Ecole Polytechnique Fédérale de Lausanne

CS-473

Embedded Systems

Lab 2.2: Custom slave programmable interface

Report

Authors:
Victor Casas - 237770
Hugo Miranda Queiros - 336706

Supervisors:
Prof. René Beuchat

Due November 27, 2021

Contents

1 Documentation **3**
1.1 Problem Statement . . . 3
1.2 High level description . . . 3
1.3 Low level description . . . 4
1.4 Strategy to generate the PWM signal . . . 5

2 Writing VHDL code of PWM component and testing on ModelSim **6**
2.1 VHDL Code of PWM component . . . 6
2.2 Modelsim testing . . . 8

3 Qsys implementation and execute C-code to FPGA board **11**
3.1 Qsys implementation . . . 11
3.2 C code . . . 12

4 Conclusion **13**

List of Figures

1 ***High level diagram*** . . . 3
2 ***PIO component interface*** . . . 4
3 ***Custom PIO component implementation*** . . . 5
4 ***Clock divider: The clock divider triggers Period and Duty cycle modules periodically. Example of a PWM signal of period 20 ms and duty cycle of 1 ms*** . . . 6
5 ***PWM strategy*** . . . 7
6 ***Testbench simulation: Clock and async_reset*** . . . 9
7 ***Testbench simulation: writedata*** . . . 10
8 ***Testbench simulation: readdata*** . . . 10
9 ***Testbench simulation: PWM signal*** . . . 11
10 ***Qsys implementation.*** . . . 11
11 ***PWM signal measured directly on the GPIO pins.*** . . . 13

1 Documentation

1.1 Problem Statement

The goal of this lab is to design a custom slave IP component in the FPGA-based system. We have decided to design a PWM generator. The program has been evaluated thanks to the VHDL test bench on ModelSim and by wiring a servomotor to the FPGA board.

1.2 High level description

The PWM signal is generated by our system and routed physically to the GPIO ports of the board (we can generate until 8 similar PWM signals simultaneously on 8 different GPIO ports) . The first step of this lab is to draw block diagrams in order to understand where our custom programmable interface belongs and how it is bound to the other instances. Figure 1 shows the high level diagram of the system and the implementation of our custom programmable interface "PWM_gen", which interfaces to the processor (the Nios II CPU) through the Avalon bus.

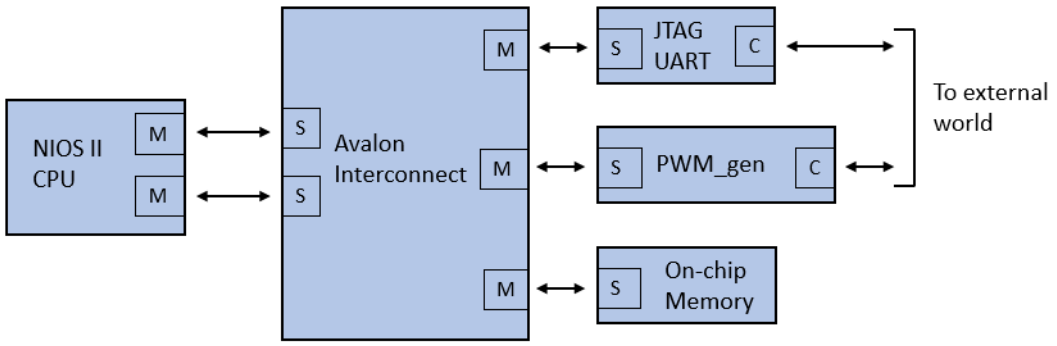


Figure 1: ***High level diagram***

Figure 2 shows PWM_gen which has a slave config interface and a peripheral specific conduit interface. The slave interface binds PWM_gen to its master interface (NIOS II CPU) and the signals are defined by Avalon. The conduit interface outputs the signals from PWM_gen to the external world.

