# University of Salerno



Department of *Ingegneria dell’Informazione ed Elettrica e Matematica Applicata*

Masters Degree in Computer Engineering

Final Report for System on Chip

Academic Year 2023/2024

Partecipating Students:

Battipaglia Valerio [0622702091]

Caso Antonio [0622702068]

De Stefano Sabatino [0622702086]

Dell’Orto Giuseppe Maria [0622701935]

High Speed Quadrature Encoder

**Target**: Design a hardware platform that contains the Zynq PS, one IP, and two AXI connections for both the **direction** and the **tick\_count** outputs of the IP. The platform is used to implement a High Speed Quadrature Encoder, divided into a counting section implemented on the PL and an elaborating section, used to calculate **RPMs** and **acceleration**, for the software application. In addition, both the hardware and software sides must come together in order to send final (post-processing) information to the PmodOLEDrgb display.

**Board to be used** : Zybo-Z720

**IDE to be used**: Vivado ver. 2021.2 from Xilinx

Vitis ver. 2021.2 from Xilinx

# Specifications

Create a platform that includes both the Zynq and one IP. It will need two AXI connections between the PL and the PS portion of the system. The SoC implements a High Speed Quadrature Encoder using both the Programmable Logic and the Processing System of the Zybo board.

The High Speed (this implies that the quadrature encoder must sample at an high frequency according to Nyquist, well above the maximum frequency of the motor encoder) Quadrature Encoder must be designed to have a **sensitivity** of **1 RPM**, which can be accomplished choosing to provide the output by bringing together the ticks counted in a span of **50ms**. Nonetheless, the output must be display on a **PmodOLEDrgb** display with a refresh rate of 5ms. To clarify, the precise calculation of the 5ms must be achieved by means of a **timer-related interrupt**.

# Design

As previously mentioned, the subdivision is as follows:

- **Programmable Logic**: It works like a counting timer implemented in Verilog HDL, outputting only the count of edges on both channels A and B, and also providing the direction based on the phase shift between the two square waves.

- **Processing System:** integrated into the block design, contains the software application, which is responsible for collecting the edge count data. It contains both the data processing routines and the instructions for sending this data to the display. Additionally, it includes the code to be executed, inside the callback function, upon firing the interrupt, which is used to implement a fixed temporal rate of 5ms.HW and SW will be connected through the AXI bus. Since there is no need for high bandwidth data transfer, we decide to use the GP port of the Zynq. Using the GP port it is known that the AXI master is yet implemented on the PS while the PL portion needs to implement the AXI slave. For this simple piece of HW without tight design constraint, we decide to use a predefined schematic block for the AXI slave.

**Design phases**:

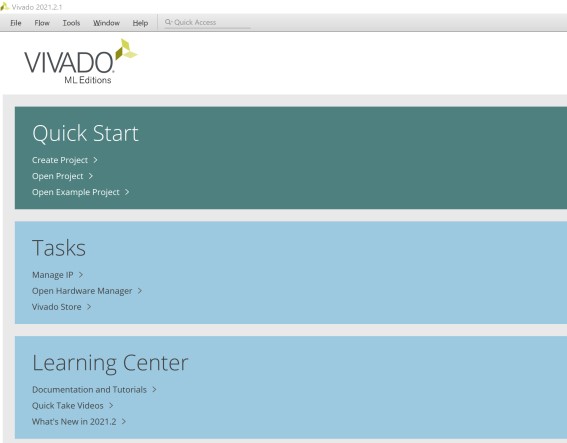
1. Uses Vivado. Design the HW and simulate it.
2. Uses Vivado. Design the schematic architecture of the SoC 3) Uses Vitis. Design the SW application.

# Using the Vivado IDE

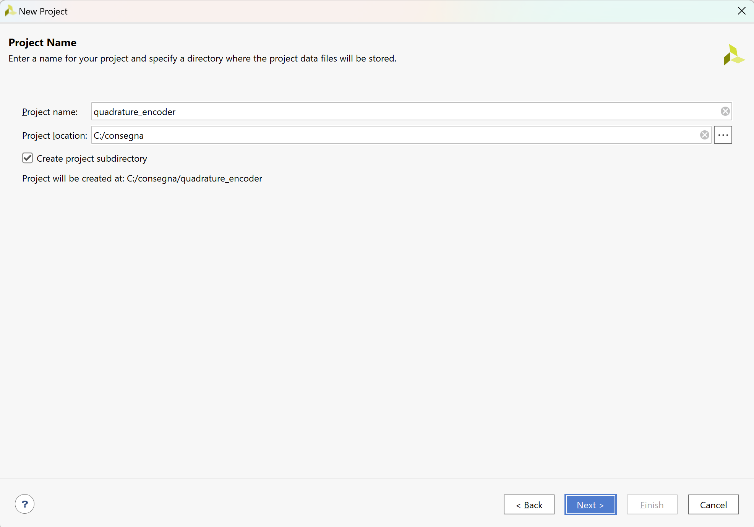
## Setting the environment

Launch Vivado 2021.2

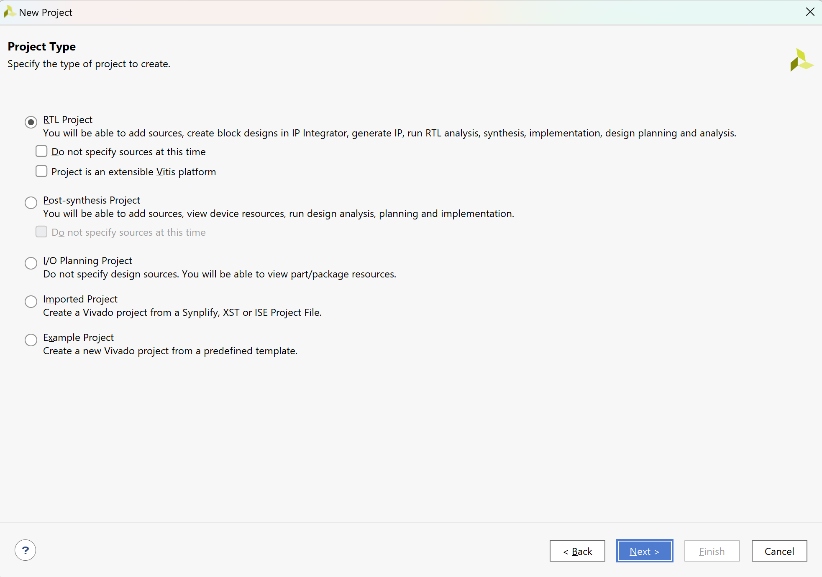
Select ‘Create Project’. Click Next.



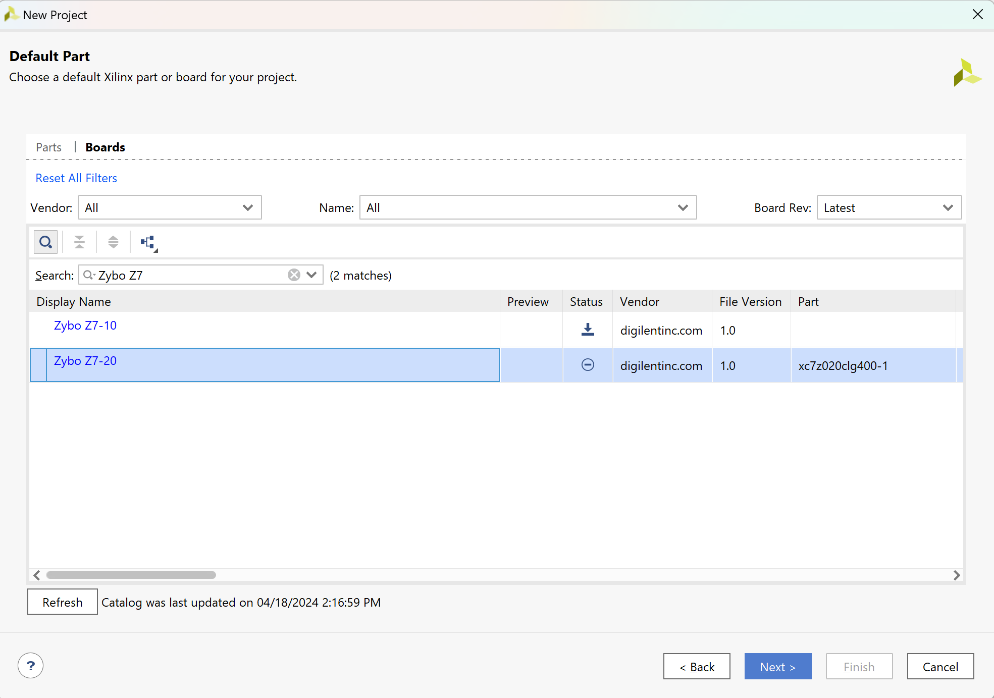
Indicate a name (e.g. quadrature\_encoder) and a path (allow to create a subdirectory to avoid cluttering the workspace). Click Next.



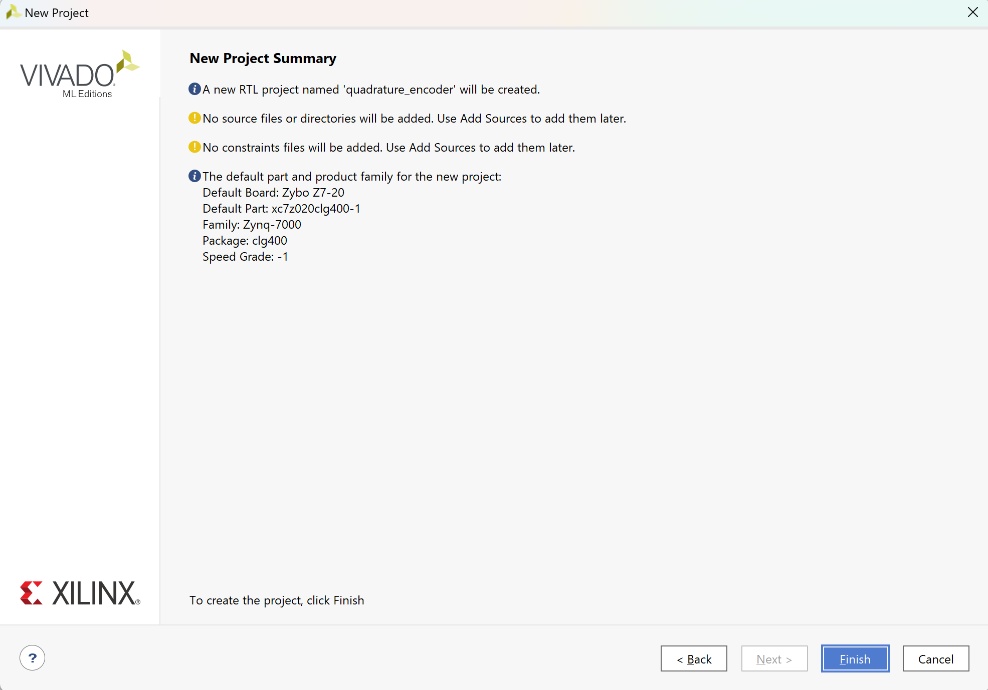
Chose an RTL project. Click Next.



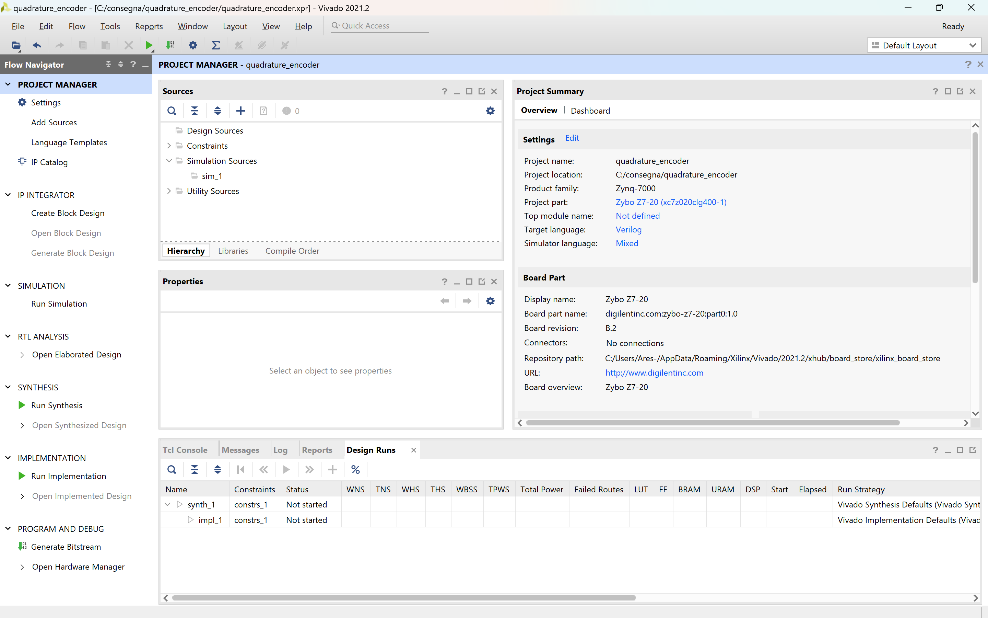
Click Next more than once to reach the Default part window. Click on ‘Boards’ and select Zybo Z7-20 (if this is the first time you should initially download the board definitions). Click Next.



Click Finish. The IDE is now open and ready to work with.



Below is the main window containing all the necessary tools of the IDE.



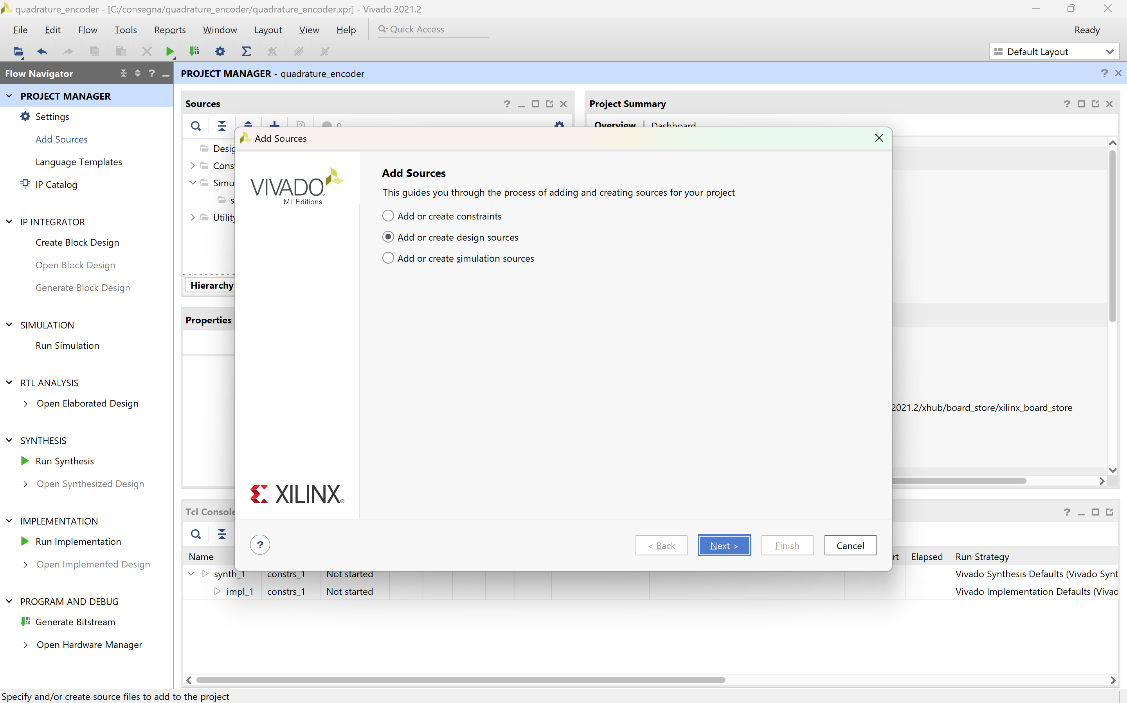
The IDE shows on the left a vertical window with the design flow steps that guides the engineer through the needed steps to complete the design task. The Sources panel is very important and will show the components of the design (design sources in HDL, constraints, testbenches, block schematic, Integrated Logic Analyzer). The windows on the bottom shows flow messages and is important to check for warnings and errors. The large window on the right is where the actual designs take place. Be aware that many design steps can take up to ten minutes of processing time. The tiny label on the top right, that is stating ‘Ready’, is what tells you that the required design process is completed. Always wait for the ready status to activate a new step of the design. The properties window provides info regarding the selected object and can be used to change the properties.

The layout can be changed at will and the windows can undock to float and be enlarged to cover the entire screen. It is up to you to decide your preferred layout and, possibly, save it.

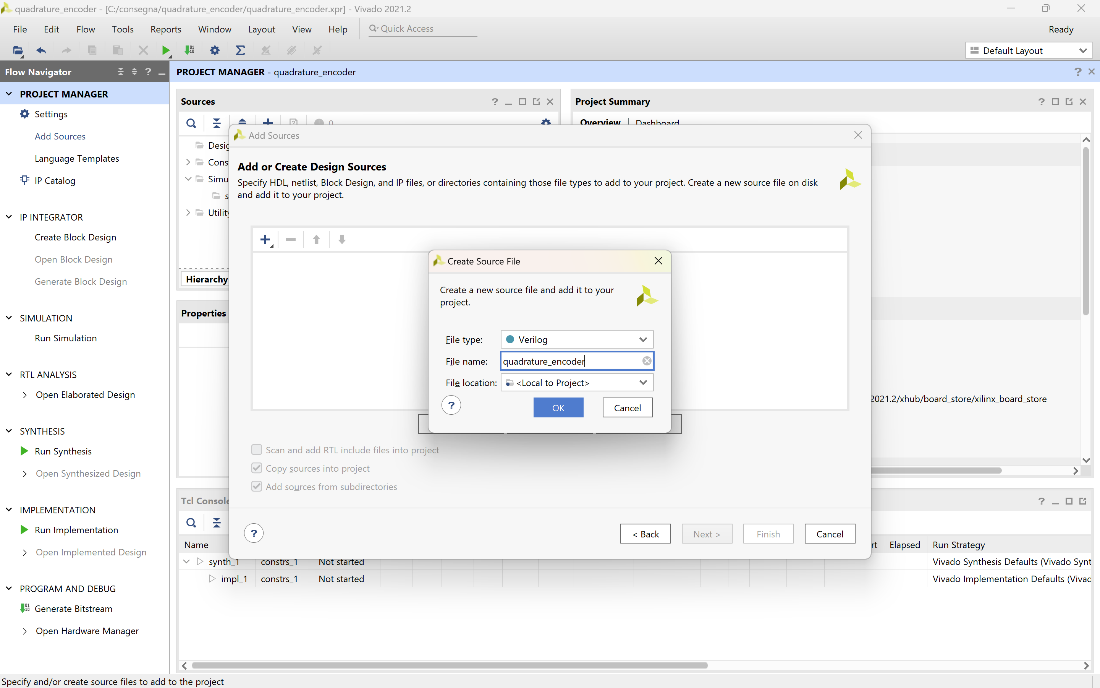
## Designing the quadrature encoder and simulate the HW

The design of the HW does not require sophisticated steps. Only describe the circuit in HDL Verilog and define the testbench in the same language.

