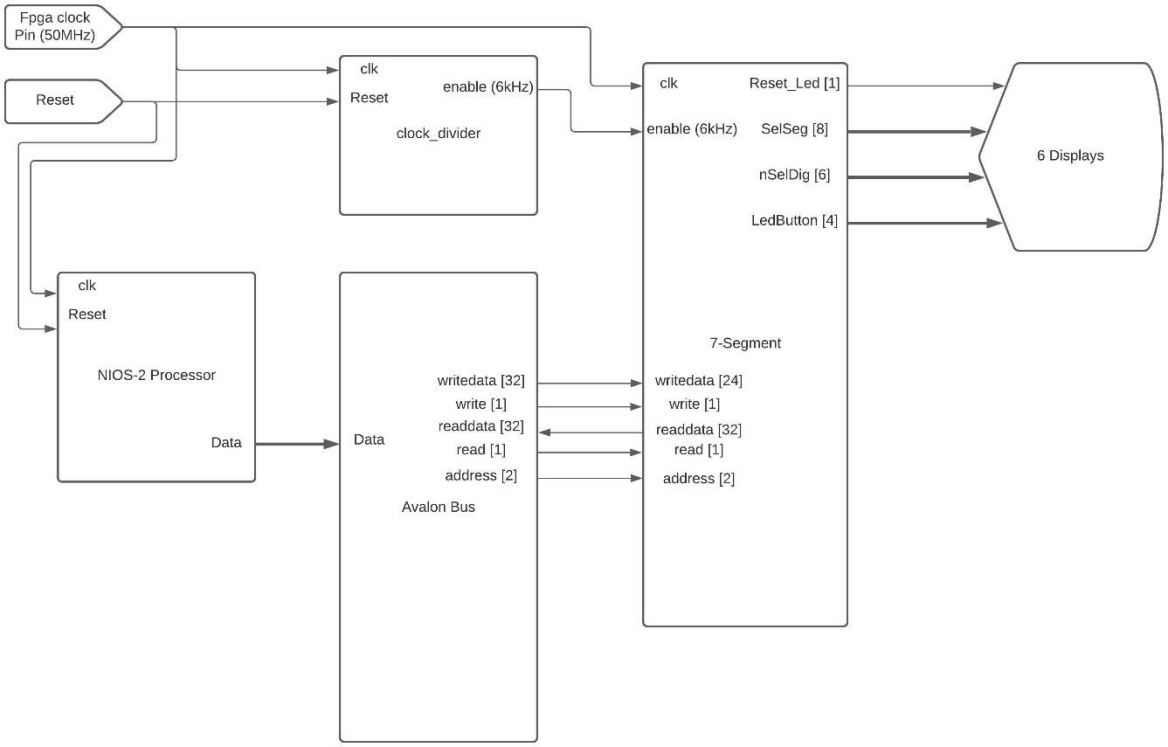
Embedded Systems – Lab 2.2: Custom Slave Programmable Interface

Introduction

For this second Embedded Systems Lab, we will be creating an RTC on an FPGA. The aim is to develop application specific hardware by using VHDL.

High level description



The schematic illustrated above represents our initial plan for the architecture. Some modifications have been made along the way, like merging the clock_divider and the 7-Segment units.

The seven-segment displays can be switched between displaying minutes-seconds-centiseconds and hour-minutes-seconds. It can also display a custom 6-digit hex number provided by the processor.

We will be using the NIOS-2 as our softcore processor and it will be programmed in C and can transfer data via the Avalon Bus to the seven-segment display. The RTC can be set from the processor by providing it with an 8-digit hexadecimal number (32 bits) representing the time in the format “hhmmsscc”.

Component

This part will look in detail at the VHDL definition of the Custom Slave Programmable Interface.

