意：

使用相对路径.\release，以便将工程文件拷贝在不同的 PC 机上运行。

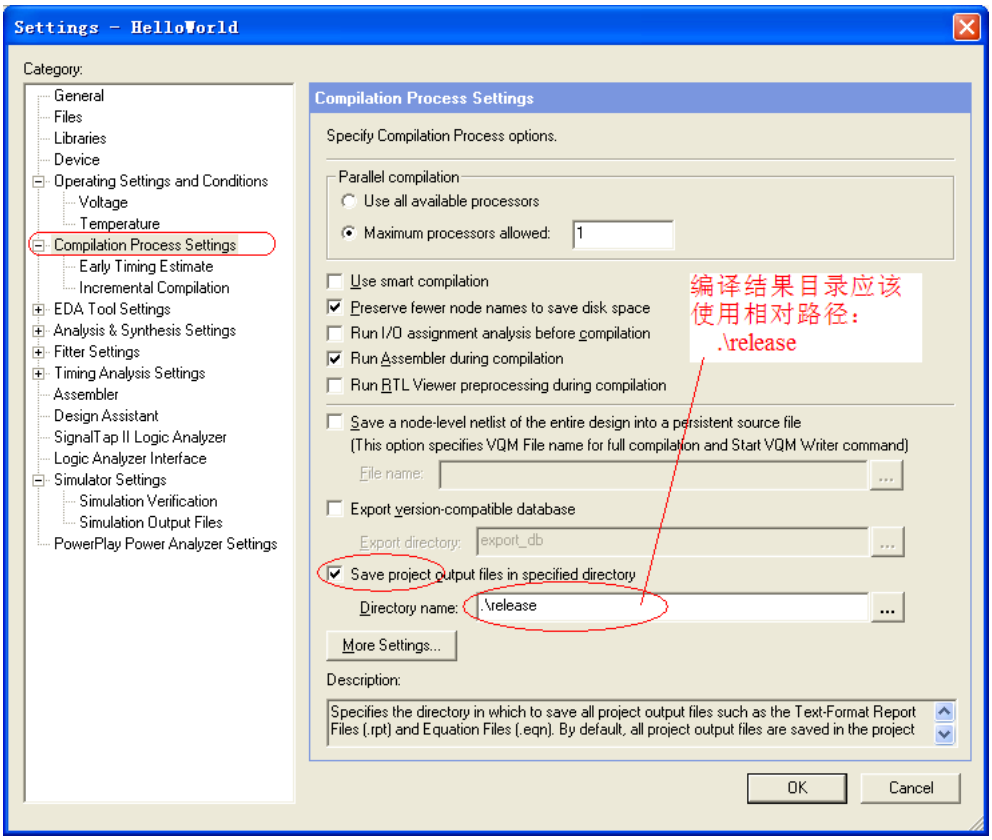


图 3-10　指定单独的编译结果文件目录（相对路径）

9. 添加所需设计文件。

点击菜单项 File->New 或者点击图标 新建一个设计文件，选择 Verilog HDL File，如图 3-11 所示，点击 OK。建立 Verilog 源代码文件。

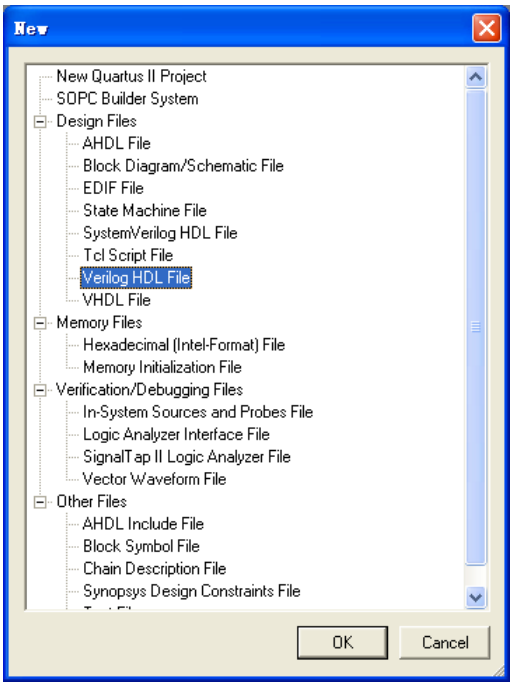


图 3-11　选择设计文件类型

输入如下 Verilog HDL 语言的设计代码：

```
module Counter
(
iclk，
rst_n，
q，
overflow
);
input iclk;
input rst_n;
output reg [3:0] q;
output overflow;

always @(posedge iclk or negedge rst_n)
begin
    if(~rst_n) q <= 4'h0;
    else
    begin
        if(4'h9 == q) q <= 4'h0;
        else q <= q + 4'h1;
    end
end

assign overflow = 4'h9 == q;

endmodule
```

10. 保存设计。点击菜单项 File->Save、点击图标  或者使用快捷键 Ctrl+S 保存设计，如图 3 -12 所示。给设计文件命名 Counter，与模块名相同，注意不是 Counter10，点击保存。

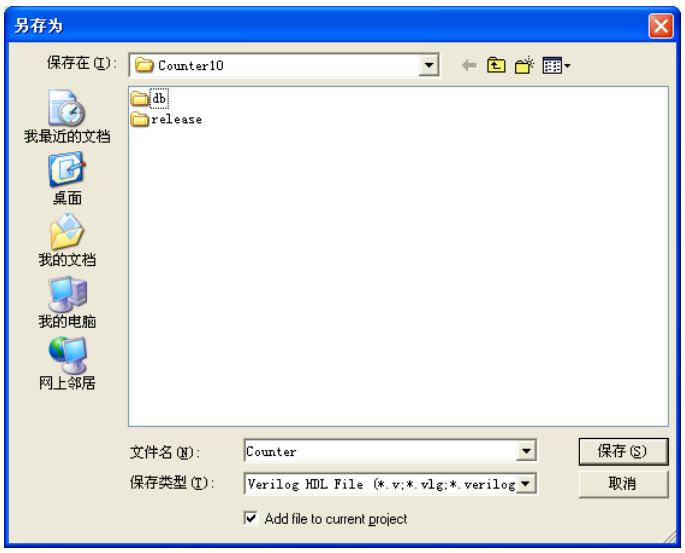


图 3-12　保存设计文件

11. 分析与综合。点击菜单项 Processing->start->Start Analysis & Synthesis、点击图标  或者使用快捷键 Ctrl+K 执行分析与综合。参看图 3-13。

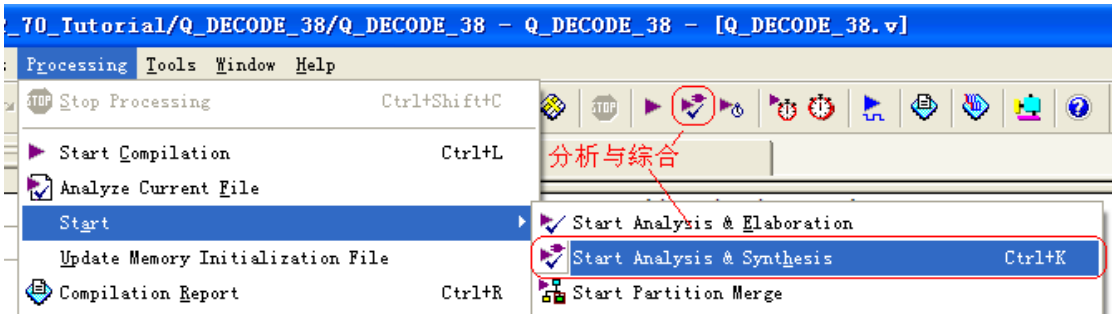


图 3-13　执行 start Analysis & Synthesis（开始分析与综合）

分析与综合完成后，编译出错，错误原因如图 3-14 所示。

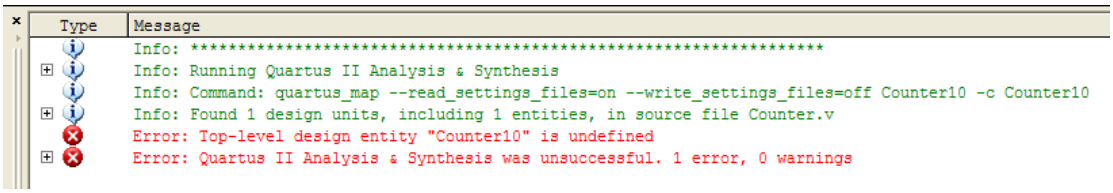


图 3-14　分析与综合错误原因

顶层实体 Counter10 未在源码中定义，必须更改顶层实体为 Counter，这在多文件的工程中经常需要用到。

将左侧的 Project Navigator 切到 Files 标签，对着 Counter.v 文件右击，选择 Set as Top-Level Entity，如图 3-15。

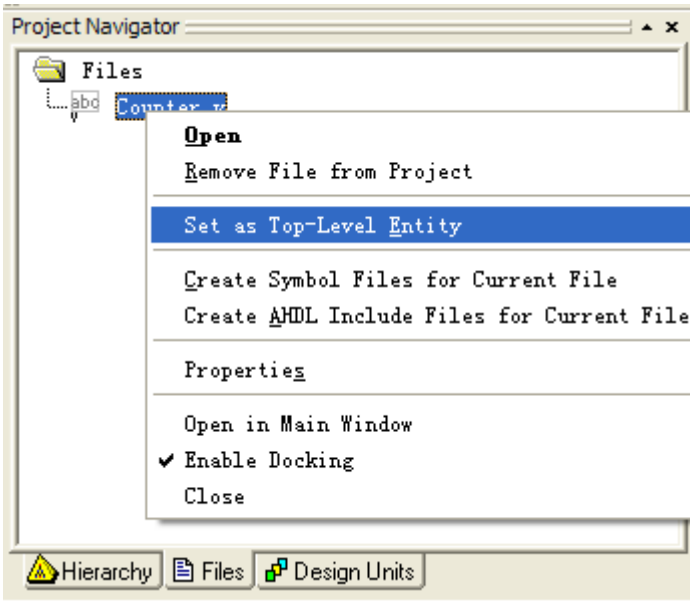


图 3-15　重新指定顶层实体

12. 重新执行分析与综合，结果如图 3-16，出现了 12 个警告，这是因为 qsf 文件中记录的顶层实体在这一步执行时还未更新。

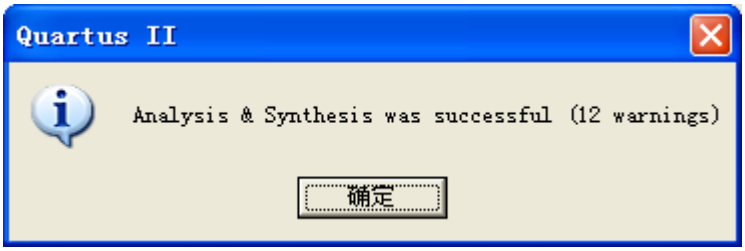


图 3-16　分析与综合结果(第二次执行)

如果再次执行分析与综合，无论你是否删掉原先的编译结果，都会完全成功，如图 3-17。

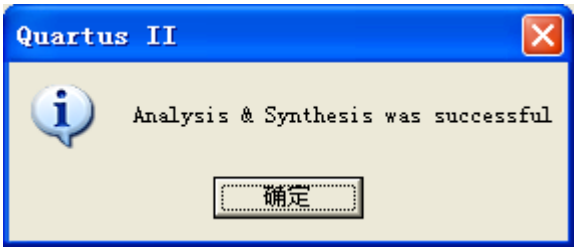


图 3-17　分析与综合结果(第三次执行)

3.2 电路仿真

13. 功能仿真。它是为了检查设计是否在理论上达到预期功能，该仿真不考虑期间实际物理特性。首先创建仿真输入波形文件。仿真时需要对顶层实体的输入管脚提供激励信号，在 Quartus 软件中可以通过波形文件方便的输入。点击菜单项 File->New->Vector Waveform File，如图 3-18 所示。

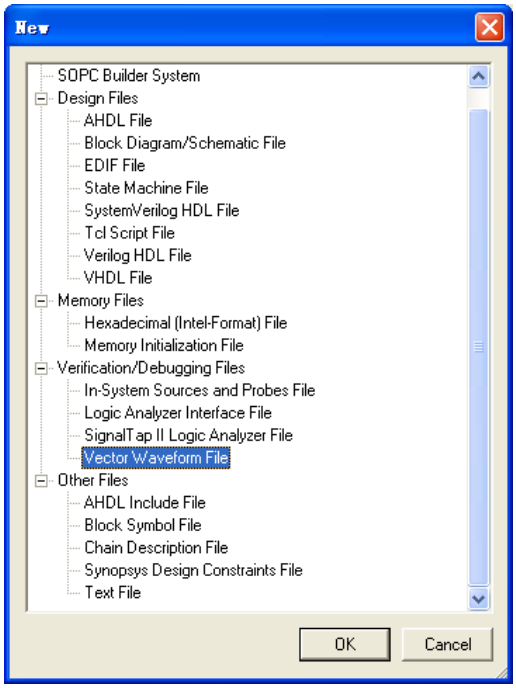


图 3-18　创建波形文件

14. 添加信号结点。在空波形文件中点击右键，如图 3-19 进行选择（或者直接双击）。

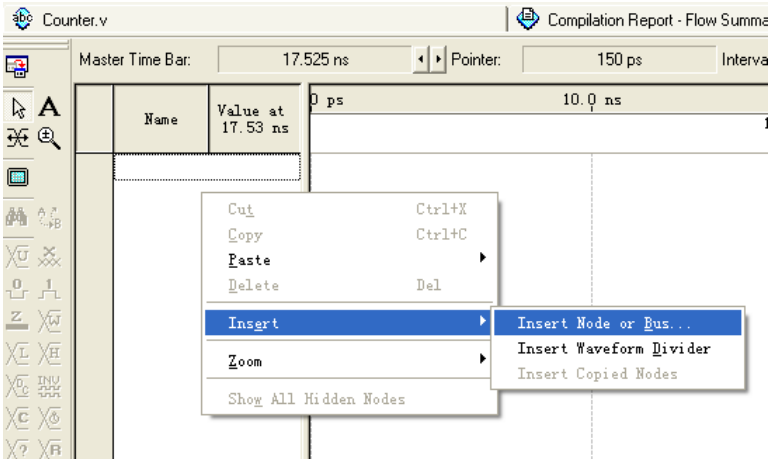


图 3-19　添加结点右键菜单

单击 Insert Node or Bus 后，出现如图 3-20 所示对话框。

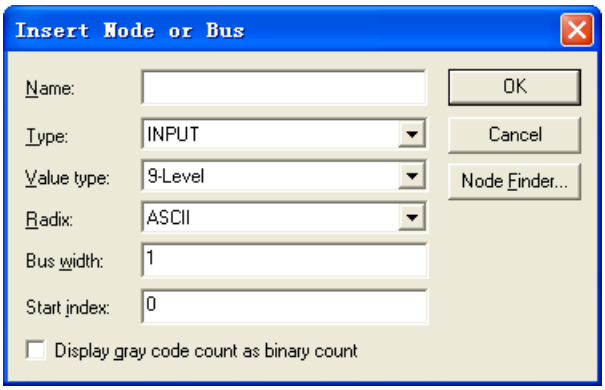


图 3-20　添加结点对话框

选择 Node Finder 按钮可以从结点列表中选择我们需要的，而避免一个一个输入结点

的麻烦。

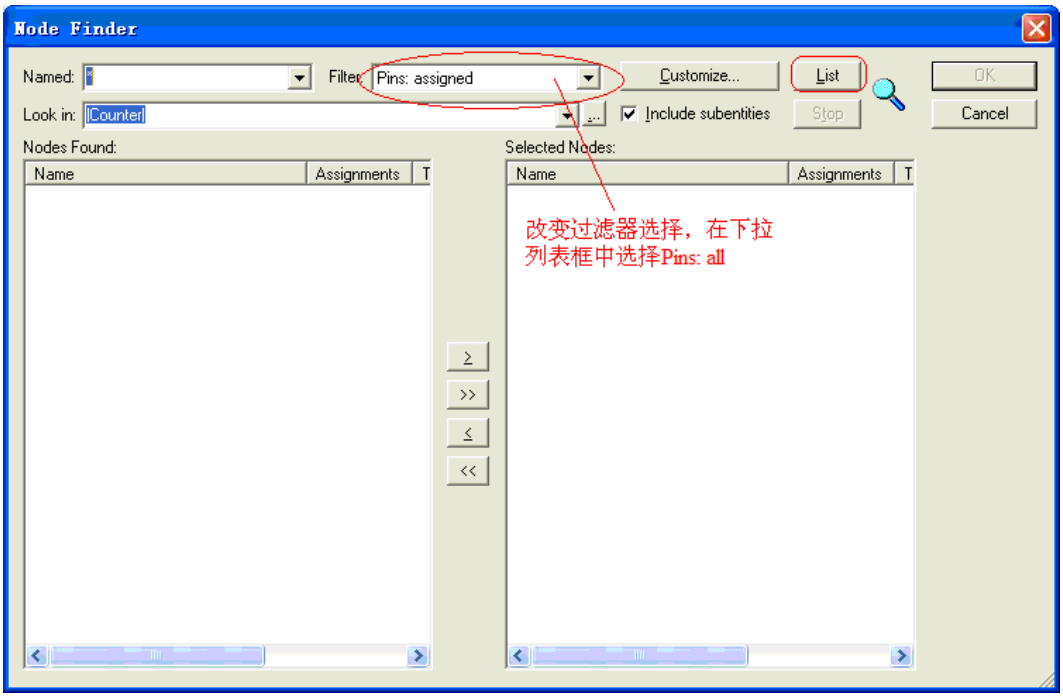


图 3-21　Node Finder 对话框

Fitter 选择 Pin:all，点击 List 按钮。出现如图 3-22 所示的结点列表。

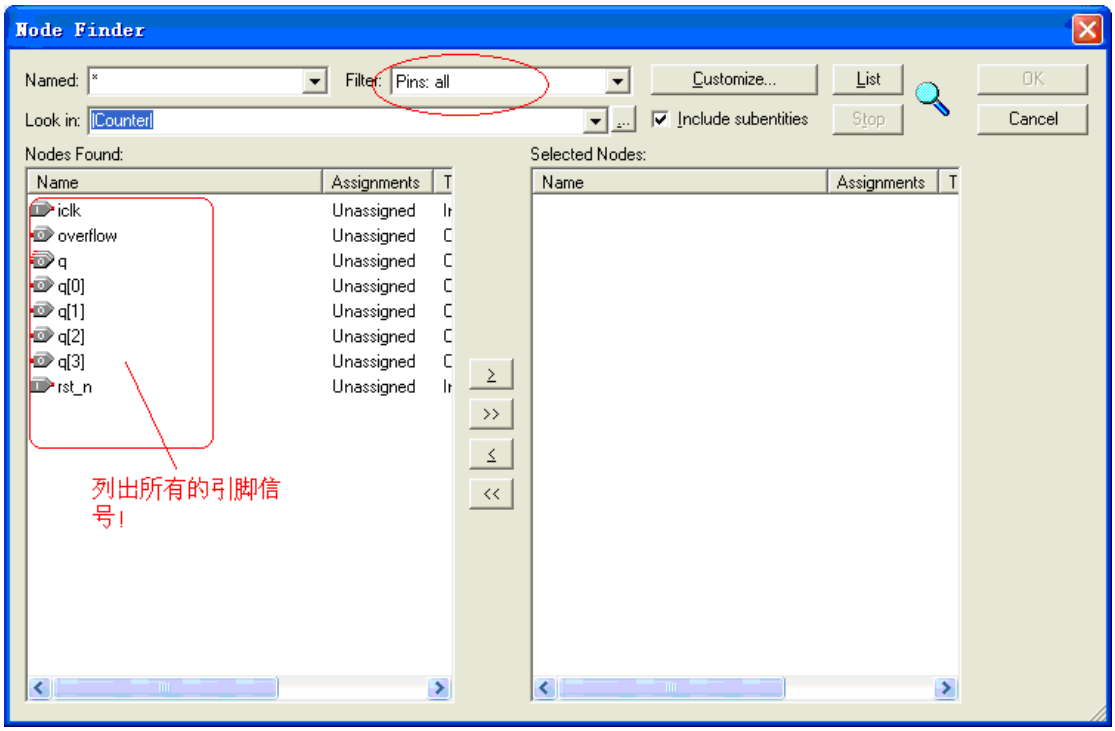


图 3-22　结点列表

简单起见，可以直接点>>按钮，将所有结点加入右侧 Select Nodes 栏中。完成后如图 3-23 所示。点击 OK 按钮确认。

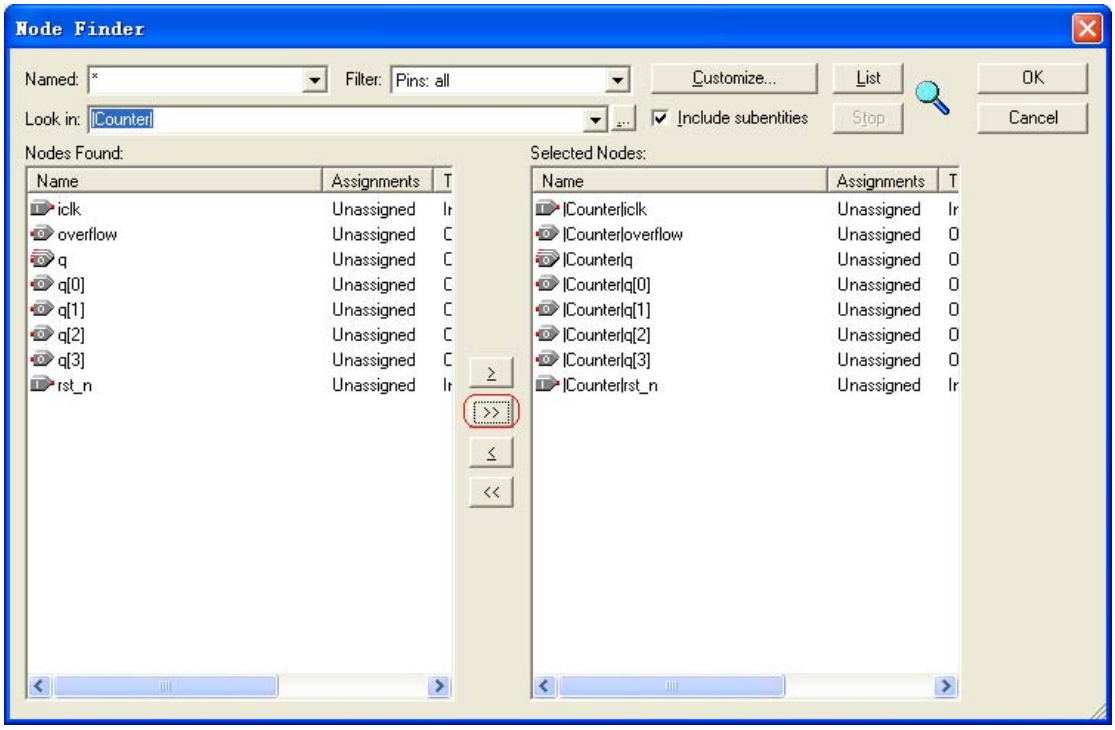


图 3-23　添加结点到右侧

点击 OK 后返回添加结点对话框。如图 3-24 所示。

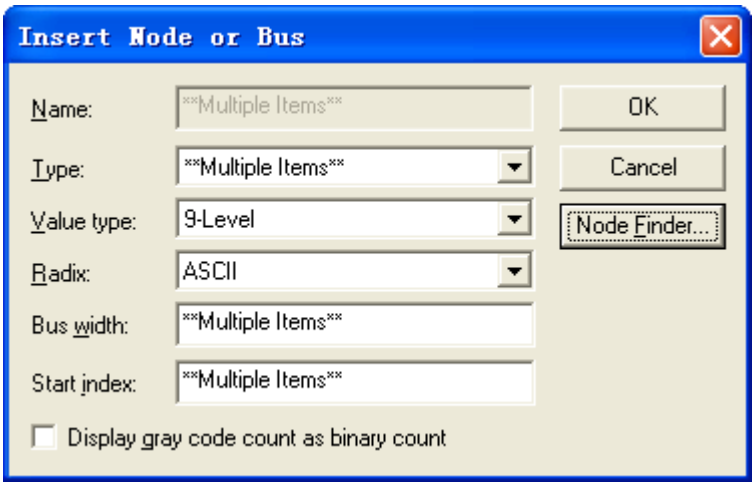


图 3-24　添加结点后的对话框

点击 OK 确定，波形文件将如图 3-25 所示。

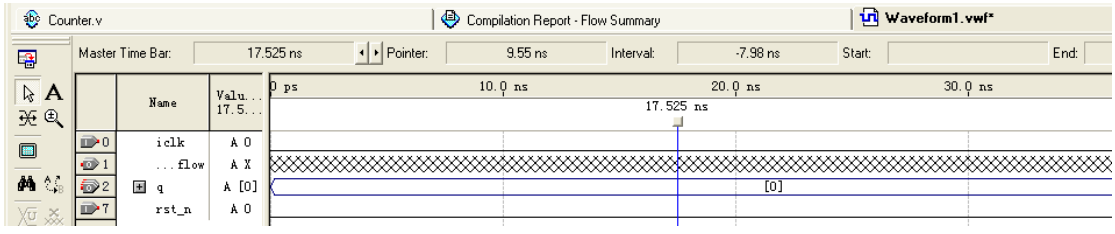


图 3-25　波形文件

15. 将 iclk 设为方波。右击 iclk 信号，选择 value->clock..，如图 3-26 所示。

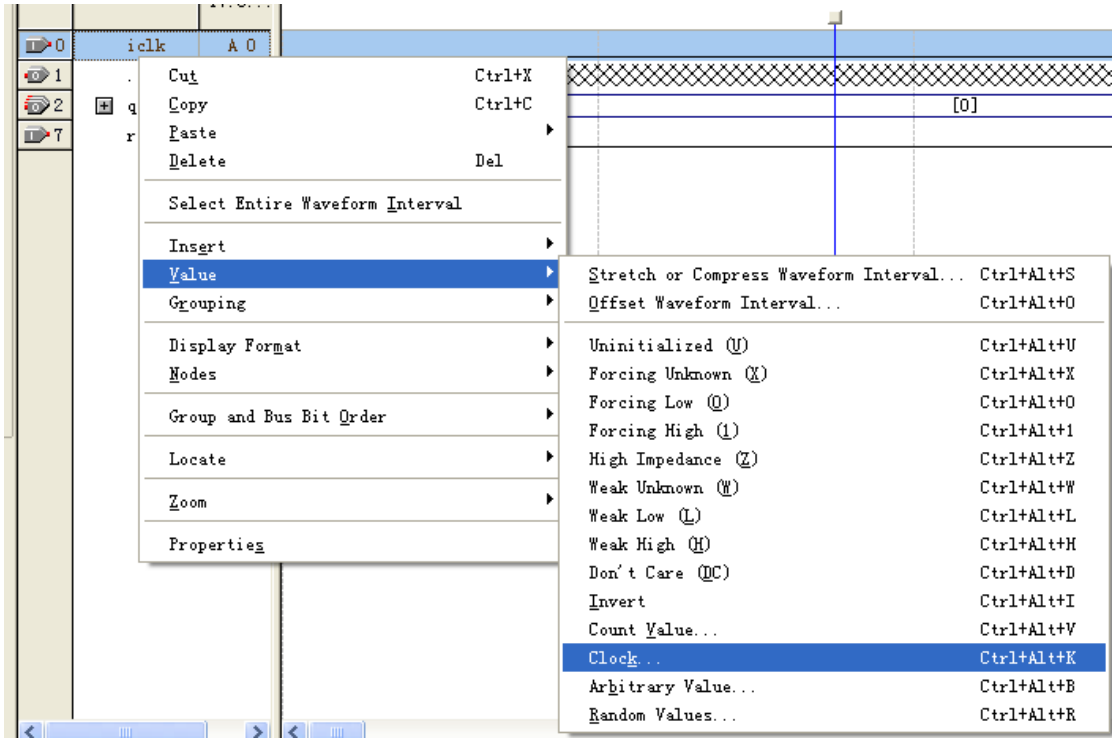


图 3-26　将 iclk 改为方波

在弹出的 clock 设定对话框中把周期调整为 20ns，如图 3-27。Duty cycle 的意思是占空比，即是指高电平在一个周期之内所占的时间比率。

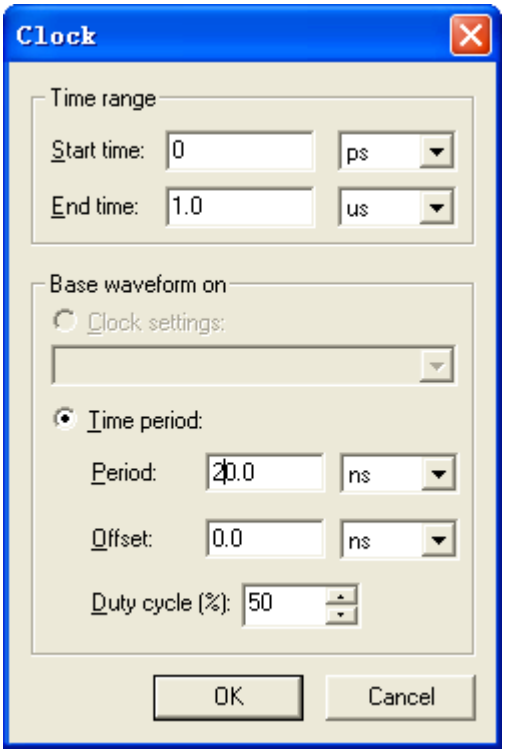


图 3-27　时钟的周期设置

16. 将 rst_n 改成低 20ns 后持续高电平。选中 rst_n 信号，单击左侧图标强制设为高电平。在波形上拖动鼠标选中前 20ns，单击左侧图标强制设为低电平。

完成后波形如图 3-28 所示。输出波形可不管。

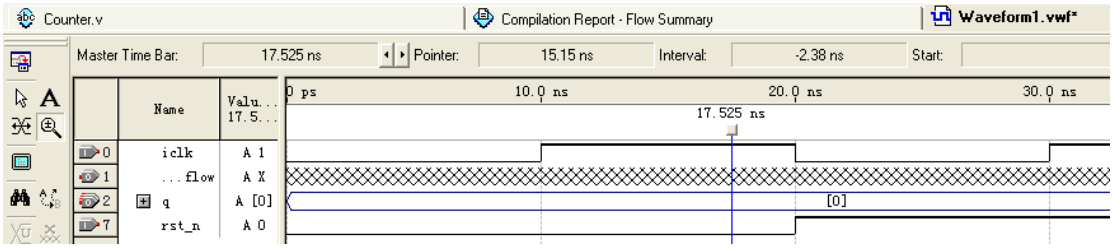


图 3-28 波形文件

17. 保存波形文件 counter.vwf，如图 3-29，这里的命名可以随意。

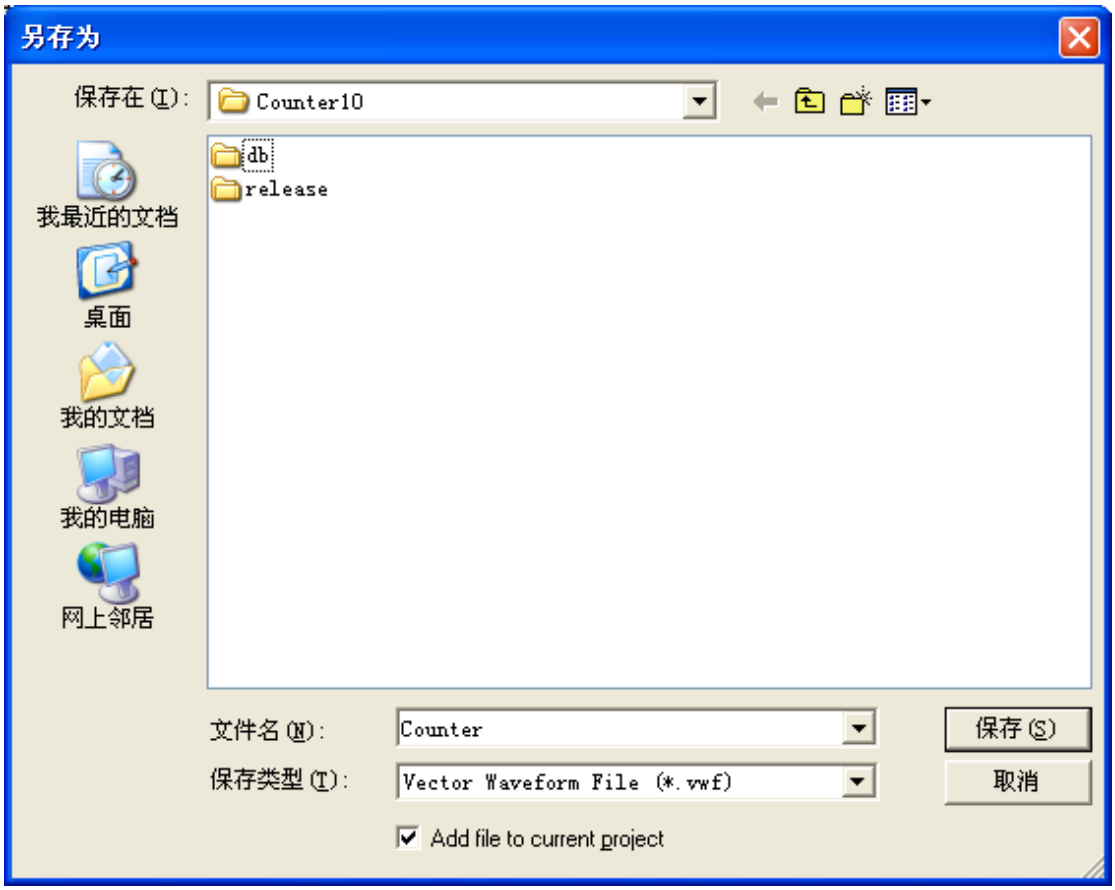


图 3-29 保存波形文件

18. 波形文件生成后，直接点击仿真按钮会提示错误，见图 3-30，这是因为没有先产生功能仿真网表。

图 3-30 未生成网表错误

19. 要生成功能仿真网表，首先设置仿真模式。点击菜单项 Assignment->Settings，选中 Simulator Settings 选项卡，出现图 3-31 所示对话框。在 Simulation mode 中选择 Functional，Simulation input 选择刚才建立的波形文件，完成后点击 OK。

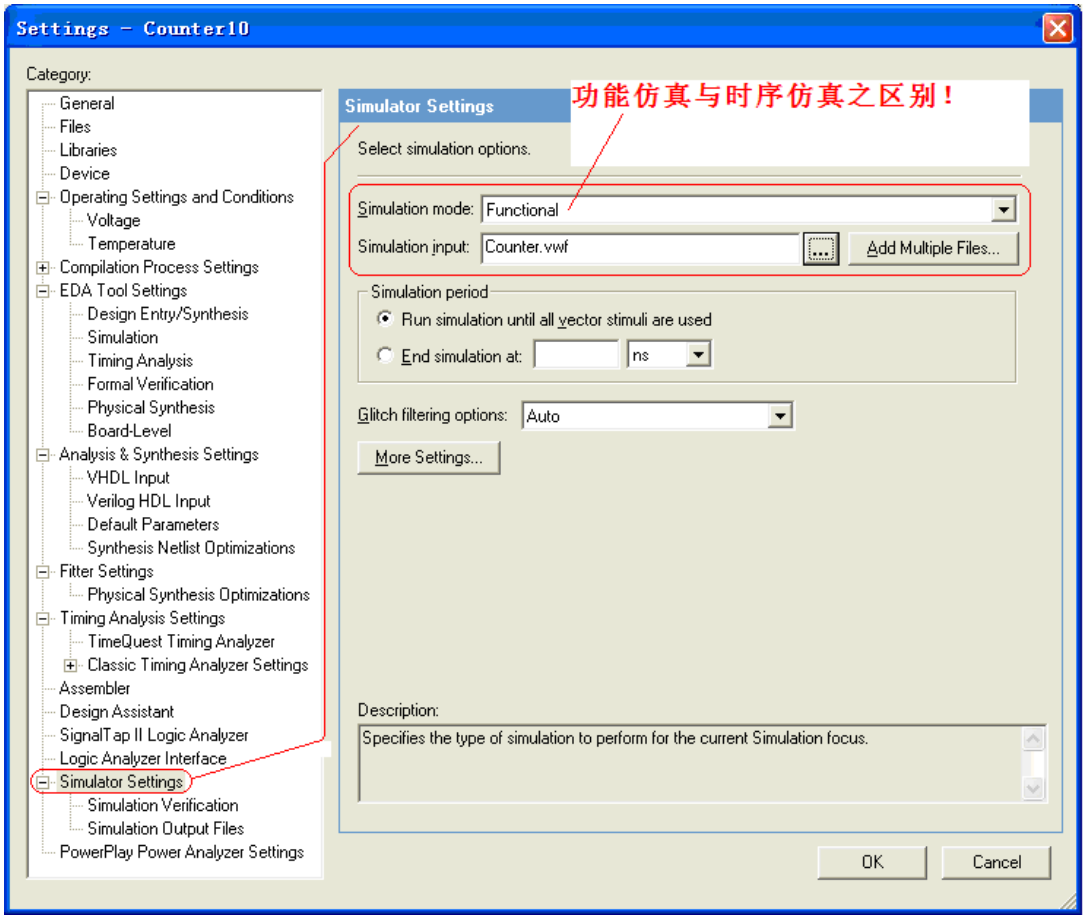


图 3-31 仿真模式设置对话框

点击菜单项 Processing->Generate Functional Simulation Netlist，产生功能仿真所需的网表，参看图 3-32。完成后结果显示如图 3-33。

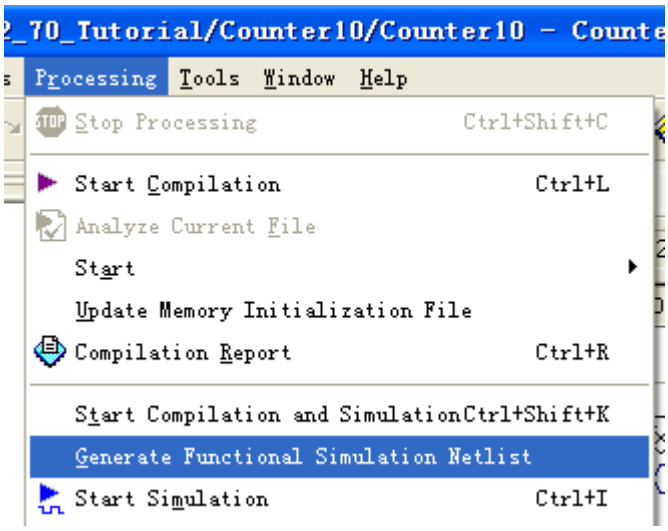


图 3-32　生成功能仿真网表的操作菜单项

图 3-33　功能仿真网表产生结果显示图

20. 点击菜单项 Processing->Start Simulation 或 工具按钮启动功能仿真。如图 3-34，完成后结果显示如图 3-35。

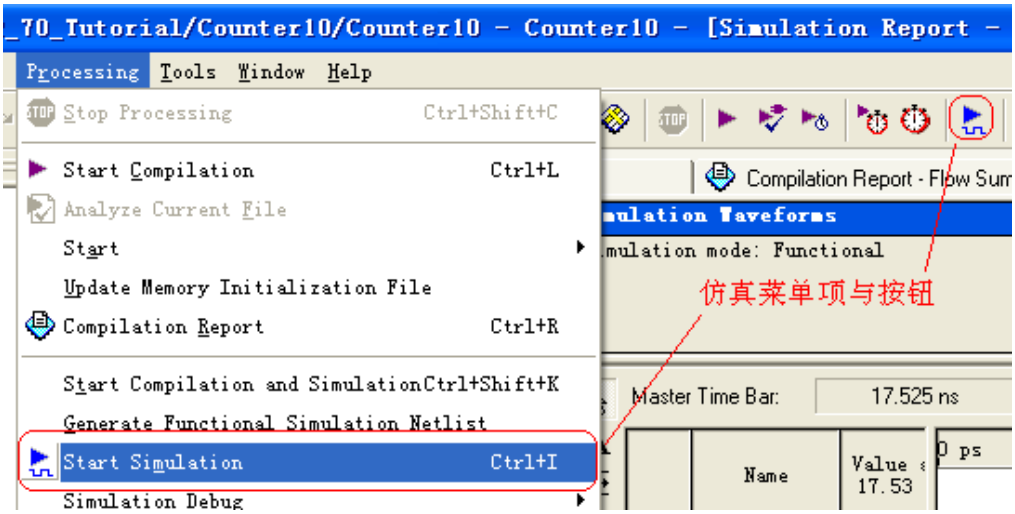


图 3-34　仿真菜单项与按钮

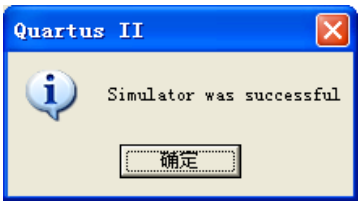
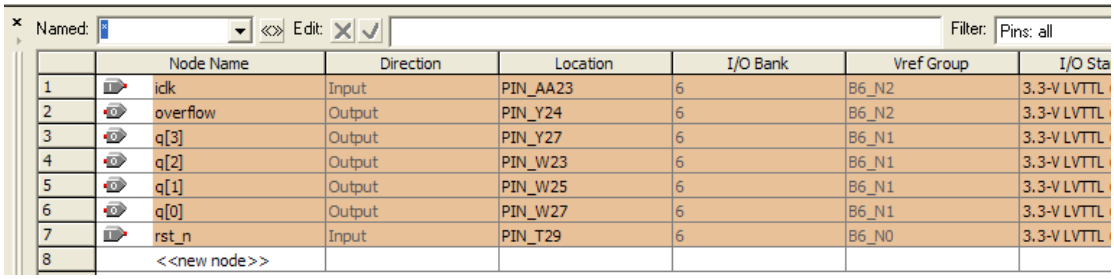


图 3-35　仿真结果

21. 配置引脚。仿真完成后，确认功能正确后，可以进行分配引脚的操作。根据所提供的 DE2-70 用户指导手册，将计数器的 q 输出配置到 DE2-70 开发板的 4 个绿 LED（LEDG[3]-LEDG[0]）上，overflow 接 LEDG[4]，rst_n 接 KEY[0]，clk 接开关 SW[0]。（参考实验一）参考图 3-36，注意 Y24 不是 V24。



	Node Name	Direction	Location	I/O Bank	Vref Group	I/O Sta
1	clk	Input	PIN_AA23	6	B6_N2	3.3-V LVTTL
2	overflow	Output	PIN_Y24	6	B6_N2	3.3-V LVTTL
3	q[3]	Output	PIN_Y27	6	B6_N1	3.3-V LVTTL
4	q[2]	Output	PIN_W23	6	B6_N1	3.3-V LVTTL
5	q[1]	Output	PIN_W25	6	B6_N1	3.3-V LVTTL
6	q[0]	Output	PIN_W27	6	B6_N1	3.3-V LVTTL
7	rst_n	Input	PIN_T29	6	B6_N0	3.3-V LVTTL
8	<<new node>>					

图 3-36 分配引脚图

注意：clock 相关：DE2_70 开发板没有办法直接输出低频方波，使用开关手动控制。

22. 完成引脚分配后，全编译文件。点击菜单项 Processing->start compilation、点击图

标或使用 CTRL+L 执行全编译，如图 3-37 所示。

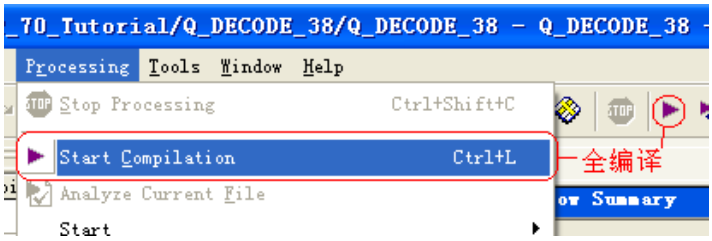


图 3-37　执行 start compilation

编译结果如图 3-38 所示。

图 3-38　全编译结果显示

23. 时序仿真。其主要用途是查看实际设计的电路运行时是否满足延时要求，时序仿真考虑了电路实际运行的延时等因素。

单击菜单中 Assignment->Settings，选中 Simulator Settings 选项卡，在 Simulation mode 中选择 Timing，Simulation input 选择刚才建立的波形文件，完成后点击 OK，如图 3-39。

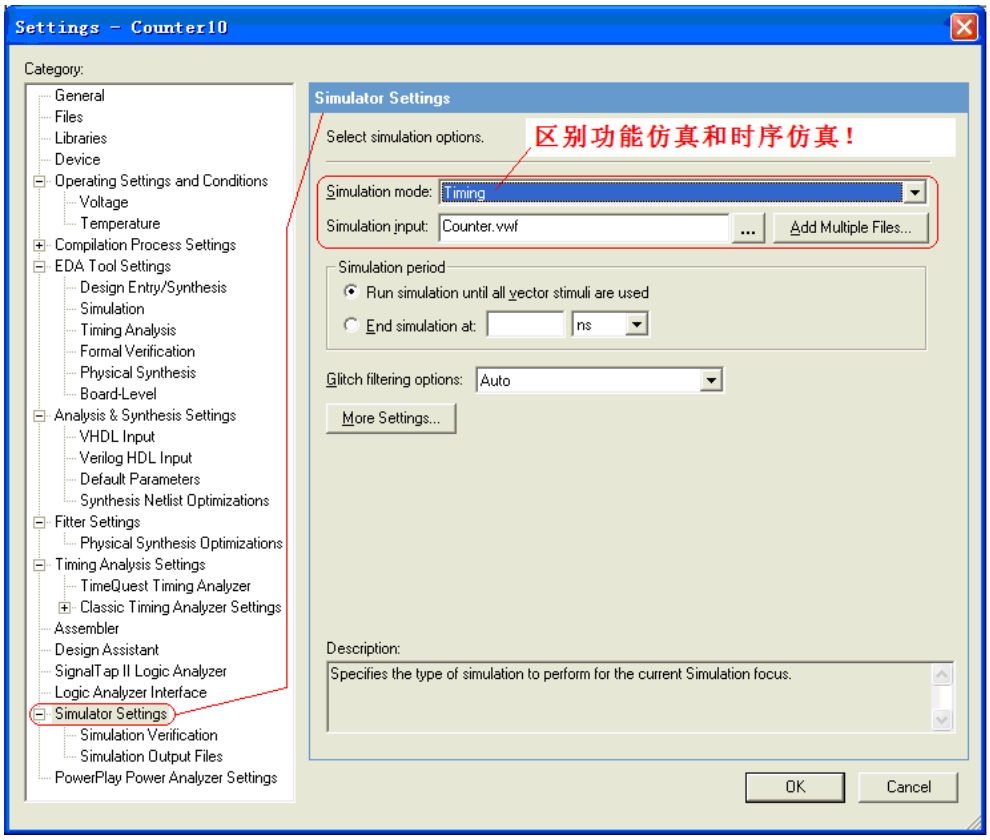


图 3-39 仿真模式设置对话框（时序仿真）

特别注意：图 3-31 和图 3-39 区别了功能仿真和时序仿真。

如果是 8.0 版，在左侧带问号的 Quartus II Simulator (Timing)处右击 start，启用时序仿

真，如图 3-40A.

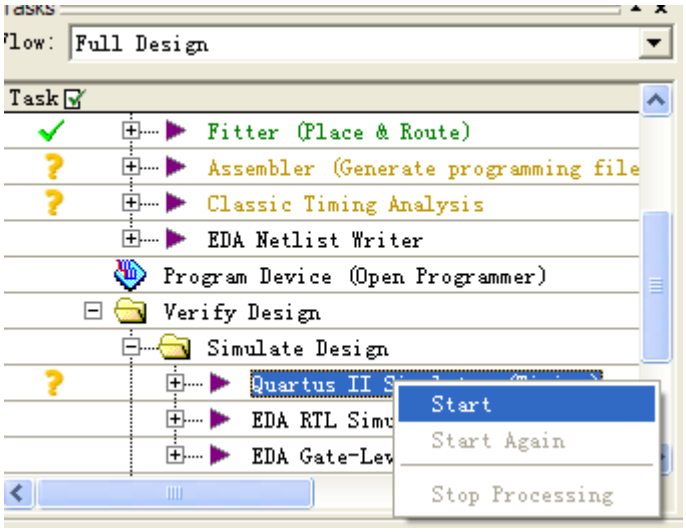


图 3-40A 启用时序仿真

如果是 7.2 版，由于没有 Tasks 窗口，需要在 Processing->Start 菜单按照 **A—E** 的步骤执行。如图 3-40B 所示。每一步骤完成会弹出一个对话框，单击 OK 或者确定。

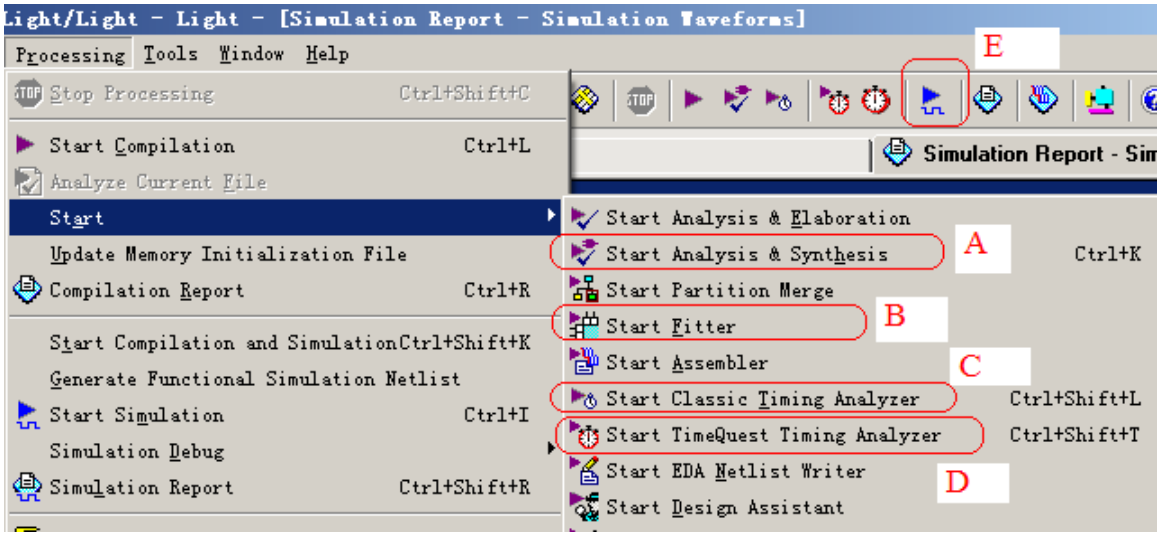


图 3-40B　时序仿真的后五步操作图解

仿真结果如图 3-41 与 3-42 所示。

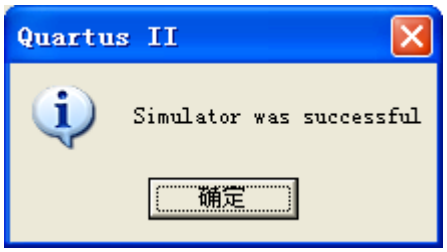


图 3-41 仿真结果图

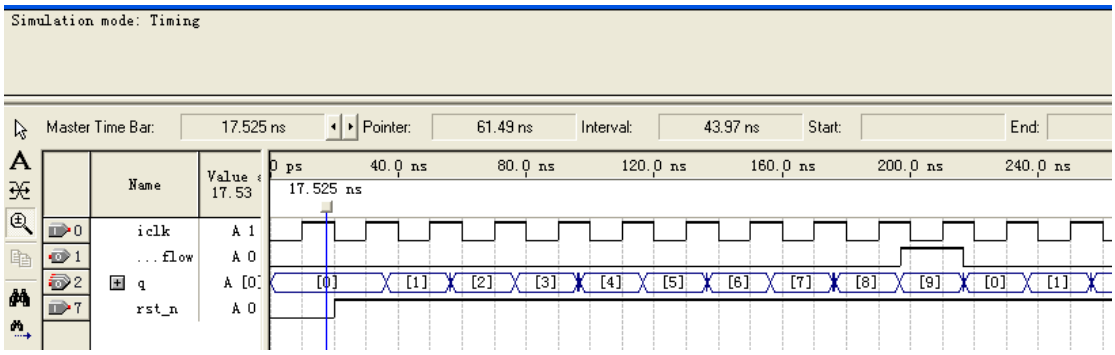


图 3-42 时序仿真波形

24. 将设计下载在 FPGA 中。完成设计后就可以下载到板上实际运行，点击菜单项 Tools->Programmer 或点击图标  打开程序下载环境。点击 start 开始下载。(参考实验一)

25. 手工拨动 SW[0]，测试实验结果。

3.3 逻辑分析仪 SignalTap II 的使用

26. **首先将手工开关时钟换回 50Mhz 的时钟，否则由于时钟过于低速，SignalTap II 抓取不到波形。**方法是在引脚配置中将 iclk 指定 AD15，之后全编译工程，并且**下载运行**！

可以看到绿灯有 5 个在亮，最左边的暗一点，如图 3-43A 所示。否则，很可能是引脚分配出错，如图 3-43B 中出现了 Y27 设成了 V27 的错误。

图 3-43A　5 个灯都亮，正确。

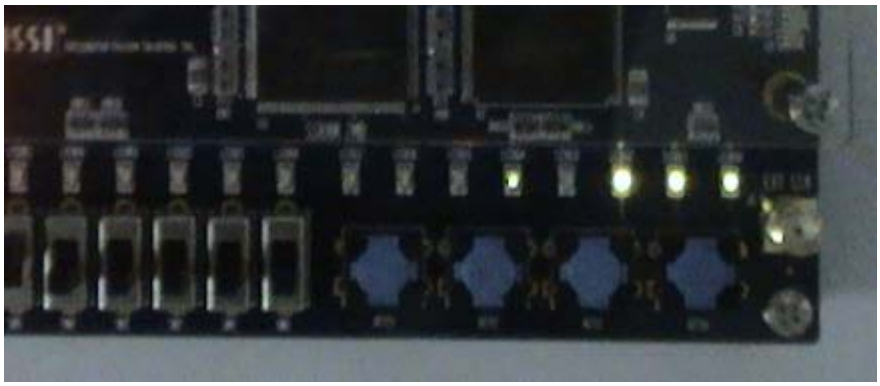


图 3-43B　只有 4 个灯亮，错误。

27. 新建 SignalTap II 文件。点击菜单项 File->SignalTap II Logic Analyzer File

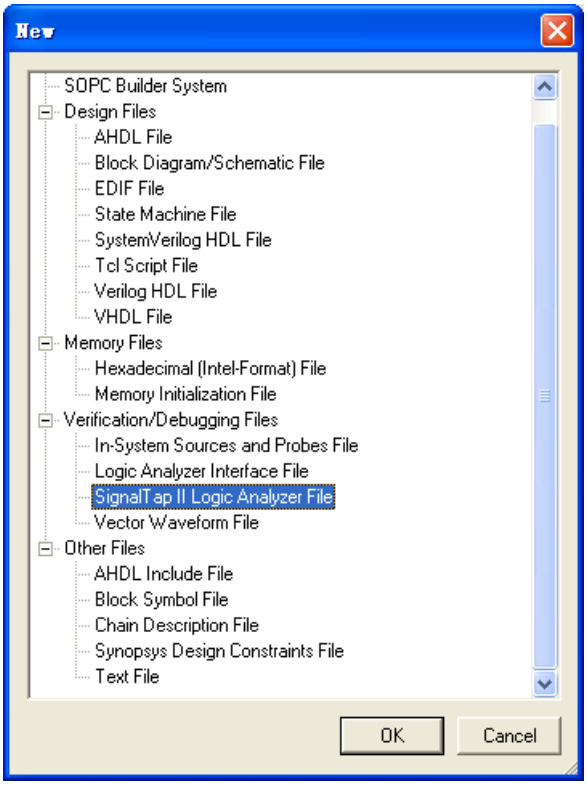


图 3-44　新建逻辑分析仪文件

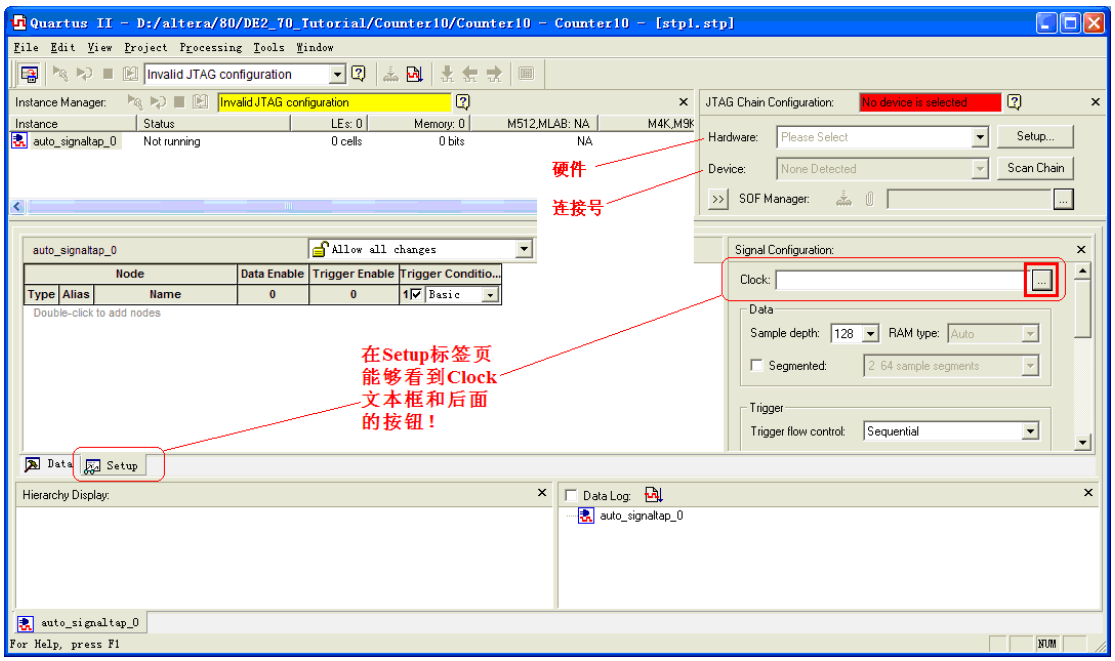


图 3-45　逻辑分析仪文件

由于窗口界面面积较小，可以通过文件左上角的  按钮将文件子窗口与主窗口分离。

28. 选择硬件，首先连接号 DE2-70，然后在文件右上的 Hardware 下拉菜单中选择 USB-Blaster，选好后应能自动识别出 Device 是 EP2C70。选择后的情况如图 3-46 所示。

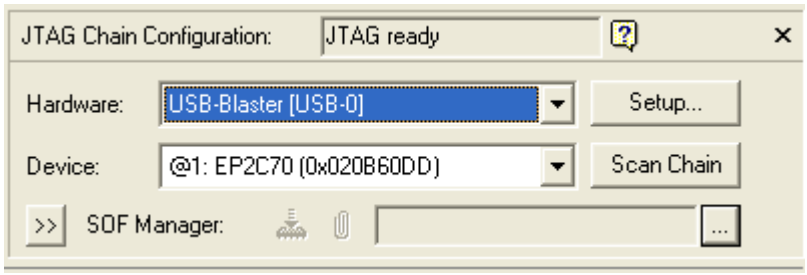


图 3-46　选择硬件环境

29. 选择逻辑分析仪时钟，本实验中就以计数器时钟作为逻辑分析仪时钟。确认左下角的标签页是 setup，然后点击右下侧 SignalConfiguration 中的 Clock 栏后的按钮。出现如图 3-47 所示。

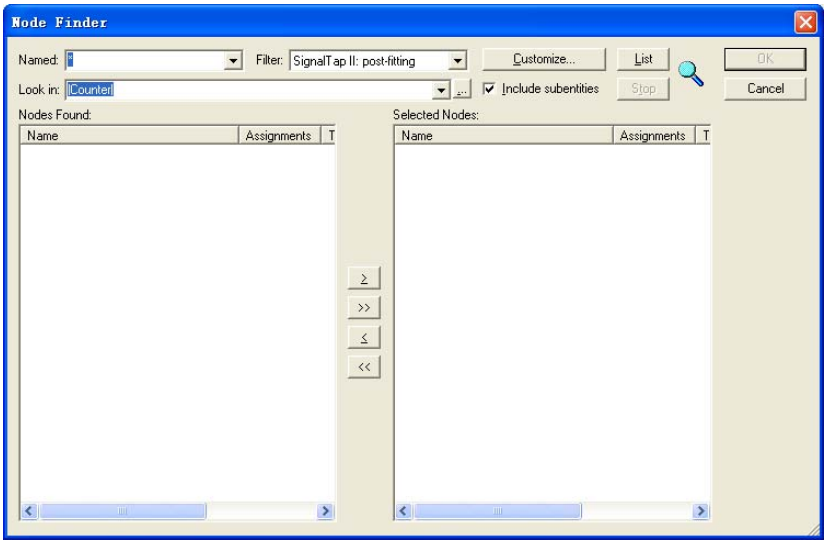


图 3-47　结点查找对话框

Fitter 选择 SignalTap II：post-fitting，点击 List 按钮，左侧出现可选结点，选择其中的 iclk，点击中间的≥按钮。完成后如图 3-48 所示。

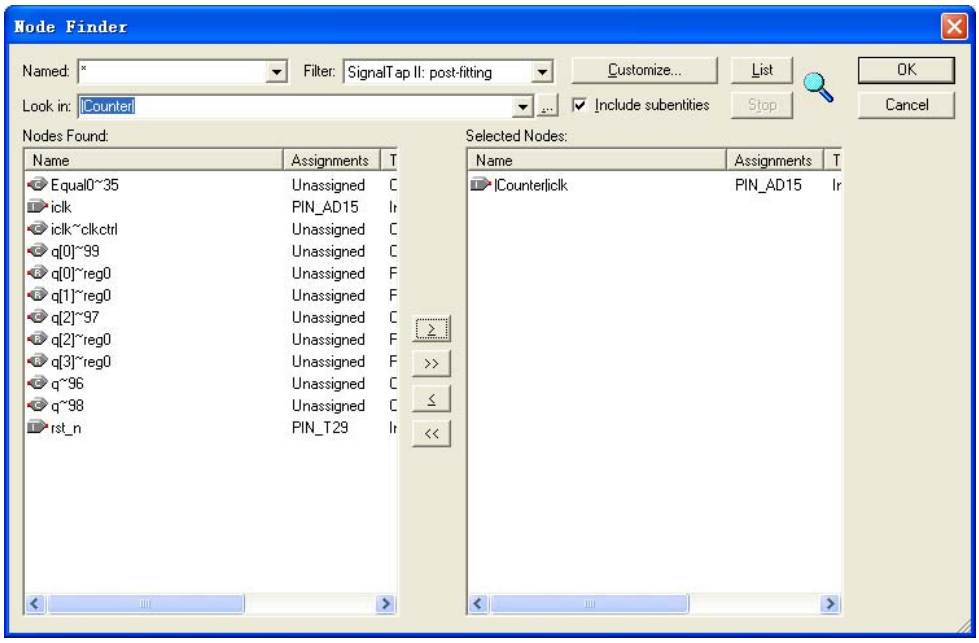


图 3-48　选择时钟结点

完成后出现确认对话框。点击 OK 按钮。参看图 3-49

图 3-49　确认对话框

而后出现时钟结点选择界面，如图 3-50 所示。

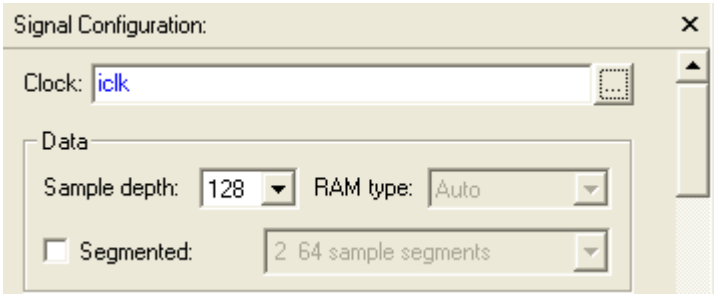


图 3-50　完成时钟结点选择

30.　完成后可在栏中选择存储深度、触发级等选项，这里采用默认设置。下面加入需要观测的结点。在左侧空白区域双击，再次出现选择结点对话框。点击 List 列出所有可选结点。将关心的结点选择好，选择 Pins:all 列出所有引脚，除 iclk 外全部导入。完成后如图 3-51 所示。点 OK 确定。

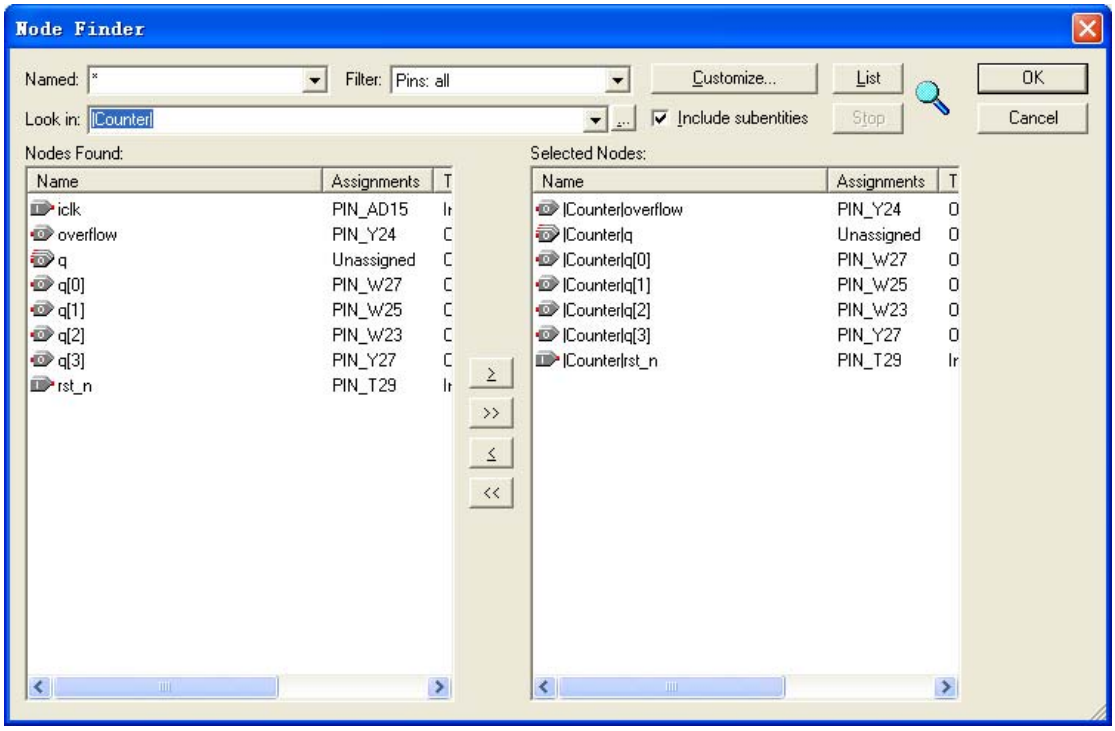


图 3-51 选择观察结点

完成后如图 3-52 所示。

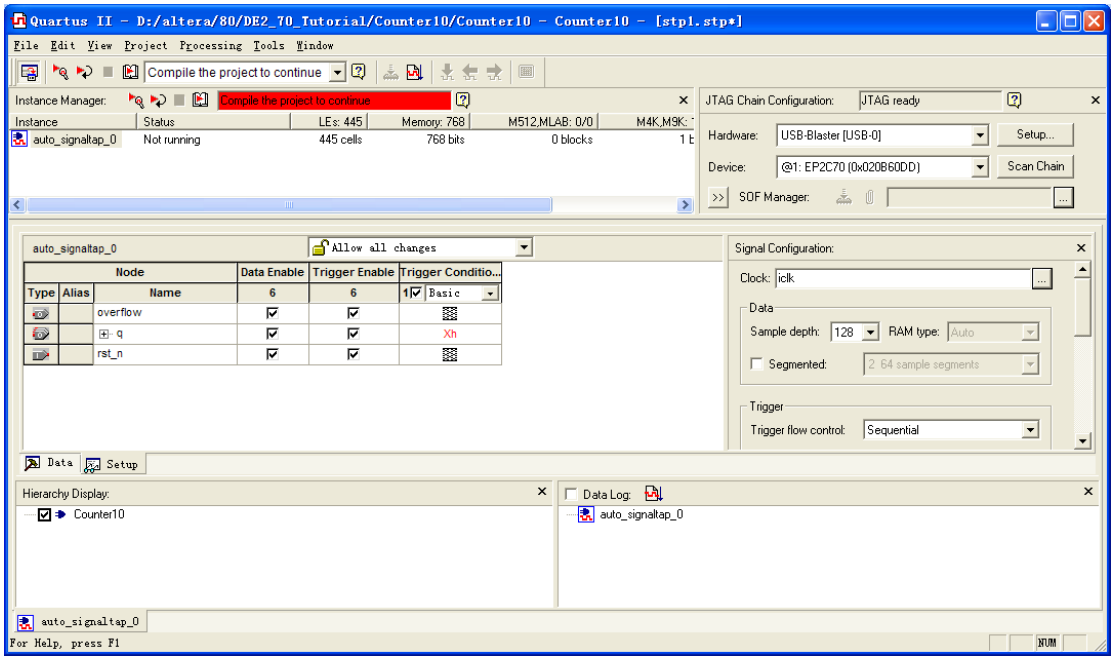


图 3-52　结点完成后界面

31.　完成以上步骤后保存 SignalTap II 文件，可取名为 Counter.stp，参看图 3-53。并在询问是否设置为当前工程的 SignalTap 时选“确定”。再到 Quartus 主界面中执行全编译。编译完成后下载文件到 FPGA。

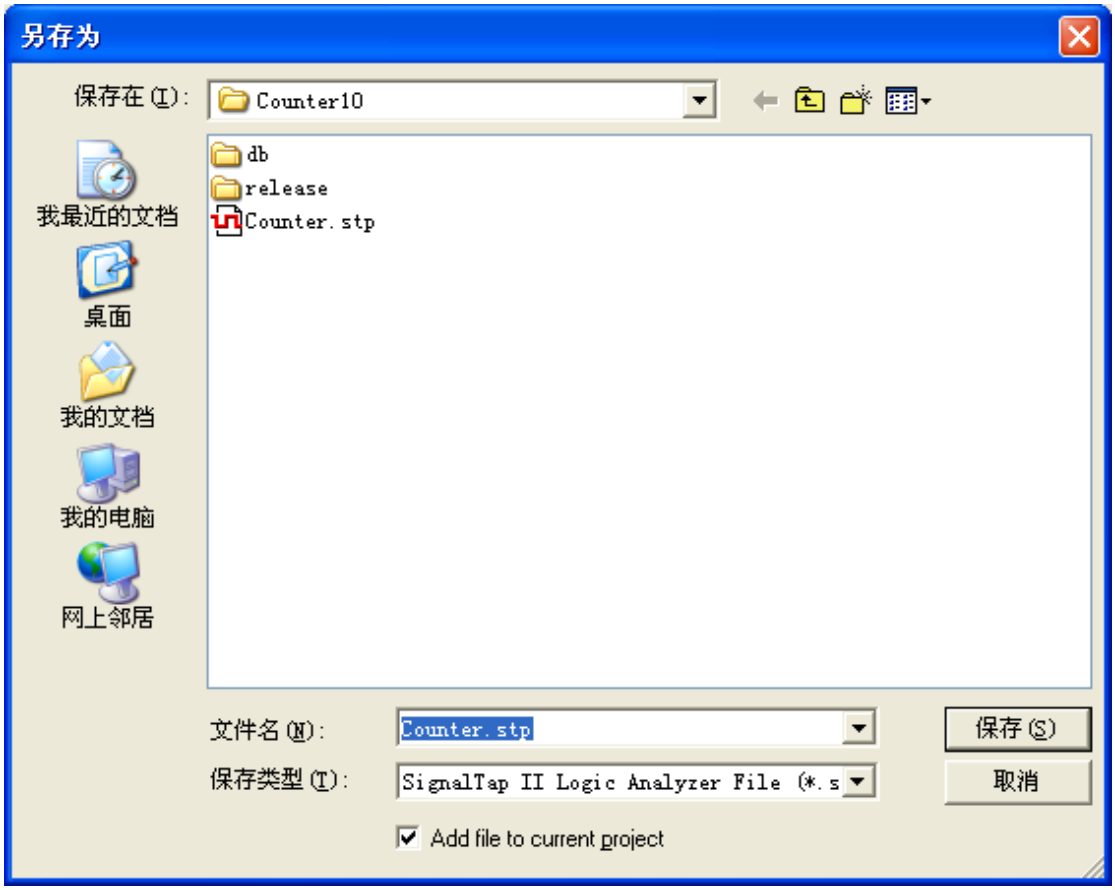


图 3-53　保存 SignalTap II 文件

32.　完成后，再次回到逻辑分析仪文件，点击左上 按钮开始分析。就可以观察到

实际捕获的波形。如图 3-54 所示。

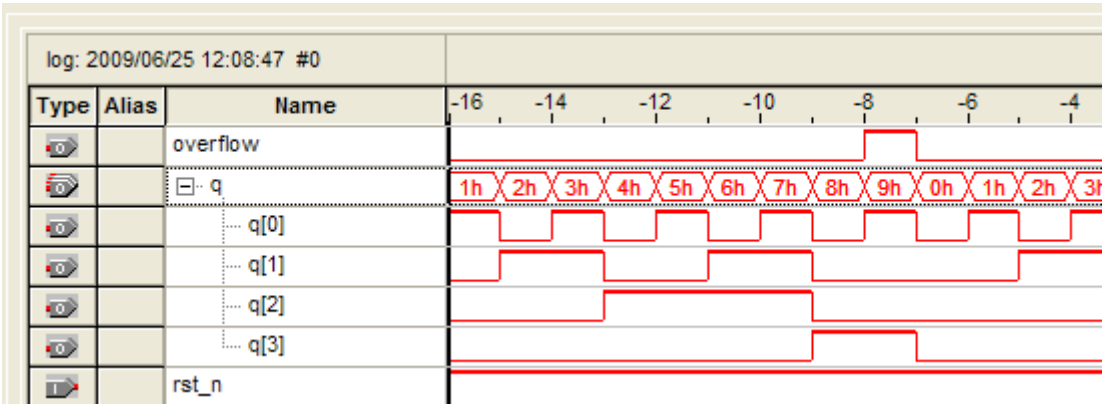


图 3-54 SignalTap II 抓取到的波形

至此，整个实验结束。读者应该体会到了从设计->仿真->下载->波形抓取的四个经典步骤。

◆ 本实验指导结束

第 4 章　实验三　灯光控制实验

● 实验说明

该实验将使用符号框图设计在 DE2-70 开发平台上设计一个非常基本的组合逻辑电路。通过该实验，让读者学会不同于硬件描述语言的一种基于符号框图的设计方案。实验中，还将介绍两种新的引脚配置方法并且简单示意 Quartus 对电路的优化。

