第 5 章 FPGA 数字系统综合专题

CHAPTER 5

5.1 专题一：频率计设计与实现

1. 专题介绍

完成频率计设计，并在 Atlys 开发板上实现和验证。

2. 专题目标

进一步熟悉和掌握数字系统设计的一般方法和流程。

了解应用于数字系统的 UART 串口。

熟悉 PicoBlaze 硬件，设计 PicoBlaze 输入输出和中断，掌握 PicoBlaze 汇编代码的写法。

掌握有限状态机的设计与实现方法。

3. 专题步骤

(1) 明确频率计的设计目标。

(2) 顶层模块设计，即明确顶层模块应包含哪些功能子模块。

(3) 各子模块设计。

(4) 各部分代码设计。

(5) 综合、实现并下载验证。

5.1.1 明确设计目标

本专题的目标是设计并实现一个频率计，它测量被测信号(由 FPGA 内部的 Signal_Gen 模块产生)的频率，而将频率值由 UART 串口传送到 PC，由 PC 进行显示。

图 5-1 是以上设想的示意图。可见，图中问号部分就是我们设计和实现的主要内容。

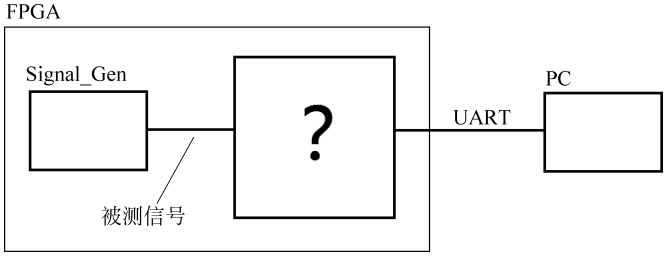

图 5-1

5.1.2　顶层模块设计

数字逻辑系统的设计一般采用自顶向下的设计方法。首先设计顶层模块，将顶层模块中的分模块功能和它们之间的连接关系、信号流向确定下来；再针对各分模块进行细化设计，直到细化后的模块已经比较简单而易实现；最后，实现各个模块、进行装配并进入调试环节。

图 5-2 给出了频率计实现的一个方案(顶层模块)。

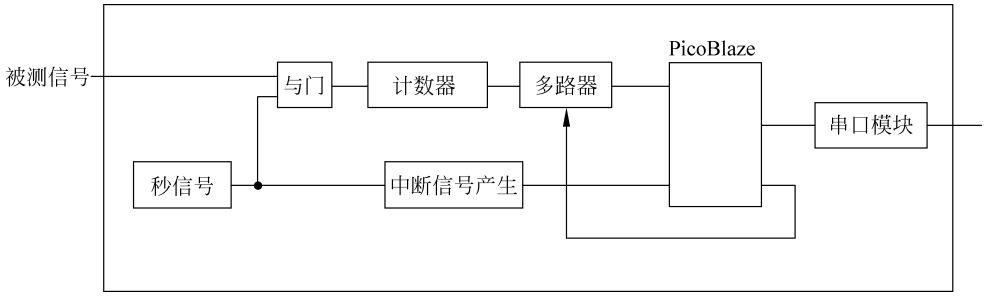

图　5-2

对这个框图的理解，应从理顺一些主要信号的因果关系开始。至于信号的时序细节，在这一阶段只能了解得比较粗略，各分模块的信号时序还有待细化(一般在分模块的设计中使用时序图进行表达)。下面将频率计方案的主要信号梳理一下。

图 5-2 中，由中断信号产生模块获取秒信号的下降沿(也就是用于显示的 0.1s 的起始时刻)，并产生一个不小于两个时钟周期宽度的脉冲，送至 PicoBlaze 的中断输入。这引起 PicoBlaze 中断，在中断处理程序中，PicoBlaze 通过读端口读取计数器产生的 5 个 BCD 码。

计数器产生的个十百千万共 5 位十进制数，其 BCD 码一共是 20 位(bit)。PicoBlaze 的输入端口只有 8 位，无法一次读入 20 位数据。因此要使用多路器对数据进行选择，由 PicoBlaze 控制选择。

PicoBlaze 的输出口也只有 8 位，将 20 位数据发送出去需要多次发送。另外，需要设计 PicoBlaze 输出端口的逻辑，来满足串口模块接收数据的时序要求。

5.1.3　秒信号发生器的设计与实现

秒信号发生器用于选取 1s 内的被测信号，提供给计数器，计数器将计数结果传送给后续电路进行处理和显示。

考虑到对信号计数后，还需要一定时间来处理和显示，可在秒信号的高电平时间对信号进行计数，而在低电平时间进行处理和显示。实际的秒信号如图 5-3。

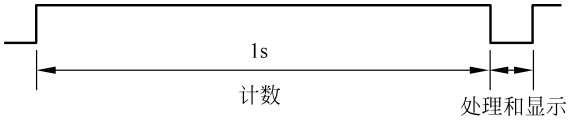

图　5-3

数字系统中，通常对高频数字信号分频得到频率较低(1MHz 以下)的数字信号。在 Xilinx 公司的 FPGA 中，DCM 可以产生高频的数字时钟信号。因此，可用 DCM 产生一个时钟信号，再使用分频器分频来产生秒信号。

通常，采用目标频率整数倍的时钟信号(FPGA 中的时钟资源)作为基准。本例中，将 20MHz 时钟信号二分频后得到 10MHz 信号(非时钟资源)；再将 10MHz 信号经过六次十分频得到 10Hz 信号，这六次分频所采用的是更高频率的时钟信号(100MHz)；最后，对 10Hz 信号进行十一分频，得到秒信号。那么，处理和显示时间为 0.1s(已足够完成处理和显示)。

产生秒信号的系统框图如图 5-4。

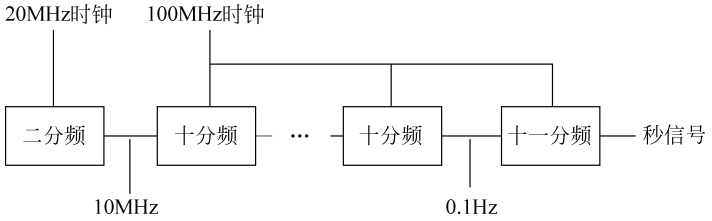

图 5-4

1. 二分频模块的设计与验证

双击桌面上 Xilinx ISE 图标，启动 ISE 软件。单击 File 菜单中的 New Project 选项，在弹出的对话框中输入工程名 half 并指定工程路径。单击 Next 按钮进入下一页，选择所使用的芯片及综合、仿真工具。选用 Spartan6 XC6SLX45 芯片，采用 CSG324 封装。另外，选择 Verilog 作为默认的硬件描述语言。再单击 Next 按钮进入下一页，这里显示了新建工程的信息，确认无误后，单击 Finish 按钮就可以建立一个完整的工程了。

在工程管理区任意位置右击，在弹出的菜单中选择 New Source 命令，选择 Verilog Module 输入，并输入 Verilog 文件名(最好与工程名同名)。单击 Next 按钮进入端口定义对话框，这一步可以略过，在源程序中再行添加。单击 Next 按钮进入下一步，单击 Finish 按钮完成创建。这样，ISE 就会自动创建一个 Verilog 模块的模板，并且在源代码编辑区打开。简单的注释、模块和端口定义已经自动生成，接下来需要将代码编写完整。

参考代码：

```
module half(input  Signal_clk,
            input  rst,
           output reg Signal
            );

always @( posedge Signal_clk )
begin
    if( rst )
       Signal <= 0;
    else
       Signal <= !Signal;
end
endmodule
```

接着，对模块进行综合。在 Hierarchy 栏中，选中 half. v，双击 Processes 栏中的 Synthesize-XST 开始综合。综合结束后，Synthesize-XST 旁的图标若为绿色打勾，表示没有错误和警告，综合通过；若为黄色惊叹号，表示综合通过但存在一些问题，这些问题以警告方式出现；若为红色打叉，表示综合未通过，存在错误。在 Errors 和 Warnings 栏中，可以看到具体的错误和警告条目。

综合通过后，就可以进行仿真了。在工程管理区将 view 设置为 Simulation。再右击，并在弹出的菜单中选择 New Source，在对话框中选择 Verilog Test Fixture，输入测试文件名 half_test，单击“下一步”按钮，这时工程中所有的模块名都会显示出来，在其中选择要进行测试的模块。单击 Next 按钮，再单击 Finish 按钮，ISE 会在源代码编辑区自动生成测试模块的代码。我们看到，ISE 已经自动生成了基本的信号并对被测模块做了例化。添加代码，完成测试。

```
initial begin
    // Initialize Inputs
    Signal_clk = 0;
    rst = 1;

    // Wait 210 ns for global reset to finish
    #210  rst = 0;

    // Add stimulus here
end
always #25 Signal_clk = ~Signal_clk;
```

完成测试文件后，确认工程管理区中 view 选项设置为 Simulation，这时在 Processes 栏会显示与仿真有关的进程，双击 Simulate Behavioral Model，ISE 将启动 ISE Simulator，可以得到仿真结果，如图 5-5 所示。

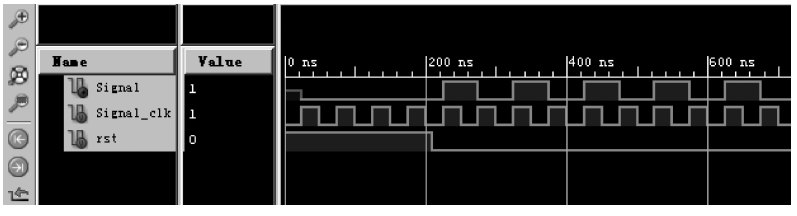

图　5-5

从图 5-5 中可以看出，Signal 信号的频率是时钟的一半。需要注意的是，Signal 是普通信号，而不是时钟，不使用时钟资源。在后续的分频链中，都是将信号分频而非时钟。

2. 十分频模块设计与仿真

双击桌面上 Xilinx ISE 图标，启动 ISE 软件。单击 File 菜单中的 New Project 选项，在弹出的对话框中输入工程名 Div_10 并指定工程路径。单击 Next 按钮进入下一页，选择所使用的芯片及综合、仿真工具。选用 Spartan6 XC6SLX45 芯片，采用 CSG324 封装。另外，选择 Verilog 作为默认的硬件描述语言。再单击 Next 按钮进入下一页，这里显示了新建工程的信息，确认无误后，单击 Finish 按钮建立工程。

在工程管理区任意位置右击，在弹出的菜单中选择 New Source 命令，选择 Verilog

Module 输入，并输入 Verilog 文件名(最好与工程名同名)。单击 Next 按钮进入端口定义对话框，这一步可以略过，在源程序中再行添加。单击 Next 进入下一步，单击 Finish 按钮完成创建。这样，ISE 就会自动创建一个 Verilog 模块的模板，并且在源代码编辑区打开。简单的注释、模块和端口定义已经自动生成，接下来需要将代码编写完整。

十分频模块代码：

```
module Div_10( input  clk,
               input  rst,
              input  Signal_In,
              output Div_Out
              );

reg [3:0] Cnt;
reg Signal_Buf;
wire Signal_Up;

always @( posedge clk )
begin
    Signal_Buf <= Signal_In;
end

assign  Signal_Up = Signal_In & !Signal_Buf;

always @( posedge clk )
begin
   if( rst )
       Cnt <= 0;
    else if( Signal_Up )
    begin
       if( Cnt == 4'd9 )
            Cnt <= 0;
        else
           Cnt <= Cnt + 1;
    end
end

assign Div_Out = ( Cnt == 4'd9);

endmodule
```

接着，对模块进行综合。在 Hierarchy 栏中，选中 Div_10. v，双击 Processes 栏中的 Synthesize-XST 开始综合。

在工程管理区将 view 设置为 Simulation 并右击，并在弹出的菜单中选择 New Source，在对话框中选择 Verilog Test Fixture，输入测试文件名 Div_10_test，单击“下一步”按钮，这时工程中所有的模块名都会显示出来，在其中选择要进行测试的模块。单击 Next 按钮，再单击 Finish 按钮，ISE 会在源代码编辑区自动生成测试模块的代码。我们看到，ISE 已经自动生成了基本的信号并对被测模块做了例化。

3. 秒信号发生器合成与验证

在 ISE 软件中新建一个工程，命名为 Second_Gen 。工程中选用 Spartan6 XC6SLX45 芯片，封装为 CSG324。另外，选择 Verilog 作为默认的硬件描述语言。在工程中新建 Verilog 顶层模块，命名为 Second_Gen。

在工程管理区任意位置右击，单击 Add Copy of Source，找到 Step2 中 half. v 文件所在的路径并选中，将 half. v 加入到本工程。在 Hierarchy 栏中可以看到，half. v 与 Second_Gen. v 没有从属关系。

在 Second_Gen. v 的 endmodule 前添加以下代码：

```
half     U1( .Signal_clk( clk_20M ),
                .rst( rst ),
           .Signal( Signal_10M )
                );
```

并保存。此时观察 Hierarchy 栏中的层次关系可以发现，half. v 成为 Second_Gen. v 的一个子模块。调用关系及信号连接关系如图 5-6。

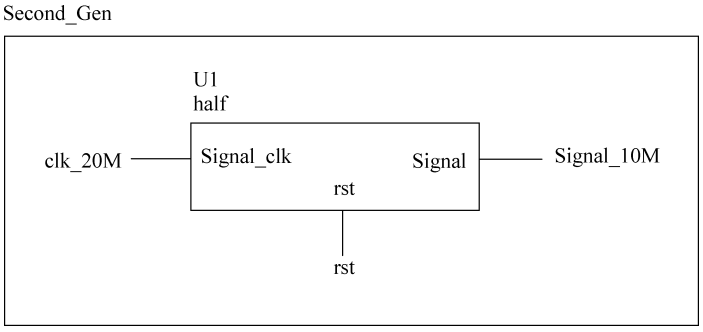

图　5-6

与实物元件类比，half 是被调用模块名，相当于元件名。U1 是模块的实例化名，相当于元件序号。

添加十分频模块和十一分频模块。

(1) 在 Second_Gen. v 中模块名后面的括号中添加模块的输入输出信号：

```
input  clk,
input  rst,
output Second
```

(2) 在括号后添加模块的内部信号：

```
wire Signal_100M;
wire Signal_20M;
wire Signal_10M;
wire Signal_1M;
wire Signal_100K;
wire Signal_10K;
wire Signal_1K;
wire Signal_100;
```

```
wire Signal_10;
```

(3) 依照 half 模块的添加方法,在 Sencond_Gen. v 中添加 6 个十分频模块和 1 个十一分频模块。

在工程管理区任意位置右击,单击 New Source,在 Select Source Type 栏中选择 IP,在 File Name 栏中输入 My_DCM,单击 Next 按钮。在 View by Function 栏中找到 FPGA Features and Design 并展开,找到 Clocking 下的 Clocking Wizard 并选中,单击 Next 按钮。再单击 Filish 按钮。

在 Clocking Wizard 的 Page 1,保持默认选项,单击 Next 按钮。在 Page 2,将 CLK_OUT1 的 Output Freq 的 Requested 的值设为 20,将 CLK_OUT2 的 Output Freq 的 Requested 的值设为 100,单击 Next 按钮。后面的步骤均保持默认,单击 Generate 按钮。

在 Second. v 中添加 DCM 组件。

(1) 在 Hierarchy 栏中,选中 My_DCM 模块,在 Processes 栏中双击 View HDL Instantiatuon Template,在打开的文档 My_DCM. veo 中选择 Begin Cut here for INSTANTIATION Template 和 End INSTANTIATION Template 之间的文本并复制,在 Second_Gen. v 中粘贴。

