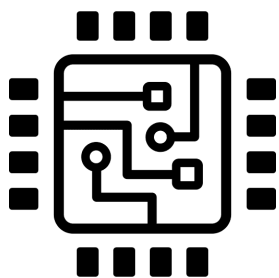

Cora Z7-10 HDL project

Detailed guide on Vivado project creation

Dmitrii Matafonov



Contents

Introduction	3
Tools for this guide	3
Project creation guide	4
Cora Z7, new project	4
Creating the project	7
Creating base design	7
Custom AXI IP creation	15
Adding debug nodes	29
Exporting the design for petalinux tools to use	30
Appendix A (simple_hardware_timer_v0.1.vhd)	32
Appendix B (simple_hardware_timer_v0.1_S00_AXI.vhd)	36

Introduction

This document is intended to give the reader a quick, but detailed guide on how to create a Vivado project specifically for Cora Z7-10 development board (but generally for any base).

This guide is rich for screenshots and remarks for what one may want to consider during custom project development.

Tools for this guide

One would need Vivado 2021.2¹ with USB cable drivers installed.

This guide is not specific to the PC's operating system.

¹<https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/archive.html>

Project creation guide

In order to streamline the process of making oneself familiar with the tool, the guide will give MOSTLY step-by-step instructions. But it doesn't mean that it provides detailed actions for literally each button click. Most probably, if something is missing, it is intended that it is very easy and intuitive to guess or part of the base knowledge of working with IDEs such as this one. Also, it is implied that the reader of this document is generally aware of the regular FPGA design flow and their basic principles.

Cora Z7, new project

After opening Vivado click on the “New project” button to call the screen that welcomes us in this endeavour.

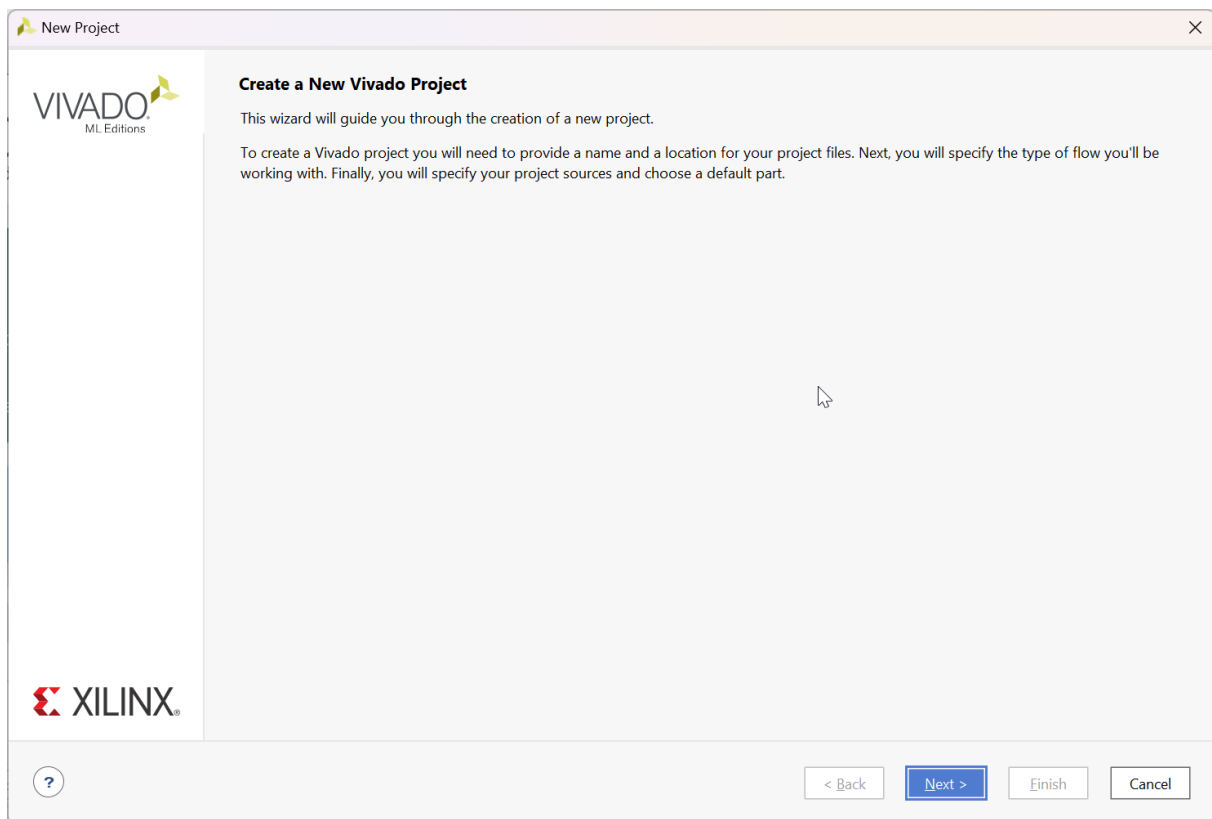
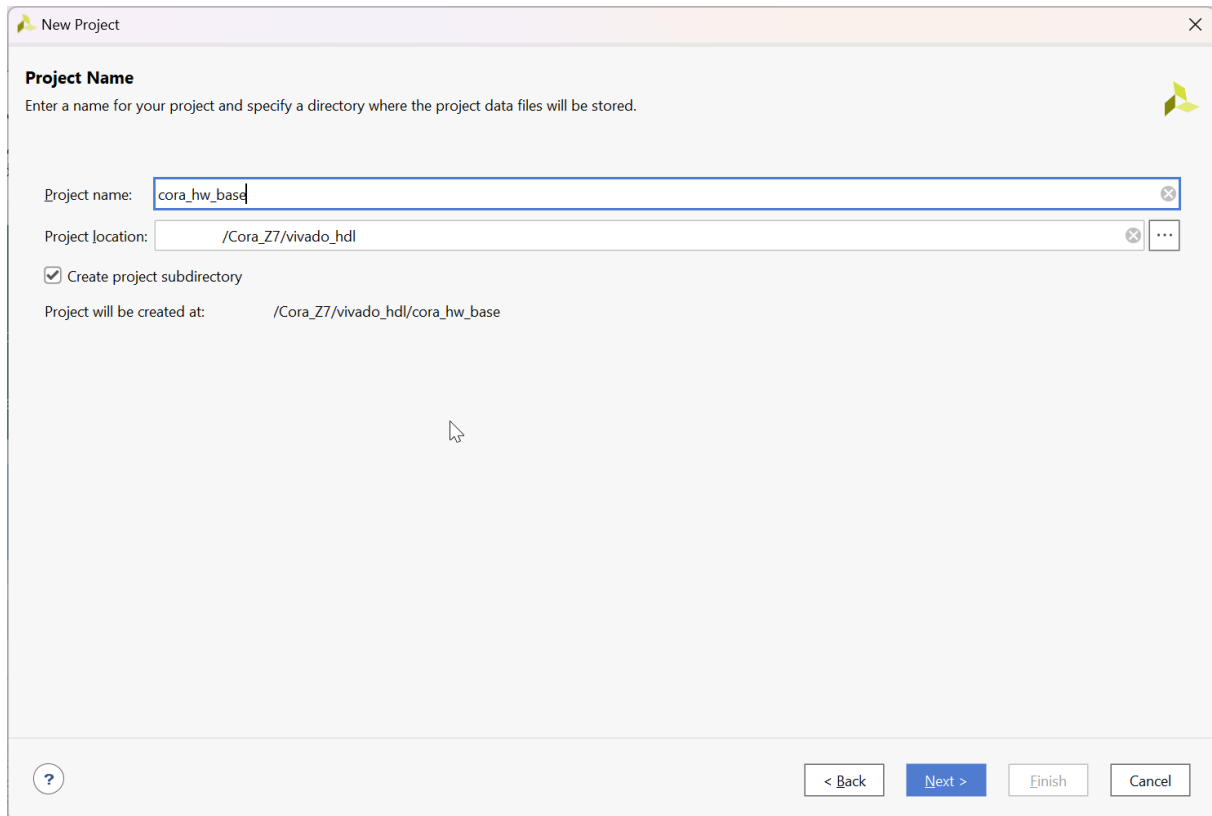


Figure 1: New project welcome page

Let's name the project `cora_hw_base` and specify a location. The screenshots contain the locations of this repository. If one creates a from-scratch project, maybe they'd want to specify another location and/or name.

Cora Z7-10 HDL project



New Project

Project Name
Enter a name for your project and specify a directory where the project data files will be stored.

Project name:

Project location:

☒ Create project subdirectory

Project will be created at:

Figure 2: Project name

After clicking next, let's choose these options since we have no pre-made sources.

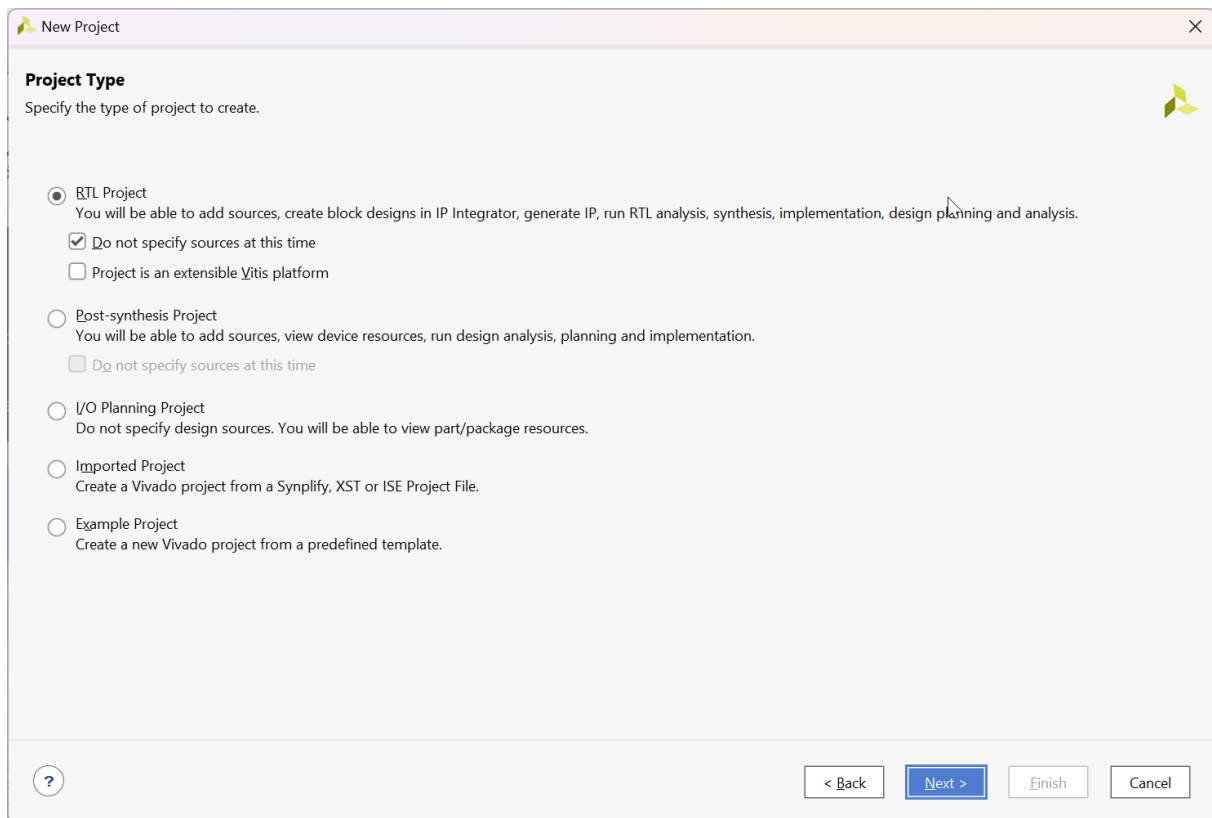


Figure 3: Project type

Clicking **Next** will guide to the part choosing screen.

If one is creating a custom solution and has only one SoC in mind, but no board, I'd recommend going through the process of creating a basic project BEFORE designing a schematic and layout. Creating a basic project with at least pins assigned will eliminate a lot of failure points in the schematic design in the future.

In the scope of this guide, everything is made simpler with the predefined board support packages. They contain presets with correct constants, project settings and pins. We will briefly touch what could be different when creating custom board.

So, switching to **Boards** tab provides an option to choose a preset for a specific board. The full list may not be available by default, clicking **Refresh** is necessary.

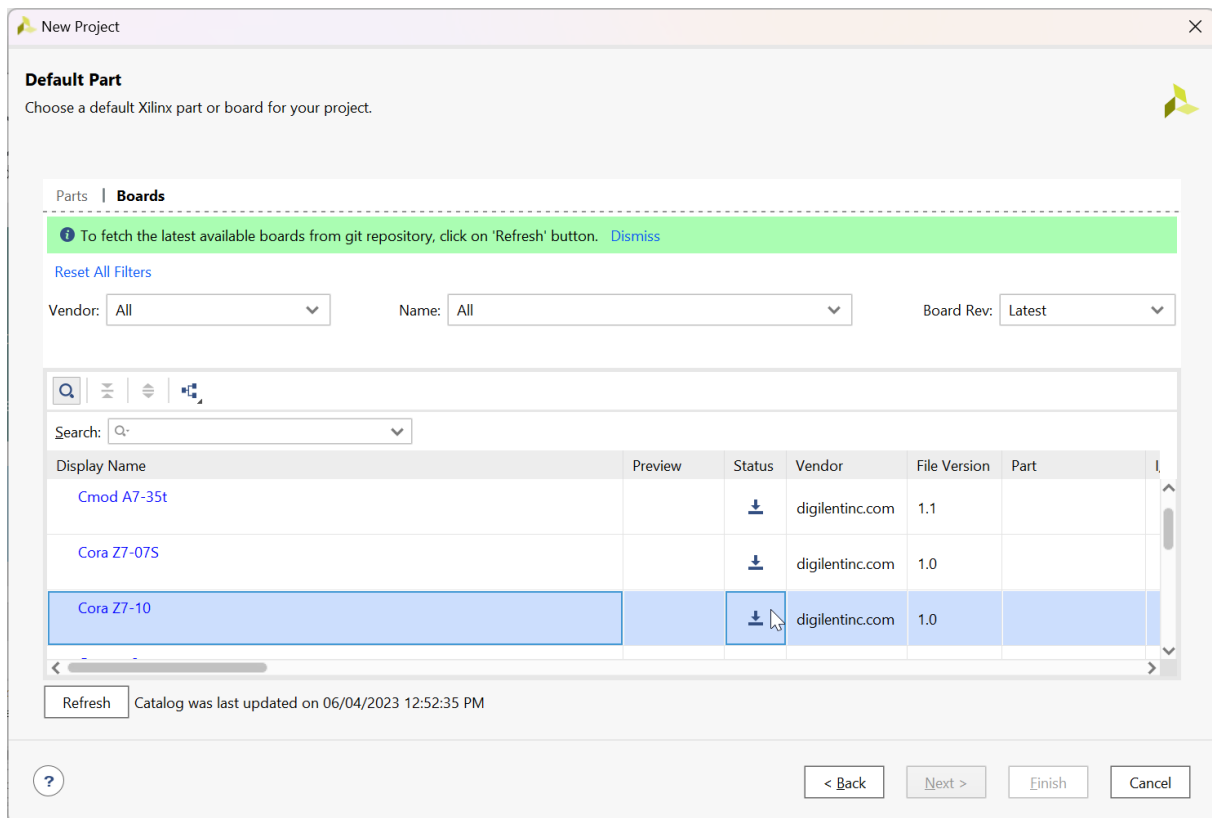


Figure 4: Boards screen

Click on the download button in **Status** column in order to download the support package. After downloading, choose **Cora Z7-10** (or another one that fits one's needs) and click next.