● 实验步骤

4.1 建立 Quartus 工程

1. 新建 Quartus 工程 Light，顶层实体名 Light
2. 重新设置编译输出目录为../Light/release。

4.2 使用符号框图描述完成硬件描述设计

3. 选择菜单项 File -> New…，在弹出的对话框中选择 Block Diagram/Schematic File，点击 OK。如图 4-1。

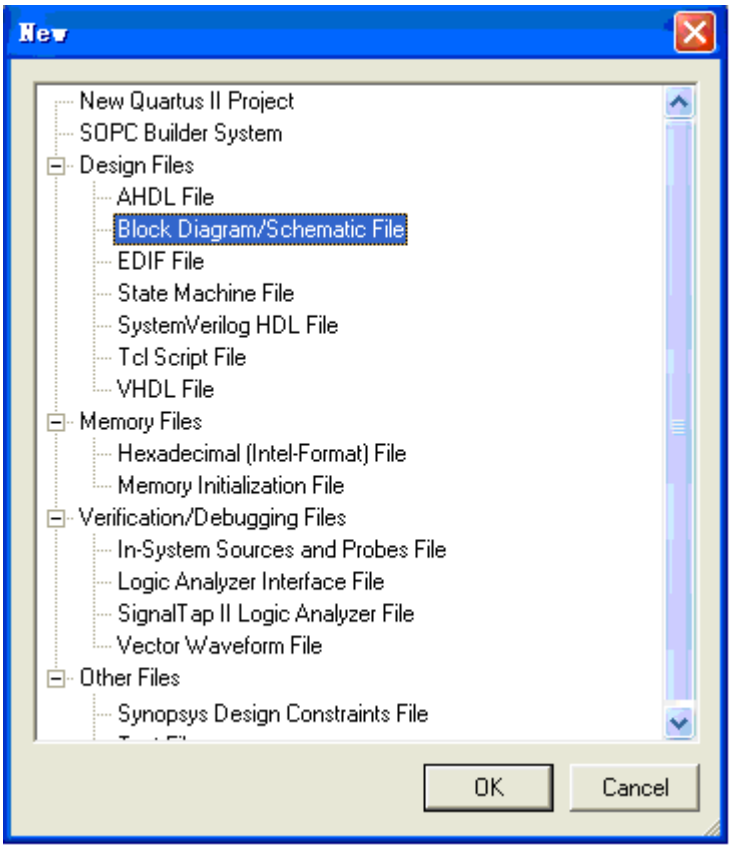


图 4-1　选择设计文件

4. 导入逻辑门电路符号

用鼠标双击图形编辑器窗口的空白处或者短纳期图像编辑器窗口左侧工具条中

图标，选择门电路符号，进行添加。

展开左侧树状目录到…/primitives/logic/and2，找到二输入与门，点击 OK 后在白板

上单击放置即可。参看图 4-2。

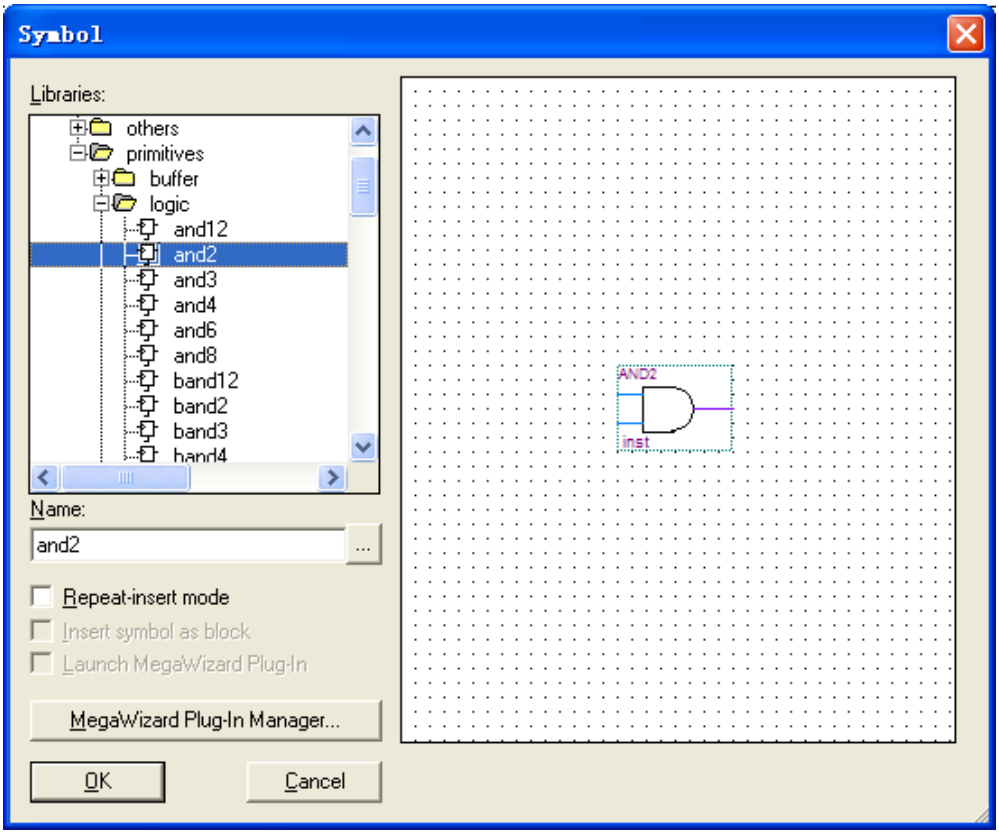


图 4-2　选择二输入与门（and2）

展开左侧树状目录到…/primitives/logic/or2，添加二输入或门。参看图 4-3。

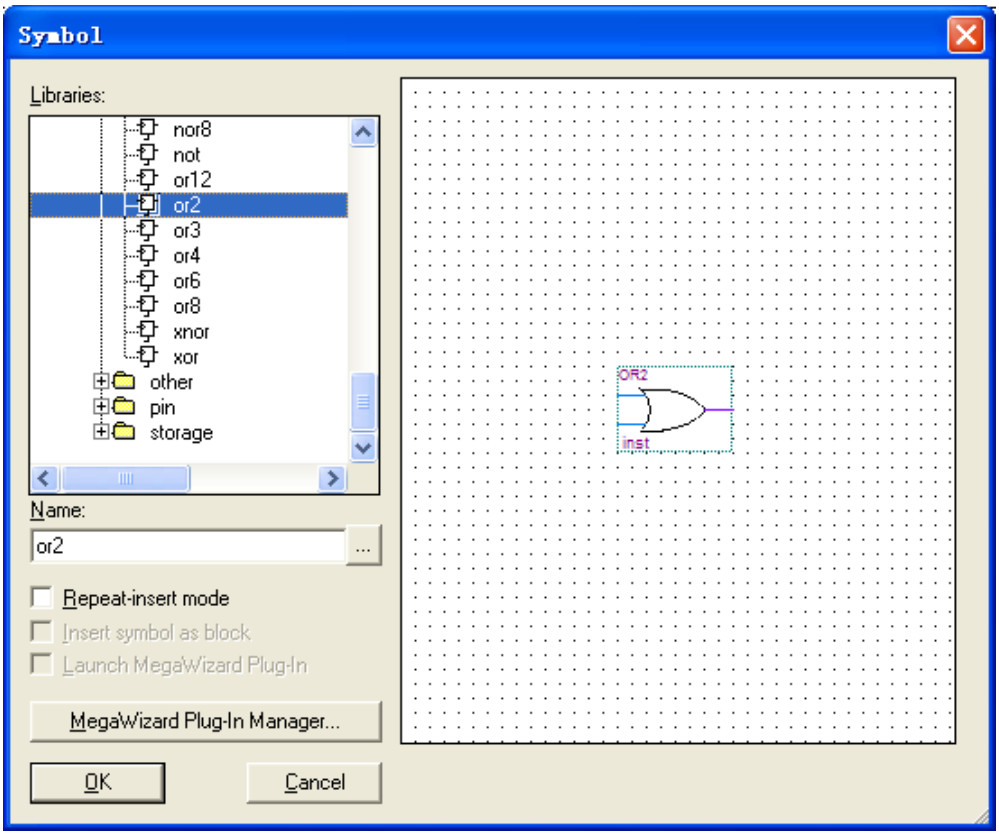


图 4-3　选择二输入或门（OR2）

依上再添加两个取反门，元件整体布局如图 4-4 所示。

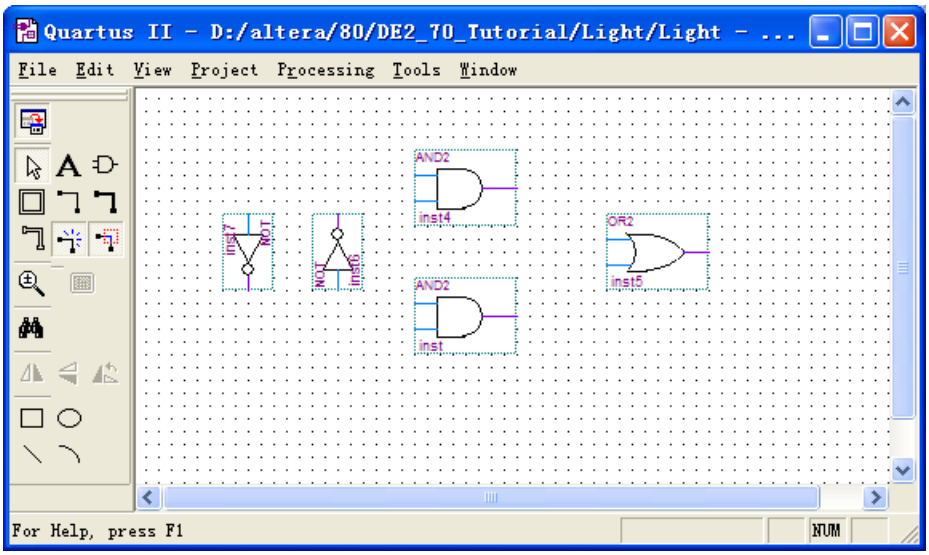


图 4-4　放置门电路符号

5. 放置输入/输出引脚，位置在../primitives/pin/input 与../primitives/pin/output，参看图 4-5 与图 4-6。

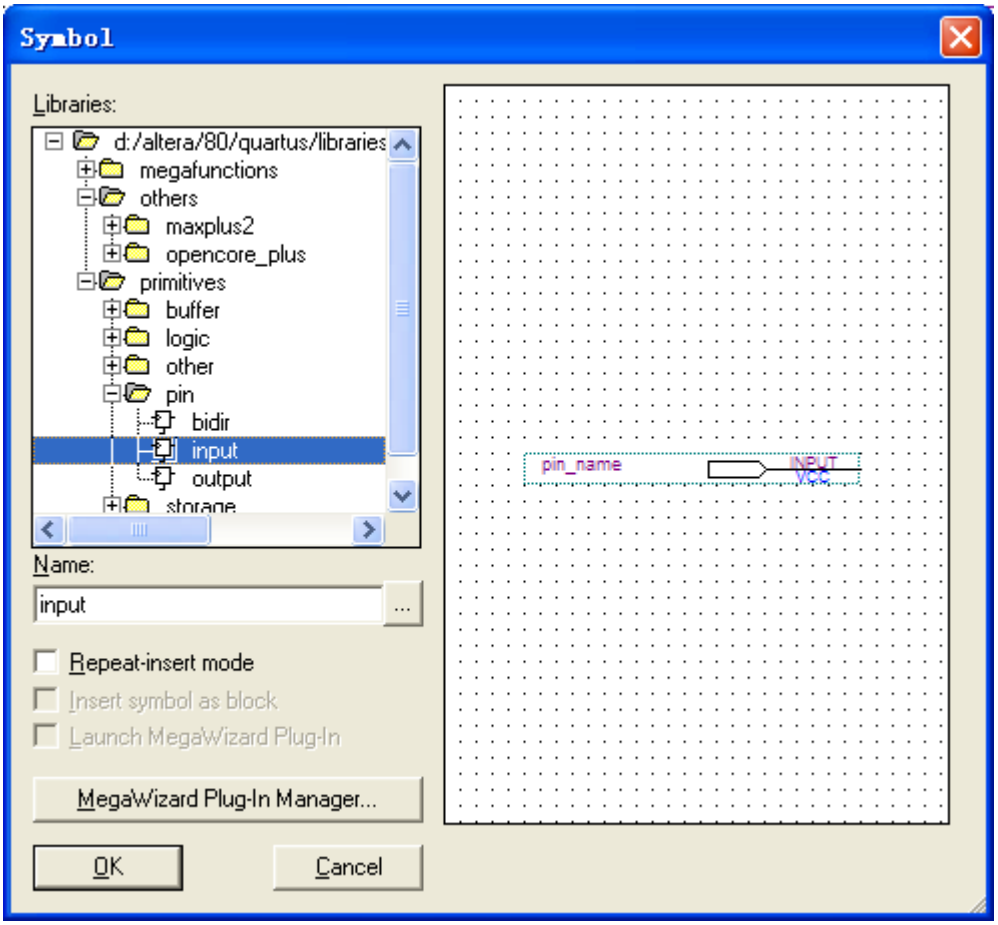


图 4-5　添加输入引脚

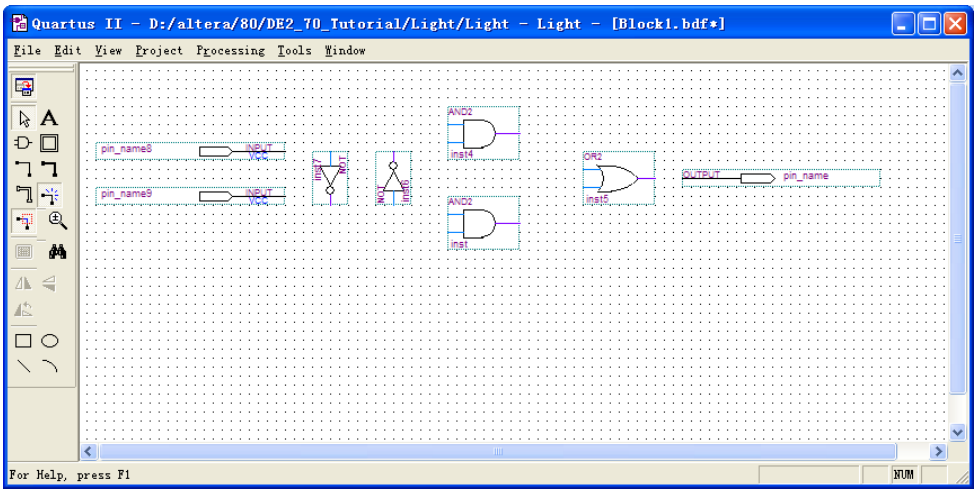


图 4-6 放置输入/输出引脚

6. 电路图连接，用鼠标拖动引线即可连接元件，完成后如图 4-7。

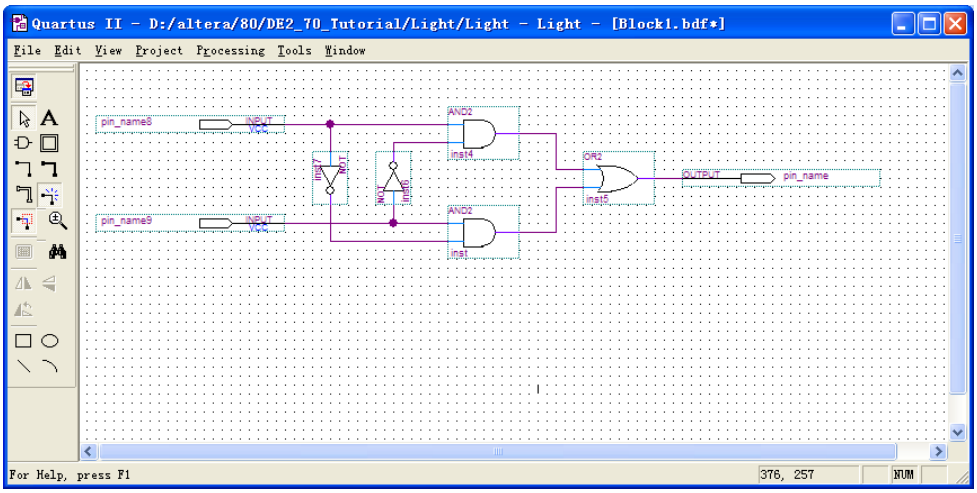


图 4-7　连接电路

7. 修改电路结点的名称，双击元件即可修改，将输出结点名称改为 oLEDG[0]，将两个输入分别改为 iSW[1]和 iSW[0]，并保存设计文件 Light.bdf。如图 4-8。

注意：使用符号框图与使用 Verilog 语言实现完全等价。

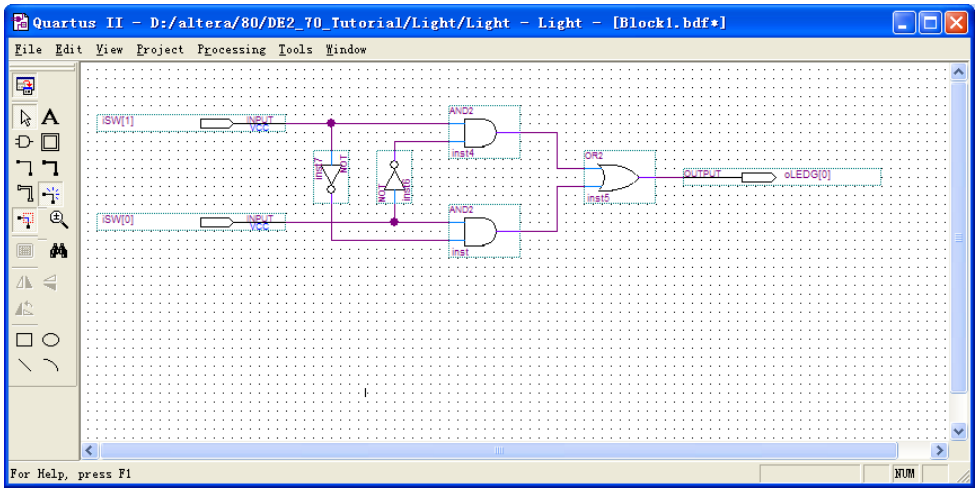


图 4-8　修改引脚名称

8. 分配引脚。前面讲过手工分配引脚，这次使用自动导入。用 Assignments->Import

Assignments…菜单。如图 4-9 所示。

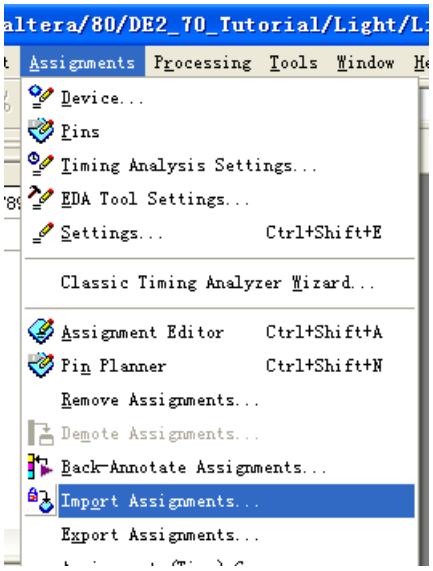


图 4-9　导入引脚配置菜单项

选择文件名，导入 DE2_70_pin_assignments.csv 中的引脚配置。如图 4-10。

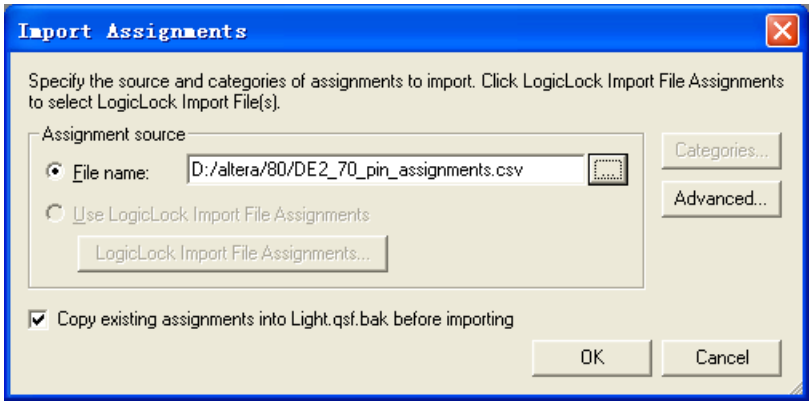


图 4-10　导入引脚配置文件

9. 编译电路，如图 4-11。

图 4-11　编译结果

10. 结果出现 530 个警告，这主要是因为引脚分配文件中含有大量引脚本实验没有用到。如图 4-12。

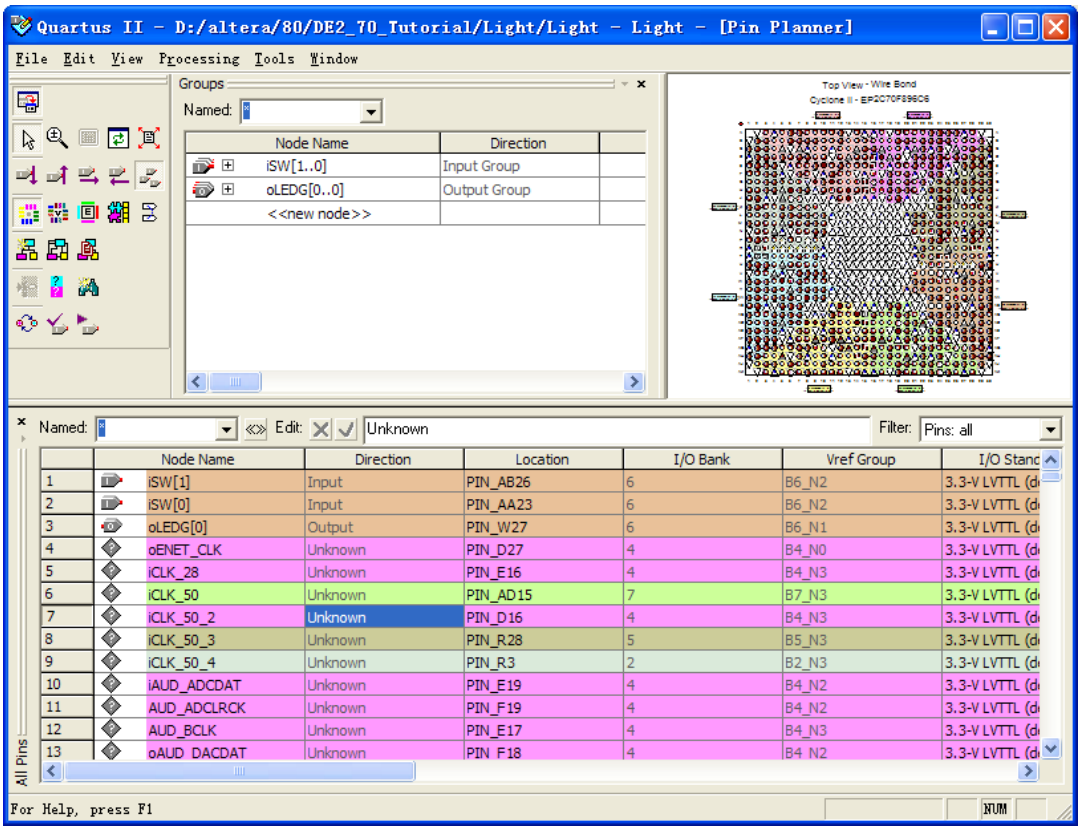


图 4-12　csv 文件自动导入的引脚分配图

11. 删除多余的引脚。用记事本等外部编辑器打开 Light.qsf 文件。发现大量的 set_location_assignment PIN_XXX –to YYYY 信息。将不用的信息删除，留下以下引脚设定。

```
set_location_assignment PIN_AA23 -to iSW[0]
set_location_assignment PIN_AB26 -to iSW[1]
set_location_assignment PIN_W27 -to oLEDG[0]
```

一共三行，保存。回到 Quartus 重新观察引脚。如图 4-13。

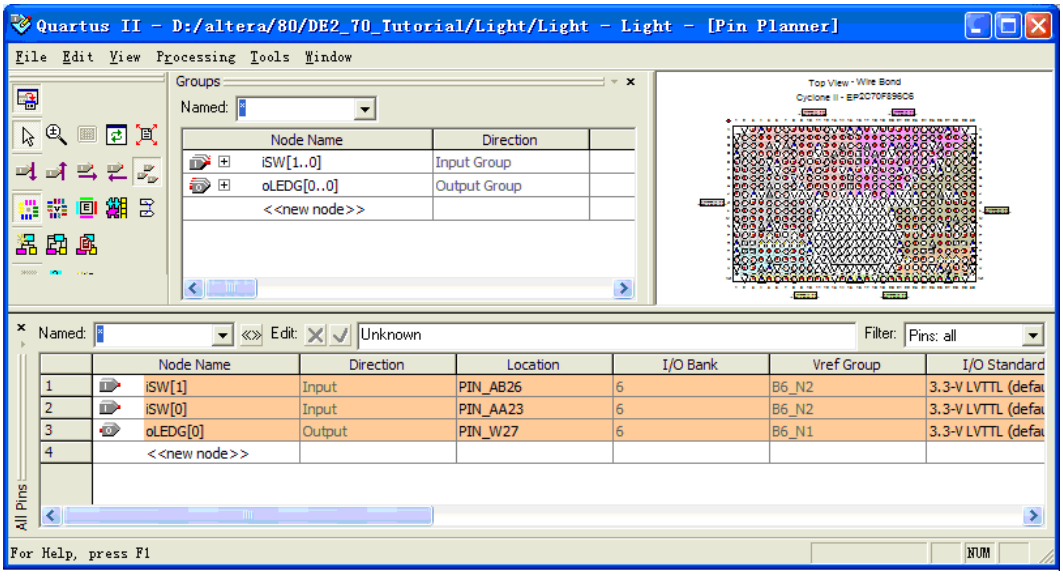


图 4-13　修改 qsf 文件后的引脚分配图

12. 执行全编译，发现警告信息恢复正常。如图 4-14。

图 4-14　编译结果

注意：以后分配引脚，都可使用标准文件名(参考 DE2_70_pin_assignments.csv 文件)加上手工编辑 qsf 文件的方案。

13. 观察电路图，不难发现电路功能仅仅是求一个异或：A^B=A(~B)+(~A)B= A+B-AB

使用一个逻辑单元即可实现，点击菜单项 Tools->Netlist Viewers->Technology Map Viewer，如图 4-15，可以看到只使用了一个逻辑单元，如图 4-16。

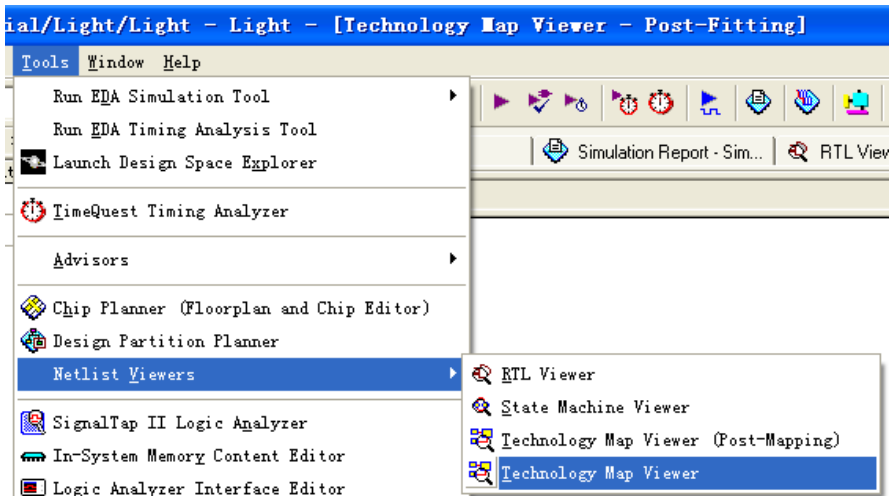


图 4-15　查看 Technology Map 菜单项

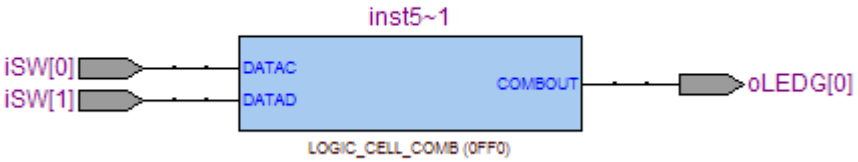


图 4-16　查看 Technology Map

双击进入，可以看到实际只有一个异或门。如图 4-17。这是 Quartus 对电路自动进行的优化。

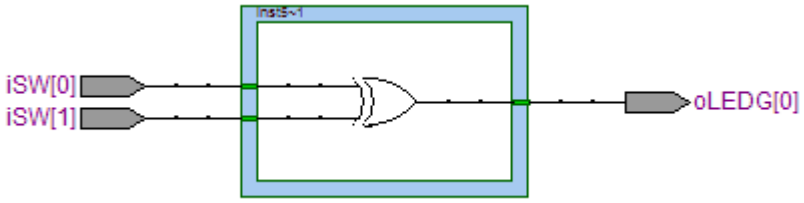


图 4-17　查看 Technology Map 内部

4.3 电路仿真

在 DE2-70 平台上实现该电路之前，我们先在 Quartus II 软件中对电路进行功能仿真，以测试设计的正确性。

14. 点击菜单项 File->New，打开如图所示对话框，并选择 Verification/Debugging Files 中的 Vector Waveform File，单击 OK。

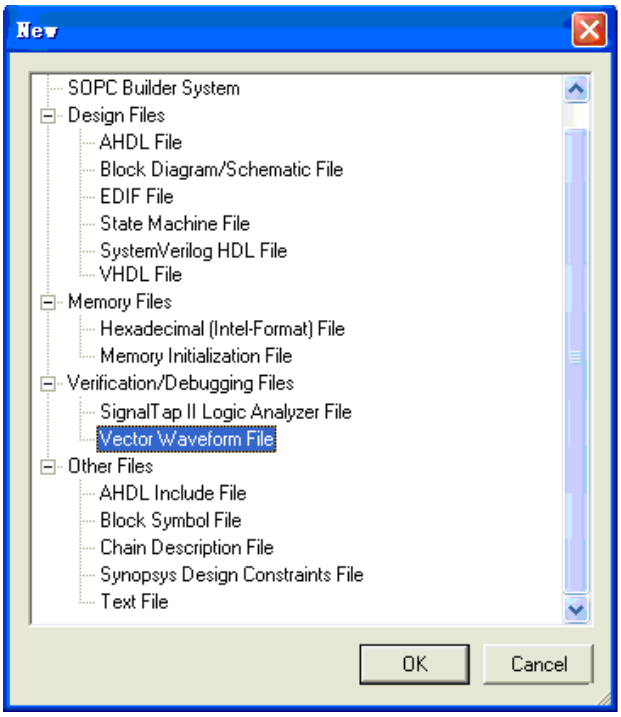


图 4-18　建立矢量波形文件

15. 点击菜单项 Edit->End Time，设定仿真终止时间为 200ns。如图 4-19 与图 4-20。

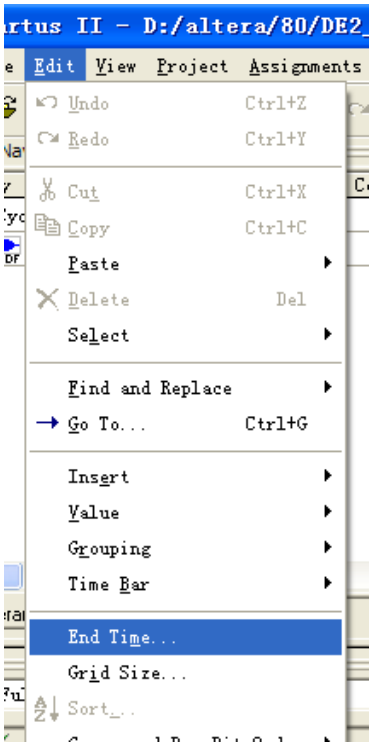


图 4-19 设定仿真结束时间菜单项

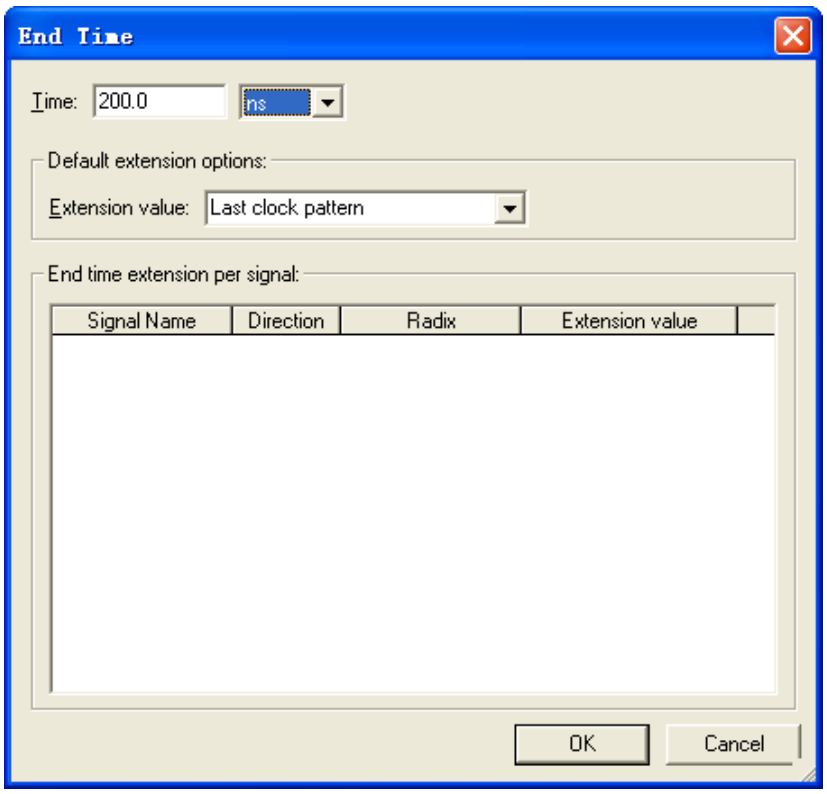


图 4-20　设定仿真结束时间

16. 将要仿真的输入/输出等电路结点加入到波形中来。用 Edit->Insert->Insert Node or Bus 菜单如图所示。如图 4-21。

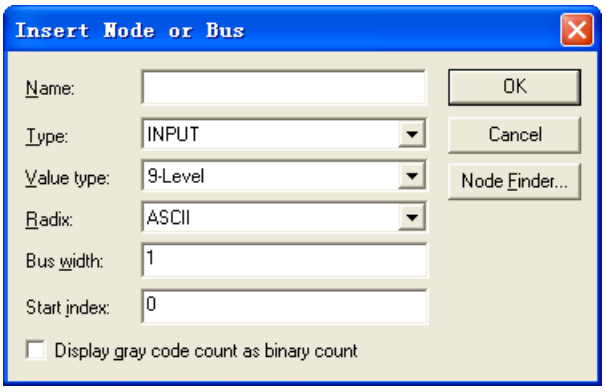


图 4-21　添加结点到波形图

17. 单击 Node Finder..按钮，添加相关的结点 iSW[0]、iSW[1]、oLEDG[0]到波形图。将 iSW[0]设为周期 200ns 的方波，iSW[1]设为周期 100ns 的方波。如图 4-22。

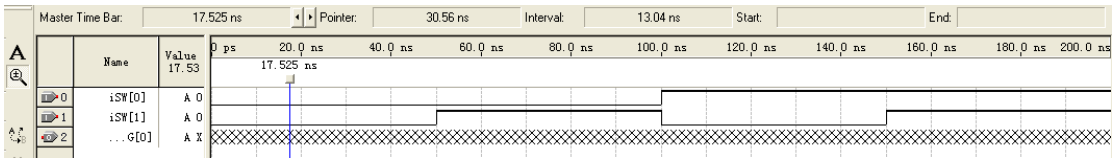


图 4-22　测试用的矢量波形

18. 功能仿真

点击菜单项 Assignment->Settings，选中 Simulator Settings 选项卡，出现图 4-23 所示对话框。在 Simulation mode 中选择 Functional，Simulation input 选择刚才建立的波形文件，完成后点击 OK。

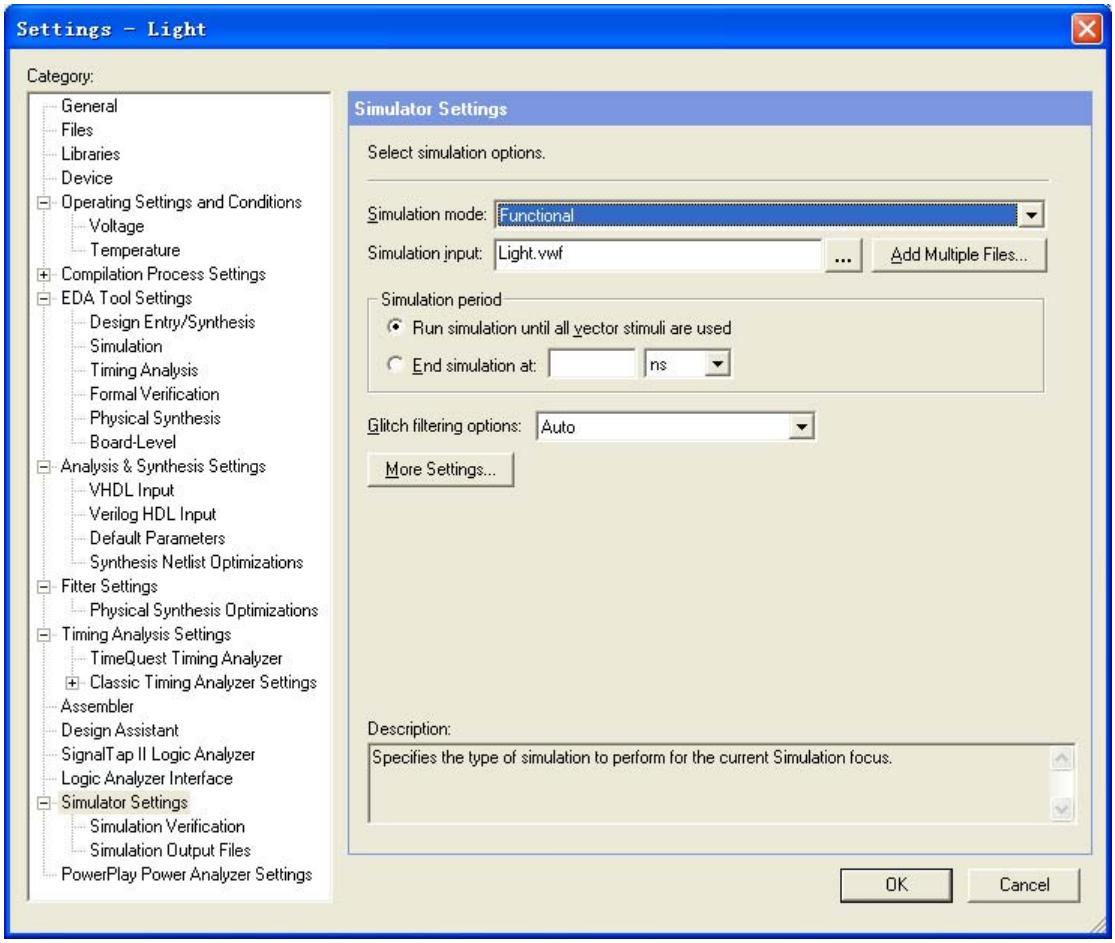


图 4-23　设置仿真模式

点击菜单项 Processing->Generate Functional Simulation Netlist 产生功能仿真所需的网表。点击菜单项 Processing->Start Simulation 或 工具按钮启动功能仿真。结果如图 4-24。

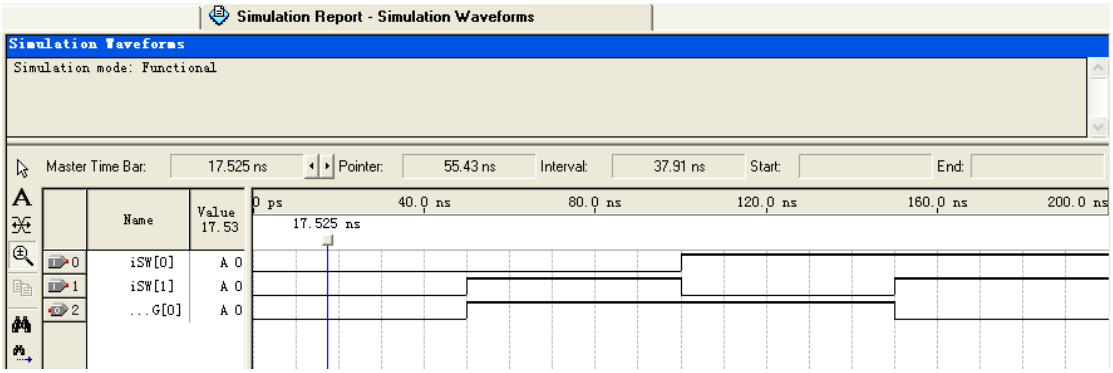


图 4-24　功能仿真结果图

19. 时序仿真

点击菜单项 Assignment->Settings 菜单打开 Settings 窗口，在 Simulation mode 中选择 Timing，Simulation input 选择刚才建立的波形文件，单击 OK。

8.0 版在左侧 Tasks 窗口中 Quartus II Simulator (Timing)处右击 start，执行时序仿真。

7.2 版需要依次执行 Start Analysis&Synthesis、Start Fitter、Start Classic Timing Analyzer、Start TimeQuest Timing Abalyzer、Start Simulation(参看实验二)。

结果如图 4-25 与图 4-26

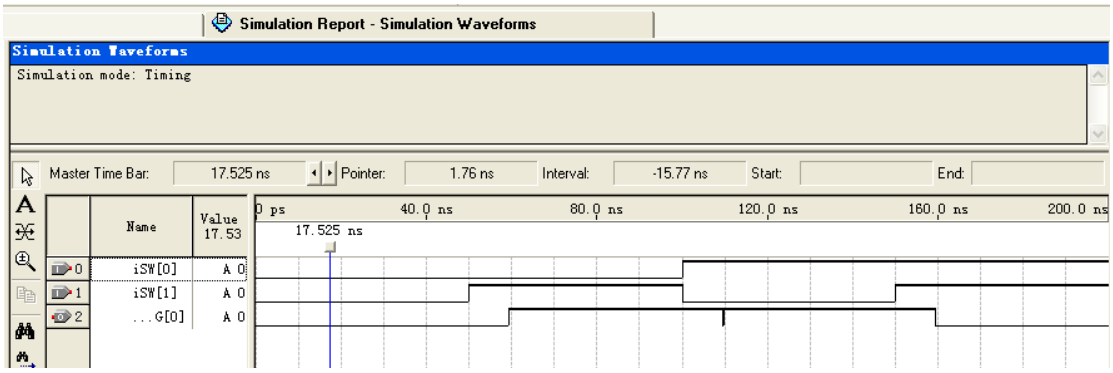


图 4-25　时序仿真结果图

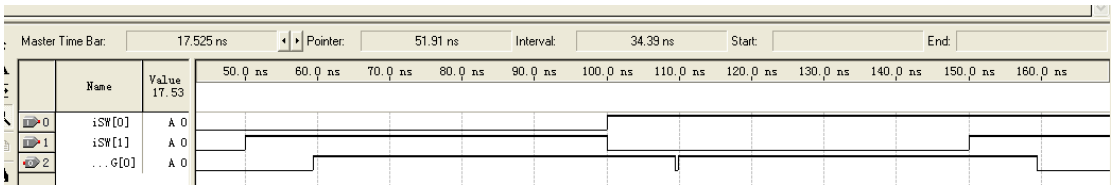


图 4-26　时序仿真结果图放大

注意：图中从 iSW[1]状态变化到输出 oLEDG[0]状态发生相应的变化之间有 9.44-9.45ns 的延时，而从输入 iSW[0]状态变化到输出 oLEDG[0]状态发生相应的变化之间约有 9.84ns 的延时。该延时为 FPGA 布线所决定，不同开发板均不同。

20. 下载到开发板上，拨动开关查看结果，应与异或操作结果一致。

◆ 本实验指导结束

第 5 章　实验四　移位寄存器实验

● 实验说明

Altera 的 Megafunction 是重要的设计输入资源。由于 Megafunction 是基于 Altera 底层硬件结构最合理的成熟应用模块的表现，所以在代码中尽量使用 Megafunction 这类 IP 资源，不但能将设计者从繁琐的代码编写中解脱出来，更重要的是在大多数情况下 Megafunction 的综合和实现结果比用户编写的代码更优。

Megafunction 包括 Altera 的参数化模块库（LPM，library of parameterized modules），器件专有的 Megafunction 模块，用 Altera MegaCore IP 生成工具调用的 IP Core，以及 Altera Megafunction 计划协作者（AMPP, Altera Megafunction Parterners Program）提供的第三方 IP Core。

特别是针对一些与 Altera 器件底层结构相关的特性，必须通过 Megafunction 实现，例如一些存储器模块（DPRAM、SPRAM、FIFO、 CAM 等），DSP 模块，LVDS 驱动器，PLL，高速串行收发器（SERDERS），DDR 输入/输出（DDIO）等。另外一些诸如乘法器、计数器、加法器、滤波器等电路虽然也可以直接用代码描述，然后用通用逻辑资源实现，但是这种描述方法不但费时费力，而且在速度和面积上比不上 Megafunction 的实现结果。

本实验使用两种方法在 DE2-70 平台上设计一个移位寄存器。通过这个实验，读者可以了解使用 Quartus 工具中的 MegaFunction 功能的基本流程。实验中还通过 Verilog 语言实现了同样功能的移位寄存器作为对比。

实验中的 MegaFunction 模块通过符号框图方法使用，这是 Quartus 的另一种设计输入方法。

● 实验步骤

5.1 建立 Quartus 工程

1. 新建 Quartus 工程 ShiftRegCore，顶层实体名 ShiftRegCore
2. 重新设置编译输出目录为../ ShiftRegCore/release。

5.2 使用 MegaFunction+符号框图描述完成硬件描述设计

3. 用 MegaFunction 创建移位寄存器模块，首先点击菜单项 Tools->MegaWizard Plug-In Manager…。如图 5-1。

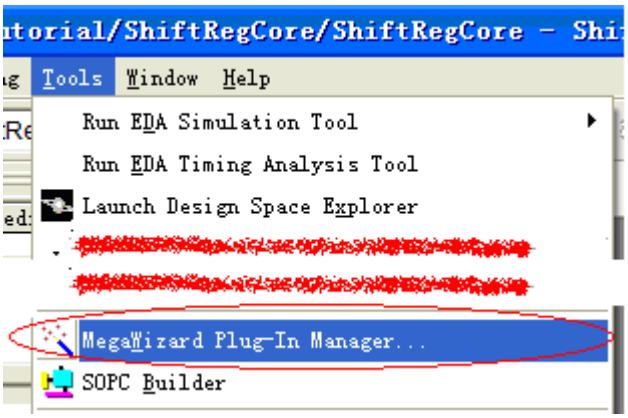


图 5-1　MegaFunction 菜单

打开后的界面如图 5-2 所示。选择其中的第一项，新建一个模块。选好后点 Next。

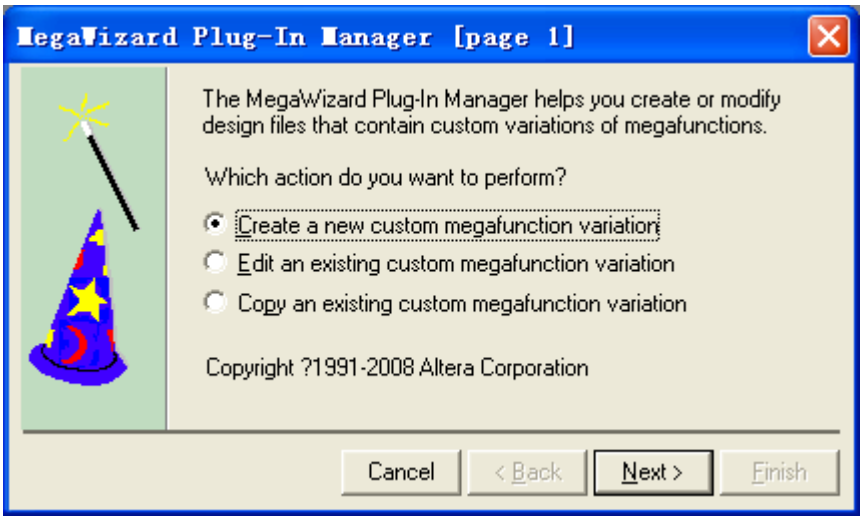


图 5-2　MegaFunction 第一步

接下来出现如图 5-3 所示的界面。如图在左侧选择 Installed Plug-Ins->Storage-> LPM_SHIFTREG。然后在右边的输出文件类型选择 VHDL、Verilog 或者 AHDL。并在右侧给出的输出文件名栏目中，已有的文件路径后面添加输入所要的文件名。本例中输入 ShiftReg。完成后点 Next。

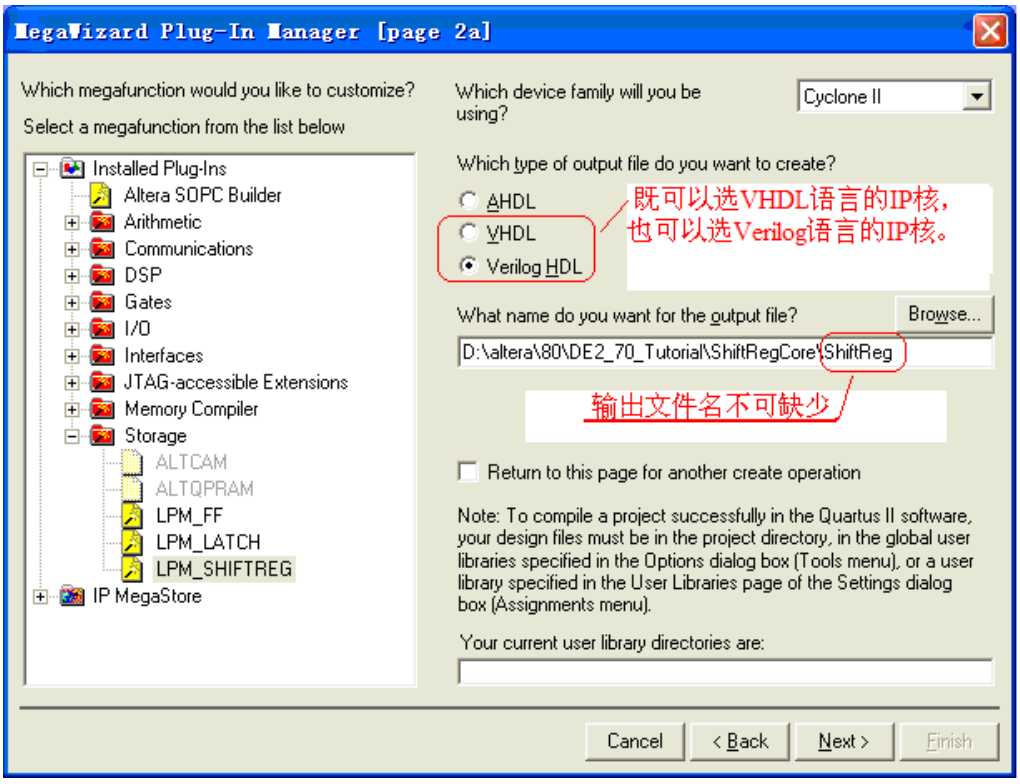


图 5-3 选择所需的 MegaFunction

注意：

在使用 Quartus II 现成的参数化模块以及 IP 核时，可以选择不同语言的实现方案。

完成后出现图 5-4 所示界面。按图选择选项，创建一个带有 load 端、串行输入输出、并行输出功能的右移寄存器。点 Next 进入下一页。

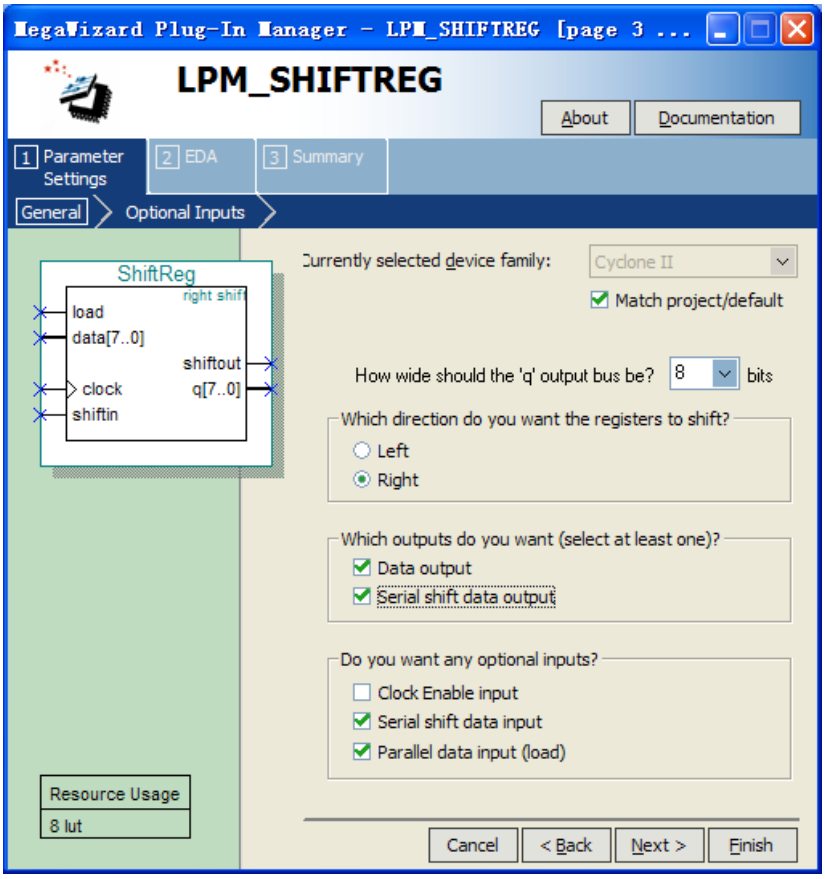


图 5-4　选择移位寄存器功能

完成后如图 5-5 所示。可进一步选择其功能。为移位寄存器增加异步清零端。完成后点击 Next 进入下一页。

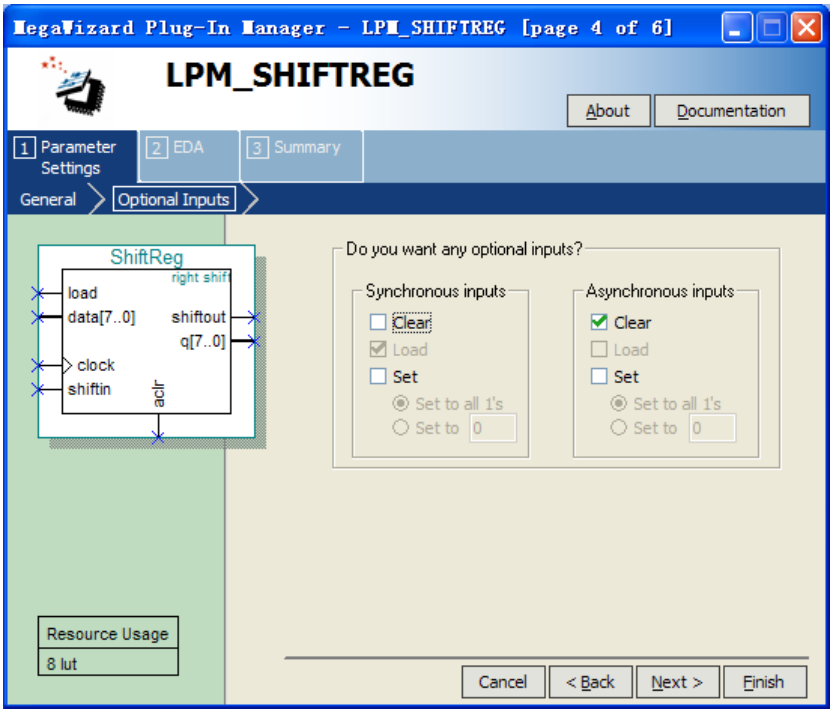


图 5-5 进一步选择

完成后如图 5-6 所示。可以选择生成相应的仿真库。本实验采用默认操作。点 Next 进入下一页。

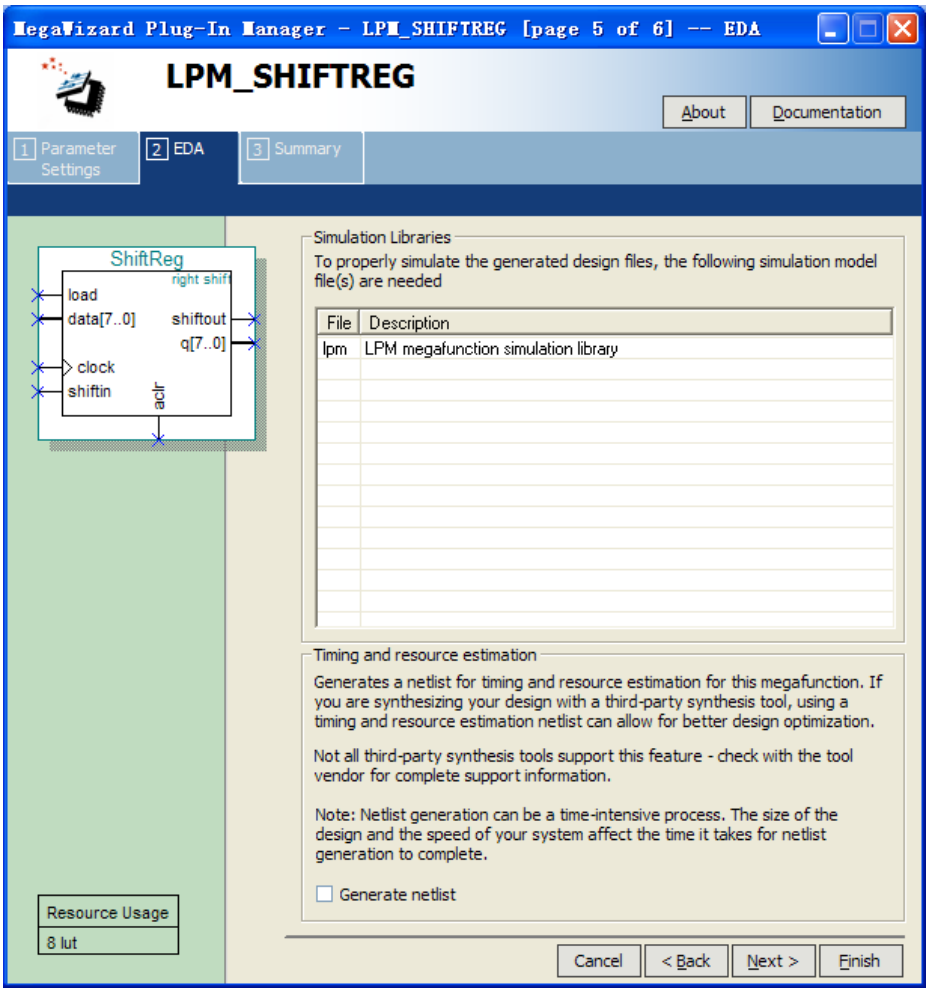


图 5-6 与仿真工具接口

完成后如图 5-7 所示。该页给出了可选的生成文件。注意选择上 ShiftReg.bsf 文件以生成符号文件，为符号框图输入提供源文件。完成此步后直接点 Finish 完成 MegaFunction 设计。完成后如果有对话框问你是否添加此 ip，选否。

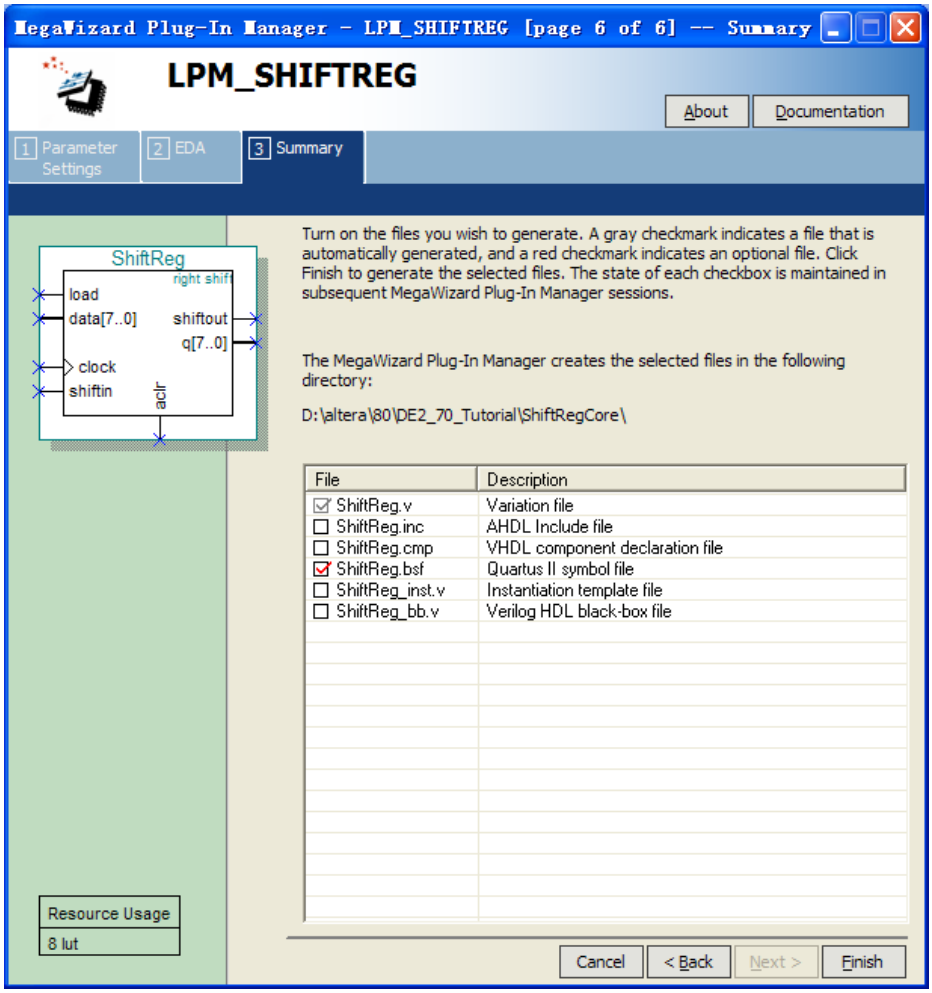


图 5-7 输出文件列表

4. 创建顶层实体描述文件。本次实验使用符号框图来完成这一步。实际上这与使用 Verilog 语言实现顶层实体完全等价。首先点击新建文件，选择其中的 Block Diagram/Schematic File。文件如图 5-8 所示。

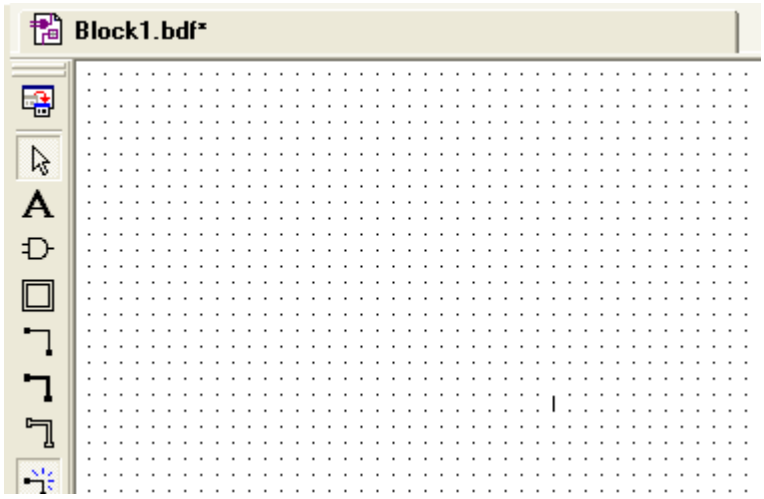


图 5-8 符号框图文件

5. 加入移位寄存器模块。在文件空白处双击进入选择模块对话框。在左侧选择 Project->ShiftReg，即刚才创建的模块，点 OK 可以选择插入点，在适当位置点击左键插入。插入后如图 5-9、5-10 所示。

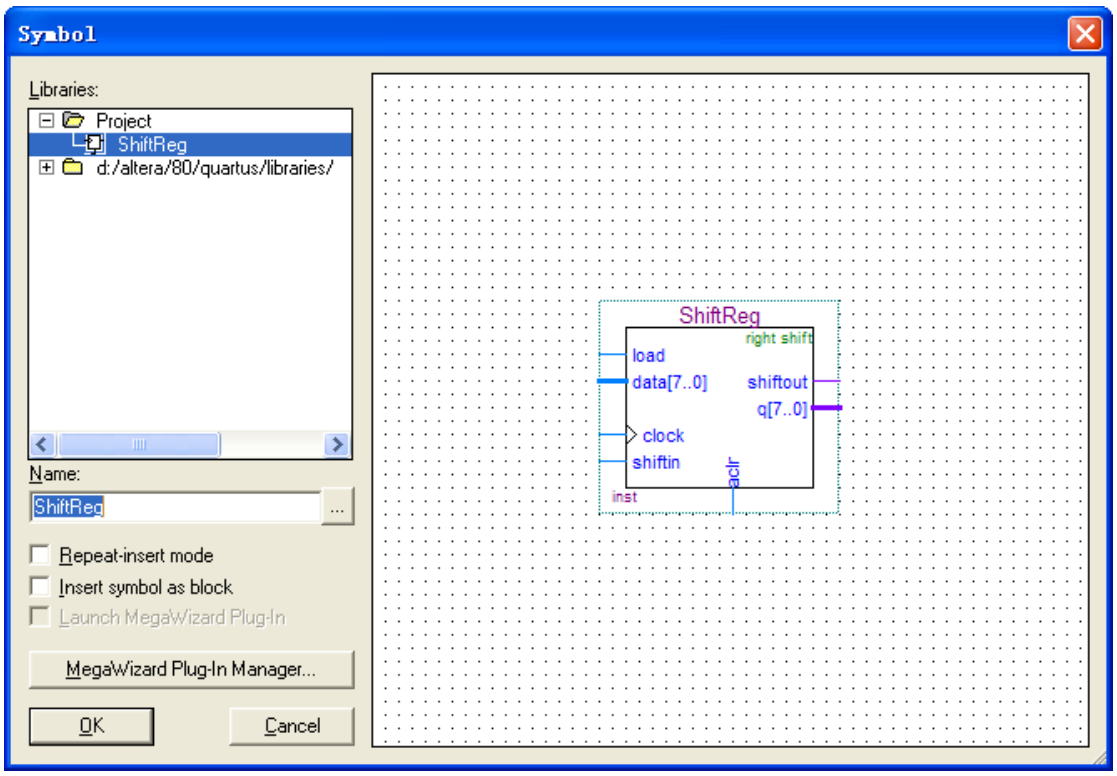


图 5-9 选择模块对话框

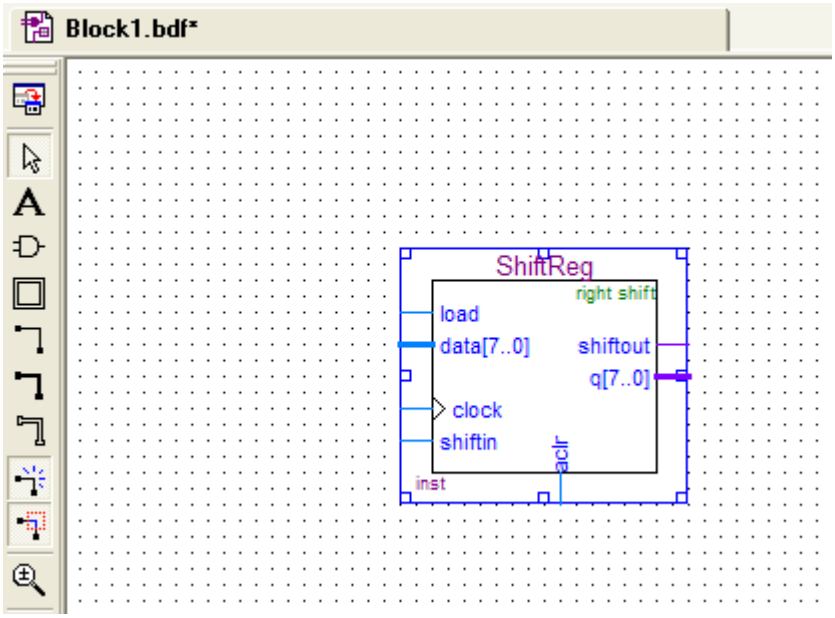


图 5-10 插入移位寄存器

6. 加入输入输出引脚。依照上一步，选择输入输出引脚，插入到符号框图中。如图 5-11 所示。

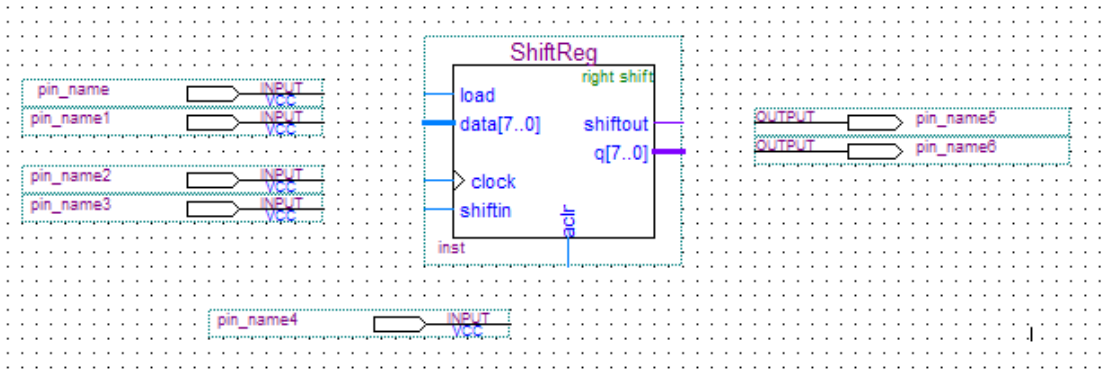


图 5-11 放置输入输出引脚

连接并修改引脚名称。注意数组的写法 iSW[7..0]，这与硬件描述语言中的 iSW[7:0]不同。如图 5-12。

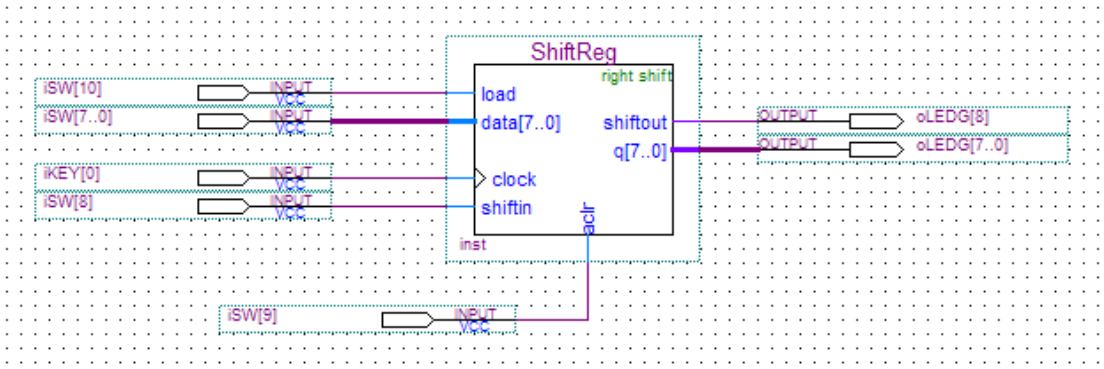


图 5-12 连接并修改引脚名称

完成后保存该文件。

7. 分析与综合，这一步除了检查电路图是否出错外，尚有另一个目的，就是为了将已有配置信息写入 qsf 文件。以免分配引脚时造成 qsf 文件的修改时间比内存更新，丢失以往配置信息(如编译输出路径)。

8. 配置引脚。直接编译 qsf 文件添加引脚。（参见实验三）结果如图 5-13。

```
set_location_assignment PIN_AA23 -to iSW[0]
set_location_assignment PIN_AB26 -to iSW[1]
set_location_assignment PIN_AB25 -to iSW[2]
set_location_assignment PIN_AC27 -to iSW[3]
set_location_assignment PIN_AC26 -to iSW[4]
set_location_assignment PIN_AC24 -to iSW[5]
set_location_assignment PIN_AC23 -to iSW[6]
set_location_assignment PIN_AD25 -to iSW[7]
set_location_assignment PIN_AD24 -to iSW[8]
set_location_assignment PIN_AE27 -to iSW[9]
set_location_assignment PIN_W5 -to iSW[10]
set_location_assignment PIN_W27 -to oLEDG[0]
set_location_assignment PIN_W25 -to oLEDG[1]
set_location_assignment PIN_W23 -to oLEDG[2]
set_location_assignment PIN_Y27 -to oLEDG[3]
set_location_assignment PIN_Y24 -to oLEDG[4]
```

set_location_assignment PIN_Y23 -to oLEDG[5]
set_location_assignment PIN_AA27 -to oLEDG[6]
set_location_assignment PIN_AA24 -to oLEDG[7]
set_location_assignment PIN_T29 -to iKEY[0]
set_location_assignment PIN_AC14 -to oLEDG[8]

Named: Edit:

	Node Name	Direction	Location	I/O Bank	Vr
1	iKEY[0]	Input	PIN_T29	6	B6_N0
2	iSW[10]	Input	PIN_W5	1	B1_N1
3	iSW[9]	Input	PIN_AE27	6	B6_N2
4	iSW[8]	Input	PIN_AD24	6	B6_N3
5	iSW[7]	Input	PIN_AD25	6	B6_N3
6	iSW[6]	Input	PIN_AC23	6	B6_N3
7	iSW[5]	Input	PIN_AC24	6	B6_N3
8	iSW[4]	Input	PIN_AC26	6	B6_N3
9	iSW[3]	Input	PIN_AC27	6	B6_N2
10	iSW[2]	Input	PIN_AB25	6	B6_N2
11	iSW[1]	Input	PIN_AB26	6	B6_N2
12	iSW[0]	Input	PIN_AA23	6	B6_N2
13	oLEDG[8]	Output	PIN_AC14	8	B8_N0
14	oLEDG[7]	Output	PIN_AA24	6	B6_N2
15	oLEDG[6]	Output	PIN_AA27	6	B6_N1
16	oLEDG[5]	Output	PIN_Y23	6	B6_N2
17	oLEDG[4]	Output	PIN_Y24	6	B6_N2
18	oLEDG[3]	Output	PIN_Y27	6	B6_N1
19	oLEDG[2]	Output	PIN_W23	6	B6_N1
20	oLEDG[1]	Output	PIN_W25	6	B6_N1
21	oLEDG[0]	Output	PIN_W27	6	B6_N1

图 5-13　分配引脚

9.　编译

一般情况下你会遇到 Quartus II 与 DE2-70 的一个重大兼容性 bug，如图 5-14。

```
Info: Device migration not selected. If you intend to use device migration la
Info: Fitter converted 3 user pins into dedicated programming pins
Error: Can't place pins assigned to pin location Pin_AD25 (IOC_X95_Y2_N1)
Info: Fitter preparation operations ending: elapsed time is 00:00:00
Error: Can't fit design in device
Error: Quartus II Fitter was unsuccessful. 2 errors, 0 warnings
Error: Quartus II Full Compilation was unsuccessful. 4 errors, 2 warnings
```

图 5-14　Quartus II 引脚无法分配的问题

出现的条件似乎是使用了包含 iSW[7]的多数开关，也就是 AD25 对应的引脚。（即使没有遇到，也请看一下，只要是大量使用开关引脚，将来一定会遇到。）

注意：遇到这个问题的解决方法如下。

点击菜单项 Assignments->device…，在对话框里单击“Device and Pin Options”按钮，如图 5-15。

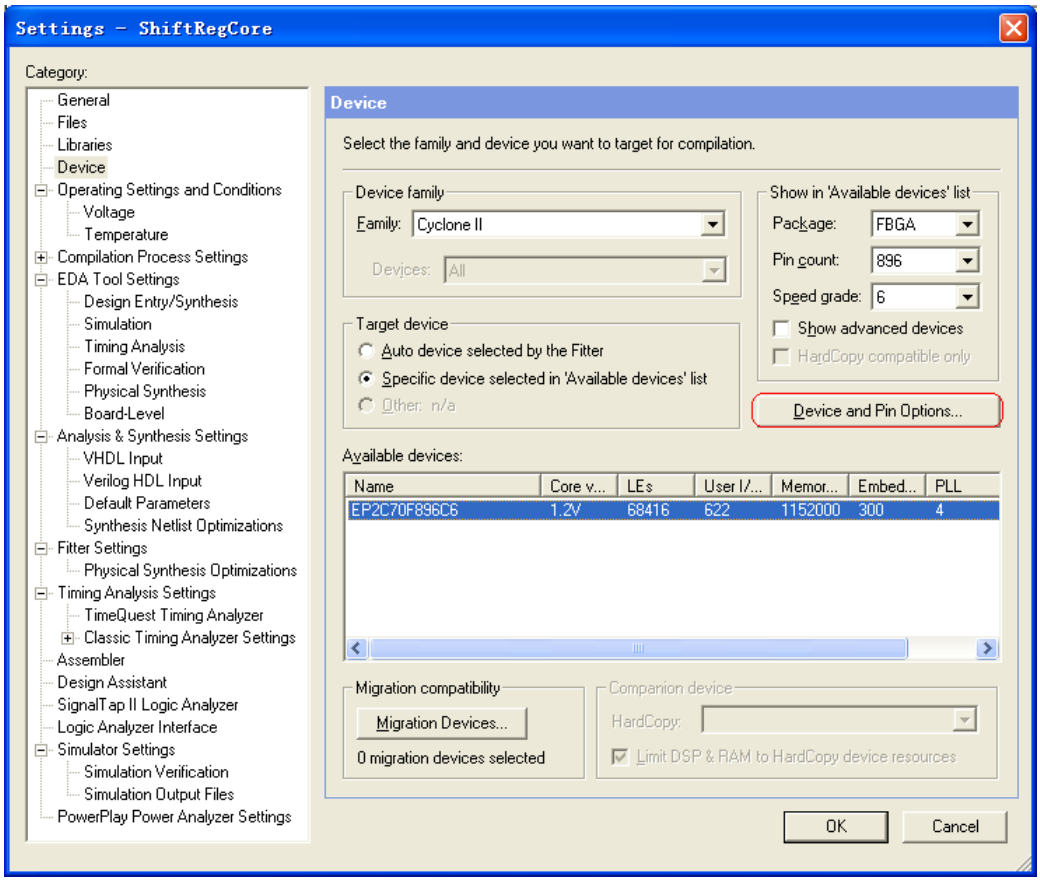


图 5-15　Device 设置对话框

选择 Dual-Purpose Pins 标签页，然后双击 nCEO 那一行的 Value 栏目，系统立即在该栏目显示一个下拉列表，请选择下拉列表的 Used as regular I/O 项目，如图 5-16。

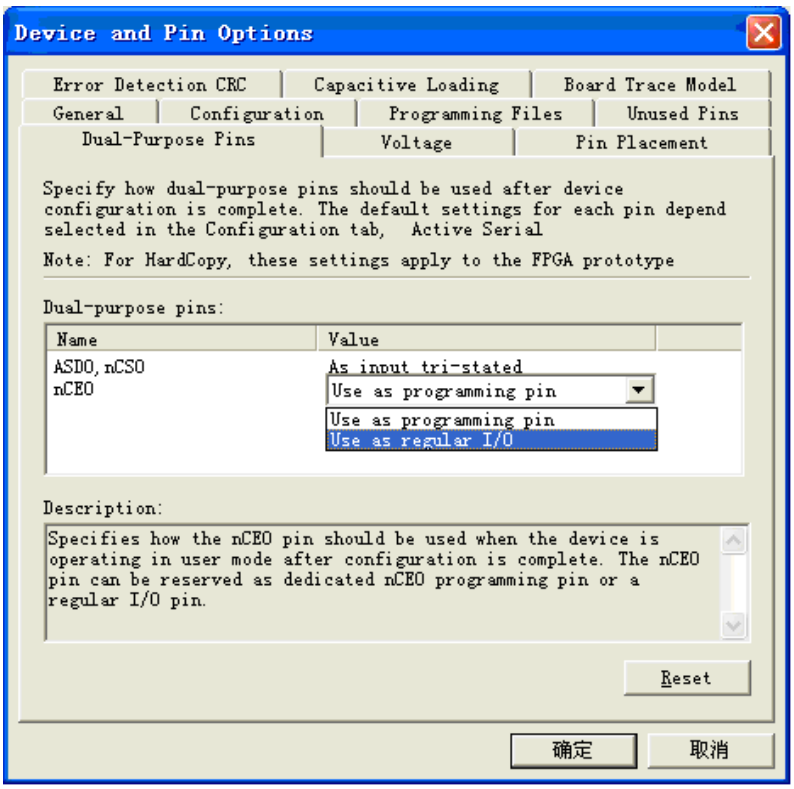


图 5-16　Dual-Purpose 引脚设置对话框

10. 重新编译，错误已经消失。

11. 下载运行。

运行操作非常繁琐，故特别说明：首先将低八位开关定为你想要的数据初始值，比如01101101，iSW[8](shiftin)置 1，然后将 iSW[10](load)拨上，按 KEY[0]载入数据，再将iSW[10](load)拨下，按 KEY[0]数据右移一位。反复按，数据不断右移。iSW[9](aclr)按钮是重置按键。oLEDG[8](shiftout)与数据最右边一位是始终一致的。

下面是一个实验原理图的说明：

图 5-17 是一个信号逻辑概略图，其中不包括功能切换信号 iSW[10]。如果想要了解详细的逻辑信号布局，请使用菜单项 Tools->Netlist Viewers->Technology Map Viewer(Post Mapping)，如图 5-18。

