UNIVERSITÀ DEGLI STUDI DI SASSARI

PS-PL COMMUNICATION MANAGEMENT ON A HETEROGENEOUS SYSTEM OS

Author
RAFFAELE MELONI

25 March 2022

Contents

1 Direct Memory Access

The communication management among the PS and PL sides is one of the most important functionalities needed by a heterogeneous system OS [1], in order to develop software applications which exploit the hardware acceleration of FPGAs. The **Direct Memory Access** (DMA) is a method for accessing the main memory (DDR) without tying up the CPU. Therefore, it leaves the CPU available to perform other operations during the read/write cycle. For this reason it has been used as the method of communication between Processing System (PS) and Programmable Logic (PL).

This document presents[1] how to develop a heterogeneous application, divided in two main part, whose communication has been implemented using AXI DMA [3] and AXI-Stream interface modules [4]:

- **Hardware application**, running on the FPGA, made up by a custom hardware accelerator used to speed up the most onerous operations
- **Software application**, running on the CPU, delegated for control operations, like communication management, and the least onerous operation

Figure 1 shows the main structure. Multiple DMAs and FIFOs have been used, one for each data sent from CPU to FPGA (MM2S DMA) and one per data sent from FPGA to CPU (S2MM DMA) in order to separate input/output workflows and generalize the implementations.

1.1 Communication protocol

The software application is a C-based application in the Linux Userspace. It accesses the DDR, through `/dev/mem` file, using `mmap()`[2], it handles the DMA control addresses, and it write data (MM2S) in the source addresses, wait the hardware application and, once data comes back, it reads data (S2MM) from destination addresses.

On hardware side, the DMAs access the DDR directly through their Slave AXI Lite interface (s_axi_lite). The MM2S DMA reads data from DDR and sends them to the accelerator through the input AXI FIFOs Data Stream, at this point data reaches the accelerator. Processed data are sent back to the output AXI FIFOs Data Stream, and then to S2MM DMAs which will write data in the DDR.

In order to evaluate the communication, a hardware accelerator which implements the Advanced Encryption Standard with 256 bit key and 128 bit text (AES256) has been used.

[1]This work is partially derived by Introduction to using axi dma in embedded linux guide [2] and it has been adapted to use the DMA by Yocto linux OS with a custom accelerator

