



目录

SF-CY3 FPGA 套件开发指南	1
1 概述	7
1.1 功能框图	10
1.2 Cyclone III 系列 FPGA 器件简介	11
1.3 EP3C5E144C8 器件资源一览	12
2 SF-CY3 硬件电路解析	13
2.1 实物示意图	13
2.2 原理图解析	15
2.2.1 系统电源电路	15
2.2.2 FPGA 电源电路	15
2.2.3 时钟和复位电路	17
2.2.4 FPGA 配置电路	18
2.2.5 SDRAM 电路	20
2.2.6 LED 指示灯	21
2.2.7 连接器电路	21
3 SF-CY3 基本使用安装说明	22
3.1 电路板安装	22
3.2 Quartus II 与 ModelSim 软件下载与安装	23
3.2.1 EDA 工具概述	23
3.2.2 软件下载和 license 申请	24
3.2.3 Quartus II 的安装	28
3.2.4 ModelSim 的安装	31
3.3 USB Blaster 驱动安装	33
4 FPGA 的下载配置	35
4.1 FPGA 的上电启动原理	35
4.2 JTAG 在线烧录 FPGA	37
4.3 JTAG 烧录配置芯片	40
5 SF-CY3 工程实例	44
5.1 逻辑 (Verilog) 实例 1——LED 闪烁	44
5.1.1 新建工程	44
5.1.2 输入源码	48
5.1.3 ModelSim 仿真	50
5.1.4 管脚分配与编译	55
5.1.5 下载配置与板级调试	57
5.2 逻辑 (Verilog) 实例 2——PLL 配置	59
5.2.1 新建工程	59
5.2.2 PLL 配置和例化	60
5.2.3 ModelSim 仿真	69
5.2.4 管脚分配与编译	72
5.2.5 下载配置与板级调试	72
5.3 基于 Qsys 的 NIOS II 实例 1——LED 闪烁	73
5.2.1 新建工程	73



5.1.2 Qsys 硬件系统架构.....	74
5.1.3 例化 Qsys 系统.....	85
5.1.4 管脚分配与编译	86
5.1.5 EDS 中新建软件工程	87
5.1.6 ModelSim 仿真.....	97
5.1.7 下载配置与板级调试	101
5.4 基于 Qsys 的 NIOS II 实例 2——Hello NIOS II	104
5.4.1 JTAG UART 外设概述.....	104
5.4.2 编写软件代码	105
5.4.3 下载配置与板级调试	106
5.5 基于 Qsys 的 NIOS II 实例 3——集成 SDRAM 外设	107
5.5.1 系统概述	107
5.5.2 Qsys 组件添加.....	108
5.5.3 系统例化和管脚分配	116
5.5.4 时序约束与工程编译	118
5.5.5 软件工程	127
6 SF-BASE 子板开发指南	128
6.1 功能与原理图介绍	128
6.1.1 主要外设芯片及装配	128
6.1.2 插座管脚定义	129
6.1.3 蜂鸣器电路	130
6.1.4 LED 指示灯电路	130
6.1.5 拨码开关电路	131
6.1.6 数码管电路	132
6.1.7 AD 转换电路.....	133
6.1.8 DA 转换电路.....	134
6.2 逻辑 (Verilog) 实例 3——PWM 驱动蜂鸣器	134
6.2.1 实验原理	134
6.2.2 Verilog 参考代码	135
6.2.3 仿真验证	136
6.2.4 工程实践	137
6.3 逻辑 (Verilog) 实例 4——流水灯.....	139
6.3.1 实验原理	139
6.3.2 Verilog 参考代码	140
6.3.3 仿真验证	140
6.3.4 工程实践	142
6.4 逻辑 (Verilog) 实例 5——模式流水灯.....	144
6.4.1 实验原理	144
6.4.2 Verilog 参考代码	144
6.4.3 仿真验证	145
6.4.4 工程实践	147
6.5 逻辑 (Verilog) 实例 6——数码管显示.....	148
6.5.1 实验原理	148
6.5.2 Verilog 参考代码	149



6.5.3 仿真验证	152
6.5.4 工程实践	152
6.6 逻辑 (Verilog) 实例 7——基于 In-System Sources and Probes Editor 的 AD 采集	153
6.6.1 概述	153
6.6.2 AD 采样控制原理	154
6.6.3 In-System Sources and Probes Editor 例化	155
6.6.4 Verilog 参考代码	159
6.6.5 仿真验证	162
6.6.6 工程实践	162
6.7 逻辑 (Verilog) 实例 8——基于 In-System Sources and Probes Editor 的 DA 输出	166
6.7.1 概述	166
6.7.2 DA 采样控制原理	166
6.7.3 In-System Sources and Probes Editor 例化	167
6.7.4 Verilog 参考代码	168
6.7.5 仿真验证	173
6.7.6 工程实践	173
6.8 基于 Qsys 的 NIOS II 实例 4——PIO 中断控制	174
6.8.1 工程移植	174
6.8.2 添加组件	175
6.8.3 例化系统	179
6.8.4 时序约束	181
6.8.5 软件编程	182
6.9 基于 Qsys 的 NIOS II 实例 5——数码管定时器中断	184
6.9.1 功能概述	184
6.9.2 组件编辑	185
6.9.3 组件添加	188
6.9.4 例化系统	197
6.9.5 软件编程	200
6.10 基于 Qsys 的 NIOS II 实例 6——AD/DA 组件	202
6.9.1 功能概述	202
6.9.2 组件编辑	203
6.9.3 组件添加	210
6.9.4 例化系统	217
6.9.5 软件编程	219
7 SF-LCD 子板开发指南	221
7.1 功能与原理图介绍	221
7.1.1 主要外设芯片及电路图解析	221
7.1.2 装配示意图	227
7.2 逻辑 (Verilog) 实例 9——LCD 的基本驱动	232
7.2.1 LCD 驱动原理	232
7.2.2 Verilog 代码	235
7.2.3 工程实践	238
7.3 逻辑 (Verilog) 实例 10——LCD 的 32 级红色显示	240
7.3.1 色彩显示原理	240



7.3.2 Verilog 代码	241
7.3.3 工程实践	244
7.4 逻辑 (Verilog) 实例 11——基于 FPGA 内嵌 RAM 的 LCD 字符显示	245
7.4.1 字符取模	245
7.4.2 字符显示原理	248
7.4.3 内嵌 RAM 的配置和例化	249
7.4.3 Verilog 代码	260
7.4.4 工程实践	266
7.5 逻辑 (Verilog) 实例 12——基于 In-System Memory Content Editor 的 LCD 实时显示 字符更改	268
7.6 基于 Qsys 的 NIOS II 实例 7——Qsys 的 LCD 组件设计	273
7.6.1 系统原理概述	273
7.6.2 LCD 驱动移植	274
7.6.3 SDRAM 控制器设计	278
7.6.4 Avalon-MM 从机接口设计	282
7.6.5 数据缓存模块和 FIFO 配置	284
7.6.6 PLL 配置与复位设计	295
7.6.7 Qsys 系统构建	300
7.6.8 管脚分配与时序约束	311
7.6.9 软件工程实例	321
8 SF-SENSOR 子板开发指南	330
8.1 功能与原理图介绍	330
8.1.1 主要外设芯片及电路图解析	330
8.1.2 装配示意图	336
8.2 基于 Qsys 的 NIOS II 实例 8——SPI 接口字库芯片控制	340
8.2.1 新 Qsys 系统——添加 SPI 组件	340
8.2.2 SPI 外设驱动——编程原理	354
8.2.3 字库芯片驱动——编程原理	356
8.2.4 软件工程实例	364
8.3 基于 Qsys 的 NIOS II 实例 9——IIC 接口实时时钟 (RTC) 芯片控制	366
8.3.1 RTC 实时时钟芯片驱动原理	366
8.3.2 IIC 控制器组件设计	373
8.3.3 Qsys 系统构建	384
8.3.4 软件工程实例	390
8.4 逻辑 (Verilog) 实例 13——超声波测距数据采集	399
8.4.1 超声波模块驱动原理	399
8.4.2 数据采集平台构建	400
8.4.3 数据采集在线调试	403
8.5 基于 Qsys 的 NIOS II 实例 10——超声波测距换算	406
8.5.1 超声波模块组件创建	406
8.5.2 硬件系统搭建	408
8.5.3 软件工程调试	412
8.6 逻辑 (Verilog) 实例 14——基于 CMOS Sensor 的视频采集显示	416
8.6.1 CMOS 摄像头应用背景与驱动原理	416



8.6.2 采集系统设计概述	418
8.6.3 IIC 接口配置模块设计	420
8.6.4 视频流采集模块设计	428
8.6.5 工程移植	434
8.6.6 CMOS Sensor 接口时序约束	446
8.6.7 板级调试	455
附录 A 实例与工程映射	457
附录 B 套件淘宝购买链接	459

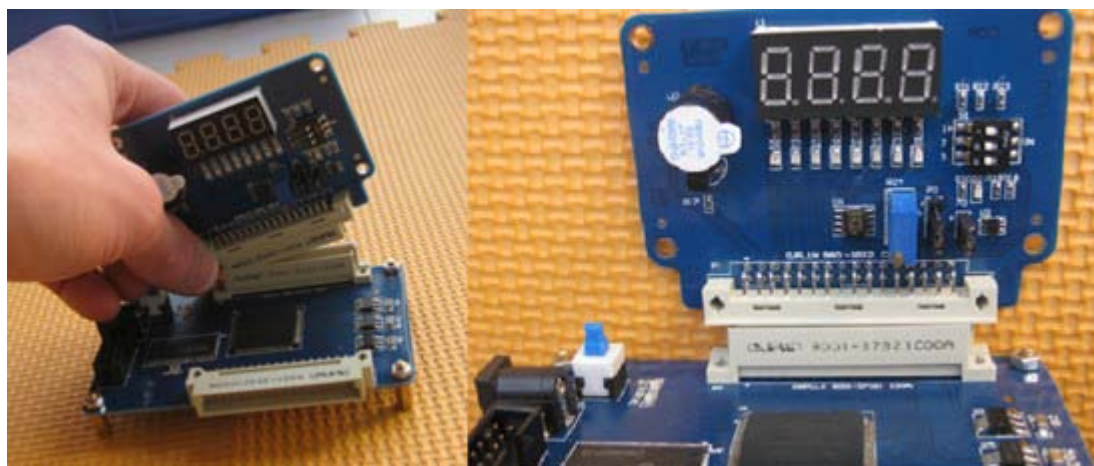


1 概述

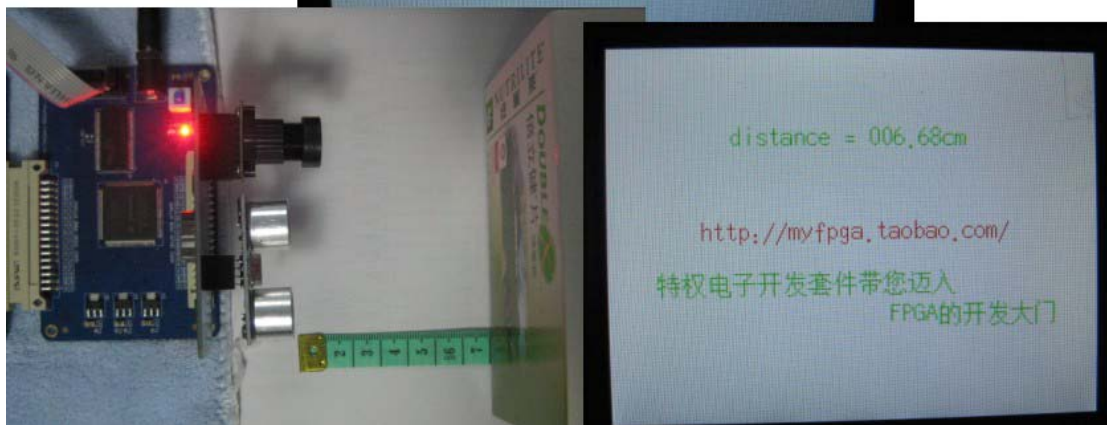


这一块小小的核心板，我们不希望它本身能做任何事，我们只是希望通过这个小小的板子以及它的两组可扩展的接口，将来连接各种各样的外设，作为一个电子爱好者无限 DIY 的平台。你热爱电子设计吗？你希望有一个自己可以随意扩展的 DIY 平台吗？我希望将来 SF-CY3 能够满足你的这些愿望！这个开发指南不会就此停止，它会不停的升级版本，它会一直以最接近初学者的方法图文并茂手把手的展示每一个新的设计实例。想得到最新版本的文档吗？OK，请到首页的 FPGA/CPLD 助学小组链接中寻找！

下面是我们的 SF-CY3 核心板和各个功能子板的连接效果，先给大家一点视觉冲击，哈哈，更多精彩，大家好好消化每个章节的具体内容。本文档最后的附录 B 也给出来所有模块的淘宝购买链接。



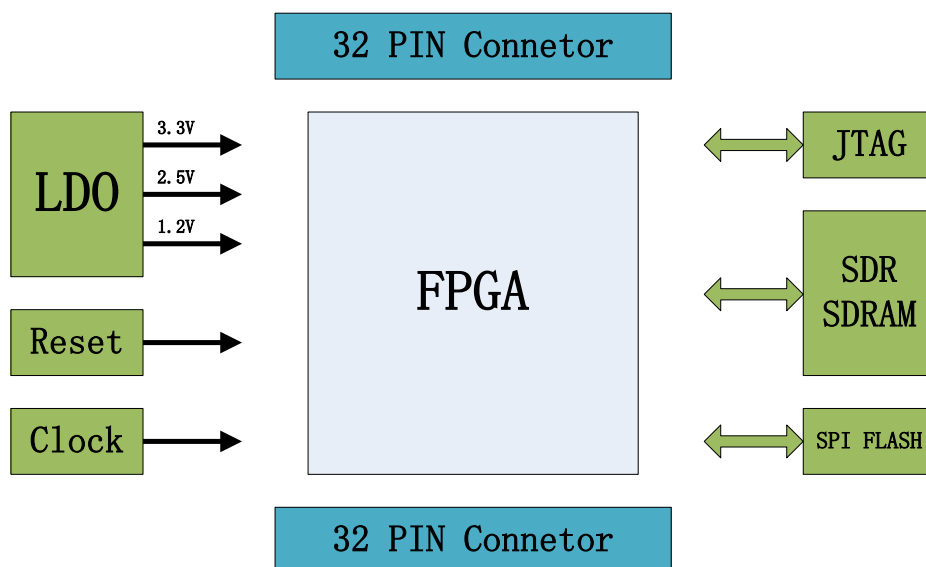






1.1 功能框图

SF-CY3 核心板除了一颗昂贵的 FPGA 芯片外, 电源、复位、时钟、JTAG 一个不能少。我们这颗芯片的电源有三档, 即 3.3V、2.5V 和 1.2V。3.3V 是供给 FPGA 的 I/O 电压, 也是系统的其他外设芯片(如 SDR SDRAM 和 SPI FLASH)的电源电压; 2.5V 是供给 JTAG 电路和 FPGA 的 PLL 电源所需要的; 1.2V 则是 FPGA 的内核电压。使用的时钟是 25MHz, 有人说这么低, 不够用的? 非也, FPGA 内部的 PLL 就是专门负责管理时钟的, 它可以对外部输入的 25MHz 时钟倍频或分频, 甚至非整数倍的倍频或分频也能够办到。JTAG 是用于 PC 和 FPGA 连接的电路, PC 上的 Quartus II 下载烧录就是通过这个接口。



另外, 有两颗存储器, SPI FLASH 是用来给 FPGA 做配置芯片的, 我们都知道 FPGA 是基于 RAM 结构的, 下电后不能够保存, 所以需要配一颗非易失的 FLASH 用于存储 FPGA 的配置数据, 当 FPGA 上电后, 它本身是空白的, 但是通过专有的 SPI 接口连接到 SPI FLASH 将数据搬运到 RAM 上, 然后 FPGA 就 RUN 起来了。SDR SDRAM 是用于做扩展使用的, 它既可以作为 NIOS II 处理器的 RAM 运行程序, 也可以作为后续 LCD 等需要实时大数据量存储的应用。另外, 上下两个 32PIN 的连接器, 分别引出 23 个 I/O 管脚, 将来各种扩展办卡就通过他们做文章了。



1.2 Cyclone III 系列 FPGA 器件简介

SF-CY3 FPGA 核心板使用 Altera 公司的 Cyclone III 家族 EP3C5E144C8 器件, Cyclone III 系列 FPGA 器件具有以下特性:

Cyclone III FPGA 系列: 一切皆有可能



前所未有的同时实现了低功耗、高性能和低成本

Cyclone® III FPGA 系列前所未有的同时实现了低功耗、高性能和低成本, 大大提高了您的竞争力。其特性以及 Cyclone III FPGA 体系结构为您的大批量、低功耗、低成本应用提供了理想的解决方案。为满足您独特的设计需求, 这一 FPGA 系列包括:

- **Cyclone III:** 功耗最低、成本最低的高性能 FPGA
- **Cyclone III LS:** 具有安全特性、功耗最低的 FPGA

Cyclone III LS 器件具有 200K 逻辑单元、8 Mbits 嵌入式存储器以及 396 个嵌入式乘法器, 是高性能处理、低功耗应用的理想选择, 包括:

- 汽车
- 消费类
- 显示
- 工业
- 视频和图像处理
- 无线

轻松达到您的功耗目标

具有 200K 逻辑单元(LE)、8-Mbits 存储器, 而静态功耗不到 1/4 瓦, 该系列设立了功耗标准。采用台积电(TSMC)的低功耗(LP)工艺技术进行制造, 无论是通信设备、手持式消费类产品, 还是软件无线电设备, 这些 FPGA 都能够轻松满足您的功耗预算。

设计安全性



Cyclone III LS FPGA 利用低功耗、高性能 FPGA 平台,在硬件、软件和知识产权(IP)层面上率先实现了一系列安全特性。一系列安全特性保护了您的 IP 不被篡改、逆向剖析和克隆。而且,这些器件还使您能够通过设计分离特性,在一个芯片中实现冗余功能,从而减小了实际应用的体积、重量和功耗。

全面的设计资源

为确保流畅、成功的设计流程,帮助您更快的将构思变为收益,Altera 提供全面的 Cyclone III FPGA 设计环境,包括:

- Quartus® II 开发软件
- 成熟的 IP 库
- Nios® II, 世界上最通用的嵌入式处理器
- 低成本开发套件
- 专用参考设计

将您的设计从构思变为产品,更迅速推向市场。采用 Cyclone III FPGA,一切皆有可能。

1.3 EP3C5E144C8 器件资源一览

在过去,衡量一个逻辑器件的资源情况,仅仅看他的逻辑资源便可知其一二,但随着制造工艺的不断进步,大量的存储器、乘法器资源可以很轻易的潜入到可编程逻辑器件之中,大大便利设用户的设计。所以,今天我们必须同时去衡量这些逻辑器件中所包含的存储器、乘法器甚至时钟、I/O 的资源情况,毕竟他们也和我们的实际应用息息相关。Cyclone III 器件采用了成熟的 65ns 工艺,除了拥有丰富的逻辑资源外,存储器资源、乘法器资源、时钟资源、I/O 资源也非常丰富,可以满足大多数的中等规模以下的的应用。从 handbook 中截下一个资源分布表,从这里我们便可对 Cyclone III 家族各个型号器件的资源情况一目了然。



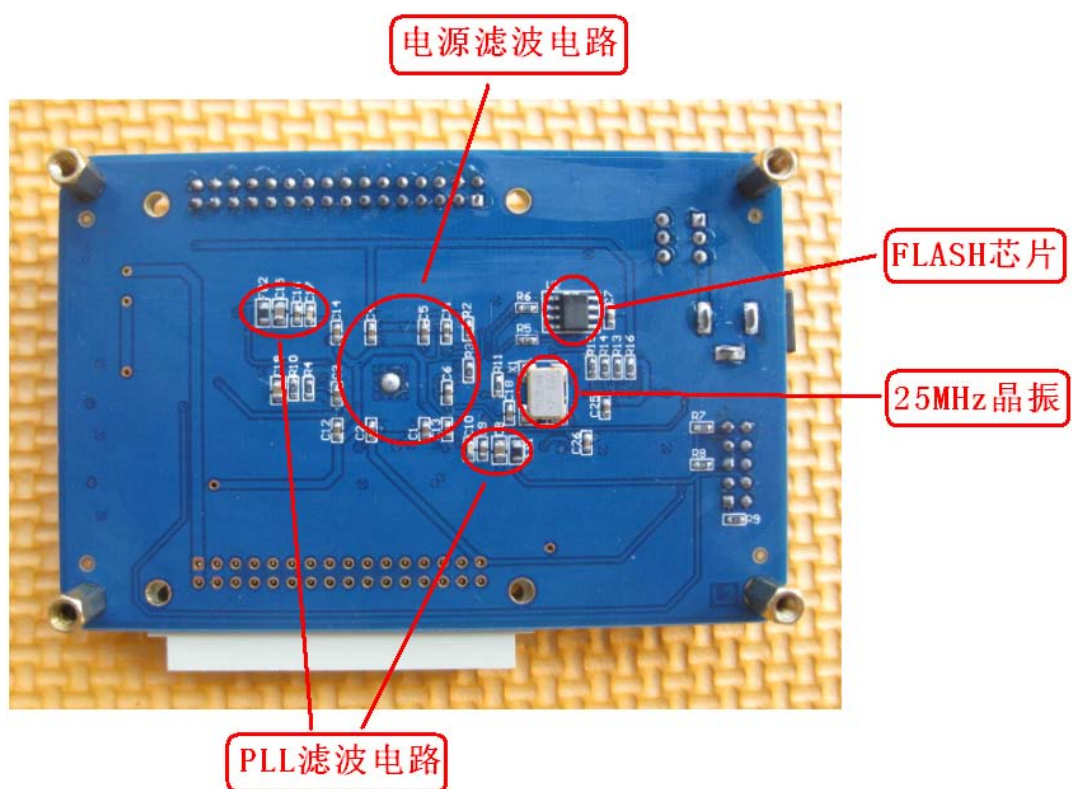
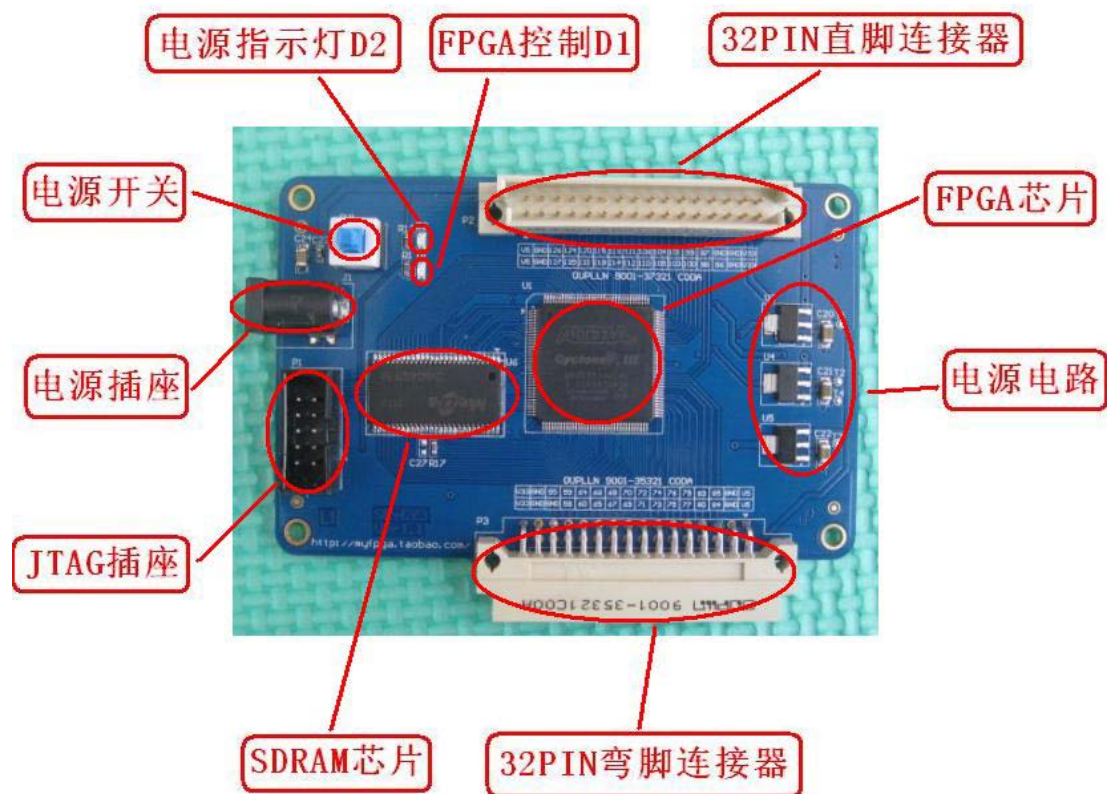
Family	Device	Logic Elements	Number of M9K Blocks	Total RAM Bits	18 x 18 Multipliers	PLLs	Global Clock Networks	Maximum User I/Os
Cyclone III	EP3C5	5,136	46	423,936	23	2	10	182
	EP3C10	10,320	46	423,936	23	2	10	182
	EP3C16	15,408	56	516,096	56	4	20	346
	EP3C25	24,624	66	608,256	66	4	20	215
	EP3C40	39,600	126	1,161,216	126	4	20	535
	EP3C55	55,856	260	2,396,160	156	4	20	377
	EP3C80	81,264	305	2,810,880	244	4	20	429
	EP3C120	119,088	432	3,981,312	288	4	20	531

当然了，这里的资源情况，并不是我们 SF-CY3 板上那颗 EQFP144 封装器件的实际资源情况，它是 EP3C5 这个型号所有封装中最大支持的资源。不过，您也别嗤之以鼻，除了 I/O 数量不足 182 以外，其他的资源还真是实打实的。对于很多的工业应用来说，这个规模的 FPGA 器件足矣；对于初学者入门 FPGA 来说，那更是绰绰有余了，我还真不信有哪几个同学能够写下数万行代码充分的使用这颗芯片。总之，一句话，学习，不求最贵的（不过它已经够贵了 🤔），但求最适用的。

2 SF-CY3 硬件电路解析

2.1 实物示意图

前面已经看过 SF-CY3 核心板的硬件框图，下面我们通过实物照片来认识下各个元器件。

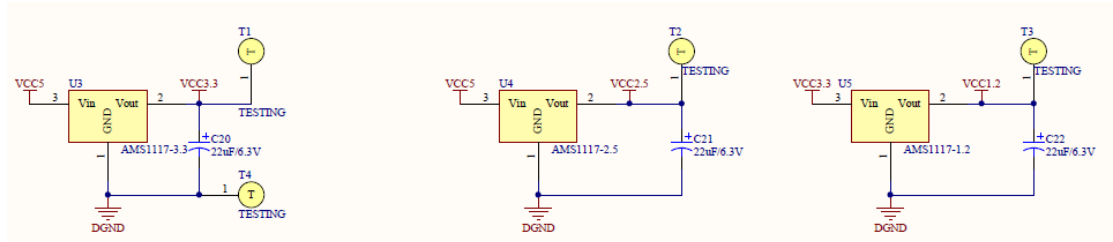




2.2 原理图解析

2.2.1 系统电源电路

整个系统需要三档不同的电源电压，即 3.3V、2.5V 和 1.2V，我们使用了三颗 LDO 分别产生。T1/T2/T3/T4 则是测试点，便于生产或调试过程中的电压测量。



2.2.2 FPGA 电源电路

从 Cyclone III Device Handbook 中可以查到，Cyclone 家族器件的供电一般是分 4 大类，压值有 2-3 档。这 4 类电源分别是内核电压 VCCINT、I/O 电压 VCCIO、PLL 模拟电压 VCCA 和 PLL 数字电压 VCCD_PLL。这 4 类电源电压中，内核电压固定 1.2V、PLL 模拟电压固定 2.5V、PLL 数字电压固定 1.2V；唯一不确定，或者说有选择余地的电源是 I/O 电压，它可以根据用户的实际应用 I/O 标准选择不同的电压，我们的板子使用最常用的 3.3V 的 I/O 供电。

Table 1-3. Cyclone III Devices Recommended Operating Conditions (1), (2)

Symbol	Parameter	Conditions	Min	Typ	Max	Unit
V _{CCINT} (3)	Supply voltage for internal logic	—	1.15	1.2	1.25	V
	Supply voltage for output buffers, 3.3-V operation	—	3.135	3.3	3.465	V
	Supply voltage for output buffers, 3.0-V operation	—	2.85	3	3.15	V
	Supply voltage for output buffers, 2.5-V operation	—	2.375	2.5	2.625	V
V _{CCIO} (3), (4)	Supply voltage for output buffers, 1.8-V operation	—	1.71	1.8	1.89	V
	Supply voltage for output buffers, 1.5-V operation	—	1.425	1.5	1.575	V
	Supply voltage for output buffers, 1.2-V operation	—	1.14	1.2	1.26	V
V _{CCA} (3)	Supply (analog) voltage for PLL regulator	—	2.375	2.5	2.625	V
V _{CCD_PLL} (3)	Supply (digital) voltage for PLL	—	1.15	1.2	1.25	V
	Input voltage	—	-0.5	—	3.6	V
	Output voltage	—	0	—	V _{CCIO}	V

说到 I/O 电压，我们不得不多提两句，毕竟可以兼容非常多的 I/O 电压标准是 FPGA 的一大优势，尤其是各种高速差分信号的支持。下图中列出了我们这款器件支持的各种 I/O 电

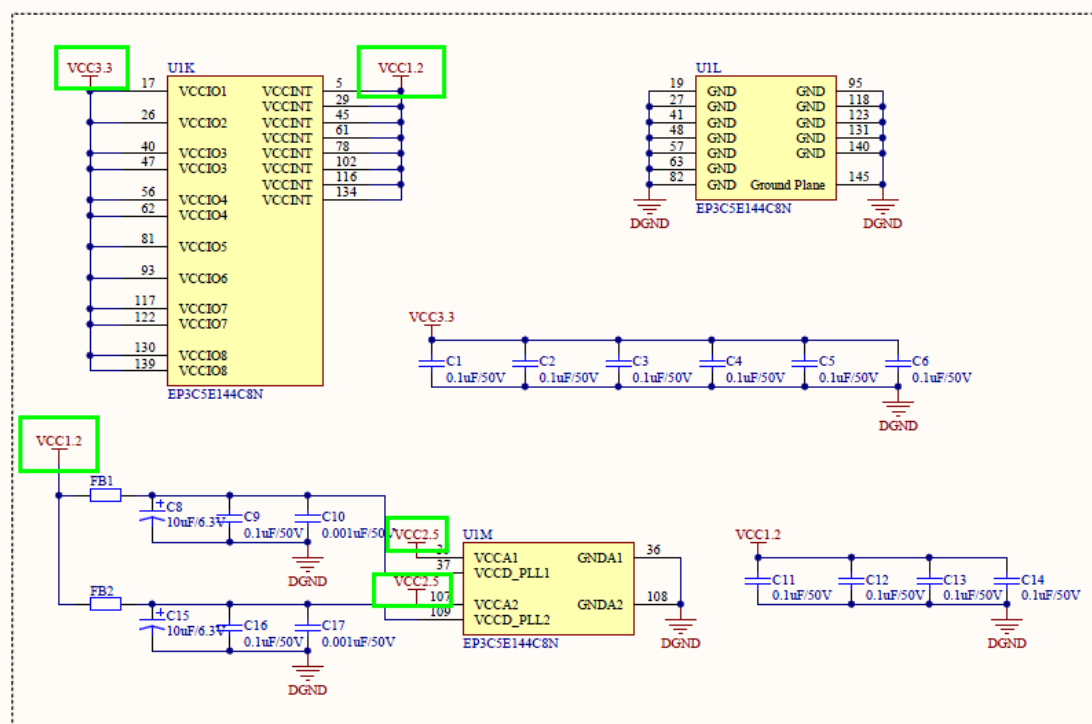
《圣经》箴言九 11 “敬畏耶和华是智慧的开端，认识至胜者便是聪明。”



平标准。

I/O Standard	I _{OH} /I _{OL} Current Strength Setting (mA)	
	Top and Bottom I/O Pins	Left and Right I/O Pins
1.2-V LVCMOS	2, 4, 6, 8, 10, 12	2, 4, 6, 8, 10
1.5-V LVCMOS	2, 4, 6, 8, 10, 12, 16	2, 4, 6, 8, 10, 12, 16
1.8-V LVTTTL/LVCMOS	2, 4, 6, 8, 10, 12, 16	2, 4, 6, 8, 10, 12, 16
2.5-V LVTTTL/LVCMOS	4, 8, 12, 16	4, 8, 12, 16
3.0-V LVCMOS	4, 8, 12, 16	4, 8, 12, 16
3.0-V LVTTTL	4, 8, 12, 16	4, 8, 12, 16
3.3-V LVCMOS ⁽²⁾	2	2
3.3-V LVTTTL ⁽²⁾	4, 8	4, 8
HSTL-12 Class I	8, 10, 12	8, 10
HSTL-12 Class II	14	—
HSTL-15 Class I	8, 10, 12	8, 10, 12
HSTL-15 Class II	16	16
HSTL-18 Class I	8, 10, 12	8, 10, 12
HSTL-18 Class II	16	16
SSTL-18 Class I	8, 10, 12	8, 10, 12
SSTL-18 Class II	12, 16	12, 16
SSTL-2 Class I	8, 12	8, 12
SSTL-2 Class II	16	16
BLVDS	8, 12, 16	8, 12, 16

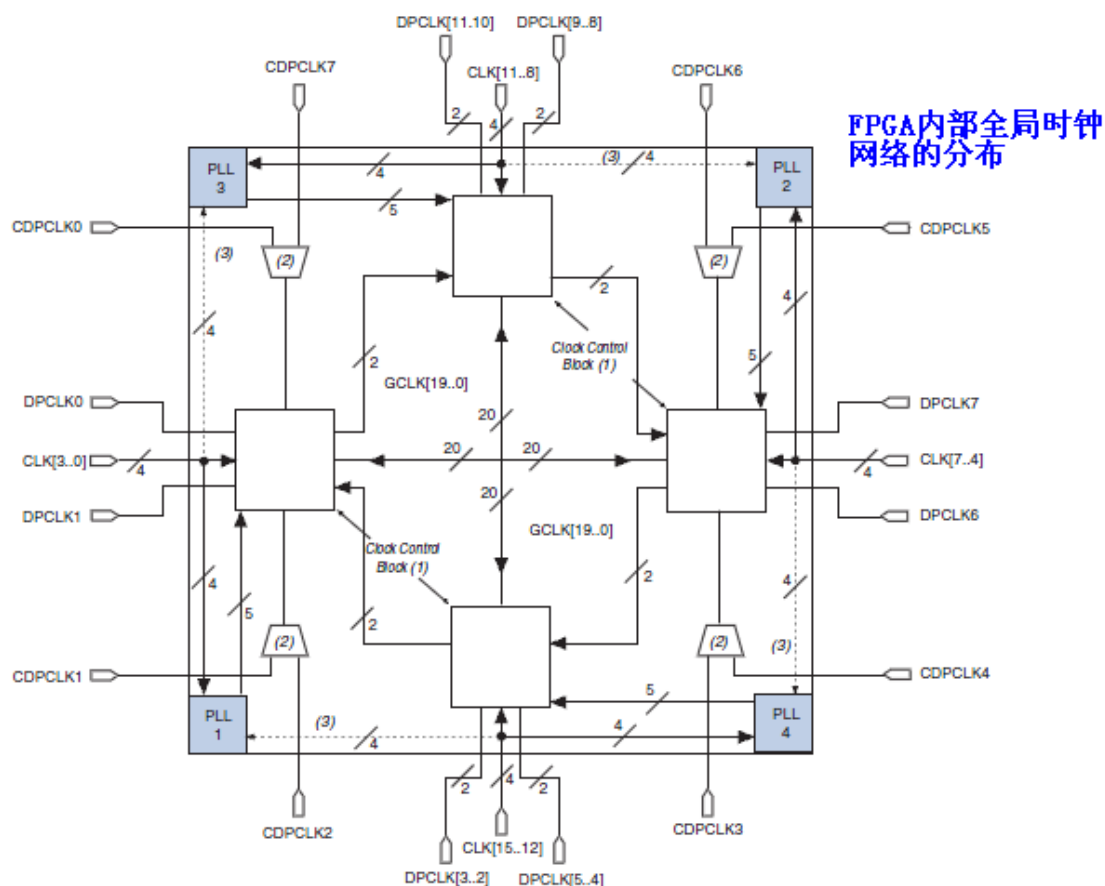
除了上述提及的 4 类电源电压外，其实我们这款 FPGA 器件的配置电路的供电电压也有学问，虽然它的 I/O 也是 3.3V 的，但一些地方却需要用到 2.5V。这个内容我们到配置电路再详细讨论。



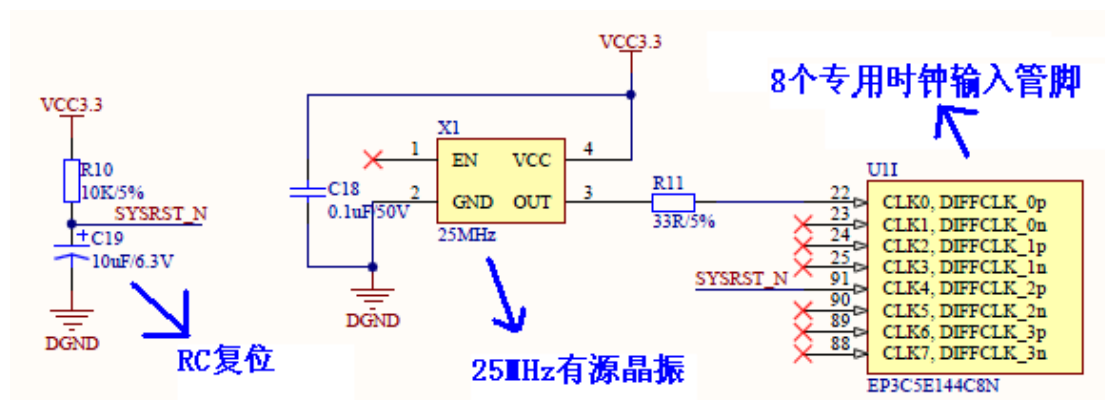


2.2.3 时钟和复位电路

FPGA 的时钟输入都有专用管脚, 通过这些专用管脚输入的时钟信号, 在 FPGA 内部可以很容易的连接到全局时钟网络上。所谓的全局时钟网络, 是 FPGA 内部专门用于走一些有高扇出、低时延要求的信号, 这样的资源相对有限, 但是非常实用。FPGA 的时钟和复位通常是需要走全局时钟网络的。



复位使用简单 RC 电路, 也是连接到 FPGA 的专用输入时钟管脚, 走内部全局时钟网络。

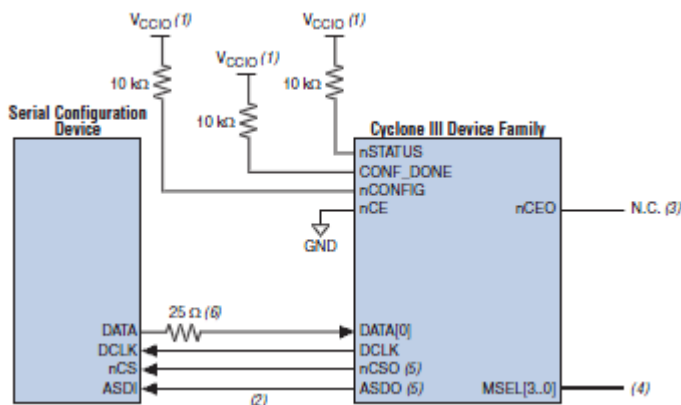




2.2.4 FPGA 配置电路

我们所说的 FPGA 配置电路，一方面要完成从 PC 上把 bit 文件下载到 FPGA 或存储器的任务，另一方面则要完成 FPGA 上电启动时加载配置数据的任务。

先看 handbook 中给出的一些相关参考设计，下面是 FPGA 和用于配置的 SPI FLASH 的接口连接方式，注意 FPGA 的几个主要管脚 Nstatus\CONF_DONE\Nconfig\nce 的连接，或者上拉，或者接地。DATA\DCLK\NCSO\ASDO 这 4 个管脚便是 SPI 接口，连接到 SPI FLASH。

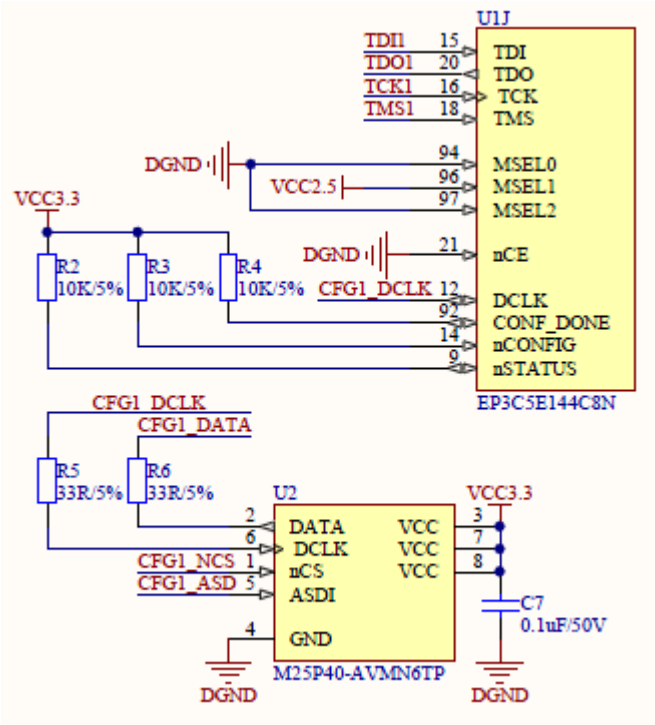


FPGA 有组 MSEL 管脚是用于设置 FPGA 初上电时的启动模式，我们使用的 EP3C5 器件没有 bit3，只有 bit2-0，上电使用 FastAS 模式从 SPI FLASH 里面加载配置数据。

Table 9-7. Cyclone III Device Family Configuration Schemes ⁽¹⁾ (Part 1 of 2)

Configuration Scheme	MSEL				Configuration Voltage Standard (V) ^{(2), (3)}
	3	2	1	0	
Fast Active Serial Standard (AS Standard POR)	0	0	1	0	3.3
Fast Active Serial Standard (AS Standard POR)	0	0	1	1	3.0/2.5
Fast Active Serial Fast (AS Fast POR)	1	1	0	1	3.3
Fast Active Serial Fast (AS Fast POR)	0	1	0	0	3.0/2.5
Active Parallel ×16 Standard (AP Standard POR, for Cyclone III devices only)	0	1	1	1	3.3
Active Parallel ×16 Standard (AP Standard POR, for Cyclone III devices only)	1	0	1	1	3.0/2.5
Active Parallel ×16 Standard (AP Standard POR, for Cyclone III devices only)	1	0	0	0	1.8
Active Parallel ×16 Fast (AP Fast POR, for Cyclone III devices only)	0	1	0	1	3.3

有了前面的理论做铺垫，我们的设计也就有理有据了。

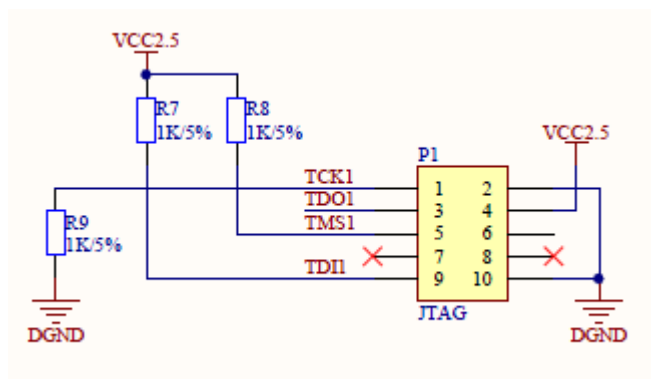


至于 SPI FLASH 存储大小选择，也是有据可依的。EP3C5 的配置数据需要 3Mbit 左右，因此我们选择了 4Mbit 的 M25P40 芯片。

Table 9-3. Cyclone III Device Family Uncompressed Raw Binary File (.rbf) Sizes

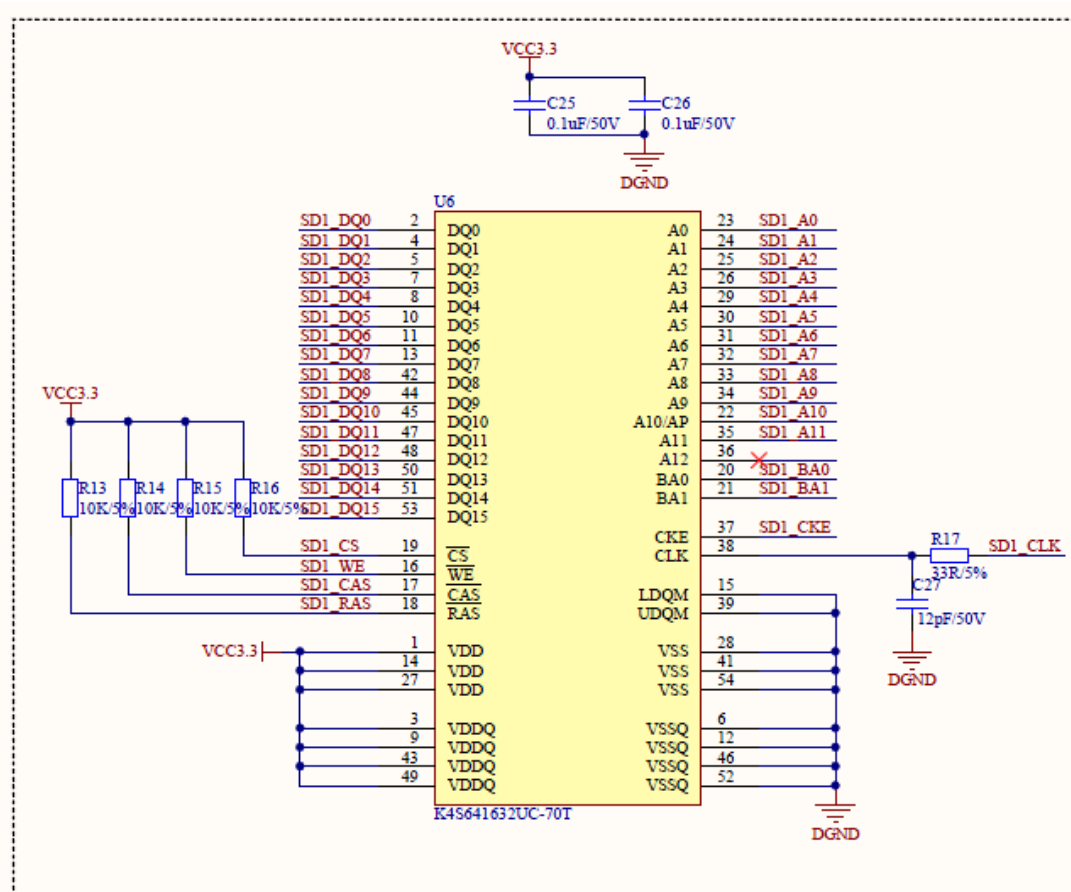
Device		Data Size (bits)
Cyclone III	EP3C5	3,000,000
	EP3C10	3,000,000
	EP3C16	4,100,000
	EP3C25	5,800,000
	EP3C40	9,600,000
	EP3C55	14,900,000
	EP3C80	20,000,000
	EP3C120	28,600,000
Cyclone III LS	EP3CLS70	26,766,760
	EP3CLS100	26,766,760
	EP3CLS150	50,610,728
	EP3CLS200	50,610,728

前面是配置电路的一大任务，即上电启动配置数据的电路，通过一个 SPI FLASH 来实现。而另一个任务即下载，则是通过 JTAG 来实现的，JTAG 这个概念网络上满天飞了，大家自己去消化，但凡有 CPU 的地方，基本都有 JTAG 的存在，FPGA 也不例外。唯一需要大家注意的是 Cyclone 器件的 JTAG 电压必须是 2.5V。



2.2.5 SDRAM 电路

SDRAM 的电路很简单，只要将地址总线、数据总线、控制总线连接到 FPGA 的 I/O 口上即可。而需要特别小心的是 SDRAM 的时钟信号 SD1 CLK 可不能随便乱接。

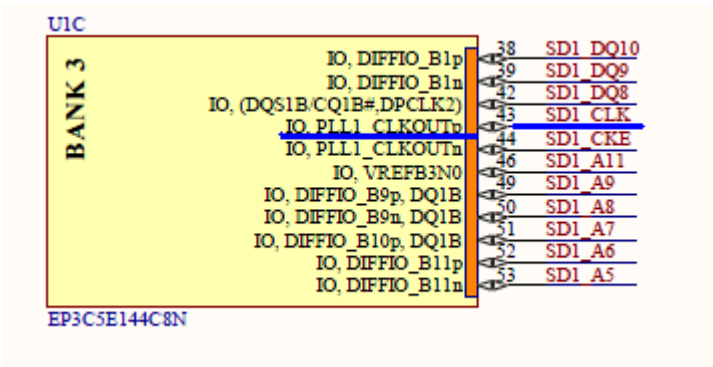


看 FPGA 这端，我们是吧 SD1_CLK 专门连接到了 PLL1_CLKOUTp 这个管脚上。这个管脚有什么特别的，它又有什么学问？它的作用和它的名字一样，我们可以先找到它下面的一个管脚名为 PLL1_CLKOUTn，他们是一对的，他们的时钟源是来自于 FPGA 的 PLL。为什么 PLL

《圣经》箴言九 11 “敬畏耶和华是智慧的开端，认识至胜者便是聪明。”

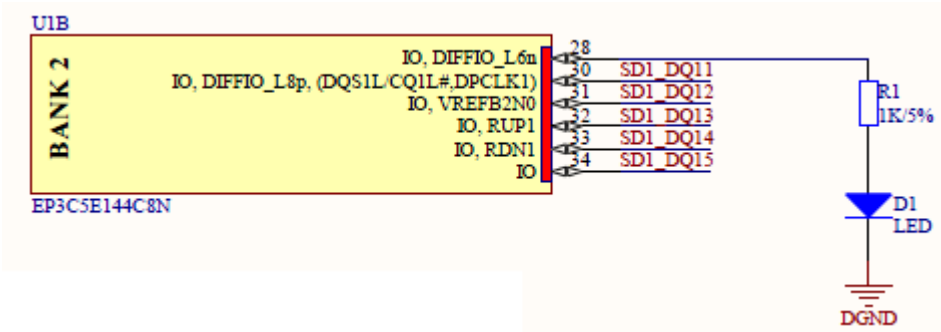


输出的时钟一定要有专门的这样一对管脚呢？和前面的全局时钟网络存在的意义有异曲同工之妙。PLL 到这对管脚上的延时相对是比较受控的，目的就是为了得到更低延时更稳定可靠的时钟信号。SDRAM 的时钟高达 100MHz 以上，所以就必须使用这个专用的管脚。



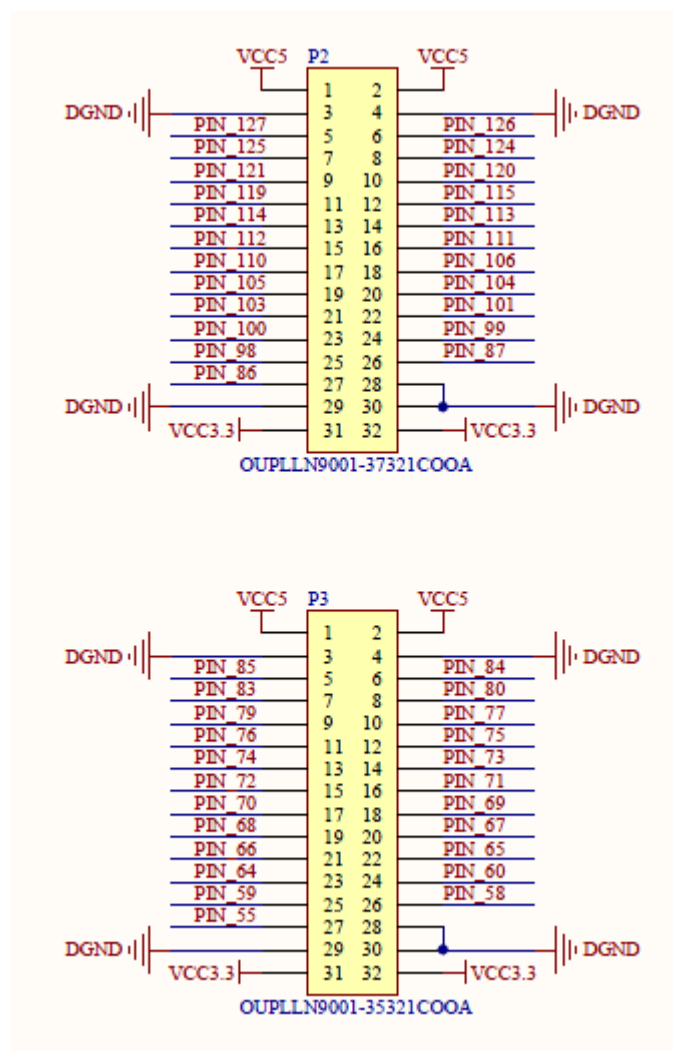
2.2.6 LED 指示灯

板上专门留了一个 LED 指示灯，用于板子的测试。I/O_28 输出高电平 LED 导通发光，低电平 LED 截止则不亮。



2.2.7 连接器电路

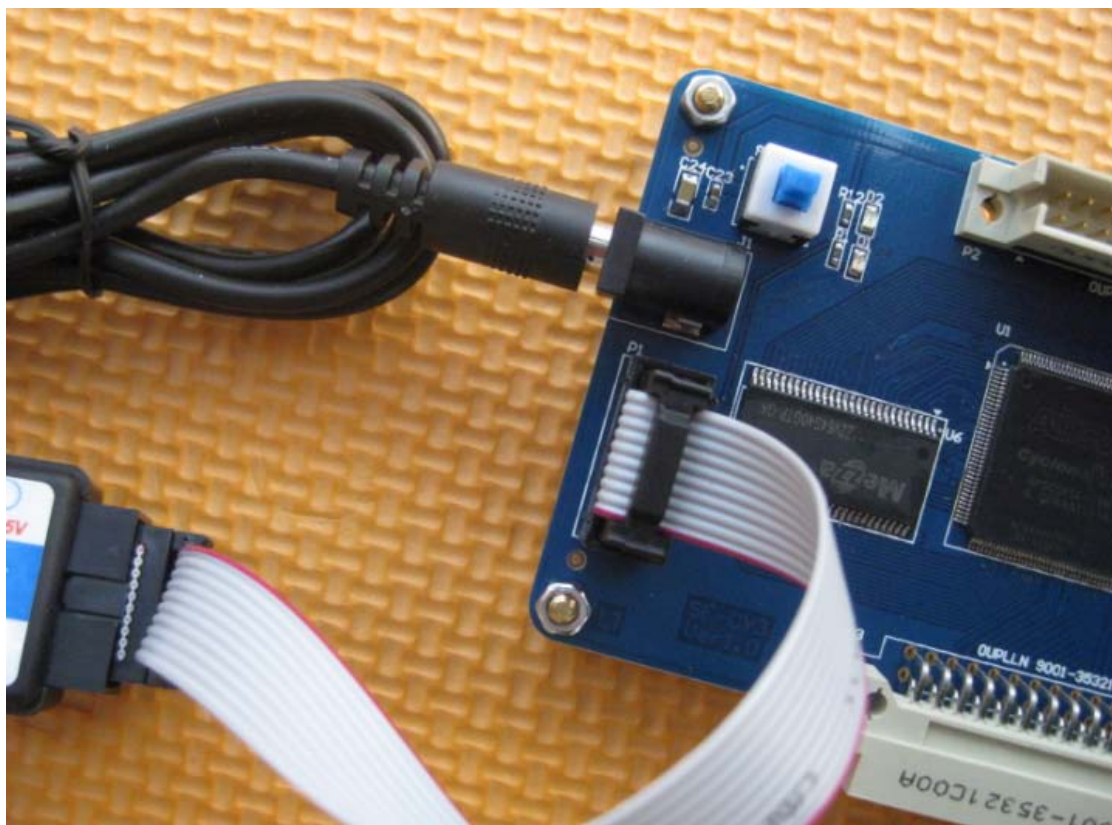
该板子通过两个 32PIN 的连接器将 46 个 I/O 脚引出。每个连接器都有可用的 23 个 I/O。5V\3.3V 电源也都引出扩展。



3 SF-CY3 基本使用安装说明

3.1 电路板安装

要玩转咱的板子，除了一块光溜溜的核心板还不够，需要两样东西，即 5V 电源和 USB Blaster 下载线，如图所示。USB Blaster 的 USB 端口必须连接到 PC 上。



3.2 Quartus II 与 ModelSim 软件下载与安装

3.2.1 EDA 工具概述

话说“工欲善其事，必先利其器”，FPGA/CPLD 开发所涉及的 EDA 工具的安装和使用比《圣经》箴言九 11 “敬畏耶和华是智慧的开端，认识至胜者便是聪明。”



一般的软硬件开发要复杂和麻烦得多,为了帮助广大初学者少走弯路,笔者觉得也很有必要通过手把手比较详细的介绍让大家快速完成这项艰巨任务。

因为使用的是 **Altera** 公司的器件,所以我们别无选择的锁定了 **Quartus II** 这款 **Altera** 面对自身器件的集成开发工具。我们可以在 **Quartus II** 上完成设计输入(主要是编写代码)、语法检查、时序约束、综合、映射(布局布线)以及配置文件的下载操作。在早期的 **Quartus II** 上还能看到集成的仿真功能,可以用波形方式产生激励信号,不过功能过于单一,应付非常简单的设计仿真还凑合,稍复杂点就力不从心了。因此,现在的 **Quartus II** 上已经让这个“鸡肋”功能彻底消失了,取而代之的是与更专业的 **Mentor Graphics** 公司合作推出了 **ModelSim-Altera** 版本的仿真工具。

关于 EDA 工具的问题,使用 **Altera** 公司的器件,在设计规模并不很大很复杂的情况下,其实我们推荐大家使用 **Quartus II + ModelSim-Altera** 或 **Quartus II + ModelSim SE** 的搭配就可以完成大多数的工程。

3.2.2 软件下载和 license 申请

Altera 公司的官网也有直接的工具体下载支持,大家可以直接访问他们的软件下载支持网页(<http://www.altera.com.cn/support/software/sof-index.html>)。在他们的网页中,我们可以点击相关的链接进入下载页面,如前文提到我们将会使用的 **Quartus II** 和 **Altera-ModelSim**。而对于这些工具的安装许可,有多种方式,据了解,大致可以有以下几种方式(可参考官方文档 http://www.altera.com.cn/literature/manual/quartus_install.pdf):

- 网络版本无须 license,只是部分高级功能受限。如 **Quartus II Web Edition** 和 **ModelSim-Altera Starter Edition**。对于一般的初学者,这样的版本绝对够用。
- 订购版本的 **Quartus II** 可以提供 30 天的试用期限。
- 找国内的代理商艾瑞或骏龙,他们应该都能够提供 60 天的试用 license,据说没有申请次数限制,只要每隔 60 天向他们提交一次申请即可。



The screenshot shows the Altera website's 'Design Software Support' page. The header includes the Altera logo and navigation links. The sidebar on the left lists various categories: Products (Quartus II, Qsys, SOPC Builder, MAX+PLUS II, ModelSim-Altera), Resource Center (Introduction, Installation & Licensing, Scripts, PCB Design & I/O, Netlist Reader & Synthesis, Compilation Enhancement, Optimization, Power Management, TimeQuest Timing Analyzer, Standard Timing Analyzer, Emulation & Verification, On-chip Debugging, HardCopy Design), Software Resources (OS Support, Driver Installation), Downloads & Licenses (Download, License), and Quartus II EDA Support (Quartus II Interface, Synthesis Tools, Emulation Tools, Verification Tools). The main content area is titled 'Design Software Support' and lists supported products: Quartus II Software Support, SOPC Builder Support, Nios II Integrated Development Environment (IDE) Support, ModelSim-Altera Software Support, DSP Builder, and MAX+PLUS II Software Support. It also includes a 'Resource Center' section with links to Introduction, Installation & Licensing, Scripts, PCB Design & I/O, Netlist Reader & Synthesis, Compilation Enhancement, Optimization, Power Management, TimeQuest Timing Analyzer, Standard Timing Analyzer, Emulation & Verification, On-chip Debugging, and HardCopy Design. There are also promotional banners for downloading Quartus II v6.0 SP1 and joining the Altera forum.

而 license 通常有两种, 即 Fixed license (固定的) 和 floating license (浮动), 安装方式稍有差异。Fixed license 是用户根据固定的电脑申请的, 只能由于申请的本地 PC。floating license 更实用一些, 一些公司申请多个 floating license, 然后局域网内所有的 PC 都可以使用这个 floating license, 只不过同时在线数受到 license 数量的限制。

我们的学习将使用 Quartus II Web Edition 和 ModelSim-Altera Starter Edition。首先要到官方网站下载这两个软件。打开 Quartus II Web Edition 下载网页后, 直接点击 12.0sp1 quartus free windows.exe (默认大家都使用 windows 系统) 进行下载。

Quartus II Web Edition 下载链接:

https://www.altera.com/download/software/quartus-ii-we/zh_cn



ALTERA

Search

下载 文档资料 myAltera / 退出

器件 设计软件及服务 最终市场 技术中心 教育与活动 支持 公司介绍 在线购买

设计软件

- Quartus II 订购版
- Quartus II 网络版
- MegaCore IP库
- ModelSim-Altera
- Nios II EDS
- DSP Builder
- OpenCL
- OS支持

存档

- 所有服务包
- 设计软件

许可

- 获取和管理许可
- 许可FAQ
- 许可监控

Quartus II 网络版 (服务包)

主页 > 支持 > Quartus II 网络版 (服务包)

发布日期: 2012 年 7 月

Quartus II 网络版 12.0 版 服务包 1

下载选项 1: 单独的文件

Quartus II Web Edition	平台	文件名称	大小
Quartus II Web Edition 服务包 1	Windows	12.0sp1_quartus_free_windows.exe MD5: b014edc485d0d1b17df7cb3b3e9ee56e	2.7 GB
Quartus II Web Edition 服务包 1	Linux	12.0sp1_quartus_free_linux.tar.gz MD5: 968ae9c1b7bccf5ae45f24758f754473	3.7 GB

[MD5的总价值体现在哪里, 它是用来做什么的](#)

其他单独的下载文件

如果大家是第一次下载 Altera 官方网站的软件, 那么也会弹出如图所示的下载工具 Download Manager 安装提示。建议点击“安装”选项, 随后再点击“Click to Download Your File Now”。

ALTERA

About the Download Manager

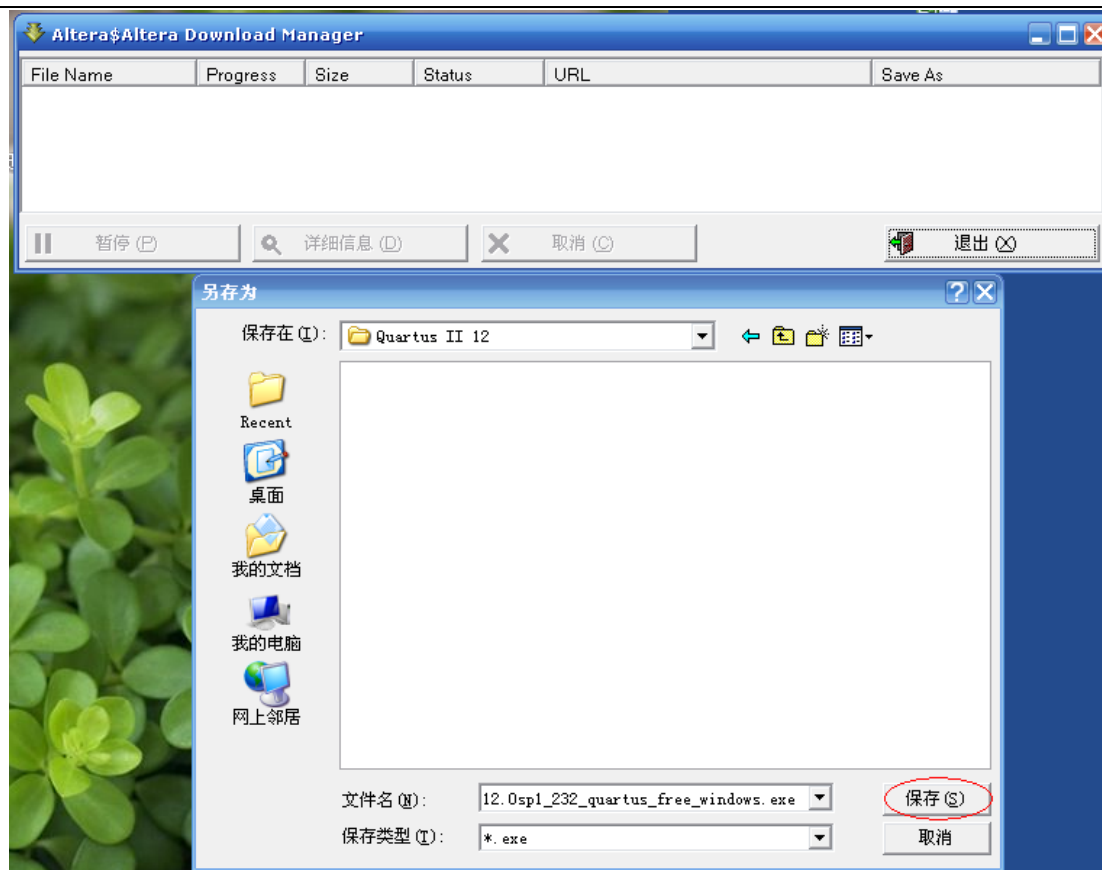
The Download Manager provides a more effective, more efficient way to download files than you r warning window, click Install to continue downloading your file. The Download Manager will insta

[Click to Download Your File Now](#)

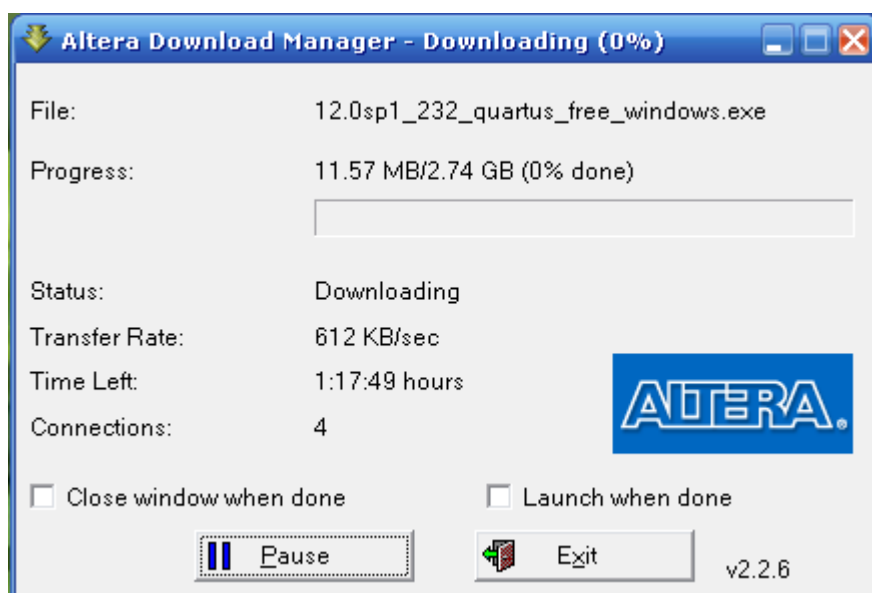
If the Download Manager fails to start, or if you do not accept the security certificate, you can [dowr](#)



在接下来弹出的窗口选择下载保存目录, 接着点击“保存”。



最后弹出 Altera Download Manager 中出现了下载文件的进度。



同样的方式, 打开 ModelSim-Altera Starter Edition 的下载页面, 需要分别选择并下载 12.0_modelsimsim_ae_windows.exe 和 12.0sp1_modelsimsim_ae_windows.exe。

ModelSim-Altera Starter Edition 下载地址如下:

https://www.altera.com/download/software/modelsim-starter/12.0/zh_CN



ModelSim-Altera入门版

[主页](#) > [支持](#) > [ModelSim-Altera入门版](#)

发布日期: 2012 年 6 月

用于Quartus II 12.0 SP2 的ModelSim-Altera入门版10.0d

下载选项1: Altera安装程序

下载一个小安装程序, 采用非NTLM代理, 选择您的设计软件。

使用Altera®安装程序, 采用非NTLM代理, 选择您的设计软件。您必须首先下载这一Altera小安装程序, 然后, 您可以选择您希望安装的软件和器件支持。Altera安装程序将文件大小减小了50%。

下载
Windows 版本
(14 MB)

下载
Linux 版本
(21 MB)

[安装订阅版服务包2](#)

Altera为Altera Quartus® II 软件订购版的客户提供Mentor Graphics® ModelSim® -Altera® 仿真软件的许可。网络版和订购版用户可获得ModelSim-Altera版和ModelSim-Altera入门版软件。

- ModelSim-Altera 版 – 需要许可 (付费)
- ModelSim-Altera 入门版 – 不需要许可 (免费)

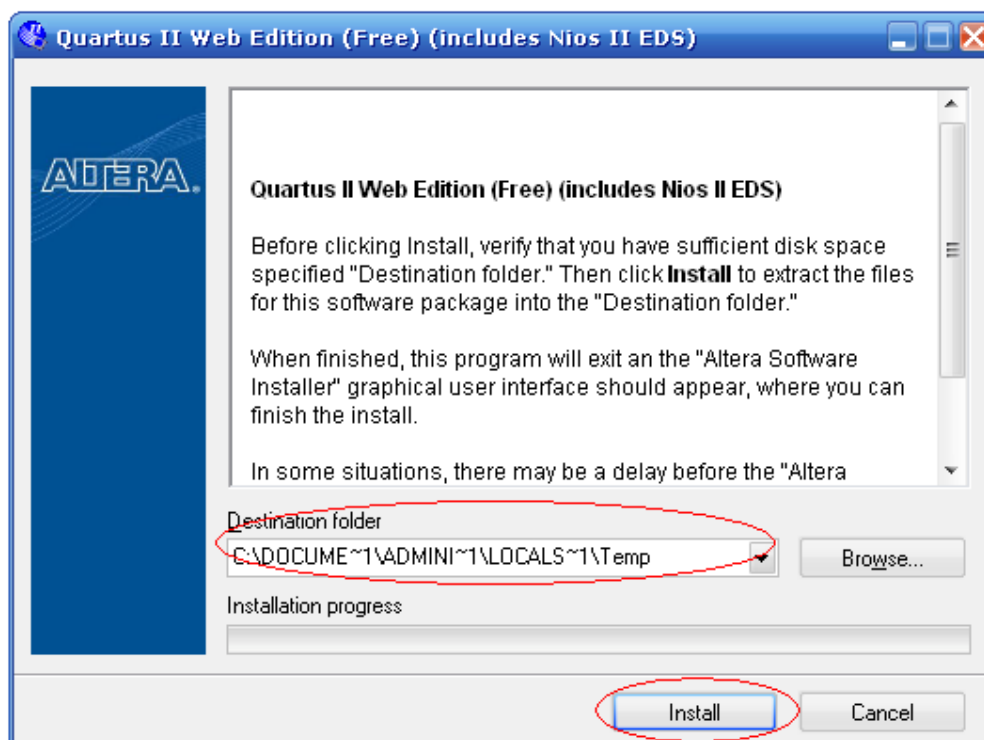
[阅读Altera软件12.0安装FAQ.](#)

下载选项2: 单独的文件

ModelSim-Altera Starter Edition	平台	文件名称	大小
Quartus II 12.0 的 10.0d 软件下载	Windows	12.0_modelsim_ase_windows.exe MD5: c9fef2ea96a70d2c7bb89398a2eebe2c	426 MB
Quartus II 12.0 的 10.0d SP 2	Windows	12.0sp2_modelsim_ase_windows.exe MD5: 56890d3bea802f632cb60f27abfb36c5	286 MB
Quartus II 12.0 的 10.0d SP 1	Windows	12.0sp1_modelsim_ase_windows.exe MD5: f0991364ce16318d5c1988efa7791d6e	286 MB
Quartus II 12.0 的 10.0d 软件下载	Linux	12.0_modelsim_ase_linux.tar.gz MD5: 08d499f1a3e00349e873c5b7834f8ef4	752 MB
Quartus II 12.0 的 10.0d SP 2	Linux	12.0sp2_modelsim_ase_linux.tar.gz MD5: ddb7478a1a9dc3b8d8365ae8b797a2ed	413 MB
Quartus II 12.0 的 10.0d SP 1	Linux	12.0sp1_modelsim_ase_linux.tar.gz MD5: a1e5a5e704e4feeb29a8fa9c8c309382	413 MB

3.2.3 Quartus II 的安装

接下来我们找到前面软件工具的下载保存路径, 首先安装 Quartus II Web Edition。双击“12.0sp1_232_quartus_free_windows.exe”, 弹出如图所示的安装页面, 我们所下载的文件实际上相当于一个压缩包, 点击后弹出的页面需要我们设置临时文件的存放路径 (即 Destination folder), 只要你确保该路径所在盘符有足够的硬盘空间即可 (通常应该是要有 3-5G 的空间)。点击 “Install” 开始解压。

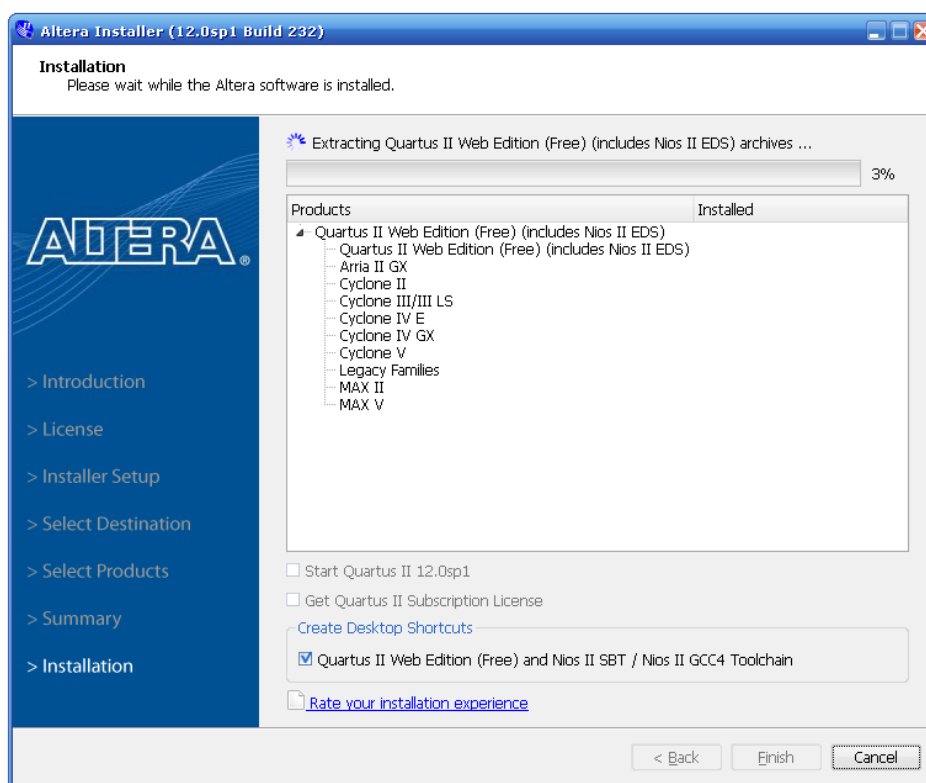


在解压完成后, 自动弹出如图所示的安装提示, 一路 Next 下去。当然了, 期间有一些个性化的设置是可以根据自己需要进行更改的, 如安装路径和实际器件家族的使用安装, 推荐大家使用默认设置。

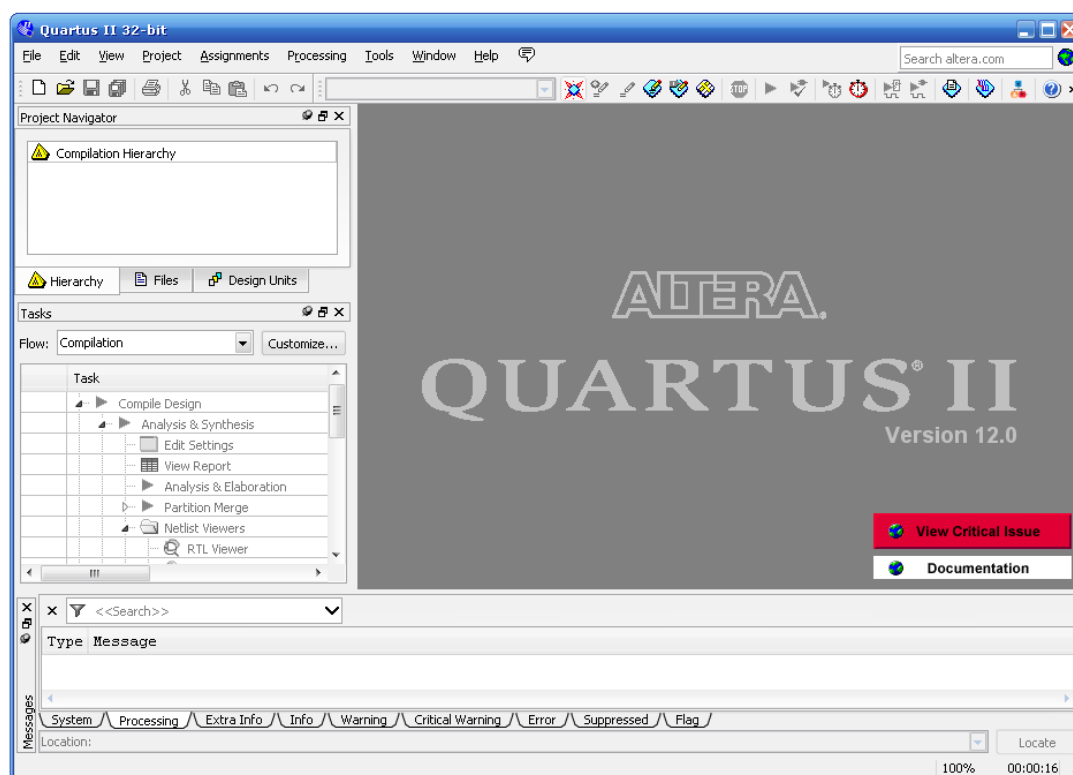




如图所示, Quartus II Web Edition 正在安装中。



打开最终安装好的 Quartus II Web Edition

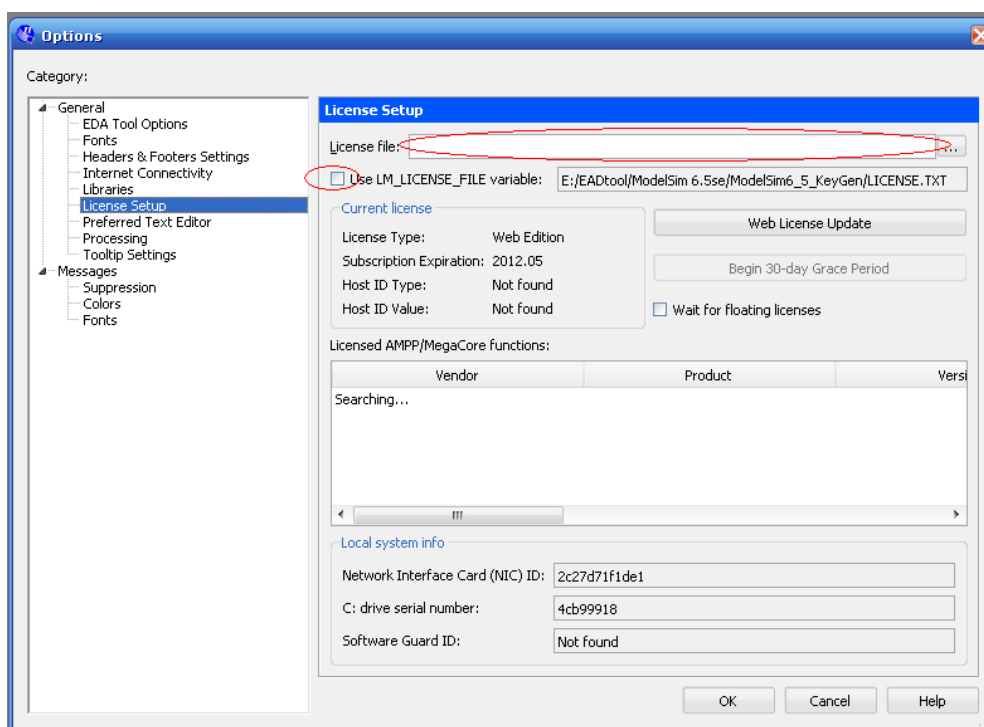


因为 Quartus II Web Edition 是完全免 license 的,但是有些网友会遇到安装好软件仍无法正常编译使用的情况,笔者也曾遇到过。这里提醒大家注意,如果 PC 机上安装过其他版本

《圣经》箴言九 11 “敬畏耶和华是智慧的开端,认识至胜者便是聪明。”



的 Quartus II, 很可能会出现无法正常使用 Quartus II Web Edition 的情况, 这很可能是 License Setup 隐射到了不正确的 license 文件造成的。大家可以打开菜单栏的 Tools→License Setup, 如图所示, 这里请大家确保 License file 后面是空白的 (因为不需要 license, 如果设置了某个 license 路径反而让软件无法使用), 也不要勾选 Use LM_LICENSE_FILE variable。

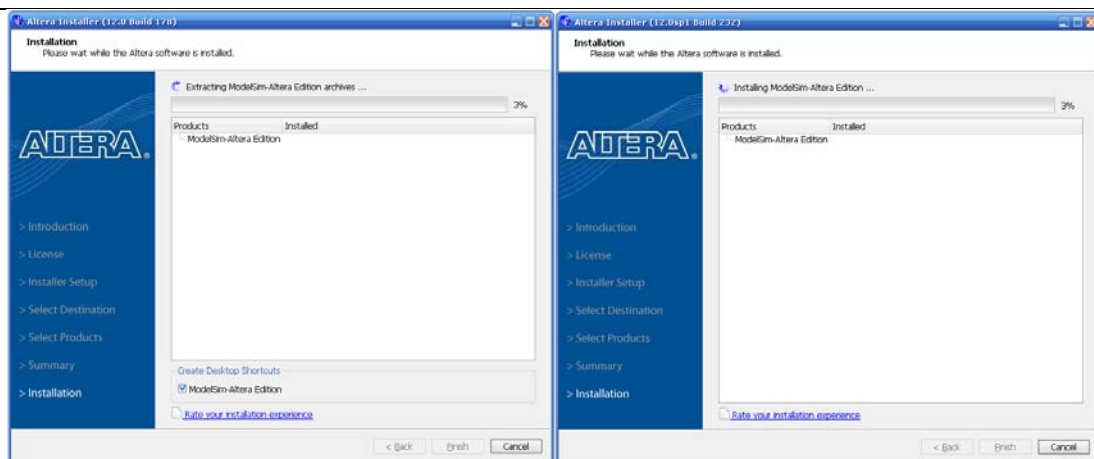


那么就此 Quartus II Web Edition 就安装好并可以正常使用了。

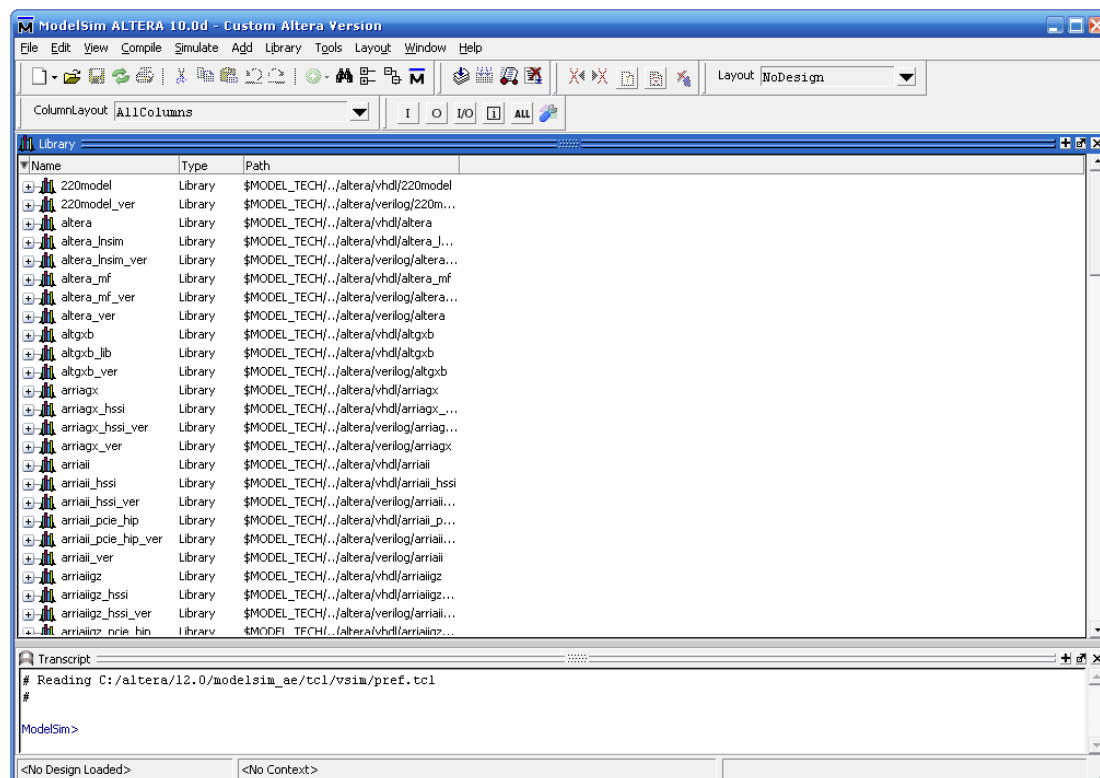
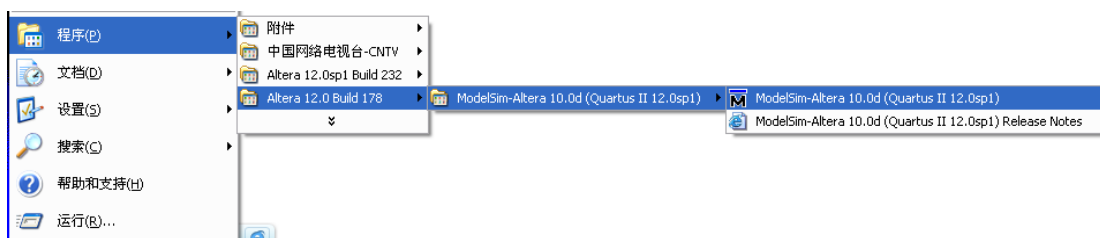
3.2.4 ModelSim 的安装

接下来我们要分别安装已经下载好的可执行文件 12.0_178_modelsimeae_windows.exe (先安装) 和 12.0sp1_232_modelsimeae_windows.exe。

与 Quartus II Web Edition 的安装类似, 点击可执行文件后, 在首先弹出的解压页面做好设置后点击 “Install” 开始解压操作, 接着一路 Next, 如图所示, 为 ModelSim-Altera Starter Edition 安装过程。



ModelSim-Altera Starter Edition 安装完毕, 双击桌面的快捷菜单或通过开始菜单进入软件主界面。



至此, 我们也就安装好了 ModelSim-Altera Starter Edition。



3.3 USB Blaster 驱动安装

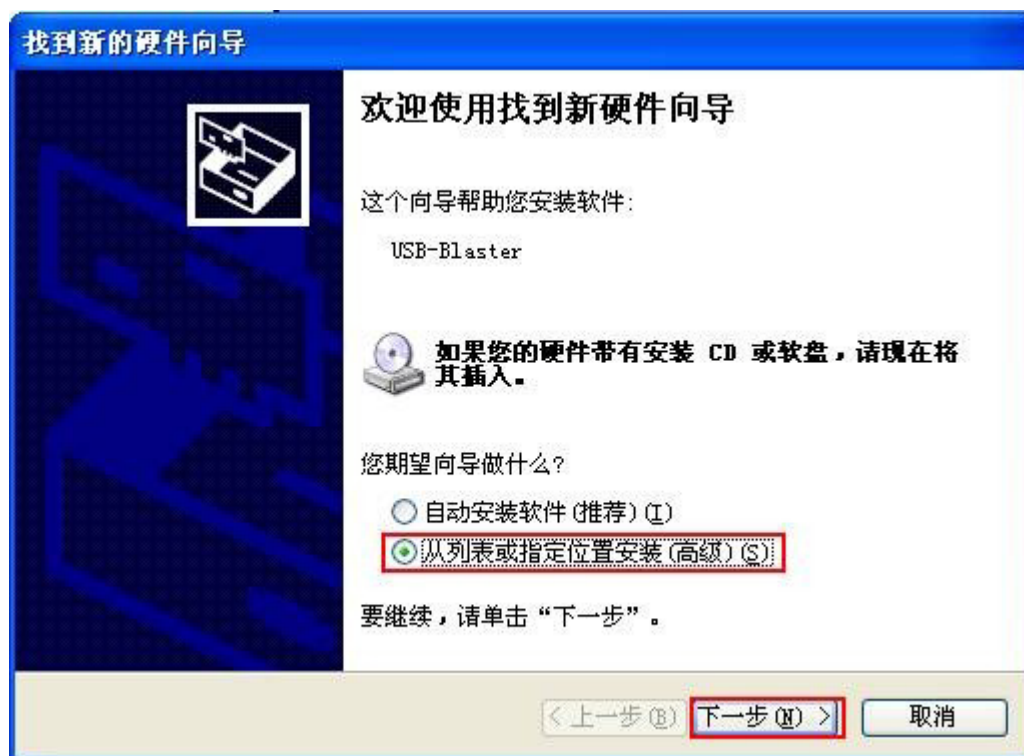
在电脑插入 USB Blaster 后, 屏幕右下角出现如下提示。



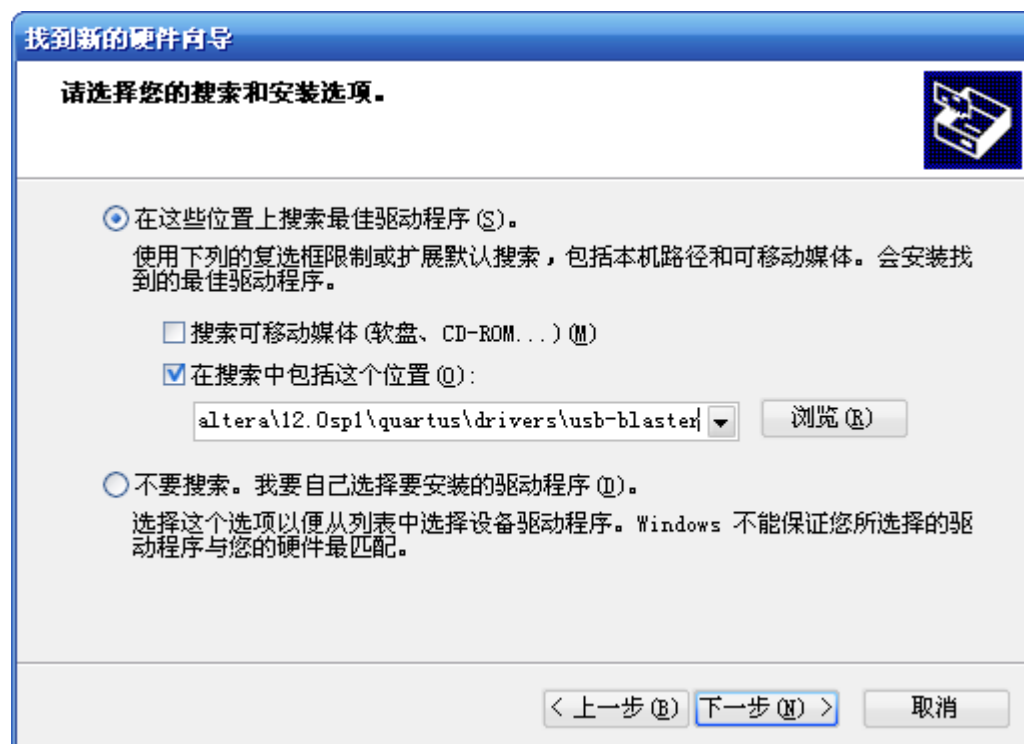
接着出现找到新的硬件向导。



选中从列表或指定位置安装 (高级) 后点击下一步。



在此步中选择在搜索中包括这个位置, 指向 Quartus 安装目录下的 drivers\usb-blaster 目录 (请选择你实际的安装目录) 后, 点击下一步便开始安装, 直到提示完成安装。





4 FPGA 的下载配置

4.1 FPGA 的上电启动原理

前面已经强调过, FPGA 是基于 RAM 结构的, 当然了, 也有基于 FLASH 结构的, 但 RAM 结构的是主流, 也是我们讨论的重点。而 RAM 是易失存储器, 在掉电后保存在上面的数据就丢失了, 重新上电后需要再下载一次才可以。因此, 我们肯定不希望每次重新上电后都去用 PC 下载一次, 工程实现也不允许我们这么做。所以, 通常 FPGA 旁边都有一颗配置芯片, 它通常是一片 FLASH, 或者是并行或者是串行接口的。不管是串行还是并行的 FLASH, 它们的启动加载原理基本相同, 后面我们会专门讨论。

FPGA 器件有三类配置下载方式: 主动配置方式 (AS)、被动配置方式 (PS) 和最常用的基于 JTAG 的配置方式。AS 和 PS 模式主要是将 bit 流下载到配置芯片中; 而 JTAG 模式则既能将代码下载到 FPGA 中直接在线运行 (速度快, 调试时优选), 也能够通过 FPGA 将 bit 流下载到配置芯片中。我们的 SF-CY3 就只预留了 JTAG 接口。

AS 配置方式:

AS 由 FPGA 器件引导配置操作过程, 它控制着外部存储器和初始化过程, EPCS 系列配置芯片如 EPCS1, EPCS4 配置器件专供 AS 模式, 目前只支持 Cyclone/ Cyclone II/ Cyclone III 系列。使用 Altera 串行配置器件来完成, Cyclone 器件处于主动地位, 配置器件处于从属地位。配置数据通过 DATA0 引脚送入 FPGA。配置数据被同步在 DCLK 输入上, 1 个时钟周期传送 1 位数据。

PS 配置方式:

PS 则由外部计算机或其它控制器控制配置过程。通过加强型配置器件 (EPC16, EPC8, EPC4) 等配置器件来完成, 在 PS 配置期间, 配置数据从外部储存部件, 通过 DATA0 引脚送入 FPGA。配置数据在 DCLK 上升沿锁存, 1 个时钟周期传送 1 位数据。

JTAG 配置方式:

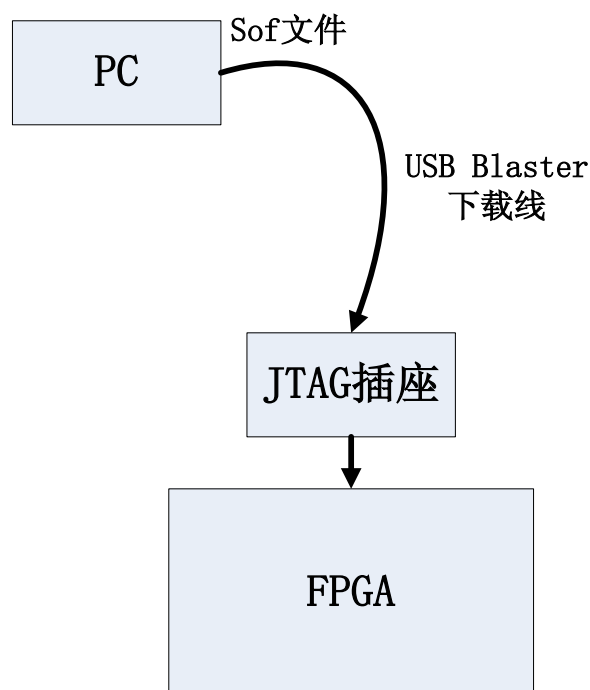
JTAG 接口是一个业界标准, 主要用于芯片测试等功能, 使用 IEEE Std 1149.1 联合边界扫描接口引脚, 支持 JAM STAPL 标准, 可以使用 Altera 下载电缆或主控器来完成。

FPGA 在正常工作时, 它的配置数据存储在 SRAM 中, 加电时须重新下载。在实验系统中, 通常用计算机或控制器进行调试, 因此可以使用 PS。在实用系统中, 多数情况下必须

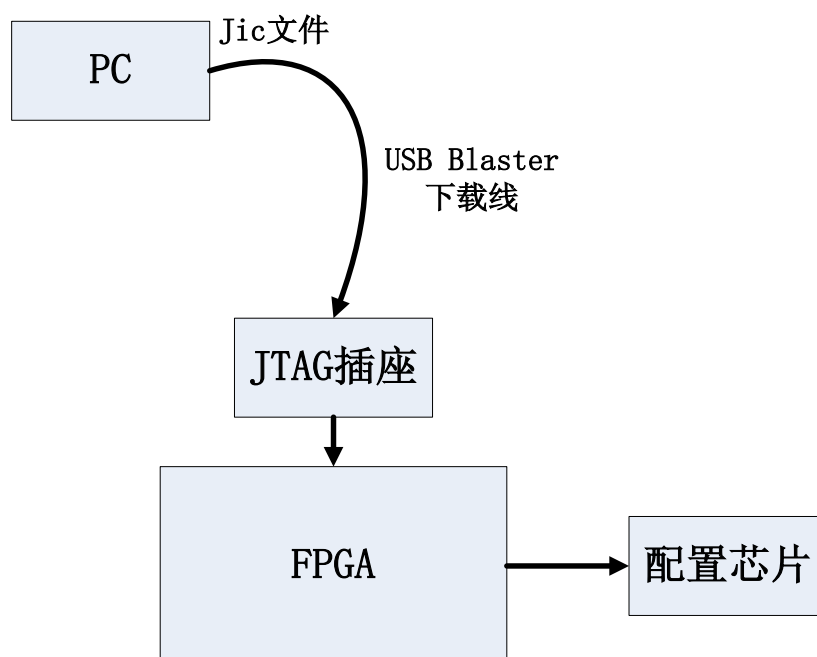


由 FPGA 主动引导配置操作过程, 这时 FPGA 将主动从外围专用存储芯片中获得配置数据, 而此芯片中 FPGA 配置信息是用普通编程器将设计所得的 pof 格式的文件烧录进去。

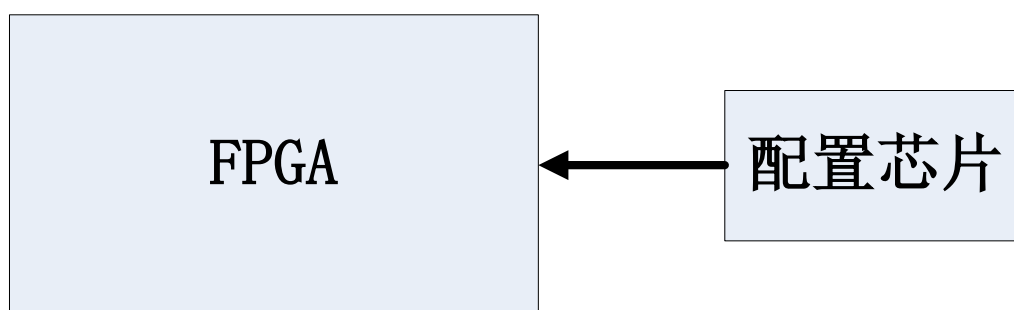
JTAG 模式在线下载 FPGA 的原理如图所示, PC 端的 Quartus II 软件通过下载线缆将 bit 流 (sof 文件) 下载到 FPGA 内部, 下载完成后 FPGA 中立刻执行下载代码, 速度很快, 非常适合调试。



FPGA 下载数据到配置芯片的原理如图所示, PC 端的 Quartus II 软件通过下载线缆将 bit 流 (jic 文件) 下载到配置芯片中。由于配置芯片和 JTAG 接口都是分别连接到 FPGA 的, 他们不是直接连接, 所以配置文件从 PC 先是传送到 FPGA, 然后 FPGA 内部再转送给配置芯片, 这个过程 FPGA 相当于起到一个桥接的作用。



看完 JTAG 模式下在线配置 FPGA 和烧录配置芯片的原理,我们再了解一下 FPGA 上电初始的配置过程。FPGA 上电后,内部的控制器首先工作,确认当前的配置模式,如果是外部配置芯片启动,则通过和外部配置芯片的接口(如我们的 SPI 接口)将配置芯片的数据加载到 FPGA 的 RAM 中,配置完成后开始正式运行。当然了,有人可能在想, JTAG 在线配置是否和配置芯片加载相冲突呢?非也, JTAG 在线配置的优先级是最高的,无论此时 FPGA 中在运行什么逻辑,只要 JTAG 下载启动,则 FPGA 便停下当前的工作,开始运行 JTAG 下载的新的配置数据。



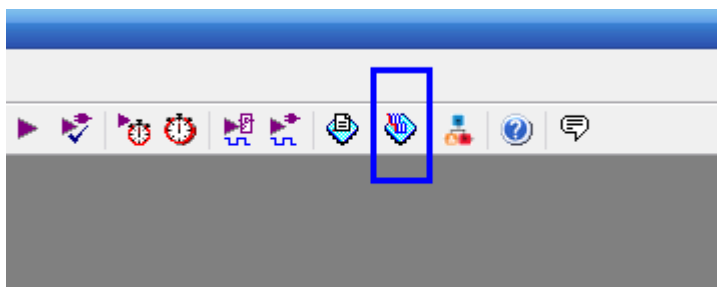
4.2 JTAG 在线烧录 FPGA

下面我们以配套例程中的 ex0 例程(LED 闪烁)做示范,看看 JTAG 模式是如何执行下载操作的。

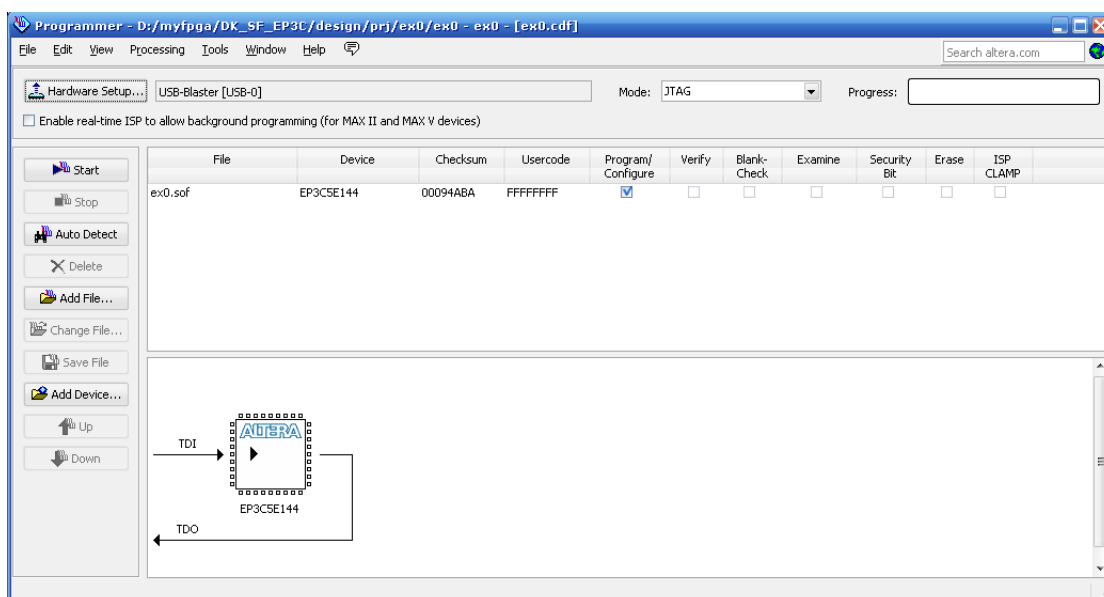
《圣经》箴言九 11 “敬畏耶和华是智慧的开端,认识至胜者便是聪明。”



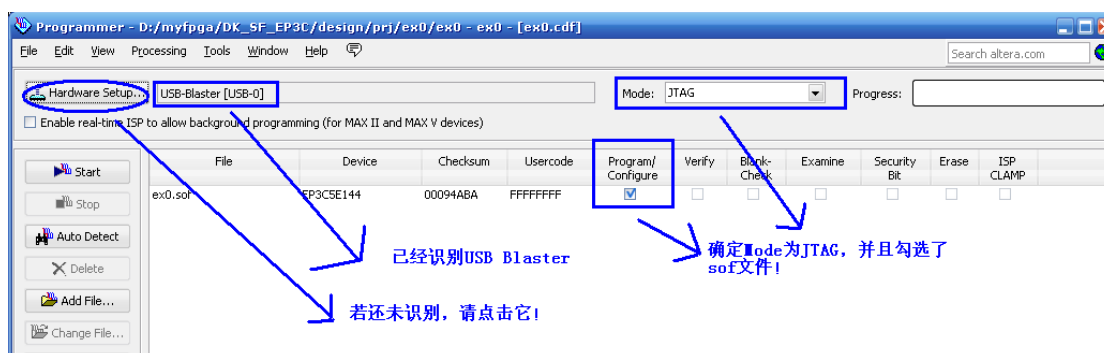
连接好 USB Blaster 下载线。给 SF-CY3 开发板上电, 同时打开 prj\ex0 目录下的工程 (双击 ex0.qpf)。点击菜单栏的 Programmer 按钮, 进入下载配置页面,



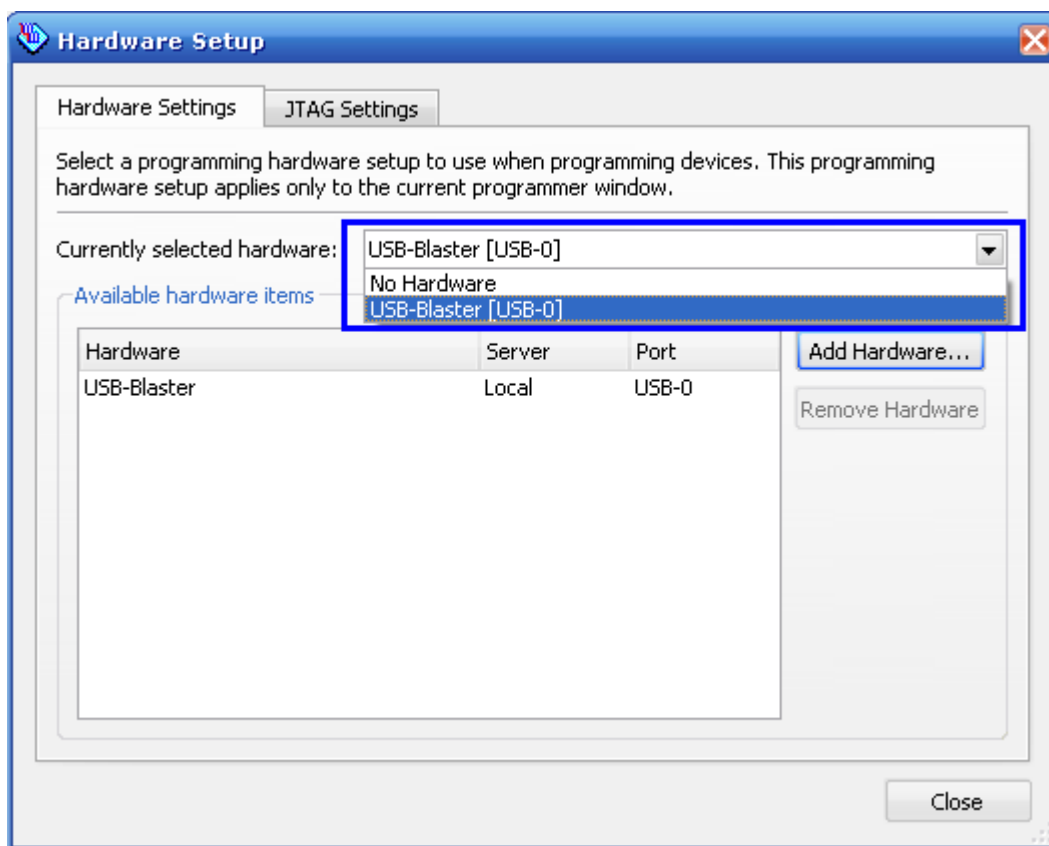
进入下载页面如下。



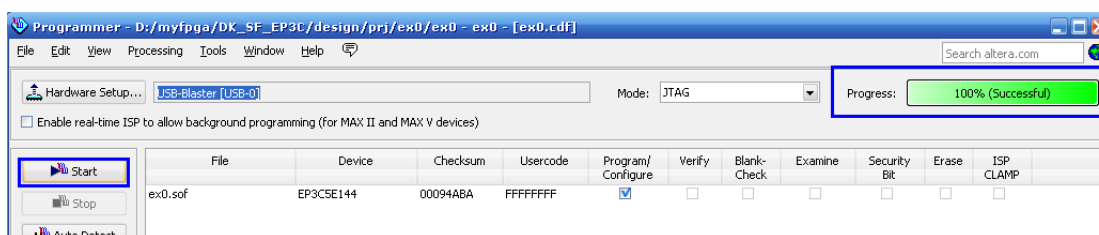
然后确认 Quartus II 是否识别了 USB Blaster 下载线。若没有识别, 则按照图示点击左上角的 Hardware Setup...。



在弹出的 Hardware Setup 页面里, 选择当前硬件为 USB Blaster, 然后 close。如果当前硬件里面没有 USB Blaster 选项, 首先确认硬件上是否已经把 USB Blaster 和 PC 连接好, 然后再尝试多次拔插一下看看, 或者重新启动 Quartus II 软件看看。



前面的步骤都确定好，直接点击右侧的 **Start** 按钮就可以启动下载操作，观察左上角的 **Process** 是否会从 0 到 100%。最后我们看看 SF-CY3 核心板上的 D1 指示灯是否很欢快的开始闪烁了。



JTAG 模式主要是将工程编译生成的.sof 烧录到 FPGA 中，如果下载完成后断电重新上电，那么你会看到你刚下载进行的代码不见了。（注意，如果配置芯片本身的代码就是闪烁灯，那么你重新上电后肯定还是闪烁灯，如果大家要看看是否真的重新上电后 JTAG 在线下载的数据丢失了，那么不妨按照后面的 JTGA 烧录配置芯片的步骤烧录一个不闪烁灯的代码，然后再做前面的下载看看）

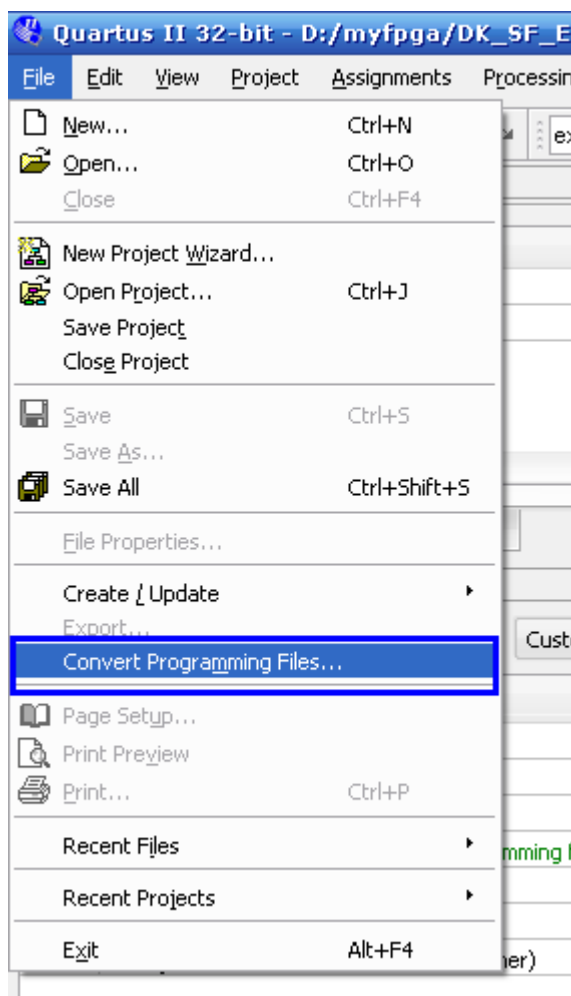


4.3 JTAG 烧录配置芯片

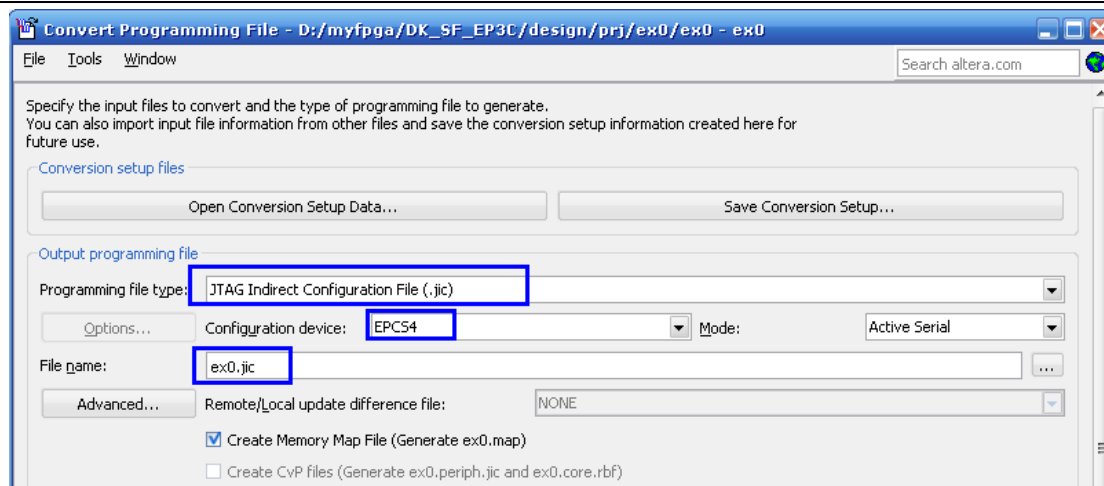
其实和一般的下载方式相比,这种下载方需要先把*.sof 文件转成*.jic 文件,然后在 JTAG 模式下选择*.jic 文件下载即可。

首先,您的工程必须编译并产生一个包含 FPGA 配置数据的 SRAM 目标文件(*.sof)。默认情况下 Quartus II 在编译后都会产生*.sof 的目标文件。

进入转换目标文件窗口,点击 Quartus II 软件的 File→Convert Programming Files...



弹出编程转换窗口如下图所示。



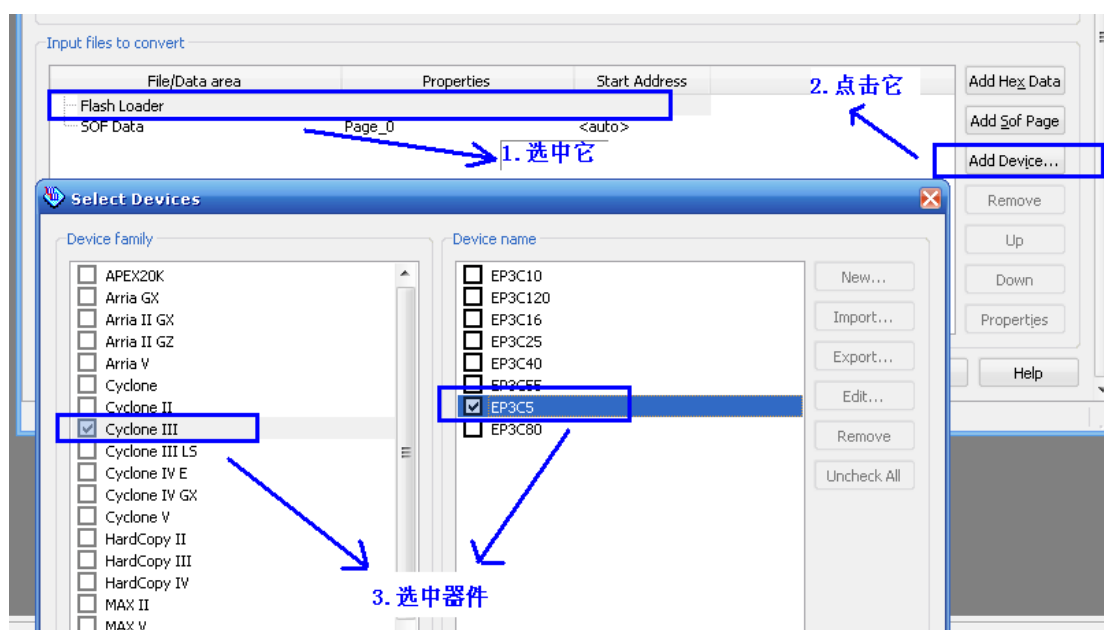
需要对上述窗口做如下设置:

Output programming file 下的 Programming file type: 选择我们需要转换的文件类型 JTAG Indirect Configuration File (.jic)。

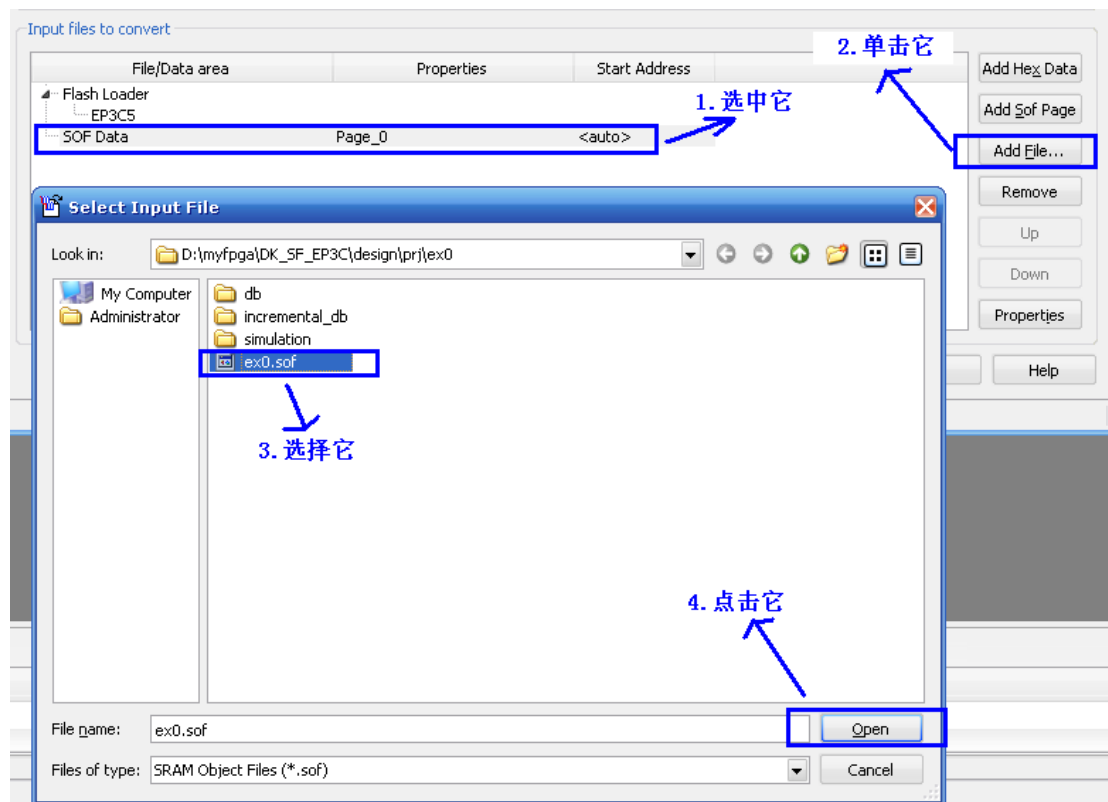
Configuration device: 选择我们 SF-CY3 开发板上使用的配置器件 EPCS4 (和 M25P40 完全兼容的 SPI FLASH)。

File name: 输入转换后的文件名, 我们命名为 ex0.jic。

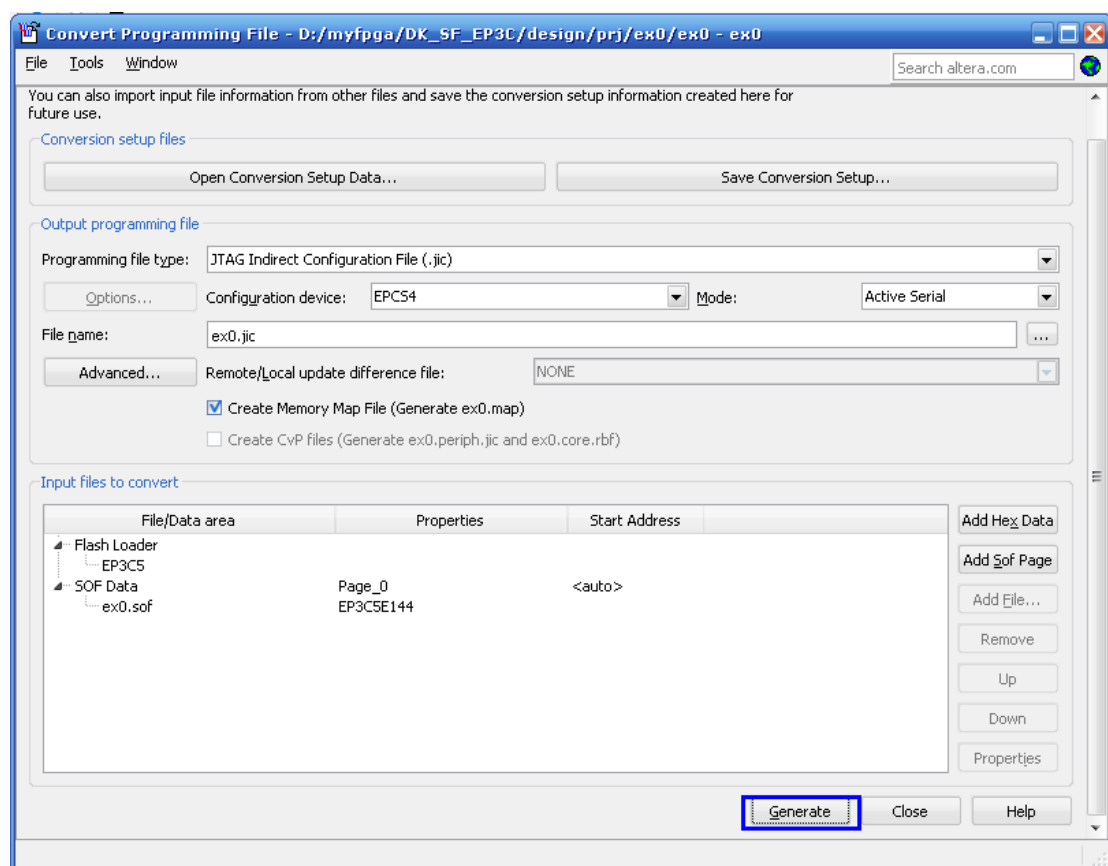
在后面的 Input files to convert 中, 首先单击 Options 行, 然后右侧的 Add File...选项高亮, 单击它。在弹出的窗口中选择 Cyclone III→EP3C5, 然后点击 OK。



再单击 SOF Data 行, 然后右侧的 Add File...高亮, 单击它。在弹出的窗口中选择*.sof 文件, 这里就选择 ex0.sof。



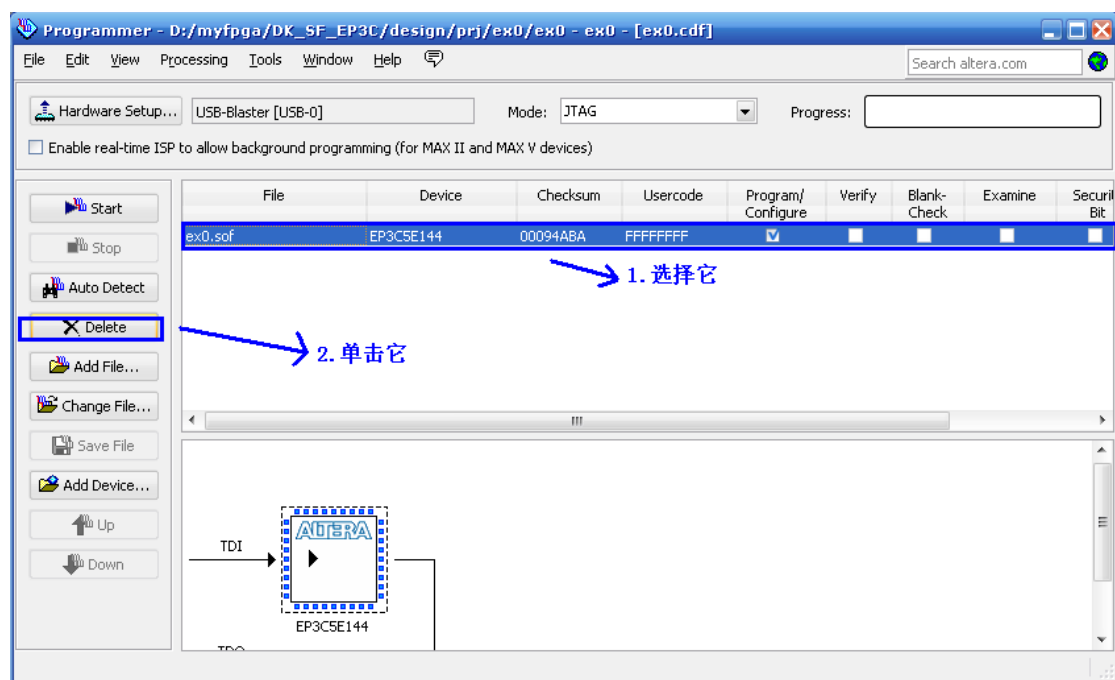
最后设置如下。



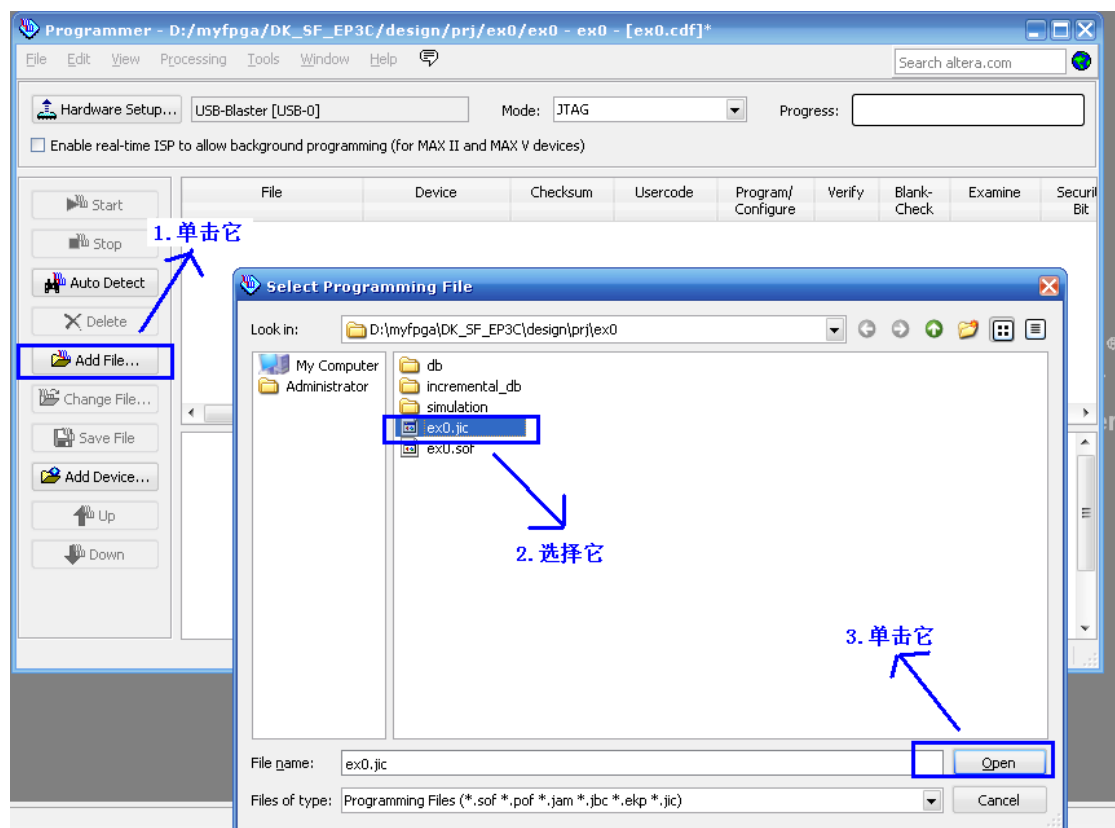
完成设置, 点击 **Generate** 生成*.jic。



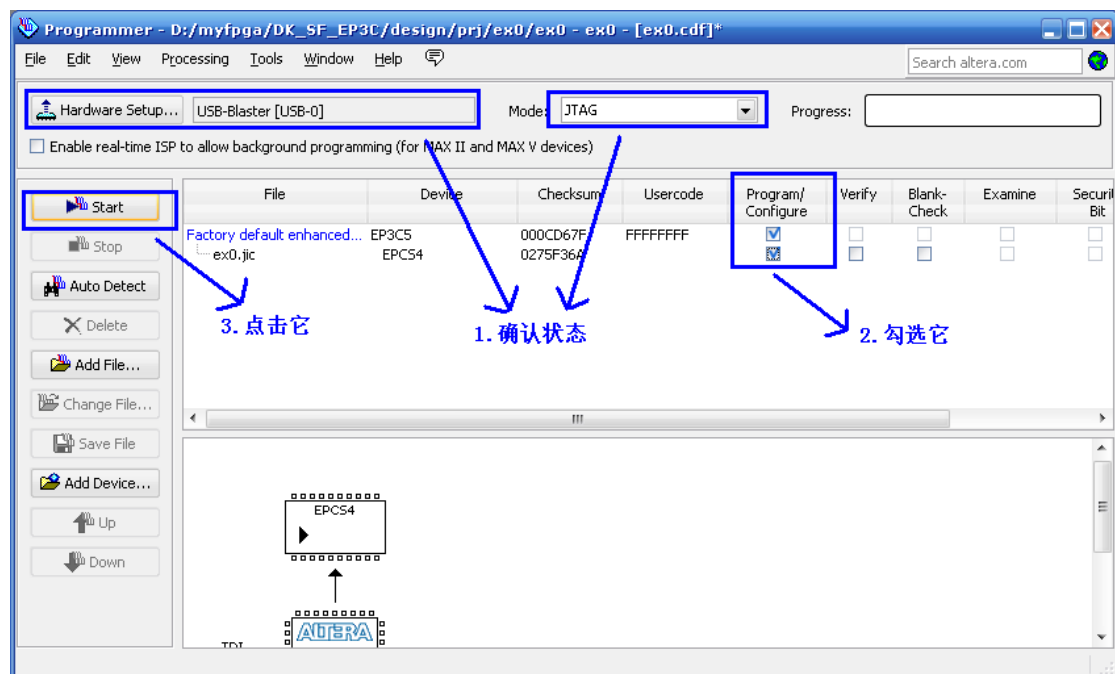
回到下载页面，我们先把之前的 sof 文件删除。



接着选择刚才生成的 jic 文件。



然后我们执行下载操作，同样的，我们等待进度条到 100%则表示下载完成，大家注意 jic 文件的下载要比 sof 文件的下载慢很多，要近 10 秒才能完成。



完成下载后，SF-CY3 板子处于不工作状态，需要重启 SF-CY3 开发板，我们就能看到刚才下载的代码已经生效运行了。

5 SF-CY3 工程实例

5.1 逻辑（Verilog）实例 1——LED 闪烁

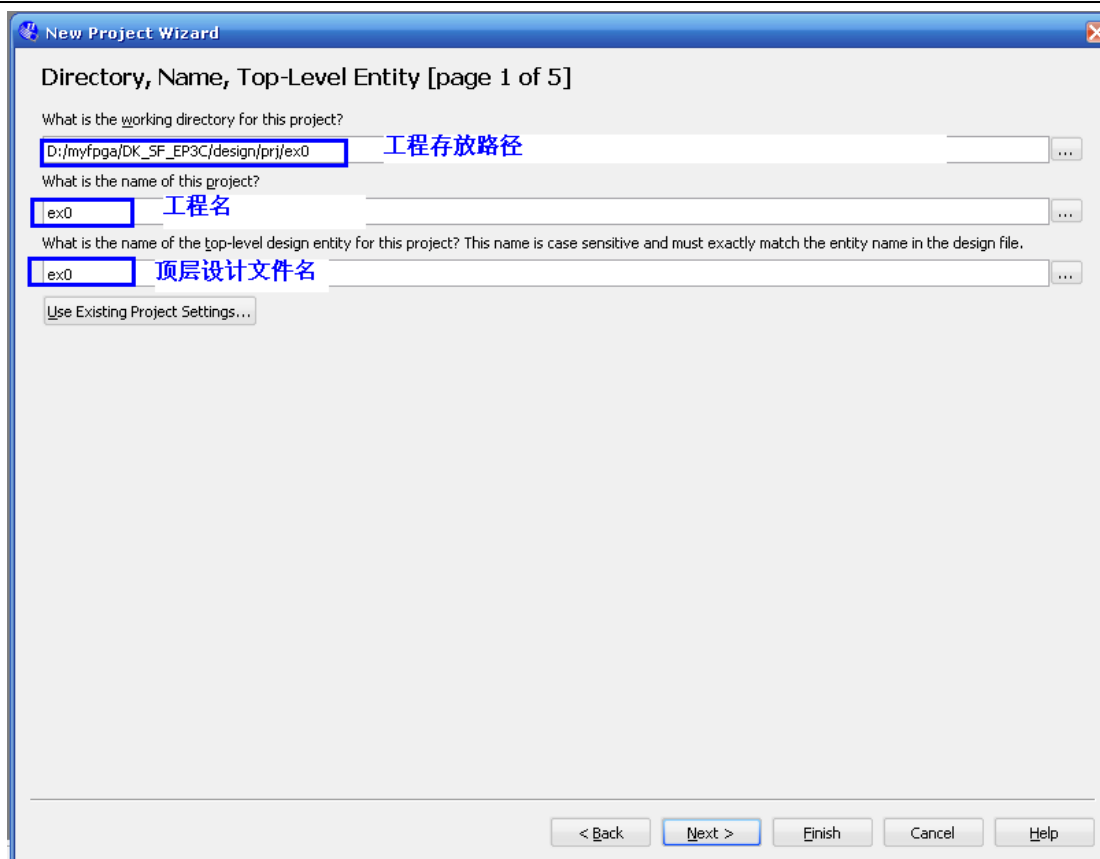
下面我们将以一个简单的实例开始带大家熟悉基于 Quartus II 的 FPGA 工程开发。在这个实例当中，我们将用到开发板上的一个 LED 指示灯。我们在 FPGA 内部产生一个分频计数器，对 FPGA 的输入 25MHz 时钟进行分频，得到的分频信号值就输出给 LED 指示灯，实现 1 秒或半秒让它闪烁一次。

5.1.1 新建工程

双击电脑桌面上的 Quartus II 12.0sp1 Web Edition (32-Bit)图标，或者单击开始→程序→Altera 12.0sp1 Build232→Quartus II 12.0SP1 Web Edition→Quartus II 12.0sp1 Web Edition，打开 Quartus II 软件。Quartus II 软件主界面如图所示，第一次打开通常默认由菜单栏、工具栏、工程文件导航窗口、编译流程窗口、主编辑窗口以及各种输出打印窗口组成。



下面我们要新建一个工程,在这之前建议大家硬盘中专门建议一个文件夹用于存储我们的 Quartus II 工程,这个工程目录的路径名应该只有字母、数字和下划线,不要包含中文和其他符号。在菜单栏上点击 **File**→**New Project Wizard...**,首先弹出了 Introduction 页面,点击 **Next** 进入 **Directory,Name,Top-Level Entity** 页面,如图所示。在该页面中,需要选择一个文件夹用于存储后续工程生成的所有相关文件,如这里将本实例的工程存储在了 D 盘的一个名叫 ex0 的文件夹下,“What is the name of this project?”下输入工程名,“What is the name of the top-level design entity for this project?”下输入工程顶层设计文件的名称。通常我们建议工程名和工程顶层文件保持一致,如这里统一命名 ex0。



接着点击 Next, Add Files 通常需要选择添加设计文件 (Verilog 或 VHDL 文件) 到工程中来, 因为我们是完全新建的工程, 没有任何预先可用的设计文件, 所以不用选择。接着点击 Next, 进入 Family & Device Setting 页面如图所示。该页面主要是选择元器件, 我们在 Family 中选择 Cyclone III 系列, Available device 中选择具体型号 EP3C5E144C8。接着再点击 Next 进入下一个页面。



New Project Wizard
 Family & Device Settings [page 3 of 5]

Select the family and device you want to target for compilation.

