

件便会自动识别我们开发板上的Qsys系统，并显示Qsys系统的相关信息。我们接着点击【Apply】→【Run】，软件会把hello_world.elf文件下载至开发板中，如图 2.5.2所示。

下载结束后，程序自动开始运行，在软件下方的“Nios II Console”中会打印“Hello,World！”信息，说明我们的Hello World实验在新起点开发板上下载验证成功，如图 2.5.3所示：

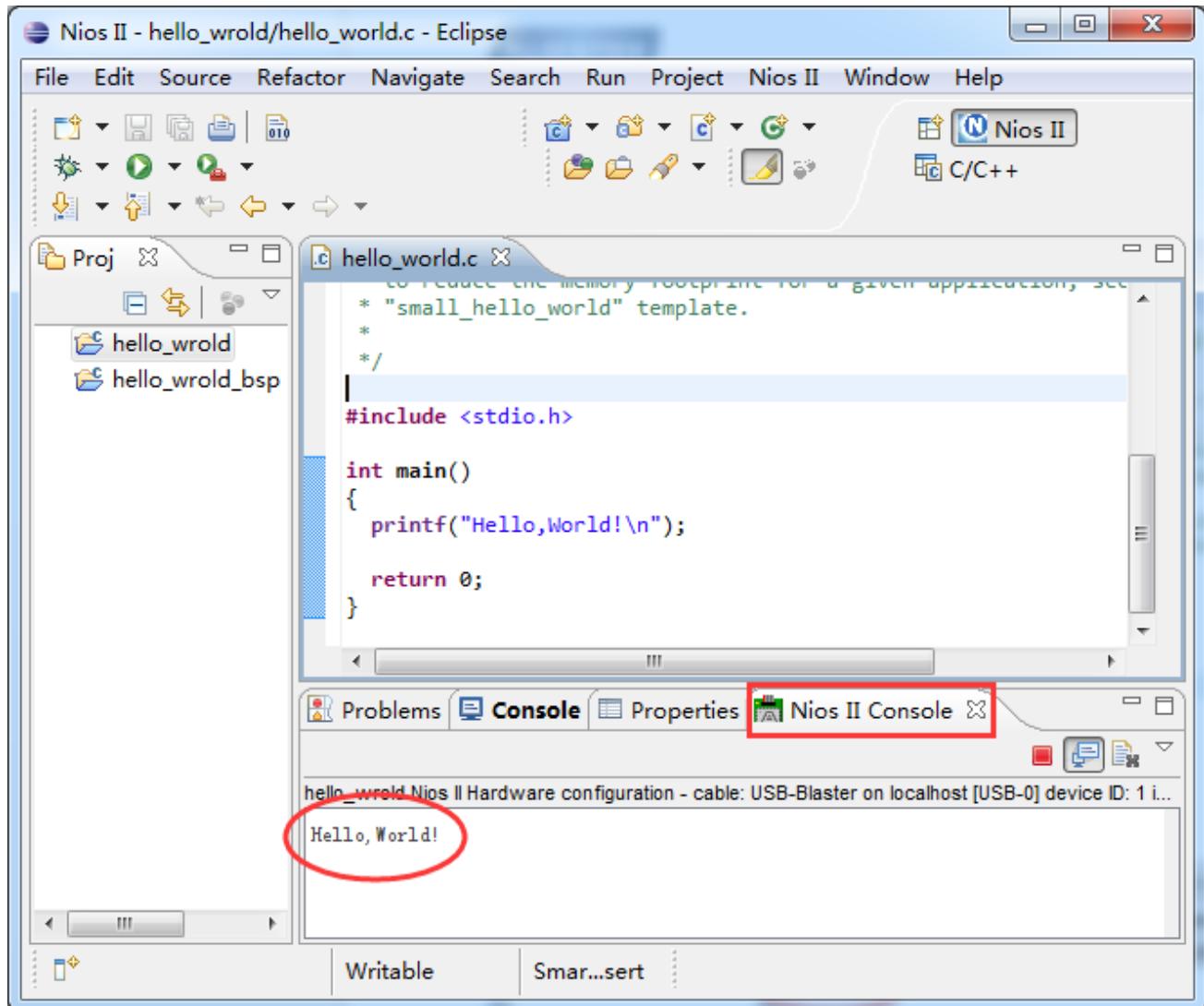


图 2.5.3 程序运行结果

第三章 PIO IP核

PIO是一个带有Avalon接口的并行输入/输出（Parallel input/output）IP核，我们可以利用PIO来实现Qsys系统与外设的简单通信。在本章中，我们将详细讲解它的功能和使用方法，并在实验过程中带领大家了解Qsys系统是怎么与外界传递信息的。

本章包括以下几个部分：

- 3.1 简介
- 3.2 实验任务
- 3.3 硬件设计
- 3.4 软件设计
- 3.5 下载验证

3.1 简介

PIO IP核为 Avalon-MM 从端口和通用I/O端口提供了一个存储器映射（memory-mapped）接口。PIO的I/O端口可以连接到片内用户逻辑（verilog语言完成的电路部分），也可以连接到与外部器件相连的FPGA引脚。每个PIO核能提供最高32个I/O端口，用户可以在一个Qsys系统里添加多个PIO IP核。

PIO 为用户逻辑或外围器件提供了简单的 I/O 控制，如控制 LED 灯、获取按键的电平变化信息、控制显示设备、与片外器件通信等。

寄存器描述

PIO IP核的寄存器相关信息如图表 3.1-1所示。

表 3.1-1 PIO IP核寄存器描述

偏移量	寄存器名称	读/写	(n-1)	...	2	1	0
0	data	读	读当前时刻 PIO 输入端口的值				
		写	往 PIO 输出端口写新的值				
1	direction (1)	读/写	控制每个 I/O 的数据方向，0 为输入；1 为输出				
2	Interruptmask (1)	读/写	打开或关闭每个输入端口的 IRQ 功能				
3	edgecapture (1) , (2)	读/写	设定的边沿产生时，对应的位置 1				
4	outset	写	设定输出位为 1				
5	outclear	写	设定输出位为 0				

注：（1）这个寄存器的存在与否，取决于硬件配置。如果寄存器不存在，读寄存器时将会返回未定义的值。同理，写寄存器也是无意义的（no effect）。

（2）如果将边沿捕获寄存器的位清除使能关掉，那么给边沿捕获寄存器写任意值都将把寄存器里的所有位清零。否则，给寄存器的某一位写 1，只会将写 1 的那一位清零。

接下来我们将对表中的寄存器一一介绍：

（1）数据寄存器（data Register）

读数据寄存器，将会返回输入端口的当前数据。如果PIO核的硬件设置为输出端口（output ports only），那么读数据寄存器将会返回未定义的值。

将数据写到数据寄存器中，寄存器中的数据会被送到输出端口。如果PIO核的硬件设置为输入端口（input ports only），那么将数据写到寄存器中是没有作用的。如果PIO核的硬件设置为双向模式（bidirectional mode）时，只有当方向寄存器（direction）对应位的值为1的时候，才输出数据。

（2）方向寄存器（direction Register）

当PIO核的硬件设置为双向端口时（bidirectional ports），方向寄存器才会存在。PIO核设置为只输出或只输入模式时，方向寄存器不存在，此时读方向寄存器将会返回未定义的值，写方向寄存器也是无意义的。方向寄存器能够设置每一个端口的数据方向，当方向寄存器中的第n位设置为1时，n端口就会将数据寄存器（data）的第n位数据输出出去。方向寄存器中的第n位为0时，n端口将会把数据传输给数据寄存器的第n位。

（3）中断屏蔽寄存器（interruptmask Register）

只有当中断屏蔽寄存器的硬件设置为“Generate IRQ”时，这个寄存器才存在。如果硬件设置时未选中“Generate IRQ”，那么读中断屏蔽寄存器会返回未定义的值，写中断屏蔽寄存器无作用。复位后，中断屏蔽寄存器所有位为0，因此禁止所有PIO端口的中断。当中断屏蔽寄存器的某一位设为1时，才能使相对应的PIO输入口中断。注意，PIO IP核的硬件配置只为输入口时才能进行中断操作。

（4）边沿捕获寄存器（edgecapture Register）

当硬件设置有边沿捕获功能时，边沿捕获寄存器才存在。否则，读边沿捕获寄存器会返回未定义的值，写边沿捕获寄存器无作用。要检测的边沿类型需要在PIO IP核添加时设定。在输入口上检测到选定的边沿时，边沿捕获寄存器中对应位n置1。Avalon主控制器通过读边沿捕获寄存器来判断，是否有某一个输入端口发生了边沿事件。写任意值到边沿捕获寄存器将使寄存器所有位清0。

（5）置位和清零寄存器（outset and outclear Register）

只有当“Enable individual bit set/clear output register”选项选中时，这类寄存器才存在。我们可以使用置位或清零寄存器设置输出端口的一些位为1或0。举个例子，当我们想把输出端口的第6位设置成1时，我们把0x40写入置位寄存器就可以了。当我们把0x08写入清零寄存器时，输出端口的第3位就设置成0了。

与PIO核相搭配的是altera_avalon_pio_regs.h文件，这个文件提供了和硬件之间的底层接触（low-level access）。开发者不能修改这个文件。

PIO IP 核的配置界面介绍

从设置界面中可以看到，IP核有5个部分可以配置，下面我们会讲解这些选项的功能。

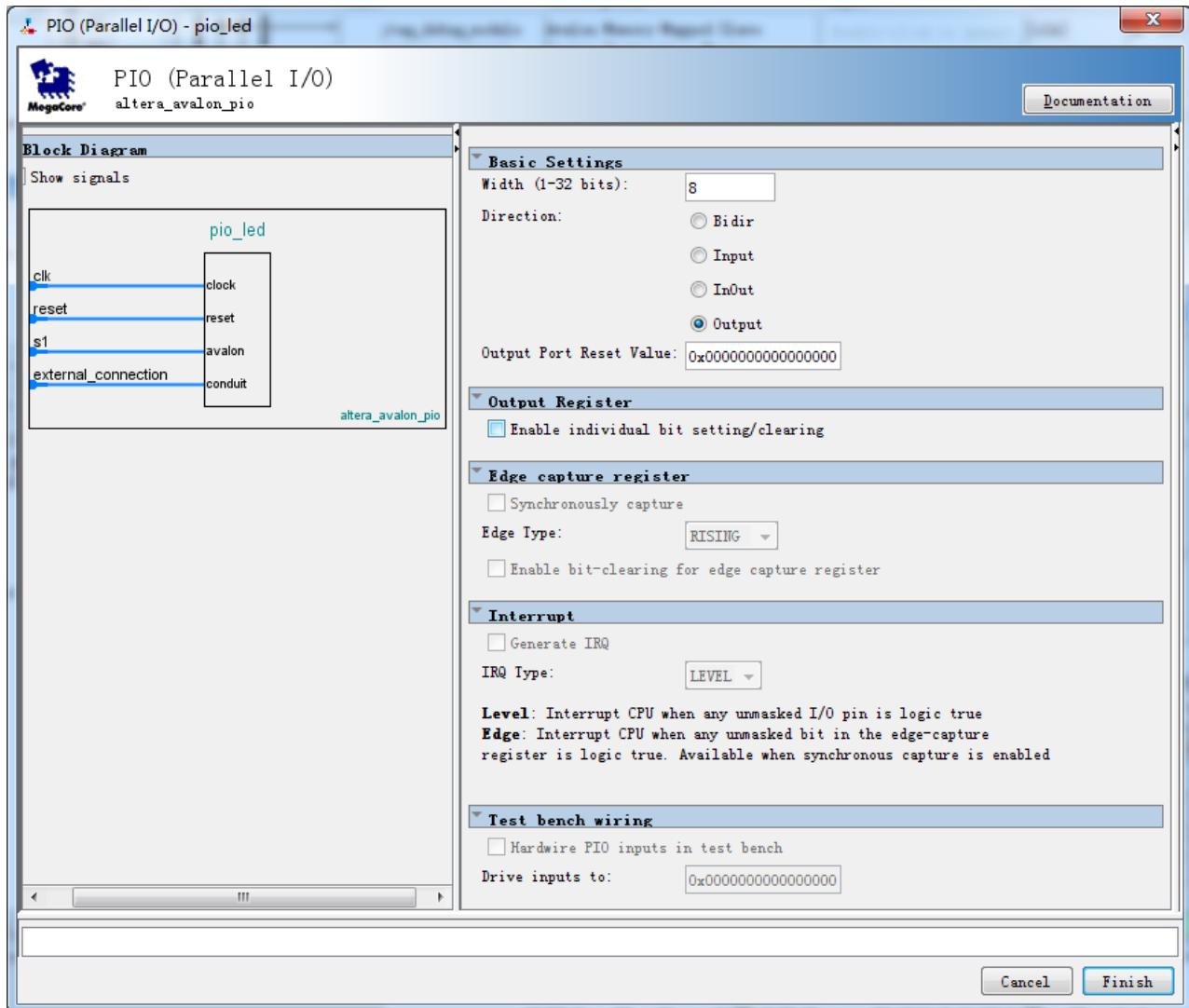


图 3.1.1 PIO IP核配置界面

(1) 基础设置 (Basic Settings)

在基础设置部分，我们可以设置PIO端口的数据方向、位宽和复位值。

Width

前面提到，PIO核能提供最高32个I/O端口，也就是说PIO端口的数据位宽最大为32bits。在Width这一项，完成对端口位宽的设置；

Direction

我们可以将端口方向设置为表中的一种

表 3.1-2 Direction设置

设置	描述
Bidirectional (tristate) ports	在这个模式下，每个 PIO 位使用一个引脚来驱动和捕获数据。可单独设定每个引脚方向。若要使 FPGA 的 I/O 引脚为三态状态，需要把 I/O 设置为输入状态。
Input ports only	在这个模式下 PIO 端口只能捕获输入数据
Output ports only	在这个模式下 PIO 端口只能驱动输出
Both input and output ports	在这个模式下，输入和输出端口总线独立，同时生成输入和输出单开

输出端口复位值

我们能确定输出端口的复位值。有效值的长度取决于端口位宽。

(2) 输出寄存器（Output register）

“Enable individual bit set/clear output register”这个选项能让我们置位或清零各自独立的输出端口。当这个选项被选中之后，会出现两个额外的寄存器——置位寄存器、清零寄存器。我们可以用这些寄存器去确定要置位或清零的位。

(3) 边沿捕获寄存器（Edge capture register）

当我们在Basic Settings中的Direction处选中Input选项时，PIO IP 核便允许我们选中边沿捕获和产生IRQ选项。当我们选择Output Ports 选项时，边沿捕获和产生IRQ 选项是无法选中的。当我们选中Synchronously capture 选项时，PIO IP 核中将包含边沿捕获寄存器（edge capture）。当一个输入端口产生了特定的边沿时，边沿捕获寄存器能让核检测或产生一个可选的中断。用户需要进一步指定以下功能：

- 要检测的边沿种类

- Rising Edge 上升沿；

- Falling Edge 下降沿；
- Either Edge 上升或下降沿。
- 选择了 enable bit-clearing for edge capture register 选项后，给 edge capture 寄存器单独的位写 1 清中断；若是没有选择 enable bit-clearing for edge capture register，则是写任意数清中断。

当指定类型的边沿在输入端口出现时，边沿捕获寄存器对应位置1。当Synchronously capture 选项未选中时，边沿捕获寄存器不存在。

(4) 中断 (Interrupt)

当选中 Generate IRQ 选项时，若输入端口发生指定的事件，则 PIO IP 核发出 IRQ 中断。用户必须进一步设定 IRQ 事件产生的条件：

- Level 只要选中了中断功能并且输入为高电平，那么 PIO IP 核产生一个 IRQ。
- Edge 只要选中了中断使能功能且边沿捕获寄存器中相应位为 1，那么 PIO IP 核产生一个 IRQ。

当Generate IRQ 选项未被选中时，不产生中断屏蔽寄存器。当硬件配置为电平触发方式时，只要选中了中断使能功能并且出现高电平，就发出一个中断。如果希望低电平时产生中断，则需在该I/O 输入引脚前将输入信号取反；当硬件配置为边沿触发方式时，只要选中中断使能功能并且捕获到边沿事件，就发出一个中断。中断 (IRQ) 一直保持有效直至禁止中断（中断屏蔽寄存器相应位清零）或清边沿捕获标志（向边沿捕获寄存器写一个任意值）为止。

(5) 仿真 (Test bench wiring)

由于这个功能不常用，所以这里就不进一步介绍了。

3.2 实验任务

本节实验任务是：在Qsys系统中加入PIO IP核，完成4个按键控制4个LED亮灭的实验，并实现上电自启动的功能。

3.3 硬件设计

创建 Quartus II 工程

仿照第二章“Hello World”实验硬件设计部分来创建本次实验的文件目录以及Quartus II工程，工程名为“Pio_led”。

创建 Qsys 系统

本次实验相对应“Hello World”实验，硬件上有一定的区别。由于本次实验要用到FPGA芯片外部的按键和LED，所以搭建的Qsys系统需要和外部器件进行通信，就要用到前面介绍的PIO IP核。另外实验任务要求能够实现上电自启动的功能，那么就需要用到非易失性存储器EPCS，也就是说需要添加EPCS Flash IP核。

首先按照“Hello Wrold”实验所介绍的方法完成最基本的Qsys系统的搭建，要用到的IP核有：clk、nios II、onchip_ram、jtag_uart、sysid_qsys。接下来，我们来介绍一下如何给该Qsys系统添加并配置PIO和epcs_flash IP核。

在Qsys工具的Library中搜索“PIO”，在搜索结果中双击PIO IP核可以打开如图 3.3.1 所示的配置界面。

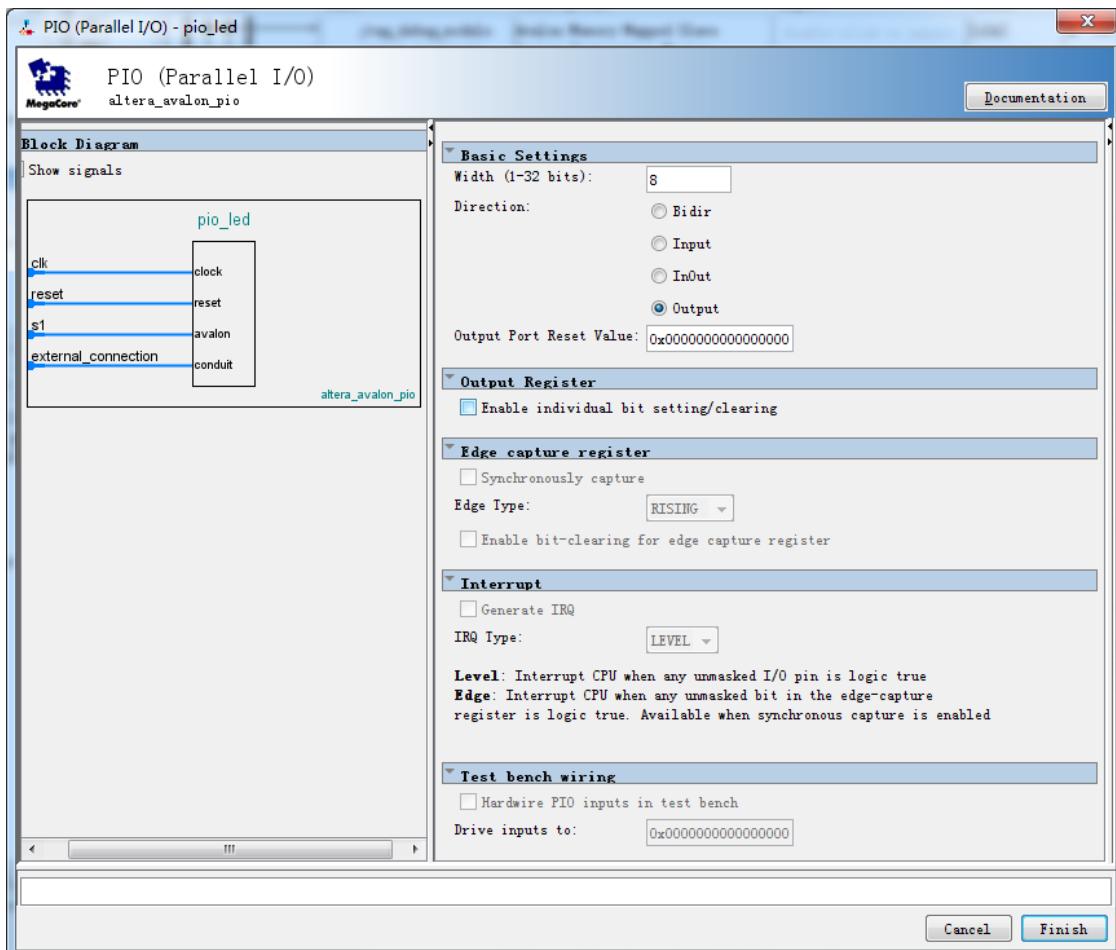


图 3.3.1 PIO IP核配置界面

我们先设置与LED灯相连的PIO。开发板上用户可用的LED灯共有4个，因此在“Basic Settings”一栏，我们将PIO位宽（Width）设置为4；除此之外，因为LED灯是一个输出设备（由FPGA输出的信号控制），所以我们把PIO的方向（Direction）设为输出（Output）即可。

其它设置如Output Register、Edge capture register等配置为默认设置，无需更改，点击Finish完成。修改完的PIO IP核界面如图 3.3.2所示。

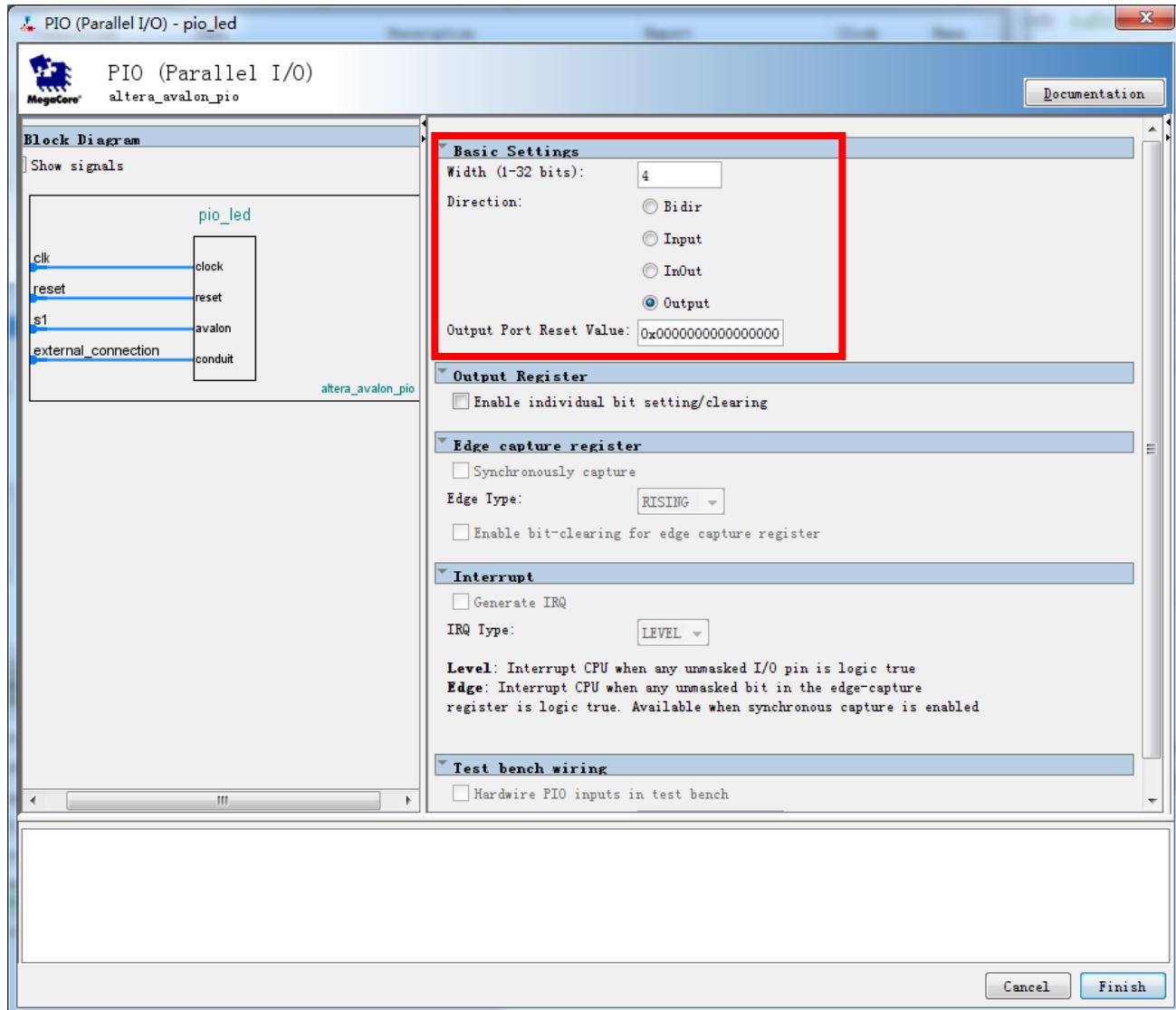


图 3.3.2 与LED灯相关的PIO IP核设置界面

由于我们的实验任务是使用4个按键控制4个LED灯的亮灭，所以还要添加一个采集按键电平信号的PIO IP核。这里因为按键的电平信号是要输入给Qsys系统的，所以在Basic Settings里，我们将PIO位宽设为4，方向设为输入（Input）即可。由于本次实验用不到中断以及边沿相关的功能，其他的设置也为默认设置，无需修改。修改完的PIO IP核界面如图

3.3.3所示。

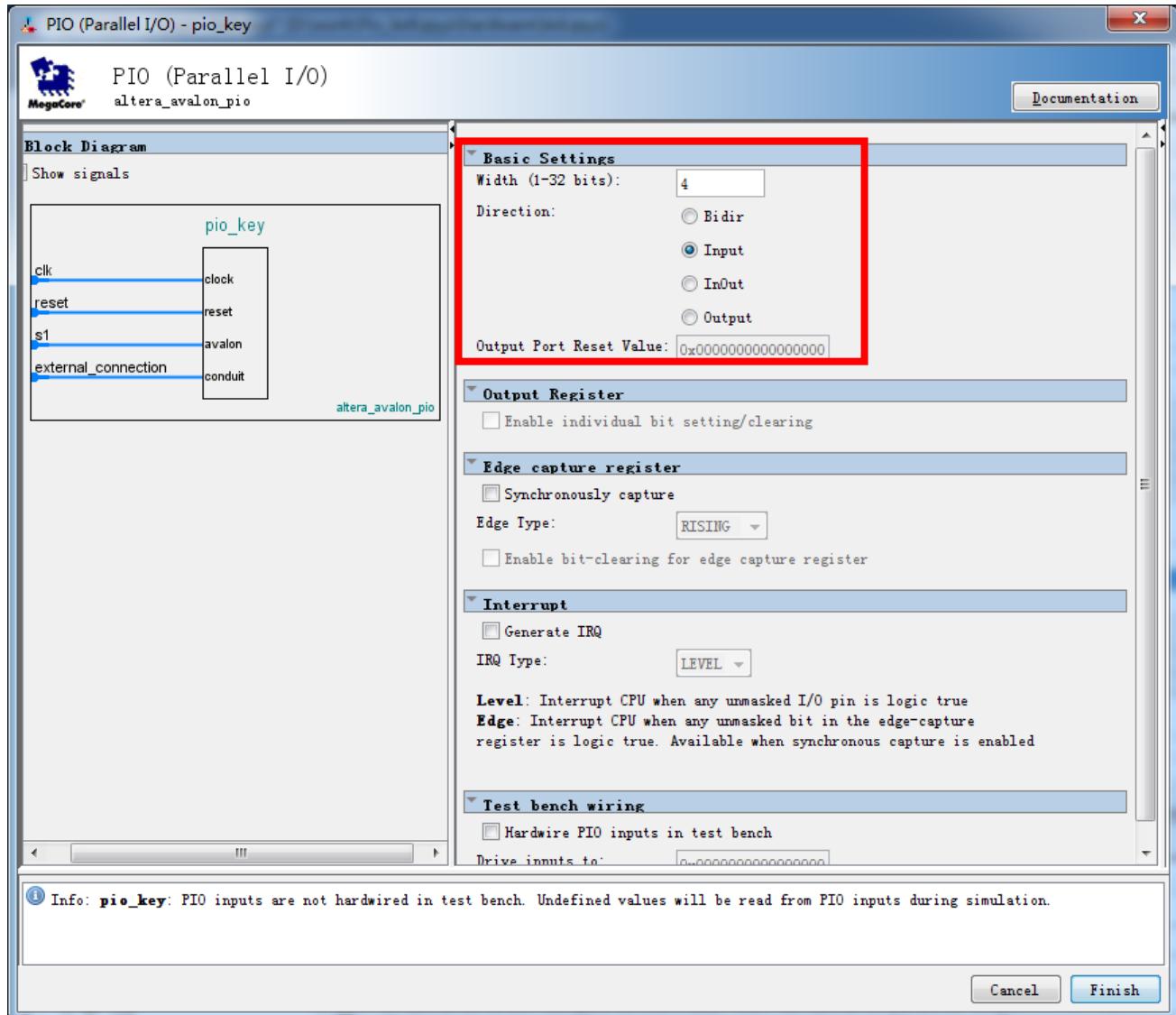


图 3.3.3 与按键相关的PIO IP核设置界面

紧接着，在Library中搜索epcs_flash IP核，双击打开后可以看到如图 3.3.4所示的配置界面。

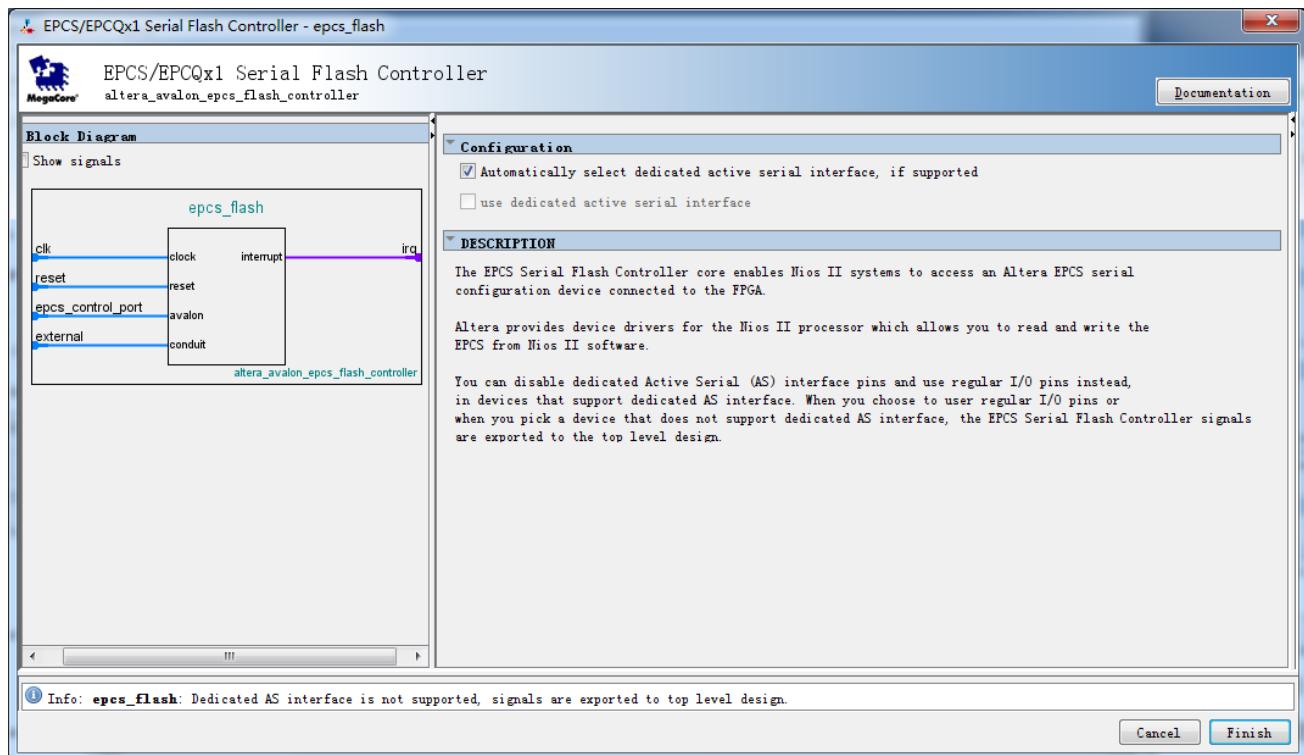


图 3.3.4 epcss_flash IP核设置界面

这里，epcs_flash IP核直接使用默认设置，点击Finish即可完成配置。由于这里只是使用这个IP核固化程序（让设计上电自启动），所以这里就不进一步详细介绍。

然后打开nios II IP核配置界面，因为这里使用的是epcs_flash IP核存储代码和指令，所以相关的设置需要进行修改。如图 3.3.5所示，在Reset Vector这一栏将Reset vector memory设置为epcs_flash即可，Exception Vector这一栏处还是使用onchip_ram。需要注意的是，在添加完epcs_flash和onchip_ram IP核之后，nios II IP核的Reset Vector和Exception Vector选项中才会出现epcs_flash以及onchip_ram选项。

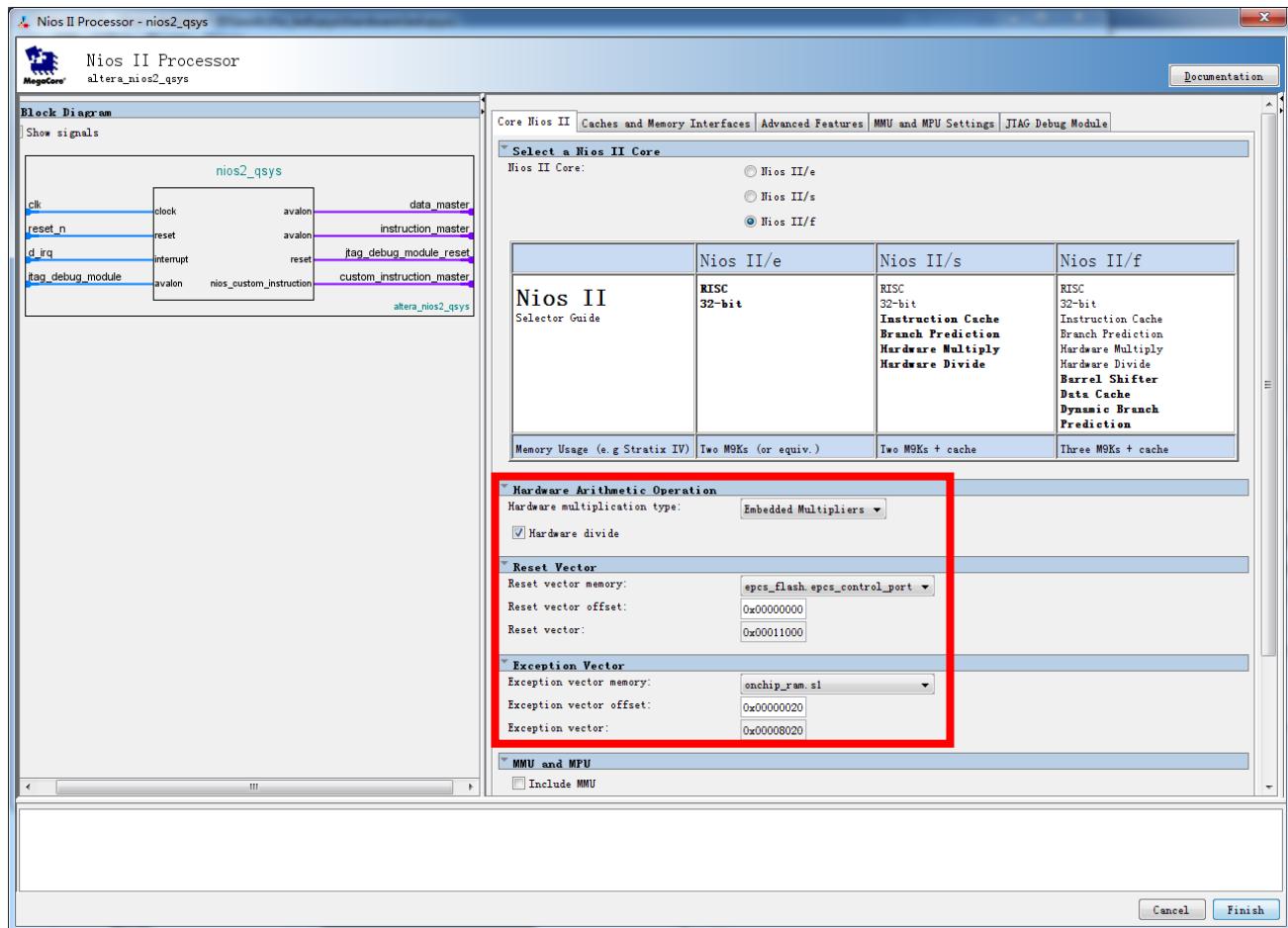


图 3.3.5 nios II IP核设置界面

然后就是进行连线，大家若是不熟悉怎么连线，可以照着下面完成的Qsys系统界面图连。需要注意的是，要将PIO和epcs_flash IP核的端口引出来，如图 3.3.6所示。引出端口的方法是双击图 3.3.6中IP核的Export一栏的红框位置，然后修改名称，按下Enter键即可。

然后，点击System→Assign Base Addresses让系统自动分配地址，这里最好把EPCS Flash的地址锁住，这是因为这个IP核里存储着指令，最好不要让其地址发生变动。锁住地址的方法是先点击IP核，然后点击右键→Lock Base Address。我们还可以将各个IP核的名称修改一下。最后就是生成系统了，操作可以按照“Hello, World”文档里的进行。

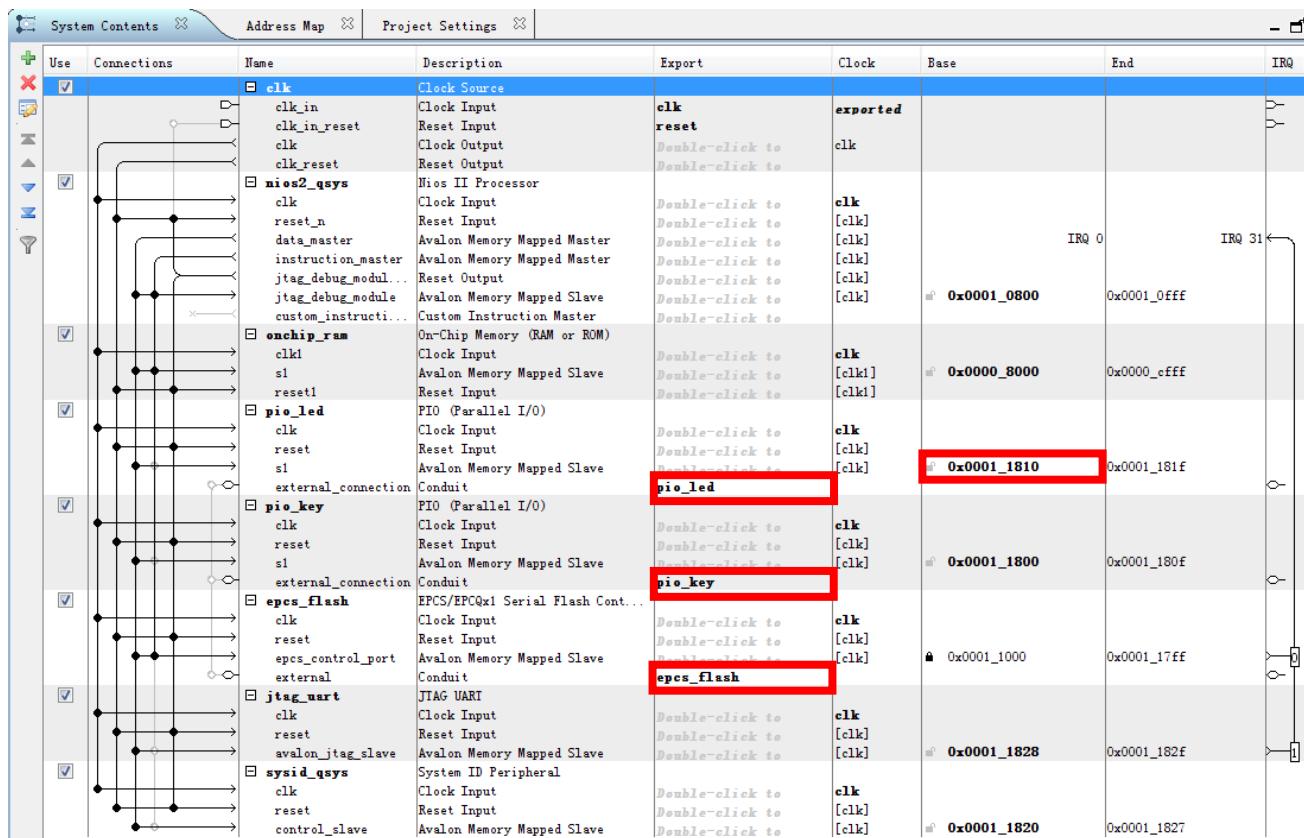


图 3.3.6 Qsys系统界面

集成 Qsys 系统

这一步依然可以按照“Hello, World”文档里的操作进行。

下面将Quartus II 工程中的顶层代码贴出来。

```

1 module Pio_led(
2     input  sys_clk,
3     input  sys_rst_n,
4
5     //flash
6     input  flash_data0,
7     output flash_sdo,
8     output flash_sce,
9     output flash_dclk,
10
11    input  [3:0] key,
```

```
12    output [3:0] led
13 );
14
15 //wire define
16 wire clk_100m ;
17
18 //例化pll(锁相环)IP核
19 pll pll_inst (
20 .inclk0      ( sys_clk ),
21 .c0          ( clk_100m )
22 );
23
24 //例化Qsys系统
25 led u0 (
26   .clk_clk      (clk_100m ),           // clk.clk
27   .reset_reset_n (sys_rst_n),           // reset.reset_n
28   .epcs_flash_dclk (flash_dclk),       // epcs_flash.dclk
29   .epcs_flash_sce (flash_sce),          // .sce
30   .epcs_flash_sdo (flash_sdo),          // .sdo
31   .epcs_flash_data0 (flash_data0),     // .data0
32   .pio_led_export (led),                // pio_led.export
33   .pio_key_export (key)                 // pio_key.export
34 );
35
36 endmodule
```

编译和下载

这时，我们便能够进行编译查错了，我们可以通过Quartus II 软件菜单栏中的【Processing】→【Start Compilation】来进行编译，也可以通过快捷栏中的快捷键进行编译。

接下来我们就需要进行配置IO，分配管脚。首先，点击Quartus II 软件菜单栏中的【Assignment】→【Device】，然后我们在Device界面中找到【Device and Pin Options...】进入图 3.3.7所示页面配置IO。将未使用引脚设置为高阻输入（As input tri-

state)，这样上电后FPGA 的所有不使用引脚都将进入高阻抗状态。

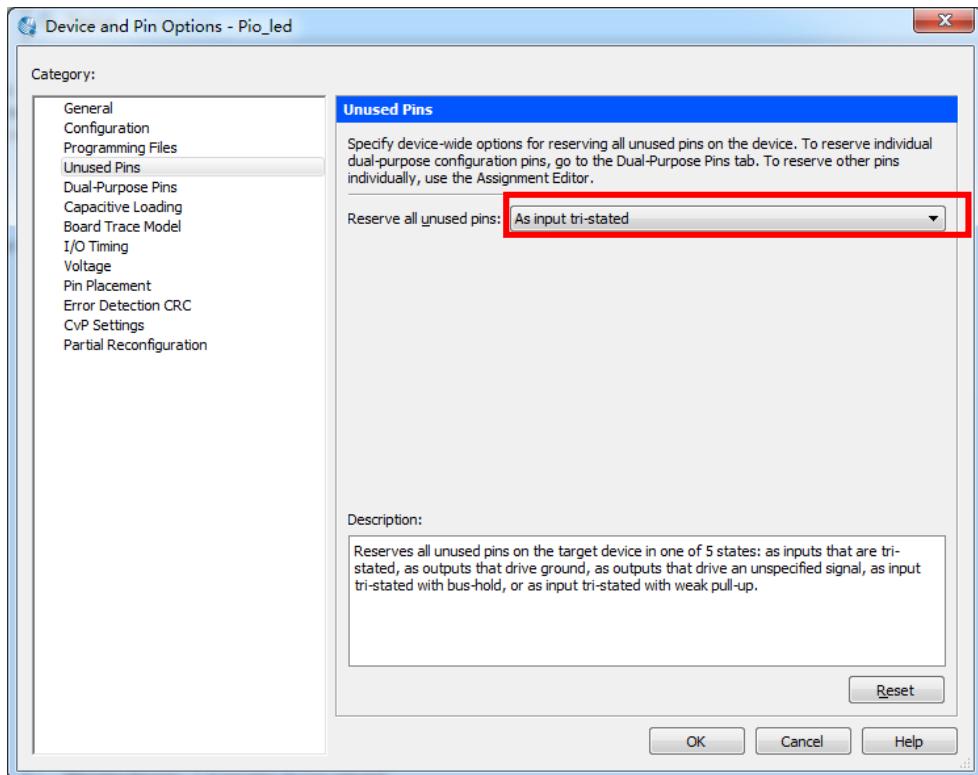


图 3.3.7 未使用引脚设置界面

接下来，将一些IO设置成普通IO，通过双击红框位置，将一个个Value的值修改过来。如图 3.3.8所示。

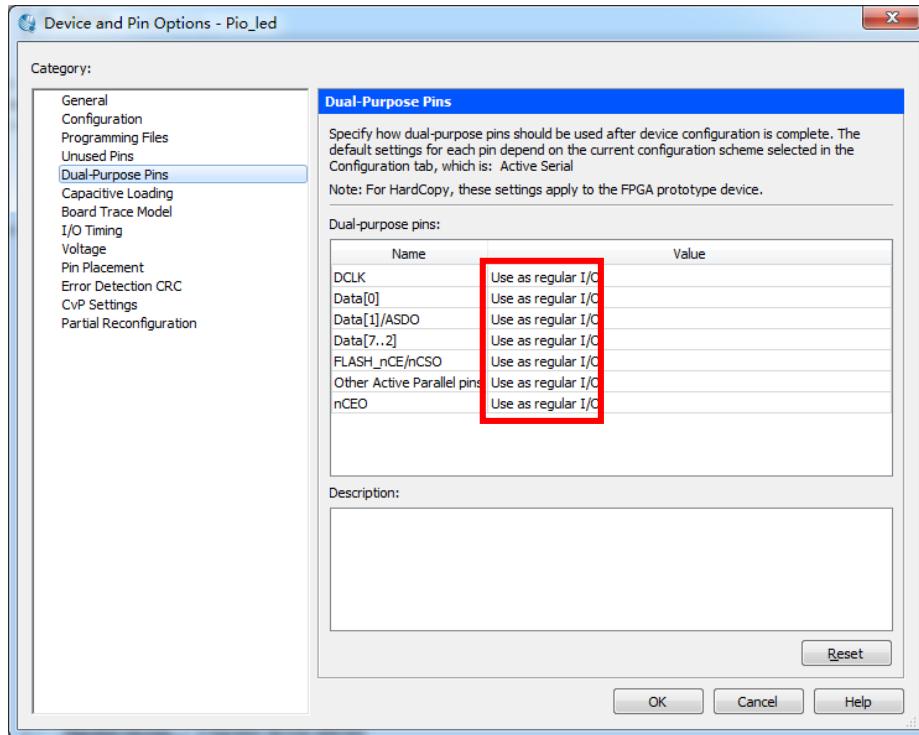


图 3.3.8 IO设置界面

我们通过Quartus II 软件菜单栏中的【Assignments】→【Pin Planner】选项分配引脚，如图 3.3.9所示。

		Type	PIN_H2	1	B1_N0	PIN_H2	2.5 V (default)
In	flash_data0	Input	PIN_H1	1	B1_N0	PIN_H1	2.5 V (default)
out	flash_ddk	Output	PIN_D2	1	B1_N0	PIN_D2	2.5 V (default)
out	flash_sce	Output	PIN_C1	1	B1_N0	PIN_C1	2.5 V (default)
in	key[3]	Input	PIN_M16	5	B5_N0	PIN_M16	2.5 V (default)
in	key[2]	Input	PIN_M2	2	B2_N0	PIN_M2	2.5 V (default)
in	key[1]	Input	PIN_E15	6	B6_N0	PIN_E15	2.5 V (default)
in	key[0]	Input	PIN_E16	6	B6_N0	PIN_E16	2.5 V (default)
out	led[3]	Output	PIN_F9	7	B7_N0	PIN_F9	2.5 V (default)
out	led[2]	Output	PIN_E10	7	B7_N0	PIN_E10	2.5 V (default)
out	led[1]	Output	PIN_C11	7	B7_N0	PIN_C11	2.5 V (default)
out	led[0]	Output	PIN_D11	7	B7_N0	PIN_D11	2.5 V (default)
in	sys_clk	Input	PIN_E1	1	B1_N0	PIN_E1	2.5 V (default)
in	sys_rst_n	Input	PIN_M1	2	B2_N0	PIN_M1	2.5 V (default)

图 3.3.9 引脚分配界面

最后我们再进行一次全编译，成功编译硬件系统后，将产生用于配置FPGA的Pio_led.sof文件。下面我们就来说明一下将.sof文件下载到FPGA目标器件的步骤。

将下载器一端连接电脑，另一端与开发板上对应端口连接，最后连接电源线并打开电源开关。新起点开发板实物图如下所示：

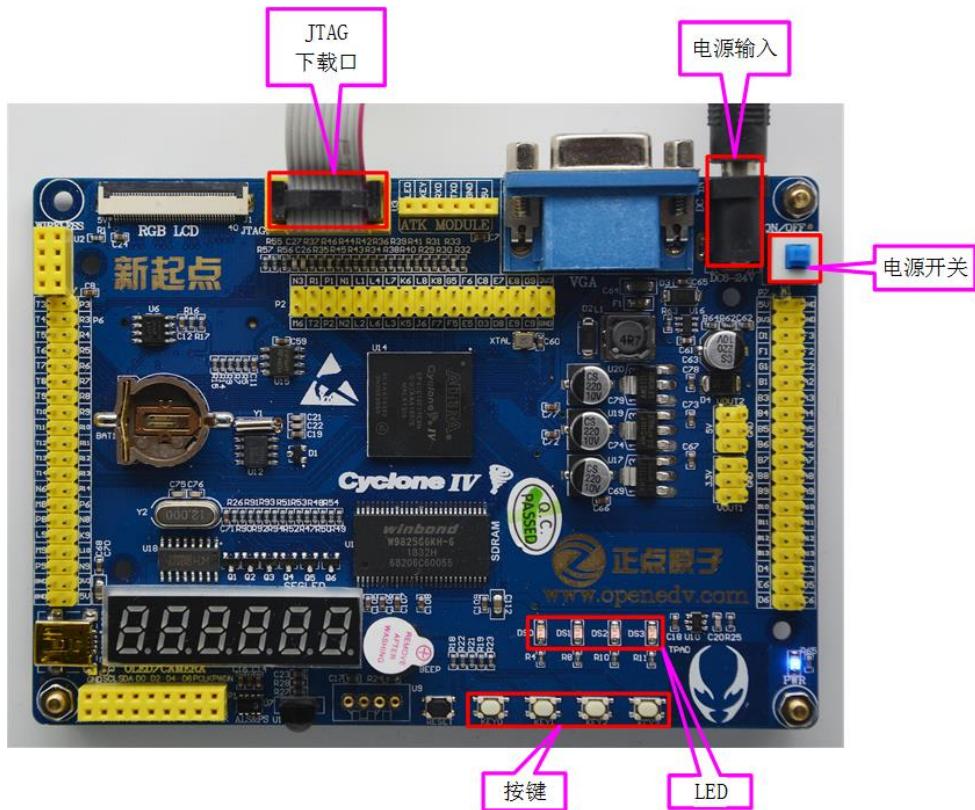


图 3.3.10 开发板实物图

接下来我们下载程序。工程打开后通过点击工具栏中的“Programmer”图标打开下载界面，通过“Add File”按钮选择Pio_led\par\output_files 目录下的“Pio_led.sof”文件。开发板电源打开后，在程序下载界面点击“Hardware Setup”，在弹出的对话框中选择当前的硬件连接为“USB-Blaster[USB-0]”。然后点击“Start”将工程编译完成后得到的 sof 文件下载到开发板中。

这里只讲到了把配置文件下载到FPGA 中，掉电后FPGA中的配置数据将会丢失。我们会在后面讲到掉电后数据不丢失的配置方法。至此，硬件部分设计完成，下面开始基于Nios II SBT for Eclipse 的软件部分的设计。

3.4 软件设计

我们通过Quartus II 软件菜单栏中的【Tools】→【Nios II SBT for Eclipse】，来启动Nios II SBT for Eclipse软件。打开Nios II SBT for Eclipse 软件后，会弹出 Workspace Launcher 页面。我们这里将工作空间设置为Pio_led\qsys路径下的software文件夹，如图 3.4.1所示。

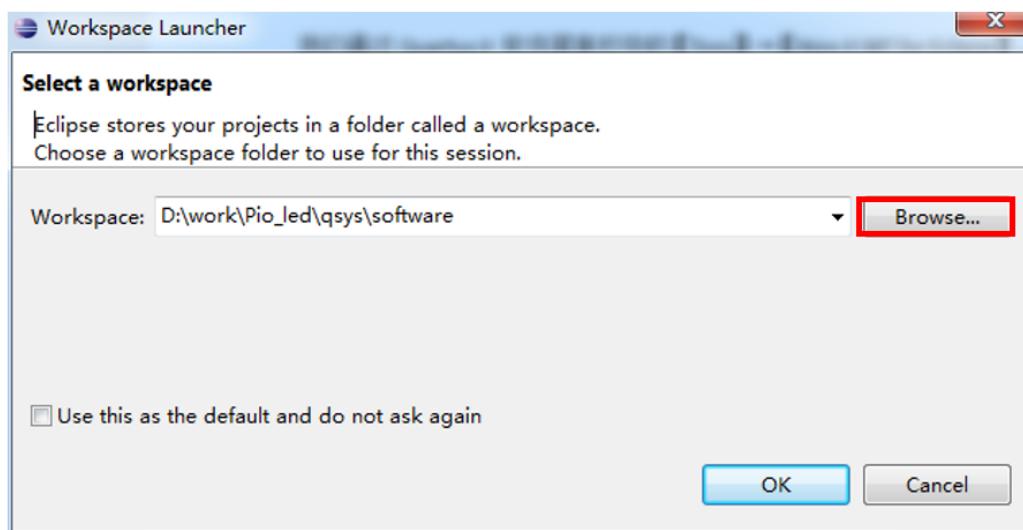


图 3.4.1 设置工作空间

设置好工作空间后，我们点击【OK】进入Nios II SBT for Eclipse 软件主界面中，在该页面我们通过单击菜单栏中的【File】→【New】→【Nios II Application and BSP from Template】，来新建工程，如图 3.4.2所示。