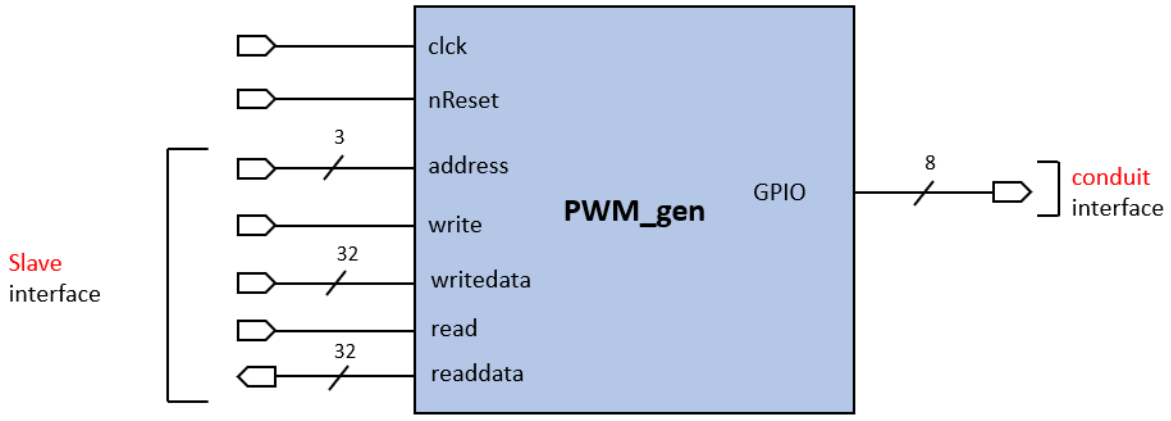


Figure 2: ***PIO component interface***

1.3 Low level description

In order to program our interface PWM_gen, we have to define the register interface:

- RegDir – A location in the register interface where the bit-level directionality of the GPIO unit can be controlled.
- RegPort – A location in the register interface that memorizes a value to be outputted by the GPIO unit on its physical pins if the direction of the GPIO unit is set to "output".
- RegPeriod - A register containing the value of clock's rising edges proportional to the value of the PWM period.
- RegDuty - A register containing the value of clock's rising edges proportional to the value of the PWM duty cycle.
- RegPolarity - A register that sets the polarity of the generated PWM signals.

Our register interface contains 5 registers, so their addresses can be stored in 3 bits. Our registers occupy addresses 0-4 and addresses 5-7 are unused. Table 1 reveals the reading and writing strategy for those registers and Figure 3 represents its associated diagram.

Address	Write register	Writedata[31...0]	Read register	Readdata[31...0]
0	RegDir	→ iRegDir	RegDir	iRegDir →
1	-	Don't care	RegPort	iRegPort →
2	RegPeriod	→ iRegPeriod	RegPeriod	iRegPeriod →
3	RegDuty	→ iRegDuty	RegDuty	iRegDuty →
4	RegPolarity	→ iRegPolarity	RegPolarity	iRegPolarity →
5	-	Don't care	-	0x00
6	-	Don't care	-	0x00
7	-	Don't care	-	0x00