Device family

Family: Cyclone III
 Devices: All

Target device

☐ Auto device selected by the Fitter
☒ Specific device selected in 'Available devices' list
☐ Other: n/a

Show in 'Available devices' list

Package: Any
 Pin count: Any
 Speed grade: Any
 Name filter:
☒ Show advanced devices ☐ HardCopy compatible only

Available devices:

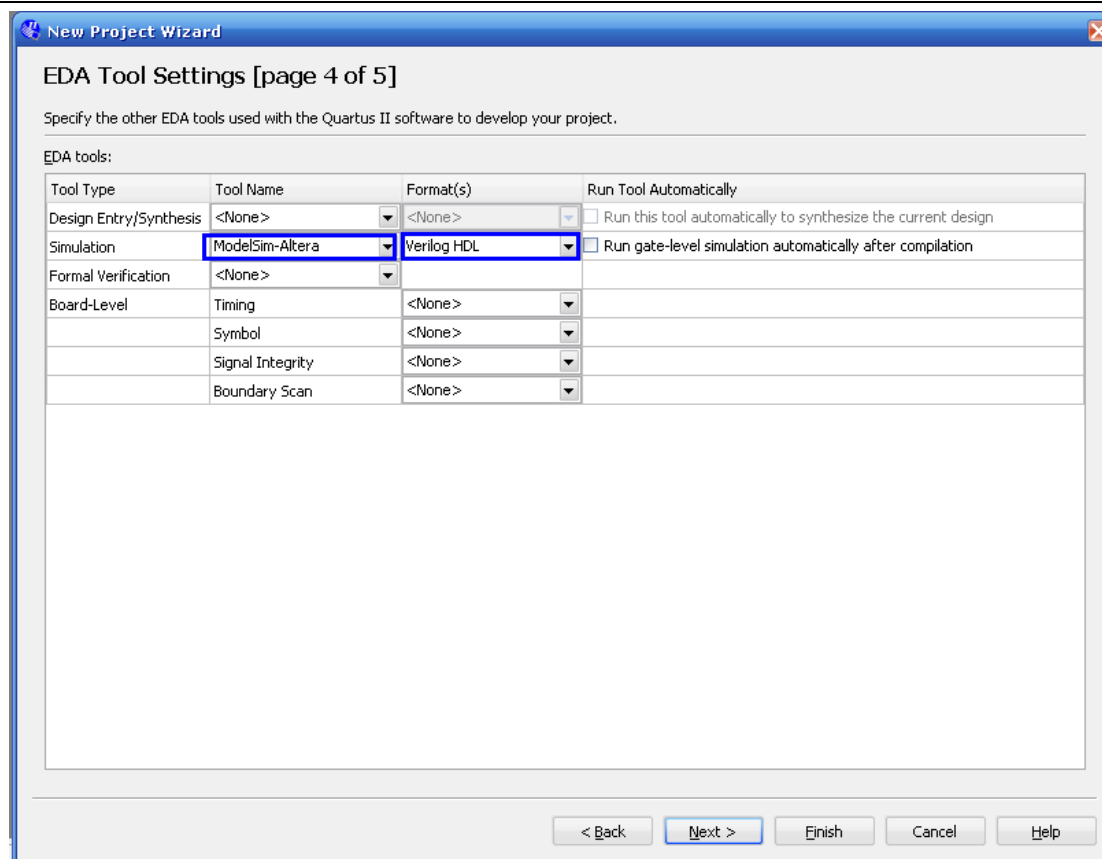
Name	Core Voltage	LEs	User I/Os	Memory Bits	Embedded multiplier 9-bit elements	PLL	Glc
EP3C5E144C7	1.2V	5136	95	423936	46	2	10
EP3C5E144C8	1.2V	5136	95	423936	46	2	10
EP3C5E144I7	1.2V	5136	95	423936	46	2	10
EP3C5F256C6	1.2V	5136	183	423936	46	2	10
EP3C5F256C7	1.2V	5136	183	423936	46	2	10
EP3C5F256C8	1.2V	5136	183	423936	46	2	10
EP3C5F256I7	1.2V	5136	183	423936	46	2	10

Companion device

HardCopy:
☐ Limit DSP & RAM to HardCopy device resources

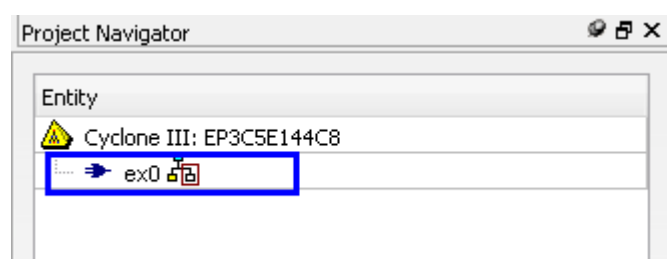
< Back Next > Finish Cancel Help

如图所示, 在 EDA Tool Setting 页面中, 可以设置工程各个开发环节中需要用到的第三方 (Altera 公司以外) EDA 工具, 我们只需要设置 Simulation 工具为 ModelSim-Altera, Format 为 Verilog HDL 即可, 其他工具不涉及, 因此都默认为 <None>。



完成这个页面的配置后，我们可以点击 **Next** 继续进入下一页面查看并核对前面设置的结果，也可以直接点击 **Finish** 完成工程创建。

工程创建完成后，如图所示，在 **Project Navigator** 下出现了我们所选择的器件以及顶层文件名，但是实际上此时我们并未创建工程的顶层设计文件，只不过给他命名为了 **ex0**。我们若双击试图打开 **ex0** 文件，系统马上会弹出“Can't find design entity “ex0””的错误提示。

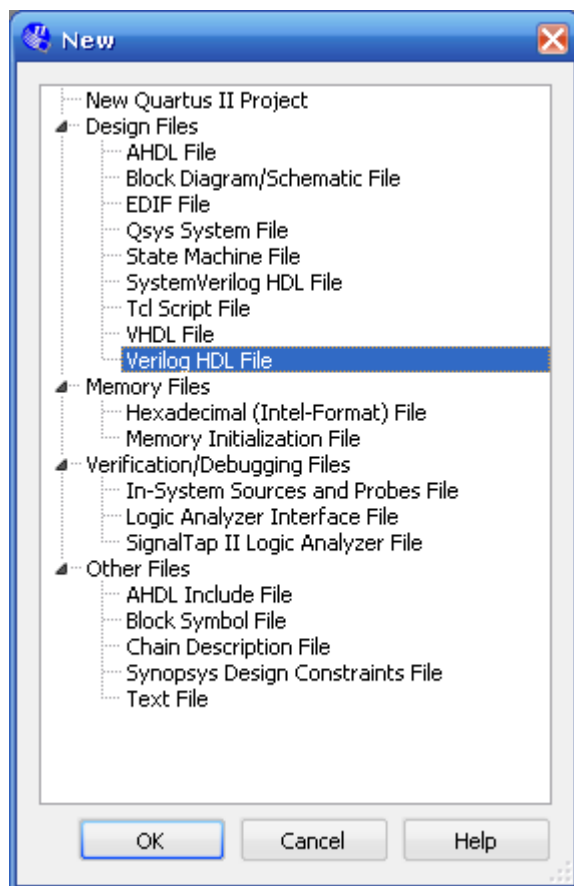


5.1.2 输入源码

下面我们就来创建工程顶层文件，我们可以点击菜单栏的 **File→New...**，然后弹出如图所示的新建文件窗口，在这里我们可以选择各种需要的设计文件格式。可以作为工程顶层设



计文件的格式主要在 **Design Files** 类别下, 我们选择 **Verilog HDL File** (或者 **VHDL File**) 并单击 **OK** 完成文件创建。



在主编辑窗口中, 出现了一个新建的空白可编辑文件, 我们接着在该文件中输入实现实验功能的一段 **Verilog** 代码:

```
module ex0(
    clk, rst_n, led
);

input clk;
input rst_n;
output led;

reg[23:0] cnt;

always @(posedge clk or negedge rst_n)
    if(!rst_n) cnt <= 24'd0;
    else cnt <= cnt+1'b1;
```



```
assign led = cnt[23];  
  
endmodule
```

该代码使用 FPGA 外部时钟不停的对 24bit 的计数器 cnt 做循环计数,然后将这个计数器 cnt 的最高位 bit23 的值赋给 led 信号。我们可以简单的计算一下这个 LED 的闪烁频率: 24bit 即 $16 * (2 \text{ 的 } 20 \text{ 次方})$, 用它去 乘以 40ns (即 25MHz 频率的对应周期), 得到的 LED 闪烁周期为 0.67108864s。

在这个刚创建的 Verilog 文件中输入代码后,快捷键 Ctrl+S 或点击 File→Save 后则会弹出一个对话框提示输入文件名和保存路径,默认文件名会和我们所命名的 module 名相一致,默认路径也会是当前的工程文件夹。我们通常也都采用默认设置进行保持即可。

自此,我们的工程创建和设计输入工作已经完成。但是为了验证一下设计输入的代码的基本语法是否正确,可以点击 Flow → Compilation 下的 Analysis & Elaboration 按钮,如图所示。同时我们可以输出打印窗口的 Processing 里的信息,包括各种 warning 和 Error。Error 是不得不关注的,因为 Error 意味着我们的代码有语法错误,后续的编译将无法继续;而 warning 则不一定是致命的,但很多时候 warning 中暗藏玄机,很多潜在的问题都可以从这些条目中寻找蛛丝马迹。当然了,也并不是说一个设计编译下来就不可以有 warning,如果设计者确认这些 warning 符合我们的设计要求,那么可以忽略它。

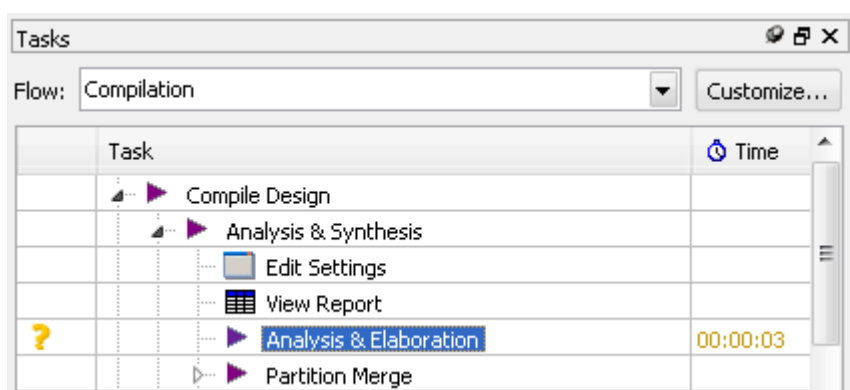


图 5.26

最后,在 Analysis & Elaboration 完成后,通常前面的问号会变成勾号,表示通过。

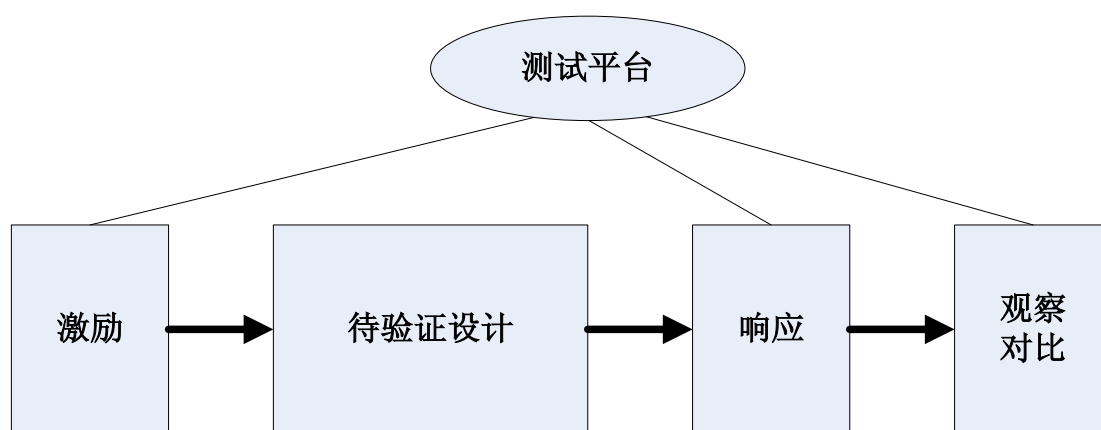
5.1.3 ModelSim 仿真

完成了前面基本的设计输入后,为了进一步的验证代码所实现功能的正确性,我们还需



要进行仿真测试。关于仿真的一些深入的介绍大家可以参考笔者的《深入浅出玩转 FPGA》一书笔记 10 的相关内容。为了让大家更好的理解什么是仿真、为什么要做仿真以及如何做仿真,这里将以 testbench 的基本概念进行简单的介绍。当然了,仿真其实在包含在 testbench 这个大概概念里面的。

所谓 testbench,即测试平台,详细的说就是给待验证的设计添加激励,同时观察它的输出响应是否符合设计要求。如图所示,测试平台就是要模拟一个和待验证设计接口的各种外围设备。

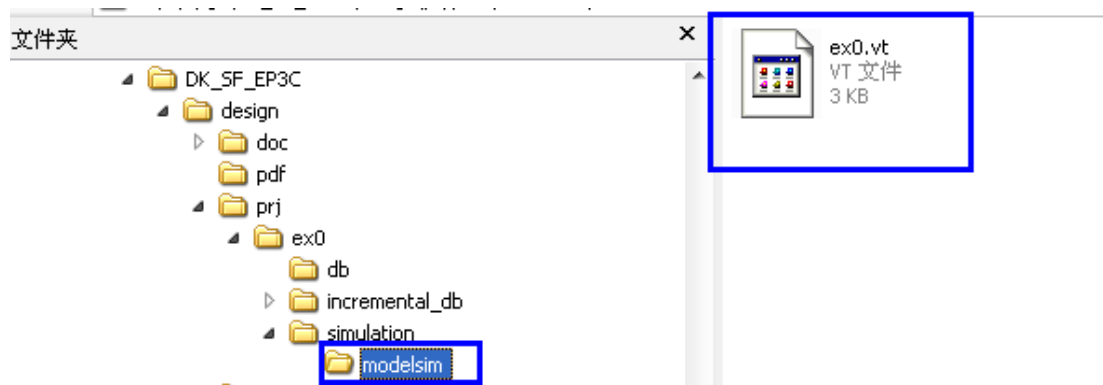


仿真测试是 FPGA 设计流程中必不可少的步骤。在今天的 FPGA 设计中,如果逻辑规模较大,一般都会使用到 IP 核或者 SOC 来加快 RTL 级设计,所以花费在仿真验证上的工作量往往能够占到这个开发流程的 70%。仿真测试的重要性可见一斑。

简单的补充了一些理论,下面我们就以当前的实例工程为例来建立一个仿真环境。这个仿真环境的最终主角当然是 ModelSim (Altera-ModelSim),但在此之前,我们还是需要把 ModelSim 工作所需的各种接口以及相关文件的关系梳理清楚。

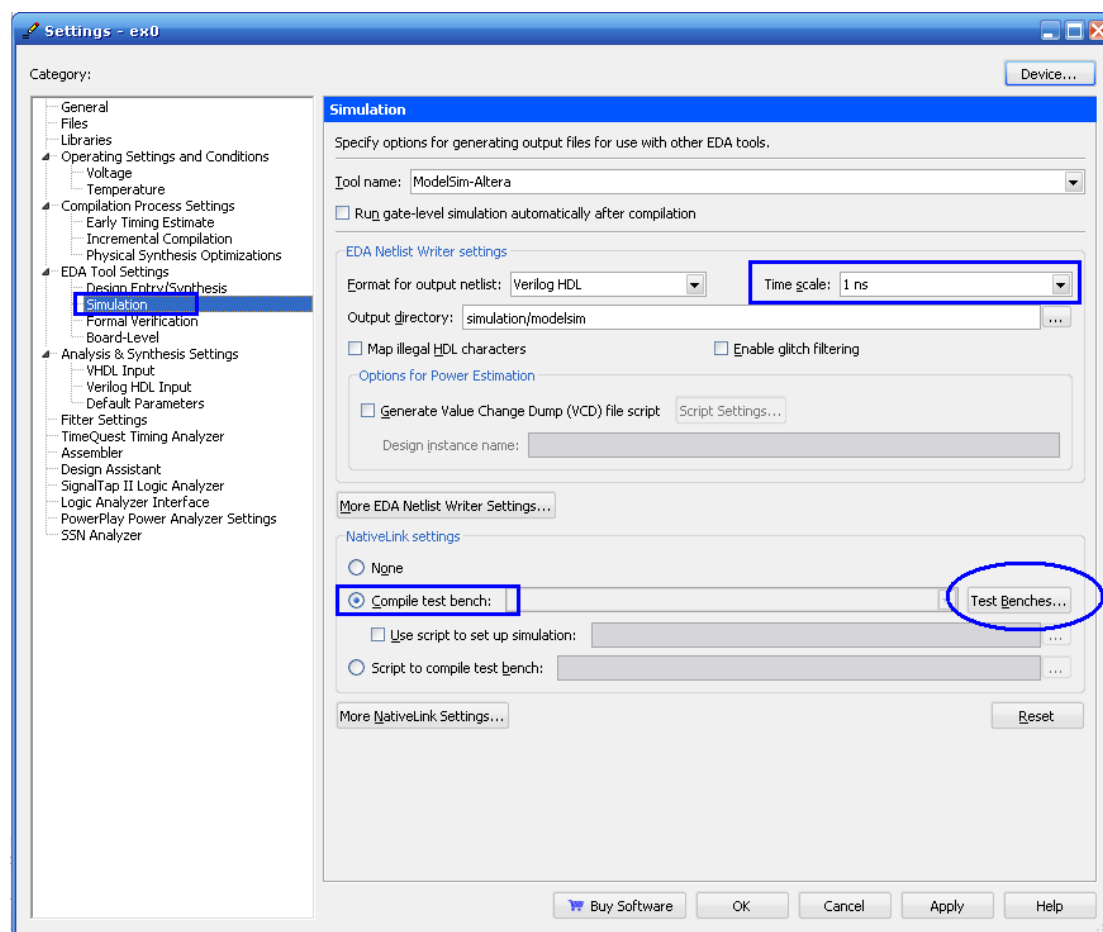
回到我们的 Quartus II 工具中,在前面创建工程向导中,我们其实已经设置了 Simulation 工具为 ModelSim-Altera,但是我们还需要做更详细的配置。首先我们可以点击菜单栏的 Processing→Start→Start Test Bench Template Writer,随后弹出提示“Test Bench Template Writer was successful”,那么我们就已经创建了一个 Verilog 测试脚本,在此脚本中,我们可以设计一些测试激励输入并且观察相应输出,借此我们就能够验证原工程的设计代码是否符合要求。

我们打开工程路径下的/simulation/modelsim 文件夹,可以看到一个名为 ex0.vt 的测试脚本文件创建了。



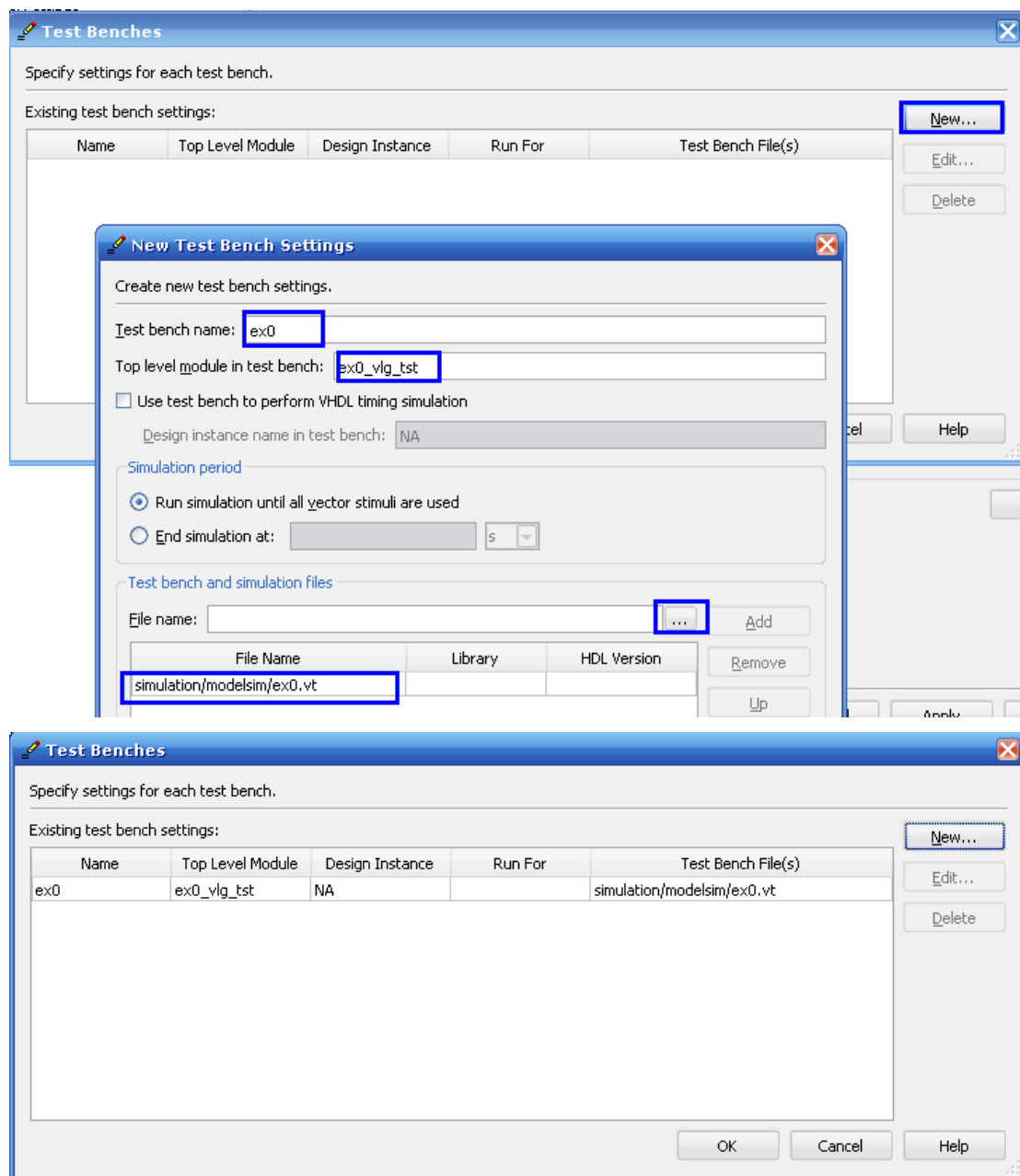
我们可以在 Quartus II 中打开这个文件，并且将其重新编辑如下：

完成测试脚本编写，我们接着需要打开菜单栏的 **Assigment→Settings** 选项，选择 **Category→EDA Tool Setting→Simulation**，在右边的相关属性中做如图所示的设置，在选中 **Comple test bench** 后，我们要点击后面的 **Test Benches...**按钮去选择刚才创建的测试脚本。



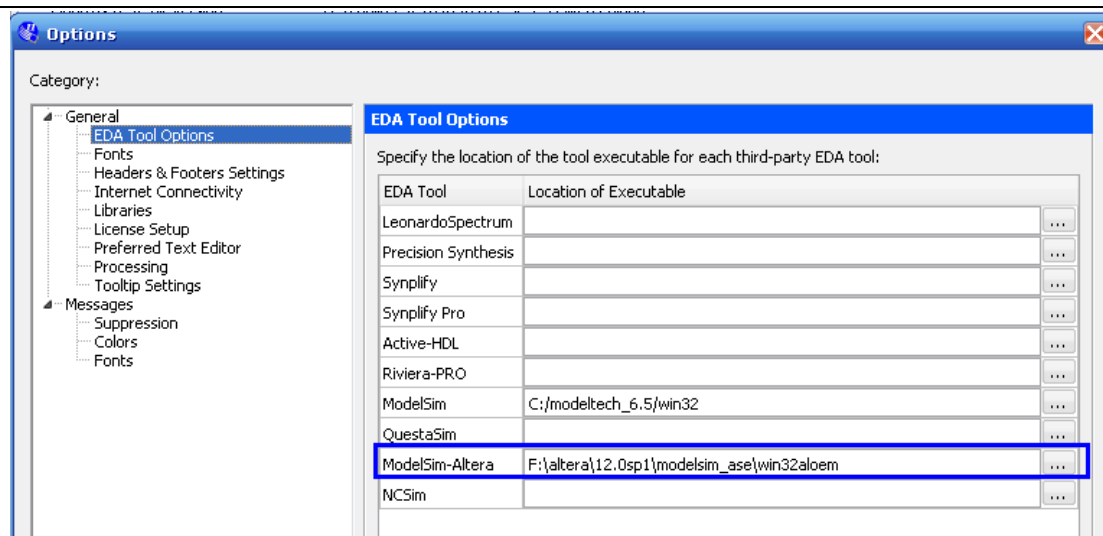


首先弹出上面的 **Test Benches** 窗口，然后我们可以点击 **New...** 按钮，接着便弹出下面的窗口。在此窗口中，我们根据实际情况输入 **Test bench name** 和 **Top level module in test bench** 的名称，接着还要在 **Test bench and simulation files** 下面选择测试脚本文件，然后 **Add** 添加到最下面的列表中。完成后点击 **OK**，我们便可看到 **Test Benches** 窗口的列表中出现了刚才添加的测试脚本相关信息，接着点击 **OK** 完成设置。

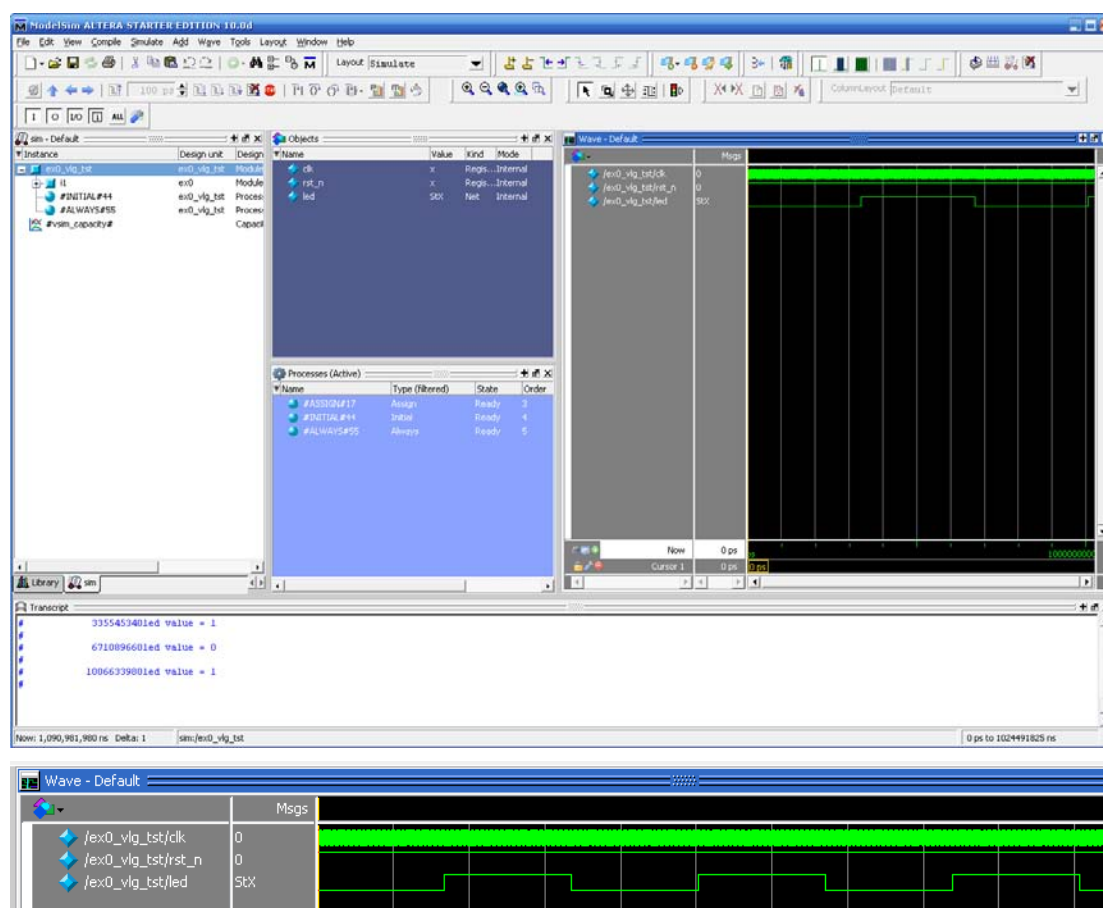


回到 **Setting** 中也点击 **OK** 完成所有相关设置。我们还需要打开菜单栏的 **Tools→Options** 配置页面，我们选择 **Category** 下的 **General→EDA Tool Options**，然后设置 **ModelSim-Altera** 软件安装路径（请根据实际安装时的路径进行设置）。当 **Quartus II** 调用 **ModelSim-Altera** 软件进行仿真时，会通过这里所设置的路径来查找并启动 **ModelSim-Altera**。

《圣经》箴言九 11 “敬畏耶和华是智慧的开端，认识至胜者便是聪明。”



仿真测试的所有准备工作就绪了，下面我们就可以一键完成仿真工作。点击菜单栏的 Tools→Run Simulation Tool→RTL Simulation。随后 ModelSim-Altera 便启动，如图 5.35 所示，这是 ModelSim-Altera 软件的工作界面。关于 ModelSim-Altera 软件的基本使用建议大家参考该软件菜单栏 Help 下自带的一些文档，尤其是 Help→PDF Documentation 里的几个文档。ModelSim-Altera 的功能也非常强大实用，如果要详细展开来探讨，恐怕也要专门写本书才可以。



《圣经》箴言九 11 “敬畏耶和华是智慧的开端，认识至胜者便是聪明。”

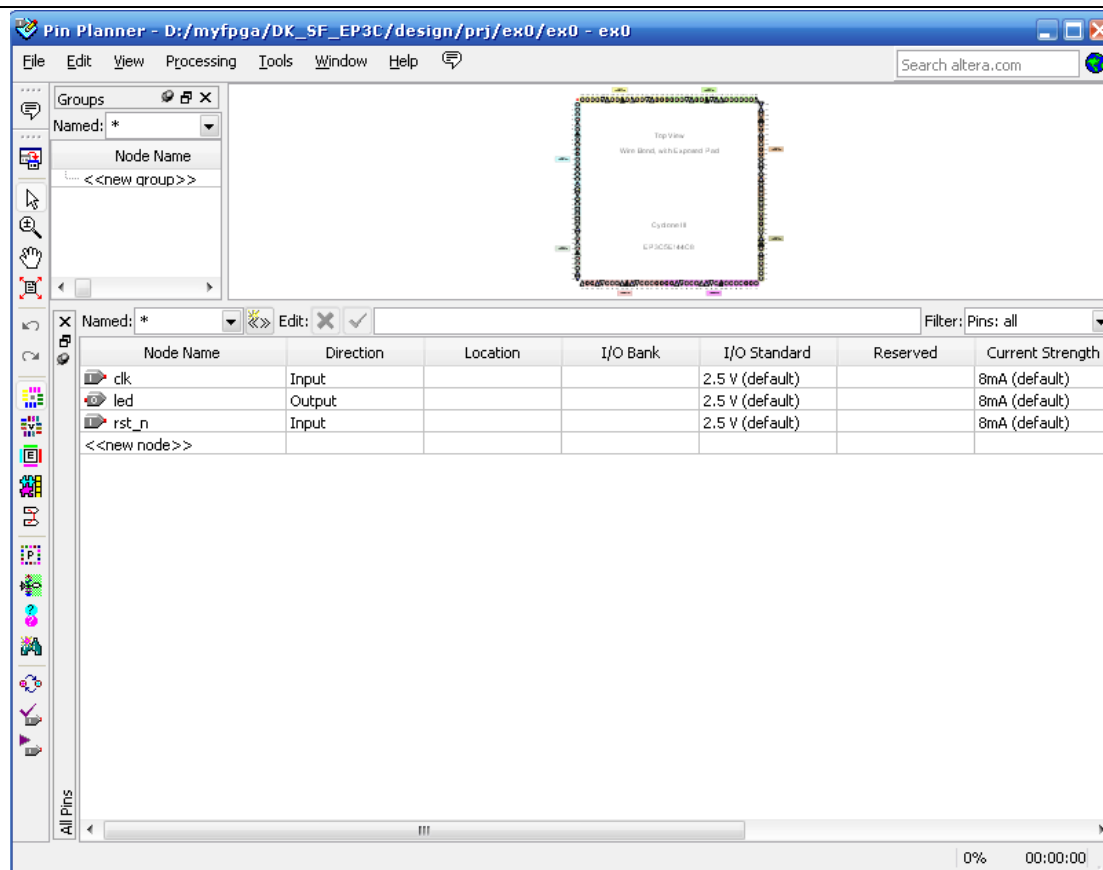


```
Transcript
#
# add wave *
# view structure
# .main_pane.structure.interior.cs.body.struct
# view signals
# .main_pane.objects.interior.cs.body.tree
# run -all
#
#                                Oled value = 0
#
#                                335545340led value = 1
#
#                                671089660led value = 0
#
#                                1006633980led value = 1
#
#                                1342178300led value = 0
#
#                                1677722620led value = 1
#
#                                2013266940led value = 0
#
#                                2348811260led value = 1
#
```

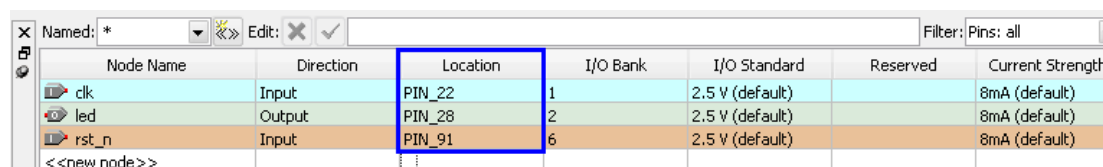
前面是仿真的结果,一方面我们可以通过观察波形,另一方面我们也可也看打印的信息。大家可以算算打印信息里面 led 值变化的周期,即 1-0-1 或者 0-1-0 的时间是否和我们设计的预期想一致。

5.1.4 管脚分配与编译

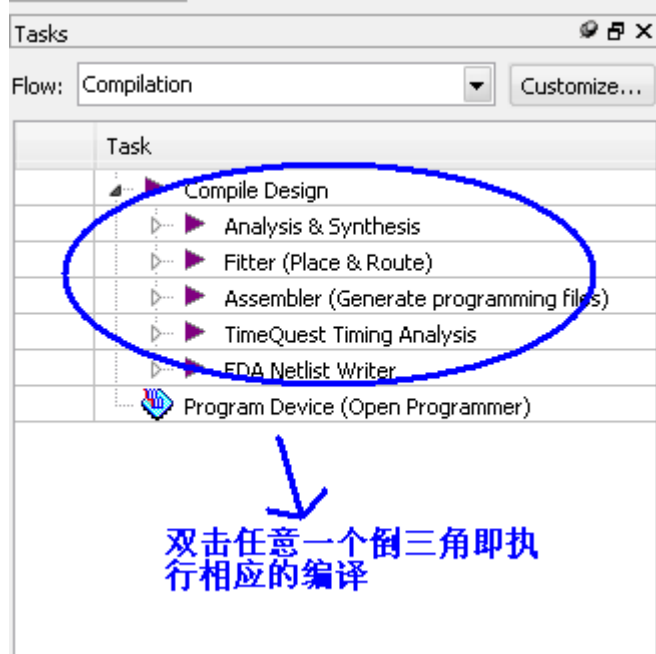
功能仿真没有问题后,我们可以点击菜单栏的 Assignments→Pin planner 对工程的 I/O 管脚进行分配,Pin Planner 界面如图所示。如果 Node Name 列里没有出现顶层文件的信号,那么建议先对整个工程执行一次编译。



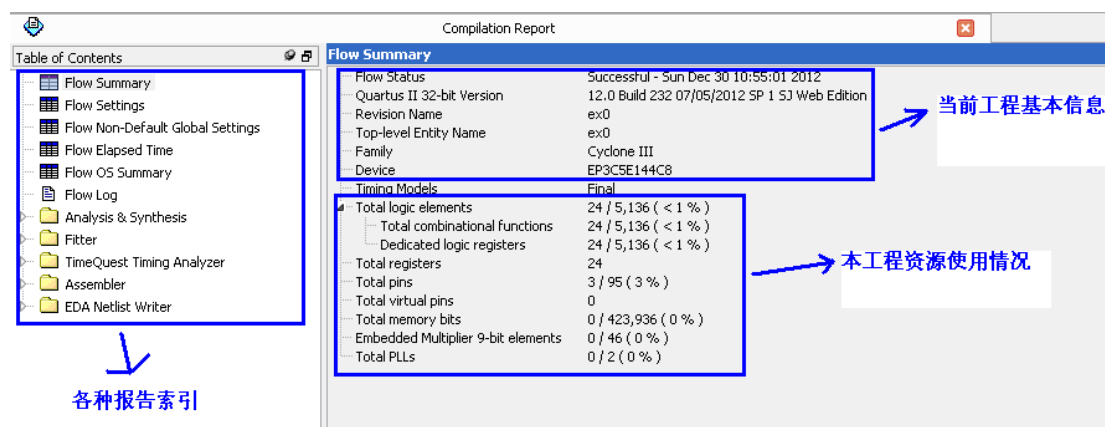
通过原理图，我们可以找到时钟管脚、复位管脚、LED 指示灯所连接的 I/O 管脚，将他们对应的管脚号输入到对应信号名的 Location 中。



接着我们回到 Quartus II 继续下面的流程。我们可以先对整个工程进行一次全编译，既可以点击工具栏的“Start Compilation”按钮，也可以在 Task→Compilation 中点击 Compile Design。

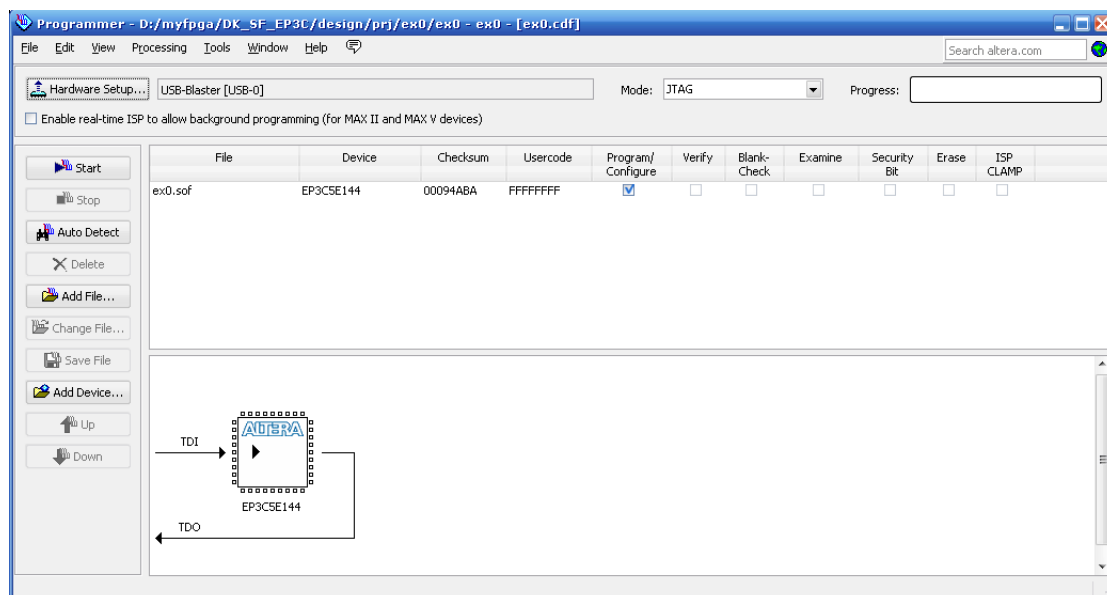


编译完成后会在主界面中弹出工程的编译报告。该报告的左侧罗列了各个阶段的各种设计相关报告索引，右侧则显示具体内容，默认的 Flow Summary 报告中则给出了该设计的最主要的一些资源利用情况。

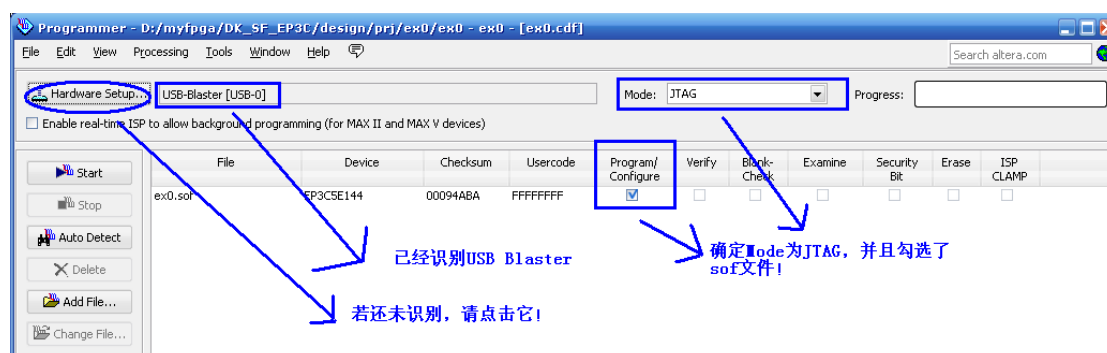


5.1.5 下载配置与板级调试

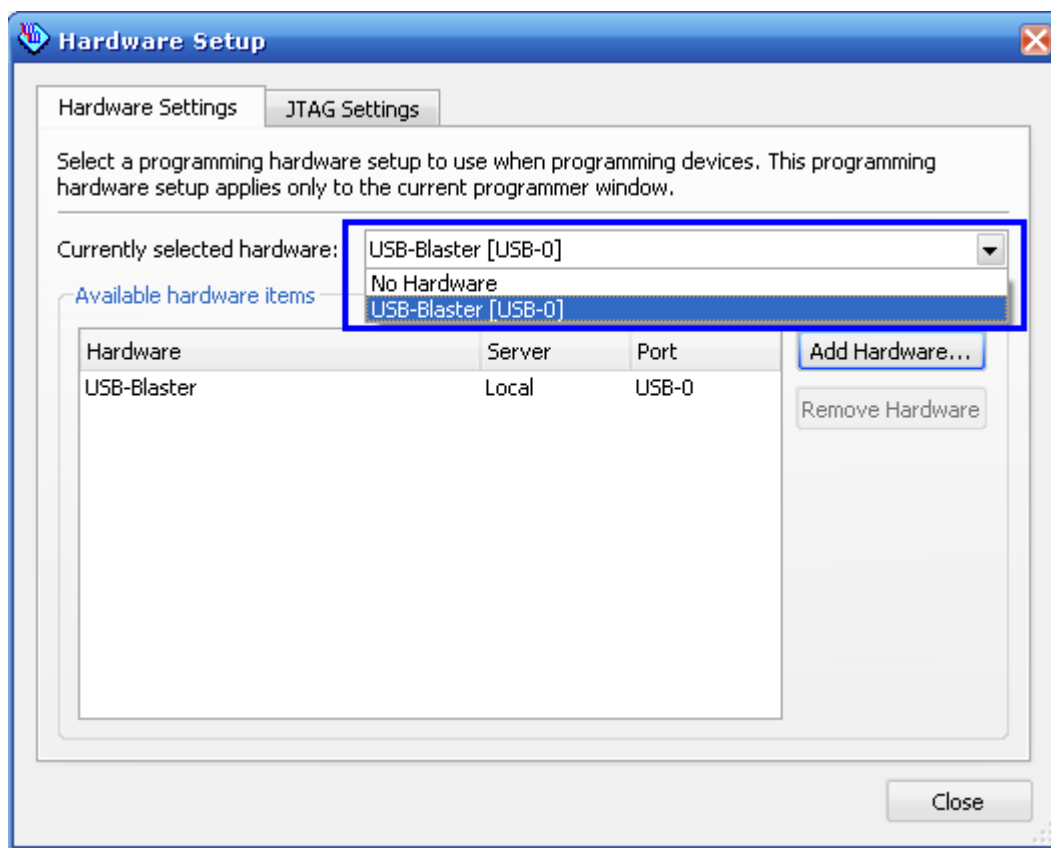
直接点击工具栏上的 Programmer 按钮或者点击菜单栏的 Tools→Programmer 选项进入下载界面。



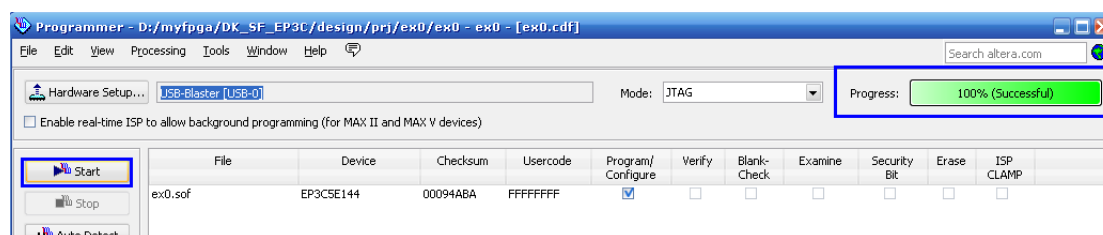
按照前面相关章节的操作方法连接好 USB blaster (PC 和 SF-CY3), 连接好电源, 按下电源开关给板子上电。然后确认 Quartus II 是否识别了 USB Blaster 下载线。若没有识别, 则按照图示点击左上角的 Hardware Setup...。



在弹出的 Hardware Setup 页面里, 选择当前硬件为 USB Blaster, 然后 close。如果当前硬件里面没有 USB Blaster 选项, 首先确认硬件上是否已经把 USB Blaster 和 PC 连接好, 然后再尝试多次拔插一下看看, 或者重新启动 Quartus II 软件看看。



前面的步骤都确定好, 直接点击右侧的 **Start** 按钮就可以启动下载操作, 观察左上角的 **Process** 是否会从 0 到 100%。最后我们看看 SF-CY3 核心板上的 D1 指示灯是否很欢快的开始闪烁了。



5.2 逻辑 (Verilog) 实例 2——PLL 配置

5.2.1 新建工程

我们这个实例和 LED 闪烁实例要达到一样的功能, 只不过这次不是直接使用 FPGA 外部输入的时钟做分频, 而是使用 PLL 处理过的时钟。本实例的主要目的是教会大家如何配置 Cyclone III 内部的 PLL 资源。



大家可以参照上一节新建一个工程，工程文件夹名和工程名为 ex1。

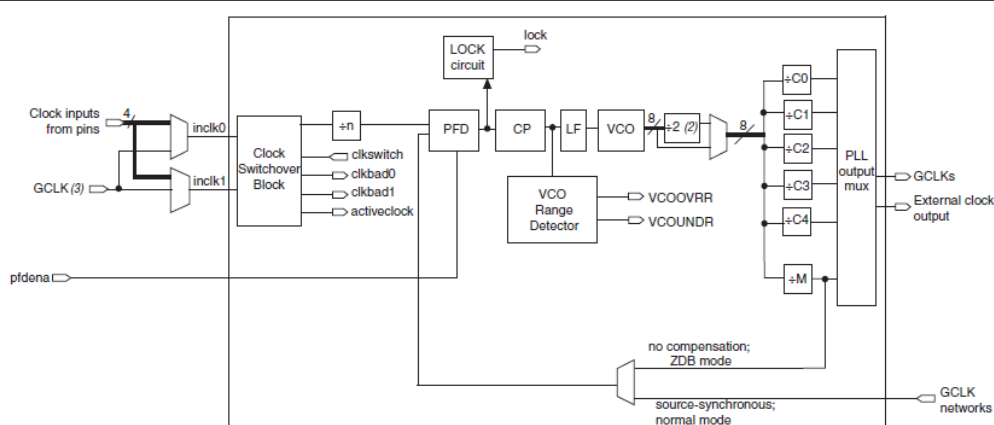
5.2.2 PLL 配置和例化

在进行实践前，我们先吸收一些理论知识。先来看看百度百科对 PLL 的定义：

PLL(Phase Locked Loop): 为锁相回路或锁相环，用来统一整合时脉讯号，使内存能正确的存取资料。PLL 用于振荡器中的反馈技术。许多电子设备要正常工作，通常需要外部的输入信号与内部的振荡信号同步，利用锁相环路就可以实现这个目的。

时钟就是 FPGA 运行的心脏，它的每次跳动必须精准而毫无偏差（当然现实世界中不存在所谓的毫无偏差，但是我们希望它的偏差越小越好）。一个 FPGA 工程中，不同的外设通常工作的在不同的时钟频率下，所以一个时钟肯定满足不了需求；此外，有时候可能两个不同的模块共用一个时钟频率，但是由于他们运转中不同的工作环境和时序下，所以他们常常是同频不同相（相位），怎么办？前面刚刚学了，用 PLL 呗。当然了，我们的 FPGA 里面定义的 PLL，可不是仅仅只有一个反馈调整功能，它还有倍频分频等功能集成其中，严格一点讲，我觉得这个 PLL 实际上应该算是一个 FPGA 内部的时钟管理模块了。不多说，大家看图自己体味体味。

Figure 5-6. Cyclone III Device Family PLL Block Diagram (1)



Notes to Figure 5-6:

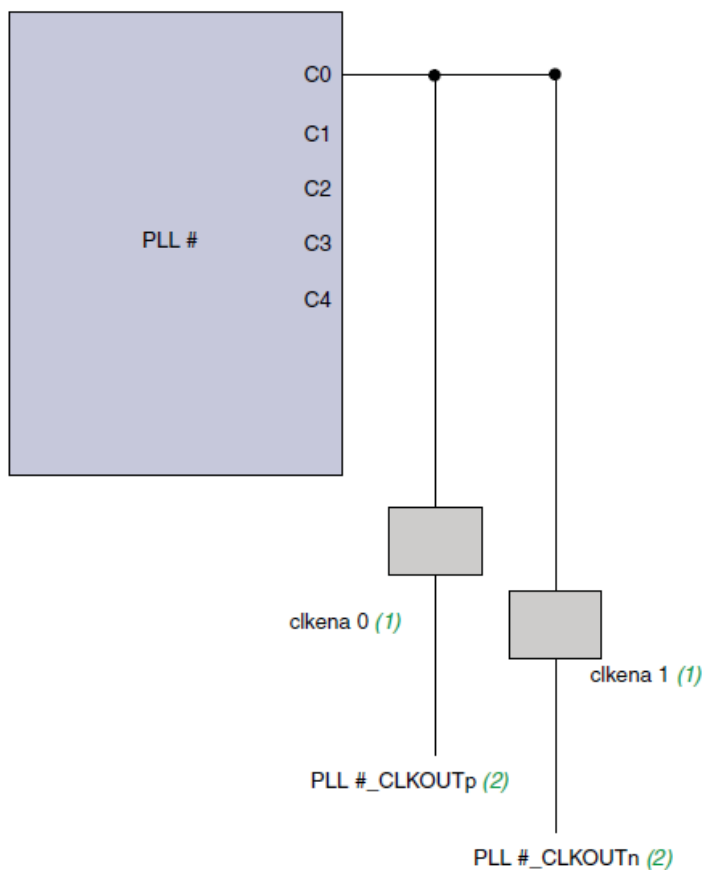
- (1) Each clock source can come from any of the four clock pins located on the same side of the device as the PLL.
- (2) This is the VCO post-scale counter K.
- (3) This input port is fed by a pin-driven dedicated GCLK, or through a clock control block if the clock control block is fed by an output from another PLL or a pin-driven dedicated GCLK. An internally generated global signal cannot drive the PLL.

详细的工作机理请大家参考 Cyclone III Device Handbook 的 5-10 章节内容。

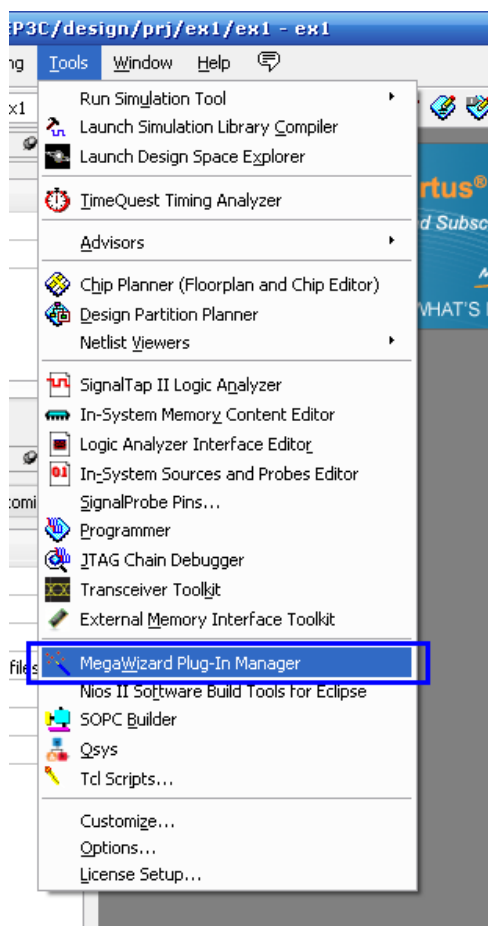
Cyclone III 的这个 PLL 一个输入，最多 5 个输出，输出的频率和相位都是可以在一定范



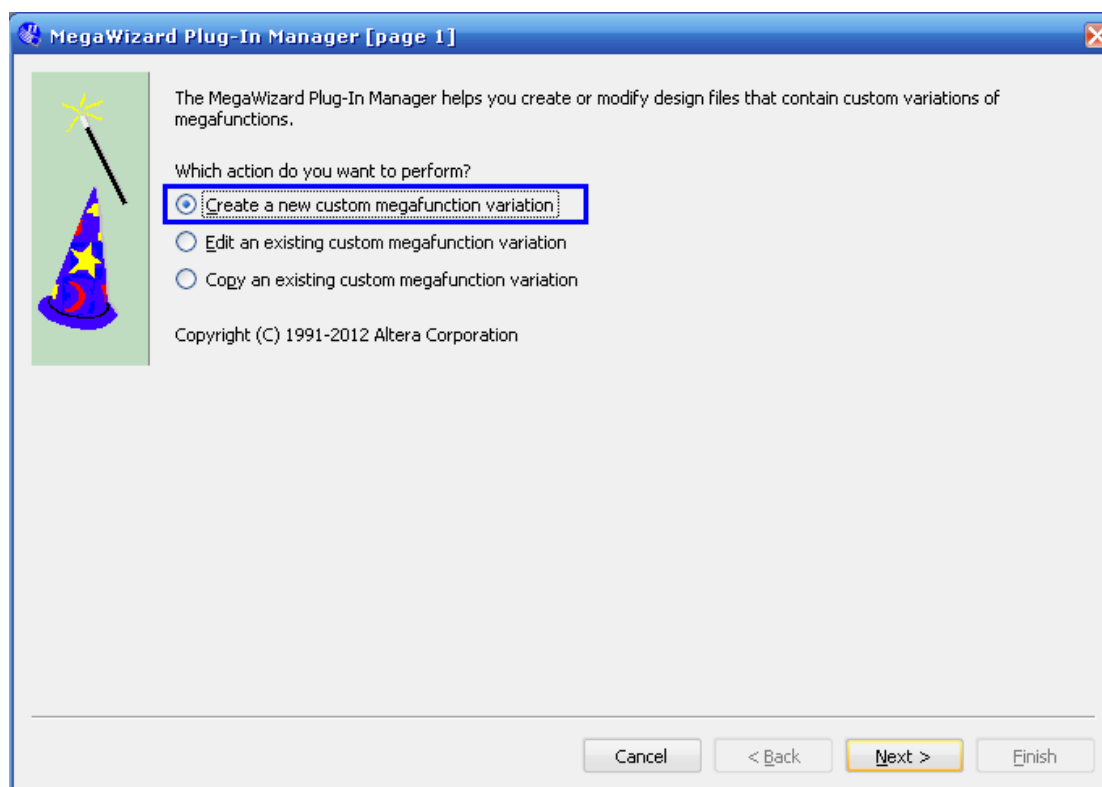
围内调整的。其中 C0 输出可以供到 FPGA 的管脚（必须是专门的某个管脚上）上，即前面提到过的 CLKOUTp 和 CLKOUTn 这对管脚。



在新建的工程中，点击菜单 Tools→MegaWizard Plug-In Manager。

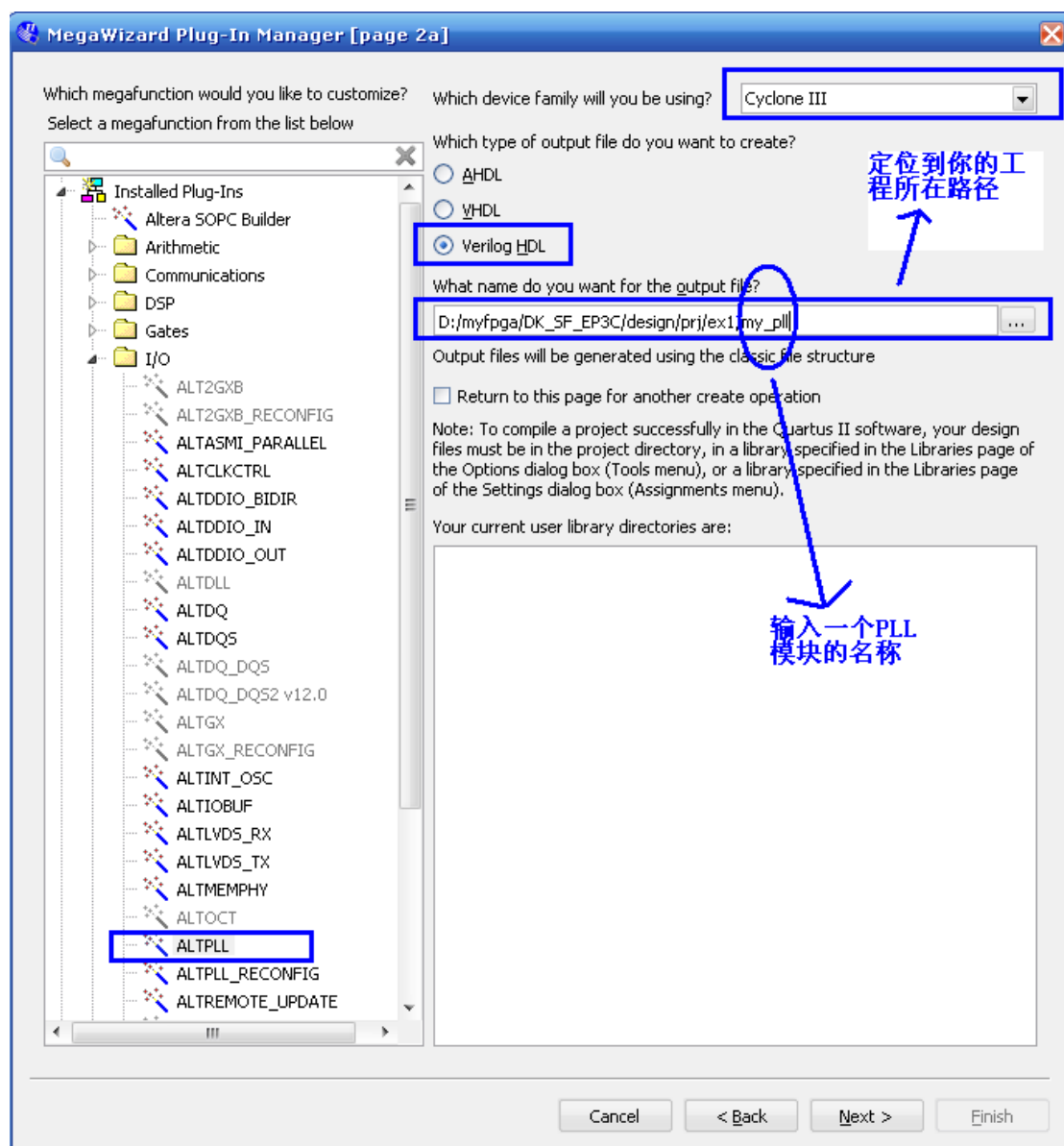


选择 Create a new custom megafunction variation, 然后 Next。

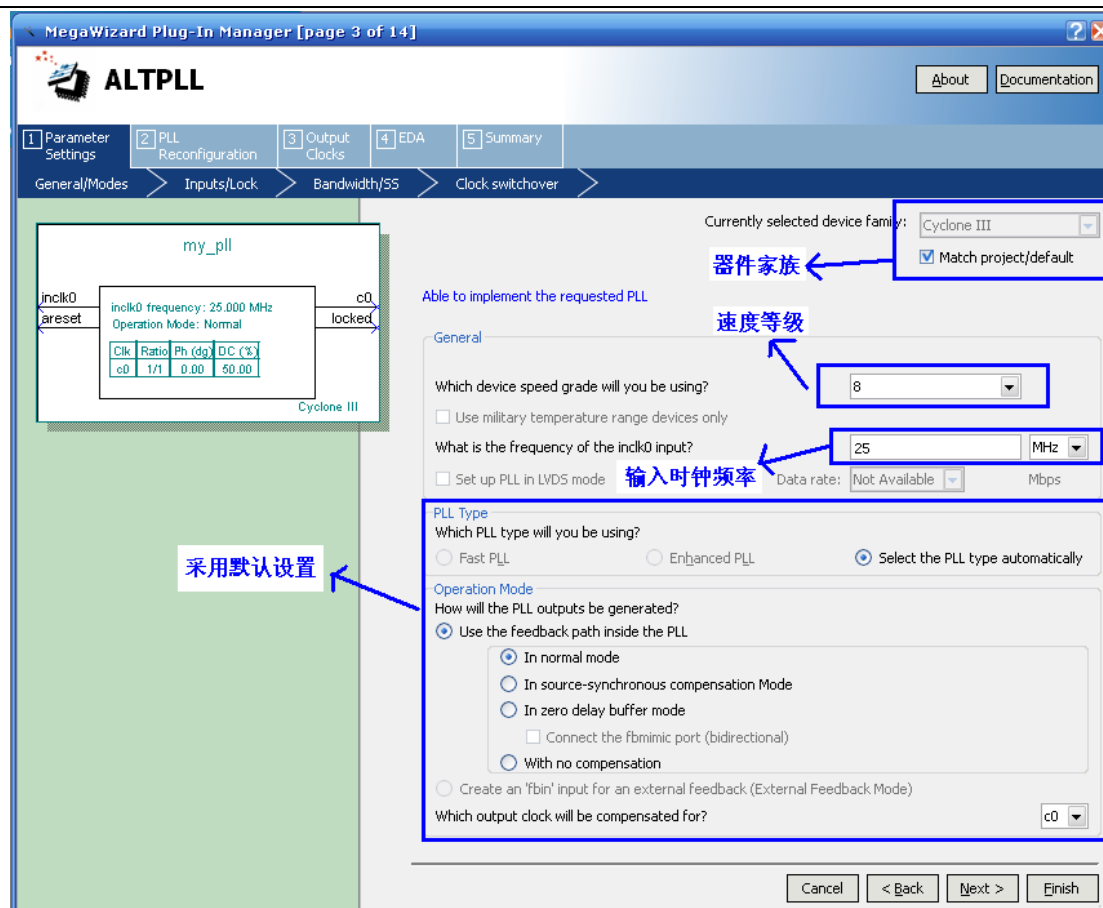




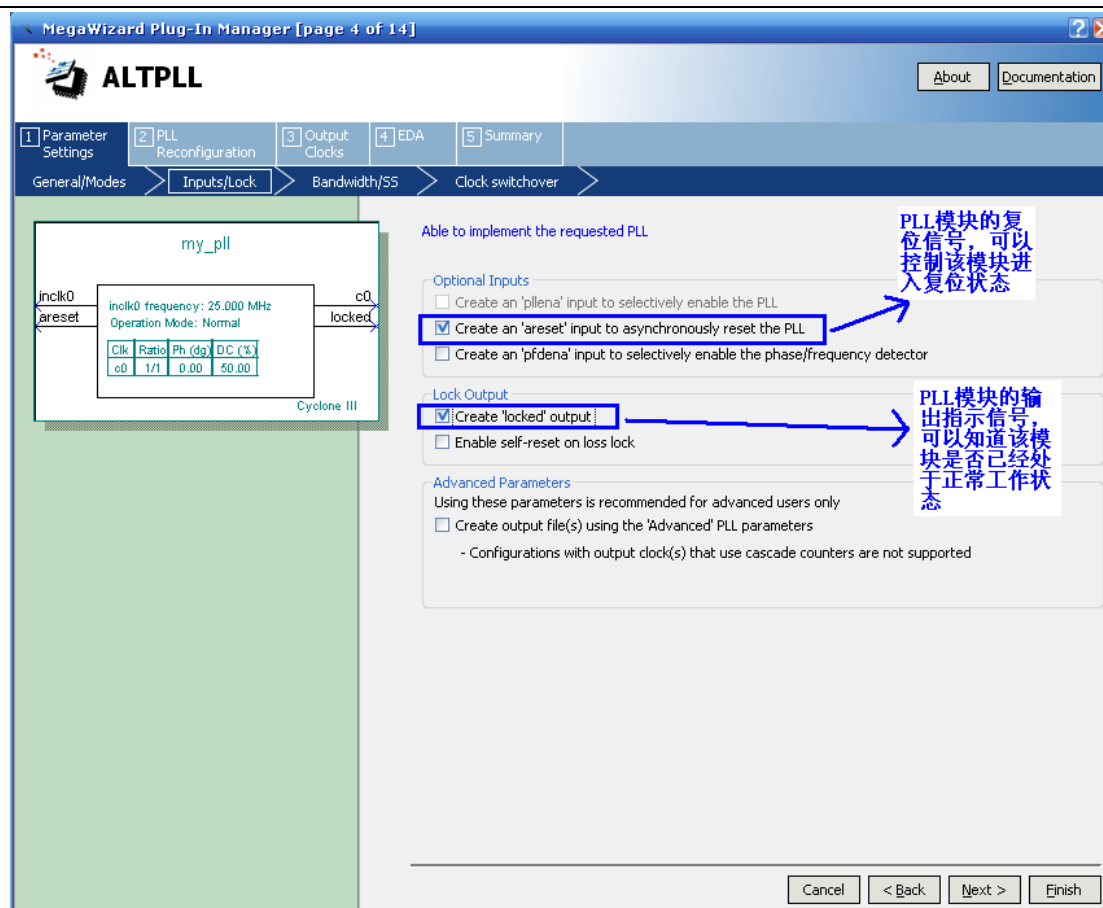
接着选择我们所需要的 IP 核, 如图进行设置, 注意在 What name do you want for the output file?下面输入工程所在的路径, 并且在最后面加上一个名称, 这个名称是我们现在正在例化的 PLL 模块的名称, 我们可以给他起名叫做 my_pll, 然后 Next。



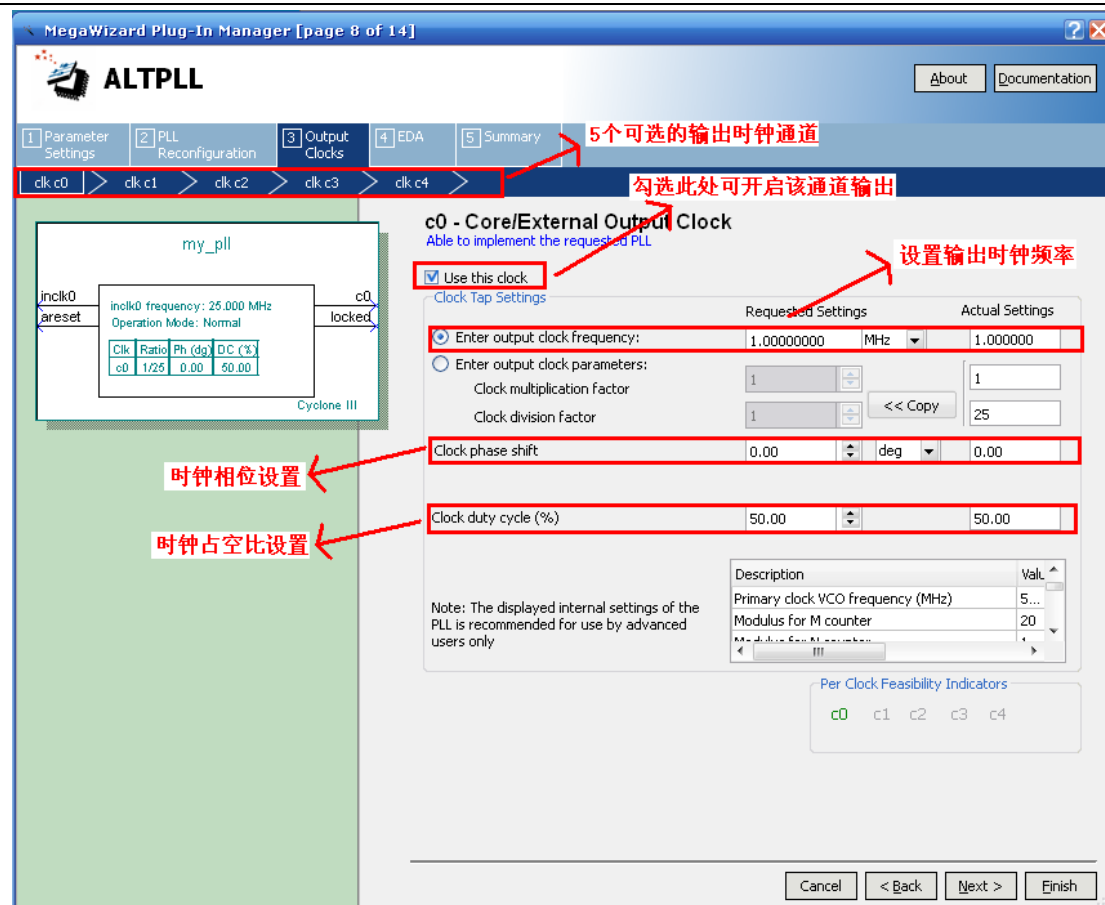
接着来到了 PLL 的参数配置页面, 我们的工程按照图示进行设置。然后 Next。



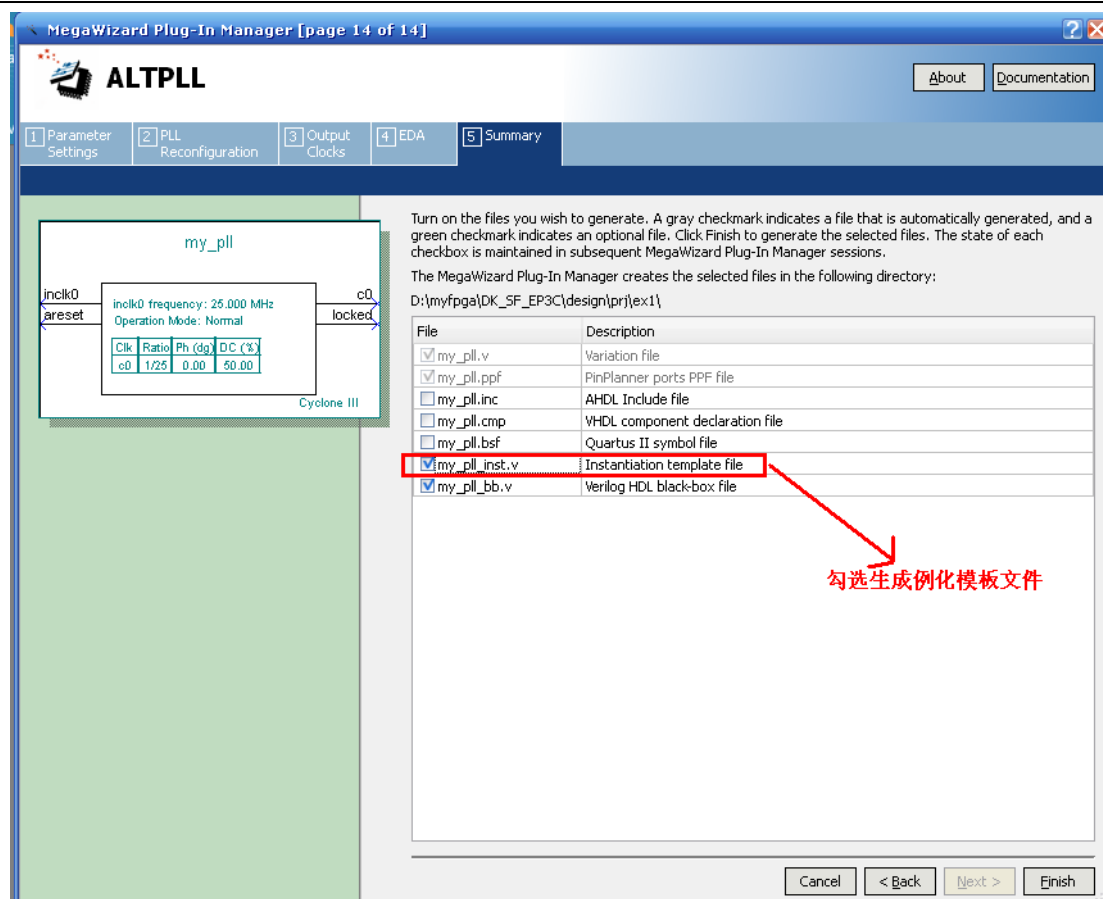
Input/lock 页面中，如图进行设置，接着 Next。



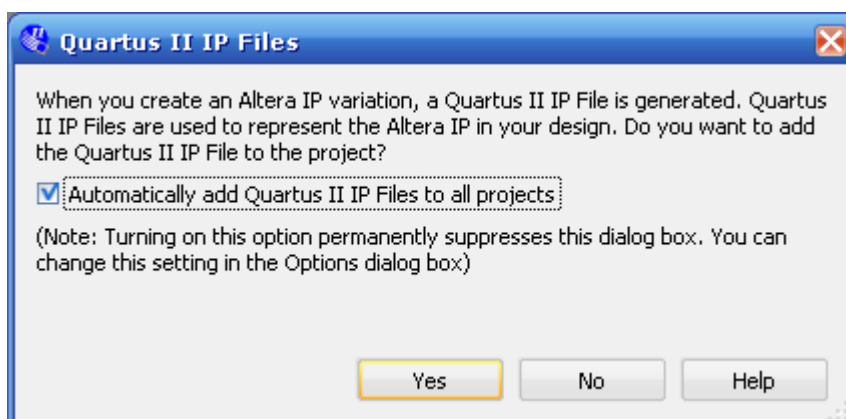
Bandwidth/SS、Clock Switchover 和 PLL Reconfiguration 页面不用设置，默认即可。直接进入 Output Clocks 页面，这里与 5 个可选的时钟输出通道，通过勾选对应通道下方的 Use this clock 选型开启。可以在配置页面中设置时钟的频率、相位和占空比。我们只选择了 C0 输出通道，频率为 1MHz，相位为 0，占空比为 50%。



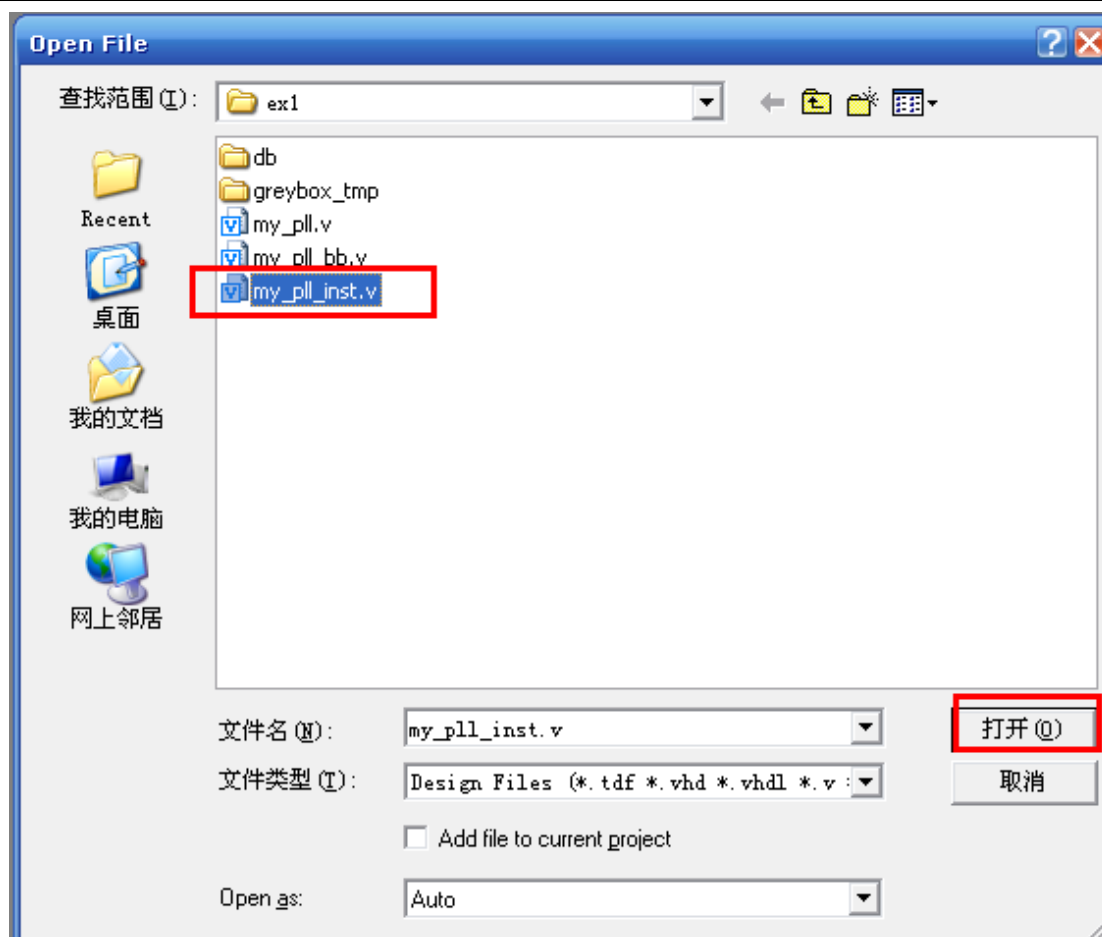
配置完成后, 最后在 Summary 页面, 建议勾选上*_inst.v 文件, 这是一个 PLL 例化的模板文件, 一会我们可以在工程目录下找到这个文件, 然后打开它, 将它的代码复制到工程中, 修改对应接口即可完成这个 IP 核的集成。



点击 Finish 完成 PLL 的配置。工程中若弹出如下对话框，勾选 Automatically...选项后，点击 Yes。



回到 Quartus II 的菜单栏点击 **File**→**Open**，然后路径定位到工程目录下，选择刚刚生成的 my_pll_inst.v，然后点击打开。



我们看到里面有一段简单的例化代码，这里面列出了 4 个接口：areset 是 PLL 模块的复位信号，高电平进入复位状态；inclko 是 PLL 输入的时钟信号；co 是 PLL 输出的时钟信号；locked 信号是 PLL 输出工作状态指示信号，高电平表示 PLL 正常工作了。复制这一段代码。

```
my_pll_inst.v
1 my_pll my_pll_inst (
2     .areset ( areset_sig ),
3     .inclko ( inclko_sig ),
4     .co ( co_sig ),
5     .locked ( locked_sig )
6 );
7
```

我们在 Quartus II 中新建一个 ex1.v 的 Verilog 源码文件。然后输入如下的一段代码：

```
module ex1(
    clk, rst_n, led
);

input clk;
```



```
input rst_n;
output led;

wire clk_lm;
wire lock;

my_pll my_pll_inst (
    .areset ( !rst_n ),
    .inclk0 ( clk ),
    .c0 ( clk_lm ),
    .locked ( lock )
);

reg[19:0] cnt;

always @(posedge clk_lm or negedge rst_n)
    if(!rst_n) cnt <= 20'd0;
    else if(lock) cnt <= cnt+1'b1;

assign led = cnt[19];

endmodule
```

完成输入后, 对该代码进行综合, 确认没有语法错误。

5.2.3 ModelSim 仿真

参照上一个工程实例, 我们点击菜单栏的 Processing→Start→Start Test Bench Template Writer 新建一个 testbench 文件, 然后直接 copy 上一节的测试脚本到测试文件中 (稍微做一些修改)。

```
`timescale 1 ns/ 1 ps
module ex1_vlg_tst();

// test vector input registers
reg clk;
reg rst_n;
```



```
// wires
wire led;

// assign statements (if any)
ex1 il (
// port map - connection between master ports and signals/registers
    .clk(clk),
    .led(led),
    .rst_n(rst_n)
);

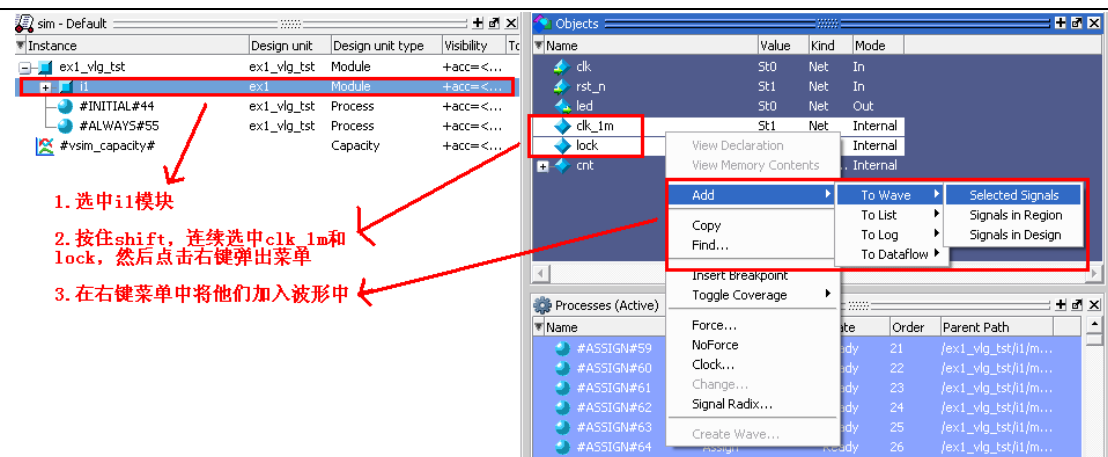
initial begin
    $monitor($time,"led value = %b\n",led); //监控 led 信号的变化, 如果其值发生变化, 立马打印出来
    rst_n = 0;
    clk = 0;
    #1000;
    @(posedge clk);
    rst_n = 1;
    repeat(5) #1_000_000_000; //延时 5s
    $stop;
end

always #20 clk = ~clk; //出生 25MHz 时钟信号

endmodule
```

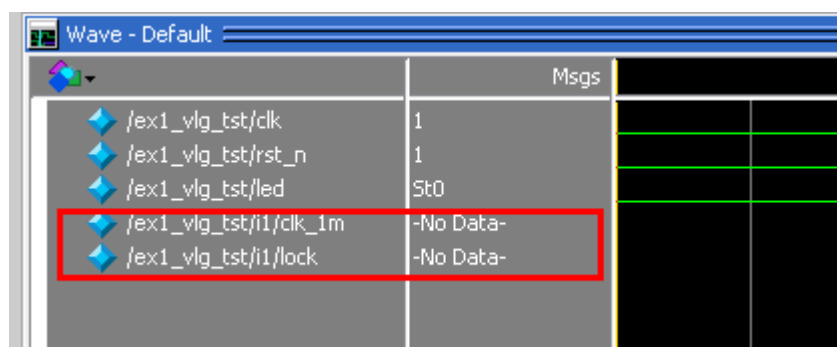
参照上一节在菜单 **Assignments→Setting** 中做好设置, 然后点击菜单栏的 **Tools→Run Simulation Tool→RTL Simulation** 运行 ModelSim-Altera。

在 ModelSim 中, 做如图示的操作, 将 **clk_1m** 和 **lock** 两个内部信号也添加到波形中。

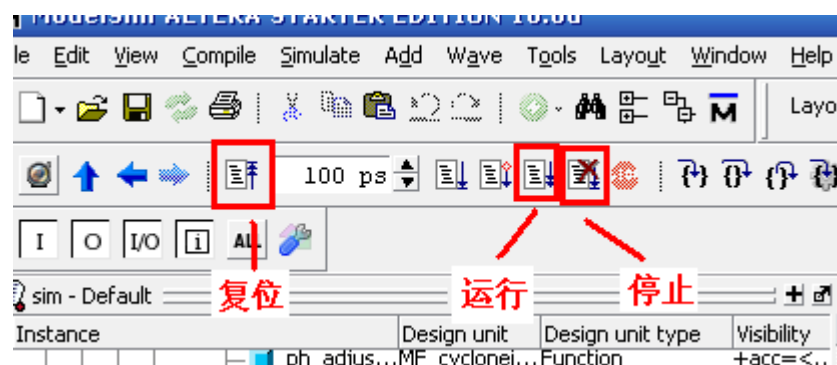


1. 选中i1模块
2. 按住shift, 连续选中clk_1m和lock, 然后点击右键弹出菜单
3. 在右键菜单中将他们加入波形中

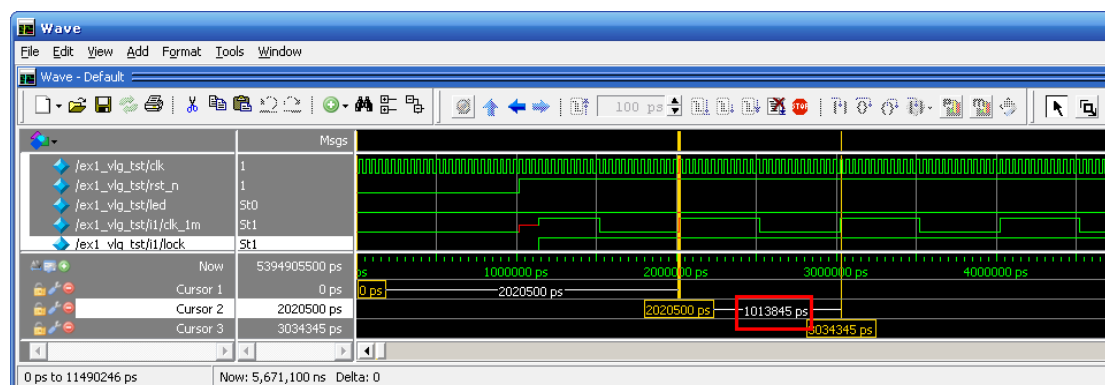
clk_1m 和 lock 两个内部信号出现在波形中了。



接着我们先停止当前的仿真运行，然后复位，接着点击运行。



看现在的波形，当 lock 信号拉高以后，clk_1m 就有正常的方波输出了，一个 clk_1m 的周期时 1MHz（即 1us）。图示中由于两个坐标没有办法精准定位，所以有一定偏差。



大家也可以观察打印窗口 led 值两次输出 1 或 0 的时间间隔, 理论上应该是 $1024 * 1024 \mu s = 1048576 \mu s$ 。

```
# 524289020led value = 1
#
# 1048577020led value = 0
#
# 1572865020led value = 1
#
# 2097153020led value = 0
#
```

5.2.4 管脚分配与编译

点击菜单栏的 Assignments→Pin planner 进行管脚分配。

Node Name	Direction	Location	I/O Bank	I/O Standard	Reserved	Current Strength
clk	Input	PIN_22	1	2.5 V (default)		8mA (default)
led	Output	PIN_28	2	2.5 V (default)		8mA (default)
rst_n	Input	PIN_91	6	2.5 V (default)		8mA (default)

接着我们回到 Quartus II 继续下面的流程。我们可以先对整个工程进行一次全编译, 既可以点击工具栏的 “Start Compilation” 按钮, 也可以在 Task→Compilation 中点击 Compile Design。

5.2.5 下载配置与板级调试

将本工程生成的 ex2.sof 文件下载到板子中, 我们看到 LED 闪烁的频率相对于上一节慢了一些, 上一节是 600 多 ms 的周期, 而这一节是 1s 多的周期。

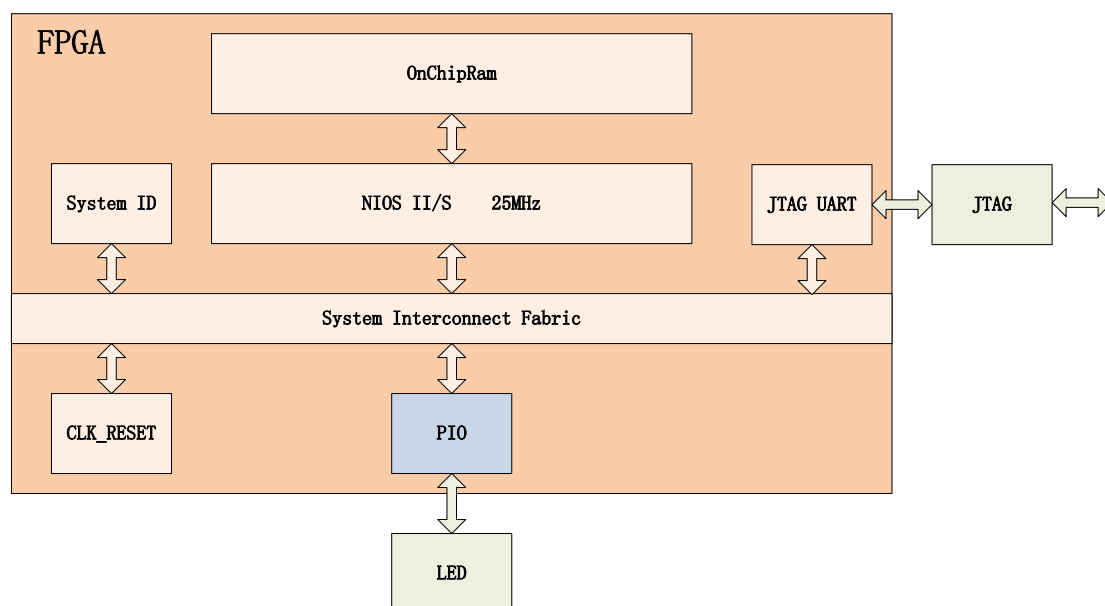


5.3 基于 Qsys 的 NIOS II 实例 1——LED 闪烁

从 Quartus II 11.0 开始,喜新厌旧的 Altera 就不厌其烦的炒作 SOPC Builder 的替代者 Qsys。记得去年参加他们的研讨会时就已经炒得火热,如今 12.0sp2 都已经 release 了,12 以后更是完全摒弃了 SOPC Builder,如果再加紧找个理由上 Qsys 练练手,咱可就要 OUT 了。

前面两个实验我们只是用 FPGA 内部的逻辑做些实验,没有任何嵌入式系统的成分在里面,这个实例我们就要充分发挥 FPGA 的灵活性,搭建一个片上系统,然后在它的处理器 NIOS II 上跑个简单的小程序。

在这个系统中,有一个 CLK_RESET 外设,负责产生整个系统所需要的时钟和复位信号。NIOS II 处理器上不仅挂了个片内 RAM 用于跑程序,而且作为主机连接到系统总线上,它上面挂了 3 个外设: System ID 用于系统标识; JTAG UART 用于系统调试; PIO 连接 LED 指示灯,我们这个实例就是要在该系统上跑个 LED 闪烁实例。(汗!都第三个例程了,还跑不出 LED 闪烁。呵呵,这个是最基础实例,大家重在学习系统的搭建和工具的使用,举一反三,以后就没啥不会的。)



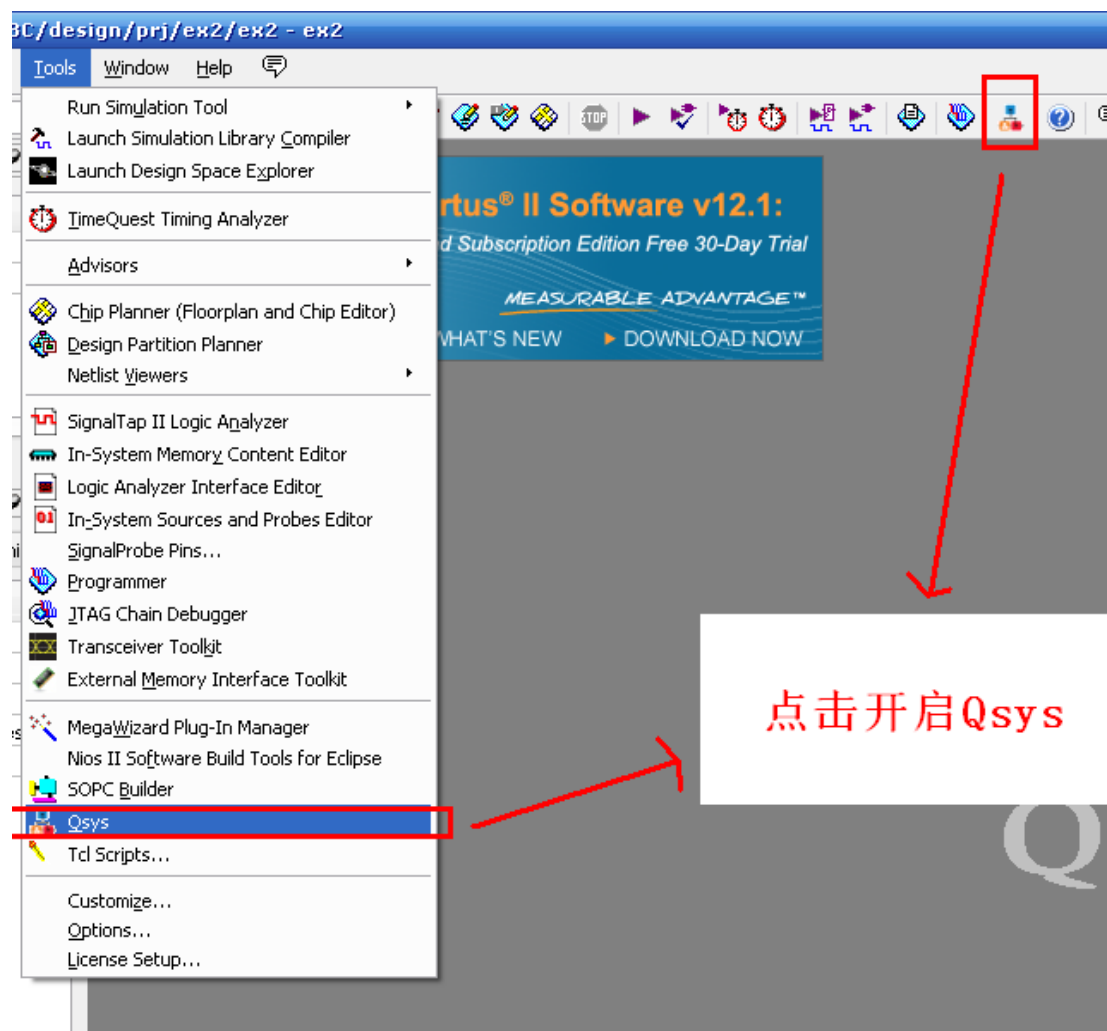
5.2.1 新建工程

首先请建立一个文件夹,名为 ex2,然后在 Quartus II 中新建一个工程,同样名为 ex2,将工程路径定位到 ex2 文件夹下。

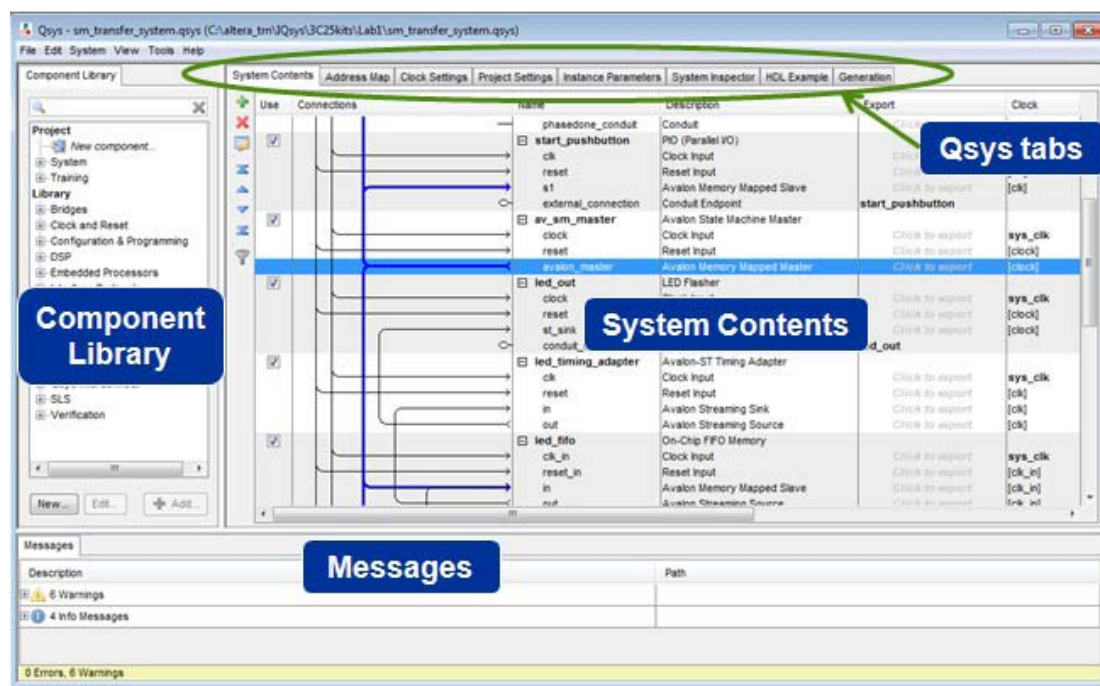


5.1.2 Qsys 硬件系统架构

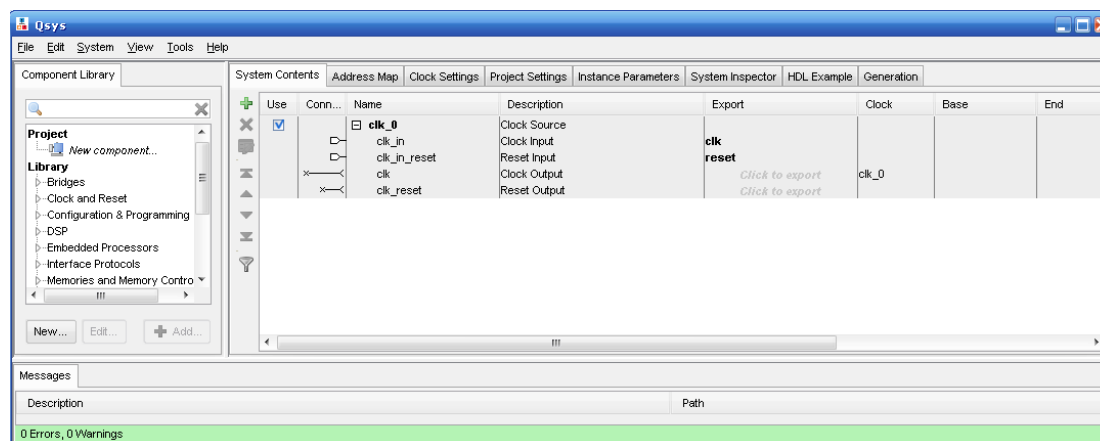
在 Quartus II 中开启 Qsys，如图所示。



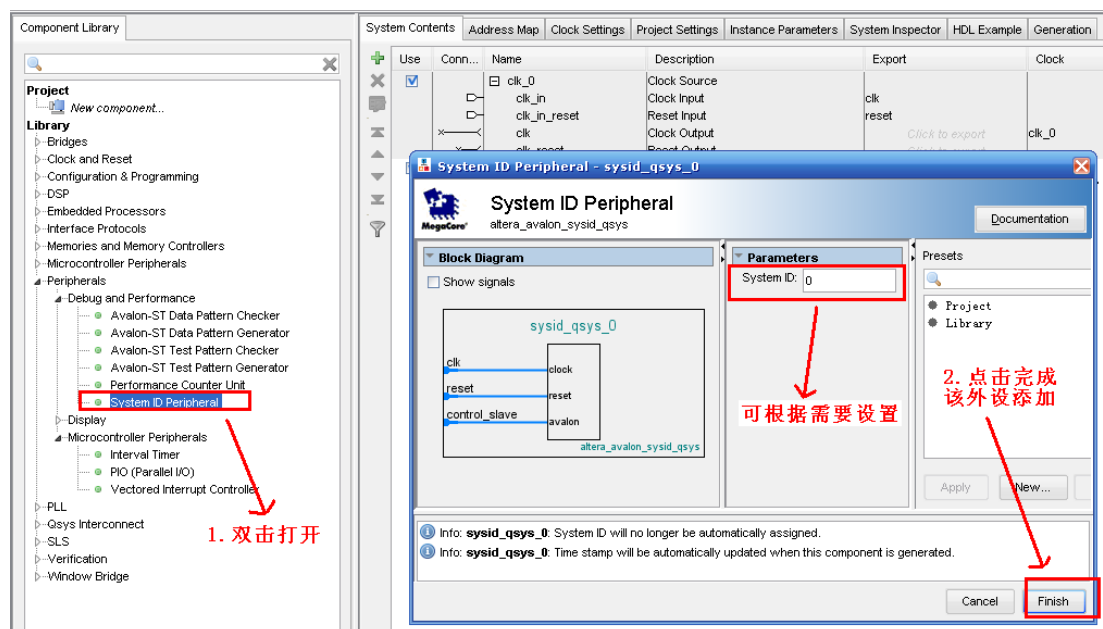
进入 Qsys 后，界面窗口的布局内容多少还有些似曾相识，毕竟还是 SOPC Builder 一脉相承的，一个最大的变化是 Qsys tabs 的选项要比 SOPC Builder 多得多，Qsys 的更多系统个性化编辑和设置也都得益于此了。



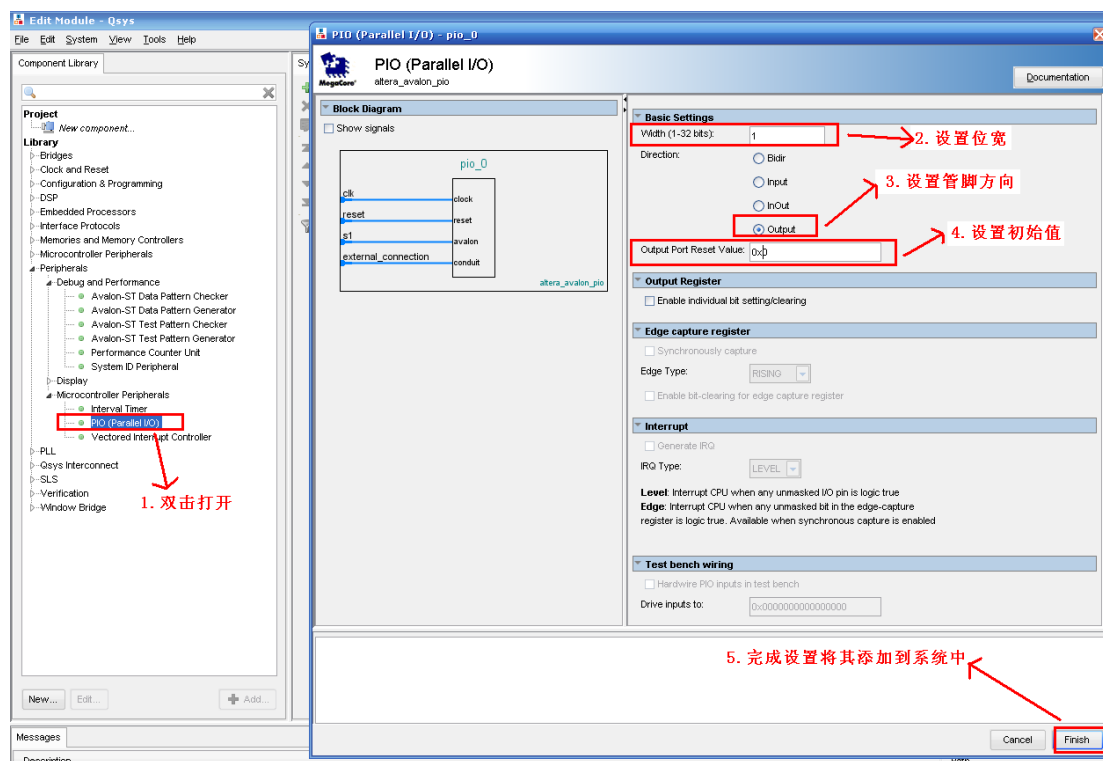
初次打开 Qsys, System Content 中默认已经添加了一个孤零零的 CLOCK 组件, 其他啥也没有, 光杆司令只是个摆设, 啥活干不了。于是乎, 咱觉得在 Component Library 中各种查找, 添加了几个常用组件, 如 NIOS II 处理器、JTAG UART、PIO、system ID 和 20KB 的片内 RAM。



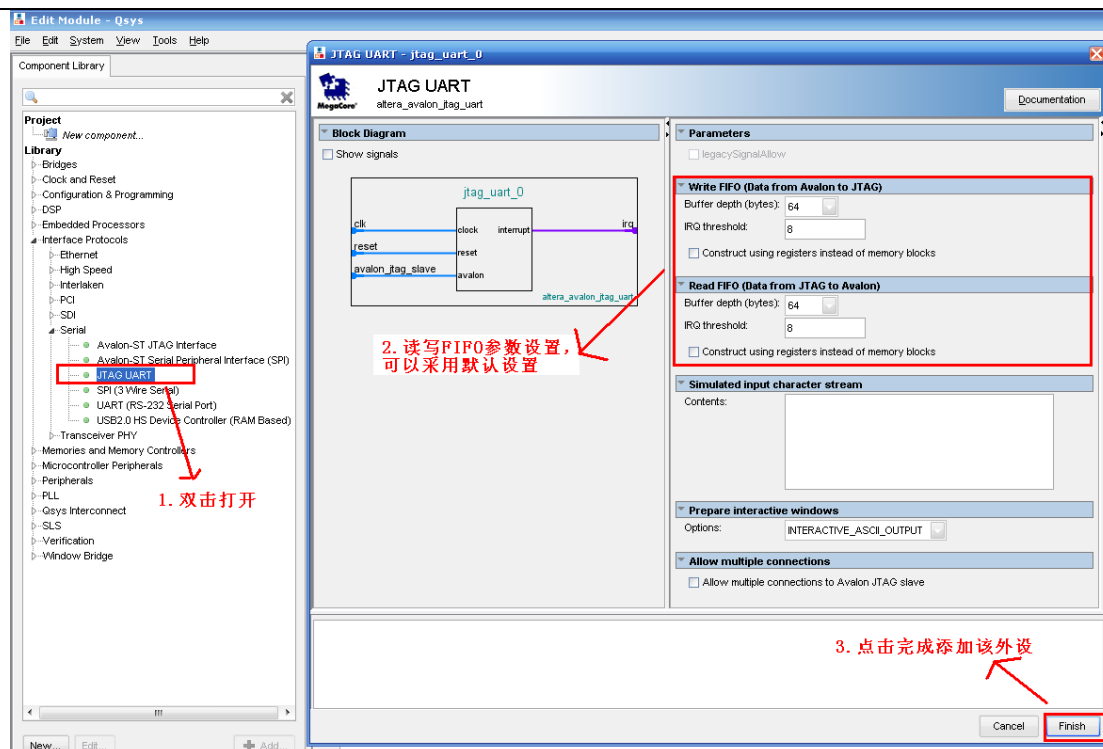
添加 System ID 外设。



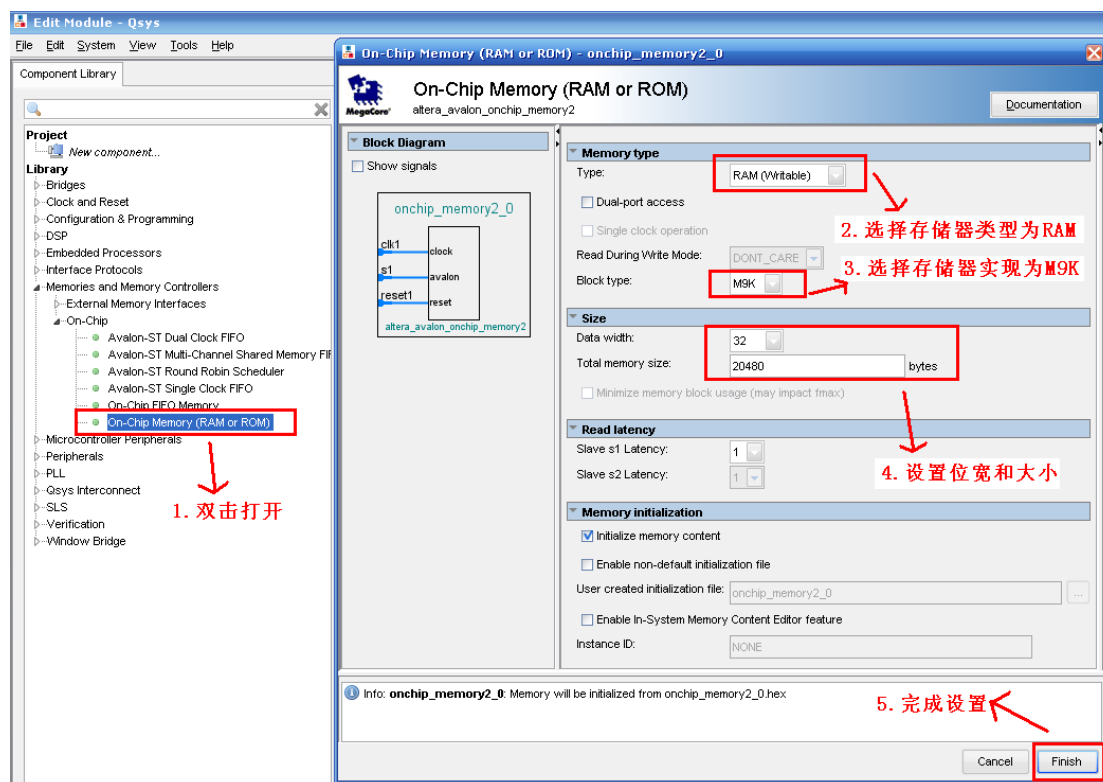
添加 PIO 外设。



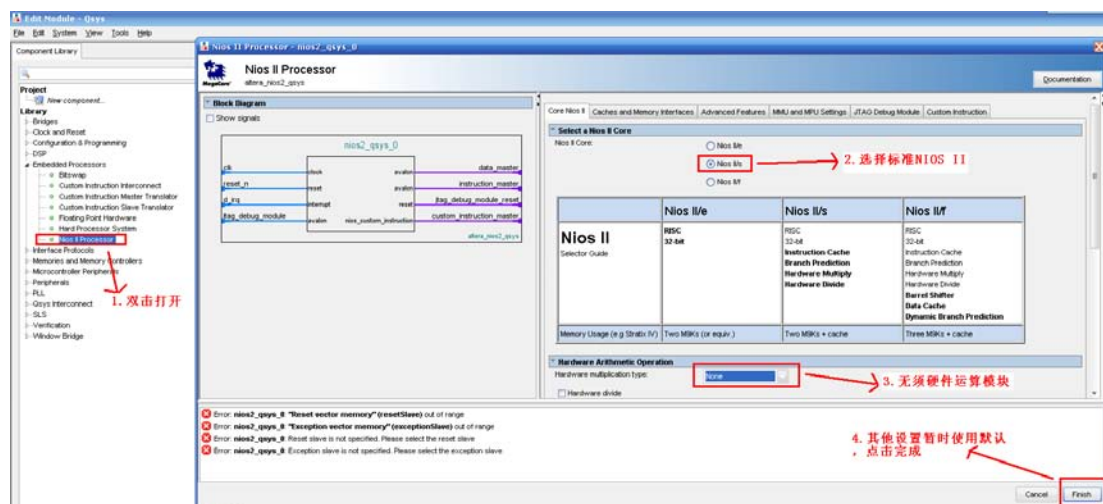
添加 JTAG UART 外设。



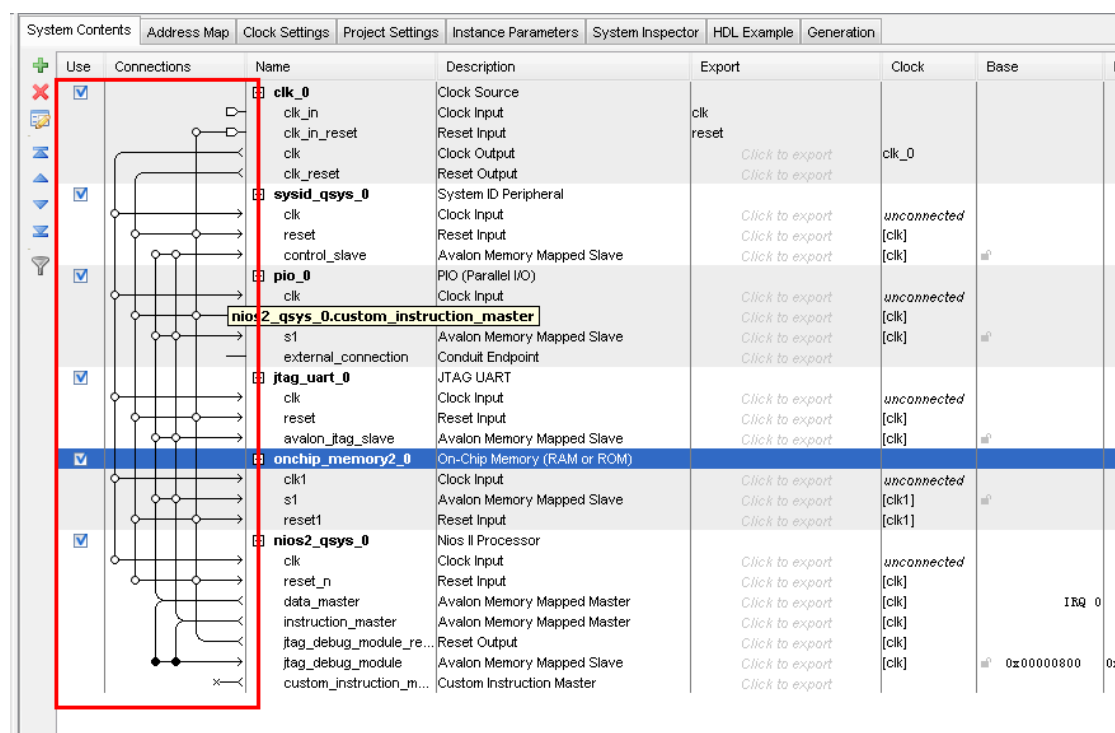
添加片内 RAM。



添加 NIOS II 处理器。



接下来我们需要在 System Contents 的 Connections 一系列中进行一些连接, 将刚才添加的一些分立的外设或 CPU 连接在一起, 构成一个真正的系统。

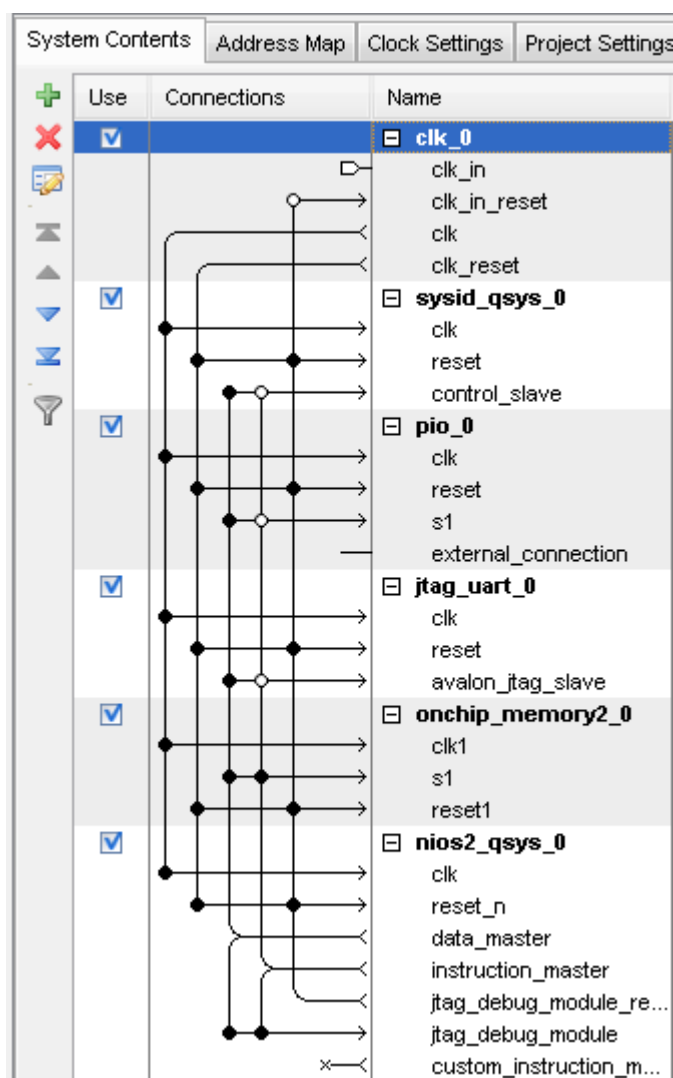


看到小圆圈点点空心实心就会变化, 实心代表连接上了。这连接的活可好玩了, 一点不比那些年不知道祸害了多少有志青年的“连连看”差多少, 那比得是速度, 咱比的是准确。系统的连接其实也非常简单, 我们的时钟 clk 和复位 reset 都没有做太复杂, 都是 clk_0 组件输出, 所以所有的组件都和 clk_0 的时钟复位连接上就对了; CPU 的数据存储器和代码存储器都必须由片内 RAM 来担当, 所以 nios2_qsys_0 的 data_master 和 instruction_master 均与代表 onchip_memory2_0 的从机总线 s1 连接上。而其他作为总线 slave 的外设均连接到 nios2_qsys_0 的 data_master 上即可。

《圣经》箴言九 11 “敬畏耶和华是智慧的开端, 认识至胜者便是聪明。”



系统互连如下。



另外, 要说明的是作为系统与外部连接的接口不像 SOPC Buider 一样直接引出了, 需要设计者特别设置一下。如图所示, 选择 Export 列的属性为*_external connection, 然后该接口前面会出现一个 export 的图标。



2. 前面的图标就会变成这样

1. 分别点击它们

在 Name 一列，我们可以选中每个外设，然后点击右键，选择 Rename，对他们的名称进行修改，最后修改如图所示。

Name

Description

clk

clk_in

clk_in_reset

clk

clk_reset

sysid_qsys

clk

reset

control_slave

pio

clk

reset

s1

external_connection

jtag_uart

clk

reset

avalon_jtag_slave

onchip_mem

clk1

s1

reset1

nios2_qsys

clk

reset_n

data_master

instruction_master

jtag_debug_module_re...

jtag_debug_module

custom_instruction_m...

Connections

Filter

Edit... Ctrl+E

Rename Ctrl+R

Remove

Details

Show Arbitration Shares

Lock Base Address

Expand All

Collapse All

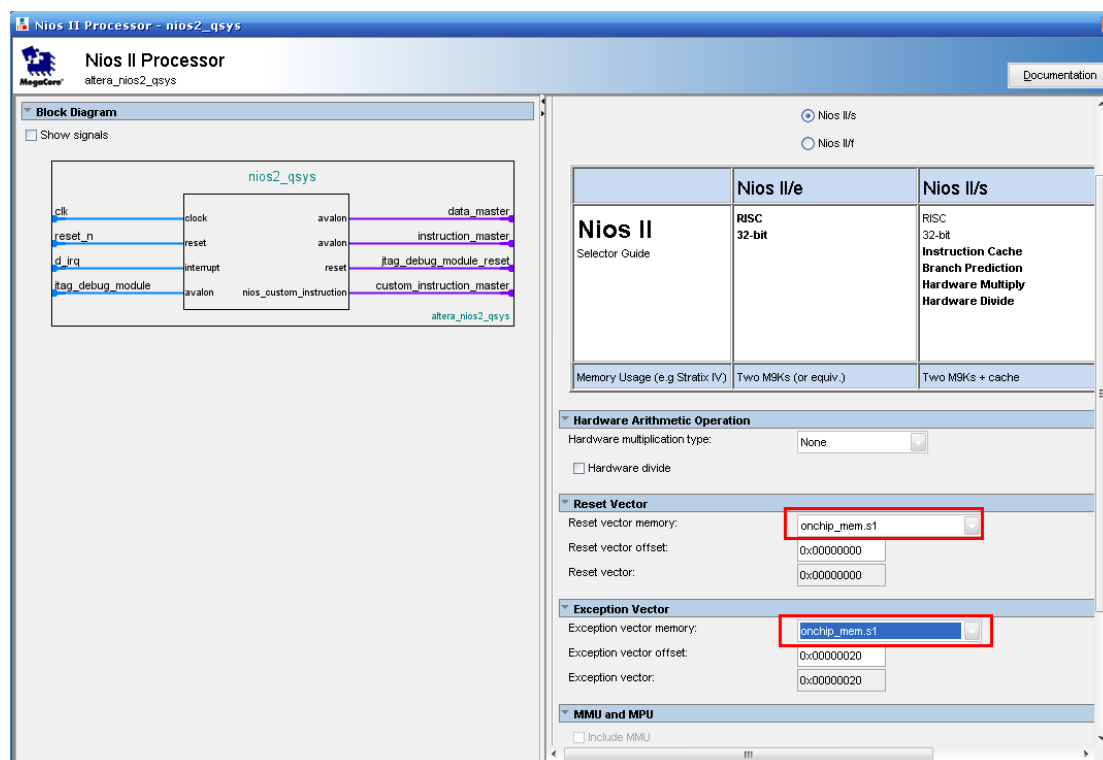
Set Color...

Print...

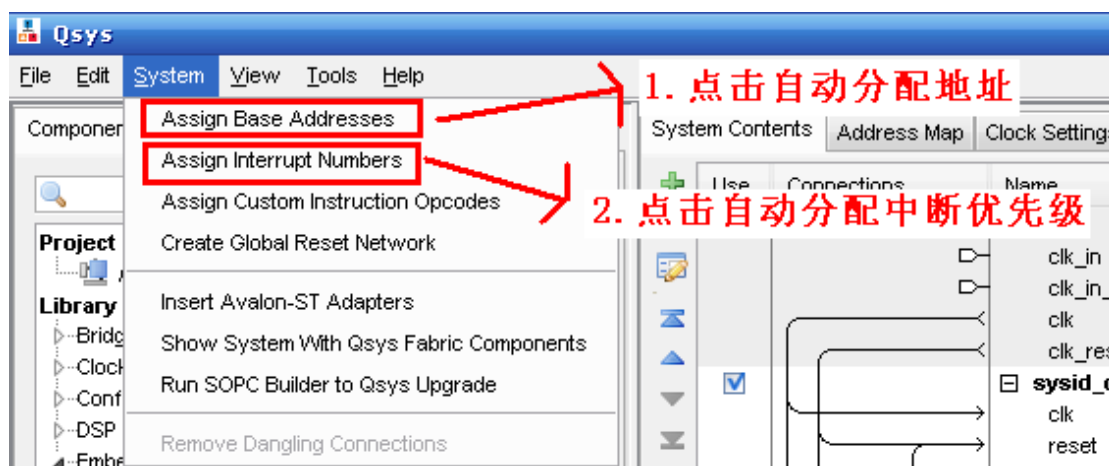
接着我们还要双击打开 nios2_qsys 组件，将其 Reset Vector 和 Exception Vector 均设为



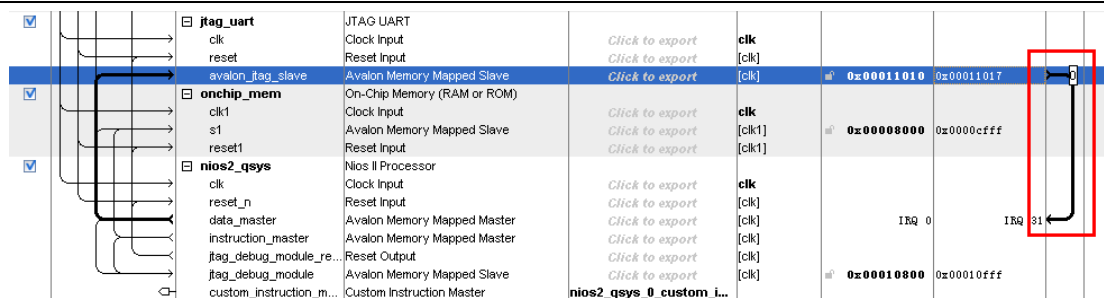
onchip_mem。



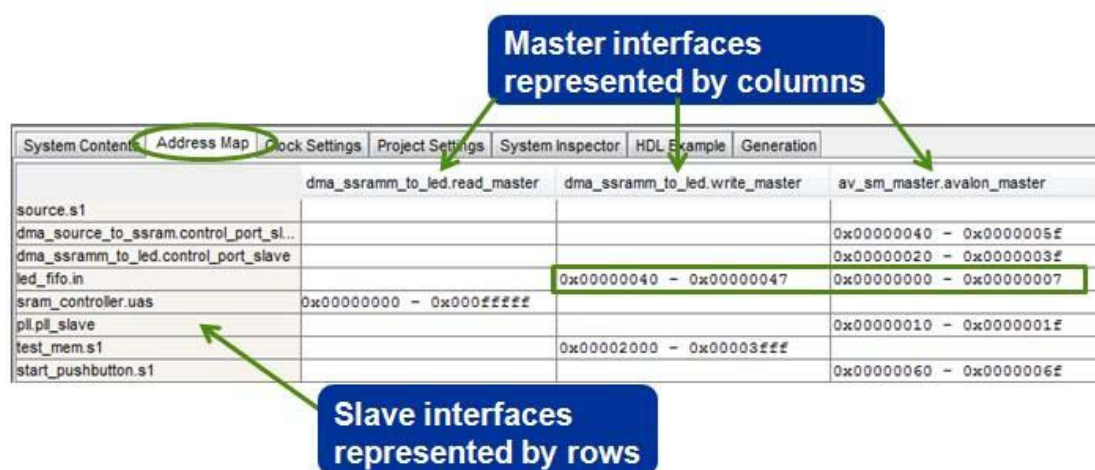
就此，一个漂亮的测试系统搭建完毕，后面的事情就是分配地址、中断优先级等，这个可以如同 SOPC Builder 一样使用菜单栏上的自动分别选项一键分配。特权同学就常常这么干，咱对地址还真没讲究，中断优先级有时还可以根据需要调整一下。



很多外设都有中断信号，在我们的 Qsys 中也需要将对应的 IRQ 一列中的外设和 NIOS II 处理器连接上。在没有连接是，对应外设上有一个空心的小点，如果点上去，则不是实心的小圆，而是可以填写数字的一个空心大圆，对应可以写上它的中断优先级。



前面提到了 Qsys tabs 是一大特色, 这里不一一细说, 偷懒贴几张图。大家自己使用的时候可以慢慢体味。Address Map 对地址的管理一目了然, 而且对于不同的 Master 可以有不同的地址空间映射。



System Inspector 中罗列所有的信号接口以及相关属性参数, 甚至可以在此处进行修改。

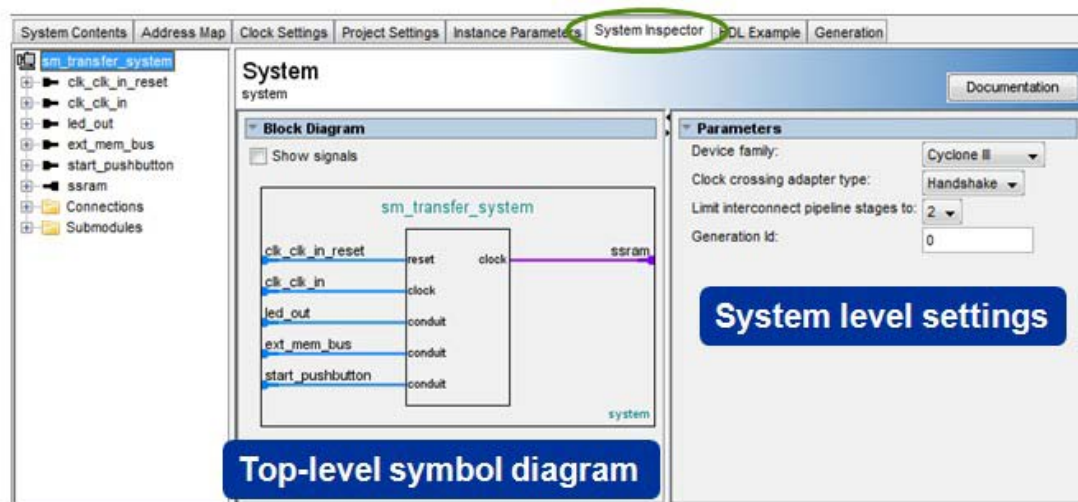


图 9

HDL Example 中直接给出了当前系统的例化模板, 直接复制到工程顶层模块后进行修改即可, 这比之前专门要到工程目录下找相关文件查看要方便得多。

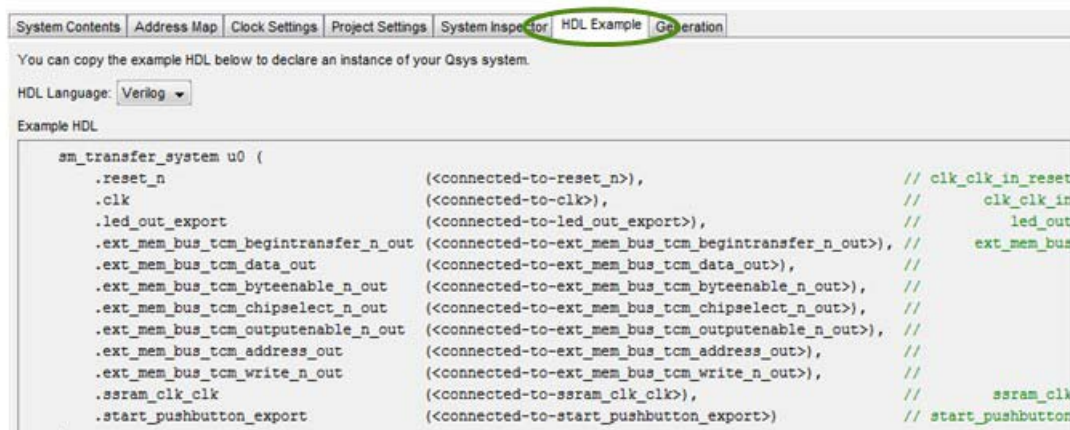


图 10

最后， Generation 里可以选择系统仿真、综合以及各种输出的相关设置，最后点击右下角的 Generate 即可启动当前系统的生成。大家可别忘了在 Output Directory 中设置后系统输出路径。

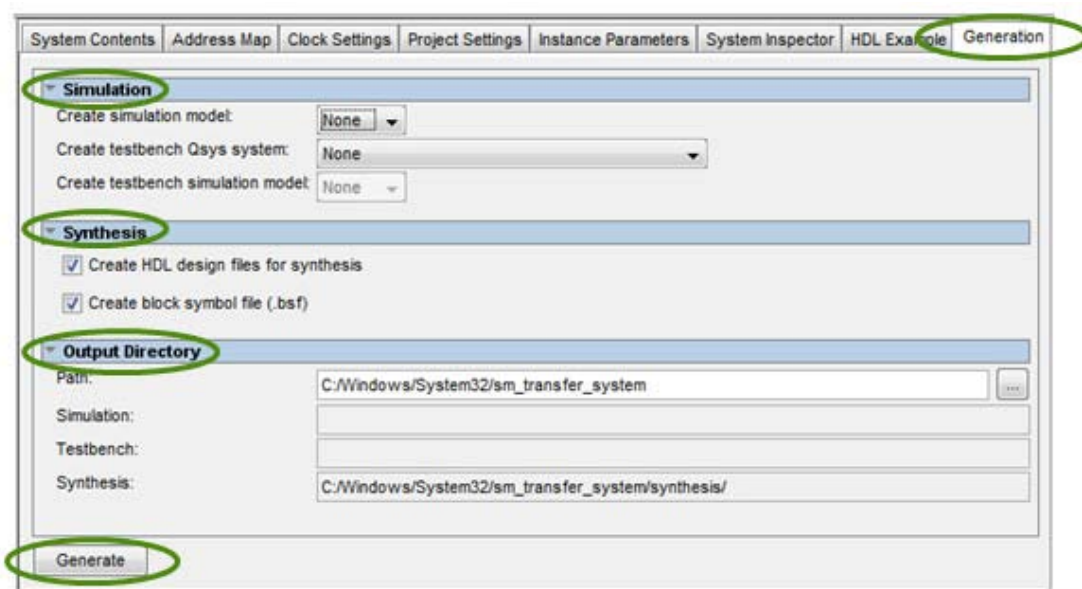
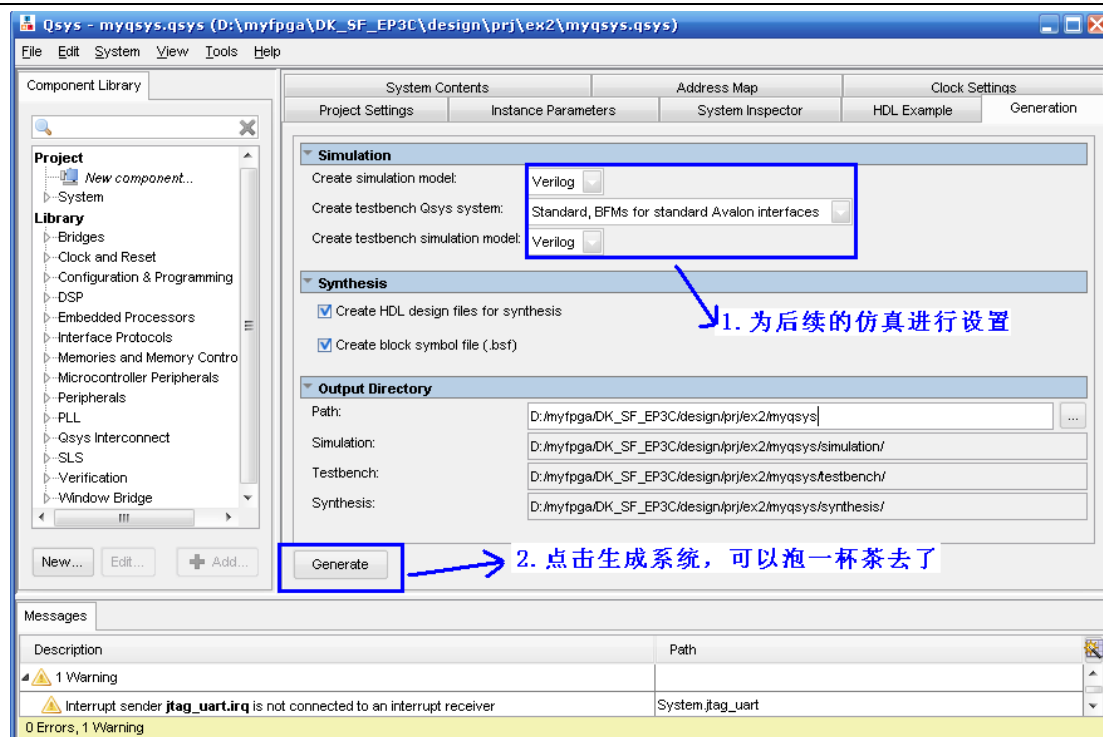
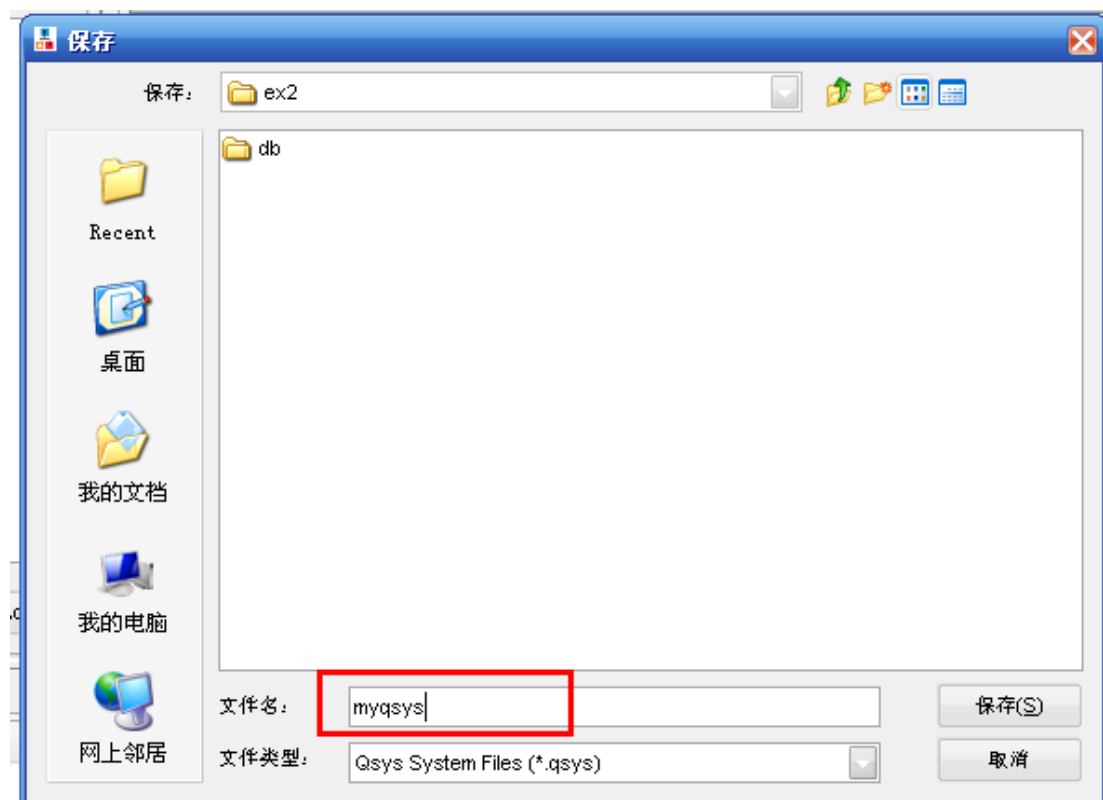


图 11

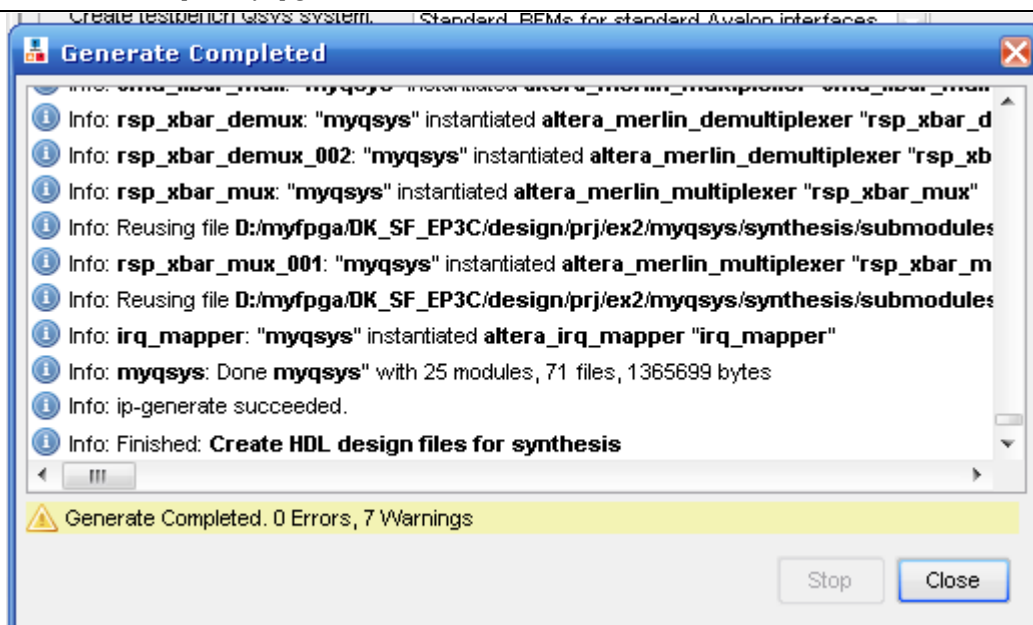
最后，我们来到 Generation 页面，首先为后续的仿真需要做一些设置，选择 Create simulation model 为 Verilog；选择 Create testbench simulation model 为 Standard, BFM's for standard Avalon interfaces。然后我们点击 Generation 开始生成系统，这个时间较长，大家可以去泡杯茶先。



系统第一次点击 **Generation** 后，会弹出提示是否保存当前系统，选择 **Yes**，然后弹出保存路径和命名，可以命名为 **myqsys**，然后保存。

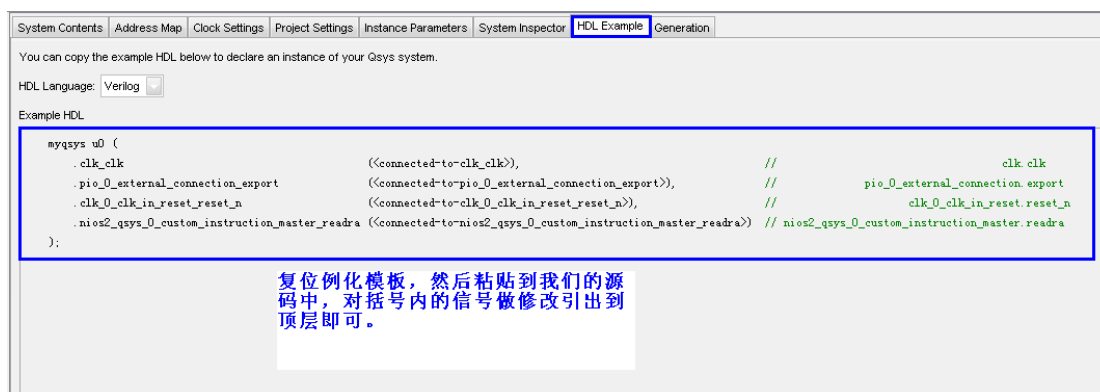


系统生成后，会弹出如下提示。



5.1.3 例化 Qsys 系统

在 Qsys 中, 我们可以先复制例化模板的代码。



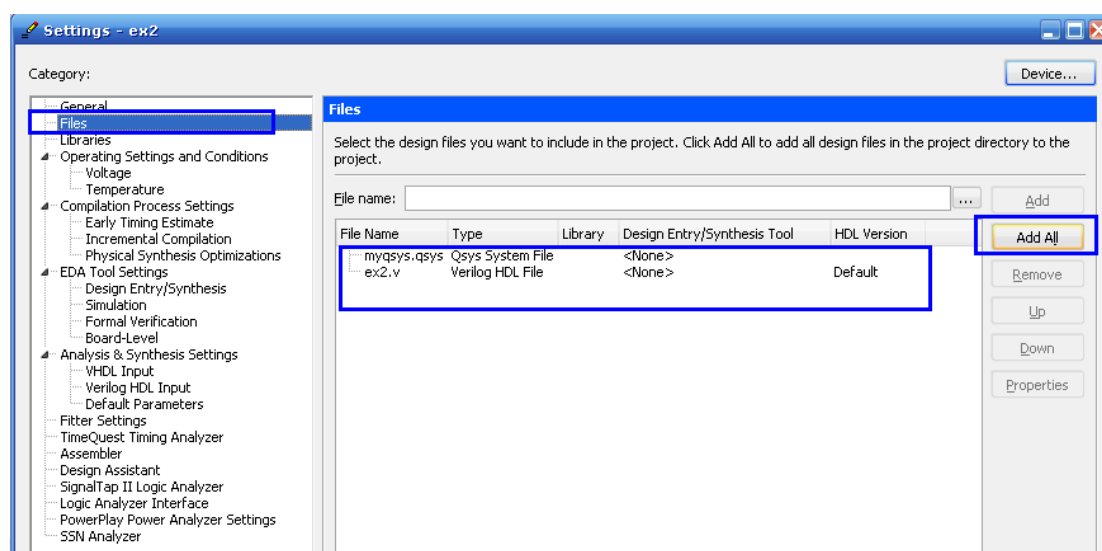
回到 Quartus II, 我们新建一个名为 ex2.v 的 verilog 源代码文件。然后输入以下的代码 (包括前面复制过来的 Qsys 工程的例化修改)。

```
module ex2(  
    clk, rst_n, led  
);  
  
input clk;  
input rst_n;  
output led;
```



```
myqsys u0 (  
    .clk_clk (clk),  
    .pio_0_external_connection_export(led),  
    .clk_0_clk_in_reset_reset_n (rst_n),  
    .nios2_qsys_0_custom_instruction_master_readra ( )  
);  
  
endmodule
```

接着先点击菜单栏的 **Assignments→Setting**, 选择 **Files**, 然后点击右侧的 **Add All** 按钮, 将 **myqsys.Qsys** 和 **ex2.v** 文件添加到工程中。



接着咱可以先对工程进行一次综合编译。

5.1.4 管脚分配与编译

完成前面的综合编译, 我们打开 **Assignments→Pin Planner** 进行管脚分配。

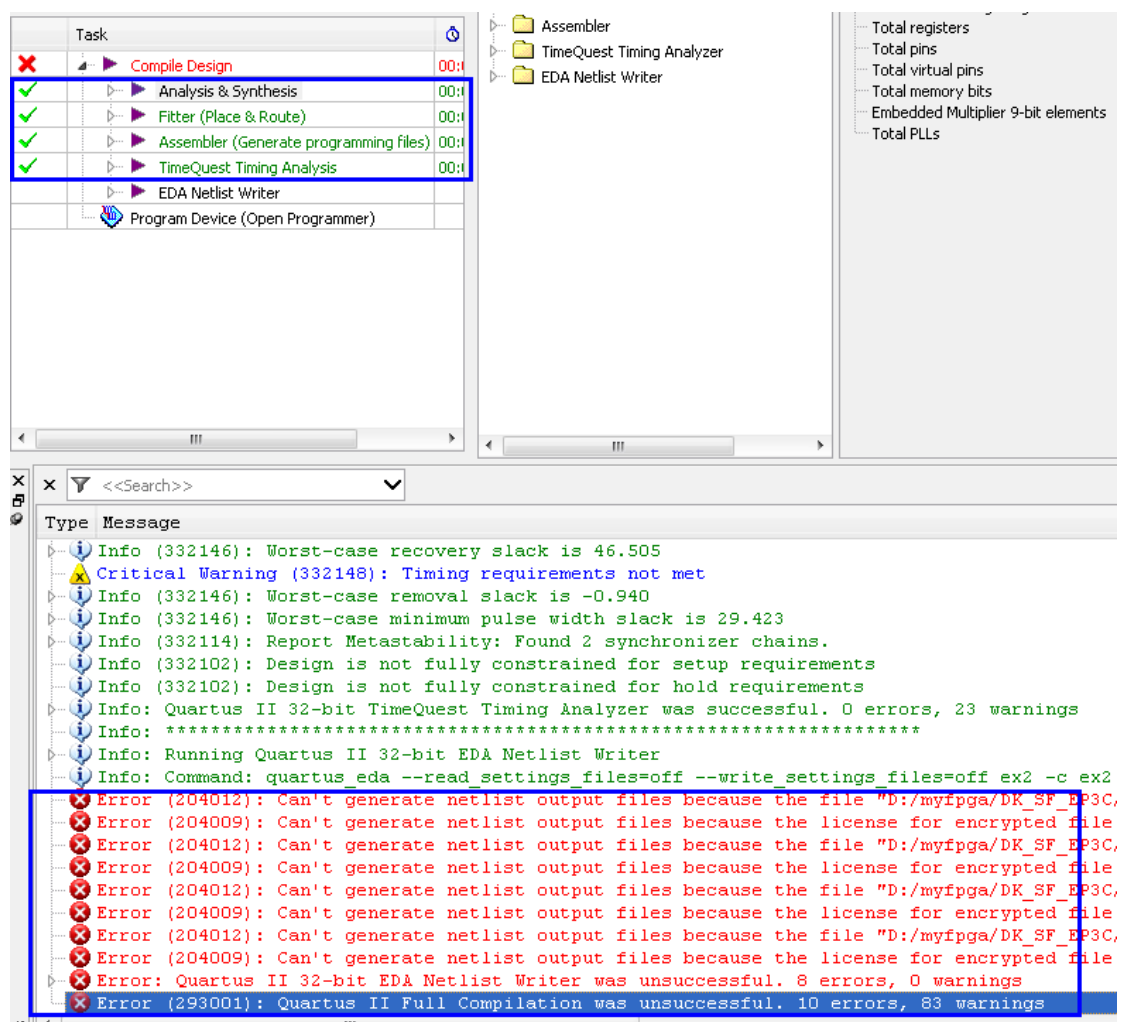
Node Name	Direction	Location	I/O Bank	I/O Standard	Reserved	Current Strength
clk	Input	PIN_22	1	2.5 V (default)		8mA (default)
led	Output	PIN_28	2	2.5 V (default)		8mA (default)
rst_n	Input	PIN_91	6	2.5 V (default)		8mA (default)
<new node>>						

接着对整个工程做一次全编译。由于我们使用的是 **Web** 版本的 **Quartus II**, 如果在编译的最后仅仅是 **EDA Netlist Writer** 编译失败, 那么请大家不必在意, 我们使用了一些 IP 核是受限的, 只能生成一个试用的下载文件, 这个试用的下载文件只能在线下载都 **FPGA** 中, 而且如果拔掉 **JTAG** 线缆, 一定的时间到了这个配置文件就失效。对于我们学习使用, 这个不

《圣经》箴言九 11 “敬畏耶和华是智慧的开端, 认识至胜者便是聪明。”

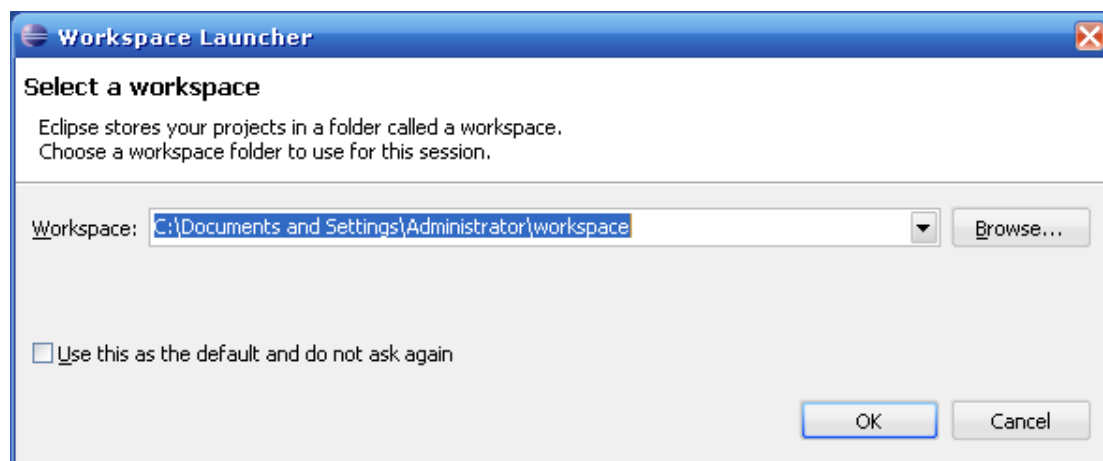


是问题，咱照样可以把相关的功能玩一遍。

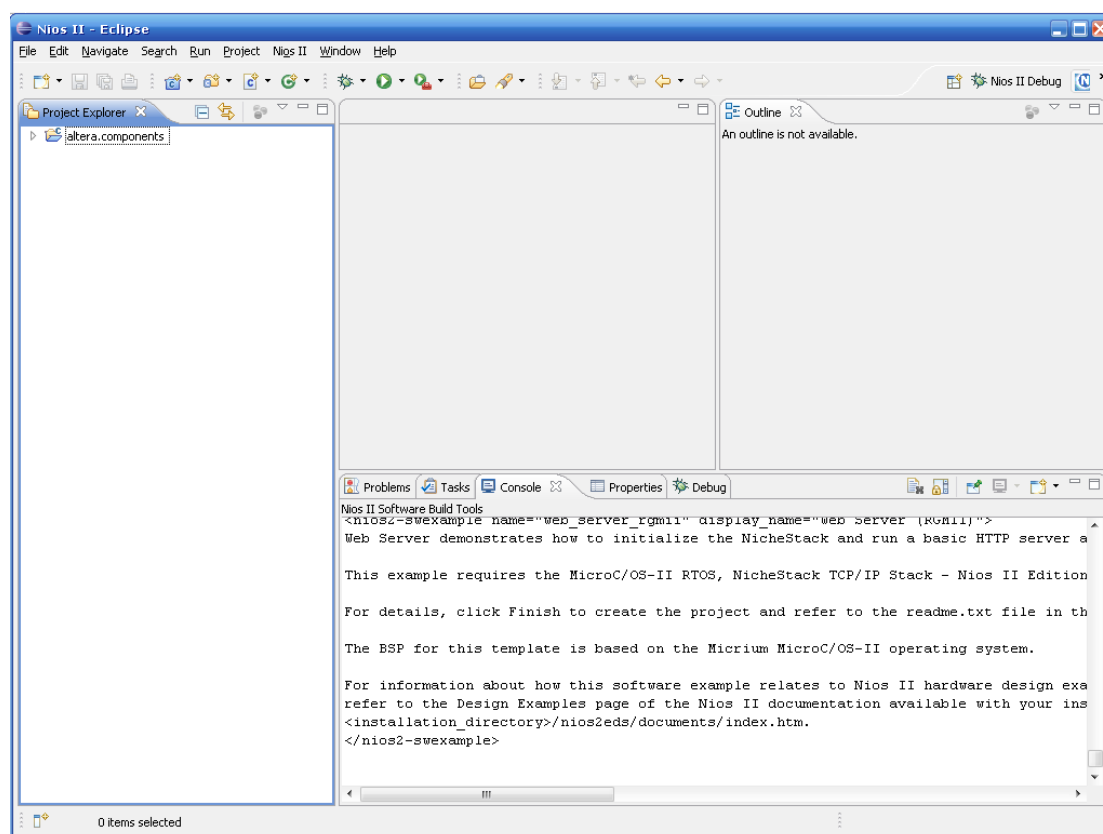


5.1.5 EDS 中新建软件工程

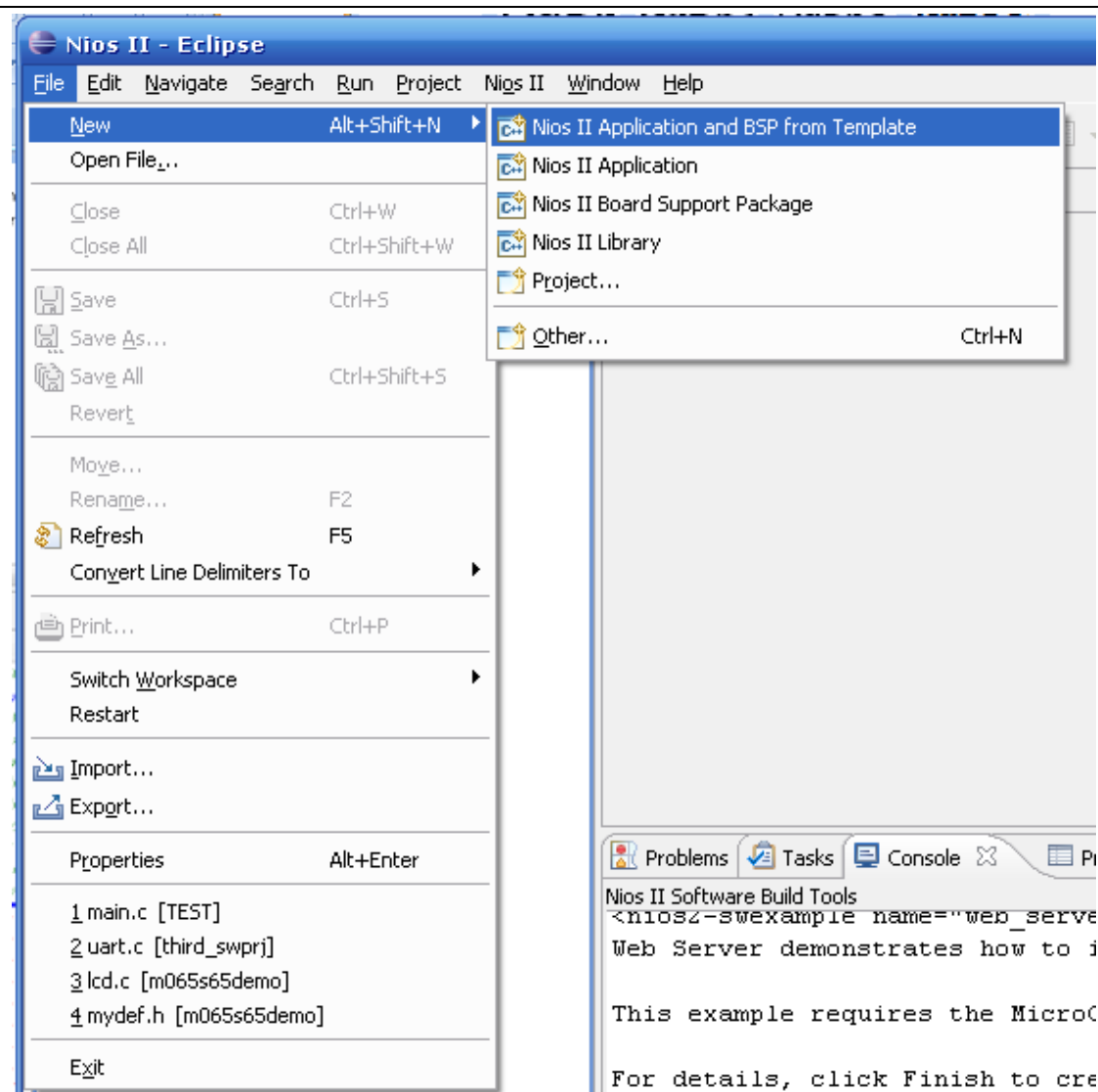
双击桌面上的 Nios II 12.0sp1 Software Build Tools for Eclipse 图标，打开 EDS 软件。首先弹出如下的 workspace 路径设置对话框，采用默认设置并点击 OK 进入软件。



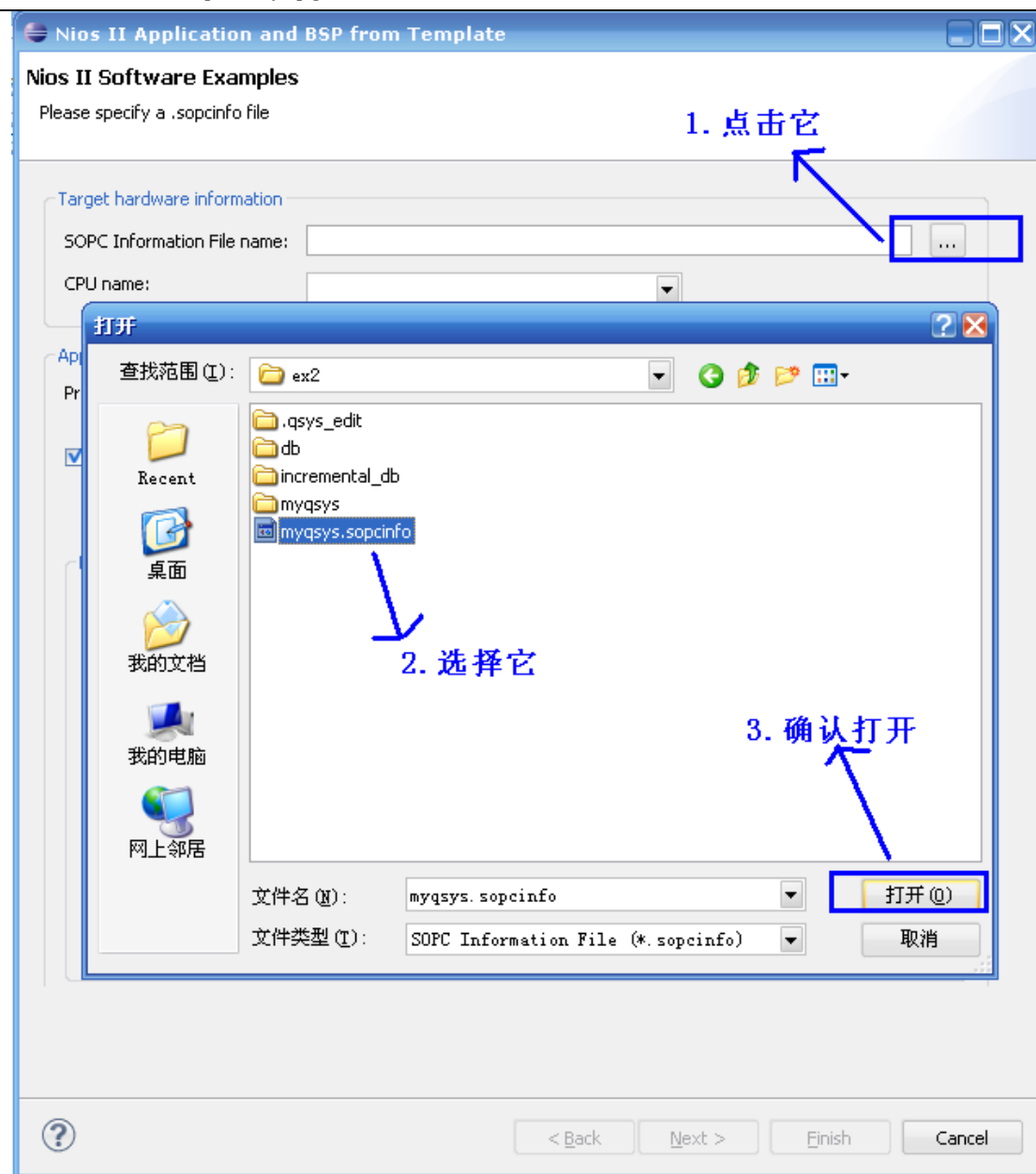
EDS 软件界面如图所示。



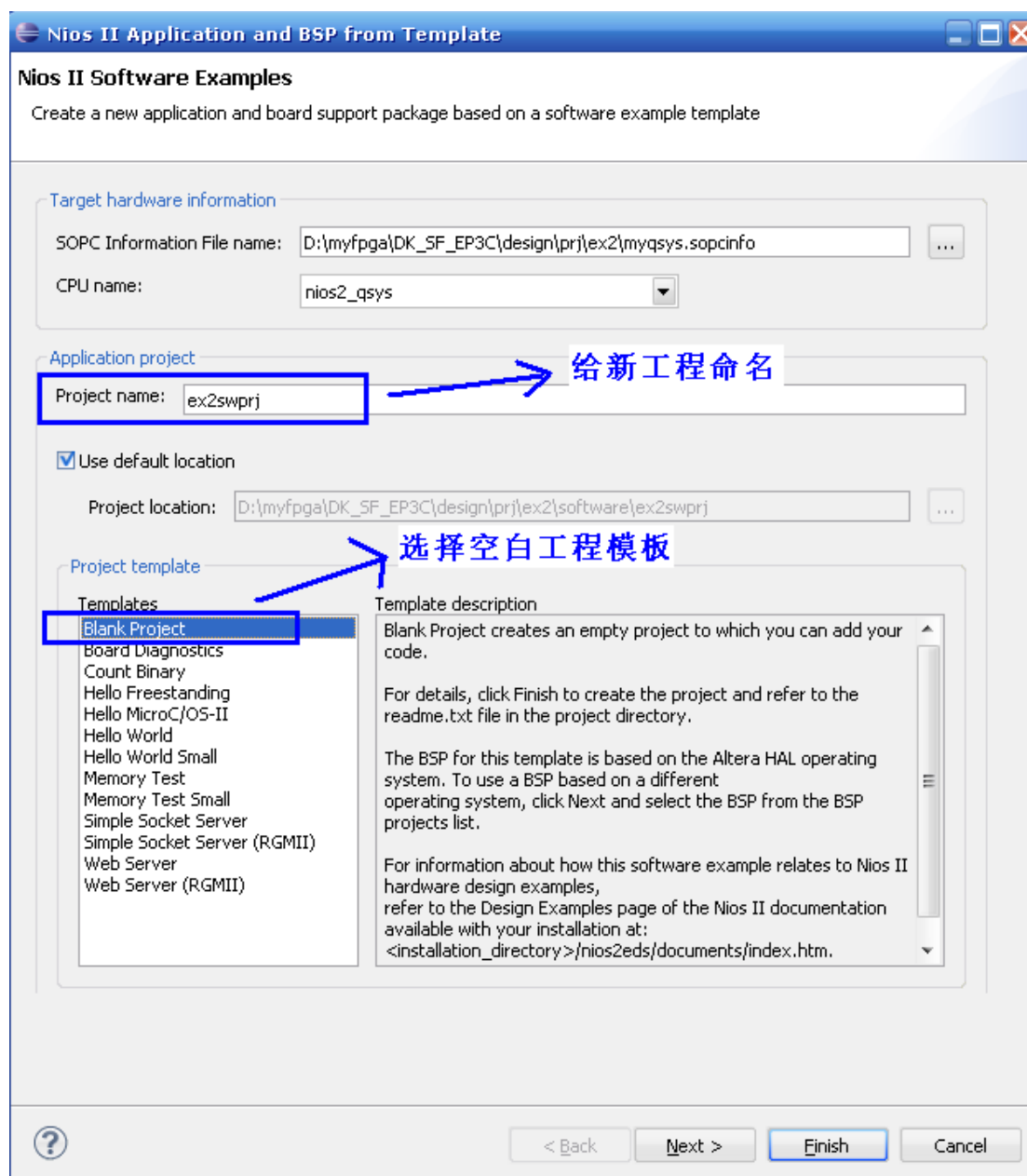
菜单栏点击 File→New→Nios II Application and BSP from Template 新建一个模板工程。



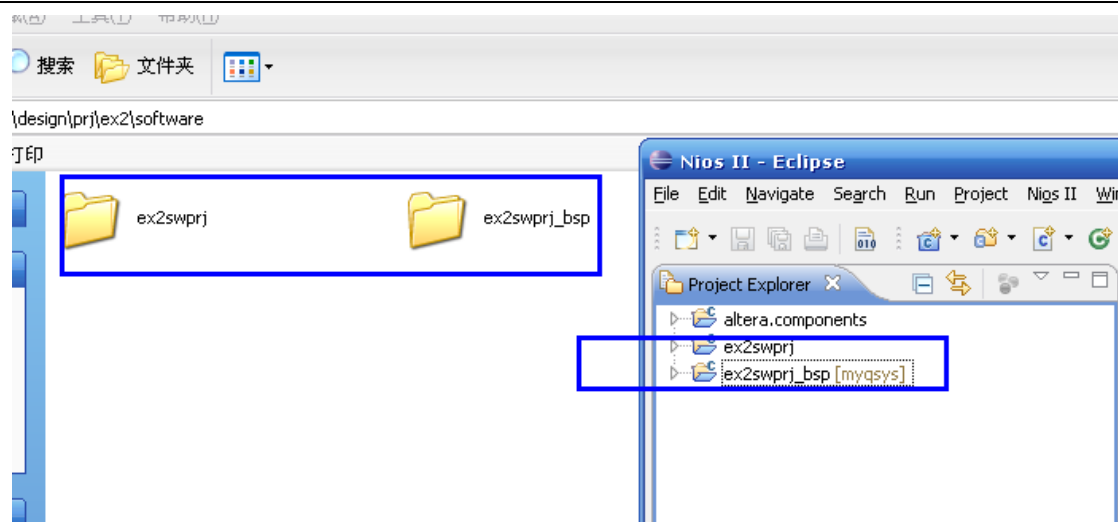
如图加载系统硬件，即 socinfo 文件。



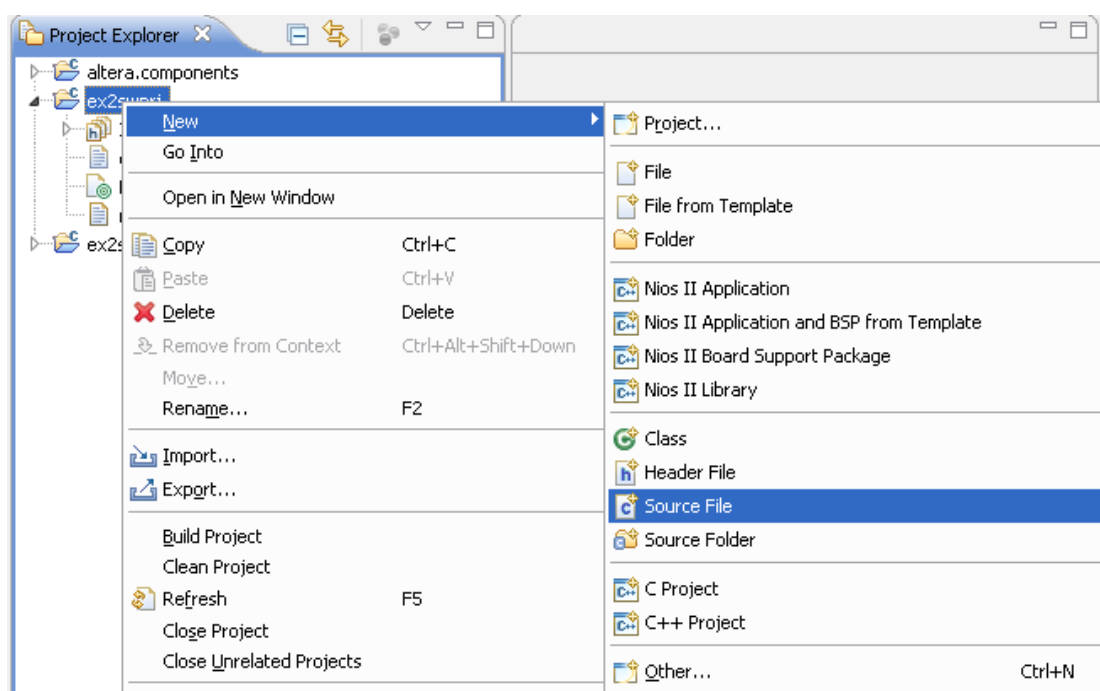
选择好 SOPC 文件后, 默认选中 CPU 为 nios2_qsys, 给软件工程命名为 ex3swprj, 最后再选择空白工程模板, 然后点击 Finish 完成工程的创建。



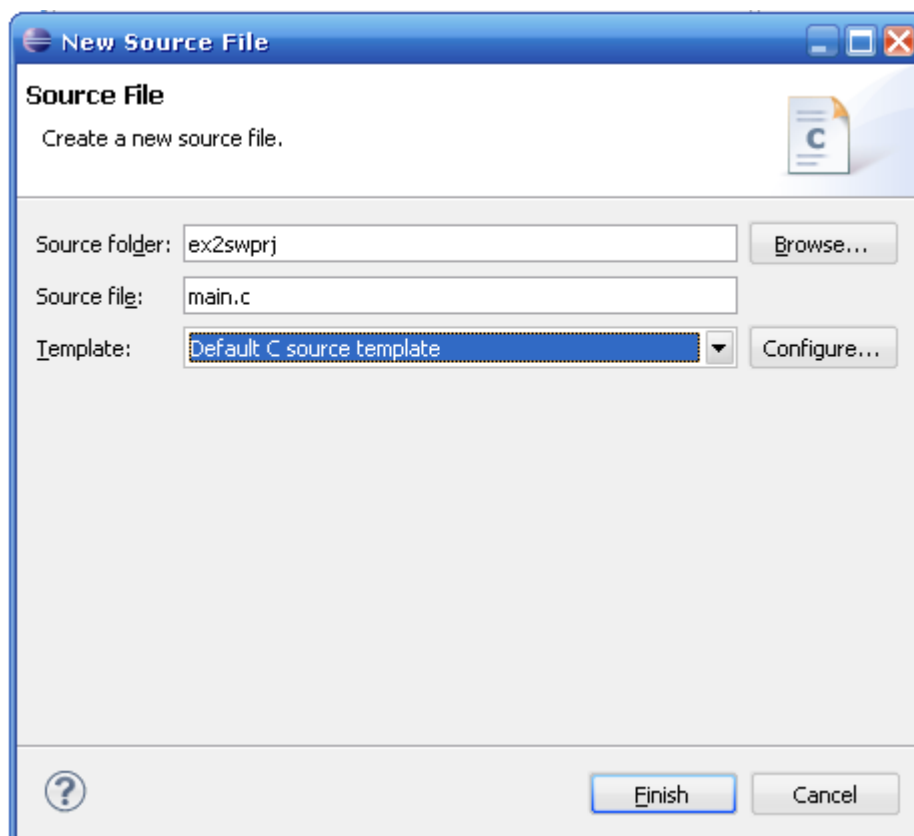
此时我们可以在工程文件夹下看到多了一个 **software** 文件下，它下面有两个文件夹 **ex3swprj** 和 **ex3swprj_bsp**，分别对应存放我们在 EDS 的 Project Explorer 下的两个工程名为 **ex3swprj** 和 **ex3swprj_bsp** 的新工程。**Ex3swprj** 工程是软件应用工程，我们在这个工程里面新建源代码；**ex3swprj_bsp** 工程则是存放硬件相关信息，里面包含了很多底层硬件驱动，用于衔接软硬件，我们一般不用去动它。



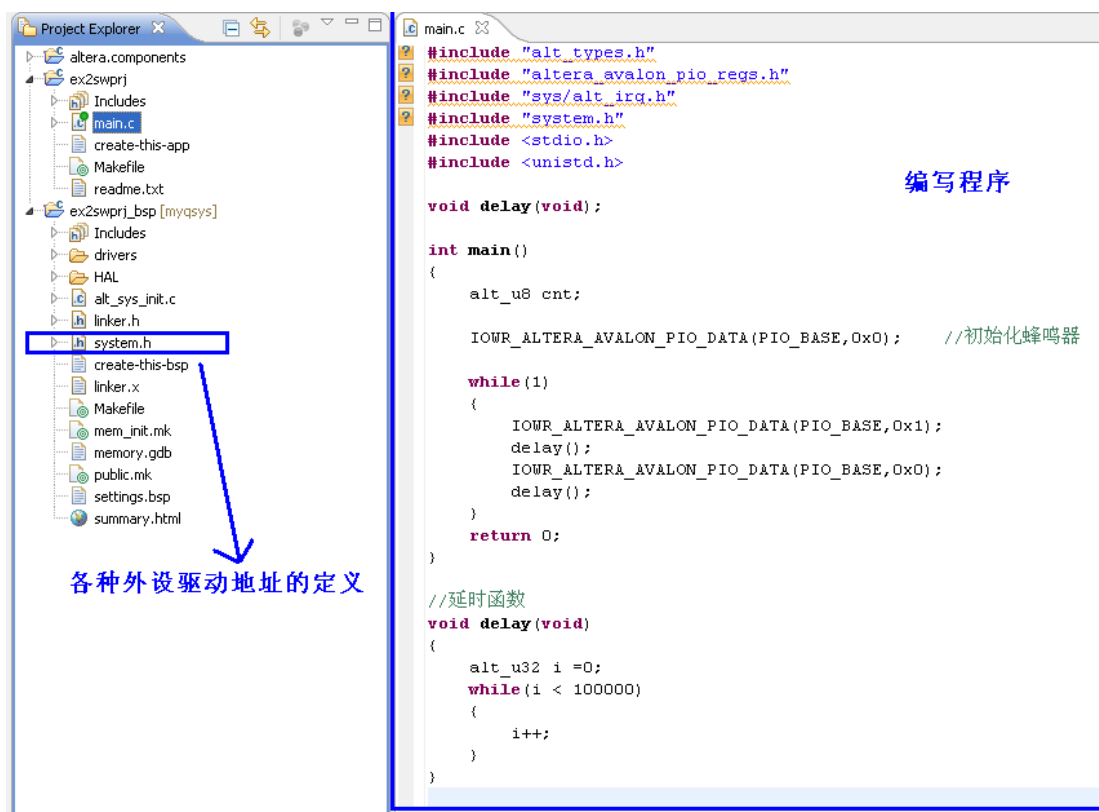
选中应用工程 `ex3swprj`, 点击右键弹出菜单, 选中 `New`→`Source File`。由此新建一个 `main.c` 的 C 文件。



新建 `main.c` 文件。



打开 main.c 文件, 编写软件程序。



我们这个实例要使用 PIO 脚控制 LED 闪烁, 因此可以编写软件程序如下。



```
/*
 * main.c
 *
 * Created on: 2013-1-1
 * Author: Administrator
 */

#include "alt_types.h"
#include "altera_avalon_pio_regs.h"
#include "sys/alt_irq.h"
#include "system.h"
#include <stdio.h>
#include <unistd.h>

void delay(void);

int main(void)
{
    IOWR_ALTERA_AVALON_PIO_DATA(PIO_BASE, 0x0);    //初始化 LED

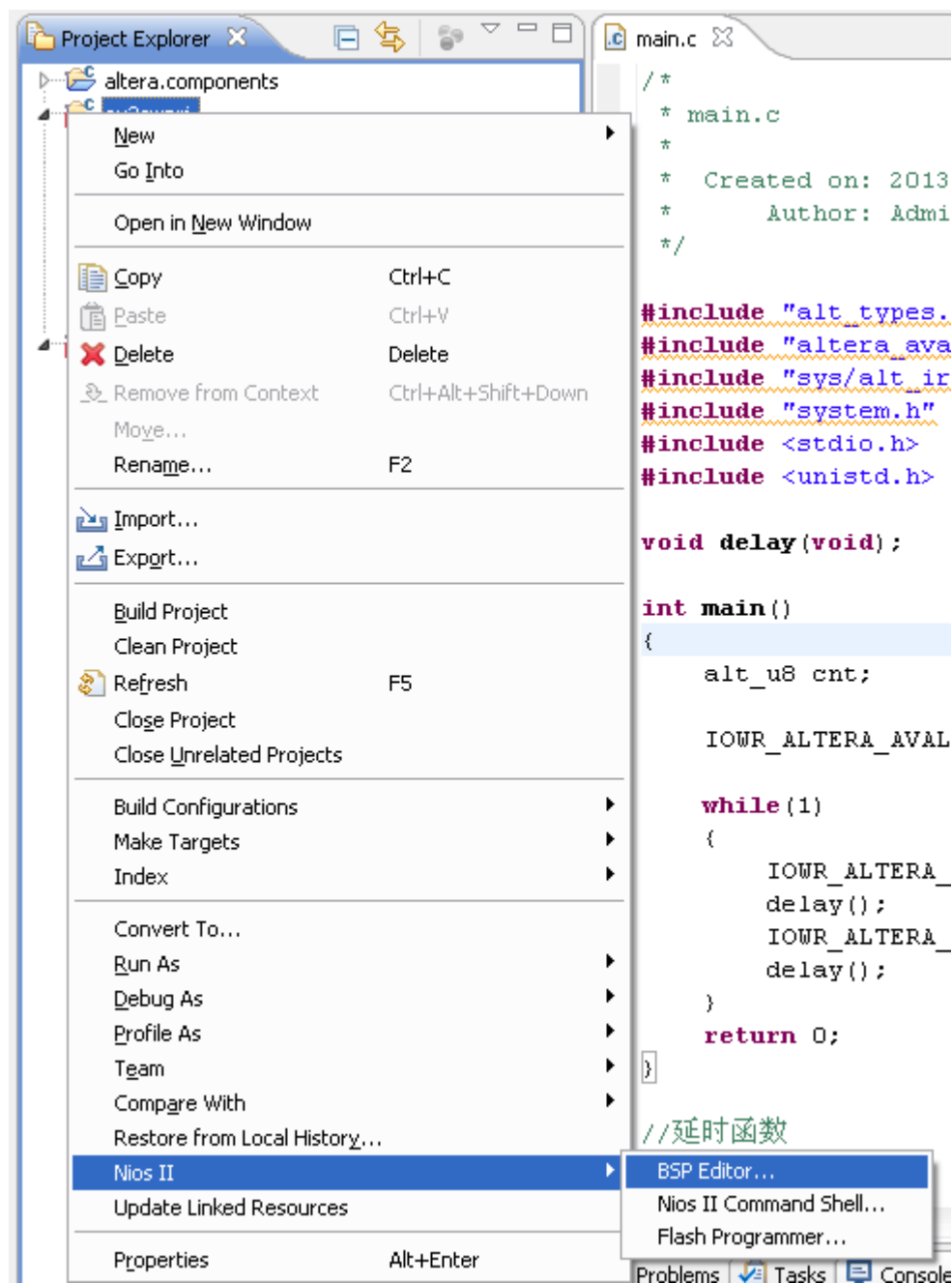
    while(1)
    {
        IOWR_ALTERA_AVALON_PIO_DATA(PIO_BASE, 0x1);
        delay();
        IOWR_ALTERA_AVALON_PIO_DATA(PIO_BASE, 0x0);
        delay();
    }
    return 0;
}

//延时函数
void delay(void)
{
    alt_u32 i =0;
    while(i < 100000)
    {
        i++;
    }
}
```

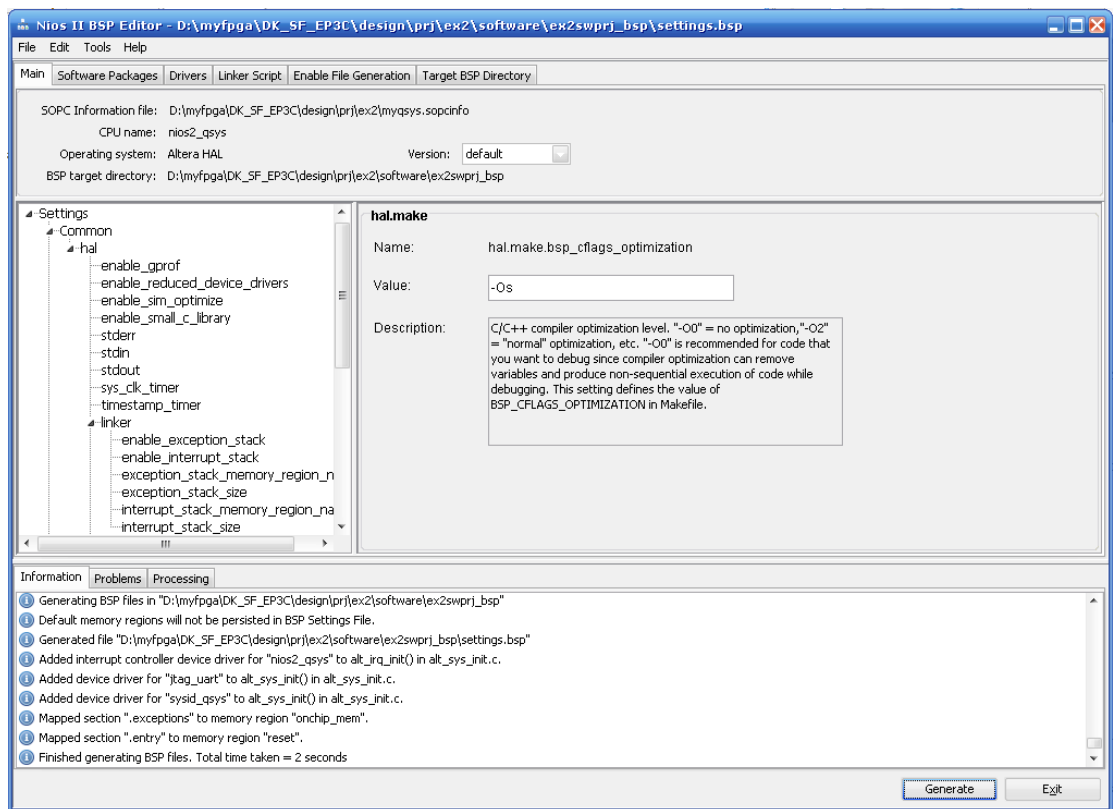


下一步, 别忙着编译, 先到 **BSP Editor** 中做一些设置, 裁剪一下代码, 否则一大堆没用的 **BSP** 会占用大量的代码控制, 以至于我们的 20KB 代码空间都不够用。

选中应用工程 **ex3swprj**, 点击右键弹出菜单, 选中 **Nios II**→**BSP Editor**。由此新建一个 **main.c** 的 C 文件。



进入 **BSP Editor**, 需要按照后面一个表格做设置, 完成后点击 **Generate** 按钮完成后 **Exit** 即可。



前面的 BSP Editor 中按照如下要求做设置。另外提醒大家注意的是，每次硬件工程（在 Quartus II 中）的任何更改，建议在编译软件工程前逗号重新在 BSP Editor 中 Generate 一次，这是为了保证软硬件的一致性，否则编译将出错。

Table 2-1. BSP Settings to Reduce Library Size

BSP Setting Name	Value
hal.max_file_descriptors	4
hal.enable_small_c_library	true
hal.sys_clk_timer	none
hal.timestamp_timer	none
hal.enable_exit	false
hal.enable_c_plus_plus	false
hal.enable_lightweight_device_driver_api	true
hal.enable_clean_exit	false
hal.enable_sim_optimize	false
hal.enable_reduced_device_drivers	true
hal.make.bsp_cflags_optimization	\ "-Os\"

接着回到 EDS 中，分别选中 ex3swprj_bsp 和 ex3swprj，然后右键点击选中 Build Project。记住，最好先去编译 ex3swprj_bsp 工程，然后编译应用工程 ex3swprj。最后编译完应用工程，我看到占用的代码空间仅有 540Byte。



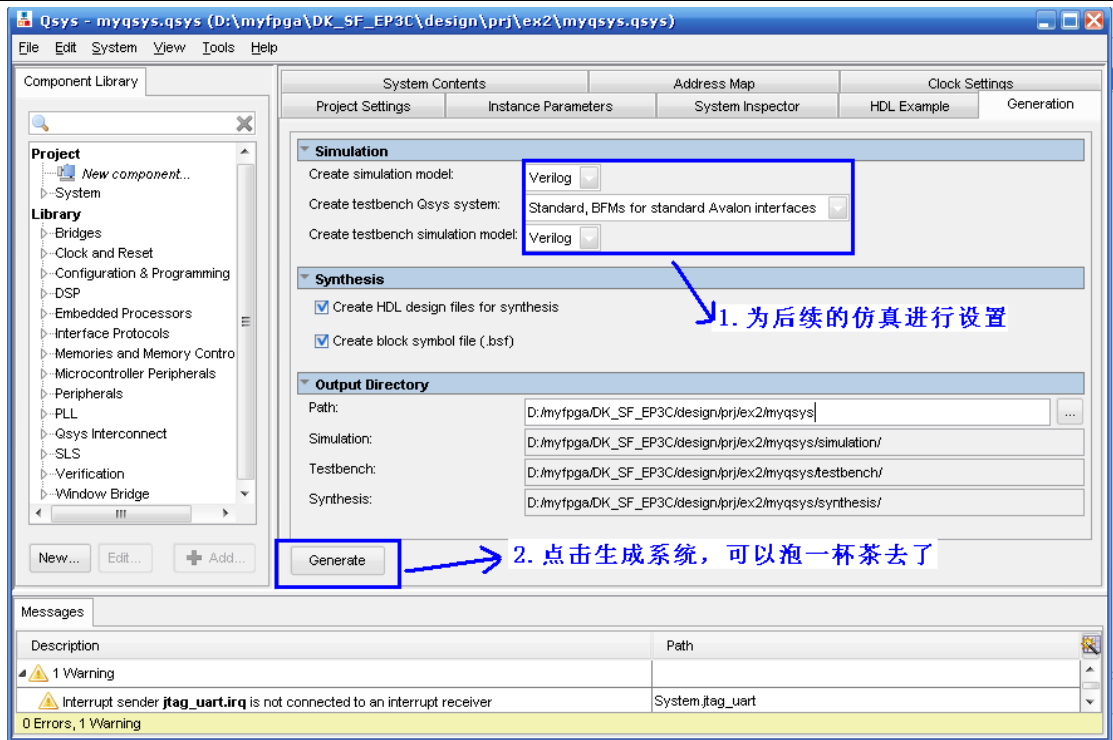
```
C-Build [ex2swprj]
-msys-crt0=../ex2swprj_bsp/obj/nal/sic/crt0.o -msys-lib=nal_bsp -L../ex2swprj_bsp/
-Wl,--defsym,exit=_exit -msmallc -Wl,-Map=ex2swprj.map -OO -g -Wall -EL
-mno-hw-div -mno-hw-mul -mno-hw-mulx -o ex2swprj.elf obj/default/main.o -lm
nios2-elf-insert ex2swprj.elf --thread_model hal --cpu_name nios2_qsys --qsys true -
-simulation_enabled false --id 0 --sidp 0x11018 --timestamp 1357005201 --stderr_dev
jtag_uart --stdin_dev jtag_uart --stdout_dev jtag_uart --sopc_system_name myqsys -
-quartus_project_dir "D:/myfpga/DK_SF_EP3C/design/prj/ex2" --jdi
D:/myfpga/DK_SF_EP3C/design/prj/ex2/ex2.jdi --sopcinfo
D:/myfpga/DK_SF_EP3C/design/prj/ex2/myqsys.sopcinfo
Info: (ex2swprj.elf) 540 Bytes program size (code + initialized data).
Info: 19 KBytes free for stack + heap.
Info: Creating ex2swprj.objdump
nios2-elf-objdump --disassemble --syms --all-header --source ex2swprj.elf
>ex2swprj.objdump
[ex2swprj build complete]
```

5.1.6 ModelSim 仿真

仿真在 FPGA 设计过程中举足轻重,在板级调试前若不好好花功夫做一些前期的验证和测试工作,后期肯定要不断的返工甚至推倒重来,这是 FPGA 设计的迭代特性所决定的。因此,在设计的前期做足了仿真测试工作,虽然不能完全避免后期问题和错误的发生,却能够大大减少后期调试和排错的工作量。

逻辑设计中需要做仿真,是因为逻辑设计大都是设计者原型开发的,不做仿真的话设计者肯定心里也没底。而用 Qsys 搭建的系统多是由已经成熟验证过的 IP 核组成的,还需要仿真否?这是个仁者见仁智者见智的问题,但是做过仿真后总是能够更让人放心一些。别人说得不算,我动手验证过的才算。

如图所示,首先在 Generations 的 Simulation 选项中做好设置(前面的步骤我们已经强调过)。



Simulation 设置选项的具体含义如下表所示。

Table 5-4. Summary of Simulation Settings on Qsys Generation Tab

Simulation Setting	Value	Description
Create simulation model	None Verilog VHDL	Creates simulation model files and simulation scripts. Use this option to include the simulation model in your own custom testbench or simulation environment. You can also use this option to generate models for a testbench system that you modify.
Create testbench Qsys system	Standard, BFM for standard Avalon interfaces	Creates testbench Qsys system with Avalon BFM attached to all exported interfaces. Includes any simulation partner modules specified by IP cores in the system. This testbench is not supported with VHDL simulation models.
	Simple, BFM for clocks and resets	Creates testbench Qsys system with Avalon BFM driving only clocks and reset interfaces. Includes any simulation partner modules specified by IP cores in the system.
Create testbench simulation model	None Verilog VHDL	Creates simulation model files and simulation scripts for the testbench Qsys system specified in the setting above. Use this option if you do not need to modify the Qsys-generated testbench before running the simulation.