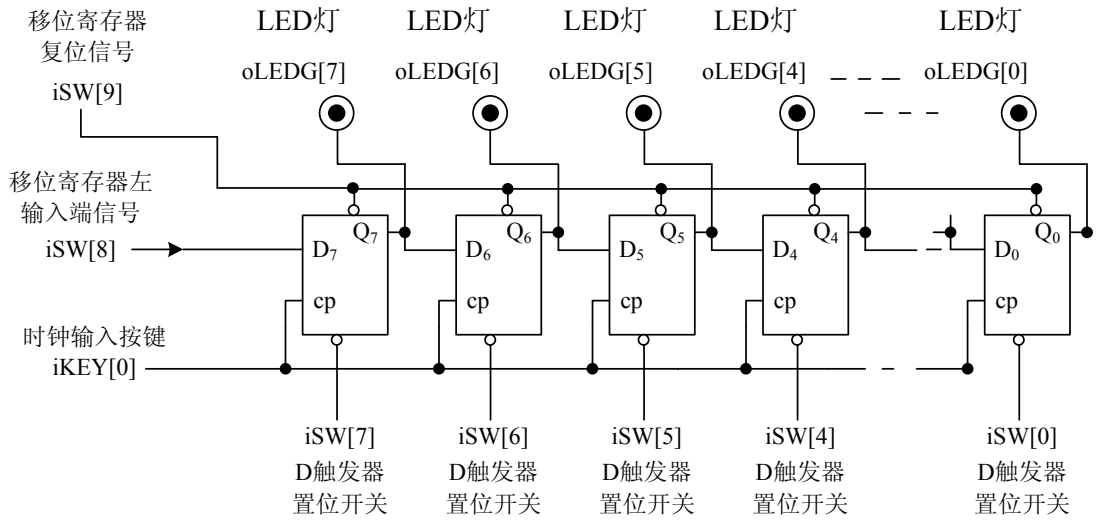


图 5-17　移位寄存器实验的信号逻辑概略图

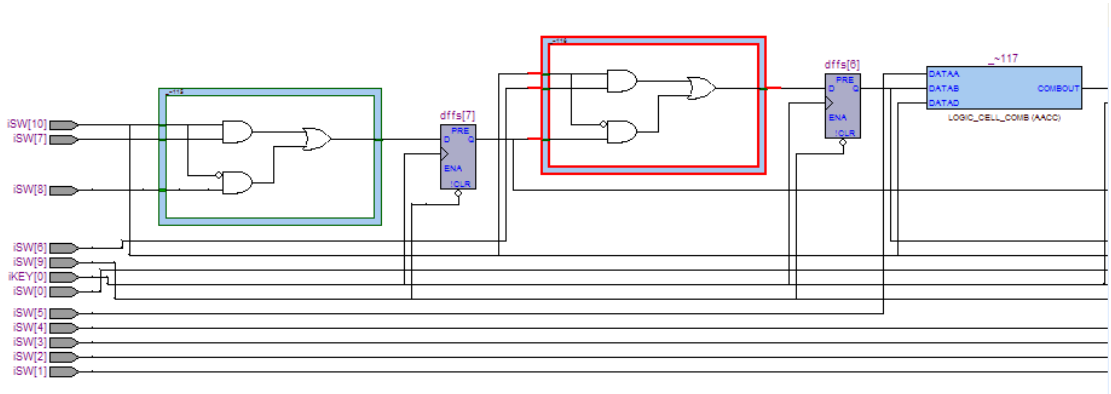


图 5-18　Technology Map Viewer(Post Mapping)（展开了左侧两个逻辑单元）

实验执行参考图 5-19、5-20 与图 5-21。

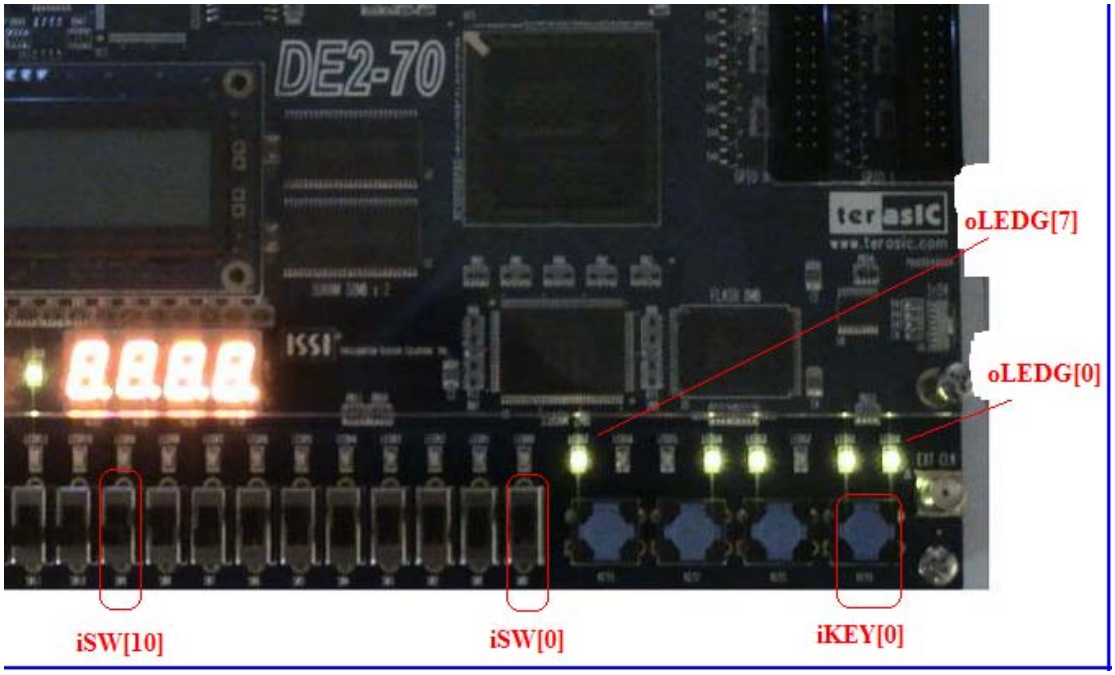


图 5-19　移位寄存器实验的运行指导图

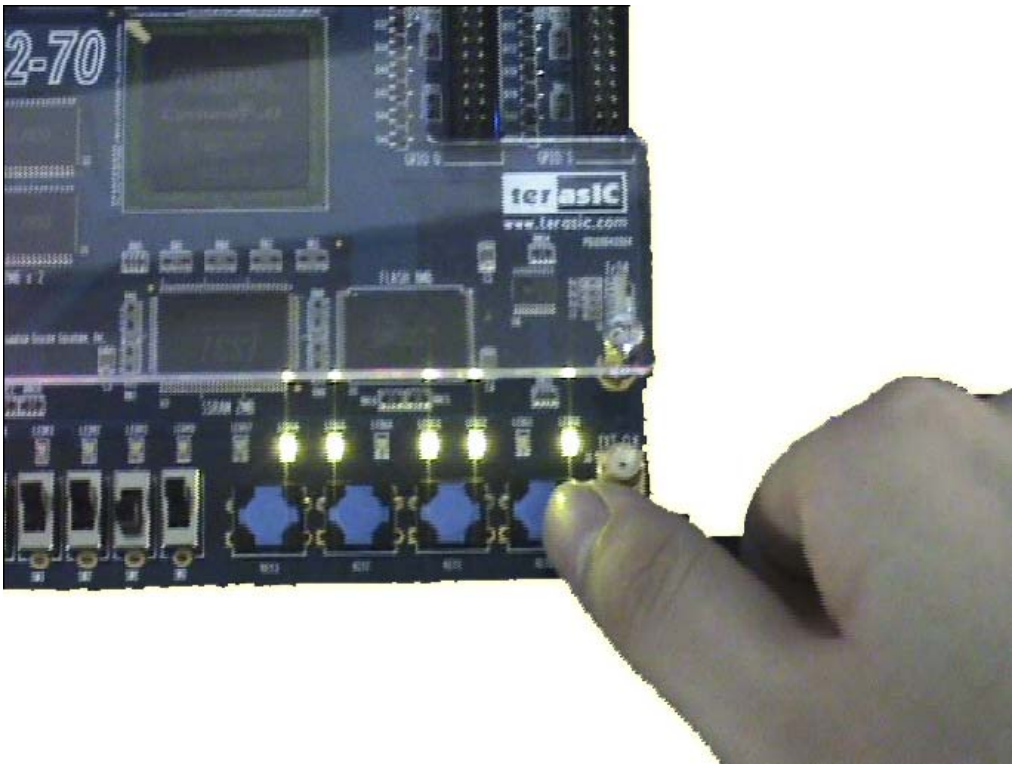


图 5-20　移位寄存器实验的运行效果图

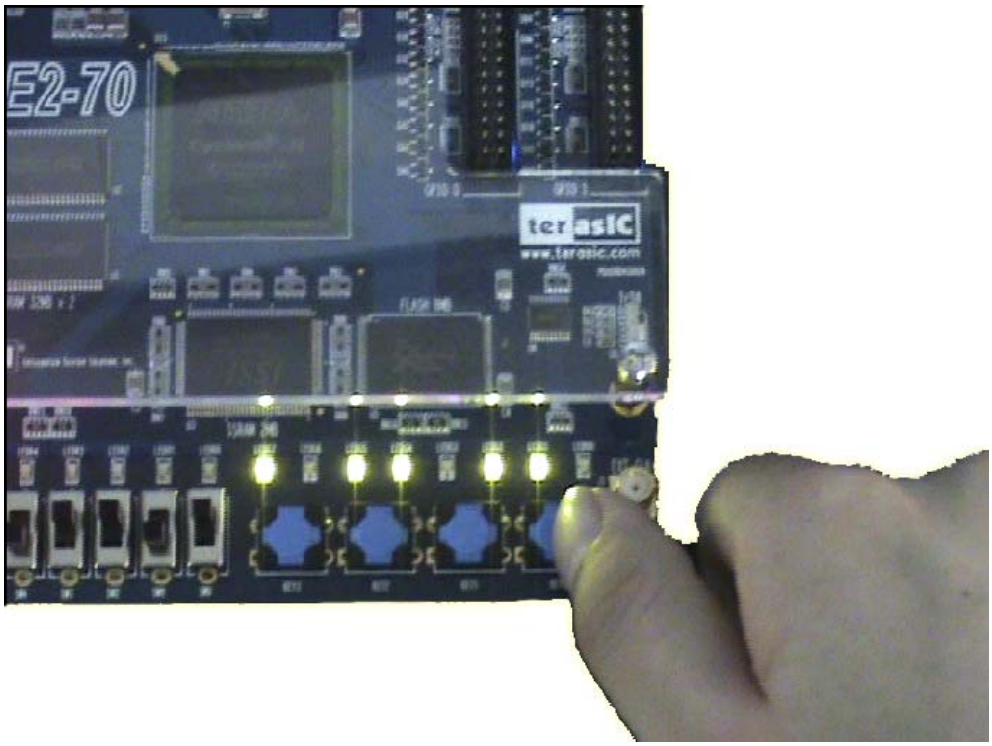


图 5-21　移位寄存器实验的运行效果图（右移一位，高位移入 1）

5.3 使用 Verilog 语言完成硬件描述设计

12. 下面通过 Verilog 代码实现同样功能的设计。

在同一个工程下新建一个 Verilog 源文件，输入如下代码，并保存为 ShiftRegCore_2.v。

```
module ShiftRegCore_2(
    aclr，
    clock，
    data，
    load，
    shiftin，
    q，
    shiftout);

    input        aclr;
    input        clock;
    input    [7:0]   data;
    input        load;
    input        shiftin;
    output   reg [7:0]   q;
    output       shiftout;
always@(posedge clock or posedge aclr)
begin
if(aclr)q = 8'h0;
else
    begin
```

```
    if(load)q = data;
    else
    q = {shiftin，q[7:1]};
    end
end

assign shiftout = q[0];

endmodule
```

13. 在左侧 Project Navigator 处右击 ShiftReg_2.v 选择 Create Symbol Files for Current File。之后就可以像 ShiftReg 一样在顶层实体的符号框图里使用了，如图 5-22 与图 5-23。不过本实验不使用这种方法。

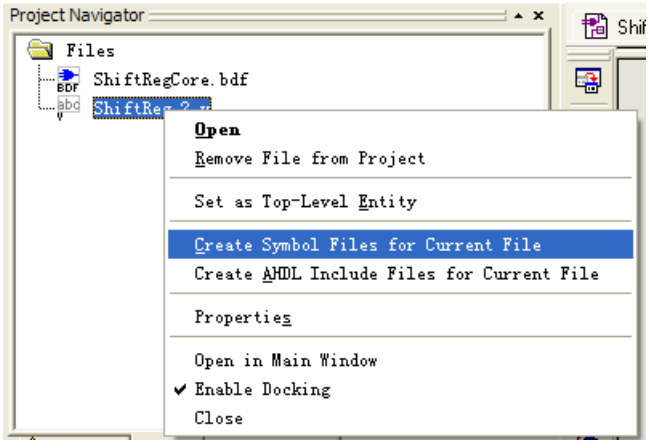


图 5-22　创建符号文件

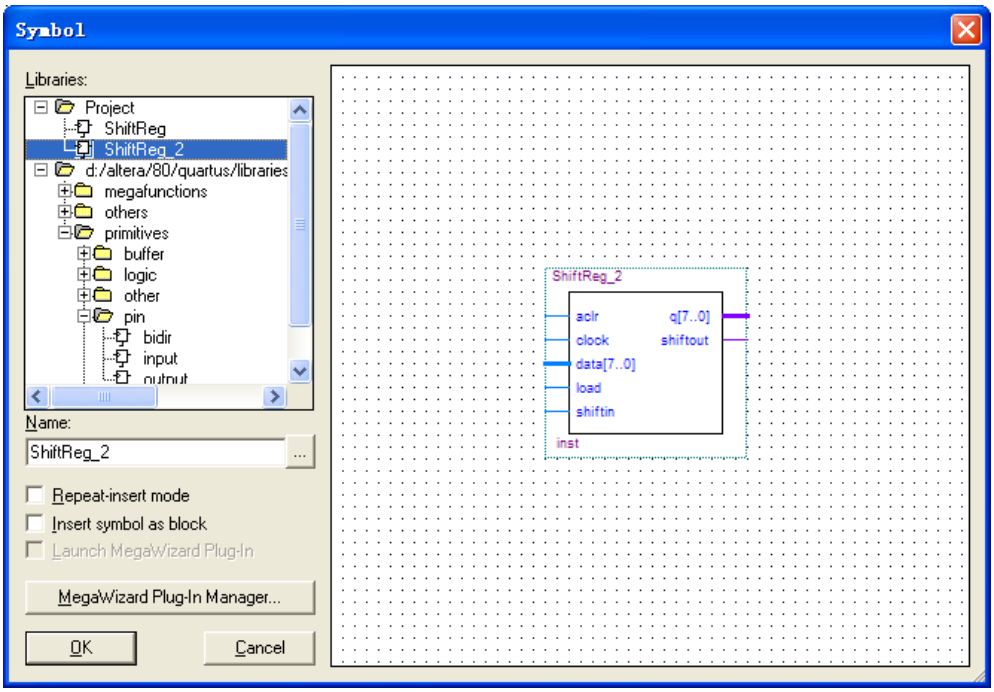


图 5-23　用 Verilog 语言创立的移位寄存器

14. 使用 Verilog 语言代替符号框图完成顶层实体。

新建一个 Verilog 文件添加以下代码，并设置其为顶层实体：

```
module ShiftRegCore_2
(
input [10:0] iSW，
input [0:0] iKEY，
output [8:0] oLEDG
);

ShiftReg_2 (
      .aclr(iSW[9])，
      .clock(iKEY[0])，
      .data(iSW[7:0])，
      .load(iSW[10])，
      .shiftin(iSW[8])，
      .q(oLEDG[7:0])，
      .shiftout(oLEDG[8])
      );
endmodule
```

15. 由于引脚名与符号框图一致，无须重新分配引脚，直接编译下载运行，结果应与使用符号文件设计的结果一致。
16. Tools->Netlist Viewers 观察，可以发现二者在 RTL Viewer 中的结构不同，但在 Technology Map Viewer (post-mapping)中的结构是一致的。

◆ 本实验指导结束

第 6 章　实验五　LCD 显示实验

● 实验说明

该实验在 LCD 上滚动显示“Hello from Nios II”。实验简单介绍了 SOPC Builder 与 Nios II IDE 的使用。通过该实验，使读者大致了解 SOPC 的建立过程，明白如何在 DE2-70 上跑简单的 C 程序。

● 实验步骤

6.1 建立 Quartus 工程

1. 新建 Quartus 工程 HelloWorld，顶层实体名 HelloWorld
2. 重新设置编译输出目录为../ HelloWorld/release。

6.2 建立 SOPC 系统

3. 点击 Quartus II 软件右上方图标打开 SOPC Builder，创建一个 SOPC 系统。填写系统名称为 HelloWorld_System，并指定 Verilog 为描述系统的语言（注意：在添加系统名称时不可以与工程名相同）如图 6-1。

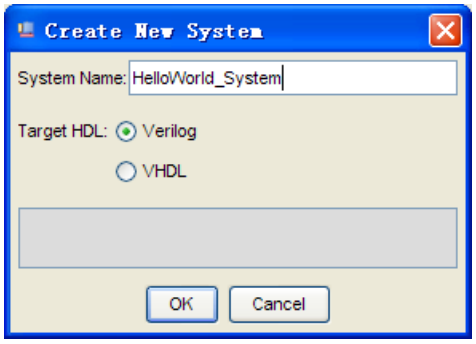


图 6-1　添加系统名称并指定语言

4. 在系统上添加 On-Chip Memory。
5. 在程序左侧列表中选择 Memory and Memory Controllers -> On-Chip -> On-Chip Memory (RAM or ROM)，双击添加至系统中。如图 6-2 所示。

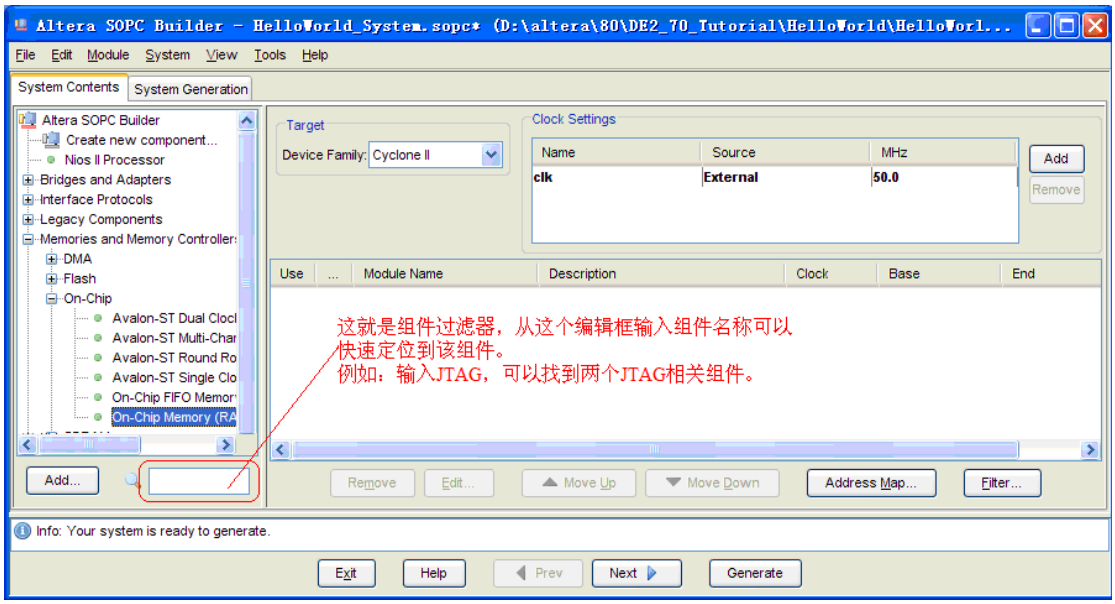
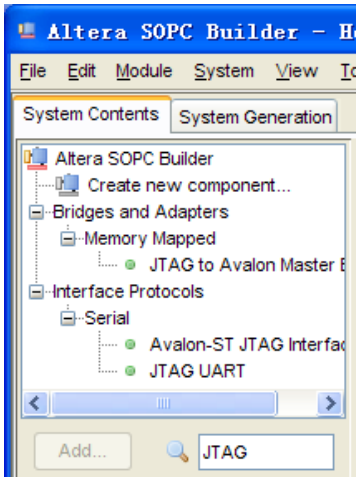


图 6-2 添加 On-Chip Memory

注意：左侧的组件均可以根据需要添加，8.0 以上的 SOPC Builder 具有过滤器，可以通过输入组件的名称进行快速定位。参看左边的 SOPC 配置过滤器位置图。



SOPC 配置过滤器位置图

在弹出的对话框中指定片上 RAM 的属性。**主要是片上存储器的大小问题，太小的情况下甚至无法编译 HelloWorld。**对于 Quartus II 8.0 自带的 HelloWorld 模板编译结果官方给出的大小是~69K。如果用编译器参数 -Os 优化，最终可以优化到 22K，这里把 On-Chip Memory 指定到 30K，如图 6-3，配置好后点击 Finish.。

注意：如果 Nios 实验过程中实在执着于空间限制，请在 C 语言编程时使用 alt 库。

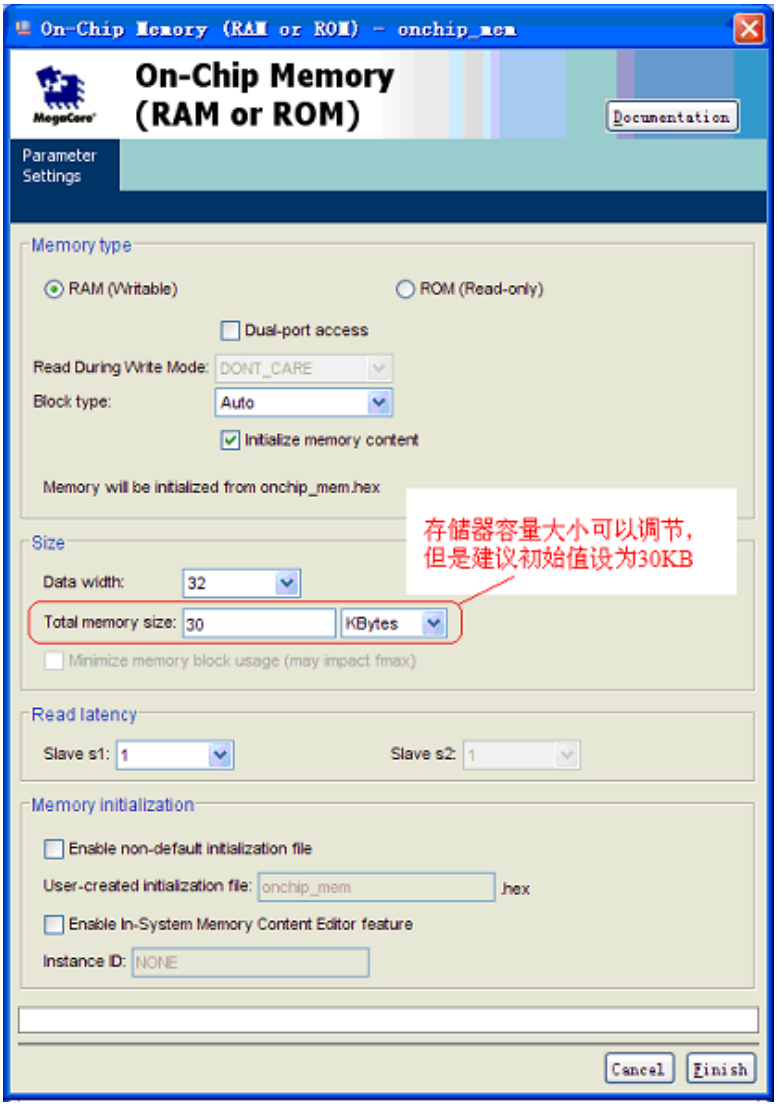


图 6-3 指定 On-Chip Memory 属性

6. 添加 Nios II Processor。

双击 Altera SOPC Builder -> Nios II Processor，在弹出的对话框中间选择第一个 Nios II/e，表示 economy，最小的 NIOS II 核心。下面的 Reset Vector 和 Exception Vector 都选择 onchip_men，即刚才添加的片上 RAM 的名称。其它的都保留默认设置即可。点击 Finish 添加 CPU 核。如图 6-4。

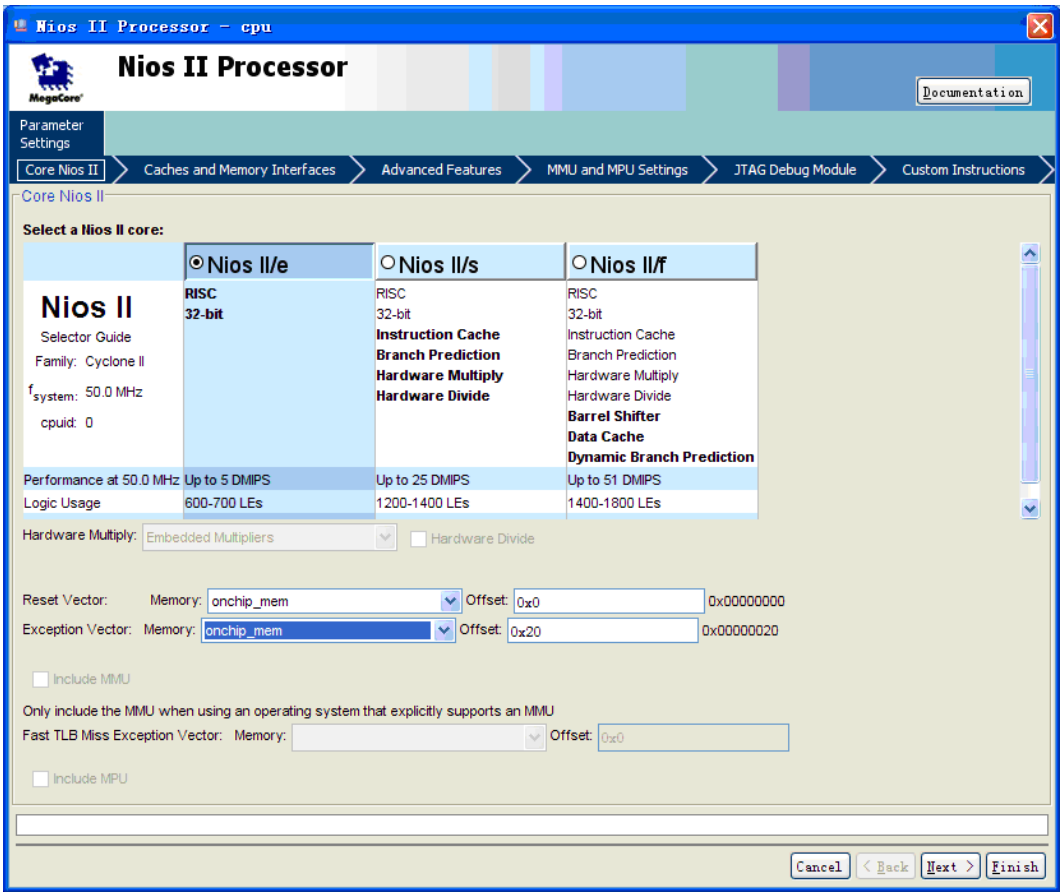


图 6-4　添加 CPU 并设置参数

7.　添加 JTAG UART，默认即可。如图 6-5。

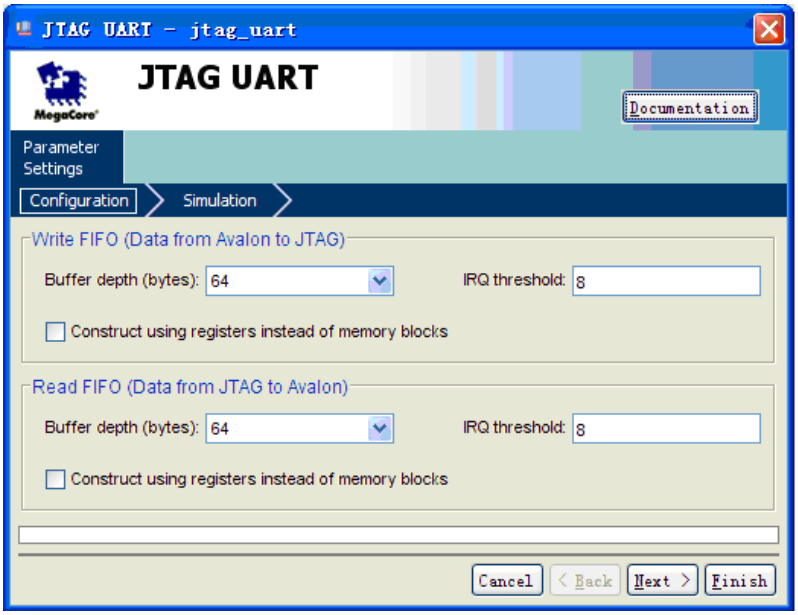


图 6-5　添加 JTAG UART 并设置参数

8. 添加 LCD，保持默认。如图 6-6。

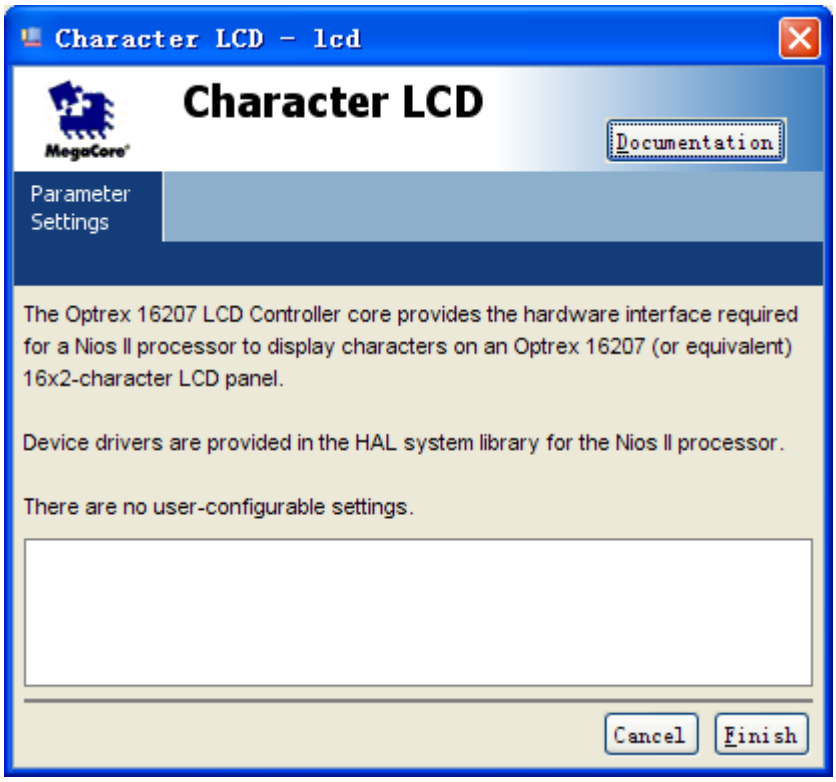


图 6-6 添加 LCD

9. 完成 SOPC 工程设计，如图 6-7。

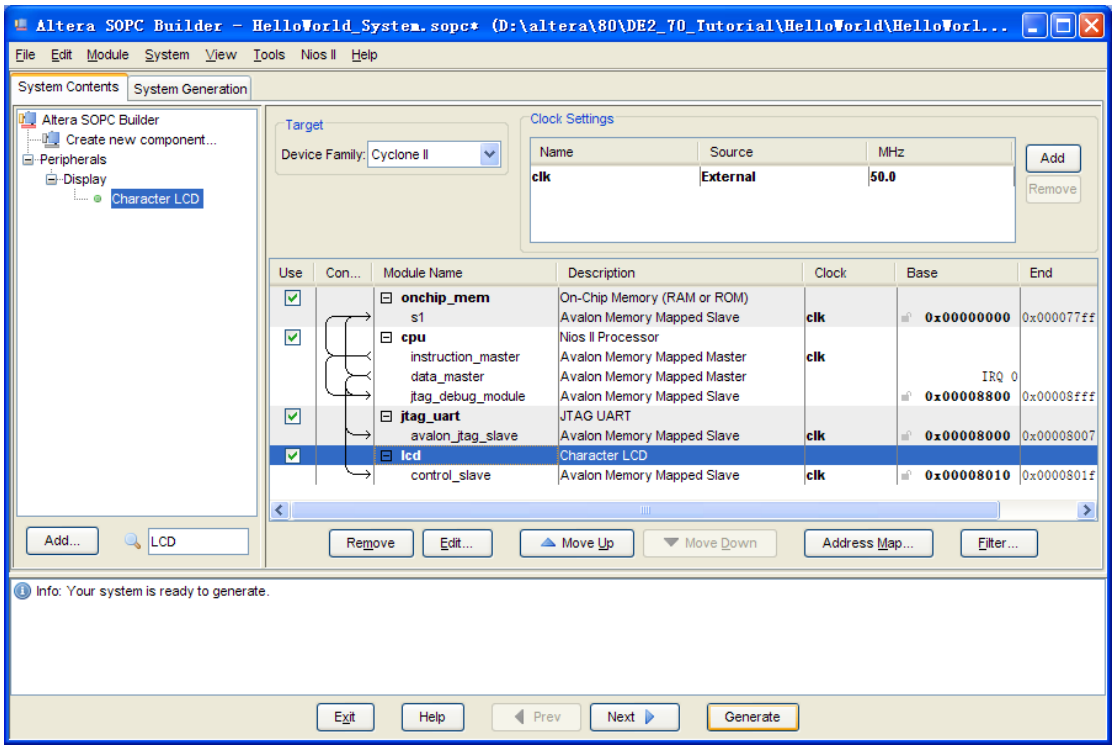


图 6-7 完成的 SOPC 工程

10. 生成 SOPC 软核处理器系统。通过点击下方 Generate 完成，如图 6-8。

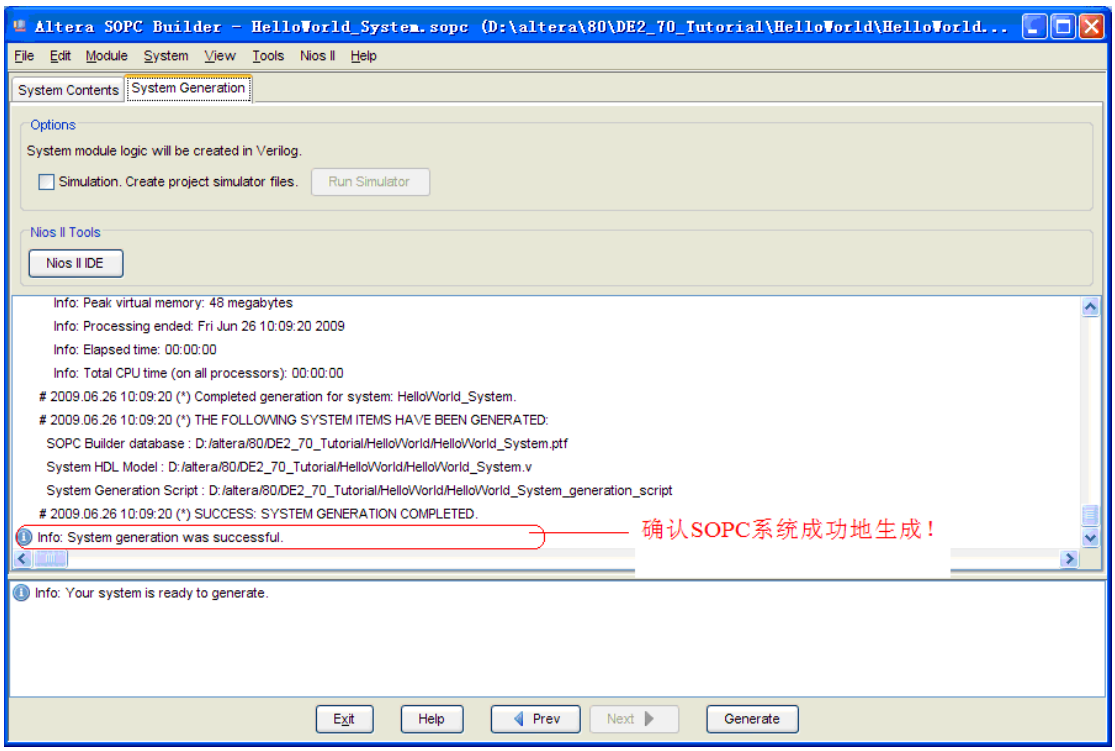


图 6-8 生成系统

注意：如果实验过程中因为某种原因修改了内存大小，例如：onchip memory。参看图 6.3。则需要执行自动指定存储器基地址操作。

自动指定存储器基地址操作选项如下图（图 6-9）所示。

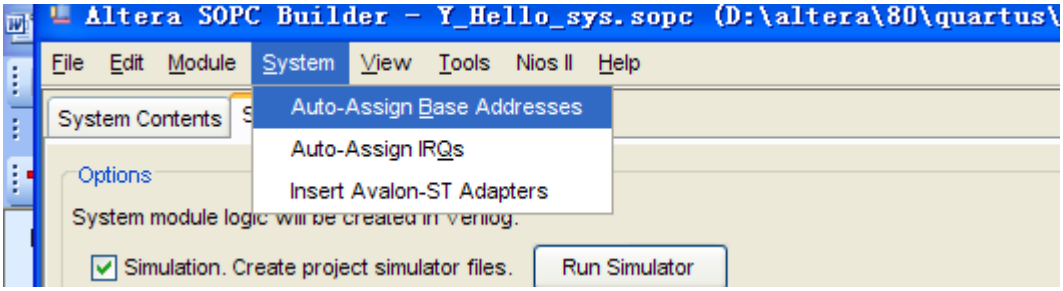


图 6-9　自动指定存储器基地址

6.3 用 Verilog 语言完成顶层实体

11. 大多数情况下，都选择使用语言完成顶层实体。这是因为引脚太多，使用符号框图容易出错。退出 SOPC Builder。

12. 回到 Quartus II 刚才建立的 HelloWorld 工程界面。将如下列出的 Verilog HDL 代码保存至 HelloWorld.v。

注意：列出了两种等价编写方法，其中右侧的源代码是一种精简编写方式。

```
module HelloWorld
(
iCLK_50,
iKEY,
oLCD_ON,
oLCD_BLON,
oLCD_RW,
oLCD_EN,
oLCD_RS,
LCD_D
);

input iCLK_50;
input [0:0] iKEY;
inout [7:0] LCD_D;

output oLCD_ON;
output oLCD_BLON;

output oLCD_RW;
output oLCD_EN;
output oLCD_RS;

assign oLCD_ON = 1'b1;
assign oLCD_BLON = 1'b1;

HelloWorld_System(
.clk(iCLK_50),
.reset_n(1),
.LCD_E_from_the_lcd(oLCD_EN),
.LCD_RS_from_the_lcd(oLCD_RS),
.LCD_RW_from_the_lcd(oLCD_RW),
.LCD_data_to_and_from_the_lcd(LCD_D)
);
endmodule
```

```
/* 实验五 LCD显示 HelloWorld源代码 */
module HelloWorld(
input iCLK_50,
input [0:0] iKEY,
inout [7:0] LCD_D,

output oLCD_BLON,
output oLCD_ON,

output oLCD_EN,
output oLCD_RS,
output oLCD_RW
);

assign oLCD_ON = 1'b1;
assign oLCD_BLON = 1'b1;

HelloWorld_System (
.clk(iCLK_50),
.reset_n(iKEY[0]),


.LCD_E_from_the_lcd(oLCD_EN),
.LCD_RS_from_the_lcd(oLCD_RS),
.LCD_RW_from_the_lcd(oLCD_RW),
.LCD_data_to_and_from_the_lcd(LCD_D)
);
endmodule
```

13. 分析与综合，这一步除了检查顶层实体是否出错外，尚有另一个目的，就是为了将已有配置信息写入 qsf 文件。以免分配引脚时造成 qsf 文件的修改时间比内存更新，丢失以往配置信息(如编译输出路径)。

14. 引脚分配，编辑.qsf 文件，添加引脚定义。结果如图 6-10 所示。

```
set_location_assignment PIN_B2 -to LCD_D[7]
set_location_assignment PIN_C3 -to LCD_D[6]
```

```
set_location_assignment PIN_C2 -to LCD_D[5]
set_location_assignment PIN_C1 -to LCD_D[4]
set_location_assignment PIN_D3 -to LCD_D[3]
set_location_assignment PIN_D2 -to LCD_D[2]
set_location_assignment PIN_E3 -to LCD_D[1]
set_location_assignment PIN_E1 -to LCD_D[0]
set_location_assignment PIN_AD15 -to iCLK_50
set_location_assignment PIN_G3 -to oLCD_BLON
set_location_assignment PIN_E2 -to oLCD_EN
set_location_assignment PIN_F1 -to oLCD_ON
set_location_assignment PIN_F2 -to oLCD_RS
set_location_assignment PIN_F3 -to oLCD_RW
```

Named: Edit:

	Node Name	Direction	Location	I/O Bank	Vref Group	I/O Standard
1	LCD_D[7]	Bidir	PIN_B2	2	B2_N0	3.3-V LVTTL (default)
2	LCD_D[6]	Bidir	PIN_C3	2	B2_N0	3.3-V LVTTL (default)
3	LCD_D[5]	Bidir	PIN_C2	2	B2_N0	3.3-V LVTTL (default)
4	LCD_D[4]	Bidir	PIN_C1	2	B2_N0	3.3-V LVTTL (default)
5	LCD_D[3]	Bidir	PIN_D3	2	B2_N1	3.3-V LVTTL (default)
6	LCD_D[2]	Bidir	PIN_D2	2	B2_N1	3.3-V LVTTL (default)
7	LCD_D[1]	Bidir	PIN_E3	2	B2_N0	3.3-V LVTTL (default)
8	LCD_D[0]	Bidir	PIN_E1	2	B2_N1	3.3-V LVTTL (default)
9	iCLK_50	Input	PIN_AD15	7	B7_N3	3.3-V LVTTL (default)
10	iKEY[0]	Input	PIN_T29	6	B6_N0	3.3-V LVTTL (default)
11	oLCD_BLON	Output	PIN_G3	2	B2_N0	3.3-V LVTTL (default)
12	oLCD_EN	Output	PIN_E2	2	B2_N1	3.3-V LVTTL (default)
13	oLCD_ON	Output	PIN_F1	2	B2_N2	3.3-V LVTTL (default)
14	oLCD_RS	Output	PIN_F2	2	B2_N2	3.3-V LVTTL (default)
15	oLCD_RW	Output	PIN_F3	2	B2_N0	3.3-V LVTTL (default)
16	<<new node>>					

图 6-10 引脚分配图

15. 编译下载。

6.4 Nios 软件设计

16. 进入 Nios II 的集成开发环境操作。

打开 **NIOS II IDE** 软件的方法如下：

在 PC 机桌面单击快捷图标  ，或者按照软件路径：开始->所有程序->Altera->Nois II EDS 8.0->Nois II 8.0 IDE。

第一次打开的时候会提示选择工作空间。也可在程序打开后选择菜单栏 File -> Switch Workspace…以选择一个合适的工作空间。如果仅仅在一台电脑上做开发，选择默认工作空间即可，但如果需要经常移动开发平台，最好还是保存到工程目录下，和软件源码放在一起。这里我们选择..\HelloWorld\Software，如图 6-11 所示。

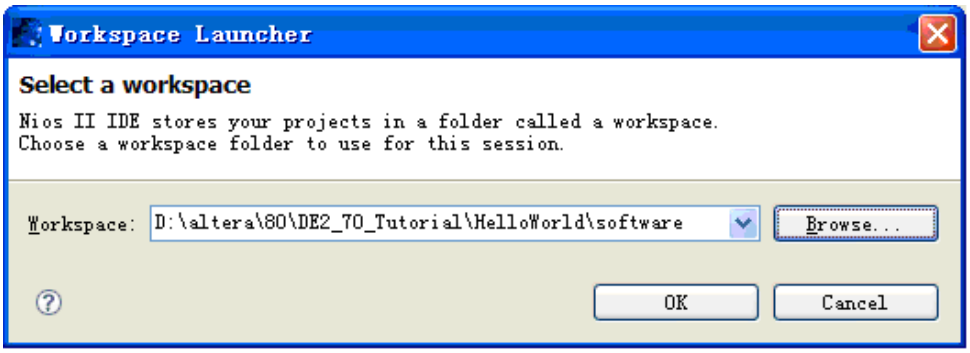


图 6-11　工作区选择

17. 自动重启 Nios II IDE 后，会进入 Nios II IDE 的主界面，如图 6-12。

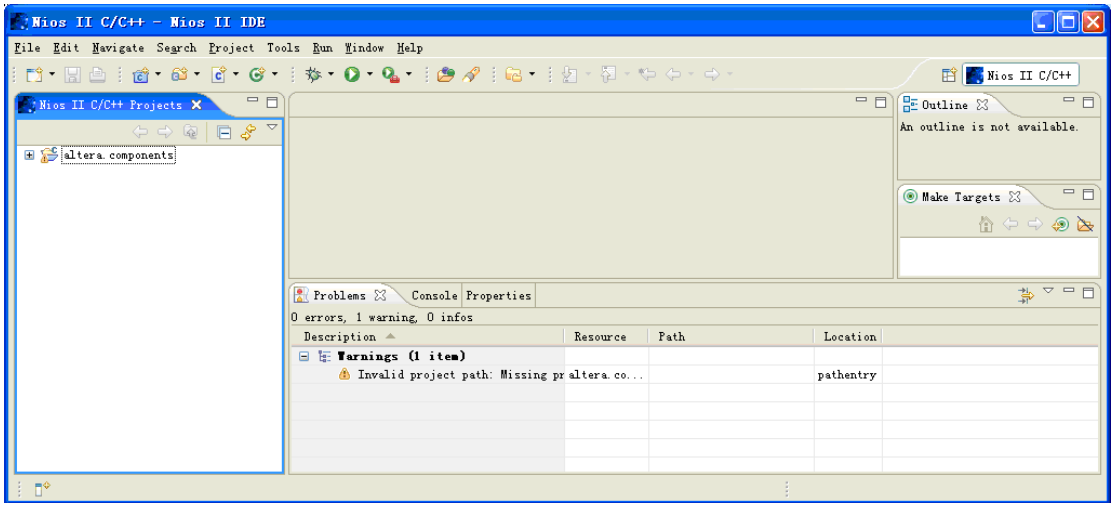


图 6-12　Nios II IDE 的主界面

18. 选择 “File->New->Nios II C/C++ Application”。参看图 6-13。

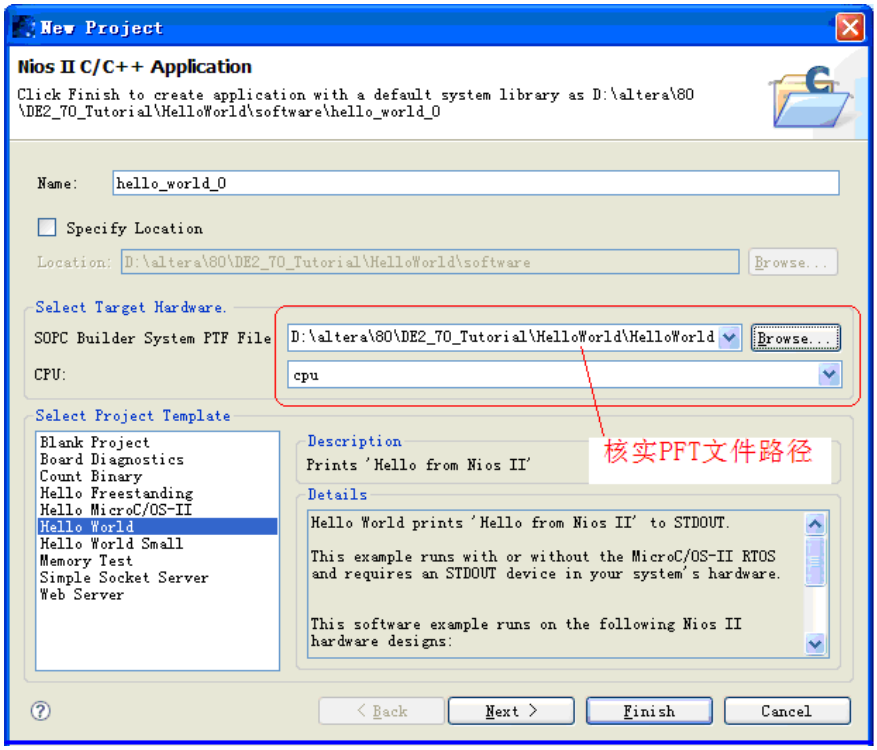


图 6-13 新建一个 NIOS 的应用工程

在“New project”对话框的中部左侧选择 HelloWorld 模板，中部右侧选择好本 HelloWorld_System 的 SOPC 系统对应的 ptf 文件，然后单击按钮“Finish”。

19. 选择系统库属性

鼠标对准随后出现的 hello_world_0.c 工程项目处，右击选择 System Library Properties（位于最下面的选择项），如图 6-14 所示。

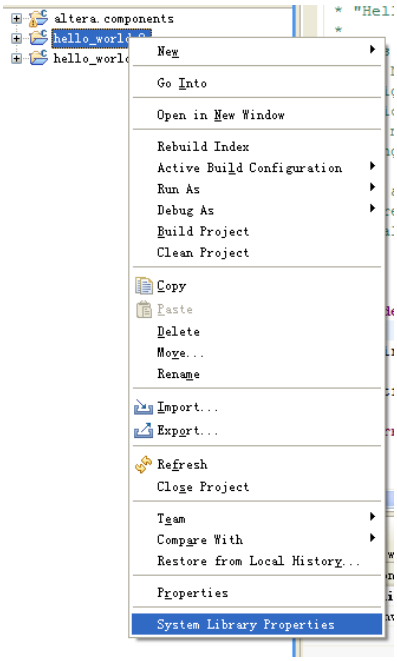


图 6-14 选择 System Library Properties

配置 System Library Properties，将 stdin，stdout，stderr 三项全部设为 lcd，在下图所示界面中，取消掉 Clean Exit (Flush Buffer)和 Support C++前的勾，因为我们的程序不会退出，也不包含 C++的函数和库。复选中 Program never exits，以减小程序体积。其他保持默认设置即可，之后单击 OK。如图 6-15 所示。

注意：Reduced device drivers 的勾绝对不能勾，否则 lcd 无响应。

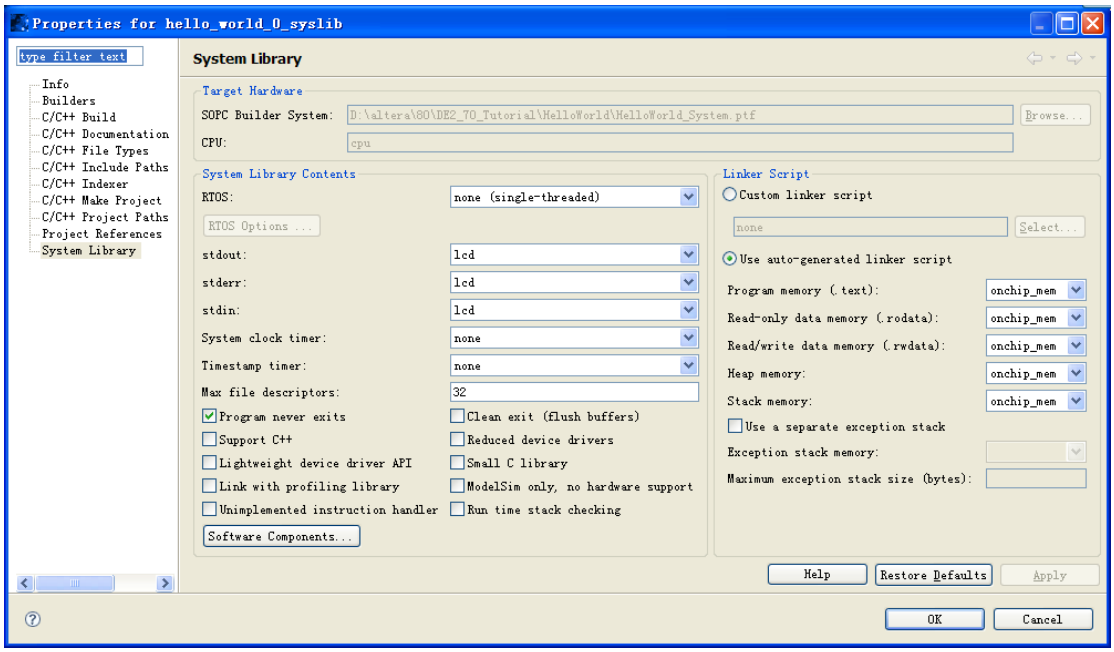


图 6-15 配置 System Library Properties

20. Project->Build All，很遗憾得到了一个内存已满的结果，如图 6-16.

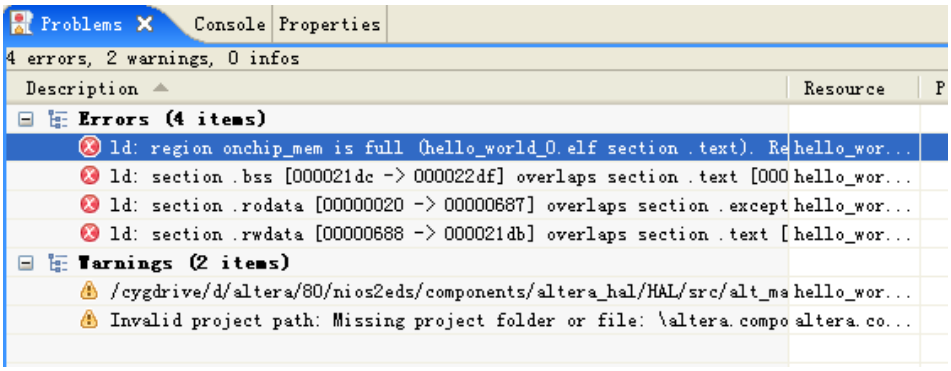


图 6-16　编译失败出错提示

21. Clean 掉刚编译的工程，Project->clean，去掉 Start a build immediately 的勾防止立即再 build。

22. 先后对准两个工程（一个应用工程与一个系统库工程，参看图 6-17。）右键单击，执行弹出菜单里的 Properties。

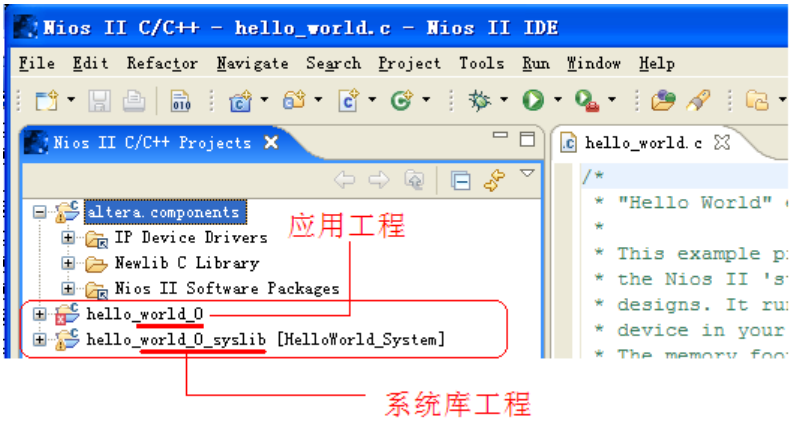


图 6-17　HelloWorld 的两个工程

23. 参看图 6-14 倒数第 2 项。

选中弹出对话框里的 C/C++ Build 选项卡，将 Configuration 设为 Release，编译器参数设为-Os，如图 6-18。

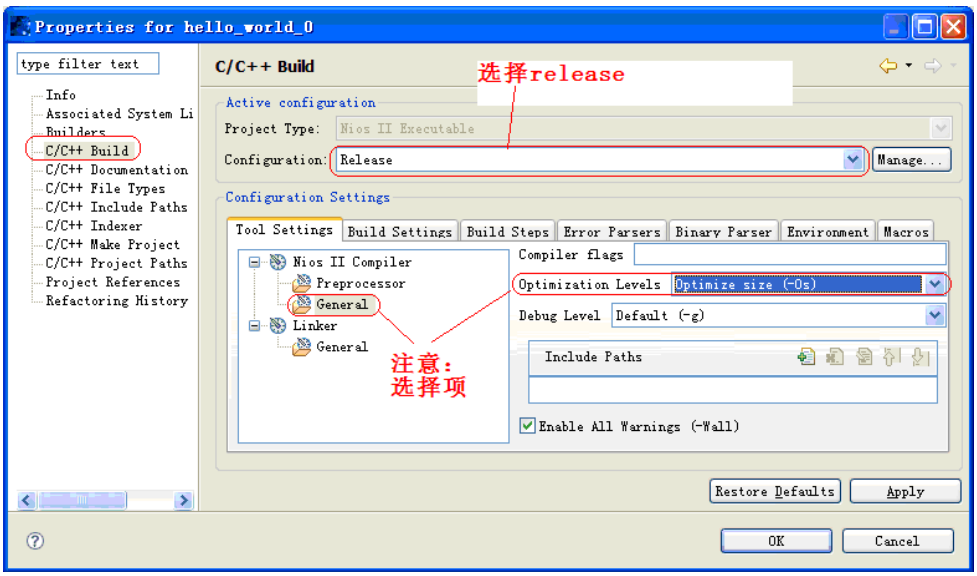


图 6-18　两个工程的 C/C++编译器参数

注意：在图 6-18 给出的画面中，先选择 General，再设置优化级别-Os。

24. Project->Build All，编译结果为 22K

25. 去 System Library Properties 勾上 Small C Library，再次编译，程序只有 12K 了。

26. 在 hello_world_0 工程上右击，选择 Run As ->Nios II Hardware，如图 6-19。

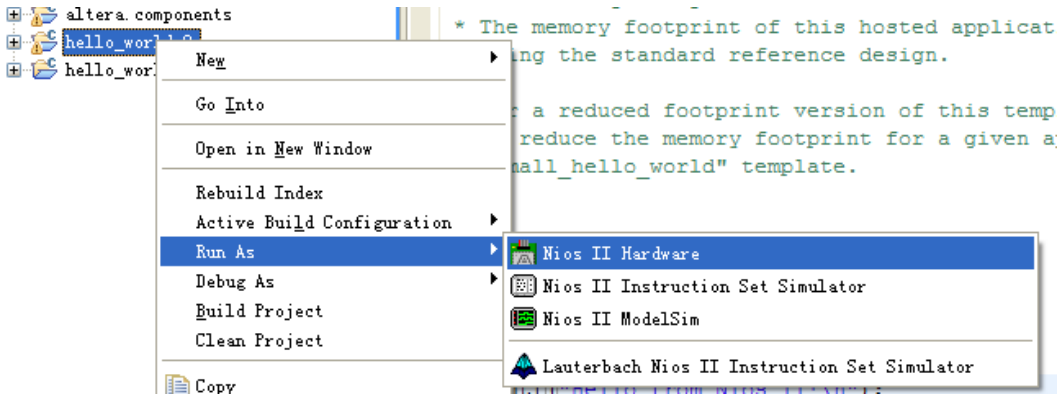


图 6-19　运行 C 代码

27. 查看实验板上的结果。运行结果为在 LCD 上显示：Hello from Nios，后面的 II 不见了。参看图 6-20。

图 6-20　DE2-70 实验板 LCD 上显示：Hello from Nios

6.5 添加间隔定时器

28. 对于较长字符串，DE2-70 的常见做法是在 LCD 屏上滚动显示。方法是加入一个间隔定时器 Interval timer。

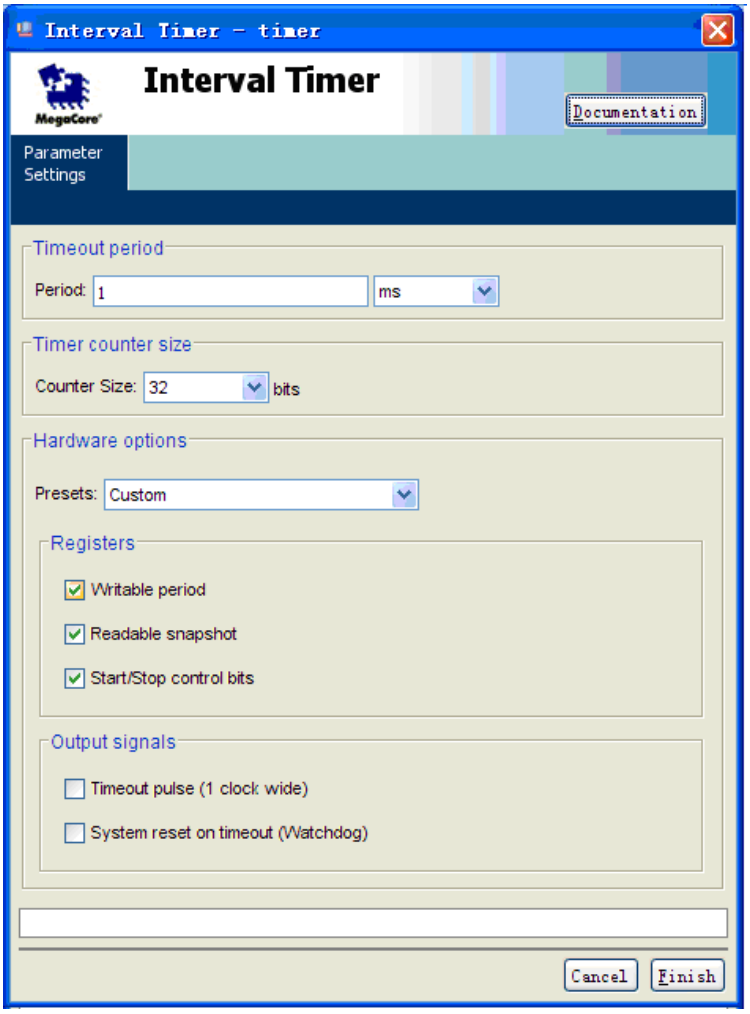


图 6-21　Interval Timer

29. 回到 Quartus II 操作界面，在 SOPC 系统中添加 Interval Timer，配置保持默认，如图 6-21。之后执行“generate”，重新生成系统。

30. 然后在 hello_world_0 应用工程的 System Library Properties 中选中那个 timer 作为 System clock timer，如图 6-22。

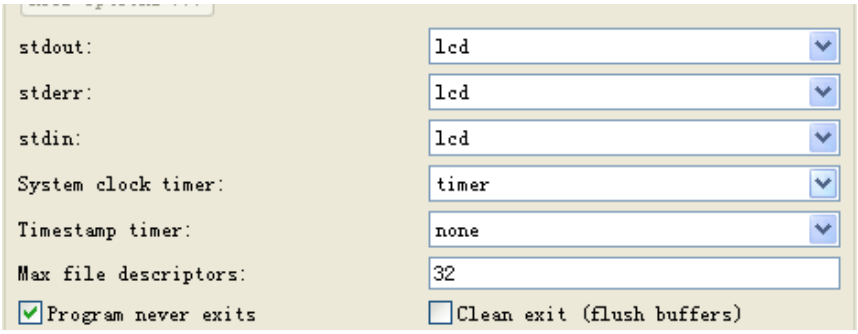


图 6-22　设定 Timer

31. 编译运行，即可在 DE2-70 实验板的 LCD 屏幕上发现滚动显示的“Hello from Nios II”。

◆ 本实验指导结束

第 7 章　实验六　跑马灯实验

● 实验说明

事实上在 FPGA 实验板上用 Verilog 语言或者 VHDL 语言都能够很快地写出跑马灯实验程序。本实验完成的是基于 SOPC 的跑马灯设计，具有一定的操作复杂性。

● 实验步骤

7.1 建立 Quartus 工程

1. 新建 Quartus 工程 RunningLED，顶层实体名 RunningLED
2. 重新设置编译输出目录为../ RunningLED/release。

7.2 建立 SOPC 系统

3. 点击 Quartus II 软件右上方图标  打开 SOPC Builder，创建一个 SOPC 系统。填写系统名称为 RinningLED_System，并指定 Verilog 为描述系统的语言，如图 7-1。

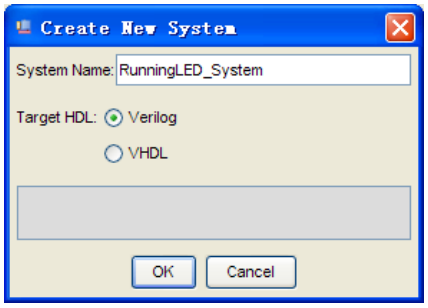


图 7-1 添加系统名称并指定语言

4. 在系统上添加 On-Chip Memory。

在程序左侧列表中选择 Memory and Memory Controllers -> On-Chip -> On-Chip Memory (RAM or ROM)，双击添加至系统中。

在弹出的对话框中指定片上 RAM 的属性，因为不需要显示，编译结果很小，保持默认即可。

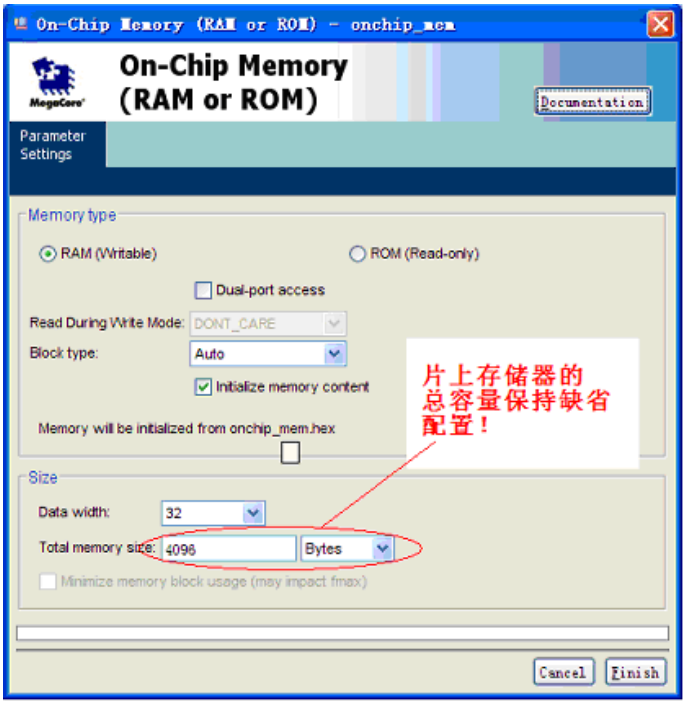


图 7-2 指定 On-Chip Memory 属性

5. 添加 Nios II Processor。

双击 Altera SOPC Builder -> Nios II Processor，在弹出的对话框中间选择第一个 Nios II/e，表示 economy，最小的NIOS II核心。下面的 Reset Vector 和 Exception Vector 都选择 onchip_men，即刚才添加的片上 RAM 的名称。其它的都保留默认设置即可。点击 Finish 添加 CPU 核。如图 7-3 所示。

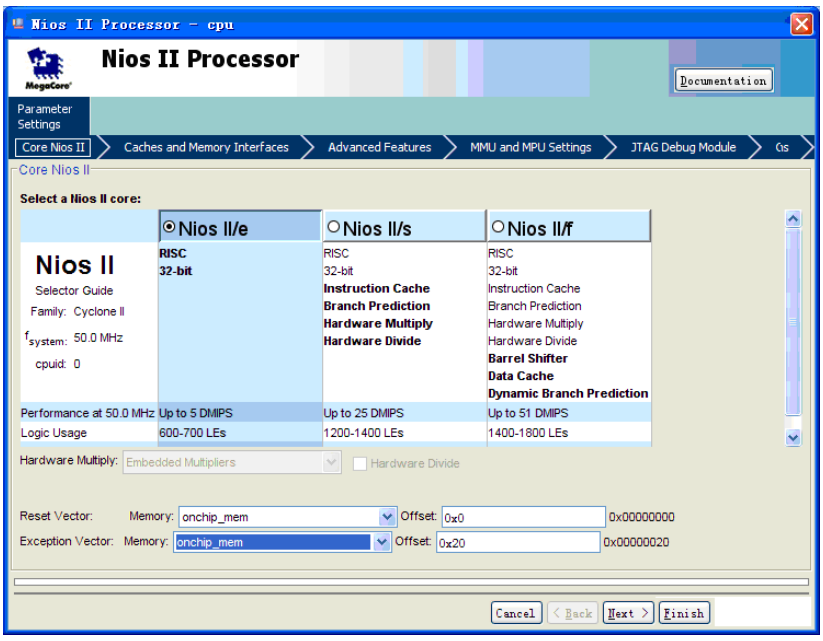


图 7-3 添加 CPU 设置参数

6. 添加定时器。

在列表中选择 Peripherals -> Microcontroller Peripherals -> Interval Timer，弹出如下对话框。定时器在本系统中主要作用是产生一个固定间隔的中断信号，让 CPU 改变 LED 灯的状态。因此在 Period 中选择 500ms，表示灯的状态每 500ms 改变一次。如图 7-4。

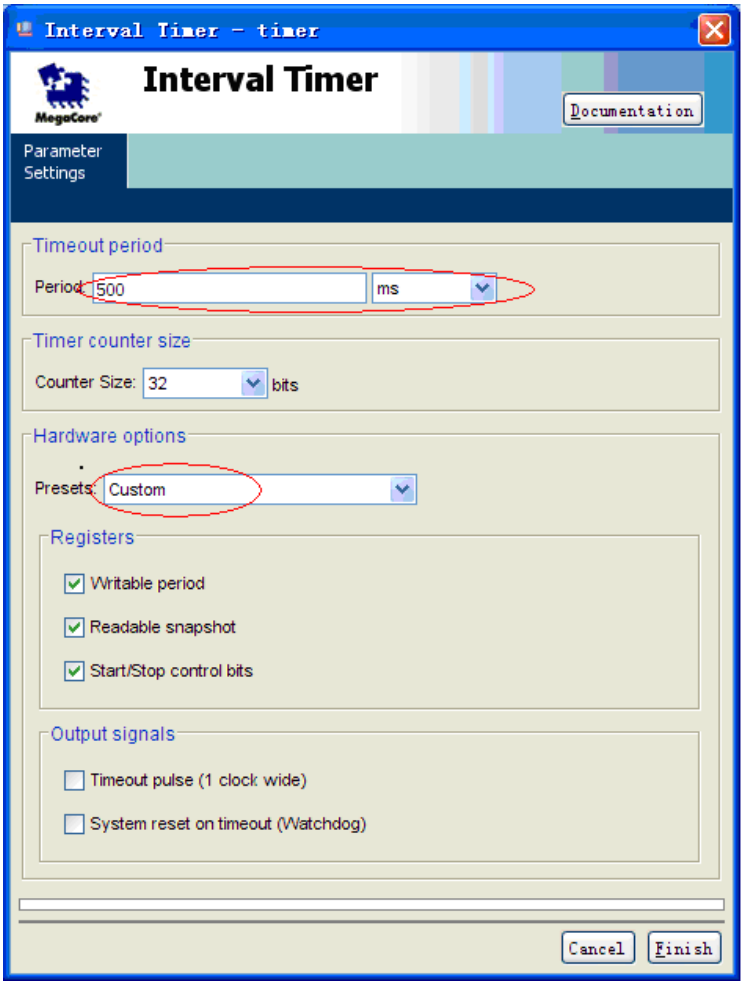


图 7-4 添加定时器并设置参数

7.　添加 IO 控制器。

双击 Peripherals -> Microcontroller Peripherals -> PIO (Parallel I/O)，保持默认设置即可，表示有 8 个输出用 IO 口，分别控制开发板上的 8 个绿色 LED 灯（LEDG[7..0]）。如图 7-5。

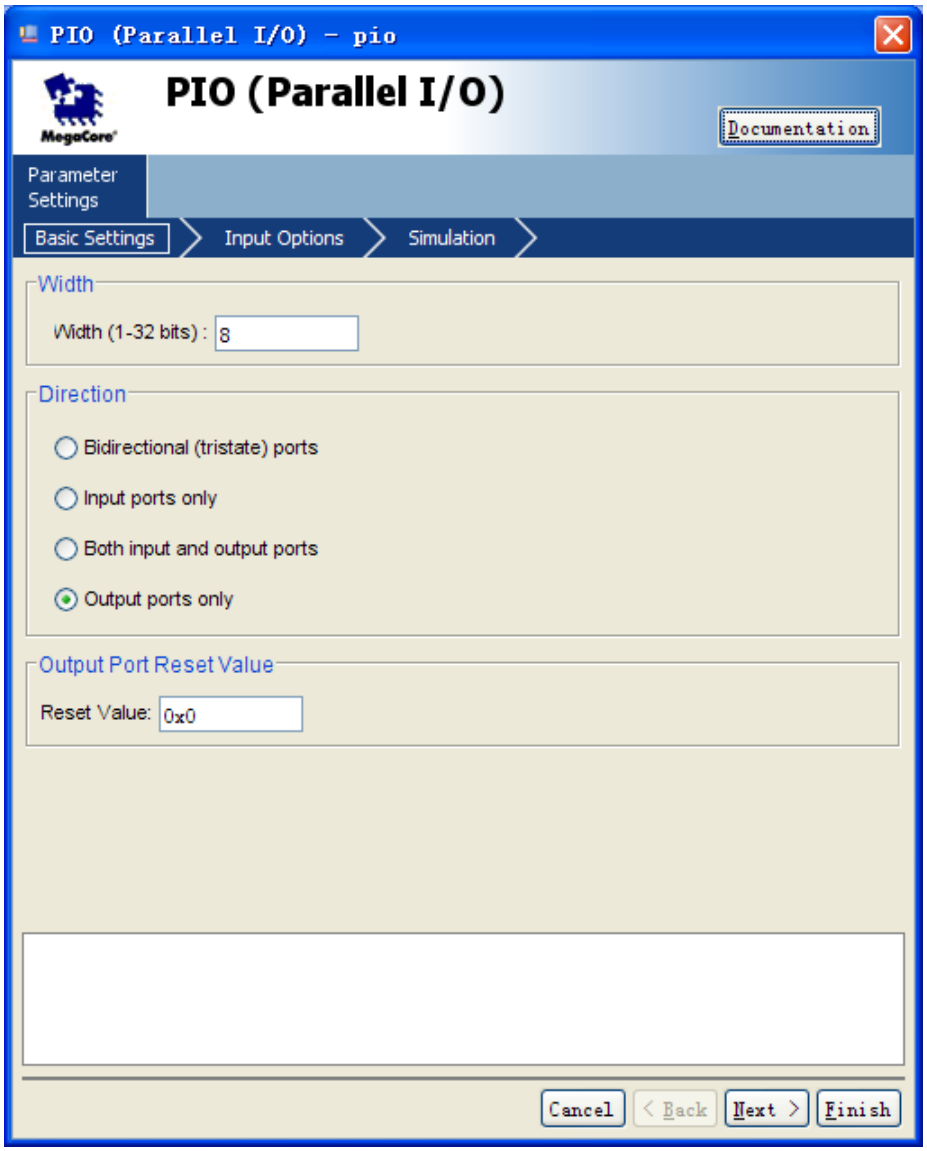


图 7-5 添加 IO 控制器并设置参数

8. 完成 SOPC 工程设计，为了方便将 PIO 的名称改为 pio_ledg。在 pio 上点击右键 -> rename，将名称改为 pio_ledg。如图 7-6。

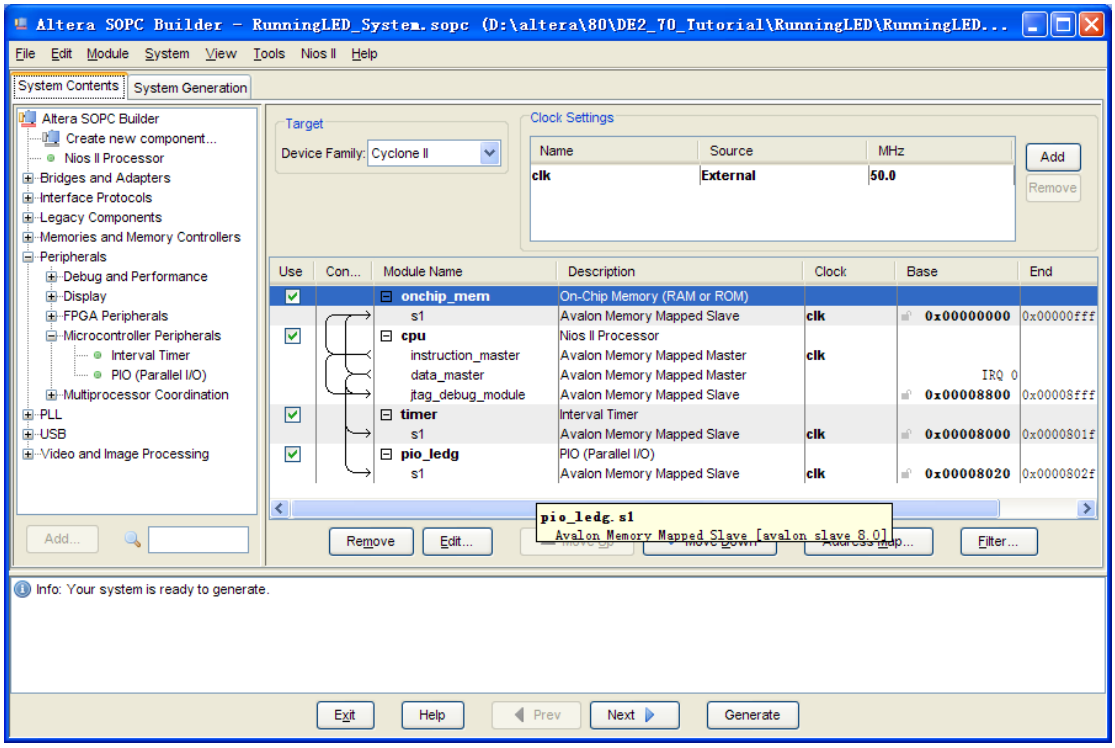


图 7-6 完成的 SOPC 工程

注意：系统的每个组件都需要一个地址才能正常工作。某些组件，如定时器（Interval Timer）还需要分配一个 IRQ 号。如果发现各组件的地址或者 IRQ 号出现冲突，可以选择菜单栏上 System -> Auto-Assign Base Addresses 以及 System -> Auto-Assign IRQs 自动设定地址和 IRQ。系统 IRQ 可以是从 0 到 31 的整数，数值越小优先级越高。