Table 1: Register map of PWM_gen component

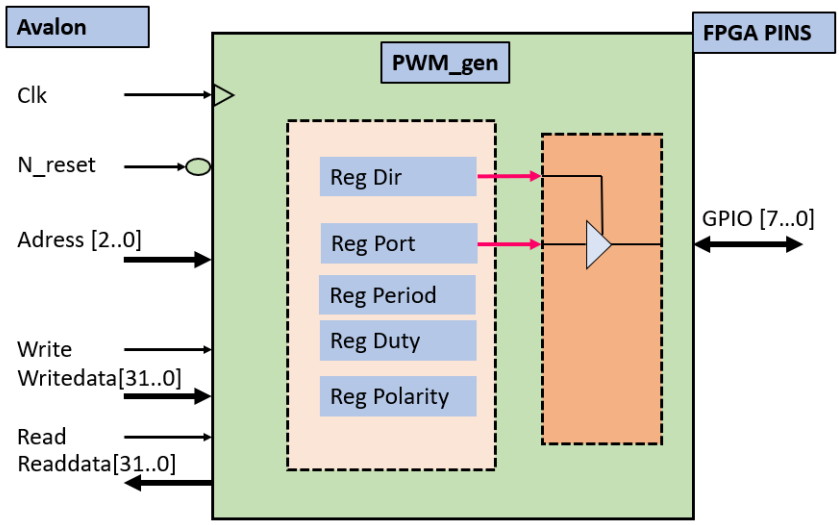


Figure 3: ***Custom PIO component implementation***

1.4 Strategy to generate the PWM signal

We want to generate a PWM signal with a 50MHz clock frequency (FPGA board). The strategy consists in incrementing a counter that counts until the period of the PWM. We decide to generate PWM signals with periods of maximum 1 second. That implies that our register has to be able to count until 50'000'000 (50MHz), which represents a variable length of 27 bits. So, our registers iRegPeriod and iRegDuty contain 27 bits as well in order to compare them to the counter. Consequently, writedata and readdata of the Avalon slave interface, have to be of size 32 bits in order to write/read in those registers.

In our situation, our servomotor demands 20 ms period and a 1-2 ms duty cycle which correspond to 1'000'000 and 50'000-100'000 clock ticks respectively (see Table2). Figure 4 shows how the GPIO port signal evolves with respect to the counter value taking a PWM signal of 1 ms duty cycle and 20 ms period.

Register	Time [ms]	number of clock ticks (counter)
iRegPeriod	20	1000000
IRegDuty	1-2	50000-100000

Table 2: ***iRegPeriod and iRegDuty time period and equivalence in number of ticks considering a FPGA's clock frequency (50 MHz).***

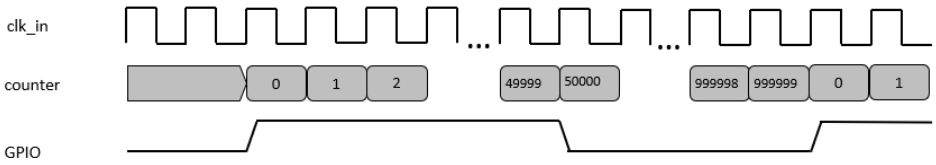


Figure 4: ***Clock divider: The clock divider triggers Period and Duty cycle modules periodically. Example of a PWM signal of period 20 ms and duty cycle of 1 ms***

2 Writing VHDL code of PWM component and testing on ModelSim

2.1 VHDL Code of PWM component

Figure 5 illustrates the logic that we have in order to produce a PWM signal. The first layer of condition verifies if the reset is active. If it is the case, GPIO ports are left undefined ('Z'). If reset is not active, then we have to set of conditions: the first set handles the counter by resetting it when reaching the final value of the period and by incrementing it otherwise. The second part sets the GPIO bits (iRegPort) to 1 (iRegPolarity) if the counter is within the duty cycle or to 0 (not(iRegPolarity)) if it is not the case. It is important to notice that if you do not provide a strictly positive value in the registers iRegPeriod and iRegDuty the GPIO ports will be left undefined ('Z').