因为都是用 Verilog，所以 simulation model 和 testbench simulation model 我们都选择 Verilog，话说 Altera 其实主打的是 Verilog 语言，所以各种功能对 Verilog 的支持都是非常到位的，VHDL 就不一定了。也用过 Xilinx 的东西，则正好相反。也不能谈论孰优孰劣，也是习惯使然。

Standard 和 Simple 的模型主要差别在于后者只是简单的在 testbench 里产生 clock 和 reset 信号，而前者则会对所有 export 信号产生激励或引出便于监视观察。

确定完成 Simulation 的设置后就可以点击左下角的 Generate 重新生成系统。

系统生成完毕，到“工程目录\myqsys\testbench\myqsys_tb\simulation”这个路径下有《圣经》箴言九 11 “敬畏耶和华是智慧的开端，认识至胜者便是聪明。”

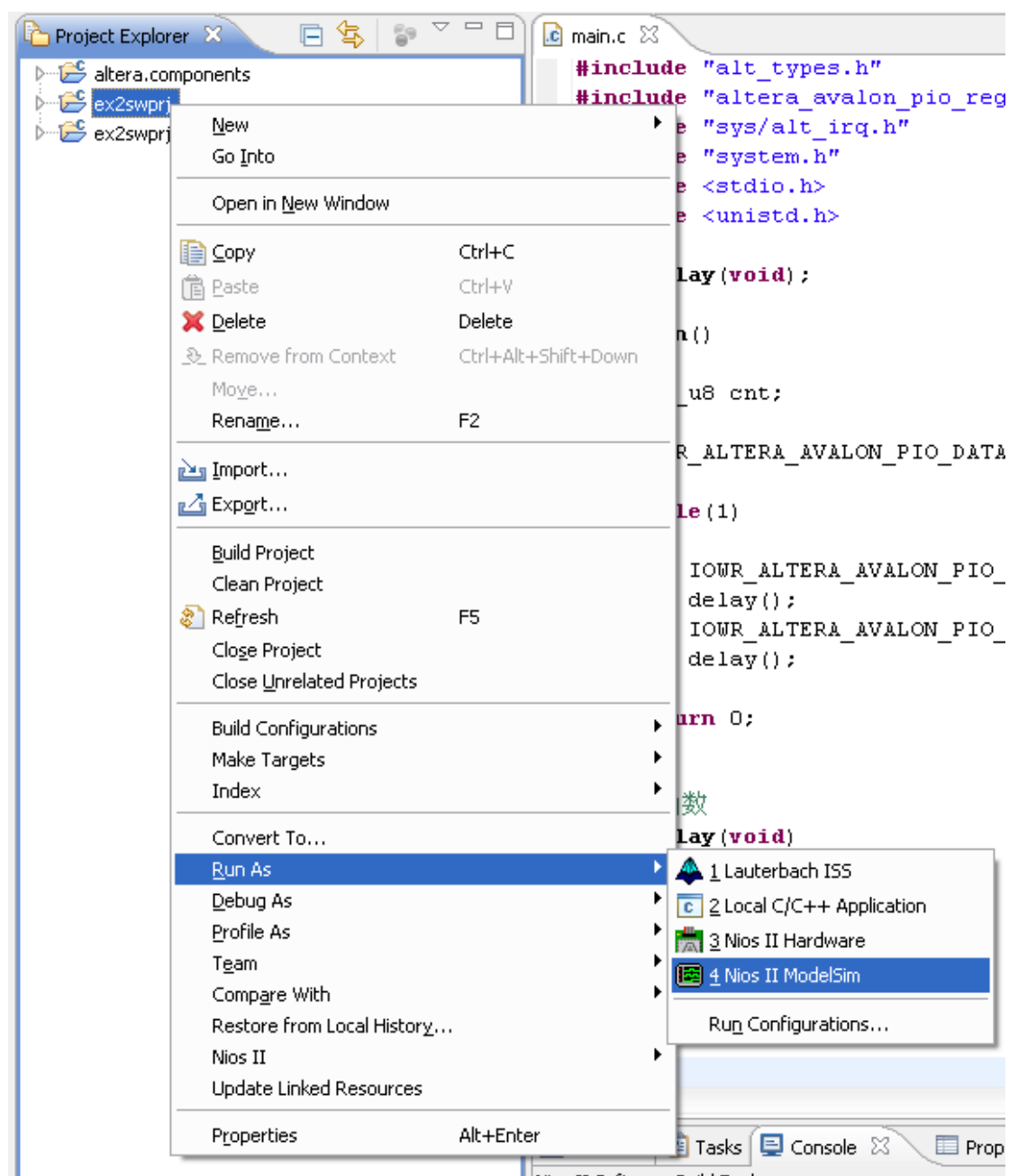


测试脚本的顶层文件 `myqsys_tb.v`。由于脚本较长, 所以这里不罗列了, 大家可以自己去看看。

该测试脚本中首先例化了被测试系统 `myqsys`, 将其 3 个 `export` 信号引出。然后分别针对这 3 个 `export` 信号产生相应的激励和响应, 即 `altera_avalon_clock_source` 用于产生 `clock`, `altera_avalon_reset_source` 用于产生 `reset` 信号, `altera_conduit_bfm` 则用于观察 `led_pio` 输出。这三个模块的详细代码都可以在同目录的 `submodules` 子文件夹下找到。

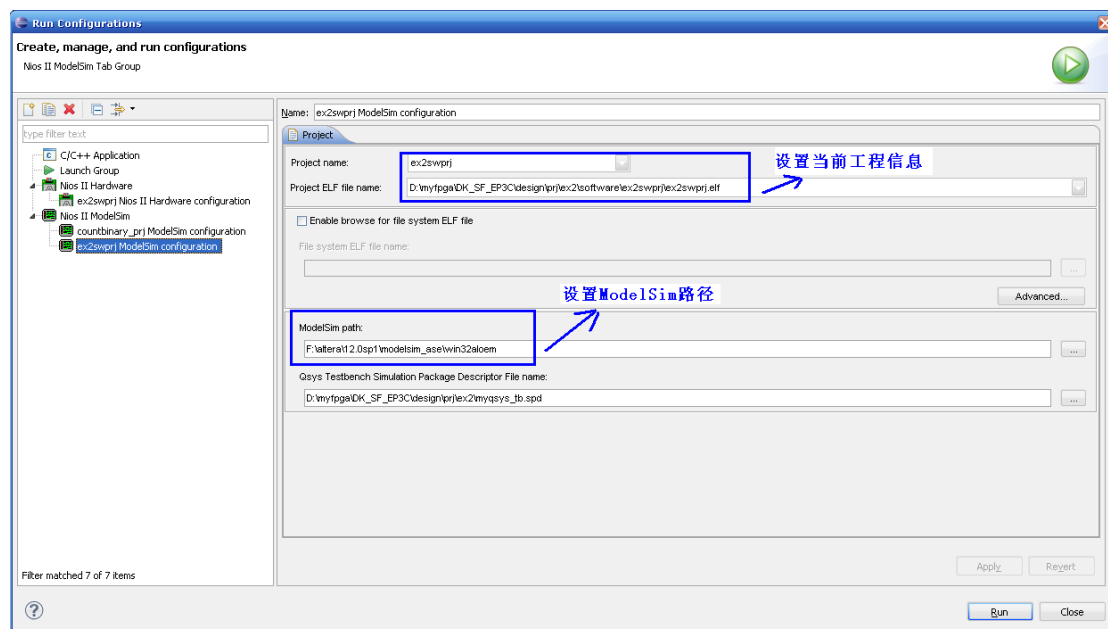
接下来, 我们需要打开 EDS 中的软件工程并对其进行仿真。

在应用工程上点击右键并选择 `Rus as→Niso II ModelSim`。

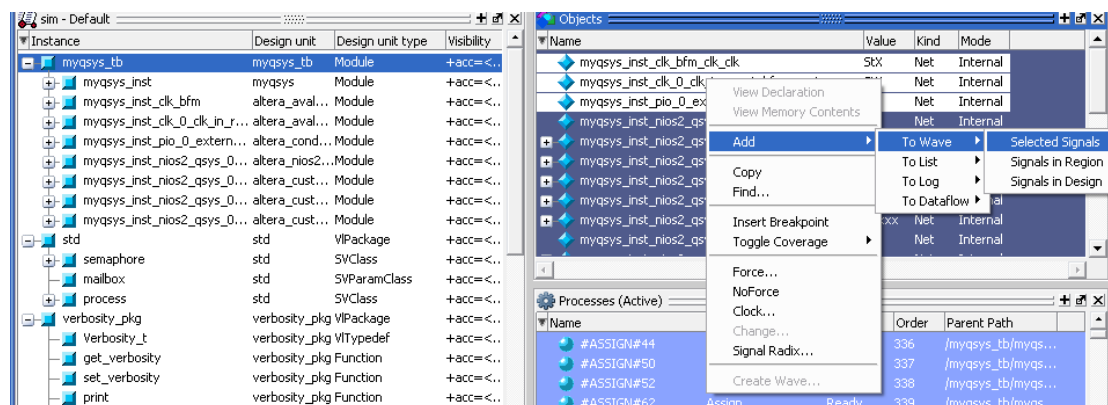




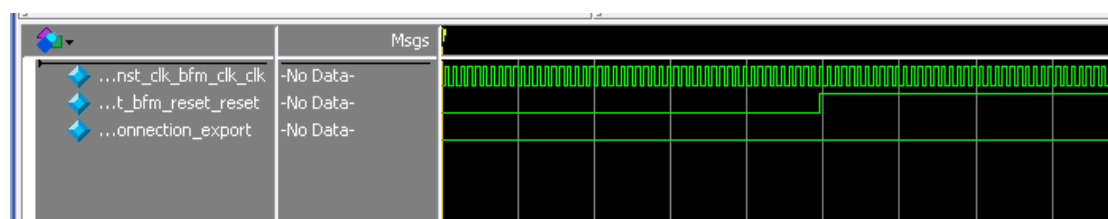
第一次运行通常会弹出如图所示的窗口, 需要对仿真选项做一些配置。选择选择仿真的工程名 (Project name)、仿真的 elf 可执行文件 (Project ELF file name)、ModelSim 软件的安装路径 (ModelSim path) 和 Qsys 测试脚本封装描述文件 (Qsys Testbench Simulation Package Descriptor File name) 存储位置。设计好后点击 Run 即启动 ModelSim 进行仿真。



弹出 ModelSim-Altera 后, 我们可以讲顶层文件的 3 个 export 信号添加的 Wave 窗口中, 然后让仿真重新 Run 起来 (参考前面两节相关操作)。

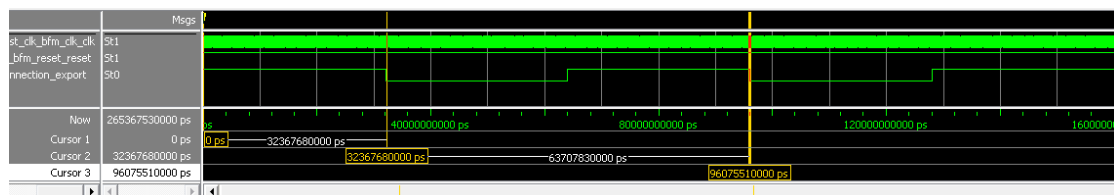


如图, 刚上电 0ns 开始, reset 信号有一段时间的低脉冲, 大约 50 个 clk 周期, 正如我们的 testbench 中所设计的。





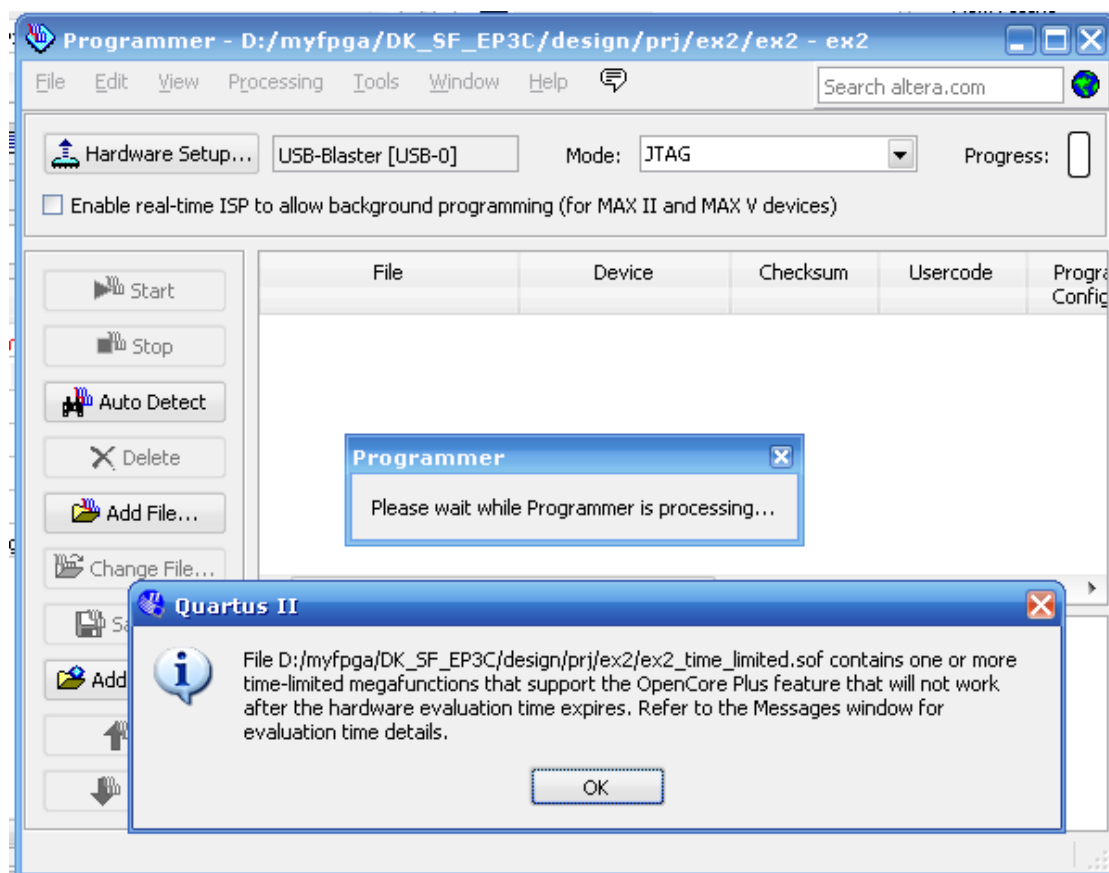
再来看 PIO 信号的变化, 一个 PIO 脉冲的周期大约是 73ms, 也就是 LED 闪烁的周期时 73ms, 要比前面两个实例快很多。



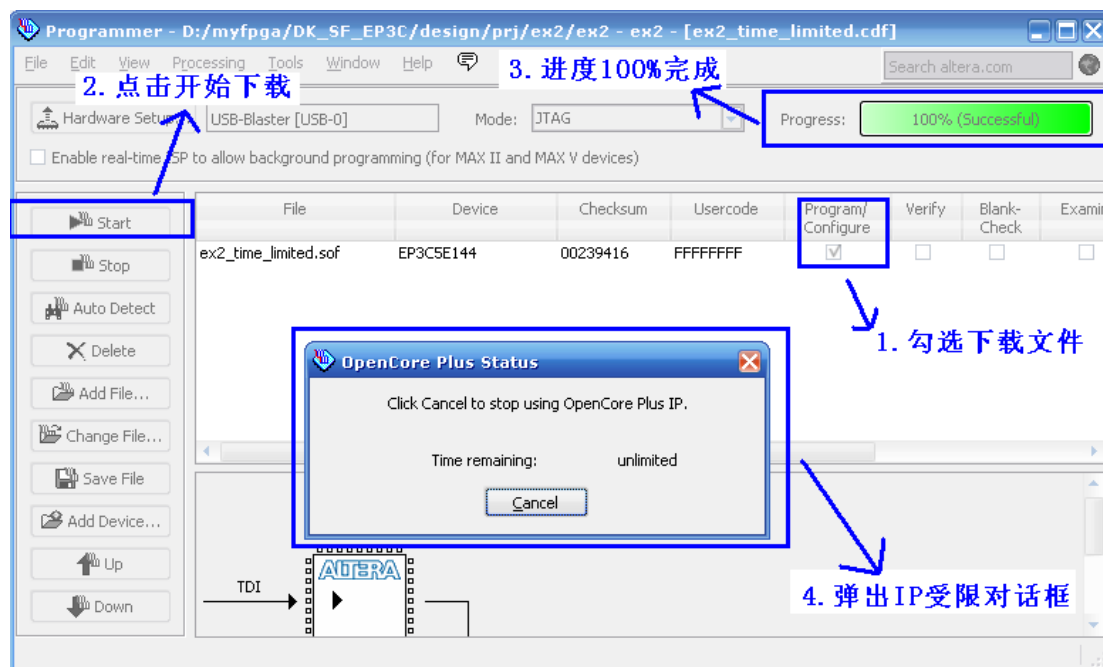
5.1.7 下载配置与板级调试

这里的下载都是在线配置, 分作两步执行。首先是在 Quartus II 中把硬件系统生成的 sof 文件下载到 FPGA 中, 然后在 EDS 里进行在线调试。

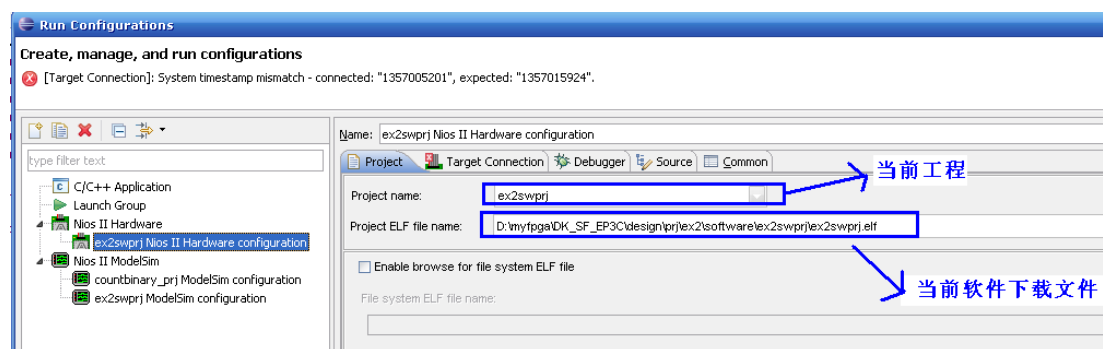
在 Quarts II 点击菜单栏 Tools→Programmer, 会弹出一个提示框, 不用理会它, 点击 OK 进入。



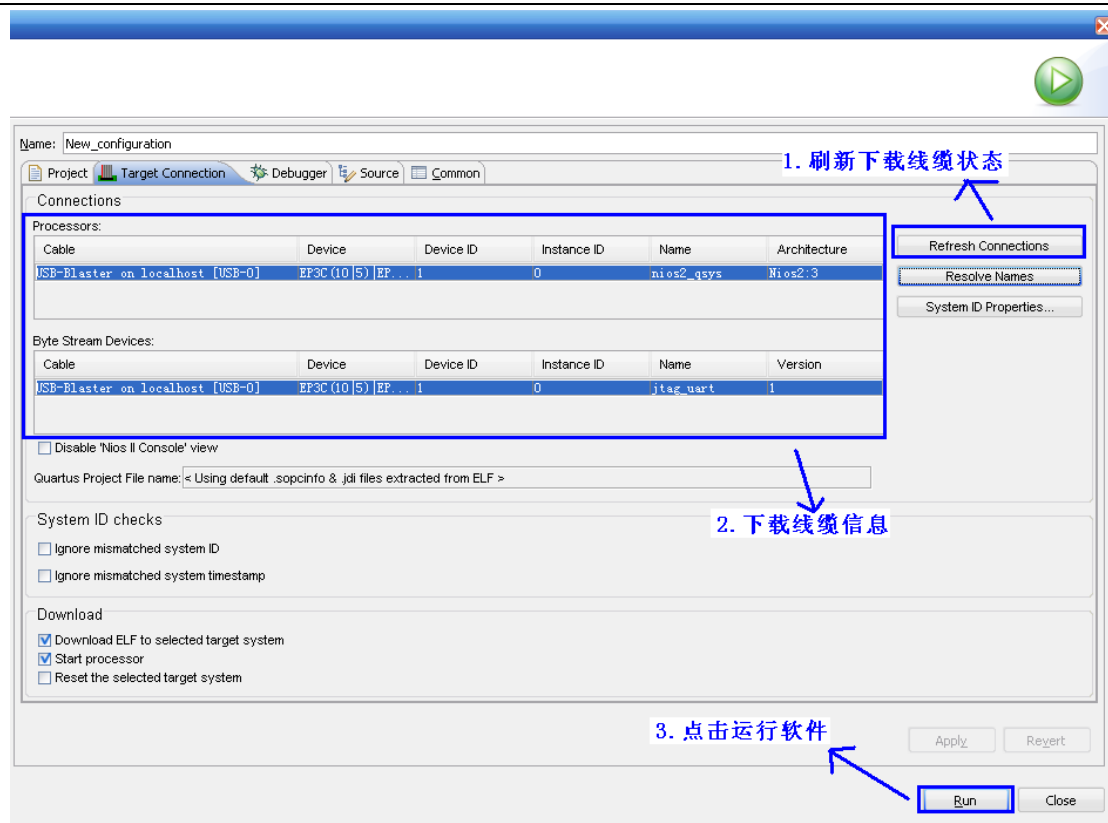
确认下载线缆连接, 给板子上电, 然后执行下载操作, 知道弹出如图所示第 4 步的对话框表示下载成功。



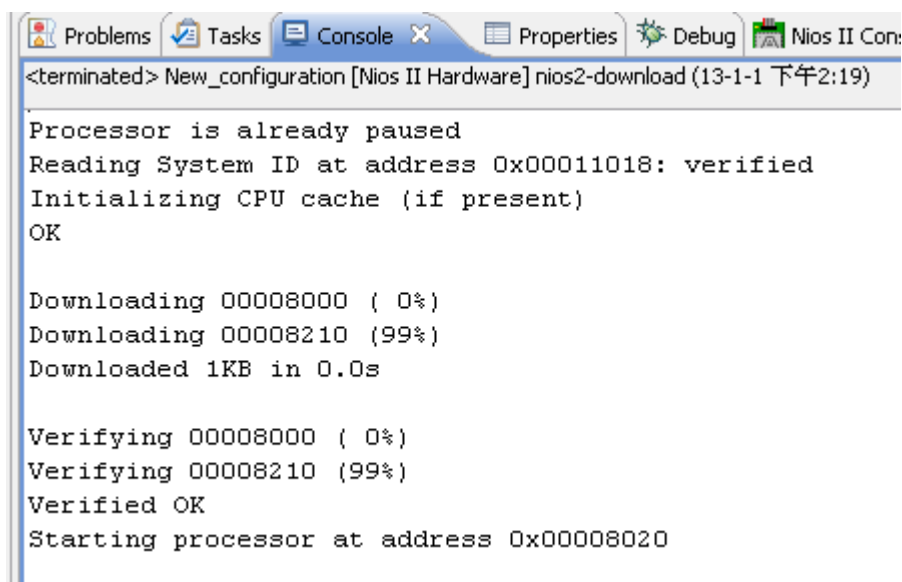
在 EDS 里, 选中应用工程, 右键点击并选择 **Rus as**→**Niso II Hardware**。第一次操作通常会弹出 **Run Configuration** 界面, 首先在 **Project** 一栏中选择好当前工程和当前的软件下载文件 (在应用工程目录下的 **elf** 文件)。



在 **Target Connection** 页面, 需要检测下载线缆。可以多次点击右侧的 **Refresh Connections** 按钮, 直到左侧的 **Connections** 下出现我们的下载线信息, 此外, 我们还需要让左下角的 **Run** 按钮有效 (即变亮可以点击)。点击 **Run** 开始软件的在线运行。



此时我们再看 EDS 下面的 Console 会有在线下载成功的状态提示。



我们可以看看 SF-CY3 开发板, 是否 LED 非常快速的在闪烁。恭喜你, 你的 NIOS II 系统已经 RUN 起来了。

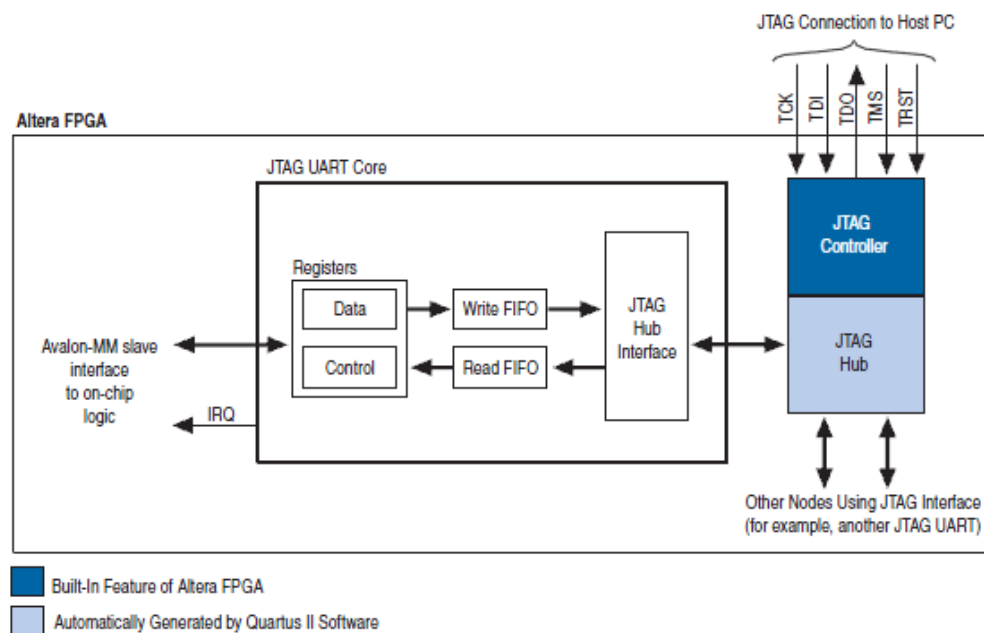


5.4 基于 Qsys 的 NIOS II 实例 2——Hello NIOS II

5.4.1 JTAG UART 外设概述

前面在 Qsys 架构系统时，我们添加了一个名为 JTAG UART 的外设，不知道大家对这个外设的了解有多少，先给个参考文档，真正想学习的人可以在这个文档中找到 Altera 的所有 IP 核的介绍和说明。

言归正传，来看看这个 JTAG UART 的模块框图。它虽然是个 IP 核（也是要占用逻辑资源的一部分），但它通过 FPGA 外部引出的 JTAG 管脚来实现和 PC 机的通信传输。它的 IP 核内部有一个 JTAG 协议的接口控制模块，读写 FIFO 用于缓存数据，还有用于连接到系统的 Avalon-MM 总线上的接口逻辑。



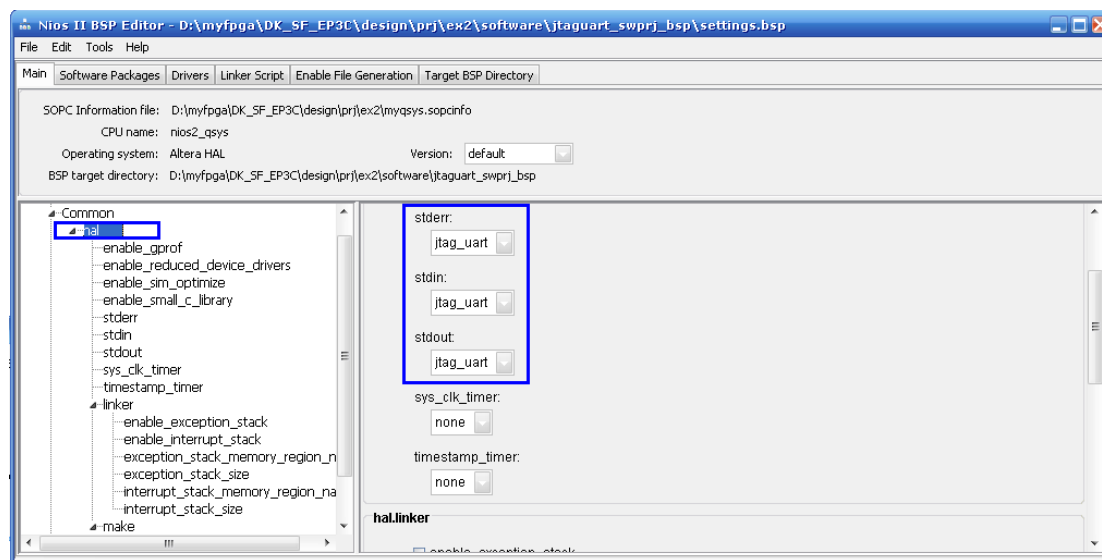
NIOS II 处理器可以通过 Avalon-MM 总线来访问 JTAG UART 外设，从而与 PC 端的 EDS 传输数据。在我们调试过程中，如果有这样一个接口协助打印一些调试中间过程的信息，那么无疑能够大大提高调试效率。

本节我们就要通过一个简单的实例来看看他是如何将 NIOS II 处理器内的信息打印到 EDS 中的。



5.4.2 编写软件代码

我们沿用上一个工程的硬件系统，参照上一节的操作在 EDS 中新建一个软件工程，命名为 jtaguart_swprj。然后打开工程的 BSP Editor 做裁剪代码的设置，需要特别注意 stderr、stdin 和 stdout 位置的设置一定都是 jtag_uart。因为这里的设置关系到我们调用驱动层的一些函数时，它实际底层相关联的硬件外设。



回到 EDS，我们在应用工程中新建一个 main 函数，实现一个定时打印 Hello NIOS II 字符串的信息，代码如下。

```
/*  
 * main.c  
 *  
 * Created on: 2013-1-1  
 * Author: Administrator  
 */  
  
#include "alt_types.h"  
#include "altera_avalon_pio_regs.h"  
#include "sys/alt_irq.h"  
#include "system.h"  
#include <stdio.h>  
#include <unistd.h>  
  
void delay(void);
```



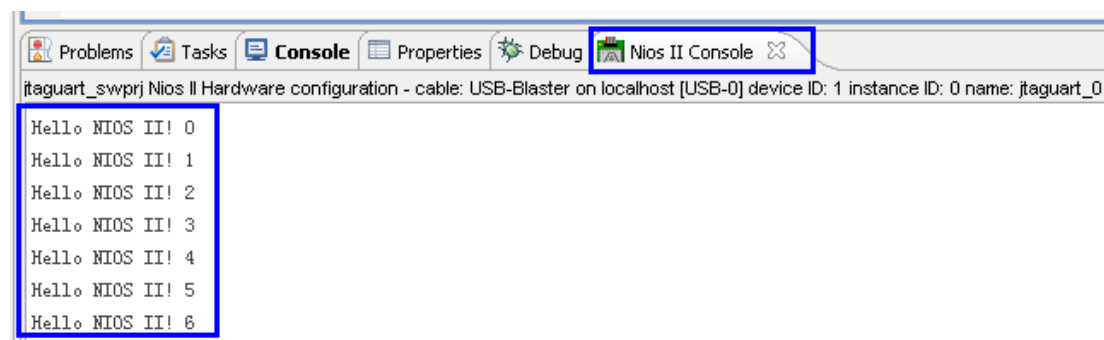
```
int main(void)
{
    alt_u8 timer = 0;

    while(1)
    {
        printf("Hello NIOS II! %d\n", timer);
        delay();
        timer++;
    }
    return 0;
}

//延时函
void delay(void)
{
    alt_u32 i=0;
    while(i < 4000000)
    {
        i++;
    }
}
```

5.4.3 下载配置与板级调试

首先将系统的 sof 文件下载到 SF-CY3 开发板中,接着我们在线将软件工程 Run 起来,此时注意观察 Nios II Console 窗口,不断的打印“Hello NIOS II!”的字符串,同时后面会有一个递增的计数值显示,这个也是我们软件程序中设置的一个计数器。



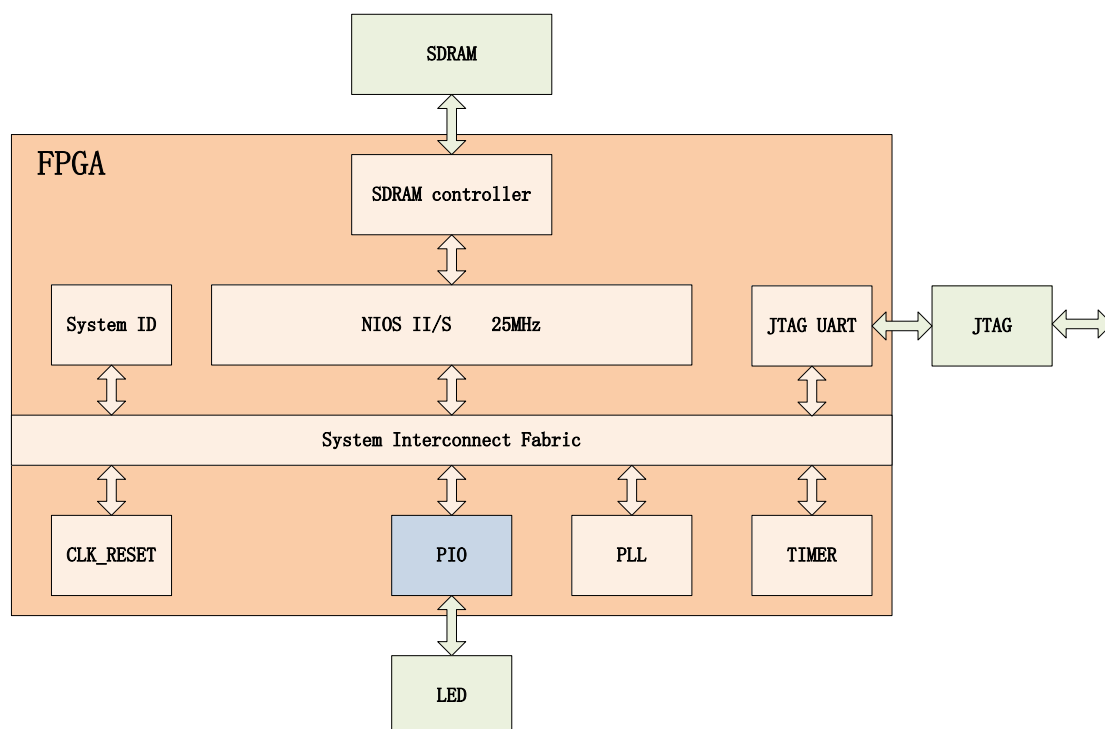


5.5 基于 Qsys 的 NIOS II 实例 3——集成 SDRAM 外设

5.5.1 系统概述

前面的 NIOS II 系统都是使用了片内的 RAM 做存储器运行程序, 由于 FPGA 片内的存储资源相对匮乏, 而且性价比极低, 因此在真正的应用中, 除非非常小的应用, 基本都是使用外部的存储器来跑程序的。而存储器中性价比相对较好的就要数 SDRAM 了, 速度快、容量大而且价格低, 适合大都数的嵌入式系统应用。只不过唯一的缺点是控制起来稍显复杂, 不过不用担心, 这一节我们不用自己写 SDRAM 控制逻辑, Qsys 中集成了这样的 IP 核, 和前面的组件添加方式一样非常简单就能够拿来就用。

在这个新的系统中, 原来的 onchip_mem 组件删除, 添加了 SDRAM Controller 组件, 它会预留一组接口到 FPGA 的管脚上, 最终要引到外部的 SDRAM 上。由于 SDRAM 的频率通常较高, 所以我们这个实例将使用 PLL 产生 100MHz 的时钟用于 SDRAM 的控制, 这个 PLL 也挂在系统的总线上。此外, 我们还添加了一个 TIMER 即定时器组件, 用于后续的实例。

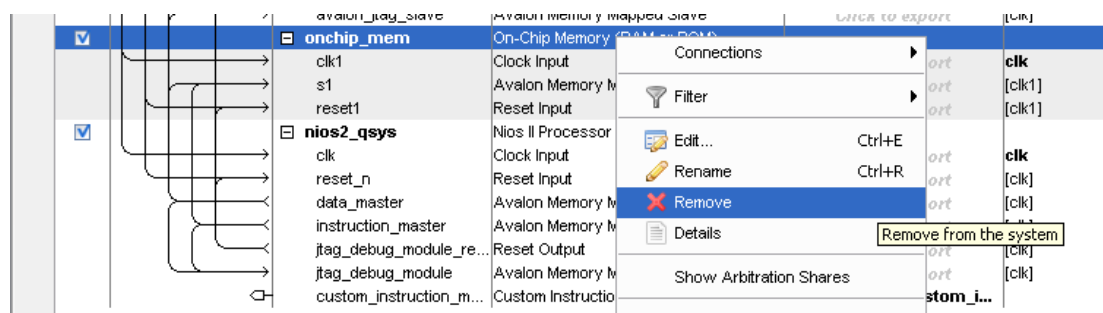


在开始这个实例之前, 我们不用新建 Quartus II 工程, 只要将上一个工程 ex2 拷贝一份, 然后改名为 ex3 即可。接着我们可以打开工程, 进入 Qsys。

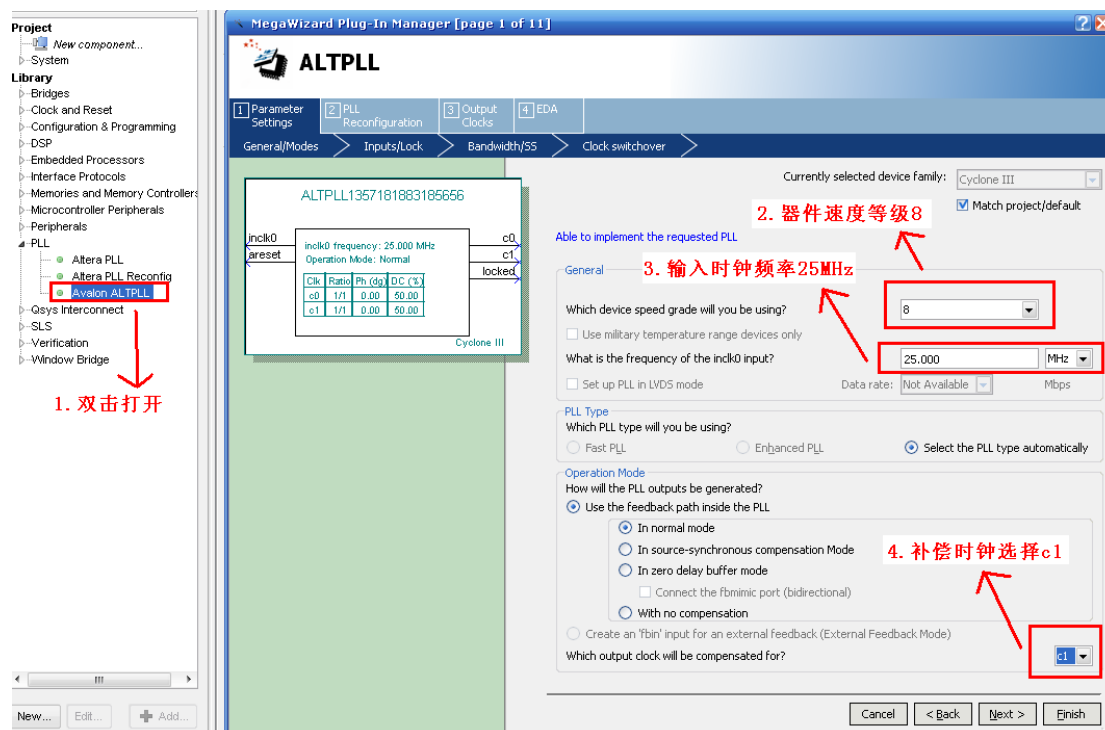


5.5.2 Qsys 组件添加

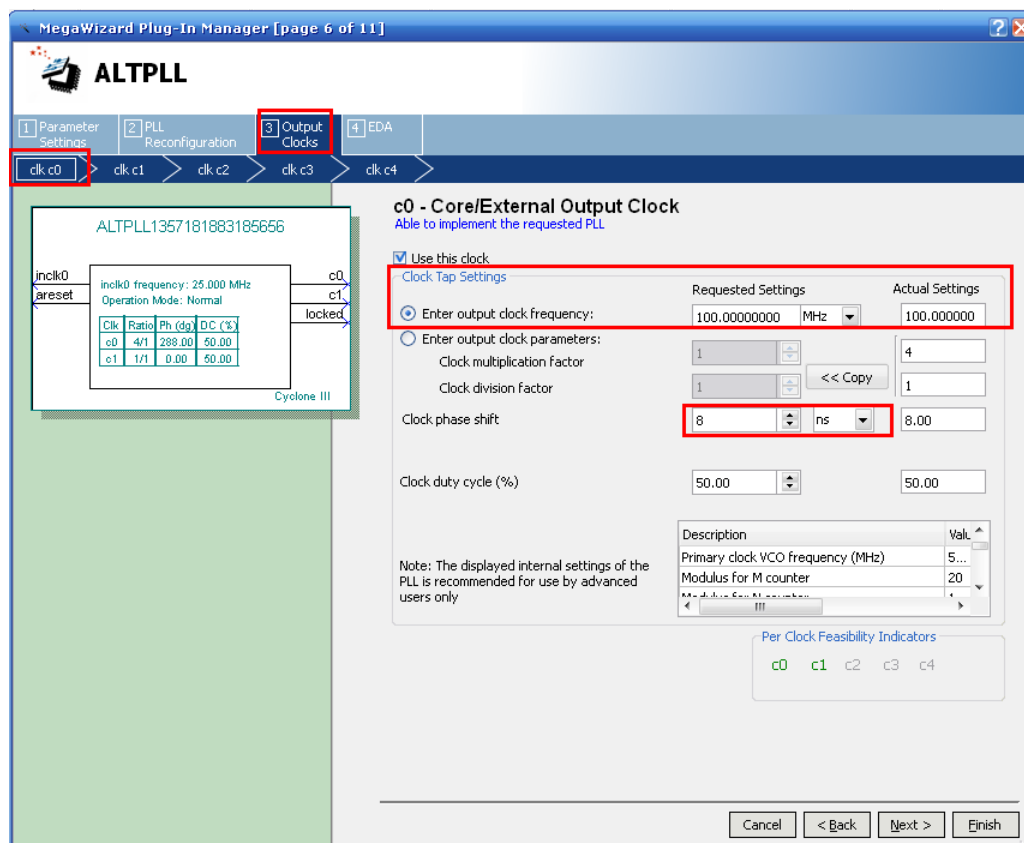
我们先删除 onchip_mem 这个组件，选中这个组件，然后点击右键，选中 Remove 删除它。



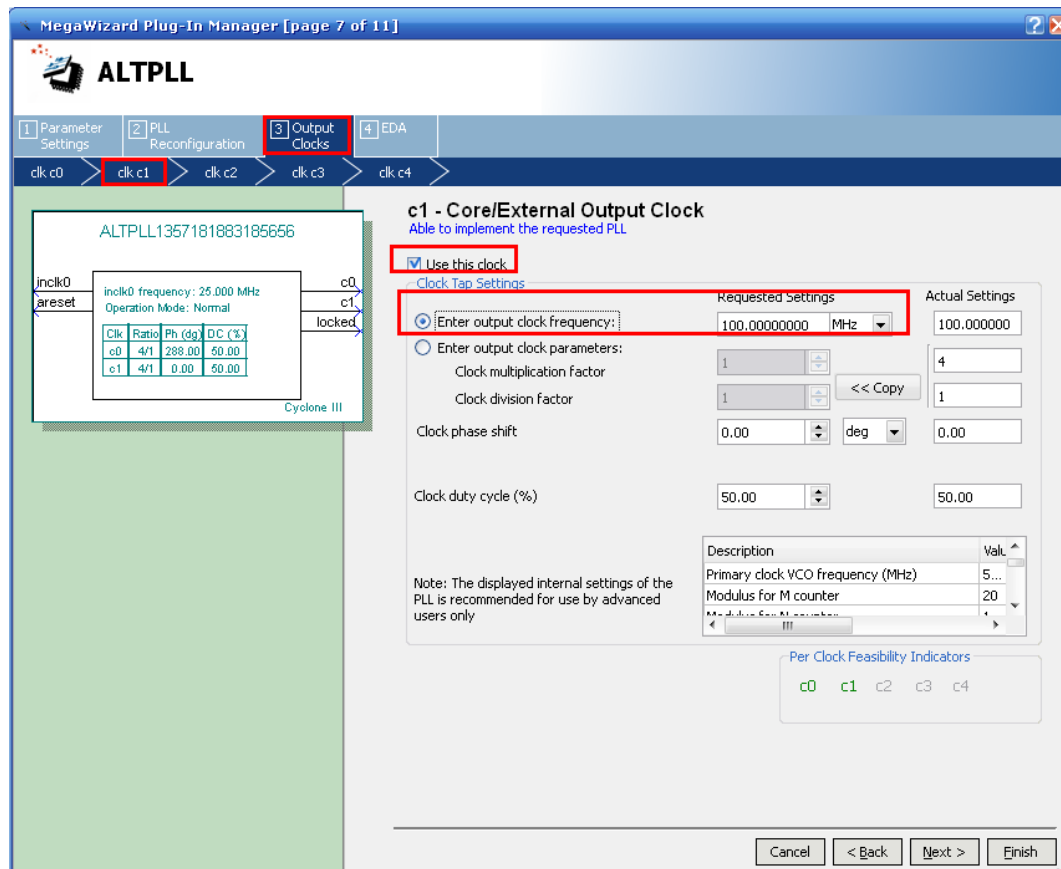
接着添加 PLL 组件（PLL-Avalon ALTPLL）。设置输入时钟。



设置 c0 输出时钟为 100MHz，相位差 8ns。

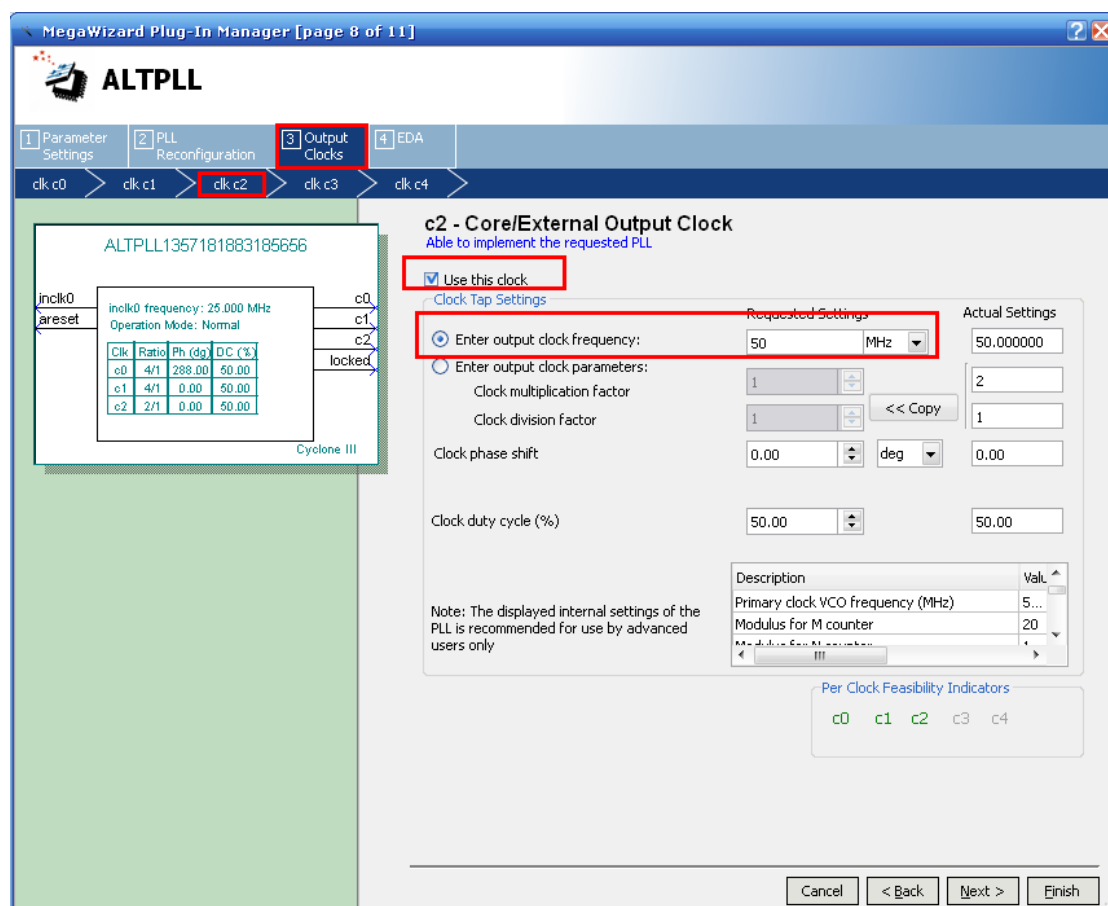


设置 c1 输出时钟为 100MHz, 相位 0ns/0deg。



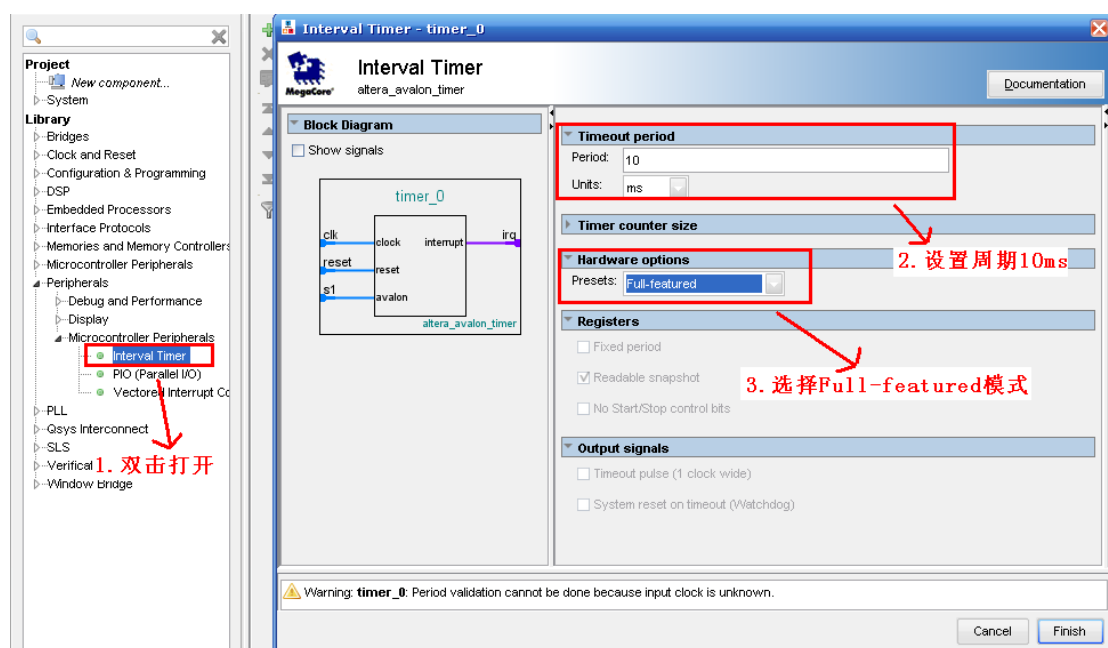


设置 c2 时钟频率 50MHz, 相位为 0ns/0deg。



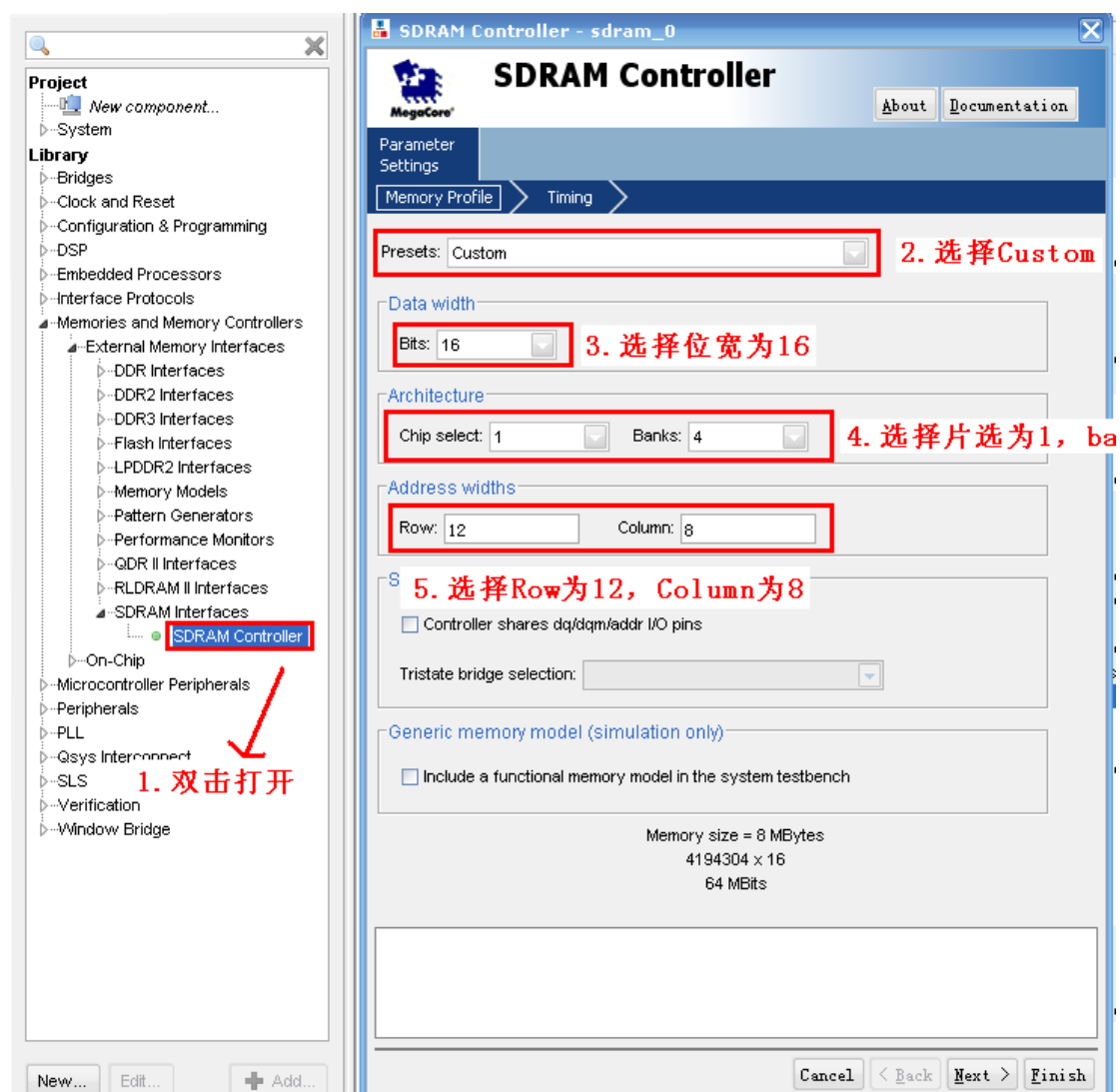
完成设置, 点击 Finish。

添加 Timer 组件 (Peripherals→Microcontroller Peripherals→Interval Timer), 设置定时周期为 10ms。

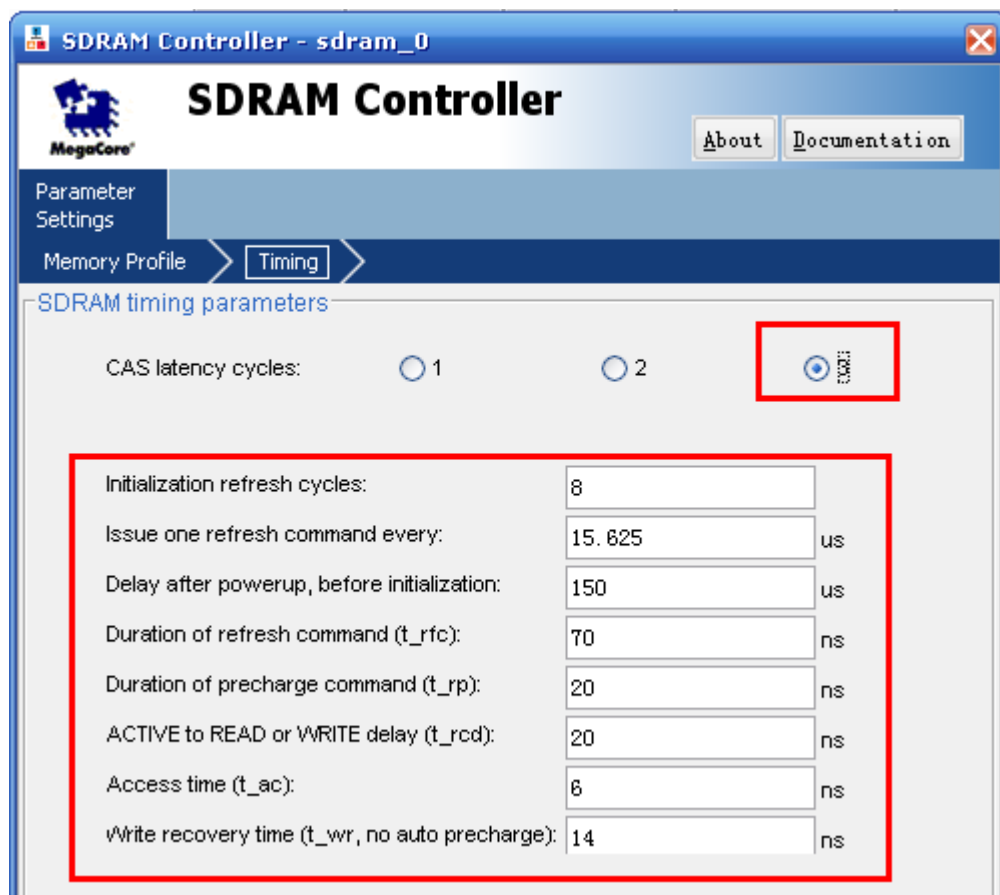




添加 SDRAM 控制器组件。Memory Profile 页面配置如图所示, 这里的配置信息主要和所使用的 SDRAM 存储位宽大小有关。

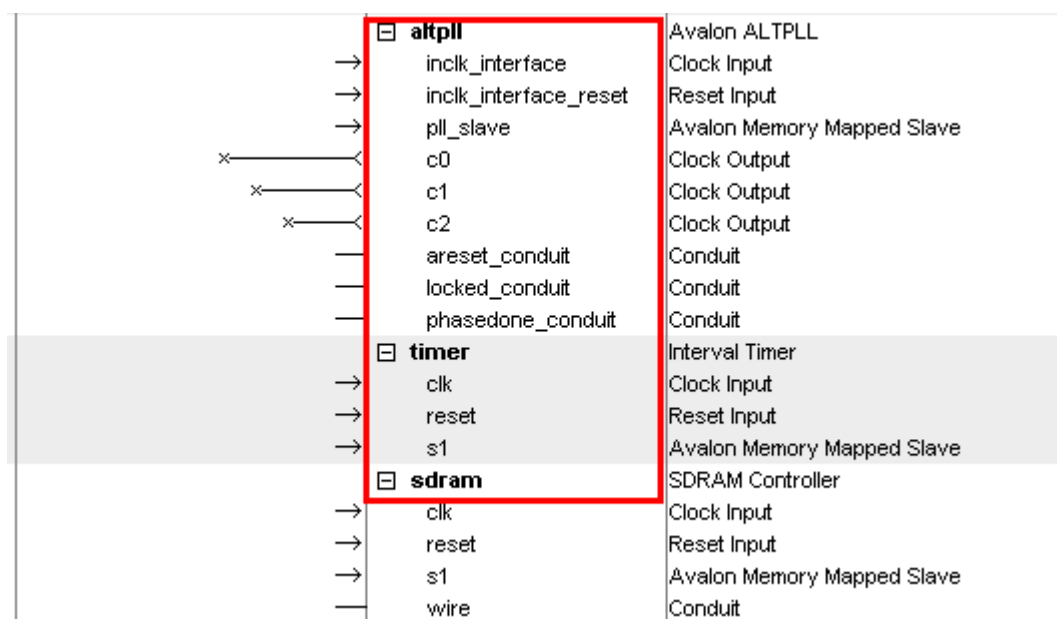


Timing 页面设置如图所示。这里的配置信息需要参考所使用的 SDRAM 的 datasheet。详细的和 SDRAM 时序相关的内容请大家参考《爱上 FPGA 开发——特权和你一起学 NIOS II》一书的第四和第五章, 那里有较详细的知识点说明。

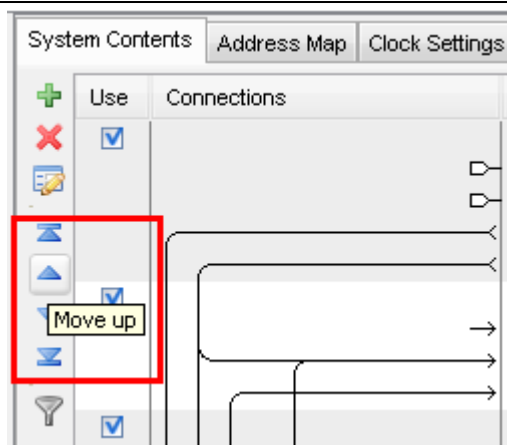


设置好后，点击 Finish 完成添加。

三个新外设都添加完毕，我们修改它们的名称如图所示。同时我们注意的这三个新外设前面和互联接口没有任何连接。

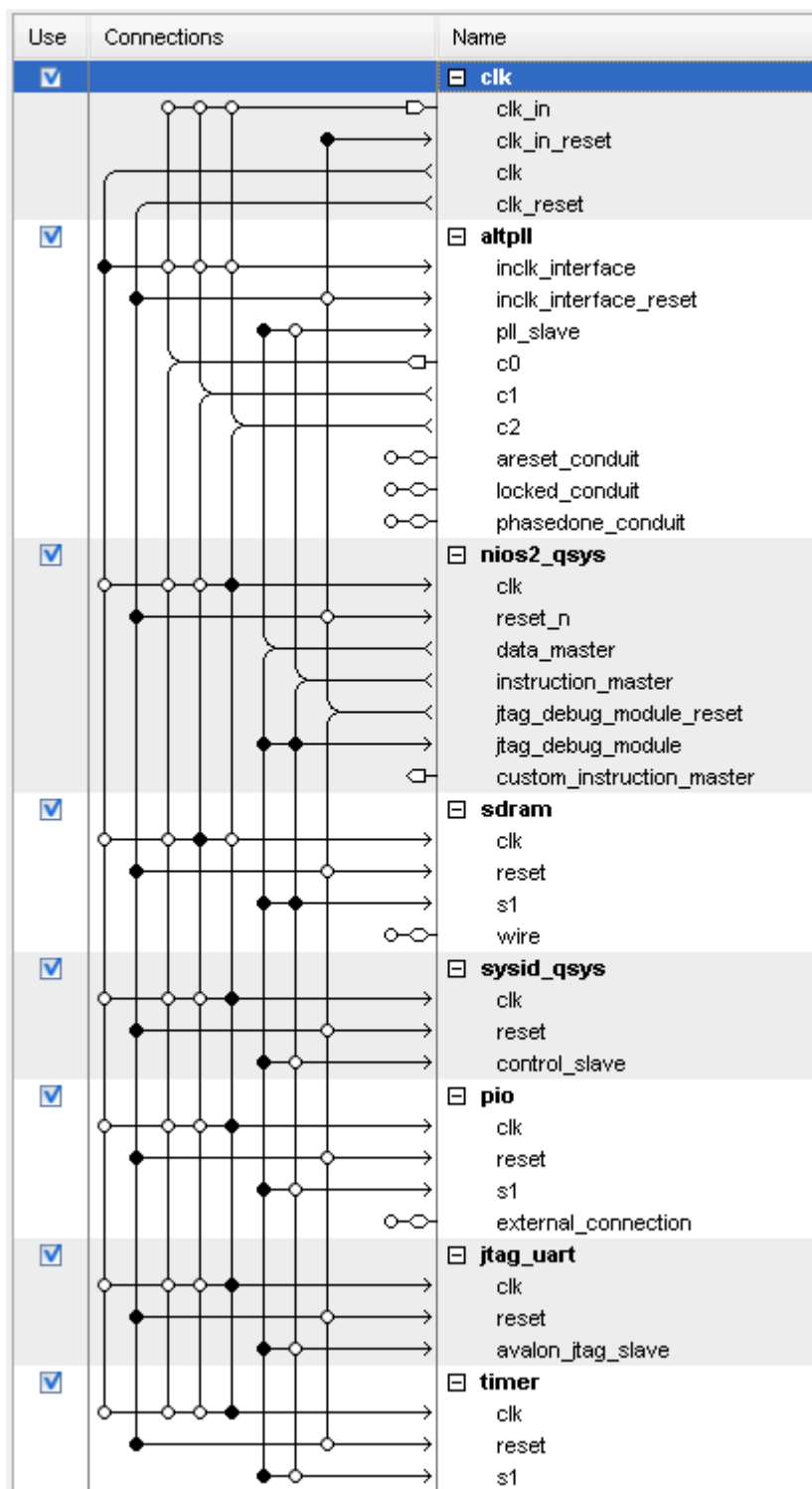


接着我们可以选择各个组件，然后使用面板左侧的几个图标对他们的上下位置进行调整。



接下来重新对整个系统的总线互联做一些修改,修改后如图所示。主要连接的原则如下。

- Clk 组件的输入时钟和复位是整个系统的基准输入的时钟和复位,从 clk 组件输出的时钟直接送到 PLL 输入,输出的复位即各个组件的复位。
- PLL 输出的 3 个时钟, c0 引出到外部供给 SDRAM, c1 供给 SDRAM 控制器模块,这两个时钟虽然都是 SDRAM 使用的,但是存在相位差。C2 供给系统的其他外设(包括 NIOS II 处理器)。
- NIOS II 处理器的 data_master 即数据总线接口连接到所有的外设,它的 instruction_master 即指令总线连接到 SDRAM 组件(也就是说代码的运行通过这条总线从 SDRAM 取指令)。
- Pll 的 c0、areset、locked、phasedone、sdram 的 wire 信号(包括之前工程里的一些信号)等都是要引出到 FPGA 顶层文件的,所以对应的在 Export 列里面单击对应信号所在行便可。



中断优先级需要设置一下，如图所示。



<input type="checkbox"/> nios2_qsys	Nios II Processor						
clk	Clock Input		Click to export	altpll_c2			
reset_n	Reset Input		Click to export	[clk]			
data_master	Avalon Memory Mapped Master		Click to export		IRQ 0	IRQ 31	
instruction_master	Avalon Memory Mapped Master		Click to export				
jtag_debug_module_reset	Reset Output		Click to export				
jtag_debug_module	Avalon Memory Mapped Slave		Click to export				
custom_instruction_master	Custom Instruction Master	nios2_qsys_0_custom_i...	Click to export		0x00010800	0x00010fff	
<input type="checkbox"/> sdram	SDRAM Controller						
clk	Clock Input		Click to export	altpll_c1			
reset	Reset Input		Click to export	[clk]			
s1	Avalon Memory Mapped Slave		Click to export		0x00000000	0x007fffff	
wire	Conduit	sdram_wire					
<input type="checkbox"/> sysid_qsys	System ID Peripheral						
clk	Clock Input		Click to export	altpll_c2			
reset	Reset Input		Click to export	[clk]			
control_slave	Avalon Memory Mapped Slave		Click to export		0x00011018	0x0001101f	
<input type="checkbox"/> pio	PIO (Parallel I/O)						
clk	Clock Input		Click to export	altpll_c2			
reset	Reset Input		Click to export	[clk]			
s1	Avalon Memory Mapped Slave		Click to export		0x00011000	0x0001100f	
external_connection	Conduit Endpoint	pio_0_external_connection					
<input type="checkbox"/> jtag_uart	JTAG UART						
clk	Clock Input		Click to export	altpll_c2			
reset	Reset Input		Click to export	[clk]			
avalon_jtag_slave	Avalon Memory Mapped Slave		Click to export		0x00011010	0x0001101f	
<input type="checkbox"/> timer	Interval Timer						
clk	Clock Input		Click to export	altpll_c2			
reset	Reset Input		Click to export	[clk]			
s1	Avalon Memory Mapped Slave		Click to export		0x00000000	0x0000001f	

接着双击再次打开 nios2_qsys 组件，修改 Reset Vector 和 Exception Vector 为 sdram.s1。

Hardware Arithmetic Operation

Hardware multiplication type: None

☐ Hardware divide

Reset Vector

Reset vector memory: sdram.s1

Reset vector offset: 0x00000000

Reset vector: 0x00800000

Exception Vector

Exception vector memory: sdram.s1

Exception vector offset: 0x00000020

Exception vector: 0x00800020

MMU and MPU

☐ Include MMU

Only include the MMU using an operating system that explicitly supports an MMU.

Fast TLB Miss Exception vector memory None

Fast TLB Miss Exception vector offset 0x00000000

Fast TLB Miss Exception vector 0x00000000

就此，Qsys 中的修改完毕，到 Generation 页面点击 Generate 重新生成系统。



5.5.3 系统例化和管脚分配

拷贝 Qsys 中的 HDL Example, 回到 Quartus II 中, 粘贴并修改 Qsys 系统的例化。我们看到相比之前, 对了一组 SDRAM 相关的接口, 我们分别将他们引出到系统的顶层接口中。代码如下。

```
module ex2(
    clk, rst_n,
    led,
    sdram_addr, sdram_data, sdram_ba, sdram_cas_n, sdram_ras_n,
    sdram_cke, sdram_cs_n, sdram_we_n, sdram_clk_c0
);

input clk;
input rst_n;
output led;

output[11:0] sdram_addr;
inout[15:0] sdram_data;
output[1:0] sdram_ba;
output sdram_cas_n;
output sdram_ras_n;
output sdram_cke;
output sdram_cs_n;
output sdram_we_n;
output sdram_clk_c0;

wire pll_locked;

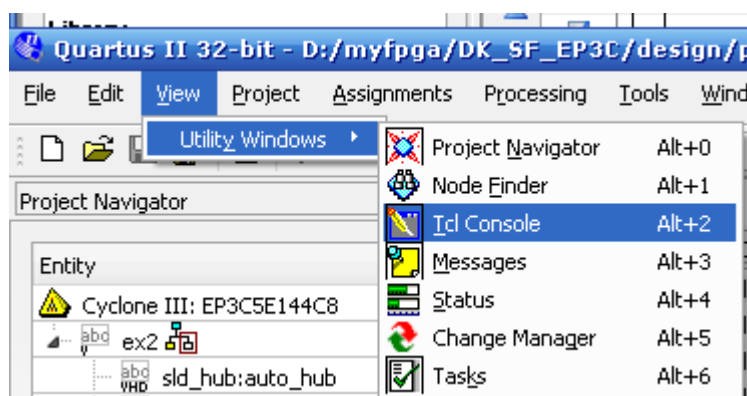
myqsys u0 (
    .clk_clk (clk),
    .pio_0_external_connection_export (led),
    .nios2_qsys_0_custom_instruction_master_readra ( ),
    .sdram_wire_addr (sdram_addr),
    .sdram_wire_ba (sdram_ba),
    .sdram_wire_cas_n (sdram_cas_n),
    .sdram_wire_cke (sdram_cke),
    .sdram_wire_cs_n (sdram_cs_n),
```



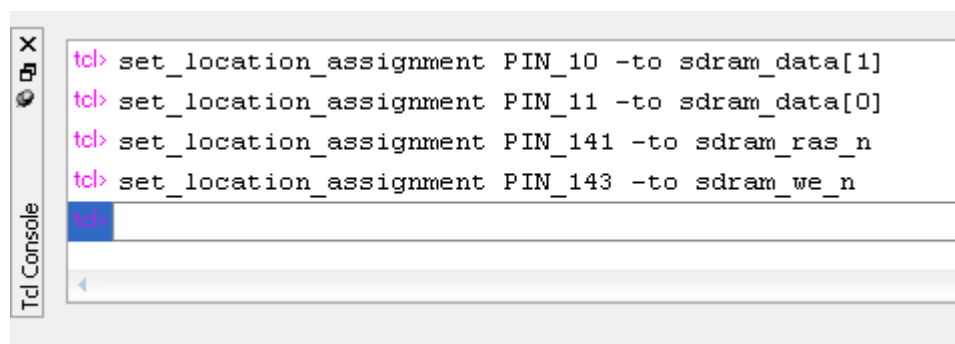
```
.sdram_wire_dq (sdram_data),  
.sdram_wire_dqm ( ),  
.sdram_wire_ras_n (sdram_ras_n),  
.sdram_wire_we_n (sdram_we_n),  
.altpll_c0_clk (sdram_clk_c0),  
.altpll_areset_conduit_export (!rst_n),  
.altpll_locked_conduit_export (pll_locked),  
.altpll_phasedone_conduit_export ( )  
);
```

endmodule

接着我们来分配新添加的 SDRAM 的管脚, 我们可以有另一种 tcl 脚本的方式来分配。点击菜单的 View→Utility Windows→Tcl Console。



出现了 Tcl Console 窗口, 点击 tcl>后面的空白处, 则可以输入脚本。



我们输入 SDRAM 信号各个管脚的分配脚本如下。

```
set_location_assignment PIN_46 -to sdram_addr[11]  
set_location_assignment PIN_135 -to sdram_addr[10]  
set_location_assignment PIN_49 -to sdram_addr[9]  
set_location_assignment PIN_50 -to sdram_addr[8]  
set_location_assignment PIN_51 -to sdram_addr[7]
```



```
set_location_assignment PIN_52 -to sdram_addr[6]
set_location_assignment PIN_53 -to sdram_addr[5]
set_location_assignment PIN_54 -to sdram_addr[4]
set_location_assignment PIN_128 -to sdram_addr[3]
set_location_assignment PIN_129 -to sdram_addr[2]
set_location_assignment PIN_132 -to sdram_addr[1]
set_location_assignment PIN_133 -to sdram_addr[0]
set_location_assignment PIN_136 -to sdram_ba[1]
set_location_assignment PIN_137 -to sdram_ba[0]
set_location_assignment PIN_142 -to sdram_cas_n
set_location_assignment PIN_44 -to sdram_cke
set_location_assignment PIN_43 -to sdram_clk_c0
set_location_assignment PIN_138 -to sdram_cs_n
set_location_assignment PIN_34 -to sdram_data[15]
set_location_assignment PIN_33 -to sdram_data[14]
set_location_assignment PIN_32 -to sdram_data[13]
set_location_assignment PIN_31 -to sdram_data[12]
set_location_assignment PIN_30 -to sdram_data[11]
set_location_assignment PIN_38 -to sdram_data[10]
set_location_assignment PIN_39 -to sdram_data[9]
set_location_assignment PIN_42 -to sdram_data[8]
set_location_assignment PIN_144 -to sdram_data[7]
set_location_assignment PIN_1 -to sdram_data[6]
set_location_assignment PIN_2 -to sdram_data[5]
set_location_assignment PIN_3 -to sdram_data[4]
set_location_assignment PIN_4 -to sdram_data[3]
set_location_assignment PIN_7 -to sdram_data[2]
set_location_assignment PIN_10 -to sdram_data[1]
set_location_assignment PIN_11 -to sdram_data[0]
set_location_assignment PIN_141 -to sdram_ras_n
set_location_assignment PIN_143 -to sdram_we_n
```

完成管脚分配后，我们对整个工程做一次全编译。

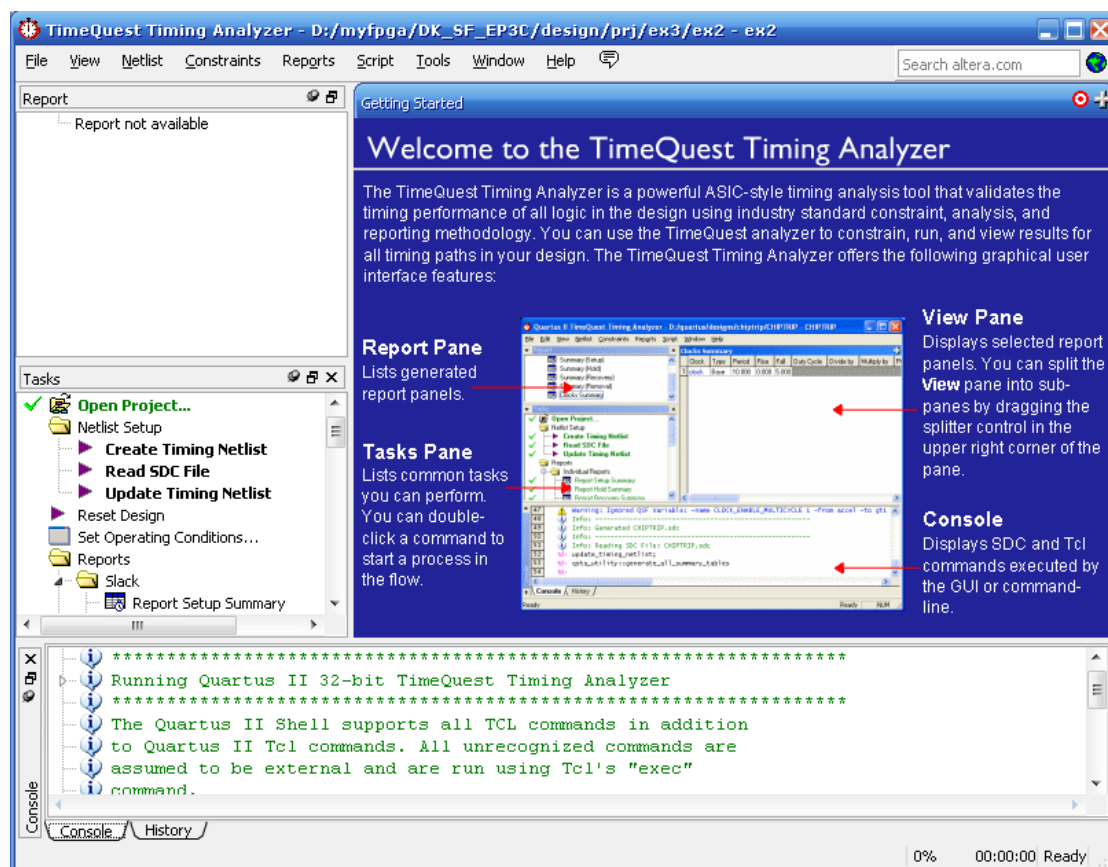
5.5.4 时序约束与工程编译

如果其他的工程你不做时序约束可以正常工作那不足为奇，毕竟频率不高。但是这个

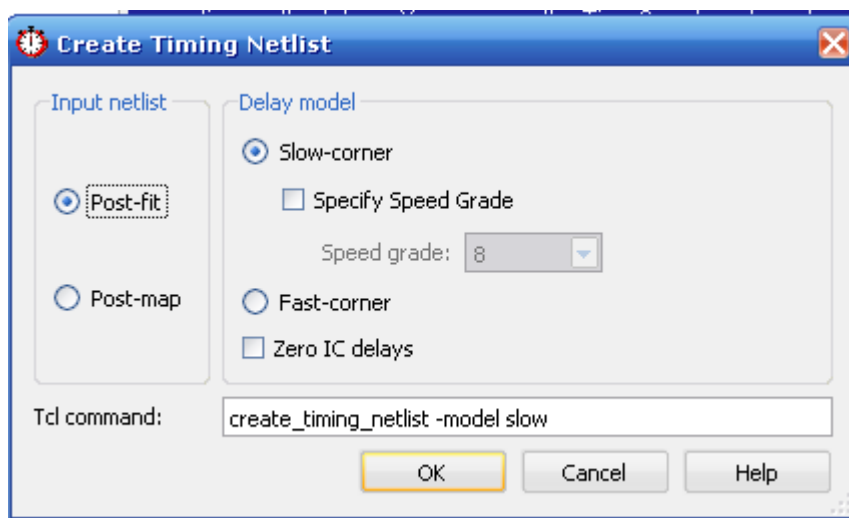


SDRAM 的工程你要是不做时序约束再能正常那就要叫做侥幸了。关于时序方面的基础知识以及 SDRAM 的时序约束理论方面的知识,建议大家去看看特权同学写的《深入浅出玩转 FPGA》和《爱上 FPGA 开发——特权和你一起学 NIOS II》中的相关章节。本教程不做详细时序分析理论的讨论,只是教会大家如何具体操作整个约束过程。

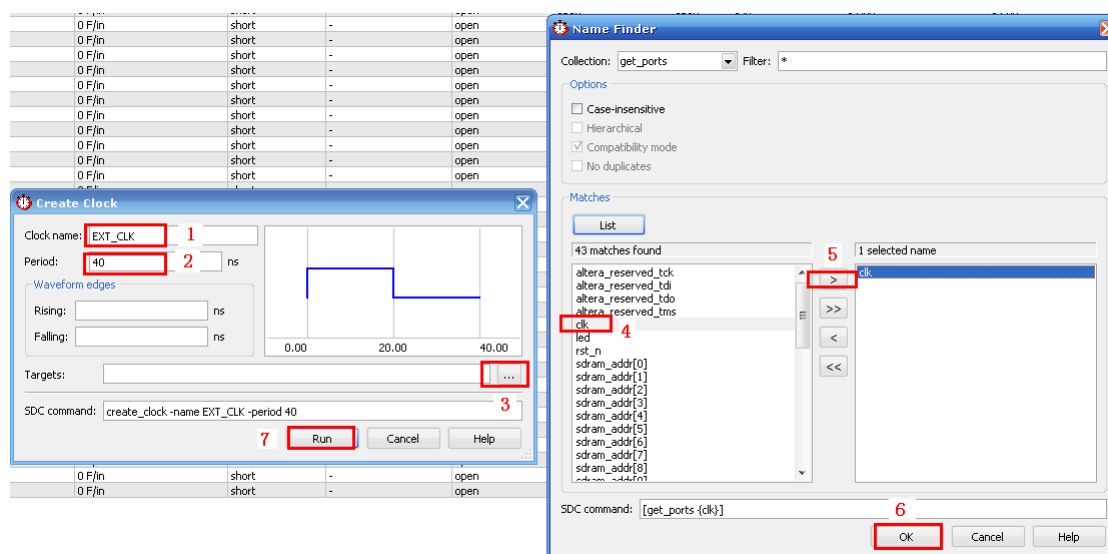
对工程做一次全编译,接着打开 TimeQuest (时序约束和分析工具),可以点击菜单栏的 Tools→TimeQuest Timing Analyzer 打开。



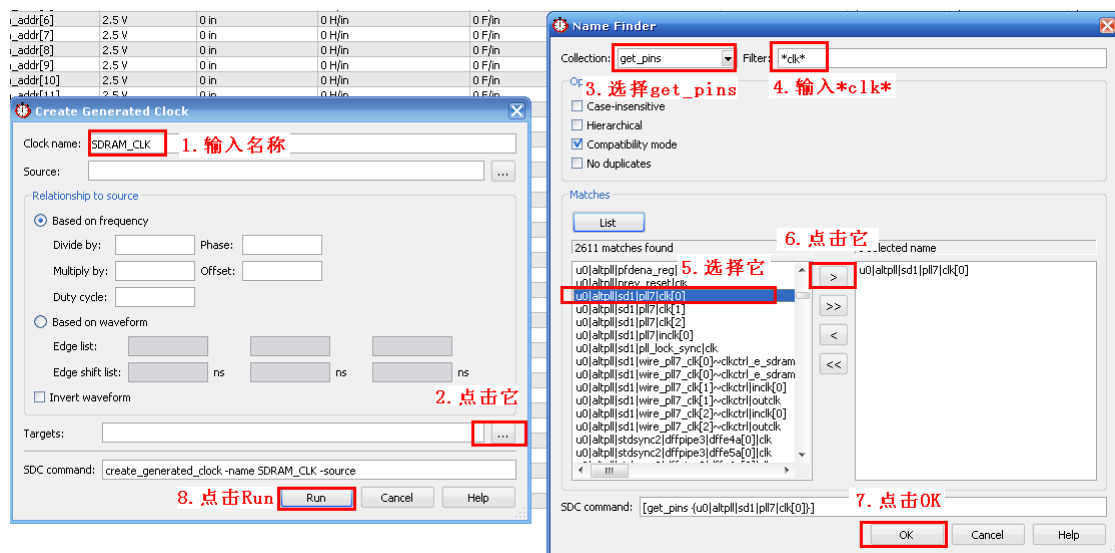
首先新建一个 SDC 文件,点击菜单栏 Netlist→Create Timing Netlist。弹出图示对话框,使用默认设置,点击 OK。



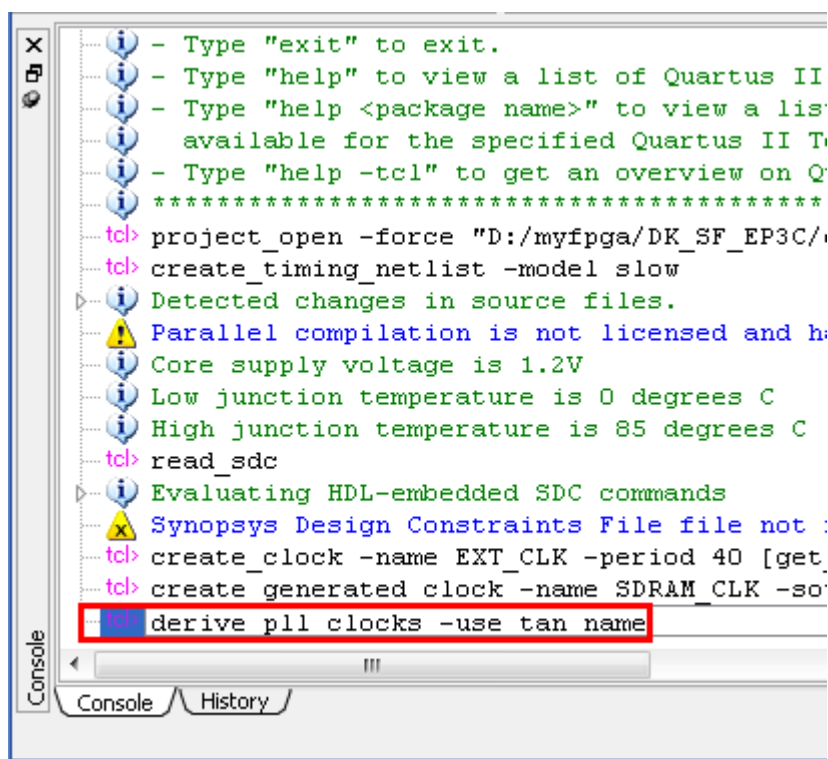
约束全局输入时钟，点击菜单栏 Constraints→Create Clock。设置时钟名称为 EXT_CLK、时钟周期 40ns，然后选择 Targets 即目标管脚为 clk。



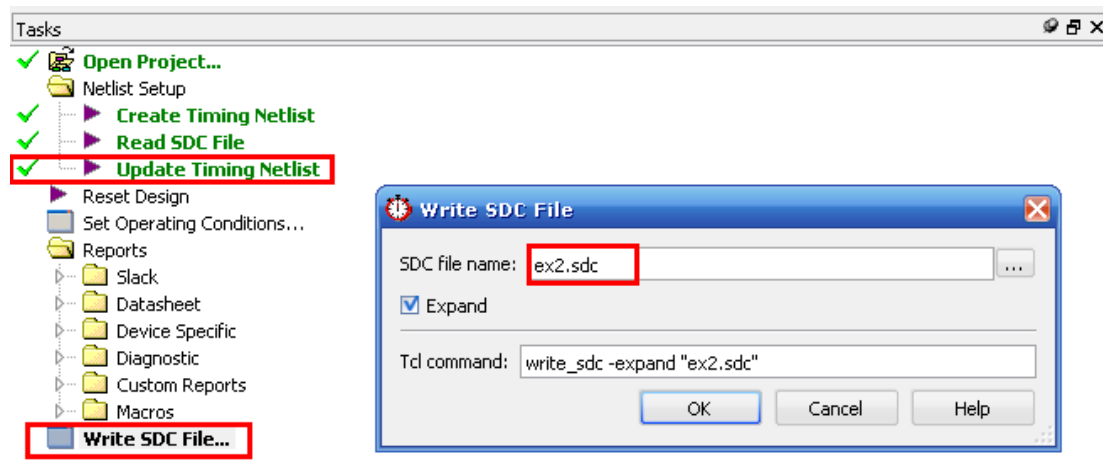
约束 SDRAM 输出时钟，点击菜单栏 Constraints→Create Generated Clock。输入名称为 SDRAM_CLK，然后点击 Targets 后的按钮，弹出的新窗口中选择 get_pins，关键词*clk*，找到时钟信号 u0/altpll|sd1|pll7|clk[0]。



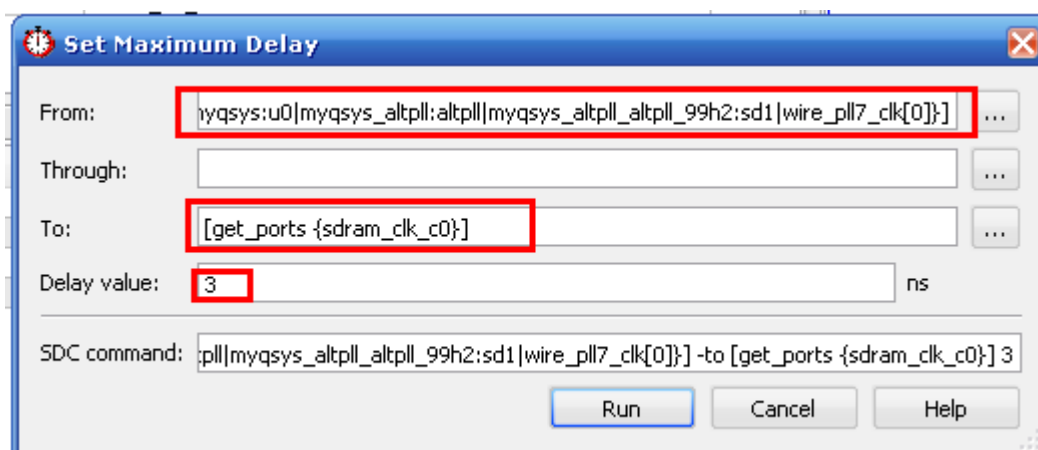
在 Console 中输入 `tcl>derive_pll_clocks -use_tan_name`。让 TimeQuest 自动去约束所有的 PLL 输出时钟。



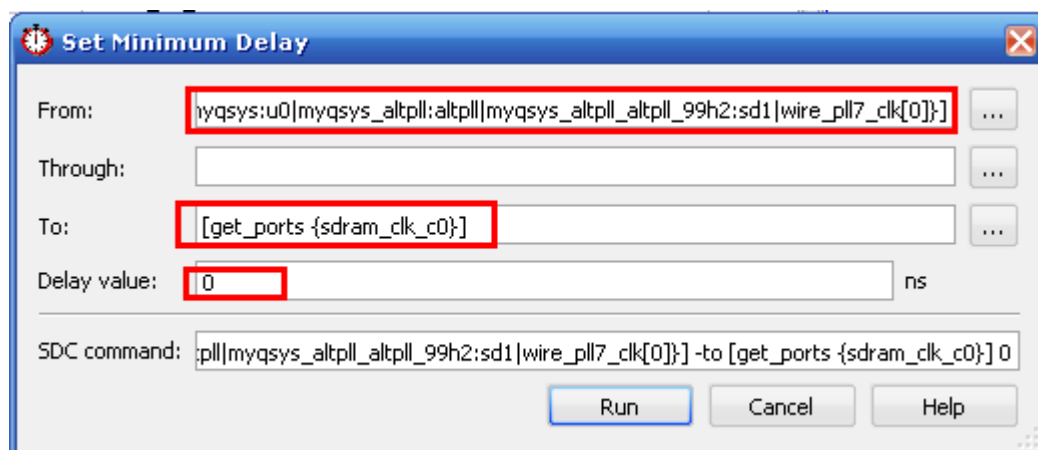
接下来我们分别点击 Tasks 面板的 Update Timing Netlist 和 Write SDC File, 然后在弹出的对话框中输入 ex2.sdc 文件, 点击 OK 完成 sdc 文件的创建, 刚才的几个脚本也都写入 sdc 文件中。



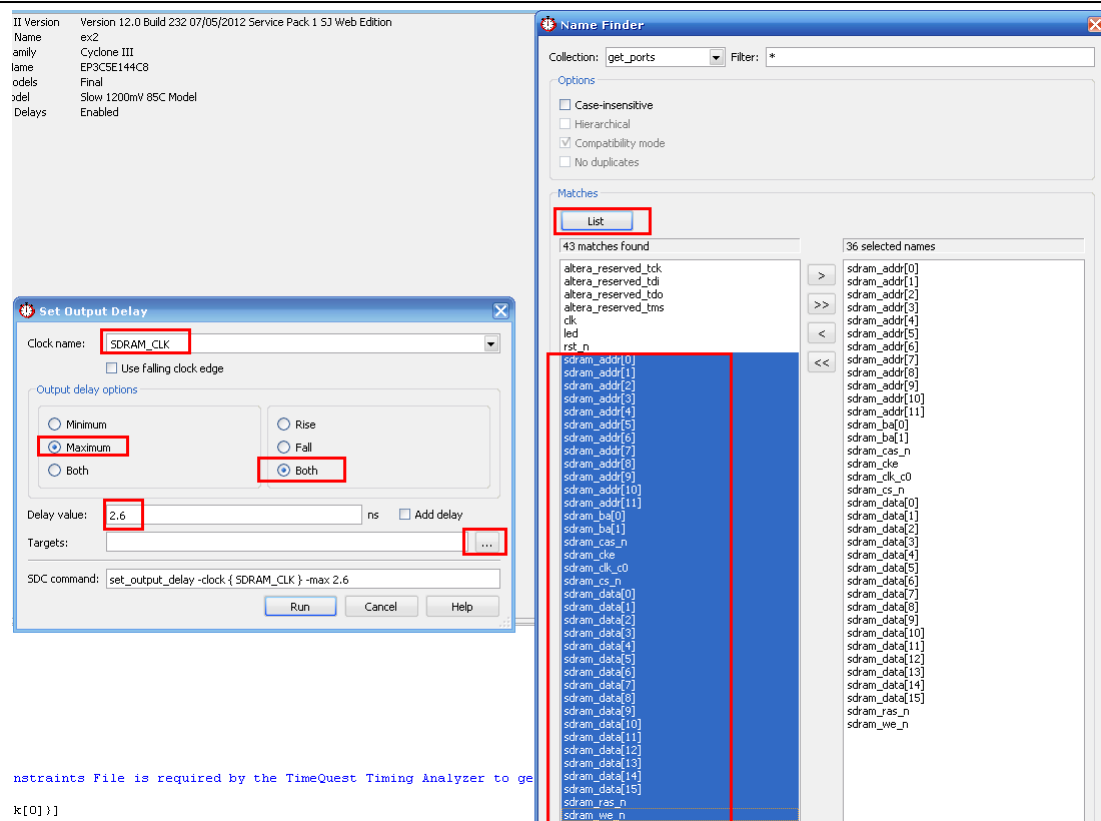
下面需要对 PLL 输出的 c0 时钟到达 FPGA 的 pin 上的延时做约束, 点击菜单栏 Constraints→Set Maximum Delay, 做如下约束。



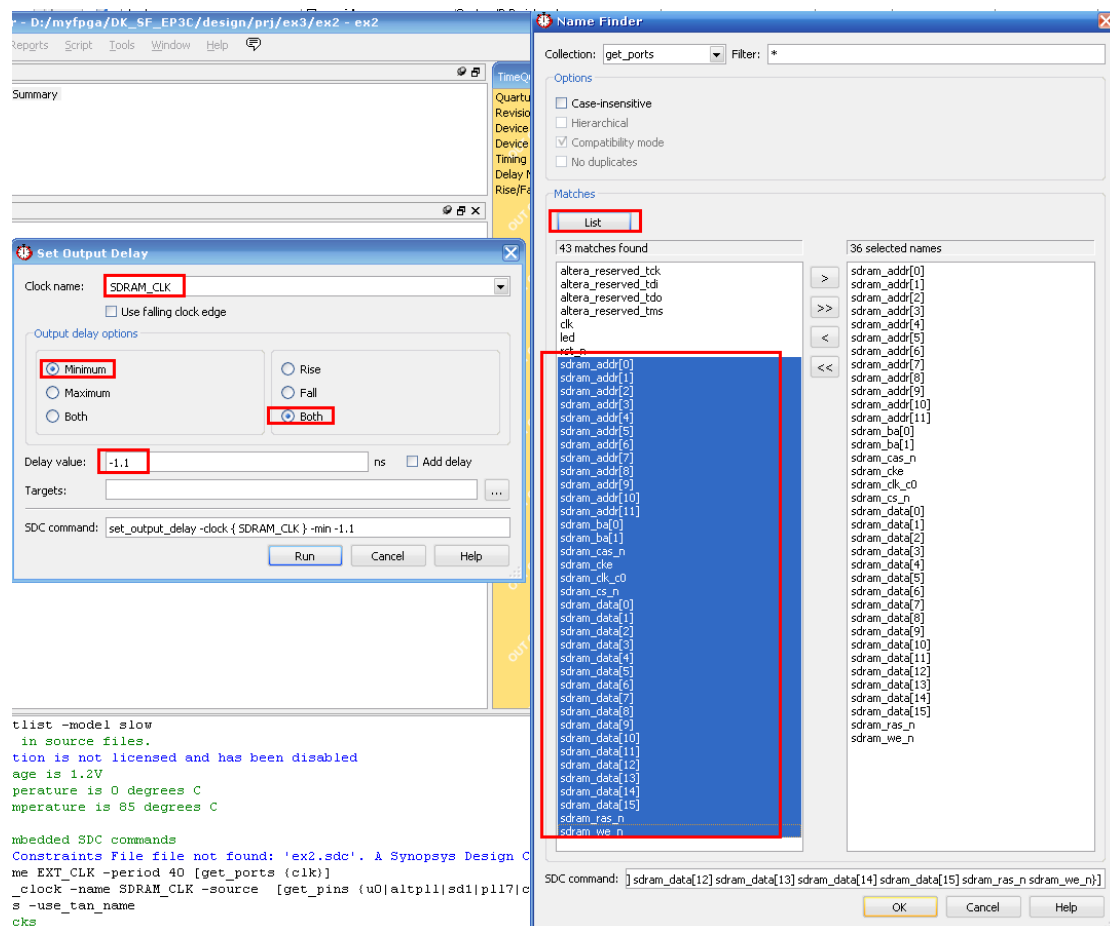
点击菜单栏 Constraints→Set Minimum Delay, 做如下约束。



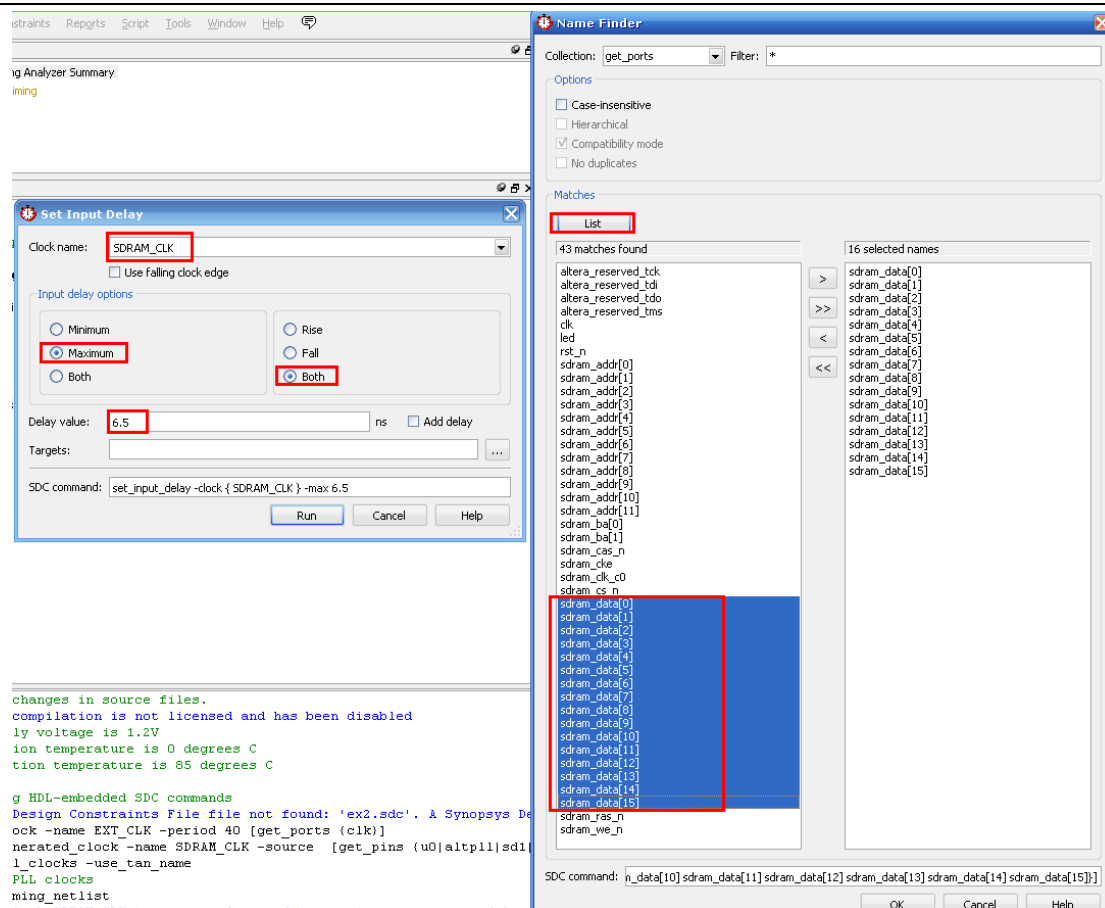
SDRAM 管脚的 input delay 和 output delay 也需要约束。点击菜单栏 Constraints→Set Output Delay。输入 Clock name 为 SDRAM_CLK, delay options 里选择 Maximum 和 Both, Delay value 为 2.6ns, 然后点击 Targets 后面的按钮, 添加所有 SDRAM 的信号。设置完点击 Run。



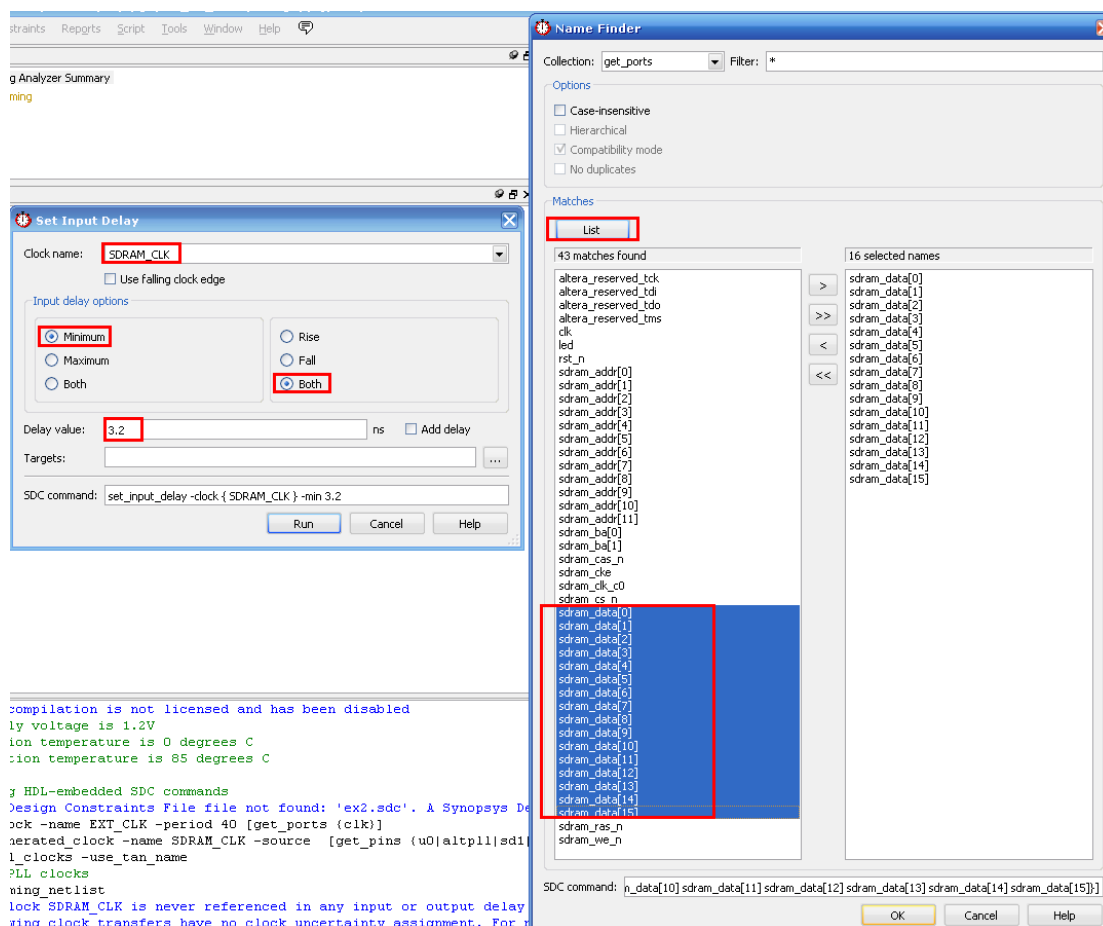
点击菜单栏 Constrains→Set Output Delay。输入 Clock name 为 SDRAM_CLK, delay options 里选择 Minimum 和 Both, Delay value 为-1.1ns, 然后点击 Targets 后面的按钮, 添加所有 SDRAM 的信号。设置完点击 Run。



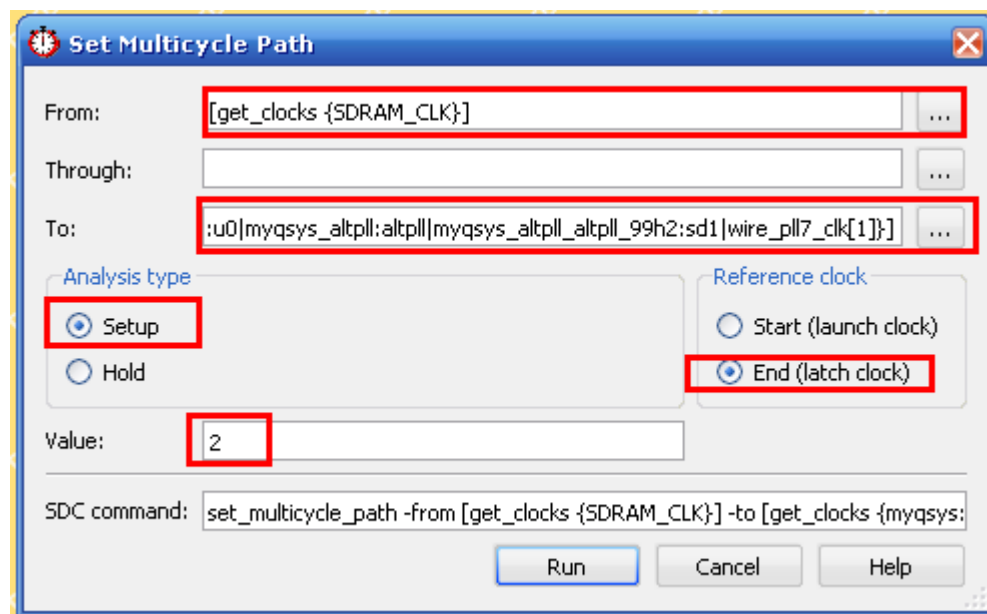
点击菜单栏 Constrains→Set Input Delay。输入 Clock name 为 SDRAM_CLK，delay options 里选择 Maximum 和 Both，Delay value 为 6.5ns，然后点击 Targets 后面的按钮，添加所有 SDRAM 的数据总线信号（sdr*_data[0]到 sdr*_data[15]）。设置完点击 Run。



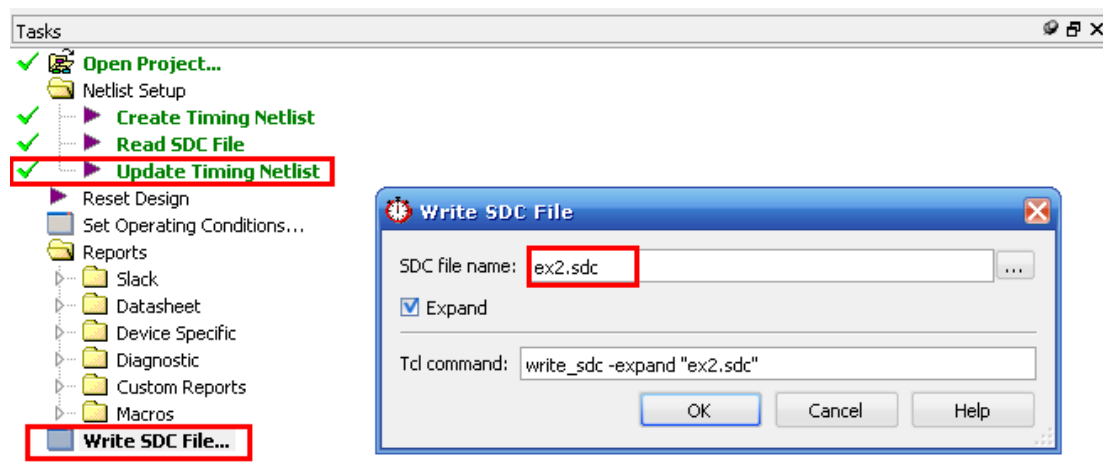
点击菜单栏 Constrains→Set Input Delay。输入 Clock name 为 SDRAM_CLK，delay options 里选择 Minimum 和 Both，Delay value 为 3.2ns，然后点击 Targets 后面的按钮，添加所有 SDRAM 的数据总线信号（sdram_data[0]到 sdram_data[15]）。设置完点击 Run。



点击菜单栏 Constrains→Set Multicycle Path，做如下选择设置。



为了将新做的约束添加到 sdc 文件中，我需要做和前面同样的操作：分别点击 Tasks 面板的 Update Timing Netlist 和 Write SDC File，然后在弹出的对话框中输入 ex2.sdc 文件，点击 OK 完成 sdc 文件的创建，刚才的几个脚本也都写入 sdc 文件中。

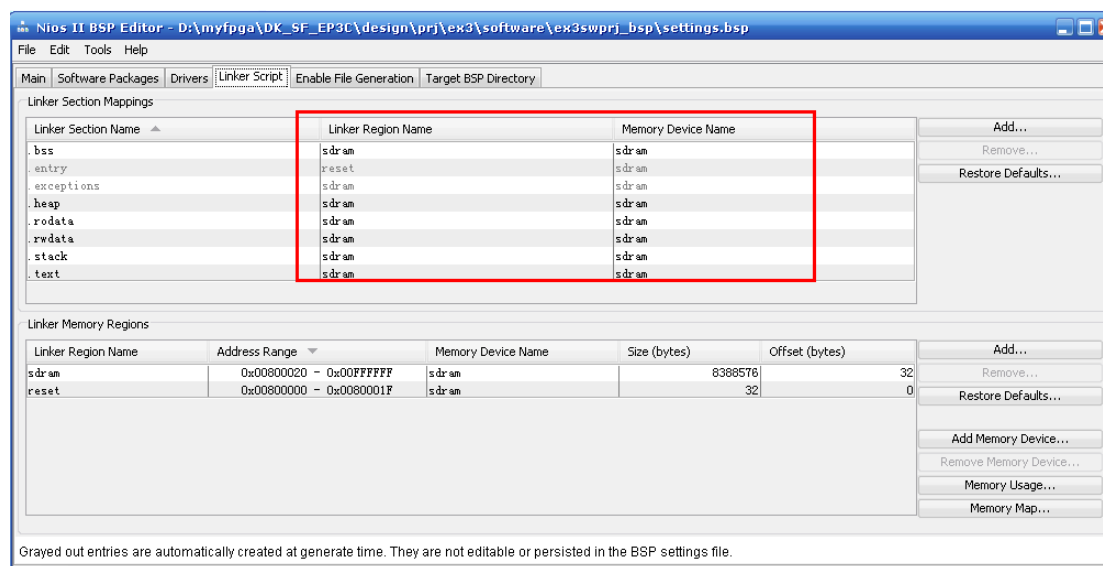


完成这些时序约束设置后, 我们重新编译工程。

5.5.5 软件工程

参照前面的操作, 我们在 EDS 中新建一个名为 ex3swprj 的软件应用工程和相应的 BSP 工程。然后我们进入应用工程的 BSP Editor 中, 无须在 Main 页面做任何设置, 因为我们不需要裁剪代码, 64Mbit 的 SDRAM 远远比之前的 20KB 片内 RAM 大, 一会我们也可以看看不裁剪代码编译后的代码量和前面裁剪过的比较。

如果前面的几个工程实例中注意看过 BSP Editor 其他几个页面的朋友, 如果再看看现在的 Linker Script 页面, 则会发现原来的 onchip_mem 全部由 sdram 取代了。



后面大家可以自己创建一个新的软件工程, 接着新建一个 main.c 文件, 将前面两个工程的代码 copy 过来也行, 在线运行到 FPGA 中看看是否 Run 起来了。

《圣经》箴言九 11 “敬畏耶和华是智慧的开端, 认识至胜者便是聪明。”



这里移植了前面的 LED 闪烁灯的程序，编译信息如下。

```

C-Build [ex3swprj]
ex3swprj.elf obj/default/main.o -lm
nios2-elf-insert ex3swprj.elf --thread_model hal --cpu_name nios2_qsys --qsys true -
--simulation_enabled false --id 0 --sidp 0x1001048 --timestamp 1357189828 --stderr_dev
jtag_uart --stdin_dev jtag_uart --stdout_dev jtag_uart --sopc_system_name myqsys -
--quartus_project_dir "D:/myfpga/DK_SF_EP3C/design/prj/ex3" --jdi
D:/myfpga/DK_SF_EP3C/design/prj/ex3/ex2.jdi --sopcinfo
D:/myfpga/DK_SF_EP3C/design/prj/ex3/myqsys.sopcinfo
Info: (ex3swprj.elf) 15 KBytes program size (code + initialized data).
Info: 8170 KBytes free for stack + heap.
Info: Creating ex3swprj.objdump
nios2-elf-objdump --disassemble --syms --all-header --source ex3swprj.elf
>ex3swprj.objdump
[ex3swprj build complete]
    
```

代码量有 15KB 之多，记得之前的代码量不过几百个 Byte，看来前面做得代码裁剪设置还真有用！

这个实例的目的在于把 SDRAM 添加到系统中，大家可以自己跑跑前面做过的软件例程看看是否能够正常运行，后面的例程将充分发挥这个用于 SDRAM 的系统的性能，跑个 uC/OS-II 系统。

6 SF-BASE 子板开发指南

6.1 功能与原理图介绍

6.1.1 主要外设芯片及装配

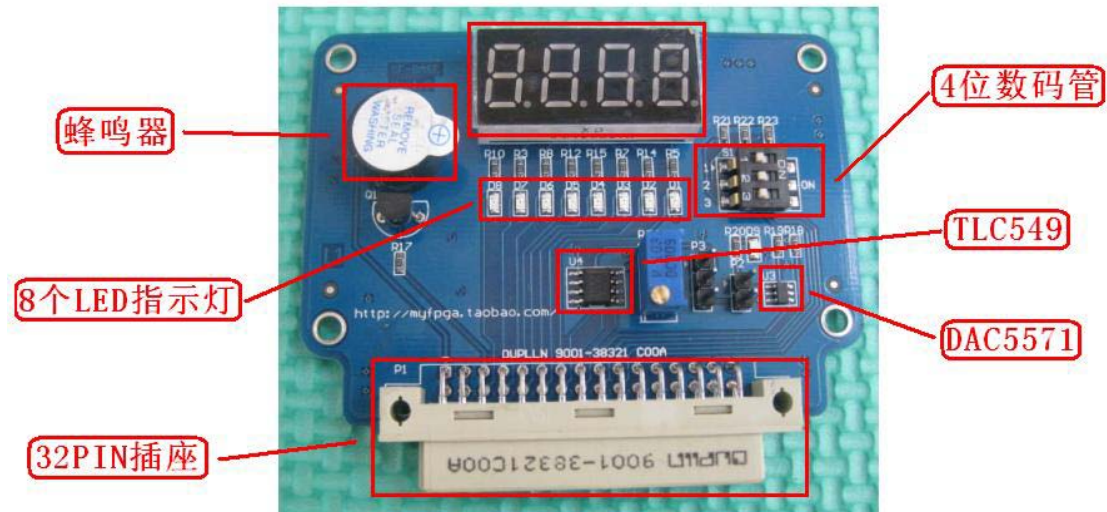
SF-BASE 子板主要外设芯片及其功能描述见下表。

外设芯片	主要功能
32PIN 的 OUPLLN 插座	用于 SF-BASE 子板上的各个外设与 SF-CY3 核心板相连。
5V 有源蜂鸣器	FPGA 的 IO 口控制该蜂鸣器发声。
8 个 LED 指示灯	用于流水灯实验。
4 位数码管	可以显示 0-F 的字符。
TLC549CD（AD 芯片）	AD 转换芯片，可以通过改变外部可变电阻器的值从而改变该芯片采集的模拟电平值。

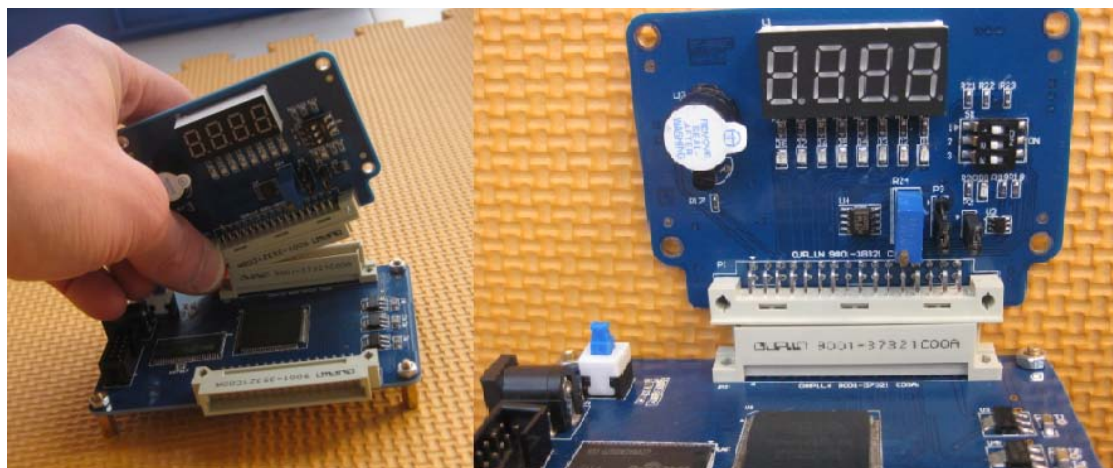


DAC5571（DA 芯片）	DA 转换芯片，可以通过外部 LED 亮度来观察不同的输出模拟电压值。
----------------	-------------------------------------

各个主要外设芯片的实物位置如图所示。

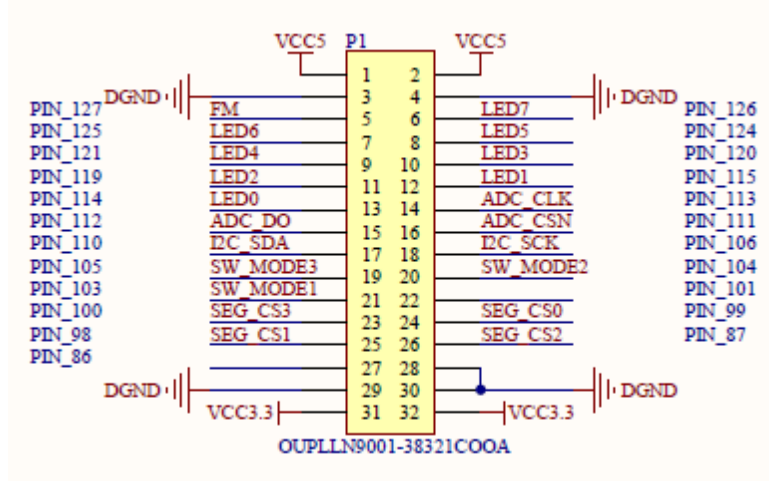


SF-BASE 板载 32PIN 插座 P1, 对应与 SF-CY3 核心板的 32PIN 插座 P2 相连。连接后, SF-CY3 核心板和 SF-BASE 子板呈直角, 如图所示。



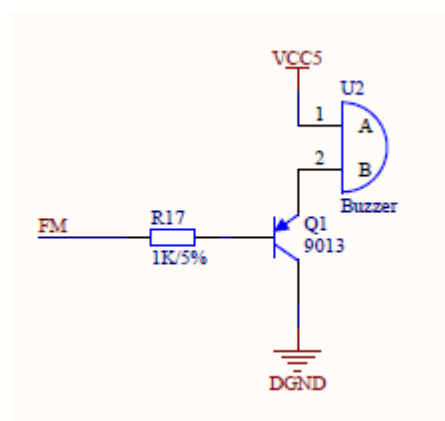
6.1.2 插座管脚定义

主要插座管脚的定义如图所示。这里蓝色字体列出的管脚号都是对应 SF-CY3 的 P2 插座到 FPGA 的 IO 管脚号。



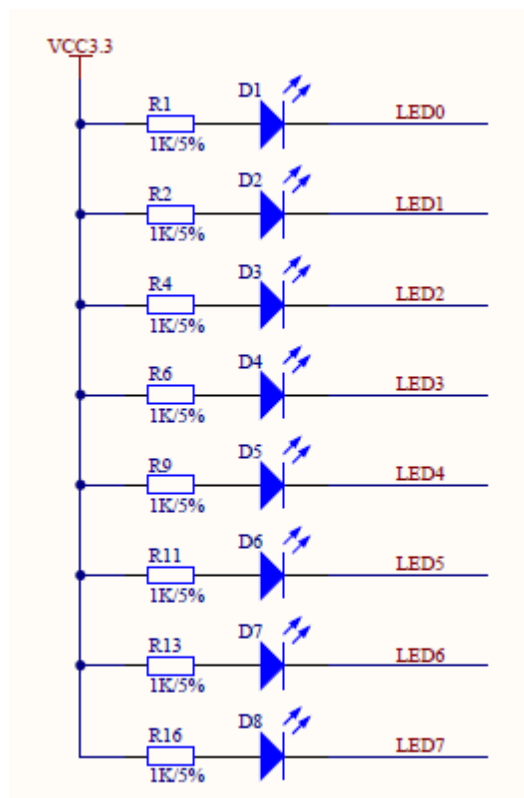
6.1.3 蜂鸣器电路

蜂鸣器电路如图所示。FM 信号由 FPGA 的 IO 口控制。当 FM 为高电平时，Q1 的 BE 导通，则 CE 导通，蜂鸣器的 5V 和 GND 形成回路，发出声音。当 FM 为低电平时，Q1 的 BE 断开，则 CE 断开，蜂鸣器的 5V 和 GND 断开，因此没有电流流过蜂鸣器，蜂鸣器便不发声。在后面的实验中，我们可以使用 PWM 信号，即以固定的时高时低的电平控制 Q1 的导通与否，然后达到蜂鸣器的时断时开，在人耳听到的便是不同频率的声响。



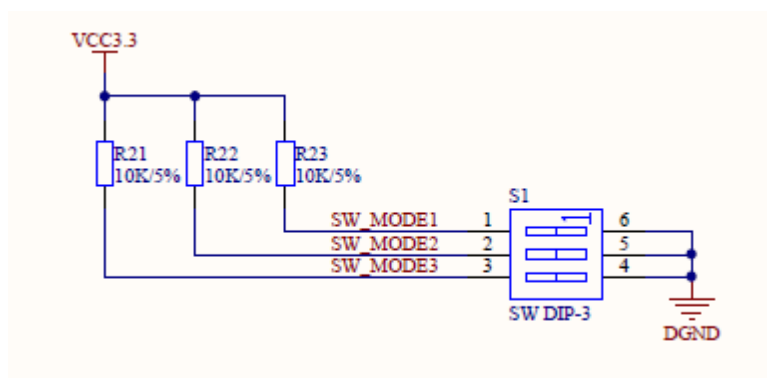
6.1.4 LED 指示灯电路

8 个 LED 指示灯的电路如图所示，他们是共 3.3V，连接 FPGA 的 IO 断若输出高电平在 LED 不亮，若输出低电平则 LED 发光。这 8 个 LED 的接口是与数码管的段选信号复用的。



6.1.5 拨码开关电路

3 位的拨码开关电路如图所示。

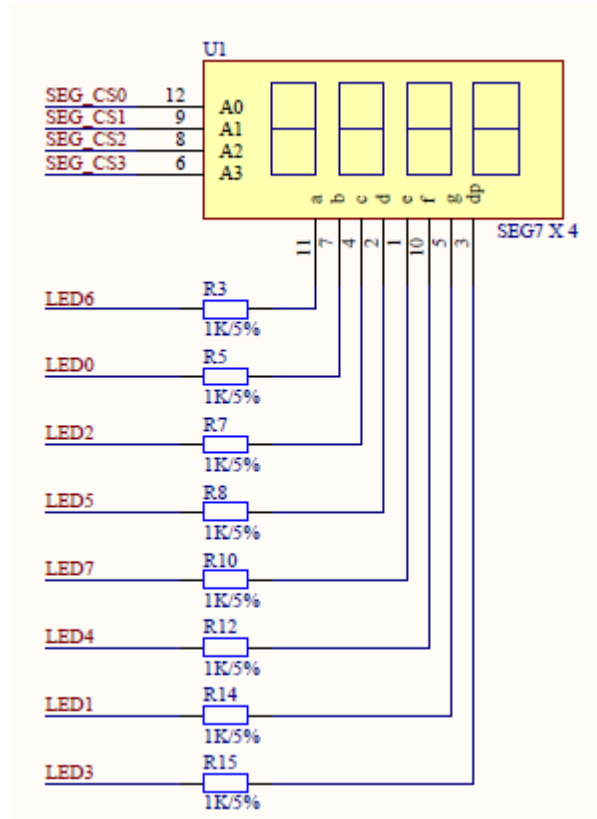


我们可以对照实物，默认 3 个拨码开关应该都是拨向左侧（即 1、2、3 标记侧），在电路图上就是 VCC3.3。就是说，默认情况下，3 个连接 FPGA 的 IO 口的信号 SW_MODE1、SW_MODE2、SW_MODE3 均为高电平。若拨码开关被拨到右侧（即标记 ON 侧），则采集到的输入就是低电平了。



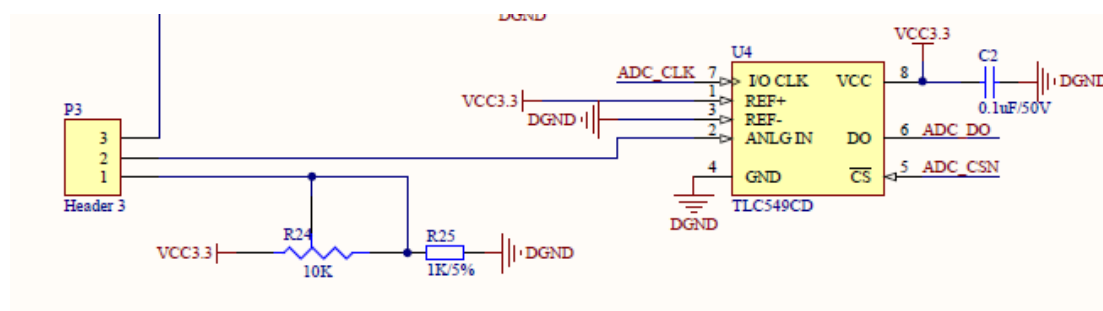
6.1.6 数码管电路

数码管电路如图所示。SEG_CS0、SEG_CS1、SEG_CS2、SEG_CS3 这 4 个信号数码管 4 位显示的片选信号，低电平有效，若 4 个片选信号都为 0，则 4 位数码管都有显示。LED0-7 则为数码管的段选信号，控制一个数码管的对应段 LED 的亮灭状态，这一组信号对于 4 位的数码管是共用的。在实际控制时，我们一般会分时点亮需要显示的各个位数码管，只要时间控制得合理，人眼是很容易被“蒙骗”的，我们很容易就能看到 4 个不同的数字显示在数码管上。



6.1.7 AD 转换电路

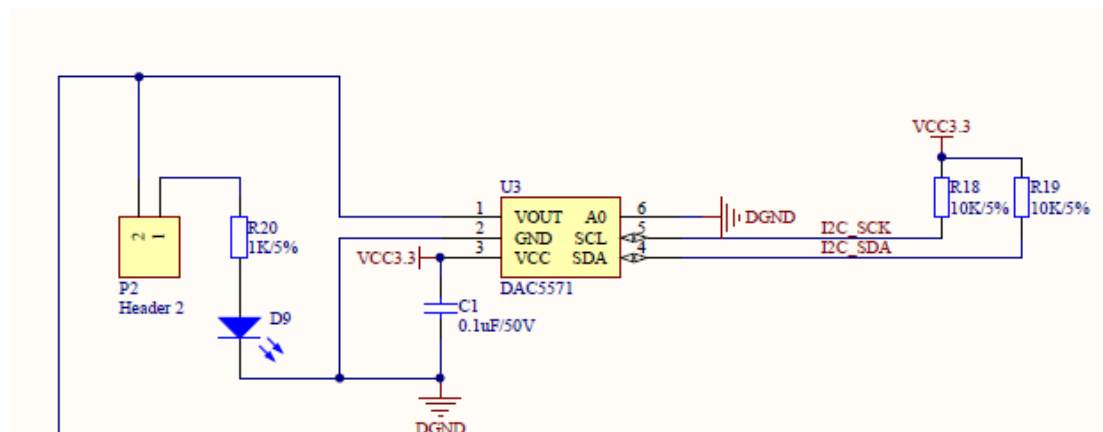
AD 芯片的电路如图所示。它通过一个单向（从 AD 芯片到 FPGA）数据传输的 SPI 接口与 FPGA 相连。FPGA 通过这组 SPI 接口读取当前模拟电压值。为了得到不同的模拟电压值，我们的板子在 AD 芯片的模拟输入端设置了一个 3.3V 的分压电阻，当跳线帽连接了 P3 的 1-2 管脚时，调节可变电阻 R24 的阻值便能改变当前 AD 采样的数据。跳线帽若连接 P3 的 2-3 管脚，则 AD 芯片的输入模拟电压来自于 DA 芯片的当前输出。





6.1.8 DA 转换电路

DA 转换电路如图所示。这个 DA 芯片通过 IIC 接口与 FPGA 连接，FPGA 通过这组 IIC 接口输出数据，相应 DA 芯片的 VOUT 输出模拟电压值。若跳线帽连接 P2 的 1-2 管脚，则不同的模拟电压值可以看到在 D9 指示灯上呈现不同的亮度。

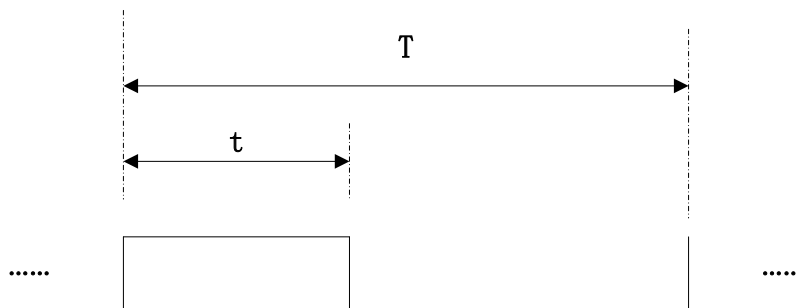


6.2 逻辑（Verilog）实例 3——PWM 驱动蜂鸣器

6.2.1 实验原理

蜂鸣器是一种最简单的发声元器件，它的应用也非常广泛，大都作为一个简单的报警或发声提示装置。比如我们家里的电脑在刚开启时，通常主板上会发出一声较短的尖锐的“滴……”的鸣叫声，提示用户主板自检通过，可以正常进行后面的启动；而如果是 1 长 1 短或 1 长 2 短的鸣叫声，则表示可能发生了电脑内存或显卡故障；当然还可以有其他不同的鸣叫声提示其他的故障，总而言之，可别小看了这颗区区几毛钱的小家伙，关键时刻还挺有用的。可以毫不夸张的说，蜂鸣器也算是一种人机交互的手段。

PWM（Pulse Width Modulation），即脉冲宽度调制，如图所示，PWM 的输出只有高电平 1 和低电平 0。PWM 不停的重复输出周期为 T ，其中高电平 1 时间为 t 的脉冲， t/T 是它的占空比， $1/T$ 是它的频率。



$$\text{占空比} = t/T$$

$$\text{频率} = 1/T$$

基于蜂鸣器受控于 GPIO 输出 1 就发声、0 则不发声的原理，我们给 GPIO 口一个占空比为 50% 的 PWM 的信号，如果它的频率高则发声就显得相对尖锐急促一些，如果它的发声频率低则发声就显得低沉平缓一些。

在我们给出的实例代码中，我们期望产生一个输出频率为 25Hz (20ms)、占空比为 50% 的 PWM 信号去驱动蜂鸣器的发声。因此，我们使用系统时钟 25MHz (20ns) 进行计数，每计数 1000000 次这个计数器就重新清零。因为这个计数器是 2 级制的，要能够表达 0-999999 的任意一个计数值，那么这个 2 进制计数器至少必须是 20 位的。此外，为了得到输出的 PWM 占空比为 50%，那么我们只要判断计数值小于最大计数值的一半即 500000 时，输出高电平 1，反之输出低电平 0。

6.2.2 Verilog 参考代码

```
module ex4(  
    clk, rst_n,  
    fm  
);  
input clk;  
input rst_n;  
output reg fm;  
  
//-----  
reg[19:0] cnt;
```



```
always @ (posedge clk or negedge rst_n)
    if(!rst_n) cnt <= 20'd0;
    else if(cnt < 20'd999_999) cnt <= cnt+1'b1;
    else cnt <= 20'd0;

//-----
always @ (posedge clk or negedge rst_n)
    if(!rst_n) fm <= 1'b0;
    else if(cnt < 20'd500_000) fm <= 1'b1;
    else fm <= 1'b0;

endmodule
```

6.2.3 仿真验证

这个实验我们建议大家做功能仿真，体验一下分频的效果，具体的仿真方法和步骤请大家参考上一章的实例。本实验的测试脚本如下：

```
`timescale 1 ns/ 1 ps
module ex4_vlg_tst();

reg clk;
reg rst_n;
wire fm;

ex4 il (
    .clk(clk),
    .fm(fm),
    .rst_n(rst_n)
);

initial begin
    rst_n = 0;
    clk = 0;
    #1000;
    @(posedge clk);
    rst_n = 1;
```



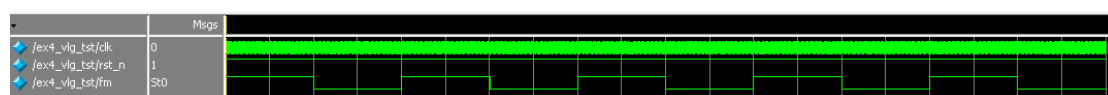
```
#200_000_000;
$stop;
end

always #20 clk = ~clk;

always @(posedge fm)
    if(fm) $display("fm posedge time is: %t\n", $time);

endmodule
```

仿真测试的波形如图所示,在仿真运行的 100ms 时间里,大约产生了 5 个 fm 分频周期。



如图所示,在仿真软件的打印窗口输出了我们监控到的 fm 信号上升沿时间。我们可以看到,任意两个相邻上升沿的时间间隔均为 40,000,000,000ps,即 40ms,和我们的功能目标完全吻合。

```
Transcript
# vsim -t lps -L altera_ver -L lpm_ver -L sgate_v
# vsim -L altera_ver -L lpm_ver -L sgate_ver -L a
# Loading work.ex4_vlg_tst
# Loading work.ex4
#
# add wave *
# view structure
# .main_pane.structure.interior.cs.body.struct
# view signals
# .main_pane.objects.interior.cs.body.tree
# run -all
# fm posedge time is:          1060000
#
# fm posedge time is:          40001060000
#
# fm posedge time is:          80001060000
#
# fm posedge time is:          120001060000
#
# fm posedge time is:          160001060000
#
```

6.2.4 工程实践

- (1) 新建工程,命名为 ex4,把这个工程放在专门的文件夹下,其他设置参考前面



章节。

- (2) 新建 Verilog 源文件, 命名为 ex4, 输入前面给出的设计代码。使用 ModelSim-Altera 对设计代码进行仿真验证。
- (3) 综合编译后进行管脚分配, 本例程的管脚分配如下所示。

Node Name	Direction	Location	I/O Bank
clk	Input	PIN_22	1
fm	Output	PIN_127	7
rst_n	Input	PIN_91	6

- (4) 参考前面章节, 打开 TimeQuest, 我们新建一个 SDC 文件, 然后对时钟 clk 做约束, 约束脚本如下:

```
create_clock -name {SYSCLK} -period 40.000 -waveform { 0.000 20.000 } [get_ports {clk}]
```

- (5) 接下来, 我们对工程进行全编译, 不仅要让刚刚添加的时钟约束生效, 也要生成可以下载到 FPGA 芯片中的配置文件。

有些朋友肯定还关心刚刚设置好的 25MHz 时钟频率是否达到了目标, 打开 TimeQuest 可以查看到所有相关的报告, 但是我们走个捷径, 大家只要展开编译完自动生成并打开的 Compilation Report 便可, 如图所示, 在 TimeQuest Timing Analyzer→Fmax Summary 里, 我们可以看到给出的 Fmax 达到了 160.62MHz, 超过了我们的 25MHz, 说明时钟频率已经达到目标。



	Fmax	Restricted Fmax	Clock Name	Note
1	160.62 MHz	160.62 MHz	SYSCLK	

(6) 最后, 我们便是要将配置文件下载到实验板中, 听听蜂鸣器的发声效果如何。

这个操作和前面章节的步骤基本一致, 这里不再赘述。

6.3 逻辑 (Verilog) 实例 4——流水灯

6.3.1 实验原理

流水灯的控制对象 LED 指示灯很简单, 流水原理很简单。8 个 LED 指示灯, 我们依次给他们赋值, 每次只有一个 LED 点亮, 每次点亮某个 LED 一定的时间 (固定延时)。8 个 LED 依次被点亮一次, 如此循环便成就了流水灯的效果。

在后面的代码中, 我们设计一个 20bit 的计数器, 在系统输入时钟 25MHz 频率下不断计数, 一个计数周期时 $1024 \times 1024 \times 40\text{ns}$ 的时间 (是多少大家自己算去)。在计数到最大的数 20'hffff 时, 我们改变一次别点亮的 LED, 即使用一个含有一个 0 的 8bit 的 2 进制数不断的循环赋值给 8 个 LED 输出管脚。



6.3.2 Verilog 参考代码

```
module ex5(
    clk, rst_n,
    led
);
input clk;
input rst_n;
output reg[7:0] led;

//-----
reg[19:0] cnt;

always @ (posedge clk or negedge rst_n)
    if(!rst_n) cnt <= 20'd0;
    else cnt <= cnt+1'b1;

//-----
always @ (posedge clk or negedge rst_n)
    if(!rst_n) led <= 8'b1111_1110;
    else if(cnt == 20'hffff) led <= {led[6:0], led[7]};
    else ;

endmodule
```

6.3.3 仿真验证

```
`timescale 1 ns/ 1 ps
module ex5_vlg_tst();

reg clk;
reg rst_n;
wire[7:0] led;
```




```
ex5 il (
    .clk(clk),
    .led(led),
    .rst_n(rst_n)
);

initial begin
    rst_n = 0;
    clk = 0;
    #1000;
    @(posedge clk);
    rst_n = 1;
    #1000_000_000;
    $stop;
end

always #20 clk = ~clk;

always @(led)
    $display("current led value : %x,time :%t\n",led,$time);

endmodule
```

这个仿真生成了时钟和复位信号，并且监视 8 位 led 指示灯信号的输出，如果 led 任意一个位发生变化，则打印当前值并显示时间。如图所示，我们看到 led 每次变化的时间都是在 40ms 多，并且从最低位为 0 依次变化到最高位为 0，如此反复。













```

Transcript
# view signals
# .main_pane.objects.interior.cs.body.tree
# run -all
# current led value : fe,time :          0
#
# current led value : fd,time :        41944060000
#
# current led value : fb,time :        83887100000
#
# current led value : f7,time :       125830140000
#
# current led value : ef,time :       167773180000
#
# current led value : df,time :       209716220000
#
# current led value : bf,time :       251659260000
#
# current led value : 7f,time :       293602300000
#
# current led value : fe,time :       335545340000
#
# current led value : fd,time :       377488380000
#
# current led value : fb,time :       419431420000
#
    
```

6.3.4 工程实践

- (1) 新建工程，命名为 ex5，把这个工程放在专门的文件夹下，其他设置参考前面章节。
- (2) 新建 Verilog 源文件，命名为 ex5，输入前面给出的设计代码。使用 ModelSim-Altera 对设计代码进行仿真验证。
- (3) 综合编译后进行管脚分配，本例程的管脚分配如下所示。

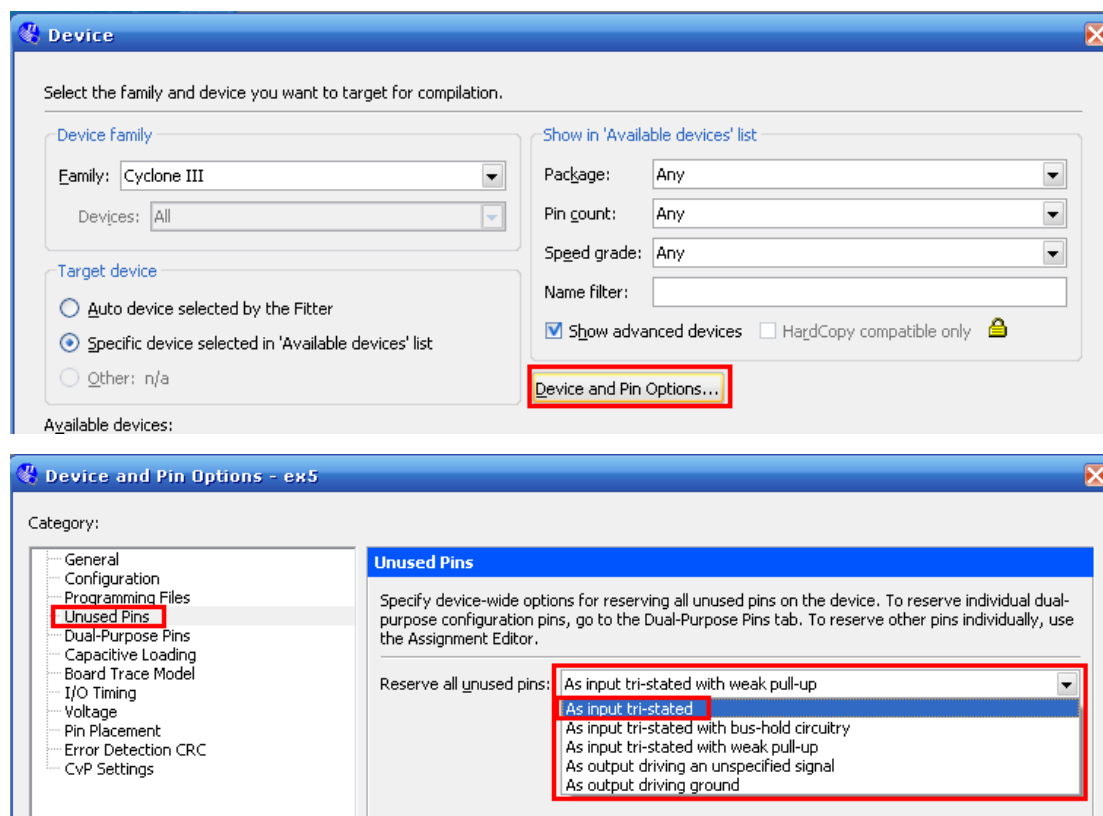
Node Name	Direction	Location	I/O Bank
 clk	Input	PIN_22	1
 led[7]	Output	PIN_126	7
 led[6]	Output	PIN_125	7
 led[5]	Output	PIN_124	7
 led[4]	Output	PIN_121	7
 led[3]	Output	PIN_120	7
 led[2]	Output	PIN_119	7
 led[1]	Output	PIN_115	7
 led[0]	Output	PIN_114	7
 rst_n	Input	PIN_91	6

- (4) 参考前面章节，打开 TimeQuest，我们新建一个 SDC 文件，然后对时钟 clk 做约束，约束脚本如下：



```
create_clock -name {SYSCLK} -period 40.000 -waveform { 0.000 20.000 } [get_ports {clk}]
```

- (5) 接下来,我们对工程进行全编译,不仅要让刚刚添加的时钟约束生效,也要生成可以下载到 FPGA 芯片中的配置文件。
- (6) 最后,我们便是要将配置文件下载到实验板中,接着我们便会看到 8 个 LED 飞速的在流水闪烁着。若想改变 LED 闪烁的速度,大家在那个 20 位的计数器上做点改动即可。
- (7) 在本节中,如果大家发现蜂鸣器在流水灯跑起来以后,蜂鸣器也跟着叫个不停,一定不要吃惊,大家可以点击菜单栏的 Assignments→Device, 接着点击 Device and Pin Options。选择 Unused Pins, 我们看到 Reserve all unused pins 后面默认是 As input tri-stated with weak pull-up, 我们需要将其改为 As input tri-stated。这个选项主要是设置 FPGA 中不使用管脚(即未做管脚分配)的默认状态。如果设为默认的输入弱上拉,那么意味着控制蜂鸣器的三极管将会导通,那么蜂鸣器发声也就不足为奇了。





6.4 逻辑（Verilog）实例 5——模式流水灯

6.4.1 实验原理

本实验的原理是 3 个拨码开关控制流水灯处于三种不同的工作模式，功能如表所示。

拨码开关状态	流水灯工作模式
1=OFF, 2=OFF, 3=OFF	8 个 LED 停止闪烁，某个 LED 将处于点亮状态
1=ON, 2=DC, 3=DC	流水灯从右往左流动
1=DC, 2=ON, 3=DC	流水灯从左往右流动
1=DC, 2=DC, 3=ON	所有 LED 一起闪烁，某个 LED 和其他 7 个 LED 闪烁状态正好相反

说明：

- 1、2、3 对应 SF-BASE 板上 S1 的三个拨码开关，和板上的丝印对应。
- OFF 表示拨码开关处于关闭状态，ON 表示拨码开关处于开启状态，DC（don't care）表示处于 OFF 或 ON 状态都可以。

这里我们需要注意，当 3 个拨码开关都处于 OFF 时，8 个 LED 熄灭；而当任意一个或多个拨码开关处于 ON 状态时，流水灯处于点亮工作状态，3 个拨码开关是由优先级关系的，拨码开关 1 优先级最高，它只要处于 ON，不用 care 其他两个拨码开关的状态，就一定是流水灯从右往左流动；拨码开关 2 次之，当拨码开关 1 处于 OFF 且拨码开关处于 ON 状态时，不用 care 拨码开关 3 的状态，流水灯从左往右流动；最后，如果只有拨码开关 3 处于 ON，那么所有 LED 一起闪烁。

6.4.2 Verilog 参考代码

```
module ex6(  
    clk, rst_n,  
    sw, led  
);  
input clk;  
input rst_n;  
input[3:1] sw;  
output reg[7:0] led;
```



```
//-----  
reg[22:0] cnt;  
  
always @ (posedge clk or negedge rst_n)  
    if(!rst_n) cnt <= 23'd0;  
    else cnt <= cnt+1'b1;  
  
//-----  
always @ (posedge clk or negedge rst_n)  
    if(!rst_n) led <= 8'b1111_1110;  
    else if(cnt == 23'h7fffff) begin  
        if(!sw[1]) led <= {led[6:0],led[7]};  
        else if(!sw[2]) led <= {led[0],led[7:1]};  
        else if(!sw[3]) led <= ~led;  
        else ;  
    end  
    else ;  
  
endmodule
```

6.4.3 仿真验证

```
`timescale 1 ns/ 1 ps  
module ex6_vlg_tst();  
  
    reg clk;  
    reg rst_n;  
    reg[3:1] sw;  
    wire[7:0] led;  
  
    ex6 il (  
        .clk(clk),  
        .led(led),  
        .sw(sw),  
        .rst_n(rst_n)  
    );  
endmodule
```



```
integer i;

initial begin
    rst_n = 0;
    sw = 3'b000;
    clk = 0;
    #1000;
    @(posedge clk);
    rst_n = 1;
    for(i=0;i<8;i=i+1) begin
        #4000_000_000;
        sw = sw+1'b1;
    end
    $stop;
end

always #20 clk = ~clk;

always @(led)
    $display("current led value : %x,time :%t\n",led,$time);

endmodule
```

在上一个实例的仿真代码基础上，我们使用 **for** 语句来产生全部 8 中拨码开关状态，每种状态下都保持 4s，最后通过打印出来的 LED 输出的值去判断是否满足设计要求。



```

# run -all
# current led value : fe,time : 0
# current led value : fd,time : 335545340000
# current led value : fb,time : 671089660000
# current led value : f7,time : 1006633980000
# current led value : ef,time : 1342178300000
# current led value : df,time : 1677722620000
# current led value : bf,time : 2013266940000
# current led value : 7f,time : 2348811260000
# current led value : fe,time : 2684355580000
# current led value : fd,time : 3019899900000
# current led value : fb,time : 3355444220000
# current led value : f7,time : 3690988540000
# current led value : fb,time : 4026532860000
# current led value : fd,time : 4362077180000
# current led value : fe,time : 4697621500000
# current led value : 7f,time : 5033165820000
# current led value : bf,time : 5368710140000
# current led value : df,time : 5704254460000
# current led value : ef,time : 6039798780000
# current led value : f7,time : 6375343100000
# current led value : fb,time : 6710887420000
# current led value : fd,time : 7046431740000
# current led value : fe,time : 7381976060000
# current led value : 7f,time : 7717520380000
# current led value : fe,time : 8053064700000
# current led value : fd,time : 8388688880000
    
```

6.4.4 工程实践

- (1) 新建工程，命名为 ex6，把这个工程放在专门的文件夹下，其他设置参考前面章节。
- (2) 新建 Verilog 源文件，命名为 ex6，输入前面给出的设计代码。使用 ModelSim-Altera 对设计代码进行仿真验证。
- (3) 综合编译后进行管脚分配，本例程的管脚分配如下所示。

Node Name	Direction	Location	I/O Bank
clk	Input	PIN_22	1
led[7]	Output	PIN_126	7
led[6]	Output	PIN_125	7
led[5]	Output	PIN_124	7
led[4]	Output	PIN_121	7
led[3]	Output	PIN_120	7
led[2]	Output	PIN_119	7
led[1]	Output	PIN_115	7
led[0]	Output	PIN_114	7
rst_n	Input	PIN_91	6
sw[3]	Input	PIN_105	6
sw[2]	Input	PIN_104	6
sw[1]	Input	PIN_103	6

《圣经》箴言九 11 “敬畏耶和华是智慧的开端，认识至胜者便是聪明。”



- (4) 参考前面章节, 打开 TimeQuest, 我们新建一个 SDC 文件, 然后对时钟 clk 做约束, 约束脚本如下:

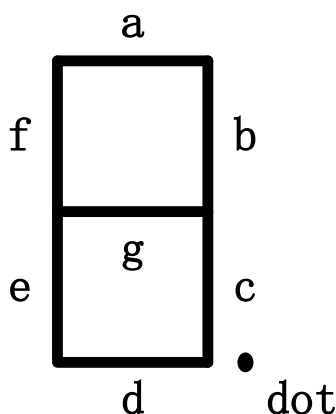
```
create_clock -name {SYSCLK} -period 40.000 -waveform { 0.000 20.000 } [get_ports {clk}]
```

- (5) 接下来, 我们对工程进行全编译, 不仅要让刚刚添加的时钟约束生效, 也要生成可以下载到 FPGA 芯片中的配置文件。
- (6) 最后, 我们便是要将配置文件下载到实验板中, 接着我们以此波动 3 个拨码开关, 将所有 8 中可能的情况都尝试一下, 看看 LED 的工作模式是否和我们设计的相一致。

6.5 逻辑 (Verilog) 实例 6——数码管显示

6.5.1 实验原理

先来了解一下数码管的工作原理。如图所示, 这是一个典型的带小数点的一位数码管。如果忽略小数点, 我们通常称它为 7 段数码管, 所谓 7 段, 也是 7 个发光二极管。任意一个 0-9 的阿拉伯数字的显示, 只要通过这 7 个发光二极管进行亮或灭的组合都可以实现。例如, 我们要显示数字 0, 那么只要让发光二极管 a、b、c、d、e、f 点亮 (g 和 dot 熄灭) 就可以了。



接下来, 大家可能就要关心着这 7 个发光二极管是如何控制的, 我们又是如何通过 FPGA 的 I/O 口去点亮或熄灭任意一个发光二极管? 很简单, 如图所示, 一个带小数点的数码管的



所有 8 个发光二极管的正极或负极有一个公共端，通常必须接 GND（共阴极数码管）或者接 VCC（共阳极数码管），而另一端不共接的 8 个引脚就留给用户控制了。例如，如果我们使用的是共阴极的数码管，那么我们在使用该数码管时就要将其公共端接地（或者接低电平 0），我们的应用中，把这个公共端连接到了 FPGA 的 I/O 脚上。如果 FPGA 的这个 I/O 脚输出低电平 0，那么这个数码管就能够显示数字；如果这个 I/O 输出高电平 1，那么无论数码管的 8 个段选端输出 0 还是 1，都无法将 8 个发光二极管的任意一个点亮，这也达到了关闭数码管显示的效果。这样一来，这个数码管的公共端被我们当做了数码管片选管脚使用了，虽然不是名副其实的“片选”，但还真达到了异曲同工之妙。

我们的例程要实现的功能比较简单基础：让 4 个数码管每隔 1s 不断的递增计数显示，计数范围为 0-F。为了便于代码编写控制 7 个用于段选（不包括小数点）的发光二极管显示不同的字符，这里只做了一个简单的对应表，把不同字符显示时的 7 个 I/O 值进行编码，如表所示。

数码管编码表

数字/字符	0	1	2	3	4	5	6	7
编码(16 进制)	3f	06	5b	4f	66	6d	7d	07
数字/字符	8	9	A	B	C	D	E	F
编码(16 进制)	7f	6f	77	7c	39	5e	79	71