[2]Linux man page mmap: https://man7.org/linux/man-pages/man2/mmap.2.html

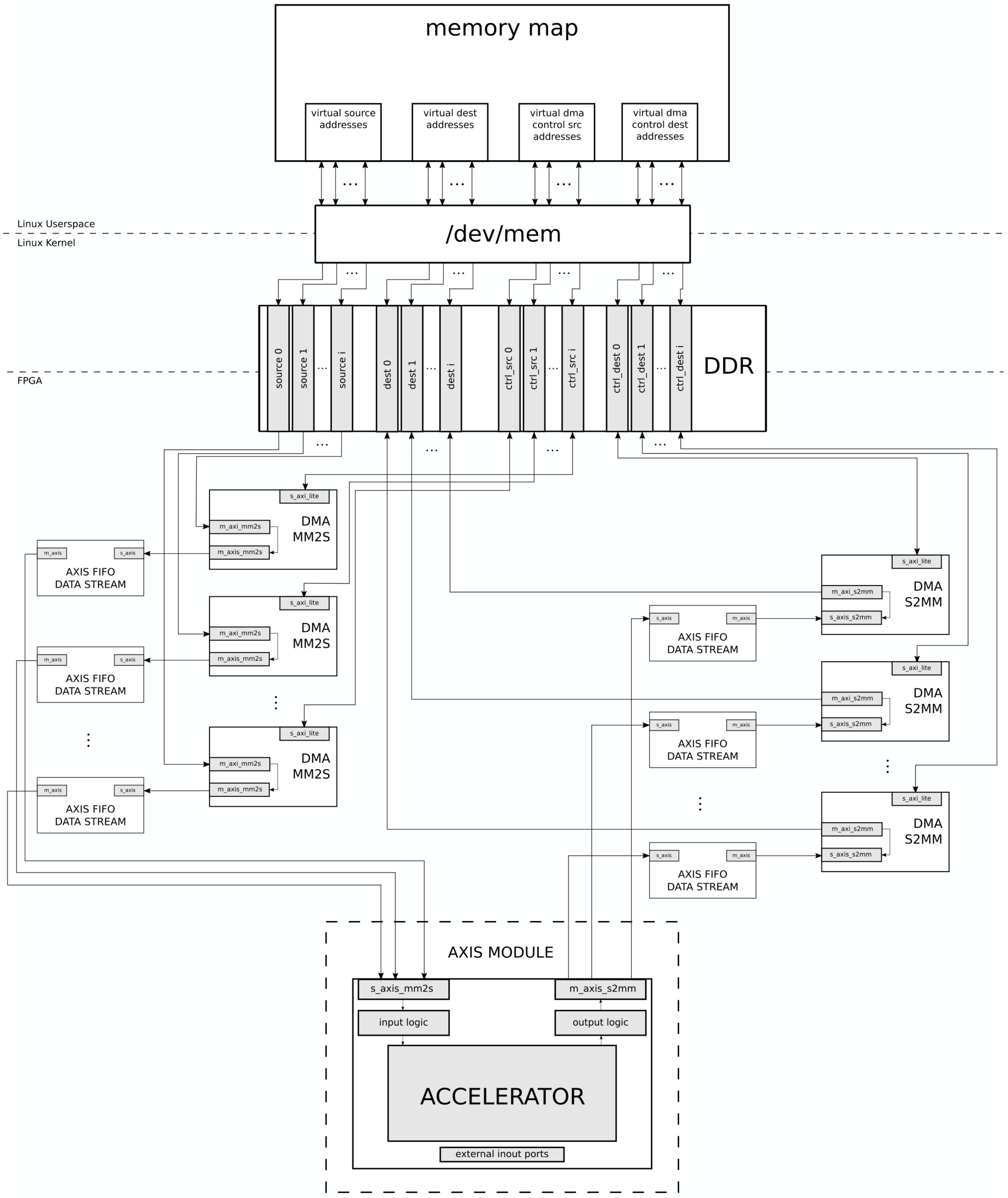


Figure 1: Heterogeneous application based on multiple DMA

2 Hardware Application

The final hardware application is a *.bin file, generated by Vivado IP integrator [5] and [6], which can be loaded by the software application, Section 3. The ZynqMP processing system DMA channels built-in allow only memory-to-memory transfers, not stream-to-memory or memory-to-stream transfers. Custom IPs like accelerators, peripherals, and any other hardware block, which are "stream oriented", needs a specific interface: the AXI DMA IP. It allows any block with the AXI Stream (AXIS) interface to access the DDR for receiving and sending data – in other words, it allows the communication between PS and PL. Thus, the hardware application, Figure 2, consists of four main blocks:

- AXI DMA IP blocks
- AXI4-Stream Data input FIFOs[3]
- AXI4-Stream Data output FIFOs
- Custom IP block with AXIS interface

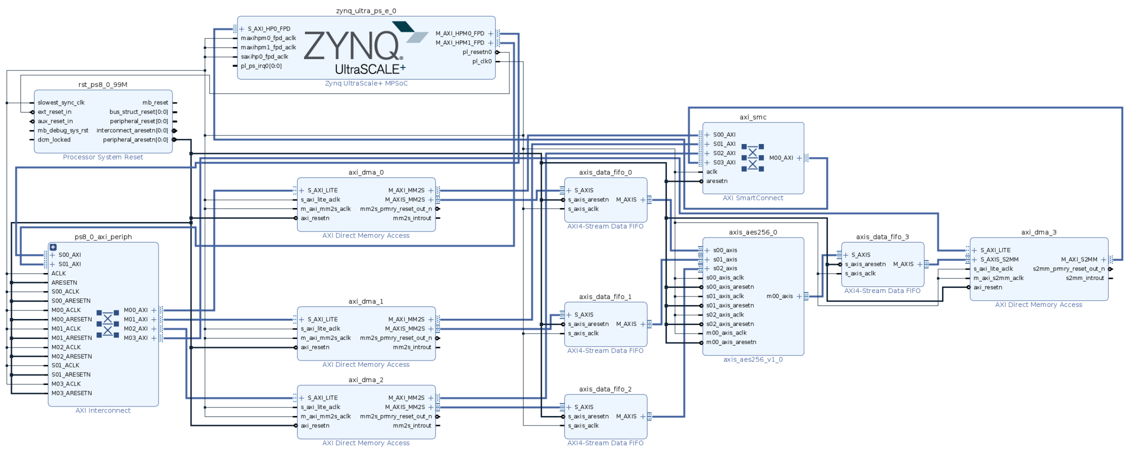


Figure 2: IP block design aes256_dma

The steps needed to create such application are:

1. Create a custom IP block
2. Connect DMA, custom IP and Processing System blocks
3. Edit XDC (external ports only)
4. Generate bitstream

2.1 Create a custom IP block

The first step is to create the custom IP block with an AXIS interface, starting from a Verilog module.