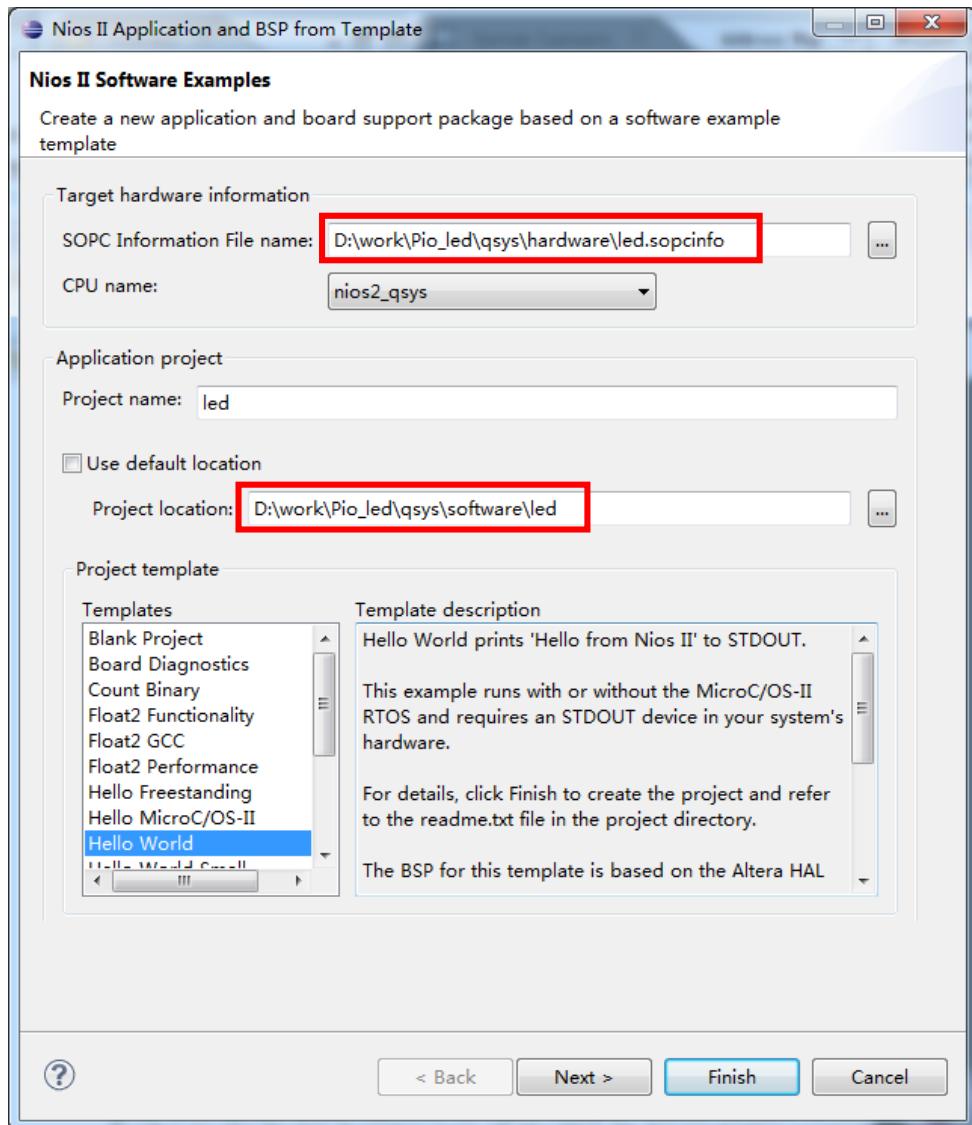


图 3.4.2 新建Nios II SBT for Eclipse 工程

单击【...】按钮来选择Pio_led\qsys\hardware下的Pio_led.sopcinfo文件，即指向当前硬件设计系统。Nios II SBT for Eclipse 软件会自动识别Qsys 系统中CPU 的名称，所以CPU name一项会自动生成。接下来，要给Nios II SBT for Eclipse 软件中的工程命名，这里的名称没有特殊要求，我们这里名为led。然后将工程存放的位置修改为 Pio_led\qsys\software\ led。注意不要漏掉了“\led”，不然生成系统的时候会报错。最后我们来看下Project template窗口，该窗口中陈列的都是已经设计好的软件工程。我们可以从中选择一个，作为自己的工程的模板来使用。当然也可以选择Bland Project（空白工程），就需要自己写所有的代码。这里我们选择的是Hello World 模板工程，然后我们在它的基础上进行修改，这样比空白工程更加方便。

设置完工程后，直接点击【Finish】完成工程创建。然后，在Nios II SBT for Eclipse

软件的左侧Project Explorer 窗口中有两个工程： led 和led _bsp。其中led 是C/C++应用工程，而led _bsp 是描述Qsys系统硬件细节的系统库。打开led工程里的hello_world.c文件，出现如图 3.4.3所示的图。

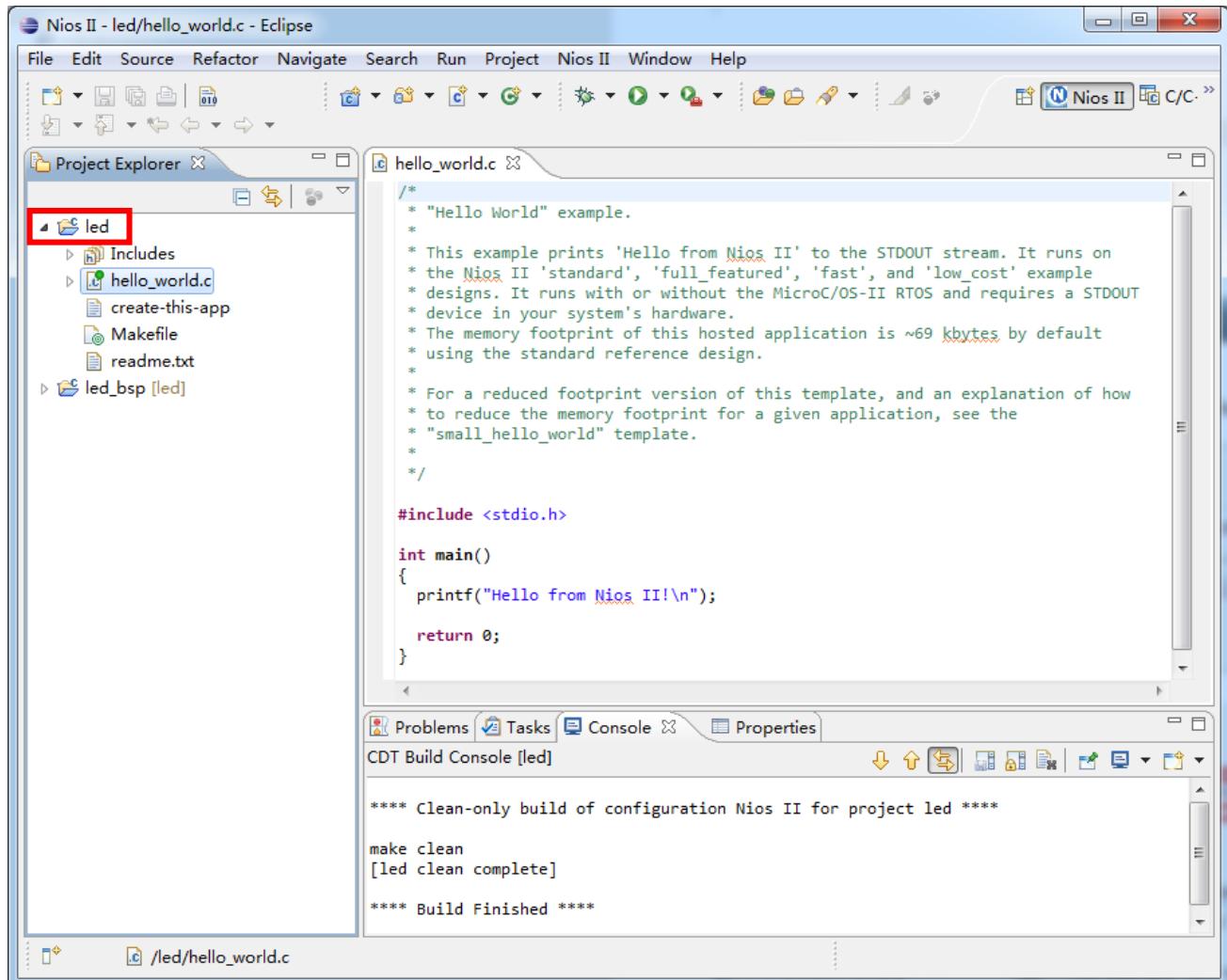


图 3.4.3 hello_world工程代码图

由代码可知，下载程序到开发板后会在窗口上输出 “Hello from Nios II!”。我们在这里要先验证之前创建的Qsys系统是否能正常工作。验证方法是先编译led工程，然后将工程模板程序下载到开发板上，看是否能正常运行。方法是右键led工程，点击build project。Console窗口会报出下图这样的错误。

```

Info: Linking led.elf
nios2-elf-g++ -T'D:/work/Pio_led/qsys/software/led_bsp//linker.x'
-msys-crt0='D:/work/Pio_led/qsys/software/led_bsp//obj/HAL/src/crt0.o'
-msys-lib=hal_bsp -LD:/work/Pio_led/qsys/software/led_bsp/ -Wl,-Map=led.map -O0 -g
-Wall -mhw-div -mhw-mul -mno-hw-mulx -o led.elf obj/default/hello_world.o -lm
d:/program/quartus/nios2eds/bin/gnu/h-i686-mingw32/bin/..../lib/gcc/nios2-elf/4.7.3/...
/.../.../H-i686-mingw32/nios2-elf/bin/ld.exe: led.elf section `.text' will not fit in
region `onchip_ram'
d:/program/quartus/nios2eds/bin/gnu/h-i686-mingw32/bin/..../lib/gcc/nios2-elf/4.7.3/...
/.../.../H-i686-mingw32/nios2-elf/bin/ld.exe: region `onchip_ram' overflowed by 12876
bytes
collect2.exe: error: ld returned 1 exit status
make: *** [led.elf] Error 1

**** Build Finished ****

```

图 3.4.4 编译工程后的console窗口

在“Hello, World”实验中也出现了类似的错误，其原因是存储空间不够。所以这里也要像“Hello, World”实验里一样，优化一下代码。方法是一样的，大家可以照着操作，这里就不再赘述了。

优化完代码之后，再编译一次led工程，会出现以下的界面。这表示编译通过，可以将程序下载到开发板上了。

```

#include <stdio.h>

int main()
{
    printf("Hello from Nios II!\n");

    return 0;
}

CDT Build Console [led]
-msys-crt0='D:/work/Pio_led/qsys/software/led_bsp//obj/HAL/src/crt0.o'
-msys-lib=hal_bsp -LD:/work/Pio_led/qsys/software/led_bsp/ -msmallc -Wl,-Map=led.map
-O0 -g -Wall -mhw-div -mhw-mul -mno-hw-mulx -o led.elf obj/default/hello_world.o -lm
nios2-elf-insert led.elf --thread_model hal --cpu_name nios2_qsys --qsys true
--simulation_enabled false --id 0 --sidp 0x11820 --timestamp 1531920027 --stderr_dev
jtag_uart --stdin_dev jtag_uart --stdout_dev jtag_uart --sopc_system_name led
--quartus_project_dir "d:/work/Pio_led/qsys/hardware" --sopcinfo
D:/work/Pio_led/qsys/hardware/led.sopcinfo

```

图 3.4.5 编译通过后的console窗口

这时大家点击【Run As】→【Nios II Hardware】，然后点击【Target Connection】标签，然后在Target Connection 窗口中点击【Refresh Connections】按钮后。这时软件便会自动识别我们开发板上的Qsys 系统，并显示Qsys 系统的相关信息。我们接着点击【Run】，

软件会把led.elf 文件下载至开发板中运行起来。更加详细的图和文字描述，可以在“Hello, World”实验的下载验证部分查看。

这时，若之前创建的Qsys系统无误，代码下载完成后在Nios II console窗口会显示“Hello from Nios II!”字符，如下图所示。

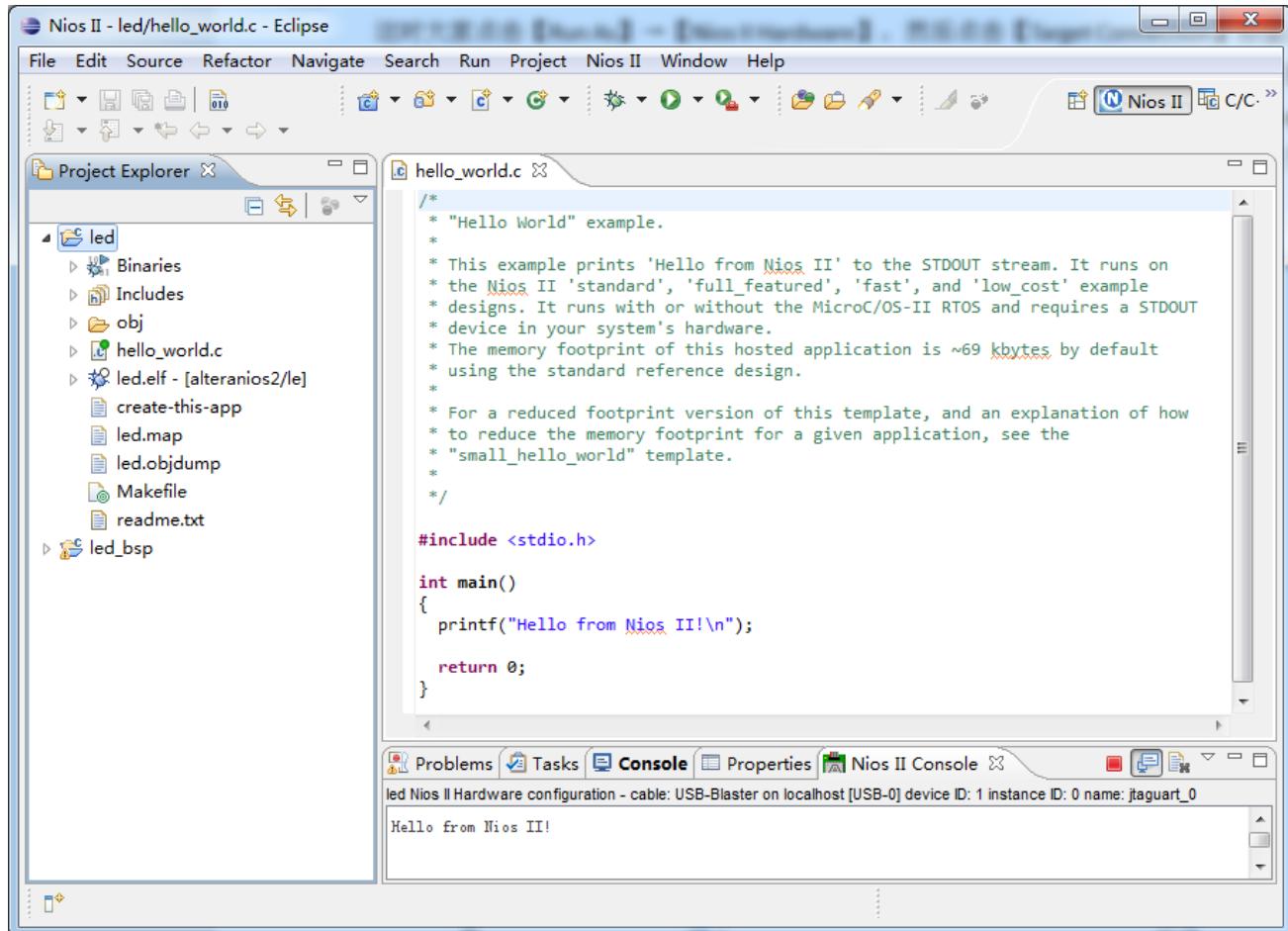


图 3.4.6 下载代码后的Nios II console窗口

验证完Qsys系统是否能正常运行之后，我们就可以开始软件部分的设计了。这时只需要在当前的代码窗口修改代码就可以了。代码如下所示。

```

1 #include <stdio.h>
2 #include "system.h" //系统头文件
3 #include "alt_types.h" //数据类型头文件
4 #include "altera_avalon_pio_regs.h"//pio 寄存器头文件
5
6 //-----
7 //-- 名称 : main()

```

```
8  //-- 功能 : 程序入口
9  //-- 输入参数 : 无
10 //-- 输出参数 : 无
11 //-----
12 int main(void)
13 {
14     alt_u32 key, led; //key 和 led 缓存变量
15     while(1)
16     {
17         //读取按键的值, 并赋值给 key。
18         key = IORD_ALTERA_AVALON_PIO_DATA(PIO_KEY_BASE);
19         //Key 按下时为低电平, 没有按下时为高电平。Led 在高电平时亮, 低电平灭。我们要将按键的值按
位取反后, 再赋值给 led。
20         led = ~key;
21         //用 led 的值控制 Led 亮灭。
22         IOWR_ALTERA_AVALON_PIO_DATA(PIO_LED_BASE, led);
23     }
24
25     return(0);
26 }
```

修改完代码的窗口如下所示:

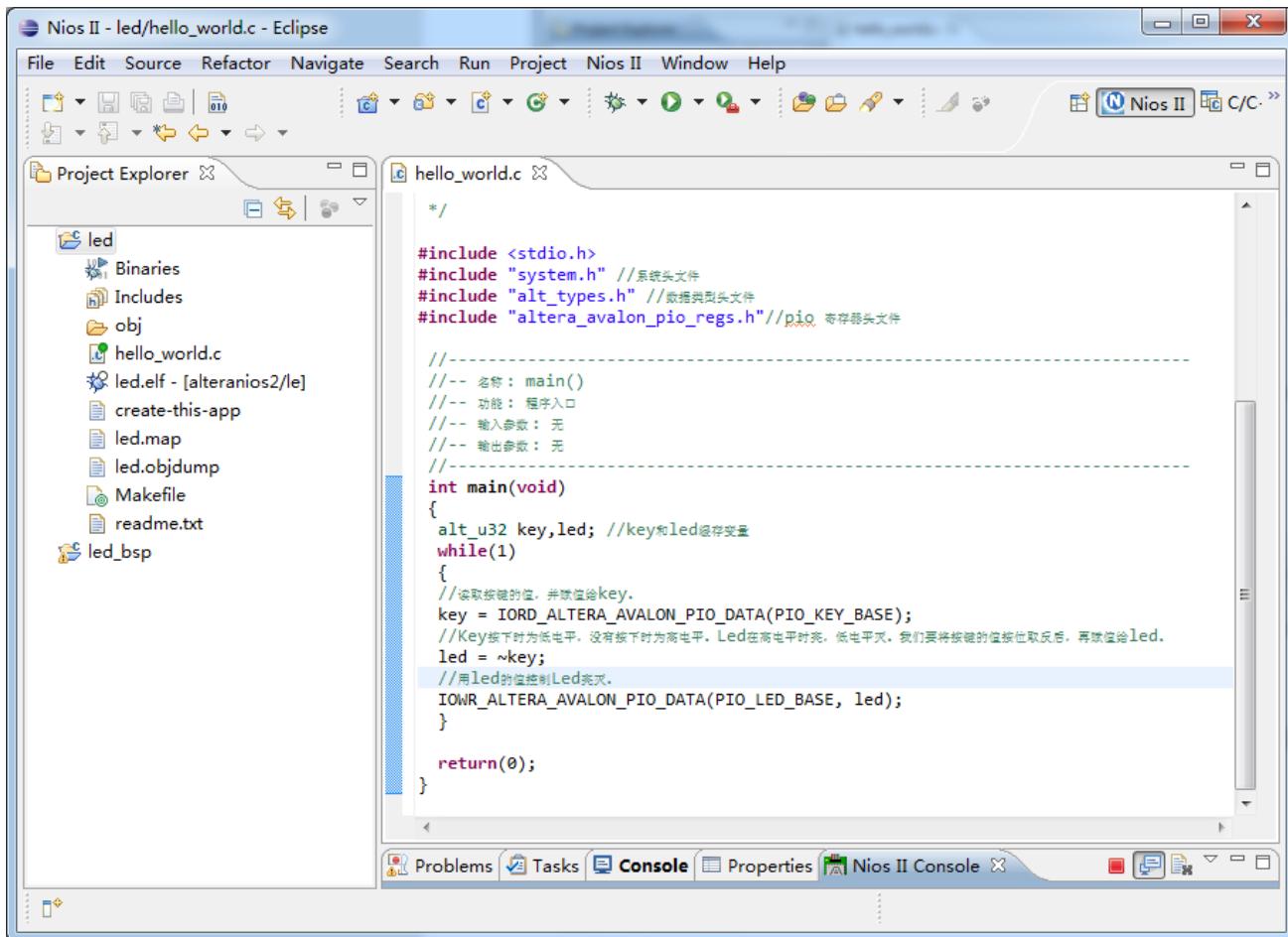


图 3.4.7 修改后的工程代码图

代码修改完成后，大家记得要点一下快捷菜单中的【Save】，或者菜单栏中的【File】→【Save】，来保存修改后的程序。然后再编译led工程，并下载到开发板上。若代码下载完成后，按下开发板上的4个按键分别能控制4个led灯的亮灭。说明代码设计无误，可以开始下一步——让设计上电自启动。

单击Nios II SBT for Eclipse 工具菜单栏中的【Nios II】→【Flash Programmer】，出现如图所示的界面。注意，在这一步开始之前，开发板上需要预先下载了sof文件，否则界面会报错。

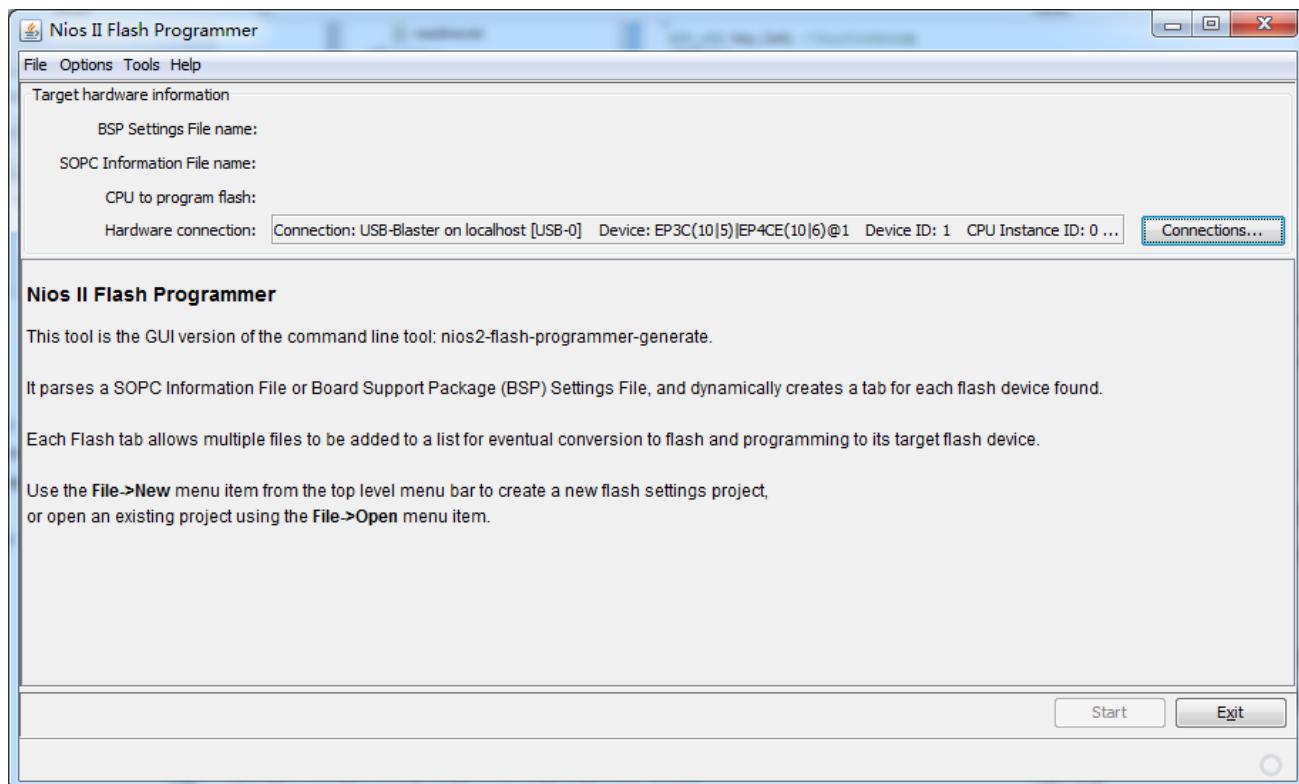


图 3.4.8 Nios II Flash Programmer界面

接下来点击Nios II Flash Programmer 工具菜单栏中的【File】→【New...】，出现如下图所示界面：

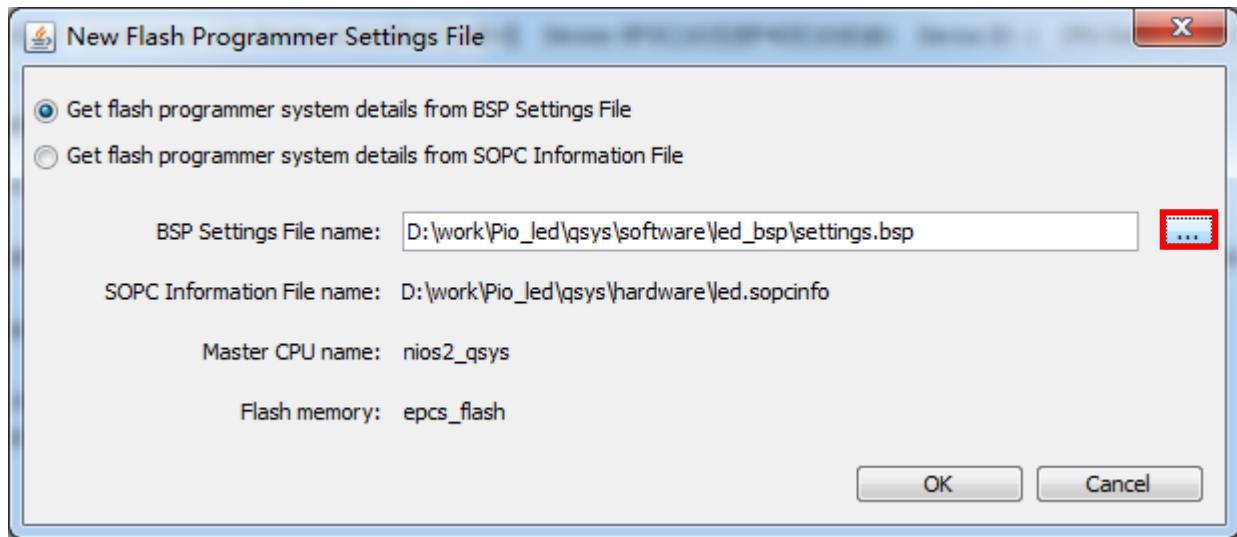


图 3.4.9 flash配置文件界面图

这里配置有两种方法，一种方法是使用settings.bsp 文件，另一种是使用led.sopcinfo 文件。这两种方法都可以完成配置任务，这里就讲解第一种方法的具体操作步骤。在选中bsp

文件后，单击【OK】，出现如图所示界面：

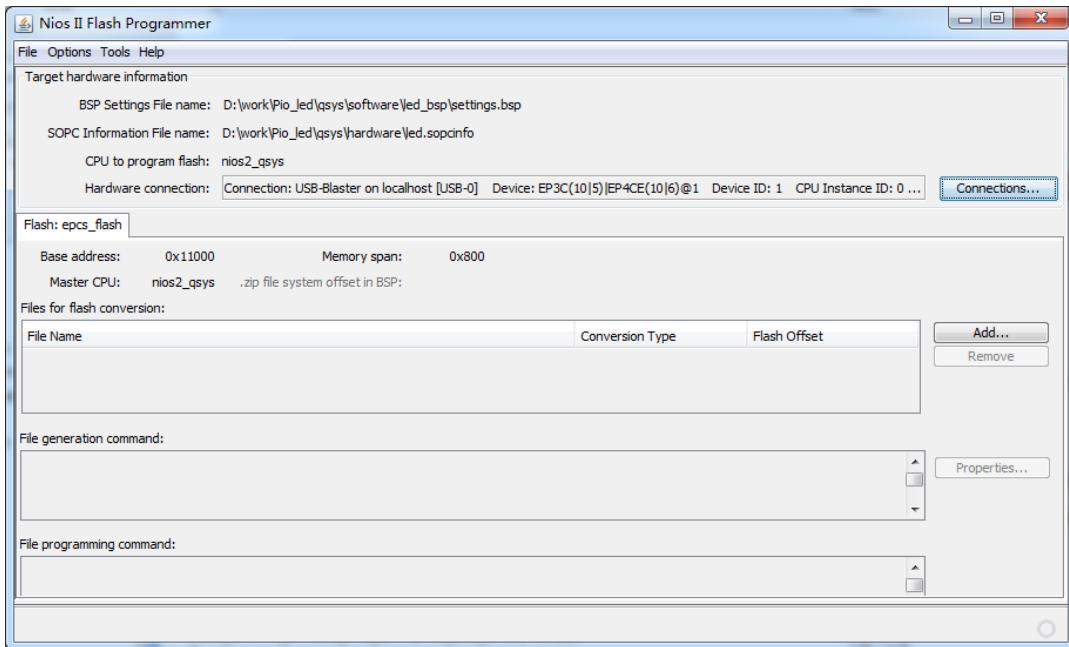


图 3.4.10 flash添加下载文件界面图

单击【Add】，先将Qsys_First.sof 文件添加进去，然后再将Qsys_First.elf 文件添加进去。

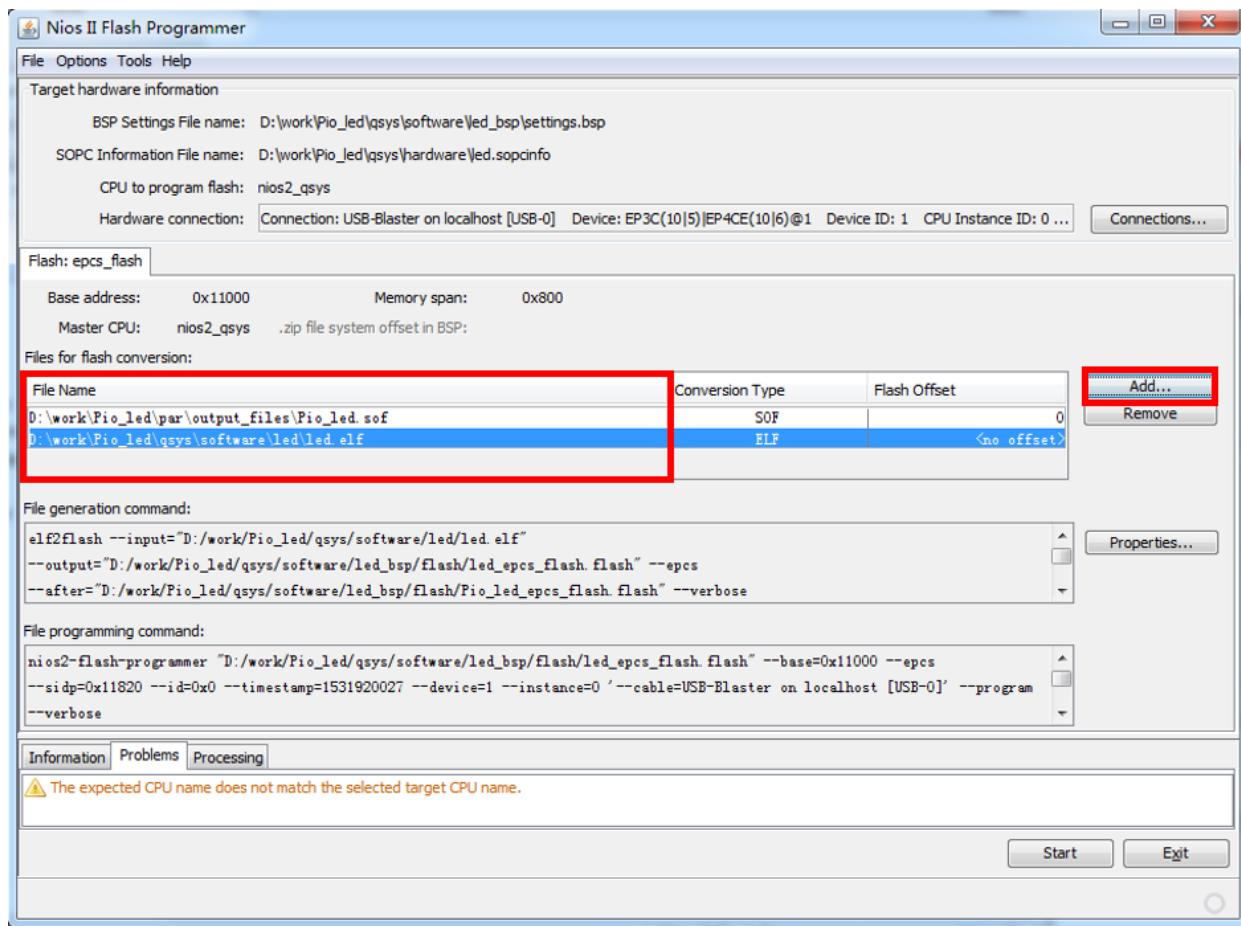


图 3.4.11 flash添加下载文件界面图

Sof文件在Pio_led\par\output_files目录下，elf文件在Pio_led\qsys\software\led目录下。文件添加完成后如图 3.4.12所示，此时就可以点击【Start】了。稍等一会，就会将两个文件固化到开发板的FPGA芯片中，就可以实现上电自启动的功能了。下载完成了出现下图所示界面：

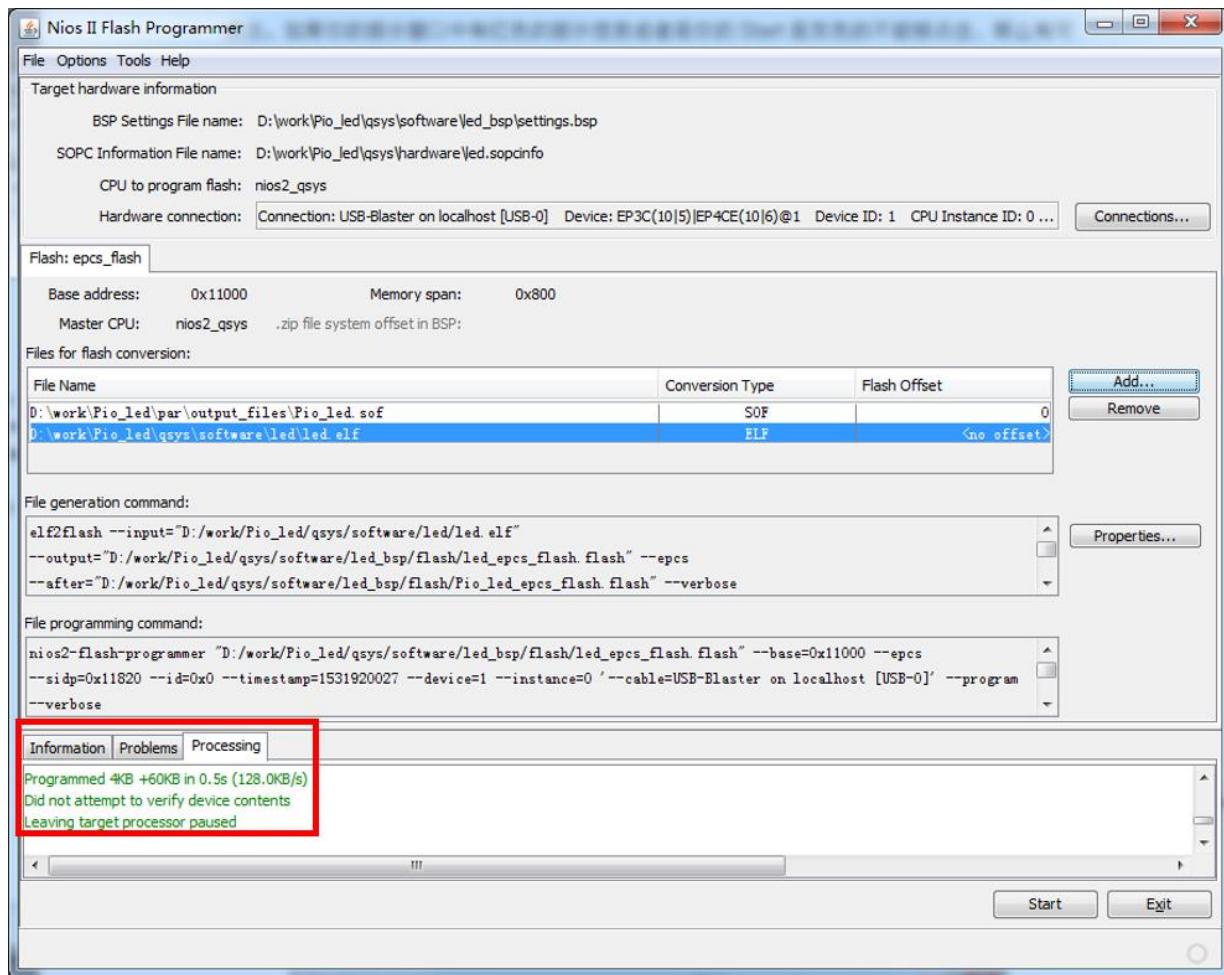


图 3.4.12 flash下载完成界面图

至此，固化程序的任务已经完成。

3.5 下载验证

由于我们已经将代码固化到开发板的芯片中，不需要再下载程序了。现在只需先断开电源，然后打开电源，分别按下 4 个按键，测试按键是否能分别控制 LED 的亮灭。若前面的操作无误的话，现在整个实验任务就已经实现了。

第四章 PIO中断

在前面的章节里，我们把PIO IP核的寄存器相关内容进行了详细的讲解，并以按键控制LED灯为例，提到了PIO IP核的一个比较简单的应用。在这一章中，我们将对该IP核的一个十分重要的功能——中断，进行详细地说明。由于这个功能的应用比较广泛，希望大家能够熟练掌握相关的内容。

本章包括以下几个部分：

- 4.1 简介
- 4.2 实验任务
- 4.3 硬件设计
- 4.4 软件设计
- 4.5 下载验证

4.1 简介

在上一章我们给大家介绍了如何使用PIO IP核与外设通信，在这里我们将通过PIO IP核的中断功能来讲解如何在Nios II开发过程中使用中断。

中断与 IRQ

PIO IP核有一个与中断相关功能——IRQ，它的全称是interrupt request。单从英文名来看，能够得知它是中断请求。那么什么是中断呢？在计算机相关方面，中断是指：在计算机的运行过程中，由于系统内部或外部或现行程序本身出现紧急事件，计算机立即自动停止正在运行的程序，并开始处理新的程序（中断程序），在处理完中断程序后返回原来的程序接着运行，这一完整的过程。

PIO IP 核产生 IRQ 的宏观条件

在PIO IP核的输入端口，发生了硬件预先设置的中断触发事件，PIO IP核就会输出IRQ。

PIO IP 核产生 IRQ 的实际条件

触发IRQ的方式有两种：一种是电平触发（Level），当输入端口为高电平时触发；另一种是边沿触发（Edge），当输入端口出现了上升沿或下降沿或者双沿中的一种（取决于硬件的配置）。接下来讲解产生IRQ需要的具体条件。

1. 产生电平触发 IRQ 需要的条件

(1) 首先 PIO 核的端口需要是输入型的端口，也就是需要在配置界面的 Direction 一项中选择 Input 选项；

(2) 在 PIO IP 核中加入 IRQ 这个功能，也就是在 Interrupt 项中，选中 Generate IRQ 这一选项；

(3) 选中电平触发选项，即在 IRQ type (IRQ 类型) 中选中 Level。

(4) 使能中断。在硬件设置完成后，需要在软件中给中断屏蔽寄存器 (interruptmask Register) 写 1。

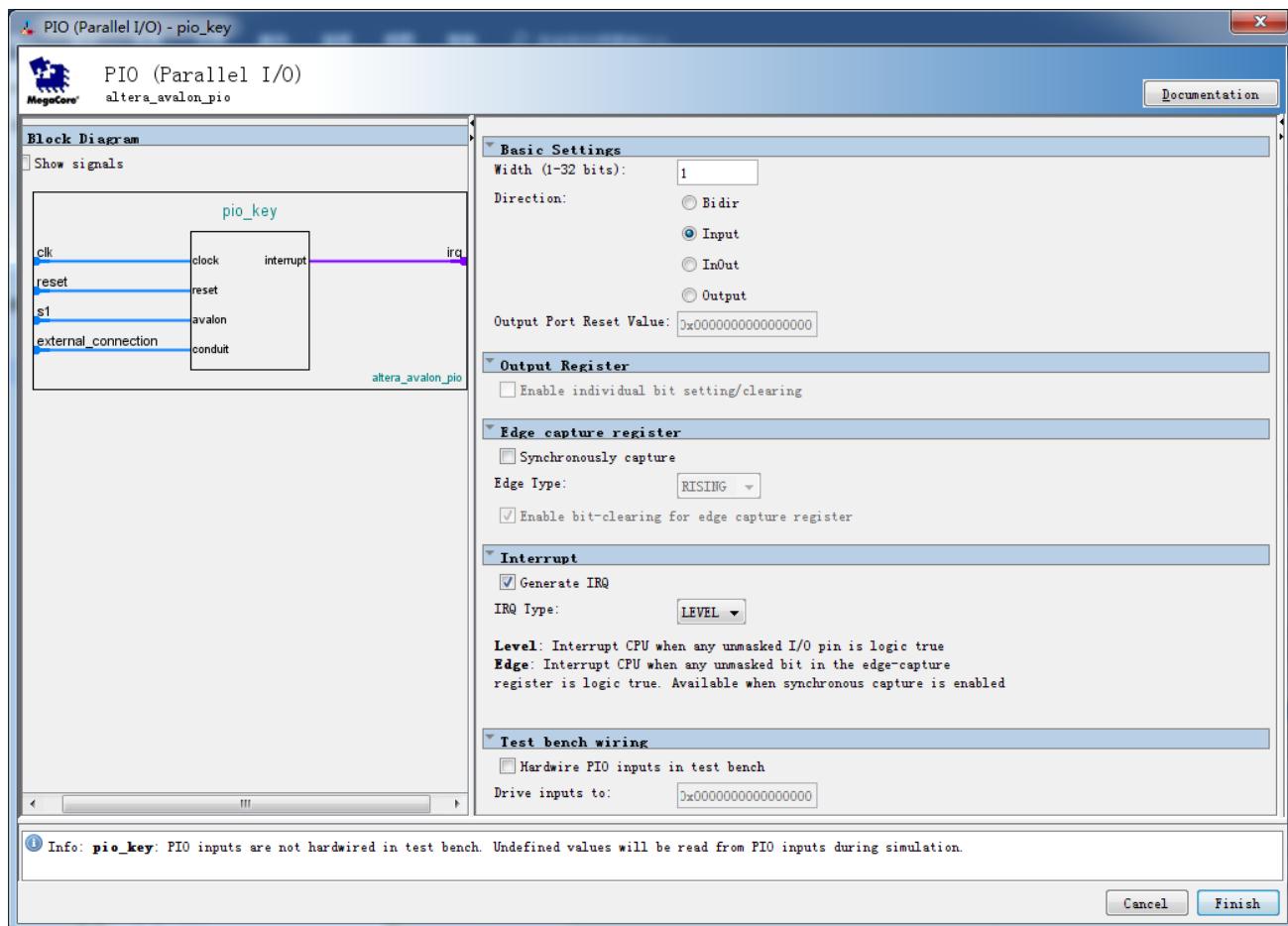


图 4.1.1 电平触发IRQ的配置界面

配置完成后如图 4.1.1 所示，输入端口的位数根据大家的需求而定，图中的设置只是作为一个例子。这里补充说明一下电平触发的 IRQ 工作的方式：

(1) 当硬件设置成电平触发模式时，PIO IP 核的输入端口出现高电平，就能触发 IRQ；若想要低电平触发 IRQ，需要在 Quartus II 工程的顶层，在输入端口前面加一个非门。本章的实验里也用到了这种操作，我们会在后面 4.3 小节硬件设计部分提到这个应用。

(2) 当 PIO IP 核某一个输入端口满足 IRQ 触发条件，PIO IP 核发出 IRQ 后，程序开始处理中断函数。假如此时输入端口一直满足 IRQ 触发条件，那么程序在处理完中断函数后会再次进入中断函数接着运行。直到触发条件不存在了，程序才能在处理完中断函数后回到主函数。

(3) 若程序中有复数个中断函数，且中断函数之间还有优先级的区别，就可能发生这样的情况：主程序正在处理某一个中断函数，这时突然触发一个优先级更高的 IRQ，那么主程序会从当前处理的中断函数跳到优先级更高的中断函数。本章用不到复数的中断函数，所以这里不做详细的讲解。

2.产生边沿触发 IRQ 需要的条件

(1) 首先 PIO 核的端口需要是输入型的端口，也就是需要在配置界面的 Direction 一项中选择 Input 选项；

(2) 在 Edge capture register (边沿捕获寄存器) 栏选中 Synchronously capture 选项。此时可以在 Edge type 一栏选择触发电平，里面的选项有 RISING (上升沿)、FALLING (下降沿)、ANY (任意边沿)。在 Edge capture register 栏还有 enable bit-clearing for edge capture register 选项。选择了 enable bit-clearing for edge capture register 选项后，给 edge capture 寄存器单独的位写 1 清中断；若是没有选择 enable bit-clearing for edge capture register 选项，则是写任意数清中断。每触发一次边沿类型的 IRQ 后，需要给 edge capture 寄存器写 1 清一次中断，否则 PIO IP 核会一直输出 IRQ。

(3) 在 PIO IP 核中加入 IRQ 这个功能，也就是在 Interrupt 项中，选中 Generate IRQ 这一选项；

(4) 选中边沿触发选项，即在 IRQ type (IRQ 类型) 中选中 EDGE。

(5) 使能中断。在硬件设置完成后，需要在软件中给中断屏蔽寄存器 (interruptmask Register) 写 1。

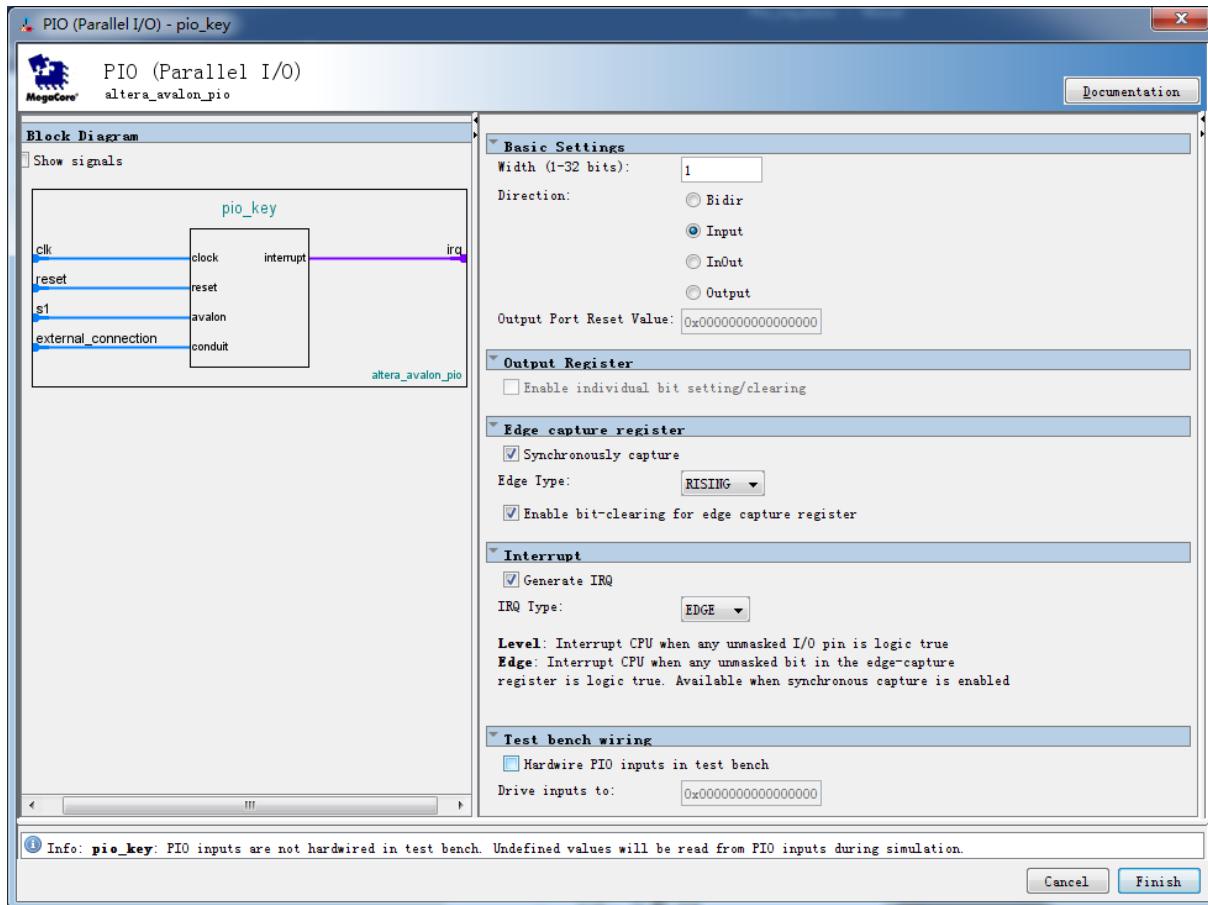


图 4.1.2 边沿触发IRQ的配置界面

上升沿的触发模式配置完成后如图 4.1.2 所示，图中配置只是作为一个例子，实际配置根据大家的需求来定。这里补充说明一下边沿触发的 IRQ 工作的方式：

- (1) 当硬件设置成边沿触发模式时，PIO IP 核的输入端口出现相应的边沿，就能触发 IRQ；
- (2) 当 PIO IP 核某一个输入端口满足 IRQ 触发条件，PIO IP 核发出 IRQ 后，程序开始处理中断函数。在中断函数中一般包含给 edge capture 寄存器写 1 清中断的操作。假如此时输入端口一直满足 IRQ 触发条件，那么程序在处理完中断函数后会再次进入中断函数接着运行。直到触发条件不存在了，程序才能在处理完中断函数后回到主函数。
- (3) 若程序中有多个中断函数，且中断函数之间还有优先级的区别，就可能发生这样的情况：主程序正在处理某一个中断函数，这时突然触发一个优先级更高的 IRQ，那么主程序会从当前处理的中断函数跳到优先级更高的中断函数。

4.2 实验任务

本节实验任务是：在Qsys系统中加入PIO IP核，并通过按键中断控制流水灯的运行状

态。没有按按键时，流水灯正常运行；当按键按下时，4个LED灯全部点亮。

4.3 硬件设计

创建 Quartus II 工程

首先要创建Quartus II工程，工程名为“pio_irq”。

创建 Qsys 系统

实验中要用到的IP 核有：clk（时钟）、nios II（处理器）、onchip_ram（片内存储） 、两个PIO、jtag_uart、sysid_qsys。其中只有PIO IP核和nios II IP核需要稍微讲解一下，其他的IP核都是按照以前的配置方法进行设置，本节就主要讲如何配置PIO IP核和nios II IP核。