6.5.2 Verilog 参考代码

```
module ex7(  
    clk, rst_n,  
    seg_db, seg_cs  
);  
input clk;  
input rst_n;  
output reg[7:0] seg_db;  
output reg[3:0] seg_cs;  
  
//-----  
//产生 1s 定时的时间
```



```
reg[23:0] cnt;

always @ (posedge clk or negedge rst_n)
    if(!rst_n) cnt <= 24'd0;
    else if(cnt >= 24'd25_000_000) cnt <= 24'd0;
    else cnt <= cnt+1'b1;

    //1s 定时标志位, 高电平有效一个时钟周期
wire timer_1s = (cnt == 24'd25_000_000);

//-----
    //产生按秒不断计数的 4 个 16 进制显示到数码管的数据
reg[15:0] dis_db;

always @ (posedge clk or negedge rst_n)
    if(!rst_n) dis_db <= 16'd0;
    else if(timer_1s) dis_db <= dis_db+1'b1;
    else ;

//-----
    //采用分时机制, 分别将当前显示的数码管选中并赋值
reg[3:0] cur_disdb;

always @ (posedge clk or negedge rst_n)
    if(!rst_n) cur_disdb <= 4'd0;
    else begin
        case(cnt[7:6])
            2'b00: begin //个位显示
                seg_cs <= 4'b1110;
                cur_disdb <= dis_db[3:0];
            end
            2'b01: begin //十位显示
                seg_cs <= 4'b1101;
                cur_disdb <= dis_db[7:4];
            end
            2'b10: begin //百位显示
                seg_cs <= 4'b1011;
                cur_disdb <= dis_db[11:8];
            end
        endcase
    end
```



```
        end
        2'b11: begin    //千位显示
            seg_cs <= 4'b0111;
            cur_disdb <= dis_db[15:12];
        end
        default: ;
    endcase
end

//-----
//数码管显示 0~F 对应段选输出
parameter  SEG_NUM0    = 8'h3f, //c0,
            SEG_NUM1    = 8'h06, //f9,
            SEG_NUM2    = 8'h5b, //a4,
            SEG_NUM3    = 8'h4f, //b0,
            SEG_NUM4    = 8'h66, //99,
            SEG_NUM5    = 8'h6d, //92,
            SEG_NUM6    = 8'h7d, //82,
            SEG_NUM7    = 8'h07, //F8,
            SEG_NUM8    = 8'h7f, //80,
            SEG_NUM9    = 8'h6f, //90,
            SEG_NUMA    = 8'h77, //88,
            SEG_NUMB    = 8'h7c, //83,
            SEG_NUMC    = 8'h39, //c6,
            SEG_NUMD    = 8'h5e, //a1,
            SEG_NUME    = 8'h79, //86,
            SEG_NUMF    = 8'h71; //8e;

//段选数据译码
always @(cur_disdb) begin
    case(cur_disdb)
        4'h0: seg_db <= SEG_NUM0;
        4'h1: seg_db <= SEG_NUM1;
        4'h2: seg_db <= SEG_NUM2;
        4'h3: seg_db <= SEG_NUM3;
        4'h4: seg_db <= SEG_NUM4;
        4'h5: seg_db <= SEG_NUM5;
        4'h6: seg_db <= SEG_NUM6;
```



```
4'h7: seg_db <= SEG_NUM7;
4'h8: seg_db <= SEG_NUM8;
4'h9: seg_db <= SEG_NUM9;
4'ha: seg_db <= SEG_NUMA;
4'hb: seg_db <= SEG_NUMB;
4'hc: seg_db <= SEG_NUMC;
4'hd: seg_db <= SEG_NUMD;
4'he: seg_db <= SEG_NUME;
4'hf: seg_db <= SEG_NUMF;
default: ;
endcase
end

endmodule、
```

6.5.3 仿真验证

本实例不给出具体的仿真脚本，下面给出几个设计要点供参考：

- 产生时钟和复位信号。
- 对接收到的 seg_cs 和 seg_db 做解码，确定在某个数码管被选中的情况下，当前显示的数据是什么。
- 产生一个和系统的秒同步的信号，判断在每秒时间周期中，对应的数码管显示是否正确。

6.5.4 工程实践

- (1) 新建工程，命名为 ex7，把这个工程放在专门的文件夹下，其他设置参考前面章节。
- (2) 新建 Verilog 源文件，命名为 ex7，输入前面给出的设计代码。使用 ModelSim-Altera 对设计代码进行仿真验证。
- (3) 综合编译后进行管脚分配，本例程的管脚分配如下所示。



Node Name	Direction	Location	I/O Bank
clk	Input	PIN_22	1
rst_n	Input	PIN_91	6
seg_cs[3]	Output	PIN_99	6
seg_cs[2]	Output	PIN_98	6
seg_cs[1]	Output	PIN_87	5
seg_cs[0]	Output	PIN_100	6
seg_db[7]	Output	PIN_120	7
seg_db[6]	Output	PIN_115	7
seg_db[5]	Output	PIN_121	7
seg_db[4]	Output	PIN_126	7
seg_db[3]	Output	PIN_124	7
seg_db[2]	Output	PIN_119	7
seg_db[1]	Output	PIN_114	7
seg_db[0]	Output	PIN_125	7

- (4) 参考前面章节, 打开 TimeQuest, 我们新建一个 SDC 文件, 然后对时钟 clk 做约束, 约束脚本如下:

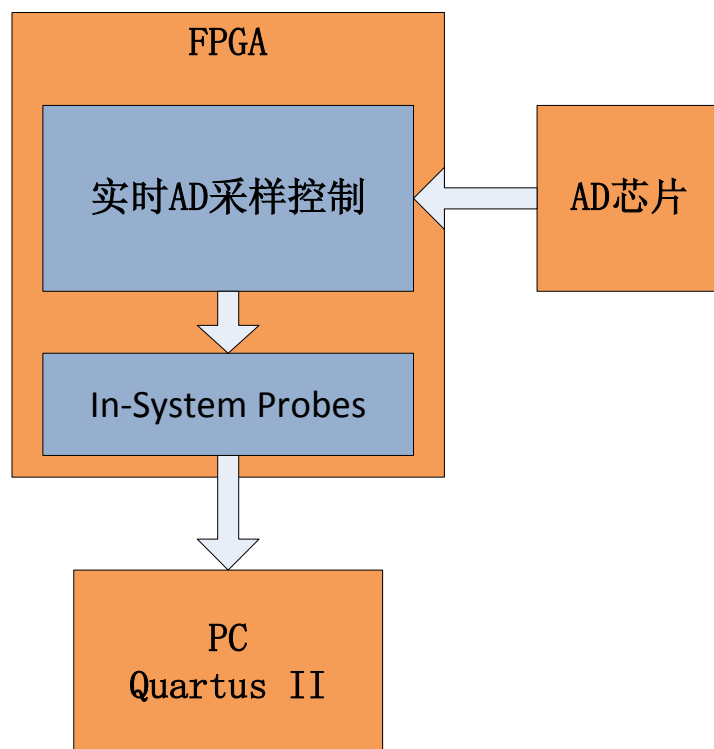
```
create_clock -name {SYSCLK} -period 40.000 -waveform { 0.000 20.000 } [get_ports {clk}]
```

- (5) 接下来, 我们对工程进行全编译, 不仅要让刚刚添加的时钟约束生效, 也要生成可以下载到 FPGA 芯片中的配置文件。
- (6) 最后, 我们便是要将配置文件下载到实验板中, 接着我们可以看到 4 位数码管开始不断的计数递增, 同时由于我们的管脚适合 8 个 LED 管脚复用的, 所以 LED 也会很有节奏的不断变化。

6.6 逻辑(Verilog)实例 7——基于 In-System Sources and Probes Editor 的 AD 采集

6.6.1 概述

该实例用 FPGA 的内部逻辑设计一个实时 AD 采样控制功能, 该模块一方面不断的通过 SPI 接口采集 AD 芯片 TLC549 的模拟电压值, 另一方面我们例化一个 In-System Sources and Probes Editor 用于在 PC 端查看当前采样值。



6.6.2 AD 采样控制原理

AD 芯片 TLC549 的控制使用了比较简化（单向数据传输）的 SPI 接口，接口上只需要片选信号 `adc_cs_n`、时钟信号 `adc_clk` 和输入数据信号 `adc_data`。控制时序如图所示，只要每次片选有效后产生 8 个时钟周期依次读取 AD 采样数据即可。在片选信号拉低后大约 1.4us（ t_{su} ）第一个采样数据出现在 `adc_data` 上，此时时钟 `adc_clk` 上升沿可以采样数据，时钟信号 `adc_clk` 的最高频率可以达到 1.1MHz。两次数据采样间隔必须大于 17us（ t_{wh} ）。

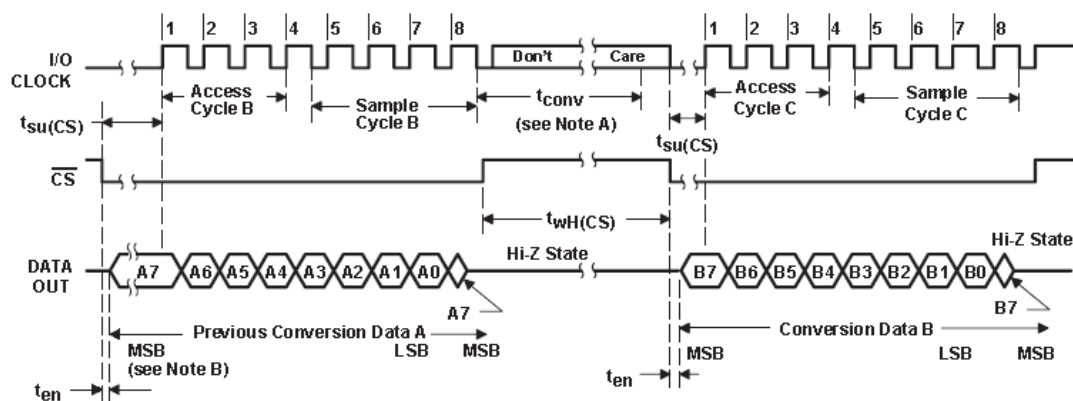


图 7.15 AD 芯片控制时序

逻辑实现上，使用了三段式状态机。状态 TSUDL 和 START 拉低片选信号并等待第一个转换数据出现在数据信号 `adc_data` 上；状态 DTRAN 进行 8 个串行输入数据的采样；状态 STOP 《圣经》箴言九 11 “敬畏耶和华是智慧的开端，认识至胜者便是聪明。”

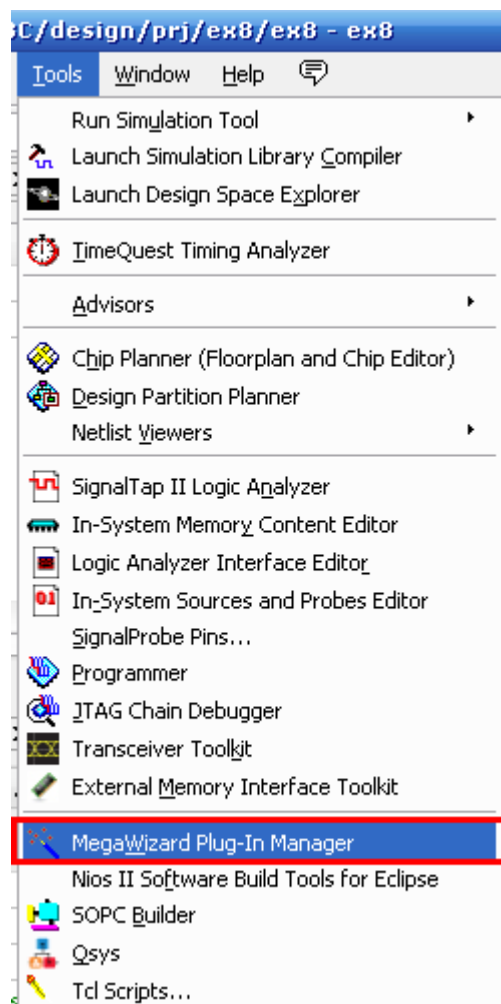


完成一次采样; 状态 TWHDL 延时等待至少 17us, 保证两次数据采样有足够的时间间隔。

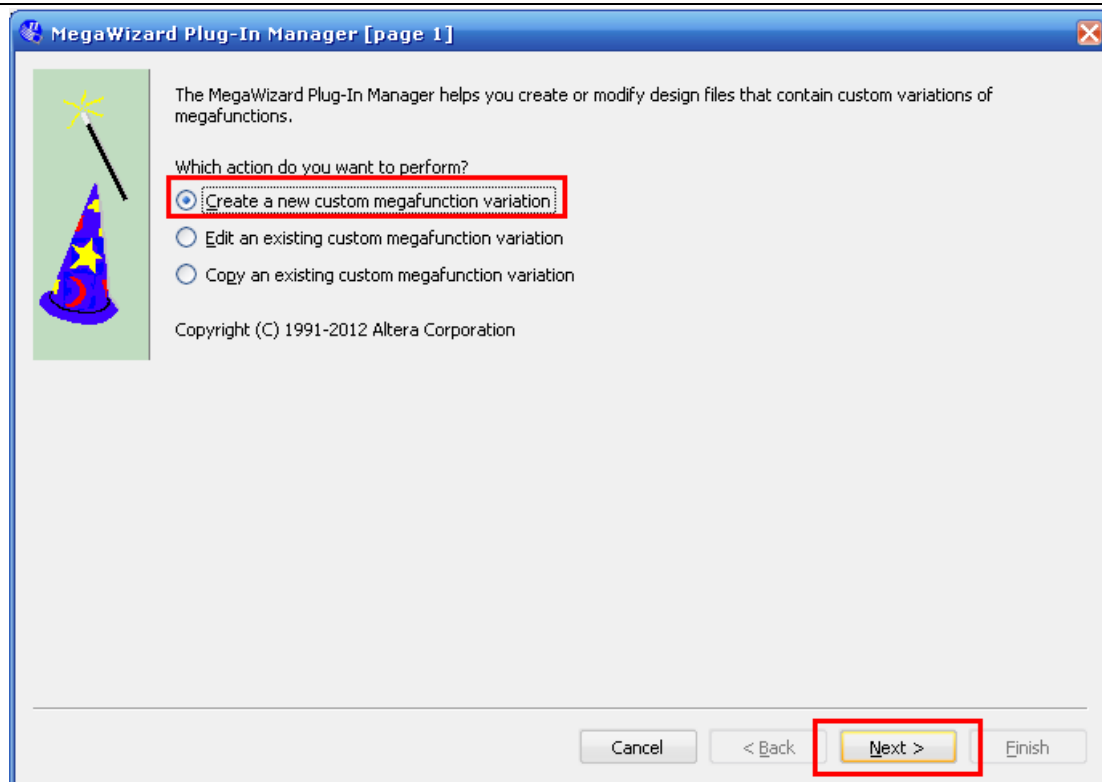
6.6.3 In-System Sources and Probes Editor 例化

因为 In-System Sources and Probes Editor 是 Quartus II 中的一个 IP 核, 所以我们需要先对其进行设置添加, 然后例化到我们的代码中。

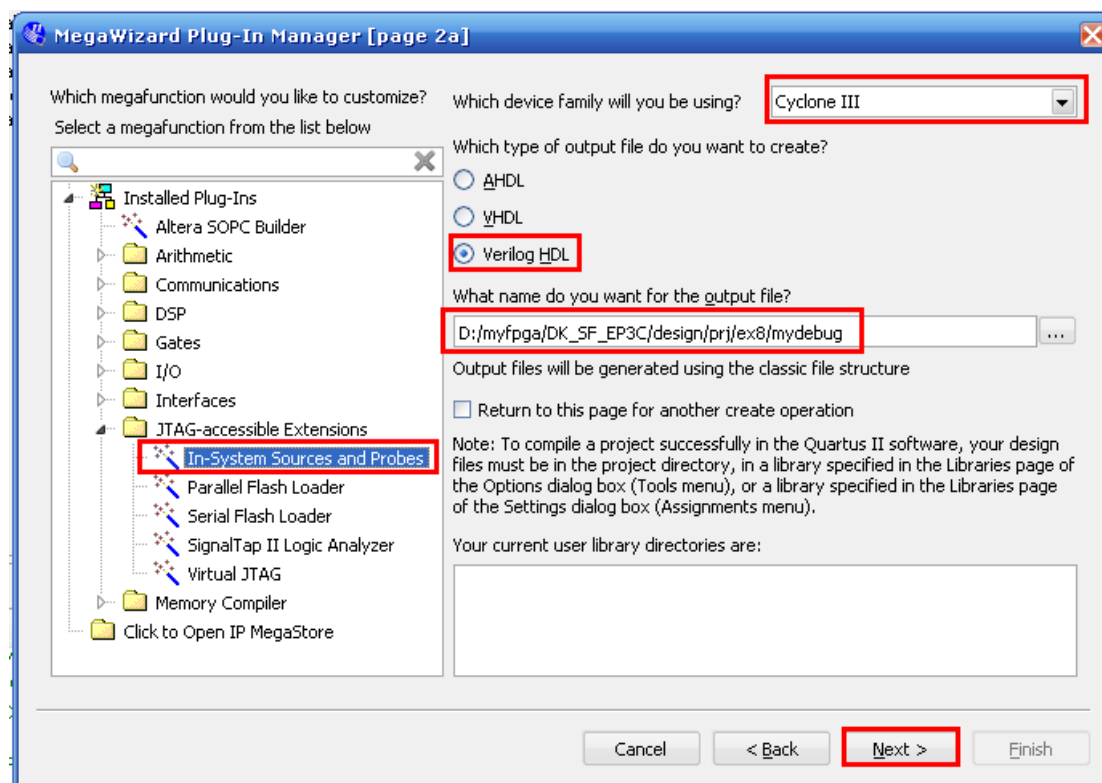
点击菜单栏 Tools→MegaWizard Plug-In Manager。



选择 Create a new custom megafunction variation。



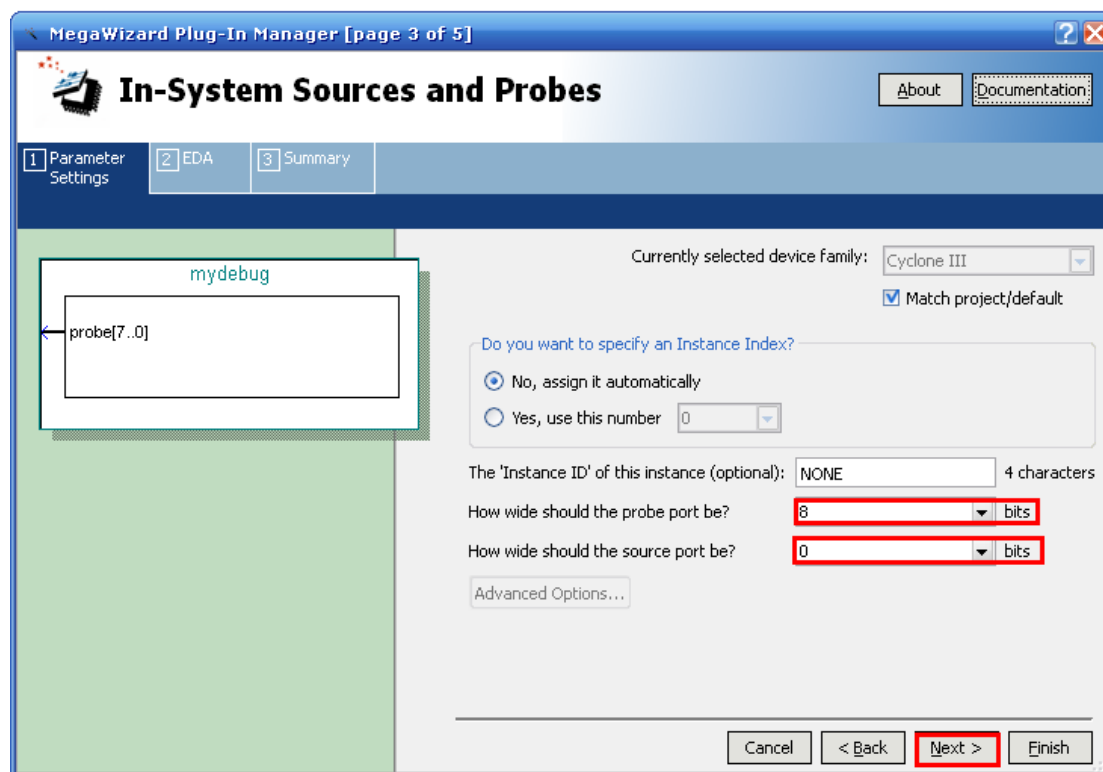
在 MegaWizard Plug-In Manager 中, 做如图所示的设置, 注意在 What name do you want for the output file 中, 需要输入一个该代码的文件名, 如这里在工程目录 ex8 后面命名 mydebug. 最后点击 Next 进入下一步。



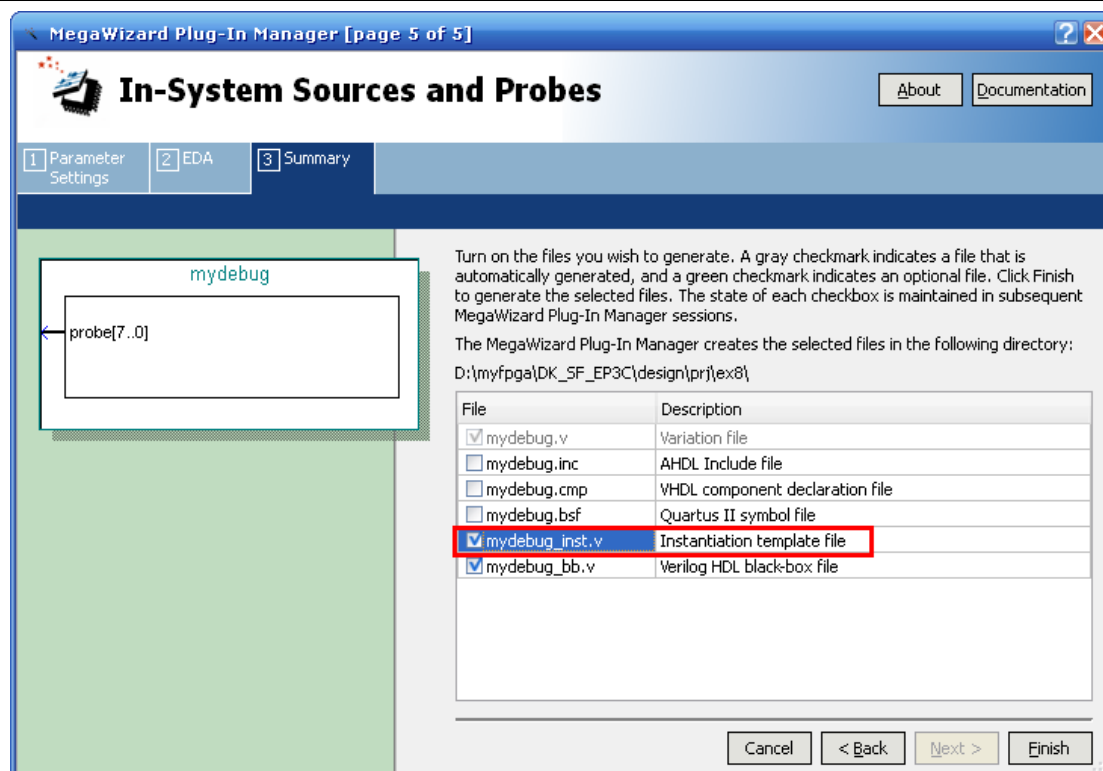
如图所示, 我们需要在 In-System Sources and Probes Editor 配置页面中, 设置 probe port



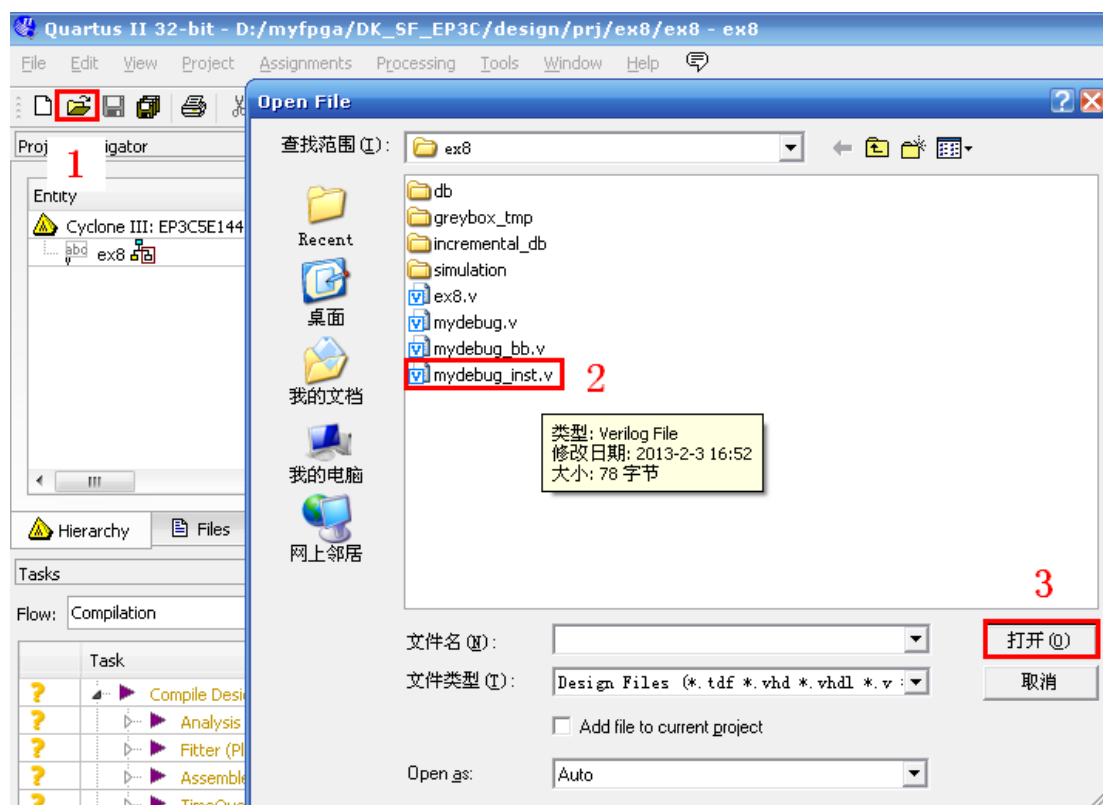
为 8bits, source port 为 0bits。Probe 的功能是读出（对于工程设计为输出）我们的 FPGA 设计中的某些寄存器，而 source 则是写数据（对于工程设计为输入）到 FPGA 设计中。本实例是要实时监控查看输出的 AD 采样值，所以选择 probe。完成设置选择 Next。



第二步 EDA 不需要做设置，我们来到 Summary 配置页面，注意要勾选 mydebug_inst.v 文件，这个文件里面有 In-System Sources and Probes Editor 的例化模板。完成这些设置后，点击 Finish 完成所有配置。



回到 Quartus II 界面, 我们点击 Open File 快捷按钮, 找到工程目录 ex8 下的 mydebug_inst.v 文件, 打开它。



我们可以看到如下的代码, 可以将它复制到工程源码中, 对括号内的信号做相应的映射



更改, 具体参考后面的源码。

```
mydebug mydebug_inst (  
    .probe ( probe_sig ),  
    .source ( source_sig )  
);
```

6.6.4 Verilog 参考代码

```
module ex8(  
    clk, rst_n,  
    adc_data, adc_cs_n, adc_clk,  
);  
input clk;        //25MHz  
input rst_n;      //低电平复位信号  
input adc_data;    //TLC549 数据信号  
output adc_cs_n;   //TLC549 片选信号, 低电平有效  
output adc_clk;    //TLC549 时钟信号  
  
//-----  
//定时计数逻辑  
reg[5:0] cntus; //2us 计数器  
  
always @(posedge clk or negedge rst_n)  
    if(!rst_n) cntus <= 6'd0;  
    else if((cntus < 6'd49) && (cstate != IDLE)) cntus <= cntus+1'b1;  
    else cntus <= 6'd0;  
  
wire dchag_flag = (cntus == 6'd0); //ADC 时钟下降沿标志位, 高有效一个时钟周期  
wire dlock_flag = (cntus == 6'd24); //ADC 时钟上升沿标志位, 高有效一个时钟周期  
  
//-----  
//ADC 工作状态机  
parameter IDLE    = 3'd0,  
           TSUDL   = 3'd1,  
           START   = 3'd2,
```



```
DTRAN    = 3'd3,
STOP     = 3'd4,
TWHDL    = 3'd5;

reg[2:0] bitnum;    //采样数据位寄存器 7-0
reg[4:0] d17uscnt;  //Twh 延时计数器
reg[7:0] adc_dinr;  //模数转换数据寄存器
reg[7:0] adc_dinlock; //模数转换数据寄存器, 实时锁存
reg[2:0] cstate, nstate; //状态寄存器

//状态迁移
always @(posedge clk or negedge rst_n)
    if(!rst_n) cstate <= IDLE;
    else cstate <= nstate;

//数据采集位寄存器控制
always @(posedge clk or negedge rst_n)
    if(!rst_n) bitnum <= 3'd0;
    else if(nstate == IDLE) bitnum <= 3'd7;
    else if((nstate == DTRAN) && dlock_flag) bitnum <= bitnum-1'b1;

//Twh 延时计数器控制
always @(posedge clk or negedge rst_n)
    if(!rst_n) d17uscnt <= 5'd0;
    else if((nstate == TWHDL) && dchag_flag) d17uscnt <= d17uscnt+1'b1;
    else if(nstate == IDLE) d17uscnt <= 5'd0;

//状态控制
always @(cstate or dchag_flag or bitnum or d17uscnt)
    case(cstate)
        IDLE:    nstate <= TSUDL;
        TSUDL:   if(dchag_flag) nstate <= START;
                 else nstate <= TSUDL;
        START:   if(dchag_flag) nstate <= DTRAN;
                 else nstate <= START;
        DTRAN:   if(dchag_flag && (bitnum == 3'd7)) nstate <= STOP;
                 else nstate <= DTRAN;
        STOP:    if(dchag_flag) nstate <= TWHDL;
```



```
        else nstate <= STOP;
    TWHDL: if(dchag_flag && (d17uscnt == 5'd18)) nstate <= IDLE;
        else nstate <= TWHDL;
    default: nstate <= IDLE;
endcase

//位数据锁存
always @(posedge clk or negedge rst_n)
    if(!rst_n) adc_dinlock <= 8'h00;
    else if((nstate == DTRAN) && dlock_flag) adc_dinlock[bitnum] <= adc_data;
//帧数据锁存
always @(posedge clk or negedge rst_n)
    if(!rst_n) adc_dinr <= 8'h00;
    else if(nstate == STOP) adc_dinr <= adc_dinlock;

assign adc_cs_n = ~((cstate == DTRAN) | (cstate == START) | (cstate == TSUDL));

//-----
//时钟速率控制, 1MHz
reg adc_clkr; //TLC549 时钟信号寄存器

always @(posedge clk or negedge rst_n)
    if(!rst_n) adc_clkr <= 1'b0;
    else if((nstate == DTRAN) && (cntus > 5'd12)) adc_clkr <= 1'b1;
    else adc_clkr <= 1'b0;

assign adc_clk = adc_clkr;

//-----
//In-System Sources and Probes Editor 例化
mydebug mydebug_inst (
    .probe ( adc_dinr ),
    .source ( )
);

endmodule
```



6.6.5 仿真验证

对于本实例的仿真，这里也提几点注意事项和设计关键点。

- 参考 AD 芯片 TLC549 的 datasheet，把它的 SPI 接口时序弄明白，设计一个 SPI 从机。
- 模拟产生一个固定的 AD 采样值，或者根据时间变化的值，在后面验证结果时方便查看验证。

6.6.6 工程实践

- (1) 新建工程，命名为 **ex8**，把这个工程放在专门的文件夹下，其他设置参考前面章节。
- (2) 新建 Verilog 源文件，命名为 **ex8**，输入前面给出的设计代码。使用 ModelSim-Altera 对设计代码进行仿真验证。
- (3) 综合编译后进行管脚分配，本例程的管脚分配如下所示。

Node Name	Direction	Location	I/O Bank
adc_clk	Output	PIN_113	7
adc_cs_n	Output	PIN_111	7
adc_data	Input	PIN_112	7
clk	Input	PIN_22	1
rst_n	Input	PIN_91	6
<<new node>>			

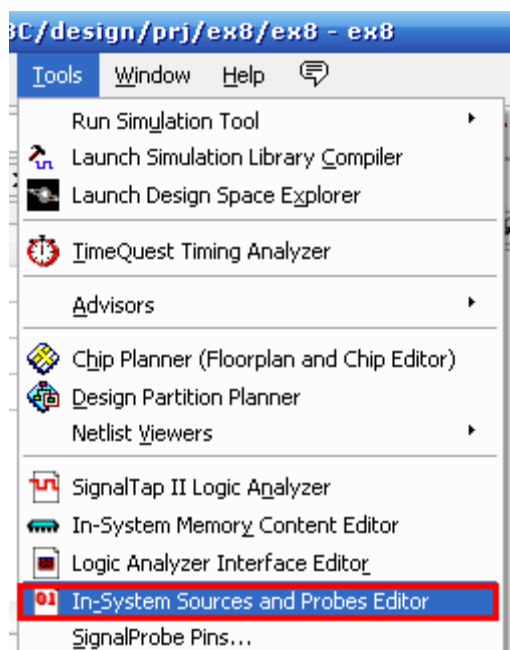
- (4) 参考前面章节，打开 TimeQuest，我们新建一个 SDC 文件，然后对时钟 clk 做约束，约束脚本如下：

```
create_clock -name {SYSCLK} -period 40.000 -waveform { 0.000 20.000 } [get_ports {clk}]
```

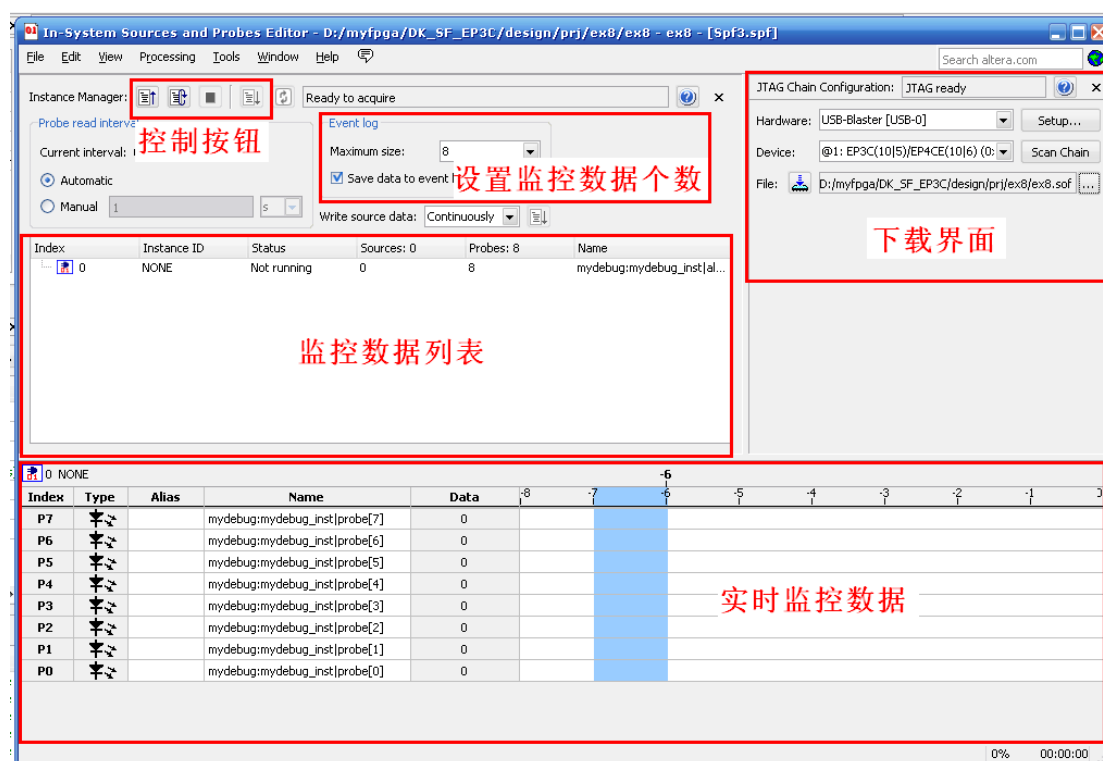
- (5) 我们对工程进行全编译，不仅要让刚刚添加的时钟约束生效，也要生成可以下载到 FPGA 芯片中的配置文件。
- (6) 确认 SF-BASE 板上，跳线帽连接好 P3 的 PIN1 和 PIN2。给板子上电。
- (7) 既可以通过正常的 Programmer 进行下载，也可以通过 In-System Sources and Probes Editor 调试界面进行下载。
- (8) 接下来使用 In-System Sources and Probes Editor 进行调试，如图点击菜单栏的



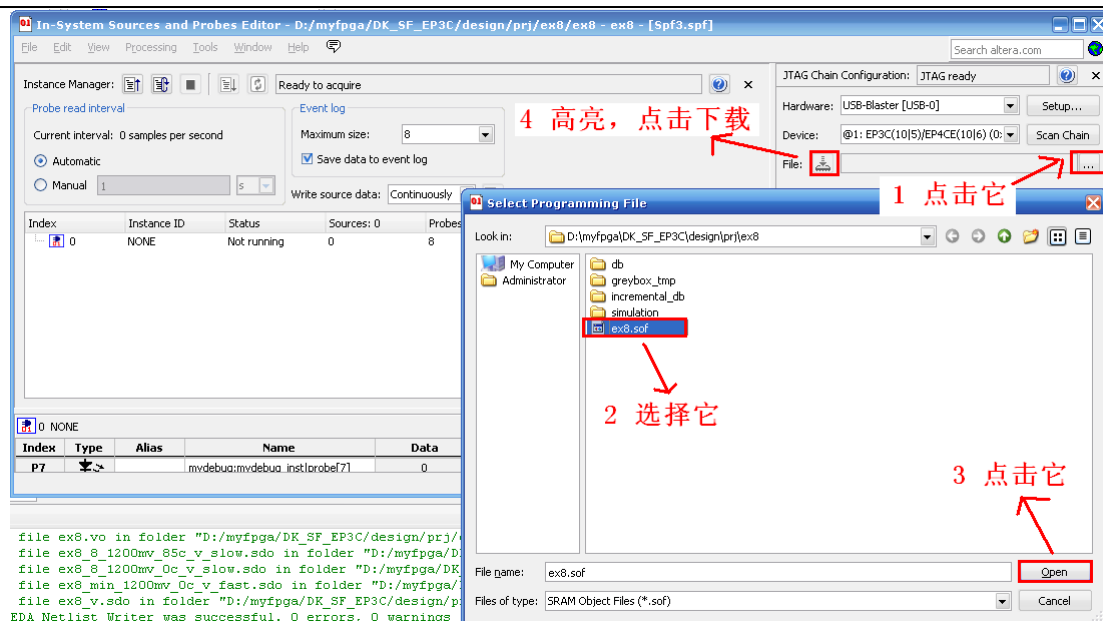
Tools→In-System Sources and Probes Editor。



In-System Sources and Probes Editor 的主界面如图所示。



如果还未使用 Programmer 执行过下载操作,可以如图所示在 In-System Sources and Probes Editor 界面中执行。



在实时数据监控窗口里，我们看到前面设置的所有的 Probe 的 8 个位都列出来了，为了后面便于观察，我们可以在最下方实时监控数据的窗口右键单击，选择 Select All。

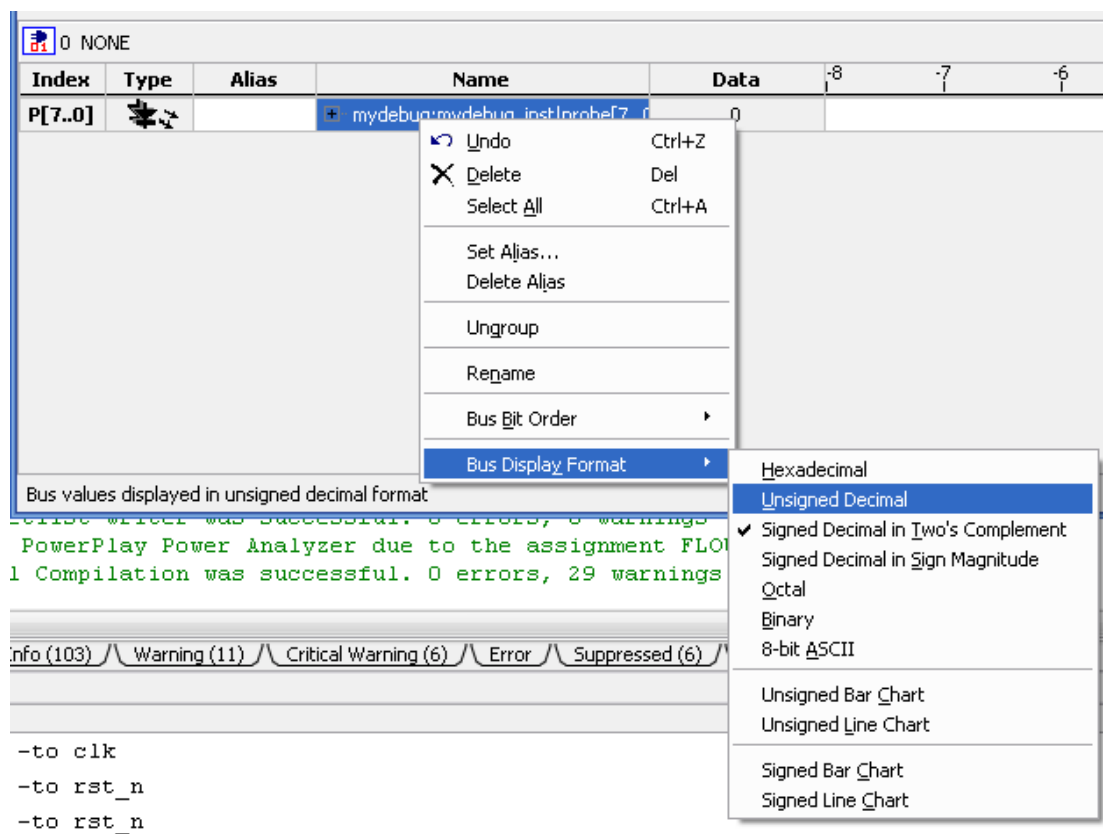
Index	Type	Alias	Name	Data
P7			mydebug:mydebug_inst probe[7]	0
P6			mydebug:mydebug_inst probe[6]	0
P5			mydebug:mydebug_inst probe[5]	0
P4			mydebug:mydebug_inst probe[4]	0
P3			mydebug:mydebug_inst probe[3]	0
P2			mydebug:mydebug_inst probe[2]	0
P1			mydebug:mydebug_inst probe[1]	0
P0			mydebug:mydebug_inst probe[0]	0

Name 下面的所有位都变蓝表示选中，接着右击选择 Group。

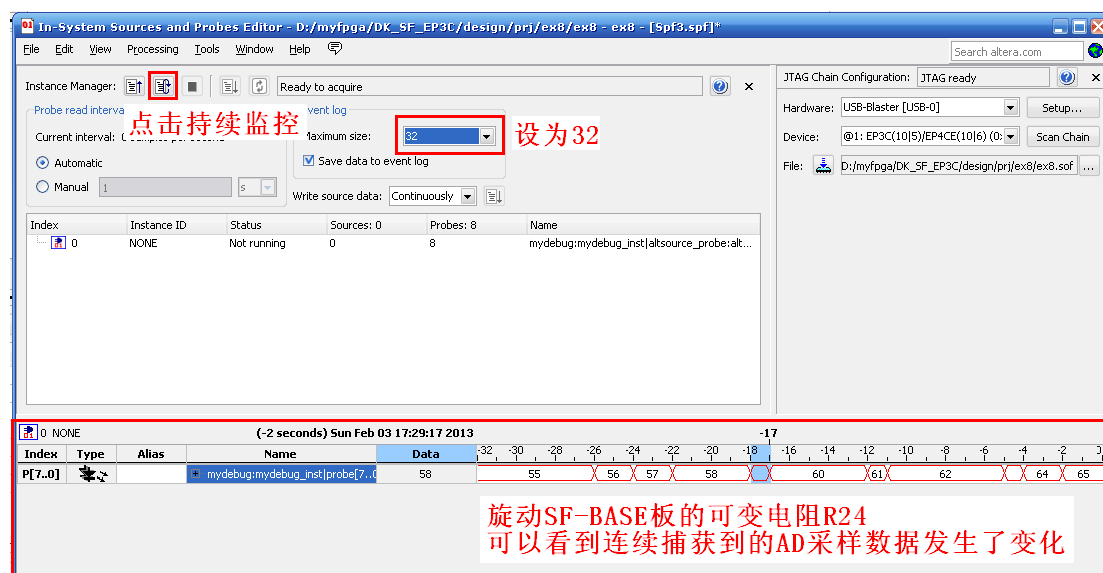
Index	Type	Alias	Name	Data
P7			mydebug:mydebug_inst probe[7]	0
P6			mydebug:mydebug_inst probe[6]	0
P5			mydebug:mydebug_inst probe[5]	0
P4			mydebug:mydebug_inst probe[4]	0
P3			mydebug:mydebug_inst probe[3]	0
P2			mydebug:mydebug_inst probe[2]	0
P1			mydebug:mydebug_inst probe[1]	0
P0			mydebug:mydebug_inst probe[0]	0



此时所有的 8bit 都成为一组里，我们再次右键单击然后选择 Bus Display Format→Unsigned Decimal。这个设置将这个 group 显示的 8bit 数据设为无符号十进制，便于后续的观察。



最后，我们设置监控数组为 32（当然也可以更大，这个设置就是监控窗口中显示出来的数据个数），点击持续监控的按钮，开始数据捕获。如图所示，我们连续的旋动 SF-BASE 板的可变电阻 R24，改变输入模拟值，对应的 8bit 采集电压值也持续的变化了。





6.7 逻辑(Verilog)实例 8——基于 In-System Sources and Probes Editor 的 DA 输出

6.7.1 概述

与上一节类似，本节我们用 FPGA 的内部逻辑设计一个实时 DA 转换控制功能，该模块的 DA 转换数通过 In-System Sources and Probes Editor 的 source 来设置，当 source 发生变化时，会触发 IIC 接口执行一次 DA 转换输出相应的模拟电压值。

6.7.2 DA 采样控制原理

DA 芯片 DAC5571 的控制使用了标准模式，它的接口是大家耳熟能详的 IIC 接口，关于 IIC 通信的基本数据接口模式这里不详细介绍。如图所示，FPGA 作为 IIC 总线的主机，若要控制芯片 DAC5571 完成一次转换，则需要总共传输三个字节数据。首字节主要内容是从机地址（SLAVE ADDRESS）和读或写指示（R\W#）；第二个字节的高 4bit 是控制数据，低 4bit 是有效数据的高 4bit；第三字节的高 4bit 是有效数据的低 4bit，后 4bit 无效。

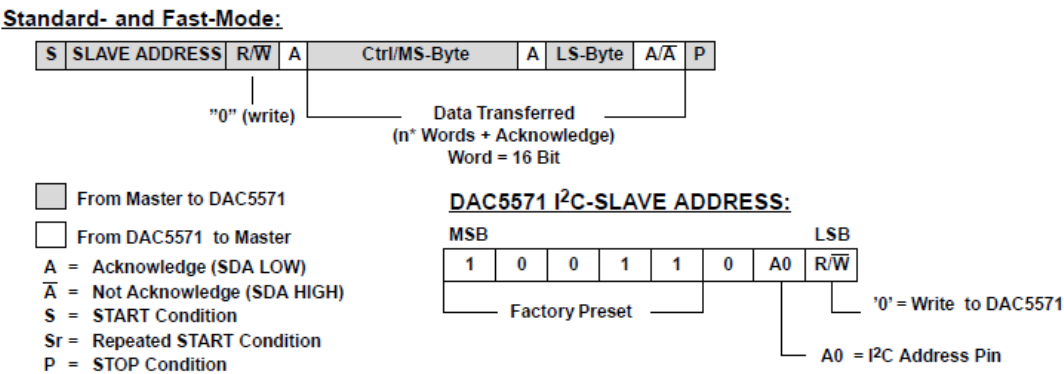


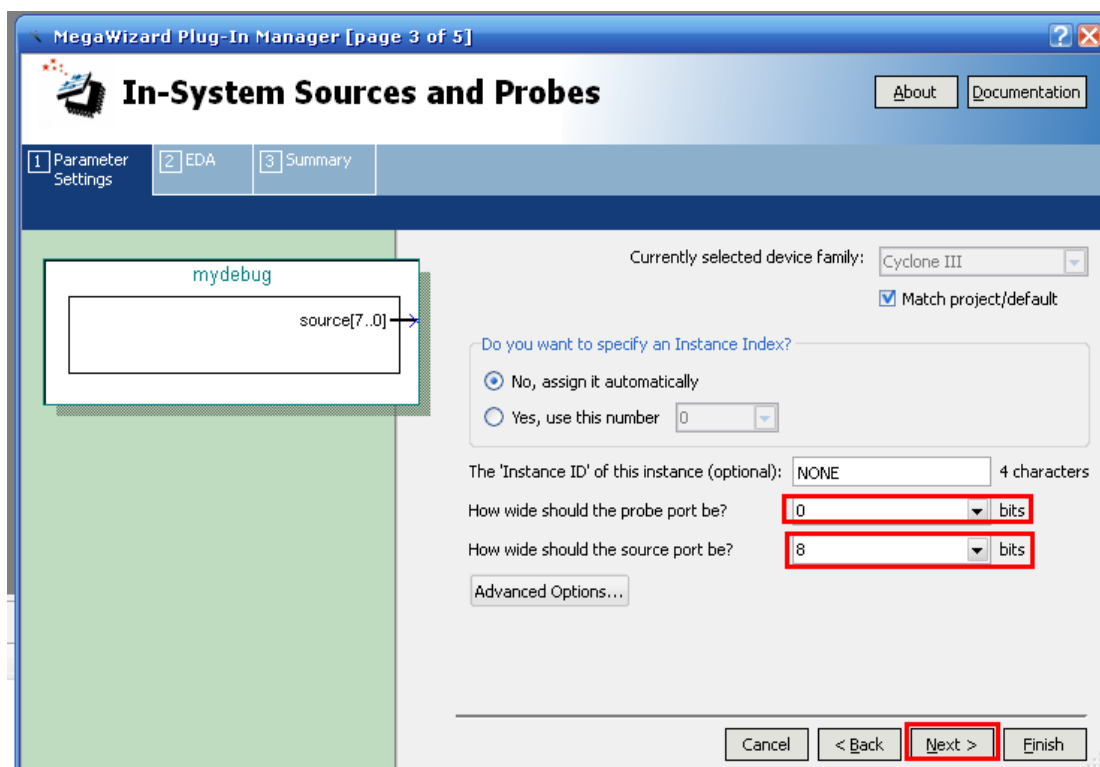
图 7.28 DA 芯片的 IIC 接口协议

IIC 读写控制的详细设计参考请参考源代码。



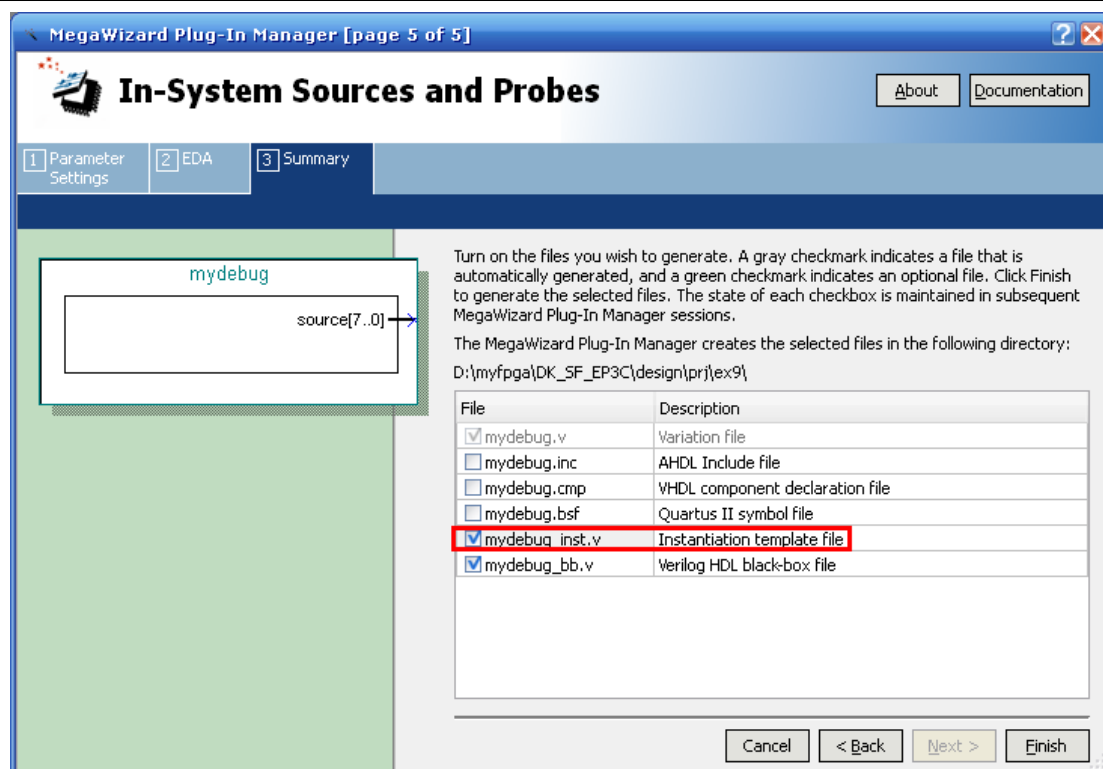
6.7.3 In-System Sources and Probes Editor 例化

基本的 IP 核的创建方法和上一节一样,不同的是在 In-System Sources and Probe 的 Parameter setting 中 probe 设置为 0bits、source 设置为 8bits, 如图所示。



这里的 source 是我们在后面使用 Quartus II 在线调试时可编辑更改的,通过更改它的值,就能够让 DA 芯片执行一次转换,不同的模拟电压值去驱动 LED 指示灯,它的亮度就会有所不同。

在 Summary 配置页面中,还是提醒大家别忘了勾选 mydebug_inst.v 文件。



打开 `mydebug_inst.v` 文件, 复制模板例化代码, `copy` 到工程源代码中, 对括号里面的信号名做修改, 和我们的实际工程信号映射上, 详细请参考源代码。

```
mydebug mydebug_inst (
    .probe ( probe_sig ),
    .source ( source_sig )
);
```

6.7.4 Verilog 参考代码

```
module ex9(
    clk,rst_n,
    scl,sda
);

input clk;      //25MHz
input rst_n;    //低电平复位信号
output scl;     //DAC5571 数据信号
inout sda;     //DAC5571 时钟信号

//-----
```

《圣经》箴言九 11 “敬畏耶和华是智慧的开端, 认识至胜者便是聪明。”



```
//In-System Sources and Probes Editor 例化
wire[7:0] dac_data; //数模转换数据

mydebug mydebug_inst (
    .probe ( ),
    .source ( dac_data )
);

//-----

//dac_en 信号产生, 以此执行一次 DAC 操作
reg[7:0] dac_datar;
reg dac_en; //DAC 输出使能, 高电平有效

//锁存一拍 source 输出的数据
always @(posedge clk or negedge rst_n)
    if(!rst_n) dac_datar <= 8'd0;
    else dac_datar <= dac_data;

//判断是否 dac_data 发生变化, 以此产生以此 DAC 操作
always @(posedge clk or negedge rst_n)
    if(!rst_n) dac_en <= 1'b0;
    else if(dac_datar != dac_data) dac_en <= 1'b1;
    else dac_en <= 1'b0;

//-----

//IIC 的数据速率控制 50kbps
reg[8:0] cnti; //分频计数寄存器, 0-499, 25M/500=50K

//计数逻辑
always @(posedge clk or negedge rst_n)
    if(!rst_n) cnti <= 9'd0;
    else if(cnti < 9'd499 && cstate != IDLE) cnti <= cnti + 1'b1;
    else cnti <= 9'd0;

wire scl_low = (cnti == 9'd374); //scl 的低电平中间点, 即 sda 的最佳数据更新点
wire scl_high = (cnti == 9'd124); //scl 的高电平中间点, 用于产生起始位或停止位

//scl 输出控制
```



```
assign scl = ~cnti[8];          //IIC 时钟信号

//-----
//DAC 的 IIC 总线传输控制
//状态参数定义
parameter IDLE      = 4'd0;
parameter START     = 4'd1;
parameter ADDR      = 4'd2;
parameter ACK1      = 4'd3;
parameter CMSB      = 4'd4;
parameter ACK2      = 4'd5;
parameter LSBI      = 4'd6;
parameter ACK3      = 4'd7;
parameter ACK4      = 4'd8;
parameter STOP      = 4'd9;

parameter DEVICE_ADDR = 8'b1001_1000; //参考 DAC5571.pdf 的 P15
wire[7:0] dac_mdata = {4'b0000, dac_data[7:4]}; //control/MSB 字节
wire[7:0] dac_ldata = {dac_data[3:0], 4'b0000}; //LSB/invalid 字节

reg[3:0] cstate, nstate; //状态寄存器
reg sdar; //IIC 数据信号寄存器
reg[2:0] bcnt; //传输位计数器 0-7
reg sdlink; //sda 数据方向控制位, 1--输出, 0--输入

always @(posedge clk or negedge rst_n)
    if(!rst_n) cstate <= IDLE;
    else cstate <= nstate;

//IIC 状态转换控制
always @(cstate or dac_en or scl_high or scl_low or bcnt) begin
    case(cstate)
        IDLE: if(dac_en) nstate <= START;
              else nstate <= IDLE;
        START: if(scl_high) nstate <= ADDR;
              else nstate <= START;
        ADDR: if(scl_low && bcnt == 3'd0) nstate <= ACK1;
              else nstate <= ADDR;
```



```
    ACK1:  if(scl_low) nstate <= CMSB;
           else nstate <= ACK1;
    CMSB:  if(scl_low && bcnt == 3'd0) nstate <= ACK2;
           else nstate <= CMSB;
    ACK2:  if(scl_low) nstate <= LSBI;
           else nstate <= ACK2;
    LSBI:  if(scl_low && bcnt == 3'd0) nstate <= ACK3;
           else nstate <= LSBI;
    ACK3:  if(scl_low) nstate <= ACK4;
           else nstate <= ACK3;
    ACK4:  if(scl_low) nstate <= STOP;
           else nstate <= ACK4;
    STOP:  if(scl_high) nstate <= IDLE;
           else nstate <= STOP;
    default: nstate <= IDLE;
endcase
end

//IIC 输出控制
always @(posedge clk or negedge rst_n)
    if(!rst_n) begin
        sdar <= 1'b1;
        sdlink <= 1'b1; //sda 输出
    end
    else begin
        case(cstate)
            IDLE: begin
                sdar <= 1'b1;
                sdlink <= 1'b1; //sda 输出
            end
            START: if(scl_high) begin
                sdar <= 1'b0; //起始位
                sdlink <= 1'b1; //sda 输出
            end
            ADDR: if(scl_low) begin
                sdar <= DEVICE_ADDR[bcnt]; //依次发送地址 bit7-0
                sdlink <= 1'b1; //sda 输出
            end
        end
    end
```



```
CMSB: if(scl_low) begin
    sdar <= dac_mdata[bcnt];    //送 control/MSB 字节
    sdlink <= 1'b1; //sda 输出
end
LSBI: if(scl_low) begin
    sdar <= dac_ldata[bcnt];    //送 LSB/invalid 字节
    sdlink <= 1'b1; //sda 输出
end
ACK1,ACK2,ACK3: if(scl_low) begin
    sdar <= 1'b0;
    sdlink <= 1'b0; //sda 输入
end
ACK4: if(scl_low) begin
    sdar <= 1'b0;
    sdlink <= 1'b1; //sda 输出
end
STOP: if(scl_high) begin
    sdar <= 1'b1;
    sdlink <= 1'b1; //sda 输出
end
default: ;
endcase
end

assign sda = sdlink ? sdar : 1'bz; //SDA 输入输出控制逻辑

//IIC 传输位控制计数器
always @(posedge clk or negedge rst_n)
    if(!rst_n) bcnt <= 3'd0;
    else begin
        case(cstate)
            ADDR, CMSB, LSBI: begin
                if(scl_low) bcnt <= bcnt-1'b1; //bit7-0
                else ;
            end
            default: bcnt <= 3'd7;
        endcase
    end
end
```




```
endmodule
```

6.7.5 仿真验证

对于本实例的仿真，这里也提几点注意事项和设计关键点。

- 参考 DA 芯片 DAC5571 的 datasheet，把它的 IIC 接口时序弄明白，设计一个 IIC 从机。
- 模拟更改 source 的值，对应查看模拟 IIC 从机接收到的 DAC 数据是否变化。

6.7.6 工程实践

- (1) 新建工程，命名为 **ex9**，把这个工程放在专门的文件夹下，其他设置参考前面章节。
- (2) 新建 Verilog 源文件，命名为 **ex9**，输入前面给出的设计代码。使用 ModelSim-Altera 对设计代码进行仿真验证。
- (3) 综合编译后进行管脚分配，本例程的管脚分配如下所示。

Node Name	Direction	Location	I/O Bank
clk	Input	PIN_22	1
rst_n	Input	PIN_91	6
scl	Output	PIN_106	6
sda	Bidir	PIN_110	7
<<new node>>			

- (4) 参考前面章节，打开 TimeQuest，我们新建一个 SDC 文件，然后对时钟 clk 做约束，约束脚本如下：

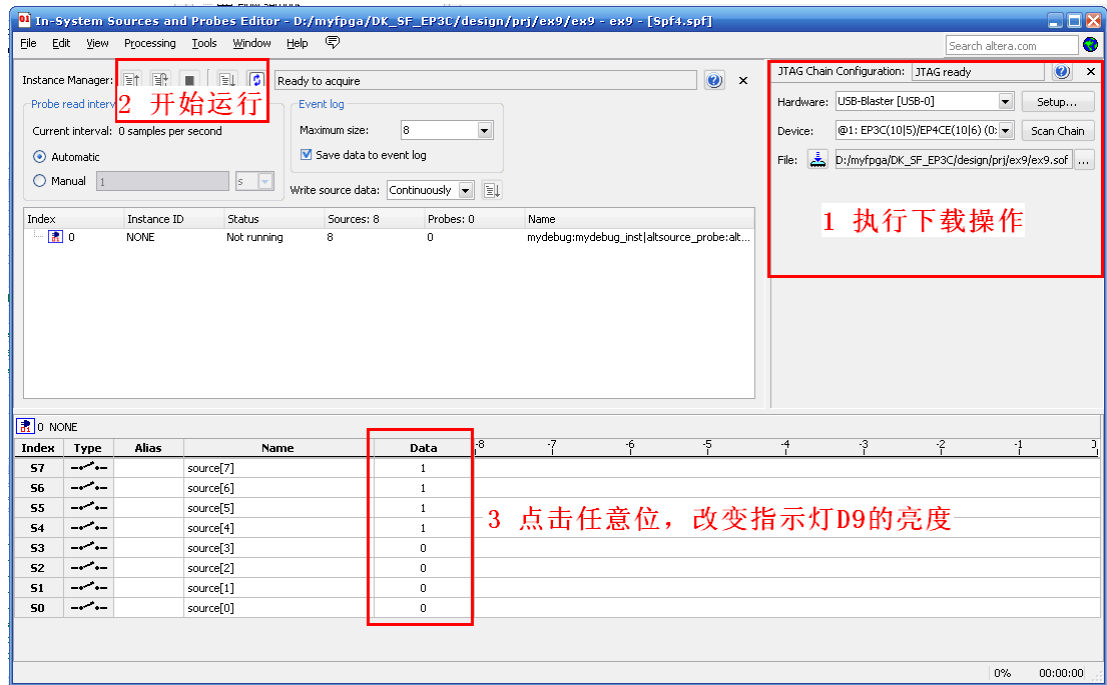
```
create_clock -name {SYSCLK} -period 40.000 -waveform { 0.000 20.000 } [get_ports {clk}]
```

- (5) 我们对工程进行全编译，不仅要让刚刚添加的时钟约束生效，也要生成可以下载到 FPGA 芯片中的配置文件。
- (6) 确认 SF-BASE 板上，跳线帽连接好 P2 的 PIN1 和 PIN2。给板子上电。
- (7) 既可以通过正常的 Programmer 进行下载，也可以通过 In-System Sources and



Probes Editor 调试界面进行下载。

- (8) 接下来使用 In-System Sources and Probes Editor 进行调试，点击菜单栏的 Tools→In-System Sources and Probes Editor。
- (9) 如图所示，执行下面的几步，当我们点击更改任意一个 bit 的 source 值时，SF-BASE 板上的 LED 指示灯 D9 的亮度就会发生变化。



6.8 基于 Qsys 的 NIOS II 实例 4——PIO 中断控制

6.8.1 工程移植

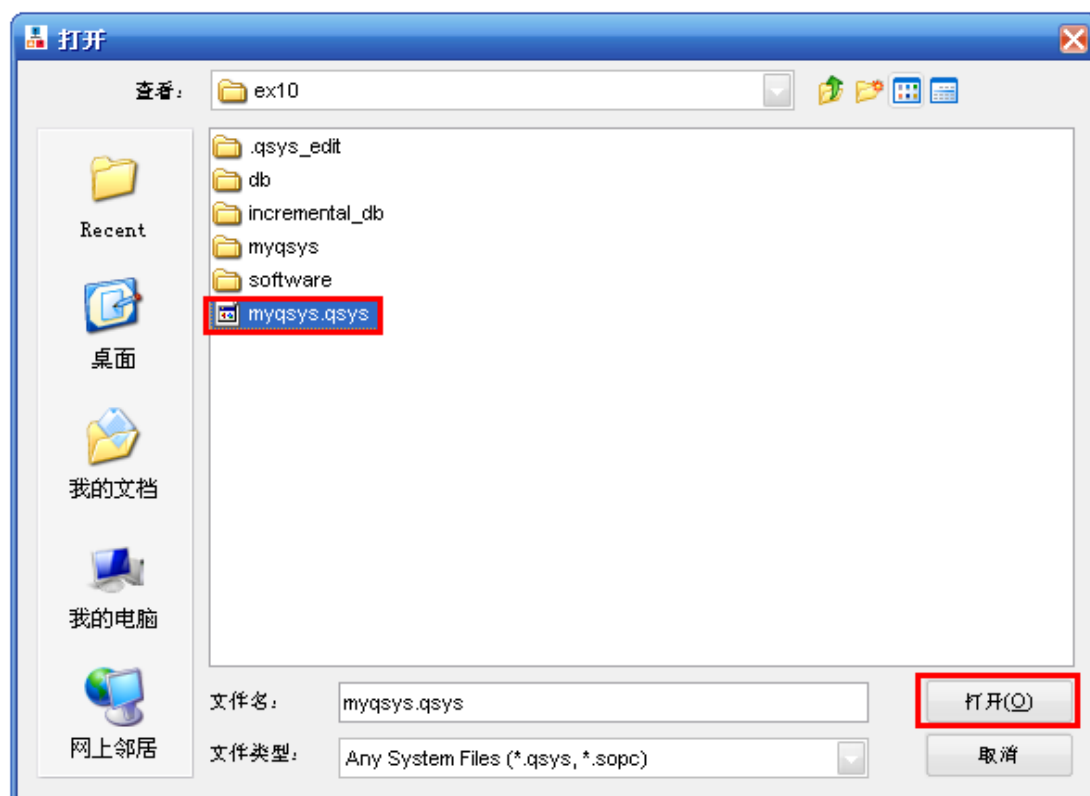
本节实例中，首先复制 ex2 文件夹下的工程，重新命名为 ex10，然后打开工程。在这个工程的基础上，我们会添加一些 PIO 控制的组件，然后创建软件工程进行实验。

本实验所要实现的功能和 6.5 节逻辑（Verilog）实例 5——模式流水灯一样，6.5 节是用逻辑来实验，本节将使用软件编程来实现。

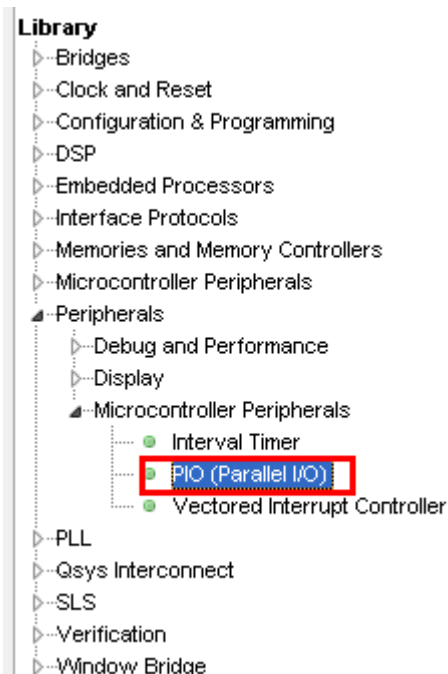


6.8.2 添加组件

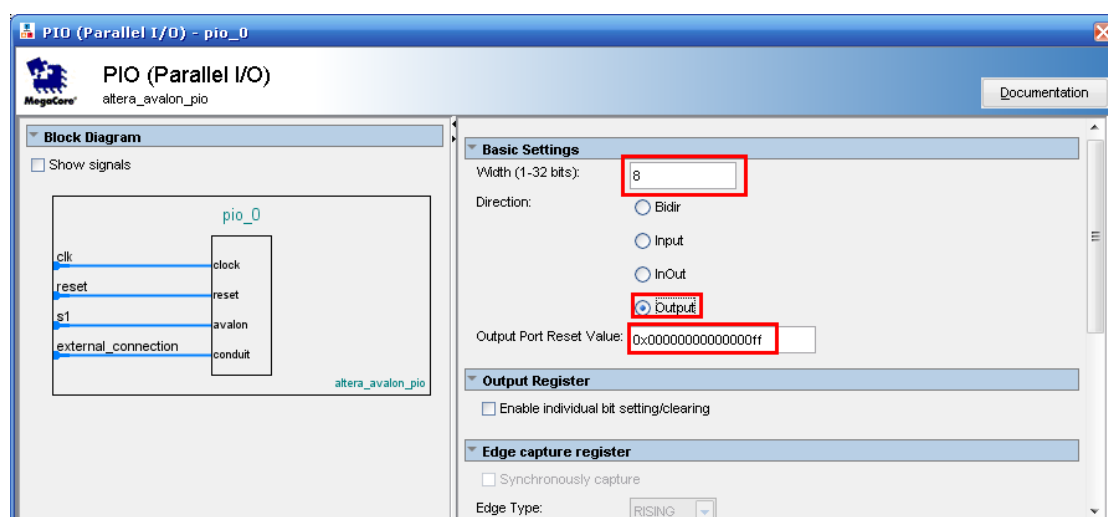
点击 Qsys 图标（什么？别告诉我你不会。算了，咱再提醒一次，在当前工程中，点击 Quartus II 菜单栏的 Tools→Qsys）。在最先弹出的窗口中选择 ex10 工程目录下的 myqsys.qsys 文件，再单击“打开”。此时我们在 ex2 创建的系统原封不动的呈现在眼前，接下来的任务咱们就要在这个既有系统基础上做些添添补补。



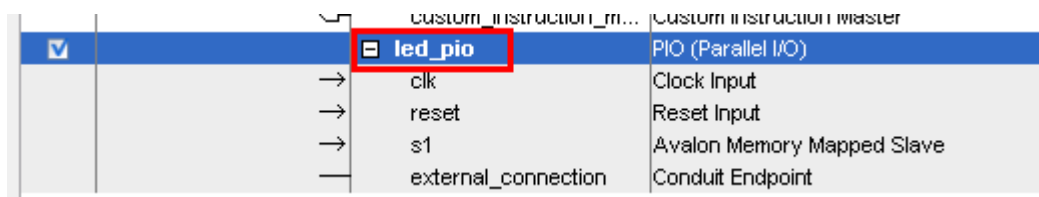
双击 Library 中的 PIO 组件。



这里我们要添加 8 个用于控制 LED 的输出 PIO。设置 Width 为 8, Direction 为 Output, Output Port Reset Value 为 0xff (LED 默认熄灭状态)。完成后点击确认。



更改该组件的名称为 led_pio。



再次双击 Library 中的 PIO 组件。这次我们要添加 3 个用于连接拨码开关的输入 PIO。设置 Width 为 3, Direction 为 Input, 另外, 如图所示, 设置开启这些输入 PIO 的任意边沿捕获中断功能。完成后点击确认。



Basic Settings

Width (1-32 bits): 3

Direction: ☐ Bidir ☒ Input ☐ InOut ☐ Output

Output Port Reset Value: 0x0000000000000000

Output Register

☐ Enable individual bit setting/clearing

Edge capture register

☒ Synchronously capture

Edge Type: ANY

☐ Enable bit-clearing for edge capture register

Interrupt

☒ Generate IRQ

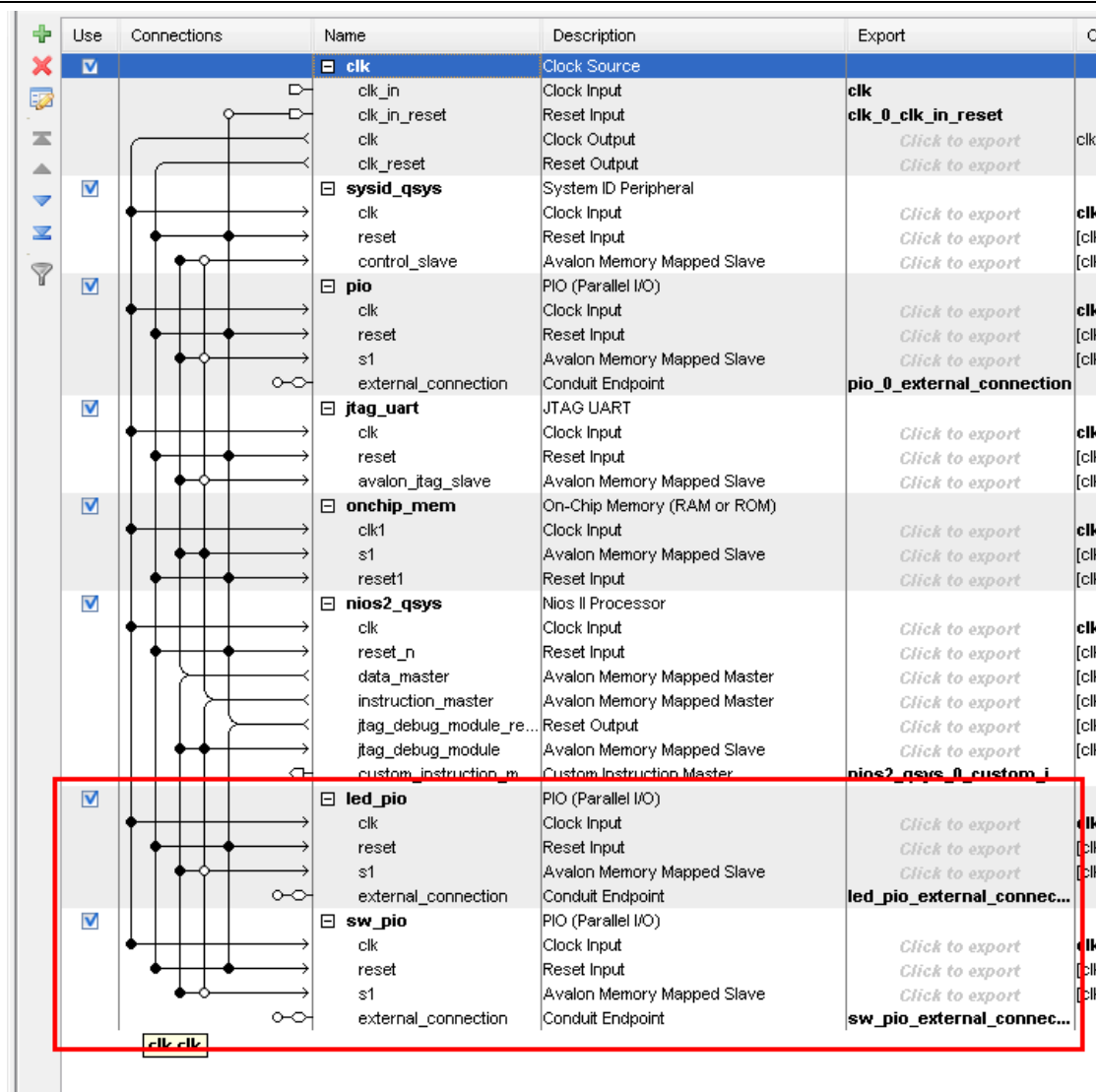
IRQ Type: EDGE

Level: Interrupt CPU when any unmasked I/O pin is logic true
Edge: Interrupt CPU when any unmasked bit in the edge-capture register is logic true. Available when synchronous capture is enabled

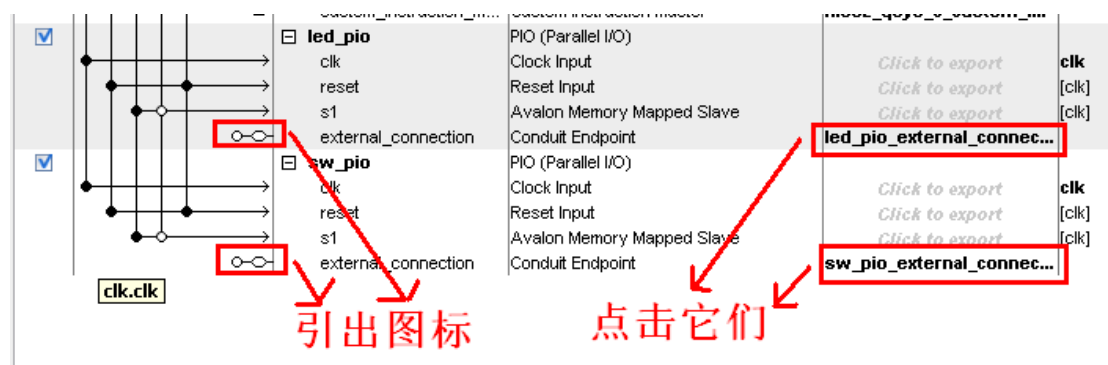
更改该组件的名称为 sw_pio。

external_connection	Conduit Endpoint
<input checked="" type="checkbox"/> sw_pio	PIO (Parallel I/O)
→ clk	Clock Input
→ reset	Reset Input
→ s1	Avalon Memory Mapped Slave
external_connection	Conduit Endpoint

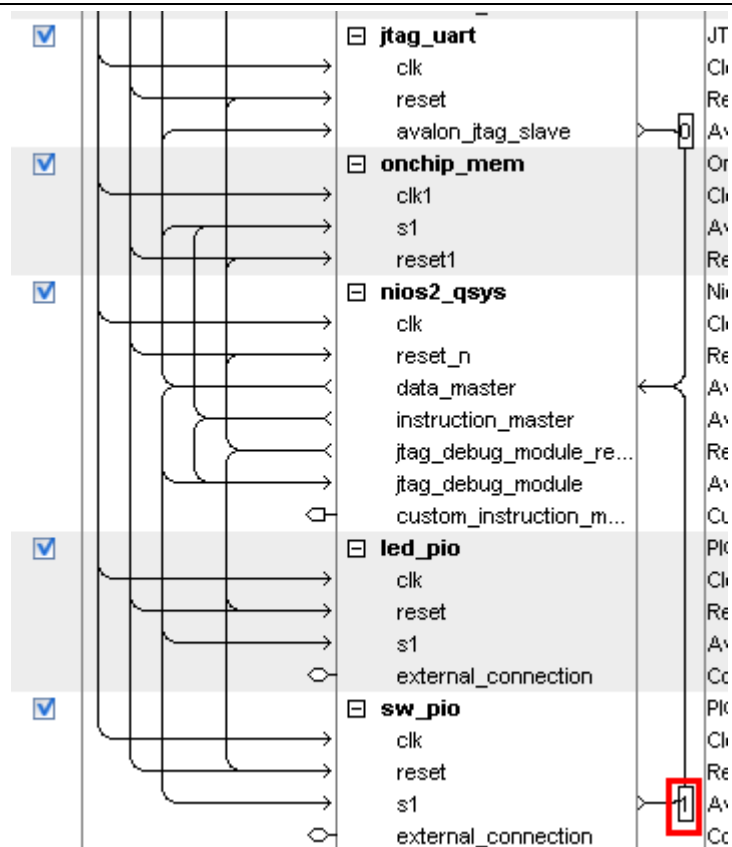
接下来,我们需要在 Connections 中做连接,如图所示。将新添加的两个 PIO 外设的 clk、reset、s1 都分别连接到相应总线上。



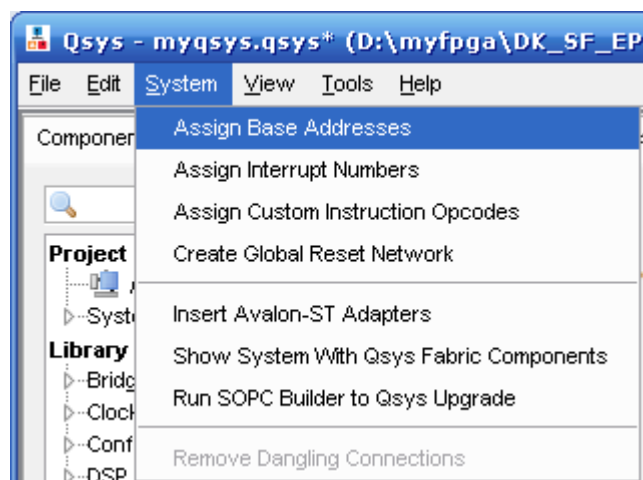
另外对于这两个 PIO 外设的 external_connection，要点击对应行的 Export，然后它们的最前面便出现了相应的引出图标。



找到 sw_pio 的 IRQ 列，在没有点击它之前，是一个空心；点击它，就出现如图所示的数字 1。表示这个外设的中断优先级。



点击菜单栏的 System→Assign Base Addresses, 重新分配系统各个外设的地址。



最后, 我们可以来到 Generation 中点击右下角的 Generate, 重新生成这个新的系统。

6.8.3 例化系统

在 Qsys 的 HDL Example 中, 我们可以 copy 这个系统例化模板, 然后 paste 到工程源码中, 做相应的映射。工程顶层源代码如下。

《圣经》箴言九 11 “敬畏耶和华是智慧的开端, 认识至胜者便是聪明。”



```
Module ex2(
    clk, rst_n, led,
    led_pio, sw_pio
);

input clk;
input rst_n;
output led;
output[7:0] led_pio;
input[2:0] sw_pio;

myqsys u0 (
    .clk_clk                      (clk), //
clk.clk
    .pio_0_external_connection_export (led), //
pio_0_external_connection.export
    .clk_0_clk_in_reset_reset_n      (rst_n), //
clk_0_clk_in_reset.reset_n
    .nios2_qsys_0_custom_instruction_master_readra (    ), //
nios2_qsys_0_custom_instruction_master.readra
    .led_pio_external_connection_export (led_pio), //
led_pio_external_connection.export
    .sw_pio_external_connection_export (sw_pio) //
sw_pio_external_connection.export
);

endmodule
```

综合一下工程，然后分配管脚如下：



Node Name	Direction	Location
clk	Input	PIN_22
led	Output	PIN_28
led_pio[7]	Output	PIN_126
led_pio[6]	Output	PIN_125
led_pio[5]	Output	PIN_124
led_pio[4]	Output	PIN_121
led_pio[3]	Output	PIN_120
led_pio[2]	Output	PIN_119
led_pio[1]	Output	PIN_115
led_pio[0]	Output	PIN_114
rst_n	Input	PIN_91
sw_pio[2]	Input	PIN_105
sw_pio[1]	Input	PIN_104
sw_pio[0]	Input	PIN_103

大家也可以直接将以下的脚本复制到工程文件夹下的 ex2.qsf 文件中。

```
Set_location_assignment PIN_114 -to led_pio[0]
set_location_assignment PIN_126 -to led_pio[7]
set_location_assignment PIN_125 -to led_pio[6]
set_location_assignment PIN_124 -to led_pio[5]
set_location_assignment PIN_121 -to led_pio[4]
set_location_assignment PIN_120 -to led_pio[3]
set_location_assignment PIN_119 -to led_pio[2]
set_location_assignment PIN_115 -to led_pio[1]
set_location_assignment PIN_103 -to sw_pio[0]
set_location_assignment PIN_104 -to sw_pio[1]
set_location_assignment PIN_105 -to sw_pio[2]
```

完成管脚分配后，对整个工程做一次全编译，然后下载 sof 文件到 SF-CY3 开发板中。

6.8.4 时序约束

参考 5.5 节，进入 TimeQuest，点击菜单栏 Netlist→Create Timing Netlist。弹出新建 sdc 文件的对话框后，使用默认设置，点击 OK。

约束全局输入时钟，点击菜单栏 Constraints→Create Clock。设置时钟名称为 EXT_CLK、时钟周期 40ns，然后选择 Targets 即目标管脚为 clk。

接着参考 5.5 节保存当前新建的 sdc 文件，命名为 ex2.sdc。在菜单栏中打开工程目录下的 ex2.sdc，最后重新编译整个工程。完成后将 sof 文件下载到 SF-CY3 目标板中。



6.8.5 软件编程

使用当前的硬件系统, 参考 5.1 节的方法创建一个 Blank 的模板工程, 命名为 `pio_prj`。此外, 打开 BSP Editor, 同样参考 5.1 节对软件工程的代码做裁剪。最后在应用工程中新建一个 C 代码源文件 `main.c`。在 `main.c` 文件中输入以下代码:

```
/*
 * main.c
 *
 * Created on: 2013-2-6
 *      Author: Administrator
 */

#include "alt_types.h"
#include "altera_avalon_pio_regs.h"
#include "sys/alt_irq.h"
#include "system.h"
#include <stdio.h>
#include <unistd.h>
#include <io.h>

void init_sw_pio(void);
void delay(void);

alt_u8 edge_capture_value; //当前拨码开关值

// 拨码开关中断服务程序
static void handle_sw_interrupts(void)
{
    //捕获当前 PIO 值
    edge_capture_value = IORD_ALTERA_AVALON_PIO_DATA(SW_PIO_BASE);
    //清除边沿中断标志位
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(SW_PIO_BASE, 0x00);
}

//主函数
int main(void)
{
```



```
    alt_u8 cnt;

    init_sw_pio();
    while(1)
    {
        if(~edge_capture_value & 0x01) //流水灯从右往左流动
        {
            for(cnt=0;cnt<8;cnt++) //循环 LED 显示
            {
                IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, ~(1 << cnt)); //给 8
个 LED 送数据

                delay();
            }
        }
        else if(~edge_capture_value & 0x02) //流水灯从左往右流动
        {
            for(cnt=8;cnt>0;cnt--) //循环 LED 显示
            {
                IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, ~(1 << (cnt-1))); //
给 8 个 LED 送数据

                delay();
            }
        }
        else if(~edge_capture_value & 0x04) //所有 LED 一起闪烁
        {
            IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, 0xff);
            delay();
            IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, 0x00);
            delay();
        }
    }
    return 0;
}

//GIO 初始化
void init_sw_pio(void)
{
    //初始化拨码开关状态值
```



```
edge_capture_value = 0xff;
//使能 3 个拨码开关中断
IOWR_ALTERA_AVALON_PIO_IRQ_MASK(SW_PIO_BASE, 0xff);
//复位拨码开关边沿状态
IOWR_ALTERA_AVALON_PIO_EDGE_CAP(SW_PIO_BASE, 0x00);
//拨码开关输入 GIO 中断复位申明
alt_irq_register(SW_PIO_IRQ, SW_PIO_BASE, handle_sw_interrupts);
//初始点亮指示灯 D1
IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, 0xfe);
}

//延时函数
void delay(void)
{
    alt_u32 i =0;
    while(i < 100000)
    {
        i++;
    }
}
```

编译代码, 并且运行到目标板中, 拨动拨码开关, 查看流水灯的工作模式是否达到功能要求。

6.9 基于 Qsys 的 NIOS II 实例 5——数码管定时器中断

6.9.1 功能概述

本节在 6.5 节逻辑控制的数码管实验基础上, 将其改为 Qsys 系统组件, 显示值由系统写 Avalon 总线实现。此外, 我们添加了一个定时器 (TIMER) 组件, 软件编程上用定时器中断来触发显示递增的计数值。

关于 Qsys 组件的添加, 大家可以参考 Quartus II Handbook 的“Creating Qsys Components”一章。这里简单的普及一些理论知识。

Qsys 组件其实也相当于添加一个自己定制的 IP 核模块, 定制这样的模块, 通常需要添加以下的模块信息:

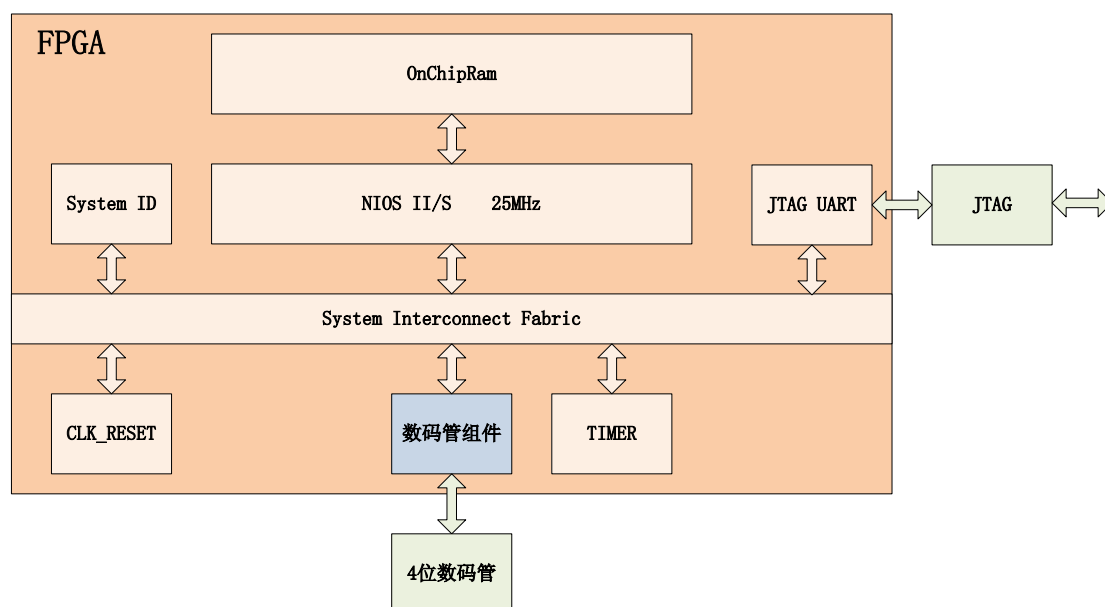
《圣经》箴言九 11 “敬畏耶和华是智慧的开端, 认识至胜者便是聪明。”



- 组件的硬件描述代码，如 Verilog 或 VHDL
- 组件的硬件接口信号的详细描述，如接口名称、I/O 方向等
- 决定组件的不同工作运行模式的可配置参数
- 在 Qsys 系统中可用于被例化的编辑参数
- 集成到 Qsys 系统中需要的其他脚本文件
- 其他相关信息，如软件驱动等

当然了，不要被吓住了，其实上面的部分信息在 Qsys 集成的 Component Editor 中都已经包办了，我们只要根据它给出的一些 GUI 进行选项配置即可。

首先复制上一节的工程(工程目录 ex10)，然后更改文件夹名称为 ex11，接着打开 Quartus II 工程。



6.9.2 组件编辑

本节需要添加两个组件，一个是定时器（TIMER），这个组件是 Qsys 集成的标准 IP 核；另一个是数码管组件，该组件则是我们自定义编辑，在 6.5 节实例代码的基础上改造而成。

我们先来看看数码管组件的修改代码。将 6.5 节的代码复制到 ex11 工程文件夹下，修改文件名为 seg7.v。

```
module seg7(  
    clk, rst_n,
```



```
        seg_db, seg_cs,
        sys_cs_n, sys_wr_n, sys_wrddata
    );
input clk;
input rst_n;
output reg[7:0] seg_db;
output reg[3:0] seg_cs;
    //avalon 总线接口
input sys_cs_n; //总线读片选, 低电平有效
input sys_wr_n; //总线写使能信号, 低电平有效
input[15:0] sys_wrddata; //总线写入数据

//-----
//avalon 从接口逻辑

wire wrcs_n = sys_cs_n | sys_wr_n;

reg[15:0] dis_db; //显示数据寄存器

//总线地址译码锁存
always @(posedge clk or negedge rst_n)
    if(!rst_n) dis_db <= 16'd0;
    else if(!wrcs_n) dis_db <= sys_wrddata;

//-----
    //产生数码管位选定时的时间
reg[7:0] cnt;

always @ (posedge clk or negedge rst_n)
    if(!rst_n) cnt <= 8'd0;
    else cnt <= cnt+1'b1;

//-----
    //采用分时机制, 分别将当前显示的数码管选中并赋值
reg[3:0] cur_disdb;

always @ (posedge clk or negedge rst_n)
    if(!rst_n) cur_disdb <= 4'd0;
```



```
else begin
    case(cnt[7:6])
        2'b00: begin    //个位显示
            seg_cs <= 4'b1110;
            cur_disdb <= dis_db[3:0];
        end
        2'b01: begin    //十位显示
            seg_cs <= 4'b1101;
            cur_disdb <= dis_db[7:4];
        end
        2'b10: begin    //百位显示
            seg_cs <= 4'b1011;
            cur_disdb <= dis_db[11:8];
        end
        2'b11: begin    //千位显示
            seg_cs <= 4'b0111;
            cur_disdb <= dis_db[15:12];
        end
        default: ;
    endcase
end

//-----
//数码管显示 0~F 对应段选输出
parameter    SEG_NUM0    = 8'h3f, //c0,
              SEG_NUM1    = 8'h06, //f9,
              SEG_NUM2    = 8'h5b, //a4,
              SEG_NUM3    = 8'h4f, //b0,
              SEG_NUM4    = 8'h66, //99,
              SEG_NUM5    = 8'h6d, //92,
              SEG_NUM6    = 8'h7d, //82,
              SEG_NUM7    = 8'h07, //F8,
              SEG_NUM8    = 8'h7f, //80,
              SEG_NUM9    = 8'h6f, //90,
              SEG_NUMA    = 8'h77, //88,
              SEG_NUMB    = 8'h7c, //83,
              SEG_NUMC    = 8'h39, //c6,
              SEG_NUMD    = 8'h5e, //a1,
```



```
        SEG_NUME    = 8'h79, //86,
        SEG_NUMF    = 8'h71; //8e;

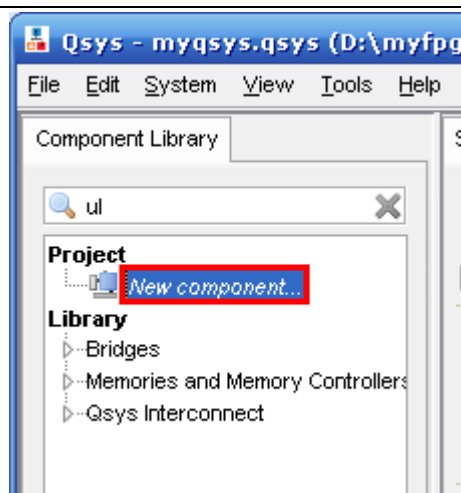
//段选数据译码
always @(cur_disdb) begin
    case(cur_disdb)
        4'h0: seg_db <= SEG_NUM0;
        4'h1: seg_db <= SEG_NUM1;
        4'h2: seg_db <= SEG_NUM2;
        4'h3: seg_db <= SEG_NUM3;
        4'h4: seg_db <= SEG_NUM4;
        4'h5: seg_db <= SEG_NUM5;
        4'h6: seg_db <= SEG_NUM6;
        4'h7: seg_db <= SEG_NUM7;
        4'h8: seg_db <= SEG_NUM8;
        4'h9: seg_db <= SEG_NUM9;
        4'ha: seg_db <= SEG_NUMA;
        4'hb: seg_db <= SEG_NUMB;
        4'hc: seg_db <= SEG_NUMC;
        4'hd: seg_db <= SEG_NUMD;
        4'he: seg_db <= SEG_NUME;
        4'hf: seg_db <= SEG_NUMF;
        default: ;
    endcase
end

endmodule
```

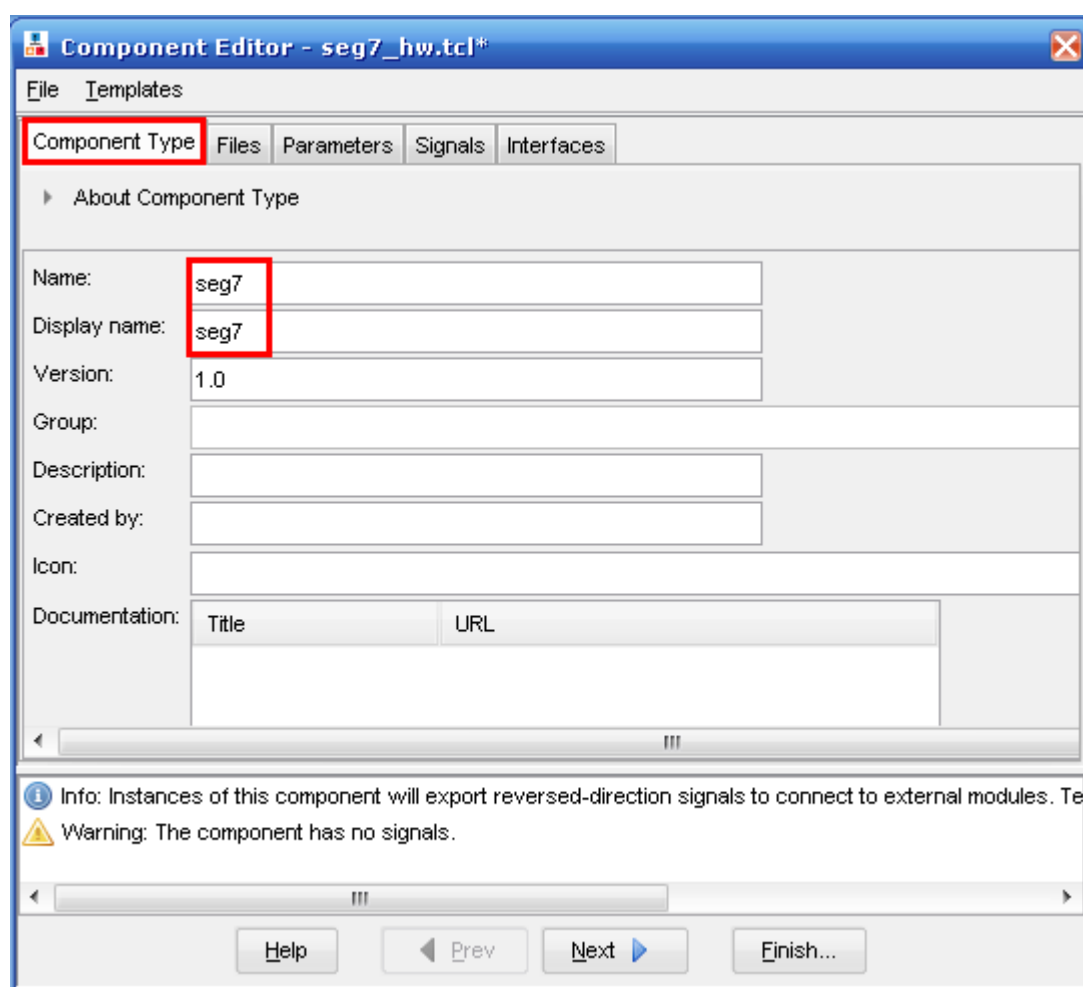
6.9.3 组件添加

打开 ex11 工程目录下的 qsys 系统, 先删除原来的 led_pio 组件。直接选择该组件, 然后点击键盘的 Delete 键即可。

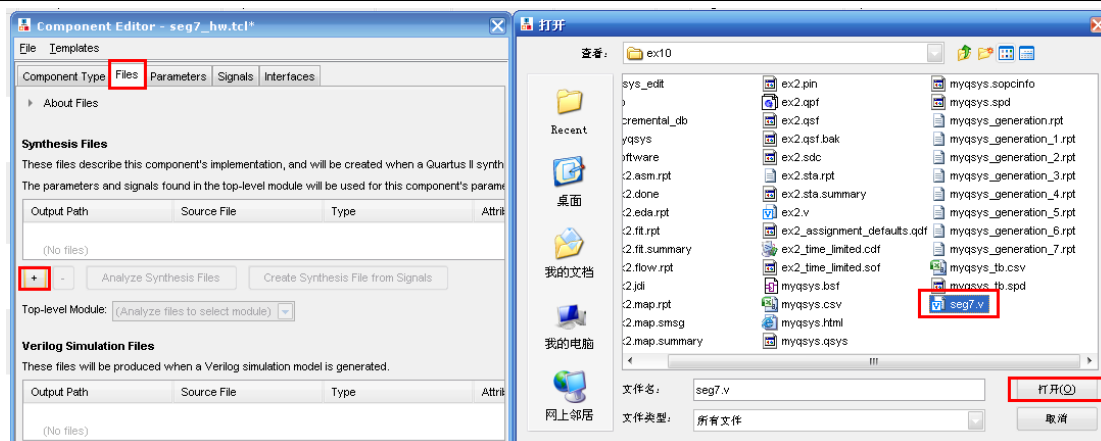
下来要来添加新组件。双击 Component Library 下的 Project→New component。如图所示。



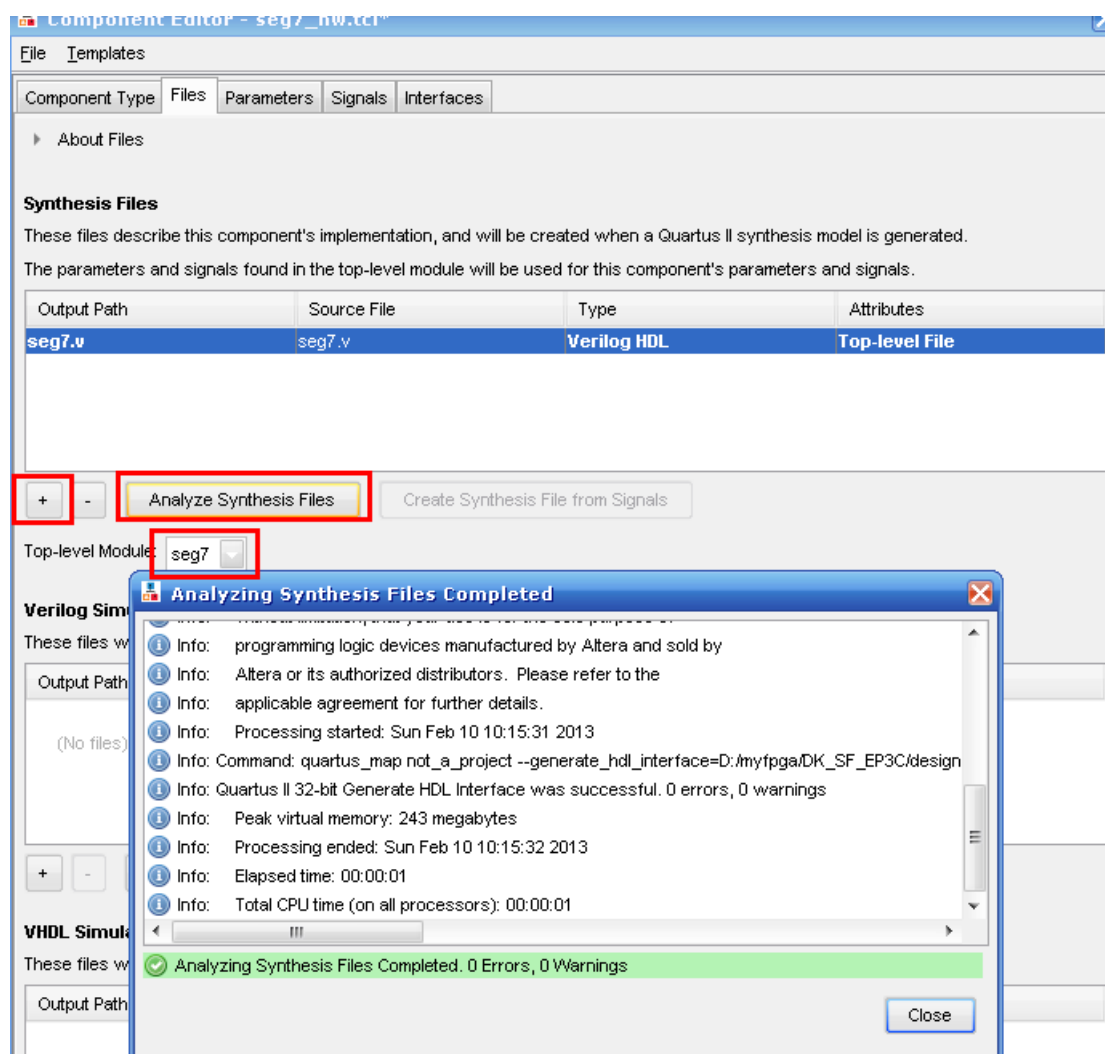
进入 Component Editor 后, 首先在 Component Type 页面, 输入新组件的名称和显示名称, 我们都命名为 seg7, 如图所示。



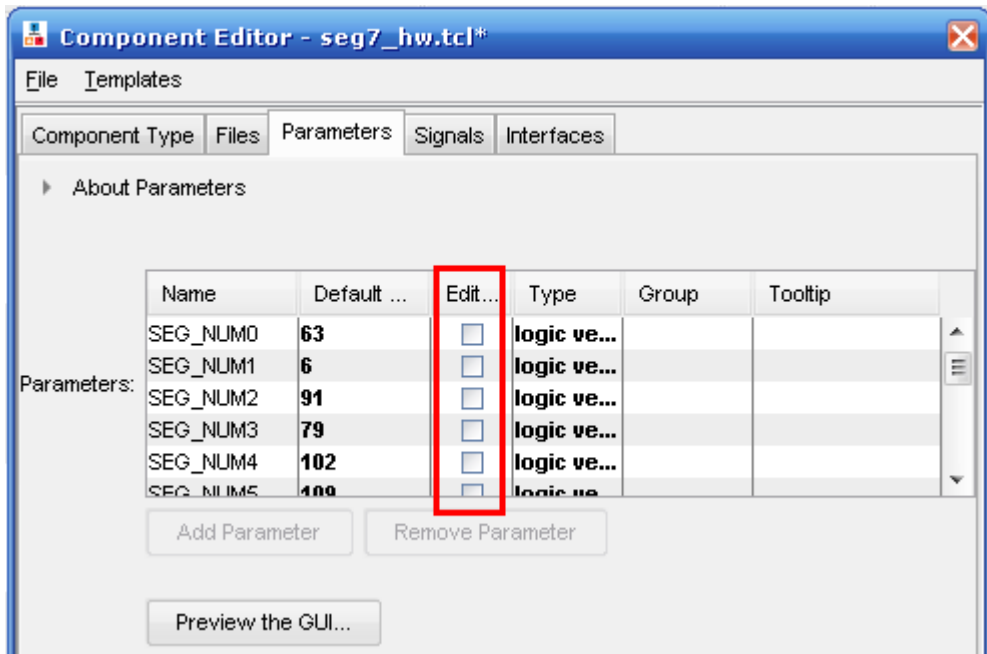
在 Files 页面, 需要添加组件的逻辑代码文件 seg7.v。如图所示, 点击 Synthesis Files 下面的+号, 弹出文件选择窗口, 定位到 ex11 文件夹下, 选择刚刚创建的 seg7.v 文件。



接着点击 Analyze Synthesis Files 进行该逻辑文件的综合, 完成后弹出如图所示信息, 点击 Close 回到主编辑界面。



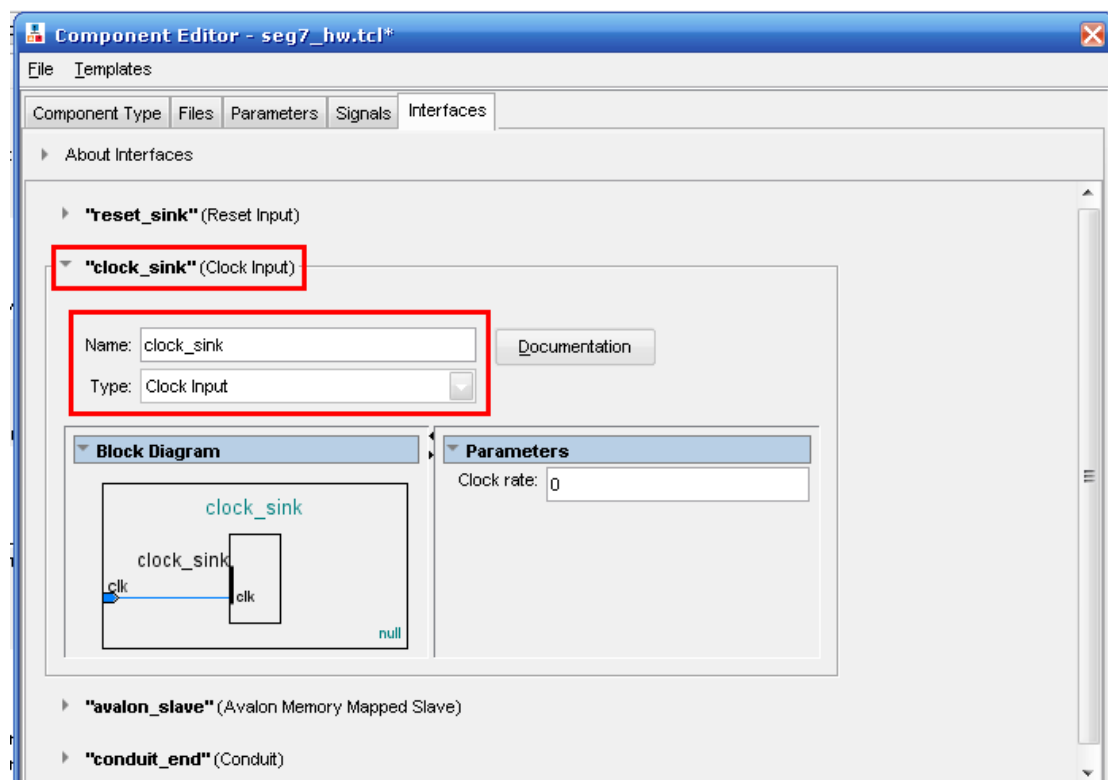
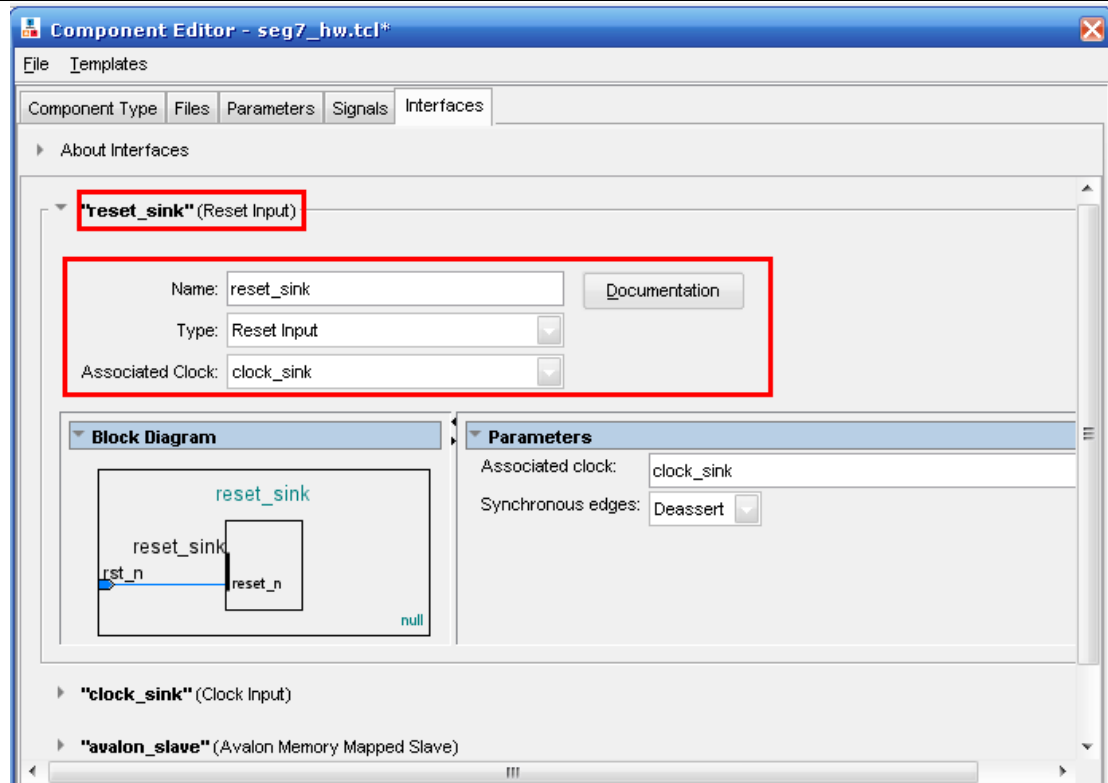
Parameter 页面, 主要是对我们的组件代码中定义的 Parameter 的设置, 我们无需在组件中更改它的值。因此, 如图所示, 我们将 Edit 列的勾号全部取消。

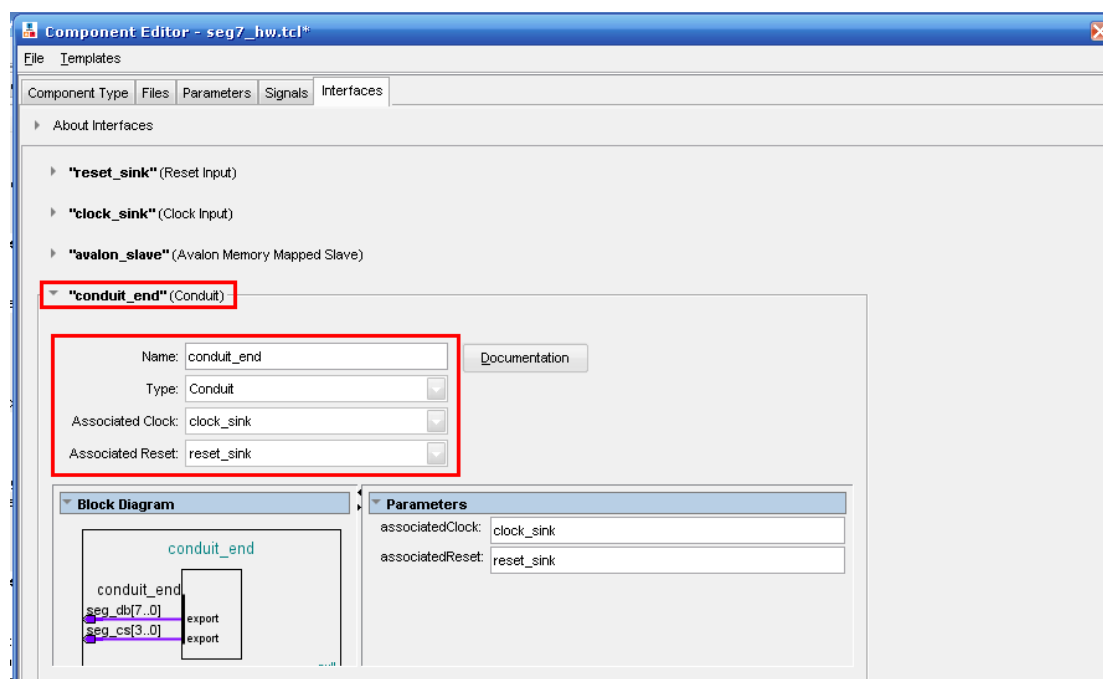
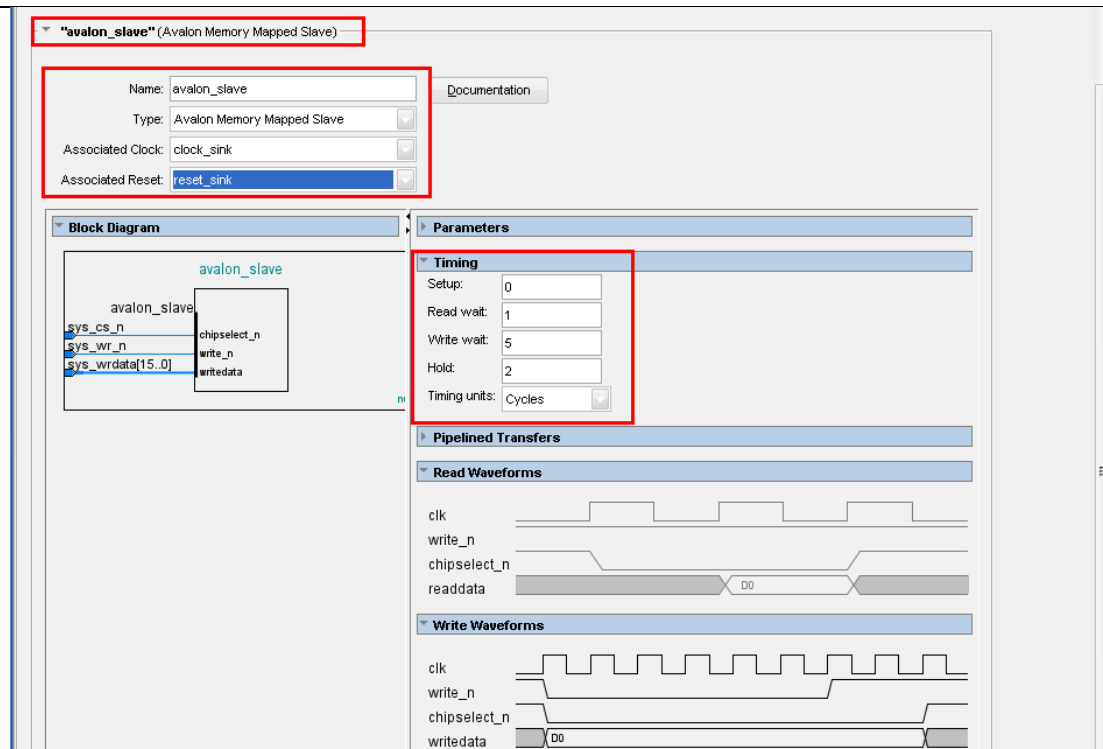


Signals 设置页面如图所示，这里的对外接口可以分为三类，一类是系统控制信号，如时钟 `clk` 和复位 `rst_n`；另一类是与外部器件的接口信号，如连接数码管的 `seg_db`、`seg_cs`，最后一类是 Avalon-MM 接口，如 `sys_cs_n`、`sys_wr_n`、`sys_wrddata`。该页面中的 Interface 和 Signal Type 都需要做如图所示的设置。在 Interface 列中，注意分别先选择对应类别的 new...选项，随后同组接口显示一样的名称即可。

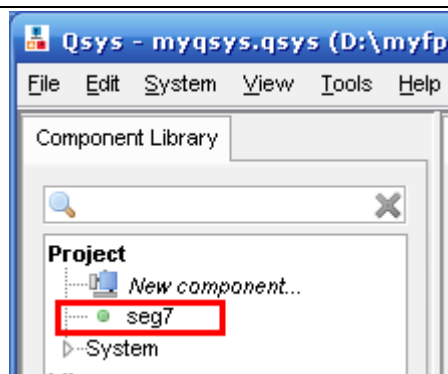
Component Editor - seg7_hw.tcl				
About Signals				
Name	Interface	Signal Type	Width	Direction
clk	clock_sink	clk	1	input
rst_n	reset_sink	reset_n	1	input
seg_db	conduit_end	export	8	output
seg_cs	conduit_end	export	4	output
sys_cs_n	avalon_slave	chipselect_n	1	input
sys_wr_n	avalon_slave	write_n	1	input
sys_wrddata	avalon_slave	writedata	16	input

最后在 Interface 配置页面中，分别作如图所示的设置。

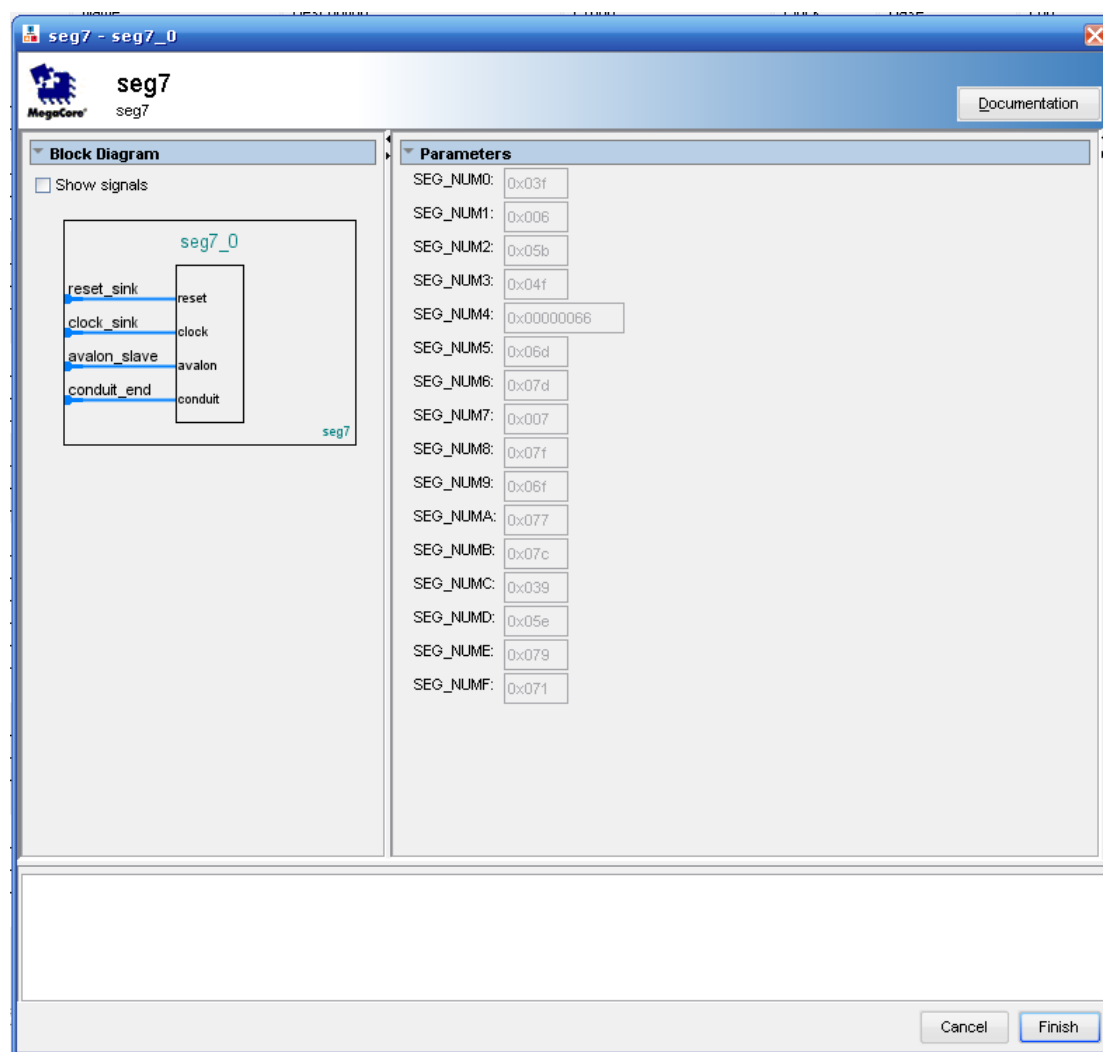




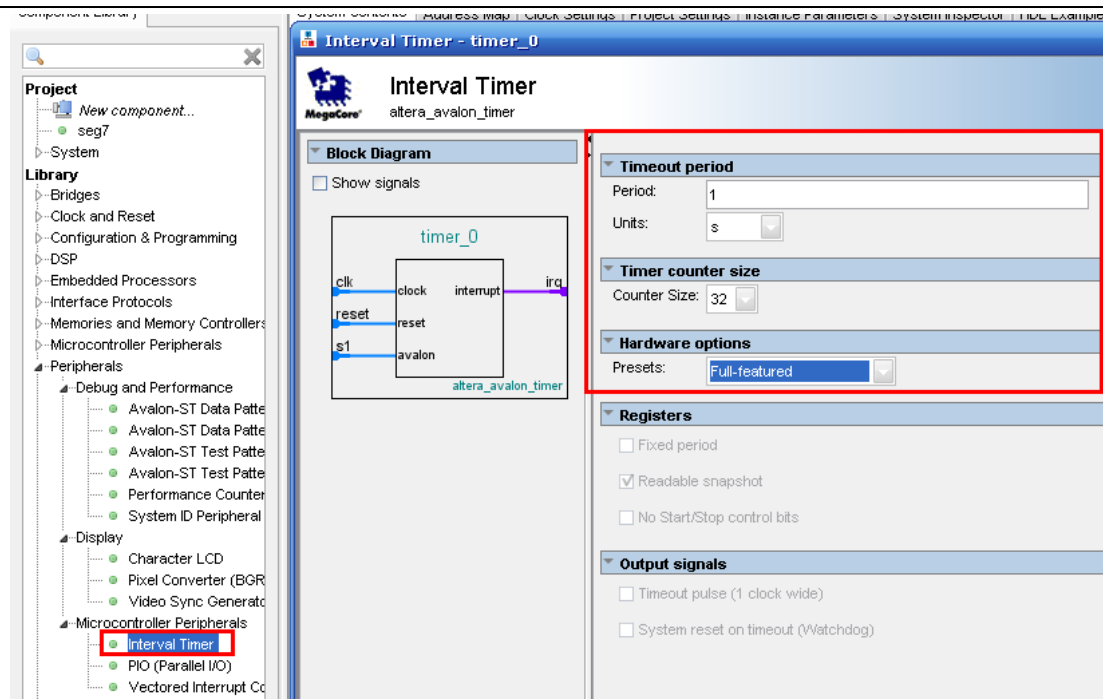
完成前面所有的设置之后, 点击 Finish。此时, 我们看到如图所示, 在 Project 下多了一个 seg7 组件, 这便是我们刚刚编辑添加的。



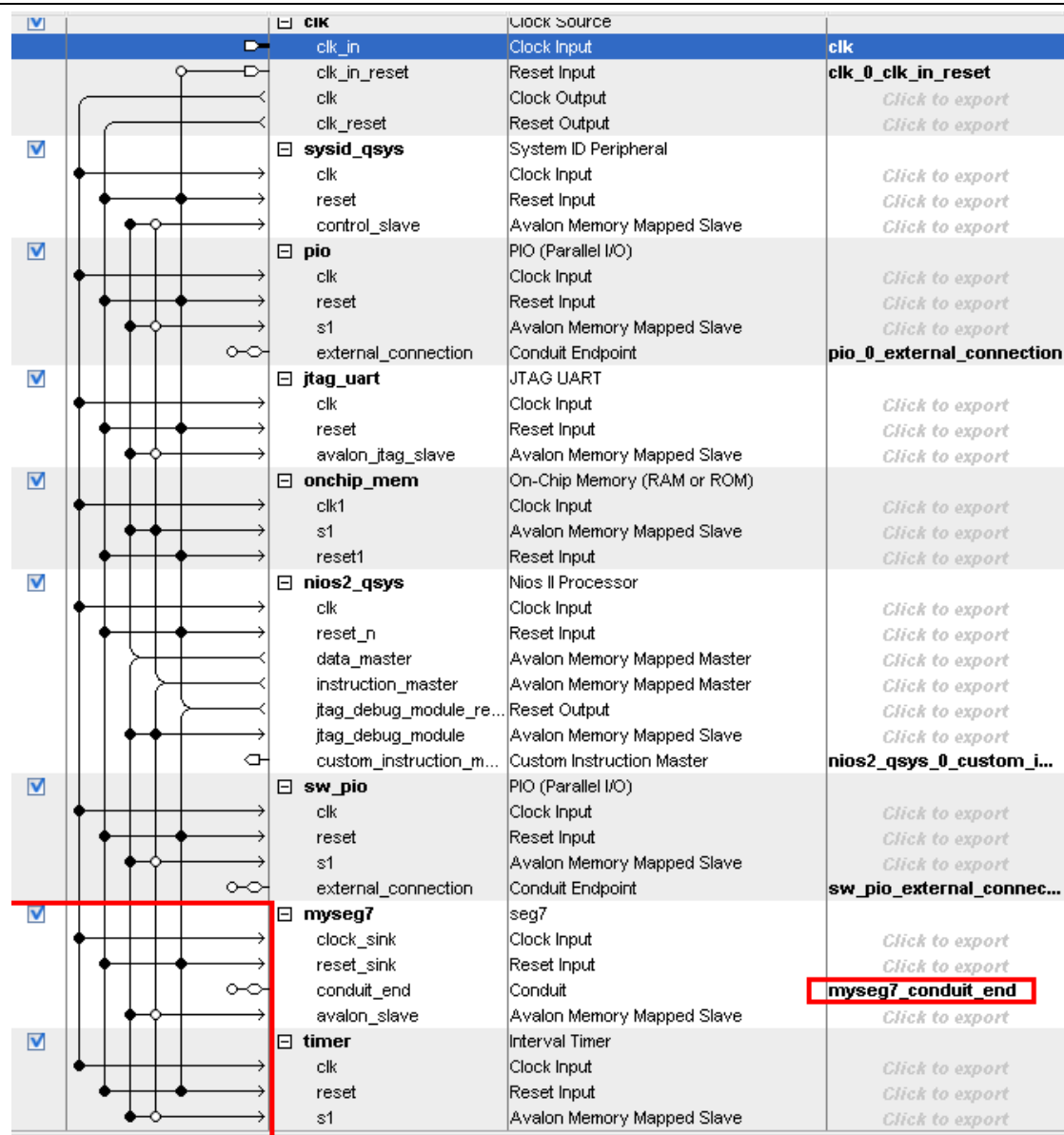
双击 seg7 组件，弹出组件配置界面如图所示，点击 Finish 完成组件添加。



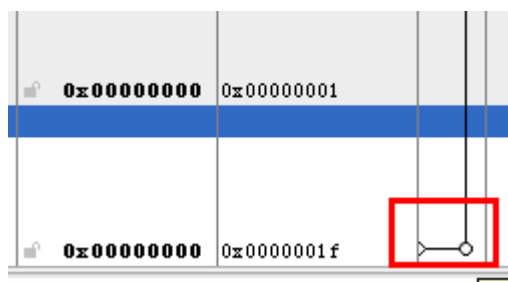
接着我们点击 Interval Timer 添加定时器组件，设置如图所示，1s 的 Full-featured 模式。



接着更改 seg7 组件的名称为 myseg7，更改定时器组件为 timer。然后如图所示，在 Connections 中对他们的接口做好连接，尤其要注意 myseg7 组件的 conduit 信号，需要点击 Export 列的对应信号，然后该信号接口的最前方便出现了外部接口的标记。



最后把鼠标放置到 timer 组件的 IRQ 列中, 出现如图所示的小空心, 点击它后便出现一个数字, 代表这个外设的中断优先级。





clk		
[clock_sink]	0x00000000	0x00000001
[clock_sink]		
clk		
[clk]	0x00000000	0x0000001f
[clk]		

完成组件添加配置后, 点击菜单栏的 **System**→**Assign Base Addresses** 进行地址的重新自动分配。最后来到 **Generation** 页面, 点击右下角的 **Generate** 按键生成新系统。

在 **Generation** 中, 建议关闭 **Simulation** 功能, 因为我们新组件 **seg7** 没有仿真模型, 可能在 **Generation** 时会报错。

System Contents	Address Map	Clock Settings	Project Settings	Instance Parameters	System Inspector	HDL Example	Generation
Simulation							
Create simulation model:	None						
Create testbench Qsys system:	None						
Create testbench simulation model:	Verilog						
Synthesis							
<input checked="" type="checkbox"/> Create HDL design files for synthesis							
<input checked="" type="checkbox"/> Create block symbol file (.bsf)							
Output Directory							
Path:	D:/myfpga/DK_SF_EP3C/design/prj/ex11/myqsys						
Simulation:							
Testbench:							
Synthesis:	D:/myfpga/DK_SF_EP3C/design/prj/ex11/myqsys/synthesis/						

6.9.4 例化系统

在 Qsys 的 HDL Example 中, 我们可以 copy 这个系统例化模板, 然后 paste 到工程源码中, 做相应的映射。工程顶层源代码如下。其中 **LED_PIO** 外设由于和数码管的数据总线管脚复用, 所以将其注释, 将这组管脚用于数码管控制。

```
module ex2(  
    clk, rst_n, led,  
    sw_pio,  
    seg_db, seg_cs  
);  
input clk;
```

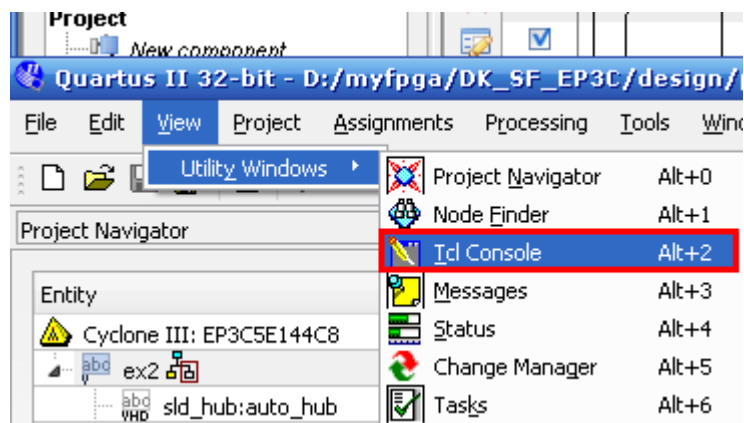


```
input rst_n;
output led;
input[2:0] sw_pio;
output[7:0] seg_db;
output[3:0] seg_cs;

myqsys u0 (
    .clk_clk (clk), //clk.clk
    .pio_0_external_connection_export(led),
        //pio_0_external_connection.export
    .clk_0_clk_in_reset_reset_n(rst_n),
        //clk_0_clk_in_reset.reset_n
    .nios2_qsys_0_custom_instruction_master_readra ( ),
        //nios2_qsys_0_custom_instruction_master.readra
    .sw_pio_external_connection_export(sw_pio),
        //sw_pio_external_connection.export
    .myseg7_conduit_end_db (seg_db), //myseg7_conduit_end.db
    .myseg7_conduit_end_cs(seg_cs) //cs
);

endmodule
```

对于新添加的数码管控制管脚,我们需要做管脚分配。我们不一定非要在 pin assignment 中进行分配,还有一个使用脚本的简单分配方法。打开 Quartus II 的菜单栏的 View→Utility Windows→Tcl Console。



如图所示,在 Tcl Console 中输入以下脚本:

```
set_location_assignment PIN_99 -to seg_cs[3]
set_location_assignment PIN_98 -to seg_cs[2]
```



```
set_location_assignment PIN_87 -to seg_cs[1]
set_location_assignment PIN_100 -to seg_cs[0]
set_location_assignment PIN_120 -to seg_db[7]
set_location_assignment PIN_115 -to seg_db[6]
set_location_assignment PIN_121 -to seg_db[5]
set_location_assignment PIN_126 -to seg_db[4]
set_location_assignment PIN_124 -to seg_db[3]
set_location_assignment PIN_119 -to seg_db[2]
set_location_assignment PIN_114 -to seg_db[1]
set_location_assignment PIN_125 -to seg_db[0]
```

```
Quartus II Tcl Console
tcl> set_location_assignment PIN_99 -to seg_cs[3]
tcl> set_location_assignment PIN_98 -to seg_cs[2]
tcl> set_location_assignment PIN_87 -to seg_cs[1]
tcl> set_location_assignment PIN_100 -to seg_cs[0]
tcl> set_location_assignment PIN_120 -to seg_db[7]
tcl> set_location_assignment PIN_115 -to seg_db[6]
tcl> set_location_assignment PIN_121 -to seg_db[5]
tcl> set_location_assignment PIN_126 -to seg_db[4]
tcl> set_location_assignment PIN_124 -to seg_db[3]
tcl> set_location_assignment PIN_119 -to seg_db[2]
tcl> set_location_assignment PIN_114 -to seg_db[1]
tcl> set_location_assignment PIN_125 -to seg_db[0]
```

我们可以再看看 Pin Planner 中的管脚分配情况，正如我们脚本所分配的。

Node Name	Direction	Location
clk	Input	PIN_22
led	Output	PIN_28
rst_n	Input	PIN_91
seg_cs[3]	Output	PIN_99
seg_cs[2]	Output	PIN_98
seg_cs[1]	Output	PIN_87
seg_cs[0]	Output	PIN_100
seg_db[7]	Output	PIN_120
seg_db[6]	Output	PIN_115
seg_db[5]	Output	PIN_121
seg_db[4]	Output	PIN_126
seg_db[3]	Output	PIN_124
seg_db[2]	Output	PIN_119
seg_db[1]	Output	PIN_114
seg_db[0]	Output	PIN_125
sw_pio[2]	Input	PIN_105
sw_pio[1]	Input	PIN_104
sw_pio[0]	Input	PIN_103
<<new node>>		

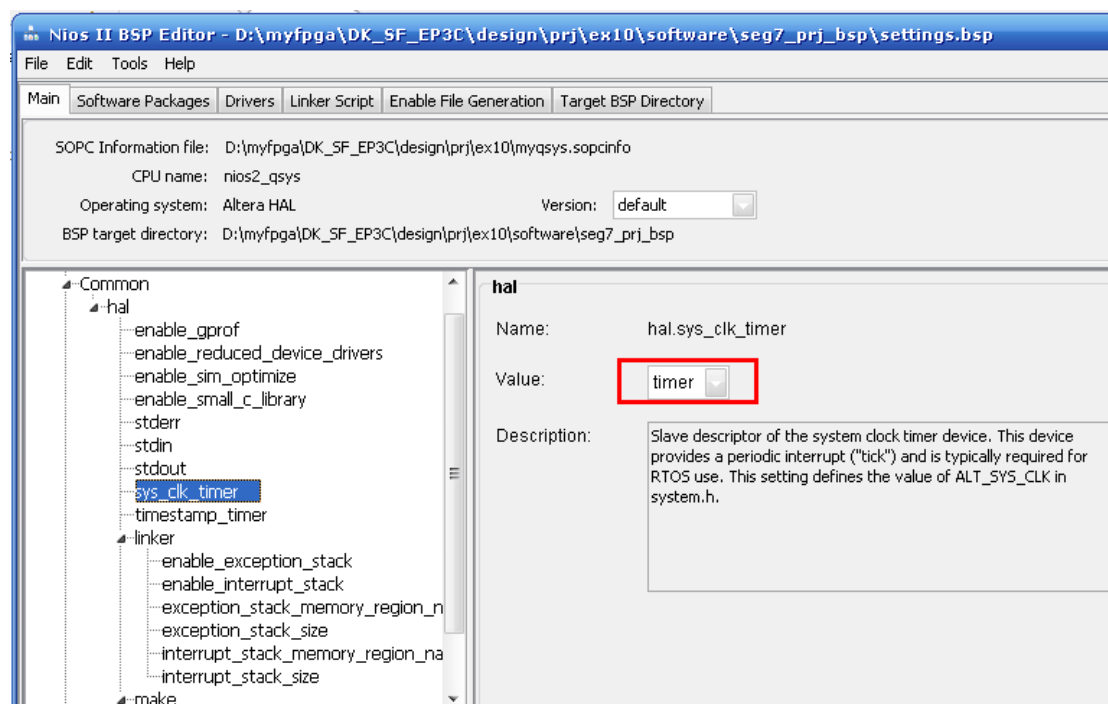
由于之前工程已经对时钟做过约束，外设管脚的频率都比较慢，可以不做约束。所以我
 《圣经》箴言九 11 “敬畏耶和华是智慧的开端，认识至胜者便是聪明。”



们直接编译整个工程即可。完成编译后, 可以先将生成的 `sof` 文件下载到 `SF-CY3` 目标板中。

6.9.5 软件编程

使用当前的硬件系统, 参考 5.1 节的方法创建一个 `Blank` 的模板工程, 命名为 `seg7_prj`。此外, 打开 `BSP Editor`, 同样参考 5.1 节对软件工程的代码做裁剪。在 `BSP` 中, 和之前的设置有一点不同, 即 `sys_clk_timer` 中的 `Value` 必须选择我们新添加的 `timer` 组件, 因为我们的软件实例中必须用到这个组件的功能。



最后在应用工程中新建一个 C 代码源文件 `main.c`。在 `main.c` 文件中输入以下代码:

```
/*
 * main.c
 *
 * Created on: 2013-2-9
 * Author: Administrator
 */

#include "alt_types.h"
#include "altera_avalon_pio_regs.h"
#include "altera_avalon_timer_regs.h"
#include "sys/alt_irq.h"
```



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

void init_timer_pio(void);

alt_u8 flag;    //定时中断标志, 1--产生中断, 还未处理; 0--无中断或已经处理
alt_ul6 second; //秒计数器

//延时函数
void delay(void)
{
    alt_u32 i = 0;
    while(i < 100000)
    {
        i++;
    }
}

//秒定时中断函数
static void handle_timer_interrupts(void)
{
    IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_BASE, 0); //清 TO 标志
    flag = 1;
    second++;
}

//主函数
int main(void)
{
    init_timer_pio();

    while(1)
    {
        if(flag)
        {
            flag = 0;
        }
    }
}
```

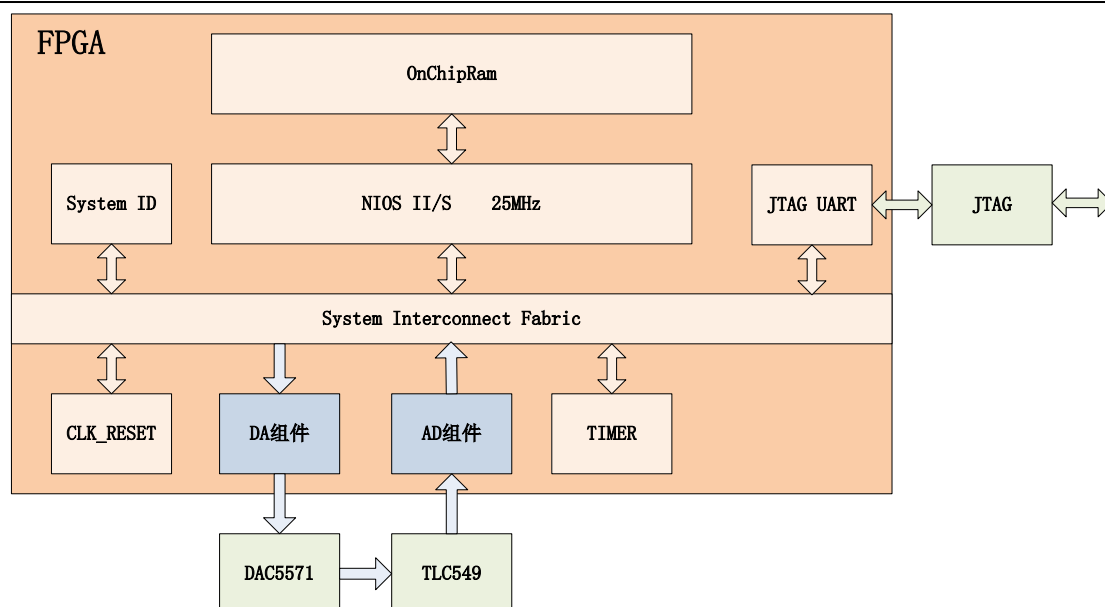


```
IOWR_16DIRECT(MYSEG7_BASE, 0, second);  
  
    }  
}  
return 0;  
}  
  
//定时器 Timer 初始化  
void init_timer_pio(void)  
{  
    //注册定时器中断函数  
    alt_irq_register(TIMER_IRQ, TIMER_BASE, handle_timer_interrupts);  
    //启动 timer 允许中断, 连续计数  
    IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE, 7);  
    //清除标志位  
    flag = 0;  
}
```

6.10 基于 Qsys 的 NIOS II 实例 6——AD/DA 组件

6.9.1 功能概述

前面章节中我们分别实现了 AD 和 DA 芯片的控制, 本节我们首先要把这两个独立的组件分别做成 Avalon 总线可访问的从机, 然后将他们添加到 Qsys 系统中。接着在软件编程上, 要实现的功能也很简单: DAC 组件不断输出递增的数据, 同时在芯片的输出端就会送出递增的模拟电压值, 而此时由于 DA 芯片的输出端会直接连到 AD 芯片的输入端 (用跳线帽连接 SF-BASE 板插座 P3 的 PIN2 和 PIN3), 因此 DA 芯片的递增模拟电压也会被 ADC 芯片转换为递增的数据传送回 NIOS II 处理器。最终我们通过 JTAG UART 定时打印 AD 采样模拟电压值, 并且这个值会是逐渐的递增。这个 DA/AD 系统的数据流示意如图。



6.9.2 组件编辑

首先复制 ex11 工程的整个文件夹, 改名为 ex12。接着拷贝 ex8 和 ex9 工程目录下的 Verilog 文件到 ex12 文件夹下, 分别将他们重命名为 adc_ctrl.v (ex8) 和 dac_ctrl.v (ex9)。然后对他们进行编辑, 添加对他们控制的 avalon 总线逻辑。代码如下。

```
module adc_ctrl(  
    clk, rst_n,  
    adc_data, adc_cs_n, adc_clk,  
    sys_cs_n, sys_rd_n, sys_rddata  
);  
input clk;        //25MHz  
input rst_n;      //低电平复位信号  
input adc_data;    //TLC549 数据信号  
output adc_cs_n;   //TLC549 片选信号, 低电平有效  
output adc_clk;    //TLC549 时钟信号  
  
input sys_cs_n;    //总线读片选, 低电平有效  
input sys_rd_n;    //总线读使能信号, 低电平有效  
output[7:0] sys_rddata; //总线读取数据  
  
//-----  
//定时计数逻辑
```



```
reg[5:0] cntus; //2us 计数器

always @(posedge clk or negedge rst_n)
    if(!rst_n) cntus <= 6'd0;
    else if((cntus < 6'd49) && (cstate != IDLE)) cntus <= cntus+1'b1;
    else cntus <= 6'd0;

wire dchag_flag = (cntus == 6'd0); //ADC 时钟下降沿标志位, 高有效一个时钟周期
wire dlock_flag = (cntus == 6'd24); //ADC 时钟上升沿标志位, 高有效一个时钟周期

//-----
//ADC 工作状态机
parameter IDLE = 3'd0,
           TSUDL = 3'd1,
           START = 3'd2,
           DTRAN = 3'd3,
           STOP = 3'd4,
           TWHDL = 3'd5;

reg[2:0] bitnum; //采样数据位寄存器 7-0
reg[4:0] d17uscnt; //Twh 延时计数器
reg[7:0] adc_dinr; //模数转换数据寄存器
reg[7:0] adc_dinlock; //模数转换数据寄存器, 实时锁存
reg[2:0] cstate, nstate; //状态寄存器

//状态迁移
always @(posedge clk or negedge rst_n)
    if(!rst_n) cstate <= IDLE;
    else cstate <= nstate;

//数据采集位寄存器控制
always @(posedge clk or negedge rst_n)
    if(!rst_n) bitnum <= 3'd0;
    else if(nstate == IDLE) bitnum <= 3'd7;
    else if((nstate == DTRAN) && dlock_flag) bitnum <= bitnum-1'b1;

//Twh 延时计数器控制
always @(posedge clk or negedge rst_n)
```




```
    if(!rst_n) d17uscnt <= 5'd0;
    else if((nstate == TWHDL) && dchag_flag) d17uscnt <= d17uscnt+1'b1;
    else if(nstate == IDLE) d17uscnt <= 5'd0;

    //状态控制
always @(cstate or dchag_flag or bitnum or d17uscnt)
    case(cstate)
        IDLE:  nstate <= TSUDL;
        TSUDL: if(dchag_flag) nstate <= START;
                else nstate <= TSUDL;
        START: if(dchag_flag) nstate <= DTRAN;
                else nstate <= START;
        DTRAN: if(dchag_flag && (bitnum == 3'd7)) nstate <= STOP;
                else nstate <= DTRAN;
        STOP:  if(dchag_flag) nstate <= TWHDL;
                else nstate <= STOP;
        TWHDL: if(dchag_flag && (d17uscnt == 5'd18)) nstate <= IDLE;
                else nstate <= TWHDL;
        default: nstate <= IDLE;
    endcase

    //数据锁存
always @(posedge clk or negedge rst_n)
    if(!rst_n) adc_dinlock <= 8'h00;
    else if((nstate == DTRAN) && dlock_flag) adc_dinlock[bitnum] <= adc_data;
    //数据锁存

always @(posedge clk or negedge rst_n)
    if(!rst_n) adc_dinr <= 8'h00;
    else if((nstate == STOP) && sys_rdcn) adc_dinr <= adc_dinlock;

assign adc_cs_n = ~((cstate == DTRAN) | (cstate == START) | (cstate == TSUDL));

//-----
//时钟速率控制, 1MHz
reg adc_clkr; //TLC549 时钟信号寄存器
```



```
always @(posedge clk or negedge rst_n)
    if(!rst_n) adc_clkr <= 1'b0;
    else if((nstate == DTRAN) && (cntus > 5'd12)) adc_clkr <= 1'b1;
    else adc_clkr <= 1'b0;

assign adc_clk = adc_clkr;

//-----
//Avalon-MM 接口逻辑
wire sys_rdcn_n = sys_cs_n | sys_rdn_n;

reg sys_dlink; //数据输出控制, 高电平有效

always @(posedge clk or negedge rst_n)
    if(!rst_n) sys_dlink <= 1'b1;
    else sys_dlink <= sys_rdcn_n; //锁存读片选译码

assign sys_rddata = sys_dlink ? 8'hzz:adc_dinr;

endmodule
```

```
module dac_ctrl(
    clk, rst_n,
    scl, sda,
    sys_cs_n, sys_wr_n, sys_wrdata
);

input clk; //25MHz
input rst_n; //低电平复位信号

output scl; //DAC5571 数据信号
inout sda; //DAC5571 时钟信号

input sys_cs_n; //总线读片选, 低电平有效
input sys_wr_n; //总线写使能信号, 低电平有效
input[7:0] sys_wrdata; //总线写入数据

//-----
```



```
//avalon 从接口逻辑
reg[7:0] dac_data; //数模转换数据寄存器 0-256

wire wrcs_n = sys_cs_n | sys_wr_n;
wire dac_en = ~wrcs_n; //DAC 输出使能, 高电平有效

//总线地址译码锁存
always @(posedge clk or negedge rst_n)
    if(!rst_n) dac_data <= 8'd0;
    else if(!wrcs_n) dac_data <= sys_wrddata;

//-----
//IIC 的数据速率控制, 100kbps
reg[8:0] cnti; //分频计数寄存器, 0-499, 25M/500=50K

//计数逻辑
always @(posedge clk or negedge rst_n)
    if(!rst_n) cnti <= 9'd0;
    else if(cnti < 9'd499 && cstate != IDLE) cnti <= cnti + 1'b1;
    else cnti <= 9'd0;

wire scl_low = (cnti == 9'd374); //scl 的低电平中间点, 即 sda 的最佳数据更新点
wire scl_high = (cnti == 9'd124); //scl 的高电平中间点, 用于产生起始位或停止位

//scl 输出控制
assign scl = ~cnti[8]; //IIC 时钟信号

//-----
//DAC 的 IIC 总线传输控制
//状态参数定义
parameter IDLE      = 4'd0;
parameter START     = 4'd1;
parameter ADDR      = 4'd2;
parameter ACK1      = 4'd3;
parameter CMSB      = 4'd4;
parameter ACK2      = 4'd5;
parameter LSBI      = 4'd6;
parameter ACK3      = 4'd7;
```



```
parameter ACK4      = 4'd8;
parameter STOP      = 4'd9;

parameter DEVICE_ADDR = 8'b1001_1000; //参考 DAC5571.pdf 的 P15
wire[7:0] dac_mdata = {4'b0000, dac_data[7:4]}; //control/MSB 字节
wire[7:0] dac_ldata = {dac_data[3:0], 4'b0000}; //LSB/invalid 字节

reg[3:0] cstate, nstate; //状态寄存器
reg sdar; //IIC 数据信号寄存器
reg[2:0] bcnt; //传输位计数器 0-7
reg sdlink; //sda 数据方向控制位, 1--输出, 0--输入

always @(posedge clk or negedge rst_n)
    if(!rst_n) cstate <= IDLE;
    else cstate <= nstate;

//IIC 状态转换控制
always @(cstate or dac_en or scl_high or scl_low or bcnt) begin
    case(cstate)
        IDLE: if(dac_en) nstate <= START;
              else nstate <= IDLE;
        START: if(scl_high) nstate <= ADDR;
              else nstate <= START;
        ADDR: if(scl_low && bcnt == 3'd0) nstate <= ACK1;
              else nstate <= ADDR;
        ACK1: if(scl_low) nstate <= CMSB;
              else nstate <= ACK1;
        CMSB: if(scl_low && bcnt == 3'd0) nstate <= ACK2;
              else nstate <= CMSB;
        ACK2: if(scl_low) nstate <= LSBI;
              else nstate <= ACK2;
        LSBI: if(scl_low && bcnt == 3'd0) nstate <= ACK3;
              else nstate <= LSBI;
        ACK3: if(scl_low) nstate <= ACK4;
              else nstate <= ACK3;
        ACK4: if(scl_low) nstate <= STOP;
              else nstate <= ACK4;
        STOP: if(scl_high) nstate <= IDLE;
```



```
        else nstate <= STOP;
    default: nstate <= IDLE;
endcase
end

//IIC 输出控制
always @(posedge clk or negedge rst_n)
    if(!rst_n) begin
        sdar <= 1'b1;
        sdlink <= 1'b1; //sda 输出
    end
    else begin
        case(cstate)
            IDLE: begin
                sdar <= 1'b1;
                sdlink <= 1'b1; //sda 输出
            end
            START: if(scl_high) begin
                sdar <= 1'b0; //起始位
                sdlink <= 1'b1; //sda 输出
            end
            ADDR: if(scl_low) begin
                sdar <= DEVICE_ADDR[bcnt]; //依次发送地址 bit7-0
                sdlink <= 1'b1; //sda 输出
            end
            CMSB: if(scl_low) begin
                sdar <= dac_mdata[bcnt]; //送 control/MSB 字节
                sdlink <= 1'b1; //sda 输出
            end
            LSBI: if(scl_low) begin
                sdar <= dac_ldata[bcnt]; //送 LSB/invalid 字节
                sdlink <= 1'b1; //sda 输出
            end
            ACK1, ACK2, ACK3: if(scl_low) begin
                sdar <= 1'b0;
                sdlink <= 1'b0; //sda 输入
            end
            ACK4: if(scl_low) begin
```



```
        sdar <= 1'b0;
        sdlink <= 1'b1; //sda 输出
    end
    STOP: if(scl_high) begin
        sdar <= 1'b1;
        sdlink <= 1'b1; //sda 输出
    end
    default: ;
    endcase
end

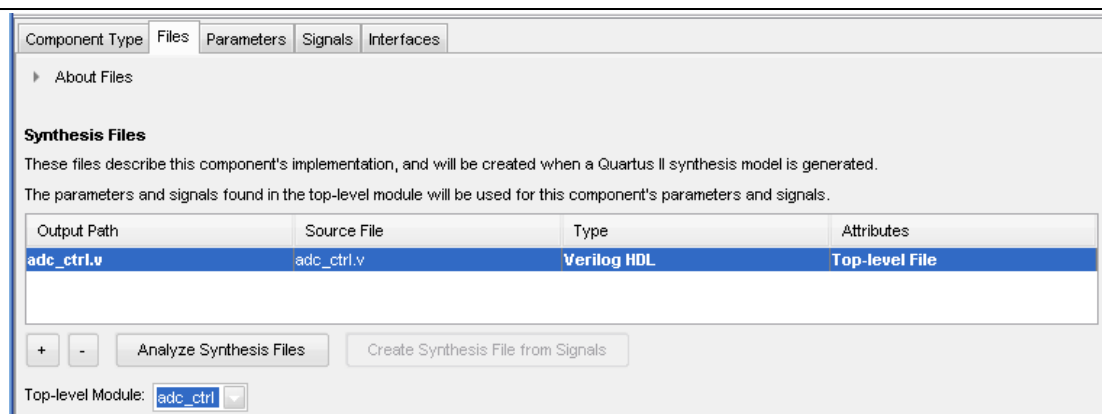
assign sda = sdlink ? sdar : 1'bz; //SDA 输入输出控制逻辑