[3]FIFOs are not strictly necessary, but they allow to speed up and to make easier the communication. In fact, they receive data from DMA, make them available to logic with a known interface, and send processed data back to DMA

```
// Top module template

module axis_top_hw #(parameter C_AXIS_TDATA_WIDTH = 32)
(
    // User inputs

    // User outputs

    /*
     * AXIS slave interface (input data)
     */
    input  wire                      s00_axis_aclk,
    input  wire                      s00_axis_aresetn,
    input  wire [C_AXIS_TDATA_WIDTH-1:0]  s00_axis_tdata, // input data
    input  wire                      s00_axis_tvalid,         // input data valid
    output wire                      s00_axis_tready,         // slave ready
    // input  wire                   s00_axis_tlast,          // not used
    /*
     * Other AXIS slaves
     */
    //input  wire                      s0i_axis_aclk,
    //input  wire                      s0i_axis_aresetn,
    //input  wire [C_AXIS_TDATA_WIDTH-1:0]  s0i_axis_tdata, // input data
    //input  wire                      s0i_axis_tvalid,         // input data valid
    //output wire                      s0i_axis_tready,         // slave ready

    /*
     * AXIS master interface (output data)
     */
    input  wire                      m00_axis_aclk,
    input  wire                      m00_axis_aresetn,
    output wire [C_AXIS_TDATA_WIDTH-1:0]  m00_axis_tdata, // output data
    output wire                      m00_axis_tvalid,         // output data valid
    input  wire                      m00_axis_tready,         // output ready
    output wire                      m00_axis_tlast           // data last signal
    /*
    * Other AXIS masters
    */
    //input  wire                      m0i_axis_aclk,
    //input  wire                      m0i_axis_aresetn,
    //output wire [C_AXIS_TDATA_WIDTH-1:0]  m0i_axis_tdata, // output data
    //output wire                      m0i_axis_tvalid,         // output data valid
    //input  wire                      m0i_axis_tready,         // output ready
    //output wire                      m0i_axis_tlast           // data last signal
);

// External inputs (switches, pushbuttons etc.)


// Input slave logic

// Accelerator

// Output master logic


// External outputs (leds etc.)
endmodule
```

`axis_aes256.v` provides a real example of such top module template. It has 3 8-bit input data (text_data, key_data, and rc_data) and 1 8-bit output data (chiped_text_data). All data comes from CPU and the output is sent to CPU, so, the module interface is compounded by 3 axis slaves and 1 axis master for the output. The input logic consists of 3 input FIFOs and the output logic consists of an output FIFO.

The `m0i_axis_tlast` signal is very important, since it signals to the DMA that `m0i_axis_tdata` is the last one, allowing the DMA to send the interrupt properly. So, it is necessary to rise it when the last data has been sent. Specifically, in `axis_aes256.v`, the tlast signal is handled by a counter and it is raised while the 16th output is sending, Figure 3 shows the waveforms.

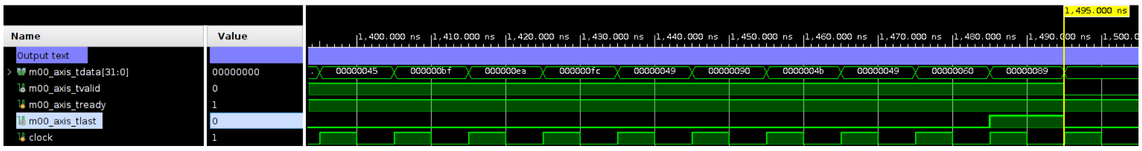
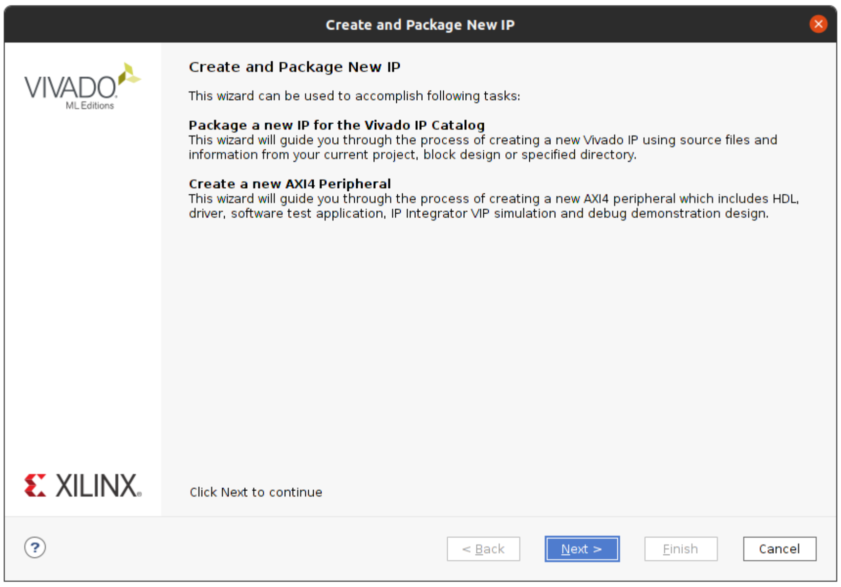


