PLD 期末课程设计

基于 NIOS II 处理器的闹钟设计

姓名：董校廷　　学号：SC21002046　　完成时间：2021-12-25

一、功能简介

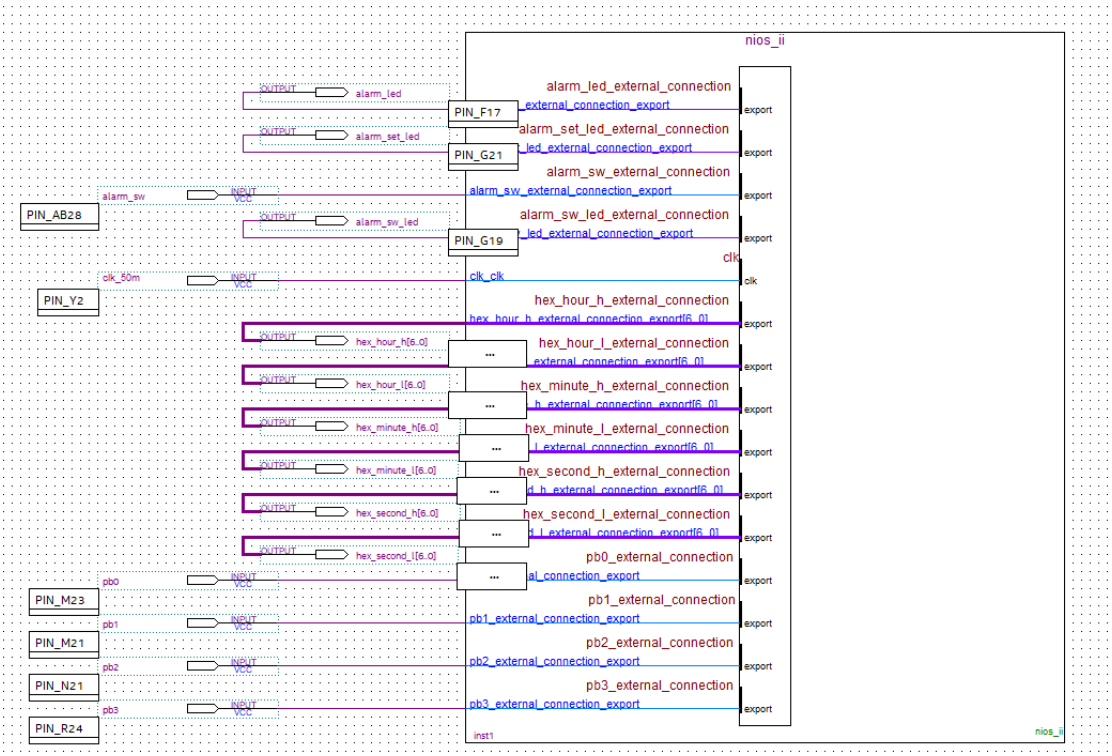
在 FPGA 开发板 DE2-115 上设计了一个 NIOS II 嵌入式软核，并将 ROM、RAM、JTAG、PIO、TIMER 等外设通过 Avalon 总线搭载在软核上，驱动数码管、LED、按键、滑动开关等完成闹钟系统的功能设计。

本系统实现的功能有 24 小时计时时钟与闹钟两部分；时钟可以进行最低精度为 1 秒的时间设置；闹钟的设置精度为 1 分钟，并由滑动开关控制闹钟开启与关闭，同时，可以取消当前设定的闹钟；如果时钟的小时位和时钟的分钟位和闹钟的设定都对应相等，并且滑动控制开关在开启位置，即进入闹钟响起状态（由 LED 闪烁表示），闹钟最长持续时间为 1 分钟。

二、实现方案与详细设计

1、硬件部分：

硬件电路的 block design 如下图：



模块的硬件部分由一个外部 50Mhz 时钟源信号，一个 NIOS II 处理器，三个指示 LED，六个 7 段数码管、四个 Push Button、一个 Slide Switch 组成：

NIOS II 处理器设定内核频率为 50Mhz，由于 DE2-115 的开发板拥有 50Mhz 时钟源，因此不需要进行 PLL 进行时钟分频/倍频，直接将 clk 配置到对应管脚上即可。

