

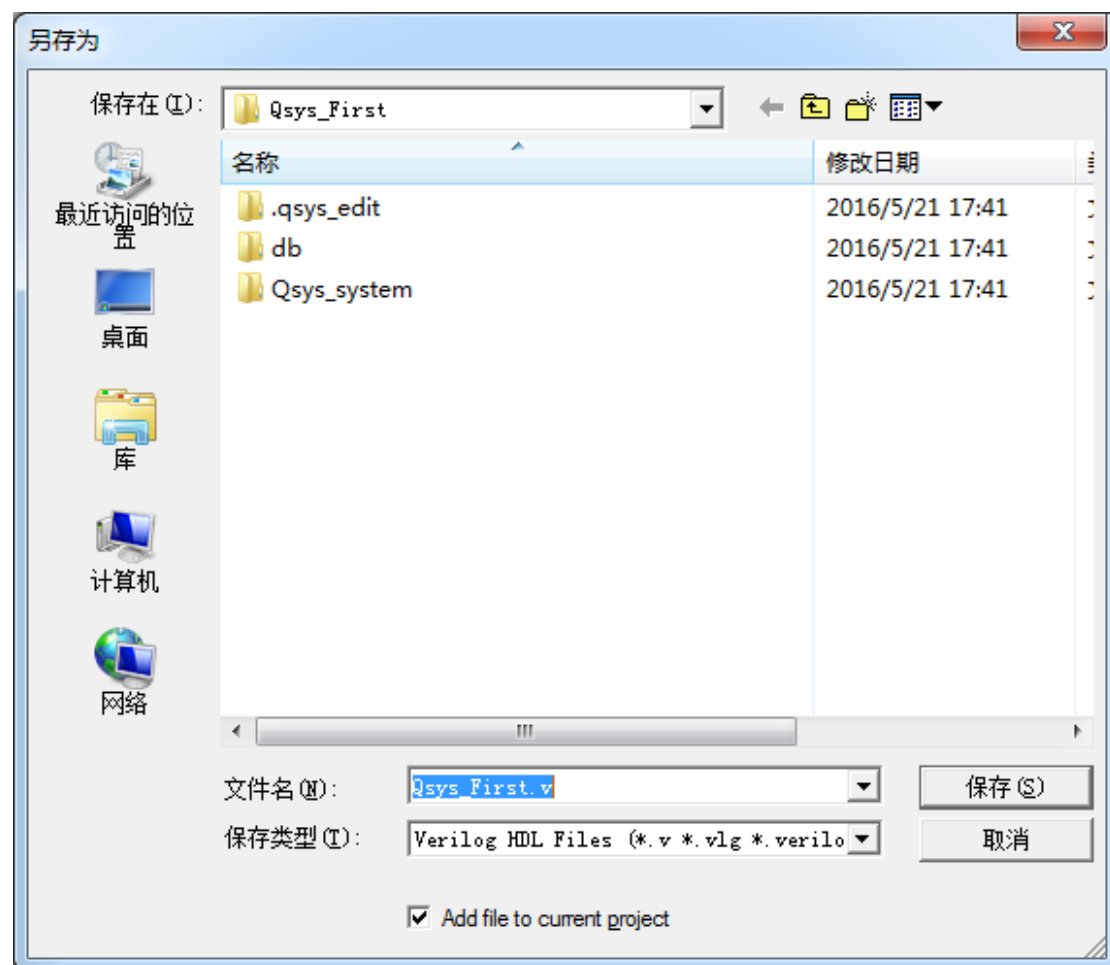


图 2.33 保存顶层文件路径

在上述代码中，我们主要介绍两点：第一点是 PLL IP 核是如何使用的，这里我们并不给出详细讲解，只给出创建过程，想要深入了解 PLL IP 核的读者，可以参考《软件工具篇中》的介绍来进一步学习。第二点是 Qsys_system 代码是如何编写的。首先我们介绍的是 PLL IP 核的创建。在 Quartus II 软件中，我们单击菜单栏中的【Tools】→【MegaWizard Plug-In Manager】出现图 2.34 所示页面。

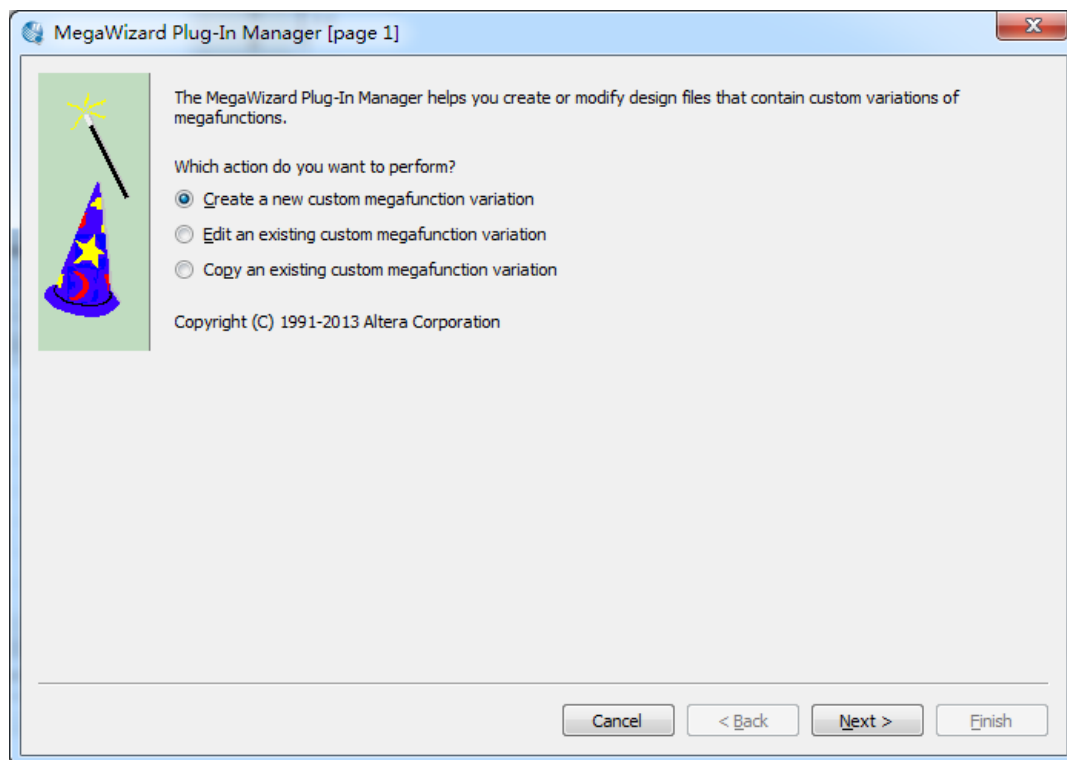


图 2.34 创建 IP 核向导页面

我们这里直接点击【Next>】，我们可以通过搜索栏进行搜索 ALTPLL IP 核，也可以直接找到 I/O 文件夹下的 ALTPLL IP 核，这里我们需要给 PLL IP 核命名，这里同样也可以任意命名，我们这里就命名为 PLL。如图 2.35 所示。

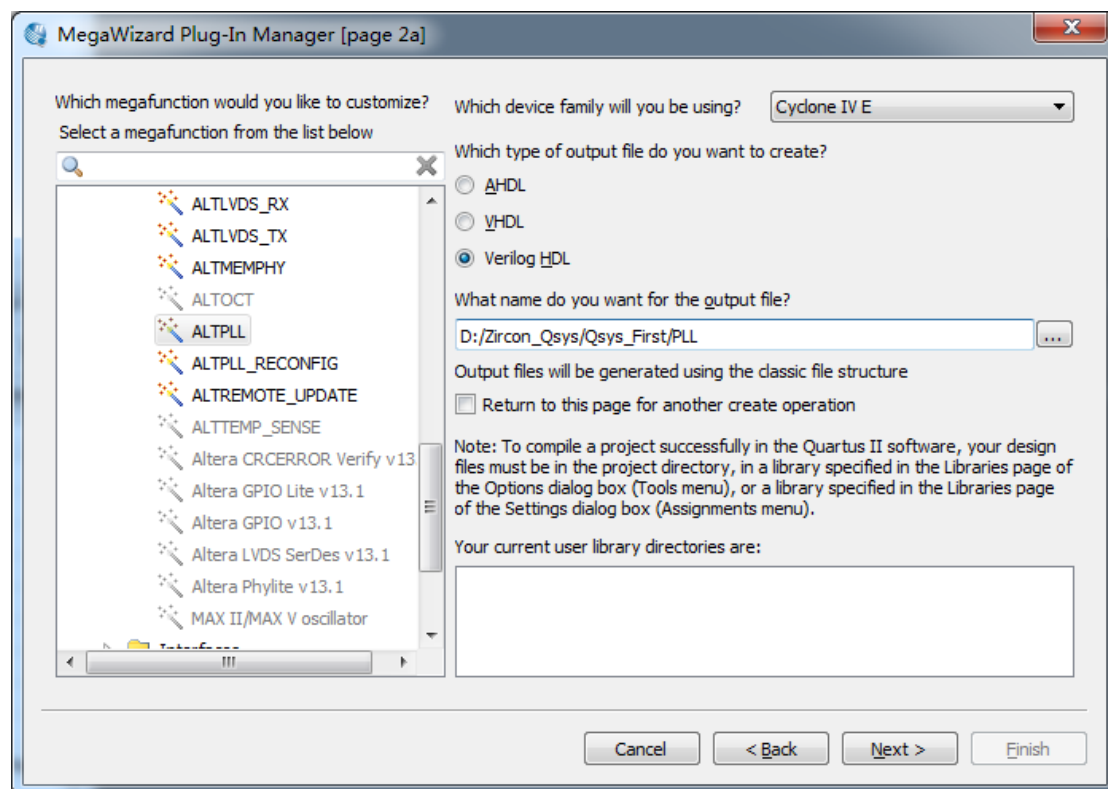


图 2.35 选择创建 PLL 向导页面

接下来我们点击【Next>】进入 PLL IP 核配置向导页面，如图 2.36 所示。由于我们使用的开发板晶振时钟为 50MHz，所以我们这里 PLL IP 核的时钟输入需要设置成 50Mhz。

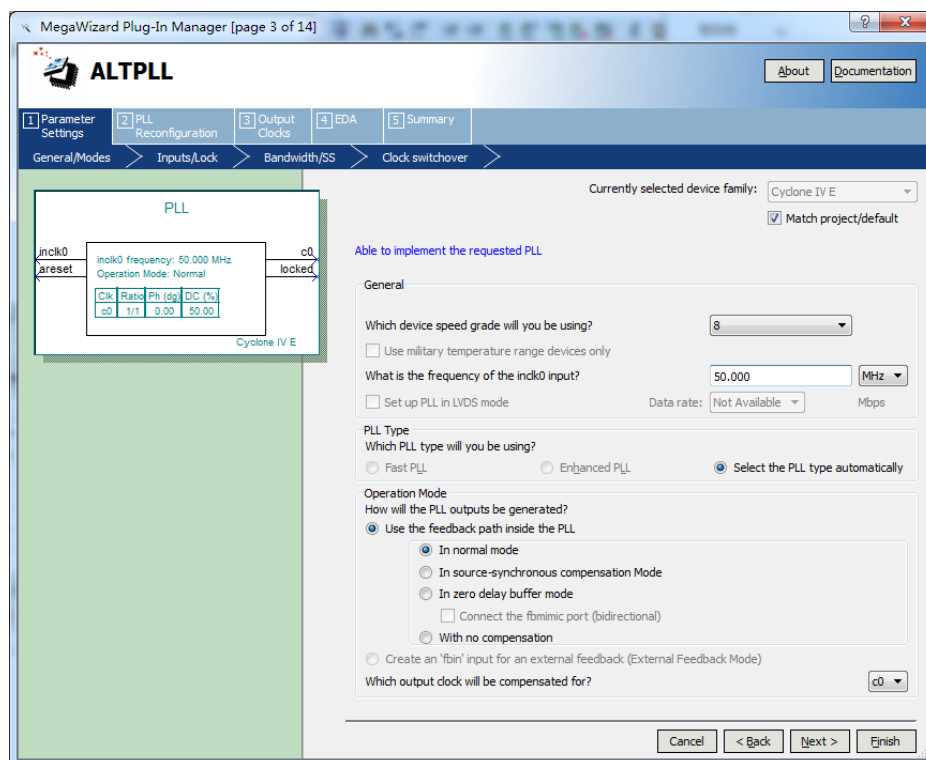


图 2.36 PLL IP 核输入时钟配置向导页面

接下来我们点击【Next>】进入“Inputs/Lock”页面，如图 2.37 所示。由于我们这里用不到上面的两个功能，所以我们这里直接将勾号取消掉。

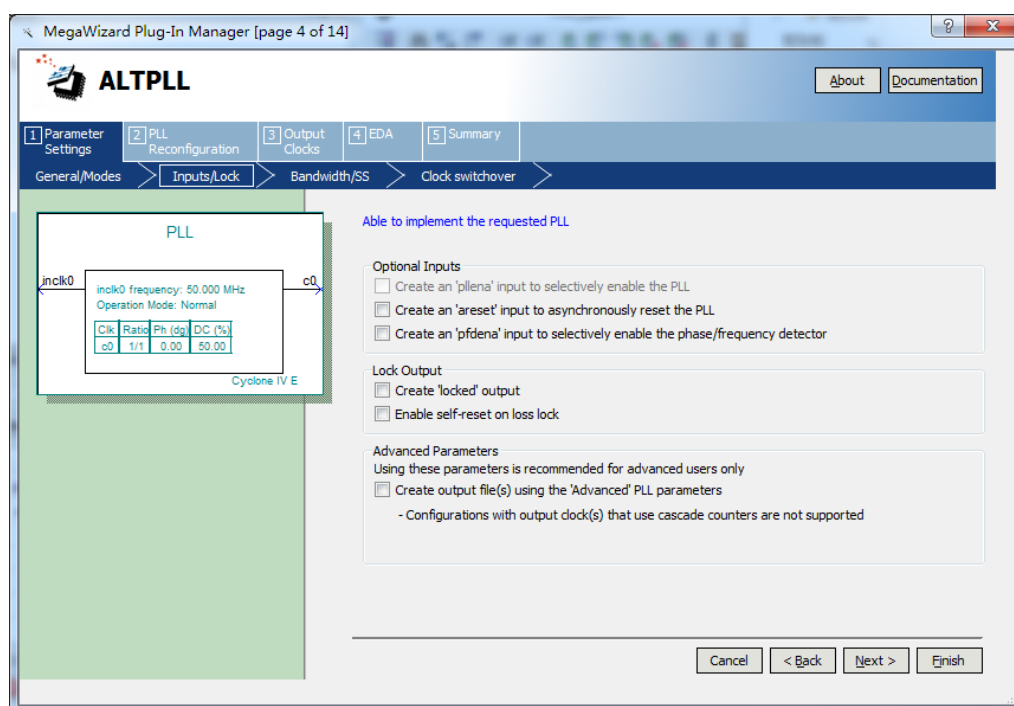


图 2.37 PLL IP 核“Inputs/Lock”配置向导页面

接下来我们一路狂点【Next>】，直到进入“Output Clocks”页面，如图 2.38 所示。由于我们的 Qsys 系统的时钟频率为 100MHz，我们这里的输出时钟也需要设置成 100MHz，以供给 Qsys 系统使用。

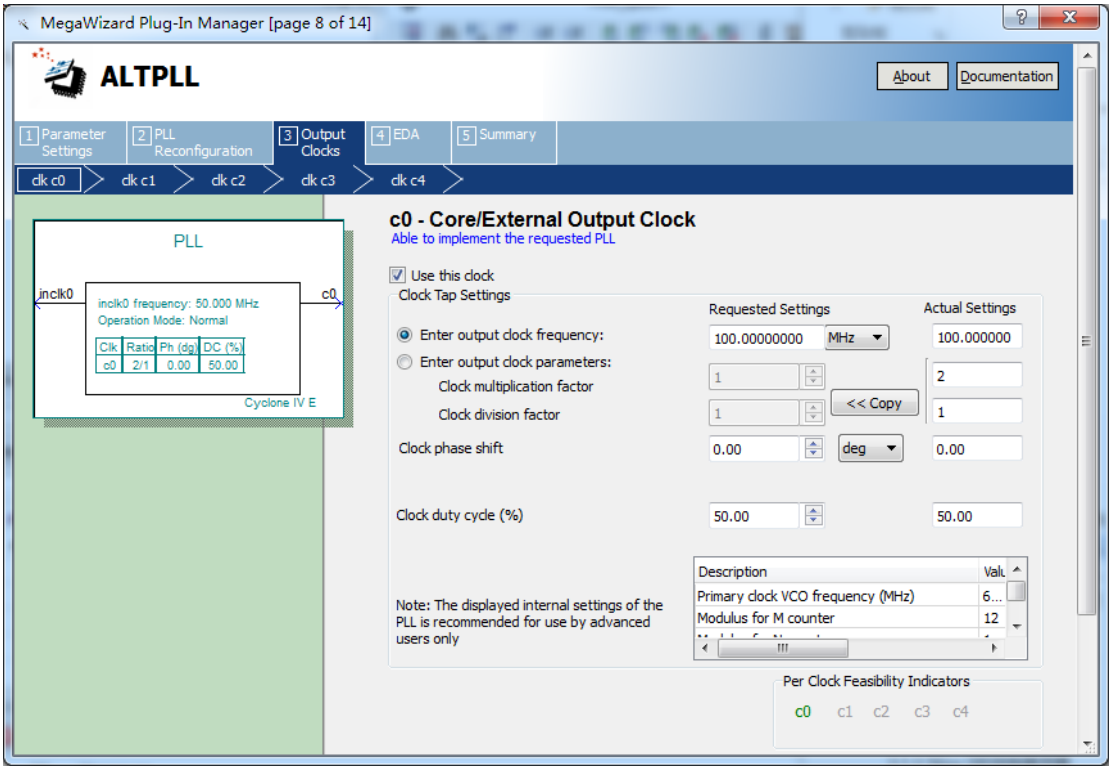


图 2.38 PLL IP 核输出时钟配置向导页面

接下来我们便可以直接点击【Finish】，进入最后的总结页面。如图 2.39 所示。这里我们需要将 PLL_inst.v 勾选上，PLL_inst.v 文件是 ALTPLL IP 核自动生成的使用模板代码，我们只需要将模板中的代码复制到顶层文件中修改端口便可以直接使用。

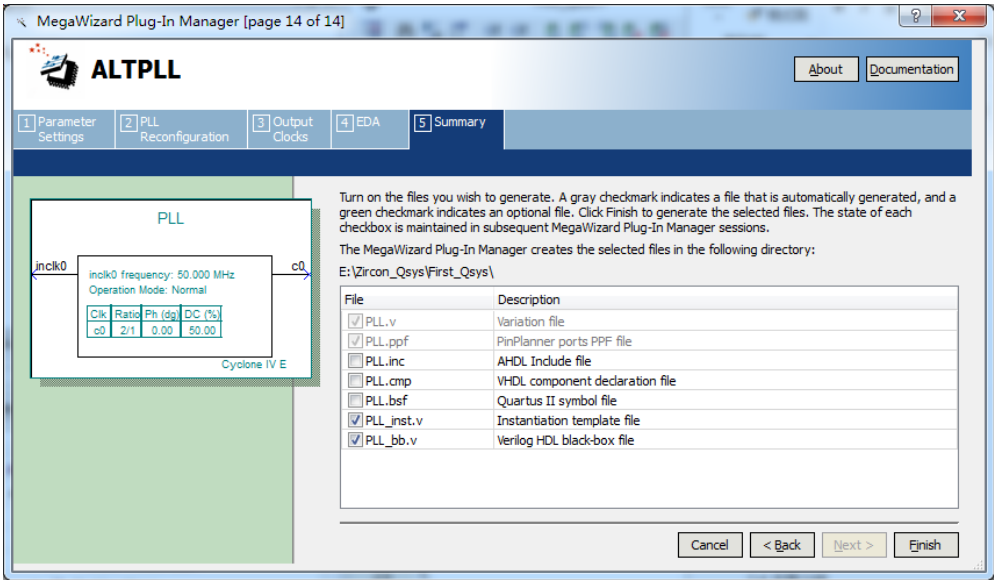


图 2.39 PLL IP 核最后的配置总结页面

我们点击【Finish】完成 PLL IP 核的创建。Quartus II 软件会自动将我们创建的 PLL IP 核自从添加到工程项目中，我们可以在工程文件夹 Qsys_First 中找到生成的 PLL_inst.v 文件，这里我们就不在给出代码内容以做对比。

接下来我们要介绍的是，第二点是 Qsys_system 代码是如何编写的，我们返回到 Qsys 系统软件中，我们单击菜单栏中的【Generate】→【HDL Example…】选项，弹出图 2.40 所示内容。我们只需要点击【Copy】，便可以直接复制到顶层文件中进行修改。

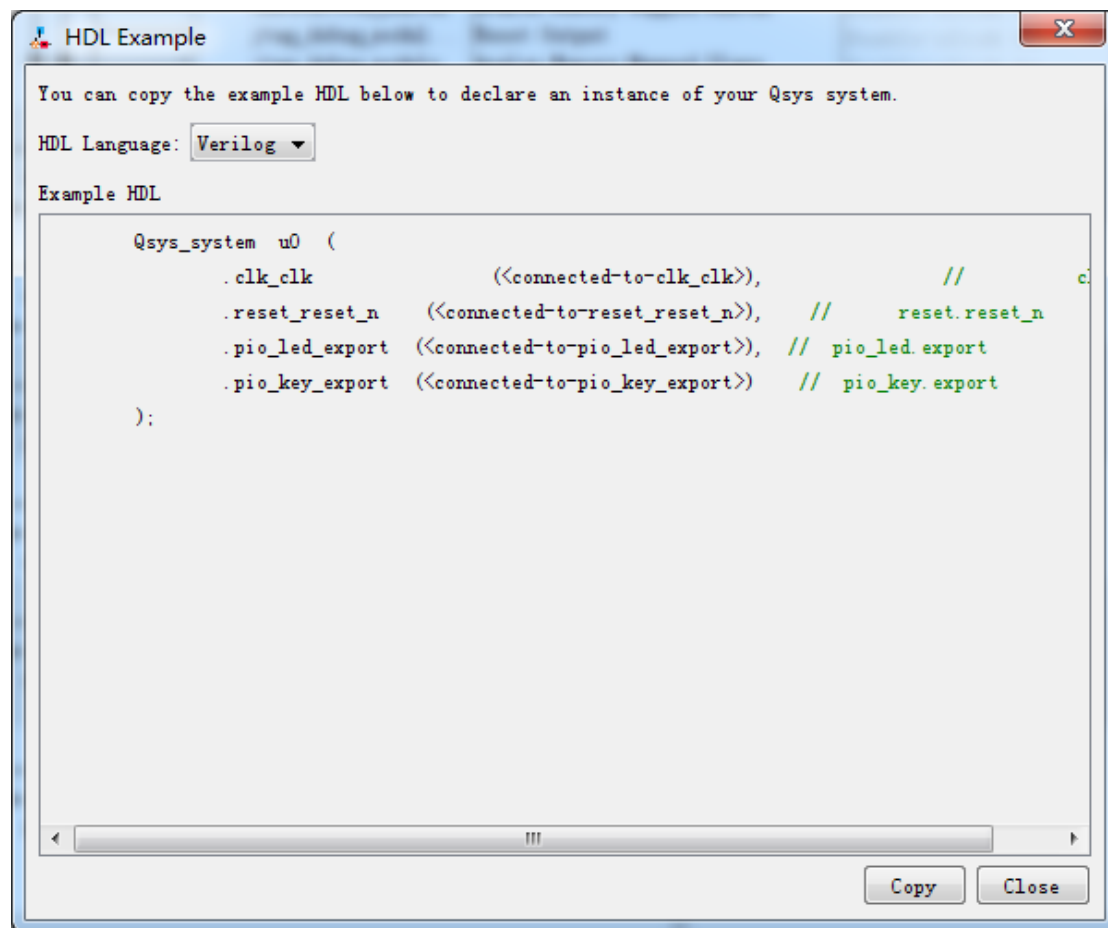


图 2.40 Qsys 系统自动生成的 HDL 代码

接下来我们就需要将 Qsys 系统添加到 Quartus II 工程项目中，进行编译，看看有没有出现一些由于操作失误而引起的小错误。我们可以通过 Quartus 软件菜单栏【Assignment】→【Settings】进入图 2.41 所示页面。

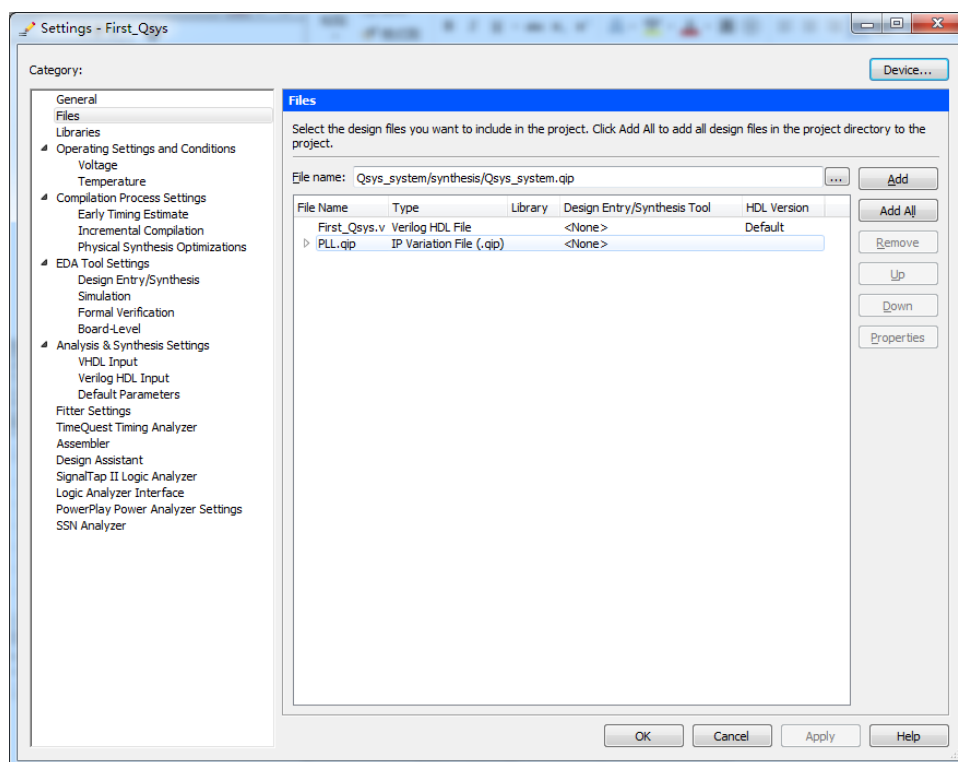


图 2.41 将 Qsys 系统添加到 Quartus 项目工程中

我们点击【…】找到 Qsys_system/synthesis/Qsys_system.qip，这里要注意，一定要记得点击【Add】后，再去点击【OK】，否则将会出现没有添加成功尴尬场面。添加完毕后，我们会在 Project Navigator 窗口中看到我们工程中的所有的文件，如图 2.42 所示。

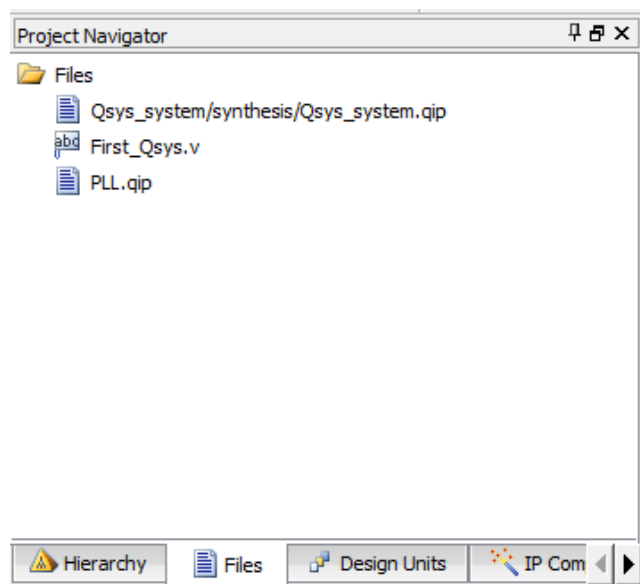


图 2.42 Project Navigator 窗口页面

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，这里的编译速度也是随着电脑性能的越高而越快。我们稍等片刻后。编译就会完成，出现如图 2.43 所示。

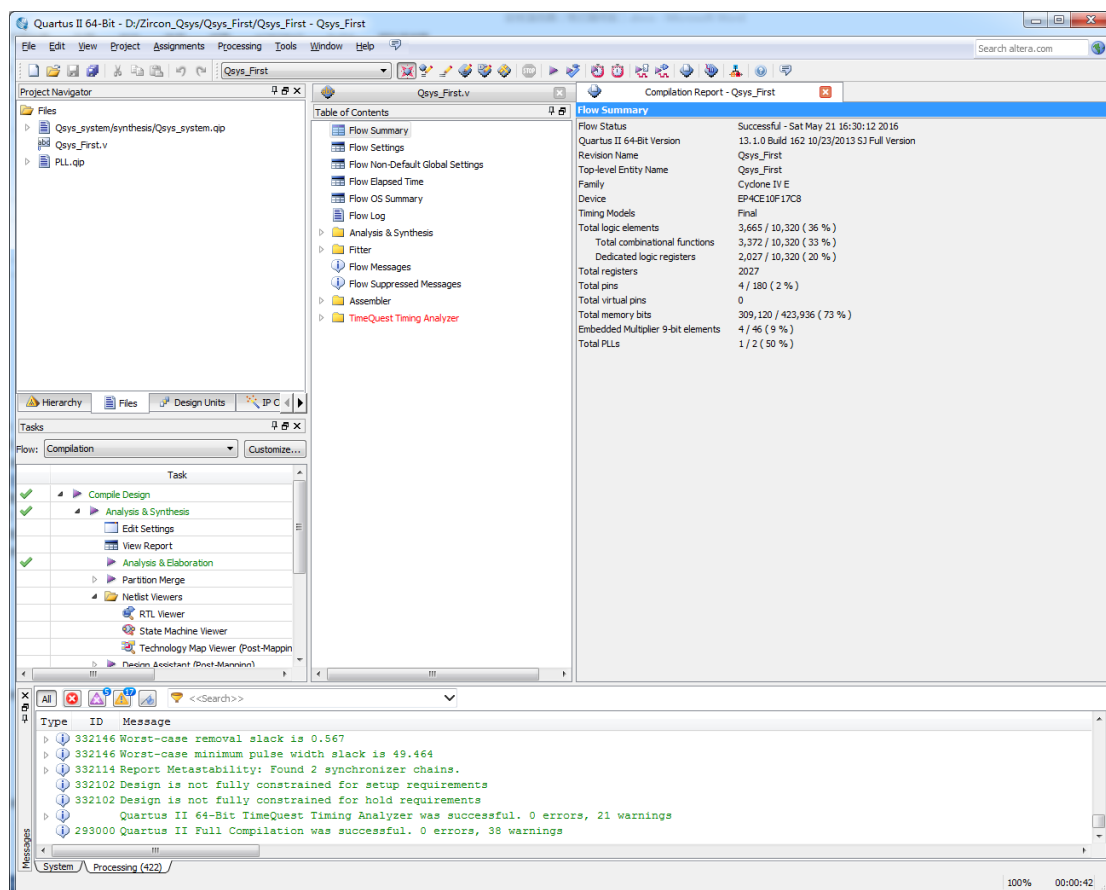


图 2.43 编译完成生成报告窗口页面

接下来我们就需要进行 Quartus II 软件中的扫尾工作，配置 IO，分配管脚，下面我们依次进行设置。首先我们可以通过 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 2.44 所示页面配置 IO。

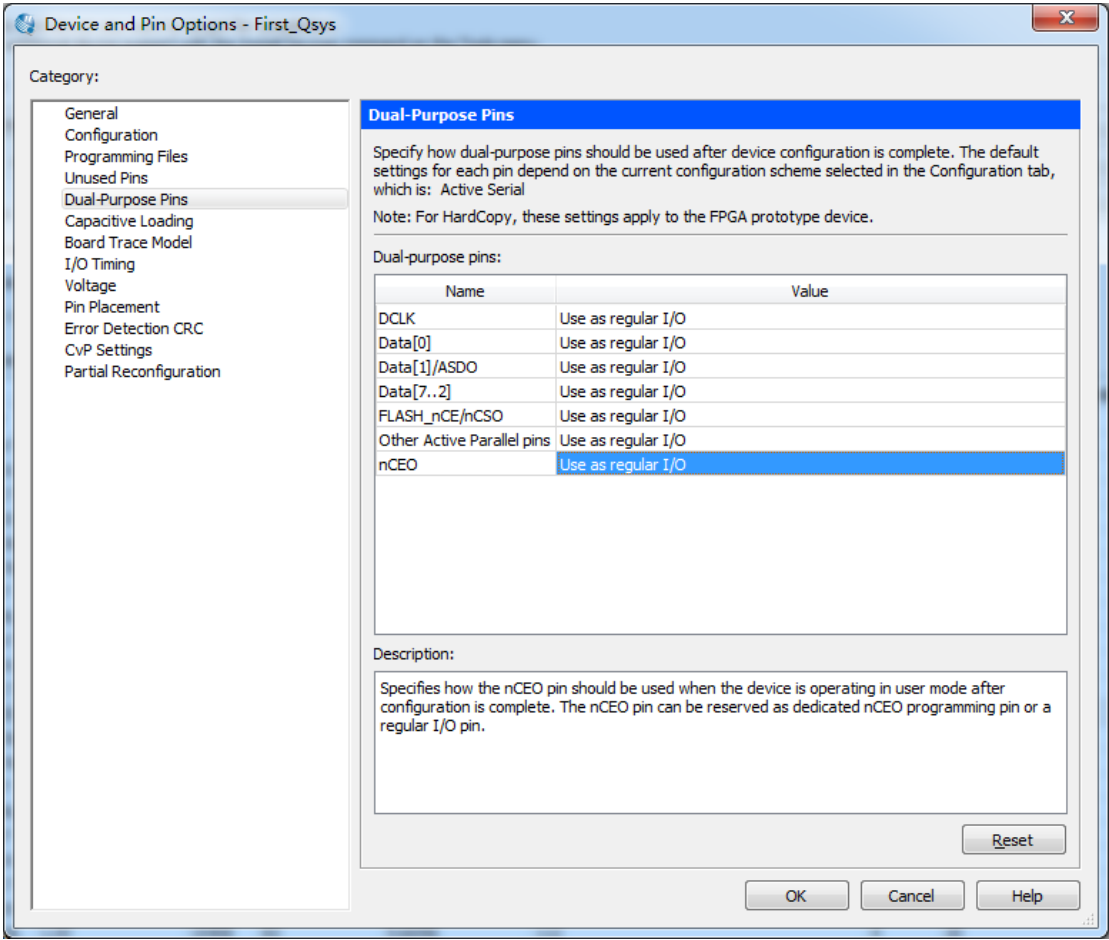


图 2.44 设置 IO 界面

接下来我们便将进入 Unused Pins 页面，对没有使用引脚的进行设置，按照图 2.45 所示，将未使用引脚设置为高阻输入，这样上电后 FPGA 的所有不使用引脚都将进入高阻抗状态。

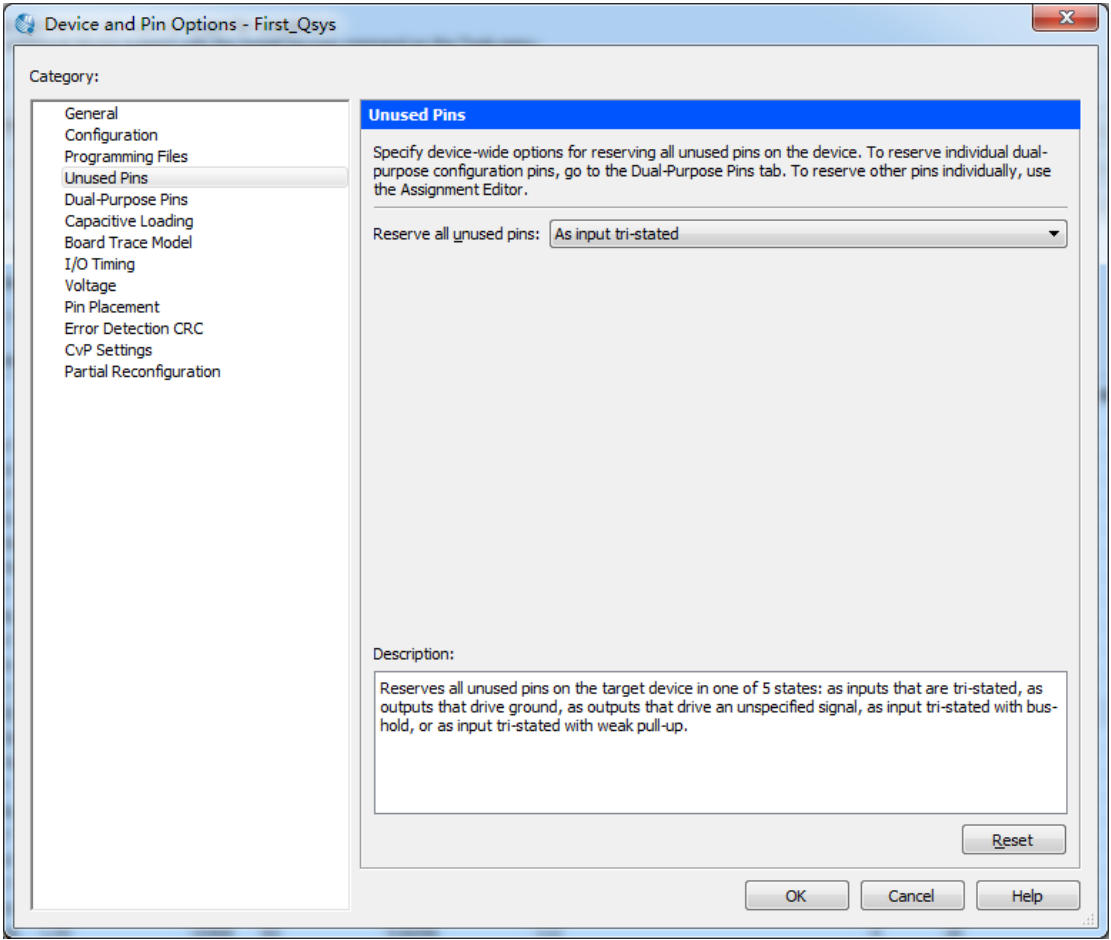


图 2.45 未使用引脚设置界面

最后我们通过 Quartus II 软件菜单栏中的【Assignments】→【Pin Planner】选项进行分配管脚，如图 2.46 所示。

Named: * Edit: [X] [Y]						
Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard
in CLK_50M	Input	PIN_E1	1	B1_N0	PIN_E1	2.5 V (default)
in KEY_PIO	Input	PIN_R9	4	B4_N0	PIN_R12	2.5 V (default)
out LED_PIO	Output	PIN_R11	4	B4_N0	PIN_T11	2.5 V (default)
in RST_N	Input	PIN_F11	7	B7_N0	PIN_F10	2.5 V (default)
<<new node>>						

