

Getting Started with tinyTPU

with Xilinx Zynq SoC

© Jonas Fuhrmann

2018-11-22

Create a Project	2
Create AXI IP Core	2
Create Block Design	5
Synthesise the Design	8
Implement the Design	9
Add Sample Project in Xilinx SDK	11
Create FSBL and Boot Image	13
Load and execute the Sample Model	14

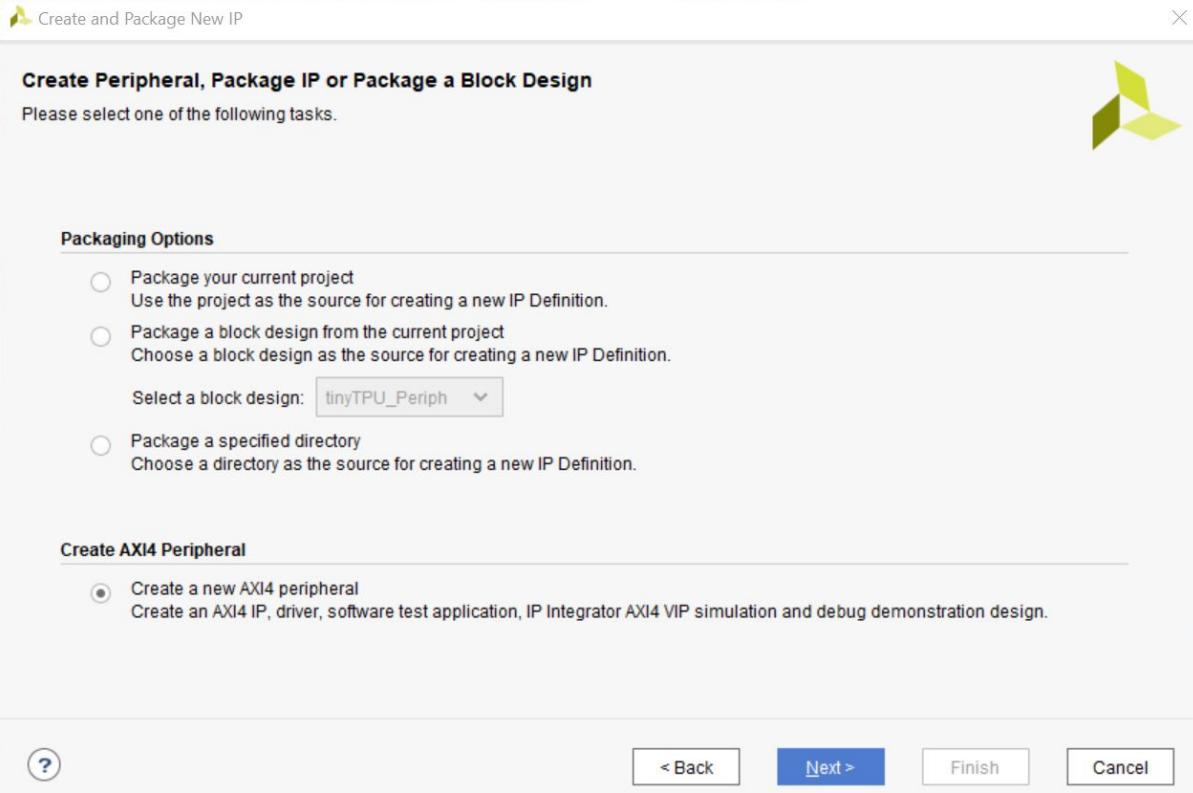
Create a Project

Open Vivado and create a new Project. Give it a name and choose the board/part you want to use. In our case, we are using the MYIR z-turn board with Zynq 7020 SoC. Board files for this board can be found here: <https://github.com/q3k/zturn-stuff>

Create AXI IP Core

Click on *Tools* → *Create and Package new IP*.

Choose *Create a new AXI4 peripheral* on the next slide.



Create and Package New IP

Create Peripheral, Package IP or Package a Block Design

Please select one of the following tasks.

Packaging Options

- ☐ Package your current project
Use the project as the source for creating a new IP Definition.
- ☐ Package a block design from the current project
Choose a block design as the source for creating a new IP Definition.
Select a block design:
- ☐ Package a specified directory
Choose a directory as the source for creating a new IP Definition.

Create AXI4 Peripheral

- ☒ Create a new AXI4 peripheral
Create an AXI4 IP, driver, software test application, IP Integrator AXI4 VIP simulation and debug demonstration design.

? < Back Next > Finish Cancel

Name your IP *tinyTPU* with version 1.0 and click next.

Choose the AXI4 Lite Slave interface type and use a 32-Bit data width. The name should be S00_AXI.

The screenshot shows the 'Add Interfaces' tab of the 'Create and Package New IP' dialog. On the left, there is a checkbox for 'Enable Interrupt Support' and a diagram showing the 'S00_AXI' interface connected to the 'tinyTPU_v1.1' block. In the center, a tree view under 'Interfaces' shows 'S00_AXI' selected. On the right, configuration fields are set: Name is 'S00_AXI', Interface Type is 'Lite', Interface Mode is 'Slave', Data Width (Bits) is '32', Memory Size (Bytes) is '64', and Number of Registers is '4'. The 'Next >' button is highlighted in blue.

Click *Edit IP* and a new project will open up.