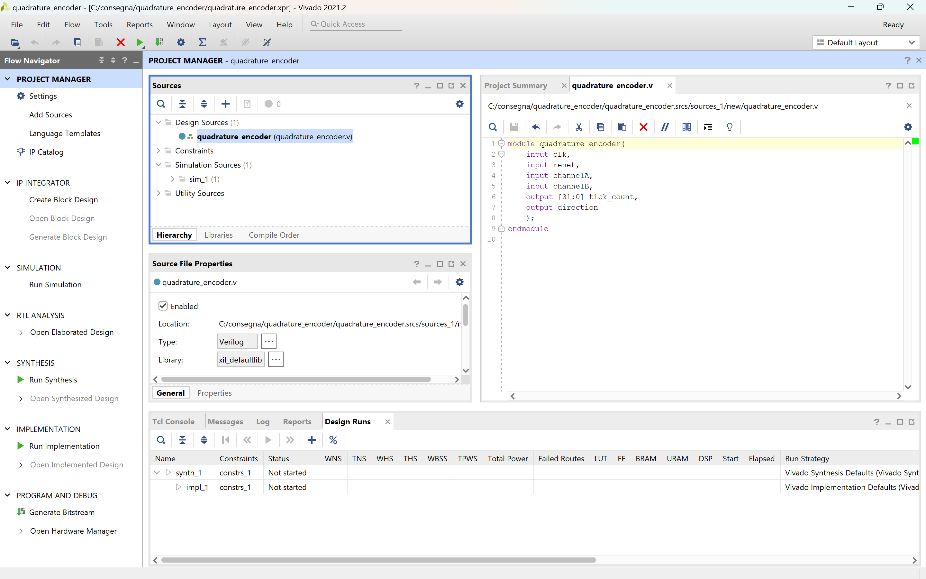
Click ‘Add sources’ in the Project Flow pane. Select ‘Add or create design sources’.



Click Next. Select ‘Create File’ and name the Verilog file quadrature\_module\_pl. Click ‘Ok’ then ‘Finish’.



Click ‘Ok’ and ‘Yes’ to skip the module definition window. The HDL file is now hidden in the Project Manager pane under Design Sources. Expand it and double click on the file to open it.



Describe the counter timer in HDL. Below the code:

module quadrature\_encoder(

    input wire clk,           // clock signal

    input wire reset,         // reset signal

    input wire channelA,      // quadrature encoder channel A

    input wire channelB,      // quadrature encoder channel B

    output reg direction,      // direction of rotation: 0 - clockwise, 1 - counter-clockwise

    output reg [31:0]tick\_count //ever increasing number of ticks

);

reg [2:0] channelA\_delayed, channelB\_delayed;

    always @(posedge clk or negedge reset) begin

        if (reset == 1’b0) begin

              tick\_count <= 32'd0;

              direction <= 1'b0;

              channelA\_delayed <= 3'b000;

              channelB\_delayed <= 3'b000;

        end else begin

              if (channelA\_delayed[1] ^ channelA\_delayed[2]

^channelB\_delayed[1] ^ channelB\_delayed[2])

begin

                  if(channelA\_delayed[1] ^ channelB\_delayed[2]) direction <= 1'b1;

                  else direction <= 1'b0;

                  tick\_count <= tick\_count + 1;

              end

        channelA\_delayed <= {channelA\_delayed[1:0], channelA};

        channelB\_delayed <= {channelB\_delayed[1:0], channelB};

        end

  end

endmodule

Let us analyze the HDL, in order to have a clear picture of how it manages to count each tick efficiently and precisely.

module quadrature\_encoder(

    input wire clk,           // clock signal

    input wire reset,         // reset signal

    input wire channelA,       // quadrature encoder channel A

    input wire channelB,       // quadrature encoder channel B

    output wire direction,       // direction of rotation: 0 – counter-clockwise, 1 - clockwise

    output reg [31:0]tick\_count //ever increasing number of ticks

);

In this first segment, we have the interface of the quadrature encoder. It takes in a clock signal (clk), a reset signal (reset), and two signals from the encoder (channelA and channelB). It outputs the direction of rotation (direction) and a 32-bit count of the ticks detected (tick\_count). This module helps in determining how far and in which direction the encoder has moved. The tick\_count output has been chosen as a bus of 32 bits in order to have plenty of counting space before having an overflow occur.

Each edge, whether rising or falling, translates into a tick count, regardless of the motor's rotation direction.

reg [2:0] channelA\_delayed, channelB\_delayed;

The two “delayed” signals are used in both for the direction and the tick count, and are defined as “reg” given that they are assigned inside the procedural block.

    always @(posedge clk or negedge reset) begin

        if (reset==1’b0) begin

              tick\_count <= 32'd0;

              direction <= 1'b0;

              channelA\_delayed <= 3'b000;

              channelB\_delayed <= 3'b000;

        end else begin

if (channelA\_delayed[1] ^ channelA\_delayed[2]

^channelB\_delayed[1] ^ channelB\_delayed[2])

begin

                  if(channelA\_delayed[1] ^ channelB\_delayed[2]) direction <= 1'b1;

                  else direction <= 1'b0;

                  tick\_count <= tick\_count + 1;

              end

        channelA\_delayed <= {channelA\_delayed[1:0], channelA};

        channelB\_delayed <= {channelB\_delayed[1:0], channelB};

        end

  end

endmodule

Subsequently, the procedural block begins, being activated with each negative edge of the reset and positive edge of the clock signal. Leaving aside the easily understandable reset logic, the core of the Verilog HDL works as follows:

When the reset signal is low (reset == 1'b0), the system initializes by setting tick\_count to 0, direction to 0, and both channelA\_delayed and channelB\_delayed to 3'b000. This ensures the system starts from a known state.

The edge detection is done by checking specific transitions in the delayed signals of channelA and channelB. Specifically, the condition “if (channelA\_delayed[1] ^ channelA\_delayed[2] ^ channelB\_delayed[1] ^ channelB\_delayed[2])” looks for changes in the state of the signals. The XOR operation (^) checks for differences between consecutive bits in the delay registers.

The three flip-flops (implemented as 3-bit shift registers), for each channel register, are employed to mitigate phase discrepancies between the clock signal and the encoder signals. Their primary purpose is to synchronize the signals from the quadrature encoder channels with the system clock, ensuring accurate and reliable detection of changes in the encoder state. This process is actuated by the following lines (past two values plus the current one):

channelA\_delayed <= {channelA\_delayed[1:0], channelA};

channelB\_delayed <= {channelB\_delayed[1:0], channelB};

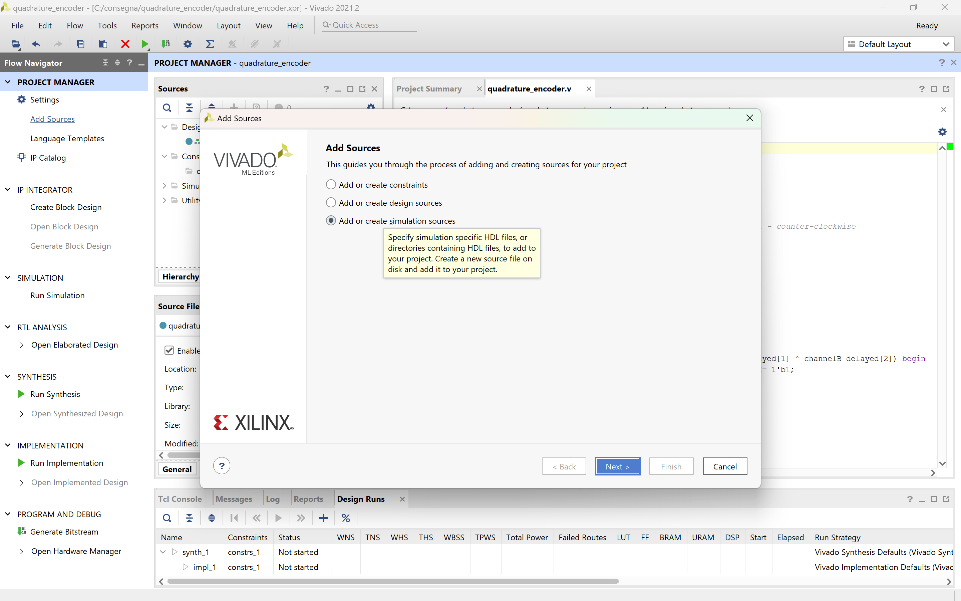
The encoder signals (channelA and channelB) are asynchronous with respect to the system clock (clk). Directly sampling these asynchronous signals can lead to metastability issues and unreliable signal capture. The use of three flip-flops in a shift register configuration delays the encoder signals by three clock cycles, effectively synchronizing them with the clock signal. This ensures that any transitions in the encoder signals are reliably captured on the rising edge of the clock, minimizing the risk of metastability and timing errors.

Regarding the direction, the condition checks the relative phase of channelA and channelB. If channelA\_delayed[1] is different from channelB\_delayed[2], the direction is set to 1 (clockwise), otherwise, it is set to 0 (counter-clockwise). This is due to the fact that, when in clockwise movement, channelA will have edge that come before edges on channelB. This means that channelA\_delayed[1] will be 1, whereas channelB\_delayed[2] will be 0, thus giving 1 as a result.

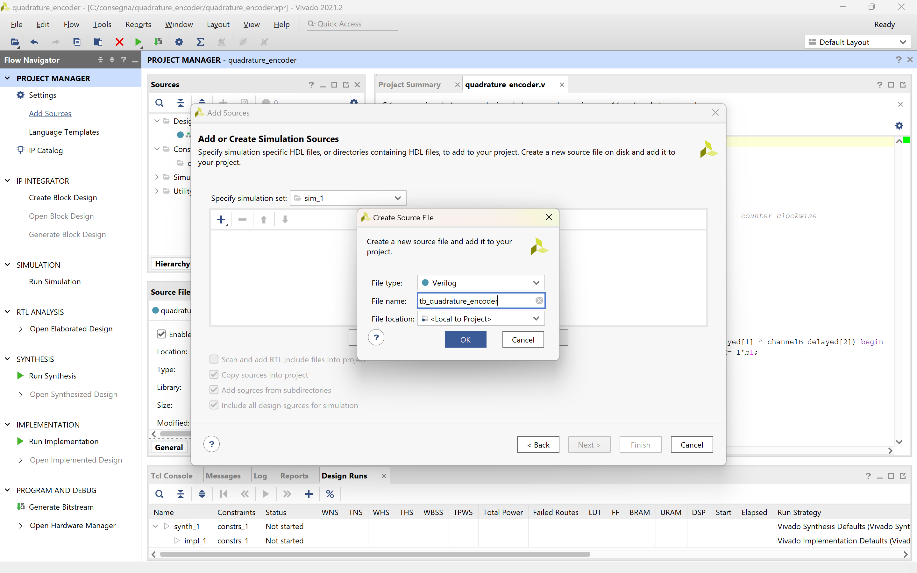
A syntax check is carried out every time one HDL file is saved. This is true for both the circuit description and the testbench. Syntax error are highlighted in the Sources pane and a short description is given in the Messages pane on the bottom. Not all the syntax errors are caught at this stage. Some of them are only available when simulation and/or design are carried out.

## Define the testbench

Click ‘Add sources’ in the Project Flow pane. Select ‘Add or create simulation sources’. Click Next.

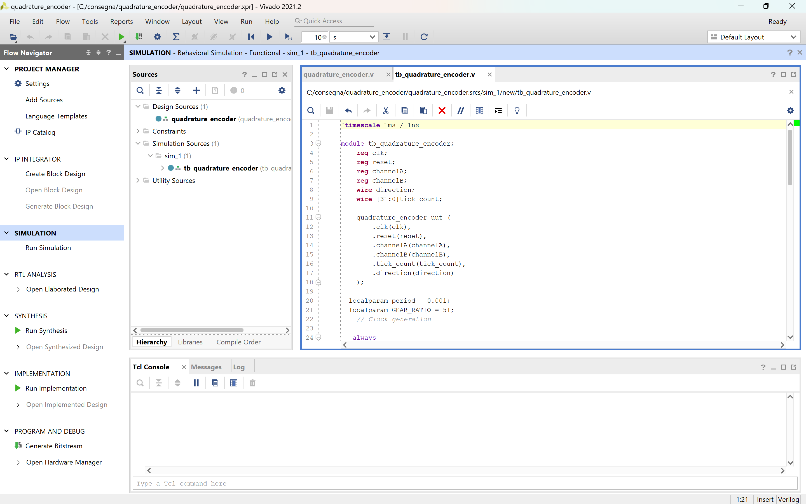


Select ‘Create File’ and name the Verilog file tb\_quadrature\_encoder. Click ‘Ok’ then ‘Finish’.



Click ‘Ok’ and ‘Yes’ to skip the module definition window.

Your file is now hidden in the Project Manager pane under Simulation Sources. Expand it and double click on the file to open it.



Describe the testbench in HDL. Below the code:

`timescale 1ms / 1ns

module tb\_quadrature\_encoder;

    reg clk;

    reg reset;

    reg channelA;

    reg channelB;

    wire direction;

    wire [31:0]tick\_count;

    quadrature\_encoder uut (

        .clk(clk),

        .reset(reset),

        .channelA(channelA),

        .channelB(channelB),

        .tick\_count(tick\_count),

        .direction(direction)

    );

  localparam period = 0.001;

localparam GEAR\_RATIO = 51;

    // Clock generation

   always

   begin

    clk = 1'b1; #0.0005;

    clk = 1'b0; #0.0005;

   end

    // Test sequence

    initial begin

        // Initialize signals

        reset = 1;

        channelA = 0;

        channelB = 0;

        // Release reset

        #1000 reset = 0;

        repeat ((0.5/(1.25/(1\*51)))) begin

            channelA = ~channelA; #((1.25/(1\*GEAR\_RATIO))\*1000); // 1 RPM

            channelB = ~channelB; #((1.25/(1\*GEAR\_RATIO))\*1000); // 1 RPM

        end

        repeat ((0.5/(1.25/(2\*51)))) begin

            channelA = ~channelA; #((1.25/(2\*GEAR\_RATIO))\*1000); // 2 RPM

            channelB = ~channelB; #((1.25/(2\*GEAR\_RATIO))\*1000); // 2 RPM

        end

        repeat ((0.5/(1.25/(3\*51)))) begin

            channelA = ~channelA; #((1.25/(3\*GEAR\_RATIO))\*1000); // 3 RPM

            channelB = ~channelB; #((1.25/(3\*GEAR\_RATIO))\*1000); // 3 RPM

        end

        repeat ((0.5/(1.25/(10\*51)))) begin

            channelB = ~channelB; #((1.25/(10\*GEAR\_RATIO))\*1000); // -10 RPM

            channelA = ~channelA; #((1.25/(10\*GEAR\_RATIO))\*1000); // -10 RPM

        end

        repeat ((0.5/(1.25/(123\*51)))) begin

            channelB = ~channelB; #((1.25/(123\*GEAR\_RATIO))\*1000); // -123 RPM

            channelA = ~channelA; #((1.25/(123\*GEAR\_RATIO))\*1000); // -123 RPM

        end

        repeat ((0.5/(1.25/(89\*51)))) begin

            channelB = ~channelB; #((1.25/(89\*GEAR\_RATIO))\*1000); // -89 RPM

            channelA = ~channelA; #((1.25/(89\*GEAR\_RATIO))\*1000); // -89 RPM

        end

        repeat ((0.5/(1.25/(186\*GEAR\_RATIO)))) begin

            channelA = ~channelA; #((1.25/(186\*GEAR\_RATIO))\*1000); // 186 RPM

            channelB = ~channelB; #((1.25/(186\*GEAR\_RATIO))\*1000); // 186 RPM

        end

        $display("lol");

        // Stop simulation

        $stop;

    end

endmodule

In the testbench, aside from instantiating the quadrature\_encoder module defined previously, we connect its interface with the testbench wires using the appropriate syntax.

We then create the clock signal with a frequency of 1 MHz: despite this frequency being more than enough to achieve a clear-enough sampling, we still experimented with other frequencies (notably, 50 MHz) to make sure that the module would work as intended. Following that, we first force reset on the module and then begin modifying the channels, effectively testing the written code.

An explanation regard the timing used in the testbench has to be made, particularly regarding the value each repeat construct receives.

For the motor under examination, its specifications (in relation to the encoder) are:

* 2 channels
* 12 PPR
* 48 CPR
* 1:51 GEAR\_RATIO

If we make 48 counts every 60 seconds, it means that 60/48 seconds elapse between each count, which is 1.25 seconds. If we make 48\*GEAR\_RATIO\*RPM counts every 60 seconds, it means that 60/(48\*GEAR\_RATIO\*RPM) seconds elapse between each count, which is 1.25/(GEAR\_RATIO\*RPM) seconds. In milliseconds (the timescale of the testbench), this would be 1.25/(GEAR\_RATIO\*RPM)\*1000. This duration will subsequently be used to determine the necessary delay between an edge on one channel and an edge on the other.

To ensure that the two signals from the respective encoder channels are correctly phased and held low/high for the necessary duration to represent the actual behavior of the encoder (at a certain RPM), the following conditions must be met:

* Between each logical level change on the same encoder channel, 2\*((1.25/(RPM\*GEAR\_RATIO))\*1000) milliseconds must elapse.
* Between an edge on one channel and an edge on the other, ((1.25/(RPM\*GEAR\_RATIO))\*1000) milliseconds must elapse.

It is desired that each repeat construct lasts exactly one second to effectively test the functionality of the Verilog HDL (i.e., the actual capability of Verilog HDL to count the edges on the two channels) while optimizing simulation times.

To achieve this, consider the following example:

repeat ((0.5/(1.25/(1\*51)))) begin

channelA = ~channelA; #((1.25/(1\*GEAR\_RATIO))\*1000); // 1 RPM

       channelB = ~channelB; #((1.25/(1\*GEAR\_RATIO))\*1000); // 1 RPM

end

As mentioned earlier, the presented code correctly emulates the dynamics of the two square waves produced by the motor's physical encoder when operating at 1 RPM.

Based on this, to achieve a total duration of the repeat construct of 1 second (for simulation efficiency), it is necessary to carefully evaluate the parameter provided as input to the repeat, which follows this logic:

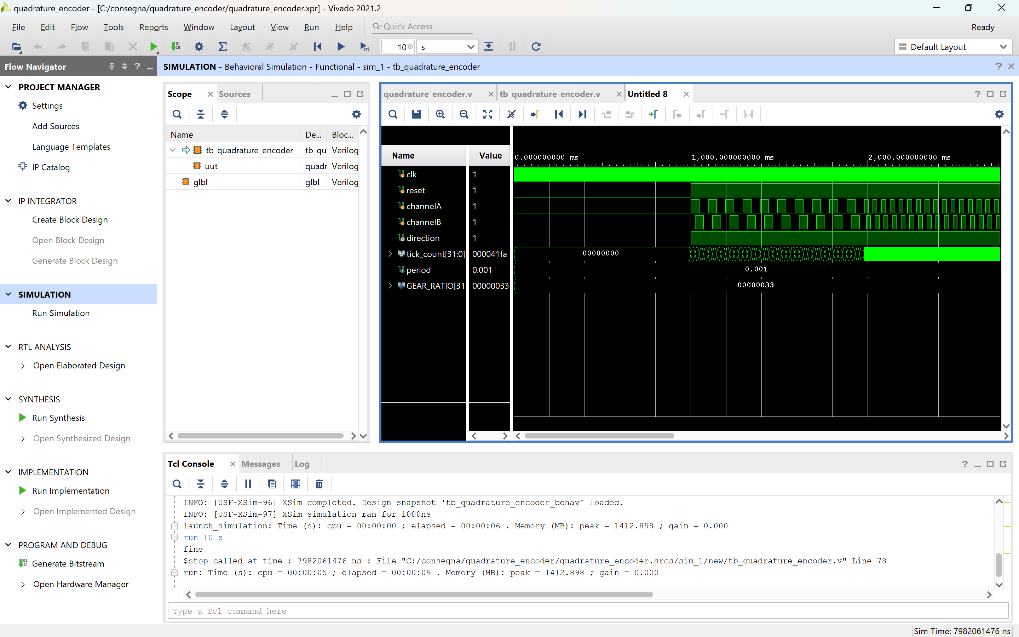
The number of repetitions is calculated based on the desired simulation duration (1 second) divided by the duration of a single repeat (as highlighted within the construct itself). Specifically, within the repeat, there are two delays. Therefore, it is necessary to halve the number of repetitions compared to the case where there would be only one delay within the repeat. For this reason, the number of repetitions will not only be 1/((1.25/(1\*GEAR\_RATIO))\*1000) but the same quantity divided by two.