图 2.46 管脚分配界面

最后我们在进行一次全编译，在编译过程中，可能产生很多警告信息，但这些不会影响设计结果。成功编译硬件系统后，将产生 Qsys_First.sof 的 FPGA 配置文件输出。下面我们就来讲解一下将.sof 文件下载到目标 FPGA 器件的步骤。

- (1) 通过 USB-Blaster 下载器连接开发板和计算机,接通 A4 开发板电源,如图 2.47 所示;

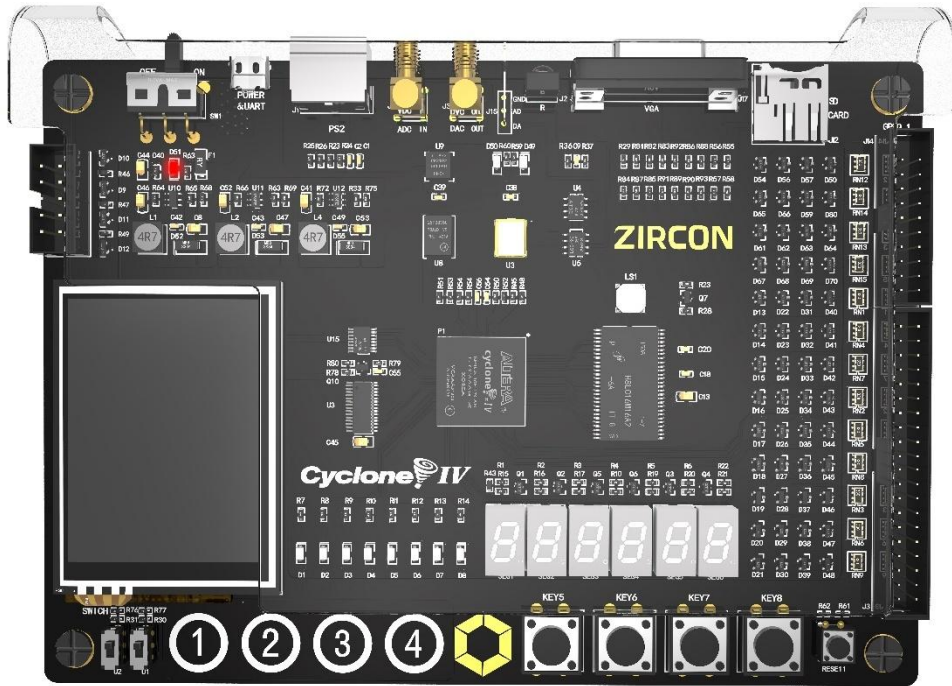


图 2.47 接通 A4 开发板电源示意图

- (2) 在 Quartus II 软件中选择【Tools】→【Programmer】，打开编程器窗口后确保编程器窗口的 Hardware Setup 栏中硬件已经安装。如果硬件没有安装，可以参考《驱动安装指导手册》中的介绍。如图 2.48 所示；

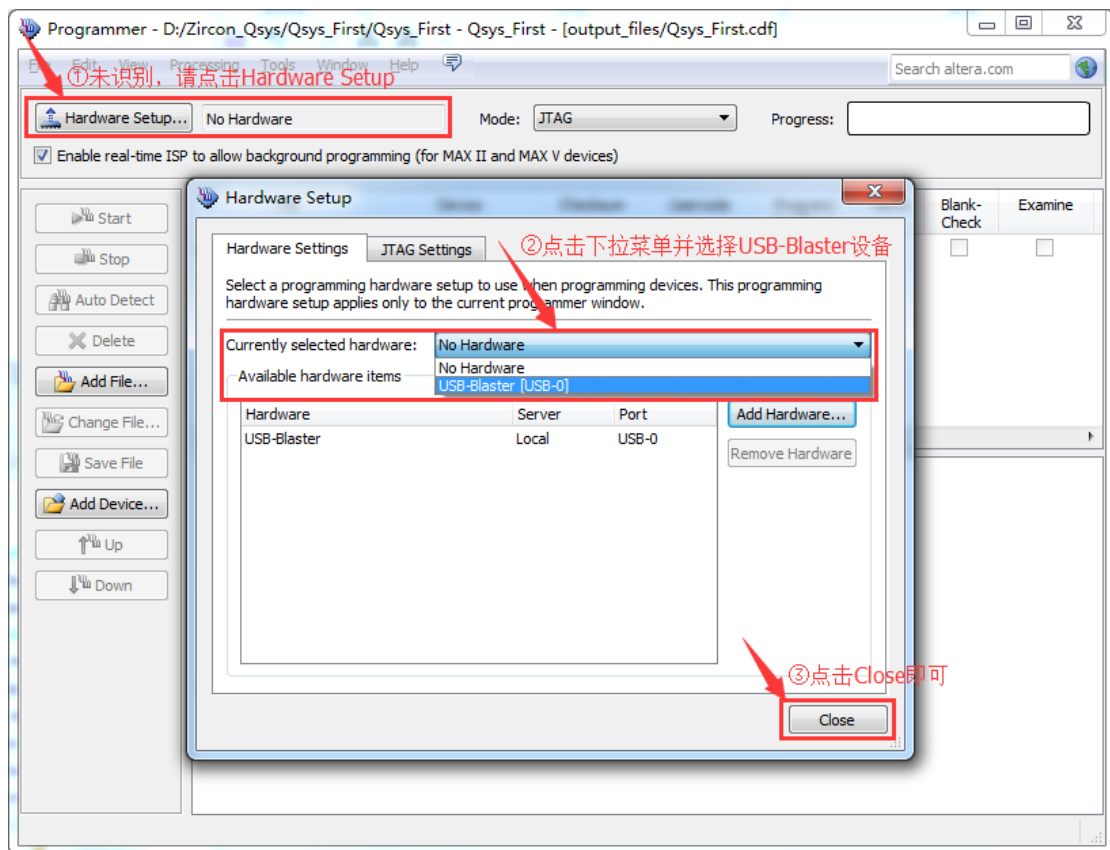


图 2.48 USB-Blaster 设备连接图

- (3) 打开配置文件 (Qsys_First.sof), 确认下载模式, 确保 Program/Configure 下的方框选中, 最后点击【Start】按钮, 开始使用配置文件对 FPGA 进行配置, Progress 栏显示配置进度, 如图 2.49 所示。

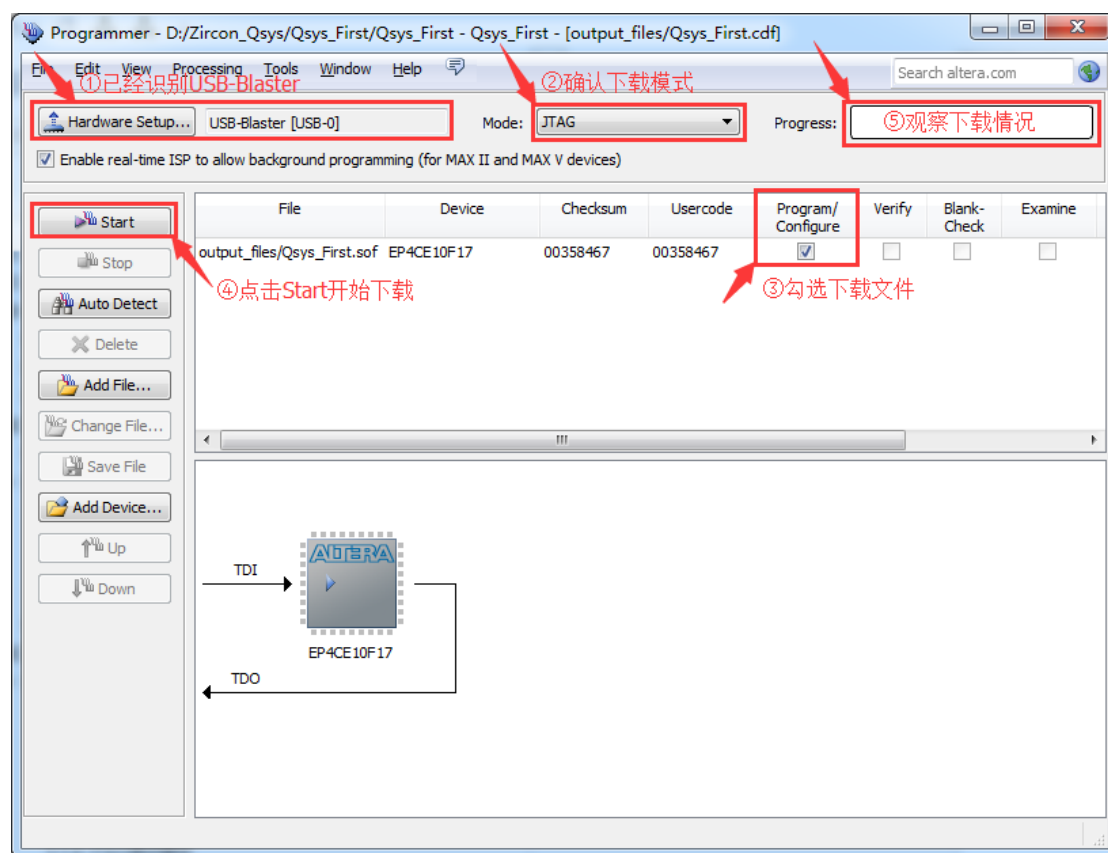


图 2.49 设备连接完成下载图

本节只讲述了将配置文件下载到 FPGA 中, 掉电后 FPGA 中的配置数据将丢失, 可以将配置文件写入掉电保持的 EPCS 器件。在上电时使用 EPCS 对 FPGA 进行配置, 详细内容见 2.3 节中的介绍。至此, 硬件部分设计完成, 下面进行基于 Nios II SBT for Eclipse 的软件方面的设计。

2.2.4 使用 Nios II SBT Eclipse 建立用户程序

我们可以通过 Quartus II 软件菜单栏中的【Tools】→【Nios II SBT for Eclipse】, 来启动 Nios II SBT for Eclipse。打开 Nios II SBT for Eclipse 后, 首先弹出的是 Workspace Launcher 页面, 如图 2.50 所示。

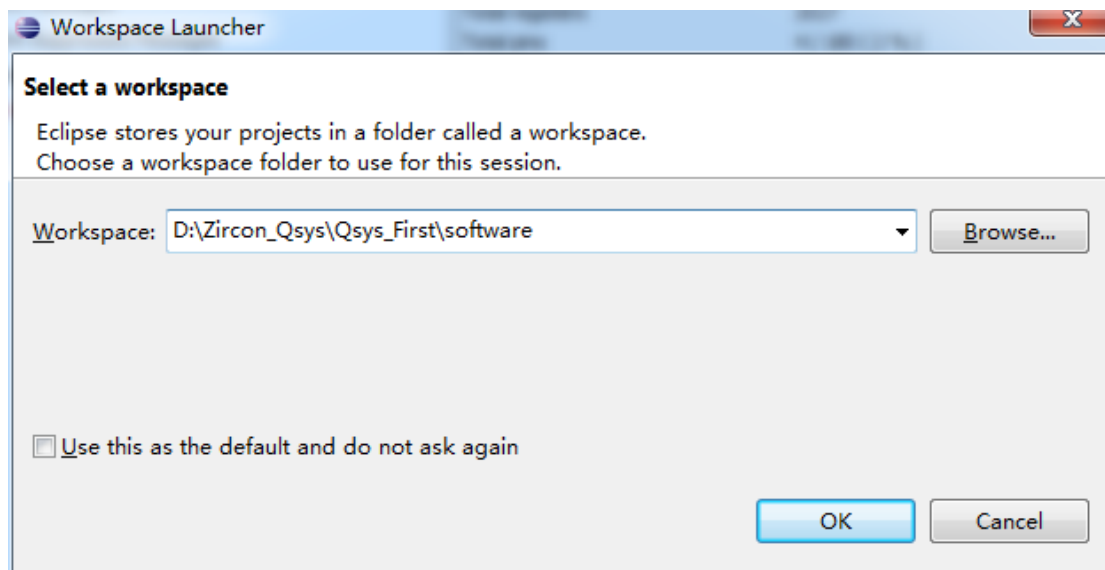


图 2.50 设置 Nios II SBT for Eclipse 工作空间

为了方便工程的管理，我们将 Nios II SBT for Eclipse 工程文件放在我们的 Quartus II 工程项目 Qsys_First 文件夹下的 software 中。

设置好工作空间后，我们点击【OK】进入 Nios II SBT for Eclipse 软件编辑页面中，在该页面我们通过单击菜单栏中的【File】→【New】→【Nios II Application and BSP from Template】，来打开新建工程向导，如图 2.51 所示。

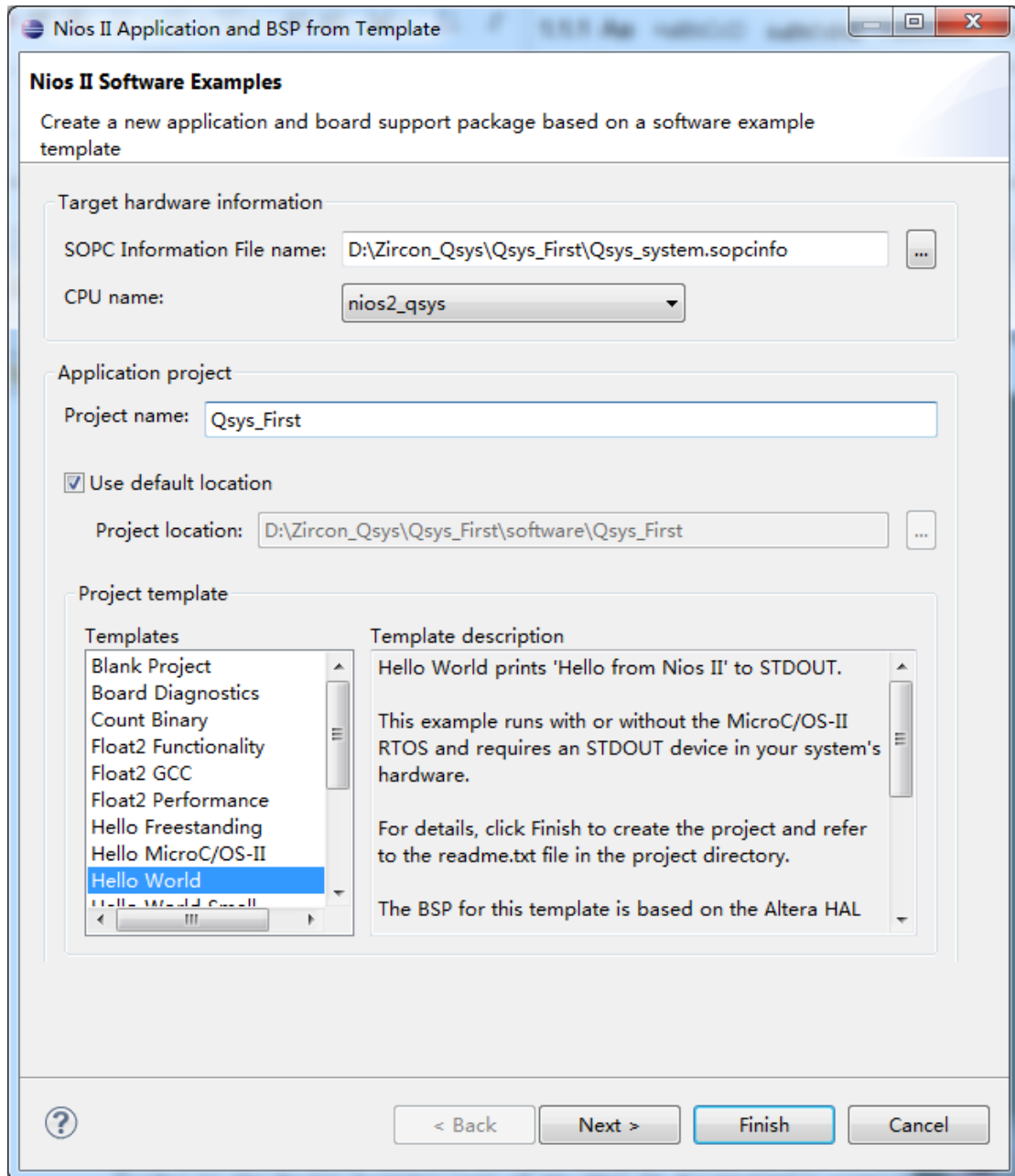


图 2.51 打开新建 Eclipse 工程向导

我们单击【...】按钮来选择 Qsys_First 工程文件夹下的 Qsys_system.sopcinfo，即指向当前硬件设计系统，这时，Nios II SBT for Eclipse 软件会自动识别我们在 Qsys 系统中 CPU 的名称，记忆力比较好的读者一定还记得我们在 Qsys 软件中为我们的 Nios II CPU 核命名为 nios2_qsys。接下来，我们便给我们 Nios II SBT for Eclipse 软件中的工程起个名称，这里的名称可以任意命名，我们这里就命名为 Qsys_First。下面我们在来看下向导中的 Project template 窗口，该窗口中是已经设计好的软件工程，我们可以选择其中的一个，把它当作模板来创建自己的工程。当然也可以选择 Bland Project（空白工程），完全由我们自己写所有的代码。这里我们就以模板为例进行讲解，我们选择的是 Hello World 模板工程，然后我们只需要在此基础上进行适当的修改，一般情况下这比空白工程更加容易，也更方便。

设置好工程后，我们便可以直接点击【Finish】来完成工程创建，这时，你会发现，在 Nios

II SBT for Eclipse 软件的左侧 Project Explorer 窗口中出现两个新的工程：Qsys_First 和 Qsys_First_bsp。Qsys_First 是 C/C++应用工程，而 Qsys_First_bsp 是描述 Qsys_system 系统硬件细节的系统库，关于系统库的详细信息可参考 3.2 节中的介绍。如图 2.52 所示。

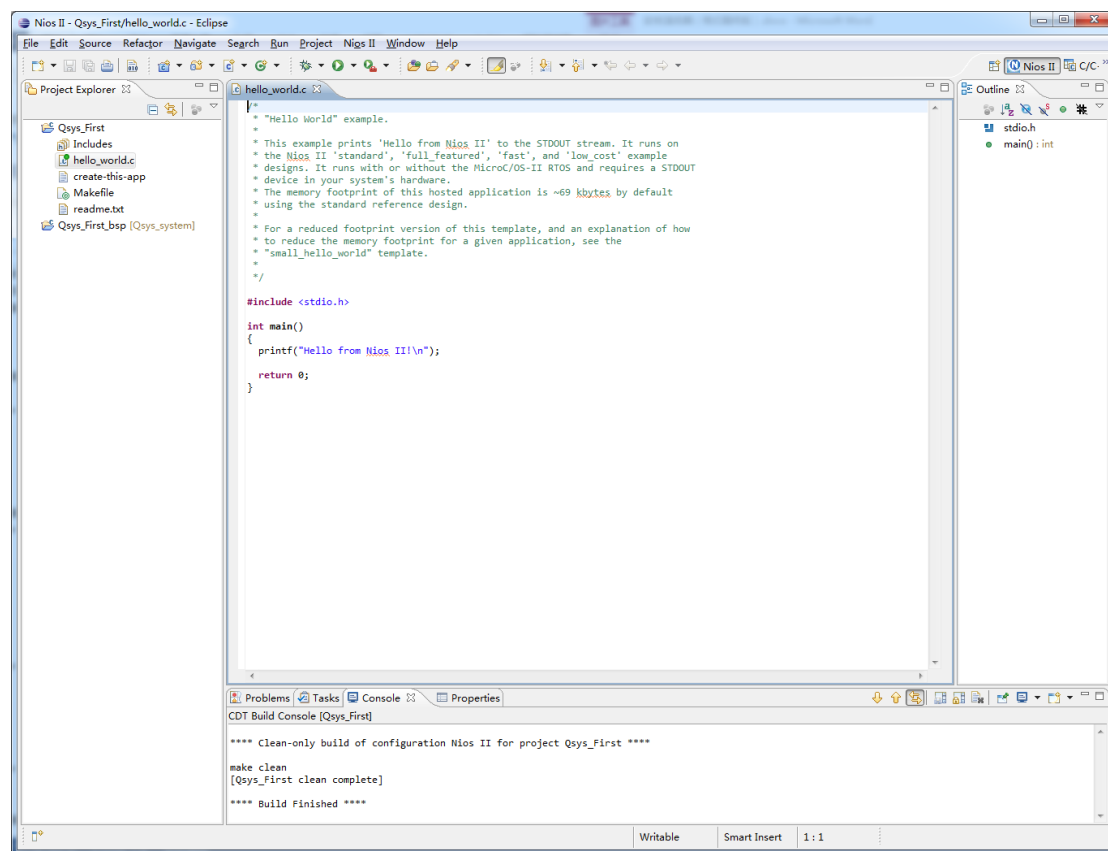


图 2.52 完成工程创建后的软件界面

下面我们右键选择左栏中的 Qsys_First 工程，在出现的菜单中的选择【Build Project】进行工程编译，则会出现如图 2.53 所示错误。

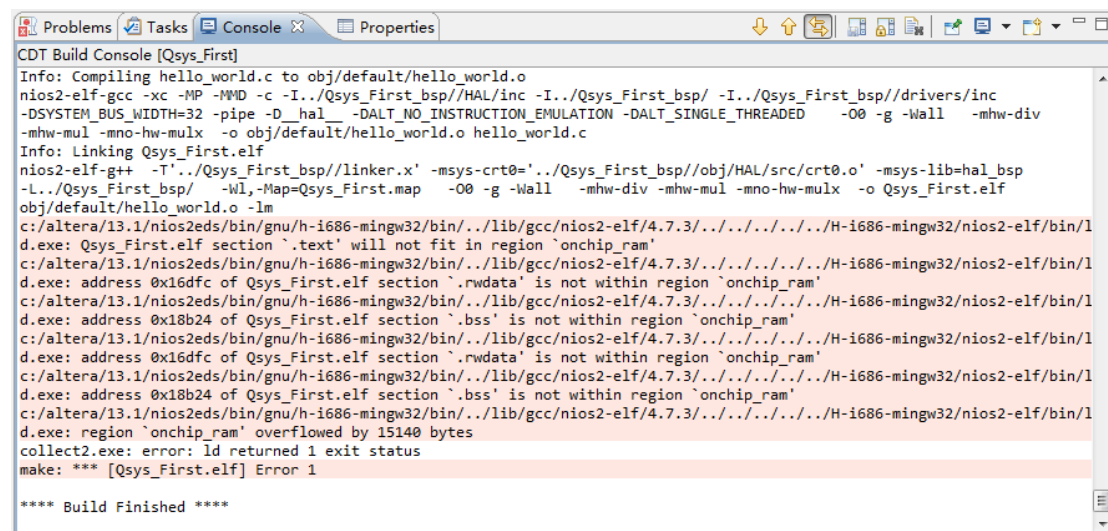


图 2.53 编译错误信息提示

我们可以从错误信息提示中看到,它告诉我们,你的 onchip_ram 太小啦,我还有 15140bytes 的代码没有地方存放,万万没有想到,一个简单的 Hello world 代码竟然这么大, 20K 的片内存储器都放不下,疯了,居然还差 15140bytes...先擦擦冷汗,没事,我们有绝招! Nios 减少代码量的方法有很多,这里我们推荐使用如下方式进行减少代码量。我们右键选择左栏中的 Qsys_First 工程,在出现的菜单中的选择【Nios II】→【BSP Editor...】出现如图 2.54 所示界面。

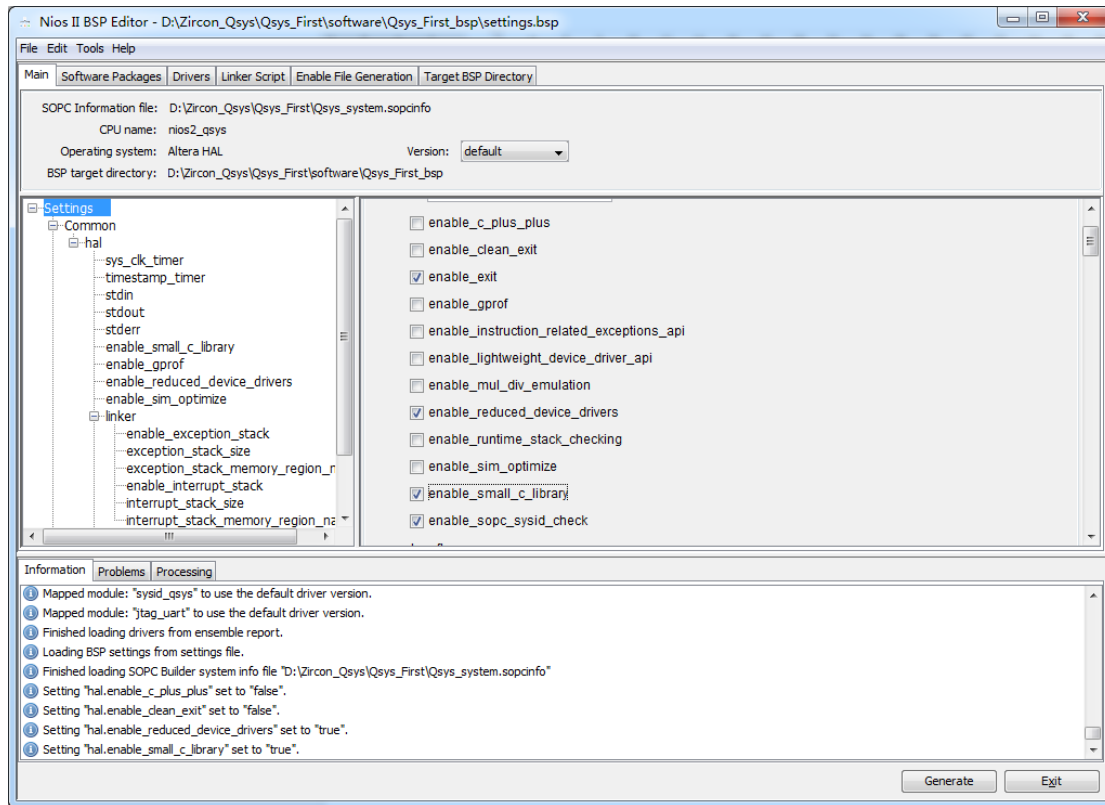


图 2.54 Nios II BSP Editor 配置页面

下面我们就来介绍一下这几个选项:

- (1) enable_c_plus_plus: 由于我们这里用的是 C 语言来编写的软件程序,所以这里我们可以将这个选项去掉。
- (2) enable_clean_exit: 当该选项选中时,系统库在 main()返回时调用 exit(),调用 exit()时,首先清 I/O 的缓冲区,然后调用 _exit()。当该选项不选中时,系统库仅调用 _exit(),能节省程序空间。对于嵌入式系统程序来说,一般都不从 main()返回,所以该选项也可以不用选中。
- (3) enable_reduced_device_drivers: HAL 为处理器的外设提供了两种驱动库:一种是执行速度快,但代码量大的版本;另一种是小封装版本。默认情况下,HAL 系统使用的是代码量大的版本,可以通过【enable_reduced_device_drivers】选项来选择小封装版本,从而减少代码量。
- (4) enable_small_c_library: 完整的 ANSI C 标准库通常不适用于嵌入式系统,HAL 提供了一系列经过裁剪的新的 ANSI C 标准库,占用非常小的代码,我们可以通过

【enalbe_small_c_library】选项来选择新的 ANSI C 标准库。

选择完毕后，我们点击【Generate】，然后点击【Exit】，再一次进行【Build Project】工程编译，如图 2.55 所示。

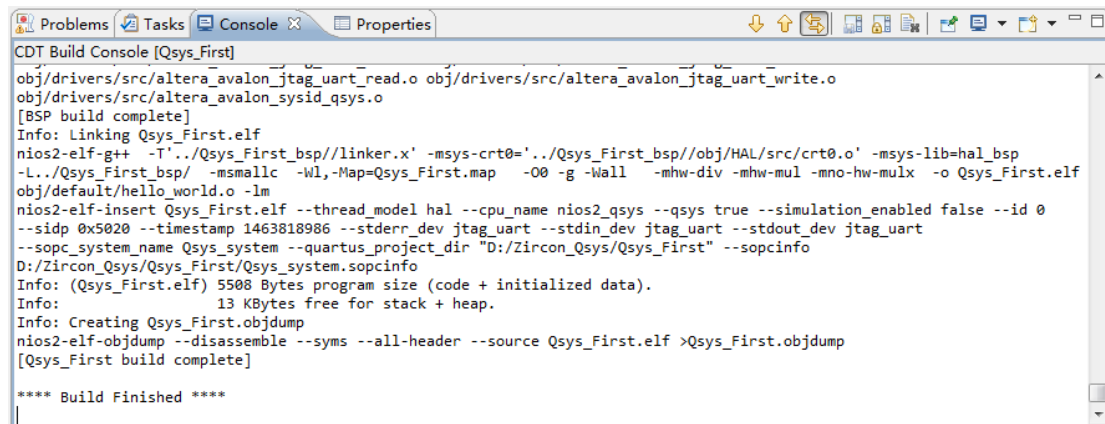


图 2.55 编译成功后的信息提示

这里我们就来看看我们的优化成果，现在仅用 5508B 的代码与初始化数据，对于我们的 20KB 的 onchip_RAM 来说是小菜一碟。这里要注意：虽然优化可以有效减少代码量，但有的时候也会带来意想不到的问题，所以大家请慎用。遇到异常时，先关闭优化，Clean Project 代码后再次编译，测试一下是否是优化所致。

一切都已就绪，那么我们就先来用 Hello world 工程来测试我们的硬件系统是否正常，在开始下载 Frist_Qsys.elf 文件之前，我们要确保两点，第一点，开发板已经通过 USB-Blaster 下载器连接至计算机并打开了开发板电源；第二点，通过 Quartus II 软件中的 Programmer 程序将我们的硬件系统 Qsys_First.sof 下载至开发板。如果以上两点都已经完成，那么才可以进行我们的下一步，右键 Qsys_First 工程，在出现的菜单栏中选择【Run As】→【Nios II Hardware】，出现图 2.56 所示页面。

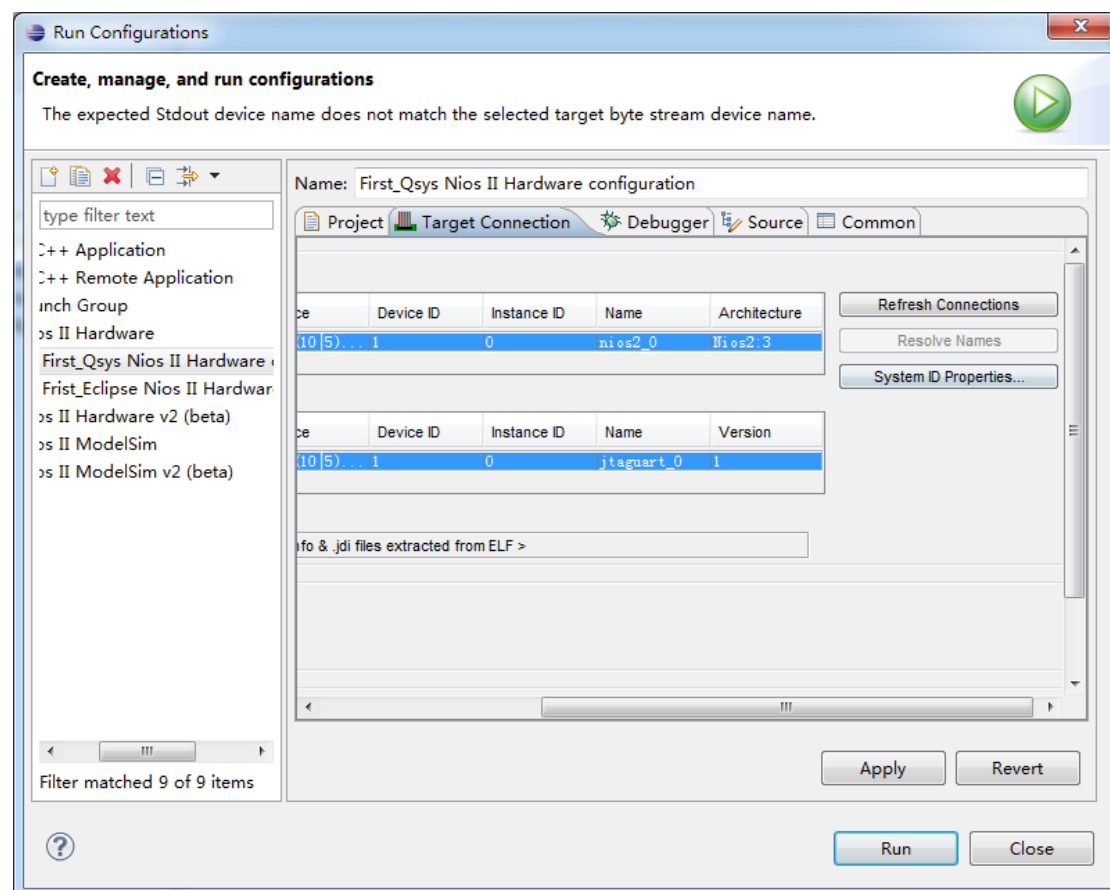


图 2.56 Run Configurations 配置页面

我们在该页面中选择【Target Connection】标签,在 Target Connection 窗口中单击【Refresh Connections】按钮后,软件便会自动识别我们开发板上的 Qsys 系统,并给出 Qsys 系统的一些信息。我们点击【Run】,程序便会自动将 Frist_Qsys.elf 文件下载至开发板中自动进行运行。如图 2.57 所示。

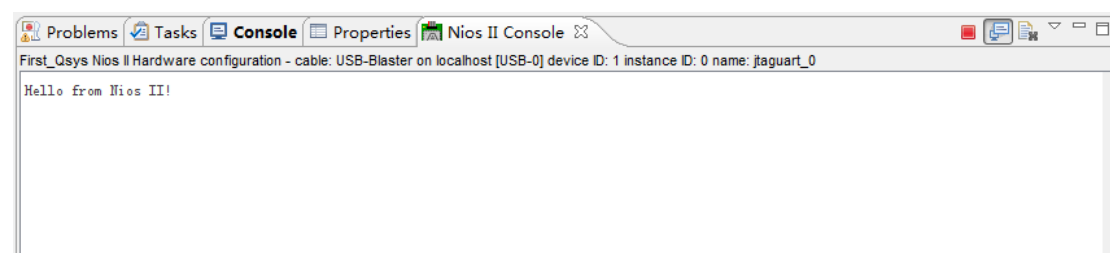


图 2.57 程序运行输出结果窗口

Nios II SBT for Eclipse 中每个工作界面都包括一个或多个窗口,每个窗口都有其特定的功能。在工作界面中包括的主要窗口有编辑器窗口和一个或多个浏览器。编辑器可以用于打开并编辑一个工程。浏览器用于对编辑器提供各种支持,可由我们根据需要进行选择。我们可以同时打开多个编辑器,但在同一时刻只能有一个编辑器处于激活状态。在工作界面上的主菜单和工具条的各种操作只对处于激活状态的编辑器起作用。在编辑区中的各个标签上是当前被打开的文件名,带有“*”号的标签表示这个编辑器中的内容还没有被保存。接下来我们只需要更改 hello_world.c 中的代码就可以完成我们的软件设计,我们将 hello_world.c 文件中的代码更改,

如代码 2.2 所示。

代码 2.2 hello_world.c 代码

```

1  //-----
2  //-- 文件名    : hello_world.c
3  //-- 描述      : 使用按键中断控制 LED 亮灭
4  //-- 修订历史  : 2014-1-1
5  //-- 作者      : Zircon Opto-Electronic Technology CO.,Ltd.
6  //-----
7  #include <stdio.h>                //标准的输入输出头文件
8  #include "system.h"              //系统头文件
9  #include "altera_avalon_pio_regs.h" //pio 寄存器头文件
10 #include "alt_types.h"            //数据类型头文件
11 #include "sys/alt_irq.h"          //中断头文件
12
13 alt_u8 LED_Toggle = 0;             //LED 状态切换全局变量
14
15 //-----
16 //-- 名称      : KeyDown_interrupts()
17 //-- 功能      : 中断服务子程序
18 //-- 输入参数  : context:一般用于传递中断状态寄存器的值;id:中断号,这里都未使用
19 //-- 输出参数  : 无
20 //-----
21 static void KeyDown_interrupts(void* context, alt_u32 id)
22 {
23     /* 更改 LED 的显示状态 */
24     LED_Toggle = ~LED_Toggle;
25     /* 清中断捕获寄存器 */
26     IOWR_ALTERA_AVALON_PIO_EDGE_CAP(PIO_KEY_BASE, 0x1);
27 }
28
29 //-----
30 //-- 名称      : Init_Interrupt()
31 //-- 功能      : 中断初始化
32 //-- 输入参数  : 无
33 //-- 输出参数  : 无
34 //-----
35 void Init_Interrupt(void)
36 {
37     /* 开 KEY 的中断 */
38     IOWR_ALTERA_AVALON_PIO_IRQ_MASK(PIO_KEY_BASE, 0x1);
39     /* 清边沿捕获寄存器 */
40     IOWR_ALTERA_AVALON_PIO_EDGE_CAP(PIO_KEY_BASE, 0x1);
41     /* 注册中断服务子程序 */
42     alt_irq_register(PIO_KEY_IRQ, NULL, KeyDown_interrupts);

```

```
43 }
44
45 //-----
46 //-- 名称      : main()
47 //-- 功能      : 程序入口
48 //-- 输入参数  : 无
49 //-- 输出参数  : 无
50 //-----
51 int main(void)
52 {
53     printf("Code Running...");
54     Init_Interrupt();
55
56     while(1)
57     {
58         IOWR_ALTERA_AVALON_PIO_DATA(PIO_LED_BASE, LED_Toggle);
59     }
60
61     return(0);
62 }
```

由于该程序代码以 C 语言为主，所以我们就不再进一步对 C 语言进行讲解了。这里我们主要说明两点，第一点是程序中的外设地址是如何来的，这部分的详细内容可以参考第 3.2.1 节中的介绍。第二点要说明的便是程序中的中断问题，这部分的详细内容可以参考第 3.2.5 节中的介绍。代码更改完毕后，接下来我们就将该工程下载至我们的 A4 开发板进行验证，首先我们需要在 Quartus II 软件中将 Qsys_First.sof 下载至我们的 A4 开发板，Qsys_First.sof 下载完成后，我们还需要在 Eclipse 软件中将 Qsys_First.elf 文件下载至我们的 A4 开发板，Qsys_First.elf 下载完成以后，我们的 C 程序将会执行在我们的 A4 开发板上，我们可以看到 A4 开发板上的 D1 被点亮，如图 2.58 所示。