六个 7 端数码管（HEX5--0）以两个为一组，分别显示时：分：秒。

四个 Push Button（KEY3--0）用来进行时间设置、闹钟设置、移动修改光标、加/减设定值。

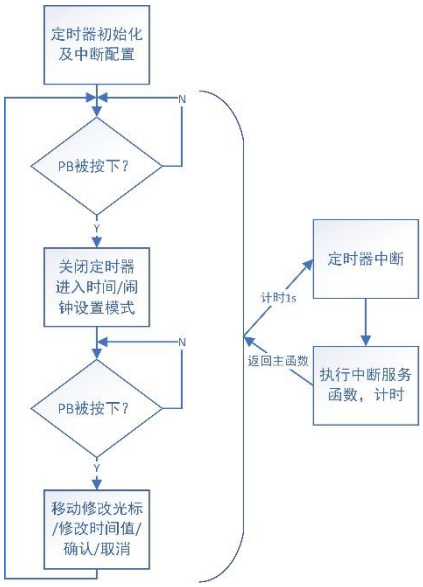
一个 Slide Switch（SW0）用来控制已设定的闹钟是否开启，如果位置在上方，即为开启。
三个 LED（LEDG7、LEDR0、LEDG8）分别用来指示闹钟是否已设定、已设定的闹钟是否开启（由 SW 控制）、闹钟响起（LEDG8 闪烁）。

NIOS II 处理器软核外设如下图：

Use	...	Name	Description	Export	Clock	Base	End	IRQ
☑		clk_0	Clock Source		*exported*			
☑		nios2_gen2_0	Nios II Processor		clk_0	0x0008_0800	0x0008_0fff	
☑		ram	On-Chip Memory (RAM or ROM) I...		clk_0	0x0006_0000	0x0007_8fff	
☑		rom	On-Chip Memory (RAM or ROM) I...		clk_0	0x0004_0000	0x0005_8fff	
☑		jtag_uart_0	JTAG UART Intel FPGA IP		clk_0	0x0008_1218	0x0008_121f	0
☑		sysid_qsys_0	System ID Peripheral Intel FP...		clk_0	0x0008_1210	0x0008_1217	
☑		hex_hour_h	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0008_11f0	0x0008_11ff	
☑		hex_hour_l	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0008_11e0	0x0008_11ef	
☑		hex_minute_h	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0008_11d0	0x0008_11df	
☑		hex_minute_l	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0008_11c0	0x0008_11cf	
☑		hex_second_h	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0008_11b0	0x0008_11bf	
☑		hex_second_l	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0008_11a0	0x0008_11af	
☑		pb0	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0008_1190	0x0008_119f	
☑		pb1	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0008_1180	0x0008_118f	
☑		pb2	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0008_1170	0x0008_117f	
☑		pb3	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0008_1160	0x0008_116f	
☑		alarm_set_led	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0008_1150	0x0008_115f	
☑		alarm_sw	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0008_1140	0x0008_114f	
☑		alarm_sw_led	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0008_1130	0x0008_113f	
☑		alarm_led	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0008_1120	0x0008_112f	
☑		timer	Interval Timer Intel FPGA IP		clk_0	0x0008_1020	0x0008_103f	1

两个片上 Memory 分别设置为 ram 和 rom；ram 用来存储程序运行过程的临时数据如堆栈；rom 则用来存放不会轻易擦除的数据如程序代码。
JTAG UART 用来向 NIOS II 处理器中下载软件代码以及串口通信。
此外还有一个 Interval Timer 用来进行定时器计时，这里设定默认值为计数 1s。
通过最右侧的 IRQ 可以看到 UART 中断向量号为 0，TIMER 中断向量号为 1。

2、软件部分：
程序逻辑框架如下：



首先进行定时器初始化及其中断配置，开启定时器中断，接着进入 while(1)循环中，不断进行显示（数码管时间显示、LED 闹钟指示）和按键扫描（扫描到不同的按键被按下进入不同的状态）；每 1s 产生一次定时器中断，进入中断处理函数，对计时变量（时、分、秒）进行修改，然后退出中断处理函数，返回主函数。

部分核心代码：

定时器初始化及中断配置：

```
void init_timer()
{
    void* isr_context_ptr = (void*) &timer_isr_context;
    //1s-0x2FAF07F
    alt_ic_isr_register(TIMER_IRQ_INTERRUPT_CONTROLLER_ID, TIMER_IRQ, timer_interrupts, isr_context_ptr, 0x0);
    IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE, 7);
    hour = 0;
    minute = 0;
    second = 0;
}
```