After that the project creation is over and clicking **Finish** will switch the layout to the default Vivado layout.

Creating the project

Creating base design

Note: the following step regarding preferred language will affect the guide until the end. I'm going to use VHDL as a matter of preference, but the reader is free to skip the following change and stick to Verilog

Click on **Settings** on the left sidebar (Flow manager) and choose **VHDL** in the first screen you see. Click OK afterwards. This setting affects which language is used for Vivado-generated templates and auto-generated code.

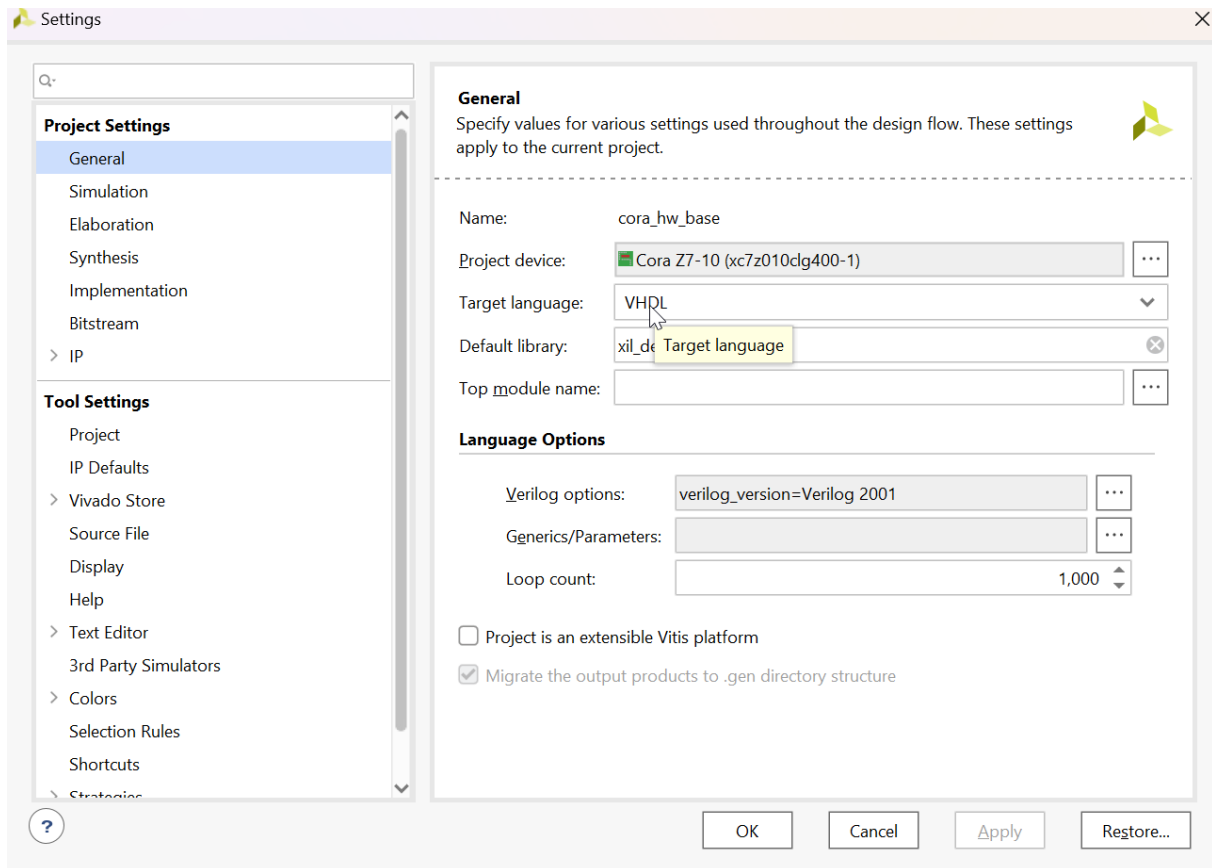


Figure 5: Target language change

It is a matter of preference and one could use Vivado in text-only design mode, I would suggest sticking to the block design flow. It gives the ease of perception and ability to quickly change or add modules to the project. This guide uses block-design flow and does not cover the text-only design input.

Click **Create block design** on the left sidebar.

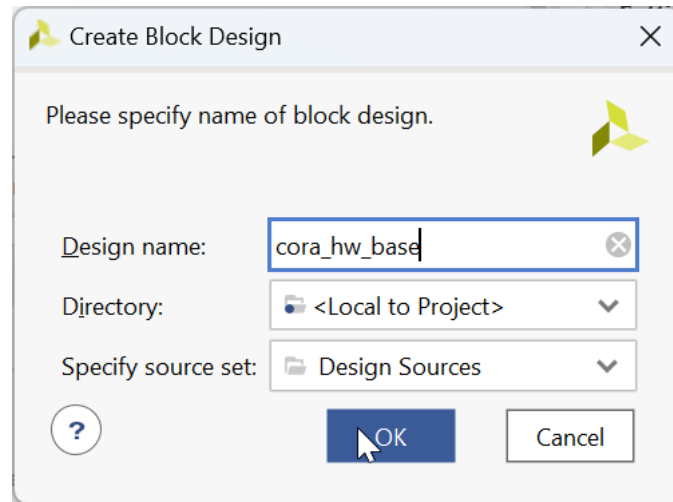


Figure 6: Creation and naming of the block design

After that on the blank page hit + sign to add: - Processing system - Processing system reset - AXI interconnect - 2 AXI GPIO blocks

Do not connect anything yet

Configure an AXI GPIO block to work with LEDs: 1. Rename one to `leds` (Block properties window left to the block design) 2. Double click on it to edit the properties and choose Board interface `rgb_leds`

It connects the pins automatically to the correct locations thanks to the predefined board preferences. In case you have a custom board, you'll need to cover the connections in an *.xdc constraints file with correct assignments. I will briefly touch this part further in the document.

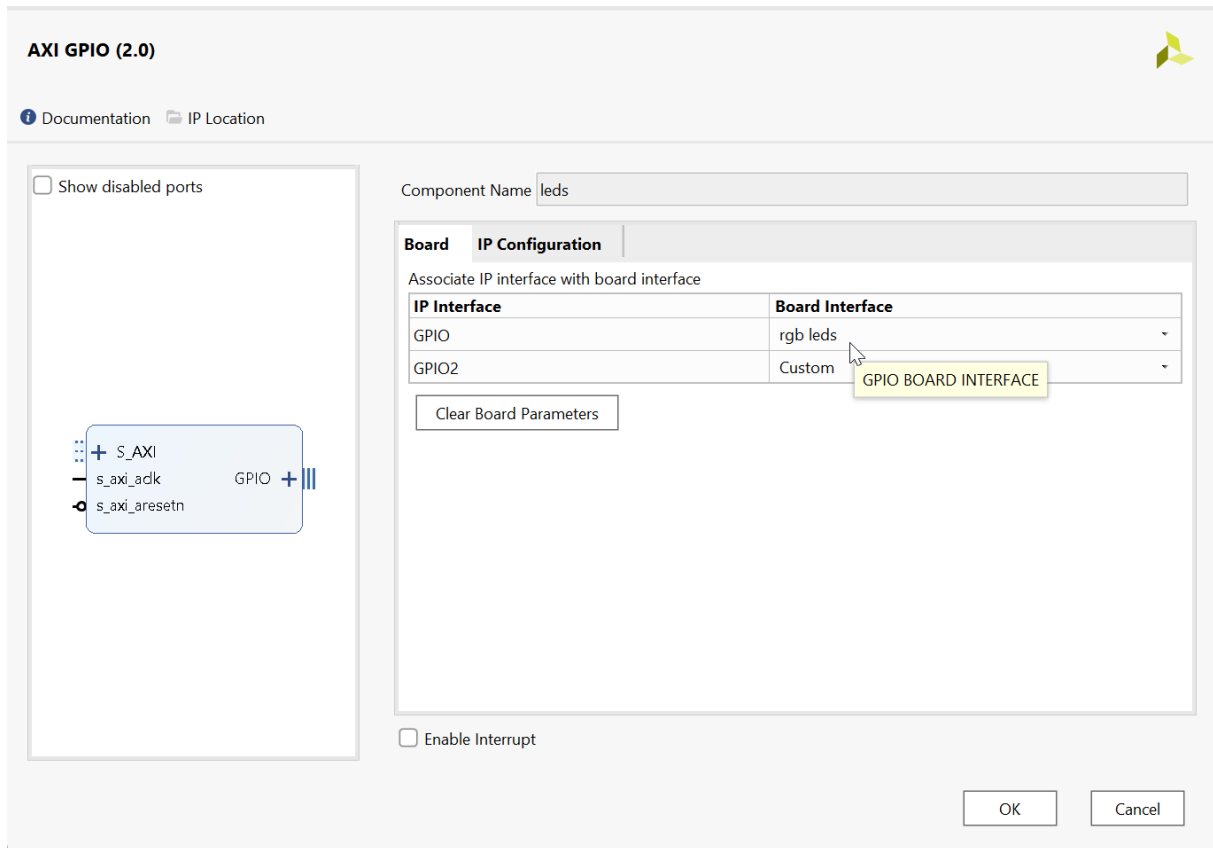


Figure 7: Configuring an AXI GPIO for LEDs

Configure the other AXI GPIO block to work with buttons:

1. Name the block to `btns`
2. Configure to use buttons interface in a similar fashion.

You can make the changes manually for the GPIOs which are self-explanatory.

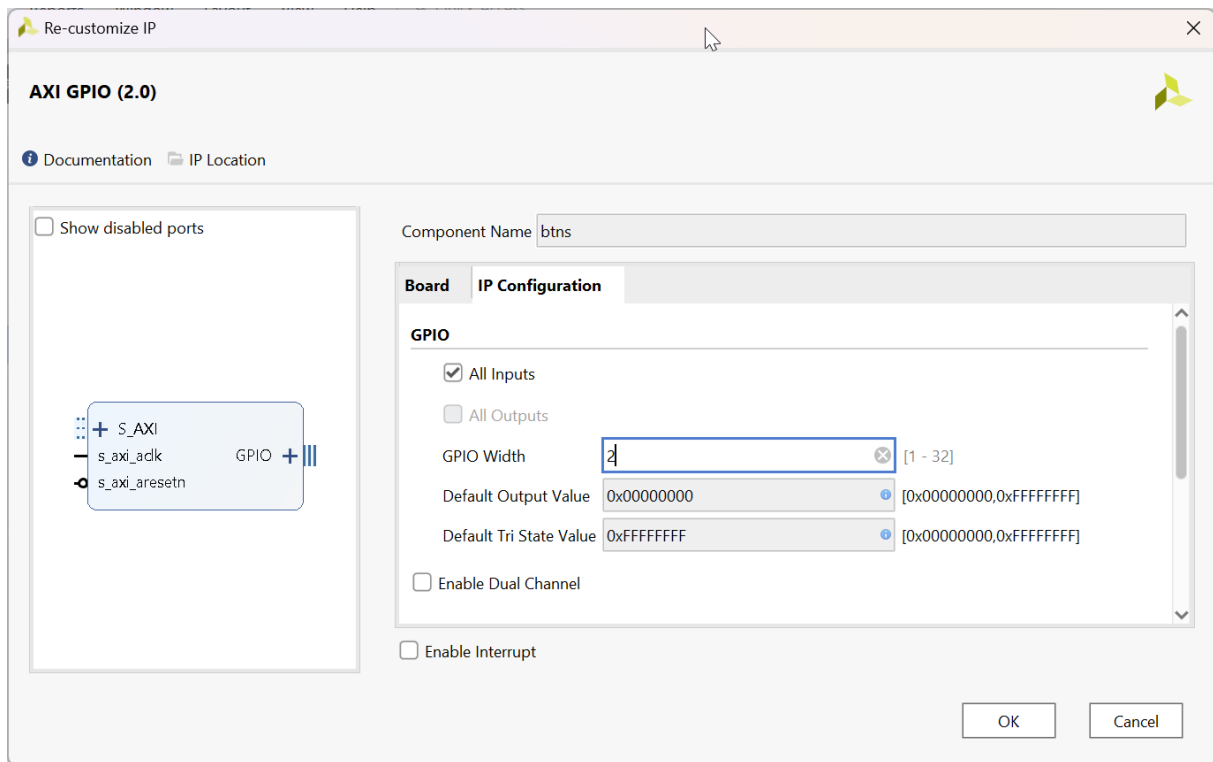


Figure 8: Settings for buttons

After that, the block design window would greet the user with suggested helpers that would automatically connect the blocks. It quickens the design, but sometimes it may not be optimal and it could be for the best to connect the blocks manually.

Designer Assistance available. [Run Block Automation](#) [Run Connection Automation](#)

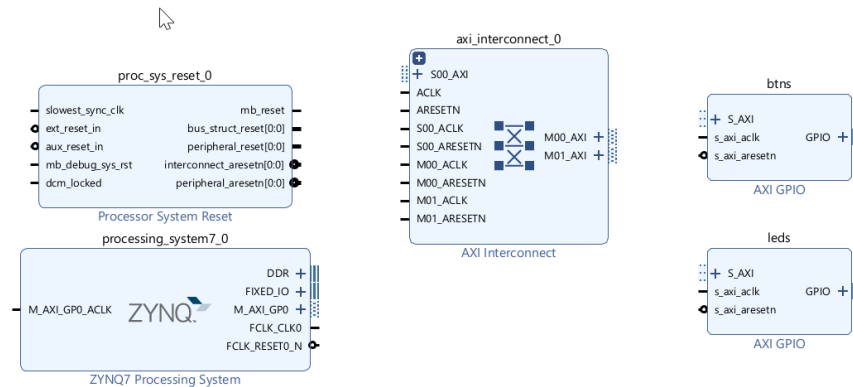


Figure 9: Automation helpers in block design

Clicking on **Run Block Automation** will apply presets from the board support package and configure the processing system accordingly to the board's real connections.

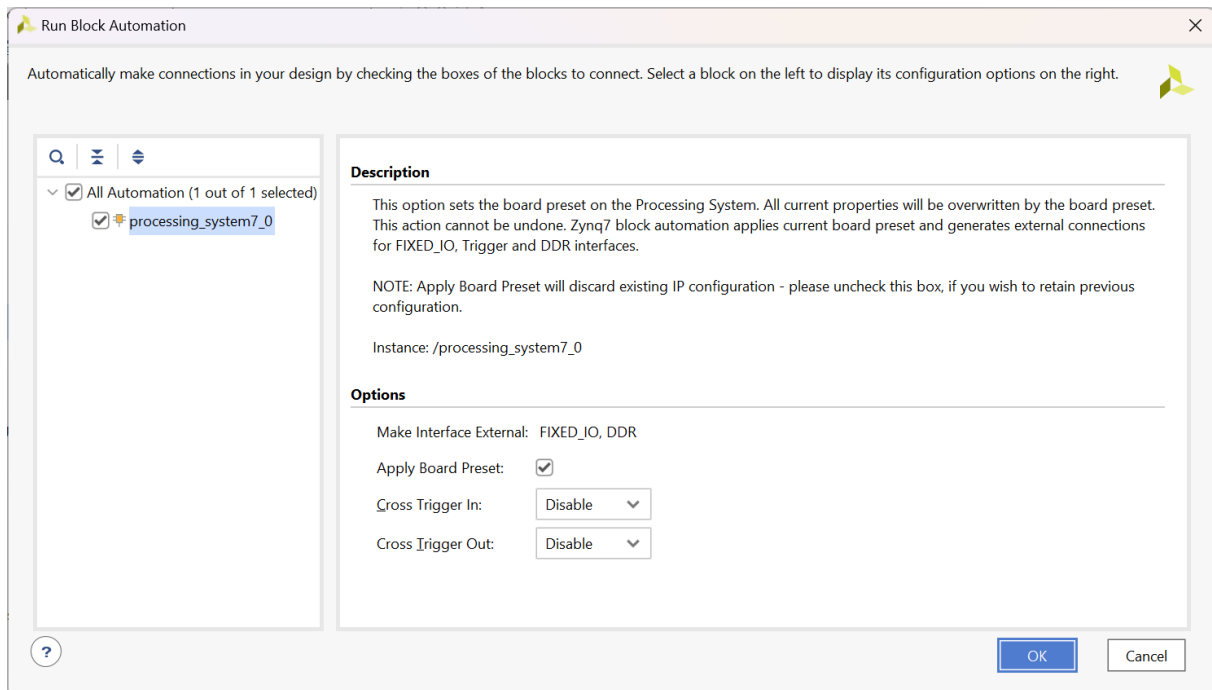


Figure 10: Block automation window

Note: Running Automations for custom board will not give you automatic connections of the PS interfaces. You need to specifically set them manually in the Processing System's settings according to your design, like USB, Ethernet PS connections etc. They are self-explanatory.

