VHDL | FPGA synthesis

Environment

Nous utiliserons la suite logicielle **Vivado** de **Xilinx** (now AMD) qui regroupe sous une même interface graphique un ensemble intégrant les outils de compilation, élaboration, synthèse et simulation de code HDL. A noter également la présence de l'outil **Vitis_HLS** qui permet l'exploration architecturale à partir de code C/C++
En complément, nous disposons également du simulateur **QuestaSim** (formerly known as **modelsim**).

Enfin, accessibles à tous sans licence, les outils **ghdl** et **gtkwave** vous permettent vos premiers pas dans l'univers du VHDL.

Quelques fonctions de conversions :

- *conv_std_logic_vector (integer, size)* ⇨ *std_logic_vector* use ieee.std_logic_arith.all
- *conv_integer (std_logic_vector)* ⇨ *integer* use ieee.std_logic_unsigned.all
- *to_stdlogicvector (bit_vector)* ⇨ *std_logic_vector* use ieee.std_logic_1164.all
- F <= std_logic(signal A or signal B) ⇨ *std_logic* *conversion en std_logic*

Design Flow

- **Compilation** des entités, architectures, [*configurations*] et d'une architecture de test.
- **Élaboration** de l'architecture de test précédemment compilée générant un *snapshot.*
- **Simulation** du *snapshot.*

GHDL & GTKwave for behavioural simulation

```
* Compilation
ghdl -a --ieee=synopsys -fexplicit <packages.vhd> <components.vhd> <testbench.vhd>

* Elaborate
ghdl -e --ieee=synopsys -fexplicit testbench

* Run simulation
ghdl -r --ieee=synopsys -fexplicit testbench --wave=testbench.ghw

* View results
gtkwave testbench.ghw
```

→ Xilinx series 7 (Zynq, Spartan7 …)
e.g boards like Zybo, Pynq …

Xilinx webpack suite 4 students

→ Xilinx series 6 (Spartan6)
e.g boards like Basys2 or Basys3
*(for **older** designs)*

Links

Master Secil/Siame VHDL
https://moodle.univ-tlse3.fr/mod/page/view.php?id=456296
(you'll find some useful links intended to speed up your learning curve)

VHDL course
https://secil.univ-tlse3.fr/teaching/francois/UE-VHDL/VHDL_course.pdf

VHDL 4 FPGA synthesis practical exercises (this file)
https://secil.univ-tlse3.fr/teaching/francois/UE-VHDL/VHDL_practical_exercises.pdf

Table of contents

Xilinx Vivado setup

Vivado is part of the '**Hardware targets**' tools from Xilinx ecosystem. It enables you to design FPGA's content through its UI, to synthesise and to simulate them.
*Note: it's also possible to simulate your design with external tools like **modelsim** … some additional setup and extra care are required.*

According to the envisioned targets, there exists two flavours:

- Xilinx ISE ⇒ for series 6 and older FPGA
- Xilinx Vivado ⇒ for series 7 and newer boards

We'll make use of the **Xilinx Vivado** tools in combination with our **Zybo**, **Zybo-z7** and **Pynq** boards. For more advanced designs, for example those that make use of the embedded ARM processor (Hard IP[1]), you'll need to have access to Intellectual Property blocks coming both from Xilinx and third-party designers.

Regarding the boards support files, extra care has already been undertaken leading to a seamless access to the boards definitions themselves. Henceforth, you'll just need to select the proper board from the Vivado's UI.

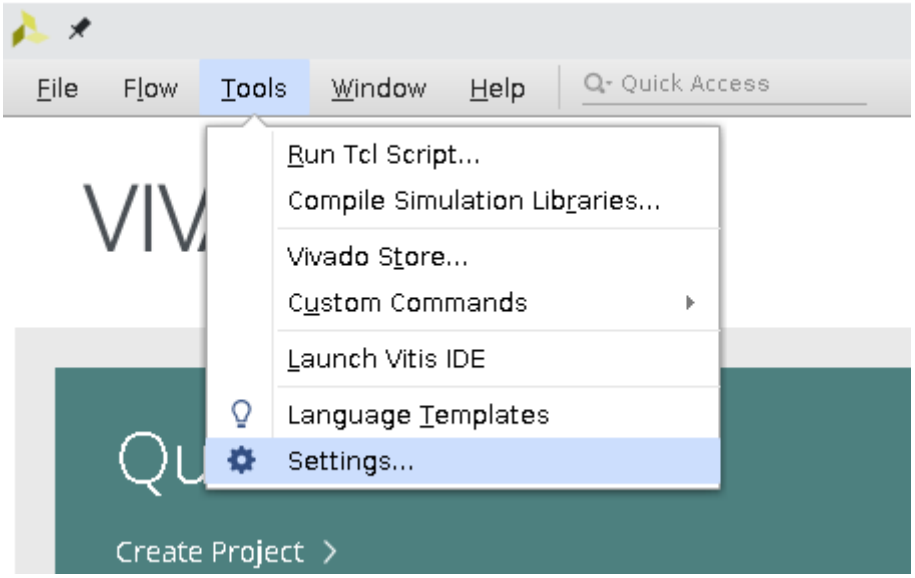
additional IP blocks

Zybo boards (and derivatives) from **Digilent** manufacturer makes use of specific IPs blocks. Those IPs have already been copied within the Xilinx tools repository, but you need to register them on a per user basis.

Launch the tools suite

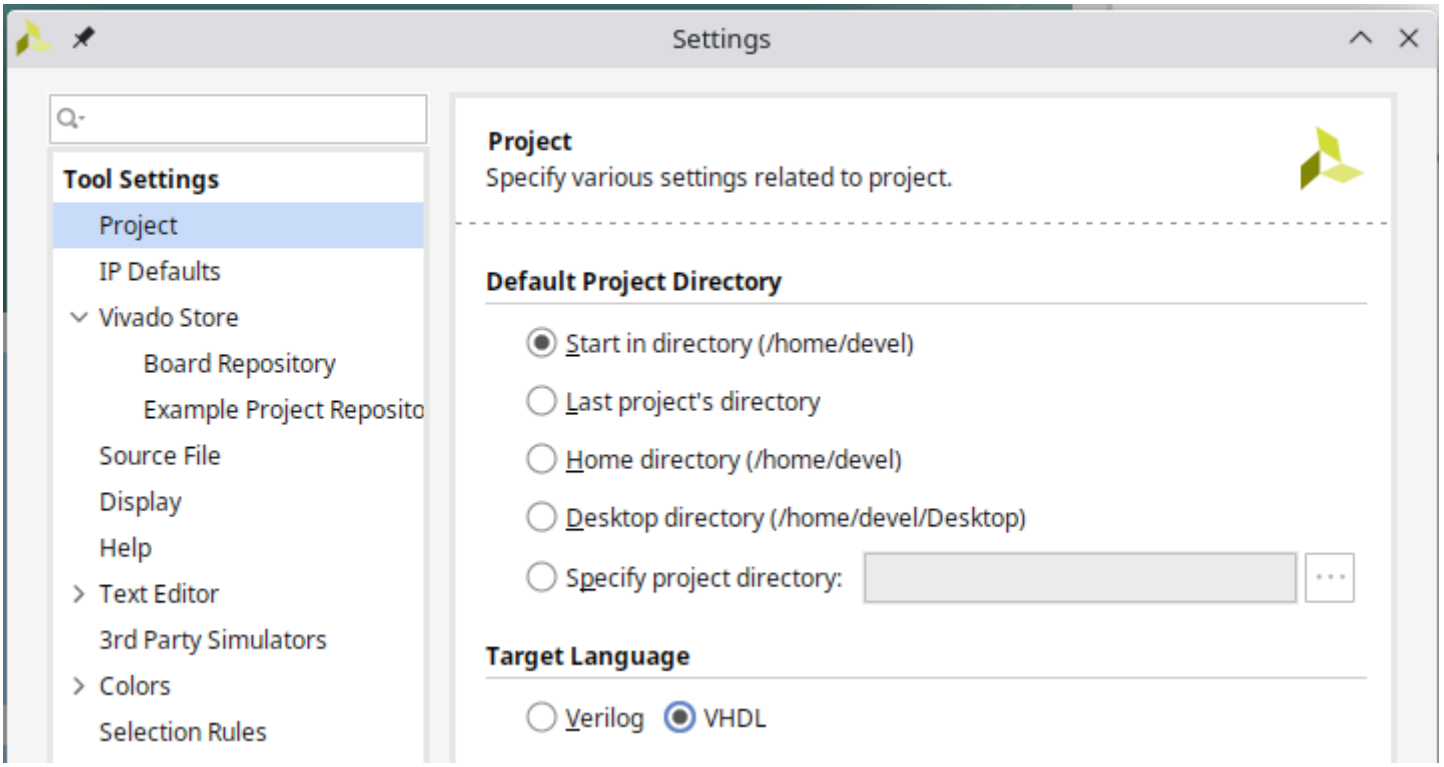
```
vivado
```

From main UI → *Tools* → *Settings*

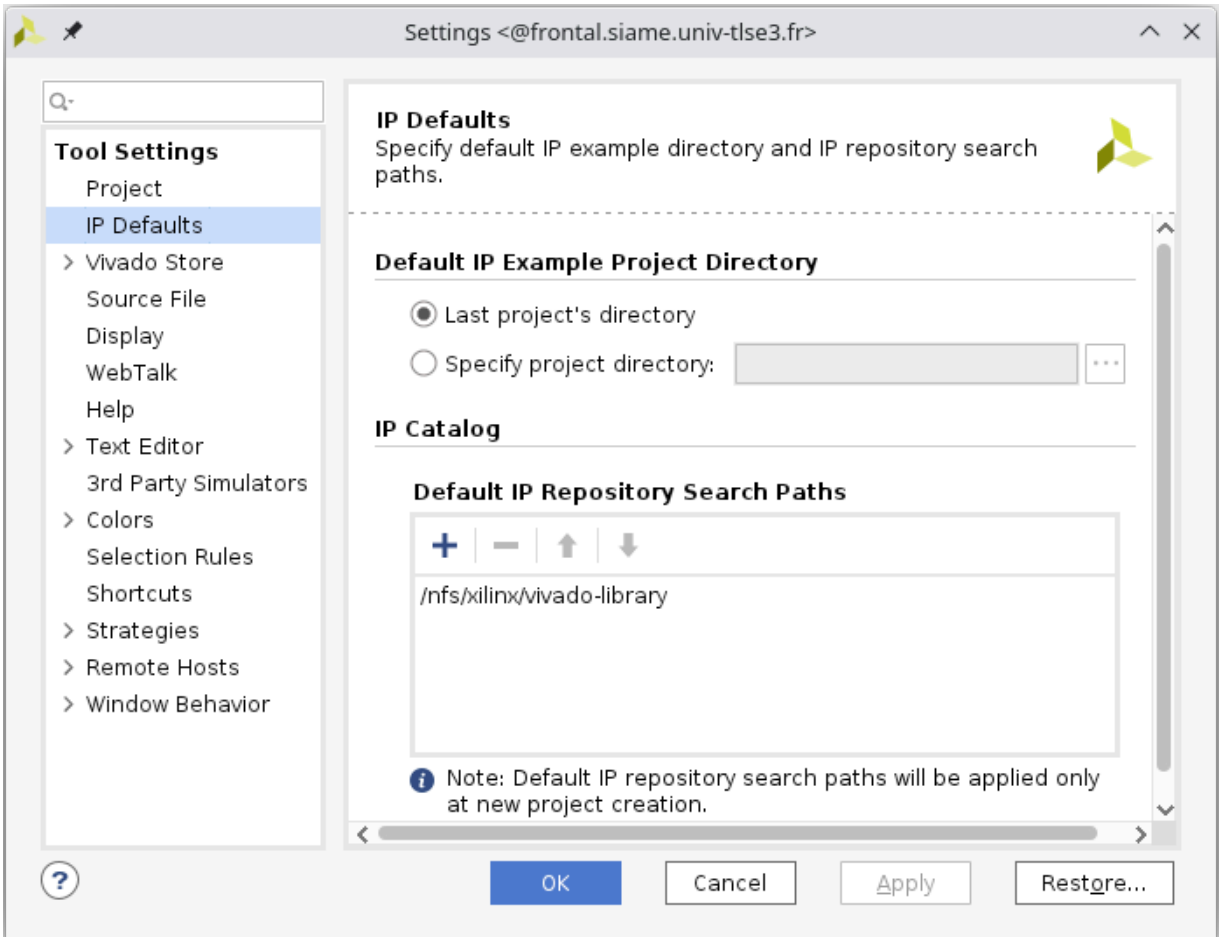


Project → *Target Language* → ***VHDL***

[1] IP stands for **I**ntellectual **P**roperty. ARM processors in Zynq devices are **Hard IPs** while others will be **Soft IPs**.



IP Defaults → IP catalog → add `vivado-libray` path



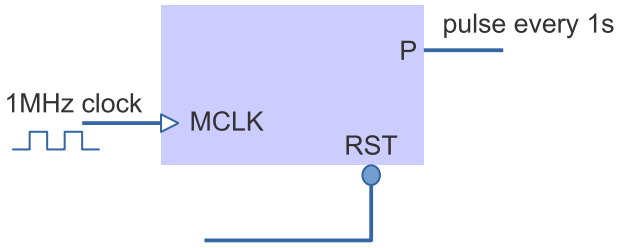
Practical exercises base files

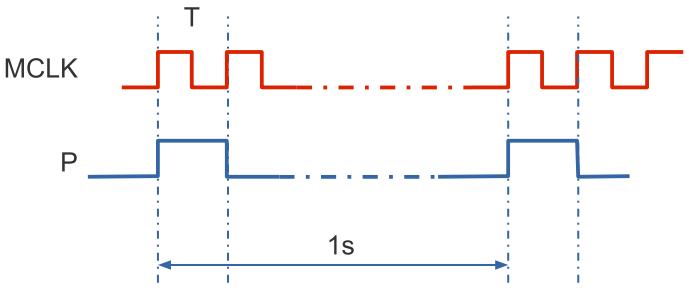
In order to ease things a bit, you can start from files available at

[M1] https://secil.univ-tlse3.fr/teaching/francois/UE-VHDL/M1/
[M2] https://secil.univ-tlse3.fr/teaching/francois/UE-VHDL/M2/

TP1 - Pulse generator

This first exercise will enable you to undertake a **behavioural** simulation of a simple component: a **pulse generator.**





As a first step, retrieve the following files

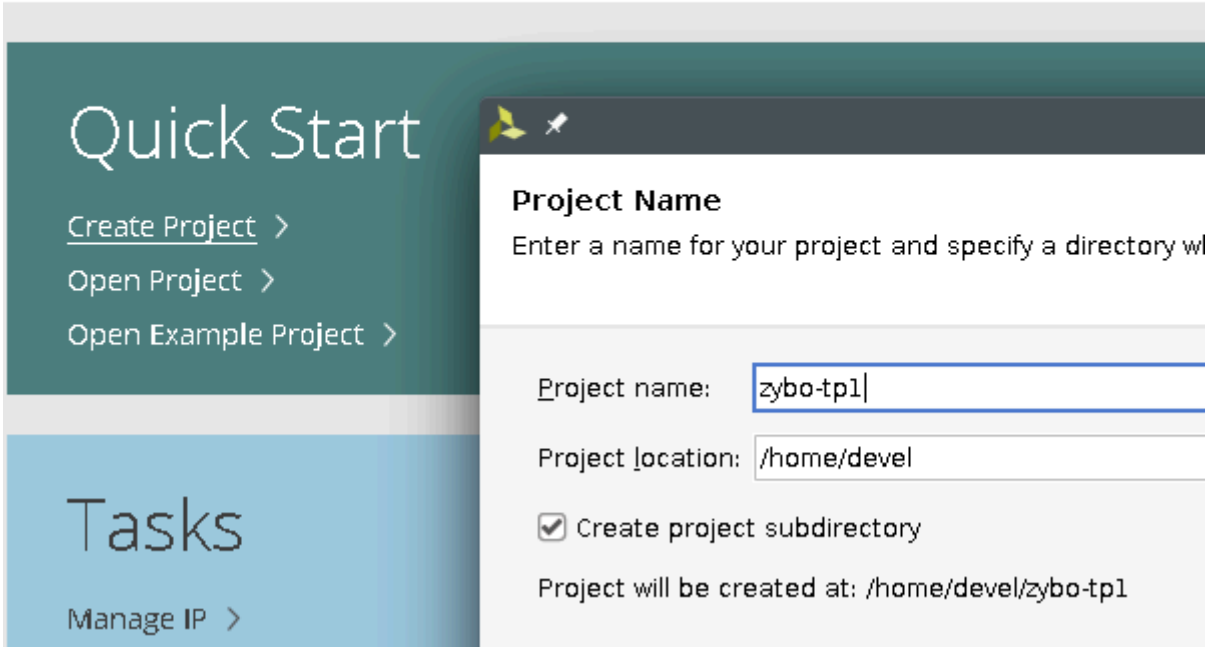
https://secil.univ-tlse3.fr/teaching/francois/UE-VHDL/M1/pulse_gen.vhd
https://secil.univ-tlse3.fr/teaching/francois/UE-VHDL/M1/test_pulse_gen.vhd

project setup

Launch the Vivado tools suite

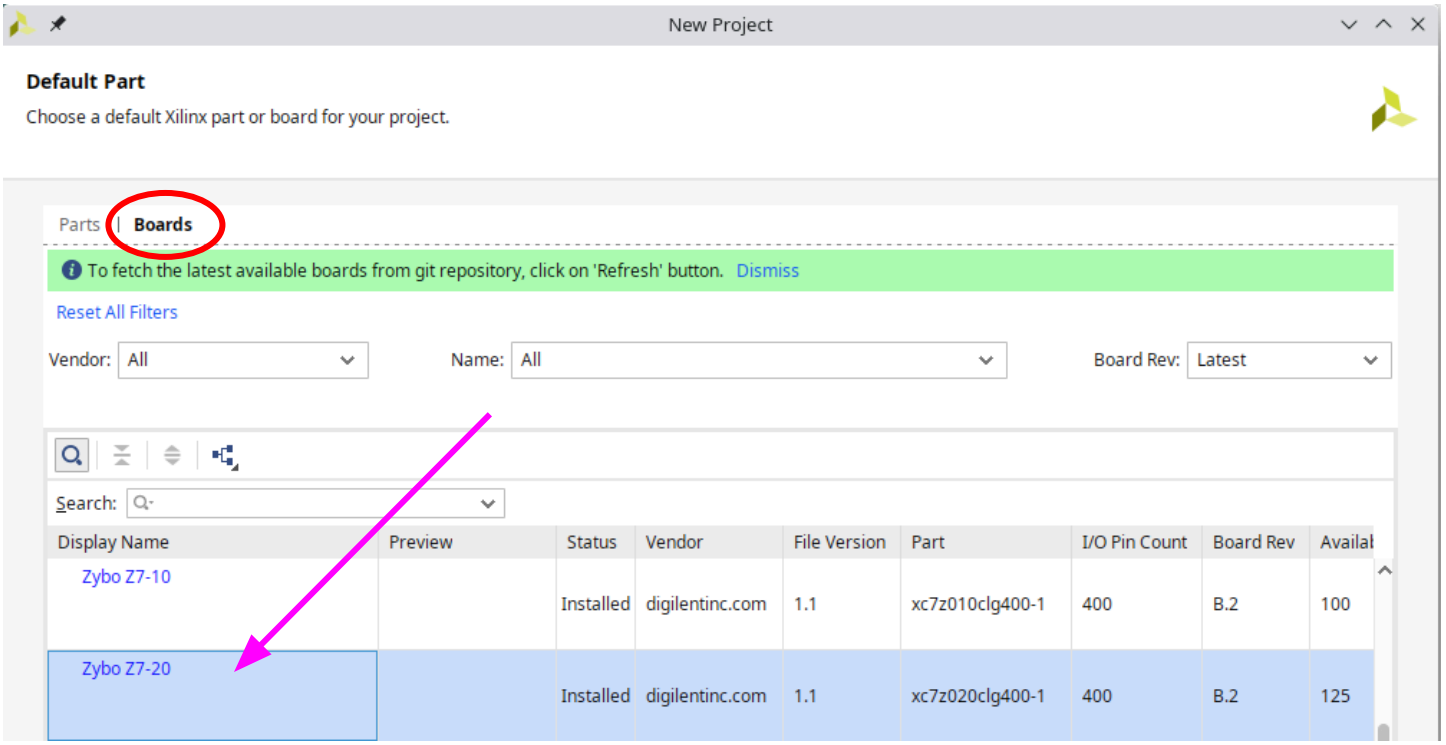
```
vivado
```

You create a first project named '`zybo-tp1`'

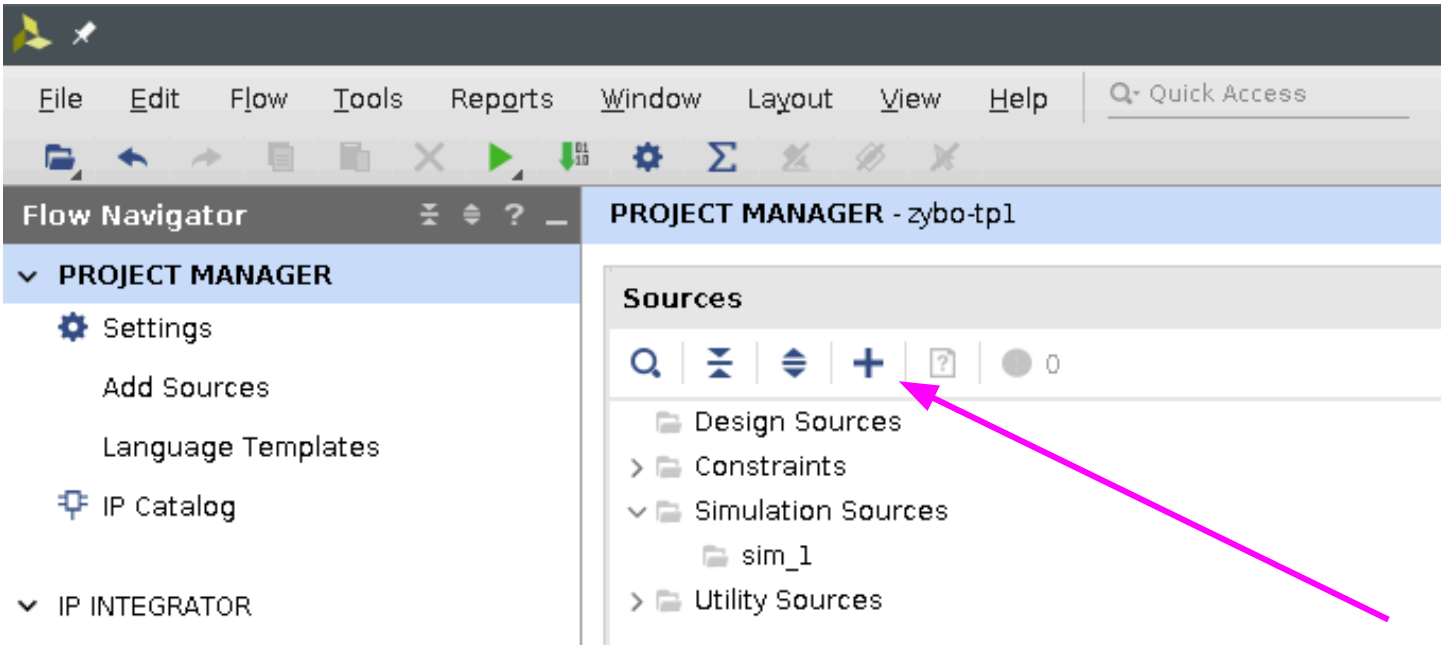


This is a RTL project but we'll add files later

... click on `boards` and select `Zybo-Z7-20`

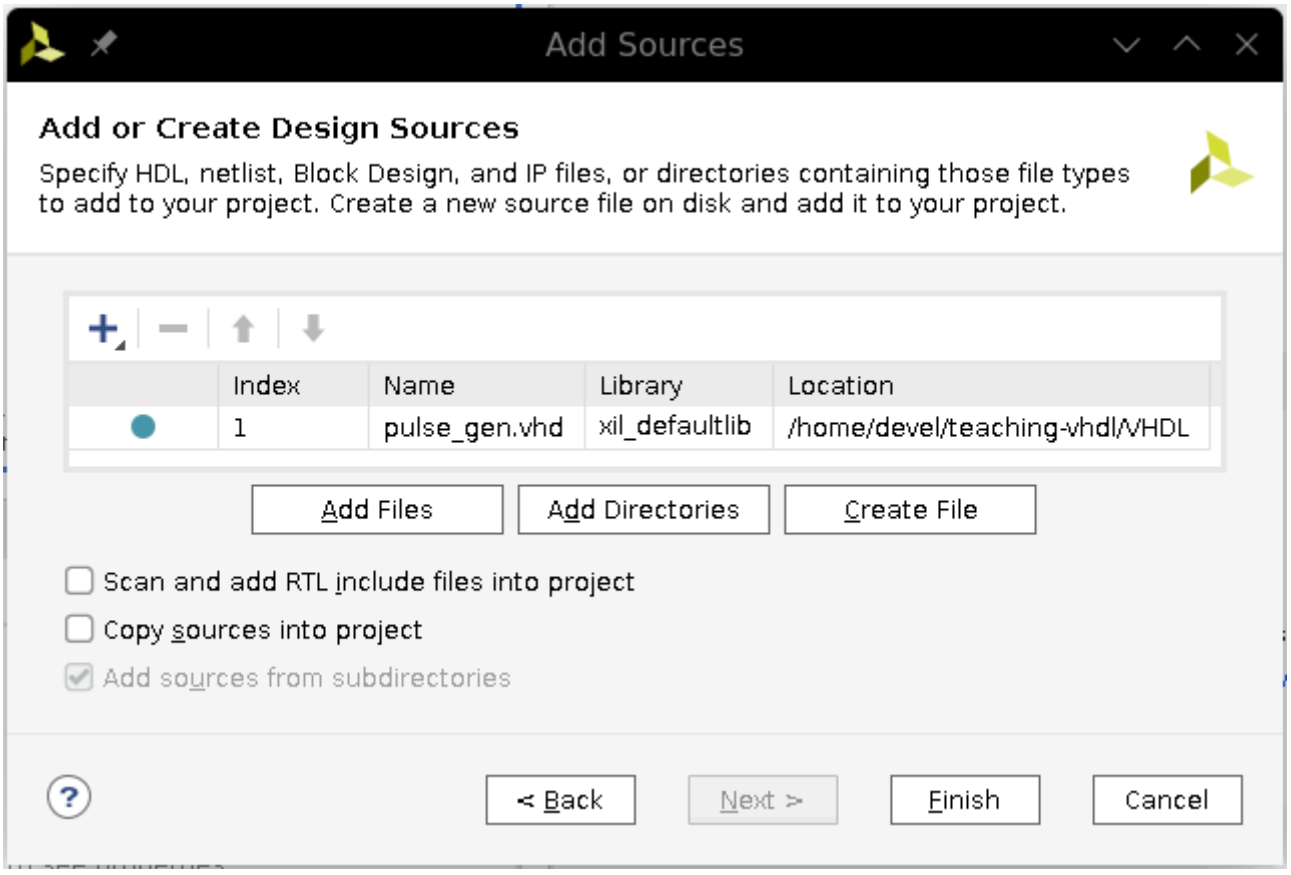


The project is now initialised, the main UI will appear and we'll start to add files



We'll start to add '`design sources`' files

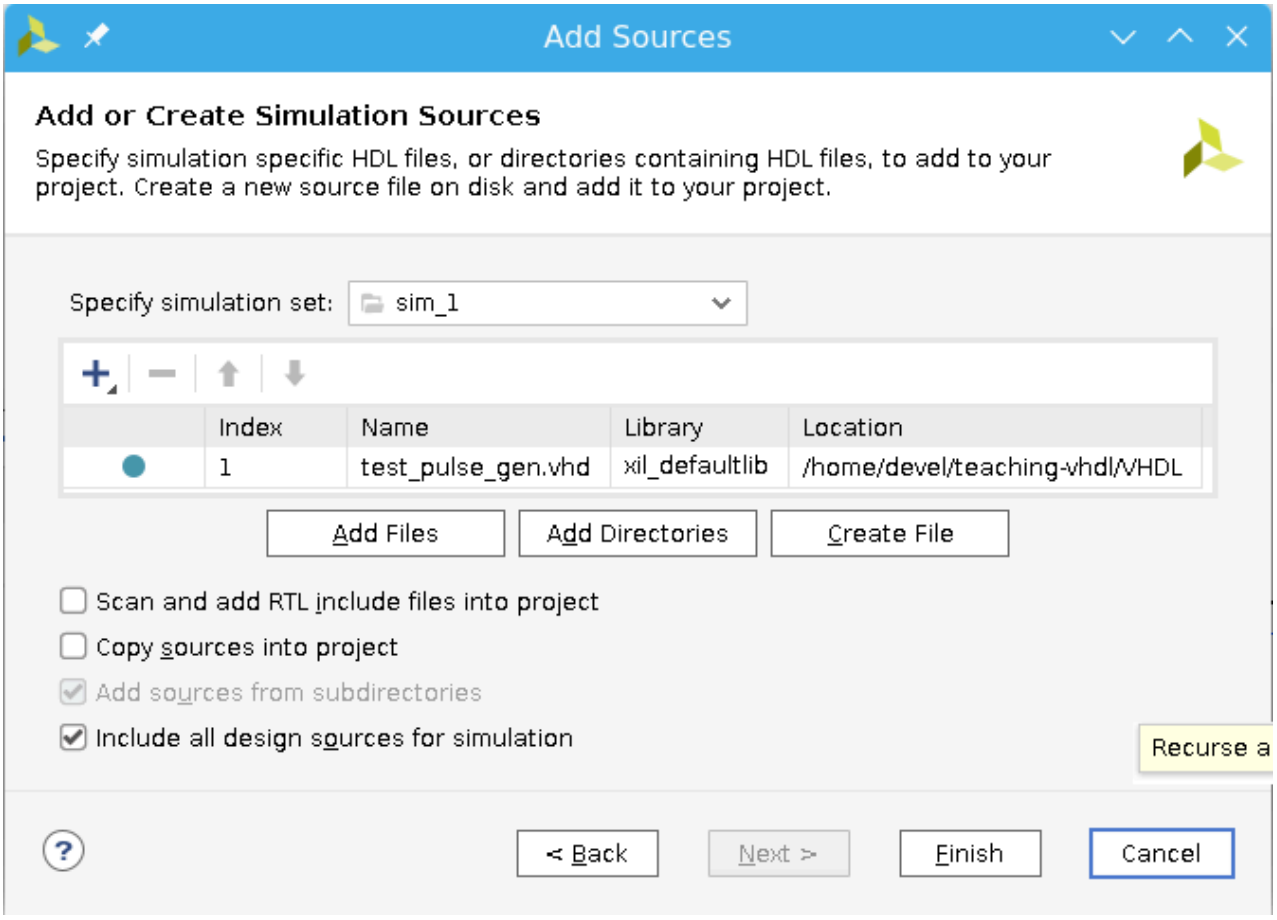
select '`pulse_gen.vhd`' with '`copy sources into project`' option **disabled**



there are some errors … quite normal since it's not finished

Now we'll add a simulation file in order to test our future pulse generator. Click on 'add file' in the project navigation bar and select '`simulation sources`':

select '`test_pulse_gen.vhd`' along with '`copy sources into project`' option **disabled**

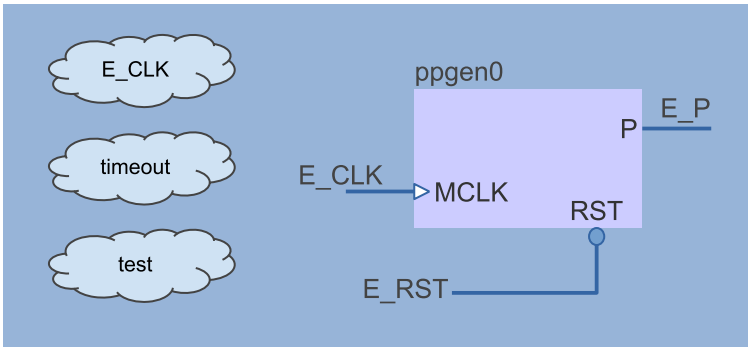


pulse_gen design

Our pulse generator will feature a **1MHz input clock**, an **asynchronous active low RST** line and will deliver a 1s output pulse. To achieve this, you need to complete the '`pulse_gen.vhd`' file.

The test bench file '`test_pulse_gen.vhd`' is a **test architecture** (i.e without I/O) that will define both a 1MHz internal clock, a timeout process along with another process that will feed signals on your `pulse_gen` component.