9. 生成系统。通过点击下方 Generate 完成。如图 7-7。

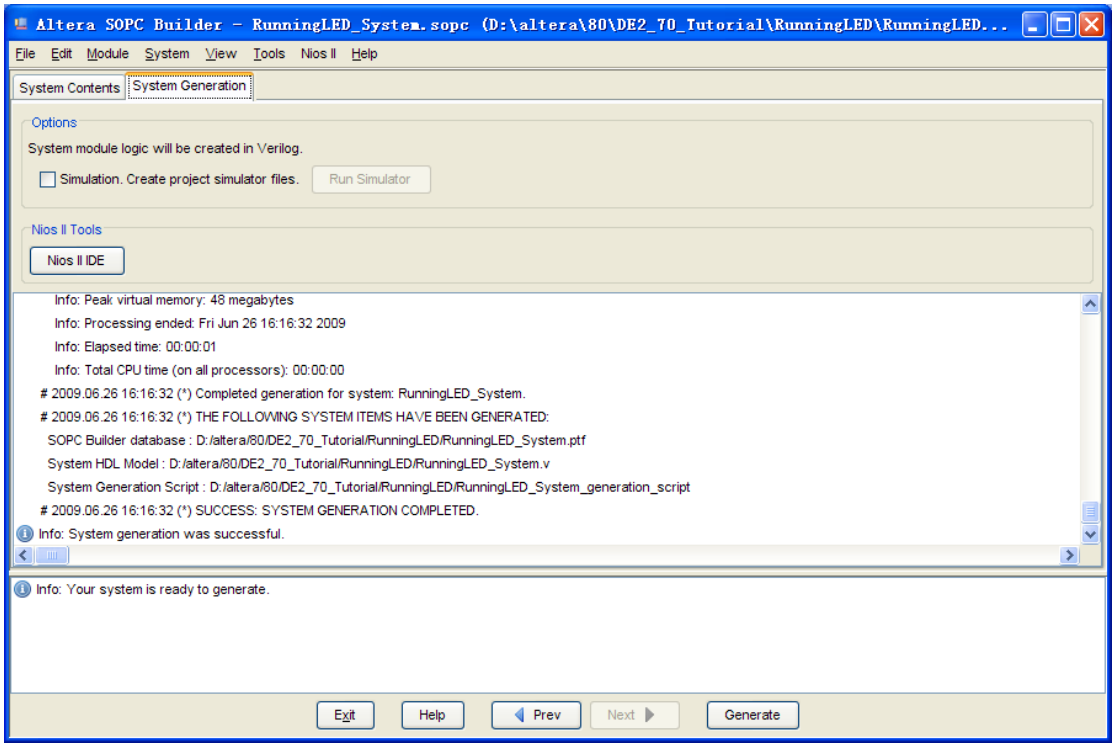


图 7-7 生成系统

7.3 用符号框图完成顶层实体

10. 这次使用符号框图完成顶层实体。新建一个符号文件，添加刚才建立的 SOPC 系统。如图 7-8。

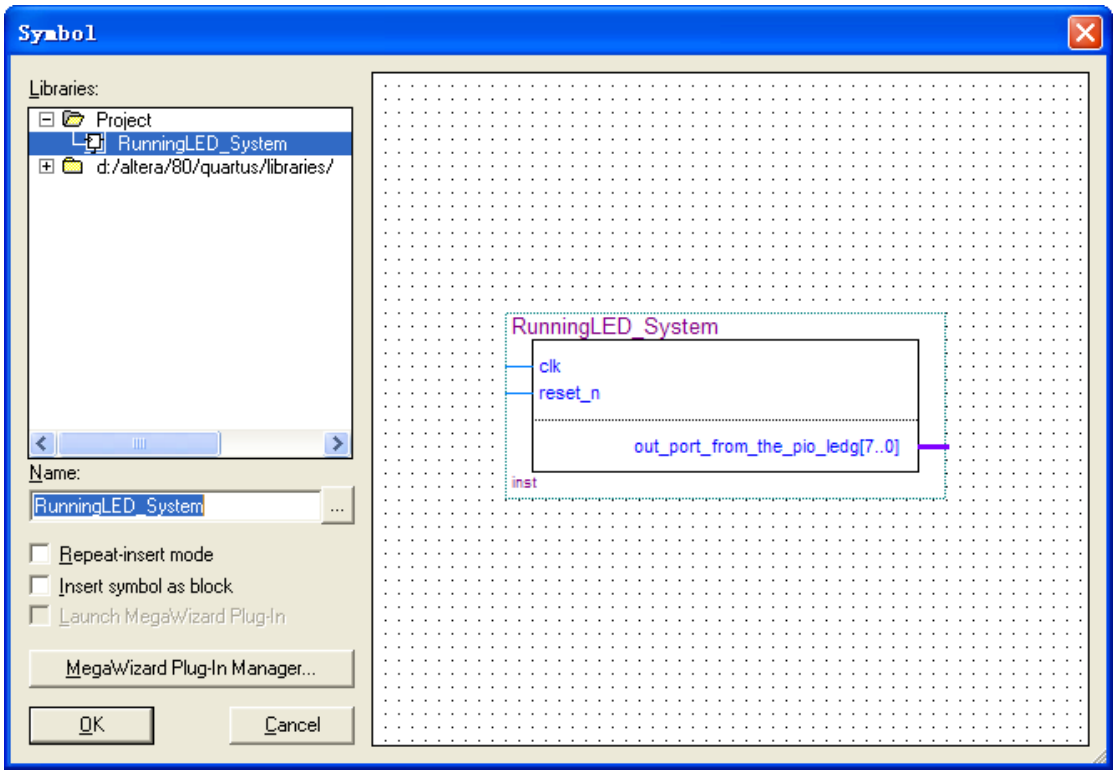


图 7-8 添加 SOPC 系统

11. 添加输入与输出端口。在空白部分双击，在 Name 框内输入 input 可以快速定位，添加输入端口。一共需要两个。然后使用同样步骤添加一个 ouput 输出端口。结果应如图 7-9 所示。

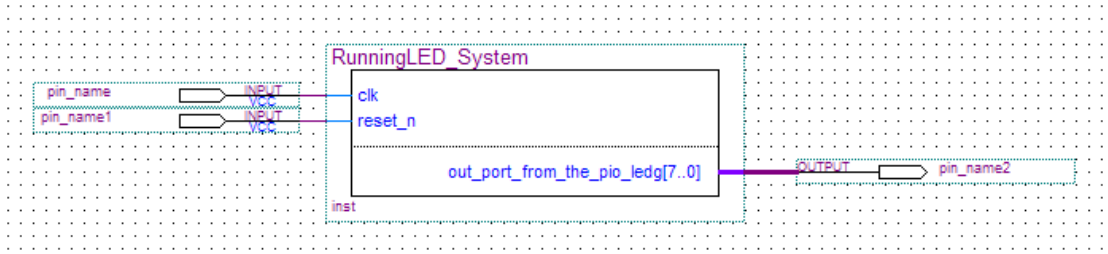


图 7-9 添加结果图

将两个输入端分别改名为 iCLK_50 及 iKEY[0]，代表开发板上的 50MHz 晶振和 KEY0 按钮。将输出端改名为 oLEDG[7..0]，代表开发板上的 oLEDG7 到 oLEDG0 共 8 个绿色 LED 灯。需要注意的是 SOPC Builder 生成的系统的重启信号为低电平有效，开发板上的按键按下后代表低电平，弹起代表高电平。然后将这几个元件连接起来，硬件电路部分设计完毕。电路应如下图所示。(注意：此处的名称修改应该与 DE2-70 引脚的配置相一致)

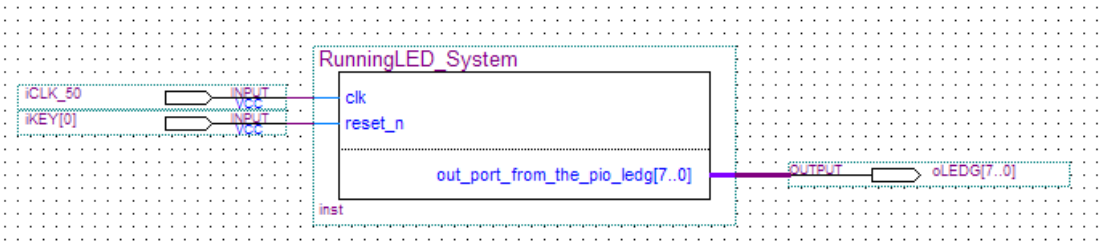


图 7-10　电路图

12. 保存 bdf 文件，然后执行分析与综合
13. 分配引脚

```
set_location_assignment PIN_AD15 -to iCLK_50
set_location_assignment PIN_AC14 -to oLEDG[8]
set_location_assignment PIN_W27 -to oLEDG[0]
set_location_assignment PIN_W25 -to oLEDG[1]
set_location_assignment PIN_W23 -to oLEDG[2]
set_location_assignment PIN_Y27 -to oLEDG[3]
set_location_assignment PIN_Y24 -to oLEDG[4]
set_location_assignment PIN_Y23 -to oLEDG[5]
set_location_assignment PIN_AA27 -to oLEDG[6]
set_location_assignment PIN_AA24 -to oLEDG[7]
set_location_assignment PIN_T29 -to iKEY[0]
```

14. 编译下载。编译完成后将程序烧写至 FPGA 开发板。由于目前还没有编写软件，因此开发板上不会有什么现象。

7.4 软件设计

15. 打开 Nios II IDE，首先选择一个合适的工作空间，依旧设置在<工程所在目录>\softawre。如图 7-11。

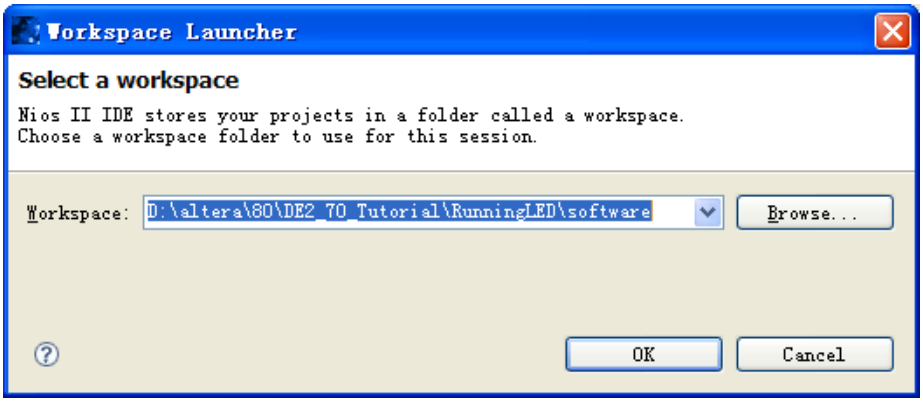


图 7-11　选择工作空间

确认以后软件会重新启动。在欢迎界面中选择 Workbench，进入主界面

16. 选择 File -> New -> Nios II C/C++ Application

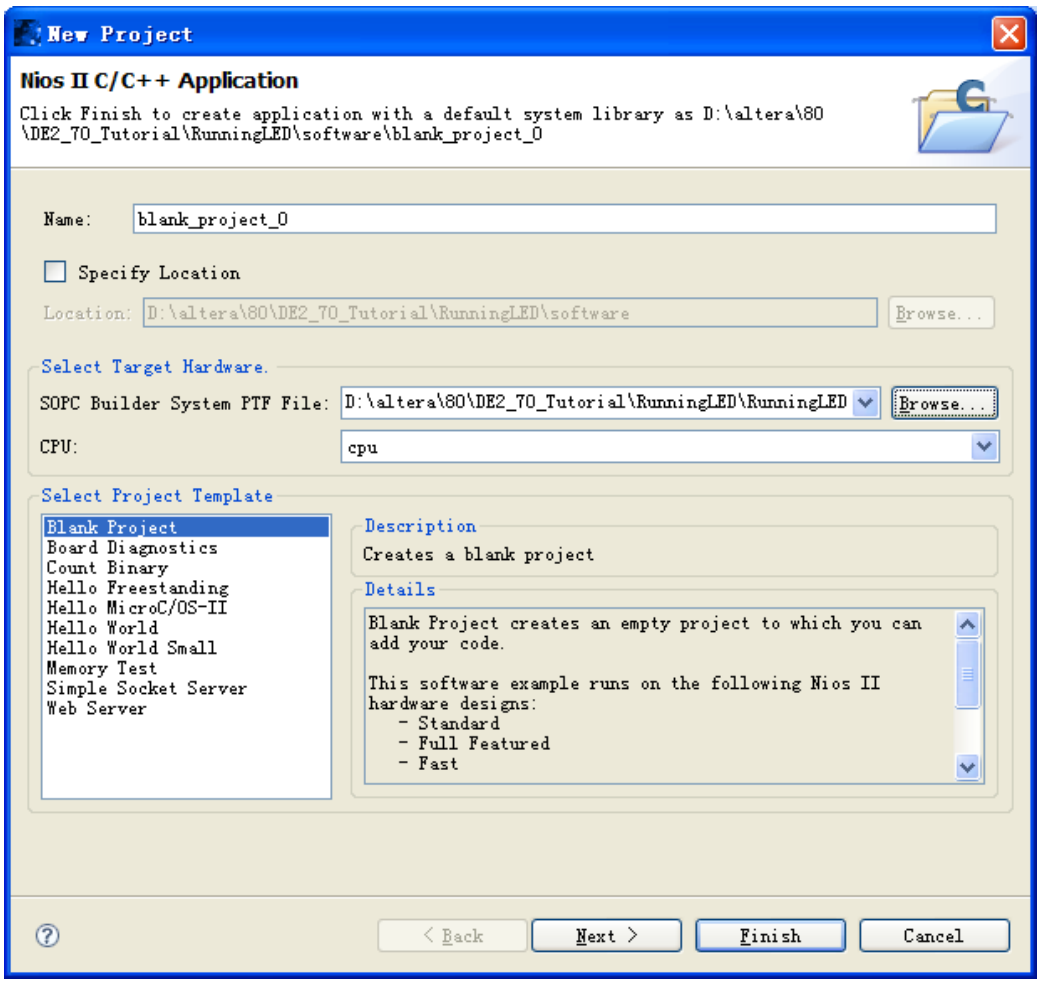


图 7-12　选择工程模板

在 Select Project Template 内选择第一项 Blank Project，Name 使用默认 blank_project_0，SOPC Builder System PTF File 使用默认设置，即刚才生成的 SOPC 系统，点击 Finish 完成。如图 7-12。

17. 选择工程 blank_project_0，右键单击，选择 Properties，在 C/C++ build 选项卡中配置编译器参数，跟上次一样依旧是-DALT_RELEASE -Os -g –Wall，如图 7-13。

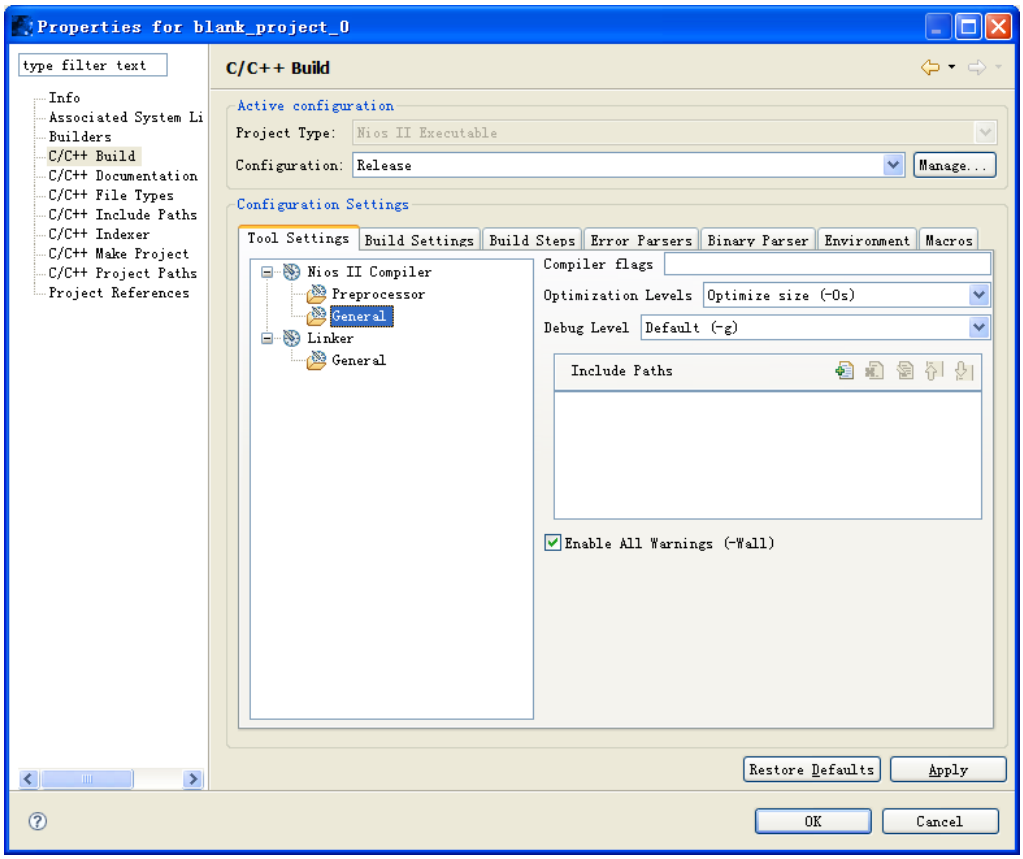


图 7-13　配置工程编译选项

18. 配置 blank_project_0_syslib 的编译器参数。跟 blank_project_0 一样。

19. System Library Properties 设置。

右击工程 blank_project_0_syslib -> Properties，选中 System Library 选项卡，在下图所示界面中，取消掉 Clean Exit (Flush Buffer)和 Support C++前的勾，因为我们的程序不会退出，也不包含 C++的函数和库。选中 Program never exits，Reduce device drivers 和 Small C library 以减小程序体积。其他保持默认设置即可。如图 7-14。**再次声明如果使用 lcd 输出，则不能勾选 Reduce device drivers。**

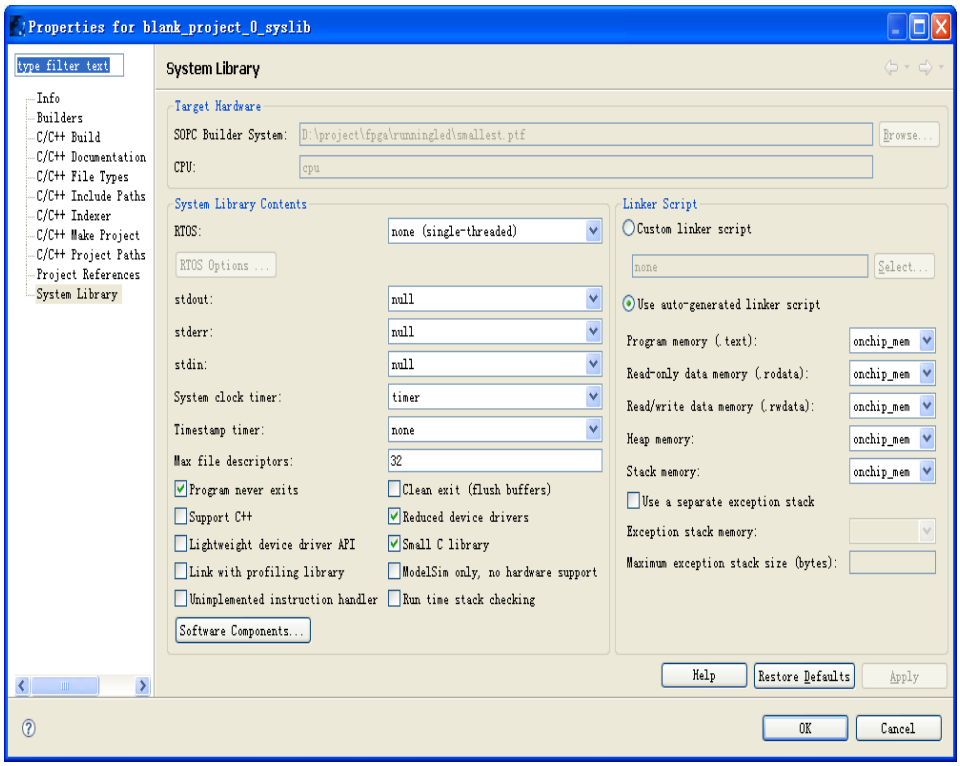


图 7-14　System Library Properties 设置

20. 右键点击工程 blank_project_0，选择 New -> Source File。如图 7-15。

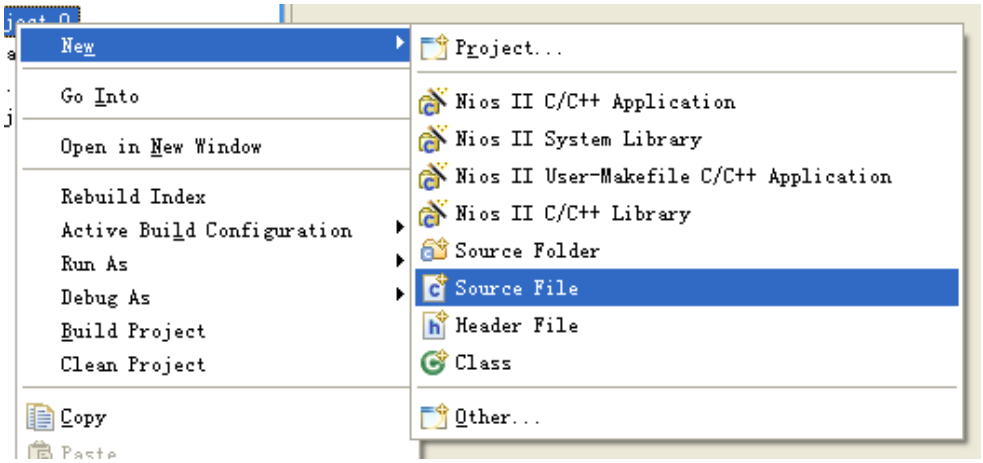


图 7-15 添加源文件图

在弹出的对话框中指定文件名为 main.c，并将以下代码复制进去，并保存。

```
#include "system.h"
#include <sys/alt_irq.h>
#include "alt_types.h"
#include <io.h>

// Internal Timer Overflow interrupt
static void timer_overflow(void* context，  alt_u32 id)
{
    IOWR(TIMER_BASE，0，0);
    if (*(alt_u8 *)context & 0x80)
```

```
    {
        *(alt_u8 *)context = 0x01;
    }
    else
    {
        *(alt_u8 *)context = *(alt_u8 *)context << 1;
    }
    IOWR(PIO_LEDG_BASE，0，*(alt_u8 *)context);
    return;
}

int main( )
{
    alt_u8 led = 0x01;
    // Register Interrupt Service Routine (ISR)
    alt_irq_register(TIMER_IRQ，  (void*)&led，  timer_overflow);
    while(1);
}
```

21. Project->Build All，编译，结果只用了 2K。
22. 右击 blank_project_0，选择 Run As ->Nios II Hardware 如图 7-16。

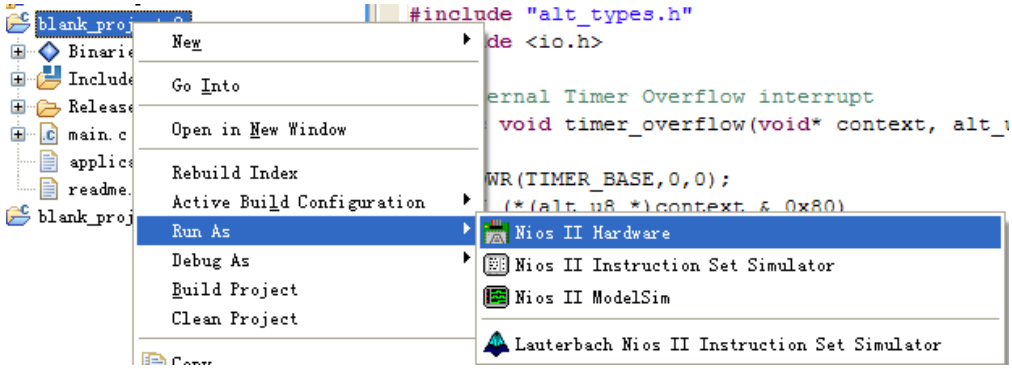


图 7-16 下载运行 C 程序

23. NIOS II IDE 将程序下载到开发板上，之后就能看到 8 个 LEDG 灯轮流点亮了。

◆ 本实验指导结束

第 8 章　实验七　C2H 编译器实验

● 实验说明

该实验将演示在 Nios II 处理器编写程序的时候，Nios II IDE 中所带的 C2H Compiler 工具的作用。什么是 C2H?它是(C to Hardware)的缩写，能将你原本的软件 C 语言程序代码变成硬件实现。

● 实验步骤

8.1 建立 Quartus 工程

1. 新建 Quartus 工程 C2H，顶层实体名 C2H
2. 重新设置编译输出目录为../ C2H/release。

8.2 建立 SOPC 系统

3. 打开 SOPC Builder，建立一个名为 C2H_System 的 SOPC 系统，并指定 Verilog 为描述系统的语言。
4. 在系统上添加 On-Chip Memory。保持默认设置即可。
5. 添加 Nios II Processor。依旧选择 E 型。
6. 添加串行 I/O，将 Width 改为 18，对应 18 个红色 LED 灯。如图 8-1。

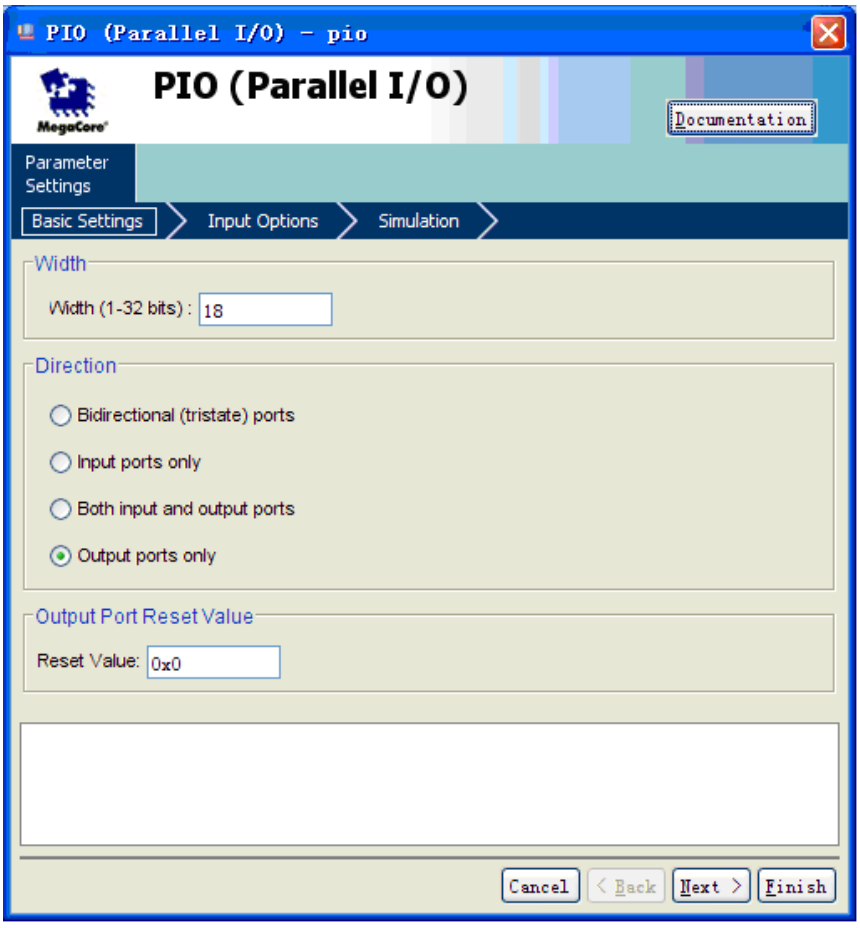


图 8-1　PIO 参数设置

在系统组件列表中，右键单击新添加的 pio，选择 Rename，重命名为 pio_ledr。

7. 选择菜单项 System -> Auto-Assign Base Addresses 去除基址错误。如图 8-2。

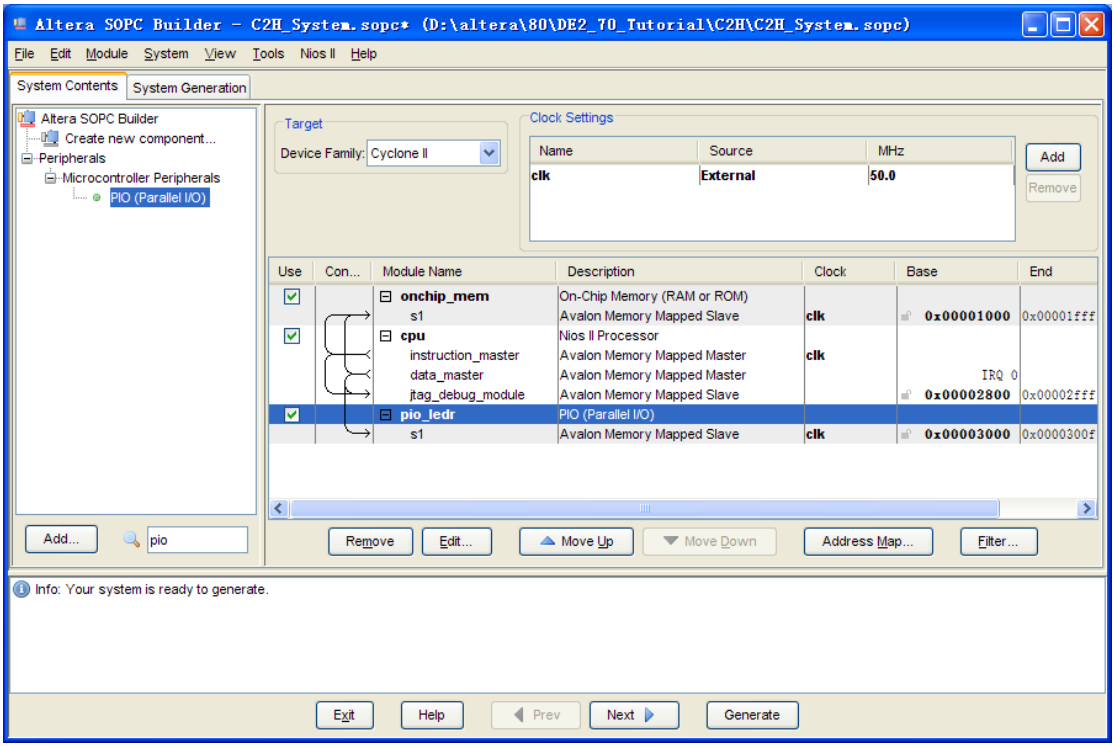


图 8-2　完成的 SOPC 工程

8. 生成系统。通过点击下方 Generate 完成。

8.3 如何用 Verilog 语言完成顶层实体

9. 这次具体讲解如何建立一个使用 Verilog 语言描述的顶层实体文件，之前所做的都是直接给代码样例，现在具体讲解这些顶层实体代码是怎么来的。

10. 顶层实体只要干两件事：

（1）决定最顶层的输入输出使用哪个器件。

（2）实例化之前建立的模块。

11. 这次实验的输出只用到了 oLEDR，输入由于用到 SOPC，需要一个时钟信号与复位信号，时钟信号 DE2-70 开发板上常用的是 iCLK_50，复位信号常用 iKEY[0]。

12. 可以写一个顶层实体头如下：

```
module C2H (
input iCLK_50，
input [0:0] iKEY，
output [17:0] oLEDR
);
```

13. 打开之前 SOPC 建立的创建 C2H_System.v 文件，查找 module C2H_System，发现

```
module C2H_System (
// 1) global signals:
clk，
reset_n，

// the_pio_ledr
```

```
out_port_from_the_pio_ledr
)
;
```

这就是需要在顶层实例化的模块了，

14. 将以上代码的 module 关键字去掉，随便起个实例名 u0，每个参数变成“.参数名(输入输出信号)”的样式，例如 clk 就改为.clk(iCLK_50)，修改代码如下

```
C2H_System u0 (
.clk(iCLK_50)，
.reset_n(iKEY[0])，
.out_port_from_the_pio_ledr(oLEDR[17:0])
);
```

15. 加上 endmodule，如此一来顶层实体就完成了。
16. 分析与综合
17. 分配引脚

```
set_location_assignment PIN_AD15 -to iCLK_50
set_location_assignment PIN_T29 -to iKEY[0]
set_location_assignment PIN_AJ6 -to oLEDR[0]
set_location_assignment PIN_AK5 -to oLEDR[1]
set_location_assignment PIN_AC13 -to oLEDR[10]
set_location_assignment PIN_AB13 -to oLEDR[11]
set_location_assignment PIN_AC12 -to oLEDR[12]
set_location_assignment PIN_AB12 -to oLEDR[13]
set_location_assignment PIN_AC11 -to oLEDR[14]
set_location_assignment PIN_AD9 -to oLEDR[15]
set_location_assignment PIN_AD8 -to oLEDR[16]
set_location_assignment PIN_AJ7 -to oLEDR[17]
set_location_assignment PIN_AJ5 -to oLEDR[2]
set_location_assignment PIN_AJ4 -to oLEDR[3]
set_location_assignment PIN_AK3 -to oLEDR[4]
set_location_assignment PIN_AH4 -to oLEDR[5]
set_location_assignment PIN_AJ3 -to oLEDR[6]
set_location_assignment PIN_AJ2 -to oLEDR[7]
set_location_assignment PIN_AH3 -to oLEDR[8]
set_location_assignment PIN_AD14 -to oLEDR[9]
```

18. 编译下载

8.4 软件设计

19. 打开 Nios II IDE，选择好合适的工作空间，新建空白工程。如图 8-3。

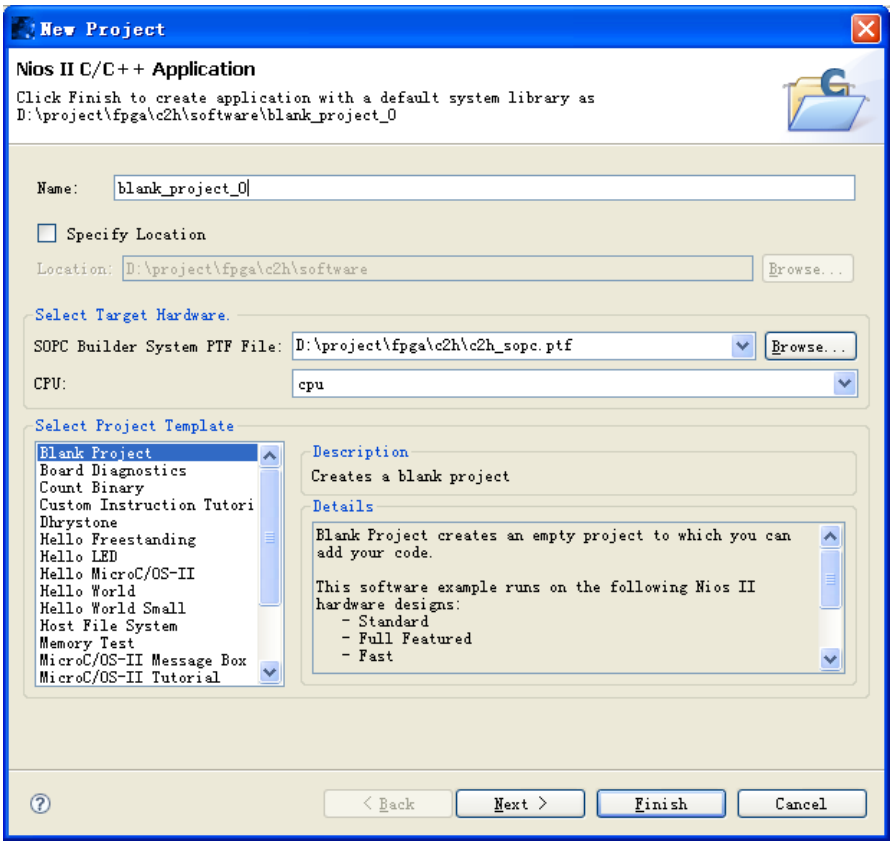


图 8-3 选择工程模板

20. 配置编译器参数为-DALT_RELEASE -Os -g –Wall
21. 配置 System Library Properties。
22. 新建 main.c，添加如下代码。

```
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "alt_types.h"


int main (void) __attribute__ ((weak，  alias ("alt_main")));

void delay (void)
{
      alt_u32 i=0，  j=0;
      while (i<100000)
            i++;
      while (j<100000)
            j++;
      return;
}

int alt_main (void)
{
   alt_u32 led = 0x2;
   alt_u8 dir = 0;
   /*
     * Infinitely shift a variable with one bit set back and forth，  and write
     * it to the LED PIO.    Software loop provides delay element.
     */
```

```
    while (1)
    {
      if (led & 0x00020001)
      {
        dir = (dir ^ 0x1);
      }

      if (dir)
      {
        led = led >> 1;
      }
      else
      {
        led = led << 1;
      }
      IOWR_ALTERA_AVALON_PIO_DATA(PIO_LEDR_BASE,  led);

      delay( );
      }

      return 0;
}
```

23. Project->Build All

24. 右击工程，Run As->Nios II Hardware，可以看到红色 LED 灯轮流闪烁。如果是以 Release 方式编译，那么闪烁将相对较快，如果以 Debug 方式编译，那么闪烁和使用 500ms 的 Timer 速度差不多。

8.5 C2H 编译器

这一节是本实验的重点。前面的所有操作基本上都同实验六跑马灯相一致。

25. 指定 function 用硬件实现

现在将 delay( )操作改为硬件实现，右击 delay( )选择 Accelerate with the Nios II C2H Compiler。如图 8-4。

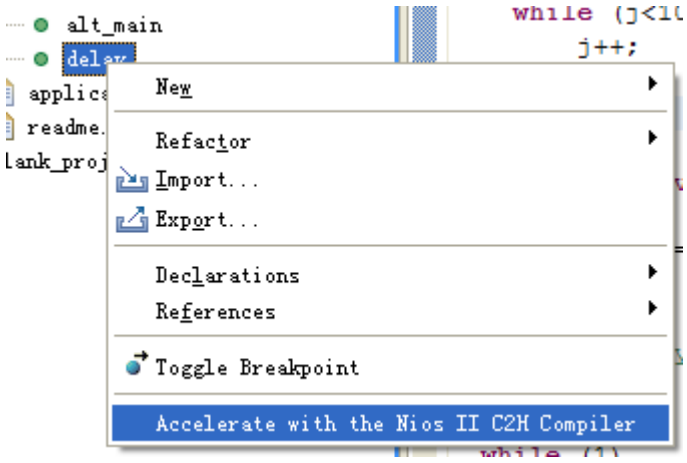


图 8-4 开启 C2H 编译器加速

26. 选了之后会在下方多出 C2H 的设定，选择“Build Software and generate SOPC Builder system”和“Use hardware accelerator in place of software implementation. Flush data cache before each call”。如图 8-5。

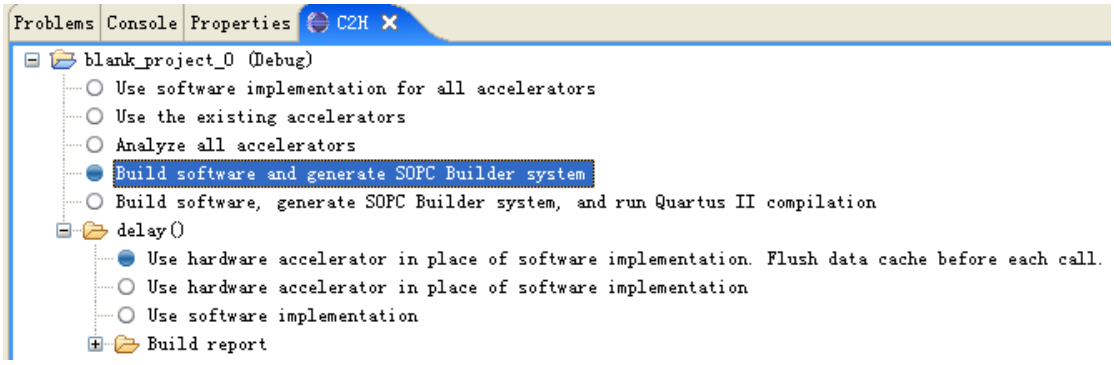


图 8-5 C2H 编译器加速选项（硬件加速）

27. Project->Build All，比纯编译软件要久，结果 576B，非常小。

28. 回到 Quartus II 查看 SOPC，发现确实生成了新的硬件模块，如图 8-6

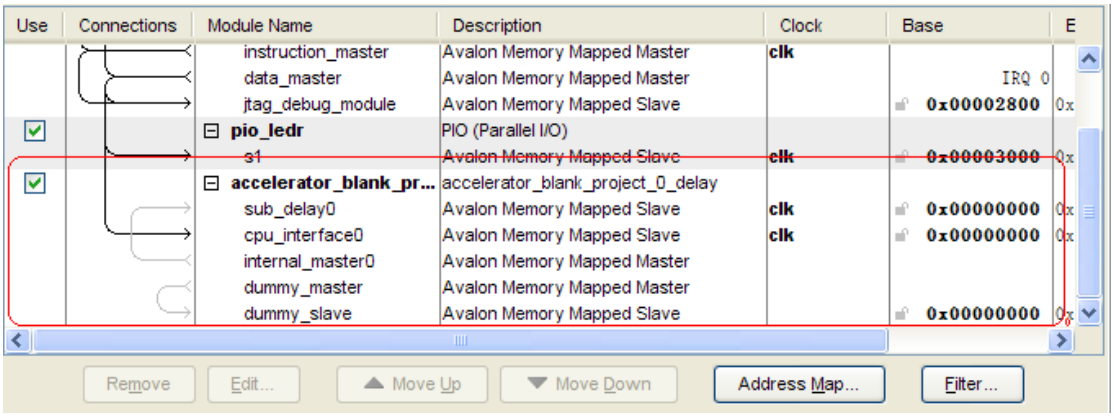


图 8-6 C2H 编译结果在 SOPC 中

29. 退出 SOPC，Quartus II 全编译，很遗憾生成的 sof 文件被受限，不是我们想要的 C2H.sof，而是 C2H_time_limited.sof，即表示下在到板上后无法长期使用。

这是因为我们的 Quartus 没有使用完全的 license，是破解版的。这也是之前为什么不选择“Build Software and generate SOPC Builder system，and run Quartus II compilation”而选择“Build Software and generate SOPC Builder system”的原因。

不过没有关系，这只是实验，不需要长期运行，像平时一样用 programmer 将其烧入 DE2-70 上运行即可。

30. 点击 Programmer，弹出警告对话框，如图 8-7。

图 8-7 使用 C2H 编译器后 Quartus II 编译结果受限警告

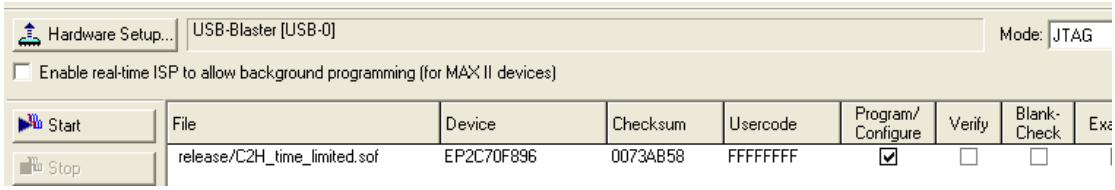


图 8-8　使用 C2H 编译器后 Quartus II 编译结果下载窗口

31. 参看图 8-8，点击 start 下载，出现 Error: The OpenCore Plus IP in device 1 is not responding。可以无视。

32. 回到 Nios II IDE 运行，发现灯全面点亮，这是因为原本每次加一都是 CPU 指令执行，现在是硬件自己完成，硬件实现比软件快很多。

33. 如欲关闭 C2H 硬件加速，在 C2H 的 View 里选择对应函数下面的 Use Software implementation 即可，如图 8-9。

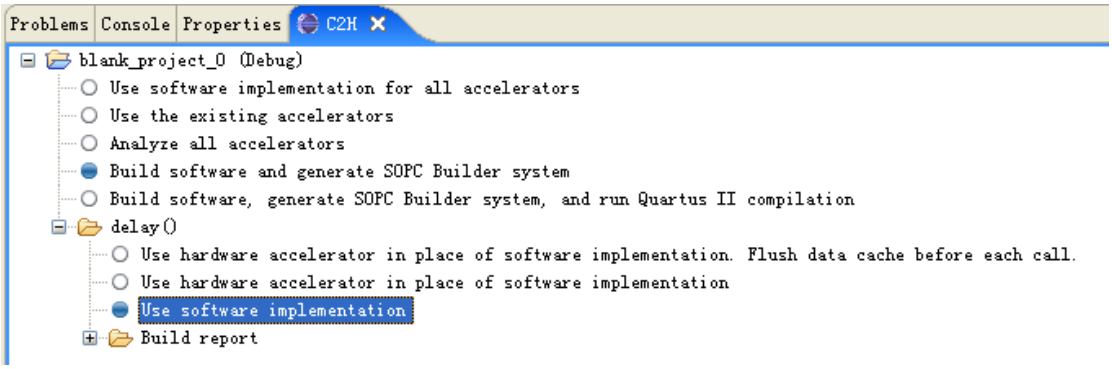


图 8-9 C2H 编译器加速选项（软件实现）

34. 小结：使用软件 Debug 方式，灯闪烁速度较慢；使用软件 Release 方式，灯闪烁速度较快，但肉眼可识别；使用 C2H 硬件方式，灯闪烁速度肉眼不可识别。

◆ 本实验指导结束

第 9 章　实验八 上电自动加载软硬件程序

● 实验说明

实际应用中往往需要上电时加载 FPGA 配置文件，然后自动加载软件运行，FPGA 配置文件可以通过 Quartus II Programmer 烧写在 EPCS 芯片中，软件同样可以通过 Nios II IDE Flash Programmer 烧写进板上带有的 Flash 或 EPCS 中。本实验以上电加载 Flash 中的跑马灯程序为例。本实验还演示了如何从已有的工程上再加工。

● 实验步骤

9.1 建立 Quartus 工程

1. 复制原来的 RunningLED 文件夹为 RunningLED_Flash。
2. 打开 RunningLED.qpf 文件进入工程
3. 重新设置编译输出目录为../ RunningLED_Flash/release
4. 删除原本编译结果

9.2 建立 SOPC 系统

5. 打开 SOPC Builder。
6. 添加 Bridges and Adapters ->Memory Mapped-> Avalon-MM Tristate Bridge：属性第一页 Incoming Signals 选择 Registered。如果是 8.0 可以直接 Finish，7.2 在第二页 Shared Signals 中将 data，address，read_n 都选上。如图 9-1。

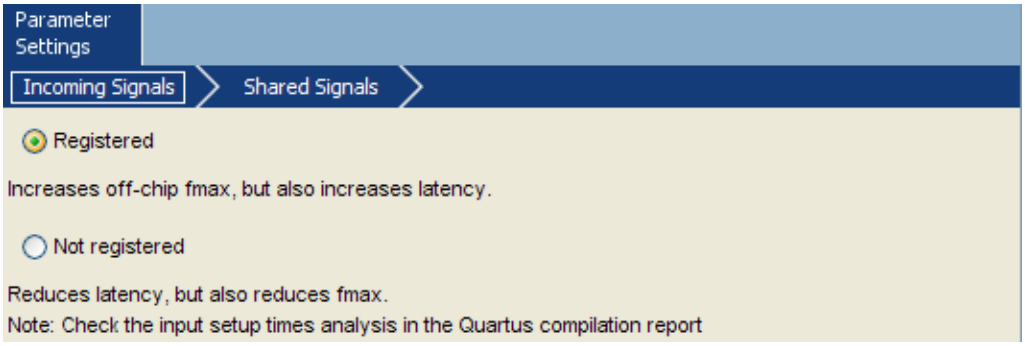


图 9-1 三态桥属性页

7. Memories and Memory Controller->Flash->Flash Memory(CFI)：属性第一页 Attributes 中的 Presets 选择 Custom。Size 中地址线宽度选择 22 位，数据线选择 16 位。第二页 Timing 中 Setup=0，Wait=100，Hold=0，Unit=ns。如图 9-2 与图 9-3。

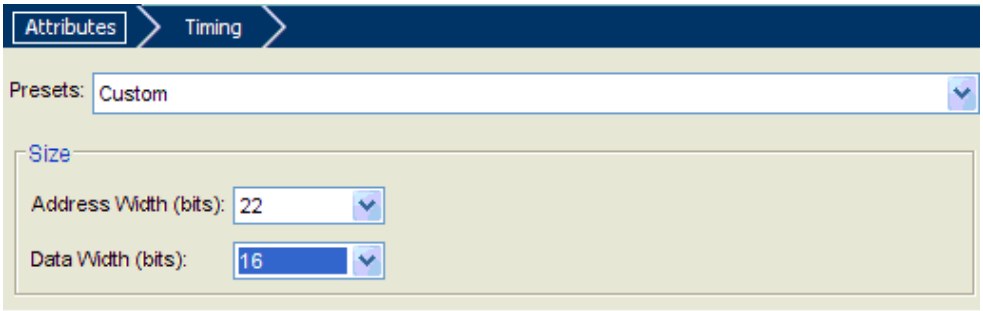


图 9-2 Flash 属性页（1）

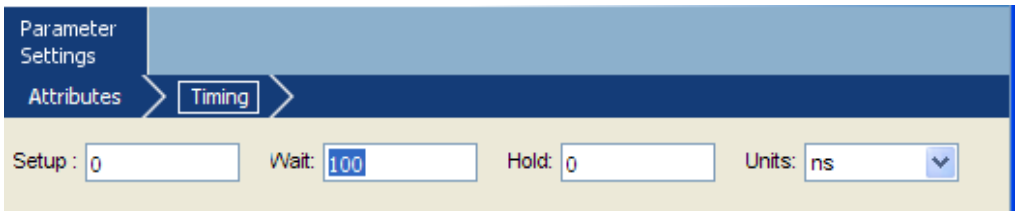


图 9-3 Flash 属性页（2）

注意：这两页的配置数据出自友晶 DE2-70 的光盘上的样例。

8. 将 Avalon-MM Tristate Bridge 与 Flash 连起来，点击图 9-4 中红圈圈住的位置

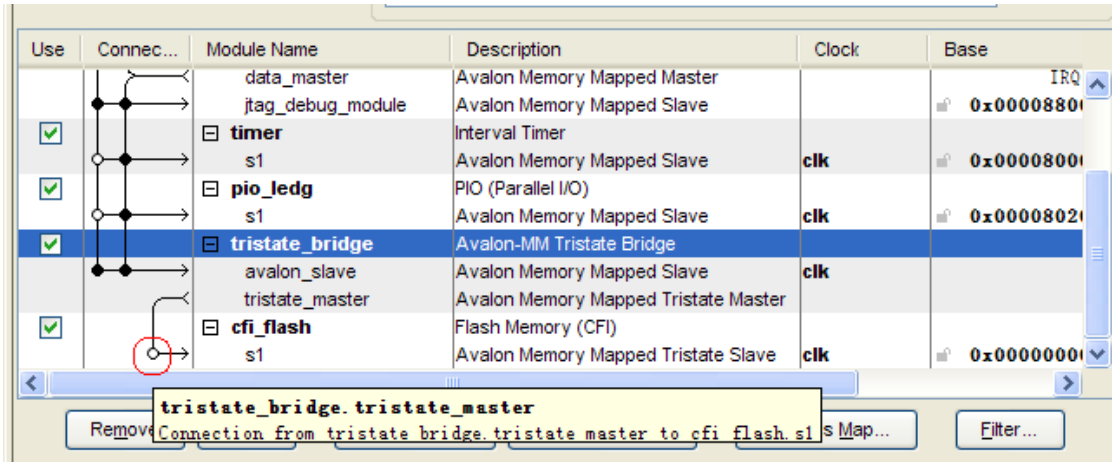


图 9-4　Flash 属性页

9. 完成后双击加入的 CPU，选择其 Reset Vector 在 cfi_flash 中。如图 9-5。

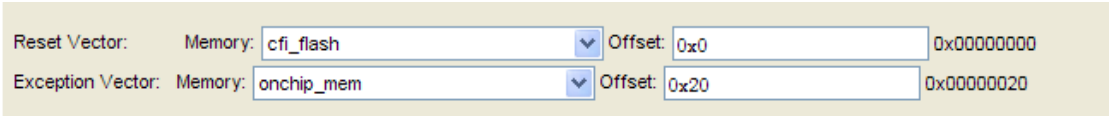


图 9-5　CPU 属性设置

10. 选择菜单栏上 System -> Auto-Assign Base Addresses 重新分配基址，去除基址错误，如图 9-6。

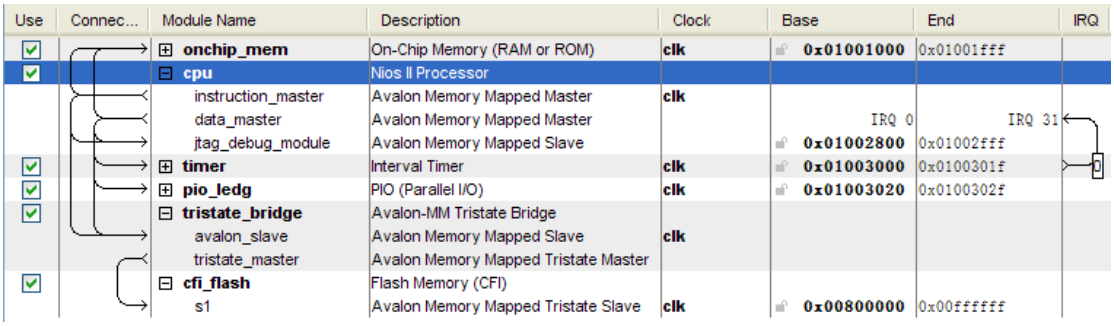


图 9-6 完成的 SOPC 工程

11. 生成系统。点击 Generate 按钮生成系统。

9.3 完成顶层实体

12. 回到 Quartus，把原来的顶层 bdf 文件移除，右击选择 Remove File from project

13. 建立 Verilog 语言程序的顶层实体，代码如下：

```
module RunningLED_Flash(
input [0:0] iKEY,
input iCLK_50,
```

```
output [7:0] oLEDG，
inout   [14:0] FLASH_DQ，              // FLASH Data bus 15 Bits (0 to 14)
inout          FLASH_DQ15_AM1，  // FLASH Data bus Bit 15 or Address A-1
output [21:0] oFLASH_A，      // FLASH Address bus 22 Bits
output         oFLASH_WE_N，  // FLASH Write Enable
output         oFLASH_RST_N，  // FLASH Reset
output         oFLASH_WP_N，  // FLASH Write Protect Programming Acceleration
input          iFLASH_RY_N，         // FLASH Ready/Busy output
output         oFLASH_BYTE_N，     // FLASH Byte/Word Mode Configuration
output         oFLASH_OE_N，         // FLASH Output Enable
output         oFLASH_CE_N       // FLASH Chip Enable
);

wire FLASH_16BIT_IP_A0;
assign oFLASH_BYTE_N = 1'b1; // FLASH Byte/Word Mode Configuration
assign oFLASH_RST_N    = 1'b1; // FLASH Reset
assign oFLASH_WP_N     = 1'b1; // FLASH Write Protect /Programming Acceleration

RunningLED_System u0(
.clk(iCLK_50)，
.reset_n(iKEY[0])，
.out_port_from_the_pio_ledg(oLEDG[7:0])，
//the_tristate_bridge_avalon_slave
.address_to_the_cfi_flash({oFLASH_A[21:0]，  FLASH_16BIT_IP_A0})，
// FLASH Address bus 26 Bits
.data_to_and_from_the_cfi_flash({FLASH_DQ15_AM1，  FLASH_DQ})，
// FLASH Data bus 15 Bits (0 to 14)
.read_n_to_the_cfi_flash(oFLASH_OE_N)，       // FLASH Output Enable
.select_n_to_the_cfi_flash(oFLASH_CE_N)，  // FLASH Chip Enable
.write_n_to_the_cfi_flash(oFLASH_WE_N)，     // FLASH Write Enable
);
Endmodule
```

注意：该顶层实体的建立牵扯到 Flash 的使用，简单使用的话，按照本例的代码用就可以了，如果想了解细节，请参考 DE2-70 光盘上关于 Flash 的 Datasheet。

14. 将其设为顶层实体并分配引脚

除原有引脚外，新加入 Flash 的引脚

```
set_location_assignment PIN_AF24 -to oFLASH_A[0]
set_location_assignment PIN_AG24 -to oFLASH_A[1]
set_location_assignment PIN_AH26 -to oFLASH_A[10]
set_location_assignment PIN_AJ26 -to oFLASH_A[11]
set_location_assignment PIN_AK26 -to oFLASH_A[12]
set_location_assignment PIN_AJ25 -to oFLASH_A[13]
set_location_assignment PIN_AK25 -to oFLASH_A[14]
set_location_assignment PIN_AH24 -to oFLASH_A[15]
```

```
set_location_assignment PIN_AG25 -to oFLASH_A[16]
set_location_assignment PIN_AF21 -to oFLASH_A[17]
set_location_assignment PIN_AD21 -to oFLASH_A[18]
set_location_assignment PIN_AK28 -to oFLASH_A[19]
set_location_assignment PIN_AE23 -to oFLASH_A[2]
set_location_assignment PIN_AJ28 -to oFLASH_A[20]
set_location_assignment PIN_AE20 -to oFLASH_A[21]
set_location_assignment PIN_AG23 -to oFLASH_A[3]
set_location_assignment PIN_AF23 -to oFLASH_A[4]
set_location_assignment PIN_AG22 -to oFLASH_A[5]
set_location_assignment PIN_AH22 -to oFLASH_A[6]
set_location_assignment PIN_AF22 -to oFLASH_A[7]
set_location_assignment PIN_AH27 -to oFLASH_A[8]
set_location_assignment PIN_AJ27 -to oFLASH_A[9]
set_location_assignment PIN_Y29 -to oFLASH_BYTE_N
set_location_assignment PIN_AG28 -to oFLASH_CE_N
set_location_assignment PIN_AF29 -to FLASH_DQ[0]
set_location_assignment PIN_AE28 -to FLASH_DQ[1]
set_location_assignment PIN_AD29 -to FLASH_DQ[10]
set_location_assignment PIN_AC28 -to FLASH_DQ[11]
set_location_assignment PIN_AC30 -to FLASH_DQ[12]
set_location_assignment PIN_AB30 -to FLASH_DQ[13]
set_location_assignment PIN_AA30 -to FLASH_DQ[14]
set_location_assignment PIN_AE24 -to FLASH_DQ15_AM1
set_location_assignment PIN_AE30 -to FLASH_DQ[2]
set_location_assignment PIN_AD30 -to FLASH_DQ[3]
set_location_assignment PIN_AC29 -to FLASH_DQ[4]
set_location_assignment PIN_AB29 -to FLASH_DQ[5]
set_location_assignment PIN_AA29 -to FLASH_DQ[6]
set_location_assignment PIN_Y28 -to FLASH_DQ[7]
set_location_assignment PIN_AF30 -to FLASH_DQ[8]
set_location_assignment PIN_AE29 -to FLASH_DQ[9]
set_location_assignment PIN_AG29 -to oFLASH_OE_N
set_location_assignment PIN_AH28 -to oFLASH_RST_N
set_location_assignment PIN_AH30 -to iFLASH_RY_N
set_location_assignment PIN_AJ29 -to oFLASH_WE_N
set_location_assignment PIN_AH29 -to oFLASH_WP_N
```

15. 进行全编译。编译完成后将程序烧写到 FPGA。

9.4 软件设计

16. 打开 Nios II IDE 进行软件设计。将工作空间切换到../RunningLED_Flash /software。
17. 检查编译器参数，编译工程并运行测试之

9.5 软件固化

18. 打开 Tool -> Flash Programmer。如图 9-7 所示。

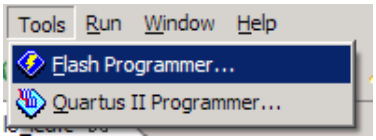


图 9-7 Flash Programmer

19. 打开后，如图 9-8 所示。

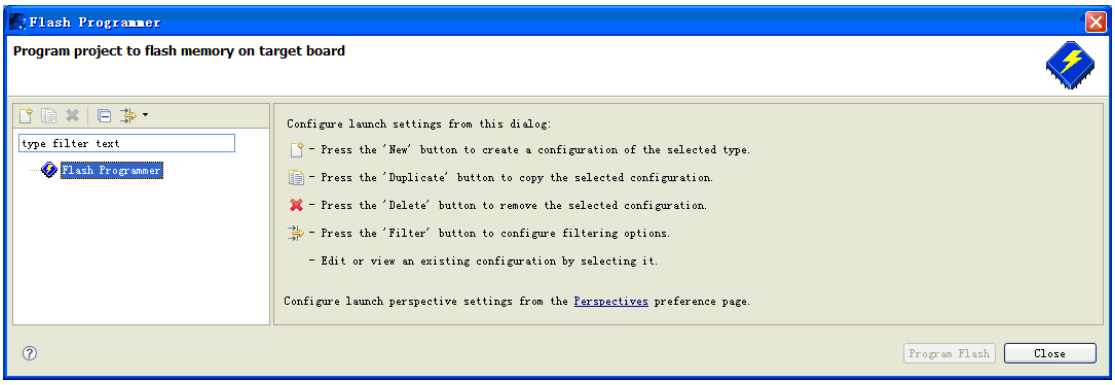


图 9-8 Flash Programmer 界面

20. 左侧 new 一个项目，随便填写个名称比如 RunningLED，Apply 刷新，如图 9-9

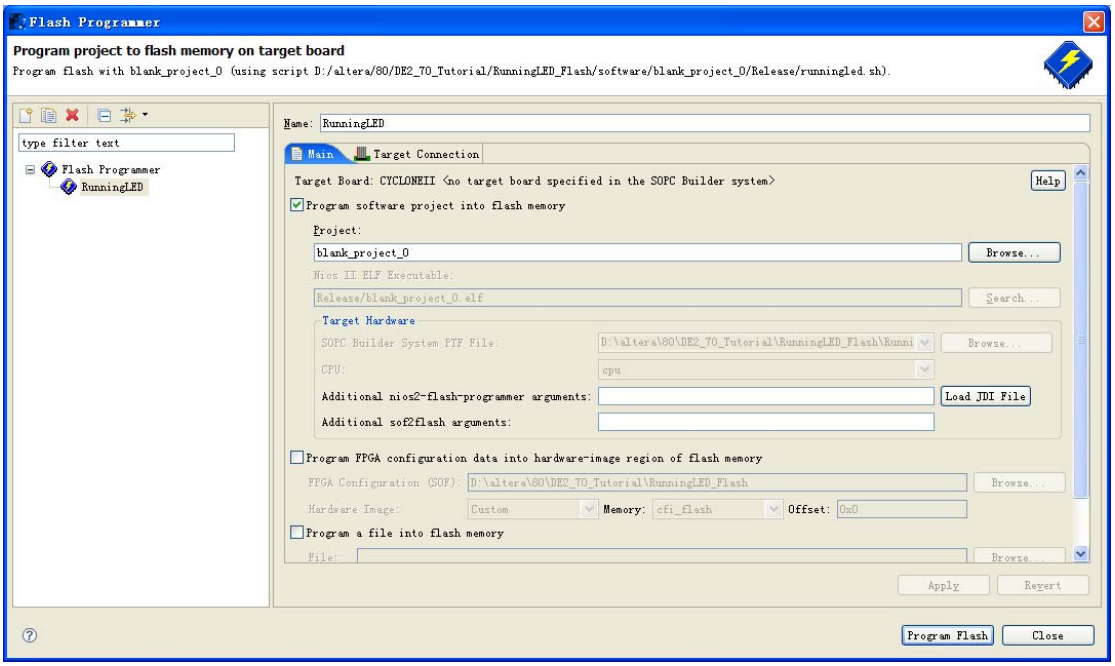


图 9-9　Flash Programmer 新建任务

21. 点击 Program Flash 就可以将程序烧入 Flash。完成后如图 9-10。

```
Problems  Console  Properties
<terminated> RunningLED [Flash Programmer] runningled.sh (09-6-28 下午3:16)
#!/bin/sh
#
# This file was automatically generated by the Nios II IDE Flash Programmer.
#
# It will be overwritten when the flash programmer options change.
#

cd D:/altera/80/DE2_70_Tutorial/RunningLED_Flash/software/blank_project_0/Releas
e

# Creating .flash file for the project
"$SOPC_KIT_NIOS2/bin/elf2flash" --base=0x00800000 --end=0xffffff --reset=0x80000
0 --input="blank_project_0.elf" --output="cfi_flash.flash" --boot="D:/altera/80/
ip/nios2_ip/altera_nios2/boot_loader_cfi.srec"

# Programming flash with the project
"$SOPC_KIT_NIOS2/bin/nios2-flash-programmer" --base=0x00800000  "cfi_flash.flash
"
Using cable "USB-Blaster [USB-0]", device 1, instance 0x00
Resetting and pausing target processor: OK

              : Checksumming existing contents

00000000      : Reading existing contents

Checksummed/read 7kB in 0.1s

00000000 ( 0%): Erasing

Erased 8kB in 0.0s

00000000 ( 0%): Programming

Programmed 2KB +6KB in 0.1s
Device contents checksummed OK
Leaving target processor paused
```

图 9-10 烧写程序到 Flash 图

22. 关闭 Nios II IDE，断电后再打开，烧入 FPGA 的配置程序就可以看到程序直接运行了。

9.6 FPGA 配置固化

23. 现在加电启动后依旧需要烧入 FPGA 配置文件方可引导软件程序，如果把 sof 文件也固化掉，那么加电启动后就可以脱离开发主机直接运行你所设计的程序。

方法归纳为三步：

（1）把 DE2-70 开发板左侧的开关从 RUN 拨到 PROG

（2）在平时经常使用的 Programmer 窗口中，把 mode 从 JTAG 改为 Active Serial Programming，遇到一个提示窗口选是确认。如图 9-11。

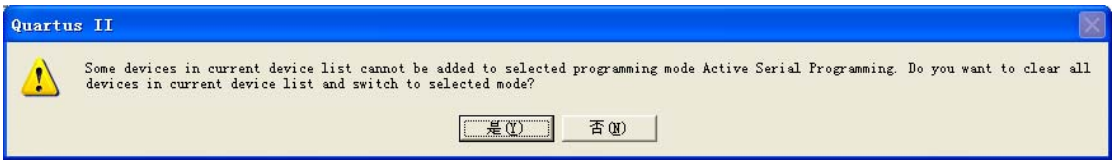


图 9-11 烧写 FPGA 配置提示窗口

（3）重新添加 pof 文件，不是 sof 文件，勾上 Program/Configure，start。如图 9-12

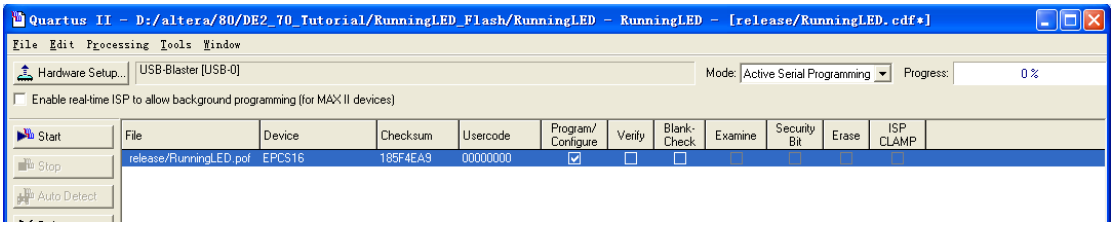


图 9-12 烧写 FPGA 配置到 Flash 图

24. 等待烧写完毕，直到右上角进度条 100%，状态栏显示 Ended 信息，如图 9-13。

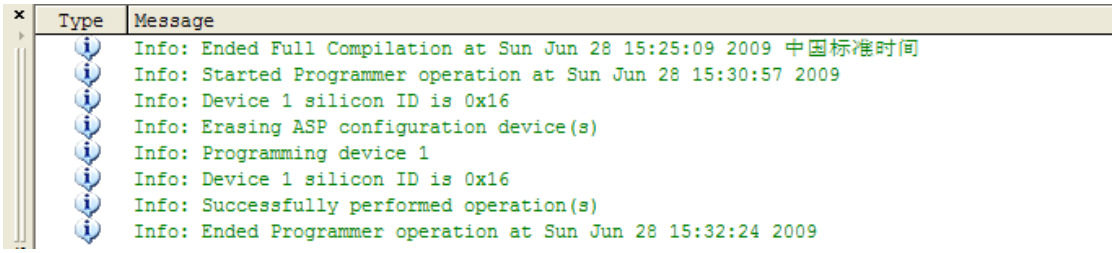


图 9-13 烧写 FPGA 配置到 Flash 完成

25. 把 DE2-70 开发板上的开关从 PROG 拨回 RUN。
26. 重新加电测试，如果忘了上一步，那么将会看到一排很暗的灯。
27. 如想恢复初始设置，烧入 DE2-70 光盘上的 DE2_70_Default 即可。

◆ 本实验指导结束

第 10 章　实验九　SDRAM 读写测试实验

● 实验说明

该实验主要完成对 SDRAM 读写的测试。主要讲解如何使用 SDRAM，由于 DE2-70 上的 SDRAM 是两片，所以比使用 Flash 稍微复杂一点点。通过本实验，读者应该了解不同器件对时钟的需求不同，并学会如何创建新的时钟。两片 SDRAM 的用法可以是统一使用，即只建立一个 SOPC 的 SDRAM 模块，数据宽 32 位，也可以分开使用，即建立两个 SOPC 的模块，数据宽度 16 位，本例是读写测试，需要知道两片各自独立的基址，故使用后一种用法。

● 实验步骤

10.1 建立 Quartus 工程

1. 建立一个新的工程 SDRAMTest。
2. 重新设置编译输出目录为../ SDRAMTest/release

10.2 建立 SOPC 系统

3. 打开 SOPC Builder，建立一个名为 SDRAMTest_System 的 SOPC 系统，并指定 Verilog 为描述系统的语言。
4. 在系统上添加 On-Chip Memory。大小设置 20k。
5. 添加 Nios II Processor。依旧选择 E 型。
6. 添加左侧的 Memories and Memory Controllers->SDRAM->SDRAM Controller：

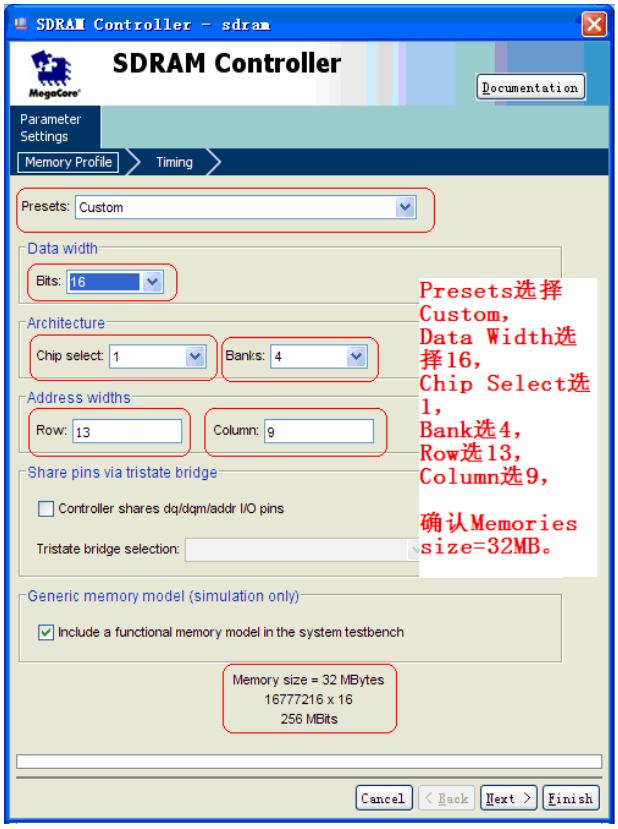


图 10-1 SDRAM Controller 设置第一页

配置第一页中，presets 选择 Custom，Data Width 选择 16，Chip Select 选 1，Bank4，Row 选 13，Column 选 9，确认 Memories size=32MB。第二页中，Issue one refresh command every 填 7.8125us，Delay after powerup， before initialization 填 200us。如图 10-1 与图 10-2。

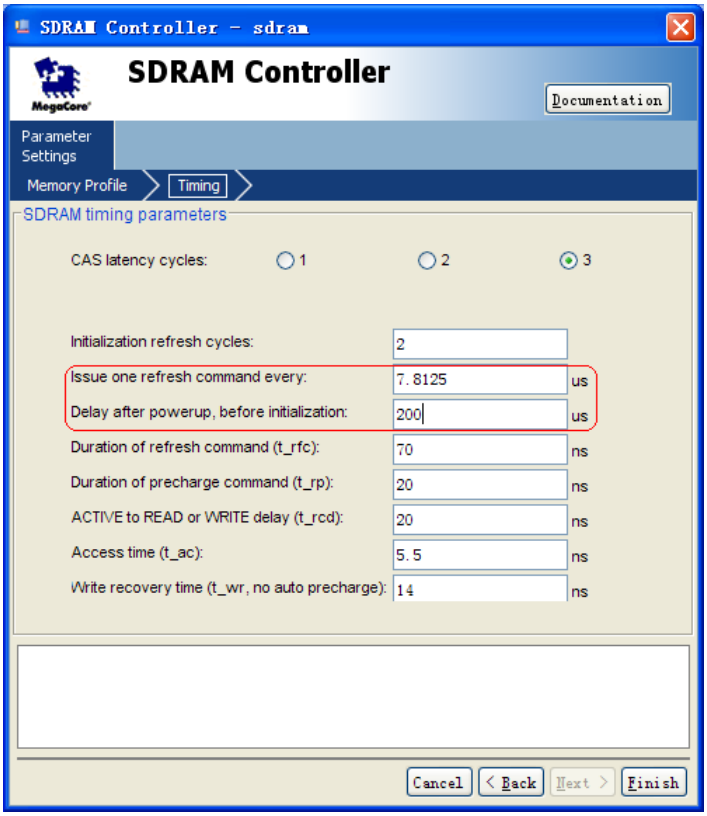


图 10-2 SDRAM Controller 设置第二页

注意：这两页的配置数据出自友晶 DE2-70 的光盘上的样例

7. 如此再添加一块 SDRAM，分别命名为 sdram_u1 与 sdram_u2。

8. 添加 jtag_uart，这个系统的输出连到 Nios II IDE 的 Console，需要 jtag_uart 支持。

9. 添加 pll，在左侧 PLL->PLL，在弹出的窗口中选择 Launch Altera's ALTPLL MegaWizard，如图 10-3。

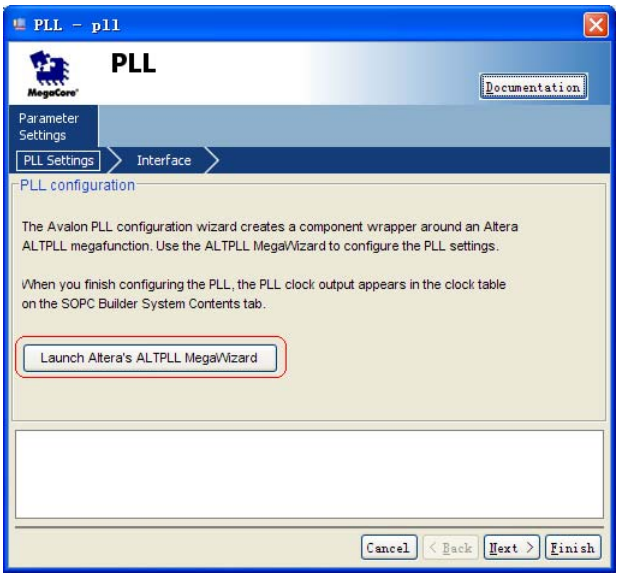


图 10-3　PLL 设置第一页

10. 弹出窗口如图 10-4 所示，单击 next 按钮。

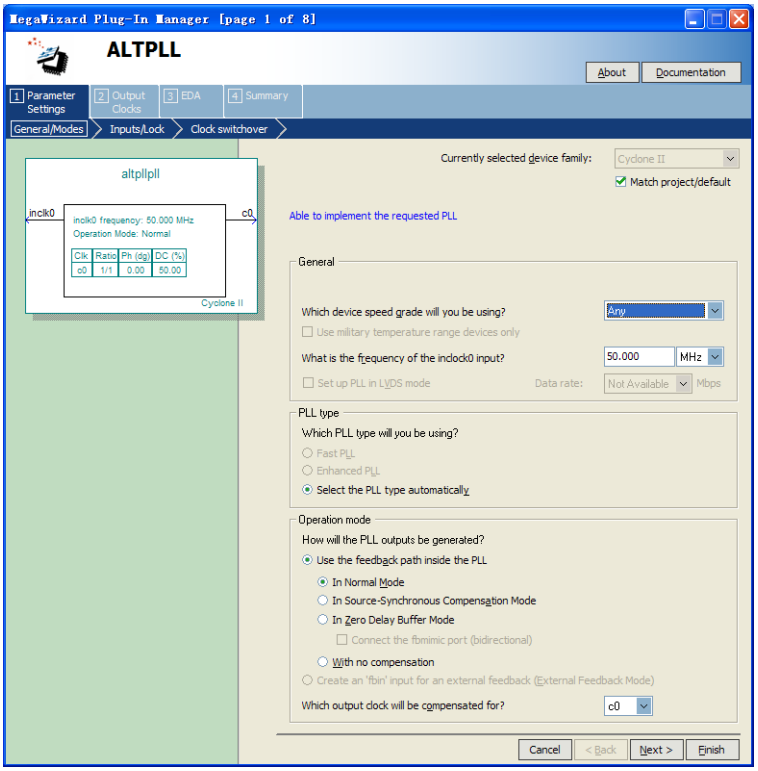


图 10-4　ALTPLL 设置第一页

11. 如图 10-5 所示，继续执行 next 按钮。

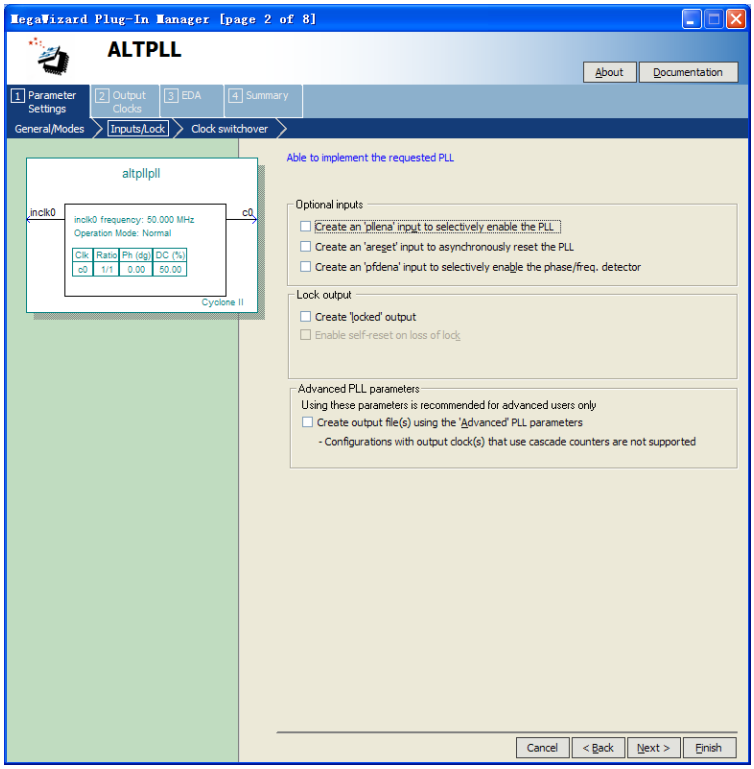


图 10-5　ALTPLL 设置第二页

12. 如图 10-6 所示，继续执行 next。

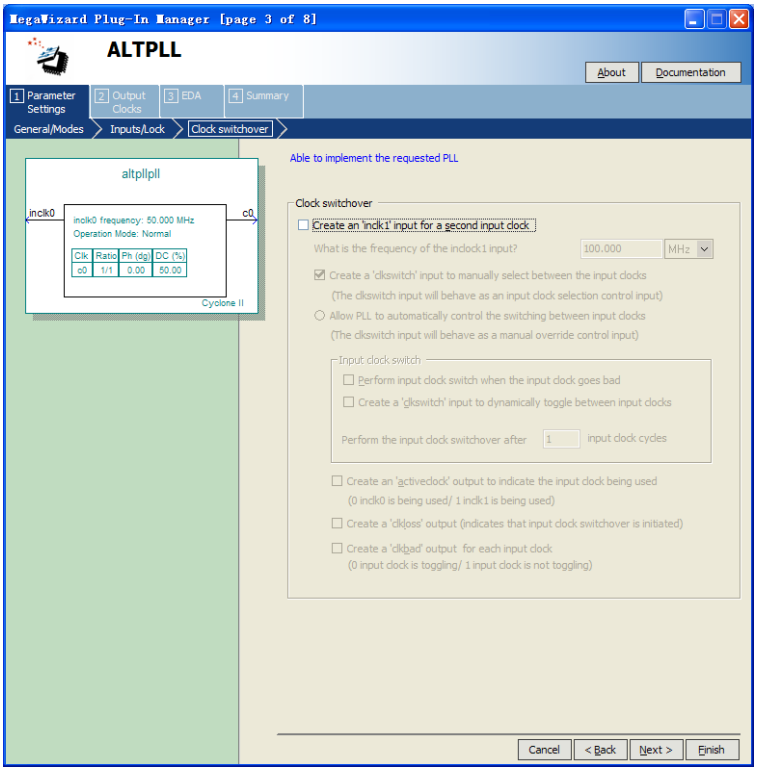


图 10-6　ALTPLL 设置第三页

13. 来到第一个输出时钟的设置，倍频选 2，即给系统时钟 100MHz，如图 10-7。第二输出时钟设为一个负 65 度相位的 100MHz 时钟，给 SDRAM，如图 10-8。

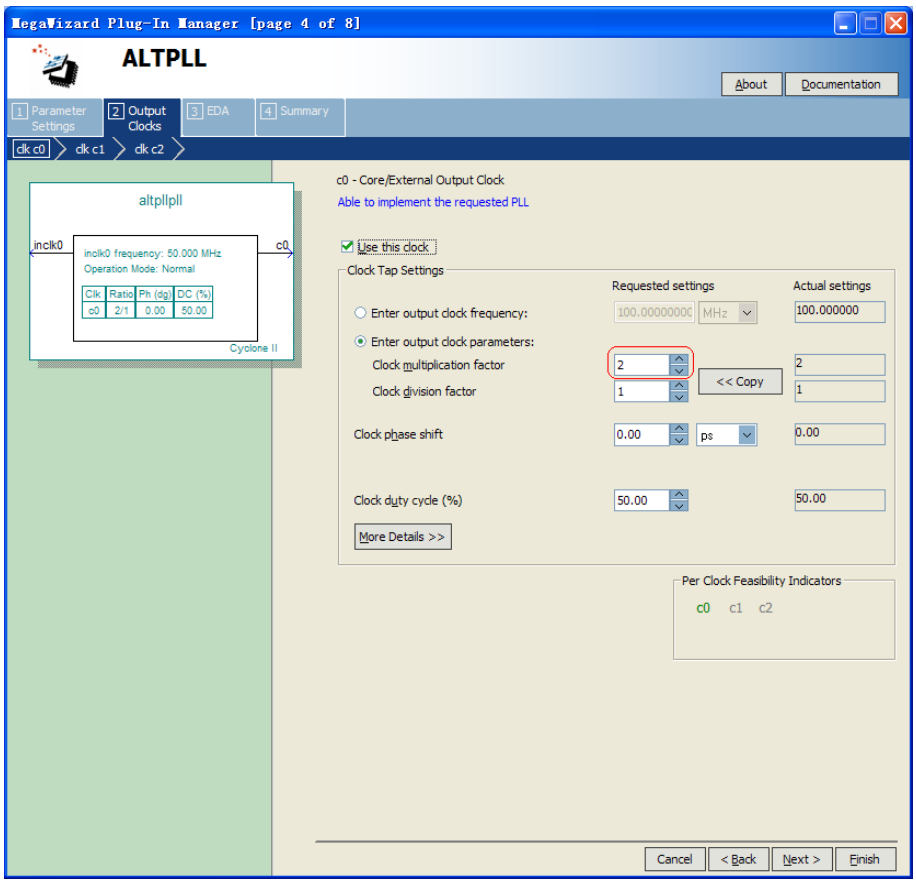


图 10-7　ALTPLL 设置第四页

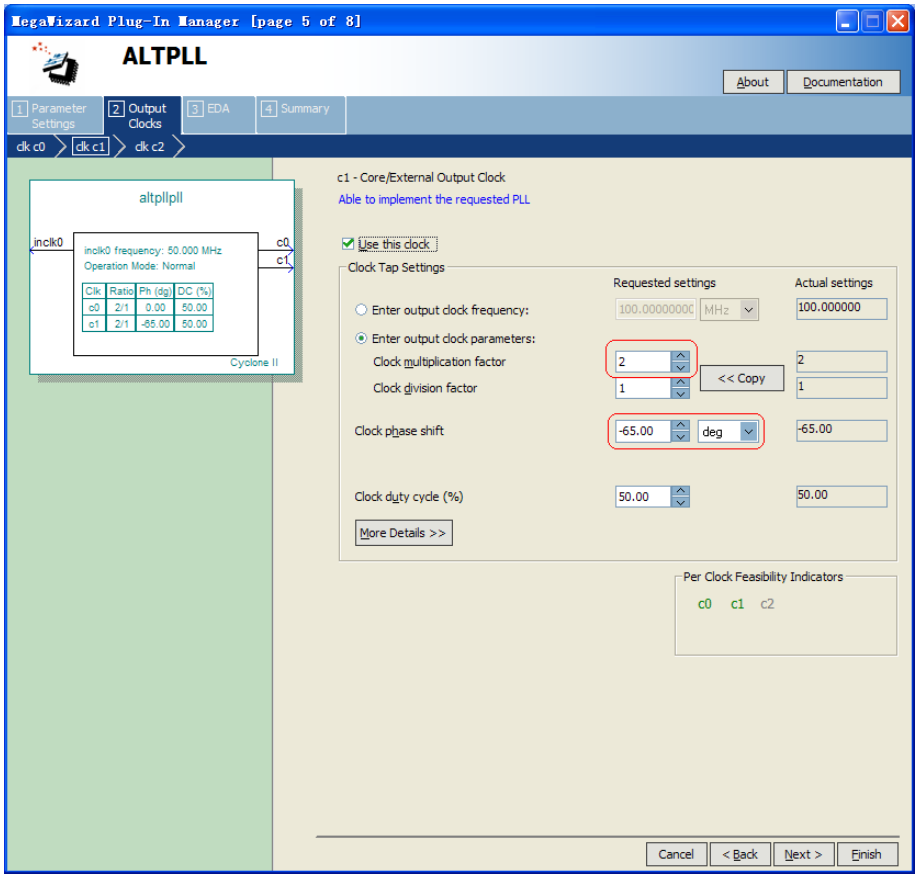


图 10-8　ALTPLL 设置第五页

14. 之后一路执行 next，直到 MegaWizard finish，回到 PLL 添加页面。如图 10-9 所示，单击 Finish。

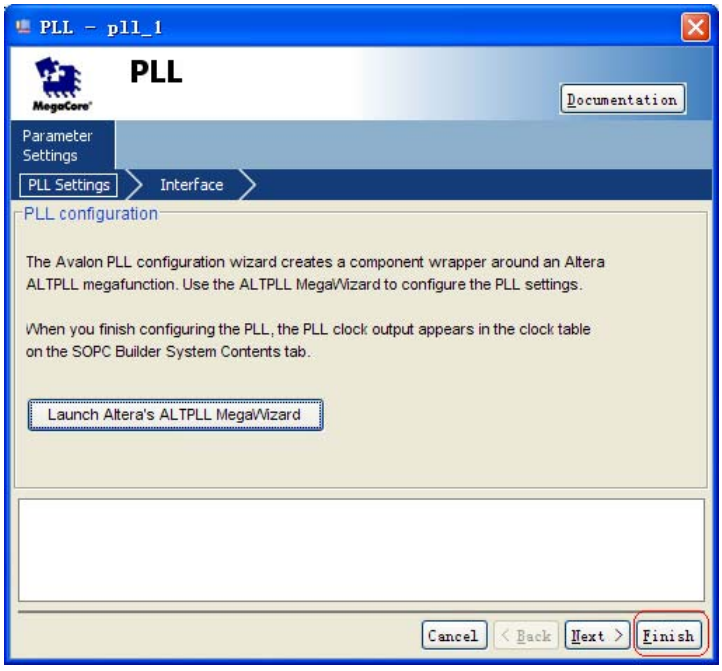


图 10-9　ALTPLL 设置第五页

15. 回到 SOPC 系统视图，在右上方时钟视图中右击对应时钟信号选择 Rename 修改时钟名字，clk 改为 clk_50，pll.c0 改为 pll.c0_system，pll.c1 改为 pll.c1_memory，并将除 pll 外各器件的时钟用下拉框选择系统时钟，如图 10-10.