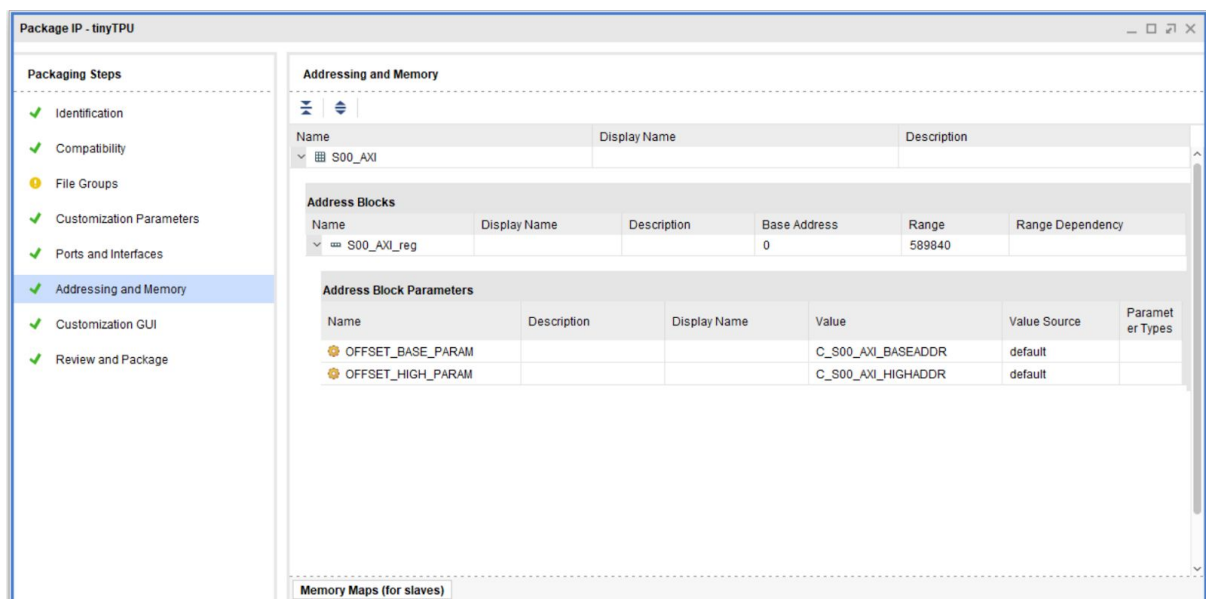
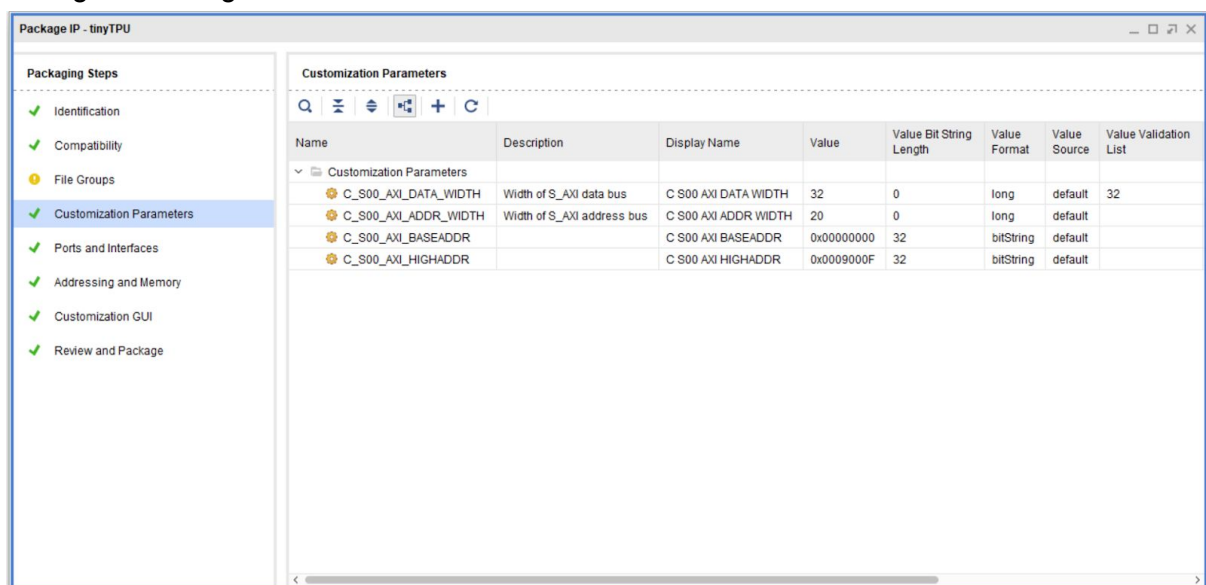
The screenshot shows the 'Create Peripheral' tab of the 'Create and Package New IP' dialog. It displays a 'Peripheral Generation Summary' with four items: 1. IP (xilinx.com:user.tinyTPU:1.1) with 1 interface(s), 2. Driver(v1_00_a) and testapp, 3. AXI4 VIP Simulation demonstration design, and 4. AXI4 Debug Hardware Simulation demonstration design. Below this, it states the peripheral will be available in the catalog at 'C:/Users/Jonas/Documents/tinyTPU/VIVADO/tinyTPU_z_turn/..ip_repo'. Under 'Next Steps', four options are listed: 'Add IP to the repository', 'Edit IP' (which is selected with a radio button), 'Verify Peripheral IP using AXI4 VIP', and 'Verify peripheral IP using JTAG interface'. At the bottom, it says 'Click Finish to continue'. The 'Finish' button is highlighted in blue.

In the new project remove the existing VHDL files and add all VHDL files from *src/vhdl*, excluding all files starting with *TB_*. The type of the entities *RUNTIME_COUNTER*, *ACTIVATION*, *WEIGHT_CONTROL*, *DSP_COUNTER* and *MATRIX_MULTIPLY_CONTROL* should be changed to *VHDL 2008* from *VHDL*.

The TPU is configured to use all possible DSP and BRAM resources on the Zynq 7020. If you want to change the size of the TPU, you should change the constants *MATRIX_WIDTH*, *WEIGHT_BUFFER_DEPTH* and *UNIFIED_BUFFER_DEPTH* in *tinyTPU_v1_0_S00_AXI.vhd* as desired.

Press *Merge File Groups* in *Packaging Steps*.

Change all settings as shown below.



Merge all changes and press *Re-Package IP*. The project can then be closed.

Create Block Design

In the main project, click *Create Block Design*. Add the *ZYNQ7 Processing System* and apply the board presets. Click on the Zynq IP and change the fabric clock as seen below.