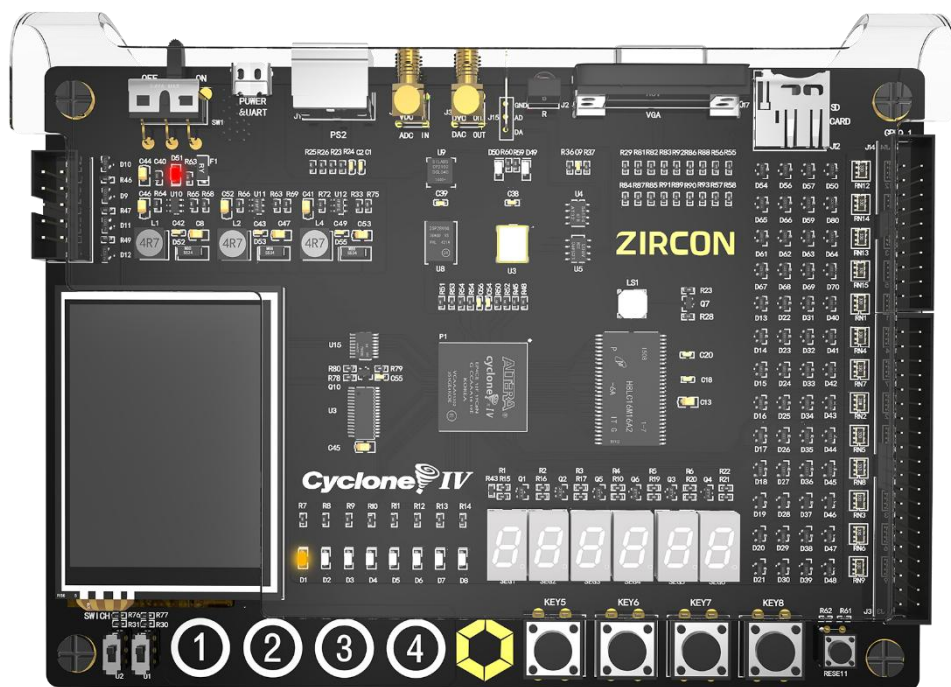


图 2.58 A4 开发板运行图

与此同时，我们还可以在 Eclipse 软件的控制台中看到我们的打印信息，如图 2.59 所示。

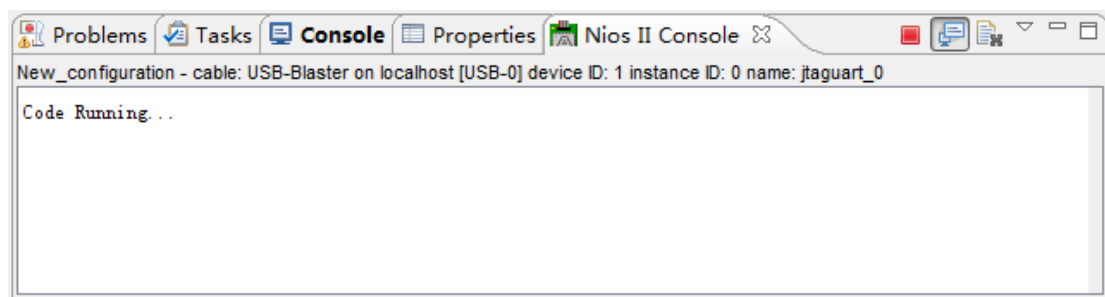


图 2.59 Eclipse 软件的控制台中打印的信息图

此时，如果我们按下触摸按键 KEY1，那么 A4 开发板上的 D1 就会熄灭，我们再一次按下，D1 就会点亮。至此，我们就完成了整个工程。最后我们再补充说明一点，如果在硬件配置成功后，开发板上不能正常运行程序，说明我们的程序存在问题，我们可以通过调试来解决。启动调试程序和启动运行程序类似，我们右键选择左栏中的 Qsys_First 工程，在出现的菜单中的选择【Debug As】→【Nios II Hardwar】，在出现的对话框中我们点击【Yes】，调试运行后，将会出现如图 2.60 所示页面。

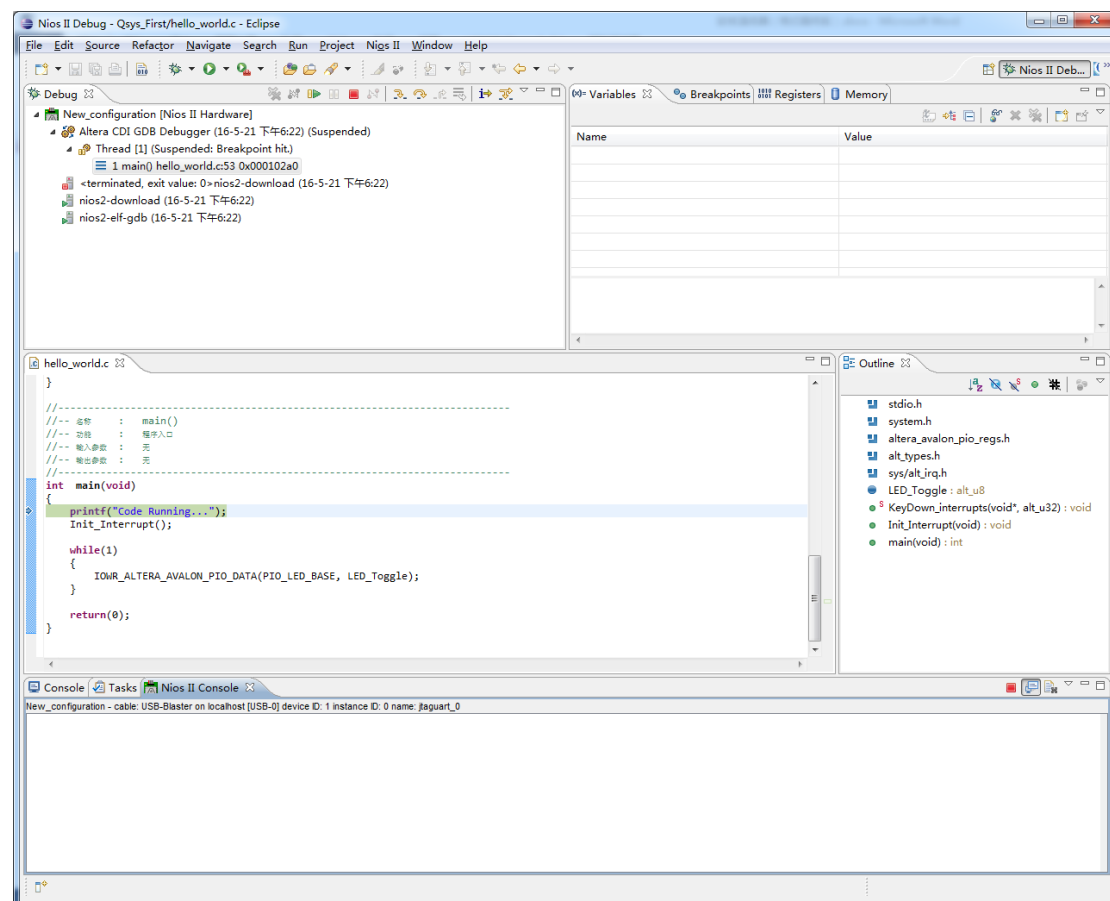


图 2.60 程序调试界面

在调试页面，我们可以通过选取图 2.61 所示的通用调试控制工具来跟踪程序的运行。当然，我们还可以在程序运行观察窗口中可以观察程序的运行情况；在局部变量观察窗口中可以观察变量、寄存器、存储器的值等。

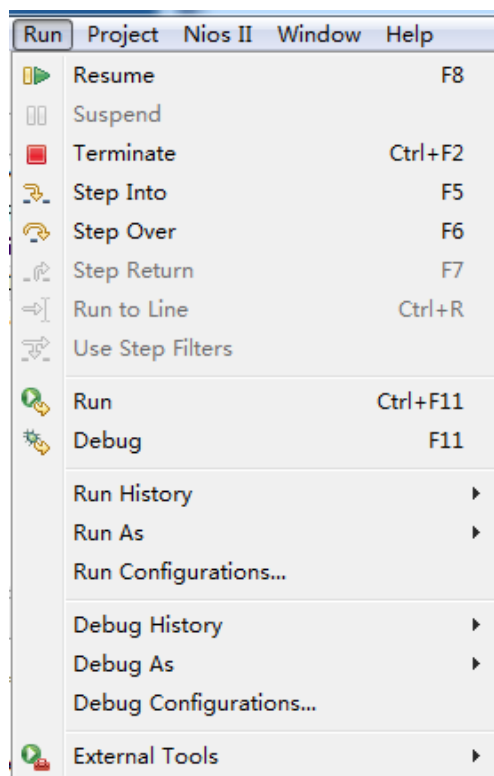


图 2.61 调试工具菜单栏

这里我们就简单介绍一下常用的几个调试命令：

- (1) Resume: 从当前代码处继续执行；
- (2) Suspend: 暂停运行；
- (3) Terminate: 停止调试；
- (4) Step Into: 单步跟踪时进入子程序；
- (5) Step Over: 单步跟踪时不进入子程序；
- (6) Step Return: 运行并跳转出子程序；
- (7) Run to Line: 运行到光标所在的当前行；

至此，我们已经建立好了我们的第一个 Qsys 系统，但是这里要注意，在这里我们只讲述了将配置文件下载到 FPGA 中，掉电后 FPGA 中的配置数据将丢失，如果想要程序不丢失，那么可以将配置文件写入掉电保持的 EPCS 器件中，在上电时使用 EPCS 对 FPGA 进行配置，接下来我们就来讲解让你的 Qsys 系统上电自启动吧。

§ 2.3 让你的Qsys系统上电自启动吧

由于前面已经给出了建立工程的详细步骤，这里我们就不在手把手的一步建立工程，这里我们会给出几个关键部分，想要让 Qsys 系统上电自启动，我们需要将配置文件写入掉电保持的 ECPS 器件中，因此我们就需要在我们的 Qsys 系统中添加 EPCS 组件。我们这里就直接在第一个工程上进行修改，我们将修改完毕的 Qsys 系统直接给出，Qsys 系统如图 2.62 所示。

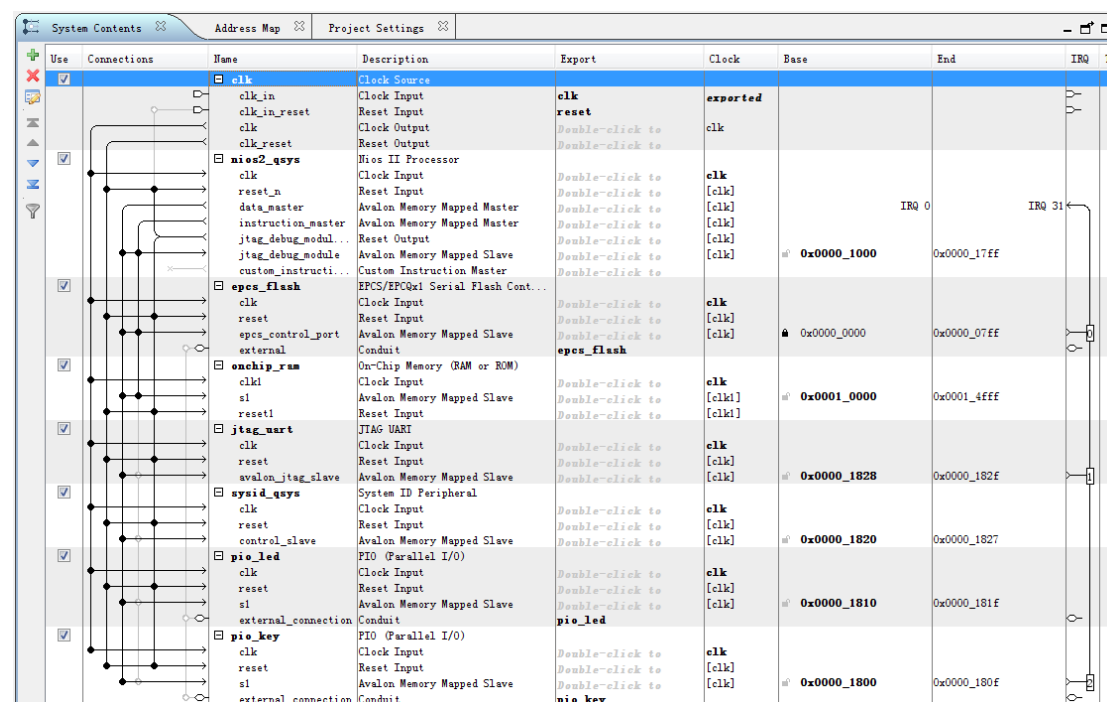


图 2.62 添加了 EPCS 的 Qsys 系统

从图 2.62 中,我们可以看出,我们将 onchip_rom 组件换成了 epcs_flash,这里的 epcs_flash 配置,我们没有改变任何的选项,我们是直接使用的默认配置,关于 EPCS 组件的详细内容请参考 4.3 节中的介绍。添加好了 EPCS 组件后,我们修改了名称,引出了管脚,并添加了中断,这里要注意的是,和 onchip_rom 一样, epcs_flash 的地址也是上了锁的。还有一点不要忘记, nios2_qsys 中的 Reset vector memory 需要改成 epcs_flash。完成了 Qsys 系统的创建后,接下来就要修改 Quartus II 工程中顶层文件的代码,下面我们直接给出修改完成的代码,如代码 2.3 所示。

代码 2.3 Qsys_First.v 代码

```

1  module Qsys_First
2  (
3      //输入端口
4      CLK_50M, RST_N, KEY_PIO, EPCS_DATA0,
5      //输出端口
6      LED_PIO, EPCS_DCLK, EPCS_SCE, EPCS_SDO
7  );
8
9  //-----
10 //--  外部端口声明
11 //-----
12 input          CLK_50M;
13 input          RST_N;
14 input          KEY_PIO;
15 output         LED_PIO;
16 input          EPCS_DATA0;

```

```

17 output          EPCS_DCLK;
18 output          EPCS_SCE;
19 output          EPCS_SDO;
20 //-----
21 //-- 内部端口声明
22 //-----
23 wire            clk_100m;
24
25 //-----
26 //-- 逻辑功能实现
27 //-----
28 PLL             PLL_Init
29 (
30   .inclk0        (CLK_50M  ),
31   .c0            (clk_100m )
32 );
33
34 //-----
35
36 Qsys_system     Qsys_system_Init
37 (
38   .clk_clk        (clk_100m ), //      clk.clk
39   .reset_reset_n  (RST_N    ), //      reset.reset_n
40   .pio_led_export (LED_PIO   ), //      pio_led.export
41   .pio_key_export (KEY_PIO   ), //      pio_key.export
42   .epcs_flash_dclk (EPCS_DCLK), //epcs_flash.dclk
43   .epcs_flash_sce  (EPCS_SCE ), //      .sce
44   .epcs_flash_sdo  (EPCS_SDO ), //      .sdo
45   .epcs_flash_data0 (EPCS_DATA0 ) //      .data0
46 );
47
48 endmodule

```

从代码中我们可以看出，我们仅增加了 epcs 的四个管脚，接下来就是编译，分配管脚，如图 2.63 所示，最后我们编译并生成 Qsys_First.sof 并将该文件下载至开发板。

X Named: * Edit: CLK_50M						
Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard
in CLK_50M	Input	PIN_E1	1	B1_N0	PIN_E1	2.5 V (default)
in EPCS_DATA0	Input	PIN_H2	1	B1_N0	PIN_E16	2.5 V (default)
out EPCS_DCLK	Output	PIN_H1	1	B1_N0	PIN_T10	2.5 V (default)
out EPCS_SCE	Output	PIN_D2	1	B1_N0	PIN_T9	2.5 V (default)
out EPCS_SDO	Output	PIN_C1	1	B1_N0	PIN_R8	2.5 V (default)
in KEY_PIO	Input	PIN_R9	4	B4_N0	PIN_R9	2.5 V (default)
out LED_PIO	Output	PIN_R11	4	B4_N0	PIN_R11	2.5 V (default)
in RST_N	Input	PIN_F11	7	B7_N0	PIN_F11	2.5 V (default)
<<new node>>						

图 2.63 EPCS 的管脚分配图

接下来便是 Nios II SBT for Eclipse 工程了, 由于 Qsys 系统有改动, 所以我们必须要在 Nios II BSP Editor 中重新【Generate】一次。我们打开右键 Nios II SBT for Eclipse 软件在工程目录中找到 Qsys_First 工程右键选择【Nios II】→【BSP Editor】, 这里我们会发现之前减少代码量的配置依然没有改变, 我们打开 Linker Script 标签如图 2.64 所示。

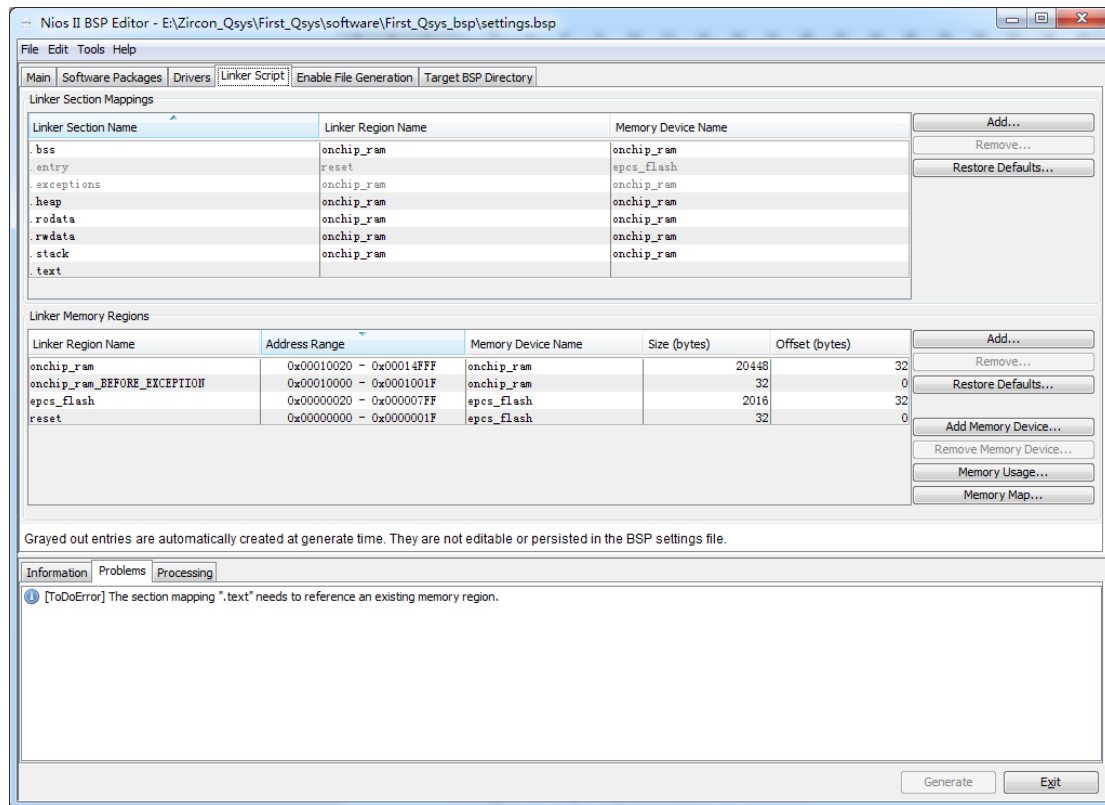


图 2.64 Linker Script 配置页面

从该图中我们可以看出, 由于我们在 Qsys 软件中将 Reset Address 设置成了 epcs_flash, 所以这里的.entry 也变成了 epcs_flash, 关于.text、.stack、.rdata、.rodata、.heap、.bss 详细内容请参考 3.3.1 节中的介绍。我们再一次点击【Generate】, 随后我们将工程重新编译一次, 生成新的 Qsys_First.elf, Qsys_First.elf 文件生成好了以后, 我们就可以单击 Nios II SBT for Eclipse 软件菜单栏中的【Nios II】→【Flash Programmer】, 将会出现如图 2.65 所示。

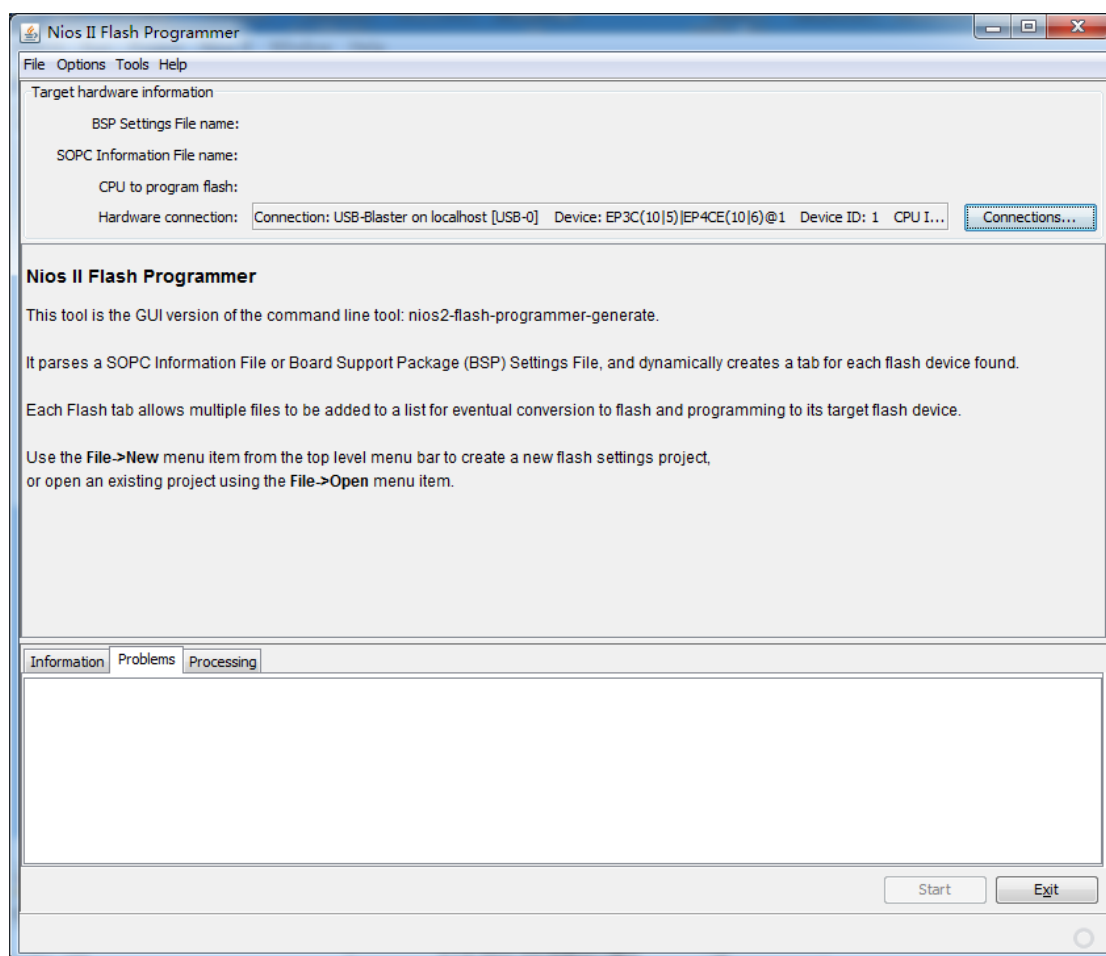


图 2.65 Nios II Flash Programmer 界面

接下来我们点击 Nios II Flash Programmer 菜单栏中的【File】→【New...】，如图 2.66 所示。

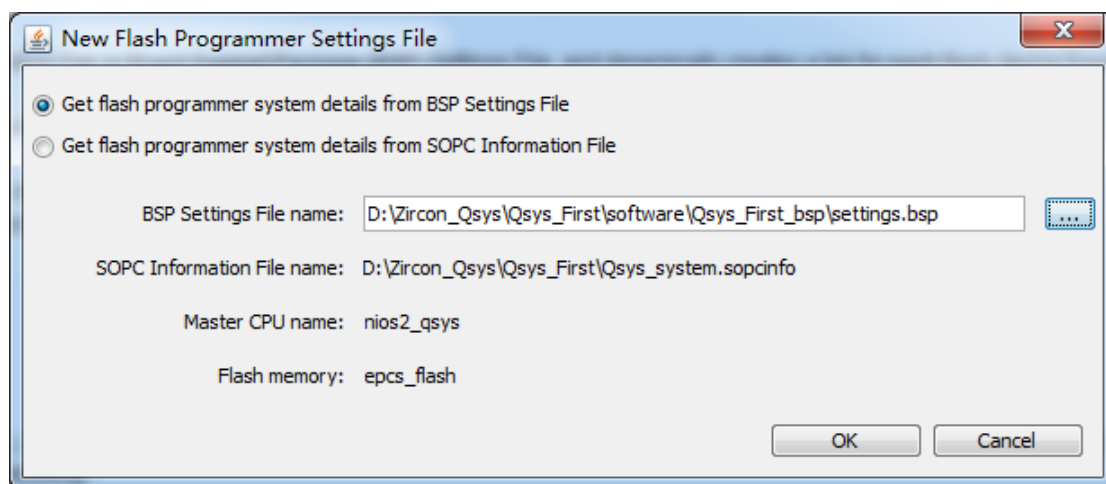


图 2.66 New Flash Programmer Settings File 界面

从图 2.66 中我们可以看到有两种方法，一种方法是利用 settings.bsp 文件，另一种方法是利用 Qsys_system.sopcinfo 文件，这两种方法都可以，我们这里就以第一种方法为例，单击【OK】，将会出现如图 2.67 所示界面。

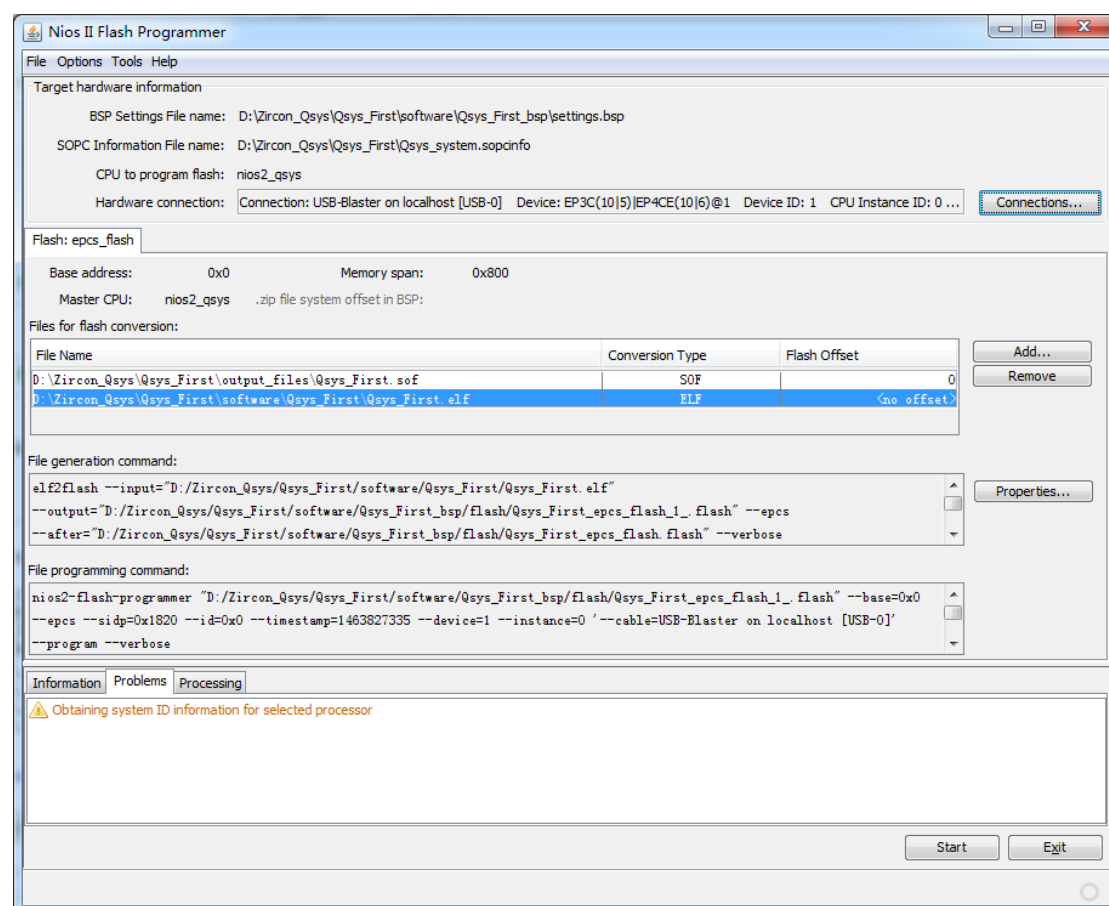


图 2.67 添加 Qsys_First.sof 文件和 Qsys_First.elf 文件

我们单击【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。如果你的提示窗口中有红色的提示信息或者是你的 Start 是灰色的不能够点击，那么有可能是你没有将 Qsys_First.sof 文件下载至 A4 开发板，只有当我们的 A4 开发板中的硬件信息与我们软件工程相匹配，我们才可以点击【Start】烧写程序，点击【Start】以后，我们稍等片刻，程序便下载到 EPCS 中，弹出如图 2.68 所示页面，

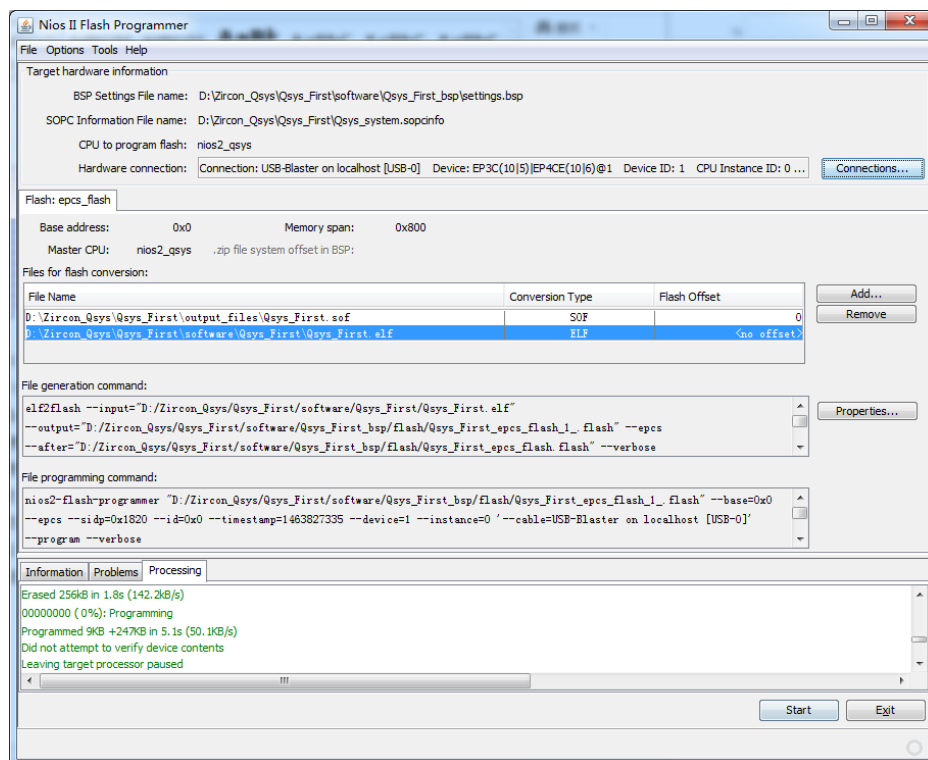


图 2.68 程序下载完成图

下载完毕后，我们需要将 A4 开发板电源关闭，然后再将 A4 开发板电源打开，这时我们可以看到我们的 A4 开发板实现了 Qsys 系统上电自启动，如图 2.69 所示。

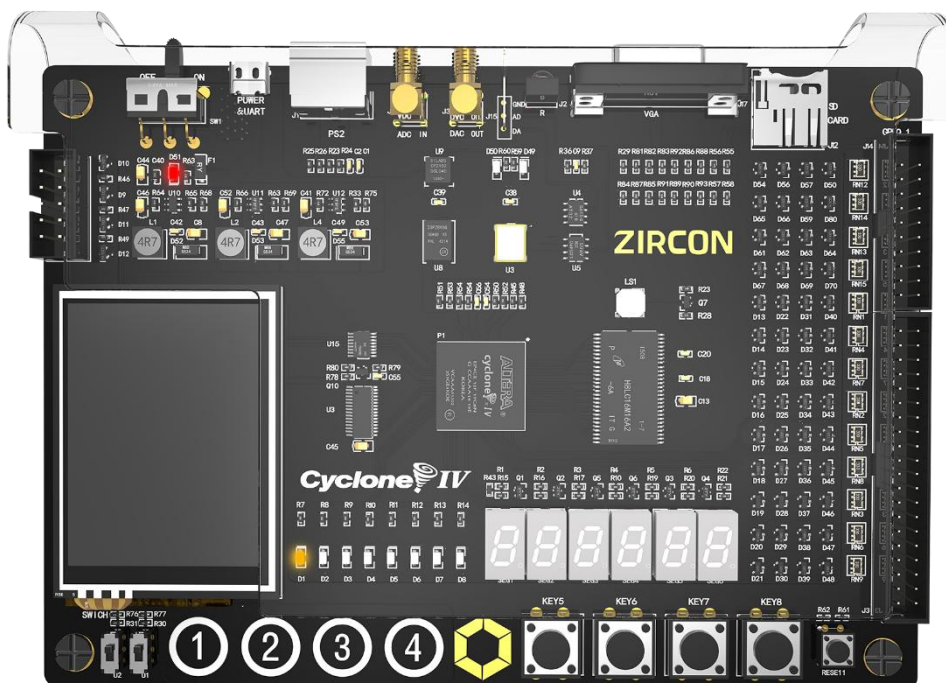


图 2.69 A4 开发板上电自启动

第三章

剥去神秘的外衣——Qsys 到底是如何运行的

第三章

剥去神秘的外衣——Qsys 到底是如何运行的

通过上一章建立你的第一个 Qsys 系统的学习，我想各位一定对 Qsys 的恐惧程度打消了大半了吧！是的，Qsys 就是这么简单易用。你有时只需要轻点几下鼠标，简单敲了几行的代码，就能够完成了极其庞大的工程。是不是感觉有点像把大象装进冰箱的感觉？①打开冰箱门，②把大象放进去，③关门大吉，简单利落！且慢且慢，这可不是一个优秀的嵌入式工程师的风格啊！下面引用一段某国内知名企业对工程师的招聘要求：

- (1) 对待工作认真负责，一丝不苟，求实求精、勤奋进取，积极主动，追求上进；
- (2) 对待困难乐于钻研，不畏枯燥，在置疑中追寻，耐得住寂寞；
- (3) 对知识有强烈的求知欲和责任感，遇到问题不但要知其然而且要究其所以然。

看到了没有，第三条你做到没？如果你不想要知道我们 Qsys 系统，点了几下鼠标、敲了几行代码，它内部到底是如何运行的话，那么你根本就没有做到，遇到问题不但要知其然而且要究其所以然。所以，我们想要成为一个优秀的工程师，我们还是要踏踏实实的深入了解一下我们的 Qsys 系统到底是如何运行的。本章里，我们就要带领大家剥去 Qsys 的神秘外衣，告诉大家它究竟是如何运行的；我们轻点几下鼠标，简单敲了几行的代码后，Qsys 究竟帮我们做了哪些工作。虽然这部分会有些枯燥，但是请务必坚持下去，原因请参考上面的第二条。

§ 3.1 Nios II 硬件框架结构的深入剖析

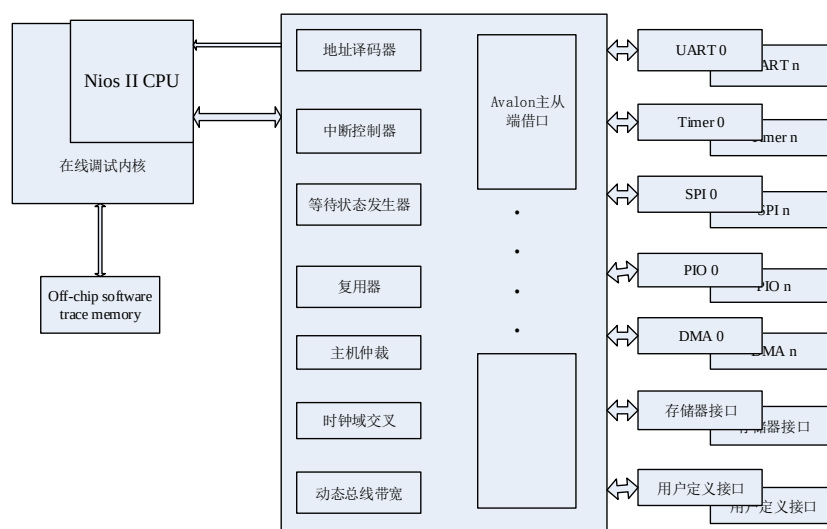


图 3.1 Qsys 系统结构框图

图 3.1 给出了一个 Qsys 系统的结构框图。由 Qsys 系统的结构框图我们可以看出，Qsys 系统是指包含 Nios II 软核处理器和 UART、SPI、定制外设等外设集合的系统。其中 Nios II 是软核处理器，可以借助开发工具对其进行配置，然后下载到可编程芯片中。其优势就是可以根据设计需要选择处理器和外设，并进行灵活配置。图中间的 Avalon 互连模块将系统互连。Nios II 提供 JTAG 接口以方便用户下载和调试。Qsys 系统还可以通过外部存储器接口和 FPGA 片外的

处理器通信，共享外部处理器。它还支持与高性能的 Avalon ST 设备的互连。除了定制模块需要客户的开发，其余部分全部由 Qsys 系统集成工具来生成。本节将向大家介绍 Qsys 硬件系统的架构等相关知识。

Nios II 处理器是整个 Qsys 硬件系统的核心，所以我们先对 Nios II 处理器的结构作深入剖析，为了让大家建立一个完整的 Nios II 处理器的概念，了解一些 Nios II 处理器的工作细节，这对进一步深入学习 Qsys 的开发工作是非常有帮助的。下面我们给出 Nios II 处理器的框架结构图，如图 3.2 所示。