(2) 完成 Second_Gen. v 中 My_DCM 模块的信号连接。

接着,对这个工程进行综合。综合过程包含了语法检查和逻辑规则检查。若存在错误或警告,应进行排错。

要实现设计,还需要为模块中的输入/输出信号添加管脚约束,这就需要在工程中添加 UCF 文件。在工程管理区右击,单击 New Source,选择 Implementation-Constraints File,出现一个空白的约束文件(文件名与工程同名),就可以为设计添加各种约束。

在 Second_Gen. v 文件中添加:

```
NET "clk" LOC = L15;
NET "rst" LOC = T15;
NET "Second" LOC = U18;
NET "clk" IOSTANDARD = LVCMOS33;
NET "rst" IOSTANDARD = LVCMOS33;
NET "Second" IOSTANDARD = LVCMOS33;
```

在 Hierarchy 栏中选中 Second_Gen. v,在 Processes 栏双击 Implementation Design 选项,就可以自动完成实现步骤。

至此,信号发生器设计完成。

5.1.4 中断信号产生模块的设计

PicoBlaze 提供一个单路的中断输入信号。PicoBlaze 在复位后处于禁止中断的状态,只有运行了 ENABLE INTERRUPT 指令才能使能中断相应。之后若关闭中断,需要在程序中加入 DISABLE INTERRPT 指令。

中断允许时,INTERRUPT 输入信号为高电平且保持两个时钟周期以上,才会产生中断事件。中断事件会强行停止当前程序的执行(执行完正在执行的指令后,保存断点地址和标志位),并执行一条 CALL 3FF 指令。3FF 是程序存储器的最后地址,所以通常会在 3FF 地址处放一条跳转指令,跳转到中断处理程序的首地址去执行。

中断处理程序以 RETURNI ENABLE(中断返回后允许中断)或 RETURNI DISABLE(中断返回后不允许中断)指令返回被中断的程序继续执行,返回的过程实际上是断点的恢复(即将断点地址存入指令指针寄存器并恢复标志位)。

中断响应后,INTERRUPT_ACK 信号会立即有效,用于清零中断请求触发器,防止中断响应后再次发生中断。PicoBlaze 的中断电路如图 5-7 所示。

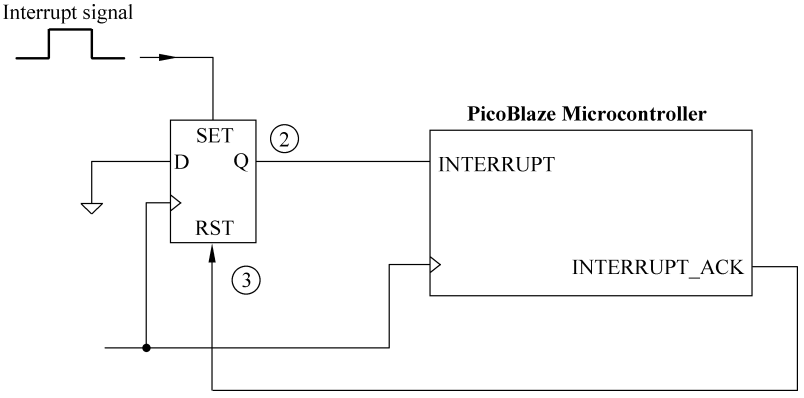

图　5-7

秒信号的低电平期间,PicoBlaze 要做两件事,一个是读取数据并发送,一个是清零计数器。这两个任务都要在中断处理程序当中完成。因此,秒信号的下降沿到来后,就应立即请求中断。

这样,就需要一个中断信号产生模块。这个模块的输入是秒信号,输出是中断请求信号。图 5-8 给出了中断请求信号与秒信号之间的时序关系(PicoBlaze 要求中断请求信号宽度不小于两个时钟周期)。

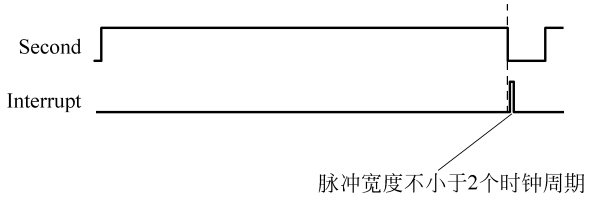

图　5-8

这个模块的实现,可以使用有限状态机。状态机的时钟应和 PicoBlaze 处理器的时钟相同。该状态机的状态图如图 5-9 所示。

中断响应流程,如图 5-10 所示。

PicoBlaze 的中断响应流程包含了以下步骤:

① 开中断。默认情形下,中断是关闭的,可通过运行 ENABLE INTERRUPT 指令来打开中断。

② 中断请求。在运行 INPUT S1,01 指令时发生了中断请求。

③ 中断响应。运行完 INPUT S1,01 指令后,强行停止当前程序的运行,对断点进行保护(即将 ADD S0,S1 的地址以及标志位 Z 和 C 保存到堆栈中),并以指令 CALL 3FF 跳转到 3FF 地址处运行。

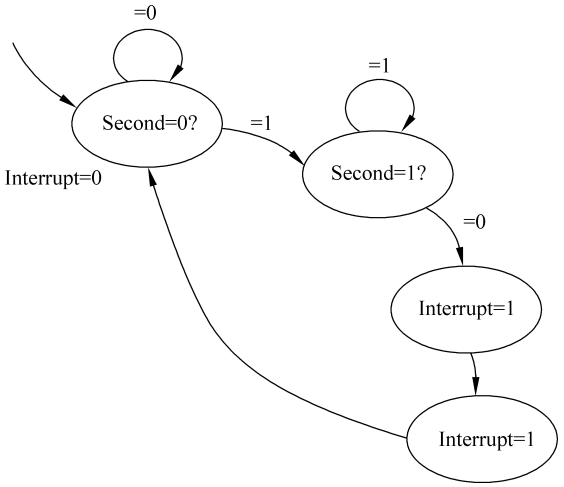
=0
=1
Second=0?
=1
Second=1?
Interrupt=0
=0
Interrupt=1
Interrupt=1

图 5-9

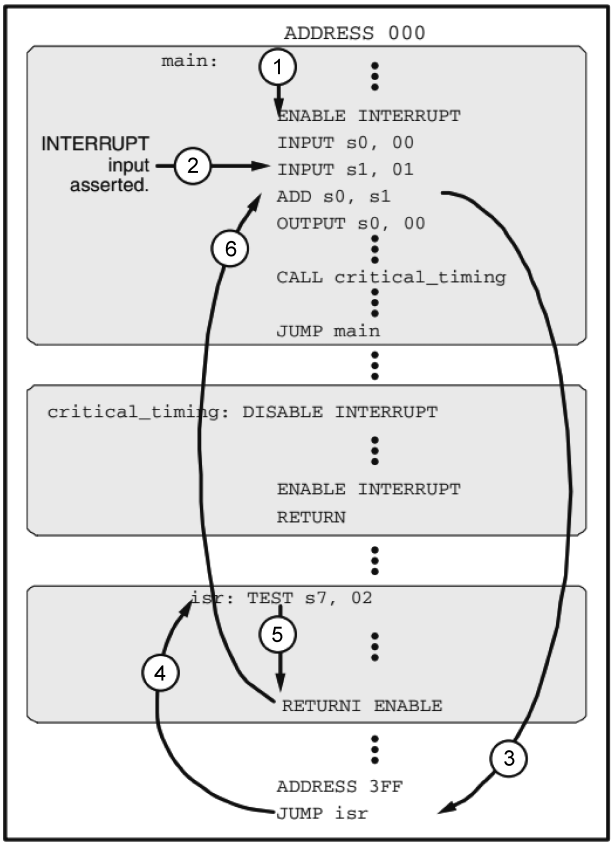
ADDRESS 000
main:
1
ENABLE INTERRUPT
INTERRUPT
input
asserted.
2
INPUT s0, 00
INPUT s1, 01
ADD s0, s1
OUTPUT s0, 00
6
CALL critical_timing
JUMP main
critical_timing: DISABLE INTERRUPT
ENABLE INTERRUPT
RETURN
isr: TEST s7, 02
5
4
RETURNI ENABLE
3
ADDRESS 3FF
JUMP isr

图 5-10

④ 3FF 地址处为一条跳转指令，跳转到中断处理程序去运行。

⑤ 中断处理。

⑥ 中断返回。恢复断点（包括指令指针、标志位等）并继续执行 ADD S0，S1 指令。

根据图 5-10 所示状态图，完成中断程序 Verilog 设计和中断响应的汇编设计。

5.1.5　多路器和 PicoBlaze 的输入端口

观察图 5-7 中多路器和 PicoBlaze 连接部分后会发现，它们的连接关系与图 5-11 的结构非常接近（图 5-11 是 PicoBlaze 的输入端口示意图）。

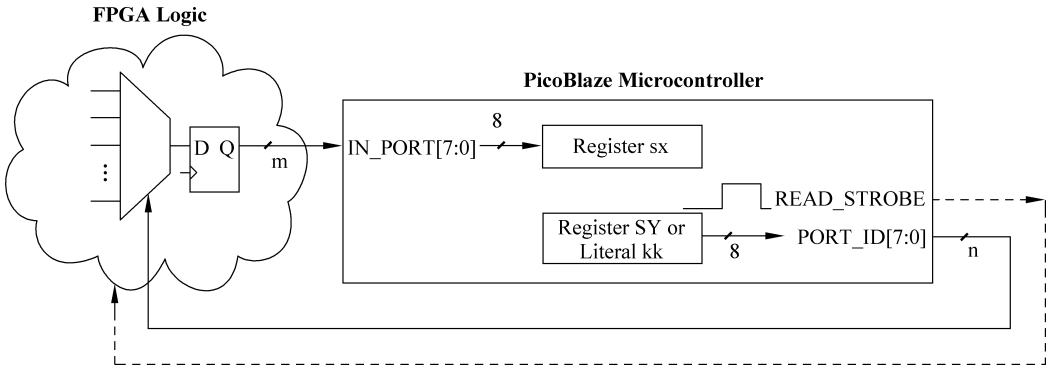

图　5-11

从图 5-11 中我们可以得到以下信息：

（1）输入端口是 8 位的，且输入数据将存入寄存器 sX。

（2）PORT_ID 表示输入设备的编号（地址），8 位，最多支持 256 个外设，地址来自寄存器或取立即数。

（3）一次输入需要 IN_PORT、READ_STROBE 和 PORT_ID 三组信号同时作用来完成。

（4）有时需要外部（对 FPGA 芯片来说，是内部）逻辑来完成输入数据的锁存。

图 5-11 中 FPGA Logic 中左侧的元件（梯形）就是一个多路器，它的选择控制端（其下方有箭头的实线）来自于 PicoBlaze 的端口地址 PORT_ID。那么，选择多路器不同的输入，对应于 PicoBlaze 的不同的输入设备地址。也就是说，PicoBlaze 读不同的地址，就可以读到不同的输入数据，如图 5-12 所示。

计数器的输出为 5 组 BCD 码，所以需要 PORT_ID 中的 3 位来确定地址。

另外，需通过对 PicoBlaze 输入时序的分析来确定多路器的逻辑，主要是判定在 PicoBlaze 读之前多路器的输出是否稳定，如图 5-13 所示。

从图 5-13 中，我们可以得到以下信息。

（1）系统在时钟上升沿的同步下工作。

（2）执行一条 INPUT 指令的耗时是 2 个周期。

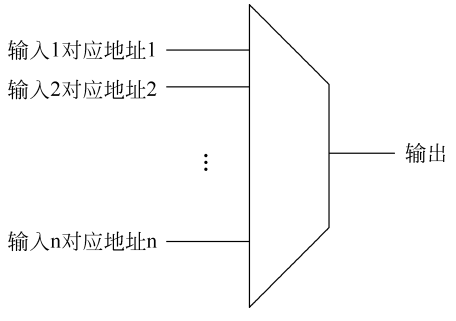

图　5-12

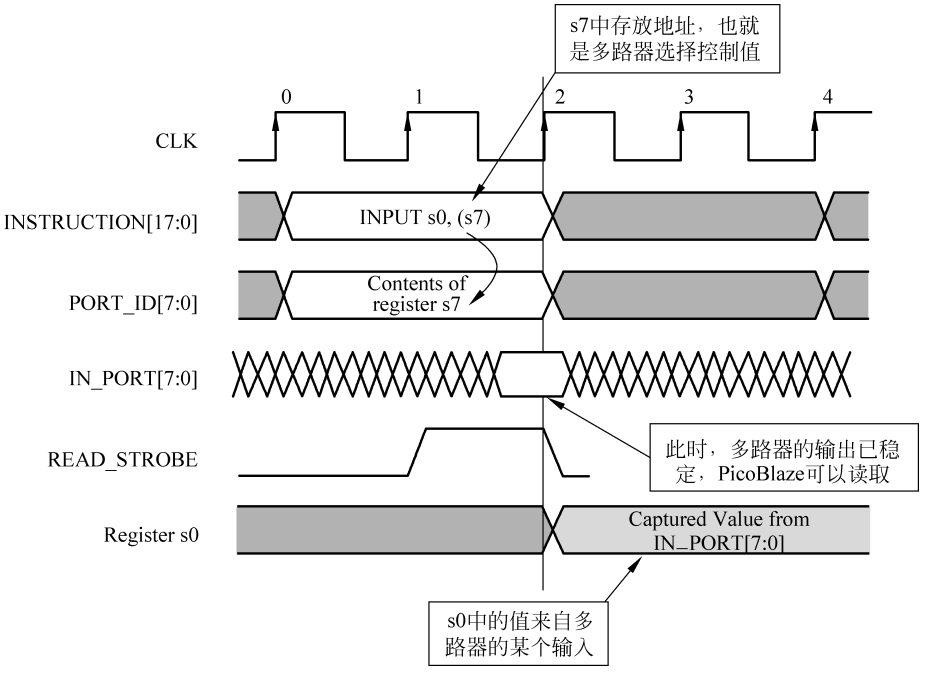

图 5-13

(3) 一条输入指令的位宽是 18 位。

(4) 指令“INPUT s0,(s7)”表示读地址为(s7)的外设并存到寄存器 s0。

(5) READ_ STROBE 信号用于通知 FPGA 逻辑：此时可将数据放置于 PicoBlaze 的输入端口。

5.1.6 串口和 PicoBlaze 的输出端口

PicoBlaze 的输出端口结构如图 5-14 所示。

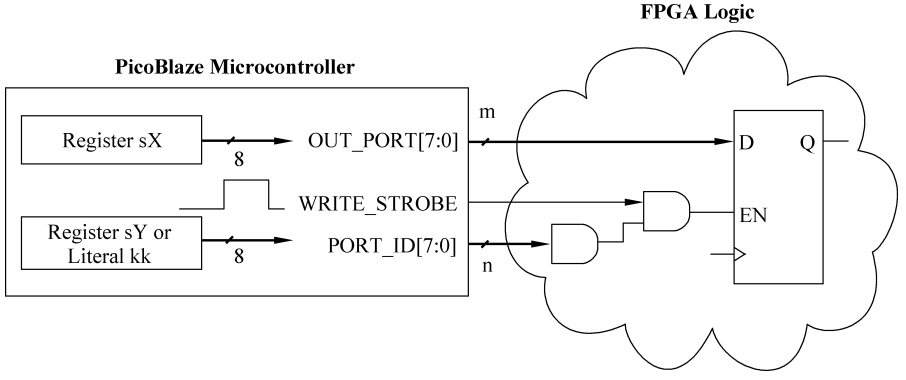

图 5-14

从图 5-14 可以得到以下信息：

(1) 输出端口是 8 位的，且输出数据来自寄存器 sX。

(2) PORT_ID 从字面理解应是端口编号(地址)，8 位，最多支持 256 个外设，地址也来

自寄存器或取立即数。

(3) 一次输出需要 OUT_PORT、WRITE_STROBE 和 PORT_ID 三组信号同时作用来完成。

(4) 还需要外部(对 FPGA 芯片来说,是内部)的逻辑来完成数据的锁存。

再来看输出端口的时序图(如图 5-15)。

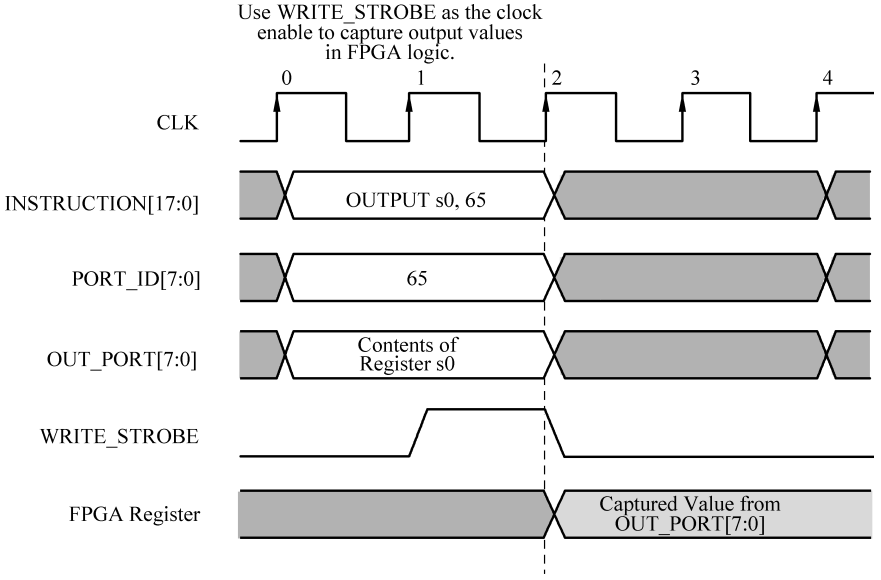

图　5-15

从图 5-15 我们可以得到以下信息:

(1) 系统在时钟上升沿的同步下工作。

(2) 执行一条 OUTPUT 指令的耗时是 2 个周期。

(3) 一条指令的位宽是 18 位。

(4) 指令"OUTPUT s0,65"表示将寄存器 s0 的值输出到 65 端口。

(5) WRITE_ STROBE 信号用于通知 FPGA 逻辑:有数据来自 PicoBlaze 的输出端口。

(6) 需要一个触发器来锁存 OUT_PORT[7:0]信号。

这样,我们可以根据以上信息设计一个逻辑来完善 PicoBlaze 端口功能,如图 5-16 端口锁存器所示。

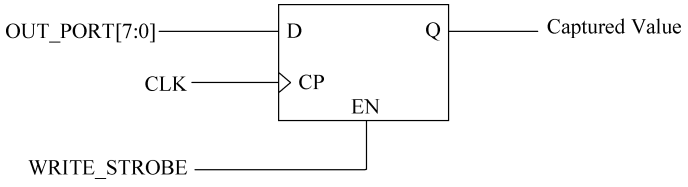

图　5-16

1. PicoBlaze 汇编编写

PicoBlaze 软核开发中,使用汇编语言来编写程序代码。我们来看 PicoBlaze 的指令集。

Program Control Group

```
JUMP aaa
JUMP Z,aaa
JUMP NZ,aaa
JUMP C,aaa
JUMP NC,aaa

CALL aaa
CALL Z,aaa
CALL NZ,aaa
CALL C,aaa
CALL NC,aaa

RETURN
RETURN Z
RETURN NZ
RETURN C
RETURN NC
```

Note that call and return supports up to a stack depth of 31.

Arithmetic Group

```
ADD sX,kk
ADDCY sX,kk
SUB sX,kk
SUBCY sX,kk
COMPARE sX,kk

ADD sX,sY
ADDCY sX,sY
SUB sX,sY
SUBCY sX,sY
COMPARE sX,sY
```

Interrupt Group

```
RETURNI ENABLE
RETURNI DISABLE

ENABLE INTERRUPT
DISABLE INTERRUPT
```

Logical Group

```
LOAD sX,kk
AND sX,kk
OR sX,kk
XOR sX,kk
TEST sX,kk

LOAD sX,sY
AND sX,sY
OR sX,sY
XOR sX,sY
TEST sX,sY
```

Storage Group

```
STORE sX,ss
STORE sX,(sY)
FETCH sX,ss
FETCH sX,(sY)
```

Shift and Rotate Group

```
SR0 sX
SR1 sX
SRX sX
SRA sX
RR sX

SL0 sX
SL1 sX
SLX sX
SLA sX
RL sX
```

Input/Output Group

```
INPUT sX,pp
INPUT sX,(sY)
OUTPUT sX,pp
OUTPUT sX,(sY)
```

如 PicoBlaze 指令有 7 类：程序控制类指令、算术指令、中断指令、逻辑指令、存储指令、移位指令和输入输出指令。

编写好的 psm 汇编代码，用 kcpsm3 编译，生成 v 文件即可使用的专题中。

2. 分析输出 uart 原理

本专题使用给定的串口模块实现将数据传输到 PC。串口模块 uart_tx 由 3 个子模块组成：波特率产生模块 Band_Gen. v、串口发送模块 kcuart_tx. v 和字符缓冲模块 bbfifo_16x8. v。其层次结构和 RTL 原理图如图 5-17 所示。注意，RTL 图中，模块左边的信号是输入，模块右边的信号是输出。

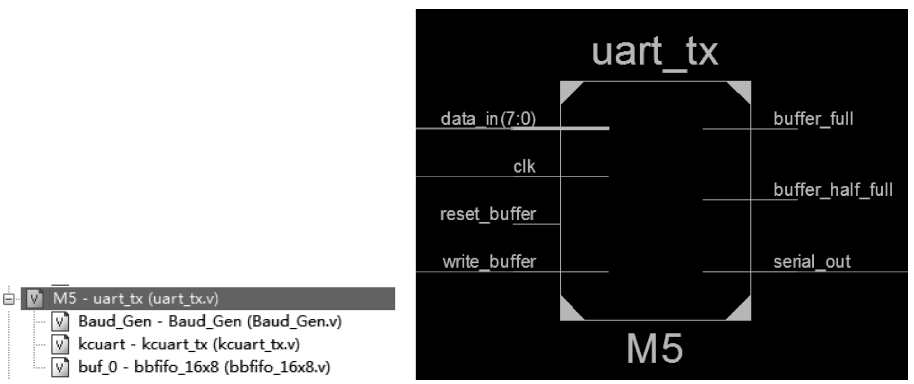

图 5-17

串口模块 uart_tx 的时序如图 5-18 所示。从图 5-18 中可以看出，信号 write_buffer 用于启动串口发送数据。而在 write_buffer 信号有效前，应使发送数据 data_in 有效。

将图 5-18 和 PicoBlaze 的输出时序图相比较，容易发现 write_buffer 信号和 PicoBlaze 的输出锁存信号 Write_Strobe 作用相似。可以利用这一点，将 Write_Strobe 与 PicoBlaze 的 port_id 中某些位运算后连接到 write_buffer（串口与 PicoBlaze 的某些输出地址关联），或将 Write_Strobe 与 write_buffer 直接相连（串口与 PicoBlaze 的所有输出地址关联）。

3. 建立输出模块

新建一个工程，命名为 Out_Port_Test，器件的型号及封装与之前实验相同（型号为

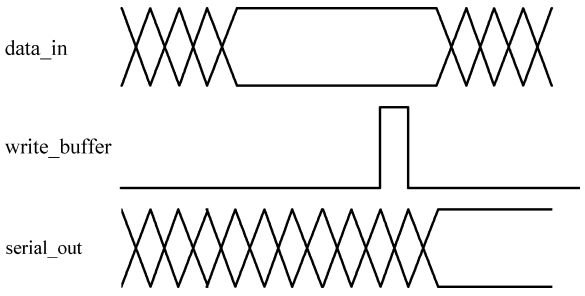

图　5-18

xc6slx45 封装为 3csg324)。

分析 PicoBlaze 端口功能,确定需要使用哪些端口。

address 和 instruction: 读取汇编程序所需的地址和指令接口,必须连接。

out_port: 输出数据。

write_strobe: 输出数据有效,这里需要用它来锁存数据(至 LED)。

port_id: 本例只有一个外设,故此接口可悬空。

in_port 和 read_strobe: 输入端口,对本例来说不需要,但建议 in_port 接 0。

interrupt 和 interrupt_ack: 中断相关端口,也不需要,但建议 interrupt 接 0。

reset : 复位,为 1 时复位,本例中可接 0。

可得出端口的具体连接,其示意见图 5-19。

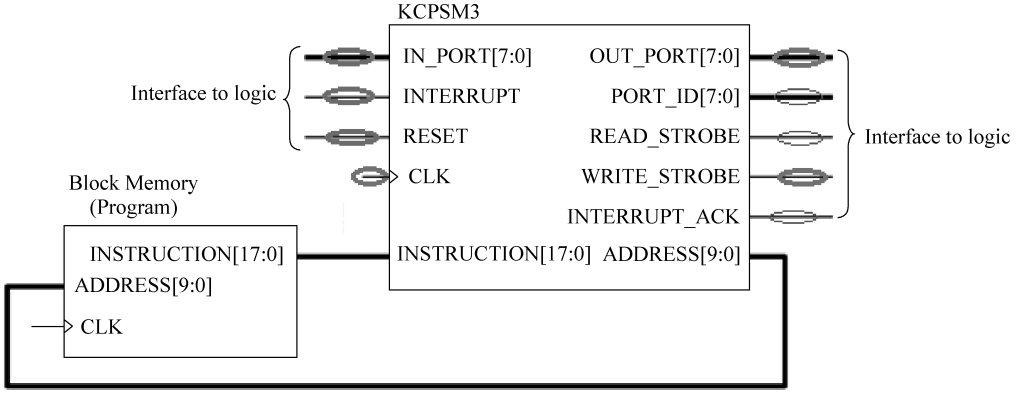

图　5-19

图 5-19 中(参见显示器),绿色表示对外(整个 PicoBlaze 模块之外)需要连接的信号,蓝色表示此信号需要连接但不连接到模块之外,红色表示此信号可悬空。

编写相应的代码,将上一步生成的 v 文件添加到工程中综合,生成 RTL 原理图,如图 5-20 所示。

4. 添加 PicoBlaze 外围电路

(1) 在 Out_Port_Test 工程中添加 DCM 模块,其输出频率为 50MHz。

(2) 在 Out_Port_Test 工程中添加练习一产生的 Verilog 模块 Out_Latch. v。

(3) 在 Out_Port_Test 工程中新建 Verilog 模块,命名为 Out_Port_Test. v(顶层模块,与工程同名)。在顶层模块中逐个调用 my_PicoBlaze 模块、DCM 模块、Out_Latch 模块,并

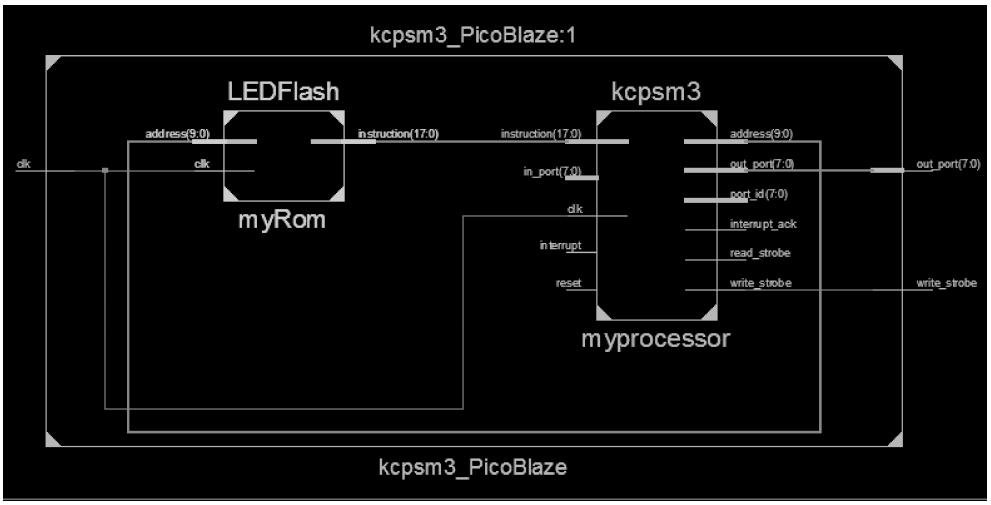

图 5-20

完成信号的连接。

(4) 综合并观察所生成的 RTL 原理图(其结构与图 5-21 相同)。

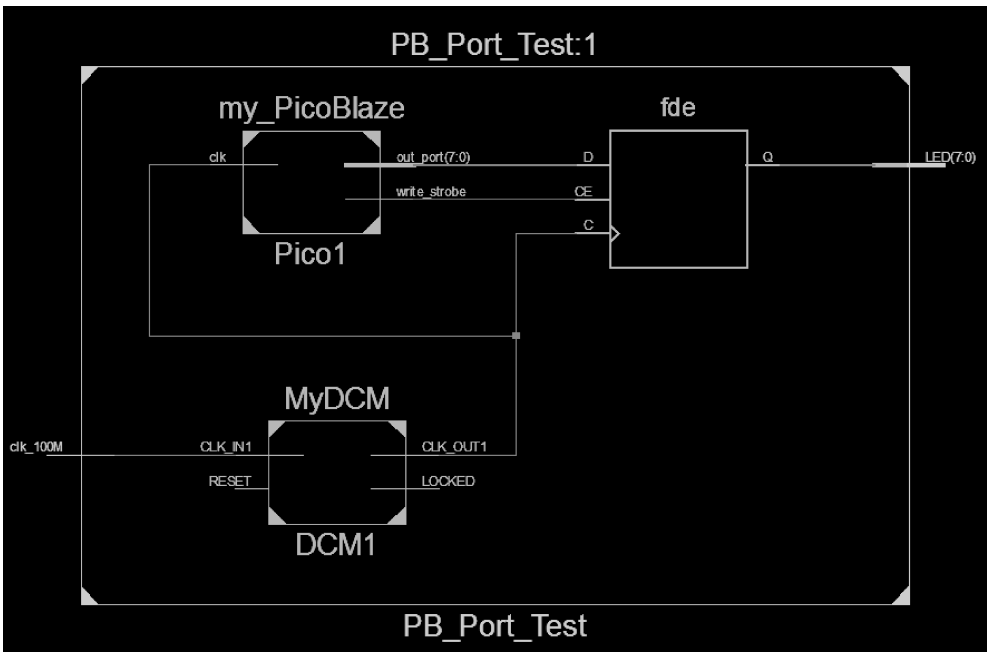

图 5-21

5.1.7 6PicoBlaze 的软件设计

秒信号生成了中断申请信号,送给 PicoBlaze。秒信号利用时钟管理器生成。那么,读取 BCD 码、通过串口发送和清零等动作应该在中断处理程序中完成。

用以生成 PicoBlaze 的汇编代码:

```
        CONSTANT UART_write_port,05
        CONSTANT WAN_port,01
        CONSTANT QIAN_port,02
        CONSTANT BAI_port,03
        CONSTANT SHI_port,04
        CONSTANT GE_port,05
        CONSTANT Clear_Cnt,06
        NAMEREG sF, UART_data
cold_start: ENABLE INTERRUPT
loop:JUMP loop
        ;
  isr: INPUT UART_data,WAN_port
        OUTPUT UART_data, UART_write_port
        INPUT UART_data,QIAN_port
        OUTPUT UART_data, UART_write_port
        INPUT UART_data,BAI_port
        OUTPUT UART_data, UART_write_port
        INPUT UART_data,SHI_port
        OUTPUT UART_data, UART_write_port
        INPUT UART_data,GE_port
        OUTPUT UART_data, UART_write_port
        ;
        LOAD UART_data,character_space
        OUTPUT UART_data, UART_write_port
        ;
        INPUT UART_data,Clear_Cnt
        RETURNI ENABLE
        ;
        ADDRESS 3FF
        JUMP isr
        ;
        CONSTANT character_space, 20
```

