Motor Matrix Control

Roberto Augusto Philippi Martins

September 29, 2016

Abstract

A description and short tutorial on the Matrix Control System IP Core.

Aimed at managing multiple targets simultaneously, it is developed as a FPGA project using Altera's Mapped Memory Avalon interface or as a standalone module.

1 Introduction

The main objective of this module is to control multiple motors in parallel, each containing its own PWM signal generator. Commands must be received by the module and used to drive each motor individually.

As means to achieve that, this IP Core generates is able to process commands, store characteristic values and drive each motor.

2 Commands

All commands have a 32bit long data format.

Command (cmd): Bits 31-24 resolve the command function.
Row (row): Bits 23-16 are the target's row value.
Column (col): Bits 15-8 are the target's column value.
Intensity (val): Bits 7-0 indicate the command's value.

2.1 Set power

Changes the current power assigned to a target motor.
This command's 'cmd' code is 0x00.
The row and column values can range from 0 to 254, according to the amount of motors specified.
The intensity value ranges from 0 to 255.

It is possible to address multiple motors at once, by putting the value 255 at the row or col fields.
Setting row (or col) to 255 means addressing all rows (or all columns) in the same command.

2.2 Set shifting amount

Changes the current shifting amount assigned to a target motor.
This command gives the possibility of changing a motor's power gradually, giving the impression of a slow and steady change.

This command's 'cmd' code is 0x01.
The row and column values can range from 0 to 254.
Putting a intensity value of 0 means that the power change will be instant.

It is possible to address multiple motors at once, by putting the value 255 at the row or col fields.
Setting row (or col) to 255 means addressing all rows (or all columns) in the same command.

2.3 Set decay amount

Changes the current decay amount assigned to a target motor.
Decay means that the current power assigned to a motor will automatically fall down to zero. This decay amount only takes action after the motor is settled with its assigned power value.

This means that Set Power has priority over Decay.

The code for this command is 0x02.
The row and column values can range from 0 to 254.
Putting a decay value of 0 means that the power will NOT decay.
A decay value of 255 means that the motor will turn off instantly after reaching full power.

It is possible to address multiple motors at once, by putting the value 255 at the row or col fields.
Setting row (or col) to 255 means addressing all rows (or all columns) in the same command.

2.4 Power, Shifting and Decay example

Follows an example of application of these three commands.

0x 01 FF FF 03 // Set shifting amount to all rows and all columns
0x 02 FF FF 05 // Set Decay amount to all rows and all columns
0x 00 FF 03 5D // Set power to all motors

As consequence of the three previous commands, all motors will slowly rise to a power level of 5D (93_{10}) at a change rate of '3'. After reaching the desired level, all motors in column 03 will start to decay, with a decay rate of '5'.

2.5 Copy to next row

This command is used to copy the value of all the motors on a row to the next one.
It can be used to simulate a flowing motion of the motors.
The values of power, shifting amount and decay amount are all copied down from the specified motor to the motor immediately below.

The code for this command is 0x03.
The row value can range from 0 to 254. A value of 255 means that all rows will be copied down.
The values of 'col' and 'val' do not affect this command.

2.6 Copy to next col

This command is used to copy the value of all the motors on a column to the next one.
The values of power, shifting amount and decay amount are all copied from the specified motor to the one at its right.
The code for this command is 0x04.
The row value can range from 0 to 254. A value of 255 means that all rows will be copied down.
The values of 'col' and 'val' do not affect this command.

3 Parameters

3.1 Generic parameters

There are three generic parameters in this IP Core:

- rows : number of rows contained in the matrix. This value can range from 0 to 254.

- cols : number of columns contained in the matrix. This value can range from 0 to 254.

- time_constant : reference time for the shifting and decaying processes. This value determines how many clock cycles are between each shift/decay operation.
  Having $time\_constant = 50000000$ means that there is one value update every 50M clock cycles.

3.2 Internal parameters

Other important parameters in this module:

- The PWM driver works with a fixed frequency, doing one work cycle every 256 clock cycles. This parameter is set inside the timer_pwm component.

- Data input and output are 32bit wide each.

- The conduit output of the module is a vector with $rows * cols$ bits.