//IIC 传输位控制计数器
always @(posedge clk or negedge rst_n)
    if(!rst_n) bcnt <= 3'd0;
    else begin
        case(cstate)
            ADDR, CMSB, LSBI: begin
                if(scl_low) bcnt <= bcnt-1'b1; //bit7-0
                else ;
            end
            default: bcnt <= 3'd7;
        endcase
    end

endmodule
```

6.9.3 组件添加

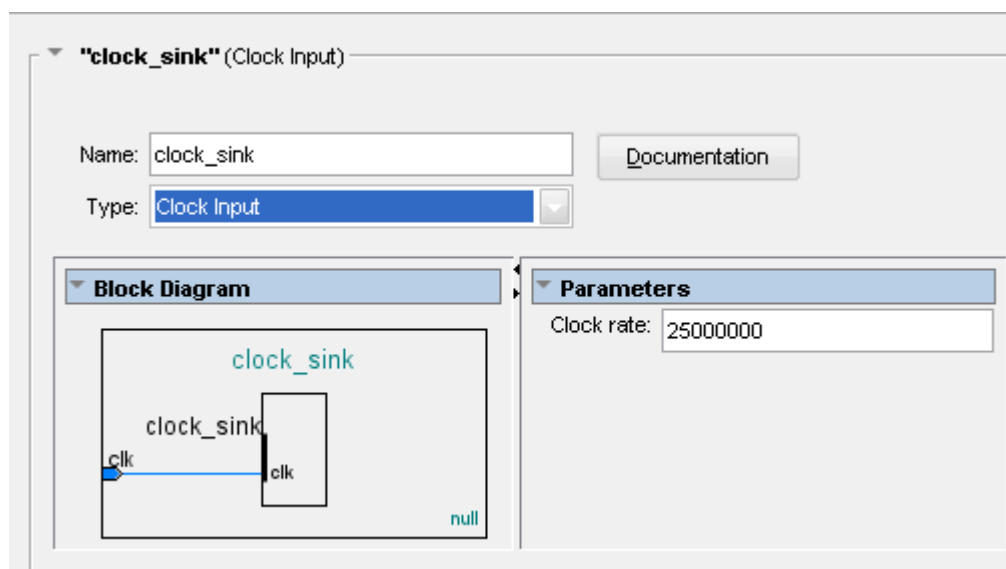
打开 ex12 目录下的 Quartus II 工程, 进入 Qsys 编辑界面。下面我们要分别添加 AD 组件和 DA 组件。如上一节所示, 点击界面左侧的 New Component 按钮, 在 Component Type 编辑页面中, 输入 Name 和 Display name 名称为 adc; 在 Files 页面中, 添加 ex12 目录下的 adc_ctrl.v 文件, 接着点击 Analyze Synthesis Files 按钮, 完成综合无误后, 在 Top-level Module 中选择新添加的文件 adc_ctrl.v, 如图所示。



Parameter 页面中, 同样取消 Edit 列下的所有勾选; Signals 页面中, 配置如图所示。

Component Type Files Parameters Signals Interfaces				
About Signals				
Name	Interface	Signal Type	Width	Direction
clk	clock_sink	clk	1	input
rst_n	reset_sink	reset_n	1	input
adc_data	conduit_end	export	1	input
adc_cs_n	conduit_end	export	1	output
adc_clk	conduit_end	export	1	output
sys_cs_n	avalon_slave	chipselect_n	1	input
sys_rd_n	avalon_slave	read_n	1	input
sys_rddata	avalon_slave	readdata	16	output

进入 Interfaces 页面, 首先可以点击右下角的 Remove Interfaces With No Signals 按钮, 将不使用的接口配置项删除。接着对各个接口分别作如图所示的设置。





"reset_sink" (Reset Input)

Name:

Type:

Associated Clock:

Block Diagram

Parameters

Associated clock:

Synchronous edges:

"conduit_end" (Conduit)

Name:

Type:

Associated Clock:

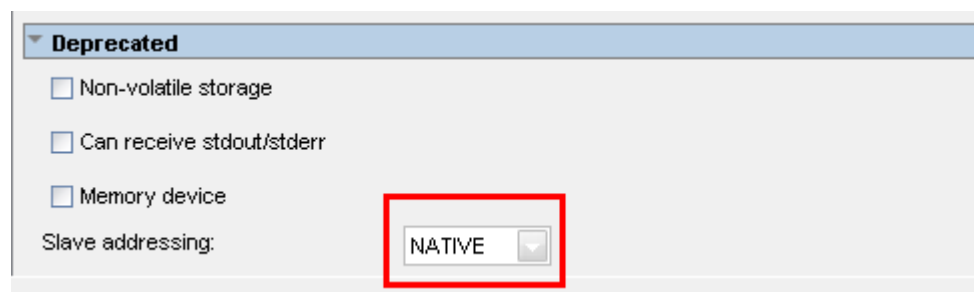
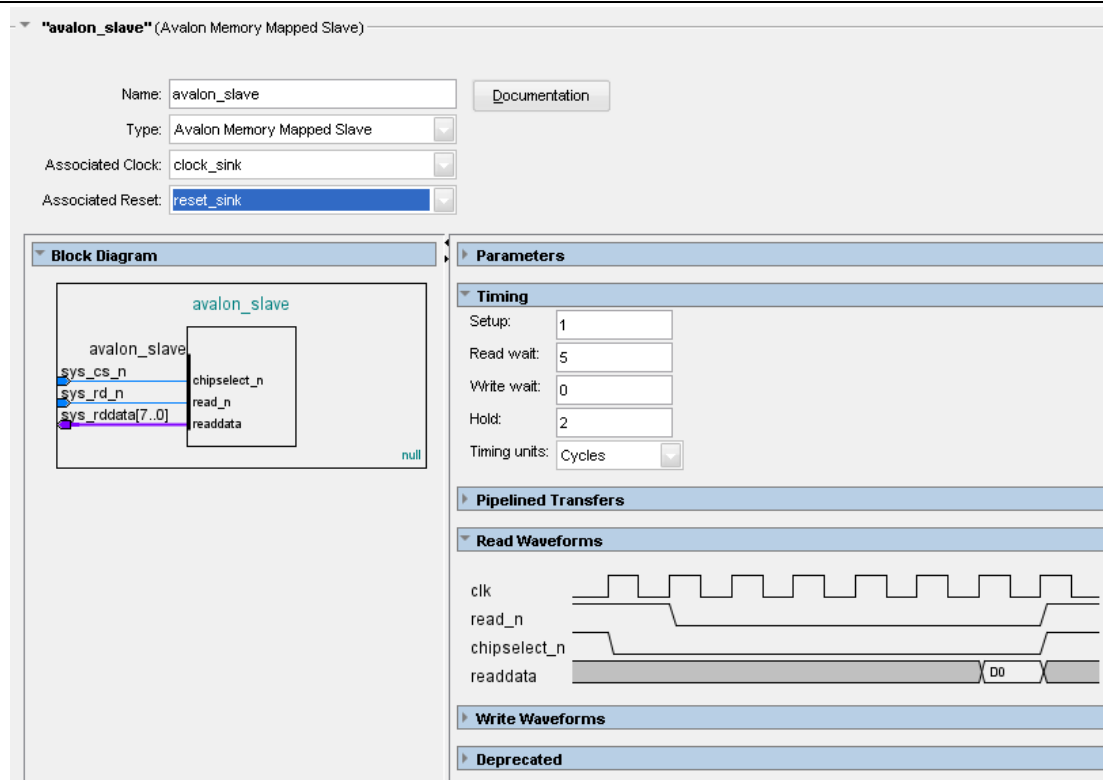
Associated Reset:

Block Diagram

Parameters

associatedClock:

associatedReset:



Avalon_slave 的 Parameters 下图示未展开的配置项可以不用做任何设置，默认即可。

我们再次点击 New Component 创建 dac 组件。在 Component Type 编辑页面中，输入 Name 和 Display name 名称为 dac；在 Files 页面中，添加 ex12 目录下的 dac_ctrl.v 文件，接着点击 Analyze Synthesis Files 按钮，完成综合无误后，在 Top-level Module 中选择新添加的文件 dac_ctrl.v，如图所示。



Component Type Files Parameters Signals Interfaces

► About Files

Synthesis Files

These files describe this component's implementation, and will be created when a Quartus II synthesis model is generated.

The parameters and signals found in the top-level module will be used for this component's parameters and signals.

Output Path	Source File	Type	Attributes
dac_ctrl.v	dac_ctrl.v	Verilog HDL	Top-level File

+ - Analyze Synthesis Files Create Synthesis File from Signals

Top-level Module:

Parameter 页面中, 同样取消 Edit 列下的所有勾选; Signals 页面中, 配置如图所示。

Component Type Files Parameters Signals Interfaces

► About Signals

Name	Interface	Signal Type	Width	Direction
clk	clock_sink	clk	1	input
rst_n	reset_sink	reset_n	1	input
scl	conduit_end	export	1	output
sda	conduit_end	export	1	bidir
sys_cs_n	avalon_slave	chipselect_n	1	input
sys_wr_n	avalon_slave	write_n	1	input
sys_wrdta	avalon_slave	writedata	16	input

进入 Interfaces 页面, 首先可以点击右下角的 Remove Interfaces With No Signals 按钮, 将不使用的接口配置项删除。接着对各个接口分别作如图所示的设置。

▼ "clock_sink" (Clock Input)

Name: Documentation

Type:

Block Diagram

Parameters

Clock rate:



"reset_sink" (Reset Input)

Name: [Documentation](#)

Type:

Associated Clock:

Block Diagram

Parameters

Associated clock:

Synchronous edges:

"conduit_end" (Conduit)

Name: [Documentation](#)

Type:

Associated Clock:

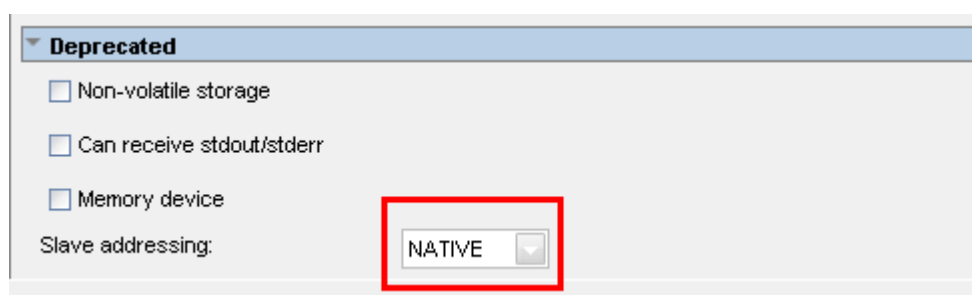
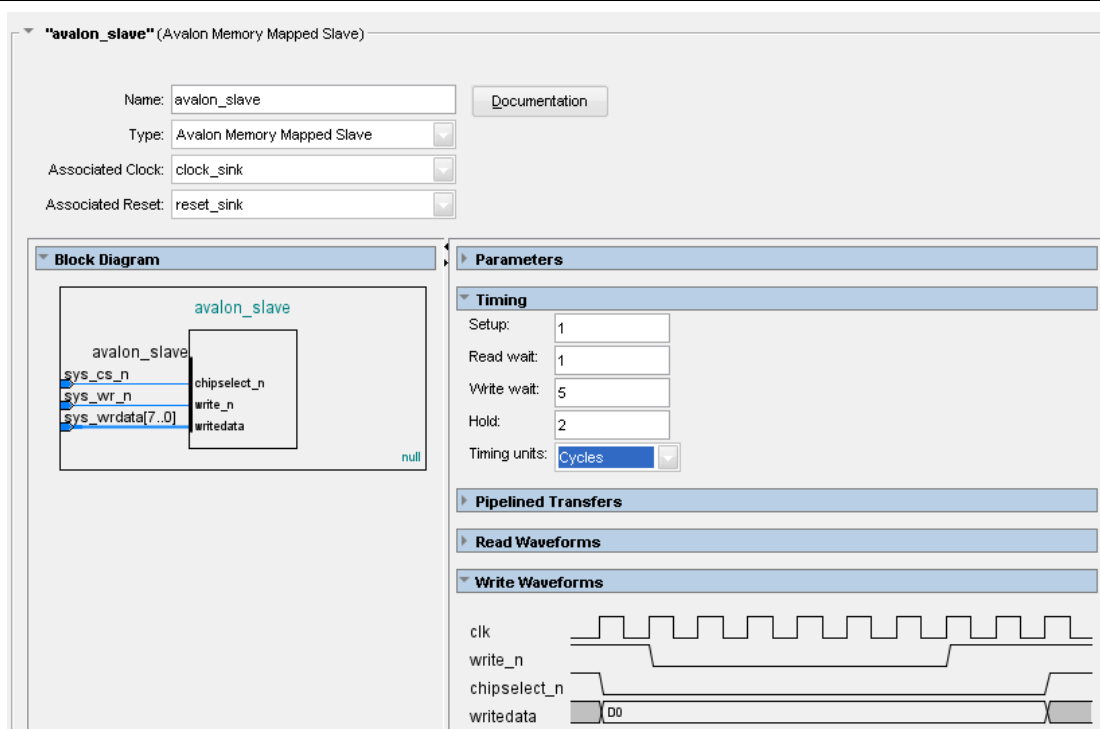
Associated Reset:

Block Diagram

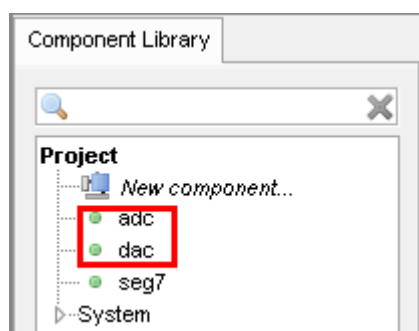
Parameters

associatedClock:

associatedReset:



完成 adc 组件和 dac 组件的创建, 我们可以看到, 在 Component Library 下出现了两个新创建的组件。分别双击添加这两个组件到 Qsys 系统中。



对这两个新添加的组件, 分别重命名为 myadc 和 mydac。同时对他们的 clock、reset 以及 avalon_slave 进行连接, 并且引出的接口 conduit_end 做好设置。如图所示。



Use	Connections	Name	Description
<input checked="" type="checkbox"/>		<div> <div>clk</div> <div> <div>clk_in</div> <div>clk_in_reset</div> <div>clk</div> <div>clk_reset</div> </div> </div>	<div> <div>Clock Source</div> <div>Clock Input</div> <div>Reset Input</div> <div>Clock Output</div> <div>Reset Output</div> </div>
<input checked="" type="checkbox"/>		<div> <div>sysid_qsys</div> <div> <div>clk</div> <div>reset</div> <div>control_slave</div> </div> </div>	<div> <div>System ID Peripheral</div> <div>Clock Input</div> <div>Reset Input</div> <div>Avalon Memory Mapped Slave</div> </div>
<input checked="" type="checkbox"/>		<div> <div>pio</div> <div> <div>clk</div> <div>reset</div> <div>s1</div> <div>external_connection</div> </div> </div>	<div> <div>PIO (Parallel I/O)</div> <div>Clock Input</div> <div>Reset Input</div> <div>Avalon Memory Mapped Slave</div> <div>Conduit Endpoint</div> </div>
<input checked="" type="checkbox"/>		<div> <div>jtag_uart</div> <div> <div>clk</div> <div>reset</div> <div>avalon_jtag_slave</div> </div> </div>	<div> <div>JTAG UART</div> <div>Clock Input</div> <div>Reset Input</div> <div>Avalon Memory Mapped Slave</div> </div>
<input checked="" type="checkbox"/>		<div> <div>onchip_mem</div> <div> <div>clk1</div> <div>s1</div> <div>reset1</div> </div> </div>	<div> <div>On-Chip Memory (RAM or ROM)</div> <div>Clock Input</div> <div>Avalon Memory Mapped Slave</div> <div>Reset Input</div> </div>
<input checked="" type="checkbox"/>		<div> <div>nios2_qsys</div> <div> <div>clk</div> <div>reset_n</div> <div>data_master</div> <div>instruction_master</div> <div>jtag_debug_module_re...</div> <div>jtag_debug_module</div> <div>custom_instruction_m...</div> </div> </div>	<div> <div>Nios II Processor</div> <div>Clock Input</div> <div>Reset Input</div> <div>Avalon Memory Mapped Master</div> <div>Avalon Memory Mapped Master</div> <div>Reset Output</div> <div>Avalon Memory Mapped Slave</div> <div>Custom Instruction Master</div> </div>
<input checked="" type="checkbox"/>		<div> <div>myadc</div> <div> <div>clock_sink</div> <div>reset_sink</div> <div>conduit_end</div> <div>avalon_slave</div> </div> </div>	<div> <div>adc</div> <div>Clock Input</div> <div>Reset Input</div> <div>Conduit</div> <div>Avalon Memory Mapped Slave</div> </div>
<input checked="" type="checkbox"/>		<div> <div>mydac</div> <div> <div>clock_sink</div> <div>reset_sink</div> <div>conduit_end</div> <div>avalon_slave</div> </div> </div>	<div> <div>dac</div> <div>Clock Input</div> <div>Reset Input</div> <div>Conduit</div> <div>Avalon Memory Mapped Slave</div> </div>
<input checked="" type="checkbox"/>		<div> <div>sw_pio</div> <div> <div>clk</div> <div>reset</div> <div>data_master</div> <div>instruction_master</div> <div>jtag_debug_module_re...</div> <div>jtag_debug_module</div> <div>custom_instruction_m...</div> </div> </div>	<div> <div>PIO (Parallel I/O)</div> <div>Clock Input</div> <div>Reset Input</div> <div>Avalon Memory Mapped Master</div> <div>Avalon Memory Mapped Master</div> <div>Reset Output</div> <div>Avalon Memory Mapped Slave</div> <div>Custom Instruction Master</div> </div>

完成组件添加配置后，点击菜单栏的 **System→Assign Base Addresses** 进行地址的重新自动分配。最后来到 **Generation** 页面，点击右下角的 **Generate** 按键生成新系统。

在 **Generation** 中，同样请关闭 **Simulation** 功能。

6.9.4 例化系统

回到 Quartus II 工程的顶层文件中，修改代码如下，例化新添加的 AD/DA 芯片管脚。

《圣经》箴言九 11 “敬畏耶和华是智慧的开端，认识至胜者便是聪明。”



```
module ex2(
    clk, rst_n, led,
    sw_pio,
    seg_db, seg_cs,
    adc_data, adc_cs_n, adc_clk,
    dac_scl, dac_sda
);
input clk;
input rst_n;
output led;
input[2:0] sw_pio;
output[7:0] seg_db;
output[3:0] seg_cs;
input adc_data;    //TLC549 数据信号
output adc_cs_n;   //TLC549 片选信号, 低电平有效
output adc_clk;    //TLC549 时钟信号
output dac_scl;    //DAC5571 数据信号
inout dac_sda;     //DAC5571 时钟信号

myqsys u0 (
    //clk.clk
    .clk_clk (clk),
    //pio_0_external_connection.export
    .pio_0_external_connection_export(led),
    //clk_0_clk_in_reset.reset_n
    .clk_0_clk_in_reset_reset_n(rst_n),
    //nios2_qsys_0_custom_instruction_master.readra
    .nios2_qsys_0_custom_instruction_master_readra ( ),
    //sw_pio_external_connection.export
    .sw_pio_external_connection_export(sw_pio),
    //myseg7_conduit_end.db
    .myseg7_conduit_end_db (seg_db),
    //.cs
    .myseg7_conduit_end_cs(seg_cs),
    //mydac_conduit_end.scl
    .mydac_conduit_end_scl(dac_scl),
    //.sda
    .mydac_conduit_end_sda(dac_sda),
```



```
//myadc_conduit_end.data
.myadc_conduit_end_data(adc_data),
    //.cs_n
.myadc_conduit_end_cs_n(adc_cs_n),
    //.clk
.myadc_conduit_end_clk(adc_clk)
);

Endmodule
```

分配新添加的管脚,可以在 Pin Planner 中分配,也可以在 Tcl console 窗口中输入如下的管脚分配脚本:

```
set_location_assignment PIN_113 -to adc_clk
set_location_assignment PIN_111 -to adc_cs_n
set_location_assignment PIN_112 -to adc_data
set_location_assignment PIN_106 -to dac_scl
set_location_assignment PIN_110 -to dac_sda
```

对工程做全编译,然后将 sof 文件下载到目标板中。提醒大家注意,请再次确认 SF-BASE 板上 P3 插座的 PIN2 和 PIN3 用跳线帽短接了。

6.9.5 软件编程

打开 EDS,以 ex12 工程目录下的 myqsys.sopcinfo 为硬件驱动,新建软件工程,命名为 adda_prj。在应用工程下新建一个 main.c 文件。输入以下代码。

```
/*
 * main.c
 *
 * Created on: 2013-2-10
 * Author: Administrator
 */

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



```
#include <unistd.h>

void delay(alt_u32 cnt);

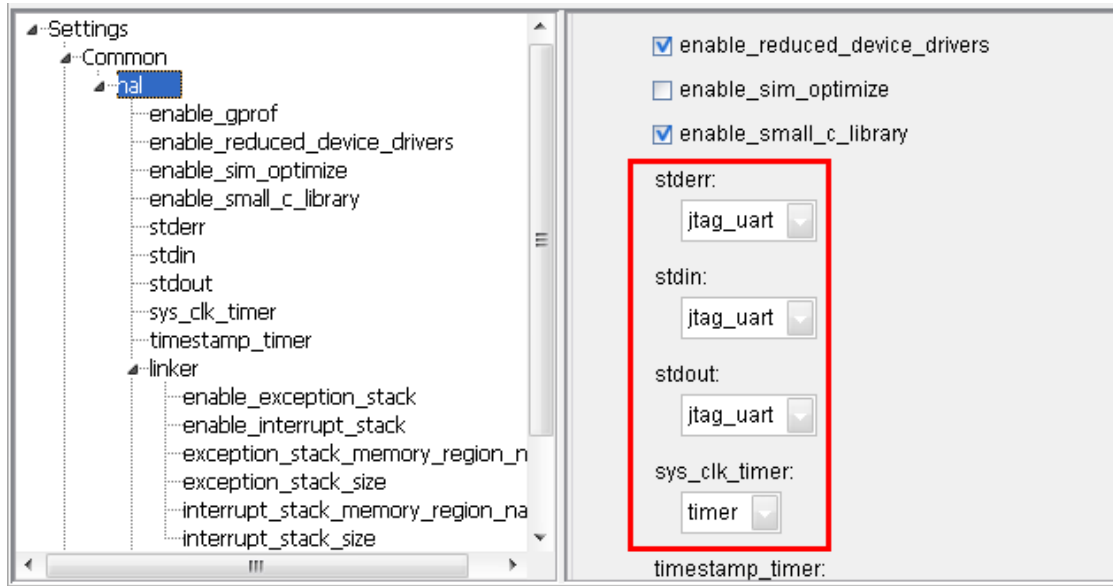
//主函数
int main()
{

    alt_u16 ad_dis;
    alt_u8 cnt = 0;

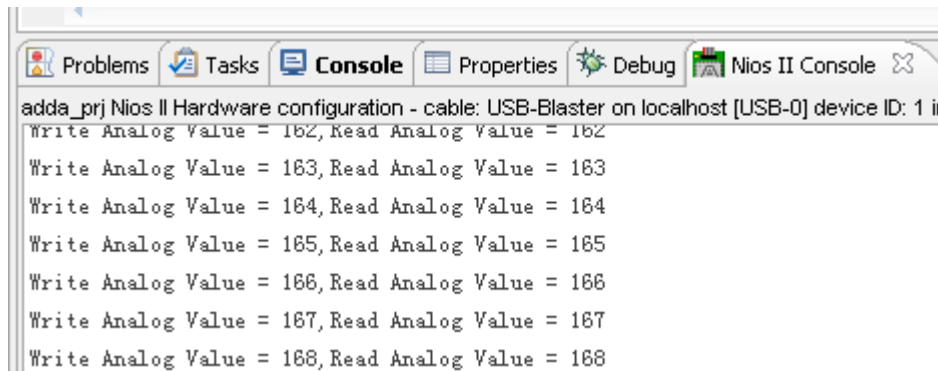
    while(1)
    {
        IOWR_8DIRECT(MYDAC_BASE, 0, cnt);
        printf("Write Analog Value = %d,", cnt);
        delay(60000);    //延时
        ad_dis = IORD_8DIRECT(MYADC_BASE, 0);    //读取当前 AD 采样值
        printf("Read Analog Value = %d\n", ad_dis);
        cnt++;
    }
    return 0;
}

//延时函数（延时时间为(2+2*i)us）
void delay(alt_u32 cnt)
{
    alt_u32 i =0;
    while(i < cnt)
    {
        i++;
    }
}
```

打开 **BSP Editor**，做代码裁剪，但是注意如下图所示的几处设置不要弄错。



在目标板上 Run 起来我们的软件以后，我们会看到 EDS 的 Nios II Console 中不断的打印 AD/DA 转换后的数据，完全的一致。与此同时，我们可以看到 D9 指示灯也是根据我们输出的模拟电压值不停的亮暗闪烁。



7 SF-LCD 子板开发指南

7.1 功能与原理图介绍

7.1.1 主要外设芯片及电路图解析

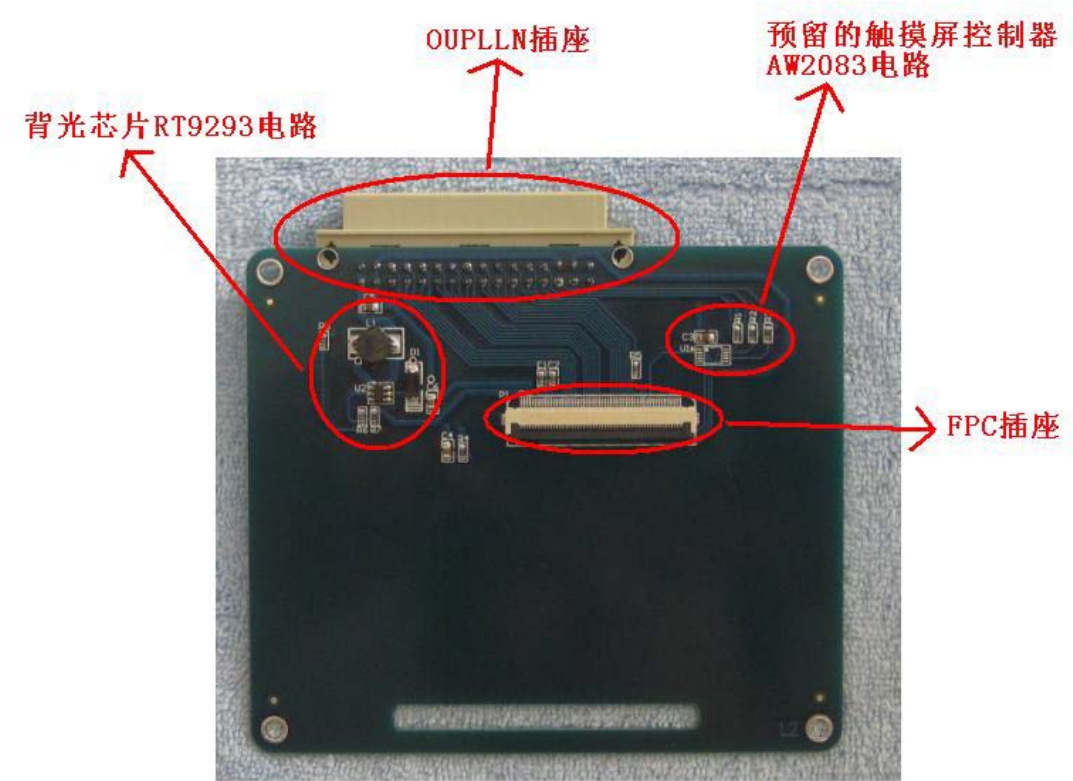
SF-LCD 子板主要外设芯片及其功能描述见下表。

外设芯片	主要功能
------	------

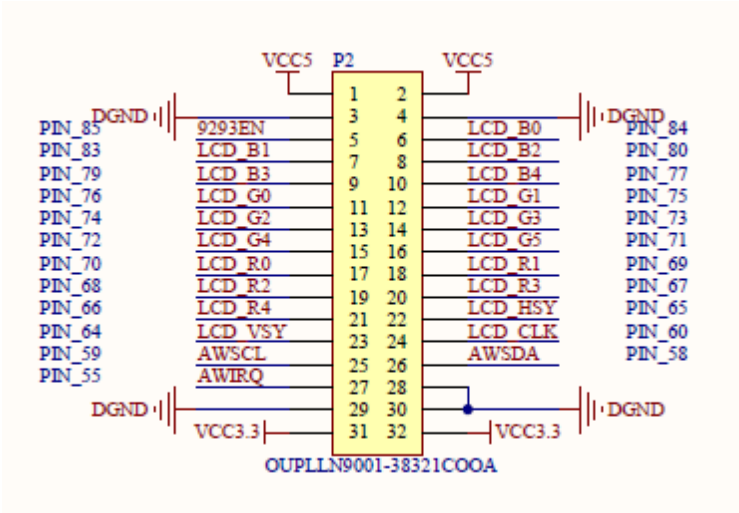


32PIN 的 OUPLLN 插座	用于 SF-LCD 子板上的各个外设与 SF-CY3 核心板相连。
54PIN 的 FPC 插座	用于连接奇美公司的 3.5 寸彩色液晶屏。
背光芯片 RT9293	用于产生 3.5 寸液晶屏的背光电压。
触摸屏控制器 AW2083	用于电阻触摸屏的模拟电压采集和转换。目前该电路不焊接。

各个主要外设芯片的实物位置如图所示。



32PIN 的 OUPLLN 插座原理图如图所示。该插座用于连接到 SF-CY3 核心板的插座 P3。



54PIN 的 FPC 插座定义如图所示。这个 LCD 的驱动接口定义如下。



管脚	信号	功能
1	VBL-	LED 背光地。
2	VBL-	LED 背光地。
3	VBL+	LED 背光电源。
4	VBL+	LED 背光电源。
5	Y1	触摸屏: 上。
6	X1	触摸屏: 右。
7	NC	无连接。
8	RESET#	硬件复位。
9	SPENA	SPI 接口数据使能信号。
10	SPCLK	SPI 接口数据锁存时钟。
11	SPDAT	SPI 接口数据信号。
12	B0	蓝色数据位 0。
13	B1	蓝色数据位 1。
14	B2	蓝色数据位 2。
15	B3	蓝色数据位 3。
16	B4	蓝色数据位 4。
17	B5	蓝色数据位 5。
18	B6	蓝色数据位 6。
19	B7	蓝色数据位 7。
20	G0	绿色数据位 0。
21	G1	绿色数据位 1。
22	G2	绿色数据位 2。
23	G3	绿色数据位 3。
24	G4	绿色数据位 4。
25	G5	绿色数据位 5。
26	G6	绿色数据位 6。
27	G7	绿色数据位 7。
28	R0	红色数据位 0。



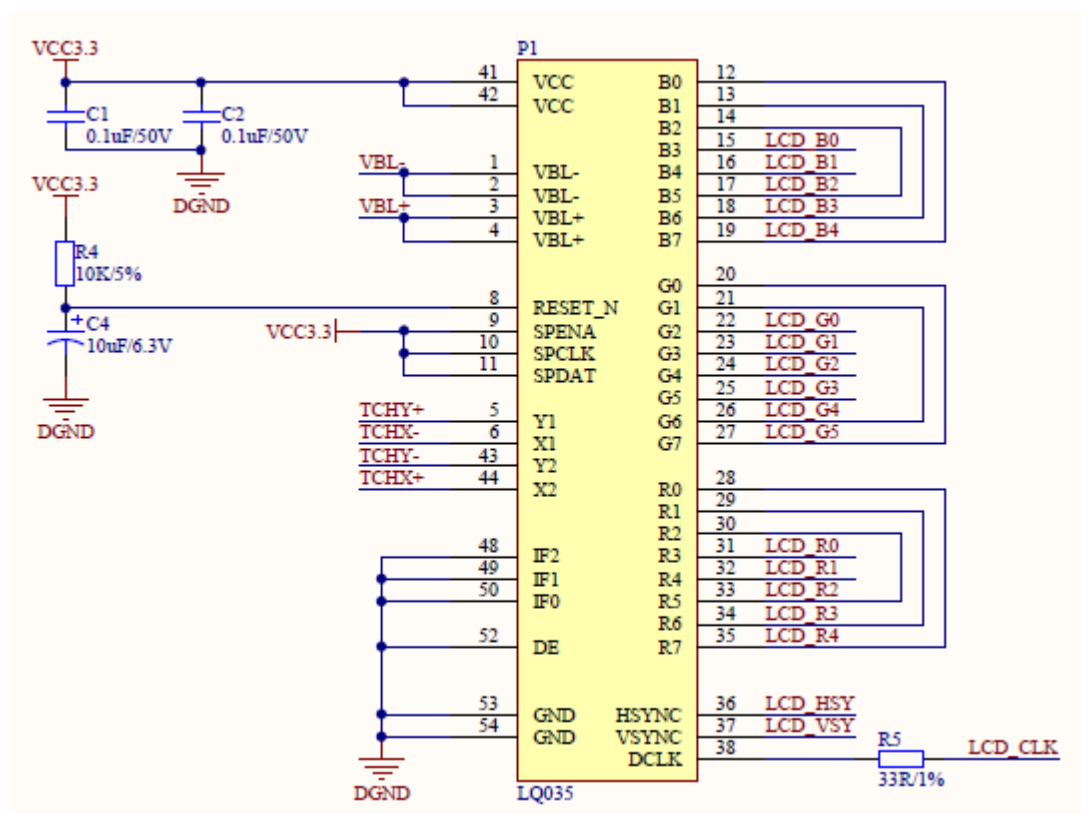
29	R1	红色数据位 1。
30	R2	红色数据位 2。
31	R3	红色数据位 3。
32	R4	红色数据位 4。
33	R5	红色数据位 5。
34	R6	红色数据位 6。
35	R7	红色数据位 7。
36	HSYNC	行同步信号。
37	VSYNC	场同步信号。
38	DCLK	显示数据锁存时钟。
39	NC	不连接。
40	NC	不连接。
41	VCC	数字电源。
42	VCC	数字电源。
43	Y2	触摸屏: 下。
44	X2	触摸屏: 左。
45	NC	不连接。
46	NC	不连接。
47	NC	不连接。
48	IF2	数据输入格式控制管脚。
49	IF1	数据输入格式控制管脚。
50	IF0	数据输入格式控制管脚。
51	NC	不连接。
52	DE	数据输入使能。
53	GND	数字地。
54	GND	数字地。

我们可以将该表中的信号接口归为五类。第一类是数字信号接口, 如 RESET、SPENA、SPCLK、SPDAT、Rx (x 为 0 到 7)、Gx、Bx、HSYNC、VSYNC、DCLK 和 DE。此类信号主要是传输显示数据给 LCD 面板, 这么多接口, 是不是所有的管脚都要用上呢? 不是的, 其实行《圣经》箴言九 11 “敬畏耶和华是智慧的开端, 认识至胜者便是聪明。”



细看这款 LCD 的 datasheet, 我们发现它提供了多种数据传输方式, 有常见的并行 RGB 数据传输, 也有 CCIR601/656 等方式。前者通常驱动时钟慢一些, 而需要的数据总线宽一些, 传输协议也更简单, 我们也更倾向于采用前者进行通信。由于这里的数据接口合计是 24bit 的, 也就是说每个像素点的色彩可以显示 2 的 24 次方种, 即通常所说的 1600 万色。不过实际上我们并没有用足这 24bit 数据线, 我们的图片是 16bit 的, 基本上人眼感觉已经够绚丽了。因此, 在硬件连接上, 我们做了如原理图所示的处理, 为的是减少数据位宽。SPENA、SPCLK、SPDAT 是 SPI 接口, 用于给 LCD 的一些控制寄存器写数据, 有些液晶屏需要在 LCD 上电后用该接口做一些配置才能够正常使用, 而我们使用的这款屏则不需要, 因此我们可以不必理会这些管脚。时序的控制上既可以用 HSYNC/VSYN 模式 (我们的电路上使用了该模式), 也可以用只有 DE 的模式。

第二类接口, 即液晶的模式设置专用输入接口, 包括了 IF0、IF1、IF2 等接口, 它们的主要功能就是设置使用哪种数据传输方式, 本实验采用并行 RGB 数据传输。第三类接口是触摸屏信号接口, 是模拟信号, 如 Y1、X1、Y2、X2, 这些管脚是否使用需要看液晶屏是否真的接好了触摸屏。第四类接口是电源接口, 即 Vcc (接 3.3V) 和 GND 信号。第五类是背光电源, 即 VBL+和 VBL-信号, 这部分硬件上设计了专门的背光电源电路产生 19.8V 电压进行供电。



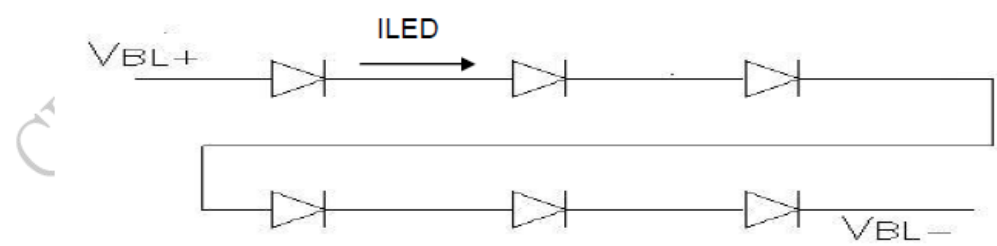


LCD 内部的背光是 6 个 LED 串联，每个 LED 的额定电压是 3.3V，电流是 20mA。因此，要驱动这 6 个 LED 就需要 19.8V/20mA 的电压。我们的系统输入电源是 5V，必须升压才能够得到 19.8V。

5.2 LED driving conditions

Parameter	Symbol	Min.	Typ.	Max.	Unit	Remark
LED current		-	20	-	mA	
Power Consumption		-	400	420	mW	
LED voltage	VBL+	18.6	19.8	21	V	Note 1
LED Life Time	-		(50,000)-	-	Hr	Note 2,3

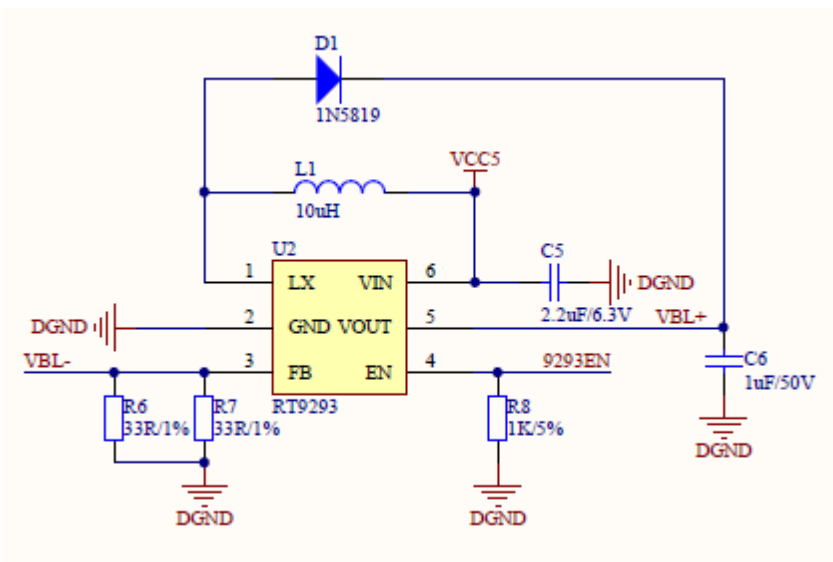
Note 1 : There are 1 Groups LED



Note 2 : Ta = 25°C

Note 3 : Brightness to be decreased to 50% of the initial value

LCD 驱动的背景光电路如图所示，这里使用了专用升压芯片 RT9293，该芯片为恒流控制，只要设置电流就能够自动产生所需压降。FB 管脚和 GND 之间有两个 33ohm 电阻并联，得到的电阻是 16.5ohm，该阻值对应设置了约 20mA 的驱动电流。



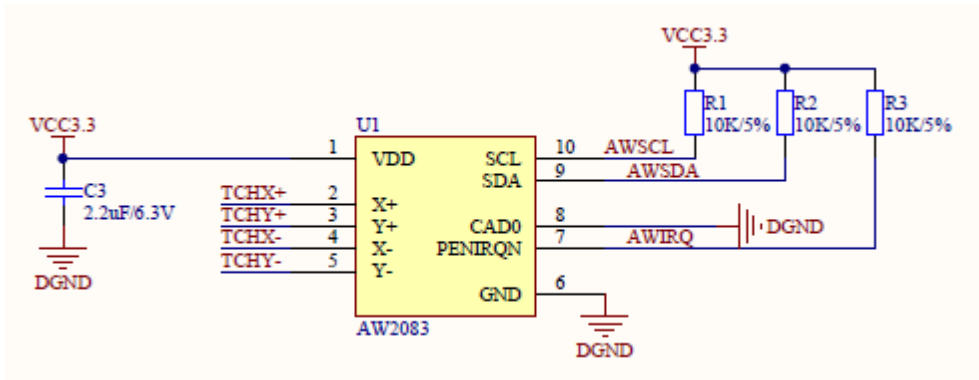
RT9293 的管脚定义如下。



Functional Pin Description

Pin No.		Pin Name	Pin Function
RT9293□GJ6	RT9293□GQW		
1	8	LX	Switching Pin.
2	1, 5, 9 (Exposed pad)	GND	Ground Pin. The exposed pad must be soldered to a large PCB and connected to GND for maximum power dissipation.
3	6	FB	Feedback Pin, put a resistor to GND to setting the current.
4	4	EN	Chip Enable (Active High).
5	3	VOUT	Output Voltage Pin.
6	2	VIN	Input Supply.
--	7	NC	No Internal Connection.

触摸屏驱动器芯片 AW2083 电路如图。它由两路的差分模拟电压输入，用于采集电阻式触摸屏的模拟电压。数字端有一个 IRQ 中断信号可以连接到 FPGA，FPGA 接收到中断后，作为在主机，通过 IIC 接口读取的当前的触摸屏坐标数据。（由于我们所使用的 3.5 寸液晶屏 LQ035NC111 带触摸屏版本已经停产，所以我们电路上只是保留，但不焊接芯片）



7.1.2 装配示意图

如果你购买我们的 SF-LCD 子模块，那么我们将提供以下的各个配件。

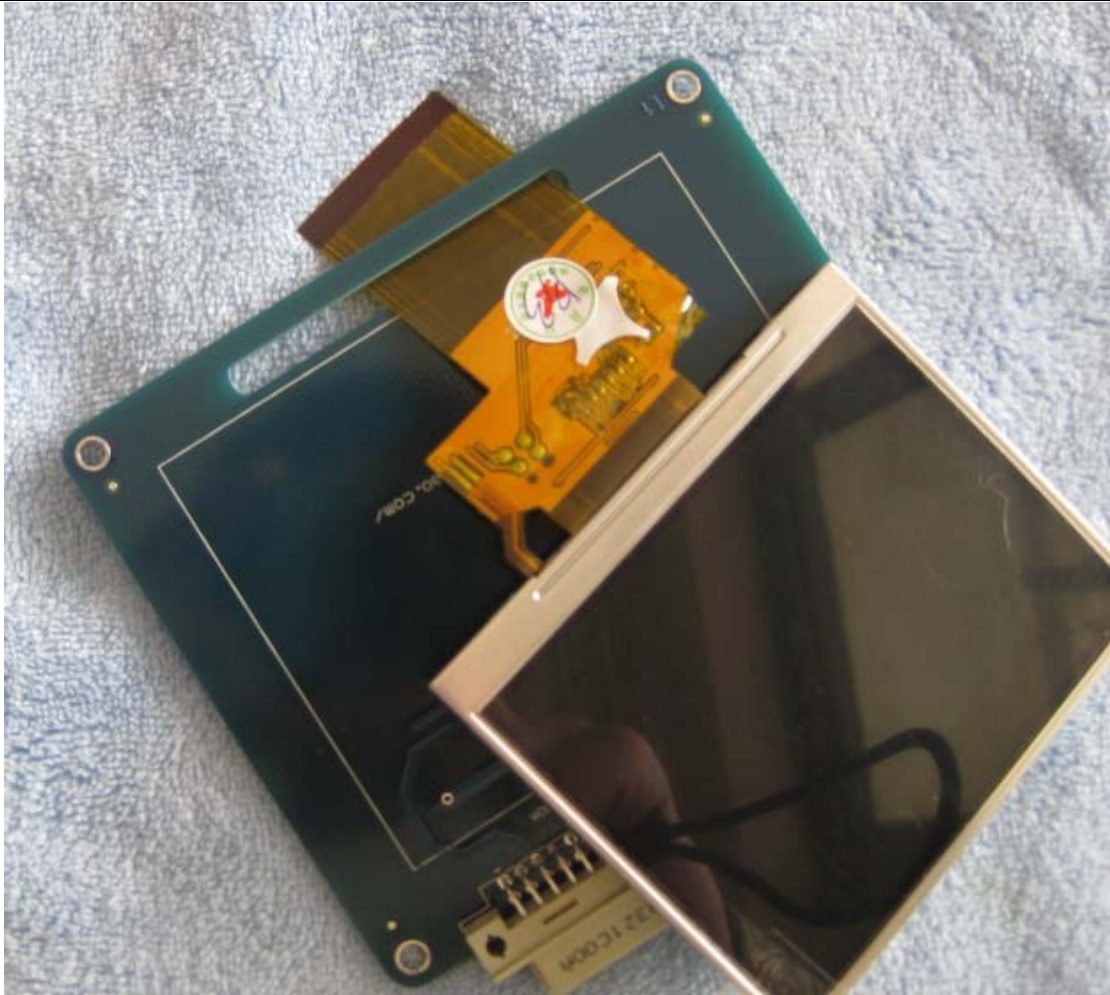
名称	数量
SF-LCD 电路板	一块
3.5 寸液晶屏	一块
资料光盘	一张
3M 双面胶	一片



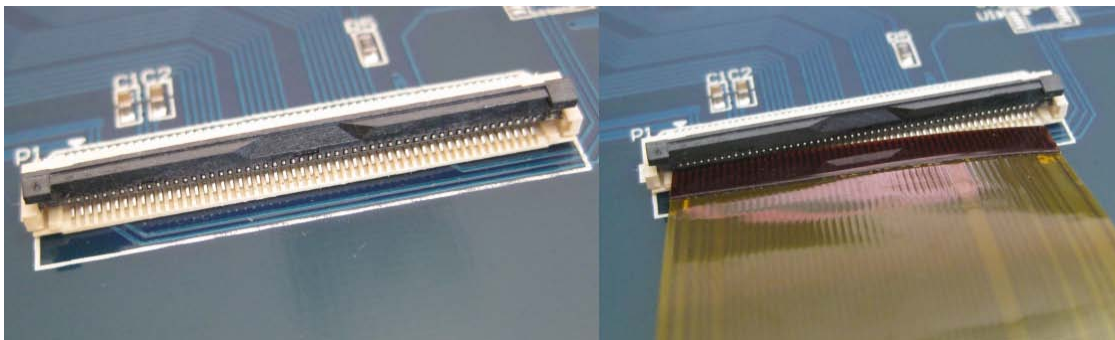
下面我们来看看如何把液晶屏安装到电路板上。首先，撕下一面的双面胶，将其贴在液晶屏的背面，注意先别急着把双面胶的另一面也撕掉。

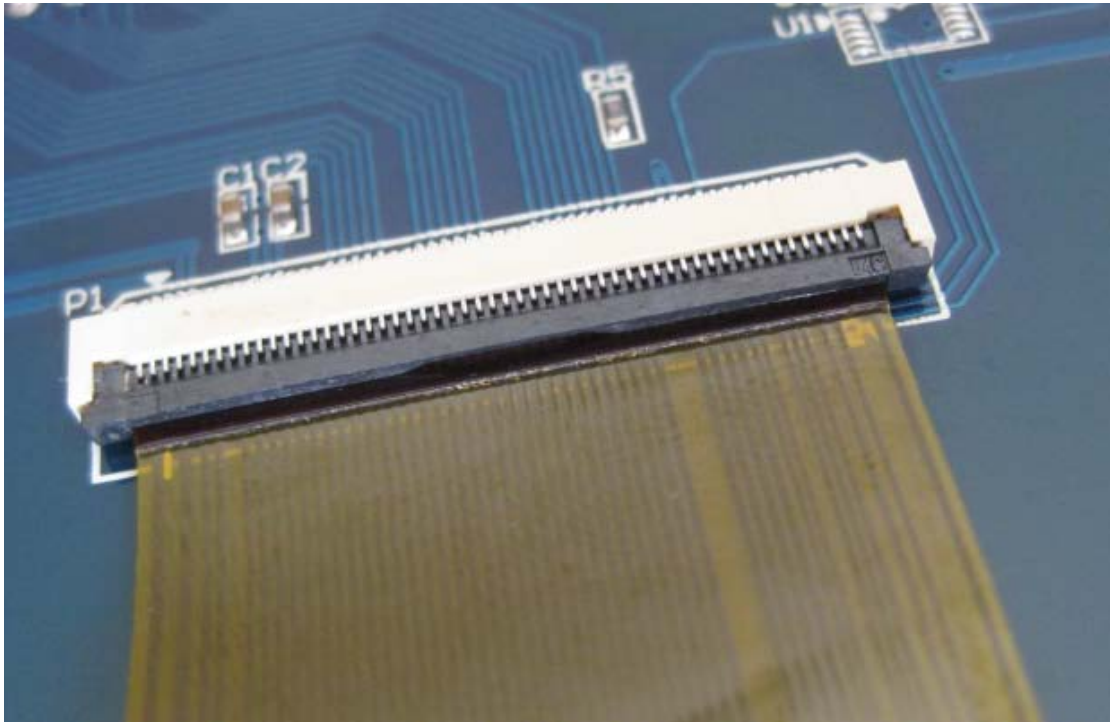


将液晶屏的 FPC 线插入电路板预留的长条形镂空孔中。

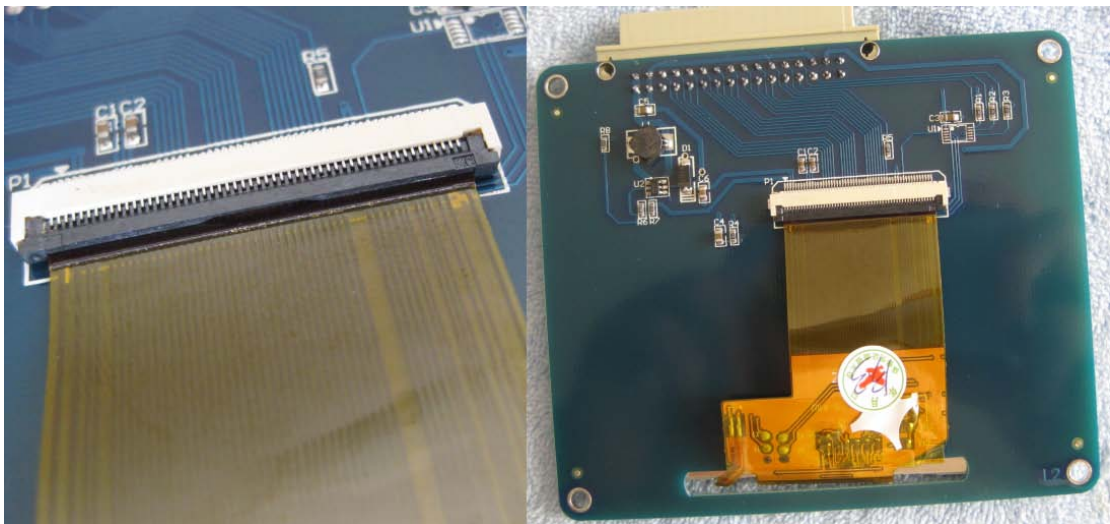


翻开电路板上的插座 P1 的盖子，将液晶屏的软排线小心的插入其中，注意一定要插到位，但也不能过于用力，以免损坏插座。





插好 FPC 插座后，如图所示。



接着我们回到另一面，把液晶屏背面的双面胶揭掉。



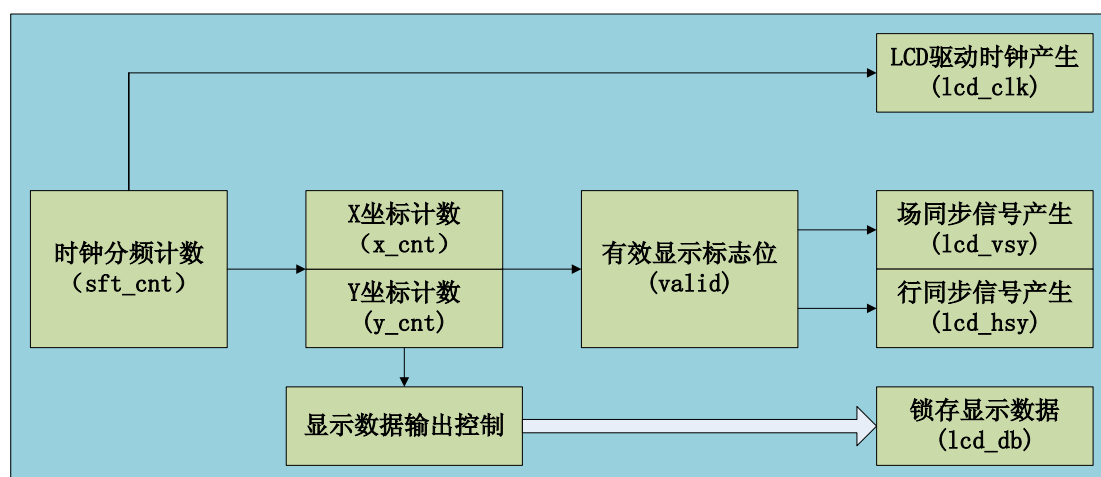
找准位置，将液晶屏贴到电路板上。最后，我们将装配好的 SF-LCD 子模块连接到 SF-CY3 核心模块的 P3 插座上。效果如图所示。



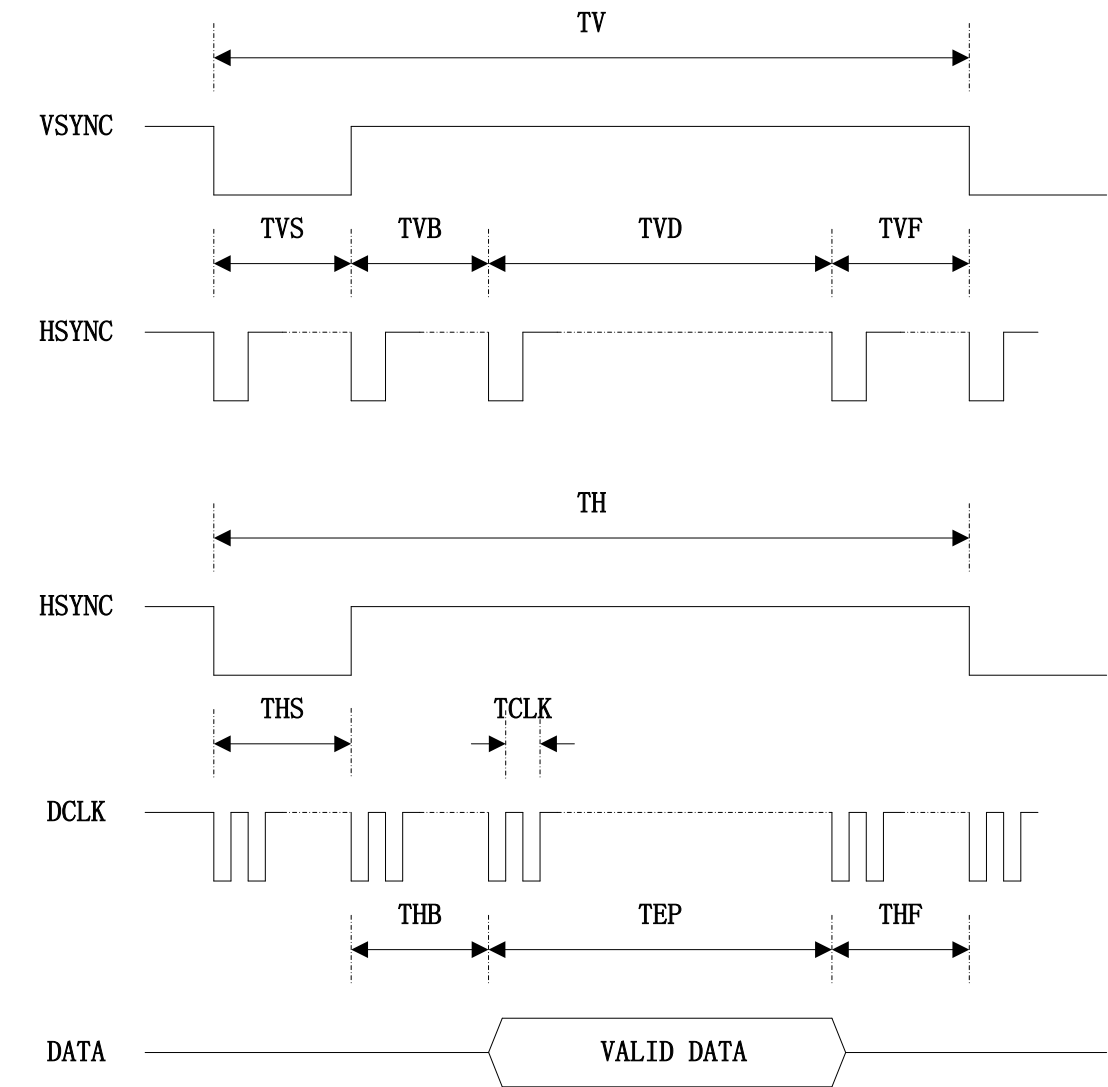
7.2 逻辑（Verilog）实例 9——LCD 的基本驱动

7.2.1 LCD 驱动原理

LCD 显示驱动模块的内部逻辑功能大体可以如图所示。每个功能块都有相应的时钟驱动和复位。图中右侧部分均为 FPGA 直接与 LCD 连接的驱动信号，FPGA 通过这些信号来产生一些符合 LCD 操作时序的控制信号（包括 `lcd_clk`、`lcd_en`、`lcd_vsy`、`lcd_hsy`）并在指定的时刻送当前显示的彩色数据（`lcd_db_r`、`lcd_db_g`、`lcd_db_b`）即可。



再看 LCD 的接口时序, 如图所示。VSYNC 是场同步信号, 低电平有效, 从时序图示可以看出, VSYNC 是每一场 (即也可以理解为每送一幅完整图像) 的同步信号; 与此类似, HSYNC 是行同步信号, 也是在每一行数据传输的开始产生几个时钟周期的低脉冲。这两个信号用于同步当前的数据信号, 根据固定的脉冲约定, 我们在某些时钟上升沿前将图像数据送到数据总线上供 LCD 内部锁存。



下表是 LCD 时序图中对应的时间参数。

信号	列项	标记	最小	标准	最大	单位
Dclk	频率	Tosc		156		ns
	最大时间	Tch		78		ns
	最小时间	Tcl		78		ns
Data	建立时间	Tsu	12			ns
	保持时间	Thd	12			ns
Hsync	周期	TH		408		Tosc
	脉冲宽度	THS	5	30		Tosc
	后沿	THB		38		Tosc



	显示周期	TEP		320		Tosc
	同步周期	THE	36	68	88	Tosc
	前沿	THF		20		Tosc
Vsync	周期	TV		262		TH
	脉冲宽度	TVS	1	3	5	TH
	后沿	TVB		15		TH
	显示周期	TVD		240		TH
	前沿	TVF	2	4		TH

在实际逻辑设计中, 由于系统时钟 50MHz, LCD 时钟通常为 6.25MHz 左右, 因此为了产生 LCD 时钟就需要做一些分频, 时钟分频计数单元的逻辑即完成此功能。此外, 因为我们内部的操作都是以 50MHz 时钟为基准, 所以这个分频计数单元也相应产生一个使能时钟, 即 dchange 信号, 它在每 160ns (即 6.25MHz) 周期内产生一个 20ns (即 50MHz) 的高脉冲, 作为后续逻辑电路的坐标计数使能信号。

x_cnt 和 y_cnt 计数器用于产生符合 LCD 逻辑的行和场同步信号, 也能够对应的计数范围内产生数据输出有效指示信号。LCD 的复位信号在上电初始的时间内有效一段时间后 (即处于复位状态) 撤销, 其控制时间也和 x_cnt、y_cnt 计数器有关。FIFO 读请求信号的产生也和 x_cnt、y_cnt 计数器以及分频计数单元产生的 dchange 信号有关, 外部的 SDRAM 控制器在每一场的开始都会清空 FIFO, 然后保持 FIFO 中有数据 (但不溢出) 可供当前显示。在每次 FIFO 读请求信号有效 (高电平) 后, 相应的 FIFO 输出数据也会送到该模块内部进行锁存。简单的说, 可以理解为该模块的数据流都是通过 SDRAM 控制器模块的读 FIFO 进行交互的, 他们之间不仅有数据总线, 还有一些控制信号, 如 FIFO 异步清除信号、FIFO 读请求信号以及该模块所使用的时钟信号作为读 FIFO 时钟。

7.2.2 Verilog 代码

在本实例的驱动中, 我们特别将全屏显示为黑屏, 同时让 LCD 四个最边角的显示线为蓝色。以此可以验证我们的驱动有效覆盖到了 LCD 的全部显示区域。

```
module ex13(
```



```
        clk,rst_n,
        lcd_en,lcd_clk,lcd_hsy,lcd_vsy,lcd_db_r,lcd_db_g,lcd_db_b
    );
input clk;        //25MHz
input rst_n;      //低电平复位
    // FPGA 与 LCD 接口信号
output lcd_en;    //背光使能信号,高有效
output lcd_clk;   //时钟信号
output lcd_hsy;   //行同步信号
output lcd_vsy;   //场同步信号
output[4:0] lcd_db_r;
output[5:0] lcd_db_g;
output[4:0] lcd_db_b;

//-----
assign lcd_en = 1'b1;

//-----
//lcd_clk 时钟周期为 160ns (6.25MHz),即 4 个 25MHz 的时钟周期
reg[1:0] sft_cnt;

always @(posedge clk or negedge rst_n)
    if(!rst_n) sft_cnt <= 2'd0;
    else sft_cnt <= sft_cnt+1'b1;

assign lcd_clk = sft_cnt[1];    //0-1:low,2-3:high

wire dchange = (sft_cnt == 2'd2);    //数据变化标志位,高有效

//-----
//坐标计数
//x = 0-407; y = 0-261
reg[8:0] x_cnt; //x 计数器
reg[8:0] y_cnt; //y 计数器

always @(posedge clk or negedge rst_n)
    if(!rst_n) x_cnt <= 9'd0;
    else if(dchange) begin
```




```
        if(x_cnt == 9'd407) x_cnt <= 9'd0;
        else x_cnt <= x_cnt+1'b1;
    end

always @(posedge clk or negedge rst_n)
    if(!rst_n) y_cnt <= 9'd0;
    else if(dchange && (x_cnt == 9'd407)) begin
        if(y_cnt == 9'd261) y_cnt <= 9'd0;
        else y_cnt <= y_cnt+1'b1;
    end

//-----
//有效显示标志位产生
reg valid_yr;    //行显示有效信号

    //行显示有效信号
always @ (posedge clk or negedge rst_n)
    if(!rst_n) valid_yr <= 1'b0;
    else if(y_cnt == 9'd18) valid_yr <= 1'b1;
    else if(y_cnt == 9'd258) valid_yr <= 1'b0;

reg valid;

always @ (posedge clk or negedge rst_n)
    if(!rst_n) valid <= 1'b0;
    else if((x_cnt == 9'd68) && valid_yr) valid <= 1'b1;
    else if((x_cnt == 9'd388) && valid_yr) valid <= 1'b0;

//-----
// LCD 场同步, 行同步信号
reg lcd_hsy_r, lcd_vsy_r;    //同步信号

always @ (posedge clk or negedge rst_n)
    if(!rst_n) lcd_hsy_r <= 1'b1;
    else if(x_cnt == 9'd0) lcd_hsy_r <= 1'b0;    //产生 lcd_hsy 信号
    else if(x_cnt == 9'd30) lcd_hsy_r <= 1'b1;

always @ (posedge clk or negedge rst_n)
```



```
        if(!rst_n) lcd_vsy_r <= 1'b1;
        else if(y_cnt == 9'd0) lcd_vsy_r <= 1'b0;    //产生 lcd_vsy 信号
        else if(y_cnt == 9'd3) lcd_vsy_r <= 1'b1;

assign lcd_hsy = lcd_hsy_r;
assign lcd_vsy = lcd_vsy_r;

//-----
// LCD 色彩信号产生
reg[15:0] lcd_db_rgb;    // LCD 色彩显示寄存器

always @ (posedge clk or negedge rst_n)
    if(!rst_n) lcd_db_rgb <= 16'd0;
    else if(x_cnt == 9'd68) lcd_db_rgb <= 16'h001f;
    else if(x_cnt == 9'd387) lcd_db_rgb <= 16'h001f;
    else if(y_cnt == 9'd18) lcd_db_rgb <= 16'h001f;
    else if(y_cnt == 9'd257) lcd_db_rgb <= 16'h001f;
    else lcd_db_rgb <= 16'd0;

//r, g, b 控制液晶屏颜色显示
assign lcd_db_r = valid ? lcd_db_rgb[15:11]:5'd0;
assign lcd_db_g = valid ? lcd_db_rgb[10:5]:6'd0;
assign lcd_db_b = valid ? lcd_db_rgb[4:0]:5'd0;

endmodule
```

7.2.3 工程实践

- (1) 新建工程, 命名为 **ex13**, 把这个工程放在专门的文件夹下, 其他设置参考前面章节。
- (2) 新建 Verilog 源文件, 命名为 **ex13**, 输入前面给出的设计代码。使用 ModelSim-Altera 对设计代码进行仿真验证。
- (3) 综合编译后进行管脚分配, 本例程的管脚分配如下所示。



Node Name	Direction	Location
clk	Input	PIN_22
lcd_clk	Output	PIN_60
lcd_db_b[4]	Output	PIN_77
lcd_db_b[3]	Output	PIN_79
lcd_db_b[2]	Output	PIN_80
lcd_db_b[1]	Output	PIN_83
lcd_db_b[0]	Output	PIN_84
lcd_db_g[5]	Output	PIN_71
lcd_db_g[4]	Output	PIN_72
lcd_db_g[3]	Output	PIN_73
lcd_db_g[2]	Output	PIN_74
lcd_db_g[1]	Output	PIN_75
lcd_db_g[0]	Output	PIN_76
lcd_db_r[4]	Output	PIN_66
lcd_db_r[3]	Output	PIN_67
lcd_db_r[2]	Output	PIN_68
lcd_db_r[1]	Output	PIN_69
lcd_db_r[0]	Output	PIN_70
lcd_en	Output	PIN_85
lcd_hsy	Output	PIN_65
lcd_vsy	Output	PIN_64
rst_n	Input	PIN_91

- (4) 参考前面章节, 打开 TimeQuest, 我们新建一个 SDC 文件, 然后对时钟 clk 做约束, 约束脚本如下:

```
create_clock -name {SYSCLK} -period 40.000 -waveform { 0.000 20.000 } [get_ports {clk}]
```

- (5) 我们对工程进行全编译, 不仅要让刚刚添加的时钟约束生效, 也要生成可以下载到 FPGA 芯片中的配置文件。
- (6) 连接好 SF-LCD 子板和 SF-CY3 核心板, 连接好下载线以及电源, 给板子上电。
- (7) 通过 Programmer 下载 sof 到 SF-CY3 板中, 观察液晶屏此时的显示。应该是如图所示, 整个液晶屏为黑色, 但是蓝色的边框。



7.3 逻辑（Verilog）实例 10——LCD 的 32 级红色显示

7.3.1 色彩显示原理

我们知道自然界的任何一种色彩都是由红、绿、蓝三种基本色组成的。我们实验所用的 3.5 寸 LCD 面板上是由 320×240 个像素点组成显像的，每个独立的像素色彩是由红、绿、蓝（R、G、B）三种基本色来控制。大部分厂商生产出来的液晶显示器，每个基本色（R、G、B）达到 6 位，即 64 种表现度，那么每个独立的像素就有 $64 \times 64 \times 64 = 262144$ 种色彩。也有不少厂商使用了所谓的 FRC（Frame Rate Control）技术以仿真的方式来表现出全彩的画面，也就是每个基本色（R、G、B）能达到 8 位，即 256 种表现度，那么每个独立的像素就有高达 $256 \times 256 \times 256 = 16777216$ 种色彩了。

我们所选的 LCD 实际上是支持 3 个 8bit 的 R、G、B 色彩，但是出于 I/O 的考虑，我们只是连接为了 5bit 的 R 色彩、6bit 的 G 色彩、5bit 的 B 色彩。R、G、B 不同值的组合相应的会衍生出各种各样不同的色彩。当 R、G、B 都为 0 时，我们看到的色彩是纯黑；当 R、G、B 都为最大值时，我们看到的色彩是纯白；而当 R、G、B 其中的两个色彩为 0，另一个色彩的高低值就是它的深浅单色显示。如我们这个实例要将 G 和 B 色彩设置为 0，更改 5bit 的 R 值，那么在图像上就能够看到 32 级不同深浅度的红色。

在我们的 LCD 中，有 320×240 个像素点，对应为 320 列和 240 行的有效显示点。我们这个实例要在上一节的基础上，用逻辑控制每 10 列的显示区域为一种色彩，320 列一共就

《圣经》箴言九 11 “敬畏耶和华是智慧的开端，认识至胜者便是聪明。”



能出现 32 种 R 的色彩。

7.3.2 Verilog 代码

```
module ex13(
    clk, rst_n,
    lcd_en, lcd_clk, lcd_hsy, lcd_vsy, lcd_db_r, lcd_db_g, lcd_db_b
);
input clk;        //25MHz
input rst_n;      //低电平复位
    // FPGA 与 LCD 接口信号
output lcd_en;    //背光使能信号, 高有效
output lcd_clk;   //时钟信号
output lcd_hsy;   //行同步信号
output lcd_vsy;   //场同步信号
output[4:0] lcd_db_r;
output[5:0] lcd_db_g;
output[4:0] lcd_db_b;

//-----
assign lcd_en = 1'b1;

//-----
//lcd_clk 时钟周期为 160ns (6.25MHz), 即 4 个 25MHz 的时钟周期
reg[1:0] sft_cnt;

always @(posedge clk or negedge rst_n)
    if(!rst_n) sft_cnt <= 2'd0;
    else sft_cnt <= sft_cnt+1'b1;

assign lcd_clk = sft_cnt[1];    //0-1:low, 2-3:high

wire dchange = (sft_cnt == 2'd2);    //数据变化标志位, 高有效

//-----
//坐标计数
```



```
//x = 0-407; y = 0-261
reg[8:0] x_cnt; //x 计数器
reg[8:0] y_cnt; //y 计数器

always @(posedge clk or negedge rst_n)
    if(!rst_n) x_cnt <= 9'd0;
    else if(dchange) begin
        if(x_cnt == 9'd407) x_cnt <= 9'd0;
        else x_cnt <= x_cnt+1'b1;
    end

always @(posedge clk or negedge rst_n)
    if(!rst_n) y_cnt <= 9'd0;
    else if(dchange && (x_cnt == 9'd407)) begin
        if(y_cnt == 9'd261) y_cnt <= 9'd0;
        else y_cnt <= y_cnt+1'b1;
    end

//-----
//有效显示标志位产生
reg valid_yr; //行显示有效信号

//行显示有效信号
always @ (posedge clk or negedge rst_n)
    if(!rst_n) valid_yr <= 1'b0;
    else if(y_cnt == 9'd18) valid_yr <= 1'b1;
    else if(y_cnt == 9'd258) valid_yr <= 1'b0;

reg validr,valid;

always @ (posedge clk or negedge rst_n)
    if(!rst_n) validr <= 1'b0;
    else if((x_cnt == 9'd67) && valid_yr) validr <= 1'b1;
    else if((x_cnt == 9'd387) && valid_yr) validr <= 1'b0;

always @ (posedge clk or negedge rst_n)
    if(!rst_n) valid <= 1'b0;
    else valid <= validr;
```



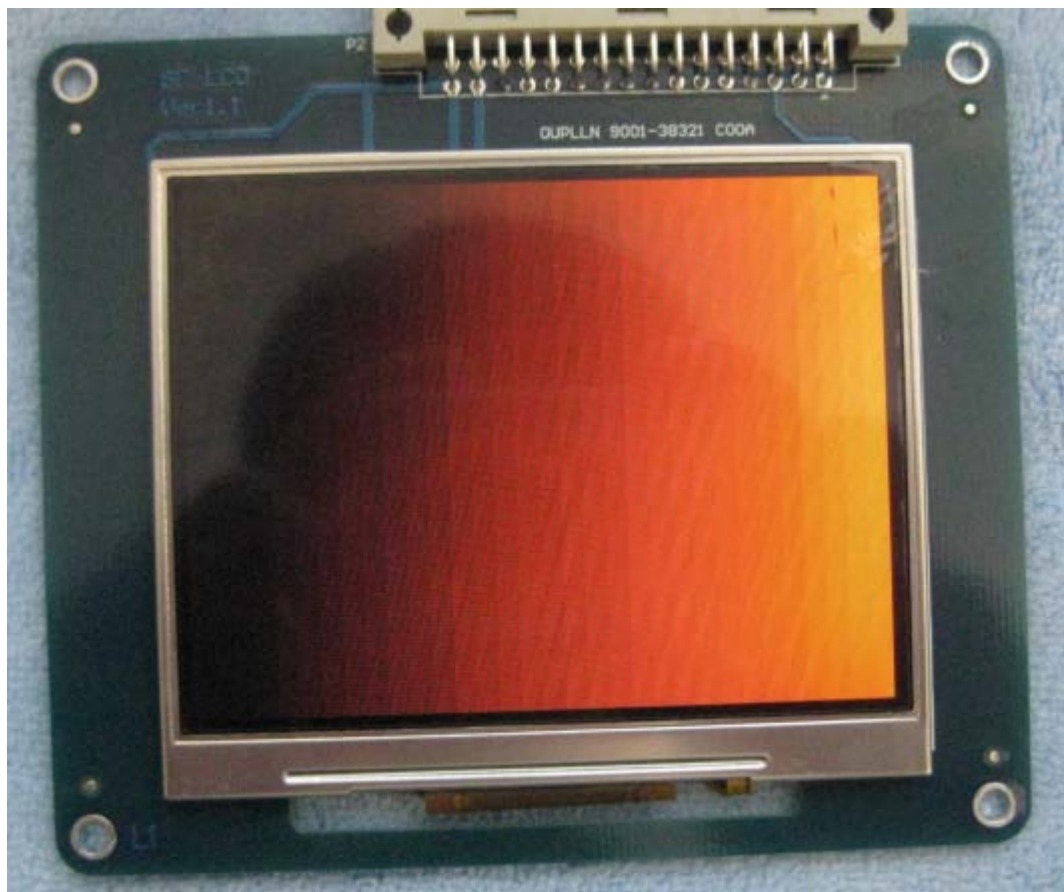
```
//-----  
// LCD 场同步, 行同步信号  
reg lcd_hsy_r, lcd_vsy_r;    //同步信号  
  
always @ (posedge clk or negedge rst_n)  
    if(!rst_n) lcd_hsy_r <= 1'b1;  
    else if(x_cnt == 9'd0) lcd_hsy_r <= 1'b0;    //产生 lcd_hsy 信号  
    else if(x_cnt == 9'd30) lcd_hsy_r <= 1'b1;  
  
always @ (posedge clk or negedge rst_n)  
    if(!rst_n) lcd_vsy_r <= 1'b1;  
    else if(y_cnt == 9'd0) lcd_vsy_r <= 1'b0;    //产生 lcd_vsy 信号  
    else if(y_cnt == 9'd3) lcd_vsy_r <= 1'b1;  
  
assign lcd_hsy = lcd_hsy_r;  
assign lcd_vsy = lcd_vsy_r;  
  
//-----  
    // LCD 色彩信号产生  
reg[3:0] tmp_cnt;    //0-9 循环计数, 每 10 个 x_cnt 有效显示计数周期递增一个 新的色彩值  
  
always @(posedge clk or negedge rst_n)  
    if(!rst_n) tmp_cnt <= 4'd0;  
    else if(!validr) tmp_cnt <= 4'd0;  
    else if(validr && dchange) begin  
        if(tmp_cnt < 4'd9) tmp_cnt <= tmp_cnt+1'b1;  
        else tmp_cnt <= 4'd0;  
    end  
  
reg[15:0] lcd_db_rgb;    // LCD 色彩显示寄存器  
  
always @ (posedge clk or negedge rst_n)  
    if(!rst_n) lcd_db_rgb <= 16'd0;  
    else if(validr) begin  
        if((tmp_cnt == 4'd9) && dchange) lcd_db_rgb[15:11] <=  
lcd_db_rgb[15:11]+1'b1;
```



```
        else ;  
    end  
    else  lcd_db_rgb <= 16'd0;  
  
    //r, g, b 控制液晶屏颜色显示  
assign lcd_db_r = valid ? lcd_db_rgb[15:11]:5'd0;  
assign lcd_db_g = valid ? lcd_db_rgb[10:5]:6'd0;  
assign lcd_db_b = valid ? lcd_db_rgb[4:0]:5'd0;  
  
endmodule
```

7.3.3 工程实践

- (1) 复制 **ex13** 整个工程文件夹，直接移植过来，更改文件夹名称为 **ex14**。
- (2) 使用本实例给出的 Verilog 代码替代 **ex13** 的工程代码。使用 ModelSim-Altera 对设计代码进行仿真验证。
- (3) 对工程进行全编译，生成可以下载到 FPGA 芯片中的配置文件。
- (4) 连接好 SF-LCD 子板和 SF-CY3 核心板，连接好下载线以及电源，给板子上电。
- (5) 通过 Programmer 下载 sof 到 SF-CY3 板中，观察液晶屏此时的显示。32 级的红色显示效果如图所示。



7.4 逻辑 (Verilog) 实例 11——基于 FPGA 内嵌 RAM 的 LCD 字符显示

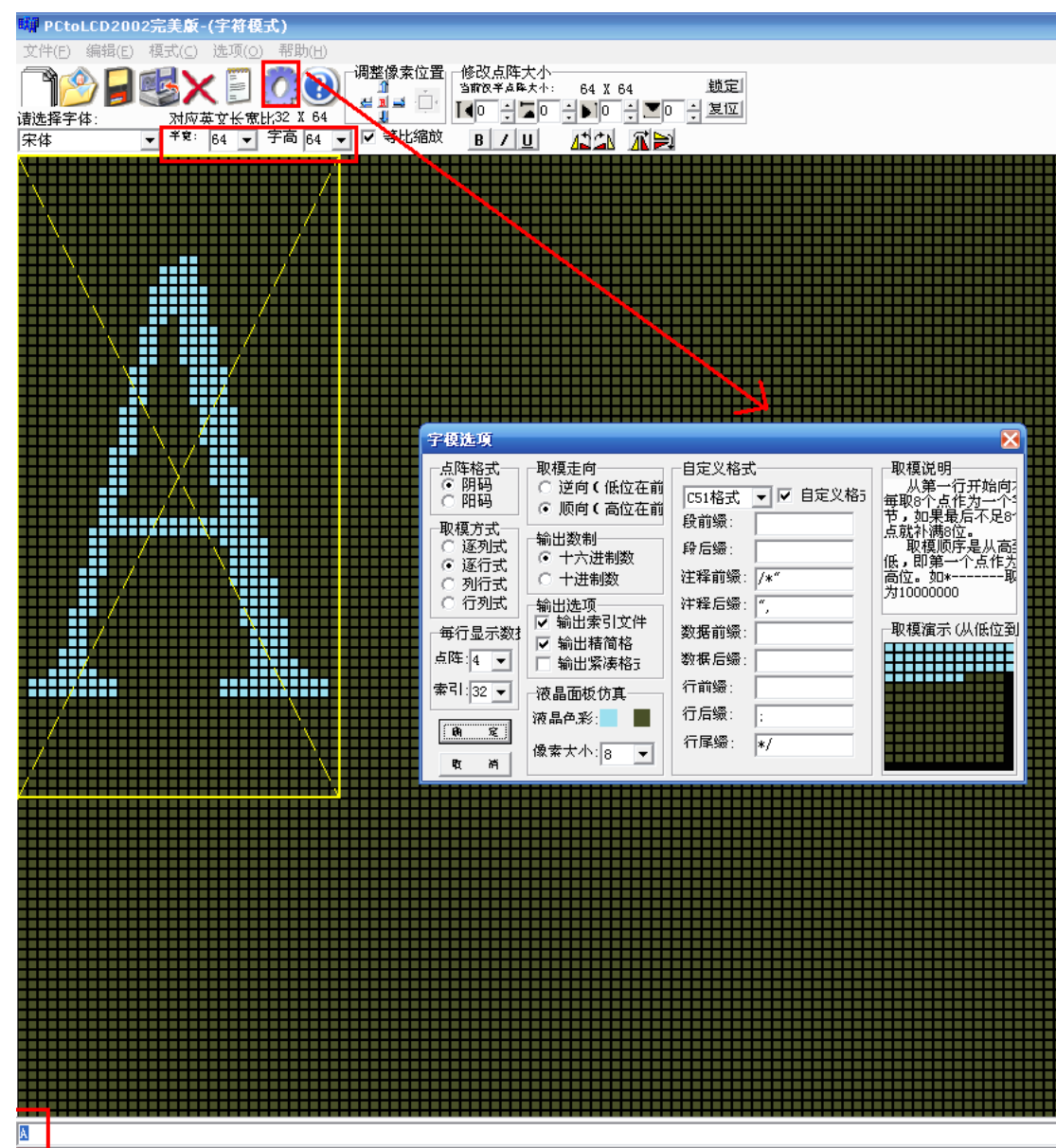
7.4.1 字符取模

要显示字符，首先需要获得字模数据，我们使用字模软件 PCtoLCD2002。该字模软件用 1bit 代表一个像素点，即它只能表示 2 种颜色的图像，当然不是仅仅局限于黑和白了，用户可以根据需要来决定这 1bit 数据 (0 或 1) 代表的色彩。

下面说明我们的设计中需要的字符是如何取模的，启动取模软件 PCtoLCD2002，点击菜单栏的“模式”，选择“字符模式”。再点击菜单栏的“选项”(或点击如图所示的齿轮图标)，在弹出的对话框中设置如图所示(行后缀为英文的“;”)。此外，在主界面中，我们设置字符宽度为 64*64 (实际上如果我们是给字符取模，它默认为 32*64)，在主界面下方的字模输入框中输入了大写字母 A，接着点击它右侧的“生成字模”按钮(图中没有示意)，则在输



出栏中出现了一大串 32bit 一行, 并且行后缀为 “;” 的字符, copy 他们, 后面会用得上滴。



32*64 点阵的字符 “A” 取模后的数据如下。实际上这些数据如果我们用二进制的 0 和 1 一位位的将他们排列开来, 则我们可以看到 1 可以排列出一个字母 “A” 出来。正是根据这个原理, 我们后面会每行 32 位的将他们送往液晶屏显示, 一共有 64 行这样的显示。

A(0)

```
00000000;  
00000000;  
00000000;  
00000000;  
00000000;  
00000000;
```

《圣经》箴言九 11 “敬畏耶和华是智慧的开端, 认识至胜者便是聪明。”



```
00000000;  
00000000;  
00000000;  
00000000;  
0000C000;  
0003C000;  
0003C000;  
0007E000;  
0007E000;  
0007E000;  
0006E000;  
000CF000;  
000CF000;  
000CF000;  
000CF000;  
00187800;  
00187800;  
00187800;  
00187800;  
00303C00;  
00303C00;  
00303C00;  
00303C00;  
00701C00;  
00601E00;  
00601E00;  
00601E00;  
00E00E00;  
00C00F00;  
00C00F00;  
00FFFF00;  
01FFFF00;  
01800F80;  
01800780;  
01800780;  
03800780;  
030007C0;  
030003C0;
```



```
030003C0;  
070003C0;  
060003E0;  
060001E0;  
060001E0;  
0E0001E0;  
0E0001F0;  
1F0001F8;  
7FC00FFE;  
7FC00FFE;  
00000000;  
00000000;  
00000000;  
00000000;  
00000000;  
00000000;  
00000000;  
00000000;  
00000000;  
00000000;  
00000000;/*"A", 0*/
```

7.4.2 字符显示原理

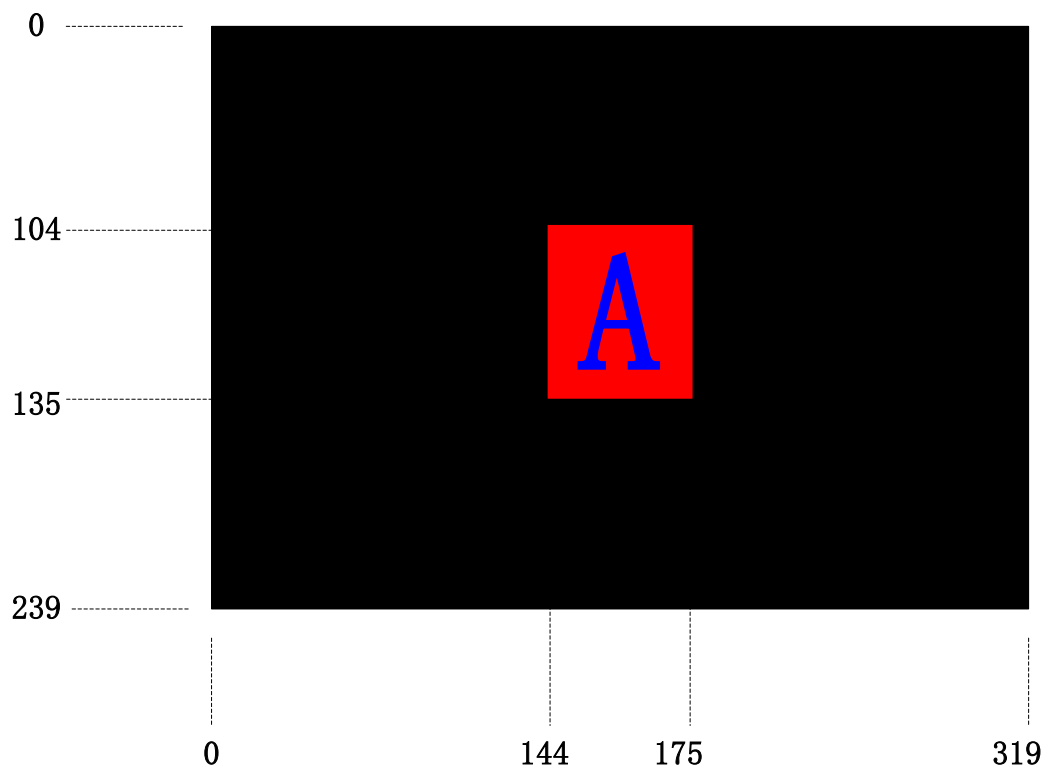
基于前面取到的字模数据,我们假定从屏幕的(0,0)坐标到(31,63)坐标区域(对应就是32*64的点阵)内显示字符。那么当坐标计数器刷新到(0,0)坐标点的时候我们就要相应判断第一行数据的bit31的值,然后决定送哪种色彩(0代表一种色彩,1代表另一种色彩)。当坐标计数器刷新到(1,0)坐标点的时候我们就要相应判断第一行数据的bit30的值……直到刷新到(31,0)时判断第一行数据的bit0的值,由此完成了首行字模数据的译码。往后的译码都和首行类似,64行字模数据寻址完毕后,大写字母“A”便出现在我们的屏幕上。

当然了,为了显示得美观,我们特意将这个32*64的大写字母“A”放到了320*240的LCD的正中央。那么它的坐标就不是(0,0)到(31,63)的区域了。而是(144,104)到(175,135)这个区域。我们这个实例最终要显示的效果如图所示。在(144,104)到(175,135)这个区域内,字符“A”以蓝(16'h001f)字红(16'hf800)底显示,LCD的其他显示区域则

《圣经》箴言九 11 “敬畏耶和华是智慧的开端,认识至胜者便是聪明。”

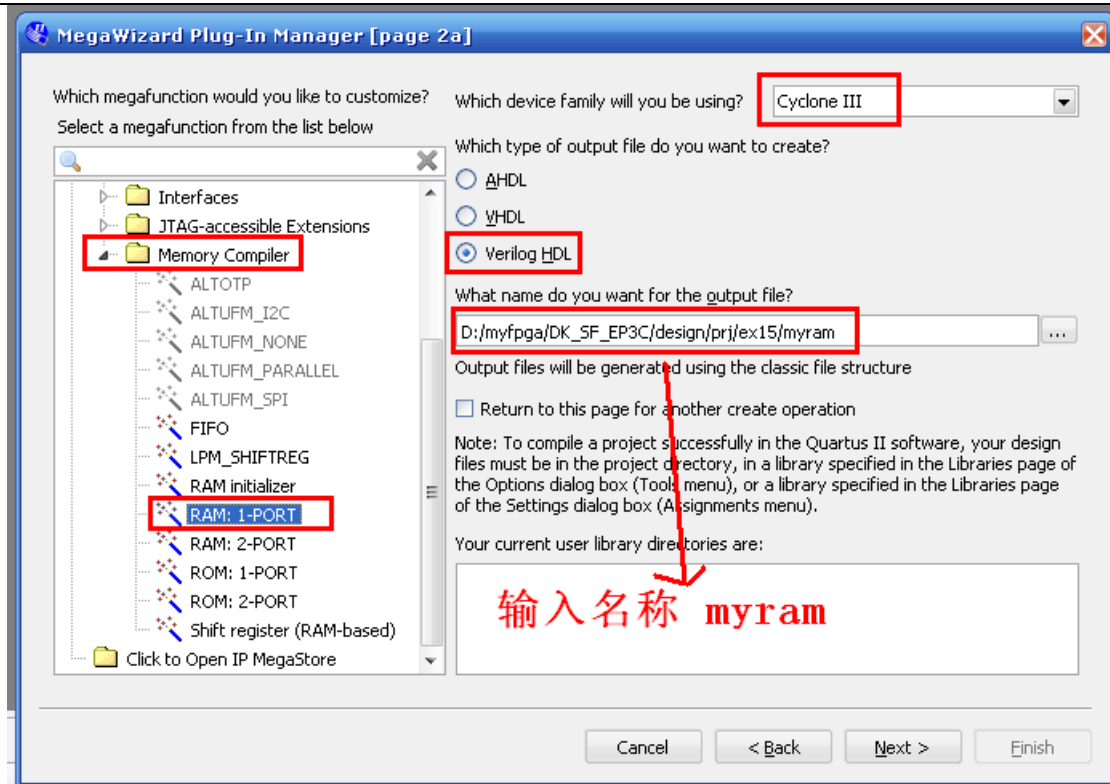


为黑色 (16'h0000)。

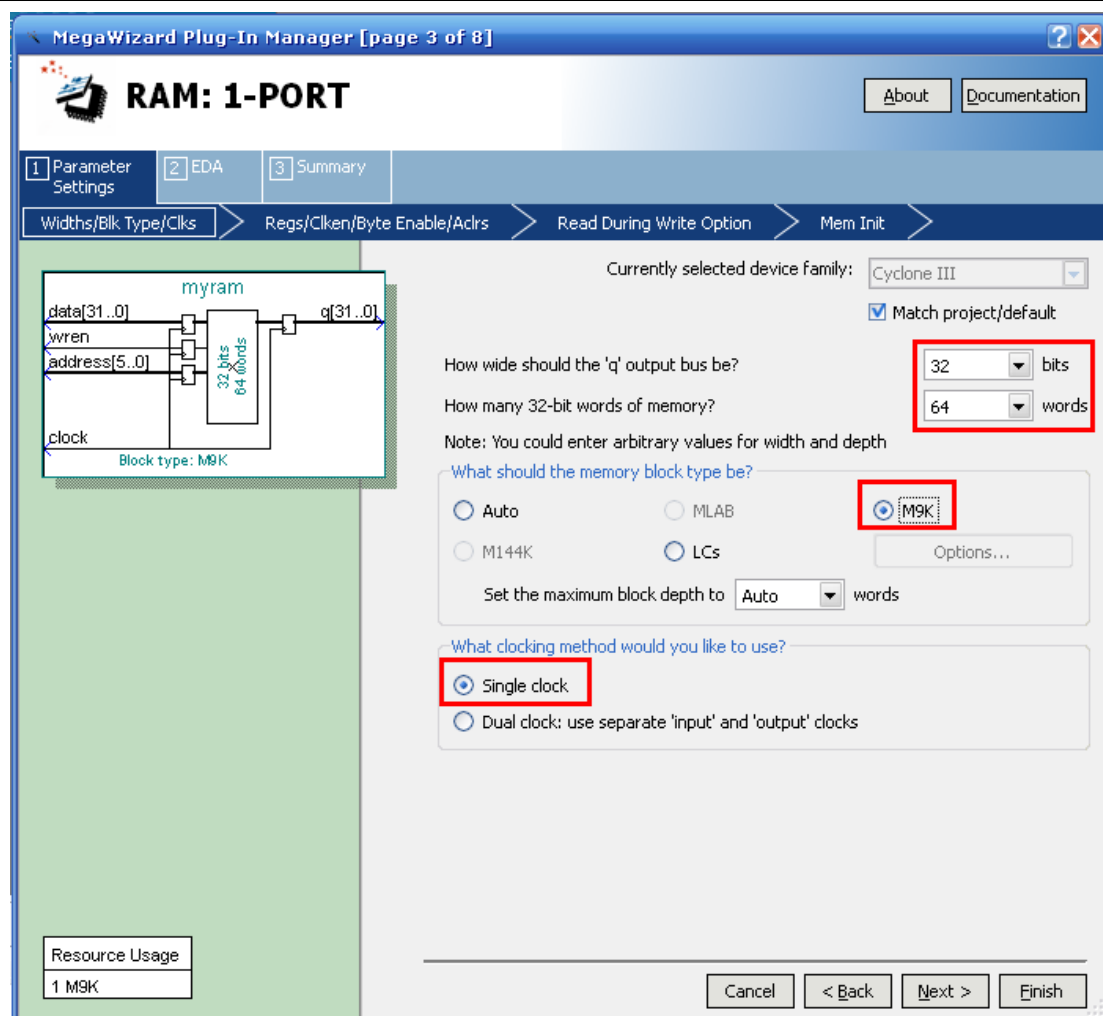


7.4.3 内嵌 RAM 的配置和例化

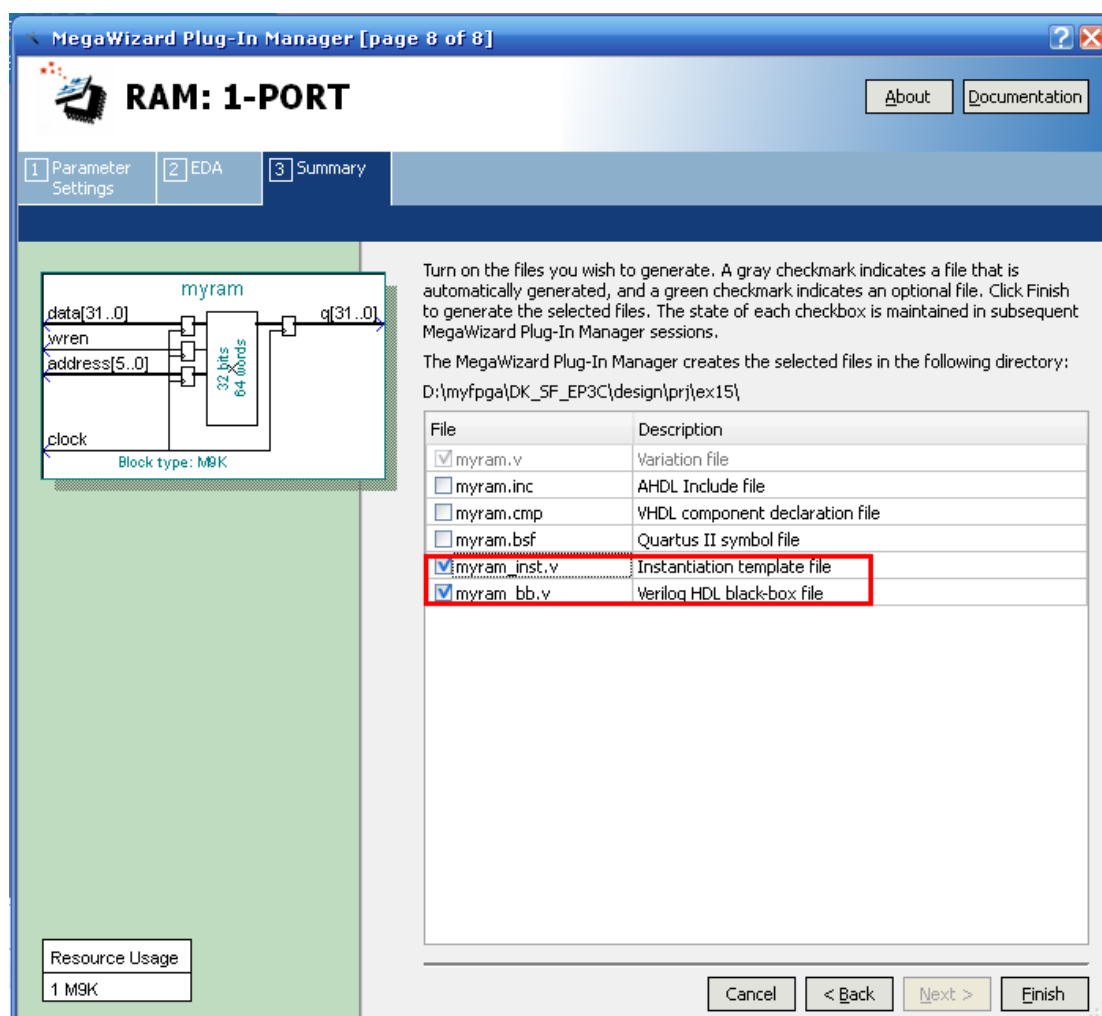
这里我们要在 Quartus II 工程中配置一个 32bit*64 的单口 RAM, 用于存储前面的取模数据。首先点击菜单栏的 Tools→MegaWizard Plug-In Manager, new 一个 megafunction, 然后在 page 2a 中, 做如图所示的设置, 选择我们需要的 RAM:1-PORT, 设置其名称为 myram (前面的路径为当前工程所在路径, 具体请参考“工程实践”一节)。完成后, 进入下一步。



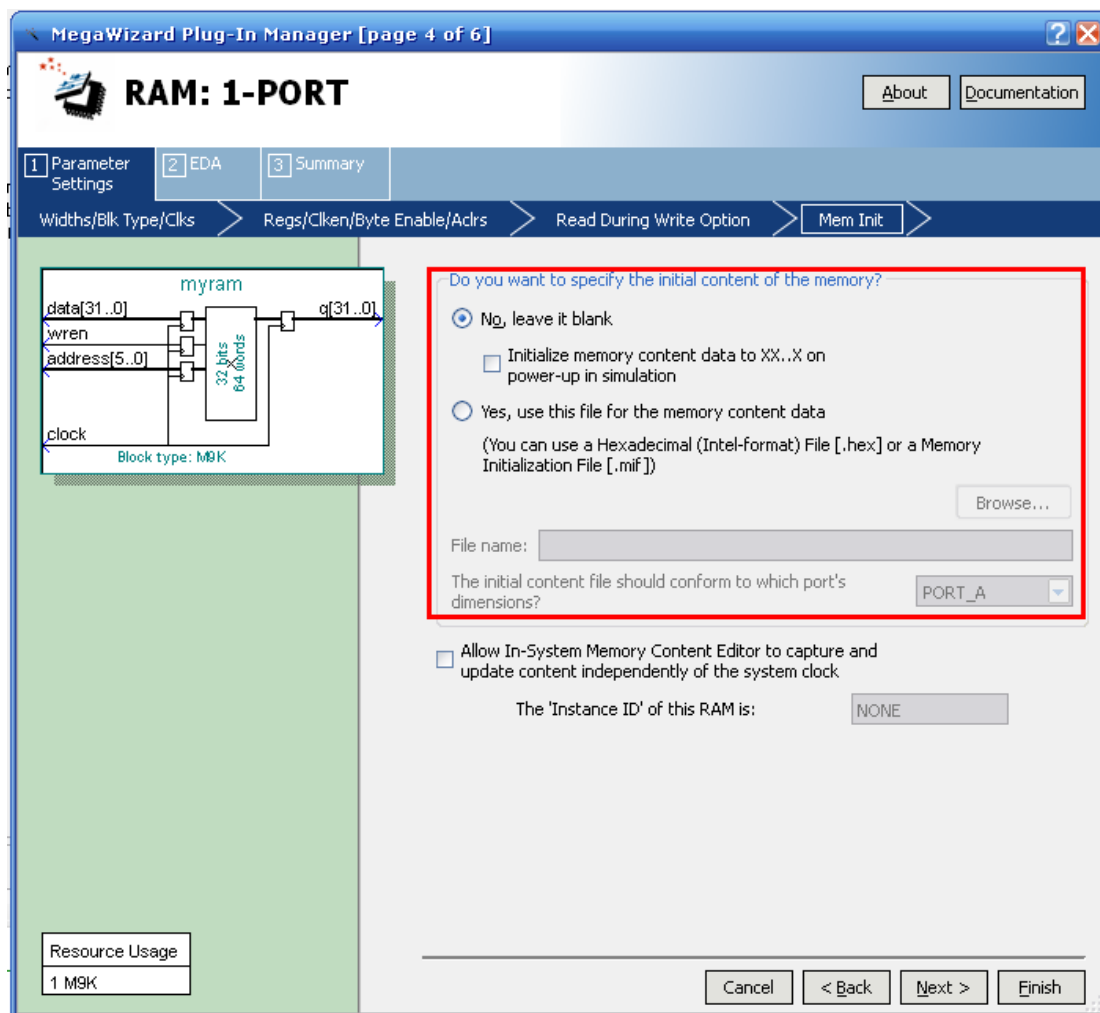
在 Parameter Settings 的 Widths/Blk Type/Clocks 页面中, 配置数据位宽为 32bit, 数据个数为 64, 实现存储器类型为 Cyclone III 系列器件内嵌的 M9K 存储块, 时钟驱动方式为 Single clock。



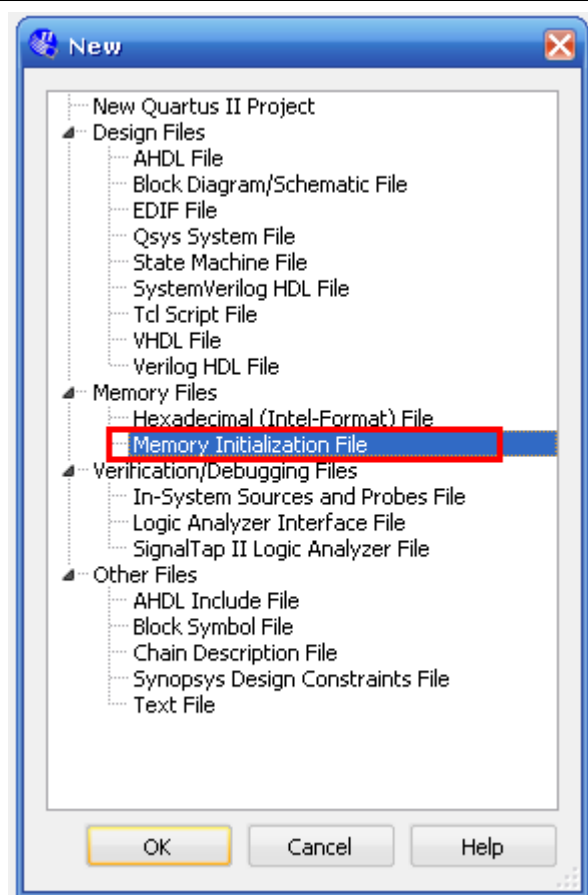
其他页面暂时都采用默认的设置，直接到最后一个 Summary 页面，勾选 myram_inst.v 和 myram_bb.v 两个文件，表示生成这个 RAM 配置时，同时产生这两个文件。点击 Finish 完成设置。



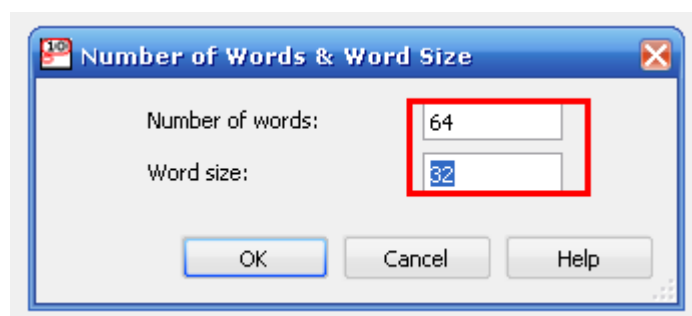
对了，不是说会把前面 copy 并且还 hold 中的字模数据放到这个 RAM 中吗？怎么好像还没有 paste 这个动作啊，不着急，马上说，其实在刚才配置 RAM 的时候，如果细心的朋友一定已经注意到了，有一个 Mem Init 页面，它不就是用于初始化 RAM 的嘛，Yes，恭喜你，回答正确。如图所示，默认情况下，“Do you want to specify the initial of the memory?”的设置是“No, Leave it blank”。我们看后面一个 Yes 选择，如果点击它，那么后面的 File name 则高亮并且可以被设置，它需要选择一个.hex 或.mif 文件。我们好像还没有啊？是的，别担心，马上就“变”一个出来。



回到 Quartus II 中, 我们点击 **File→New**, 弹出窗口如图所示, 选择 **Memory Files** 下的 **Memory Initialization File** (当然了, 创建 **Hexadecimal(Intel-Format) File** 也是可以的), 点击 **OK**。



此时自动创建了一个.mif 文件，首先弹出了一个设置位宽和字符数量的对话框，我们设置 32bit 位宽，64 个数据。点击 OK。

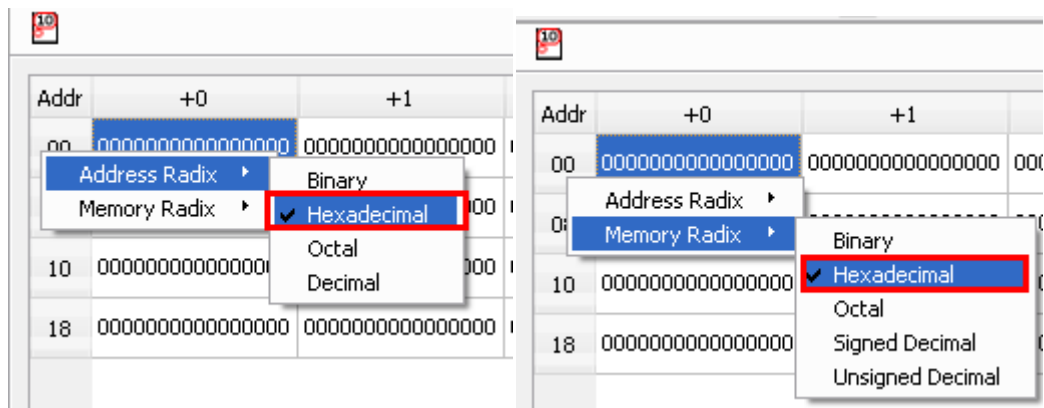


此时，.mif 的主编辑界面出现了，如图所示。真的创建了吗？非也，此时你在工程目录下是找不到这个所谓的 Mif4.mif 文件的，因为它只是一个临时文件，名称都是临时的，怎么会出现我们的工程中。下面要做一件事，先把这个文件保存下来。怎么保存，你看你个保存按钮都是灰色的，连个保存的机会都不给。还真是这样，不过大家如果随便在编辑界面的数据中更改数值，然后那个保存按钮就高亮了，意味着可以保存了，不信就试试看。

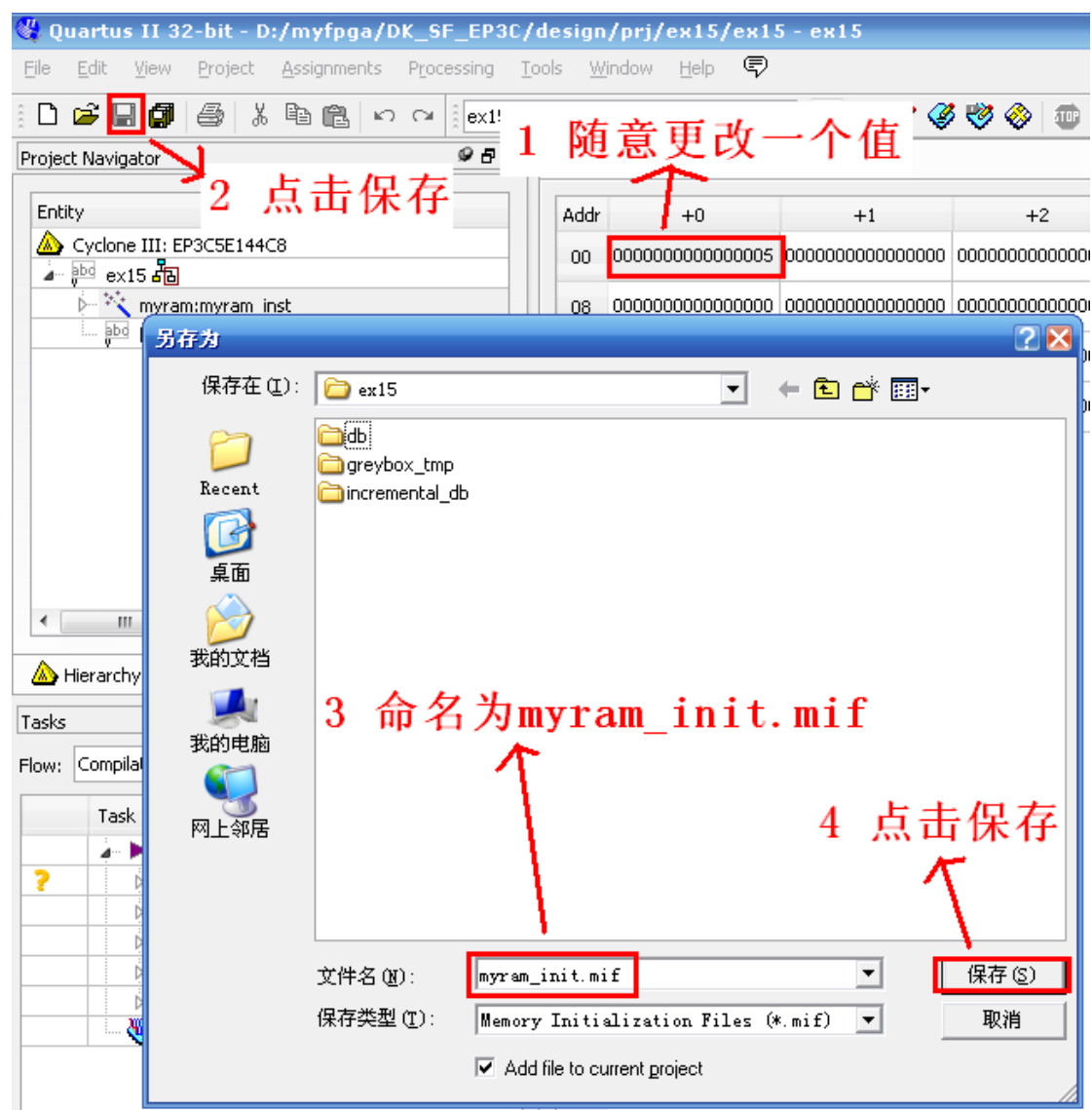


Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
00	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000
08	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000
10	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000
18	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000

大家别急着保存，跟着大部队走。如图所示，先请大家在 **Addr** 下面的任意数值处点击右键，然后在弹出的菜单中分别设置 **Address Radix** 和 **Memory Radix** 为 **Hexadecimal** 选项，默认可不一定是这个状态，还是请大家先确认下，免得后面麻烦。也就是说，这里我们把地址和数据显示都设置为 16 进制模式，主要还是为了后续编辑的方便。



下一步，大家可以在这文件随意更改一个数值，然后点击保存，命名为 **myram_init.mif**，如图所示。



完成这一步后,大家当然可以把前面 copy 的 64 个 32bit 数据挨个填写到这个编辑窗口中,不过这个效率不高,不推荐,我们先关闭这个编辑窗口,然后又 txt 或者其他编辑器的模式下打开 mif 文件。如图所示,注意 CONTENT BEGIN 和 EDN 之间的数据,不就是我们输入的数据吗?对的,聪明人可能已经明白了填充这个文件的快速办法,没错直接按照这个格式把 64 个数据填充进去即可。



```
myram_init.mif
1  -- Copyright (C) 1991-2012 Altera Corporation
2  -- Your use of Altera Corporation's design tools, logic functions
3  -- and other software and tools, and its AMPP partner logic
4  -- functions, and any output files from any of the foregoing
5  -- (including device programming or simulation files), and any
6  -- associated documentation or information are expressly subject
7  -- to the terms and conditions of the Altera Program License
8  -- Subscription Agreement, Altera MegaCore Function License
9  -- Agreement, or other applicable license agreement, including,
10 -- without limitation, that your use is for the sole purpose of
11 -- programming logic devices manufactured by Altera and sold by
12 -- Altera or its authorized distributors. Please refer to the
13 -- applicable agreement for further details.
14
15 -- Quartus II generated Memory Initialization File (.mif)
16
17 WIDTH=64;
18 DEPTH=32;
19
20 ADDRESS_RADIX=HEX;
21 DATA_RADIX=HEX;
22
23 CONTENT BEGIN
24     00 : 00000000000000000005;
25     [01..1F] : 00000000000000000000;
26 END;
```

最终编辑好的 mif 文件数据如下。

```
CONTENT BEGIN
    [00..09] : 00000000;
    0A : 0000C000;
    0B : 0003C000;
    0C : 0003C000;
    0D : 0007E000;
    0E : 0007E000;
    0F : 0007E000;
    10 : 0006E000;
    11 : 000CF000;
    12 : 000CF000;
    13 : 000CF000;
    14 : 000CF000;
    15 : 00187800;
    16 : 00187800;
    17 : 00187800;
```



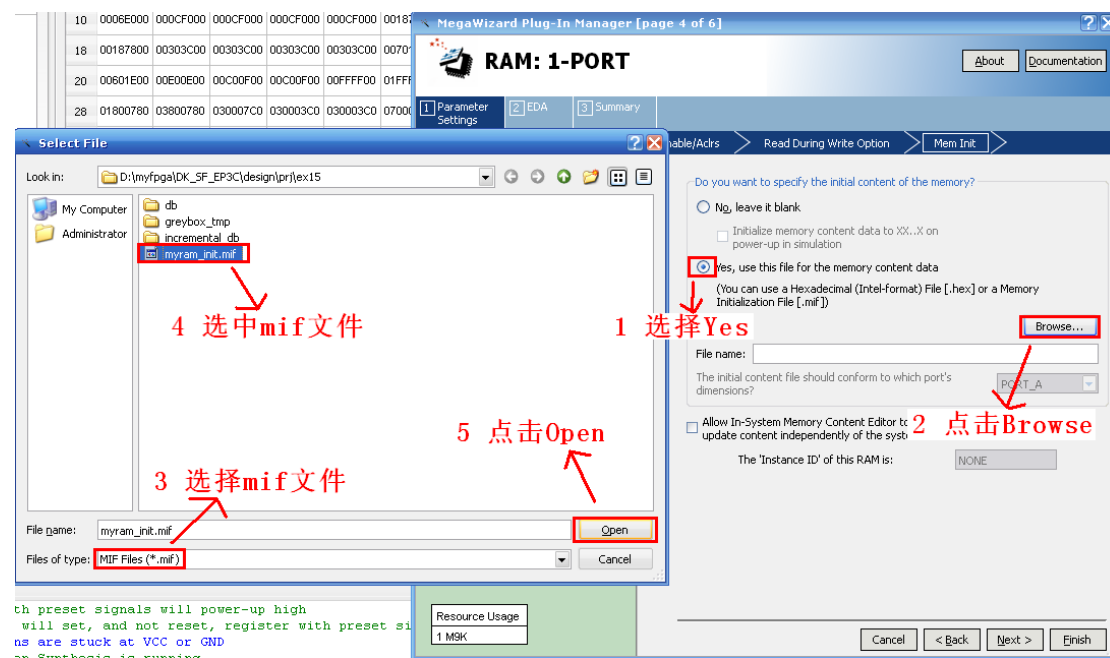
```
18 : 00187800;
19 : 00303C00;
1A : 00303C00;
1B : 00303C00;
1C : 00303C00;
1D : 00701C00;
1E : 00601E00;
1F : 00601E00;
20 : 00601E00;
21 : 00E00E00;
22 : 00C00F00;
23 : 00C00F00;
24 : 00FFFF00;
25 : 01FFFF00;
26 : 01800F80;
27 : 01800780;
28 : 01800780;
29 : 03800780;
2A : 030007C0;
2B : 030003C0;
2C : 030003C0;
2D : 070003C0;
2E : 060003E0;
2F : 060001E0;
30 : 060001E0;
31 : 0E0001E0;
32 : 0E0001F0;
33 : 1F0001F8;
34 : 7FC00FFE;
35 : 7FC00FFE;
[36..3F] : 00000000;
END;
```

接着我们可以重新在 Quartus II 中打开这个 mif 文件, 如图所示, 我们刚刚输入的数据已经生效了。

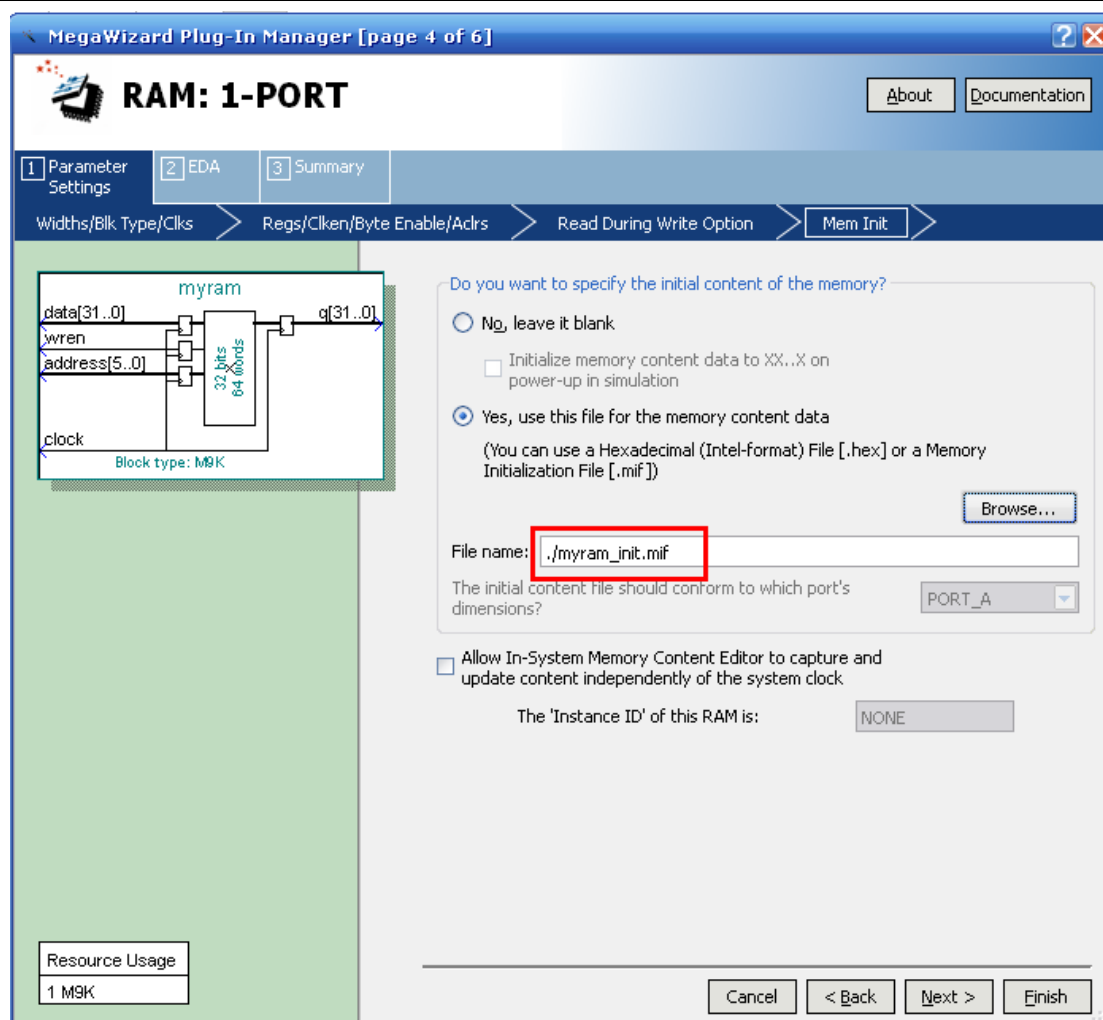


myram_init.mif									
Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
00	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
08	00000000	00000000	0000C000	0003C000	0003C000	0007E000	0007E000	0007E000
10	0006E000	000CF000	000CF000	000CF000	000CF000	00187800	00187800	00187800
18	00187800	00303C00	00303C00	00303C00	00303C00	00701C00	00601E00	00601E00
20	00601E00	00E00E00	00C00F00	00C00F00	00FFFF00	01FFFF00	01800F80	01800780
28	01800780	03800780	030007C0	030003C0	030003C0	070003C0	060003E0	060001E0
30	060001E0	0E0001E0	0E0001F0	1F0001F8	7FC00FFE	7FC00FFE	00000000	00000000
38	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

OK, 回到 RAM 的配置中, 如图所示, 我们选择 Yes 选项, 然后添加新建的 myram_init.mif 文件, 如图所示。

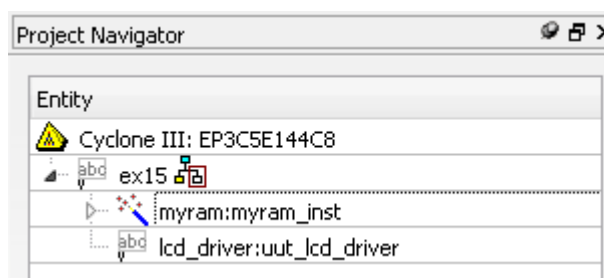


完成后, 我们看到 File name 后面出现了新添加进来的 mif 文件名。此时, 我们可以点自己 Finish 完成 RAM 的配置。



7.4.3 Verilog 代码

在工程导航窗口中，我们可以看到 3 个工程模块，一个顶层模块和两个子模块，其中一个子模块是前面例化的 RAM，不需要我们额外创建 verilog 代码。但是顶层文件 ex15.v 和另一个子模块 lcd_driver.v 模式是我们创建。



在顶层模块 ex15.v 中，我们对 RAM 和 lcd_driver.v 模块进行例化，并对他们之间连接的信号进行申明。代码如下。

《圣经》箴言九 11 “敬畏耶和华是智慧的开端，认识至胜者便是聪明。”



```
module ex15(
    clk,rst_n,
    lcd_en,lcd_clk,lcd_hsy,lcd_vsy,lcd_db_r,lcd_db_g,lcd_db_b
);
input clk;      //25MHz
input rst_n;    //低电平复位
    // FPGA 与 LCD 接口信号
output lcd_en;  //背光使能信号, 高有效
output lcd_clk; //时钟信号
output lcd_hsy; //行同步信号
output lcd_vsy; //场同步信号
output[4:0] lcd_db_r;
output[5:0] lcd_db_g;
output[4:0] lcd_db_b;

//-----
    //LCD 与字符存储 RAM 的接口
wire[31:0] ram_db; //RAM 数据总线
wire[5:0] ram_ab;  //RAM 地址总线

//-----
    //RAM 例化, 该 RAM 中存储一个 32*64 的 ASCII 字符的字模
myram  myram_inst (
    .address ( ram_ab ),
    .clock ( clk ),
    .data ( 32'd0 ),
    .wren ( 1'b0 ),
    .q ( ram_db )
);

//-----
    //LCD 显示驱动模块
lcd_driver  uut_lcd_driver(
    .clk(clk),
    .rst_n(rst_n),
    .lcd_en(lcd_en),
    .lcd_clk(lcd_clk),
    .lcd_hsy(lcd_hsy),
```



```
        .lcd_vsy(lcd_vsy),  
        .lcd_db_r(lcd_db_r),  
        .lcd_db_g(lcd_db_g),  
        .lcd_db_b(lcd_db_b),  
        .ram_db(ram_db),  
        .ram_ab(ram_ab)  
    );  
  
endmodule
```

RAM 的例化中, 我们看到有 5 个接口, 他们的定义如下。

信号名称	方向	位宽	描述
clock	input	1	RAM 读写的同步时钟信号
wren	input	1	RAM 写入使能信号, 高电平表示写入
address	input	6	RAM 地址总线
data	input	32	RAM 写入数据总线
q	output	32	RAM 读出数据总线

这个实例中, 我们只是读取初始化的 RAM 数据, 不会对 RAM 进行写操作。所以我们只要把 wren 信号拉低, 使其始终处于读状态即可。当 wren 为读状态时, address 就是此时 RAM 读地址, 它决定了当前 RAM 输出 q 信号的值。我们后面的实例中就是通过更改 address 来改变需要读取的 RAM 数据。

lcd_driver.v 文件中, 有和前面两节一样的对 LCD 的驱动代码, 也有本节特别的需要开辟的一个现实坐标区域的色彩显示判断处理逻辑。大家可以仔细研读代码, 体会其中的设计原理。

```
module lcd_driver(  
    clk, rst_n,  
    lcd_en, lcd_clk, lcd_hsy, lcd_vsy, lcd_db_r, lcd_db_g, lcd_db_b,  
    ram_db, ram_ab  
);  
  
input clk;        //25MHz  
input rst_n;      //低电平复位  
    // FPGA 与 LCD 接口信号  
output lcd_en;    //背光使能信号, 高有效  
output lcd_clk;   //时钟信号  
output lcd_hsy;   //行同步信号
```

《圣经》箴言九 11 “敬畏耶和华是智慧的开端, 认识至胜者便是聪明。”



```
output lcd_vsy; //场同步信号
output[4:0] lcd_db_r;
output[5:0] lcd_db_g;
output[4:0] lcd_db_b;

    //LCD 与字符存储 RAM 的接口
input[31:0] ram_db; //RAM 数据总线
output[5:0] ram_ab; //RAM 地址总线

//-----

assign lcd_en = 1'b1;

//-----

//lcd_clk 时钟周期为 160ns (6.25MHz), 即 4 个 25MHz 的时钟周期
reg[1:0] sft_cnt;

always @(posedge clk or negedge rst_n)
    if(!rst_n) sft_cnt <= 2'd0;
    else sft_cnt <= sft_cnt+1'b1;

assign lcd_clk = sft_cnt[1];    //0-1:low, 2-3:high

wire dchange = (sft_cnt == 2'd2);    //数据变化标志位, 高有效

//-----

//坐标计数
//x = 0-407; y = 0-261
reg[8:0] x_cnt; //x 计数器
reg[8:0] y_cnt; //y 计数器

always @(posedge clk or negedge rst_n)
    if(!rst_n) x_cnt <= 9'd0;
    else if(dchange) begin
        if(x_cnt == 9'd407) x_cnt <= 9'd0;
        else x_cnt <= x_cnt+1'b1;
    end

always @(posedge clk or negedge rst_n)
    if(!rst_n) y_cnt <= 9'd0;
```



```
        else if(dchange && (x_cnt == 9'd407)) begin
            if(y_cnt == 9'd261) y_cnt <= 9'd0;
            else y_cnt <= y_cnt+1'b1;
        end

//-----
//有效显示标志位产生
reg valid_yr;    //行显示有效信号

        //行显示有效信号
always @ (posedge clk or negedge rst_n)
    if(!rst_n) valid_yr <= 1'b0;
    else if(y_cnt == 9'd18) valid_yr <= 1'b1;
    else if(y_cnt == 9'd258) valid_yr <= 1'b0;

reg validr,valid;

always @ (posedge clk or negedge rst_n)
    if(!rst_n) validr <= 1'b0;
    else if((x_cnt == 9'd67) && valid_yr) validr <= 1'b1;
    else if((x_cnt == 9'd387) && valid_yr) validr <= 1'b0;

always @ (posedge clk or negedge rst_n)
    if(!rst_n) valid <= 1'b0;
    else valid <= validr;

//-----
// LCD 场同步, 行同步信号
reg lcd_hsy_r, lcd_vsy_r;    //同步信号

always @ (posedge clk or negedge rst_n)
    if(!rst_n) lcd_hsy_r <= 1'b1;
    else if(x_cnt == 9'd0) lcd_hsy_r <= 1'b0;    //产生 lcd_hsy 信号
    else if(x_cnt == 9'd30) lcd_hsy_r <= 1'b1;

always @ (posedge clk or negedge rst_n)
    if(!rst_n) lcd_vsy_r <= 1'b1;
    else if(y_cnt == 9'd0) lcd_vsy_r <= 1'b0;    //产生 lcd_vsy 信号
```



```
        else if(y_cnt == 9'd3) lcd_vsy_r <= 1'b1;

assign lcd_hsy = lcd_hsy_r;
assign lcd_vsy = lcd_vsy_r;

//-----
//RAM 地址产生
assign ram_ab = y_cnt-(9'd18+9'd88);

//-----
// LCD 色彩信号产生
reg[4:0] tmp_cnt; //0-31 计数, 对应一行的 32 个有效显示位

always @(posedge clk or negedge rst_n)
    if(!rst_n) tmp_cnt <= 5'd0;
    else if(!validr) tmp_cnt <= 5'd0;
    else if((x_cnt >= (9'd67+9'd143)) && (x_cnt <= (9'd67+9'd174)) && dchange) begin
        tmp_cnt <= tmp_cnt+1'b1;
    end

reg[15:0] lcd_db_rgb; // LCD 色彩显示寄存器

always @ (posedge clk or negedge rst_n)
    if(!rst_n) lcd_db_rgb <= 16'd0;
    else if((y_cnt >= (9'd18+9'd88)) && (y_cnt <= (9'd18+9'd151))
        && (x_cnt >= (9'd67+9'd144)) && (x_cnt <= (9'd67+9'd175))) begin
        if(dchange) begin //数字显示区域
            if(ram_db[tmp_cnt]) lcd_db_rgb <= 16'h001f; //显示蓝色
            else lcd_db_rgb <= 16'hf800; //显示红色
        end
    else ;
    end

    else lcd_db_rgb <= 16'd0;

//r, g, b 控制液晶屏颜色显示
assign lcd_db_r = valid ? lcd_db_rgb[15:11]:5'd0;
assign lcd_db_g = valid ? lcd_db_rgb[10:5]:6'd0;
assign lcd_db_b = valid ? lcd_db_rgb[4:0]:5'd0;
```



```
endmodule
```

7.4.4 工程实践

- 新建工程，命名为 `ex15`，把这个工程放在专门的文件夹下，其他设置参考前面章节。
- 新建 Verilog 源文件，命名为 `ex15`，输入前面给出的设计代码。再新建一个 Verilog 源文件，命名为 `lcd_driver`，输入前面给出的设计代码。使用 ModelSim-Altera 对设计代码进行仿真验证。
- 综合编译后进行管脚分配，本例程的管脚分配如下所示。

Node Name	Direction	Location
clk	Input	PIN_22
lcd_clk	Output	PIN_60
lcd_db_b[4]	Output	PIN_77
lcd_db_b[3]	Output	PIN_79
lcd_db_b[2]	Output	PIN_80
lcd_db_b[1]	Output	PIN_83
lcd_db_b[0]	Output	PIN_84
lcd_db_g[5]	Output	PIN_71
lcd_db_g[4]	Output	PIN_72
lcd_db_g[3]	Output	PIN_73
lcd_db_g[2]	Output	PIN_74
lcd_db_g[1]	Output	PIN_75
lcd_db_g[0]	Output	PIN_76
lcd_db_r[4]	Output	PIN_66
lcd_db_r[3]	Output	PIN_67
lcd_db_r[2]	Output	PIN_68
lcd_db_r[1]	Output	PIN_69
lcd_db_r[0]	Output	PIN_70
lcd_en	Output	PIN_85
lcd_hsy	Output	PIN_65
lcd_vsy	Output	PIN_64
rst_n	Input	PIN_91

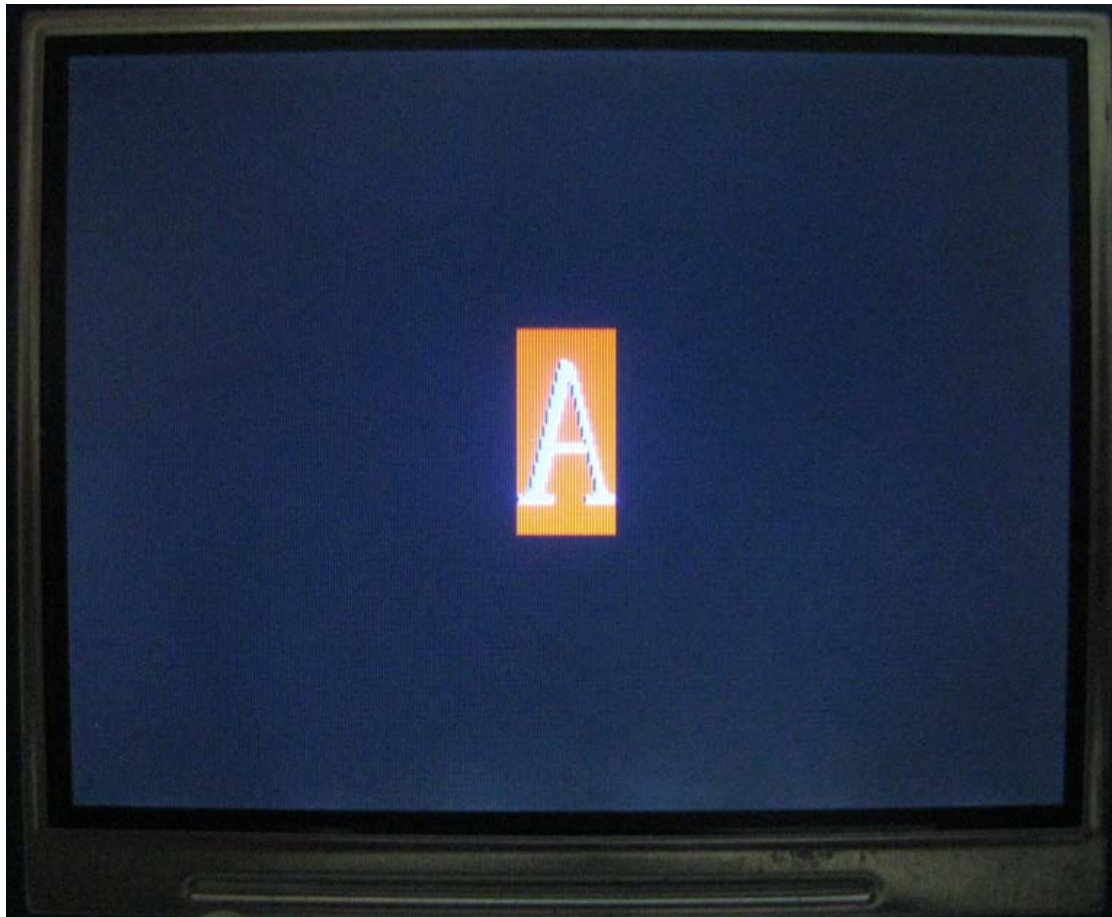
- 参考前面章节，打开 TimeQuest，我们新建一个 SDC 文件，然后对时钟 `clk` 做约束，约束脚本如下：

```
create_clock -name {SYSCLK} -period 40.000 -waveform { 0.000 20.000 } [get_ports {clk}]
```

- 我们对工程进行全编译，不仅要让刚刚添加的时钟约束生效，也要生成可以下载到 FPGA 芯片中的配置文件。



- (6) 连接好 SF-LCD 子板和 SF-CY3 核心板, 连接好下载线以及电源, 给板子上电。
- (7) 通过 Programmer 下载 sof 到 SF-CY3 板中, 观察液晶屏此时的显示。如图所示。

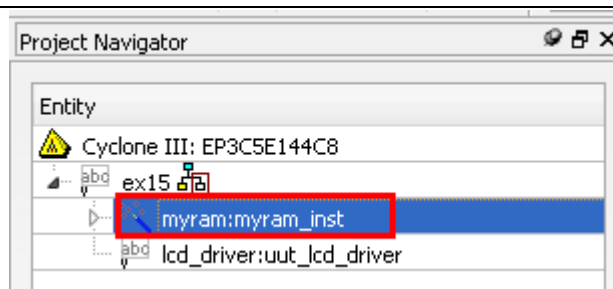




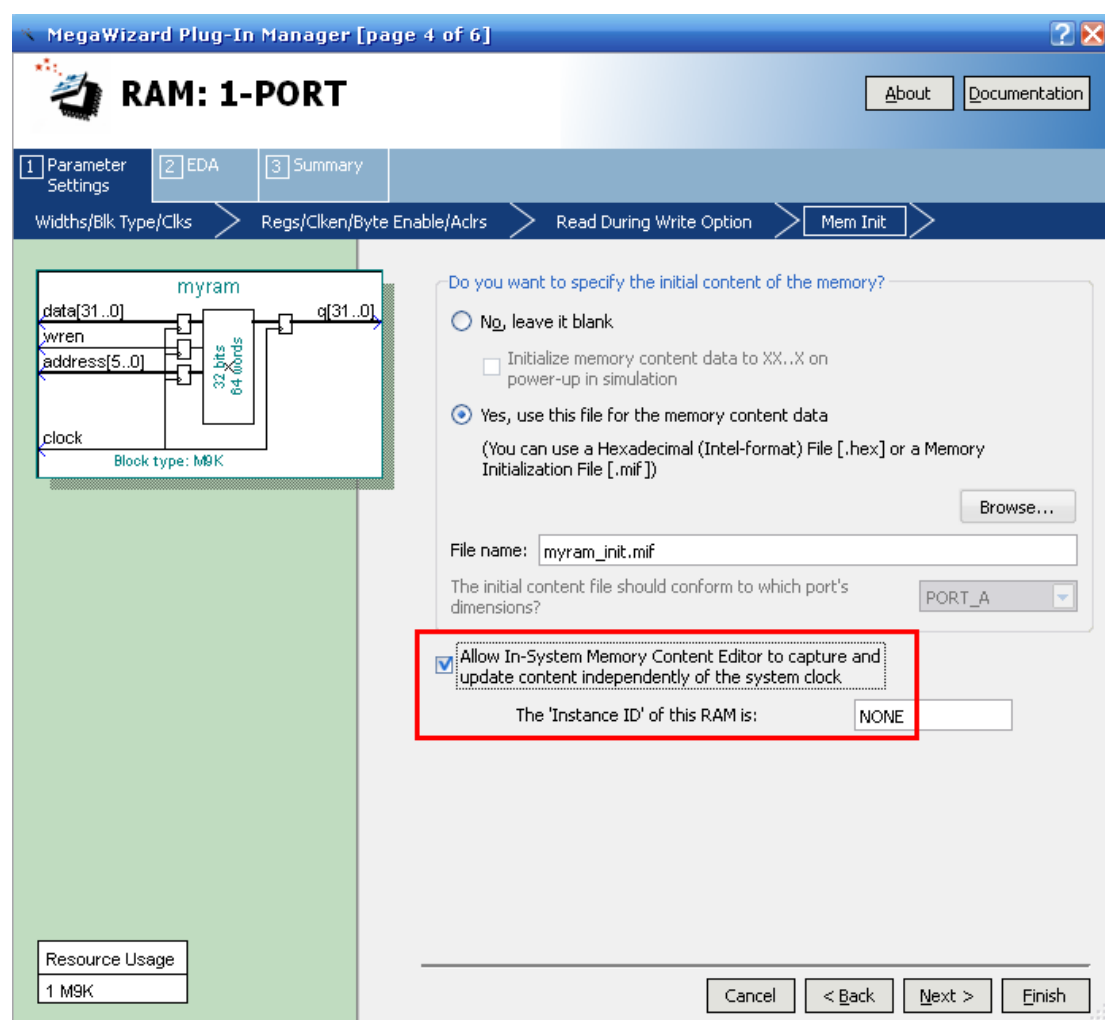
7.5 逻辑（Verilog）实例 12——基于 In-System Memory Content Editor 的 LCD 实时显示字符更改

Quartus II 中的 In-System Memory Content Editor 是用于对工程中已经例化的内嵌 RAM 进行在线编辑的工具，它非常适合调试过程的使用，有时候我们不仅是想参考当前 RAM 中的数据，甚至希望更改它的值，这个调试工具都能够实现。本节我们就要利用上一节已经例化的 RAM，来实时的更改它的 RAM 值，然后看看显示的字符是否就发生了变化。

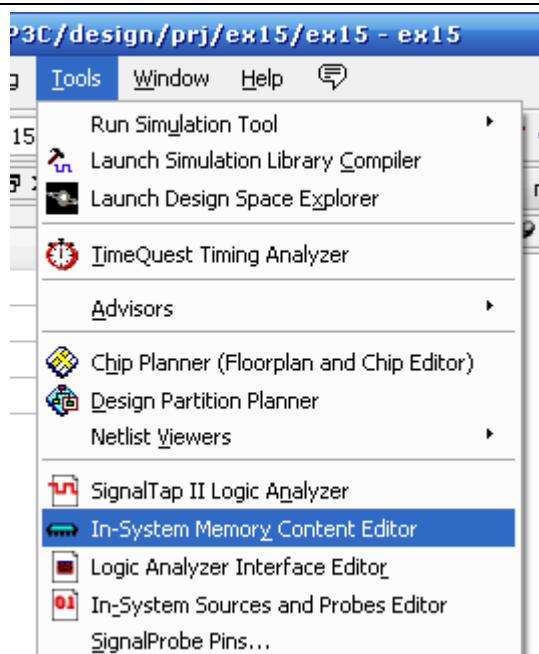
首先打开上一节的工程 ex15，双击打开工程导航窗口的 myram。



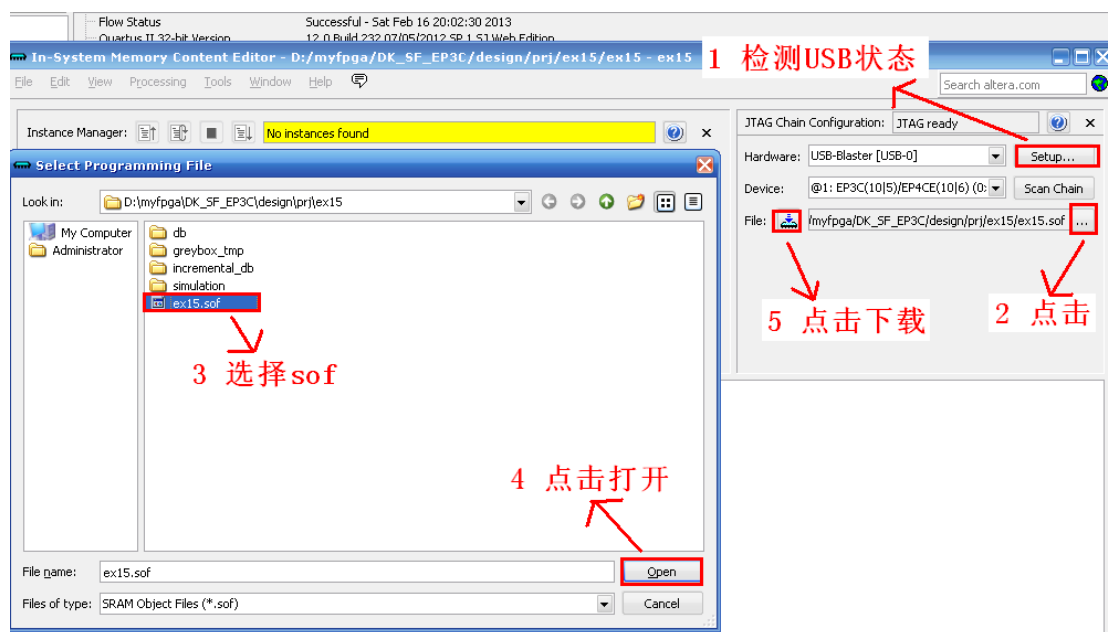
请大家确认在 Mem Init 页面的下方有一个 Allow In-System Memory Content Editor to capture and update content independently of the system clock 选项被勾选上了, 勾选它以后, 就意味着在我们的 In-System Memory Content Editor 调试工具中可以实时的 更改当前 RAM 中的数据。设置好以后, 重新编译整个工程。



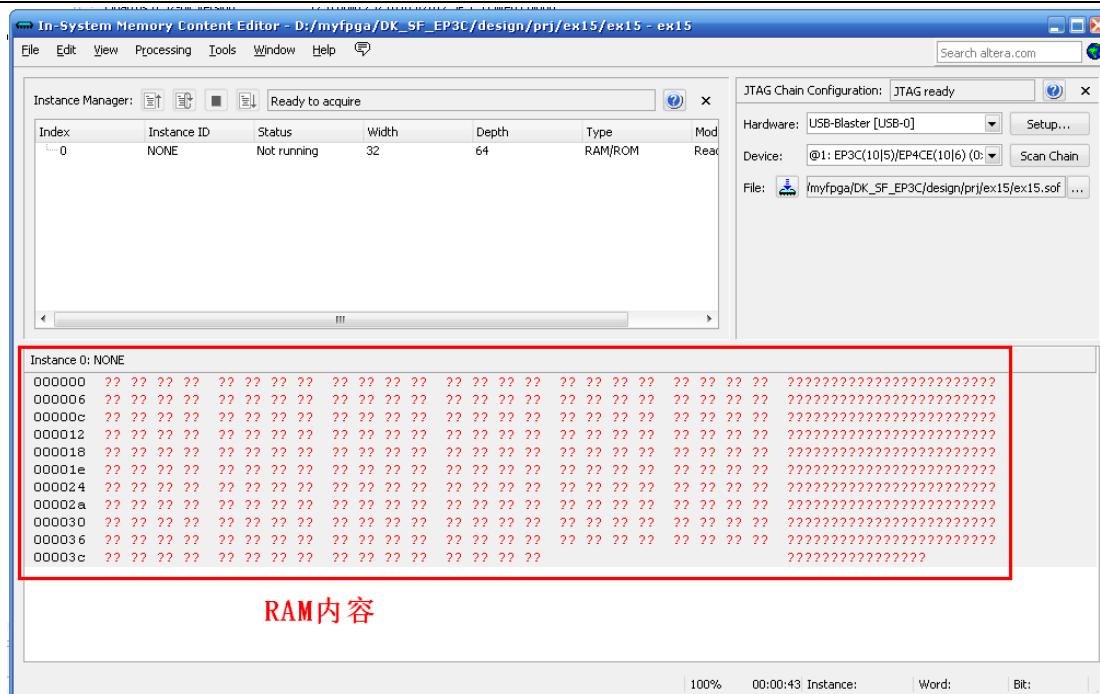
接着点击菜单栏的 Tools→In-System Memory Content Editor, 如图所示。



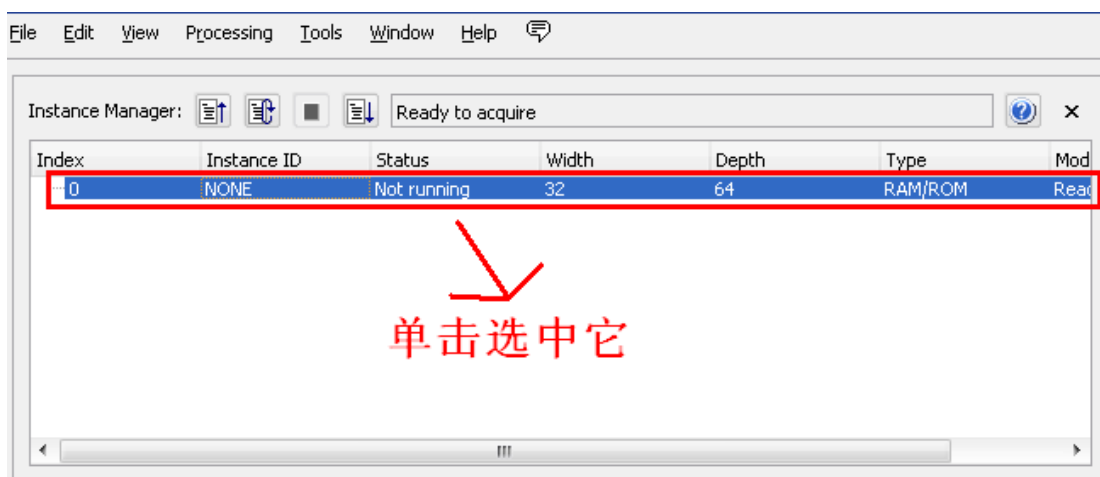
打开 In-System Memory Content Editor 界面, 首先点击右侧的 Setup 建立 USB Blaster 连接(这之前请先连接好 SF-CY3 和 SF-LCD 板, 连接好下载线和电压, 给板子上电)。接着点击 File 最后面的按钮加载当前工程的 sof 文件, 最后点击挨着 File 的按钮开始下载。



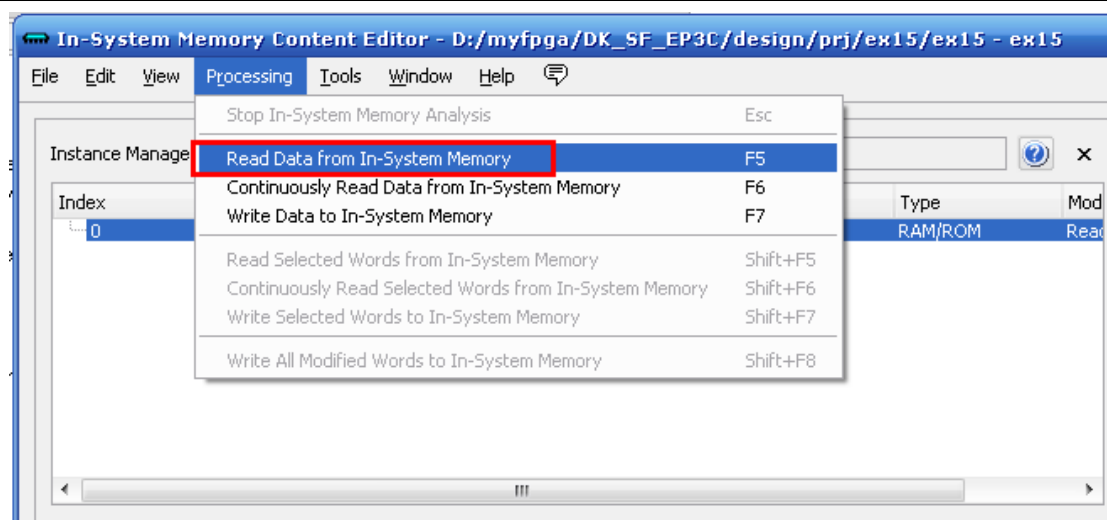
下载完成后, 如上一个实例所示, 在 LCD 中央有一个蓝字红底的字母“A”显示出来了。此时, 我们看到整个 In-System Memory Content Editor 界面如图所示, 在下方有 RAM 的内容, 此时都为??。



在菜单栏里,有按钮操作可以执行 RAM 的在线读写。我们首先读 RAM 的内容,先选择如图所示的 RAM 选项,因为在一个工程中,可能会有多个 RAM,那么这里给出了 RAM 选项是为了选择区别当前要操作的 RAM。



接着点击 In-System Memory Content Editor 菜单栏的 Processing→Read Data from In-System Memory。



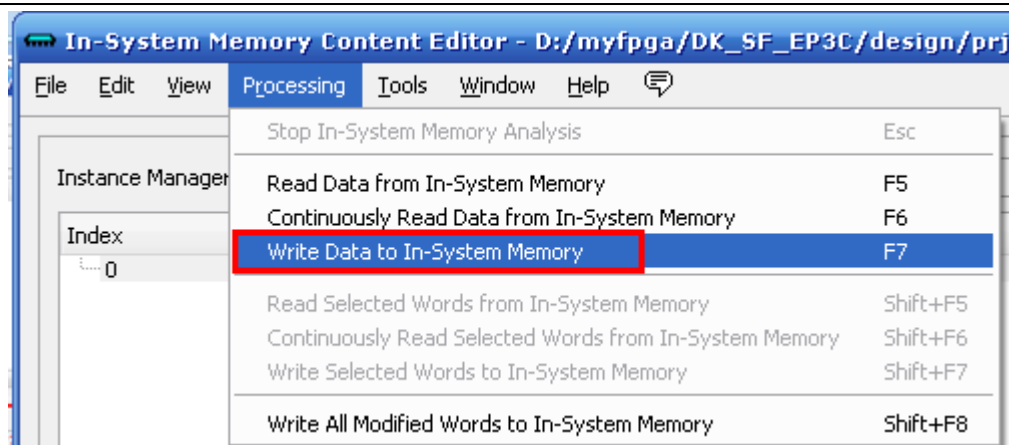
此时我们看到 RAM 的内容不再是?? 了, 都是实打实的数据, 大家不妨对照一下, 和我们之前编辑的 mif 文件中的数据是完全一致的。

Instance 0: NONE															
000000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000006	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000c	00	03	c0	00	00	07	e0	00	00	07	e0	00	00	06	e0
000012	00	0c	f0	00	00	0c	f0	00	00	0c	f0	00	00	18	78
000018	00	18	78	00	00	30	3c	00	00	30	3c	00	00	30	3c
00001e	00	60	1e	00	00	60	1e	00	00	60	1e	00	00	00	00
000024	00	ff	ff	00	01	ff	ff	00	01	80	0f	80	01	80	07
00002a	03	00	07	c0	03	00	03	c0	03	00	03	c0	07	00	03
000030	06	00	01	e0	0e	00	01	e0	1f	00	01	f8	7f	c0	0f
000036	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00003c	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

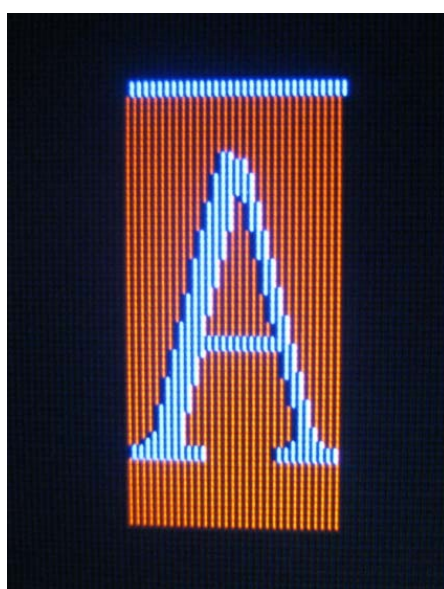
接下来我们可以更改数据, 直接点击所需要更改数据的位置, 然后输入更改的值即可。如图所示, 我们将 RAM 的地址 0 和地址 1 的 32bit 数据都改为了 FFFFFFFF, 那么意味着显示字符头两行的色彩都会变为字符所显示的蓝色。看看 LCD, 好像没变啊。

Instance 0: NONE															
000000	ff	ff	ff	ff	ff	ff	ff	ff	30	00	00	00	00	00	00
000006	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000c	00	03	c0	00	00	07	e0	00	00	07	e0	00	00	06	e0
000012	00	0c	f0	00	00	0c	f0	00	00	0c	f0	00	00	18	78
000018	00	18	78	00	00	30	3c	00	00	30	3c	00	00	30	3c
00001e	00	60	1e	00	00	60	1e	00	00	60	1e	00	00	00	00
000024	00	ff	ff	00	01	ff	ff	00	01	80	0f	80	01	80	07
00002a	03	00	07	c0	03	00	03	c0	03	00	03	c0	07	00	03
000030	06	00	01	e0	0e	00	01	e0	1f	00	01	f8	7f	c0	0f
000036	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00003c	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

哈哈, 还差一步操作, 回到菜单栏中, 点击 Processing→Write Data to In-System Memory。



如图所示, 此时字符有效区域原本为红色的头两行都被我们更改为了蓝色。



大家也可以用字模工具生成一个字母“B”, 然后实时的更改当前 RAM 中的字模数据, 可以预见的是, “A” 字符将会实时的被更改为 “B”。

7.6 基于 Qsys 的 NIOS II 实例 7——Qsys 的 LCD 组件设计

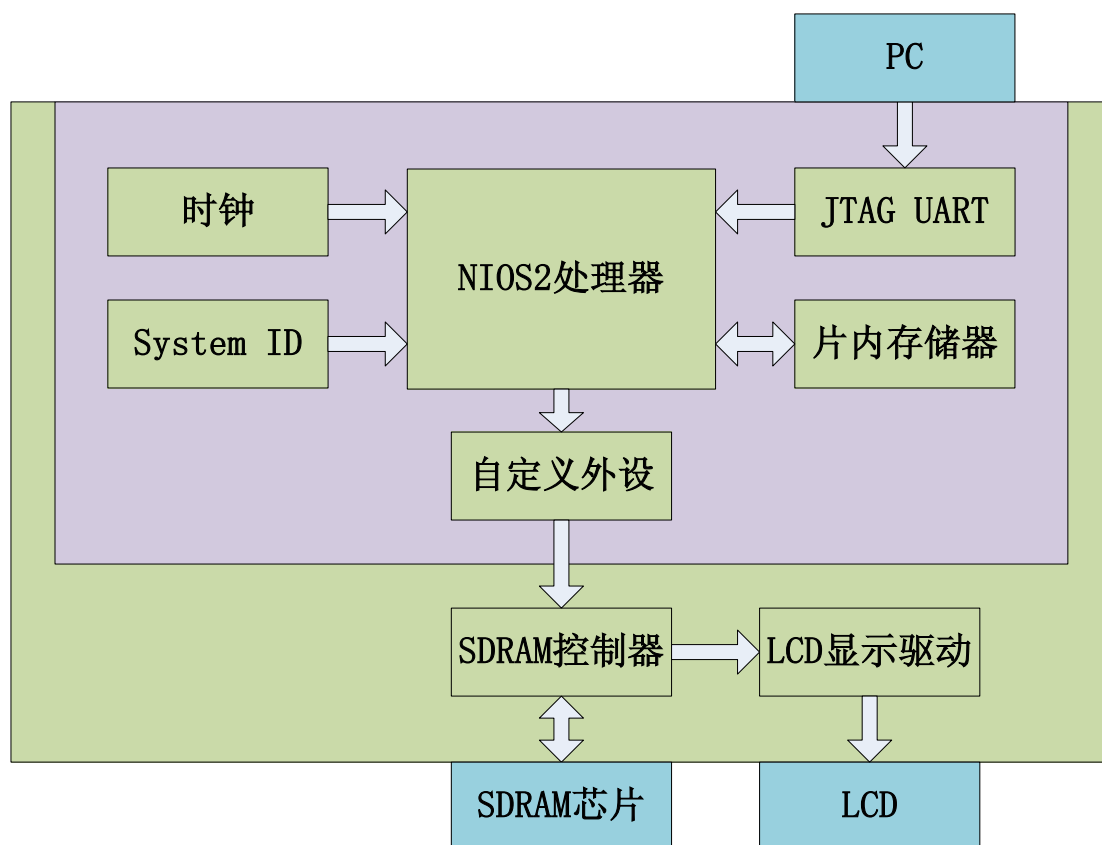
7.6.1 系统原理概述

上一个实例中, 我们已经学会了 LCD 驱动的编写, 并且能够通过片内 RAM 存储的字模显示字符; 在更早的实例中, 我们也学会了 Qsys 自定义组件的添加和使用, 如数码管组件、AD/DA 组件。没错, 在这个实例中, 我们要结合前面的一些实践案例, 创建一个可以使用软件编程驱动 LCD 字符显示的 LCD 组件。当然了, 由于 LCD 的显示驱动实时性很高, 通常需



要 60Hz 的刷新率, 数据吞吐量至少是 $320 \times 240 \times 16\text{bit} \times 60\text{Hz} = 70.3125\text{Mbps}$, 如果让 CPU 来干这个活, 恐怕别的事情都不干还不一定能忙活过来, 毕竟 CPU 的长处不在此。所以, 我们需要一片 SDRAM, 用于缓存实时显示的数据, FPGA 的 LCD 驱动模块专门来搬数据用于驱动 LCD 显示。

本工程的硬件架构如图所示。在该系统中, 从各个组件 (或者称为模块) 所依附的硬件环境看, 其实可以把它们看做三个等级。最高层次的是 PCB 板, 该板上集成了除 PC 机和 LCD 外的其它芯片, 包括 FPGA 芯片、SDRAM 芯片等。下一个层次是基于 FPGA 器件的, FPGA 内既有 Qsys 中集成的常用组件, 也有用户自己用逻辑资源搭建的控制或处理模块。再低一个层次, 逻辑搭建的模块就包括 SDRAM 控制器和 LCD 显示驱动模块。严格来说, 自定义外设也算是用户搭建起来的一个接口模块, 用于衔接 Qsys 和用户逻辑。Qsys 下也有多个组件构成, 包括了 NIOS2 处理器、片内存储器、JTAG UART 外设、时钟、系统 ID 等。

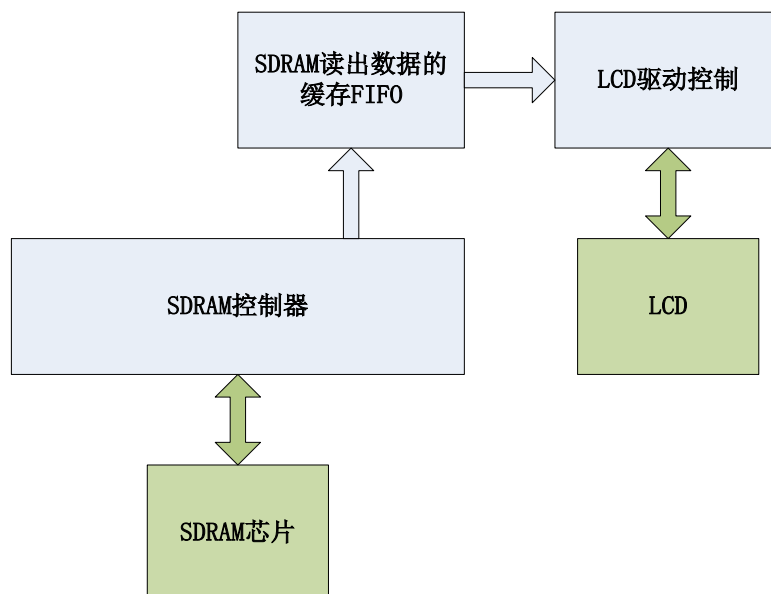


7.6.2 LCD 驱动移植

移植上一个实例中的 LCD 驱动代码, 这里我们对显示部分的逻辑做了修改。如图所示,



最终要送到 LCD 显示的数据是从一个 FIFO 中传送出来的。这个 FIFO 会根据 LCD 驱动模块的控制信号来不断的读取 SDRAM 中相对应的数据。



LCD 驱动模块用于和 FIFO 接口的信号有三个, `rdfifo_rddb` 即从 FIFO 读出的有效显示数据总线; `rdfifo_rdreq` 是 LCD 有效显示区需要更新每个像素点数据时发出一个请求, 它在对应送每个 LCD 像素有效数据前就会拉高一个时钟周期, 并且随后就能够得到 `rdfifo_rddb` 上的当前像素点色彩数据, 这个色彩数据会传送给 LCD 的 `r\g\b` 三组色彩接口, 由 `lcd_clk` 的上升沿进行锁存; `rdfifo_clr` 是控制 FIFO 清零的复位信号, 它指示 FIFO 每一个显示图像帧后进行复位, 以准备下一帧的图像。

```
module lcd_driver(
    clk, rst_n,
    lcd_en, lcd_clk, lcd_hsy, lcd_vsy, lcd_db_r, lcd_db_g, lcd_db_b,
    rdfifo_rddb, rdfifo_rdreq, rdfifo_clr
);
input clk; //25MHz
input rst_n; //低电平复位
// FPGA 与 LCD 接口信号
output lcd_en; //背光使能信号, 高有效
output lcd_clk; //时钟信号
output lcd_hsy; //行同步信号
output lcd_vsy; //场同步信号
output[4:0] lcd_db_r;
output[5:0] lcd_db_g;
output[4:0] lcd_db_b;
```



```
//LCD 与 FIFO 的接口
input[15:0] rdfifo_rddb;          //FIFO 读出数据总线
output rdfifo_rdreq;             //FIFO 读请求信号
output rdfifo_clr;               //FIFO 复位信号, 高电平有效

//-----
assign lcd_en = 1'b1;

//-----
//lcd_clk 时钟周期为 160ns (6.25MHz), 即 4 个 25MHz 的时钟周期
reg[1:0] sft_cnt;

always @(posedge clk or negedge rst_n)
    if(!rst_n) sft_cnt <= 2'd0;
    else sft_cnt <= sft_cnt+1'b1;

assign lcd_clk = sft_cnt[1];      //0-1:low, 2-3:high

wire dchange = (sft_cnt == 2'd2); //数据变化标志位, 高有效

//-----
//坐标计数
//x = 0-407; y = 0-261
reg[8:0] x_cnt; //x 计数器
reg[8:0] y_cnt; //y 计数器

always @(posedge clk or negedge rst_n)
    if(!rst_n) x_cnt <= 9'd0;
    else if(dchange) begin
        if(x_cnt == 9'd407) x_cnt <= 9'd0;
        else x_cnt <= x_cnt+1'b1;
    end

always @(posedge clk or negedge rst_n)
    if(!rst_n) y_cnt <= 9'd0;
    else if(dchange && (x_cnt == 9'd407)) begin
        if(y_cnt == 9'd261) y_cnt <= 9'd0;
        else y_cnt <= y_cnt+1'b1;
    end
```




```
end

//-----
//有效显示标志位产生
reg valid_yr;    //行显示有效信号

    //行显示有效信号
always @ (posedge clk or negedge rst_n)
    if(!rst_n) valid_yr <= 1'b0;
    else if(y_cnt == 9'd18) valid_yr <= 1'b1;
    else if(y_cnt == 9'd258) valid_yr <= 1'b0;

reg validr,valid;

always @ (posedge clk or negedge rst_n)
    if(!rst_n) validr <= 1'b0;
    else if((x_cnt == 9'd67) && valid_yr) validr <= 1'b1;
    else if((x_cnt == 9'd387) && valid_yr) validr <= 1'b0;

always @ (posedge clk or negedge rst_n)
    if(!rst_n) valid <= 1'b0;
    else valid <= validr;

//-----
// LCD 场同步,行同步信号
reg lcd_hsy_r,lcd_vsy_r;    //同步信号

always @ (posedge clk or negedge rst_n)
    if(!rst_n) lcd_hsy_r <= 1'b1;
    else if(x_cnt == 9'd0) lcd_hsy_r <= 1'b0;    //产生 lcd_hsy 信号
    else if(x_cnt == 9'd30) lcd_hsy_r <= 1'b1;

always @ (posedge clk or negedge rst_n)
    if(!rst_n) lcd_vsy_r <= 1'b1;
    else if(y_cnt == 9'd0) lcd_vsy_r <= 1'b0;    //产生 lcd_vsy 信号
    else if(y_cnt == 9'd3) lcd_vsy_r <= 1'b1;

assign lcd_hsy = lcd_hsy_r;
```



```
assign lcd_vsy = lcd_vsy_r;

//-----

//FIFO 读请求信号和复位信号产生
assign rdfifo_rdreq = validr & (sft_cnt == 2'd3);
assign rdfifo_clr = (y_cnt == 9'd0);

//-----

// LCD 色彩信号产生
reg[15:0] lcd_db_rgb; // LCD 色彩显示寄存器

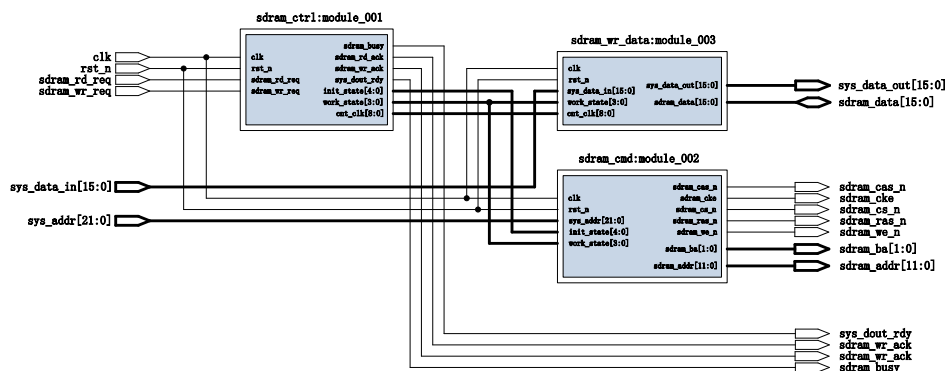
always @ (posedge clk or negedge rst_n)
    if(!rst_n) lcd_db_rgb <= 16'd0;
    else lcd_db_rgb <= rdfifo_rddb;

//r, g, b 控制液晶屏颜色显示
assign lcd_db_r = valid ? lcd_db_rgb[15:11]:5'd0;
assign lcd_db_g = valid ? lcd_db_rgb[10:5]:6'd0;
assign lcd_db_b = valid ? lcd_db_rgb[4:0]:5'd0;

endmodule
```