Figure 3: Waveforms - `m0i_axis_tlast`

Once the verilog top module is written, it is necessary to package it into an IP block. To do that from the Vivado project, open 'Tools' and then choose 'Create and Package new IP'.



Create and Package New IP

Create and Package New IP

This wizard can be used to accomplish following tasks:

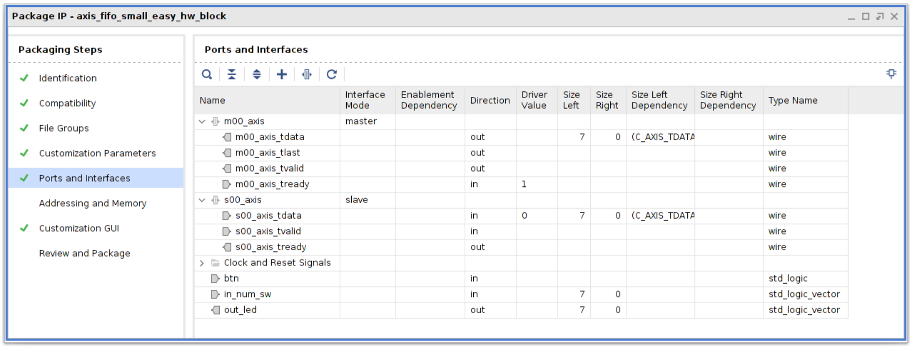
Package a new IP for the Vivado IP Catalog
This wizard will guide you through the process of creating a new Vivado IP using source files and information from your current project, block design or specified directory.

Create a new AXI4 Peripheral
This wizard will guide you through the process of creating a new AXI4 peripheral which includes HDL, driver, software test application, IP Integrator VIP simulation and debug demonstration design.

Click Next to continue

Figure 4: Create and package new IP

Click next and then select 'Package your current project', choose the IP location and click 'Finish', the 'tmp' project will be opened. At this point, make sure the ports and the interfaces are properly connected.