4 Interrupt and acknowledge

This IP Core follows the Altera Avalon Interface protocol. It is possible, although not required, to perform a handshake between each data transfer.

This handshake operation works as follow:

1. The module receives a command, executes it and raises an interrupt flag.

2. The module will, then, wait for a acknowledge signal before doing anything else.

3. If there is acknowledgment, the module will resume listening for commands.

5 Avalon Interface

Follows a tutorial on how to make this IP Core compatible with the Altera Avalon Interface protocol.

This module already uses a compatible set of signals, so most of the work is already done.

For this tutorial we will not be using the interrupt/acknowledge handshake method.

5.1 Creating a interface component

It is easier to first create a top level component that only calls the Matrix Control System. This top level component will work as a mask between Qsys and our component.

```vhdl
library ieee;
 use ieee.std_logic_1164.all;

 entity matriz_avalon_interface is
        generic (
                rows : positive := 4;
                cols : positive := 4;
          time_constant : integer := 512

        );
        port (
                -- clock and reset
                csi_csink_clock : in std_logic;
                rsi_rsink_resetn : in std_logic;

                -- avalon slave
                avs_aslave_read : in std_logic;
                avs_aslave_write : in std_logic;
                avs_aslave_address : in std_logic;
                avs_aslave_writedata : in std_logic_vector(data_width-1 downto 0);
                avs_aslave_readdata : out std_logic_vector(data_width-1 downto 0);

                -- output signals
                coe_pwms_export : out std_logic_vector((rows*cols)-1 downto 0)
        );
 end entity matriz_avalon_interface;
```

Note that all signals instantiated here have a counterpart in our project.

Some signals, such as test, bistIn, bistOut, acknowledge and interrupt are not being used here.

The signal 'readwrite' has been divided into two separated signals.

In the following page we show the instantiation of the Matrix Control System itself, as well as some modifications needed to make this interface work.

Here is the Matrix Control System instantiation. These attributions are self-explanatory, but we should pay attention to the Avalon naming standard.

```vhdl
-- matriz instantiation
matriz_inst : MotorMatrixControl
        generic map (
    rows => rows,
    cols => rows,
    dataWidth => 32,
    time_constant => time_constant

    )
    port map (
    clock => csi_csink_clock,
    reset => not(rsi_rsink_resetn),
    enable => enable(2),
    acknowledge => acknowledge,
    test => test,
    bistIn => bistIn,
    readwrite => readwrite,
    writedata => avs_aslave_writedata,
    readdata => avs_aslave_readdata,
    interrupt => interrupt,
    bistOut => bistOut,
    motors_out => coe_pwms_export
        );
```

As follows, we are forcing the unused signals to a known value.
Notice that our 'readwrite' signal is the sum of two Avalon standard signals, 'avs_aslave_read' and 'avs_aslave_write'.

The enableProcess is a workaround for creating a enable signal for our module. It may seem weird, but it's just the write signal delayed by two clock cycles.

```vhdl
acknowledge <= '1';
bistIn <= '0';
test <= '0';
readwrite <= avs_aslave_read & avs_aslave_write;

enableProcess:
process(csi_csink_clock,avs_aslave_write)
begin
        if rising_edge(csi_csink_clock) then
                enable <= enable(1 downto 0) & avs_aslave_write;
        end if;
end process;
```

6   Simulating the Matrix Control System IP Core

It is possible to simulate the module with ModelSim, or most other VHDL simulation tool.

In here, we se a sequence of three commands and how they behave.