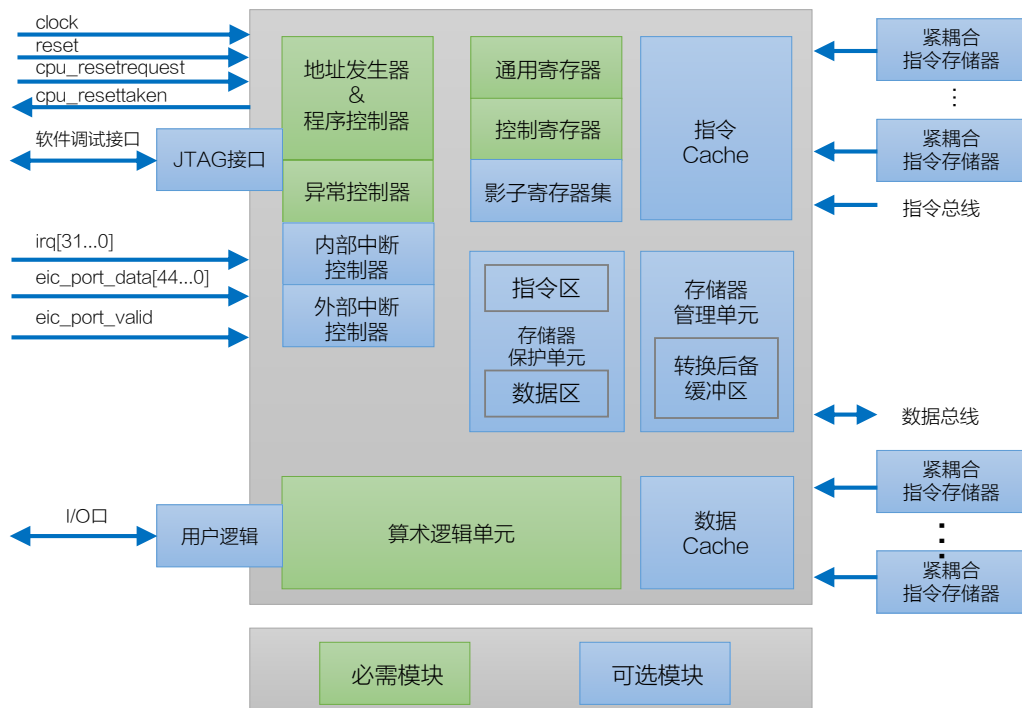


图 3.2 Nios II 处理器的框架结构图

由 Nios II 处理器的框架结构图我们可以看出，Nios II 处理器主要包含以下六个用户可见的功能模块：1、寄存器文件；2、算术逻辑单元；3、复位信号；4、异常和中断；5、存储器和 I/O 结构；6、JTAG 调试模块。接下来我们就对上面的这六个功能模块分别进行介绍。

3.1.1 寄存器文件

首先我们介绍的是寄存器文件，寄存器文件是 Nios II 处理器内部用来存放数据的一些小型存储区域，这些小型存储区域可以用来暂时存放参与运算的数据和运算结果。Nios II 寄存器文件包括 32 个通用寄存器、32 个控制寄存器，以及影子寄存器组。

(1) 通用寄存器

所谓通用寄存器，就是它可以用于多种用途，不是限定好的，它们可用来暂存指令、数据和地址。Nios II 的 32 个通用寄存器命名和用法如表 3.1 所示。

表 3.1 Nios II 的 32 个通用寄存器命名和用法

通用寄存器组					
寄存器	助记符	功能	寄存器	助记符	功能
r0	zero	0x00000000	r16		被调用保存寄存器
r1	at	暂存寄存器	r17		被调用保存寄存器
r2		返回值	r18		被调用保存寄存器
r3		返回值	r19		被调用保存寄存器
r4		参数寄存器	r20		被调用保存寄存器
r5		参数寄存器	r21		被调用保存寄存器
r6		参数寄存器	r22		被调用保存寄存器
r7		参数寄存器	r23		被调用保存寄存器
r8		调用保存寄存器	r24	et	异常暂存寄存器
r9		调用保存寄存器	r25	bt	断点暂存寄存器
r10		调用保存寄存器	r26	gp	全局指针
r11		调用保存寄存器	r27	sp	堆栈指针
r12		调用保存寄存器	r28	fp	帧指针
r13		调用保存寄存器	r29	ea	异常返回地址
r14		调用保存寄存器	r30	ba	断点返回地址
r15		调用保存寄存器	r31	ra	函数返回地址

(2) 控制寄存器

所谓控制寄存器，就是它可以用于控制和确定 Nios II 处理器的操作模式，以及当前执行任务的特性。Nios II 的 32 个控制寄存器命名和用法如表 3.2 所示。

表 3.2 Nios II 的 32 个控制寄存器命名和用法

控制寄存器组					
寄存器	名字	功能	寄存器	名字	功能
ctl0	status	状态寄存器	ctl16	Reserved	Reserved
ctl1	estatus	异常状态寄存器	ctl17	Reserved	Reserved
ctl2	bstatus	断点状态寄存器	ctl18	Reserved	Reserved
ctl3	ienable	中断允许寄存器	ctl19	Reserved	Reserved
ctl4	ipending	中断发生寄存器	ctl20	Reserved	Reserved
ctl5	cpuid	处理器标识寄存器	ctl21	Reserved	Reserved
ctl6	Reserved	Reserved	ctl22	Reserved	Reserved
ctl7	exception	异常寄存器	ctl23	Reserved	Reserved
ctl8	pteaddr	页表寄存器	ctl24	Reserved	Reserved
ctl9	tlbacc	TLB 访问寄存器	ctl25	Reserved	Reserved
ctl10	tlbmisc	TLB 余项寄存器	ctl26	Reserved	Reserved
ctl11	eccinj	ECC 注入寄存器	ctl27	Reserved	Reserved
ctl12	badaddr	异常寄存器	ctl28	Reserved	Reserved
ctl13	config	配置寄存器	ctl29	Reserved	Reserved
ctl14	mpubase	MPU 基地址寄存器	ctl30	Reserved	Reserved
ctl15	mpuacc	MPU 入口寄存器	ctl31	Reserved	Reserved

通过该表我们可以看出，虽然控制寄存器它有 32 个，但是我们只用了 15 个，其余的控制寄存器都用作了保留。在这 15 个寄存器中，其中 ctl0~ctl5 这 6 个寄存器我们是比较常用的，ctl7~ctl15 这 9 个寄存器我们一般是用不到的，它们主要是用于 MMU 或者 MPU 中。下面我们就主要介绍一下 ctl0~ctl5 这 6 个寄存器。首先我们介绍的是 ctl0 (status 状态寄存器)，status 寄存器如图 3.3 所示。

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
Reserved								RSIE	NMI	PRS				CRS				IL				IH	EH	U	PIE						

图 3.3 status 状态寄存器

从该图中我们可以看出，status 的第 0 位 PIE 是外设中断允许位：1 表示允许外设中断；0 表示禁止外设中断。第 1 位 U 是反应计算机当前状态：1 表示处于用户模式；0 表示处于管理员模式。第 2 位 EH 是异常处理程序模式位，如果系统中没有 MMU，EH 将始终为 0。第 3 位 IH 是中断处理模式位，当 IH 为 1 时，表示 Nios II 处理器需要一个外部中断。第 4 位 IL 是中断级字段。第 5 位 CRS 是当前执行的寄存器组；第 6 位 PRS 是之前执行的寄存器组。第 7 位 NMI 是不可屏蔽中断模式位。第 8 位 RSIE 是寄存器设置中断允许位。

简单的介绍完了 status，接下来我们就来介绍 estatus 异常状态寄存器 (ctl1)，在异常处理过程中，异常状态寄存器保存状态寄存器的备份。这个寄存器仅定义了 EPIE，EPIE 位是状态寄存器中 PIE 位的备份。异常管理器可以检查异常状态寄存器，确定异常处理之前处理的状态。在中断返回时，异常返回指令使处理器把异常状态寄存器复制到状态寄存器，恢复到异常处理之前的状态。

在调试断点处理过程中，断点状态寄存器 bstatus (ctl2) 保存状态寄存器的备份。这个寄存器仅定义了 BPIE，BPIE 是 PIE 的备份。当系统产生一个断点时，状态寄存器的值被复制到断点状态寄存器。用断点状态寄存器可以把状态寄存器恢复到处理断点之前的值。

中断允许寄存器 ienable (ctl3) 控制外部硬件中断，中断允许寄存器的每一位对应着一个硬件中断输入 IRQ0~IRQ31 中的一个中断源。如果对应位为 1，则相应的中断被允许；如果对应位为 0，则相应的中断被禁止。

中断发生寄存器 ipending (ctl4) 表示还没有被处理器接受的中断源，如果中断发生寄存器的某位为 1，则表示处理器已经接收 IRQn 的中断请求，并且相应的中断允许位被使能。这时一个只读寄存器，写这个寄存器没有定义。

处理器标识寄存器 cpuid (ctl5) 保存一个静态数值，在多处理器系统中，用这个数值作为处理器的唯一标识。处理器标识寄存器的值是在处理器产生时确定的。写这个寄存器不会影响处理器标识值。

(3) 影子寄存器组

所谓影子寄存器组，其实就是用来备份通用寄存器和状态寄存器的。影子寄存器组通常会和外部中断控制器联合使用，关于影子寄存器组的使用我们将会在外中断控制器中进行讲解。

3.1.2 算术逻辑单元

算术逻辑单元 (ALU), 对于 ALU 我相信大家都是有所耳闻的, 它的主要功能就是对存储在通用寄存器中的数据进行操作, ALU 从寄存器中取一个或者两个操作数, 经过计算后并将运算结果存回寄存器中。Nios II 的处理器中的 ALU 支持以下四种运算:

- (1) 算术运算: 加法、减法、乘法和除法;
- (2) 关系运算: 有符号和无符号的等于、不等于、大于等于和小于运算;
- (3) 逻辑运算: 与、或、非和异或;
- (4) 移位运算: 逻辑右移、逻辑左移、算术右移、算术左移、循环右移和循环左移。

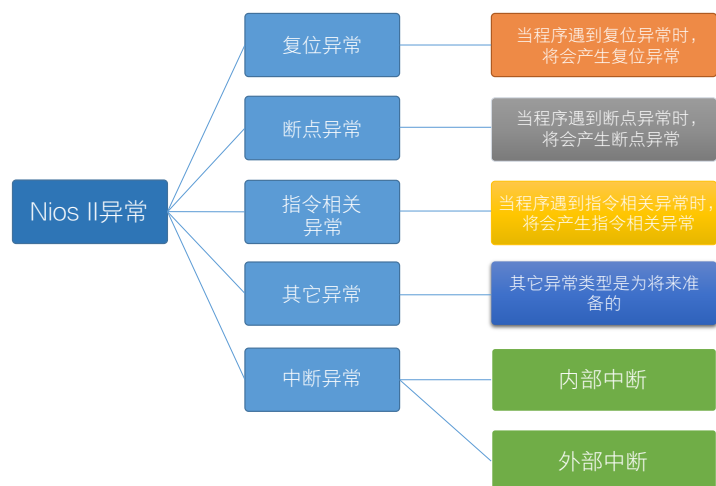
我们知道, 利用 ALU 实现乘法、除法和可控位数的移位等操作是非常复杂的, 需要很多逻辑资源, 因此, 在 Nios II 设置中, 还可以选择硬件算法操作, 通过嵌入式乘法器和嵌入式除法器来实现。

3.1.3 复位信号

Nios II 处理器支持两个复位信号, 其中一个复位信号是全局硬件复位信号 `reset`, 这个复位信号是一个外部输入信号, 高电平有效, 用这个信号可以强制处理器核立即进入复位。另一个复位信号是 `cpu_resetrequest`, 这个复位信号是一个本地复位信号, 也是高电平有效, 它可以只让 CPU 复位, 而 Nios II 系统中的其他元件不受这个复位信号影响。当 Nios II 处理器检查到这个复位信号有效后, 将执行完流水线中的全部指令, 然后进入复位状态, 这个复位过程可能需要数个时钟周期才能完成。当 Nios II 处理器完成复位后, Nios II 处理器将输出 `cpu_resettaken` 信号, 表明复位完成。这时 Nios II 处理器将保持复位状态到复位信号 `cpu_resetrequest` 无效为止。这里要注意的是, 当 Nios II 处理器处于 JTAG 调试模式控制下, Nios II 处理器是不会响应复位信号 `cpu_resetrequest` 的。

3.1.4 异常和中断

异常和中断我想大家都不会陌生。我们举个简单的例子, 当我们在敲击计算机键盘空格的时候, 计算机为什么会在文档中添加一个空格呢? 这是因为我们在敲击键盘的一刹那, 我们键盘的硬件控制器会发送一个中断给我们计算机的 CPU, 我们计算机的 CPU 会响应这个中断, 它先停下当前的工作, 转而去调用中断服务程序(Interrupt Service Routine)来执行我们的这个空格操作。如果你学习过单片机或者是微机原理, 那么你对这个中断过程会非常熟悉。在 Nios II 中也有类似的中断机制, 我们将这种类似的机制统称为异常。异常通常分为以下五种类型, 中断只是其中的一类, 如图 3.4 所示。



从该图中我们可以看出，Nios II 异常主要包括复位异常、断点异常、指令相关异常、其它异常和中断异常。在这五种类型异常中，最重要的异常就属我们的中断异常了，下面我们就来重点介绍中断异常，在 Nios II 中，中断异常又分为内部中断和外部中断。首先我们介绍的是内部中断，Nios II 体系结构支持 32 个内部中断，即 irq0~irq31，IRQ 的优先级由 Qsys 软件决定，如图 3.5 所示。

Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	I
☒		clk	Clock Source						
		clk_in	Clock Input	clk	exported				
		clk_in_reset	Reset Input	reset					
		clk	Clock Output	clk					
		clk_reset	Reset Output	reset					
☒		mies2_qsys	Nios II Processor						
		clk	Clock Input	clk					
		reset_n	Reset Input	reset					
		data_master	Avalon Memory Mapped Master						
		instruction_master	Avalon Memory Mapped Master						
		jtag_debug_module...	Reset Output	reset					
		jtag_debug_module...	Avalon Memory Mapped Slave						
		custom_instructi...	Custom Instruction Master						
☒		onchip_rom	On-Chip Memory (RAM or ROM)						
		clk	Clock Input	clk					
		sl	Avalon Memory Mapped Slave						
		reset	Reset Input	reset					
☒		onchip_ram	On-Chip Memory (RAM or ROM)						
		clk	Clock Input	clk					
		sl	Avalon Memory Mapped Slave						
		reset	Reset Input	reset					
☒		jtag_uart	JTAG UART						
		clk	Clock Input	clk					
		reset	Reset Input	reset					
		avalon_jtag_slave	Avalon Memory Mapped Slave						
☒		sysid_qsys	System ID Peripheral						
		clk	Clock Input	clk					
		reset	Reset Input	reset					
		control_slave	Avalon Memory Mapped Slave						
☒		pio_led	PIO (Parallel I/O)						
		clk	Clock Input	clk					
		reset	Reset Input	reset					
		sl	Avalon Memory Mapped Slave						
☒		external_connection	Conduit						
		clk	Clock Input	clk					
☒		pio_key	PIO (Parallel I/O)						
		clk	Clock Input	clk					
		reset	Reset Input	reset					
		sl	Avalon Memory Mapped Slave						
		external_connection	Conduit						

图 3.5 在 Qsys 软件中的内部中断

从该图右侧部分我们可以看出，我们连接了 2 个内部中断，一个是 jtag_uart 的 0 中断，一个是 pio_key 的 1 中断，0 中断的优先级要高于 1 中断，每个中断对应一个独立的中断通道。针对每个 IRQ 输入，处理器中的 ienable 中断允许寄存器中都有一个相应的中断使能位，处理器能够通过 ienable 控制器来独立地使能或者禁止每个中断源。处理器也可以通过 status 控制寄存器的 PIE 为来使能或者禁止所有的中断，如图 3.6 所示。

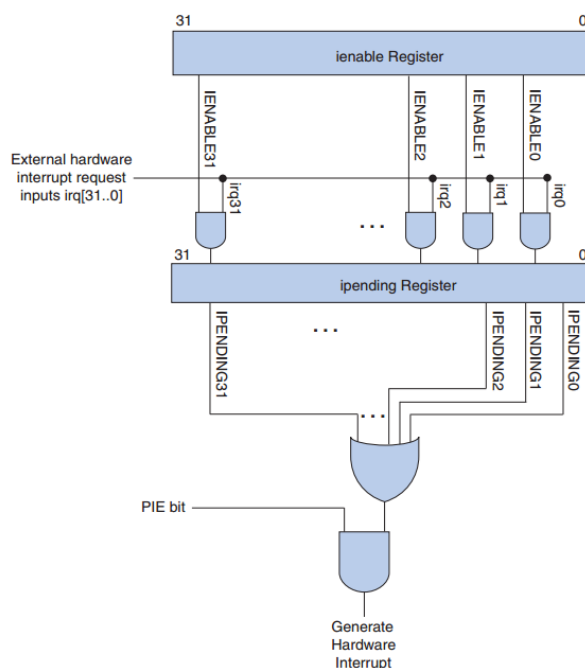


图 3.6 内部中断的产生图

通过该图我们可以看出，只有具备以下条件，系统才会产生内部中断：

- (1) Status 控制寄存器中的 PIE 位为 1；
- (2) 某个中断请求 irqn 有效；
- (3) 在 ienable 寄存器中，该中断源相应位为 1；

知道了如何产生内部中断以后，下面我们再来看下内部中断的处理流程，当内部中断发生后，Nios II 处理器会依次完成以下工作：

- (1) 把 status 寄存器内容复制到 estatus 寄存器中，保存当前处理器状态，
- (2) 清除 status 寄存器的 PIE 位为 0，禁止所有的中断；
- (3) 把异常返回地址写入 ea 寄存器 (r29)；
- (4) 跳转到异常处理地址。

讲完了内部中断，接下来我们再来看下外部中断。外部中断相对于内部中断来说比较抽象，这是因为我们在之前的工程中只对内部中断进行了配置和使用，并没有对外部中断进行操作说明。下面我们来看下内部中断和外部中断究竟有着怎样的区别与不同。首先我们先来看下外部中断是如何配置的，如图 3.7 所示。

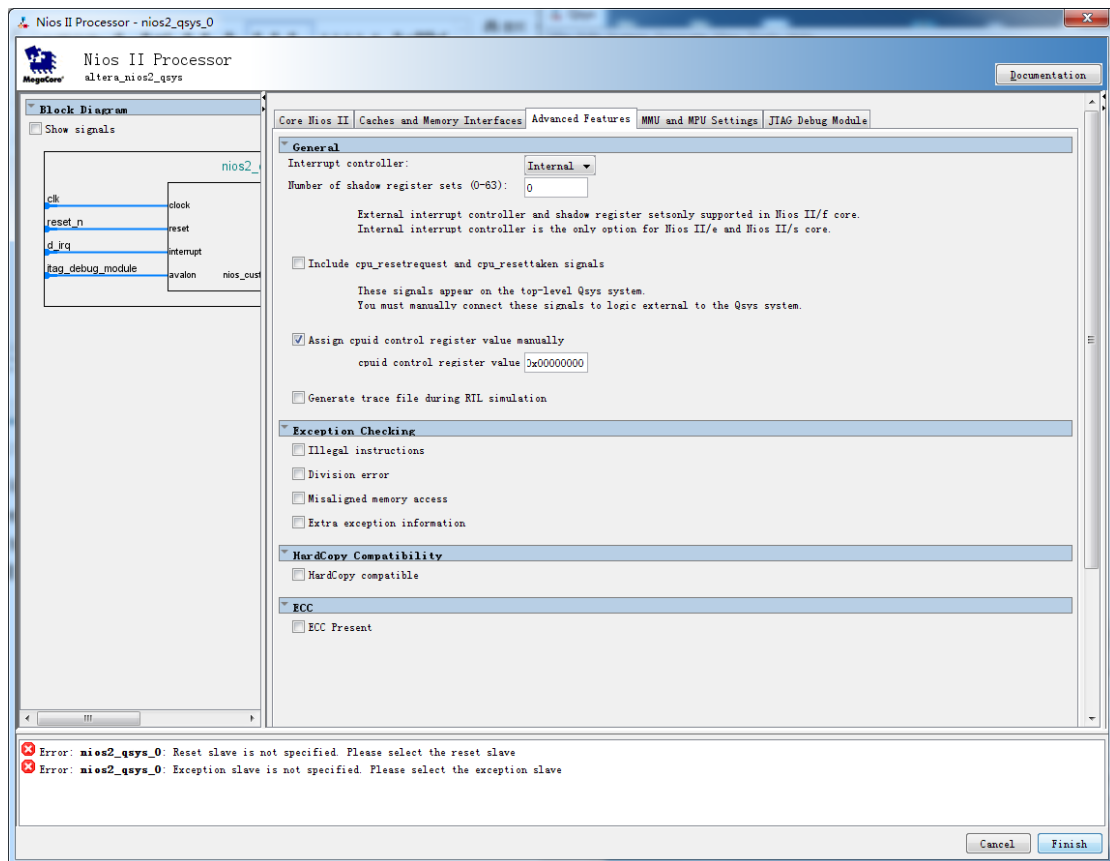


图 3.7 外部中断的配置图

从该图中我们可以看出，我们可以在【 Advanced Features 】标签下的【 Interrupt Controller 】中进行设置。如果我们想要使用内部中断，那么我们只需要选择【 Internal 】选项就可以了。如果我们想要使用外部中断，那么我们就需要选择【 External 】选项。这里需要我們注意的是，如果我们选择了【 External 】选项，那么我们还需要设置相对应的影子寄存器组。也就是说，我们针对每一个外部中断都会给它分配一个相对应的影子寄存器，用来备份通用寄存器和状态寄存器的。一旦外部中断来了之后，我们的 CPU 就会切换到影子寄存器组中去运行。运行完后，我们的 CPU 就会回到之前的寄存器中继续运行，因此能够完成保护现场的工作。最后我们再来补充说明一下，对于 Nios II/e 和 Nios II/s 这两个等级中，它是不支持外部中断的。只有 Nios II/f 等级中，它才会支持外部中断。当我们选择使用外部中断控制器以后，会出现如图 3.8 所示页面。

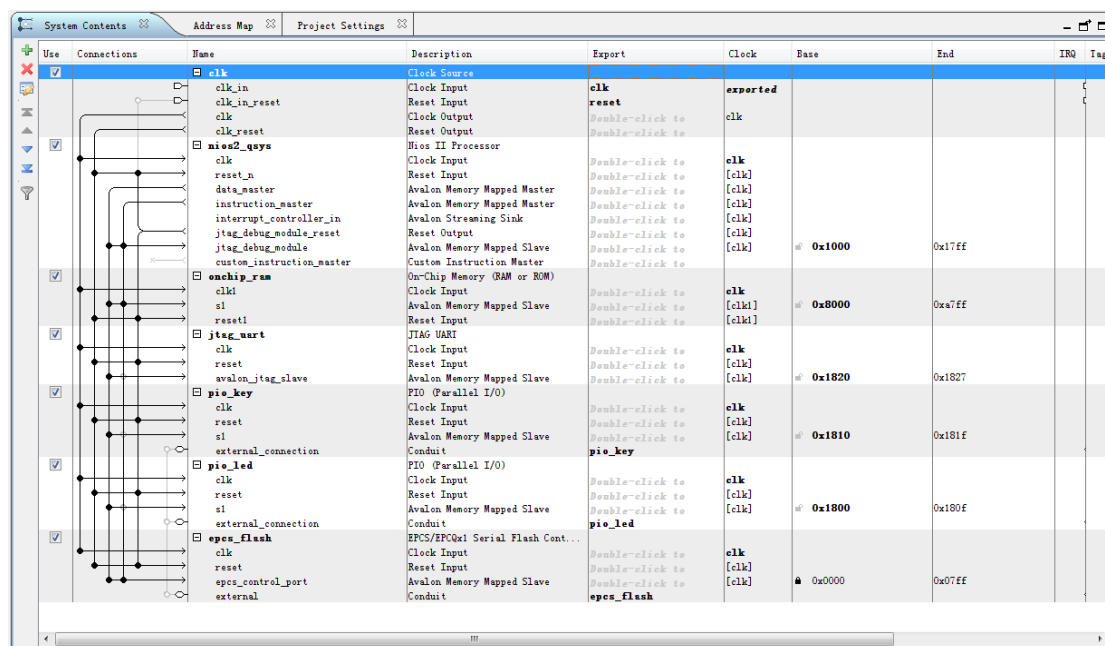


图 3.8 在 Qsys 软件中的外部中断

在该图中我们可以看到，我们的内部中断连接已经全部消失，同时 nios2_qsys 多出了一个端口，该端口叫做 interrupt_controller_in，它就是用来连接外部中断的。当然，只使用这一个端口我们是实现不了外部中断的，我们还需要添加一个 Vectored Interrupt Controller IP 核，该 IP 核的配置如图 3.9 所示。

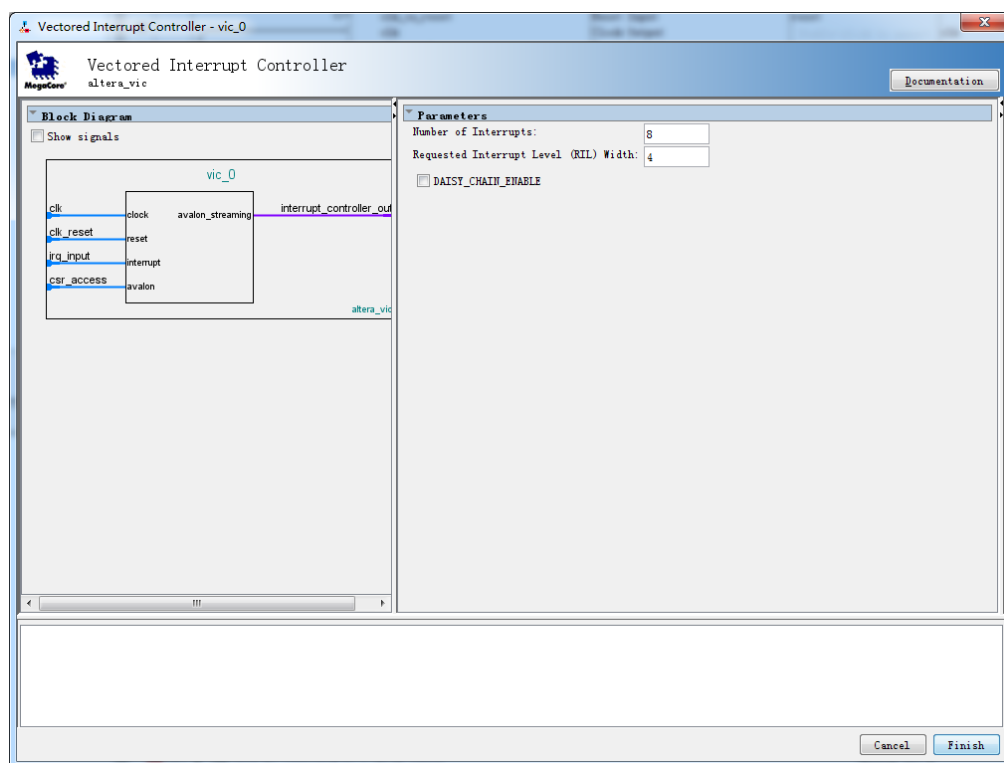


图 3.9 Vectored Interrupt Controller 配置页面

从该页面中我们可以看到，该 IP 核的配置很简单，只有三个选项，首先第一个选项是用来

配置外部中断的个数，第二个选项是用来配置每个中断的中断级别的位数。第三个选项就是菊花链，可以用来连接多个 Vectored Interrupt Controller IP 核，支持更多的外部中断。我们将该 IP 核添加至 Qsys 软件中，如图 3.10 所示。



图 3.10 外部中断在 Qsys 软件中的连接

从该图中我们可以看出，首先我们将 Vectored Interrupt Controller IP 核连接到了我们的 nios2_qsys 上，然后我们就可以在该图的右侧像连接内部中断一样去连接外部中断。至此我们的外部中断就配置完成了。也许有的朋友会有这么一个疑问，我们的内部中断配置起来那么简单，我们的外部中断配置起来那么复杂，为什么我们不用内部中断，而去选择用外部中断呢？这主要有以下四点原因：

- (1) 你需要一个或多个中断来减少平均响应时间；
- (2) 你对中断性能有很强的实时性要求；
- (3) 你需要不可屏蔽中断；
- (4) 你需要处理超过 32 个中断。

通过以上四点，我们相信大家都能够理解，对于一般的中断我们都可以去使用内部中断去处理，但是对于一些内部中断没有办法处理的中断，我们必须使用外部中断去处理。也许有的朋友还会有这么一个疑问，外部中断为什么要比内部中断要快呢？这主要是因为外部中断用到了影子寄存器组。我们在之前寄存器文件讲解中，我们只是简单的介绍了一下影子寄存器组，并没有进一步去讲解它，下面我们就结合外部中断来对影子寄存器组进行一个进一步的介绍。影子寄存器组，说白了它其实就是之前我们所说的通用寄存器组的一个完整拷贝，它是用来代替我们的通用寄存器的。当外部中断来了，影子寄存器组就会为这个中断，制造一个完全属于它自己的一个运行环境，它运行完了以后，再回到原来的通用寄存器中继续运行。我们要知道，程序运行过程中，我们不仅需要通用寄存器，我们还需要知道程序运行的状态等等，因此，影子寄存器组它

还包括一个控制寄存器中 cti0 寄存器的拷贝,我们称这个寄存器为 sstatus,其中 s 就是 shadow, status 就是我们的状态寄存器的意思,如图 3.11 所示。

Table 3-33. sstatus Control Register Fields

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
SRS	Reserved							RSIE	NMI	PRS				CRS				IL				IH		EH		U	PIE				

图 3.11 sstatus 寄存器

从该图中我们可以看到,这个 sstatus 和我们之前的状态寄存器几乎是一样,唯一不同的就是第 31 位的 SRS,这个 SRS 是 Switched register set bit 的缩写,意思是切换寄存器比特位。当 SRS 为 1 的时候,就表示我们的寄存器要开始切换到另一组影子寄存器组中了。介绍完了影子寄存器组,下面我们来看下外部中断如何与我们的影子寄存器组配合来处理中断流程的,如图 3.12 所示。

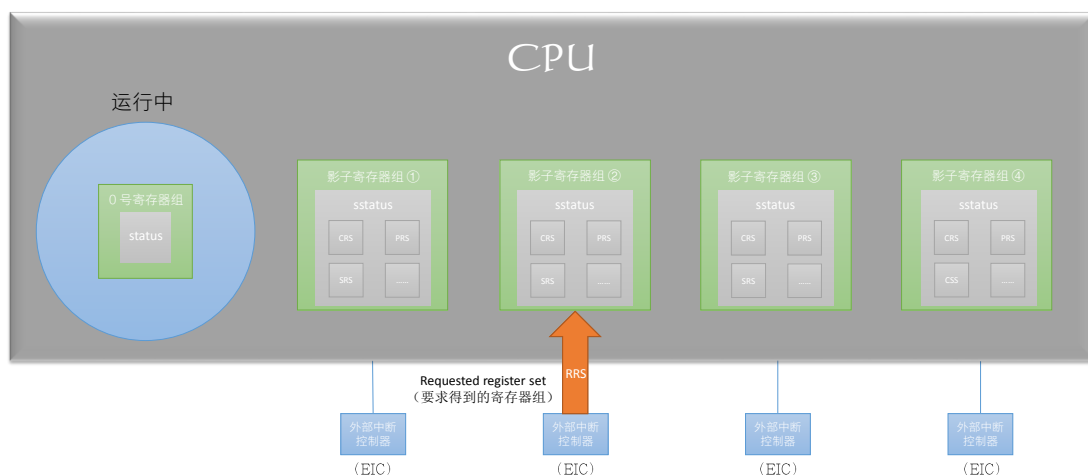


图 3.12 外部中断处理过程图

从该图中我们可以看出,我们这里有四个外部中断,每个外部中断对应一个影子寄存器组,此时,我们的程序正运行在 0 号寄存器组中,也就是运行在我们 CPU 本身正常工作模式下的寄存器组中,在我们的 0 号寄存器组中我们还可以看到,它包含着一个 status 状态寄存器。由于我们影子寄存器组是对通用寄存器和状态寄存器的一个拷贝,所以我们影子寄存器中也包含有通用寄存器和 sstatus 状态寄存器。下面我们就来分析一下外部中断产生后,这几个寄存器是如何变化的。比如说,我们的 2 号中断来了,我们的外部中断控制器会产生一个 RRS (Requested register set, 即要求得到的寄存器组) 信号告诉我们的 CPU, 2 号中断来了,此时我们的 CPU 就会将 RRS 与我们的 CRS (Current register set field, 即当前执行的寄存器组) 做对比,看是否吻合,如果不吻合,那么 sstatus 中的 SRS (Switched register set bit, 即切换寄存器比特位) 就会被打开。SRS 被打开以后,我们的 CPU 就会将 CRS 中的值写入到 PRS (Previous register set field, 即之前执行的寄存器组) 中,同时还会将 RRS 中的 2 号写入到 CRS 中。CRS 一旦被写入以后,那么整个运行环境就会立刻切换到 2 号影子寄存器组中,如图 3.13 所示。

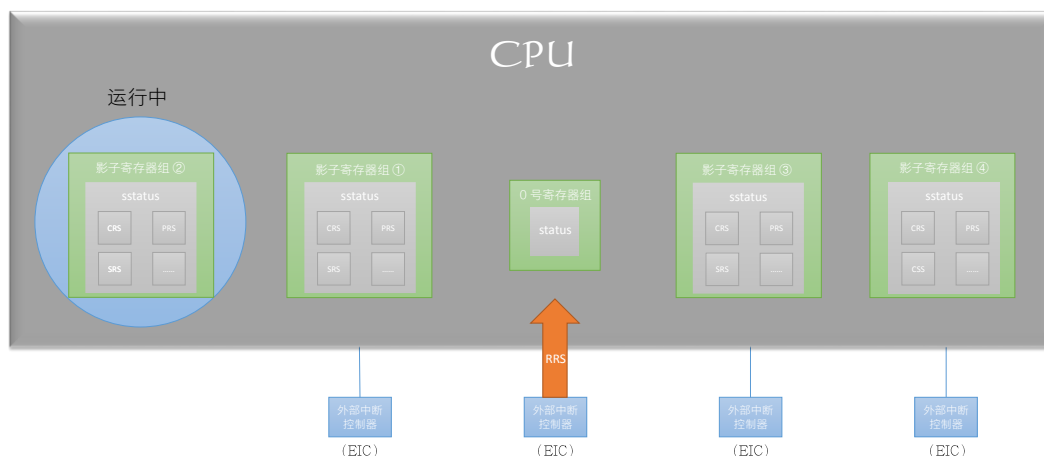


图 3.13 外部中断处理过程图

从该图中我们可以看出，我们的程序刚开始全部是跑在 0 号寄存器组中的，当 2 号外部中断来了以后，我们的程序会立刻运行在 2 号影子寄存器组。这样就会完成一个非常快捷的中断响应过程，从而避免了地址跳转等复杂操作。当 2 号外部中断运行完了以后，我们的程序会回到 0 号寄存器中继续运行。这里我们只是简单的定性的去描述了一下整个过程，如果大家想了解更多关于外部中断和影子寄存器组的知识，那么可以去看下 Altera 提供的官方手册。

3.1.5 存储器和 I/O 结构

Nios II 存储器和 I/O 结构非常灵活，这是 Nios II 处理器系统与传统微控制器之间的最显著的区别。因为 Nios II 处理器系统可配置，存储器和外设随着系统的不同而不同，最终使得存储器和 I/O 结构随着系统不同发生变化。Nios II 内核可使用下面的一种或多种方式访问存储器和 I/O：

- (1) 指令主端口：Avalon 主端口，通过 Avalon 系统互连结构连接到指令存储器；
- (2) 数据主端口：Avalon 主端口，通过 Avalon 系统互连结构连接到数据存储器；
- (3) 指令高速缓存：Nios II 内核里面的高速缓存；
- (4) 数据高速缓存：Nios II 内核里面的高速缓存；
- (5) 紧耦合指令或数据存储器端口：与 Nios II 内核外的快速存储器相连。

说完了 Nios II 内核访问存储器和 I/O 的 5 种方式，下面我们来看下 Nios II 处理器内核的存储器和 I/O 结构框图，如图 3.14 所示。

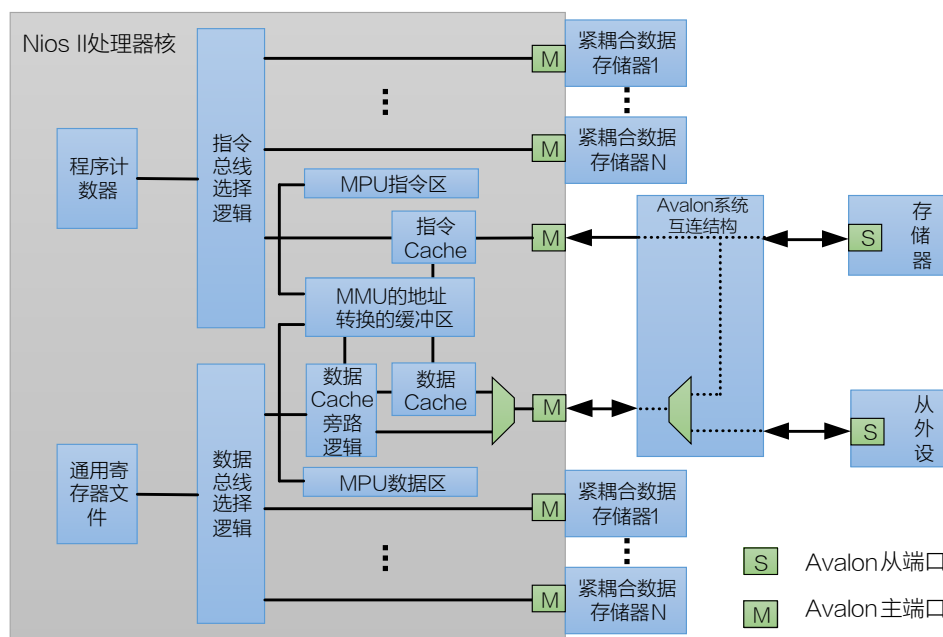


图 3.14 Nios II 处理器内核的存储器和 I/O 结构图

从该图中我们可以看出，Nios II 处理器属于哈佛结构，支持独立的指令和数据总线。指令和数据总线都遵循 Avalon 接口规范。数据主端口连接存储器和外设元件，而指令主端口只连接存储器元件，我们在“连连看”的时候就是遵循的这个规则。由于上面的这个框架图比较复杂，它包含了所有的端口，为了让大家容易理解学习，我们将上面那个复杂的框图分解成了四部分，1、指令主端口和数据主端口；2、指令高速缓存和数据高速缓存；3、MMU 和 MPU；4、紧耦合存储器。下面我们就对这四部分内容分别进行介绍。

首先我们介绍的是第一部分指令主端口和数据主端口，如图 3.15 所示。

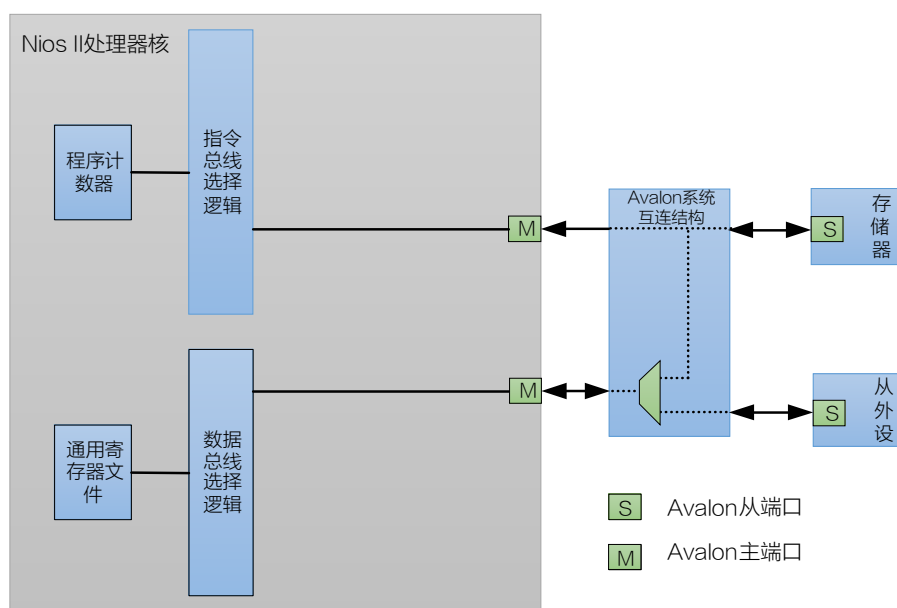


图 3.15 指令主端口和数据主端口的结构图

从该图中我们可以看到，指令总线选择逻辑和数据总线选择逻辑，它们通过 Avalon 系统互

连结构连接到我们外部设备，包括存储器和外设。对于指令主端口，它只执行一个功能，就是对处理器将要执行的指令进行取指，并且它不执行任何写操作。对于数据主端口，它执行两个功能，第一个是当处理器执行装载指令时，从存储器或外设中读数据；第二个是当处理器执行存储指令时，将数据写入存储器或外设。这时候我们再回想一下，想必大家就能明白为什么数据主端口连接存储器和外设元件，而指令主端口只连接存储器元件。这是因为我们的外设元件不进行写操作就无法控制，并且我们的外设元件不包含任何指令，所以我们的外设元件不需要连接指令主端口了，只要连接数据端口就可以了。当然，由于我们的存储器能够为系统提供指令，所以我们的存储器需要连接指令端口和数据端口。