ZYNQ7 Processing System (5.5)

Documentation Presets IP Location Import XPS Settings

Page Navigator

- Zynq Block Design
- PS-PL Configuration
- Peripheral I/O Pins
- MIO Configuration
- Clock Configuration**
- DDR Configuration
- SMC Timing Calculation
- Interrupts

Clock Configuration [Summary Report](#)

Basic Clocking **Advanced Clocking**

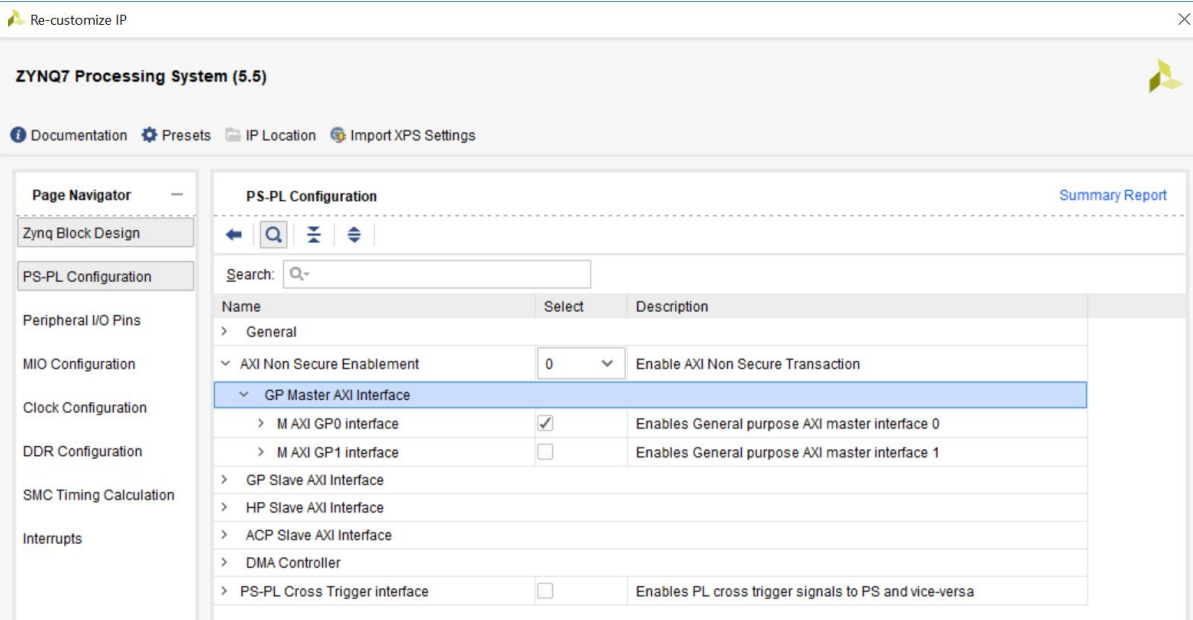
Input Frequency (MHz) 33.333333 CPU Clock Ratio 6:2:1

Search: Q-

Component	Clock Source	Requested Frequ...	Actual Frequency(...)	Range(MHz)
> Processor/Memory Clocks				
> IO Peripheral Clocks				
▼ PL Fabric Clocks				
<input checked="" type="checkbox"/> FCLK_CLK0	IO PLL	180	177.777771	0.100000 : 250.000000
<input type="checkbox"/> FCLK_CLK1	IO PLL	50	10.000000	0.100000 : 250.000000
<input type="checkbox"/> FCLK_CLK2	IO PLL	50	10.000000	0.100000 : 250.000000
<input type="checkbox"/> FCLK_CLK3	IO PLL	50	10.000000	0.100000 : 250.000000
> System Debug Clocks				
> Timers				

You may be able to reach higher clock speeds than 177.77 MHz, but this was the maximum, when using all DSP blocks and BRAM on Zynq 7020.

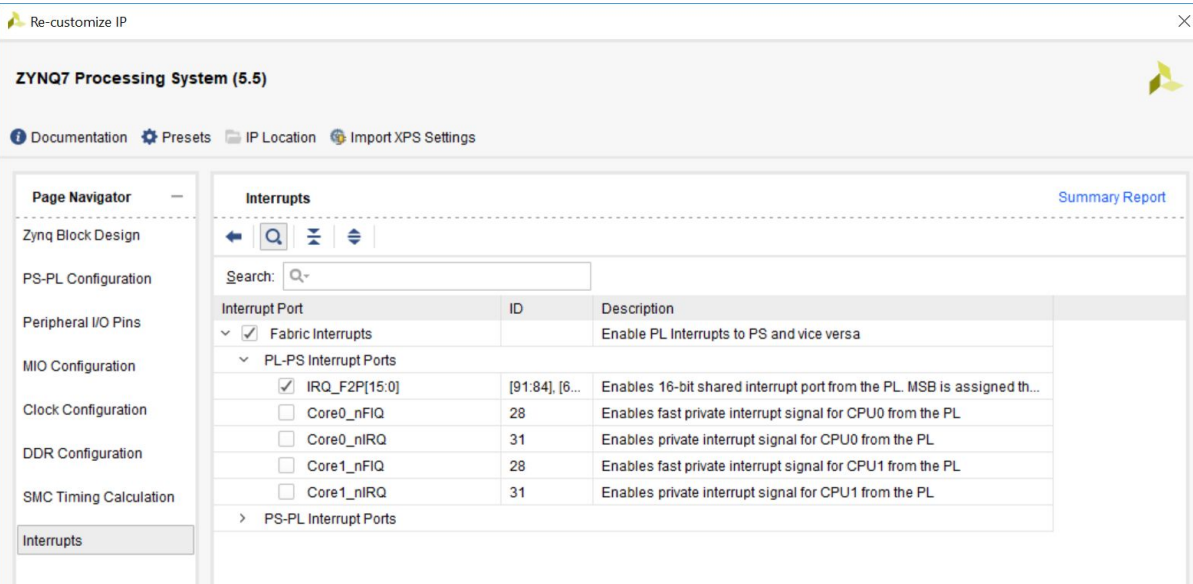
Make sure that one *General Purpose Master AXI Interface* is enabled.