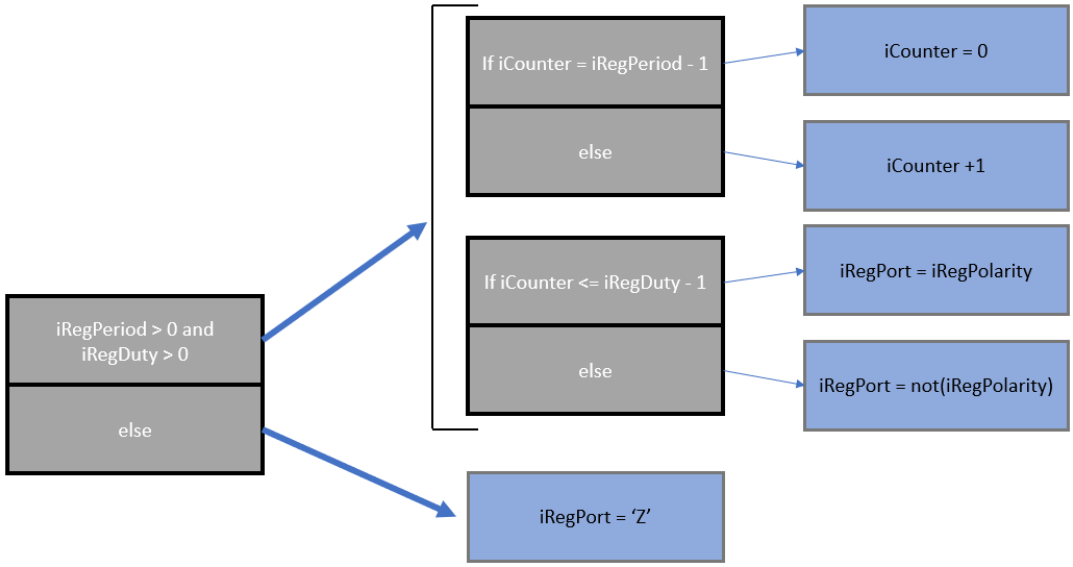


Figure 5: ***PWM strategy***

See the VHDL code below of pwm_gen.vhdl that implements the strategy shown in Figure 5.

```
51    -- PWM_gen
52    process(clk, nReset)
53    begin
54       if nReset = '0' then
55          iCounter <= (others => '0');
56          iRegPort <= (others => 'Z');
57
58       elsif rising_edge(clk) then
59          if (iRegPeriod > 0 and iRegDuty > 0) then
60             if (iCounter = iRegPeriod - 1) then
61                iCounter <= (others => '0');
62             else
63                iCounter <= iCounter + 1;
64             end if;
65
66             if (iCounter <= iRegDuty - 1) then
67                for i in 0 to 7 loop
68                   iRegPort(i) <= iRegPolarity(0);
69                end loop;
70             else
71                for i in 0 to 7 loop
72                   iRegPort(i) <= not(iRegPolarity(0));
73                end loop;
74             end if;
75
76          else
77             for i in 0 to 7 loop
78                   iRegPort(i) <= 'Z';
79             end loop;
80          end if;
81
82       end if;
83    end process;
```

Below, you can find the architecture of our components. Note that iRegPeriod and iRegDuty have 27 bits in order to compare them to iRegcounter that can count till 50 000 000 and that iRegpolarity has to be a vector in order to communicate with the Avalon bus.

```
26  architecture comp of PWM_gen is
27
28     signal iRegDir          : std_logic_vector(7 downto 0)   := (others => '0');
29     signal iRegPort         : std_logic_vector(7 downto 0)   := (others => '0');
30     signal iRegPin          : std_logic_vector(7 downto 0)   := (others => '0');
31     signal iRegPeriod       : unsigned(26 downto 0)          := (others => '0');
32     signal iRegDuty         : unsigned(26 downto 0)          := (others => '0');
33     signal iRegPolarity     : std_logic_vector(0 downto 0)   := (others => '0');
34     signal iCounter         : unsigned(26 downto 0)          := (others => '0');
35
```

Then, just after are the two basic synchronous processes that allow our component to communicate with the Avalon data bus (line 89-127).

```
88    -- Avalon slave write to registers.
89    process(clk, nReset)
90    begin
91       if nReset = '0' then
92          iRegDir <= (others => '0');
93          iRegPeriod <= (others => '0');
94          iRegDuty <= (others => '0');
95          iRegPolarity <= (others => '0');
96
97       elsif rising_edge(clk) then
98          if write = '1' then
99             case Address is
100               when "000"  => iRegDir       <= writedata(7 downto 0);
101               when "010"  => iRegPeriod    <= unsigned(writedata(26 downto 0));
102               when "011"  => iRegDuty      <= unsigned(writedata(26 downto 0));
103               when "100"  => iRegPolarity   <= writedata(0 downto 0);
104               when others => null;
105            end case;
106         end if;
107      end if;
108   end process;
```

```
111   -- Avalon slave read from registers.
112   process(clk)
113   begin
114      if rising_edge(clk) then
115         readdata <= (others => '0');
116         if read = '1' then
117            case address is
118               when "000" => readdata <= std_logic_vector(resize(unsigned(iRegDir), readdata'length));
119               when "001" => readdata <= std_logic_vector(resize(unsigned(iRegPort), readdata'length));
120               when "010" => readdata <= std_logic_vector(resize(iRegPeriod, readdata'length));
121               when "011" => readdata <= std_logic_vector(resize(iRegDuty, readdata'length));
122               when "100" => readdata <= std_logic_vector(resize(unsigned(iRegPolarity), readdata'length));
123               when others => null;
124            end case;
125         end if;
126      end if;
127   end process;
```