介绍完了第一部分指令主端口和数据主端口，下面我们再介绍第二部分指令高速缓存和数据高速缓存，如图 3.16 所示。

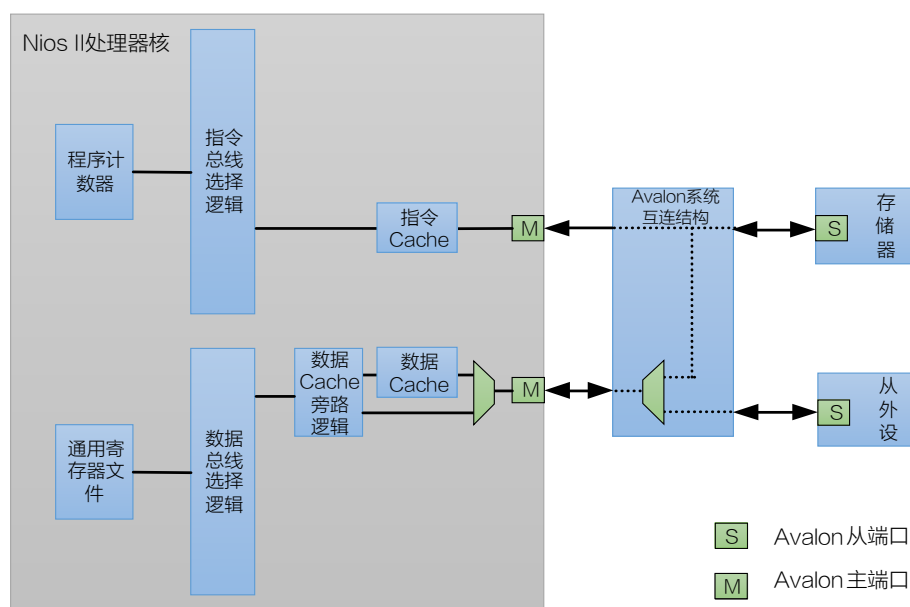


图 3.16 指令高速缓存和数据高速缓存的结构图

从该图中我们可以看出，Nios II 结构的指令主端口和数据主端口都支持高速缓存（Cache）。指令主端口使用指令高速缓存，数据主端口使用数据高速缓存。高速缓存使用片内存储器，是 Nios II 处理器内核的重要组成部分。它能够改善使用较慢片存储器（如用来存放程序和数据的 SDRAM）的 Nios II 处理器系统的平均存储器访问时间。指令 Cache 和数据 Cache 不仅仅加速了存储器的平均访问时间，它在软件上也几乎是透明的，不需要我们写任何代码就可以实现。作为 Nios II 处理器组成部分的告诉缓存在 Qsys 中是可选的，这取决于用户对系统存储性能以及 FPGA 资源的使用要求。Nios II 处理器内核可以含有数据或指令缓存，也可以两者都有或都没有，高速缓存大小可由用户配置，如图 3.17 所示。

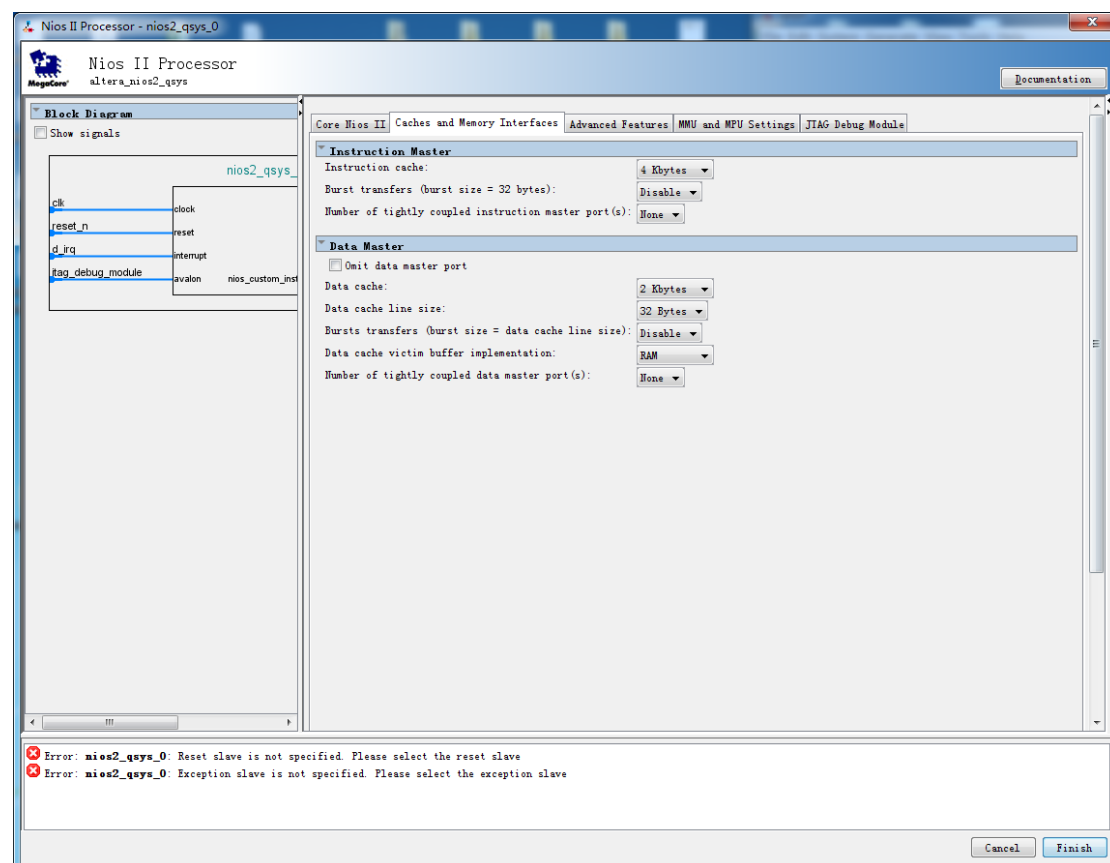


图 3.17 指令高速缓存和数据高速缓存在 Qsys 软件中的配置

这里需要我们注意的是，指令 Cache 和数据 Cache 虽然可以改善系统的整体性能，但是它会使程序的执行时间变得不可预测，对于实时系统来说，代码执行的不确定性——装载和存储指令或数据的时间必须是可预测的，这一点至关重要。

介绍完了第二部分指令高速缓存和数据高速缓存，下面我们再来介绍第三部分 MMU 和 MPU，如图 3.18 所示。

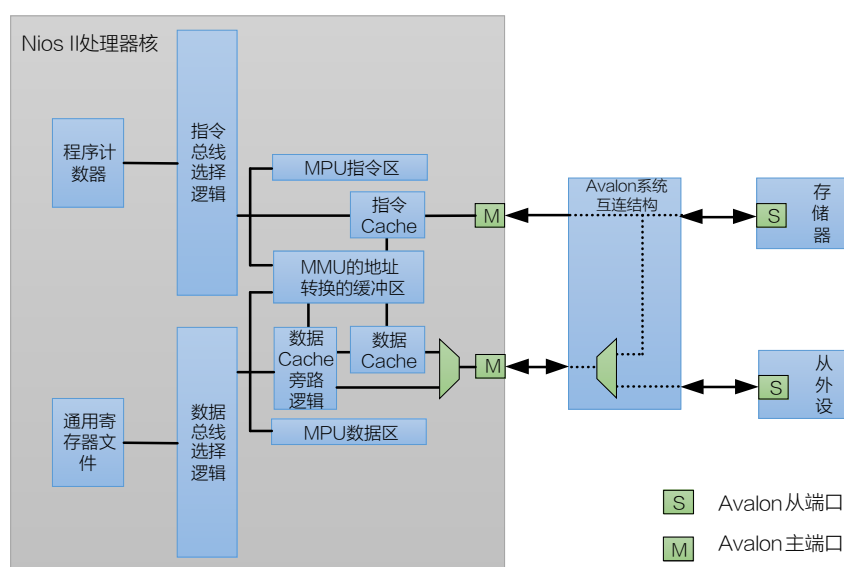


图 3.18 MMU 和 MPU 的结构图

MMU (Memory Management Unit 即, 存储器管理单元), 说到 MMU, 我们不得不说一下虚拟内存这个概念, 虚拟内存是计算机系统内存管理的一种技术。它使得应用程序认为它拥有连续的可用的内存(一个连续完整的地址空间), 而实际上, 它通常是被分隔成多个物理内存碎片, 还有部分暂时存储在外部磁盘存储器上, 在需要时进行数据交换。这里我们举个简单的例子, 比如对一个 16MB 的程序和一个内存只有 4MB 的机器, 操作系统通过选择, 可以决定各个时刻将哪 4M 的内容保留在内存中, 并在需要时在内存和磁盘间交换程序片段, 这样就可以把这个 16M 的程序运行在一个只具有 4M 内存机器上了。实现这种虚拟内存系统需要一种机制, 就是将虚拟地址转换为实际的物理地址。这一机制的实现过程通常是由一个操作系统和特定的硬件联合完成。在 Nios II/f 型处理器中, 可选的 MMU 就是用来完成这一工作的特定硬件。当然, 仅有硬件无法实现这一功能的。我们还需要一个能支持虚拟内存的操作系统, 常见的系统有 Linux 和 Windows CE 等, 常见的不支持虚拟内存的操作系统有 uC/OS、uClinux。

MPU(Memory Protection Unit 即, 存储器保护单元), 现代操作系统包含保护机制, 限制用户应用程序访问关键的系统资源。例如, 一些操作系统将程序的执行分为内核模式(没有限制) 和用户模式(有限制), 这个方案的实现也需要特殊硬件的支持了, 在 Nios II/f 等级的处理器中, 可选的 MPU 就是用来完成这项工作的。MPU 仅提供存储器保护, 它是不支持内存映射和管理的。在这里我们需要注意的是, 由于 Altera HAL 为 MPU 提供了基本的支持, 所以, 没有 OS 也是可以使用的, 在 Nios II 配置中, MMU 和 MPU 的使用是互斥的, 这意味着两者之中仅有一个可用。它们在 Qsys 软件中的配置如图 3.19 所示。

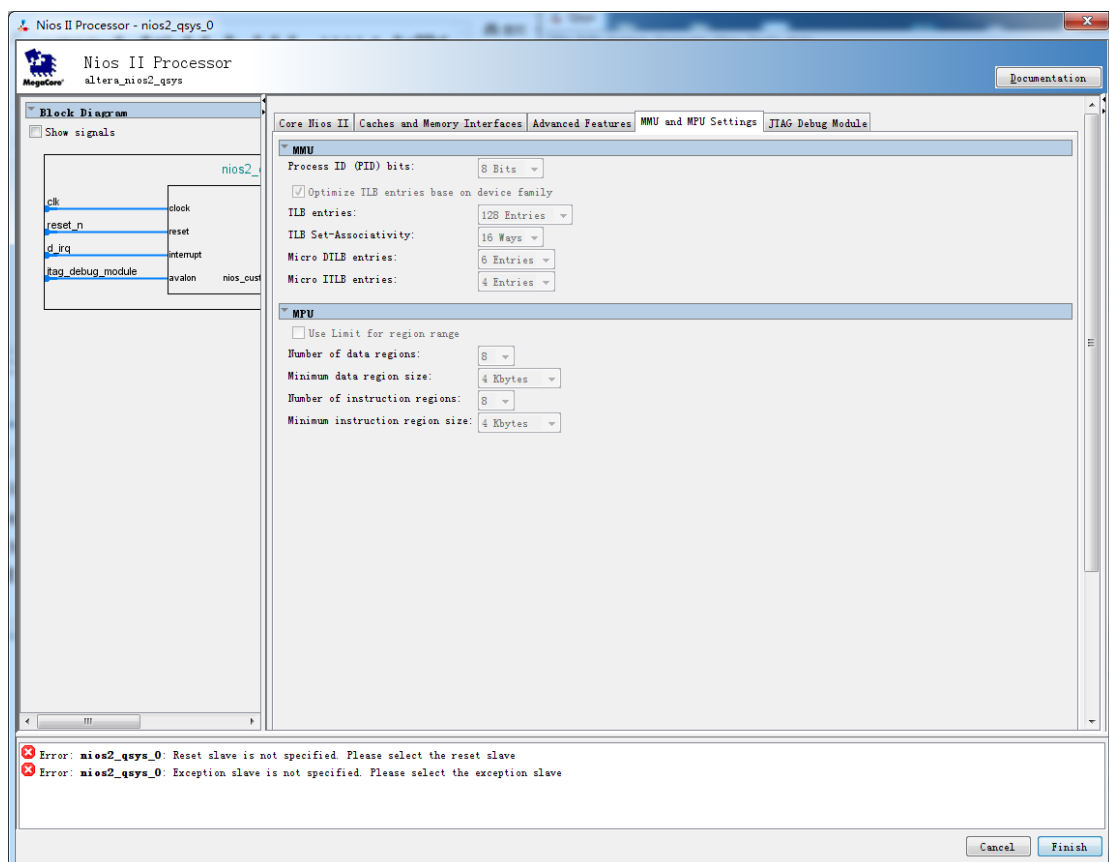


图 3.19 MMU 和 MPU 在 Qsys 软件中的配置

介绍完了第三部分 MMU 和 MPU，下面我们再来介绍最后一部分紧耦合存储器，如图 3.20 所示。

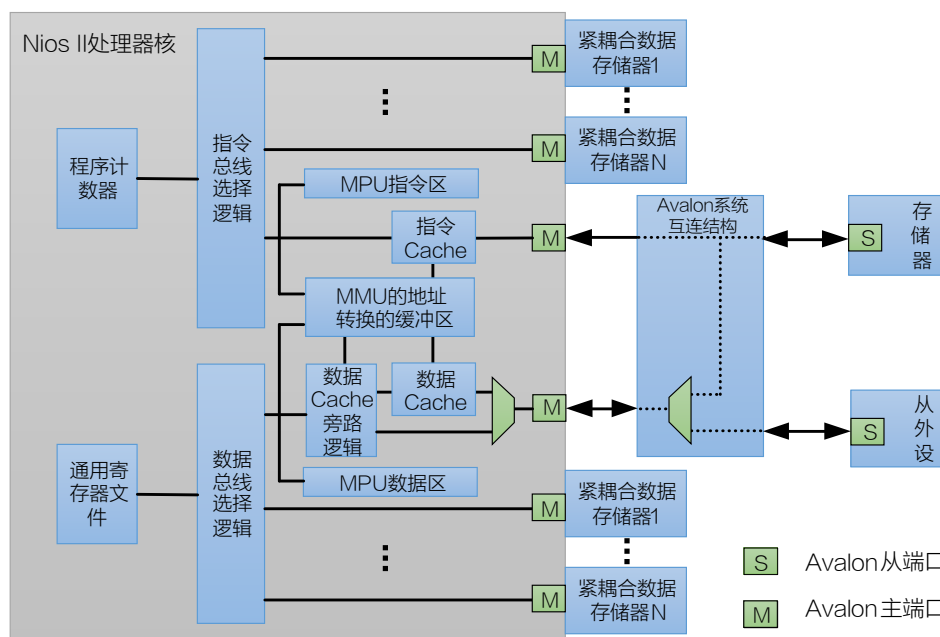


图 3.20 紧耦合存储器的结构图

为了提高系统的整体性能，Nios II 内核不仅可以集成数据 Cache 和指令 Cache，还可带有紧耦合存储器接口。紧耦合存储器是一种紧挨着内核的快速 SRAM，它不仅能改善系统性能，而且保证了装载和存储指令或数据的时间是确定的。紧耦合存储器既能使 Nios II 处理器提高性能，又能获得可预测的实时响应。紧耦合存储器可向对性能要求严格的应用提供低延迟访问。与高速缓存相比，使用紧耦合存储器有以下优点：

- (1) 性能类似于高速缓存；
- (2) 软件能够保证将关键性能的代码或数据存放在紧耦合存储器中；
- (3) 代码执行的确定性——装载和存储指令或数据的时间是可预测的。

实际上，紧耦合存储器是 Nios II 处理器内核上的一个独立的主端口，与指令或数据主端口类似。Nios II 结构指令和数据访问都支持紧耦合存储器。Nios II 内核可以不包含紧耦合存储器，也可以包含一个或多个紧耦合存储器。每个紧耦合存储器端口直接与具有固定的低延迟的存储器相连，该存储器在 Nios II 内核的外部，通常使用 FPGA 片内存储器。紧耦合存储器与其它通过 Avalon 交换结构连接的存储器件一样，占据标准的地址空间。它的地址范围在生成系统时确定。软件使用常规的装载和存储指令访问紧耦合存储器。从软件的角度来看，访问紧耦合存储器与访问其他存储器没什么不同。系统在访问指定的代码或数据时，能够使用紧耦合存储器来获得最高性能。例如，中断频繁的应用能够将异常处理代码放在紧耦合存储器中来降低中断延迟。类似的，计算密集型的数字信号处理（DSP）应用能够将紧耦合存储器指定为数据缓存区，实现最快的数据访问。如果应用程序的存储器需求足够小，能够完全在片内实现，可以使用专门针对代码和数据的紧耦合存储器。如果应用程序较大，那么必须仔细选择放入紧耦合存储器中的内容，使成本和性能能够实现最佳平衡。

讲完了什么是紧耦合存储器，下面我们再来看下如何在 Qsys 软件中配置我们的紧耦合存储器，如图 3.21 所示。

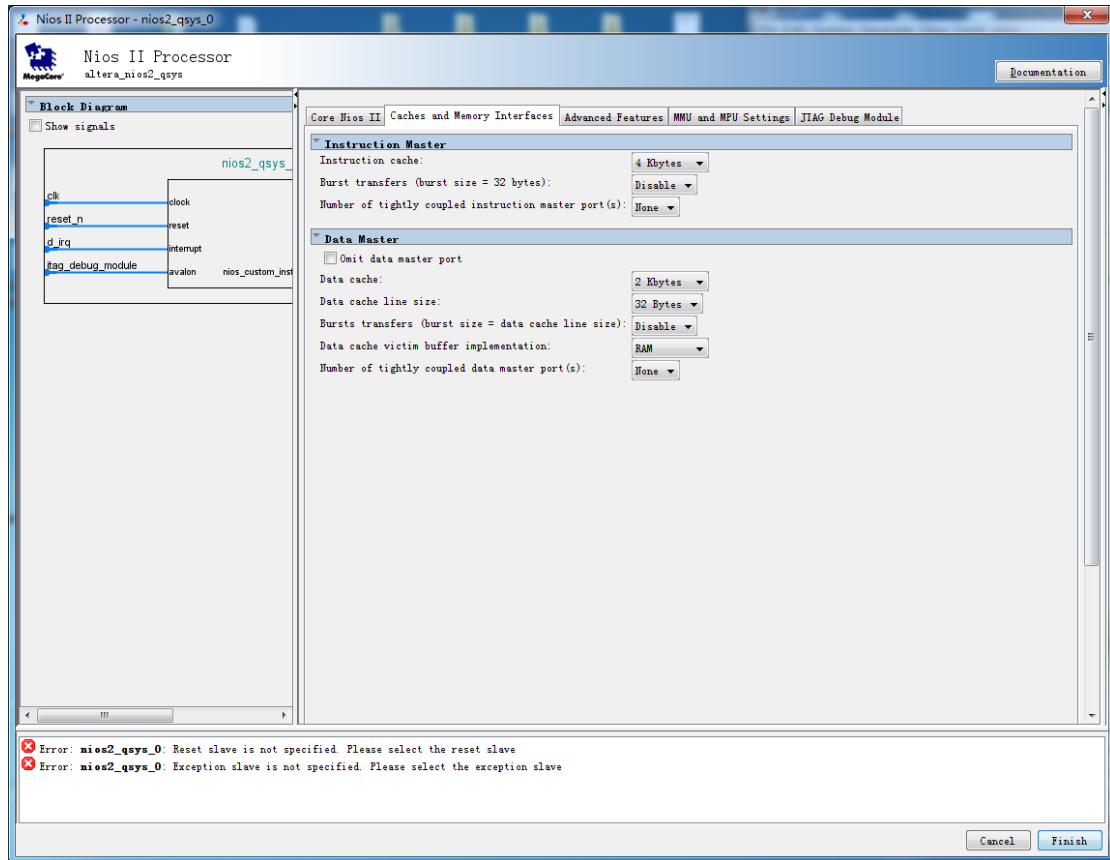


图 3.21 紧耦合存储器在 Qsys 软件中的配置

在该图中我们可以看到，我们可以在【Include tightly coupled instruction master pods】和【Include tightly coupled data master pods】这两个选项中，进行选择是否添加紧耦合存储器。当我们都选择 1 以后，我们的 nios2_qsys 会出现如图 3.22 所示页面。

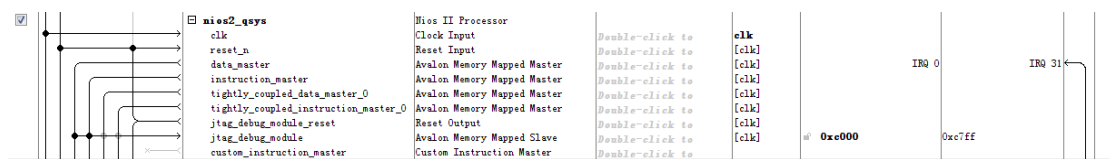


图 3.22 紧耦合存储器在 Qsys 软件中的连接

在该页面中我们可以看到，我们的 nios2_qys 多出了两个端口，一个是 tightly_coupled_data_master_0 接口，一个是 tightly_coupled_instruction_master_0 接口。当然，只使用这两个端口我们是实现不了的，我们还需要添加两个片上存储器，也就是我们的 onchip_memory IP 核，如图 3.23 所示。

Figure 1. Nios II System with Tightly Coupled Instruction and Data Memory

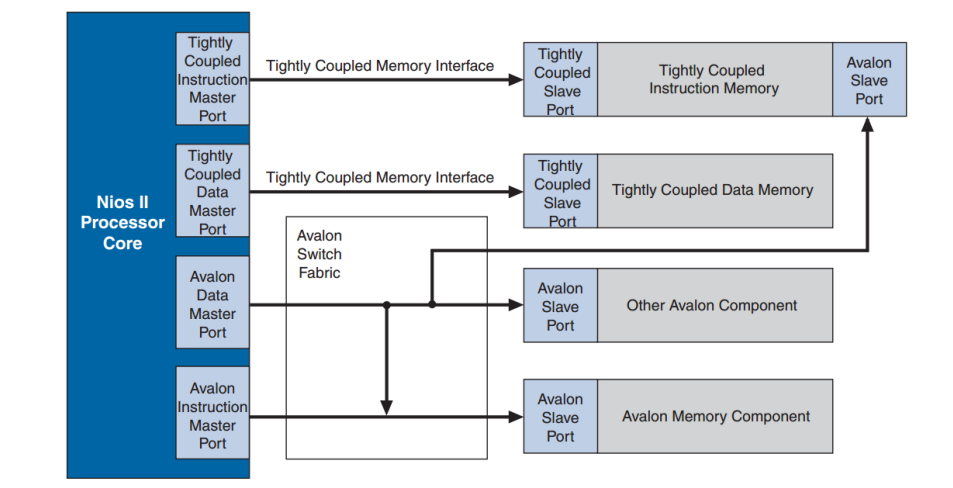


图 3.23 紧耦合存储器的连接示意图

从该图中我们可以看出，我们添加的两个片上存储器，一个是单端口的名称叫 Tightly Coupled Data Memory。另一个是双端口的，名叫 Tightly Coupled Instruction Memory。其中双端口的片上存储器，一端接到我们的 Tightly Coupled Instruction Memory Port，另一端则接到我们的 Avalon Data Master Port，它们最终的连接如图 3.24 所示。

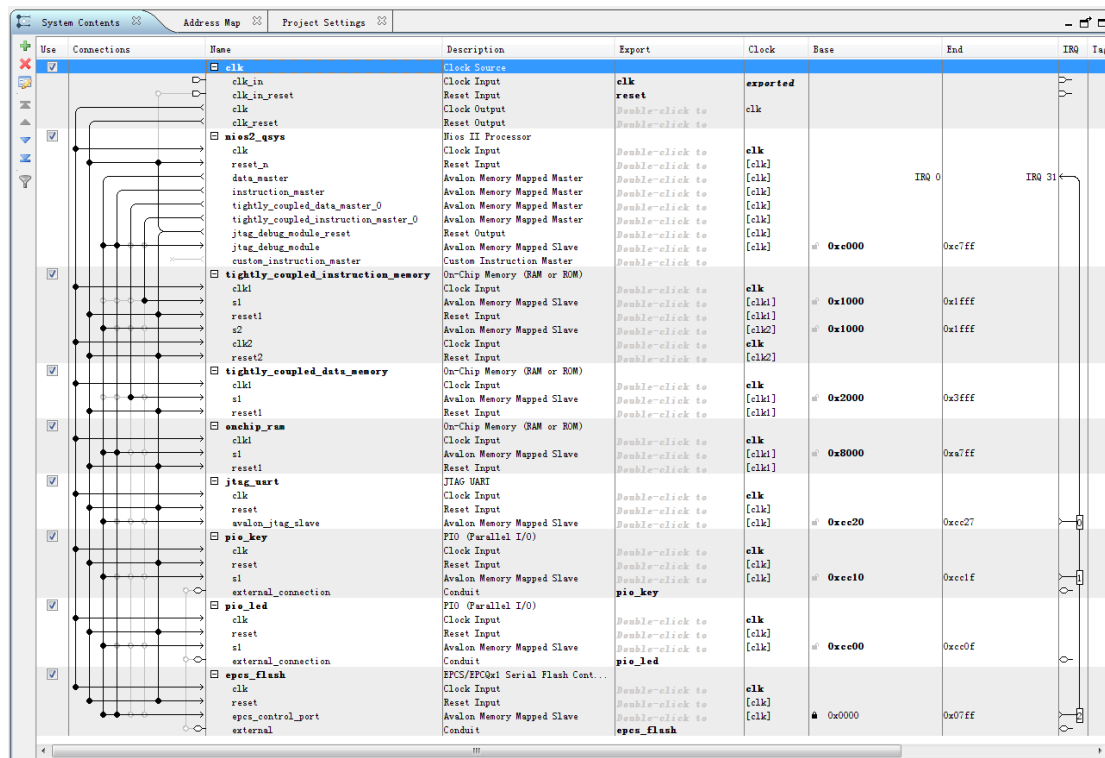


图 3.24 紧耦合存储器在 Qsys 软件中的连接

这里我们需要注意的是，当我们在 Qsys 软件完成了连接以后，我们还需要将 nios2_qsys 中的异常向量设置为 tightly_coupled_instruction_memory。至此，我们的紧耦合存储器就配置完毕了。

3.1.6 JTAG 调试模块

Nios II 结构中包含 JTAG 调试模块，提供片上仿真部件，便于 PC 主机对处理器进行远程控制。PC 上的软件调试工具可以通过与 JTAG 调试模块通信，提供调试和诊断功能。比如说：

- 把程序下载到存储器中。
- 启动和停止程序的执行。
- 设置断点和观察点。
- 分析寄存器和存储器。
- 采集实时的执行跟踪数据。

由于我们 Nios II 是一个高度可定制的处理器的，所以我们的 JTAG 调试模块也是一个完全可定制的，如图 3.25 所示。

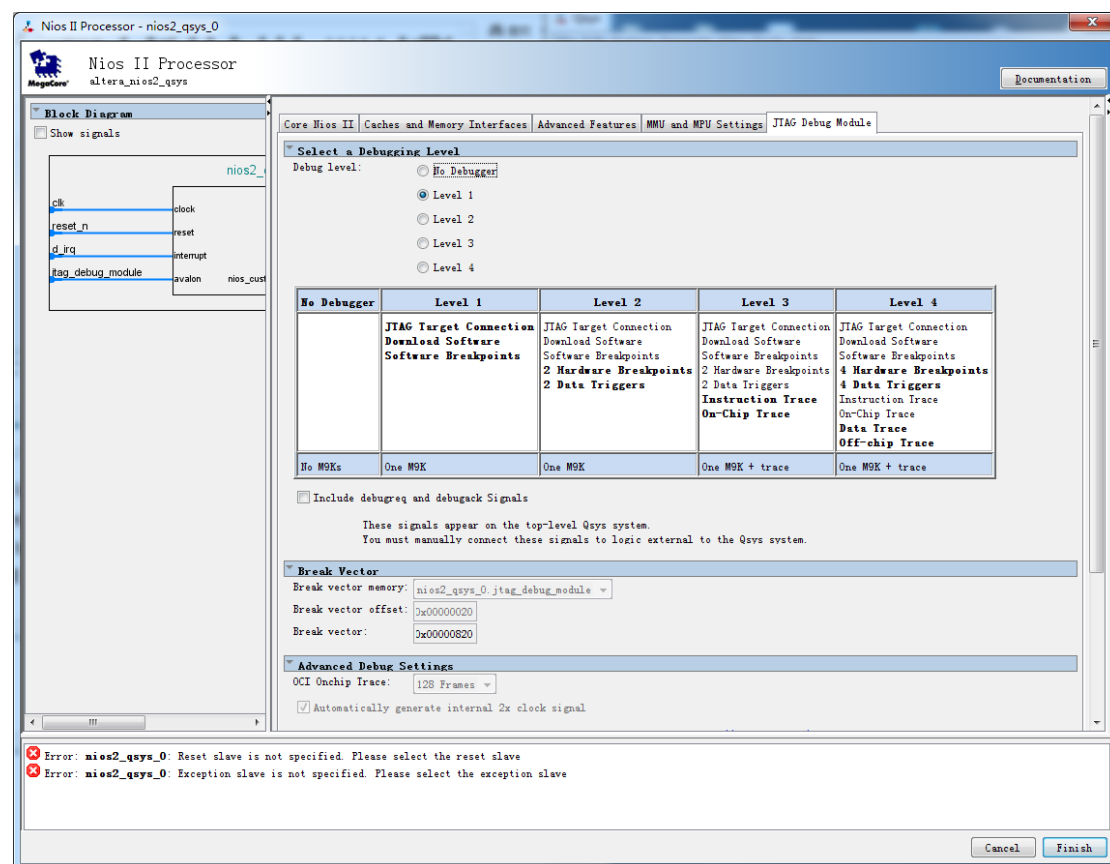


图 3.25 JTAG 调试模块在 Qsys 软件中的配置

从该图中我们可以看出，我们的 JTAG 调试模块分为 4 个等级，不同等级的 JTAG 调试模块，它所消耗的逻辑资源是不同的，同时它所具有的功能也是不同的，我们将它们总结如表 3.3 所示。

表 3.3 不同等级的 JTAG 调试模块对比表

Debug Feature	No Debug	Level 1	Level 2	Level 3	Level 4 (7)
Logic Usage	0	300—400 LEs	800—900 LEs	2,400—2,700 LEs	3,100—3,700 LEs
On-Chip Memory Usage	0	Two M4Ks	Two M4Ks	Four M4Ks	Four M4Ks
External I/O Pins Required (2)	0	0	0	0	20
JTAG Target Connection	No	Yes	Yes	Yes	Yes
Download Software	No	Yes	Yes	Yes	Yes
Software Breakpoints	None	Unlimited	Unlimited	Unlimited	Unlimited
Hardware Execution Breakpoints	0	None	2	2	4
Data Triggers	0	None	2	2	4
On-Chip Trace	0	None	None	Up to 64-KB frames (3)	Up to 64-KB frames
Off-Chip Trace (4)	0	None	None	None	128-KB frames

从表中我们可以看出，等级越高，消耗的逻辑资源也就越高，但相对的功能也就越强大，在调试的过程中，如果你只是需要一些简单的调试功能，那么你就可以选择一个低等级的 JTAG 调试模块。如果你需要一些比较复杂的调试功能，那么就需要选择一个高等级的 JTAG 调试模块。

3.1.7 Nios II 处理器性能

Nios II 处理器有 3 种类型：Nios II/e、Nios II/s 和 Nios II/f。我们将 3 种 Nios II 处理器的功能、面积和性能总结如表 3.4 所示。关于各处理器的更详细内容请参考 Altera 的官方文档。

表 3.4 3 种 Nios II 处理器的性能对比表

特性		处理器内核		
		Nios II/e	Nios II/s	Nios II/f
性能	DMIPS/MHz	0.15	0.74	1.16
	最大 DMIPS	31	127	218
	最大工作频率(f_{MAX})	200 MHz	165 MHz	185 MHz
大致尺寸（以 LE 为单位）		<700	<1,400	<1,800
流水线阶数		1	5	6
外部寻址空间		2GB	2GB	2GB
指令总线	高速缓存	—	512B 到 64KB	512B 到 64KB
	存储器流水线访问	—	可选	可选
	分支预测	—	静态	动态
	紧耦合存储器	—	可选	可选
数据总线	高速缓存	—	—	512B 到 64KB
	存储器流水线访问	—	—	—
	紧耦合存储器	—	—	可选
算术逻辑单元	硬件乘法器	—	3 周期	1 周期
	硬件除法器	—	可选	可选
	移位器	1 周期每比特	3 周期桶形移位器	1 周期 barrel 移位器
JTAG 调试模块	JTAG 接口、运行控制、软件断点支持	可选	可选	可选

	硬件断点支持	-	可选	可选
	片外跟踪缓冲区支持	-	可选	可选
存储器管理单元		-	-	可选
存储器保护单元		-	-	可选
异常处理	内部中断控制器	是	是	是
	外部中断控制器	否	否	可选
影子寄存器组		否	否	可选
用户模式支持		否	否	是
定制指令支持		是	是	是
ECC 支持		否	否	是

§ 3.2 Nios II软件框架结构的深入剖析

回顾第二章的 Qsys 的开发流程, 我们知道 Qsys 系统设计包括硬件设计与软件设计两部分。前一小节中我们对 Qsys 的硬件系统的架构与 Nios II 处理器内核作了较为详细的介绍。在本节中我们将从硬件转向软件, 着重为大家介绍 Nios II 程序框架结构。

在进入具体介绍之前, 请大家先回想一下之前直接用 Verilog 硬件描述语言驱动外设的情形: 我们根据外设的数据手册与时序规则, 将 FPGA 管脚与外设管脚的连接, 通过一系列硬件描述语言控制管脚电平的高低, 从而让 FPGA 实现外设的直接驱动。当这个外设比较容易, 如数码管, LED 时。我们可以直接控制最底层资源, 甚至对每个管脚在每个时刻的电平输出了如指掌。但是如果设计稍显复杂, 如设计 SDRAM 的驱动时, 往往对底层细节的过多关注就会成为一种累赘。

试想我们平时在电脑上编写 C 程序, 比如在显示器上输出一行字, 我们只用一句 printf() 即可完成, 至于打印命令怎么传到显示芯片上, 哪个芯片管脚怎么变化, 又怎么传到显示器上输出, 诸如此类涉及底层硬件的问题, 我们没必要关注太多。于是, 我们把用 printf() 这类高级语言描述设计逻辑的工作称为软件设计。然而, 软件只是一种抽象的看不见摸不着的东西, 它的结构接近于人类思维逻辑。无论软件再怎样构思精妙, 只有在硬件上才能体现出实际效果。做计算机开发应用程序的时候, 硬件是现成的, 软硬件之间的桥梁早就由操作系统给你搭好了, 我们只需专心完成软件的构思和设计就 OK。显而易见, 这种软硬件的分工会给电子设计带来极大的方便, 自然有人把这种分工方式引进 FPGA 设计领域, 琢磨着怎么在小小的 FPGA 上也搞个软硬件协同设计, 负责硬件设计的和负责软件的各司其职, 最后一整合, 效率倍增。

那么再来看看 Qsys 是怎么做的。拿出我们的 Zircon 开发板, 上面很多外设接口和与之连接的芯片, 那是硬件, 中间有一块大的 CycloneIV 芯片, 里面是空的, 等着我们编写逻辑下载到里面运行。而上一章里, 我们已经知道 Qsys 系统是用软硬件协同设计来完成。回顾 Qsys 的开发流程中, 我们需要用到的三个工具, 它们分别是 Quartus II、Qsys 和 Nios II SBT for Eclipse。Quartus II、Qsys 中进行的设计完成了硬件的控制与设备的驱动, 这些驱动有的是需要我们自行编写的, 有些是可以采用 Qsys 自带的 IP 核可以直接生成的。而基于 Nios II SBT for Eclipse 所完成的设计我们称为软件设计或软件开发, 这种软件开发和我们在 VC 中的设计大部分是极其类似的, 我们希望用行为级的设计语言去操作整个系统, 而不想过多关注硬件的细节。说白了, 就叫应用程序开发。那么跟计算机对比一下, 我们想一下: 在 Qsys 中连接软件和硬件之间的那座桥到底是什么呢? ALETRA 给了我们问题的答案, 如图 3.26 所示。