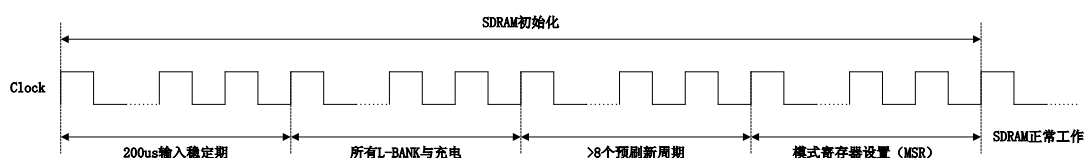
7.6.3 SDRAM 控制器设计

SDRAM 控制器一方面用于产生 SDRAM 芯片的读写控制时序, 另一方面则用于控制 SDRAM 读写数据与 FPGA 内其他模块的传输。首先, 设计者需要对这个控制器内部的逻辑功能做细分, 将其划分为多个子模块来实现。如图所示, `sdram_ctrl` 是 SDRAM 状态控制模块, 该模块主要完成 SDRAM 的上电初始化以及定时刷新、读写控制等状态的变迁, 内部设计了两个状态机, 一个用于上电初始化的状态控制, 另一个则用于正常工作时的状态控制; `sdram_cmd` 是 SDRAM 命令模块, 该模块根据 `sdram_ctrl` 模块的不同状态指示输出相应的 SDRAM 控制命令和地址(控制总线信号如 `sdram_cke`、`sdram_cs_n`、`sdram_we_n`、`sdram_ras_n`、`sdram_cas_n`, 地址总线信号 `sdram_addr`); `sdram_wr_data` 是 SDRAM 数据读写模块, 该模块同样是根据 `sdram_ctrl` 模块的状态指示完成 SDRAM 数据总线的控制, SDRAM 的数据读写都在该模块完成。

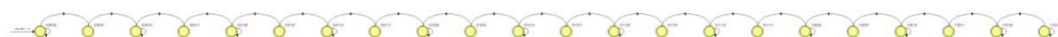


如图所示, SDRAM 的上电初始化步骤一般是(相关的基本概念请参考前文推荐的参考文章):

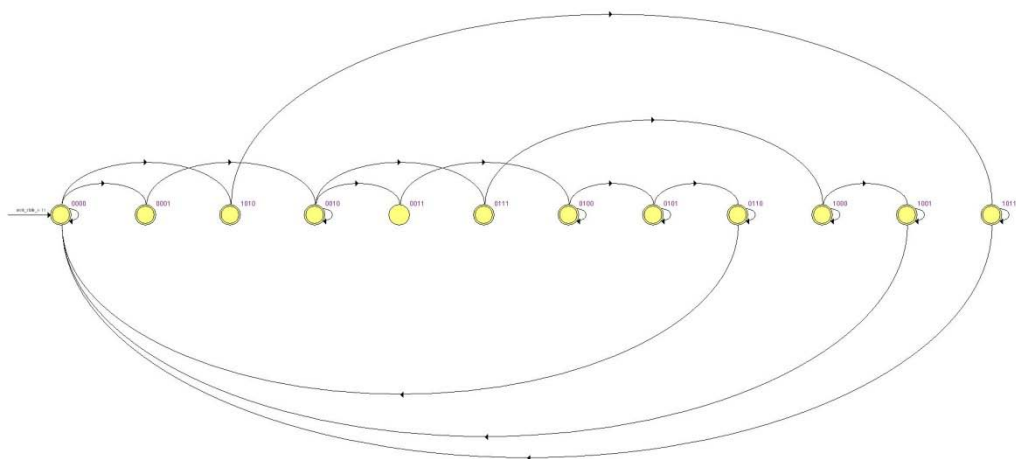
- ① 等待 200us, 这是 SDRAM 的输入稳定期;
- ② 所有 L-BANK 预充电;
- ③ 至少 8 个预刷新周期;
- ④ 模式寄存器设置 (MSR), 完成 SDRAM 读写相关的配置。



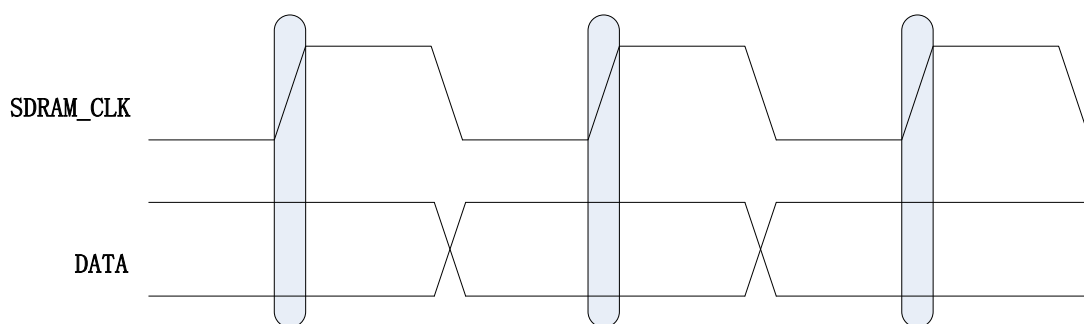
如图所示, SDRAM 的初始化状态机有 20 个状态, 最后到达 I_DONE 状态后停止, 说明初始化完成, 然后另一个用于指示 SDRAM 正常工作状态的状态机将被激活。



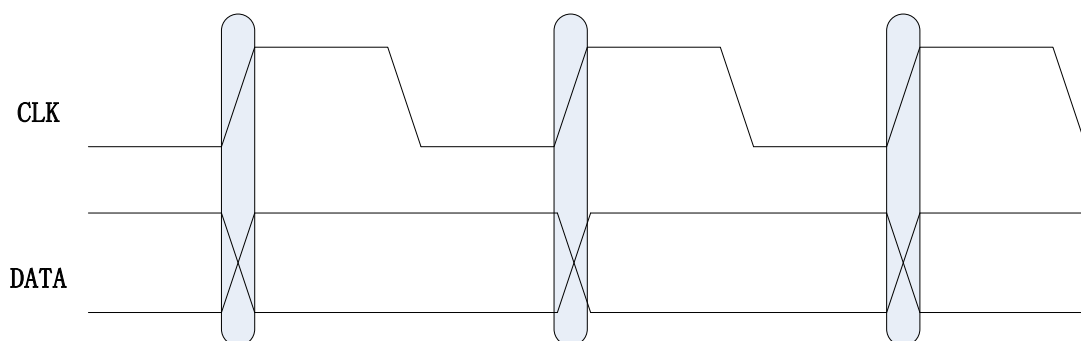
关于 SDRAM 正常工作的读写时序请读者参考相关 datasheet《K4S641632C.pdf》, 这里不花太多篇幅讨论。SDRAM 正常工作状态下的状态迁移如图所示。不操作 SDRAM 时处于 W_IDLE 状态, 如果有读请求、写请求或者自刷新请求信号产生, 则进入的相应的状态, 在这些不同的响应状态中, 设计者需要协调好 SDRAM 的控制总线、地址总线、数据总线, 从而保证稳定可靠的读写 SDRAM 的数据。



再说 SDRAM 的时钟信号产生,这个时钟信号主要是输出给 SDRAM 使用,用于同步 FPGA 传输给 SDRAM 的信号。如图所示,这个时钟和信号间必须是中央对齐的,以保证传输信号在时钟的上升沿正确的被 SDRAM 接收。



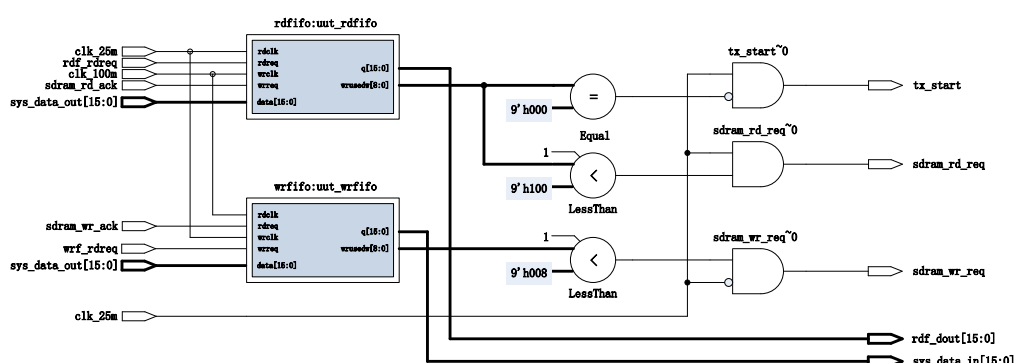
而系统时钟（FPGA 内部的工作时钟）和输出信号的关系却如图所示。



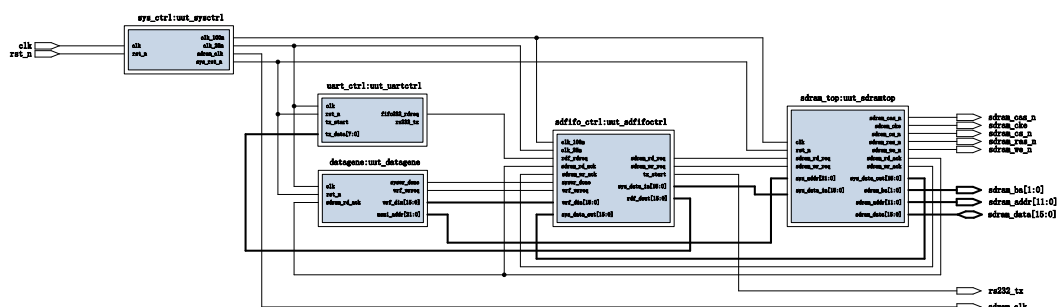
那么如何能够保证 SDRAM 的驱动时钟和信号的关系如图所示呢? 使用 PLL 来调整时钟频率和相位是不错的选择,因此特权同学根据调试和时序分析的结果,对传输到 SDRAM 的时钟信号添加了+3ns 的偏移,从而保证了 SDRAM 在锁存数据时有可靠的建立时间和保持时间裕量。这个相位偏移是必需的,但不是固定的,设计者需要具体问题具体分析。



说完这个 SDRAM 的底层控制方式, 我们还要来看其它模块如何利用它达到有效的数据读写。这里借助了两个存储器(异步 FIFO)来达到这个目的, 如图所示, wrfifo 用于写 SDRAM 数据, rdfifo 用于读 SDRAM 数据。由于该工程中 SDRAM 读写都是以 8 个字(16bit)为单位, 所以使用了 FIFO 当前数据量作为操作 SDRAM 的状态指示。当 wrfifo 数量超过一定个数则发出写 SDRAM 请求, 在写选通期间, 适当的时候就将相应的读出 wrfifo 中的 8 个数据。同样, 在 rdfifo 数据少于固定数量(rdfifo 接近空)时发出读 SDRAM 请求, 适当的时候将读出 8 个新的数据写入 rdfifo 中, 以保证后续电路总是持续的传输从 rdfifo 读出的数据。



从总体上来看这个工程, 工程内部分为 PLL 以及复位处理模块、SDRAM 控制器模块、读写 SDRAM 数据缓存模块、模拟写 SDRAM 模块和串口发送模块, RTL 视图如图所示。



首先由写 SDRAM 逻辑模块在上电延时后从 SDRAM 的 0 地址开始写入递增数据, 随后通过内部 FIFO 依次送入 SDRAM。SDRAM 的所有地址写完数据后, 启动 SDRAM 读逻辑, 从 0 地址开始读出 SDRAM 内的数据放入缓存 FIFO 中, 然后串口模块把该 FIFO 中的数据依次上传到 PC 机(串口线接到 PC 机, 使用串口调试助手观察即可)。整个过程主要就是测试 SDRAM 读写, 内部逻辑大都使用 25MHz 的时钟, SDRAM 读写使用了 100MHz。

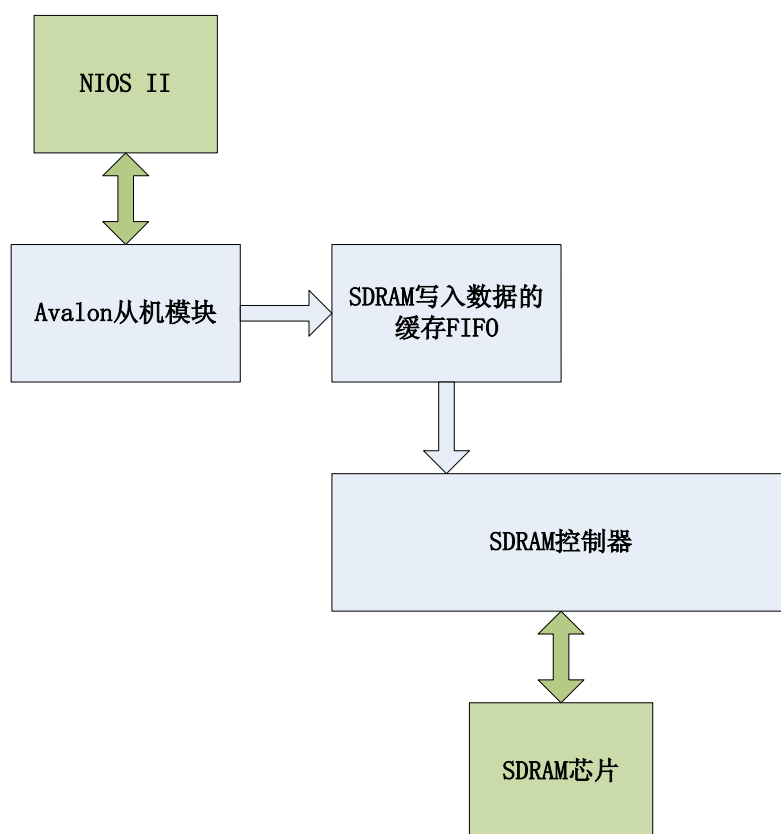
SDRAM 控制器一个有 4 个设计模块, 还有 1 个用于存储参数的模块。4 个设计模块是 sdr_top.v、sdr_ctrl.v、sdr_wr_data.v、sdr_cmd.v, 其中 sdr_top.v 是顶层模块; 而 sdr_para.v 是用于存储参数的一个专门模块。限于篇幅, 由于代码量较大, 这里不一一贴



出来了, 请大家参考实例工程。

7.6.4 Avalon-MM 从机接口设计

在另一侧, 有一个 SDRAM 写入数据的缓存 FIFO, 这个 FIFO 写入的数据源头是 NIOS II 的软件程序。NIOS II 通过 Avalon-MM 总线来执行写操作, 因此我们逻辑上必须设计一个模块用于衔接 Avalon-MM 总线和 FIFO, 那么就专门设计了一个 Avalon-MM 从机适配模块, 它将 NIOS II 送过来的数据和地址信息写入到 FIFO 中, FIFO 中的数据都会被送到 SDRAM 指定地址中, 这些固定地址的数据都是最终会在 LCD 中显示出来的像素点色彩。



Avalon-MM 从机适配模块代码如下, 它就是后面要挂到 Qsys 上的 LCD 自定义组件。它的接口中除了 Avalon-MM 从机接口, 还有 3 个和 FIFO 连接的写信号。wrf_wrreq 是 FIFO 的写请求, 当它为高电平时, 对应的数据总线 wrf_din 和地址总线 wrf_ain 被送入 FIFO 中缓存, 在正确到 SDRAM 控制权后 wrf_ain 地址上将写入数据 wrf_din。

```
module as_lcdwr(
    clk, rst_n,
    avalone_cs_n, avalone_wr_n, avalone_wrdata, avalone_wraddr,
```



```
wrf_din, wrf_ain, wrf_wrreq
);

input clk; //PLL 输出 50MHz 时钟
input rst_n; //系统复位信号, 低有效

input avalone_cs_n; //总线读片选, 低电平有效
input avalone_wr_n; //总线写使能信号, 低电平有效
input[15:0] avalone_wrddata; //总线写入数据
input[21:0] avalone_wraddr; //总线写入地址

//wrFIFO 输入控制接口
output reg [15:0] wrf_din; //sdram 数据写入缓存 FIFO 输入数据总线
output reg [21:0] wrf_ain; //sdram 数据写入缓存 FIFO 输入地址总线
output wrf_wrreq; //sdram 数据写入缓存 FIFO 数据输入请求, 高有效

//-----
//avalon 从接口逻辑
reg wrcs_nr; //avalone 片选和写选通有效, 锁存四拍
reg[21:0] avawradr; //avalone 总线地址锁存
reg[2:0] pos_wrcsnr; //pos_wrcsn 锁存一拍

wire wrcs_n = avalone_cs_n | avalone_wr_n; //avalone 片选和写选通有效

always @(posedge clk or negedge rst_n)
    if(!rst_n) wrcs_nr <= 1'b1;
    else wrcs_nr <= wrcs_n;

wire pos_wrcsn = wrcs_nr & ~wrcs_n; //wrcs_n 上升沿捕获

//pos_wrcsn 锁存 3 拍
always @(posedge clk or negedge rst_n)
    if(!rst_n) pos_wrcsnr <= 3'd0;
    else pos_wrcsnr <= {pos_wrcsnr[1:0], pos_wrcsn};

//数据总线锁存
always @(posedge clk or negedge rst_n)
```



```
        if(!rst_n) wrf_din <= 16'd0;
        else if(pos_wrdsn) wrf_din <= avalone_wrdata;

        //总线地址锁存
always @(posedge clk or negedge rst_n)
    if(!rst_n) avawradr <= 22'd0;
    else if(pos_wrdsn) avawradr <= avalone_wraddr;

wire[8:0] wrf_ainr = avawradr[8:0]-8'd160;

        //总线地址译码
always @(posedge clk or negedge rst_n)
    if(!rst_n) wrf_ain <= 22'd0;
    else if(pos_wrdsn[1]) begin
        wrf_ain[21:9] <= avawradr[21:9];
        if(avawradr[8:0] < 9'd160) wrf_ain[8:0] <= avawradr[8:0]; //page0
        else begin
            wrf_ain[8] <= 1'b1;
            wrf_ain[7:0] <= wrf_ainr[7:0]; //page1
        end
    end
end

assign wrf_wrreq = pos_wrdsn[2];

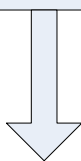
endmodule
```

7.6.5 数据缓存模块和 FIFO 配置

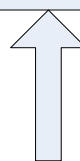
如图所示,这里有一个专门用于缓存 SDRAM 读写数据的模块。该模块配置了两个 FIFO。其中,用于缓存 SDRAM 写入数据的 FIFO 命名为 wrfifo,而用于缓存 SDRAM 读出数据的 FIFO 命名为 rdfifo。



SDRAM写入数据的
缓存FIFO

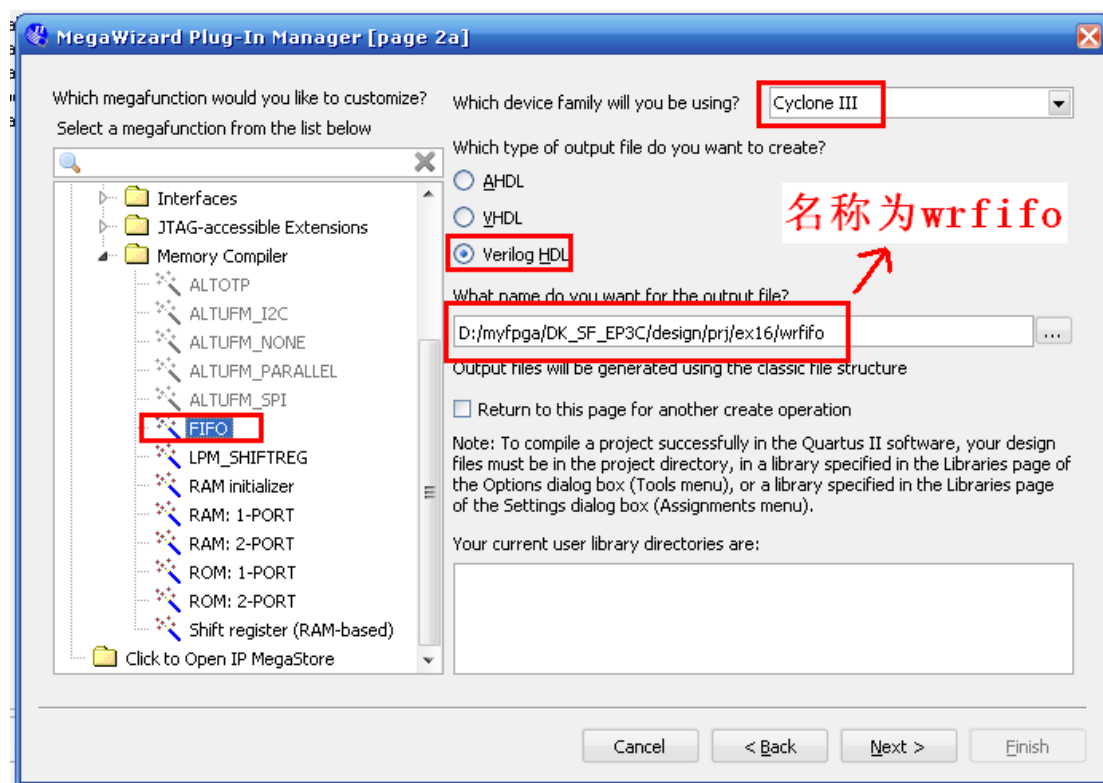


SDRAM读出数据的
缓存FIFO



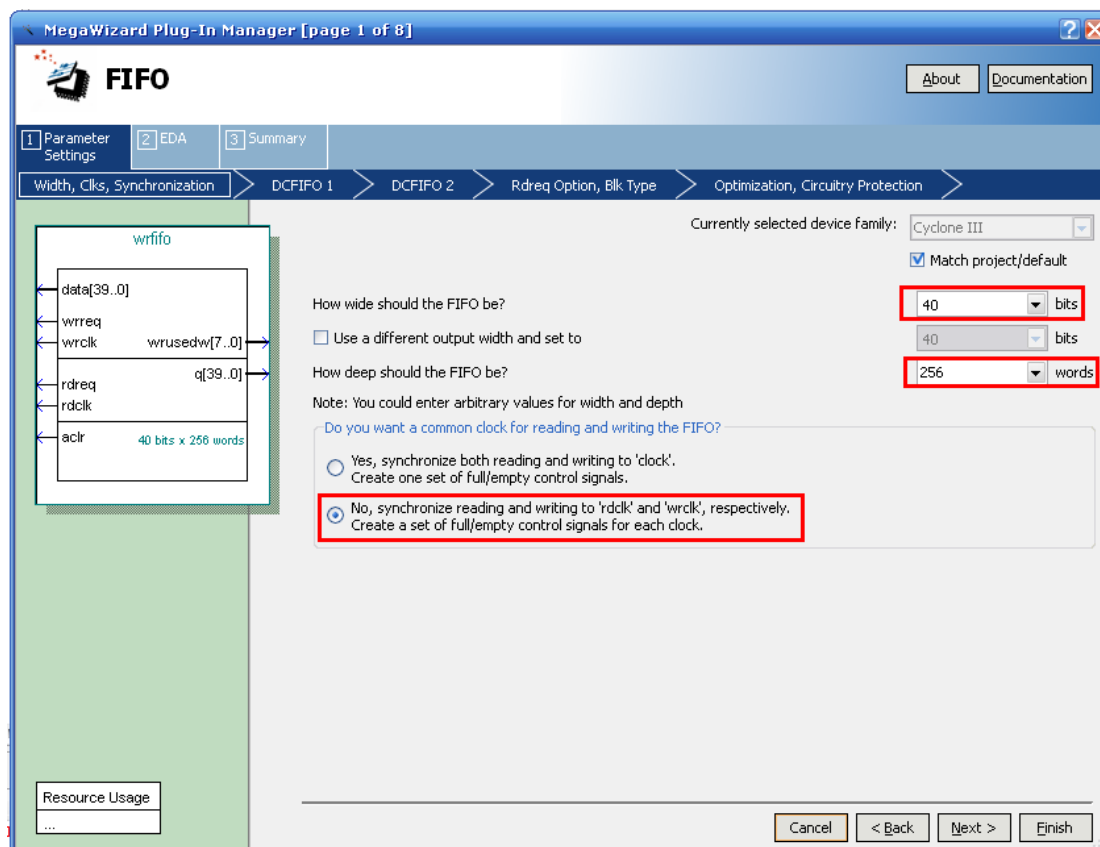
Wrfifo 的创建和配置如下。

① 点击菜单栏的 Tools→MegaWizard Plug-In Manager, new 一个 megafunction。在 page 2a 的页面中做如图设置, 注意这个新 FIFO 的名称输入为 wrfifo, 路径为当前工程所在路径。

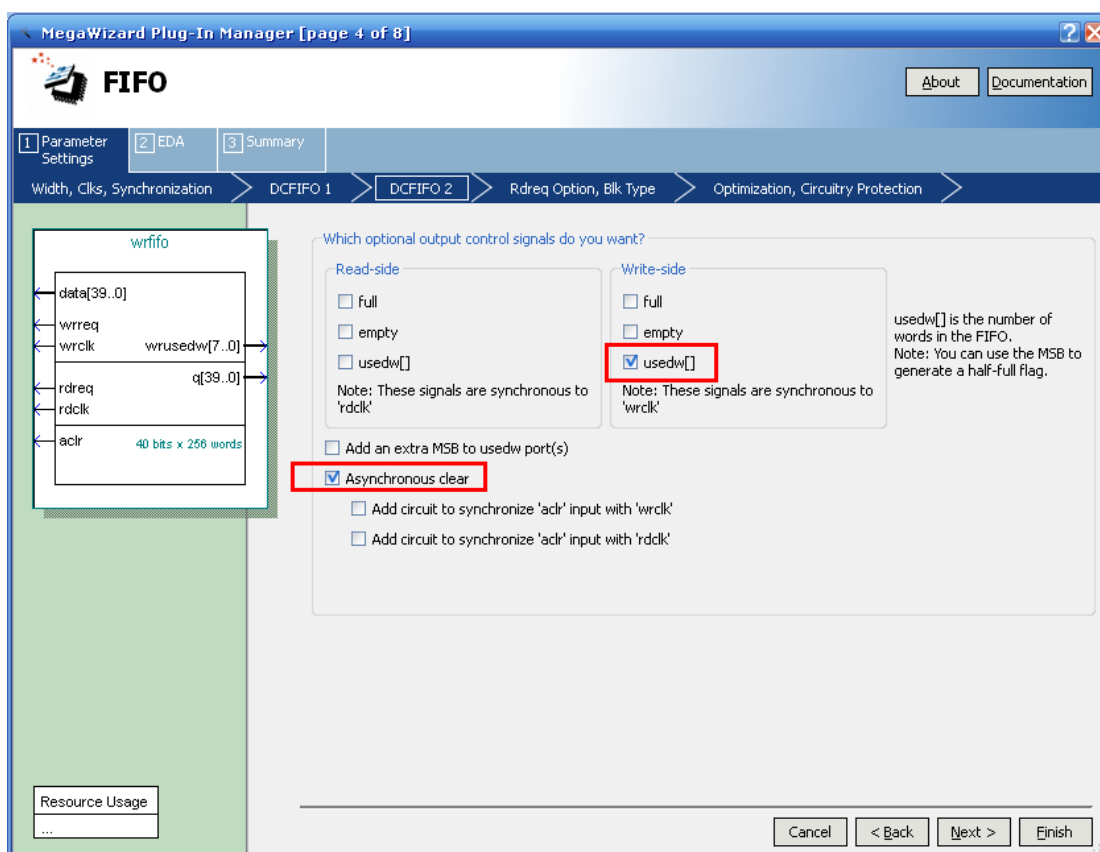


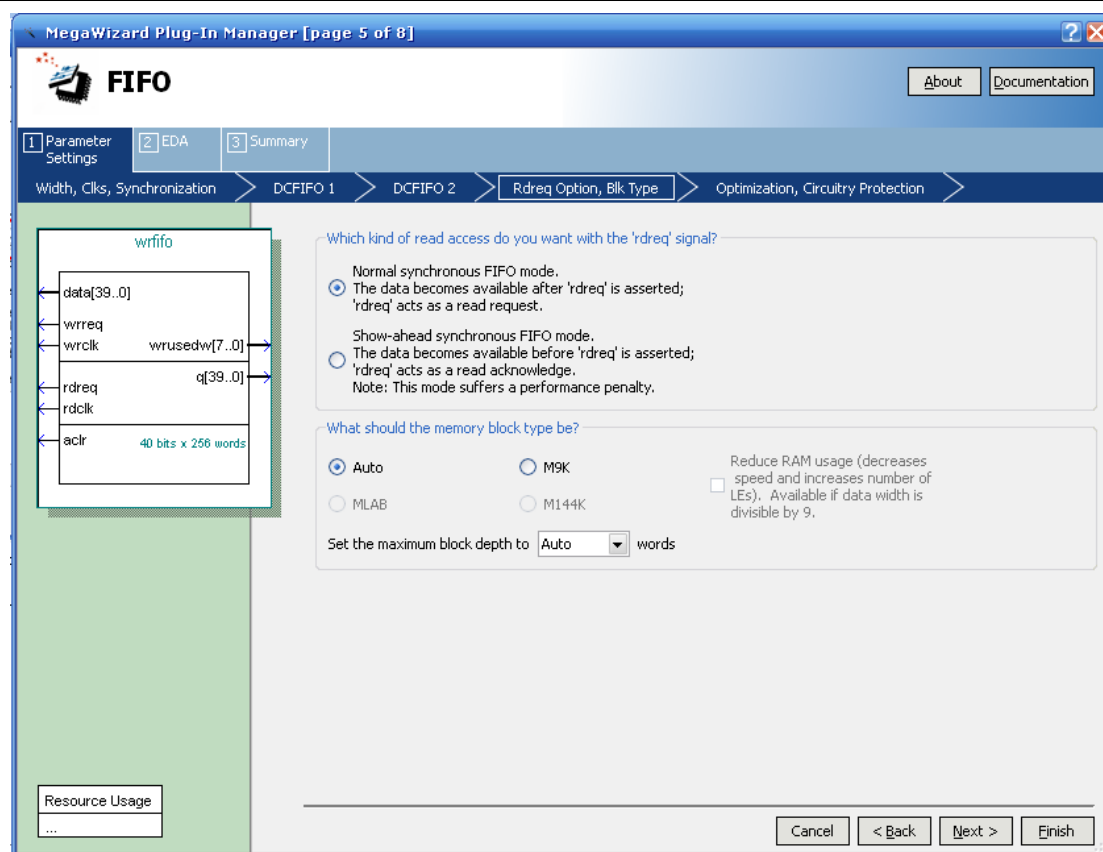
② Parameter Setting 各个页面的配置如图所示, 没有特别说明的页面采用默认设置即可。

设置当前 FIFO 的位宽为 40bits, 深度为 256words。

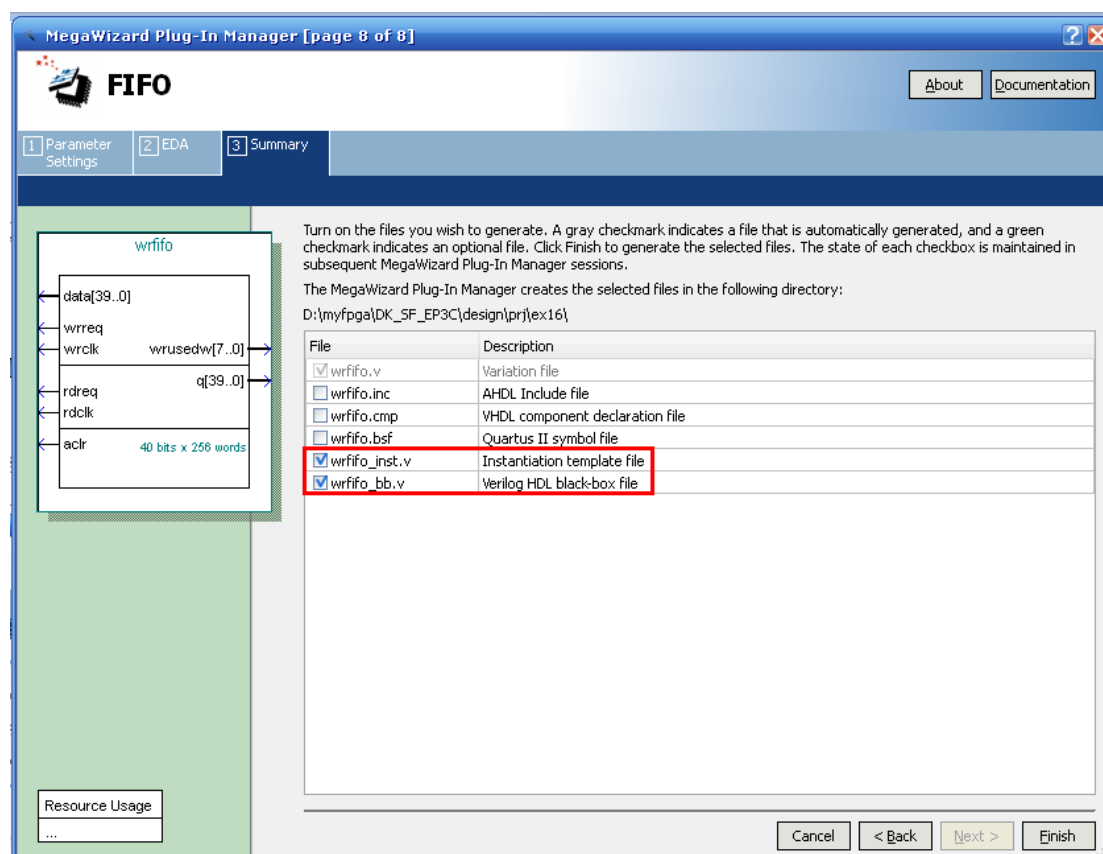


FIFO 的 Write-side 开启写入字节数指示接口 `usedw[]`。开启异步清除功能。





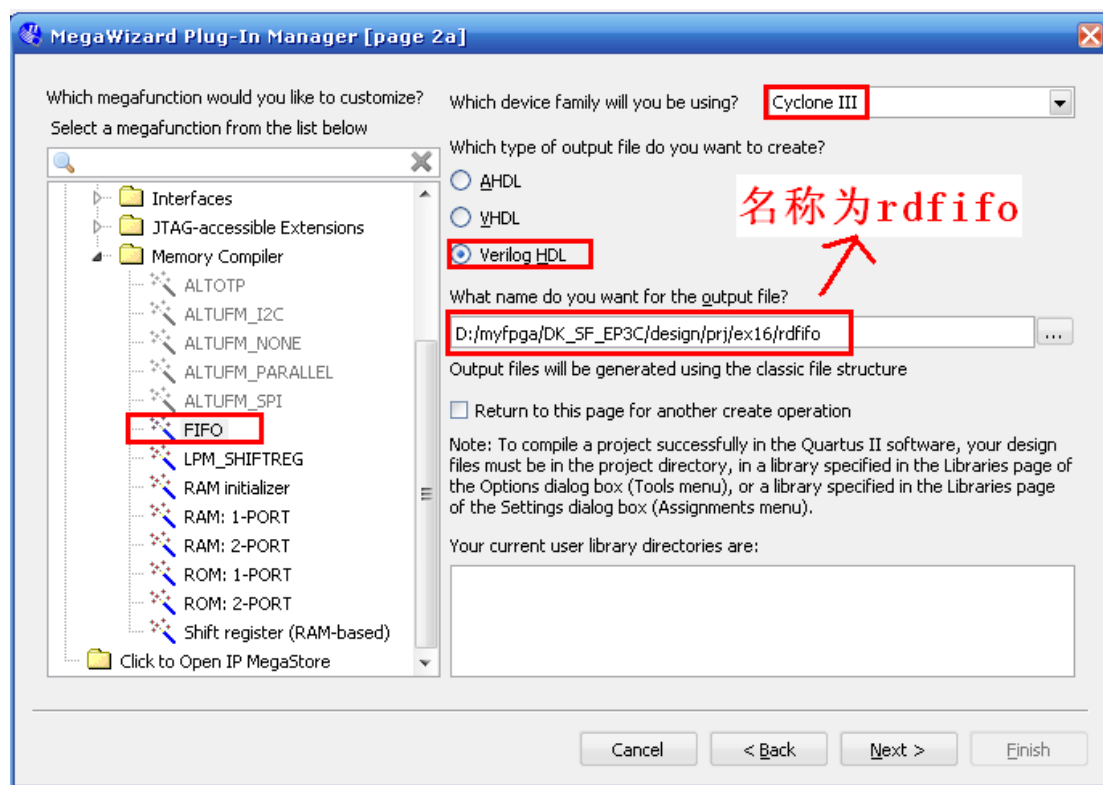
③ Summary 页面中, 勾选 wrfifo_inst.v 和 wrfifo_bb.v 文件。完成 wrfifo 的配置。



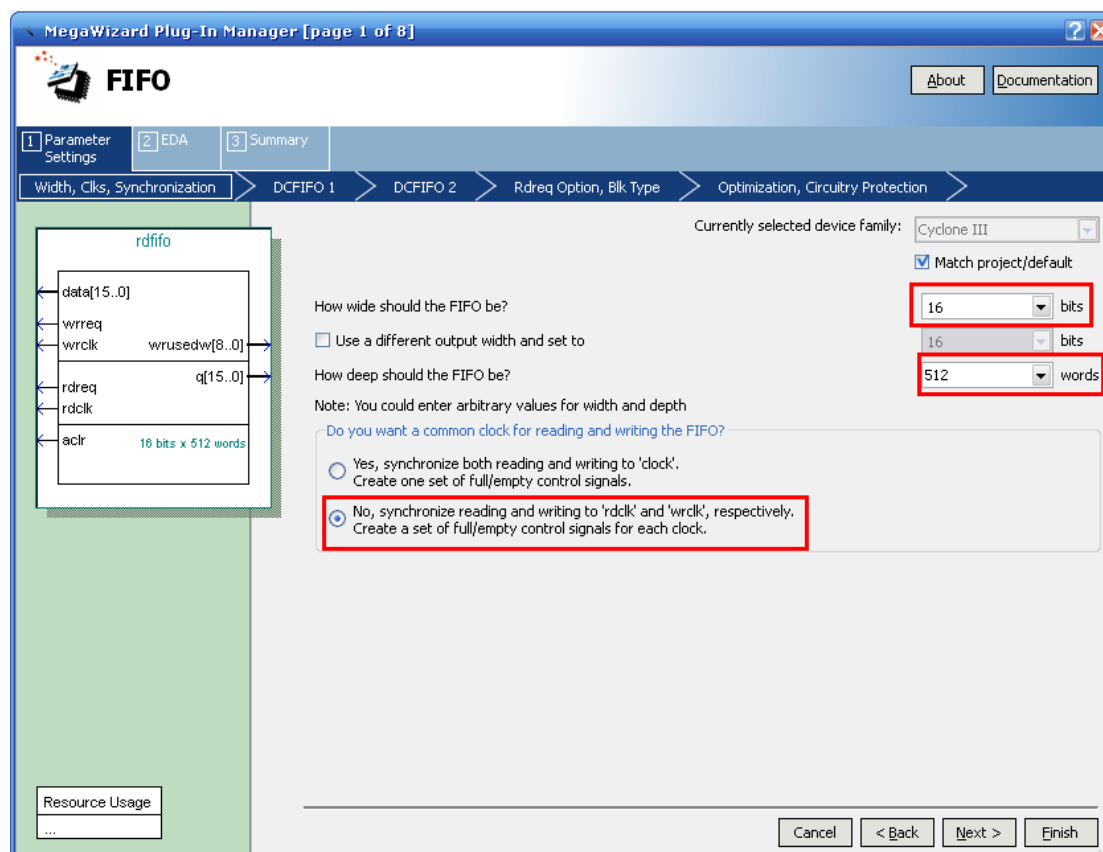


和 wrfifo 的配置类似, rdfifo 的创建和配置如下。

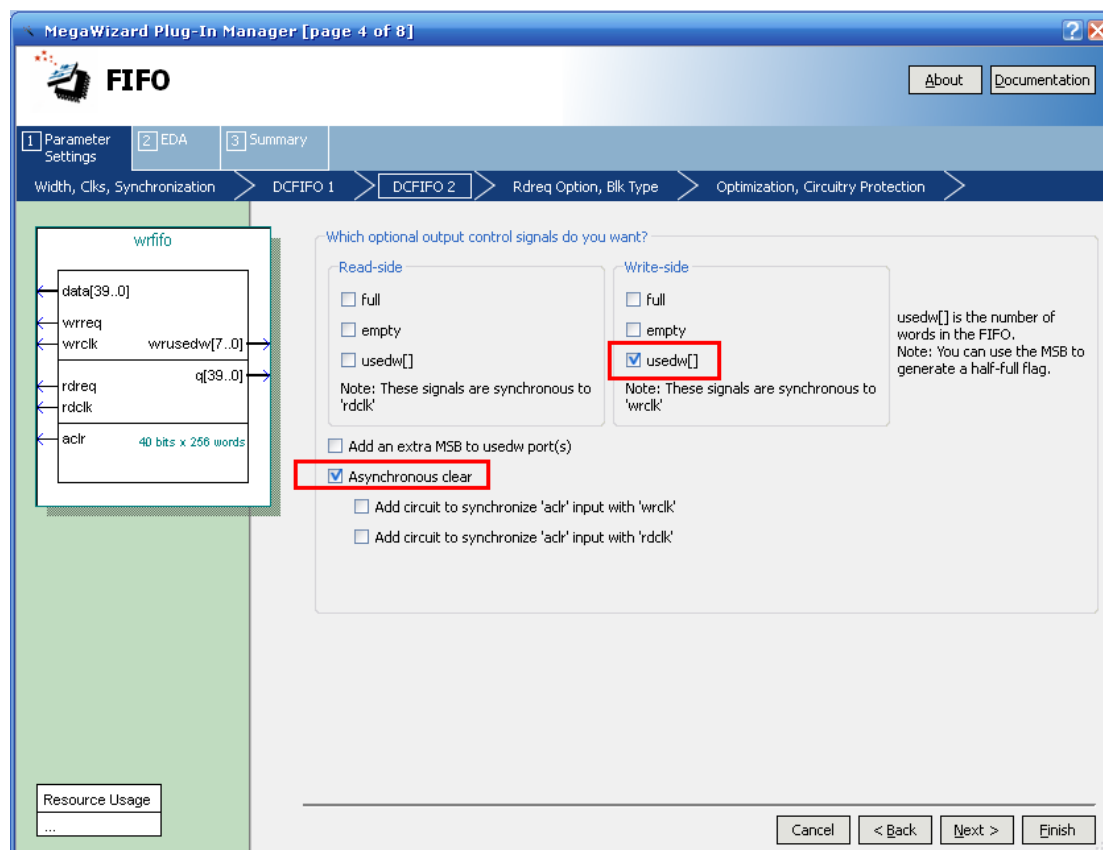
① 点击菜单栏的 Tools→MegaWizard Plug-In Manager, new 一个 megafunction。在 page 2a 的页面中做如图设置, 注意这个新 FIFO 的名称输入为 rdfifo, 路径为当前工程所在路径。

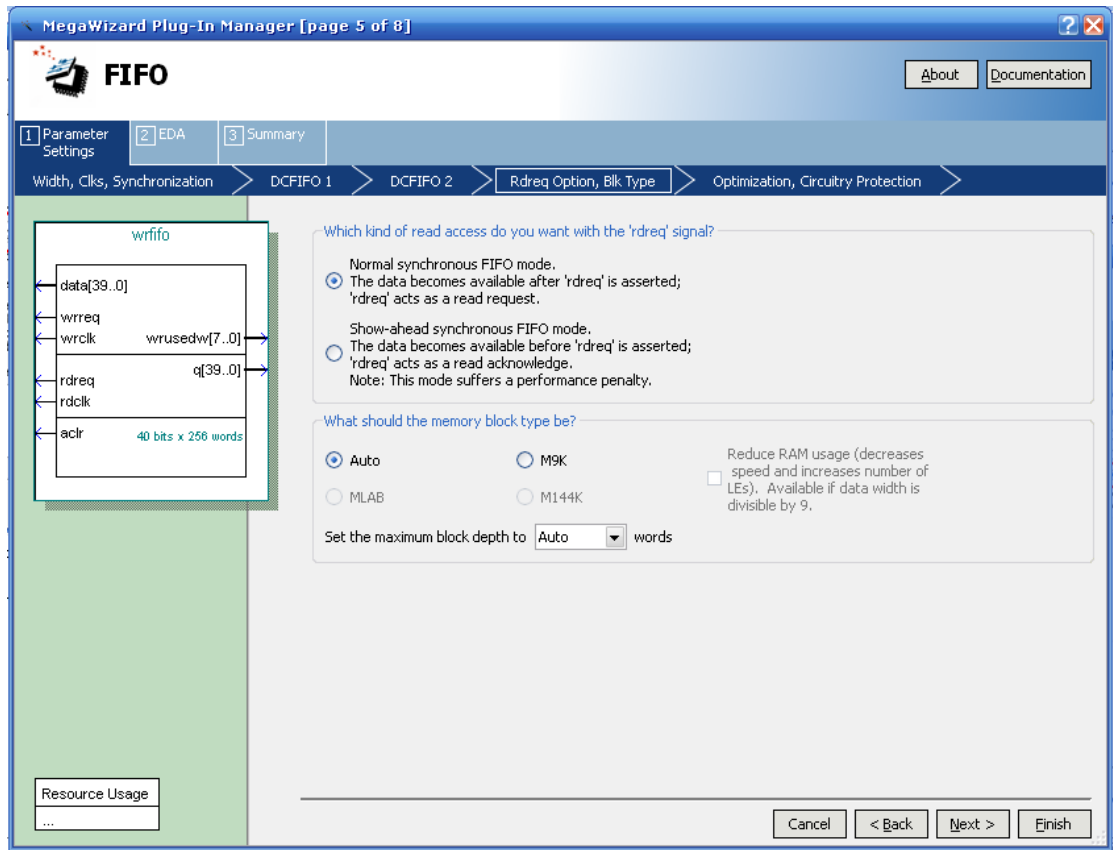


② Parameter Setting 各个页面的配置如图所示, 没有特别说明的页面采用默认设置即可。
设置当前 FIFO 的位宽为 16bits, 深度为 512words。

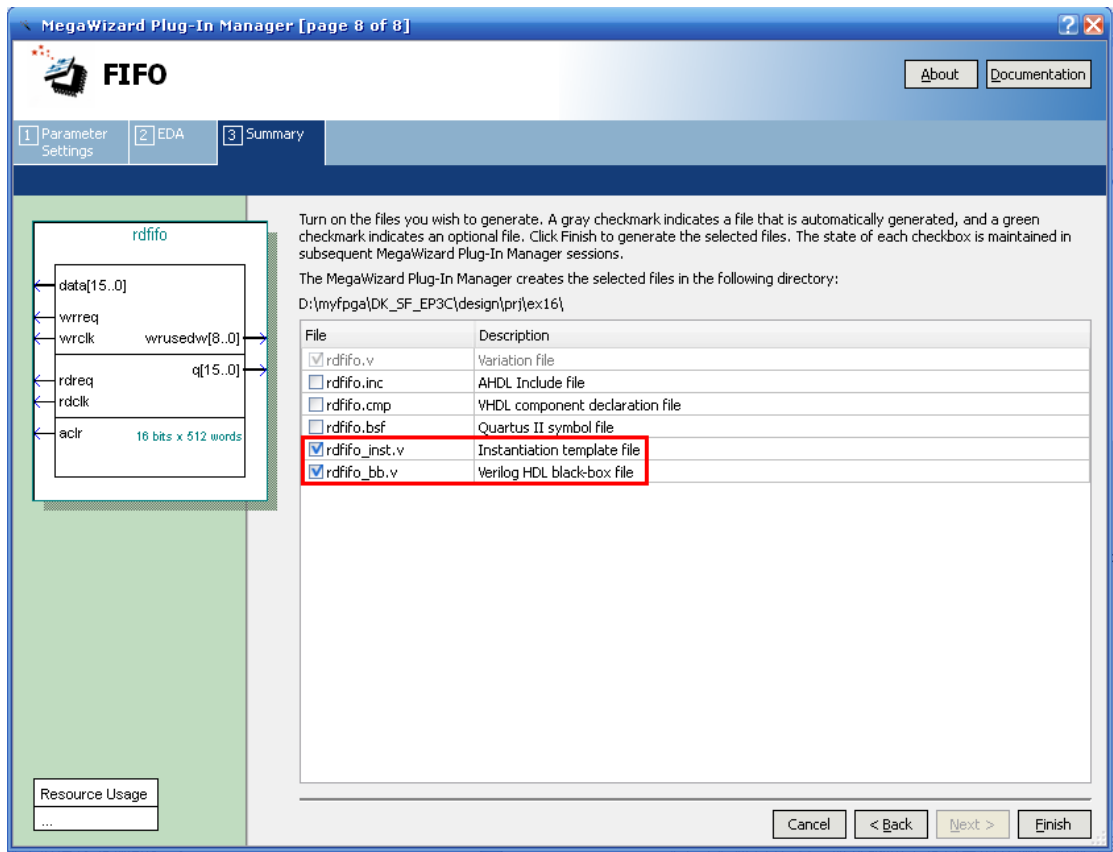


FIFO 的 Write-side 开启写入字节数指示接口 `usedw[]`。开启异步清除功能。





③ Summary 页面中，勾选 wrfifo_inst.v 和 wrfifo_bb.v 文件。完成 wrfifo 的配置。





最后我们看这个模块的代码，还是好好消化了它才是王道。

```
module sdfifo_ctrl(
    clk_25m, clk_100m, rst_n,
    wrf_din, wrf_wrreq, wrf_ain,
    sdram_wr_ack, sys_wraddr, sys_rdaddr, sys_data_in, sdram_wr_req,
    sys_data_out, sdram_rd_ack, sdram_rd_req,
    rdfifo_rdreq, rdfifo_clr, rdfifo_rddb
);

input clk_25m; //PLL 输出 25MHz 时钟
input clk_100m; //PLL 输出 100MHz 时钟
input rst_n; //系统复位信号，低有效

//wrFIFO 控制
input wrf_wrreq; //sdram 数据写入缓存 FIFO 数据输入请求，高有效
input[21:0] wrf_ain; //sdram 数据写入缓存 FIFO 输入地址总线
input[15:0] wrf_din; //sdram 数据写入缓存 FIFO 输入数据总线
input sdram_wr_ack; //系统写 SDRAM 响应信号, 作为 wrFIFO 的输出有效信号
output sdram_wr_req; //系统写 SDRAM 请求信号
output[15:0] sys_data_in; //sdram 数据写入缓存 FIFO 输出数据总线，即写 SDRAM 时
数据暂存器
output[21:0] sys_wraddr; // 写 SDRAM 时地址暂存器，(bit21-20)L-Bank 地址: (bit19-8)为行地址，(bit7-0)为列地址

//rdFIFO 写入控制
input sdram_rd_ack; //系统读 SDRAM 响应信号, 作为 rdFIFO 的输写有效信号
input[15:0] sys_data_out; //sdram 数据读出缓存 FIFO 输入数据总线
output sdram_rd_req; //系统读 SDRAM 请求信号
output[21:0] sys_rdaddr; // 读 SDRAM 时地址暂存器，(bit21-20)L-Bank 地址: (bit19-8)为行地址，(bit7-0)为列地址

//rdFIFO 读出控制
input rdfifo_rdreq; //sdram 数据读出缓存 FIFO 数据输出请求，高有效
input rdfifo_clr; //高有效，用于使能 SDRAM 读数据单元进行寻址或地址清零
output[15:0] rdfifo_rddb; //VGA 显示数据

//-----
//sdram 读写响应完成标致捕获
reg sdwrackr1, sdwrackr2; //sdram_wr_ack 寄存器
reg sdrdackr1, sdrdackr2; //sdram_rd_ack 寄存器
```



```
//锁存两拍 sdram_wr_ack, 用于下降沿捕获
always @(posedge clk_100m or negedge rst_n)
    if(!rst_n) begin
        sdwrackr1 <= 1'b0;
        sdwrackr2 <= 1'b0;
    end
    else begin
        sdwrackr1 <= sdram_wr_ack;
        sdwrackr2 <= sdwrackr1;
    end

wire neg_sdwrack = ~sdwrackr1 & sdwrackr2; //sdram_wr_ack 下降沿标志位, 高有效
一个时钟周期

//锁存两拍 sdram_rd_ack, 用于下降沿捕获
always @(posedge clk_100m or negedge rst_n)
    if(!rst_n) begin
        sdrdackr1 <= 1'b0;
        sdrdackr2 <= 1'b0;
    end
    else begin
        sdrdackr1 <= sdram_rd_ack;
        sdrdackr2 <= sdrdackr1;
    end

wire neg_sdrdack = ~sdrdackr1 & sdrdackr2; //sdram_rd_ack 下降沿标志位, 高有效
一个时钟周期

//-----
reg rdfifo_clrr;          //将 50M 时钟域的 rdfifo_clr 打一拍以同步到 100M 的
rdfifo_clrr

always @(posedge clk_100m or negedge rst_n)
    if(!rst_n) rdfifo_clrr <= 1'b0;
    else rdfifo_clrr <= rdfifo_clr;

//-----
//读写 sdram 请求信号产生
```




```
wire[7:0] wrf_use;      //sdram 数据写入缓存 FIFO 已用存储空间数量
wire[8:0] rdf_use;      //sdram 数据读出缓存 FIFO 已用存储空间数量

reg[2:0] swstate;      //SDRAM 写数据状态机
parameter SW_IDLE = 3'd0;
parameter SW_RFIO = 3'd1;
parameter SW_RFI1 = 3'd2;
parameter SW_WSD0 = 3'd3;
parameter SW_WSD1 = 3'd4;

always @(posedge clk_100m or negedge rst_n)
    if(!rst_n) swstate <= SW_IDLE;
    else begin
        case(swstate)
            SW_IDLE: if(wrf_use >= 8'd1) swstate <= SW_RFIO;    //FIFO 非空, 进入读 FIFO 状态
                        else swstate <= SW_IDLE;
            SW_RFIO: swstate <= SW_RFI1;    //读 FIFO
            SW_RFI1: swstate <= SW_WSD0;    //FIFO 数据有效
            SW_WSD0: if(sdram_wr_ack) swstate <= SW_WSD1;    //SDRAM 写响应有效, 进入下一状态
                        else swstate <= SW_WSD0;
            SW_WSD1: swstate <= SW_IDLE;
            default: swstate <= SW_IDLE;
        endcase
    end

wire wrf_rdreq = (swstate == SW_RFIO); //sdram 数据写 FIFO 读请求信号
assign sdram_wr_req = (swstate == SW_WSD0); //FIFO(1 个 16bit 数据)即发出写 SDRAM 请求信号

assign sdram_rd_req = (rdf_use < 9'd350) & ~rdfifo_clrr;    //VGA 显示有效且 FIFO 不满 350 个数据即发出读 SDRAM 请求信号

//-----
//sdram 读地址产生逻辑
reg[13:0] sys_rdabr;    //SDRAM 读数据页
```



```
//sdram 读地址产生
always @(posedge clk_100m or negedge rst_n)
    if(!rst_n) sys_rdabr <= 14'd0;
    else if(rdfifo_clrr) sys_rdabr <= 14'd0;    //从起始地址读数据
    else if(neg_sdrdack) sys_rdabr <= sys_rdabr+1'b1;    //一次读出完成后地址递增

assign sys_rdaddr = {sys_rdabr, 8'd0};

//-----
//例化 SDRAM 写入数据缓存 FIFO 模块
wrfifo      uut_wrfifo(
    .aclr(!rst_n),
    .data({wrf_ain, wrf_din}),    //写入地址和数据缓存
    .rdclk(clk_100m),
    .rdreq(wrf_rdreq),
    .wrclk(clk_25m),
    .wrreq(wrf_wrreq),
    .q({sys_wraddr, sys_data_in}),    //写入地址和数据输出
    .wrusedw(wrf_use)
);

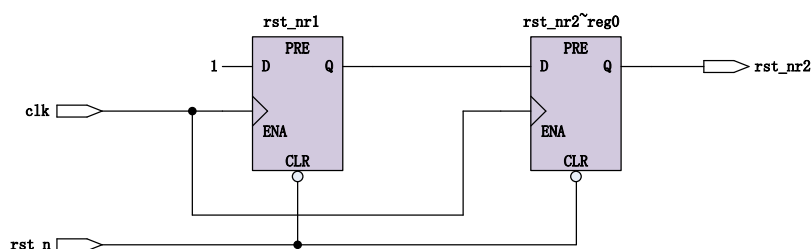
//-----
//例化 SDRAM 读出数据缓存 FIFO 模块
rdfifo      uut_rdfifo(
    .aclr(rdfifo_clrr),
    .data(sys_data_out),
    .rdclk(clk_25m),
    .rdreq(rdfifo_rdreq),
    .wrclk(clk_100m),
    .wrreq(sdram_rd_ack),
    .q(rdfifo_rddb),
    .wrusedw(rdf_use)
);

endmodule
```



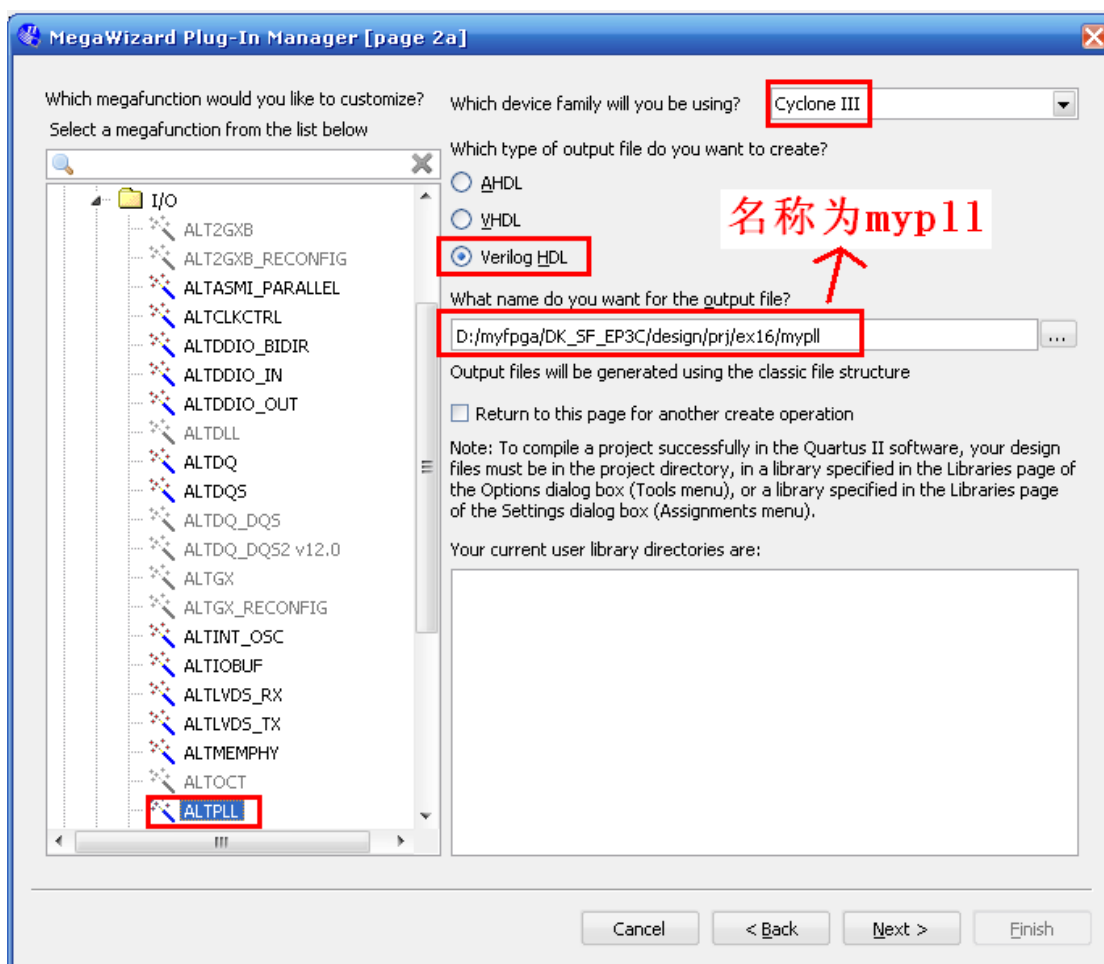
7.6.6 PLL 配置与复位设计

该模块例化了配置的 PLL，并且对系统的外部输入复位信号以及 PLL 生效后的复位做了“异步复位，同步释放”的处理。

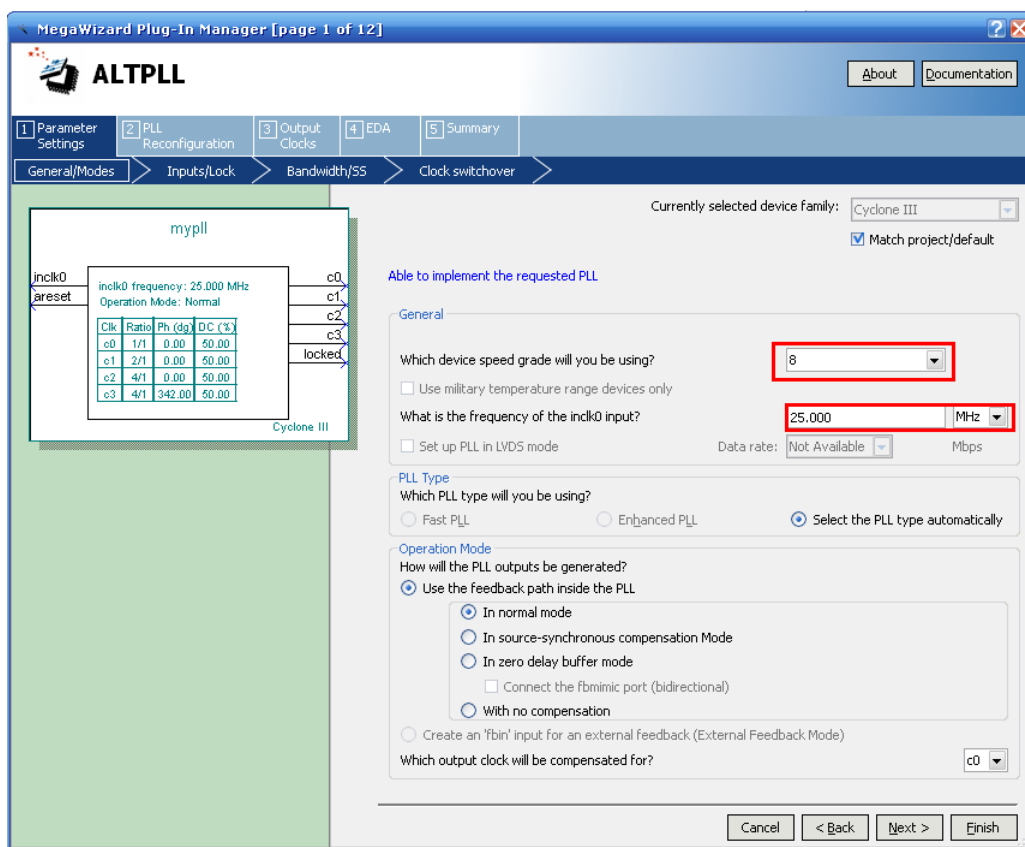


PLL 的配置步骤如下。

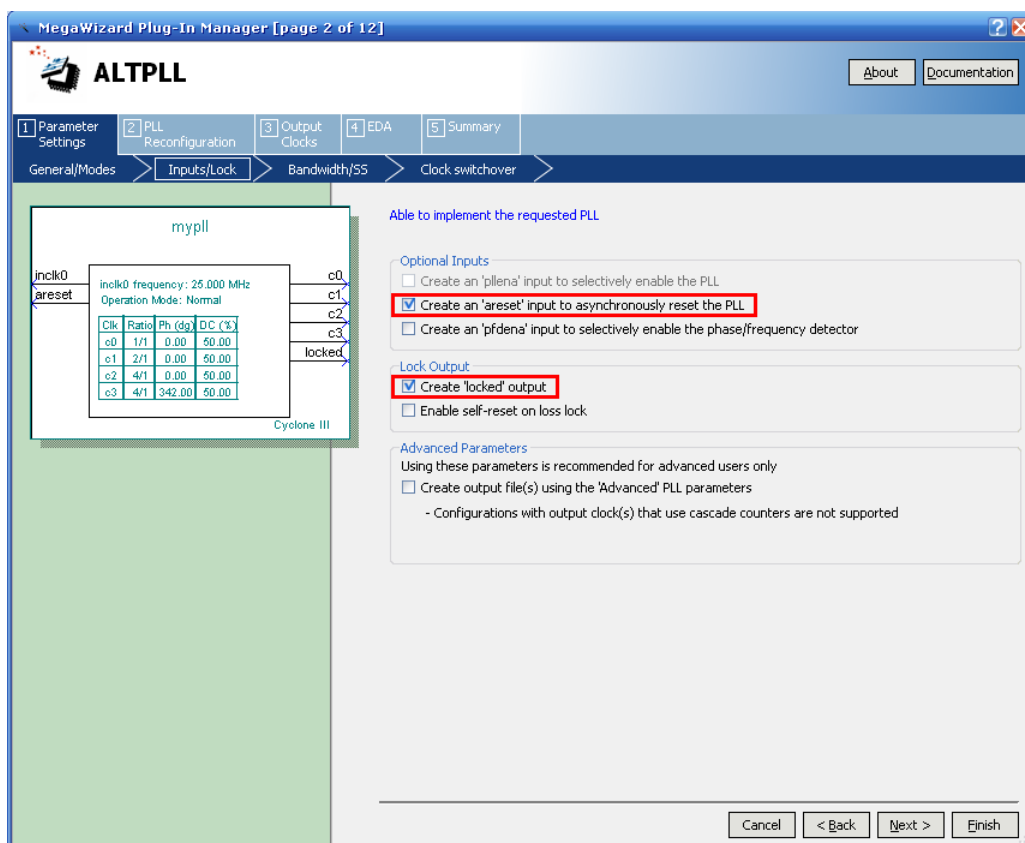
① 点击菜单栏的 Tools→MegaWizard Plug-In Manager，new 一个 megafunction。在 page 2a 的页面中做如图设置，注意这个新 PLL 的名称输入为 mypll，路径为当前工程所在路径。



② 器件速度等级为 8，PLL 的输入时钟频率为 25MHz。

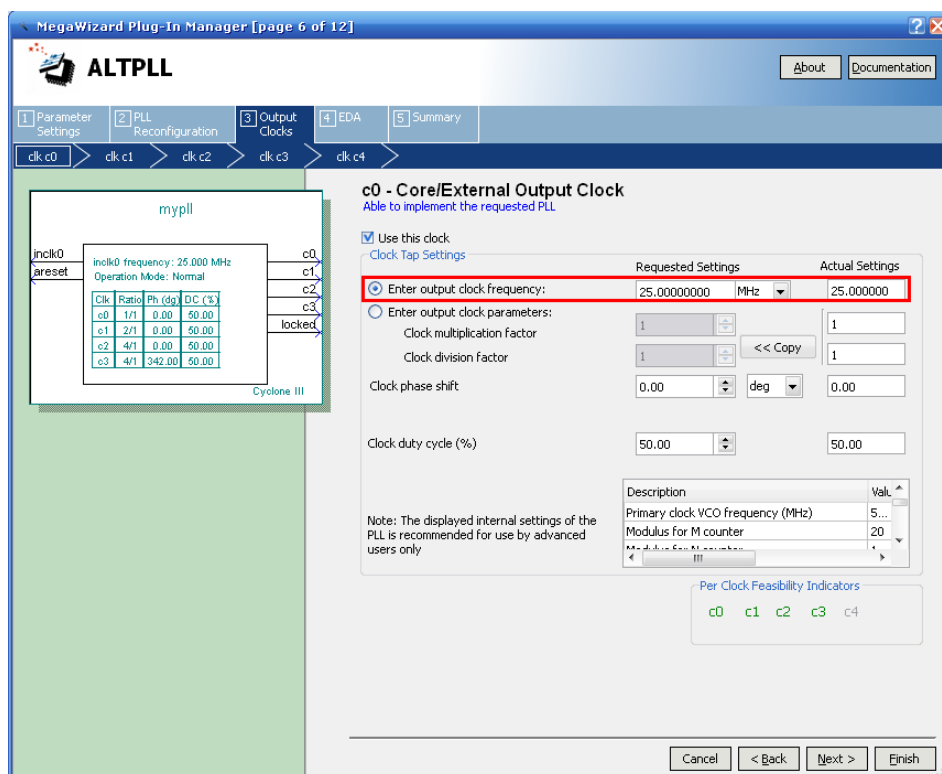


③ 使能复位输入信号 areset 和 PLL 有效指示输出信号 locked。

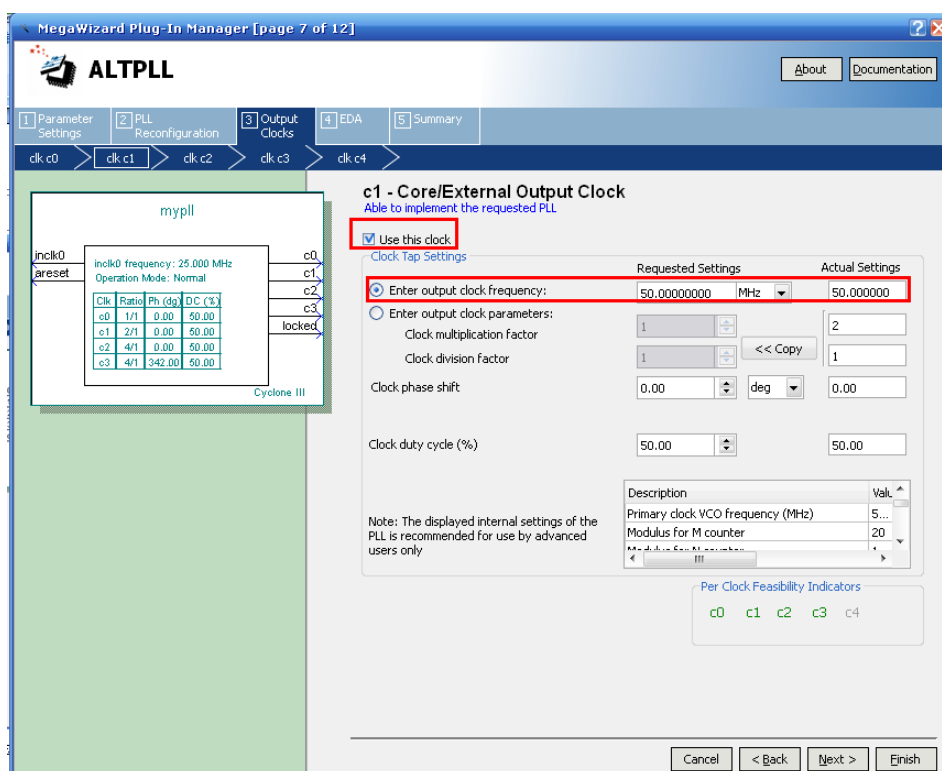




④ PLL 输出时钟 c0 的频率为 25MHz, 相位为 0, FPGA 内部的基本逻辑模块都是用这个时钟。



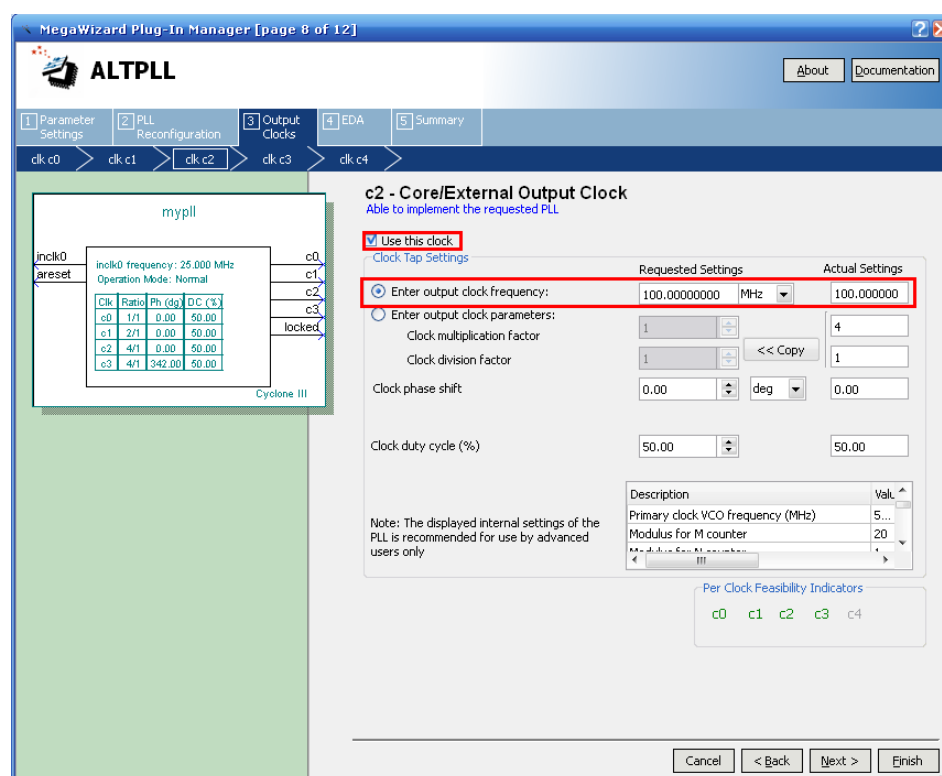
PLL 输出时钟 c1 的频率为 50MHz, 相位为 0。



PLL 输出时钟 c2 的频率为 100MHz, 相位为 0, 该时钟用于 FPGA 内部驱动 SDRAM 控制

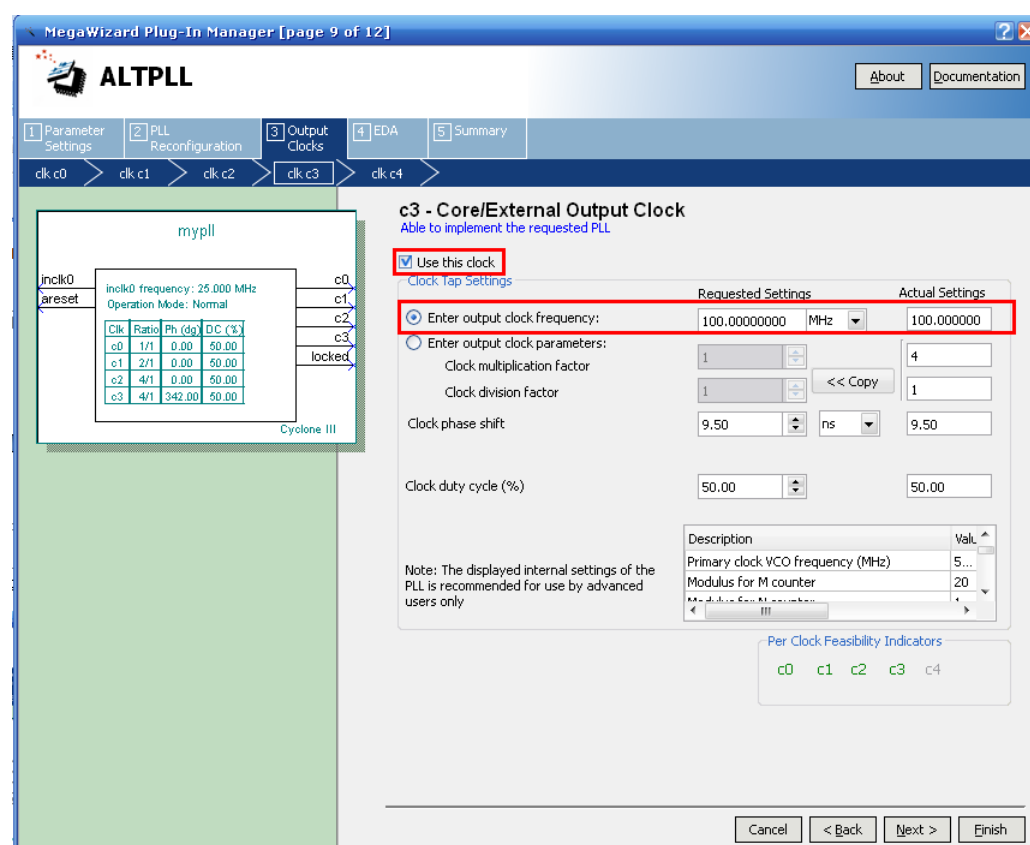


器的逻辑。



PLL 输出时钟 c2 的频率为 100MHz, 相位为 9.5ns, 该时钟用于 SDRAM 的时钟 sdram_clk。

c3 则无需开启。





该模块的代码如下。

```
module sys_ctrl(
    clk, rst_n, sys_rst_n,
    clk_25m, clk_50m, clk_100m, sdram_clk
);

input clk;      //FPAG 输入时钟信号 25MHz
input rst_n;    //FPGA 输入复位信号

output sys_rst_n; //系统复位信号, 低有效
output clk_25m;   //PLL 输出 25MHz 时钟
output clk_50m;   //PLL 输出 50MHz 时钟
output clk_100m;  //PLL 输出 100MHz 时钟
output sdram_clk; //用于外部 SDAM 的时钟 100M

wire locked;      //PLL 输出有效标志位, 高表示 PLL 输出有效

//-----
//PLL 复位信号产生, 高有效
//异步复位, 同步释放
wire pll_rst;    //PLL 复位信号, 高有效

reg rst_r1, rst_r2;

always @(posedge clk or negedge rst_n)
    if(!rst_n) rst_r1 <= 1'b1;
    else rst_r1 <= 1'b0;

always @(posedge clk or negedge rst_n)
    if(!rst_n) rst_r2 <= 1'b1;
    else rst_r2 <= rst_r1;

assign pll_rst = rst_r2;

//-----
//系统复位信号产生, 低有效
//异步复位, 同步释放
wire sys_rst_n; //系统复位信号, 低有效
```



```
wire sysrst_nr0;
reg sysrst_nrl,sysrst_nr2;

assign sysrst_nr0 = rst_n & locked; //系统复位直到 PLL 有效输出

always @(posedge clk_100m or negedge sysrst_nr0)
    if(!sysrst_nr0) sysrst_nrl <= 1'b0;
    else sysrst_nrl <= 1'b1;

always @(posedge clk_100m or negedge sysrst_nr0)
    if(!sysrst_nr0) sysrst_nr2 <= 1'b0;
    else sysrst_nr2 <= sysrst_nrl;

assign sys_rst_n = sysrst_nr2;

//-----
//例化 PLL 产生模块
mypll      mypll_inst(
                .areset(pll_rst),    //PLL 复位信号, 高电平复位
                .inclk0(clk),        //PLL 输入时钟, 25MHz
                .c0(clk_25m),        //PLL 输出 25MHz 时钟
                .c1(clk_50m),        //PLL 输出 50MHz 时钟
                .c2(clk_100m),       //PLL 输出 100MHz 时钟
                .c3(sdram_clk),      //用于外部 SDRAM 的时钟 100M
                .locked(locked)      //PLL 输出有效标志位, 高表示 PLL 输出有效
            );

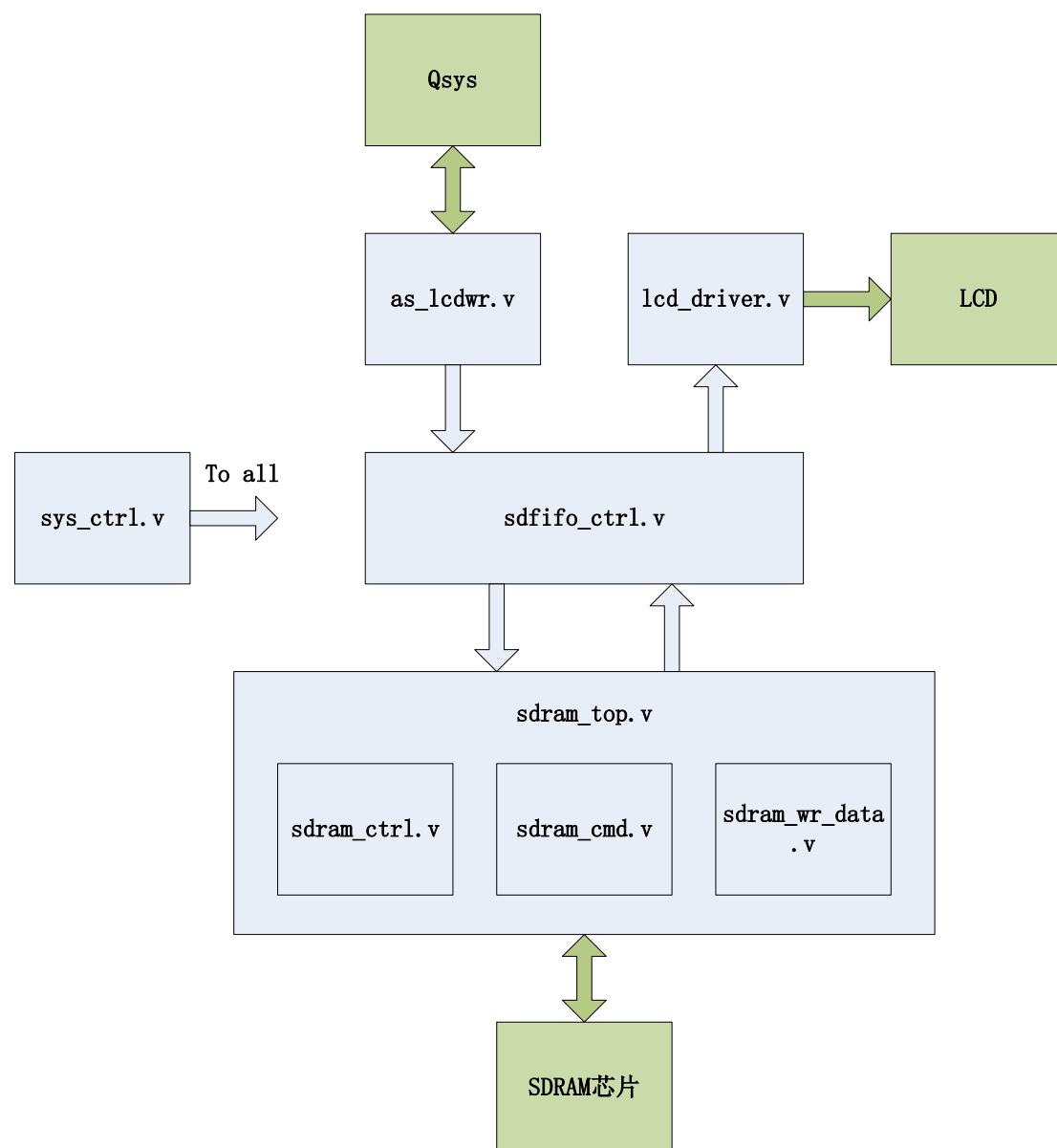
endmodule
```

7.6.7 Qsys 系统构建

我们先简单的把前面的各个模块的关系理一下。以下的模块基本涵盖了前面提到的所有新设计的模块。Sys_ctrl.v 模块产生系统各个模块所需的时钟和复位信号; As_lcdwr.v 模块用于衔接 Qsys 的总线控制以及写入数据到 Sdram 中; Sdfifo_ctrl.v 模块中主要配置了两个 FIFO, 用户缓存读和写数据到 SDRAM 的缓存控制; sdram_top.v 以及其下例化的 3 个模块主要是用

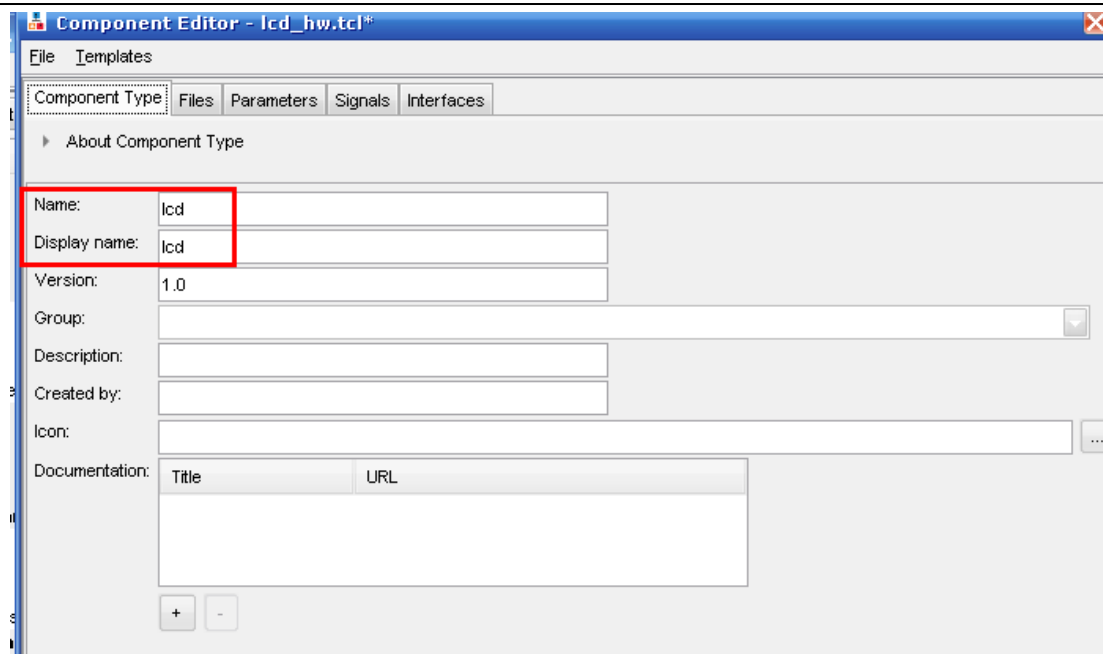


于实际的 SDRAM 芯片读写和控制操作；`lcd_driver.v` 模块用于产生 LCD 的驱动时序，同时发出 LCD 每个显示行列需要的读数据请求，接着从 `sdfifo_ctrl.v` 模块的数据读缓存 FIFO 中得到需要送往 LCD 显示的数据。

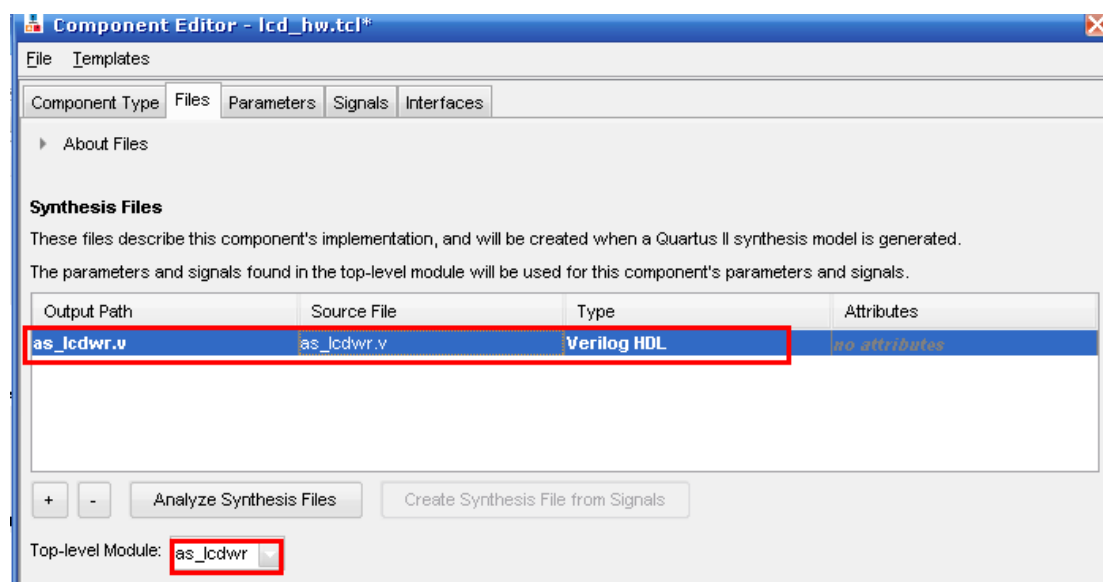


前面已经将各个逻辑模块的设计介绍了一遍，下面我们还差一步才能够完成硬件系统的构建，那边是将 `as_lcdwr.v` 这个模块作为 `Qsys` 的组件集成到系统中。

进入工程，打开 `Qsys`。点击界面左侧 `Project→New Component...`（基本步骤和前面几个例程相应的操作类似，这里不做详细讲解，只是将重要的设置参数简单讲解下），在 `Component Type` 页面中设置如图，`name` 和 `Display name` 均为 `lcd`。



Files 页面中, 点击“+”添加 as_lcdwr.v 文件, 进行 Synthesis, 然后设置它为 Top-level Module。



Signals 页面中, 做如图的配置。接口信号的具体含义前面已经讲解过, 如果不理解这里的设置, 大家可以回头对照信号接口的定义来消化。



Component Editor - lcd_hw.tcl				
File Templates				
Component Type Files Parameters Signals Interfaces				
About Signals				
Name	Interface	Signal Type	Width	Direction
clk	clock_sink	clk	1	input
rst_n	reset_sink	reset_n	1	input
avalone_cs_n	avalon_slave	chipselect_n	1	input
avalone_wr_n	avalon_slave	write_n	1	input
avalone_wraddr	avalon_slave	address	22	input
avalone_wrdata	avalon_slave	writedata	16	input
wrf_din	conduit_end	export	16	output
wrf_wrreq	conduit_end	export	1	output
wrf_ain	conduit_end	export	22	output

Interface 页面的配置如后面的几个图所示。

"clock_sink" (Clock Input)

Name: Documentation

Type:

Block Diagram

Parameters

Clock rate:

"reset_sink" (Reset Input)

Name: Documentation

Type:

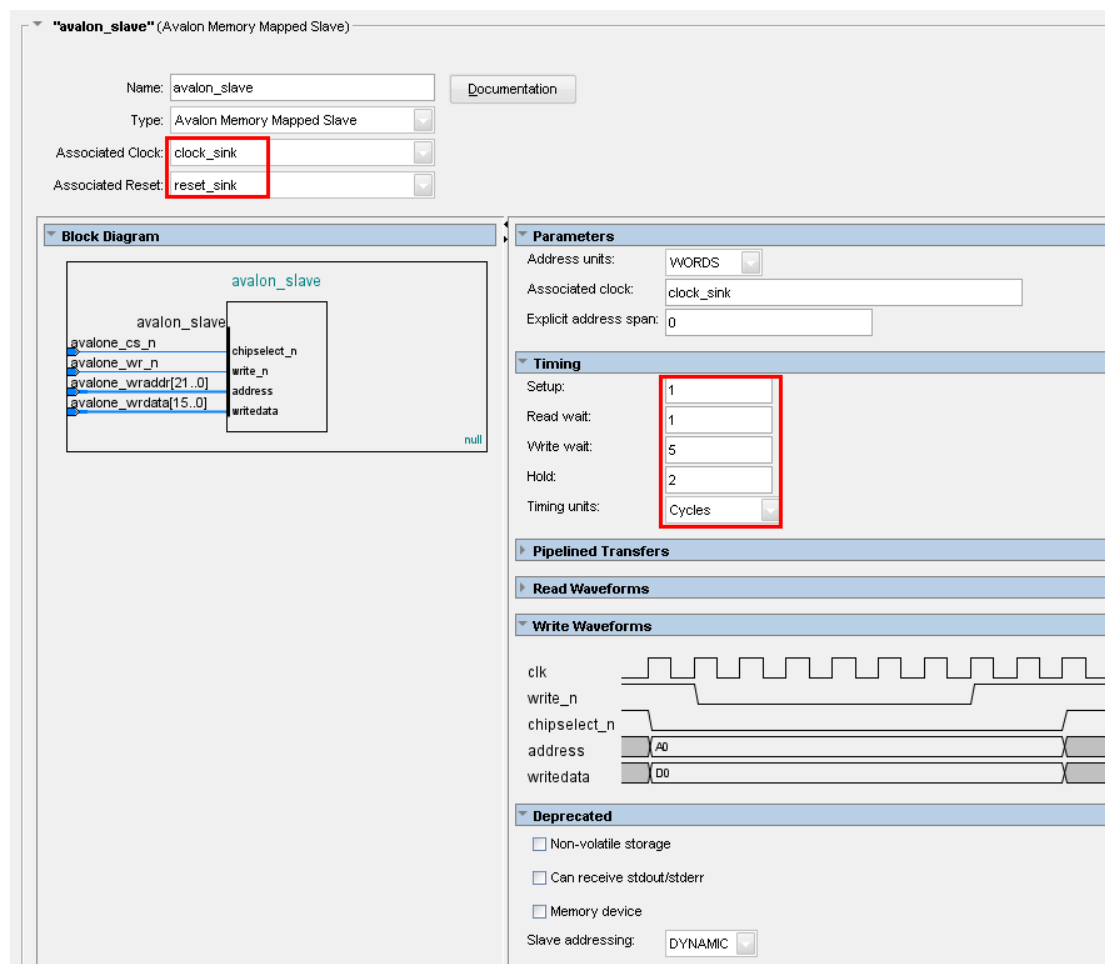
Associated Clock:

Block Diagram

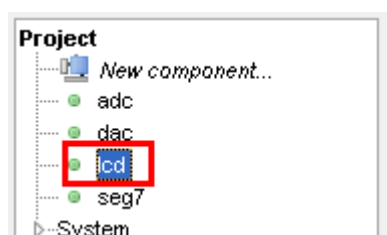
Parameters

Associated clock:

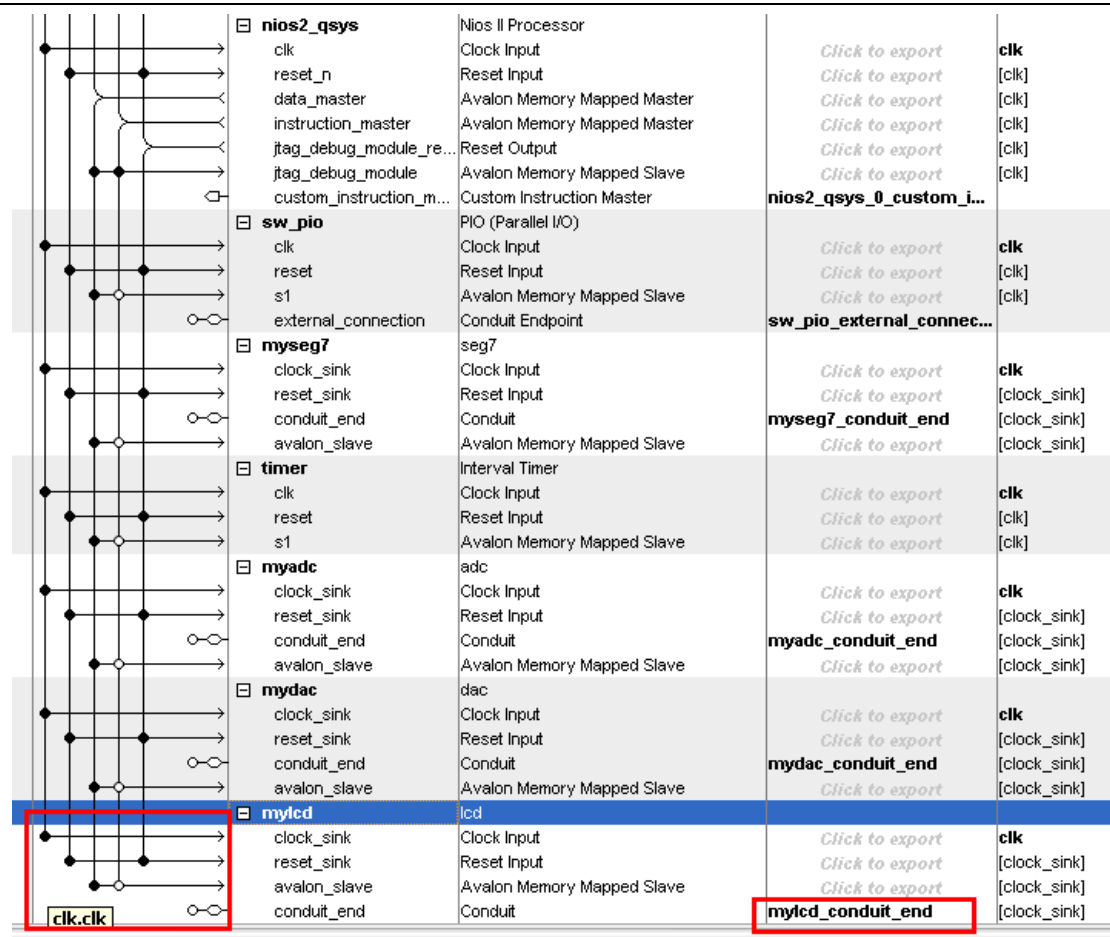
Synchronous edges:



完成配置后，我们看到 Project 栏下面多了一个 lcd 组件，双击添加这个组件。



将这个新添加的组件改名为 mylcd。Clock_sink 连接到系统的 clock 上；reset_sink 连接到系统的 reset 上；avalon_slave 连接到系统的 nios2_qsys 的 data_master 上；conduit_end 连接到 qsys 的外部。最终设置如图。



点击菜单栏的 **System**→**Assign Base Addresses**，对系统地址重新做分配。然后来到 **Generation** 页面，点击右下角的 **Generate** 生成新系统。

最后，回到工程顶层文件，对 **ex2.v** 文件的代码做如下修改。

```
module ex2(
    clk, rst_n, led,
    sw_pio,
    seg_db, seg_cs,
    adc_data, adc_cs_n, adc_clk,
    dac_scl, dac_sda,
    lcd_en, lcd_clk, lcd_hsy, lcd_vsy, lcd_db_r, lcd_db_g, lcd_db_b,
    sdram_clk, sdram_cke, sdram_cs_n, sdram_ras_n, sdram_cas_n, sdram_we_n,
    sdram_ba, sdram_addr, sdram_data
);
input clk;
input rst_n;
output led;
input[2:0] sw_pio;
```



```
output[7:0] seg_db;
output[3:0] seg_cs;
input adc_data;      //TLC549 数据信号
output adc_cs_n;     //TLC549 片选信号, 低电平有效
output adc_clk;      //TLC549 时钟信号
output dac_scl;      //DAC5571 数据信号
inout dac_sda;       //DAC5571 时钟信号
    // FPGA 与 LCD 接口信号
output lcd_en; //背光使能信号, 高有效
output lcd_clk; //时钟信号
output lcd_hsy; //行同步信号
output lcd_vsy; //场同步信号
output[4:0] lcd_db_r;
output[5:0] lcd_db_g;
output[4:0] lcd_db_b;
    // FPGA 与 SDRAM 硬件接口
output sdram_clk;      // SDRAM 时钟信号
output sdram_cke;      // SDRAM 时钟有效信号
output sdram_cs_n;     // SDRAM 片选信号
output sdram_ras_n;    // SDRAM 行地址选通脉冲
output sdram_cas_n;    // SDRAM 列地址选通脉冲
output sdram_we_n;     // SDRAM 写允许位
output[1:0] sdram_ba;   // SDRAM 的 L-Bank 地址线
output[11:0] sdram_addr; // SDRAM 地址总线
inout[15:0] sdram_data; // SDRAM 数据总线

//-----
    //LCD 与 FIFO 的接口
wire[15:0] rdfifo_rddb; //FIFO 读出数据总线
wire rdfifo_rdreq; //FIFO 读请求信号
wire rdfifo_clr;      //FIFO 复位信号, 高电平有效
    // SDRAM 的封装接口
wire sdram_wr_req;     //系统写 SDRAM 请求信号
wire sdram_rd_req;     //系统读 SDRAM 请求信号
wire sdram_wr_ack;     //系统写 SDRAM 响应信号, 作为 wrFIFO 的输出有效信号
wire sdram_rd_ack;     //系统读 SDRAM 响应信号, 作为 rdFIFO 的输写有效信号

wire[8:0] sdwr_byte = 9'd1; //突发写 SDRAM 字节数 (1-256 个)
```



```
wire[8:0] sdrd_byte = 9'd160; //突发读 SDRAM 字节数 (1-256 个)
wire[21:0] sys_wraddr; // 写 SDRAM 时地址暂存器, (bit21-20)L-Bank 地址: (bit19-8)为行地址, (bit7-0)为列地址
wire[21:0] sys_rdaddr; // 读 SDRAM 时地址暂存器, (bit21-20)L-Bank 地址: (bit19-8)为行地址, (bit7-0)为列地址
wire[15:0] sys_data_in; //写 SDRAM 时数据暂存器
wire[15:0] sys_data_out; //sdrd 数据读出缓存 FIFO 输入数据总线
//wrFIFO 输入控制接口
wire[15:0] wrf_din; //sdrd 数据写入缓存 FIFO 输入数据总线
wire[21:0] wrf_ain; //sdrd 数据写入缓存 FIFO 输入地址总线
wire wrf_wrreq; //sdrd 数据写入缓存 FIFO 数据输入请求, 高有效
//系统控制相关信号接口
wire clk_25m; //PLL 输出 25MHz 时钟
wire clk_50m; //PLL 输出 50MHz 时钟
wire clk_100m; //PLL 输出 100MHz 时钟
wire sys_rst_n; //系统复位信号, 低有效

//-----
//例化系统复位信号和 PLL 控制模块
sys_ctrl uut_sysctrl(
    .clk(clk),
    .rst_n(rst_n),
    .sys_rst_n(sys_rst_n),
    .clk_25m(clk_25m),
    .clk_50m(clk_50m),
    .clk_100m(clk_100m),
    .sdrd_clk(sdrd_clk)
);

//-----
//Qsys 系列例化
myqsys u0 (
    //clk.clk
    .clk_clk (clk_25m),
    //pio_0_external_connection.export
    .pio_0_external_connection_export(led),
    //clk_0_clk_in_reset.reset_n
    .clk_0_clk_in_reset_reset_n(sys_rst_n),
```



```
//nios2_qsys_0_custom_instruction_master.readra
.nios2_qsys_0_custom_instruction_master_readra ( ),
    //sw_pio_external_connection.export
.sw_pio_external_connection_export(sw_pio),
    //myseg7_conduit_end.db
.myseg7_conduit_end_db (seg_db),
    //.cs
.myseg7_conduit_end_cs(seg_cs),
    //mydac_conduit_end.scl
.mydac_conduit_end_scl(dac_scl),
    //.sda
.mydac_conduit_end_sda(dac_sda),
    //myadc_conduit_end.data
.myadc_conduit_end_data(adc_data),
    //.cs_n
.myadc_conduit_end_cs_n(adc_cs_n),
    //.clk
.myadc_conduit_end_clk(adc_clk),
    //mylcd_conduit_end.din
.mylcd_conduit_end_din(wrf_din),
    //.wrreq
.mylcd_conduit_end_wrreq(wrf_wrreq),
    //.ain
.mylcd_conduit_end_ain(wrf_ain)
);

//-----
//LCD 驱动模块
lcd_driver uut_lcd_driver(
    .clk(clk_25m), //25MHz
    .rst_n(sys_rst_n),
    .lcd_en(lcd_en),
    .lcd_clk(lcd_clk),
    .lcd_hsy(lcd_hsy),
    .lcd_vsy(lcd_vsy),
    .lcd_db_r(lcd_db_r),
    .lcd_db_g(lcd_db_g),
    .lcd_db_b(lcd_db_b),
```




```
.rdfifo_rddb(rdfifo_rddb),//
.rdfifo_rdreq(rdfifo_rdreq),//
.rdfifo_clr(rdfifo_clr)//
);

//-----
//例化 SDRAM 封装控制模块
sdram_top      uut_sdramtop(          // SDRAM
    .clk(clk_100m),
    .rst_n(sys_rst_n),
    .sdram_wr_req(sdram_wr_req),//
    .sdram_rd_req(sdram_rd_req),//
    .sdram_wr_ack(sdram_wr_ack),//
    .sdram_rd_ack(sdram_rd_ack),//
    .sys_wraddr(sys_wraddr),//
    .sys_rdaddr(sys_rdaddr),//
    .sys_data_in(sys_data_in),//
    .sys_data_out(sys_data_out),//
    .sdwr_byte(sdwr_byte),//
    .sdrd_byte(sdrd_byte),//
    .sdram_cke(sdram_cke),
    .sdram_cs_n(sdram_cs_n),
    .sdram_ras_n(sdram_ras_n),
    .sdram_cas_n(sdram_cas_n),
    .sdram_we_n(sdram_we_n),
    .sdram_ba(sdram_ba),
    .sdram_addr(sdram_addr),
    .sdram_data(sdram_data),
    .sdram_udqm(),
    .sdram_ldqm()
);

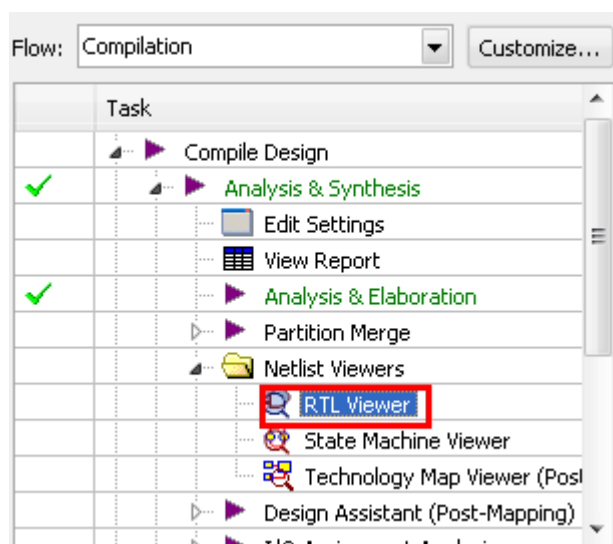
//-----
//读写 SDRAM 数据缓存 FIFO 模块例化
sdfifo_ctrl      uut_sdffifoctrl(
    .clk_25m(clk_25m),
    .clk_100m(clk_100m),
```



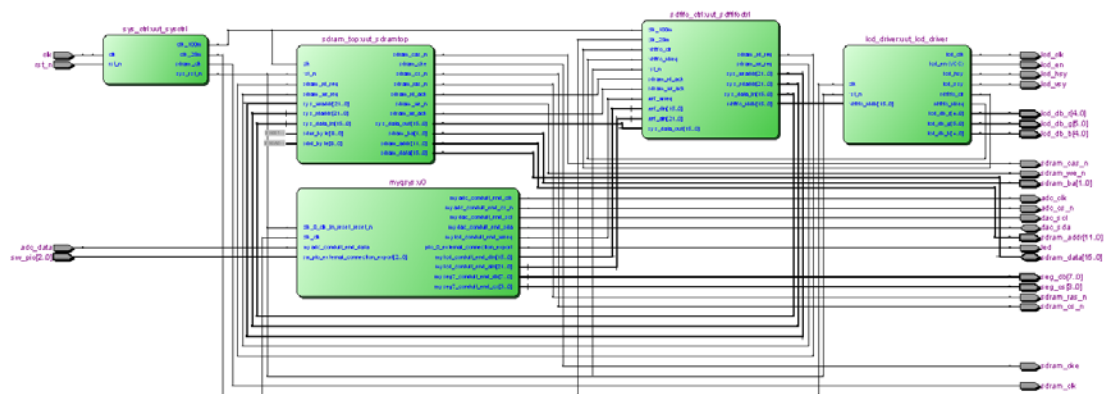
```
.rst_n(sys_rst_n),  
.wrf_din(wrf_din),  
.wrf_wrreq(wrf_wrreq),  
.wrf_ain(wrf_ain),  
.sdram_wr_ack(sdram_wr_ack), //  
.sys_wraddr(sys_wraddr), //  
.sys_rdaddr(sys_rdaddr), //  
.sys_data_in(sys_data_in), //  
.sdram_wr_req(sdram_wr_req), //  
.sys_data_out(sys_data_out), //  
.sdram_rd_ack(sdram_rd_ack), //  
.sdram_rd_req(sdram_rd_req), //  
.rdfifo_rdreq(rdfifo_rdreq), //  
.rdfifo_clr(rdfifo_clr), //  
.rdfifo_rddb(rdfifo_rddb) //
```

endmodule

接着可以先对工程做一次全编译。大家可以通过 **RTL Viewer** 来查看工程的整体模块连接以及接口状态。



本工程的 **RTL Viewer** 视图如下，这里几个主要的模块和接口一目了然，大家也可以点击各个具体模块，看看模块内部的架构和接口。RTL 视图尤其对于从一个比较高层次角度阅读代码很有帮助。



7.6.8 管脚分配与时序约束

对管脚分配如下。

```
set_location_assignment PIN_103 -to sw_pio[0]
set_location_assignment PIN_99 -to seg_cs[3]
set_location_assignment PIN_98 -to seg_cs[2]
set_location_assignment PIN_87 -to seg_cs[1]
set_location_assignment PIN_100 -to seg_cs[0]
set_location_assignment PIN_120 -to seg_db[7]
set_location_assignment PIN_115 -to seg_db[6]
set_location_assignment PIN_121 -to seg_db[5]
set_location_assignment PIN_126 -to seg_db[4]
set_location_assignment PIN_124 -to seg_db[3]
set_location_assignment PIN_119 -to seg_db[2]
set_location_assignment PIN_114 -to seg_db[1]
set_location_assignment PIN_125 -to seg_db[0]
set_location_assignment PIN_113 -to adc_clk
set_location_assignment PIN_111 -to adc_cs_n
set_location_assignment PIN_112 -to adc_data
set_location_assignment PIN_106 -to dac_scl
set_location_assignment PIN_110 -to dac_sda
set_global_assignment -name VERILOG_FILE sys_ctrl.v
set_global_assignment -name VERILOG_FILE sdr_wr_data.v
set_global_assignment -name VERILOG_FILE sdr_ctrl.v
set_global_assignment -name VERILOG_FILE sdr_top.v
set_global_assignment -name VERILOG_FILE sdr_cmd.v
set_global_assignment -name VERILOG_FILE sdr_para.v
```

《圣经》箴言九 11 “敬畏耶和华是智慧的开端，认识至胜者便是聪明。”

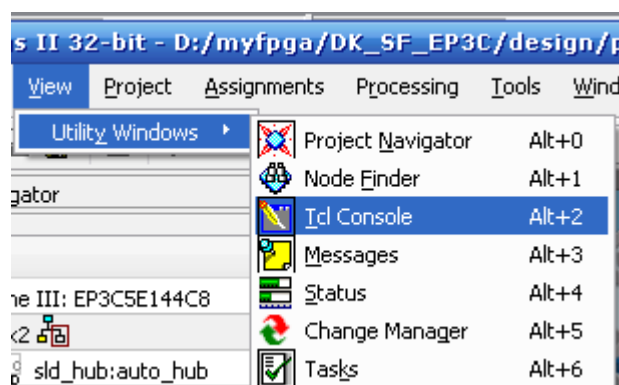


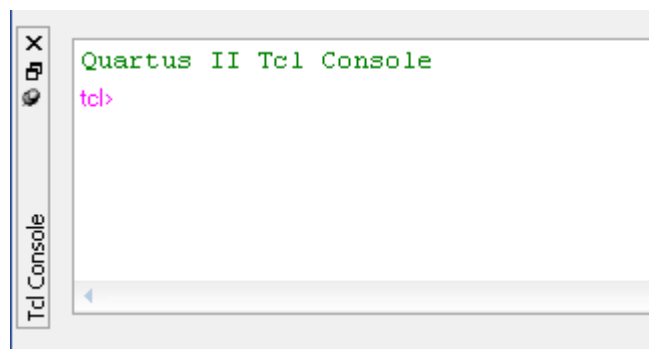
```
set_global_assignment -name VERILOG_FILE sdfifo_ctrl.v
set_global_assignment -name VERILOG_FILE lcd_driver.v
set_global_assignment -name QSYS_FILE myqsys.qsys
set_global_assignment -name VERILOG_FILE ex2.v
set_global_assignment -name QIP_FILE mypll.qip
set_global_assignment -name QIP_FILE rdfifo.qip
set_global_assignment -name QIP_FILE wrfifo.qip
set_location_assignment PIN_60 -to lcd_clk
set_location_assignment PIN_77 -to lcd_db_b[4]
set_location_assignment PIN_79 -to lcd_db_b[3]
set_location_assignment PIN_80 -to lcd_db_b[2]
set_location_assignment PIN_83 -to lcd_db_b[1]
set_location_assignment PIN_84 -to lcd_db_b[0]
set_location_assignment PIN_71 -to lcd_db_g[5]
set_location_assignment PIN_72 -to lcd_db_g[4]
set_location_assignment PIN_73 -to lcd_db_g[3]
set_location_assignment PIN_74 -to lcd_db_g[2]
set_location_assignment PIN_75 -to lcd_db_g[1]
set_location_assignment PIN_76 -to lcd_db_g[0]
set_location_assignment PIN_66 -to lcd_db_r[4]
set_location_assignment PIN_67 -to lcd_db_r[3]
set_location_assignment PIN_68 -to lcd_db_r[2]
set_location_assignment PIN_69 -to lcd_db_r[1]
set_location_assignment PIN_70 -to lcd_db_r[0]
set_location_assignment PIN_85 -to lcd_en
set_location_assignment PIN_65 -to lcd_hsy
set_location_assignment PIN_64 -to lcd_vsy
set_location_assignment PIN_46 -to sdram_addr[11]
set_location_assignment PIN_135 -to sdram_addr[10]
set_location_assignment PIN_49 -to sdram_addr[9]
set_location_assignment PIN_50 -to sdram_addr[8]
set_location_assignment PIN_51 -to sdram_addr[7]
set_location_assignment PIN_52 -to sdram_addr[6]
set_location_assignment PIN_53 -to sdram_addr[5]
set_location_assignment PIN_54 -to sdram_addr[4]
set_location_assignment PIN_128 -to sdram_addr[3]
set_location_assignment PIN_129 -to sdram_addr[2]
set_location_assignment PIN_132 -to sdram_addr[1]
```



```
set_location_assignment PIN_133 -to sdram_addr[0]
set_location_assignment PIN_136 -to sdram_ba[1]
set_location_assignment PIN_137 -to sdram_ba[0]
set_location_assignment PIN_142 -to sdram_cas_n
set_location_assignment PIN_44 -to sdram_cke
set_location_assignment PIN_138 -to sdram_cs_n
set_location_assignment PIN_34 -to sdram_data[15]
set_location_assignment PIN_33 -to sdram_data[14]
set_location_assignment PIN_32 -to sdram_data[13]
set_location_assignment PIN_31 -to sdram_data[12]
set_location_assignment PIN_30 -to sdram_data[11]
set_location_assignment PIN_38 -to sdram_data[10]
set_location_assignment PIN_39 -to sdram_data[9]
set_location_assignment PIN_42 -to sdram_data[8]
set_location_assignment PIN_144 -to sdram_data[7]
set_location_assignment PIN_1 -to sdram_data[6]
set_location_assignment PIN_2 -to sdram_data[5]
set_location_assignment PIN_3 -to sdram_data[4]
set_location_assignment PIN_4 -to sdram_data[3]
set_location_assignment PIN_7 -to sdram_data[2]
set_location_assignment PIN_10 -to sdram_data[1]
set_location_assignment PIN_11 -to sdram_data[0]
set_location_assignment PIN_141 -to sdram_ras_n
set_location_assignment PIN_143 -to sdram_we_n
set_location_assignment PIN_43 -to sdram_clk
```

大家可以通过打开 **Tcl Console** 面板, 通过输入以上脚本完成管脚分配。

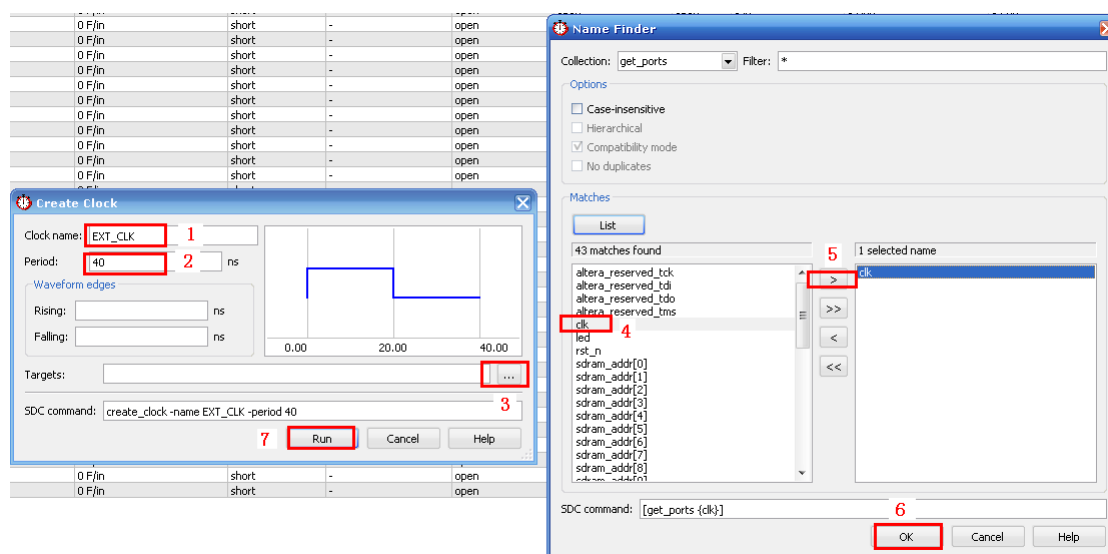




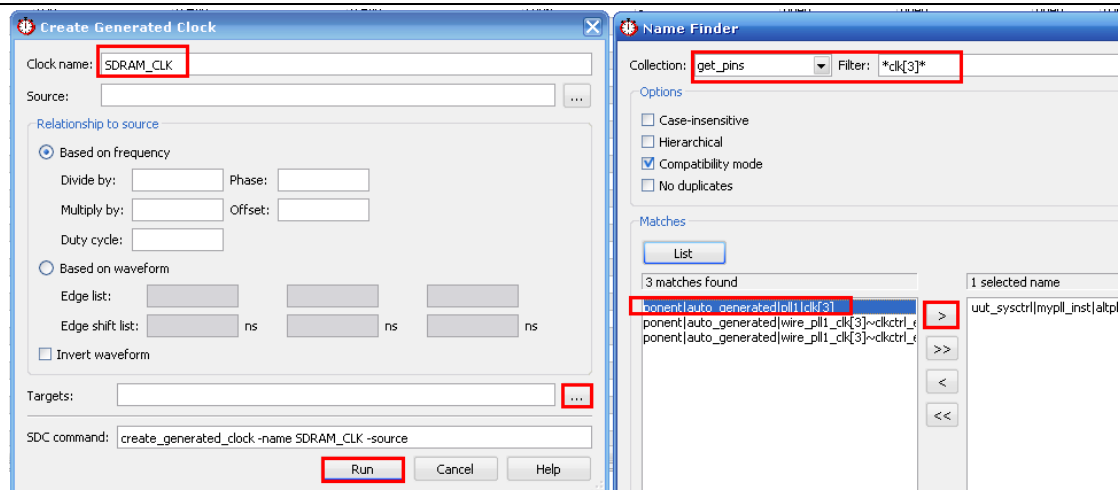
接下来要进行时序约束，某些具体步骤请参考 5.5.4 节。

进入当前工程的 TimeQuest，首先新建一个 SDC 文件。

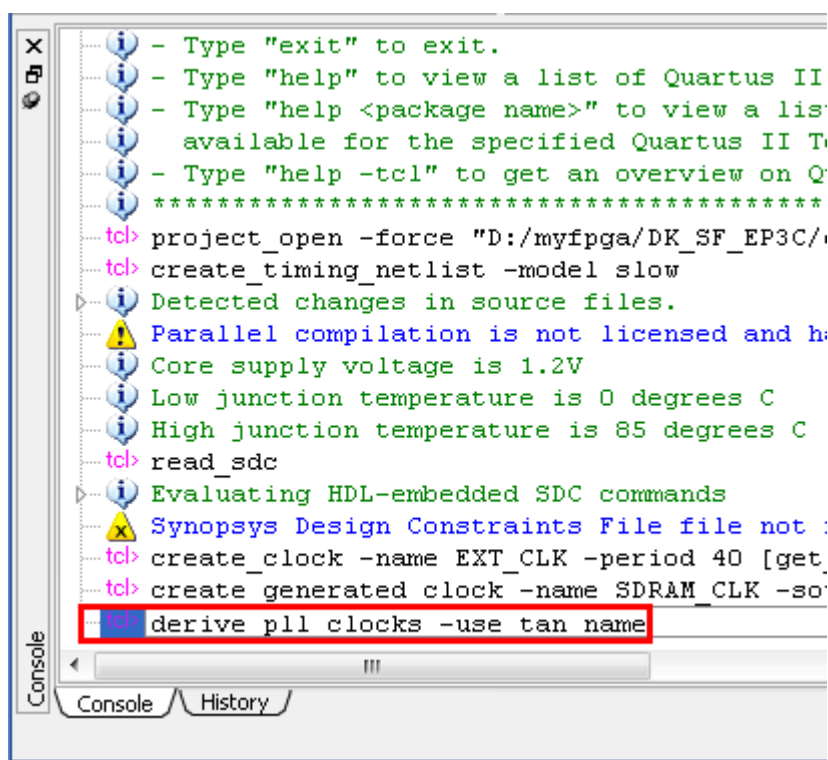
约束全局输入时钟，点击菜单栏 Constraints→Create Clock。设置时钟名称为 EXT_CLK、时钟周期 40ns，然后选择 Targets 即目标管脚为 clk。



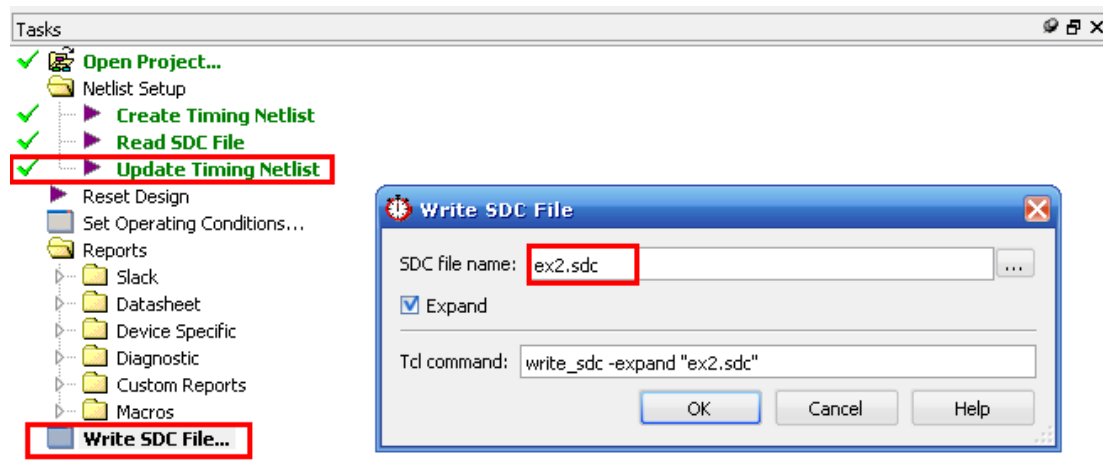
约束 SDRAM 输出时钟，点击菜单栏 Constraints→Create Generated Clock。输入名称为 SDRAM_CLK，然后点击 Targets 后的按钮，弹出的新窗口中选择 get_pins，关键词*clk[3]*，找到时钟信号 uut_sysctrl|mypll_inst|altpll_component|auto_generated|pll1|clk[3]。



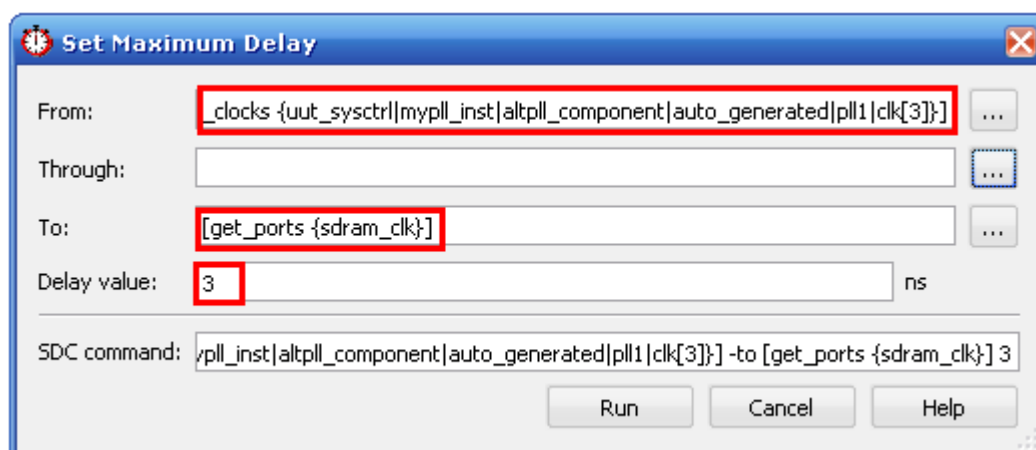
在 Console 中输入 `tcl>derive_pll_clocks -use_tan_name`。让 TimeQuest 自动去约束所有的 PLL 输出时钟。



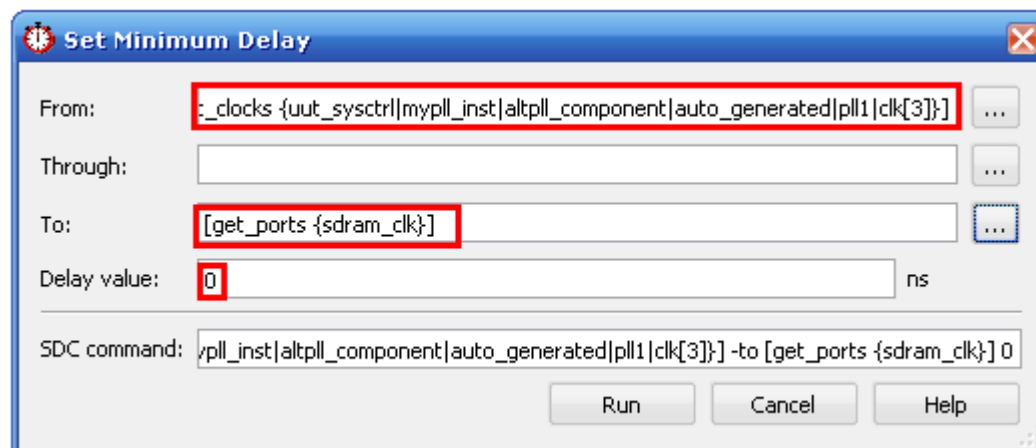
接下来我们分别点击 Tasks 面板的 Update Timing Netlist 和 Write SDC File, 然后在弹出的对话框中输入 ex2.sdc 文件, 点击 OK 完成 sdc 文件的创建, 刚才的几个脚本也都写入 sdc 文件中。



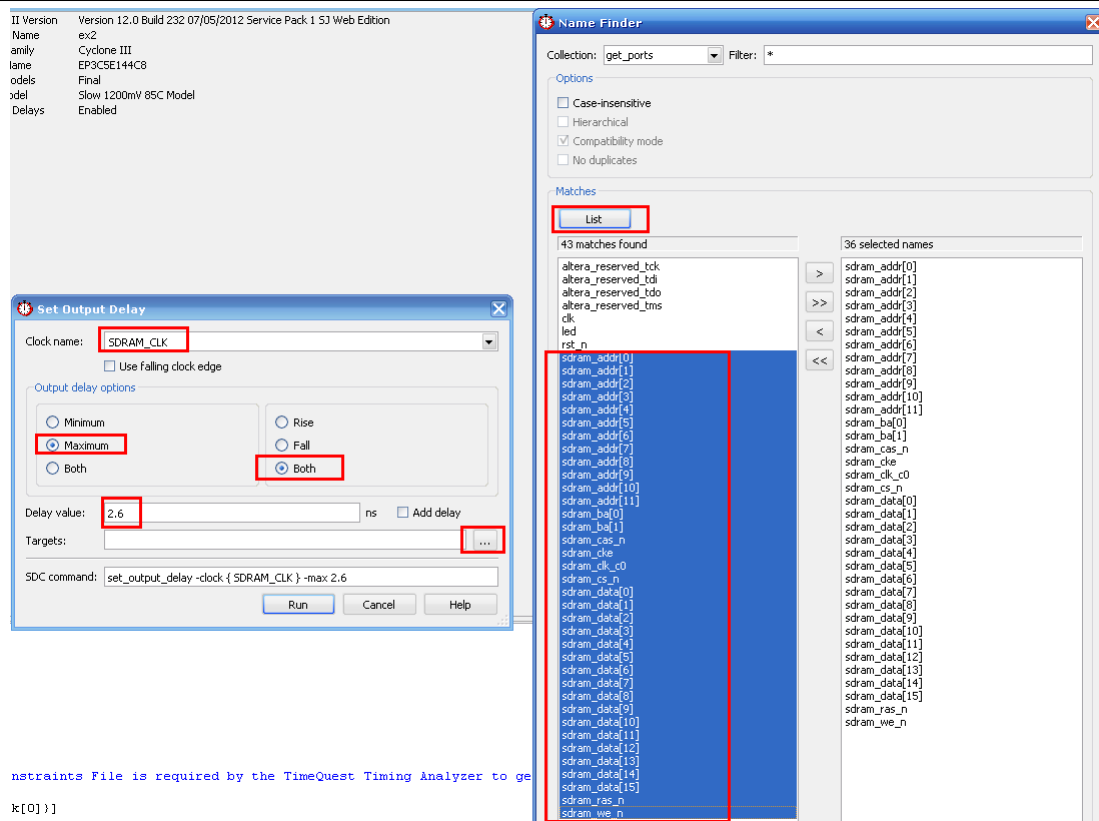
下面需要对 PLL 输出的 c3 时钟到达 FPGA 的 pin 上的延时做约束, 点击菜单栏 Constraints→Set Maximum Delay, 做如下约束。



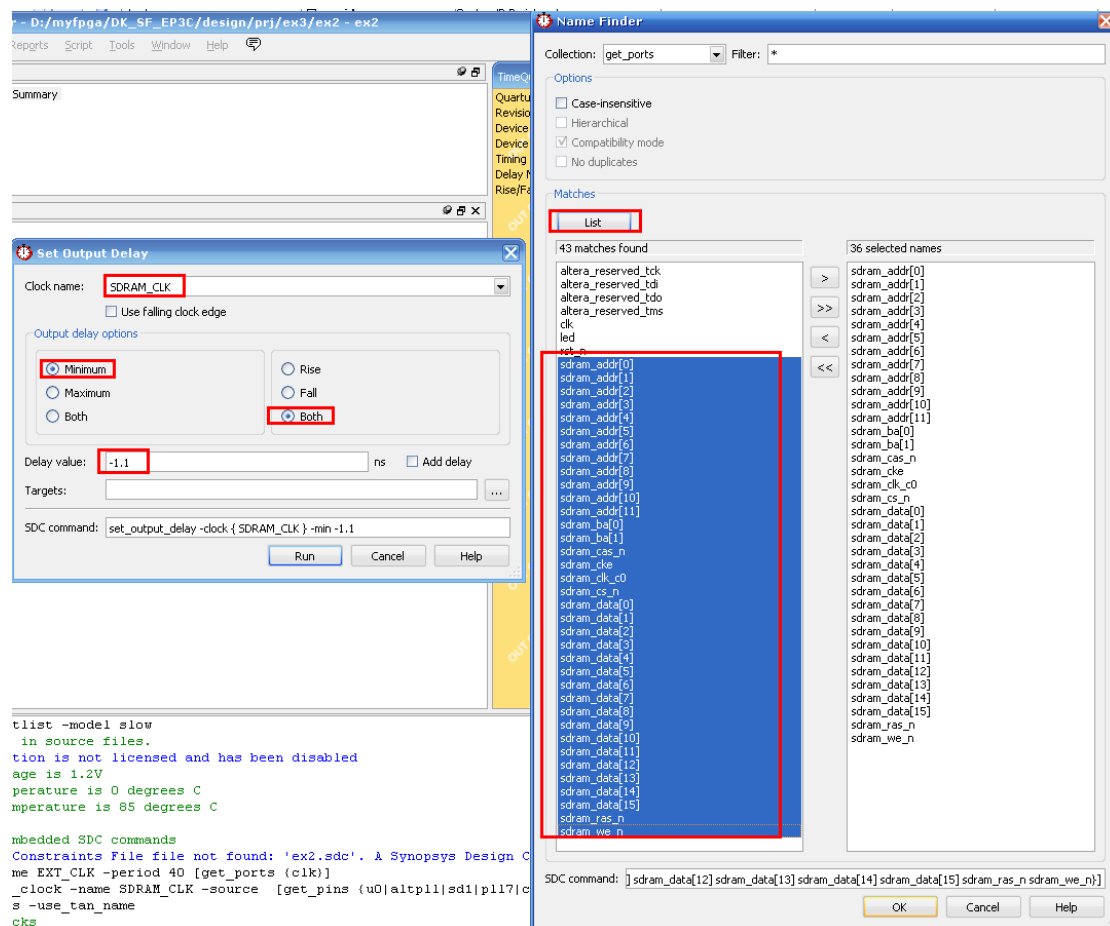
点击菜单栏 Constraints→Set Minimum Delay, 做如下约束。



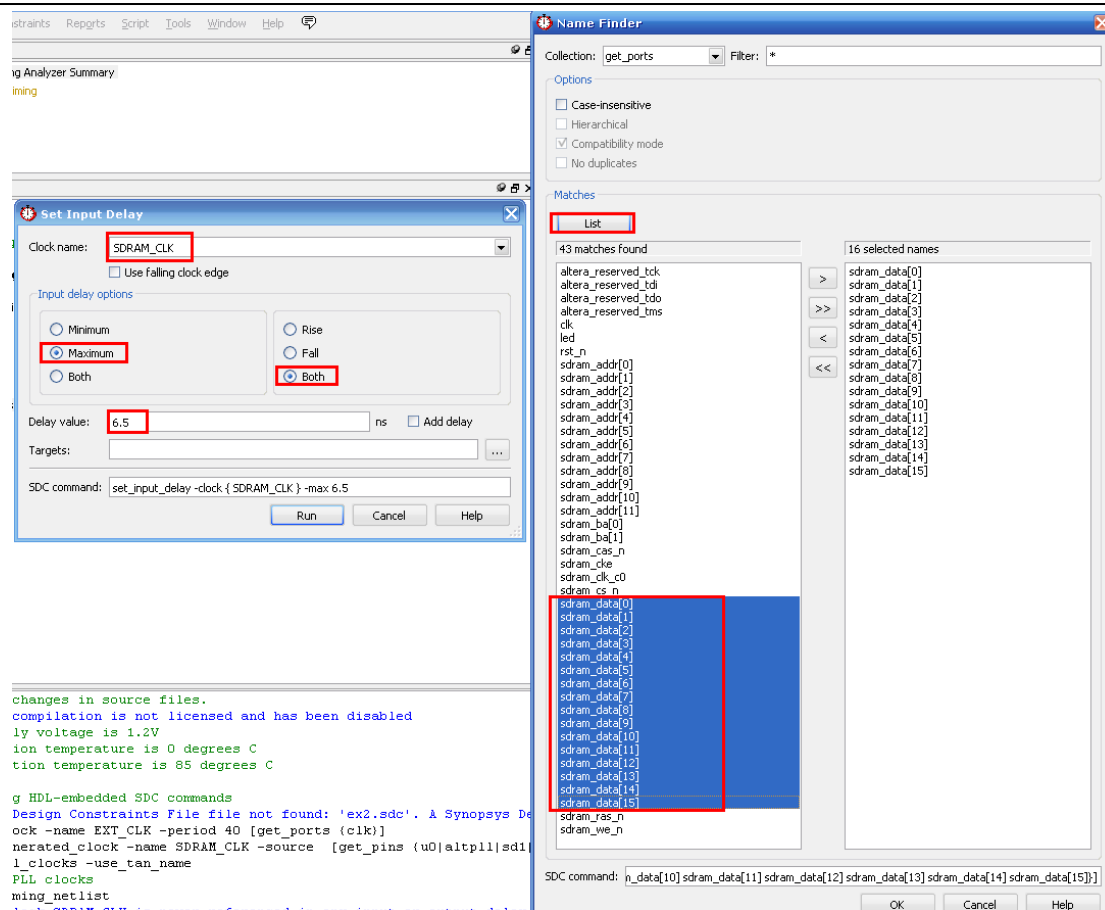
SDRAM 管脚的 input delay 和 output delay 也需要约束。点击菜单栏 Constraints→Set Output Delay。输入 Clock name 为 SDRAM_CLK, delay options 里选择 Maximum 和 Both, Delay value 为 2.6ns, 然后点击 Targets 后面的按钮, 添加所有 SDRAM 的信号。设置完点击 Run。



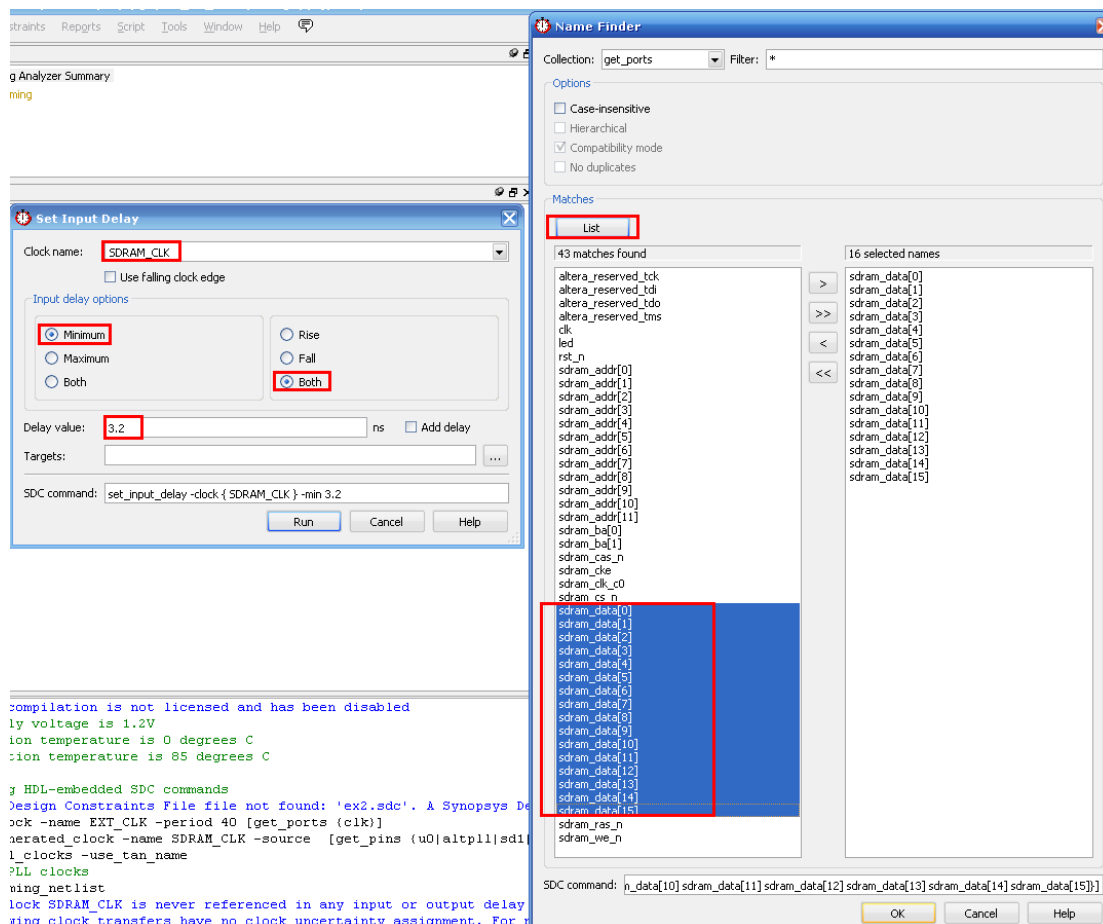
点击菜单栏 Constrains→Set Output Delay。输入 Clock name 为 SDRAM_CLK, delay options 里选择 Minimum 和 Both, Delay value 为 -1.1ns, 然后点击 Targets 后面的按钮, 添加所有 SDRAM 的信号。设置完点击 Run。



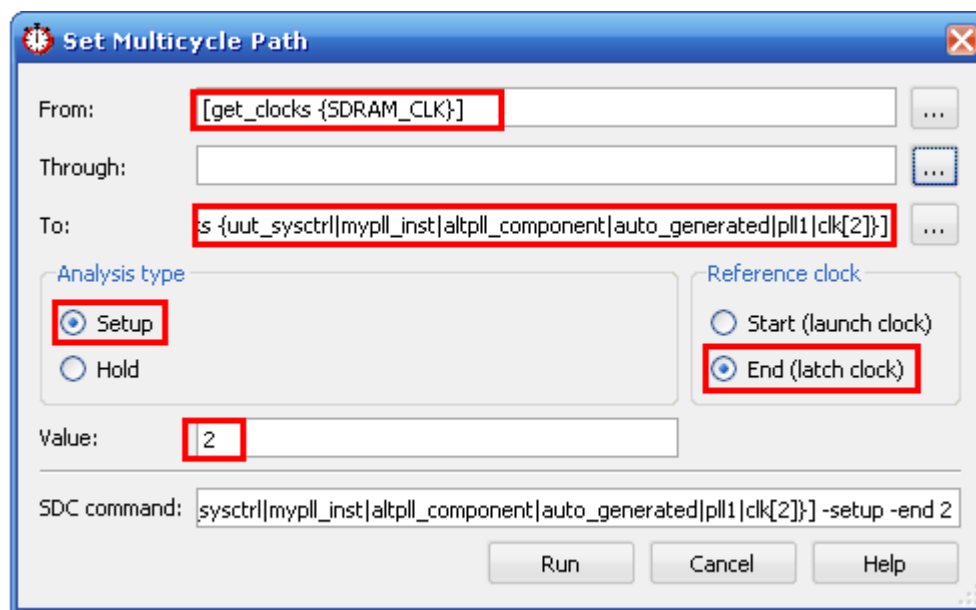
点击菜单栏 Constraints→Set Input Delay。输入 Clock name 为 SDRAM_CLK, delay options 里选择 Maximum 和 Both, Delay value 为 6.5ns, 然后点击 Targets 后面的按钮, 添加所有 SDRAM 的数据总线信号 (sdr*_data[0]到 sdr*_data[15])。设置完点击 Run。



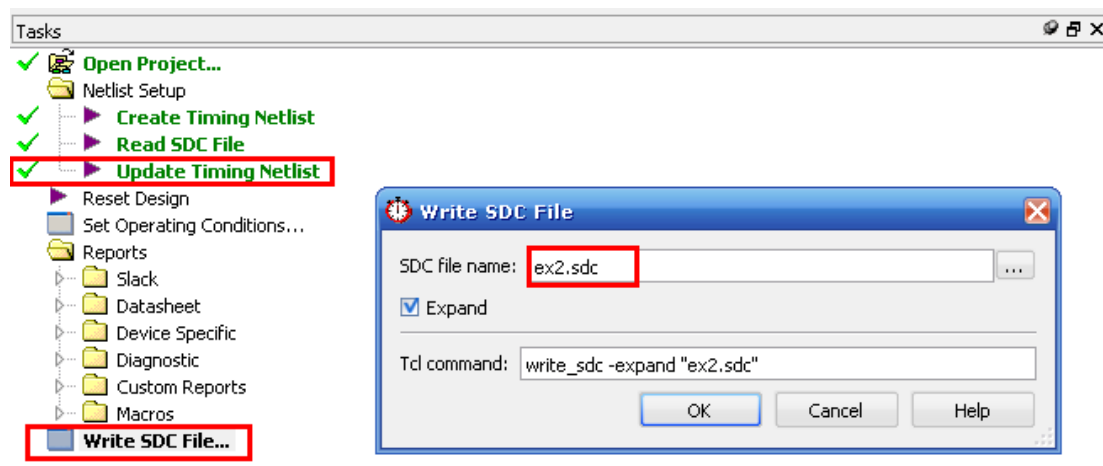
点击菜单栏 Constrains→Set Input Delay。输入 Clock name 为 SDRAM_CLK，delay options 里选择 Minimum 和 Both，Delay value 为 3.2ns，然后点击 Targets 后面的按钮，添加所有 SDRAM 的数据总线信号（sdram_data[0]到 sdram_data[15]）。设置完点击 Run。



点击菜单栏 Constrains→Set Multicycle Path，做如下选择设置。



为了将新做的约束添加到 sdc 文件中，我需要和前面同样的操作：分别点击 Tasks 面板的 Update Timing Netlist 和 Write SDC File，然后在弹出的对话框中输入 ex2.sdc 文件，点击 OK 完成 sdc 文件的创建，刚才的几个脚本也都写入 sdc 文件中。



完成这些时序约束设置后，我们重新编译工程。编译完成，将生成的.sof 文件下载到 SF-CY3 目标板中。

7.6.9 软件工程实例

参照前面的例程，打开 EDS 软件，以当前工程的硬件为基础新建一个 blank 工程命名为 mylcd_prj。打开 BSP Editor 做代码裁剪的设置，然后进入应用工程新建一个 main.c 文件，另外再新建一个 ASCII.h 文件，该文件用于存储 8*16 的 ASCII 码字模（如 6.4 节，也是使用取模软件 PCtoLCD2002 得到的字模数据）。

ASCII.h 文件代码如下。在对 ASCII.h 的 ascii[] 数组进行寻址时，对应 16 个字节为一组表示一个字模，从数值为 0x20 的 ASCII 字符“ ”（空格）开始寻址到最后一个字模“.”。具体寻址算法见 main.c 中的程序。

```
/*显示常用的 ASCII 码，像素为 8×16*/
alt_u8 ascii[96*16] = {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,    /*—    —*//0x20
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x18, 0x3C, 0x3C, 0x3C, 0x18,    /*—    !  —*//0x21
    0x18, 0x00, 0x18, 0x18, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x66, 0x66, 0x66, 0x00, 0x00,    /*—    "  —*//0x22
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x36, 0x36, 0x7F, 0x36, 0x36,    /*—    #  —*/
    0x36, 0x7F, 0x36, 0x36, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x18, 0x18, 0x3C, 0x66, 0x60, 0x30, 0x18,    /*—    $  —*/
```



```
0x0C, 0x06, 0x66, 0x3C, 0x18, 0x18, 0x00, 0x00,
0x00, 0x00, 0x70, 0xD8, 0xDA, 0x76, 0x0C, 0x18,    /*---  %  ---*/
0x30, 0x6E, 0x5B, 0x1B, 0x0E, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x38, 0x6C, 0x6C, 0x38, 0x60,    /*---  &  ---*/
0x6F, 0x66, 0x66, 0x3B, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x18, 0x18, 0x18, 0x00, 0x00,    /*---  '  ---*/
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x0C, 0x18, 0x18, 0x30, 0x30,    /*---  (  ---*/
0x30, 0x30, 0x30, 0x18, 0x18, 0x0C, 0x00, 0x00,
0x00, 0x00, 0x00, 0x30, 0x18, 0x18, 0x0C, 0x0C,    /*---  )  ---*/
0x0C, 0x0C, 0x0C, 0x18, 0x18, 0x30, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x36, 0x1C, 0x7F,    /*---  *  ---*/
0x1C, 0x36, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x18, 0x18, 0x7E,    /*---  +  ---*/
0x18, 0x18, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,    /*---  ,  ---*/
0x00, 0x00, 0x1C, 0x1C, 0x0C, 0x18, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x7E,    /*---  -  ---*/
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,    /*---  .  ---*/
0x00, 0x00, 0x1C, 0x1C, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x06, 0x06, 0x0C, 0x0C, 0x18,    /*---  /  ---*/
0x18, 0x30, 0x30, 0x60, 0x60, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x1E, 0x33, 0x37, 0x37, 0x33,    /*---  0  ---*/
0x3B, 0x3B, 0x33, 0x1E, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x0C, 0x1C, 0x7C, 0x0C, 0x0C,    /*---  1  ---*/
0x0C, 0x0C, 0x0C, 0x0C, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x3C, 0x66, 0x66, 0x06, 0x0C,    /*---  2  ---*/
0x18, 0x30, 0x60, 0x7E, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x3C, 0x66, 0x66, 0x06, 0x1C,    /*---  3  ---*/
0x06, 0x66, 0x66, 0x3C, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x30, 0x30, 0x36, 0x36, 0x36,    /*---  4  ---*/
0x66, 0x7F, 0x06, 0x06, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x7E, 0x60, 0x60, 0x60, 0x7C,    /*---  5  ---*/
0x06, 0x06, 0x0C, 0x78, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x1C, 0x18, 0x30, 0x7C, 0x66,    /*---  6  ---*/
0x66, 0x66, 0x66, 0x3C, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x7E, 0x06, 0x0C, 0x0C, 0x18,    /*---  7  ---*/
```



```
0x18, 0x30, 0x30, 0x30, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x3C, 0x66, 0x66, 0x76, 0x3C,    /*--- 8 ---*/
0x6E, 0x66, 0x66, 0x3C, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x3C, 0x66, 0x66, 0x66, 0x66,    /*--- 9 ---*/
0x3E, 0x0C, 0x18, 0x38, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x1C, 0x1C, 0x00,    /*--- : ---*/
0x00, 0x00, 0x1C, 0x1C, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x1C, 0x1C, 0x00,    /*--- ; ---*/
0x00, 0x00, 0x1C, 0x1C, 0x0C, 0x18, 0x00, 0x00,
0x00, 0x00, 0x00, 0x06, 0x0C, 0x18, 0x30, 0x60,    /*--- < ---*/
0x30, 0x18, 0x0C, 0x06, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x7E, 0x00,    /*--- = ---*/
0x7E, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x60, 0x30, 0x18, 0x0C, 0x06,    /*--- > ---*/
0x0C, 0x18, 0x30, 0x60, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x3C, 0x66, 0x66, 0x0C, 0x18,    /*--- ? ---*/
0x18, 0x00, 0x18, 0x18, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x7E, 0xC3, 0xC3, 0xCF, 0xDB,    /*--- @ ---*/
0xDB, 0xCF, 0xC0, 0x7F, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x18, 0x3C, 0x66, 0x66, 0x66,    /*--- A ---*/
0x7E, 0x66, 0x66, 0x66, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x7C, 0x66, 0x66, 0x66, 0x7C,    /*--- B ---*/
0x66, 0x66, 0x66, 0x7C, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x3C, 0x66, 0x66, 0x60, 0x60,    /*--- C ---*/
0x60, 0x66, 0x66, 0x3C, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x78, 0x6C, 0x66, 0x66, 0x66,    /*--- D ---*/
0x66, 0x66, 0x6C, 0x78, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x7E, 0x60, 0x60, 0x60, 0x7C,    /*--- E ---*/
0x60, 0x60, 0x60, 0x7E, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x7E, 0x60, 0x60, 0x60, 0x7C,    /*--- F ---*/
0x60, 0x60, 0x60, 0x60, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x3C, 0x66, 0x66, 0x60, 0x60,    /*--- G ---*/
0x6E, 0x66, 0x66, 0x3E, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x66, 0x66, 0x66, 0x66, 0x7E,    /*--- H ---*/
0x66, 0x66, 0x66, 0x66, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x3C, 0x18, 0x18, 0x18, 0x18,    /*--- I ---*/
0x18, 0x18, 0x18, 0x3C, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x06, 0x06, 0x06, 0x06, 0x06,    /*--- J ---*/
```



```
0x06, 0x66, 0x66, 0x3C, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x66, 0x66, 0x6C, 0x6C, 0x78,    /*--- K ---*/
0x6C, 0x6C, 0x66, 0x66, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x60, 0x60, 0x60, 0x60, 0x60,    /*--- L ---*/
0x60, 0x60, 0x60, 0x7E, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x63, 0x63, 0x77, 0x6B, 0x6B,    /*--- M ---*/
0x6B, 0x63, 0x63, 0x63, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x63, 0x63, 0x73, 0x7B, 0x6F,    /*--- N ---*/
0x67, 0x63, 0x63, 0x63, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x3C, 0x66, 0x66, 0x66, 0x66,    /*--- O ---*/
0x66, 0x66, 0x66, 0x3C, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x7C, 0x66, 0x66, 0x66, 0x7C,    /*--- P ---*/
0x60, 0x60, 0x60, 0x60, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x3C, 0x66, 0x66, 0x66, 0x66,    /*--- Q ---*/
0x66, 0x66, 0x66, 0x3C, 0x0C, 0x06, 0x00, 0x00,
0x00, 0x00, 0x00, 0x7C, 0x66, 0x66, 0x66, 0x7C,    /*--- R ---*/
0x6C, 0x66, 0x66, 0x66, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x3C, 0x66, 0x60, 0x30, 0x18,    /*--- S ---*/
0x0C, 0x06, 0x66, 0x3C, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x7E, 0x18, 0x18, 0x18, 0x18,    /*--- T ---*/
0x18, 0x18, 0x18, 0x18, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x66, 0x66, 0x66, 0x66, 0x66,    /*--- U ---*/
0x66, 0x66, 0x66, 0x3C, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x66, 0x66, 0x66, 0x66, 0x66,    /*--- V ---*/
0x66, 0x66, 0x3C, 0x18, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x63, 0x63, 0x63, 0x6B, 0x6B,    /*--- W ---*/
0x6B, 0x36, 0x36, 0x36, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x66, 0x66, 0x34, 0x18, 0x18,    /*--- X ---*/
0x2C, 0x66, 0x66, 0x66, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x66, 0x66, 0x66, 0x66, 0x3C,    /*--- Y ---*/
0x18, 0x18, 0x18, 0x18, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x7E, 0x06, 0x06, 0x0C, 0x18,    /*--- Z ---*/
0x30, 0x60, 0x60, 0x7E, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x3C, 0x30, 0x30, 0x30, 0x30,    /*--- [ ---*/
0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x3C, 0x3C,
0x00, 0x00, 0x00, 0x60, 0x60, 0x30, 0x30, 0x18,    /*--- \ ---*/
0x18, 0x0C, 0x0C, 0x06, 0x06, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x3C, 0x0C, 0x0C, 0x0C, 0x0C,    /*--- ] ---*/
```




```
0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x3C, 0x3C,
0x00, 0x18, 0x3C, 0x66, 0x00, 0x00, 0x00, 0x00, /*--- ^ ---*/
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /*--- _ ---*/
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF,
0x00, 0x38, 0x18, 0x0C, 0x00, 0x00, 0x00, 0x00, /*--- ` ---*/
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x3C, 0x06, 0x06, /*--- a ---*/
0x3E, 0x66, 0x66, 0x3E, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x60, 0x60, 0x7C, 0x66, 0x66, /*--- b ---*/
0x66, 0x66, 0x66, 0x7C, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x3C, 0x66, 0x60, /*--- c ---*/
0x60, 0x60, 0x66, 0x3C, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x06, 0x06, 0x3E, 0x66, 0x66, /*--- d ---*/
0x66, 0x66, 0x66, 0x3E, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x3C, 0x66, 0x66, /*--- e ---*/
0x7E, 0x60, 0x60, 0x3C, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x1E, 0x30, 0x30, 0x30, 0x7E, /*--- f ---*/
0x30, 0x30, 0x30, 0x30, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x3E, 0x66, 0x66, /*--- g ---*/
0x66, 0x66, 0x66, 0x3E, 0x06, 0x06, 0x7C, 0x7C,
0x00, 0x00, 0x00, 0x60, 0x60, 0x7C, 0x66, 0x66, /*--- h ---*/
0x66, 0x66, 0x66, 0x66, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x18, 0x18, 0x00, 0x78, 0x18, 0x18, /*--- i ---*/
0x18, 0x18, 0x18, 0x7E, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x0C, 0x0C, 0x00, 0x3C, 0x0C, 0x0C, /*--- j ---*/
0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x78, 0x78,
0x00, 0x00, 0x00, 0x60, 0x60, 0x66, 0x66, 0x6C, /*--- k ---*/
0x78, 0x6C, 0x66, 0x66, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x78, 0x18, 0x18, 0x18, 0x18, /*--- l ---*/
0x18, 0x18, 0x18, 0x7E, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x7E, 0x6B, 0x6B, /*--- m ---*/
0x6B, 0x6B, 0x6B, 0x63, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x7C, 0x66, 0x66, /*--- n ---*/
0x66, 0x66, 0x66, 0x66, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x3C, 0x66, 0x66, /*--- o ---*/
0x66, 0x66, 0x66, 0x3C, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x7C, 0x66, 0x66, /*--- p ---*/
```



```
0x66, 0x66, 0x66, 0x7C, 0x60, 0x60, 0x60, 0x60,
0x00, 0x00, 0x00, 0x00, 0x00, 0x3E, 0x66, 0x66,    /*---  q  ---*/
0x66, 0x66, 0x66, 0x3E, 0x06, 0x06, 0x06, 0x06,
0x00, 0x00, 0x00, 0x00, 0x00, 0x66, 0x6E, 0x70,    /*---  r  ---*/
0x60, 0x60, 0x60, 0x60, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x3E, 0x60, 0x60,    /*---  s  ---*/
0x3C, 0x06, 0x06, 0x7C, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x30, 0x30, 0x7E, 0x30, 0x30,    /*---  t  ---*/
0x30, 0x30, 0x30, 0x1E, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x66, 0x66, 0x66,    /*---  u  ---*/
0x66, 0x66, 0x66, 0x3E, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x66, 0x66, 0x66,    /*---  v  ---*/
0x66, 0x66, 0x3C, 0x18, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x63, 0x6B, 0x6B,    /*---  w  ---*/
0x6B, 0x6B, 0x36, 0x36, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x66, 0x66, 0x3C,    /*---  x  ---*/
0x18, 0x3C, 0x66, 0x66, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x66, 0x66, 0x66,    /*---  y  ---*/
0x66, 0x66, 0x66, 0x3C, 0x0C, 0x18, 0xF0, 0xF0,
0x00, 0x00, 0x00, 0x00, 0x00, 0x7E, 0x06, 0x0C,    /*---  z  ---*/
0x18, 0x30, 0x60, 0x7E, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x0C, 0x18, 0x18, 0x18, 0x30,    /*---  {  ---*/
0x60, 0x30, 0x18, 0x18, 0x18, 0x0C, 0x00, 0x00,
0x00, 0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18,    /*---  |  ---*/
0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18,
0x00, 0x00, 0x00, 0x30, 0x18, 0x18, 0x18, 0x0C,    /*---  }  ---*/
0x06, 0x0C, 0x18, 0x18, 0x18, 0x30, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x71, 0xDB,    /*---  ~  ---*/
0x8E, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,    /*---  .  ---*/
0x00, 0x00, 0x1C, 0x1C, 0x00, 0x00, 0x00, 0x00
};
```

main.c 文件如下。print_ascii 函数实现的功能主要是将一个字符串显示到 LCD 的某个固定坐标点上, 显示前景和背景色也都是可以设定的。lcd_wrdb 函数则是最基本的对 lcd 组件某个固定坐标点地址写数据操作。main 函数中首先将 LCD 全屏清为蓝色, 然后显示一个字符串在显示屏中央。

```
/*
```



```
* main.c
*
* Created on: 2013-2-18
* Author: Administrator
*/
#include "alt_types.h"
#include "io.h"
#include "sys/alt_irq.h"
#include "system.h"
#include <stdio.h>
#include <unistd.h>
#include "ASCII.h"

extern alt_u8 ascii[96*16]; //8*16 的 ASCII 字库
void lcd_wrdb(alt_u16 xaddr,alt_u16 yaddr,alt_u16 cor); //LCD 写数据函数
void print_ascii(alt_u16 uRow,alt_u16 uCol,alt_u8 *ptr,alt_u16 Cor_b0,alt_u16
Cor_q0); //在指定位置显示 8*16 的 ASCII 码字符串

////////////////////////////////////
//函数名: main
//功 能: 主函数
//输 入: 无
//返 回: int
////////////////////////////////////
int main()
{
    alt_u16 x,y;

    //清全屏为蓝色
    for(y=0;y<240;y++)
    {
        for(x=0;x<320;x++)
        {
            lcd_wrdb(x,y,0x001f);
        }
    }

    print_ascii(60,111,"http://myfpga.taobao.com/",0x001f,0xf800);
```



```
while(1);

return 0;
}

////////////////////////////////////
//函数名: lcd_wrdb
//功    能: LCD 写数据函数
//输    入: alt_u16 xaddr--X 坐标地址; alt_u16 yaddr--Y 坐标地址; alt_u16 cor--
显示色彩
//返    回: 无
////////////////////////////////////
void lcd_wrdb(alt_u16 xaddr,alt_u16 yaddr,alt_u16 cor)
{

    IOWR_16DIRECT(MYLCD_BASE, ((yaddr<<10)+(xaddr<<1)),cor); //LCD 显示内存映射
地址写入色彩数据
}

/*****
函数名: print_ascii
功 能: 在指定位置显示 8*16 的 ASCII 码字符串
输 入: alt_u16 uRow:      字符显示起始 x 坐标
        alt_u16 uCol:      字符显示起始 y 坐标
        alt_u8 *ptr:      写入的 ASCII 码字符串
        alt_u16 Cor_b0: 前景色彩
        alt_u16 Cor_q0: 背景色彩
返 回: 无
*****/
void print_ascii(alt_u16 uRow,alt_u16 uCol,alt_u8 *ptr,alt_u16 Cor_b0,alt_u16
Cor_q0)
{
    alt_u16 uLen=0; //字符串长度
    alt_u16 i,j,k=0;

    while ((alt_u8)ptr[uLen] >= 0x10)    {uLen++;};        //探测字符串长度
```



```
while(k < uLen) //写入 uLen 个 ASCII 字符
{
    if(ptr[k] <= 128)          //ASCII 码
    {
        if(ptr[k] >= 0x10)
        {
            for(j=0;j<16;j++)    //显示 8×16 的 ASCII 码, 分 16 行输出
            {
                for(i=0;i<8;i++)
                {
                    if((ascii[(ptr[k]-0x20)*16+j] & (1<<(7-i))) != 0x00)
lcd_wrdp(uRow+i,uCol+j,Cor_q0); //送像素点前景色
                    else lcd_wrdp(uRow+i,uCol+j,Cor_b0);    //送像素点背景
色
                }
            }
        }
        uRow+=8;                // x 坐标加 8, 即下一个字符位
    }
    k++;    //下一个字符
}
}
```