在Qsys系统的Library中搜索PIO IP核，双击打开后可以看到如图 4.3.1所示的配置界面。

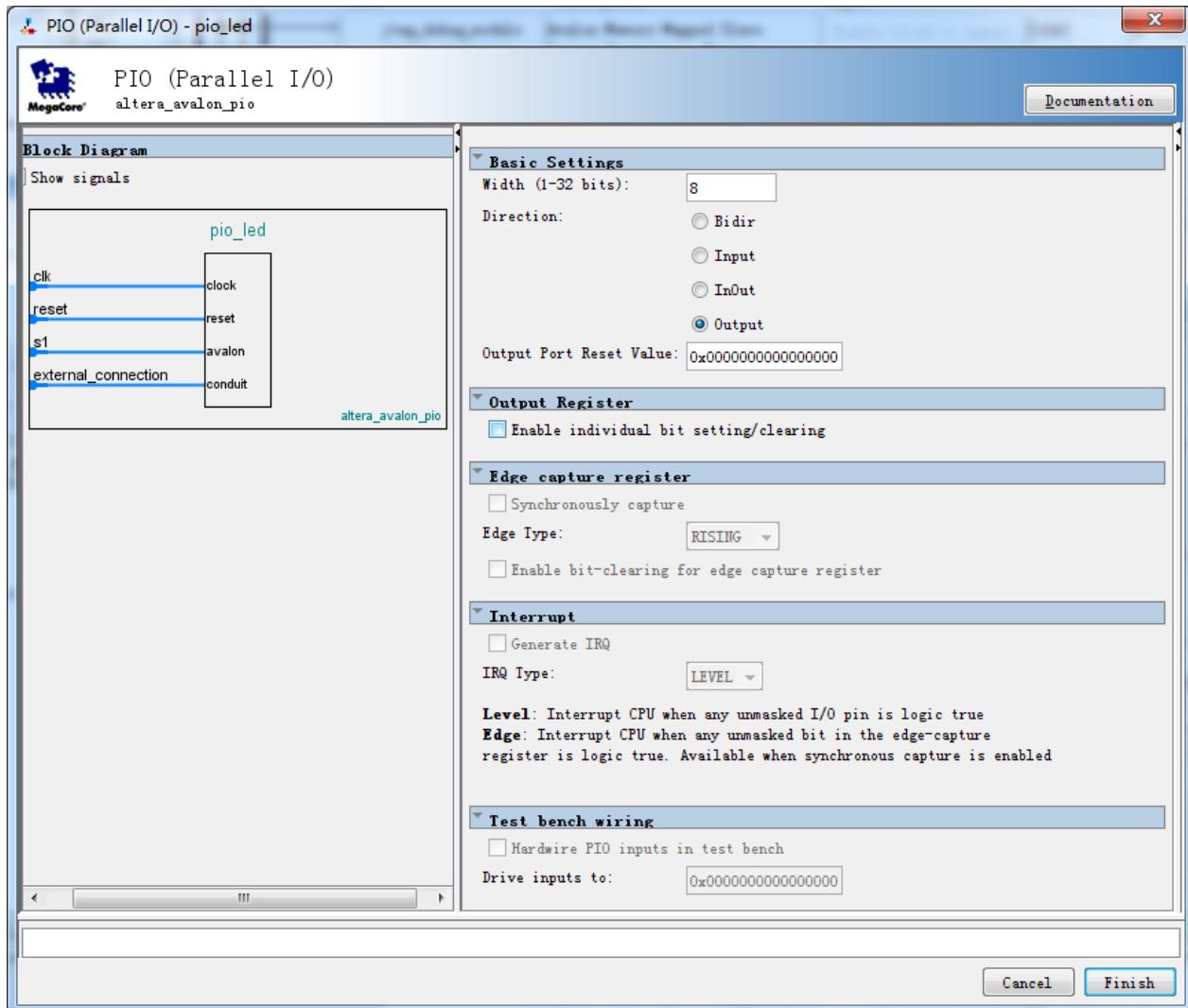


图 4.3.1 PIO IP核配置界面

我们先设置与 LED 灯相关的 PIO。由于开发板上给用户用的 LED 灯共有 4 个，所以在 Basic Settings 里，我们将 PIO 位宽设为 4，方向设为输出(Output)即可。其它设置如 Output Register、Edge capture register 等配置为默认设置，无需更改，点击 Finish 完成。修改完的 PIO IP 核界面如图 4.3.2 所示。

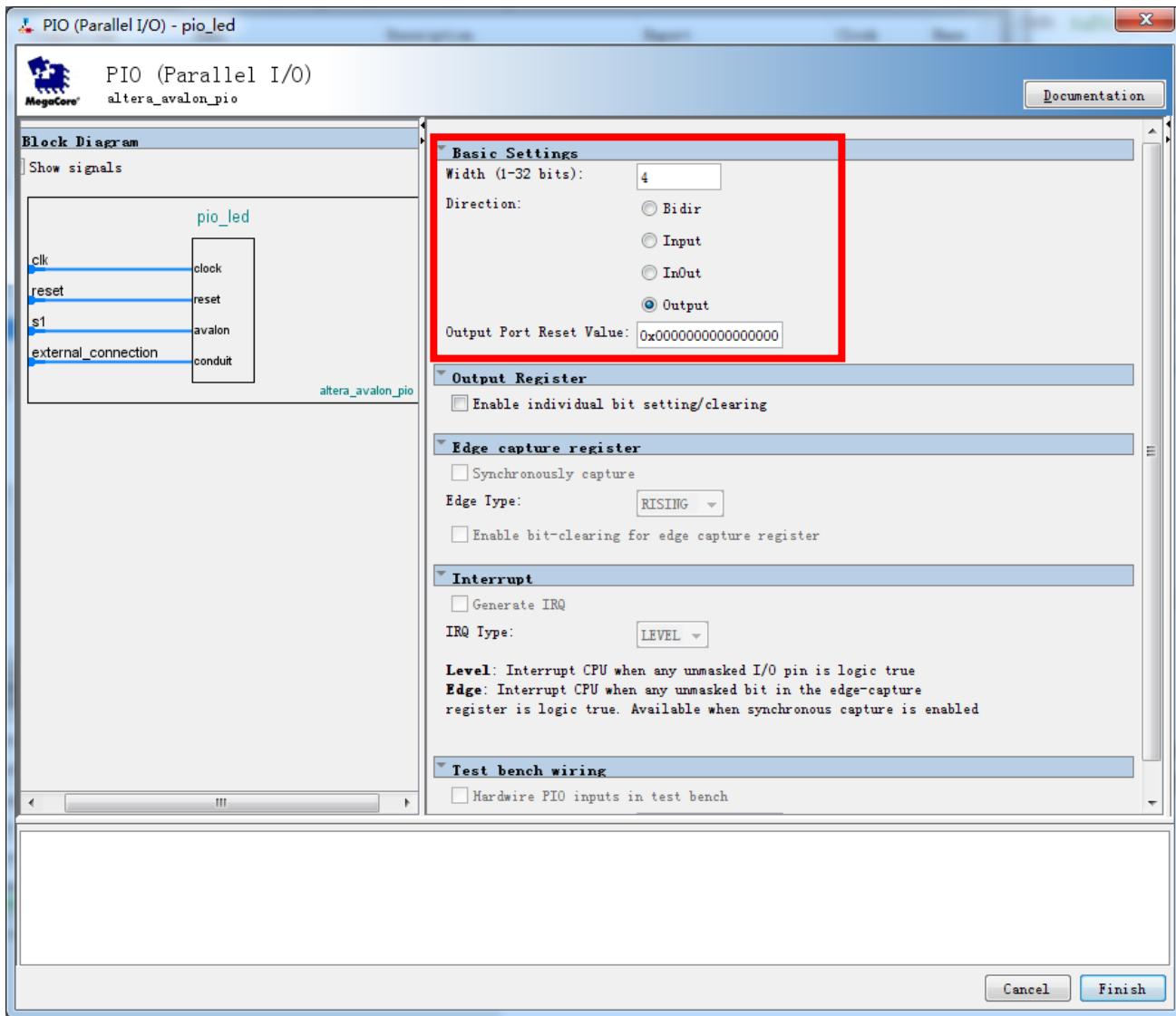


图 4.3.2 与LED灯相关的PIO IP核设置界面

由于我们的实验任务是使用按键中断控制流水灯的运行状况，所以还要添加一个与按键相关的 PIO IP 核。由于实验任务要求按键中断时，LED 灯全亮，其他情况下 LED 灯处于流水灯的运行状态。所以，根据任务要求以及简介中提到的两种触发 IRQ 的方式，我们可以猜想：按键按下时，程序一直在处理中断函数（让 LED 灯全亮），没有按下按键时，程序在处理主函数（运行流水灯函数）。那么要用到的 IRQ 触发模式是电平类型的，并且这里只需要一个按键就能满足实验要求。因此我们在 Basic Settings 里，将 PIO 位宽设为 1，方向设为输入（Input），将 Interrupt 一栏中的 Generate IRQ 选项选中，此时 IRQ type 自动选择为 LEVEL（电平）触发模式。修改完的 PIO IP 核界面如图 4.3.3 所示。

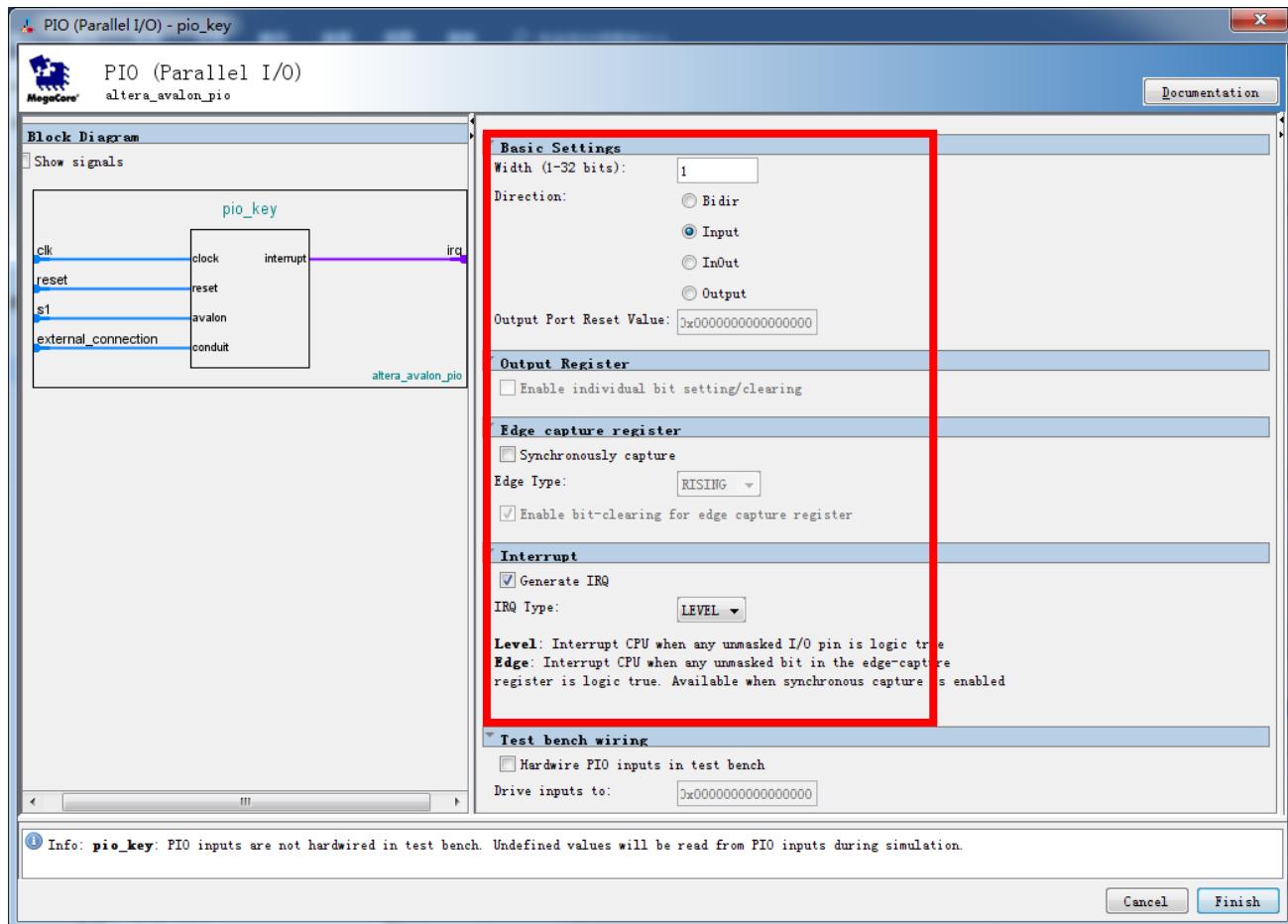


图 4.3.3 与按键相关的PIO IP核设置界面

我们打开nios II IP核配置界面，因为这里只用了onchip_ram IP核存储代码和指令，所以需要对相关的设置进行修改。如图 4.3.4所示，在Reset Vector处将Reset vector memory处的选项选为onchip_ram，同时在Exception Vector处也将Exception vector memory处的选项选为onchip_ram。需要注意的是，onchip_ram IP核之后，nios II IP核的Reset Vector和Exception Vector选项中才会出现onchip_ram选项。

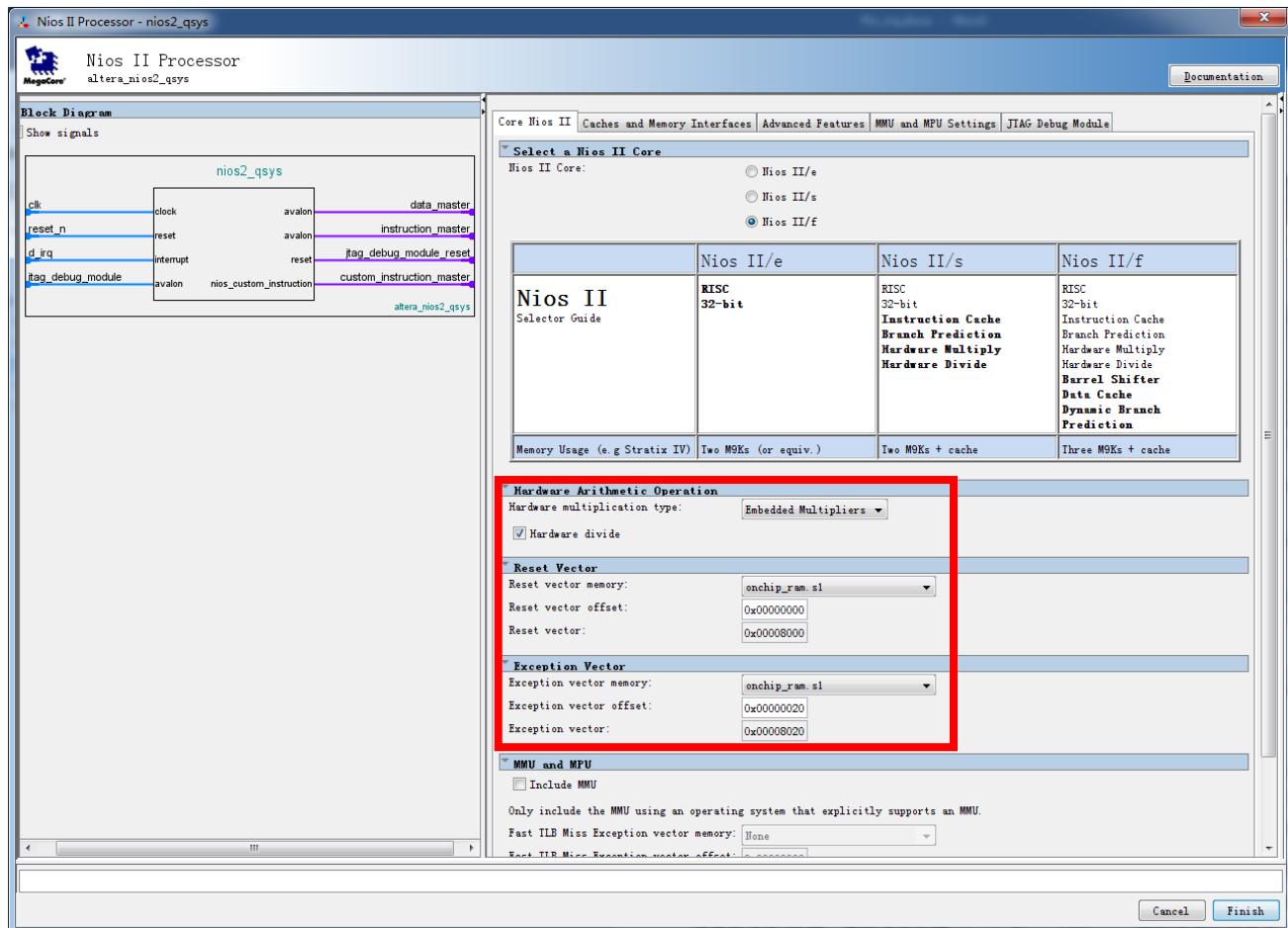


图 4.3.4 nios II IP核设置界面

添加完 IP 核后就可以开始连线，大家若是不熟悉怎么连线，可以照着下面完成的 Qsys 系统界面图连。需要注意的是，要将 PIO IP 核的端口引出来，如图 4.3.5 所示。引出端口的方法是双击图 4.3.5 中 IP 核的 Export 一栏的红框位置，然后修改名称，按下 Enter 键即可。

然后，点击 System→Assign Base Addresses 让系统自动分配地址，这里最好把 onchip_ram 的地址锁住，这是因为这个 IP 核里存储着指令，最好不要让其地址发生变动。锁住地址的方法是先点击 IP 核，然后点击右键→Lock Base Address。我们还可以将各个 IP 核的名称修改一下。最后就是生成系统了，操作可以按照“Hello, World”文档里的进行。

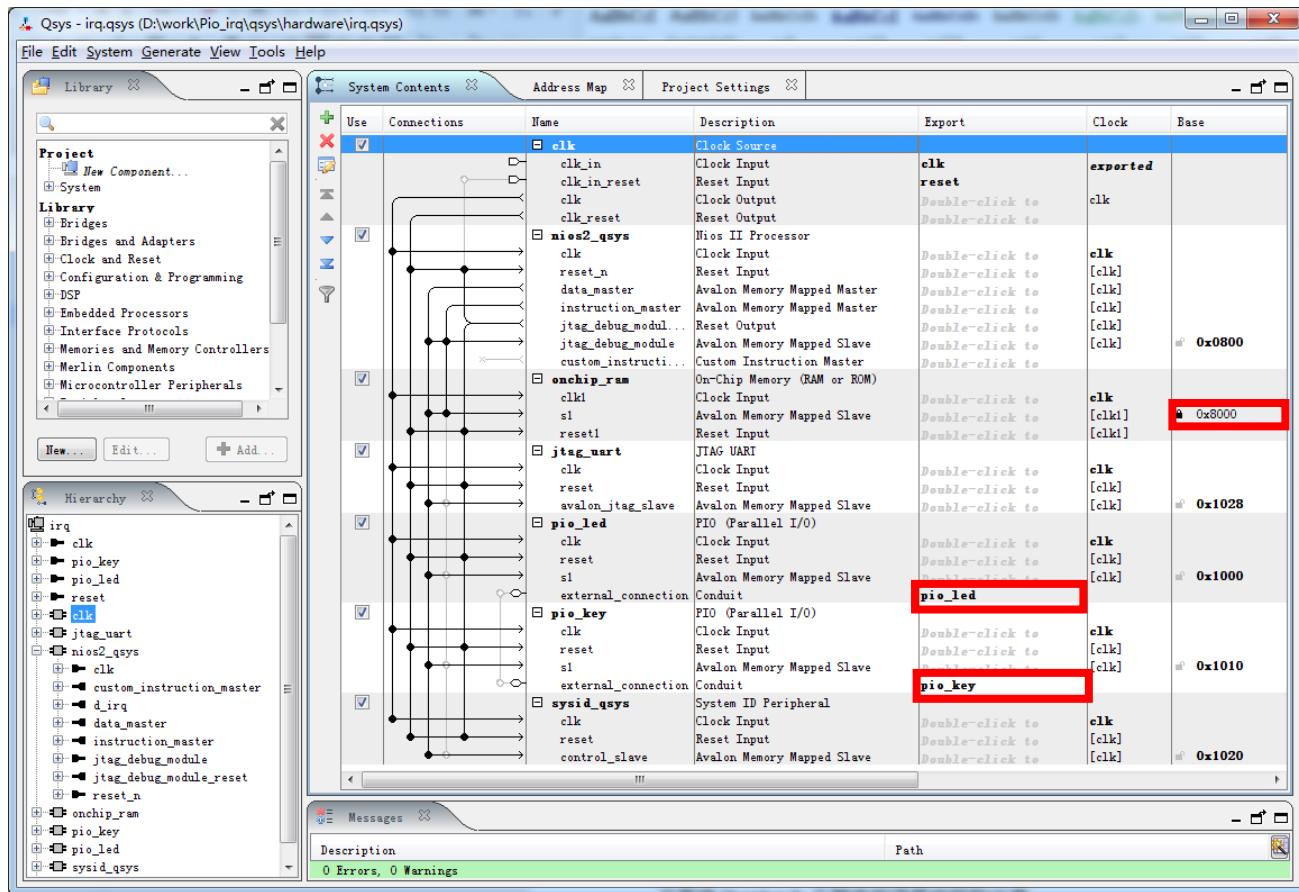


图 4.3.5 nios II IP核设置界面

集成 Qsys 系统

这一步依然可以按照“Hello, World”文档里的操作进行。

下面将Quartus II 工程中的顶层代码贴出来。

```

1 module Pio_irq(
2     input  sys_clk      ,
3     input  sys_rst_n   ,
4
5     input  key          ,
6     output [3:0] led
7 );
8
9 //wire define
10 wire clk_100M;
```

```
11
12 //锁相环
13 pll pll_inst (
14 .inclk0 (sys_clk),
15 .c0      (clk_100M)
16 );
17
18 //例化 Qsys 系统
19 irq u0 (
20   .clk_clk      (clk_100M),          // clk.clk
21   .reset_reset_n (sys_rst_n),        // reset.reset_n
22   .pio_led_export (led),            // pio_led.export
23   .pio_key_export (~key)           // pio_key.export
24 );
25
26 endmodule
```

在代码的第23行，我们将key信号的值取反后，再送给跟key相关的PIO IP核。简介中曾提到过这个问题：开发板上的按键按下时，输出低电平。而PIO IP核配置成电平触发IRQ模式时，触发条件是高电平。所以才有了代码第23行的取反操作，这个取反操作实际上相当于在PIO IP核前加了一个反相器。

编译和下载

这时，我们便能够进行编译查错了，我们可以通过Quartus II 软件菜单栏中的【Processing】→【Start Compilation】来进行编译，也可以通过快捷栏中的快捷键进行编译。

接下来我们就需要进行配置IO，分配管脚。首先，点击Quartus II 软件菜单栏中的【Assignment】→【Device】，然后我们在Device 界面中找到【Device and Pin Options...】进入图 4.3.6所示页面配置IO。将未使用引脚设置为高阻输入（As input tri-state），这样上电后FPGA 的所有不使用引脚都将进入高阻抗状态。

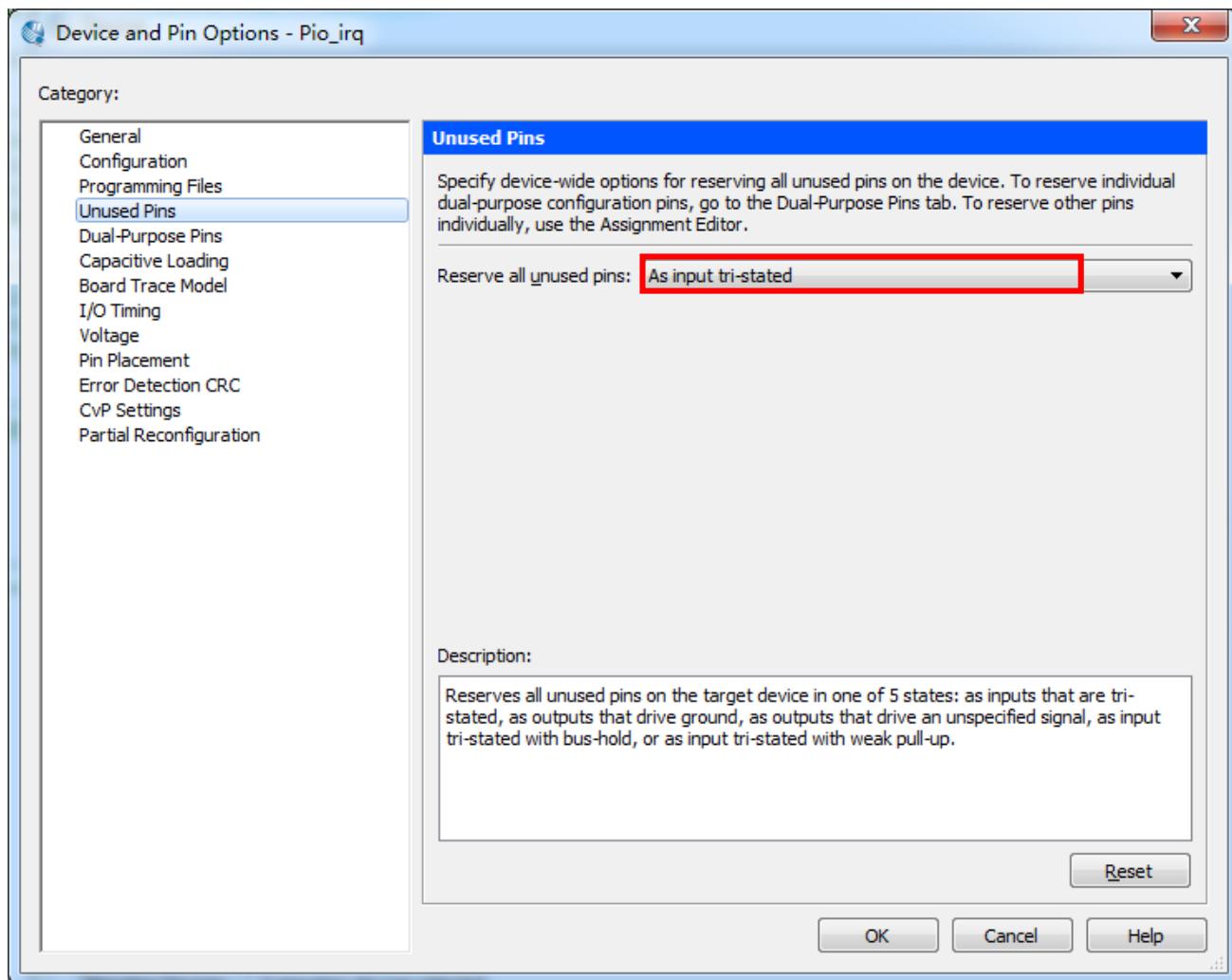


图 4.3.6 未使用引脚设置界面

接下来，将一些 IO 设置成普通 IO，通过双击红框位置，将一个个 Value 的值修改过来。如图 4.3.7 所示。

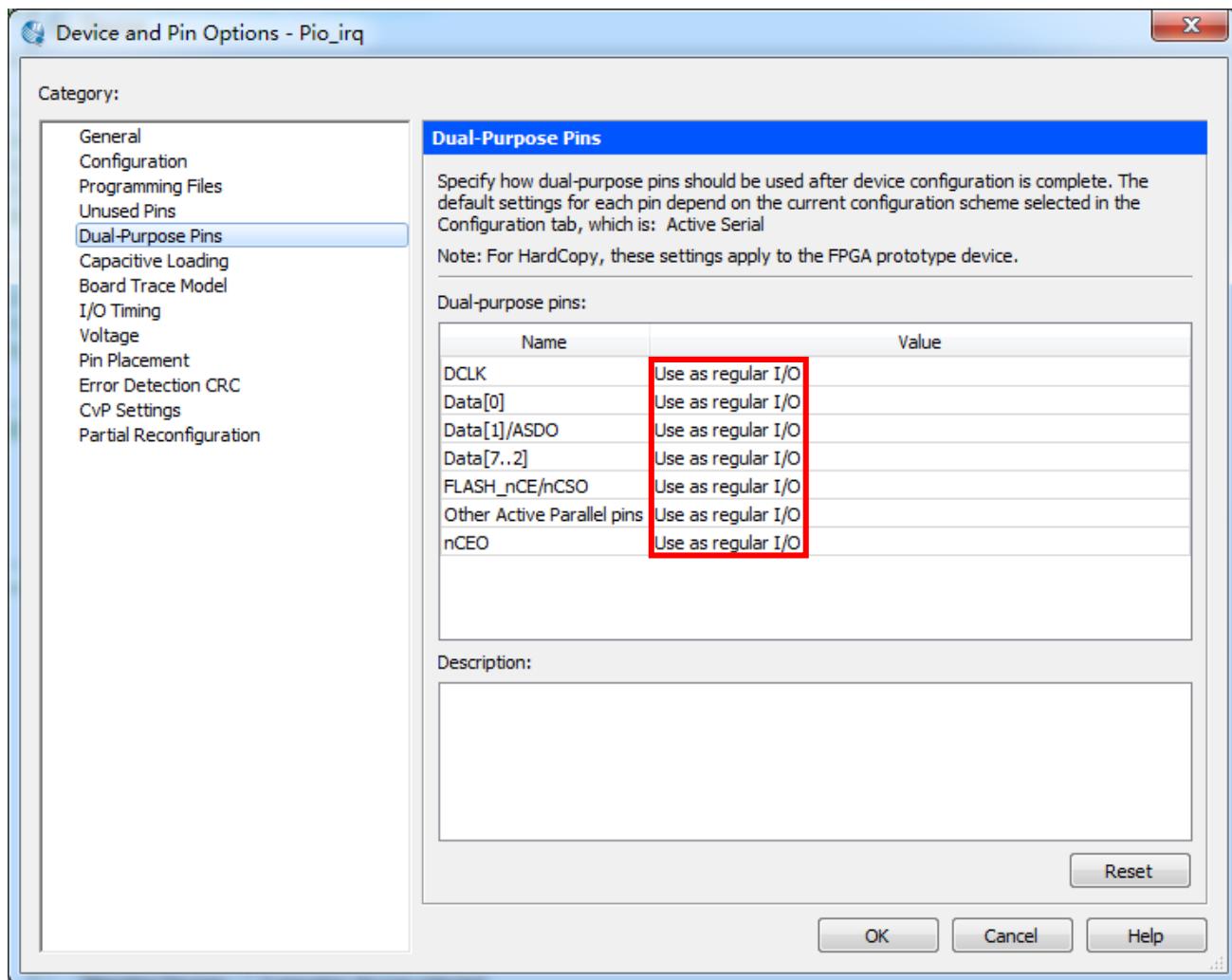


图 4.3.7 IO设置界面

我们通过 Quartus II 软件菜单栏中的【Assignments】→【Pin Planner】选项分配引脚，如图 4.3.8 所示。

in key	Input	PIN_E15	6	B6_N0	PIN_E15	2.5 V (default)
out led[3]	Output	PIN_F9	7	B7_N0	PIN_F9	2.5 V (default)
out led[2]	Output	PIN_E10	7	B7_N0	PIN_E10	2.5 V (default)
out led[1]	Output	PIN_C11	7	B7_N0	PIN_C11	2.5 V (default)
out led[0]	Output	PIN_D11	7	B7_N0	PIN_D11	2.5 V (default)
in sys_clk	Input	PIN_E1	1	B1_N0	PIN_E1	2.5 V (default)
in sys_rst_n	Input	PIN_M1	2	B2_N0	PIN_M1	2.5 V (default)

图 4.3.8 引脚分配界面

最后我们再进行一次全编译，成功编译硬件系统后，将产生用于配置 FPGA 的 Pio_irq.sof 文件。下面我们就来说明一下将.sof 文件下载到 FPGA 目标器件的步骤。

(1) 将下载器一端连接电脑，另一端与开发板上对应端口连接，最后连接电源线并打开电源开关。

接下来我们下载程序。工程打开后通过点击工具栏中的“Programmer”图标打开下载界面，通过“Add File”按钮选择 pio_irq\par\output_files 目录下的“pio_irq.sof”文件。开发板电源打开后，在程序下载界面点击“Hardware Setup”，在弹出的对话框中选择当前的硬件连接为“USB-Blaster[USB-0]”。然后点击“Start”将工程编译完成后得到的 sof 文件下载到开发板中。

至此，硬件部分设计完成，下面开始基于 Nios II SBT for Eclipse 的软件部分的设计。

4.4 软件设计

我们通过 Quartus II 软件菜单栏中的【Tools】→【Nios II SBT for Eclipse】，来启动 Nios II SBT for Eclipse 软件。打开 Nios II SBT for Eclipse 软件后，会弹出 Workspace Launcher 页面。我们这里将工作空间设置为 Pio_irq\qsys 路径下的 software 文件夹，如图 4.4.1 所示。

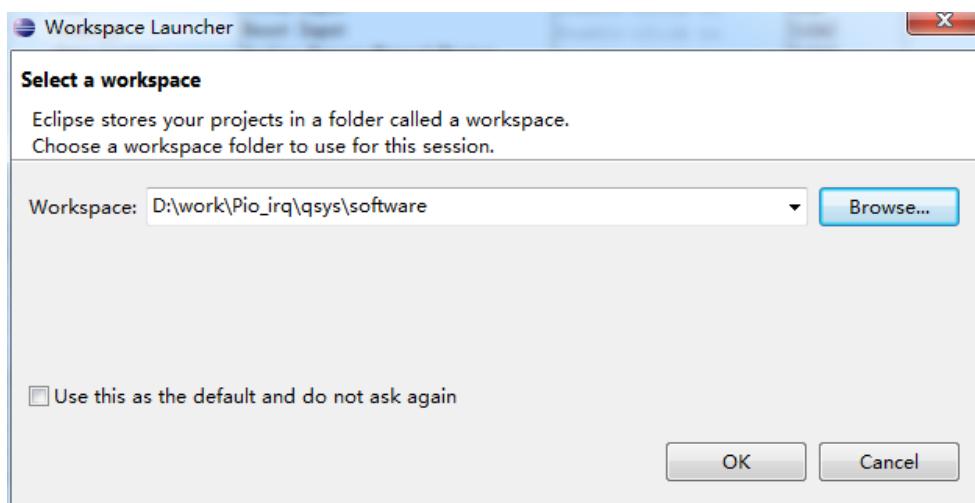


图 4.4.1 设置工作空间

设置好工作空间后，我们点击【OK】进入 Nios II SBT for Eclipse 软件主界面中，在该页面我们通过单击菜单栏中的【File】→【New】→【Nios II Application and BSP from Template】，来新建工程，如图 4.4.2 所示。

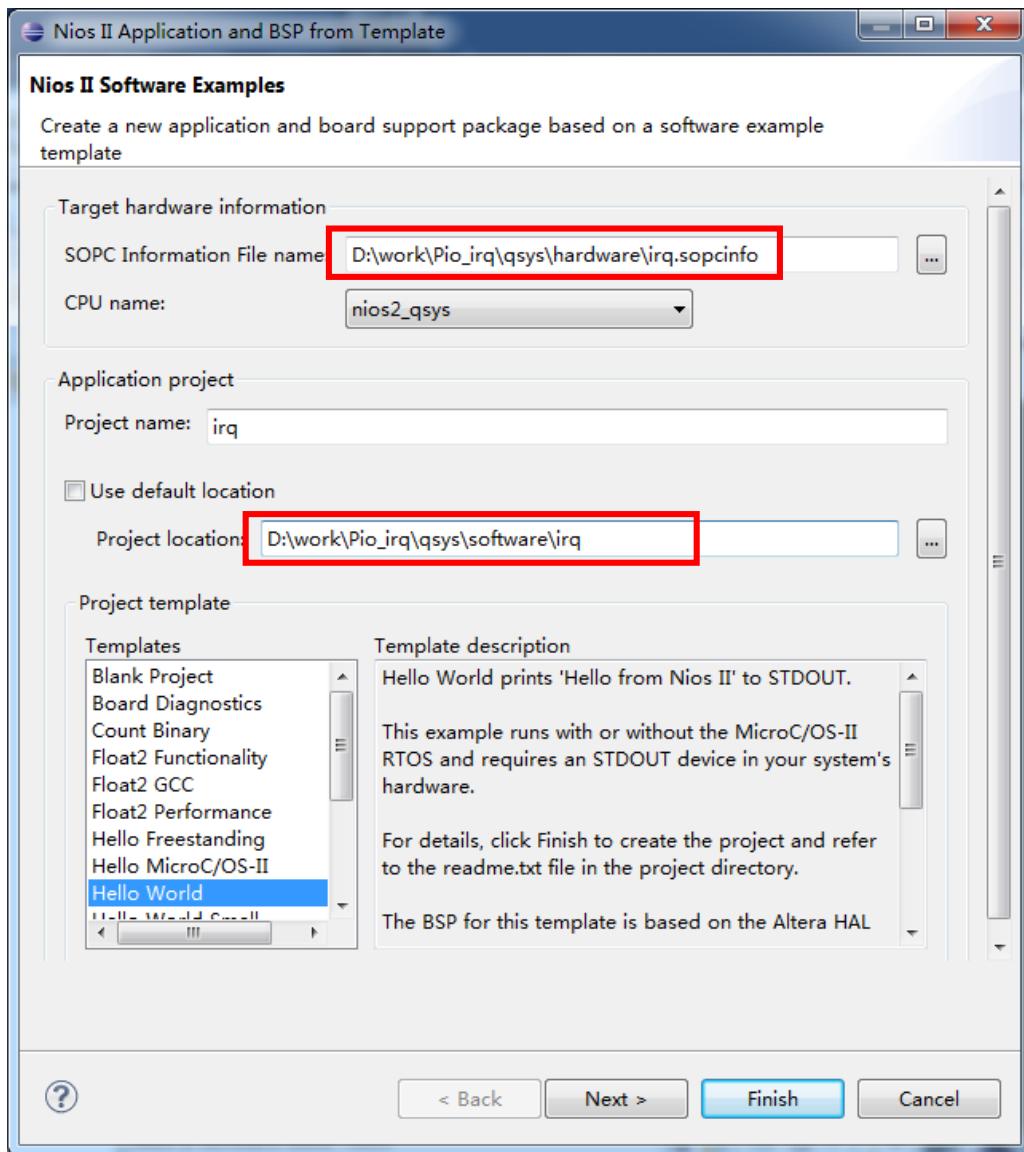


图 4.4.2 新建Nios II SBT for Eclipse 工程

单击【...】按钮来选择 Pio_irq\qsys\hardware 下的 Pio_irq.sopcinfo 文件，即指向当前硬件设计系统。Nios II SBT for Eclipse 软件会自动识别 Qsys 系统中 CPU 的名称，所以 CPU name 一项会自动生成。接下来，要给 Nios II SBT for Eclipse 软件中的工程命名，这里的名称没有特殊要求，我们这里名为 irq。然后将工程存放的位置修改为 Pio_irq\qsys\software\irq。注意不要漏掉了“\irq”，不然生成系统的时候会报错。最后我们来看下 Project template 窗口，该窗口中陈列的都是已经设计好的软件工程。我们可以从中选择一个，作为自己的工程的模板来使用。当然也可以选择 Bland Project（空白工程），就需要自己写所有的代码。这里我们选择的是 Hello World 模板工程，然后我们在它的基础上进行修改，这样比空白工程更加方便。

设置完工程后，直接点击【Finish】完成工程创建。然后，在 Nios II SBT for Eclipse 软件的

左侧 Project Explorer 窗口中有两个工程：irq 和 irq_bsp。其中 irq 是 C/C++ 应用工程，而 irq_bsp 是描述 Qsys 系统硬件细节的系统库。打开 irq 工程里的 hello_world.c 文件，出现如图 4.4.3 所示的图。

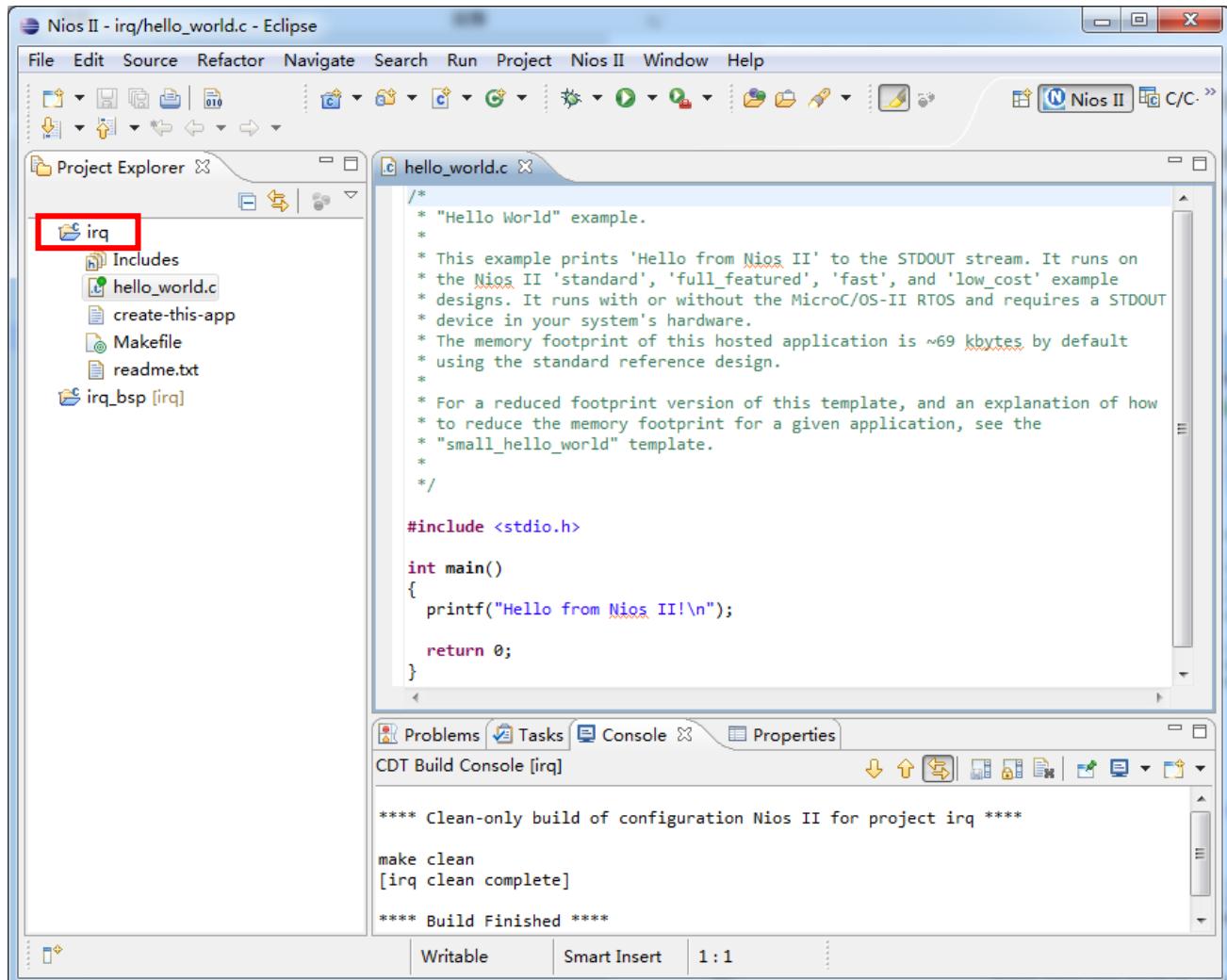


图 4.4.3 hello_world 工程代码图

由代码可知，下载程序到开发板后会在窗口上输出“Hello from Nios II!”。我们在这里要先验证之前创建的 Qsys 系统是否能正常工作。验证方法是先编译 irq 工程，然后将工程模板程序下载到开发板上，看是否能正常运行。方法是右键 irq 工程，点击 build project。Console 窗口会报出下图这样的错误。

```

d:/program/quartus/nios2eds/bin/gnu/h-i686-mingw32/bin/..../lib/gcc/nios2-elf/4.7.3/...
/.../.../.../H-i686-mingw32/nios2-elf/bin/ld.exe: irq.elf section `.text' will not
fit in region `onchip_ram'
d:/program/quartus/nios2eds/bin/gnu/h-i686-mingw32/bin/..../lib/gcc/nios2-elf/4.7.3/...
/.../.../.../H-i686-mingw32/nios2-elf/bin/ld.exe: address 0xedfc of irq.elf section
`_rwddata' is not within region `onchip_ram'
d:/program/quartus/nios2eds/bin/gnu/h-i686-mingw32/bin/..../lib/gcc/nios2-elf/4.7.3/...
/.../.../.../H-i686-mingw32/nios2-elf/bin/ld.exe: address 0x10b24 of irq.elf section
`.bss' is not within region `onchip_ram'
d:/program/quartus/nios2eds/bin/gnu/h-i686-mingw32/bin/..../lib/gcc/nios2-elf/4.7.3/...
/.../.../.../H-i686-mingw32/nios2-elf/bin/ld.exe: address 0xedfc of irq.elf section
`_rwddata' is not within region `onchip_ram'
d:/program/quartus/nios2eds/bin/gnu/h-i686-mingw32/bin/..../lib/gcc/nios2-elf/4.7.3/...
/.../.../.../H-i686-mingw32/nios2-elf/bin/ld.exe: address 0x10b24 of irq.elf section
`.bss' is not within region `onchip_ram'
d:/program/quartus/nios2eds/bin/gnu/h-i686-mingw32/bin/..../lib/gcc/nios2-elf/4.7.3/...
/.../.../.../H-i686-mingw32/nios2-elf/bin/ld.exe: region `onchip_ram' overflowed by
15140 bytes
collect2.exe: error: ld returned 1 exit status
make: *** [irq.elf] Error 1

```

图 4.4.4 编译工程后的console窗口

在“Hello,World”实验中也出现了类似的错误，其原因是存储空间不够。所以这里也要像“Hello,World”实验里一样，优化一下代码。方法是一样的，大家可以照着操作，这里就不再赘述了。

优化完代码之后，再编译一次 led 工程，会出现以下的界面。这表示编译通过，可以将程序下载到开发板上了。

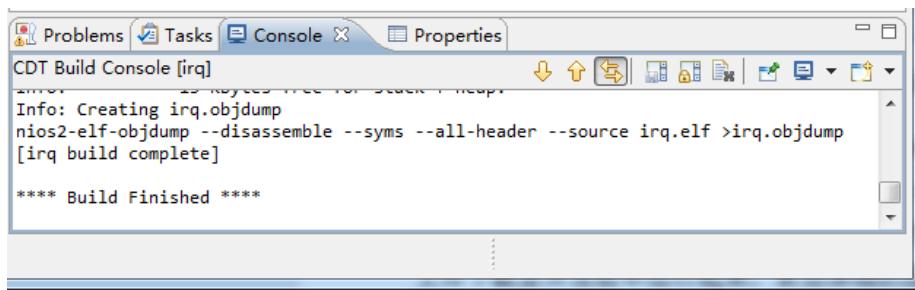


图 4.4.5 编译通过后的console窗口

这时大家点击【Run As】→【Nios II Hardware】，然后点击【Target Connection】标签，然后在 Target Connection 窗口中点击【Refresh Connections】按钮后。这时软件便会自动识别我们开发板上的 Qsys 系统，并显示 Qsys 系统的相关信息。我们接着点击【Run】，软件会把 irq.elf 文件下载至开发板中运行起来。更加详细的图和文字描述，可以在“Hello,World”实验的下载验证部分查看。

这时，若之前创建的 Qsys 系统无误，代码下载完成后在 Nios II console 窗口会显示“Hello

from Nios II!” 字符，如下图所示。

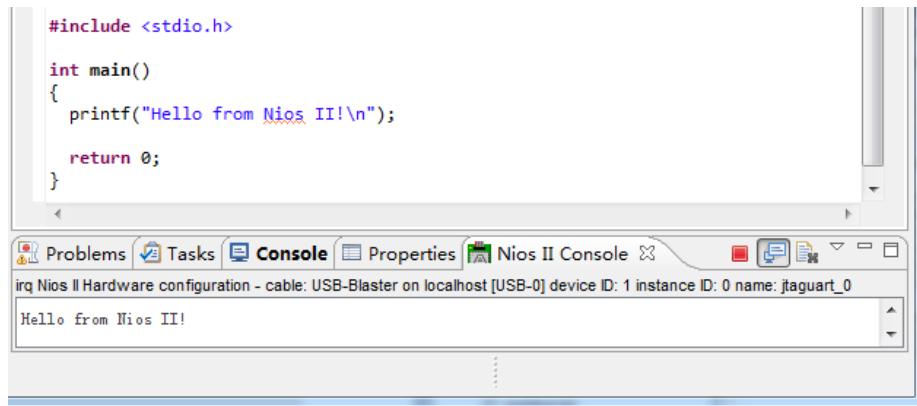


图 4.4.6 下载代码后的console窗口

验证完 Qsys 系统是否能正常运行之后，我们就可以开始软件部分的设计了。这时只需要在当前的代码窗口修改代码就可以了。代码如下所示。

```
1 #include <stdio.h>
2 #include "system.h"           //系统头文件
3 #include "altera_avalon_pio_regs.h" //pio 寄存器头文件
4 #include "sys/alt_irq.h"       //中断头文件
5 #include "unistd.h"           //延迟头文件
6
7 void IRQ_Init();             //中断初始化函数
8 void IRQ_Key Interrupts();   //中断服务子程序
9
10 int main(void)
11 {
12     alt_u8 led, i;           //有符号 8 位整数
13     IRQ_Init();              //初始化 PIO 中断
14     IOWR_ALTERA_AVALON_PIO_DATA(PIO_LED_BASE, 0x0F); //初始化 LED 全亮
15     //流水灯
16     while(1)
17     {
18         for(i=0;i<4;i++)
19         {
20             led = 1 << i;          //从低位到高位移位
21         }
22     }
23 }
```

```
21     IOWR_ALTERA_AVALON_PIO_DATA(PIO_LED_BASE, led);
22     usleep(100000);           // 延迟一段时间
23 }
24 }
25 }
26
27
28 void IRQ_Init()
29 {
30     IOWR_ALTERA_AVALON_PIO_IRQ_MASK(PIO_KEY_BASE, 0x01); // 使能中断
31     // 注册 ISR
32     alt_ic_isr_register(
33         PIO_KEY_IRQ_INTERRUPT_CONTROLLER_ID, // 中断控制器标号, 从 system.h 复制
34         PIO_KEY_IRQ,                      // 硬件中断号, 从 system.h 复制
35         IRQ_Key_Interrupts,             // 中断服务子函数
36         0x0,                            // 指向与设备驱动实例相关的数据结构体
37         0x0);                          // flags, 保留未用
38 }
39
40 void IRQ_Key_Interrupts()
41 {
42     IOWR_ALTERA_AVALON_PIO_DATA(PIO_LED_BASE, 0x0f); // 中断函数, 让 LED 全亮
43 }
```

代码的第 10 行至第 25 行是主函数。主函数中第 13 行代码完成了初始化中断的功能，完整的初始化中断函数在代码的第 28 行到第 38 行。在初始化中断的时候，需要使能中断，也就是给中断屏蔽寄存器写 1，这个操作在代码的第 30 行完成。在代码的第 16 行至第 25 行，在主函数中完成了流水灯的功能。也就是每间隔一段时间，给 PIO IP 核的输出端口，按照一定的规律进行赋值。而这个 PIO IP 核是用来驱动 LED 灯发光的。代码的第 40 行至第 43 行是一个中断函数，它实现的功能是让 4 个 LED 灯一起发光。

联系整个工程，我们可以得出：在一般情况下，程序一直在运行流水灯程序。当我们按下按键且不松开时会一直触发 IRQ，程序因此会一直运行中断函数，让 4 个 LED 灯一起发光。当我们松开按键后，程序则会回到主函数，接着运行流水灯程序。

修改完代码的窗口如下所示：

The screenshot shows the Eclipse IDE interface for the 'Nios II - irq/hello_world.c' project. The Project Explorer view on the left lists files such as 'Binaries', 'Includes', 'obj', 'hello_world.c', 'irq.elf - [alteranios2/le]', 'create-this-app', 'irq.map', 'irq.objdump', 'Makefile', 'readme.txt', and 'irq_bsp'. The main editor window displays the 'hello_world.c' source code. The code includes headers for stdio.h, system.h, altera_avalon_pio_regs.h, sys/alt_irq.h, and unistd.h. It defines two functions: IRQ_Init() and IRQ_Key Interrupts(). The main() function initializes the PIO and sets up a loop to toggle eight LEDs. A screenshot of the Eclipse interface with the code editor open.

```
#include <stdio.h>
#include "system.h"
#include "altera_avalon_pio_regs.h" //pio 等芯片头文件
#include "sys/alt_irq.h"           //中断头文件
#include "unistd.h"                //延时头文件

void IRQ_Init();                  //中断初始化函数
void IRQ_Key Interrupts();       //中断服务子程序

int main(void)
{
    alt_u8 led,i;                //有符号8位整数
    IRQ_Init();                   //初始化PIO 中断
    IOWR_ALTERA_AVALON_PIO_DATA(PIO_LED_BASE, 0x0F); //初始化LED 全亮
    //流水灯
    while(1)
    {
        for(i=0;i<4;i++)
        {
            led = 1 << i;          //从低位到高位移位
            IOWR_ALTERA_AVALON_PIO_DATA(PIO_LED_BASE, led);
            usleep(100000);         //延迟一段时间
        }
    }

    void IRQ_Init()
    {
        IOWR_ALTERA_AVALON_PIO IRQ MASK(PIO KEY BASE, 0x01); // 使能中断
    }
}
```

图 4.4.7 修改后的工程代码图

代码修改完成后，大家记得要点一下快捷菜单中的【Save】，或者菜单栏中的【File】→【Save】，来保存修改后的程序。

4.5 下载验证

现在可以编译 irq 工程了。右键 irq 工程，点击 build project。稍等片刻，Console 窗口显示的内容如下图所示，这表示工程编译成功。

```
Info: 13 KBytes free for stack + heap.  
Info: Creating irq.objdump  
nios2-elf-objdump --disassemble --syms --all-header --source irq.elf >irq.objdump  
[irq build complete]  
***** Build Finished *****
```

图 4.5.1 编译工程后的console窗口图

这时大家右键 irq 工程，点击【Run As】→【Nios II Hardware】，代码就被下载到开发板上了，此时开发板上的 4 个 LED 以流水灯的方式依次点亮，循环往复。然后按下 key1，流水灯停止运行，4 个 LED 灯保持常亮；松开按键后，4 个 LED 继续以流水灯的方式运行。到这里，我们本次利用 PIO 中断来控制 LED 灯运行状态的实验就结束了。

第五章 串口IP核

我们在让 Altera FPGA 上的嵌入式系统和外部设备进行通信方式的时候，通常会用到拥有 Avalon 接口的通用异步收发传输器——UART IP 核。UART 核实现了 RS-232 通讯协议，并使得大家可以设置串口通信相关的波特率、奇偶校验位、停止位和数据位，以及可选的 RTS/CTS 流控制信号等参数。

本章包括以下几个部分：

5.1 简介

5.2 实验任务

5.3 硬件设计

5.4 软件设计

5.5 下载验证

5.1 简介

UART 核提供了一个 Avalon 存储器映射 (Avalon-MM) 的接口，这个接口使得 Avalon-MM 的主要周边设备（例如 Nios II 处理器）通过读和写数据、控制寄存器，就能实现和 UART 核通信的任务。所以在本章的简介部分，我们会对常用的寄存器进行详细的介绍。