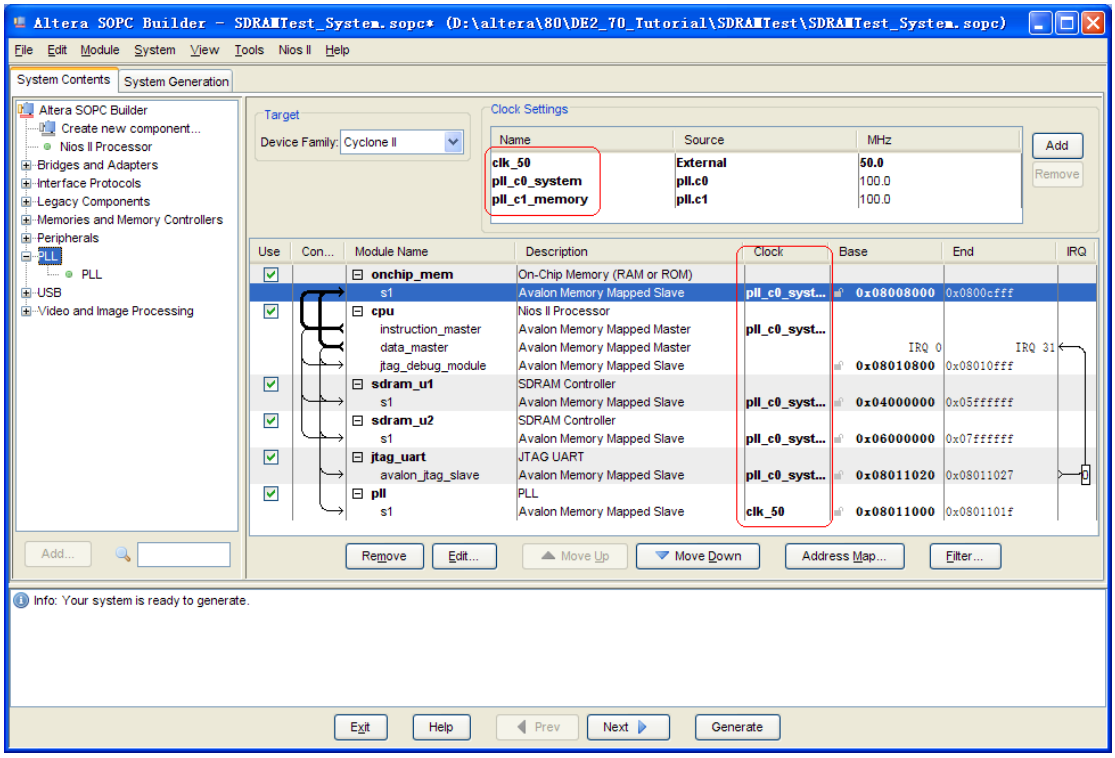


图 10-10　ALTPLL 设置第五页

16. System Auto-Assign Address 分配地址，generate 生成系统。

10.3 完成顶层实体

17. 添加以下代码到 SDRAMTest.v：

```
module SDRAMTest (
    input    iCLK_50,
    input    [0:0]    iKEY,
    inout    [31:0] DRAM_DQ,            // SDRAM Data bus 32 Bits
    output [12:0] oDRAM0_A,           // SDRAM0 Address bus 12 Bits
    output [12:0] oDRAM1_A,           // SDRAM1 Address bus 12 Bits
    output           oDRAM0_LDQM0,    // SDRAM0 Low-byte Data Mask
    output           oDRAM1_LDQM0,    // SDRAM1 Low-byte Data Mask
    output           oDRAM0_UDQM1,    // SDRAM0 High-byte Data Mask
    output           oDRAM1_UDQM1,    // SDRAM1 High-byte Data Mask
    output           oDRAM0_WE_N,     // SDRAM0 Write Enable
    output           oDRAM1_WE_N,     // SDRAM1 Write Enable
    output           oDRAM0_CAS_N,    // SDRAM0 Column Address Strobe
    output           oDRAM1_CAS_N,    // SDRAM1 Column Address Strobe
    output           oDRAM0_RAS_N,    // SDRAM0 Row Address Strobe
    output           oDRAM1_RAS_N,    // SDRAM1 Row Address Strobe
    output           oDRAM0_CS_N,     // SDRAM0 Chip Select
    output           oDRAM1_CS_N,     // SDRAM1 Chip Select
    output [1:0]    oDRAM0_BA,        // SDRAM0 Bank Address
    output [1:0]    oDRAM1_BA,        // SDRAM1 Bank Address
    output           oDRAM0_CLK,      // SDRAM0 Clock
```

```
    output          oDRAM1_CLK,         // SDRAM0 Clock
    output          oDRAM0_CKE,         // SDRAM0 Clock Enable
    output          oDRAM1_CKE          // SDRAM1 Clock Enable
);
wire CPU_CLK;

// the sdram is shahred with rtl and nios
assign  oDRAM1_CLK = oDRAM0_CLK;

SDRAMTest_System u0(
    //globalsignals:
    .clk_50(iCLK_50),
    .pll_c0_system(CPU_CLK),
    .pll_c1_memory(oDRAM0_CLK),
    .reset_n(iKEY[0]),

    // the_sdram (u1)
    .zs_addr_from_the_sdram_u1(oDRAM0_A),
    .zs_ba_from_the_sdram_u1(oDRAM0_BA),
    .zs_cas_n_from_the_sdram_u1(oDRAM0_CAS_N),
    .zs_cke_from_the_sdram_u1(oDRAM0_CKE),
    .zs_cs_n_from_the_sdram_u1(oDRAM0_CS_N),
    .zs_dq_to_and_from_the_sdram_u1(DRAM_DQ[15:0]),
    .zs_dqm_from_the_sdram_u1({oDRAM0_UDQM1，oDRAM0_LDQM0}),
    .zs_ras_n_from_the_sdram_u1(oDRAM0_RAS_N),
    .zs_we_n_from_the_sdram_u1(oDRAM0_WE_N),

    // the_sdram (u2)
    .zs_addr_from_the_sdram_u2(oDRAM1_A),
    .zs_ba_from_the_sdram_u2(oDRAM1_BA),
    .zs_cas_n_from_the_sdram_u2(oDRAM1_CAS_N),
    .zs_cke_from_the_sdram_u2(oDRAM1_CKE),
    .zs_cs_n_from_the_sdram_u2(oDRAM1_CS_N),
    .zs_dq_to_and_from_the_sdram_u2(DRAM_DQ[31:16]),
    .zs_dqm_from_the_sdram_u2({oDRAM1_UDQM1，oDRAM1_LDQM0}),
    .zs_ras_n_from_the_sdram_u2(oDRAM1_RAS_N),
    .zs_we_n_from_the_sdram_u2(oDRAM1_WE_N)
);
endmodule
```

注意：这种 SDRAM 用法属于两片分开使用，联合一起使用时顶层实体与此不同。

18. 分配引脚

```
set_location_assignment PIN_AD15 -to iCLK_50
set_location_assignment PIN_AC1 -to DRAM_DQ[0]
set_location_assignment PIN_AC2 -to DRAM_DQ[1]
```

```
set_location_assignment PIN_AF2 -to DRAM_DQ[10]
set_location_assignment PIN_AF3 -to DRAM_DQ[11]
set_location_assignment PIN_AG2 -to DRAM_DQ[12]
set_location_assignment PIN_AG3 -to DRAM_DQ[13]
set_location_assignment PIN_AH1 -to DRAM_DQ[14]
set_location_assignment PIN_AH2 -to DRAM_DQ[15]
set_location_assignment PIN_U1 -to DRAM_DQ[16]
set_location_assignment PIN_U2 -to DRAM_DQ[17]
set_location_assignment PIN_U3 -to DRAM_DQ[18]
set_location_assignment PIN_V2 -to DRAM_DQ[19]
set_location_assignment PIN_AC3 -to DRAM_DQ[2]
set_location_assignment PIN_V3 -to DRAM_DQ[20]
set_location_assignment PIN_W1 -to DRAM_DQ[21]
set_location_assignment PIN_W2 -to DRAM_DQ[22]
set_location_assignment PIN_W3 -to DRAM_DQ[23]
set_location_assignment PIN_Y1 -to DRAM_DQ[24]
set_location_assignment PIN_Y2 -to DRAM_DQ[25]
set_location_assignment PIN_Y3 -to DRAM_DQ[26]
set_location_assignment PIN_AA1 -to DRAM_DQ[27]
set_location_assignment PIN_AA2 -to DRAM_DQ[28]
set_location_assignment PIN_AA3 -to DRAM_DQ[29]
set_location_assignment PIN_AD1 -to DRAM_DQ[3]
set_location_assignment PIN_AB1 -to DRAM_DQ[30]
set_location_assignment PIN_AB2 -to DRAM_DQ[31]
set_location_assignment PIN_AD2 -to DRAM_DQ[4]
set_location_assignment PIN_AD3 -to DRAM_DQ[5]
set_location_assignment PIN_AE1 -to DRAM_DQ[6]
set_location_assignment PIN_AE2 -to DRAM_DQ[7]
set_location_assignment PIN_AE3 -to DRAM_DQ[8]
set_location_assignment PIN_AF1 -to DRAM_DQ[9]
set_location_assignment PIN_AA4 -to oDRAM0_A[0]
set_location_assignment PIN_AA5 -to oDRAM0_A[1]
set_location_assignment PIN_Y8 -to oDRAM0_A[10]
set_location_assignment PIN_AE4 -to oDRAM0_A[11]
set_location_assignment PIN_AF4 -to oDRAM0_A[12]
set_location_assignment PIN_AA6 -to oDRAM0_A[2]
set_location_assignment PIN_AB5 -to oDRAM0_A[3]
set_location_assignment PIN_AB7 -to oDRAM0_A[4]
set_location_assignment PIN_AC4 -to oDRAM0_A[5]
set_location_assignment PIN_AC5 -to oDRAM0_A[6]
set_location_assignment PIN_AC6 -to oDRAM0_A[7]
set_location_assignment PIN_AD4 -to oDRAM0_A[8]
set_location_assignment PIN_AC7 -to oDRAM0_A[9]
set_location_assignment PIN_AA9 -to oDRAM0_BA[0]
```

```
set_location_assignment PIN_AA10 -to oDRAM0_BA[1]
set_location_assignment PIN_W10 -to oDRAM0_CAS_N
set_location_assignment PIN_AA8 -to oDRAM0_CKE
set_location_assignment PIN_AD6 -to oDRAM0_CLK
set_location_assignment PIN_Y10 -to oDRAM0_CS_N
set_location_assignment PIN_V9 -to oDRAM0_LDQM0
set_location_assignment PIN_Y9 -to oDRAM0_RAS_N
set_location_assignment PIN_AB6 -to oDRAM0_UDQM1
set_location_assignment PIN_W9 -to oDRAM0_WE_N
set_location_assignment PIN_T5 -to oDRAM1_A[0]
set_location_assignment PIN_T6 -to oDRAM1_A[1]
set_location_assignment PIN_T4 -to oDRAM1_A[10]
set_location_assignment PIN_Y4 -to oDRAM1_A[11]
set_location_assignment PIN_Y7 -to oDRAM1_A[12]
set_location_assignment PIN_U4 -to oDRAM1_A[2]
set_location_assignment PIN_U6 -to oDRAM1_A[3]
set_location_assignment PIN_U7 -to oDRAM1_A[4]
set_location_assignment PIN_V7 -to oDRAM1_A[5]
set_location_assignment PIN_V8 -to oDRAM1_A[6]
set_location_assignment PIN_W4 -to oDRAM1_A[7]
set_location_assignment PIN_W7 -to oDRAM1_A[8]
set_location_assignment PIN_W8 -to oDRAM1_A[9]
set_location_assignment PIN_T7 -to oDRAM1_BA[0]
set_location_assignment PIN_T8 -to oDRAM1_BA[1]
set_location_assignment PIN_N8 -to oDRAM1_CAS_N
set_location_assignment PIN_L10 -to oDRAM1_CKE
set_location_assignment PIN_G5 -to oDRAM1_CLK
set_location_assignment PIN_P9 -to oDRAM1_CS_N
set_location_assignment PIN_M10 -to oDRAM1_LDQM0
set_location_assignment PIN_N9 -to oDRAM1_RAS_N
set_location_assignment PIN_U8 -to oDRAM1_UDQM1
set_location_assignment PIN_M9 -to oDRAM1_WE_N
set_location_assignment PIN_T29 -to iKEY[0]
```

19. 编译下载

10.4 软件设计

20. 打开 Nios II IDE 进行软件设计。将工作空间切换到../SDRAMTest /software。
21. 新建空白工程，添加以下 C 代码到 main.c。

```
#include "stdio.h"
#include "system.h"

int main( )
{
    short *sdram1=(short*)SDRAM_U1_BASE;
```

```
short *sdram2=(short*)SDRAM_U2_BASE;
int i;
int right;
short temp;

printf("testing sdram1... \n");
for(i=0;i<0x1000000;i++)
{
    *(sdram1+i)=0x55aa;
}
right=1;
for(i=0;i<0x1000000;i++)
{
    temp=*(sdram1+i);
    if(temp!=0x55aa)
    {
        right=0;
        printf("sdram1 test failed at %d"，i);
        break;
    }
}
if(right)
printf("sdram1 test ok!\n");

 printf("testing sdram2... \n");
 for(i=0;i<0x1000000;i++)
{
    *(sdram2+i)=0x55aa;
}
right=1;
for(i=0;i<0x1000000;i++)
{
    temp=*(sdram2+i);
    if(temp!=0x55aa)
    {
        right=0;
        printf("sdram2 test failed at %d"，i);
        break;
    }
}
if(right)
printf("sdram2 test ok!\n");

return 0;
```

```
}
```

22. 配置编译器参数为-DALT_RELEASE -Os -g –Wall。

23. 配置 System Library Properties。，去掉 Support C++，勾上 Small C library，确认右边使用的是 onchip_mem，没有使用两块 SDRAM，输入输出使用 jtag_uart，如图 10-11 所示。

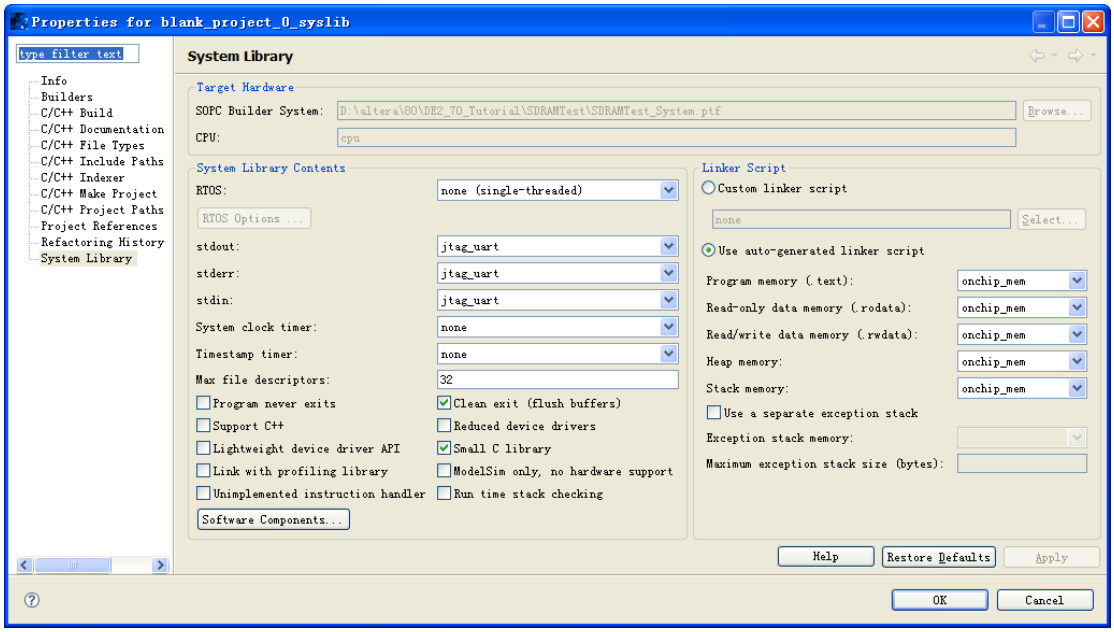


图 10-11　System Library 配置

24. 编译工程并运行之，结果正确。参看图 10-12。

```
testing sdram1...
sdram1 test ok!
testing sdram2...
sdram2 test ok!
```

图 10-12　100MHz 下的测试结果

25. 如图 10-13 所示。如果没有按照前文中指示的那样，修改时钟为 100MHz，而使用默认的 50MHz，则会出现下面这种情况：

```
testing sdram1...
sdram1 test ok!
testing sdram2...
sdram2 test failed at 5165
```

图 10-13　50MHz 下的测试结果

这主要是器件切换的时延和 CPU 指令执行的时延不匹配造成的。

◆ 本实验指导结束

第 11 章　基于 NIOS 的 μC/OS-II 实验

11.0 实验简介

● 实验说明

本实验项目使用 Quartus II、SOPC Builder、Nios II EDS 从零开始构建一个能够在 DE2-70 实验平台上运行的 μC/OS-II 操作系统的 Nios II 系统。初学者可以借此范例熟悉 Quartus II、SOPC Builder、Nios II EDS 的使用，并且了解基于 FPGA 的嵌入式系统开发流程。

● 资料来源

因特网网页：国立台湾大学王胜德教授关于软硬件协同设计的实验指导。
实验名：Altera SPOC Lab 1-uC/OS-II
主页：http://quest.ee.ntu.edu.tw/jenny/homepage/sdwang_codesign2008/index.html
其网站上只公开了这一网页：名称是 Lab1

● 实验环境

Quartus II 8.1 + Nios II EDS 8.1 + DE2-70 (Cyclone II EP2C70F896C6N)

● 本实验系统结构图

图 11-1 给出了本实验十（NIOS + UCOS-II）的系统结构图。

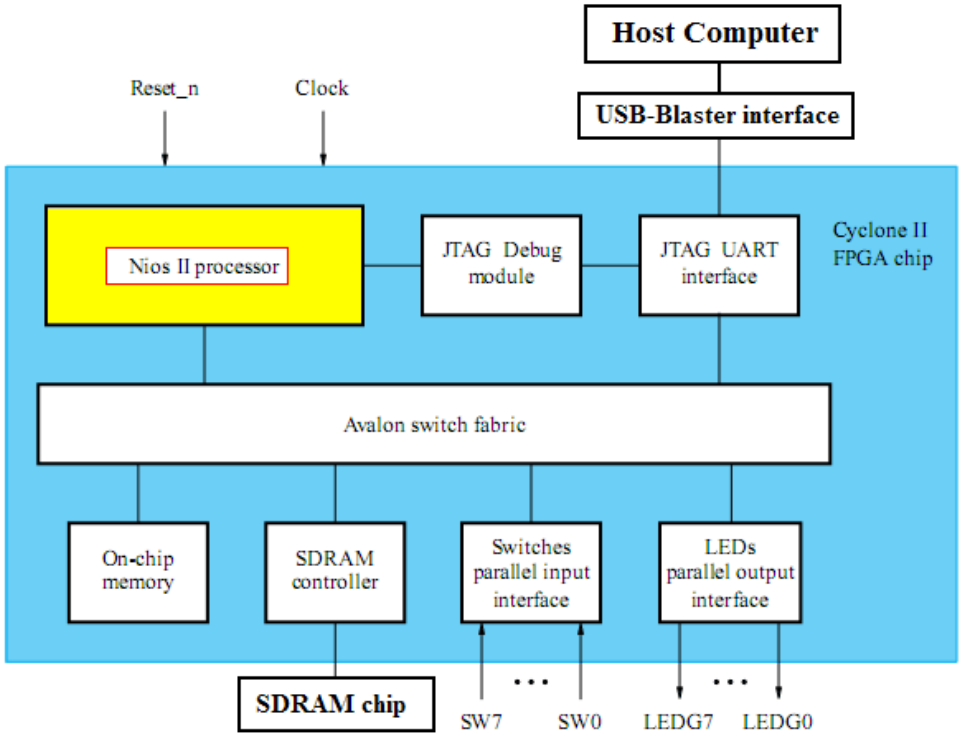


图 11-1　NIOS + UCOS-II 实验的系统结构图

从图 11-1 中可以看到 FPGA 芯片上有 NiosII 软核处理器、Avalon 交换结构（片上总线）、JTAG 调试模块、LED 并行输出接口、并行输入开关接口、SDRAM 控制器和片上存

储器。

外部输入器件采用 SW[17:0]。外部输出器件采用 LEDG[7:0]。最后的实验結果希望在 μC/OS-II 下实现多任务执行（多任务执行），并且 LEDG[17:0]能透过软件被 SW[17:0]控制。

● 本实验特色

实验者从零开始建立一个基于 Nios II 的 μC/OS-II 应用实验系统（也可以认为是一个 Nios II+μC/OS-II 的应用框架）具有以下一些作用。

1. 读者可以借助 SOPC Builder 工具自行对 Nios II 软核处理器进行配置。

2. 很多范例都是纯硬件的 Verilog 代码，需要自行从零开始建立 Nios II 系统，不能够直接使用 Altera 或友晶科技公司已经建立好的 Nios II 系统。

3. DE2-70 并非 Altera 公司原创的开发板，而是友晶科技 ODM 的电路板，很多外围设备都与 Altera 提供的电路板不一样，所以很多 Altera 手册中范例都无法执行，必须要有自己从硬件到软件建立系统的能力，将来才有办法将 Altera 提供的范例移植到 DE2-70 上执行并做到最佳化。

● 实验步骤

11.1 使用 Quartus II 建立一个新工程

Step 1：

建立一个新 project，如图 11-2 所示。

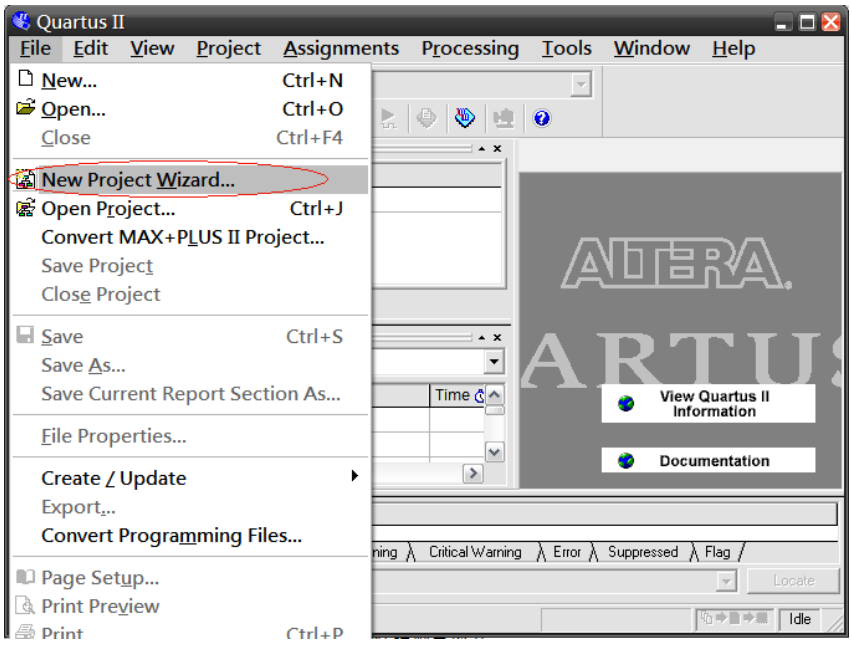


图 11-2　创建一个新工程

Step 2：

接下去 Quartus II 给出“**新建工程向导：简介**”显示框，按 Next 继续。如图 11-3 所示。

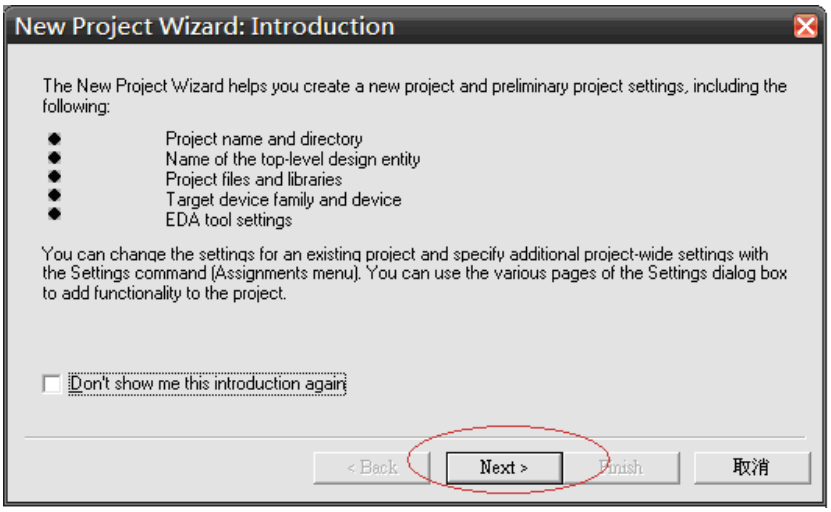


图 11-3　新建工程向导：简介

Step 3：

输入 project 路径名称、project 名称与 top module 名称，按 Next 继续。如图 11-4 所示。

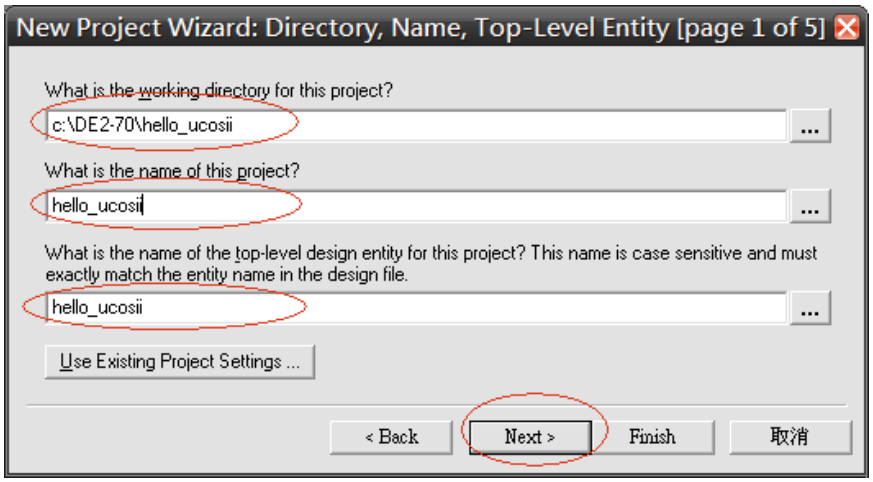


图 11-4　输入工程路径名/工程名/顶层设计实体对话框

Step 4：

C:/DE2-70/hello_ucosii 目录尚未建立，是否建立此目录。按是(Y)继续。如图 11-5 所示。

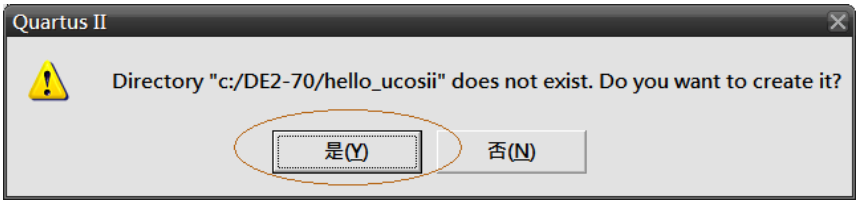


图 11-5　询问是否创建新文件夹

Step 5：

这一步将加入现有文件夹到 project。由于我们目前还没有建立任何文件夹，所以按 Next 继续。如图 11-6 所示。

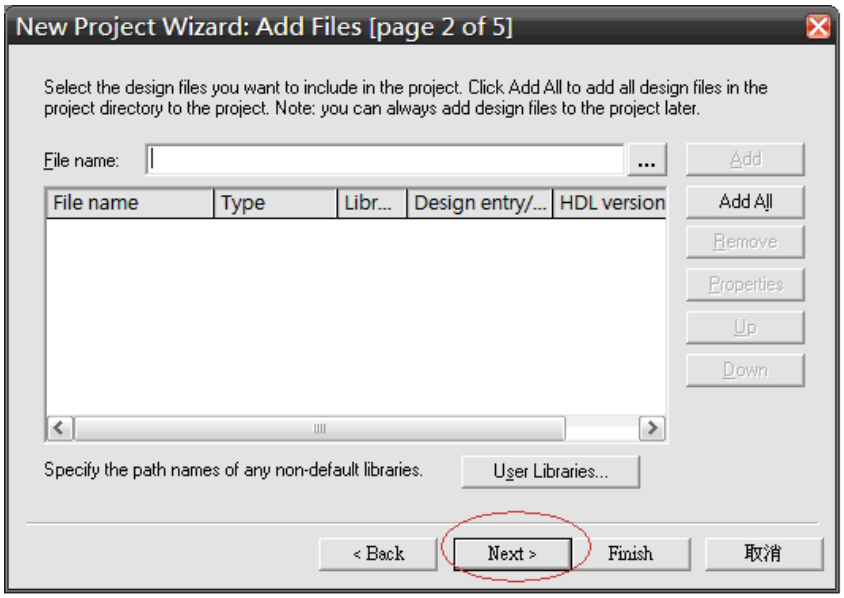


图 11-6　添加代码文件对话框

Step 6：

选择 FPGA 型号。如图 11-7 所示。

图 11-7　指定使用的 FPGA 器件

DE2-70 使用的 FPGA 是 Cyclone II EP2C70F896C6N，由 Altera 对 FPGA 的命名规则可得知：

芯片系列：Cyclone II；

封装：FBGA；

引脚数：896

速度级别：6

得到的芯片名称是：EP2C70F896C6

按 Next 继续。

Step 7：

选择第三方的 EDA 工具。Quartus II 支持第三方的 EDA 工具，如 ModelSim，若有用到可在此设定，目前没用到，按 Next 继续。如图 11-8 所示。

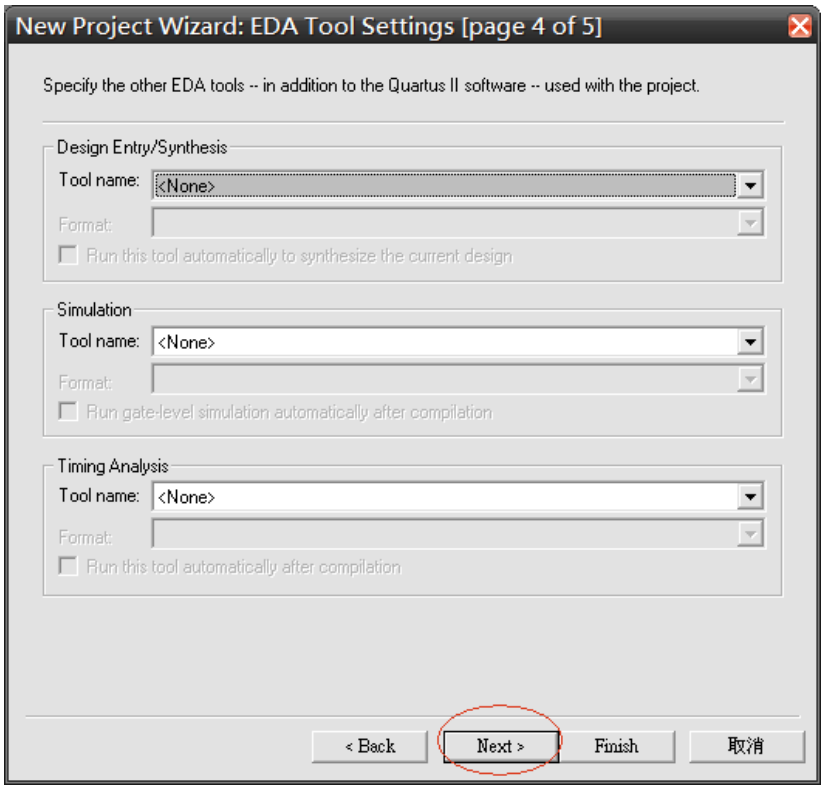


图 11-8　指定 EDA 工具

Step 8：

最后给出小结（Summary）对话框。实验者阅读之后可以按“Finish”按钮结束。如图 11-9 所示。

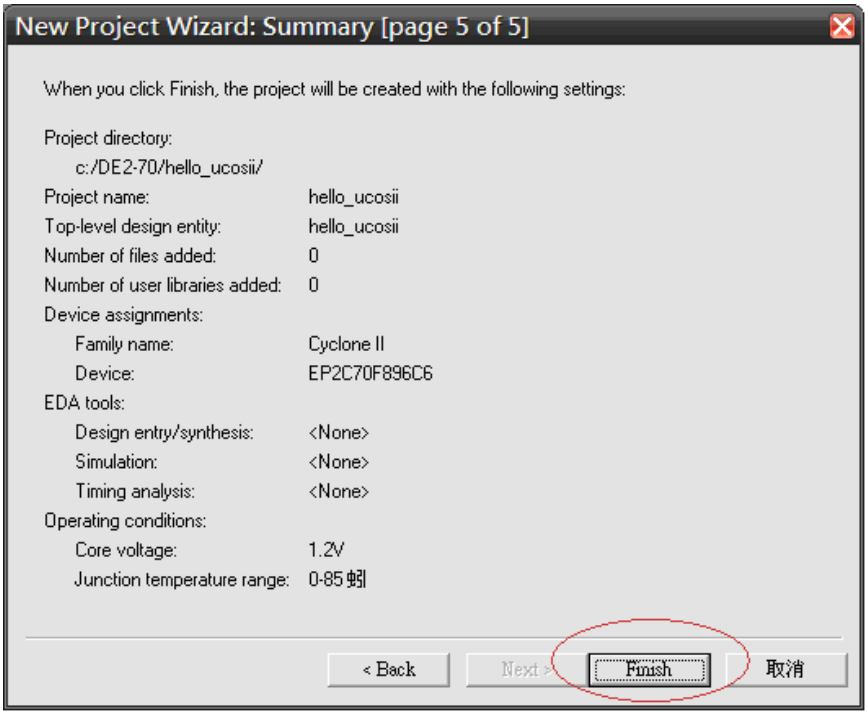


图 11-9　工程建立总结显示框

11.2 使用 SOPC Builder 建立 SOPC 系统

下面讲解如何使用 SOPC Builder 建立一个全新的 Nios II 系统。

Step 9：

单击 Quartus II 7.2/8.0 界面的 SOPC 图标，启动 SOPC Builder。参看图 11-10。

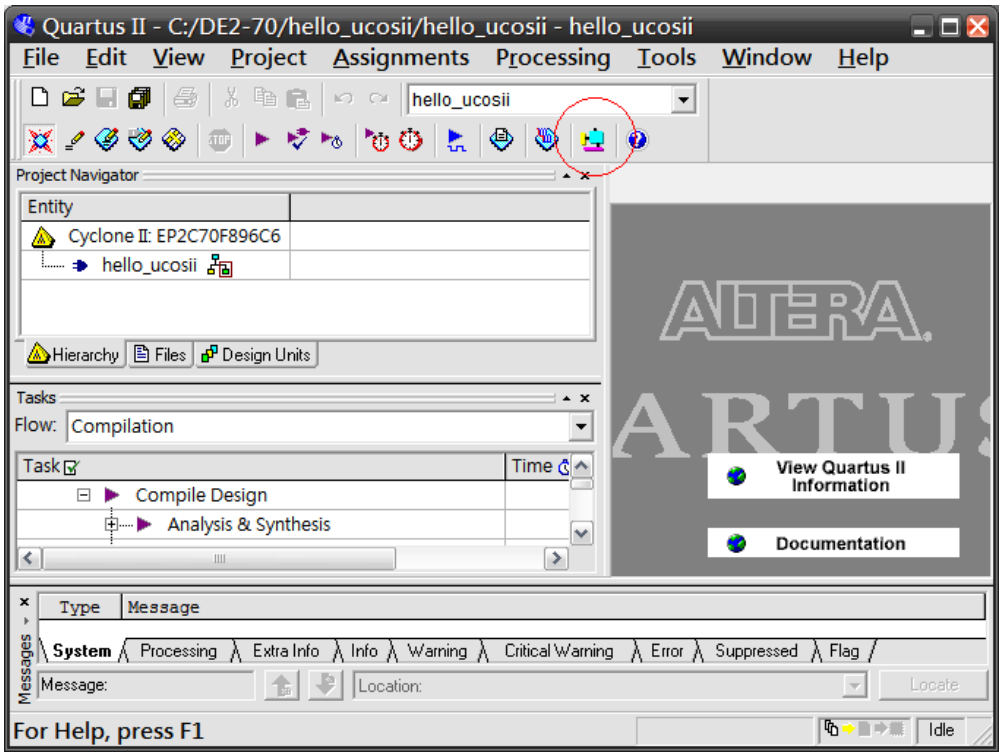


图 11-10　启动 SOPC Builder 的按钮

Step 10：

输入 System name，并选择 Verilog。参看图 11-11。选择 Verilog，表示 SOPC Builder

会用 Verilog 代码写成的 IP 核构建实验者的系统。如果实验者熟悉 VHDL，也可以选择 VHDL。

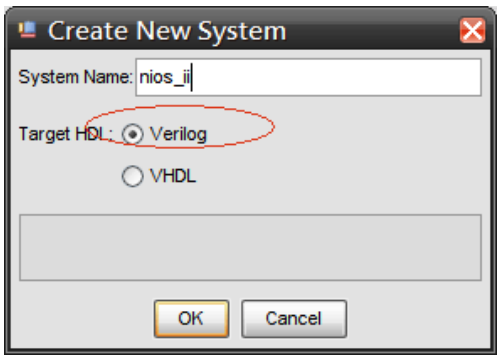


图 11-11　指定 SOPC 系统的名称

这里的设定不限制读者日后只能用 Verilog 或 VHDL 写 code，因为 Quartus II 本来就允许你 Verilog 与 VHDL 混合编程，也就是說 Verilog 的 module 可以使用 VHDL 的 entity，VHDL 的 entity 可以使用 Verilog 的 module，最后都能顺利编译。

参看图 11-12，注意左上角 Device Family 为 Cyclone II，且右上角 clk_0 为 50.0Mhz，虽然 Nios II 在 DE2-70 可以只跑 50.0MHz，但这等于是 CPU 降频在跑，正常情况下，Nios II 在 DE2-70 可以跑 100.0MHz，所以我们接着打算用 PLL 将 clk 倍频成 100.0Mhz。

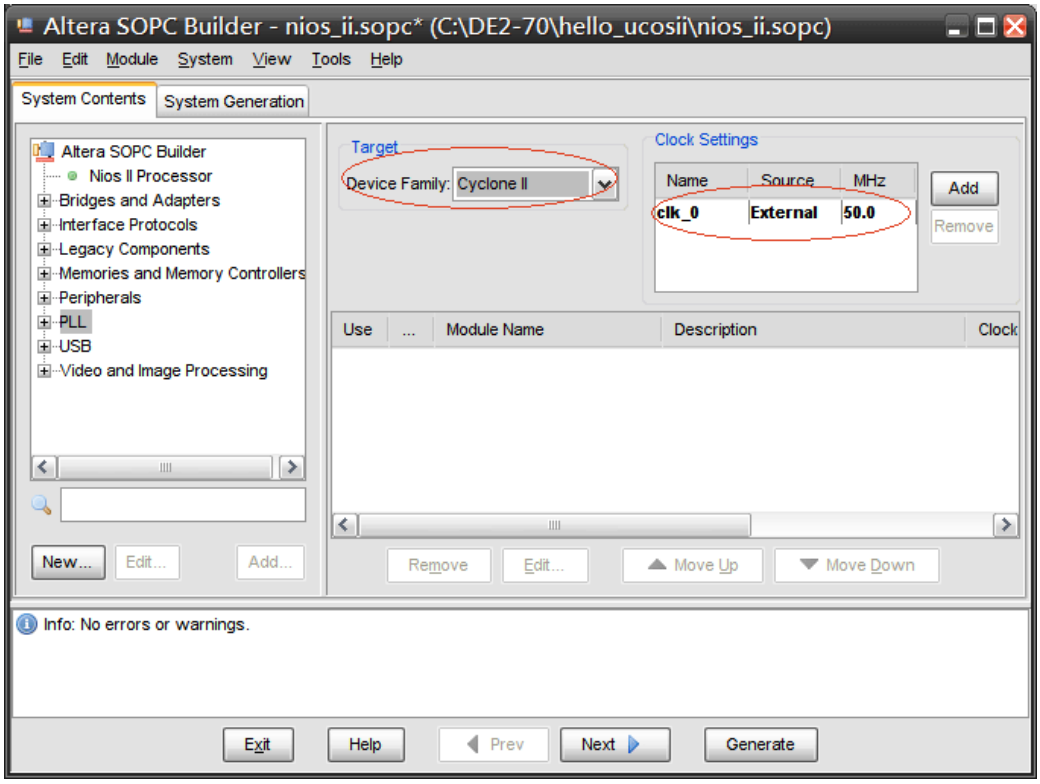


图 11-12　clk_0 的时钟频率为 50.0MHz

11.3 向 SOPC 添加锁相环 PLL

Step 11：

加入 PLL（锁相环，控制处理器主频速率），产生 Nios II CPU 与 SDRAM 所需要的时钟信号 clk。在 SOPC Builder 主界面的 System Contents 标签页的左侧，用鼠标对准 PLL 选

项双击，加入 SOPC。参看图 11-13。

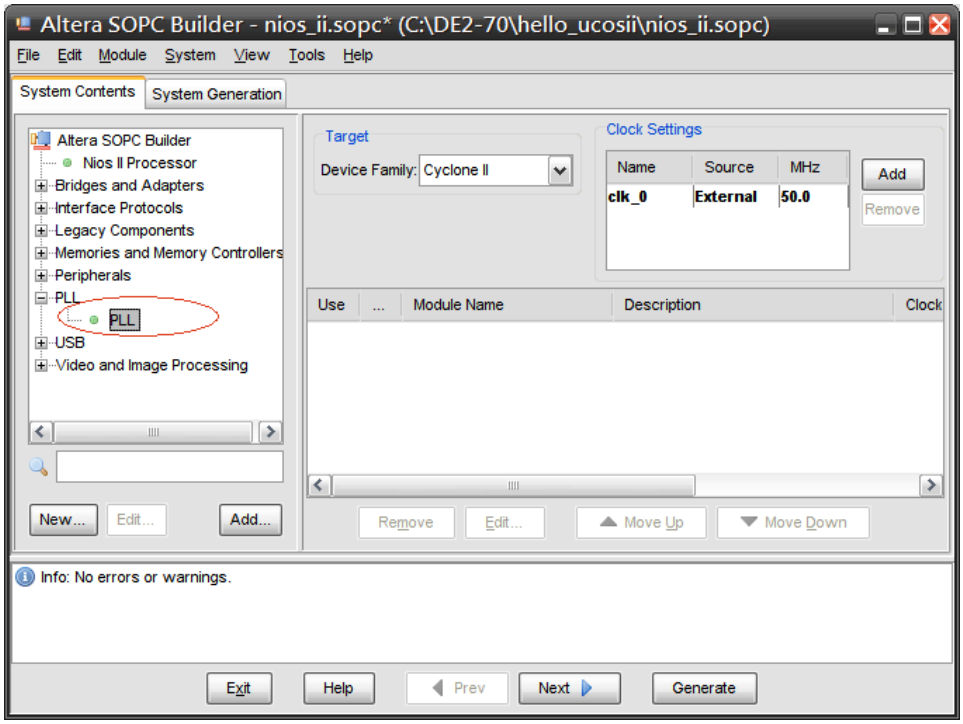


图 11-13　选中 PLL 组件

随后，在 SOPC 配置清单上会有 PLL 条目。参看图 11-14。

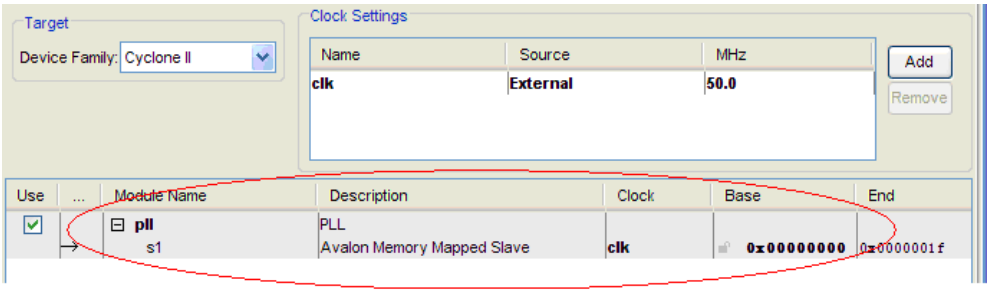


图 11-14　pll 组件进入 SOPC 的组件配置清单

此时在图 11-14 中可以看到 pll 已经进入了 SOPC 组件清单。与此同时，Quartus II 系统会弹出一个 PLL 设置的起始对话框。参看图 11-15。

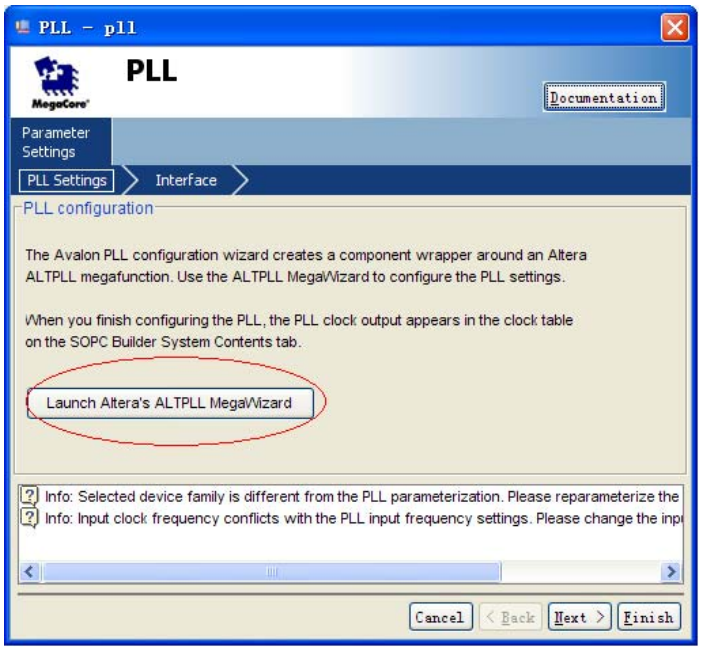


图 11-15　PLL 配置向导对话框

在图 11-15 所示的对换框中按下“Launch Altera's ALTPLL MegaWizard”。

◆ALTPLL 设置 Page1

参看图 11-16 示出的对话框，这是 PLL 设置的 8 个对话框的第 1 个。接受缺省值即可，单击“Next”继续。

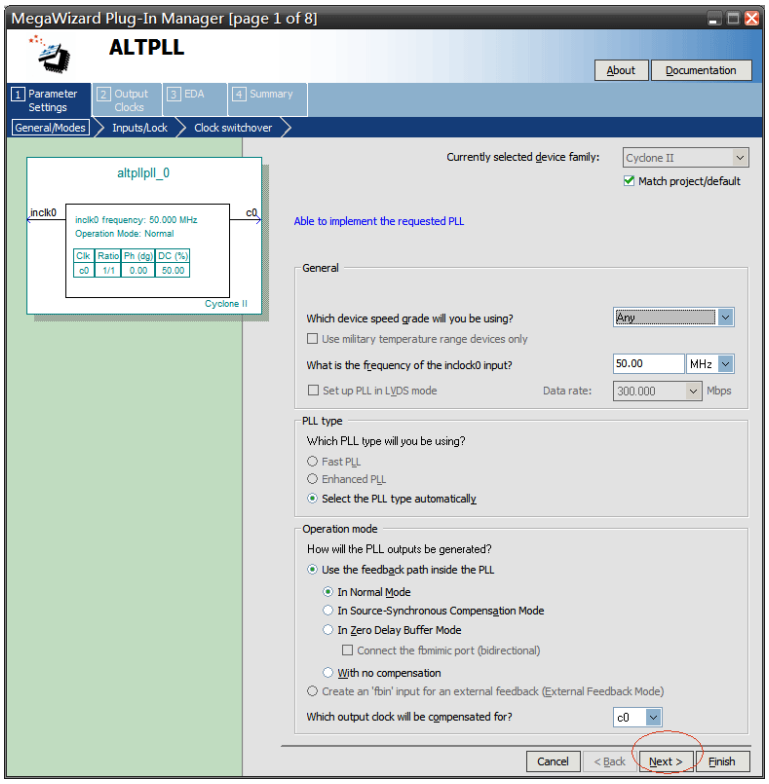


图 11-16　PLL 设置对话框之一

ALTPLL 设置 Page2

系统给出 PLL 设置对话框之二，参看图 11-17。实验者接受缺省值即可，按 Next 继续。

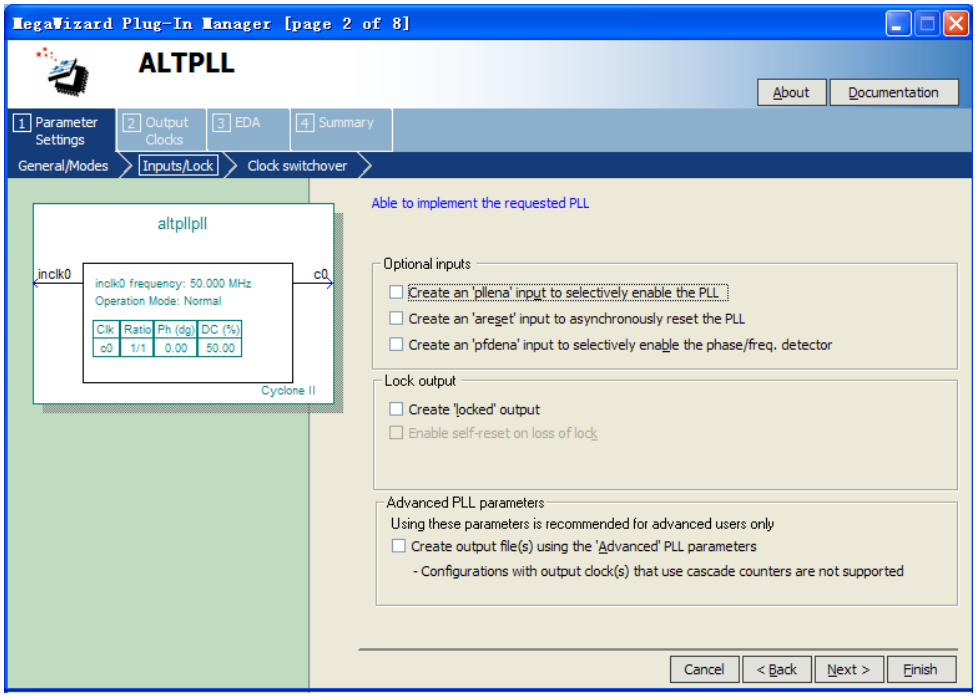


图 11-17　PLL 设置对话框之二

ALTPLL 设置 Page3

系统给出 PLL 设置对话框之三，参看图 11-18。接受缺省值即可，按 Next 继续。

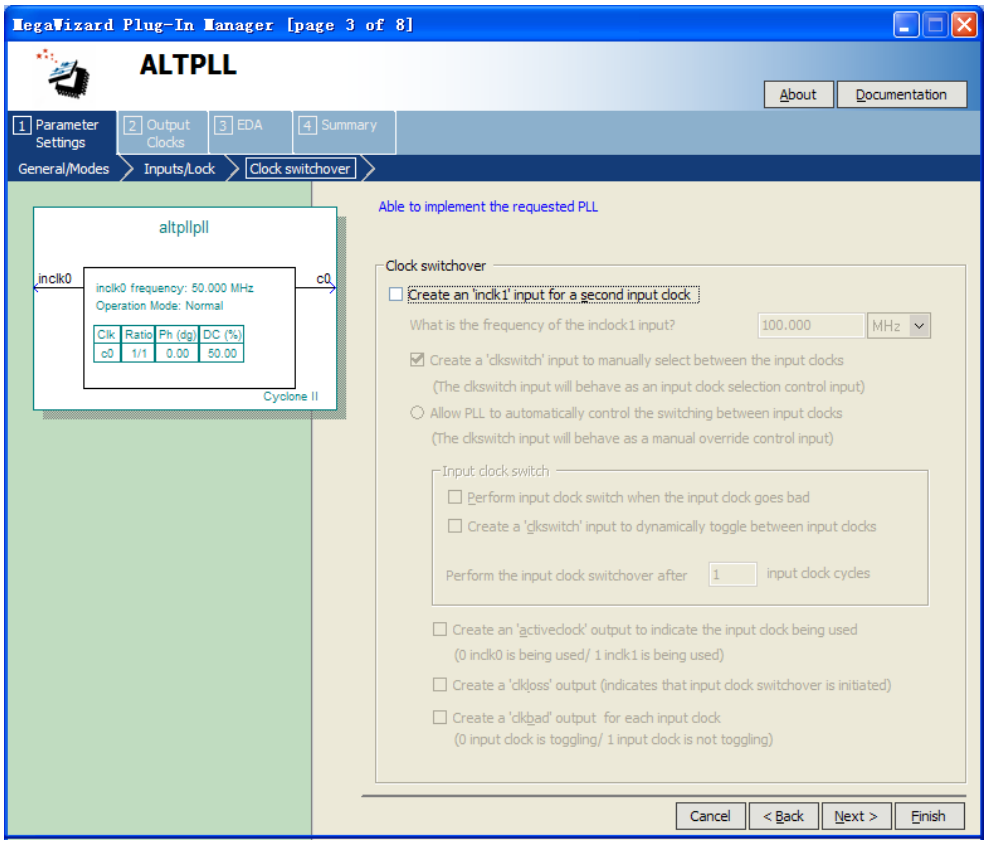


图 11-18　PLL 设置对话框之三

ALTPLL 设置 Page4

这一步涉及时钟 C0 的设置。设定该时钟为 CPU 所需要的 100MHz clk，将 Clock

multiplication factor 设为 2，注意 Actual settings 出現 100.000000Mhz。参看图 11-19。

使用 ALTPLL 产生 clk 有一点需注意，并不是任何 clk 都可以产生，若 ATLPLL 可以合成的 clk，会在上方出现 Able to implement the requested PLL。按 Next 继续。

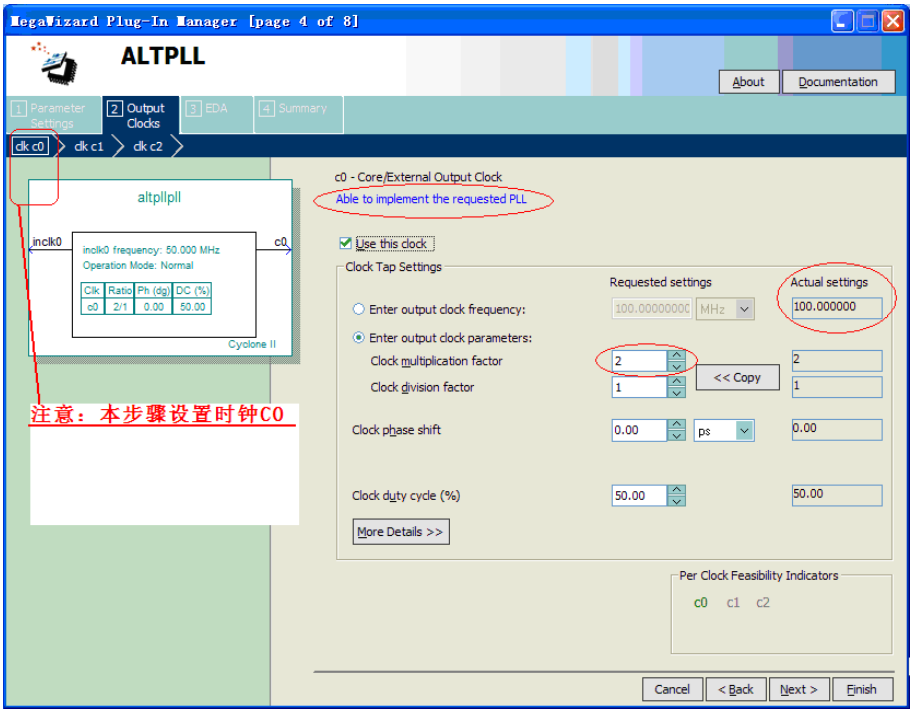


图 11-19　涉及时钟 C0 的 PLL 设置对话框之四

ALTPLL 设置 Page5

这一步操作涉及时钟 C1。在 PLL 设置对话框之五设定 SDRAM 所需要的 100MHz clk，但必须有-65 度的时钟相位偏移（Clock phase shift）。按 Next 继续。参看图 11-20。

注意：偏移角度的单位是 deg。

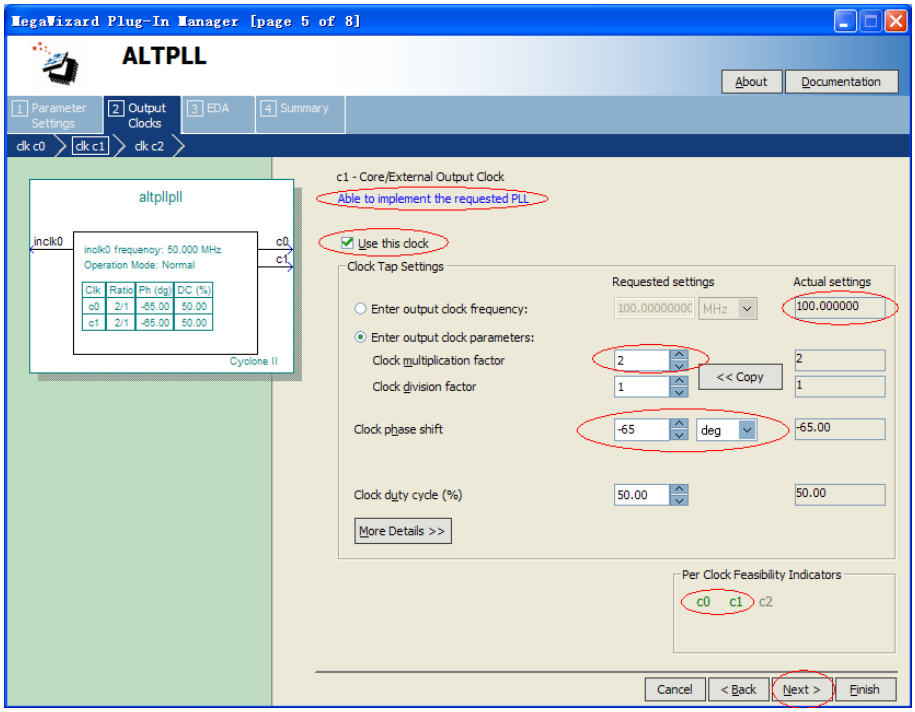


图 11-20　涉及 C1 时钟的 PLL 设置对话框之五

ALTPLL 设置 Page6

本操作涉及时钟 C2 设置。已经有了时钟 C0 和 C1，由于不再需要其他 clk，C2 不再需要，因此按“Next”继续。参看图 11-21。

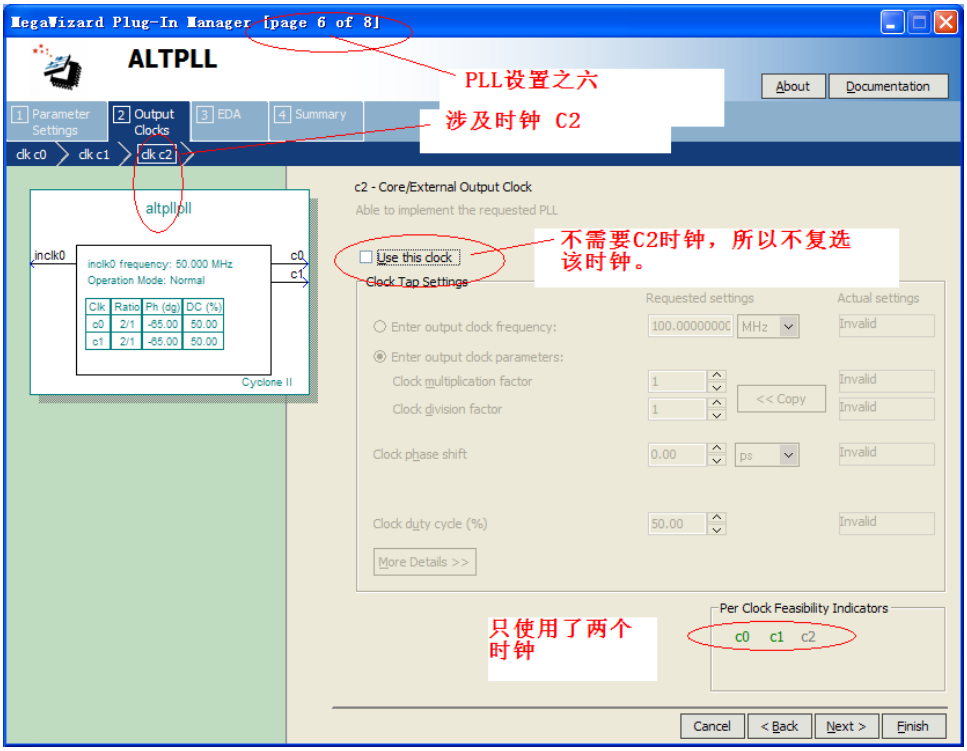


图 11-21　PLL 设置对话框之六

ALTPLL 设置 Page 7

在 PLL 设置对话框之七，实验者接受缺省值即可，按“Finish”完成。参看图 11-22。

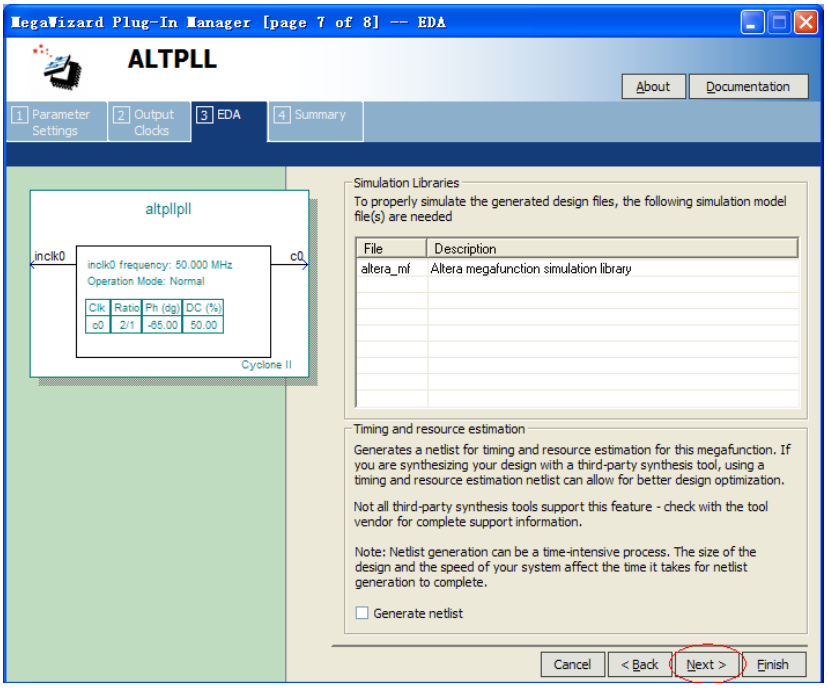


图 11-22　PLL 设置对话框之七

ALTPLL 设置 Page 8

这一步骤 SOPC Builder 给出了 PLL 设置的总结信息。实验者基本不需要做什么操作。参看图 11-23。

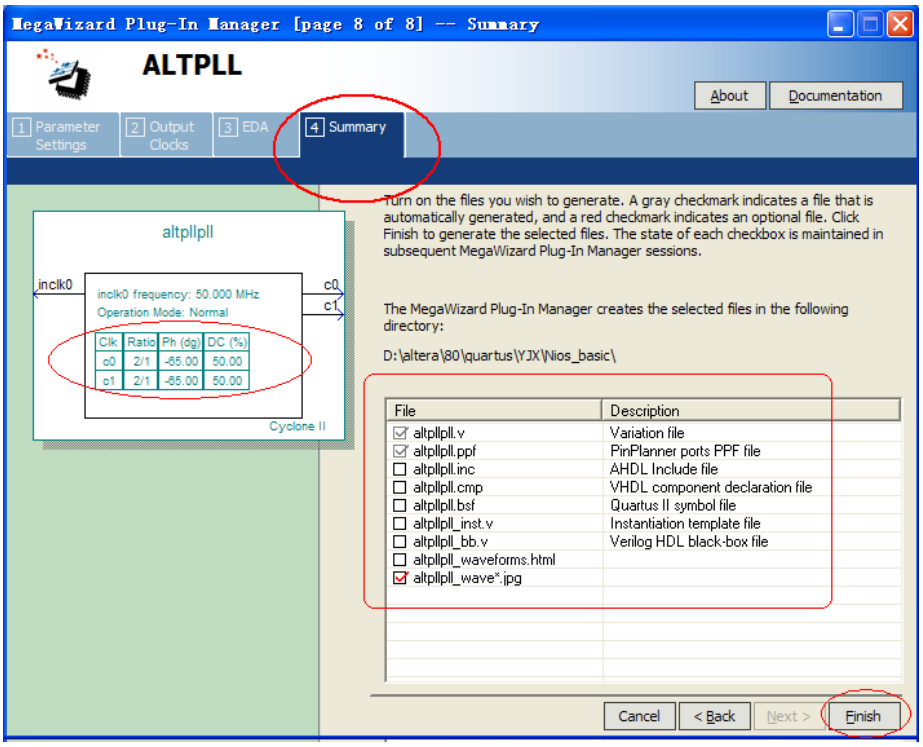


图 11-23　PLL 设置对话框之八

最后按“Finish”完成。参看图 11-24。

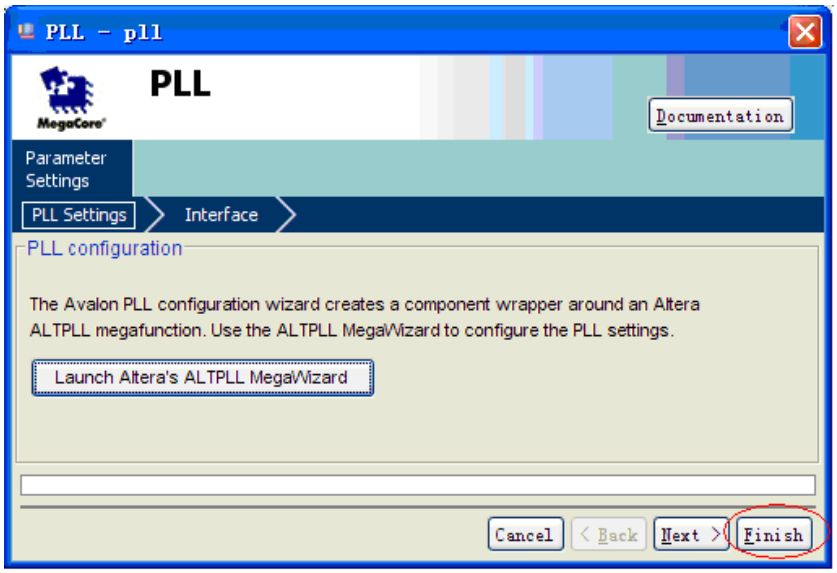


图 11-24　操作完毕时的 PLL 配置向导对话框

最后会出现 pll_0: pll_0.s1 must be connected to an Avalon-MM master 的错误信息。参看图 11-25。因为 pll_0 是个 slave ip，必须被动的受 master ip 控制，稍后等到 SOPC 加入了 master ip(此系统就是 Nios II CPU)后，就可解决。