*Note: in order to clarify the simulation results, you'll set a pulse output every **ten** MCLK's rising edge.*

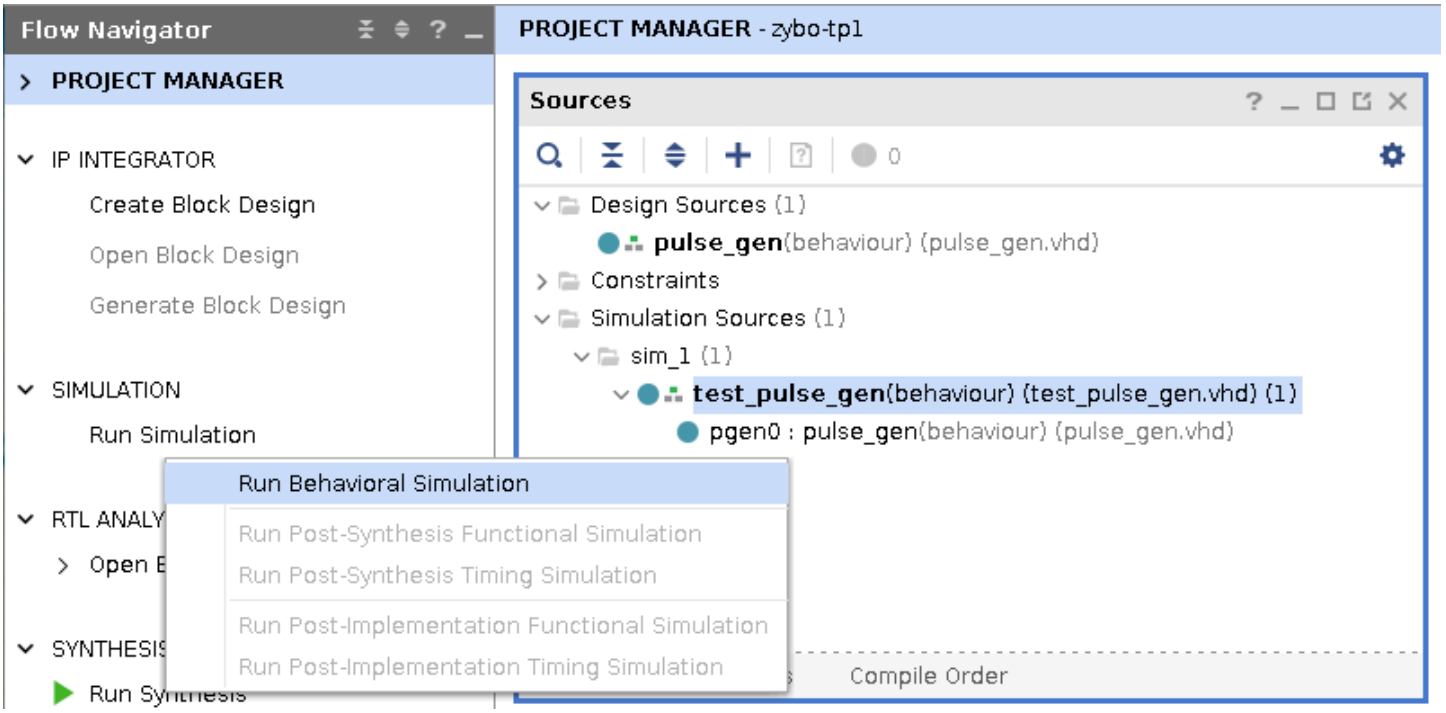


behavioural simulation

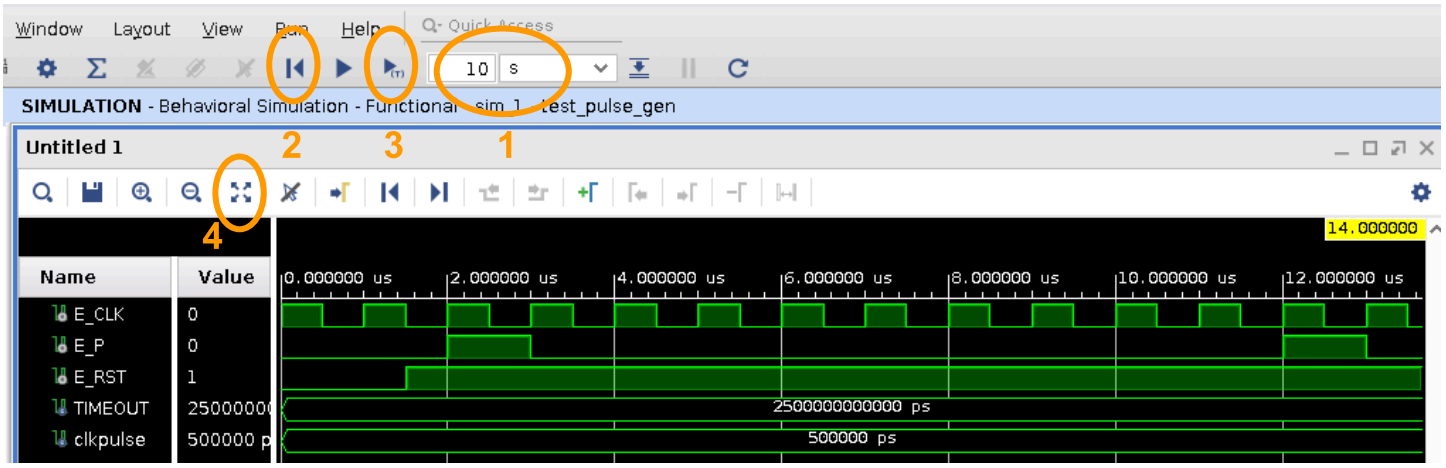
After having completed the '`pulse_gen.vhd`' file, it's now time to achieve your first simulation. It it worth mentioning that there exists 3 levels of simulation:

- **behavioural** → to test functionalities of your design,
- **post-synthesis** → inner components of your design are mapped on hardware resources, now you'll have propagation delays,
- **post-implementation** (i.e place and route) → post-synthesis + routing delays like in real life.

Hence we'll start with the first level of simulation: **behavioural**
On the vertical side panel, click '`Run Simulation`' and select '`Run Behavioural Simulation`'



You'll need to specify a **10s simulation time**, **restart simulation button** then **run for specified time** button and finally press **zoom fit** button.



Now … Why did you get such an '**ASSERTION ERROR**' in your test bench ??

>>> WELL DONE, you successfully ran your simulation test bench <<<

pulse_gen generic parameter(s)

You'll now add the **MAX_CPT** generic parameter to your `pulse_gen` component. It's default value will be set to **1E06**.

Update `pulse_gen` architecture.

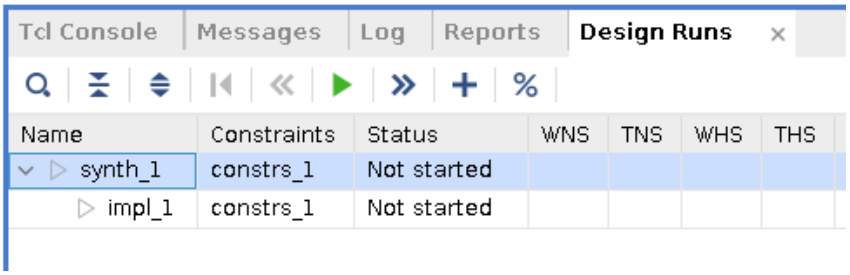
Update `test_pulse_gen.vhd` to apply **MAX_CPT=10** at the `ppgen0` instantiation, then **restart the behavioural simulation**.

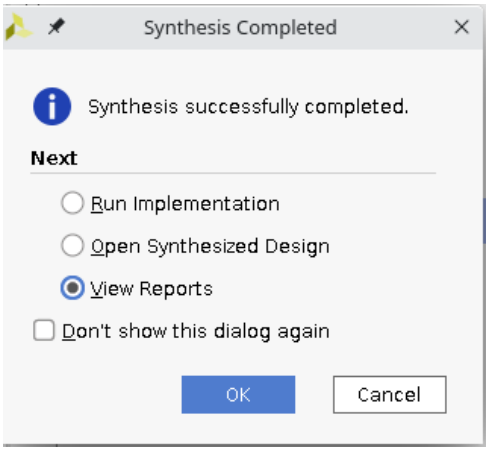
```
entity pulse_gen is
    generic (
        MAX_CPT : natural := 1E06
    );
    port (
        RST, MCLK: in std_logic;
        P : out std_logic
    );
end pulse_gen;
```

design synthesis

Well now synthesise our design ⇒ Vivado will instantiate hardware elements like flip-flops, logic cells etc to implement your design.

In the 'Design Runs' tab (lowest part of UI), click the green arrow and click OK not modifying default synthesis parameters.



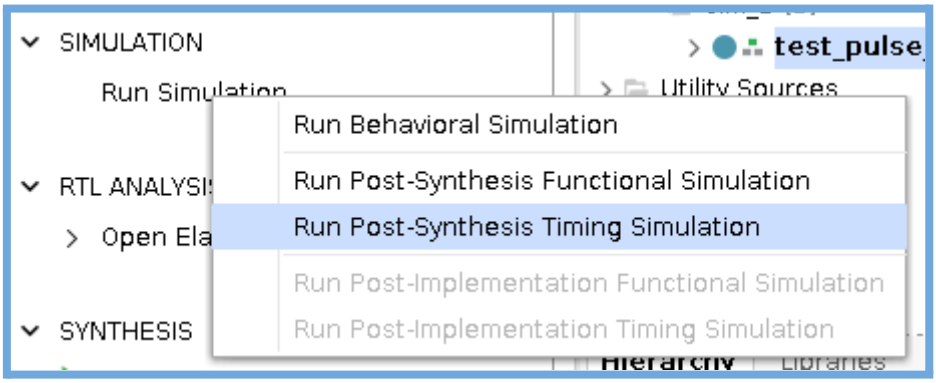


After your design has been successfully synthesised, select the 'View Reports' option.

```
+-------------------------+------+-------+------------+-----------+-------+
|        Site Type        | Used | Fixed | Prohibited | Available | Util% |
+-------------------------+------+-------+------------+-----------+-------+
| Slice LUTs*             |   28 |     0 |          0 |     17600 |  0.16 |
|   LUT as Logic          |   28 |     0 |          0 |     17600 |  0.16 |
|   LUT as Memory         |    0 |     0 |          0 |      6000 |  0.00 |
| Slice Registers         |   21 |     0 |          0 |     35200 |  0.06 |
|   Register as Flip Flop |   21 |     0 |          0 |     35200 |  0.06 |
|   Register as Latch     |    0 |     0 |          0 |     35200 |  0.00 |
| F7 Muxes                |    0 |     0 |          0 |      8800 |  0.00 |
| F8 Muxes                |    0 |     0 |          0 |      4400 |  0.00 |
+-------------------------+------+-------+------------+-----------+-------+
```

post-synthesis simulation

Now having your design synthesised, click on the 'Run Simulation' on the side panel and choose post-synthesis Timing Simulation



>>> What's happening ? What do you conclude ?? <<<

TP2 - Pulse generator synthesis

In the previous practical exercises TP1 - Pulse generator, we've not been able to simulate the synthesised design … because synthesis applied **default generic values** and thus removed the generic parameter !

Hence, we need to manage two issues:

- having our `pulse_gen` component synthesised with the requested generic value,
- remove **generic map** from `test_pulse_gen.vhd` since our **synthesised component** does not exhibit generic parameters

Note: it is worth mentioning that this issue only occurs with top-level designs having generic parameters.

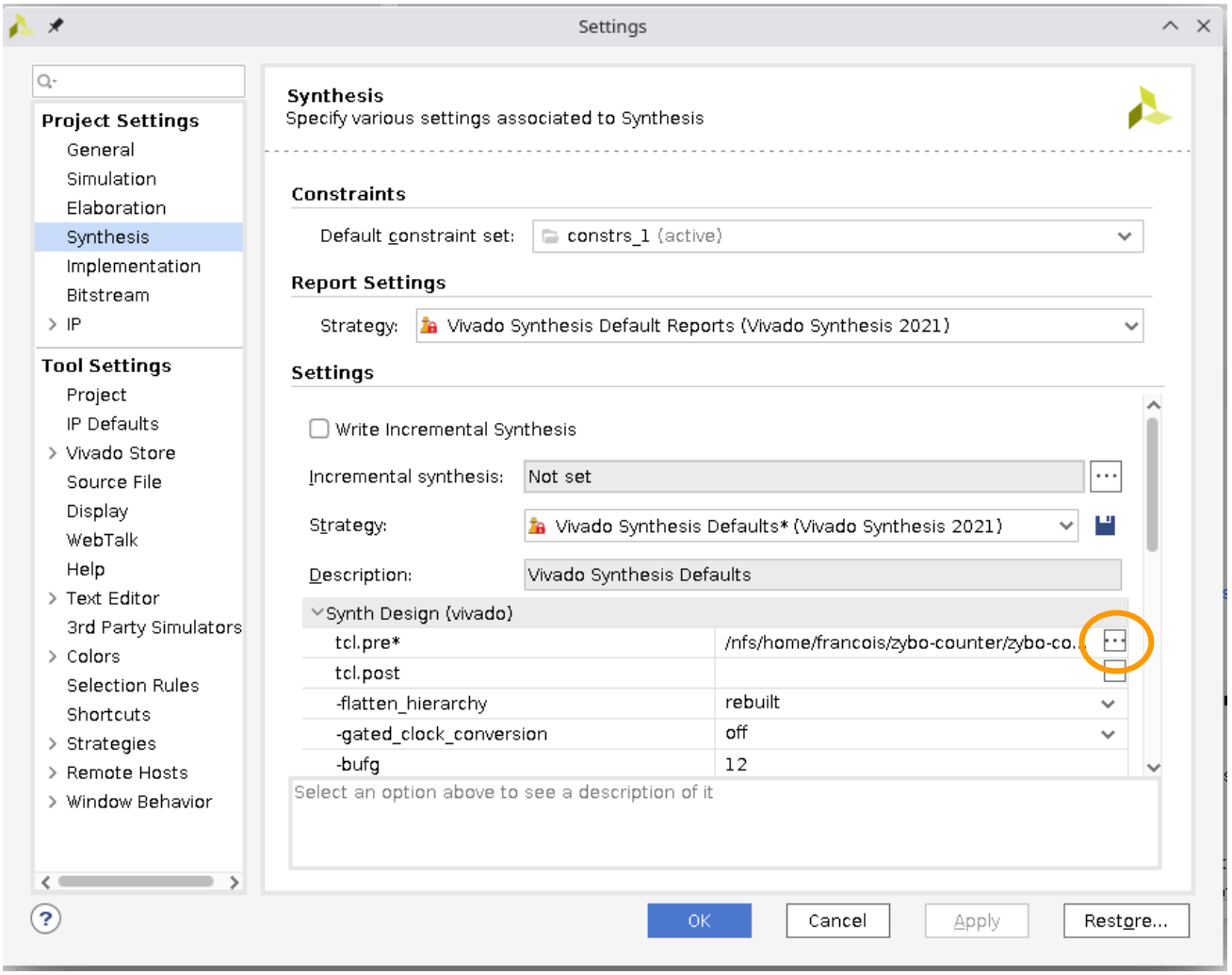
top-level synthesis and generic parameters

In order to give generic parameters values to top-level components, you'll need to create a TCL file at the root of your project:

- `pulse_gen_synth.pre.tcl`

```
# set generic parameter for synthesis
set_property generic {MAX_CPT=8} [current_fileset]
```

Settings → Project Settings → Synthesis → tcl.pre field



Then, this new

file will appear as an `utility source file`

New synthesis run will reports differently this time:

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	3	0	0	17600	0.02
LUT as Logic	3	0	0	17600	0.02
LUT as Memory	0	0	0	6000	0.00
Slice Registers	4	0	0	35200	0.01
Register as Flip Flop	**4**	0	0	35200	0.01
Register as Latch	0	0	0	35200	0.00
F7 Muxes	0	0	0	8800	0.00
F8 Muxes	0	0	0	4400	0.00

Ref Name	Used	Functional Category
FDCE	4	Flop & Latch
LUT3	3	LUT
LUT1	2	LUT
IBUF	2	IO
OBUF	1	IO
BUFG	1	Clock

FDCE: D flip-flop with Clock Enable and Asynchronous Clear
FDPE: D flip-flop with Clock Enable and Asynchronous Preset
FDSE: D flip-flop with Clock Enable and Synchronous Set
FDRE: D flip-flop with Clock Enable and Synchronous Reset
[ug901] Xilinx Vivado design suite user guide: synthesis

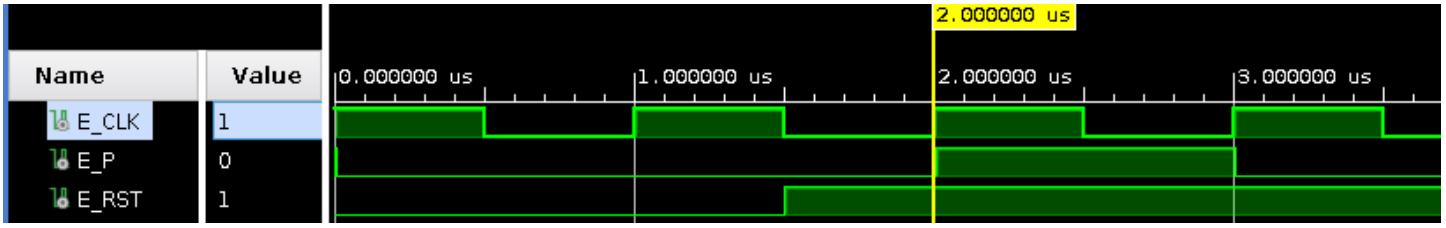
comment-out generic map

- `test_pulse_gen.vhd`

```
-------------------------------------------------
-- instantiation et mapping du composant registres
pgen0 : entity work.pulse_gen(behaviour)
--                  generic map (10)
                    port map (MCLK => E_CLK,
                          RST => E_RST,
                          P => E_P);
```

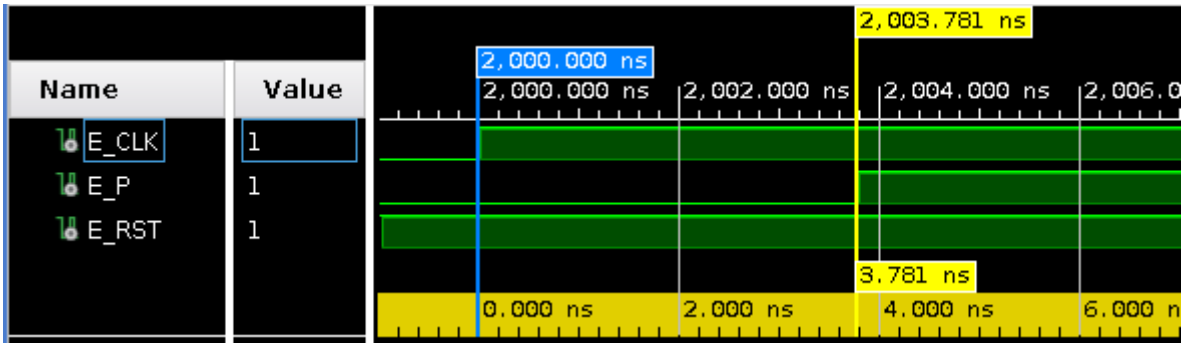
post-synthesis timing simulation

Restart the post-synthesis timing simulation, your output ought to change according to MAX_CPT value.



>>> Do you notice something ?? <<<

Propagation delays now appear between E_CLK rising edge and E_P output update:



⇒ What's the **MCLK's maximum frequency** of your pulse_gen component on this chip ?

$$f = \frac{1}{T}$$

pulse_gen with synchronous RST

This time, we'd like you to modify your `pulse_gen` architecture to implement a **synchronous** reset.

```
+----------+------+---------------------+
| Ref Name | Used | Functional Category |
+----------+------+---------------------+
| FDRE     |    4 |         Flop & Latch |
| LUT4     |    2 |                  LUT |
| IBUF     |    2 |                   IO |
| OBUF     |    1 |                   IO |
| LUT3     |    1 |                  LUT |
| LUT2     |    1 |                  LUT |
| LUT1     |    1 |                  LUT |
| BUFG     |    1 |                Clock |
+----------+------+---------------------+
```

FDCE: D flip-flop with Clock Enable and Asynchronous Clear
FDPE: D flip-flop with Clock Enable and Asynchronous Preset
FDSE: D flip-flop with Clock Enable and Synchronous Set
FDRE: D flip-flop with Clock Enable and Synchronous Reset
see [ug901] Xilinx Vivado design suite user guide: synthesis

What do you notice on chronogrammes ? regarding the instantiated resources ??

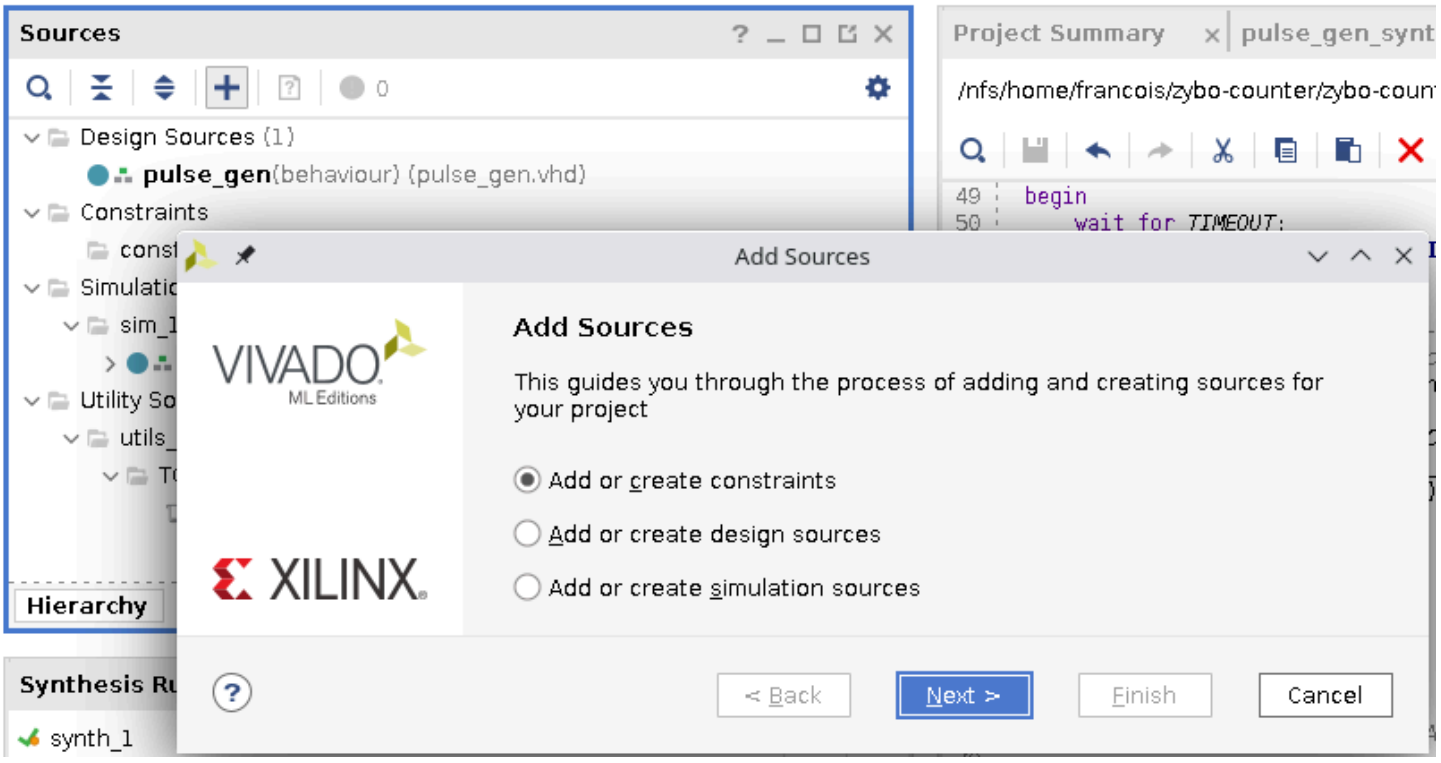
TP3 - Pulse generator constraints

In the previous exercises, we simulated our `pulse_gen` component **without any timing constraints** to validate against … hence the `unconstrained paths` reported by the Vivado's **Report Timing summary** (left toolbar *Synthesis → Report Timing Summary*)
see [pluse_gen unconstrained paths](#) on next page

Timing constraints

A *constraint file* enables you to express timing requirements against a clock frequency your design is supposed to perform with. In subsequent practical exercises, you'll discover that this constraint file is also used for I/O Planning (i.e which IO ports your design makes use of). These files exhibit a xdc extension.

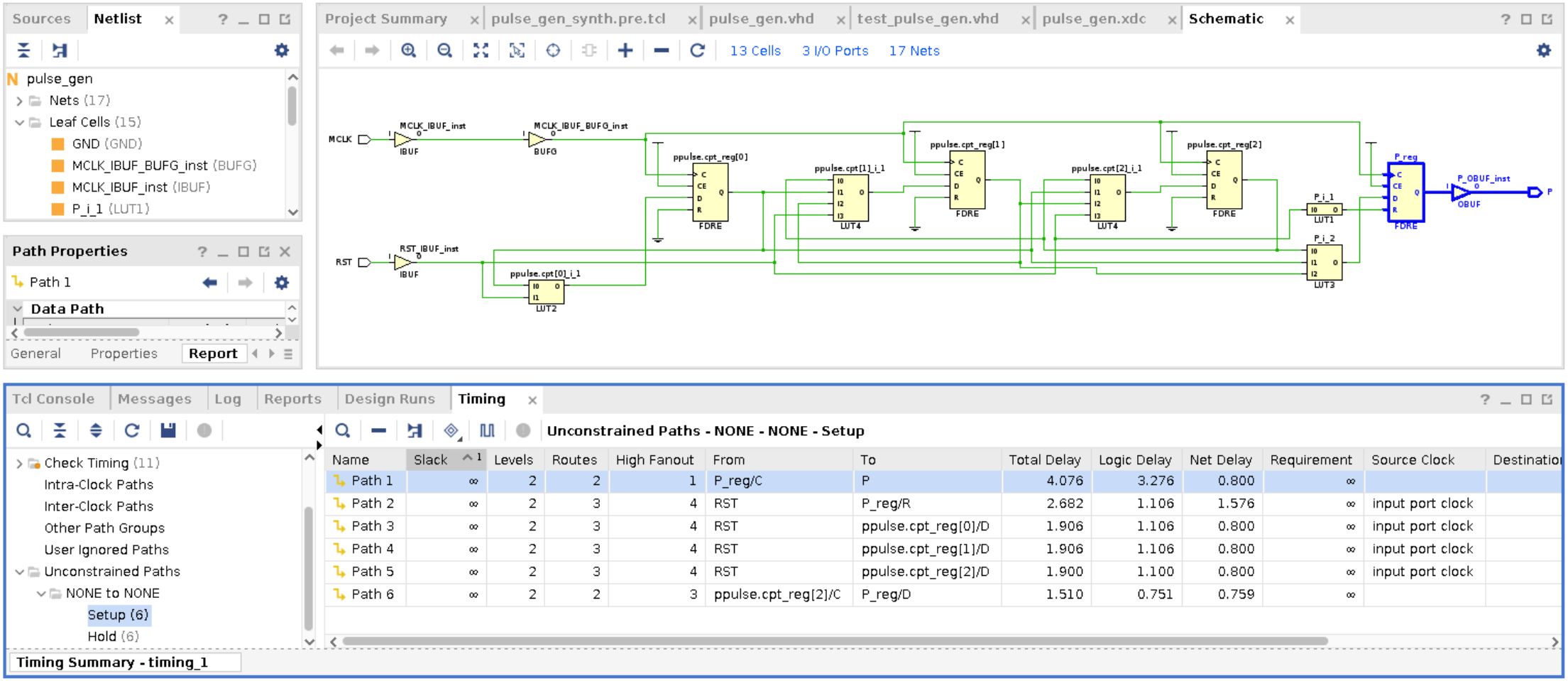
You'll now create a constraint file. To achieve this, click on '+' in `Sources` area and select `add or Create constraints` → create a file named '`pulse_gen.xdc`'



Your design now features a new constraint file named '`pulse_gen.xdc`'.

Instead of trying to fill it on yourself, we'll make use of the *Synthesis* → ***Constraints Wizard***

pluse_gen unconstrained paths

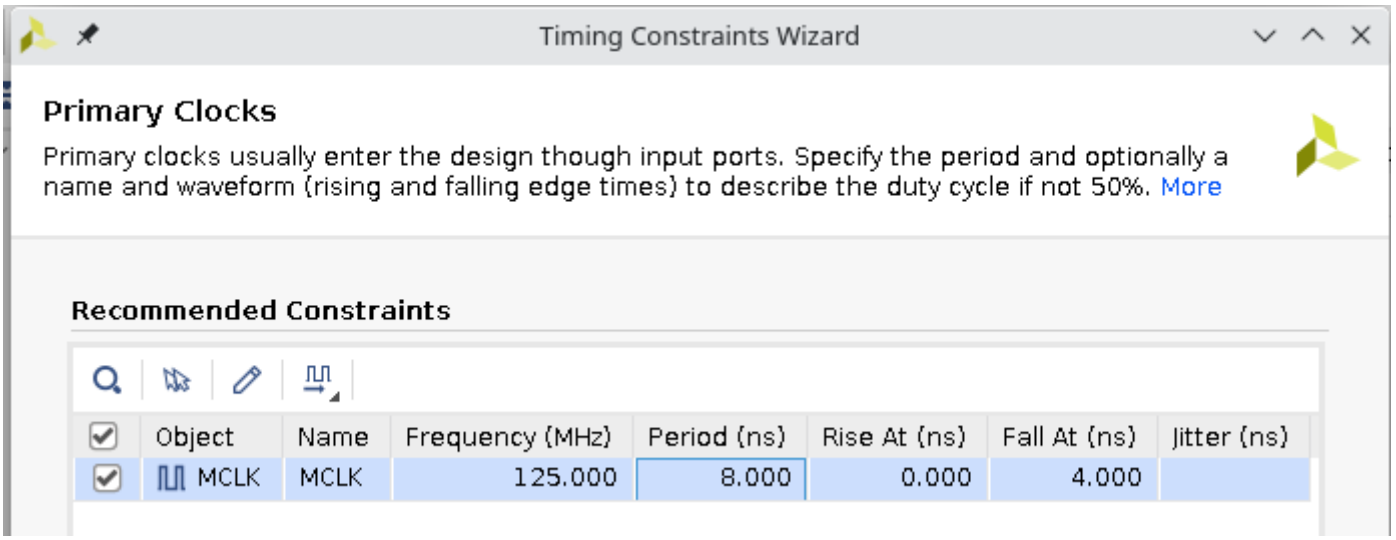


Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination
Path 1	∞	2	2	1	P_reg/C	P	4.076	3.276	0.800	∞		
Path 2	∞	2	3	4	RST	P_reg/R	2.682	1.106	1.576	∞	input port clock	
Path 3	∞	2	3	4	RST	ppulse.cpt_reg[0]/D	1.906	1.106	0.800	∞	input port clock	
Path 4	∞	2	3	4	RST	ppulse.cpt_reg[1]/D	1.906	1.106	0.800	∞	input port clock	
Path 5	∞	2	3	4	RST	ppulse.cpt_reg[2]/D	1.900	1.100	0.800	∞	input port clock	
Path 6	∞	2	2	3	ppulse.cpt_reg[2]/C	P_reg/D	1.510	0.751	0.759	∞		

Synthesis Constraints Wizard

The first time you click on this feature, it will ask you to select a target (i.e which XDC file) ⇒ pulse_gen.xdc
The second time, the wizard will start :)

In this application case, we'll only focus on the MCLK constraints. Set MCLK as being a 125MHz clock then *skip to finish*.

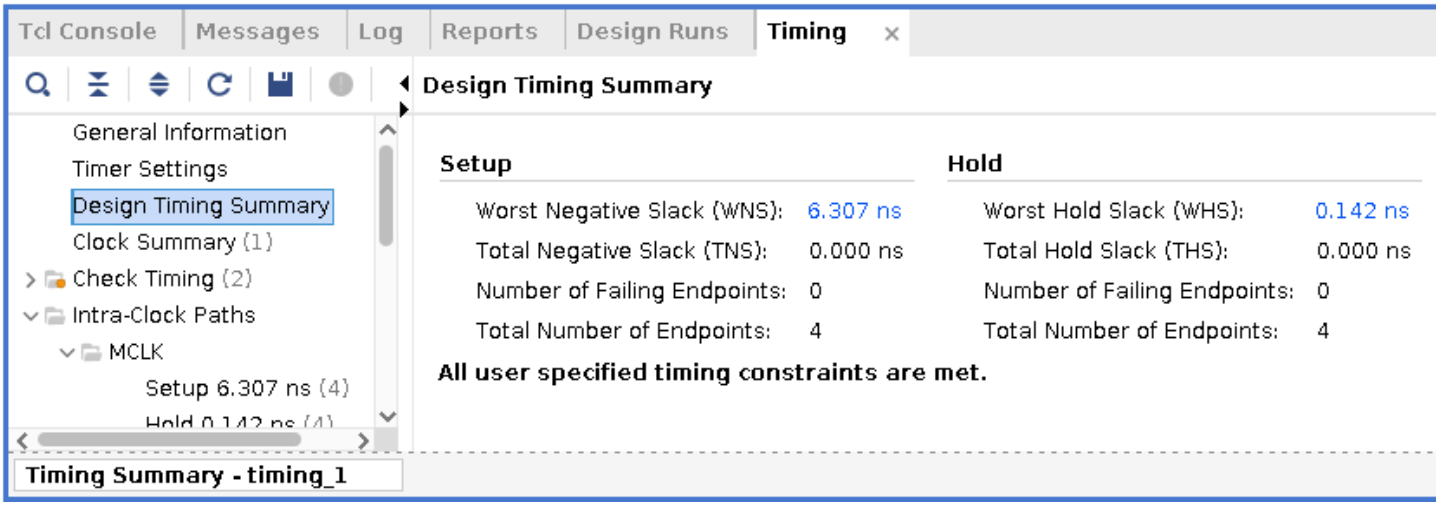


This will lead to a constraint rule added to your `pulse_gen.xdc` :

```
# Master Clock timing constraint
create_clock -period 8.000 -name MCLK -waveform {0.000 4.000} [get_ports MCLK]
```

Note: this rule DOES NOT DEFINE a clock, instead, it specifies a clk our system is supposed to comply with.