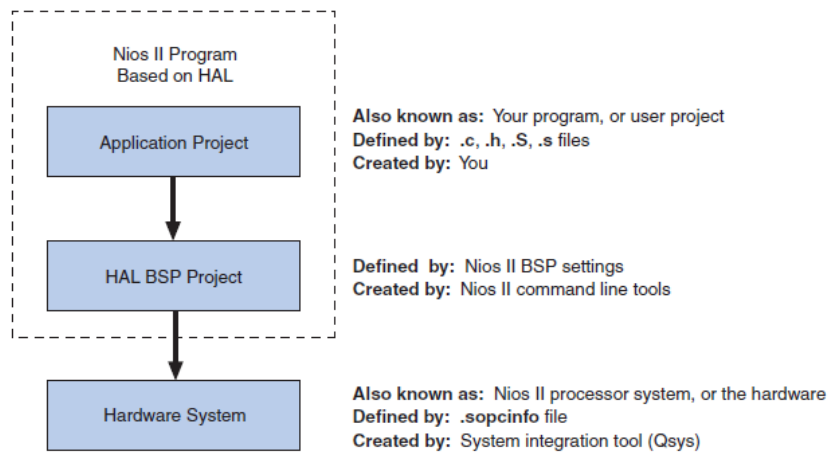


图 3.26 Nios II SBT for Eclipse 工程结构

从该图中我们可以看出，Application Project 是我们的软件工程，Hardware System 则是我们的硬件系统，而连接硬件与软件的这座桥叫做 HAL BSP Project，也就是我们所说的硬件抽象层（HAL）系统库或者是板级支持包（BSP）。硬件抽象层系统库是指在应用程序和系统硬件之间的一个系统库，HAL 应用程序接口（API）与 ANSI C 标准库综合在一起，可使用类似 C 语言的库函数来访问硬件设备或文件，如 printf()、fopen()、fwrite() 等函数。软件工程师可非常方便地使用这些系统库来与底层硬件通信，而无需关心底层硬件实现细节。这样在上层应用程序和底层硬件之间就构成了一个明显的界限，底层驱动程序的修改不会对应用程序造成任何影响。HAL 可以看作是一个支持应用程序开发的软件平台，它提供 API 函数接口，屏蔽硬件访问细节，为了方便用户编程，用户只要利用 HAL 提供的各种函数就可以编写应用程序。虽然占用了一些额外的资源，但是大大增加了应用程序的开发速度和可移植性。HAL 系统库在 Nios II SBT for Eclipse 中创建一个新的工程时，由 Nios II SBT for Eclipse 基于用户在 Qsys 中创建的 Nios II 处理器系统自动生成。用户不用创建或拷贝 HAL 文件，也不用编辑 HAL 中的任何源代码。HAL 设备驱动配置是和 Qsys 紧密相关的，如果硬件配置有了变化，那么 HAL 设备驱动配置也需要随之更改，从而避免了由于底层硬件的变化而产生的编程错误。

3.2.1 system.h 系统描述文件

system.h 文件是 HAL 系统库的基础，它提供了关于 Nios II 系统硬件的描述，是硬件和软件之间的桥梁。在第一次编译 Nios II SBT for Eclipse 过程中，编译工具会根据硬件系统文件（.sopcinfo 文件）生成一个描述硬件信息的 system.h 文件。system.h 文件是由编译工具根据 Qsys 系统的内容（.sopcinfo 文件）和系统库的配置设置生成的一个头文件，如图 3.27 所示。

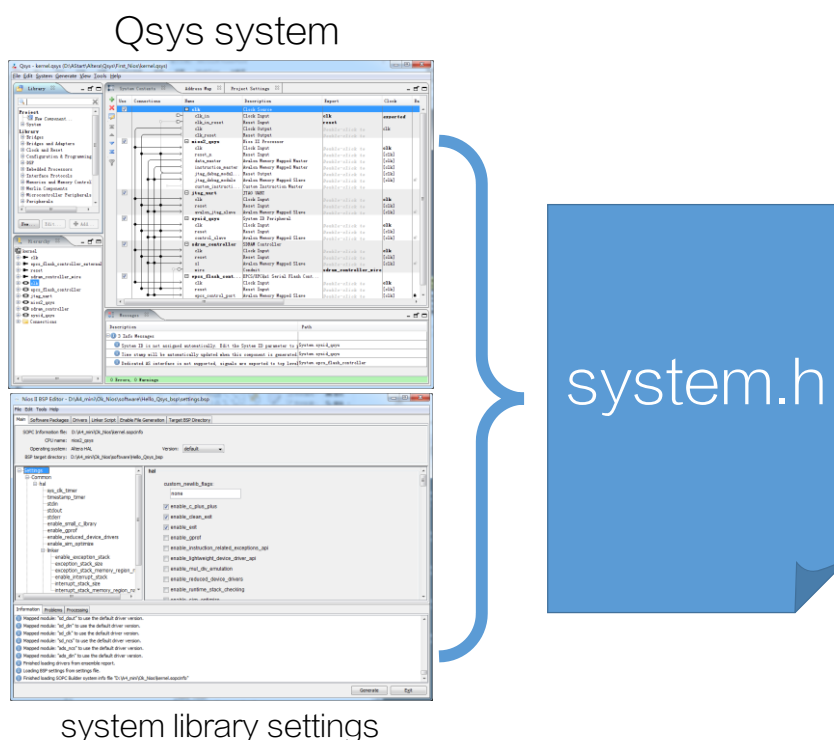


图 3.27 system.h 文件构成过程

从该图中我们可以看出，system.h 文件的内容主要由 2 个部分构成，一部分是描述系统库的设置信息；另一部分是给出了每个外围设备的详细信息，其内容有：外围设备的硬件配置、外设的基地址、中断优先级（如果外设中有中断）、外围器件的符号名称。如果用户选择了 uC/OS II 支持，system.h 文件中还将包括 uC/OS II 的配置信息。对于程序员来说，system.h 文件中的系统库设置信息基本用不到，所需要关心的仅仅是外围设备信息的宏定义及相关用法。接下来，我们就以第二章的 Qsys_First 工程中的 system.h 文件中 LED_PIO 设备的宏定义描述为例，详细分析该设备的宏定义及用法，如代码 3.1 所示。其他设备的宏定义与 LED_PIO 设备相似，这里就不在给出。

代码 3.1 LED_PIO 在 system.h 文件中的宏定义

1	<code>#define LED_PIO_BASE 0x1810</code>	//LED_PIO 设备的寄存器基地址
2	<code>#define LED_PIO_BIT_CLEARING_EDGE_REGISTER 0</code>	//LED_PIO 位清除边沿寄存器
3	<code>#define LED_PIO_BIT_MODIFYING_OUTPUT_REGISTER 0</code>	//LED_PIO 位修改输出寄存器
4	<code>#define LED_PIO_CAPTURE 0</code>	//LED_PIO 的同步捕获设置
5	<code>#define LED_PIO_DATA_WIDTH 8</code>	//LED_PIO 的数据位宽
6	<code>#define LED_PIO_DO_TEST_BENCH_WIRING 0</code>	//LED_PIO 的仿真线设置
7	<code>#define LED_PIO_DRIVEN_SIM_VALUE 0</code>	//LED_PIO 的仿真值
8	<code>#define LED_PIO_EDGE_TYPE "NONE"</code>	//LED_PIO 的边沿触发类型
9	<code>#define LED_PIO_FREQ 100000000</code>	//LED_PIO 的时钟频率
10	<code>#define LED_PIO_HAS_IN 0</code>	//LED_PIO 输入端口设置
11	<code>#define LED_PIO_HAS_OUT 1</code>	//LED_PIO 输出端口设置
12	<code>#define LED_PIO_HAS_TRI 0</code>	//LED_PIO 三态设置
13	<code>#define LED_PIO_IRQ -1</code>	//LED_PIO 中断号
14	<code>#define LED_PIO_IRQ_INTERRUPT_CONTROLLER_ID -1</code>	//LED_PIO 中断控制 ID

15	<code>#define LED_PIO_IRQ_TYPE "NONE"</code>	<code>//LED_PIO 中断类型</code>
16	<code>#define LED_PIO_NAME "/dev/led_pio"</code>	<code>//LED_PIO 设备名称及所在路径</code>
17	<code>#define LED_PIO_RESET_VALUE 0</code>	<code>//LED_PIO 的复位值</code>
18	<code>#define LED_PIO_SPAN 16</code>	<code>//LED_PIO 的地址长度</code>
19	<code>#define LED_PIO_TYPE "altera_avalon_pio"</code>	<code>//LED_PIO 的内核类型</code>

3.2.2 硬件抽象层的构成

HAL 系统库为用户提供下列支持：

- (1) 集成了 ANSI C 标准函数库，允许调用类似 C 标准库函数；
- (2) 提供访问 NiosII 系统每个设备的驱动程序；
- (3) 提供 HAL API，用于标准的函数接口如设备访问、中断处理以及 ALARM 等；
- (4) 提供系统初始化函数，为 main()函数和 C 库函数建立运行时环境。由于这里包含了 Bootload 以及程序重定位等工作，所以 Nios II 开发中没有像 ARM 系统开发中涉及 Bootload 等问题；
- (5) 提供设备初始化函数，在 main()函数前，分配设备空间并初始化所有的外围设备。

HAL 系统库与 ANSI C 标准库一起构成 HAL 的运行环境。HAL 使用的 C 标准库是在嵌入式上使用的免许可和版税的 C 语言程序库——Newlib。Newlib 是 C 语言标准库的一种开源码的实现。Nios II HAL 的结构如图 3.28 所示。

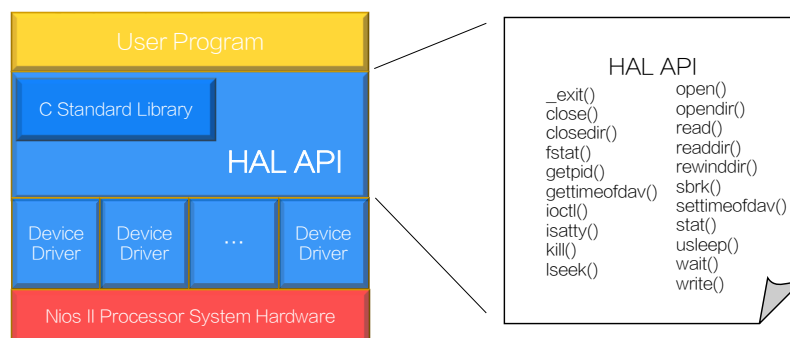


图 3.28 Nios II HAL 的结构

从该图中我们可以看出，用户程序架构在硬件抽象层和 C 标准库函数上，这说明用户程序要访问硬件设备至少有 4 种方法：

- (1) 调用 C 标准库函数，如 printf()和 fwrite()；
- (2) 调用硬件抽象层的 API 函数，如 write()；
- (3) 调用设备驱动程序，如 alt_avalon_uart_write()；
- (4) 直接访问设备寄存器（如 IORD_ALTERA_AVALON_PIO_DATA(base)）。

前两种方法的抽象层度最高，可移植性最好，对用户来说最容易实现，但是需要最多的额外开销；第三种方法抽象层度较低，有一定的移植性，额外开销较少，但还需要用户考虑对设备写数据前的其他硬件操作；最后一种方法完全是对硬件的直接访问，需要用户关注设备的每一个硬

件细节，无额外开销。对于简单的硬件设备，如 I/O 的访问，4 种方法都可以考虑；但是对于复杂的设备，建议使用前两种来访问。

3.2.3 类型定义

在编程的过程中，我们必须知道数据类型的宽度和精度。ANSI C 数据类型没有非常确切地定义数据宽度，它们的数据宽度取决于编译器的定义。HAL 使用 alt_types.h 头文件定义了一套支持 ANSI C 类型的数据类型，如表 3.5 所示。

表 3.5 HAL 数据类型定义

类型	说明
alt_8	有符号 8 位整数
alt_u8	无符号 8 位整数
alt_16	有符号 16 位整数
alt_u16	无符号 16 位整数
alt_32	有符号 32 位整数
alt_u32	无符号 32 位整数

Altera 提供的 GNU 编译器下的 ANSI C 数据类型宽度如表 3.6 所示。

表 3.6 ANSI C 数据类型宽度

类型	说明
char	8bits
short	16bits
long	32bits
int	32bits

3.2.4 通用设备模型

HAL 为嵌入式系统中最常见的外围设备提供了通用的设备模型。各类通用设备模型使用相同的应用编程接口（API），用户无需考虑底层硬件而利用与该设备模型相一致的 API 编写应用程序。HAL 支持以下几种通用的设备模型：

- (1) 字符型设备：发送和接收字符串的外围硬件设备；
- (2) 文件子系统：提供访问存储在物理设备中的文件操作；
- (3) DMA 设备：执行大量数据在数据源和目的地之间传输的外围设备。数据源和目的地可以是存储设备；
- (4) 定时器设备：对时钟脉冲计数并能产生周期行中断请求的外围设备。
- (5) Flash 设备。
- (6) 以太网设备：对 Altera 提供的轻量级 IP 协议提供访问的以太网连接。

上述的通用设备模型, 由于具有相同且与硬件细节无关的应用编程接口 (API), 大大方便了应用程序开发, 增加了应用程序的开发速度和可移植性。HAL 系统库定义了一套函数, 用户可以用来初始化和访问上述类型的设备。例如, 可以使用 C 的标准库函数 `printf()` 和 `fopen()` 来访问字符型设备和文件子系统。对于应用程序开发者来说, 不需要写底层驱动就可以和这些类型的设备进行基本通信。另一方面, 通用设备模型也大大方便了驱动程序的开发。各种类型的设备都定义了一套必须的驱动函数。此外, HAL 函数和应用程序都可以作为用户访问设备, 从而提高软件开发的效率, 节省开发时间。HAL 系统库调用驱动程序访问硬件, 应用程序不用直接调用驱动程序访问硬件, 而是调用 ANSI C 或 HAL API。从这个意义上说, 用户驱动程序其实已经作为 HAL API 的一部分在使用。

Altera 提供了许多在 Nios II 处理器系统中使用的外围设备, 且大多数都提供了 HAL 设备驱动, 以便直接通过 HAL API 来访问这些设备。下面列出 Altera 提供的完整 HAL 支持的外围设备:

- (1) 字符型设备包括 UART 核、JTAG 核以及 LCD16207 显示控制器;
- (2) 文件子系统包括只读文档系统;
- (3) DMA 设备包括 DMA 控制器核;
- (4) 定时器设备包括定时器核;
- (5) Flash 存储器设备包括通用 Flash 接口芯片和主动串行配置器件 EPCS 控制器。
- (6) 以太网设备包括 LAN9111 以太网 MAC/PHY 控制器并且需要 uC/OS-II 支持。

除了以上所列的 Altera 提供的外围设备外, 还有一些第三方提供的外围设备, 用户也可以定制自己的外围设备。所有这些为 Nios II 提供的外围设备都有一个头文件, 这个头文件定义了外围设备底层硬件接口。对于没有提供设备驱动程序的外围设备, 用户就只能利用头文件中的宏定义来访问该外设了。例如 Altera 提供了 PIO 核就不属于上述类型的外设, 只能通过其他提供了头文件的宏定义来操作 PIO。还有一些外设, 属于上述外设类型中一类, 但由于其特有的硬件特性, 不能完全使用相同的应用编程接口 (API) 来访问。HAL 系统库为此提供给了 UNIX 类型的 `ioctl()` 函数。由于 `ioctl()` 依赖于具体的外围设备的硬件特性, 所有其具体内容要查看各外设的文档说明。

3.2.5 异常处理

Nios II 采用典型的、简单的、异常处理方式, 使用一个简单的异常处理器来处理所有类型的中断。通过前面的异常和中断的学习, 我们可以知道当跳转到异常处理地址后, 处理器开始执行一段由 HAL 插入的代码, 判断中断源和中断优先级, 然后再跳转到用户的中断服务子程序 (ISR) 中。即异常发送后, 异常处理除 ISR 外的所有工作都由 HAL 系统库代码替用户完成。下面我们给出异常处理过程和中断服务程序间的关系图, 如图 3.29 所示。

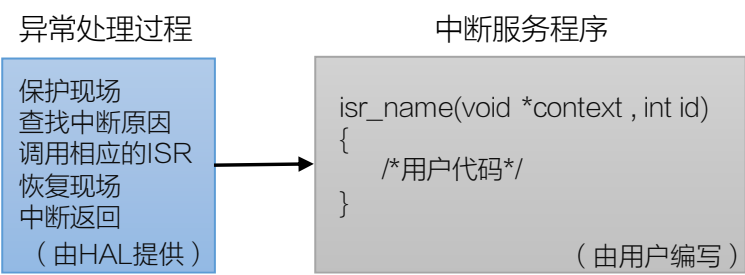


图 3.29 异常处理过程和中断服务程序

从图中我们可以看出，当异常发生后，处理器会依次执行保护现场、查找中断原因、调用相应的 ISR、恢复现场和中断返回。在 Nios II 中的异常处理中不采用中断向量表，所有异常和中断都驻留在一个单一存储空间的代码来处理，这个存储空间称为 exception address。在该存储空间，HAL 系统库会自动插入判断终端元和中断优先级的代码。对于用户，所要做的工作就是编写中断服务程序。为了方便创建和维护中断服务程序，HAL 系统库提供了中断 API 函数，如代码 3.2 所示。

代码 3.2 HAL 系统库提供的中断相关函数

```
1  /* 新版本的中断相关函数 */
2  int alt_ic_isr_register(alt_u32 ic_id,alt_u32 irq,alt_isr_func isr,void
   *isr_context,void *flags); //注册中断服务程序
3  int alt_ic_irq_enable (alt_u32 ic_id, alt_u32 irq); //使能单个中断
4  int alt_ic_irq_disable(alt_u32 ic_id, alt_u32 irq); //禁止单个中断
5  alt_irq_disable_all (void) //禁止所有中断
6  alt_irq_enable_all (alt_irq_context context) //使能所有中断
7  alt_irq_cpu_enable_interrupts() //允许嵌套
8  alt_u32 alt_ic_irq_enabled(alt_u32 ic_id, alt_u32 irq); //检测中断的状态
9  alt_irq_enabled (void) //检测中断的状态
10 alt_irq_pending() //返回当前挂起的中断
11 /* 老版本的中断相关函数 */
12 int alt_irq_register (alt_u32 id, void* context,alt_isr_func handler);//注册中断
   服务程序
13 alt_irq_enable(alt_u32 id) //使能单个中断
14 alt_irq_disable(alt_u32 id) //禁止单个中断
15 alt_irq_interruptible(alt_u32 priority) //允许嵌套
16 alt_irq_non_interruptible(alt_u32 mask) //禁止嵌套
```

上面所给出的中断相关函数源码均可在“C:\altera\13.1\nios2eds\components\altera_nios2\HAL\inc\sys\alt_irq.h”中找到，我们在使用这些函数时，必须包含该头文件#include<sys/alt_irq.h>。这里需要我们注意的是：alt_irq_interruptible()和alt_irq_non_interruptible()必须配对使用：当要实现中断嵌套时，先在中断服务程序中调用alt_irq_interruptible()，高优先级的中断就可以中断当前中断服务程序，在中断返回时一定要调用alt_irq_non_interruptible()关闭中断嵌套，否则会引起中断自锁。使用 HAL API 函数编写中断服务程序分 2 步：

- (1) 参照 void isr_name(void *context,alt_u32 id)函数原型编写中断服务子程序。其中

isr_name 是用户给 ISR 取的函数名; void *context 是指向传递给 ISR 的信息的全局变量指针 (通常是寄存器信息); id 是硬件中断号, 在 system.h 中声明。

- (2) 调用 alt_irq_register(alt_u32 id, void *context, void (*isr)(void *, alt_u32)) 或者是 alt_ic_isr_register(alt_u32 ic_id, alt_u32 irq, alt_isr_func isr, void *isr_context, void *flags); 函数向硬件抽象层登记中断服务子程序。其中 alt_u32 id 和 alt_u32 irq 是硬件中断号; void *context 和 void *isr_context 是指向传递给 ISR 的信息的全局变量指针 (通常是寄存器信息); void (*isr)(void *, alt_u32) 和 alt_isr_func isr 是指向 ISR 的函数指针; alt_u32 ic_id 是中断控制器标号, *flags 保留未用。

关于中断服务程序编写的例子可以参考 Altera 提供的模板程序 “C:\altera\13.1\nios2eds\examples\software\count_binary\count_binary.c”。中断服务程序主要处理一些高优先级, 实时性比较强的操作, 所以不可以在中断服务程序里面进行等待或者其他阻塞性的操作。下面给出一些编写中断服务程序的注意事项, 希望能在应用开发中起到一些帮助作用。

- (1) 尽量保持中断服务程序精简 (精简以为这快速和可靠);
- (2) 把无关紧要的事情放在中断服务程序之外处理;
- (3) 尽量避免调用 C 库函数 (如 printf(), 因为可能引起阻塞且运行时间无法预知);
- (4) 尽量避免进行浮点操作。

中断服务程序的调试和一般程序调试有些区别, 也复杂, 主要是因为中断事件何时发生是无法预测的。下面给出一些调试中断服务程序的经验。

- (1) 在中断服务程序内设置断点。当中断发生后, 处理器会在断点处停下, 用户可以单步调试中断服务程序。需要注意的是, 在单步调试的过程中, 硬件会忽略其他中断。
- (2) 使用 sprintf() 函数把关键数据写到内存中, 然后触发外部分析程序, 在中断服务程序中, 不可以调用 printf() 函数, 但可以调用 sprintf()。

§ 3.3 Nios II 用户程序引导过程深入剖析

3.3.1 什么是 Bootloader?

通过前面章节的学习, 我们知道了 Qsys 系统是可以下载到开发板的 Flash 中, 并且还能使开发板在一上电就能够运行我们的程序, 有的读者就会认为这一现象很神奇, 但好奇心比较强的读者便会有这么一个疑问, 为什么烧写到 Flash 中, 程序就能够上电自启动呢? 这是一个好问题, 下面我们就来揭开它的神秘面纱, 详细了解开发板在上电的一瞬间究竟做了什么。Nios II 的启动过程要经历两个过程:

- (1) FPGA 器件本身的配置过程。FPGA 器件在外部配置控制器或自身携带的配置控制器 (EPCS) 的控制下配置 FPGA 的内部逻辑。如果内部逻辑中使用了 Nios II, 则配置完成的 FPGA 中包含有 Nios II 软核 CPU。
- (2) Nios II 本身的引导过程。一旦 FPGA 配置成功后, Nios II 就被逻辑中的复位电路复位, 然后开始从头执行软件代码。

这两个过程应该不难理解，前一个可以理解为硬件构建，即把 FPGA 从白片变成一个包含 Qsys 的 SOPC 硬件系统，这部分我们并不陌生了；后一个可以理解为软件复位，让我们所开发的软件程序运行于 Nios II 的 CPU 上。而在此 Bootloader 起了关键的作用。那么到底什么是 Bootloader 呢？要了解 Bootloader，首先我们要弄清楚几个地址概念，如图 3.30 所示。

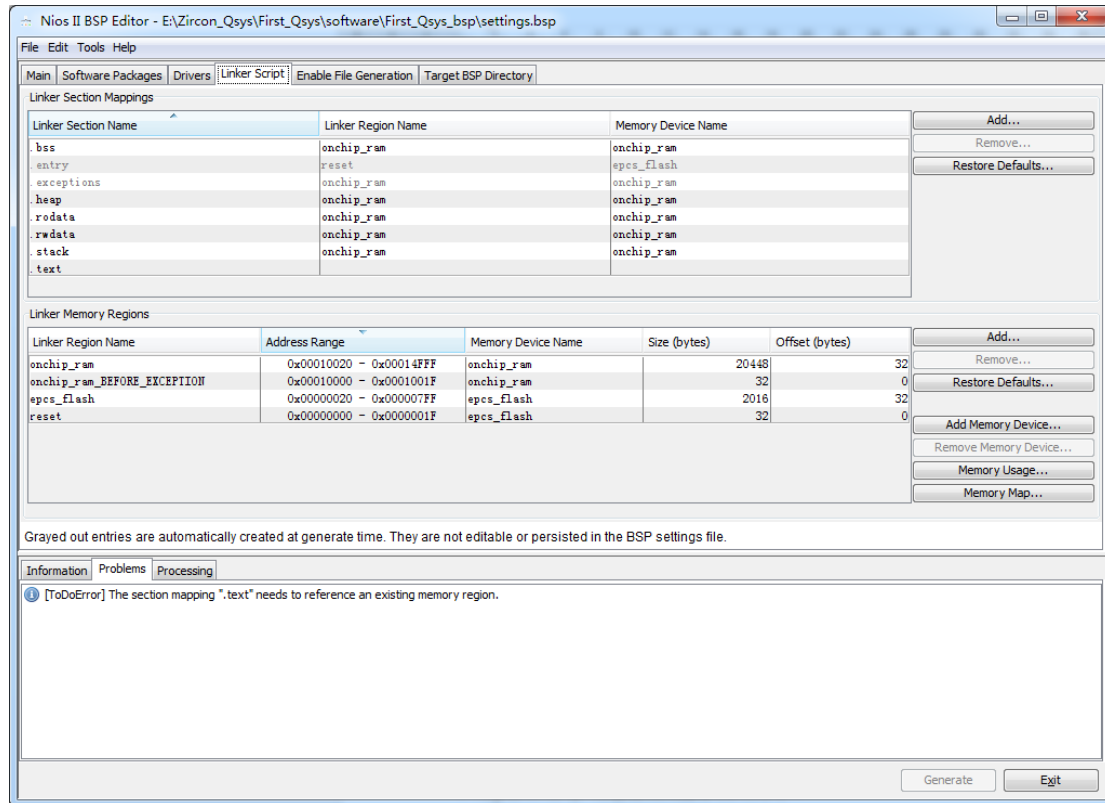


图 3.30 Linker Script 配置页面

从该图中我们可以看到，有.bss、.entry、.exceptions、.heap、.rodata、.rdata、.stack和.text 这几个地址，下面我们就对它们分别进行介绍：

- (1) .bss (未初始化变量数据段)：未初始化变量，如全局变量定义后，但未初始化的都放这里；
- (2) .entry(复位地址)：最终软件代码存储区，程序从此处开始启动；
- (3) .exceptions(异常处理地址)：系统异常处理代码存放的地方
- (4) .heap (堆)：动态分配内存的存储区，从小地址向大地址方向增长；
- (5) .rodata (只读数据段)：用于存放程序中的静态全局变量，所有赋了初值的全局变量都放在这里；
- (6) .rdata (可读写数据段)：用于存放程序中可读写变量和指针变量；
- (7) .stack (栈)：函数调用变量与临时数据存储器区（如返回地址，现场信息），从大地址向小地址方向增长；
- (8) .text (正文段，只读)：代码执行区，也就是就是 Nios II 程序最终运行的地方；一个程序只有一个副本；只读，防止程序由于意外事故而修改自身指令；

知道了这些地址的功能，接下来我们再来看下如何设置这些地址。对于.bss、.heap、.rodata、.rwdata、.stack 和.text 地址，我们可以在 Nios II BSP Editor 中的 Linker Script 配置页面中直接设置，对于.entry 和.exceptions 这两个地址，我们在 Nios II BSP Editor 中的 Linker Script 配置页面中是不可以设置的，从上面的图片中我们可以看到它们是灰色的。如果我们想要设置这两个地址，那么我们需要在 Qsys 软件中进行设置，如图 3.31 所示。

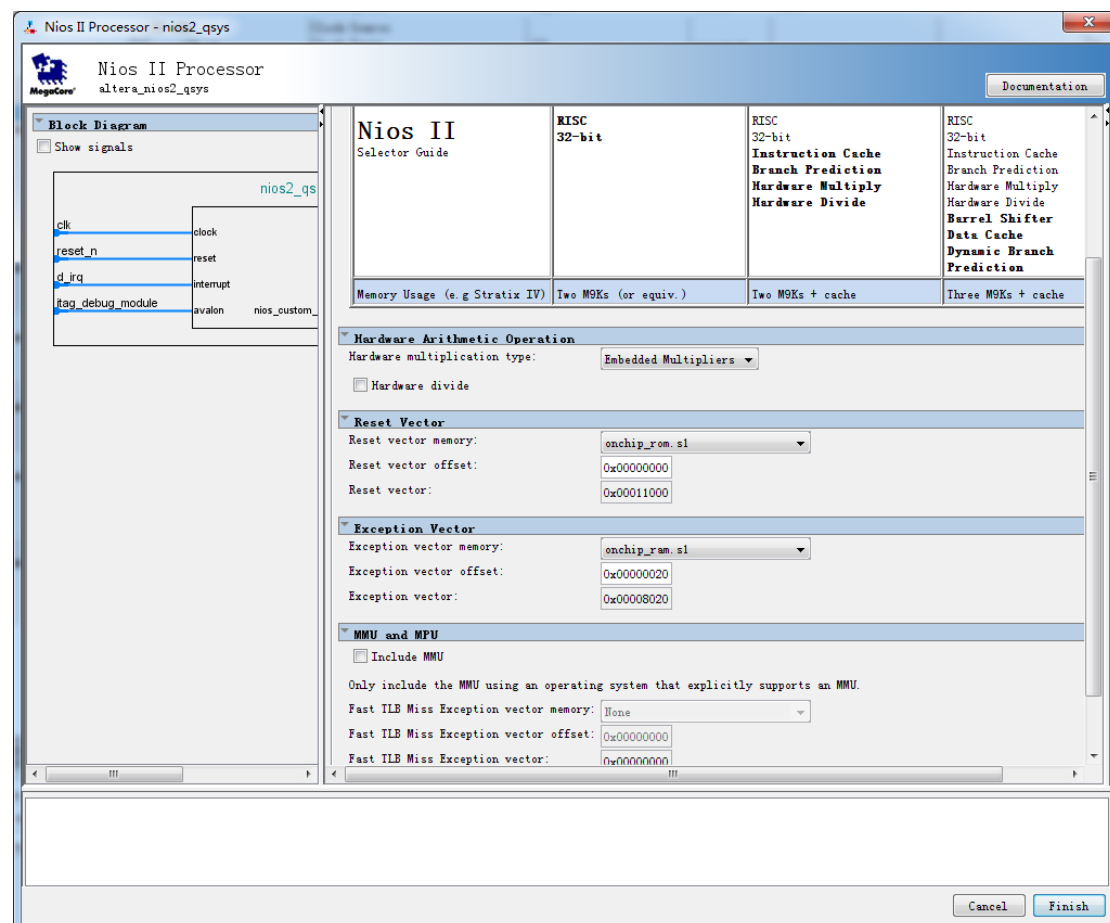


图 3.31 复位地址和异常地址设置页面

讲完了这些地址，下面我们举一个简单的例子：如果我们将 Reset Vector memory 和 Exception Vector Memory 都指向 epcs_flash，如图 3.32 所示。

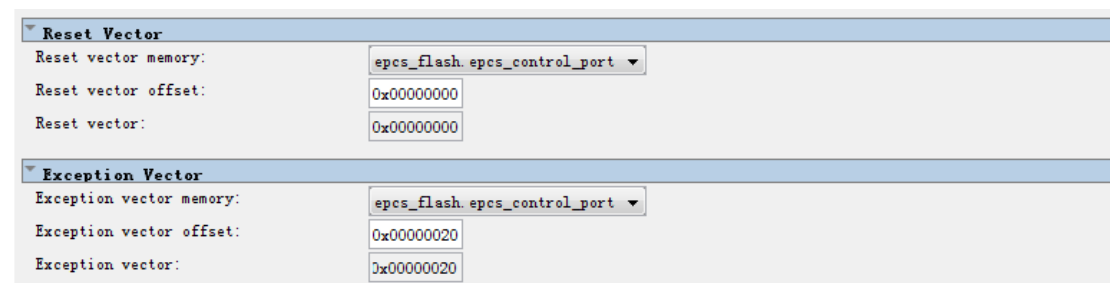


图 3.32 将复位地址和异常地址都指向了 epcs_flash

并且在 Nios II BSP Editor 中 System Library 下的.bss、.heap、.rodata、.rwdata、.stack 和.text 都设为 cfi_flash，如图 3.33 所示。

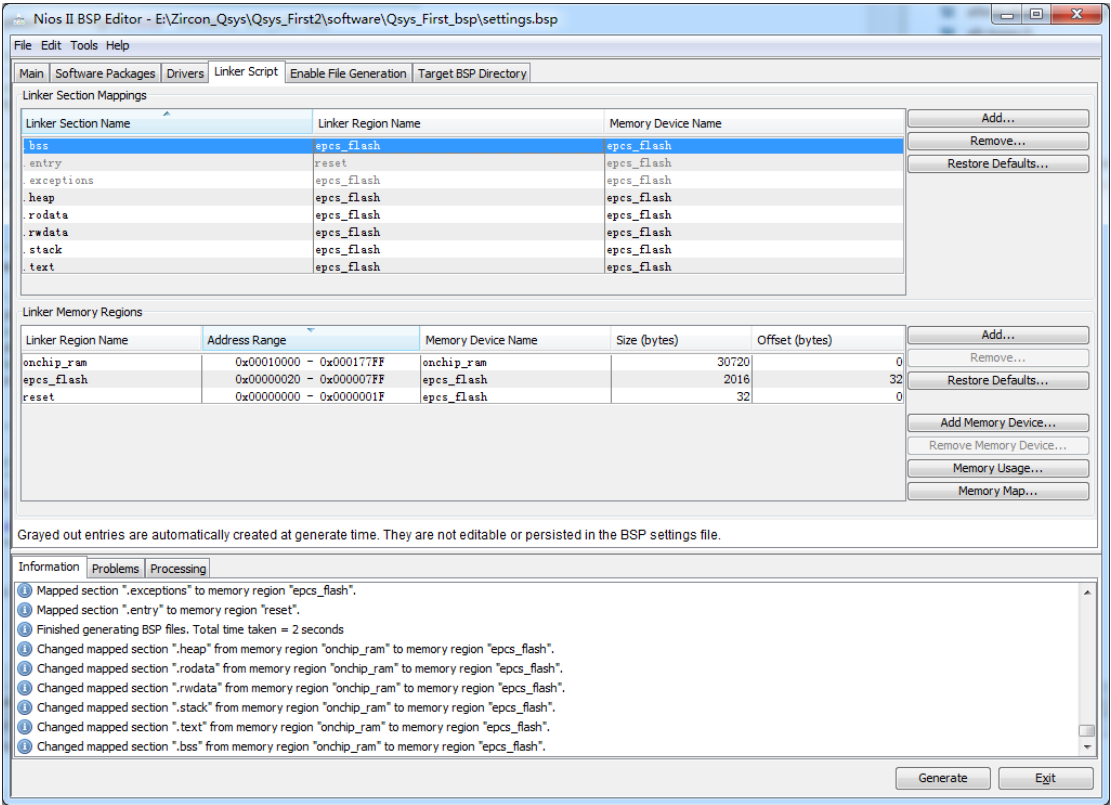


图 3.33 Linker Script 配置页面

那么这时候会如何呢？显然，这样设置就表示下载的时候程序代码会固化到 epcs_flash 芯片上，并且程序也在 epcs_flash 中运行。通过学习《数字电路篇》，我们知道虽然 Flash 具有较低的价格、较高的容量、掉电数据不丢失等优点，但是 Flash 的运行速度很慢；程序运行在 Flash 中对运行速度有影响，也容易损坏 Flash 芯片。相反，易失性存储器，如 sdram。具有较快的运行速度，所以通常我们会把程序放到 sdram 中运行。要让程序在 sdram 中运行，那么需要将 Exception Vector 设置成 sdram，并且还将 Nios II BSP Editor 中 System Library 下的 .bss、.heap、.rodata、.rwdata、.stack 和 .text 都设为 sdram，如图 3.34 所示和图 3.35 所示。

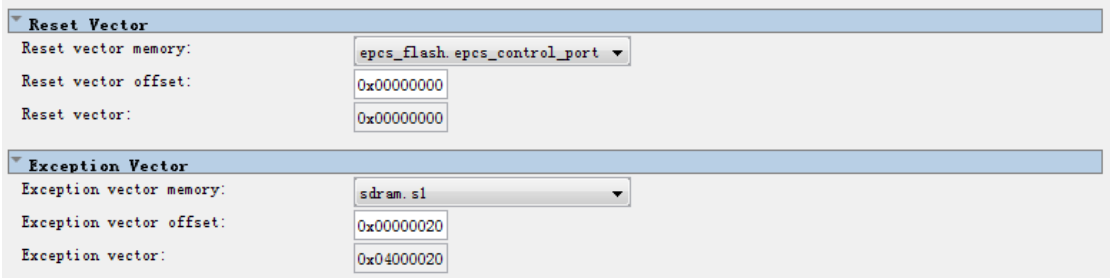


图 3.34 设置 Nios II 复位地址和异常地址

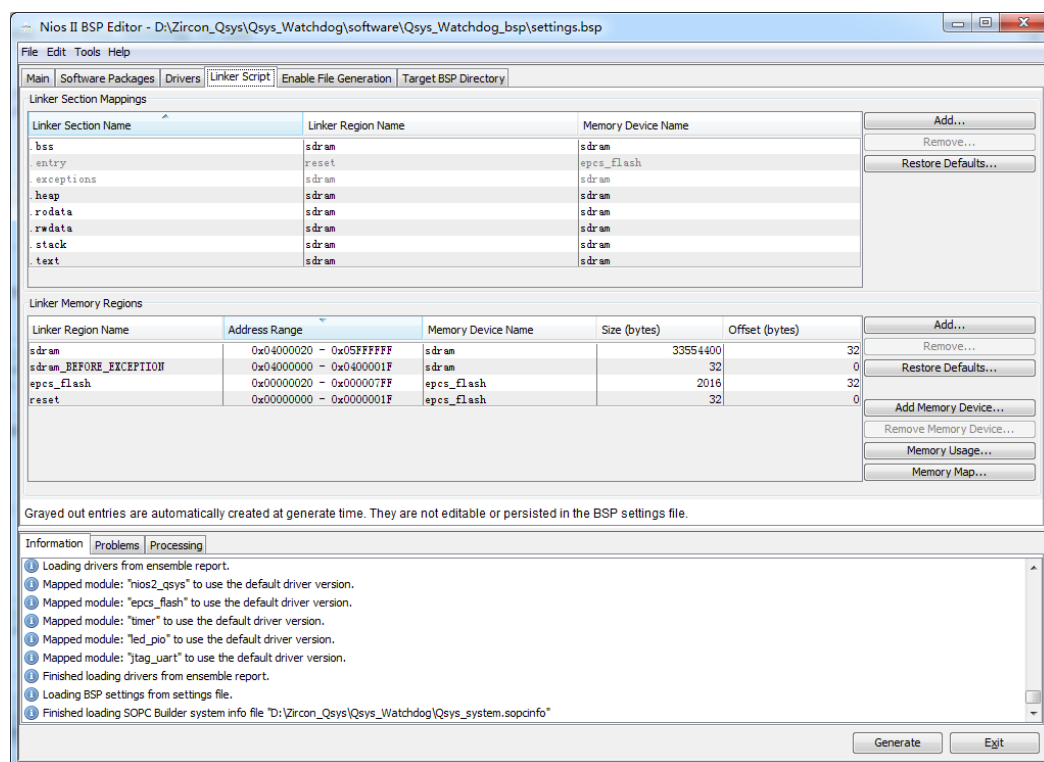


图 3.35 Linker Script 配置页面

从上面的描述我们知道，sdram 掉电之后数据都会丢失，那么问题来了，如何使用 sdram 来存储程序代码呢？不急！办法是有的，下面就听我慢慢道来，我们不是还有个 epcs_flash 吗？它不会因为掉电而丢失数据的，因此我们 Reset vector memory 指向了 epcs_flash 中去，在 Linker Script 配置页面中我们依然可以看到 reset 指向 epcs_flash，很奇怪吧，什么也没变啊？怎么让程序运行在 sdram 中呢？这时候我们需要一位助手，它就是 bootloader，这是一段引导程序，它是由 Nios II SBT for Eclipse 根据你的设置自动生成。它会把存放在 epcs_flash 中的相关的程序段搬到它们需要运行的地方，就是我们设置的 sdram 中，如果是 onchip_memory 的话，程序就搬到 onchip_memory 里面去了。这样的话，不仅实现了程序的掉电存储，而且还让它在 sdram 中运行了。这里需要我们注意的是，程序在运行前所有的数据都是存放在 EPCS Flash 中。如果你设计的系统 Reset address 与其它地址的设置不相同，比如 Reset address 和 Exception address，Reset address 和 Linker Script 配置页面中的 .text 设成不同的存储器，那么系统从 Reset address 启动时都会从 Flash 中把相关数据自动下载到相应的地址或者初始化相应地址，也就是说：