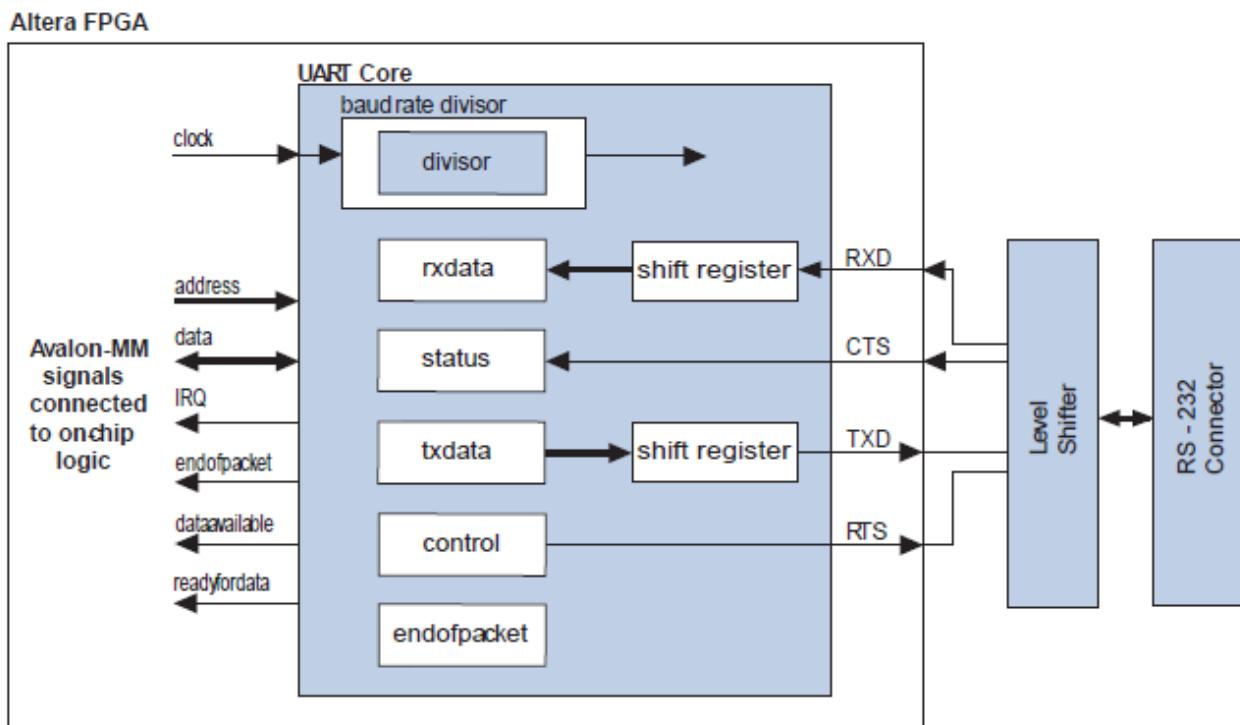


图 5.1.1 一个典型系统中的UART核

如图 5.1.1 所示为一个典型系统中的 UART 核，它有两个用户可见的部分：

- 可通过 Avalon-MM 从接口访问的寄存器
- RXD, TXD, CTS, and RTS 等 RS-232 接口信号

图中可以看到，在 RXD 端口与 rxdata 寄存器之间、TXD 与 txdata 寄存器之间存在着移位寄存器。

RS232 的工作原理可以在前面的 verilog 文档部分查阅。在 Qsys 篇，我们只讲解 UART 核的工作原理，以及该核的使用方法。前面提到，我们是通过读、写相关的寄存器，实现的串口通信功能。那么接下来，我们将着重讲解主要的寄存器——状态寄存器、控制寄存器、数据寄存器。其中状态、控制寄存器与配置 IRQ (中断请求) 相关，需要大家掌握它们的使用方法和工作原理。

(1) 寄存器相关简介

下图为 UART 核的寄存器信息表，其中数据寄存器(rxdata、txdata)、状态寄存器(status)、控制寄存器(control) 是重点，括号部分的注释内容在表的下部。

表 5.1-1 UART核寄存器

Offset	Register Name	R/W	Description/Register Bits																	
			15:13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	rxdata	RO	Reserved								(1)	(1)	Receive Data							
1	txdata	WO	Reserved								(1)	(1)	Transmit Data							
2	status (2)	RW	Reserve d	eop	cts	dcts	(1)	e	rrdy	trdy	tmt	toe	roe	brk	fe	pe				
3	control	RW	Reserve d	ieo p	rts	idcts	trb k	ie	irrd y	itrd y	itm t	itoe	iroe	ibrk	ife	ipe				
4	divisor (3)	RW	Baud Rate Divisor																	
5	endof-packet (3)	RW	Reserved								(1)	(1)	End-of-Packet Value							

(1) 这些位可能不存在，取决于数据位宽的硬件选项。如果它们不存在，读出的值为 0。且如果对它们进行写操作，则没有意义。

(2) 给 status (状态) 寄存器写 0 将清零 dcts, e, toe, roe, brk, fe, and pe 等位。

(3) 这些寄存器可能不存在，取决于硬件配置选项。如果寄存器不存在，对它进行读操作会返回未定义的值，且进行写操作则无意义。

rxdata 寄存器

rxdata 寄存器用于存储 RXD 输入引脚接收的数据。当一个新的数据被 RXD 输入引脚完全接受后，会被传输并存储到 rxdata 寄存器，此时 status 寄存器（状态寄存器）的 rrdy 位会被置 1。当 rxdata 寄存器中的值被读了之后，status 寄存器中的 rrdy 位会被清零。当 rrdy 位为 1 时，又有一个新的字符传输给 rxdata 寄存器，则会产生溢出错误，状态寄存器的 ROE 位被置 1。不管前一个字符是否被读出，新接收到的字符总是会被自动保存到 rxdata 寄存器。另外，对 rxdata 寄存器进行写操作无意义。

txdata 寄存器

Avalon 主控制器把要发送的字符写入到 txdata 寄存器中。当一个字符写入 txdata 寄存时，状态寄存器（status 寄存器）的 TRDY 位会被置 0；当字符从 txdata 寄存器传输到发送移位寄存器时，状态寄存器（status 寄存器）的 TRDY 位被置为 1。当 TRDY 位为 0 时，将字符写入 txdata 寄存器的结果是未定义的。读 txdata 返回未定义的值。

status 寄存器（状态寄存器）

状态寄存器（status 寄存器）由可以反应 UART 核状态情况的独立位组成。每个状态位和对应的控制寄存器中，能使能中断的位相联系。任何时候都可以读取状态寄存器。且读操作不会改变寄存器任何位的值。给状态寄存器写 0，会清零 DCTS、E、TOE、ROE、BRK、FE 和 PE 位。状态寄存器的位相关信息在下表中列出：

表 5.1-2 状态寄存器

位	名称	操作	描述
0	PE	读/清除	Parity Error。当读取的校验位不正确时会发生校验错误且 PE 位置 1。PE 位为 1 时，需要通过写 status 寄存器将其清零。当 PE 位置 1 时，读取 rxdata 寄存器会产生未定义值。如果没有选中 Parity 硬件选项，UART 核则没有奇偶校验，并且读 PE 位总是 0
1	FE	读/清除	Framing Error。当接收器没能检测到正确的停止位时，会产生帧错误。当核接收到错误的停止位时，FE 位置 1。FE 位置 1 后，直到给 status 寄存器写数据才会将其清零。当 FE 位置 1 时，读 rxdata 寄存器会返回未定义的值
2	BRK	读/清除	Break Detect。当 RXD 引脚连续保持低电平（逻辑 0）的时间大于一个完整字符的时间（数据位，加上起始、停止和校验位）时，接收逻辑才会检测到间断。当检测到间断时，BRK 位置 1。BRK 位为 1 时，需要通过写 status 寄存器才能将其清零
3	ROE	读/清除	Receive Overrun Error。在读取前一个字符前（也就是 RRDY 位为 1 时），将一个新的字符传输给 rxdata 寄存器，会产生接收溢出错误。此时，ROE 位置 1，且接收数据寄存器内前一个接收

			字符被新的接收字符改写。BRK 位为 1 时，需要通过写状态寄存器将其清零
4	TOE	读/清除	Transmit Overrun Error。在前一个字符传输到移位寄存器前，将新字符写入 txdata 寄存器（TRDY 位为 0 时），会产生溢出错误。此时 TOE 位置 1。TOE 位置 1 后，直到给 status 寄存器写 1 才会将其清零
5	TMT	读	Transmit Empty。TMT 位提示发送移位寄存器的当前状态。当移位寄存器正将字符从 TXD 引脚输出时，TMT 位设为 0。当移位寄存器处于空闲时（没有要输出的字符），TMT 位为 1。Avalon-MM 主控制器可以通过检查 TMT 位来确定发送是否完成
6	TRDY	读	Transmit Ready。TRDY 位提示 txdata 寄存器的当前状态。当 txdata 寄存器为空时，可以接收新字符且 TRDY 位为 1。当 txdata 寄存器为满状态时，TRDY 位为 0。Avalon-MM 主控制器必须等待 TRDY 位变为 1 后，才能新数据写入 txdata 寄存器中
7	RRDY	读	Receive Character Ready。RRDY 位提示 rxdata 寄存器的当前状态。当 rxdata 寄存器为空时，RRDY 为 0，且还不是读取 rxdata 寄存器的时刻。当新接收的值传输到 rxdata 寄存器时，RRDY 位置 1。读 rxdata 寄存器会将 RRDY 位置零。Avalon -MM 主控制器必须等待 RRDY 位变为 1 后，才能读取 rxdata 寄存器
8	E	读/清除	Exception。E 位指示发生了错误。E 位是 TOE、ROE、BRK、FE 和 PE 位的逻辑或结果。E 位和 control 寄存器相应的中断使能位（IE）给使能/禁止所有错误 IRQ 提供了便利方法。对 status 寄存器进行写操作时，E 位会置 0。
10	DCTS	读/清除	Change in Clear to Send (CTS) signal。只要在 CTS_N 输入端口上检测到逻辑电平跳变时（采样与 Avalon 时钟同步），DCTS 位置 1。该位通过 CTS_N 上的上升沿和下降沿跳变设置。DCTS 位置 1 后，直到给 status 寄存器写入数据将其清零，它的值才会发生变化。

11	CTS	读	Clear to Send (CTS) signal。CTS 位反映 CTS_N 输入的瞬时状态（采样与 Avalon 时钟同步）。CTS_N 输入对发送或接收逻辑没有影响。CTS_N 输入仅会影响 CTS 和 DCTS 位状态，且当 control 寄存器的 idcts 位使能时，有可能产生中断。
12	EOP	读	<p>End of Packet。下面任一事件发生时，EOP 位置 1：</p> <ul style="list-style-type: none"> ● 向 txdata 数据寄存器写入 EOP 字符 ● 从 rxdata 数据寄存器读取 EOP 字符 <p>EOP 字符由 endofpacket 寄存器的内容确定。EOP 位保持为 1，直到给 status 寄存器写入数据将其清零为止。如果没有选中 Include End of Packet Register 硬件选项，读 EOP 位总是为 0。</p>

Control 寄存器（控制寄存器）

控制寄存器由独立的各个位构成，每个位控制 UART 核操作的一个方面。我们在任何时间都可以读控制寄存器。

每一个控制寄存器中与状态寄存器对应的位，都可以使能一个 IRQ。当控制寄存器中与状态寄存器对应的位的值都为 1 时，就会触发一个 IRQ。

表 5.1-3 控制寄存器

位	名称	操作	描述
0	IPE	读写	奇偶校验错误中断使能
1	IFE	读写	帧错误中断使能
2	IBRK	读写	间断检测中断使能
3	IROE	读写	接收溢出错误中断使能
4	ITOE	读写	发送溢出错误中断使能

5	ITMT	读写	发送移位寄存器空中断使能
6	ITRDY	读写	发送准备好中断使能
7	IRRDY	读写	接收准备好中断使能
8	IE	读写	错误中断使能
9	TRBK	读写	发送间断。TRBK 位允许 Avalon-MM 主控制器通过 TXD 输出发送间断字符。当 TRBK 位设为 1 时，TXD 信号强制设为 0。TRBK 位会干扰任何进行中的发送。Avalon 主控制器必须在适当的间断周期后将 TRBK 位设回 0。
10	IDCTS	读写	CTS 信号改变中断使能
11	RTS	读写	请求发送 (RTS) 信号。RTS 位直接连接到 RTS_N 输出。Avalon-MM 主控制器可在任何时候写 RTS 位。RTS 位的值只能影响 RTS_N 输出；它对发送器或接收器逻辑没有影响。由于 RTS_N 输出为负逻辑。当 RTS 位为 1 时，RTS_N 输出逻辑低电平 0。
12	IEOP	读写	结束符中断使能

divisor 寄存器（可选）

divisor 寄存器中的值是用来产生波特率时钟的。有效的波特率由这个公式决定：

$$\text{波特率} = (\text{时钟频率}) / (\text{divisor} + 1)$$

divisor 寄存器是一种可选的硬件功能。如果没有使能 **Baud Rate Can Be Changed By Software**（波特率能被软件改变）这个选项，将不存在 divisor 寄存器。这种情况下，写 divisor 寄存器没有作用，读它返回一个未定义的值。

endofpacket 寄存器（可选）

endofpacket 寄存器中的值决定了可变长度 DMA 传输的结束字符。复位后，默认值是 0，是 ASCII 码中的空值字符 (\0)。endofpacket 寄存器是一种可选的硬件功能。如果没有使能 **Include end-of-packet register**（这个选项，将不存在 endofpacket 寄存器。这种情况下，写

endofpacket 寄存器没有作用，读它返回一个未定义的值。

(2) 中断操作

UART 核输出一个单独的 IRQ 信号给 Avalon-MM 接口。而 Avalon-MM 接口能连到系统中的任意一个主设备，例如 Nios II 处理器。主设备在对 status 寄存器进行读操作之后，才能确定中断产生的原因。

每一个中断在 status 寄存器有相应的位，并在 control 寄存器中有一个使能位。当一个中断发生时，相关的 status 位置 1，直到它被回应了（acknowledged）。当任意一个 status 位置 1，且相应的中断使能位是 1 时触发 IRQ。一个主设备可以通过清零 status 寄存器来回应（acknowledge）IRQ。

复位的时候，所有的中断使能位置 0。因此，核无法触发 IRQ，直到主设备使一个或多个中断使能位置 1。

这里总结一下：一个可以触发的中断与它相应的 status 和 control（中断使能）位有关。

(3) 硬件配置内容

如图为 UART 核的配置界面：

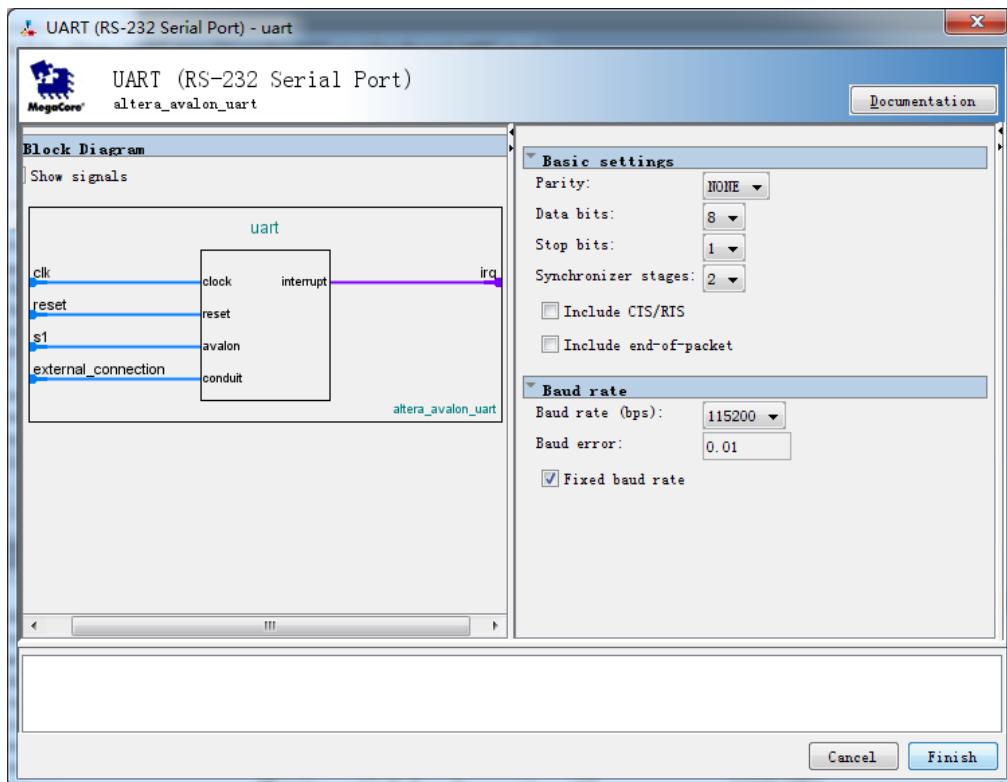


图 5.1.2 UART 核配置界面

从图中可以看出，UART 核有 Basic settings（基础设置）和 Baud rate（波特率）这两个可以设置的部分。

1. Parity（奇偶校验）

Parity 一栏有 None（无）、Even（偶）、Odd（奇）这三个选项。这个设置用来确定 UART 是否发送有奇偶校验的字符，以及它是否期望接收到的有奇偶校验的字符。

当 Parity 设为 None 时，发送逻辑发送不包含校验位的数据，且接收逻辑设定接收到的数据也不包含校验位。status 寄存器中的 PE（校验错误）位无效，其数值始终为 0。当 Parity 设为 Odd 或 Even 时，发送逻辑计算并插入所需的校验位到将要输出的 TXD 数据流，且接收逻辑检验接收到的 RXD 位流中的校验位。如果接收器发现结果不正确，则 status 寄存器中的 PE 位会被置 1。当 Parity 设置为 Even（偶）时，字符中有偶数个 1，则校验位为 0；同样，当 Parity 设为 Odd 时，若字符中有奇数个 1，则校验位为 0。

2. Data bits（数据位）

Data bits 一栏有 7、8、9 这三个可以设置的选项。这个设置决定了 txdata、rxdata、endofpacket 这三个寄存器的位宽。

3. Stop bits（停止位）

Stop bits 一栏有 7、8、9 这三个可以设置的选项。这个设置决定了核在传输每一个字符时，是有 1 还是 2 个停止位。UART 核总是在接收到第一个停止位的时候，就停止接收操作，忽略掉附带的停止位，无论什么设置。

4. Synchronizer Stages

这个设置与寄存器的长度以及亚稳态事件相关，这里一般使用默认设置即可。

5. Include CTS/RTS

选择是否使用串口的“流控”功能，一般很少使用。

6. Include end-of-packet

选择是否设置数据流的结束标志（end-of-packet），一般很少使用。

7. Baud Rate（波特率设置）

UART 内核可实现 RS-232 标准中的任意波特率。波特率可配置为以下方式中的一种：

- 固定的波特率——波特率在系统生成时被确定，且不能通过 Avalon 从控制器端口改变

它的值。

- 可变的波特率——基于 divisor 寄存器中存储的时钟分频值，波特率是可变的。主控制器通过向 divisor 寄存器中写入新值来改变波特率。

波特率的计算依赖于Avalon-MM接口提供的时钟频率。在硬件改变系统时钟频率，却没有重新生成UART核会导致错误的信号。

Baud Rate设置决定了复位后的波特率。Baud Rate选项提供了标准的预设值。也允许用户输入任何非标准波特率。为了实现所需要的波特率，通常根据波特率计算时钟分频系数。波特率与分频系数的关系如下：

$$\text{除数} = \text{int}((\text{时钟频率}) / (\text{波特率}) + 0.5)$$

$$\text{波特率} = (\text{时钟频率}) / (\text{除数} + 1)$$

当选择Fixed baud rate 时，UART 硬件中不再包括divisor寄存器。UART 硬件使用固定的波特率分频系数，且在系统生成后无法改变。这种情况下向地址偏移值4的地方写数据无作用，且读地址偏移值4的地址返回未定义的结果。当不选择Fixed baud rate 时，硬件中会在地址偏移值4生成一个16 位的divisor寄存器。divisor寄存器是可写的，所以可以通过向分频寄存器写入新值来改变波特率。

5.2 实验任务

本节实验任务是：在Qsys系统中通过使用官方UART IP核和PC进行环回，将收到的字符发送给PC。

5.3 硬件设计

创建 Quartus II 工程

首先要创建 Quartus II 工程，工程名为“top_UART”。

创建 Qsys 系统

实验中要用到的 IP 核有：clk(时钟)、nios II(处理器)、onchip_ram(片内存储)、jtag_uart、sysid_qsys、UART。其中只有 UART IP 核讲解一下，其他的 IP 核都是按照以前的配置方法进行设置，本节就讲讲如何配置 UART IP 核。

从 Library 中选择 UART IP 核 (RS-232 Serial Port) 并打开，出现以下界面：

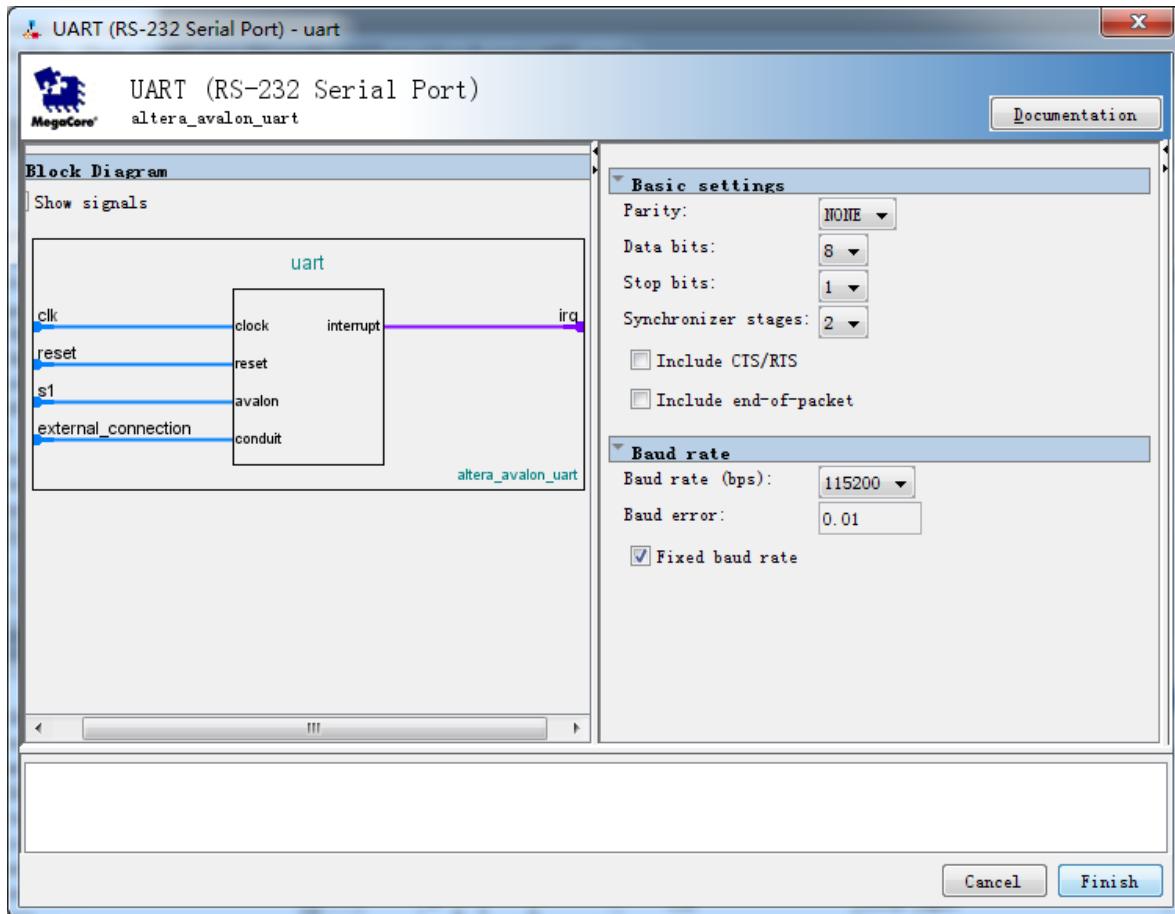


图 5.3.1 UART核的配置界面

这里我们使用默认设置，直接点击【Finish】即可。

然后，我们打开 nios II IP 核配置界面，因为这里只用了 onchip_ram IP 核存储代码和指令，所以需要对相关的设置进行修改。如图 5.3.2 所示，在 Reset Vector 处将 Reset vector memory 处的选项选为 onchip_ram，同时在 Exception Vector 处也将 Exception vector memory 处的选项选为 onchip_ram。需要注意的是，onchip_ram IP 核之后，nios II IP 核的 Reset Vector 和 Exception Vector 选项中才会出现 onchip_ram 选项。

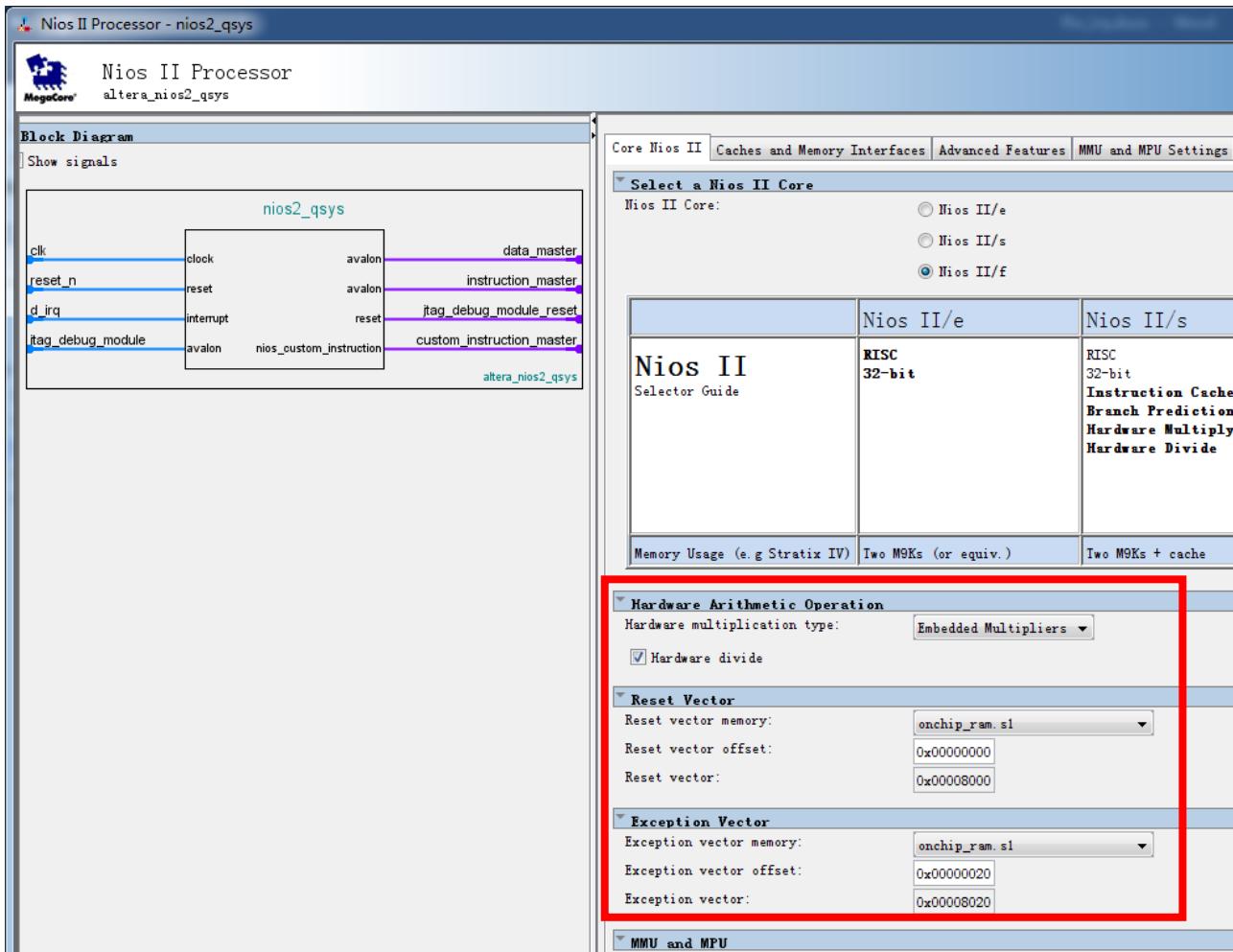


图 5.3.2 nios II IP核设置界面

添加完 IP 核后就可以开始连线，大家若是不熟悉怎么连线，可以照着下面完成的 Qsys 系统界面图连。需要注意的是，要将 UART IP 核的端口引出来，如图 5.3.3 所示。引出端口的方法是双击图 5.3.3 中 IP 核的 Export 一栏的红框位置，然后修改名称，按下 Enter 键即可。

然后，点击 System→Assign Base Addresses 让系统自动分配地址，这里最好把 onchip_ram 的地址锁住，这是因为这个 IP 核里存储着指令，最好不要让其地址发生变动。锁住地址的方法是先点击 IP 核，然后点击右键→Lock Base Address。我们还可以将各个 IP 核的名称修改一下。最后就是生成系统了，操作可以按照“Hello, World”文档里的进行。

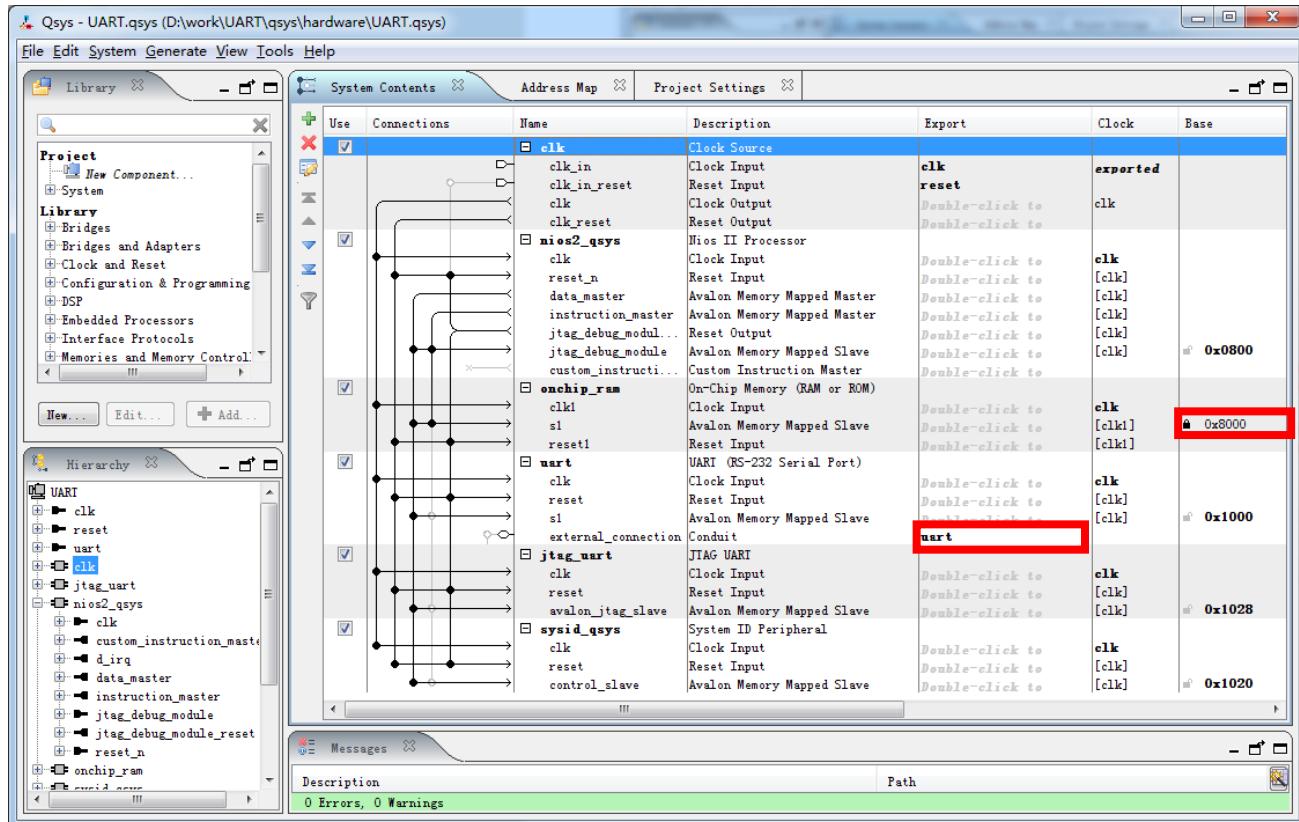


图 5.3.3 nios II IP核设置界面

集成 Qsys 系统

这一步依然可以按照“Hello, World”文档里的操作进行。

下面将 Quartus II 工程中的顶层代码贴出来。

```

1 module top_UART(
2     input          SYS_clk      ,
3     input          SYS_RST_N  ,
4
5 //UART 端口
6     input          rxd         , //UART 接收端
7     output         txd         , //UART 发送端
8 );
9
10 //wire define
11 wire          CLK_100M;    //100mHZ 时钟
12

```

```
13 //例化 pll 模块, 用以产生
14 pll pll_inst (
15   .inclk0      (sys_clk) ,
16   .c0          (clk_100m)
17 );
18
19 //例化 UART 核
20 UART u0 (
21   .clk_clk     (clk_100m),      // clk.clk
22   .reset_reset_n (sys_rst_n),    // reset.reset_n
23   .uart_rxd    (rxd),          // uart.rxd
24   .uart_txd    (txd)           // .txd
25 );
26
27 endmodule
```

编译和下载

这时, 我们便能够进行编译查错了, 我们可以通过 Quartus II 软件菜单栏中的【Processing】→【Start Compilation】来进行编译, 也可以通过快捷栏中的快捷键进行编译。

接下来我们就需要进行配置 IO, 分配管脚。首先, 点击 Quartus II 软件菜单栏中的【Assignment】→【Device】, 然后我们在 Device 界面中找到【Device and Pin Options…】进入图 5.3.4 所示页面配置 IO。将未使用引脚设置为高阻输入 (As input tri-state), 这样上电后 FPGA 的所有不使用引脚都将进入高阻抗状态。

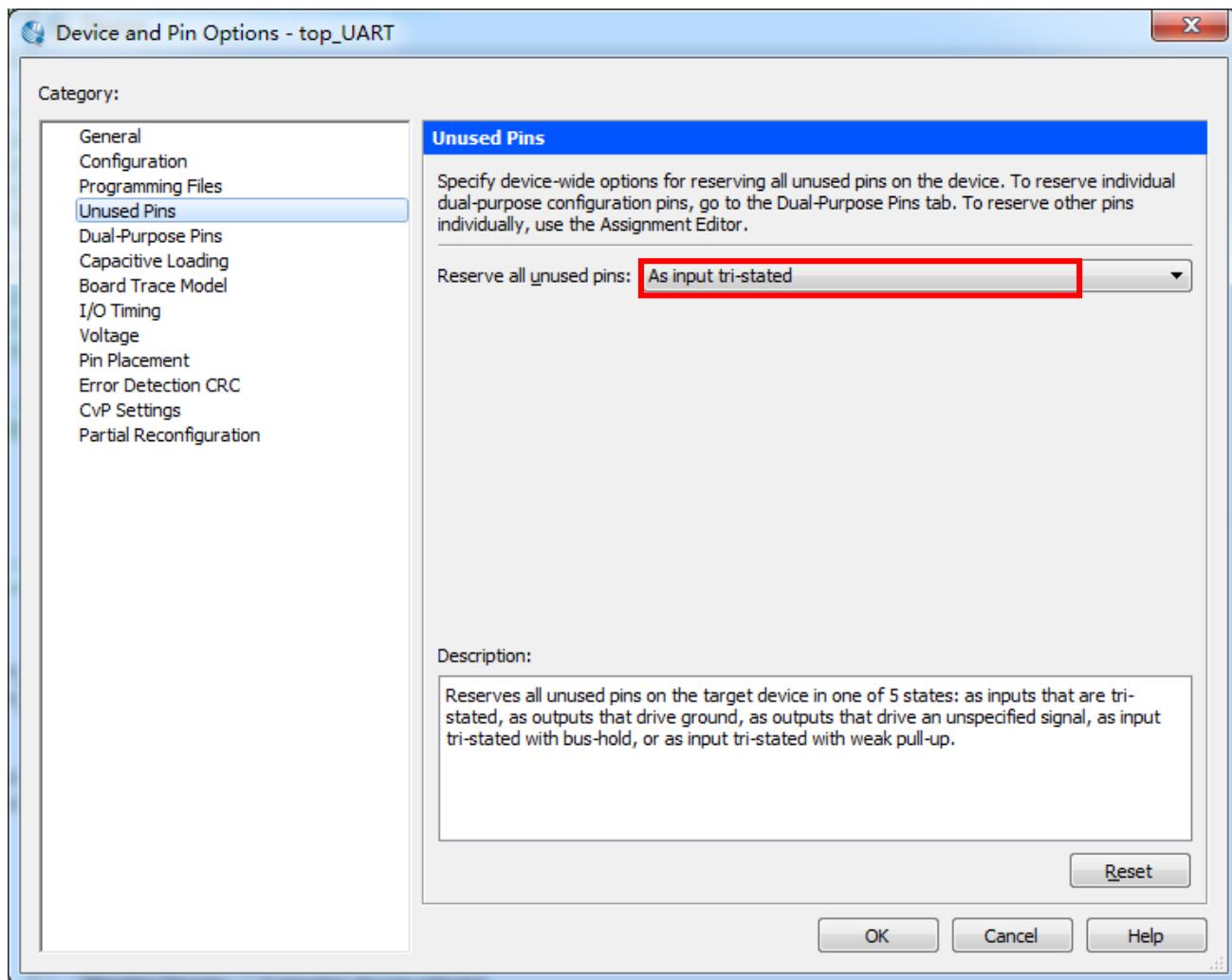


图 5.3.4 未使用引脚设置界面

接下来，将一些 IO 设置成普通 IO，通过双击红框位置，将一个个 Value 的值修改过来。如图 5.3.5 所示。

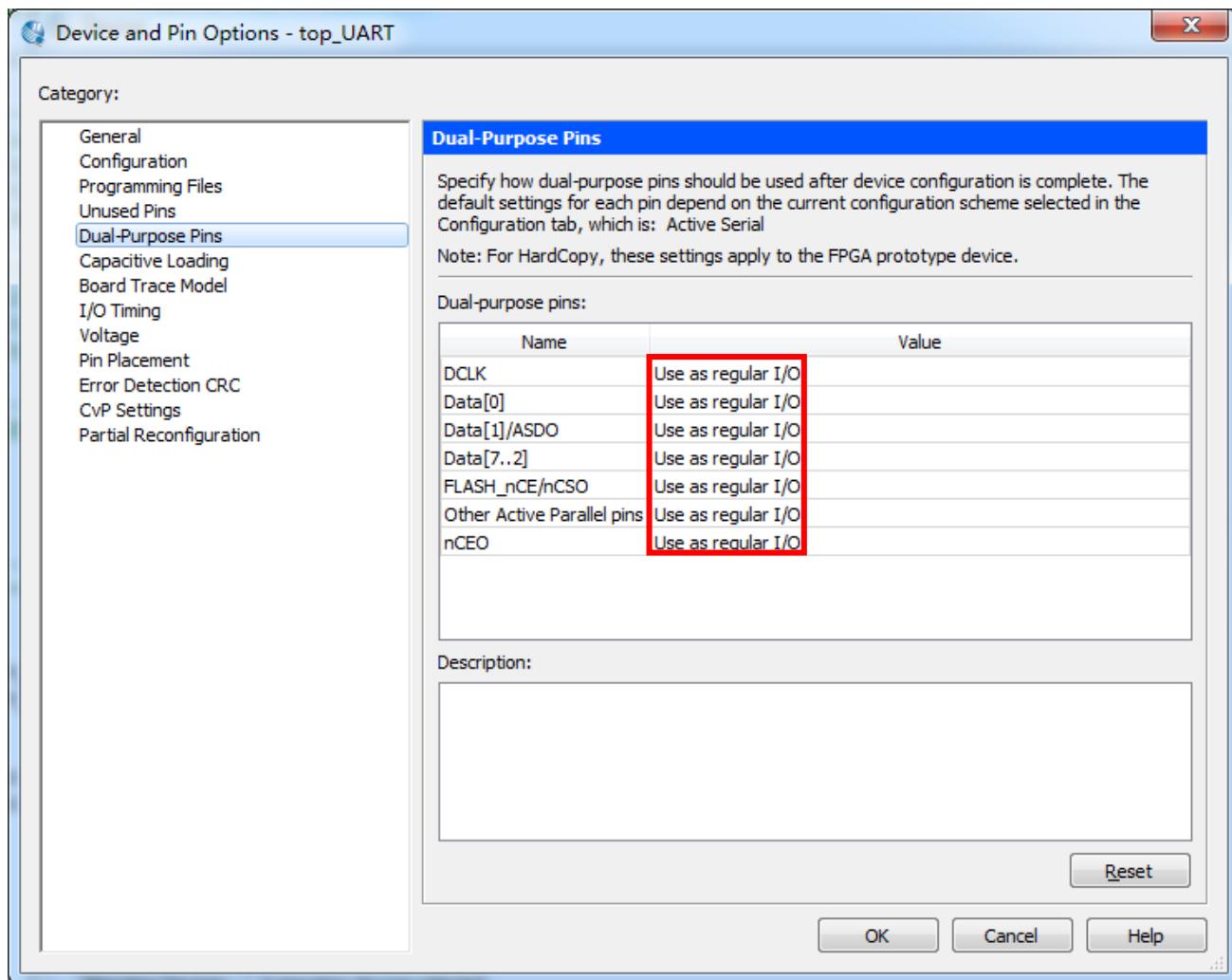


图 5.3.5 IO设置界面

我们通过 Quartus II 软件菜单栏中的【Assignments】→【Pin Planner】选项分配引脚，如图 5.3.6 所示。

Node Name	Direction	Location	I/O Bank	VREF Group	I/O Standard
in_rxd	Input	PIN_N5	3	B3_N0	2.5 V (default)
in_sys_clk	Input	PIN_E1	1	B1_N0	2.5 V (default)
in_sys_rst_n	Input	PIN_M1	2	B2_N0	2.5 V (default)
out_txd	Output	PIN_M7	3	B3_N0	2.5 V (default)

图 5.3.6 引脚分配界面

最后我们再进行一次全编译，成功编译硬件系统后，将产生用于配置 FPGA 的 top_UART.sof 文件。下面我们就来说明一下将.sof 文件下载到 FPGA 目标器件的步骤。

(1) 将下载器一端连接电脑，另一端与开发板上对应端口连接，最后连接电源线并打开电源开关。新起点开发板实物图如下所示：

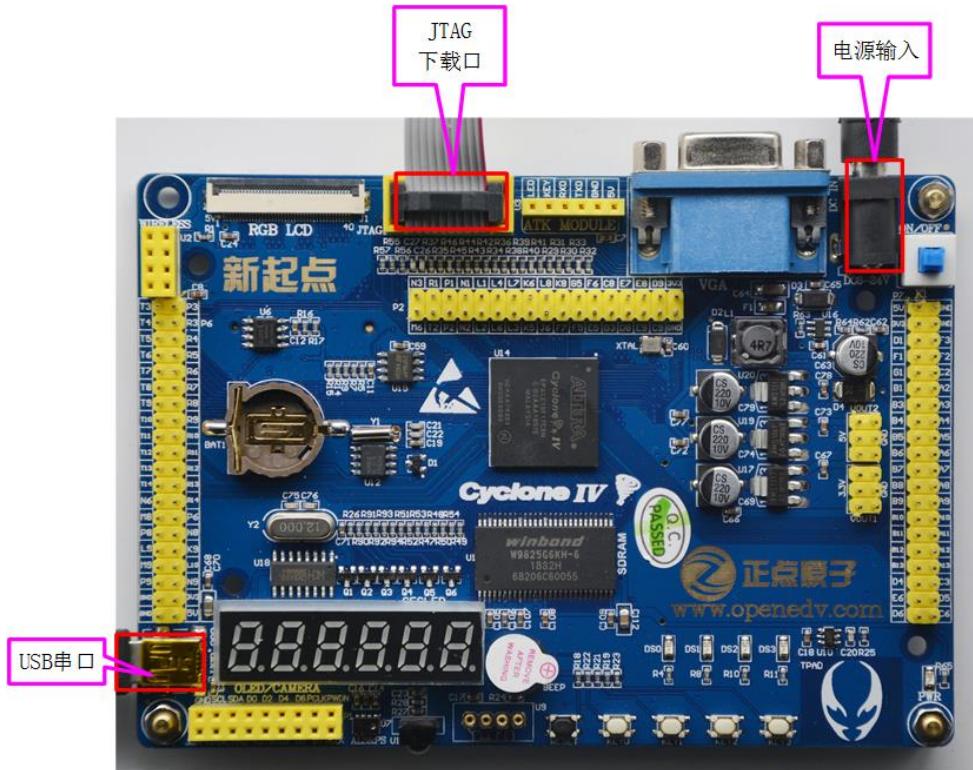


图 5.3.7 开发板实物图

接下来我们下载程序。工程打开后通过点击工具栏中的“Programmer”图标打开下载界面，通过“Add File”按钮选择 top_UART\par\output_files 目录下的“top_UART.sof”文件。开发板电源打开后，在程序下载界面点击“Hardware Setup”，在弹出的对话框中选择当前的硬件连接为“USB-Blaster[USB-0]”。然后点击“Start”将工程编译完成后得到的 sof 文件下载到开发板中。

至此，硬件部分设计完成，下面开始基于 Nios II SBT for Eclipse 的软件部分的设计。

5.4 软件设计

我们通过 Quartus II 软件菜单栏中的【Tools】→【Nios II SBT for Eclipse】，来启动 Nios II SBT for Eclipse 软件。打开 Nios II SBT for Eclipse 软件后，会弹出 Workspace Launcher 页面。我们这里将工作空间设置为 top_UART \qsys 路径下的 software 文件夹，如图 5.4.1 所示。

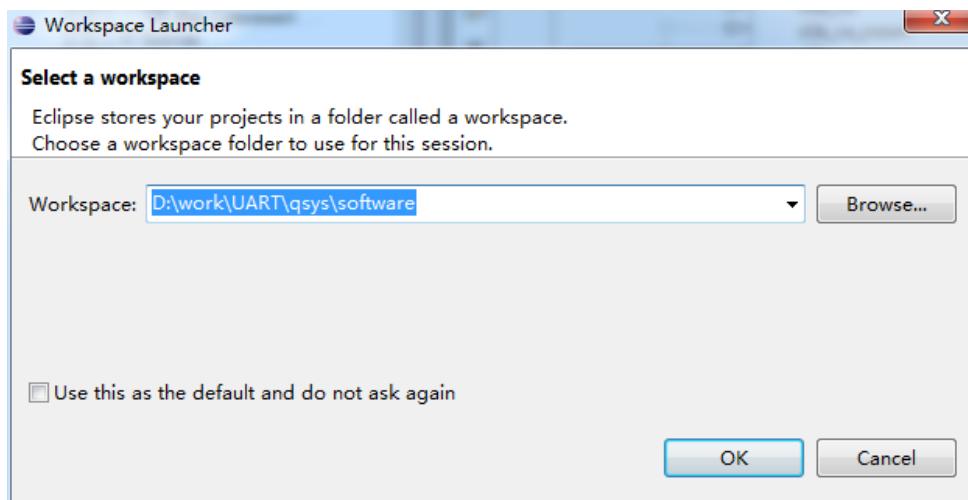


图 5.4.1 设置工作空间

设置好工作空间后，我们点击【OK】进入 Nios II SBT for Eclipse 软件主界面中，在该页面我们通过单击菜单栏中的【File】→【New】→【Nios II Application and BSP from Template】，来新建工程，如图 5.4.2 所示。

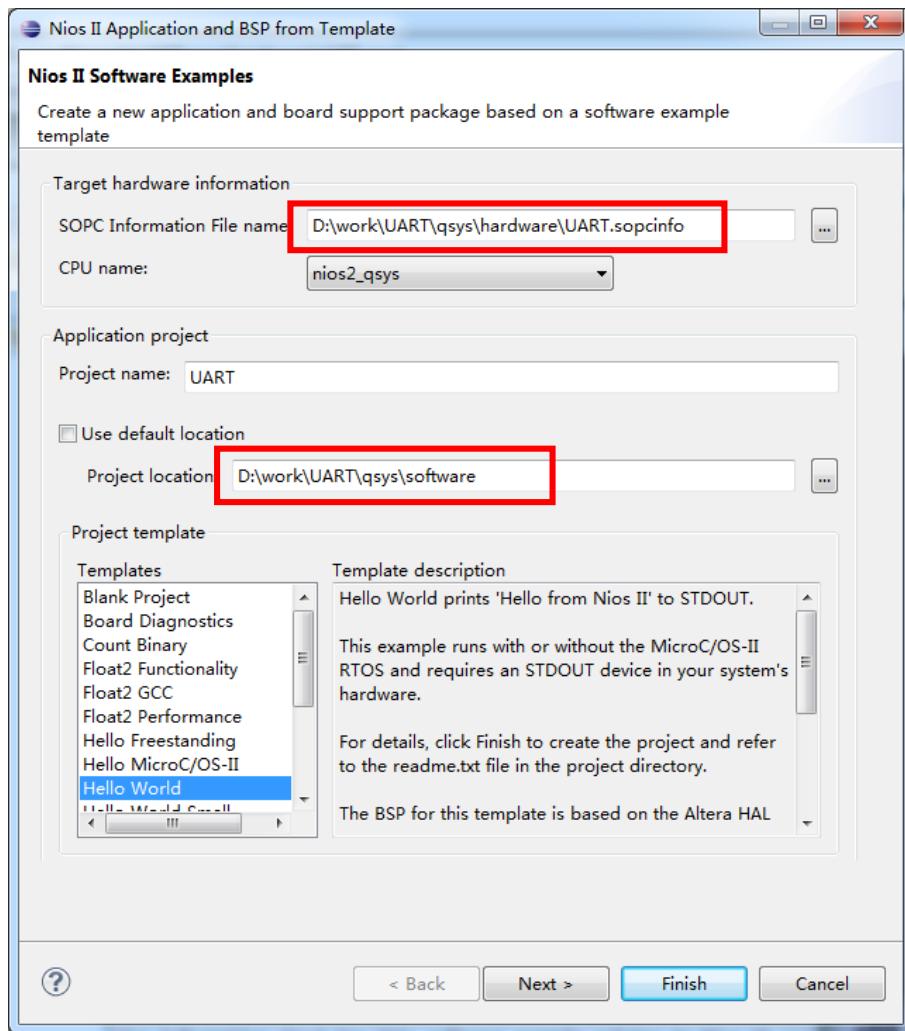


图 5.4.2 新建Nios II SBT for Eclipse 工程

单击【…】按钮来选择 top_UART\qsys\hardware 下的 top_UART.sopcinfo 文件，即指向当前硬件设计系统。Nios II SBT for Eclipse 软件会自动识别 Qsys 系统中 CPU 的名称，所以 CPU name 一项会自动生成。接下来，要给 Nios II SBT for Eclipse 软件中的工程命名，这里的名称没有特殊要求，我们这里名为 UART。然后将工程存放的位置修改为 top_UART\qsys\software\UART。注意不要漏掉了“\UART”，不然生成系统的时候会报错。最后我们来看下 Project template 窗口，该窗口中陈列的都是已经设计好的软件工程。我们可以从中选择一个，作为自己的工程的模板来使用。当然也可以选择 Bland Project（空白工程），就需要自己写所有的代码。这里我们选择的是 Hello World 模板工程，然后我们在它的基础上进行修改，这样比空白工程更加方便。

设置完工程后，直接点击【Finish】完成工程创建。然后，在 Nios II SBT for Eclipse 软件的左侧 Project Explorer 窗口中有两个工程：UART 和 UART_bsp。其中 UART 是 C/C++应用工程，而 UART_bsp 是描述 Qsys 系统硬件细节的系统库。打开 UART 工程里的 hello_world.c 文件，出现如图 5.4.3 所示的图。

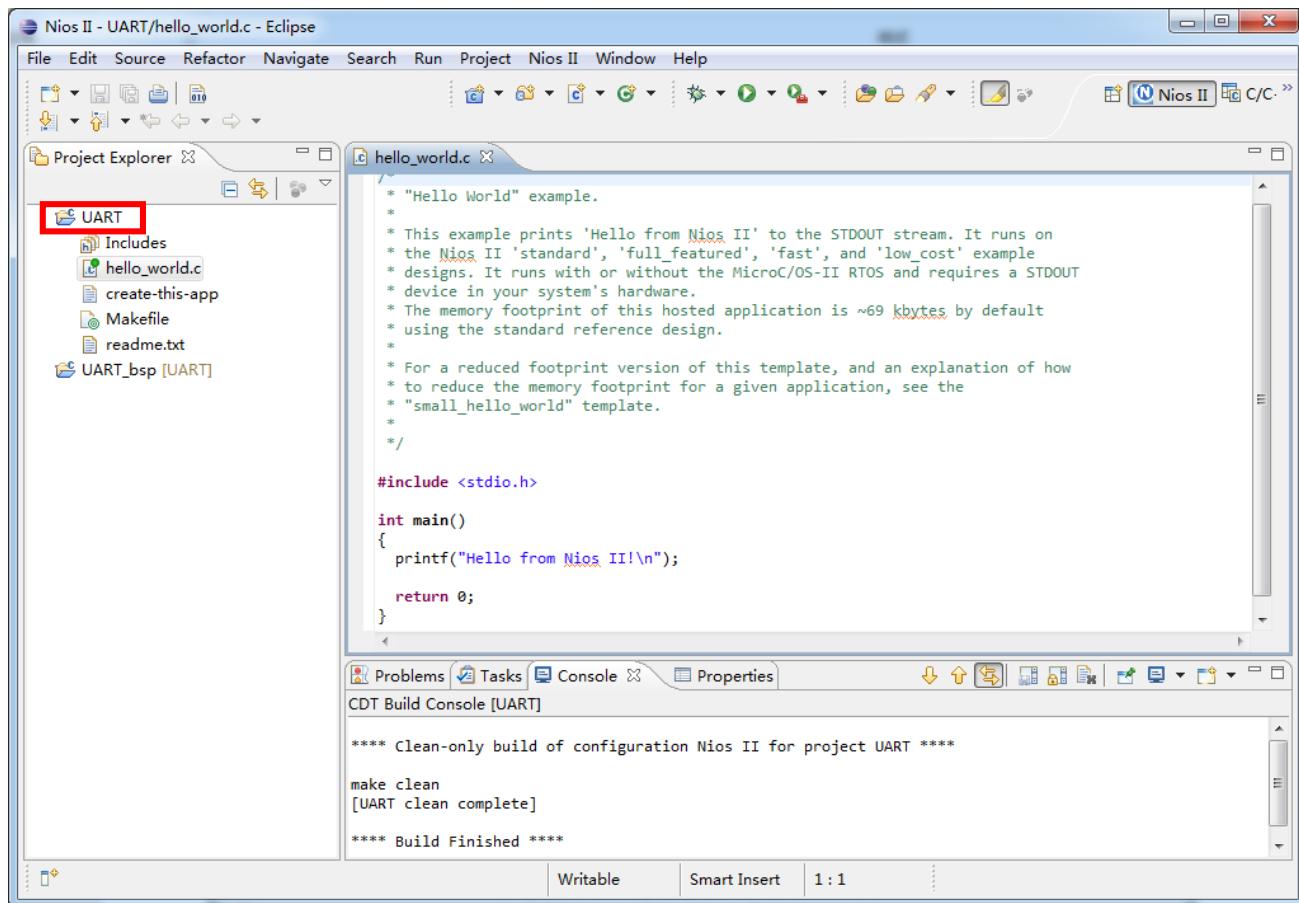


图 5.4.3 hello_world工程代码图

由代码可知，下载程序到开发板后会在窗口上输出“Hello from Nios II!”。我们在这里要验证之前创建的 Qsys 系统是否能正常工作。验证方法是先编译 UART 工程，然后将工程模板程序下载到开发板上，看是否能正常运行。再次之前，需要先简化代码。方法和“Hello,World”实验里一样是一样的，大家可以照着操作。