Now you restart *Synthesis → Report Timing summary*



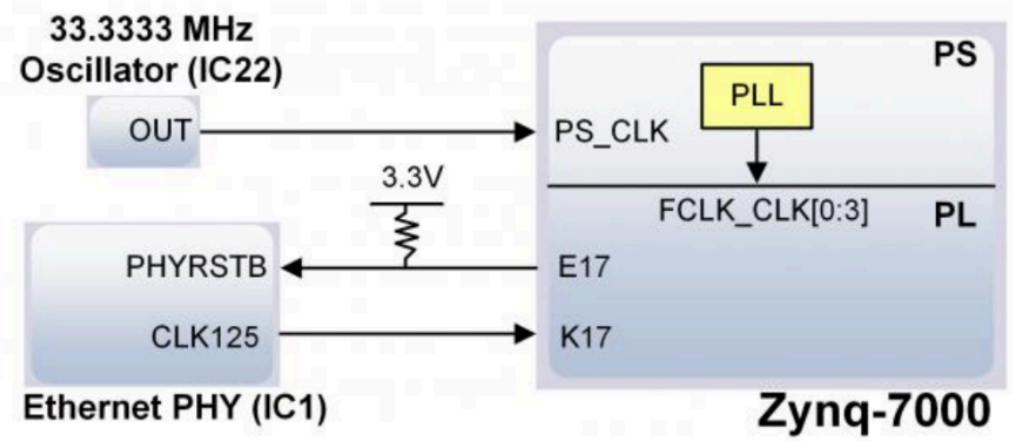
>>> Slack OUGHT to be **positive <<<**

Setup slack: Required time - Arrival time → delay before next CLK front that will memorize the data
Hold slack: Arrival time - Required time → delay the data change after the CLK front.

Introduction to slack and others timing considerations in FPGAs
http://www.ece.utep.edu/courses/web5375/Notes_files/ee5375_timing_fpga.pdf

Implementation Constraints Wizard

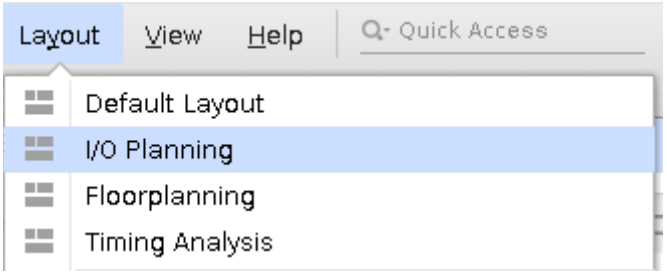
Why did we choose 125MHz for MCLK in the previous example ?
That's because the Zybo board features an independent PL_CLK at 125MHz coming from the onboard Ethernet IC.



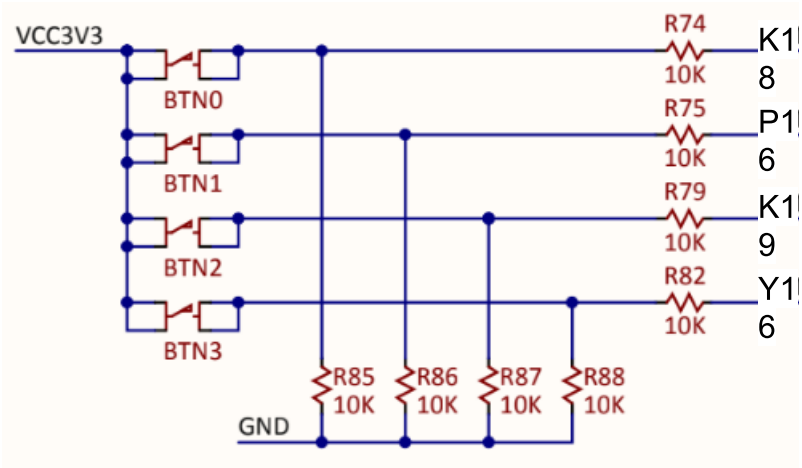
As shown in the upper figure, you can see that the **K17** pin is used as the PL_CLK input. Additionally, **E17** pin will be able to reset the Ethernet IC hence suspending the 125MHz clock.

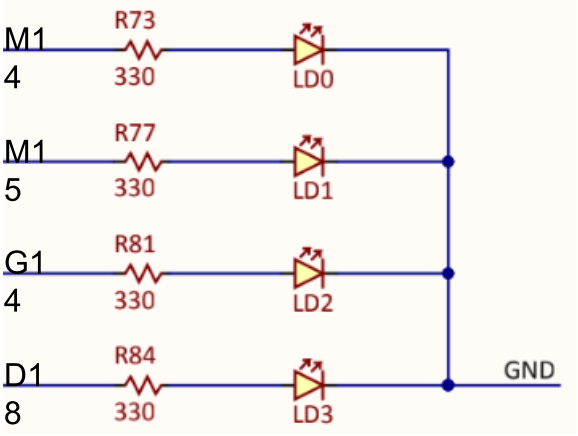
Up To now, we've been through the synthesis process that translates your VHDL code to instantiated physical elements of our FPGA. Next step is to map IO pads to our design ⇒ IO planning

Implementation → Run Implementation , then select *layout → IO planning* in the Vivado upper tool bar.

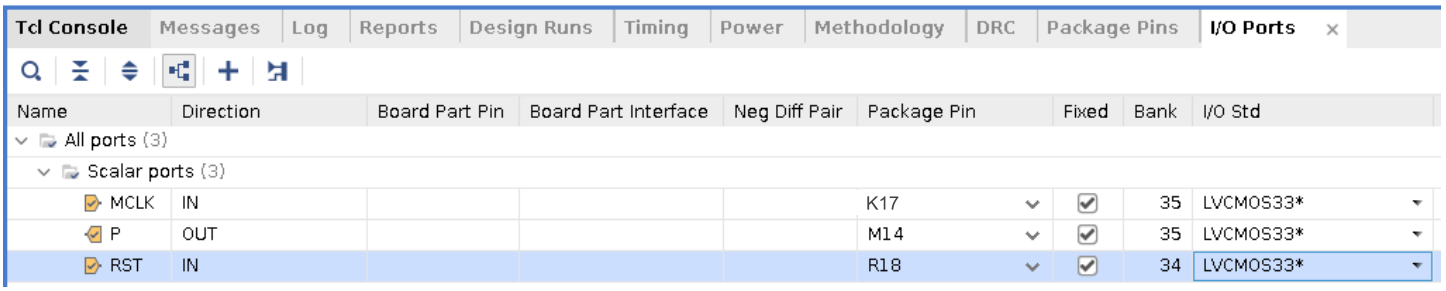


Below is a detailed overview of buttons and leds available on the Zybo board (switches not shown).





Now you'll affect Zybo GPIO to our `pulse_gen` ports according to the following setup:



When finished, save as **CTRL + S** → Vivado will ask you whether it ought to save these constraints in the default constraints file (i.e `pulse_gen.xdc`) → yes !

In the end, your `pulse_gen.xdc` file ought to looks like this:

```
# Master Clock timing constraint
create_clock -period 8.000 -name MCLK -waveform {0.000 4.000} [get_ports MCLK]

set_property PACKAGE_PIN K17 [get_ports MCLK]
set_property IOSTANDARD LVCMOS33 [get_ports MCLK]

set_property PACKAGE_PIN M14 [get_ports P]
set_property IOSTANDARD LVCMOS33 [get_ports P]

set_property PACKAGE_PIN R18 [get_ports RST]
set_property IOSTANDARD LVCMOS33 [get_ports RST]
```

… a more elegant way is to make use of the `'dict'` feature: `pulse_gen.xdc`

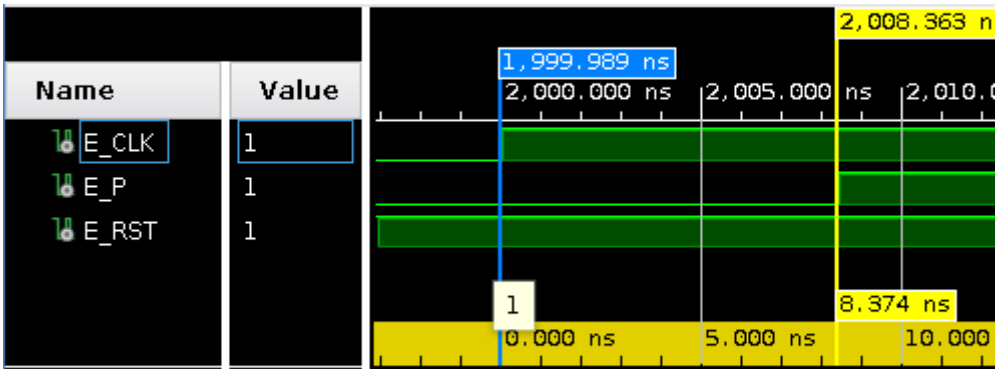
```
# Master Clock timing constraint
set_property -dict { PACKAGE_PIN K17   IOSTANDARD LVCMOS33 } [get_ports MCLK];
create_clock -period 8.000 -name MCLK -waveform {0.000 4.000} [get_ports MCLK];

# Reset (warning pull down resistors on board)
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports RST];

# P output
set_property -dict { PACKAGE_PIN M14   IOSTANDARD LVCMOS33 } [get_ports P];
```

It's now time to restart Synthesis + Implementation :)

Full timings simulation

This time, you'll select:
Simulation
→ *Run Simulation*
→ *Post-Implementation timing simulation*

What do you observe ??



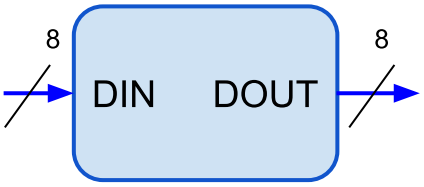
[master2] Bitstream generation and hardware download

Ultimate step that will generate the `.bit` file. This file will get downloaded into the FPGA's config RAM that will organise all resources as specified in synthesis and implementation steps. This file will get downloaded through JTAG (usb prog port) (*hint: change the pulse duration to be able to see it!*)

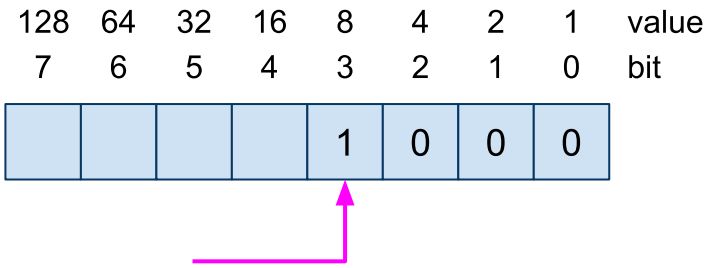
TP4 - Synthesizable Log2 function

This time, we'd like you to write a **synthesizable** Log2 component that will feature the following generic parameter(s):

- **BUS_WIDTH** : size of input and output buses with **8** as default value



log2(1) = 0
log2(2) = 1
log2(4) = 2
log2(8) = 3



log2 @ cpu_package

Once you retrieved the '`cpu_package.0.vhd`' file, you'll find definition for several log2 (overloaded) functions:

```
-- fonction log2
--          calcule le logarithme base2 d'un entier naturel, ou plus exactement
--          renvoie le nombre de bits nécessaires pour coder une valeur
function log2 (I: in natural) return natural;
function log2 (vI: in std_logic_vector) return std_logic_vector;
```

Your first task will be to implement a synthesizable version of the log2 function in the package body part of the `cpu_package` file.

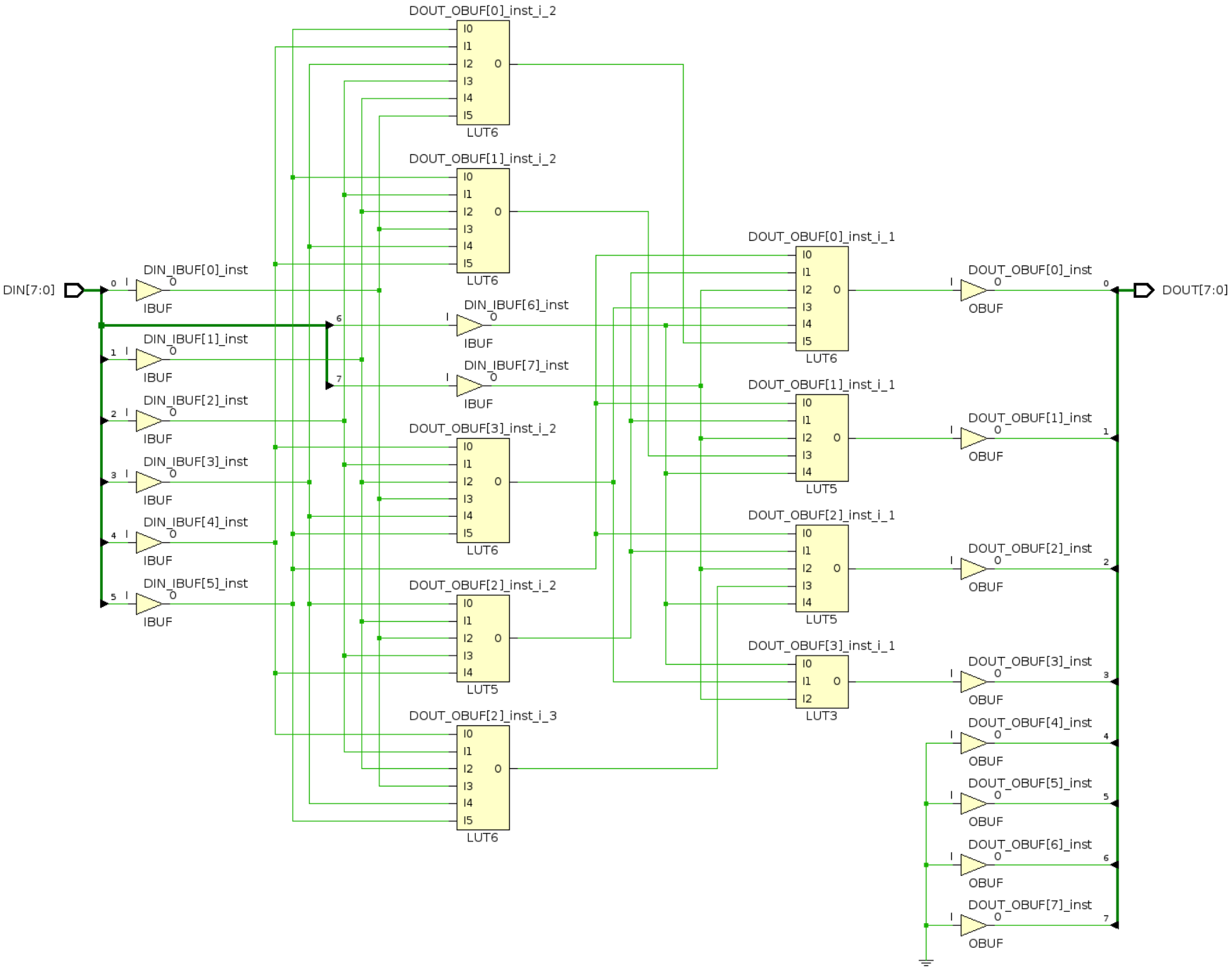
Run a behavioral simulation by means of the '`test_log2.vhd`' file.

log2 hardware component

As you noticed, your `log2` function exhibits **unconstrained** `std_logic_vector` both as input and output. Synthesis can manage this situation by dynamically selecting the proper bus size through component instantiation.
⇒ complete the `'log2_hw.vhd'` file with the generic parameter **BUS_WIDTH**; your architecture will make use of your previously written log2 function.

Undertake a **post-synthesis timing** simulation.

Next step: have a closer look at the schematic of the resulting synthesis: what do you notice on output ??

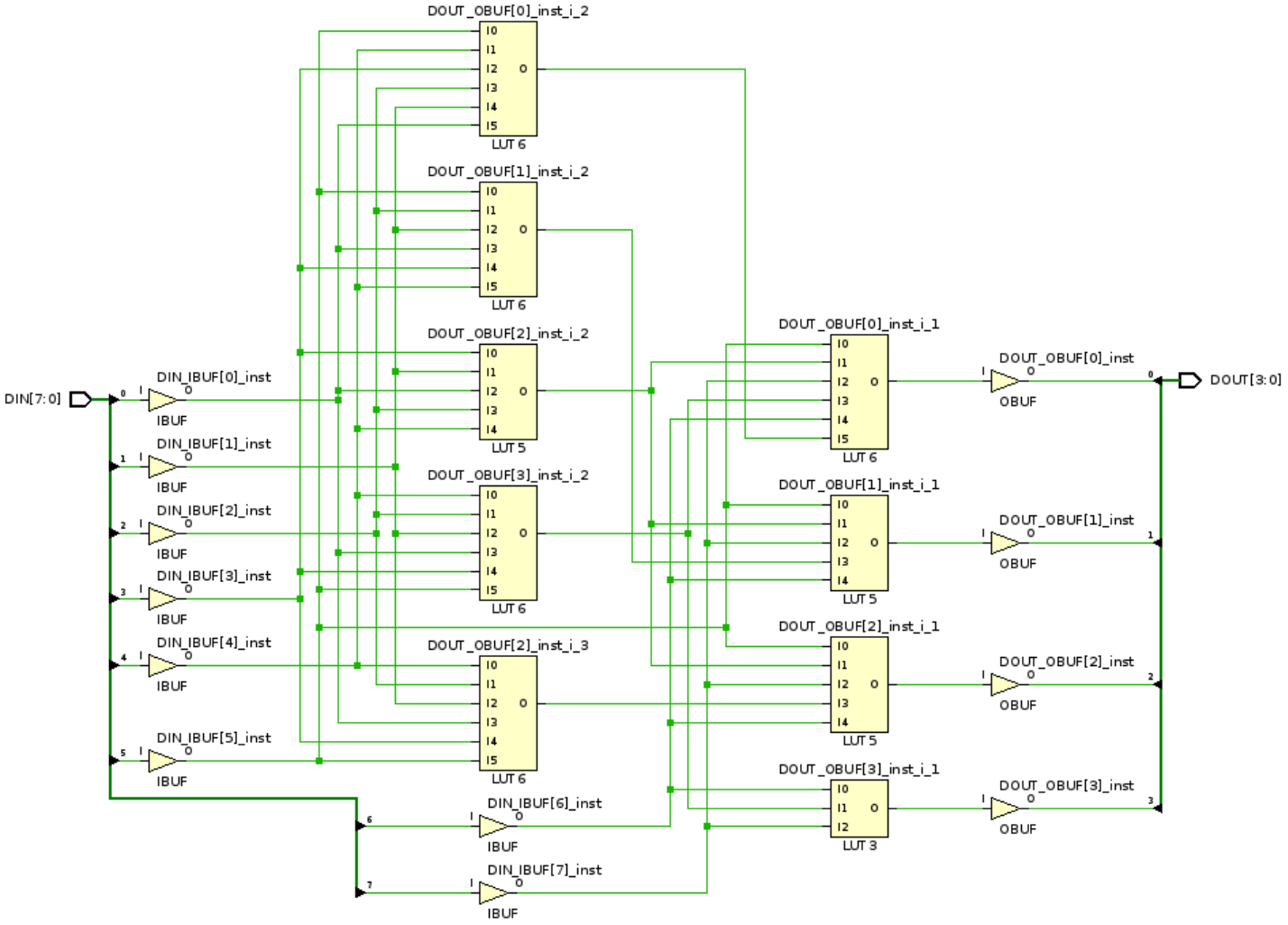

DIN[7:0]
DIN_IBUF[0]_inst
DIN_IBUF[1]_inst
DIN_IBUF[2]_inst
DIN_IBUF[3]_inst
DIN_IBUF[4]_inst
DIN_IBUF[5]_inst
DIN_IBUF[6]_inst
DIN_IBUF[7]_inst
IBUF
DOUT_OBUF[0]_inst_i_2
DOUT_OBUF[1]_inst_i_2
DOUT_OBUF[3]_inst_i_2
DOUT_OBUF[2]_inst_i_2
DOUT_OBUF[2]_inst_i_3
LUT6
DOUT_OBUF[0]_inst_i_1
DOUT_OBUF[1]_inst_i_1
DOUT_OBUF[2]_inst_i_1
LUT5
DOUT_OBUF[3]_inst_i_1
LUT3
DOUT_OBUF[0]_inst
DOUT_OBUF[1]_inst
DOUT_OBUF[2]_inst
DOUT_OBUF[3]_inst
DOUT_OBUF[4]_inst
DOUT_OBUF[5]_inst
DOUT_OBUF[6]_inst
DOUT_OBUF[7]_inst
OBUF
DOUT[7:0]

[master2] optimized log2 component

In order to overcome the issue related to the wrong size of the log2 component's output, you'll need to:

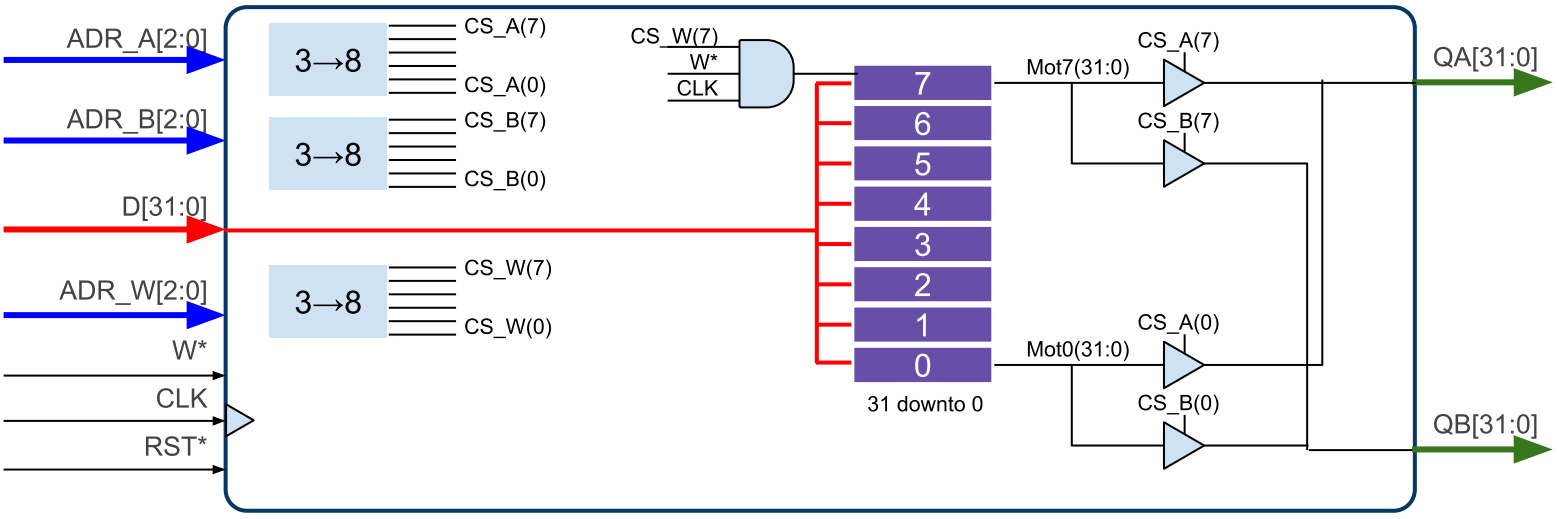
- reshape output and subsequents to `std_logic_vector(log2(BUS_WIDTH)-1 downto 0 );`
- import your `log2` function from `cpu_package` as a new `myLog2` function to settle in the declarative part of your log2 component's architecture.

Undertake a **post-synthesis timing** simulation and have a look at the generated schematic that ought to looks like this later:



TP5 - Registers bank for Risc processor

We'd like you to create a bank of 8 registers 32 bits featuring dual read access. One notable point is that reading register 0 **always** sends back value 0.



Signals featuring **'*'** means that they are active low. ADR_A and ADR_B addresses buses lead to asynchronous reading on ports QA and QB respectively.
RST* is synchronous and leads to all registers filled with 0. Write operations through ADR_W and active W* signal will be undertaken on the rising edge of CLK.

generic parameters

- **DBUS_WIDTH** internal words and data buses size, default to **32**
- **ABUS_WIDTH** size of all addresses buses, default to **5**

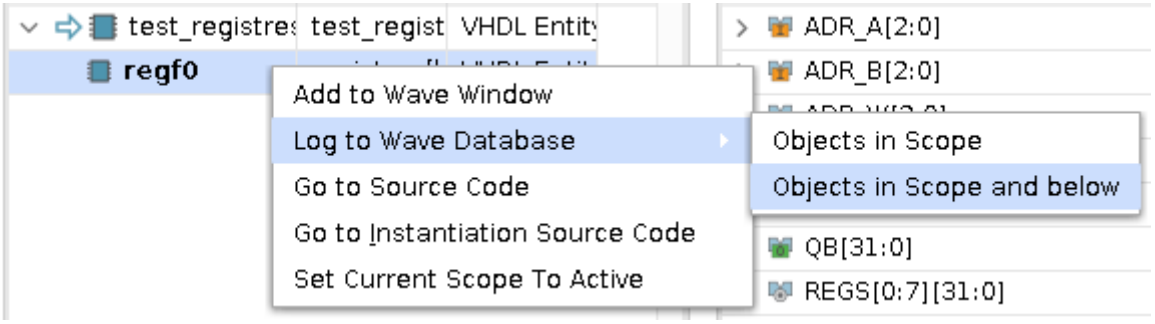
Notes
To declare a 32bits wide 8 registers bank in the declarative part of your architecture:

*type FILE_REGS is array (integer range 0 to (2**5)-1) of std_logic_vector (31 downto 0);*
signal REGS : FILE_REGS;

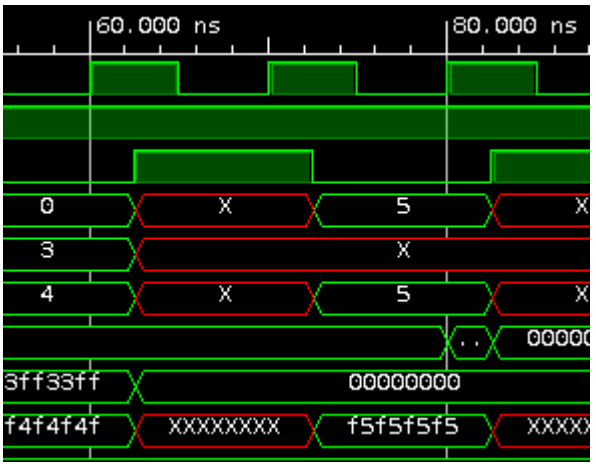
Behavioural simulation

You'll complete the design of our registers bank from the `'registres.0.vhd'` file and then, add the simulation source `'test_registres.0.vhd'`.

Note: to gain access to ALL signals in objects and below, you'll need to tell the simulator to undertake a recursive inclusion



Around 80ns in the waveform window, you notice an **issue** that occurs whenever the processor is trying to **read and write** the same register at the same time !
To overcome this ~~feature~~ bug, we want you to implement the **bypass mechanism**.



Registers Bypass design and simulation

We need to detect the condition which lead to the previously mentioned issue:

if write and ADR_A or ADR_B equal ADR_W then QA or QB <= D;

This time, you'll need to **remove** the `'registres.0.vhd'` file with the new one `'registres.1.vhd'`. In this file, we want you to rewrite in the concurrent area of your architecture the two existing processes named P_READQA and P_READQB.

Check the operationality of your **bypass** by means of a **behavioural simulation**.

Synthesis with constraints

This step will try to tackle the following objectives:

- determining the **maximum frequency** a processor can access our bank of registers,
- use **BlockRAMs** instead of traditional FF

Note: remember to set generic parameters for synthesis as what has been done in [top-level synthesis and generic parameters]

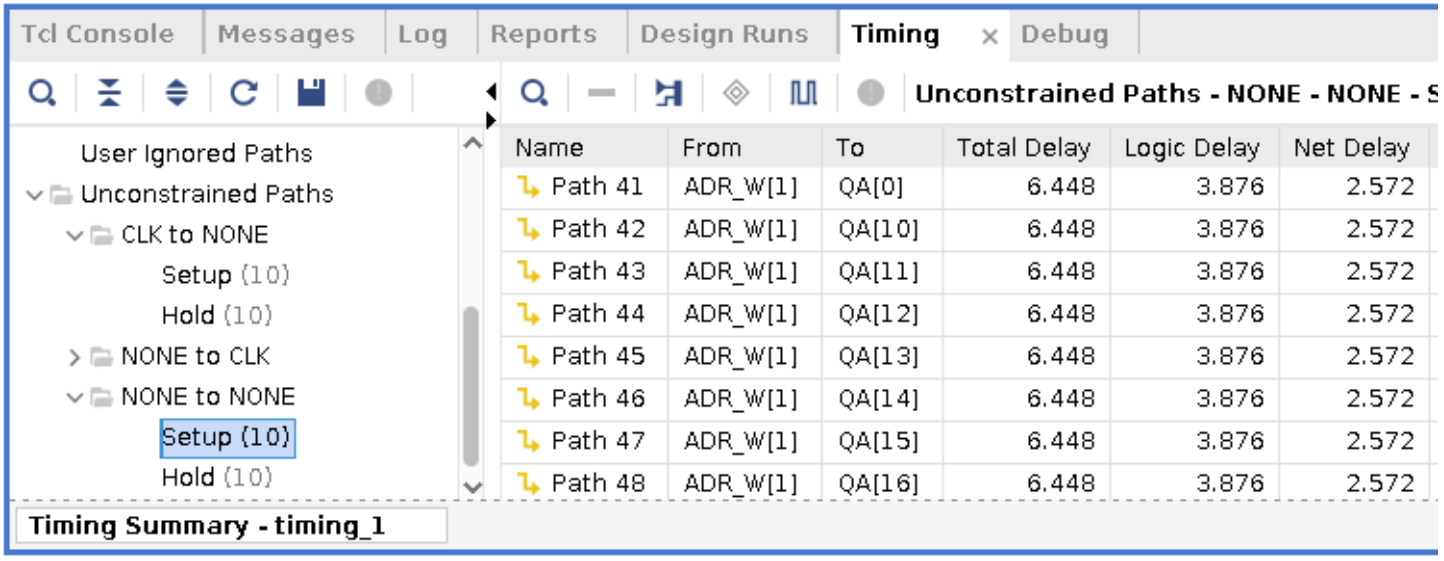
Note: in a similar way to what was done in [Synthesis Constraints Wizard], define the same 125MHz clock.