- (1) 如果 Exception address 和 Reset address 不一样，那么程序从 Reset address 启动后将把放在 Reset address 处的系统异常处理代码拷贝到 Exception address。
- (2) 如果 .text address 和 Reset address 不一样，那么程序从 Reset address 启动后将把放在 Reset address 处的普通只读程序代码拷贝到 text address。

通过上面的认识，我们就比较清楚了各存储器间可进行哪些搭配了，Reset address 的选择可选 EPCS Flash 或者 CFI Flash；program memory(.text)可选择 Onchip Memory、SDRAM 等。

3.3.2 用户程序引导过程

说了这么多我想大家都已经对 Nios II 的 boot 设置以及 Bootloader 都有所了解了吧，在整个 boot 过程中，bootloader 作为“代码搬用工”起到了十分关键的作用。如果存在，那么 Bootloader 就是系统加电启动后运行的第一段代码，Bootloader 能够根据用户在 Nios II SBT for Eclipse 中设置的用户程序文件(.elf)连接地址来重新装载程序，然后跳过.elf 文件的连接地址执行程序。下面我们给出 Bootloader 的执行过程：

- (1) 完成复位子程序的功能。在从复位地址开始的 32 字节（指令缓存行的尺寸）空间内完成复位的软件响应。
- (2) 加载应用项目程序映像中的代码和数据段（从装载地址 Reset address 加载到程序运行地址.text）。
- (3) 跳转到应用程序入口（通常是_start）。

当 Bootloader 跳转到应用程序入口后，便会用到 Nios II 软件架构为我们准备的启动代码，它们分别是 Crt0.S 和 alt_main.c 两个 HAL 源文件。Crt0.S 位于“C:\altera\13.1\nios2eds\components\altera_nios2\HAL\src\crt0.s”中，主要完成了以下功能：

- (1) 初始化指令 Cache，初始化数据 Cache；
- (2) 设置堆栈指针(Stack Pointer)寄存器；
- (3) 全局指针(Global Pointer)寄存器；
- (4) 初始化 bss 段，__bss_start ~ __bss_end 清零；
- (5) 如果系统中没有 bootloader，则把可读写数据(.rwdata),只读数据(.rodata)和异常向量表装入 RAM 中；
- (6) 调用 alt_main()函数，也就是执行 alt_main.c 文件中的内容。

alt_main.c 位于“C:\altera\13.1\nios2eds\components\altera_hal\HAL\src\alt_main.c”文件中，主要完成了以下功能：

- (1) 调用 alt_irq_init()初始化中断控制器。
- (2) 调用 ALT_OS_INIT()初始化 OS。
- (3) 如果使用了操作系统，那么初始化用于控制访问文件描述器列表的信号量。
- (4) 使能中断。
- (5) 调用 alt_sys_init()初始化添加的 IP 核的设备驱动。
- (6) 把标准输入输出设备(stdin, stdout, stderr)映射到合适的设备上
- (7) 调用 C++的构造器；
- (8) 调用 main()函数
- (9) 调用 exit()函数。

最后我们将整个用户程序引导过程，总结如图 3.36 所示。

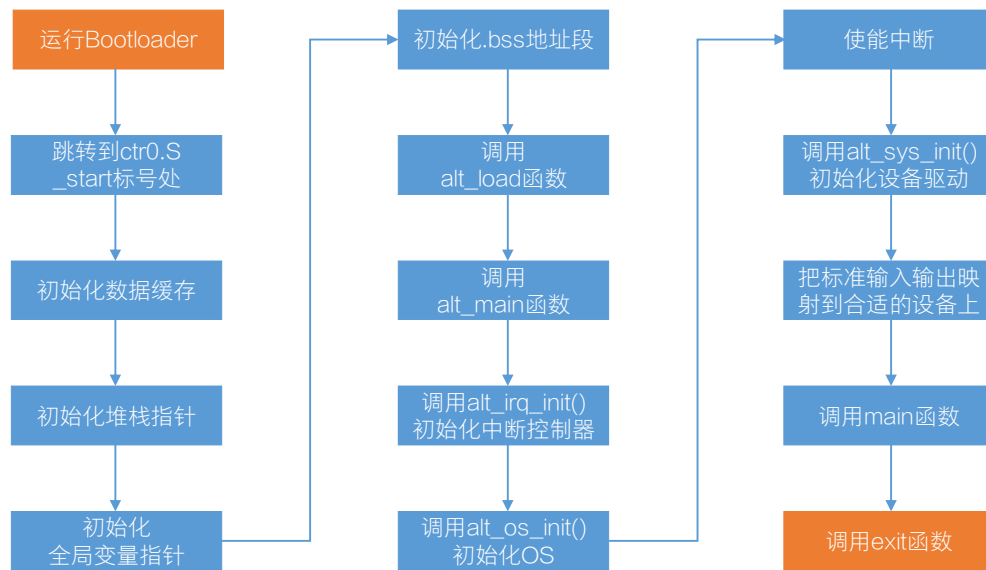


图 3.36 整个用户程序引导过程图

至此，我们第三章剥去神秘的外衣——Qsys 到底是如何运行的就讲解完了。接下来我们讲解的是第四章 Qsys 丰富多彩的内置 IP 核。

第四章

Qsys 丰富多彩的内置 IP 核

第四章

Qsys 丰富多彩的 IP 核

终于熬过了枯燥的第三章，接下来，我们将迎来更加轻松有趣的第四章。为什么这么说呢，因为我们在第三章节中主要讲解了一些理论层面的知识，所以大家听起来难免会枯燥乏味。而我们的第四章主要讲解一些应用层面的知识，这些应用知识我们大家看的见、摸的着，能够真枪实战的去操练，因此，我们的第四章要比第三章更加轻松有趣。在开始进入第四章节之前，我们觉得有必要要和大家说明一下，我们后续章节的内容安排，说到这内容安排，我们不得不要从我们开发板上的 UI 系统说起，大家看，如图 4.1 所示。



图 4.1 A4 开发板的 UI 系统主界面

这就是我们的开发板上的 UI 系统，从图中我们可以看出，主页面中有简介、测试、演示、提示四个功能选项，下面我们就来简单的介绍一下这四个功能选择，简介页面是用来介绍我们整个 UI 系统相关功能的；测试页面是用来测试我们开发板上的外设，演示页面是演示我们《项目实战篇》中的工程功能的；提示页面是用来告诉我们一些开发板的使用事项。

如果你已经购买了我们的开发板，那么我相信这些功能选项你都已经比较熟悉了，用过了我们的这个 UI 系统，你有没有感觉到它很人性化。当你在操作这个 UI 系统的时候，你有没有一种错觉，你使用的不在是一个开发板，而是一个手机！看到这里，也许你会这么有一个想法，我们的这个 UI 系统是如何制作的呢？大家不要着急，下面我们就给大家娓娓道来。由于我们的 UI 系统涉及的知识面比较广泛，所以我们的 UI 系统制作起来比较复杂。我们想要实现 UI 系统，我们需要掌握以下三个方面的知识：

- (1) 内置的 IP 核：这部分的内容也就是我们本章节中所讲的内容。
- (2) 自定义 IP 核：这部分的内容我们将会在本章节中进行详细的讲解。
- (3) 操作系统及 UI 界面：这部分的内容我们将会在本章节中进行详细的讲解。

我们只有掌握了这三个方面的知识，我们才能设计出 UI 系统，大家在学习这三个方面的内容时，要脚踏实地，一步一个脚印。相信大家都听说过这么一句话：Rome was not built in a day。

接下来我们就来进行第四章，Qsys 丰富多彩的内置 IP 核的内容讲解，我们已经知道，Qsys 能够使我们简单直观的构建一个片上嵌入式系统，不需要拘泥于具体底层逻辑代码的实现。这其中的关键就在于 Altera 已经帮助大家设计出了许多常用外设接口及其协议模块的硬件描述语言

实现框架与软件驱动，也就是我们前面所说的 IP 核。这些常用外设 IP 核不仅仅只有我们使用 PIO 和 JTAG_UART 这些 IP 核，它还包含了大量的其它 IP 核，比如定时器、串行通信接口、SDRAM 控制器和其他的存储器接口等。这里我们将常用的 IP 核总结如表 4.1 所示。

表 4.1 Qsys 中常用的 IP 核

外围设备	功能说明
PIO	1~32 位并行 I/O 接口（输入、输出和边沿捕捉）
SDRAM	提供一个连接片外 SDRAM 芯片的 Avalon 接口
EPCS	实现处理器系统访问 Altera EPCS 串行配置器件
Timer	32 位定时器，能被用作周期性脉冲发生器或系统监视定时器
System ID	实现 Nios II 系统与所指定的硬件系统间的匹配
DMA	在外围设备和存储器之间有效地进行批量数据传送
JTAG UART	实现基于 FPGA 的嵌入式系统与主机之间的串行符号流通信
UART	通用串行接口，波特率、数据位等可调，流量控制信号可选
SPI	3 线主/从串行外设接口

这些 IP 核我们将会在后继章节中一一介绍，这些 IP 核虽然功能不同，但是它们使用起来的过程是极其类似的。因此，对于每个外设 IP 核的介绍，我们都遵循如下的模式：

- (1) IP 核的综述；
- (2) IP 核的寄存器描述；
- (3) IP 核的配置选项；
- (4) IP 核的软件编程；
- (5) IP 核的应用实例；

值得注意的是，由于每个 IP 核的功能不同，所以它们是相对独立的。我们尽可能完整地给出了每个 IP 的特点、配置以及软件编程，供大家在使用这些外设定制 Nios II 系统时查阅。但由于篇幅冗长，在阅读过程中，你可以任意跳过里面的部分章节，并不会影响整个知识框架的理解。这里我们推荐你先精读前三个外设 IP，后面的可以当有需要用到的时候再一一查阅。

此外，学习这部分的建议大家先不求甚解，再做到深入理解。为什么要不求甚解？因为 Altera 为我们提供这些 IP 的目的就是为了让大家更方便快捷地对这些通用外设进行驱动。每个 IP 的作用就是为了驱动一个特定的外设，即使我们不知道它底层是如何实现的，这也并不妨碍我们用好它。如果你时间有限，且只是想做到“会用”，那么开始可以不需要花很多的时间把文档里面的每一行都了解清楚，只需要跟着我们的例程走下去就行了。因为你最想知道的是怎么让它工作起来，而不是它怎么工作的，最终你一定能够娴熟地使用每一种外设。那么又为什么要再做到深入理解呢？原因之一，这里先卖个关子，读到下一章你自然就知道了。第二个原因嘛……因为我们最终的目标就是做一名优秀的工程师，优秀的工程师怎么能不求甚解呢？

§ 4.1 PIO IP核

4.1.1 PIO IP 核的综述

首先我们介绍的是 PIO IP 核的综述，PIO IP 核是 Nios II 处理器支持的可编程并行输入/输出，它是一种带有 Avalon 总线借口的核，提供了一个介于 Avalon 存储器映射的从端口和普通 I/O 端口之间的存储器映射借口。I/O 端口一端连接偏上的用户逻辑，另一端链接 FPGA 外部的 I/O 管脚。PIO IP 核为用户逻辑货外部器件提供了简单的直接 I/O 操作，可以直接完成位操作。每个 Avalon 接口的 PIO 内核可提供 32 个 I/O 端口且端口数可设置，用户可以添加一个或多个 PIO IP 核。CPU 通过 I/O 寄存器控制 I/O 端口的行为。I/O 可以配置为输入、输出和三态，还可以用来检测电平事件和边沿事件。

4.1.2 PIO IP 核的寄存器描述

介绍完了 PIO IP 核的综述，接下来我们再来看看 PIO IP 核的寄存器描述，PIO IP 核的寄存器描述如表 4.2 所示。

表 4.2 PIO IP 核的寄存器描述

偏移量	寄存器名称	操作	(n-1) ... 2 1 0
0	data	读	返回在输入引脚上出现的值
		写	向 PIO 写入新值
1	direction	读/写	控制每个 I/O 口的输入/输出方向。0: 输入; 1: 输出
2	interruptmask	读/写	使能或禁止每个输入端口的 IRQ。1: 中断使能; 0: 禁止中断
3	edgedcapture	读/写	当边沿事件发生时对应位置 1
4	outset	写	指定输出位为 1
5	outclear	写	指定输出位为 0

从 PIO IP 核寄存器的描述表格中我们可以看出，PIO IP 核有 6 个寄存器，接下来我们就对这 6 个寄存器分别进行一个介绍：

(1) 数据寄存器 (Data)

读数据寄存器，返回在输入引脚上出现的值。如果 PIO IP 核硬件配置为“Output Ports Only”，则读数据寄存器返回未定义的值。写数据寄存器驱动输出口输出写入的值。如果 PIO IP 核硬件配置为“Input Ports Only”，则写数据寄存器无效。如果 PIO IP 核硬件配置在双向模式下，那么方向寄存器中对应的位设 1（输出）时，值才输出。

(2) 方向寄存器 (Direction)

只有 PIO IP 核工作模式配置为“Bidirectional ports”时，方向寄存器才存在。PIO IP 核工作模式（输入、输出或双向）在添加 PIO IP 核时指定，且在系统生成后不能改变。在仅为输入或仅为输出模式下，方向寄存器并不存在，此时，读方向寄存器返回未定义的值，写方向寄存器无效。方向寄存器控制每个 PIO 口的数据方向。当方向寄存器中的位 n 设为 1 时，端口 n 为输出模式；0 时，端口 n 为输入模式。复位后，方向寄存器的所有位设置为 0。因此所有双向 I/O 口配置为输入。

(3) 中断屏蔽寄存器 (Interruptmask)

<http://www.fpga.gs/>

当中断屏蔽寄存器的位设为 1 时，使能相对应的 PIO 输入口中断。中断操作取决 PIO IP 核的硬件配置，只有配置为输入口时才能进行中断操作。中断屏蔽寄存器只有在硬件配置为“Generate IRQ”时才存在。如果硬件配置未选中“Generate IRQ”，那么读中断屏蔽寄存器返回未定义的值，写中断屏蔽寄存器无效。复位后，中断屏蔽寄存器所有位为 0，因此禁止所有 PIO 口的中断。

(4) 边沿捕获寄存器 (Edgedcapture)

只要在输入口上检测到边沿事件时，边沿捕获寄存器中对应位 n 置 1。Avalon 主控制器可读边沿捕获寄存器来确定边沿在哪个 PIO 输入口出现。写任意值到边沿捕获寄存器将使寄存器所有位清 0。要检测的边沿类型在 PIO IP 核添加时指定。当硬件配置有边沿捕获功能时，边沿捕获寄存器才存在，否则，读边沿捕获寄存器返回未定义的值，写边沿捕获寄存器无效。

(5) 置位和清零寄存器 (outset and outclear)

你可以使用置位和清零寄存器设置输出端口为 1 或 0。比如，当我们想把输出端口的第 6 位设置成 1 时，我们可以写 0x40 到置位寄存器。当我们想把输出端口的第 3 位设置成 0 时，我们可以写 0x08 到清零寄存器。

4.1.3 PIO IP 核的配置选项

介绍完了我们的 PIO IP 核的寄存器描述，接下来我们再来看一下 PIO IP 核在 Qsys 中的配置选项，图 4.2 给出了 PIO IP 核的配置选项图。

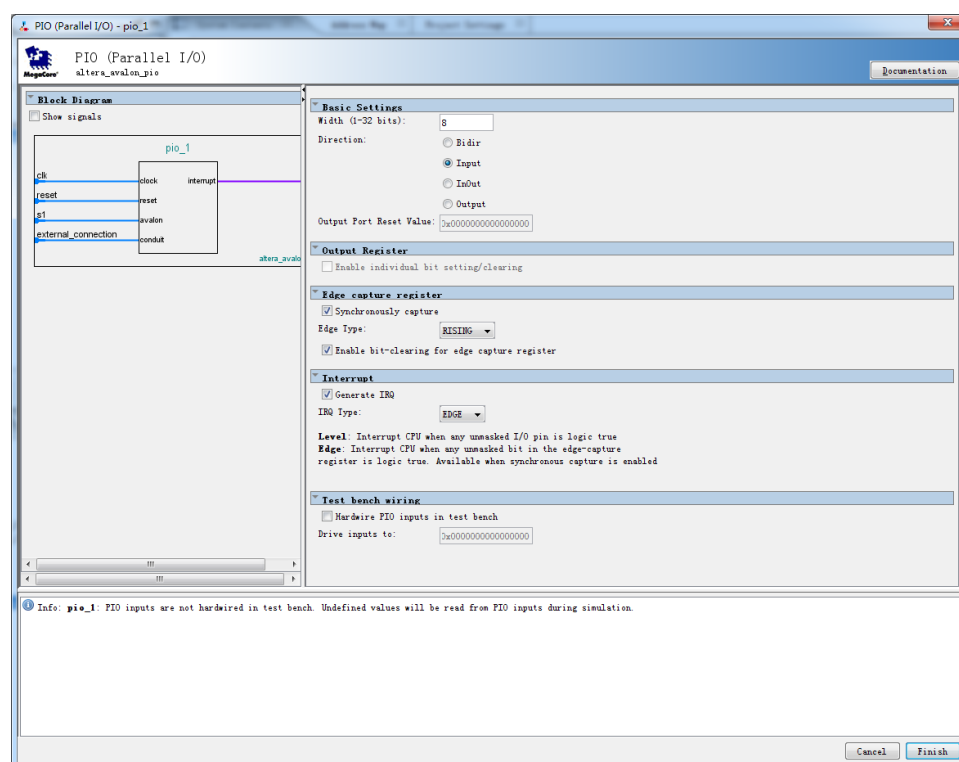


图 4.2 PIO IP 核的配置选项图

从 PIO IP 核的配置选项图中我们可以看出，PIO IP 核在 Qsys 有 5 种设置方式，接下来我们就对这 5 种设置方式分别进行介绍：

(1) 基础设置 (Basic Settings)

Basic Settings 允许用户指定 I/O 口的位宽和方向。Width 设置可为 1~32 的任何整数值，如果 Width 设置为 8，则 PIO 为 8 位宽。Direction 设置有 4 个选项，各选项描述如表 4.3 所示。

表 4.3 Direction 设置的 4 个选项描述

设置	描述
双向 (三态) 端口 [Bidirectional (tri-state) Ports]	该模式下，每个 PIO 位共用一个设备引脚来驱动和捕获数据。每个引脚方向可单独选择。若要使 FPGA 的 I/O 引脚为三态，则把 I/O 设置为输入状态。可选择输入时是否产生中断。
仅为输入端口 (Input Ports Only)	在该模式下，PIO 端口仅为输入状态，可选是否产生中断。
仅为输出端口 (Output Ports Only)	在该模式下，PIO 端口仅为输出状态，不能产生中断。
输入和输出端口 (Both Input and Output Ports)	在该模式下，输入和输出端口总线独立，相当于同时生成“仅为输入端口”+“仅为输出端口”。

(2) 输出寄存器 (Ouput Register)

这个选项就是设置我们所说的置位寄存器和清零寄存器。在 Direction 为 Input Ports Only 时，输出寄存器是不可以使用的。

(3) 边沿捕获寄存器 (Ouput Register)

当我们选中 Input Options 选项时，PIO IP 核便允许我们设定边沿捕获和产生 IRQ。当我们选择 Output Ports 选项时，边沿捕获和产生 IRQ 选项是不可使用的。当我们 Synchronously capture 选项选中时，PIO IP 核包含边沿捕获寄存器 (Edgecapture)。我们必须进一步指定要检测哪种边沿类型：

- Rising Edge 上升沿；
- Falling Edge 下降沿；
- Any Edge 上升或下降沿。

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。这里需要注意，若是在 Qsys 中选择了 enable bit-clearing for edge capture register 的话，那么对于 edge capture 就应该是写 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。

(4) 中断 (Interrupt)

当 Generate IRQ 选项选中时，若指定的事件出现在输入端口，则 PIO IP 核申请 IRQ 中断。用户必须进一步指定 IRQ 事件产生的原因：

- Level 只要输入为高电平且中断使能，那么 PIO IP 核产生一个 IRQ。
- Edge 只要边沿捕获寄存器中相应位为 1 且中断使能，那么 PIO IP 核产生一个 IRQ。

如果希望低电平时中断，则需在该 I/O 输入引脚前加一个非门（使用片内逻辑最方便）。当 Generate IRQ 选项未选中时，中断屏蔽寄存器不存在。当硬件配置为电平触发方式时，只要高电平出现并且中断使能，就申请一个中断。当硬件配置为边沿触发方式时，只要捕获到边沿事件并且中断使能时，就申请一个中断。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。每个 PIO IP 核的 I/O

口共用一个中断号（系统生成时指定），用户需要在中断服务子程序中通过中断掩码的方式来查明是哪个 I/O 产生了中断。

（5）仿真（Test bench wiring）

当我们需要进行仿真时，我们便可以在这里设置，由于仿真很少使用，所以我们这里就不在进一步进行讲解。

4.1.4 PIO IP 核的软件编程

介绍完了 PIO IP 核的配置选项，接下来我们再来看看 PIO IP 核的软件编程，PIO IP 核提供了对硬件进行寄存器级访问的文件 `altera_avalon_pio_regs.h`。该文件定义了内核的寄存器映射并提供硬件设备访问宏定义。我们可以通过阅读上述文件来熟悉 PIO IP 核的软件访问方法，我们这里建议大家不要修改该文件。

4.1.5 PIO IP 核的应用实例——流水

介绍完了 PIO IP 核的软件编程，接下来我们再来看看 PIO IP 核的应用实例，对于应用实例的讲解，我们这里需要说明的是，我们将会从以下四个方面进行讲解：

- （1）实验目的：该方面主要是用来讲解，我们的实验将要完成一个怎样的功能，我们从该实验中将会学到什么知识。
- （2）硬件框架：该方面主要是用来讲解，如何构建一个 Qsys 系统，以及我们构建该实验都用到了哪些 IP 核。
- （3）软件工程：该方面主要是用来讲解，如何通过 C 语言程序来完成我们的设计，实现我们的功能。
- （4）板级调试：该方面主要是用来验证，我们的实验是否正确，以及能否在锆石 A4 开发板上正常运行。

（1）实验目的

首先我们讲解的是第一部分实验目的，该实验主要是利用 PIO IP 核来控制 8 个 LED 进行流水灯显示，在这个实验中，我们将学习到 PIO IP 核在 Qsys 软件中是如何进行配置的，以及我们如何利用 C 语言程序来通过 PIO 输出数据。

（2）硬件框架

由于之前我们已经给出了详细的工程建立步骤，那么这里我们就不再给出大量的重复步骤，大家只需要根据我们给出的关键性步骤，便能创建出可以用于实验的工程。如果此时你对 Qsys 软件创建流程还不是很熟悉，那么你可以参考该篇第 2 章中介绍的内容。下面我们便给出该实验的硬件框架，如图 4.3 所示。

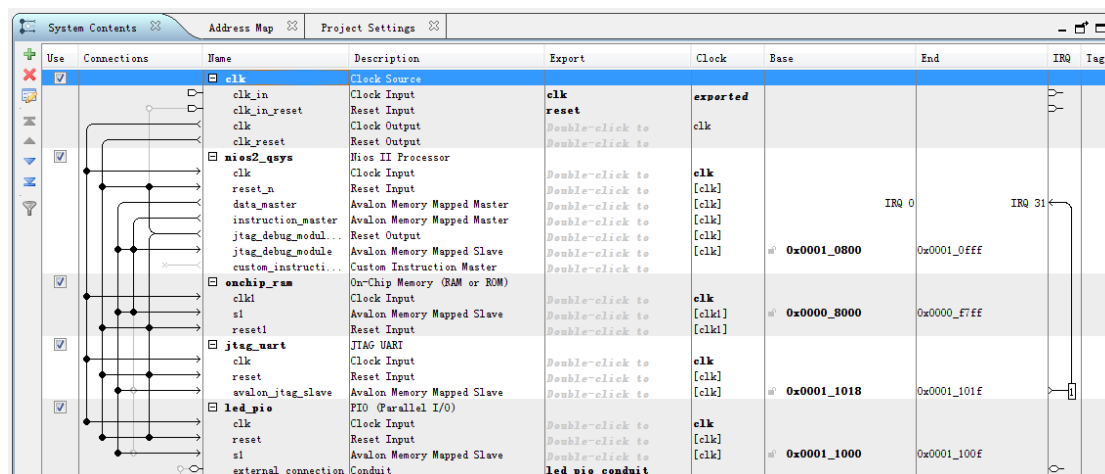


图 4.3 LED 流水灯实验的硬件框架图

从该图中我们可以看出，我们使用了 Nios、Onchip_Memory、Jtag_Uart 和 PIO 四个 IP 核，对于这四个 IP 核，相信大家都不会陌生，我们在之前工程中都有用过，这里我们就不再给出它们的详细配置图。下面我们就对它们的配置进行一个简单的讲解，在这四个 IP 核中，我们 Nios 配置使用的是 Nios II/f 等级；onchip_memory 配置使用的是单端口 RAM，RAM 的大小为 30KB。jtag_uart 配置使用的是默认配置；pio 配置使用的是输出端口，数据宽度为 8。这里要注意的是，由于我们这里没有用到 onchip_rom，只用了一个 onchip_ram，所以读者在设置 Reset Vector 和 Exception Vector 的时候全都选择 onchip_ram 即可。

(3) 软件工程

讲完了硬件框架，接下来我们就来讲解软件工程，该实验的软件工程代码，如代码 4.1 所示。

代码 4.1 Qsys_Led.c 代码

```

1  //-----
2  //-- 文件名    : Qsys_Led.c
3  //-- 描述      : 通过 PIO 直接控制 8 个 Led 产生流水灯效果
4  //-- 修订历史  : 2014-1-1
5  //-- 作者      : Zircon Opto-Electronic Technology CO.,Ltd.
6  //-----
7  #include "system.h"           //系统头文件
8  #include "unistd.h"           //延迟函数头文件
9  #include "alt_types.h"        //数据类型头文件
10 #include "altera_avalon_pio_regs.h" //pio 寄存器头文件
11
12 /* 流水灯花样，Led 低电平亮，高电平灭 */
13 alt_u32 LED_TBL[] = {
14     0xFF, 0x00,                // 全部熄灭后，再全部点亮
15     0xFE, 0xFD, 0xFB, 0xF7, 0xEF, 0xDF, 0xBF, 0x7F, // 依次逐个点亮
16     0xFE, 0xFC, 0xF8, 0xF0, 0xE0, 0xC0, 0x80, 0x00, // 依次逐个叠加
17     0x00, 0x80, 0xC0, 0xE0, 0xF0, 0xF8, 0xFC, 0xFE, // 依次逐个递减
18     0x7E, 0xBD, 0xDB, 0xE7, 0xE7, 0xDB, 0xBD, 0x7E, // 两个靠拢后分开
19     0x7E, 0x3C, 0x18, 0x00, 0x00, 0x18, 0x3C, 0x7E // 从两边叠加后递减

```

```

20 };
21
22 //-----
23 //-- 名称      : main()
24 //-- 功能      : 程序入口
25 //-- 输入参数  : 无
26 //-- 输出参数  : 无
27 //-----
28 int main (void)
29 {
30     alt_u8 i;
31
32     while(1)
33     {
34         for(i=0; i<=41; i++)
35         {
36             //将 LED_TBL 数组中的数值依次赋值给 Led 进行显示
37             IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, LED_TBL[i]);
38             usleep(500000); //等待 100ms
39         }
40     }
41     return 0;
42 }

```

由于该代码比较简单，并且我们也给出了详细的注释，所以这里我们就不再一句一句的讲解代码了，我们下面主要为大家讲解代码中的一些关键部分。

- (1) 为什么要包含 system.h 文件？这主要是因为 system.h 头文件是系统硬件信息的宏定义文件，程序中 LED_PIO_BASE 就是从该文件中获取的。
- (2) altera_avalon_pio_regs.h 寄存器头文件一般有什么用？altera_avalon_pio_regs.h 头文件提供 PIO 内核寄存器访问宏定义，该文件路径为 C:\altera\13.1\ip\altera\sopc_builder_ip\altera_avalon_pio\inc\ altera_avalon_pio_regs.h。程序中对 I/O 口操作的宏定义都在这个文件中给出。
- (3) altera_types.h 头文件的作用是什么？altera_types.h 头文件定义了与 Nios II 相关的数据类型，它的路径为 C:\altera\13.1\nios-2eds\components\altera_nios2\HAL\inc\alt_types.h。

(4) 板级调试

讲完了软件工程，接下来我们就将该实验下载至我们的 A4 开发板进行验证，首先我们需要在 Quartus II 软件中将 Qsys_Led.sof 下载至我们的 A4 开发板，Qsys_Led.sof 下载完成后，我们还需要在 Eclipse 软件中将 Qsys_Led.elf 文件下载至我们的 A4 开发板，Qsys_Led.elf 下载完成以后，我们的 C 程序将会执行在我们的 A4 开发板上，我们可以看到 A4 开发板上的 Led 全部熄灭后，再全部点亮，依次逐个点亮，依次逐个叠加，依次逐个递减，两个靠拢后分开，从

两边叠加后递减等等各种流水花样。由于我们的图片不能显示动态效果，所以我们这里就给出一个 LED 全部点亮的图片，如图 4.4 所示。

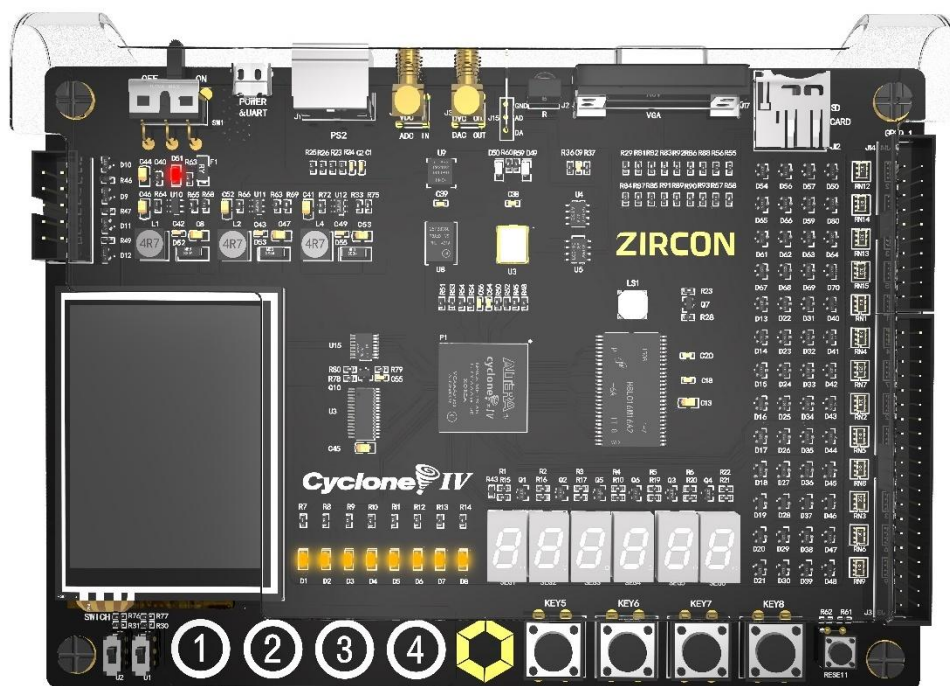


图 4.4 LED 流水灯实验的板级调试图

4.1.6 PIO IP 核的应用实例——按键

(1) 实验目的

介绍完了 PIO 流水实验，接下来我们再来看看 PIO 按键实验，对于 PIO 按键实验，我们将和前面一样，也是从四个方面进行讲解。首先我们讲解的是第一个方面实验目的。该实验主要是利用 PIO IP 核读取按键的值，然后将读到的值输出到 LED 上进行显示。在这个实验中，我们将学习到 PIO IP 核在 Qsys 软件中是如何进行配置，以及我们如何利用 C 语言程序来通过 PIO 读取数据。

(2) 硬件框架

讲完了实验目的，接下来我们就来讲解硬件框架，该实验的硬件框架，如图 4.5 所示。

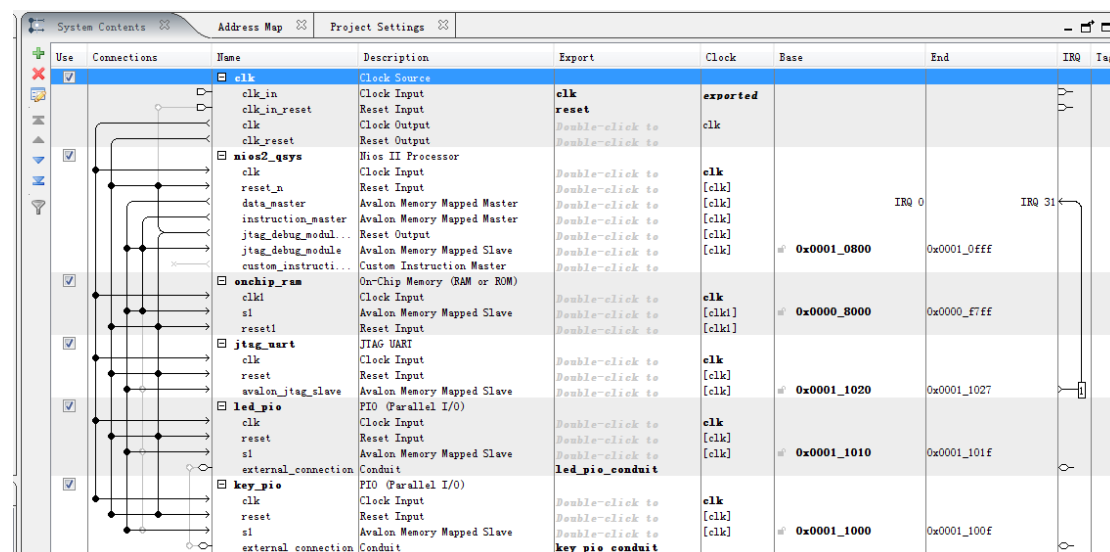


图 4.5 按键实验的硬件框架图

从该图中我们可以看出，我们的 PIO 按键实验在 PIO 流水实验的基础上多添加了一个 PIO IP 核，该 PIO 的配置我们使用的是输入端口，数据宽度为 8，至于其他的 IP 核配置，我们这里和 PIO 流水实验中的配置是一样的，没有做任何改动。这里需要注意的是，我们的按键没有连接内部中断。

(3) 软件工程

讲完了硬件框架，接下来我们就来讲解软件工程，该实验的软件工程代码，如代码 4.2 所示。

代码 4.2 Qsys_Key.c 代码

```

1  //-----
2  //-- 文件名    : Qsys_Key.c
3  //-- 描述      : 按下不同的按键，点亮相对应的 Led。
4  //-- 修订历史  : 2014-1-1
5  //-- 作者      : Zircon Opto-Electronic Technology CO.,Ltd.
6  //-----
7  #include "system.h"           //系统头文件
8  #include "alt_types.h"        //数据类型头文件
9  #include "altera_avalon_pio_regs.h" //pio 寄存器头文件
10
11 //-----
12 //-- 名称      : main()
13 //-- 功能      : 程序入口
14 //-- 输入参数  : 无
15 //-- 输出参数  : 无
16 //-----
17 int main(void)
18 {
19     alt_u32 key_state, led_state; //Key 和 Led 缓存变量
20

```

```

21  while(1)
22  {
23      //读取按键的值，并赋值给 key_state。
24      key_state = IORD_ALTERA_AVALON_PIO_DATA(KEY_PIO_BASE);
25      //注意：Key 悬空为低电平，按下为高电平。Led 低电平亮，高电平灭。
26      //为了保证按下不同按键点亮相对应 Led，因此，我们将按键的值取反后，再赋值给 led_state。
27      led_state = ~key_state;
28      //将 led_state 中的值输出到 Led 上进行显示。
29      IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, led_state);
30  }
31
32  return(0);
33  }

```

(4) 板级调试

讲完了软件工程，接下来我们就将该实验下载至我们的 A4 开发板进行验证，首先我们需要在 Quartus II 软件中将 Qsys_Key.sof 下载至我们的 A4 开发板，Qsys_Key.sof 下载完成后，我们还需要在 Eclipse 软件中将 Qsys_Key.elf 文件下载至我们的 A4 开发板，Qsys_Key.elf 下载完成以后，我们的 C 程序将会执行在我们的 A4 开发板上，我们可以看到 A4 开发板上的 Led 全部熄灭了，当我们按下 KEY1 时，D1 就会被点亮，如图 4.6 所示。

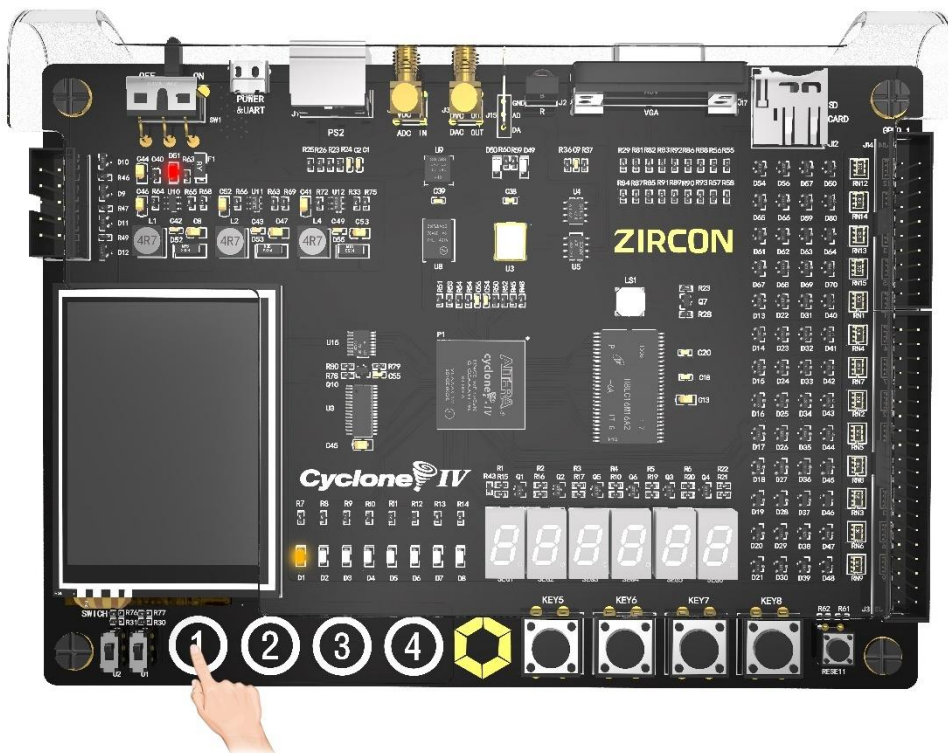


图 4.6 按键实验的板级调试图

从该图中可以看出，我们的按键实验程序是正确的，我们在 A4 开发板上实现了按下不同的按键，点亮相对应的 Led 功能。

4.1.7 PIO IP 核的应用实例——中断

(1) 实验目的

介绍完了 PIO 按键实验，接下来我们再来看看 PIO 中断实验，该实验主要是利用 PIO IP 核内部中断来实现控制 LED 灯亮灭。中断信号由按键提供，产生中断后，在中断事件中将读取按键的值并将值输出到 LED 上进行显示。在这个实验中，我们将学习到 PIO IP 核中断在 Qsys 软件中是如何进行配置，以及中断服务程序的编写、注册和调试方法。