5.1.8　验证

(1) 下载后将连接 Atlys 开发板 PROG 位置的 USB 线连接到 Atlys 开发板 UART 位置的 USB 口上,并安装驱动。

(2) 在 PC 上运行串口精灵软件,将串口波特率设为 38400Baud。

(3) 单击串口精灵中的运行,可观察到接收界面中的频率值。

5.2　专题二：Atlys 开发板的 AC97 固件设计

1. 专题介绍

了解基于 Atlys 开发板 AC97 固件的实现。

2. 专题目标

(1) 了解 FPGA 基础。

(2) 了解 FIR 滤波器。

(3) 掌握 FPGA 架构与设计流程。

(4) 滤波系统完整实现。

3. 专题步骤

(1) 设计任务。

(2) FIR 滤波器简述。

(3) FPGA 架构与设计流程。

(4) 滤波系统完整实现。

5.2.1 设计任务

本书主要设计任务是设计一个 FIR 滤波器,从 CD 质素的音乐(48kHz)中滤除 4kHz 的噪声,滤波器的参数设置如下:

Fs=48kHz, FPASS1=2000Hz, PSTOP1=3800Hz, FSTOP2=4200Hz,
FPASS2=6000Hz, APASS1=APASS2=1dB, ASTOP=60dB

本书介绍将设计的滤波器开发成外设 IP 核,将其例化到处理系统中。系统通过板上的编解码芯片和 AC97 控制器获得一段立体声音乐,经过带阻滤波,最后输出到耳机。

5.2.2 滤波器简述

1. 数字滤波器概述

数字滤波主要作用是要抑制干扰,其是数字信号处理中的重要组成部分。FIR 滤波器作为数字信号处理应用中的一个重要的应用课题,在相当长的时间里,其选择都是数字信号处理器。而 FPGA 具有十分灵活的可编程逻辑,随着性能的不断增强,同时其具有查找表结构、可重构特性、可并行处理、流水线操作等特点,这些使得基于 FPGA 的数字滤波器的设计日益广泛。目前,随着数字信号处理技术的不断进步与深亚微米集成电路工艺的快速发展,数字滤波器正在逐步替代传统的模拟滤波器,发展成为一种最主要的滤波器系统。数字滤波器之所以被看好是因为数字滤波器相比模拟滤波器具有以下优点。

(1) 数字滤波器具有很高的精度,甚至模拟滤波器理论上都不能达到数字滤波器所能达到的性能,这显然是非常重要的。

(2) 数字滤波器具有很高的信噪比。这主要因为数字滤波器以数字器件执行运算,从而避免了各种电路噪声的影响,不会像模拟滤波器那样受到元件参数的影响,现代深亚微米技术使得数字设计在集成度方面可以更好实现,这也是从另一方面更有益于系统的集成,也有益于获得更高的信噪比。

(3) 数字滤波器可靠性很高。这主要是因为数字滤波器避免了电子元件的电路特性随着时间、温度、电压、电流等变化而变化所带来的影响。在数字电路工作环境下,数字滤波器具有非常高的可靠性;相比之下模拟滤波器受到电子元件电路特性的影响便显得稍差了。此外,数字滤波器还可以轻易满足幅度和相位线性的严格要求,易在硬件上实现,易于和数字信号处理系统集成。

然而,数字滤波器的处理能力受到系统采样频率的限制,根据奈奎斯特采样定理,输入信号频率分量必须小于滤波器 1/2 采样频率的分量,否则会因“混叠”而无法正常工作。所以在某些场合,模拟滤波器仍然是很重要的器件。

2. IIR 和 FIR 数字滤波器

线性时不变(LTI)滤波器是最常见的数字滤波器系统,其原理为线性卷积运算。LTI 数字滤波器分为无限长冲击响应(IIR)滤波器和有限长冲击响应(FIR)滤波器两类。与 IIR

滤波器相比，FIR 滤波器的优点是可以在设计任意幅频特性的同时，保持严格的线性相位特性。线性相位特性对于图像处理、语言信号处理、高质量音频处理等一些性能要求较高的系统是非常重要的，所以 FIR 滤波器在现代信号处理领域更受欢迎，得到了更广泛的应用。要提一下的是，达到同样的衰减特性时，FIR 滤波器的阶数应高于 IIR 滤波器。

FIR 滤波器的实现包括软件实现方案与硬件实现方案。在非实时系统与低速系统中，可在 DSP 或 CPU 上用软件实现 FIR 滤波算法，其特点是虽然实现方法简单但是实时性比较差。在实时性要求较高的系统中，此种方法难以满足设计需求，常需要采用硬件实现。常用于硬件实现的器件有 DSP 器件、定制 ASIC 芯片、FPGA 等。随着可编程逻辑器件高速发展，容量与速度均有增加，FPGA 的并行处理特质使得其非常适合在实时性要求比较高或计算量比较大的场合组建硬件滤波系统，用来实现 FIR 滤波器。

3. 数字滤波器的原理结构

FIR 滤波器的实质是输入采样序列与滤波器系数序列进行卷积，从而得到输出序列，长度为 N 的 FIR 滤波器(阶数为 $N-1$)包括 N 个抽头 $h(n)$，其数学表达式为：

$$y(n)=\sum_{i=0}^{N-1}h(n)\cdot x(n-i)$$

其中 $x(n)$为输入序列，$y(n)$为输出序列，$h(n)$为单位采样响应。

FIR 滤波器的基本结构如图 5-22 所示。

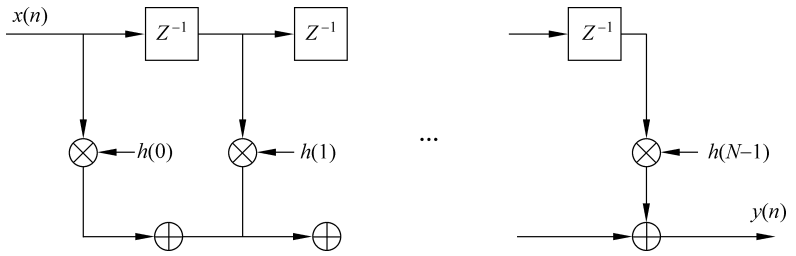

图　5-22

FIR 滤波器的传递函数为：

$$H(z)=\sum_{n=0}^{N-1}h(n)z^{-n}$$

其中，$h(0)$、$h(1)\cdots h(N-1)$为系统的单位冲击响应。

4. FIR 数字滤波器的实现

本设计最终要实现从 48kHz 的 CD 音乐中滤除 4kHz 噪声信号，采用带阻型滤波，滤波器的参数为：N = 59，Fs = 48kHz，FPASS1 = 2kHz，FSTOP1 = 3.8kHz，FSTOP2 = 4.2kHz，FPASS2 = 6kHz，APASS1 = APASS2 = 1dB，ASTOP = 60dB。

利用 MATLAB 里的 FDATool 工具实现这个滤波器，用该软件可以得到滤波器的系数，设置如图 5-23 所示。

FIR 带阻滤波器幅频响应如图 5-24 所示，两个截止频率之间所有成分幅度衰减高于 60dB，信号通过此滤波器时，4kHz 处于这个频段之中，其幅值衰减将超过 60dB，便被成功抑制掉。而两个通带边缘频率之间的频率成分便是有效信号也受到一定衰减影响的部分，显然这段频率跨度越小，衰减越小越好。

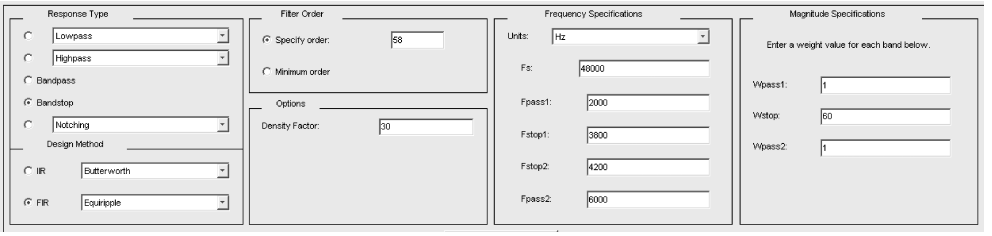

图 5-23

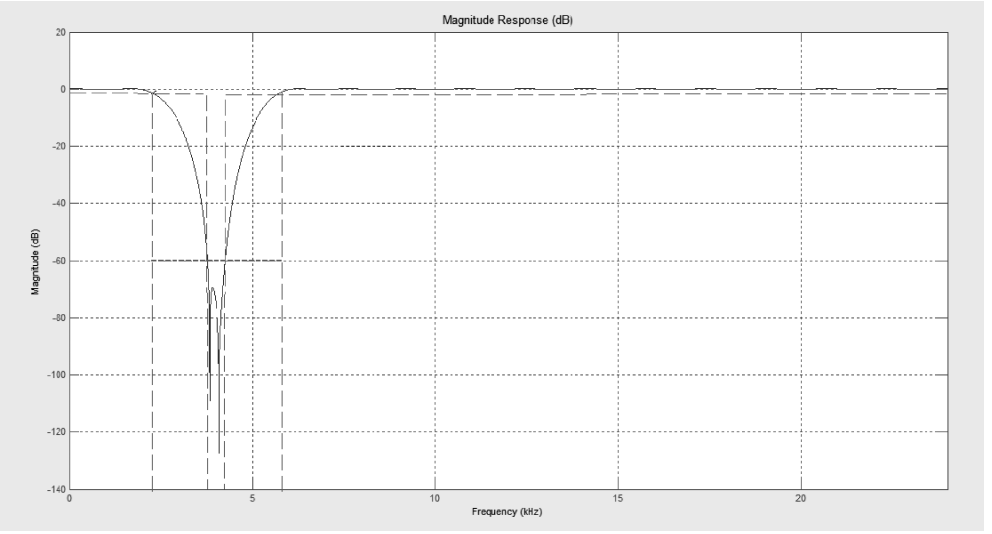

图 5-24

5.2.3 FPGA 架构

1. 设计平台介绍

Atlys 开发板

Atlys FPGA 开发板是一块基于 Xilinx Spartan-6 LX45 FPGA 的功能强大的数字电路开发平台。其板载外围集成了 Xilinx Spartan-6 LX45 FPGA 芯片、128MByte DDR2 存储阵列、16MByte×4 SPI FLASH、10\100\1000 以太网接口、HDMI 视频输入输出、AC97 音频编解码器、100MHz CMOS 晶振、USB-UART 和 USB-PROG 等接口以及 GPIO 外设(8 个 LED 灯,6 个按钮,8 个滑动开关)等。配置如图 5-25 所示。

Atlys 开发平台在基于嵌入式处理器构建完整的数字应用系统时是个理想的选择。Atlys 的编程下载方式选择也很多样,通过 Digilent Adept 软件可以实现 Atlys 开发板的通信和编程下载。此外,通过 Digilent Plugin for Xilinx Tools,也可以使用 Xilinx 自带的 iMPACT 实现 Atlys 开发板的电路编程下载。

LM4550 AC97 音频编解码器

LM4550 AC97 音频编解码器包含 4 个音频插孔,耳机输出(J7),LINE OUT(J5),LINE IN(J4)和麦克风 MIC(J6),麦克风插孔是单声道的,所有其他接口均为立体声。音频支持 18 位数据位、48kHz 采样频率,音频输入和音频输出的采样率可以不同。结构如图 5-26 所示。

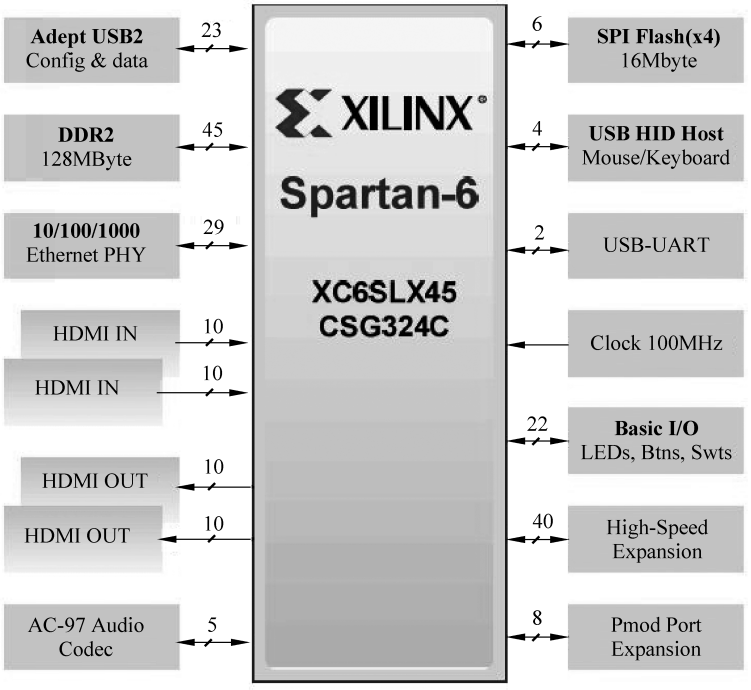

图　5-25

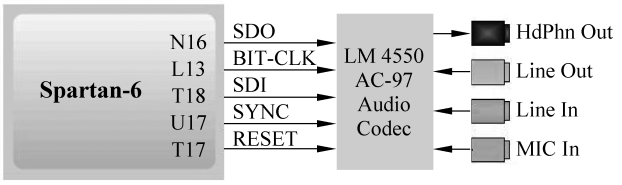

图　5-26

LM4550 AC97 音频编解码器符合 AC97 2.1 版标准，连接成初级编解码器(ID1＝0，ID0＝0)。AC97 编解码器的控制和数据信号如表 5-1 所示。

表　5-1

信 号 名 称	引　　脚	类　　型
SDO	N16	LVCMOS33
BIT-CLK	L13	LVCMOS33
SDI	T18	LVCMOS33
SYNC	U17	LVCMOS33
RESET	T17	LVCMOS33

2. FPGA 设计流程概述

FPGA 多使用 4(6)输入的查找表(LUT)，每一个 LUT 可以看成一个有 4(6)位地址线的 16×1(64×1)的 RAM。FPGA 芯片主要构成有：可编程输入输出单元(IOB)、可配置逻辑块(CLB)、时钟管理模块、嵌入式块 RAM(BRAM)、丰富的布线资源、底层内嵌功能单

元、内嵌专用硬核，每个部分都具有各自具体的功能。目前，主流 FPGA 都是基于 SRAM 工艺的，还有一些采用 FLASH 或熔丝与反熔丝技术的军用和宇航级的 FPGA。

基于 FPGA 的设计是指利用 FPGA 芯片作为硬件基础，借助相关的 EDA 开发软件和编程工具，实现具有一定功能的数字电路。开发流程一般包括电路功能设计、设计输入、功能仿真、综合、综合后仿真、实现与布局布线、时序仿真与验证、板级验证以及芯片编程与调试等主要步骤。

本设计中使用的软件平台为 Xilinx EDA 软件：2012.3 Vivado HLS，V14.3 EDK(XPS & SDK)，V14.3 ISE Foundation Software。

Vivado HLS

Vivado HLS 是 Xilinx 的高层次综合解决方案，它综合 C、C++和 System C 代码形成 Verilog 和 VHDL RTL 结构，并将其封装集成到 Vivado IP 集成器里。Vivado 高层次综合速度非常快，可以在短短几分钟处理成千上万的 C 代码，提供了设计探究的可能，可以通过性能、资源和功耗指标去微调架构，也可以在函数引入约束和指令，创造不同的架构。Vivado HLS 工程的实现框图如图 5-27 所示。