The screenshot shows the 'Re-customize IP' window for the 'ZYNQ7 Processing System (5.5)'. The 'Page Navigator' on the left lists various configuration options, with 'PS-PL Configuration' selected. The main panel displays the 'PS-PL Configuration' settings. A search bar is at the top. Below it, a table lists configuration items:

Name	Select	Description
> General		
> AXI Non Secure Enablement	0	Enable AXI Non Secure Transaction
> GP Master AXI Interface		
> M AXI GP0 interface	<input checked="" type="checkbox"/>	Enables General purpose AXI master interface 0
> M AXI GP1 interface	<input type="checkbox"/>	Enables General purpose AXI master interface 1
> GP Slave AXI Interface		
> HP Slave AXI Interface		
> ACP Slave AXI Interface		
> DMA Controller		
> PS-PL Cross Trigger interface	<input type="checkbox"/>	Enables PL cross trigger signals to PS and vice-versa

Enable the *Fabric Interrupts* as shown below and press OK.

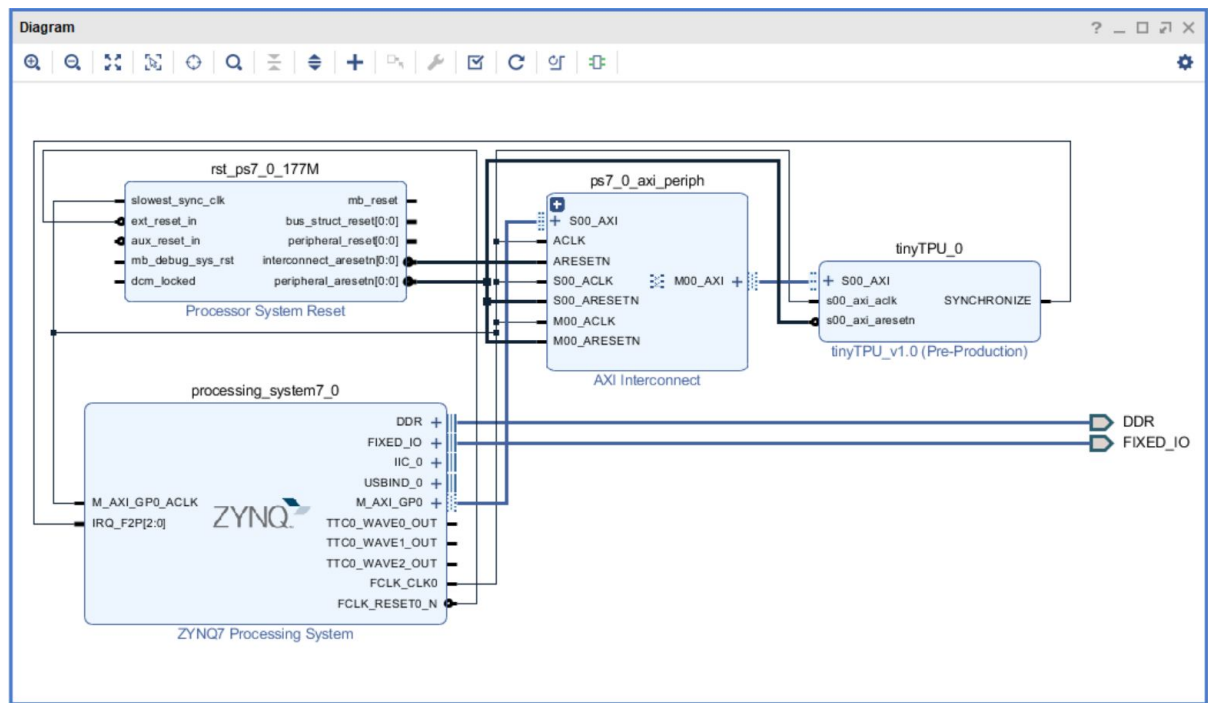


The screenshot shows the 'Re-customize IP' window for the 'ZYNQ7 Processing System (5.5)'. The 'Page Navigator' on the left lists various configuration options, with 'Interrupts' selected. The main panel displays the 'Interrupts' settings. A search bar is at the top. Below it, a table lists configuration items:

Interrupt Port	ID	Description
> <input checked="" type="checkbox"/> Fabric Interrupts		Enable PL Interrupts to PS and vice versa
> PL-PS Interrupt Ports		
<input checked="" type="checkbox"/> IRQ_F2P[15:0]	[91:84], [6...	Enables 16-bit shared interrupt port from the PL. MSB is assigned th...
<input type="checkbox"/> Core0_nFIQ	28	Enables fast private interrupt signal for CPU0 from the PL
<input type="checkbox"/> Core0_nIRQ	31	Enables private interrupt signal for CPU0 from the PL
<input type="checkbox"/> Core1_nFIQ	28	Enables fast private interrupt signal for CPU1 from the PL
<input type="checkbox"/> Core1_nIRQ	31	Enables private interrupt signal for CPU1 from the PL
> PS-PL Interrupt Ports		

Now add the *tinyTPU AXI IP* and press *Run Block Automation*. The tool should then connect the AXI Interfaces correctly by inferring a *Processor System Reset* and *AXI Interconnect*. You then have to connect the *SYNCHRONIZE* signal with the Zynq's *IRQ_F2P* port.