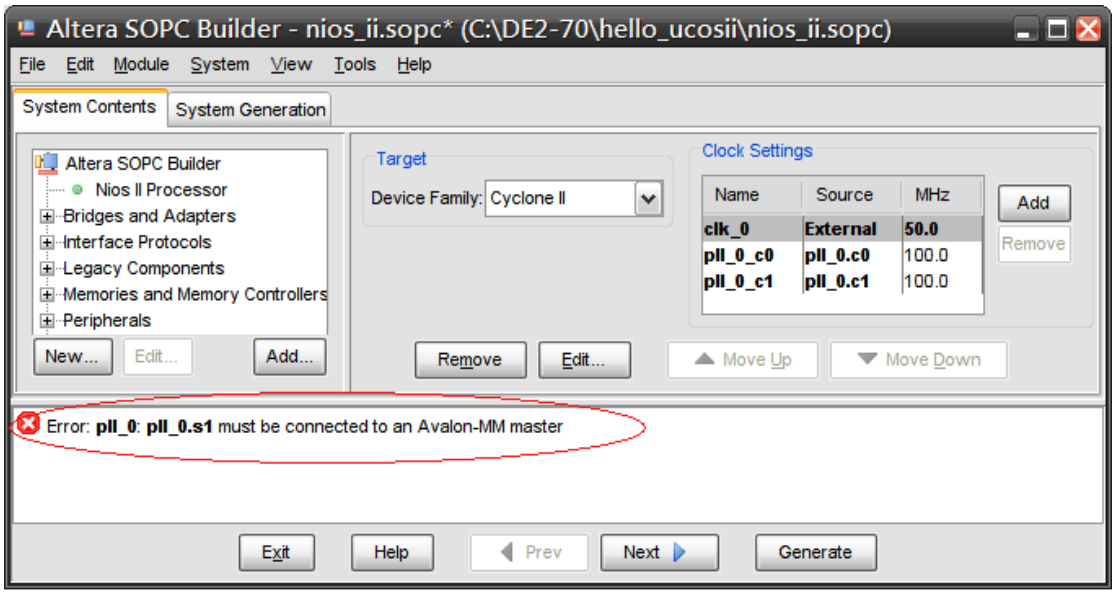


图 11-25　出现 pll_0: pll_0.s1 必须连接到 Avalon-MM master 的提示

Step 12：

更改 clk 名称。参看图 11-26。

将 clk 改成有意义的名称。

clk_0 改成 clk_50。

pll_0_c0 改成 cpu_clk。

pll_0_c1 改成 sdram_clk。

pll_0 改成 pll。

11.4　向 SOPC 添加 NIOS II CPU

Step 13 ：

加入 Nios II CPU。操作画面省略。

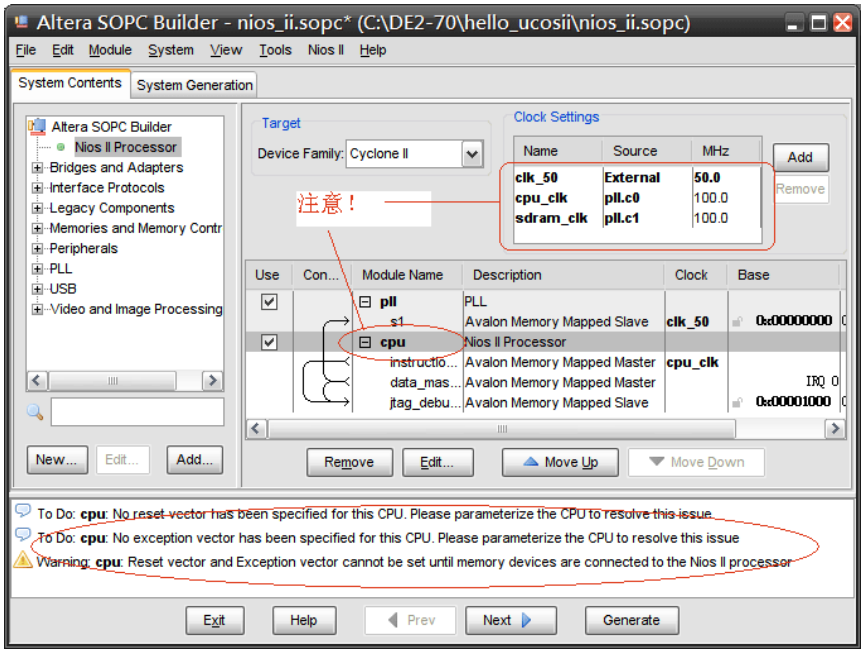


图 11-26　在 SOPC 界面添加 CPU

根据用户需要选择 Nios II Core：

1.Nios II/e (economic)：占用 LE 最少，功能最少，速度最慢。

2.Nios II/s (standard)：速度与 LE 平衡，具有 Nios II CPU 一般功能。

3.Nios II/f (full)：占用 LE 最多，功能也最多，速度最快。

本范例选用 Nios II/f。

Reset Vector 与 Exception Vector 暂时不需设定，因为还未将存储器 ip 挂上，最后再设定。其他设定接受缺省值即可，按 Finish 完成。

将 cpu_0 改成 cpu。参看图 11-26。

因为 Nios II CPU 的加入，而 Nios II CPU 就是个典型的 master ip，SOPC 自动会将 cpu(master)与 pll(slave)相连，因此 pll_0: pll_0.s1 must be connected to an Avalon-MM master 错误信息不见了。新增的 warning 是刚提到的 reset vector 与 exception vector，最后再设定。

Step 14：

往 SOPC 里追加 On-Chip Memory。参看图 11-27。

FPGA 内少量的 M4K 存储器，是 DE2-70 所有存储器中速度最快，但是是容量最小的存储器。

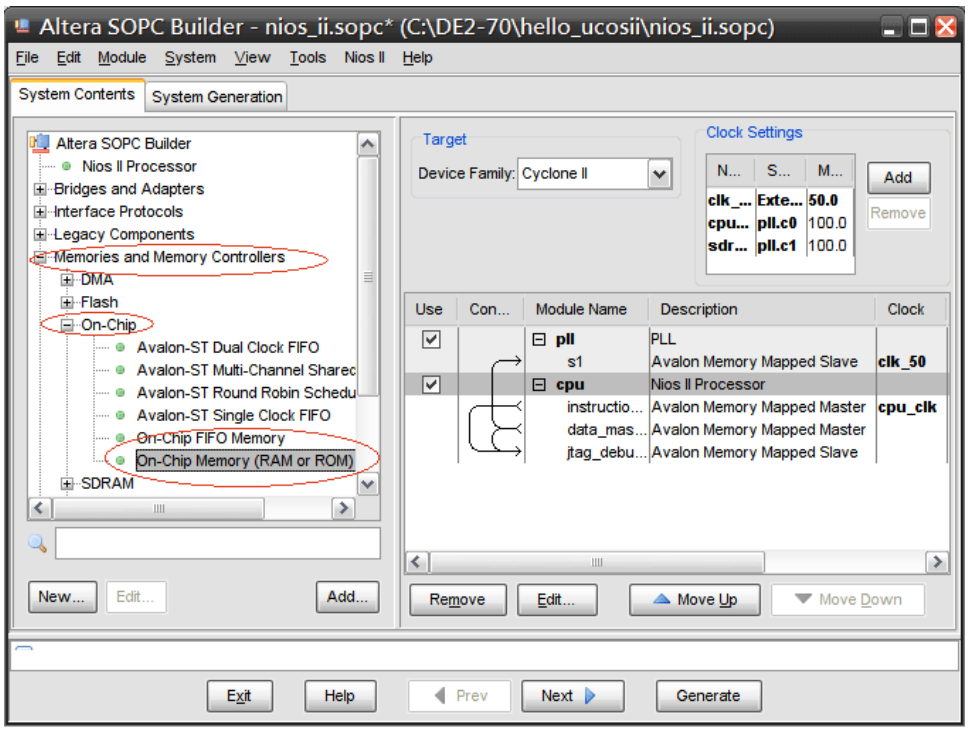


图 11-27　添加片上存储器 On-chip Memory

设定 30 KBytes 的 On-chip Memory。

Total memory size 与能使用的 M4K 存储器数量有关。不同的 FPGA 型号有关、选择不同的 Nios II CPU

Core、使用 Megafunction(如 fcfifo)都会影响 On-chip Memory 的 size。

将 onchip_memory2_0 改成 onchip_mem。

出现 2 个 cannot be at 0x1000 的错误信息是正常的，因为 SOPC Builder 初步为 onchip_mem 的定址 0x1000 并不合法，等所有 ip 都加入后，最后会一起重新对所有 slave ip 定址。

若 Nios II 的程序很小，可将软件完全跑在 On-chip Memory，而不需 SSRAM、SDRAM 或 Flash 等其他存储器。由于 On-chip Memory 速度最快，也可以将较常用的变量、阵列放

在 On-Memory 加快读取速度，或将 Exception vector 指定于 On-chip Memory，加快 interrupt 处理。

11.5 向 SOPC 添加三态桥和 SSRAM

Step 15：

在 SOPC 操作界面，加入 SSRAM 的 Tristate Bridge。因为 SSRAM 与 Flash 的 databus 是三态（tristate），所以 Nios II CPU 与 SSRAM、Flash 相连时需要透过 Tristate Bridge。操作界面图省略。

接受缺省值即可，按 Finish 完成。将 tri_state_bridge_0 改成 tristate_bridge_ssram。

出现了 tristate_bridge_ssram: tristate_bridge_ssram.tristate_master must connected to Avalon-MM Tristate slave 的错误信息，因为 tristate_bridge_ssram 的 master interface 需要与 slave ip 相连，下一步加上 SSRAM controller 后，就可以解决此问题。

Step 16：

加入 SSRAM。接受缺省值即可，按 Finish 完成。

将 ssram_0 改成 ssram。参看图 11-28。

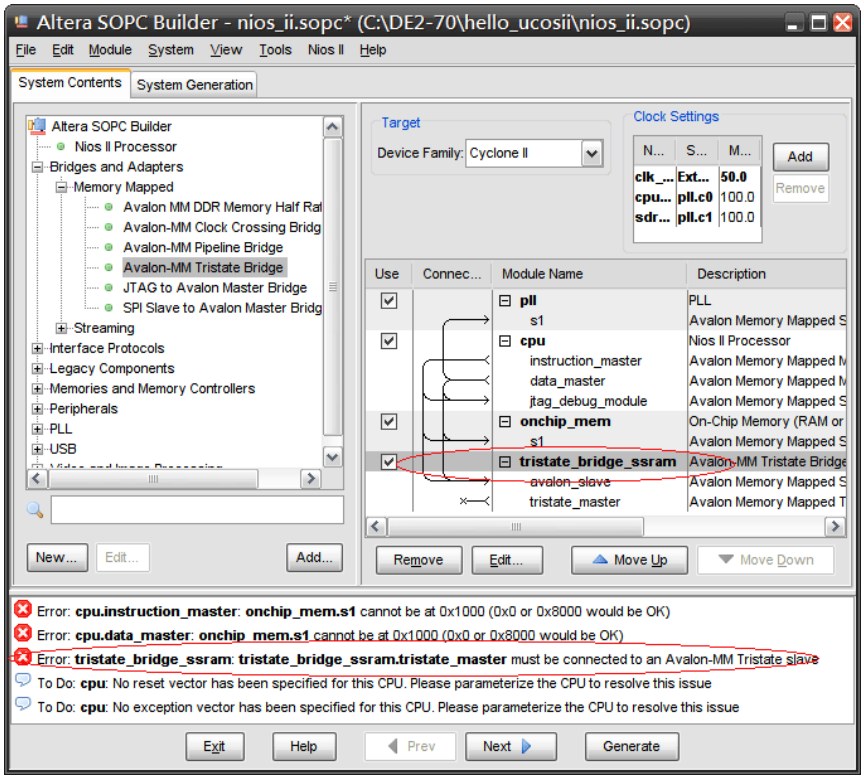


图 11-28 向 SOPC 添加三态桥

将 ssram 与 tristate_bridge_ssram 相连，这样就可解掉 tristate_bridge_ssram: tristate_bridge_ssram.tristate_master must connected to Avalon-MM Tristate slave 错误信息。

SSRAM 是 DE2-70 上速度仅次于 On-chip Memory 的存储器，且容量从 DE2 的 512KB 进步到 2MB，实用性大增。已经足夠大部分的软件跑在 SSRAM 了。

11.6 向 SOPC 添加 Flash

Step 17：

在 SOPC 操作界面，加入 Flash 的 Tristate Bridge。如同 Step 15 加入一个 Tristate Bridge，并将名称改成 tristate_bridge_flash。操作界面图省略。

Step 18：

在 SOPC 操作界面，加入 Flash。参看图 11-29。

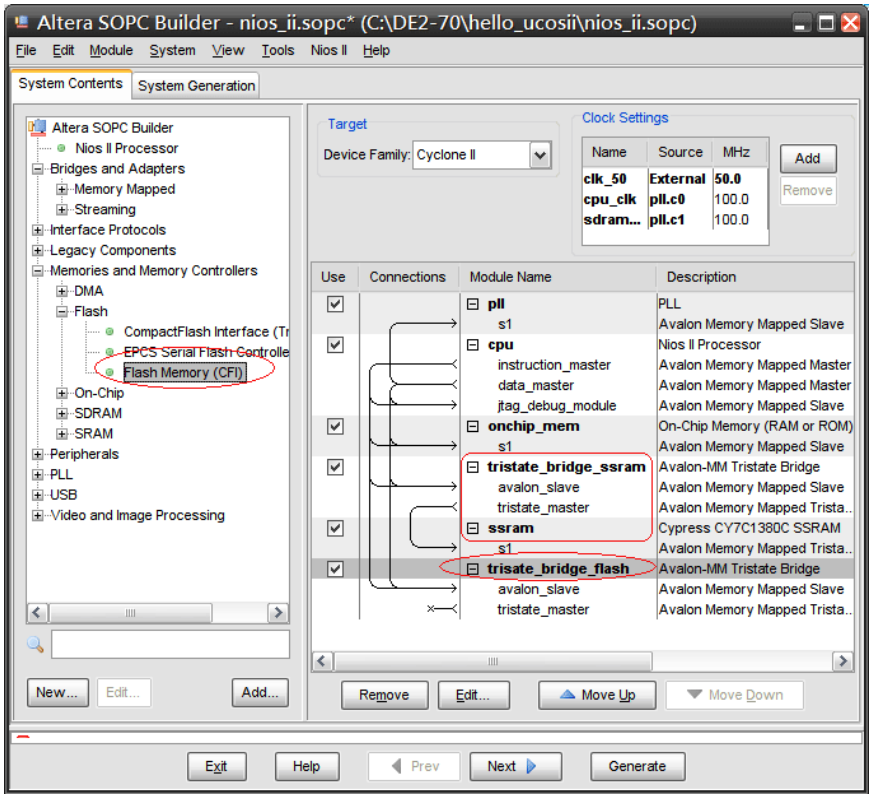


图 11-29　向 SOPC 添加闪存

在 Attribute 这一页，将 Address Width 设为 22，Data Width 设为 16。在 Timing 这一页，将 Wait 设为 100。按 Finish 完成。参看图 11-30。

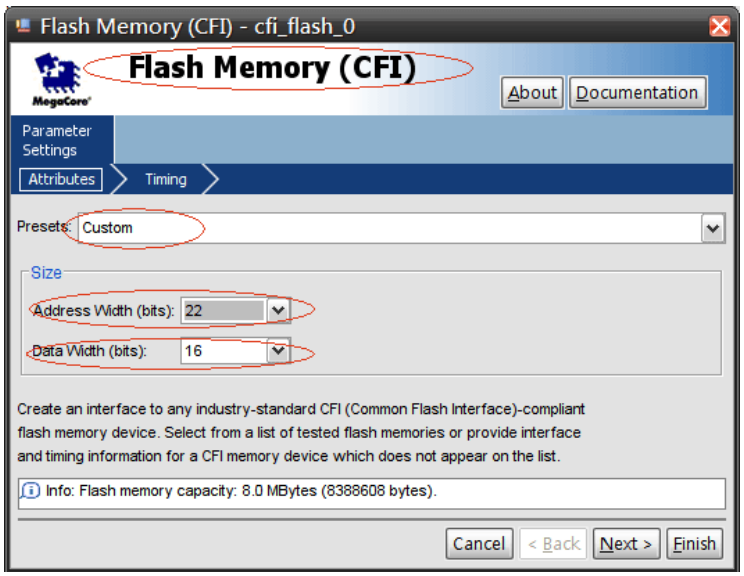


图 11-30　配置闪存的 CFI 参数

将 cfi_flash_0 改成 cfi_flash。将 cfi_flash 与 tristate_bridge_flash 相连，这样就可解决掉 tristate_bridge_flash: tristate_bridge_flash.tristate_master must connected to Avalon-MM Tristate slave 错误信息。

多了两个错误信息，列出如下：

1，onchip_mem.s1 (0x1000..0x8fff) overlaps cfi_flash.s1(0x0..0x7fffff)

与

2，onchip_mem.s1 (0x1000..0x8fff) overlaps cfi_flash.s1(0x0..0x7fffff)

因为 SOPC Builder 为 cfi_flash 初步的定址 0x00000000..0x007fffff 已经与 onchip_mem 的 0x00001000..0x000087ff 相重叠，等所有 controler 都加入后，最后会一起重新对所有 controller 定址。

Flash 为 DE2-70 上唯一断电后仍保存资料的存储器，若想 Nios II 程序断电后，一通电就可以执行，就要将软件放在 Flash 上。

11.7 向 SOPC 添加 SDRAM

Step 19：

加入 SDRAM。参看图 11-31。

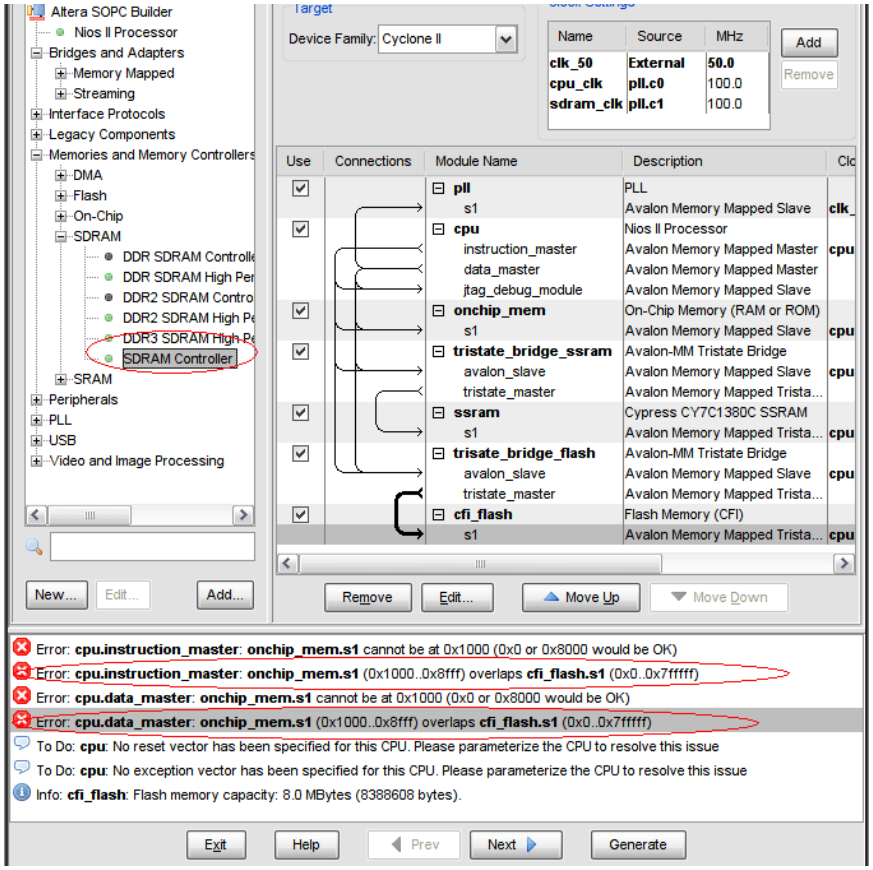


图 11-31　添加 SDRAM 控制器

在 Memory Profile 这一页，设定

Presets : Custom

Data width : 32

Chip select : 1

Banks : 4

Row : 13

注意最后 Memory size 为 64MBytes。

在 Timing 这一页，设定：

Issue one refresh command every: 7.8125 us

Delay after powerup, before initialization : 200 us

将 sdram_0 改成 sdram。

一样会出现 sdram base address 的错误信息，最后会一起解决。

Step 20：

向 SOPC 系统添加 JTAG UART。参看图 11-32。

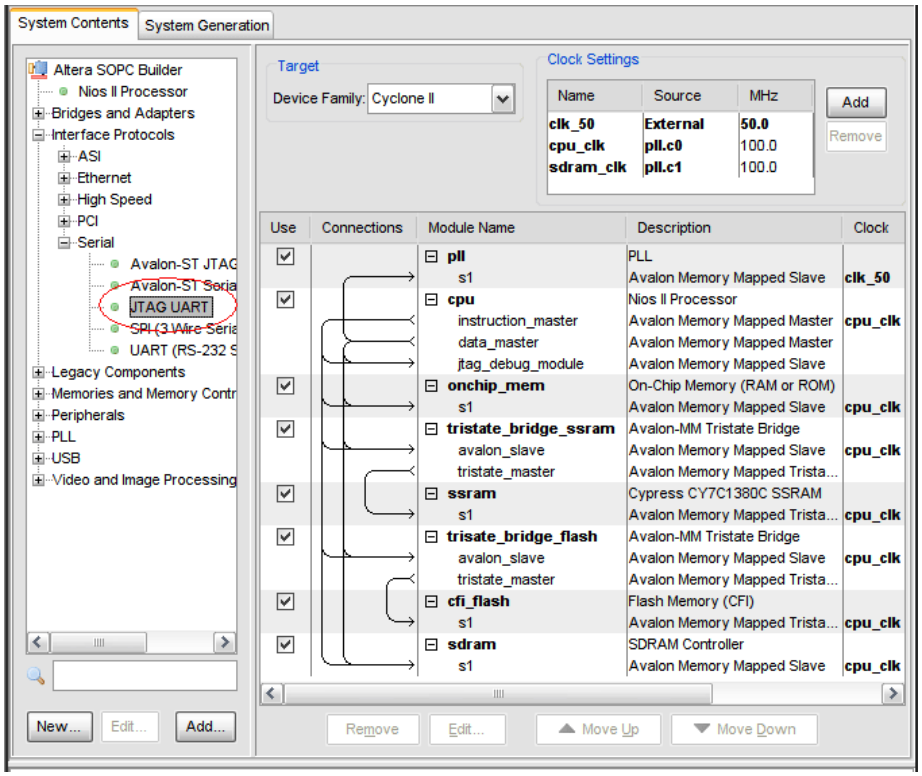


图 11-32　添加 JTAG UART

JTAG UART 是 PC 与 SOPC 进行序列传输的一种方式，也是 Nios II 标准的输出/输入设备。如 printf( )透过 JTAG UART，经过 USB Blaster 将输出結果显示在 PC 的 Nios II EDS 上的 console，scanf( )透过 USB Blaster 经过 JTAG UART 将输入传给 SOPC 的 Nios II。

接受缺省值即可，按 Finish 完成。将 jtag_uart_0 改成 jtag_uart。

11.8 向 SOPC 添加 UART 和 Timer

Step 21：

加入 UART(RS-232 Serial Port)。参看图 11-33。

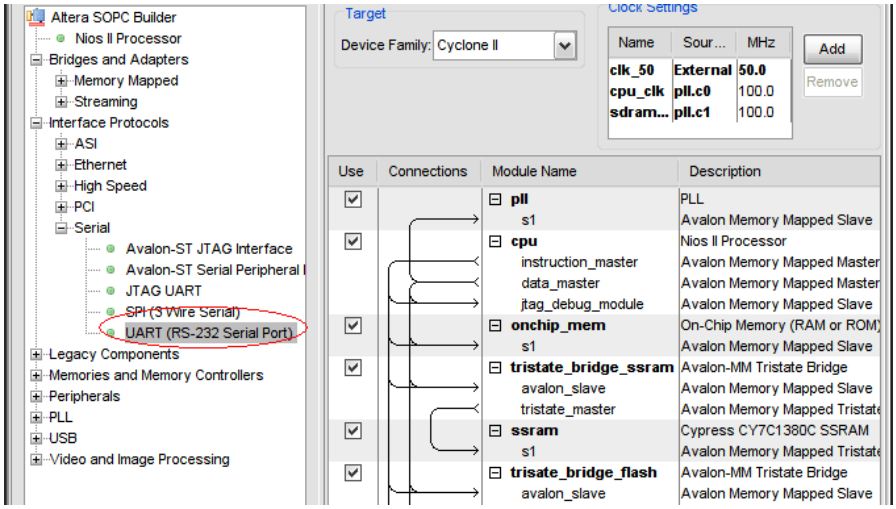


图 11-33　向 SOPC 添加 UART

UART 是 PC 与 SOPC 进行序列传输的另一种方式，也是 Nios II 标准的输出/输入设备。如 printf( )也可透过 UART，经过 RS232 将输出结果显示在 PC 的 Nios II EDS 上的 console，scanf( )也可透过 RS232 经过 UART 将输入传给 SOPC 的 Nios II。

将 Include CTS/RTS pins and control register bits 打勾，其他接受缺省值即可，按 Finish 完成。参看图 11-34。

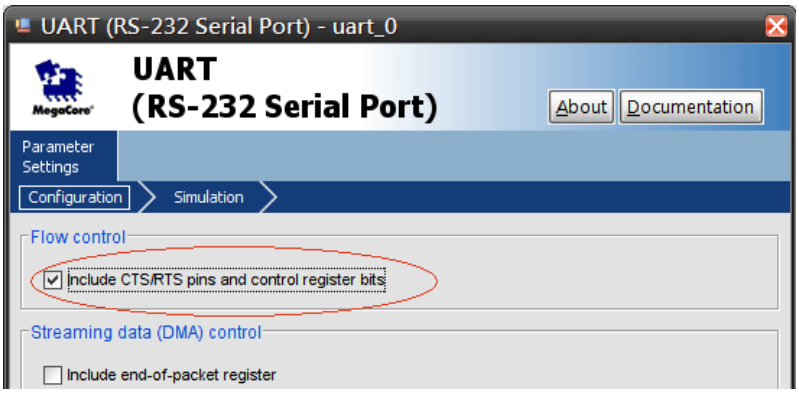


图 11-34　对 UART 设置流量控制

将 uart_0 改成 uart。

Step 22：

加入 Timer。参看图 11-35。

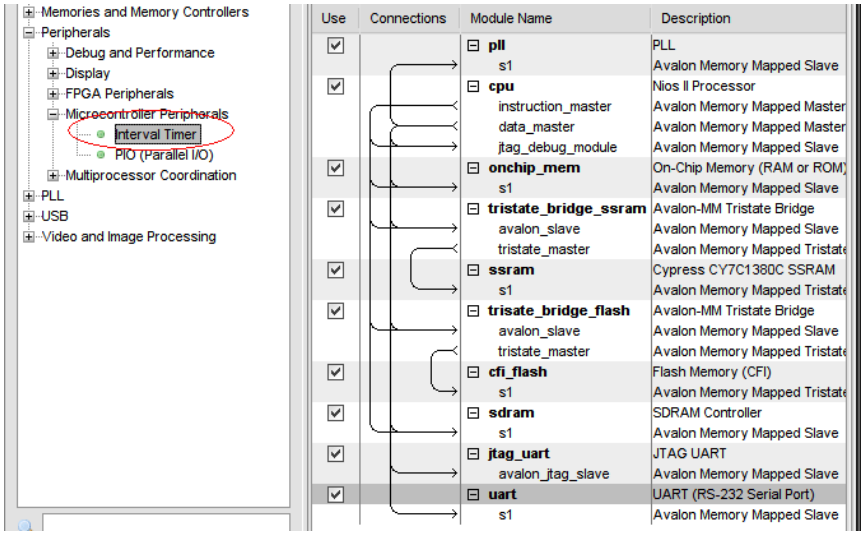


图 11-35　添加间隔定时器

此处要加入 2 个 timer，一个做为 system clock，一个做为 timestamp。尤其是 μC/OS-II 操作系统一定需要 timestamp timer，否则无法执行。

接受缺省值即可，按 Finish 完成。将 timer_0 改成 sys_clk_timer

重复相同的步骤，加入 timestamp_timer。

11.9 向 SOPC 添加 System ID

Step 23：

加入 System ID。参看图 11-36。

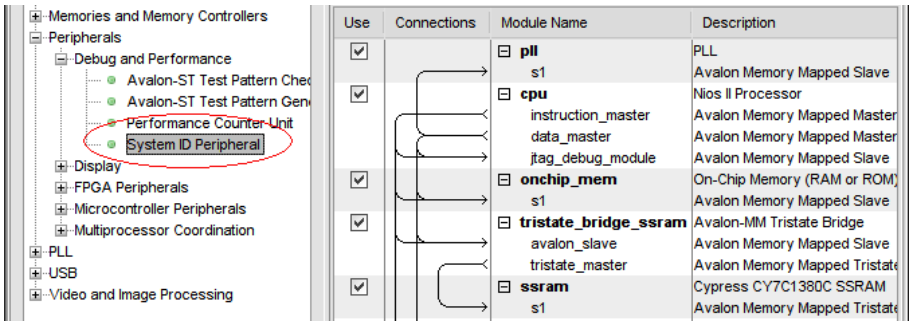


图 11-36　添加 System ID Peripheral 操作画面

SOPC Builder 会使用 System ID 为每个系统提供识别符号，Nios II EDS 可以此防止使用者烧录了与*.ptf 不符合的*.sof。

接受缺省值即可，按 Finish 完成。

将 sysid_0 改成 sysid。

注意：

一定要将名称改成 sysid，因为 Nios II EDS 将会使用 sysid 做判断，否则会产生错误。

11.10 向 SOPC 添加字符 LCD

Step 24：

加入 Character LCD。参看图 11-37。

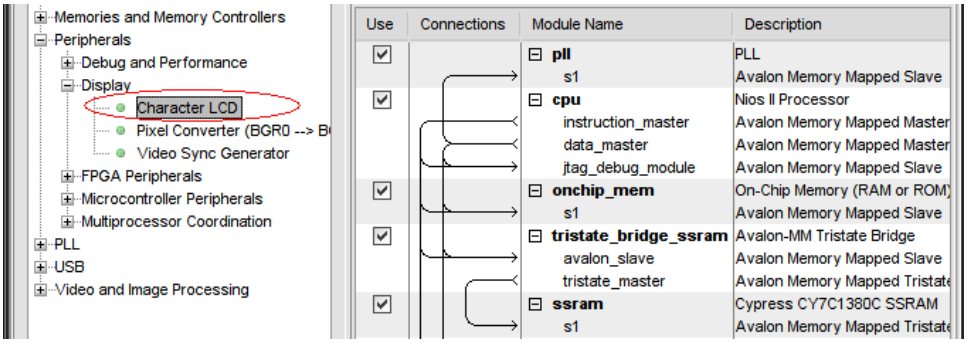


图 11-37　添加 Character LCD 的操作画面

为 DE2-70 上的 16 x 2 字元型 LCD，可做为 Nios II 的标准输出设备。如 printf( )也可透过 Character LCD 将字串显示在 LCD 上。接受缺省值即可，按 Finish 完成。

将 lcd_0 改成 lcd。

Step 25：

加入 LED PIO。参看图 11-38。

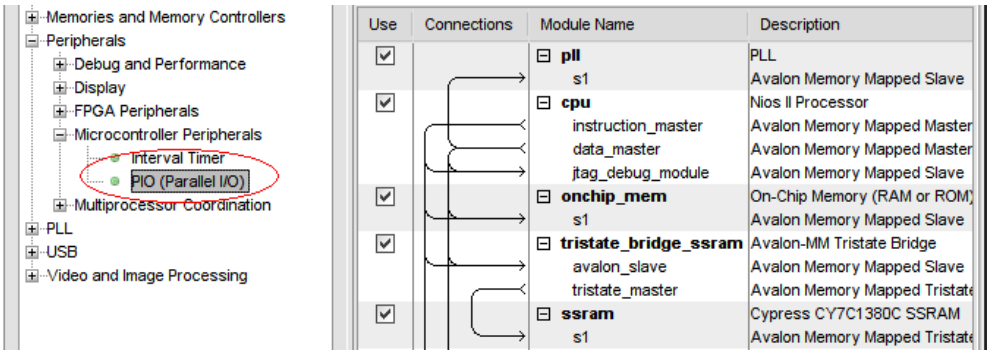


图 11-38　添加 PIO 的操作画面

实际上在本例的 NIOS SOPC 里面一共要添加 4 个 PIO 外设。它们分别是红色 LEDR 发光管 18 个，绿色 LEDG 发光管 9 个，KEY 弹性按钮 4 个和 SW 开关 18 个。它们都属于并行 IO 接口上的外设。参看图 11-39。

此处要先加入 2 个发光管 LED PIO，一个为 LEDG，一个做为 LEDR。

参看图 11-39。

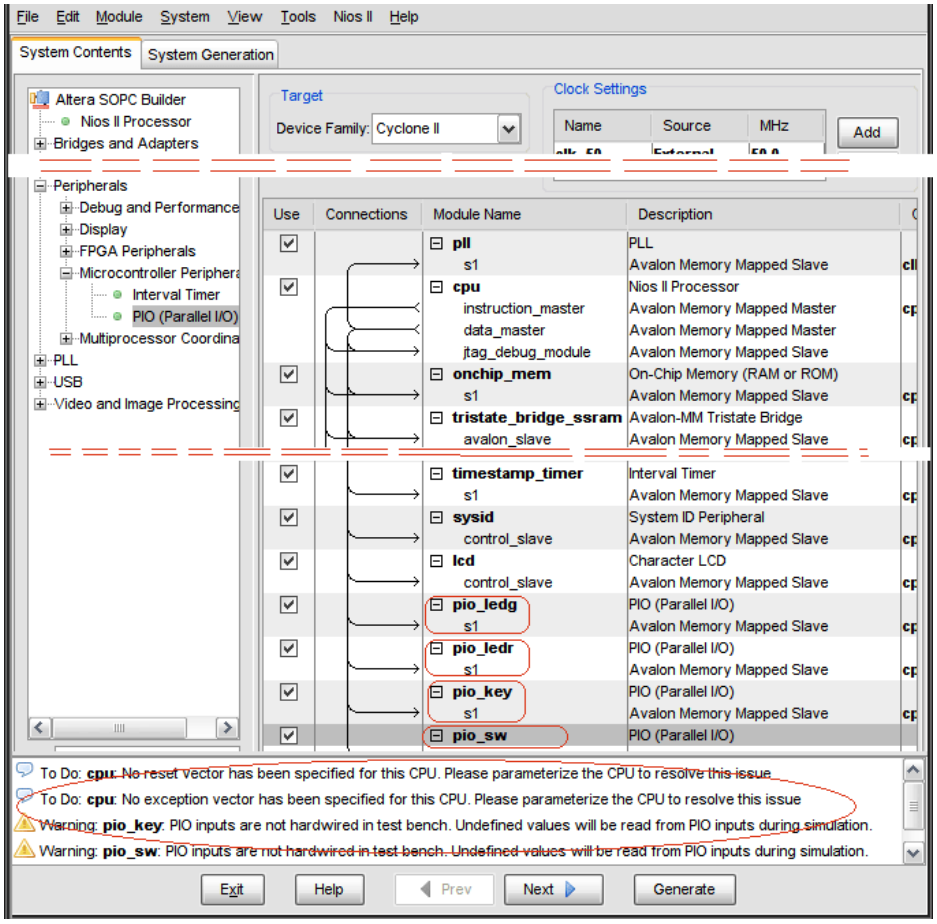


图 11-39　先后添加四个并行输入输出（PIO）外设

LEDG

因为 LEDG[8:0]且仅用于输出，所以 width 设为 9，且选择 Output ports only，按 Finish 完成。将 pio_0 改成 pio_ledg。参看图 11-40。

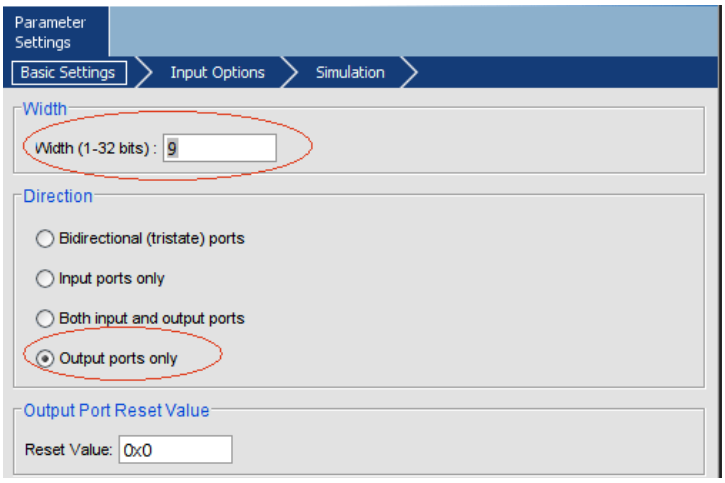


图 11-40　LEDG 的宽度位设置

LEDR

如同增加 LEDG PIO 的方式一样，加入 LEDR PIO。因为 LEDR[17:0]且仅用于输出，所以 width 设为 18，且选择 Output ports only，按 Finish 完成。将名称 pio_0 改成 pio_ledr。参看图 11-41。

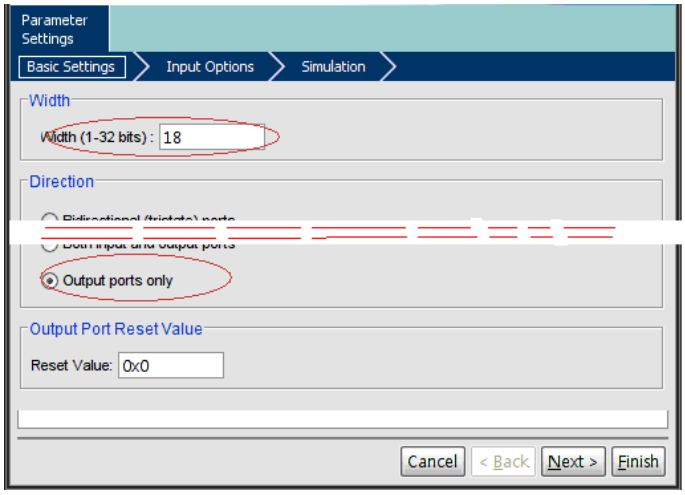


图 11-41　通用 IO 口的参数设置

Step 26：

向 SOPC 加入 Input PIO。参看图 11-42。

图 11-42　KEY 在 DE2-70 的位置

此处要加入 2 个 Input PIO，一个为 KEY，一个为 SW。

KEY

如同 Step 25 一样，选择 PIO(Parallel I/O)。

因为 KEY[3:0]且仅用于输入，所以 width 设为 4，且选择 Input ports only，按 Finish 完成。参看图 11-43。

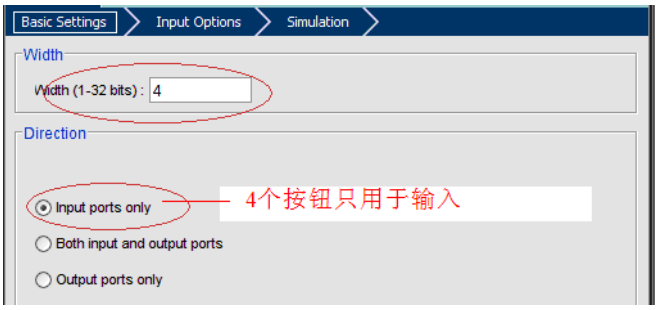


图 11-43　按钮的位宽和输入输出选择的设置

将 pio_0 改成 pio_key。

SW

因为 SW[17:0]且仅用于输入，所以 width 设为 18，且选择 Input ports only，按 Finish 完成。将 pio_0 改成 pio_sw。参看图 11-44。

图 11-44　开关 SW 的位宽设置

Step 27：

重新设定 Base Address。参看图 11-45。

之前很多 error 与 warning 都是因为都是因为 controller 的 base address 重复，目前已经将所有用到的 controller 加到 SOPC

Builder，可以让 SOPC Builder 重新设定 base address。

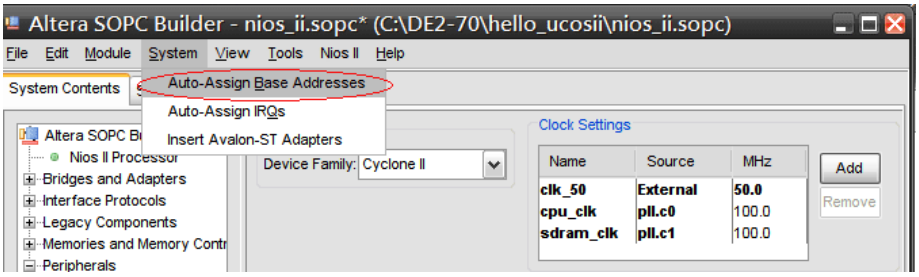


图 11-45　重新自动设置内存基地址

错误信息都不见了。参看图 11-46。

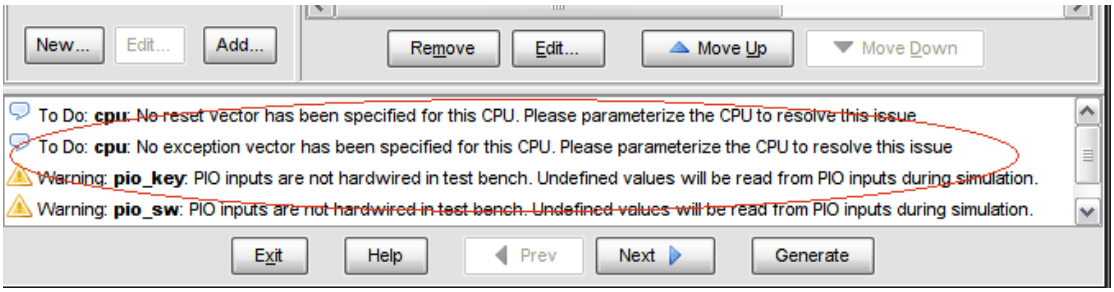


图 11-46　自动设置内存地址之后错误信息消失

Step 28：

设定 Nios II CPU 的 Reset Vector 与 Exception Vector

在 Step 13 设定 Nios II CPU 时，因为当时还没有挂上任何存储器，所以当时还没设定 Reset Vector 与 Exception Vector，现在再来设定。

简单地说，reset vector 就是当系统 reset 时，CPU 会跳到 reset vector 所指定的位址执行，所以 reset vector 所指定的存储器必须是非挥发性存储器，在 DE2-70 只有 flash。

而 exception vector 则是当发生 hardware interrupt 或 software exception 时，CPU 会跳到 exception vector 所指定的位址执行，为了更有效率，我们会将 exception vector 指向速度最快的存储器，通常是 on-chip memory 或 SSRAM。

最后按 Finish 完成。

最后完成所有 SOPC 设定，注意下方只有 3 条信息，前 2 个 warning 是提醒 pio_key 与 pio_sw 需要 input testbench，这可以忽略。

11.11 生成 NIOS SOPC

Step 29：

本步骤产生 SOPC

按下 SOPC 对话框的 Generate 按钮，正式产生 nios_ii.ptf，这需要一点时间产生，依 CPU 运算速度而定。 按 Save 将 nios_ii.sopc 存文件，这个文件记载著所有 SOPC Builder 的 controller 与设定值。

参看图 11-47。

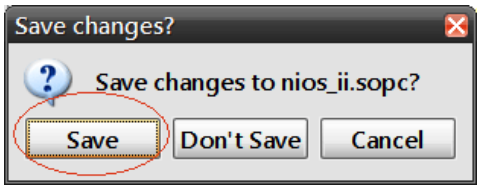


图 11-47　SOPC 设定值的存盘界面

约 1 分半后，SOPC Builder generate 成功。参看图 11-48。

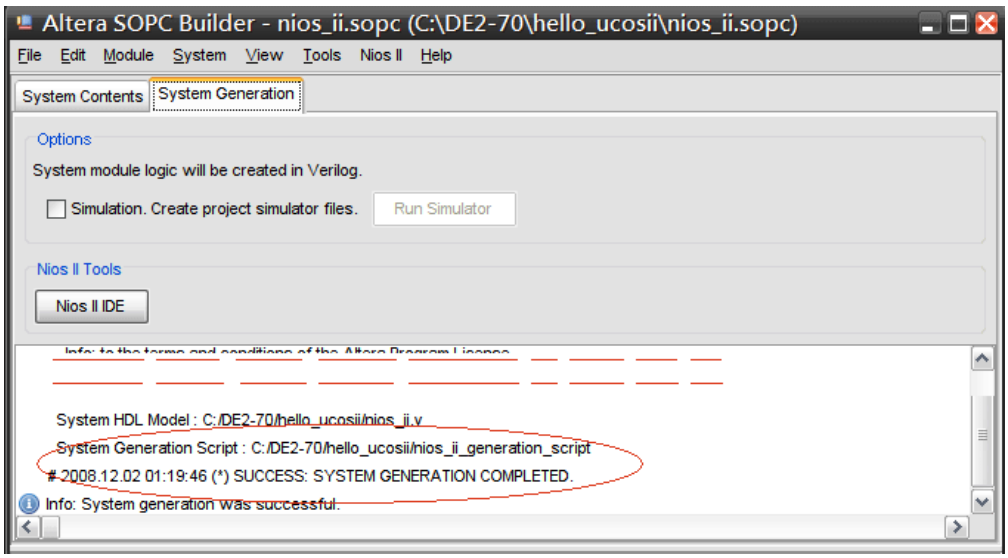


图 11-48　SOPC Builder generate 的显示画面

SOPC Builder Generate 主要做 2 件事情。

1.产生 nios_ii.ptf 文件：

这个文件将来是 Nios II EDS 用来产生 System Library 的依据，Nios II 软件与 μC/OS-II 才能借此存取 SOPC。

2.产生各种控制器（controller）的 Verilog 文件：

由于 SOPC 基于 FPGA 技术，最后必须由 Quartus II 产生*.sof 文件烧到 FPGA 内，SOPC Builder 会产生 controller 的 Verilog code 供 Quartus II 编译。你可以发现 project 的根目录多了很多*.v，这就是 SOPC Builder 所产生的。

顶层模块 Top Module

SOPC Builder 为我们产生了各种 controller 的 module 后，需要有一个顶层模块（top module）使之与 DE2-70 的 I/O port 相连。

11.12 建立顶层模块（Top Module）

Step 30：

建立顶层模块（Top Module）

在 C:\DE2-70\hello_ucosii\建立 hardware 目录。

下载：　http://www.blogjava.net/Files/oomusou/Lab1_files.zip

其中的 hello_ucosii.v 是一个未完成的 top module，将 hello_ucosii.v 与 Reset_Delay.v 放在 C:\DE2-70\hello_ucosii\目录下。

有 4 个地方必须修改后，才能完成顶层模块（top module）。

256 行

```
.sdram_clk( ),                // SDRAM Clock // To do!! SDRAM Clock output
```

改成

```
.sdram_clk(sdram_clk), // SDRAM Clock
```

将 pll 产生的 SDRAM clk 连出來。

217 行

```
// To do!! assign sdram clk to oDRAM0_CLK
```

改成

```
assign oDRAM0_CLK     = sdram_clk;     // SDRAM0 Clock
```

229 行

```
// To do!! assign sdram clk to oDRAM1_CLK
```

改成

```
assign oDRAM1_CLK     = sdram_clk;     // SDRAM1 Clock
```

因为 DE2-70 有 2 片 SDRAM，所以要 assign 两次 clk。

277 行

```
.zs_dq_to_and_from_the_sdram( ),      // SDRAM Data bus 32 Bits
                                      // To do!! SDRAM Data bus output
```

改成

```
.zs_dq_to_and_from_the_sdram(DRAM_DQ),   // SDRAM Data bus 32 Bits
```

因为 DRAM_DQ 是 inout 类型，不能使用 wire 连接，要直接连到 inout port。

如何知道 SOPC System 有哪些 port 呢?

打开 nios_ii.v，这是由 SOPC Builder 所产生的 Verilog code，搜索“module nios_ii (”，可以找到 SOPC System 的 module 定义，top module 就是根据这里的定义，连到 DE2-70 的 I/O port。

11.13 设定所有未使用引脚为三态输入引脚

Step 31：

将所有未使用引脚设定为三态输入引脚。也就是执行 Reserved all unused pins As input tri-stated。

从主菜单的 Assignment -> Device 进入。如图 11-49 所示。

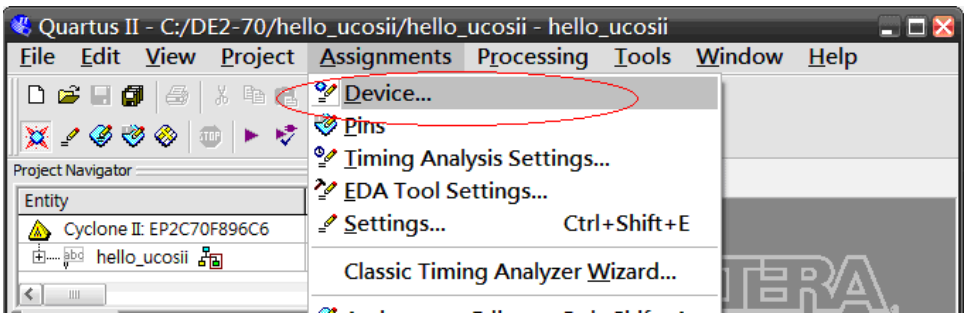


图 11-49　Device…选择项目的位置

选择 Device 对话框里面的 Device and Pin Options...。再打开 Unused Pins 选项卡。参看图 11-50。设置 Unused Pins : Reserve all unused pins: As input tri-stated。

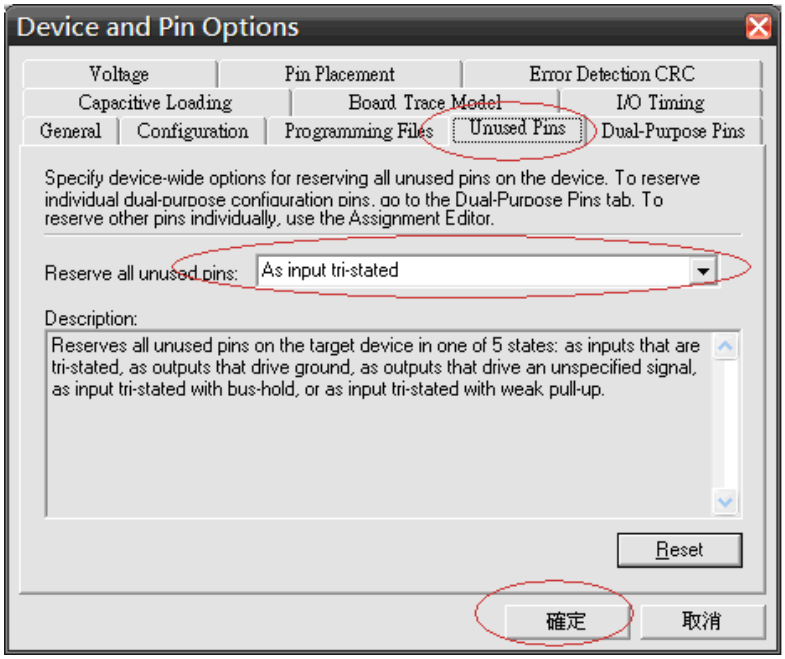


图 11-50 将所有未使用的引脚设定为三态输入引脚

这个步骤一定要做，否则 Nios II 无法执行，会出现以下如图 11-51 的错误信息，初学者常常忽略这个步骤!!

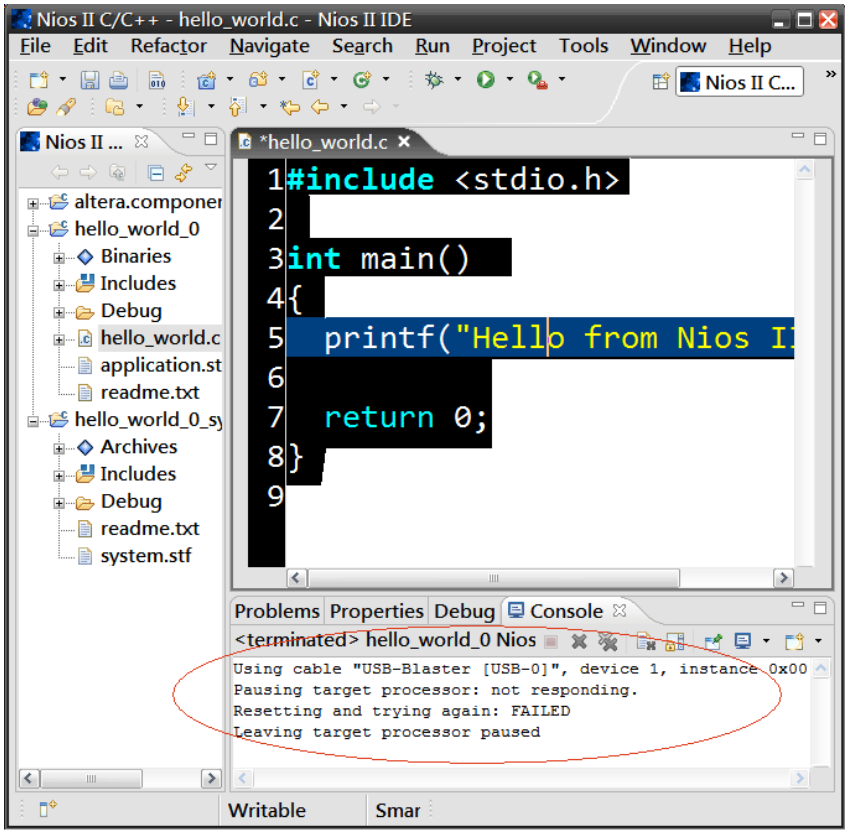


图 11-51　未使用引脚没有被设定为三态输入引脚的错误之一

11.14 引脚分配

Step 32：

Import Pin Assignment

下载 http://www.blogjava.net/Files/oomusou/Lab1_files.zip 。

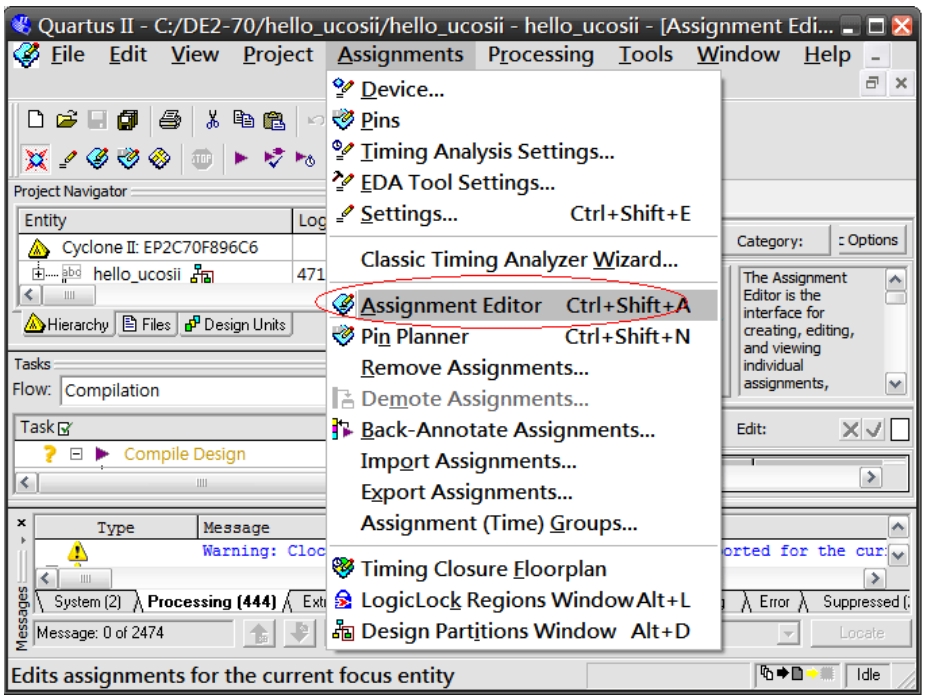


图 11-52　引脚分配编辑器选项

其中的 DE2_70_pin_assignments.csv 是逗号分割字段的 txt 文件，记录了 DE2-70 所有 I/O 的连接引脚。执行 Assignments -> Import Assignments... 菜单选项。载入 DE2_70_pin_assignments.csv，按 OK 继续。查看 Pin Assignment 结果。这个查看需要执行 Assignments -> Assignment Editor 菜单项。如图 11-52 所示。

假如引脚分配结果如图 11-53 所示，就表示 Pin Assignment 设定成功。

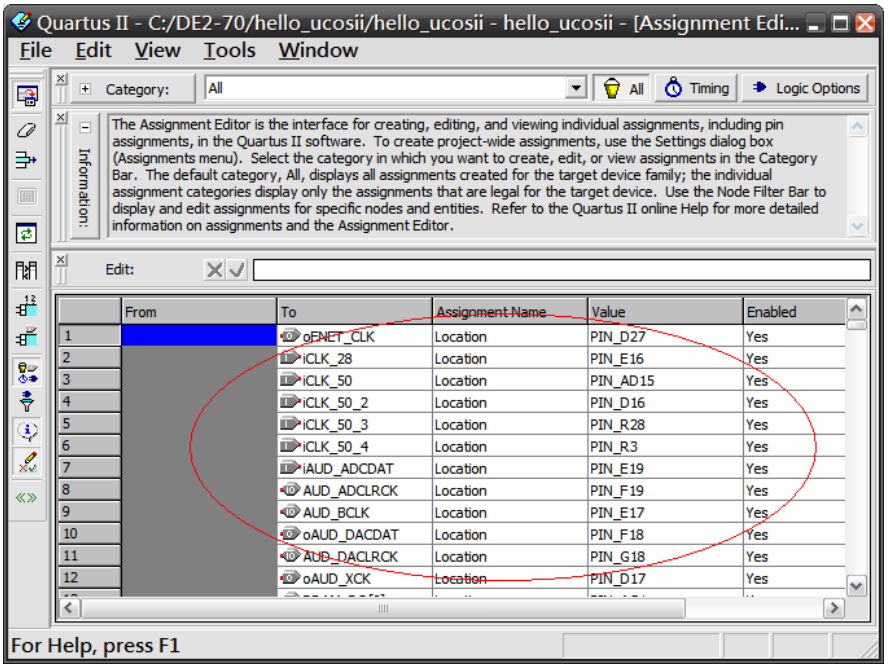


图 11-53　将 DE2-70 的.CSV 格式的引脚定义文件读入之后的画面

注意：这个步骤一定要做，而且要在 SOPC 正确生成之后做，否则 Nios II 无法执行。而且也会出现以下如图 11-54 给出的错误信息，初学者常常忽略这个步骤!!

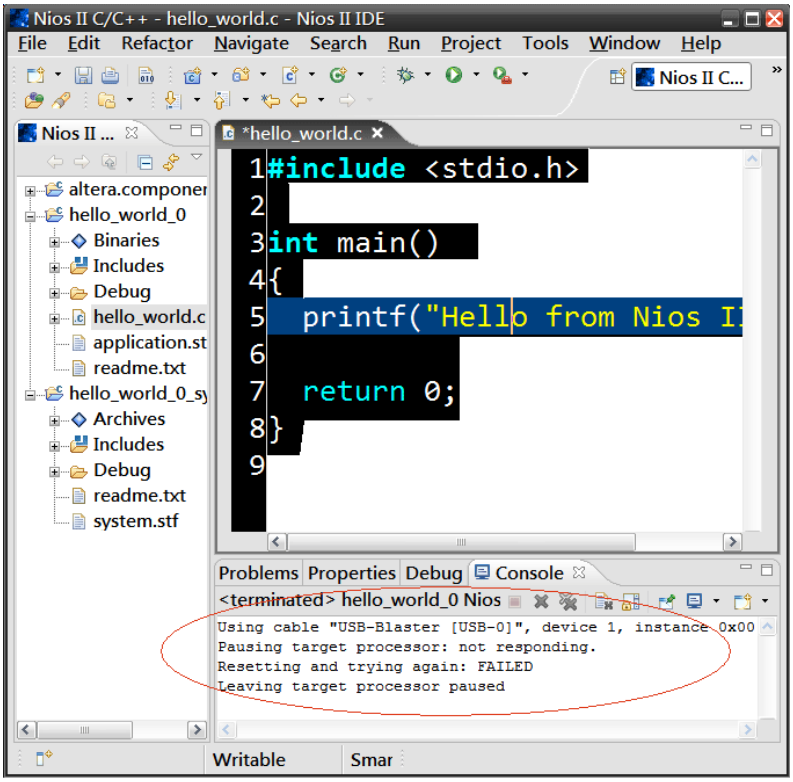


图 11-54　引脚没有定义可能产生的故障现象

11.15 设定 nCEO 的属性为 Use as regular I/O

Step 33：

设定双用途信号（Dual-Purpose Pin）nCEO 为 Use as regular I/O。

注意：

这是一个重要的操作步骤。在其他操作中也要保证将 nCEO 信号设定为 Use as regular I/O 属性。参看第 12.2 节。

操作步骤如下：

⑴从主菜单进入，点击 Assignments -> Device…项目。这时会弹出一个器件选择对话框，如第 12 章的图 12-3 所示。

⑵单击其中的 Device and Pin Options…按钮。

⑶在弹出的 Setting 对话框里，打开 Dual-Purpose Pins 选项卡。如第 12 章的图 12-4 所示。

⑷在 Dual-Purpose Pins 选项卡里有两个信号：ASDO,nCSO 和 nCEO。参看第 12 章的图 12-5。

⑸将 nCEO 信号的属性值改为 Use as regular I/O。参看第 12 章的图 12-5。

这是 DE2-70 设计上的问题，在 DE2 或其他的 Altera 的开发版并不需要如此设定。若不这样设定，Quartus II 在编译时会出现以下错误信息而编译失败。图 11-55 所示。

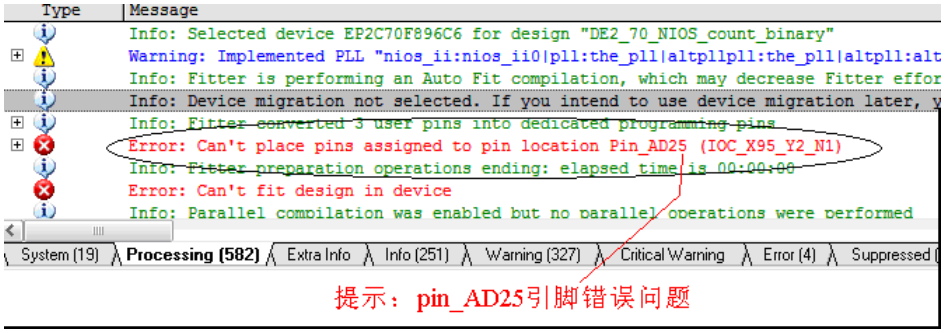


图 11-55　nCEO 没有设定为 Use as regular I/O 时可能产生的错误

Step 34：

Quartus II 全编译。大约需要 2 到 10 分钟的时间，视 CPU 速度而定。可以用菜单项或者按钮启动全编译操作。图 11-56 给出了全编译操作按钮位置。

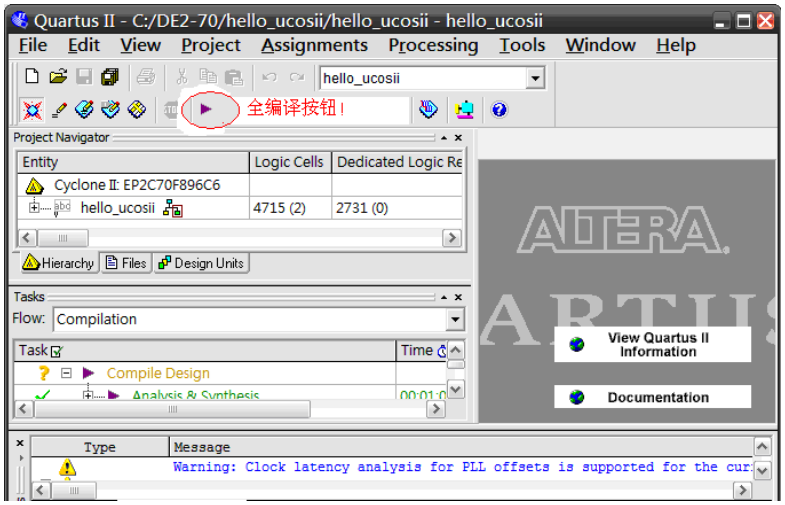


图 11-56　编译 hello_ucosii 的界面

当全编译正常完成时，Quartus II 给出如图 11-57 的对话框，指明全编译正常结束。此时应该点击“确定”按钮。全编译结果保存在.ptf 文件。

图 11-57　全编译通过对话框

Step 35：

在 Quartus II 的主菜单执行编程选项（Programmer），或者点击按钮，将编译好的*.sof（编译结果映像文件）烧写到 FPGA 里。

操作步骤有 3 个，它们是：

⑴单击 Hardware Setup 按钮，设定 USB-Blaster。

⑵选择可用的 USB-Blaster

⑶单击 Start 按钮，开始将*.sof 烧进 FPGA，当 100%出现时，表示烧录成功。

图 11-58 给出了 SOPC 烧写成功的快照。

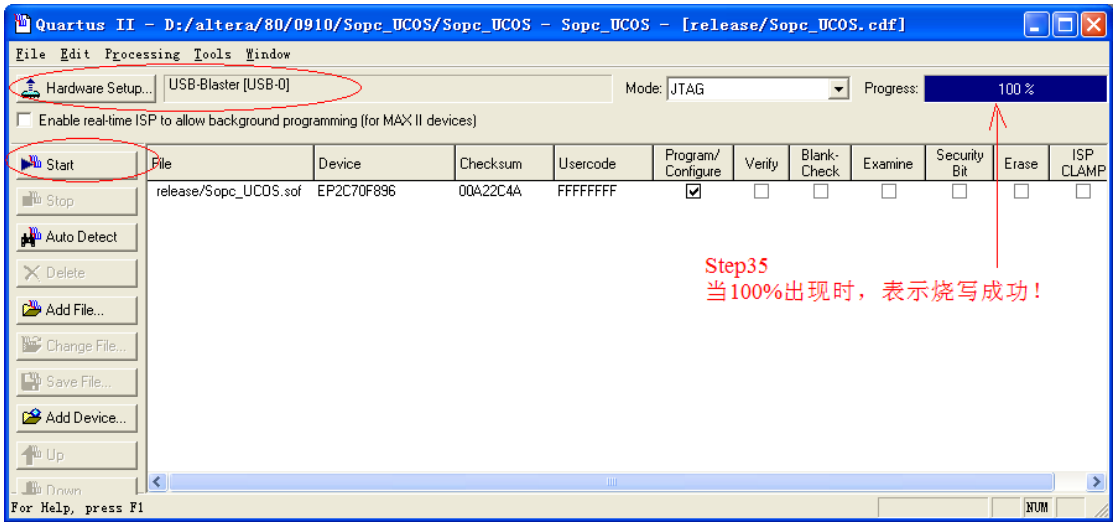


图 11-58　SOPC 烧写到 DE2-70 实验板的画面快照

注意：

至目前为止，**SOPC** 硬件部分的配置、编译和下载完成。接下来是 **Nios II** 的软件开发操作部分。这需要使用另外一个开发工具 **Nios II IDE**。

11.16 开发基于 Nios II 软核处理器的软件

Step 36：

有了 Nios II 硬件之后，就可以开发 Nios 的软件。从桌面的图标或者开始->程序启动 Nios II IDE 工具。进入 Nios II IDE 后，如果是第一次进入，会出现一个起始画面，参看图 11-59。

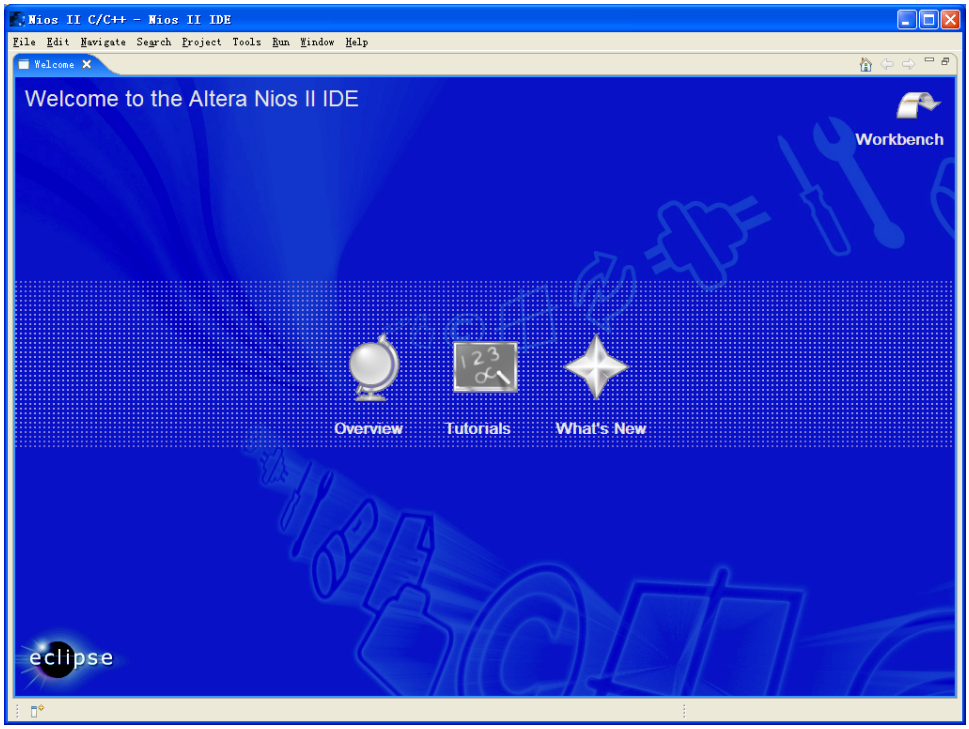


图 11-59　Nios-II IDE 的启动画面

此时，实验者应该点击右上角的 Workbench 按钮区域。如图 11-60 所示。

图 11-60　Nios II IDE 的欢迎画面

随后，如果在 Nios-II IDE 里还没有建立工作空间，则实验者要在 Workbench 界面里选择工作空间。涉及的菜单操作选项见图 11-61。

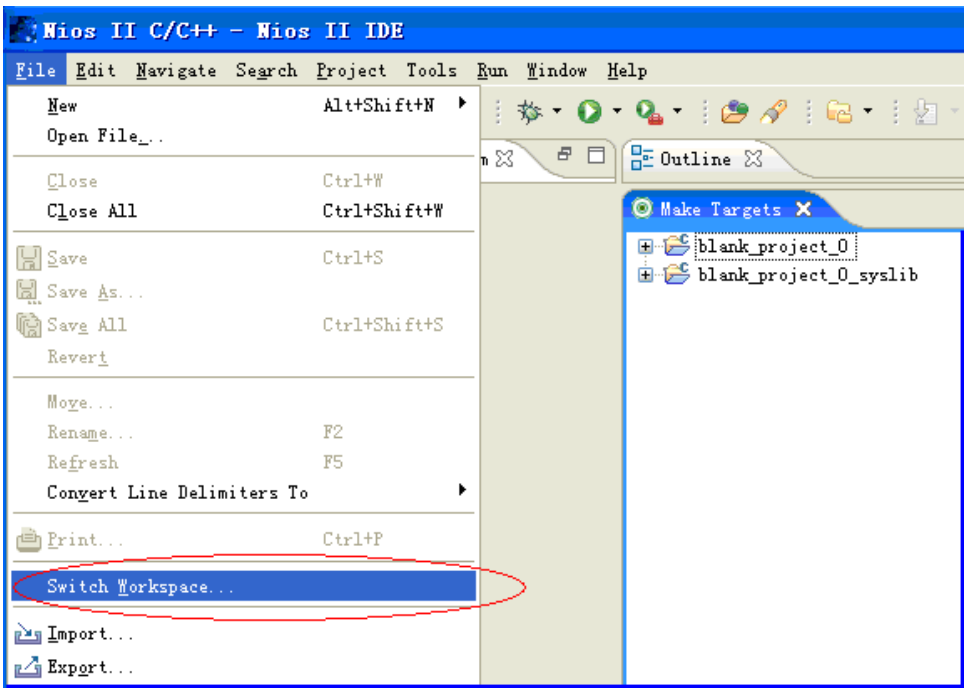


图 11-61 在 Workbench 对话框切换工作空间

参照前面的实验例子，实验者仍然可以把工作空间设置在<工程所在目录>\softawre。如图 11-62 所示。

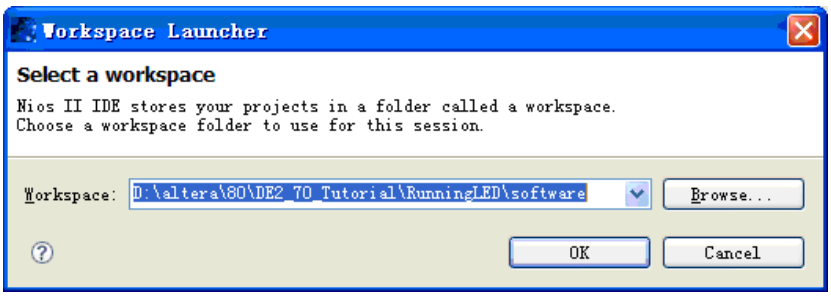


图 11-62　选择 Nios II IDE 的工作空间

随后，在 Nios II IDE 的主界面选择 File -> New -> Nios II C/C++ Application。如图 11-63 所示。

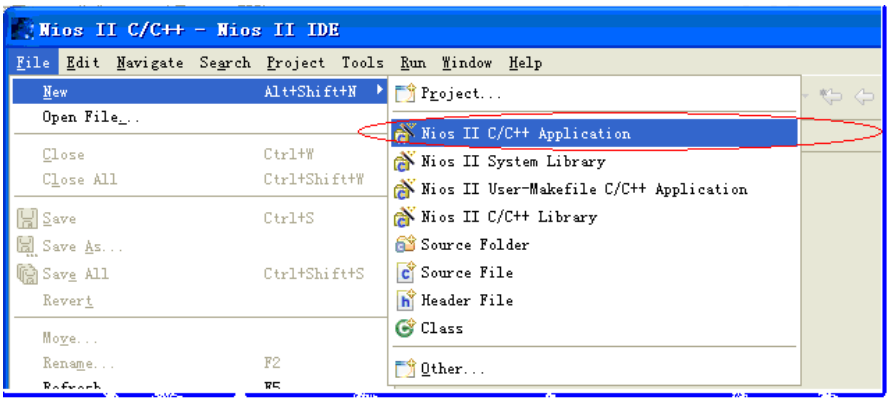


图 11-63　建立 Nios 的 C/C++应用程序工程

在随后弹出的新工程对话框里。执行以下操作：

⑴为这个工程取名 Hello_world_0。

⑵在 SOPC Builder System PTF Files 的编辑框里，填入 Quartus II 工程文件夹里由 SOPC

Builder 生成的 PTF 文件。参看图 11-64。也就是 c:\DE2-80\...具体路径…\nios_ii.ptf，这是先前 SOPC Builder 所 generate 的 ptf 文件。最后直接按 Finish 按钮完成。

注意：

PTF 文件是 SOPC Builder 对 Nios II IDE 的接口文件，要做到两者的配对性。必须避免填写错误。

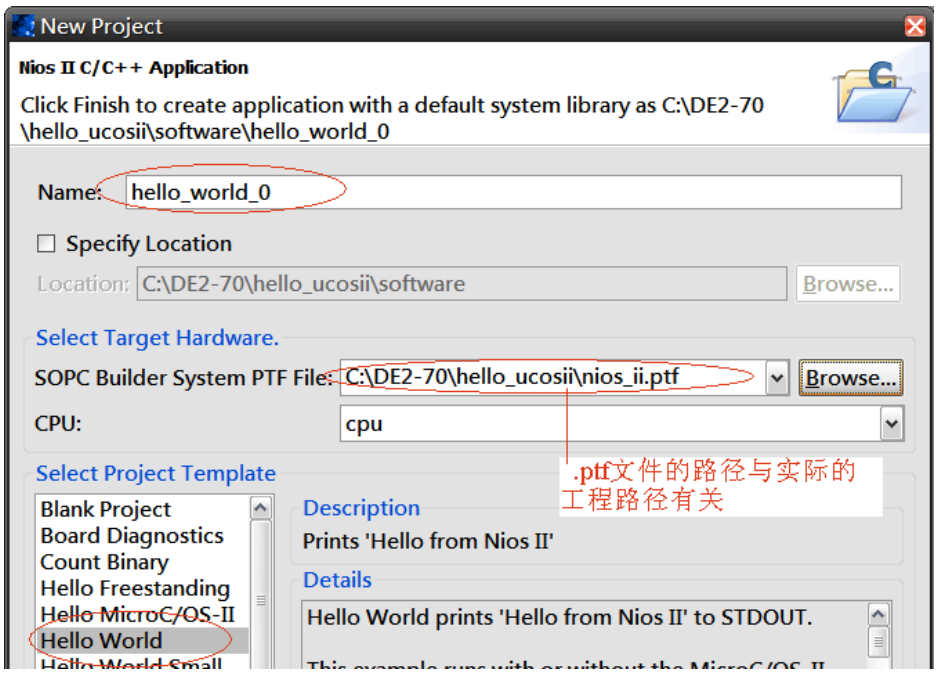


图 11-64 配置 Nios-ii 工程的属性为 Hello world

⑶在 **CPU** 属性输入框内保持缺省的 **CPU** 不变。

⑷在右侧的选择工程模板（**Select Project Template**）列表框里选择 Hello World。这是由于为了开发基于 Nios II 软核处理器的软件，首先需要使用最简单的 Hello World 程序来测试硬件配置是否符合要求。

11.17 测试硬件设计是否成功

Step 37：

为了进一步验证 SOPC 硬件配置/设计的正确性，首先要使用最简单的 Hello world 工程模板进行测试。

为此，在第 1 次配置 Nios-II 工程属性时（参看图 11-64），选择使用 **Hello World 工程模板**以测试硬件是否设计成功。

注意：Quartus II 能正常编译，不代表硬件设计成功，SOPC Builder 各 controller 的参数设定错误、clk 设定错误、top module 连线错误、Quartus II 设定错误...等，都可能造成 Nios II 无法执行，所以先用最简单的 Hello World 测试硬件，若连 Hello World 都不能执行，软件部分就不用继续了，先回去找硬件部分的 bug。