To check that the settings were properly applied, one can go to one of the most important parts of the Processing system's settings - DDR traces.

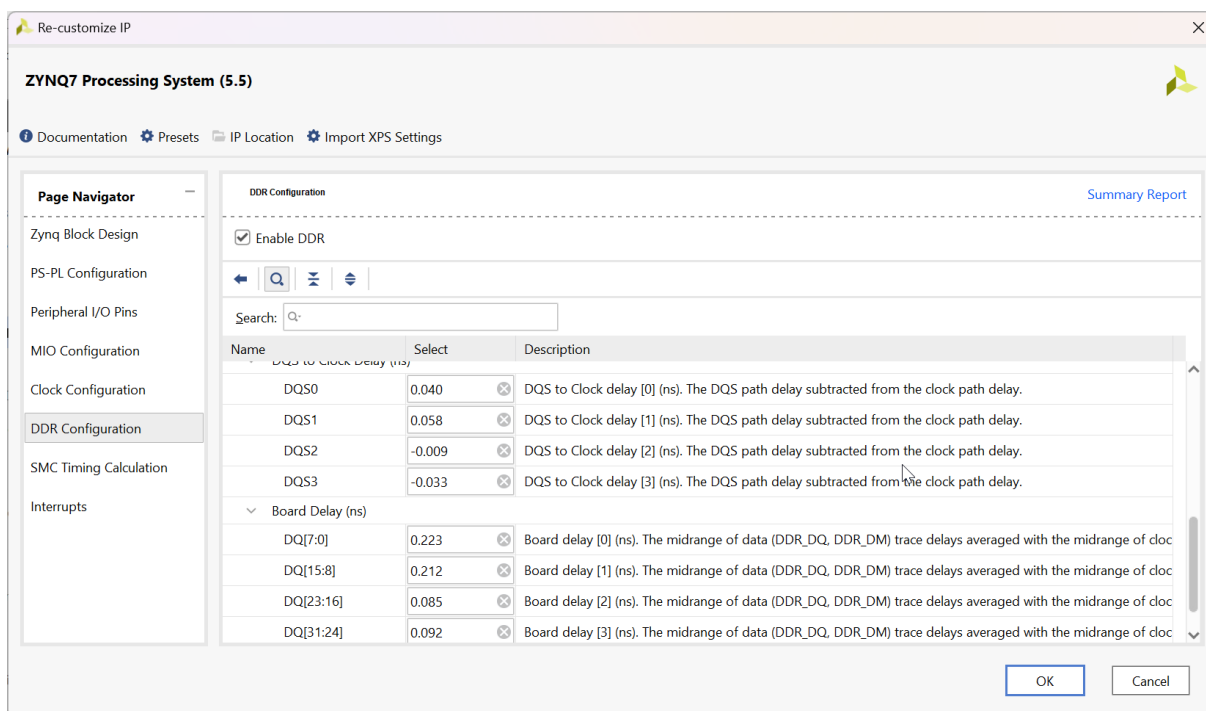


Figure 11: DDR traces' settings, one of the most important parts for custom boards

As it can be noted, the settings are already there and were provided by the developers.

Custom boards: there is a calculator that can be called i one chooses **User input** in these DDR configuration settings. It would give the user an option to provide signal propagation speeds (can be taken from PCB factory's documentation) and traces' lengths (from PCB design). The tool would automatically calculate the values and save them there. It is very important to correctly specify these parameters or the user may end up with suboptimal performance for DDR chips.

Clicking on **Run Connection Automation** will result in the blocks being properly connected. Examine the result to ensure the connections are properly made.

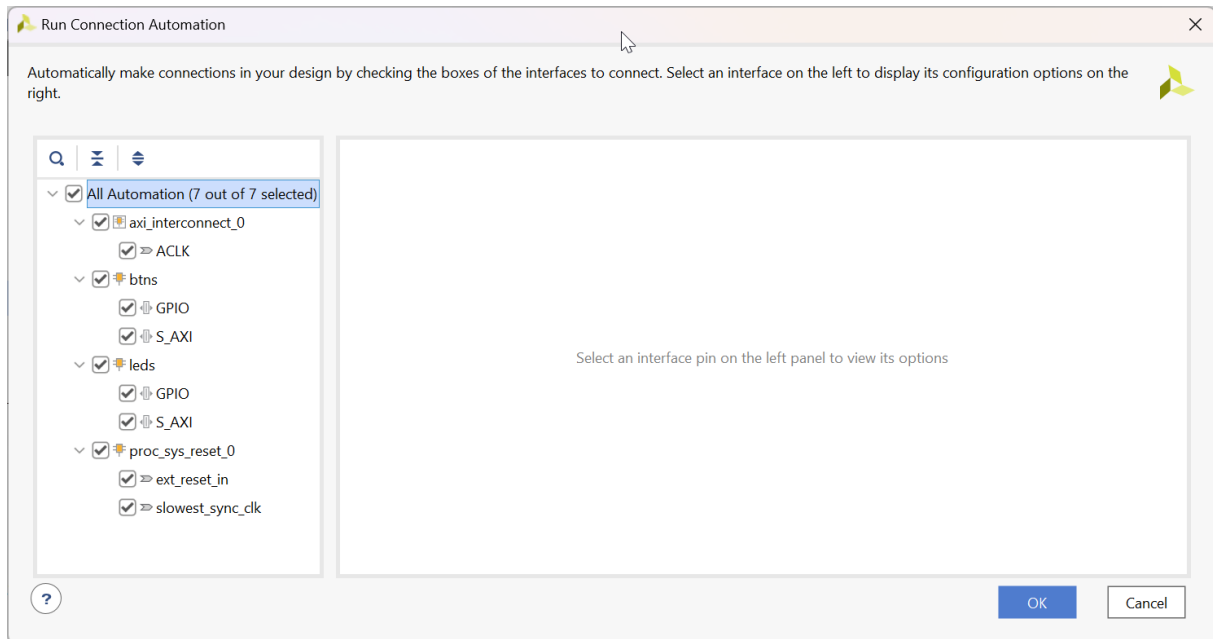


Figure 12: Connection automation window

After that, the settings for the base blocks are applied and connections are made. The result should be similar to the following.

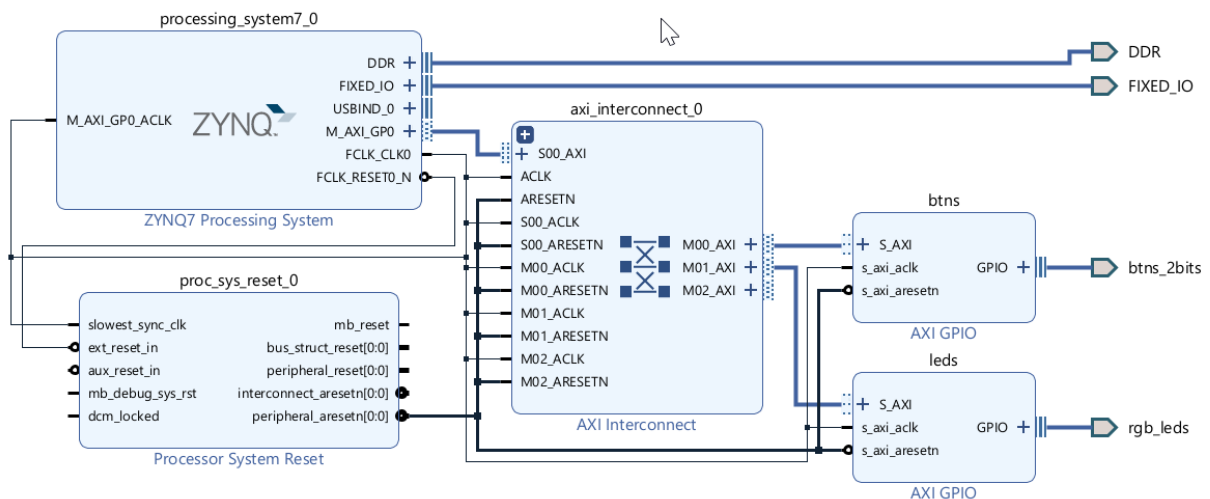


Figure 13: Base block design result

To make it more clear to the reviewer, let's hide the following service blocks into a subhierarchy called

Processing system:

1. Processing system
2. Processing system Reset
3. AXI interconnect

In order to do that, choose multiple blocks with `ctrl` in context menu choose `Create hierarchy`. Name those 3 blocks

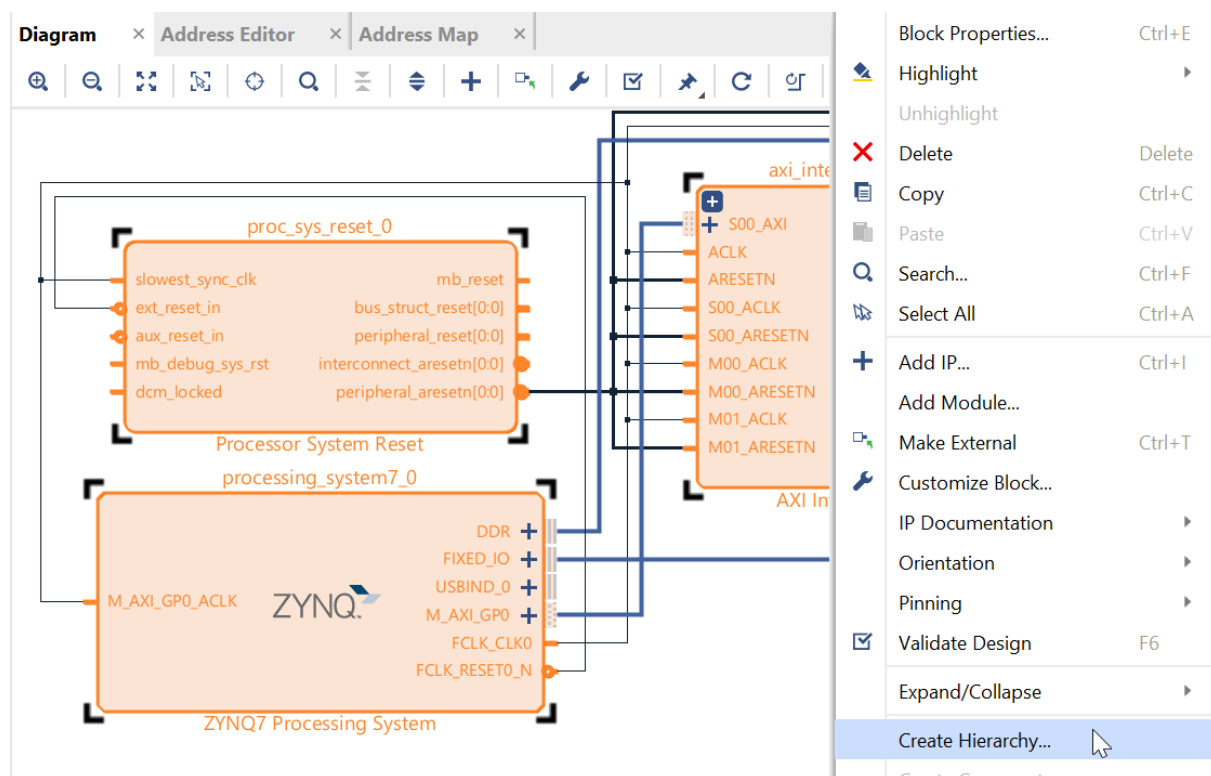


Figure 14: Create hierarchy action

After that we can finish with customizing the Xilinx-provided blocks that are going to be the core of our design and transition to Custom IP creation.

Custom AXI IP creation

Most probably the designers are going to use the SoCs to create custom logic that would provide some specific functionality they need, that is usually not achievable by more commons MCUs.

In the scope of this flow we are going to create a very simple custom IP with intention to use it in our Linux system later. The approach that this project proposes in general may be useful to developers

who need to combine the ease of implementation and linux interrupts for custom FPGA modules.

In order to create a custom IP block that would be used in our block design later, click on **Tools** menu and **Create and Package New IP**

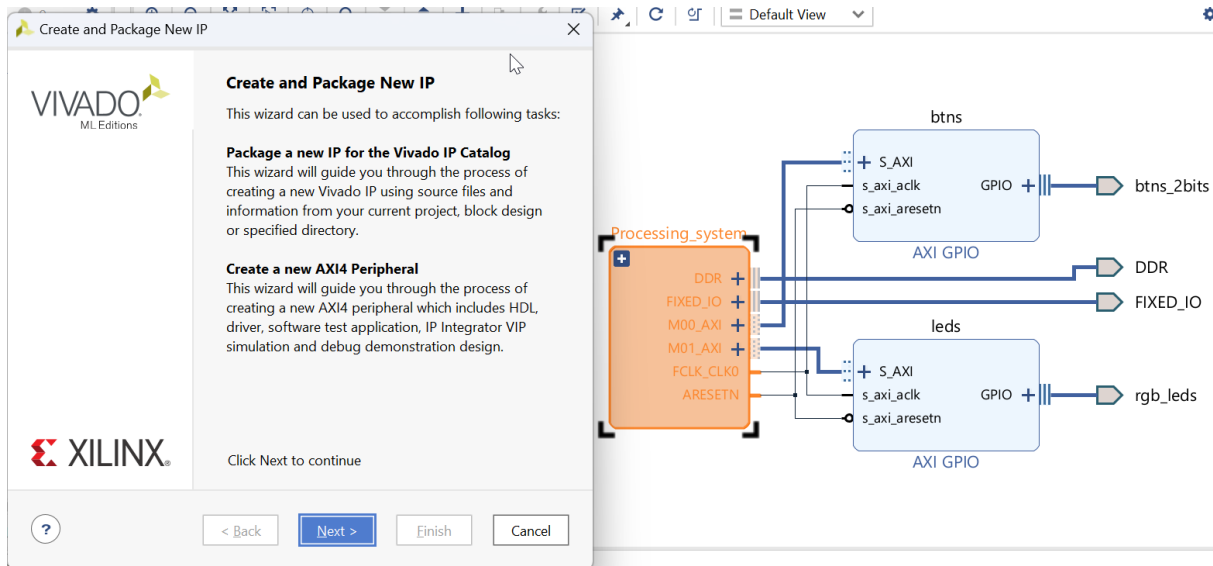


Figure 15: Create and Package New IP welcome screen

One the next step choose “Create a new AXI4 peripheral”. It would generate a project template with ready-to-use AXI slave registers. Designers can easily connect custom logic to these registers to connect their FPGA functionality to the system’s AXI4 bus and access that by simply interacting with those meory-mapped registers.