编译软件工程, 然后 Run 到目标硬件中。如图所示, 我们可以看到此时 LCD 中央出现了一个字符串。



8 SF-SENSOR 子板开发指南

8.1 功能与原理图介绍

8.1.1 主要外设芯片及电路图解析

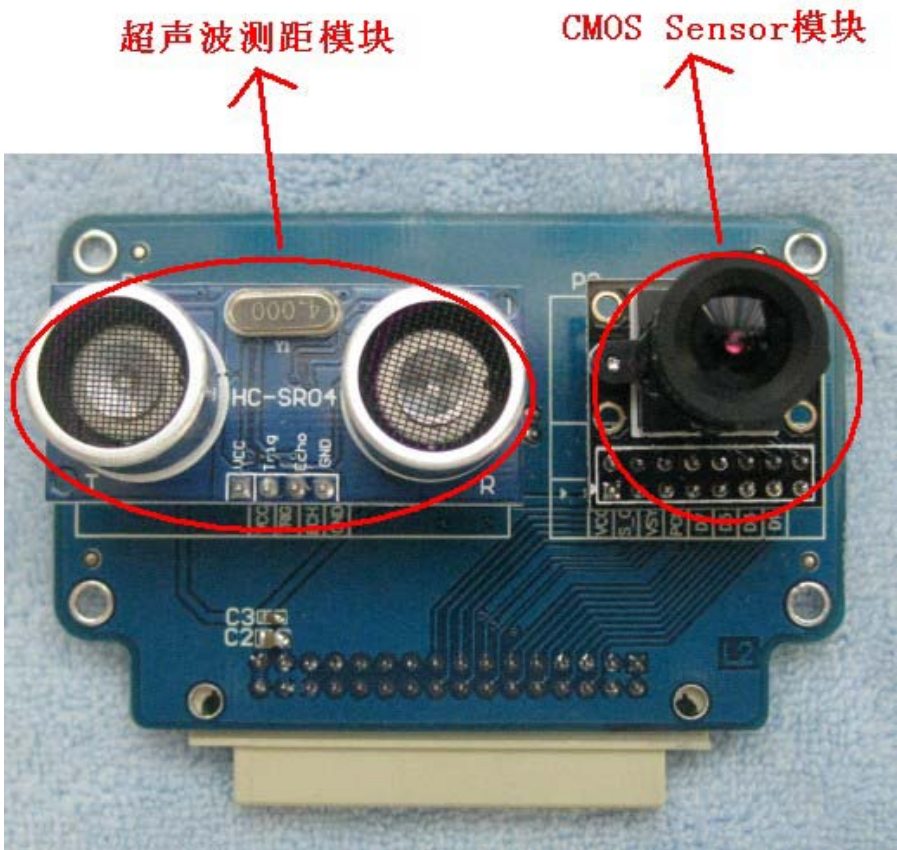
SF-SENSOR 子板主要外设芯片及其功能描述见下表。

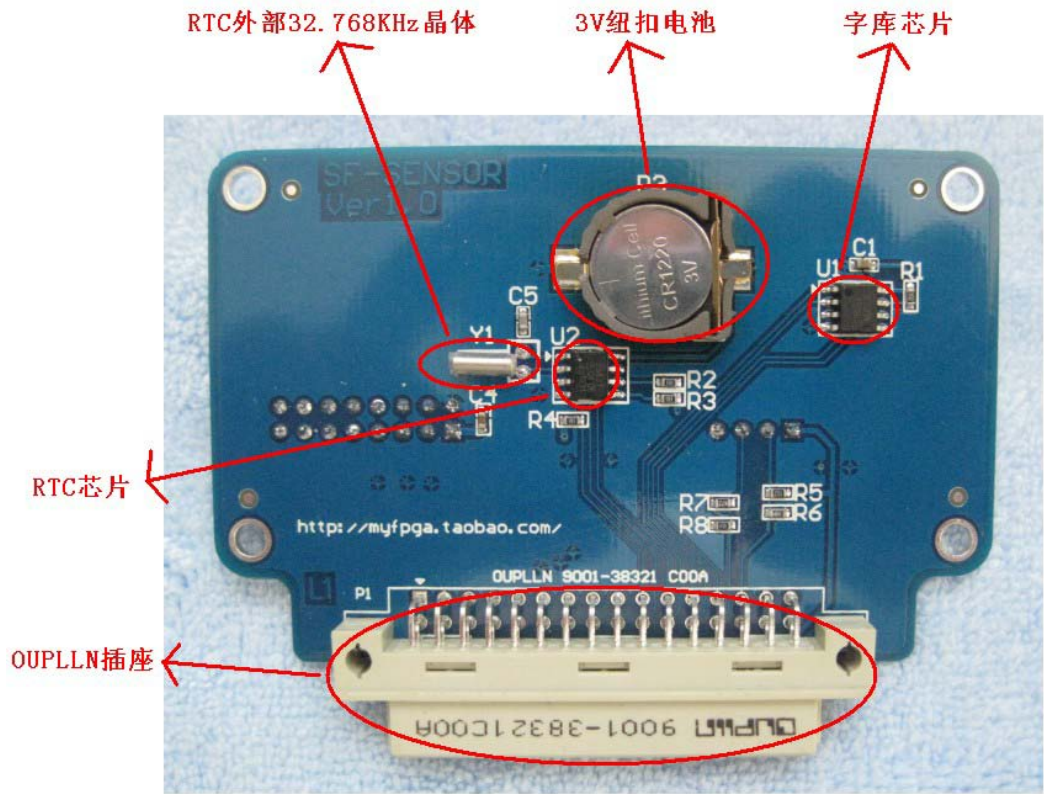
外设芯片	主要功能
32PIN 的 OUPLLN 插座 P1	用于 SF-LCD 子板上的各个外设与 SF-CY3 核心板相连。
16PIN 的插座 P2	用于连接 CMOS Sensor 视频模块。
4PIN 的插座 P4	用于连接超声波模块。
2PIN 的插座 P3	焊接 RTC 电池座，安装纽扣电池后，用于给 RTC 芯片供电。



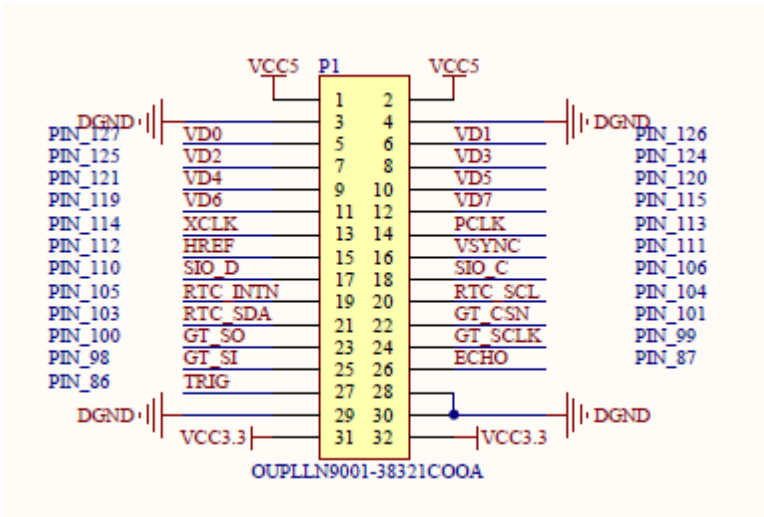
芯片 PCF8563T (U2)	NXP 的实时时钟 (RTC) 芯片。
芯片 GT21L16S2W (U1)	集通公司的中文字库芯片。

各个主要外设芯片的实物位置如图所示。

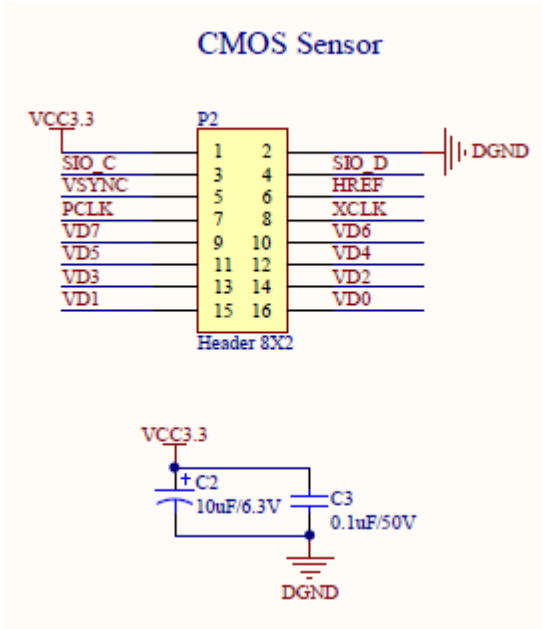




和 SF-CY3 核心模块互联的是 P1 插座，该插座连接到 SF-CY3 的 P2 插座上。P1 插座上的蓝色字体定义了各个信号对应连接到 SF-CY3 核心板上的管脚，方便大家在做管脚分配时参考。



CMOS Sensor 是一个视频图像采集模块，该模块预留了 16PIN 的插针接口，对应的插入到我们 SF-SENSOR 板的 P2 插座上，在实际连接时，请大家参考后续的装配章节，切勿误差。

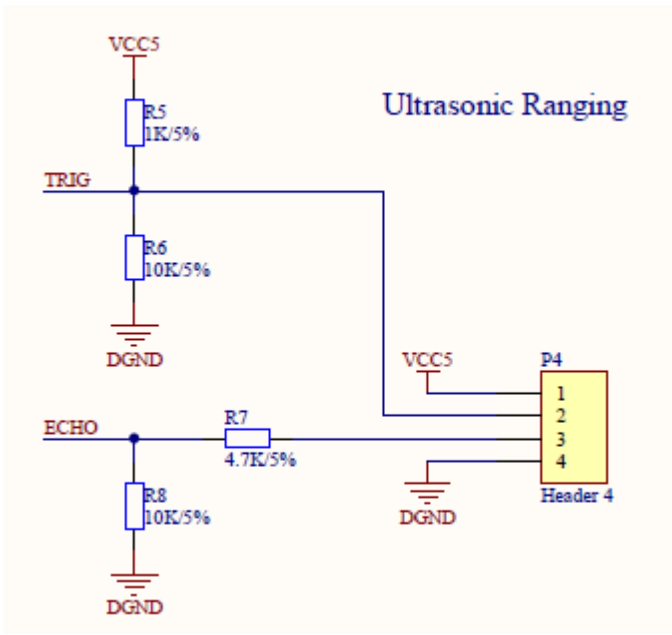


P2 插座上各个管脚的定义如下。

管脚	信号	功能
1	VCC3.3	电源电压 3.3V。
2	DGND	电源地。
3	SIO_C	用于配置 Sensor 芯片内部寄存器的 IIC 接口时钟信号。
4	SIO_D	用于配置 Sensor 芯片内部寄存器的 IIC 接口数据信号。
5	VSYNC	伴随视频数据输出的场同步信号。
6	HREF	伴随视频数据输出的行同步信号。
7	PCLK	用于视频数据总线同步的时钟信号。
8	XCLK	外部（FPGA）输给 Sensor 芯片工作的时钟信号。
9	VD7	Sensor 输出的视频数据总线。
10	VD6	Sensor 输出的视频数据总线。
11	VD5	Sensor 输出的视频数据总线。
12	VD4	Sensor 输出的视频数据总线。
13	VD3	Sensor 输出的视频数据总线。
14	VD2	Sensor 输出的视频数据总线。
15	VD1	Sensor 输出的视频数据总线。
16	VD0	Sensor 输出的视频数据总线。



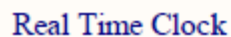
P4 插座将用于超声波测距模块的插入。



P4 的 4 个管脚定义如下所示。

管脚	信号	功能
1	VCC5	电源电压 5V。
2	TRIG	超声波模块的触发输入信号。
3	ECHO	超声波模块的响应输出信号。
4	DGND	电源地。

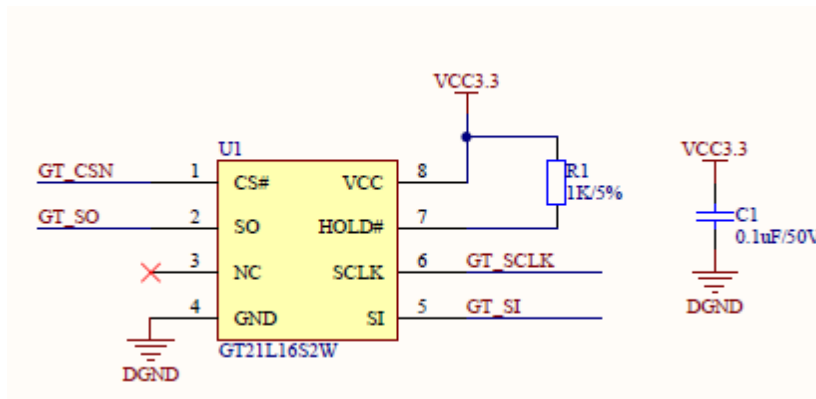
U2 是一颗实时时钟（RTC）芯片，该芯片需要外部供 32.768KHz 的时钟晶体，这个晶体的振荡需要借组一颗 30pF 的电容。该芯片和 FPGA 之间通过 IIC 总线接口进行数据传输。该芯片的供电要借组外部连接的一颗纽扣电池。



U2 的各个管脚定义如下。

管脚	信号	功能
1	OSCI	32.768KHz 晶体输入管脚。
2	OSCO	32.768KHz 晶体输出管脚。
3	INT#	RTC 芯片中断产生管脚。
4	DGND	电源地。
5	SDA	用于读写寄存器的 IIC 总线数据信号。
6	SCL	用于读写寄存器的 IIC 总线时钟信号。
7	CLKOUT	时钟输出管脚，本实例不使用，直接悬空。
8	VDD	芯片电源输入，连接到 3V 的纽扣电池供电。

最后，我们再来看看字库芯片 **U1**，该芯片的控制也不难，**4** 个信号是标准的 **SPI** 接口，它连接到 **FPGA** 的 **I/O** 管脚上。**FPGA** 通过 **SPI** 接口读出所需字模数据即可。其实这颗字库芯片和一般的 **SPI** 接口存储芯片几乎没有什么两样，在控制上反而更加简单。



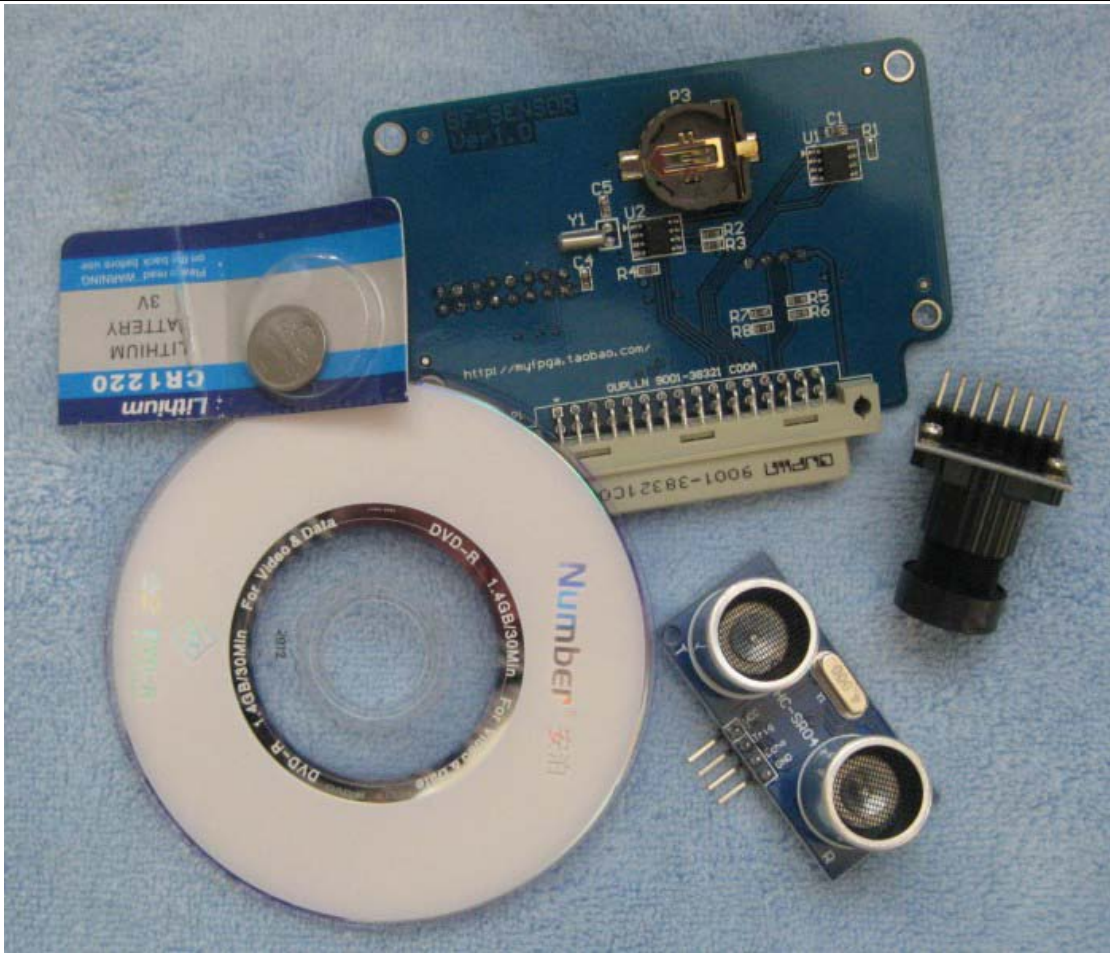


U1 的各个管脚定义如下。

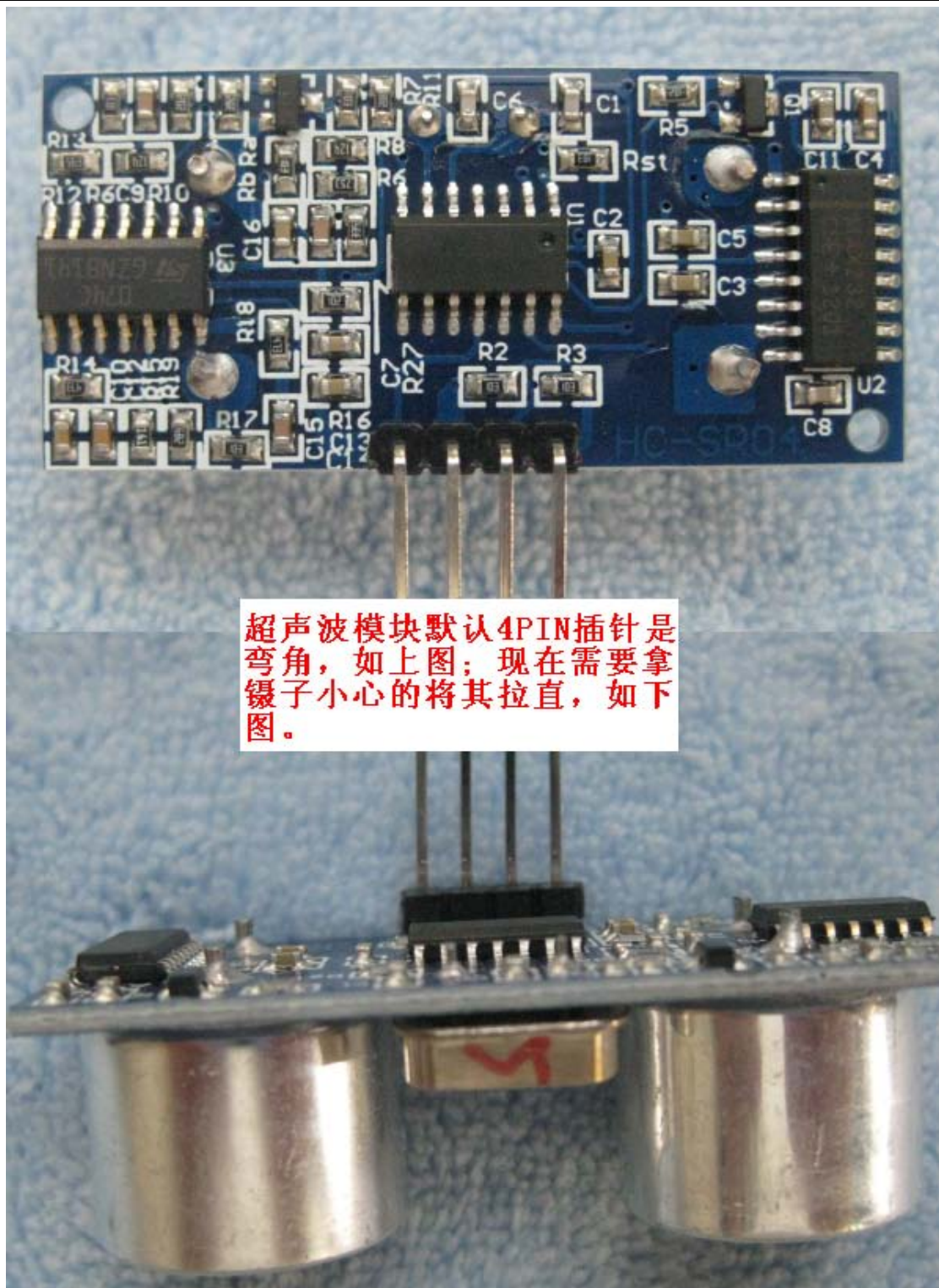
管脚	信号	功能
1	CS#	SPI 总线的片选信号，低电平有效。
2	SO	SPI 总线的 MISO 信号，即主机（FPGA）输出从机（U1 芯片）输入信号。
3	NC	无连接不使用信号。
4	GND	电源地信号。
5	SI	SPI 总线的 MOSI 信号，即主机（FPGA）输入从机（U1 芯片）输出信号。
6	SCLK	SPI 总线的时钟信号。
7	HOLD#	总线挂起信号。
8	VCC	电源信号，3.3V 供电。

8.1.2 装配示意图

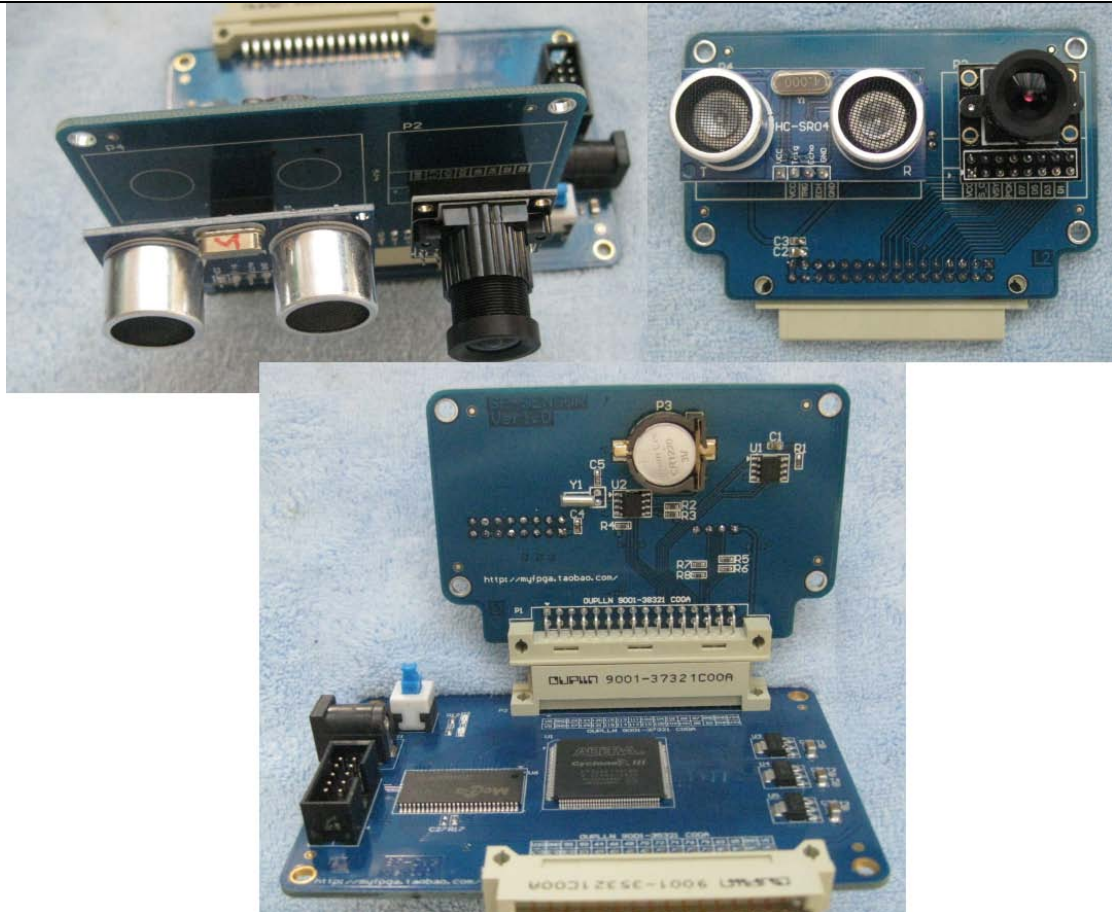
我们 SF-Sensor 模块提供了如下的 5 件宝贝。



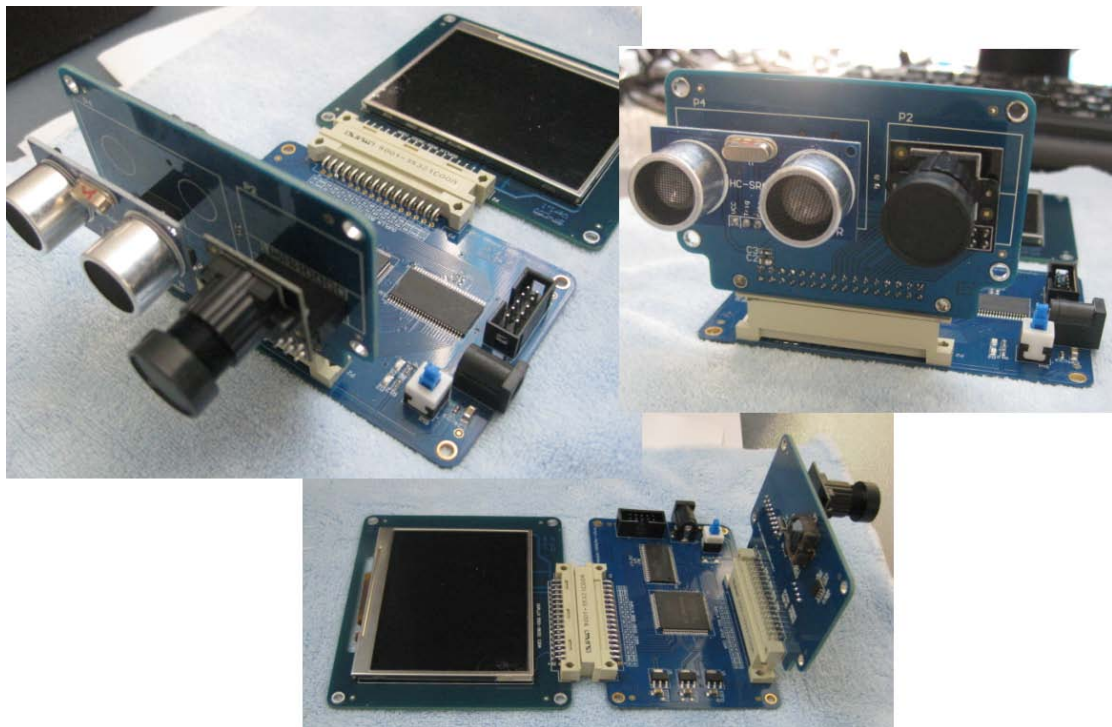
名称	数量
SF-Sensor 电路板	一块
CMOS Sensor 模块	一块
超声波测距模块	一块
3V 纽扣电池	一颗
资料光盘	一张



SF-SENSOR 板的插座连接的 SF-CY3 的 P2 插座，二者呈直角。



SF-SENSOR、SF-CY3 和 SF-LCD 三个板子互联，大有作为。





8.2 基于 Qsys 的 NIOS II 实例 8——SPI 接口字库芯片控制

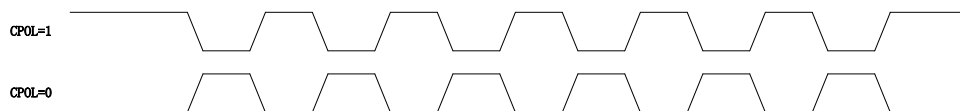
8.2.1 新 Qsys 系统——添加 SPI 组件

SPI (Serial Peripheral Interface) 即串行外围设备接口, 是一种高速、全双工、同步的通信总线。只需要四条信号线即可, 节约管脚, 同时有利于 PCB 的布局。正是出于这种简单易用的特性, 现在越来越多的芯片集成了这种通信协议。

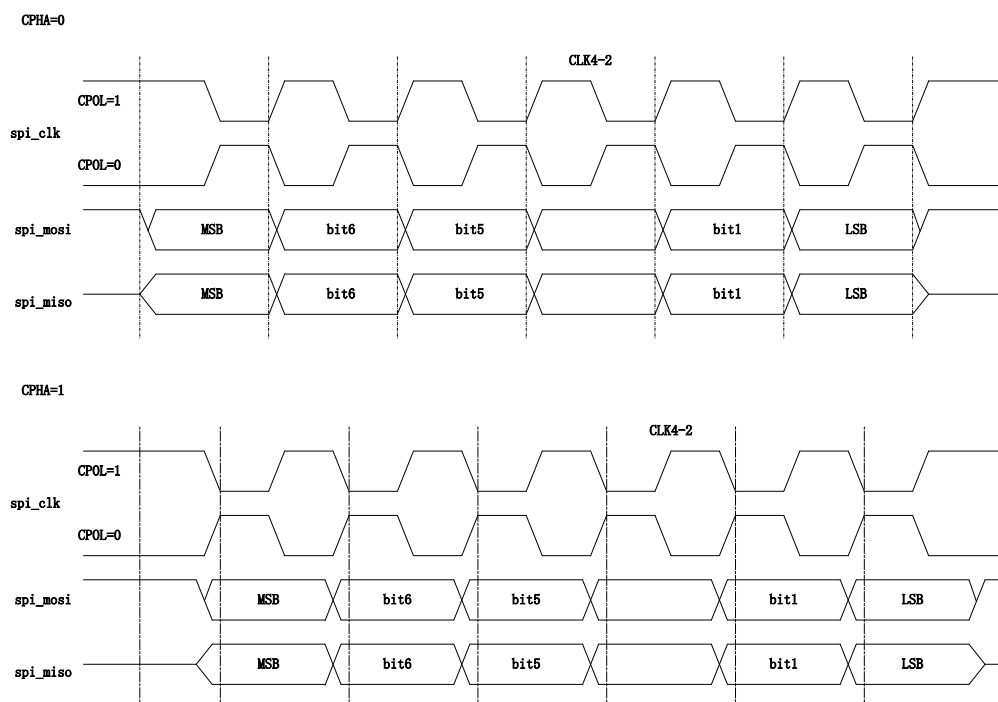
该工程模块的 SPI 接口四条信号线分别定义为 `spi_cs_n`、`spi_clk`、`spi_miso` 和 `spi_mosi`。其中 `spi_cs_n` 是控制芯片是否被选中的, 只有片选信号有效时 (一般为低电平有效), 对此芯片的操作才有效。这就使得在同一总线上连接多个 SPI 设备成为可能。`spi_clk` 是 SPI 同步时钟信号, 数据信号在该时钟的控制下逐位进行传输。`spi_miso` 和 `spi_mosi` 是主从机进行通信的数据信号, `spi_miso` 即主机的输入或者说是从机的输出, `spi_mosi` 即主机的输出或者说是从机的输入。

SPI 的工作模式有两种: 主模式和从模式。SPI 总线可以配置成单主单从、单主多从和互为主从。该工程的 FPGA 是 SPI 主机, SD 卡是从机, 处于单主单从模式。因此, FPGA 将控制产生 `spi_cs_n` 和 `spi_clk` 的时序。

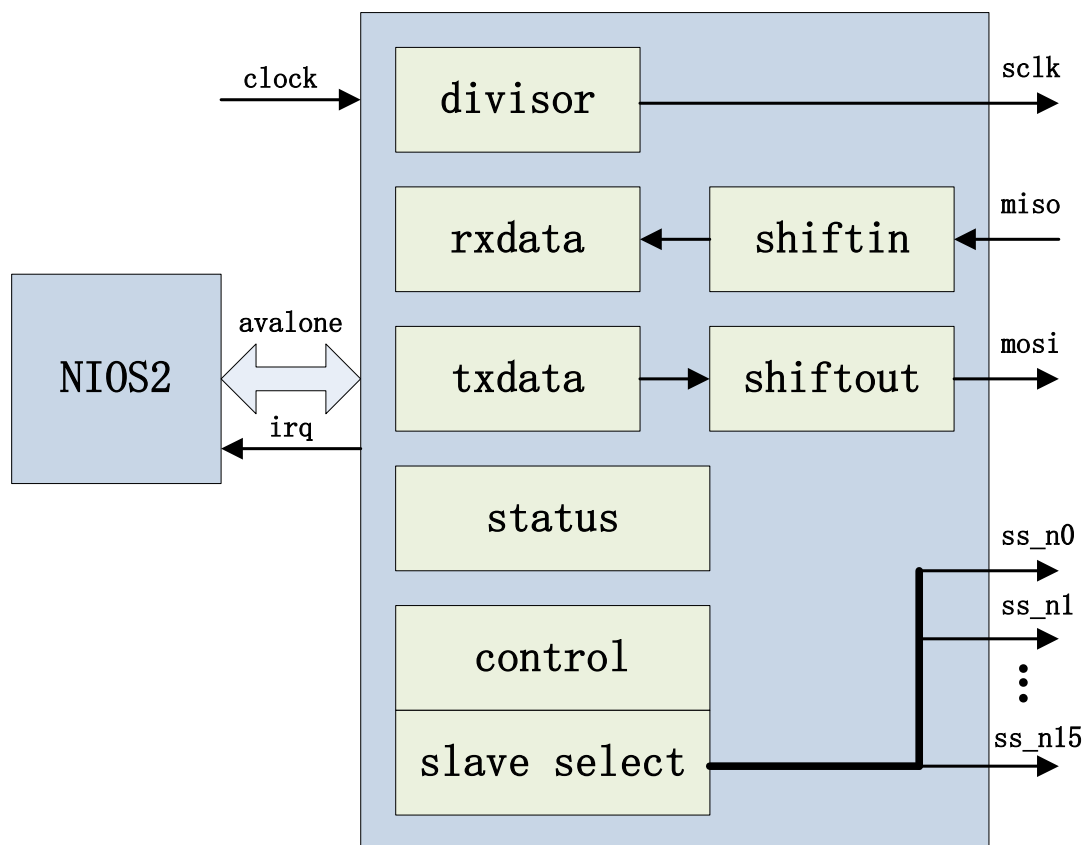
一般而言, SPI 通信可以配置成四种不同的传输模式。随便翻看一些内嵌有 SPI 接口外设的 datasheet 都会提到 CPOL 和 CPHA 这两个参数。如图所示, CPOL=1 时, SPI 时钟信号 `spi_clk` 闲置时总是高电平, 发起通信后的第一个时钟沿是下降沿; CPOL=0 时, SPI 时钟信号 `spi_clk` 闲置时总是低电平, 发起通信后的第一个时钟沿是上升沿。而 CPHA 则用于控制数据与时钟的对齐模式, CPHA=1 时, 时钟的第一个变化沿 (上升沿或者下降沿) 数据变化, 那么也意味着时钟的第二个沿 (与第一个沿相反) 锁存数据; CPHA=0 时, 时钟的第一个变化沿之前数据变化, 那么也意味着时钟的第一个沿锁存数据。



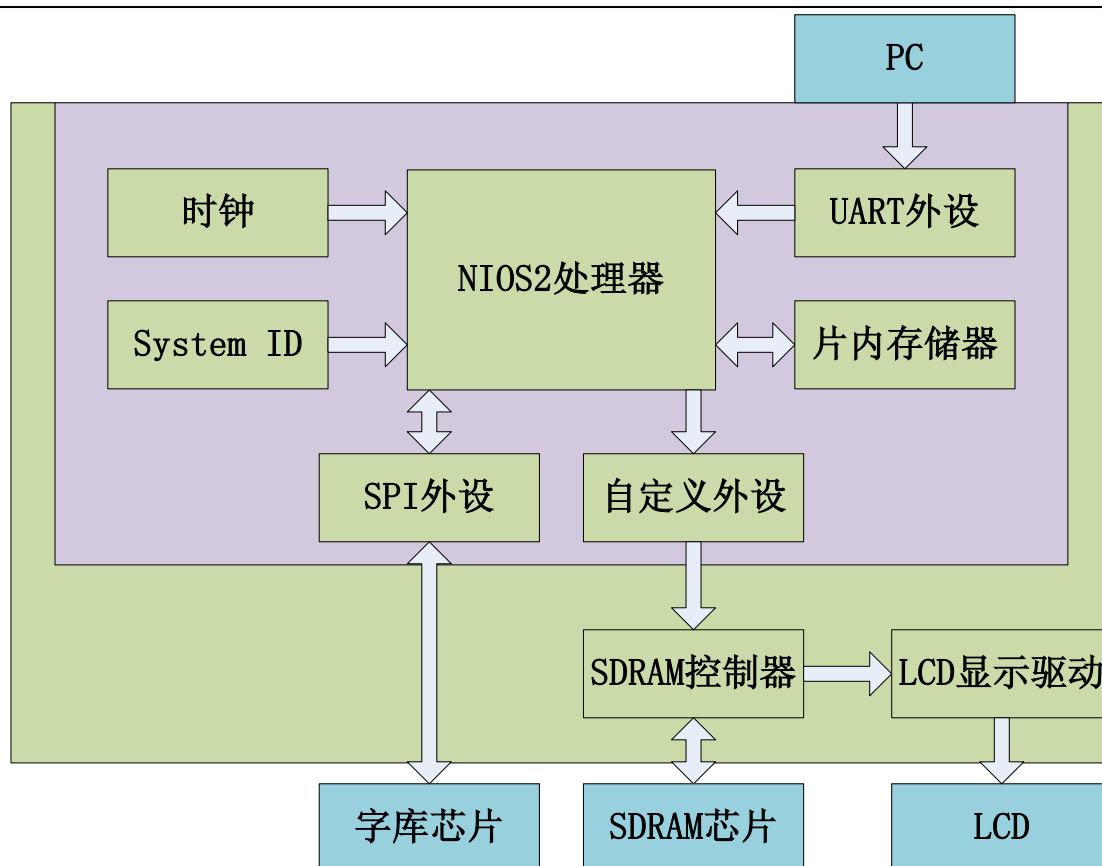
不同的 CPOL 和 CPHA 可以配置成 4 种 SPI 传输模式, 其时序如图所示。



下面我们来深入了解一下 Qsys 中集成的 SPI 组件的内部结构。SPI 组件是通过 Avalone 总线和中断请求信号 irq 与 NIOS2 处理器相连。内部也有一个 divisor 寄存器，也是对其输入工作时钟进行分频得到最终与外设接口的时钟 sclk。rxdata 和 txdata 寄存器用于 NIOS2 处理器读写收发数据，与他们直接接口的还有一个串并转换的移位寄存器 shifin 和一个并串转换的移位寄存器 shifout。NIOS2 可以从 status 寄存器读取当前 SPI 组件的状态，slave select 寄存器主要是在拥有多个从机时控制其片选。



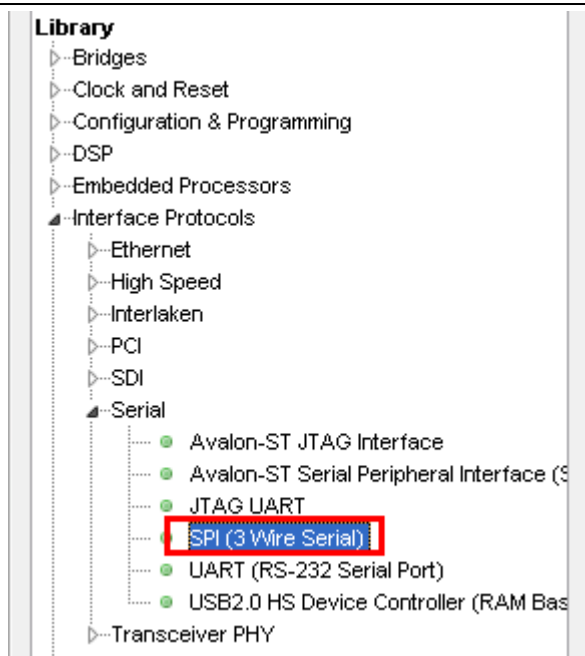
这个实例是基于上一个工程,即 SF-LCD 子板的工程 ex16 的基础上来搭建的。如图所示,该工程除了之前的一些 LCD 驱动相关控制组件和模块外,需要在 Qsys 中添加一个 SPI 外设组件,这是个工具已经集成的 IP 核,将用于连接字库芯片。



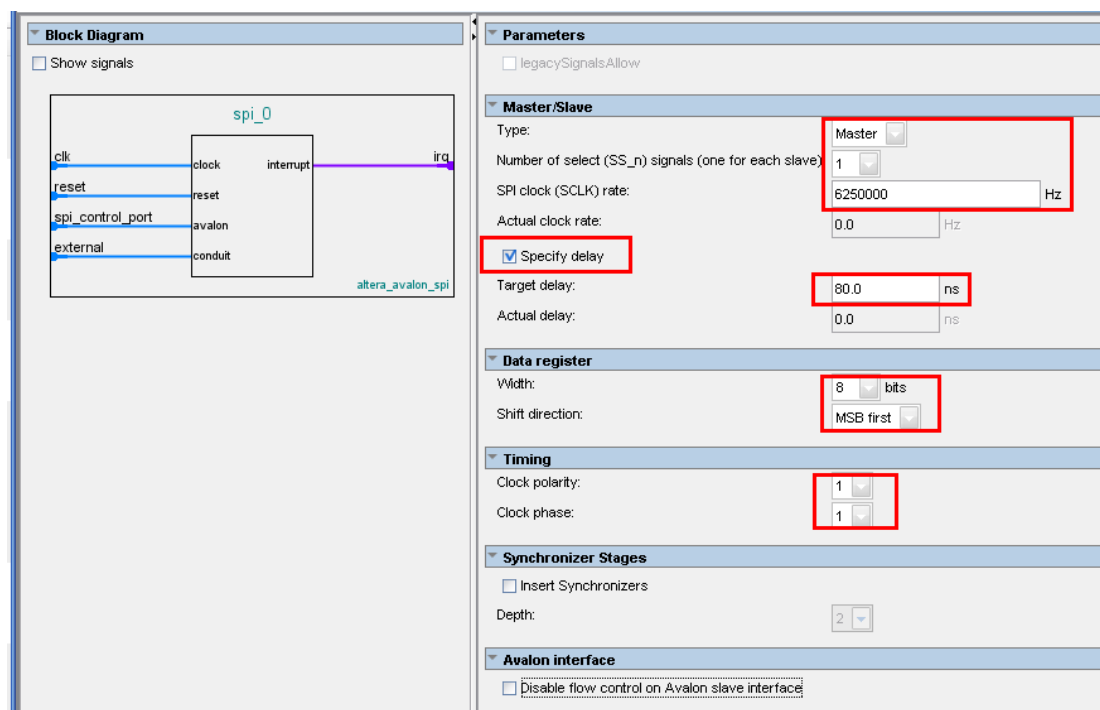
在这个实例中，我们是通过 NIOS II 处理器的 SPI 外设读取字库芯片的数据，然后送给 LCD 显示。LCD 组件的设计上一个实例（SF-LCD 模块的 ex16 工程）已经介绍了，这里不过多介绍，将会坚决执行“拿来主义”。不同的是，上一个实例 NIOS II 在软件上只是定义了一组 8*16 的 ASCII 码字库进行显示，而这个实例中，我们有一颗“神通广大”的字库芯片，什么 ABCD，什么 1234，肯定不在话下，甚至咱这“啊哦呀吧”的中文都能显示。怎么样？还算是有点技术含量吧。呵呵，下一小节我们会显示介绍 NIOS II 的软件是如何控制字库芯片的。

好了，下面我们先搭建硬件系统吧。先去拷贝一下整个 ex16 工程目录，重新命名为 ex17，接着在 Quartus II 中打开这个工程，进入 Qsys 界面。

建议先将一些不用的组件删除，如 myseg7、sw_pio、pio、myadc、mydac 组件。接着我们双击 Component Library 的 SPI(3 Wire Serial)选项，打开 SPI 外设的配置页面。



在 SPI 外设的配置页面中，我们需要设置这个 SPI 外设为主机（Master），1 个 SPI 外设从机，通信时钟频率为 6.25MHz（6250000Hz）；勾选 Specify Delay，取值 80.0ns；数据位宽为 8bits，MSB 即高位在前；选择 SPI 的 Clock polarity 和 Clock phase 均为 1。

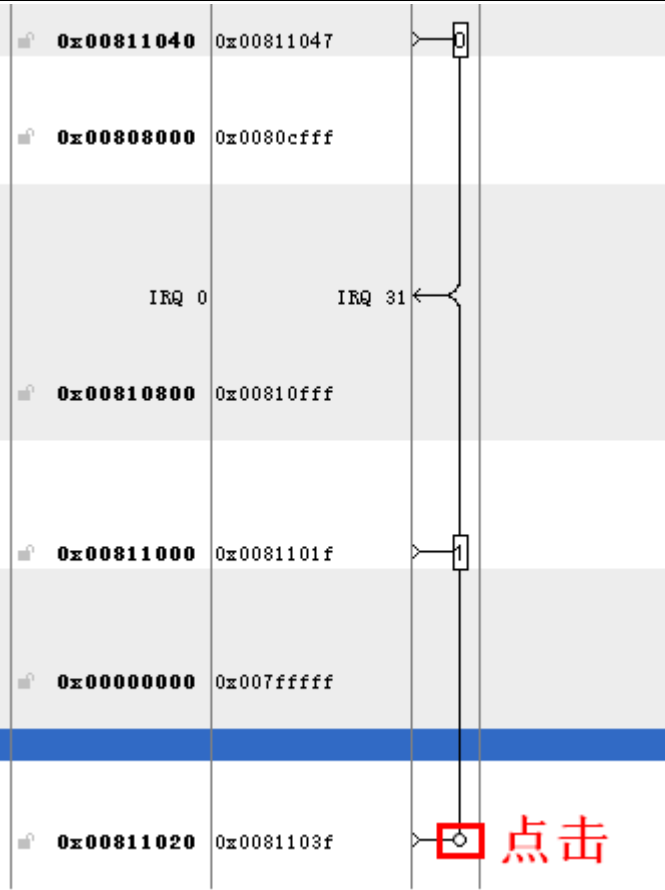


完成配置后，修改该组件的名称为 mysapi，将它的 clk 连接到系统 clk 上，它的 reset 连接到系统 clk_reset 上，它的 spi_control_port 连接到 NIOS II 处理器(nios2_qsys)的 data_master 上，external 则连接到系统外部的最终连接字库芯片的 SPI 管脚上。

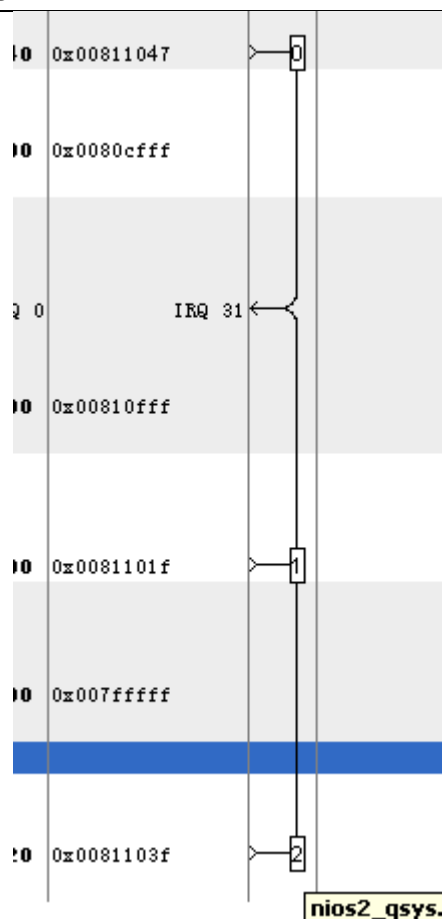


Use	Connections	Name	Description	Export
<input checked="" type="checkbox"/>		clk	Clock Source	clk
		clk_in	Clock Input	clk_0_clk_in_reset
		clk_in_reset	Reset Input	<i>Click to export</i>
		clk	Clock Output	<i>Click to export</i>
		clk_reset	Reset Output	<i>Click to export</i>
<input checked="" type="checkbox"/>		sysid_qsys	System ID Peripheral	
		clk	Clock Input	<i>Click to export</i>
		reset	Reset Input	<i>Click to export</i>
		control_slave	Avalon Memory Mapped Slave	<i>Click to export</i>
<input checked="" type="checkbox"/>		jtag_uart	JTAG UART	
		clk	Clock Input	<i>Click to export</i>
		reset	Reset Input	<i>Click to export</i>
	avalon_jtag_slave	Avalon Memory Mapped Slave	<i>Click to export</i>	
<input checked="" type="checkbox"/>	onchip_mem	On-Chip Memory (RAM or ROM)		
	clk1	Clock Input	<i>Click to export</i>	
	s1	Avalon Memory Mapped Slave	<i>Click to export</i>	
	reset1	Reset Input	<i>Click to export</i>	
<input checked="" type="checkbox"/>	nios2_qsys	Nios II Processor		
	clk	Clock Input	<i>Click to export</i>	
	reset_n	Reset Input	<i>Click to export</i>	
	data_master	Avalon Memory Mapped Master	<i>Click to export</i>	
	instruction_master	Avalon Memory Mapped Master	<i>Click to export</i>	
	jtag_debug_module_re...	Reset Output	<i>Click to export</i>	
	jtag_debug_module	Avalon Memory Mapped Slave	<i>Click to export</i>	
	custom_instruction_m...	Custom Instruction Master	nios2_qsys_0_custom_i...	
<input checked="" type="checkbox"/>	timer	Interval Timer		
	clk	Clock Input	<i>Click to export</i>	
	reset	Reset Input	<i>Click to export</i>	
	s1	Avalon Memory Mapped Slave	<i>Click to export</i>	
<input checked="" type="checkbox"/>	mylcd	lcd		
	clock_sink	Clock Input	<i>Click to export</i>	
	reset_sink	Reset Input	<i>Click to export</i>	
	avalon_slave	Avalon Memory Mapped Slave	<i>Click to export</i>	
	conduit_end	Conduit	mylcd_conduit_end	
<input checked="" type="checkbox"/>	myspi	SPI (3 Wire Serial)		
	clk	Clock Input	<i>Click to export</i>	
	reset	Reset Input	<i>Click to export</i>	
	spi_control_port	Avalon Memory Mapped Slave	<i>Click to export</i>	
	external	Conduit Endpoint	myspi_external	

在 myspi 组件的右侧 IRQ 一列中，我们鼠标放置上去后有一个空心的原点，如图所示，这是 SPI 总线的中断连接。

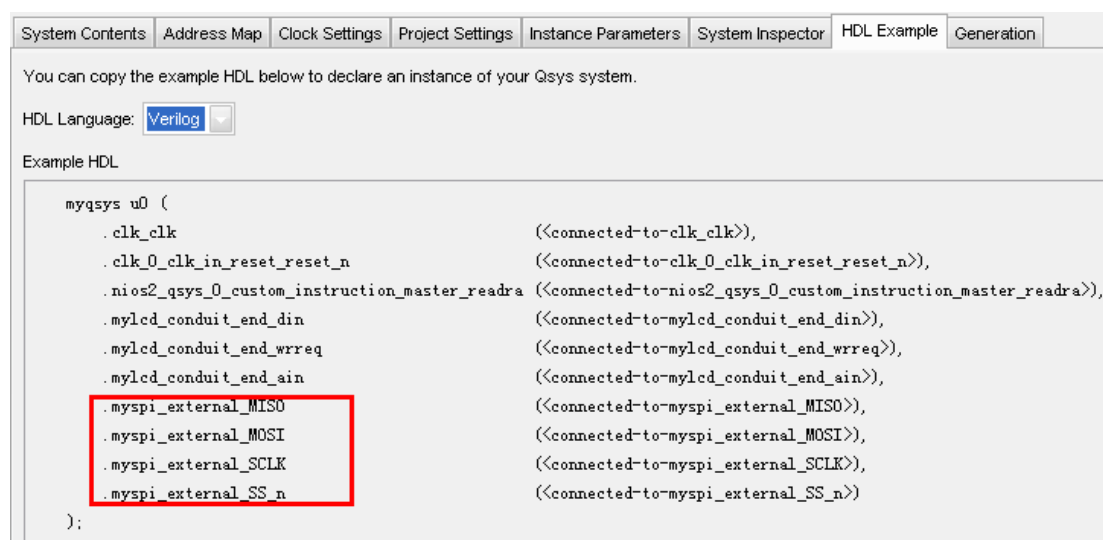


点击空心圆点后，如图所示，上面出现了一个数字，是当前 `mypsi` 组件的 IRQ 中断等级号。当然了，我们可以根据我们的需要去更改这些 IRQ 号的分配。



最后，我们需要点击菜单栏的 **System→Assign Base Addresses**，对系统的地址做重新自动分配，接着就可以来到 **Generation** 页面，点击右下角的 **Generate**。好了，大家可以先去泡杯茶，等待 Qsys 完成 **Generate**。

完成 **Generate** 后，我们在看看 **HDL Example** 页面，注意之前我们删除的那些外设相关的 **external** 接口都没有了，但是多出了 4 个 **SPI** 的接口信号。Copy 这个模块，到工程顶层代码文件中进行映射。





我们先看看在 ex16 工程基础上修改后的新工程代码顶层文件。

```
module ex2(
    clk, rst_n,
    lcd_en, lcd_clk, lcd_hsy, lcd_vsy, lcd_db_r, lcd_db_g, lcd_db_b,
    sdram_clk, sdram_cke, sdram_cs_n, sdram_ras_n, sdram_cas_n, sdram_we_n,
    sdram_ba, sdram_addr, sdram_data,
    spi_miso, spi_mosi, spi_sclk, spi_cs_n
);
input clk;
input rst_n;
    // FPGA 与 LCD 接口信号
output lcd_en; //背光使能信号, 高有效
output lcd_clk; //时钟信号
output lcd_hsy; //行同步信号
output lcd_vsy; //场同步信号
output[4:0] lcd_db_r;
output[5:0] lcd_db_g;
output[4:0] lcd_db_b;
    // FPGA 与 SDRAM 硬件接口
output sdram_clk; // SDRAM 时钟信号
output sdram_cke; // SDRAM 时钟有效信号
output sdram_cs_n; // SDRAM 片选信号
output sdram_ras_n; // SDRAM 行地址选通脉冲
output sdram_cas_n; // SDRAM 列地址选通脉冲
output sdram_we_n; // SDRAM 写允许位
output[1:0] sdram_ba; // SDRAM 的 L-Bank 地址线
output[11:0] sdram_addr; // SDRAM 地址总线
inout[15:0] sdram_data; // SDRAM 数据总线
    // FPGA 与字库芯片的 SPI 接口
input spi_miso; //SPI 总线的 MISO 信号, 即主机 (FPGA) 输出从机 (字库芯片) 输入信号
output spi_mosi; //SPI 总线的 MOSI 信号, 即主机 (FPGA) 输入从机 (字库芯片) 输出信号
output spi_sclk; //SPI 总线的时钟信号
output spi_cs_n; //SPI 总线的片选信号, 低电平有效

//-----
//LCD 与 FIFO 的接口
```




```
wire[15:0] rdfifo_rddb;    //FIFO 读出数据总线
wire rdfifo_rdreq;    //FIFO 读请求信号
wire rdfifo_clr;    //FIFO 复位信号, 高电平有效
    // SDRAM 的封装接口
wire sdram_wr_req;    //系统写 SDRAM 请求信号
wire sdram_rd_req;    //系统读 SDRAM 请求信号
wire sdram_wr_ack;    //系统写 SDRAM 响应信号, 作为 wrFIFO 的输出有效信号
wire sdram_rd_ack;    //系统读 SDRAM 响应信号, 作为 rdFIFO 的输写有效信号

wire[8:0] sdwr_byte = 9'd1;    //突发写 SDRAM 字节数 (1-256 个)
wire[8:0] sdrd_byte = 9'd160;    //突发读 SDRAM 字节数 (1-256 个)
wire[21:0] sys_wraddr;    // 写 SDRAM 时地址暂存器, (bit21-20)L-Bank 地址: (bit19-8)为行地址, (bit7-0)为列地址
wire[21:0] sys_rdaddr;    // 读 SDRAM 时地址暂存器, (bit21-20)L-Bank 地址: (bit19-8)为行地址, (bit7-0)为列地址
wire[15:0] sys_data_in;    //写 SDRAM 时数据暂存器
wire[15:0] sys_data_out;    //sdram 数据读出缓存 FIFO 输入数据总线
    //wrFIFO 输入控制接口
wire[15:0] wrf_din;    //sdram 数据写入缓存 FIFO 输入数据总线
wire[21:0] wrf_ain;    //sdram 数据写入缓存 FIFO 输入地址总线
wire wrf_wrreq;    //sdram 数据写入缓存 FIFO 数据输入请求, 高有效
    //系统控制相关信号接口
wire clk_25m;    //PLL 输出 25MHz 时钟
wire clk_50m;    //PLL 输出 50MHz 时钟
wire clk_100m;    //PLL 输出 100MHz 时钟
wire sys_rst_n;    //系统复位信号, 低有效

//-----
//例化系统复位信号和 PLL 控制模块
sys_ctrl    uut_sysctrl(
    .clk(clk),
    .rst_n(rst_n),
    .sys_rst_n(sys_rst_n),
    .clk_25m(clk_25m),
    .clk_50m(clk_50m),
    .clk_100m(clk_100m),
    .sdram_clk(sdram_clk)
);
```



```
//-----  
//Qsys 系列例化  
myqsys u0 (  
    //clk.clk  
    .clk_clk (clk_25m),  
    //clk_0_clk_in_reset.reset_n  
    .clk_0_clk_in_reset_reset_n(sys_rst_n),  
    //nios2_qsys_0_custom_instruction_master.readra  
    .nios2_qsys_0_custom_instruction_master_readra ( ),  
    //mylcd_conduit_end.din  
    .mylcd_conduit_end_din(wrf_din),  
    //.wrreq  
    .mylcd_conduit_end_wrreq(wrf_wrreq),  
    //.ain  
    .mylcd_conduit_end_ain(wrf_ain),  
    //.myspi_external.MISO  
    .myspi_external_MISO(spi_miso),  
    //.MOSI  
    .myspi_external_MOSI(spi_mosi),  
    //.SCLK  
    .myspi_external_SCLK(spi_sclk),  
    //.SS_n  
    .myspi_external_SS_n(spi_cs_n)  
);  
  
//-----  
//LCD 驱动模块  
lcd_driver uut_lcd_driver(  
    .clk(clk_25m), //25MHz  
    .rst_n(sys_rst_n),  
    .lcd_en(lcd_en),  
    .lcd_clk(lcd_clk),  
    .lcd_hsy(lcd_hsy),  
    .lcd_vsy(lcd_vsy),  
    .lcd_db_r(lcd_db_r),  
    .lcd_db_g(lcd_db_g),  
    .lcd_db_b(lcd_db_b),
```



```
.rdfifo_rddb(rdfifo_rddb),//
.rdfifo_rdreq(rdfifo_rdreq),//
.rdfifo_clr(rdfifo_clr)//
);

//-----
//例化 SDRAM 封装控制模块
sdram_top      uut_sdramtop(          // SDRAM
    .clk(clk_100m),
    .rst_n(sys_rst_n),
    .sdram_wr_req(sdram_wr_req),//
    .sdram_rd_req(sdram_rd_req),//
    .sdram_wr_ack(sdram_wr_ack),//
    .sdram_rd_ack(sdram_rd_ack),//
    .sys_wraddr(sys_wraddr),//
    .sys_rdaddr(sys_rdaddr),//
    .sys_data_in(sys_data_in),//
    .sys_data_out(sys_data_out),//
    .sdwr_byte(sdwr_byte),//
    .sdrd_byte(sdrd_byte),//
    .sdram_cke(sdram_cke),
    .sdram_cs_n(sdram_cs_n),
    .sdram_ras_n(sdram_ras_n),
    .sdram_cas_n(sdram_cas_n),
    .sdram_we_n(sdram_we_n),
    .sdram_ba(sdram_ba),
    .sdram_addr(sdram_addr),
    .sdram_data(sdram_data),
    .sdram_udqm(),
    .sdram_ldqm()
);

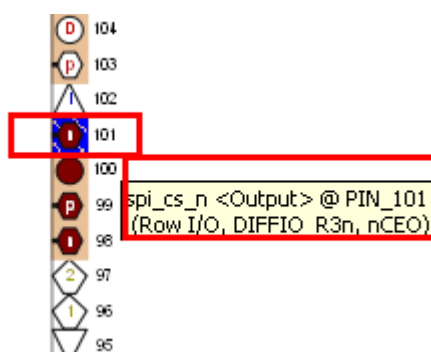
//-----
//读写 SDRAM 数据缓存 FIFO 模块例化
sdfifo_ctrl      uut_sdfifofctrl(
    .clk_25m(clk_25m),
    .clk_100m(clk_100m),
```



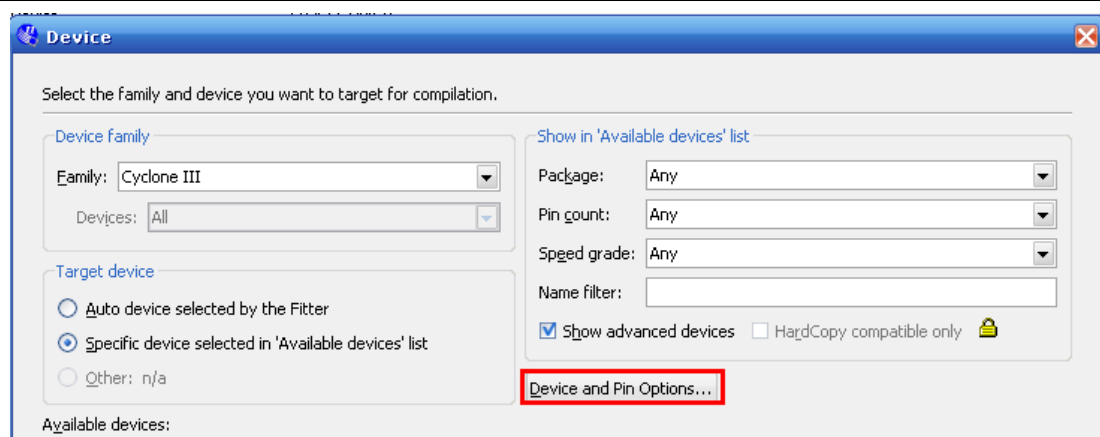
```
.rst_n(sys_rst_n),  
.wrf_din(wrf_din),  
.wrf_wrreq(wrf_wrreq),  
.wrf_ain(wrf_ain),  
.sdram_wr_ack(sdram_wr_ack), //  
.sys_wraddr(sys_wraddr), //  
.sys_rdaddr(sys_rdaddr), //  
.sys_data_in(sys_data_in), //  
.sdram_wr_req(sdram_wr_req), //  
.sys_data_out(sys_data_out), //  
.sdram_rd_ack(sdram_rd_ack), //  
.sdram_rd_req(sdram_rd_req), //  
.rdfifo_rdreq(rdfifo_rdreq), //  
.rdfifo_clr(rdfifo_clr), //  
.rdfifo_rddb(rdfifo_rddb) //  
);
```

endmodule

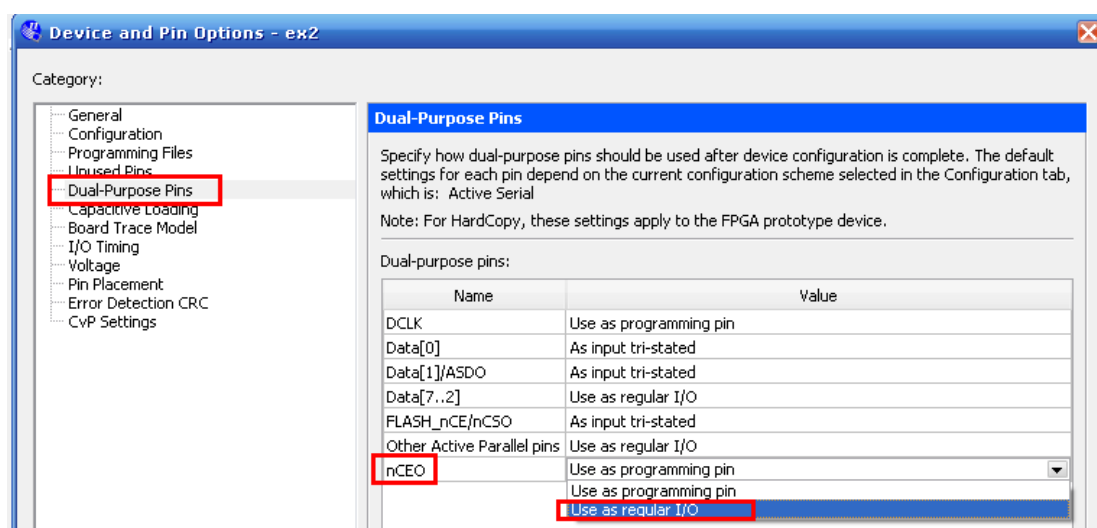
下面, 哦, 别急着编译, 还有管脚分配的任务, 但在这之前, 还有一项更重要的任务需要做。如图所示, **PIN101** 是我们将要分配的 **spi_cs_n** 的管脚, 它有一个功能是 **nCEO**, 这是一个配置功能管脚, 若想用这些管脚做 I/O, 那么需要在 **Quartus II** 工具做一些配置, 否则编译时肯定要报错。



点击 **Quartus II** 菜单栏的 **Assignment**→**Device**, 接着在页面中点击 **Device and Pin Option** 按钮。



如图所示,新弹出的页面何种,左侧 Category 下选择 Dual-Purpose Pins, 右侧有一个 nCEO 即我们想复用为 I/O 管脚的配置管脚, 选择它的 Value 为 Use as regular I/O 即可。



进行一次综合编译, 然后对新添加的 4 个 SPI 总线接口做管脚分配, 如图所示。

Node Name	Direction	Location	I/O Bank
sdram_data[2]	Bidir	PIN_7	1
sdram_data[1]	Bidir	PIN_10	1
sdram_data[0]	Bidir	PIN_11	1
sdram_ras_n	Output	PIN_141	8
sdram_we_n	Output	PIN_143	8
spi_cs_n	Output	PIN_101	6
spi_miso	Input	PIN_100	6
spi_mosi	Output	PIN_98	6
spi_sclk	Output	PIN_99	6

最后进行一次全编译, SF-CY3 核心板连接上 SF-LCD 板以及 SF-SENSOR 板, 连接好 USB 下载线, 给板子上电。下载该工程编译成功的 sof 文件到 SF-CY3 核心板上。好, 硬件平台一切就绪, 下面我们就要开始软件编程的相关工作了。



8.2.2 SPI 外设驱动——编程原理

如前面的 SPI 外设框图所示，这个外设主要有 4 个寄存器，如图所示。数据收发寄存器用于存储当前收发数据，状态寄存器指示当前收发状态，控制寄存器则用于收发使能等控制操作。而对于有多个从机的组件，还有一个片选寄存器，用于控制选择不同的从机。

数据接收寄存器
 数据发送寄存器
 状态寄存器
 控制寄存器
 片选寄存器（多个从机）

Table 8-3. Register Map for SPI Master Device

Internal Address	Register Name	Type [R/W]	32..11	10	9	8	7	6	5	4	3	2	1	0
0	rxdata (1)	R	RXDATA (n-1..0)											
1	txdata (1)	W	TXDATA (n-1..0)											
2	status (2)	R/W				E	RRDY	TRDY	TMT	TOE	ROE			
3	control	R/W		SSO (3)		IE	IRRDY	ITRDY		ITOE	IROE			
4	Reserved	—												
5	slaveselct (3)	R/W	Slave Select Mask											

状态寄存器的各个具体位定义如下。

Table 8-4. status Register Bits

#	Name	Description
3	ROE	Receive-overflow error The ROE bit is set to 1 if new data is received while the rxdata register is full (that is, while the RRDY bit is 1). In this case, the new data overwrites the old. Writing to the status register clears the ROE bit to 0.
4	TOE	Transmitter-overflow error The TOE bit is set to 1 if new data is written to the txdata register while it is still full (that is, while the TRDY bit is 0). In this case, the new data is ignored. Writing to the status register clears the TOE bit to 0.
5	TMT	Transmitter shift-register empty In master mode, the TMT bit is set to 0 when a transaction is in progress and set to 1 when the shift register is empty. In slave mode, the TMT bit is set to 0 when the slave is selected (ss_n is low) or when the SPI Slave register interface is not ready to receive data.
6	TRDY	Transmitter ready The TRDY bit is set to 1 when the txdata register is empty.
7	RRDY	Receiver ready The RRDY bit is set to 1 when the rxdata register is full.
8	E	Error The E bit is the logical OR of the TOE and ROE bits. This is a convenience for the programmer to detect error conditions. Writing to the status register clears the E bit to 0.

控制寄存器的各个具体位的定义如下。

Table 8-5. control Register Bits

#	Name	Description
3	IROE	Setting IROE to 1 enables interrupts for receive-overflow errors.
4	ITOE	Setting ITOE to 1 enables interrupts for transmitter-overflow errors.
6	ITRDY	Setting ITRDY to 1 enables interrupts for the transmitter ready condition.
7	IRRDY	Setting IRRDY to 1 enables interrupts for the receiver ready condition.
8	IE	Setting IE to 1 enables interrupts for any error condition.
10	SSO	Setting SSO to 1 forces the SPI core to drive its ss_n outputs, regardless of whether a serial shift operation is in progress or not. The slaveselct register controls which ss_n outputs are asserted. SSO can be used to transmit or receive data of arbitrary size, for example, greater than 32 bits.



光是看看寄存器，恐怕还不知道如何操作这个 SPI 组件，让他能够正常收发。其实很简单，我们的实例中为这个 SPI 外设的驱动写了两个函数，一个是 SPI 外设的初始化，另一个是 SPI 外设的收发数据函数。聪明人一看代码就能明白，况且咱的程序还有详细注释呢。

SPI 外设初始化函数如下。

```
/******  
* 名 称: 初始化 SPI 的 control 和 status 寄存器  
* 函数名: init_spi  
* 入 口: 无  
* 出 口: 无  
*****/  
void init_spi(void)  
{  
    IOWR_ALTERA_AVALON_SPI_CONTROL(MYSPI_BASE, 0xc0);    //设置SPI的control寄存器  
    //0xc0 即 bit6=1, bit7=1  
    //Bit6 ITRDY 置 1 则使能发送等待标志位 (TRDY)  
    //Bit7 IRRDY 置 1 则使能发送等待标志位 (RRDY)  
  
    IOWR_ALTERA_AVALON_SPI_STATUS(MYSPI_BASE, 0x60);    //SPI 的 status 寄存器清零  
    //0x60 即 bit5=0, bit6=0  
    //Bit5 TMT 传输移位寄存器空标志。(主机模式下)当 TMT=0 时表示一次传输正在进行,  
    TMT=1 表示移位寄存器空。  
    //Bit6 TRDY 发送等待。TRDY=1 表示 txdata 寄存器空, 可以发起新的一次写入。  
}
```

SPI 收发数据函数如下。

```
/******  
* 名 称: SPI 主机进行一次数据读写  
* 函数名: spi_process  
* 入 口: alt_u8 txdata--SPI 主机传送数据  
* 出 口: alt_u8 rxdata--SPI 主机接收数据  
*****/  
alt_u8 spi_process(alt_u8 txdata)  
{  
    alt_u8 rxdata;  
    alt_u32 spi_status_reg;
```



```
//发送数据
do{
    spi_status_reg = IORD_ALTERA_AVALON_SPI_STATUS(MYSPI_BASE); //读 SPI 的
status 寄存器
    }while((spi_status_reg & 0x40) != 0x40);    //TRDY=1 表示 txdata 寄存器空，可
以发起新的一次写入，TRDY=0 则等待

    IOWR_ALTERA_AVALON_SPI_TXDATA(MYSPI_BASE, txdata);    //写 SPI 的 tx_data 寄存
器，发送 8bit 数据给从机

//接收数据
do{
    spi_status_reg = IORD_ALTERA_AVALON_SPI_STATUS(MYSPI_BASE); //读 SPI 的
status 寄存器
    }while((spi_status_reg & 0x80) != 0x80);    //RRDY=1 表示 rxdata 寄存器满，可
供读取，RRDY=0 则等待

    rxdata = IORD_ALTERA_AVALON_SPI_RXDATA(MYSPI_BASE); //读 SPI 的 rxdata 寄存器

    return(rxdata);
}
```

8.2.3 字库芯片驱动——编程原理

我们使用的字库芯片 GT21L16S2W，它包含的字库文件如下表所列。正可谓琳琅满目，把大家看晕了，咱可不想一口吃成大胖子。不求多，我们只需要其中两个字库，即 15X16 点 GB2312 标准点阵字库（是一种常用中文字库标准，大家可以 baidu 下这个标准）和 8X16 点 ASCII 字符。



	字库内容	编码体系	码位范围	字符数	起始地址	结束地址	参考算法
1	15X16 点 GB2312 标准点阵字库	GB2312	A1A1-F7FE	6763+376	00000	3B7BF	6.3.1.2
2	GB2312 到 Unicode 内码转换表				2F00	66BF	6.4
3	7X8 点 ASCII 字符	ASCII	20~7F	96	66C0	69BF	6.3.2.2
4	8X16 点国标扩展字符	GB2312	AAA1-ABC0	126	3B7C0	3BFBF	6.3.1.4
5	8X16 点 ASCII 字符	ASCII	20~7F	96	3B7C0	3BFBF	6.3.2.4
6	5X7 点 ASCII 字符	ASCII	20~7F	96	3BFC0	3C2BF	6.3.2.1
7	16 点阵不等宽 ASCII 方头 (Arial) 字符	ASCII	20~7F	96	3C2C0	3CF7F	6.3.2.6
8	11X12 点 GB2312 标准点阵字库	GB2312	A1A1-F7FE	6763+376	3CF80	66D3F	6.3.1.3
9	6X12 点国标扩展字符	GB2312	AAA1-ABC0	126	66D40	6733F	6.3.1.3
10	6X12 点 ASCII 字符	ASCII	20~7F	96	66D40	6733F	6.3.2.3
11	12 点阵不等宽 ASCII 方头 (Arial) 字符	ASCII	20~7F	96	67340	67CFF	6.3.2.5
12	保留区				67D00	67D6F	
13	Unicode 到 GB2312 内码转换表				67D70	7278F	6.5
14	GT 快捷拼音输入法码表				72790	7FA33	
15	保留区				7FA33	7FFFF	

先来说 ASCII 字符的字模数据获取原理, 毕竟 ex16 工程中我们刚刚用自己生成的字库文件做过显示。芯片手册上给出的算法提示如下。

6.3.2.4 8X16 点 ASCII 字符

说明:

ASCIICode: 表示 ASCII 码 (8bits)

BaseAdd: 说明该套字库在芯片中的起始地址。

Address: ASCII 字符点阵在芯片中的字节地址。

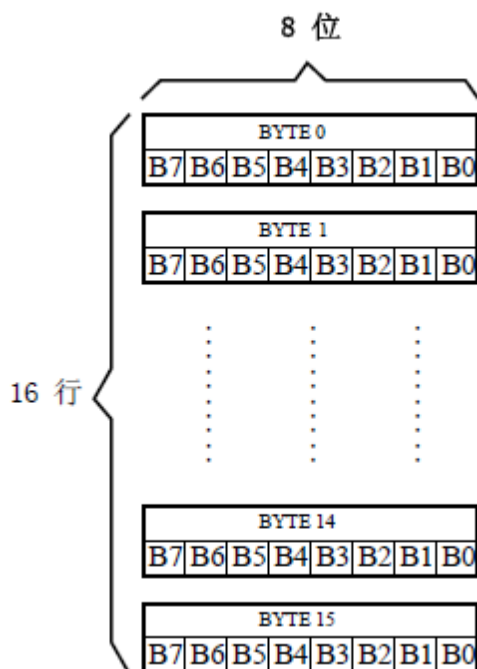
计算方法:

BaseAdd=0x3b7c0

if (ASCIICode >= 0x20) and (ASCIICode <= 0x7E) then

Address = (ASCIICode - 0x20) * 16 + BaseAdd

其实我们只关心一件事, 即我们给定了一个字模数据, 那么这个字模数据的其实地址和一共有多少个地址数据需要读出。哈哈, 这个其实非常简单, “计算方法” 下面写得明明白白。8X16 点阵的字模数据量也就是 16 个字节正好, 每个字节数据对应一行 8 个点的显示。具体的映射如图所示。每个 bit 的数据为 1 或为 0, 我们后面都可以对应的给 LCD 上的像素赋予一个色彩。



好, 原理清楚了, 下面给出这部分的函数。

```
alt_u8 char_db[32]; //当前字模数据缓存数组

/*****
* 名称: 8X16 点 ASCII 码字模数据读出函数
* 函数名: ascii_read
* 入口: alt_u8 ascii--需要读出 ascii 数据的字符
* 出口: 无
*****/
void ascii_read(alt_u8 ascii)
{
    alt_u32 addr;
    alt_u8 i;

    if((ascii >= 0x20) && (ascii <= 0x7e))
    {
        addr = ((ascii-0x20)<<4)+0x3b7c0;
        IOWR_ALTERA_AVALON_SPI_CONTROL(MYSPI_BASE, 0x4c0); //强行拉低 spi_cs_n
        信号, 使其保持有效
        spi_process(0x03); //读命令
        spi_process(addr>>16); //送地址 bit23-bit16
        spi_process(addr>>8); //送地址 bit15-bit8
    }
}
```



```
spi_process(addr);      //送地址 bit7-bit0
for(i=0;i<16;i++)
{
    char_db[i] = spi_process(0x00);
}
IOWR_ALTERA_AVALON_SPI_CONTROL(MYSPI_BASE, 0x0c0); //释放 spi_cs_n 信号
}
}
```

下一步我们要将这个 ASCII 码送给液晶屏显示, 可以使用 ex16 的函数 `print_ascii`, 将原来使用固定数组的字模数据, 改为先读取对应 ASCII 码的字模, 然后调用全局数组 `char_db` 来显示即可。函数如下。

```
/******
函数名: print_ascii
功 能: 在指定位置显示 8*16 的 ASCII 码字符串
输 入: alt_u16 uRow:      字符显示起始 x 坐标
      alt_u16 uCol:      字符显示起始 y 坐标
      alt_u8 *ptr:      写入的 ASCII 码字符串
      alt_u16 Cor_b0: 前景色彩
      alt_u16 Cor_q0: 背景色彩
返 回: 无
*****/
void print_ascii(alt_u16 uRow, alt_u16 uCol, alt_u8 *ptr, alt_u16 Cor_b0, alt_u16 Cor_q0)
{
    alt_u16 uLen=0; //字符串长度
    alt_u16 i, j, k=0;

    while ((alt_u8)ptr[uLen] >= 0x10) {uLen++;}; //探测字串长度

    while(k < uLen) //写入 uLen 个 ASCII 字符
    {
        if(ptr[k] <= 128) //ASCII 码
        {
            if(ptr[k] >= 0x10)
            {
                ascii_read(ptr[k]); //读出 ASCII 码字模数据
                for(j=0; j<16; j++) //显示 8×16 的 ASCII 码, 分 16 行输出

```



```
        {
            for(i=0;i<8;i++)
            {
                if((char_db[j] & (1<<(7-i))) != 0x00)
lcd_wrdp(uRow+i,uCol+j,Cor_q0); //送像素点前景色
                else lcd_wrdp(uRow+i,uCol+j,Cor_b0); //送像素点背景
色
            }
        }
    }
    uRow+=8; // x 坐标加 8, 即下一个字符位
}
k++; //下一个字符
}
}

////////////////////////////////////
//函数名: lcd_wrdp
//功 能: LCD 写数据函数
//输 入: alt_u16 xaddr--X 坐标地址; alt_u16 yaddr--Y 坐标地址; alt_u16 cor--
显示色彩
//返 回: 无
////////////////////////////////////
void lcd_wrdp(alt_u16 xaddr,alt_u16 yaddr,alt_u16 cor)
{
    IOWR_16DIRECT(MYLCD_BASE, ((yaddr<<10)+(xaddr<<1)),cor); //LCD 显示内存映射
地址写入色彩数据
}
```

下面我们再乘热打铁,看看本实例大家最期待的中文字库的驱动原理。同样的,芯片手册上给出的算法提示如下。



6.3.1.2 15X16 点 GB2312 标准点阵字库

参数说明:

GBCode表示汉字内码。

MSB 表示汉字内码GBCode 的高8bits。

LSB 表示汉字内码GBCode 的低8bits。

Address 表示汉字或ASCII字符点阵在芯片中的字节地址。

BaseAdd: 说明点阵数据在字库芯片中的起始地址。

计算方法:

BaseAdd=0;

if(MSB >=0xA4 && MSB <= 0xA8 && LSB >=0xA1)

Address = BaseAdd;

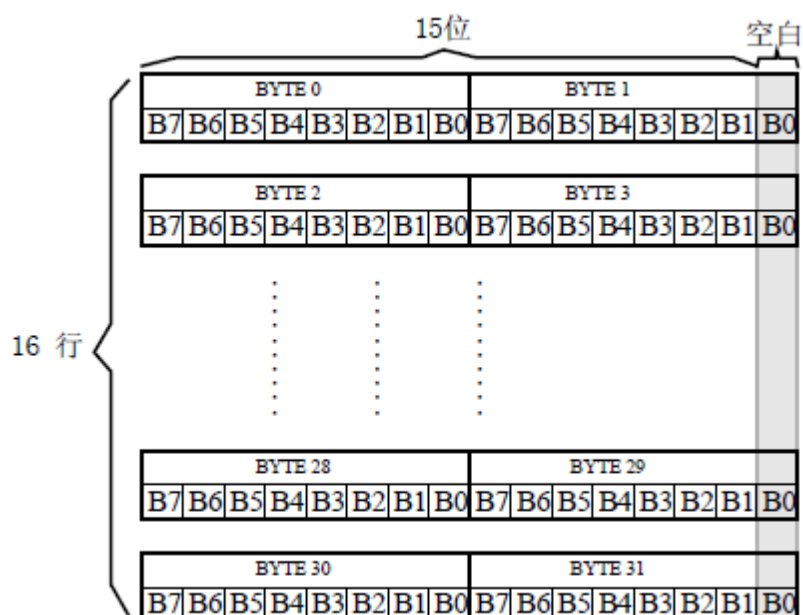
else if(MSB >=0xA1 && MSB <= 0xA9 && LSB >=0xA1)

Address = (MSB - 0xA1) * 94 + (LSB - 0xA1)*32+ BaseAdd;

else if(MSB >=0xB0 && MSB <= 0xF7 && LSB >=0xA1)

Address = ((MSB - 0xB0) * 94 + (LSB - 0xA1)+ 846)*32+ BaseAdd;

关于 GB2312 标准对中文字符的定义和表示方法,大家最好 baidu 一下,至少要明白一二,后面特权同学才好点拨。当然了,某些码农就是不喜欢看什么厚厚一摞又啰嗦又繁琐的标准,就是喜欢操起代码程序没命的试,最终也能把原理搞懂。这个技能必须由,呵呵,大家看上面的计算方法自己消化吧——只给一点提示,GB2312 定义的中文字符都是两个字节的,即没两个自己代表一个中文字符。好了,不多说,免得抱怨我糟蹋大家的智商。应广大码农要求,后面马上送出代码。15X16 点阵读出数据对应的显示位如图所示。





```
* 名 称: 15X16 点 ASCII 码字模数据读出函数
* 函数名: gb2312_read
* 入 口: alt_u8 MSB, LSB -- 需要读出字模的国标码
* 出 口: 无
*****/
void gb2312_read(alt_u8 MSB, alt_u8 LSB)
{
    alt_u32 addr;
    alt_u8 i;

    if((MSB >=0xa4) && (MSB <= 0xa8) && (LSB >=0xa1))
    {
        addr = 0;
    }
    else if((MSB >=0xa1) && (MSB <= 0xa9) && (LSB >=0xa1))
    {
        addr =( MSB - 0xa1) * 94 + (LSB - 0xa1))*32;
    }
    else if((MSB >=0xb0) && (MSB <= 0xf7) && (LSB >=0xa1))
    {
        addr = ((MSB - 0xb0) * 94 + (LSB - 0xa1)+ 846)*32;
    }

    IOWR_ALTERA_AVALON_SPI_CONTROL(MYSPI_BASE, 0x4c0); //强行拉低 spi_cs_n 信号,
    使其保持有效
    spi_process(0x03); //读命令
    spi_process(addr>>16); //送地址 bit23-bit16
    spi_process(addr>>8); //送地址 bit15-bit8
    spi_process(addr); //送地址 bit7-bit0
    for(i=0; i<32; i++)
    {
        char_db[i] = spi_process(0x00);
    }
    IOWR_ALTERA_AVALON_SPI_CONTROL(MYSPI_BASE, 0x0c0); //释放 spi_cs_n 信号
}
```

```
/******
```



函数名: print_gb2312

功 能: 在指定位置显示 15*16 的 GB2312 字符串

输 入: alt_ul6 uRow: 字符显示起始 x 坐标

alt_ul6 uCol: 字符显示起始 y 坐标

alt_u8 *ptr: 写入的 GB2312 字符串

alt_ul6 Cor_b0: 前景色彩

alt_ul6 Cor_q0: 背景色彩

返 回: 无

```
*****/
void print_gb2312(alt_ul6 uRow,alt_ul6 uCol,alt_u8 *ptr,alt_ul6 Cor_b0,alt_ul6
Cor_q0)
{
    alt_ul6 uLen=0; //字符串长度
    alt_ul6 i,j,k=0;

    while ((alt_u8)ptr[uLen] >= 0x10)    {uLen++;};    //探测字符串长度

    while(k < uLen) //写入(uLen/2)个 gb2312 字符
    {
        gb2312_read(ptr[k],ptr[k+1]);    //读出 ASCII 码字模数据
        for(j=0;j<16;j++)    //显示 15×16 的 GB2312 字符,分 16 行输出
        {
            for(i=0;i<8;i++)
            {
                if((char_db[(j<<1)]    &    (1<<(7-i)))    !=    0x00)
lcd_wrdp(uRow+i,uCol+j,Cor_q0); //送像素点前景色
                else lcd_wrdp(uRow+i,uCol+j,Cor_b0);    //送像素点背景色
            }
            for(i=0;i<8;i++)
            {
                if((char_db[(j<<1)+1]    &    (1<<(7-i)))    !=    0x00)
lcd_wrdp(uRow+8+i,uCol+j,Cor_q0); //送像素点前景色
                else lcd_wrdp(uRow+8+i,uCol+j,Cor_b0); //送像素点背景色
            }
        }
        uRow+=16;    // x 坐标加 15,即下一个字符位
        k+=2;    //下一个字符
    }
}
```



```
}
```

8.2.4 软件工程实例

新建软件工程, 命名为 `ex17swprj`。参照前面的工程在 `BSP Editor` 中做代码裁剪。新建 `main.c` 文件, 输入前面的一些函数代码。这里, 我们再简单的看看这个程序的主函数。

```
#include "alt_types.h"
#include "sys/alt_irq.h"
#include "system.h"
#include <stdio.h>
#include <unistd.h>
#include <altera_avalon_spi.h>
#include <altera_avalon_spi_regs.h>

//LCD 驱动函数
void lcd_wrdb(alt_u16 xaddr, alt_u16 yaddr, alt_u16 cor); //LCD 写数据函数
void print_ascii(alt_u16 uRow, alt_u16 uCol, alt_u8 *ptr, alt_u16 Cor_b0, alt_u16
Cor_q0); //在指定位置显示 8*16 的 ASCII 码字符串
void print_gb2312(alt_u16 uRow, alt_u16 uCol, alt_u8 *ptr, alt_u16 Cor_b0, alt_u16
Cor_q0); //在指定位置显示 15*16 的 GB2312 字符串
//字库芯片驱动函数
void ascii_read(alt_u8 ascii); //8X16 点 ASCII 码字模数据读出函数
void gb2312_read(alt_u8 MSB, alt_u8 LSB); //15X16 点 ASCII 码字模数据读出函数
//延时函数
void delay(alt_u32 cnt); //延时函数
//SPI 函数
void init_spi(void); //初始化 SPI 的 control 和 status 寄存器
alt_u8 spi_process(alt_u8 txdata); //SPI 主机进行一次数据读写

alt_u8 char_db[32]; //当前字模数据缓存数组

/*****
* 名 称: 主函数
* 函数名: main
* 入 口: 无
* 出 口: 无
```




```
*****/
void main(void)
{
    alt_ul6 x,y;

    //delay(1000);

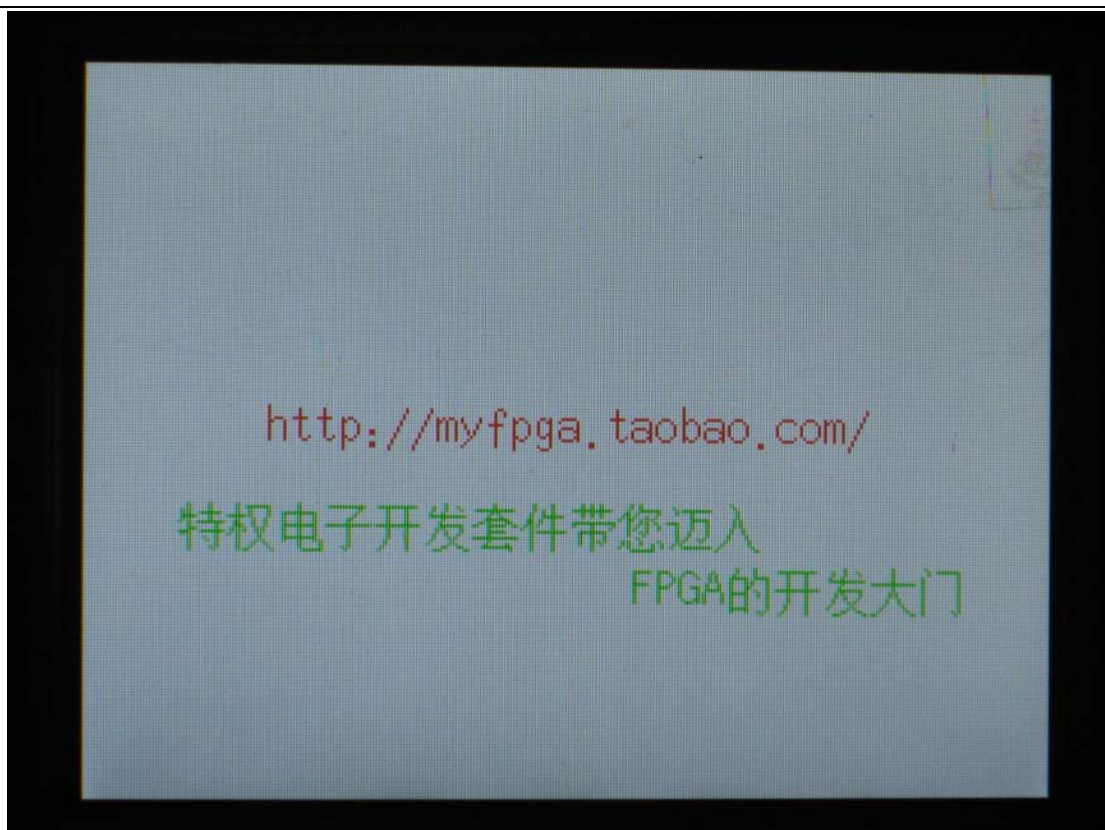
    //清全屏为蓝色
    for(y=0;y<240;y++)
    {
        for(x=0;x<320;x++)
        {
            lcd_wrdp(x,y,0xffff);
        }
    }

    init_spi(); //初始化 SPI 的 control 和 status 寄存器

    print_ascii(60,111,"http://myfpga.taobao.com/",0xffff,0xf800);
    print_gb2312(32,143,"特权电子开发套件带您迈入",0xffff,0x07e0);
    print_ascii(180,164,"FPGA",0xffff,0x07e0);
    print_gb2312(212,164,"的开发大门",0xffff,0x07e0);

    while(1);
}
```

最终,我们将软件工程运行到目标板上,可以看到 SF-LCD 的液晶屏上显示出了我们编程所设置的字符串。这个实例只是给大家一些基本编程驱动思路,当然也是实现了一个还算 cool 的效果,不过想要不断的完善这个字库芯片显示功能,还需要靠大家一起动手努力。革命尚未成功,同志们还需努力啊!



8.3 基于 Qsys 的 NIOS II 实例 9——IIC 接口实时时钟(RTC) 芯片控制

8.3.1 RTC 实时时钟芯片驱动原理

本实例使用的 RTC 实时时钟芯片是型号为 PCF8563，是 PHILIPS 公司推出的一款工业级内含 IIC 总线接口功能的具有极低功耗的多功能时钟/日历芯片。PCF8563 的多种报警功能、定时器功能、时钟输出功能以及中断输出功能能完成各种复杂的定时服务，甚至可为单片机提供看门狗功能。内部时钟电路、内部振荡电路、内部低电压检测电路（1.0V）以及两线制 IIC 总线通讯方式，不但使外围电路及其简洁，而且也增加了芯片的可靠性。同时每次读写数据后内嵌的字地址寄存器会自动产生增量，因而 PCF8563 是一款性价比极高的时钟芯片，它已被广泛用于电表、水表、气表、电话、传真机、便携式仪器以及电池供电的仪器仪表等产品领域。

特性



- 宽电压范围 1.0 5.5V 复位电压标准值 Vlow=0.9V
- 超低功耗典型值为 0.25 A VDD=3.0V,Tamb=25° C
- 可编程时钟输出频率为 32.768KHz/1024Hz/32Hz/1Hz
- 四种报警功能和定时器功能
- 内含复位电路振荡器电容和掉电检测电路
- 开漏中断输出
- 400kHz 的 IIC 总线(VDD=1.8-5.5V) 其从地址读 0xa3，写 0xa2

PCF8563 的管脚排列及描述如下所示。

表 1 PCF8563 管脚描述		
符号	管脚号	描 述
OSCI	1	振荡器输入
OSCO	2	振荡器输出
/INT	3	中断输出（开漏；低电平有效）
VSS	4	地
SDA	5	串行数据 I/O
SCL	6	串行时钟输入
CLKOUT	7	时钟输出（开漏）
VDD	8	正电源

PCF8563 有 16 个位寄存器：一个可自动增量的地址寄存器，一个内置 32.768KHz 的振荡器（带有一个内部集成的电容），一个分频器用于给实时时钟 RTC 提供源时钟，一个可编程时钟输出，一个定时器，一个报警器，一个掉电检测器和一个 400KHz 的 IIC 总线接口。

所有 16 个寄存器设计成可寻址的 8 位并行寄存器，但不是所有位都有用。前两个寄存器（内存地址 00H 01H）用于控制寄存器和状态寄存器，内存地址 02H~08H 用于时钟计数器(秒~年计数器),地址 09H~0CH 用于报警寄存器(定义报警条件),地址 0DH 控制 CLKOUT 管脚的输出频率,地址 0EH 和 0FH 分别用于定时器控制寄存器和定时器寄存器。秒、分钟、小时、日、月、年、分钟报警、小时报警、日报警寄存器，编码格式为 BCD，星期和星期报警寄存器不以 BCD 格式编码。

当一个 RTC 寄存器被读时，所有计数器的内容被锁存。因此，在传送条件下，可以禁止对时钟日历芯片的错读。

PCF8563 共有 16 个寄存器，其中 00H~01H 为控制方式寄存器；09H~0CH 为报警功能寄存器；0DH 为时钟输出寄存器；0EH 和 0FH 为定时器功能寄存器，02H~08H 为秒~年时间寄存器。各寄存器的位描述如下示。



二进制格式寄存器概况

地址	寄存器名称	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
00H	控制/状态寄存器 1	TEST1	0	STOP	0	TESTC	0	0	0
01H	控制/状态寄存器 2	0	0	0	TI/TP	AF	TF	AIE	TIE
0DH	CLKOUT 输出寄存器	FE	—	—	—	—	—	FD1	FD0
0EH	定时器控制寄存器	TE	—	—	—	—	—	TD1	TD0
0FH	定时器倒数计数 数值寄存器	定时器倒数计数数值(二进制)							

BCD 格式寄存器概况

地址	寄存器名称	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
02h	秒	VL	00~59BCD 码格式数						
03h	分钟	—	00~59BCD 码格式数						
04h	小时	—	—	00~59BCD 码格式数					
05h	日	—	—	01~31BCD 码格式数					
06h	星期	—	—	—	—	—	0~6		
07h	月/世纪	C	—	—	01~12 BCD 码格式数				
08h	年	00~99 BCD 码格式数							
09h	分钟报警	AE	00~59 BCD 码格式数						
0Ah	小时报警	AE	—	00~23 BCD 码格式数					
0BH	日报警	AE	—	01~31 BCD 码格式数					
0CH	星期报警	AE	—	—	—	—	0~6		

注:标明“—”的位无效

(1) 控制/状态寄存器 1

表 4 控制/状态寄存器 1 位描述 (地址 00H)

Bit	符号	描 述
7	TEST1	TEST1=0, 普通模式; TEST1=1, EXT_CLK 测试模式
5	STOP	STOP=0, 芯片时钟运行; STOP=1, 所有芯片分频器异步置逻辑 0。芯片时钟停止运行 (CLKOUT 在 32.768kHz 时可用)
3	TESTC	TESTC=0, 电源复位功能失效 (普通模式时置逻辑 0) TESTC=1, 电源复位功能有效
6,4,2,1,0	0	缺省值置逻辑 0



(2) 控制/状态寄存器 2

表 5 控制/状态寄存器 2 位描述 (地址 01H)

Bit	符号	描述
7,6,5	0	缺省值置逻辑 0
4	TI/TP	TI/TP=0: 当 TF 有效时 INT 有效 (取决于 TIE 的状态) TI/TP=1: INT 脉冲有效, 参见表 6 (取决于 TIE 的状态)。注意: 若 AF 和 AIE 都有效时, 则 INT 一直有效
3	AF	当报警发生时, AF 被置逻辑 1; 在定时器倒数计数结束时, TF 被置逻辑 1, 它们在被软件重写前一直保持原有值, 若定时器和报警中断都请求时, 中断源由 AF 和 TF 决定, 若要使清除一个标志位而防止另一标志位被重写, 应运用逻辑指令 AND, 标志位 AF 和 TF 值描述参见表 7
2	TF	
1	AIE	标志位 AIE 和 TIE 决定一个中断的请求有效或无效, 当 AF 或 TF 中一个为“1”时中断是 AIE 和 TIE 都置“1”时的逻辑或。
0	TIE	AIE=0, 报警中断无效; AIE=1, 报警中断有效 TIE=0, 定时器中断无效; TIE=1, 定时器中断有效

表 6 /INT 操作 (bit TI/TP=1)

源时钟 (Hz)	/INT 周期	
	n=1	n>1
4096	1/8192	1/4096
64	1/128	1/64
1	1/64	1/64
1/60	1/64	1/64

注 1. TF 和 /INT 同时有效

注 2. n 为倒数计数定时器的数值, 当 n=0 时定时器停止工作。

表 7 AF 和 TF 值描述

R/W	Bit: AF		Bit: TF	
	值	描述	值	描述
Read 读	0	报警标志无效	0	定时器标志无效
	1	报警标志有效	1	定时器标志有效
Write 写	0	报警标志被清除	0	定时器标志被清除
	1	报警标志保持不变	1	定时器标志保持不变



(3) 秒、分钟和小时寄存器

表 8 秒/VL 寄存器位描述 (地址 02H)

Bit	符号	描 述
7	VL	VL=0: 保证准确的时钟/日历数据 VL=1: 不保证准确的时钟/日历数据
6~0	<秒>	代表 BCD 格式的当前秒数值, 值为 00~99 例如: <秒>=1011001, 代表 59 秒

表 9 分钟寄存器位描述 (地址 03H)

Bit	符号	描 述
7	—	无效
6~0	<分钟>	代表 BCD 格式的当前分钟数值, 值为 00~59

表 10 小时寄存器位描述 (地址 04H)

Bit	符 号	描 述
7~6	—	无效
5~0	<小时>	代表 BCD 格式的当前小时数值, 值为 00~23

(4) 日、星期、月/世纪和年寄存器

表 11 日寄存器位描述 (地址 05H)

Bit	符号	描 述
7~6	—	无效
5~0	<日>	代表 BCD 格式的当前日数值, 值为 01~31。当年计数器的值是闰年时, PCF8563 自动给二月增加一个值, 使其成为 29 天

表 12 星期寄存器位描述 (地址 06H)

Bit	符号	描 述
7~3	—	无效
2~0	<星期>	代表当前星期数值 0~6, 参见表 13, 这些位也可由用户重新分配



表 13 星期分配表

日 (Day)	Bit2	Bit1	Bit0
星期日	0	0	0
星期一	0	0	1
星期二	0	1	0
星期三	0	1	1
星期四	1	0	0
星期五	1	0	1
星期六	1	1	0

表 14 月/世纪寄存器位描述 (地址 07H)

Bit	符号	描 述
7	C	世纪位; C=0 指定世纪数为 20××, C=1 指定世纪数为 19××, “××” 为年寄存器中的值, 参见表 16。当年寄存器中的值由 99 变为 00 时, 世纪位会改变
6~5	—	无用
4~0	<月>	代表 BCD 格式的当前月份, 值为 01~12; 参见表 15

表 15 月分配表

月份	Bit4	Bit3	Bit2	Bit1	Bit0
一月	0	0	0	0	1
二月	0	0	0	1	0
三月	0	0	0	1	1
四月	0	0	1	0	0
五月	0	0	1	0	1
六月	0	0	1	1	0
七月	0	0	1	1	1
八月	0	1	0	0	0
九月	0	1	0	0	1
十月	1	0	0	0	0
十一月	1	0	0	0	1
十二月	1	0	0	1	0

表 16 年寄存器位描述 (地址 08H)

Bit	符号	描 述
7~0	<年>	代表 BCD 格式的当前年数值, 值为 00~99



看了这么多条条框框的寄存器描述,大家一定有些累了,下面我们来电实用的,和大家简单说说如何使用这个芯片初始设置或读出年月日时分秒等信息。

我们将并且只能够使用 IIC 接口来读写这个芯片的各个寄存器, IIC 接口有一定的协议,需要按照协议规定送起始位、器件地址、读写寄存器地址、读写数据、停止位等,这个内容我们下一小节的设计中详细探讨,后面我们先抛开 IIC 具体读写控制时序,先从宏观角度来把该读写哪些寄存器这码事理清楚。

正常来说,一个芯片的使用,无外乎设置一下控制寄存器,然后读写相关数据,必要的话产生一个中断,此时可能回去看看状态寄存器。不过,我们这颗 RTC 更简单,地址 0x00 和 0x01 的控制寄存器 1 和 2 默认状态即可,我们只需要读写时间便可,其他什么报警、中断等功能留待大家有兴趣自己琢磨去。好,那么简单又是怎么操作的,不急,精彩马上送到。

地址 0x02~0x08 寄存器的内容是秒、分、时、日、星期、月、年信息,我们只要操作他们便可以了。

假设现在我们就是要把这些基本的时间信息读出来,然后以我们最常规的大家都能看得懂的 10 进制方式显示出来,那么如何操作?就按下面这个步骤就好,至于原理,大家回去对照各个寄存器的定义稍微一想也就能够领会。

- ① 读地址 0x02 的秒寄存器数据 `second`, 在显示时, 十位数据为 $((second \& 0x70) \gg 4)$, 个位的数据为 $(second \& 0x0f)$ 。
- ② 读地址 0x03 的分钟寄存器数据 `minute`, 在显示时, 十位数据为 $((minute \& 0x70) \gg 4)$, 个位的数据为 $(minute \& 0x0f)$ 。
- ③ 读地址 0x04 的小时寄存器数据 `hour`, 在显示时, 十位数据为 $((hour \& 0x30) \gg 4)$, 个位的数据为 $(hour \& 0x0f)$ 。
- ④ 读地址 0x05 的日寄存器数据 `day`, 在显示时, 十位数据为 $((day \& 0x30) \gg 4)$, 个位的数据为 $(day \& 0x0f)$ 。
- ⑤ 读地址 0x06 的星期寄存器数据 `week`, 在显示时, 数据为 $(week \& 0x07)$ 。
- ⑥ 读地址 0x07 的月份寄存器数据 `month`, 在显示时, 十位数据为 $((month \& 0x10) \gg 4)$, 个位的数据为 $(month \& 0x0f)$ 。
- ⑦ 读地址 0x08 的年寄存器数据 `year`, 在显示时, 十位数据为 $((year \& 0xf0) \gg 4)$, 个位的数据为 $(year \& 0x0f)$ 。

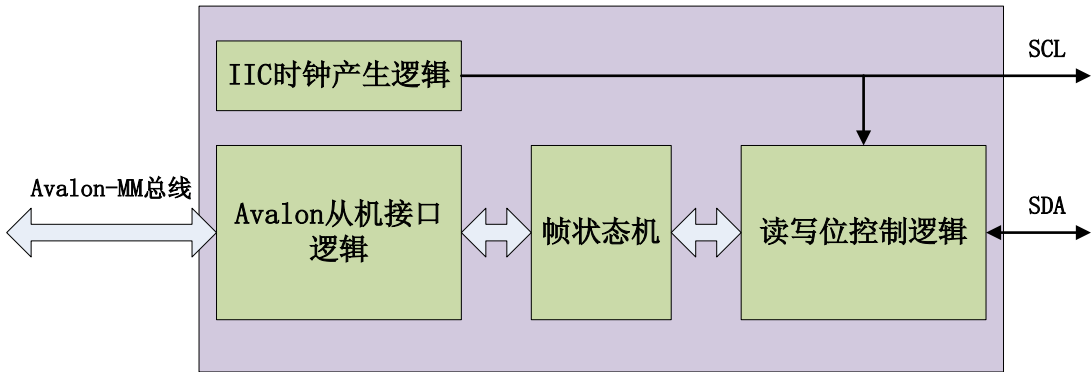
以上这些寄存器值,如果我们希望重设,直接往里面写数据即可,这样我们便可以调整



当前的时间和实际时间同步，因为我们芯片是由 3V 的纽扣电池供电的，所以即便我们的电路板下电后，芯片内部的时间计数单元还是在正常工作运转的。

8.3.2 IIC 控制器组件设计

在本实例中，为了能够对 RTC 芯片进行读写，我们必须使用代码设计一个 IIC 控制器。虽然 Qsys 的自带组件库中没有 IIC 组件，但是我们可以参考 SPI 或者 UART 外设的一些常用驱动方式，来创建一个 IIC 控制器组件。这个组件的内部功能框图如图所示。



通过 Avalon-MM 总线，可以读写以下 4 个寄存器。

地址	读/写状态	功能
0	读/写	Bit7-0: 数据寄存器。
1	读/写	控制/状态寄存器。 Bit0: IIC 写操作使能位，1--发起一次 IIC 写操作。 Bit1: IIC 读操作使能位，1--发起一次 IIC 读操作。 Bit2: IIC 写操作中断屏蔽位，1—IIC 写操作完成后产生中断，0—IIC 写操作完成后不产生中断。 Bit3: IIC 读操作中断屏蔽位，1—IIC 读操作完成后产生中断，0—IIC 读操作完成后不产生中断。 Bit4: IIC 写操作中断，1—IIC 写操作完成，0—IIC 写操作执行中（bit2=0 则此位始终为 0）。 Bit5: IIC 读操作中断，1—IIC 读操作完成，0—IIC 读操作执行中（bit3=0 则此位始终为 0）。
2	只写	Bit7-0: IIC 读操作器件地址寄存器。

《圣经》箴言九 11 “敬畏耶和华是智慧的开端，认识至胜者便是聪明。”



3	只写	Bit7-0: IIC 写操作器件地址寄存器。
4	只写	Bit7-0: IIC 数据读写地址。

我们设计的这个 IIC 控制器,实际上驱动速率已经固定,而且只能读写一个字节的数据,只能说是个简化版的 IIC 控制器。在实际 NIOS II 上软件编程控制是,只要对相关寄存器做配置即可,无须再去管 IIC 总线协议那些复杂的时序了。当然了,想了解 IIC 协议具体实现时序的童鞋,还要好好消化下后面的代码,毕竟这个还是最优技术含量的底层代码。

另外,下面我们将给出读 IIC 总线执行读或写操作的软件配置各个寄存器的流程。

读一个 IIC 地址寄存器的数据:

- ① IIC 读操作器件地址寄存器(地址: 2)配置。
- ② IIC 数据读写地址(地址: 4)配置。
- ③ 控制/状态寄存器(地址: 1)配置,拉高 IIC 读操作中断屏蔽位,同时拉高 IIC 读操作使能位。
- ④ 等待控制/状态寄存器(地址: 1)的 IIC 读操作中断位拉高。
- ⑤ 将控制/状态寄存器(地址: 1)的 IIC 读操作中断位清零。
- ⑥ 读取数据寄存器(地址: 0)。

写一个 IIC 地址寄存器的数据:

- ① IIC 读操作器件地址寄存器(地址: 2)配置。
- ② IIC 写操作器件地址寄存器(地址 3)配置。
- ③ IIC 数据读写地址(地址: 4)配置。
- ④ 写数据寄存器(地址: 0)。
- ⑤ 控制/状态寄存器(地址: 1)配置,拉高 IIC 写操作中断屏蔽位,同时拉高 IIC 写操作使能位。
- ⑥ 等待控制/状态寄存器(地址: 1)的 IIC 写操作中断位拉高。
- ⑦ 将控制/状态寄存器(地址: 1)的 IIC 写操作中断位清零。

```
module iic_ctrl(  
    clk, rst_n,  
  
    sys_cs_n, sys_rd_n, sys_wr_n, sys_addr, sys_wrddata, sys_rddata, sys_irq,  
    scl, sda
```



```
);

input clk;          // 50MHz 主时钟
input rst_n;        //低电平复位信号

input sys_cs_n; //总线读片选, 低电平有效
input sys_rd_n; //总线读使能信号, 低电平有效
input sys_wr_n; //总线写使能信号, 低电平有效
input[2:0] sys_addr; //总线读写地址
input[7:0] sys_wrddata; //总线写入数据
output reg[7:0] sys_rddata; //总线读取数据
output sys_irq; //中断信号

output scl; //串行配置 IIC 时钟信号
inout sda; //串行配置 IIC 数据信号

//-----
//Avalon-MM 接口逻辑
reg[1:0] sys_rd_nr; //锁存两拍 sys_rd_n
reg[1:0] sys_wr_nr; //锁存两拍 sys_wr_n

always @(posedge clk or negedge rst_n)
    if(!rst_n) sys_rd_nr <= 2'b11;
    else sys_rd_nr <= {sys_rd_nr[0], sys_rd_n};

always @(posedge clk or negedge rst_n)
    if(!rst_n) sys_wr_nr <= 2'b11;
    else sys_wr_nr <= {sys_wr_nr[0], sys_wr_n};

wire sys_rd_neg = sys_rd_nr[1] & ~sys_rd_nr[0]; //sys_rd_n 下降沿标志位
wire sys_wr_pos = ~sys_wr_nr[1] & sys_wr_nr[0]; //sys_wr_n 上升沿标志位

wire[7:0] tiic_rddb; //需要通过 IIC 接口配置 ADV7180 的读出数据

always @(posedge clk or negedge rst_n) //reg0 读出 IIC 数据
    if(!rst_n) sys_rddata <= 8'd0;
    else if(sys_rd_neg) begin
        case(sys_addr)
```



```
        3'd0: sys_rddata <= tiic_rddb;
        3'd1: sys_rddata <= {2'd0, tiic_ctrl};
        default: ;
    endcase
end

reg[7:0] tiic_wrdb; //IIC 写数据寄存器
reg[5:0] tiic_ctrl; //IIC 控制/状态寄存器
reg[7:0] device_rdadd; // IIC 读操作器件地址寄存器
reg[7:0] device_wradd; //写操作器件地址寄存器
reg[7:0] tiic_ab; //IIC 数据读写地址
reg tiic_wr_rdn; //IIC 数据读或写选择, 1--写, 0--读

always @(posedge clk or negedge rst_n)
    if(!rst_n) begin
        tiic_wrdb <= 8'd0;
        tiic_ctrl <= 6'd0;
        device_rdadd <= 8'd0;
        device_wradd <= 8'd0;
        tiic_ab <= 8'd0;
    end
    else if(sys_wr_pos) begin
        case(sys_addr)
            3'd0: tiic_wrdb <= sys_wrdata;
            3'd1: tiic_ctrl <= sys_wrdata[5:0];
            3'd2: device_rdadd <= sys_wrdata;
            3'd3: device_wradd <= sys_wrdata;
            3'd4: tiic_ab <= sys_wrdata;
            default: ;
        endcase
    end
    else begin
        tiic_ctrl[1:0] <= 2'b00; //IIC 读写操作使能一次后立即拉低
        if(iic_ack) begin
            if(tiic_wr_rdn && tiic_ctrl[2]) tiic_ctrl[4] <= 1'b1; //写完成中
            断
            else if(!tiic_wr_rdn && tiic_ctrl[3]) tiic_ctrl[5] <= 1'b1; //读完成
            中断
        end
    end
end
```



```
        end
    end

assign sys_irq = tiic_ctrl[4] | tiic_ctrl[5];    //中断信号

always @(posedge clk or negedge rst_n)
    if(!rst_n) tiic_wr_rdn <= 1'b0;
    else if(tiic_ctrl[0]) tiic_wr_rdn <= 1'b1;    //写
    else if(tiic_ctrl[1]) tiic_wr_rdn <= 1'b0;    //读

reg tiic_en;    //IIC 数据读写使能信号

always @(posedge clk or negedge rst_n)
    if(!rst_n) tiic_en <= 1'b0;
    else if(tiic_ctrl[1] || tiic_ctrl[0]) tiic_en <= 1'b1;
    else tiic_en <= 1'b0;

//-----
//IIC 控制读写状态机

//IIC 时钟信号 scl 产生逻辑
reg[8:0] icnt;    //分频计数寄存器, 50M/95K=512

always @(posedge clk or negedge rst_n)
    if(!rst_n) icnt <= 9'd0;
    else icnt <= icnt+1'b1;

wire scl = ~icnt[8];    //0<=icnt<50 时 scl=1; 50<=icnt<100 时 scl=0
wire scl_hs = (icnt == 9'd1);    //scl high start
wire scl_hc = (icnt == 9'd128);    //scl high center
wire scl_ls = (icnt == 9'd256);    //scl low start
wire scl_lc = (icnt == 9'd384);    //scl low center

//IIC 读或写状态控制
parameter    DIDLE    = 4'd0,    //idle
             DSTAR    = 4'd1,    //start transfer
             DSABW    = 4'd2,    //slave addr (write cmd)
```



```
D1ACK    = 4'd3, //ACK1
DRABW    = 4'd4, //device addr write
D2ACK    = 4'd5, //ACK2
/*wr data*/ DWRDB = 4'd6, //write data
D3ACK    = 4'd7, //ACK3
/*rd data*/ DRSTA  = 4'd8, //restart transfer
DSABR    = 4'd9, //slave addr (read cmd)
D4ACK    = 4'd10, //ACK4
DRDDB    = 4'd11, //read data
D5ACK    = 4'd12, //ACK5
DSTOP    = 4'd13; //stop transfer

//parameter DEVICE_WRADD    = 8'ha2,    //write device addr
//      DEVICE_RDADD    = 8'ha3;    //read device addr

//IIC 状态机控制信号
reg iicwr_req; //IIC 写请求信号, 高电平有效
reg iicrd_req; //IIC 读请求信号, 高电平有效
wire[7:0] cmd_addr = tiic_ab;
wire[7:0] iic_wrdb = tiic_wrdb;
reg[7:0] iic_rddb; //IIC 读数据寄存器

reg[2:0] bcnt; //数据位寄存器, bit0-7
reg sdar; //sda 输出数据寄存器
reg sdalink; //sda 方向控制寄存器, 0--input, 1--output
reg[3:0] dcstate, dnstate;

//当前和下一状态切换
always @(posedge clk or negedge rst_n)
    if(!rst_n) dnstate <= DIDLE;
    else dnstate <= dcstate;

//状态变迁
always @(dnstate or iicwr_req or iicrd_req or scl_hc or bcnt or scl_ls or scl_hs
or scl_lc) begin
    case(dnstate)
        DIDLE: if((iicwr_req || iicrd_req) && scl_hs) dcstate <= DSTAR;    //
发出读或写 IIC 请求
```



```
        else dcstate <= DIDLE;
DSTAR:  if(scl_ls) dcstate <= DSABW;
        else dcstate <= DSTAR;
DSABW:  if(scl_lc && (bcnt == 3'd0)) dcstate <= D1ACK;
        else dcstate <= DSABW; //slave addr (write cmd)
D1ACK:  if(scl_ls && (bcnt == 3'd7)) dcstate <= DRABW;
        else dcstate <= D1ACK;
DRABW:  if(scl_lc && (bcnt == 3'd0)) dcstate <= D2ACK;
        else dcstate <= DRABW; //device addr write
D2ACK:  if(scl_ls && (bcnt == 3'd7) && iicwr_req) dcstate <= DWRDB; //
写数据
        else if(scl_ls && (bcnt == 3'd7) && iicrd_req) dcstate <= DRSTA;
//读数据
        else dcstate <= D2ACK;
/*wr_db*/DWRDB: if(scl_lc && (bcnt == 3'd0)) dcstate <= D3ACK;
        else dcstate <= DWRDB; //write data
D3ACK:  if(scl_ls && (bcnt == 3'd7)) dcstate <= DSTOP;
        else dcstate <= D3ACK;
/*rd_db*/DRSTA: if(scl_ls) dcstate <= DSABR;
        else dcstate <= DRSTA;
DSABR:  if(scl_lc && (bcnt == 3'd0)) dcstate <= D4ACK;
        else dcstate <= DSABR; //slave addr (read cmd)
D4ACK:  if(scl_ls && (bcnt == 3'd7)) dcstate <= DRDDB;
        else dcstate <= D4ACK;
DRDDB:  if(scl_hc && (bcnt == 3'd7)) dcstate <= D5ACK;
        else dcstate <= DRDDB; //read data
D5ACK:  if(scl_ls && (bcnt == 3'd6)) dcstate <= DSTOP;
        else dcstate <= D5ACK;
DSTOP:  if(scl_ls) dcstate <= DIDLE;
        else dcstate <= DSTOP;
default: dcstate <= DIDLE;
endcase
end

//数据位寄存器控制
always @(posedge clk or negedge rst_n)
    if(!rst_n) bcnt <= 3'd0;
    else begin
```



```
        case(dnstate)
            DIDLE: bcnt <= 3'd7;
            DSABW, DRABW, DWRDB, DSABR: if(scl_hs) bcnt <= bcnt-1'b1;
            DRDDB: if(scl_hs) bcnt <= bcnt-1'b1;
            D1ACK, D2ACK, D3ACK: if(scl_lc) bcnt <= 3'd7;
            D4ACK: if(scl_ls) bcnt <= 3'd7;
            D5ACK: if(scl_lc) bcnt <= bcnt-1'b1;
            default: ;
        endcase
    end

    //IIC 数据输入输出控制
always @(posedge clk or negedge rst_n)
    if(!rst_n) begin
        sdar <= 1'b1;
        sdalink <= 1'b1;    //output
        iic_rddb <= 8'd0;
    end
    else begin
        case(dnstate)
            DIDLE: begin
                sdar <= 1'b1;
                sdalink <= 1'b1;    //output
            end
            DSTAR: begin
                if(scl_hc) sdar <= 1'b0;
            end
            DSABW: begin
                if(scl_lc) sdar <= device_wradd[bcnt];
            end
            D1ACK: begin
                if(scl_lc) begin
                    sdar <= 1'b1;
                    sdalink <= 1'b0;
                end
            end
            DRABW: begin
                if(scl_lc) begin
```




```
        sdar <= cmd_addr[bcnt];
        sdalink <= 1'b1;
    end

    end

D2ACK: begin
    if(scl_lc) begin
        sdar <= 1'b1;
        sdalink <= 1'b0;
    end

    end

/*wr_db*/DWRDB: begin
    if(scl_lc) begin
        sdar <= iic_wrdb[bcnt];
        sdalink <= 1'b1;
    end

    end

D3ACK: begin
    if(scl_lc) begin
        sdar <= 1'b1;
        sdalink <= 1'b0;
    end

    end

/*rd_db*/DRSTA: begin
    if(scl_hc) sdar <= 1'b0;
    else if(scl_lc) begin
        sdar <= 1'b1;
        sdalink <= 1'b1;
    end

    end

    end

DSABR: begin
    if(scl_lc) sdar <= device_rdadd[bcnt];
    end

D4ACK: begin
    if(scl_lc && (bcnt == 3'd7)) sdalink <= 1'b0;    //input
    end

DRDDB: begin
    if(scl_hc) iic_rddb[bcnt+1'b1] <= sda;
    sdar <= 1'b1;
end
```



```
        end
    D5ACK: begin
        if(scl_lc) begin
            sdar <= 1'b0;
            sdalink <= 1'b1;
        end
    end
    DSTOP: begin
        if(scl_lc) begin
            sdalink <= 1'b1;    //output
            sdar <= 1'b0;
        end
        else if(scl_hc) sdar <= 1'b1;
    end
    default: ;
    endcase
end

assign sda = sdalink ? sdar : 1'bz;
wire iic_ack = (dnstate == DSTOP) && scl_hs;    //IIC 操作响应, 高电平有效
assign tiic_rddb = iic_rddb;

//-----
//简单的 IIC 控制指令发送和接收
parameter  WIRST  = 4'd0,
           WAODWR  = 4'd1, //write 8'h0d addr
           WIDLE   = 4'd2,
           WA0ORD  = 4'd3; //read 8'h00 addr

reg[3:0] wcstate,wnstate;

//工作状态迁移
always @(posedge clk or negedge rst_n)
    if(!rst_n) wnstate <= WIRST;
    else wnstate <= wcstate;

//状态控制
always @(wnstate or tiic_en or tiic_wr_rdn or iic_ack) begin
```



```
case(wnstate)
    WIRST: if(tiic_en) begin
        if(tiic_wr_rdn) wcstate <= WAODWR;
        else wcstate <= WAOORD;
    end
    else wcstate <= WIRST;
    WAODWR: if(iic_ack) wcstate <= WIRST;
    else wcstate <= WAODWR;
    WAOORD: if(iic_ack) wcstate <= WIRST;
    else wcstate <= WAOORD;
    default: wcstate <= WIRST;
endcase
end

//IIC 读写请求信号产生
always @(posedge clk or negedge rst_n)
    if(!rst_n) begin
        iicwr_req <= 1'b0;
        iicrd_req <= 1'b0;
    end
    else begin
        case(wnstate)
            WIRST: if(tiic_en) begin
                if(tiic_wr_rdn) begin
                    iicwr_req <= 1'b1;
                    iicrd_req <= 1'b0;
                end
                else begin
                    iicwr_req <= 1'b0;
                    iicrd_req <= 1'b1;
                end
            end
            else begin
                iicwr_req <= 1'b0;
                iicrd_req <= 1'b0;
            end
        end
        WAODWR: if(iic_ack) begin
            iicrd_req <= 1'b0;
        end
    end
end
```



```
        iicwr_req <= 1'b0;
    end
    else begin
        iicwr_req <= 1'b1;
        iicrd_req <= 1'b0;
    end
    WA00RD: if(iic_ack) begin
        iicrd_req <= 1'b0;
        iicwr_req <= 1'b0;
    end
    else begin
        iicwr_req <= 1'b0;
        iicrd_req <= 1'b1;
    end
    default: begin
        iicwr_req <= 1'b0;
        iicrd_req <= 1'b0;
    end
endcase
end

endmodule
```

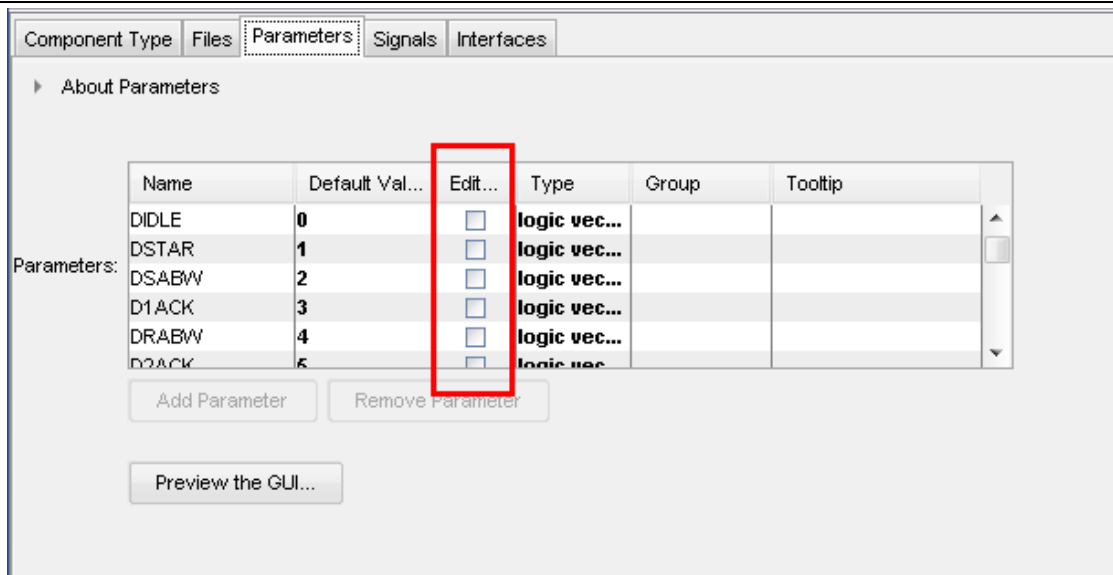
8.3.3 Qsys 系统构建

我们首先 copy 上一个工程 ex17 的文件夹, 然后重命名为 ex18, 再打开工程, 进入 Qsys 页面。进入 Component Editor 页面 (怎么, 你还不会啊, 赶快面壁思过, 到前面的章节反省一下), 在 Component Type 页面, 设置 Name 和 Display Name 均为 iic_controller。



在 Files 页面, 先添加 iic_ctrl.v 文件, 然后点击 Analyze Synthesis Files 进行综合, 通过综合后, 将 iic_ctrl 设置为 Top-level Module。

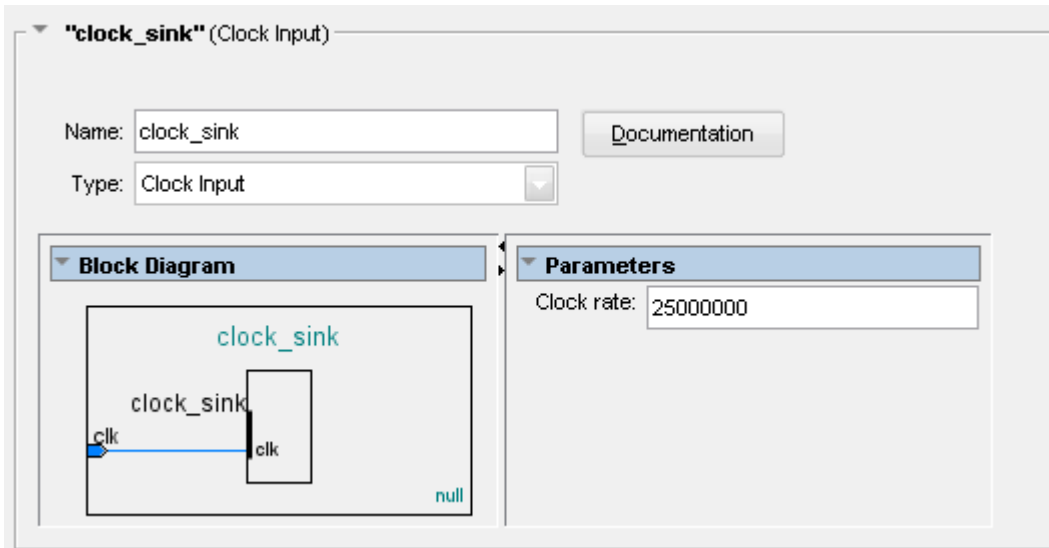
Parameters 页面罗列了我们代码中的所有 parameter 定义, 由于我们不需要这些参数在添加组件时是可调的, 所以将 Edit 一系列的勾选全部取消。



Signal 页面，我们一共有 5 大类的 interface，具体大家可以参考代码中对这些信号的注释描述以及在实际逻辑中的功能。

Name	Interface	Signal Type	Width	Direction
clk	clock_sink	clk	1	input
rst_n	reset_sink	reset_n	1	input
sys_cs_n	avalon_slave	chipselect_n	1	input
sys_rd_n	avalon_slave	read_n	1	input
sys_wr_n	avalon_slave	write_n	1	input
sys_addr	avalon_slave	address	3	input
sys_wrddata	avalon_slave	writedata	8	input
sys_rddata	avalon_slave	readdata	8	output
sys_irq	interrupt_sender	irq	1	output
scl	conduit_end	export	1	output
sda	conduit_end	export	1	bidir

接下来是各个 Interfaces 的具体设置。





"reset_sink" (Reset Input)

Name: [Documentation](#)

Type:

Associated Clock:

Block Diagram

```
graph LR; rst_n --> reset_sink[reset_sink]; reset_sink --> reset_n[reset_n]; reset_n --> null[null];
```

Parameters

Associated clock:

Synchronous edges:

"avalon_slave" (Avalon Memory Mapped Slave)

Name: [Documentation](#)

Type:

Associated Clock:

Associated Reset:

Block Diagram

```
graph LR; sys_cs_n --> avalon_slave[avalon_slave]; sys_rd_n --> avalon_slave; sys_wr_n --> avalon_slave; sys_addr[2..0] --> avalon_slave; sys_wrddata[7..0] --> avalon_slave; sys_rddata[7..0] --> avalon_slave; avalon_slave --> chipselect_n[chipselect_n]; avalon_slave --> read_n[read_n]; avalon_slave --> write_n[write_n]; avalon_slave --> address[address]; avalon_slave --> writedata[writedata]; avalon_slave --> readdata[readdata]; chipselect_n --> null[null]; read_n --> null; write_n --> null; address --> null; writedata --> null; readdata --> null;
```

Parameters

Timing

Setup:	<input type="text" value="1"/>
Read wait:	<input type="text" value="5"/>
Write wait:	<input type="text" value="4"/>
Hold:	<input type="text" value="3"/>
Timing units:	<input type="text" value="Cycles"/>

Pipelined Transfers

Read Waveforms

clk, read_n, write_n, chipselect_n, address (A0), readdata (D0)

Write Waveforms

clk, read_n, write_n, chipselect_n, address (A0), writedata (D0)

Deprecated



"interrupt_sender" (Interrupt Sender)

Name: [Documentation](#)

Type:

Associated Clock:

Associated Reset:

Block Diagram

Parameters

Associated addressable interface:

Waveform

"conduit_end" (Conduit)

Name: [Documentation](#)

Type:

Associated Clock:

Associated Reset:

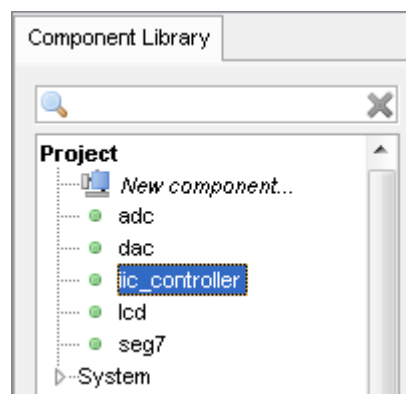
Block Diagram

Parameters

associatedClock:

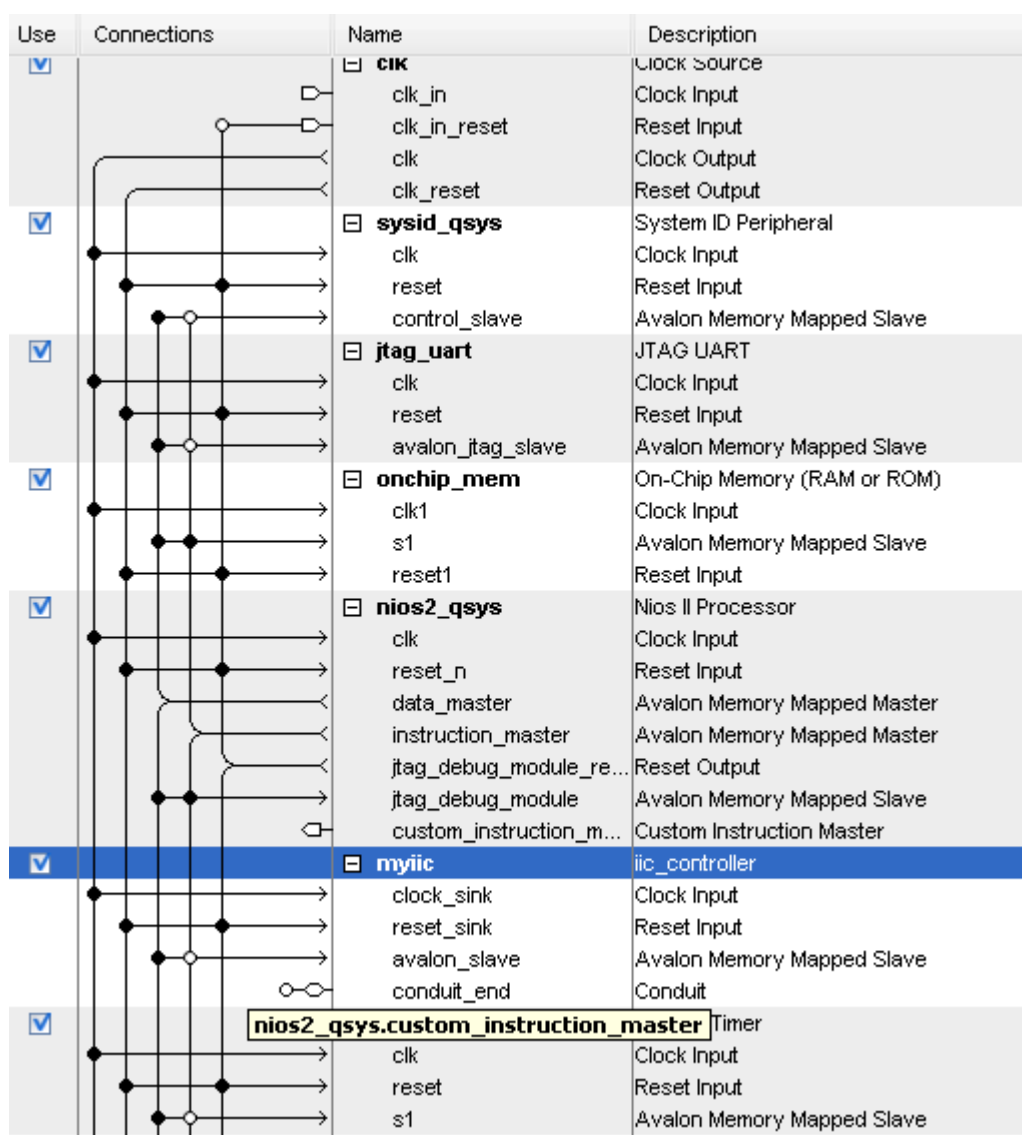
associatedReset:

完成设置后, 我们看到 Component Library 中出现了新的组件 iic_controller, 双击它添加到 Qsys 中。





将新组件更名为 **myiic**，它的系统互连如图所示。



同时注意右侧的 **IRQ** 一列，点击空心圆开启 **IRQ** 功能。最后是重新分配地址。然后 **Generate** 整个系统。回到 **Quartus II** 的顶层文件，只需要添加以下的 **IIC** 接口即可。

```
module (
    .....
    scl, sda
);
.....
//FPGA 与 RTC 芯片 IIC 接口
output scl; //串行配置 IIC 时钟信号
inout sda; //串行配置 IIC 数据信号
.....
```



```
//-----
//Qsys 系列例化
myqsys u0 (
    .....

    .myiic_conduit_end_scl(scl),
        //.sda
    .myiic_conduit_end_sda(sda)
);
.....
endmodule
```

可以先对系统进行综合，然后进入管脚分配页面，对添加的两个 IIC 信号做如下分配。

Node Name	Direction	Location	I/O Bank
lcd_hsy	Output	PIN_65	4
lcd_vsy	Output	PIN_64	4
rst_n	Input	PIN_91	6
scl	Output	PIN_104	6
sda	Bidir	PIN_103	6
sdr	Output	PIN_46	3
sdr	Output	PIN_135	8
sdr	Output	PIN_49	3
sdr	Output	PIN_50	3
sdr	Output	PIN_51	3

最后,对整个工程进行编译,然后将硬件 sof 文件下载到 SF-CY3 板中,等待软件的到来。

8.3.4 软件工程实例

以 ex18 的硬件为基础，创建软件工程，命名为 ex18swprj，新建 main.c 文件。基本上，我们的软件也是以 ex17 为基础，增加一些新功能。我们这个实例，将会对 RTC 芯片进行操作，将时间信息实时显示到液晶屏上。

首先，我们将上一个软件例程中的 ASIIC 码和 GB2312 码的显示整合到了一个函数中，这样无论是 ASCII 码还是 GB2312 码，一个字符串混合着送过来也能够招架住了。

```

/*****
函数名: lcd_print
功 能: 在指定位置显示 8*16 的字符串（GB2312 或 ASCII 码均可）
输 入: alt_ul6 Row:      字符显示起始 x 坐标
        alt_ul6 Col:      字符显示起始 y 坐标
*****/
```



```
    alt_u8 *ptr:    写入的字符串
    alt_u16 Cor_b0: 前景色彩
    alt_u16 Cor_q0: 背景色彩

返 回: 无

*****/
void lcd_print(alt_u16 uRow, alt_u16 uCol, alt_u8 *ptr, alt_u16 Cor_b0, alt_u16
Cor_q0)
{
    alt_u16 uLen=0; //字符串长度
    alt_u16 i, j, k=0;

    while ((alt_u8)ptr[uLen] >= 0x10)    {uLen++;};        //探测字符串长度

    while(k < uLen) //写入 uLen 个 ASCII 字符
    {
        if(ptr[k] <= 128)        //ASCII 码
        {
            if(ptr[k] >= 0x10)
            {
                ascii_read(ptr[k]); //读出 ASCII 码字模数据
                for(j=0; j<16; j++)    //显示 8×16 的 ASCII 码, 分 16 行输出
                {
                    for(i=0; i<8; i++)
                    {
                        if((char_db[j] & (1<<(7-i))) != 0x00)
lcd_wrdp(uRow+i, uCol+j, Cor_q0); //送像素点前景色
                        else lcd_wrdp(uRow+i, uCol+j, Cor_b0);    //送像素点背景
                        色
                    }
                }
            }
        }
        uRow+=8;                // x 坐标加 8, 即下一个字符位
        k++;    //下一个字符
    }
    else    //GB2312
    {
        gb2312_read(ptr[k], ptr[k+1]);    //读出 ASCII 码字模数据
        for(j=0; j<16; j++)    //显示 15×16 的 GB2312 字符, 分 16 行输出
```



```
{
    for(i=0;i<8;i++)
    {
        if((char_db[(j<<1)] & (1<<(7-i))) != 0x00)
        lcd_wrdp(uRow+i,uCol+j,Cor_q0); //送像素点前景色
        else lcd_wrdp(uRow+i,uCol+j,Cor_b0); //送像素点背景色
    }
    for(i=0;i<8;i++)
    {
        if((char_db[(j<<1)+1] & (1<<(7-i))) != 0x00)
        lcd_wrdp(uRow+8+i,uCol+j,Cor_q0); //送像素点前景色
        else lcd_wrdp(uRow+8+i,uCol+j,Cor_b0); //送像素点背景色
    }
    }
    uRow+=16; // x 坐标加 15, 即下一个字符位
    k+=2; //下一个字符
}

}
```

下面这个函数读出 **RTC** 芯片的时间信息, 然后缓存到一个数组 **time[]** 中。该函数会返回一个变量, 指示当前读出的 **RTC** 秒和上一次读出的是否一致, 主要是方便主函数中确定是否更新液晶屏上的显示。大家可以对照前面给出的编程顺序, 基本是一致的。

```
/******
* 函数名: rtc_read
* 功 能: RTC 芯片读出时间信息
* 入 口: 无
* 出 口: time[]—将相应的时间信息填充到该数组中
*      return 1—秒更新;0—秒保持
*****/
alt_u8 rtc_read(void)
{
    alt_u8 temp;
    alt_u8 d[2];
    alt_u8 return_db = 0;
    alt_u8 state;
```



```
IOWR_8DIRECT(MYIIC_BASE, 2, 0xa3);    //IIC 读操作器件地址寄存器
IOWR_8DIRECT(MYIIC_BASE, 3, 0xa2);    //IIC 写操作器件地址寄存器

//second
IOWR_8DIRECT(MYIIC_BASE, 4, 0x2);      //IIC 数据读写地址
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0e);     //发起读操作
do {
    state = IORD_8DIRECT(MYIIC_BASE, 1);
} while(!(state&0x20));
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0c);     //清中断标志位
d[0] = time[22];
d[1] = time[23];
temp = IORD_8DIRECT(MYIIC_BASE, 0);
time[22] = ((temp&0x70)>>4)+0x30;
time[23] = (temp&0x0f)+0x30;
if((d[0] != time[22]) || (d[1] != time[23])) return_db = 1;

//minute
IOWR_8DIRECT(MYIIC_BASE, 4, 0x3);      //IIC 数据读写地址
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0e);     //发起读操作
do {
    state = IORD_8DIRECT(MYIIC_BASE, 1);
} while(!(state&0x20));
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0c);     //清中断标志位
temp = IORD_8DIRECT(MYIIC_BASE, 0);
time[19] = ((temp&0x70)>>4)+0x30;
time[20] = (temp&0x0f)+0x30;

//hour
IOWR_8DIRECT(MYIIC_BASE, 4, 0x4);      //IIC 数据读写地址
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0e);     //发起读操作
do {
    state = IORD_8DIRECT(MYIIC_BASE, 1);
} while(!(state&0x20));
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0c);     //清中断标志位
temp = IORD_8DIRECT(MYIIC_BASE, 0);
time[16] = ((temp&0x30)>>4)+0x30;
time[17] = (temp&0x0f)+0x30;
```



```
//day
IOWR_8DIRECT(MYIIC_BASE, 4, 0x5);    //IIC 数据读写地址
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0e);    //发起读操作
do {
    state = IORD_8DIRECT(MYIIC_BASE, 1);
} while(!(state&0x20));
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0c);    //清中断标志位
temp = IORD_8DIRECT(MYIIC_BASE, 0);
time[10] = ((temp&0x30)>>4)+0x30;
time[11] = (temp&0x0f)+0x30;

//week
IOWR_8DIRECT(MYIIC_BASE, 4, 0x6);    //IIC 数据读写地址
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0e);    //发起读操作
do {
    state = IORD_8DIRECT(MYIIC_BASE, 1);
} while(!(state&0x20));
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0c);    //清中断标志位
temp = IORD_8DIRECT(MYIIC_BASE, 0);
time[30] = week[(temp&0x03+1)<<1];
time[31] = week[((temp&0x03+1)<<1)+1];

//month
IOWR_8DIRECT(MYIIC_BASE, 4, 0x7);    //IIC 数据读写地址
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0e);    //发起读操作
do {
    state = IORD_8DIRECT(MYIIC_BASE, 1);
} while(!(state&0x20));
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0c);    //清中断标志位
temp = IORD_8DIRECT(MYIIC_BASE, 0);
time[6] = ((temp&0x10)>>4)+0x30;
time[7] = (temp&0x0f)+0x30;

//year
IOWR_8DIRECT(MYIIC_BASE, 4, 0x8);    //IIC 数据读写地址
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0e);    //发起读操作
do {
```



```
state = IORD_8DIRECT(MYIIC_BASE, 1);
} while(!(state&0x20));
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0c);    //清中断标志位
temp = IORD_8DIRECT(MYIIC_BASE, 0);
time[2] = ((temp&0xf0)>>4)+0x30;
time[3] = (temp&0x0f)+0x30;

return return_db;
}
```

下面这个函数是对 RTC 芯片写入新数据, 也就是重设 RTC 芯片内部的时间信息。

```
/******
* 函数名: rtc_set
* 功 能: RTC 芯片设置时间信息
* 入 口: second -- 秒
*         minute -- 分钟
*         hour -- 小时
*         day -- 天
*         week -- 周
*         month -- 月
*         year -- 年
* 出 口: 无
*****/
void rtc_set(alt_u8 second, alt_u8 minute, alt_u8 hour, alt_u8 day, alt_u8 week, alt_u8
month, alt_u8 year)
{
    alt_u8 temp;
    alt_u8 state;

    IOWR_8DIRECT(MYIIC_BASE, 2, 0xa3);    //IIC 读操作器件地址寄存器
    IOWR_8DIRECT(MYIIC_BASE, 3, 0xa2);    //IIC 写操作器件地址寄存器

    //second
    IOWR_8DIRECT(MYIIC_BASE, 4, 0x2);    //IIC 数据读写地址
    IOWR_8DIRECT(MYIIC_BASE, 1, 0x0d);    //发起写操作
    IOWR_8DIRECT(MYIIC_BASE, 0, second);
    do {
        state = IORD_8DIRECT(MYIIC_BASE, 1);
    } while(!(state&0x10));
}
```



```
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0c);    //清中断标志位

//minute
IOWR_8DIRECT(MYIIC_BASE, 4, 0x3);      //IIC 数据读写地址
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0d);    //发起写操作
IOWR_8DIRECT(MYIIC_BASE, 0, minute);
do {
    state = IORD_8DIRECT(MYIIC_BASE, 1);
} while(!(state&0x10));
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0c);    //清中断标志位

//hour
IOWR_8DIRECT(MYIIC_BASE, 4, 0x4);      //IIC 数据读写地址
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0d);    //发起写操作
IOWR_8DIRECT(MYIIC_BASE, 0, hour);
do {
    state = IORD_8DIRECT(MYIIC_BASE, 1);
} while(!(state&0x10));
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0c);    //清中断标志位

//day
IOWR_8DIRECT(MYIIC_BASE, 4, 0x5);      //IIC 数据读写地址
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0d);    //发起写操作
IOWR_8DIRECT(MYIIC_BASE, 0, day);
do {
    state = IORD_8DIRECT(MYIIC_BASE, 1);
} while(!(state&0x10));
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0c);    //清中断标志位

//week
IOWR_8DIRECT(MYIIC_BASE, 4, 0x6);      //IIC 数据读写地址
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0d);    //发起写操作
IOWR_8DIRECT(MYIIC_BASE, 0, week);
do {
    state = IORD_8DIRECT(MYIIC_BASE, 1);
} while(!(state&0x10));
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0c);    //清中断标志位
```




```
//month
IOWR_8DIRECT(MYIIC_BASE, 4, 0x7);    //IIC 数据读写地址
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0d);    //发起写操作
IOWR_8DIRECT(MYIIC_BASE, 0, month);
do {
    state = IORD_8DIRECT(MYIIC_BASE, 1);
} while(!(state&0x10));
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0c);    //清中断标志位

//year
IOWR_8DIRECT(MYIIC_BASE, 4, 0x8);    //IIC 数据读写地址
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0d);    //发起写操作
IOWR_8DIRECT(MYIIC_BASE, 0, year);
do {
    state = IORD_8DIRECT(MYIIC_BASE, 1);
} while(!(state&0x10));
IOWR_8DIRECT(MYIIC_BASE, 1, 0x0c);    //清中断标志位
}
```

我们再来看看主函数。除了和上一节一样的一些常规显示外，我们会重新设置一下当前的 RTC 时间，以后 RTC 便以这个时间开始一秒一秒的往下走。在主循环中，我们会不断的读取 RTC 中的时间信息，判断若是秒发生变化，则更新 JTAG_UART 和液晶屏的显示。

```
alt_u8 time[34] = "2013 年 01 月 01 日  00:00:00  星期一"; //该数组用于缓存时间信息
alt_u8 week[14] = "一二三四五六日";

/*****
* 名 称: 主函数
* 函数名: main
* 入 口: 无
* 出 口: 无
*****/
int main(void)
{
    alt_u16 x,y;

    //清全屏为蓝色
    for(y=0;y<240;y++)
```



```
{
    for(x=0;x<320;x++)
    {
        lcd_wrdw(x,y,0xffff);
    }
}

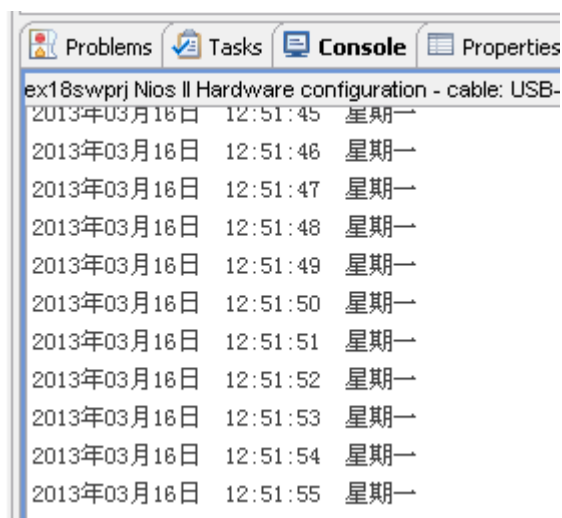
init_spi(); //初始化 SPI 的 control 和 status 寄存器

lcd_print(60,111,"http://myfpga.taobao.com/",0xffff,0xf800);
lcd_print(32,143,"特权电子开发套件带您迈入",0xffff,0x07e0);
lcd_print(180,164,"FPGA",0xffff,0x07e0);
lcd_print(212,164,"的开发大门",0xffff,0x07e0);

//BCD 码格式设置 RTC 芯片的初始时间: 秒、分、时、天、周、月、年
rtc_set(0x00,0x30,0x12,0x16,0x2,0x03,0x13);

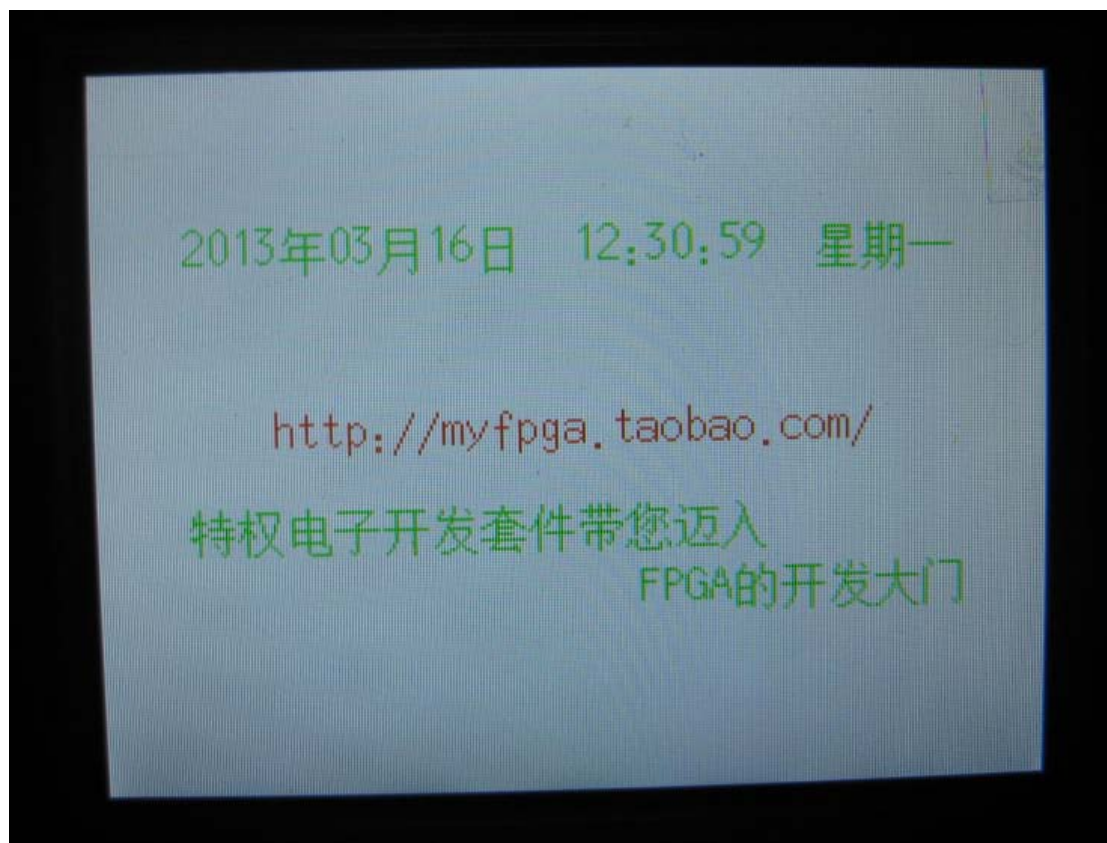
while(1)
{
    if(rtc_read())
    {
        printf("%s\n",time);
        lcd_print(32,51,time,0xffff,0x07e0);
    }
}
```

运行软件,我们可以看到在 EDS 的 Console 窗口不断的有数据更新。





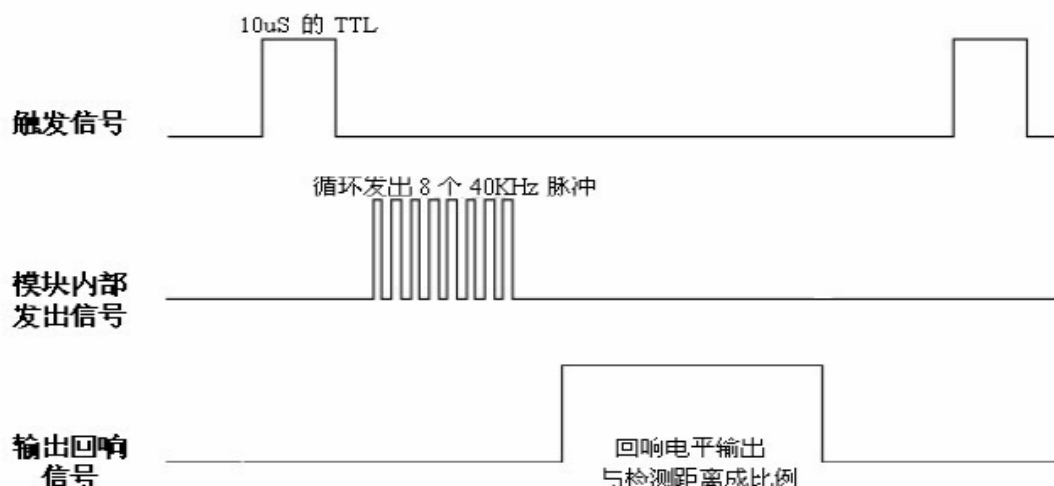
与此同时，液晶屏上的显示如下（Console 和 LCD 截图与不同的时间）。



8.4 逻辑（Verilog）实例 13——超声波测距数据采集

8.4.1 超声波模块驱动原理

超声波模块的驱动控制原理很简单。如图所示，我们用 FPGA 产生一个大于 10us 的触发信号（TRIG）给超声波模块，超声波模块内部会产生一些脉冲信号，经过内部的滤波处理，最终他反应到与 FPGA 连接的输出回响信号（ECHO）上则是一个高脉冲信号。这个高脉冲信号的宽度通过一个公式换算后就能够知道当前障碍物和模块间的距离。



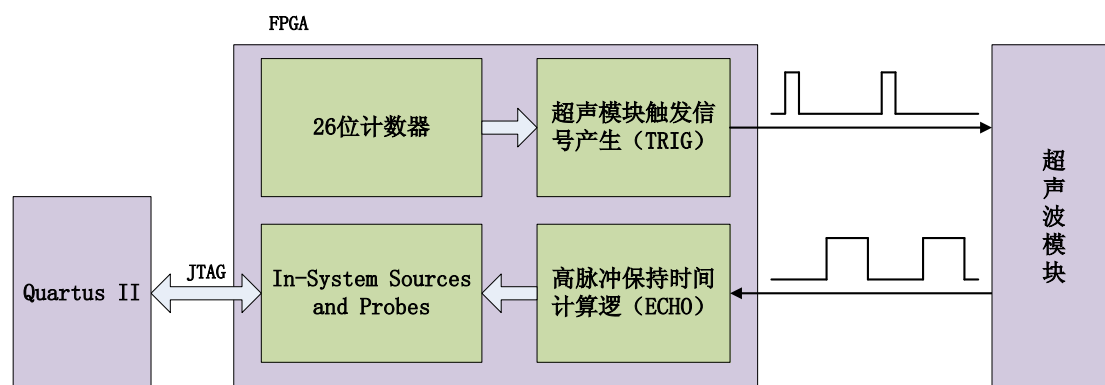
该超声波模块的有效测试距离为 2cm~400cm，测距精度可以达到 2mm。

假设超声波模块与障碍物间的距离为 S (单位: m)，ECHO 输出的高脉冲宽度为 T (单位: s)，声速定义为 340 (单位: m/s)。那么 ECHO 脉冲宽度与测试距离的关系如下。

$$S = (T * 340) / 2$$

8.4.2 数据采集平台构建

如果所示，FPGA 内部用一个 26 位的计数器循环计数，定时产生一个 10us 左右的控制超声模块触发工作的 TRIG 信号。超声波模块则每次触发后回返回一个宽度和障碍物距离远近成正比的高脉冲信号，FPGA 则用一个 24 位的计数器计算出该高脉冲信号的时钟周期数。这个高脉冲时钟周期数最终通过 FPGA 内部的 In-System Sources and Probes 传递给 JTAG，在 PC 的 Quartus II 相应调试界面中则能够实时的显示这个值。