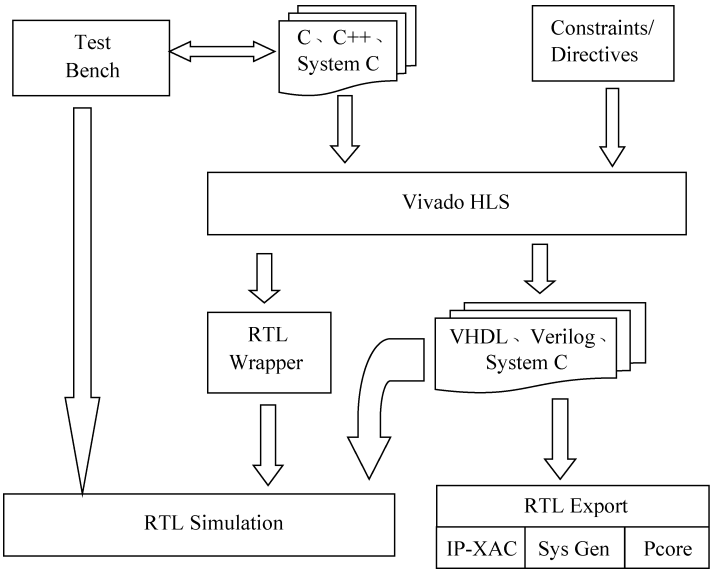

图 5-27

ISE Design Tools Project Navigator

Project Navigator 创建的工程主要用来进行逻辑设计，设计语言可为 Verilog 或 VHDL，此外还可选择原理图输入等其他方式。其通常包括设计输入、行为仿真、综合、实现和下载调试等过程。

本实验中主要通过 Project Navigator 创建工程，并添加所生成的 RTL 文件用来进行 ISIM 仿真以观测滤波器模块的时序逻辑。

EDK

EDK 用于嵌入式系统的设计，使用 C 或 C++语言，其包括 XPS(Xilinx Platform Studio)和 SDK(Xilinx Software Development Kit)，前者用于构建嵌入式硬件平台，后者专门用于开

发应用软件。EDK 的主要开发步骤有：利用 BSB 创建硬件平台、添加 IP Core 以及用户自定义外设、生成仿真文件并测试硬件系统、生成硬件比特流、开发软件系统、合并软硬件比特流、下载和在线调试等。

滤波器系统设计流程

在 Xilinx 相应的软件和 FPGA 硬件平台的基础上，FIR 滤波处理系统的具体设计流程如图 5-28 所示。

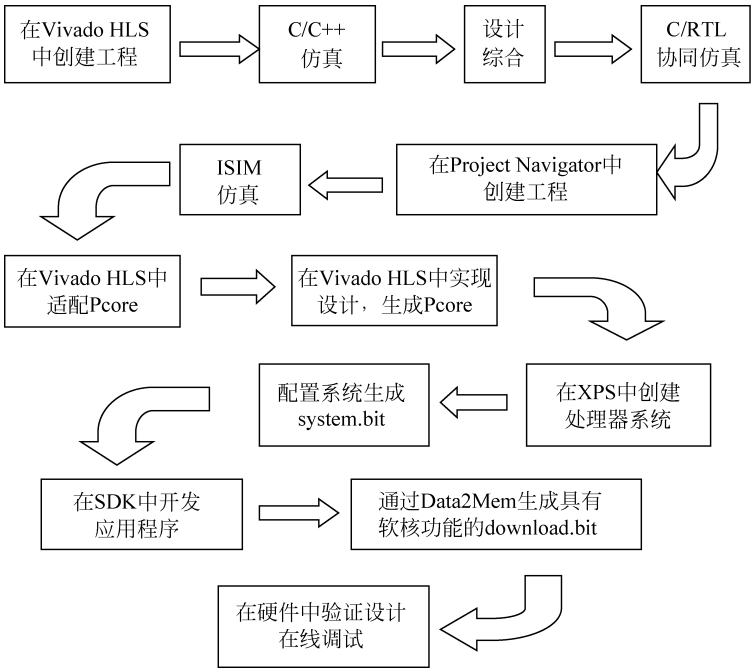

图　5-28

根据图 5-28 可知，滤波系统完整设计流程大约可按软硬件使用划分为以下几步：Vivado HLS 中 C 代码到 VHDL RTL 的综合实现；Project Navigator 中的 ISIM 仿真；Vivado HLS 中生成 Pcore；XPS 中创建处理器系统；SDK 中开发应用程序；硬件实现。

5.2.4　FPGA 设计流程

1. Vivado HLS 中 C 到 VHDL RTL 的综合实现

滤波器函数的 C 语言实现

fir.c 文件内容如图 5-29 所示。

FIR 滤波器将 x 设定为抽样输入，y 指向计算的抽样输出，两者类型 data_t。滤波器参数放在从文件 fir_coef.dat 装载的类型为 coef_t 的数组 c 中。使用顺序算法，在类型为 acc_t 的变量 acc 中计算抽样输出的累积值。

关于数据类型的具体定义如图 5-30 所示。

头文件包括 ap_cint.h，所以用户自定义的任意精度的字节宽度可以被使用。它还定义了抽头数 N、抽样数目(测试平台中)以及数据类型 coef_t、data_t 和 acc_t。coef_t 和 data_t 是 16 位短整型，使用的参数是 16 位有符号短整型，来自编解码芯片的抽样信号也是 16 位宽的。由

```
1 #include "fir.h"
2
3 void fir (
4   data_t *y,
5   data_t x
6   ) {
7   const coef_t c[N+1]={
8  #include "fir_coef.dat"
9     };
10
11
12   static data_t shift_reg[N];
13   acc_t acc;
14   int i;
15
16   acc=(acc_t)shift_reg[N-1]*(acc_t)c[N];
17   loop: for (i=N-1;i!=0;i--) {
18     acc+=(acc_t)shift_reg[i-1]*(acc_t)c[i];
19     shift_reg[i]=shift_reg[i-1];
20   }
21   acc+=(acc_t)x*(acc_t)c[0];
22   shift_reg[0]=x;
23  *y = acc>>15;
24 }
```

图　5-29

```
#ifndef _FIR_H_
#define _FIR_H_
#include "ap_cint.h"
#define N   58
#define SAMPLES N+10 // just few more samples then number of taps
typedef shortcoef_t;
typedef shortdata_t;
typedef int38acc_t;
#endif
```

图　5-30

于该算法迭代(乘法和积累)超过 59 个抽头,有可能位增长 6 比特,所以 acc_t 被定义为 int38。由于 acc_t 比抽样信号和参数宽,使用前要先计算它,如 fir.c 的第 16、18 和 21 行。

测试平台 C 语言描述

fir_test.c 文件如图 5-31 所示。

```
1 #include <stdio.h>
2 #include <math.h>
3 #include "fir.h"
4 void fir (
5   data_t *y,
6   data_t x
7   );
8
9 int main () {
10   FILE   *fp;
11
12   data_t signal, output;
13
14   fp=fopen("fir_impulse.dat","w");
15   int i;
16   for (i=0;i<SAMPLES;i++) {
17       if(i==0)
18           signal = 0x8000;
19       else
20           signal = 0;
21       fir(&output,signal);
22           fprintf(fp,"%i %d %d\n",i,signal,output);
23   }
24   fclose(fp);
25   return 0;
26 }
27
```

图　5-31

注意到测试平台以写模式打开了 fir_impulse.dat，发出一个脉冲信号（第一抽样值为0x8000，其余为零）并将输出的抽样值保存在文件中。

Vivado HLS 中 C 到 VHDL RTL 的综合实现

启动 Vivado HLS，Start→All Programs→Xilinx Design Tools→Vivado 2012.3→Vivado HLS。新建工程并命名为 fir_prj，保存目录设置为 D:\xup\lab。进入设置向导，在源文件添加/移除的窗口中，输入 fir 作为函数名（与源文件相符，必须命名为 fir 以便能顺利综合）并添加 fir.c 和 fir_coef.dat。在测试平台添加/移除的窗口中添加 fir_test.c。实现方案配置，保留方案名 solution1 和时钟周期 10，不确定度的空白档将默认取值 0.125。部件选择按照 Atlys 的标准应该设置为如下：

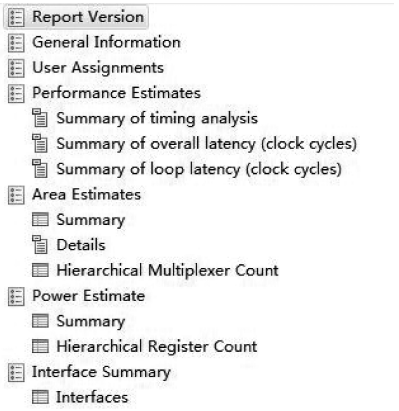

图　5-32

Family：Spartan6，Sub-Family：Spartan6，Package：csg324，Speed Grade：－2，选择 XC6SLX45CSG324-2。

成功创建工程后打开 fir.c，运行 Active Solution 进行综合，综合结束，一些报告文件将会生成，综合的结果也在其中显示。综合报告给出了本实验设计的性能和资源的评估、延迟以及顶层接口信号。具体可以从报告中看到的内容如图 5-32 所示，这里只看一下性能评估（Performance Estimates）和接口信息（Interface Summary），分别见图 5-33 和图 5-34。

Performance Estimates

- **Summary of timing analysis**
  - Estimated clock period (ns): 9.78
- **Summary of overall latency (clock cycles)**
  - Best-case latency: 175
  - Average-case latency: 175
  - Worst-case latency: 175
- **Summary of loop latency (clock cycles)**
  - **loop**

图　5-33

Interface Summary

Interfaces

	Object	Type	Scope	IO Protocol	IO Config	Dir	Bits
ap_clk	fir	return value	-	ap_ctrl_hs	-	in	1
ap_rst	-	-	-	-	-	in	1
ap_start	-	-	-	-	-	in	1
ap_done	-	-	-	-	-	out	1
ap_idle	-	-	-	-	-	out	1
ap_ready	-	-	-	-	-	out	1
y	y	pointer	-	ap_vld	-	out	16
y_ap_vld	-	-	-	-	-	out	1
x	x	scalar	-	ap_none	-	in	16

图　5-34

报告里包含用户设定、速度(如 latency、trip count)、面积(FFs、LUTs)、功耗、接口等信息。

由图 5-33 可见,时钟周期 9.78ns,延时为 175 个时钟周期。

接口信息显示目前 ap_clk,y,x 三个信号有对应对象 fir,y,x。

选择 fir.c,将 PIPELINE 流水线指令应用到 loop 上去。重新综合结束,生成新的综合报告如图 5-35 所示。注意延迟减少到 64 个时钟周期,接口信息不变。

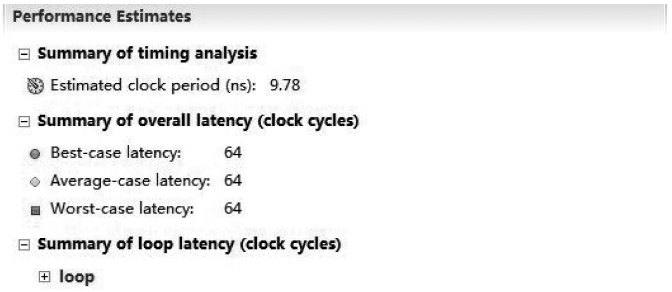

图 5-35

进行 C/RTL 协同仿真,协同仿真生成和编译一些文件,并对设计进行仿真。在控制窗口可以观察这个过程。完成后,RTL 仿真报告成功,延迟仍为 64,接口信息不变。

2. Project Navigator 中的 ISIM 仿真

滤波器仿真激励源

此文件用来进行仿真,通过对输入信号的控制来得到自己想要的仿真效果,根据仿真波形分析时序逻辑。

Project Navigator 中的 ISIM 仿真

启动 Project Navigator,新建工程并命名为 fir_projnav。器件设定如图 5-36 所示。

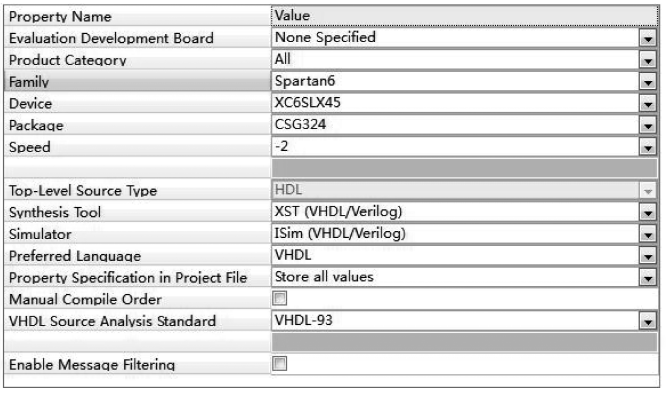

图 5-36

Family: Spartan-6,Device: XC6SLX45,Package: CSG324,Speed: -2, Preferred Language: VHDL

新建完毕后,将刚才用 Vivado HLS 生成的 VHDL 文件全部添加到工程中,并将 fir 设置为顶层模块。添加 testbench 文件,运行行为仿真,仿真执行 4000ns,当 ap_done 为高电平时,输出滤波器参数,参数输出见图 5-37 是正确的。

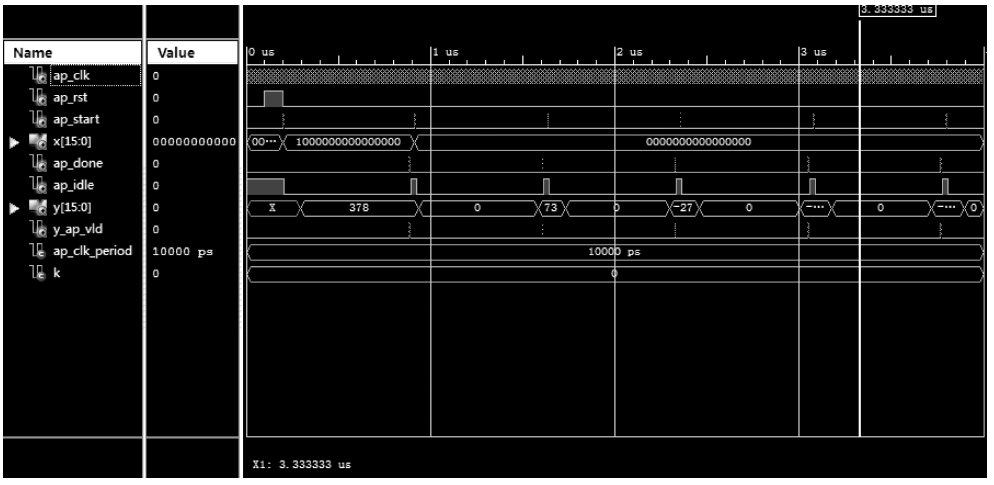

图　5-37

3. Vivado HLS 中生成 Pcore

重新回到 Vivado HLS，打开 fir.c，选择 Directive 标签，右键单击 x，打开指令编辑对话框，设置如图 5-38，其中元数据 metadata 框内，输入-bus_bundle fir_io，使输入输出通过名为 fir_io 的 AXI4Lite 适配器联系在一起。

类似地，对输出信号 y 做同样的设置。

对顶层模块 fir 应用 Resource 指令，变量名字显示为 return，元数据也设定为-bus_bundle fir_io。以上步骤将为信号 x、y、ap_start、ap_valid、ap_done 和 ap_idle 信号创建地址映射。作为核上独立的端口，ap_start、ap_valid、ap_done 和 ap_idle 信号依次产生，前提是，不对顶层模块 fir 应用 Resource 指令。这些端口再通过 GPIO P 核在处理系统中被连通。

对 P 核适配时，将 x、y 以及顶层模块 fir 的 Direcive 类型都设置为 Resource，选用 AXI4LiteS 并作相同命名，以便将输入输出端口连接。

对设计重新综合，通过 Export RTL 生成 Pcore。展开 impl 文件夹，观察到在 Pcores 文件夹（如图 5-39 所示）下，创建了 fir_top_v1_00_a，它将在开发处理系统的过程中被使用，包括数据、网表和其他子文件。