在图 11-64 的工程属性对话框的右则选择 Hello World template，并指定 SOPC Builder System PFT。

随后，Nios-ii IDE 会给出一个对话框，参看图 11-65。这时使用缺省的单选按钮配置。

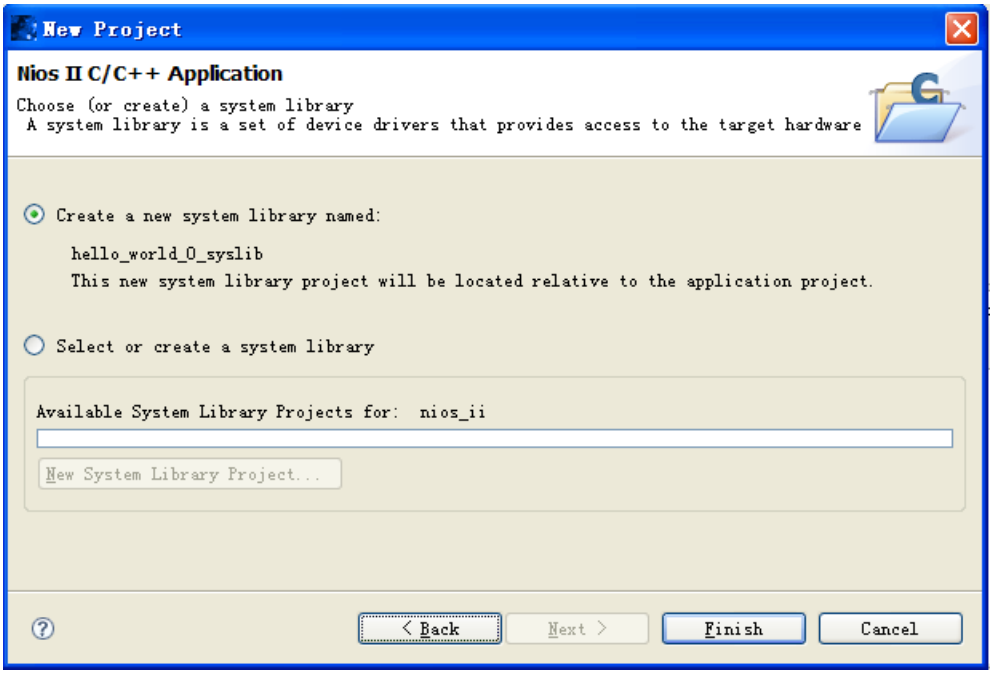


图 11-65　选择/创建 Hello World 工程模板的系统函数库

Nios II EDS 会根据你选的 project template 与 SOPC Builder System File 产生 2 个 project：

1. **hello_world_0：Nios II Software project。**
2. **hello_world_0_syslib：Nios II System Library project。**

参看图 11-66。对准 hello_world_0 工程，按鼠标右鍵，在弹出式菜单选 System Library Properties 项目。

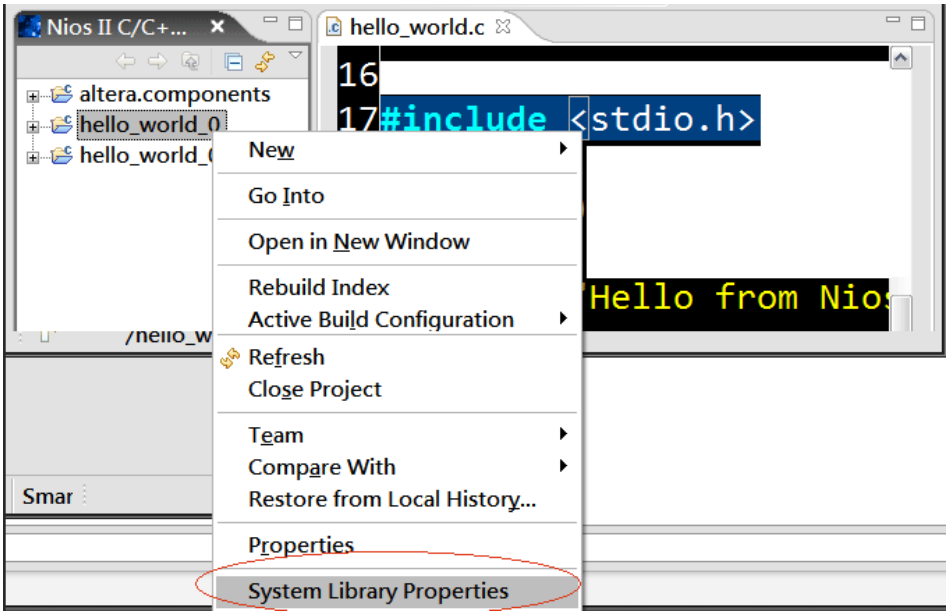


图 11-66　选择 hello_world_0 工程的 System Library Properties 项目

注意

System Library = HAL(Hardware Abstraction Layer) = BSP(Board Support Package) = Driver。这个等式的含义是：系统库=硬件抽象层=板级支持包=驱动程序。

再在弹出的 System Library Properties 属性对话框里，选择 System clock timer 栏目值为 sys_clk_timer。参看图 11-67。

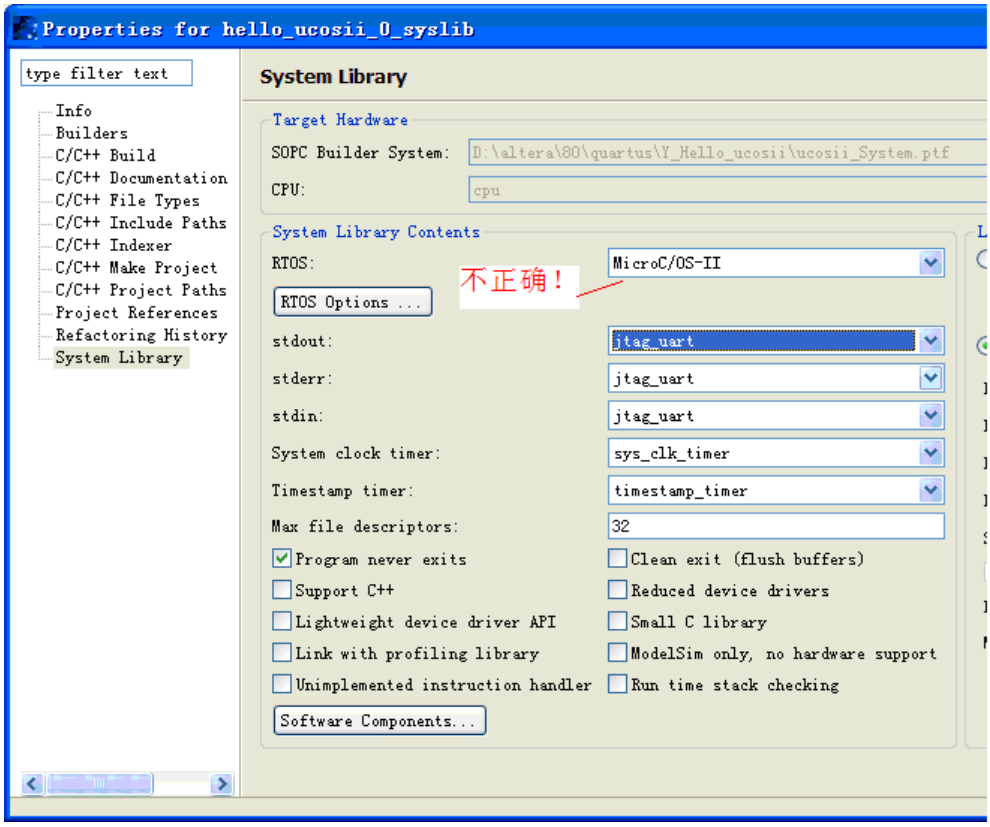


图 11-67　system Library 的属性

另外将软件全部跑在 SDRAM，当然也可以跑在其他存储器，只是因为 SDRAM 容量最大，而且 SDRAM 的 clk 需要 phase shift，所以最常出现问题都是在 SDRAM，所以在此特别使用 SDRAM，至于其他存储器可自行测试。

选择 Hello_world_0 工程，执行 Rus As-> Nios II Hardware。操作步骤分别为 A/B/C。参看图 11-68。

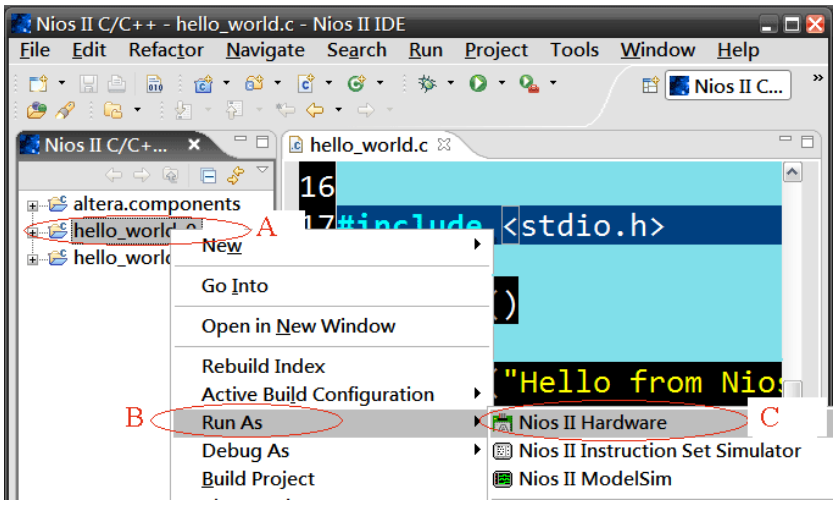


图 11-68　在 Nios II 硬件上执行最简单的 Hello World 程序

此时 Nios II 软件会透过 JTAG UART 传到 SDRAM 开始执行。

若第一次执行，Nios II EDS 会编译整个 System Library，需要一点时间，约 2 到 3 分钟。最后执行结果，Nios II 由 JTAG UART 传回 Hello from Nios II 到 PC 的 Console。

若能正确执行最简单的 C 语言测试程序 Hello World，则表示 SOPC 硬件正确，可以继续基于该 SOPC 的软件开发。

11.18 设定所有未使用引脚为三态输入引脚

Step 38：

接下来，在 Nios ii 的 SOPC 硬件平台上测试最简单的 **Hello µC/OS-II** 的应用项目是否能够执行。到目前为止，单任务（single thread）的软件程序已经能跑在 Nios II 上，若要让 Nios II 能跑多任务程序（multi thread），就必须靠 OS 才行。

为此，必须在 C/C++的工程属性中选择 Hello MicroC/OS-II。参看图 11-69。

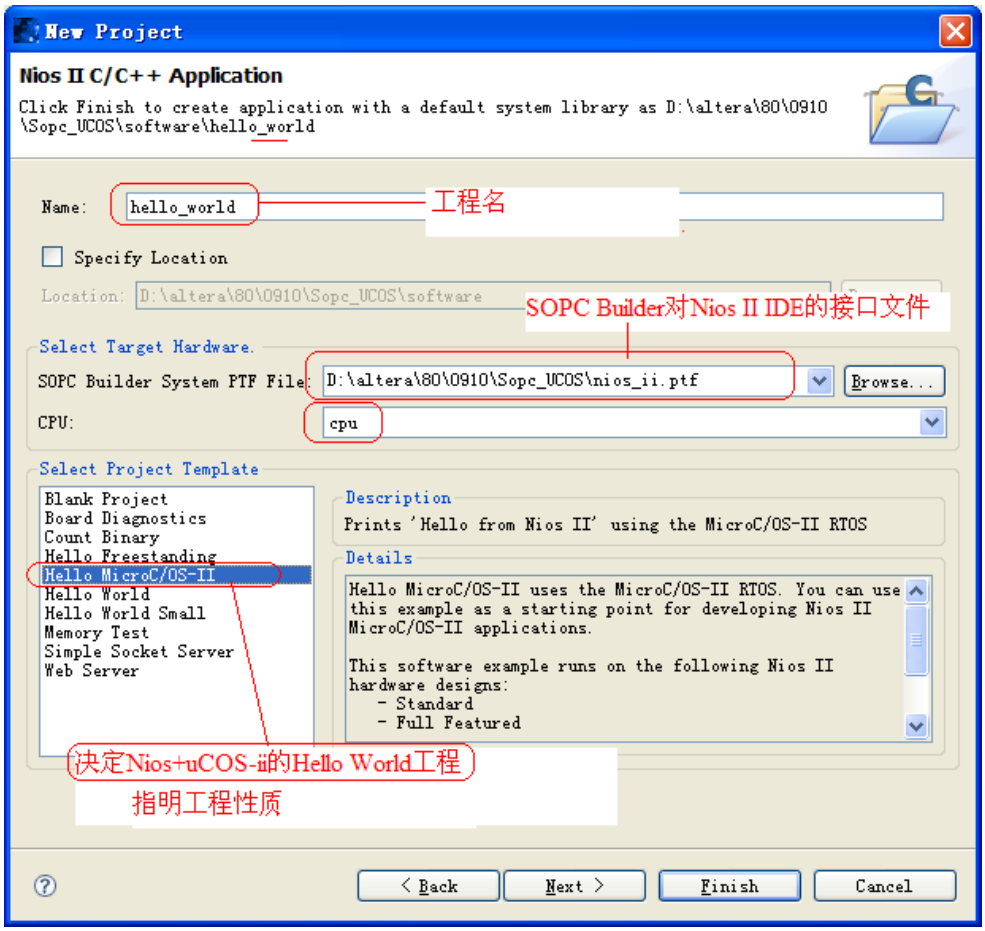


图 11-69 配置 Nios-ii 工程的属性为 Hello MicorC/OS-II

Atlera 公司的系统工程师已经在 Nios II EDS 中将 µC/OS-II 软件包集成在它的 System Library 里面。在这个实验中，实验者需要编写一个简单的多任务应用程序以验证 SOPC+UCOS-II 在 FPGA 运行的可行性。

注意：

选择工程模板非常重要。只有选择 MicroC/OS-II 工程模板才能够开发一个最简单的 SOPC+NIOS 的应用程序。

11.19 编写 µC/OS-II 的多任务控制程序

Step 39：

开发一个多任务且控制硬件的程序

能成功执行 Hello Wrold 与 Hello MicroC/OS-II，表示软硬件都已经设定妥当，可以正式用 C 写程序了。

再用 Hello MicroC/OS-II template 建立一个 project。
将 hello_ucosii.c 改成如以下的程序
hello_ucosii.c / C

```
#include <stdio.h>
#include "includes.h"
#include "system.h"
#include <io.h>
/* Definition of Task Stacks */
#define      TASK_STACKSIZE             2048
OS_STK       task1_stk[TASK_STACKSIZE];
OS_STK       task2_stk[TASK_STACKSIZE];
/* Definition of Task Priorities */
#define TASK1_PRIORITY          1
#define TASK2_PRIORITY          2
/* Prints "Hello World" and sleeps for three seconds */
void task1(void* pdata)
{
  while (1)
  {
    printf("Hello uCOS-II\n");
    OSTimeDlyHMSM(0, 0, 3, 0);
  }
}
/* Prints "Hello World" and sleeps for three seconds */
void task2(void* pdata)
{
  unsigned int i;
  while (1)
  {
    // read switch
    i = IORD(PIO_SW_BASE, 0);
    // write ledr
    IOWR(PIO_LEDR_BASE, 0, i);
  }
}
/* The main function creates two task and starts multi-tasking */
int main(void)
{
  OSTaskCreateExt(task1,
                  NULL,
                  (void *)&task1_stk[TASK_STACKSIZE-1],
                  TASK1_PRIORITY,
                  TASK1_PRIORITY,
                  task1_stk,
```

```
                    TASK_STACKSIZE,
                    NULL,
                    0);
  OSTaskCreateExt(task2,
                    NULL,
                    (void *)&task2_stk[TASK_STACKSIZE-1],
                    TASK2_PRIORITY,
                    TASK2_PRIORITY,
                    task2_stk,
                    TASK_STACKSIZE,
                    NULL,
                    0);
  OSStart( );
  return 0;
}
```

代码注释

```
#include "system.h"
#include <io.h>
```

system.h 记载着 SOPC Builder 里各 controller 的信息，稍后会讨论。

io.h 定义了 IORD( )与 IOWR( )两个巨集（宏），可以利用此巨集（宏）存取各 controller 的 register。

```
void task1(void* pdata)
{
  while (1)
  {
    printf("Hello uCOS-II\n");
    OSTimeDlyHMSM(0, 0, 3, 0);
  }
}
void task2(void* pdata)
{
  unsigned int i;
  while (1)
  {
    // read switch
    i = IORD(PIO_SW_BASE, 0);
    // write ledr
    IOWR(PIO_LEDR_BASE, 0, i);
  }
}
i = IORD(PIO_SW_BASE, 0);          // 读取 SW 目前的值。
IOWR(PIO_LEDR_BASE, 0, i);         // 将 SW 的值马上给 LEDR 显示。
```

也就是说，若 SW 为 ON 时，LEDR 会亮，若 SW 为 OFF 时，LEDR 就不亮。我们是怎么做到的呢?

我们通过 IORD( )宏，去读取 pio_sw controller 目前 register 的值，然后通过 IOWR( )宏，将值写入 pio_ledr controller 的 register，让 LEDR 显示。

问题来了!!我们怎么知道 pio_sw 与 pio_ledr 在哪里?

特别注意：

实际上 pio_sw 与 pio_ledr 在#include <system.h>做了定义。

现在让我们查看一下 system.h 里面的内容。

打开下面的文件

C:\DE2-70\hello_ucosii\software\hello_ucosii_1_syslib\Debug\system_description\system.h

考察第 459 行开始处

```
#define PIO_SW_NAME "/dev/pio_sw"
#define PIO_SW_TYPE "altera_avalon_pio"
#define PIO_SW_BASE 0x094110c0
#define PIO_SW_SPAN 16
#define PIO_SW_DO_TEST_BENCH_WIRING 0
#define PIO_SW_DRIVEN_SIM_VALUE 0
#define PIO_SW_HAS_TRI 0
#define PIO_SW_HAS_OUT 0
#define PIO_SW_HAS_IN 1
#define PIO_SW_CAPTURE 0
#define PIO_SW_DATA_WIDTH 18
#define PIO_SW_RESET_VALUE 0
#define PIO_SW_EDGE_TYPE "NONE"
#define PIO_SW_IRQ_TYPE "NONE"
#define PIO_SW_BIT_CLEARING_EDGE_REGISTER 0
#define PIO_SW_FREQ 100000000
#define ALT_MODULE_CLASS_pio_sw altera_avalon_pio
```

其中的#define PIO_SW_BASE 0x094110c0 定义了 SOPC Builder 为 pio_sw 所规定的端口位地址。

而 system.h 也是 Nios II EDS 根据 nios_ii.ptf 所产生的。

考察第 413 行开始处

```
#define PIO_LEDR_NAME "/dev/pio_ledr"
#define PIO_LEDR_TYPE "altera_avalon_pio"
#define PIO_LEDR_BASE 0x094110a0
#define PIO_LEDR_SPAN 16
#define PIO_LEDR_DO_TEST_BENCH_WIRING 0
#define PIO_LEDR_DRIVEN_SIM_VALUE 0
#define PIO_LEDR_HAS_TRI 0
#define PIO_LEDR_HAS_OUT 1
#define PIO_LEDR_HAS_IN 0
#define PIO_LEDR_CAPTURE 0
#define PIO_LEDR_DATA_WIDTH 18
#define PIO_LEDR_RESET_VALUE 0
#define PIO_LEDR_EDGE_TYPE "NONE"
#define PIO_LEDR_IRQ_TYPE "NONE"
```

#define PIO_LEDR_BIT_CLEARING_EDGE_REGISTER 0
#define PIO_LEDR_FREQ 100000000
#define ALT_MODULE_CLASS_pio_ledr altera_avalon_pio

其中的#define PIO_LEDR_BASE 0x094110a0 定义了 SOPC Builder 为 pio_ledr 所规定的位址。设定标准输出为 LCD

执行结果

通过 μC/OS-II 的控制，一个 thread 在 LCD 显示，另一个 thread 控制 SW 与 LEDR。

问题：

1.为什么只有 SDRAM 的 clock 需要 phase shift -65 度?为什其他的硬件的 clk 都不需要 phase shift?

2.可以使用 pointer，而不使用 IORD( )与 IOWR( )吗?解释你的理由。

```
#define SW (unsigned int *)SW_BASE
#define LEDR (unsigned int *)LEDR_BASE
void task2(void* pdata)
{
  while (1)
  {
    *LEDR = *SW;
  }
}
```

11.20 替换练习

Step 40：

1）试着将 Hello World 代码执行在 On-Chip Memory 上，并解释你所使用的方法。

2）将含有两个任务的 Hello World 代码改写含有三个多任务的 Hello World 代码，输出到 DE2-70 实验板的 LCD 和 LEDR 上。并且观察输出结果是否正确。

代码示例如下：

```
#include <stdio.h>
#include "includes.h"

/* Definition of Task Stacks */
#define   TASK_STACKSIZE       2048
OS_STK    task1_stk[TASK_STACKSIZE];
OS_STK    task2_stk[TASK_STACKSIZE];
OS_STK    task3_stk[TASK_STACKSIZE];

/* Definition of Task Priorities */

#define TASK1_PRIORITY      1
#define TASK2_PRIORITY      2
#define TASK3_PRIORITY      3

/* Prints "Hello World1" and sleeps for one seconds */
void task1(void* pdata)
```

```
{
  while (1)
  {
    printf("Hello from task1\n");
    OSTimeDlyHMSM(0, 0, 1, 0);   /*  延时 1 秒  */
  }
}
/* Prints "Hello World2" and sleeps for three seconds */
void task2(void* pdata)
{
  while (1)
  {
    printf("Hello from task2\n");
    OSTimeDlyHMSM(0, 0, 3, 0);   /*  延时 3 秒  */
  }
}
/* Prints "Hello World3" and sleeps for two seconds */
void task3(void* pdata)
{
  while (1)
  {
    printf("Hello from task3\n");
    OSTimeDlyHMSM(0, 0, 2, 0);   /*  延时 2 秒  */
  }
}
/* The main function creates two task and starts multi-tasking */
int main(void)
{
  OSTaskCreateExt(task1,
                  NULL,
                  (void *)&task1_stk[TASK_STACKSIZE],
                  TASK1_PRIORITY,
                  TASK1_PRIORITY,
                  task1_stk,
                  TASK_STACKSIZE,
                  NULL,
                  0);


  OSTaskCreateExt(task2,
                  NULL,
                  (void *)&task2_stk[TASK_STACKSIZE],
                  TASK2_PRIORITY,
                  TASK2_PRIORITY,
```

```
                    task2_stk,
                    TASK_STACKSIZE,
                    NULL,
                    0);

    OSTaskCreateExt(task3,
                    NULL,
                    (void *)&task3_stk[TASK_STACKSIZE],
                    TASK3_PRIORITY,
                    TASK3_PRIORITY,
                    task3_stk,
                    TASK_STACKSIZE,
                    NULL,
                    0);
    OSStart( );
    return 0;
}
```

11.21 如何运行一个现存的 Nios+UCOS-II 应用程序

根据我们课题组的实践，总结出对于既有的基于 Nios 软核处理器加 UCOS-II 操作系统应用工程（外购来的或者拷贝来的），有以下操作失败场合和操作成功场合。

失败场合 1，工程文件从一个 PC 主机移动到另外一台 PC 主机。参看图 11-70。该图表示工程文件夹移动到另外一台 PC 之后，Quartus II 不能完成全编译。

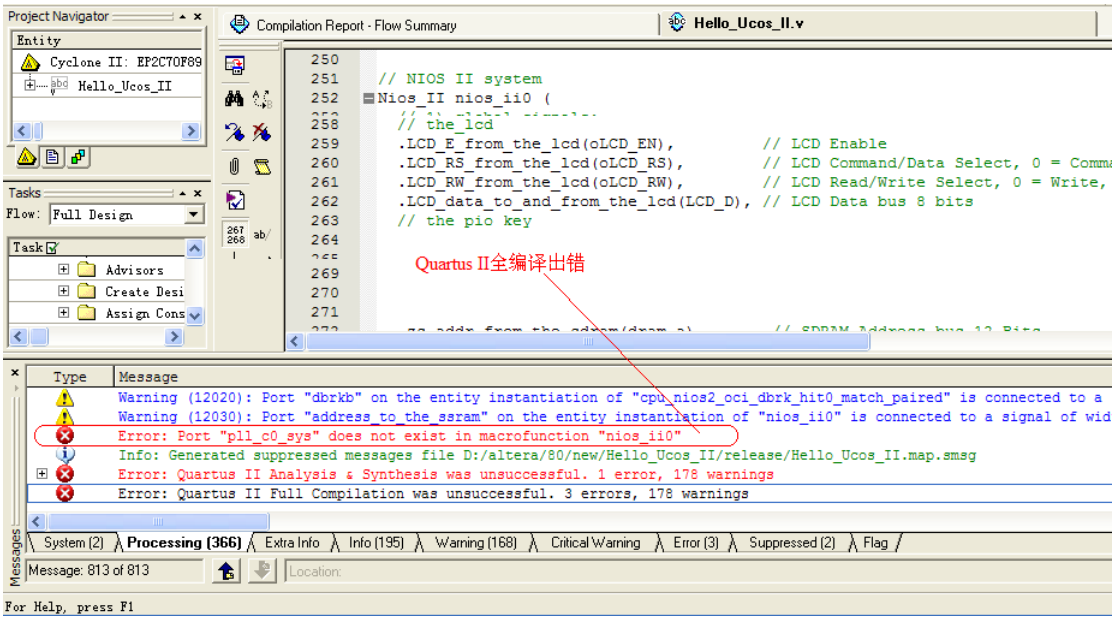


图 11-70　Nios-ii 工程移动到其它机器之后再在 Quartus 下全编译会出错

失败场合 2，对于一个原来在本 PC 机上正常编译通过的工程。如果拷贝出去之后，再拷贝近来，则要注意编译次序。即使该工程自完工之后没有更动过，但是如果按照先编译 SOPC Builder 软核处理器，再全编译 Quartus II 的工程，则在 NIOS –II 的集成开发环境下，运行 C/C++工程还是会出错。参看图 11-71。

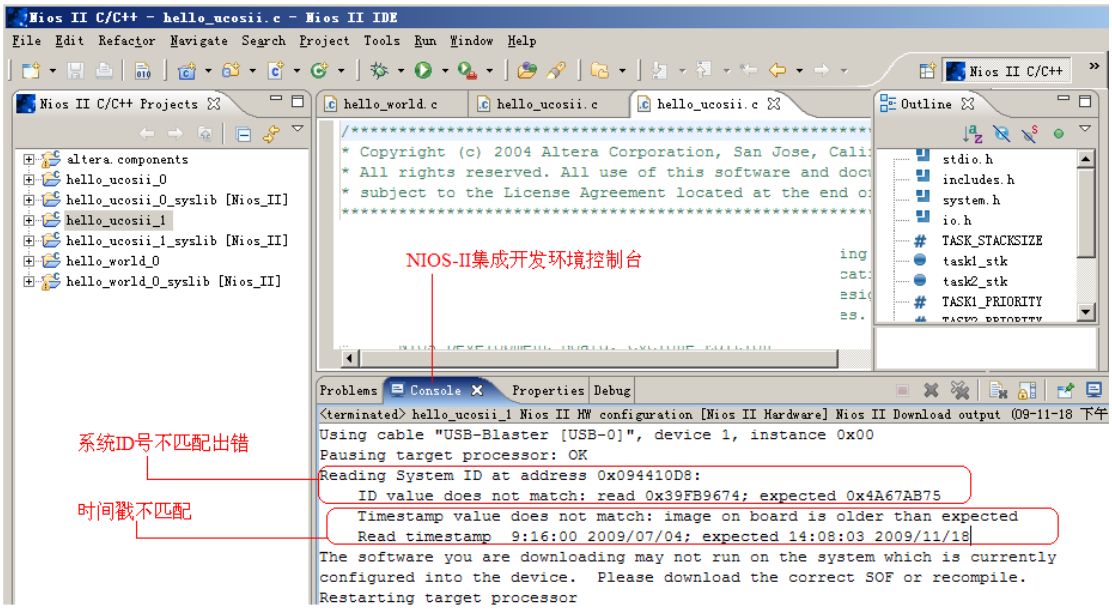


图 11-71　既有的 Nios-ii 工程如果编译次序不对运行时也会报错

成功场合 1，对于一个原来在本 PC 机上正常编译通过的工程。无论向外传输路径如何，再拷贝到本 PC 机时，只要按照下列的顺序进行操作，就可以正常运行。

①编译 **SOPC Builder** 的软核处理器工程。参看图 11-72。

②编译 **Quartus II** 的 **FPGA** 工程。参看图 11-73。

③在 **Nios II** 集成开发环境打开该工程的 **C/C++**应用工程

④运行输出到 JTAG_UART 口的 C 代码。参看图 11-74。

⑤运行输出到 LCD 的 C 代码。参看图 11-75。

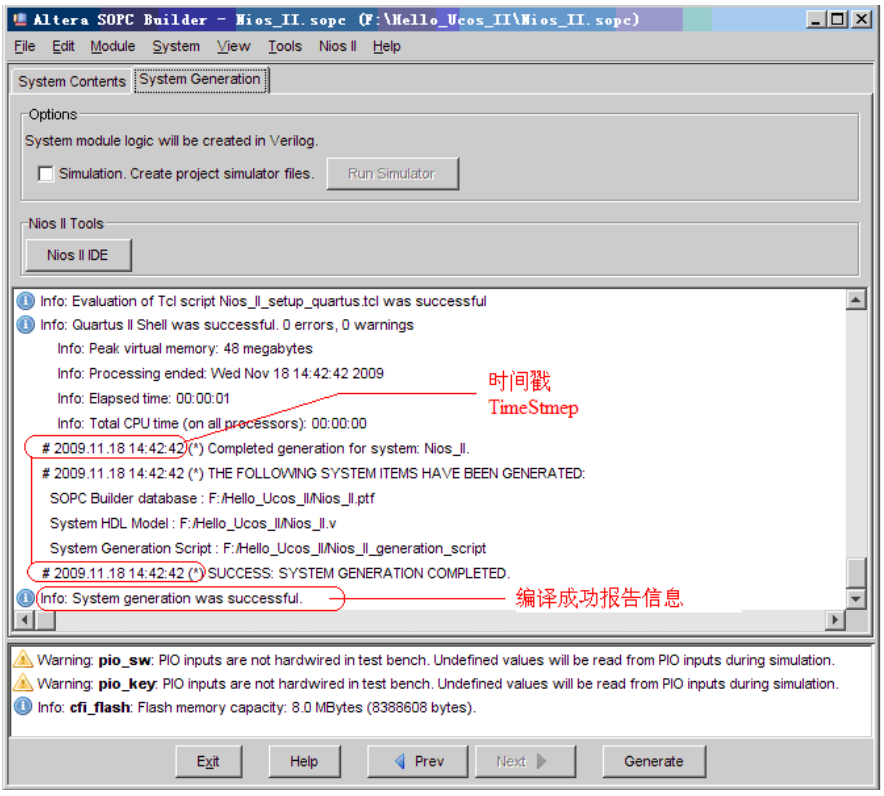


图 11-72　既有 Nios+uCOS-II 的 SOPC Builder 工程编译成功

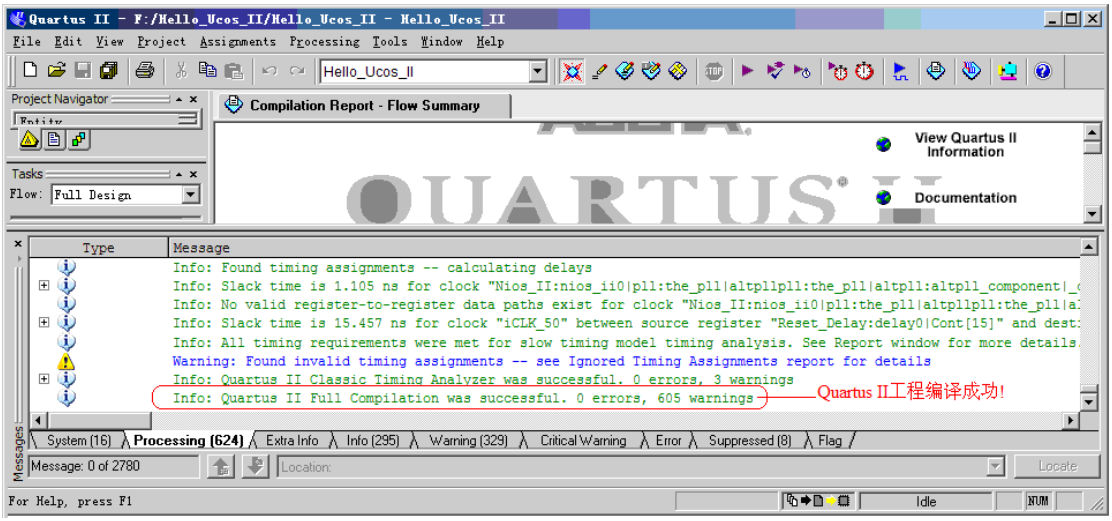


图 11-73　既有 Quartus 工程编译成功

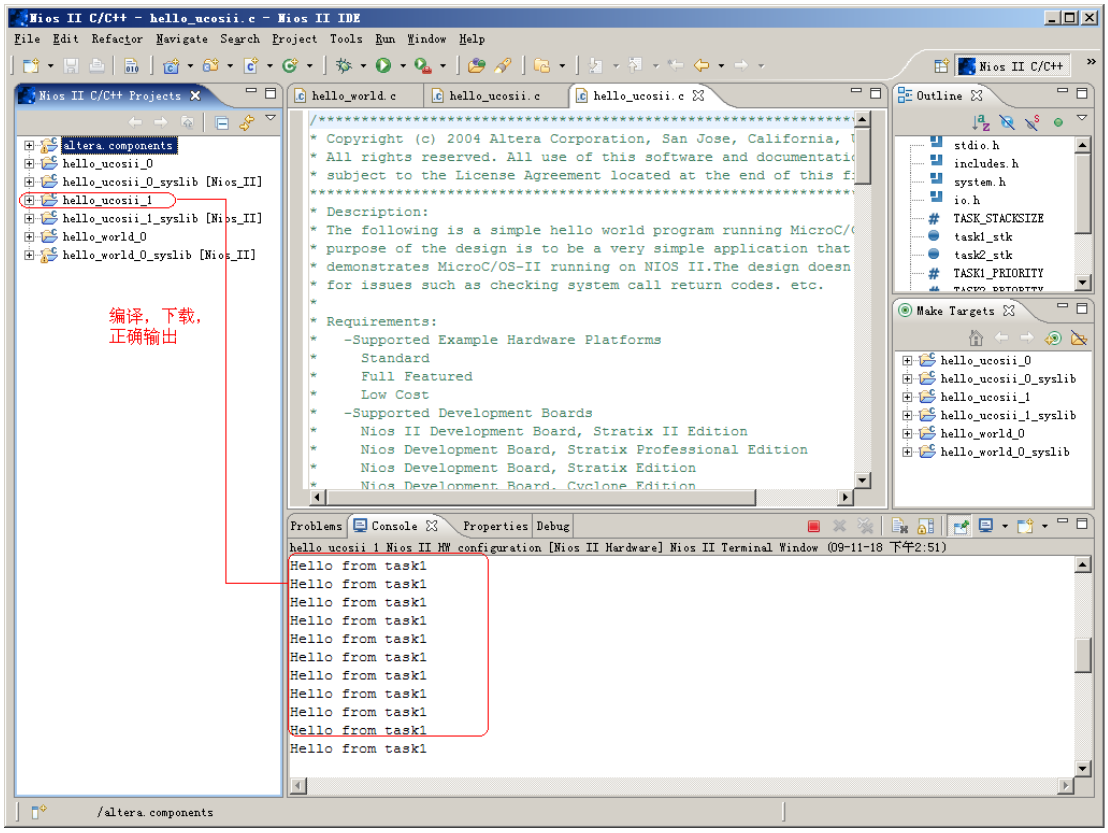


图 11-74　NIOS-II 环境下运行的 C 程序向 JTAG_UART 口输出 Hello 信息成功

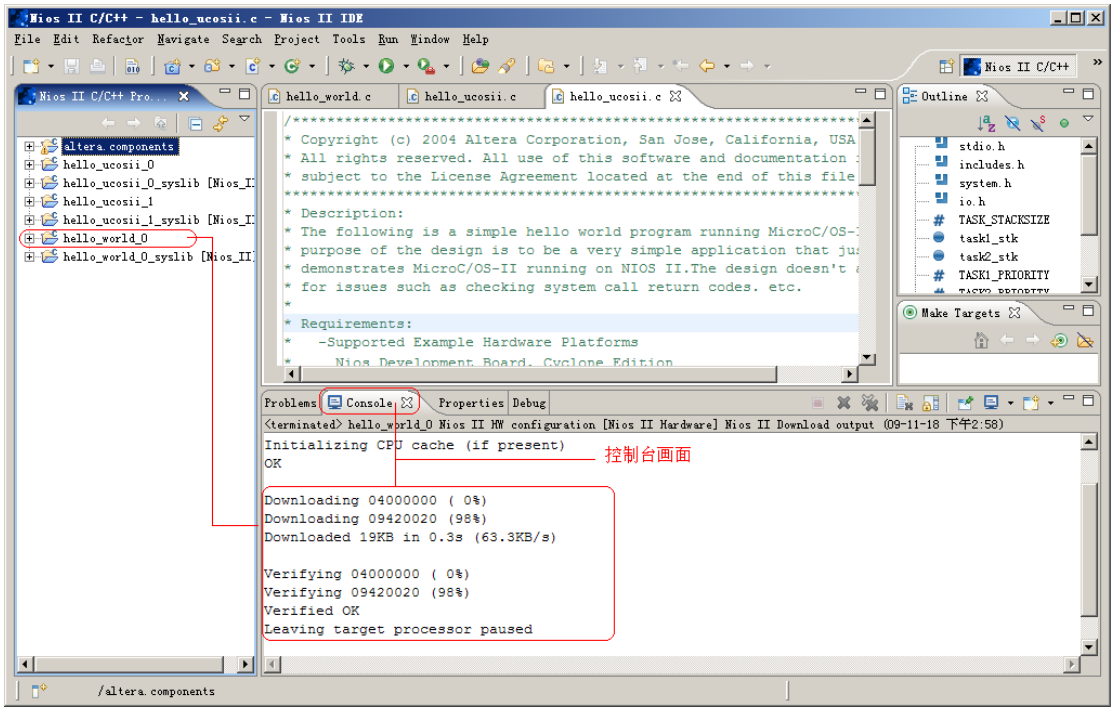


图 11-75　NIOS-II 环境下运行的 C 程序向 LCD 输出 Hello 信息成功

本实验指导结束

第 12 章　VHDL 语言实验项目 12 例

在 DE2-70 实验板上可以运行的 HDL 语言实验程序有两种：Verilog 和 VHDL。前面我们用 10 章篇幅介绍了典型的 DE2-70 实验项目，这些实验项目的代码主要涉及 Verilog。本章以 VHDL 为硬件描述语言，介绍一组初学者的数字系统入门级实验项目。其目的是让同学们更好地掌握 DE2-70 平台的使用。

12.1　VHDL 标准逻辑位实验

IEEE 库的 STD_LOGIC_1164 程序包中定义了 STD_LOGIC 数据类型，通常被称为标准逻辑位。这种标准数据类型具有 9 种状态，但是 Quartus II 综合器只支持其中 4 种状态，它们是‘1’=逻辑 1；‘0’=逻辑 0；‘Z’=高阻；‘X’=不定。

下面给出一个描述 VHDL 标准逻辑位数据类型的 VHDL 实验代码。它在 DE2-70 实验平台上调试通过。通过本实验读者主要了解 VDHL 四种标准逻辑位的仿真显示。操作步骤如下：

1. 建立工程文件，命名为 **StdLogic**。
2. 设置规范的编译结果文件路径。
3. 建立 **VHDL** 文件。

源代码如下：

```
-- Standard Logic Bit in VHDL --

LIBRARY ieee;                    --打开 IEEE 库
USE ieee.std_logic_1164.ALL;     --打开 IEEE 库中 STD_LOGIC_1164

ENTITY StdLogic IS               --定义程序的实体
PORT(     a,b: IN std_logic;          --定义输入端口
          x,y,z: OUT std_logic;  --定义输出端口
          xyz: OUT std_logic_vector(1 DOWNTO 0)      --定义输出端口
          );
END StdLogic;

ARCHITECTURE first OF stdlogic IS                          --定义结构体
BEGIN x<=a AND b;
y<='1'; Z<='Z';
xyz<="01";

END first;
```

4. 分析与解析。
5. 分析与综合。
6. 建立波形文件，添加结点，确定时钟（clock）信号之类的时钟周期，并且为其他结点添加合适的信号波形。
7. 电路功能仿真。参看图 12-1。
8. 电路时序仿真。参看图 12-2。
9. 存盘保存工程文件。

初学者注意：

本实验程序主要掌握 VHDL 语言的源代码编辑、分析与解析、全编译、建立波形文件、波形文件中的结点设置、功能仿真和时序仿真。主要观察 4 种标准的 VHDL 逻辑位取值，不需要进行 DE2-70 实验板的结点分配和下载测试。

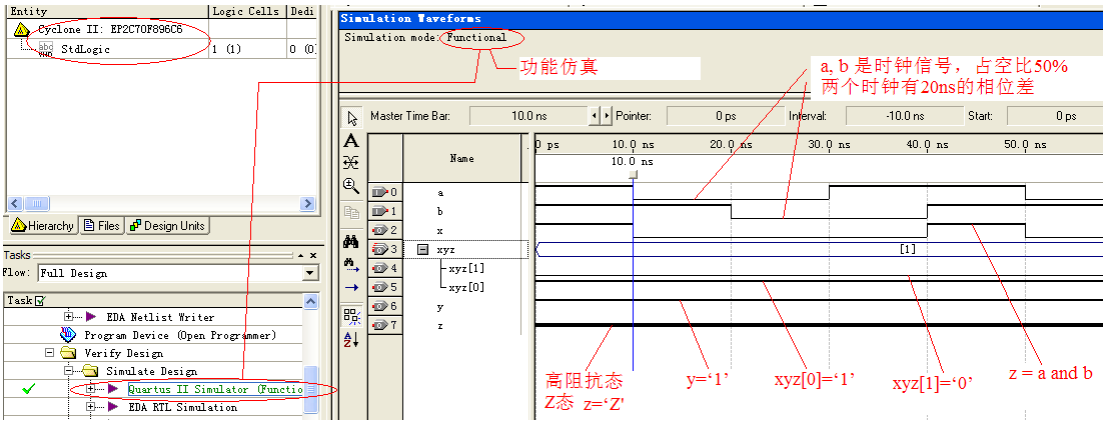


图 12-1　标准逻辑位数据类型实验的功能仿真快照

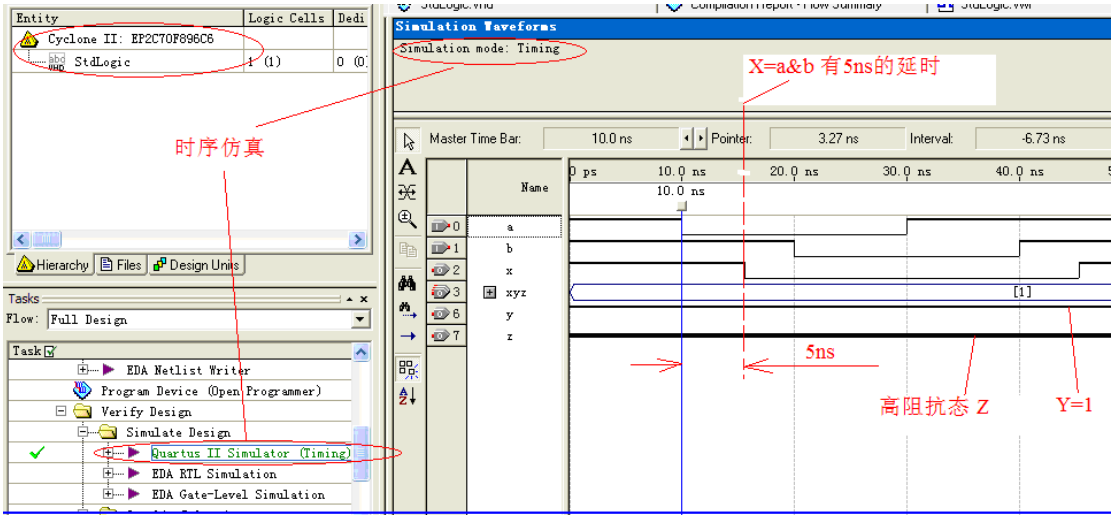


图 12-2　标准逻辑位数据类型实验的时序仿真快照

◆本实验指导结束

12.2　8-3 编码器实验

这是一个组合电路的 VHDL 语言实验代码，在 DE2-70 实验平台上调试通过。

1. 建立工程文件，命名为 encode2。
2. 设置规范的编译结果文件路径。
3. 建立 VHDL 文件。

源代码如下：

```
--2008-08-15 源代码源自[3]  --
-- Tested OK --
LIBRARY ieee;                          --打开IEEE库
USE ieee.std_logic_1164.ALL;           --打开IEEE库中STD_LOGIC_1164
```

```
ENTITY encoder2 IS                          --定义程序的实体
PORT(x: IN std_logic_vector(7 DOWNTO 0);    --定义输入端口
     y: OUT std_logic_vector(2 DOWNTO 0));  --定义输出端口
END encoder2;                               --实体结束语句

ARCHITECTURE second OF encoder2 IS          --定义结构体
BEGIN                                       --开始电路功能描述
    WITH x SELECT
    y<= "000"WHEN "11111110",
        "001"WHEN "11111101",
        "010"WHEN "11111011",
        "011"WHEN "11110111",
        "100"WHEN "11101111",
        "101"WHEN "11011111",
        "110"WHEN "10111111",
        "111"WHEN "01111111",
        "ZZZ"WHEN OTHERS;
END second;
```

4. 分析与解析。
5. 分析与综合。
6. 建立波形文件，添加结点，确定时钟（clock）信号之类的时钟周期，并且为其他结点添加合适的信号波形。
7. 电路功能仿真。
8. 生成功能仿真网表。
9. 分配引脚。参看图 12-3。

初学者注意：

在 DE2-70 实验板引脚分配时，如果将 SW[7]按顺序分配给 x[7]时，编译会报错。错误信息是：

Can't place pins assigned to pin location Pin_AD25 (IOC_X95_Y2_N1)

正确的解决方法如下：

步骤 1：选择主菜单->Assignment->Device..，会弹出一个器件选择对话框，如图 12-3 所示。单击其中的 Device and Pin Options…按钮。

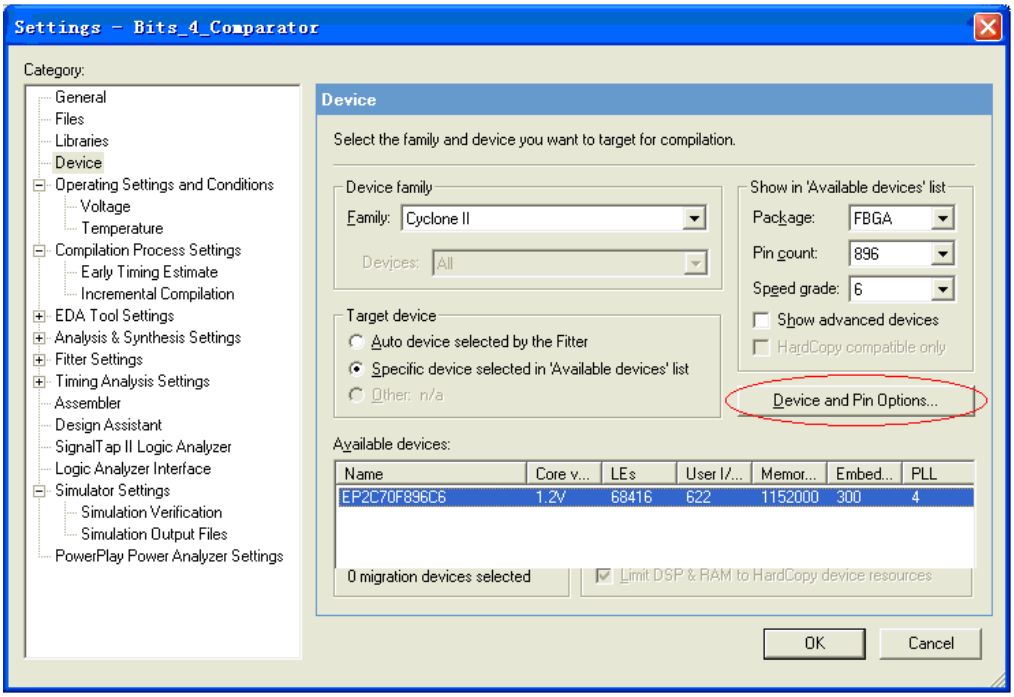


图 12-3　器件选择对话框

步骤 2：在 Device and Pin Options 对话框里点击 Dual-Purpose Pins 选项卡。参看图 12-4。

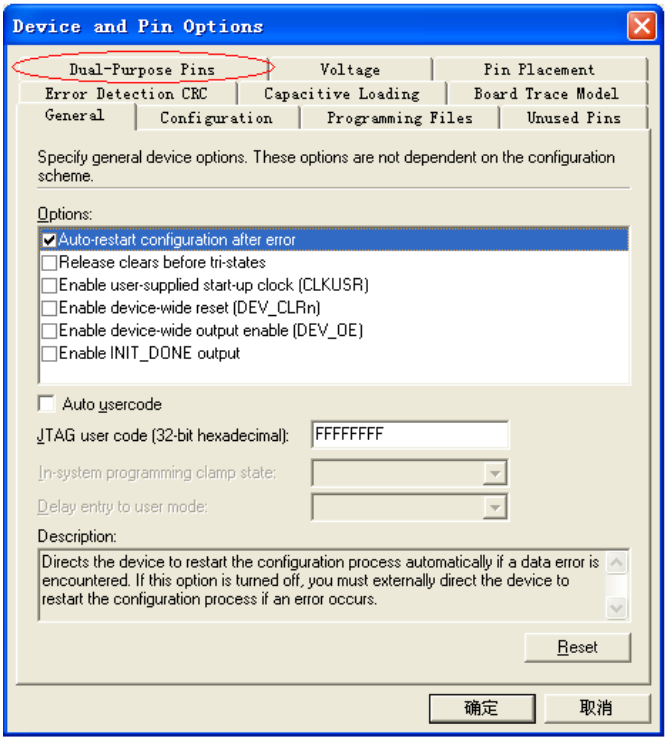


图 12-4　点击 Dual-Purpose Pins 选项卡

步骤 3：将 Dual-Purpose Pins 选项卡里面的信号 nCEO 由缺省的 Use as programming pin 设置更改为 Use as regular I/O。参看图 12-5。

图 12-5　更改 nCEO 的设定值

nCEO 的信号值改变之后，再进行编译就没有出错现象了。

还有一个变通的解决方法。就是为 x[7]信号分配另外一个引脚。

在本实验指导书中，将 SW[8]分配给 x[7]，编译成功。参看图 12-6。

10. 全编译。

Named: * Edit:

	Node Name	Direction	Location	I/O Bank	Vref Group	I/O Standard
1	x[7]	Input	PIN_AD24	6	B6_N3	3.3-V LVTTL (default)
2	x[6]	Input	PIN_AC23	6	B6_N3	3.3-V LVTTL (default)
3	x[5]	Input	PIN_AC24	6	B6_N3	3.3-V LVTTL (default)
4	x[4]	Input	PIN_AC26	6	B6_N3	3.3-V LVTTL (default)
5	x[3]	Input	PIN_AC27	6	B6_N2	3.3-V LVTTL (default)
6	x[2]	Input	PIN_AB25	6	B6_N2	3.3-V LVTTL (default)
7	x[1]	Input	PIN_AB26	6	B6_N2	3.3-V LVTTL (default)
8	x[0]	Input	PIN_AA23	6	B6_N2	3.3-V LVTTL (default)
9	y[2]	Output	PIN_AJ5	8	B8_N3	3.3-V LVTTL (default)
10	y[1]	Output	PIN_AK5	8	B8_N3	3.3-V LVTTL (default)
11	y[0]	Output	PIN_AJ6	8	B8_N2	3.3-V LVTTL (default)
12	<<new node>>					

iSW[8]
iSW[6:0]
LEDR[2:0]

图 12-6　8-3 编码器的引脚分配

11. 时序仿真。

图 12-7　8-3 编码器的测试快照之一

12. 生成映像文件，下载到电路板。

13. 进行测试操作。参看图 12-7。该图是测试结果的一个快照，读者可注意到 SW[6]开关扳到了低电平的位置。其余均为高电平，这样编码输出就是“110”。这是 8-3 编码器的正确测试结果。

◆本实验指导结束

12.3　3-8 译码器 74LS138 实验

74LS138 是常用的小规模集成电路芯片，用于译码。它有 3 个使能引脚，3 个译码输入引脚和 8 个低电平的译码输出引脚。其逻辑电路如图 12-8 所示。

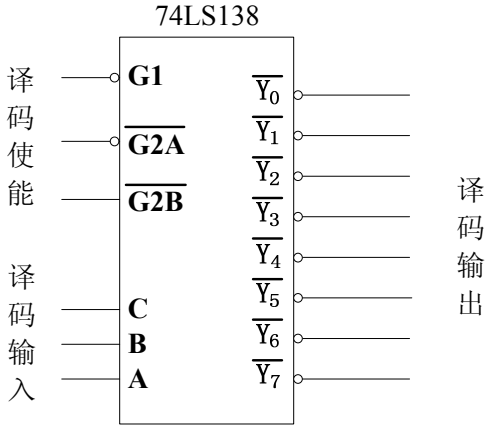


图 12-8　74LS138 译码器的引脚图

本实验采用 VHDL 语言描述 74LS138 译码器，以下列出实验程序的 VHDL 源代码。这个实验程序在 DE2-70 实验平台上调试通过。

1. 建立工程文件，命名为 DCD74LS138。之所以名称前面用 DCD，表示译码器。它是 decoder 的缩写。
2. 设置规范的编译结果文件路径。

注意：这一步不要忽略。否则编译结果文件不会集中在一个专门的文件目录。

3. 建立 VHDL 语言的源代码文件。

源代码如下：

```
--2008-08-15 源代码源自[3]  --
-- Tested OK --

LIBRARY ieee;                                   --打开IEEE库
USE ieee.std_logic_1164.ALL;                    --打开IEEE库中STD_LOGIC_1164

ENTITY DCD74LS138 IS
PORT( e: IN std_logic_vector(3 DOWNTO 1);       --相当于逻辑图中的G1/G2/G3
      a: IN std_logic_vector(2 DOWNTO 0);       --相当于逻辑图中的A/B/C
      o: OUT std_logic_vector(7 DOWNTO 0) );    --相当于逻辑图中的Y7~Y0
END DCD74LS138;

ARCHITECTURE first OF DCD74LS138 IS             --定义结构体
SIGNAL ea: std_logic_vector(5 DOWNTO 0);        --定义中间信号ea
```

```
BEGIN                                                    --开始电路功能描述
    ea<= e & a;
    o<="11111110"WHEN ea = "100000" ELSE                 --条件赋值语句
          "11111101"WHEN ea = "100001" ELSE              --条件赋值语句
          "11111011"WHEN ea = "100010" ELSE              --条件赋值语句
          "11110111"WHEN ea = "100011" ELSE
          "11101111"WHEN ea = "100100" ELSE
          "11011111"WHEN ea = "100101" ELSE
          "10111111"WHEN ea = "100110" ELSE
          "01111111"WHEN ea = "100111" ELSE
          "11111111";
END first;                                               --结构体结束语句
```

4. 分析与解析。
5. 分析与综合。
6. 建立波形文件，添加结点，确定时钟（clock）信号之类的时钟周期，并且为其他结点添加合适的信号波形。

	Node Name	Direction	Location	I/O Bank	Vref Group	I/O Standard
1	a[2]	Input	PIN_AB25	6	B6_N2	3.3-V LVTTL (default)
2	a[1]	Input	PIN_AB26	6	B6_N2	3.3-V LVTTL (default)
3	a[0]	Input	PIN_AA23	6	B6_N2	3.3-V LVTTL (default)
4	e[3]	Input	PIN_AC24	6	B6_N3	3.3-V LVTTL (default)
5	e[2]	Input	PIN_AC26	6	B6_N3	3.3-V LVTTL (default)
6	e[1]	Input	PIN_AC27	6	B6_N2	3.3-V LVTTL (default)
7	o[7]	Output	PIN_AJ2	8	B8_N3	3.3-V LVTTL (default)
8	o[6]	Output	PIN_AJ3	8	B8_N3	3.3-V LVTTL (default)
9	o[5]	Output	PIN_AH4	8	B8_N3	3.3-V LVTTL (default)
10	o[4]	Output	PIN_AK3	8	B8_N3	3.3-V LVTTL (default)
11	o[3]	Output	PIN_AJ4	8	B8_N3	3.3-V LVTTL (default)
12	o[2]	Output	PIN_AJ5	8	B8_N3	3.3-V LVTTL (default)
13	o[1]	Output	PIN_AK5	8	B8_N3	3.3-V LVTTL (default)
14	o[0]	Output	PIN_AJ6	8	B8_N2	3.3-V LVTTL (default)
15	<<new node>>					

图 12-9　DE2-70 平台的 74LS138 译码器引脚分配

7. 电路功能仿真。
8. 生成功能仿真网表。
9. 分配引脚。参看图 12-9。
10. 全编译。
11. 时序仿真。
12. 生成映像文件，下载到电路板，测试。

◆本实验指导结束

12.4　两个 4 位数比较器实验

1. 实验目的：利用 VHDL 的进程和顺序语句中的条件语句实现两个 4 位二进制数据 a 和 b 的比较，如果 a 大于等于 b 输出高电平，否则输出低电平。

本实验程序的 a 变量通过定义在 SW[3:0] 的 4 个开关输入，b 变量通过定义在 SW[7:4]的 4 个开关输入。判断结果通过 LEDR[0]给出。当 a>=b 时，LEDR[0]发亮，否则不亮。

2. 建立工程文件，命名为 Bits_4_Comparator。

3. 设置规范的编译结果文件路径。

注意：这一步不要忽略。否则编译结果文件不会集中在一个专门的文件目录。

4. 建立 VHDL 语言的源代码文件，源代码如下列出。

```
--  2008-08-15  源代码源自[3]  --
--  4 bits Comparator  --
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY Bits_4_Comparator IS
PORT( a,b: IN std_logic_vector(3 DOWNTO 0);
        y: OUT std_logic
  );
END Bits_4_Comparator;

ARCHITECTURE first OF Bits_4_Comparator IS
BEGIN
PROCESS (a,b)
BEGIN
IF (b<=a) THEN
  Y <='0';
ELSE
  Y <='1';
END IF;

END PROCESS;
END first;
```

5. 分析与解析。
6. 分析与综合。
7. 建立波形文件，添加结点。对两个手动输入开关信号，可以加时钟（clock）信号进行仿真。注意：时钟周期要设置为秒级范围。
8. 电路功能仿真。
9. 生成功能仿真网表。

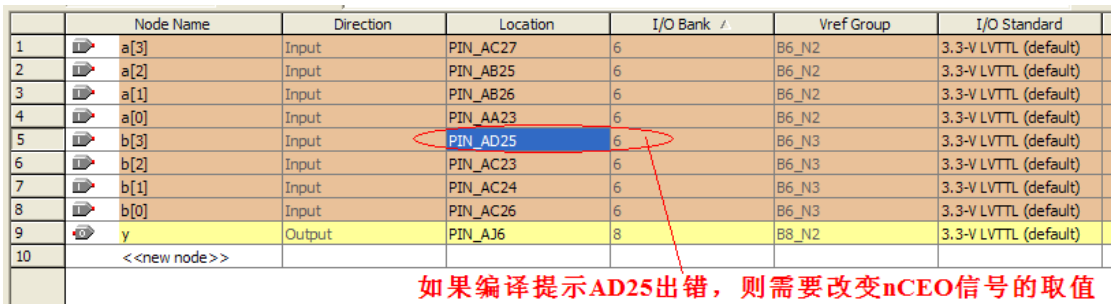


图 12-10　4 位比较器的引脚配置

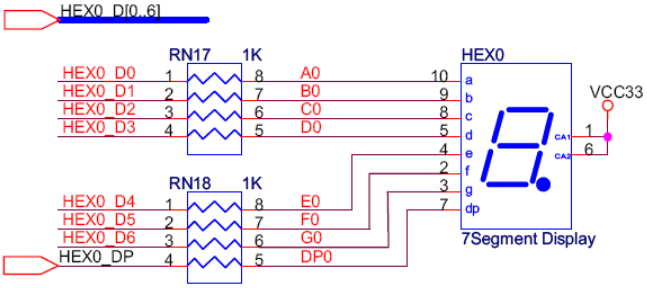
10. 分配引脚。参看图 12-10。
11. 全编译。
12. 时序仿真。
13. 生成映像文件，下载到电路板，测试。

◆本实验指导结束

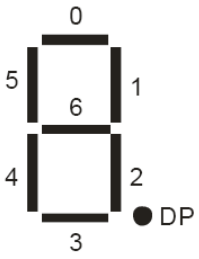
12.5　七段 LED 数码管显示实验

1. 实验目的：DE2-70 实验开发板上的 8 个 7 段 LED 数码管是重要的输出器件。本实验给一个示例，如何在一个最简单的单独 7 段 LED 数码管上显示十六进制数。

DE2-70 教学实验板共有 8 个数码管，编号分别是 HEX0~HEX7。下面的图 12-11 给出了单个 LED 数码管的电原理图和段编号。



Schematic diagram of the 7-segment displays.



Position and index of each segment in a 7-segment display.

图 12-11　DE2-70 的七段 LED 数码管电原理图和段编号图

实验具体要求是：做到 8 个数码管中只有最右边的显示一位十六进制数，其显示值反映了 SW[3:0]的开关位置。

2. 建立工程文件，命名为 SevenSegLED。
3. 设置规范的编译结果文件路径。

注意：这一步不要忽略。否则编译结果文件不会集中在一个专门的文件目录。

4. 建立 VHDL 语言的源代码文件，源代码如下列出。

```
-- DE2-70实验板上LED7段数码管显示十六进制数的实验
-- VHDL代码，源自参考图书[3]第89页，  2009-08-16
--  要求：8个数码管中只有最右边的显示一位十六进制数，其值等于SW[3:0]的开关位置。
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY   SevenSegLED   IS
PORT( bcd: IN std_logic_vector(3 DOWNTO 0);          -- 输入端口
      HEX0: OUT std_logic_vector(6 DOWNTO 0);        -- 以下端口均为输出端口
      HEX1: OUT std_logic_vector(6 DOWNTO 0);
      HEX2: OUT std_logic_vector(6 DOWNTO 0);
      HEX3: OUT std_logic_vector(6 DOWNTO 0);
      HEX4: OUT std_logic_vector(6 DOWNTO 0);
      HEX5: OUT std_logic_vector(6 DOWNTO 0);
      HEX6: OUT std_logic_vector(6 DOWNTO 0);
      HEX7: OUT std_logic_vector(6 DOWNTO 0);
      HEX0_DP: OUT std_logic;
      HEX1_DP: OUT std_logic;
      HEX2_DP: OUT std_logic;
```

```
    HEX3_DP: OUT std_logic;
    HEX4_DP: OUT std_logic;
    HEX5_DP: OUT std_logic;
    HEX6_DP: OUT std_logic;
    HEX7_DP: OUT std_logic
    );
END SevenSegLED;

ARCHITECTURE first OF SevenSegLED IS
BEGIN
            HEX1 <= "1111111";   -- 包括本语句的以下取值为'1'的LED数码管不显示
            HEX2 <= "1111111";
            HEX3 <= "1111111";
            HEX4 <= "1111111";
            HEX5 <= "1111111";
            HEX6 <= "1111111";
            HEX7 <= "1111111";
            HEX0_DP<= '1';
            HEX1_DP<= '1';
            HEX2_DP<= '1';
            HEX3_DP<= '1';
            HEX4_DP<= '1';
            HEX5_DP<= '1';
            HEX6_DP<= '1';
            HEX7_DP<= '1';
            WITH bcd SELECT     -- 共阳极七段LED数码管的段码表
            HEX0 <= "1000000" WHEN "0000",      -- 0  取值等于SW[3:0]
                    "1111001" WHEN "0001",      -- 1
                    "0100100" WHEN "0010",      -- 2
                    "0110000" WHEN "0011",      -- 3
                    "0011001" WHEN "0100",      -- 4
                    "0010010" WHEN "0101",      -- 5
                    "0000010" WHEN "0110",      -- 6
                    "1111000" WHEN "0111",      -- 7
                    "0000000" WHEN "1000",      -- 8
                    "0011000" WHEN "1001",      -- 9
                    "0001000" WHEN "1010",      -- A
                    "0000011" WHEN "1011",      -- B
                    "1000110" WHEN "1100",      -- C
                    "0100001" WHEN "1101",      -- d
                    "0000110" WHEN "1110",      -- E
                    "0001110" WHEN "1111",      -- F
                    "1111111" WHEN OTHERS;        -- 其余不显示
```

```
END first;          -- 结构体结束语句
```

5. 分析与解析。
6. 分析与综合。
7. 建立波形文件，添加结点。对两个手动输入开关信号，可以加时钟（clock）信号进行仿真。注意：时钟周期要设置为秒级范围。
8. 电路功能仿真。
9. 生成功能仿真网表。
10. 分配引脚。参看表 12-1。

初学者注意：

⑴这个引脚分配表主要涉及 DE2-70 的 8 个 LED 数码管的 64 个引脚编号。对于类似的实验项目具有参考价值。⑵这段代码可以被直接引用，引用时把这段代码直接追加到 Quartue II 工程的.qsf 文件中去即可。

表 12-1　七段 LED 数码管和 SW[3:0]的引脚分配

```
set_location_assignment PIN_AC27 -to bcd[3]
set_location_assignment PIN_AB25 -to bcd[2]
set_location_assignment PIN_AB26 -to bcd[1]
set_location_assignment PIN_AA23 -to bcd[0]

set_location_assignment PIN_AD12 -to HEX0[6]
set_location_assignment PIN_AD11 -to HEX0[5]
set_location_assignment PIN_AF10 -to HEX0[4]
set_location_assignment PIN_AD10 -to HEX0[3]
set_location_assignment PIN_AH9 -to HEX0[2]
set_location_assignment PIN_AF9 -to HEX0[1]
set_location_assignment PIN_AE8 -to HEX0[0]
set_location_assignment PIN_AF12 -to HEX0_DP
set_location_assignment PIN_AD17 -to HEX1[6]
set_location_assignment PIN_AF17 -to HEX1[5]
set_location_assignment PIN_AE17 -to HEX1[4]
set_location_assignment PIN_AG16 -to HEX1[3]
set_location_assignment PIN_AF16 -to HEX1[2]
set_location_assignment PIN_AE16 -to HEX1[1]
set_location_assignment PIN_AG13 -to HEX1[0]
set_location_assignment PIN_AC17 -to HEX1_DP
set_location_assignment PIN_AE19 -to HEX2[6]
set_location_assignment PIN_AB19 -to HEX2[5]
set_location_assignment PIN_AB18 -to HEX2[4]
set_location_assignment PIN_AG4 -to HEX2[3]
set_location_assignment PIN_AH5 -to HEX2[2]
set_location_assignment PIN_AF7 -to HEX2[1]
set_location_assignment PIN_AE7 -to HEX2[0]
set_location_assignment PIN_AC19 -to HEX2_DP
```

```
set_location_assignment PIN_M6 -to HEX3[6]
set_location_assignment PIN_M7 -to HEX3[5]
set_location_assignment PIN_M8 -to HEX3[4]
set_location_assignment PIN_N7 -to HEX3[3]
set_location_assignment PIN_N10 -to HEX3[2]
set_location_assignment PIN_P4 -to HEX3[1]
set_location_assignment PIN_P6 -to HEX3[0]
set_location_assignment PIN_M4 -to HEX3_DP
set_location_assignment PIN_M2 -to HEX4[6]
set_location_assignment PIN_M1 -to HEX4[5]
set_location_assignment PIN_N3 -to HEX4[4]
set_location_assignment PIN_N2 -to HEX4[3]
set_location_assignment PIN_P3 -to HEX4[2]
set_location_assignment PIN_P2 -to HEX4[1]
set_location_assignment PIN_P1 -to HEX4[0]
set_location_assignment PIN_L6 -to HEX4_DP
set_location_assignment PIN_K5 -to HEX5[6]
set_location_assignment PIN_K4 -to HEX5[5]
set_location_assignment PIN_K1 -to HEX5[4]
set_location_assignment PIN_L3 -to HEX5[3]
set_location_assignment PIN_L2 -to HEX5[2]
set_location_assignment PIN_L1 -to HEX5[1]
set_location_assignment PIN_M3 -to HEX5[0]
set_location_assignment PIN_K6 -to HEX5_DP
set_location_assignment PIN_E4 -to HEX6[6]
set_location_assignment PIN_F4 -to HEX6[5]
set_location_assignment PIN_G4 -to HEX6[4]
set_location_assignment PIN_H8 -to HEX6[3]
set_location_assignment PIN_H7 -to HEX6[2]
set_location_assignment PIN_H4 -to HEX6[1]
set_location_assignment PIN_H6 -to HEX6[0]
set_location_assignment PIN_K2 -to HEX6_DP
set_location_assignment PIN_G1 -to HEX7[6]
set_location_assignment PIN_H3 -to HEX7[5]
set_location_assignment PIN_H2 -to HEX7[4]
set_location_assignment PIN_H1 -to HEX7[3]
set_location_assignment PIN_J2 -to HEX7[2]
set_location_assignment PIN_J1 -to HEX7[1]
set_location_assignment PIN_K3 -to HEX7[0]
set_location_assignment PIN_G2 -to HEX7_DP
```

11. 全编译。

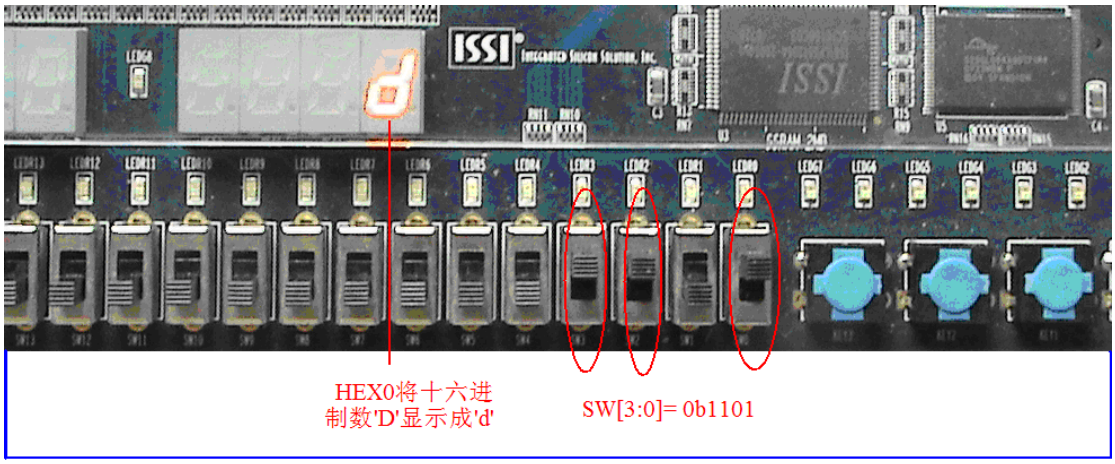


图 12-12　DE2-70 的单独 LED 数码管显示快照

12. 时序仿真。
13. 生成映像文件，下载到电路板，测试。
14. 测试结果快照。参看图 12-12。

◆本实验指导结束

12.6　时钟上沿属性实验

1.　实验原理和实验目的

实验原理和实验目的：VHDL 有 2 个描述时钟信号的属性。

‘EVENT: 如果当前的信号发生了变化则返回真，否则返回假。

‘LAST_VALUE: 返回信号变化之前的值

本实验程序验证 VHDL 的这两个描述时钟信号属性的语句。

2.　建立工程文件：clock1。

3.　设置规范的编译结果文件路径。

4.　建立 VHDL 语言的源代码文件，源代码如下列出。

```
-- Clock1 Tested 2009-08-17
--  验证 VHDL 描述时钟信号上沿属性的语句
--  本实验代码参考郑燕图书[3]第 92 页

LIBRARY ieee;                   --打开 IEEE 库
USE ieee.std_logic_1164.all;        --打开 STD_LOGIC_1164 程序包

ENTITY Clock1 IS                    --定义程序名（实体）
PORT( clk: IN std_logic;        --定义时钟输入端口
    y: OUT std_logic            --定义时钟输出端口
    );
END Clock1;                         --实体结束语句

ARCHITECTURE first OF Clock1 IS     --定义结构体
SIGNAL sig:std_logic:='0';      --定义结构体中的信号
BEGIN
```

```
PROCESS (clk)                    --过程语句，定义敏感信号
BEGIN                                --进程功能描述开始
IF (clk'EVENT AND (clk='1')AND(clk'LAST_VALUE='0')) THEN
                                 --判断时钟前沿（上升沿）是否到来
    sig<=NOT sig;                --条件成立，信号取反
END IF;

END PROCESS;
y<=sig;                              --利用信号实现进程之间通信
END first;                       --结构体结束语句
```

5. 分析与解析，而后分析与综合。

建立波形文件，添加结点。对 clk 信号进行功能仿真，参看图 12-13。对 clk 信号进行时序仿真，参看图 12-13。注意：时钟周期要设置为纳秒级范围，否则观察不出信号的延时。

6. 进行全编译。通过后保存工程文件。

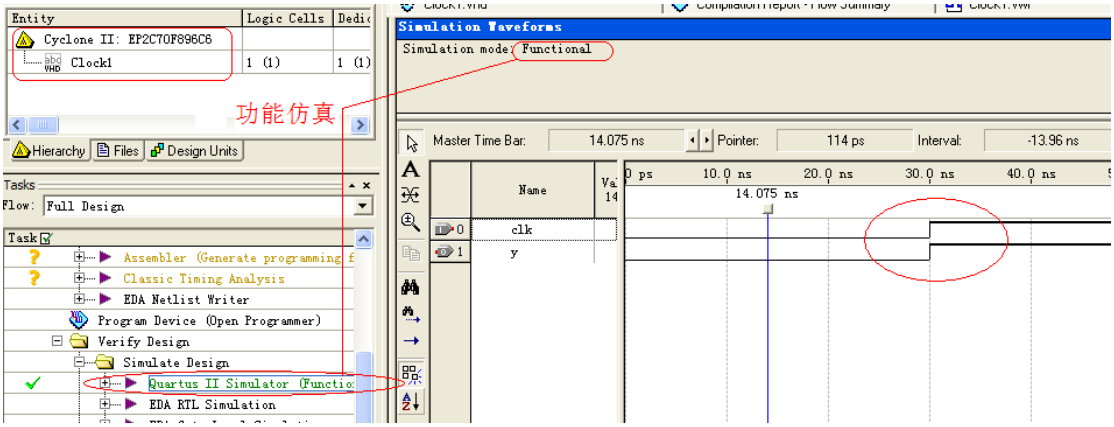


图 12-13　时钟上沿属性实验的功能仿真结果

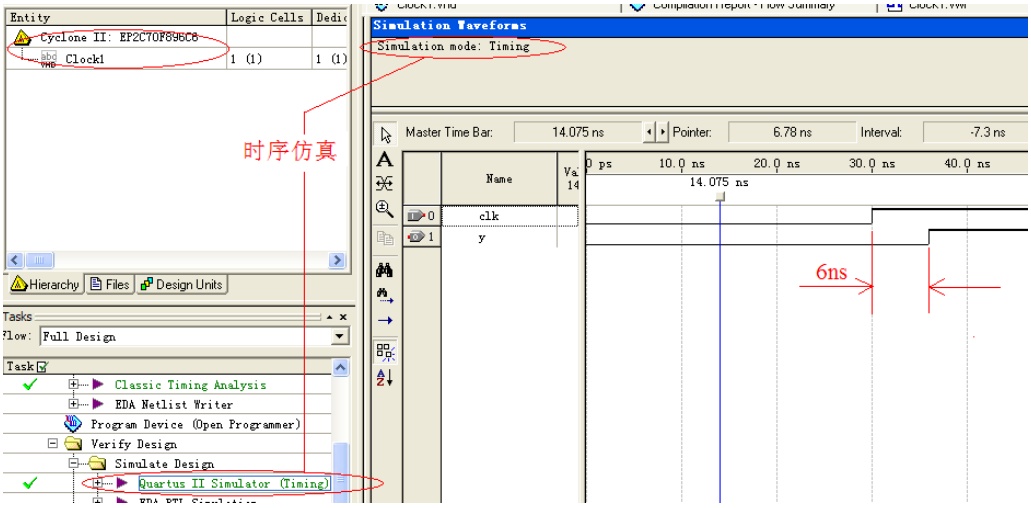


图 12-14　时钟上沿属性实验的时序仿真结果

◆本实验指导结束

12.7　简单 D 触发器

1. 实验原理和实验内容

简单 D 触发器没有复位功能和置位功能。当时钟信号前沿到来时输入信号被送到输出端，否则输出信号保持不变。

2. 实验源代码

```
-- DTrigger 2009-08-08
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY DTrigger IS
PORT( clk,data: IN std_logic;
          q: OUT std_logic
      );
END DTrigger;

ARCHITECTURE first OF DTrigger IS
BEGIN
PROCESS (clk)
BEGIN
IF rising_edge(clk) THEN
    q <=data;
END IF;
END PROCESS;
END first;
```