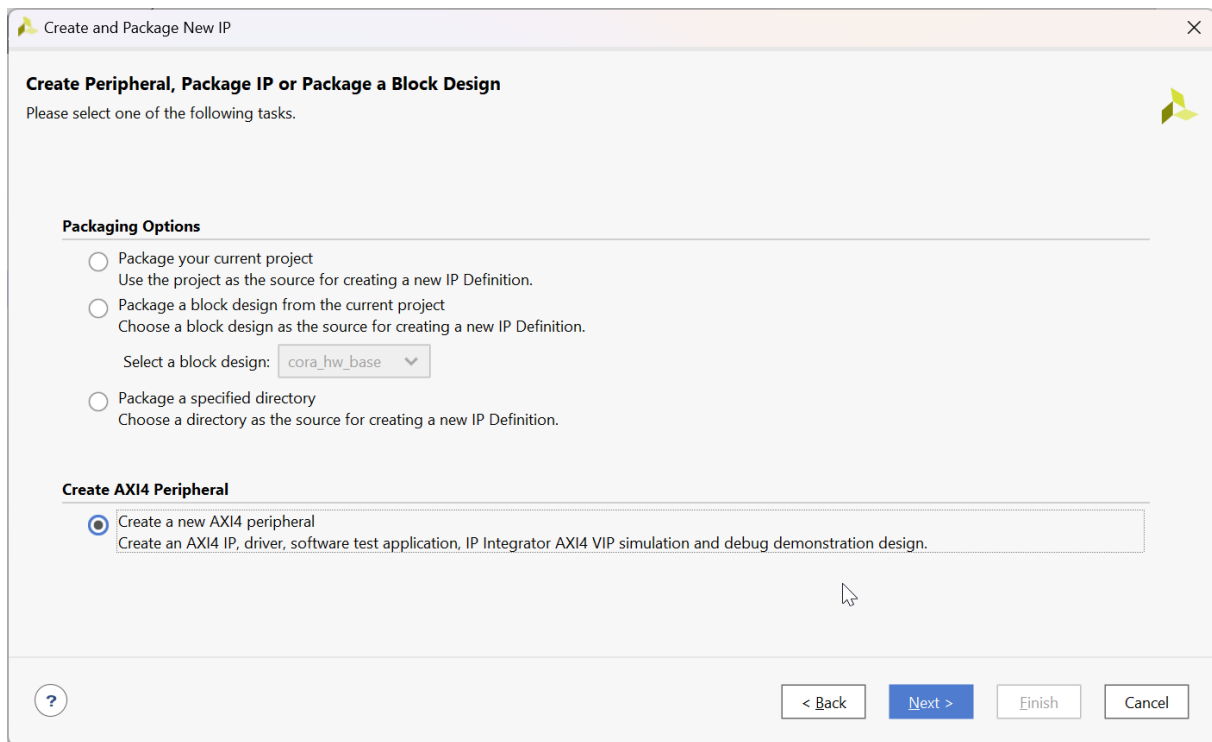
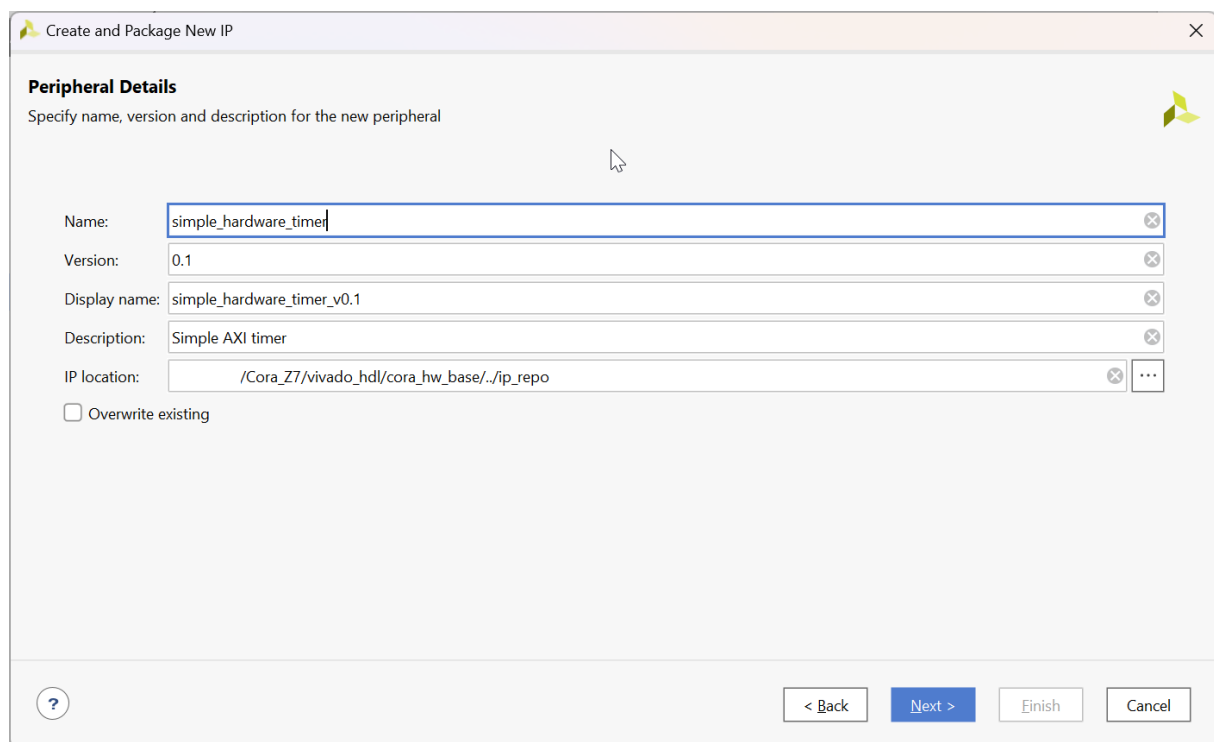


Figure 16: Create new AXI4 peripheral

Let's name the IP block accordingly to the functionality it is going to provide.



Create and Package New IP

Peripheral Details
Specify name, version and description for the new peripheral

Name:

Version:

Display name:

Description:

IP location:

☐ Overwrite existing

[?](#) [< Back](#) [Next >](#) [Finish](#) [Cancel](#)

Figure 17: Naming the new custom IP block

After that on the last step choose “Add IP to the user repository”. This will let one to choose the newly created IP block in the + menu of the block design editor.

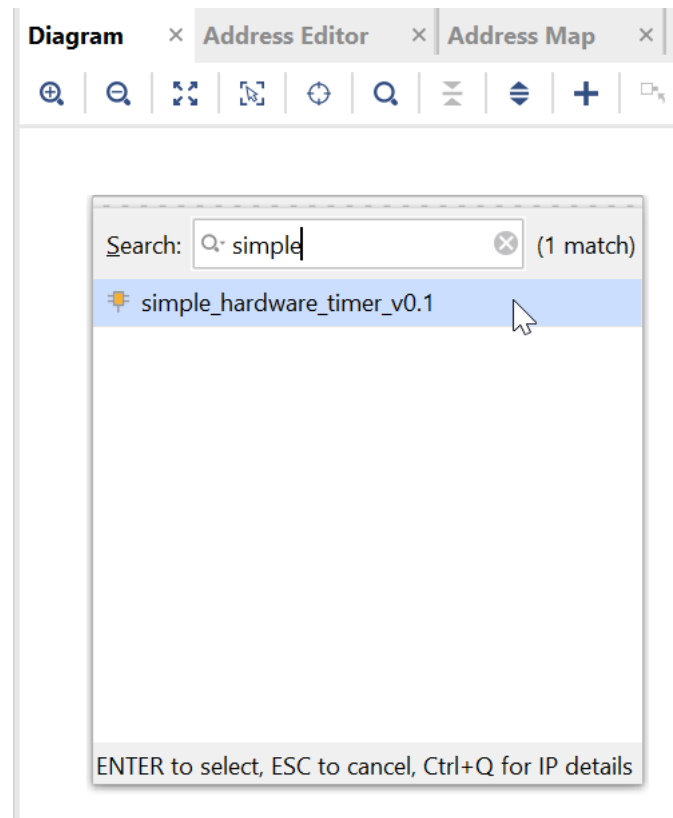


Figure 18: Adding newly created IP block to the block design

The suggestion to automate the connections will appear and using this helper tool the resulting design should be the following.

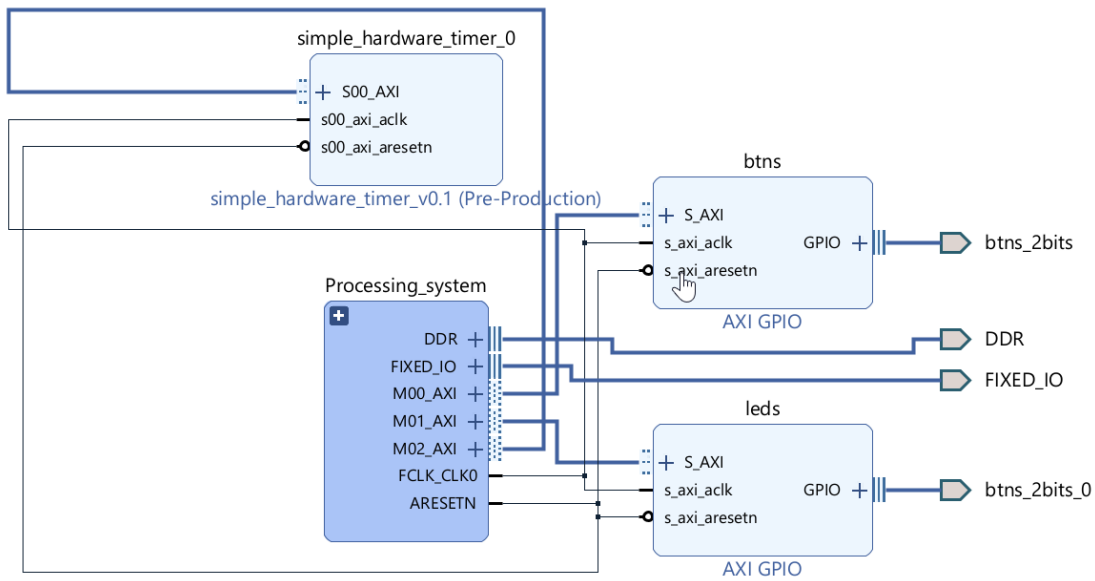


Figure 19: New custom IP connected to the AXI system bus

It is now only a functional template that provides memory-mapped RW registers on the AXI system bus. To add functionality we need to edit the contents of the block.

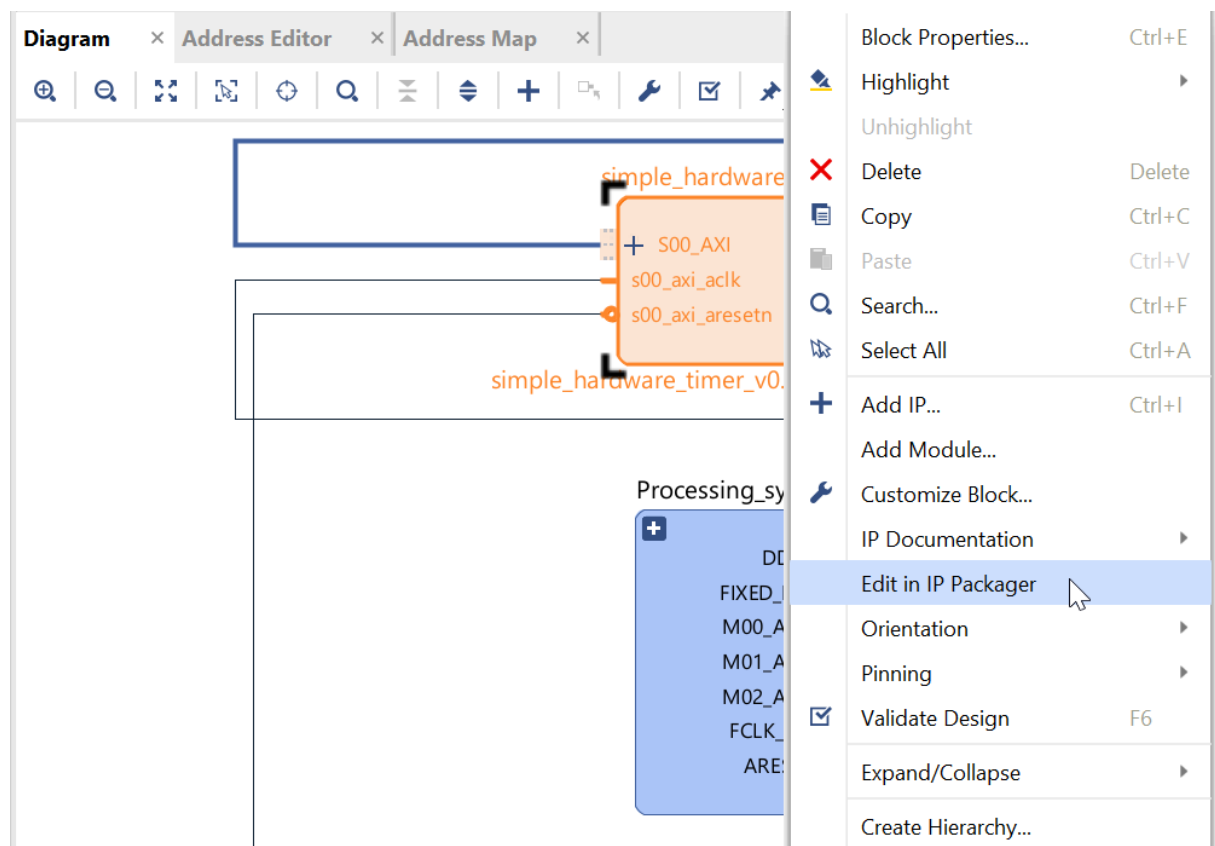


Figure 20: How to edit the custom IP block

After that a new temporary project window will appear and let us edit the contents of the custom IP as a regular project in a context that it would eventually be packaged into an IP for block design.

Use Appendix A and B to find the actual VHDL contents of the suggested modifications to the custom IP block. They are rich for comments, but to make the explanation flow fluid, here's a recap.

The custom IP module is a timer that counts based on the provided `CLK` clock signal. `CLK` is 100MHz by default.

The timer has: - Short CPU interrupt (1 cycle of incoming `CLK`) - Long software interrupt (clear-on-read, COR) - Customizable threshold (default one at the compile time and dynamically changeable through AXI)

The timer issues 2 types of interrupts: a short 1-cycle interrupt for the processing system and a "long" one for the software that would poll the FPGA fabric in order to wait for the interrupt ("soft" interrupt). User-added customizable parameter provides an option to manually set the default (on-boot) timer threshold value.

After pasting replacing the code with the contents from Appendixes A and B, let's add customization

GUI parameters for the ease of changes in the future. In the “Package IP” choose “Customization Parameters” and add `timer_max_value` parameter that is present in the code (in both files and routed through hierarchy).