## 

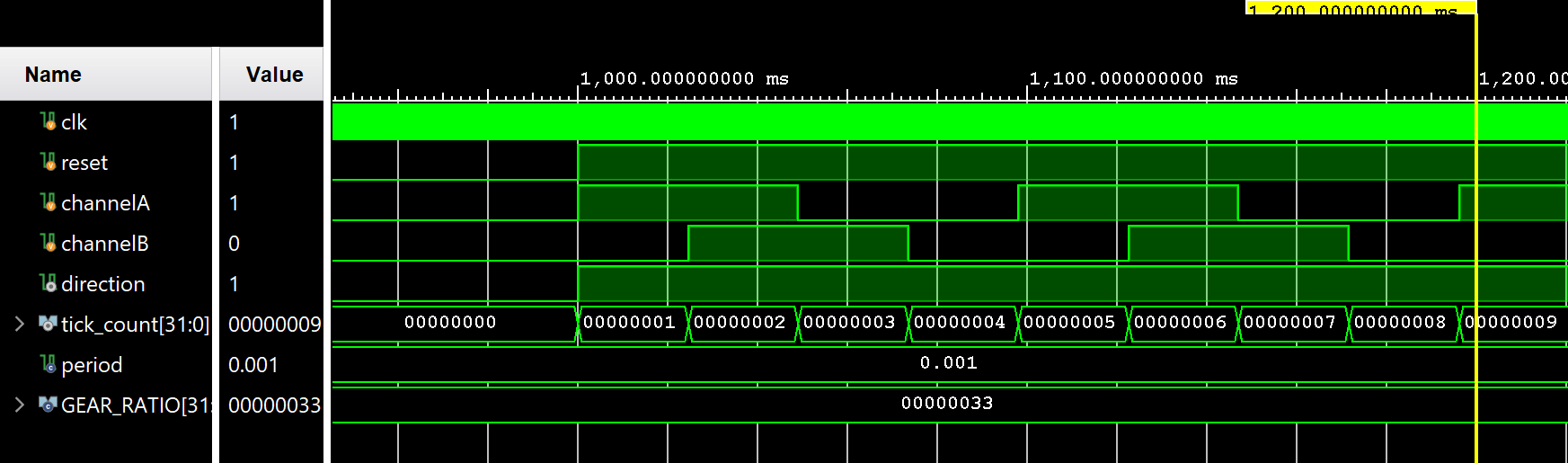
## Simulate

Click on ‘Run Simulation’ in the Flow Navigator’ pane. Accept the behavioral simulation.

The simulation is visible in the ‘Untitled 1’ window that is probably hidden close to the HDL code. Select it.



Since the simulation that is requested lasts longer, it is necessary to wait for the simulation to run and then execute in the TCL Console “run 10 s” or “run -all”; then verify that the circuit works as expected.



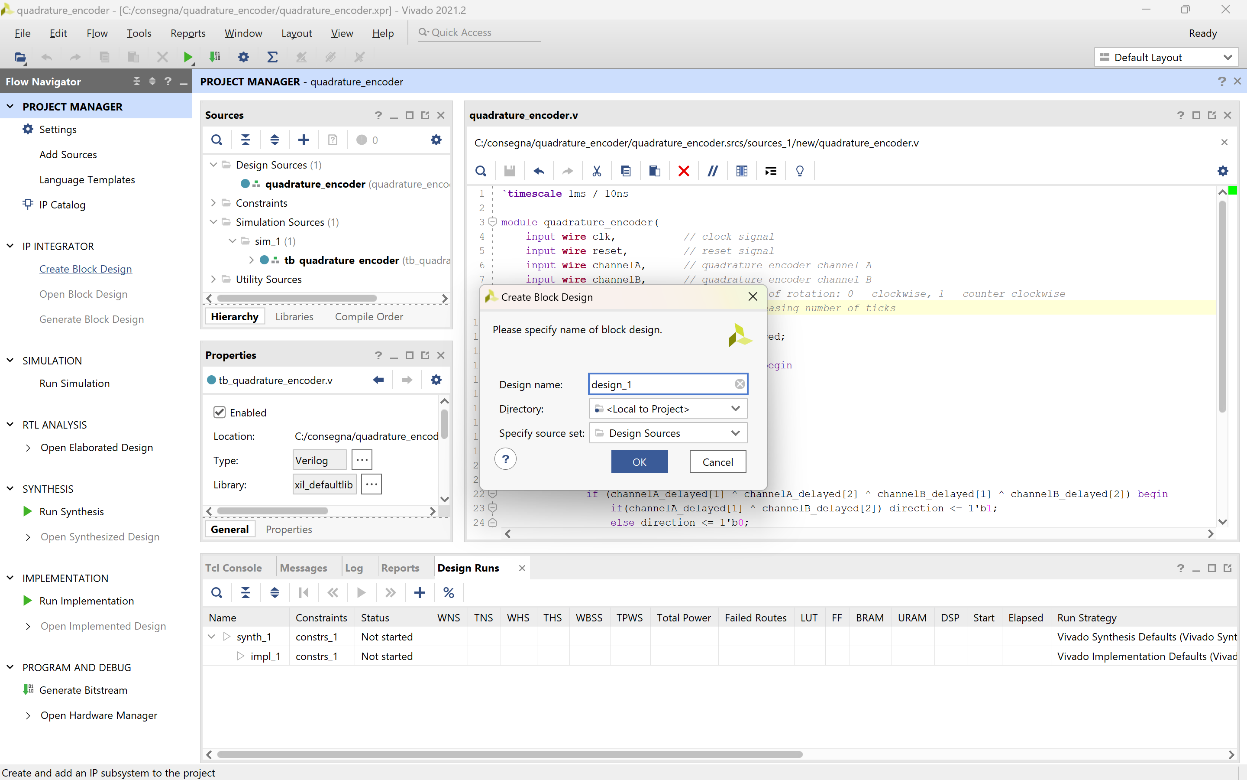
In our case, we must check that:

* the clock switches as we designed, at 1MHz
* the reset works as expected, by staying up for 1s, while all the inputs stay down
* after 1s, the reset goes down, and the signals channelA and channelB turn on and off alternately, according to the RPM of the simulation case.
* the tick\_count signal increments by 1 each time either of the two channels has a rising or falling edge.
* the direction signal is coherent with the progressing of the simulation, that is it has value 1 when the motor is operating in a clockwise fashion, and 0 when it is operating in a counter-clockwise fashion.

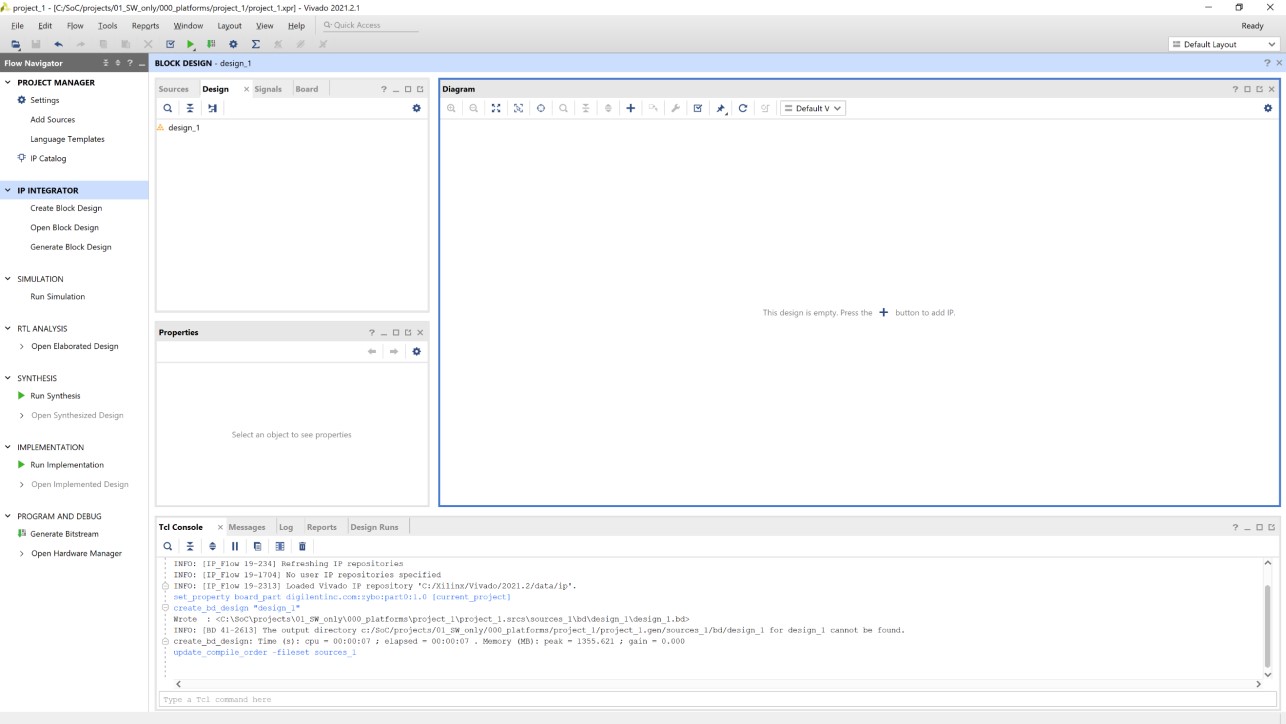
## Define the schematic of the HW platform

In this phase other HW component are added to the entire system to design the HW platform. This could also be done in HDL using the instantiation templates of the components. The procedure would be very tedious and error prone. It is then much easier to using a schematic description of the entire system and add to the same the HW component designed at the previous sections.

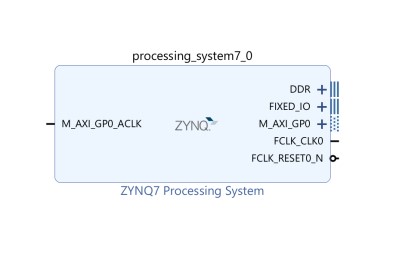
To design the schematic of the system, start by clicking on ‘Create Block Design’ in the ‘Flow Navigator’ window. Chose a name for the design even if the default one is usually adequate.



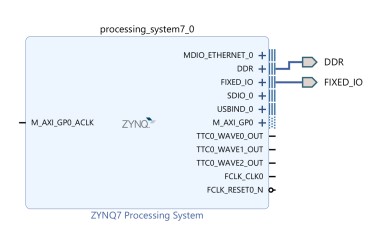
A blank slate will be shown in the main window in which the desired block can be instantiated and connected in an intuitive manner.



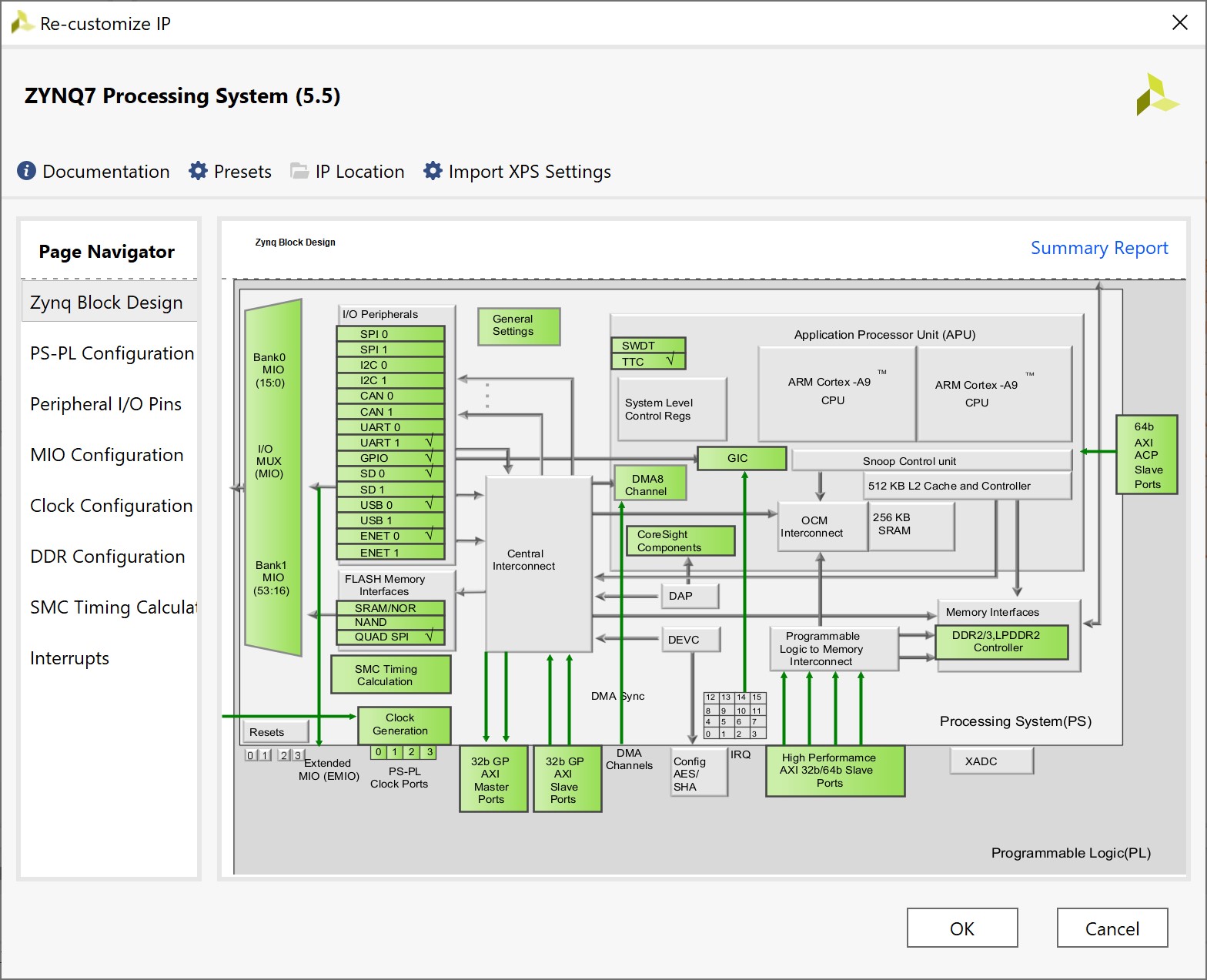
Click on the ‘+’ symbol to add new components. Write Zynq in the search box and select ‘ZYNQ7 Processing System’. The ARM based processing system will be added to the design,



The top of the window is now showing a green strip which says, ‘Designer Assistance available’ and then, ‘Run Block Automation’ (in blue to show that the writing can be clicked). Most of the times this allows to run a set of useful automatic tasks. Click on ‘Run Block Automation’, leave the default configuration, and let Vivado configure the PS for the Zybo board. The PS is now configured. Note the connection to the DDR available on the board and the fixed I/O connections.



The configuration can be further simplified and adapted the needs of the design. Double click on the PS to open the configuration window.



It shows all the components of the PS. In green those that can be configured (double clicking on the block or by selecting the proper section in the ‘Page Navigator’ on the left. Let us proceed by selecting the sections on the left one by one.

1. Peripheral I/O Pins:
   1. Disable Quad SPI Flash
   2. Disable Ethernet 0
   3. Disable USB 0
   4. Disable SD0
   5. Disable TTC0
2. DDR configuration:
   1. Expand Training/Board Details. Expand DQS to Clock Delay (ns). Set DQS0, DQS1, DQS2, and DQS3 to 0.

3) Clock configuration:

a) In PL Fabric Clocks, we set the FCLK\_CLK0 frequency at 50 MHz, which is more than enough to allow us both to have an efficient sampling of channelA and channelB signals and to match the requirements for the display implementation ([Getting the PmodOLEDrgb to Work on Zybo : 10 Steps - Instructables](https://www.instructables.com/Getting-the-PmodOLEDrgb-to-Work-on-Zybo/)).

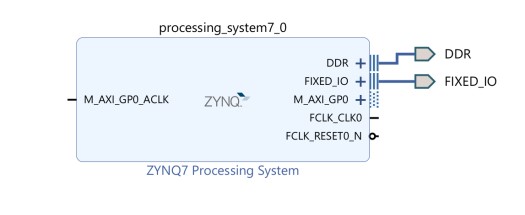
4) Interrupts:

a) To utilize the interrupt mechanism associated with timers, specifically to ensure that the interrupt handler is invoked (and thus the ISR is executed) precisely every 5 milliseconds, it is necessary to navigate to the Fabric Interrupts section > PL-PS Interrupt Ports > IRQ\_F2P[15:0] and enable both IRQ\_F2P[15:0] and Fabric Interrupts.

Click OK. The PS is now reconfigured. Click on the button on the top that optimizes the routing of the scheme whose icon is shown below on the left. This is only useful for visualization purposes. The button is often used in conjunction with its companion that also moves the blocks, shown below on the right, always with the goal of optimizing the visualization.