优化完代码之后，再编译一次 UART 工程，会出现以下的界面。这表示编译通过，可以将程序下载到开发板上了。

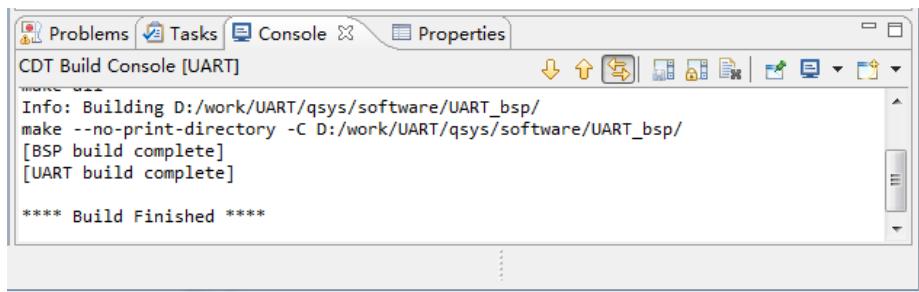


图 5.4.4 编译通过后的console窗口

这时大家点击【Run As】→【Nios II Hardware】，然后点击【Target Connection】标签，然后在 Target Connection 窗口中点击【Refresh Connections】按钮后。这时软件便会自动识别我们开发板上的 Qsys 系统，并显示 Qsys 系统的相关信息。我们接着点击【Run】，软件会把 irq.elf 文件下载至开发板中运行起来。更加详细的图和文字描述，可以在“Hello,World”实验的下载验证部分查看。

这时，若之前创建的 Qsys 系统无误，代码下载完成后在 Nios II console 窗口会显示“Hello from Nios II!”字符，如下图所示。

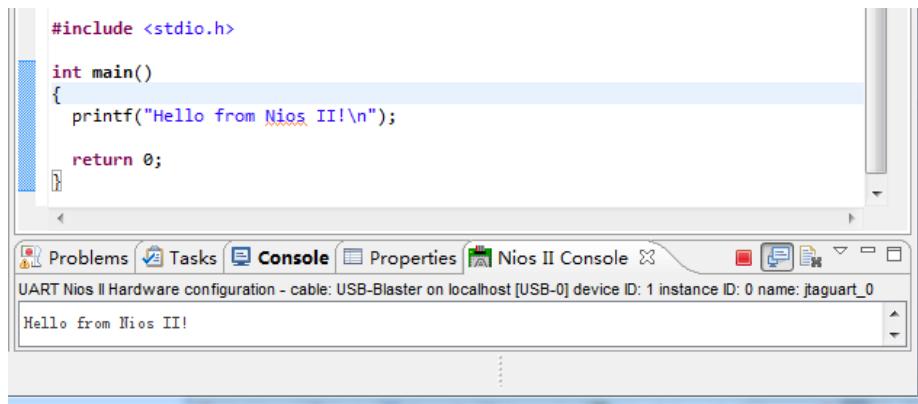


图 5.4.5 下载代码后的console窗口

验证完 Qsys 系统是否能正常运行之后，我们就可以开始软件部分的设计了。这时只需要在当前的代码窗口修改代码就可以了。代码如下所示。

```

1 #include <stdio.h>
2 #include "unistd.h"
3 #include "system.h"
4 #include "alt_types.h"
5 #include "altera_avalon_uart_regs.h"
6 #include "sys\alt_irq.h"
7 #include "stddef.h"
8 #include "priv/alt_legacy_irq.h"
9 static alt_u8 txdata = 0;
10 static alt_u8 rxdata = 0;
11
12 void IRQ_UART_Interrupts();           //中断初始化函数
13 void IRQ_init();                     //中断服务子程序

```

```
14
15 int main()
16 {
17     printf("Hello from Nios II!\n");
18     IRQ_init();
19     return 0;
20 }
21
22 void IRQ_init()
23 {
24     //清除状态寄存器
25     IOWR_ALTERA_AVALON_UART_STATUS(UART_BASE, 0);
26     //使能接受准备好中断, 给控制寄存器相应位写 1
27     IOWR_ALTERA_AVALON_UART_CONTROL(UART_BASE, 0X80);
28     // 注册 ISR
29     alt_ic_isr_register(
30         UART_IRQ_INTERRUPT_CONTROLLER_ID, // 中断控制器标号, 从 system.h 复制
31         UART_IRQ, // 硬件中断号, 从 system.h 复制
32         IRQ_UART Interrupts, // 中断服务子函数
33         0x0, // 指向与设备驱动实例相关的数据结构体
34         0x0); // flags, 保留未用
35 }
36
37 //UART 中断服务函数
38 void IRQ_UART_Interrupts()
39 {
40     //将 rxdata 寄存器中存储的值读入变量 rxdata 中
41     rxdata = IORD_ALTERA_AVALON_UART_RXDATA(UART_BASE);
42     //进行串口自收发, 将变量 rxdata 中的值赋值给变量 txdata
43     txdata = rxdata;
44     //查询发送准备好信号, 如果没有准备好, 则等待。
45     while(!(IORD_ALTERA_AVALON_UART_STATUS(UART_BASE) &
46             ALTERA_AVALON_UART_STATUS_TRDY_MSK));
```

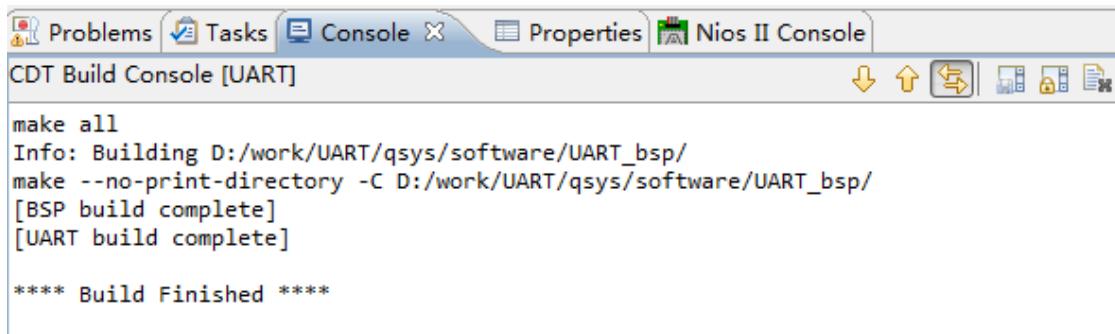
```
47     //发送准备好，发送 txdata  
48     IOWR_ALTERA_AVALON_UART_TXDATA(UART_BASE, txdata);  
49 }
```

代码的第 10 行至第 20 行是主函数。主函数中第 18 行代码完成了初始化中断的任务，完整的初始化中断函数在代码的第 22 行到第 35 行。在初始化中断的时候，需要先清除 status 寄存器（状态寄存器），这是在代码的第 25 行完成的。然后是使能中断，也就是给 control 寄存器（控制寄存器）相应的位写 1，这个操作在代码的第 27 行完成，通过简介部分的内容我们知道：给控制寄存器第 7 位置 1 的时候（使能第 7 位也就是 bit7 的中断），rxdata 寄存器每接收一次数据就发出 ISR。在代码的第 29 行至第 34 行，完成了注册中断的任务。在代码的第 38 行至代码的第 49 行是程序的中断函数。当 rxdata 寄存器接收到数据后，就触发 ISR，然后主程序开始运行中断函数。在中断函数中，先把 rxdata 寄存器中存储的值读入变量 rxdata 中，然后将变量 rxdata 中的值赋值给变量 txdata，再判断 txdata 寄存器是否处于空闲状态，若处于空闲状态，就把变量 txdata 中的值发送给 txdata 寄存器，否则就一直等待直到空闲状态的出现。在代码的第 45 行至 46 行完成了判断 txdata 寄存器工作状态的任务：首先通过 IORD_ALTERA_AVALON_UART_STATUS(UART_BASE) 这段代码读取状态寄存器的值，然后与 ALTERA_AVALON_UART_STATUS_TRDY_MSK（系统定义其值为 0x40）按位相与，这样可以得出状态寄存器第 6 位的值。若状态寄存器第 6 位为 1，则表示 txdata 寄存器处于空闲状态，那么代码相与的结果为非零值，经过逻辑取反则为零。若第 6 位的值为 0，经过逻辑取反后的结果为 1。当结果为 0 时，经过 while 判断，主程序会接着运行下一条代码。若结果为 1 时，经过 while 判断，主程序会一直在当前代码处运行。

代码修改完成后，大家记得要点一下快捷菜单中的【Save】，或者菜单栏中的【File】→【Save】，来保存修改后的程序。

5.5 下载验证

现在可以编译 UART 工程了。右键 UART 工程，点击 build project。稍等片刻，Console 窗口显示的内容如下图所示，这表示工程编译成功。



The screenshot shows the Eclipse CDT Build Console window titled "CDT Build Console [UART]". The console output is as follows:

```
make all
Info: Building D:/work/UART/qsys/software/UART_bsp/
make --no-print-directory -C D:/work/UART/qsys/software/UART_bsp/
[BSP build complete]
[UART build complete]

**** Build Finished ****
```

图 5.5.1 编译工程后的console窗口图

这时大家右键 UART 工程，点击【Run As】→【Nios II Hardware】，代码就被下载到开发板上了。接下来需要连接串口线并利用串口助手进行验证，这里请大家参考《新起点 FPGA 开发指南》第十六章下载验证部分进行操作。

第六章 定时器IP核

这一章中，我们将会讲到 Qsys 系统中一个比较重要的 IP 核——Interval Timer。拥有 Avalon 总线接口的 Interval Timer 核是，基于 Avalon 技术(Avalon-based)处理器系统中的一种计时器。它可被用作系统周期性的时钟源，还具有计时器、“看门狗”等功能。本章包括以下几个部分：

- 6.1 简介
- 6.2 实验任务
- 6.3 硬件设计
- 6.4 软件设计
- 6.5 下载验证

6.1 简介

Timer 核的结构图如下所示：

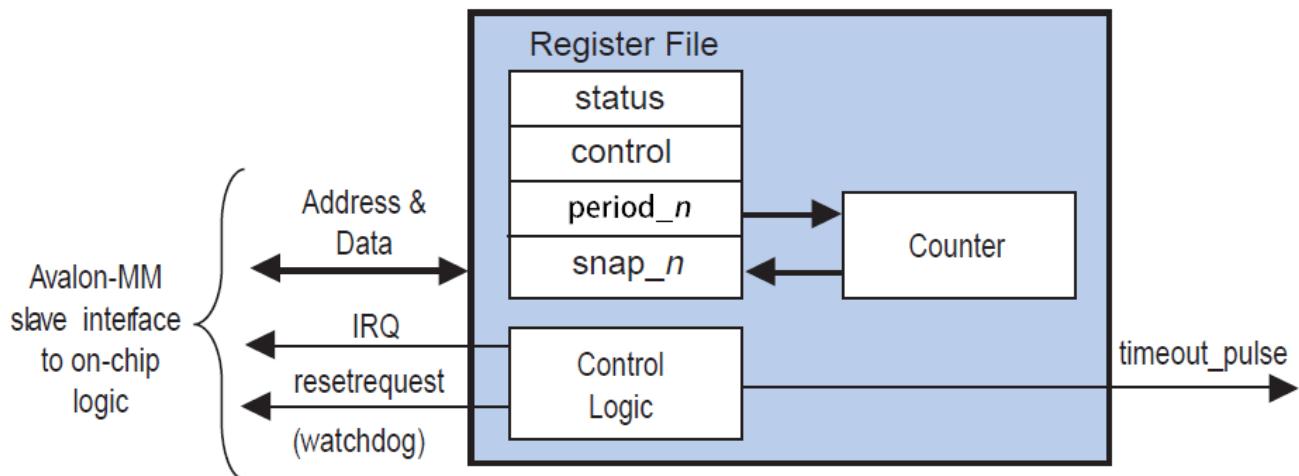


图 6.1.1 Interval Timer核结构图

从图中我们可以看出 Timer 核以下的功能：

- 与 6 个 16 位寄存器连接的 Avalon-MM 接口
- 可选的脉冲输出：能被用作周期脉冲产生器

所有的寄存器都是 16 位位宽，这样使得与 16 位以及 32 位的处理器都相兼容。有些寄存器只有在配置了之后才会存在于硬件中。例如，如果核被配置成固定周期，硬件中则没有周期寄存器（period registers）。

下面的内容描述了 interval timer 核的基础特性：

- 类似于 Nios II 处理器的 Avalon-MM 主设备可以通过写控制寄存器（control register）来完成以下任务：
 1. 启动和停止定时器
 2. 使能/禁止 IRQ
 3. 明确累减计数一次或重复性累减计数
- 处理器通过读状态寄存器（status register），能够得知定时器所处的状态
- 处理器可以通过给周期寄存器（period registers）写数值来指定定时器的周期

- 内部计数器累减到 0，然后会从周期寄存器（period registers）获取数值，继续累减计数
- 处理器可以通过一些操作来获取当前计数器（counter）的值：先给其中一个捕获寄存器（snap registers）写数据，然后读捕获寄存器（snap registers）来获得完整的值。
- 当计数器计数到 0 的时候，以下的事件可能会触发：
 1. 如果 IRQ 被使能，此时会触发 IRQ
 2. 可选的一个时钟周期的脉冲输出
 3. 可选的看门狗输出，复位系统

(1) 寄存器相关简介

32 位定时器的寄存器概述表如下图所示：

表 6.1-1 寄存器概述表

Offset	Name	R/W	Description of Bits							
			15	...	4	3	2	1	0	
0	status	RW	(1)					RUN	TO	
1	control	RW	(1)		STOP	START	CONT	IT	O	
2	periodl	RW	Timeout Period - 1 (bits [15:0])							
3	periodh	RW	Timeout Period - 1 (bits [31:16])							
4	snapl	RW	Counter Snapshot (bits [15:0])							
5	snaph	RW	Counter Snapshot (bits [31:16])							

表中的 (1) 指的是保留部分，读它会返回未定义的值。进行写操作要写 0

表中我们可以看出，Timer 核有 6 个寄存器，接下来我们会对这些寄存器进行详细的讲解。

status 寄存器（状态寄存器）

表 6.1-2 status 寄存器概述表

位	名称	写/读/擦除	描述
0	TO	读和擦除	当内部计数器计数到 0 的时候 TO 位 (timeout) 置 1。一旦被置 1

			后，只有给 status 寄存器写 0 才能将 TO 位置 0
1	RUN	读	内部计数器在运行的时候，读 RUN 位会返回 1。否则读这个位会返回 0。对 status 寄存器进行写操作不会改变 RUN 位的值。

control 寄存器（控制寄存器）

表 6.1-3 control 寄存器概述表

位	名称	写/读/擦除	描述
0	ITO	读写	如果 ITO 位的值是 1，当 status 寄存器中的 TO 位是 1 的时候，interval timer 核产生一个 IRQ。当 ITO 位的值是 0 的时候，timer 核无法产生 IRQ
1	CONT	读写	CONT 位决定了内部计数器计数到 0 后的操作。如果 CONT 位的值是 1，计数器会一直运行，直到被 stop 位停止。如果 CONT 位的值是 0，计数器计数到 0 后就停止。无论 CONT 位的值是多少，计数器计数到 0 后，都会加载存储在 period registers 中的值。
2	START	写	给 START 位写 1 会让内部计数器运行（累减）。如果定时器已经停止运行，给 START 位写 1 会使定时器从当前计数器中存储的值开始继续计数。如果定时器正在运行，对 START 位进行写操作无影响。
3	STOP	写	给 STOP 位写 1 会让内部计数器停止运行。如果定时器已经停止运行，对 START 位进行写操作无影响。 如果定时器硬件配置中选中了 Start/Stop control bits off 选项，写 STOP 位无影响。

注：给 START 位以及 STOP 同时写 1 会造成未定义的影响。

period 寄存器（周期寄存器）

所有的 period 寄存器一起存储着周期极大值。发生以下事件时，内部计数器会装载 period 寄存器中存储的值：

- 对任一个周期寄存器（period registers）进行写操作
- 内部计数器计数到 0

定时器的实际周期要比存储在周期寄存器（period registers）中的值多一个周期，这是因为计数器认定 0 值也算一个周期。

对任一个 period 寄存器进行写操作都会使内部定时器停止运行，除非硬件配置中没有选中 Start/Stop control bits。

Snap 寄存器（捕获寄存器）

主设备可通过对 snap 寄存器进行写操作来获得内部计数器的当前值。当一个写操作发生后，计数器中的值会被赋值到 snap 寄存器中。无论定时器是否正在运行，获取当前计数器中的值的操作都会进行。这种操作不会影响到内部计数器的运行状况。

(2) 中断操作

当 control 寄存器的 ITO 位置 1，且内部计数器计数到 0 的时候，会触发 IRQ。回应 IRQ 有两种操作：

- 清除 status 的 TO 位
- 清除 control 寄存器的 ITO 位来禁止中断

没有回应 IRQ 会产生未定义的结果。

(3) 硬件配置内容

如图为 Timer 核的配置界面：

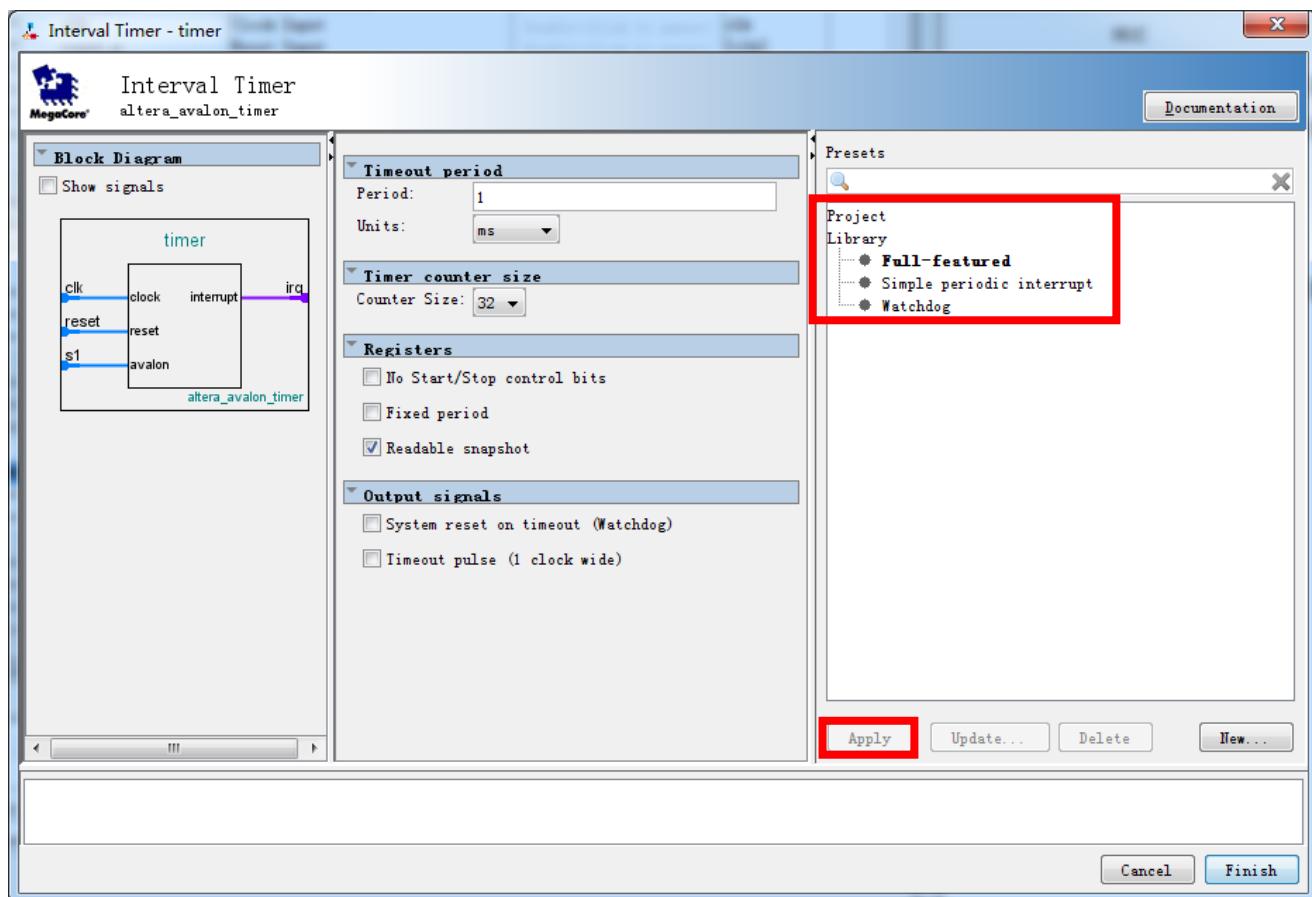


图 6.1.2 Timer核配置界面

图中红框中的选项能影响到 interval timer 核的硬件结构。比较便利的是，预先配置列表提供了几个预定义的硬件配置，例如：

- Simple periodic interrupt（简单周期中断）——这个配置对于只需要周期性地触发 IRQ 的系统，是十分有用的。周期是固定的，定时器无法被停止。但是 IRQ 是可以被禁止的。
- Full-featured（全功能）——这个配置对于这样的嵌入式处理器系统是十分有用的：要求定时器能被处理器启动或停止，但能禁止 IRQ。
- Watchdog（看门狗）——这个配置对于这种系统是比较便利的：需要“看门狗”定时器去复位停止响应的系统

在选择其中的一个选项的时候，需要先双击该选项，然后点 Apply。接下来我们来讲解其他的配置项目。

1.Timeout Period (超时周期)

超时周期决定了周期寄存器（period registers）的初始值。当没有选中 Fixed period（固定周期）选项的时候，处理器可以通过给周期寄存器（period registers）赋值来改变周期的值（在软件中）。如果选中了 Fixed period（固定周期）选项，那么周期是固定的，且无法在软件中更改。超时周期（Timeout Period）是定时器频率（Timer Frequency）的整数倍。定时器频率固定为系统时钟的频率。定时器频率设置的单位可以是 μs (microseconds), ms (milliseconds), seconds, 或者 clocks（系统时钟周期个数）。

- **Period:** 定时器周期，没有选中 Fixed period 选项的话，可以在软件中修改。若软件中没有修改操作，period 中的值就会作为超时周期（Timeout Period）
- **Units:** 周期的单位，可以是 μs 、ms、seconds、clocks。

2. Counter Size（计数器大小）

Counter Size 设置决定了定时器的位宽，它可以设置成 32 位或 64 位。一个 32 位的定时器拥有 2 个 16 位的 period registers（周期寄存器）。64 位的定时器拥有 4 个 16 位的 period registers（周期寄存器）。

3. Registers（寄存器）

- **No Start/Stop control bits:** 没有选中该选项时，主设备可以通过写 control 寄存器中的 START 和 STOP 位来启动或者停止定时器。当选中该选项后，定时器会一直运行。当硬件中使能了“watchdog”功能后，无论 No Start/Stop control bits 选项如何，control 寄存器中 START 位有效。
- **Fixed period:** 选中该选项后，就无法在软件中更改定时器周期，且定时器周期为 Timeout Period 处设置的结果；若没有选中该选项，可以通过在软件中给 period registers 赋值来更改定时器周期（timeout Period）
- **Readable snapshot:** 选中该选项后，主设备可以读取当前的计数值。没有选中该选项时，硬件中将不存在 snap 寄存器，且读 snap 寄存器会返回未定义的值。在这种配置下，计数器（counter）的状态只能通过 status 寄存器或 IRQ 信号之类的来观测

4. Output signals（输出信号）

- **System reset on timeout(watchdog):** 这个选项被选中后，核的 Avalon-MM 从端口会包含 resetrequest（复位请求）信号。当定时器计数到 0 时，resetrequest 信号会持续一个时钟周期的高电平，因此使得系统复位。在复位的时候，内部定时器是停止的。写 control 寄存器的 START 位会启动定时器。

没有选中该选项的时候，`resetrequest`（复位请求）信号不会被产生。

- `Timeout pulse(1 clock wide)`: 选中这个选项后，当定时器计数到 0 时，会输出 `timeout_pulse` 信号高电平一个周期。没有选中该选项的时候，`timeout_pulse` 信号不会被产生。

6.2 实验任务

本节实验任务是：在 Qsys 系统中通过使用官方 Interval Timer IP 核实现定时器的功能，并使蜂鸣器周期性地发声。

6.3 硬件设计

创建 Quartus II 工程

首先要创建 Quartus II 工程，这里就不赘述了。

创建 Qsys 系统

实验中要用到的 IP 核有：`clk`(时钟)、`nios II`(处理器)、`onchip_ram`(片内存储)、`jtag_uart`、`sysid_qsys`、`PIO`、`Interval Timer`（定时器）。其中只有 `Interval Timer` IP 核、需要讲解一下，其他的 IP 核都是按照以前的配置方法进行设置，本节就讲讲如何配置 `Timer` IP 核。

从 Library 中选择 `Timer` IP 核并打开，出现以下界面：

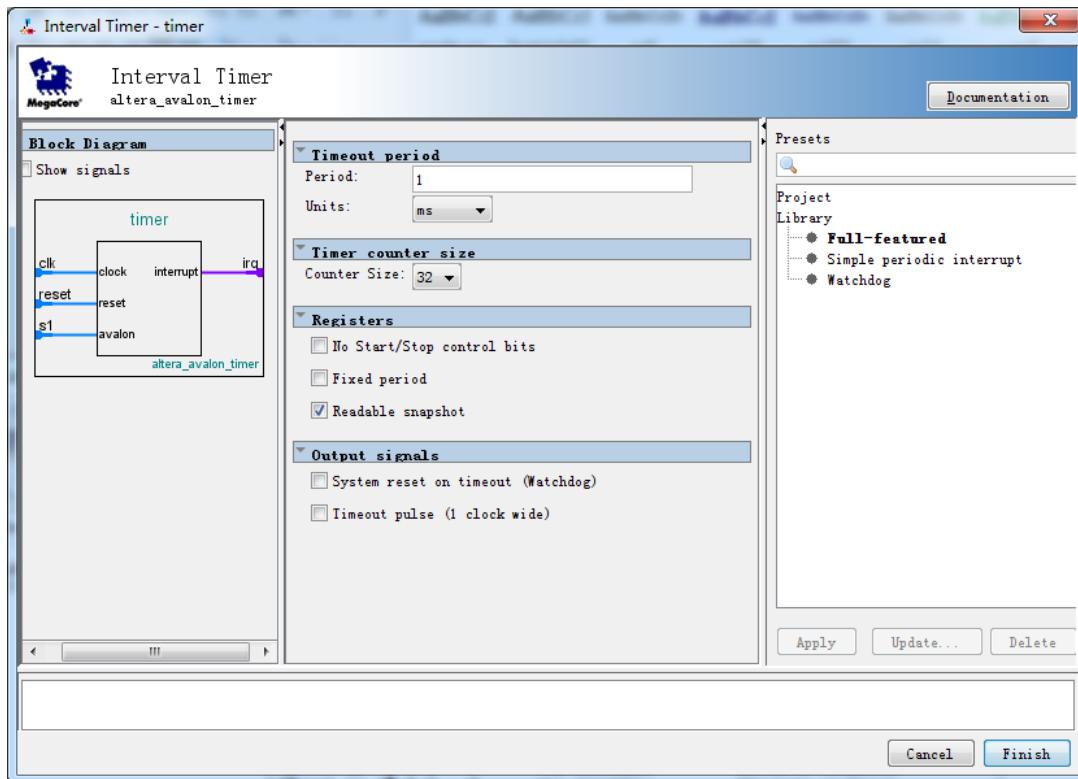


图 6.3.1 Timer核的配置界面

这里我们使用默认设置，直接点击【Finish】即可。

然后，我们打开 nios II IP 核配置界面，因为这里只用了 onchip_ram IP 核存储代码和指令，所以需要对相关的设置进行修改。如图 6.3.2 所示，在 Reset Vector 处将 Reset vector memory 处的选项选为 onchip_ram，同时在 Exception Vector 处也将 Exception vector memory 处的选项选为 onchip_ram。需要注意的是，onchip_ram IP 核添加之后，nios II IP 核的 Reset Vector 和 Exception Vector 选项中才会出现 onchip_ram 选项。

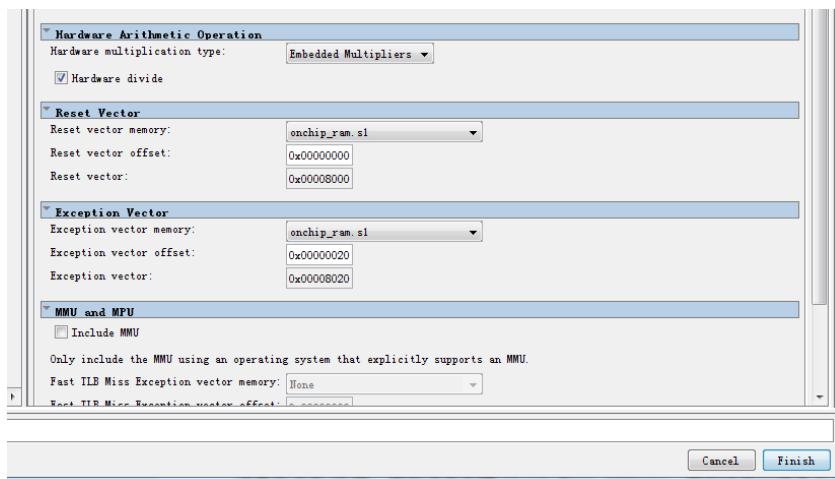


图 6.3.2 nios II IP核设置界面

然后是 PIO IP 核，由于我们只需要和蜂鸣器这一个外部设备进行通信，所以只需要设置一个位宽为 1 的 PIO IP 核，如下图所示。

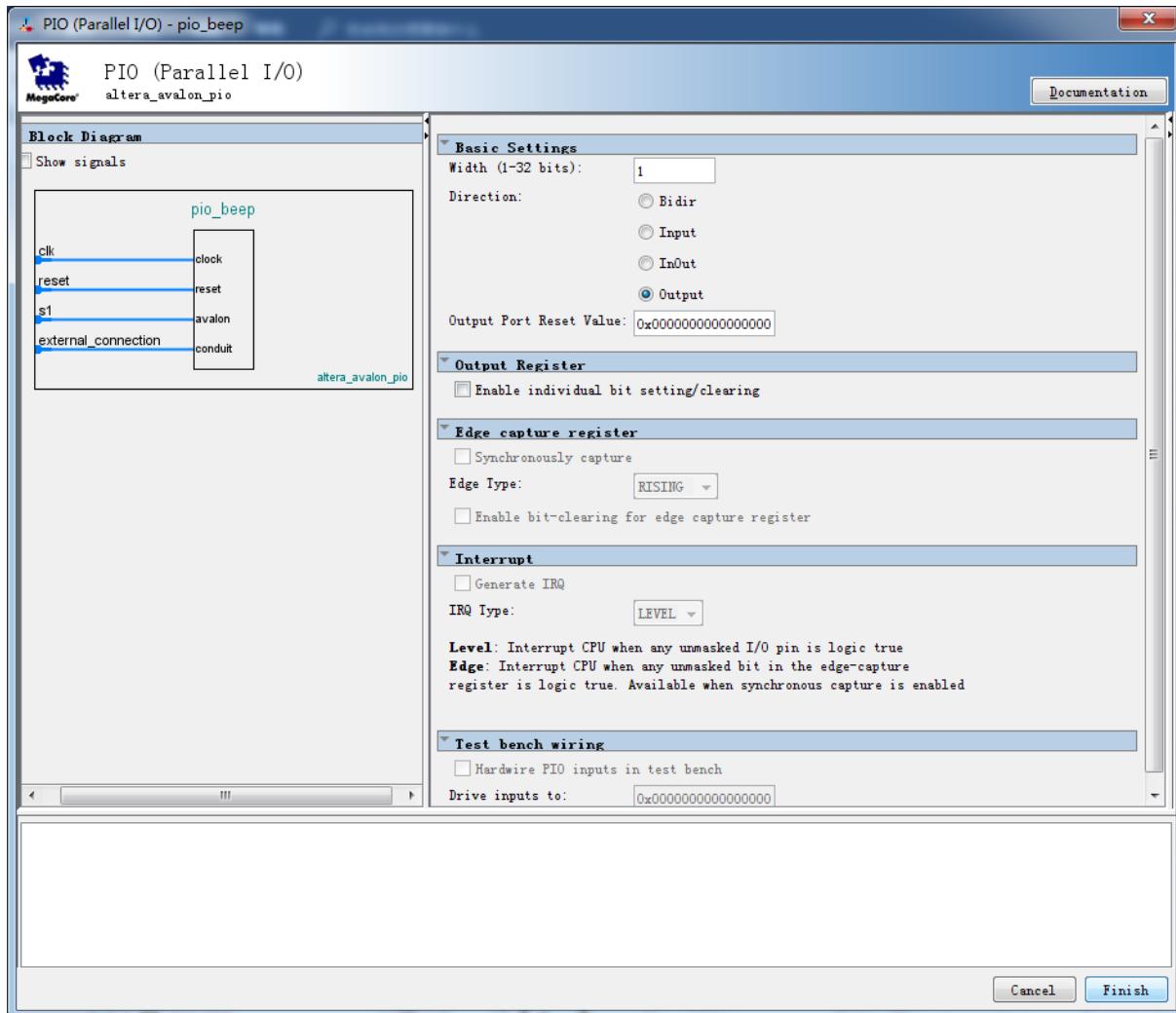


图 6.3.3 PIO IP核设置界面

添加完 IP 核后就可以开始连线，大家若是不熟悉怎么连线，可以照着下面完成的 Qsys 系统界面图连。需要注意的是，要将 PIO IP 核的端口引出来，如图 6.3.4 所示。引出端口的方法是双击图 6.3.4 中 IP 核的 Export 一栏的红框位置，然后修改名称，按下 Enter 键即可。

然后，点击 System→Assign Base Addresses 让系统自动分配地址，这里最好把 onchip_ram 的地址锁住，这是因为这个 IP 核里存储着指令，最好不要让其地址发生变动。锁住地址的方法是先点击 IP 核，然后点击右键→Lock Base Address。我们还可以将各个 IP 核的名称修改一下。最后就是生成系统了，操作可以按照“Hello, World”文档里的进行。

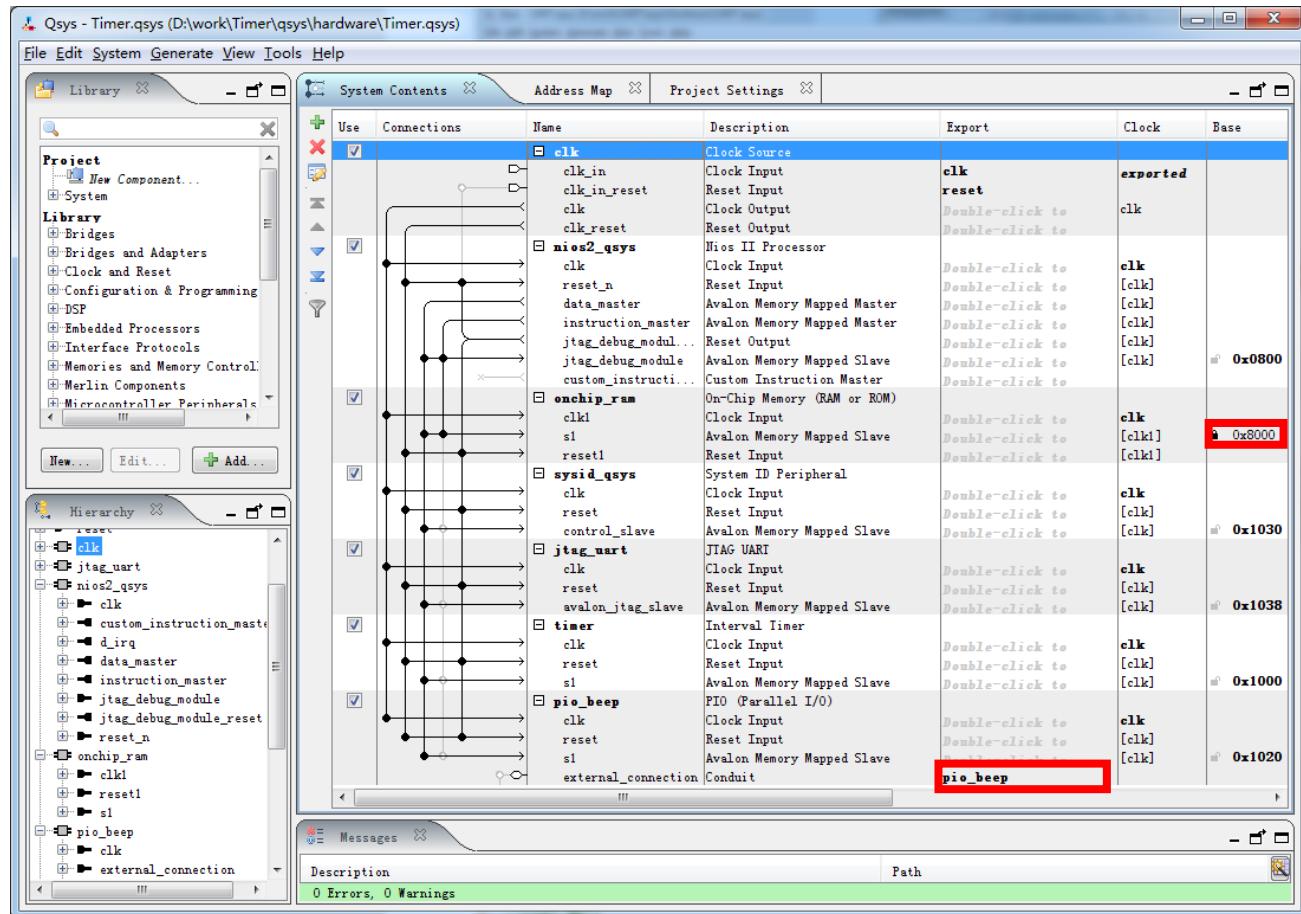


图 6.3.4 nios II IP核设置界面

集成 Qsys 系统

这一步依然可以按照“Hello, World”文档里的操作进行。

下面将 Quartus II 工程中的顶层代码贴出来。

```

1 module top_Timer(
2     input      sys_clk      ,
3     input      sys_RST_N   ,
4
5 //蜂鸣器
6     output     beep        //beep
7 );
8
9 //wire define
10 wire    clk_100m;           //100mHZ 时钟

```

```
11
12 //例化 pll 模块, 用以产生
13 pll pll_inst (
14     .inclk0      (sys_clk) ,
15     .c0          (clk_100m)
16 );
17
18 //例化 Qsys 系统
19 Timer u0 (
20     .clk_clk      (clk_100m),      //      clk.clk
21     .reset_reset_n (sys_rst_n),    //      reset.reset_n
22     .pio_beep_export (beep)       //      pio_beep.export
23 );
24
25 endmodule
```

编译和下载

这时, 我们便能够进行编译查错了, 我们可以通过 Quartus II 软件菜单栏中的【Processing】→【Start Compilation】来进行编译, 也可以通过快捷栏中的快捷键进行编译。

接下来我们就需要进行配置 IO, 分配管脚。首先, 点击 Quartus II 软件菜单栏中的【Assignment】→【Device】, 然后我们在 Device 界面中找到【Device and Pin Options…】进入图 6.3.5 所示页面配置 IO。将未使用引脚设置为高阻输入 (As input tri-state), 这样上电后 FPGA 的所有不使用引脚都将进入高阻抗状态。

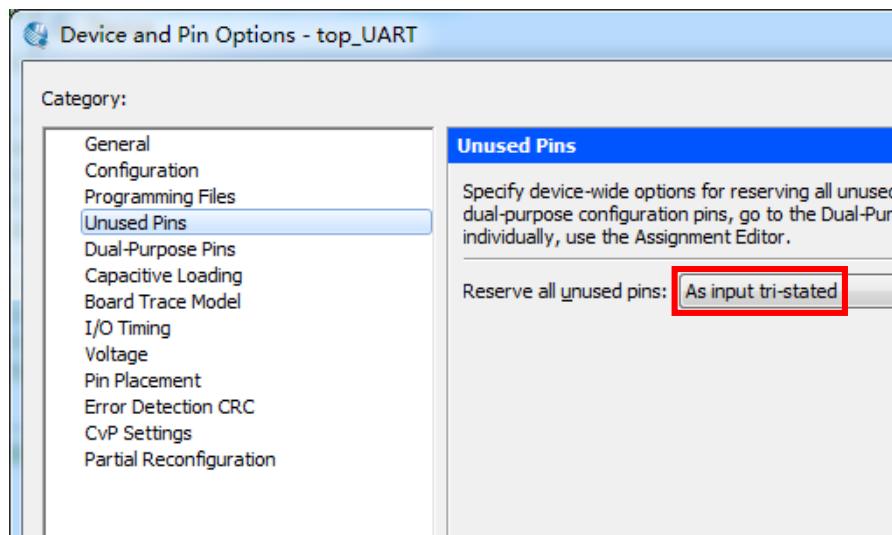


图 6.3.5 未使用引脚设置界面

接下来，将一些 IO 设置成普通 IO，通过双击红框位置，将一个个 Value 的值修改过来。如图 6.3.6 所示。

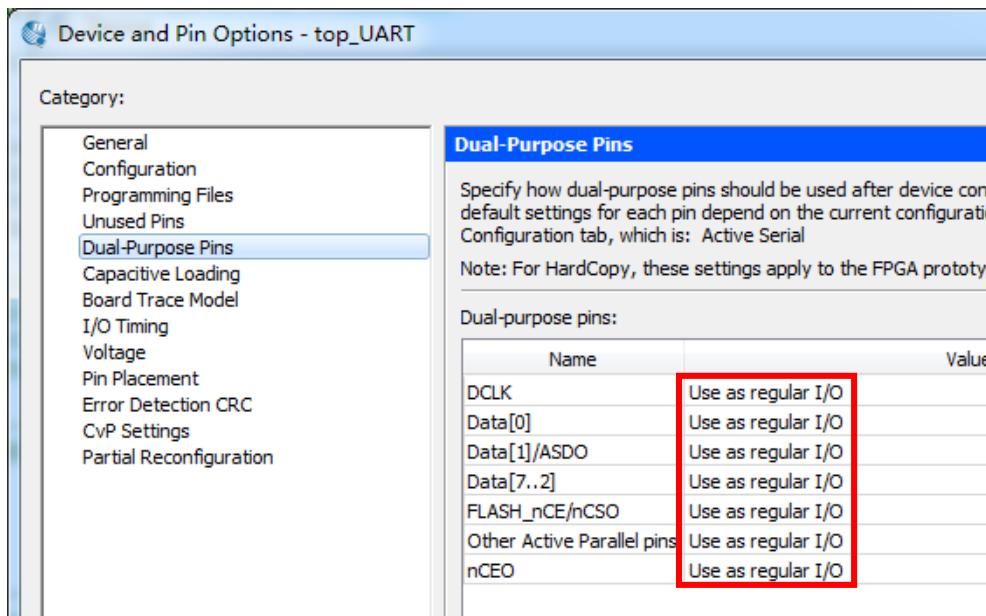


图 6.3.6 IO设置界面

我们通过 Quartus II 软件菜单栏中的【Assignments】→【Pin Planner】选项分配引脚。需要说明的是，由于蜂鸣器需要的驱动电流较大，使用默认 8mA 的驱动电流有可能出现蜂鸣器发声较小的情况，解决方法是将蜂鸣器输出的驱动电流修改成 12mA 或者是 16mA，如图 6.3.7 所示。

out	beep	Output	PIN_D12	7	B7_N0	PIN_D12	2.5 V (default)	12mA
in	sys_dk	Input	PIN_E1	1	B1_N0	PIN_E1	2.5 V (default)	8mA (default)
in	sys rst n	Input	PIN_M1	2	B2_N0	PIN_M1	2.5 V (default)	8mA (default)

图 6.3.7 引脚分配界面

最后我们再进行一次全编译，成功编译硬件系统后，将产生用于配置 FPGA 的 top_Timer.sof 文件。下面我们就来说明一下将.sof 文件下载到 FPGA 目标器件的步骤。

(1) 将下载器一端连接电脑，另一端与开发板上对应端口连接，最后连接电源线并打开电源开关。新起点开发板实物图如下所示：

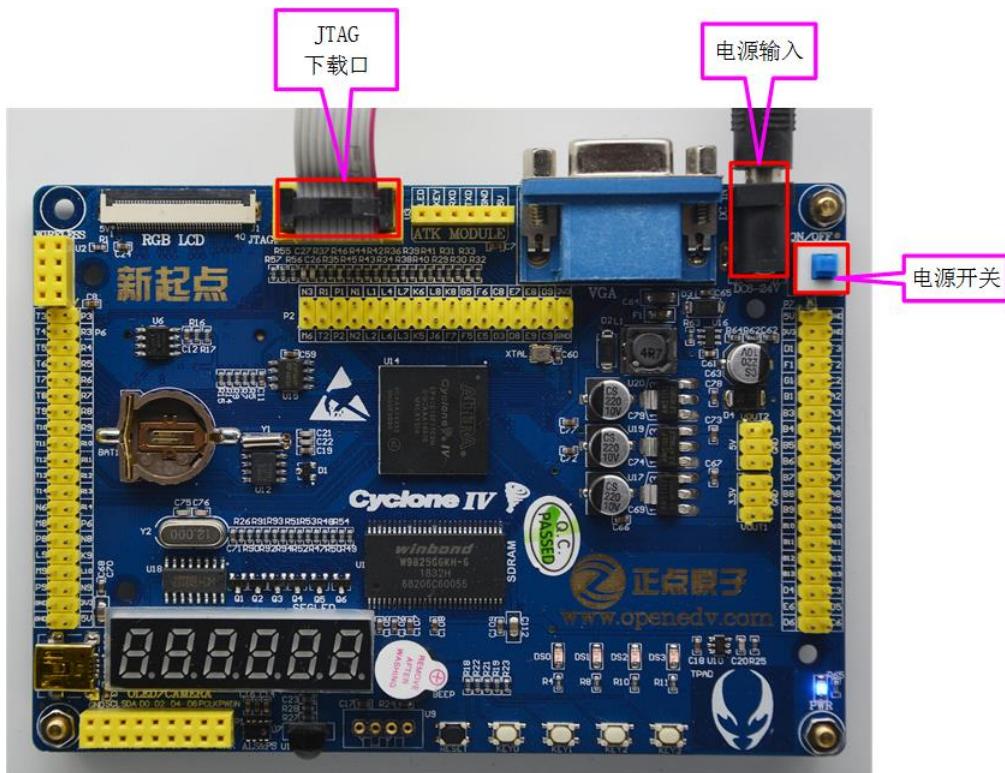


图 6.3.8 开发板实物图

接下来我们下载程序。工程打开后通过点击工具栏中的“Programmer”图标打开下载界面，通过“Add File”按钮选择 top_Timer\par\output_files 目录下的“top_Timer.sof”文件。开发板电源打开后，在程序下载界面点击“Hardware Setup”，在弹出的对话框中选择当前的硬件连接为“USB-Blaster[USB-0]”。然后点击“Start”将工程编译完成后得到的 sof 文件下载到开发板中，如图 6.3.9 所示：

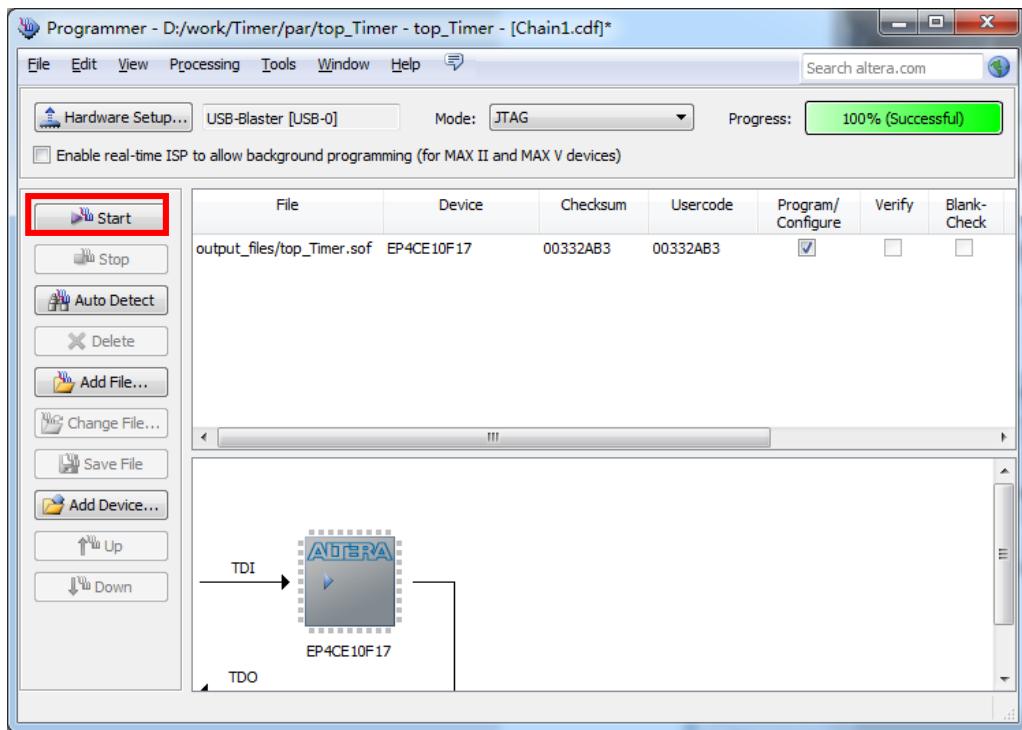


图 6.3.9 程序下载完成界面

至此，硬件部分设计完成，下面开始基于 Nios II SBT for Eclipse 的软件部分的设计。

6.4 软件设计

我们通过 Quartus II 软件菜单栏中的【Tools】→【Nios II SBT for Eclipse】，来启动 Nios II SBT for Eclipse 软件。打开 Nios II SBT for Eclipse 软件后，会弹出 Workspace Launcher 页面。我们这里将工作空间设置为 top_UART \qsys 路径下的 software 文件夹，如图 6.4.1 所示。

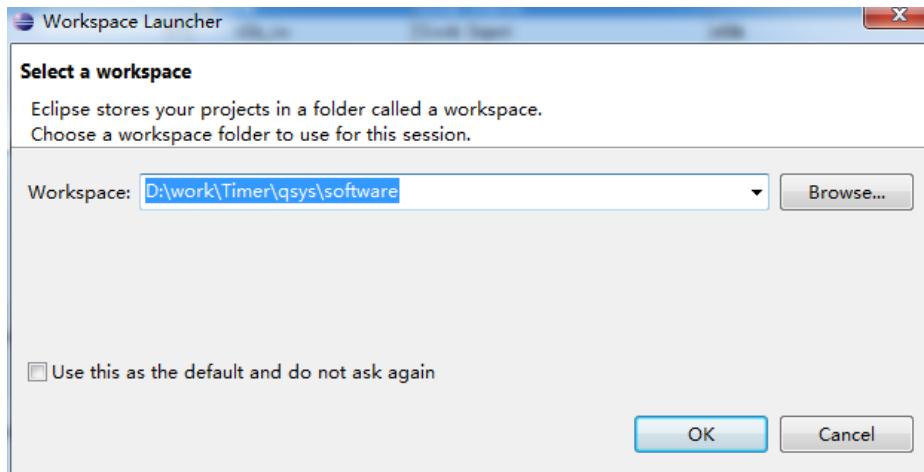


图 6.4.1 设置工作空间

设置好工作空间后，我们点击【OK】进入 Nios II SBT for Eclipse 软件主界面中，在该页面我们通过单击菜单栏中的【File】→【New】→【Nios II Application and BSP from Template】，来新建工程，如图 6.4.2 所示。