注册定时器中断处理函数为 timer_interrupt()，对定时器的控制寄存器写 7（0b0111）开启定时器工作，清零时间变量。

32bit 定时器的寄存器结构如下图，其中 periodl 和 periodh 存储的是定时器的初始计数值的低 16bit 和高 16bit，该定时器通过递减实现，因此如果要实现 1s 的定时，初值应该设置为 (1 / T – 1) = (f - 1) = (50M – 1) = 49999999 = 0x2FAF07F；即 periodl = 0xF07F，periodh = 0x02FA；由于在设计 nios ii 时默认值已设置为 1s，这里不用再配置。

Table 25-3: Register Map—32-bit Timer

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

查阅手册，定时器控制寄存器意义如下：

Table 25-6: control Register Bits

Bit	Name	R/W/C	Description
0	ITO	RW	If the ITO bit is 1, the interval timer core generates an IRQ when the status register's TO bit is 1. When the ITO bit is 0, the timer does not generate IRQs.
1	CONT	RW	The CONT (continuous) bit determines how the internal counter behaves when it reaches zero. If the CONT bit is 1, the counter runs continuously until it is stopped by the STOP bit. If CONT is 0, the counter stops after it reaches zero. When the counter reaches zero, it reloads with the value stored in the period registers, regardless of the CONT bit.
2	START (1)	W	Writing a 1 to the START bit starts the internal counter running (counting down). The START bit is an event bit that enables the counter when a write operation is performed. If the timer is stopped, writing a 1 to the START bit causes the timer to restart counting from the number currently stored in its counter. If the timer is already running, writing a 1 to START has no effect. Writing 0 to the START bit has no effect.
3	STOP (1)	W	Writing a 1 to the STOP bit stops the internal counter. The STOP bit is an event bit that causes the counter to stop when a write operation is performed. If the timer is already stopped, writing a 1 to STOP has no effect. Writing a 0 to the stop bit has no effect. If the timer hardware is configured with **Start/Stop control bits** off, writing the STOP bit has no effect.

写入控制寄存器为 7，即 ITO = 1，当状态寄存器的 TO 位 1 时会产生 IRQ 中断信号；CONT = 1，即计数器会在减计数至 0 的下一个周期进行重新载入计数值循环计数，直到 STOP 为 1 停止，进行重复计数；START = 1，开始内部定时器的运行。

定时器中断处理函数代码：

```
void timer_interrupts()
{
    IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_BASE, 0);

    if(second >= 59)
    {
        second = 0;
        if(minute >= 59)
        {
            minute = 0;
            if(hour >= 23)
            {
                hour = 0;
            }
            else
            {
                hour++;
            }
        }
        else
        {
            minute++;
        }
    }
    else
    {
        second++;
    }
```

进入中断处理函数，首先将状态寄存器写 0，接着对时间变量进行对应修改。

Table 25-5: status Register Bits

Bit	Name	R/W/C	Description
0	TO	RC	The TO (timeout) bit is set to 1 when the internal counter reaches zero. Once set by a timeout event, the TO bit stays set until explicitly cleared by a master peripheral. Write zero to the status register to clear the TO bit.
1	RUN	R	The RUN bit reads as 1 when the internal counter is running; otherwise this bit reads as 0. The RUN bit is not changed by a write operation to the status register.

根据状态寄存器表，TO 会在计数器自减到 0 后置 1，即每秒产生一个中断信号，进入中断函数后清 0 是为了保证下一次定时器中断可以正常产生。

数码管显示代码：

```
    unsigned int _hour_l = hour % 10;
    unsigned int _hour_h = hour / 10;
    unsigned int _minute_l = minute % 10;
    unsigned int _minute_h = minute / 10;
    unsigned int _second_l = second % 10;
    unsigned int _second_h = second / 10;
```

```
#define HEX_HOUR_H_SHOW(data) IOWR_ALTERA_AVALON_PIO_DATA(HEX_HOUR_H_BASE, data)
#define HEX_HOUR_L_SHOW(data) IOWR_ALTERA_AVALON_PIO_DATA(HEX_HOUR_L_BASE, data)
```

```
    HEX_HOUR_H_SHOW(hex_decoder[_hour_h]);
    HEX_HOUR_L_SHOW(hex_decoder[_hour_l]);
```