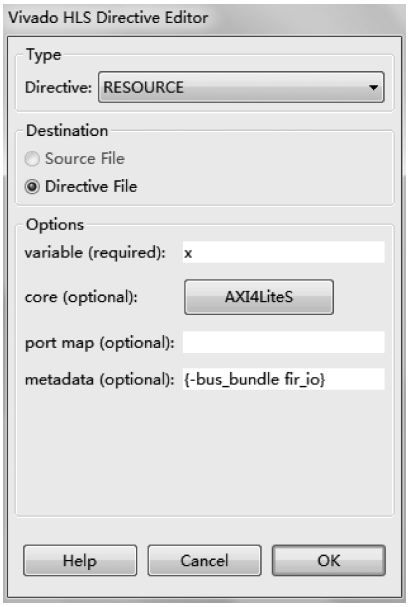

图　5-38

4. XPS 中创建处理器系统

通过 XPS 的 Base System Bulider 创建以 Single MicroBlaze 软核处理器为基础的 AXI 系统：频率 100MHz，本地内存 16KB，只选外设 RS232_Uart_1（AXI UARTLITE，115200 baud rate，8 Data bits，no interrupt，no parity）。使用 IP Catalog 添加其他满足需要的 IP（ac97_0），将由 Vivado HLS 生成的 P 核的文件夹放在 pcores 下，在 XPS 中重新扫描用户数据库，找到新创建的 P 核（fir_top）并添加（fir_left、fir_right）。

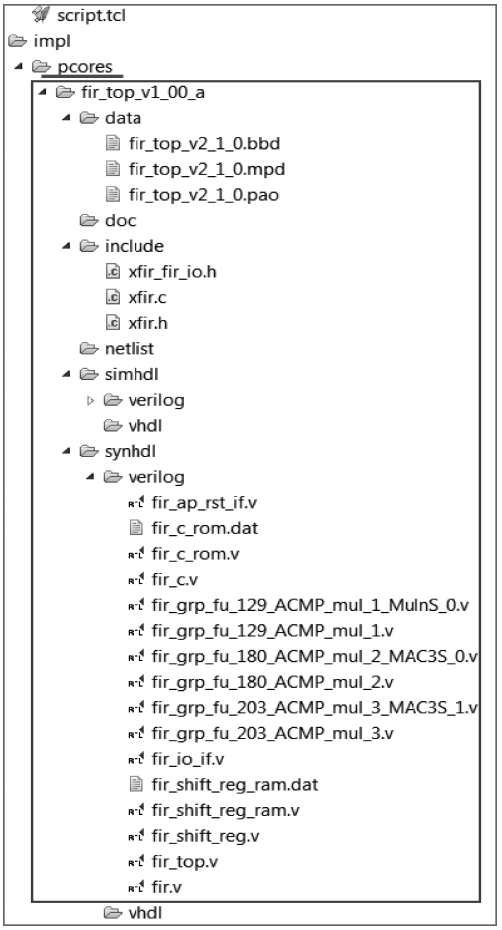

图 5-39

ac97 核的结构图如图 5-40 所示。新创建的 P 核结构图如图 5-41 所示。

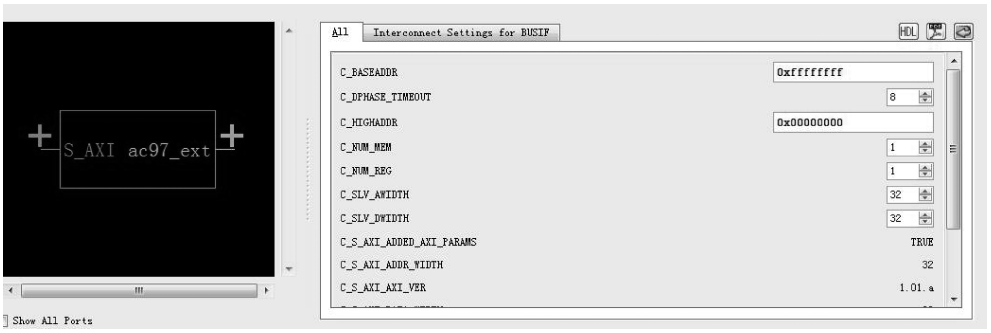

图 5-40

Ports 设置，确认 ac97_0 下的 BITCLK、RESET_N、SDATA_IN、SDATA_OUT 和 SYNC 连接到外部端口。并将 S_AXI_ACLK 与信号 clock_generator_0：CLKOUT0 相连。将同样的时钟信号给 fir_left 和 fir_right 的 SYS_CLK 端口，SYS_RESET 端口设为 proc_sys_reset_0：Interconnect_aresetn。地址映射表如图 5-42 所示。

图　5-41

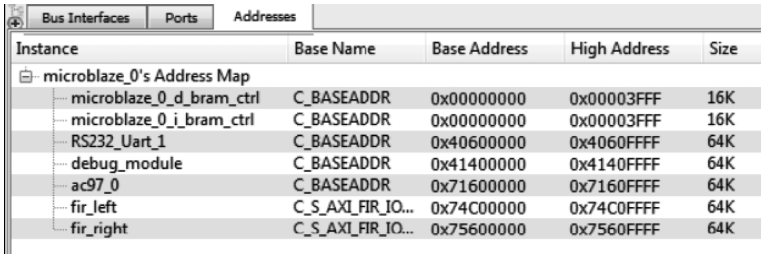

图　5-42

修改约束文件 system. ucf，给 ac97 定义引脚，I/O 标准均为 LVCMOS33：

```
NET ac97_0_BITCLK_pin LOC = "L13"  |  IOSTANDARD = "LVCMOS33";
NET ac97_0_RESET_N_pin LOC = "T17"  |  IOSTANDARD = "LVCMOS33";
NET ac97_0_SDATA_IN_pin LOC = "T18"  |  IOSTANDARD = "LVCMOS33";
NET ac97_0_SDATA_OUT_pin LOC = "N16"  |  IOSTANDARD = "LVCMOS33";
NET ac97_0_SYNC_pin LOC = "U17"  |  IOSTANDARD = "LVCMOS33";
```

Generate Bitstream，生成 system. bit 文件。

5. SDK 中开发应用程序

XPS 中的 bit 文件生成后，选择命令 Export Hardware Design to SDK 以启动 SDK 软件。可以在 overview 查看处理器的地址映射(图 5-43)与设计中的 IP 块(图 5-44)。

新建一个无需任何库支持的 standalone_bsp_0 软件平台工程。库生成器将在后台运行，产生 xparameters. h 头文件。新建 C 工程命名为 TestApp，使用 Empty Application 工程模块。将 testapp. c 和 xfir_fir_io. h 通过 import 添加到 testapp. c 工程中，编译生成 TestApp. elf。

6. 硬件实现

连接 Atlys 的 usb-uart 到电脑的 USB 端口，用音频线连接 Atlys 的 Line-in 到电脑的耳机输出口，连接耳机到 Atlys 的 Line-out 或 HP-OUT 接口。连接 Atlys 电源适配器，给 Atlys 板上电。新建一个终端连接，在弹出的窗口中作如图 5-45 所示的设置。

Address Map for processor microblaze_0

microblaze_0_d_bram_ctrl	0x00000000 0x00003fff
microblaze_0_i_bram_ctrl	0x00000000 0x00003fff
debug_module	0x41400000 0x4140ffff
rs232_uart_1	0x40600000 0x4060ffff
ac97_0	0x71600000 0x7160ffff
fir_left	0x74c00000 0x74c0ffff
fir_right	0x75600000 0x7560ffff

图 5-43

IP blocks present in the design

proc_sys_reset_0	proc_sys_reset	3.00.a	Datasheet
microblaze_0_ilmb	lmb_v10	2.00.b	
microblaze_0_i_bram_ctrl	lmb_bram_if_cntlr	3.10.b	
microblaze_0_dlmb	lmb_v10	2.00.b	
microblaze_0_d_bram_ctrl	lmb_bram_if_cntlr	3.10.b	
microblaze_0_bram_block	bram_block	1.00.a	
microblaze_0	microblaze	8.40.b	Datasheet
debug_module	mdm	2.10.a	
clock_generator_0	clock_generator	4.03.a	
axi4lite_0	axi_interconnect	1.06.a	Datasheet
rs232_uart_1	axi_uartlite	1.02.a	Datasheet
ac97_0	ac97	1.00.a	
fir_left	fir_top	1.00.a	
fir_right	fir_top	1.00.a	

图 5-44

然后选择命令 Program FPGA，选择之前生成的 system.bit 和 system_bd.bmm 两个文件。同时选择刚生成的 TestApp.elf 来初始化块 RAM，单击 Program。Data2Mem 工具将 bit 文件、BMM 文件和 SDK 生成的 elf 文件整合到一起，生成带有软核功能的比特流文件：download.bit。然后断开终端连接，将 USB 连接至 Atlys 的 USB-PROG 接口，用 Adept 软件将该文件下载到 FPGA 中。之后再将线接至 usb-uart 接口，重新建立一个终端，设置和上面一样。

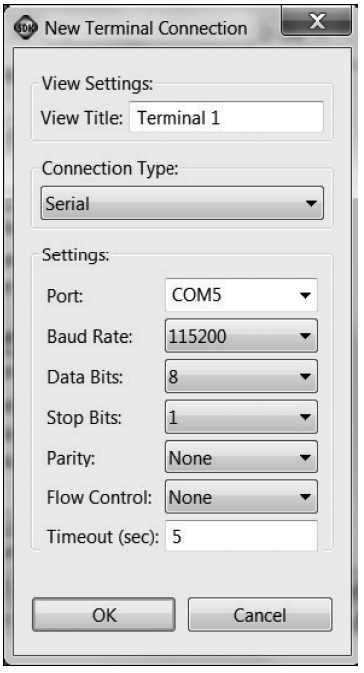

图 5-45

在计算机中以循环模式播放加噪 wav 音乐，通过音频线传输到板上进入 FPGA 滤波处理，从耳机口输出已滤除 4kHz 单音噪声的音乐。在终端界面中，键入“i”，显示滤波器系数，如图 5-46 和图 5-47 所示；键入“b”，输出未滤波前的原始加噪音乐；键入“f”，输出滤波后的无噪音乐。至此验证了整个系统，将 Atlys 板断电，退出 EDK。

7. 本系统实现思路总结

以 FPGA 为系统硬件核心，在 Atlys Spartan-6 FPGA 开发平台上，采用软、硬件结合的方式，本书提出的是一种具有灵活配置、实时性好、易于扩展等优点的 FIR 数字滤波系统，它的系统框图如图 5-48 所示。

8. 设计总结

图 5-49 为下载时的实物图，电源线连至电源口，USB 连接至 USB-PROG 接口，用音频线连接 AC97 编码器的 Line-in 与计算机的耳机输出接口，外放音箱或耳机接至 AC97 编码器的耳机接口。

图 5-50 是调试时的实物图，与下载时唯一区别是，计算机的 USB 接至 Atlys 板上的 USB-UART 口，而不 USB-PROG 口，这是为了通过这个口在 PC 上创建终端连接，从而控制 Atlys 的动作。

```
Impulse Response
Left Output(0)=378, Right Output(0)=378
Left Output(1)=73, Right Output(1)=73
Left Output(2)=-27, Right Output(2)=-27
Left Output(3)=-170, Right Output(3)=-170
Left Output(4)=-298, Right Output(4)=-298
Left Output(5)=-352, Right Output(5)=-352
Left Output(6)=-302, Right Output(6)=-302
Left Output(7)=-168, Right Output(7)=-168
Left Output(8)=-14, Right Output(8)=-14
Left Output(9)=80, Right Output(9)=80
Left Output(10)=64, Right Output(10)=64
Left Output(11)=-53, Right Output(11)=-53
Left Output(12)=-186, Right Output(12)=-186
Left Output(13)=-216, Right Output(13)=-216
Left Output(14)=-40, Right Output(14)=-40
Left Output(15)=356, Right Output(15)=356
Left Output(16)=867, Right Output(16)=867
Left Output(17)=1283, Right Output(17)=1283
Left Output(18)=1366, Right Output(18)=1366
Left Output(19)=954, Right Output(19)=954
Left Output(20)=51, Right Output(20)=51
Left Output(21)=-1132, Right Output(21)=-1132
Left Output(22)=-2227, Right Output(22)=-2227
Left Output(23)=-2829, Right Output(23)=-2829
Left Output(24)=-2647, Right Output(24)=-2647
Left Output(25)=-1633, Right Output(25)=-1633
Left Output(26)=-25, Right Output(26)=-25
Left Output(27)=1712, Right Output(27)=1712
Left Output(28)=3042, Right Output(28)=3042
Left Output(29)=-29229, Right Output(29)=-29229
Left Output(30)=3042, Right Output(30)=3042
```

图　5-46

```
Left Output(31)=1712, Right Output(31)=1712
Left Output(32)=-25, Right Output(32)=-25
Left Output(33)=-1633, Right Output(33)=-1633
Left Output(34)=-2647, Right Output(34)=-2647
Left Output(35)=-2829, Right Output(35)=-2829
Left Output(36)=-2227, Right Output(36)=-2227
Left Output(37)=-1132, Right Output(37)=-1132
Left Output(38)=51, Right Output(38)=51
Left Output(39)=954, Right Output(39)=954
Left Output(40)=1366, Right Output(40)=1366
Left Output(41)=1283, Right Output(41)=1283
Left Output(42)=867, Right Output(42)=867
Left Output(43)=356, Right Output(43)=356
Left Output(44)=-40, Right Output(44)=-40
Left Output(45)=-216, Right Output(45)=-216
Left Output(46)=-186, Right Output(46)=-186
Left Output(47)=-53, Right Output(47)=-53
Left Output(48)=64, Right Output(48)=64
Left Output(49)=80, Right Output(49)=80
Left Output(50)=-14, Right Output(50)=-14
Left Output(51)=-168, Right Output(51)=-168
Left Output(52)=-302, Right Output(52)=-302
Left Output(53)=-352, Right Output(53)=-352
Left Output(54)=-298, Right Output(54)=-298
Left Output(55)=-170, Right Output(55)=-170
Left Output(56)=-27, Right Output(56)=-27
Left Output(57)=73, Right Output(57)=73
Left Output(58)=378, Right Output(58)=378
Left Output(59)=0, Right Output(59)=0
Left Output(60)=0, Right Output(60)=0
```

图　5-47

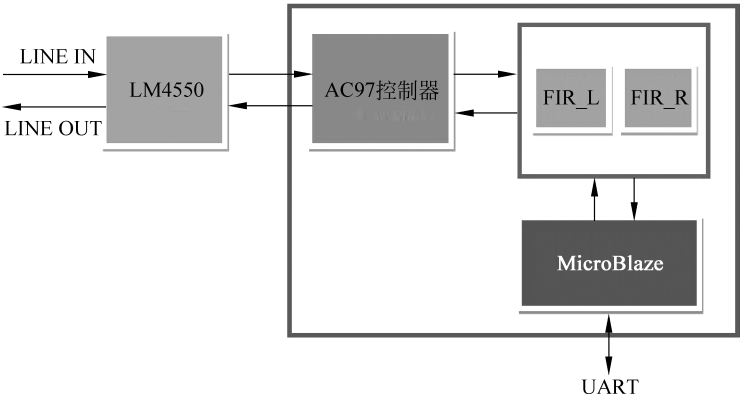

图 5-48

图 5-49

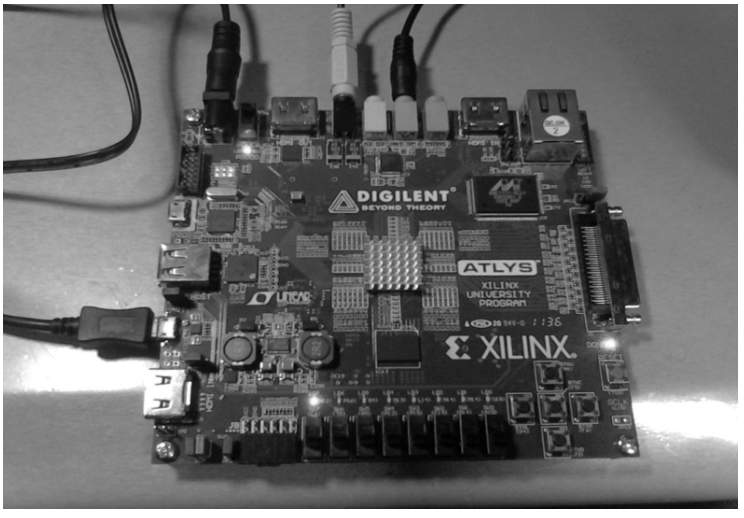

图 5-50

最终顺利实现了预想的效果，播放加了 4kHz 噪声的 CD 音乐传输到 Atlys 上进行滤波处理后，通过耳机播放时，没有了刺耳的 4kHz 的噪声，音乐质量明显提高，滤波系统性能基本达到要求。