Clock Dividers

```
58      -- 6kHz clock divider
59
60      process(clk, nReset)
61      begin
62
63          if nReset='0' then
64              count_6kHz <= 0;
65              enable_6kHz <= '0';
66
67          elsif rising_edge(clk) then
68              count_6kHz <= count_6kHz + 1;
69              if count_6kHz = 8333 then
70                  enable_6kHz <='1';
71                  count_6kHz <= 0;
72              else
73                  enable_6kHz <='0';
74              end if;
75          end if;
76      end process;
77
78      -- 100Hz clock divider
79
80      process(clk, nReset)
81      begin
82
83          if nReset='0' then
84              count_100Hz <= 0;
85              enable_100Hz <= '0';
86
87          elsif rising_edge(clk) then
88              count_100Hz <= count_100Hz + 1;
89              if count_100Hz = 499999 then
90                  enable_100Hz <='1';
91                  count_100Hz <= 0;
92              else
93                  enable_100Hz <='0';
94              end if;
95          end if;
96      end process;
97
```

The first clock divider divides the 50Mhz clock into a 6kHz clock. This is used for the refresh rate of the displays. Each display will be alternatively refreshed after a 1ms interval. The 100Hz clock is used to control the real time clock. Each tick increments the RTC by one centisecond.

The period for each clock divider must be 50*10^6 / 6*10^3 = 8333.3 clock cycles for the 6kHz clock divider, and 50*10^6 / 100 = 500000 clock cycles for the 100Hz clock divider.

Avalon Slave write to registers

```
-- Avalon slave write to registers
process(clk, nReset, address, writedata, write)
begin
    if nReset = '0' then
        disp_mode <= "00";
        cpu_disp <= x"000000";
        centi_u <= "0000";
        centi_d <= "0000";
        seconds_u <= "0000";
        seconds_d <= "0000";
        minutes_u <= "0000";
        minutes_d <= "0000";
        hours_u <= "0000";
        hours_d <= "0000";
    elsif rising_edge(clk) then
        if write = '1' then
            case address is
              when "00" =>
                  disp_mode <= writedata(1 downto 0);
              when "01" =>
                  cpu_disp <= writedata(23 downto 0);
              when "10" =>
                  centi_u <= writedata(3 downto 0);
                  centi_d <= writedata(7 downto 4);
                  seconds_u <= writedata(11 downto 8);
                  seconds_d <= writedata(15 downto 12);
                  minutes_u <= writedata(19 downto 16);
                  minutes_d <= writedata(23 downto 20);
                  hours_u <= writedata(27 downto 24);
                  hours_d <= writedata(31 downto 28);

              when others => null;
            end case;
        elsif enable_100Hz = '1' then

                    -- Centiseconds
                    centi_u <= centi_u + "0001";
                    if centi_u = "1001" then
                        centi_u <= "0000";
                        centi_d <= centi_d + "0001";
                        if centi_d = "1001" then
                            centi_d <= "0000";
                            -- Seconds
                            seconds_u <= seconds_u + "0001";
                            if seconds_u = "1001" then
                                seconds_u <= "0000";
                                seconds_d <= seconds_d + "0001";
                                if seconds_d = "0101" then
                                    seconds_d <= "0000";
                                    -- Minutes
                                    minutes_u <= minutes_u + "0001";
                                    if minutes_u = "1001" then
                                        minutes_u <= "0000";
                                        minutes_d <= minutes_d + "0001";
                                        if minutes_d = "0101" then
                                            minutes_d <= "0000";
                                            hours_u <= hours_u + "0001";
                                            if hours_u = "1001" then
                                               hours_u <= "0000";
                                               hours_d <= hours_d + "0001";
                                            elsif hours_d = "0010" and hours_u = "0011" then
                                                  hours_u <= "0000";
                                                  hours_d <= "0000";
                                            end if;
                                        end if;
                                    end if;
                                end if;
                            end if;
                        end if;
                    end if;

        end if;
```

This part handles the input connection from the Avalon Bus. The address enables the NIOS-2 processor to write in 3 different input registers. The first one is the display mode (disp_mode) which lets us decide what the 7-segment displays. It can either be the time in minutes/seconds/centiseconds or in hours/minutes/seconds, and it can also display a hexadecimal 6-digit number given by the processor.

The second one is the 6-digit hexadecimal number the processor want to display and is called “cpu_disp”.

The third one is the 8-digit hexadecimal representing the time (hhmmsscc) and is used to set the time in the Real Time Clock.

Avalon slave read registers

```
-- Avalon slave read registers

process(clk, read)
begin
   if rising_edge(clk) then
      readdata <= (others => '0');
      if read = '1' then
         case address is
            when "00" => readdata(1 downto 0) <= disp_mode;
            when "01" => readdata(23 downto 0) <= cpu_disp;
            when "10" =>
            readdata <= full_time;

            when others => null;
         end case;
      end if;
   end if;
end process;
```

This part handles the output connection to the Avalon Bus. The address works the same way as for the "write". The main purpose of this part is to enable the processor to get the time by reading the "full_time" data in the RTC slave.