数码管显示首先要将高位和低位分离开，接着通过一个数组存储字符‘0’~‘9’的 7 段数码管译码值，将译码值输出到对应 PIO 即可。

修改光标显示及闹钟 LED 闪烁代码：

```
if(light_cnt >= 30000)
{
    light_cnt = 0;
    light_flag = (light_flag == 0) ? 1 : 0;
}
else
{
    light_cnt ++;
}
```

```
if(index == 0)
{
    if(light_flag == 0)
    {
        HEX_HOUR_H_OFF;
        HEX_HOUR_L_OFF;
    }
    else
    {
        HEX_HOUR_H_SHOW(hex_decoder[_hour_h]);
        HEX_HOUR_L_SHOW(hex_decoder[_hour_l]);
    }
    HEX_MINUTE_H_SHOW(hex_decoder[_minute_h]);
    HEX_MINUTE_L_SHOW(hex_decoder[_minute_l]);
    HEX_SECOND_H_SHOW(hex_decoder[_second_h]);
    HEX_SECOND_L_SHOW(hex_decoder[_second_l]);
}
```

以时钟设置模式为例，系统通过闪烁对应的数码管指示修改光标所在的位置，通过在 while(1) 中设置一个计数值和点亮 flag 来控制修改光标 index 所在位置数码管的闪烁。
同理，闹钟 LED 闪烁也是用这种方式实现的。

按键检测代码：

```
int getPB3()
{
    if(PB3_VALUE == PB_DOWN)
    {
        delay_ms(10);
        if(PB3_VALUE == PB_DOWN)
        {
            while(PB3_VALUE == PB_DOWN);
            return 1;
        }
    }
    return 0;
}
```

代码进行了软件消抖，当检测到按键按下，延时 10ms（延时函数通过循环计数粗略实现），再次检测按键是否按下，如果是，则认为按键被按下，等到按键松开，返回捕捉按键按下成功值。

按键实现的功能逻辑：
在正常计数状态下不断捕捉按键被按下的信号，进入时间设置/闹钟设置模式，停止定时器

运行（对控制寄存器的 STOP 位进行写入 1）：

```
#define TIMER_START IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE, 7)
#define TIMER_STOP IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE, 8)
```

在对应模式下，通过 index 定位修改光标，检测按键，修改时/分/秒的设定值，在确认设置之后，开启定时器运行。以修改时间为例，代码如下：

```
else if(state == 1)
{
    if(getPB2())//move
    {
        idx = (idx == 2) ? 0 : idx + 1;
    }
    else if(getPB1())//add
    {
        if(idx == 0)
        {
            hour = (hour == 23) ? 0 : hour + 1;
        }
        else if(idx == 1)
        {
            minute = (minute == 59) ? 0 : minute + 1;
        }
        else if(idx == 2)
        {
            second = (second == 59) ? 0 : second + 1;
        }
    }
    else if(getPB0())//sub
    {
        if(idx == 0)
        {
            hour = (hour == 0) ? 23 : hour - 1;
        }
        else if(idx == 1)
        {
            minute = (minute == 0) ? 59 : minute - 1;
        }
        else if(idx == 2)
        {
            second = (second == 0) ? 59 : second - 1;
        }
    }
    else if(getPB3())//ok
    {
        printf("time set succeed: %.2d:%.2d:%.2d\n", hour, minute, sec
        idx = 0;
        state = 0;
        IOWR_ALTERA_AVALON_TIMER_PERIODH(TIMER_BASE, 0x02FA);
        IOWR_ALTERA_AVALON_TIMER_PERIODL(TIMER_BASE, 0xF07F);
        IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_BASE, 0);
        TIMER_START;
        TIMER_IRQ_ENABLE;
    }
}
```

三、模块仿真

本模块采用 nios ii 处理器实现，因此仿真利用 UART 串口 printf 相关提示信息，来验证模块的逻辑功能正确。

首先每一次进入中断会打印当前时间精确到秒；对于按键进入不同设定模式都会打印一次信息，在设定完成后也会输出设定的具体数组。

截取的控制台输出截图如下：

```
alarm clock based on nios ii by dongxiaoting
TIME— 00:00:01
TIME— 00:00:02
TIME— 00:00:03
```

首先是通过一条字符串显示程序下载成功，接着开始正常计时。

```
alarm setting mode
alarm set succeed: 07h 10m
TIME— 00:00:04   ALARM TIME— 07h 10m
TIME— 00:00:05   ALARM TIME— 07h 10m
```