Ref Name	Used	Functional Category
FDRE	224	Flop & Latch
LUT5	128	LUT
LUT6	66	LUT
OBUF	64	IO
MUXF7	64	MuxFx
IBUF	44	IO
LUT4	7	LUT
LUT3	2	LUT
LUT1	1	LUT
BUFG	1	Clock

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	0	0	0	60	0.00
RAMB36/FIFO*	0	0	0	60	0.00
RAMB18	0	0	0	120	0.00

We're not using any **BRAM** primitives :(

To found out the maximum affordable frequency of our design, we need to discover the longest path in our design ⇒ *Synthesis → Report Timing summary*



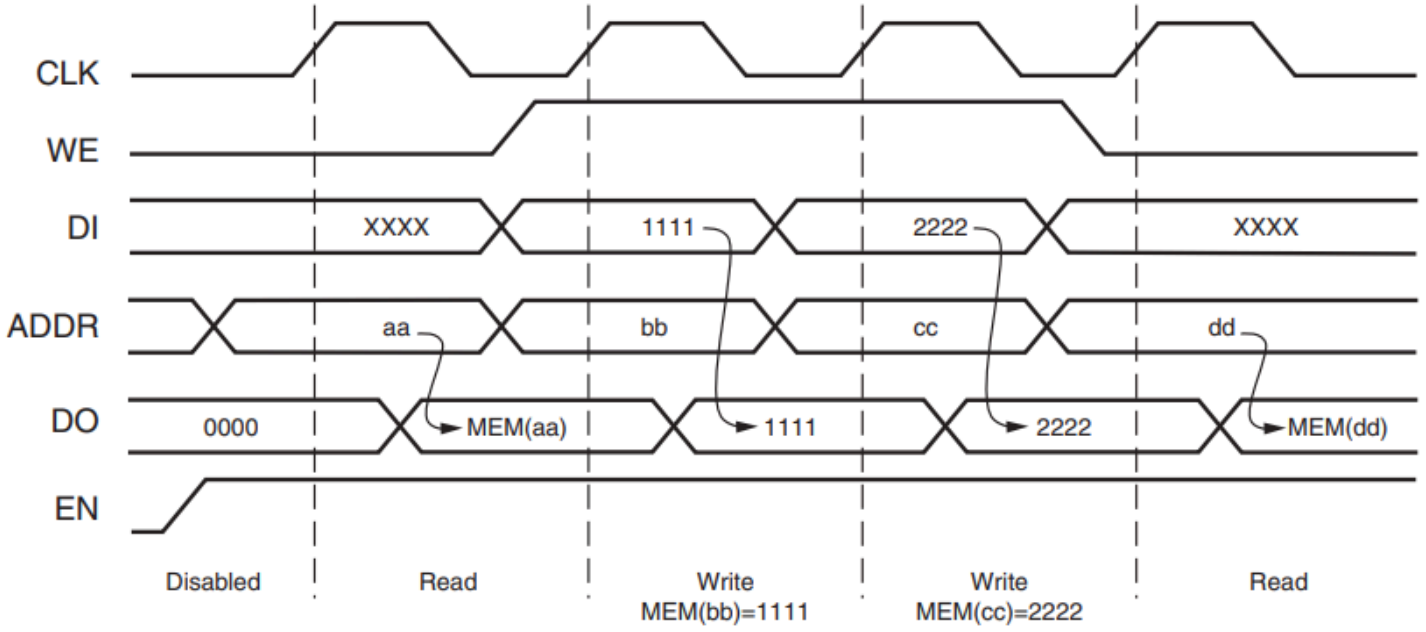
Name	From	To	Total Delay	Logic Delay	Net Delay
Path 41	ADR_W[1]	QA[0]	6.448	3.876	2.572
Path 42	ADR_W[1]	QA[10]	6.448	3.876	2.572
Path 43	ADR_W[1]	QA[11]	6.448	3.876	2.572
Path 44	ADR_W[1]	QA[12]	6.448	3.876	2.572
Path 45	ADR_W[1]	QA[13]	6.448	3.876	2.572
Path 46	ADR_W[1]	QA[14]	6.448	3.876	2.572
Path 47	ADR_W[1]	QA[15]	6.448	3.876	2.572
Path 48	ADR_W[1]	QA[16]	6.448	3.876	2.572

Then, as you may already have guessed … the longest path is when you're undertaking both read and write operations on the same register involving all of the bypass logic
⇒ 150MHz will be our most achievable operational frequency.

[master2] BRAM inference and maximum reachable frequency

For comparison purposes, we want you to design our bank of registers by means of Block RAM primitives. You'll drive synthesis and post-synthesis timing simulation to find out the max sustainable frequency.

Below is **ug473** excerpt showing **WRITE_FIRST** (i.e transparent mode) policy. Note that pipeline register output s not used

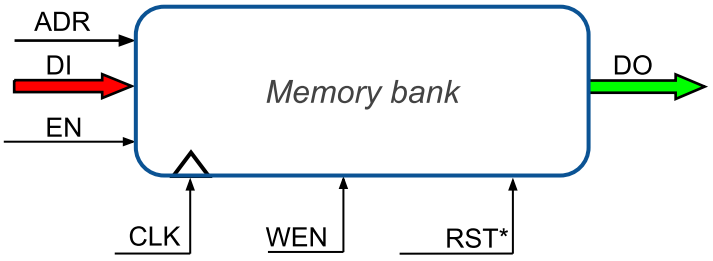


To achieve this, best is to start with TP6 - Memory bank for RISC processor

TP6 - Memory bank for RISC processor

We propose you to carry out our future data & instructions L1 caches as a generic memory bank featuring the following parameters:

- **DBUS_WIDTH** internal words and data buses size, default to **32**
- **MEM_SIZE** nb words within our memory bank, default to **8**
- **FILENAME** file whose content will be our memory bank defaults values, default to *empty string*.



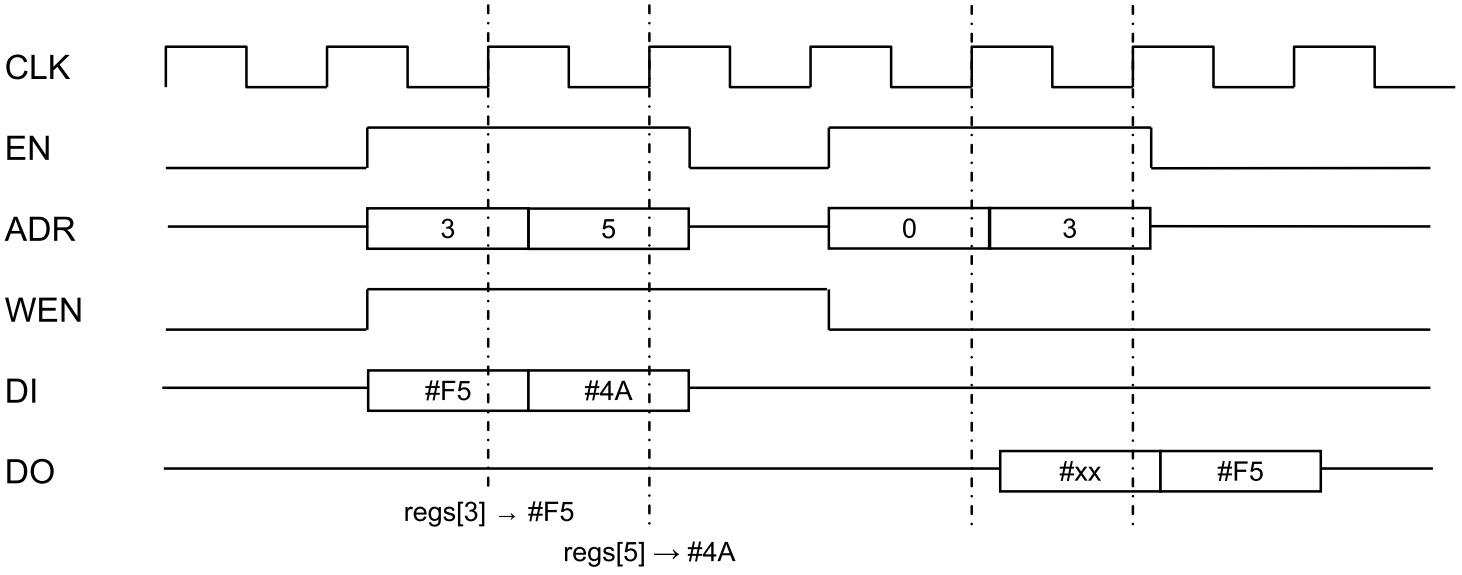
Functional principle

*RST** will be **synchronous**. It either enables loading a file's content whose name has been given as the generic parameter **FILENAME**, or all memory elements will get assigned value **0**.

EN signal validates memory access whose address is set on the ADR bus while WEN signal will tell this operation is a write (i.e read otherwise). On the next rising edge of CLK, data will be set on the `DO` bus (read) or written within the corresponding word.

Memory mapping: 1@ → 1 word

EN disabled ⇒ DO sets to high impedance **Z**.



Simulation

Undertake a behavioural simulation of your memory component by means of the `'memory.3.vhd'` and '`test_memory.3.vhd`' test bench. You'll also need your '`cpu_package.0.vhd`' as source file (log2 function) along with the '`rom_file.0.vhd`' that will get used to pre-fill your memory on reset.

Note: '`rom_file.0.txt`' ought to be added as source file (it will be tagged as TEXT file).

WARNING: in `'test_memory.3.vhd'`, you ought to specify the absolute filename to the `'rom_file.0.txt'` file:

```
.....
    constant FILENAME : string := "/home/<user>/tpvhdl/rom_file.0.txt";
.....
```

Once functional behaviour is achieved, start **design synthesis** and then carry out a **post-synthesis timing simulation** to determine the maximum sustainable frequency.

Hint: don't forget to add the following pre-synthesis constraints like what was done in [top-level synthesis and generic parameters]

- `memory_synth.pre.tcl`

```
# set generic parameter for synthesis
set_property generic {DBUS_WIDTH=32 MEM_SIZE=32 FILENAME="rom_file.0.txt"} [current_fileset]
```

Note: below some size threshold, Vivado may decide to infer RAM as distributed RAM (i.e LUT)

>>> How many BRAM did you infer ??? <<<

Block RAM inference

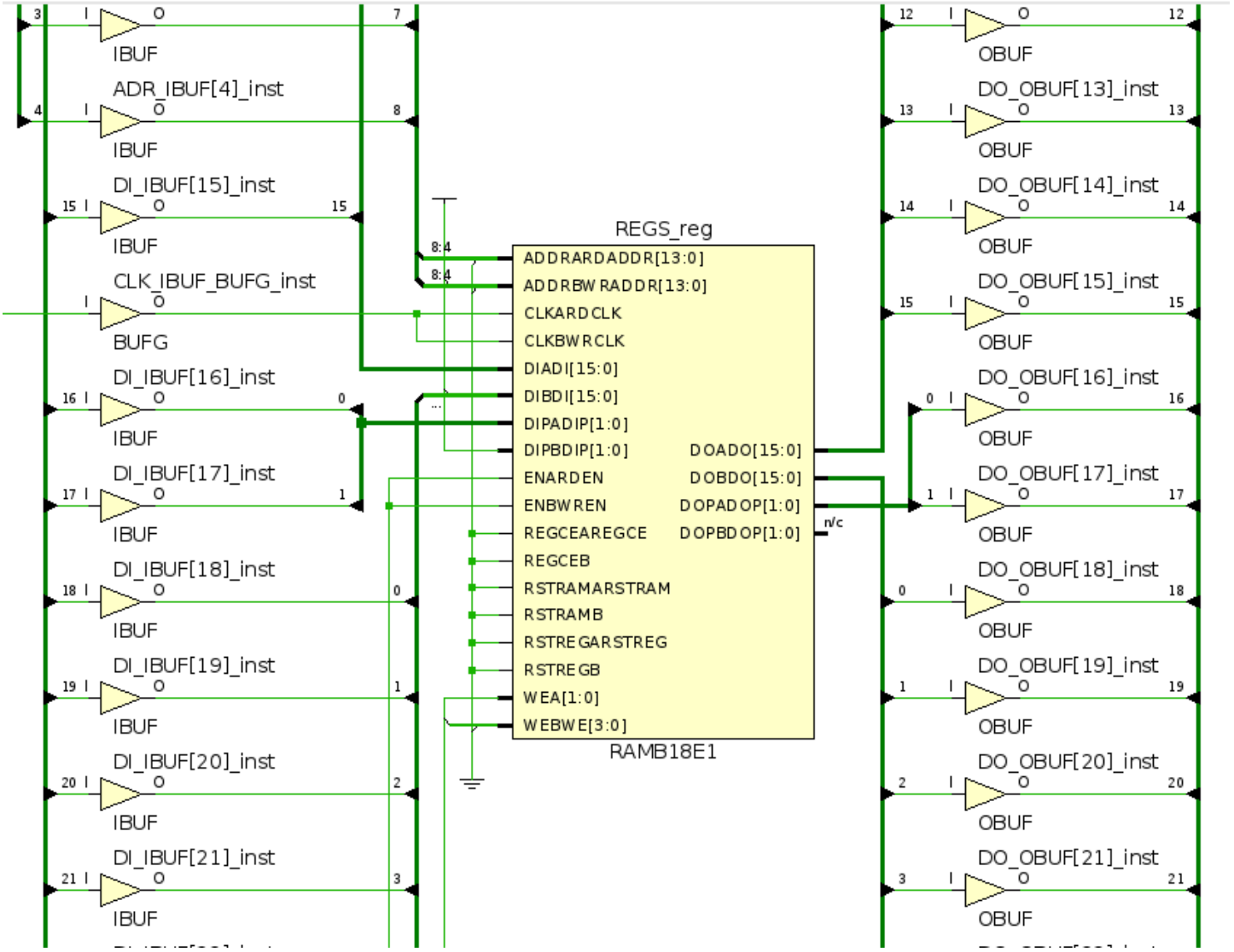
As you may have noticed, you'll need your previously written **log2** function to express the size of the ADR bus.
Starting from the `memory.3.vhd` file, you infer Block RAMs from the Xilinx architecture.

Hint: have a look at ug901 'RAM HDL Coding Techniques' to know how to infer BRAM at synthesis without any architecture specific component instantiation.

⇒ you discover that you need to remove the RST line and disable the 'Z' feature when EN='0'

Restart design synthesis and observe the output results:

```
+-------------------------+------+-------+------------+-----------+-------+
|        Site Type        | Used | Fixed | Prohibited | Available | Util% |
+-------------------------+------+-------+------------+-----------+-------+
| Slice LUTs*             |    0 |     0 |          0 |     17600 |  0.00 |
|   LUT as Logic          |    0 |     0 |          0 |     17600 |  0.00 |
|   LUT as Memory         |    0 |     0 |          0 |      6000 |  0.00 |
| Slice Registers         |    0 |     0 |          0 |     35200 |  0.00 |
|   Register as Flip Flop |    0 |     0 |          0 |     35200 |  0.00 |
|   Register as Latch     |    0 |     0 |          0 |     35200 |  0.00 |
| F7 Muxes                |    0 |     0 |          0 |      8800 |  0.00 |
| F8 Muxes                |    0 |     0 |          0 |      4400 |  0.00 |
+-------------------------+------+-------+------------+-----------+-------+
+-------------------+------+-------+------------+-----------+-------+
|     Site Type     | Used | Fixed | Prohibited | Available | Util% |
+-------------------+------+-------+------------+-----------+-------+
| Block RAM Tile    |  0.5 |     0 |          0 |        60 |  0.83 |
|   RAMB36/FIFO*    |    0 |     0 |          0 |        60 |  0.00 |
|   RAMB18          |    1 |     0 |          0 |       120 |  0.83 |
|     RAMB18E1 only |    1 |       |            |           |       |
+-------------------+------+-------+------------+-----------+-------+
```



Ok, good you really got so far :)
Unfortunately, whatever simulator you may use, VHDL timing simulation does not exist for Xilinx primitives (only verilog and system verilog) … ⇒ it's time to test in real life :)

[master2] bitstream & JTAG debug

Same steps as [master2] Bitstream generation and hardware download then open the **Hardware Debugger**

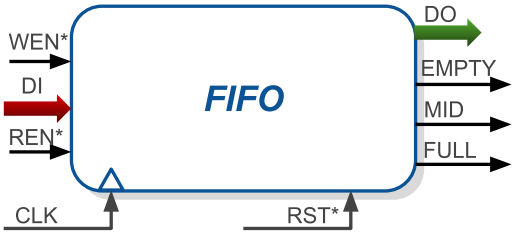
to be continued …

TP7 - dual ports FIFO

On se propose de réaliser une FIFO* synchrone double port selon les caractéristiques génériques suivantes :

- **DBUS_WIDTH** Taille d'un mot du banc avec 32 bits par défaut.
- **ABUS_WIDTH** Profondeur d'utilisation avec 3 bits d'adresse par défaut - soit 2**3 mots -.

*First In First Out



Principe de fonctionnement

Le signal RST*, synchrone, remet à **0** les pointeurs lecture et écriture **mais n'efface pas** le contenu des registres. Il positionne également la sortie DO <= 'Z'.

Tant que WEN* est actif, la FIFO continue de se remplir de façon circulaire à chaque **front montant** d'horloge écrasant ainsi les données les plus anciennes.

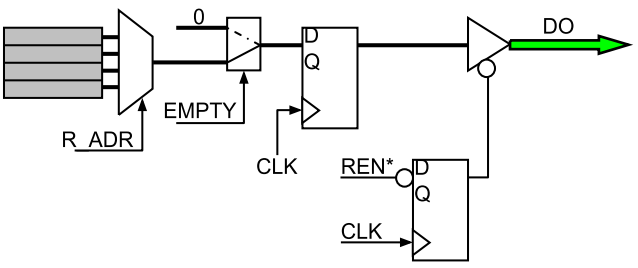
Tant que REN* est actif, la FIFO continue de présenter une donnée sur le bus à chaque **front montant** d'horloge jusqu'à être vide, auquel cas elle présentera la valeur **0**.

Le bus de sortie Q devant être raccordé à d'autres unités, la sortie passe en haute impédance quand REN* inactif.

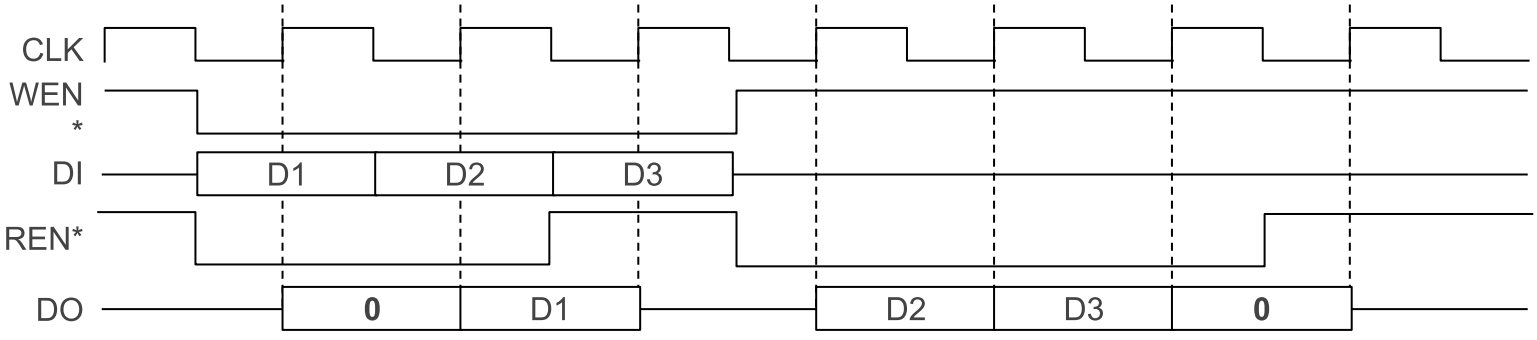
Les signaux EMPTY, MID et FULL représentent respectivement l'état vide, au moins à moitié plein* et plein. Ces derniers seront mis à jour sur front montant d'horloge.

*Le signal MID='1' quand remplissage FIFO >= 50%.

Les signaux WEN* et REN* peuvent être actifs simultanément.



Chronogramme



Implémentation

Vous compléterez le fichier `fifo.0.vhd` comprenant entité et architecture.

La lecture / écriture étant circulaire, vous ferez attention aux conditions limites pour l'établissement des indicateurs de remplissage. Ensuite, afin d'accélérer les transferts, les pointeurs devront déjà être positionnés sur le futur emplacement de lecture ou d'écriture, l'incrément se faisant après l'opération proprement dite. Enfin, le pointeur de lecture référence **toujours** la donnée la plus ancienne.

Cas particuliers

Lecture & Ecriture & FIFO vide ⇒ Q=0 & mémorisation de la donnée en entrée.

FIFO pleine & Écriture ⇒ mémorisation de la donnée et incrément du pointeur de lecture.

FIFO pleine & Lecture & Ecriture ⇒ Q=donnée la plus ancienne et mémorisation de la nouvelle.

Notes

- Le type **buffer** associé à un signal permet de relire un port configuré en sortie.

Behavioural simulation

Vous compléterez le fichier `test_fifo.0.vhd` avec une instance du composant `fifo` possédant **4** mots mémoire de **4** bits.

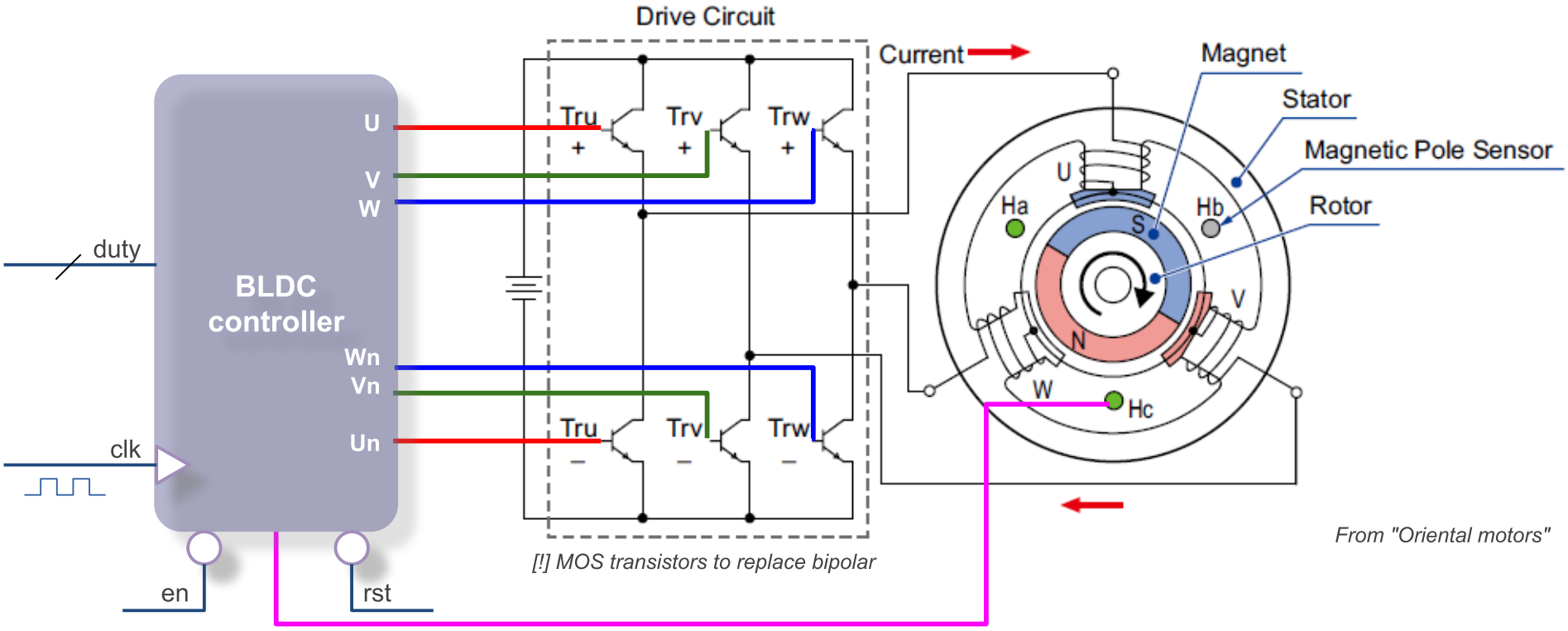
Synthesis

You will, as usual, undertake a synthesis of your design.
However, having already been using **BRAM** primitives, we'd like you to implement our FIFO based on these primitives. Indeed, according to Xilinx ug473 documentation, BRAMs embed all logic bases to implement a FIFO. You are welcomed to tamper on the initial features of our FIFO, the main goal being to reduce the overall usage of LUT and FF vs BRAM elements.

>>> the less LUTs and FFs you use, the better will be your design (and rating) !! <<<

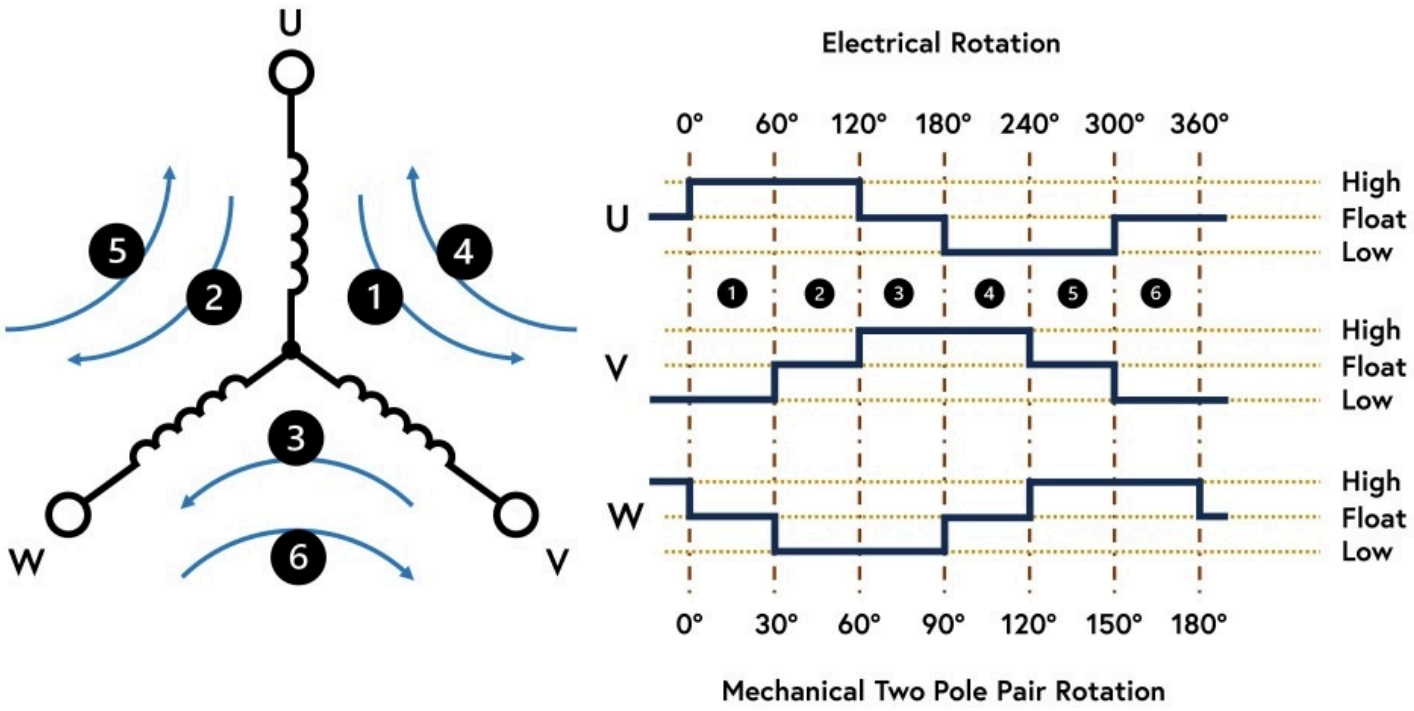
TP8 - BLDC controller

BLDC stands for "brushless direct current" and is related to a type of motor making use of permanent magnets on their rotor. These high reliability motors are intended for heavy duty purposes and can be found in drones, servers … and hard drives !

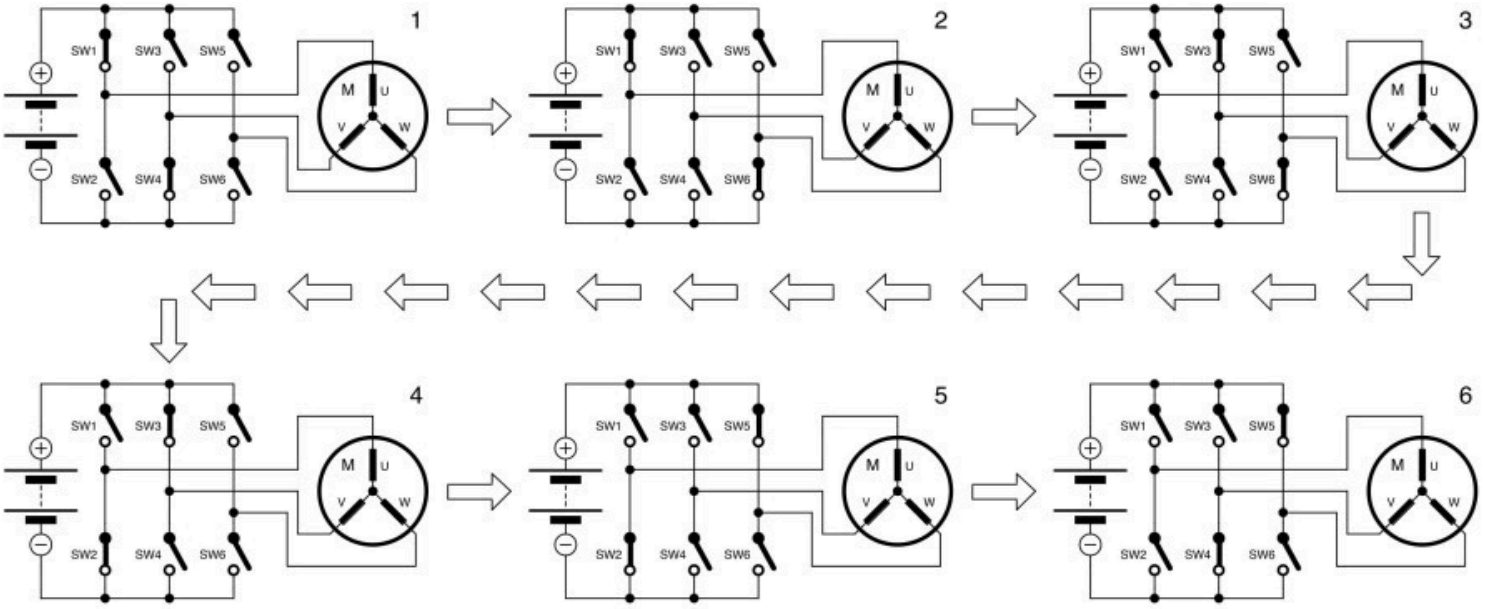


[!] MOS transistors to replace bipolar

Unlike DC motors, a BLDC motor requires 3 phases sequenced in a precise timing.
As shown above, your "*BLDC controller*" circuit will be in charge of driving the six transistors of the "*driver*" part of the system. It is worth mentioning that "*electrical speed*" may differ from "*mechanical speed*" (e.g startup phase with mechanical load), hence we'll make use of a feedback signal. This signal will either come from a hall effect sensor located within the stator … or (and probably easier to use) a simple optical signal that detects the position of the disk in the hard drive BLDC use case (see photos below).



Excerpt from Elektor magazine: "BLDC Guide du débuttant"



Excerpt from Elektor magazine: "BLDC Guide du débuttant"

Speed control

You'll make use of PWM control to adjust the speed of the brushless motor. Additionally, there will be a ramp UP and ramp DOWN effect to avoid high pulses of energy, exemple:

- motor is at stop and the speed control is set at its maximum … then you'll smoothly increase the speed up to the specified value.

BLDC controller

By means of a FPGA, we'd like you to implement the logical part of the controller as a component.
Unfortunately, this kind of controller requires a LOT of parameters … but to ensure your success as a first step undertaking such a device, we'll restrain ourselves to the following generic parameters:

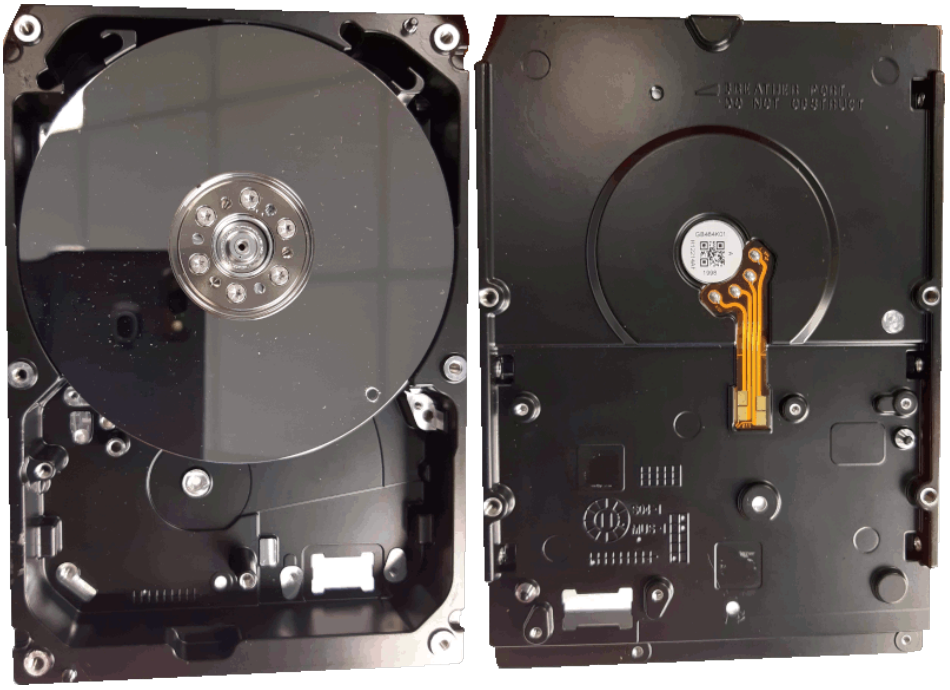
- **MAX_CPT** → defines a whole phase cycle expressed as number of ticks of the main clock

As an example, let's say our main clock has a 1MHz frequency. On the other end, our motor requires a 50Hz phase cycle ⇒ MAX_CPT will get set at (1MHz/50) at instantiation (or synthesis) time.

to be continued

Use case

In order to avoid destroying one of our electrical vehicles (!), we'll make use of specifically prepared hard drives for such an experiment.