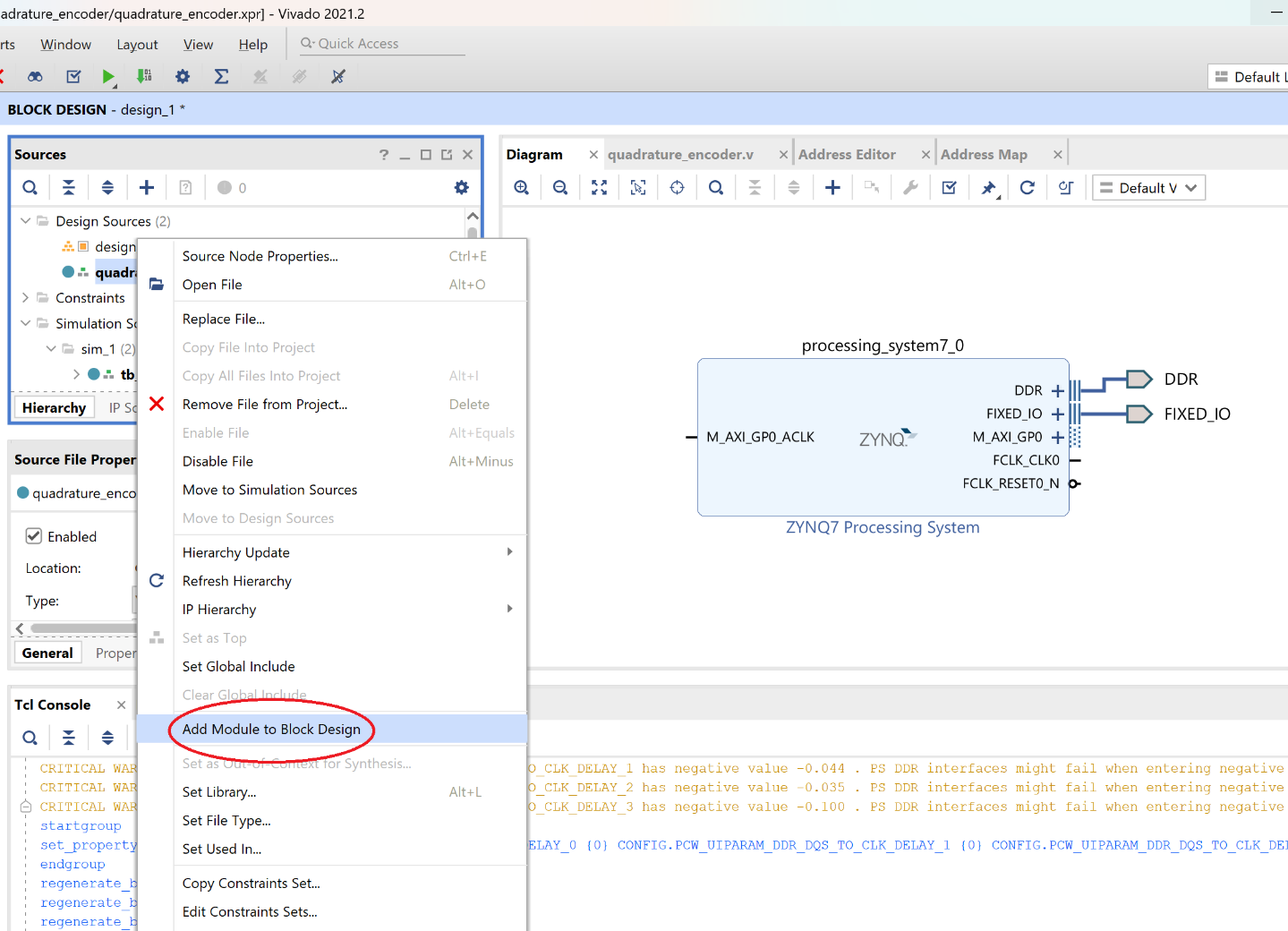
 

Note the reduced number of interfaces for the reconfigured PS that leave activate the Master AXI GP port and clock/reset signals directed to the PL.

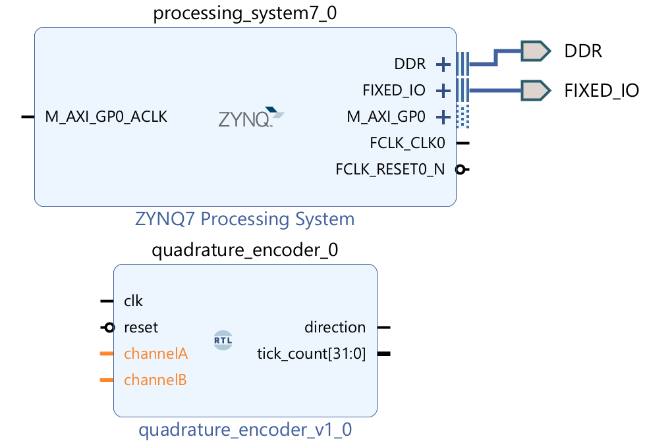


*Complete the schematic*

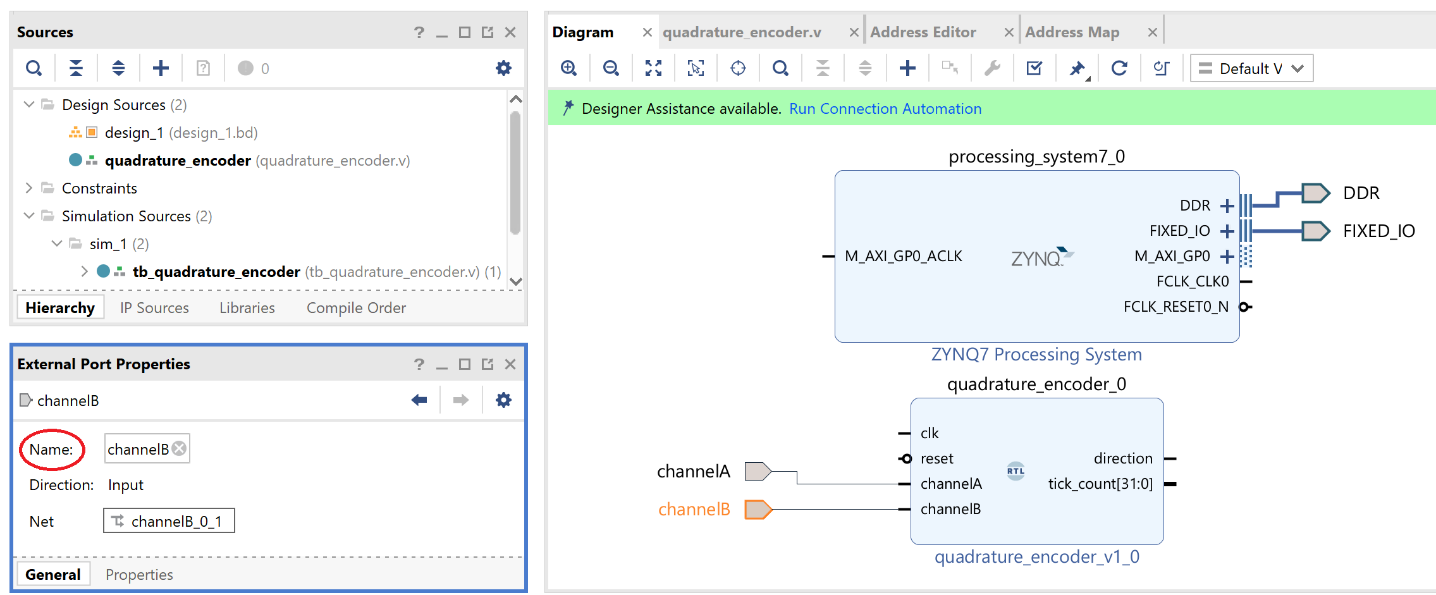
In the Sources pane of the Block Design, expand Design Sources and right click on the HDL description of the circuit (file quadrature\_encoder.v). Select ‘Add Module to Block Design’.



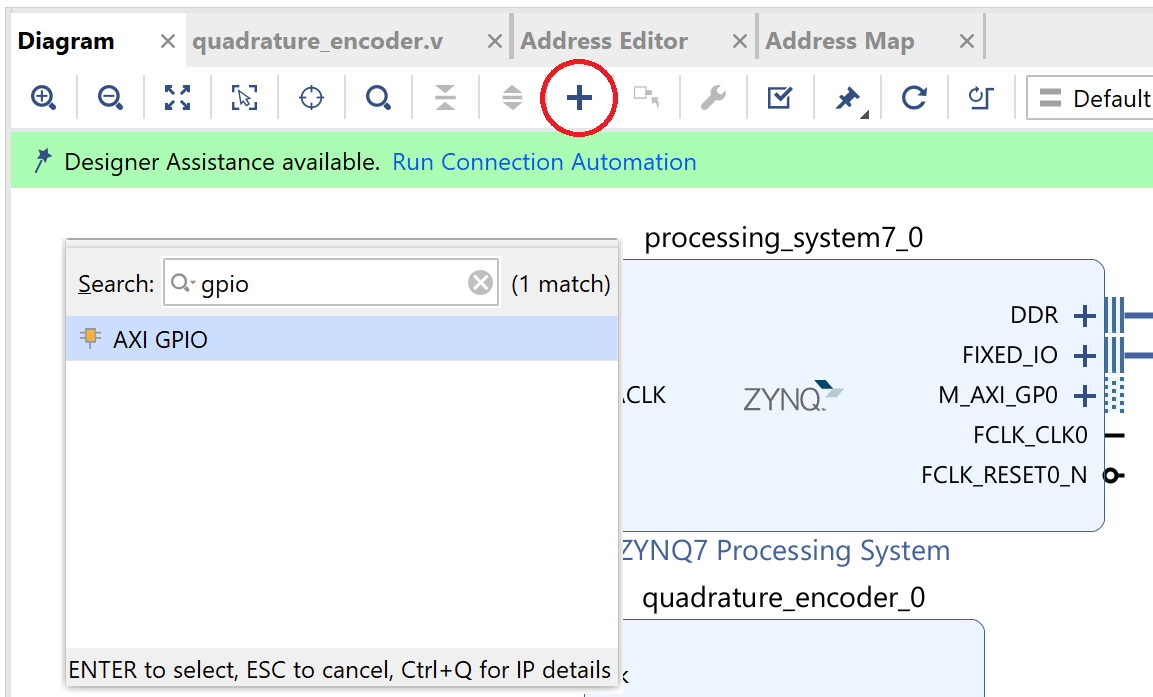
A new block is generated and added to the design.



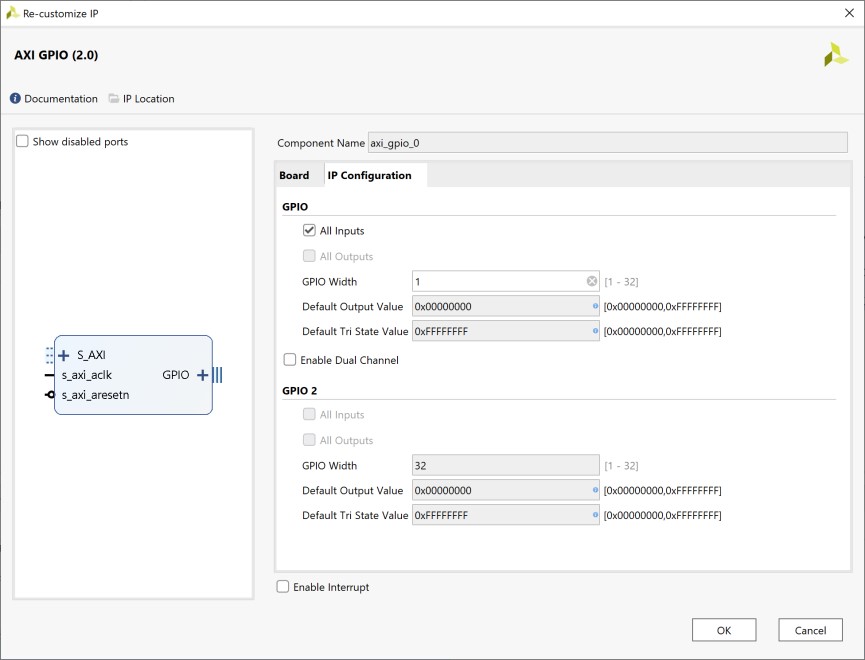
From design specification we need the input of the quadrature\_encoder block to derive from outside the chip. Select the ‘channelA’ and ‘channelB’ ports, right click and select ‘Make External’ (or type Ctrl+T). Select the output pins and name them simply ‘channelA’ and ‘channelB’ (not ‘channelA\_0’ and ‘channelB\_0’) in the External Port Properties window (located below the Sources window).



The ‘tick\_count’ and ‘direction’ ports of the ‘quadrature\_encoder’ component need to reach the PS portion of the Zynq. We decided to make the connection through the GP port. This cannot be done immediately but needs to happen thorough the AXI protocol. One possibility for such a simple interface is using two AXI Slave interfaces that are yet made available in Vivado. In the Block Schematic Diagram click on the plus icon and add the component ‘AXI GPIO’ two times.



Double click on the first AXI GPIO (axi\_gpio\_0) component to customize the IP. In the ‘IP Configuration’ pane do not enable the dual channel option, set the ‘GPIO Width’ to 32, and select ‘All Inputs’. Then, double click on the second AXI GPIO component (axi\_gpio\_1) and in the IP Configuration pane set the ‘GPIO Width’ to 1, and again select ‘All Inputs’.

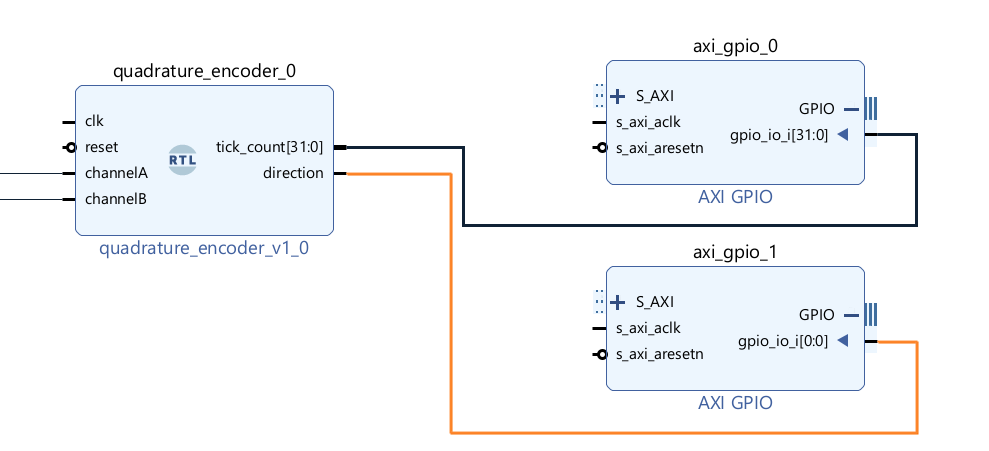


The ‘tick\_count’ and ‘direction’ ports need to be connected the GPIO ports of the AXI Slave modules.

Hover the mouse on the GPIO ports of the AXI GPIO blocks. A double down arrow appears. Left click to expand the interface and show the input ports.

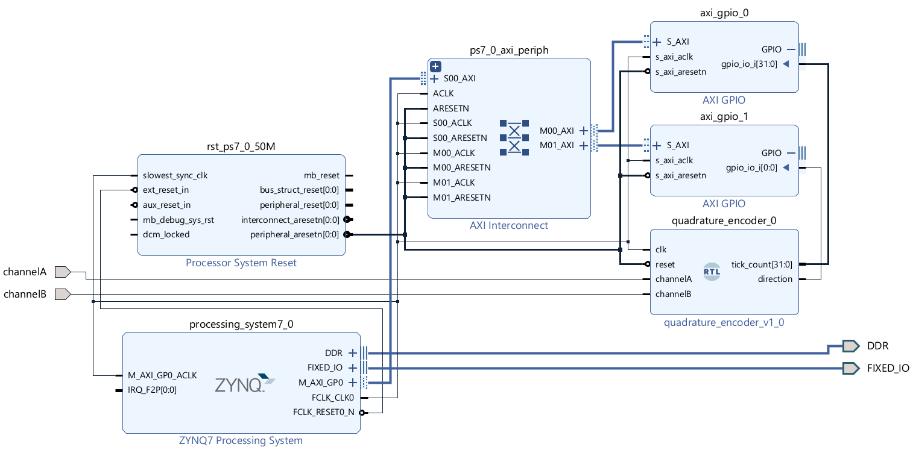


Click and drag the wire studs exiting from the IP ports of the quadrature\_encoder block and connect them to the gpio\_io\_i[31:0] port (tick\_count) and to the gpio\_io\_i[0:0] port (direction) as shown below.

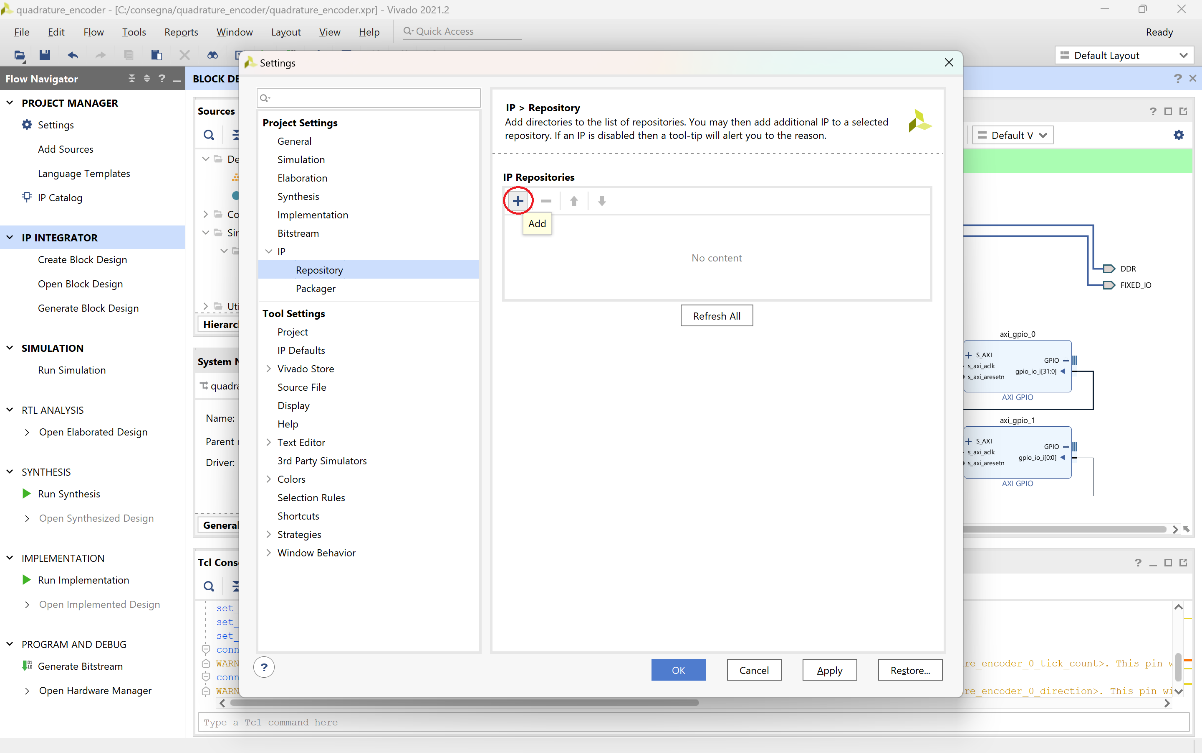


To complete the connection of the quadrature\_encoder block to the PS the AXI interconnect is needed. This is better done by the ‘Run Connection Automation’ command that is visible at the top of the Bloch Schematic Diagram. Click on ‘Run Connection Automation’. Enable, if not yet present, the automatic connection of the S\_AXI interfaces and click OK. Click the icon ‘Regenerate Layout’. The diagram is now similar to what is shown below.

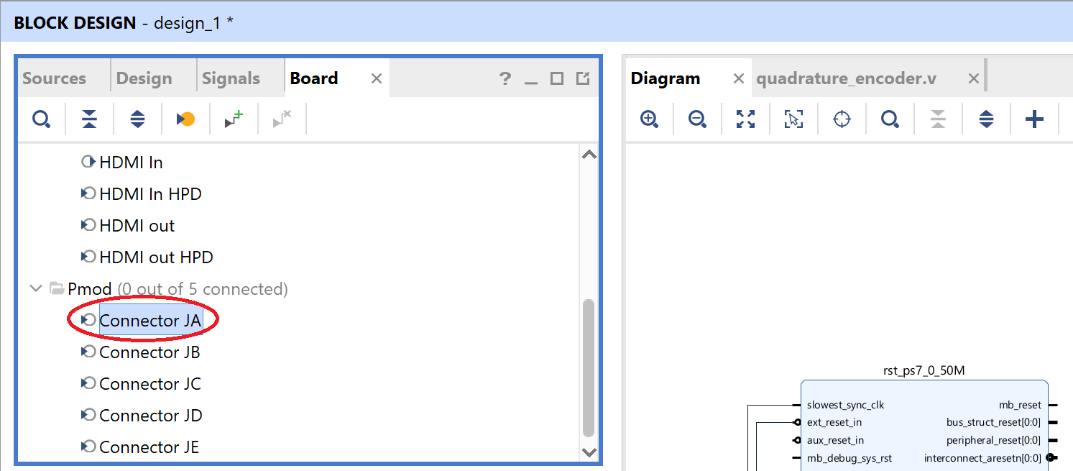
Note the presence of the AXI Interconnect block that exposes a Slave interface towards the GP AXI Master interface and two Master interfaces towards the AXI GPIO Slave interfaces (for both tick\_count and direction). Note also the presence of a reset block that handles the correct reset sequence of the platform and the FCLK\_CLK0 signal that arrives from the PS and drives the PL logic.