Applying the display mode chosen

```
-- Decoding min/sec/centi/hours/cpu_disp to the displayed hex_num

process(clk, nReset)
begin

   if nReset = '0' then
      hex_num <= (others => '0');

   elsif rising_edge(clk) then
      case disp_mode is
         when "00" =>
            hex_num(3 downto 0) <= centi_u;
            hex_num(7 downto 4) <= centi_d;
            hex_num(11 downto 8) <= seconds_u;
            hex_num(15 downto 12) <= seconds_d;
            hex_num(19 downto 16) <= minutes_u;
            hex_num(23 downto 20) <= minutes_d;
         when "01" =>
            hex_num(3 downto 0) <= seconds_u;
            hex_num(7 downto 4) <= seconds_d;
            hex_num(11 downto 8) <= minutes_u;
            hex_num(15 downto 12) <= minutes_d;
            hex_num(19 downto 16) <= hours_u;
            hex_num(23 downto 20) <= hours_d;
         when "10" =>
            hex_num(23 downto 0) <= cpu_disp;
         when others =>
            hex_num(23 downto 0) <= x"ABCDEF"; --Debugging
      end case;

   end if;
end process;
```

This part is used to fill the displayed register "hex_num" with the registers we want to display.

The three possibilities are the same as before: minutes/seconds/centiseconds or hours/minutes/seconds or a custom 6-digit hexadecimal named "cpu_disp".

This process takes care of showing the correct information on the displays as a function of disp_mode. The first display mode (00) is the chronometer mode, displaying minutes, seconds and centiseconds. The second display mode (01) is the time mode, displaying hours, minutes and seconds. The third display mode (10) is the custom display mode. This mode displays the hex number passed from the processor. The last display mode (11) is used for debugging purposes.

Transforming a hexadecimal into a 7-Segment logic

```
-- Decoding : hex -> 7 Segments

process(clk)
begin
   if rising_edge(clk) then
        case hex_led is
           when "0000" => SelSeg <= "00111111"; -- "0"
           when "0001" => SelSeg <= "00000110"; -- "1"
           when "0010" => SelSeg <= "01011011"; -- "2"
           when "0011" => SelSeg <= "01001111"; -- "3"
           when "0100" => SelSeg <= "01100110"; -- "4"
           when "0101" => SelSeg <= "01101101"; -- "5"
           when "0110" => SelSeg <= "01111101"; -- "6"
           when "0111" => SelSeg <= "00000111"; -- "7"
           when "1000" => SelSeg <= "01111111"; -- "8"
           when "1001" => SelSeg <= "01101111"; -- "9"
           when "1010" => SelSeg <= "01110111"; -- A
           when "1011" => SelSeg <= "01111100"; -- b
           when "1100" => SelSeg <= "00111001"; -- C
           when "1101" => SelSeg <= "01011110"; -- d
           when "1110" => SelSeg <= "01111001"; -- E
           when "1111" => SelSeg <= "01110001"; -- F
        end case;
   end if;
end process;
```

This process is a simple switch-case for each hex number. They are translated into its corresponding binary input to the seven-segment display. The port SelSeg is exported in the conduit and used by the 7-Segment extension to activate or not the different segments of one display.

Refreshing the 6 displays

```
-- Refresh each of the 6 displays
process(clk, nReset)
begin
   if nReset = '0' then
      Reset_Led <= '1';
      nSelDig <= (others => '1');
   elsif rising_edge(clk) then
      if enable_6kHz = '1' then
         Reset_Led <= '0';
         nSelDig <= (others => '1');
         case count_dig is
            when 0 =>
               nSelDig(0) <= '0';
               hex_led <=  hex_num(3 downto 0);
            when 1 =>
               nSelDig(1) <= '0';
               hex_led <=  hex_num(7 downto 4);
            when 2 =>
               nSelDig(2) <= '0';
               hex_led <=  hex_num(11 downto 8);
            when 3 =>
               nSelDig(3) <= '0';
               hex_led <=  hex_num(15 downto 12);
            when 4 =>
               nSelDig(4) <= '0';
               hex_led <=  hex_num(19 downto 16);
            when 5 =>
               nSelDig(5) <= '0';
               hex_led <=  hex_num(23 downto 20);
            when others => count_dig <= 0;
         end case;
         if count_dig = 5 then
            count_dig <= 0;
         else
            count_dig <= count_dig + 1;
         end if;

      end if;
   end if;
end process;
```

The 6 displays are refreshed alternatively every 1/6ms, which means it takes 1 millisecond to refresh every display once. The port nSelDig is exported in the conduit and is used by the 7-Segment extension to activate or not the different displays. The port Reset_Led is also exported and is used to reset all 6 of the 7-Segment displays.