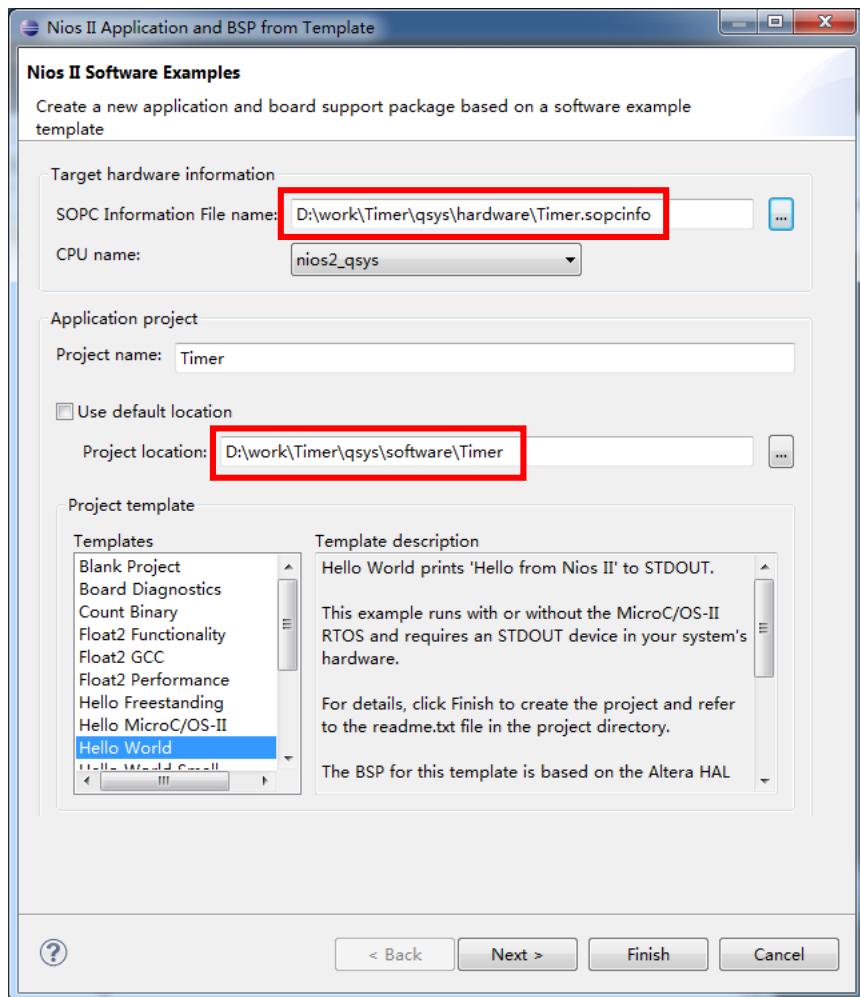


图 6.4.2 新建Nios II SBT for Eclipse 工程

单击【...】按钮来选择 top_Timer\qsys\hardware 下的 top_Timer.sopcinfo 文件，即指向当前硬件设计系统。Nios II SBT for Eclipse 软件会自动识别 Qsys 系统中 CPU 的名称，所以 CPU name 一项会自动生成。接下来，要给 Nios II SBT for Eclipse 软件中的工程命名，这里的名称没有特殊要求，我们这里名为 UART。然后将工程存放的位置修改为 top_Timer\qsys\software\Timer。注意不要漏掉了“\Timer”，不然生成系统的时候会报错。最后我们来看下 Project template 窗口，该窗口中陈列的都是已经设计好的软件工程。我们可以从中选择一个，作为自己的工程的模板来使用。当然也可以选择 Blank Project（空白工程），就需要自己写所有的代码。这里我们选择的是 Hello World 模板工程，然后我们在它的基础上进行修改，这样比空白工程更加方便。

设置完工程后，直接点击【Finish】完成工程创建。然后，在 Nios II SBT for Eclipse 软件的左侧 Project Explorer 窗口中有两个工程：Timer 和 Timer_bsp。其中 UART 是 C/C++应用工程，而 UART_bsp 是描述 Qsys 系统硬件细节的系统库。打开 UART 工程里的 hello_world.c 文件，出现如图 6.4.3 所示的图。

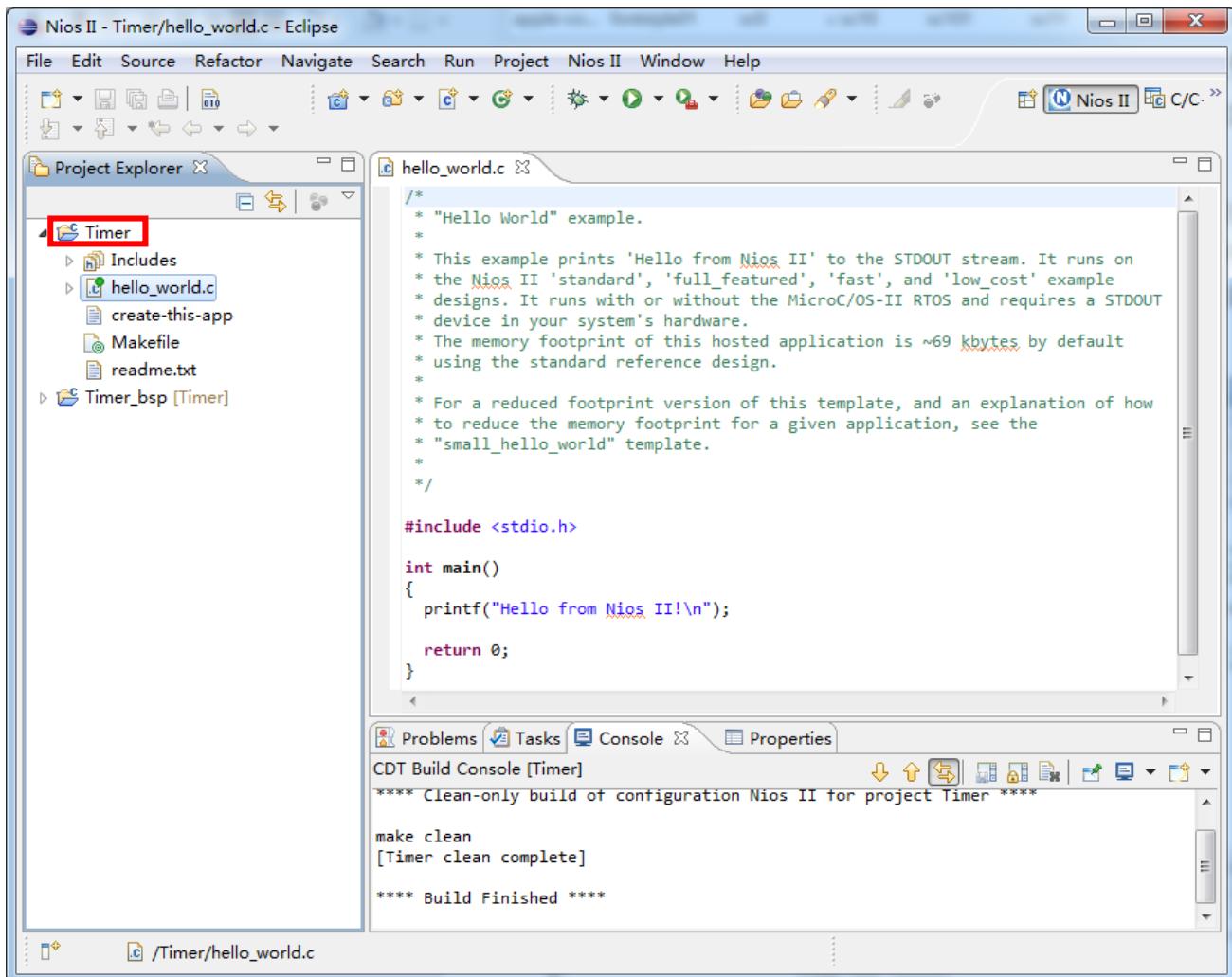


图 6.4.3 hello_world 工程代码图

由代码可知，下载程序到开发板后会在窗口上输出“Hello from Nios II!”。我们在这里要验证之前创建的 Qsys 系统是否能正常工作。验证方法是先编译 UART 工程，然后将工程模板程序下载到开发板上，看是否能正常运行。再次之前，需要先简化代码。方法和“HelloWorld”实验里一样是一样的，大家可以照着操作，这里就不再赘述了。

优化完代码之后，再编译一次 Timer 工程，会出现以下的界面。这表示编译通过，可以将程序下载到开发板上了。

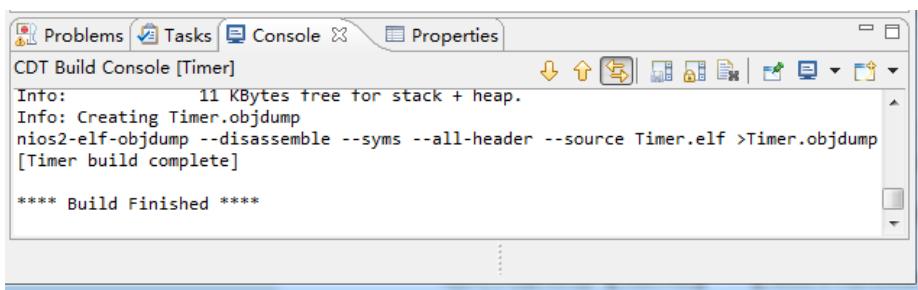


图 6.4.4 编译通过后的console窗口

这时大家点击【Run As】→【Nios II Hardware】，然后点击【Target Connection】标签，然后在 Target Connection 窗口中点击【Refresh Connections】按钮后。这时软件便会自动识别我们开发板上的 Qsys 系统，并显示 Qsys 系统的相关信息。我们接着点击【Run】，软件会把 irq.elf 文件下载至开发板中运行起来。更加详细的图和文字描述，可以在“Hello,World”实验的下载验证部分查看。

这时，若之前创建的 Qsys 系统无误，代码下载完成后在 Nios II console 窗口会显示“Hello from Nios II!”字符，如下图所示。



图 6.4.5 下载代码后的console窗口

验证完 Qsys 系统是否能正常运行之后，我们就可以开始软件部分的设计了。这时只需要在当前的代码窗口修改代码就可以了。代码如下所示。

```

1 #include "system.h"           //系统头文件
2 #include "altera_avalon_timer_regs.h" //定时器头文件
3 #include "altera_avalon_pio_regs.h" //PIO 头文件
4 #include "sys/alt_irq.h"        //中断头文件
5 #include "unistd.h"            //延迟函数头文件
6 #include <stdio.h>             //标准的输入输出头文件

```

```
7
8 alt_u8 i=0;
9 alt_u32 timer_isr_context; //定义全局变量以储存 isr_context 指针
10 void Timer_initial(void); //定时器中断初始化
11 void Timer_ISR_Interrupt(void* isr_context , alt_u32 id); //定时器中断服务子程序
12
13 int main(void)
14 {
15     alt_u8 beep = 0x0; //初始化让 beep 静音
16     Timer_initial(); //初始化定时器中断
17     printf("stand by \n");
18     while(1)
19     {
20         if(i == 1)
21         {
22             IOWR_ALTERA_AVALON_PIO_DATA(PIO_BEEP_BASE, beep); //驱动 beep 的 IO
23             beep = ~beep;
24             i = 0;
25         }
26     }
27 }
28
29 //初始化定时器中断
30 void Timer_initial(void)
31 {
32     //改写 timer_isr_context 指针以匹配 alt_irq_register() 函数原型
33     void* isr_context_ptr = (void*) &timer_isr_context;
34     //设置 PERIOD 寄存器
35     //PERIODH | PERIODL = 计数器周期因子 * 系统时钟频率因子 - 1
36     //PERIODH | PERIODL = 1s*100M - 1 = 99999999 = 0x05F5_E0FF
37     IOWR_ALTERA_AVALON_TIMER_PERIODH(TIMER_BASE, 0x05F5);
38     IOWR_ALTERA_AVALON_TIMER_PERIODL(TIMER_BASE, 0xE0FF);
39     //设置 CONTROL 寄存器
```

```
40     IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE,  
41     ALTERA_AVALON_TIMER_CONTROL_START_MSK | //0x4, START = 1, 计数器开始运行  
42     ALTERA_AVALON_TIMER_CONTROL_CONT_MSK | //0x2, CONT = 1, 计数器连续运行直到 STOP 位被置 1  
43     ALTERA_AVALON_TIMER_CONTROL_IT0_MSK); //0x1, IT0 = 1, 产生 IRQ  
44     //注册定时器中断  
45     alt_ic_isr_register(  
46     TIMER_IRQ_INTERRUPT_CONTROLLER_ID, //中断控制器标号, 从 system.h 复制  
47     TIMER_IRQ, //硬件中断号, 从 system.h 复制  
48     Timer_ISR_Interrupt, //中断服务子函数  
49     isr_context_ptr, //指向与设备驱动实例相关的数据结构体  
50     0x0); //flags, 保留未用  
51 }  
52  
53 //定时器中断函数  
54 void Timer_ISR_Interrupt(void* timer_isr_context, alt_u32 id)  
55 {  
56     //用户中断代码  
57     i = 1;  
58     //应答中断, 将 STATUS 寄存器清零, ALTERA_AVALON_TIMER_STATUS_TO_MSK=0x1  
59     IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_BASE, ~ALTERA_AVALON_TIMER_STATUS_TO_MSK);  
60 }
```

代码的第 13 行至第 27 行是主函数。主函数中第 16 行代码完成了初始化中断的任务，完整的初始化中断函数在代码的第 30 行到第 51 行。在初始化中断的时候，需要给 PERIOD 寄存器赋初值，这是在代码的第 37 行和第 38 行完成的。PERIOD 寄存器初值的计算公式已经包含在注释中了。然后是使能中断，并设置定时器的运行模式，也就是给 control 寄存器（控制寄存器）相应的位写 1，这个操作在代码的第 40 行至 43 行完成。接下来在代码的第 45 行至代码的第 50 行注册定时器中断。在主函数的第 18 行至第 26 行，当 i 等于 1 的时候，将变量 beep 中存储的值输送给 PIO 核的 data 寄存器，并驱动与蜂鸣器相连的引脚，以此来控制蜂鸣器的状态。然后将 beep 的值取反，并将 i 的值置 0。在定时器的运行过程中，每当计数到 0 的时候，会触发 IRQ。在中断函数中，首先会将 i 的值置 1，然后给 status 寄存器写 0 来应答 IRQ。中断函数的这些操作在代码的第 54 行至第 60 行完成。

修改完代码的窗口如下所示：

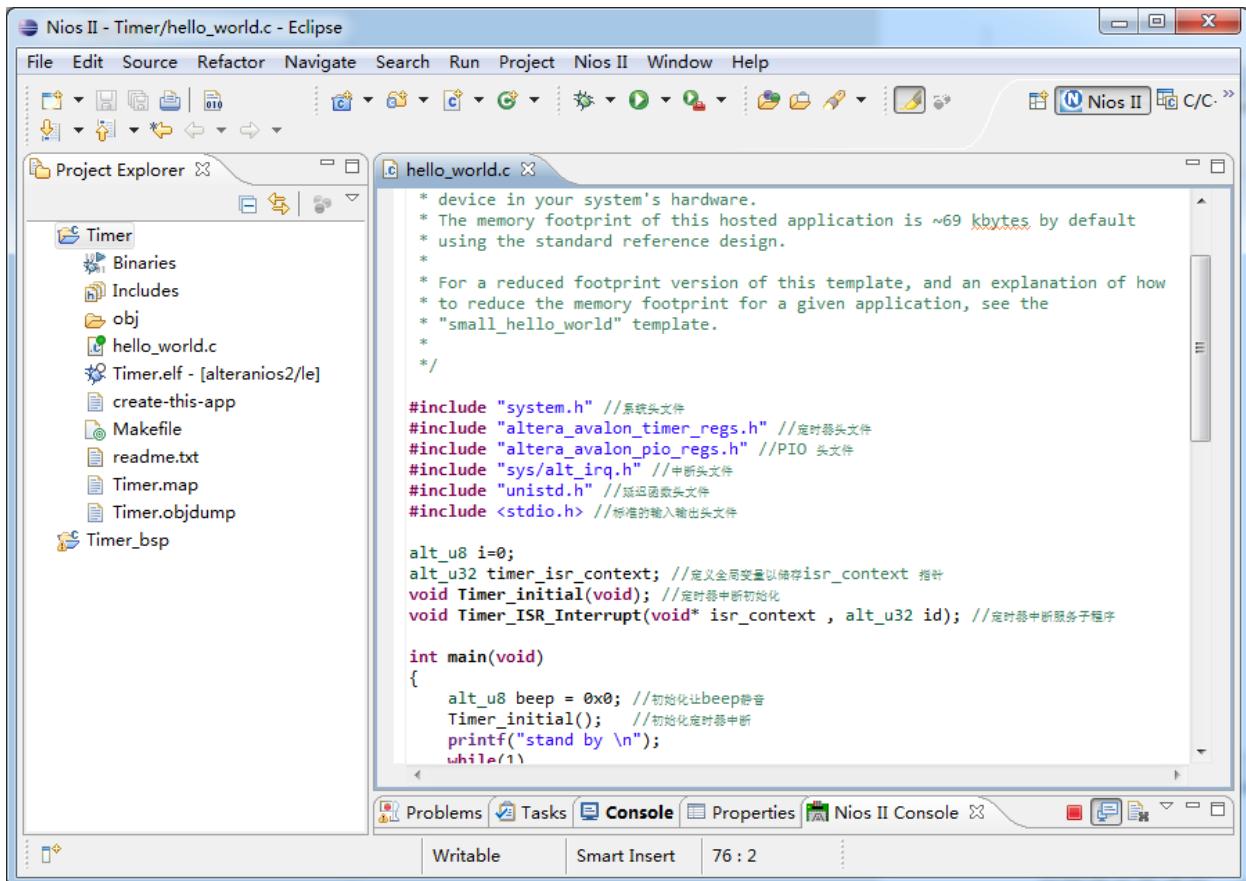


图 6.4.6 修改后的工程代码图

代码修改完成后，大家记得要点一下快捷菜单中的【Save】，或者菜单栏中的【File】→【Save】，来保存修改后的程序。

6.5 下载验证

现在可以编译 Timer 工程了。右键 Timer 工程，点击 build project。稍等片刻，Console 窗口显示的内容如下图所示，这表示工程编译成功。

```

CDT Build Console [Timer]
--quartus_project -u:work/Timer/qsys/hardware/Timer.sopcinfo
D:/work/Timer/qsys/hardware/Timer.sopcinfo
Info: (Timer.elf) 7796 Bytes program size (code + initialized data).
Info: 11 KBytes free for stack + heap.
Info: Creating Timer.objdump
nios2-elf-objdump --disassemble --all-header --source Timer.elf >
[Timer build complete]

**** Build Finished ****

```

图 6.5.1 编译工程后的console窗口图

这时大家右键 Timer 工程，点击【Run As】→【Nios II Hardware】，代码就被下载到开发板上了。然后大家就能观察到，蜂鸣器会周期性地鸣叫，说明我们本次实验下载验证成功。

第七章 SDRAM IP核

SDRAM 是一种 RAM 类型的易失性存储器件，因其具有较大的容量和相对较低的价格在嵌入式系统中应用广泛。然而应用 SDRAM 需要实现刷新操作、行列管理、不同延迟和命令序列等逻辑，控制复杂，而 Qsys 提供的 SDRAM 控制器 IP 核接口极大的方便了 SDRAM 的使用，本章我们使用 SDRAM 控制器 IP 核对 SDRAM 进行读写实验。

本章包括以下几个部分：

7.1 简介

7.2 实验任务

7.3 硬件设计

7.4 软件设计

7.5 下载验证

7.1 简介

SDRAM 控制器 IP 核能够处理所有的 SDRAM 协议要求，包括上电初始化、地址复用、刷新、读写时序等，极大的方便了 SDRAM 的使用。下面我们先来看看 SDRAM 控制器 IP 核和 SDRAM 芯片的连接框图，如下所示：

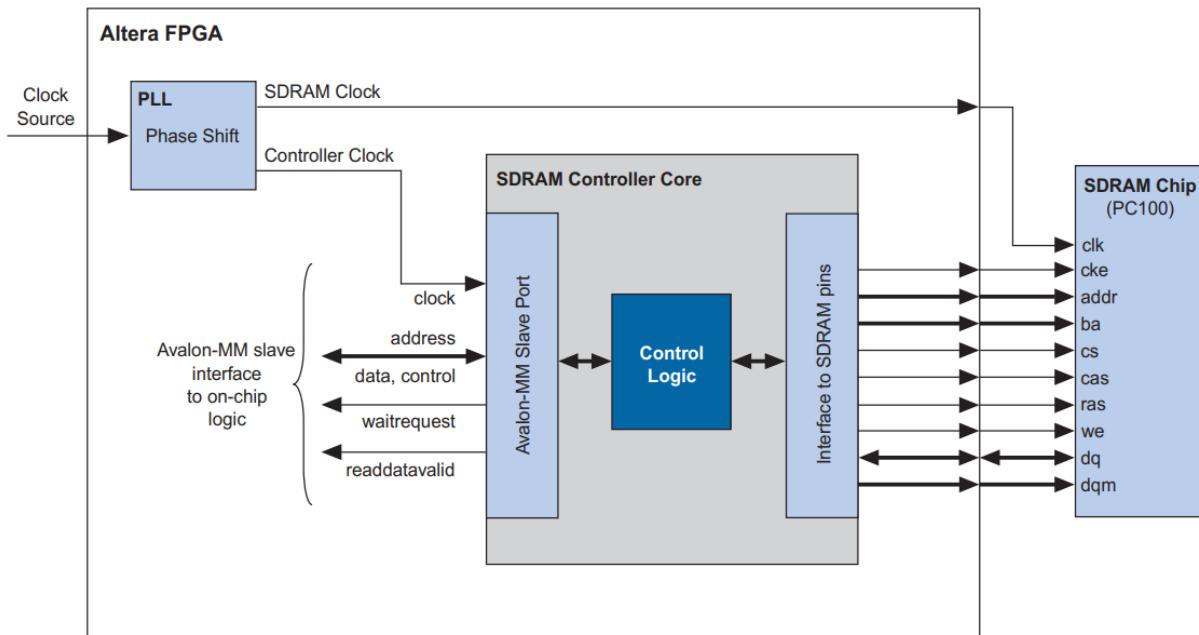


图 7.1.1 SDRAM 控制器 IP 核连接到外部 SDRAM 芯片的结构框图

从上面的这个结构框图中，我们可以看到 SDRAM 控制器 IP 核产生于 FPGA 内部，它带有接口引脚、控制逻辑、以及 Avalon 从机接口。接口引脚用来连接外部 SDRAM 芯片管脚，这些接口引脚通过 Altera FPGA 上的 I/O 引脚连接到 SDRAM 芯片管脚上。控制逻辑用来实现 SDRAM 的操作，比如，SDRAM 初始化，自刷新，突发读写等，这些全都是控制逻辑来完成的，控制逻辑不需要我们编写，当我们生成 SDRAM 控制器 IP 核之后，它会自动生成。Avalon 从机接口用来连接我们的 CPU，Avalon 从机接口是 SDRAM 控制器 IP 核中仅为用户可见的部分。从控制器端口提供一个如 SDRAM 芯片一样大的平滑、线性存储器空间。当访问从控制器端口时，SDRAM 协议的细节完全透明。Avalon 接口作为一个简单的存储器接口操作，没有存储器映射的配置寄存器。这里我们需要注意的是：SDRAM 芯片必须和 Avalon 接口一样以相同的时钟来驱动。我们可以看到图中的片内锁相环（PLL），它就是用来调整 SDRAM 控制器与 SDRAM 芯片之间的时钟相位差。在较低的时钟频率下，可能不需要 PLL。在较高的时钟频率下，当信号在引脚上有效时，需要 PLL 来调整 SDRAM 时钟。PLL 并没有包括在 SDRAM 控制器内。如果需要 PLL，设计者必须在生成 Qsys 系统模块以外手动添加 PLL。Altera FPGA 和 SDRAM 芯片的不同组合将

要求不同 PLL 的设置。

还有一点我们需要说明的是 f_{max} 性能取决于整个硬件设计。Qsys 系统模块的主控制器时钟驱动 SDRAM 控制器和 SDRAM 芯片。因此，整个系统模块的性能决定 SDRAM 控制器的性能。例如，为了实现 100MHz 的 f_{max} 性能，系统模块必须设计为 100MHz 时钟率，且 Quartus II 软件的时序分析必须检验硬件设计是否能够进行 100MHz 的操作。说完了 SDRAM 的综述之后下面我们就总结给出 SDRAM 控制器 IP 核的功能特性：

(1) SDRAM 控制器 IP 核具有不同数据宽度（8、16、32 或 64 位）、不同内存容量和多片选择等设置。

(2) SDRAM 控制器 IP 核可以全面支持符合 PC100 标准的 SDRAM 芯片。（PC100，表明时钟信号为 100，数据读写速率也为 100）

(3) SDRAM 控制器 IP 核可选择与其他的片外 Avalon 三态器件共用地址和数据总线，该特性在 I/O 引脚资源紧张的系统中很有用。

我们可以在 Qsys 中使用 SDRAM IP 核的配置向导来指定硬件特性和仿真特性。SDRAMIP 核配置向导有两个选项卡：Memory Profile 和 Timing，如下图所示。

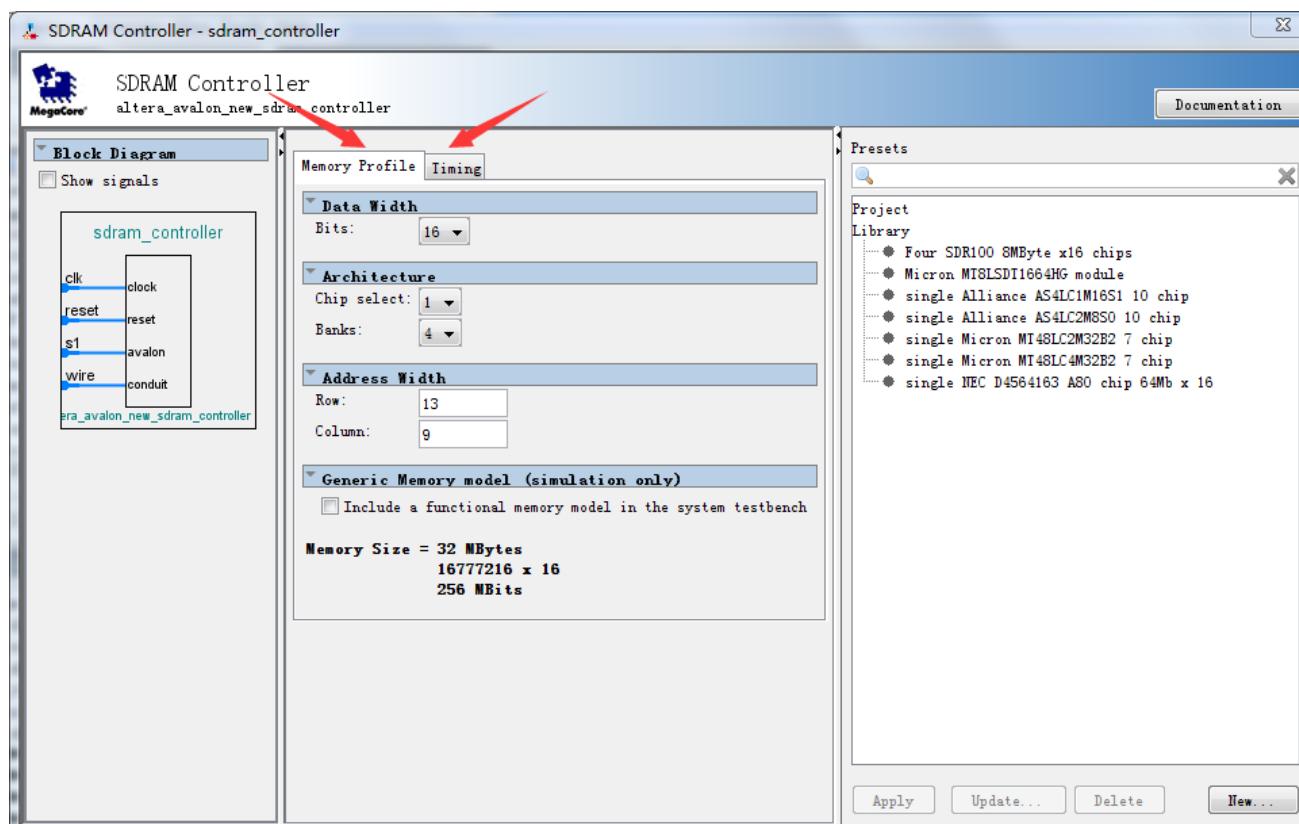


图 7.1.2 SDRAM控制器IP核的配置选项

为了使用方便，Presets 列表提供几个预定义的 SDRAM 配置。如果实际使用的 SDRAM 芯片型号与列表中的一致，可直接选用而不用设置其他选项。选择不同的预配置，SDRAM IP 核将自动改变 Memory Profile 和 Timing 选项卡上的值来匹配指定的配置。如果实际使用的 SDRAM 芯片与列表中的不相同，则需要设计者根据 SDRAM 芯片数据手册的参数来设置 Memory Profile 和 Timing 标签上的值，改变任何选项卡上的配置设置转变 Preset 值为 custom。

当然我们也可以将我们配置好的 SDRAM 参数添加到预定义的 SDRAM 配置，在今后的使用过程中我们就直接选择我们添加的预定义的 SDRAM 配置。接下来我们就来简单的介绍一下 Memory Profile 和 Timing。

(1) Memory Profile 选项卡

Memory Profile 选项卡允许设计者指定 SDRAM 的结构，例如地址和数据总线宽度、片选信号的数目和区的数目等。Memory Profile 选项卡设置项如下表所示。

表 7.1-1 Memory Profile 选项卡设置

设置		允许值	默认值	描述
数据宽度		8,16,32,64	32	SDRAM 数据总线宽度。该值确定 dq (数据) 总线和 dqm 总线的宽度。
结构 设置	片选	1,2,4,8	1	SDRAM 芯片的数目。通过使用多个片选信号，SDRAM 控制器可组合多个 SDRAM 芯片为一个存储器子系统。
	Banks	2,4	4	Bank 的数量。该值确定连接到 SDRAM 的 ba (Bank 地址) 总线宽度。
地址 宽度 设计	行	11,12,13,14	12	行地址线的宽度。该值确定 addr 总线的宽度。
	列	≥8 且小于行的值	8	列地址线的宽度。
功能存储器 模块	—	—	—	当打开选项时，Qsys 创建 SDRAM 芯片的功能仿真模型，仅用于仿真。

这些参数值可参照使用的 SDRAM 手册来设置。通过 Memory Profile 选项卡上的设置后，消息框以兆字节、兆 bit 位以及可寻址的字长显示 SDRAM 预期的内存容量。将这些预期值与选择的 SDRAM 的实际大小相比较可以检验设置是否正确。说完了 Memory Profile 选项卡，接下来我们看看 Timing 选项卡。

(2) Timing 选项卡

Timing 选项卡允许设计者设置 SDRAM 芯片的时序规范。正确值在 SDRAM 芯片数据手册中提供。Timing 选项卡上可用的设置如下表所示。

表 7.1-2 Timing 选项卡设置

设置	允许值	默认值	描述
CAS 等待时间	1,2,3	3	读命令到数据输出的等待时间（以时钟周期计算）
初始化刷新周期	1~8	2	复位后，该值指定 SDRAM 控制器将执行多少个刷新周期作为初始化序列的一部分
每隔一段时间执行一个刷新命令	—	15.625us	该值指定 SDRAM 控制器多久刷新一次 SDRAM。典型的 SDRAM 每 64ms 需要 4096 个刷新命令，通过每 $64\text{ms}/4096=15.625\mu\text{s}$ 执行一个刷新命令以符合此要求
上电后、初始化前的延时	—	100us	从稳定的时钟和电源到 SDRAM 初始化的延时
刷新命令(t_rfC)的持续时间	—	70ns	自动刷新周期
预充电命令(t_rp)的持续时间	—	20ns	预充电命令周期
Active 到读或写的延时(t_rcd)	—	20ns	ACTIVE 到读或写的延时
访问时间(t_ac)	—	17ns	时钟边沿后的访问时间。该值由 CAS 的等待时间决定
写恢复时间(t_wr, 无自动预充电)	—	14ns	如果执行了明确的预充电命令，写恢复。该 SDRAM 控制器总是执行明确的预充电命令

我们需要注意的是无论我们输入的精确时序值如何，每个参数实现的实际时序将为 Avalon 时钟的整数倍。对于每隔一段时间执行一个刷新命令的参数，实际时序将不超出目标值，而其他所有参数，实际时序将大于或等于目标值。

7.2 实验任务

本章我们利用官方 SDRAM ControllerIP 核实现对 SDRAM 的读写操作。

7.3 硬件设计

本章实验的硬件框架如下图所示：

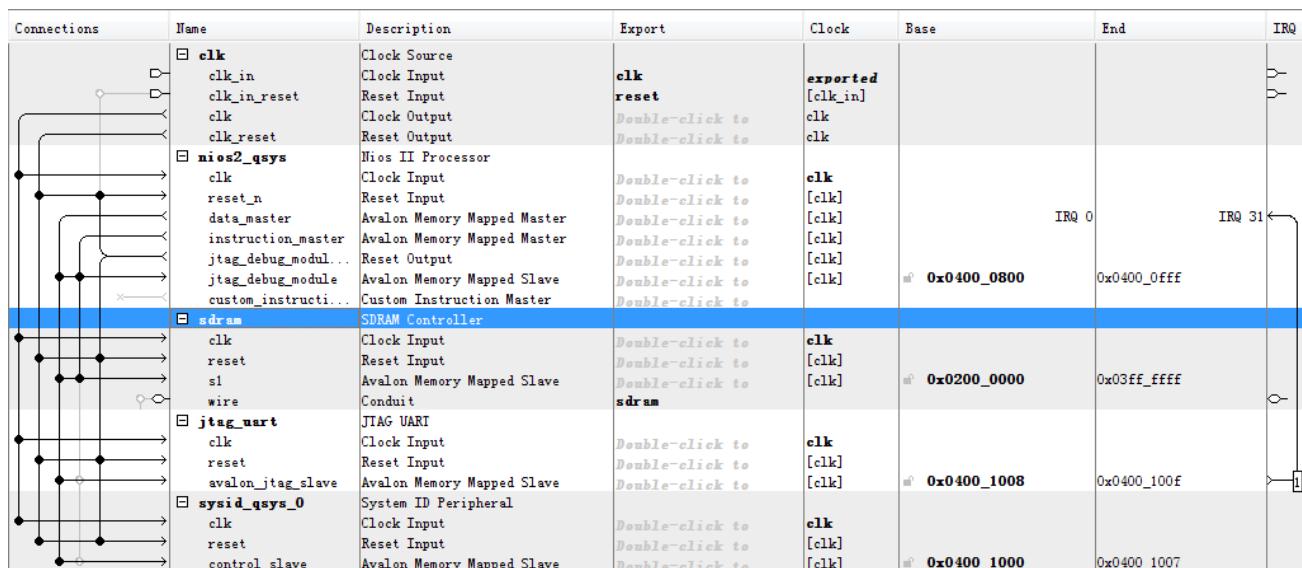


图 7.3.1 SDRAM 实验的硬件框架图

图中，我们要把 clk IP 核的时钟频率设置为 100MHz。

另外需要注意的是，Nios II IP 核需要将复位向量 Reset Vector 和异常向量 Exception Vector 都设置为 SDRAM，如下图所示：

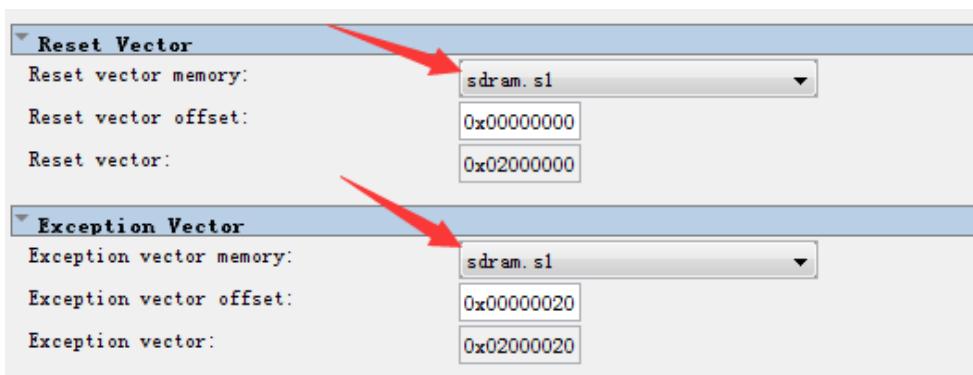


图 7.3.2 设置 Nios II 的复位向量和异常向量

现在我们主要来介绍一下新添加的 SDRAM IP 核，按照新起点使用的 SDRAM 型号为

W9825G6KH 的 datasheet，配置如下图所示。

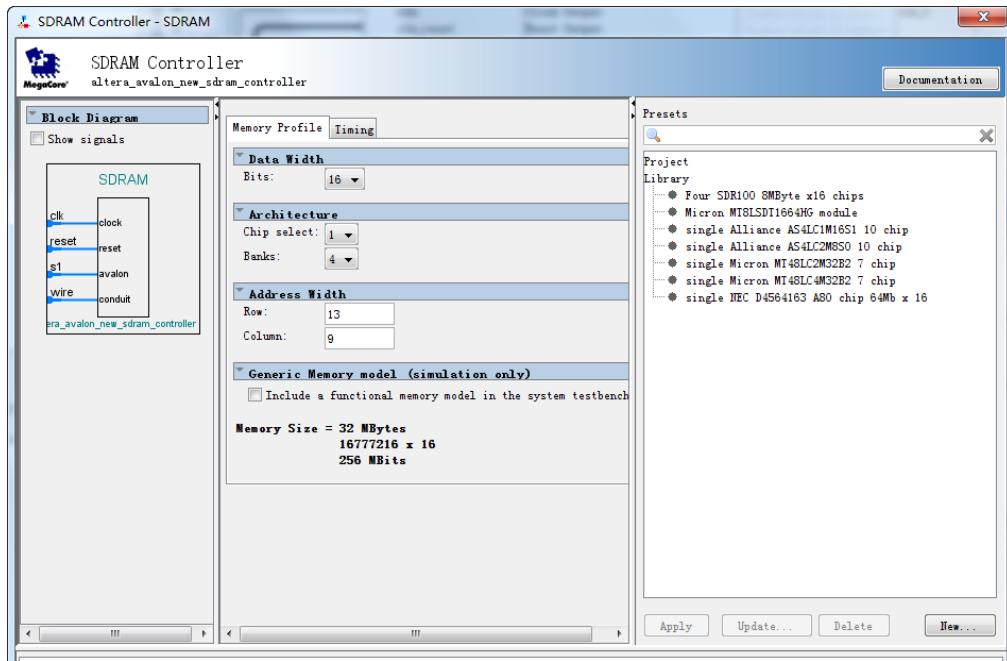


图 7.3.3 SDRAM控制器IP核的Memory Profile配置页面

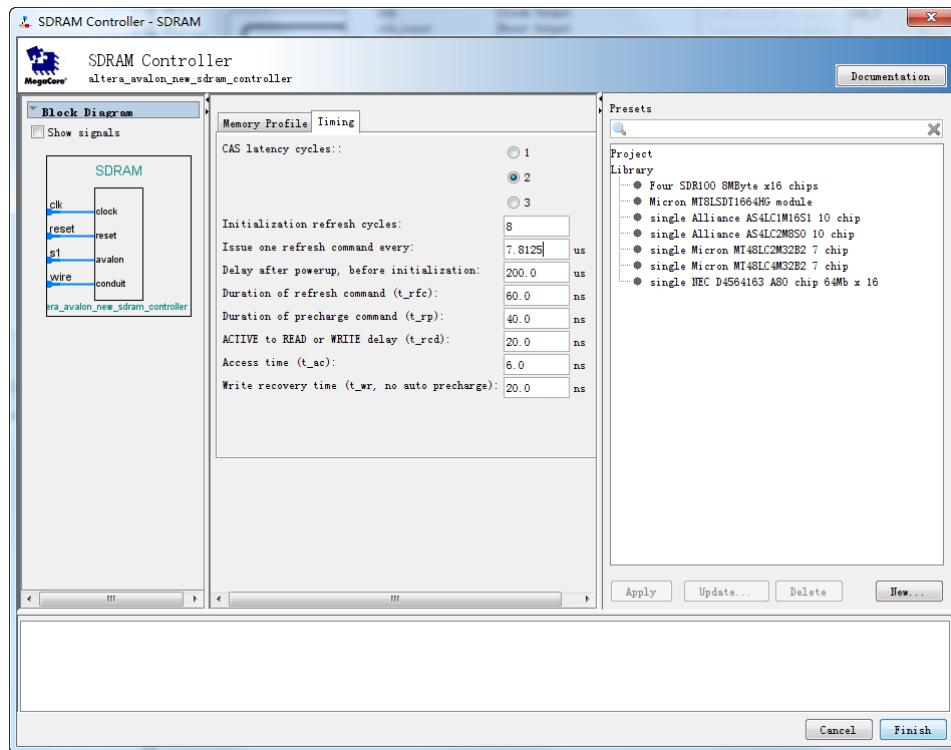


图 7.3.4 SDRAM控制器IP核的Timing配置页面

为了方便大家以后的使用，下面我们就简单为大家讲解一下如何将自己的 SDRAM 配置添

加至 Library 中。当我们配置好 SDRAM 以后，我们可以在窗口的右下方找到【New】按钮并点击，弹出如下图所示页面。

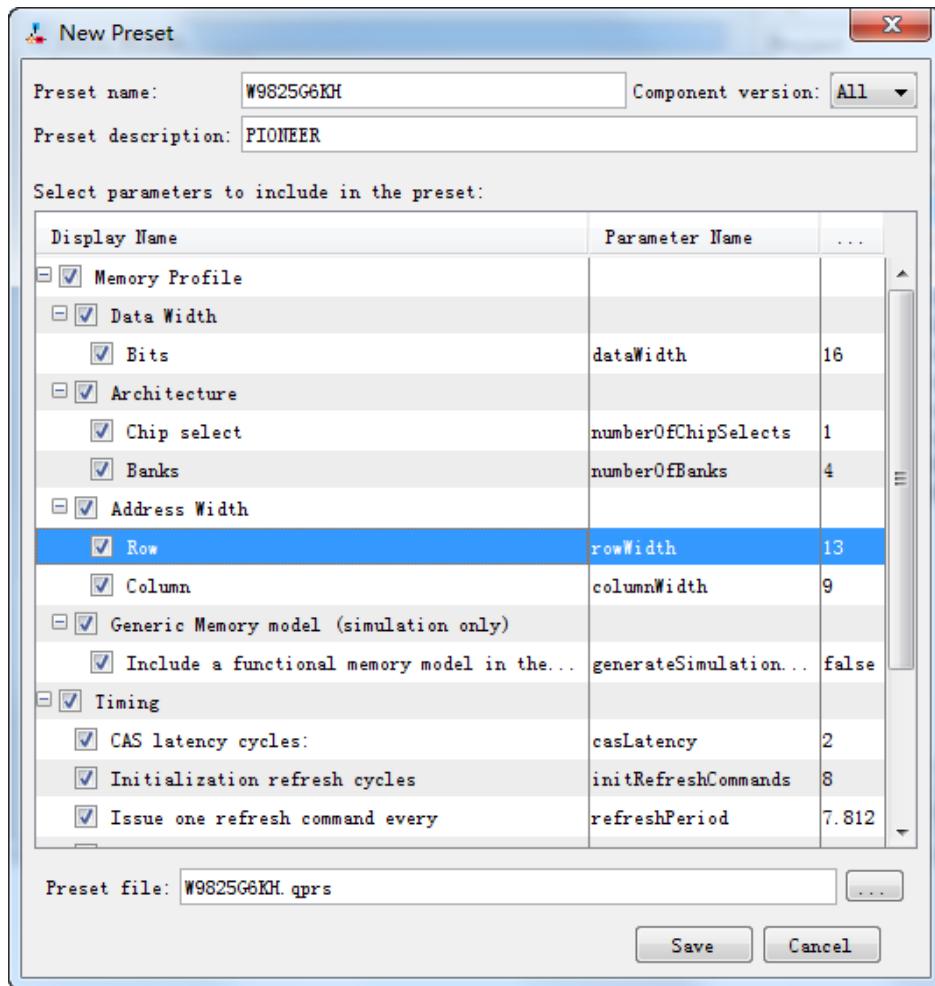


图 7.3.5 将 SDRAM 配置添加至Library中

在该页面中，我们将 Preset name 和 Preset description 填写好以后，我们就可以点击【Save】按钮，弹出如下图所示提示窗口。

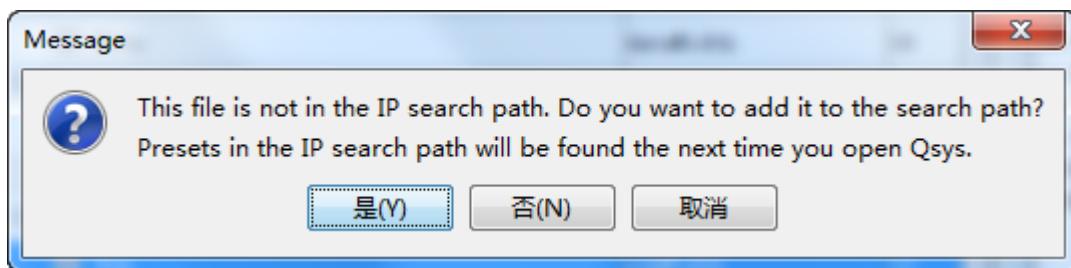


图 7.3.6 路径添加提示窗口

在该提示窗口中我们选择是，这时我们就可以在 Library 中看到我们添加的 SDRAM 配置

了。然后我们重新打开 Qsys 软件，这时，我们就可以在 SDRAM 的 Library 中看到我们添加的 W9825G6KH。最后我们再补充说明一点，SDRAM 为动态存储器对时序要求比较高，由于 FPGA 内部有延迟，所以 PLL 输出 100Mhz 时钟频率给 SDRAM_SCLK 时，PLL 时钟需要设置相位偏移，相位偏移我们设置为-75deg。

顶层代码如下：

```
1 module qsys_sdram(
2     //module clock
3     input      sys_clk      ,      // 时钟信号
4     input      sys_rst_n   ,      // 复位信号（低有效）
5
6     //SDRAM interface
7     output      sdram_clk,      // SDRAM 芯片时钟
8     output      sdram_cke,      // SDRAM 时钟有效
9     output      sdram_cs_n,     // SDRAM 片选
10    output      sdram_ras_n,    // SDRAM 行有效
11    output      sdram_cas_n,    // SDRAM 列有效
12    output      sdram_we_n,     // SDRAM 写有效
13    output      [ 1:0] sdram_ba,    // SDRAM Bank 地址
14    output      [12:0] sdram_addr,  // SDRAM 行/列地址
15    inout      [15:0] sdram_data,   // SDRAM 数据
16    output      [ 1:0] sdram_dqm,    // SDRAM 数据掩码
17     //user interface
18 );
19
20 //wire define
21 wire      clk_100m;           // SDRAM 控制器时钟
22 wire      clk_100m_shift;     // 相位偏移时钟
23 wire      rst_n ;            // 复位信号（低有效）
24 wire      locked;           // PLL 输出有效标志
25
26 //*****
27 /*          main code
```

```
28 //*****
29
30 assign sdram_clk = clk_100m_shift;      // SDRAM 驱动时钟
31
32 //例化 PLL
33 pll_clk u_pll_clk(
34   .inclk0 (sys_clk),
35   .c0     (clk_100m),
36   .c1     (clk_100m_shift)
37 );
38
39 //SDRAM 控制器顶层模块, 封装成 FIFO 接口
40 sdram u_sdram (
41   .clk_clk           (clk_100m),    // SDRAM 控制器驱动时钟
42   .reset_reset_n    (sys_rst_n),   // 复位信号(低有效)
43   .sdram_addr       (sdram_addr),  // SDRAM 行/列地址
44   .sdram_ba         (sdram_ba),    // SDRAM Bank 地址
45   .sdram_cas_n     (sdram_cas_n), // SDRAM 列有效
46   .sdram_cke        (sdram_cke),   // SDRAM 时钟有效
47   .sdram_cs_n      (sdram_cs_n),  // SDRAM 片选
48   .sdram_dq         (sdram_data), // SDRAM 数据
49   .sdram_dqm        (sdram_dqm),   // SDRAM 数据掩码
50   .sdram_ras_n     (sdram_ras_n), // SDRAM 行有效
51   .sdram_we_n      (sdram_we_n),  // SDRAM 写有效
52 );
53
54 endmodule
```

从顶层代码可以看到，我们主要例化 PLL 和 SDRAM 控制器，PLL 生成两个 100MHz 的时钟，其中一个偏移-75 度用于驱动 SDRAM 芯片。

7.4 软件设计

本实验的软件工程代码如下：

```
1 #include <stdio.h>      //标准输入输出头文件
2 #include "system.h"      //系统头文件
3 #include "alt_types.h"   //数据类型头文件
4 #include "string.h"
5
6 //SDRAM 地址
7 alt_u8 *ram = (alt_u8 *) (SDRAM_BASE + 0x10000);
8
9 //-----
10 //-- 名称    : main()
11 //-- 功能    : 程序入口
12 //-- 输入参数 : 无
13 //-- 输出参数 : 无
14 //-----
15
16 int main(void) {
17     int i;
18     memset(ram, 0, 100);
19     //向 ram 中写数据, 当 ram 写完以后, ram 的地址已经变为(SDRAM_BASE+0x10000+200)
20     for(i=0;i<100;i++) {
21         *(ram++) = i;
22     }
23     //逆向读取 ram 中的数据
24     for(i=0;i<100;i++) {
25         printf("%d ",*(--ram));
26     }
27     return 0;
28 }
```

在代码中，首先定义了一个 `aut_u8` 型的指针 `ram` 指向 SDRAM 的基地址+`0x10000`，之后我们改变或读取指针指向的地址（SDRAM 基地址+偏移地址）的值，就改变了 SDRAM 相应地址（偏移地址/2）的值。在主函数中，我们通过 `memset` 函数将从 `ram` 指向地址开始的 100 个地址的值全部清 0，再通过一个 `for` 循环向从 `ram` 指向地址开始的 100 个地址的赋相应的值，最后再将这 100 个值逆向读取打印出来，这样就完成了 SDRAM 的读写操作。可以看出，通过

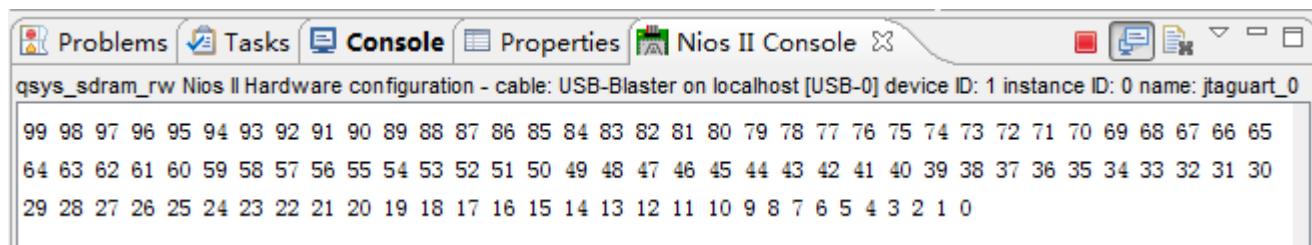
SDRAM 控制器的使用，对 SDRAM 的读写操作变得非常简单。

之所以对 SDRAM 的读写要偏移 0x10000，是因为 CPU 程序的运行占用了从 SDRAM 基址开始的部分内存，如果我们不做偏移直接从基地址开始读写，则很有可能破坏程序正常运行，0x10000 这个值并不固定，只要别占用程序运行的内存就可以了。

7.5 下载验证

讲完了软件工程，接下来我们就要在开拓者开发板上验证本次实验结果。首先我们需要在 Quartus II 软件中将 `qsys_sdram.sof` 文件下载到开发板中；然后在 Eclipse 软件中将 `qsys_sdram_rw.elf` 文件下载进去。

`qsys_sdram_rw.elf` 下载完成以后，我们的 C 程序会自动运行，同时在 Nios II Console 界面会显示程序的打印信息，如图 7.5.1 所示。图中可以看到从 SDRAM 中读出的数据为 99 到 0，与我们写入的数据一致，说明本次实验下载验证成功。



```
qsys_sdram_rw Nios II Hardware configuration - cable: USB-Blaster on localhost [USB-0] device ID: 1 instance ID: 0 name: jtaguart_0
99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77 76 75 74 73 72 71 70 69 68 67 66 65
64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30
29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

图 7.5.1 SDRAM IP核实验结果

如果大家在下载elf文件的过程中工具提示错误，如下图所示：

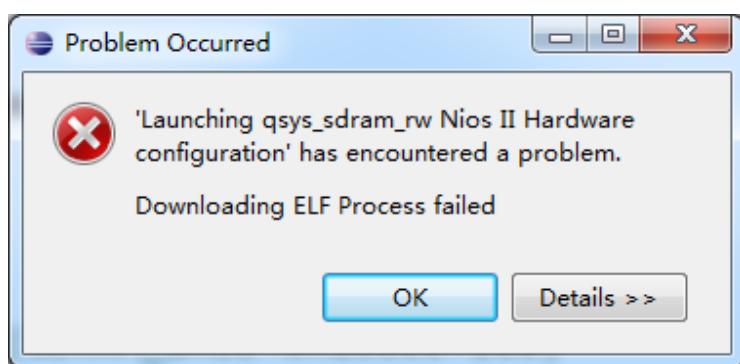


图 7.5.2 下载elf文件过程中报错

我们留意到在下载过程中，Console 会提示如下图所示的信息，说地址“0x2000020”到“0x200D757”之间验证错误。

```

Problems Tasks Console Properties Nios II Console
qsys_sdram_rw Nios II Hardware configuration [Nios II Hardware] nios2-download (18-11-6 上午10:32)

Downloading 02000020 ( 0%)
Downloaded 54KB in 1.0s (54.0KB/s)