At time = 0, we put full power into the motor[0,1].
Right after, we put full power into motor[2,1].
Shortly after, we put decay=5 into all motors.
Close to the end, we apply power to motor[2,3]. Notice that this motor already
has a internal decay of 5, and starts to fall down automatically.
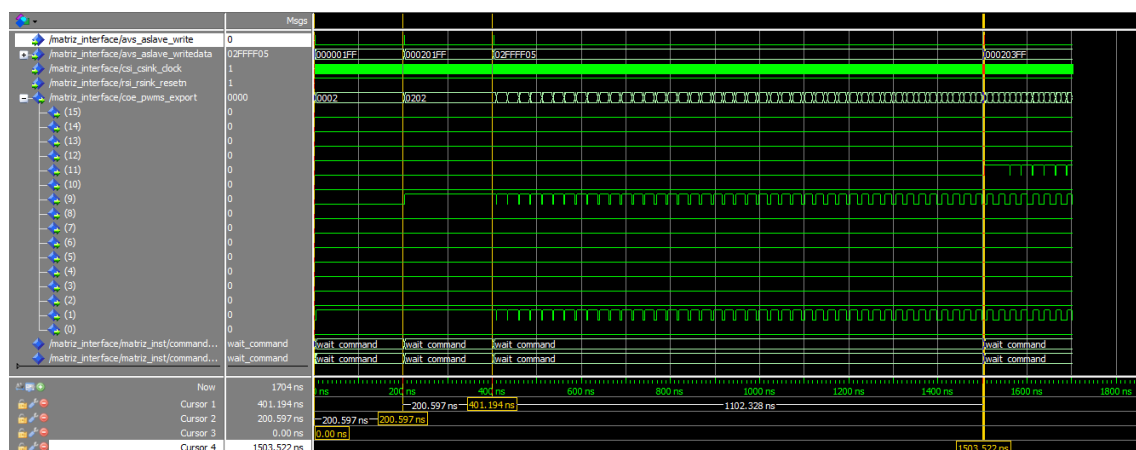
Here we have a close up of two commands.
As the control block is defined by a Finite State Machine, we also show its actual
and next state value.
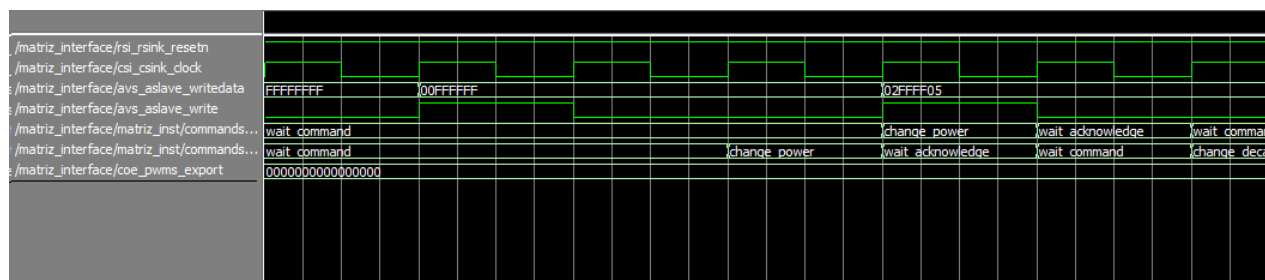
7 Creating the Qsys IP Core

7.1 Avalon Memory Mapped Slave

"To give a command to an I/O device, the processor must be able to address the device and to supply one or more commands words. Two methods are used to address the device: memory-mapped I/O and special I/O instructions. In **memory-mapped I/O,** portions of the address space are assigned to the I/O devices. Reads and writes to those addresses are interpreted as commands to the I/O device"

From Hennessy, John L. **Computer and organization Design. The hardware / Software Interface.**

We will use this method of addressing to control our IP Core.