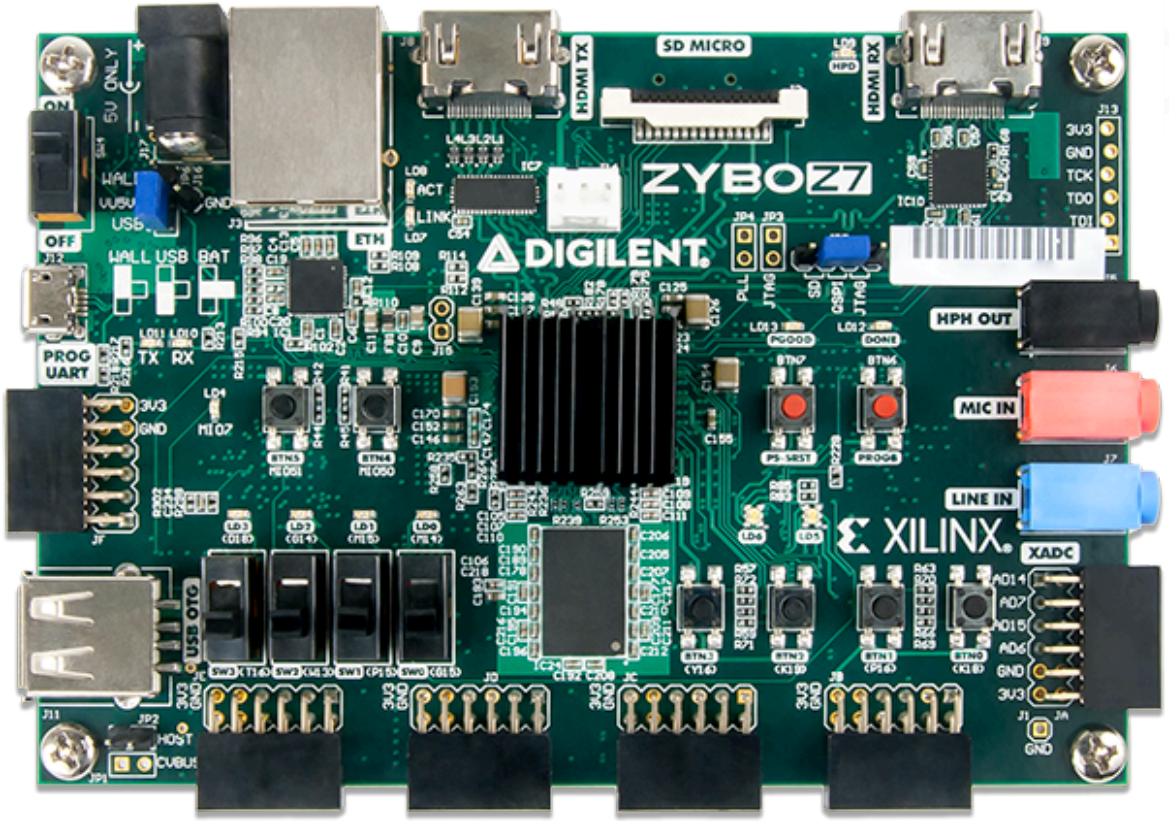
to be continued

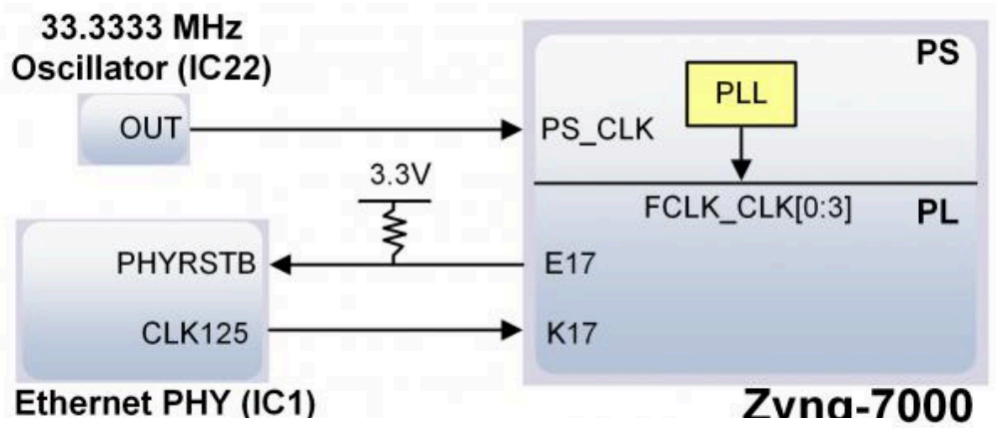
Links

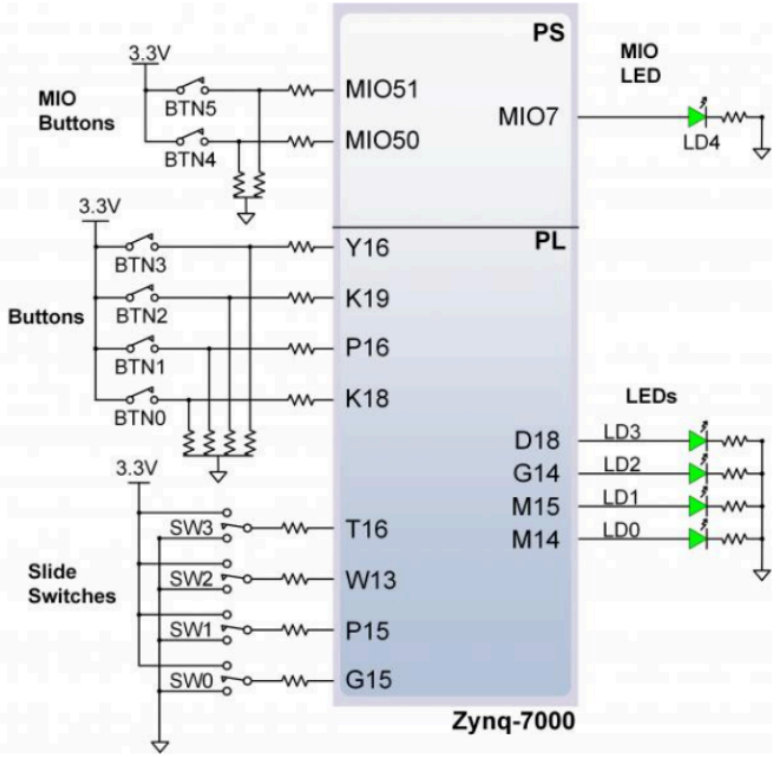
[Elektor] BLDC newbies guide
https://www.elektormagazine.fr/articles/contr%C3%B4le-des-moteurs-bldc-guide-du-d%C3%A9butant

Master 2

[M2] Zybo Z7-20 board







Links

VHDL base files
https://secil.univ-tlse3.fr/teaching/francois/UE-VHDL/M2/

Zybo-Z7 reference manual
https://secil.univ-tlse3.fr/teaching/francois/UE-M2-VHDL/ZYBO-Z7-reference-manual-B.pdf

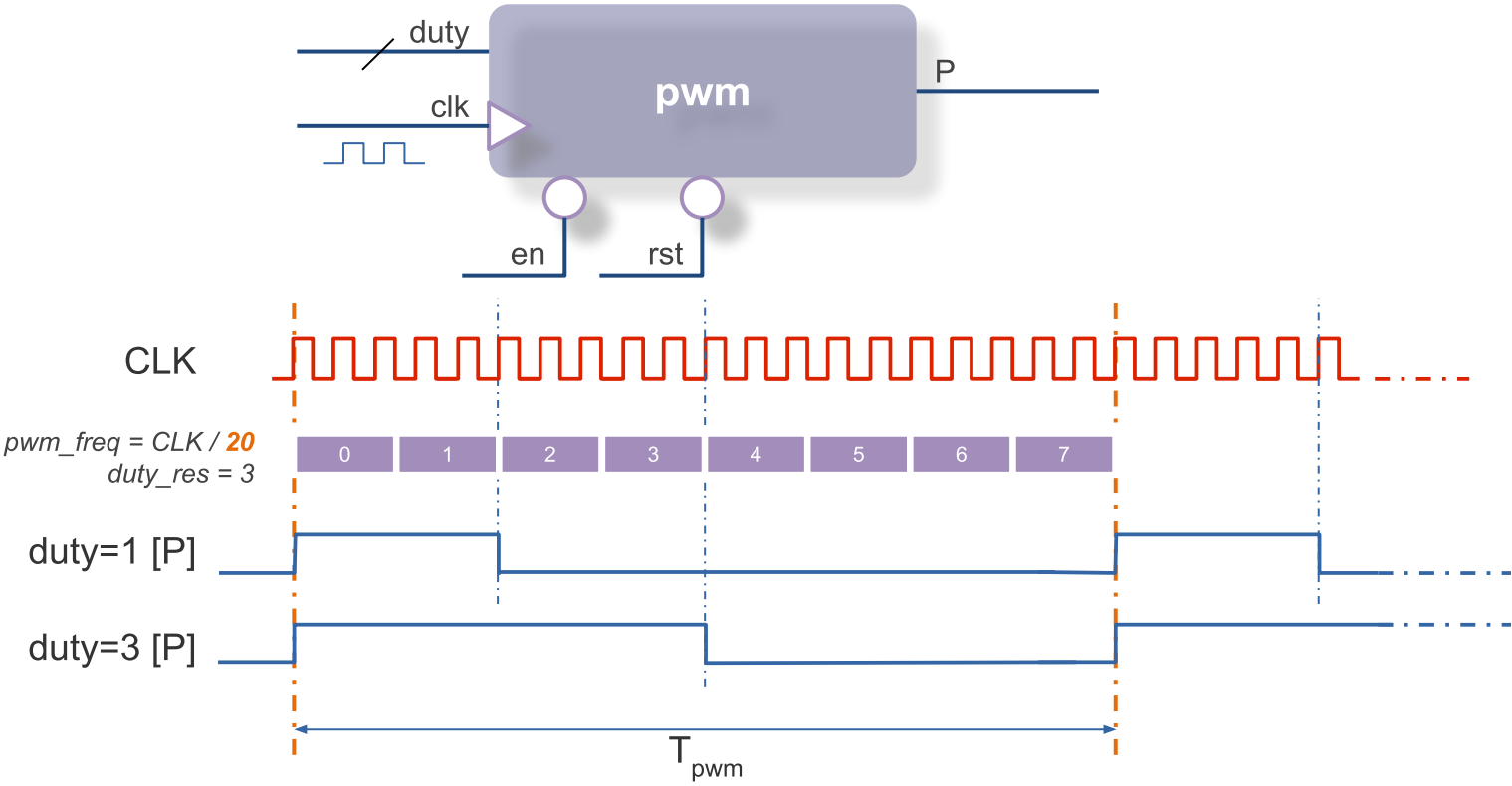
Zybo-Z7 schematic
https://secil.univ-tlse3.fr/teaching/francois/UE-M2-VHDL/ZYBO-Z7_schematic-D1.pdf

[M2] PWM

As a first step, we'll implement a PWM module that will get connected to a 125MHz clock generator located on our Zybo-Z7. The outcome will be a 1s variable pulse width driving a led.

To help you starting, you'll need the following files:

https://secil.univ-tlse3.fr/teaching/francois/UE-VHDL/M2/pwm.vhd
https://secil.univ-tlse3.fr/teaching/francois/UE-VHDL/M2/test_pwm.vhd



The `duty=0` corner case: since our PWM features an EN pin, we'll consider `duty=0` as an active slot.

Behavioural simulation

You'll need to follow the TP1 - Pulse generator as a guideline:

- select Zybo-Z7 20 board
- add `pwm.vhd` component file + `test_pwm.vhd` simulation file
- complete these files
- launch behavioural simulation

Note: to launch the simulation, don't forget to re-enable `generic map` *along with the* `behaviour` *architecture.*

Synthesis and Implementation

Now you'll follow the TP2 - Pulse generator synthesis as a guideline:

Add the 'pwm_synth.pre.tcl' script to the synthesis of your design:

- pwm_synth.pre.tcl

```
# set generic parameter for synthesis
set_property generic {SYS_CLK=125000000 PWM_FREQ=1 DUTY_RES=4} [current_fileset]

# set generic parameters intended to post-synthesis simulation pwm_clk = sys_clk / 20)
#set_property generic {SYS_CLK=125000000 PWM_FREQ=6250000 DUTY_RES=4} [current_fileset]
```

Synthesis output will exhibits:

```
---------------------------------------------------------------------------------
Starting RTL Elaboration : Time (s): cpu = 00:00:03 ; elapsed = 00:00:04 . Memory (MB): peak =
2810.660 ; gain = 0.000 ; free physical = 16041 ; free virtual = 37790
---------------------------------------------------------------------------------
INFO: [Synth 8-638] synthesizing module 'pwm' [/home/devel/teaching-vhdl/VHDL/pwm.vhd:40]
      Parameter sys_clk bound to: 125000000 - type: integer
      Parameter pwm_freq bound to: 1 - type: integer
      Parameter duty_res bound to: 4 - type: integer
INFO: [Synth 8-256] done synthesizing module 'pwm' (0#1)
[/home/devel/teaching-vhdl/VHDL/pwm.vhd:40]
```

Note: `SYS_CLK=125E06` will be considered a string … without issue in the upcoming steps 🙂

Post-synthesis simulation

Please pay attention to the fact that being a top-level design, our `'pwm'` module won't be able to make use of VHDL **generic** parameters.
Hence, we need to remove the *generic map* part of the `'test_pwm.vhd'` file. Please pay attention to the `'structure'` architecture now in use.

- test_pwm.vhd

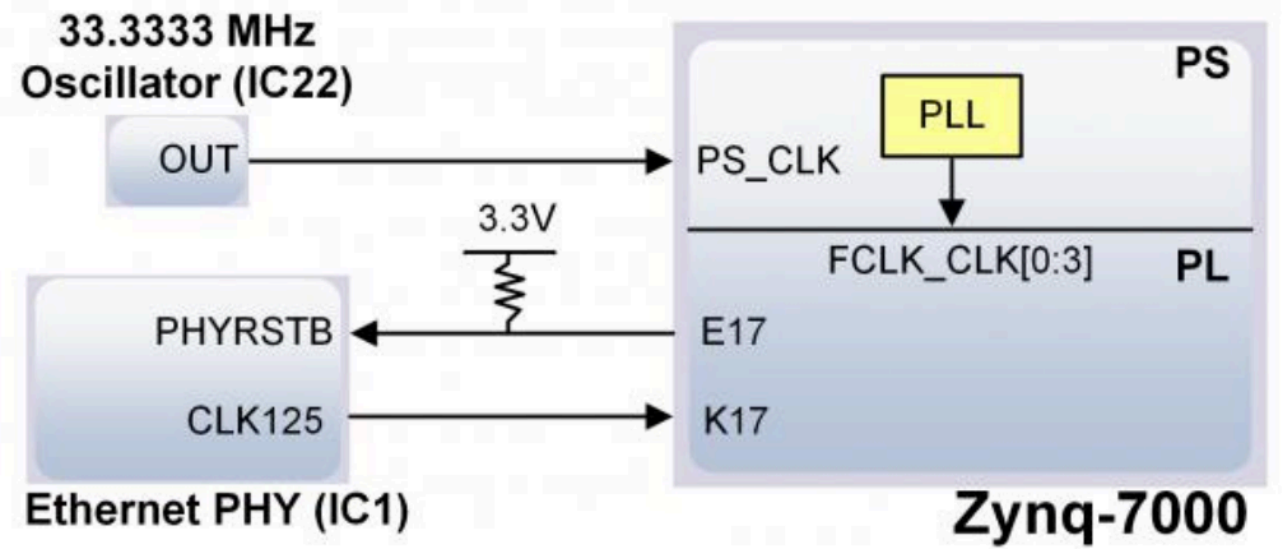
```
……
-------------------------------------------------
-- instantiation et mapping du composant registres
--pwm0 : entity work.pwm(behaviour) --behavioural simulation
--          generic map (sys_clk => sys_freq,
--                               pwm_freq => pwm_freq,
--                               duty_res => duty_res)
pwm0 : entity work.pwm(structure) --post-synthesis functional simulation
            port map (CLK => E_CLK,
                      RST => E_RST,
                      EN => E_EN,
                      DUTY => E_DUTY,
                      P => E_P);
……
```

In addition, you may consider using **alternate generic values** for post-synthesis simulation, see `pwm_synth.pre.tcl`.

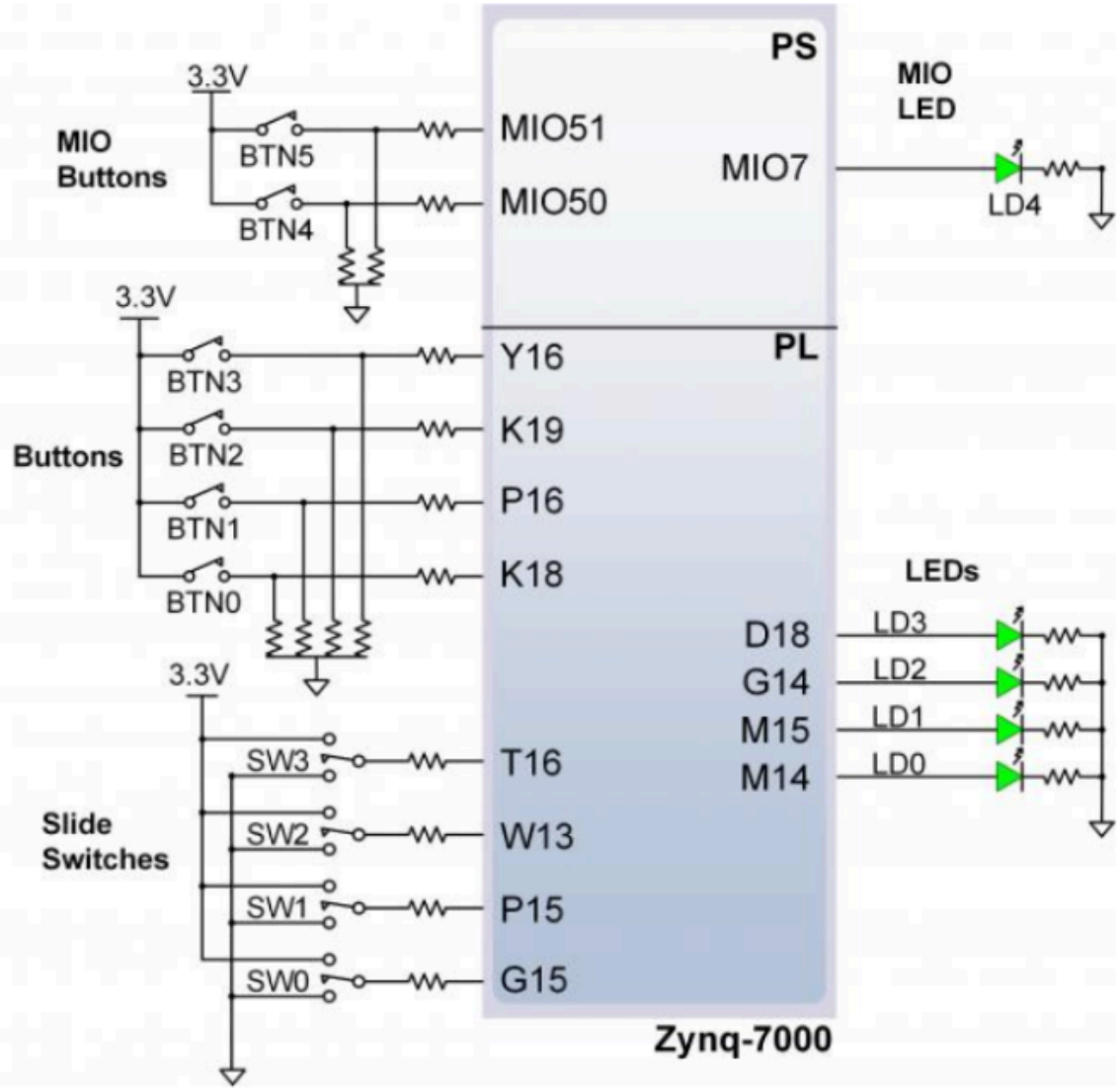
IO planning (implementation)

It's now time to start the **Implementation** (i.e IO planning):

- connect the onboard 125MHz clock to our CLK input



- connect the 4 slides switches SW[3..0] as the DUTY input bus,
- connect the BTN0 as the EN input (active low ---look at the pull-down resistors!)
- connect the 4 leds as the P output
- ~~RST signal will get tied to '1'~~ RST signal will get tied to BTN1 (pull-down resistor!)



See TP3 - Pulse generator constraints as a guideline:

Launch *Synthesis* → *Open Synthesized design* → *Constraint Wizard* to create the `pwm.xdc` constraint file:

- `pwm.xdc`

```
# Master Clock timing constraint
create_clock -period 8.000 -name CLK -waveform {0.000 4.000} [get_ports CLK]
set_property PACKAGE_PIN K17 [get_ports CLK]
set_property IOSTANDARD LVCMOS33 [get_ports CLK]

set_property PACKAGE_PIN K18 [get_ports EN]
set_property IOSTANDARD LVCMOS33 [get_ports EN]
set_property PACKAGE_PIN P16 [get_ports RST]
set_property IOSTANDARD LVCMOS33 [get_ports RST]
set_property PACKAGE_PIN G15 [get_ports {DUTY[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {DUTY[0]}]
```

```
set_property PACKAGE_PIN P15 [get_ports {DUTY[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {DUTY[1]}]
set_property PACKAGE_PIN W13 [get_ports {DUTY[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {DUTY[2]}]
set_property PACKAGE_PIN T16 [get_ports {DUTY[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {DUTY[3]}]

set_property PACKAGE_PIN D18 [get_ports P]
set_property IOSTANDARD LVCMOS33 [get_ports P]
```

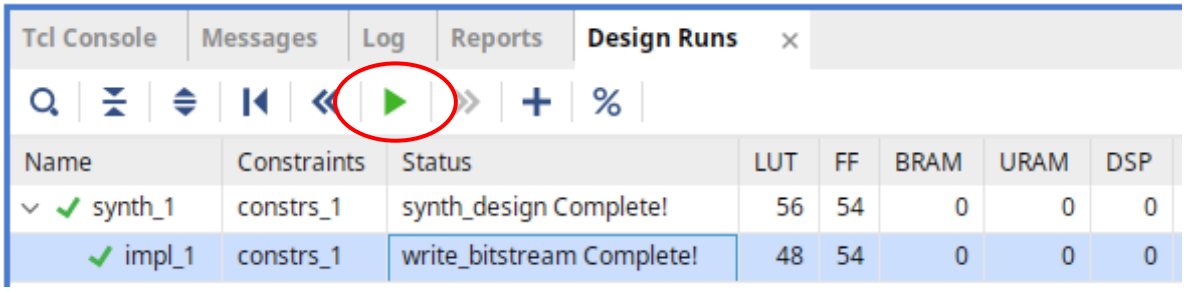
Note: best is to first create this XDC file through the Wizard … then to modify it on your own

TODO: set RST as '1'

TODO: add debug core to our design

Bitstream generation

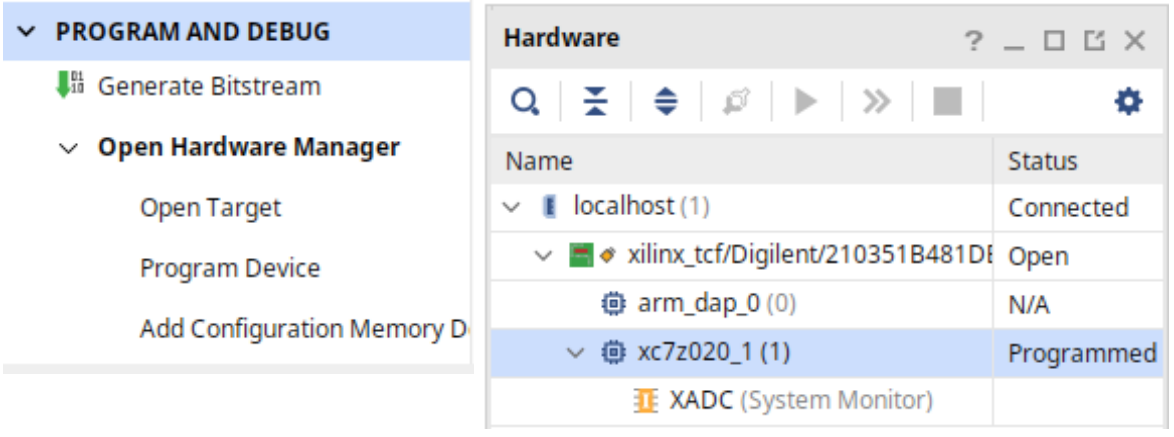
Select 'impl_1' (i.e implementation), click the green RUN icon, then continue with *Bitstream generation*



Hardware Manager

This is the main application responsible for establishing a dialog with the hardware … that will enable you to upload or readback a bitstream.

Open Hardware Manager → Open Target → auto connect



[optional] Bitstream readback

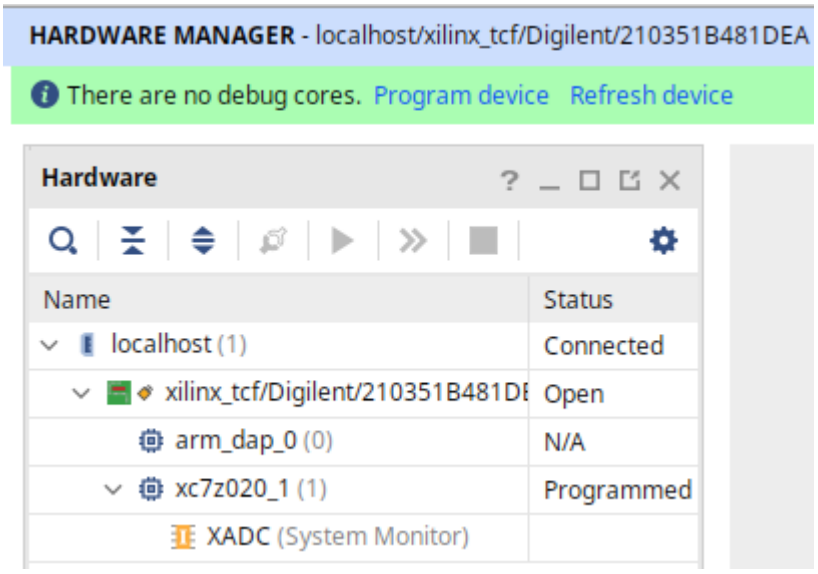
In order to backup a readable bitstream, use the following command in the `TCL console`:

```
readback_hw_device [current_hw_device] -bin_file zyboZ7-20_original.bin
```

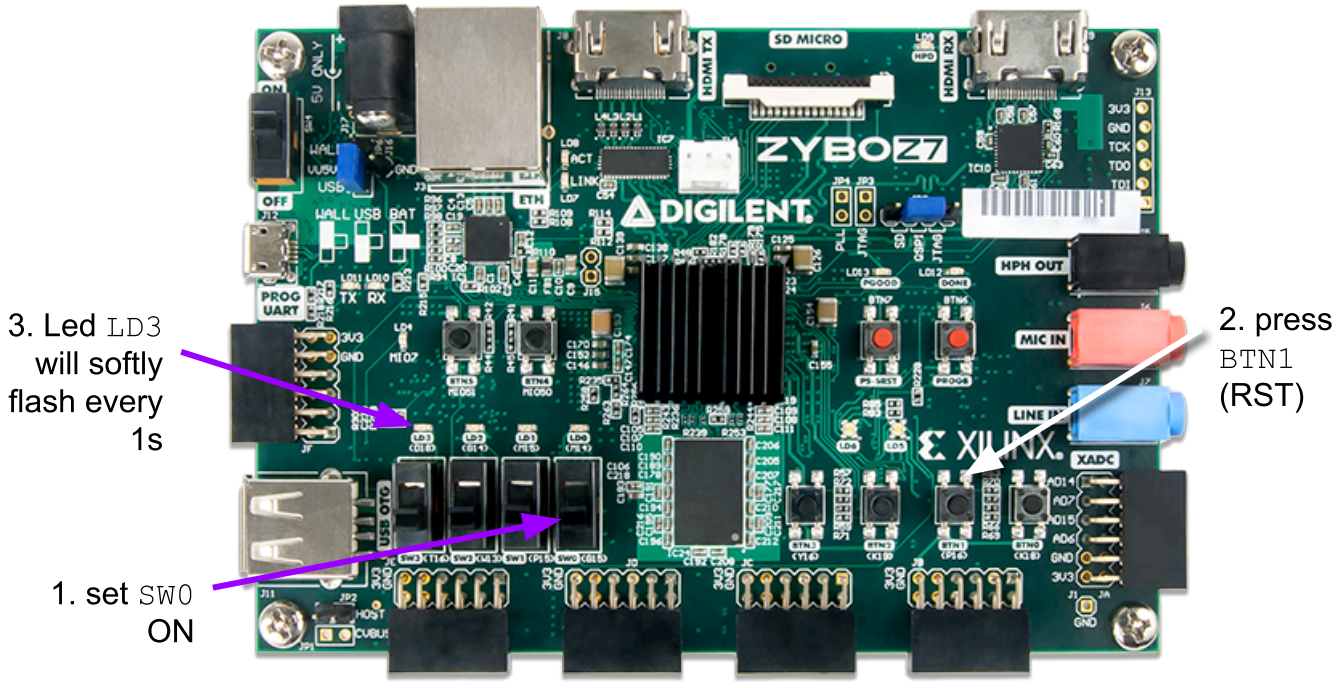
https://docs.xilinx.com/r/en-US/ug908-vivado-programming-debugging/Bitstream-Verify-and-Readback-for-FPGAs-and-MPSoCs

Download bitstream

Now you just need to click on '*Program device*' and it will select the latest generated bit stream that will get downloaded to the config RAM of your FPGA.



Tests 🙂

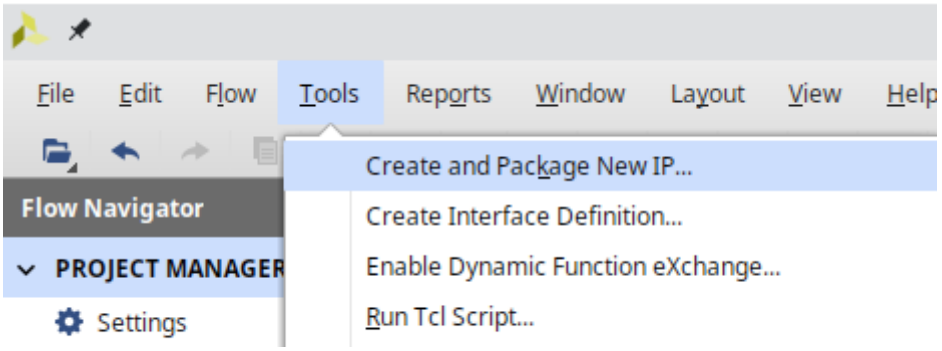


[M2] my AXI-enabled PWM IP[2]

We'll now turn our PWM to an AXI-enabled PWM IP. This leverages the needs for large systems integration involving the PS part (i.e on board ARM9 processors).

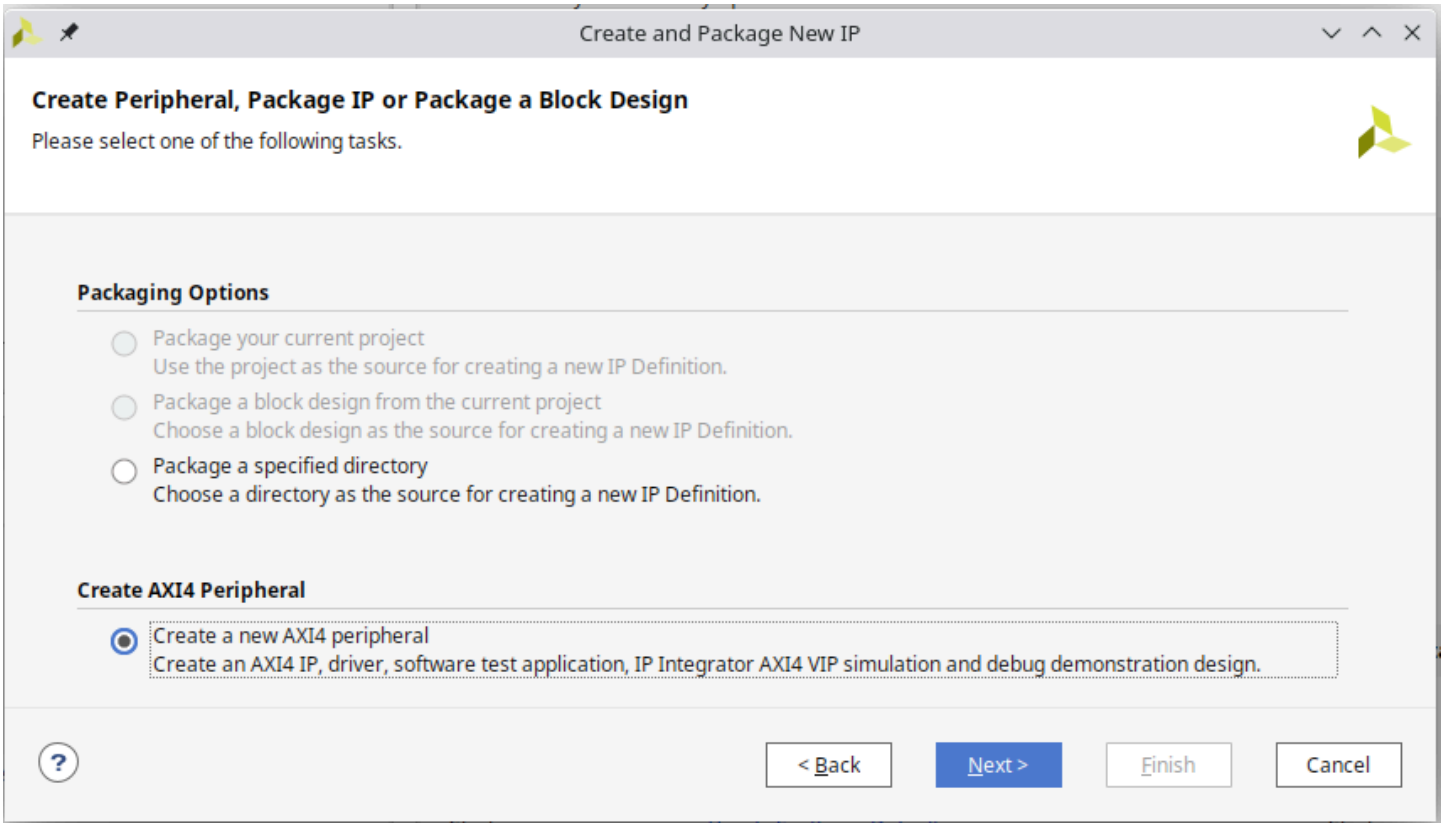
To reach this goal, we'll start creating a new VHDL project without specifying any HDL source code.