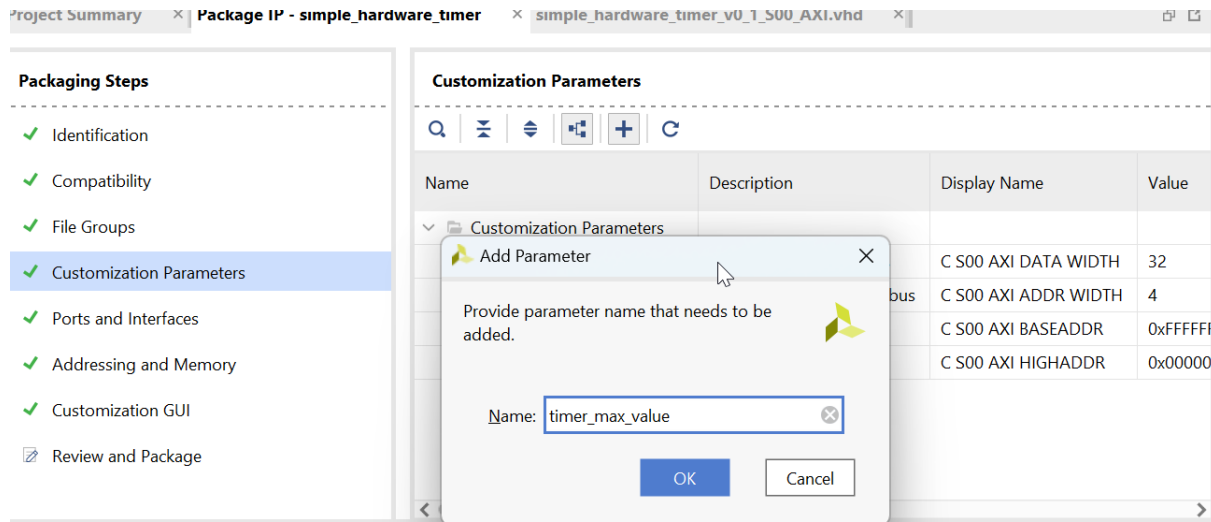
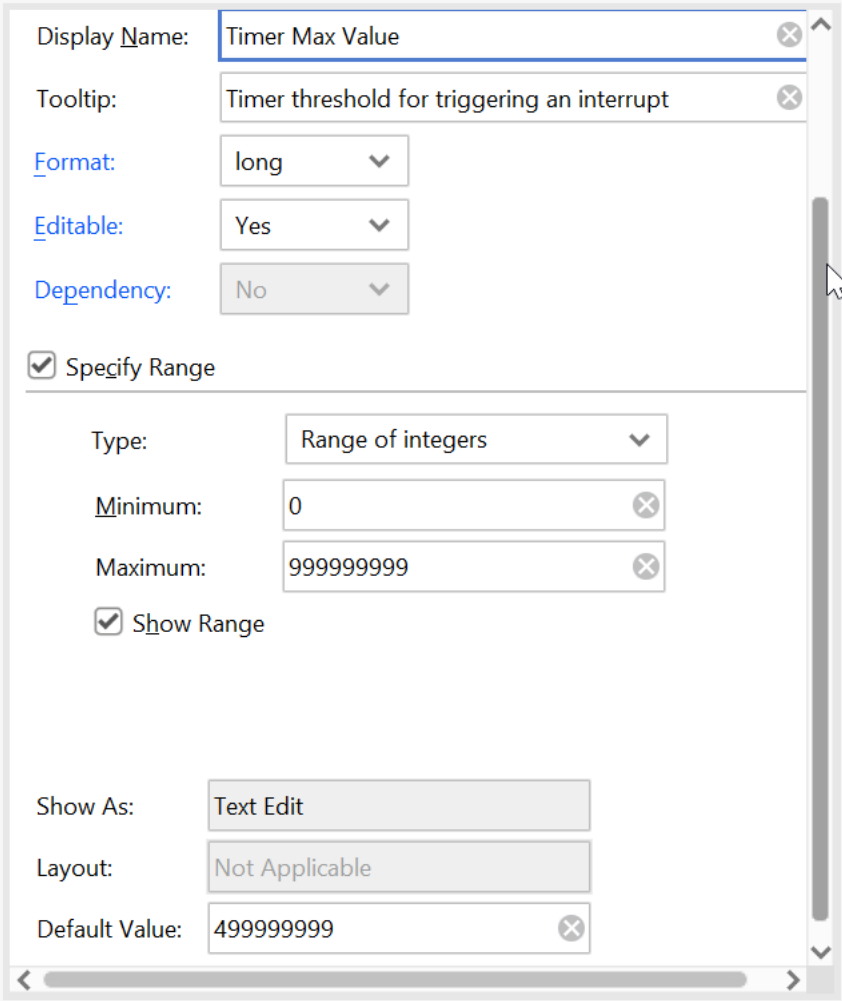


Figure 21: Adding new parameters to the custom IP customization GUI

After that, I suggest editing the newly created parameter in this fashion. Limiting the maximum threshold value to 1 seconds (based on 100 MHz `CLK` input) is “just because” and doesn’t affect the design drastically.

Use the options below to customize how the parameter will appear in the Customization GUI for users of the IP.



The dialog box contains the following fields and options:

- Display Name:** Timer Max Value
- Tooltip:** Timer threshold for triggering an interrupt
- Format:** long
- Editable:** Yes
- Dependency:** No
- ☒ **Specify Range**
 - Type:** Range of integers
 - Minimum:** 0
 - Maximum:** 999999999
 - ☒ **Show Range**
- Show As:** Text Edit
- Layout:** Not Applicable
- Default Value:** 499999999

Buttons: OK, Cancel

Figure 22: Editing the customizable parameter

Compile the project to be sure everything is sound. Also, it is intended to have testbench to test the functionality at this stage but I'm going to omit it for now in order to speed up the development.

Use "Re-Package IP" button on "Review and Package" tab, agree to close the temporary project.

After that the block design will offer the user to refresh the IP because it has changed. It is noticeable that our IP doesn't reflect the added `timer_interrupt` output (other internal changes are not there

yet, too).

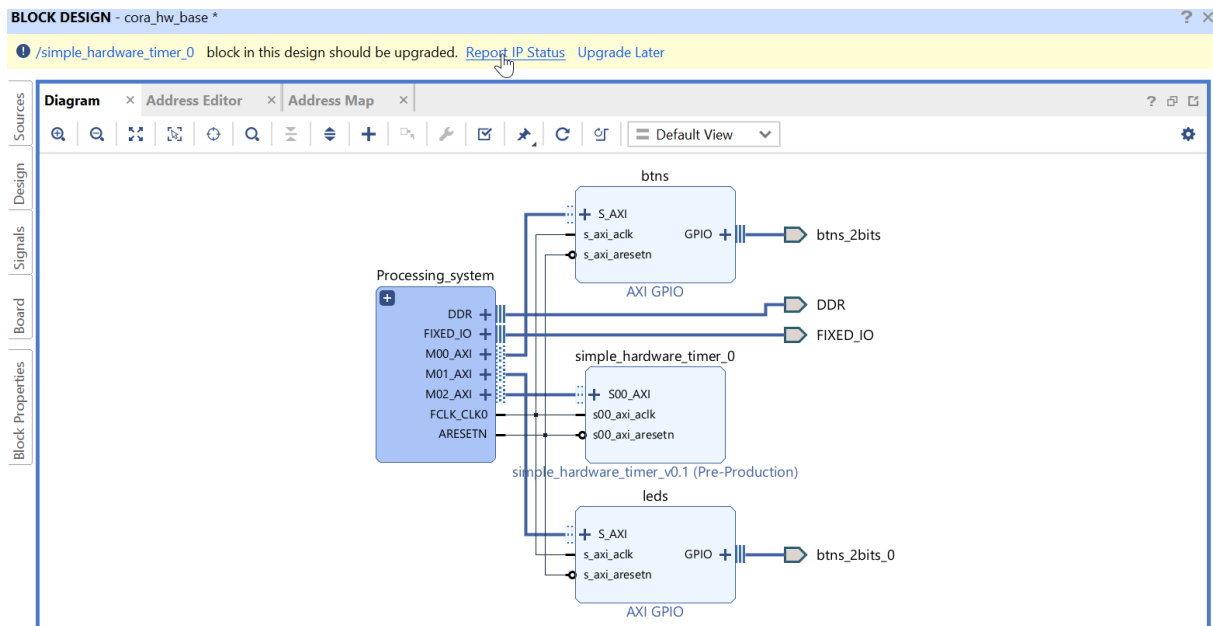


Figure 23: Afteredit IP refresh suggestion

Vivado tracks the changes of the IPs in the connected repositories and offers updates if they are needed. It can be seen in the following IP report.

IP Status								
<input checked="" type="checkbox"/> Revision Change (1) <input checked="" type="checkbox"/> Up-to-dates (5) <input type="button" value="Hide All"/>								
Source File	IP Status	Recommendation	Change ...	IP Name	Current Ver...	Recommended Ver...	Licen...	Current Pa
cora_hw_base (6)	<input checked="" type="checkbox"/>							
/simple_hardware_timer_0	<input checked="" type="checkbox"/> IP revision change. IP definition 'simple_...	Upgrade IP		simple_hardware_timer_v0.1	0.1 (Rev. 1)	0.1 (Rev. 2)	Included	xc7z010cl
/Processing_system/axi_interconnect_0	<input type="checkbox"/> Up-to-date	No changes required	More info	AXI Interconnect	2.1 (Rev. 26)	2.1 (Rev. 26)	Included	xc7z010cl
/Processing_system/proc_sys_reset_0	<input type="checkbox"/> Up-to-date	No changes required	More info	Processor System Reset	5.0 (Rev. 13)	5.0 (Rev. 13)	Included	xc7z010cl
/Processing_system/processing_system7_0	<input type="checkbox"/> Up-to-date	No changes required	More info	ZYNQ7 Processing System	5.5 (Rev. 6)	5.5 (Rev. 6)	Included	xc7z010cl

Figure 24: IP report window reflecting the changes made

After upgrading the IP to the new version, Vivado will suggest generating the output products for the IP. The default option is “Out-of-context per IP” and it would suffice in our case. “Global” is rarely needed because of the modular nature of the SoC designs. Agreeing to generate the output products would result in the generation process put to background.

Upgrading IP will result in timer_interrupt appearing on the IP block.

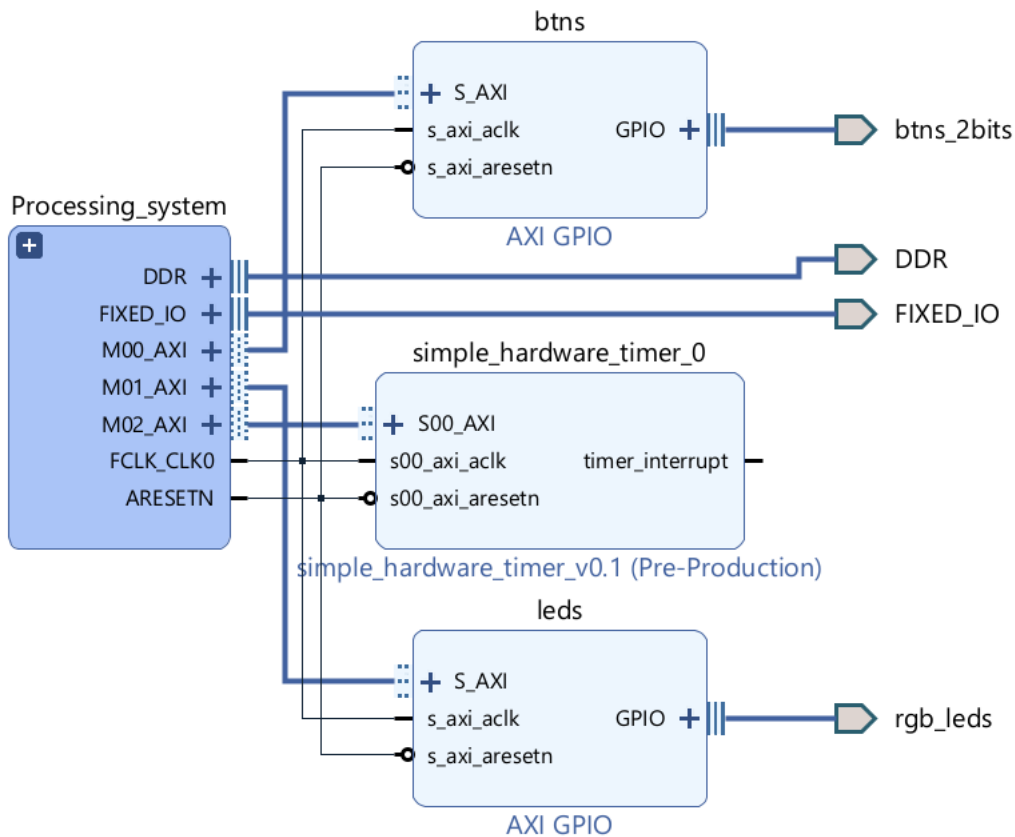


Figure 25: Block design with upgraded custom IP

In order to be able to use the interrupt later, it needs to be connected to the PS.

Expand the hierarchy by hitting “+” on the **Processing_system** block. Enter the PS settings and enable the interrupts as shown on the following figure.

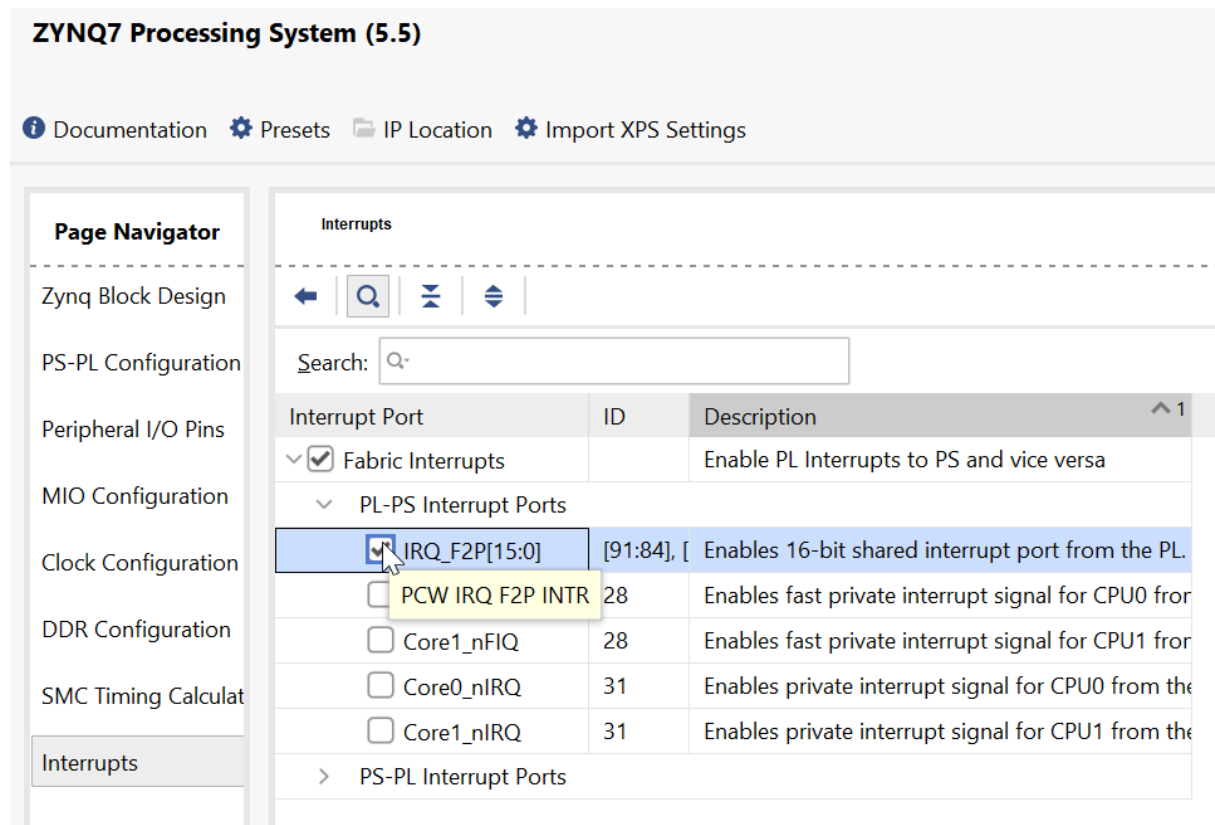


Figure 26: Enabling the PL -> PS interrupts

In order to connect one or more interrupts, I suggest adding [Concat](#) block to the block design. It would let the user connect multiple interrupts from the PL part in the future. Creating concat with the “16” parameter value would let the user have the more predictable values when they would be connecting their peripherals through device tree (not only the custom ones). For some reason (maybe it’s fixed in some version) automatically assigned numbers of interrupts vary if the concat block contains only one or 2 pins.