Verifying 02000020 ( 0%)
Verify failed between address 0x2000020 and 0x200D757
Leaving target processor paused

```

图 7.5.3 下载过程中的提示信息

错误的这段地址并不固定，但它们在 Qsys 系统中刚好处于 SDRAM 的地址范围内。此时，我们可以通过以下方式解决下载报错的问题。

在 Eclipse 中右击应用工程“qsys_sdram_rw”，在弹出的菜单中选择“Run As”
→ “Run Configurations”，会弹出“Run Configurations”配置页面，如下图所示：

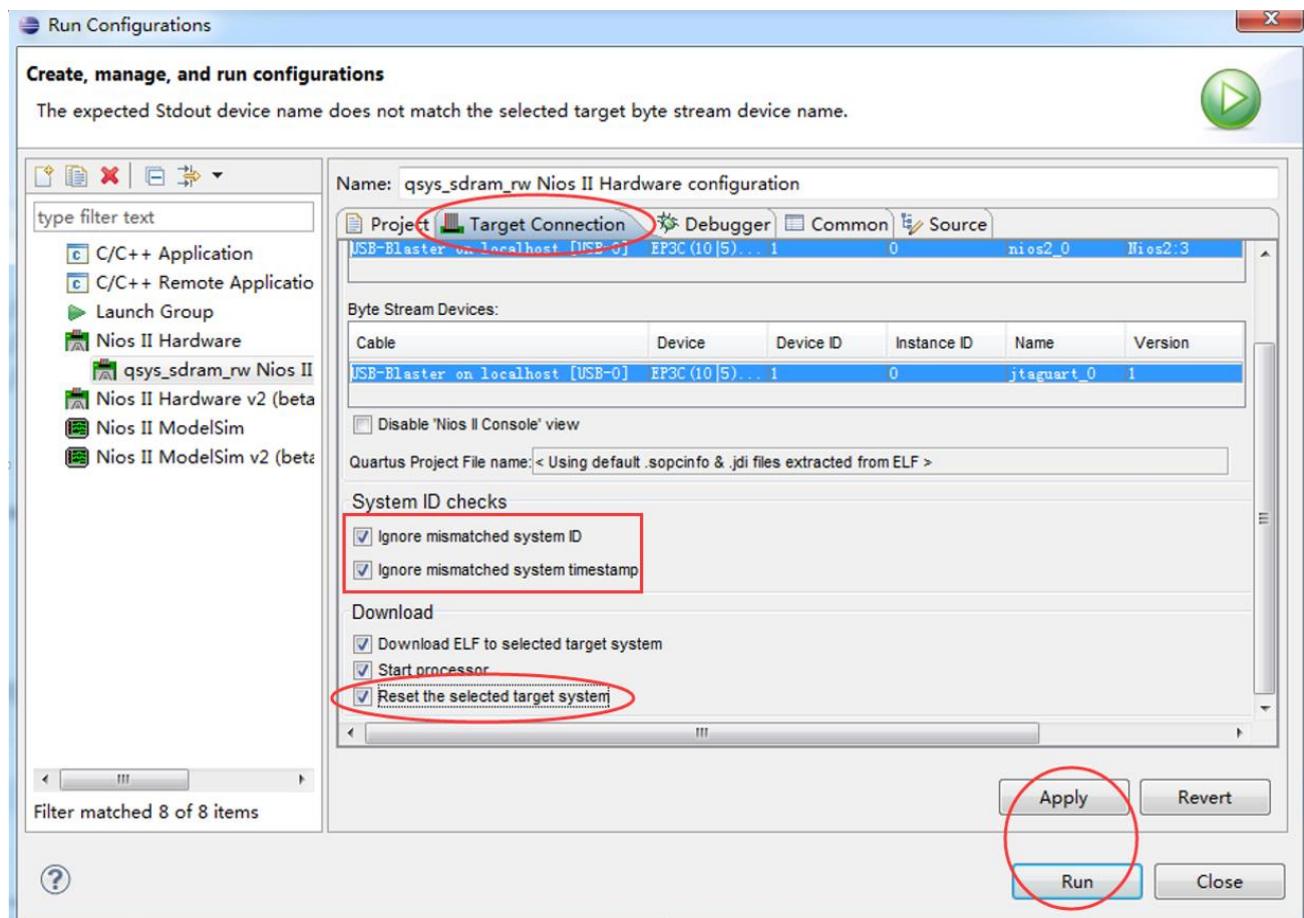


图 7.5.4 Run Configurations 配置页面

在上图所示页面中的“Target Connection”标签页最下方，勾选“Reset the selected

target system”。在上图中，我们同时勾选了“Ignore mismatched system ID”以及“Ignore mismatched system timestamp”。

设置完成后，点击“Apply”，最后点击“Run”来重新下载elf文件，这样在下载的过程中就不会报错了。

第八章 EPCS IP核

Altera 公司的 FPGA 芯片大多采用 EPCS 系列存储器作为配置数据的存储空间，带 Avalon 接口的 EPCS IP 核允许 Nios II 系统访问 EPCS 串行配置器件。Altera 提供集成到 Nios II 硬件抽象层（HAL）系统库的驱动程序，允许用户使用 HAL 应用程序接口（API）来读写 EPCS 器件。本章我们通过对 EPCS 的读写操作来学习 EPCS IP 核的使用。

本章包括以下几个部分：

- 8.1 简介
- 8.2 实验任务
- 8.3 硬件设计
- 8.4 软件设计
- 8.5 下载验证

8.1 简介

我们的新起点开发板使用的 Flash 为采用 SPI 协议的串行 Flash，大小为 16Mbit。内部分为 32 个扇区（sector），每个扇区有 256 页（page），每页有 256 个字节。该 Flash 兼容 EPCS，可以使用 EPCS IP 核，通过 Nios II 系统对 Flash 执行以下操作：

(1) 在 Flash 器件中存储程序代码。EPCS 控制器自带 Boot-Loader 代码，因此 Nios II 系统允许用户在 Flash 器件中存储程序代码。

(2) 存储非易失性数据，例如串行号、NIC 号和其他需要长久储存的数据。

(3) 管理 FPGA 配置数据。Flash 可存储 FPGA 的配置数据，并在上电时自动完成对 FPGA 的配置。具有网络接口的嵌入式系统可从网络上接收新的 FPGA 配置数据，并通过 EPCS 控制器将新的配置数据下载到 EPCS 串行配置器件中。下面我们先来看一下 EPCS IP 核和 EPCS 芯片的连接框图，如下图所示：

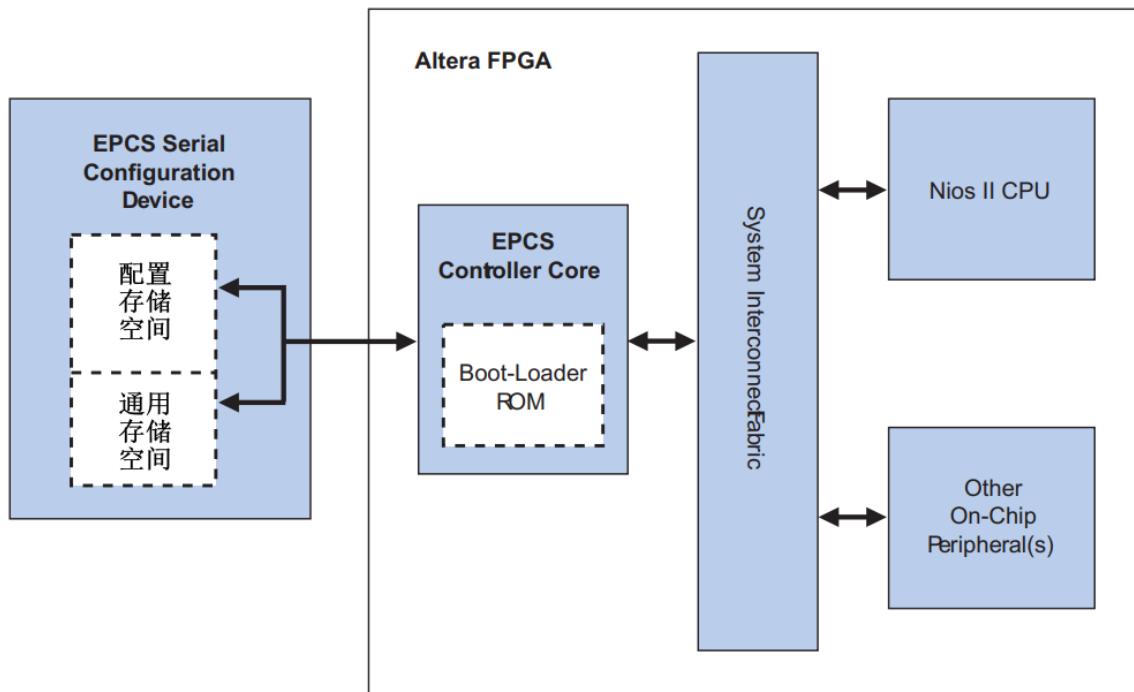


图 8.1.1 EPCS控制器IP核连接到外部EPCS芯片的结构框图

从上面的这个结构框图中，我们可以看到 EPCS Flash 器件的存储空间被分成了两个独立的区域：配置存储空间和通用存储空间。配置存储空间主要用于存储 FPGA 配置数据，如果 FPGA 配置数据没有填满整个 EPCS 器件，那么剩下空间可以存储用户非易失性数据即通用存储空间。

EPCS 控制器包含一个用于存储 Bootloader 程序的片上 ROM 存储器，当 EPCS 控制器与 Cyclone 和 Cyclone II 器件一起使用时，EPCS 控制器需要 512 字节的 Bootloader 的 ROM 存储空间。当 EPCS 控制器与 Cyclone III、Cyclone IV、Stratix II 等器件一起使用时，EPCS 控制器需要 1024 字节（也就是 1KB）的 Bootloader 的 ROM 存储空间。如果把 Nios II 处理器的复位地址放在 EPCS 控制器 IP 核的基址处，可以让 Nios II 处理器从 EPCS 控制器 IP 核开始引导，在这种情况下，复位后 CPU 首先执行引导 EPCS 控制器的 Bootloader ROM 中的代码，把存储在 EPCS 中通用内存区域的数据复制到指定的 RAM 存储器，然后把系统控制权转移给存储在 RAM 中的程序。实现这些操作的程序代码无需用户编写，由 Nios II SBT for Eclipse 软件自动生成。Nios II SBT for Eclipse 软件提供了编程 EPCS 的工具并编译产生用于存储在 EPCS 中文件的程序代码。

EPCS 控制器有一个 Avalon-MM 从接口，这个接口提供访问 Bootloader 代码和控制寄存器的能力。下表给出了 EPCS IP 核的寄存器描述。

表 8.1-1 EPCS IP核的寄存器描述

偏移地址	寄存器名称	读/写	描述
0x00~0xFF	Boot ROM Memory	读	Boot Loader 代码
0x100	Read Data	读	
0x101	Write Data	写	
0x102	Status	读/写	
0x103	Control	读/写	未公布
0x104	Reserved	—	
0x105	Slave Enable	读/写	
0x106	End of Packet	读/写	

从 EPCS IP 核的寄存器描述表格中我们可以看出：

(1) 偏移地址 0x00~0xff：这 256 个字 (32bits) 是专用于存储 Bootloader 代码，当 Nios II 系统的复位地址指向 EPCS 控制器时，处理器便会从 EPCS 控制器基地址开始的 256 个字存储空间读取 Bootloader 代码。并且 EPCS 控制器包含有一个中断信号，当把程序代码全部加载到指定的 RAM 中时产生中断请求。

(2) 偏移地址 0x100-0x106：这 7 个字节是 EPCS 控制和数据寄存器，由于 Altera 没有公布 EPCS IP 核的寄存器用法，要访问 EPCS 器件，用户必须使用 Altera 提供的 HAL 驱动程序。

现在我们来看一下 Qsys 中 EPCS IP 核有哪些配置选项。打开 Qsys 后，我们在 Library 下

的搜索框中输入“flash”如下图所示：

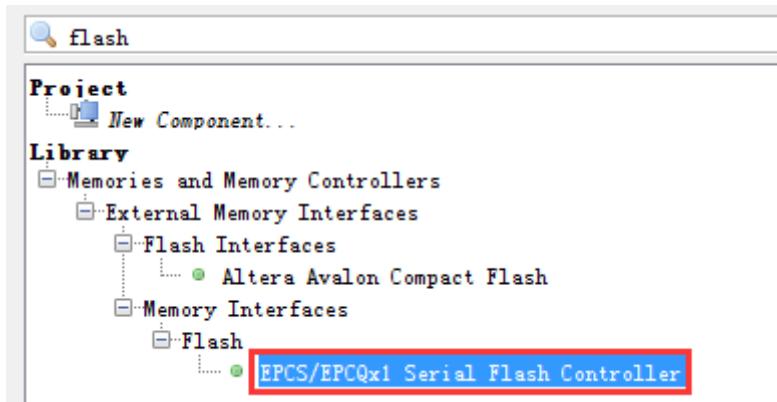


图 8.1.2 搜索flash

双击红框所圈的 EPCS/EPCQx1 Serial Flash Controller，弹出如下图所示界面：

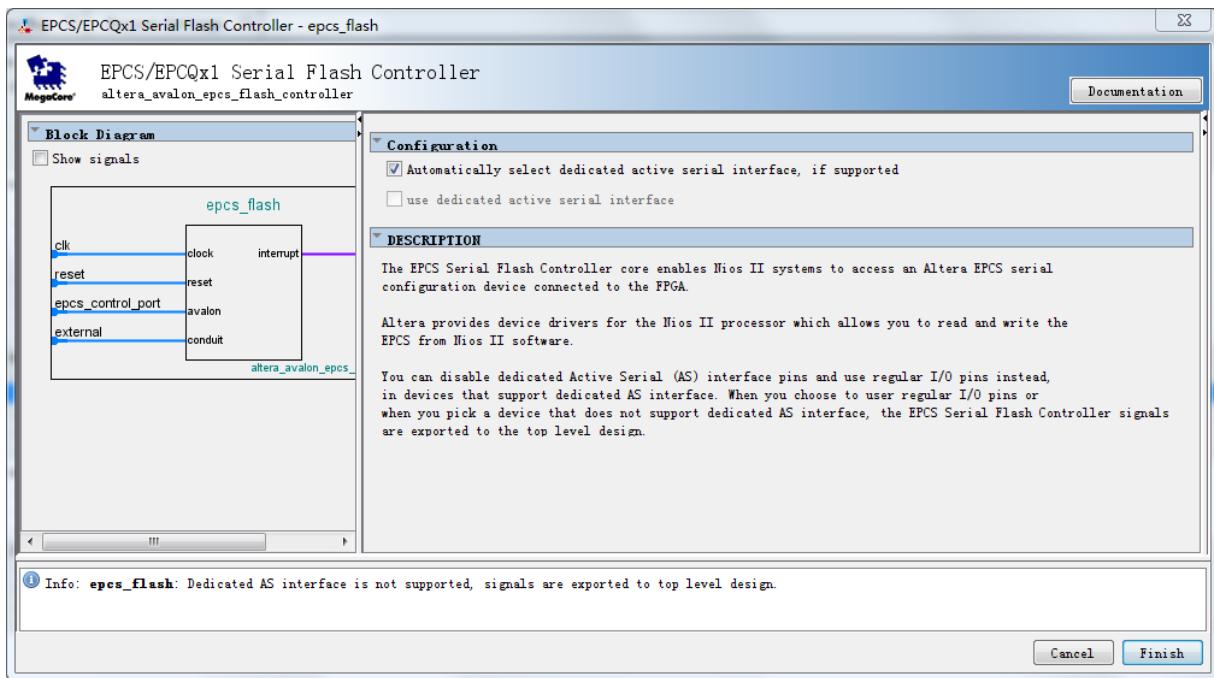


图 8.1.3 EPCS FLASH IP核的配置选项

对于配置选项只有“Automatically select dedicated active serial interface, if supported.”这一选项可选，由于我们的新起点开发板使用 JTAG 接口进行下载配置，所以也可以不选，选上也不会有影响。其它的不需要任何设置，我们只要将 EPCS IP 核添加到 Qsys 软件中就可以使用了。

8.2 实验任务

本章的实验任务是使用官方提供的 EPCS IP 核实现对 EPCS Flash 的读写操作

8.3 硬件设计

本章实验的硬件框架如下图所示：

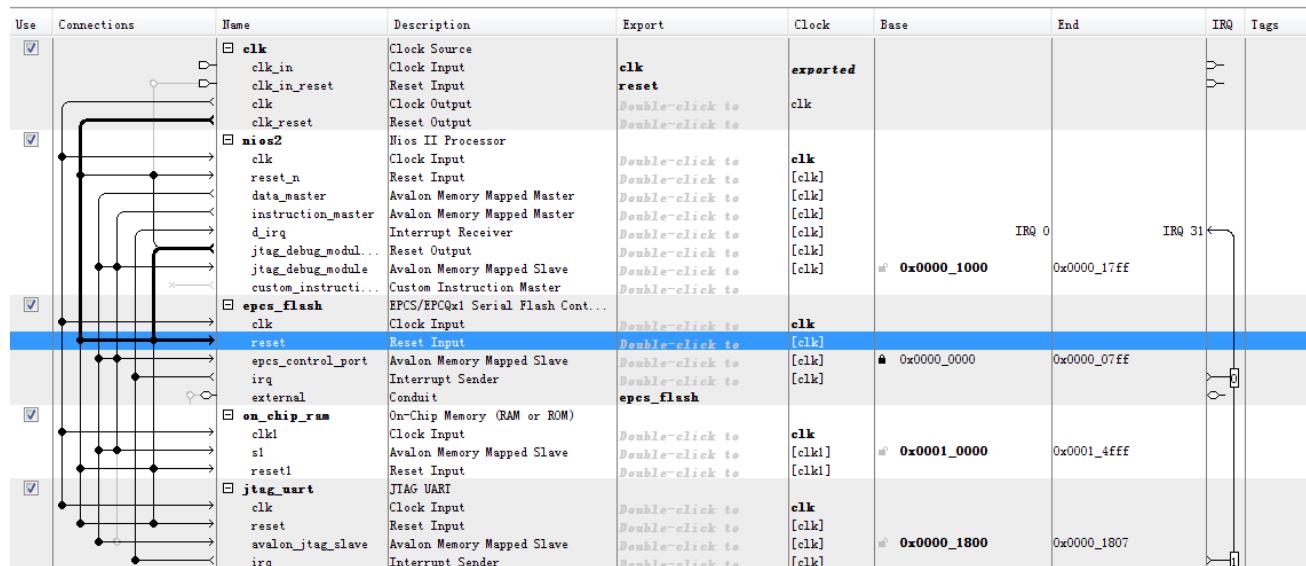


图 8.3.1 EPCS IP核实验的硬件框架图

图中， clk IP 核的时钟频率设置为 50MHz。而新添加的 EPCS IP 核，保持默认设置即可。

需要注意的是，此时 Nios II IP 核的复位向量 Reset Vector 和异常向量 Exception Vector 的设置如下图所示：

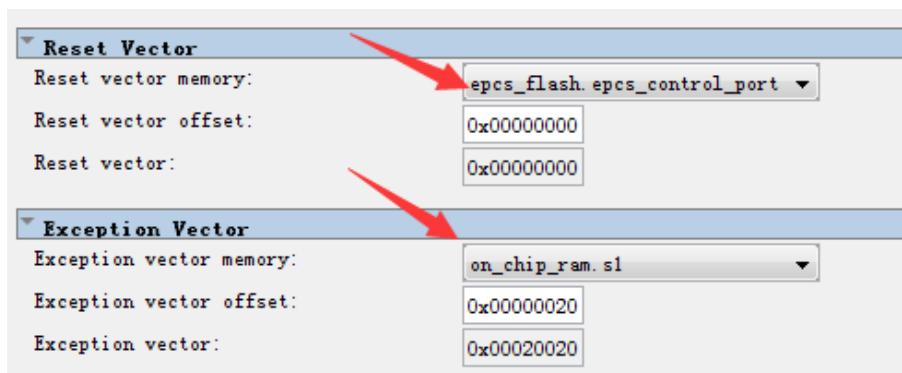


图 8.3.2 Nios II的复位向量和异常向量设置

现在我们在顶层代码中对 Qsys 系统进行例化，如下所示：

```
1 module qsys_epcs_rw(
2     //module clock
3     input      sys_clk      ,      // 时钟信号
4     input      sys_rst_n    ,      // 复位信号(低有效)
5
6     //EPCSinterface
7     output     epcs_flash_dclk ,      // EPCS FLASF 的驱动时钟
8     output     epcs_flash_sce   ,      // 片选信号
9     output     epcs_flash_sdo   ,      // 数据输出
10    input      epcs_flash_data0 // 数据输入
11
12    //user interface
13 );
14
15 //*****
16 //**          main code
17 //*****
18
19 //例化 EPCS 硬件框架
20 qsys_EPCS u_qsys_epcs(
21     .clk_clk      (sys_clk      ),      // 驱动时钟
22     .reset_reset_n (sys_rst_n    ),      // 复位信号(低有效)
23     .epcs_flash_dclk (epcs_flash_dclk ),      // EPCS FLASF 的驱动时钟
24     .epcs_flash_sce  (epcs_flash_sce  ),      // 片选信号
25     .epcs_flash_sdo  (epcs_flash_sdo  ),      // 数据输出
26     .epcs_flash_data0(epcs_flash_data0) // 数据输入
27 );
28
29 endmodule
```

从顶层代码可以看到，我们主要完成对 EPCS 硬件框架的例化。其中管脚分配如下：

表 8.3-1 EPICS IP核实验管脚分配

信号名	方向	管脚	端口说明
sys_clk	input	E1	系统时钟信号
sys_rst_n	input	M1	系统复位，低有效
epcs_flash_dclk	output	H1	EPICS FLASH 的驱动时钟
epcs_flash_sce	output	D2	片选信号
epcs_flash_sdo	output	C1	数据输出
epcs_flash_data0	input	H2	数据输入

8.4 软件设计

在介绍代码之前，我们先了解一些关键信息。

创建好软件工程后，我们打开 system.h，找到 EPICS 配置模块，如下图所示：

```
/*
 * epcss_flash configuration
 *
 */

#define ALT_MODULE_CLASS_epcs_flash altera_avalon_epcs_flash_controller
#define EPCS_FLASH_BASE 0x0
#define EPCS_FLASH_IRQ 0
#define EPCS_FLASH_IRQ_INTERRUPT_CONTROLLER_ID 0
#define EPCS_FLASH_NAME "/dev/epcs_flash"
#define EPCS_FLASH_REGISTER_OFFSET 1024
#define EPCS_FLASH_SPAN 2048
#define EPCS_FLASH_TYPE "altera_avalon_epcs_flash_controller"
```

图 8.4.1 EPICS配置模块

要注意红框中的 EPCS_FLASH_NAME，我们会用到它。接下来我们认识一下几个用于 EPCS_FLASH 的函数：

- ◆ 打开 flash 函数： alt_flash_open_dev()

函数的功能为打开 Flash，原型为

```
alt_flash_fd* alt_flash_open_dev(const char* name);
```

name 是 Flash 的名字，就是上面提到的 EPCS_FLASH_NAME，返回值的类型是 alt_flash_fd 类型的句柄，该类型在 alt_flash.h 文件中声明：返回值为 0 表示打开成功，否则不成功。

◆ 获取 Flash 信息函数: alt_epcs_flash_get_info()

函数的功能为获取 Flash 信息, 原型为

```
int alt_epcs_flash_get_info(alt_flash_fd* fd, flash_region** info, int*
    number_of_regions );
```

其中 alt_flash_fd* fd 为 Flash 的句柄, int* number_of_regions 为 region 的数量, flash_region** info 为 Flash 的信息, 是一种结构体类型, 包含的信息如下

```
typedef struct flash_region
{
    int offset;           //地址偏移
    int region_size;     //region 大小
    int number_of_blocks; //Block 数量
    int block_size;       //Block 大小
}flash_region;
```

◆ 块擦除函数: alt_epcs_flash_erase_block()

函数的功能为擦除选择块, 原型为

```
int alt_epcs_flash_erase_block(alt_flash_dev* flash_info, int block_offset);
```

alt_flash_dev* flash_info 为函数的句柄, 同 alt_flash_fd* fd, int block_offset 为块偏移量, 即要擦除的块在 flash 中的地址。

◆ Flash 写函数: alt_epcs_flash_write()

函数的功能为向 Flash 指定地址写入数据, 原型为

```
int alt_epcs_flash_write(alt_flash_dev* flash_info, int offset, const void* src_addr,
    int length);
```

int offset 为写入的起始偏移地址, const void* src_addr 为写入数据的存放地址, int length 为写入数据的字节长度。

◆ Flash 读函数: alt_epcs_flash_read()

函数的功能为从 Flash 指定地址读取数据, 原型为

```
int alt_epcs_flash_read(alt_flash_dev* flash_info, int offset, void* dest_addr, int length);
```

int offset 指定读取 Flash 数据的偏移地址, void* dest_addr 为读取后的数据存放的地址, int length 指定读取的字节长度。

现在我们来看一下本实验的软件工程代码, 如下:

```
1 #include <stdio.h>
2 #include <string.h>
3 #include "system.h"
4 #include "sys/alt_flash.h"
5 #include "altera_avalon_epcs_flash_controller.h" //EPCS 相关函数头文件
6
7 #define DATA_SIZE 100
8
9 int main(void)
10 {
11     flash_region* regions;
12     alt_flash_fd* fd;
13     int number_of_regions;
14     int i, ret_code;
15     char data_wr[DATA_SIZE];
16     char data_rd[DATA_SIZE];
17
18     printf("写入 flash 的数据 ");
19     for (i=0;i<DATA_SIZE;i++) {
20         data_wr[i] = i;           //初始化 data_wr 数组
21         printf("%d, ", data_wr[i]); //将 data_wr 数组中的值打印到控制台
22     }
23     printf("\n");
24     //打开 EPCS_FLASH 器件, 获取 EPCS_FLASH 器件句柄
25     fd = alt_flash_open_dev(EPCS_FLASH_NAME);
26     if(!fd) {
27         printf("Can't open flash device\n");
28     } else{
29         //成功打开 EPCS_FLASH 器件, 并获取 EPCS_FLASH 器件信息
```

```
30     ret_code = alt_epcs_flash_get_info(fd, &regions, &number_of_regions);
31 }
32
33 if(!ret_code) {
34     //擦除第九个 Block 的内容
35     alt_epcs_flash_erase_block(fd, regions->offset+(regions->block_size)*8);
36     //把 data_wr 数组的数据写入第九个 Block
37     alt_epcs_flash_write (fd, regions->offset+(regions->block_size)*8,
38                           data_wr, DATA_SIZE);
39     //读取写入第九个 Block 的数据
40     lt_epcs_flash_read(fd, regions->offset+(regions->block_size)*8, data_rd, DATA_SIZE);
41
42     printf("从 flash 读取到的数据 ");
43     for(i=0;i<DATA_SIZE;i++) {
44         printf("%d, ",data_rd[i]);
45     }
46     printf("\n");
47 } else{
48     printf("Can't getEPCSflash device info"); //没有获得 EPCS_FLASH 信息
49 }
50
51 alt_flash_close_dev(fd); //关闭 EPCS_FLASH 器件
52 return 0;
53 }
```

在代码中，首先宏定义了 DATA_SIZE，由来指定向 Flash 写入和读取数据的长度，然后初始化写入 Flash 数据的数组 data_wr。接着我们获取 EPCS_FLASH 的句柄，并判断是否获得。如果正确获得，就读取 EPCS_FLASH 的器件信息，如果没有，就打印相应的信息。在获得器件信息的情况下，我们向第 9 块 block 写入 data_wr 数据。写入数据之前，先擦除该块的信息，写入完成后，读取写入的信息放入 data_rd 数据中并打印到控制台。最后关闭 FLASH。

8.5 下载验证

讲完了软件工程，接下来我们就将该实验下载至我们的新起点开发板进行验证，首先我们

需要在 Quartus II 软件中将 qsys_epcs_rw.sof 文件下载至我们的新起点开发板, qsys_epcs_rw.sof 下载完成后, 我们还需要在 Eclipse 软件中将 qsys_epcs_rw.elf 文件下载至我们的新起点开发板, sdram_rw.elf 下载完成以后, 我们的 C 程序将会在我们的新起点开发板上执行, 最后, 在 Nios II Console 界面显示出代码运行结果, 如下图所示。我们可以看到从 Flash 读取到的数据和写入 Flash 的数据一致。至此, 我们的 EPICS IP 核实验就完成了。

```
qsys_epcs_rw Nios II Hardware configuration - cable: USB-Blaster on localhost [USB-0] device ID: 1 instance ID: 0 name: jtaguart_0
写入Flash的数据
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98
, 99,
从Flash读取到的数据
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98
, 99,
```

图 8.5.1 实验结果

第九章 自定义IP核—数码管

NiosII是一个嵌入式软核处理器，除了可以根据需要任意添加已经提供的各种外设外，用户还可以通过定制自定义IP核的方式来满足各种应用需求。定制IP核是使用NiosII嵌入式软核处理器的一个重要特征。定制的IP核能够以“硬件加速器”的形式实现各种各样用户要求的功能。本章我们通过自定义数码管IP核来学习如何自定义IP核。本章包括以下几个部分：

- 9.1 简介
- 9.2 实验任务
- 9.3 硬件设计
- 9.4 软件设计
- 9.5 下载验证

9.1 简介

自定义 IP 核之前我们先来看一下在系统中 Nios II 是如何与各种外设进行交互的，如下图所示。

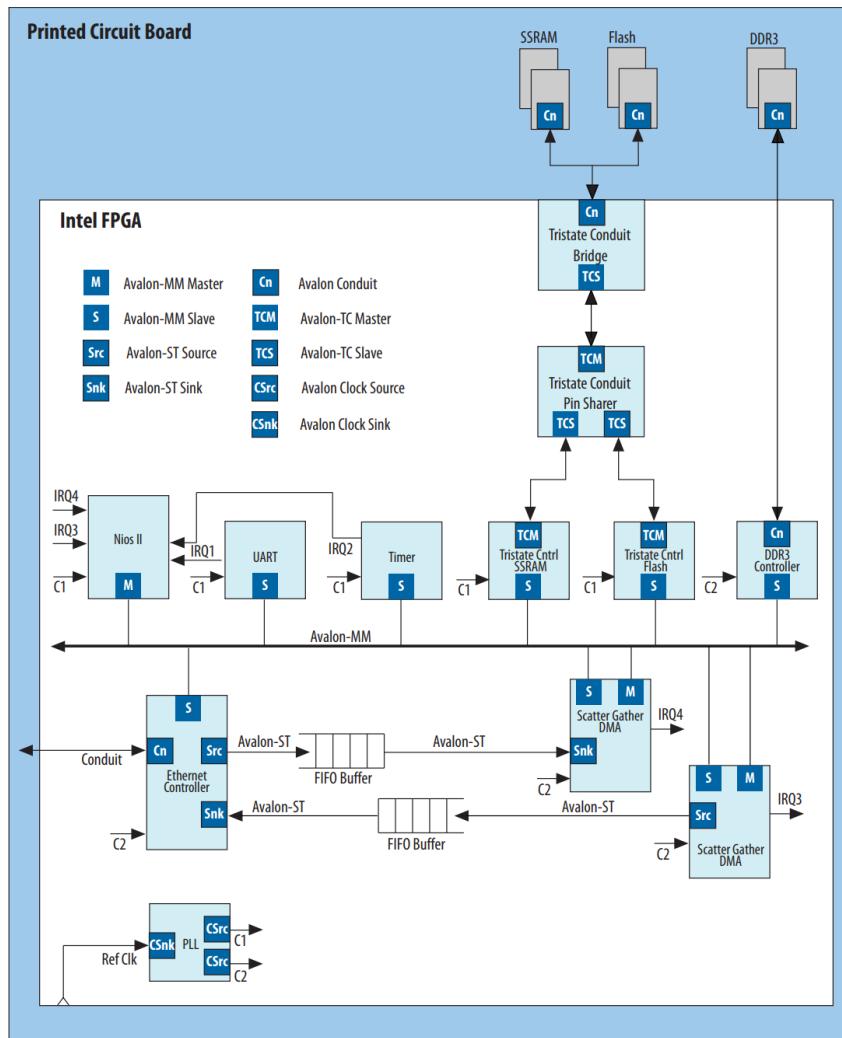


图 9.1.1 Nios II 与外设的交互

在该图中，我们可以看到 Nios II 处理器通过使用 Avalon-MM 总线接口操作片上外设的控制和状态寄存器与外设进行交互。Avalon-MM 总线接口的主端连到 Nios II 和 DMA，从端与内部各个 IP 核相连接。IP 核通过其它 Avalon 总线接口与内部 IP 和外设进行交互。如 Scatter gather DMA 通过 Avalon-ST 接口发送和接收数据。PLL 通过 Avalon Clock Sink 接口获得了一个输入时钟并产生两个输出时钟等。可见如果自定义 IP 核需要了解 Avalon 总线。

一、Avalon 总线接口规范

在 Altera FPGA 中 Avalon 允许你简单地连接组建来简化系统设计，Avalon 接口适用于高

速数据流，读写寄存器，存储器，以及控制片外设备。这些标准接口在 Qsys 中有效地设计到组件中。你可以在你定制的组件中使用这些标准化的接口来增强你设计的兼容性。在 Avalon 口规范中，定义了以下七个接口：

(1)Avalon Clock Interface：发送和接受时钟的接口。所有 Avalon 接口都是同步的。

(2)Avalon reset Interface：复位的接口。

(3)Avalon Memory Mapped Interface (Avalon-MM)：基于地址读写典型的主从连接关系的接口。

(4)Avalon Interrupt Interface：允许组件到信号事件与其他组件的接口。

(5)Avalon Streaming Interface (Avalon-ST)：支持单向数据流，包括复用流、数据包、DSP 数据。

(6)Avalon Tri-State Conduit Interface (Avalon-TC)：支持连接到片外外围设备的接口。多重外围设备可以通过信号复用（多路传输）来分享引脚，减少使用 FPGA 引脚数和 PCB 上的导线。

(7)Avalon Conduit Interface：适应不能适合任何其余 Avalon 类型的个别信号或信号组的接口类型。你可以连接在 Qsys 系统里面的 conduit 接口，或者你可以输出它们，以连接到设计中的其他模块，或者 FPGA 的引脚。

1. Avalon Clock

首先我们介绍的是 Avalon Clock 接口，Avalon Clock 接口定义了一个时钟或多个时钟用于一个组件。组件可以有时钟输入，时钟输出，或两者都有。例如，相位锁相环（PLL），它一个时钟输入和多个时钟输出，如下图所示。

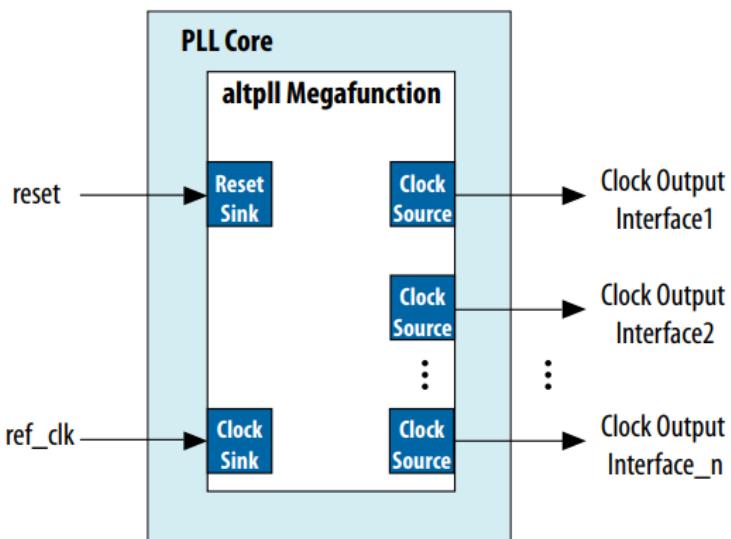


图 9.1.2 PLL 的时钟输入与输出

从该图中我们可以看出，Clock Sink 是输入信号，Clock Source 是输出信号，下面我们就对这两个信号分别进行介绍：首先我们介绍的是 Clock Sink 输入信号。Clock Sink 为其他接口和内部逻辑提供一个同步时钟。Clock Sink 信号类型，如下表所示。

表 9.1-1 Clock Sink 信号类型

信号	宽度	方向	必选	描述
clk	1	Input	Yes	一个为内部逻辑和其他接口提供同步的时钟信号

看完了 Clock Sink 信号类型，我们再来看下 Clock Sink 信号属性，如下表所示。

表 9.1-2 Clock Sink 的信号属性

名称	默认值	理论值	描述
clockrate	0	0- $2^{32}-1$	指明 clock sink 接口的时钟频率（单位：Hz）。如果为 0，时钟频率不受限，如果不为 0 并且连接的时钟源不是指定的频率，那么 Qsys 则发出警告

说完了 Clock Sink，接着我们再来看下 Clock Source，Clock Source 类型，如下表所示。

表 9.1-3 Clock Source 信号类型

信号	宽度	方向	必选	描述
clk	1	Output	Yes	一个输出时钟信号

看完了 Clock Source 信号类型，我们再来看下 Clock Source 信号属性，如下表所示。

表 9.1-4 Clock Source 信号属性

名称	默认值	理论值	描述
associatedDirectClock	N/A	一个输入时钟名称	如果存在，输入时钟直接驱动该时钟输出
clockRate	0	$0 \text{--} 2^{32} - 1$	输出时钟的频率（单位：Hz）
clockRateKnown	false	true, false	表示时钟频率是否已知，如果已知，这个信息在系统中可用于自定义其他组件。

2. Avalon Reset

说完了 AvalonClock，接下来我们再来看看 Avalon Reset，Reset Sink 信号类型，如下表所示。

表 9.1-5 Reset Sink 信号类型

信号	宽度	方向	必选	描述
reset, reset_n	1	Input	Yes	复位信号，reset 高电平有效；reset_n 低电平有效
reset_req	1	Input	No	复位请求信号

看完了 Reset Sink 信号类型，我们再来看下 Reset Sink 信号属性，如下表所示。

表 9.1-6 Reset Sink 信号属性

名称	默认值	理论值	描述
associatedClock	N/A	一个时钟名称	同步到该接口的时钟名称
synchronous-Edges	DEASSERT	NONE DEASSERT BOTH	指明要求的复位信号输入的同步类型。NONE：不要求同步。 DEASSERT：异步复位，同步释放。 BOTH：同步复位，同步释放

说完了 Reset Sink，接着我们再来看下 Reset Source, Reset Source 信号类型，如下表所示。

表 9.1-7 Reset Source信号类型

信号	宽度	方向	必选	描述
reset, reset_n	1	Output	Yes	复位信号, reset 高电平有效; reset_n 低电平有效
reset_req	1	Output	可选	复位请求信号

看完了 Reset Source 信号类型，我们再来看下 Reset Source 信号属性，如下表所示。

表 9.1-8 Reset Source信号属性

名称	默认值	理论值	描述
associatedClock	N/A	一个时钟名称	同步到该接口的时钟名称
associatedDirectReset	N/A	一个复位名称	通过一对一的连接直接驱动这个 reset source 的 reset 输入
associatedResetSinks	N/A	一个复位名称	指定最终使 reset source 有效 reset 的 reset 输入
synchronousEdges	DEASSERT	NONE DEASSERT BOTH	指明要求的复位信号输出的同步类型。NONE: 异步; DEASSERT: 异步复位, 同步释放。BOTH: 同步复位, 同步释放

3. Avalon-MM Interfaces

Avalon-MM Interfaces 是 Avalon Memory-Mapped Interfaces 的简称，即 Avalon 存储器映射接口。Avalon-MM 接口是一种交换式总线，具有良好的数据交换特性和很高的总线带宽。由于 Avalon-MM 接口是针对 Qsys 设计的，所以 Avalon-MM 接口具有结构简单，采用全同步时序，以及可以灵活地配置等特点，其运行时钟、总线位宽、各个接口位宽以及各个外设之间的互联特性等都可以灵活地配置。

一个 Avalon-MM 外设可以包含任意的信号类型，这取决于它与外设逻辑接口的需求，但外设的每个信号都要指定一个有效的 Avalon-MM 信号类型，以确定该信号的作用。Avalon-MM 信号类型可以分为从端口和主端口信号两类，这取决于 Avalon-MM 端口是主端口还是从端口。对于某些信号类型，主端口和从端口中可能都包含，但由于端口类型不同、这些信号的行为可能有所不同。每个单独的主端口或从端口使用的信号类型由外设的设计决定。Avalon-MM 主

端口或从端口的每个信号都准确地对应与一种 Avalon-MM 信号类型。对于每种信号类型，Avalon-MM 端口都只能具有一个信号类型。Avalon-MM 接口信号可配置，对于特定的 Avalon-MM 外设，并不是所有 Avalon-MM 信号都必须用到，外设设计者可以根据需要只使用必须的信号类型，从而降低系统的复杂性。例如，一个只用于输出的 16 位的通用 I/O 外设，如下图所示。

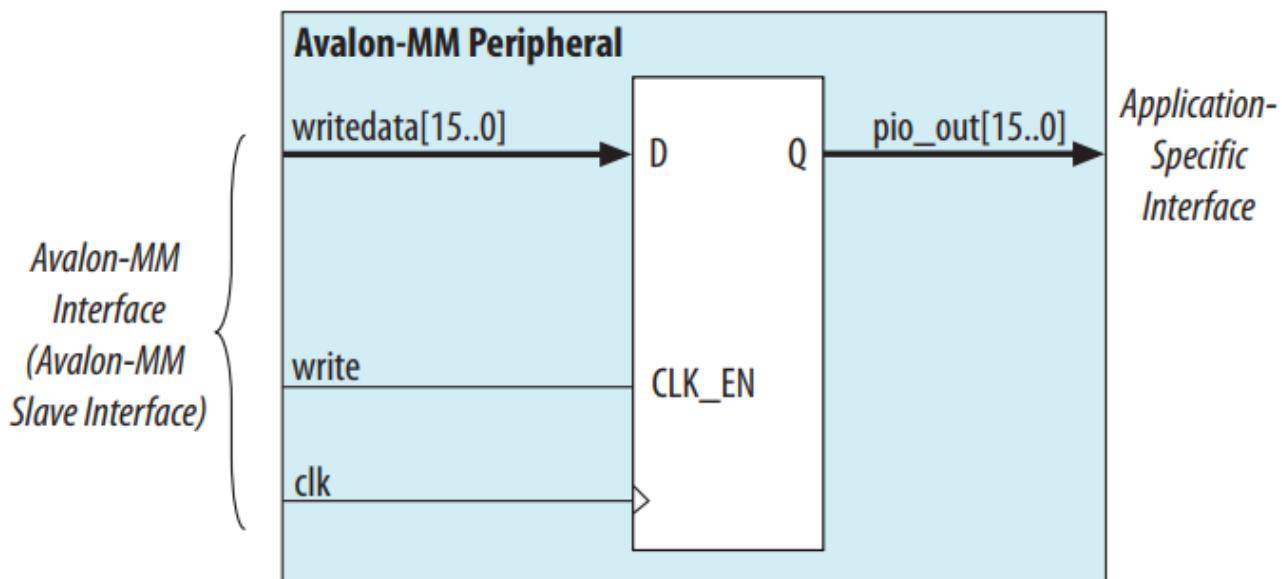


图 9.1.3 只用于输出的 16 位通用 I/O 外设

从该图中我们可以看出，这个简单的 Avalon-MM 外设只写信号和写数据信号，没有用到读信号和读数据信号。

Avalon-MM 总线的传输方式是一种主从式的传输方式，即由一个主控端外设发起并控制传输过程，而从属端外设响应经由总线模块发来的信号完成整个传输。我们看看 Avalon-MM 接口信号类型，Avalon-MM 接口信号类型，如下表所示。

表 9.1-9 Avalon-MM 接口信号类型

信号	宽度	方向	描述
基础信号			
address	1-64	主→从	读写操作地址
byteenable byteenable_n	2, 4, 8, 16, 32, 64, 128	主→从	字节使能信号
debugaccess	1	主→从	允许正常写保护的内存被写入
read	1	主→从	读信号

read_n			
readdata	8, 16, 32, 64, 128, 256, 512, 1024	从→主	读出的数据
response [1:0]	2	从→主	响应信号
write write_n	1	主→从	写信号
writedata	8, 16, 32, 64, 128, 256, 512, 1024	主→从	写入的数据
等待信号			
lock	1	主→从	信号锁存
waitrequest waitrequest_n	1	从→主	表示无法响应读写操作
流水线信号			
readdatavalid readdatavalid_n	1	从→主	表示读的数据有效
writerespon sevalid	1	从→主	有效的写应答信号
突发信号			
burstcount	1–11	主→从	突发长度
beginburstt ransfer	1	互连→从	表明开始突发传输

Avalon-MM 信号属性有很多，下面我们介绍一下接口相关的属性和相关时序：

表 9.1-10 Avalon-MM信号属性

名称	默认值	理论值	描述
associatedClock	N/A	N/A	Avalon-MM 接口同步的时钟信号
associatedReset	N/A	N/A	Avalon-MM 接口的复位信号
bridgesToMaster	0	Avalon-MM Master name on the same component	Avalon-MM 接口的桥接信号

下面我们再来看下 Avalon-MM 传输时序，

Avalon-MM 的传输定义为外设 (peripheral) 与 Avalon-MM 总线模块间的数据传输，分为主 (Master) 端传输和从 (Slave) 端传输两类，每类传输又分为基本 (fundamental) 传输、流水线 (pipelined) 传输、突发 (burst) 传输等等。所有的 Avalon-MM 传输都基于基本传输，其它传输形式都是在该传输模式下加以改进或增加某些特性以适应不同需求。一个 Master 端传输和一个对应的 Slave 端传输即可完成两个外设通过总线模块进行的一次数据传输，但 Master 端传输与 Slave 端传输的模式并不要求一致，两端传输模式可以随意搭配。同种类型的 Master 端传输与 Slave 端传输在时序上基本是一致的，其区别仅在于 Master 端传输是由 Master 端外设驱动总线模块，而 Slave 端传输是由总线模块驱动 Slave 端外设。下面我们就以 Avalon-MM 从端写基本传输时序为例进行讲解，如下图所示。

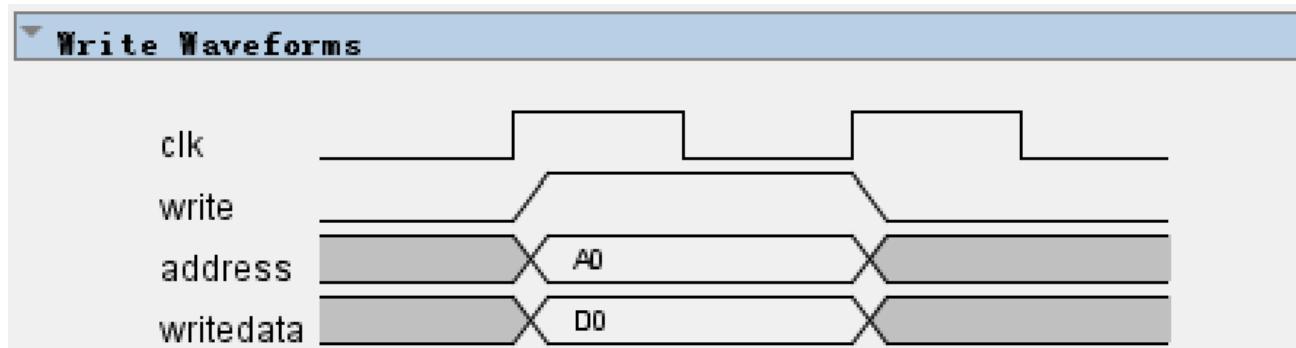


图 9.1.4 Avalon-MM 从端写基本传输时序图

从该图中我们可以看出，当时钟信号 clk 为上升沿，且写使能信号 write 为高时，地址信号 address 和写数据信号 writedata 才有效。

4. Avalon Conduit

说完了 Avalon-MM，接下来我们再来看看 AvalonConduit，Avalon Conduit 接口用于驱动片外外设信号，如驱动 SDRAM 的地址、数据和控制信号。信号类型如下表所示。

表 9.1-11 Avalon Conduit 信号类型

信号	宽度	方向	描述
<any>	<n>	In, out, or bidirectional	一个导管接口由任意宽度的一个或多个输入，输出或双向信号组成。

9.2 实验任务

本章的实验任务是自定义 IP 核实现动态数码管显示

9.3 硬件设计

了解了 Avalon 总线，现在我们就可以为我们的外设定制 IP 核了。我们以定制数码管外设的 IP 为例，介绍 IP 核的制作流程。由于我们是初次制作 IP 核，对 IP 核的制作步骤并不了解，所以，在开始制作 IP 核之前，我们先来讲一讲 IP 核的制作流程，一个典型 IP 核的制作流程主要分为以下六个步骤：

- 1) 规划 IP 核的硬件功能；
- 2) 定义一个恰当的 Avalon 接口；
- 3) 使用硬件描述语言描述硬件逻辑；
- 4) 使用 IP 核编辑器封装硬件逻辑，完成 IP 核定制；
- 5) 编写用于描述寄存器的 C 头文件和 IP 核的驱动 C 文件；
- 6) 让 Nios SBT for Eclipse 自动抓取 IP 核的 HAL

知道了 IP 核的制作流程，接下来我们可以跟着制作流程一步步往下走，定制数码管 IP 核。

(1) 规划 IP 核的硬件功能

首先我们的第一步是规划 IP 核的硬件功能，如何规划数码管 IP 核的硬件功能呢？因为 Avalon 总线是对寄存器进行操作，寄存器内的数值控制外设的状态。所以我们针对数码管的数值显示和显示控制设置数据寄存器和控制寄存器，具体如下：

表 9.3-1 寄存器描述

偏移量	寄存器名称	操作	描述
00	数据寄存器	写	显示的数值
01	控制寄存器 1	写	控制小数点的显示
02	控制寄存器 2	写	控制符号位的显示
03	控制寄存器 3	写	使能数码管显示

(2) 定义一个恰当的 Avalon 接口

规划完了数码管 IP 核的硬件功能，如果我们没有记错的话，那么便会进入第二步也就是：给我们的数码管外设定义一个恰当的 Avalon 接口。下面我们给出数码管所用的接口信号，如下表所示。

表 9.3-2 数码管所用的接口信号

信号名	Avalon 信号类型	宽度	方向
clock	Clock	1	Input
reset	Reset	1	Input
address	Avalon-MM	2	Input
write	Avalon-MM	1	Input
writedata	Avalon-MM	32	Input
sel	Conduit	6	Output
segled	Conduit	8	Output

sel 为数码管的位选端，segled 为段选端，所以选用 Conduit 接口。

(3) 使用硬件描述语言描述硬件逻辑

定义完了 Avalon 接口，接下来我们就可以进入第三步，使用硬件描述语言描述硬件逻辑。在开始描述 LEDIP 核的硬件逻辑之前，我们先来看下一个典型的 IP 核的硬件逻辑。一个典型的 IP 核的硬件逻辑由以下三个功能模块组成：

- ◆ 接口文件：作为顶层模块，定义总线接口信号；
- ◆ 寄存器文件：完成该 IP 核与外部信号进行通信，有了寄存器文件，用户就可以通过 Avalon 接口采用基地址+地址偏移量的方式来访问组件内部各寄存器。
- ◆ 硬件逻辑文件：实现 IP 核的硬件功能；

知道了这三个功能模块后，接下来我们就来编写这个三个功能模块，首先我们需要在 Quartus 软件的安装路径的 ip 文件夹下新建一个文件夹用来存放我们的 IP 核文件，这里我们创建的文件夹名字是 my_ip，然后我们在 my_ip 文件夹中创建了一个新文件夹，这里我们取名为 segled。现在我们这该文件夹下创建三个文件：分别为 Avalon 接口文件也可以称为顶层文件 segled_controller.v、硬件逻辑文件 segled_logic.v、寄存器文件 segled_register.v。创建这三个文件有很多方法，这里我们推荐使用 Quartus 来创建，我们可以创建一个以 segled_controller.v 为顶层文件的 Quartus 工程，在 Quartus 工程中我们可以编写代码并编译检查代码中的错误，代码通过编译后将.v 文件复制到 my_ip\segled 文件夹中即可。

这三个文件的 RTL 连接如下图所示：

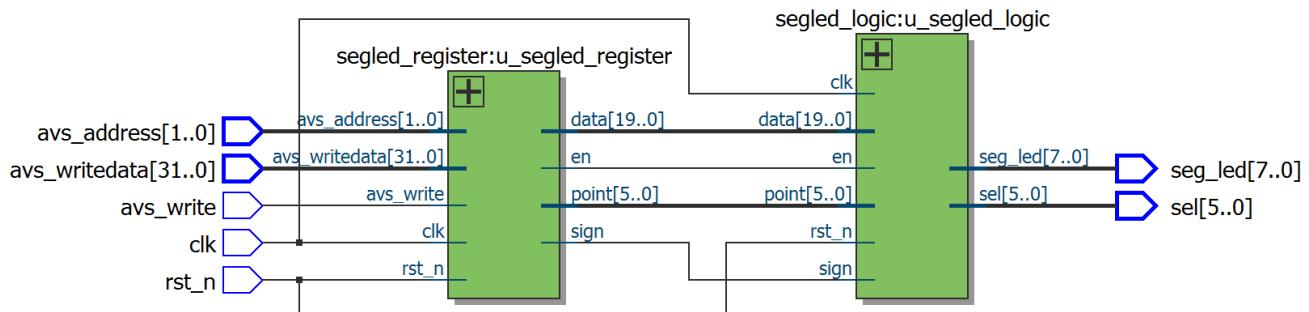


图 9.3.1 RTL连接图

首先我们给出顶层文件 `segled_controller.v`, 该文件代码如下:

```

1 module segled_controller(
2     //module clock
3     input          clk      , // 时钟信号
4     input          rst_n    , // 复位信号(低有效)
5
6     //Avalon-MM interface
7     input  [ 1:0]  avs_address , // Avalon 地址总线
8     input          avs_write   , // Avalon 写请求信
9     input  [31:0]  avs_writedata, // Avalon 写数据总线
10
11    //seg_led interface
12    output [5:0]   sel      , // 数码管位选端(选择的数码管)
13    output [7:0]   seg_led  , // 数码管段选端(数码管数值显示的段)
14 );
15
16 //wire define
17 wire  [19:0]   data ;           // 6个数码管要显示的数值
18 wire  [ 5:0]   point;          // 小数点显示的位置,从高(左)到低(右),高电平有效
19 wire          sign ;          // 显示符号位(高电平显示“-”号)
20 wire          en   ;          // 数码管使能信号
21
22 //*****
23 /**
24  main code
25 *****/

```