7.2 Starting up the process

First, open the Qsys system integration tool, available with the Quartus II software.
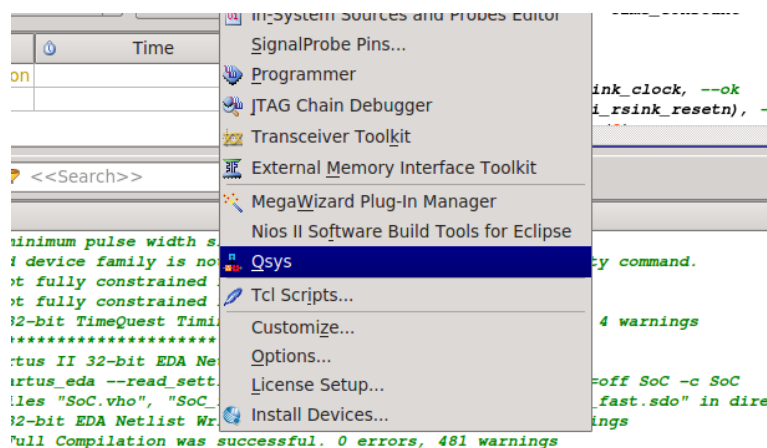
Now click on "New component", or in File - New Component.
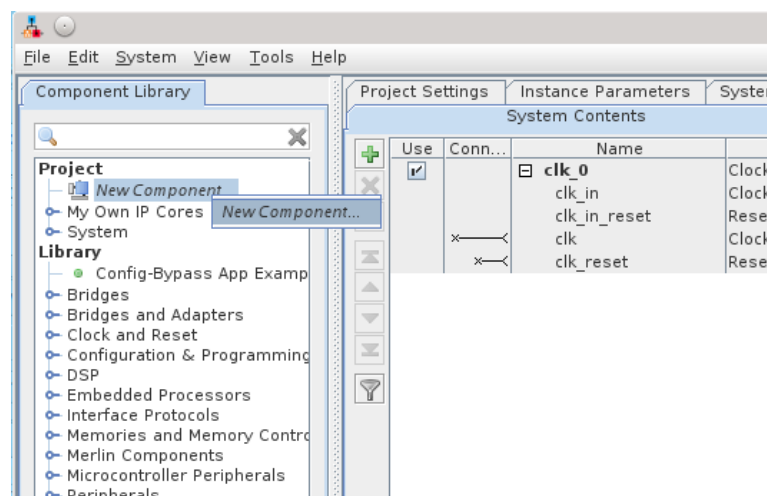
In the "Component Type" area, put the IP Core name and its display name. Add other information, such as description and documentation at will.

7.3   Files

In the "Files" area, select the previously created interface component.
Now click on "Analyze Synthesis Files", and "Next".

7.4   Parameters

In the "Parameters" area, you can check the generic parameters of the IP Core. You can also make some of them editable or not. In this case, we want to make a variable amount of rows and columns, as well as making the time constant adjustable, hence we leave them all as editable.

7.5 Signals

In the "Signals" area, we assign each of our signal to a corresponding Altera
Avalon Interface signal type. Here is where our previous naming practice makes
a difference.

Notice that each signal is assigned to a interface type and signal type.
The "aslave" name is a identification for this group we created. We have a
"Avalon Memory Mapped Slave" interface, with the name "aslave". The name
can be changed in the "Interfaces" tab.

Name	Interface
avs_aslave_readdata	aslave
avs_aslave_writedata	aslave
avs_aslave_address	aslave
avs_aslave_write	reset_sink
avs_aslave_read	new AHB Master...
csi_csink_clock	new AHB Slave...
coe_pwms_export	new APB Master...
rsi_rsink_resetn	new APB Slave...
	new Avalon Memory Mapped Master...
	new Avalon Memory Mapped Slave...
	new Avalon Streaming Source...
	new Avalon Streaming Sink...
	new Avalon Memory Mapped Tristate Slave...
	new AXI Master...
	new AXI Slave...
	new AXI4 Master...
	new AXI4 Slave

| Component Type | Files | Parameters | Signals | Interfaces |

▶ About Signals

Name	Interface	Signal Type
csi_csink_clock	clock_sink	clk
rsi_rsink_resetn	reset	reset_n
avs_aslave_read	aslave	read
avs_aslave_write	aslave	write
avs_aslave_address	aslave	address
avs_aslave_writedata	aslave	writedata
avs_aslave_readdata	aslave	readdata
coe_pwms_export	pwms	export

7.6 Interface

Here we will edit each interface created.