3. 功能仿真。如图 12-15 所示。

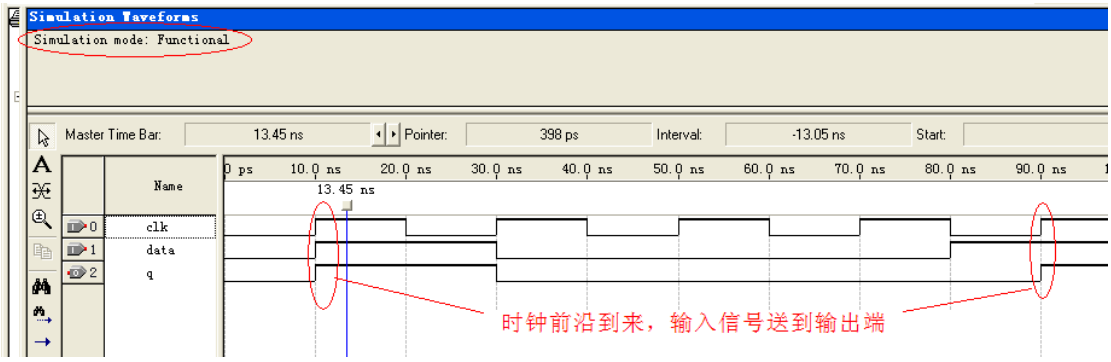


图 12-15　简单 D 触发器的功能仿真

4. 时序仿真。从仿真时序图可以观察到：当 clk 到来时，只要 data 为高电平，q 在延时 6ns 之后会输出 data 值。参看图 12-16。

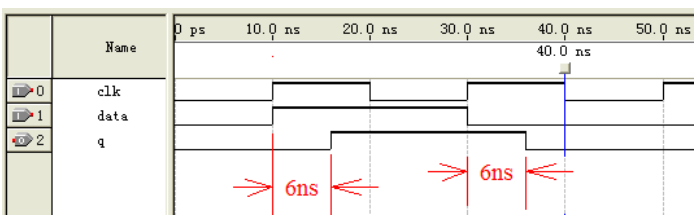


图 12-16　简单 D 触发器的时序仿真

◆本实验指导结束

12.8　具有异步复位和置位功能 D 触发器

1. 实验原理和实验内容

在简单 D 触发器上添加复位控制端和置位控制端。并且复位和置位功能实现的与时钟无关的电路就构成具有异步复位和置位功能触发器。

当复位信号 reset 有效(reset=1)，这时 D 触发器被复位(q=0)，输出低电平。

当复位信号无效(reset=0)，而置位信号有效(set=1)，这时触发器被置位(q=1)，输出高电平。

当复位信号和置位信号都无效(reset=0, set=0)，这时时钟信号前沿到来时输入信号被送到输出端。否则输出信号保持不变。

2. 实验代码 1 借鉴郑燕图书[3]第 94 页，列出如下：

```
-- 具有异步复位和置位功能的 D 触发器
--  AsynSetupDTrigger.vhd tested OK!  2009-08-18
-- 参考郑燕图书[3]第 94 页例 5.4.2

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY AsynSetupDTrigger IS
PORT( clk,data: IN std_logic;
      reset,set: IN std_logic;
      q: OUT std_logic
        );
END AsynSetupDTrigger;

ARCHITECTURE second OF AsynSetupDTrigger IS
 BEGIN
 p1: PROCESS (clk, reset, set) --三个敏感信号
  BEGIN
     IF reset='0' THEN
     q<='0';
     ELSIF set='1' THEN
     q<='1';
     ELSIF rising_edge(clk) THEN
     q<=data;
     END IF;
  END PROCESS p1;
END second;
```

3. 实验代码 1 经过编译后功能仿真和时序仿真正确。仿真画面快照略。

4. 实验代码 2 借鉴曾繁泰图书[4]第 237 页，调试通过，列出如下：

```
-- 只具有异步复位功能的 D 触发器
--  DiffLogic.vhd  2009-08-18
-- 参考曾繁泰图书[4]第 237 页

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY DiffLogic IS
   PORT( clk,data: IN std_logic;
cls: IN std_logic;
q: OUT std_logic
   );
END DiffLogic;

ARCHITECTURE second OF DiffLogic IS
 BEGIN
 p1: PROCESS (clk, cls) --三个敏感信号
        BEGIN
          IF cls='0' THEN
      q<='0';
          ELSIF rising_edge(clk) THEN
      q<=data;
             END IF;
        END PROCESS p1;
END second;
```

5. 实验代码 2 经过编译后功能仿真和时序仿真正确。仿真画面快照略。

◆本实验指导结束

12.9　具有同步复位和置位功能 D 触发器

1. 实验原理和实验内容

异步复位和置位功能 D 触发器的复位和置位与时钟无关。如果要求 D 触发器的复位和置位受时钟控制，这就构成了具有同步复位和置位功能的 D 触发器。

2. 本实验代码在上一节的实验代码基础上修改而成。代码清单如下：

```
-- 具有同步复位和置位功能的 D 触发器
-- SynSetupDTrigger.vhd tested OK!  2009-08-18
-- 参考郑燕图书[3]第 95 页例 5.4.3

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY SynSetupDTrigger IS
PORT( clk,data: IN std_logic;
```

```
    reset,set: IN std_logic;
    q: OUT std_logic
      );
END SynSetupDTrigger;

ARCHITECTURE second OF SynSetupDTrigger IS
 BEGIN
 p1: PROCESS (clk, reset, set) --三个敏感信号
  BEGIN
  IF rising_edge(clk) THEN     --判断时钟信号是否有效
    IF reset='1' THEN          --复位(reset)信号高电平有效
     q<='0';
    ELSIF set='1' THEN         --置位(set)信号高电平有效
     q<='1';
    ELSIF rising_edge(clk) THEN   --时钟上升沿到来时改变 D 触发器输出
     q<=data;
    END IF;
  END IF;                        --判断时钟信号是否有效
  END PROCESS p1;       --进程结束语句
END second;                 --结构体结束语句
```

◆本实验指导结束

12.10　VDHL 语言实现的计数器

1. 本实验代码与第 3 章的实验内容相似，但是采用了另外一种 HDL 实现，也就是 VHDL 实现。
2. VHDL 语言实验代码如下：

```
-- VHDLCounter 2009-08-18 --
-- 4 位二进制同步加法计数器的 VHDL 描述

LIBRARY ieee;                              --打开 IEEE 库
USE ieee.std_logic_1164.all;               --打开 IEEE 的 1164 程序包
USE ieee.std_logic_unsigned.all;           --打开 IEEE 的 unsigned 程序包

ENTITY VHDLCounter IS                      --定义程序名（实体）
PORT(       clk: IN std_logic;             --定义时钟输入端口
  clr,en: IN std_logic;                    --定义清零，使能信号输入端口
  q: OUT std_logic_vector(3 DOWNTO 0)      --定义输出端口
          );
END VHDLCounter;                           --实体结束语句

ARCHITECTURE first OF VHDLCounter IS       --定义结构体
```

```
SIGNAL q_t: std_logic_vector(3 DOWNTO 0);      --定义信号实现内部反馈
BEGIN                                          --开始电路描述
PROCESS (clk)                                  --进程语句，定义敏感信号
  BEGIN                                        --开始电路功能描述
        IF rising_edge( clk ) THEN             --判断时钟信号是否有效
  IF clr='1' THEN
  q_t<="0000";
  ELSIF en='1' THEN
    IF q_t="1111" THEN
      q_t<="0000";
    ELSE              --注意条件语句“ELSIF”和“ELSE”的区别使用
      q_t<=q_t+1;
    END IF;
  END IF;
          END IF;
END PROCESS;       --进程结束语句
q<=q_t;            --把最新的计数值赋给输出端口
END first;         --结构体结束语句
```

3. 仿真实验结果参看图 12-17。

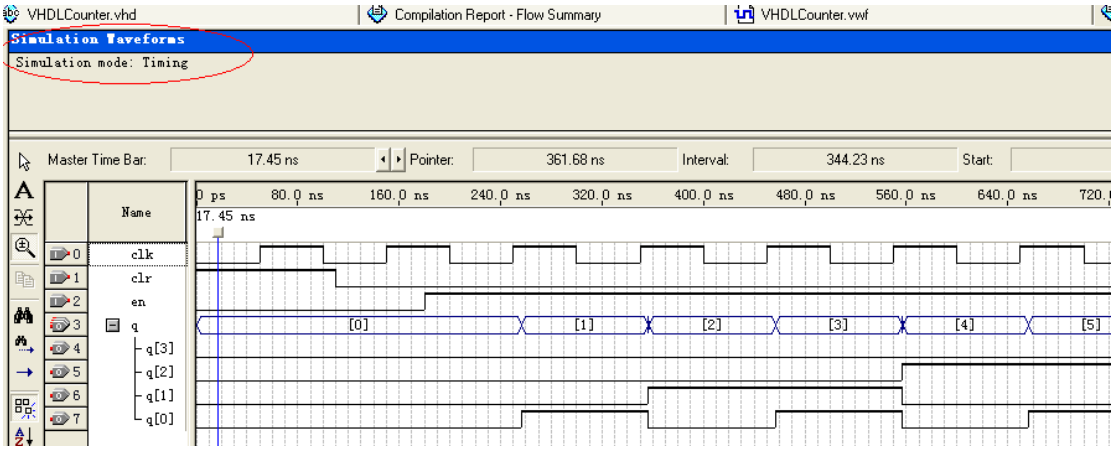


图 12-17　VHDL 计数器实验的时序仿真结果

◆本实验指导结束

12.11　简单分频器

1. 本实验代码主要让同学们掌握 CONSTANT 和 SIGNAL 语句。

CONSTANT 语句用于定义常数。SIGNAL 语句用于定义在结构体中各个进程之间传递的信息，可以出现在进程的敏感表中。

2. VHDL 语言实验代码如下：

```
-- 本程序的分频系数常量是 fpb，实现给定时钟信号的 1/fpb 分频
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY FreqDivision IS
```

```
PORT (clk_in: IN std_logic;  --定义时钟输入端口
      clk_out: OUT std_logic);  --定义时钟输出端口
END FreqDivision;

ARCHITECTURE blockone OF FreqDivision IS
    CONSTANT fpb: INTEGER:=10;       --定义 fpb 为分频常数，并规定数据类型和数值
    SIGNAL aqi: INTEGER RANGE 0 TO fpb; --定义信号 aqi，整数数据需要确定范围
BEGIN
PROCESS(clk_in)                  --进程语句，定义敏感信号
    BEGIN                        --进程功能描述开始
    IF rising_edge(clk_in) THEN--判断时钟信号是否有效
        IF aqi<fpb THEN
            aqi<=aqi+1;
            clk_out<='0';  --输出低电平
        ELSE
            aqi<=0;
            clk_out<='1';  --输出高电平，产生分频后的时钟脉冲
        END IF;
    END IF;
END PROCESS;
END blockone;
```

◆本实验指导结束

12.12　8255 芯片工作方式 0 的 VHDL 语言描述

8255 芯片是典型的微机可编程并行接口芯片。它广泛应用于各种接口电路中。本实验仅适用于 8255 的“0”型工作模式。本实验项目参考了侯伯亨编写的图书[6]第 218 页 VDHL 源代码。

1. 8255 并行接口的电路工作原理

8255A 是一种通用的可编程并行 I / O 接口芯片(Programmable Peripherial Interface，PPI)，它是为 Intel 系列微处理器设计的配套电路，也可用于其它微处理器系统以及嵌入式系统中。通过对它进行编程，芯片可工作于不同的工作方式。在微型计算机系统中，用 8255A 作接口时，通常不需要附加外部逻辑电路就可直接为 CPU 与外设之间提供数据通道，因此它得到了广泛的应用。

8255A 的内部结构如图 12-18 所示。由可图见，8255A 由以下几个部分组成：

⑴数据端口 A、B、C。（其中 C 口被分成 C 口上半部分和 C 口下半部分两个部分）

⑵A 组和 B 组控制逻辑

⑶数据总线缓冲器

⑷读 / 写控制逻辑。

8255A 内部包含 3 个 8 位的输入输出端口 A、B 和 C，通过外部的 24 根输入输出线与外设交换数据或进行通信联络。

端口 A 和端口 B 都可以用作一个 8 位的输入口或 8 位的输出口。

C 口既可以作为一个 8 位的输入口或输出口用，又可作为两个 4 位的输入输出口（C

口上半部分和 C 口下半部分）使用，还常常用来配合 A 口和 B 口工作，分别用来产生 A 口和 B 口的输出控制信号和输入 A 口和 B 口的端口状态信号。

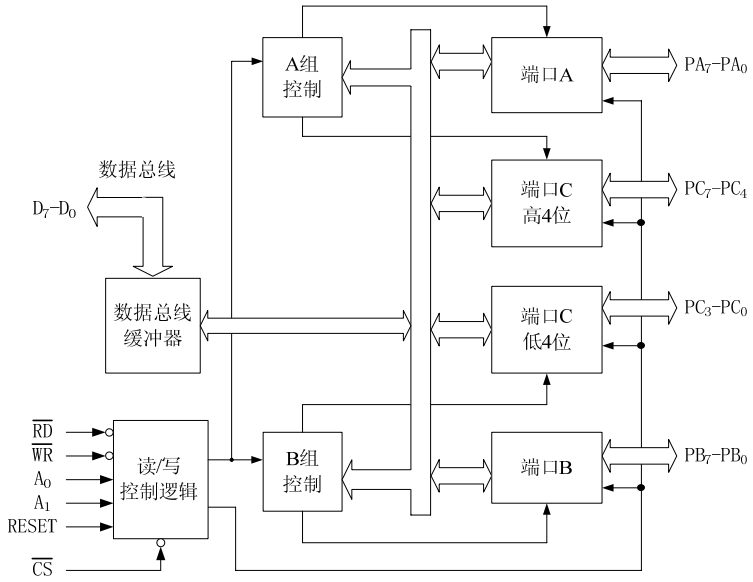


图 12-18　8255 内部电路方框图

8255A 具有 3 种基本的工作方式，它们是：方式 0－－基本输入输出方式；方式 1－－选通输入输出方式；方式 2－－双向总线 I / O 方式。

本实验完成 8255 工作方式 0 的实验。

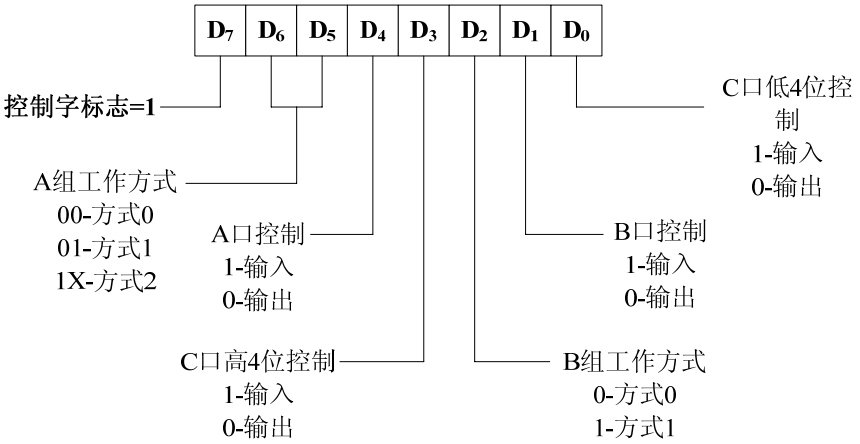


图 12-19　8255 工作方式选择控制字 ctrreg 格式

参看图 12-19。当 8255 工作在方式 0 的时候，控制字的 D7 位为‘1’，D6D5 位为“00”，D2 位为‘0’。

本实验还包括了对 C 口置 1 和置 0 的控制。图 12-20 给出了 C 口各位的置 1 置 0 控制字格式。请参看图 12-20。

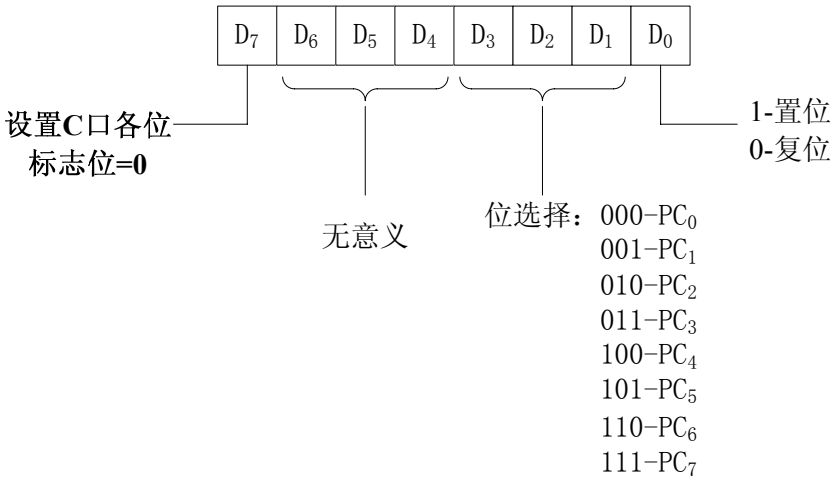


图 12-20　C 口各位的置位复位控制字

2. VHDL 的数据类型 **std_ulogic**

枚举类型 std_ulogic，它同 std_logic 类似。

std_ulogic 的定义来源于 ieee_std_1164 包，是一个有 9 个值的枚举型数据类型；std_logic 是 std_ulogic 的子类。二者的区别在于 std_logic 带有决断函数。主要是当一个信号多元驱动时，二者有所区别，而区别一般也只在功能仿真里体现，对综合语句而言，二者没什么区别。

3. 建立工程文件，命名为 **8255_Method_0**。

4. 设置规范的编译结果文件路径。

5. 建立 **VHDL** 文件。

源代码如下：

```
-- Interface8255_0 Tested at 2009-08-24
-- reference book is writen by Hou Boheng
--  本 8255 工作方式 0 的 VHDL 代码源自侯伯亨顾新编著的图书
--《VHDL 硬件描述语言与数字逻辑电路设计》第 222 页

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_ARITH.all;
USE ieee.std_logic_UNSIGNED.all;

ENTITY Interface8255_0 IS        --定义顶层结构体（实体）Interface8255_0
PORT( reset, rd, wr, cs, a0, a1: IN STD_ULOGIC;
      --  单独信号 reset(复位), rd(读), wr(写), cs(片选), a0(低位地址), a1(高位地址)
            pa:INOUT STD_ULOGIC_VECTOR(7 DOWNTO 0);    -- 8 位 A 端口，方式 0 输入输出
            pb:INOUT STD_ULOGIC_VECTOR(7 DOWNTO 0);    -- 8 位 B 端口，方式 0 输入输出
            pcl:INOUT STD_ULOGIC_VECTOR(3 DOWNTO 0);   --  低 4 位 C 端口，方式 0 输入输出
            pch:INOUT STD_ULOGIC_VECTOR(3 DOWNTO 0);   --  高 4 位 C 端口，方式 0 输入输出
            d:INOUT STD_ULOGIC_VECTOR(7 DOWNTO 0)
            );
END Interface8255_0;
```

```
ARCHITECTURE rtl OF Interface8255_0 IS                    -- 定义结构体 Interface8255_0
  SIGNAL internal_bus_out:STD_ULOGIC_VECTOR(7 DOWNTO 0);  -- 8 位内部输出总线
  SIGNAL internal_bus_in:STD_ULOGIC_VECTOR(7 DOWNTO 0);   -- 8 位内部输入总线
  SIGNAL st,ad,flag:STD_ULOGIC_VECTOR(1 DOWNTO 0);        -- 两位信号 st, ad, flag
  SIGNAL ctrreg:STD_ULOGIC_VECTOR(7 DOWNTO 0);            -- 8 位控制信号 ctrreg
  SIGNAL pa_latch,pb_latch,pc_latch:STD_ULOGIC_VECTOR(7 DOWNTO 0);
      -- A 口,B 口和 C 口的 8 位地址锁存信号

BEGIN

PROCESS (rd,cs)                          --过程定义语句，含敏感表
BEGIN
st<=ctrreg(3)&ctrreg(0);
      IF (cs='0' AND rd='0') THEN
          IF ( a0='0' AND a1='0' AND ctrreg(4)='1' ) THEN
              internal_bus_in<=(pa);                  -- A port input
          ELSIF ( a0='1' AND a1='0' AND ctrreg(1)='1' ) THEN
              internal_bus_in<=(pb);                  -- B port input
          ELSIF ( a0='0' AND a1='1' AND st="01" ) THEN
              internal_bus_in(3 DOWNTO 0)<=pcl(3 DOWNTO 0);   -- C low 4 bits port input
                  ELSIF ( a0='0' AND a1='1' AND st="10" ) THEN
              internal_bus_in(7 DOWNTO 4)<=pch(3 DOWNTO 0);   -- C high 4 bits port input
                  ELSIF ( a0='0' AND a1='1' AND st="11" AND ctrreg(7)='1') THEN
              internal_bus_in(3 DOWNTO 0)<=pcl(3 DOWNTO 0);   -- C 8 bits prot input
              internal_bus_in(7 DOWNTO 4)<=pch(3 DOWNTO 0);
          END IF;
      ELSE
          internal_bus_in<=("ZZZZZZZZ");
      END IF;
      d<=internal_bus_in;
END PROCESS;

PROCESS (cs,wr,reset)              --过程定义语句，含敏感表
      VARIABLE ctrregF:STD_ULOGIC;
      VARIABLE bctrreg_v:STD_ULOGIC_VECTOR(3 DOWNTO 0);
  BEGIN
      IF (cs='0' AND wr='0') THEN
          ad<=a1&a0;
          ctrregF:=d(1);
          internal_bus_out<=d;
      END IF;
      IF (reset='1') THEN
          pa_latch<="00000000";
          pb_latch<="00000000";
```

```
            pc_latch<="00000000";
            ctrreg<="10011011";
            bctrreg_v:="0000";
        ELSIF (wr'EVENT AND wr='1') THEN
            IF (ctrregF='1' AND ad="11" AND cs='0') THEN
            ctrreg<=internal_bus_out;
            ELSIF (ctrregF='1' AND ad="00" AND cs='0') THEN
            pa_latch<=internal_bus_out;
            ELSIF (ctrregF='1' AND ad="01" AND cs='0') THEN
            pb_latch<=internal_bus_out;
            ELSIF (ctrregF='1' AND ad="10" AND cs='0') THEN
            pc_latch<=internal_bus_out;
            ELSIF (ctrregF='1' AND ad="11" AND cs='0') THEN
            bctrreg_v:=internal_bus_out(3 DOWNTO 0);
            CASE bctrreg_v IS
                WHEN "0000" =>pc_latch(0)<='0';
                WHEN "0010" =>pc_latch(1)<='0';
                WHEN "0100" =>pc_latch(2)<='0';
                WHEN "0110" =>pc_latch(3)<='0';
                WHEN "1000" =>pc_latch(4)<='0';
                WHEN "1010" =>pc_latch(5)<='0';
                WHEN "1100" =>pc_latch(6)<='0';
                WHEN "1110" =>pc_latch(7)<='0';
                WHEN "0001" =>pc_latch(0)<='0';
                WHEN "0011" =>pc_latch(1)<='0';
                WHEN "0101" =>pc_latch(2)<='0';
                WHEN "0111" =>pc_latch(3)<='0';
                WHEN "1001" =>pc_latch(4)<='0';
                WHEN "1011" =>pc_latch(5)<='0';
                WHEN "1101" =>pc_latch(6)<='0';
                WHEN "1111" =>pc_latch(7)<='0';
                WHEN OTHERS =>flag<="11";
            END CASE;
            END IF;
        END IF;
END PROCESS;

PROCESS (pa_latch)              --过程定义语句，含敏感表
   BEGIN
   IF (ctrreg(4)='0') THEN
            pa<=(pa_latch);
   ELSE
            pa<="ZZZZZZZZ";
   END IF;
```

```
END PROCESS;

PROCESS (pb_latch)          --过程定义语句，含敏感表
  BEGIN
  IF (ctrreg(1)='0') THEN
          pb<=(pb_latch);
  ELSE
          pb<="ZZZZZZZZ";
  END IF;
END PROCESS;

PROCESS (pc_latch)              --过程定义语句，含敏感表
  BEGIN
  IF (ctrreg(0)='0') THEN
          pcl<=pc_latch(3 DOWNTO 0);
  ELSE
          pcl<="ZZZZ";
  END IF;
END PROCESS;

PROCESS (pc_latch)              --过程定义语句，含敏感表
  BEGIN
  IF (ctrreg(0)='0') THEN
          pch<=pc_latch(3 DOWNTO 0);
  ELSE
          pch<="ZZZZ";
  END IF;
END PROCESS;

END rtl;  --顶层结构体（实体）Interface8255_0 定义结束
```

6. 引脚分配。
7. 功能仿真和时序仿真。
8. 下载。
9. 测试。

◆ 本实验指导结束

12.13　完整的 8255 芯片 VHDL 语言描述

1. Altera 公司在 2002 年提供了一个完全的 8255 芯片的 VHDL 源代码。

Altera 公司 8255 芯片整个工程文件的 VDHL 文件清单如下：

```
1, A8255.VHD
2, A8255TB.VHD
3, A8255TOP.VHD
```

4, CNTL_LOG.VHD
5, DOUT_MUX.VHD
6, PORTAIN.VHD
7, PORTAOUT.VHD
8, PORTBIN.VHD
9, PORTBOUT.VHD
10, PORTCOUT.VHD
11, tb_A8251.vhd
12, tb_A8251_top.vhd

2. 波形文件的全部信号

图 12-21 给出了 8255 芯片波形文件的全部信号列表。读者从图 12-21 可以看到 Altera 公司提供的这个 VHDL 描述程序对 8255 芯片的信号描述是全面的。

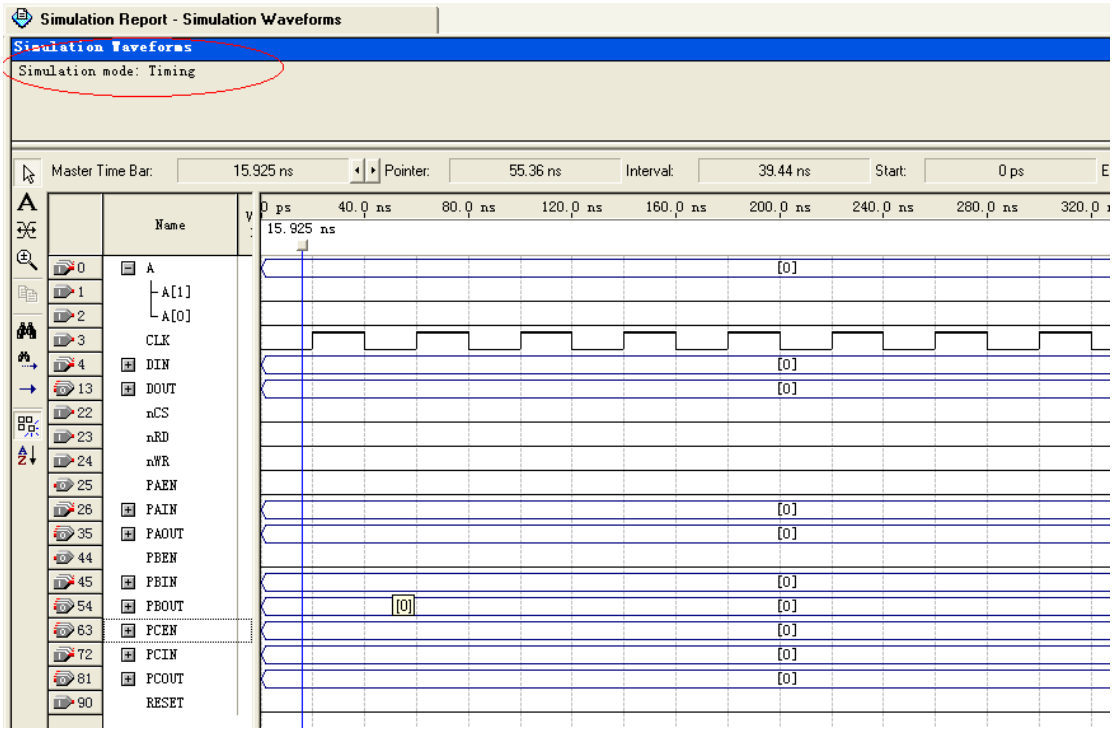


图 12-21　Altera 公司 8255 芯片的波形文件信号列表

3. 功能仿真和时序仿真。

由读者自行编写功能仿真以及时序仿真时的测试用例信号。要求做到覆盖全面和应用切合实际。

4. 下载测试

读者自行分配引脚信号，例如：A 口和 B 口输出连接 7 段 LED 数码管，C 口输出连接 LEDR 发光灯；并且自行编写映像文件 DE2-70 实验板上运行时的测试用例信号。

◆ 本实验指导结束

第 13 章　原理图输入实验项目 3 例

在 DE2-70 实验板上可以完成用原理图输入的实验项目。本章我们介绍 3 个通过原理图输入实现的典型 DE2-70 实验项目。

13.1 模 100 同步计数器

1. 实验原理和实验内容。

集成计数器在一些简单小型数字系统中被广泛应用。因为它们具有体积小、功耗低、功能灵活等优点。集成计数器的类型很多，表 13-1 列举了几种常用的集成计数器产品。可用于同步模 100 计数器的芯片是 74160。请参看表 13-1。本实验项目使用 74160 芯片两块构成一个模 100 的同步计数器。

表 13-1　常用集成计数器一览表

CP 脉冲引入方式	型号	计数模式	清零方式	预置数方式
同步	74161	4位二进制加法	异步(低电平)	同步
	74HC161	4位二进制加法	异步(低电平)	同步
	74HCT161	4位二进制加法	异步(低电平)	同步
	74LS191	单时钟4位二进制可逆	无	异步
	74LS193	双时钟4位二进制可逆	异步(高电平)	异步
	74160	十进制加法	异步(低电平)	同步
	74LS190	单时钟十进制可逆	无	异步
异步	74LS293	双时钟4位二进制加法	异步	无
	74LS290	二-五-十进制加法	异步	异步

2. 74160 集成 BCD 码计数器原理

74160 是具有清零、置数、计数和禁止计数的四功能 BCD 计数器芯片。其逻辑图和信号功能如图 13-1 所示。74160 的引脚信号列出如下：

⑴**D0~D3** 是预置数据输入端

⑵**nCLR** 是同步清零控制端

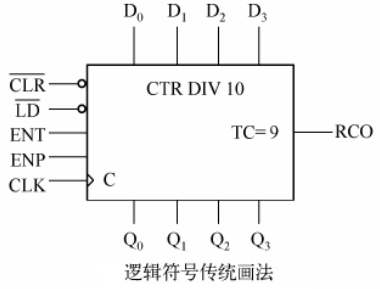
⑶**RCO** 是进位输出端

⑷**ENP** 和 **ENT** 是计数使能控制端

⑸**nLD** 同步置数控制端

⑹**CLK** 是时钟信号

⑺**Q0~Q3** 是输出信号



74160 功能表

CLK	$\overline{CLR}$	$\overline{LD}$	ENP	ENT	功　能
↑	0	×	×	×	异步清零
↑	1	0	×	×	同步置数
×	1	1	0	1	保持（包括 RCO 的状态）
×	1	1	×	0	保持（RCO=0）
↑	1	1	1	1	加计数

图 13-1　74160 芯片电原理图和功能表

74160 的工作时序如图 13-2 所示。

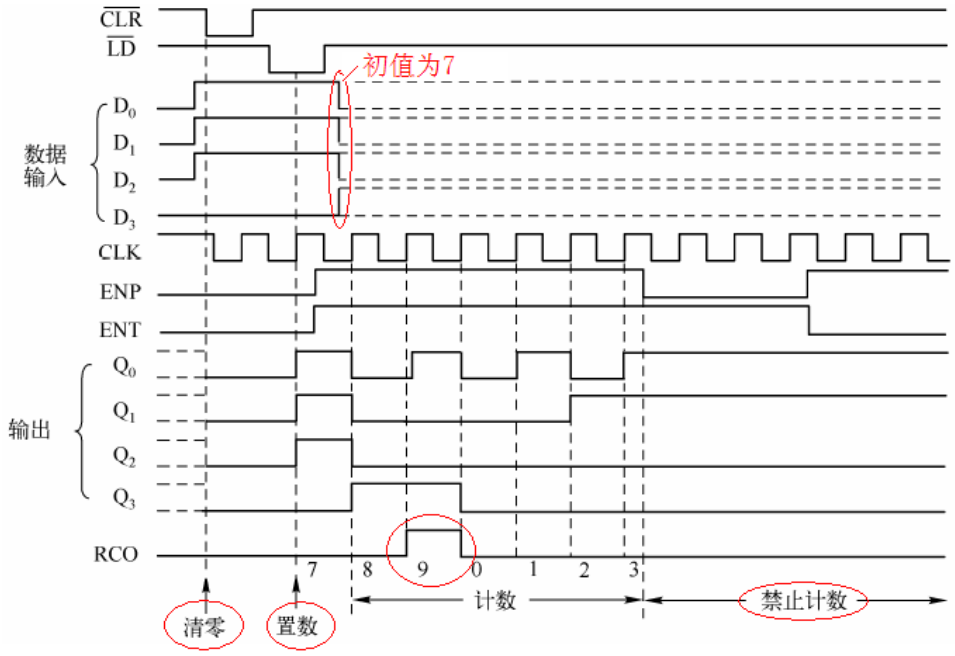
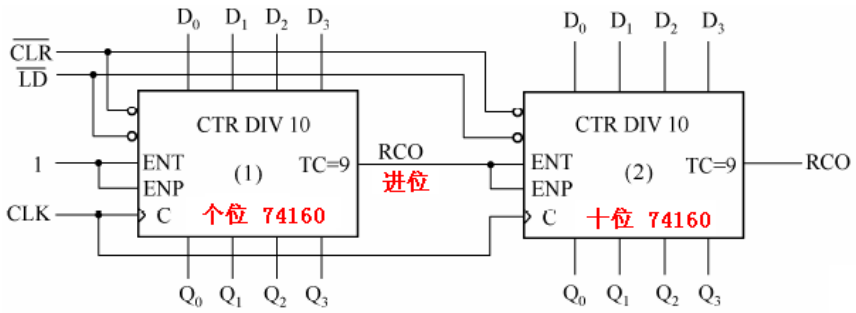


图 13-2　74160 芯片的工作时序图

3. 实验方法介绍

用两片 74160 芯片构成一个模 100 的计数器电路原理如图 13-3 所示。本实验项目参考了王振红的图书[6]。



用两片 74160 构成的模 100 计数器

图 13-3　模 100 计数器的电原理图

4. 建立工程文件，命名为 **Counter100**。
5. 设置规范的编译结果文件路径。
6. 创建一个新的电原理图文件（Block Diagram/Schematic File），其后缀为.dbf。该电原理图为两个 74160 芯片的级联。通过分析与解析处理（Analysis & Elaboration）的电路图设计（无错误，无警告，未分配 DE2-70 引脚）参看图 13-4。

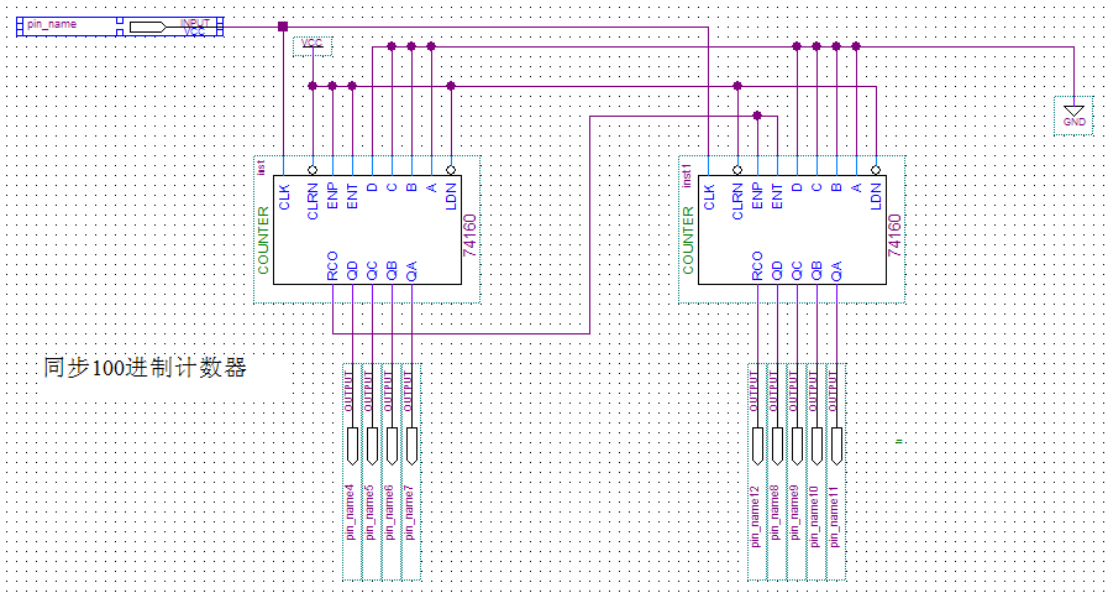


图 13-4　同步 100 计数器的电路设计（.bdf 文件视图）

7. 实验板引脚分配。参看图 13-5。注意：只有一个时钟输入信号，没有控制信号。

	Node Name	Direction	Location	I/O Bank	Vref Group	I/O Standard
1	CLK_50	Input	PIN_AD15	7	B7_N3	3.3-V LVTTL (default)
2	LEDR[17]	Output	PIN_AJ7	8	B8_N2	3.3-V LVTTL (default)
3	LEDR0[3]	Output	PIN_AJ4	8	B8_N3	3.3-V LVTTL (default)
4	LEDR0[2]	Output	PIN_AJ5	8	B8_N3	3.3-V LVTTL (default)
5	LEDR0[1]	Output	PIN_AK5	8	B8_N3	3.3-V LVTTL (default)
6	LEDR0[0]	Output	PIN_AJ6	8	B8_N2	3.3-V LVTTL (default)
7	LEDR1[3]	Output	PIN_AJ2	8	B8_N3	3.3-V LVTTL (default)
8	LEDR1[2]	Output	PIN_AJ3	8	B8_N3	3.3-V LVTTL (default)
9	LEDR1[1]	Output	PIN_AH4	8	B8_N3	3.3-V LVTTL (default)
10	LEDR1[0]	Output	PIN_AK3	8	B8_N3	3.3-V LVTTL (default)
11	<<new node...					

图 13-5　同步 100 计数器的 DE2-70 实验板引脚分配

8. 第 1 次定义模 100 计数器引脚。

参看下面的表格。

注意：图左边的 74160 是个位计数器，右边的是十位计数器。

表 12-1　同步 100 计数器初次引脚分配

信号	引脚	信号	引脚
Clk_50	Pin_AD15	RCO	LEDR[17]
左边器件 QD	LEDR0[0]	右边器件 QD	LEDR1[0]
左边器件 QC	LEDR0[1]	右边器件 QC	LEDR1[1]
左边器件 QB	LEDR0[2]	右边器件 QB	LEDR1[2]
左边器件 QA	LEDR0[3]	右边器件 QA	LEDR1[3]

9. 建立波形文件。注意：VCC 和 GND 可以不分配引脚。
10. 功能仿真和时序仿真。参看图 13-6。

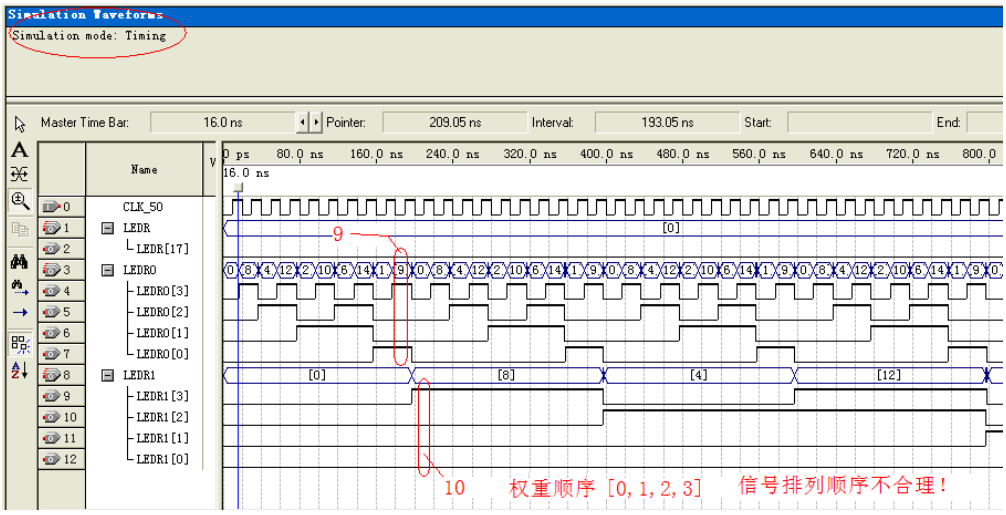


图 13-6　模 100 计数器时序仿真

11. 重新定义模 100 计数器引脚（改变 74160 芯片 BCD 码引脚的权重次序）。

表 12-2　同步 100 计数器第 2 次引脚分配

信号	引脚	信号	引脚
Clk_50	Pin_AD15	RCO	LEDR[17]
左边器件 QD	LEDR0[3]	右边器件 QD	LEDR1[3]
左边器件 QC	LEDR0[2]	右边器件 QC	LEDR1[2]
左边器件 QB	LEDR0[1]	右边器件 QB	LEDR1[1]
左边器件 QA	LEDR0[0]	右边器件 QA	LEDR1[0]

12. 再次进行仿真。参看图 13-7。可见本次两个 74160 器件的 BCD 码的权重设定是正确。

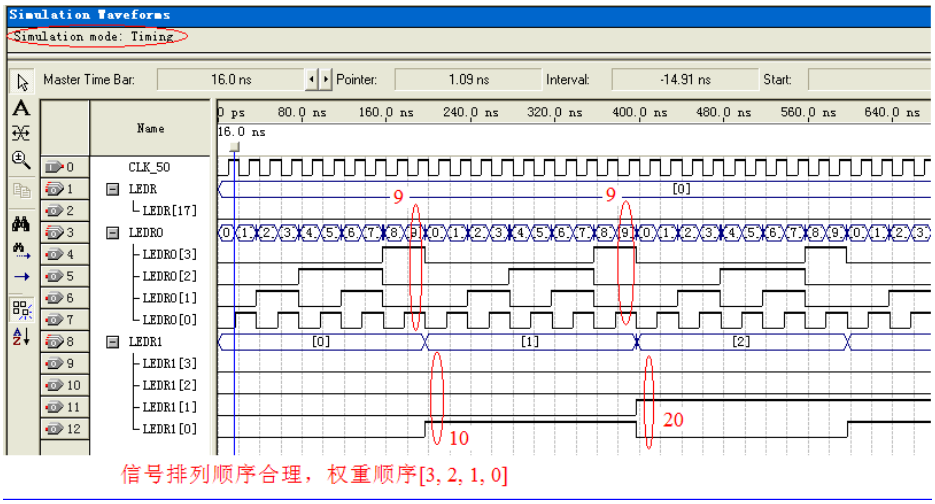


图 13-7　第 2 次模 100 计数器时序仿真

13. 增加一个控制 ENP 的输入引脚。参看图 13-8。功能仿真和时序仿真结果不太成功。

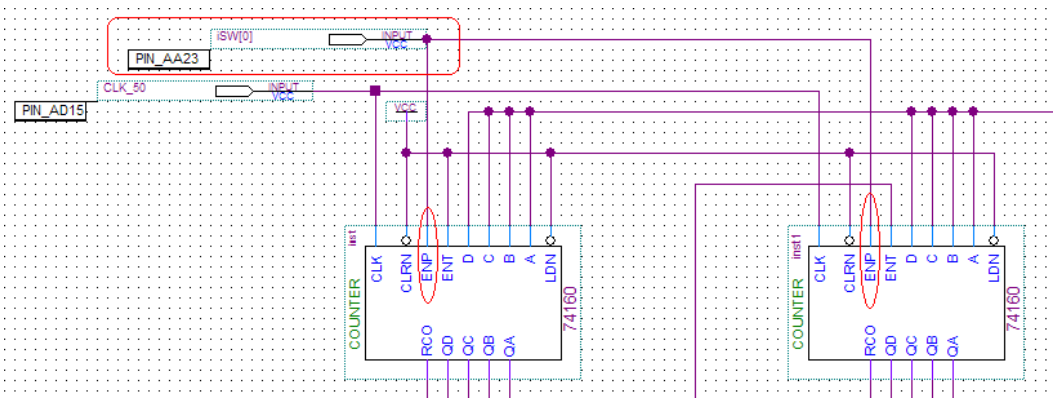


图 13-8　在模 100 计数器的电原理图上增加一个控制信号

14. 替换练习

在模 100 同步计数器的电原理图上再添加一个 74374 芯片，成为模 1000 计数器。

解答：

⑴电路设计

由三块 74374 芯片构成的模 1000 计数器电路设计图如下给出。参看图 13-9。

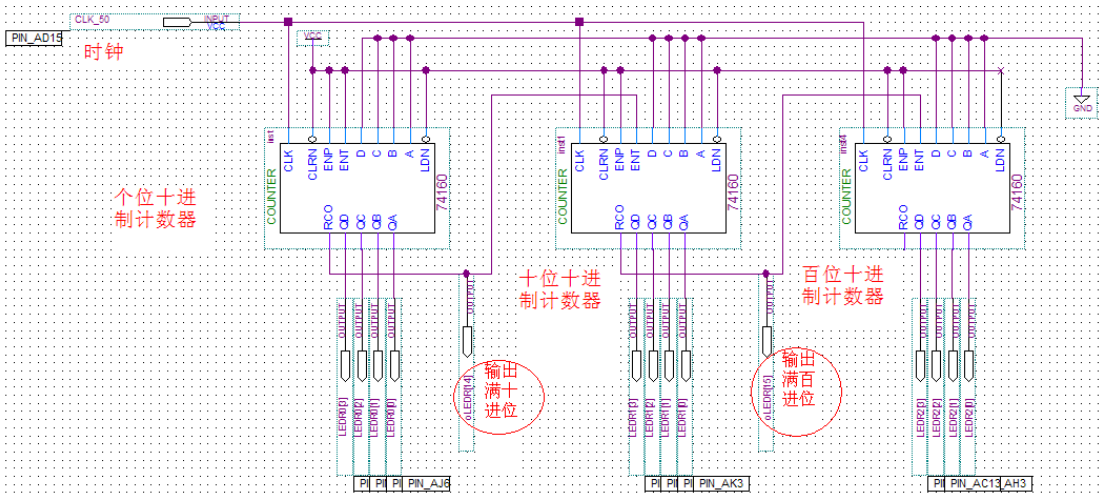


图 13-9　模 1000 同步十进制计数器电路设计图

⑵引脚分配

由于同步时钟频率高，计数值显示太快而观察不到。所以除了时钟之外，实际的引脚分配是将 3 块 74374 的输出值引脚分配到红色发光管 LEDR[11..0]上。

	Node Name	Direction	Location	I/O Bank	Vref Group	I/O Standard
1	CLK_50	Input	PIN_AD15	7	B7_N3	3.3-V LVTTL (default)
2	LEDR0[3]	Output	PIN_AJ4	8	B8_N3	3.3-V LVTTL (default)
3	LEDR0[2]	Output	PIN_AJ5	8	B8_N3	3.3-V LVTTL (default)
4	LEDR0[1]	Output	PIN_AK5	8	B8_N3	3.3-V LVTTL (default)
5	LEDR0[0]	Output	PIN_AJ6	8	B8_N2	3.3-V LVTTL (default)
6	LEDR1[3]	Output	PIN_AJ2	8	B8_N3	3.3-V LVTTL (default)
7	LEDR1[2]	Output	PIN_AJ3	8	B8_N3	3.3-V LVTTL (default)
8	LEDR1[1]	Output	PIN_AH4	8	B8_N3	3.3-V LVTTL (default)
9	LEDR1[0]	Output	PIN_AK3	8	B8_N3	3.3-V LVTTL (default)
10	LEDR2[3]	Output	PIN_AB13	8	B8_N0	3.3-V LVTTL (default)
11	LEDR2[2]	Output	PIN_AC13	8	B8_N1	3.3-V LVTTL (default)
12	LEDR2[1]	Output	PIN_AD14	8	B8_N0	3.3-V LVTTL (default)
13	LEDR2[0]	Output	PIN_AH3	8	B8_N3	3.3-V LVTTL (default)
14	oLEDR[15]	Output	PIN_AD9	8	B8_N2	3.3-V LVTTL (default)
15	oLEDR[14]	Output	PIN_AC11	8	B8_N2	3.3-V LVTTL (default)
16	<<new node>>					

图 13-10　模 1000 同步计数器的引脚分配。

图 13-10 中的 oLEDR[15..14]是进位引脚，分别表示百位进位和十位进位。如图 13-10 所示。

⑶引脚分配

对于模 1000 十进制计数器的仿真运行，如果仿真时间设置的较短，则不能够观察到模 1000 计数器计数到 999 时的进位波形。为此设定仿真时间为 8us 或者更长。设置方法：主菜单->Edit->End Time。参看图 13-11。

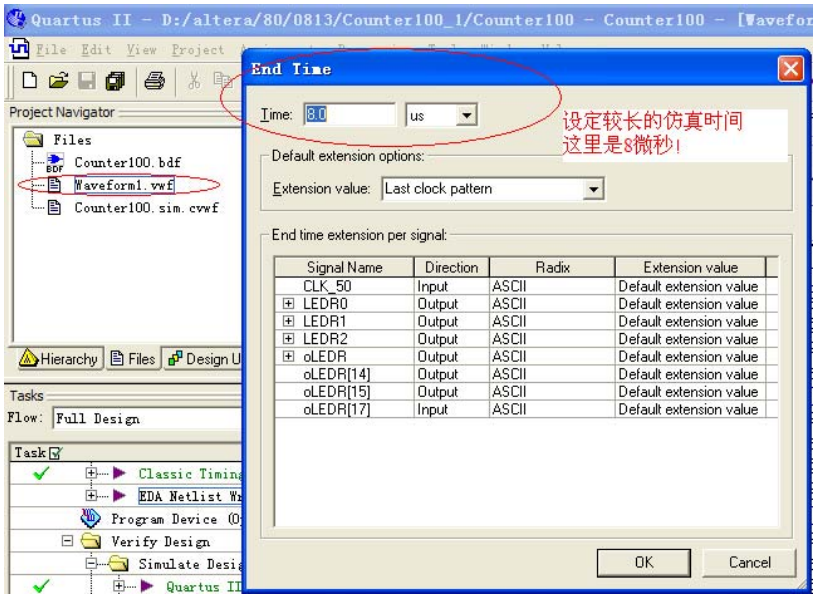


图 13-11　从主菜单 Edit 栏的 End Time 入口设置仿真时间长度

此外，在主菜单 Assignment 栏的 Setting 入口还要设置仿真的时间长度。参看图 13-12。

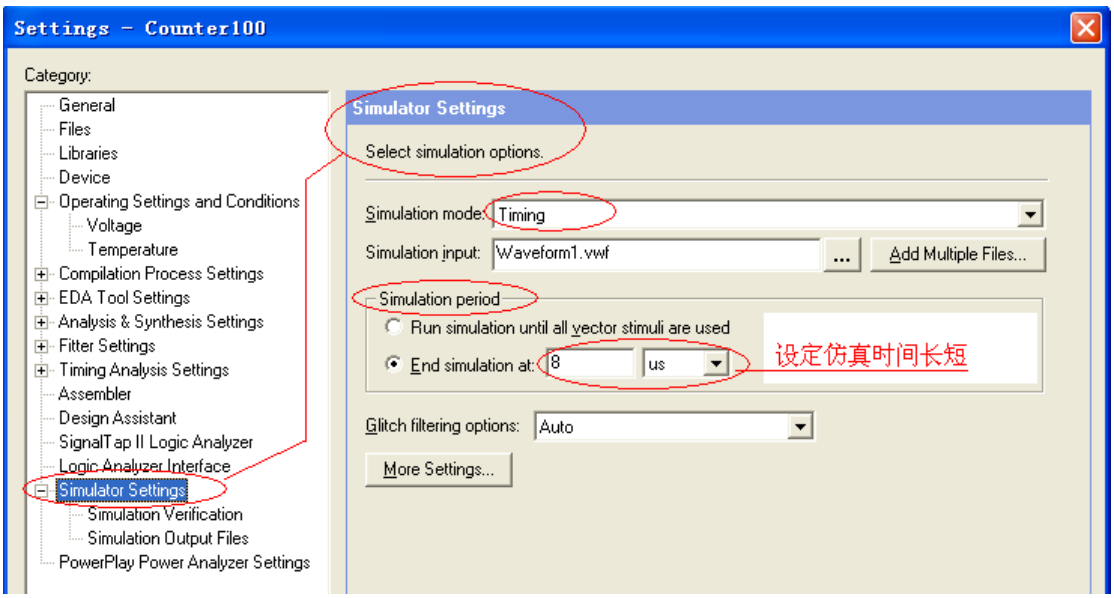


图 13-12　从主菜单 Assignment 栏的 Settings 入口设置仿真时间长度

⑷仿真结果

图 13-13 给出了模 1000 计数器的时序仿真输出结果。注意：核实计数信号脉冲的正确性。

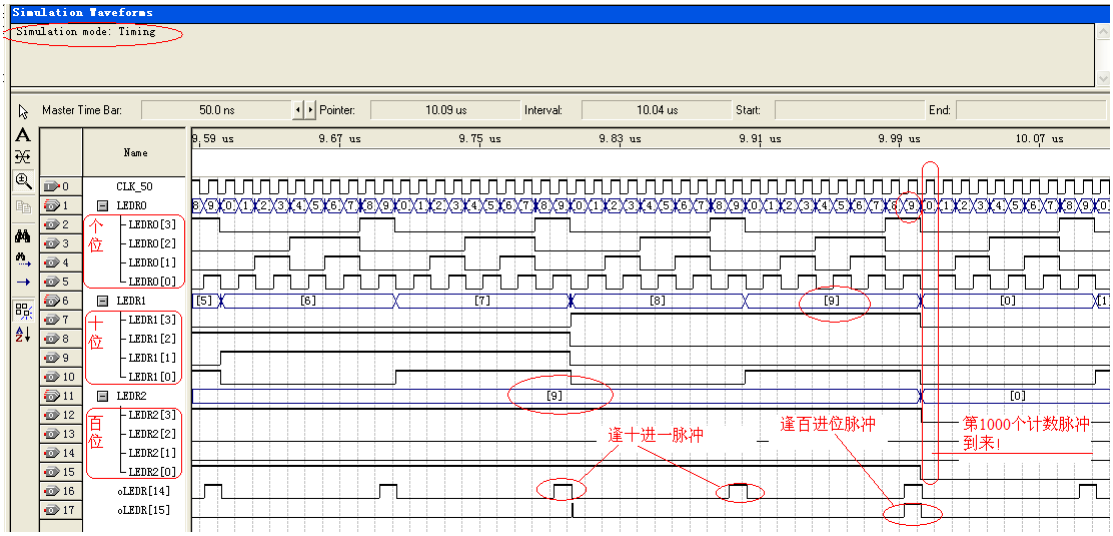


图 13-13　模 1000 计数器的时序仿真结果

◆　本实验指导结束

13.2　锁存器 74373

1.　8 位锁存器 74LS373 工作原理

74LS373 是带有三态门的八位 D 锁存器，当使能信号线 OE 为低电平时，三态门处于导通状态，允许 1Q-8Q 输出到 OUT1-OUT8，当 OE 端为高电平时，输出三态门断开，输出线 OUT1-OUT8 处于浮空状态。图 13-14 给出了 74LS373 的引脚信号和电原理图。

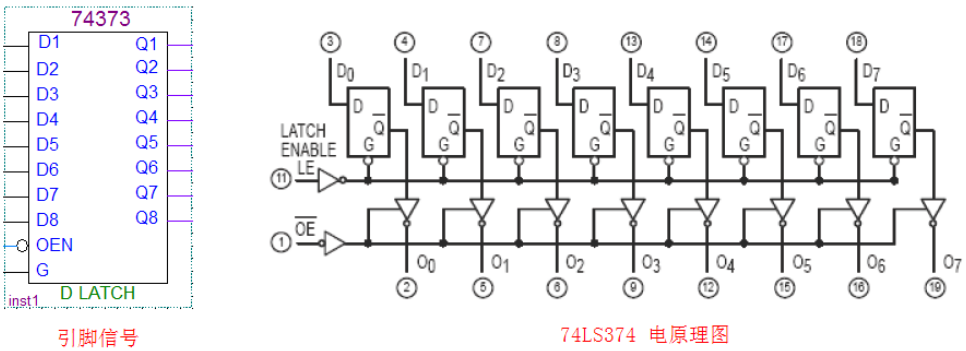


图 13-14　74LS374 芯片的引脚信号和电原理图

G 称为数据打入线，当 74LS373 用作地址锁存器时，首先应使三态门的使能信号 OE 为低电平，这时，当 G 端输入端为高电平时，锁存器输出（1Q-8Q）状态和输入端（1D-8D）状态相同；当 G 端从高电平返回到低电平（下降沿）时，输入端（1D-8D）的数据锁入 1Q-8Q 的八位锁存器中。

2.　实验原理和实验内容

用一片 74LS374（也就是 74374）芯片构成一个 8 位 D 锁存器。本实验参考了周润景图书[5]。

3.　建立工程文件，命名为 **Latch74374**。

4.　设置规范的编译结果文件路径。

创建一个新的电原理图文件（Block Diagram/Schematic File），其后缀为.dbf。参看图 13-15。

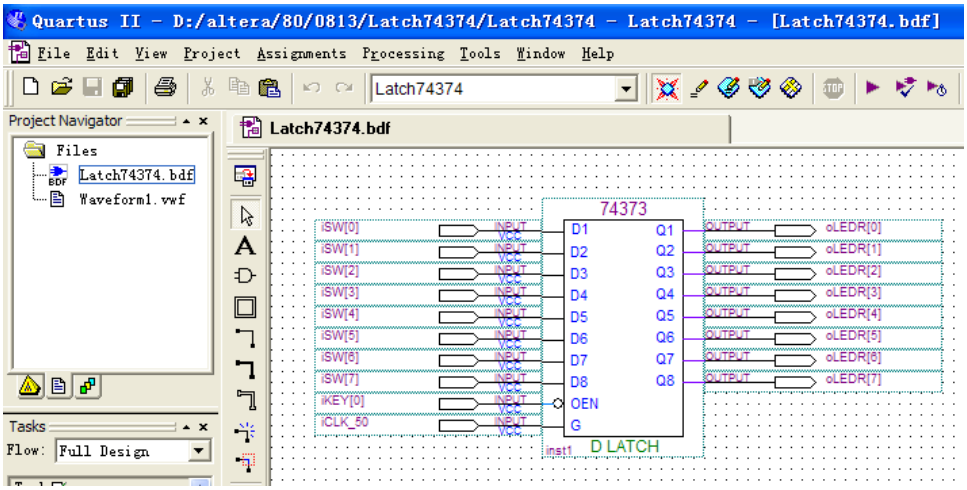


图 13-15　74374 电原理图快照

5. 建立波形文件，进行功能仿真和时序仿真。仿真结果正确。

◆ 本实验指导结束

13.3 四位串入串出移位寄存器

1. 实验原理和实验内容

采用电原理图编辑法，利用四个 D 触发器构成 4 位串入串出移位寄存器。学会系列 D 触发器级联时的简单移位功能。本实验参考了周润景等人编著的图书[5]第 188 页。

2. 建立工程文件，命名为 **ShiftInOut**。

3. 设置规范的编译结果文件路径。

创建一个新的电原理图文件（Block Diagram/Schematic File），其后缀为.dbf。该电原理图的最初电路图形设计如图 13-16 所示。注意：正确的输出信号应该是 DATAOut。图中的输出信号 DTATOut 是错误输入，但是不影响仿真处理。

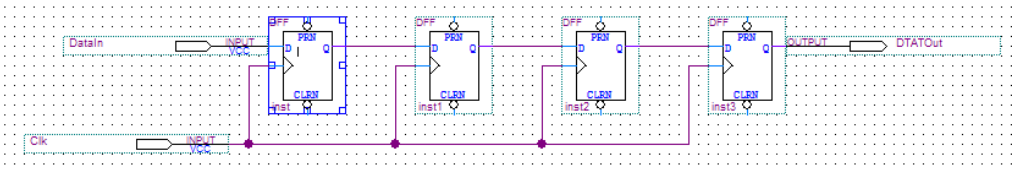


图 13-16　4 位串入串出移位寄存器电原理图

4. 建立波形文件和网表文件。

5. 进行第一次功能仿真和时序仿真。

参看图 13-17，该图给出了时序仿真结果。

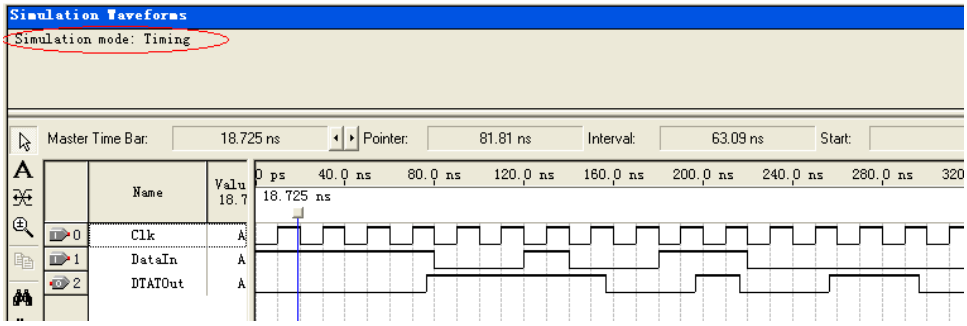


图 13-17　4 位串入串出移位寄存器

为了弄清楚 DATAOut 信号的准确时间延迟，在电路图上添加三个触发器的输出引脚 q1, q2, q3 信号。参看图 13-18。

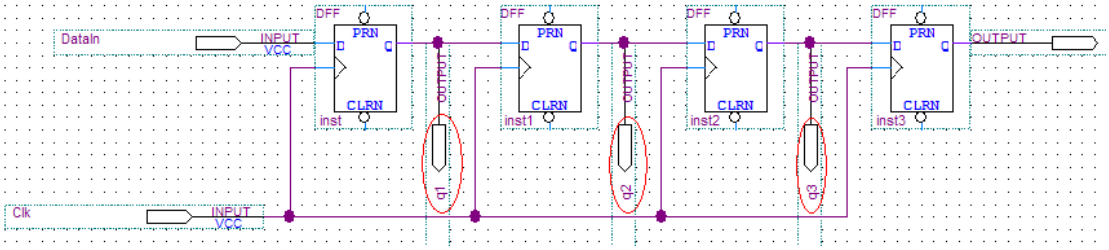


图 13-18　添加 D 触发器 q1, q2, q3 输出引脚

进行功能仿真和时序仿真遇到错误提示。要求执行 Partition Merge 操作。经过检索找到 Partition Merge 操作选项。参看图 13-19。

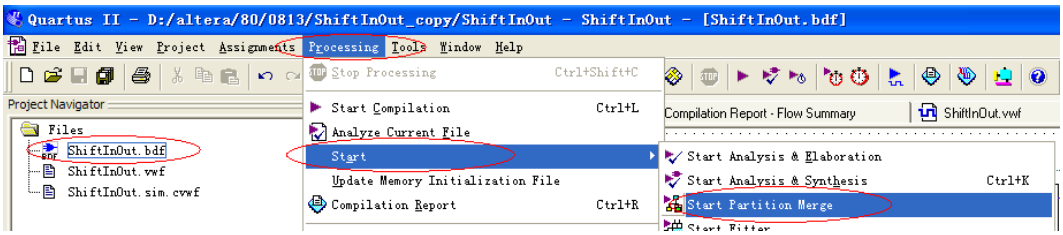


图 13-19　执行 Partition Merge 的操作项

6.　进行第二次功能仿真和时序仿真。

功能仿真和时序仿真之后，发现最后一个 D 触发器和次最后一个 D 触发器的时序关系不明显。为此希望再添加最后一个 D 触发器的输出信号。参看图 13-20。该图显示增加了最后 D 触发器的输出引脚 q4。

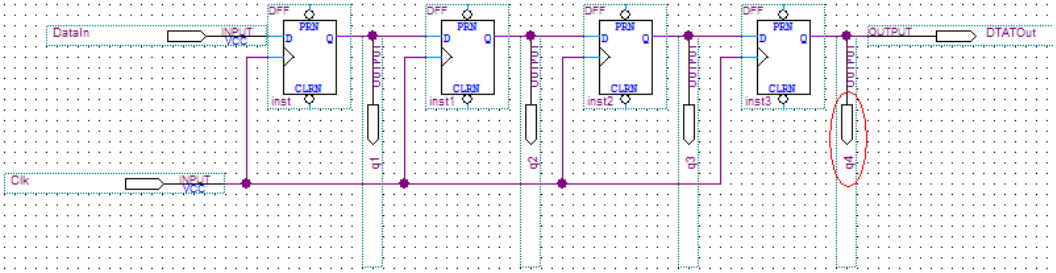


图 13-20　增加最后一个 D 触发器的输出引脚 q4

7.　进行第三次功能仿真和时序仿真。

功能仿真成功，参看图 13-21。从图中可以看出最后的 D 触发器输出相对第 1 个触发器的输出延时了三个时钟周期。

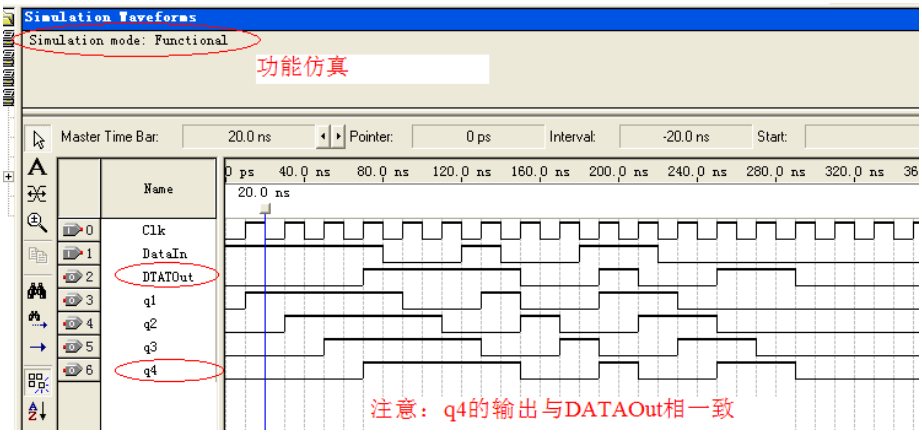


图 13-21　四个 D 触发器都添加输出引脚之后的功能仿真结果

然而接下去进行时序仿真时遇到错误提示：Can't continue simulatiom because delay annotation information for design is missing。通过因特网检索，得知进行一次全编译操作即可消除这个错误提示。为此，执行主菜单 Processing->Start Compilation（Ctrl+L）操作项。再次执行时序仿真，参看图 13-22。显然时序仿真结果是正确的。与 D 触发器的打入时钟相比较，Q 输出延时大约 6.5ns。

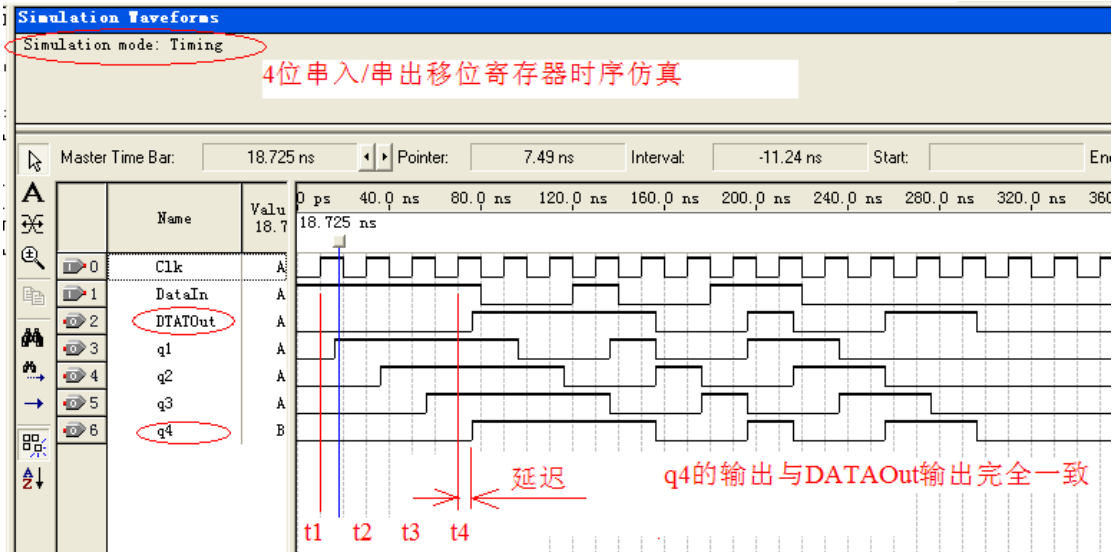


图 13-22　4 位串入串出移位寄存器的时序仿真输出快照

◆ 本实验指导结束

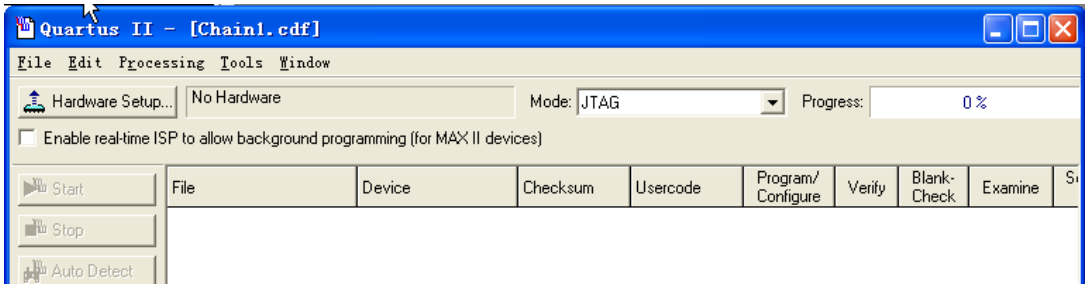
附录 1　JTAG 模式配置 FPGA

(1) 给 DE2 板子上电

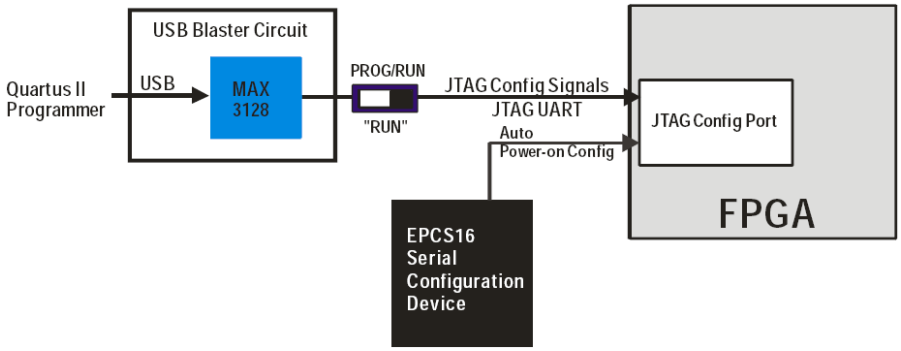
(2) 用 USB 线连接 DE2 开发板

(3) 把“RUN/PROG”开关拨到“RUN”位置

(4) 打开 Quartus II 编程器，如图附录 1-1 所示。选择 JTAG 模式，选择“.sof”结尾的配置文件，完成 FPGA 的配置。



(5) 配置完成后，FPGA 就会自动运行了。



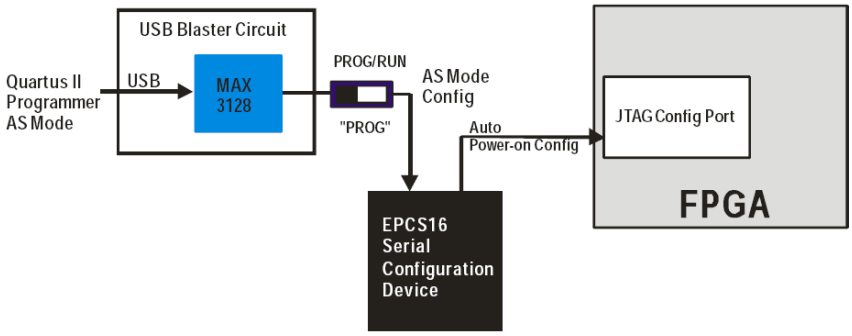
图附录 1-1　JTAG 配置方案示意图

AS 模式配置 EPCS16

(1) 给 DE2 板子上电

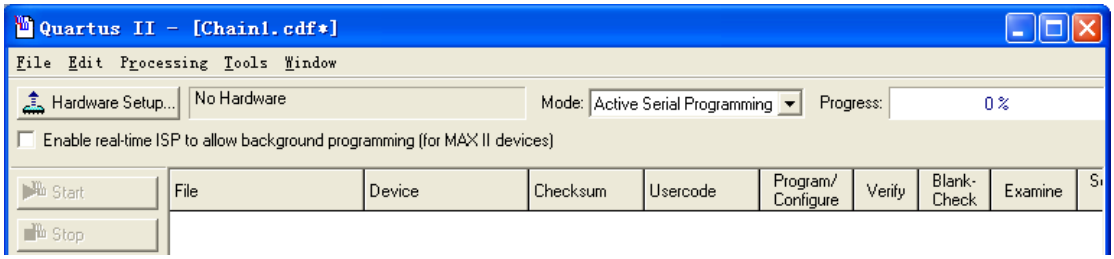
(2) 用 USB 线连接 DE2 开发板

(3) 把“RUN/PROG”开关拨到“PROG”位置，参看图附录 1-2。



图附录 1-2　AS 配置方案示意图

(4) 打开 QuartusII 编程器，如图附录 1-3 所示。选择 Active Serial Programming，选择“.pof”结尾的配置文件，完成 EPCS16 的配置。



图附录 1-3 完成 EPCS16 的配置

(5) 配置完成后，把“RUN/PROG”开关拨到“RUN”位置。

(6) 关闭电源，然后再上电。这样 EPCS16 中的数据就会自动下载到 FPGA 上。现在它会自动运行了。

附录 2　DE2-70 入门级实验操作索引

附录 3　FPGA/NIOS 常见文件格式说明

文件后缀名	用途说明
qpf	Quartus 工程文件，通常只记录 Quartus 版本号、开发日期以及各版本工程名。
qsf	Quartus 工程配置文件，记录各种配置信息，常用来直接编辑添加引脚分配信息。
qws	Quartus 工程工作空间文件，记录当前工作空间与操作系统的一些交互信息，例如强文件名
v	Verilog 源代码文件

vhd	VHDL 源代码文件
bdf	符号框图文件，等效于源代码文件
vwf	仿真波形文件，记录仿真波形信息
stp	Signal Tap II 逻辑分析仪文件，记录逻辑分析信息
sof	编译结果文件，用于下载到 FPGA 上执行
pof	FPGA 配置文件，用于修改 FPGA 加电启动项
ptf	SOPC Builder 对 Nios II IDE 的接口文件，用于生成 System.h
sopc	SOPC Builder 配置文件，记录 SOPC 系统中各器件的配置信息
qif	SOPC 路径指定文件，主要用于记录自定义 SOPC 模块的路径

附录 4　参考图书和文献

[1]夏宇闻，甘伟译，Verilog HDL 入门（第 3 版），北京航空航天大学出版社，2008 年 9 月第 1 版，ISBN 978-7-81124-248-5。

特点：①自顶向下给出了使用 Verilog 语言构建数字系统的原则和方法，视野开阔。

[2]王金明编著，Verilog 程序设计教程，人民邮电出版社，2004 年 1 月第 1 版，ISBN 7-115-11939-2。

[3]郑燕等编著，基于 VHDL 语言与 Quartus II 软件的可编程逻辑器件应用与开发，国防工业出版社，2007 年 3 月第 1 版。

特点：①实验代码逐行注释，适合初学者使用。

[4]曾繁泰，陈美金著，VHDL 程序设计，清华大学出版社，2001 年 1 月第 2 版，

特点：①详尽的语法介绍，可以供编程者查阅。

[5]周润景等编著，基于 Quartus II 的 FPGA/CPLD 数字系统设计实例，电子工业出版社，2007 年 8 月第 1 版，ISBN 978-7-121-04091-7。

特点：①结合 Quaruts II 软件工具，给出了大量的数字逻辑电路和数字系统的 FPGA 编程实例，适合初学者模仿实习、借鉴和参考。

[6]王振红主编，VHDL 数字电路设计与应用，实践教程，机械工业出版社，2003 年 6 月第 1 版，ISBN 7-111-12115-5。

[7]侯伯亨，VHDL 硬件描述语言与数字逻辑电路设计—电子工程师必备知识，西安电子科技大学出版社，1997 年 9 月第 1 版。ISBN 7-5606-0534-6。

后记

这一本 DE2-70 入门级实验指导书的撰编过程中由多位助教和老师参与。最初电子版来源于上海交通大学电子信息与电气工程学院孙广跃老师主持的 2008 年秋季 FPGA 短期培训班。除此之外，还陆续参考了 Altera 公司的大学计划网站资料，以及与 DE2 实验平台相关的其他一系列网站资料。之后在 2009 年春季，由南京大学计算机系的齐敬先、姜孟冯、刘长辉、翁基伟、杨晓亮、莫志刚、杨嘉、宗恒、叶俊杰、肖韬等硕士生助教整理成册，并做了大量试用。目前，本实验指导书能够满足大学本科生的基于 DE2-70 平台的实验教学需要。

本实验指导书编辑过程中的主要实用版是 1.0 版、1.5 版、2.0 版、2.3 版和 2.87 版。到现在为止，最新实验指导书版本为 2.90 版。在本实验指导书的编写使用过程中，计算机系的教师吴海军、蔡晓燕、张泽生、俞建新、武港山、袁春风和黄宜华等对它做了不断地精心审阅与修改。

本次打印装订的实验指导书是第 2.90 版。其中的第 2-10 章介绍了使用 Verilog HDL 编码的实验项目，第 11 章介绍了 NIOS+μCOS-II 实验项目，该实验工程是一个十分有用的 NIOS+μCOS-II 的编码框架。第 12 章介绍了 VHDL 编码的实验项目，第 13 章介绍了使用原理图输入的实验项目。可以认为全部章节都是 DE2-70 平台的基础性入门级实验项目。

由于时间仓促，水平有限，本入门级实验指导书的内容难免存在一些不足之处。为了更好地让学生在 DE2-70 实验平台上开展实习，希望使用者在使用过程中，及时地将发现的错误和不妥之处记录下来；另外，在使用过程中如果有对本实验指导书的改进建议和意见，也请及时记录下来。无论问题信息或者改进建议都请使用者反馈给南京大学计算机系的叶俊杰助教和杨晓亮助教。对于读者的反馈信息，南京大学计算机系会给以适当的回答和改进处理。

E-mail 联系方式：
叶俊杰， ye020510625@126.com
杨晓亮， yangxiaoliang2006@gmail.com

邮政联系方式：
邮寄地址：南京大学计算机系
邮编：210093

■