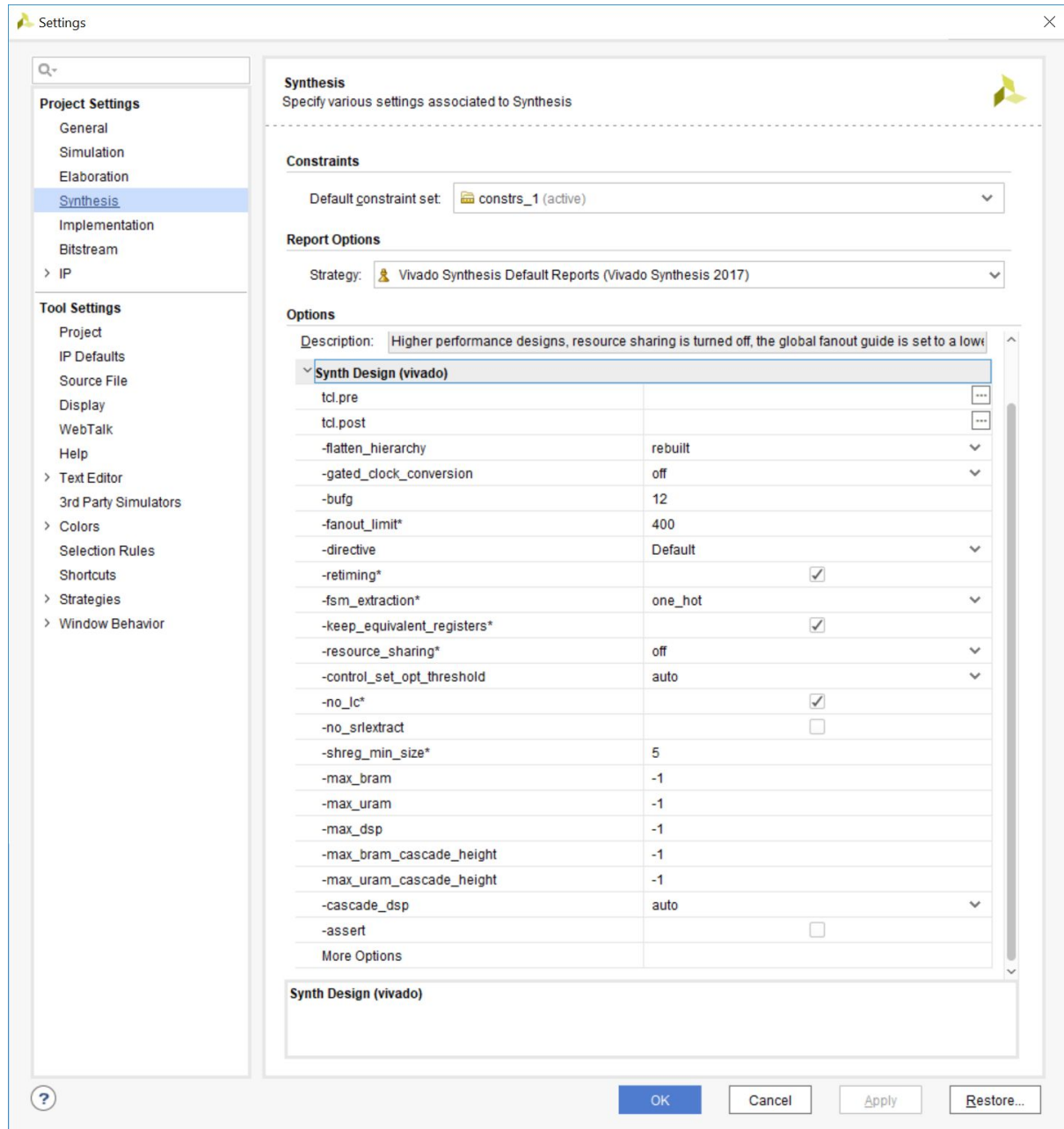
The block design should look like this:



Press *Validate Design*. There shouldn't be any error. Close the block design and generate the output product as *Global*. Point at the newly created block design in *Sources* and choose *Create HDL Wrapper* by right clicking and let Vivado handle the generation.

Synthesise the Design

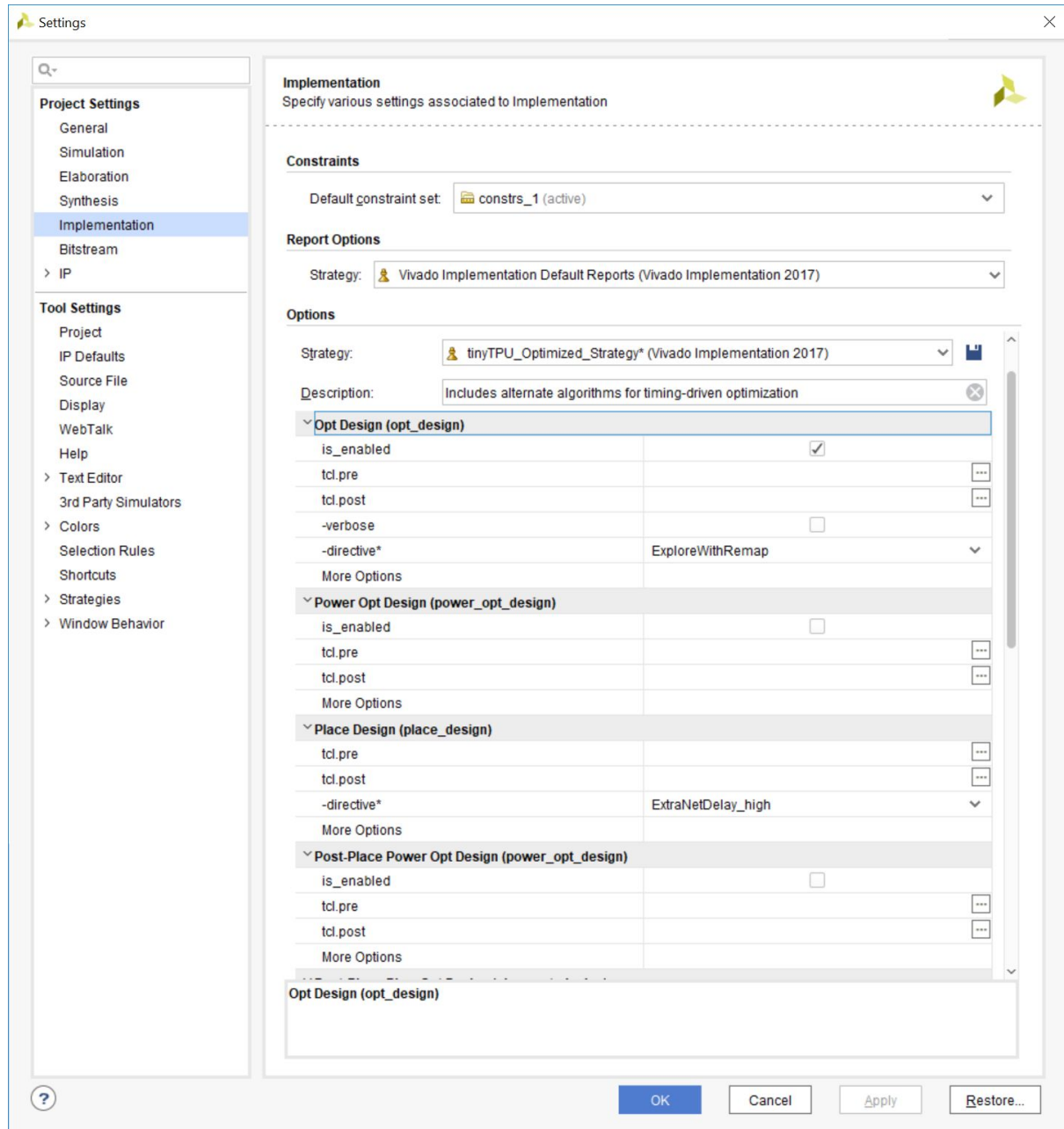
Open *Settings* and click on *Synthesis*. Change the settings shown below. This will help to reach the frequency goal.