接着按下 PB2 进入闹钟设定状态，在设定成功后输出信息：设定闹钟时间位 7 点 10 分，这时闹钟设定成功，在继续计时的同时也会显示闹钟时间。

```
time setting mode
time set succeed: 07:09:50
TIME— 07:09:51   ALARM TIME— 07h 10m
TIME— 07:09:52   ALARM TIME— 07h 10m
TIME— 07:09:53   ALARM TIME— 07h 10m
TIME— 07:09:54   ALARM TIME— 07h 10m
TIME— 07:09:55   ALARM TIME— 07h 10m
```

按下 PB3 进入时间设定状态，设定时间为 07:09:50，在设定时间的基础上继续进行计时。

```
TIME— 07:09:58   ALARM TIME— 07h 10m
TIME— 07:09:59   ALARM TIME— 07h 10m
ALARM SWITCH IS ON
TIME— 07:10:00   ALARM TIME— 07h 10m
ALARM SWITCH IS ON
THE ALARM CLOCK RANG...
TIME— 07:10:01   ALARM TIME— 07h 10m
ALARM SWITCH IS ON
THE ALARM CLOCK RANG...
TIME— 07:10:02   ALARM TIME— 07h 10m
ALARM SWITCH IS ON
THE ALARM CLOCK RANG...
TIME— 07:10:03   ALARM TIME— 07h 10m
ALARM SWITCH IS ON
THE ALARM CLOCK RANG...
```

在进入闹钟设定时间前，提前开启该闹钟提醒的开关（sw0），计时在进入 7 点 10 分之后，闹钟响起。

```
TIME— 07:10:05   ALARM TIME— 07h 10m
ALARM SWITCH IS ON
THE ALARM CLOCK RANG...
alarm setting mode
alarm cancel
TIME— 07:10:06
TIME— 07:10:07
TIME— 07:10:08
```

按下 PB2 进入闹钟设置模式，再次按下 PB3 取消闹钟，闹钟停止，计时正常进行。

四、实测结果

下载程序，设定时间为 07：10：32。

开启闹钟，设定时间为 07h11min，LEDG7 亮起，闹钟设定成功。

将 SW0 拨到上方，LEDR0 亮起，此时闹钟开关打开。

时钟运行到 07：11：00，闹钟响起，LEDG8 闪烁。

将 SW0 拨到下方，LEDG8 不再闪烁，此时闹钟开关关闭。

取消闹钟设定，这时即使闹钟开关打开，LEDG8 也不再闪烁。

五、总结

相比于使用组合逻辑/时序逻辑模块设计的电路，使用 NIOS II 处理器的方便之处在于更容易地写出一个状态机的逻辑框架，这是由于使用嵌入式处理器更加地高层次抽象化，我们只需要利用 C 语言描述清楚整个模块的逻辑即可；这样缺点就是处理器操作精度比较低，例如想要定时具体时间，就必须采用内部定时器来实现，通过 C 代码并不能直接触及时钟/门电路这一层级。

因此对于 PLD 和 SOPC 的使用需要有机地结合，例如在计算机网络中，较低的层如数据链路层的一些数据流，使用 PLD 来设计模块进行数据流传输，精度和吞吐率、速率都会比使用处理器更好，而对于底层网络这些模块工作模式的配置和切换，主机与主机之间的交互状态，在会话层或更高的应用层等，使用 SOPC 进行处理会更方便。

在开发过程中我也有一些体会，例如在模块开发前尽量设计好所需要的资源，比如需要多少个 PIO 接口，需不需要定时器、串口等等，这样就不需要在开发过程中再重复去修改 NIOS II，同时要考虑一些模块设置，例如定时器的计数值位宽需要多少等；在调用这些外设的过程中，比较关键的是要去读它的寄存器表，需要理解寄存器内设置的意义，这样才能顺利使用该外设。

对于本模块的改进有两个方向：

1、 将对于按键的检测更换为按键中断，主函数只进行相关显示，涉及到中断的嵌套，中断优先级要高于定时器中断；目前采用的是重复读取检测，在逻辑比较复杂地情况下会导致按键检测不够灵敏，同时由于按键延时消抖也大量占用了处理器资源。

2、 扩展定时器数量，最终的闹钟响起如果利用定时器可以实现非固定频率的闪烁效果（类似于闹钟不同的音效）。