```
25
26 //寄存器文件
27 segled_register u_segled_register(
28     //module clock
29     .clk          (clk          ),    // 时钟信号
30     .rst_n        (rst_n        ),    // 复位信号（低有效）
31     //Avalon-MM interface
32     .avs_address  (avs_address  ),    //Avalon 地址总线
33     .avs_write    (avs_write    ),    //Avalon 写请求信
34     .avs_writedata(avs_writedata),    //Avalon 写数据总线
35     //user interface
36     .data         (data         ),    // 6 个数码管要显示的数值
37     .point        (point        ),    // 小数点显示的位置,高电平有效
38     .sign         (sign         ),    // 显示符号位（高电平显示“-”号）
39     .en           (en           ),    // 数码管使能信号
40 );
41
42 //硬件逻辑文件
43 segled_logic u_segled_logic(
44     //module clock
45     .clk          (clk          ),    // 时钟信号
46     .rst_n        (rst_n        ),    // 复位信号（低有效）
47     //user interface
48     .data         (data         ),    // 6 个数码管要显示的数值
49     .point        (point        ),    // 小数点显示的位置,高电平有效
50     .sign         (sign         ),    // 显示符号位（高电平显示“-”号）
51     .en           (en           ),    // 数码管使能信号
52     //seg_led interface
53     .sel          (sel          ),    // 数码管位选端
54     .seg_led      (seg_led      )    // 数码管段选端
55 );
56
57 endmodule
```

从该代码中我们可以看出，该文件主要是作为顶层模块，用于连接硬件逻辑文件和寄存器文件，代码中没有编写任何的逻辑功能。下面我们给出的是硬件逻辑文件 segled_logic.v，该文件代码如下：

```
1 module segled_logic(
2     //module clock
3     input          clk      ,      // 时钟信号
4     input          rst_n   ,      // 复位信号（低有效）
5
6     //seg_led interface
7     output reg [5:0]    sel     ,      // 数码管位选端（选择的数码管）
8     output reg [7:0]    seg_led,      // 数码管段选端（数码管数值显示的段）
9
10    //user interface
11    input  [19:0]    data     ,      // 6个数码管要显示的数值
12    input  [5:0]     point   ,      // 小数点显示的位置,从左到右,高电平有效
13    input          sign    ,      // 显示符号位（高电平显示“-”号）
14    input          en      ,      // 数码管使能信号
15 );
16
17 //parameter define
18 localparam MAX_NUM    = 13'd5000 ;      // 1ms 计数值
19 localparam CLK_DIVIDE = 4'd10 ;      // 时钟分频
20
21 //reg define
22 reg  [12:0]    cnt0    ;      // 1ms 计数
23 reg          flag    ;      // 1ms 计满标志信号
24 reg  [2:0]     cnt     ;      // 切换显示数码管用
25 reg  [3:0]     num1    ;      // 送给要显示的数码管, 要亮的灯
26 reg          point1  ;      // 要显示的小数点
27 reg  [23:0]    num     ;      // 24位 bcd 码用寄存器
28 reg  [3:0]     clk_cnt ;      // 时钟计数
29 reg          dri_clk ;      // 驱动数码管操作的驱动时钟
30
```

```
31 //wire define
32 wire [3:0] data0 ; // 十万位数
33 wire [3:0] data1 ; // 万位数
34 wire [3:0] data2 ; // 千位数
35 wire [3:0] data3 ; // 百位数
36 wire [3:0] data4 ; // 十位数
37 wire [3:0] data5 ; // 个位数
38
39 //*****
40 /***          main code
41 //*****
42
43 assign data5 = data[19:0] / 17'd100000; // 十万位数
44 assign data4 = data[19:0] / 14'd10000 % 4'd10; // 万位数
45 assign data3 = data[19:0] / 10'd1000 % 4'd10 ; // 千位数
46 assign data2 = data[19:0] / 7'd100 % 4'd10 ; // 百位数
47 assign data1 = data[19:0] / 4'd10 % 4'd10 ; // 十位数
48 assign data0 = data[19:0] % 4'd10; // 个位数
49
50 //生成数码管的驱动时钟用于驱动数码管的操作
51 always @(posedge clk or negedge rst_n) begin
52     if(!rst_n) begin
53         dri_clk <= 1'b1;
54         clk_cnt <= 4'd0;
55     end
56     else if(clk_cnt == CLK_DIVIDE/2 - 1'd1) begin
57         clk_cnt <= 4'd0;
58         dri_clk <= ~dri_clk;
59     end
60     else
61         clk_cnt <= clk_cnt + 1'b1;
62 end
63
```

```
64 //将 20 位 2 进制数转换为 8421bcd 码
65 always @ (posedge dri_clk or negedge rst_n) begin
66     if (!rst_n)
67         num <= 24'b0;
68     else begin
69         if (data5 || point[5]) begin
70             num[23:20] <= data5;
71             num[19:16] <= data4;
72             num[15:12] <= data3;
73             num[11:8]  <= data2;
74             num[ 7:4]  <= data1;
75             num[ 3:0]  <= data0;
76         end
77         else begin
78             if (data4 || point[4]) begin
79                 num[19:0]  <= {data4, data3, data2, data1, data0};
80                 if(sign)
81                     num[23:20] <= 4'd11;
82                 else
83                     num[23:20] <= 4'd10;
84             end
85             else begin
86                 if (data3 || point[3]) begin
87                     num[15: 0] <= {data3, data2, data1, data0};
88                     num[23:20] <= 4'd10;
89                     if(sign)
90                         num[19:16] <= 4'd11;
91                     else
92                         num[19:16] <= 4'd10;
93                 end
94                 else begin
95                     if (data2 || point[2]) begin
96                         num[11: 0] <= {data2, data1, data0};
```

```
97          num[23:16] <= {2{4' d10}};  
98          if(sign)  
99              num[15:12] <= 4' d11;  
100         else  
101             num[15:12] <= 4' d10;  
102         end  
103     else begin  
104         if (data1 || point[1]) begin  
105             num[ 7: 0] <= {data1,data0};  
106             num[23:12] <= {3{4' d10}};  
107             if(sign)  
108                 num[11:8] <= 4' d11;  
109             else  
110                 num[11:8] <= 4' d10;  
111         end  
112     else begin  
113         num[3:0] <= data0;  
114         if(sign)  
115             num[23:4] <= {{4{4' d10}},4' d11};  
116         else  
117             num[23:4] <= {5{4' d10}};  
118         end  
119     end  
120   end  
121 end  
122 end  
123 end  
124 end  
125  
126 //计数 1ms  
127 always @ (posedge dri_clk or negedge rst_n) begin  
128     if (rst_n == 1' b0) begin  
129         flag <= 1' b0;
```

```
130      cnt0 <= 13' b0;
131    end
132  else if (cnt0 < MAX_NUM - 1' b1) begin
133    flag <= 1' b0;
134    cnt0 <= cnt0 + 1' b1;
135  end
136 else begin
137    flag <= 1' b1;
138    cnt0 <= 13' b0;
139  end
140 end
141
142 //计数器，用来计数 6 个状态（因为有 6 个灯）
143 always @ (posedge dri_clk or negedge rst_n) begin
144   if (rst_n == 1' b0)
145     cnt <= 3' b0;
146   else if(flag) begin
147     if(cnt < 3' d5)
148       cnt <= cnt + 1' b1;
149   else
150     cnt <= 3' b0;
151 end
152 end
153
154 //6 个数码管轮流显示，完成刷新（从右到左）
155 always @ (posedge dri_clk or negedge rst_n) begin
156   if(!rst_n) begin
157     sel <= 6' b000000;
158     num1 <= 4' b0;
159   end
160   else begin
161     if(en) begin
162       case (cnt)
```

```
163      3' d0: begin
164          sel    <= 6' b111110;
165          num1   <= num[3:0] ;
166          point1 <= ~point[0] ;
167      end
168      3' d1: begin
169          sel    <= 6' b1111101;
170          num1   <= num[7:4] ;
171          point1 <= ~point[1] ;
172      end
173      3' d2: begin
174          sel    <= 6' b111011;
175          num1   <= num[11:8];
176          point1 <= ~point[2] ;
177      end
178      3' d3: begin
179          sel    <= 6' b110111;
180          num1   <= num[15:12];
181          point1 <= ~point[3] ;
182      end
183      3' d4: begin
184          sel    <= 6' b101111;
185          num1   <= num[19:16];
186          point1 <= ~point[4];
187      end
188      3' d5: begin
189          sel    <= 6' b011111;
190          num1   <= num[23:20];
191          point1 <= ~point[5];
192      end
193      default: begin
194          sel    <= 6' b000000;
195          num1   <= 4' b0;
```

```
196             point1 <= 1'b1;
197         end
198     endcase
199 end
200 else
201     sel <= 6'b111111;
202 end
203 end
204
205 //数码管显示数据
206 always @ (posedge dri_clk or negedge rst_n) begin
207     if (!rst_n)
208         seg_led <= 7'h40;
209     else begin
210         case (num1)
211             4'd0 : seg_led <= {point1, 7'b1000000};
212             4'd1 : seg_led <= {point1, 7'b1111001};
213             4'd2 : seg_led <= {point1, 7'b0100100};
214             4'd3 : seg_led <= {point1, 7'b0110000};
215             4'd4 : seg_led <= {point1, 7'b0011001};
216             4'd5 : seg_led <= {point1, 7'b0010010};
217             4'd6 : seg_led <= {point1, 7'b0000010};
218             4'd7 : seg_led <= {point1, 7'b1111000};
219             4'd8 : seg_led <= {point1, 7'b0000000};
220             4'd9 : seg_led <= {point1, 7'b0010000};
221             4'd10: seg_led <= 8'b11111111;
222             4'd11: seg_led <= 8'b10111111;
223         default : seg_led <= {point1, 7'b1000000};
224     endcase
225 end
226 end
227
228 endmodule
```

该代码与我们的数码管动态显示实验中的数码管驱动代码相同，只是更改了文件名和模块名，对于此代码有不理解的地方可参考动态数码管显示实验。下面我们给出的是寄存器文件segled_register.v，该文件代码如下：

```
1 module segled_register(
2     input          clk      , // 时钟信号
3     input          rst_n   , // 复位信号（低有效）
4
5     //Avalon-MM interface
6     input [ 1:0]    avs_address , //Avalon 地址总线
7     input          avs_write   , //Avalon 写请求信
8     input [31:0]    avs_writedata, //Avalon 写数据总线
9
10    //user interface
11    output reg [19:0] data      , // 6 个数码管要显示的数值
12    output reg [ 5:0] point     , // 小数点显示的位置,从左到右,高电平有效
13    output reg        sign      , // 显示符号位(高电平显示“-”号)
14    output reg        en       // 数码管使能信号
15 );
16
17 //*****
18 //**          main code
19 //*****
20
21 //用于给数码管数据寄存器进行赋值
22 always @(posedge clk or negedge rst_n) begin
23     if(!rst_n)
24         data <= 20'd0;
25     else if((avs_write) && (avs_address == 2'b00))
26         data <= avs_writedata[19:0];           // 数码管显示的数据
27 end
28
29 //用于给数码管控制小数点显示寄存器进行赋值
30 always @(posedge clk or negedge rst_n) begin
```

```
31      if(!rst_n)
32          point <= 6'd0;
33      else if((avs_write) && (avs_address == 2'b01))
34          point <= avs_writedata[5:0];           // 小数点显示的位置, 从左到右, 高电平有效
35 end
36
37 //用于给数码管控制符号寄存器进行赋值
38 always @(posedge clk or negedge rst_n) begin
39     if(!rst_n)
40         sign <= 1'b0;
41     else if((avs_write) && (avs_address == 2'b10))
42         sign <= avs_writedata[0];           // 显示符号位(高电平显示“-”号) 3
43 end
44
45 //用于给数码管使能寄存器进行赋值
46 always @(posedge clk or negedge rst_n) begin
47     if(!rst_n)
48         en <= 1'b0;
49     else if((avs_write) && (avs_address == 2'b11))
50         en <= avs_writedata[0];           // 数码管使能信号
51 end
52
53 endmodule
```

代码第 6~8 行定义了 Avalon 的地址总线、写请求线和数据总线的接口。第 11~14 行为 4 个寄存器的接口。从第 22 行开始的语句通过写信号 ava_write 和地址信号 ava_address 给相应地址的寄存器写入数据线 avs_writedata 的数据。

(4) 使用 IP 核编辑器封装硬件逻辑，完成 IP 核定制

当我们创建好了描述数码管 IP 核的三个模块后，我们就可以使用 Qsys 中的组件编辑器将它们封装成一个 IP 核。我们打开 Qsys 软件，选择 Qsys 软件菜单栏中的【File】→【New Component…】将会出现如下图所示页面。

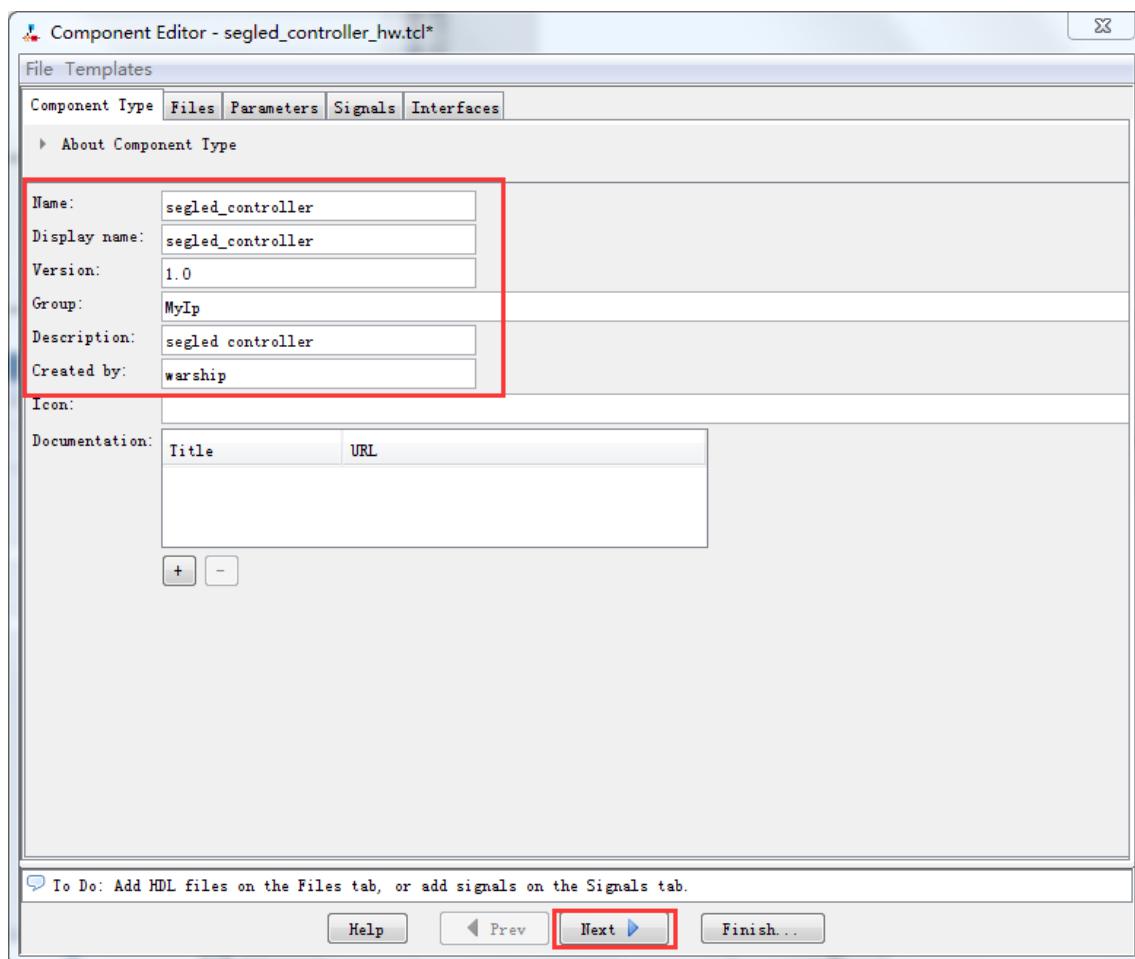


图 9.3.2 Component Editor 页面图

在组件编辑器页面中我们给创建的IP核填写基本信息，填写完毕后，我们便可以点击【Next】进入下一个页面，如下图所示。

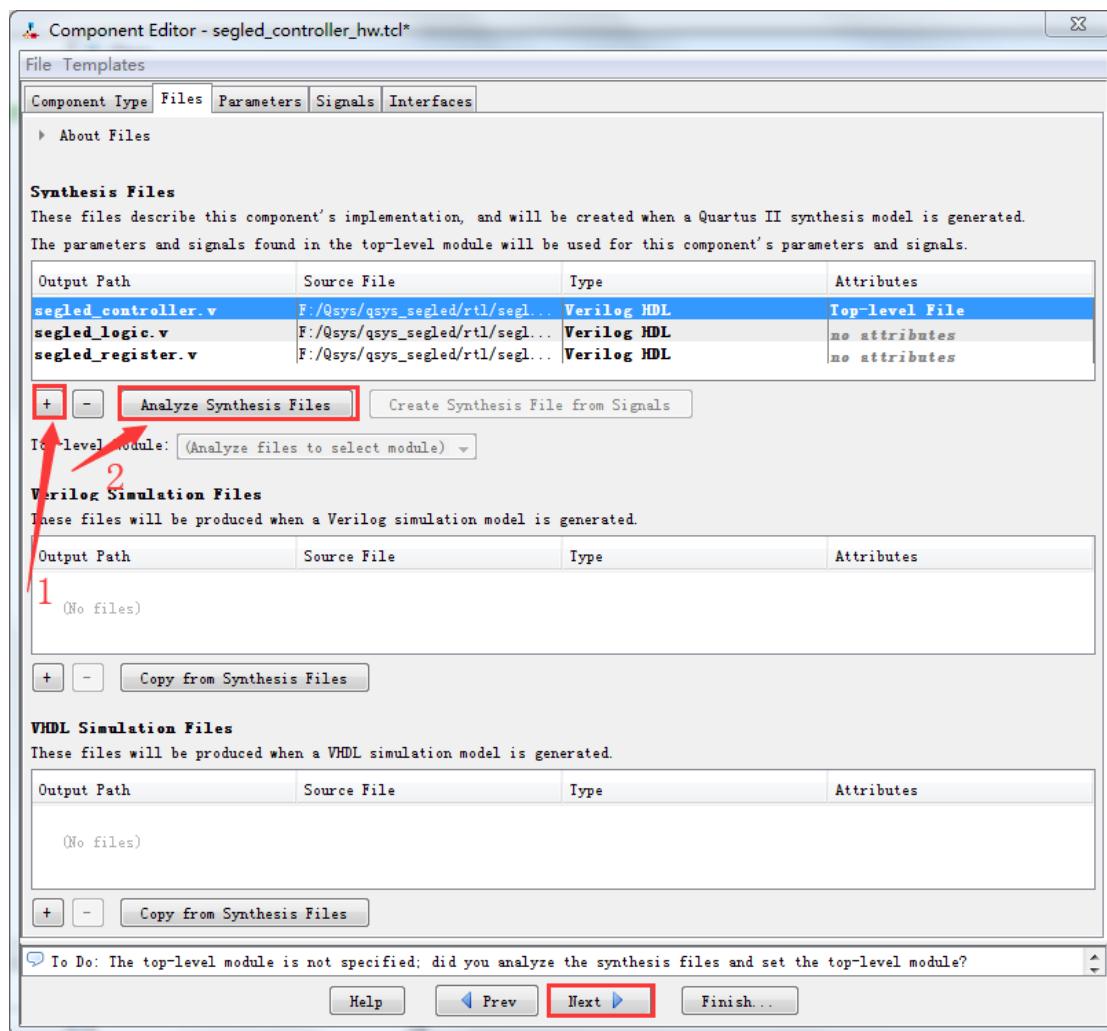


图 9.3.3 添加完硬件文件后的窗口

在该图中，我们可以点击【+】按钮打开添加文件对话框，将路径指向 segled 文件夹所在的目录，最后选中三个.v 文件添加即可，要注意的是默认第一栏为顶层文件，如果不是，双击 Attributes 栏下的 no attributes 修改为顶层文件即可。添加完硬件文件后，Synthesis Files 栏中可看到刚添加的 3 个文件。下面我们点击【Analyze Synthesis Files】按钮来分析这三个文件，分析完毕后将会出现如下图所示页面。

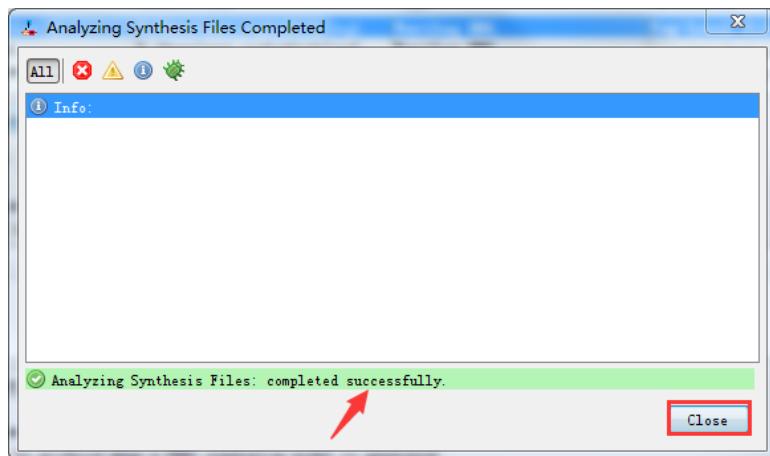


图 9.3.4 硬件描述文件分析完毕结果图

从该图中我们可以看到，我们的代码没有错误，如果代码中存在错误，这里分析就不会通过。我们点击【Close】即可，这时我们会发现最下面的提示窗口中出现了错误，如下图所示：

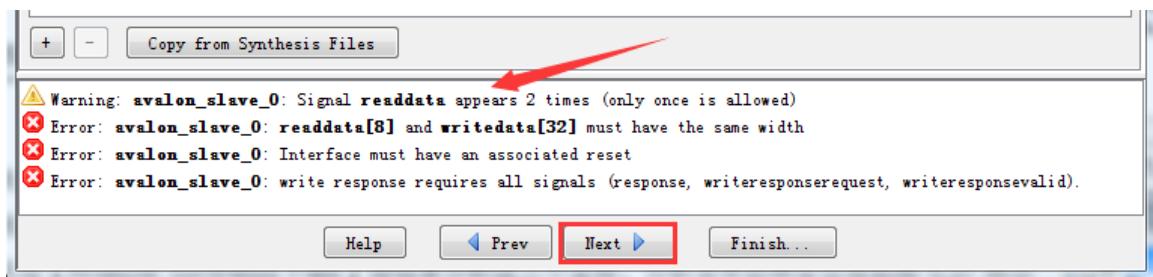


图 9.3.5 错误提示

这个错误我们会在后面的步骤中将它解决掉，这里先不用管。接下来我们继续点击【Next】进入下一个页面，如下图所示。

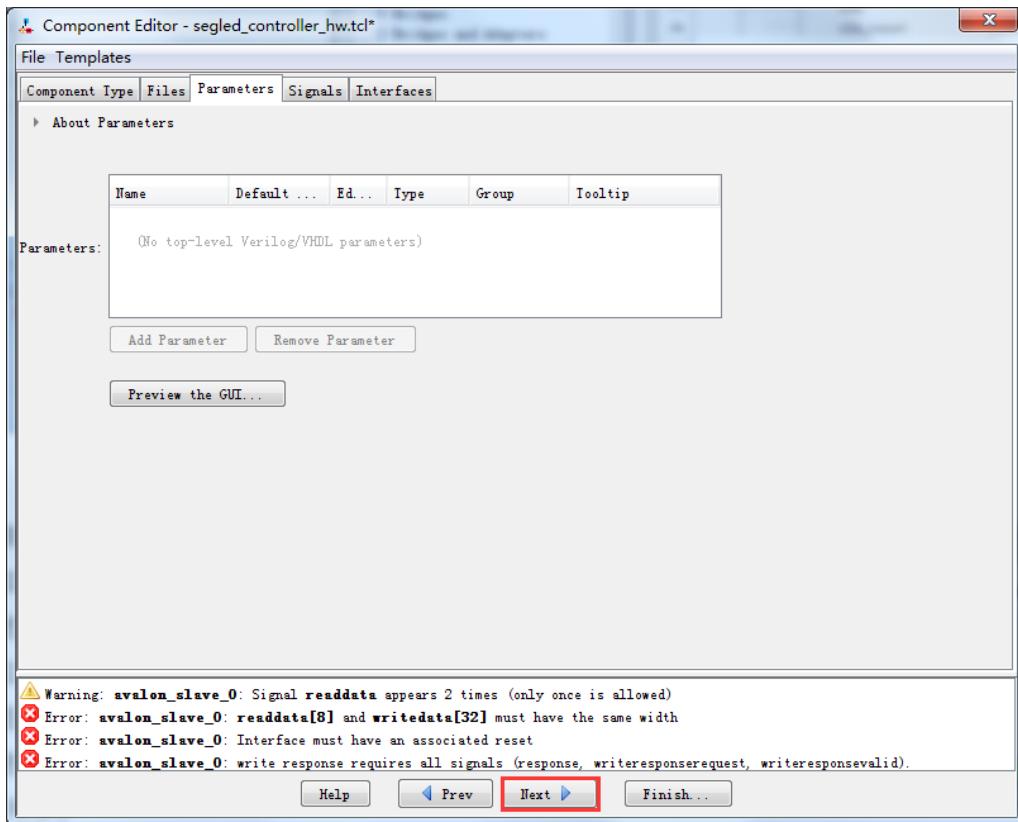


图 9.3.6 参数页面图

参数标签能够用于在 Qsys 系统中指定配置组件的实例的参数。如我们新起点开发板上使用的是 6 位数码管，所以该 IP 核用于 6 位数码管的驱动，如果我们想把该自定义 IP 核用于任意位数码管，可以定义一个参数，当用于不同位数码管，修改该参数即可，如同 SDRAM 控制器的参数设置，这样就不用再重新添加 IP 核组件。接下来我们点击【Next】进入下一个页面，如下图所示。

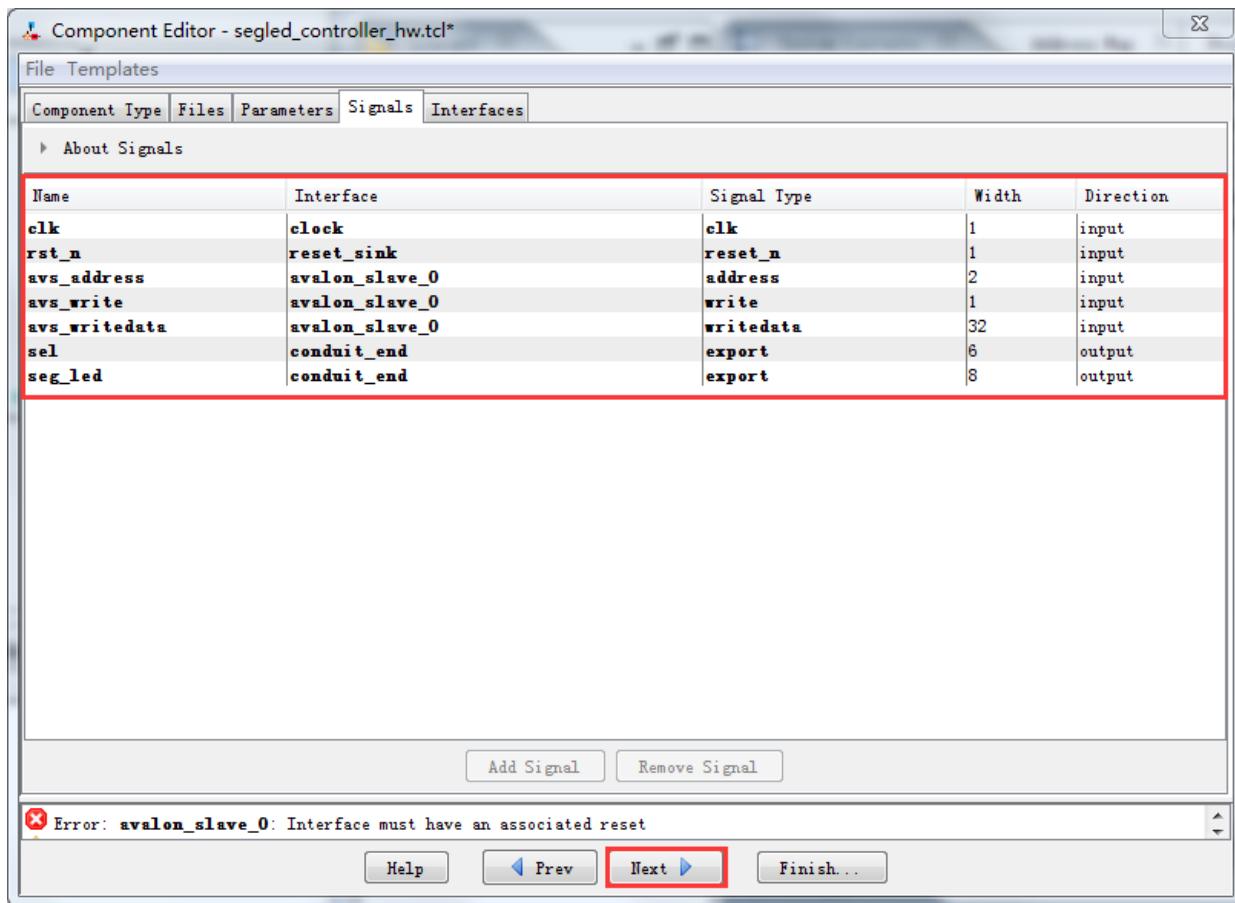


图 9.3.7 设置信号页面图

我们需要修改信号的总线接口 Interface 和信号类型 Signal Type，如上图所示。我们发现底部的错误减少了。接下来我们点击【Next】进入下一个页面，如下图所示。

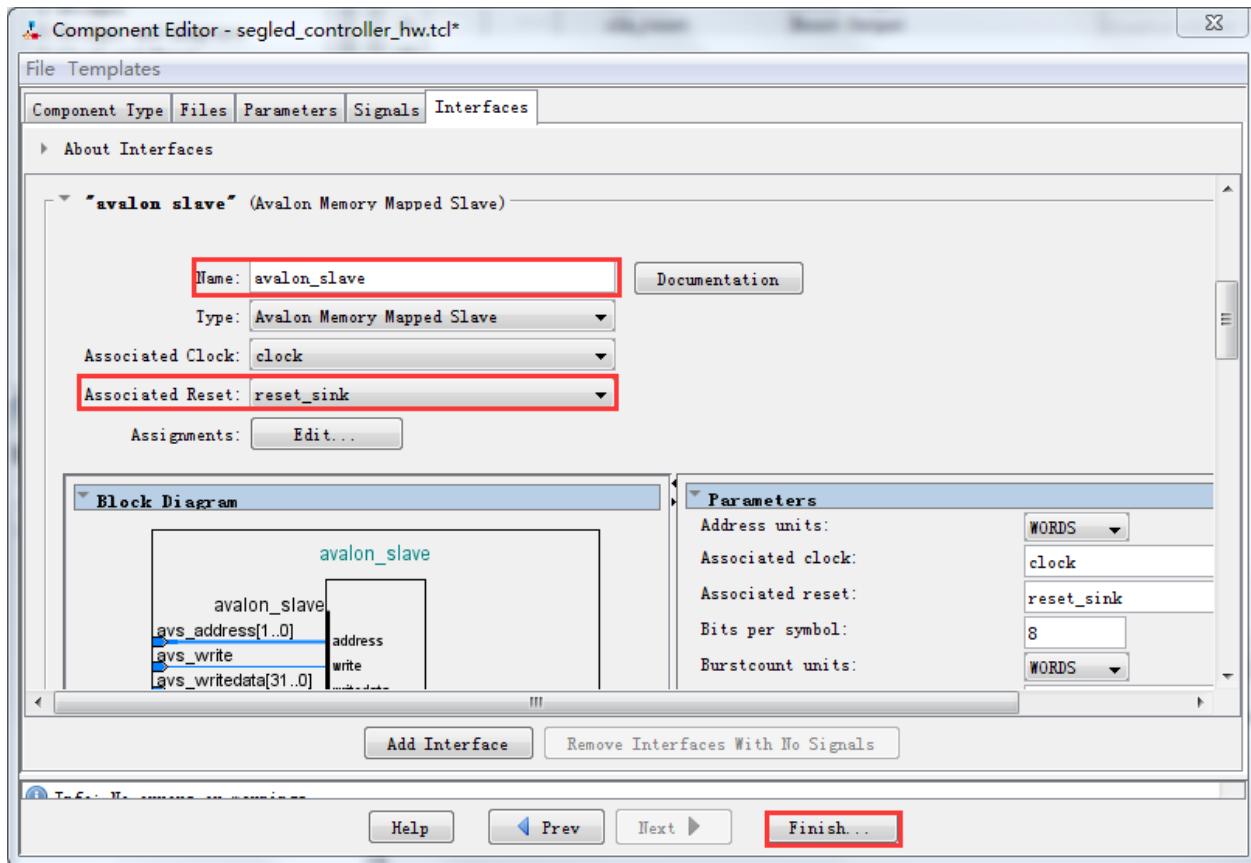


图 9.3.8 设置接口页面图

从该图中我们在 avalon slave 一栏中修改了 Name 和 Associated 两个选项，这时错误消失便消失了。下面我们便可以点击【Finish】按钮，出现如下图所示页面。

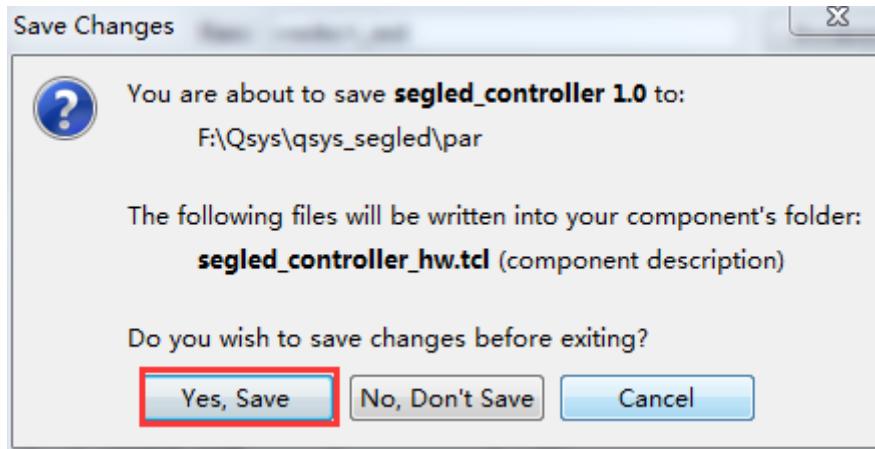


图 9.3.9 保存

这时我们可以看到 Qsys 软件将我们创建的 segled_controller_hw.tcl 文件保存到了 F:\Qsys\qsys_segled\par 路径下面。之所以保存在这里，是因为我们在创建 IP 核之前已经创建了

Quartus 工程和 Qsys 系统，如果没有创建 Quartus 工程直接在 Qsys 软件中创建 IP 核，默认路径将会是 Quartus 的安装路径，这里需要注意的是，如果默认路径是 Quartus 的安装路径，那么创建出来的 segled_controller_hw.tcl 文件我们还需要修改其中的路径，否则我们在使用时会出现错误。这里读者可以自行尝试。我们点击【Yes, Save】按钮就完成 IP 核的定制，这里需要我们注意的是，为了方便 IP 的管理，我们需要将 segled_controller_hw.tcl 文件复制到我们先前创建的放置三个.v 文件的路径中，即 D:\quartus13\ip\my_ip\segled 中。

(5) 编写用于描述寄存器的 C 头文件和 IP 核的驱动 C 文件

完成 IP 核定制，我们就可以在 Qsys 软件的 IP 核库中看到我们添加的 IP 核了，也就是说，我们已经可以使用该 IP 核了，不过，由于我们没有给该 IP 核添加寄存器头文件和底层驱动文件，我们使用起来是极其不方便的，为了以后的使用方便，下面我们就需要给该 IP 核添加寄存器头文件和底层驱动文件，首先我们需要做的是在\my_ip\segled 文件夹中创建两个新文件夹，一个为 HAL 文件夹，另一个为 inc 文件夹。如图 9.3.10 所示。

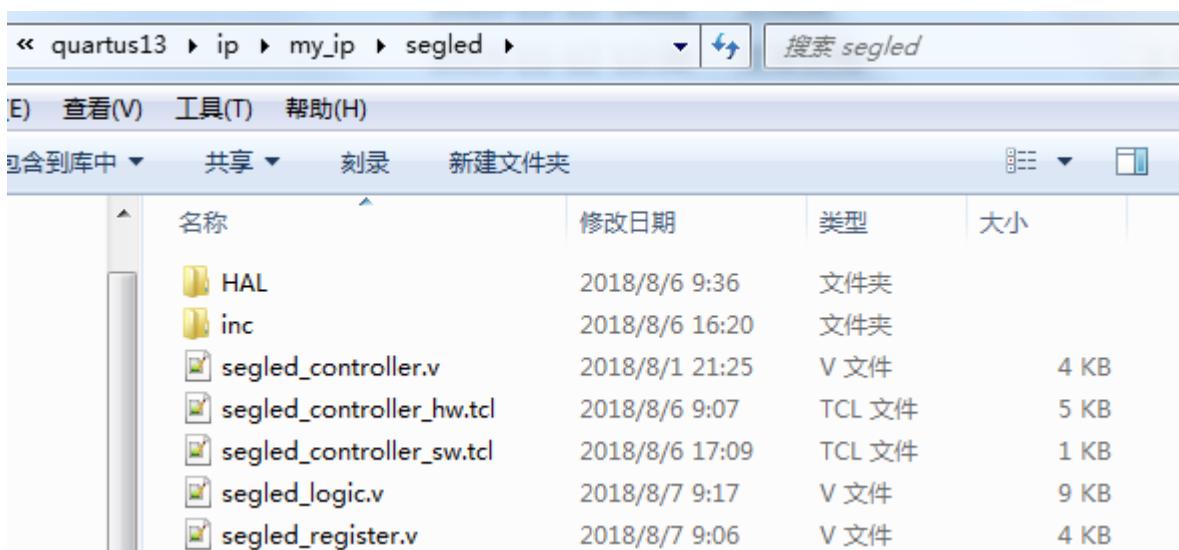


图 9.3.10 segled IP核文件夹

我们需要在\my_ip\segled\inc 文件夹下创建一个 segled_controller_regs.h 文件，还需要在\my_ip\segled\HAL\inc 文件夹下创建一个 segled_controller.h 文件。创建好了文件以后，接下来我们要分别对这两个 C 文件进行编写代码，在编写代码时我们可以参考 altera 为我们已经提供好的 IP 核，比如，可以参考 altera_avalon_pio_regs.h 文件来写我们的 segled_controller_regs.h 文件，这里我们直接给出已经编写好的代码，首先我们给出的是 segled_controller_regs.h 文件的代码，如下所示：

```
1 #ifndef __SEGLED_CONTROLLER_REGS_H__
```

```
2 #define __SEGLED_CONTROLLER_REGS_H__  
3  
4 #include <io.h>  
5  
6 //Data register  
7 #define IOWR_AVALON_SEGLED_DATA(base, data) IOWR(base, 0, data)  
8 //Control register1  
9 #define IOWR_AVALON_SEGLED_DOT(base, data) IOWR(base, 1, data)  
10 //Control register2  
11 #define IOWR_AVALON_SEGLED_SIGN(base, data) IOWR(base, 2, data)  
12 //Control register3  
13 #define IOWR_AVALON_SEGLED_EN(base, data) IOWR(base, 3, data)  
14  
15 #endif
```

从该代码中，我们定义了四个宏定义，分别用于写数据寄存器、控制小数点显示的控制寄存器 1、控制符号位显示的控制寄存器 2 和控制是否显示的控制寄存器 3。接下来我们给出的是 segled_controller.h 文件的代码，如下所示：

```
1 #ifndef __SEGLED_CONTROLLER_H__  
2 #define __SEGLED_CONTROLLER_H__  
3  
4 #include "system.h"  
5 #include "alt_types.h"  
6 #include "segled_controller_regs.h"  
7  
8 #ifdef __cplusplus  
9 extern "C"  
10 {  
11     #endif /* __cplusplus */  
12  
13  
14     /* function station */  
15 }
```

```
16 #ifdef __cplusplus  
17 }  
18 #endif /* __cplusplus */  
19  
20#endif /* __SEGLED_CONTROLLER_H__ */
```

我们可以在第 12~15 行声明函数，由于我们这里没有额外定义函数，就没有声明。

(6) 让 Nios SBT for Eclipse 自动抓取 IP 核的 HAL

编写完了寄存器头文件和驱动底层头文件，也就到了 LED IP 核定制的最后一步，创建 _sw.tcl 文件，_sw.tcl 文件同_hw.tcl 文件一样都是利用 Tcl 语言编写的，由于_sw.tcl 内容比较少，并且 Tcl 语言也是比较容易看懂的，所以我们完全没有必要再专门去学习 Tcl 语言，当然，如果有对 Tcl 语言感兴趣的朋友可以另外查找关于 Tcl 语言资料进行学习。这里我们就不再进一步介绍了。

下面我们直接给出已经编写好的 segled_controller_sw.tcl 代码，如代码 5.7 所示。

```
1 #  
2 # segled_controller.tcl  
3 #  
4  
5 # Create a new driver  
6 create_driver segled_controller  
7  
8 # Associate it with some hardware known as "segled_controller"  
9 set_sw_property hw_class_name segled_controller  
10  
11 # The version of this driver  
12 set_sw_property version 13.1  
13  
14 # This driver may be incompatible with versions of hardware less  
15 # than specified below. Updates to hardware and device drivers  
16 # rendering the driver incompatible with older versions of  
17 # hardware are noted with this property assignment.  
18 set_sw_property min_compatible_hw_version 1.0
```

```
19
20 # Initialize the driver in alt_sys_init()
21 set_sw_property auto_initialize false
22
23
24 #
25 # Source file listings...
26 #
27
28
29 # Include files
30 add_sw_property include_source HAL/inc/segled_controller.h
31 add_sw_property include_source inc/segled_controller_regs.h
32
33 # This driver supports HAL & UCOSII BSP (OS) types
34 add_sw_property supported_bsp_type HAL
35 add_sw_property supported_bsp_type UCOSII
36 add_sw_property supported_bsp_type BML
37
38 # End of file
```

这里我们需要说明的是，我们参考的是 altera_avalon_pio_sw.tcl 文件中的代码来编写的，在该代码中，比较重要的也就两处，第一处是第 21 行代码，该代码是取消用来将我们的数码管 IP 核添加至 alt_sys_init() 函数中进行自动初始化。第二处是第 30 和 31 行，该代码是用来关联我们编写的寄存器头文件和底层驱动文件。

下面我们来看一下 Qsys 的硬件框架，如下图所示

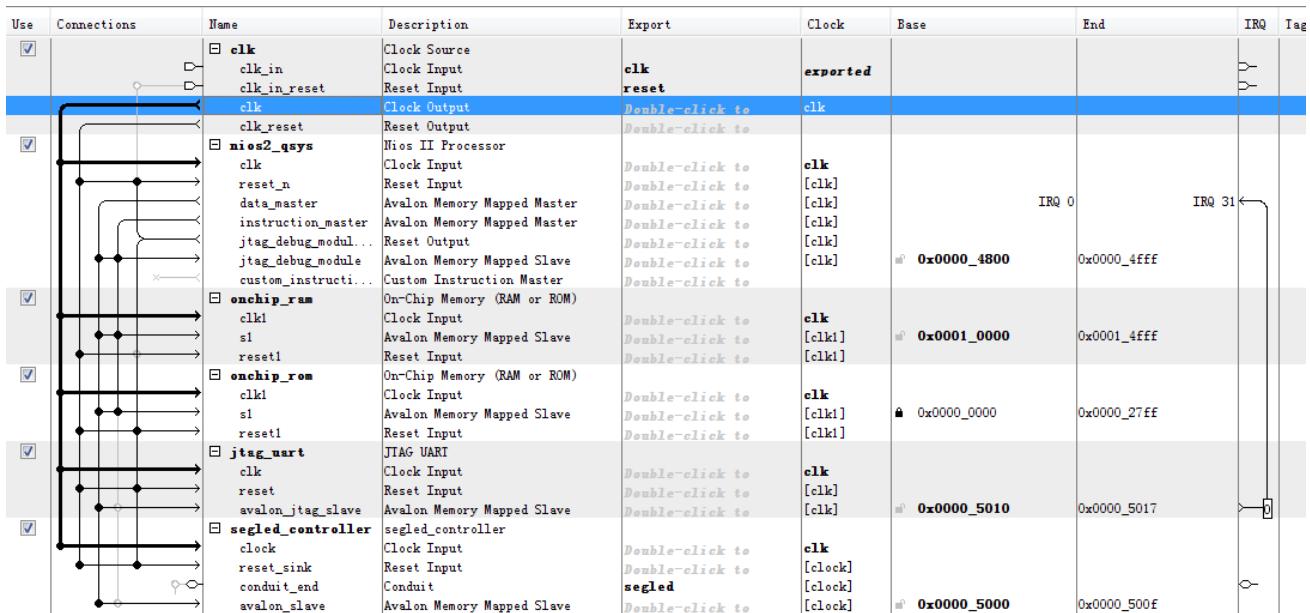


图 9.3.11 数码管 IP 核的硬件框架图

图中，clk IP 核的时钟频率设置为 50MHz。onchip_ram 存储空间大小为 20480Bytes，onchip_rom 为 10240Bytes。添加的数码管 IP 核我们不需要任何配置。

需要注意的是，此时 Nios II IP 核的复位向量 Reset Vector 和异常向量 Exception Vector 的设置如下图所示：

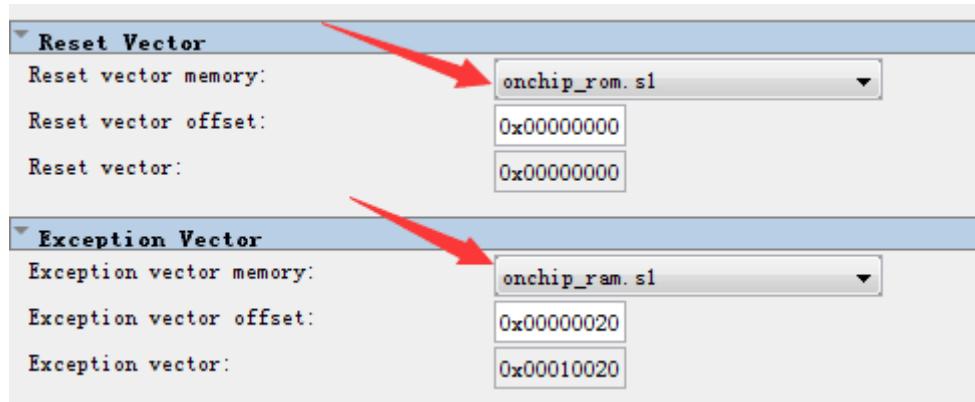


图 9.3.12 Nios II 的复位向量和异常向量

对硬件工程进行例化的顶层代码，如下所示：

```

1 module top_segled(
2     //module clock
3     input          sys_clk,           // 时钟信号

```

```

4      input          sys_rst_n,           // 复位信号（低有效）
5
6      //segled interface
7      output wire [5:0] sel,             // 数码管位选端
8      output wire [7:0] seg_led         // 数码管段选端
9  );
10
11 //*****
12 /**
13 main code
14 ****
15 //例化数码管模块
16 segled u_segled (
17     .clk_clk      (sys_clk ),        // 时钟信号
18     .reset_reset_n (sys_rst_n),       // 复位信号（低有效）
19     .segled_sel    (sel      ),       // 数码管位选端
20     .segled_seg_led (seg_led )       // 数码管段选端
21 );
22
23 endmodule

```

其中管脚分配如下：

表 9.3-3 自定义IP核实验管脚分配

信号名	方向	管脚	端口说明
sys_clk	input	E1	系统时钟, 50M
sys_rst_n	input	M1	系统复位, 低有效
sel[0]	output	N16	数码管位选0
sel[1]	output	N15	数码管位选1
sel[2]	output	P16	数码管位选2
sel[3]	output	P15	数码管位选3
sel[4]	output	R16	数码管位选4
sel[5]	output	T15	数码管位选5

seg_led[0]	output	M11	数码管段选a
seg_led[1]	output	N12	数码管段选b
seg_led[2]	output	C9	数码管段选c
seg_led[3]	output	N13	数码管段选d
seg_led[4]	output	M10	数码管段选e
seg_led[5]	output	N11	数码管段选f
seg_led[6]	output	P11	数码管段选g
seg_led[7]	output	D9	数码管段选h

9.4 软件设计

讲完了硬件框架，接下来我们来看一下本实验的软件工程代码，如下：

```

1 #include <stdio.h>           // 标准输入输出头文件
2 #include <unistd.h>          // 延迟函数头文件
3 #include "alt_types.h"        // 数据类型头文件
4 #include "segled_controller.h" // 数码管驱动文件
5
6 //-
7 //-- 名称 : main()
8 //-- 功能 : 程序入口
9 //-- 输入参数 : 无
10 //-- 输出参数 : 无
11 //-
12
13 int main() {
14
15     alt_u32 counter;
16
17     IOWR_AVALON_SEGLED_DOT(SEGLED_CONTROLLER_BASE, 04);      // 8 进制, 点亮右边第 3 个小数点
18     IOWR_AVALON_SEGLED_EN(SEGLED_CONTROLLER_BASE, 1);
19     IOWR_AVALON_SEGLED_SIGN(SEGLED_CONTROLLER_BASE, 1);       // 显示符号位
20     for(counter=200;counter>0;counter--) {                   // 从 200 开始递减

```

```
21     IOWR_AVALON_SEGLED_DATA(SEGLED_CONTROLLER_BASE, counter);  
22     usleep(10000);  
23 }  
24  
25 IOWR_AVALON_SEGLED_SIGN(SEGLED_CONTROLLER_BASE, 0); // 关闭显示符号位  
26 for(counter=0;counter<=99999;counter++) { // 从 f 开始递增  
27     IOWR_AVALON_SEGLED_DATA(SEGLED_CONTROLLER_BASE, counter);  
28     usleep(10000);  
29     if(counter==99999)  
30         counter=0;  
31 }  
32 return 0;  
33 }
```

SEGLED_CONTROLLER_BASE 是数码管 IP 的寄存器起始地址，可在 system.h 中找到。代码中第一个 for 的功能是从-200 递减到 0，第二个 for 的功能是从 0 递增到 99999。

9.5 下载验证

讲完了软件工程，接下来我们就将该实验下载至我们的新起点开发板进行验证，首先我们需要在 Quartus II 软件中将 qsys_segled.sof 文件下载至我们的新起点开发板，qsys_segled.sof 下载完成后，我们还需要在 Eclipse 软件中将 segled.elf 文件下载至我们的新起点开发板，下载完成以后，我们的 C 程序将会在我们的新起点开发板上执行，在开发板上的显示结果如下：

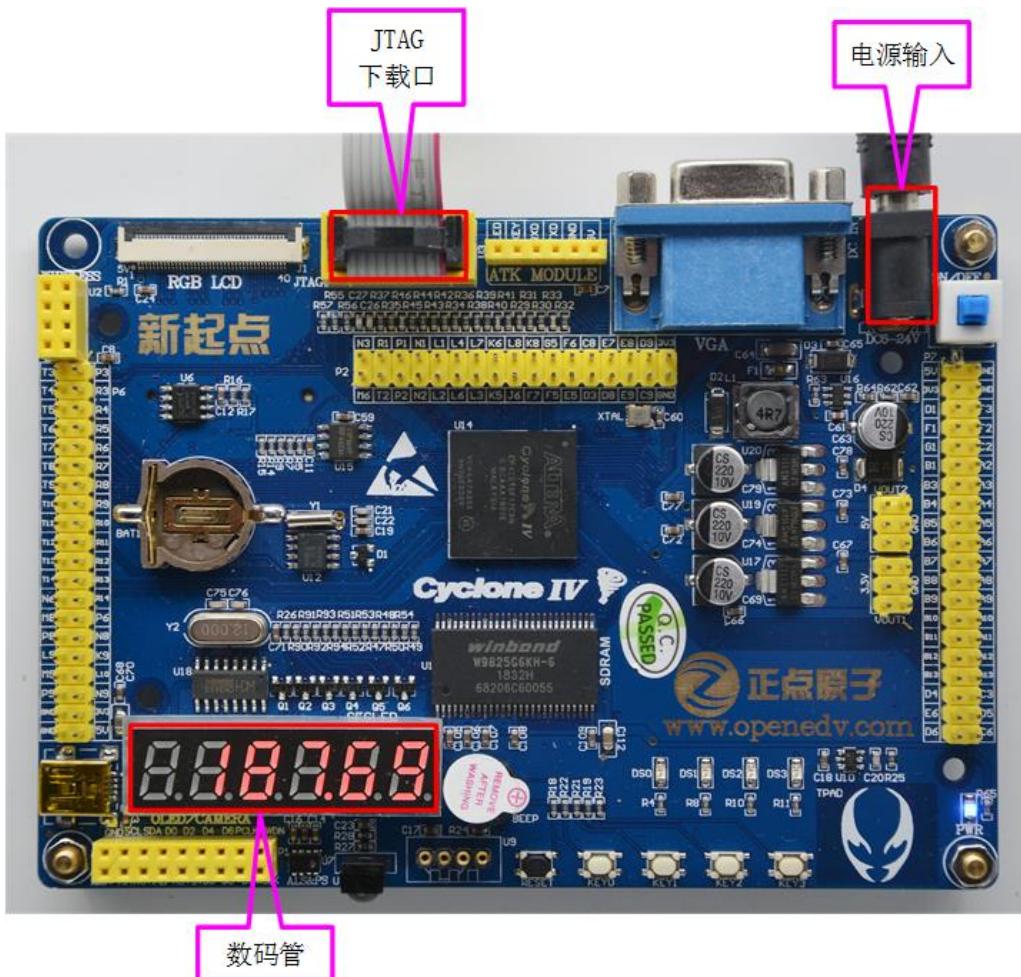


图 9.5.1 实验结果

实验结果与设计的相符，自定义数码管 IP 核实验完成。

第十章 Nios II 上电启动过程分析

程序固化到EPCS后的存储映射是什么样的？Nios II在上电过程中经历了什么？带着这些疑问我们进入本章的学习，了解Nios II不为人知的秘密。本章包括以下几个部分：

10.1 简介

10.2 Nios II在EPCS中的存储映射

10.3 Nios II上电启动过程分析