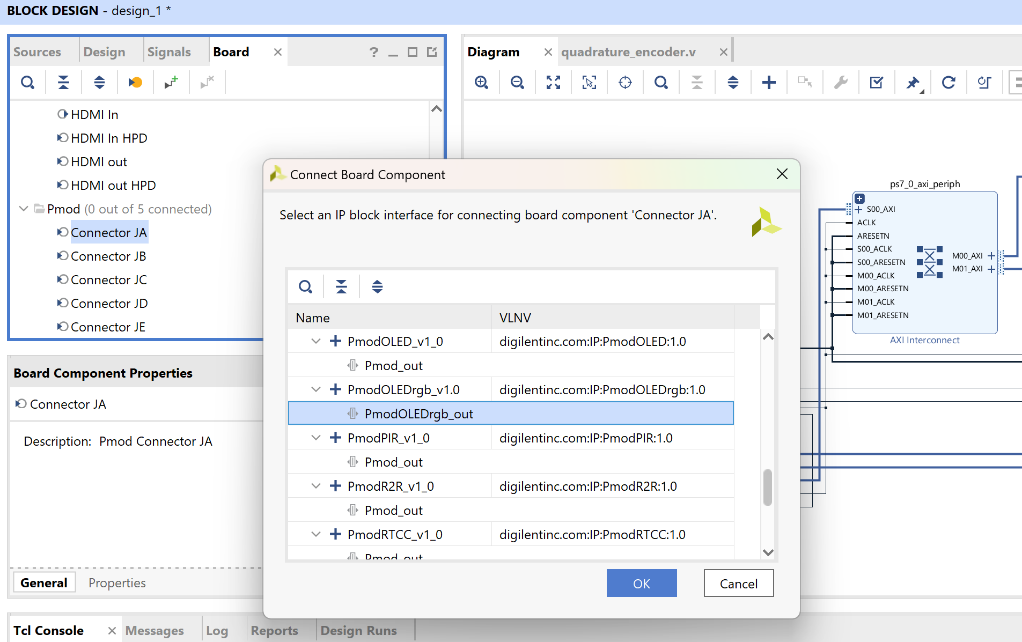
To complete the schematic, we need to add the drivers for the display. The first step is to add the "vivado-library-master" folder, as specified in the tutorial provided for the display, into the project directory. Once this is done, it is necessary to link the libraries to the Vivado project. Press "Settings" under Project Manager, then navigate to IP -> Repository, click the "+" button to add a new repository, and select the previously mentioned folder. This will correctly add the library containing the drivers for the instantiation and use of the PMODoledrgb display. Press Apply then OK.



Then navigate to the "Board" tab next to Sources, select Pmod -> Connector JA (the JA connector is a set of pins on the board).

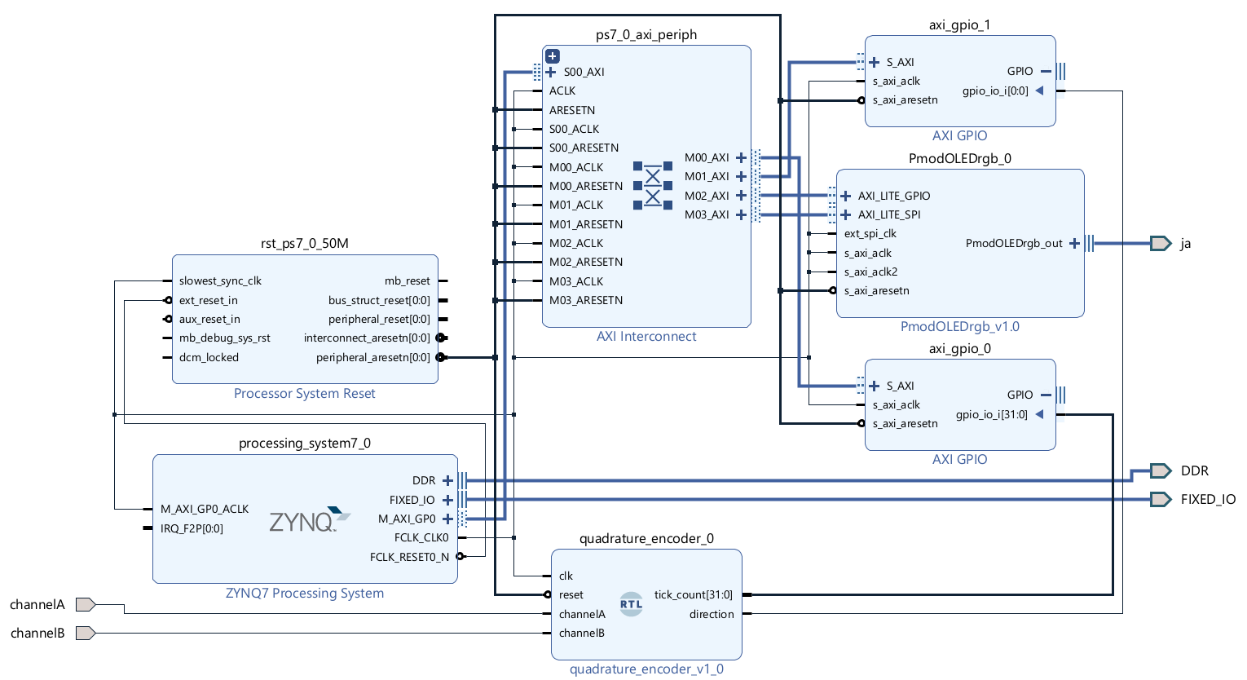


Then search for "PmodOLEDrbg\_v1.0" and select it, confirming with "OK".



Finally, run "Run Connection Automation" again, selecting "All automation". Upon completing this procedure, the display will be correctly positioned and configured within the Block Design, equipped with both an interface to the AXI interconnect (an SPI one, since the display uses the SPI protocol to receive data) and an output to the JA connector as previously specified.

The final schematic is shown below.

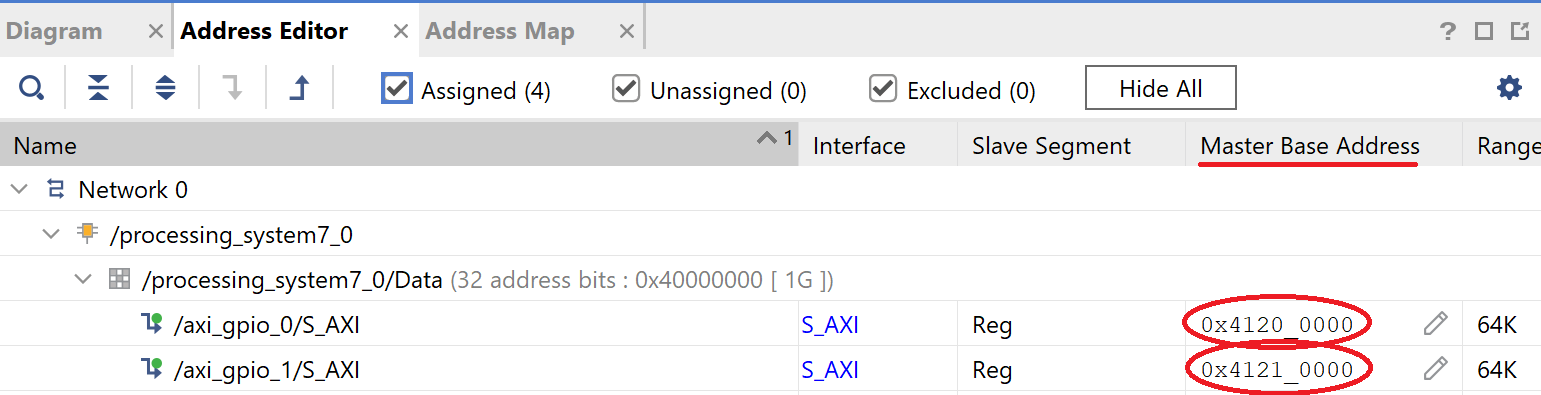


## Validate the schematic and check the memory mapping

The Block diagram can be checked for errors. Press F6 or click on the ‘Validate design’ button on the bar that contains the buttons for routing and block placement optimization. The design should highlight no critical errors or warnings.

Also, the reader can open the Address Editor tab to analyze the memory mapped components. The reader is adviced to take notes about the two Master Base Addresses since they will be used later on.

The Address Editor shows the following table. Note the Base address of the memory mapped Slave block. Every time the processors will write or read from a memory location in this range, the request will be driven to the AXI interconnect that will forward the request to the IP and not to the memory.



The last important step to be carried out is needed to let the IDE know that the top level of the circuit is composed by the block design that has been designed and to create an HDL description of the circuit starting from the Block Design. The procedure requires you to select the block design in the Design Source section of the Sources panel, right click to select ‘Create HDL Wrapper…’. Let Vivado manage the files as in the default setup and click Ok. The procedure generates a Verilog file that instantiates the blocks and selects the Block design as the top level of the circuit.

The hardware platform is now complete.

## Simulate the SoC system

The PS model provided with Vivado comes with a set of APIs that help the definition of a testbench. The APIs can be found in the document ds940-zynq-vip.pdf available on the Xilinx website.

Using the API it is possible to simulate read and write to memory locations thus stimulating the IP. The simulation will test the behavior of the IP module interfacing with AXI GPIO. There is no possibility to test the behavior of the Zynq SW and to see if the display correctly shows the information. The simulation is not very meaningful since the IP has been tested before and the AXI GPIO component is provided as is and should be working. However, this sets a simulation possibility that is useful in more complex designs.

Start adding a new simulation source to the project. Click ‘Add sources’ in the Project Flow pane. Select ‘Add or create simulation sources’. Click Next. Select ‘Create File’ and name the Verilog file ‘tb\_Zynq.v’. Clock ‘Ok’ then ‘Finish’. Click ‘Ok’ and ‘Yes’ to skip the module definition window.

Your file is now hidden in the Project Manager pane under Simulation Sources. Expand it and double click on the file to open it. Please note that now the simulation directory contains two testbenches. To define which one the designer wants to simulate right click on the desired testbench and click on ‘Set as Top’.

Describe the testbench in HDL. Below the code:

`timescale 1ns / 1ns

module tb\_Zynq();

    reg tb\_ACLK;   // PS clock

    reg tb\_ARESETn; // PS reset

    wire temp\_clk;

    wire temp\_rstn;

    // Signals received from the AXI interface

    reg [31:0] read\_data1;

    reg [31:0] read\_data2;

    reg resp;

    //    SWITCHES

    reg channelA;

    reg channelB;   // reg, will be defined in the initial block

    // IP output

    wire [31:0]tick\_count;

    wire direction;

    localparam period=5;

    // GPIO IP registers

    // simulate with a 200MHz clock. FCLK0 is 50Mhz.

    localparam IPbaseaddress=32'h4120\_0000;

    localparam GPIO\_DATA1=IPbaseaddress+4'h0;   // data register channel 1

    localparam GPIO\_DATA2=IPbaseaddress+32'h0001\_0000+4'h0;

    always

    begin

        // Clock Generator at 200MHz

        tb\_ACLK = 1'b0;#(period/2.0);

        tb\_ACLK = 1'b1;#(period/2.0);

    end

    assign temp\_clk=tb\_ACLK;

    assign temp\_rstn=tb\_ARESETn;

    // state register channel 1

    assign tick\_count = read\_data1; // Pick the bit 0 of the data for visualization purposes

    assign direction = read\_data2[0];

    // The SoC that is tested clock, reset and the switches are connected

    design\_1\_wrapper UUT

    (.DDR\_addr(),.DDR\_ba(),.DDR\_cas\_n(),.DDR\_ck\_n(),.DDR\_ck\_p(),.DDR\_cke(),.DDR\_cs\_n(),.DDR\_dm(),

    .DDR\_dq(),.DDR\_dqs\_n(),.DDR\_dqs\_p(),.DDR\_odt(),.DDR\_ras\_n(),.DDR\_reset\_n(),.DDR\_we\_n(),

    .FIXED\_IO\_ddr\_vrn(),.FIXED\_IO\_ddr\_vrp(),.FIXED\_IO\_mio(),.FIXED\_IO\_ps\_clk(temp\_clk),

    .FIXED\_IO\_ps\_porb(temp\_rstn),.FIXED\_IO\_ps\_srstb(temp\_rstn), .channelA(channelA), .channelB(channelB));

    initial

    begin

        //SW=2'b00; tb\_ARESETn = 1'b0;#(20\*period);

        //SW=2'b00; tb\_ARESETn = 1'b1;#(10\*period);

        channelA = 0;

        channelB = 0;

        tb\_ARESETn = 1'b0;#(20\*period);

        channelA = 0;

        channelB = 0;

        tb\_ARESETn = 1'b1;#(10\*period);

        //Reset the PL

        UUT.design\_1\_i.processing\_system7\_0.inst.fpga\_soft\_reset(32'h1);

        UUT.design\_1\_i.processing\_system7\_0.inst.fpga\_soft\_reset(32'h0);#(10\*period);

        UUT.design\_1\_i.processing\_system7\_0.inst.read\_data(GPIO\_DATA1,4,read\_data1, resp);#period;

        UUT.design\_1\_i.processing\_system7\_0.inst.read\_data(GPIO\_DATA2,4,read\_data2, resp);#(5\*period);

        repeat (6) begin

            channelA = ~channelA; #((1.25/(191.0\*51.0))\*1000000000); // 186 RPM

            channelB = ~channelB; #((1.25/(191.0\*51.0))\*1000000000); // 186 RPM

        end

        UUT.design\_1\_i.processing\_system7\_0.inst.read\_data(GPIO\_DATA1,4,read\_data1, resp);#period;

        UUT.design\_1\_i.processing\_system7\_0.inst.read\_data(GPIO\_DATA2,4,read\_data2, resp);#(5\*period);

        #2;

    $stop;

    end

endmodule

## tb\_Zynq.v – code analysis

module tb\_Zynq();

    reg tb\_ACLK;   // PS clock

    reg tb\_ARESETn; // PS reset

    wire temp\_clk;

    wire temp\_rstn;

    // Signals received from the AXI interface

    reg [31:0] read\_data1;

    reg [31:0] read\_data2;

    reg resp;

    reg channelA;

    reg channelB;   // reg, will be defined in the initial block

    // IP output

    wire [31:0]tick\_count;

    wire direction;

    localparam period=5;

    // GPIO IP registers

    // Simulate with a 200MHz clock. FCLK0 is 50Mhz.

    localparam IPbaseaddress=32'h4120\_0000;

    localparam GPIO\_DATA1=IPbaseaddress+4'h0;   // data register channel 1

    localparam GPIO\_DATA2=IPbaseaddress+32'h0001\_0000+4'h0;

    localparam GPIO\_TRI1=IPbaseaddress+4'h4;

    localparam GPIO\_TRI2=IPbaseaddress+32'h0001\_0000+4'h4;

Firstly, the clock and reset signals to be used in the overall system are instantiated; then, other declarations follow to be used for the correct execution of the testbench. Interestingly, read\_data1 and 2 will be used to store the contents trasmitted via the AXI GPIO interfaces. Also, the resp register is used to store the response status of the read\_data operation from the AXI interface.

After that, some constants are defined to facilitate the upkeeping and readability of the code; specifically, based on the post-synthesis address table shown previously, the base address of the memory-mapped Slave block was defined, and subsequently, the memory addresses to access the data from the PL were determined based on this base address.

One might wonder why the operating mode of the AXI bus was not set using the line UUT.design\_1\_i.processing\_system7\_0.inst.write\_data(GPIO\_TRI1,4,32'hFFFFFFFF,resp);#(period);

The reason for this omission lies in the fact that, by default, this mode is already set to "input." Therefore, it is unnecessary to redefine it. Nonetheless, the GPIO\_TRI constants have been left in the code: would there be, eventually, the need to set the bus to “output”, a write\_data operation using that constant would suffice in changing it like so.

always

    begin

        // Clock Generator at 200MHz

        tb\_ACLK = 1'b0;#(period/2.0);

        tb\_ACLK = 1'b1;#(period/2.0);

    end

    assign temp\_clk=tb\_ACLK;

    assign temp\_rstn=tb\_ARESETn;

    assign tick\_count = read\_data1;

    assign direction = read\_data2[0];

    design\_1\_wrapper UUT

    (.DDR\_addr(),.DDR\_ba(),.DDR\_cas\_n(),.DDR\_ck\_n(),.DDR\_ck\_p(),.DDR\_cke(),.DDR\_cs\_n(),.DDR\_dm(),

    .DDR\_dq(),.DDR\_dqs\_n(),.DDR\_dqs\_p(),.DDR\_odt(),.DDR\_ras\_n(),.DDR\_reset\_n(),.DDR\_we\_n(),

    .FIXED\_IO\_ddr\_vrn(),.FIXED\_IO\_ddr\_vrp(),.FIXED\_IO\_mio(),.FIXED\_IO\_ps\_clk(temp\_clk),

    .FIXED\_IO\_ps\_porb(temp\_rstn),.FIXED\_IO\_ps\_srstb(temp\_rstn), .channelA(channelA), .channelB(channelB));

Let’s dive into the actual simulation. On top of all, The clock needs to be set. Subsequently, wires for reset and clock are created to be used within the testbench. Additionally, the input from the AXI bus is connected to both tick\_count and direction, enabling the visualization of the simulation progress. One must consider that direction only receives the first bit (LSB) of the read\_data2 bus since it is only represented by 1 bit.

Following this, the component to be tested, specifically the wrapper of the block design previously created on the Vivado platform, is instantiated within the testbench. Within the parentheses, all the necessary connections between the ports are made according to the syntax.

initial

    begin

        //SW=2'b00; tb\_ARESETn = 1'b0;#(20\*period);

        //SW=2'b00; tb\_ARESETn = 1'b1;#(10\*period);

        channelA = 0;

        channelB = 0;

        tb\_ARESETn = 1'b0;#(20\*period);

        channelA = 0;

        channelB = 0;

        tb\_ARESETn = 1'b1;#(10\*period);

        //Reset the PL

        UUT.design\_1\_i.processing\_system7\_0.inst.fpga\_soft\_reset(32'h1);

        UUT.design\_1\_i.processing\_system7\_0.inst.fpga\_soft\_reset(32'h0);#(10\*period);

        UUT.design\_1\_i.processing\_system7\_0.inst.read\_data(GPIO\_DATA1,4,read\_data1, resp);#period;

        UUT.design\_1\_i.processing\_system7\_0.inst.read\_data(GPIO\_DATA2,4,read\_data2, resp);#(5\*period);

        repeat (6) begin

            channelA = ~channelA; #((1.25/(191.0\*51.0))\*1000000000); // 186 RPM

            channelB = ~channelB; #((1.25/(191.0\*51.0))\*1000000000); // 186 RPM

        end

        UUT.design\_1\_i.processing\_system7\_0.inst.read\_data(GPIO\_DATA1,4,read\_data1, resp);#period;

        UUT.design\_1\_i.processing\_system7\_0.inst.read\_data(GPIO\_DATA2,4,read\_data2, resp);#(5\*period);

        #2;

    $stop;

    end

endmodule

Within the block in question, the process begins with a reset of channels A and B, guided by the reset signal, which is applied for extended periods. Next, the PL reset is applied following the same logic as described above, waiting for 10 time periods. Subsequently, values from the AXI bus are read.