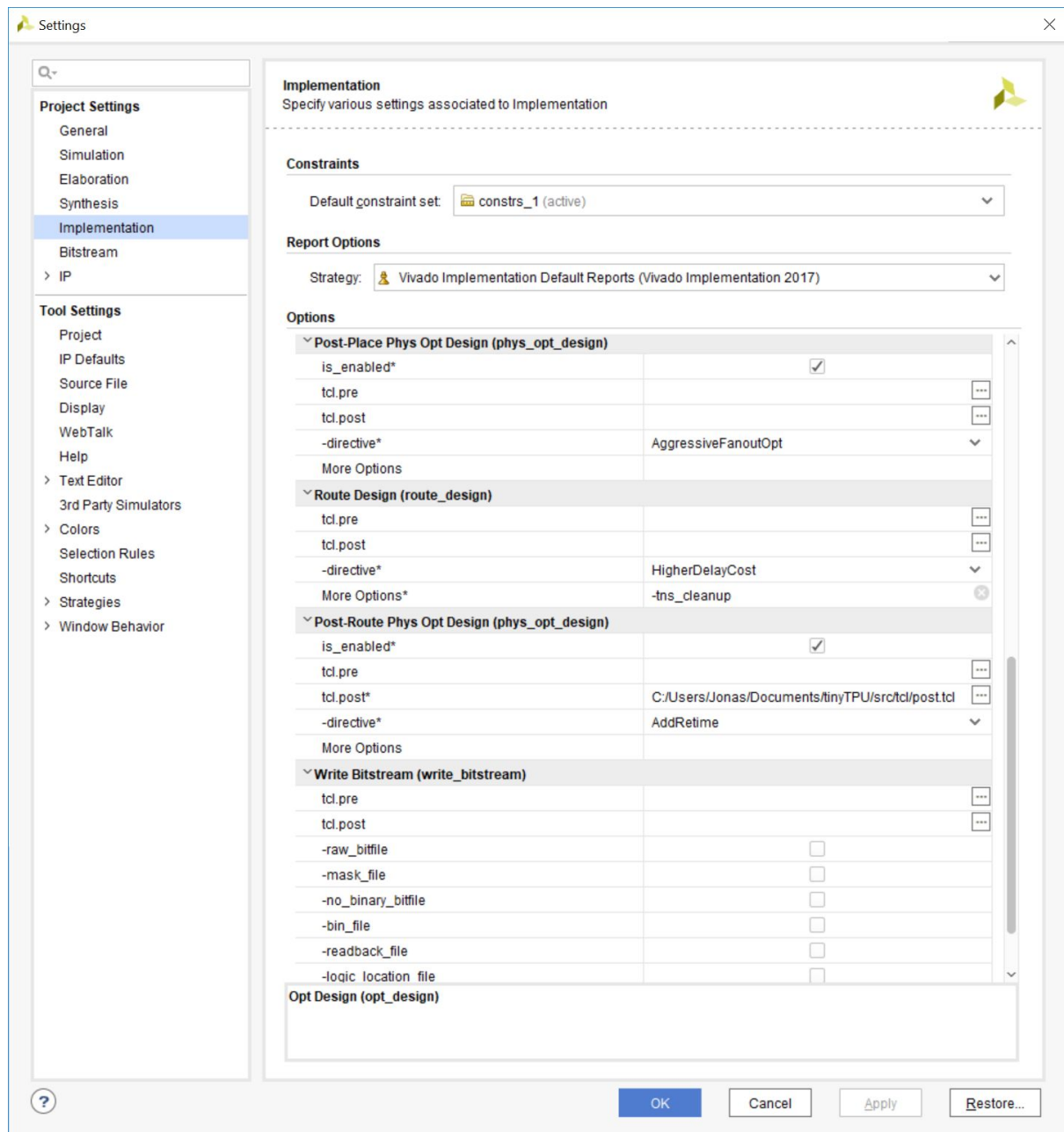


Apply the changes and press *Run Synthesis*.

Implement the Design

Open *Settings* and click on *Synthesis*. Change the settings shown below. This will help to reach the frequency goal.