For example, we assigned avs_aslave_write to a "Avalon Memory Mapped Slave Interface", with the function of the "write" signal. In this section we will make sure everything is correctly assigned.
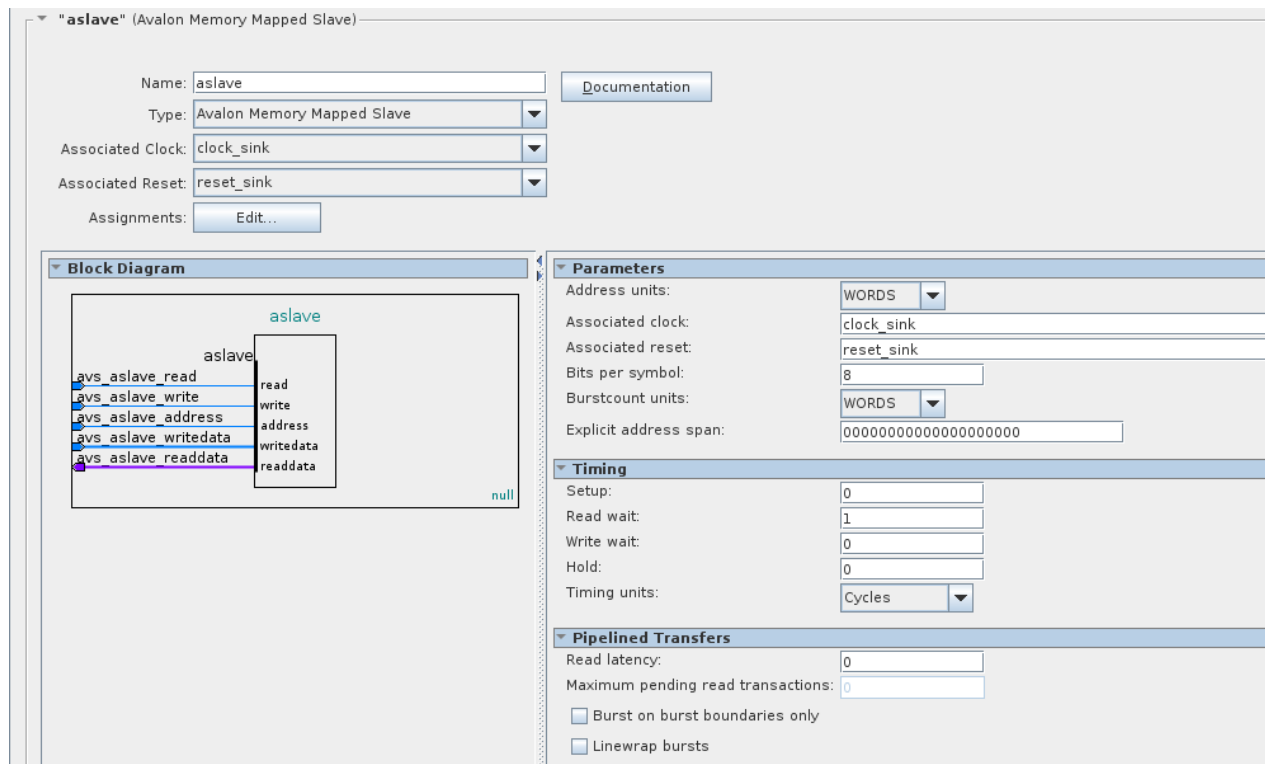
These are the corresponding waveforms for the aslave interface. Our memory