Finally, below you can find the asynchronous process that updates the outputted GPIO according to the signals iRegDir and iRegPort (line 38-48).

```
38    -- GPIO Port output value.
39    process(iRegDir, iRegPort)
40    begin
41       for i in 0 to 7 loop
42          if iRegDir(i) = '1' then
43             GPIO(i) <= iRegPort(i);
44          else
45             GPIO(i) <= 'Z';
46          end if;
47       end loop;
48    end process;
```

2.2 Modelsim testing

To test if the implementation of the architecture is correct we use a VHDL testbench, ModelSim. The code below (a snippet from "tb_PWM_gen-vhdl" shows how we initialize our registers (line 94-98) and the different orders we give to the simulation (line 103-114). At the beginning of the simulation, we reset our system (async_reset), then we write to the main registers (line 106-109) and finally we read those registers (line 111-114). Note that REGPERIOD and REGDUTY are initialised with small random values so that we can easily visualise the PWM signal.

```
91  begin
92
93      -- Default values
94      nReset <= '1';
95      address <= (others => '0');
96      write <= '0';
97      read <= '0';
98      writedata <= (others => '0');
99
100     wait for CLK_PERIOD;
101
102     -- Reset the circuit.
103     async_reset;
104
105     --Test
106     WR(0, 255);     -- Writes REGDIR
107     WR(4, 1);       -- Writes REGPOLARITY
108     WR(2, 10);      -- Writes REGPERIOD
109     WR(3, 3);       -- Writes REGDUTY
110
111     RD(0);          -- Reads REGDIR
112     RD(4);          -- Reads REGPOLARITY
113     RD(2);          -- Reads REGPERIOD
114     RD(3);          -- Reads REGDUTY
115
116     wait;
117 end process simulation;
```

The code beneath generates a clock signal with a CLK_PERIOD value of 100 ns (see code enclosed line 10). Figure 6 shows the clock signal.

```
40  -- Generate CLK signal
41  clk_generation : process
42  begin
43      clk <= '1';
44      wait for CLK_PERIOD / 2;
45      clk <= '0';
46      wait for CLK_PERIOD / 2;
47  end process clk_generation;
```

Below, the code of async_reset which enables the reset. It resets (n_Reset <= '0') for half a period of the clock. Figure 6 shows the reset signal which is enabled when the signal is low.

```
53  procedure async_reset is
54  begin
55      wait until rising_edge(CLK);
56      wait for CLK_PERIOD / 4;
57      nReset <= '0';
58
59      wait for CLK_PERIOD / 2;
60      nReset <= '1';
61  end procedure async_reset;
```