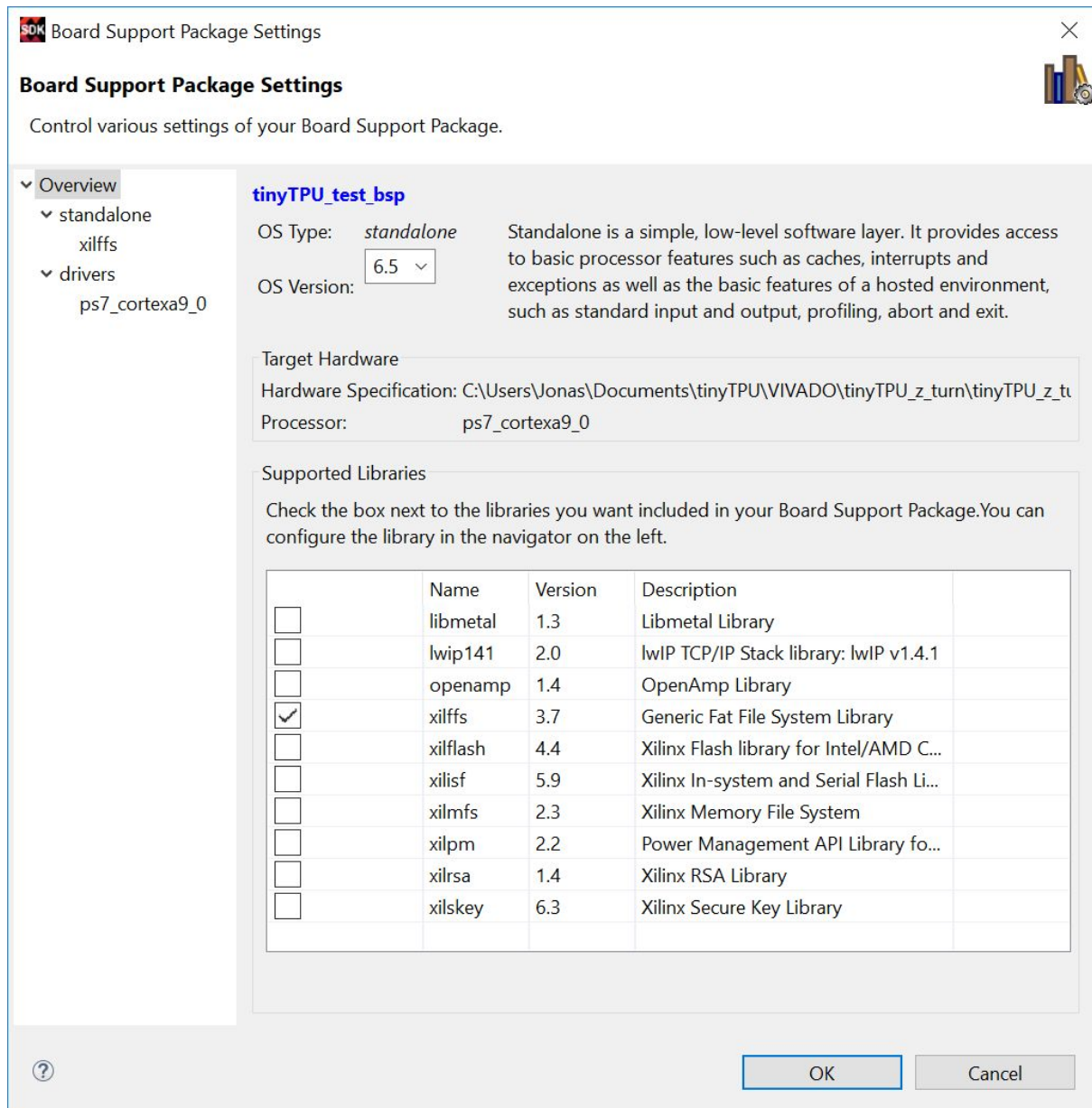


A tcl post script, which executes an added run of *Post-Route Phys Opt Design* can sometimes help reaching the goal and would just look like something like this:
`phys_opt_design -directive AddRetime`

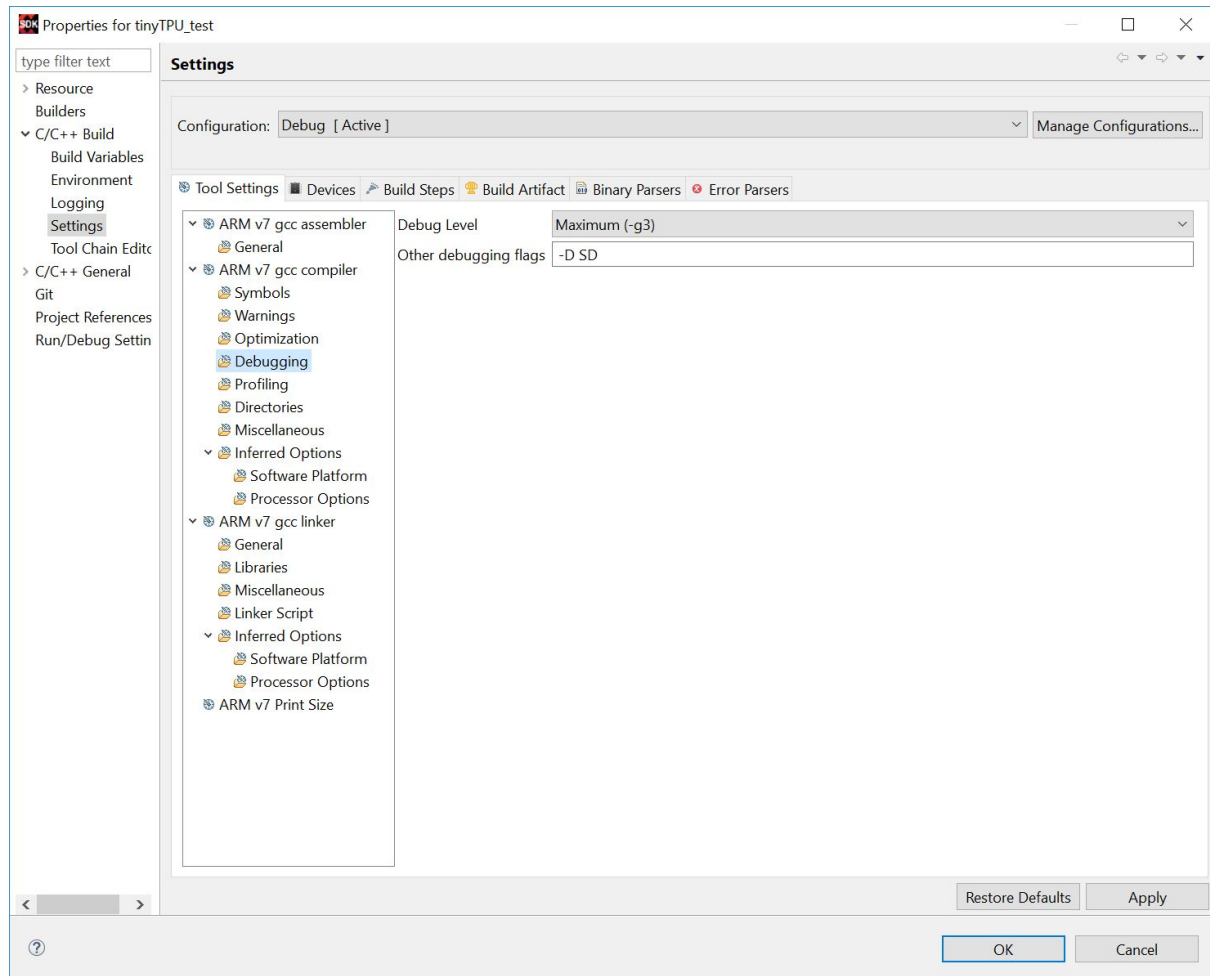
Apply the changes and press *Run Implementation*. Press *Generate Bitstream* after the implementation is finished.