It is easier to just have 16 inputs all the time in the HDL design, the ones that are not used are tied to logical “0”. The resulting IRQ numbers, as they should be put in the device tree later during linux build, are shown on the following figure.

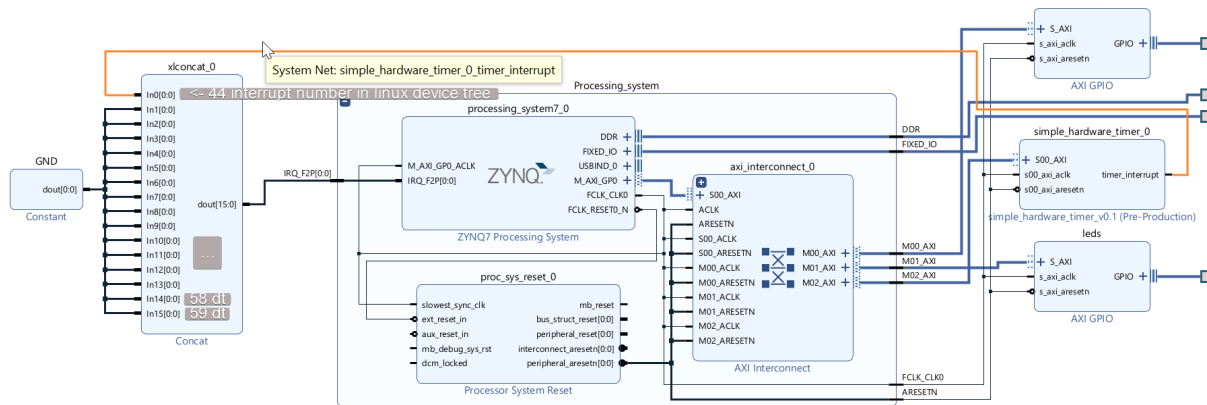


Figure 27: Interrupt connections and interrupt numbers for device tree

I suggest moving the **Concat** and **GND** blocks into the “Processing system” hierarchy in order to make the design look neat with service modules hidden.

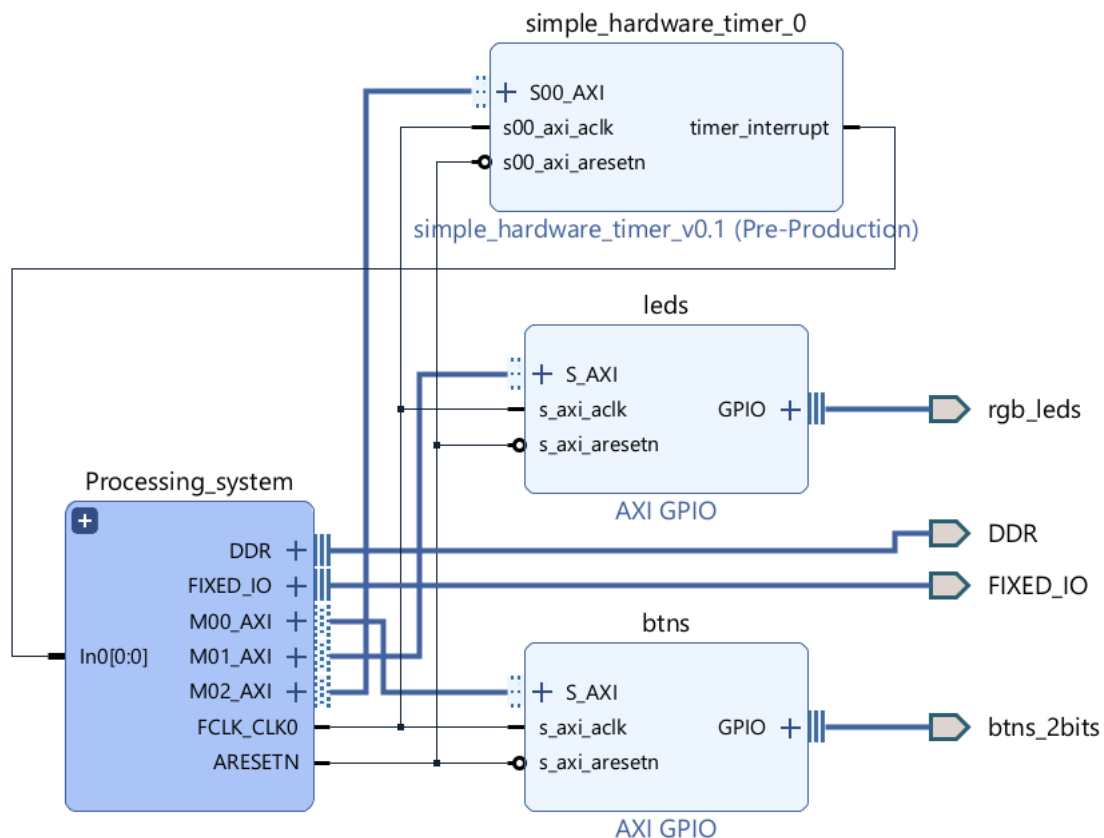


Figure 28: Resulting block design

As the last step to finalize the HDL block design development, in “Sources” tab click on the “Create HDL Wrapper...” in order to make the top-level text description of the blocks used in block design. It is going to be the top file that contains the system design that is auto-updated by Vivado when user makes changes to the block design or its subblocks. I don’t recommend turning the auto-update feature off.

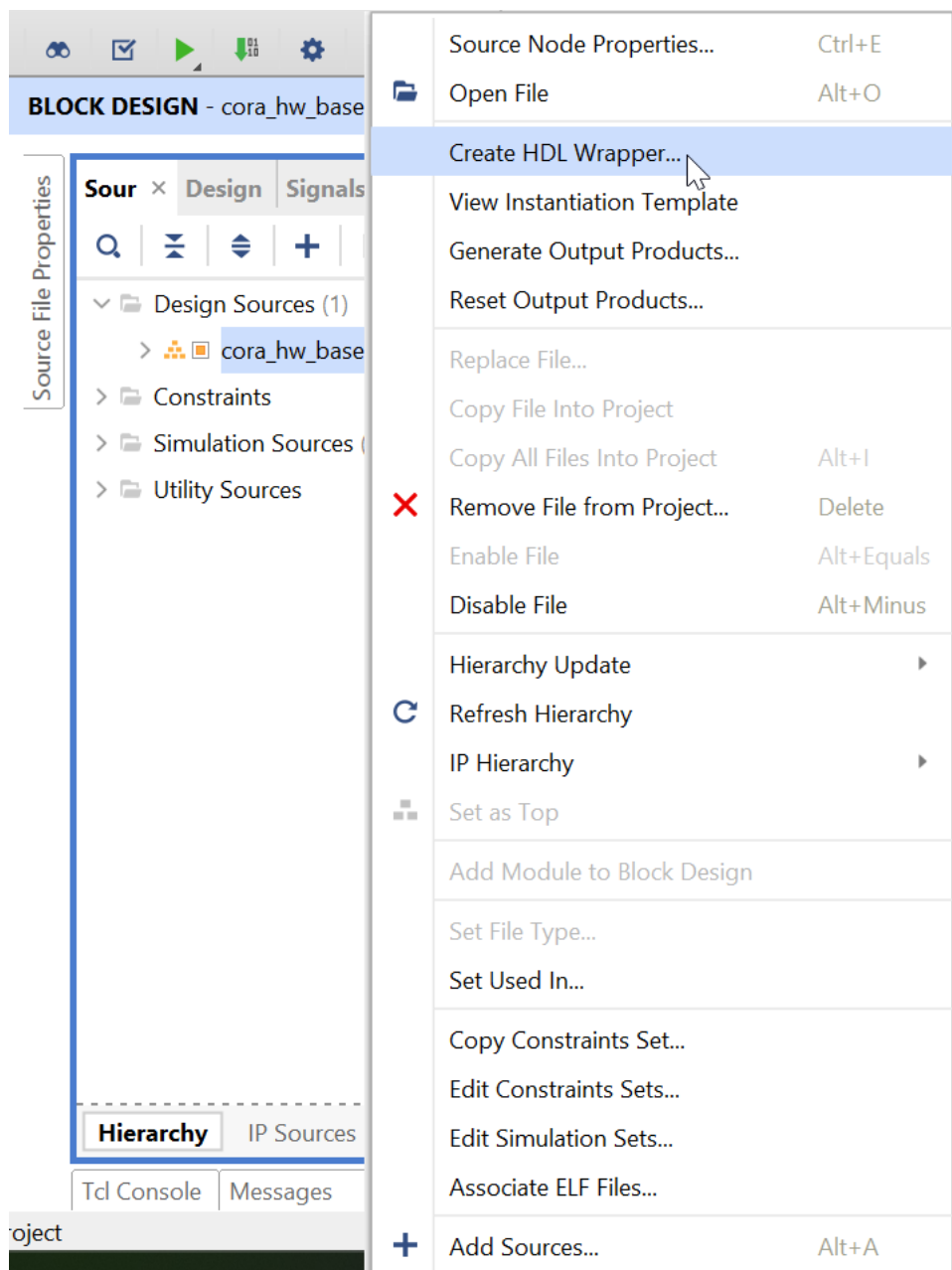


Figure 29: Creating HDL wrapper for the design

At this stage the main part of the design can be considered finished.

Adding debug nodes

In order to make the debugging easier, it is very convenient to use ChipScope tool to watch the real signals withing our FPGA design.

It is not intended that it is used as the main testing tool, but for the simple projects the testbench creating sometimes can be omitted. I don't encourage not having testbenches before going to this step.

In the Flow navigator click "Run Synthesis". After it finishes, the "Set Up Debug" button will be available.

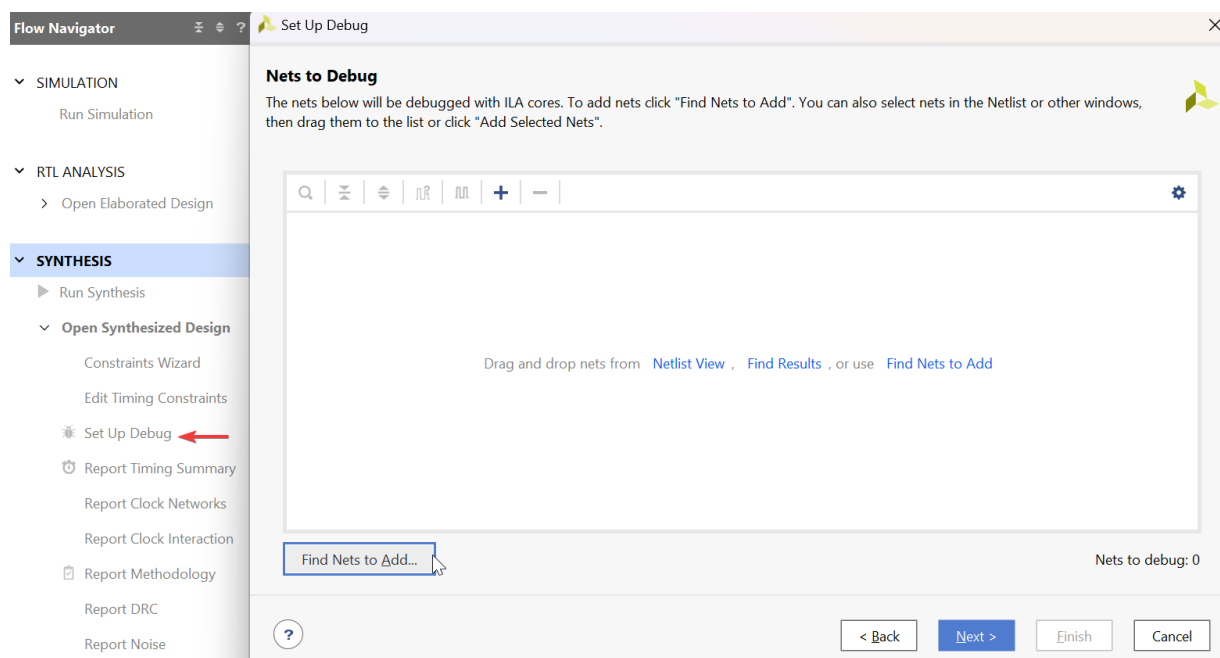


Figure 30: Set Up Debug

In this window using "Find Nets to Add..." add the interesting nodes from our custom IP to be available later for the user to see in ChipScope tool (part of Hardware manager in Vivado).

I suggest adding the following nodes.

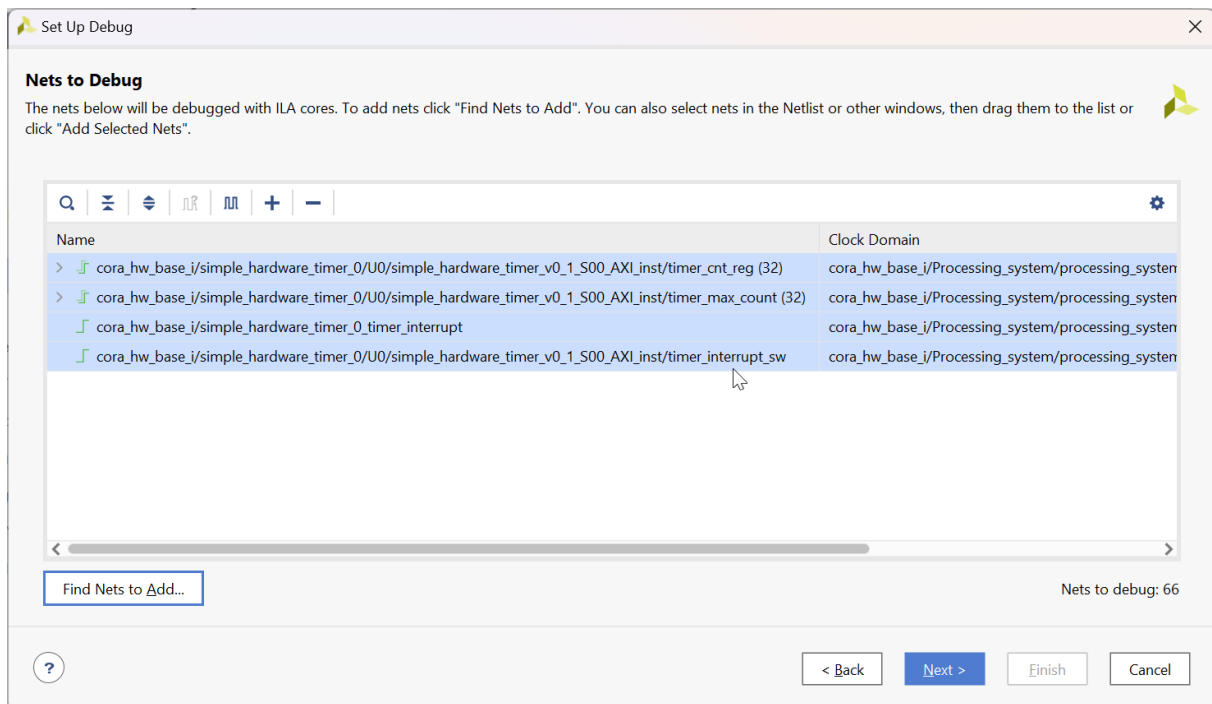


Figure 31: Adding custom IP nodes

Increase the amount of memory taken by the ChipScope debugger to 8k in order to fit more samples in one take (more timespan). Those samples are in base clock ticks, i.e. $8k * 10 \text{ ns}$ period, that is the amount of time one capture is going to have.