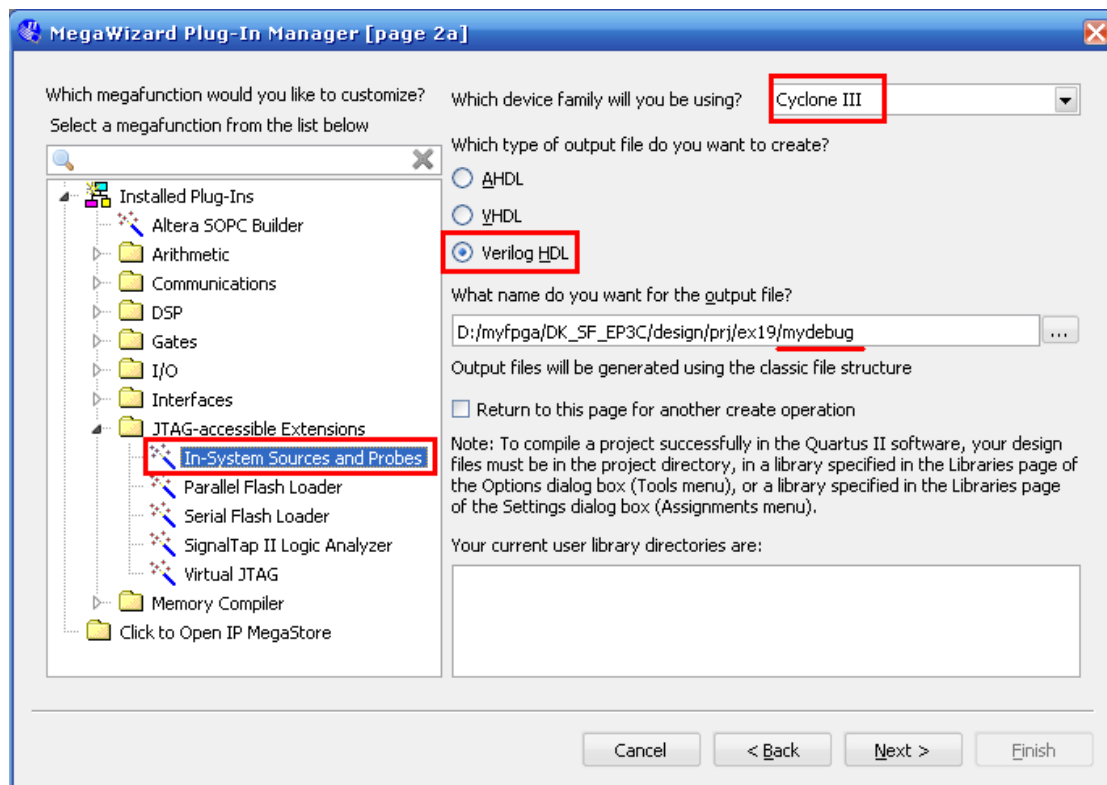
新建一个文件夹，命名为 ex19。接着打开 Quartus II，创建一个新的工程，工程名为 ex19，

工程保存到刚刚创建的 ex19 文件夹下。其他相关设置请参考 ex1 工程。

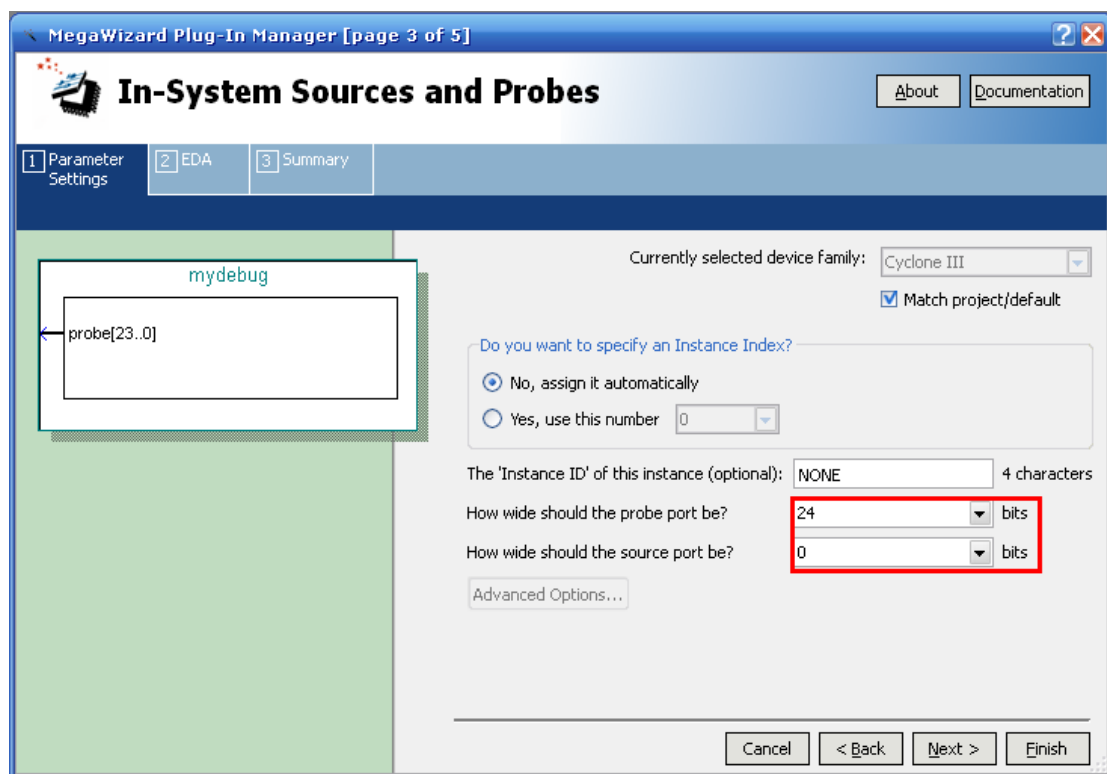
《圣经》箴言九 11 “敬畏耶和华是智慧的开端，认识至胜者便是聪明。”



接着我们直接进入 MegaWizard Plug-In Manager 新建一个 In-System Sources and Probes 模块, 命名为 mydebug。



设置这个模块的 probe port 为 24bits (和要采集的 ECHO 高脉冲的时钟计数周期数位宽相同), source port 为 0bits。





在 Quartus II 中新建一个 ex19.v 的 verilog 代码文件。输入以下设计代码。实现了前面框图所示的功能。

```
module ex19(
    clk, rst_n,
    trig, echo
);
input clk;      //25MHz
input rst_n;    //低电平复位信号

output trig;
input echo;

//-----
//每 640ms 发起一次测量
reg[25:0] cnt; //2560ms 计数器

always @(posedge clk or negedge rst_n)
    if(!rst_n) cnt <= 26'd0;
    else cnt <= cnt+1'b1;

assign trig = (cnt < 24'd26_000) ? 1'b1:1'b0; //每 640ms 有 1ms 高脉冲

//-----
//采集 echo 的高脉冲对应的时钟周期数
reg[1:0] echo_r;
reg[23:0] pulse;

always @(posedge clk or negedge rst_n)
    if(!rst_n) echo_r <= 2'b00;
    else echo_r <= {echo_r[0], echo};

always @(posedge clk or negedge rst_n)
    if(!rst_n) pulse <= 24'd0;
    else if(cnt == 26'd10) pulse <= 24'd0;
    else if(echo_r[1]) pulse <= pulse+1'b1;
    else ;
```







```
reg[23:0] pulse_r;

always @(posedge clk or negedge rst_n)
    if(!rst_n) pulse_r <= 24'd0;
    else if(cnt == 26'd1) pulse_r <= pulse;

//-----
//In-System Sources and Probes Editor 例化
mydebug mydebug_inst (
    .probe ( pulse_r ),
    .source ( )
);

endmodule
```

对工程进行综合编译，分配管脚如下。


Node Name	Direction	Location
 clk	Input	PIN_22
 echo	Input	PIN_87
 rst_n	Input	PIN_91
 trig	Output	PIN_86
<<new node>>		

编译整个工程。接着准备后面的在线调试。

8.4.3 数据采集在线调试

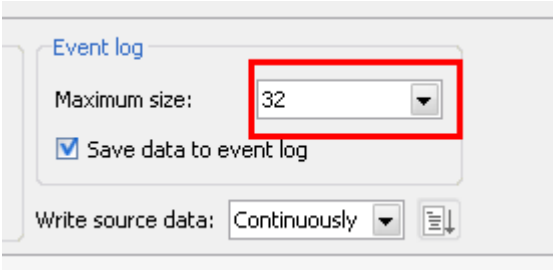
进入 In-System Sources and Probes 的在线调试界面。下载 sof 到 SF-CY3 板子中。接着选中 24bit 的 probes 信号，点击右键选择 Group，将他们归组，便于查看。



 0 NONE

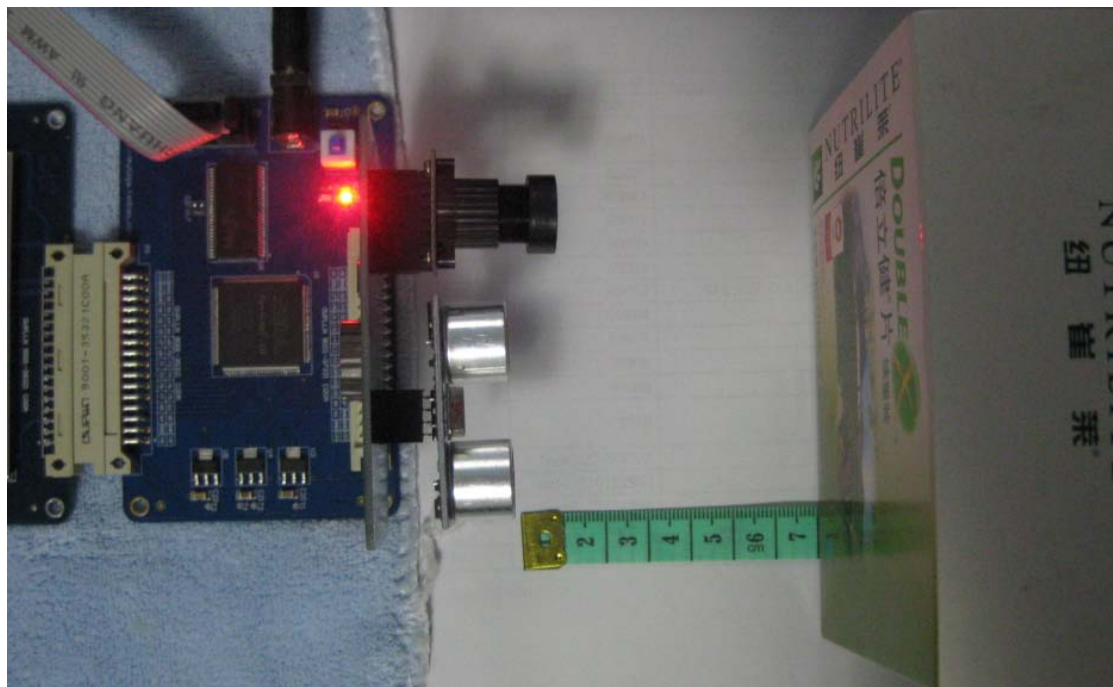
Index	Type	Alias	Name	Data	-8
P23			mydebug:mydebug_inst probe[23]	0	
P22			mydebug:myd	0	
P21			mydebug:myd	0	
P20			mydebug:myd	0	
P19			mydebug:mydebug_inst probe[19]	0	
P18			mydebug:mydebug_inst probe[18]	0	
P17			mydebug:mydebug_inst probe[17]	0	
P16			mydebug:mydebug_inst probe[16]	0	
P15			mydebug:mydebug_inst probe[15]	0	
P14			mydebug:mydebug_inst probe[14]	0	
P13			mydebug:mydebug_inst probe[13]	0	
P12			mydebug:mydebug_inst probe[12]	0	
P11			mydebug:mydebug_inst probe[11]	0	
P10			mydebug:mydebug_inst probe[10]	0	
P9			mydebug:mydebug_inst probe[9]	0	
P8			mydebug:mydebug_inst probe[8]	0	
P7			mydebug:mydebug_inst probe[7]	0	
P6			mydebug:mydebug_inst probe[6]	0	
P5			mydebug:mydebug_inst probe[5]	0	
P4			mydebug:mydebug_inst probe[4]	0	
P3			mydebug:mydebug_inst probe[3]	0	
P2			mydebug:mydebug_inst probe[2]	0	
P1			mydebug:mydebug_inst probe[1]	0	
P0			mydebug:mydebug_inst probe[0]	0	

设置 Maximum size 为 32 或更大。



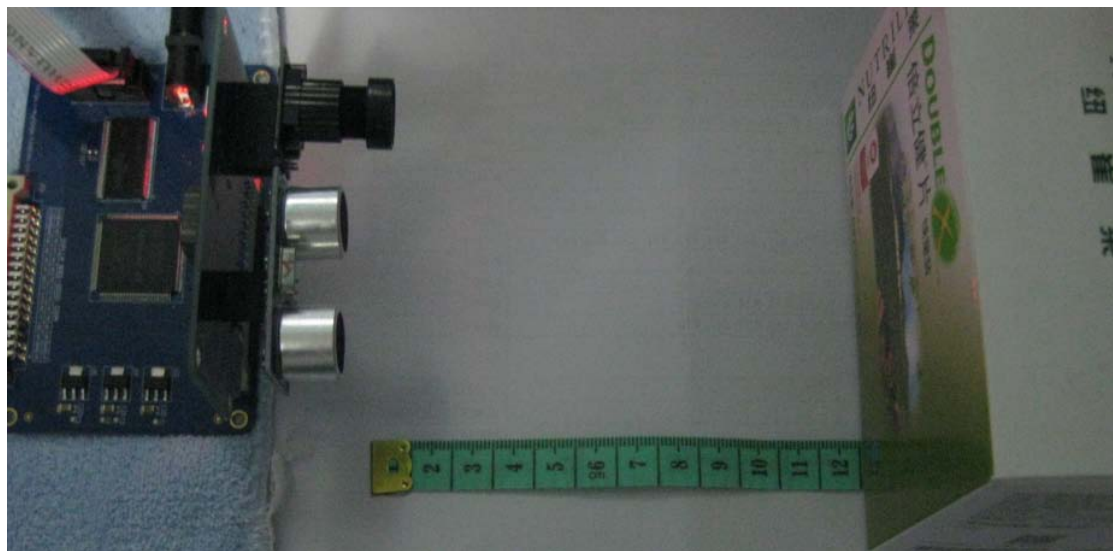
点击 RUN 开始在线调试。此时我们如果拿一本书或其他障碍物（最好是表面平整的物体），在超声波模块两个大眼睛前面来回晃动，你会发现 probes 的数据在变化，而当你停下来的时候，数据则相对稳定。没错，这就是我们采集到的回传数据，通过它我们可以换算出障碍物和超声波模块之间的距离。

下面是我们在大约 7cm 左右放置小纸盒，得到的数据时 9149，大家自己换算下，哈哈，不会算，没问题，且听下节解说。



0 NONE					-32	-31	-30	-29	-28	-27	-26	-25	-24	-23	-22
Index	Type	Alias	Name	Data											
P[23..0]			mydebug:mydebug_inst probe[23.	9150										9149	

12cm 左右放置小纸盒，得到数据时 16949。



0 NONE					-32	-31	-30	-29	-28	-27	-26	-25	-24		
Index	Type	Alias	Name	Data											
P[23..0]			mydebug:mydebug_inst probe[23.	16949							16949				



8.5 基于 Qsys 的 NIOS II 实例 10——超声波测距换算

8.5.1 超声波模块组件创建

在上一节基础上,我们将进行调试用的 In-System Sources and Probes 用 Avalon-MM 总线接口替代。此外,为了得到更实时的距离测量结果,我们也加快了测量的频率。前面已经计算过,这个超声波测距模块的有效测试距离是 2cm-400cm,它是由我们采样到的 echo 信号高脉冲的时钟周期数换算得到的,我们可以反过来,用 400cm 的情况去推断大概需要的 echo 高脉冲保持的时钟周期数。这个任务就交给大家了,我们代码中采用 20bit 的计数器肯定是难够满足最远距离测量的,接下来大家就好好消化代码吧。

```
module ultrasonic_ctrl(
    clk, rst_n,
    trig, echo,
    sys_cs_n, sys_rd_n, sys_rddata
);
input clk;        //25MHz
input rst_n;      //低电平复位信号

output trig;      //超声波模块工作触发信号
input echo;       //超声波模块反馈信号

input sys_cs_n;   //总线读片选, 低电平有效
input sys_rd_n;   //总线读使能信号, 低电平有效
output[31:0] sys_rddata; //总线读取数据

//-----
//每 40ms 发起一次测量
reg[19:0] cnt;    //40ms 计数器

always @(posedge clk or negedge rst_n)
    if(!rst_n) cnt <= 20'd0;
    else cnt <= cnt+1'b1;

assign trig = (cnt < 20'd26_000) ? 1'b1:1'b0; //每 40ms 有 1ms 高脉冲
```

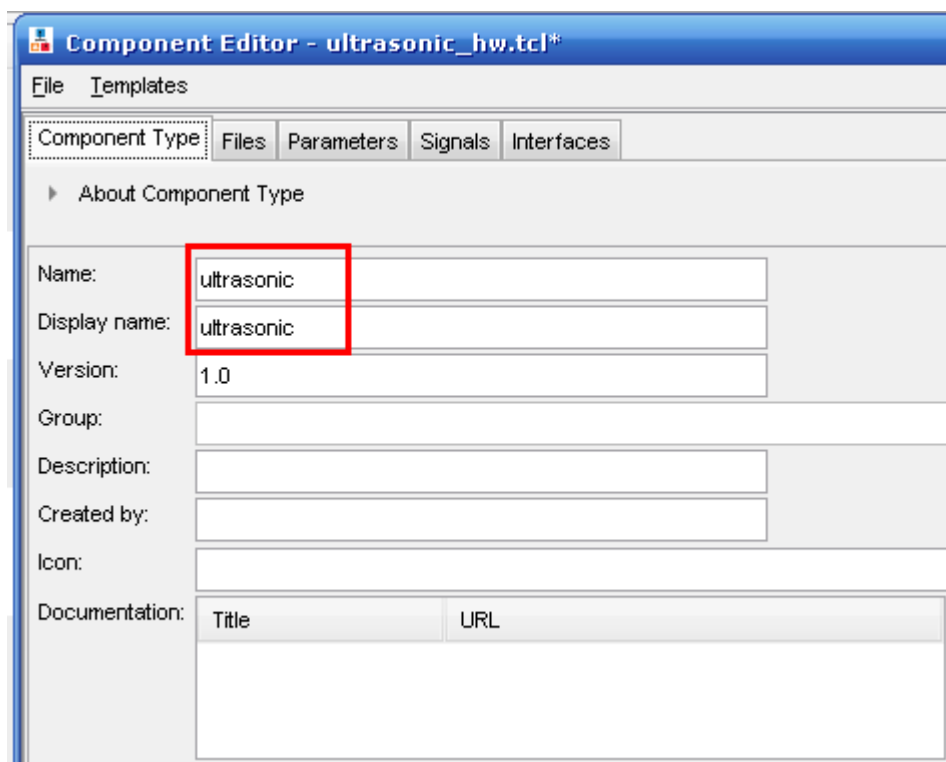


```
//-----  
//采集 echo 的高脉冲对应的时钟周期数  
reg[1:0] echo_r;  
reg[19:0] pulse;  
  
always @(posedge clk or negedge rst_n)  
    if(!rst_n) echo_r <= 2'b00;  
    else echo_r <= {echo_r[0],echo};  
  
always @(posedge clk or negedge rst_n)  
    if(!rst_n) pulse <= 20'd0;  
    else if(cnt == 20'd10) pulse <= 20'd0;  
    else if(echo_r[1]) pulse <= pulse+1'b1;  
    else ;  
  
reg[19:0] pulse_r;  
  
always @(posedge clk or negedge rst_n)  
    if(!rst_n) pulse_r <= 20'd0;  
    else if(cnt == 20'd1) pulse_r <= pulse;  
  
//-----  
//Avalon-MM 接口逻辑  
wire sys_rdcn_n = sys_cs_n | sys_rdn;  
  
reg sys_dlink; //数据输出控制, 高电平有效  
  
always @(posedge clk or negedge rst_n)  
    if(!rst_n) sys_dlink <= 1'b1;  
    else sys_dlink <= sys_rdcn_n; //锁存读片选译码  
  
assign sys_rddata = sys_dlink ? 32'hzzzzzzzz:{12'd0,pulse_r};  
  
endmodule
```

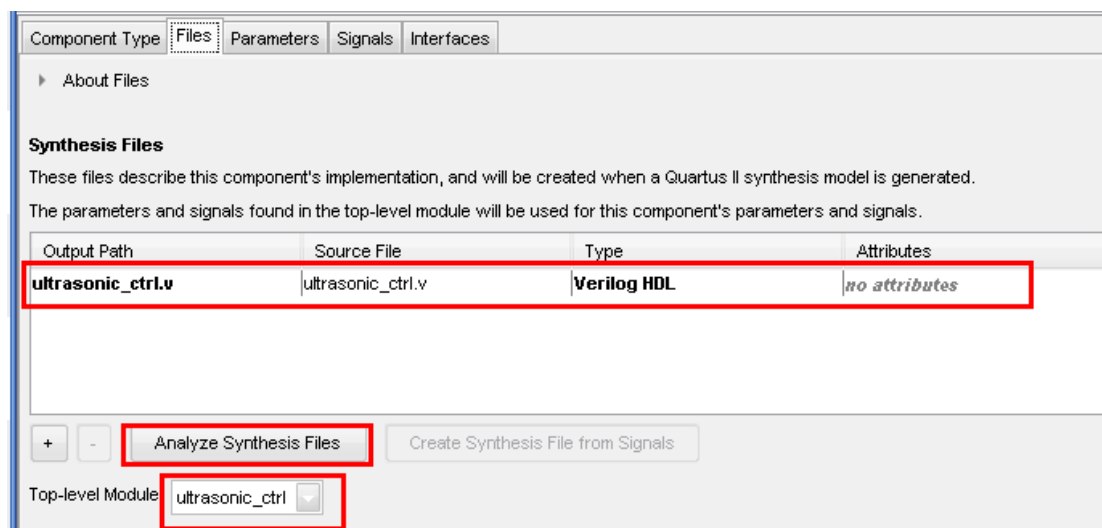


8.5.2 硬件系统搭建

复制 ex19 整个工程目录, 重新命名为 ex20。打开工程, 进入 Qsys 页面。进入 Component Editor 页面, 在 Component Type 页面, 设置 Name 和 Display Name 均为 ultrasonic。



在 Files 页面, 先添加 ultrasonic_ctrl.v 文件, 然后点击 Analyze Synthesis Files 进行综合, 通过综合后, 将 ultrasonic_ctrl.v 设置为 Top-level Module。



Signal 页面, 我们一共有 4 大类的 interface, 具体大家可以参考代码中对这些信号的注释描述以及在实际逻辑中的功能。



Component Type	Files	Parameters	Signals	Interfaces
About Signals				
Name	Interface	Signal Type	Width	Direction
clk	clock_sink	clk	1	input
rst_n	reset_sink	reset_n	1	input
trig	conduit_end	export	1	output
echo	conduit_end	export	1	input
sys_cs_n	avalon_slave	chipselect_n	1	input
sys_rd_n	avalon_slave	read_n	1	input
sys_rddata	avalon_slave	readdata	32	output

接下来是各个 Interfaces 的具体设置。

"clock_sink" (Clock Input)

Name:

Type:

Block Diagram

Parameters

Clock rate:

"reset_sink" (Reset Input)

Name:

Type:

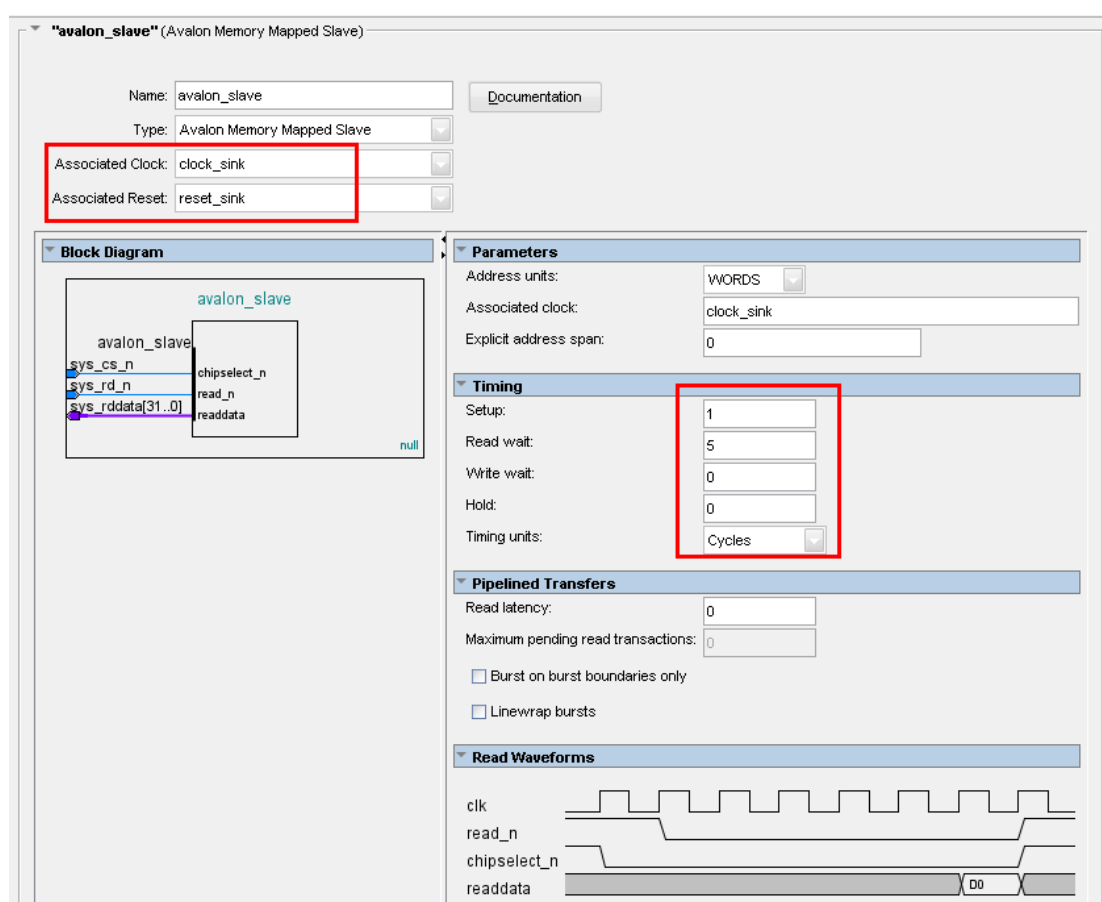
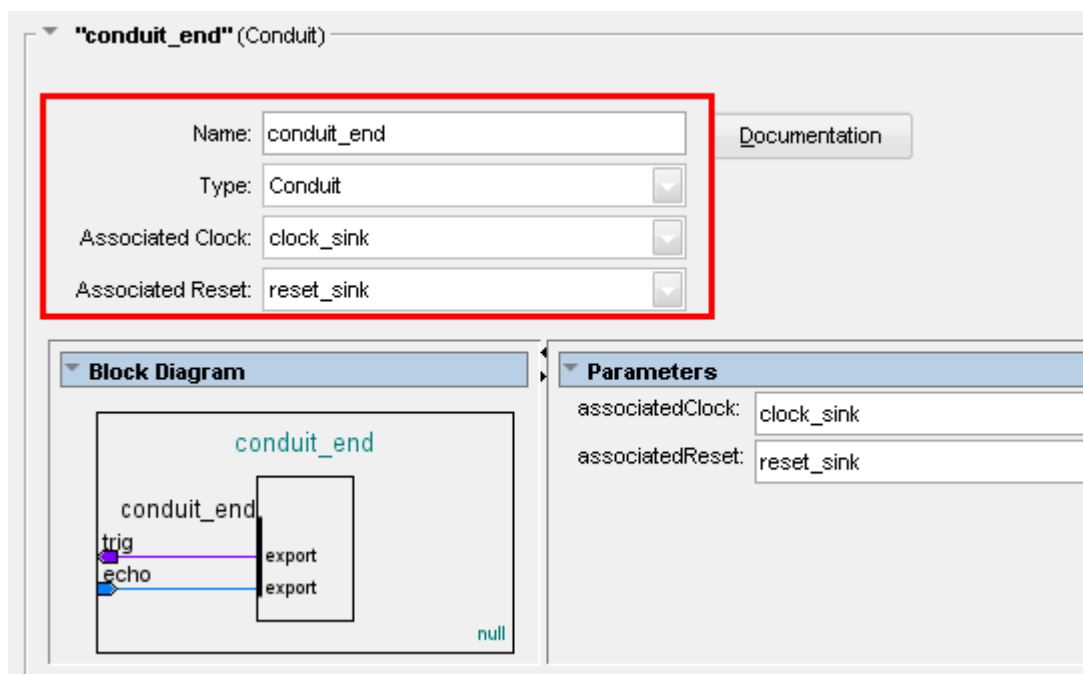
Associated Clock:

Block Diagram

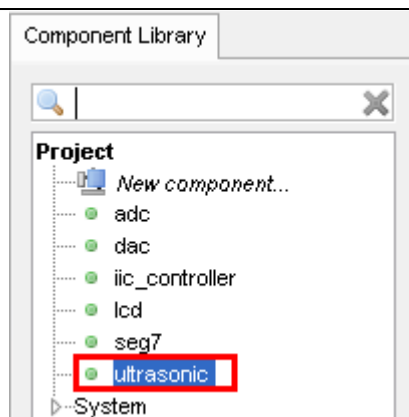
Parameters

Associated clock:

Synchronous edges:



完成设置后, 我们看到 Component Library 中出现了新的组件 ultrasonic, 双击它添加到 Qsys 中。



修改新组件的名称为 myultrasonic, 对其 clock_sink、reset_sink、conduit_end、avalon_slave 接口进行连接, 如图所示。同时重新分配地址, 最后 Generate 整个系统。

Use	Connections	Name	Description	Export
<input checked="" type="checkbox"/>		sysio_qsys	System I/O Peripheral	
		clk	Clock Input	Click to export
		reset	Reset Input	Click to export
		control_slave	Avalon Memory Mapped Slave	Click to export
<input checked="" type="checkbox"/>		jtag_uart	JTAG UART	
		clk	Clock Input	Click to export
		reset	Reset Input	Click to export
		avalon_jtag_slave	Avalon Memory Mapped Slave	Click to export
<input checked="" type="checkbox"/>		onchip_mem	On-Chip Memory (RAM or ROM)	
		clk1	Clock Input	Click to export
		s1	Avalon Memory Mapped Slave	Click to export
		reset1	Reset Input	Click to export
<input checked="" type="checkbox"/>		nios2_qsys	Nios II Processor	
		clk	Clock Input	Click to export
		reset_n	Reset Input	Click to export
		data_master	Avalon Memory Mapped Master	Click to export
		instruction_master	Avalon Memory Mapped Master	Click to export
		jtag_debug_module_re...	Reset Output	Click to export
		jtag_debug_module	Avalon Memory Mapped Slave	Click to export
		custom_instruction_m...	Custom Instruction Master	Click to export
				nios2_qsys_0_custom_i...
<input checked="" type="checkbox"/>		timer	Interval Timer	
		clk	Clock Input	Click to export
		reset	Reset Input	Click to export
		s1	Avalon Memory Mapped Slave	Click to export
<input checked="" type="checkbox"/>		mylcd	Lcd	
		clock_sink	Clock Input	Click to export
		reset_sink	Reset Input	Click to export
		avalon_slave	Avalon Memory Mapped Slave	Click to export
		conduit_end	Conduit	Click to export
				mylcd_conduit_end
<input checked="" type="checkbox"/>		myspi	SPI (3 Wire Serial)	
		clk	Clock Input	Click to export
		reset	Reset Input	Click to export
		spi_control_port	Avalon Memory Mapped Slave	Click to export
		external	Conduit Endpoint	Click to export
				myspi_external
<input checked="" type="checkbox"/>		myiic	iic_controller	
		clock_sink	Clock Input	Click to export
		reset_sink	Reset Input	Click to export
		avalon_slave	Avalon Memory Mapped Slave	Click to export
		conduit_end	Conduit	Click to export
				myiic_conduit_end
<input checked="" type="checkbox"/>		myultrasonic	ultrasonic	
		clock_sink	Clock Input	Click to export
		reset_sink	Reset Input	Click to export
		conduit_end	Conduit	Click to export
		avalon_slave	Avalon Memory Mapped Slave	Click to export
				myultrasonic_conduit_end

回到 Quartus II 的顶层模块, 添加超声波模块的接口。

《圣经》箴言九 11 “敬畏耶和华是智慧的开端, 认识至胜者便是聪明。”



```
module (
    .....
    trig, echo
);
.....

//FPGA 与超声波模块接口
output trig;    //超声波模块工作触发信号
input echo;     //超声波模块反馈信号
.....

//-----
//Qsys 系列例化
myqsys u0 (
    .....
    //trig
    .myultrasonic_conduit_end_trig(trig),
    //echo
    .myultrasonic_conduit_end_echo(echo)
);
.....
endmodule
```

对这两个新接口信号的管脚做分配。

```
set_location_assignment PIN_86 -to trig
set_location_assignment PIN_87 -to echo
```

编译整个工程。连接好 SF-LCD 和 SF-SENSOR 模块，上电后下载 sof 文件到工程中。

8.5.3 软件工程调试

假设超声波模块与障碍物间的距离为 S (单位: m), ECHO 输出的高脉冲宽度为 T (单位: s), 声速定义为 340 (单位: m/s)。那么 ECHO 脉冲宽度与测试距离的关系如下。

$$S = (T * 340) / 2$$

如果 T 为 n 个 25MHz 时钟采样周期数, 并且取 S 的单位为 cm。那么, 公式改进为:

$$S = ((n * 0.00000004 * 340) / 2) * 100 = n * 68 / 100000$$

运用上述理论, 我们接下来要创建一个软件工程, 读取超声波组件不断采集到的距离信



息, 在 LCD 上打印显示出来。

新建一个软件工程, 然后进入 BSP 裁剪代码, 接着创建一个 main.c 文件。拷贝 ex19 的软件工程, main 函数修改如下。在主循环中我们不断的读取实时的测距数据, 然后打印显示。

```
/*
*****
* 名 称: 主函数
* 函数名: main
* 入 口: 无
* 出 口: 无
*****
*/
int main(void)
{
    alt_u16 x,y;
    alt_u32 temp;

    //清全屏为蓝色
    for(y=0;y<240;y++)
    {
        for(x=0;x<320;x++)
        {
            lcd_wrdw(x,y,0xffff);
        }
    }

    init_spi(); //初始化 SPI 的 control 和 status 寄存器

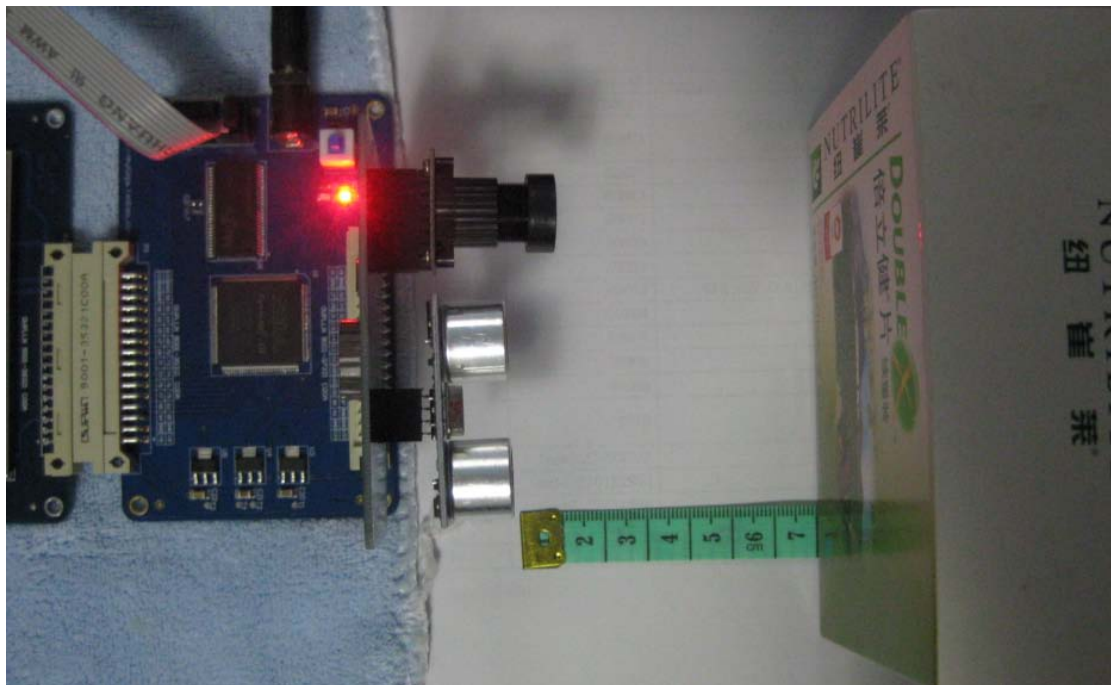
    lcd_print(60,111,"http://myfpga.taobao.com/",0xffff,0xf800);
    lcd_print(32,143,"特权电子开发套件带您迈入",0xffff,0x07e0);
    lcd_print(180,164,"FPGA",0xffff,0x07e0);
    lcd_print(212,164,"的开发大门",0xffff,0x07e0);
    lcd_print(80,51,"distance = ",0xffff,0x07e0);

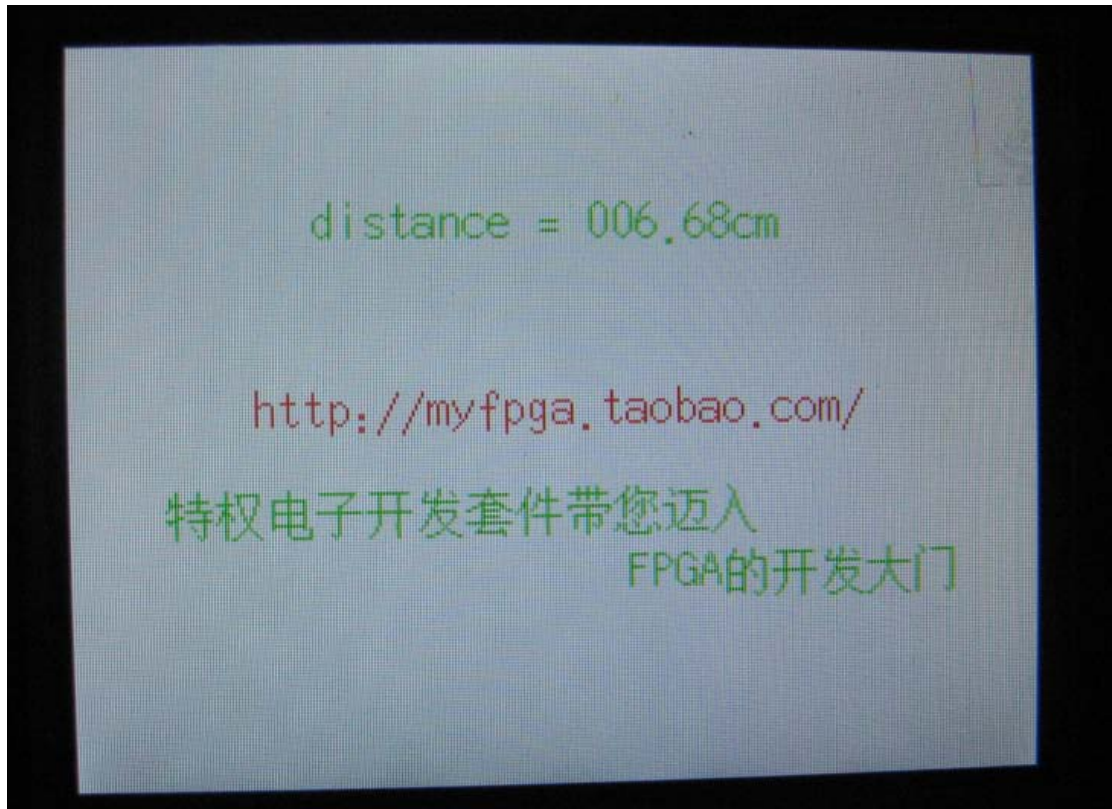
    while(1)
    {
        //显示小数点后两位距离数据
        temp = IORD_32DIRECT(MYULTRASONIC_BASE, 0); //读取当前的超声波模块数据
        temp = temp*68/1000;
```



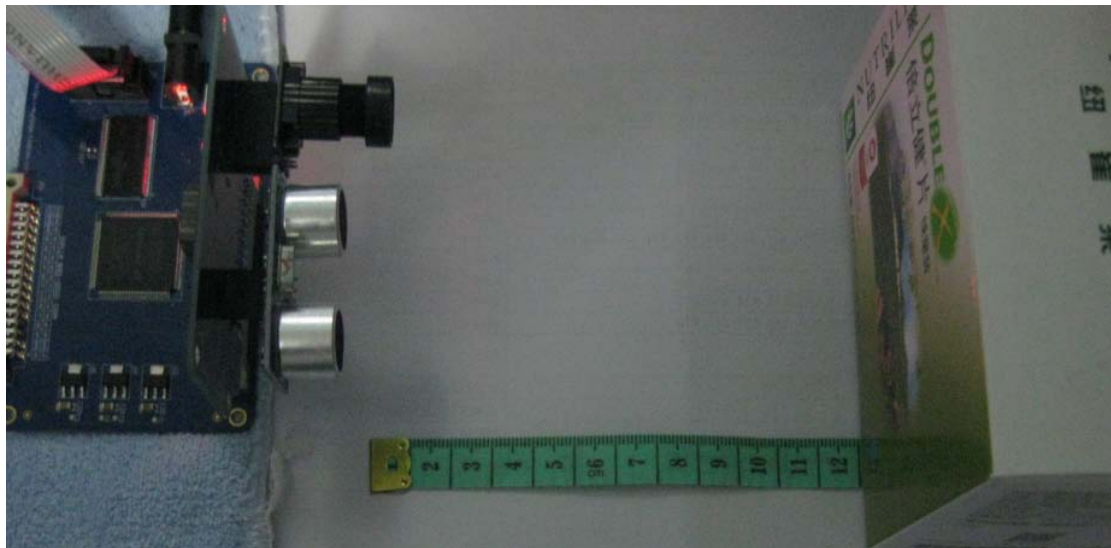
```
dis_db[5] = (alt_u8) (temp%10+0x30);  
temp = temp/10;  
dis_db[4] = (alt_u8) (temp%10+0x30);  
temp = temp/10;  
dis_db[2] = (alt_u8) (temp%10+0x30);  
temp = temp/10;  
dis_db[1] = (alt_u8) (temp%10+0x30);  
temp = temp/10;  
dis_db[0] = (alt_u8) (temp%10+0x30);  
  
lcd_print(168, 51, dis_db, 0xffff, 0x07e0); //LCD 上显示距离信息  
}  
}
```

运行软件工程, 此时我们可以看到如下的测试效果。7cm 左右的障碍物, 换算出来大约 6.68cm。





12cm 左右的障碍物，我们换算出来的大约是 11.47cm。





好像还是存在点误差，哈哈，这个就不用计较了，比较咱的尺子和盒子放置得也不能保证百分百的无误差。

8.6 逻辑 (Verilog) 实例 14——基于 CMOS Sensor 的视频采集显示

8.6.1 CMOS 摄像头应用背景与驱动原理

CMOS 摄像头 (CMOS Sensor) 是一种采用 CMOS 图像传感器的摄像头。摄像头主要有两类，CMOS 和 CCD；CMOS 一般应用在普通数码设备中，CCD 一般应用高档数码设备中，都是光学成像，CCD 比 CMOS 单位成像的效果要好。CCD 镜头比 CMOS 颜色还原要好分辨率要高。

CCD 和 CMOS 在制造上的主要区别是 CCD 是集成在半导体单晶材料上，而 CMOS 是集成在被称做金属氧化物的半导体材料上，工作原理没有本质的区别。CCD 只有少数几个厂商例如索尼、松下等掌握这种技术。而且 CCD 制造工艺较复杂，采用 CCD 的摄像头价格都会



相对比较贵。事实上经过技术改造,目前 CCD 和 CMOS 的实际效果的差距已经减小了不少。而且 CMOS 的制造成本和功耗都要低于 CCD 不少,所以很多摄像头生产厂商采用的 CMOS 感光元件。成像方面:在相同像素下 CCD 的成像通透性、明锐度都很好,色彩还原、曝光可以保证基本准确。而 CMOS 的产品往往通透性一般,对实物的色彩还原能力偏弱,曝光也都不太好,由于自身物理特性的原因,CMOS 的成像质量和 CCD 还是有一定距离的。但由于低廉的价格以及高度的整合性,因此在摄像头领域还是得到了广泛的应用。

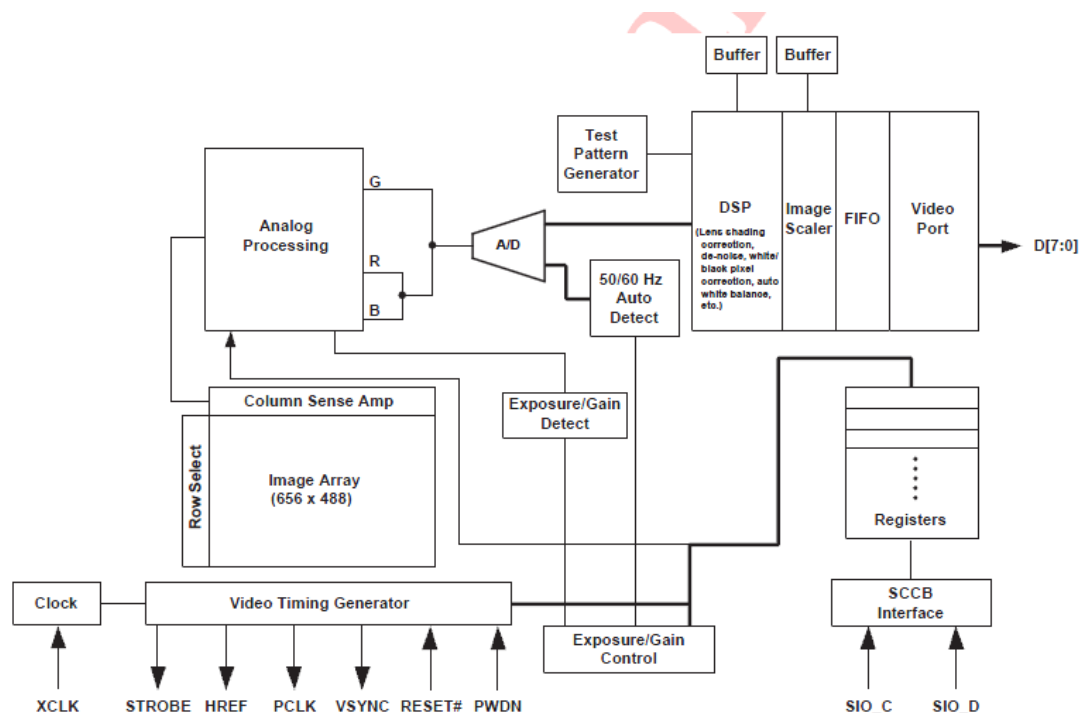
CCD 是目前比较成熟的成像器件,CMOS 被看作未来的成像器件。因为 CMOS 结构相对简单,与现有的大规模集成电路生产工艺相同,从而生产成本可以降低。从原理上,CMOS 的信号是以点为单位的电荷信号,而 CCD 是以行为单位的电流信号,前者更为敏感,速度也更快,更为省电。现在高级的 CMOS 并不比一般 CCD 差,但是 CMOS 工艺还不是十分成熟,普通的 CMOS 一般分辨率低而成像较差。

不管,CCD 或 CMOS,基本上两者都是利用矽感光二极管 (photodiode) 进行光与电的转换。这种转换的原理与各位手上具备“太阳能”电子计算机的“太阳能电池”效应相近,光线越强、电力越强;反之,光线越弱、电力也越弱的道理,将光影像转换为电子数字信号。

比较 CCD 和 CMOS 的结构,ADC 的位置和数量是最大的不同。简单的说,按我们在上一讲“CCD 感光元件的工作原理(上)”中所提之内容。CCD 每曝光一次,在快门关闭后进行像素转移处理,将每一行中每一个像素 (pixel) 的电荷信号依序传入“缓冲器”中,由底端的线路引导输出至 CCD 旁的放大器进行放大,再串联 ADC 输出;相对地,CMOS 的设计中每个像素旁就直接连着 ADC (放大兼类比数字信号转换器),讯号直接放大并转换成数字信号。



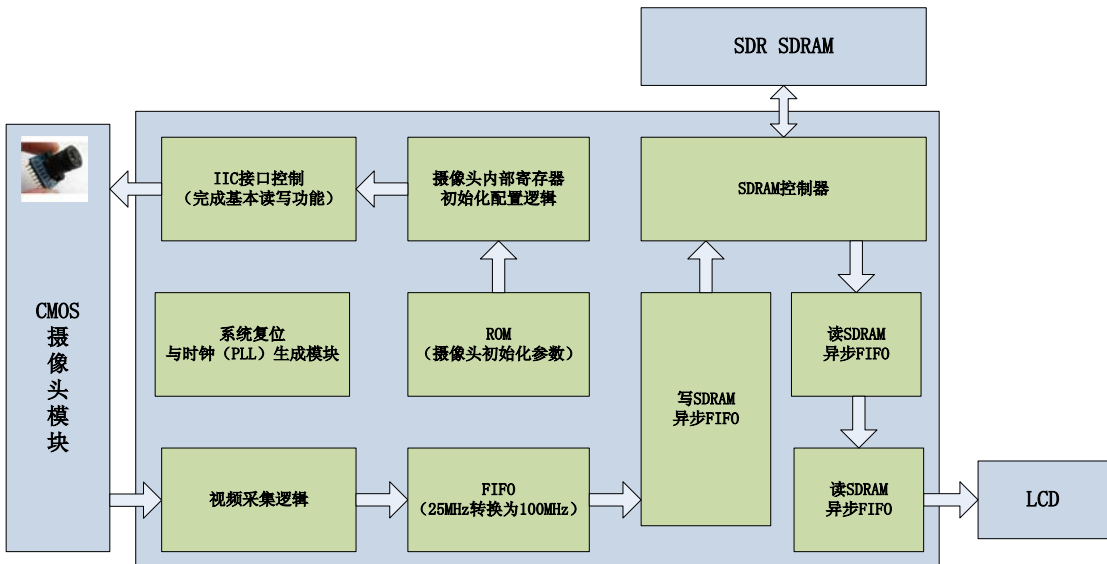
我们这个模块中所使用的 CMOS Sensor 的型号为 OV7670,其内部的功能框图如下所示。前端有模拟感光模块,经过 AD 处理后将模拟信号转换为数字信号,后端经过一些处理后输出符合一定协议标准的视频数据流。我们 FPGA 将根据这个数据流的协议对视频数据进行采集解码,最终显示的 3.5 寸的 320*240 液晶屏上。这个 Sensor 可以输出 640*480 和 320*240 两种分辨率的图像,我们通过 IIC 接口配置他们的寄存器可以切换输出分辨率。



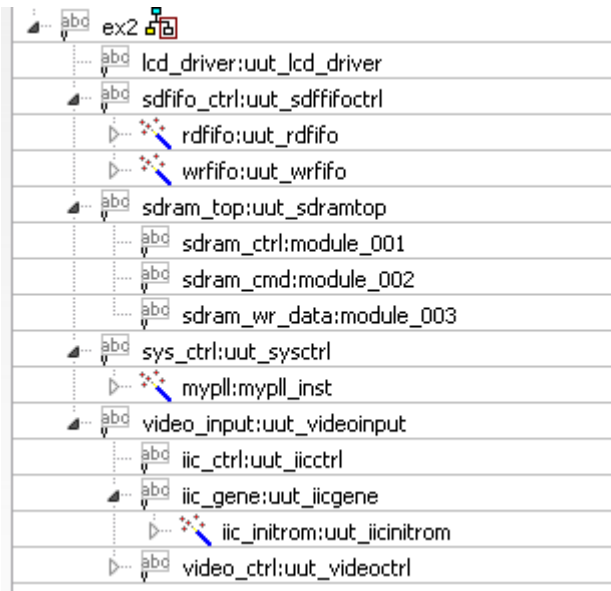
8.6.2 采集系统设计概述

FPGA 工程的功能框图如图所示。上电初始，FPGA 需要通过 IIC 接口协议对摄像头模块进行寄存器初始化配置。这个初始化的基本参数，如初始化地址和数据存储在一个预先配置好的 FPGA 内嵌 ROM 中。在初始化配置完成后，摄像头就能够持续输出 RGB 标准的视频数据流，FPGA 通过对其相应的时钟、行频和场频进行检测，从而一帧一帧的实时采集图像数据。

采集到的视频数据先通过一个 FIFO，将原本 25MHz 频率下同步的数据流转换到 100MHz 频率下。接着讲这个数据再送入写 SDRAM 缓存的 FIFO 中，最终这个 FIFO 每满 160 个数据就会将其写入 SDRAM 的相应地址中。在另一侧，使用另一个异步 FIFO 将 SDRAM 缓存的图像数据送个 LCD 驱动模块。LCD 驱动模块不断的读出新的现实图像，并且驱动 3.5 寸液晶屏工作。



如图所示，这是整个工程的各个模块层次。在顶层模块 `ex2` 下面有 5 个子模块。这 5 个子模块的功能以及他们所包含的子模块或例化功能描述如表所示。



模块名称	功能描述
lcd_driver	二级子模块，该模块驱动 LCD，同时产生控制逻辑读取 wrfifo（sdfifo_ctrl 模块的读 SDRAM 缓存 FIFO）数据。
sdfifo_ctrl	该模块主要用于缓存读或写 SDRAM 的数据，其下例化了两个 FIFO。video_input 模块将视频数据流送到这个模块中例化的 rdfifo（写 SDRAM 缓存 FIFO），在 lcd_driver 的控制信号控制下，这个模块的 wrfifo 的数据则送出用于 LCD 显示。
sdram_top	这个模块就不用多说了，是 ex16 例程中移植过来的，用于产生控制 SDRAM



	芯片的逻辑。其下有 3 个子模块。
sys_ctrl	该模块对系统的复位做“异步复位，同步释放”的复位可靠性处理。同时例化了一个 PLL 单元用于产生系统所需要的不同频率时钟信号。
video_input	该模块下有 3 个子模块。video_ctrl 模块对视频数据流进行采集，送到其内部的一个 FIFO 中，然后再送往 sdfifo_ctrl 最终写入 SDRAM；iic_ctrl 模块产生 IIC 接口一个字节有效数据的读或写时序；iic_gene 模块则产生 CMOS 摄像头的配置数据，它读取内部设置好的 ROM 数据，送往 iic_ctrl 模块，对摄像头内的各个寄存器进行初始化。

8.6.3 IIC 接口配置模块设计

iic_ctrl 模块是从 ex18 工程中移植过来的，大同小异，没有什么大的修改，这里就不再详细说明。

iic_gene 模块时通过 iic_ctrl 模块的读写接口信号实现 CMOS 摄像头寄存器的初始化配置。该模块下还例化了一个内嵌 ROM。该模块代码如下。代码中注释的一大段用于产生 IIC 的地址和数据的代码，在实际应用中则用 ROM 内的数据替代了。

```

module iic_gene(
    clk,rst_n,
    tiic_en,tiic_ab,tiic_db
);

input clk;      // 50MHz 主时钟
input rst_n;    //低电平复位信号

output tiic_en;  //需要通过 IIC 接口配置 MAX9526 使能信号，高电平有效
output reg[7:0] tiic_ab;    //需要通过 IIC 接口配置 MAX9526 地址
output reg[7:0] tiic_db;    //需要通过 IIC 接口配置 MAX9526 数据

//-----
//初始化参数 ROM 例化
wire[6:0] romab;
wire[15:0] romdb;

```




```
iic_initrom      uut_iicinitrom(
                    .address(romab),
                    .clock(clk),
                    .q(romdb)
                );

//-----
//上电延时 20ms
reg[19:0] dcnt; //延时计数器
reg[6:0] mcnt; //mcnt 个 20ms 计数周期

always @(posedge clk or negedge rst_n)
    if(!rst_n) dcnt <= 20'd0;
    else if(mcnt < 7'd126) begin
        if(dcnt < 20'd30000) dcnt <= dcnt+1'b1;
        else dcnt <= 20'd0;
    end

wire wren = (dcnt == 20'd1000);

//-----
//20ms 计数周期控制

always @(posedge clk or negedge rst_n)
    if(!rst_n) mcnt <= 7'd0;
    else if(dcnt == 20'd2000) mcnt <= mcnt+1'b1;

//-----
//初始化寄存器设置
    //12H 地址设置为 14H, bit4=1--QVGA 模式, bit2=1, bit0=0--RGB 模式
parameter  RIDLE = 3'd0;
parameter  RSET1 = 3'd1;
parameter  ROVER = 3'd2;
reg[2:0] rstate;

always @(posedge clk or negedge rst_n)
    if(!rst_n) begin
        rstate <= RIDLE;
```



```
        tiic_ab <= 8'd0;
        tiic_db <= 8'd0;
    end
    else begin
        case(rstate)
            RIDLE: if((mcnt > 7'd6) && wren) begin
                    rstate <= RSET1;
                    tiic_ab <= romdb[15:8];
                    tiic_db <= romdb[7:0];
                end
            else if(mcnt == 7'd120) rstate <= ROVER;
            else rstate <= RIDLE;
            RSET1: rstate <= RIDLE;
            ROVER: rstate <= ROVER;
            default: rstate <= RIDLE;
        endcase
    end

    assign tiic_en = (rstate == RSET1);    //IIC 写入请求信号
    assign romab = mcnt-7'd7;

endmodule
```

下面简单说说如何对这个 ROM 和初始化数据的进行创建和配置。

点击 Quartus II 菜单栏的 File-New, 选择 Memory Initialization File。然后保存为 iic_init.mif, 输入配置数据如下。

```
-- Copyright (C) 1991-2011 Altera Corporation
-- Your use of Altera Corporation's design tools, logic functions
-- and other software and tools, and its AMPP partner logic
-- functions, and any output files from any of the foregoing
-- (including device programming or simulation files), and any
-- associated documentation or information are expressly subject
-- to the terms and conditions of the Altera Program License
-- Subscription Agreement, Altera MegaCore Function License
-- Agreement, or other applicable license agreement, including,
-- without limitation, that your use is for the sole purpose of
-- programming logic devices manufactured by Altera and sold by
-- Altera or its authorized distributors. Please refer to the
```



```
-- applicable agreement for further details.

-- Quartus II generated Memory Initialization File (.mif)

WIDTH=16;
DEPTH=128;

ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;

CONTENT BEGIN
    00 : 3A04;
    01 : 4010;
    02 : 1214;
    03 : 3280;
    04 : 1716;
    05 : 1804;
    06 : 1903;
    07 : 1A7B;
    08 : 0306;
    09 : 0C0C;
    0A : 3E00;
    0B : 7000;
    0C : 7101;
    0D : 7211;
    0E : 7309;
    0F : A202;
    10 : 1101;
    11 : 7A20;
    12 : 7B1C;
    13 : 7C28;
    14 : 7D3C;
    15 : 7E55;
    16 : 7F68;
    17 : 8076;
    18 : 8180;
    19 : 8288;
    1A : 838F;
```



1B : 8496;
1C : 85A3;
1D : 86AF;
1E : 87C4;
1F : 88D7;
20 : 89E8;
21 : 13E0;
22 : 0000;
23 : 1000;
24 : 0D00;
25 : 1420;
26 : A505;
27 : AB07;
28 : 2475;
29 : 2563;
2A : 26A5;
2B : 9F78;
2C : A068;
2D : A103;
2E : A6DF;
2F : A7DF;
30 : A8F0;
31 : A990;
32 : AA94;
33 : 13E5;
34 : 0E61;
35 : 0F4B;
36 : 1602;
37 : 1E07;
38 : 2102;
39 : 2291;
3A : 2907;
3B : 330B;
3C : 350B;
3D : 371D;
3E : 3871;
3F : 392A;
40 : 3C78;



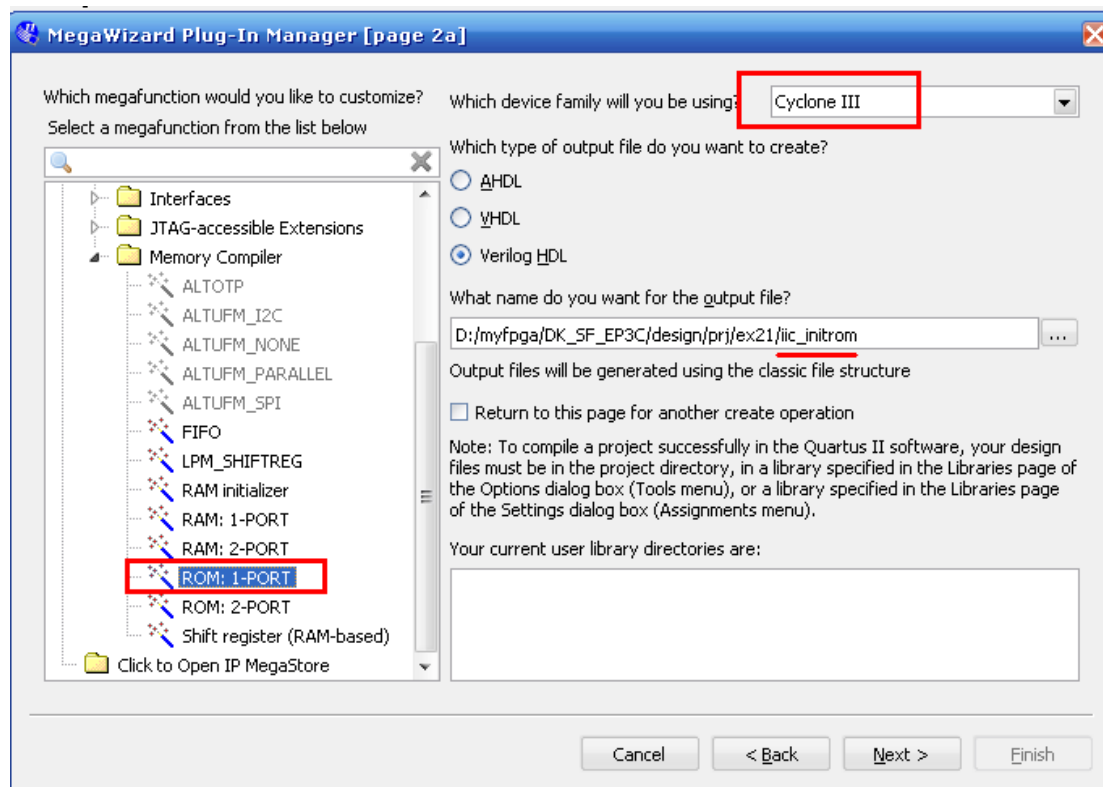
41 : 4D40;
42 : 4E20;
43 : 695D;
44 : 6B40;
45 : 7419;
46 : 8D4F;
47 : 8E00;
48 : 8F00;
49 : 9000;
4A : 9100;
4B : 9200;
4C : 9600;
4D : 9A80;
4E : B084;
4F : B10C;
50 : B20E;
51 : B382;
52 : B80A;
53 : 4314;
54 : 44F0;
55 : 4534;
56 : 4658;
57 : 4728;
58 : 483A;
59 : 5988;
5A : 5A88;
5B : 5B44;
5C : 5C67;
5D : 5D49;
5E : 5E0E;
5F : 6404;
60 : 6520;
61 : 6605;
62 : 9404;
63 : 9508;
64 : 6C0A;
65 : 6D55;
66 : 4F80;



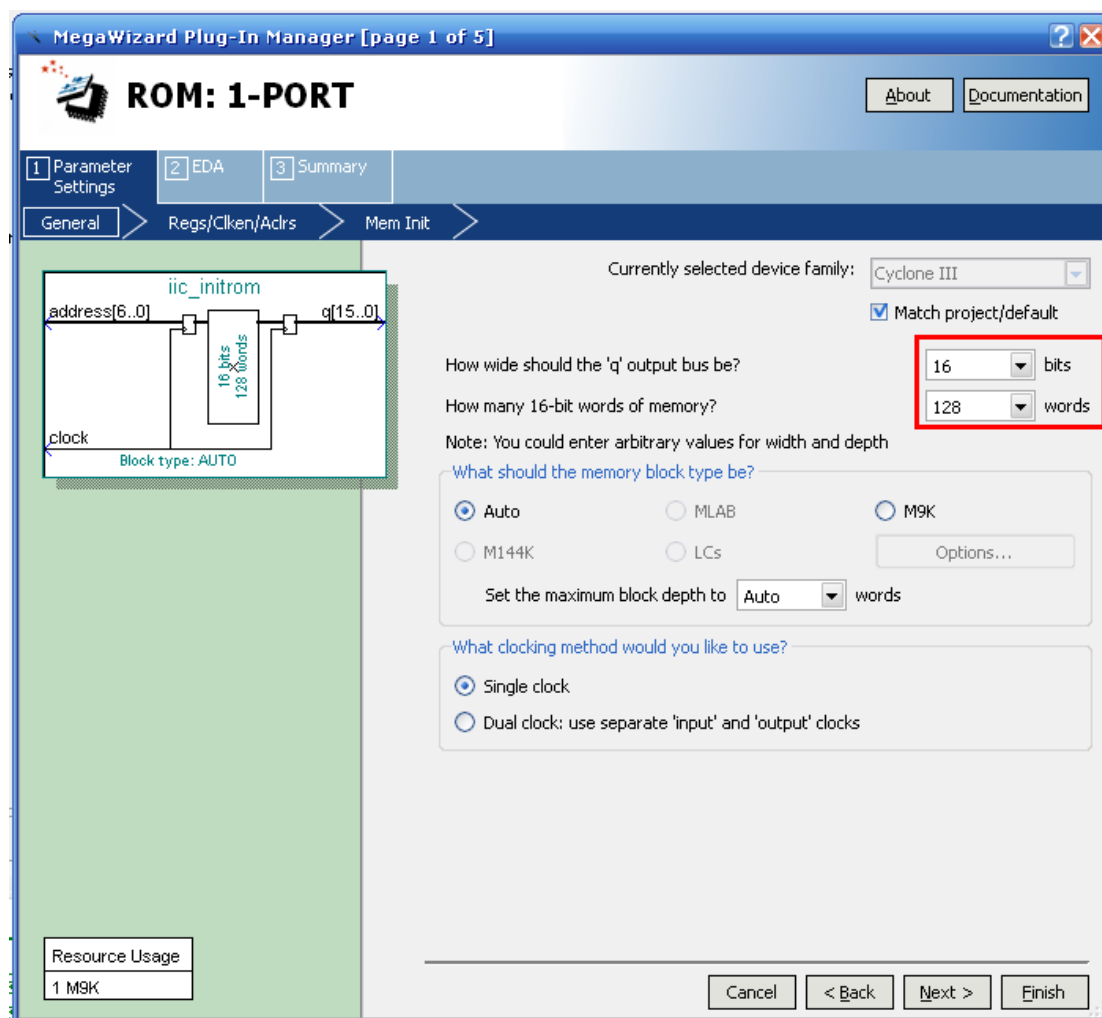
```
67 : 5080;
68 : 5100;
69 : 5222;
6A : 535E;
6B : 5480;
6C : 6E11;
6D : 6F9F;
6E : 5500;
6F : 5640;
70 : 5780;
[71..7F] : 0000;
END;
```

关于寄存器的具体配置,大家可以参考 `iic_gene` 模块代码中的注释,对应寄存器和数据,然后查看 OV7670 的 `datasheet`,对照各个功能进行解读。

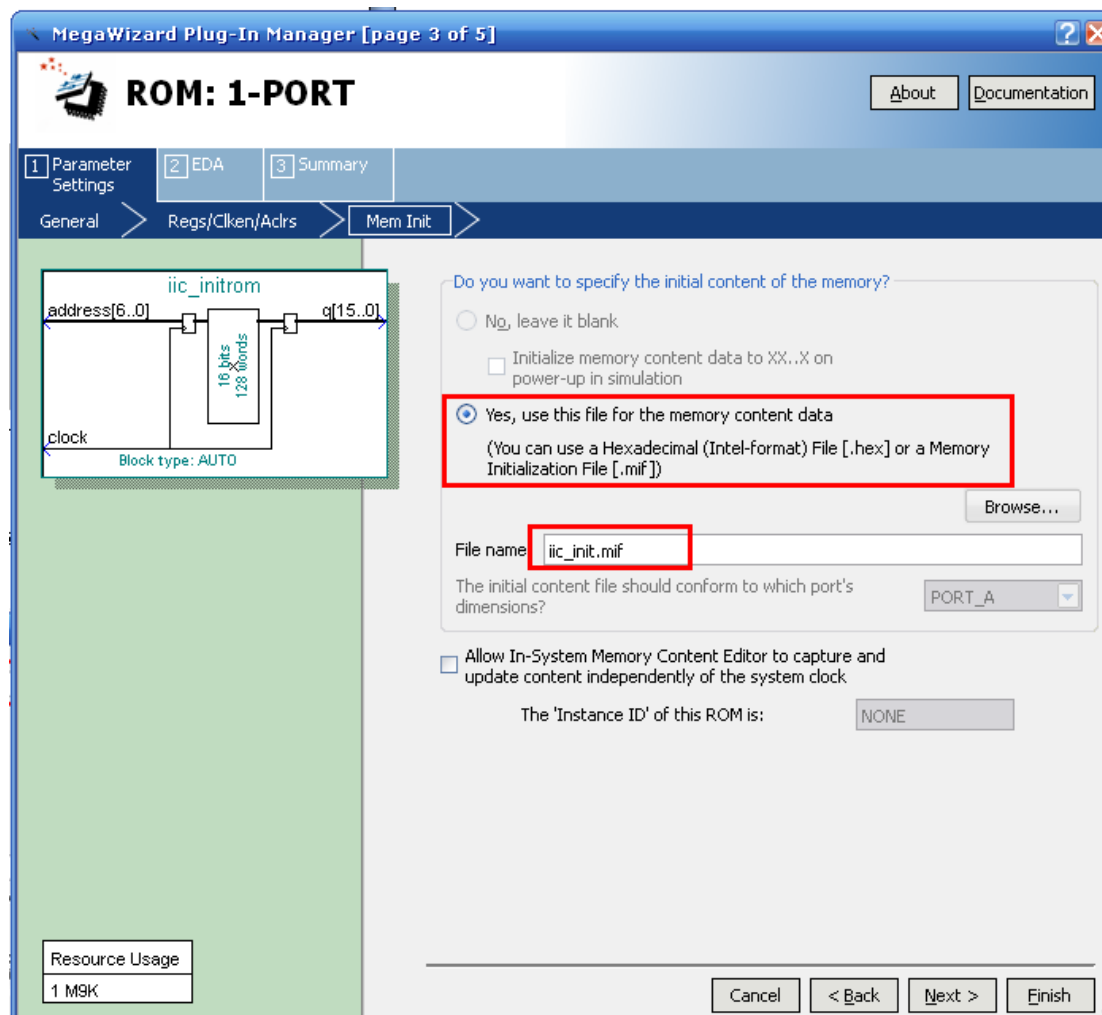
接着我们来配置 ROM。首先进入 MegaWizard Plug-In Manager 中,选择 ROM:1-PORT,设置它的路径为当前工程 `ex21` 下,名称为 `iic_initrom`。



General 页面,配置这个 ROM 的位宽 16bits,深度 128words。



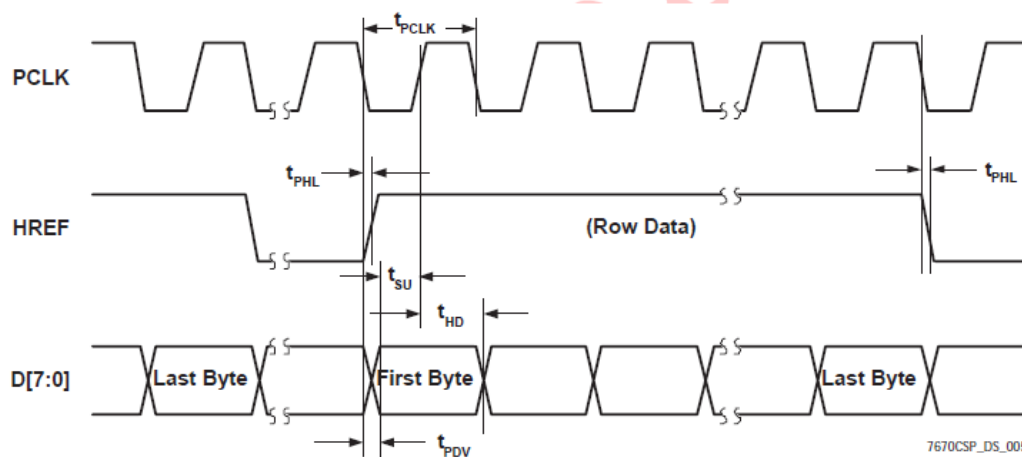
在 Mem Init 页面, 勾选 yes,use this file for the memory....选项, 然后选择我们刚才创建的 iic_init.mif 文件。



最后，点击 Finish 就完成 ROM 的配置了。

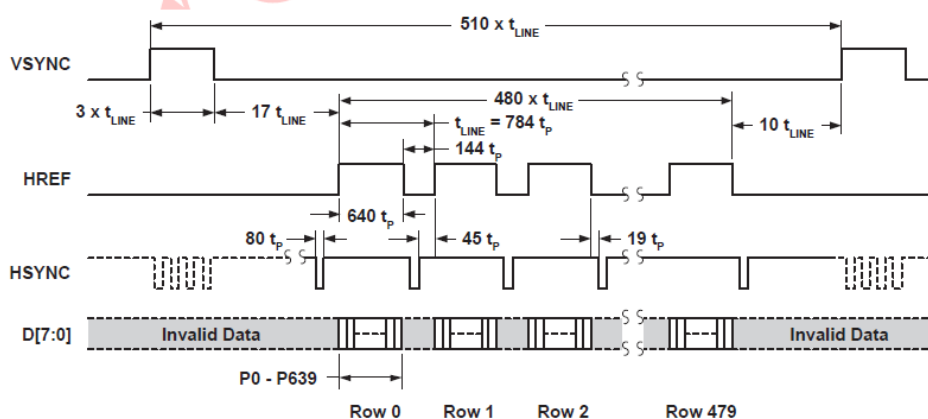
8.6.4 视频流采集模块设计

8.1 中已经介绍了 CMOS Sensor 的接口，这里我们来看看它的时序。如图所示，视频时钟 PCLK (vpclk) 的每个上升沿，有效数据 D[7:0] (vdb) 或行同步信号 HREF (vhref)、场同步信号 VSYNC (vwsync，波形中没有示意) 被锁存到 FPGA 中。



下图示意的是 VGA（640*480 分辨率）的时序波形，我们可以看到，场同步信号 VSYNC 在的每一个高脉冲表示新的一场图像（或者说是新的一帧图像）马上要开始传输；行同步信号 HREF 为高电平时，表示目前的数据总线 D[7:0]上的数据是有效的视频流。我们这个模块要采集的是 QVGA（320*240 分辨率）图像，时序和 VGA 图像是类似的，只不过有效采集的数据区域有所不同。

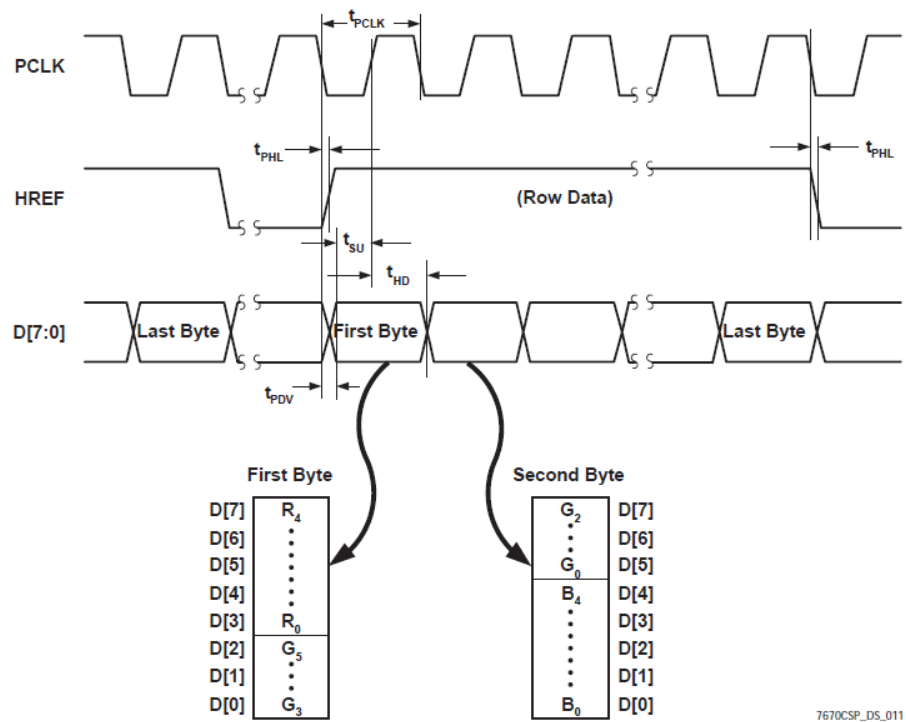
Figure 6 VGA Frame Timing



一个有效的行将传输 320*2Bytes 的数据，也就是说，一个像素点会有 2Bytes 即 16bits 的有效色彩值。对应 R、G、B 的位数分别为 5bits、6bits、5bits。传输的数据总线是 8bits，那么一个像素点对应就有 2 个 8bits 需要传输。每两个字节中的 R、G、B 格式定义如图所示。



Figure 11 RGB 565 Output Timing Diagram



理解了时序波形,我们再来看看代码中是如何对 CMOS Sensor 送来的这组源同步信号进行采集的。实际上很简单,我们只要把 vpclk、vdb、vhref 分别作为 FIFO 的写入时钟、写入数据和写入使能信号。此外,wsync 作为这个 FIFO 的复位信号,每一帧新图像前 FIFO 进行一次清空,实际操作中,为了得到稳定有效的复位信号,我们使用内部时钟对这个复位信号打了一拍。这样,我们便把持续不断的视频流有效数据缓存到了 FIFO 中。在 FIFO 的读端,在数据快满 160Bytes (SDRAM 的一页)时,送到 sdfifo_ctrl 模块的 SDRAM 写缓存 FIFO 中。设计的代码如下。

```

module video_ctrl(
    clk,rst_n,
    vpclk,vvsync,vhref,vdb,
    wrf_din,wrf_wrreq
);

input clk;           //系统时钟, 25MHz
input rst_n;         //复位信号, 低电平有效
//视频输入接口
input vpclk;         //视频时钟
input vvsync;        //视频场同步信号
input vhref;         //视频行同步信号

```

《圣经》箴言九 11 “敬畏耶和华是智慧的开端,认识至胜者便是聪明。”



```
input[7:0] vdb; //视频数据总线
    //wrFIFO 输入控制接口
output[15:0] wrf_din; //sdr 数据写入缓存 FIFO 输入数据总线
output wrf_wrreq; //sdr 数据写入缓存 FIFO 数据输入请求, 高有效

//-----

reg wrf_clr_r; //wrf_clr_r 同步锁存一拍

always @(posedge clk or negedge rst_n)
    if(!rst_n) wrf_clr_r <= 1'b0;
    else wrf_clr_r <= vvsync;

//-----

//数据缓存 FIFO 例化
wire[7:0] vf_rduse; //数据有效个数

wire vf_rdreq = ~(dcnt == 8'd0); //读 FIFO 请求

video_fifo uut_videofifo(
    .aclr(wrf_clr_r),
    .data(vdb),
    .rdclk(clk),
    .rdreq(vf_rdreq),
    .wrclk(vpclk),
    .wrreq(vhref),
    .q({wrf_din[7:0], wrf_din[15:8]}), //input first--LSB, input
second--MSB
    .rdusedw(vf_rduse)
);

//-----

//写入请求信号产生
reg[7:0] dcnt;

always @(posedge clk or negedge rst_n)
    if(!rst_n) dcnt <= 8'd0;
    else if(vf_rduse > 8'd158) dcnt <= dcnt+1'b1;
    else if((dcnt != 8'd0) && (dcnt < 8'd160)) dcnt <= dcnt+1'b1;
```



```
        else dcnt <= 8'd0;

reg wrf_wrreqr;

always @(posedge clk or negedge rst_n)
    if(!rst_n) wrf_wrreqr <= 1'b0;
    else wrf_wrreqr <= vf_rdreq;

assign wrf_wrreq = wrf_wrreqr; //sdram 数据写入缓存 FIFO 数据输入请求, 高有效

endmodule
```

前面提到过, 视频采集模块 **video_ctrl** 和 IIC 接口配置的两个模块 **iic_ctrl** 和 **iic_gene** 都是例化到 **video_input** 模块中, 这个模块的代码如下。它实际上只是起到了各个模块衔接的效果, 便于后续的管理和维护。

```
module video_input(
    clk, clk_100m, rst_n,
    vpclk, vvsync, vhref, vdb, vscl, vsda,
    wrf_din, wrf_wrreq
);

input clk;           //系统时钟, 25MHz
input clk_100m;      //PLL 输出 100MHz 时钟
input rst_n;         //复位信号, 低电平有效
    //视频输入接口
input vpclk;         //视频时钟
input vvsync;        //视频场同步信号
input vhref;         //视频行同步信号
input[7:0] vdb;      //视频数据总线
output vscl;         //串行配置 IIC 时钟信号
inout vsda;          //串行配置 IIC 数据信号
    //wrFIFO 输入控制接口
output[15:0] wrf_din; //sdram 数据写入缓存 FIFO 输入数据总线
output wrf_wrreq;     //sdram 数据写入缓存 FIFO 数据输入请求, 高有效

//-----
    //视频 IIC 配置端口
wire tiic_en;         //需要通过 IIC 接口配置 MAX9526 使能信号, 高电平有效
```



```
wire[7:0] tiic_ab; //需要通过 IIC 接口配置 MAX9526 地址
wire[7:0] tiic_db; //需要通过 IIC 接口配置 MAX9526 数据
```

```
//-----
```

```
//IIC 时序产生模块
```

```
iic_ctrl    uut_iicctrl(
                .clk(clk),
                .rst_n(rst_n),
                .tiic_en(tiic_en),
                .tiic_ab(tiic_ab),
                .tiic_db(tiic_db),
                .scl(vscl),
                .sda(vsda)
            );
```

```
//-----
```

```
//IIC 寄存器
```

```
iic_gene    uut_iicgene(
                .clk(clk),
                .rst_n(rst_n),
                .tiic_en(tiic_en),
                .tiic_ab(tiic_ab),
                .tiic_db(tiic_db)
            );
```

```
//-----
```

```
//视频输入缓存控制
```

```
video_ctrl  uut_videoctrl(
                .clk(clk_100m),
                .rst_n(rst_n),
                .vpclk(vpclk),
                .vvsync(vvsync),
                .vhref(vhref),
                .vdb(vdb),
                .wrf_din(wrf_din),
                .wrf_wrreq(wrf_wrreq)
            );
```



```
endmodule
```

8.6.5 工程移植

除了前面提到的一些模块的详细设计外, 这个工程其实是移植了 ex16 工程的 LCD 组件设计。对于 sdfifo_ctrl 模块, 需要适当的做一些修改, 原先送到 SDRAM 写缓存 FIFO 的接口, 不需要专门的地址总线, 而只是不断的送出数据和写使能信号, 同时有一个复位信号做同步。由于视频数据流的每一帧数据流都是很固定的, 所以我们可以根据每一帧的复位信号做同步, 在每次写入一页的 SDRAM 数据后, SDRAM 写入的地址也不断的递增。在读 SDRAM 端, LCD 显示的画面也是固定的, 以同样的方式产生地址去读 SDRAM, 然后送往 LCD 显示。

```
module sdfifo_ctrl(
    clk_25m, clk_100m, rst_n,
    wrf_clr, wrf_din, wrf_wrreq,
    sdram_wr_ack, sys_wraddr, sys_rdaddr, sys_data_in, sdram_wr_req,
    sys_data_out, sdram_rd_ack, sdram_rd_req,
    rdfifo_rdreq, rdfifo_clr, rdfifo_rddb
);

input clk_25m; //PLL 输出 25MHz 时钟
input clk_100m; //PLL 输出 100MHz 时钟
input rst_n; //系统复位信号, 低有效
    //wrFIFO 控制
input wrf_clr; //sdram 数据写入缓存 FIFO 复位信号, 高电平有效
input wrf_wrreq; //sdram 数据写入缓存 FIFO 数据输入请求, 高有效
input[15:0] wrf_din; //sdram 数据写入缓存 FIFO 输入数据总线
input sdram_wr_ack; //系统写 SDRAM 响应信号, 作为 wrFIFO 的输出有效信号
output sdram_wr_req; //系统写 SDRAM 请求信号
output[15:0] sys_data_in; //sdram 数据写入缓存 FIFO 输出数据总线, 即写 SDRAM 时
数据暂存器
output[21:0] sys_wraddr; // 写 SDRAM 时地址暂存器, (bit21-20)L-Bank 地址: (bit19-8)为行地址, (bit7-0)为列地址
    //rdFIFO 写入控制
input sdram_rd_ack; //系统读 SDRAM 响应信号, 作为 rdFIFO 的输写有效信号
input[15:0] sys_data_out; //sdram 数据读出缓存 FIFO 输入数据总线
```



```
output sdram_rd_req;          //系统读 SDRAM 请求信号
output[21:0] sys_rdaddr;      // 读 SDRAM 时地址暂存器, (bit21-20)L-Bank 地址: (bit19-8)为行地址, (bit7-0)为列地址
    //rdFIFO 读出控制
input rdfifo_rdreq;           //sdram 数据读出缓存 FIFO 数据输出请求, 高有效
input rdfifo_clr;             //高有效, 用于使能 SDRAM 读数据单元进行寻址或地址清零
output[15:0] rdfifo_rddb;     //VGA 显示数据

//-----
//sdram 读写响应完成标致捕获
reg sdwrackr1, sdwrackr2;     //sdram_wr_ack 寄存器
reg sdrdackr1, sdrdackr2;     //sdram_rd_ack 寄存器

    //锁存两拍 sdram_wr_ack, 用于下降沿捕获
always @(posedge clk_100m or negedge rst_n)
    if(!rst_n) begin
        sdwrackr1 <= 1'b0;
        sdwrackr2 <= 1'b0;
    end
    else begin
        sdwrackr1 <= sdram_wr_ack;
        sdwrackr2 <= sdwrackr1;
    end

wire neg_sdwrack = ~sdwrackr1 & sdwrackr2; //sdram_wr_ack 下降沿标志位, 高有效
一个时钟周期

    //锁存两拍 sdram_rd_ack, 用于下降沿捕获
always @(posedge clk_100m or negedge rst_n)
    if(!rst_n) begin
        sdrdackr1 <= 1'b0;
        sdrdackr2 <= 1'b0;
    end
    else begin
        sdrdackr1 <= sdram_rd_ack;
        sdrdackr2 <= sdrdackr1;
    end
end
```



```
wire neg_sdrdack = ~sdrdackr1 & sdrdackr2; //sdrdack 下降沿标志位, 高有效
一个时钟周期

//-----

reg rdfifo_clrr;          //将 50M 时钟域的 rdfifo_clr 打一拍以同步到 100M 的
rdfifo_clrr

always @(posedge clk_100m or negedge rst_n)
    if(!rst_n) rdfifo_clrr <= 1'b0;
    else rdfifo_clrr <= rdfifo_clr;

//-----

reg wrf_clr_r;

always @(posedge clk_100m or negedge rst_n)
    if(!rst_n) wrf_clr_r <= 1'b1;
    else wrf_clr_r <= wrf_clr;

//-----

//读写 sdrdack 请求信号产生
wire[8:0] wrf_use;        //sdrdack 数据写入缓存 FIFO 已用存储空间数量
wire[8:0] rdf_use;        //sdrdack 数据读出缓存 FIFO 已用存储空间数量

assign sdrdack_wr_req = (wrf_use > 9'd158); //FIFO (160 个 16bit 数据) 即发出写
SDRAM 请求信号
assign sdrdack_rd_req = (rdf_use < 9'd350) & ~rdfifo_clrr; //VGA 显示有效且
FIFO 不满 350 个数据即发出读 SDRAM 请求信号

//-----

//sdrdack 写地址产生逻辑
reg[13:0] sys_wrabr;      //SDRAM 写数据页

//sdrdack 写地址产生
always @(posedge clk_100m or negedge rst_n)
    if(!rst_n) sys_wrabr <= 14'd0;
    else if(wrf_clr_r) sys_wrabr <= 14'd0; //从起始地址写数据
    else if(neg_sdrdack) sys_wrabr <= sys_wrabr+1'b1; //一次写入完成后地址递
增
```




```
assign sys_wraddr = {sys_wrabr, 8'd0};

//-----
//sdr 读地址产生逻辑
reg[13:0] sys_rdabr;    //SDRAM 读数据页

    //sdr 读地址产生
always @(posedge clk_100m or negedge rst_n)
    if(!rst_n) sys_rdabr <= 14'd0;
    else if(rdfifo_clrr) sys_rdabr <= 14'd0;    //从起始地址读数据
    else if(neg_sdrdack) sys_rdabr <= sys_rdabr+1'b1;    //一次读出完成后地址递
增

assign sys_rdaddr = {sys_rdabr, 8'd0};

//-----
//例化 SDRAM 写入数据缓存 FIFO 模块
wrfifo      uut_wrfifo(
    .aclr(wrf_clr_r),
    .data(wrf_din),
    .rdclk(clk_100m),
    .rdreq(sdr_wr_ack),
    .wrclk(clk_100m),
    .wrreq(wrf_wrreq),
    .q(sys_data_in),    //写入地址和数据输出
    .wrusedw(wrf_use)
);

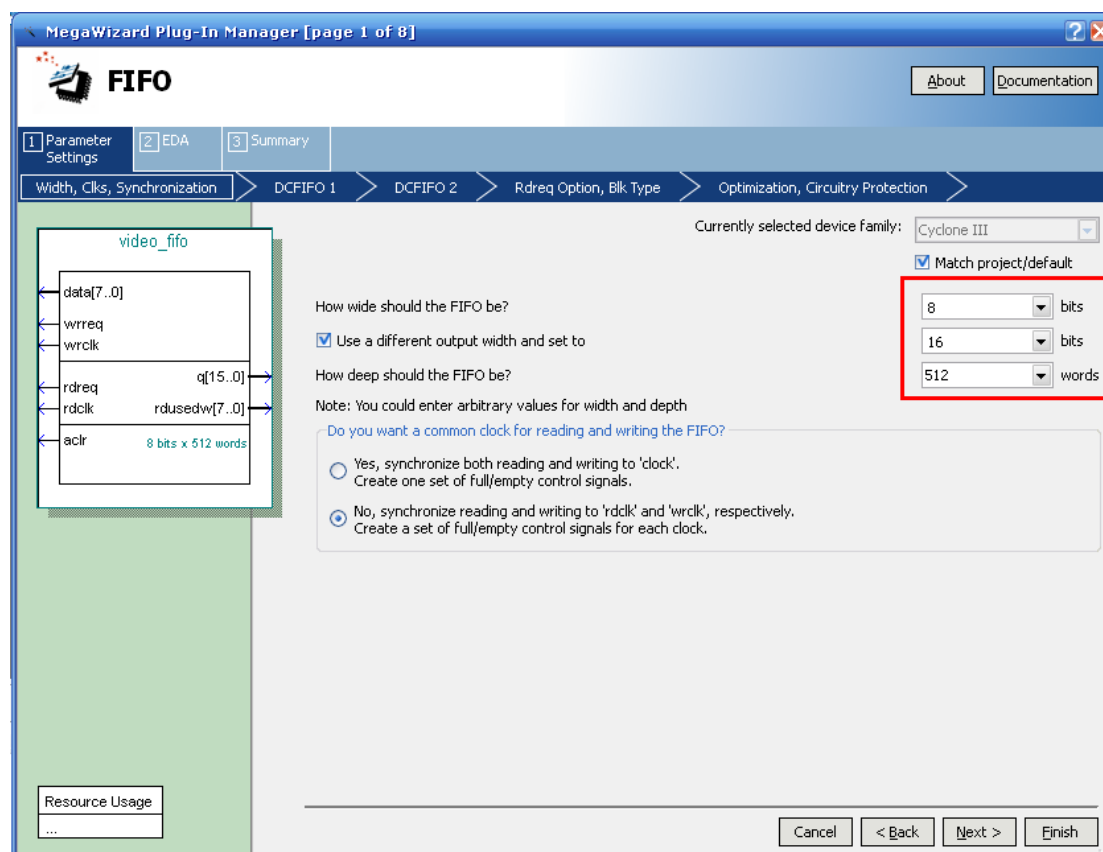
//-----
//例化 SDRAM 读出数据缓存 FIFO 模块
rdfifo      uut_rdfifo(
    .aclr(rdfifo_clrr),
    .data(sys_data_out),
    .rdclk(clk_25m),
    .rdreq(rdfifo_rdreq),
    .wrclk(clk_100m),
    .wrreq(sdr_rd_ack),
```



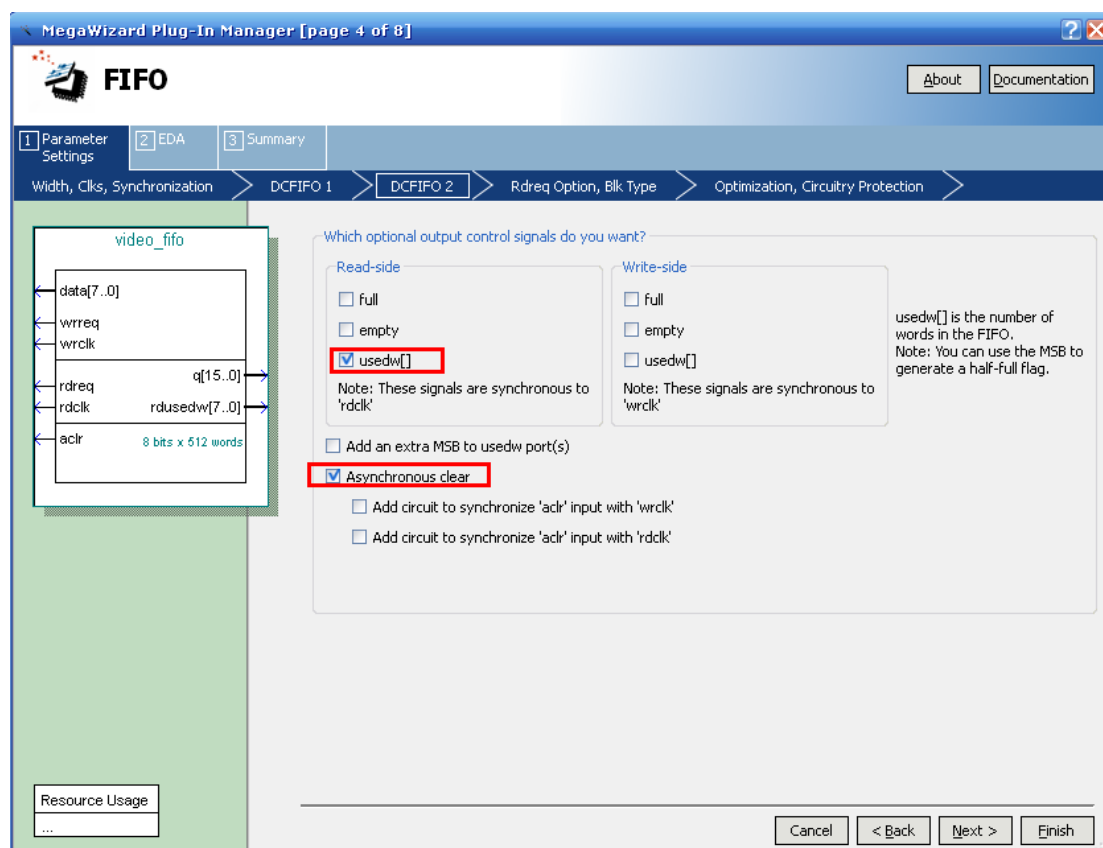
```
.q(rdfifo_rddb),  
.wrusedw(rdf_use)  
);
```

endmodule

rdfifo 的配置如下（创建方式这里不详细描述，可以参考其他 FIFO 的创建）。写入数据位宽为 8bits，读出数据位宽为 16bits，数据深度为 512words。在读出数据的时候，我们便将原先两个时钟周期送出的 16bits 数据合并到一起作为一个完整像素点的色彩数据。



在 DCFIFO2 页面，设置 Read-side 有 usedw[]信号，以及勾选 Asynchronous clear，其他设置采用默认值即可。



复制 ex16 工程的整个目录, 更名为 ex21, 根据前面给出的代码, 创建 Verilog 代码文件 sdfifo_ctrl.v、video_input.v、iic_ctrl.v、iic_gene.v、video_ctrl.v, 同时继续使用 ex16 工程原有的代码 lcd_driver.v、sys_ctrl.v、sdram_top.v 以及其下的 3 个子模块 (还包括一个参数定义 sdr_para.v 模块)。

顶层模块重新对各个模块进行例化, 如下。

```
module ex2(
    clk, rst_n,
    lcd_en, lcd_clk, lcd_hsy, lcd_vsy, lcd_db_r, lcd_db_g, lcd_db_b,
    sdram_clk, sdram_cke, sdram_cs_n, sdram_ras_n, sdram_cas_n, sdram_we_n,
    sdram_ba, sdram_addr, sdram_data,
    vpcclk, vvsync, vhref, vdb, vxclk, vscl, vsda
);
input clk;
input rst_n;
    // FPGA 与 LCD 接口信号
output lcd_en; //背光使能信号, 高有效
output lcd_clk; //时钟信号
output lcd_hsy; //行同步信号
```



```
output lcd_vsy; //场同步信号
output[4:0] lcd_db_r;
output[5:0] lcd_db_g;
output[4:0] lcd_db_b;

    // FPGA 与 SDRAM 硬件接口
output sdram_clk;           // SDRAM 时钟信号
output sdram_cke;           // SDRAM 时钟有效信号
output sdram_cs_n;          // SDRAM 片选信号
output sdram_ras_n;         // SDRAM 行地址选通脉冲
output sdram_cas_n;         // SDRAM 列地址选通脉冲
output sdram_we_n;          // SDRAM 写允许位
output[1:0] sdram_ba;        // SDRAM 的 L-Bank 地址线
output[11:0] sdram_addr;     // SDRAM 地址总线
inout[15:0] sdram_data;      // SDRAM 数据总线

    //视频输入接口
input vpclk;                //视频时钟
input vvsync;               //视频场同步信号
input vhref;                //视频行同步信号
input[7:0] vdb;             //视频数据总线
output vxclk;               //视频驱动时钟
output vscl;                //串行配置 IIC 时钟信号
inout vsda;                 //串行配置 IIC 数据信号

//-----
    //LCD 与 FIFO 的接口
wire[15:0] rdfifo_rddb;      //FIFO 读出数据总线
wire rdfifo_rdreq;          //FIFO 读请求信号
wire rdfifo_clr;            //FIFO 复位信号, 高电平有效

    // SDRAM 的封装接口
wire sdram_wr_req;          //系统写 SDRAM 请求信号
wire sdram_rd_req;          //系统读 SDRAM 请求信号
wire sdram_wr_ack;          //系统写 SDRAM 响应信号, 作为 wrFIFO 的输出有效信号
wire sdram_rd_ack;          //系统读 SDRAM 响应信号, 作为 rdFIFO 的输写有效信号

wire[8:0] sdwr_byte = 9'd160; //突发写 SDRAM 字节数 (1-256 个)
wire[8:0] sdrd_byte = 9'd160; //突发读 SDRAM 字节数 (1-256 个)
wire[21:0] sys_wraddr;       // 写 SDRAM 时地址暂存器, (bit21-20)L-Bank 地址: (bit19-8)为行地址, (bit7-0)为列地址
```



```
wire[21:0] sys_rdaddr;    // 读 SDRAM 时地址暂存器, (bit21-20)L-Bank 地址: (bit19-8)为行地址, (bit7-0)为列地址
wire[15:0] sys_data_in;   //写 SDRAM时数据暂存器
wire[15:0] sys_data_out;  //sdrn 数据读出缓存 FIFO 输入数据总线
    //wrFIFO 输入控制接口
wire[15:0] wrf_din;       //sdrn 数据写入缓存 FIFO 输入数据总线
//wire[21:0] wrf_ain;      //sdrn 数据写入缓存 FIFO 输入地址总线
wire wrf_wrreq;           //sdrn 数据写入缓存 FIFO 数据输入请求, 高有效
    //系统控制相关信号接口
wire clk_25m;             //PLL 输出 25MHz 时钟
wire clk_50m;             //PLL 输出 50MHz 时钟
wire clk_100m;            //PLL 输出 100MHz 时钟
wire sys_rst_n;           //系统复位信号, 低有效

assign vxclk = clk_25m;    //25MHz 输出给 OV7670

//-----
//例化系统复位信号和 PLL 控制模块
sys_ctrl      uut_sysctrl(
    .clk(clk),
    .rst_n(rst_n),
    .sys_rst_n(sys_rst_n),
    .clk_25m(clk_25m),
    .clk_50m(clk_50m),
    .clk_100m(clk_100m),
    .sdrn_clk(sdrn_clk)
);

//-----
//视频采集控制模块
video_input    uut_videoinput(
    .clk(clk_25m),
    .clk_100m(clk_100m),
    .rst_n(sys_rst_n),
    .vpclk(vpclk),
    .vvsync(vvsync),
    .vhref(vhref),
    .vdb(vdb),
```



```
        .vscl(vscl),
        .vsda(vsda),
        .wrf_din(wrf_din),
        .wrf_wrreq(wrf_wrreq)
    );

//-----

//LCD 驱动模块
lcd_driver uut_lcd_driver(
    .clk(clk_25m), //25MHz
    .rst_n(sys_rst_n),
    .lcd_en(lcd_en),
    .lcd_clk(lcd_clk),
    .lcd_hsy(lcd_hsy),
    .lcd_vsy(lcd_vsy),
    .lcd_db_r(lcd_db_r),
    .lcd_db_g(lcd_db_g),
    .lcd_db_b(lcd_db_b),
    .rdfifo_rddb(rdfifo_rddb),//
    .rdfifo_rdreq(rdfifo_rdreq),//
    .rdfifo_clr(rdfifo_clr)//
);

//-----

//例化 SDRAM 封装控制模块
sdram_top uut_sdramtop( // SDRAM
    .clk(clk_100m),
    .rst_n(sys_rst_n),
    .sdram_wr_req(sdram_wr_req),//
    .sdram_rd_req(sdram_rd_req),//
    .sdram_wr_ack(sdram_wr_ack),//
    .sdram_rd_ack(sdram_rd_ack),//
    .sys_wraddr(sys_wraddr),//
    .sys_rdaddr(sys_rdaddr),//
    .sys_data_in(sys_data_in),//
    .sys_data_out(sys_data_out),//
    .sdwr_byte(sdwr_byte),//
    .sdrd_byte(sdrd_byte),//
);
```



```
        .sdram_cke(sdram_cke),
        .sdram_cs_n(sdram_cs_n),
        .sdram_ras_n(sdram_ras_n),
        .sdram_cas_n(sdram_cas_n),
        .sdram_we_n(sdram_we_n),
        .sdram_ba(sdram_ba),
        .sdram_addr(sdram_addr),
        .sdram_data(sdram_data),
        .sdram_udqm(),
        .sdram_ldqm()
    );

//-----
//读写 SDRAM 数据缓存 FIFO 模块例化
sdfifo_ctrl      uut_sdffifoctrl(
    .clk_25m(clk_25m),
    .clk_100m(clk_100m),
    .rst_n(sys_rst_n),
    .wrf_clr(vvsync),
    .wrf_din(wrf_din),
    .wrf_wrreq(wrf_wrreq),
    .sdram_wr_ack(sdram_wr_ack), //
    .sys_wraddr(sys_wraddr), //
    .sys_rdaddr(sys_rdaddr), //
    .sys_data_in(sys_data_in), //
    .sdram_wr_req(sdram_wr_req), //
    .sys_data_out(sys_data_out), //
    .sdram_rd_ack(sdram_rd_ack), //
    .sdram_rd_req(sdram_rd_req), //
    .rdfifo_rdreq(rdfifo_rdreq), //
    .rdfifo_clr(rdfifo_clr), //
    .rdfifo_rddb(rdfifo_rddb) //
);

endmodule
```

对工程进行综合, 然后做管脚分配。也可以在 tcl 面板中直接输入以下的配置脚本进行管脚分配。

《圣经》箴言九 11 “敬畏耶和华是智慧的开端, 认识至胜者便是聪明。”



```
set_location_assignment PIN_22 -to clk
set_location_assignment PIN_91 -to rst_n
set_location_assignment PIN_60 -to lcd_clk
set_location_assignment PIN_77 -to lcd_db_b[4]
set_location_assignment PIN_79 -to lcd_db_b[3]
set_location_assignment PIN_80 -to lcd_db_b[2]
set_location_assignment PIN_83 -to lcd_db_b[1]
set_location_assignment PIN_84 -to lcd_db_b[0]
set_location_assignment PIN_71 -to lcd_db_g[5]
set_location_assignment PIN_72 -to lcd_db_g[4]
set_location_assignment PIN_73 -to lcd_db_g[3]
set_location_assignment PIN_74 -to lcd_db_g[2]
set_location_assignment PIN_75 -to lcd_db_g[1]
set_location_assignment PIN_76 -to lcd_db_g[0]
set_location_assignment PIN_66 -to lcd_db_r[4]
set_location_assignment PIN_67 -to lcd_db_r[3]
set_location_assignment PIN_68 -to lcd_db_r[2]
set_location_assignment PIN_69 -to lcd_db_r[1]
set_location_assignment PIN_70 -to lcd_db_r[0]
set_location_assignment PIN_85 -to lcd_en
set_location_assignment PIN_65 -to lcd_hsy
set_location_assignment PIN_64 -to lcd_vsy
set_location_assignment PIN_46 -to sdram_addr[11]
set_location_assignment PIN_135 -to sdram_addr[10]
set_location_assignment PIN_49 -to sdram_addr[9]
set_location_assignment PIN_50 -to sdram_addr[8]
set_location_assignment PIN_51 -to sdram_addr[7]
set_location_assignment PIN_52 -to sdram_addr[6]
set_location_assignment PIN_53 -to sdram_addr[5]
set_location_assignment PIN_54 -to sdram_addr[4]
set_location_assignment PIN_128 -to sdram_addr[3]
set_location_assignment PIN_129 -to sdram_addr[2]
set_location_assignment PIN_132 -to sdram_addr[1]
set_location_assignment PIN_133 -to sdram_addr[0]
set_location_assignment PIN_136 -to sdram_ba[1]
set_location_assignment PIN_137 -to sdram_ba[0]
set_location_assignment PIN_142 -to sdram_cas_n
set_location_assignment PIN_44 -to sdram_cke
```



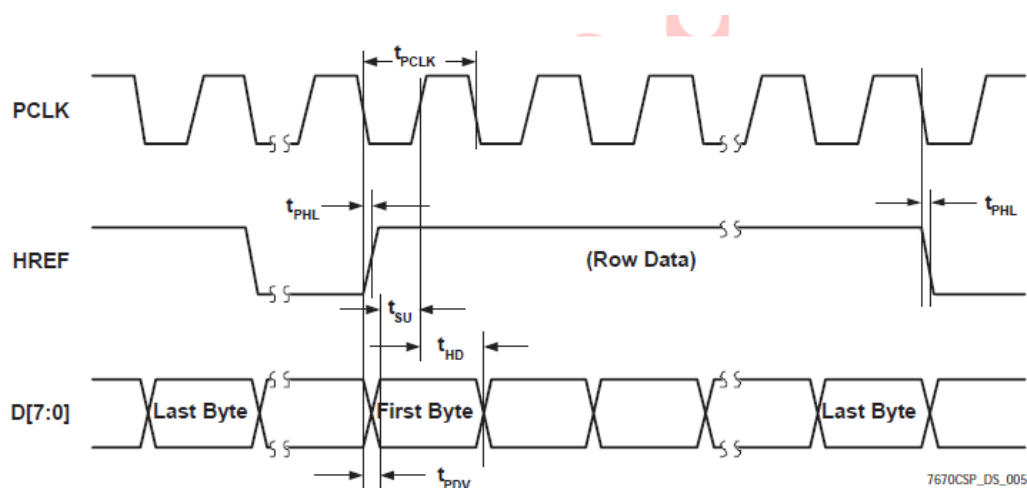

```
set_location_assignment PIN_138 -to sdram_cs_n
set_location_assignment PIN_34 -to sdram_data[15]
set_location_assignment PIN_33 -to sdram_data[14]
set_location_assignment PIN_32 -to sdram_data[13]
set_location_assignment PIN_31 -to sdram_data[12]
set_location_assignment PIN_30 -to sdram_data[11]
set_location_assignment PIN_38 -to sdram_data[10]
set_location_assignment PIN_39 -to sdram_data[9]
set_location_assignment PIN_42 -to sdram_data[8]
set_location_assignment PIN_144 -to sdram_data[7]
set_location_assignment PIN_1 -to sdram_data[6]
set_location_assignment PIN_2 -to sdram_data[5]
set_location_assignment PIN_3 -to sdram_data[4]
set_location_assignment PIN_4 -to sdram_data[3]
set_location_assignment PIN_7 -to sdram_data[2]
set_location_assignment PIN_10 -to sdram_data[1]
set_location_assignment PIN_11 -to sdram_data[0]
set_location_assignment PIN_141 -to sdram_ras_n
set_location_assignment PIN_143 -to sdram_we_n
set_location_assignment PIN_43 -to sdram_clk
set_global_assignment -name QIP_FILE video_fifo.qip
set_global_assignment -name QIP_FILE iic_initrom.qip
set_location_assignment PIN_115 -to vdb[7]
set_location_assignment PIN_119 -to vdb[6]
set_location_assignment PIN_120 -to vdb[5]
set_location_assignment PIN_121 -to vdb[4]
set_location_assignment PIN_124 -to vdb[3]
set_location_assignment PIN_125 -to vdb[2]
set_location_assignment PIN_126 -to vdb[1]
set_location_assignment PIN_127 -to vdb[0]
set_location_assignment PIN_114 -to vxclk
set_location_assignment PIN_112 -to vhref
set_location_assignment PIN_113 -to vpcclk
set_location_assignment PIN_111 -to vvsync
set_location_assignment PIN_106 -to vscl
set_location_assignment PIN_110 -to vsda
```



8.6.6 CMOS Sensor 接口时序约束

接下来,当然是要做时序约束了。由于这个工程是移植过来的,SDRAM 的时序约束已经添加好并且很好的收敛了。但是,新增加的 CMOS sensor 的接口也需要做相应的时序约束。下面我们就来探讨下它的时序该如何做约束。

先看看 CMOS Sensor 的 datasheet 中提供的时序波形和相应的建立、保持时间要求。波形如图所示。



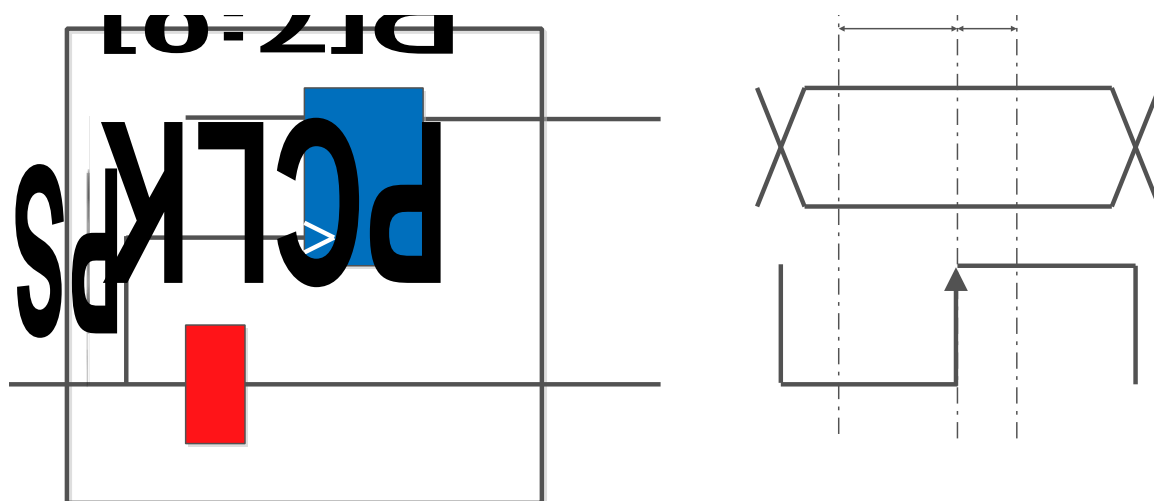
波形中出现的时间参数定义如下表所示。

Symbol	Parameter	Min	Typ	Max	Unit
Outputs (VSYNC, HREF, PCLK, and D[7:0] (see Figure 5, Figure 6, Figure 7, Figure 9, and Figure 10)					
t_{PDV}	PCLK[↓] to Data-out Valid			5	ns
t_{SU}	D[7:0] Setup time	15			ns
t_{HD}	D[7:0] Hold time	8			ns
t_{PHH}	PCLK[↓] to HREF[↑]	0		5	ns
t_{PHL}	PCLK[↓] to HREF[↓]	0		5	ns

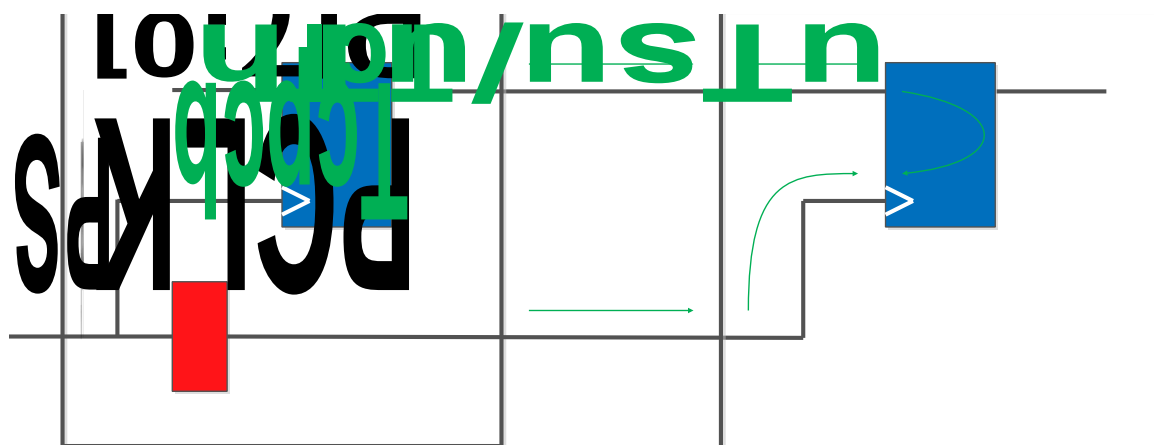
我们可以简单分析下这个 datasheet 中提供的时序波形和参数提供了一些什么样的有用信息。我们重点关注 PCLK 和 D[7:0]的关系, HREF 其实也可以归类到 D[7:0]中一起分析, 他们的时序关系基本是一致的(如果存在偏差,也可以忽略不计)。这个波形实际上表达的是从 Sensor 的芯片封装管脚上输出的 PCLK 和 D[7:0]的关系。而在理想状况下,经过 PCB 走线将这组信号连接到其他的芯片上(如 CPU 或 FPGA),若尽可能保持走线长度,在其他芯片的管脚上, PCLK 和 D[7:0]的关系基本还是不变的。那么,对于采集端来说,用 PCLK 的上升沿去锁存 D[7:0]就变得理所当然了。而对于 FPGA 而言,从它的管脚到寄存器传输路径上总归是有延时存在的,那么 PCLK 和 D[7:0]之间肯定不会是理想的对齐关系。而我们现在关心



的是, 相对于理想的对齐关系, PCLK 和 D[7:0]之间可以存在多大的相位偏差(最终可能会以一个延时时间范围来表示)。在时序图中, T_{su} 和 T_h 虽然是 PCLK 和 D[7:0]在 Sensor 内部必须保证的建立时间和保持时间关系, 但它同样是 Sensor 的输出管脚上, 必须得到保证的基本时序关系。因此, 我们可以认为: 理想相位关系情况下, PCLK 上升沿之前的 T_{su} 时间(即 15ns)到上升沿后的 T_h 时间(即 8ns)内, D[7:0]是稳定不变的。同样的, 理想情况下, PCLK 的上升沿处于 D[7:0]两次数据变化的中央。换句话说, 在 D[7:0]保持当前状态的情况下, PCLK 上升沿实际上在理想位置的 T_{su} 时间和 T_h 时间内都是允许的。请大家记住这一点, 下面我们需要利用这个信息对在 FPGA 内部的 PCLK 和 D[7:0]信号进行时序约束。



OK, 明确了 PCLK 和 D[7:0]之间应该保持的关系后, 我们再来看看他们从 CMOS Sensor 的管脚输出后, 到最终在 FPGA 内部的寄存器进行采样锁存, 这整个路径上的各种“艰难险阻”(延时)。



在这个路径分析中, 我们不去考虑 CMOS Sensor 内部的时序关系, 我们只关心它的输出



管脚上的信号。先看时钟 PCLK 的路径延时, 在 PCB 上的走线延时为 T_{pcb} , 在 FPGA 内部, 从进入 FPGA 的管脚到寄存器的时钟输入端口的延时为 T_{cl} 。再看数据 D[7:0]的延时, 在 PCB 上的走线延时为 T_{dpb} , 在 FPGA 内部的管脚到寄存器输入端口延时为 T_{p2r} 。而 FPGA 的寄存器同样有建立时间 T_{su} 和保持时间 T_h 要求, 也必须在整个路径的传输时序中予以考虑。

另外, 从前面的分析, 我们得到了 PCLK 和 D[7:0]之间应该满足的关系。那么, 为了保证 PCLK 和 D[7:0]稳定考虑的得到传输, 我们可以得到这样一个基本的关系必须满足:

对于建立时间, 有:

$$\text{Launch edge} + T_{dpb} + T_{p2r} + T_{su} < \text{latch edge} + T_{pcb} + T_{cl}$$

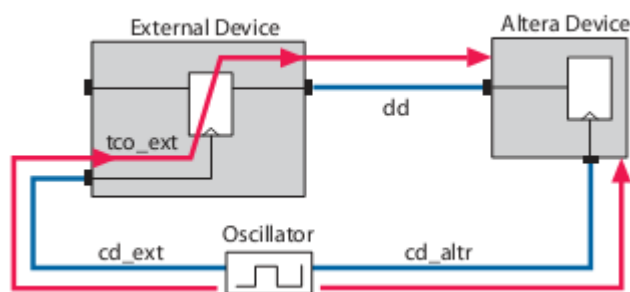
对于保持时间, 有:

$$\text{Launch edge} + T_{dpb} + T_{r2p} < \text{latch edge} + T_{pcb} + T_{cl} - T_h$$

关于 launch edge 和 latch edge, 对于我们当前的设计, 如下图所示。



在对这个 FPGA 的 input 接口的时序进行分析和约束之前, 我们先来看看 Altera 官方是如何分析此类管脚的时序。



$$\text{input delay}_{\text{MAX}} = (\text{cd_ext}_{\text{MAX}} - \text{cd_altr}_{\text{MIN}}) + \text{tco_ext}_{\text{MAX}} + \text{dd}_{\text{MAX}}$$

$$\text{input delay}_{\text{MIN}} = (\text{cd_ext}_{\text{MIN}} - \text{cd_altr}_{\text{MAX}}) + \text{tco_ext}_{\text{MIN}} + \text{dd}_{\text{MIN}}$$

具体问题具体分析, 我们当前的工程, 状况和理想模型略有区别。实际上在上面这个模型的源寄存器端的很多信息都不用详细分析, 因为我们获得的波形是来自于 Sensor 芯片的《圣经》箴言九 11 “敬畏耶和华是智慧的开端, 认识至胜者便是聪明。”



管脚。同理，我们可以得到 input delay 的计算公式如下。

$$\text{Input max delay} = (0 - \text{Tpcb_min}) + \text{Tco_max} + \text{Tdpcb_max}$$

$$\text{Input min delay} = (0 - \text{Tpcb_max}) + \text{Tco_min} + \text{Tdpcb_min}$$

在这两个公式中，参数 Tco 是前面我们还未曾提到的，下面我们就要分析下如何得到这个参数。Tco 指的是理想情况下数据在源寄存器被源时钟锁存后，经过多长时间输入到管脚上。前面我们已经得到了 PCLK 和 D[7:0] 之间的关系，其实从已知的关系中，我们不难推断出 Tco_max 和 Tco_min，如图所示。若 PCLK 的时钟周期为 Tpclk，则：

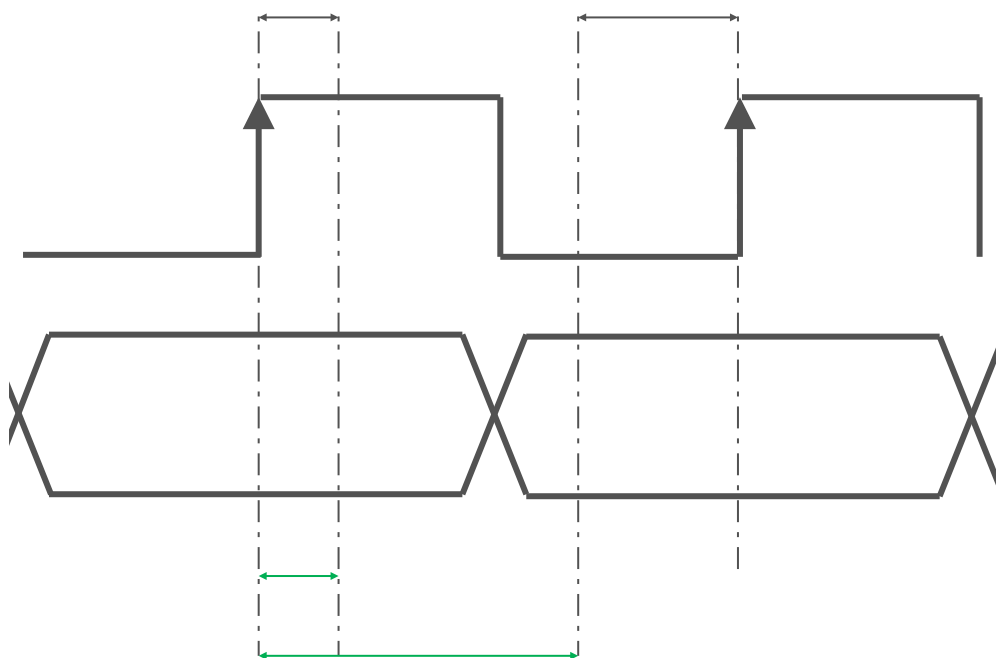
$$\text{Tco_max} = \text{Tpclk} - \text{Tsu}$$

$$\text{Tco_min} = \text{Th}$$

在我们采样的 CMOS Sensor 图像中，PCLK 频率为 12.5MHz，即 80ns。因此，我们可以计算到：

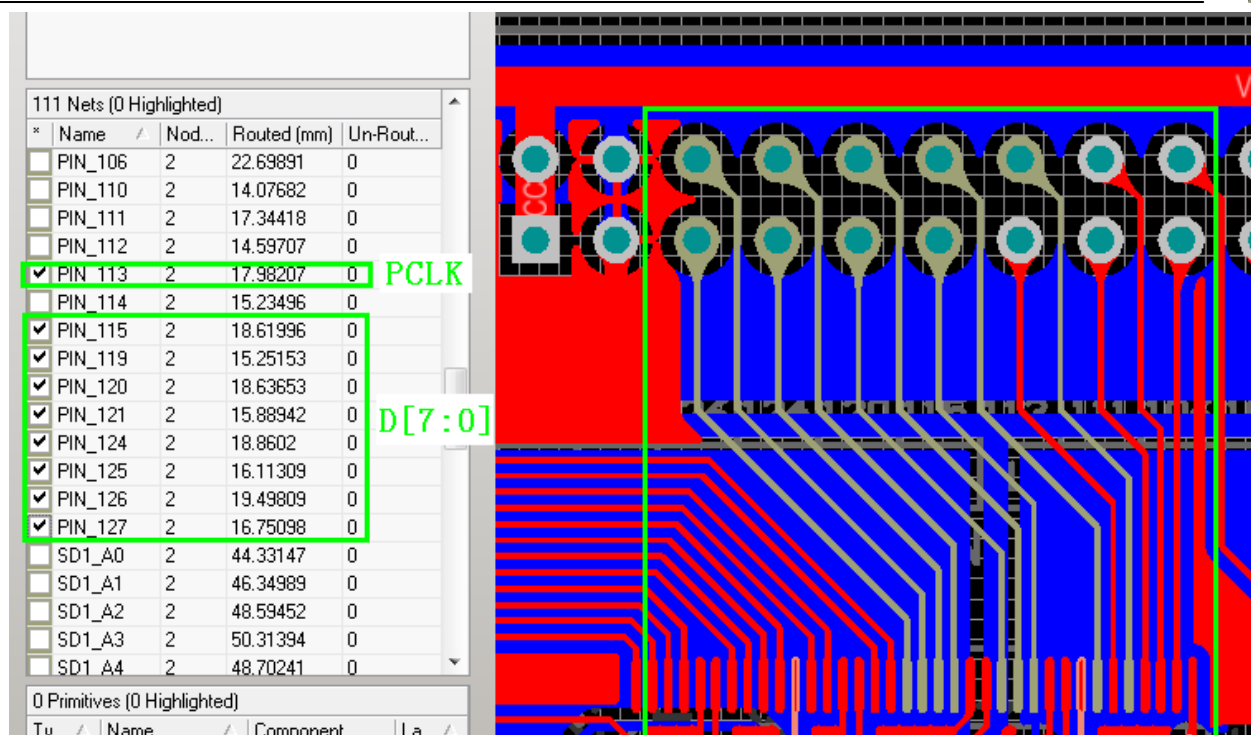
$$\text{Tco_max} = 80\text{ns} - 15\text{ns} = 65\text{ns}$$

$$\text{Tco_min} = 8\text{ns}$$

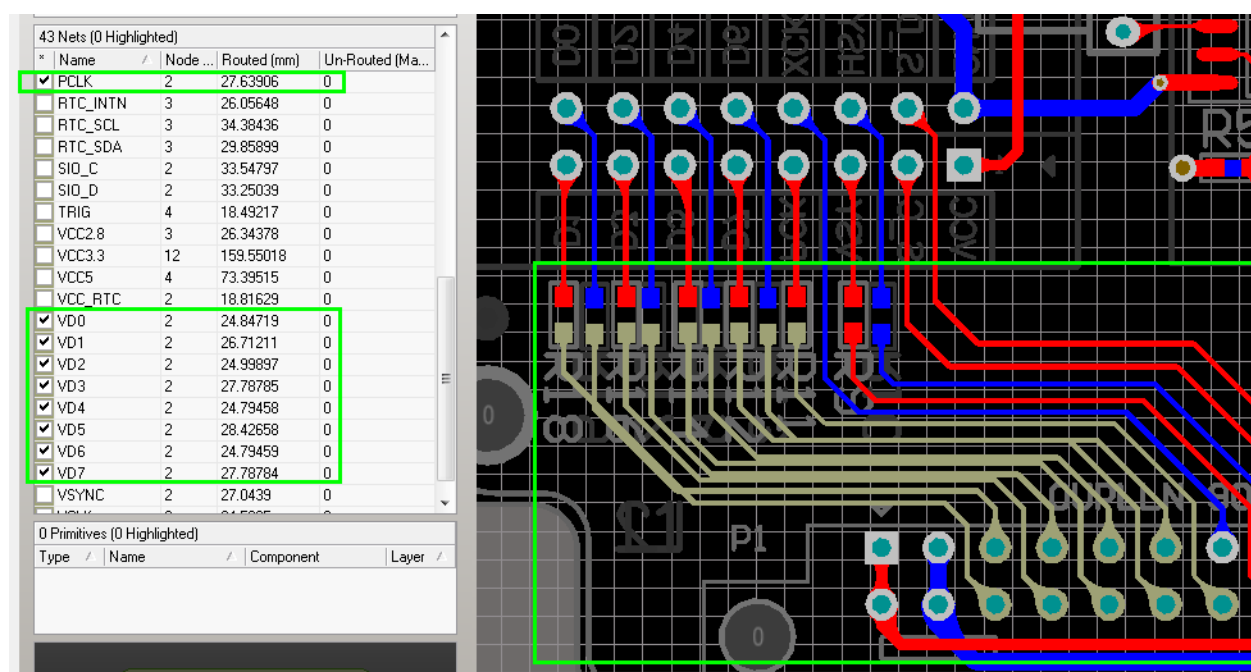


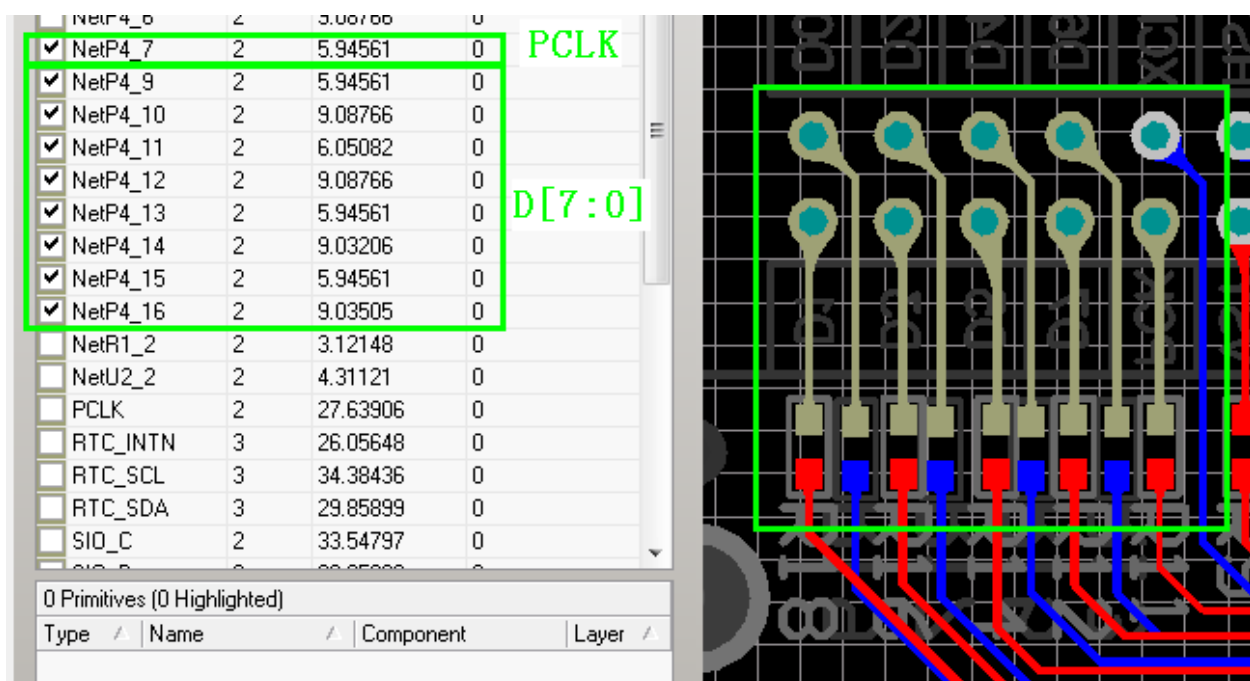
我们再看看 PCB 的走线情况，算算余下和 PCB 走线有关的延时。

如图所示，这是 PCLK 和 D[7:0] 在 SF-CY3 核心板上的走线。



如图所示, 这是 PCLK 和 D[7:0]在 SF-SENSOR 子板上的走线, 在这个板子上的走线由匹配电阻分两个部分。





根据前面的走线长度，我们可以换算一下相应的走线延时，如下表所示。因此，我们可以得到， $T_{pcb_max} = 0.35ns$, $T_{pcb_min} = 0.35ns$, $T_{dpcb_max} = 0.36ns$, $T_{dpcb_min} = 0.31ns$ 。

信号名	SF-CY3 走线长度	SF-SENSOR 走线长度 1	SF-SENSOR 走线长度 2	总长度(mm)	延时(ns)
PCLK	18	27.7	6	51.7	0.346023622
VD0	16.8	24.9	9.1	50.8	0.34
VD1	19.5	26.8	6	52.3	0.35003937
VD2	16.2	25	9.1	50.3	0.336653543
VD3	18.9	27.8	6	52.7	0.352716535
VD4	15.9	24.8	9.1	49.8	0.333307087
VD5	18.7	28.5	6	53.2	0.356062992
VD6	15.3	24.8	9.1	49.2	0.329291339
VD7	18.7	27.8	6	52.5	0.351377953
HREF	14.6	23.8	9.1	47.5	0.317913386

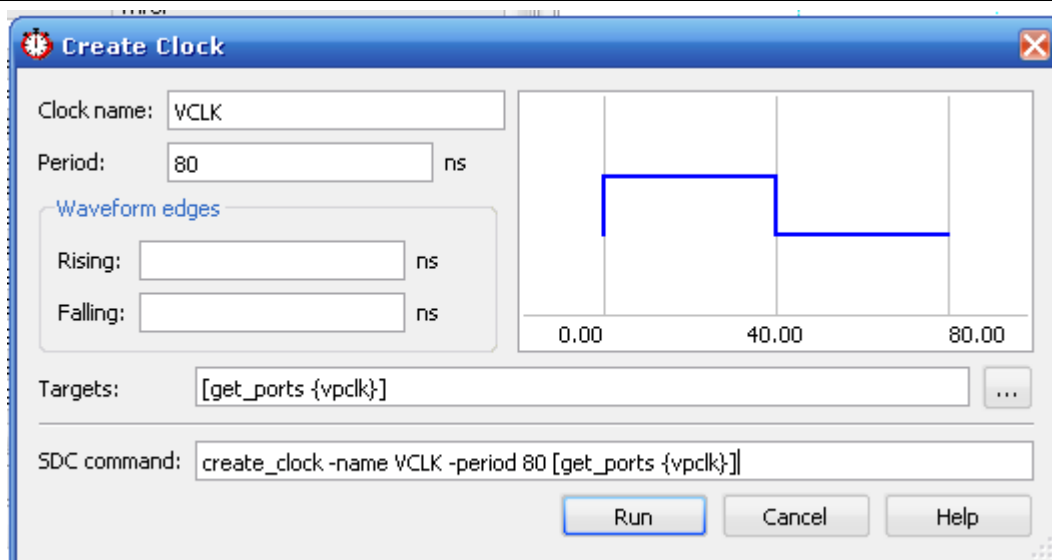
将上面得到的具体数值都代入公式，得到：

$$\text{Input max delay} = (0 - 0.35ns) + 65ns + 0.36ns = 65.01ns$$

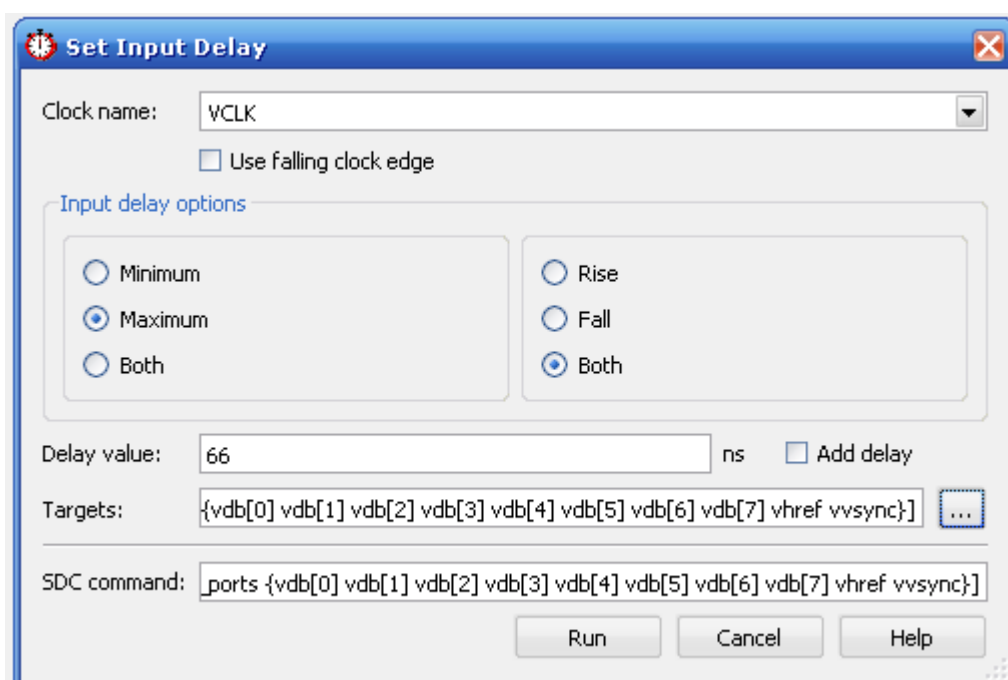
$$\text{Input min delay} = (0 - 0.35ns) + 8ns + 0.31ns = 7.96ns$$

加上一些余量，我们可以去 $\text{input max delay} = 66ns$, $\text{input min delay} = 7ns$ 。

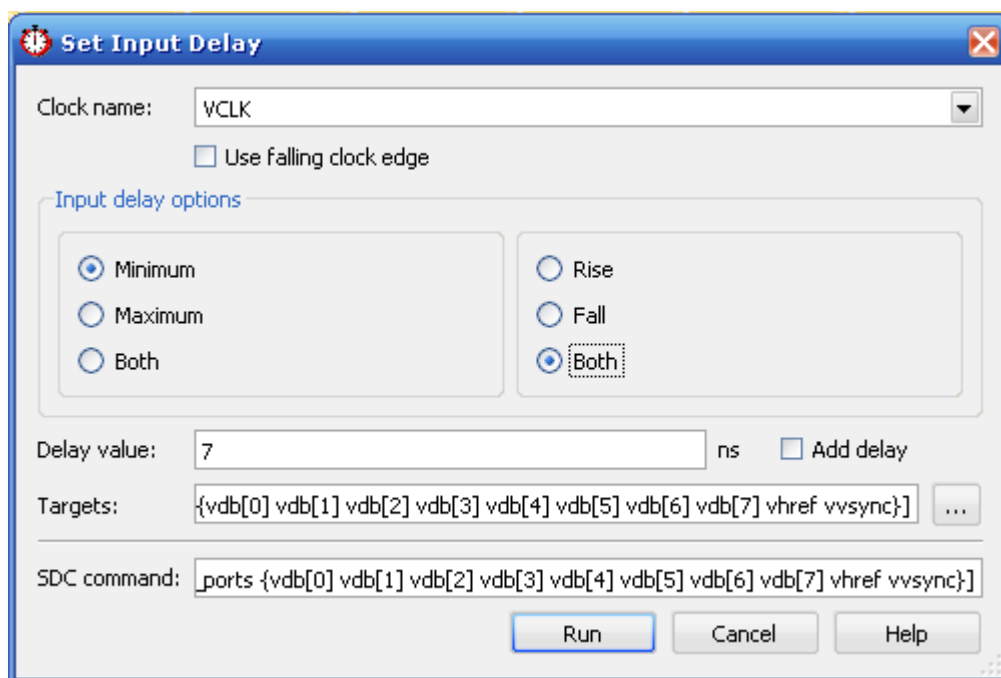
下面我们来添加时序约束，打开 TimeQuest，点击菜单栏的 Constraints→Creat Clock，做如下设置。



点击 Constraints→Set Maximum Delay, 对 vdb[0] vdb[1] vdb[2] vdb[3] vdb[4] vdb[5] vdb[6] vdb[7] vhref 的 set_max_delay 做如下设置。



点击 Constraints→Set Minimum Delay, 对 vdb[0] vdb[1] vdb[2] vdb[3] vdb[4] vdb[5] vdb[6] vdb[7] vhref 的 set_min_delay 做如下设置。



约束完成后, 参照前面章节 Update Timing Netlist 并且 Write SDC File..., 接着就可以重新编译整个工程, 再来看看这个时序分析报告。在报告中, 数据的建立时间有 9-13ns 的余量, 而保持时间也都有 7-11ns 的余量, 可谓余量充足。

Inputs to Registers (Setup)					
	Slack	From Node	To Node	Launch Clock	Latch Clock
17	9.192	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...erated a_graycounter_fic: wrptr_g1p counter10a[8]	VCLK	VCLK
18	9.192	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...erated a_graycounter_fic: wrptr_g1p counter10a[7]	VCLK	VCLK
19	9.194	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...erated a_graycounter_fic: wrptr_g1p counter10a[6]	VCLK	VCLK
20	9.278	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...erated a_graycounter_fic: wrptr_g1p sub_parity9a2	VCLK	VCLK
21	9.278	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...erated a_graycounter_fic: wrptr_g1p sub_parity9a0	VCLK	VCLK
22	9.278	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...erated a_graycounter_fic: wrptr_g1p counter10a[4]	VCLK	VCLK
23	9.278	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...to_generated a_graycounter_fic: wrptr_g1p parity8	VCLK	VCLK
24	9.278	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...erated a_graycounter_fic: wrptr_g1p counter10a[0]	VCLK	VCLK
25	9.614	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...erated a_graycounter_fic: wrptr_g1p counter10a[4]	VCLK	VCLK
26	9.769	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...erated a_graycounter_fic: wrptr_g1p counter10a[5]	VCLK	VCLK
27	10.022	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...erated a_graycounter_fic: wrptr_g1p counter10a[1]	VCLK	VCLK
28	10.218	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...component dcfifo_jd1: auto_generated wrptr_g[4]	VCLK	VCLK
29	10.218	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...component dcfifo_jd1: auto_generated wrptr_g[1]	VCLK	VCLK
30	10.218	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...auto_generated cntr_s2: cntr_b counter_reg_bit[0]	VCLK	VCLK
31	10.278	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...component dcfifo_jd1: auto_generated wrptr_g[7]	VCLK	VCLK
32	10.278	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...component dcfifo_jd1: auto_generated wrptr_g[5]	VCLK	VCLK
33	10.278	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...component dcfifo_jd1: auto_generated wrptr_g[6]	VCLK	VCLK
34	10.278	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...component dcfifo_jd1: auto_generated wrptr_g[0]	VCLK	VCLK
35	10.278	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...component dcfifo_jd1: auto_generated wrptr_g[2]	VCLK	VCLK
36	10.351	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...component dcfifo_jd1: auto_generated wrptr_g[9]	VCLK	VCLK
37	10.351	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...component dcfifo_jd1: auto_generated wrptr_g[8]	VCLK	VCLK
38	10.351	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...component dcfifo_jd1: auto_generated wrptr_g[3]	VCLK	VCLK
39	10.360	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...erated a_graycounter_fic: wrptr_g1p counter10a[2]	VCLK	VCLK
40	10.373	vhref	video_input: uut_videoinput video_ctrl: uut_video..._5h31: fifo_ram ram_block11a0~porta_address_reg0	VCLK	VCLK
41	10.373	vhref	video_input: uut_videoinput video_ctrl: uut_video..._yncram_5h31: fifo_ram ram_block11a0~porta_we_reg	VCLK	VCLK
42	10.375	vhref	video_input: uut_videoinput video_ctrl: uut_video..._m_5h31: fifo_ram ram_block11a0~porta_datain_reg0	VCLK	VCLK
43	10.615	vhref	video_input: uut_videoinput video_ctrl: uut_videooc...erated a_graycounter_fic: wrptr_g1p counter10a[3]	VCLK	VCLK
44	11.087	vhref	video_input: uut_videoinput video_ctrl: uut_video..._yncram_5h31: fifo_ram ram_block11a0~porta_we_reg	VCLK	VCLK
45	11.874	vdb[1]	video_input: uut_videoinput video_ctrl: uut_video..._m_5h31: fifo_ram ram_block11a0~porta_datain_reg0	VCLK	VCLK
46	12.012	vdb[0]	video_input: uut_videoinput video_ctrl: uut_video..._m_5h31: fifo_ram ram_block11a0~porta_datain_reg0	VCLK	VCLK
47	12.191	vdb[3]	video_input: uut_videoinput video_ctrl: uut_video..._m_5h31: fifo_ram ram_block11a0~porta_datain_reg0	VCLK	VCLK
48	12.319	vdb[2]	video_input: uut_videoinput video_ctrl: uut_video..._m_5h31: fifo_ram ram_block11a0~porta_datain_reg0	VCLK	VCLK
49	12.369	vdb[5]	video_input: uut_videoinput video_ctrl: uut_video..._m_5h31: fifo_ram ram_block11a0~porta_datain_reg0	VCLK	VCLK
50	12.393	vdb[7]	video_input: uut_videoinput video_ctrl: uut_video..._m_5h31: fifo_ram ram_block11a0~porta_datain_reg0	VCLK	VCLK
51	12.440	vdb[6]	video_input: uut_videoinput video_ctrl: uut_video..._m_5h31: fifo_ram ram_block11a0~porta_datain_reg0	VCLK	VCLK
52	12.566	vdb[4]	video_input: uut_videoinput video_ctrl: uut_video..._m_5h31: fifo_ram ram_block11a0~porta_datain_reg0	VCLK	VCLK



Inputs to Registers (Hold)					
	Slack	From Node	To Node	Launch Clock	Latch Clock
17	7.760	vdb[4]	video_input:uut_videoinput video_ctrl:uut_video...m_5h31:fifo_ram ram_block11a0~porta_datain_reg0	VCLK	VCLK
18	7.885	vdb[6]	video_input:uut_videoinput video_ctrl:uut_video...m_5h31:fifo_ram ram_block11a0~porta_datain_reg0	VCLK	VCLK
19	7.928	vdb[7]	video_input:uut_videoinput video_ctrl:uut_video...m_5h31:fifo_ram ram_block11a0~porta_datain_reg0	VCLK	VCLK
20	7.949	vdb[5]	video_input:uut_videoinput video_ctrl:uut_video...m_5h31:fifo_ram ram_block11a0~porta_datain_reg0	VCLK	VCLK
21	8.046	vdb[2]	video_input:uut_videoinput video_ctrl:uut_video...m_5h31:fifo_ram ram_block11a0~porta_datain_reg0	VCLK	VCLK
22	8.166	vdb[3]	video_input:uut_videoinput video_ctrl:uut_video...m_5h31:fifo_ram ram_block11a0~porta_datain_reg0	VCLK	VCLK
23	8.259	vdb[0]	video_input:uut_videoinput video_ctrl:uut_video...m_5h31:fifo_ram ram_block11a0~porta_datain_reg0	VCLK	VCLK
24	8.345	vdb[1]	video_input:uut_videoinput video_ctrl:uut_video...m_5h31:fifo_ram ram_block11a0~porta_datain_reg0	VCLK	VCLK
25	9.154	vhref	video_input:uut_videoinput video_ctrl:uut_video...yncram_5h31:fifo_ram ram_block11a0~porta_we_reg	VCLK	VCLK
26	9.523	vhref	video_input:uut_videoinput video_ctrl:uut_video...erated a_graycounter_fic:wrptr_glp counter10a[3]	VCLK	VCLK
27	9.685	vhref	video_input:uut_videoinput video_ctrl:uut_video...m_5h31:fifo_ram ram_block11a0~porta_datain_reg0	VCLK	VCLK
28	9.686	vhref	video_input:uut_videoinput video_ctrl:uut_video..._5h31:fifo_ram ram_block11a0~porta_address_reg0	VCLK	VCLK
29	9.686	vhref	video_input:uut_videoinput video_ctrl:uut_video...yncram_5h31:fifo_ram ram_block11a0~porta_we_reg	VCLK	VCLK
30	9.821	vhref	video_input:uut_videoinput video_ctrl:uut_video...erated a_graycounter_fic:wrptr_glp counter10a[2]	VCLK	VCLK
31	9.997	vhref	video_input:uut_videoinput video_ctrl:uut_video..._component dcfifo_jd1:auto_generated wrptr_g[9]	VCLK	VCLK
32	9.997	vhref	video_input:uut_videoinput video_ctrl:uut_video..._component dcfifo_jd1:auto_generated wrptr_g[8]	VCLK	VCLK
33	9.997	vhref	video_input:uut_videoinput video_ctrl:uut_video..._component dcfifo_jd1:auto_generated wrptr_g[3]	VCLK	VCLK
34	10.023	vhref	video_input:uut_videoinput video_ctrl:uut_video...erated a_graycounter_fic:wrptr_glp counter10a[1]	VCLK	VCLK
35	10.063	vhref	video_input:uut_videoinput video_ctrl:uut_video..._component dcfifo_jd1:auto_generated wrptr_g[7]	VCLK	VCLK
36	10.063	vhref	video_input:uut_videoinput video_ctrl:uut_video..._component dcfifo_jd1:auto_generated wrptr_g[5]	VCLK	VCLK
37	10.063	vhref	video_input:uut_videoinput video_ctrl:uut_video..._component dcfifo_jd1:auto_generated wrptr_g[6]	VCLK	VCLK
38	10.063	vhref	video_input:uut_videoinput video_ctrl:uut_video..._component dcfifo_jd1:auto_generated wrptr_g[0]	VCLK	VCLK
39	10.063	vhref	video_input:uut_videoinput video_ctrl:uut_video..._component dcfifo_jd1:auto_generated wrptr_g[2]	VCLK	VCLK
40	10.116	vhref	video_input:uut_videoinput video_ctrl:uut_video..._component dcfifo_jd1:auto_generated wrptr_g[4]	VCLK	VCLK
41	10.116	vhref	video_input:uut_videoinput video_ctrl:uut_video..._component dcfifo_jd1:auto_generated wrptr_g[1]	VCLK	VCLK
42	10.116	vhref	video_input:uut_videoinput video_ctrl:uut_video...auto_generated cntr_s2e:cntr_b counter_reg_bit[0]	VCLK	VCLK
43	10.433	vhref	video_input:uut_videoinput video_ctrl:uut_video...erated a_graycounter_fic:wrptr_glp counter10a[4]	VCLK	VCLK
44	10.457	vhref	video_input:uut_videoinput video_ctrl:uut_video...erated a_graycounter_fic:wrptr_glp counter10a[5]	VCLK	VCLK
45	10.847	vhref	video_input:uut_videoinput video_ctrl:uut_video...erated a_graycounter_fic:wrptr_glp counter10a[6]	VCLK	VCLK
46	10.848	vhref	video_input:uut_videoinput video_ctrl:uut_video...erated a_graycounter_fic:wrptr_glp counter10a[7]	VCLK	VCLK
47	10.850	vhref	video_input:uut_videoinput video_ctrl:uut_video...erated a_graycounter_fic:wrptr_glp counter10a[8]	VCLK	VCLK
48	10.993	vhref	video_input:uut_videoinput video_ctrl:uut_video...erated a_graycounter_fic:wrptr_glp sub_parity9a2	VCLK	VCLK
49	10.993	vhref	video_input:uut_videoinput video_ctrl:uut_video...erated a_graycounter_fic:wrptr_glp sub_parity9a0	VCLK	VCLK
50	10.993	vhref	video_input:uut_videoinput video_ctrl:uut_video...erated a_graycounter_fic:wrptr_glp sub_parity9a1	VCLK	VCLK
51	10.993	vhref	video_input:uut_videoinput video_ctrl:uut_video...to_generated a_graycounter_fic:wrptr_glp parity8	VCLK	VCLK
52	10.993	vhref	video_input:uut_videoinput video_ctrl:uut_video...erated a_graycounter_fic:wrptr_glp counter10a[0]	VCLK	VCLK

另外,我们也可以专门找一条路径出来,看看它的具体时序路径的分析。vdb[0]这条数据线的建立时间报告中,66ns 的 input max delay 出现在了 Data Arrival Path 中。

Path #1: Setup slack is 12.012

Path Summary

Statistics

Data Path

Waveform

Extra Filter Information

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Location	Element
1	0.000	0.000					launch edge time
2	0.000	0.000					clock path
1	0.000	0.000	R				clock network delay
3	66.000	66.000	F	iExt	1	PIN_127	vdb[0]
4	71.283	5.283					data path
1	66.000	0.000	FF	IC	1	IOIBUF_X16_Y24_N8	vdb[0]~input i
2	66.927	0.927	FF	CELL	1	IOIBUF_X16_Y24_N8	vdb[0]~input o
3	71.196	4.269	FF	IC	1	M9K_X27_Y20_N0	uut_videoinput uut_videoctrl uut_videofifo dc
4	71.283	0.087	FF	CELL	0	M9K_X27_Y20_N0	video_input:uut_videoinput video_ctrl:uut_vic

III

Data Required Path

	Total	Incr	RF	Type	Fanout	Location	Element
1	80.000	80.000					latch edge time
2	83.227	3.227					clock path
1	83.227	3.227	R				clock network delay
3	83.295	0.068		uTsu	0	M9K_X27_Y20_N0	video_input:uut_videoinput video_ctrl:uut_vic

而在 vdb[0]的保持时间报告中,7ns 的 input min delay 则出现在了 Data Arrival Path 中。



Path #1: Hold slack is 8.259

Path Summary

Statistics

Data Path

Waveform

Extra Filter Information

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Location	Element
1	0.000	0.000					launch edge time
2	0.000	0.000					clock path
1	0.000	0.000	R				clock network delay
3	7.000	7.000	R	iExt	1	PIN_127	vdb[0]
4	11.850	4.850					data path
1	7.000	0.000	RR	IC	1	IOIBUF_X16_Y24_N8	vdb[0]~input i
2	7.877	0.877	RR	CELL	1	IOIBUF_X16_Y24_N8	vdb[0]~input o
3	11.772	3.895	RR	IC	1	M9K_X27_Y20_N0	uut_videoinput uut_videoctrl uut_videofifo de
4	11.850	0.078	RR	CELL	0	M9K_X27_Y20_N0	video_input:uut_videoinput video_ctrl:uut_vic

<

III

>

Data Required Path

	Total	Incr	RF	Type	Fanout	Location	Element
1	0.000	0.000					latch edge time
2	3.337	3.337					clock path
1	3.337	3.337	R				clock network delay
3	3.591	0.254		uTh	0	M9K_X27_Y20_N0	video_input:uut_videoinput video_ctrl:uut_vic

8.6.7 板级调试

这个,是最 easy 的,连接好 SF-CY3 和 SF-LCD、SF-SENSOR 模块 (CMOS Sensor 可别忘记安装,方向更别弄错了),连接 USB Blaster,上电。下载 sof 文件。看效果。





附录 A 实例与工程映射

实例名	硬件目录	软件目录	硬件平台
5.1 逻辑 (Verilog) 实例 1——LED 闪烁	...prj\ex0	—	SF-CY3
5.2 逻辑 (Verilog) 实例 2——PLL 配置	...prj\ex1	—	SF-CY3
5.3 基于 Qsys 的 NIOS II 实例 1——LED 闪烁	...prj\ex2	...prj\ex2\software\ex2swprj	SF-CY3
5.4 基于 Qsys 的 NIOS II 实例 2——Hello NIOS II	...prj\ex2	...prj\ex2\software\jtaguart_swprj	SF-CY3
5.5 基于 Qsys 的 NIOS II 实例 3——集成 SDRAM 外设	...prj\ex3	—	SF-CY3
6.2 逻辑 (Verilog) 实例 3——PWM 驱动蜂鸣器	...prj\ex4	—	SF-CY3, SF-BASE
6.3 逻辑 (Verilog) 实例 4——流水灯	...prj\ex5	—	SF-CY3, SF-BASE
6.4 逻辑 (Verilog) 实例 5——模式流水灯	...prj\ex6	—	SF-CY3, SF-BASE
6.5 逻辑 (Verilog) 实例 6——数码管显示	...prj\ex7	—	SF-CY3, SF-BASE
6.6 逻辑 (Verilog) 实例 7——基于 In-System Sources and Probes Editor 的 AD 采集	...prj\ex8	—	SF-CY3, SF-BASE
6.7 逻辑 (Verilog) 实例 8——基于 In-System Sources and Probes Editor 的 DA 采集	...prj\ex9	—	SF-CY3, SF-BASE



6.8 基于 Qsys 的 NIOS II 实例 4——PIO 中断控制	...prj\ex10	...prj\ex10\ software\pio_prj	SF-CY3, SF-BASE
6.9 基于 Qsys 的 NIOS II 实例 5——数码管定时器中断	...prj\ex11	...prj\ex11\ software\seg7_prj	SF-CY3, SF-BASE
6.10 基于 Qsys 的 NIOS II 实例 6——AD/DA 组件	...prj\ex12	...prj\ex11\ software\adda_prj	SF-CY3, SF-BASE
7.2 逻辑 (Verilog) 实例 9——LCD 的基本驱动	...prj\ex13	—	SF-CY3, SF-LCD
7.3 逻辑 (Verilog) 实例 10——LCD 的 32 级红色显示	...prj\ex14	—	SF-CY3, SF-LCD
7.4 逻辑 (Verilog) 实例 11——基于 FPGA 内嵌 RAM 的 LCD 字符显示	...prj\ex15	—	SF-CY3, SF-LCD
7.5 逻辑 (Verilog) 实例 12——基于 In-System Memory Content Editor 的 LCD 实时显示字符更改	...prj\ex15	—	SF-CY3, SF-LCD
7.6 基于 Qsys 的 NIOS II 实例 7——Qsys 的 LCD 组件设计	...prj\ex16	...prj\ex16\ software\mylcd_prj	SF-CY3, SF-LCD
8.2 基于 Qsys 的 NIOS II 实例 8——SPI 接口字库芯片控制	...prj\ex17	...prj\ex17\ software\ex17swprj	SF-CY3, SF-LCD,SF-SENSOR
8.3 基于 Qsys 的 NIOS II 实例 9——IIC 接口实时时钟 (RTC) 芯片控制	...prj\ex18	...prj\ex18\ software\ex18swprj	SF-CY3, SF-LCD,SF-SENSOR
8.4 逻辑 (Verilog) 实例 13——超声波测距数据采集	...prj\ex19	—	SF-CY3, SF-LCD,SF-SENSOR
8.5 基于 Qsys 的 NIOS II 实例 10——超声波测距换算	...prj\ex20	...prj\ex20\ software\ex20swprj	SF-CY3, SF-LCD,SF-SENSOR
8.6 逻辑 (Verilog) 实例 14——基于 CMOS Sensor 的视频采集显示	...prj\ex21	—	SF-CY3, SF-LCD,SF-SENSOR



附录 B 套件淘宝购买链接

套件名称	主要配件	价格	淘宝链接
SF-CY3 空 PCB	SF-CY3 空 PCB 一块 资料光盘一张	RMB35	http://item.taobao.com/item.htm?spm=0.0.0.31.VcOHcT&id=22439740771
SF-CY3 单板	SF-CY3 焊接好的板子一块 资料光盘一张	RMB168	http://item.taobao.com/item.htm?spm=686.1000925.1000774.6.2gQsNr&id=17149278627
SF-CY3 套件	SF-CY3 焊接好的板子一块 资料光盘一张 5V/1A 电压源一个 USB-Blaster 下载线一条	RMB228	http://item.taobao.com/item.htm?spm=0.0.0.31.DRIlhF&id=22439872377
SF-BASE 子模 块	SF-BASE 焊接好的板子一块 资料光盘一张	RMB99	http://item.taobao.com/item.htm?spm=686.1000925.1000774.5.Se3V7t&id=18922263863
SF-LCD 子模块	SF-LCD 电路板一块 3.5 寸液晶屏一块 资料光盘一张 3M 双面胶一片	RMB 139	http://item.taobao.com/item.htm?spm=0.0.0.0.4PtYiv&id=23699188598



SF-SENSOR 子 模块	SF-SENSOR 电路板一块 CMOS Senosr 模块一块 超声波测距模块一块 资料光盘一张 3V 纽扣电池一颗	RMB 199	http://item.taobao.com/item.htm?spm=0.0.0.0.WYVetx&id=17838581080
-------------------	---	---------	---