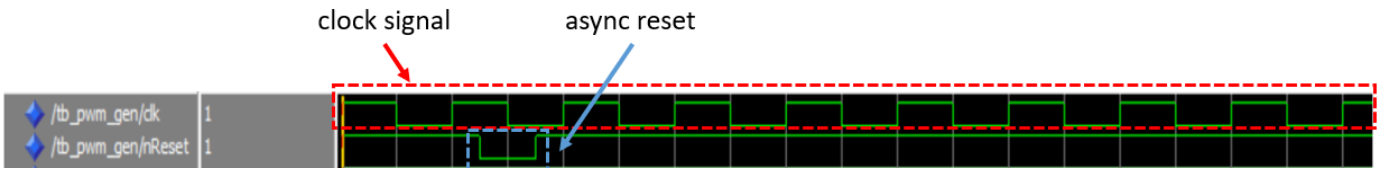


Figure 6: ***Testbench simulation: Clock and async_reset***

Beneath, the WR procedure writes to the register's address. Note that when writing to the registers (write <= '1'), we wait for the next rising edge of the clock to stop writing (write <= '0') so that it gives an extra time to the signals to spread. Figure 7 displays the writing sequence. We see well the data that has been written to the registers (255, 1, 10, 3).

```
63    procedure WR(constant REG_ID : in natural; constant data : in natural) is
64    begin
65       wait until rising_edge(CLK);
66
67       write <= '1';
68       address <= std_logic_vector(to_unsigned(REG_ID, address'length));
69       writedata <= std_logic_vector(to_unsigned(data, writedata'length));
70
71       wait until rising_edge(CLK);
72
73       write <= '0';
74       address <= (others => '0');
75       writedata <= (others => '0');
76    end procedure WR;
```

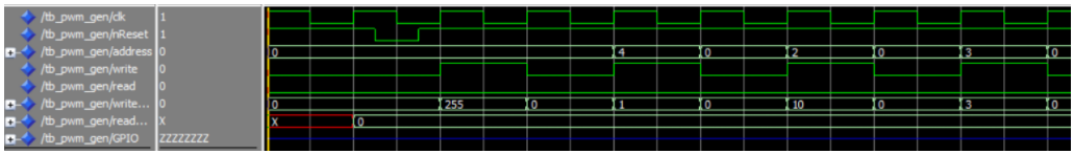


Figure 7: ***Testbench simulation: writedata***

Below, the RD procedure reads to the register's address. Similarly to WR process, when reading to the registers (read <= '1'), we wait for the next rising edge of the clock to stop writing (read <= '0') so that it gives an extra time to the signals to spread. Figure 8 displays the writing sequence. We see well how the registers have been read (255, 1, 10, 3).

```
78    procedure RD(constant REG_ID : in natural) is
79    begin
80       wait until rising_edge(CLK);
81
82       read <= '1';
83       address <= std_logic_vector(to_unsigned(REG_ID, address'length));
84
85       wait until rising_edge(CLK);
86
87       read <= '0';
88       address <= (others => '0');
89    end procedure RD;
```

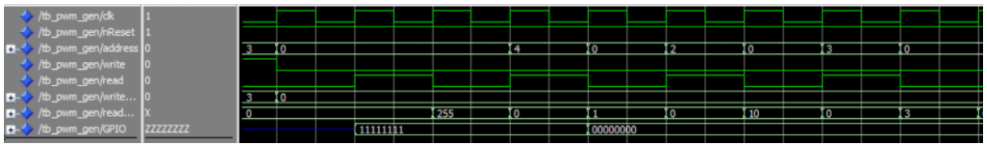
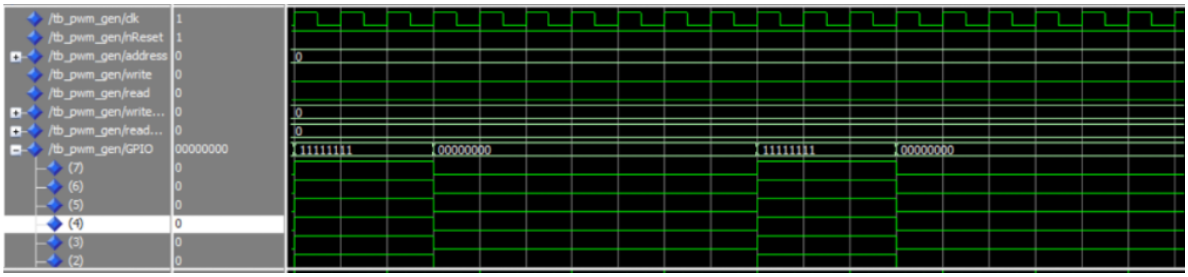
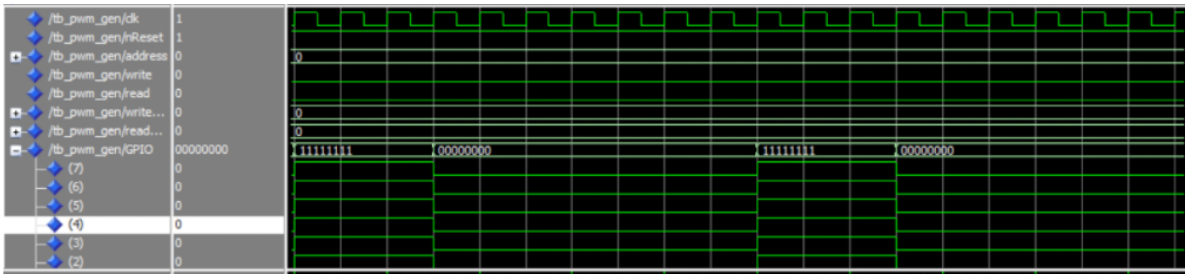