5.3　专题三：Linux 系统搭建与移植

1. 专题介绍

在 Zedboard 上搭建一个简单的 Linux 系统，继而在 Zedboard 上移植一个有图形界面的 Linaro Ubuntu。

2. 专题目标

了解 Zedboard 的硬件情况。

制作 Zedboard 的 Linux 引导，配置编译 Zedboard 的 Linux 内核，学习制作 ramdisk 根文件系统，测试完成的 Linux 系统。

学习为 Ubuntu 配置硬件，学习编译 Ubuntu 的 Linux 内核，学习为 Zedboard 上的 Ubuntu 搭建文件系统，测试 Zedboard 上的 Ubuntu Linux。

3. 专题步骤

(1) 了解 Zedboard。
(2) 生成 Linux 启动引导文件。
(3) 编译 Linux 内核。
(4) 制作 ramdisk 根文件系统。
(5) 测试系统，以及控制 GPIO。
(6) 为 Linaro Ubuntu 配置硬件。
(7) 编译 Linux 内核。
(8) 生成设备树文件。
(9) 给 SD 卡分区。
(10) 拷贝 Linaro Ubuntu 文件系统。
(11) 连接计算机，启动测试 Ubuntu。

5.3.1　Zedboard 简述

如图 5-51 所示的 Zedboard 是基于 Xilinx Zynq 的低成本开发板，可以运行 Linux、Android 及 Windows 等系统。此外，可扩展接口使得用户可以方便访问 PS 和 PL。Zynq 将 ARM 处理系统和与可编程逻辑完美地结合在一起，创建独特强大的设计。Zedboard 以 zynq7Z020 为核心，分为 PS 和 PL 两个部分。PS(processingsystem)部分包括 zynq 内的两个 CortexA9 硬核，搭载 2 片 256MB 共 512MB 的 DDR3 内存，另外还有与 MIO 直接相连的外设，如 USB-UART，CortexA9 USB-OTG，Enet，SDcard，Quad-flash 等。PL(programable logic)部分包括 zynq 的可编程逻辑块以及与之相连的一些外设例如 VGA，HDMI，8 个 LED 和 8 个开关等。

5.3.2　生成 Linux 启动引导文件 boot.bin

boot.bin 文件的生成依赖三个文件：XPS 硬件工程生成的 bit 文件(可以通过 generate

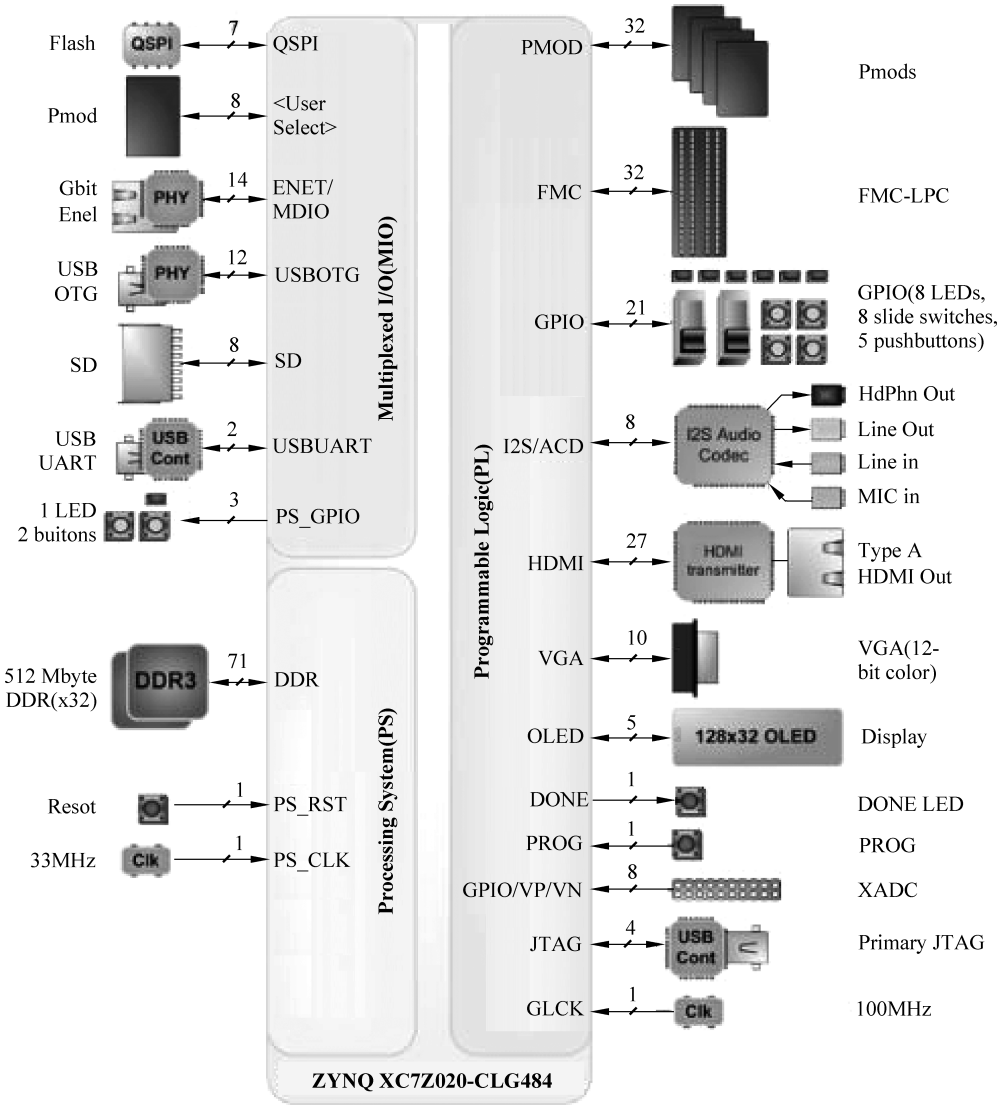

图 5-51

bitstream 生成),xsdk 生成的 first stage bootloader 文件和 u-boot 文件。

(1) bit 格式文件可以在 xps 中由 generate bitstream 功能生成。

(2) Fsbl 文件生成办法是,在 xps 中把硬件编好后,输出(export)到 SDK,在 SDK 中,建立 new project,OS platform 选 standalone,在提供的参考工程中选择 fsbl 模板,完成。Sdk 会自动根据 xps 的硬件配置生成 fsbl 文件,大致的名称为 fsbl.elf.

(3) U-boot 文件生成办法,在 Xilinx-wiki 有 u-boot-xlnx 的详细说明:

```
http://www.wiki.xilinx.com/U-boot
http://www.wiki.xilinx.com/Build+U-Boot
```

下载 Xilinx 官方编辑过的 u-boot-xlnx.git 包,命令如下:

```
~ $ :git clone git://git.xilinx.com/u-boot-xlnx.git
```

选择分支，选择的分支要对应 Xilinx ISE 的版本。命令如下：

```
~ $ :cd u - boot - xlnx && git checkout  - b xilinx - 14.2 - build1 - trd xilinx - 14.2\ - build1
- trd
```

其中 14.2-build1 可根据实际情况替换成 14.1-build1，14.3-build2。

为 Zedboard 配置 u-boot 编译选项，命令如下：

```
~ $ :make distclean && make zynq_zed_config
```

Zynq_zed_config 的具体配置选项可以在 include/configs/zynq_zed.h 中查看到。

编译生成目标

```
~ $ :make - j
```

编译完成后，拷贝目录下的 u-boot 文件，重命名成 u-boot.elf，这个就是我们需要的最终文件，使用 file 命令可以查看文件的格式：

```
~ $ :file u - boot
u - boot: ELF 32 - bit LSB shared object, ARM, EABI5 version 1 (SYSV), statically linked,
not stripped
```

准备好了这三个文件后，就可以使用 Xilinx SDK 自带的 create zynq boot image 功能生成 boot.bin 文件了。打开 SDK，单击菜单栏上 xilinx tools 和 create zynq boot image，选择 create new bif file，依次添加 3 个文件，设置目标目录，完成，就可以生成 u-boot.bin 和 u-boot.mcs 文件。把 u-boot.bin 文件重命名成 boot.bin，就是我们需要的引导文件了。

5.3.3　编译 Linux 内核

生成引导文件之后，另外两个重要的文件就是内核和设备树文件。Xilinx 提供打过补丁的 Linux 内核，可以由如下命令获取：

```
~ $ :git clone git://git.xilinx.com/linux - xlnx.git
```

选择 14.2 分支(也可以选择其他分支例如 14.1、14.3 等)：

```
~ $ :git checkout  - b xilinx - 14.2 - build1 - trd xilinx - 14.2 - build1 - trd
```

拷贝 Xilinx 已经预先设置好的内核设置选项：

```
~ $ :make ARCH = arm xilinx_zynq_defconfig
```

正式开始编译内核：

```
~ $ :make ARCH = arm
```

编译完成后，在./arch/arm/boot 目录下会产生 zImage 文件，就是我们需要的内核了。

另外，对于 Zedboard 来说，由于存在 PL 模块，需要一个设备树文件，来给系统提供相应的 PL 地址等信息。

在 $KERNEL/arch/arm/boot/dts 文件夹下，提供有很多现成对应不同硬件设置的 dts 的文件，可以在这些 dts 文件的基础上根据实际需要修改，最后由这些 dts 文件生成需

要的设备树文件的命令是：

```
~ $ : $ KERNEL/scripts/dtc/dtc -O dtb -I dts -o devicetree.dtb devicetree.dts
```

其中 devicetree.dts 是指要转换的 dts 文件，是可以阅读的文件，生成的 dtb 文件是二进制格式，机器可以阅读的格式，生成的 dtb 文件需要重命名成 devicetree.dtb 文件。

5.3.4 制作 ramdisk 根文件系统

生成内核引导文件和内核后，要正常运行起系统，只缺少一个根文件系统了。在最简 Linux 系统下，可以制作一个 ramdisk 文件系统。

Xilinx 官方合作服务商 Avnet 提供了一个简单的 32MB ramdisk 制作教程，由 busybox，dropbear 和相关的启动文件，系统设置文件组成。

(1) busybox 是一个集成了一百多个最常用 Linux 命令和工具的软件，包含了一些简单的工具和一些更大、更复杂的工具，虽然可能对这些工具的高级功能有所删减，但是常用的必需的功能完全具备。有些人将 busybox 称为 Linux 工具里的瑞士军刀。busybox 就好像是个工具箱，它集成了 Linux 的许多工具和命令，并对这些工具和命令尽可能地压缩以减小体积，满足嵌入式的要求，它将许多具有共性的小版本的 UNIX 工具结合到一个单一的可执行文件。这样的集合可以替代大部分 PC 上的常用工具，适用于小型尤其是嵌入式系统。

busybox 的安装如下。

busybox 的 git 网站上提供了 busybox 包，可以 clone 得到：

```
~ $ :git clone git://git.busybox.net/busybox
```

以默认的 arm 选项配置 busybox：

```
~ $ :make ARCH = arm defconfig
```

对默认的编译选项进行一些修改：

```
~ $ :make menuconfig
```

这是一个简单的图形化的编译选项选择界面，与 Linux 内核的 make menuconfig 类似，在里面可以按照需求增加或者删减 busybox 的一些功能，其中，需要设置编译后 busybox 输出的目录，如下：

```
Busybox Settings -- >
Installation Options -- >
BusyBox installation prefix -- >
```

设置为 ramdisk 的文件夹目录，例如～/rootfs。

正式编译：

```
~ $ :make ARCH = arm install
```

编译结束后，在 ramdisk 根目录文件夹下(例～/rootfs)可以看到，已经生成了/bin /sbin 等文件夹，里面有非常多的命令工具，而且它们全都是指向 busybox 这个可执行程序的链接。

(2) dropbear是一个开源的相对较小的SSH(security shell,安全shell)服务器和客户端。它运行在基于POSIX的各种平台,是特别有用于"嵌入"式的Linux(或其他Unix)系统,功能是提供安全shell服务,就是为用户提供安全的登录,验证用户的权限。

Dropbear的安装如下。

到dropbear官网下载最新的源代码包,例如作者下载的最新源码包是dropbear-0.53.1.tar.gz.

解压到当前目录:

```
~ $ :tar - zxvf dropbear - 0.53.1.tar.gz
```

配置编译选项:

```
~ $ : ./configure -- prefix = ~/rootfs -- host = arm - xilinx - linux - gnueabi -- disable -
zlib \
LDFLAGS = " - Wl, -- gc - sections" CFLAGS = " - ffunction - sections\
 - fdata - sections - Os"
~ $ :make PROGRAMS = "dropbear dbclient dropbearkey scp \
dropbearconvert" MULTI = 1 strip
```

编译安装:

```
~ $ :make install
~ $ :ln - s sbin/dropbear ~/rootfs/usr/bin/scp(创建超链接scp服务由dropbear来提供)
```

安装完毕之后,可以在rootfs文件夹下发现sbin/dropbear可执行程序,不是指向busybox的链接。

生成dropbear可执行程序之后还没有结束,需要产生一些用户登录的密钥等信息,以备用户安全登录时验证:

```
~ $ :mkdir etc etc/dropbear
```

Avnet公司提供了一份已经可以使用的dropbear配置(在Zedboard网站上可以下载,里面相关章节附件里),可以直接拷贝到~/rootfs中。

到这已经可以使用了,但是由于密钥是avnet提供的,不是很安全。有需要的话,可以自己重新生成密钥:

```
~ $ :dropbearkey - t rsa - f dropbear_rsa_host_key
~ $ :dropbearkey - t dss - f dropbear_dss_host_key
```

把生成的密钥文件拷贝覆盖原来的文件就可以了。

安装完busybox和dropbear之后ramdisk文件系统基本就完成了,剩下的工作就是添加一些必要的文件夹和系统必需的文件了。

(3) 补充其他必要文件。

创建系统根目录必要的文件目录:

```
~ $ : mkdir dev etc/init.d mnt opt proc root sys tmp var var/log var/www
```

建立库文件链接:

```
~ $ : ln - s /lib/libz.so.1.2.7 /lib/libz.so
~ $ : ln - s /libz.so.1.2.7 /libz.so.1
~ $ : ln - s /usr/lib/libcrypto.so.1.0.0 /usr/lib/libcrypto.so
```

复制一些 xilinx arm 工具包提供的库：

```
$ cp /opt/CodeSourcery/Sourcery_CodeBench_Lite_for_Xilinx_\
GNU_Linux/arm - xilinx - linux - gnueabi/libc/lib/ * lib
$ cp /opt//CodeSourcery/Sourcery_CodeBench_Lite_for_Xilinx_\
GNU_Linux/arm - xilinx - linux - gnueabi/libc/usr/lib/ * usr/lib
$ cp /opt//CodeSourcery/Sourcery_CodeBench_Lite_for_Xilinx_\
GNU_Linux/arm - xilinx - linux - gnueabi/libc/sbin/ * sbin
$ cp /opt//CodeSourcery/Sourcery_CodeBench_Lite_for_Xilinx_\
GNU_Linux/arm - xilinx - linux - gnueabi/libc/usr/bin/ * usr/bin
```