(2) 硬件框架

讲完了实验目的，接下来我们就来讲解硬件框架，该实验的硬件框架，如图 4.7 所示。

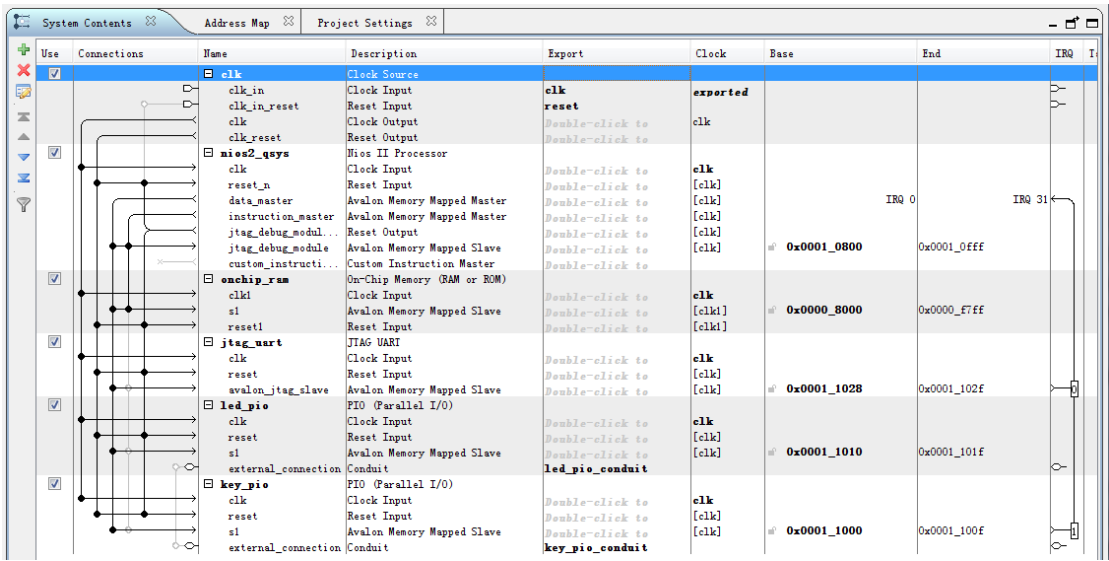


图 4.7 中断实验的硬件框架图

从该图中我们可以看出，我们 PIO 中断实验与我们 PIO 按键实验的硬件框架结构是一样的，没有添加任何组件，唯一不同的是，我们改动了 key_pio 的配置，我们为 key_pio 添加了中断选项，它的配置如图 4.8 所示。

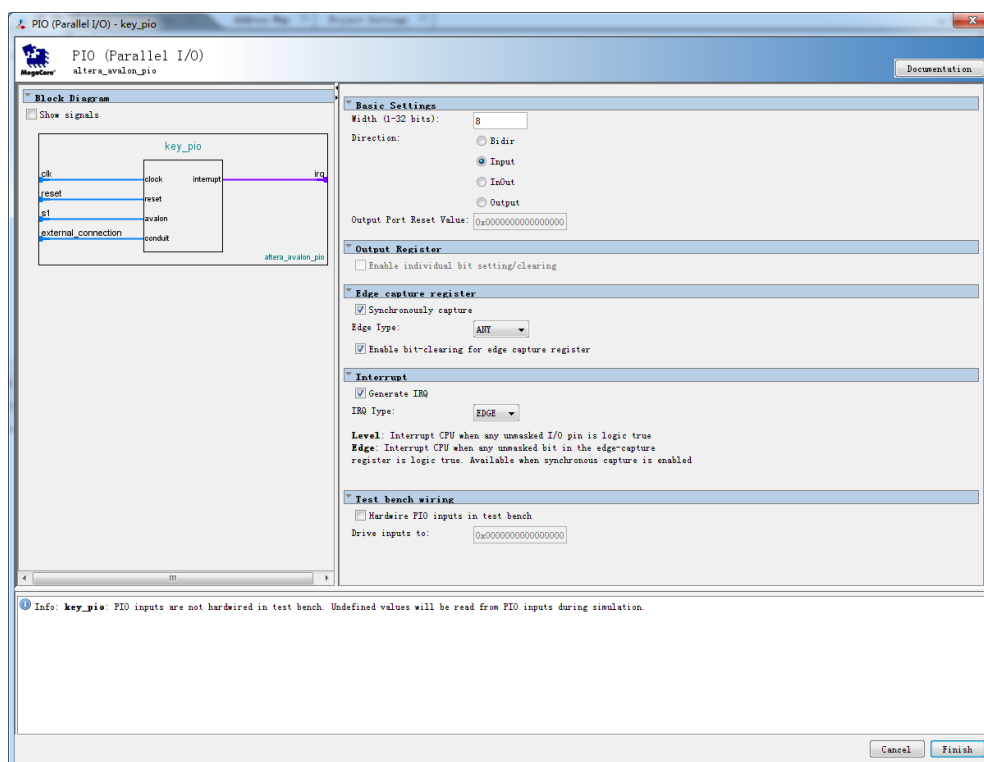


图 4.8 中断实验的 PIO 配置页面

从该图中可以看出，我们选择的是边沿上升沿触发或边沿下降沿触发中断。这里我们需要注意的是，我们还选择了 Enable bit-clearing for edge capture register 这个选项，如果大家还记得我们之前讲过的内容，那么大家肯定知道这个选项是用来干什么的，如果我们选中了该选项，那么对于 edge capture 就应该是写 1 清中断；如果我们没有选择 enable bit-clearing for edge capture register，则是写任意数清中断。

(3) 软件工程

讲完了硬件框架，接下来我们就来讲解软件工程，该实验的软件工程代码，如代码 4.3 所示。

代码 4.3 Qsys_Key_Interrupt.c 代码

```

1  //-----
2  //-- 文件名    : Qsys_Key_Interrupt.c
3  //-- 描述     : 按下不同的按键，点亮相对应的 Led。
4  //-- 修订历史  : 2014-1-1
5  //-- 作者     : Zircon Opto-Electronic Technology CO.,Ltd.
6  //-----
7  #include "system.h"           //系统头文件
8  #include "altera_avalon_pio_regs.h" //pio 寄存器头文件
9  #include "sys/alt_irq.h"      //中断头文件
10 #include "unistd.h"           //延迟头文件
11
12 void IRQ_Init();              //中断初始化函数
13 void IRQ_Key_Interrupts();    //中断服务子程序
14

```

```

15 //-----
16 //-- 名称      :  main()
17 //-- 功能      :  程序入口
18 //-- 输入参数  :  无
19 //-- 输出参数  :  无
20 //-----
21 int main()
22 {
23     IRQ_Init(); // 初始化 PIO 中断
24     IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, 0xFF); //初始化 LED 全灭
25
26     while(1)
27     {
28         usleep(1000);
29     }
30 }
31
32 //-----
33 //-- 名称      :  IRQ_Init()
34 //-- 功能      :  中断初始化函数
35 //-- 输入参数  :  无
36 //-- 输出参数  :  无
37 //-----
38 void IRQ_Init()
39 {
40     IOWR_ALTERA_AVALON_PIO_IRQ_MASK(KEY_PIO_BASE, 0xff); // 使能中断
41     IOWR_ALTERA_AVALON_PIO_EDGE_CAP(KEY_PIO_BASE, 0xff); // 清中断边沿捕获寄存器
42     // 注册 ISR
43     alt_ic_isr_register(
44         KEY_PIO_IRQ_INTERRUPT_CONTROLLER_ID, // 中断控制器标号, 从 system.h 复制
45         KEY_PIO_IRQ,                          // 硬件中断号, 从 system.h 复制
46         IRQ_Key_Interrupts,                    // 中断服务子函数
47         0x0,                                  // 指向与设备驱动实例相关的数据结构体
48         0x0);                                // flags, 保留未用
49 }
50
51 //-----
52 //-- 名称      :  IRQ_Key_Interrupts()
53 //-- 功能      :  中断服务子程序
54 //-- 输入参数  :  无
55 //-- 输出参数  :  无
56 //-----
57 void IRQ_Key_Interrupts()
58 {
59

```

```

60  alt_u32 key_state, led_state; //Key 和 Led 缓存变量
61  //读取按键的值，并赋值给 key_state。
62  key_state = IORD_ALTERA_AVALON_PIO_DATA(KEY_PIO_BASE);
63  //注意：Key 悬空为低电平，按下为高电平。Led 低电平亮，高电平灭。
64  //为了保证按下不同的按键点亮相对应 Led，因此，我们将按键的值取反后，再赋值给 led_state。
65  led_state = ~key_state;
66  //将 led_state 中的值输出到 Led 上进行显示。
67  IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, led_state);
68  IOWR_ALTERA_AVALON_PIO_EDGE_CAP(KEY_PIO_BASE, 0xff); //清中断边沿捕获寄存器
69  }

```

下面我们就来给大家讲解一下该程序中的关键点。通过前面的学习我们知道，当中断来了以后，我们的程序会跳转到异常处理地址，这时，Nios II 处理器开始执行一段硬件抽象层（HAL）插入的代码，并判断中断源和中断优先级，然后再跳转到我们编写的中断服务子程序（ISR）中，整个过程不需要我们任何操作，我们只需要将中断服务子程序的信息告知给硬件抽象层就可以了，具体如何将中断服务子程序的信息告知给硬件抽象层，我们需要完成以下两个步骤：

- 参照 void isr_name(void *context, alt_u32 id) 函数原型编写中断服务子程序。其中 isr_name 是用户给 ISR 取的函数名；void *context 是指向传递给 ISR 的信息的全局变量指针（通常是寄存器信息）；id 是硬件中断号，在 system.h 中声明，比如我们这里的 KEY_PIO_IRQ。
- 调用 alt_irq_register(alt_u32 id, void *context, void (*isr)(void *, alt_u32)) 或者是 alt_ic_isr_register(alt_u32 ic_id, alt_u32 irq, alt_isr_func isr, void *isr_context, void *flags); 函数向硬件抽象层登记中断服务子程序。其中 alt_u32 id 和 alt_u32 irq 是硬件中断号；void *context 和 void *isr_context 是指向传递给 ISR 的信息的全局变量指针（通常是寄存器信息）；void (*isr)(void *, alt_u32) 和 alt_isr_func isr 是指向 ISR 的函数指针；alt_u32 ic_id 是中断控制器标号，*flags 保留未用。

(4) 板级调试

讲完了软件工程，接下来我们就将该实验下载至我们的 A4 开发板进行验证，首先我们需要在 Quartus II 软件中将 Qsys_Key_Interrupt.sof 下载至我们的 A4 开发板，Qsys_Key_Interrupt.sof 下载完成后，我们还需要在 Eclipse 软件中将 Qsys_Key_Interrupt.elf 文件下载至我们的 A4 开发板，Qsys_Key_Interrupt.elf 下载完成以后，我们的 C 程序将会执行在我们的 A4 开发板上，我们可以看到 A4 开发板上的 Led 全部熄灭了，当我们按下 KEY3 时，D3 就会被点亮，如图 4.9 所示。

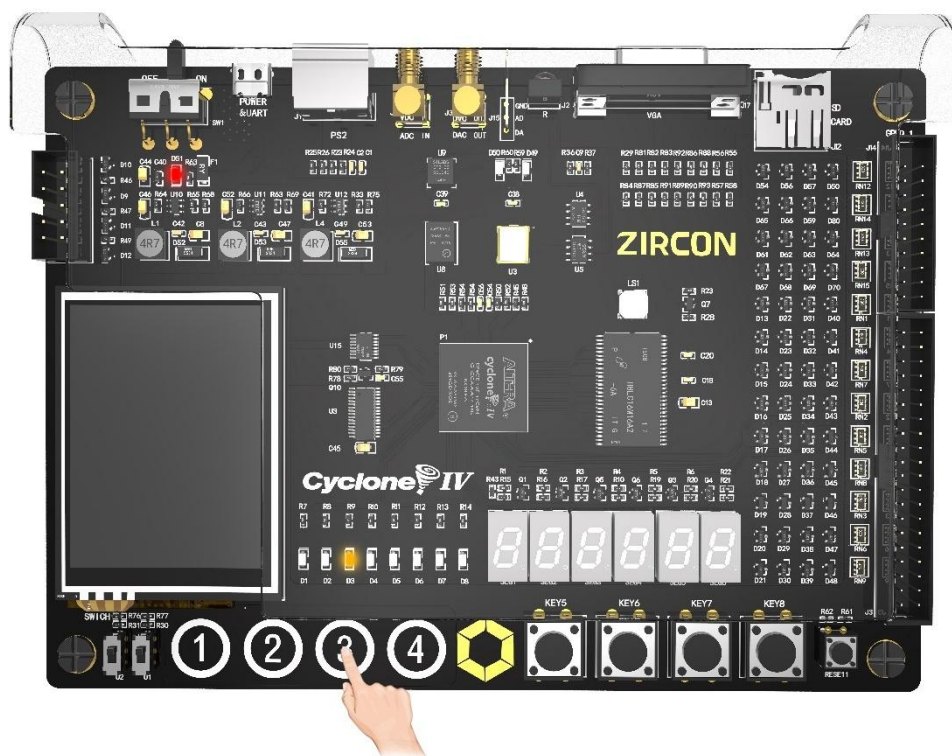


图 4.9 中断实验的板级调试图

从该图中可以看出，我们的中断实验程序是正确的，我们在 A4 开发板上利用中断实现了按下不同的按键，点亮相对应的 Led 功能。

§ 4.2 SDRAM IP核

4.2.1 SDRAM IP 核的综述

介绍完了 PIO IP 核，接下来我们就来看看我们的 SDRAM IP 核，同 PIO IP 核一样，首先我们介绍的是 SDRAM IP 的综述。学过《数字电路篇》朋友肯定都知道，SDRAM 常用于需要大量易失性存储器且成本要求高的应用系统。虽然 SDRAM 比较便宜，但是 SDRAM 应用需要实现刷新操作、行列管理、不同延迟和命令序列等逻辑。SDRAM IP 核提供了连接一个或多个 SDRAM 芯片的接口，并处理所有 SDRAM 协议要求。有了 SDRAM IP 核，在 Nios II 系统中使用 SDRAM 就像使用 SRAM 一样简单。为了让读者能够更易理解，以及有更直观的感受，下面我们就给出 SDRAM IP 核连接到外部 SDRAM 芯片的一个结构框图，如图 4.10 所示。

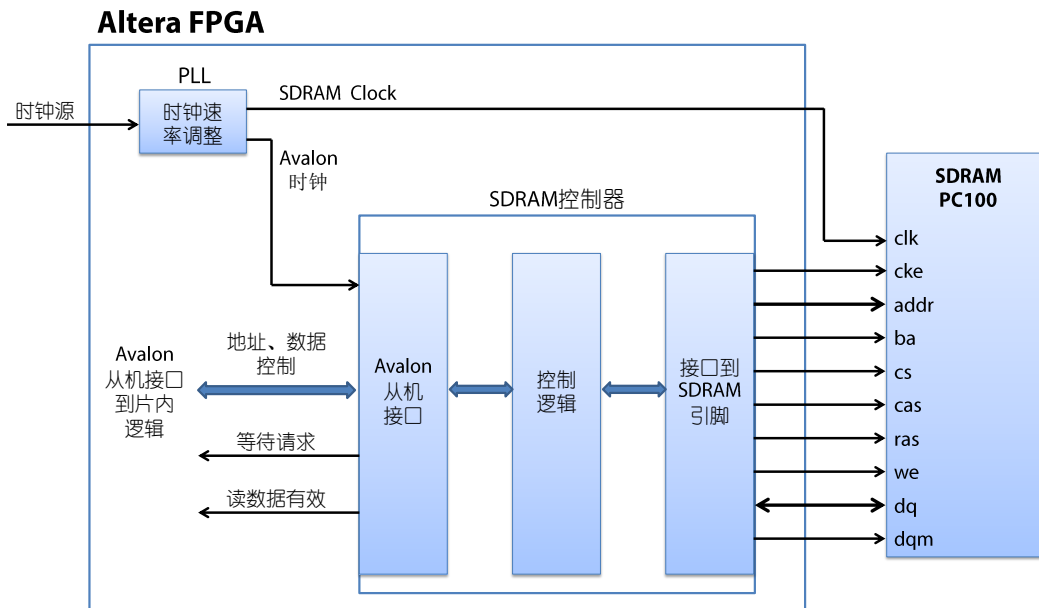


图 4.10 SDRAM IP 核连接到外部 SDRAM 芯片的结构框图

我们可以从上面的这个结构框图中看出，这个 SDRAM 控制器，它其实是在我们 Altera FPGA 中产生的，它本身包含有接口引脚、控制逻辑、以及 Avalon 从机接口。接口引脚是用来连接我们外部 SDRAM 芯片管脚的，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。控制逻辑就是用来实现 SDRAM 操作的，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 IP 核之后，它会自动生成。Avalon 从机接口是用来连接我们的 CPU 的，Avalon 从机接口是 SDRAM 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 Avalone 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 不是包括在 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的 PLL 设置。

还有一点我们需要说明的是 fmax 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 控制器与 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 fmax 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 IP 核的功能特性：

- (1) SDRAM 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 IP 核可以全面支持符合 PC100 标准的 SDRAM 芯片。

- (3) SDRAM 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。

4.2.2 SDRAM IP 核的配置选项

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。SDRAM IP 核配置向导有两个选项卡: Memory Profile 和 Timing, 如图 4.11 所示。

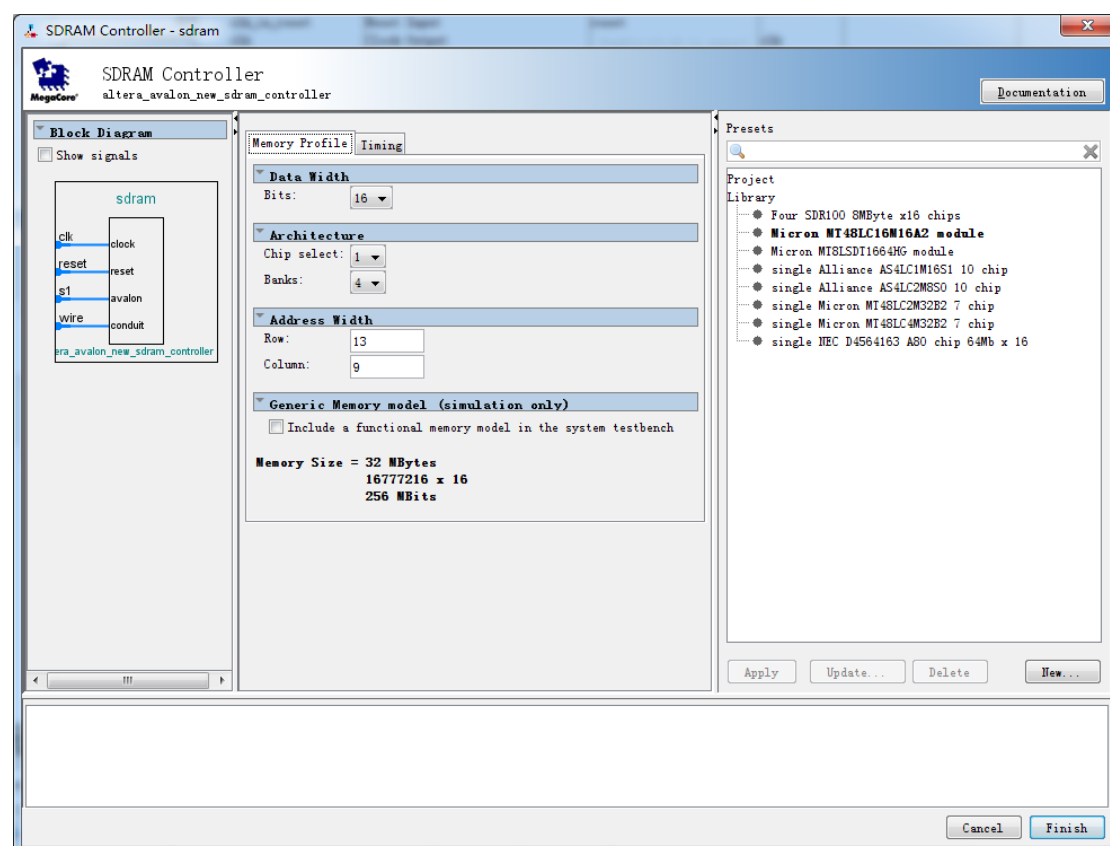


图 4.11 SDRAM 的配置选项

为了方便, Presets 下拉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, 例如地址和数据线宽度、片选信号的数据和区的数目等。表 4.4 列出 Memory Profile 选项卡可用的设置。

表 4.4 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 总线（数据）和 dqm 总线（字节使能）的宽度。具体数值请查阅 SDRAM 数据手册。
结构设置	片选	1,2,4,8	1	独立芯片的数目在 SDRAM 子系统中选择。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。
	区	2、4	4	区的数目。该值确定连接到 SDRAM 的 ba 总线（区地址）宽度。具体数值请查阅 SDRAM 数据手册。
地址宽度设计	行	11,12,13,14	12	行地址位的数目。该值确定 addr 总线的宽度。具体数值请查阅 SDRAM 数据手册。
	列	≥ 8 且小于行的值	8	列地址位的数目。例如，SDRAM 排列为 4096 行、512 (29) 列，所以列的值为 9。具体数值请查阅 SDRAM 数据手册。
控制器共用 dq/dqm/addr I/O 引脚		是、否	否	当设为 No 时,所有引脚都专用与 SDRAM 芯片。当设为 Yes 时，addr、dq 和 dqm 引脚在系统内可与三态桥共享
包括系统测试台的功能存储器模块		是、否	是	当打开选项时，Qsys 创建 SDRAM 芯片的功能仿真模型，该默认的存储器模型加速创建的过程和校验使用 SDRAM 控制器的系统。

通过 Memory Profile 选项卡上的设置，消息框显示 SDRAM 期望的内存容量，以兆字节、兆位以及可寻址的字数位单位。将这些期望值与选择的 SDRAM 的实际大小相比较可以检验设置是否正确。这里我们需要注意的是，我们在 SDRAM 的配置选项图中是看不到共用引脚这个选项的，这个共用引脚在老的 SOPC 版本是有的，不过在我们新的 Qsys 版本中给去掉了。这个功能我们只了解即可。说完了 Memory Profile 选项卡，接下来我们再来看一下 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。表 4.5 列出 Timing 选项卡上可用的设置。

表 4.5 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	从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	—	15.625 μ s	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 64ms/4096=15.625 μ s 执行一个刷新命令来符合这个要求
在初始化前、上电后延时	—	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	—	70 ns	自动刷新周期

预充电命令(t _{rp}) 的持续时间	—	20 ns	预充电命令周期
Active 到 Read 或 Write 延时(t _{rcd})	—	20 ns	ACTIVE 到 READ 或 WRITE 延时
访问时间 (t _{ac})	—	5.5 ns	时钟边沿的访问时间。该值由 CAS 的等待时间决定
写恢复时间(t _{wr} , 无自动预充电)	—	14 ns	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

无论用户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 (Issue One Refresh Command Every) 的参数, 实际时序将不超出目标值。对于其他所有参数, 实际时序将大于或等于目标值。一般情况下, 我们使用默认值就可以了。

4.2.3 SDRAM IP 核的软件编程

当通过 Avalon 接口访问时, SDRAM IP 核操作起来像简单的 SRAM 存储器, 没有可配置的软件设置, 没有存储器映射的寄存器。处理器访问 SDRAM IP 核不需要软件驱动程序。

4.2.4 SDRAM IP 核的应用实例

(1) 实验目的

介绍完了 SDRAM IP 核的软件编程, 接下来我们再来看看 SDRAM IP 核的应用实例, 该实验主要是将应用程序下载到 SDRAM 中运行, 并且将递增的数据通过 LED 进行显示, 通过 JTAG_UART 进行输出。在这个实验中, 我们将学习到 SDRAM IP 核在 Qsys 软件中是如何进行配置的, 以及我们如何利用 C 语言程序来访问 SDRAM 存储器。

(2) 硬件框架

讲完了实验目的, 接下来我们就来讲解硬件框架, 该实验的硬件框架, 如图 4.12 所示。

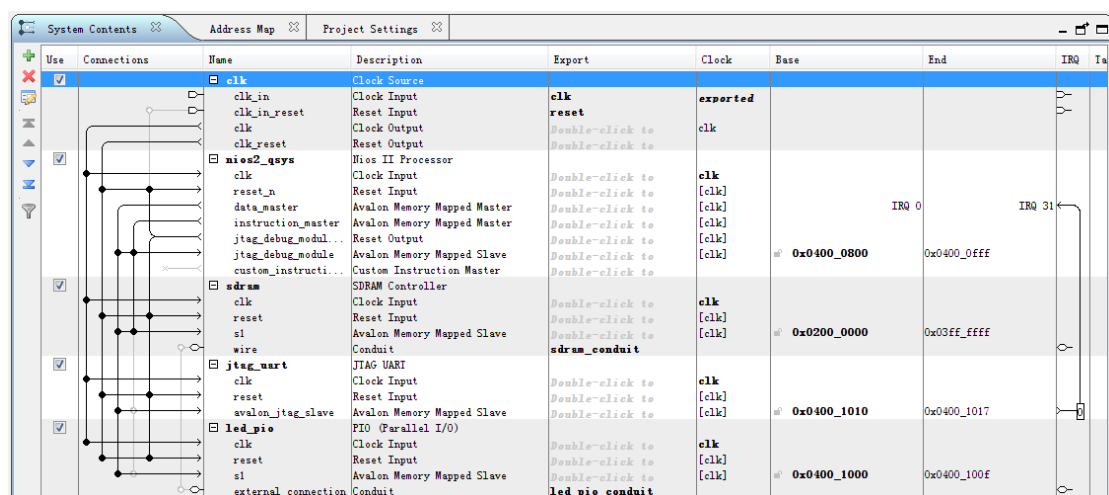


图 4.12 SDRAM 实验的硬件框架图

从该图中我们可以看出, 我们将之前的 onchip_memory IP 核修改成了我们的 SDRAM IP 核, 对于 Nios、Jtag_Uart 和 PIO 这三个 IP 核, 它们是我们的老朋友了, 我们相信大家都能够

理解了，这里我们就不再对它们进行介绍了，我们主要来介绍一下新添加的 SDRAM IP 核，它的配置如图 4.13 所示。

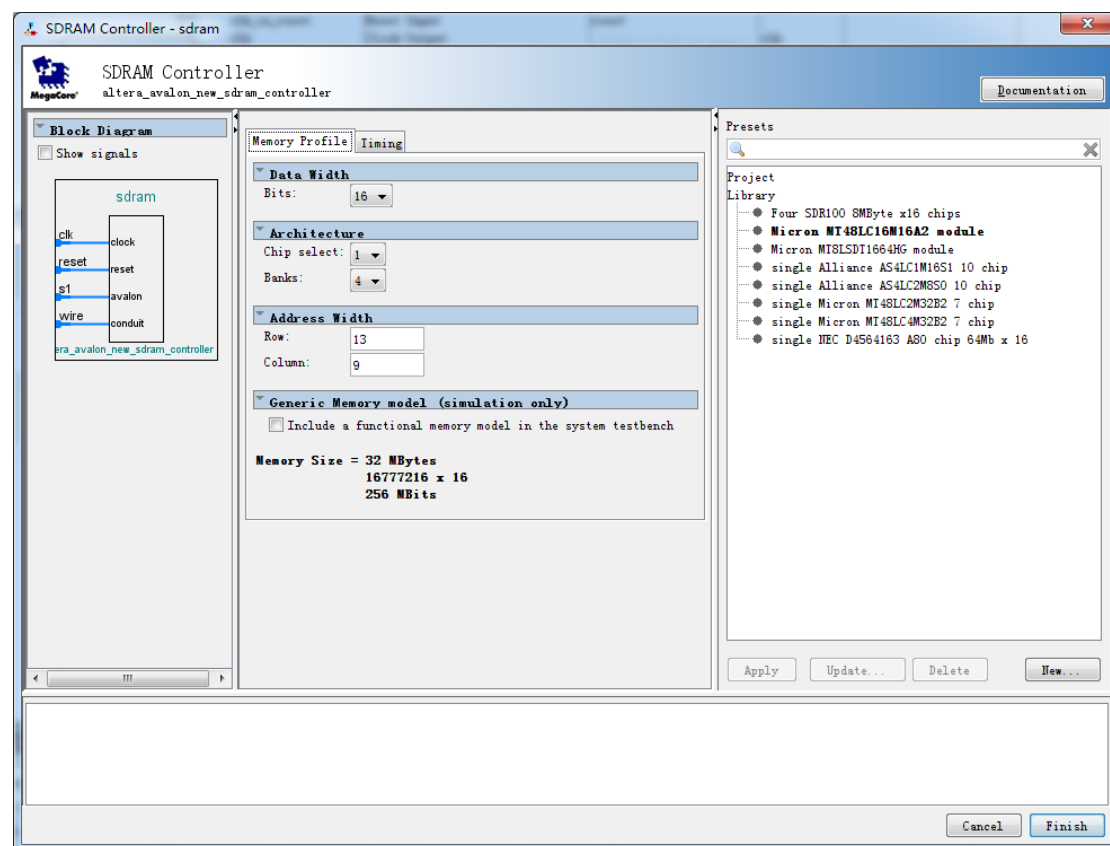


图 4.13 SDRAM IP 核的配置页面

从该图中可以看出，我们将 SDRAM 配置成了 16 位，行是 13，列是 9。关于 SDRAM 的配置，我们需要参考 SDRAM 的数据手册，不同型号的 SDRAM，它的配置也是不同的。由于我们这里的 SDRAM 型号是 MT48LC16M16A2，所以我们参考了 MT48LC16M16A2 的 datasheet。这里需要大家注意的是，由于我们将 MT48LC16M16A2 型号的 SDRAM 配置添加到了 Library 中，所以我们只需要选择这个配置即可自动完成所有设置。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，我们就可以找到 Library 窗口中的【New】按钮并点击，弹出如图 4.14 所示页面。

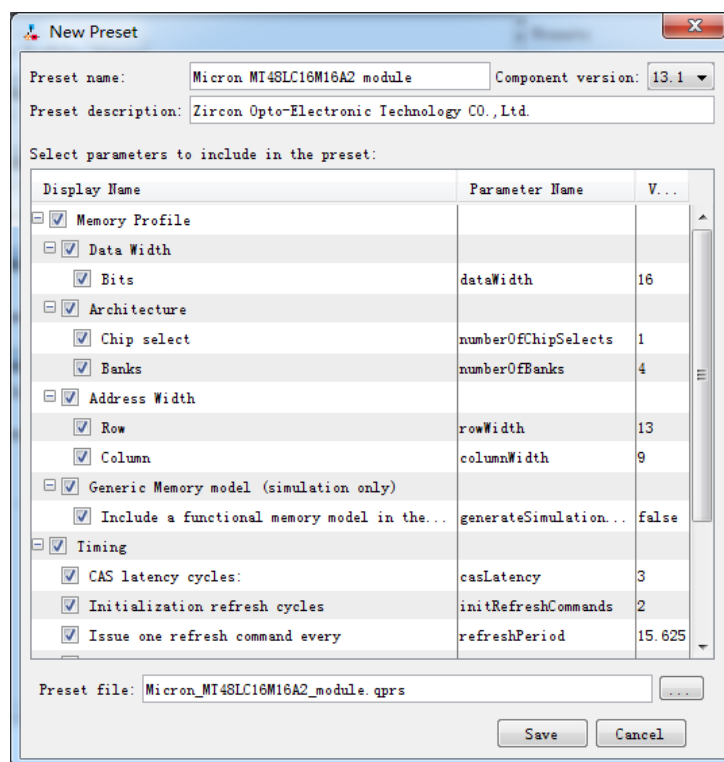


图 4.14 将 SDRAM 配置添加至 Library 中

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,弹出如图 4.15 所示提示窗口。

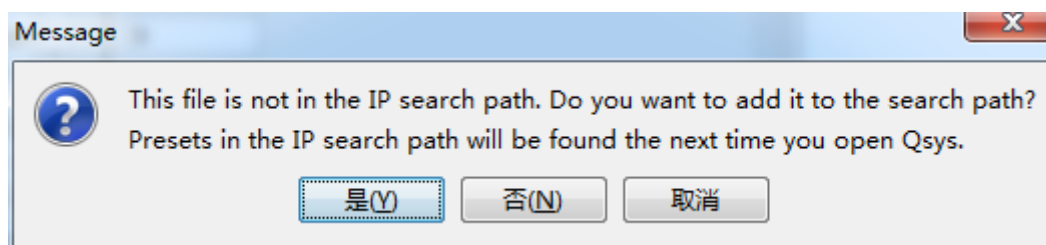


图 4.15 路径添加提示窗口

在该提示窗口中我们选择否,这时我们就可以在 Library 中看到我们添加的 SDRAM 配置了。这时,我们打开该工程文件夹可以看到多出了一个 Micron_MT48LC16M16A2_module.qprs 文件,这个文件就是我们添加的 SDRAM 配置信息,为了便于统一管理,这里我们建议大家将该文件中的内容复制到 C:\altera\13.1\ip\altera\sopc_builder_ip\altera_avalon_new_sdram_controller\altera_avalon_new_sdram_controller.qprs 文件中,当我们复制完毕以后,我们就可以将 Micron_MT48LC16M16A2_module.qprs 文件删掉了,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 Micron_MT48LC16M16A2_module。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,相位偏移我们通常设置为-63deg。

(3) 软件工程

讲完了硬件框架,接下来我们就来讲解软件工程,该实验的软件工程代码,如代码 4.4 所示。

Zircon Opto-Electronic Technology CO.,Ltd.

代码 4.4 Qsys_Sdram.c 代码

```

1  //-----
2  //-- 文件名    : Qsys_Sdram.c
3  //-- 描述      : 打印 00-FF, 并将 00-FF 的值显示到 Led 上
4  //-- 修订历史  : 2014-1-1
5  //-- 作者      : Zircon Opto-Electronic Technology CO.,Ltd.
6  //-----
7  #include <stdio.h>                //标准输入输出头文件
8  #include "system.h"              //系统头文件
9  #include "alt_types.h"           //数据类型头文件
10 #include "unistd.h"              //延迟函数头文件
11 #include "altera_avalon_pio_regs.h" //pio 寄存器头文件
12
13 //-----
14 //-- 名称      : main()
15 //-- 功能      : 程序入口
16 //-- 输入参数  : 无
17 //-- 输出参数  : 无
18 //-----
19 int main()
20 {
21     alt_u8 count = 0; //声明变量,并初始化为 0
22
23     while(1)
24     {
25         usleep(100000); //等待 100ms
26         //将 count 的值打印出来,%x 表示输出十六进制,02 表示不足两位,前面补 0 输出
27         printf("%02x,", count);
28         count++; //等价于 count = count + 1;
29         //将 count 变量的值赋值给 Led 并显示
30         IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, count);
31     }
32
33     return 0;
34 }

```

(4) 板级调试

讲完了软件工程,接下来我们就将该实验下载至我们的 A4 开发板进行验证,首先我们需要在 Quartus II 软件中将 Qsys_Sdram.sof 下载至我们的 A4 开发板, Qsys_Sdram.sof 下载完成后, 我们还需要在 Eclipse 软件中将 Qsys_Sdram.elf 文件下载至我们的 A4 开发板, Qsys_Sdram.elf 下载完成以后, 我们的 C 程序将会执行在我们的 A4 开发板上, 我们可以看到 A4 开发板上的 Led 开始有规律的进行闪烁, 与此同时, 我们还可以在 Eclipse 软件的控制台中看到我们的打印信息, 如图 4.16 所示。

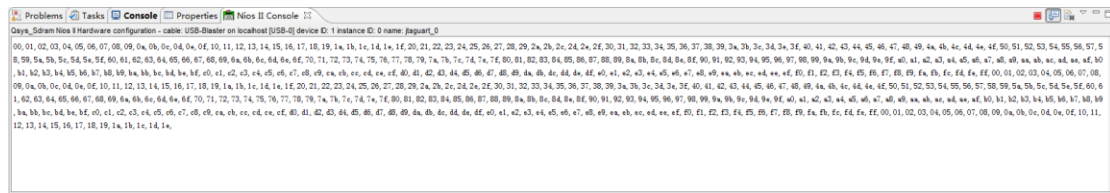


图 4.16 SDRAM 实验的板级调试图

§ 4.3 EPCS IP核

4.3.1 EPCS IP 核的综述

介绍完了我们的 SDRAM IP 核，接下来我们就来介绍一下我们的 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 Altera 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）来读取和编写 EPCS 器件。通过 EPCS IP 核，Nios II 系统可以执行以下操作：

- (1) 在 EPCS 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 EPCS 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。EPCS 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。

为了让读者能够更易理解，以及有更直观的感受，我们这里同 SDRAM IP 核一样，也给出 EPCS IP 核的结构框架图。如图 4.17 所示。

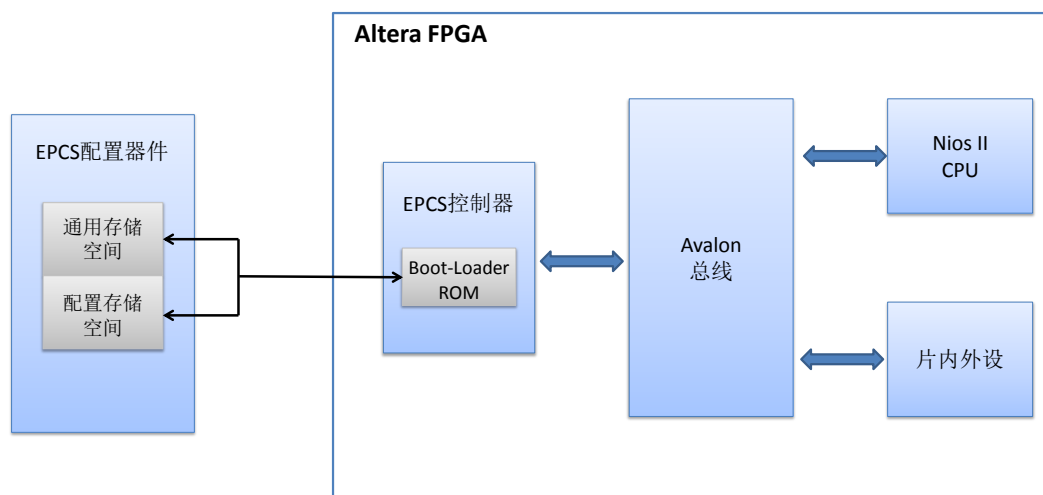


图 4.17 EPCS IP 核的结构框图

我们从 EPCS IP 核结构框图中可以看出，EPCS 器件的存储器被分成了两个独立的区域，一个是 FPGA 配置存储器，它主要是用于存储 FPGA 配置数据的，另一个是通用存储器，如果