Figure 8: ***Testbench simulation: readdata***

After writing in all the needed registers (line 106-109), the PWM starts to be generated as expected in our VHDL code PWM_gen.vhdl with the values given previously. We see well how our PWM signal is generated (Figure 9(a)) for a period of 10 and a duty cycle of 3. Figure 9(b) is similar but with a polarity of 0 (REGPOLARITY = 0) to invert the PWM signal.



(a)



(b)

Figure 9: ***Testbench simulation (a) Polarity = 1; (b) Polarity = 0;***

3 Qsys implementation and execute C-code to FPGA board

3.1 Qsys implementation

After, designing and testing our custom slave interface PWM_gen, we added it in Qsys (Figure 10).

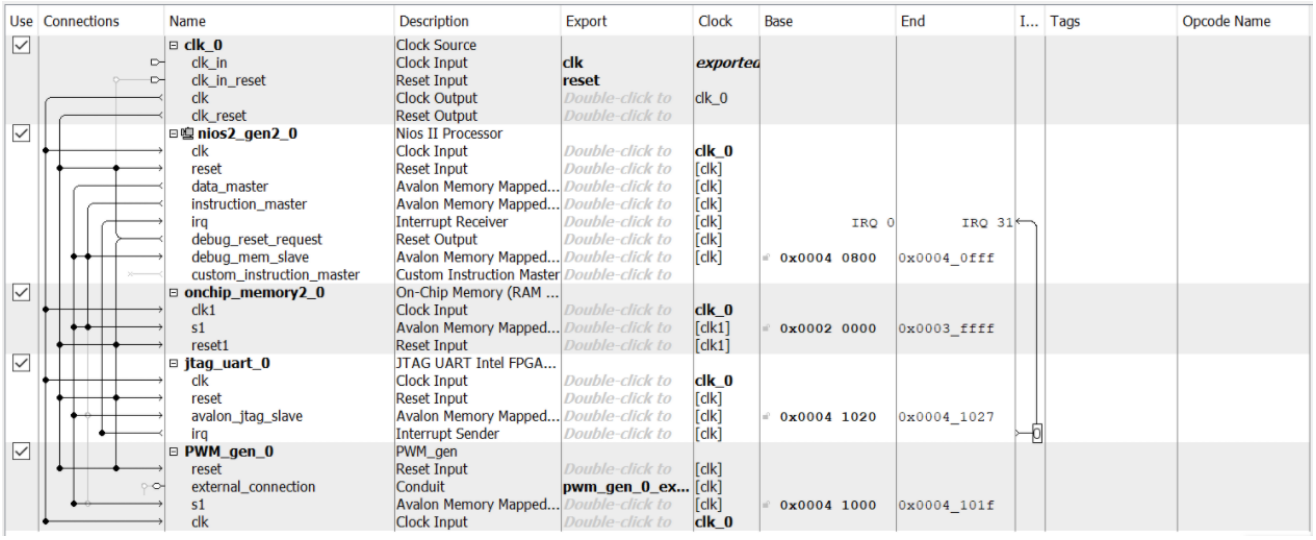


Figure 10: ***Qsys implementation.***

We added our component to the top level architecture and we mapped the ports as shown