After finishing setting up the debug nodes, use "ctrl+s" to save the changes. Maybe the Vivado would ask where to store the changes (*.xdc constraints file), I suggest naming the file "debug_nodes" and having all the ChipScope settings stored there. It would locate in "Sources" tab under "Constraints" subfolder, if added.

Close Synthesized window layout and click "Generate bitstream".

Exporting the design for petalinux tools to use

Use **File** -> **Export**... to export the hardware description that would be used by petalinux tools to create linux builds. It is also suitable for Vitis IDE that is useful to quickly create baremetal applications based on created designs.

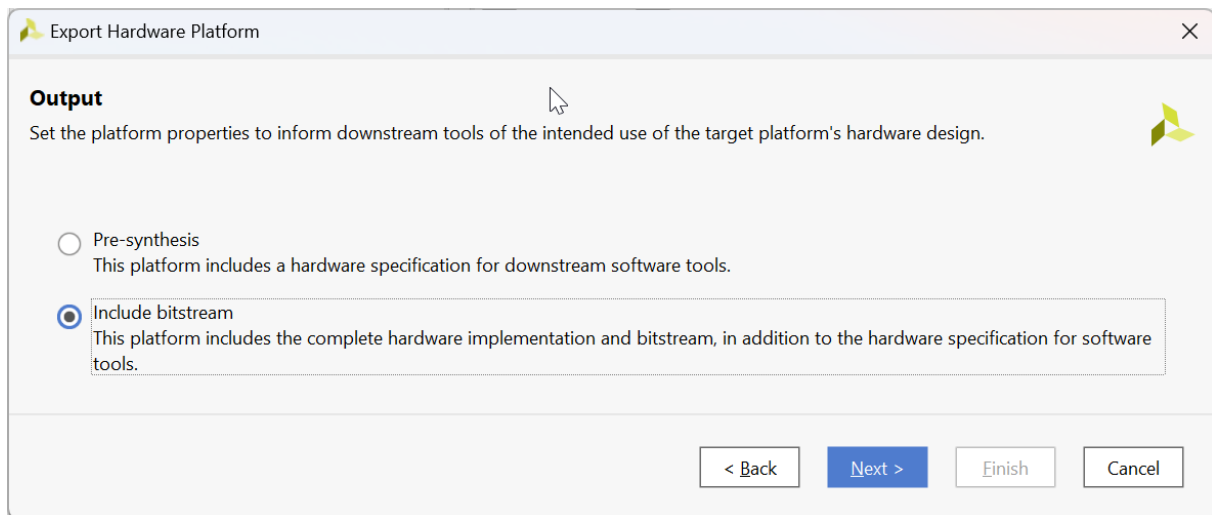


Figure 32: Export hardware window

Include bitstream in order for petalinux to automatically embedd the *.bit file into the linux build and automatically load it at launch.

I suggest having an external to the project location to store the *.xsa hardware descriptions. They can be used outside of vivado design flow by the people who don't need to know or have the Vivado design project.

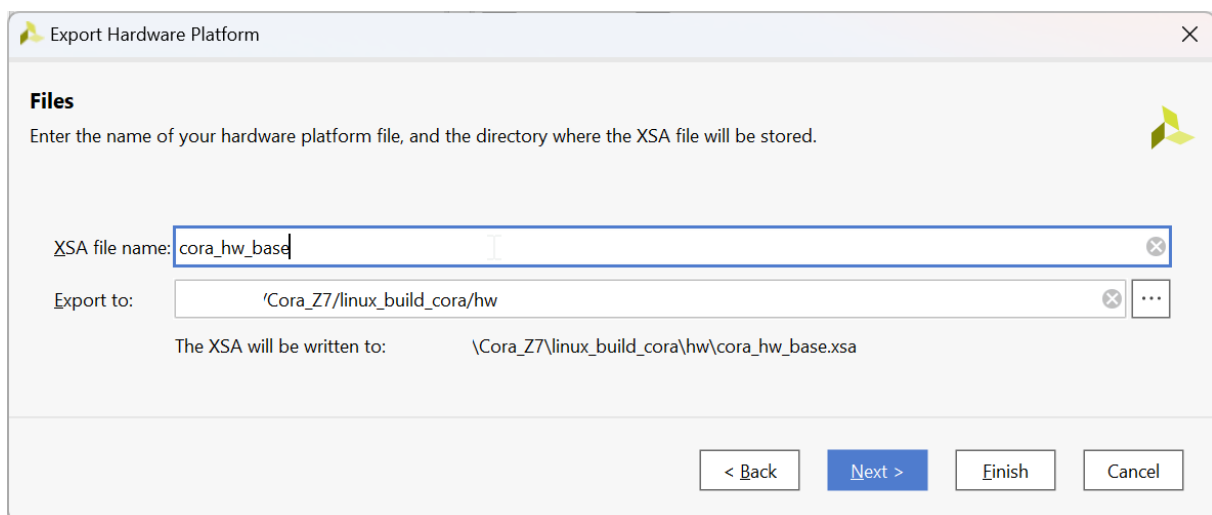


Figure 33: Export location

After that it is safe to close Vivado and navigate to petalinux flow (and instructions manual).

This guide is subject to change. More remarks may be added.

Appendix A (simple_hardware_timer_v0.1.vhd)

Listing 1: Contents of the simple_hardware_timer_v0.1.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity simple_hardware_timer_v0_1 is
6      generic (
7          -- Users to add parameters here
8          timer_max_value : integer := 499999999; -- in CLK cycles, '0'
              counts, with 100 MHz: 10 ns * 499999999+1 = 500 ms
9          -- User parameters ends
10         -- Do not modify the parameters beyond this line
11
12
13         -- Parameters of Axi Slave Bus Interface S00_AXI
14         C_S00_AXI_DATA_WIDTH    : integer    := 32;
15         C_S00_AXI_ADDR_WIDTH    : integer    := 4
16     );
17     port (
18         -- Users to add ports here
19         timer_interrupt : out std_logic; -- timer event interrupt
20         -- User ports ends
21         -- Do not modify the ports beyond this line
22
23
24         -- Ports of Axi Slave Bus Interface S00_AXI
25         s00_axi_aclk      : in  std_logic;
26         s00_axi_aresetn   : in  std_logic;
27         s00_axi_awaddr     : in  std_logic_vector(C_S00_AXI_ADDR_WIDTH-1
              downto 0);
28         s00_axi_awprot     : in  std_logic_vector(2 downto 0);
29         s00_axi_awvalid    : in  std_logic;
30         s00_axi_awready    : out std_logic;
31         s00_axi_wdata      : in  std_logic_vector(C_S00_AXI_DATA_WIDTH-1
              downto 0);
32         s00_axi_wstrb      : in
              std_logic_vector((C_S00_AXI_DATA_WIDTH/8)-1 downto 0);
33         s00_axi_wvalid     : in  std_logic;
34         s00_axi_wready     : out std_logic;
```



```

35     s00_axi_bresp    : out std_logic_vector(1 downto 0);
36     s00_axi_bvalid   : out std_logic;
37     s00_axi_bready    : in std_logic;
38     s00_axi_araddr    : in std_logic_vector(C_S00_AXI_ADDR_WIDTH-1
        downto 0);
39     s00_axi_arprot    : in std_logic_vector(2 downto 0);
40     s00_axi_arvalid   : in std_logic;
41     s00_axi_arready    : out std_logic;
42     s00_axi_rdata     : out std_logic_vector(C_S00_AXI_DATA_WIDTH-1
        downto 0);
43     s00_axi_rresp     : out std_logic_vector(1 downto 0);
44     s00_axi_rvalid    : out std_logic;
45     s00_axi_rready    : in std_logic
46 );
47 end simple_hardware_timer_v0_1;
48
49 architecture arch_imp of simple_hardware_timer_v0_1 is
50
51     -- component declaration
52     component simple_hardware_timer_v0_1_S00_AXI is
53         generic (
54             timer_max_value : integer := 499999999;
55             C_S_AXI_DATA_WIDTH : integer := 32;
56             C_S_AXI_ADDR_WIDTH : integer := 4
57         );
58         port (
59             timer_interrupt : out std_logic; -- timer event interrupt from
                the lower hierarchy
60             S_AXI_ACLK      : in std_logic;
61             S_AXI_ARESETN   : in std_logic;
62             S_AXI_AWADDR    : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1
                downto 0);
63             S_AXI_AWPROT    : in std_logic_vector(2 downto 0);
64             S_AXI_AWVALID   : in std_logic;
65             S_AXI_AWREADY   : out std_logic;
66             S_AXI_WDATA     : in std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto
                0);
67             S_AXI_WSTRB     : in std_logic_vector((C_S_AXI_DATA_WIDTH/8)-1
                downto 0);
68             S_AXI_WVALID    : in std_logic;
69             S_AXI_WREADY    : out std_logic;
70             S_AXI_BRESP     : out std_logic_vector(1 downto 0);
71             S_AXI_BVALID    : out std_logic;

```

```

72     S_AXI_BREADY      : in std_logic;
73     S_AXI_ARADDR      : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1
        downto 0);
74     S_AXI_ARPROT      : in std_logic_vector(2 downto 0);
75     S_AXI_ARVALID     : in std_logic;
76     S_AXI_ARREADY     : out std_logic;
77     S_AXI_RDATA       : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto
        0);
78     S_AXI_RRESP       : out std_logic_vector(1 downto 0);
79     S_AXI_RVALID      : out std_logic;
80     S_AXI_RREADY      : in std_logic
81 );
82 end component simple_hardware_timer_v0_1_S00_AXI;
83
84 begin
85
86 -- Instantiation of Axi Bus Interface S00_AXI
87 simple_hardware_timer_v0_1_S00_AXI_inst :
    simple_hardware_timer_v0_1_S00_AXI
88     generic map (
89         timer_max_value => timer_max_value, -- pass the parameter to
            the lower hierarchy
90         C_S_AXI_DATA_WIDTH => C_S00_AXI_DATA_WIDTH,
91         C_S_AXI_ADDR_WIDTH => C_S00_AXI_ADDR_WIDTH
92     )
93     port map (
94         timer_interrupt => timer_interrupt, -- pass the interrupt
            signal from the lower hierarchy
95         S_AXI_ACLK      => s00_axi_aclk,
96         S_AXI_ARESETN   => s00_axi_aresetn,
97         S_AXI_AWADDR     => s00_axi_awaddr,
98         S_AXI_AWPROT     => s00_axi_awprot,
99         S_AXI_AWVALID    => s00_axi_awvalid,
100        S_AXI_AWREADY    => s00_axi_awready,
101        S_AXI_WDATA      => s00_axi_wdata,
102        S_AXI_WSTRB      => s00_axi_wstrb,
103        S_AXI_WVALID     => s00_axi_wvalid,
104        S_AXI_WREADY     => s00_axi_wready,
105        S_AXI_BRESP      => s00_axi_bresp,
106        S_AXI_BVALID     => s00_axi_bvalid,
107        S_AXI_BREADY     => s00_axi_bready,
108        S_AXI_ARADDR     => s00_axi_araddr,
109        S_AXI_ARPROT     => s00_axi_arprot,

```

```
110     S_AXI_ARVALID    => s00_axi_arvalid,
111     S_AXI_ARREADY    => s00_axi_arready,
112     S_AXI_RDATA      => s00_axi_rdata,
113     S_AXI_RRESP      => s00_axi_rresp,
114     S_AXI_RVALID     => s00_axi_rvalid,
115     S_AXI_RREADY     => s00_axi_rready
116 );
117
118     -- Add user logic here
119
120     -- User logic ends
121
122 end arch_imp;
```

Appendix B (simple_hardware_timer_v0.1_S00_AXI.vhd)

Listing 2: Contents of the simple_hardware_timer_v0.1_S00_AXI.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use ieee.std_logic_unsigned.all; -- it is NOT recommended to use
   unsigned with numeric_std, but for this simple example we can do
   that safely
5
6  entity simple_hardware_timer_v0_1_S00_AXI is
7      generic (
8          -- Users to add parameters here
9          timer_max_value : integer := 499999999; -- in CLK cycles, '0'
            counts, with 100 MHz: 10 ns * 499999999+1 = 500 ms
10         -- User parameters ends
11         -- Do not modify the parameters beyond this line
12
13         -- Width of S_AXI data bus
14         C_S_AXI_DATA_WIDTH  : integer  := 32;
15         -- Width of S_AXI address bus
16         C_S_AXI_ADDR_WIDTH  : integer  := 4
17     );
18     port (
19         -- Users to add ports here
20         timer_interrupt : out std_logic; -- timer event interrupt
21         -- User ports ends
22         -- Do not modify the ports beyond this line
23
24         -- Global Clock Signal
25         S_AXI_ACLK      : in std_logic;
26         -- Global Reset Signal. This Signal is Active LOW
27         S_AXI_ARESETN   : in std_logic;
28         -- Write address (issued by master, accepted by Slave)
29         S_AXI_AWADDR    : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1
            downto 0);
30         -- Write channel Protection type. This signal indicates the
31         -- privilege and security level of the transaction, and
            whether
32         -- the transaction is a data access or an instruction
            access.
```

```
33     S_AXI_AWPROT      : in std_logic_vector(2 downto 0);
34     -- Write address valid. This signal indicates that the master
      signaling
35     -- valid write address and control information.
36     S_AXI_AWVALID     : in std_logic;
37     -- Write address ready. This signal indicates that the slave
      is ready
38     -- to accept an address and associated control signals.
39     S_AXI_AWREADY     : out std_logic;
40     -- Write data (issued by master, accepted by Slave)
41     S_AXI_WDATA       : in std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto
      0);
42     -- Write strobes. This signal indicates which byte lanes hold
43     -- valid data. There is one write strobe bit for each eight
44     -- bits of the write data bus.
45     S_AXI_WSTRB       : in std_logic_vector((C_S_AXI_DATA_WIDTH/8)-1
      downto 0);
46     -- Write valid. This signal indicates that valid write
47     -- data and strobes are available.
48     S_AXI_WVALID      : in std_logic;
49     -- Write ready. This signal indicates that the slave
50     -- can accept the write data.
51     S_AXI_WREADY      : out std_logic;
52     -- Write response. This signal indicates the status
53     -- of the write transaction.
54     S_AXI_BRESP       : out std_logic_vector(1 downto 0);
55     -- Write response valid. This signal indicates that the channel
56     -- is signaling a valid write response.
57     S_AXI_BVALID      : out std_logic;
58     -- Response ready. This signal indicates that the master
59     -- can accept a write response.
60     S_AXI_BREADY      : in std_logic;
61     -- Read address (issued by master, accepted by Slave)
62     S_AXI_ARADDR      : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1
      downto 0);
63     -- Protection type. This signal indicates the privilege
64     -- and security level of the transaction, and whether the
65     -- transaction is a data access or an instruction access.
66     S_AXI_ARPROT      : in std_logic_vector(2 downto 0);
67     -- Read address valid. This signal indicates that the channel
68     -- is signaling valid read address and control information.
69     S_AXI_ARVALID     : in std_logic;
70     -- Read address ready. This signal indicates that the slave is
```

```

71         -- ready to accept an address and associated control
           signals.
72     S_AXI_ARREADY    : out std_logic;
73     -- Read data (issued by slave)
74     S_AXI_RDATA      : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto
           0);
75     -- Read response. This signal indicates the status of the
76     -- read transfer.
77     S_AXI_RRESP      : out std_logic_vector(1 downto 0);
78     -- Read valid. This signal indicates that the channel is
79     -- signaling the required read data.
80     S_AXI_RVALID      : out std_logic;
81     -- Read ready. This signal indicates that the master can
82     -- accept the read data and response information.
83     S_AXI_RREADY      : in std_logic
84 );
85 end simple_hardware_timer_v0_1_S00_AXI;
86
87 architecture arch_imp of simple_hardware_timer_v0_1_S00_AXI is
88
89     -- AXI4LITE signals
90     signal axi_awaddr    : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto
           0);
91     signal axi_awready    : std_logic;
92     signal axi_wready    : std_logic;
93     signal axi_bresp      : std_logic_vector(1 downto 0);
94     signal axi_bvalid     : std_logic;
95     signal axi_araddr     : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto
           0);
96     signal axi_arready    : std_logic;
97     signal axi_rdata      : std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto
           0);
98     signal axi_rresp      : std_logic_vector(1 downto 0);
99     signal axi_rvalid     : std_logic;
100
101     -- Example-specific design signals
102     -- local parameter for addressing 32 bit / 64 bit
       C_S_AXI_DATA_WIDTH
103     -- ADDR_LSB is used for addressing 32/64 bit registers/memories
104     -- ADDR_LSB = 2 for 32 bits (n downto 2)
105     -- ADDR_LSB = 3 for 64 bits (n downto 3)
106     constant ADDR_LSB    : integer := (C_S_AXI_DATA_WIDTH/32)+ 1;
107     constant OPT_MEM_ADDR_BITS : integer := 1;