mapped interface has to follow these timing specifications.
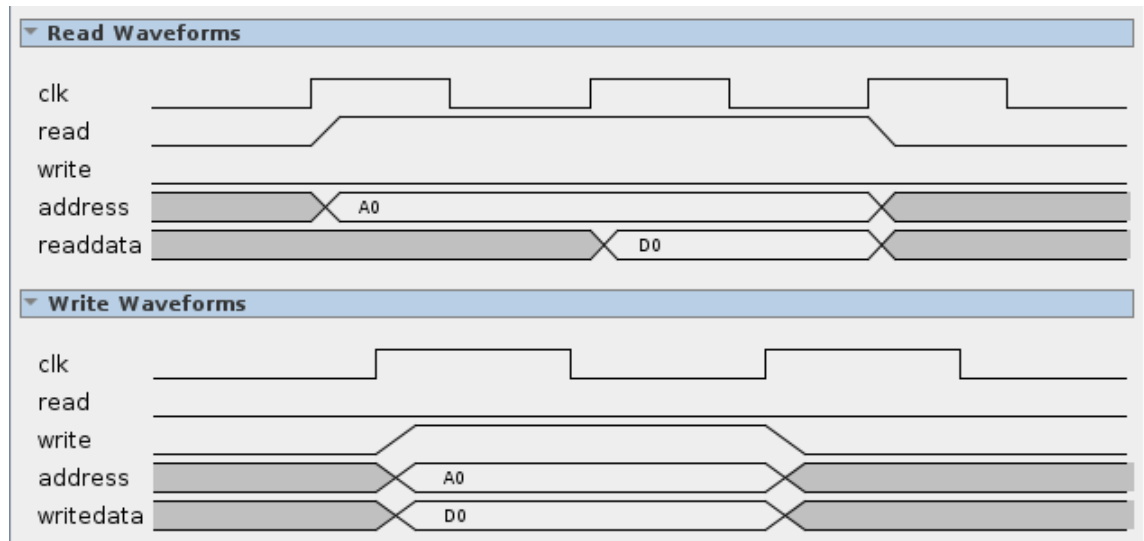
The "pwms" interface.
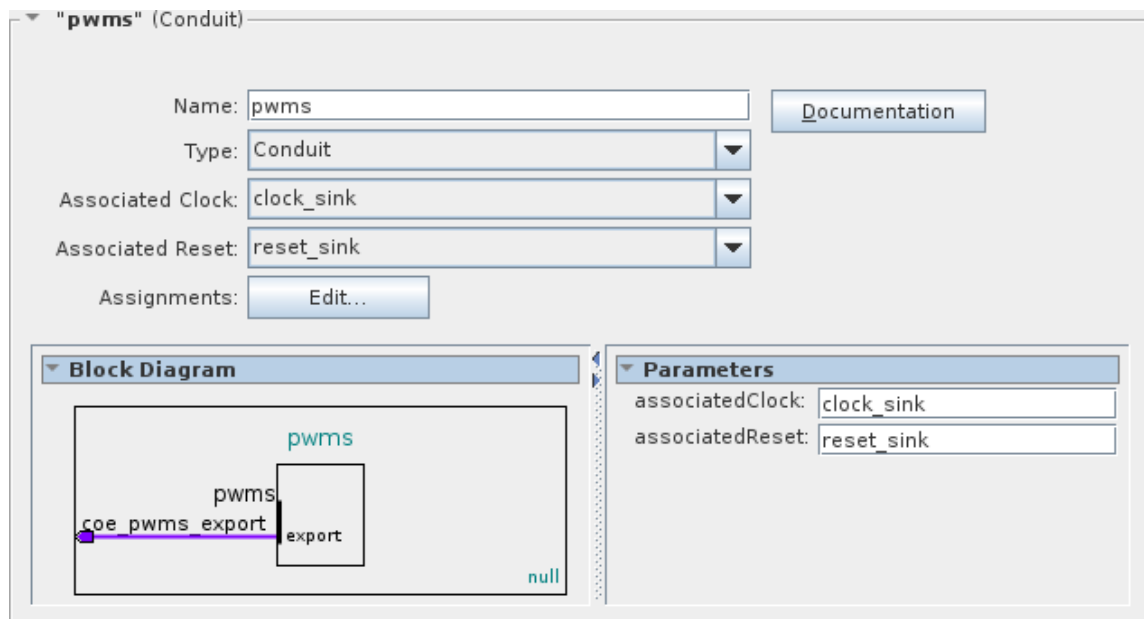
The "clock" interface.
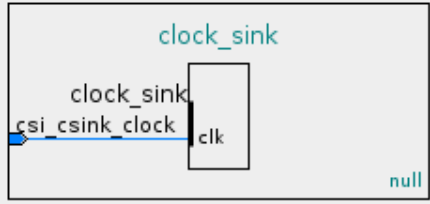
The "reset" interface
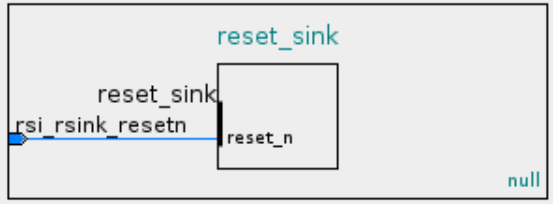
8 Using the IP core