对库文件做尽可能的压缩：

```
$ arm - xilinx - linux - gnueabi - strip lib/ *
$ arm - xilinx - linux - gnueabi - strip usr/lib/ *
```

编辑系统挂载文件 fstab：

```
$ gedit etc/fstab
```

内容如下：

```
LABEL = //       tmpfs    defaults          0   0
none   /dev/pts  devpts   gid = 5,mode = 620 0   0
none   /proc     proc     defaults          0   0
none   /sys      sysfs    defaults          0   0
none   /tmp      tmpfs    defaults          0   0
```

编辑启动文件 inittab：

```
$ gedit etc/inittab
```

内容如下：

```
::sysinit:/etc/init.d/rcS
# /bin/ash
#
# Start an askfirst shell on the serial ports
ttyPS0::respawn: - /bin/ash
# What to do when restarting the init process
::restart:/sbin/init
# What to do before rebooting
::shutdown:/bin/umount - a - r
```

建立系统密钥文件(此密钥由 Avnet 提供)：

```
$ gedit etc/passwd
```

内容如下：

```
root: $ 1 $ qC.CEbjC $ SVJyqm.IG.gkElhaeM.FD0:0:0:root:/root:/bin/sh
```

建立启动初始化文件：

```
$ gedit etc/init.d/rcS
```

内容如下：

```
#!/bin/sh
echo "Starting rcS..."
echo " ++
mount -t
mount -t
mount -t
Mounting filesystem"
proc none /proc
sysfs none /sys
tmpfs none /tmp
echo " ++ Setting up mdev"
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
mkdir -p /dev/pts
mkdir -p /dev/i2c
mount -t devpts devpts /dev/pts
echo " ++ Starting telnet daemon"
telnetd -l /bin/sh
echo " ++ Starting http daemon"
httpd -h /var/www
echo " ++ Starting ftp daemon"
tcpsvd 0:21 ftpd ftpd -w /&
echo " ++ Starting dropbear (ssh) daemon"
dropbear
echo "rcS Complete"
```

修改 rcS 文件和文件系统的权限：

```
$ sudo chmod 755 etc/init.d/rcS
$ sudo chown -R root *
$ sudo chgrp -R root *
```

到此，ramdisk 根文件系统已经全部完成，把这个文件系统做成 ramdisk 就可以使用了。

注：这个 ramdisk 是按照 avnet 官网教程做的，由于第 3 步把 Xilinx 提供的库文件全部拷贝了，实际完成后的大小超过 32MB，无法压缩成 32MB 的 ramdisk，其解决办法是删去一些没有用的文件，比如/usr/lib/下有个 locale 文件夹，放的各种语言文件，删去了就小了很多，差不多满足了 32MB 的要求。

(4) 把文件系统做成 ramdisk。

```
$ dd if=/dev/zero of=~/ramdisk32M.image bs=1024 count=32768
$ mke2fs -F ramdisk32M.image -L "ramdisk" -b 1024 -m 0
$ tune2fs ramdisk32M.image -i 0
$ mkdir ramdisk
```

```
$ sudo mount -o loop ramdisk32M.image ramdisk/
$ sudo cp -R _rootfs/* ramdisk
$ sudo umount ramdisk/
$ gzip -v9 ramdisk32M.image
```

5.3.5 测试系统,控制 GPIO

经过以上的步骤,一个基于 ramdisk 的 Linux 系统已经完成了。拷贝以上步骤生成的 boot.bin,devicetree.dtb,zImage,ramdisk32M.image 到 SD 卡 fat32 分区插入 Zedboard 的 SD 卡插槽,连接 Zedboard 电源,连接 Zedboard 的 USB-UART 口到计算机,给 Zedboard 加电,打开串口监视软件 minicom,过十几秒钟(Zedboard 在开机自检和根据 boot.bin 配置 PS,PL 部分),OLED 屏左右的两个指示灯亮,串口输出信息,最后出现如图 5-52 状态。

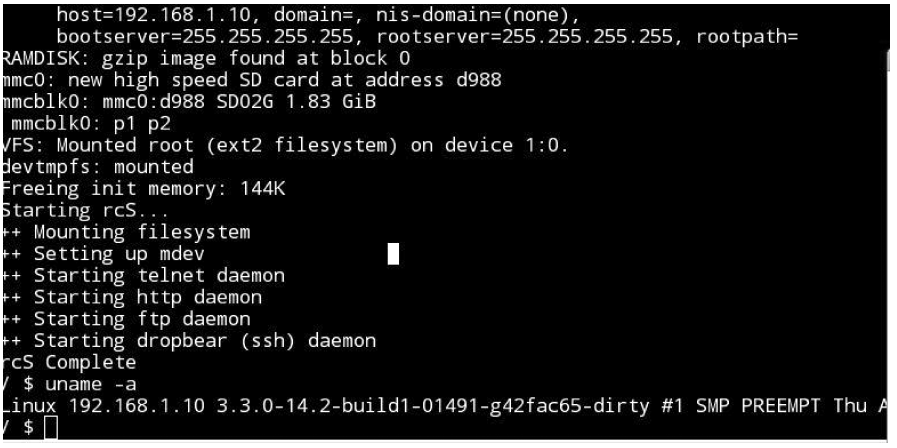

图 5-52

1. 在 Linux 系统下操作 PS 部分的 GPIO

OLED 屏幕右边有个 LED 灯 LD9,以及两个按键 BTN8,BTN9,它们都是属于 PS 部分的,可以在运行起来的 Linux 系统下直接操作。

2. 操作 LED 灯 LD9

查看 Zedboard 官方手册(例 ZedBoard_HW_UG_v1_9.pdf),LD9 小灯占用的是 PS 模块的 MIO7,在 Zedboard 板上也做了 MIO7 的标注,在 XPS 中也可以看到 MIO7 是只能输出不能做输入的一个引脚。因此对它的操作如下:

把 MIO7 设置为 export 口:

```
~ $ : echo 7 > /sys/class/gpio/export
```

小灯是只输出不输入设备,设置方向为输出:

```
~ $ : echo out > /sys/class/gpio/gpio7/direction
```

给小灯的状态赋值,1 亮 0 灭:

```
~ $ : echo 1 > /sys/class/gpio/gpio7/value
```

3. 操作按键 BTN8 BTN9

与操作 MIO 的 LED 类似，不同之处只是按键是只输入不输出的设备。在 Zedboard 板上，BTN8、BTN9 的旁边也标注了 MIO50 和 MIO51。值得注意的是，在 XPS 环境下，Xilinx 默认是把 MIO50、MIO51 作为 I2C 接口引脚的。

把 MIO50、MIO51 设置为 export 口：

```
echo 50 > /sys/class/gpio/export
echo 51 > /sys/class/gpio/export
```

设置为只输入不输出的设备：

```
echo in > /sys/class/gpio/gpio50/direction
echo in > /sys/class/gpio/gpio51/direction
```

读取按键状态：

```
cat /sys/class/gpio/gpio50/value /sys/class/gpio/gpio51/value
```

当按下按键时，相对应的按键读取到的数值会变成 1，松开后为 0。

4. 在 Linux 系统下操作 PL 部分的 GPIO

由于 PL 部分的 GPIO 不是直接连到 MIO 上，所以 PS 部分不能直接操作，需要添加相应的驱动模块。Avnet 公司给操作 8 个 LED 小灯做了示例。

在 Avnet 的 Zedboard 培训教程上可以找到小灯驱动模块补丁，该补丁的文件名如下：0001-Xilinx-ARM-Driver-for-LED-brightness-device. patch.

给内核打上 Avnet 提供的 PL 部分 LED 的驱动补丁：

```
~ $ : cd linux - xlnx
~ $ : git apply 0001 - Xilinx - ARM - Driver - for - LED - brightness - device. patch
```

选上 LED 模块选项并编译：

```
~ $ : make ARCH = arm
```

这时会出现询问是否编译进内核/编译为模块/不编译，三个询问依次选择 yes yes module，也可以直接在. config 文件中修改：

```
~ $ : vim .config
```

把相关选项改成如下所示：

```
CONFIG_PL = y
CONFIG_PL_DEBUG = y
CONFIG_LEDBRIGHTNESS = m
```

编译完成后，拷贝 arch/arm/boot/zImage 和 drivers/pl/led-brightness. ko 文件（这里新编译出的 zImage 和前面的 zImage 一样都是可以用的）。

编译出带 LED 设备地址说明的设备树文件，只有在设备树文件中列出的设备才能被内核识别并操作。

```
~ $ : mv   zedboard - 2cpu - 667mhz - 512mb - timer - uart - sdhci - usb - eth - staticip -
```

```
32Mramdisk.dts devicetree.dts
~ $ : gedit devicetree.dts
```

按照里面的格式，加入几行内容，保存。

```
(
led - brightness@41200000 {
compatible = "avnet,led - brightness";
reg = < 0x41200000 0x20 >;
};
)
```

重新生成 dtb 文件：

```
~ $ : ./scripts/dtc/dtc - O dtb - I dts - o devicetree.dtb devicetree.dts
```

以上步骤完成之后，把 zImage 文件和 led_brightness. ko，devicetree. dtb 文件拷进 SD 卡，从 SD 卡启动。

由于 ramdisk 是根目录，并且 led_brightness 是编译为模块而不是直接嵌入内核，因此需要挂载 SD 卡 fat32 分区并装载 led_brightness 驱动。

```
~ $ : mount /dev/mmcblk0p1 /mnt/
~ $ : insmod /mnt/led - brightness.ko
```

就会在 dev 目录下出现/dev/led-brightnesss 设备。

改变 LED 灯的亮度：

```
cd /mnt/driver_test_bench
./run_led_brightness_test_bench.sh 1
```

5.3.6 为 Linaro Ubuntu 配置硬件

在 Analog 网站上可以下载到提供的硬件配置，此时下载到的文件名是 cf_adv7511_zed_edk_14_4_2013_02_05. tar. gz，已经做好了 adv7511 芯片和可编程逻辑块 GPU 等资源的硬件设置，直接 generate bitstream 就可以使用，如果希望自己从零开始做出来的话，在 zedboard. org 网站上 Reference Designs 板块上有个 Building a Zynq Video Design from Scratch 教程，提供了非常详细的实现视频设计的方法，按照它的做法做出来也是可以用的。限于本书的篇幅及为了方便，这里直接使用 analog_device_inc 提供的硬件配置，有时间可以照此详细地做一遍。

生成 Generate bitstream 之后，生成 system. bit，及 export to sdk，生成对应的 fbsl，加上原先的 u-boot. elf，使用 xilinx_tools 的 create zynq boot image 功能产生 boot. bin 文件。需要注意的是，由于现在使用一个真正的 ext4 文件系统作为根文件系统而非 ramdisk，因此需要对 u-boot 做改动，在～/u-boot-xlnx/include/configs/目录下编辑 zynq_zed. h 头文件，去掉

```
fatload mmc 0 0x800000 ramdisk.img.gz;
```

这一行，重新编译出就可以。Xilinx 最新提供的版本默认使用 uImage 内核和 uramdisk 根

文件系统，还改了这个设置的位置到 zynq_common.h 中了，在 common.h 文件中做类似修改就可以。uImage 和 uramdisk 是带 u-boot 信息头的 zImage 和 ramdisk，前两者可以由后两者方便产生，生成 uramdisk 的部分可以见 xilinx wiki。

5.3.7　编译 Linux 内核

按照 analog 网站的指导，下载 Linux 内核

```
~ $ : git clone git://github.com/analogdevicesinc/linux.git ubuntu
~ $ : cd ubuntu
```

选择 xcomm_zynq 分支，这个分支是带 hdmi 显示模块的内核分支：

```
~ $ : git checkout xcomm_zynq
~ $ : make ARCH = arm distclean
```

配置为 analog 提供的 adv7511 的内核编译选项：

```
~ $ : make ARCH = arm zync_xcomm_adv7511_defconfig
```

这里比教程上要多一步，设置 config 文件中：

```
CONFIG_XILINX_FIXED_DEVTREE_ADDR = y
```

原因是后来启动时出现了以下错误：

```
Error: unrecognized/unsupported machine ID (r1 = 0x0fb71dd0)
Available machine support:
ID (hex)      NAME
00000d32   Xilinx Zynq Platform
Please check your kernel config and/or bootloader
```

注：原因网上说是 Linux 内核和 uboot 的版本不同导致的，在如下网页可以找到解决办法 http://ez.analog.com/message/87877。

编译内核：

```
~ $ : make ARCH = arm
```

5.3.8　生成设备树 devicetree

```
~ $ : make ARCH = arm zynq - zed - adv7511.dtb
~ $ : rename zynq - zed - adv7511.dtb devicetree.dtb
```

在这个设备树 dts 文件当中，也可以注意到根文件系统的变化，有一行：

```
bootargs = "console = ttyPS0, 115200 root = /dev/mmcblk0p2 rw earlyprintk rootfstype = ext4
rootwait     devtmpfs.mount = 0";
```

使用 ramdisk 文件系统时，这一行是这样的：

```
bootargs = "console = ttyPS0, 115200 root = /dev/ram rw initrd = 0x1100000, 32M ip = :::::
eth0:dhcp
```

```
earlyprintk";
```

以上设定了信息的串口输出、根文件系统设备、类型、IP 等信息。

5.3.9 给 SD 卡分区

Linux 下可以用 gparted，fdisk 等分区工具给 SD 卡分区，要求分为两个主分区，一个是 fat32 格式，存放 boot. bin，devicetree. dtb，zImage 等文件给 Zedboard 读取（Xilinx 为 Zedboard 烧录的最初始的保密的那段代码只支持读取 fat32 格式的分区），另外一个是 ext4 分区，存放 linaro-ubuntu 的文件，作为系统的根文件系统。

5.3.10 拷贝 Linaro Ubuntu 文件系统

Linaro Ubuntu 是一个开源项目，主要是致力于 Ubuntu，Android 在嵌入式以及开发板上的实现，它提供的 Lianro Ubuntu 可以用在 Zedboard 板上。在

```
http://releases.linaro.org/11.12/ubuntu/oneiric-images/ubuntu-desktop
```

可以找到很多不同时间版本的 Linaro Ubuntu 版本，这里下载使用 analog 推荐的 linaro-precise-ubuntu-desktop-20120626-247. tar. gz.

可以使用 analog 推荐的命令解压复制到 SD 卡 ext4 分区（挂载路径为/media/rootfs）：

```
~ $ : sudo tar --strip-components = 3 -C /media/rootfs -xzpf linaro-o-ubuntu-\
desktop-tar-20111219-0.tar.gz binary/boot/filesystem.dir
```

作者推荐解压后使用 rsync 命令：

```
~ $ : gunzip linaro-precise-ubuntu-desktop-20120626-247.tar.gz
~ $ : tar -xvf linaro-ubuntu linaro-precise-ubuntu-desktop-20120626-247.tar
~ $ :cd linaro-ubuntu/binary/boot/filesystem.dir/
~ $ : sudo rsync -a ./ /media/rootfs
~ $ : sudo rsync -a ./ /media/rootfs
~ $ : sudo rsync -a ./ /media/rootfs
```

最后一个命令多重复几遍，是因为 SD 卡速度慢，复制根文件系统需要的时间比较长，很容易在拷贝文件的时候发生错误。

注：作者开始拷的两遍都缺少文件或者没有拷贝正确，以致于无法启动，不知道原因，费了很多时间。使用 rsync 同步命令多次可以保证文件拷贝的正确性。

5.3.11 连接计算机屏幕，启动测试 Ubuntu

将制作完成的 SD 卡插入 Zedboard SD 插槽（fat32 分区三个文件：boot. bin，zImage，devicetree. dtb，ext4 分区是 linaro 的根文件系统），连接电源，UART，加电，经过大约几分钟启动，可以验证效果。