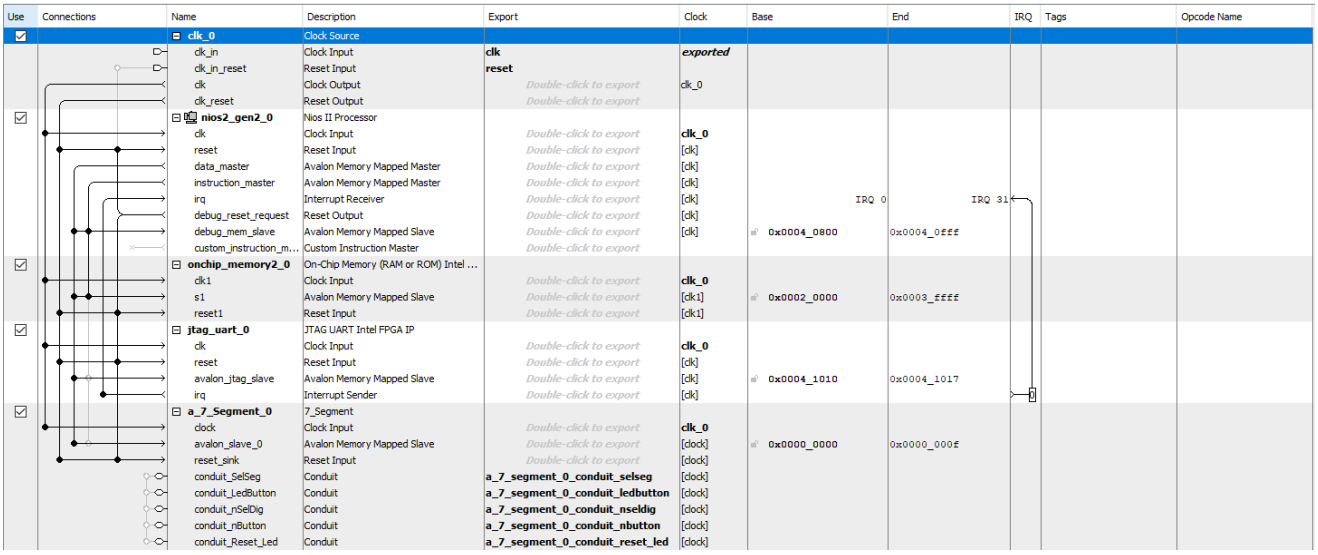
Other signals

```
full_time(3 downto 0) <= centi_u;
full_time(7 downto 4) <= centi_d;
full_time(11 downto 8) <= seconds_u;
full_time(15 downto 12) <= seconds_d;
full_time(19 downto 16) <= minutes_u;
full_time(23 downto 20) <= minutes_d;
full_time(27 downto 24) <= hours_u;
full_time(31 downto 28) <= hours_d;
LedButton <= not nButton;
```

Here we only redirect some signals, first we combine all the time signals into a “full_time” signal.

Secondly, we just use the Led next to the buttons to light when their corresponding button is pressed.