Then, we'll start the creation of an AXI component that exhibits the following features:

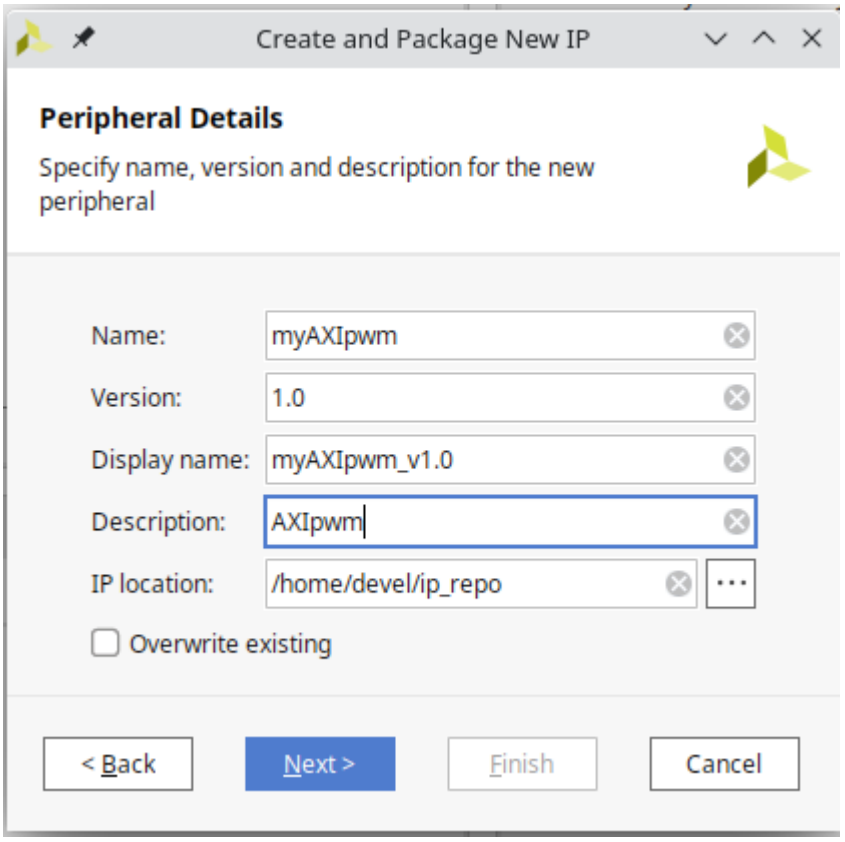


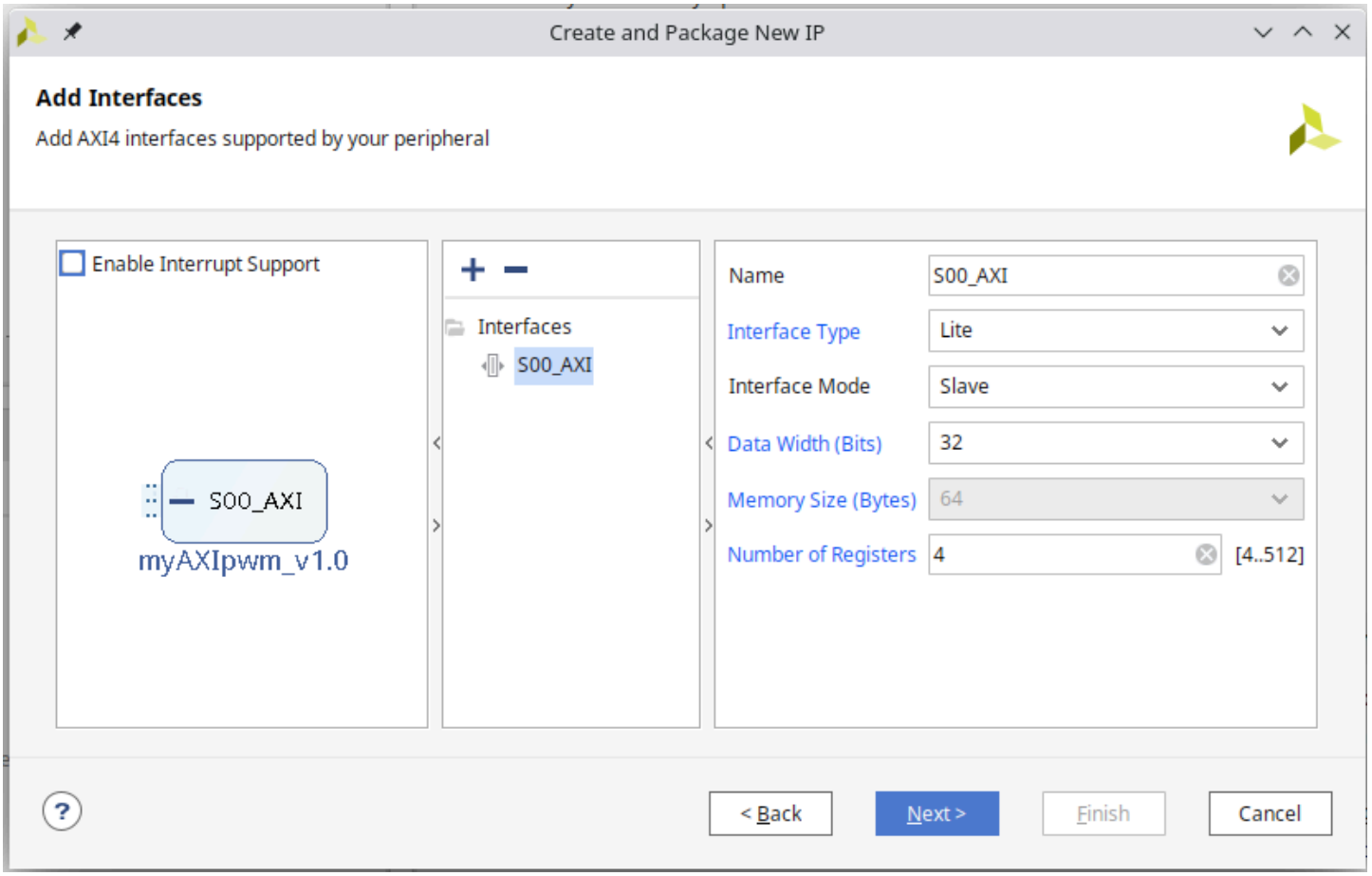
- AXI4-lite SLAVE
- 32 bits bus width
- 4 registers

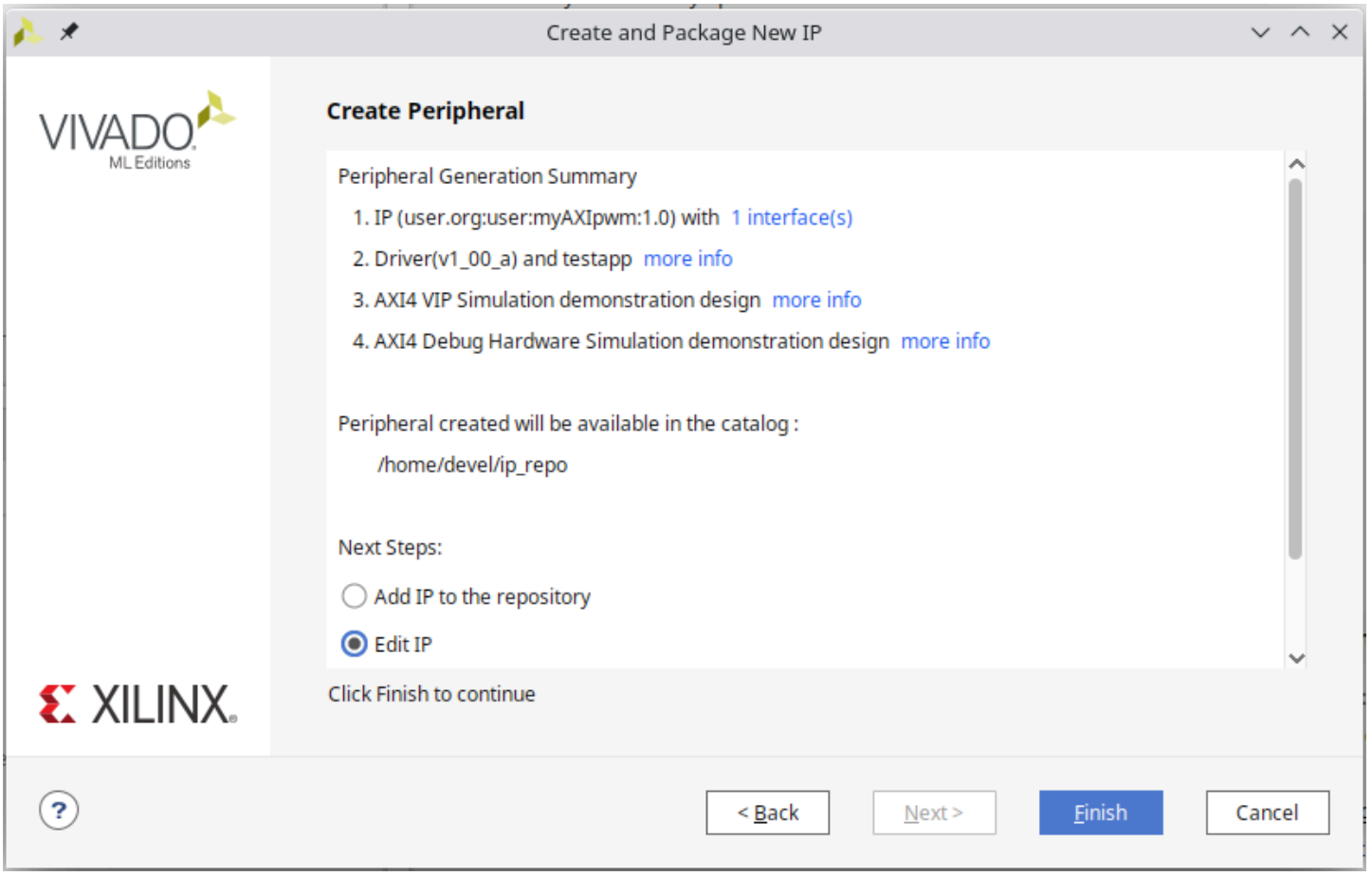
Select → *Create a new AXI4 peripheral*



[2] To help yourself creating an IP, search over the net for *"Xilinx IP integrator"* then select *"Vivado Design Suite Tutorial: Creating and Packaging Custom IP"*


Create and Package New IP
Peripheral Details
Specify name, version and description for the new peripheral
Name:
myAXIpwm
Version:
1.0
Display name:
myAXIpwm_v1.0
Description:
AXIpwm
IP location:
/home/devel/ip_repo
Overwrite existing
< Back
Next >
Finish
Cancel


Create and Package New IP
Add Interfaces
Add AXI4 interfaces supported by your peripheral
Enable Interrupt Support
S00_AXI
myAXIpwm_v1.0
Interfaces
S00_AXI
Name
S00_AXI
Interface Type
Lite
Interface Mode
Slave
Data Width (Bits)
32
Memory Size (Bytes)
64
Number of Registers
4
[4..512]
< Back
Next >
Finish
Cancel



Now you can going on with IP customization

This will dynamically create a new project `<home>/ip_repo/edit_myAXI_pwm_v1_0.xpr`

Links

Digilent Zybo's AXI IP + Vitis baremetal driver
https://digilent.com/reference/programmable-logic/guides/getting-started-with-ipi

AXI IP creation
https://indico.ictp.it/event/a14283/session/62/contribution/252/material/slides/2.pdf

[Xilinx] Vivado Design Suite Tutorial: Creating and Packaging Custom IP (UG1119)

HOW TO CREATE an AXI4-FULL CUSTOM IP with AXI4-LITE and UART INTERFACES in VIVADO
https://www.mehmetburakaykenar.com/how-to-create-an-axi4-full-custom-ip-with-axi4-lite-and-uart-interfaces-in-vivado/192/

Fixing Xilinx's Broken AXI-lite Design in VHDL
https://zipcpu.com/blog/2021/05/22/vhdlaxil.html
[sample]https://github.com/ZipCPU/wb2axip/blob/master/bench/formal/xlnxdemo_2020_2.vhd

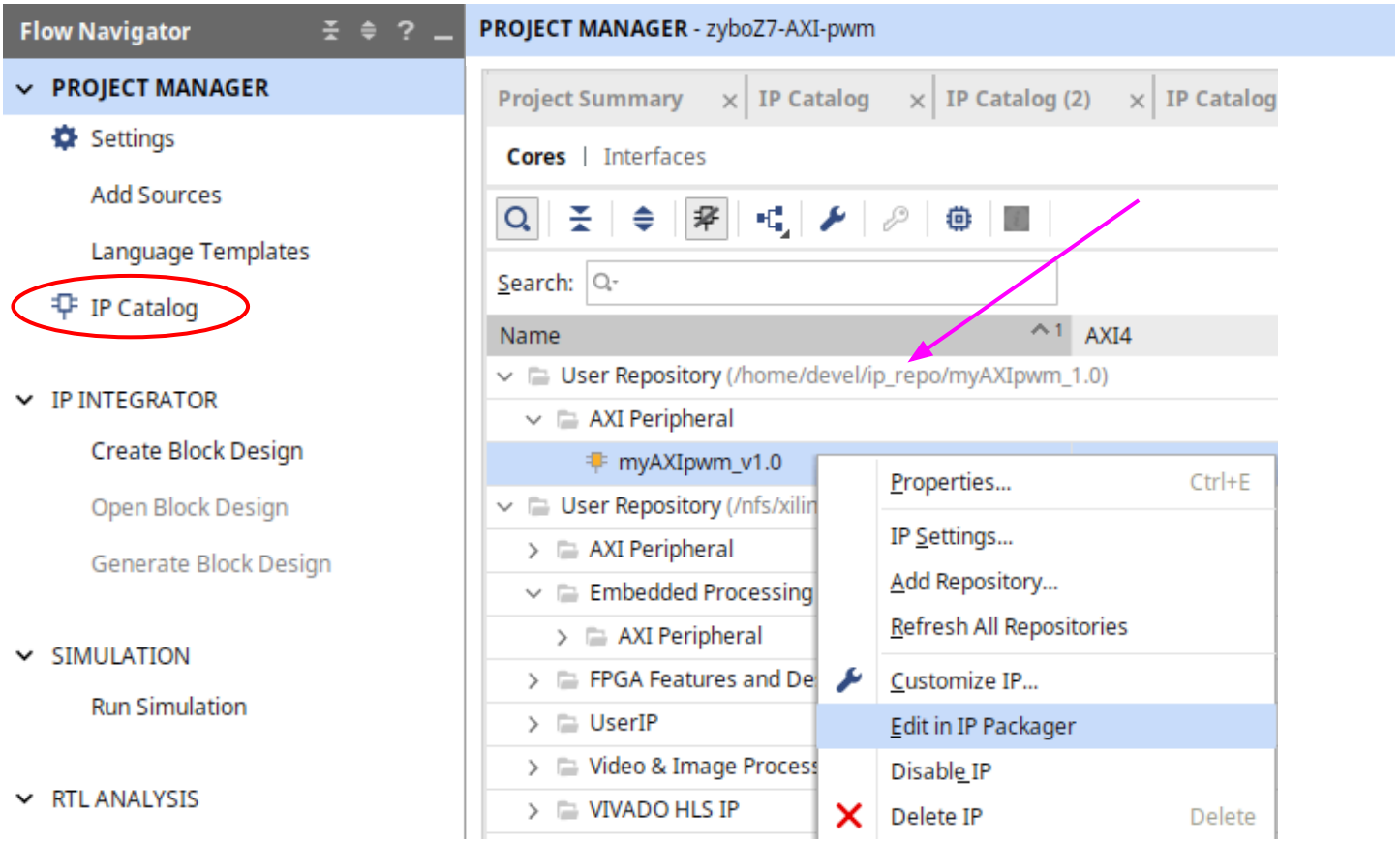
Create Custom AXI Cores Part 1: Straight to the Finish Line
https://www.hackster.io/cospan/create-custom-axi-cores-part-1-straight-to-the-finish-line-a70e5e

Designing a Custom AXI IP on Vitis

https://www.hackster.io/pablotrujillojuan/designing-a-custom-axi-ip-on-vitis-a0ad06

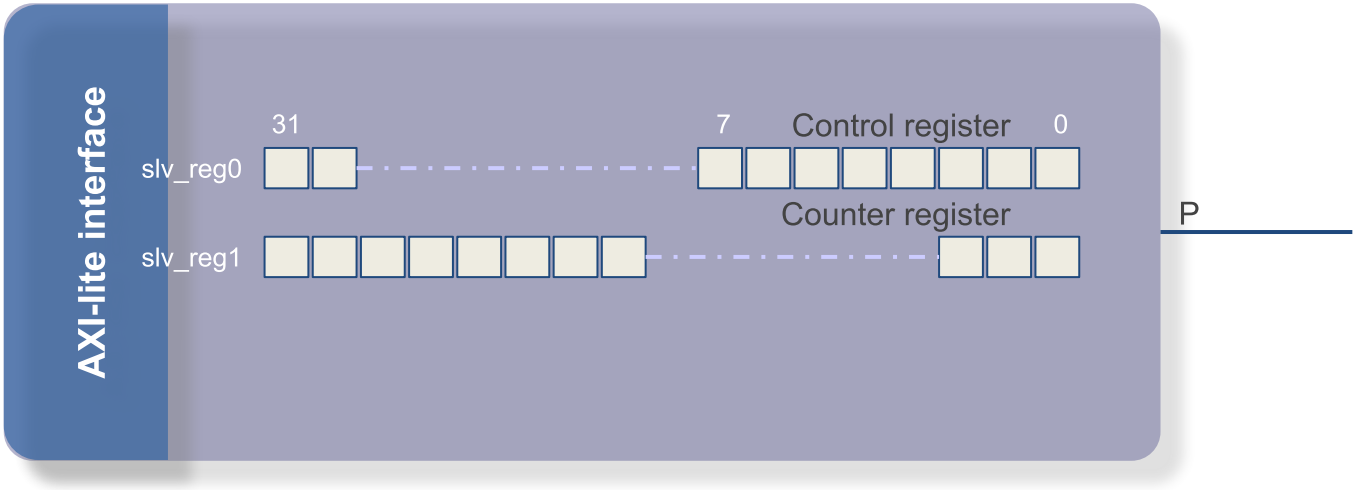
Editing an IP from main project

Editing an IP means it will **launch a new project** embedded within your current project.
To start editing your IP from your main project, click on *IP catalog* then select your IP + right click then select *Edit in IP Packager*



IP customization

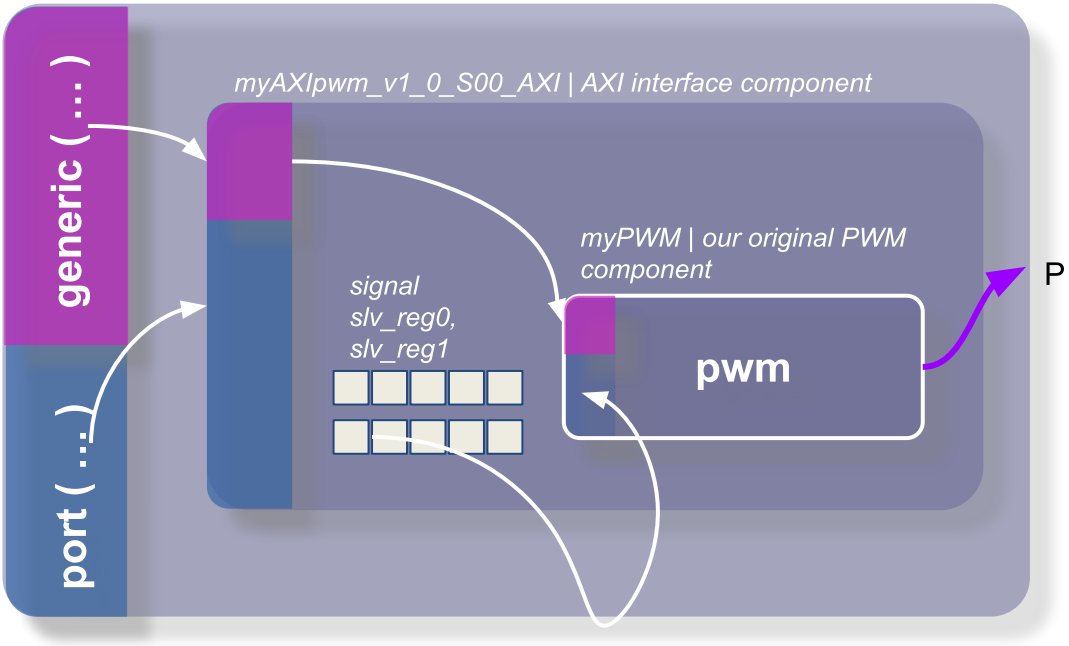
Roughly speaking, IP customization is a packaging of our previously designed PWM component.



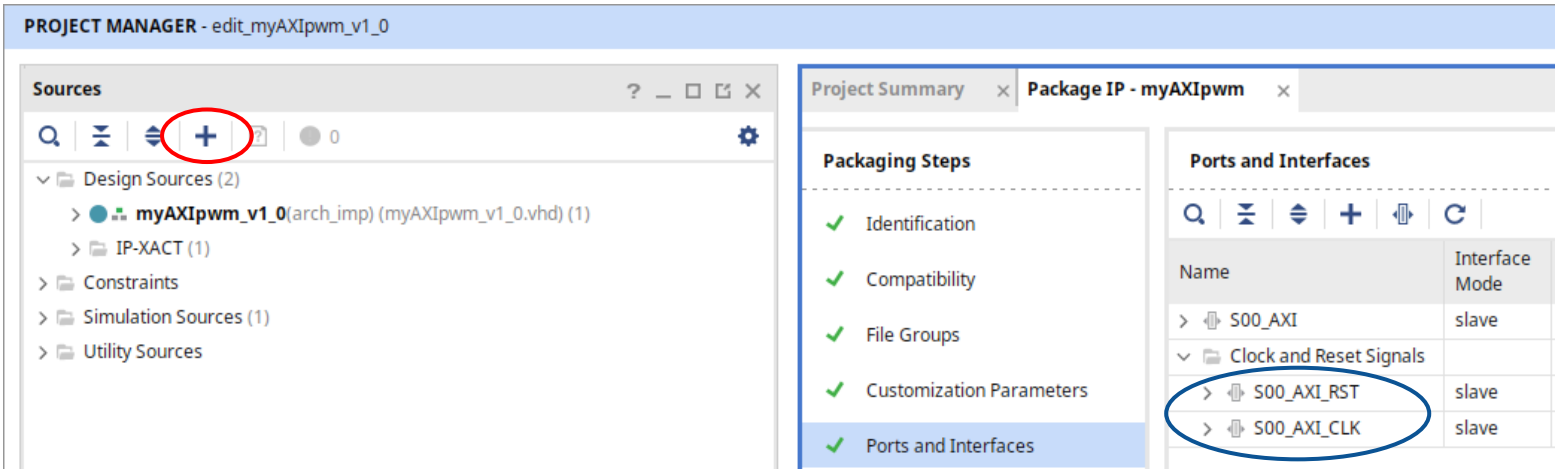
In the main window of the newly created project, you can see two files:

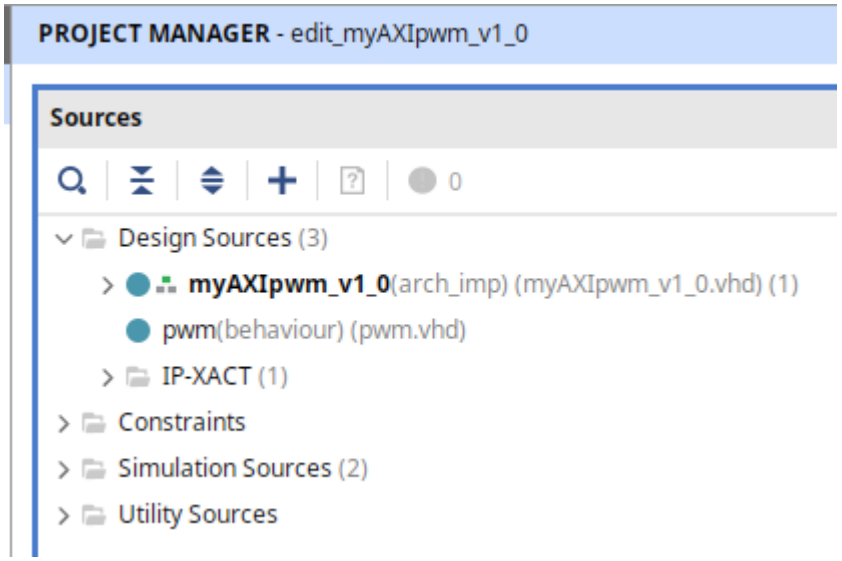
- the IP component `myAXIpwm_v1_0`
- `myAXIpwm_v1_0_S00_AXI` the AXI slave interface **component**, part of the IP component itself

myAXIpwm_v1_0 | the IP component



Now we'll add the source code of your previously designed PWM component:



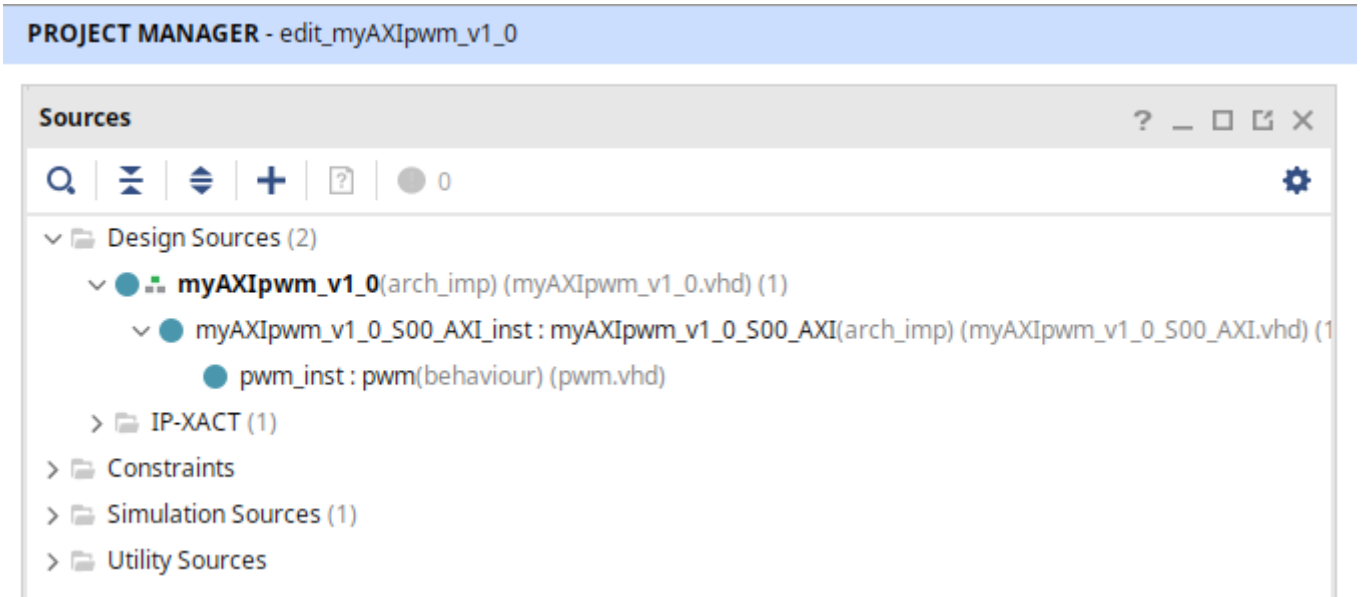


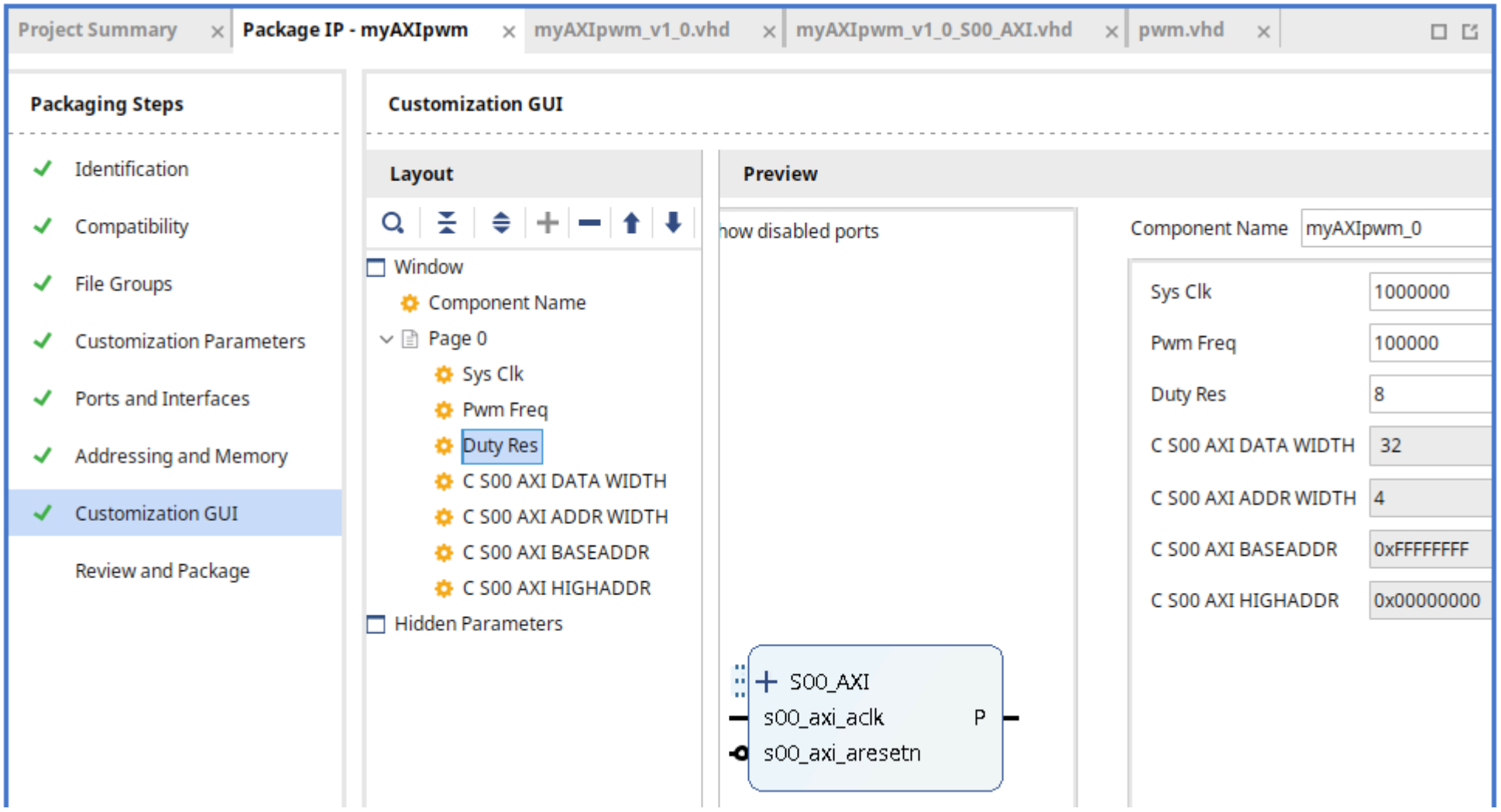
It's now time to integrate an instance of our original PWM within the AXI_S00 instance and to map ports + generic parameters … and extend AXI_S00_inst accordingly.