After this, a repeat construct is executed (only one iteration for demonstration purposes) with a speed set to 186 RPM. The multiplication by one billion, instead of one thousand, is due to the time scale now being in nanoseconds. The choice of 6 is entirely arbitrary (🤪), and it will be reflected on the AXI bus (there will be 12 edges, meaning tick\_count will have the value "c" in hexadecimal at the end of the test). Finally, the input values are read one last time.

Click on ‘Run simulation’ in the ‘Flow Navigator’ window. Run the behavioral simulation. The default configuration lets the simulation run for a maximum of 100ns. Since the simulation that is requested lasts longer it is necessary to wait for the simulation to run and then i) press F3, or ii) click on the ‘Run All’ button on the top (the button that looks like a Play button), or iii) execute the ‘run -all’ command in the Tcl Console. The result of the simulation only shows the signals of the testbench. For a better understanding of the simulation the designer can add some AXI signals to the simulation. The action can be conducted by selecting in the ‘Scope’ window the signals of interest and add them to the simulation (right click on the signal and select ‘Add to Wave window’). Note that on the AXI interface only read requests are issued.

## 

## Above is the result of the simulation: it is possible to observe how, based on the high clock frequency, signals are managed. Moreover, it is possible to see both the correct operation of the counting mechanism and the operations on the AXI BUS. In particular, the Read operations, which are clearly not instantaneous and thus have a certain duration, are prominently evident. No Write operations are performed, hence they are not visible.

## Add pin assignments constraints file

Note: adding the constrains is much simpler if you start from a predefined file for your board. Hence download and copy in the directory the file ‘Zybo\_Z7\_Master.xdc’.

Click ‘Add sources’ in the Project Flow pane. Select ‘Add or create Constraints’. Click Next. Select ‘Add File’. Select the ‘Zybo\_Z7\_Master.xdc’ file, ensure that the checkbox ‘Copy constraints files into the project’ is checked and click on ‘Finish’. Your file is now hidden in the Project Manager pane under Constraints. Expand it and double click on the file to open it.

Modify the file that links the I/O of the pins. Beware of case sensitiveness. Below the extract of the lines that must be uncommented and modified.

In case you are not starting from the ‘Zybo\_Z7\_Master.xdc’, it is enough to create a new constraints file and copy the above lines in the file. This will be used to specify the pins where the encoder will be connected. In our case, the EncoderA is connected to pin V12, EncoderB on W16, both on header JE.

##Pmod Header JE

set\_property -dict { PACKAGE\_PIN V12 IOSTANDARD LVCMOS33 } [get\_ports { channelA }]; #IO\_L4P\_T0\_34 Sch=je[1]

set\_property -dict { PACKAGE\_PIN W16 IOSTANDARD LVCMOS33 } [get\_ports { channelB }]; #IO\_L18N\_T2\_34 Sch=je[2]

## Synthesize and implement the circuit up to the bitstream

At this point the design is ready to be finalized. You can directly click on ‘Generate Bitstream’ in the Flow Navigator pane (click Ok to let the IDE run the complete implementation of the circuit). Serially going through the various steps is not very interesting being the circuit composed by one block that has a fixed HW implementation. If you decide for directly generate the bitstream there is no need to open the implemented design or view the reports, click ‘Cancel’ when this is requested.

## Moving to the SW development – VITIS

The project defined an HW platform composed by two ARM processors and a series of modules.

The design needs to be completed by designing the Software portion of the system.

The step is conducted in a completely different IDE named Vitis. However, Vitis needs to have information regarding the HW platform that the software portion is using, and this is obtained by exporting the hardware in a proper format (a .xsa file) that will be read by Vitis.

### Export Hardware

Select the menu: File -> Export -> Export Hardware..

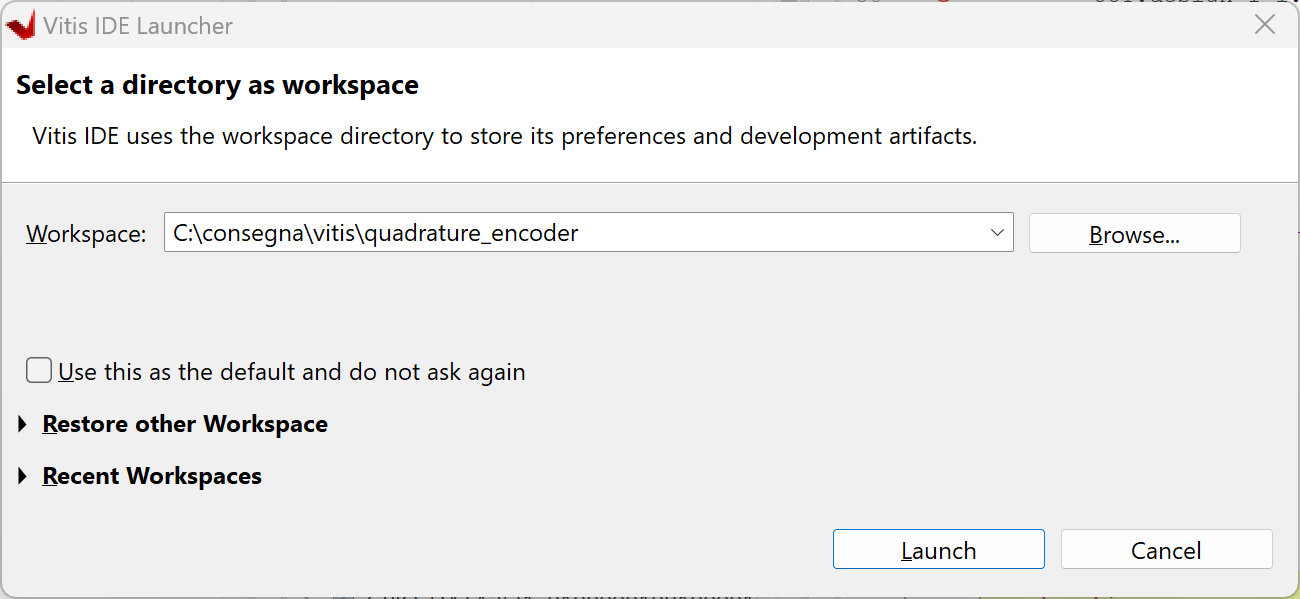
Click Next. Check the box to include the bitstream in the .XSA file (the bitstream is a .BIT file that contains the information to be downloaded on the FPGA – the Zynq in our case – to configure the hardware). Click Next to reach the window in which a name and a path for the generated .XSA file are requested. You can leave the default values for name and path or chose a more appropriate one that reflects the characteristics of the platform. Click Next and Finish.

# Using the Vitis IDE

## Setting the environment

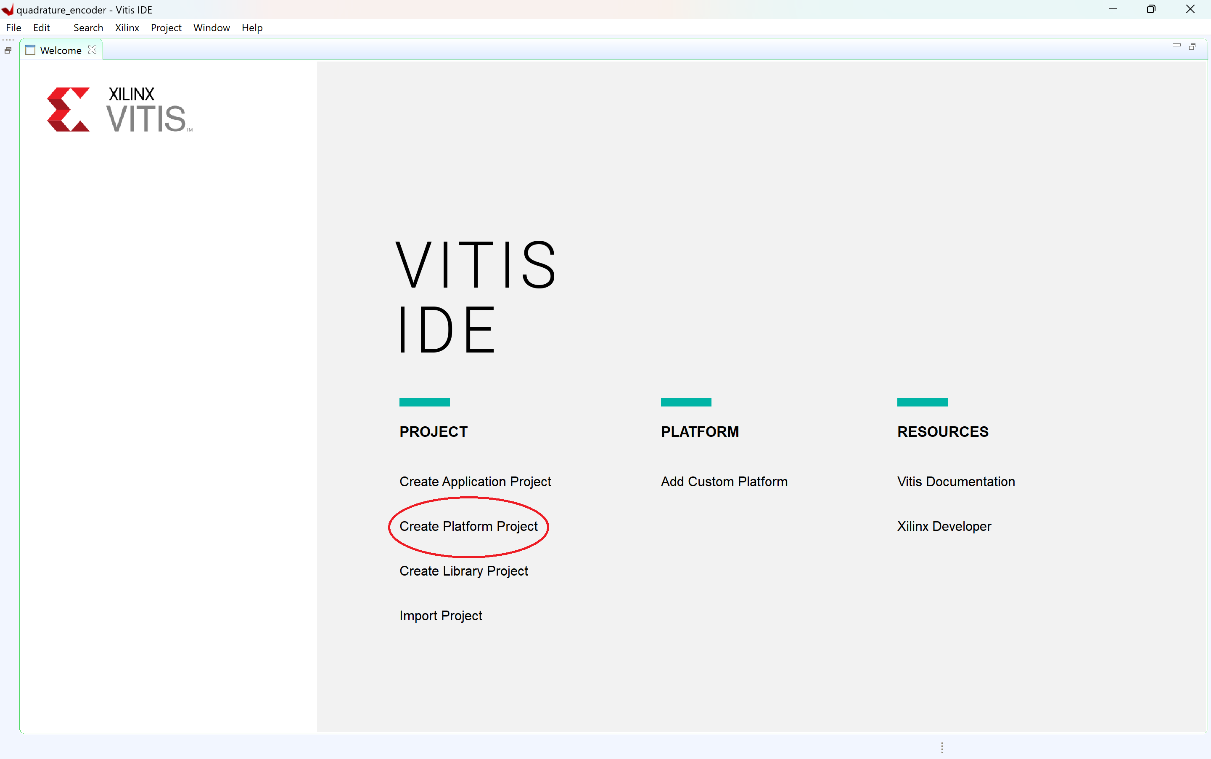
Launch Vitis 2021.2

The first window asks the directory to be used for the workspace. Here it is suggested to use a path that is only for these applications. In other cases, the choice can be different and depends on the target. As an example, if Vitis is used to design a single application to test a particular hardware, the directory could also be inside the Vivado project.



### Creating the Platform

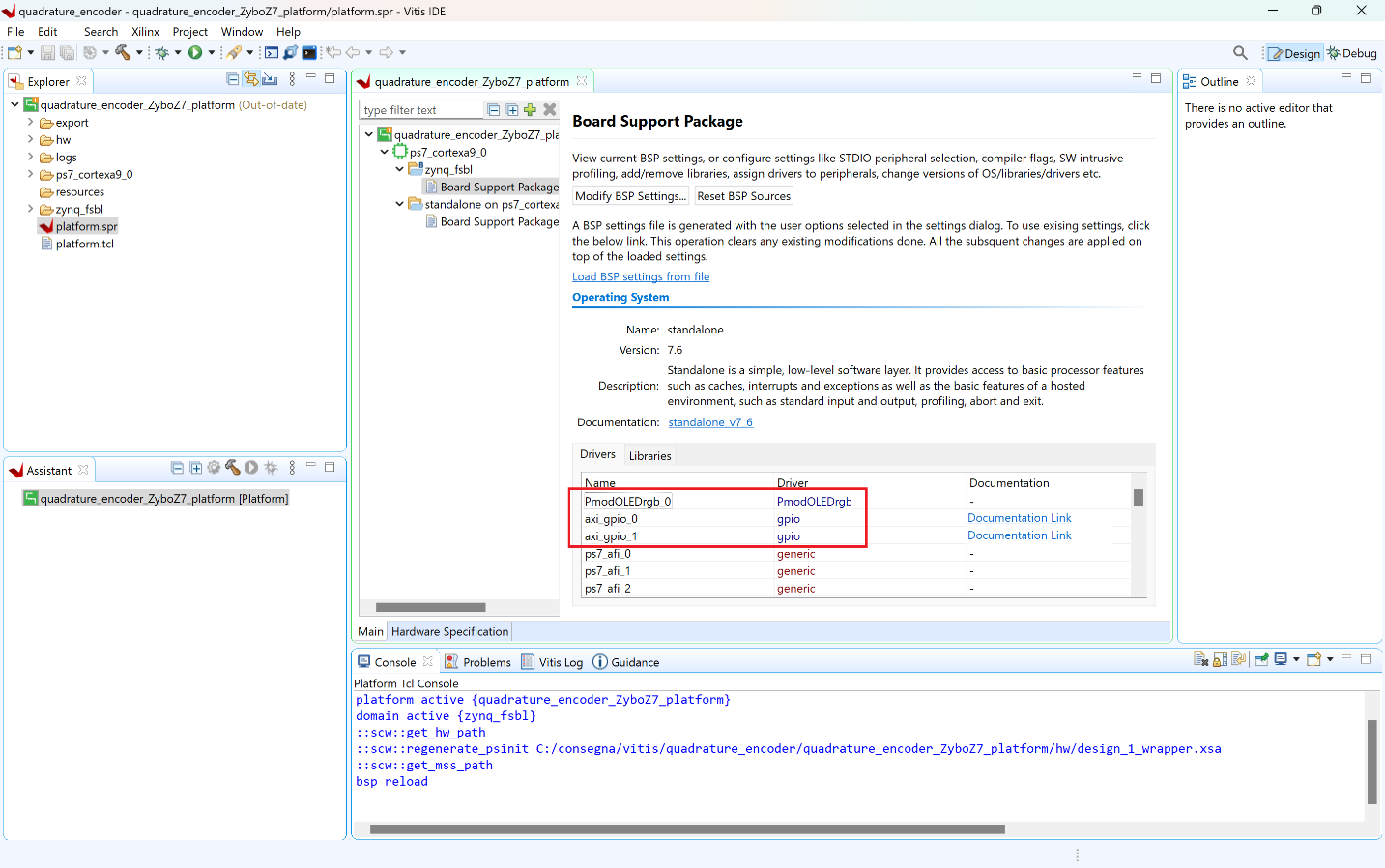
The following welcome window allows to select various activities. The first step, as said before, is generating the Platform project.



Select ‘Create Platform Project’, give a name to the platform such as quadrature\_encoder\_ZyboZ7 \_platform. Click Next. Select the .xsa file generated when defining the hardware (click on ‘Browse’, navigate to the directory that contains the .xsa file, probably named design\_1\_wrapper.xsa and select the file). Note the window in which it is possible to select the type of software that you want to run (standalone that means a software without any support from an operating system, freertos10, or Linux) and the processor (remember there are two ARM processors on the Zynq system). Be sure that the checkbox ‘Generate Boot components’ is checked, it allows the generation of the software that allows the boot of the system. Click ‘Finish’. The platform is now generated, it is shown at the left of the main window of the Vitis IDE. A single platform can host a variety of software applications (in the same way the x86 processor in a personal computer can run many applications).

The IDE is divided in many windows and recalls the Eclipse environment. On the top left the list of platform and applications that compose the environment. At the center the main window where the engineer operates. On the bottom the window that reports messages.

Double click on ‘platform.spr’ on the left pane, expand the platform icon on the main panel and click on ‘standalone on ps7\_cortex9\_0’ and on ‘Board Support Package’.



The main window shows details on the Board Support Package that is the list of drivers that enable the software at driving the component of the underlying hardware (remember that all the components are memory mapped, this means that the drivers are quite simple abstractions of memory read and write operations).

Note that selected platform includes the drivers for the **ps7\_gpio** that are for the PS connection to the MIO and includes also the drivers for the PmodOLEDrgb display which are used to interface correctly with the display. The drivers will be handy when designing the software application. The platform also reports links to documentation and examples that are very useful.

The layout can be changed at will and the windows can undock to float and be enlarged to cover the entire screen. It is up to you to decide your preferred layout and, possibly, save it.

## Designing the software application – Interface to the PL and implement the Quadrature Module

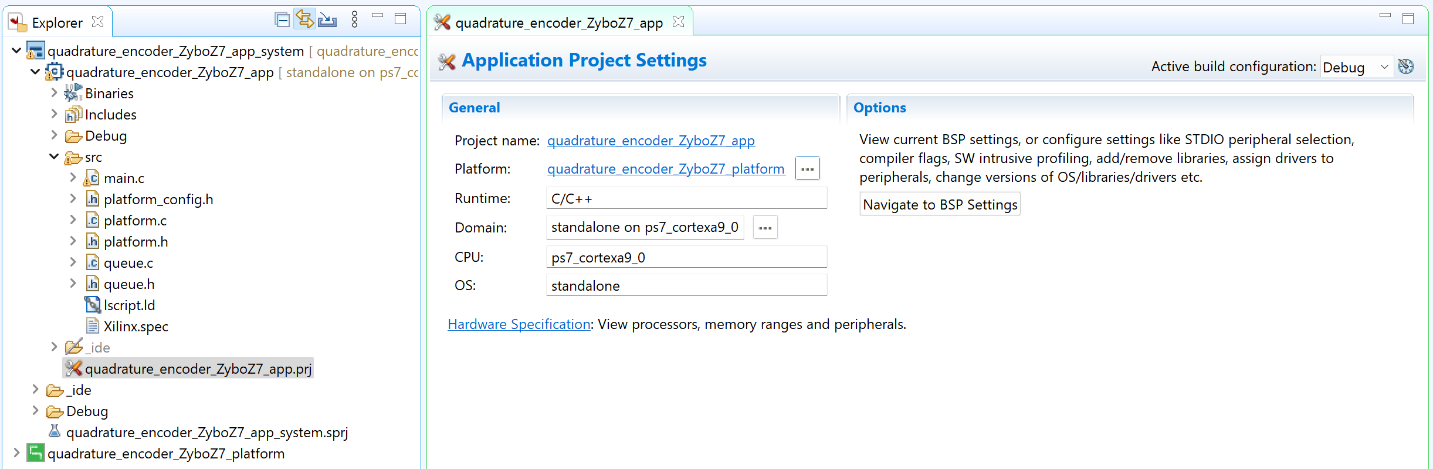
Click on File -> New -> Application Project…

An application definition wizard is created. Click Next. The following window allows to select a platform from those available. It also allows to create a new one if needed. In this case, only one platform should be present. Click ‘Next’. Give a name to the application as an example quadrature\_encoder\_ZyboZ7\_app. Click Next twice.

A list of preconfigured applications will be shown. They are handy beginning points for an application. Select ‘Hello World’ and click ‘Finish’.

The application is created and is visible in the Explorer panel.

Note the presence of a set of include files, source files, and linker files.



At this point, we can also notice that in platform > export > hw > **drivers** > PmodOLEDrgb > source will be contained the libraries (that is, the imported use-ready functions) that will be used to command the display to show the statistics regarding the motor’s movements. In the same section, we can also find the libraries for the SPI communication, which is required to send information to the display.

Finally, it is also necessary to import the self-developed libraries for queues, which are essential for managing the number of ticks received within a 50ms interval and for providing the speed to the user every 5ms. The queue is clearly circular so that after the first 10 iterations it is full and ready, and on the 11th iteration, the oldest element is replaced, and the "new" last 50ms are ready for processing the next 5ms.

To import these libraries, assuming they are already available on your machine, right-click on "source" and select "import source." From there, simply click on "browse" and navigate to the location of the libraries. One can also create both header and source files for the library by right clicking on the src folder of the project on the explorer tab, then select New -> Other -> C/C++ -> Header for the header or Source for the source.

The code for the implemented circular queue will not be illustrated here, as it is both straightforward in its implementation and widely known, far from being a novelty.

### Writing the code for the SW application

The code is to be added in the ‘src’ folder of the app. For the sake of clarity rename the helloworld.c file with a more meaningful name (e.g. main.c) and open it.