Qsys system



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags	Opcode Name
☑		**clk_0**	Clock Source							
		clk_in	Clock Input	**clk**	***exported***					
		clk_in_reset	Reset Input	**reset**						
		clk	Clock Output	*Double-click to export*	clk_0					
		clk_reset	Reset Output	*Double-click to export*						
☑		**nios2_gen2_0**	Nios II Processor							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		data_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]					
		instruction_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]					
		irq	Interrupt Receiver	*Double-click to export*	[clk]			IRQ 0		IRQ 31
		debug_reset_request	Reset Output	*Double-click to export*	[clk]					
		debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0004_0800	0x0004_0fff			
		custom_instruction_m...	Custom Instruction Master	*Double-click to export*						
☑		**onchip_memory2_0**	On-Chip Memory (RAM or ROM) Intel ...							
		clk1	Clock Input	*Double-click to export*	**clk_0**					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]	0x0002_0000	0x0003_ffff			
		reset1	Reset Input	*Double-click to export*	[clk1]					
☑		**jtag_uart_0**	JTAG UART Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		avalon_jtag_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0004_1010	0x0004_1017			
		irq	Interrupt Sender	*Double-click to export*	[clk]					
☑		**a_7_Segment_0**	7_Segment							
		clock	Clock Input	*Double-click to export*	**clk_0**					
		avalon_slave_0	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	0x0000_0000	0x0000_000f			
		reset_sink	Reset Input	*Double-click to export*	[clock]					
		conduit_SelSeg	Conduit	**a_7_segment_0_conduit_selseg**	[clock]					
		conduit_LedButton	Conduit	**a_7_segment_0_conduit_ledbutton**	[clock]					
		conduit_nSelDig	Conduit	**a_7_segment_0_conduit_nseldig**	[clock]					
		conduit_nButton	Conduit	**a_7_segment_0_conduit_nbutton**	[clock]					
		conduit_Reset_Led	Conduit	**a_7_segment_0_conduit_reset_led**	[clock]					

This is the system as defined in Qsys. We can see all the connections between the different components of the system, as well as the exported conduits for the 7-Segment extension. We can also see the base address of our component (0x0000_0000) which is going to be used by the processor to write/read to/from our Custom Avalon Slave.

Top Level description

```
architecture rtl of DE0_Nano_Soc_7_segment_extension is

   component system2_2 is
      port(
         clk_clk                              : in  std_logic                     := 'X';             -- clk
         reset_reset_n                        : in  std_logic                     := 'X';             -- reset_n
         a_7_segment_0_conduit_selseg_export  : out std_logic_vector(7 downto 0);                    -- export
         a_7_segment_0_conduit_ledbutton_export : out std_logic_vector(3 downto 0);                  -- export
         a_7_segment_0_conduit_nseldig_export : out std_logic_vector(5 downto 0);                    -- export
         a_7_segment_0_conduit_nbutton_export : in  std_logic_vector(3 downto 0) := (others => 'X'); -- export
         a_7_segment_0_conduit_reset_led_export : out std_logic                                      -- export
      );

end component system2_2;

begin

u0 : component system2_2
      port map(
         clk_clk                              => FPGA_CLK1_50,                -- clk.clk
         reset_reset_n                        => nButton(0),                  -- reset.reset_n
         a_7_segment_0_conduit_selseg_export  => SelSeg,
         a_7_segment_0_conduit_ledbutton_export => LedButton,
         a_7_segment_0_conduit_nseldig_export => nSelDig,
         a_7_segment_0_conduit_nbutton_export => nButton,
         a_7_segment_0_conduit_reset_led_export => Reset_Led
      );



end;
```

This is the Top-Level architecture of our system. We use the button 0 as our reset, the FPGA_CLK1_50 (50MHz) as our clock, and we export all the signals needed to use the 7-Segment extension.

Software access

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

#define BASE_ADDRESS    0x00000000
#define DISP_MODE       (4*0b00)
#define CPU_DISP        (4*0b01)
#define TIME            (4*0b10)

int main()
{

    //Sets the time of the RTC and the Display mode to hours/minutes/seconds
    IOWR_32DIRECT(BASE_ADDRESS, TIME,0x23595000);
    IOWR_32DIRECT(BASE_ADDRESS, DISP_MODE,0x00000001);



    //Reads the time and prints it in the console
    int temp;
    while(1)
    {

        temp = IORD_32DIRECT(BASE_ADDRESS, TIME);

        printf("%08x \n", temp);
        for(int i = 0; i <5000000; i++){
            ;
        }
    }

}
```

This is a simple example of a code where the processor can set and read the time to and from the RTC.

The base address is given by the qsys system, and the offsets are the same as the ones we defined in our “address” register, except they have to be multiplied by 4, as the processor uses bytes addresses and our registers are 32 bits long (4 bytes). A register takes 4 addresses, so to go on the next register, we have to offset ourselves by 4.

Conclusion

We have achieved the goal to create an RTC. We can see that even for such a simple device such as a seven-segment display, a lot of configuration has to be made with VHDL.