Finally, extend both generic parameters and ports of the top-level IP and map them to the AXI_S00 instance.

Add the various generic parameters as editable with a proper range of values.

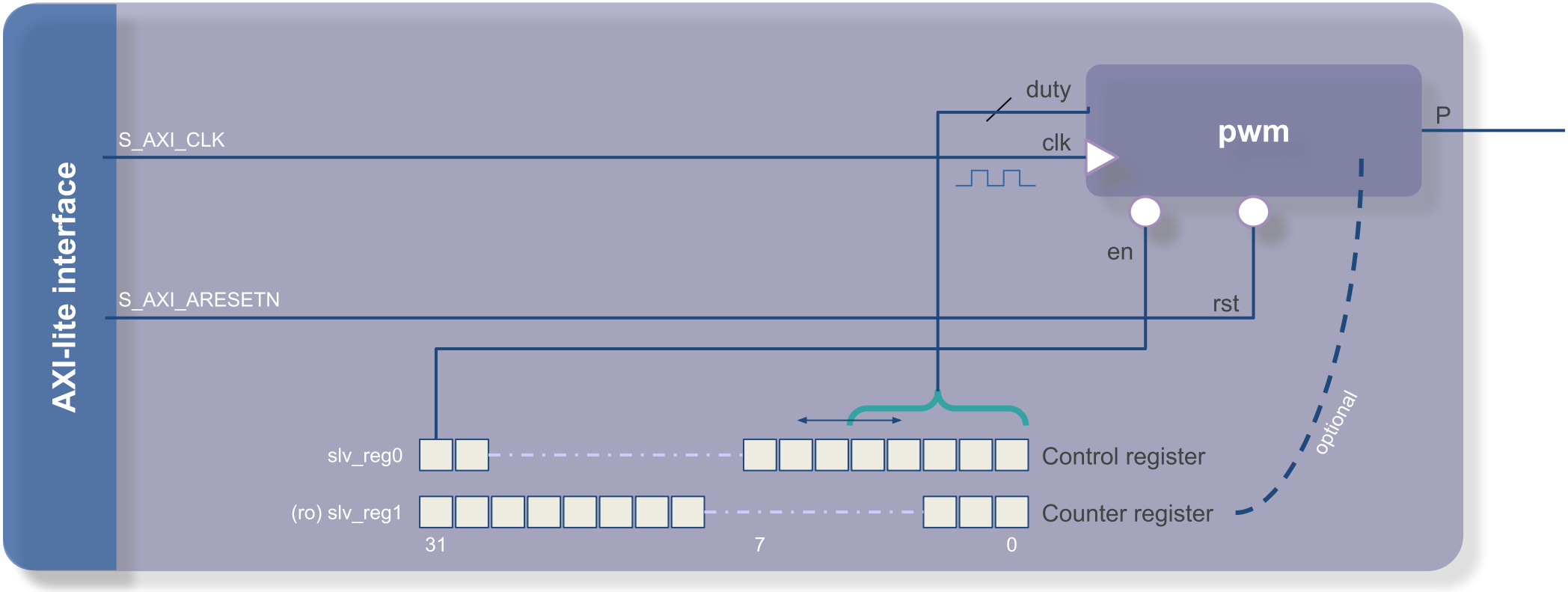
In the end, you'll see your pwm_inst integrated within the AXI interface instance





In order to help yourself: have a look to VHDL 'AXI' labelled files already available for download (see Links

Here are the details of the mapping between our AXI registers and our PWM component.
It's now time to complete the modification of our AXI interface.



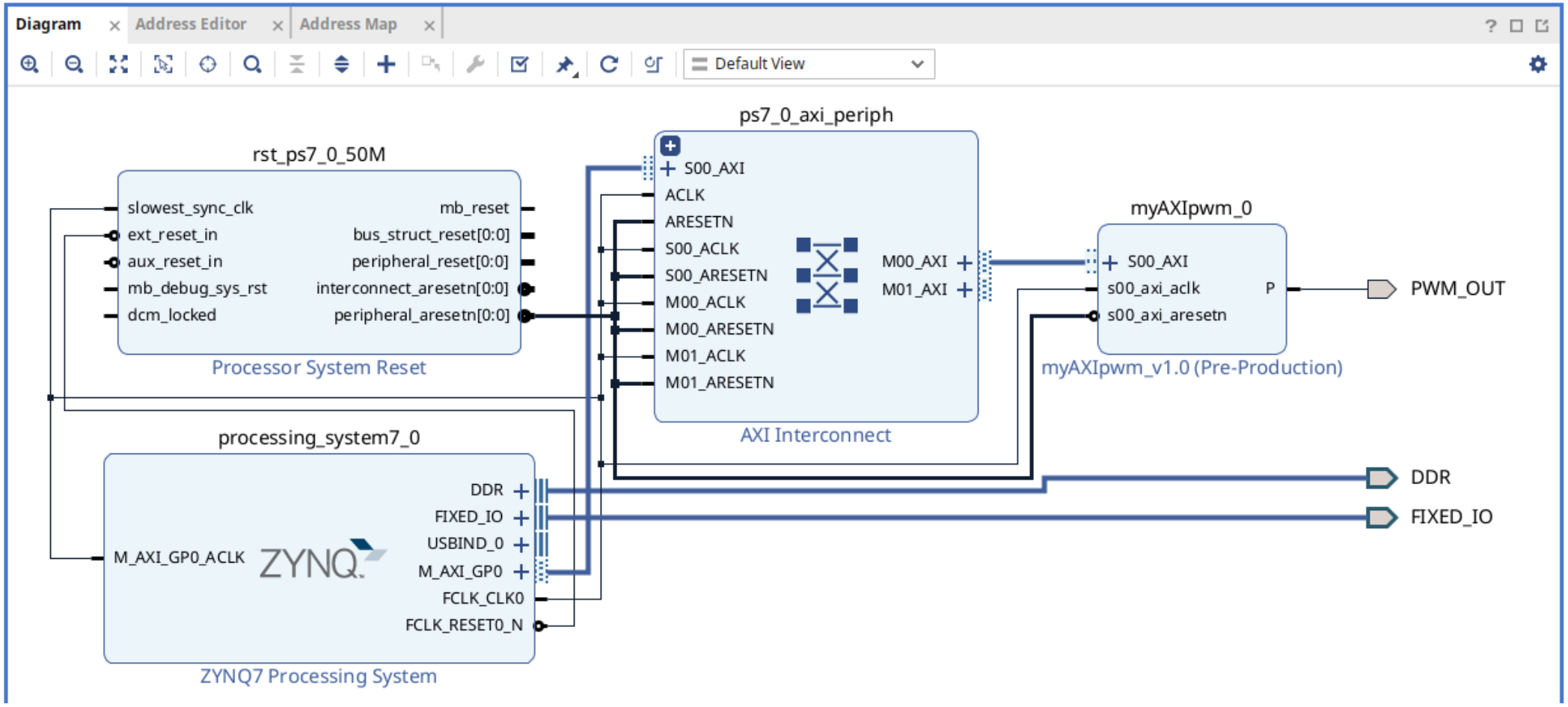
[optional] AXI simulation with the AXI VIP

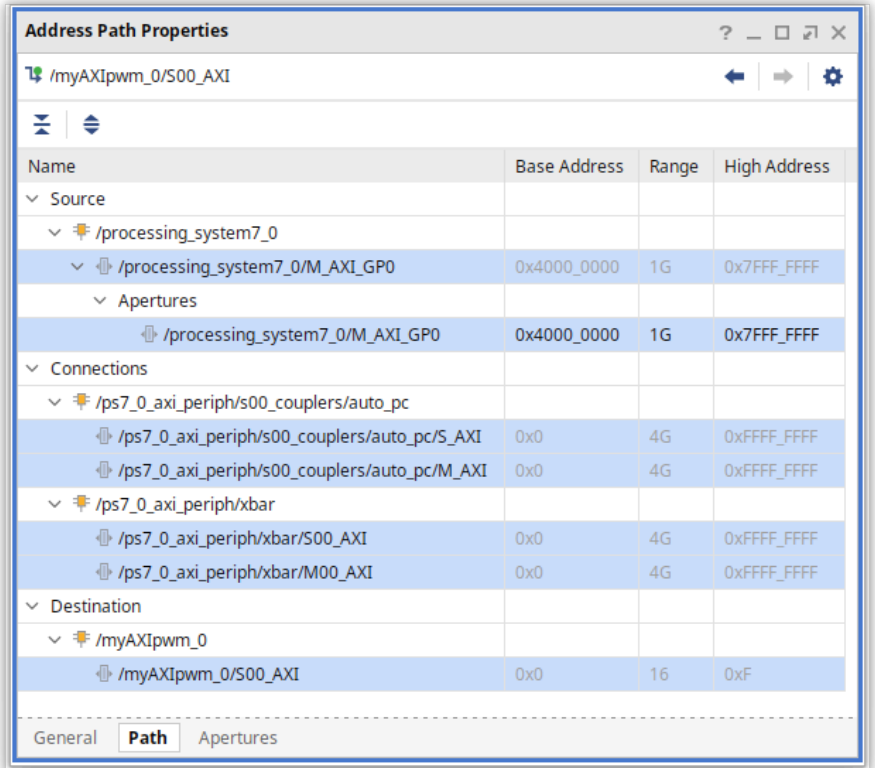
In order to enable a simple testing of your AXI-enable PWM IP, we'll make use of the AXI verification IP
https://support.xilinx.com/s/article/1058302?language=en_US

to be continued

Block design

It's now time to create a *Block Design* featuring the PS part (i.e Dual ARM9 CPU) and your AXI-enabled IP.
In the end, you'll need to *Create Port* (right-click in design) that will be named 'PWM_OUT': connect it with your PWM output.





Your PWM registers are set to 0x43C0_0000 (linux mmap later)
Using the *Address Editor*, you can change some parameters like the `range` value.

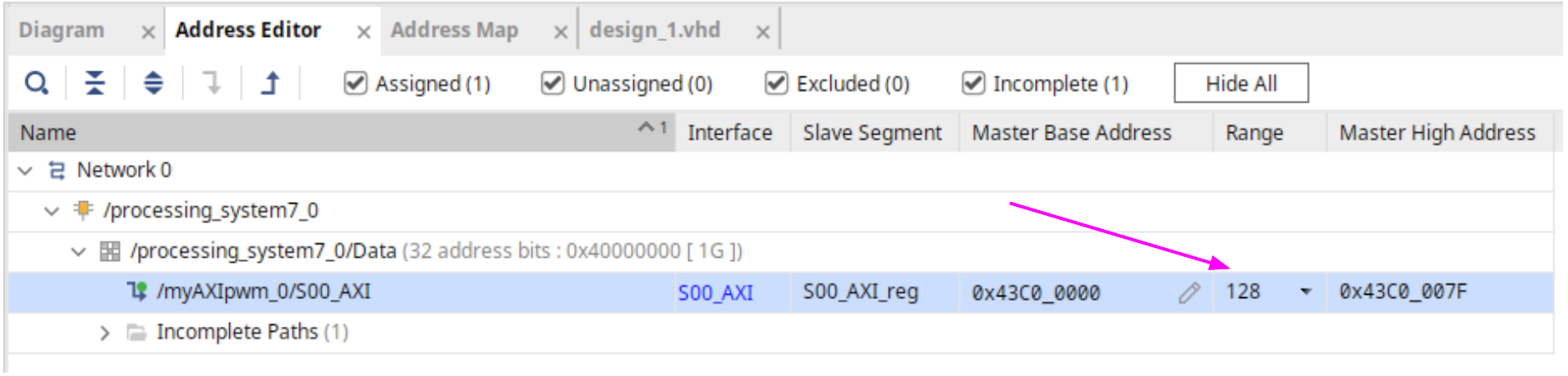
In the Block Design view, click to configure your `myAXIpwm_0`:

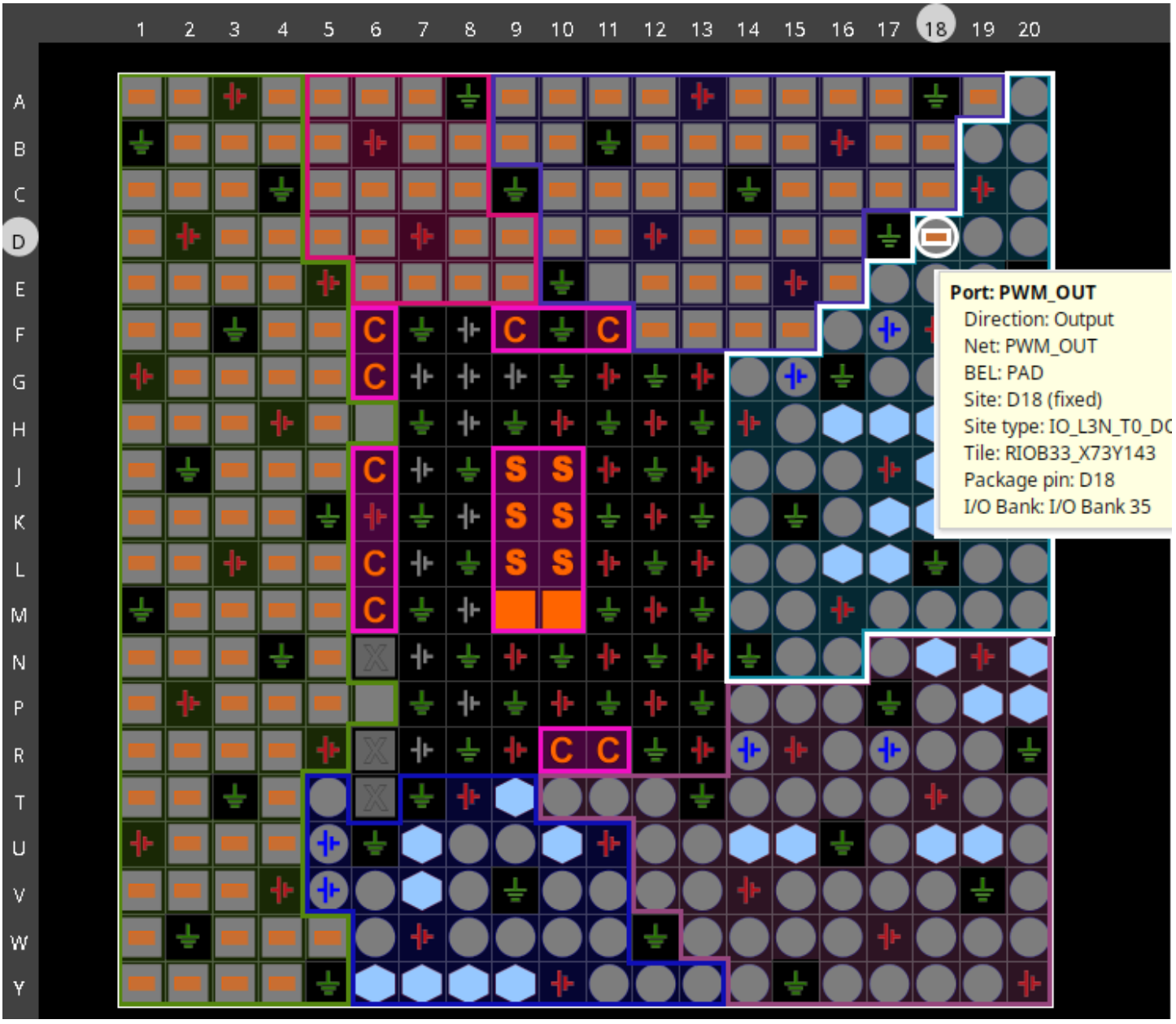
- sys_clk (50MHz),
- pwm_clk (1Hz),
- duty_res (4)

It's now time to *Validate your design* … then *Generate Block Design*

Add a constraint file to map the PWM_OUT to an onboard LED.
At the *Project Managerl*, *Design Source* right click on myAXIpwm → *add HDL wrapper*

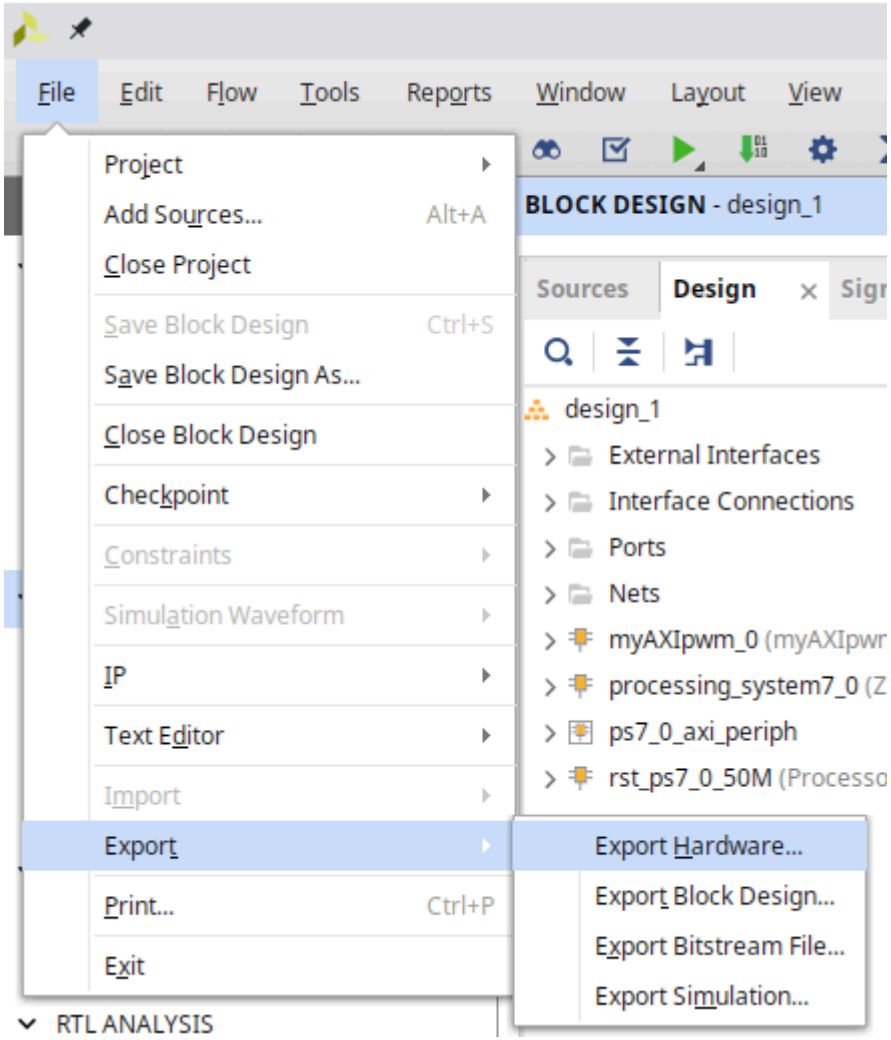
… then launch *Generate bitfile* (i.e it will undertake both IP synthesis, whole design synthesis + implementation too)

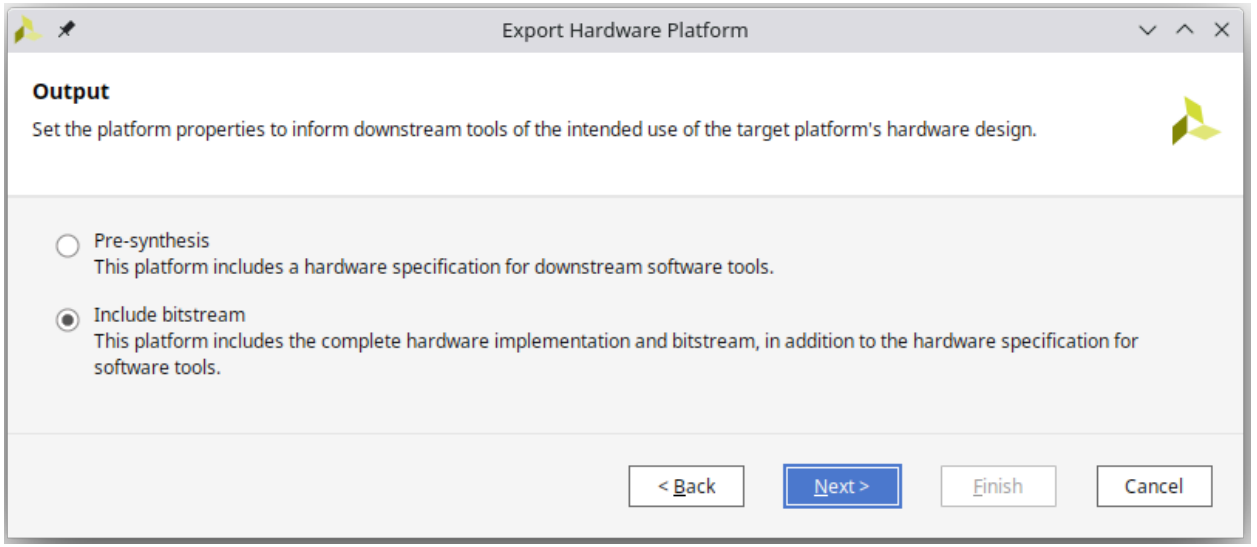


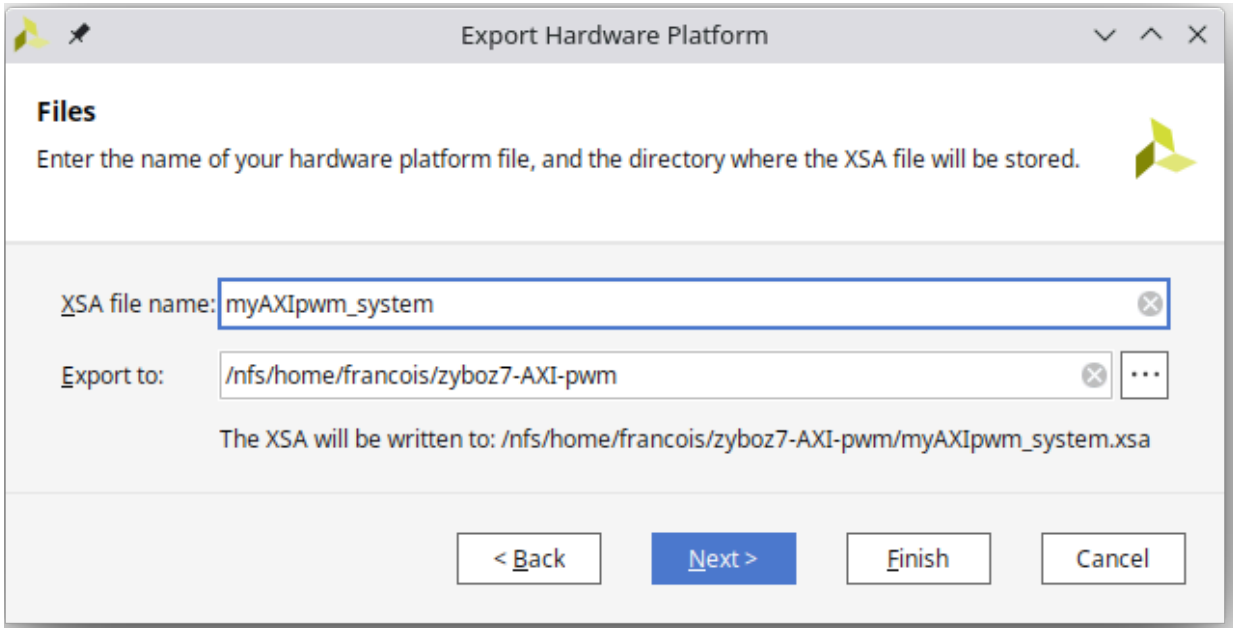


Bitstream generation and export hardware

As previously undertaken, see Bitstream generation to generate a `'.bit'` file.
Next step is to Export a hardware description of our global system in order to develop a simple, bare metal, application that will interact with our PWM IP.

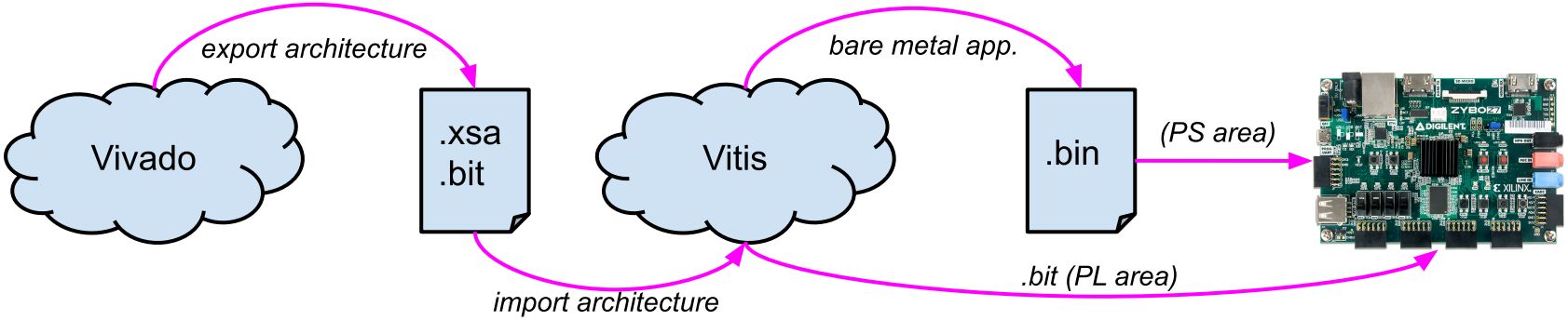

File
Edit
Flow
Tools
Reports
Window
Layout
View
Project
Add Sources... Alt+A
Close Project
Save Block Design Ctrl+S
Save Block Design As...
Close Block Design
Checkpoint
Constraints
Simulation Waveform
IP
Text Editor
Import
Export
Print... Ctrl+P
Exit
Export Hardware...
Export Block Design...
Export Bitstream File...
Export Simulation...
BLOCK DESIGN - design_1
Sources
Design
design_1
External Interfaces
Interface Connections
Ports
Nets
myAXIpwm_0 (myAXIpwr
processing_system7_0 (Z
ps7_0_axi_periph
rst_ps7_0_50M (Processo
RTL ANALYSIS

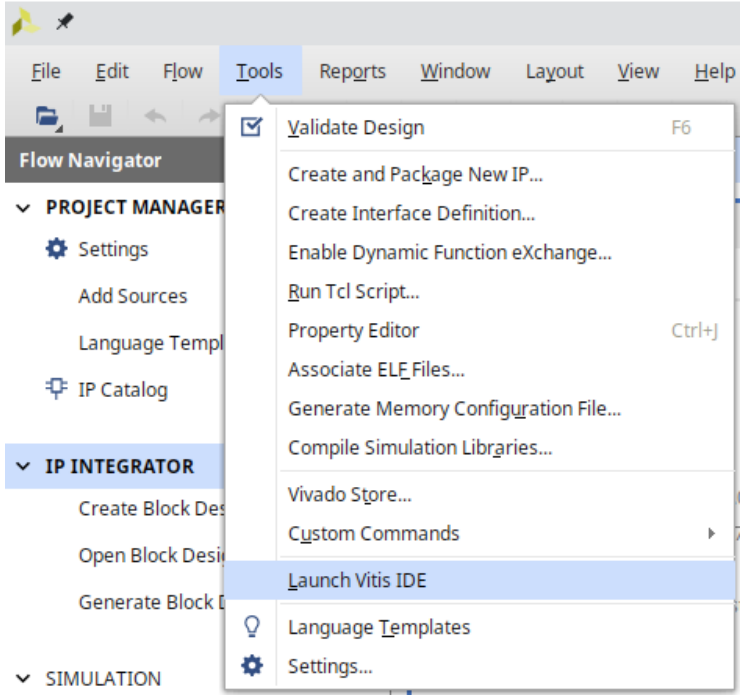

Export Hardware Platform
Output
Set the platform properties to inform downstream tools of the intended use of the target platform's hardware design.
Pre-synthesis
This platform includes a hardware specification for downstream software tools.
Include bitstream
This platform includes the complete hardware implementation and bitstream, in addition to the hardware specification for software tools.
< Back
Next >
Finish
Cancel

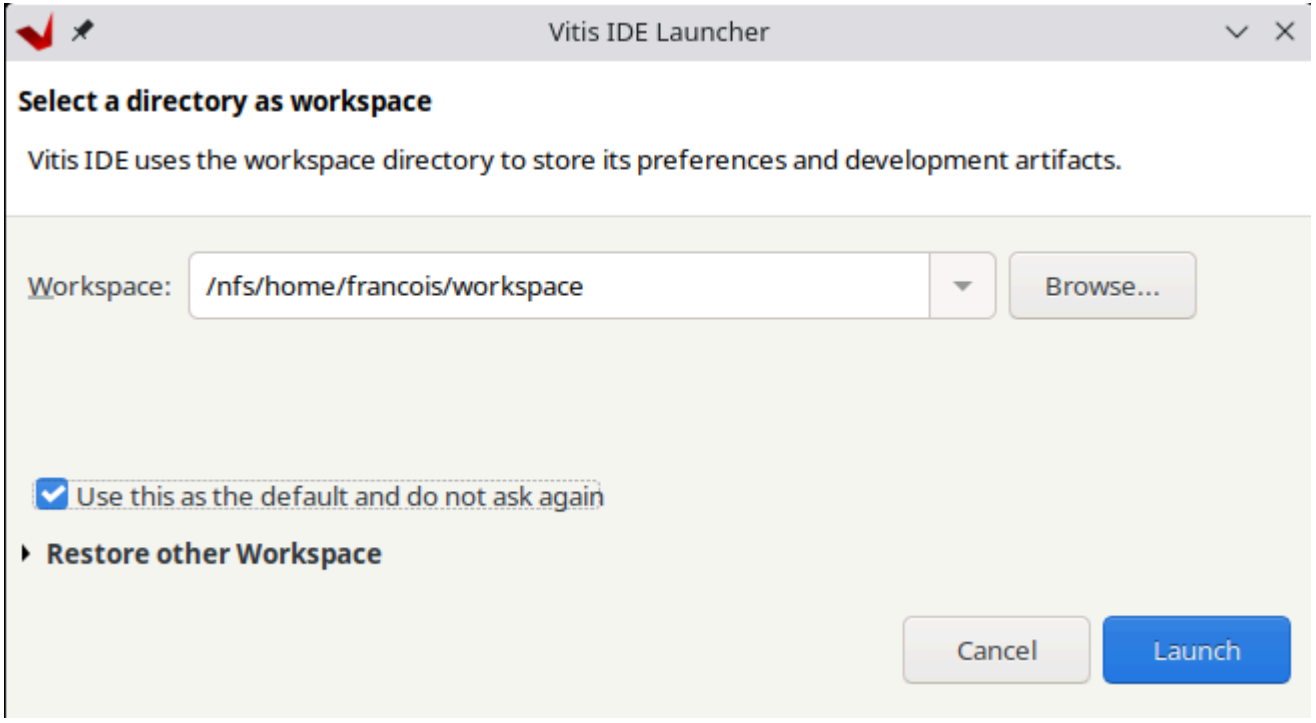

Export Hardware Platform
Files
Enter the name of your hardware platform file, and the directory where the XSA file will be stored.
XSA file name: myAXIpwm_system
Export to: /nfs/home/francois/zyboz7-AXI-pwm
The XSA will be written to: /nfs/home/francois/zyboz7-AXI-pwm/myAXIpwm_system.xsa
< Back
Next >
Finish
Cancel

Launch Vitis IDE

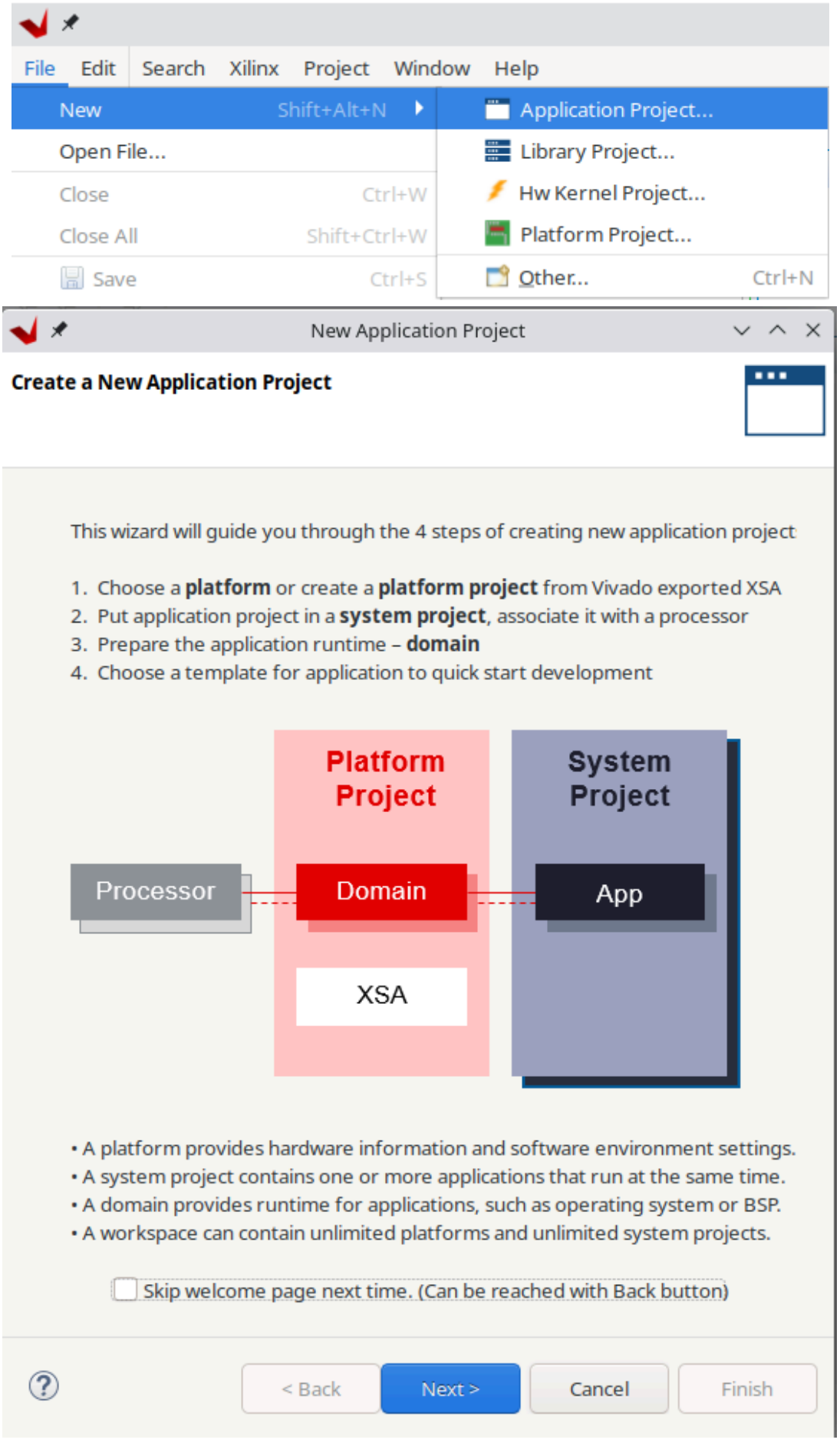
At this point, you have a `'.xsa'` file that describes our whole system featuring the *myPWM* IP block.
By means of the **Vitis IDE**, we'll import the hardware design to develop a bare metal app. able to interact with our PWM IP on its AXI bus.