below:

```
architecture rtl of DE0_Nano_SoC_top_level is
   component pwm_system is
      port (
         clk_clk                              : in    std_logic                    := 'X';             -- clk
         reset_reset_n                        : in    std_logic                    := 'X';             -- reset_n
         pwm_gen_0_external_connection_export : inout std_logic_vector(7 downto 0) := (others => 'X')  -- export
      );
   end component pwm_system;

begin
   u0 : component pwm_system
      port map (
         clk_clk                              => FPGA_CLK1_50,                      -- clk.clk
         reset_reset_n                        => KEY_N(0),                          -- reset.reset_n
         pwm_gen_0_external_connection_export => GPIO_0(7 downto 0)                 -- pwm_gen_0_external_connection.export
      );

end rtl;
```

3.2 C code

To test our system, we implemented the C code below that allows to control a servomotor by making it rotate infinitely between its two opposite positions. Firstly, one need to define the value of DUTY that represents the duty cycle in percentage and the value of F_wanted in Hz (here 50 Hz to control the servo motor with a 20 ms PWM) that defines the frequency of the PWM to generate. Then, the C code below writes in the wanted registers with the type of commands "IOWR_32DIRECT(BASE, OFFSET, DATA)". The base address of our component is PWM_BASE and each register has to be written by giving its offset address. The databus on the Avalon master side has 4 bytes for each master address. The first address starts at OFFSET 0, the second one at 4, the third one at 8 and so on... The values (DATA) of the IREGPERIOD and IREGDUTY are set according to Table 2. At the beginning the IREGDUTY is set so that the duty cycle is 1 ms (DUTY = 5%). Afterwards, in the loop, the duty cycle alternates between values between 1 and 2 ms (DUTY = 5% to 10%) (50000-100000 clock ticks) to change the position of the servomotor.

```c
#define IREGDIR 0
#define IREGPORT 1
#define IREGPERIOD 2
#define IREGDUTY 3
#define IREGPOLARITY 4
#define MODE_ALL_OUTPUT 0xFF
#define MODE_ALL_INPUT 0X00
#define PWM_BASE 0x00041000
#define F_50MHz 50000000
#define F_wanted 50
#define DUTY 5

void init()
{
    IOWR_32DIRECT(PWM_BASE, IREGDIR * 4, MODE_ALL_OUTPUT);                          // sets all the pins in mode Output
    IOWR_32DIRECT(PWM_BASE, IREGPOLARITY * 4, 1);                                   // sets the polarity to 1
    IOWR_32DIRECT(PWM_BASE, IREGPERIOD * 4, (int)(F_50MHz / F_wanted));             // sets the period of the PWM
    IOWR_32DIRECT(PWM_BASE, IREGDUTY * 4, (int)(F_50MHz / F_wanted * DUTY / 100));  // sets the duty cycle of the PWM

    volatile unsigned int k;
    volatile unsigned int a = 0;

    while(1){

        if (a > 5){
            a = 0;
        }

        IOWR_32DIRECT(PWM_BASE, IREGDUTY * 4, (int)(F_50MHz / F_wanted * (DUTY + a) / 100));  // varies the value of duty cycle
                                                                                              //to change position of the servo

        for(k=0;k<4000000;k++);

        a += 1;
     }

}
```

Figure 11 shows a measurement of a GPIO port after plugging the FPGA board and running the code. At this moment in the loop, we measure a duty cycle of 1 millisecond and a period of 20 milliseconds. By connecting the servomotor to one of the GPIO pins, we were able to rotate it.

Figure 11: ***PWM signal measured directly on the GPIO pins.***

4 Conclusion

To conclude, it was very instructive to correctly design a custom slave component and learn the basics of programming within the FPGA environment. We have been able to design a PWM signal generator component and test it in ModelSim test bench simulator before designing the hardware and uploading our program to the FPGA board. Finally, our design was able to produce simultaneously till 8 PWM signals up to 1 second period with custome duty cycle and polarity.