The now created IP Core can be used as any other.

We will add the Matrix Control System IP Core to a Qsys System that already contains a On-chip memory, a Nios II processor and a JTAG UART serial interface.

1. Add the new component to your Qsys System

2. Connect the clock and reset to the corresponding global signals

3. Connect the "aslave" interface to the Nios 2 data and instruction masters.

4. Export the "pwms" interface to a external signal (double-click on the export column)

5. Assign a memory address to your component.
   In this case, the memory address is between 2068 and 206f.
   This gives us 8 bytes (32bits for reading and 32bits for writing).
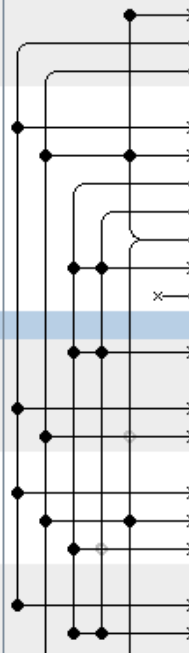
Use	Connections	Name	Description
✔		⊟ **clk**	Clock Source
		clk_in	Clock Input
		clk_in_reset	Reset Input
		clk	Clock Output
		clk_reset	Reset Output
✔		⊟ **nios2**	Nios II Processor
		clk	Clock Input
		reset_n	Reset Input
		data_master	Avalon Memory Mapped Master
		instruction_master	Avalon Memory Mapped Master
		jtag_debug_modul…	Reset Output
		jtag_debug_module	Avalon Memory Mapped Slave
		custom_instructio…	Custom Instruction Master
✔		⊟ **matriz**	matriz
		aslave	Avalon Memory Mapped Slave
		pwms	Conduit
		clock_sink	Clock Input
		reset_sink	Reset Input

Double click on "Export" to assign the "pwms" and "clk_in" signals to external signals.

Double click on the "Base" field to assign a base address to the module. You can also use the "System - Assign Base Addresses" option.

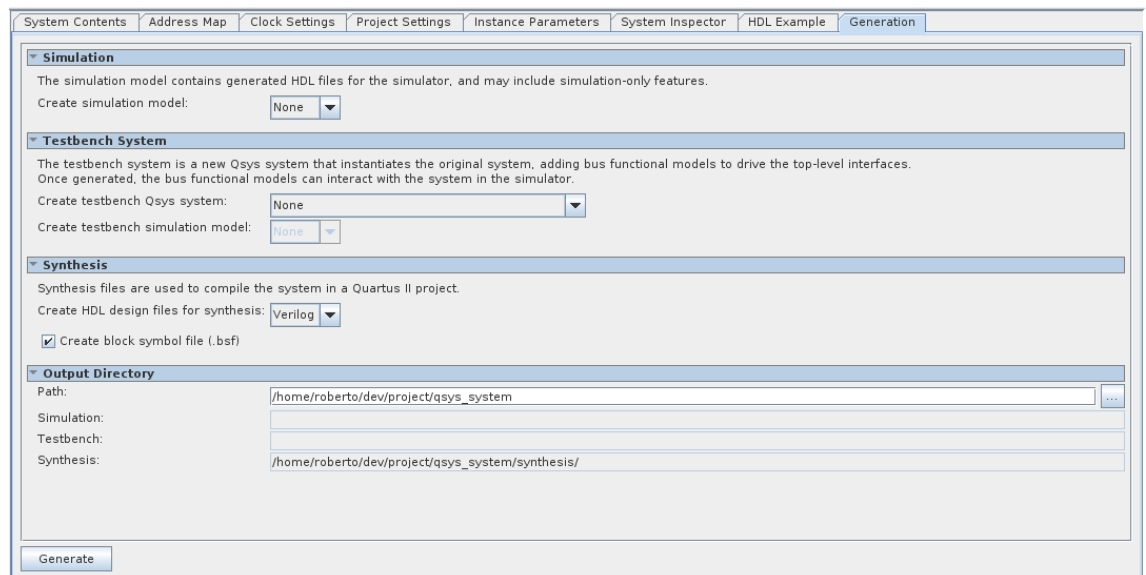Our project should now look like this:

With everything done, the only input signal in our module is the clock, and the only output is the pwms export signal.
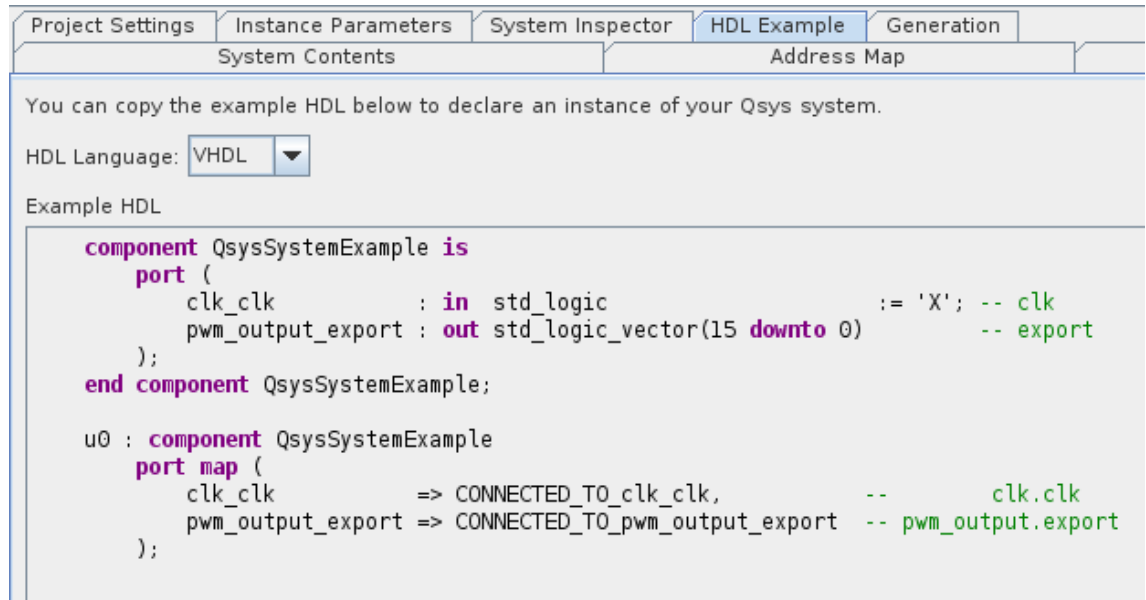
Now we can go to the "Generation" area, select the wanted options and generate the Qsys System.

9 Using the Qsys System

If the generation process has been successful, we now have a ready to use QsysSystemName.qsys file in our project folder.

Add this file to the project and create a top level VHDL file, on which we will call the Qsys System produced.

The "HDL Example" provided by the Qsys Generation Tool is a good starting point for using our Qsys System.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity SystemExample is
        port(
                clock_50 : in std_logic;
                ledr : out std_logic_vector(15 downto 0)
        );
end entity SystemExample;



architecture SystemExample_arch of SystemExample is

        signal pwm_output : std_logic_vector(15 downto 0);

    component QsysSystemExample is
        port (
            clk_clk             : in  std_logic           := 'X';
            pwm_output_export : out std_logic_vector(15 downto 0)
        );
    end component QsysSystemExample;


begin

        systemPortMap: component QsysSystemExample
        port map (
            clk_clk             => clock_50,
            pwm_output_export => pwm_output
        );

        ledr(15 downto 0) <= pwm_output(15 downto 0);

end architecture SystemExample_arch;
```

This file assigns clock_50 as our system clock, and assigns the output of our system to the signal pwm_output.
The pwm_output signal is directed to the board's ledr pins.

10 Controlling the module with a Nios II application

Now one can control the create IP Core as if it were just another memory mapped slave.

It's possible to send a command to the component by writing on the lower 4bytes of the assigned address, and read status from the upper 4bytes.

Follows a general guideline of how to create a Nios II code that can do this:

```c
//We define the assigned memory address
#define MATRIX_ADDRESS 0x00002068

//We create two pointers:
//one for writing and one for reading data
data = (unsigned int *) MATRIX_ADDRESS;
status = (unsigned int *) (MATRIX_ADDRESS + 4);

//We can create a function that assigns a value to
//the memory address
void write(unsigned int valor) {
        *data = valor;
}


//These three function calls should do the following:
//Put a shifting amount of 3 on every motor
//Put a decay amount of 15 in every motor
//Assign a power value of 255 to every motor

write((1<<24)|(255<<16)|(255<<8)|(03));
write((2<<24)|(255<<16)|(255<<8)|(15));
write((0<<24)|(255<<16)|(255<<8)|(255));

//The result is that all motors should rise to full power,
//and then decay back to zero.
```

References