The C code needs to be thoroughly altered to be able to finally make use of the interrupt system, to send information to the display, to make RPM calculations and so on.

The code structure is typical of a microcontroller standalone code with an infinite loop.

It is composed of the following blocks

1. Define the drivers and the libraries
2. Initialize the drivers for the components accessed by the CPU
3. Initialize the display and the queue
4. Define the contents of the timer callback
5. Self-defined interface functions with the display library, useful for easier and more intuitive access to the library's features.
6. Enter an infinite loop - while (1), and display on the OLED screen, but only after resetting the text color and calling the CPU initialization function.

Below the code. Let us analyze it step by step. To better understand what a particular function is doing, the reader can place the prompt on the function or on a variable and press F2. This will indicate where the object is defined and allows to read the relevant code. The reader will notice that many times the function will resolve towards a low level Xil\_Out32 or Xil\_In32 function that reads/writes to the memory.

##### SW APPLICATION CODE

#include "xil\_cache.h"

#include "PmodOLEDrgb.h"

#include "xscugic.h"

#include "xscutimer.h"

#include "xil\_exception.h"

#include "xtime\_l.h"

#include "platform.h"

#include "xil\_printf.h"

#include "xparameters.h"

#include "xgpiops.h"

#include "xgpio.h"

#include "queue.h"

#include <math.h>

#define INTC\_DEVICE\_ID XPAR\_SCUGIC\_SINGLE\_DEVICE\_ID

#define TIMER\_DEVICE\_ID XPAR\_SCUTIMER\_DEVICE\_ID

#define TIMER\_INTERRUPT\_ID XPAR\_SCUTIMER\_INTR

#define TS 0.05

#define MOTOR\_CHARACTERISTIC 3592

XScuGic InterruptController; *// Interrupt controller instance*

XScuTimer Timer; *// Timer instance*

*//XTime tEnd, tStart;*

PmodOLEDrgb oledrgb;

int tick\_count, direction;

XGpio\_Config \*pl\_tick\_count\_Ptr;

XGpio pl\_tick\_count;

XGpio\_Config \*pl\_direction\_Ptr;

XGpio pl\_direction;

CircularQueue q;

int speed, acceleration;

int last\_speed = 0;

int oldestTicks;

int delta;

void init\_Zynq(){

    init\_platform();

    init\_oled();

    init\_queue(&q);

*// GPIO initialization*

    pl\_tick\_count\_Ptr = XGpio\_LookupConfig(XPAR\_AXI\_GPIO\_0\_DEVICE\_ID);

    int status0 = XGpio\_CfgInitialize(&pl\_tick\_count, pl\_tick\_count\_Ptr, pl\_tick\_count\_Ptr->BaseAddress);

    pl\_direction\_Ptr = XGpio\_LookupConfig(XPAR\_AXI\_GPIO\_1\_DEVICE\_ID);

    int status1 = XGpio\_CfgInitialize(&pl\_direction, pl\_direction\_Ptr, pl\_direction\_Ptr->BaseAddress);

    if (status0 != XST\_SUCCESS && status1!= XST\_SUCCESS) {

        xil\_printf("GPIOs initialization error!\r");

        return XST\_FAILURE;

    }

*// set data direction for GPIO*

    XGpio\_SetDataDirection(&pl\_tick\_count, 1, 1); *// 0-> output, 1-> input.*

    XGpio\_SetDataDirection(&pl\_direction, 1, 1); *// 0-> output, 1-> input.*

*// Initialize the timer*

    int status2 = InitializeTimer(&Timer);

    if (status2 != XST\_SUCCESS) {

        xil\_printf("Timer Initialization Failed\r\n");

        return XST\_FAILURE;

    }

*// Initialize the interrupt system*

    status2 = SetupInterruptSystem(&InterruptController, &Timer);

    if (status2 != XST\_SUCCESS) {

        xil\_printf("Interrupt System Setup Failed\r\n");

        return XST\_FAILURE;

    }

*// Start the timer*

    xil\_printf("INIT\n\r");

    XScuTimer\_Start(&Timer);

}

int InitializeTimer(XScuTimer \*TimerInstancePtr) { *//Si inizializza il Timer*

    int Status;

    XScuTimer\_Config \*ConfigPtr;

*// Initialize the timer driver so that it is ready to use*

    ConfigPtr = XScuTimer\_LookupConfig(TIMER\_DEVICE\_ID);

    if (ConfigPtr == NULL) {

        return XST\_FAILURE;

    }

    Status = XScuTimer\_CfgInitialize(TimerInstancePtr, ConfigPtr,

            ConfigPtr->BaseAddr);

    if (Status != XST\_SUCCESS) {

        return XST\_FAILURE;

    }

*// Load the timer with the value that corresponds to 5ms*

    XScuTimer\_LoadTimer(TimerInstancePtr, XPAR\_CPU\_CORTEXA9\_0\_CPU\_CLK\_FREQ\_HZ / 400);

*// Enable the Auto reload mode*

    XScuTimer\_EnableAutoReload(TimerInstancePtr);

*// Enable the timer interrupt*

    XScuTimer\_EnableInterrupt(TimerInstancePtr);

    return XST\_SUCCESS;

}

void init\_oled() {

   EnableCaches();

   OLEDrgb\_begin(&oledrgb, XPAR\_PMODOLEDRGB\_0\_AXI\_LITE\_GPIO\_BASEADDR,

   XPAR\_PMODOLEDRGB\_0\_AXI\_LITE\_SPI\_BASEADDR);

   OLEDrgb\_FillBackground(&oledrgb, 0, 0, 255); *//Scegli colore background*

   change\_fontcolor\_rgb(255,255,255); *//Scegli colore delle scritte*

   OLEDrgb\_SetCursor(&oledrgb, 1, 1);

   OLEDrgb\_PutString(&oledrgb, "SPEED: " );

   OLEDrgb\_SetCursor(&oledrgb, 1, 3);

   OLEDrgb\_PutString(&oledrgb, "ACCELER: " );

   OLEDrgb\_SetCursor(&oledrgb, 1, 5);

   OLEDrgb\_PutString(&oledrgb, "DIRECT: " );

}

void TimerIntrHandler(void \*CallBackRef) { *//Effettiva CallBack,*

*//XTime\_GetTime(&tStart);*

    tick\_count = XGpio\_DiscreteRead(&pl\_tick\_count, 1);

    direction = XGpio\_DiscreteRead(&pl\_direction, 1);

    oldestTicks = getOldestElement(&q);

    delta = tick\_count - oldestTicks;

    enqueue(&q, tick\_count);

    speed = delta \* 60/(MOTOR\_CHARACTERISTIC\*TS);

    acceleration = (speed - last\_speed) \* 200;

    last\_speed = speed;

    xil\_printf("%d, %d\n\r", (direction==1) ? speed : -speed , acceleration);

    XScuTimer\_ClearInterruptStatus((XScuTimer \* )CallBackRef); *// Clear the interrupt status*

*//printf("Elapsed: %.5lf ms\r\n", ((double)(tStart-tEnd) / (double)COUNTS\_PER\_SECOND)\*1000);*

*//xil\_printf("Speed: %d RPM, Acceleration: %d RPM/s\n", (direction==1) ? speed : -speed , acceleration);*

*//XTime\_GetTime(&tEnd);*

}

int SetupInterruptSystem(XScuGic \*IntcInstancePtr, XScuTimer \*TimerInstancePtr) { *//Funzione per creare la callback*

    int Status;

*// Initialize the interrupt controller driver*

    XScuGic\_Config \*IntcConfig;

    IntcConfig = XScuGic\_LookupConfig(INTC\_DEVICE\_ID);

    Status = XScuGic\_CfgInitialize(IntcInstancePtr, IntcConfig,

            IntcConfig->CpuBaseAddress);

    if (Status != XST\_SUCCESS) {

        return XST\_FAILURE;

    }

*// Initialize the exception table and register the interrupt controller handler with the exception table*

    Xil\_ExceptionInit();

    Xil\_ExceptionRegisterHandler(XIL\_EXCEPTION\_ID\_INT,

            (Xil\_ExceptionHandler) XScuGic\_InterruptHandler, IntcInstancePtr);

*// Connect the device driver handler that will be called when an interrupt for the device occurs*

    Status = XScuGic\_Connect(IntcInstancePtr, TIMER\_INTERRUPT\_ID,

            (Xil\_InterruptHandler) TimerIntrHandler, (void \*) TimerInstancePtr);

    if (Status != XST\_SUCCESS) {

        return XST\_FAILURE;

    }

*// Enable the interrupt for the device*

    XScuGic\_Enable(IntcInstancePtr, TIMER\_INTERRUPT\_ID);

*// Enable interrupts in the ARM processor*

    Xil\_ExceptionEnable();

    return XST\_SUCCESS;

}

*//Display part*

void OLEDrgb\_FillBackground(PmodOLEDrgb \*oledrgb, u8 R, u8 G, u8 B){ *//Funzione autofatta: fa il background bianco*

    OLEDrgb\_SetFontBkColor(oledrgb, OLEDrgb\_BuildRGB(B, R, G));

    OLEDrgb\_SetFontColor(oledrgb, OLEDrgb\_BuildRGB(B, R, G));

    for(int i=0; i<8; i++){

        for(int j=0; j<12; j++){

            OLEDrgb\_SetCursor(oledrgb, j, i);

            OLEDrgb\_PutString(oledrgb, " " );

        }

    }

}

void OledCleanup() {

   DisableCaches();

}

void print\_speed(char \*\*b){

    change\_fontcolor\_rgb(255,255,255);

    sprintf(\*b, "%d    ", (direction==1) ? speed : -speed);

    OLEDrgb\_SetCursor(&oledrgb, 1, 2);

    if(speed == 0){

        change\_fontcolor\_rgb(255,0,0);

        OLEDrgb\_PutString(&oledrgb, "ZERO");

    }

    else

        OLEDrgb\_PutString(&oledrgb, \*b);

}

void print\_acceleration(char \*\*b){

    change\_fontcolor\_rgb(255,255,255);

    sprintf(\*b, "%d     ", acceleration);

    OLEDrgb\_SetCursor(&oledrgb, 1, 4);

    if(acceleration == 0){

        change\_fontcolor\_rgb(255,0,0);

        OLEDrgb\_PutString(&oledrgb, "ZERO");

    }

    else

        OLEDrgb\_PutString(&oledrgb, \*b);

}

void print\_direction(char \*\*b){

    change\_fontcolor\_rgb(255,255,255);

    OLEDrgb\_SetCursor(&oledrgb, 1, 6);

    if(direction == 1 && speed>5){

        OLEDrgb\_PutString(&oledrgb, "----->      ");

    }

    else if (direction == 0 && speed>5){

        OLEDrgb\_PutString(&oledrgb, "<-----      ");

    }

    else{

        change\_fontcolor\_rgb(255,0,0);

        OLEDrgb\_PutString(&oledrgb, "STOP    ");

    }

}

void EnableCaches() {

#ifdef \_MICROBLAZE\_

#ifdef XPAR\_MICROBLAZE\_USE\_ICACHE

   Xil\_ICacheEnable();

#endif

#ifdef XPAR\_MICROBLAZE\_USE\_DCACHE

   Xil\_DCacheEnable();

#endif

#endif

}

void DisableCaches() {

#ifdef \_MICROBLAZE\_

#ifdef XPAR\_MICROBLAZE\_USE\_DCACHE

   Xil\_DCacheDisable();

#endif

#ifdef XPAR\_MICROBLAZE\_USE\_ICACHE

   Xil\_ICacheDisable();

#endif

#endif

}

void change\_fontcolor\_rgb(u8 R, u8 G, u8 B){

    OLEDrgb\_SetFontColor(&oledrgb, OLEDrgb\_BuildRGB(B, R, G));

}

*//End display*

int main() {

    init\_Zynq();

    char b[6];

    change\_fontcolor\_rgb(255,255,255); *//Scegli colore numeri*

    while (1) {

        print\_speed(&b);

        print\_acceleration(&b);

        print\_direction(&b);

    }

    return XST\_SUCCESS;

}

### Step by step explanation of the code

##### INCLUDE Section

#include "xil\_cache.h"

#include "PmodOLEDrgb.h"

#include "xscugic.h"

#include "xscutimer.h"

#include "xil\_exception.h"

#include "xtime\_l.h"

#include "platform.h"

#include "xil\_printf.h"

#include "xparameters.h"

#include "xgpiops.h"

#include "xgpio.h"

#include "queue.h"

#include <math.h>

**xil\_cache.h** Enables control over the data and instruction caches, such as enabling/disabling and performing cache flush operations. Useful for ensuring data coherency between the processor and the FPGA fabric.

**PmodOLEDrgb.h** This header defines functions and data structures for interfacing with the PmodOLEDrgb peripheral.Provides an API for controlling the PmodOLEDrgb module, which is a color OLED display module that can be connected to the ZyboZ7.

**xscugic.h** This header provides functions for the Generic Interrupt Controller (GIC) in the Zynq-7000 SoC. Used to manage interrupts on the ARM Cortex-A9 processors, including configuration and handling of hardware interrupts.

**xscutimer.h** This header defines functions for using the private timer (SCU Timer) in the Zynq-7000. Provides an API for configuring and using the SCU Timer, which is a private timer for each ARM Cortex-A9 processor, useful for timing operations and delays.

**xil\_exception.h** This header provides functions for exception handling. Used to set up and manage exception handlers, ensuring that the system can correctly handle exceptions such as undefined instructions or data aborts.

**xtime\_l.h** This header provides functions for accessing and manipulating the system time. Useful for profiling code or implementing time-dependent functionalities by providing access to a timer/counter.

**platform.h** Seen before. Contains functions to enable/disable the caches and enable the UART.

**xil\_printf.h** Seen before. Contains a reduced version of the printf function useful for embedded designs.

**xparameters.h** It is needed whenever an HW component is added to the PS, such as an IP or the MIO pins. It contains information regarding the memory mapping of the components abstracting them with more readable strings. The reader can search gpio in the xparameters.h file to retrieve information such as deviced ID, base and high address of the memory mapped GPIO MIO pins (GPIO\_PS) and the AXI GPIO interface (GPIO). It is worth noting that valuable information can also be found regarding the memory-mapped management of other peripherals and devices such as the ScuTimer or the display. Finally, since the use of two AXI GPIO interfaces was previously requested, there will be memory information available for these two AXI GPIO peripherals.

/\* Definitions for driver GPIO \*/

**#define** XPAR\_XGPIO\_NUM\_INSTANCES 2

/\* Definitions for peripheral PS7\_GPIO\_0 \*/

**#define** XPAR\_PS7\_GPIO\_0\_DEVICE\_ID 0

**#define** XPAR\_PS7\_GPIO\_0\_BASEADDR 0xE000A000

**#define** XPAR\_PS7\_GPIO\_0\_HIGHADDR 0xE000AFFF

/\* Definitions for driver GPIO \*/

/\* These are the definitions for the AXI GPIO interface \*/

**#define** XPAR\_XGPIO\_NUM\_INSTANCES 1

/\* Definitions for peripheral PMODOLEDRGB\_0 \*/

**#define** XPAR\_PMODOLEDRGB\_0\_DEVICE\_ID 0

**#define** XPAR\_PMODOLEDRGB\_0\_AXI\_LITE\_GPIO\_BASEADDR 0x43C00000

**#define** XPAR\_PMODOLEDRGB\_0\_AXI\_LITE\_GPIO\_HIGHADDR 0x43C0FFFF

**#define** XPAR\_PMODOLEDRGB\_0\_AXI\_LITE\_SPI\_BASEADDR 0x43C10000

**#define** XPAR\_PMODOLEDRGB\_0\_AXI\_LITE\_SPI\_HIGHADDR 0x43C1FFFF

/\* Definitions for peripheral AXI\_GPIO\_0 \*/

**#define** XPAR\_AXI\_GPIO\_0\_BASEADDR 0x41200000

**#define** XPAR\_AXI\_GPIO\_0\_HIGHADDR 0x4120FFFF

**#define** XPAR\_AXI\_GPIO\_0\_DEVICE\_ID 0

**#define** XPAR\_AXI\_GPIO\_0\_INTERRUPT\_PRESENT 0

**#define** XPAR\_AXI\_GPIO\_0\_IS\_DUAL 0

/\* Canonical definitions for peripheral AXI\_GPIO\_1 \*/

**#define** XPAR\_GPIO\_1\_BASEADDR 0x41210000

**#define** XPAR\_GPIO\_1\_HIGHADDR 0x4121FFFF

**#define** XPAR\_GPIO\_1\_DEVICE\_ID XPAR\_AXI\_GPIO\_1\_DEVICE\_ID

**#define** XPAR\_GPIO\_1\_INTERRUPT\_PRESENT 0

**#define** XPAR\_GPIO\_1\_IS\_DUAL 0

/\* Peripheral Definitions for peripheral PS7\_SCUC\_0 \*/

**#define** XPAR\_PS7\_SCUC\_0\_S\_AXI\_BASEADDR 0xF8F00000

**#define** XPAR\_PS7\_SCUC\_0\_S\_AXI\_HIGHADDR 0xF8F000FC

**xgpiops.h** This header contains the functions to abstract the memory mapped driving of the MIO pins. It contains high level function that implement the low level functionality of the peripheral.

It defines structures that contain information for the GPIO. It defines a set of API functions to set the status of the pins, read or write the data. It also defines functions to ease the handle of the interrupt or to test the healthy status of the GPIO.

**xgpio.h** This header is similar to **xgpiops.h** bur contains the functions that abstract the interface to the AXI Slave interface to the PL.

**queue.h** This header is part of a data structure library that provides queue functionality. Provides a standard queue implementation, which can be used for managing data in a first-in, first-out (FIFO) manner. This is useful for buffering data.

**math.h** This is a standard C library header for mathematical functions. Provides access to common mathematical functions such as sin, cos, sqrt, etc. This is useful for performing mathematical computations within the application.

##### IP INITIALIZE SECTION

The section allocates instances that point to the IP and initializes the IP. Please note that, due to design choices that led to not using EMIO pins, and consequently not utilizing any of the GPIO ports available on the Zynq processor, there are no declarations or instances of XGpioPs. However, since we are receiving two inputs via the AXI Bus, there are two GPIO declarations instead of one.

// GPIO initialization

XGpio\_Config \*pl\_tick\_count\_Ptr;

XGpio pl\_tick\_count;

pl\_tick\_count\_Ptr = XGpio\_LookupConfig(XPAR\_AXI\_GPIO\_0\_DEVICE\_ID);

int status0 = XGpio\_CfgInitialize(&pl\_tick\_count, pl\_tick\_count\_Ptr, pl\_tick\_count\_Ptr->BaseAddress);

XGpio\_Config \*pl\_direction\_Ptr;

XGpio pl\_direction;

pl\_direction\_Ptr = XGpio\_LookupConfig(XPAR\_AXI\_GPIO\_1\_DEVICE\_ID);

int status1 = XGpio\_CfgInitialize(&pl\_direction, pl\_direction\_Ptr, pl\_direction\_Ptr->BaseAddress);

if (status0 != XST\_SUCCESS && status1!= XST\_SUCCESS) {

    xil\_printf("GPIOs initialization error!\r");

    return XST\_FAILURE;

}

##### INTERRUPT, TIMER AND OTHER INITIALIZATIONS

##### This section initializes all of the system's hardware components and peripherals, including those that are accessed by the CPU, such as interrupts and the timer.