```

```
108 -----
109 ---- Signals for user logic register space example
110 -----
111 ---- Number of Slave Registers 4
112 signal slv_reg0 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
113 signal slv_reg1 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
114 signal slv_reg2 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
115 signal slv_reg3 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
116 signal slv_reg_rden : std_logic;
117 signal slv_reg_wren : std_logic;
118 signal reg_data_out :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto
    0);
119 signal byte_index : integer;
120 signal aw_en : std_logic;
121
122 -- User signals
123 signal timer_cnt : std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto
    0); -- the same range as the AXI bus
124 signal timer_max_count : std_logic_vector(C_S_AXI_DATA_WIDTH-1
    downto 0); -- storage for max value that could be overwritten
    from AXI
125 signal timer_interrupt_sw : std_logic; -- "long" interrupt for the
    software to be able to use polling
126
127
128 begin
129     -- I/O Connections assignments
130
131     S_AXI_AWREADY <= axi_awready;
132     S_AXI_WREADY <= axi_wready;
133     S_AXI_BRESP <= axi_bresp;
134     S_AXI_BVALID <= axi_bvalid;
135     S_AXI_ARREADY <= axi_arready;
136     S_AXI_RDATA <= axi_rdata;
137     S_AXI_RRESP <= axi_rresp;
138     S_AXI_RVALID <= axi_rvalid;
139     -- Implement axi_awready generation
140     -- axi_awready is asserted for one S_AXI_ACLK clock cycle when both
141     -- S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready is
142     -- de-asserted when reset is low.
143
144     process (S_AXI_ACLK)
145     begin
```

```
146     if rising_edge(S_AXI_ACLK) then
147         if S_AXI_ARESETN = '0' then
148             axi_awready <= '0';
149             aw_en <= '1';
150         else
151             if (axi_awready = '0' and S_AXI_AWVALID = '1' and
152                 S_AXI_WVALID = '1' and aw_en = '1') then
153                 -- slave is ready to accept write address when
154                 -- there is a valid write address and write data
155                 -- on the write address and data bus. This design
156                 -- expects no outstanding transactions.
157                 axi_awready <= '1';
158                 aw_en <= '0';
159                 elsif (S_AXI_BREADY = '1' and axi_bvalid = '1') then
160                     aw_en <= '1';
161                     axi_awready <= '0';
162                 else
163                     axi_awready <= '0';
164                 end if;
165             end if;
166         end if;
167     end process;
168
169     -- Implement axi_awaddr latching
170     -- This process is used to latch the address when both
171     -- S_AXI_AWVALID and S_AXI_WVALID are valid.
172
173     process (S_AXI_ACLK)
174     begin
175         if rising_edge(S_AXI_ACLK) then
176             if S_AXI_ARESETN = '0' then
177                 axi_awaddr <= (others => '0');
178             else
179                 if (axi_awready = '0' and S_AXI_AWVALID = '1' and
180                     S_AXI_WVALID = '1' and aw_en = '1') then
181                     -- Write Address latching
182                     axi_awaddr <= S_AXI_AWADDR;
183                 end if;
184             end if;
185         end if;
186     end process;
187
188     -- Implement axi_wready generation
```



```
187      -- axi_wready is asserted for one S_AXI_ACLK clock cycle when both
188      -- S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_wready is
189      -- de-asserted when reset is low.
190
191      process (S_AXI_ACLK)
192      begin
193          if rising_edge(S_AXI_ACLK) then
194              if S_AXI_ARESETN = '0' then
195                  axi_wready <= '0';
196              else
197                  if (axi_wready = '0' and S_AXI_WVALID = '1' and
198                     S_AXI_AWVALID = '1' and aw_en = '1') then
199                      -- slave is ready to accept write data when
200                      -- there is a valid write address and write data
201                      -- on the write address and data bus. This design
202                      -- expects no outstanding transactions.
203                      axi_wready <= '1';
204                  else
205                      axi_wready <= '0';
206                  end if;
207              end if;
208          end if;
209      end process;
210
211      -- Implement memory mapped register select and write logic
212      -- generation
213      -- The write data is accepted and written to memory mapped
214      -- registers when
215      -- axi_awready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are
216      -- asserted. Write strobes are used to
217      -- select byte enables of slave registers while writing.
218      -- These registers are cleared when reset (active low) is applied.
219      -- Slave register write enable is asserted when valid address and
220      -- data are available
221      -- and the slave is ready to accept the write address and write
222      -- data.
223      slv_reg_wren <= axi_wready and S_AXI_WVALID and axi_awready and
224                      S_AXI_AWVALID ;
225
226      process (S_AXI_ACLK)
227      variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
228      begin
229          if rising_edge(S_AXI_ACLK) then
```

```
223     if S_AXI_ARESETN = '0' then
224         timer_max_count <=
            std_logic_vector(to_unsigned(timer_max_value,
            C_S_AXI_DATA_WIDTH)); -- boot with the default value for
            the timer in place
225         slv_reg1 <= (others => '0');
226         slv_reg2 <= (others => '0');
227         slv_reg3 <= (others => '0');
228     else
229         loc_addr := axi_awaddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto
            ADDR_LSB);
230         if (slv_reg_wren = '1') then
231             case loc_addr is
232                 when b"00" =>
233                     for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
234                         if ( S_AXI_WSTRB(byte_index) = '1' ) then
235                             -- Respective byte enables are asserted as per
                                write strobes
236                             -- slave registor 0
237                             timer_max_count(byte_index*8+7 downto
                                byte_index*8) <= S_AXI_WDATA(byte_index*8+7
                                downto byte_index*8);
238                         end if;
239                     end loop;
240                 when b"01" =>
241                     for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
242                         if ( S_AXI_WSTRB(byte_index) = '1' ) then
243                             -- Respective byte enables are asserted as per
                                write strobes
244                             -- slave registor 1
245                             slv_reg1(byte_index*8+7 downto byte_index*8) <=
                                S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
246                         end if;
247                     end loop;
248                 when b"10" =>
249                     for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
250                         if ( S_AXI_WSTRB(byte_index) = '1' ) then
251                             -- Respective byte enables are asserted as per
                                write strobes
252                             -- slave registor 2
253                             slv_reg2(byte_index*8+7 downto byte_index*8) <=
                                S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
254                         end if;
```

```

255         end loop;
256     when b"11" =>
257         for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
258             if ( S_AXI_WSTRB(byte_index) = '1' ) then
259                 -- Respective byte enables are asserted as per
260                 -- write strobes
261                 slv_reg3(byte_index*8+7 downto byte_index*8) <=
262                     S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
263             end if;
264         end loop;
265     when others =>
266         slv_reg0 <= slv_reg0;
267         slv_reg1 <= slv_reg1;
268         slv_reg2 <= slv_reg2;
269         slv_reg3 <= slv_reg3;
270     end case;
271 end if;
272 end if;
273 end process;
274
275 -- Implement write response logic generation
276 -- The write response and response valid signals are asserted by
277 -- the slave
278 -- when axi_wready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are
279 -- asserted.
280 -- This marks the acceptance of address and indicates the status of
281 -- write transaction.
282
283 process (S_AXI_ACLK)
284 begin
285     if rising_edge(S_AXI_ACLK) then
286         if S_AXI_ARESETN = '0' then
287             axi_bvalid <= '0';
288             axi_bresp <= "00"; --need to work more on the responses
289         else
290             if (axi_awready = '1' and S_AXI_AWVALID = '1' and axi_wready
291                 = '1' and S_AXI_WVALID = '1' and axi_bvalid = '0' ) then
292                 axi_bvalid <= '1';
293                 axi_bresp <= "00";
294             elsif (S_AXI_BREADY = '1' and axi_bvalid = '1') then
295                 --check if bready is asserted while bvalid is high)

```

```
292         axi_bvalid <= '0'; --
           (there is a possibility that bready is always asserted
           high)
293     end if;
294 end if;
295 end if;
296 end process;
297
298 -- Implement axi_arready generation
299 -- axi_arready is asserted for one S_AXI_ACLK clock cycle when
300 -- S_AXI_ARVALID is asserted. axi_arready is
301 -- de-asserted when reset (active low) is asserted.
302 -- The read address is also latched when S_AXI_ARVALID is
303 -- asserted. axi_araddr is reset to zero on reset assertion.
304
305 process (S_AXI_ACLK)
306 begin
307     if rising_edge(S_AXI_ACLK) then
308         if S_AXI_ARESETN = '0' then
309             axi_arready <= '0';
310             axi_araddr <= (others => '1');
311         else
312             if (axi_arready = '0' and S_AXI_ARVALID = '1') then
313                 -- indicates that the slave has accepted the valid read
                 address
314                 axi_arready <= '1';
315                 -- Read Address latching
316                 axi_araddr <= S_AXI_ARADDR;
317             else
318                 axi_arready <= '0';
319             end if;
320         end if;
321     end if;
322 end process;
323
324 -- Implement axi_arvalid generation
325 -- axi_rvalid is asserted for one S_AXI_ACLK clock cycle when both
326 -- S_AXI_ARVALID and axi_arready are asserted. The slave registers
327 -- data are available on the axi_rdata bus at this instance. The
328 -- assertion of axi_rvalid marks the validity of read data on the
329 -- bus and axi_rresp indicates the status of read
    transaction.axi_rvalid
330 -- is deasserted on reset (active low). axi_rresp and axi_rdata are
```

```
331      -- cleared to zero on reset (active low).
332      process (S_AXI_ACLK)
333      begin
334          if rising_edge(S_AXI_ACLK) then
335              if S_AXI_ARESETN = '0' then
336                  axi_rvalid <= '0';
337                  axi_rresp <= "00";
338              else
339                  if (axi_arready = '1' and S_AXI_ARVALID = '1' and axi_rvalid
340                      = '0') then
341                      -- Valid read data is available at the read data bus
342                      axi_rvalid <= '1';
343                      axi_rresp <= "00"; -- 'OKAY' response
344                      elsif (axi_rvalid = '1' and S_AXI_RREADY = '1') then
345                          -- Read data is accepted by the master
346                          axi_rvalid <= '0';
347                      end if;
348                  end if;
349              end process;
350
351      -- Implement memory mapped register select and read logic
352      -- generation
353      -- Slave register read enable is asserted when valid address is
354      -- available
355      -- and the slave is ready to accept the read address.
356      slv_reg_rden <= axi_arready and S_AXI_ARVALID and (not axi_rvalid)
357      ;
358
359      process (slv_reg0, slv_reg1, slv_reg2, slv_reg3, axi_araddr,
360              S_AXI_ARESETN, slv_reg_rden)
361      variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
362      begin
363          -- Address decoding for reading registers
364          loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto
365              ADDR_LSB);
366          case loc_addr is
367              when b"00" =>
368                  reg_data_out <= timer_max_count; -- let the user read the
369                  current timer threshold
370              when b"01" =>
371                  reg_data_out(0) <= timer_interrupt_sw; -- let the polling
372                  software have the "interrupt_happened" bit
```

```

366         reg_data_out(C_S_AXI_DATA_WIDTH-1 downto 1) <= (others =>
            '0'); -- null the other bits
367     when b"10" =>
368         reg_data_out <= slv_reg2; --dummy regs, leftovers from the
            example
369     when b"11" =>
370         reg_data_out <= slv_reg3;
371     when others =>
372         reg_data_out <= (others => '0');
373     end case;
374 end process;
375
376 -- Output register or memory read data
377 process( S_AXI_ACLK ) is
378 begin
379     if (rising_edge (S_AXI_ACLK)) then
380         if ( S_AXI_ARESETN = '0' ) then
381             axi_rdata <= (others => '0');
382         else
383             if (slv_reg_rden = '1') then
384                 -- When there is a valid read address (S_AXI_ARVALID) with
385                 -- acceptance of read address by the slave (axi_arready),
386                 -- output the read data
387                 -- Read address mux
388                 axi_rdata <= reg_data_out;      -- register read data
389             end if;
390         end if;
391     end if;
392 end process;
393
394
395 -- Add user logic here
396 timer_proc:process( S_AXI_ACLK )
397 variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
    -- for similar address decoding
398 begin
399     loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto
        ADDR_LSB); -- for similar address decoding
400     if rising_edge(S_AXI_ACLK) then
401         if S_AXI_ARESETN = '0' then -- synchronous reset results in a
            more expected timing behaviour
402             timer_cnt <= (others => '0'); -- reset all the bits
                regardless of vector size

```

```
403         timer_interrupt_sw <= '0'; -- reset the software interrupt
           bit
404         timer_interrupt <= '0'; -- reset the CPU interrupt bit
405     else
406         timer_cnt <= timer_cnt + 1;
407         if timer_cnt >= timer_max_count then -- in 'if'
           statements <= and >= are comparisons (less/more or
           equal), don't mix them!
408             timer_cnt <= (others => '0');
409             timer_interrupt <= '1'; -- set interrupt for 1 cycle
           for the CPU
410             timer_interrupt_sw <= '1'; -- set the "long" interrupt
           for polling software
411         else
412             timer_interrupt <= '0'; -- interrupt is always '0'
           when the counter is not at the threshold
413         end if;
414         if loc_addr = "00" and slv_reg_rden = '1' then -- if the
           software read the software interrupt bit
415             timer_interrupt_sw <= '0';
416         end if;
417     end if;
418 end if;
419 end process;
420
421 -- User logic ends
422
423 end arch_imp;
```