Add Sample Project in Xilinx SDK

Got to *File* → *Export Hardware* and check *Include Bitstream*. Press OK. Now press *File* → *Launch SDK* and the Xilinx SDK should start. Create an empty application project and a *Board Support Package* (BSP) should be generated. Add all C sources from *src/C*. Right click on the BSP and select *Board Support Package Settings* and check the *xilffs* library.



Now right click on the project and choose *C/C++ Build Settings*. Add the shown line to Debugging:



This will enable the SD-Card example program. If you want to transfer data over serial instead of using the SD-Card, you should type *RPC* instead of *SD*. You can now build the project.

Create FSBL and Boot Image

Create a new application project with *First Stage Bootloader (Zynq FSBL)* template. Right click on the main project and select *Create Boot Image*. The FSBL and Bitstream should be selected automatically.

Create Boot Image

Creates Zynq Boot Image in .bin format from given FSBL elf and partition files in specified output folder.

Architecture: Zynq

☐ Create new BIF file ☒ Import from existing BIF file

Import BIF file path: C:\Users\Jonas\Documents\tinyTPU\VIVADO\tinyTPU_z_turn\tinyTPU_z_turn.sdk\tinyTPU_test\bootimage\tinyTPU_test.bif Browse...

Basic Security

Output BIF file path: C:\Users\Jonas\Documents\tinyTPU\VIVADO\tinyTPU_z_turn\tinyTPU_z_turn.sdk\tinyTPU_test\bootimage\tinyTPU_test.bif Browse...

UDF data: Browse...

☐ Split Output format: BIN

Output path: C:\Users\Jonas\Documents\tinyTPU\VIVADO\tinyTPU_z_turn\tinyTPU_z_turn.sdk\tinyTPU_test\bootimage\BOOT.bin Browse...

Boot image partitions

File path	Encrypt...	Authent...
(bootloader) C:\Users\Jonas\Documents\...	none	none
C:\Users\Jonas\Documents\tinyTPU\VIVA...	none	none
C:\Users\Jonas\Documents\tinyTPU\VIVA...	none	none

Add Delete Edit Up Down

? Preview BIF Changes Create Image Cancel

Click *Create Image* and copy the generated file BOOT.bin to the SD-Card. The SD-Card can now be used to load the generated hardware and software on the Zynq SoC.

Load and execute the Sample Model

Definition of a sample model can be found in *src/python*.

mnist_model.py will train a model with the MNIST dataset. This model is trained without bias and the weights are limited to the range from -1 to 127/128.

quantize_weights.py modelname.h5 will quantize the weights of a given model.

export_weights.py modelname.h5 will export the weights of each layer of a given model, which will be stored as csv files.

export_inputs.py inputname.csv will acquire the MNIST test input data and export it to a given csv file.

transfer_input.py inputname.csv index matrix_width takes the given csv file, takes *matrix_width* input vectors at position *index* and exports these as splitted up matrices with the width *matrix_width* to a formatted txt file. This file can be loaded from the SD-Card from the sample program.

transfer_weights.py matrix_width takes the exported weights and splits them up into *matrix_width* by *matrix_width* matrices, which get exported to a formatted txt file for the SD-Card sample program.

transfer_instructions matrix_width will create a formatted txt file with instructions with the help of *matrix_width* and the known exported model in csv format.

The *matrix_width* is always the width of the matrix multiply unit which is 14, if not modified. The formatted txt files can be stored on the SD-Card for execution. The sample program expects a serial input, naming the file to execute (e.g. weights.txt, which loads weights to the weight buffer). Instructions are buffered to transmit them fast. After the execution of the instructions, the execution time, measured in hardware, will be printed and also stored in a file called *TIMINGS.CSV* and the results are ready to be read from the TPU.

To acquire all the calculations from the TPU, a txt file can be created, which instructs the program to read from the TPU's unified buffer and to store the results in a file called *RESULTS.CSV*. This txt file should look like this:

```
results:[  
[address,length,append]  
]
```

where address is the address of the unified buffer that should be read, length is the number of vectors which should be read and append indicates, if *RESULTS.CSV* should be appended (1) or overwritten (0) with the results.

Info: A newline character should be placed at the end of the txt file. Filenames shouldn't be too long, because the filesystem driver doesn't support long filenames.