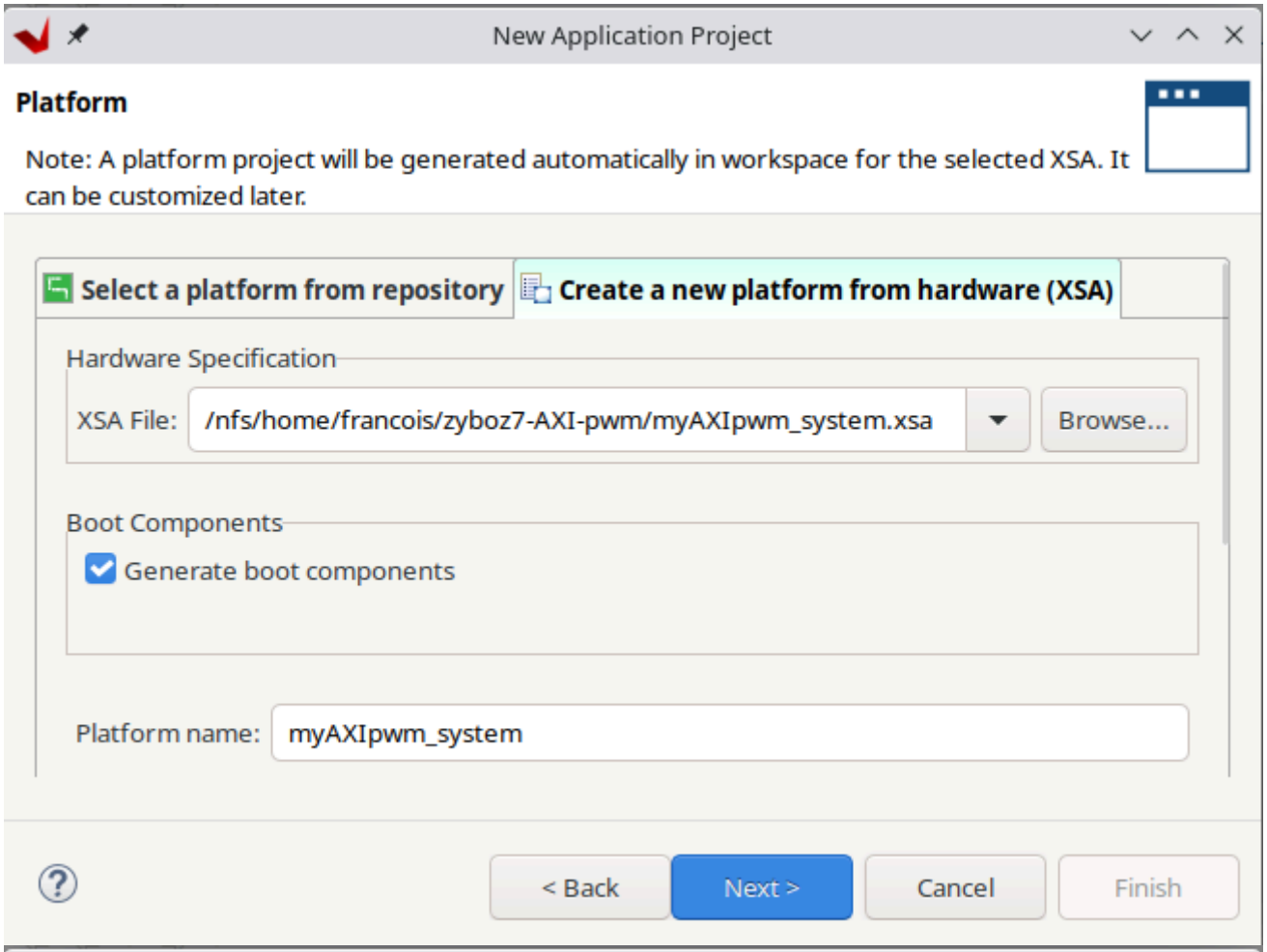


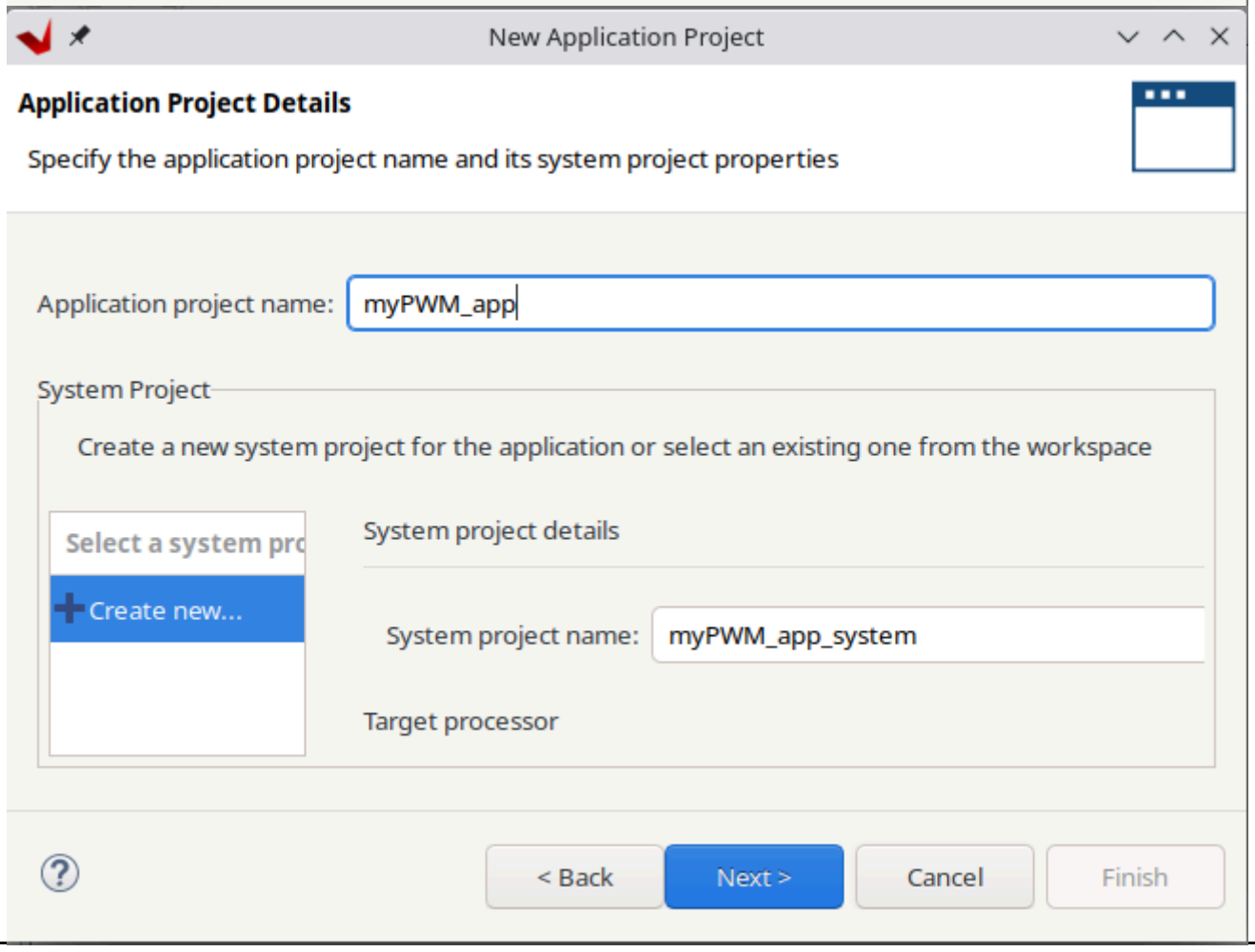


Create an '`application project`'



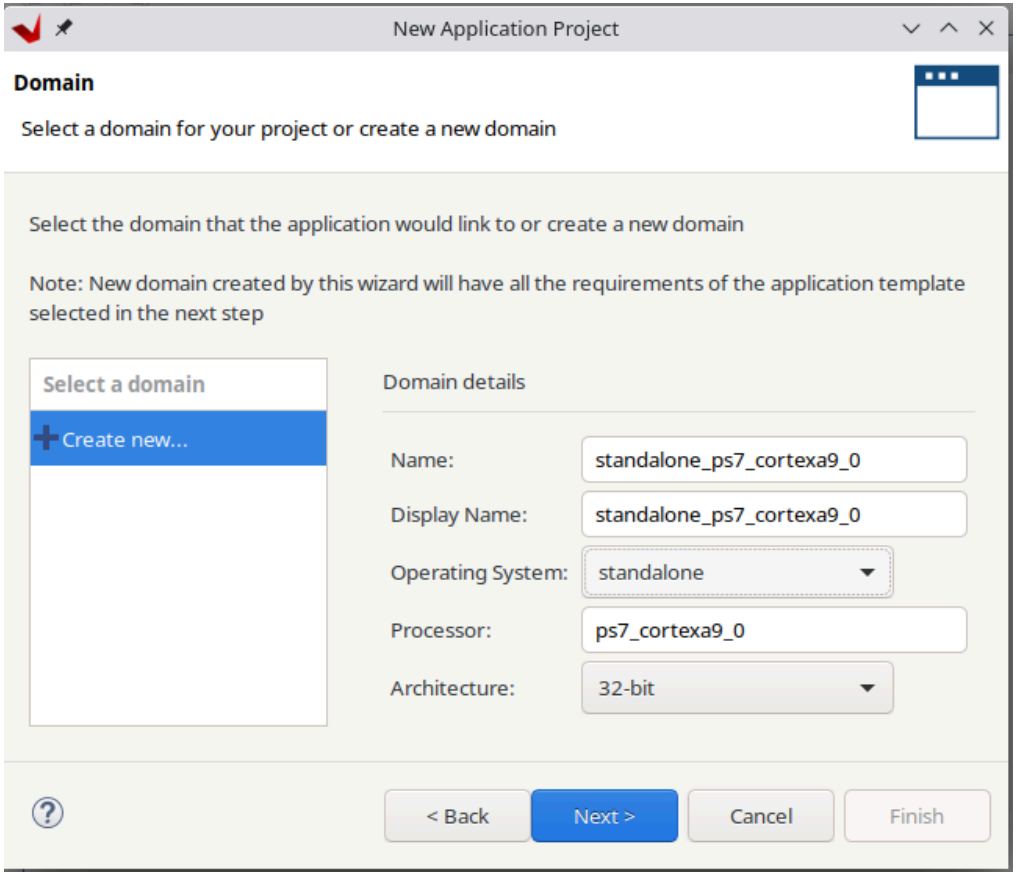
You'll then get prompted to create a new platform from your previously generated XSA file.

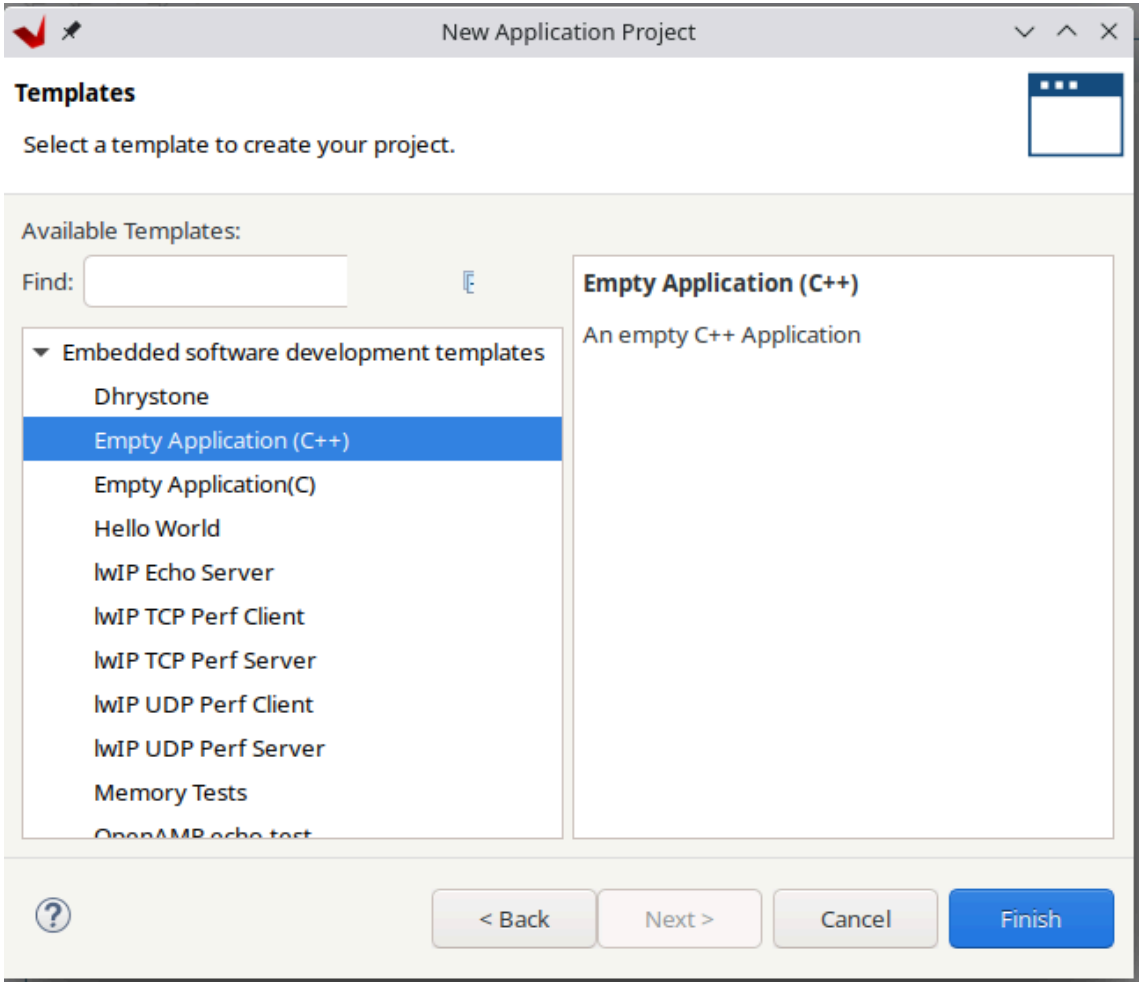




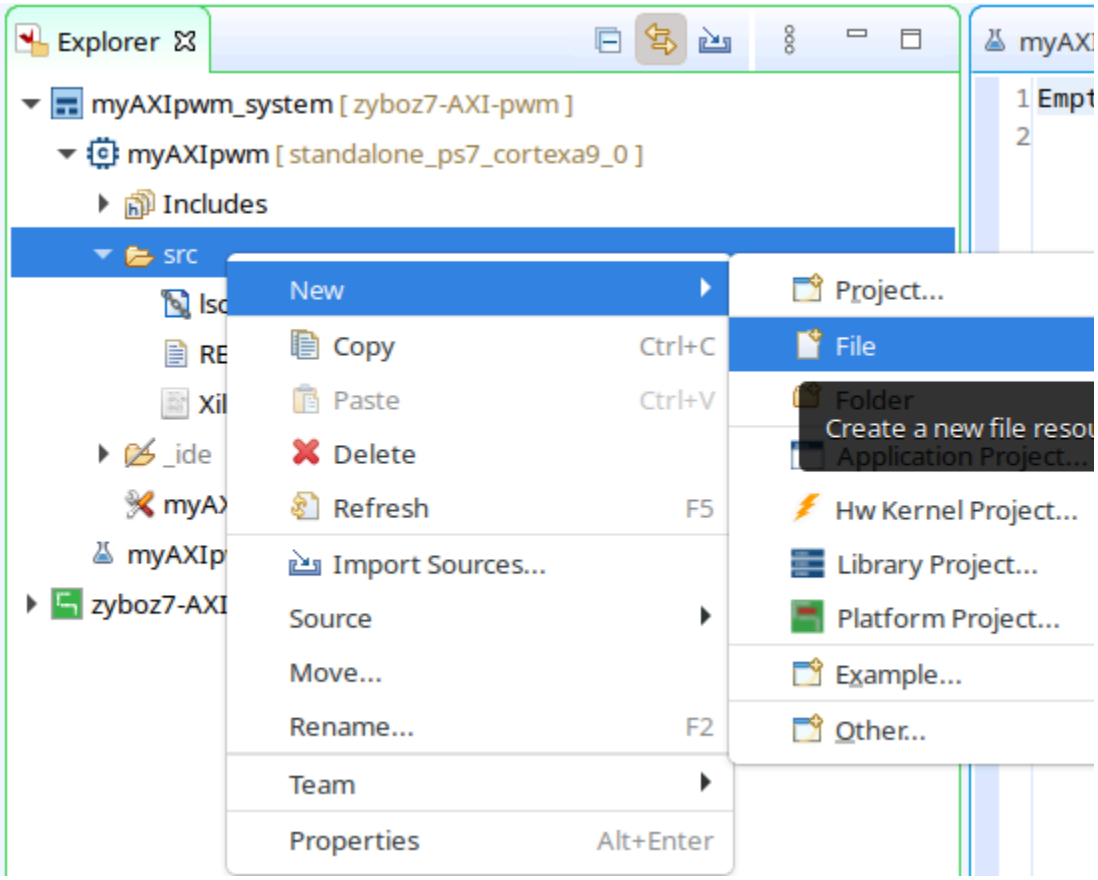
*

Once your application project name has been defined, you'll select `'standalone'` OS (i.e bare metal ---no OS) allong with '`Empty C++ app`'

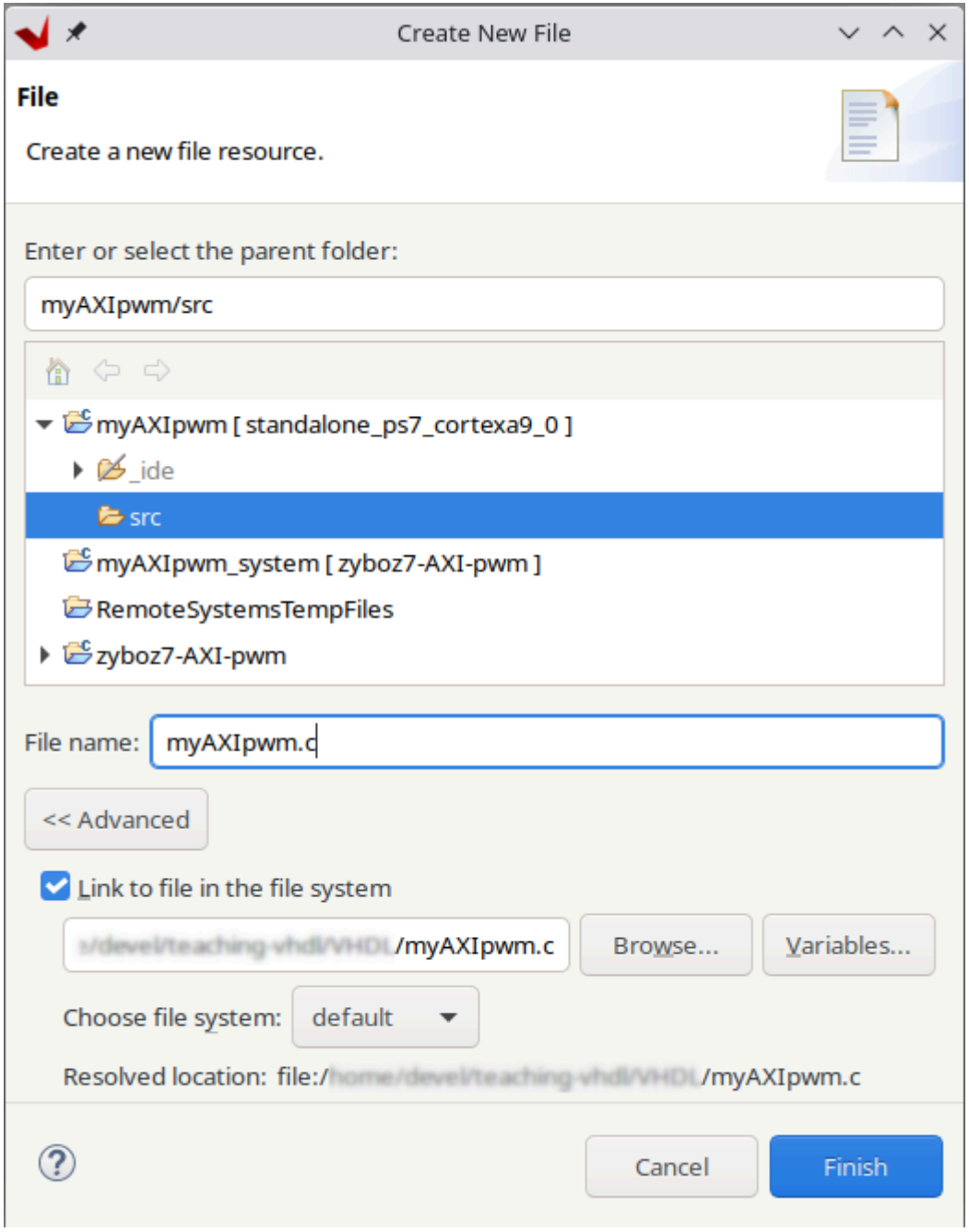




… then right click on the `src` folder to add a new file …

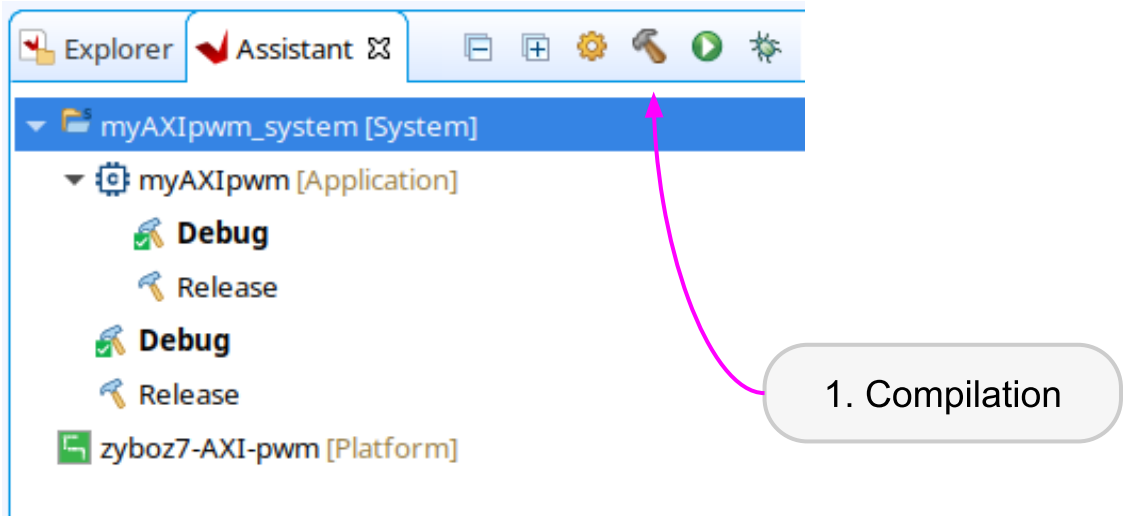


… click on ***Advanced*** to link to the existing `myAXIpwm.c` file you'll find from the [Practical exercises base files]



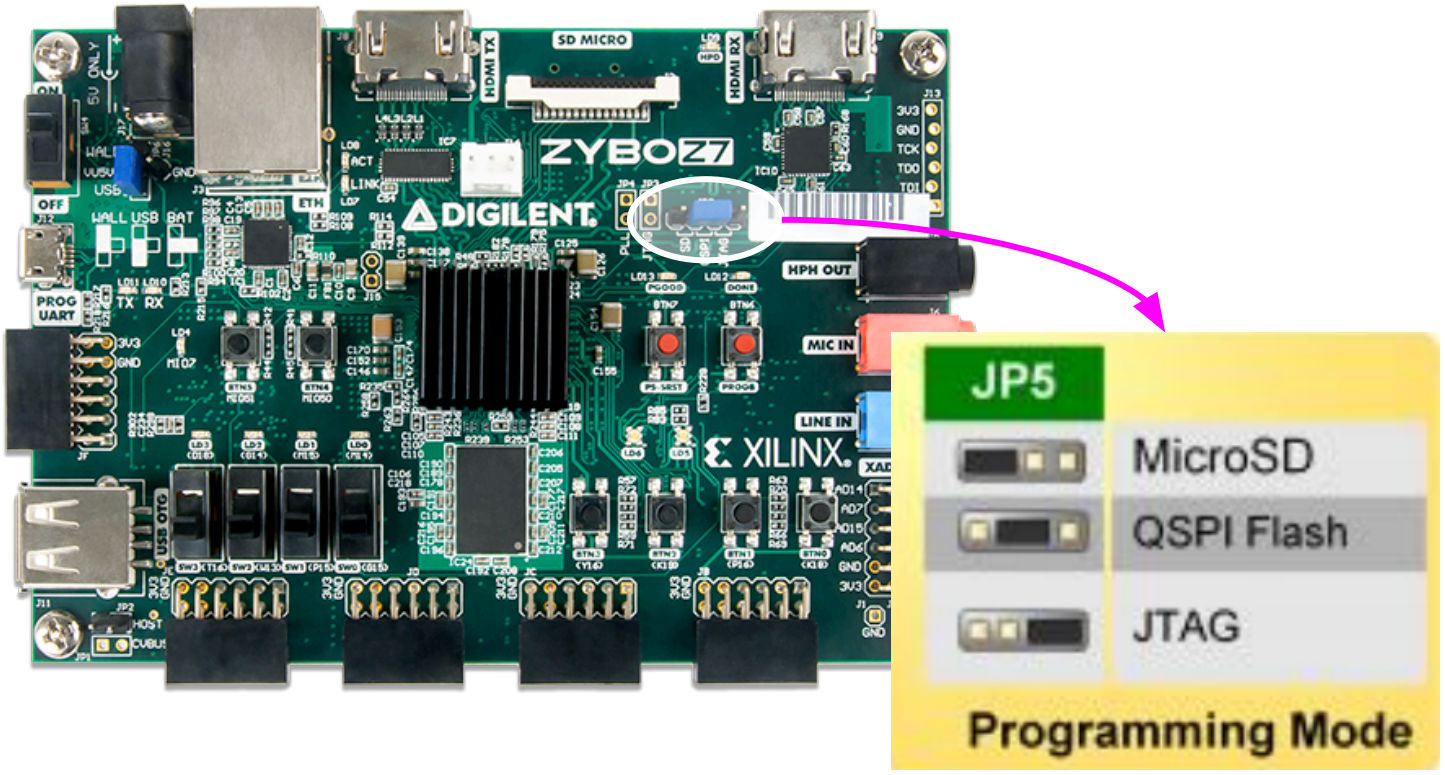
Compilation

Select `'myAXIpwn_system'` (it's important to select the system level), then click on build to launch the compilation … you may need to undertake some corrections within your C++ source code that will interact with your IP block.

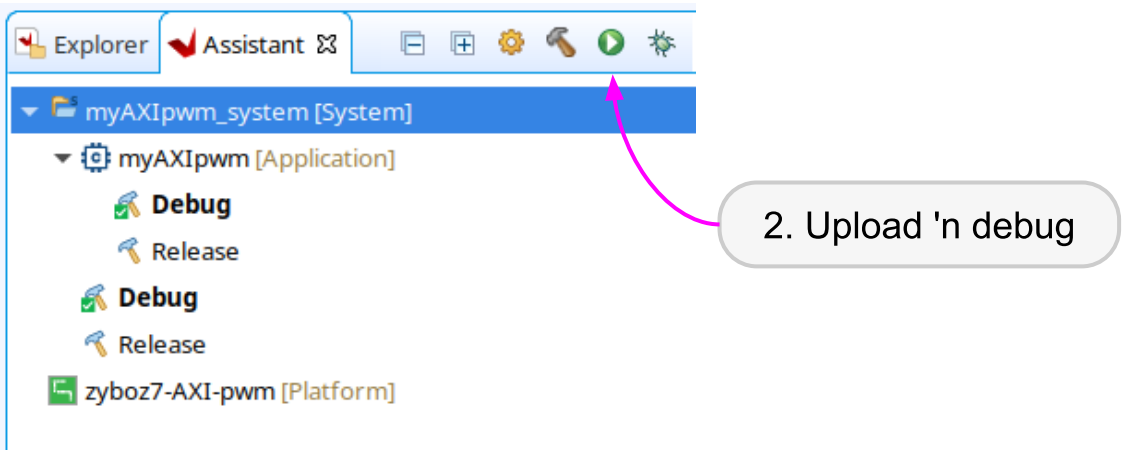


Zybo board | JTAG mode

Before testing your whole project on your hardware, set **JP5** to **JTAG** mode before powering on the Zybo board.



… then upload both FPGA configuration along with your standalone app. by clicking on the RUN icon within the assistant frame



You can follow the whole upload within the XSCT console

XSCT console

```
attempting to launch hw_server

****** Xilinx hw_server v2022.1.0
  **** Build date : Apr 10 2022 at 06:24:21
    ** Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.

INFO: hw_server application started
INFO: Use Ctrl-C to exit hw_server application

INFO: To connect to this hw_server instance use url: TCP:127.0.0.1:3121


initializing
  0%     0MB   0.0MB/s  ??:?? ETA
 28%     1MB   2.2MB/s  ??:?? ETA
 50%     1MB   1.9MB/s  ??:?? ETA
 72%     2MB   1.8MB/s  ??:?? ETA
 93%     3MB   1.8MB/s  ??:?? ETA
100%     3MB   1.8MB/s  00:02

Downloading Program -- /home/devel/workspace/myAXIpwm/Debug/myAXIpwm.elf
        section, .text: 0x00100000 - 0x001035db
        section, .init: 0x001035dc - 0x001035e7
        section, .fini: 0x001035e8 - 0x001035f3
        section, .rodata: 0x001035f4 - 0x00103798
        section, .data: 0x001037a0 - 0x0010401f
        section, .eh_frame: 0x00104020 - 0x00104023
        section, .mmu_tbl: 0x00108000 - 0x0010bfff
        section, .ARM.exidx: 0x0010c000 - 0x0010c0c7
        section, .init_array: 0x0010c0c8 - 0x0010c0cb
        section, .fini_array: 0x0010c0cc - 0x0010c0cf
        section, .bss: 0x0010c0d0 - 0x0010c133
        section, .heap: 0x0010c134 - 0x0010e13f
        section, .stack: 0x0010e140 - 0x0011193f

  0%     0MB   0.0MB/s  ??:?? ETA
100%     0MB   0.5MB/s  00:00
Setting PC to Program Start Address 0x00100000
Successfully downloaded /home/devel/workspace/myAXIpwm/Debug/myAXIpwm.elf
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0xffffff28 (Suspended)
xsct% Info: ARM Cortex-A9 MPCore #0 (target 2) Running
```

… if everything goes right, you now have a 1s blinking led 🙂

[M2] AXI IP encoder

TO BE CONTINUED

[M2] PicoRV32

PicoRV32 https://github.com/YosysHQ/picorv32

END