##### First, some system and convenience definitions are made (such as TS 0.05). Next, global variable references to the interrupt controller, timer, and display are initialized. Additionally, global variables are created to store various processing and calculation elements used later, and a circular queue instance is also created.

Inside the init\_Zynq function, three functions are immediately called:

* init\_platform() enables the caches and initializes the UART.
* init\_oled() initializes the display, providing essential information such as the memory addresses to access. It also prints fixed information to the screen, such as labels, and sets the background color.
* init\_queue() initializes the queue, allocating the necessary memory to accommodate a certain number of possible insertions and providing a pointer to the associated memory area.

Next, the InitializeTimer() function is called, passing it a reference to the timer instance. This function performs the necessary initializations (such as loading the configuration by searching for the timer ID) and then defines the timer's time, which is the period for which it should count, in this case 5ms. It is then specified that the timer should restart from 0 each time the count is finished, and finally the interrupt is enabled when 5ms is reached. The status resulting from the timer initialization is then returned for checking.

Finally, still within init\_Zynq, the SetupInterruptSystem() function is called, passing it a reference to both the interrupt controller and the timer, since the interrupt is associated with the timer. In this function, (note that the input parameters have types ScuGic, which is a general interrupt controller, and ScuTimer, which is the type of timer used), the initialization is performed using the configuration also searched for by ID here, and then the handler is registered by signing it in the exception table, so that it can be called when the interrupt occurs. Then the exception is connected to the physical device, the Gic, which will handle the interrupt, and finally the interrupt is enabled in the Gic.

#define INTC\_DEVICE\_ID XPAR\_SCUGIC\_SINGLE\_DEVICE\_ID

#define TIMER\_DEVICE\_ID XPAR\_SCUTIMER\_DEVICE\_ID

#define TIMER\_INTERRUPT\_ID XPAR\_SCUTIMER\_INTR

#define TS 0.05

#define MOTOR\_CHARACTERISTIC 3592

XScuGic InterruptController; *// Interrupt controller instance*

XScuTimer Timer; *// Timer instance*

*//XTime tEnd, tStart;*

PmodOLEDrgb oledrgb;

int tick\_count, direction;

CircularQueue q;

int speed, acceleration;

int last\_speed = 0;

int oldestTicks;

int delta;

void init\_Zynq(){

    init\_platform();

    init\_oled();

    init\_queue(&q);in

*// Initialize the timer*

    int status2 = InitializeTimer(&Timer);

    if (status2 != XST\_SUCCESS) {

        xil\_printf("Timer Initialization Failed\r\n");

        return XST\_FAILURE;

    }

*// Initialize the interrupt system*

    status2 = SetupInterruptSystem(&InterruptController, &Timer);

    if (status2 != XST\_SUCCESS) {

        xil\_printf("Interrupt System Setup Failed\r\n");

        return XST\_FAILURE;

    }

*// Start the timer*

    xil\_printf("INIT\n\r");

    XScuTimer\_Start(&Timer);

}

int InitializeTimer(XScuTimer \*TimerInstancePtr) { *//Si inizializza il Timer*

    int Status;

    XScuTimer\_Config \*ConfigPtr;

*// Initialize the timer driver so that it is ready to use*

    ConfigPtr = XScuTimer\_LookupConfig(TIMER\_DEVICE\_ID);

    if (ConfigPtr == NULL) {

        return XST\_FAILURE;

    }

    Status = XScuTimer\_CfgInitialize(TimerInstancePtr, ConfigPtr,

            ConfigPtr->BaseAddr);

    if (Status != XST\_SUCCESS) {

        return XST\_FAILURE;

    }

*// Load the timer with the value that corresponds to 5ms*

    XScuTimer\_LoadTimer(TimerInstancePtr, XPAR\_CPU\_CORTEXA9\_0\_CPU\_CLK\_FREQ\_HZ / 400);

*// Enable the Auto reload mode*

    XScuTimer\_EnableAutoReload(TimerInstancePtr);

*// Enable the timer interrupt*

    XScuTimer\_EnableInterrupt(TimerInstancePtr);

    return XST\_SUCCESS;

}

void init\_oled() {

   EnableCaches();

   OLEDrgb\_begin(&oledrgb, XPAR\_PMODOLEDRGB\_0\_AXI\_LITE\_GPIO\_BASEADDR,

   XPAR\_PMODOLEDRGB\_0\_AXI\_LITE\_SPI\_BASEADDR);

   OLEDrgb\_FillBackground(&oledrgb, 0, 0, 255); *//Scegli colore background*

   change\_fontcolor\_rgb(255,255,255); *//Scegli colore delle scritte*

   OLEDrgb\_SetCursor(&oledrgb, 1, 1);

   OLEDrgb\_PutString(&oledrgb, "SPEED: " );

   OLEDrgb\_SetCursor(&oledrgb, 1, 3);

   OLEDrgb\_PutString(&oledrgb, "ACCELER: " );

   OLEDrgb\_SetCursor(&oledrgb, 1, 5);

   OLEDrgb\_PutString(&oledrgb, "DIRECT: " );

}

int SetupInterruptSystem(XScuGic \*IntcInstancePtr, XScuTimer \*TimerInstancePtr) { *//Funzione per creare la callback*

    int Status;

*// Initialize the interrupt controller driver*

    XScuGic\_Config \*IntcConfig;

    IntcConfig = XScuGic\_LookupConfig(INTC\_DEVICE\_ID);

    Status = XScuGic\_CfgInitialize(IntcInstancePtr, IntcConfig,

            IntcConfig->CpuBaseAddress);

    if (Status != XST\_SUCCESS) {

        return XST\_FAILURE;

    }

*// Initialize the exception table and register the interrupt controller handler with the exception table*

    Xil\_ExceptionInit();

    Xil\_ExceptionRegisterHandler(XIL\_EXCEPTION\_ID\_INT,

            (Xil\_ExceptionHandler) XScuGic\_InterruptHandler, IntcInstancePtr);

*// Connect the device driver handler that will be called when an interrupt for the device occurs*

    Status = XScuGic\_Connect(IntcInstancePtr, TIMER\_INTERRUPT\_ID,

            (Xil\_InterruptHandler) TimerIntrHandler, (void \*) TimerInstancePtr);

    if (Status != XST\_SUCCESS) {

        return XST\_FAILURE;

    }

*// Enable the interrupt for the device*

    XScuGic\_Enable(IntcInstancePtr, TIMER\_INTERRUPT\_ID);

*// Enable interrupts in the ARM processor*

    Xil\_ExceptionEnable();

    return XST\_SUCCESS;

}

##### GPIO and GPIOPS DATA DIRECTION

This section defines the data direction for the desired pin. Does not require further explanation than saying that to know which data to use it is necessary to refer to the documentation of the driver. Note that the drivers have different conventions for input and output definition.

// set data direction for GPIO

 XGpio\_SetDataDirection(&pl\_tick\_count, 1, 1); // 0-> output, 1-> input.

 XGpio\_SetDataDirection(&pl\_direction, 1, 1); // 0-> output, 1-> input.

The function states that the only bank (since one bank drive up to 32 bits) present in the GPIO interface, bank 1, is configured as input.

##### TIMER INTERRUPT HANDLER

First and foremost, it is important to clarify that the commented code is used, if necessary, for debugging purposes to verify that the actual execution cycle duration is 5ms and is not delayed by resource-intensive operations (using a single CPU, placing complex operations in the callback which cannot be parallelized and consequently require more time). Within the remainder of the function, initially, the system reads the values of tick\_count and direction from the logical part of the platform. Subsequently, the delta is computed as the difference between the current tick count and that recorded 50ms earlier. Following this, the current tick\_count values are inserted into the queue, resulting in the removal of oldestTicks from the circular queue. The speed is then calculated based on the physical characteristics of the motor, as well as the acceleration, with last\_speed updated to compute acceleration in the subsequent cycle. Finally, the handler is released upon completion of these operations. At line 135, a printout occurs, which is subsequently captured by a MATLAB script over UART to graphically depict the real-time trends of the measurements.

void TimerIntrHandler(void \*CallBackRef) { *//Effettiva CallBack*

*//XTime\_GetTime(&tStart);*

    tick\_count = XGpio\_DiscreteRead(&pl\_tick\_count, 1);

    direction = XGpio\_DiscreteRead(&pl\_direction, 1);

    oldestTicks = getOldestElement(&q);

    delta = tick\_count - oldestTicks;

    enqueue(&q, tick\_count);

    speed = delta \* 60/(MOTOR\_CHARACTERISTIC\*TS);

    acceleration = (speed - last\_speed) \* 200;

    last\_speed = speed;

    xil\_printf("%d, %d\n\r", (direction==1) ? speed : -speed , acceleration);

    XScuTimer\_ClearInterruptStatus((XScuTimer \* )CallBackRef); *// Clear the interrupt status*

*//printf("Elapsed: %.5lf ms\r\n", ((double)(tStart-tEnd) / (double)COUNTS\_PER\_SECOND)\*1000);*

*//xil\_printf("Speed: %d RPM, Acceleration: %d RPM/s\n", (direction==1) ? speed : -speed , acceleration);*

*//XTime\_GetTime(&tEnd);*

}

##### INFINITE LOOP

The following are just simple convenience functions to interface with the display’s API in a more familiard manner, like using RGB instead of BGR to select the desired color. Other than that, some magic is done so to have ready-to-use functions for the colouring of the background and the printing of relevant data such as speed, direction and acceleration.

void OledCleanup() {

   DisableCaches();

}

void print\_speed(char \*\*b){

    change\_fontcolor\_rgb(255,255,255);

    sprintf(\*b, "%d    ", (direction==1) ? speed : -speed);

    OLEDrgb\_SetCursor(&oledrgb, 1, 2);

    if(speed == 0){

        change\_fontcolor\_rgb(255,0,0);

        OLEDrgb\_PutString(&oledrgb, "ZERO");

    }

    else

        OLEDrgb\_PutString(&oledrgb, \*b);

}

void print\_acceleration(char \*\*b){

    change\_fontcolor\_rgb(255,255,255);

    sprintf(\*b, "%d     ", acceleration);

    OLEDrgb\_SetCursor(&oledrgb, 1, 4);

    if(acceleration == 0){

        change\_fontcolor\_rgb(255,0,0);

        OLEDrgb\_PutString(&oledrgb, "ZERO");

    }

    else

        OLEDrgb\_PutString(&oledrgb, \*b);

}

void print\_direction(char \*\*b){

    change\_fontcolor\_rgb(255,255,255);

    OLEDrgb\_SetCursor(&oledrgb, 1, 6);

    if(direction == 1 && speed>5){

        OLEDrgb\_PutString(&oledrgb, "----->      ");

    }

    else if (direction == 0 && speed>5){

        OLEDrgb\_PutString(&oledrgb, "<-----      ");

    }

    else{

        change\_fontcolor\_rgb(255,0,0);

        OLEDrgb\_PutString(&oledrgb, "STOP    ");

    }

}

void EnableCaches() {

#ifdef \_MICROBLAZE\_

#ifdef XPAR\_MICROBLAZE\_USE\_ICACHE

   Xil\_ICacheEnable();

#endif

#ifdef XPAR\_MICROBLAZE\_USE\_DCACHE

   Xil\_DCacheEnable();

#endif

#endif

}

void DisableCaches() {

#ifdef \_MICROBLAZE\_

#ifdef XPAR\_MICROBLAZE\_USE\_DCACHE

   Xil\_DCacheDisable();

#endif

#ifdef XPAR\_MICROBLAZE\_USE\_ICACHE

   Xil\_ICacheDisable();

#endif

#endif

}

void change\_fontcolor\_rgb(u8 R, u8 G, u8 B){

    OLEDrgb\_SetFontColor(&oledrgb, OLEDrgb\_BuildRGB(B, R, G));

}

##### MAIN AND INFINITE LOOP

The main portion of the program is simply the calling of the central inizialitation function, and then setting the right color of the font before starting the while that will print all the information. An important digression must be made here, however.

It is with this while (1) loop that we managed to resolve the issue of excessive execution time: by placing the heavier instructions within the while (1) loop, they are executed only after giving priority to the interrupt handler. Consequently, the control loop manages, in a pseudo-parallel fashion, both the operations within the non-interruptible interrupt handler and those within the interruptible while loop, which can be suspended when it is time to return to executing the callback every 5 ms.

Therefore, the solution lies in the fact that, when the print instructions were placed in the callback, they could not be interrupted and their lengthy execution caused the 5 ms limit to be exceeded. By placing them in the while loop instead, they can be interrupted and effectively are interrupted to accommodate the interrupt handler. Thus, it is as if the interrupt context and the while (1) loop operate in parallel.

int main() {

    init\_Zynq();

    char b[6];

    change\_fontcolor\_rgb(255,255,255); *//Scegli colore numeri*

    while (1) {

        print\_speed(&b);

        print\_acceleration(&b);

        print\_direction(&b);

    }

    return XST\_SUCCESS;

}

## Experimental Test of the system

After writing the code the reader can build the application, connect the board (optionally open a serial terminal to read what the application tells you) and run the application to see the display show all the useful info about the motor; clearly, one has to power the motor beforehand. Please note that when the speed and acceleration are zero, they are shown in red, otherwise the information has white as the font color.

### Matlab code for real time graph plot

A screenshot of a computer

Description automatically generated

It is also possible to real time plot those values in two different graphs, by using Matlab and the serial connection, to visually see the changes of the speed and the acceleration.

Below there’s the matlab code.

function data=readFromCOM()

data = [];

device = serialport("COM17",115200);

FS = stoploop({'Click OK to stop acquisition'}) ;

disp('> Compile and run the code. If already running, compile and run the code from scratch');

%string = '';

rowNum = 1;

error = false;

time\_step = 0.005;  % Interval between values (5 ms)

total\_time = 0;     % Total elapsed time

plot\_window = 2;   % Duration of the visualization window in seconds

speed\_values = [];   % Array to store speed data

acc\_values = [];     % Array to store acceleration data

timestamps = [];    % Array to store timestamps

hFig = figure;

screenSize = get(0,'ScreenSize');

set(hFig, 'Units', 'pixels', 'OuterPosition', screenSize);

subplot(2,1,1);

h = plot(nan, nan);

xlabel('Time (s)');

ylabel('Speed (RPM)');

title('Real-Time Speed Plot');

ylim([-150, 150]);

grid on;

hold on;

subplot(2, 1, 2);

h2 = plot(nan, nan);

xlabel('Time (s)');

ylabel('Acceleration (RPM/s)');

title('Real-Time Acceleration Plot');

ylim([-3000, 3000]);

grid on;

hold on;

%wait for INIT

while(~FS.Stop())

    newLineString = readline(device);

    if(isequal(strtrim(newLineString),"INIT"))

        break;

    end

end

tempo=0;

while(~FS.Stop())

    newLineString = readline(device);

    if(not(isequal(strtrim(newLineString),"INIT")))

        newStr = split(newLineString,', ');

        for j=1:length(newStr)

            numValue = str2double(strtrim(newStr(j)));

            if isnan(numValue)

               error = true;

            end

            data(rowNum,j) = numValue;

        end

        if error

            disp(strcat('Expected numeric value, but read the following value: ',strtrim(newStr(j))));

            break;

        end

    total\_time = total\_time + time\_step;

    speed\_values = [speed\_values, data(rowNum,1)];

    acc\_values = [acc\_values, data(rowNum,2)];

    timestamps = [timestamps, total\_time];

    if total\_time <= plot\_window

        x\_min = 0;

        x\_max = plot\_window;

    else

        x\_min = total\_time - plot\_window;

        x\_max = total\_time;

    end

    tempo = tempo + 1;

    if(tempo >= 20)

        % Aggiorna il plot

        subplot(2,1,1);

        set(h, 'XData', timestamps, 'YData', speed\_values);

        xlim([x\_min, x\_max]);

        ylim([-150, 150]);

        subplot(2,1,2);

        set(h2, 'XData', timestamps, 'YData', acc\_values);

        xlim([x\_min, x\_max]);

        ylim([-3000, 3000]);

        drawnow;

        tempo = 0;

    end

    end

end

FS.Clear() ;

clear FS ;

clear device;

end

### Step by step explanation of the code

function data=readFromCOM()

data = [];

device = serialport("COM17",115200);

FS = stoploop({'Click OK to stop acquisition'}) ;

disp('> Compile and run the code. If already running, compile and run the code from scratch');

%string = '';

This first part contains the initialization used for the connection to the COM Serial. Make sure that the COM Value is the right one. To see which COM is used, go to Start Menu -> Control Panel -> Device Manager -> Ports (COM & LPT) and read the right COM.

time\_step = 0.005;  % Interval between values (5 ms)

total\_time = 0;     % Total elapsed time

plot\_window = 2;    % Duration of the visualization window in seconds

speed\_values = [];  % Array to store speed data

acc\_values = [];    % Array to store acceleration data

timestamps = [];    % Array to store timestamps

These variables manage the time step for data acquisition, the total elapsed time, the duration of the plot window, and arrays to store speed, acceleration, and timestamps respectively.

hFig = figure;

screenSize = get(0,'ScreenSize');

set(hFig, 'Units', 'pixels', 'OuterPosition', screenSize);

subplot(2,1,1);

h = plot(nan, nan);

xlabel('Time (s)');

ylabel('Speed (RPM)');

title('Real-Time Speed Plot');

ylim([-150, 150]);

grid on;

hold on;

subplot(2, 1, 2);

h2 = plot(nan, nan);

xlabel('Time (s)');

ylabel('Acceleration (RPM/s)');

title('Real-Time Acceleration Plot');

ylim([-3000, 3000]);

grid on;

hold on;

Those lines are used for setting the graph parameter, such as labels and limits.

while(~FS.Stop())

    newLineString = readline(device);

    if(isequal(strtrim(newLineString),"INIT"))

        break;

    end

end

This loop waits for the device to send an "INIT" message, indicating readiness. The loop continues until FS.Stop() is true or the "INIT" message is received.

tempo=0;

while(~FS.Stop())

    newLineString = readline(device);

    if(not(isequal(strtrim(newLineString),"INIT")))

        newStr = split(newLineString,', ');

        for j=1:length(newStr)

            numValue = str2double(strtrim(newStr(j)));

            if isnan(numValue)

               error = true;

            end

            data(rowNum,j) = numValue;

        end

        if error

            disp(strcat('Expected numeric value, but read the following value: ',strtrim(newStr(j))));

            break;

        end

In this part of the code, Matlab reads each line string from the FPGA. There are also some controls like if(not(isequal(strtrim(newLineString),"INIT"))) and if isnan(numValue) error = true;.

total\_time = total\_time + time\_step;

speed\_values = [speed\_values, data(rowNum,1)];

acc\_values = [acc\_values, data(rowNum,2)];

timestamps = [timestamps, total\_time];

if total\_time <= plot\_window

    x\_min = 0;

    x\_max = plot\_window;

else

    x\_min = total\_time - plot\_window;

    x\_max = total\_time;

end

tempo = tempo + 1;

if(tempo >= 20)

    subplot(2,1,1);

    set(h, 'XData', timestamps, 'YData', speed\_values);

    xlim([x\_min, x\_max]);

    ylim([-150, 150]);

    subplot(2,1,2);

    set(h2, 'XData', timestamps, 'YData', acc\_values);

    xlim([x\_min, x\_max]);

    ylim([-3000, 3000]);

    drawnow;

    tempo = 0;

end

This code firstly computes the total time elapsed, then gets the speed and acceleration values from the data array. Then checks if the relative time elapsed (tempo) is greather then 20\*0.005ms (100ms); if so, the matlab code plots the new values inside the graphs and shiftes the windows of 100ms.