Package IP - axis_fifo_small_easy_hw_block

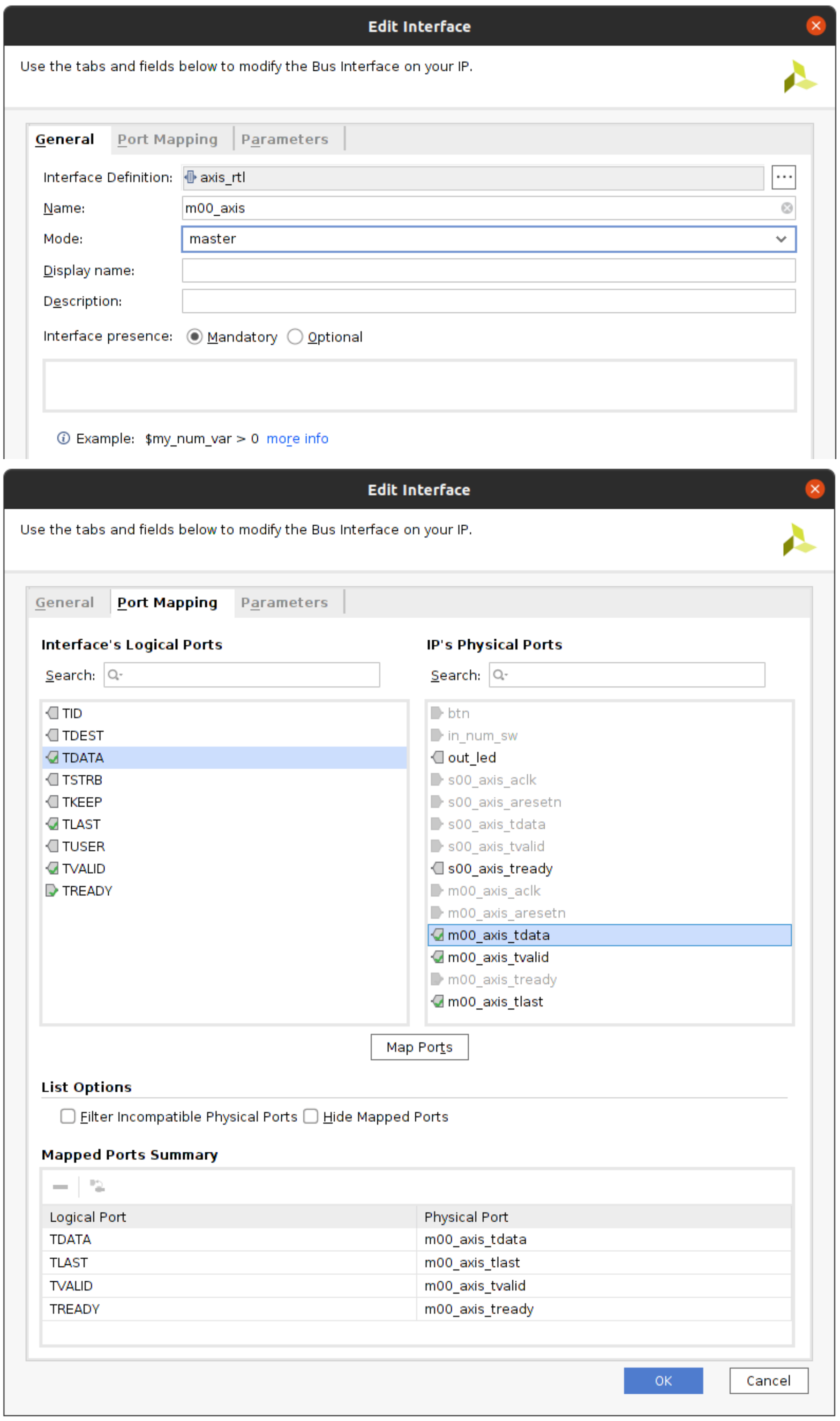
Packaging Steps
- ✓ Identification
- ✓ Compatibility
- ✓ File Groups
- ✓ Customization Parameters
- ✓ Ports and Interfaces
- Addressing and Memory
- ✓ Customization GUI
- Review and Package

Ports and Interfaces

Name	Interface Mode	Enablement Dependency	Direction	Driver Value	Size Left	Size Right	Size Left Dependency	Size Right Dependency	Type Name
m00_axis	master								
m00_axis_tdata			out		7	0	(C_AXIS_TDATA		wire
m00_axis_tlast			out						wire
m00_axis_tvalid			out						wire
m00_axis_tready			in	1					wire
s00_axis	slave								
s00_axis_tdata			in	0	7	0	(C_AXIS_TDATA		wire
s00_axis_tvalid			in						wire
s00_axis_tready			out						wire
Clock and Reset Signals									
btn			in						std_logic
in_num_sw			in		7	0			std_logic_vector
out_led			out		7	0			std_logic_vector

Figure 5: Package IP - Ports and Interfaces

Usually the ports are automatically mapped, but if not, click '+' and add master and slave interface. Choose the interface definition ('axis_rtl', mode 'master' or 'slave'), and map 'TDATA', 'TLAST' (master only), 'TVALID' and 'TREADY'.


Edit Interface

Use the tabs and fields below to modify the Bus Interface on your IP.

General | Port Mapping | Parameters

Interface Definition:	axis_rtl
Name:	m00_axis
Mode:	master
Display name:	
Description:	
Interface presence:	[✓ Mandatory] Optional

Example: $my_num_var > 0 more info

Edit Interface

Use the tabs and fields below to modify the Bus Interface on your IP.

General | Port Mapping | Parameters

Interface's Logical Ports	IP's Physical Ports
TID	btn
TDEST	in_num_sw
TDATA	out_led
TSTRB	s00_axis_aclk
TKEEP	s00_axis_aresetn
TLAST	s00_axis_tdata
TUSER	s00_axis_tvalid
TVALID	s00_axis_tready
TREADY	m00_axis_aclk
	m00_axis_aresetn
	m00_axis_tdata
	m00_axis_tvalid
	m00_axis_tready
	m00_axis_tlast

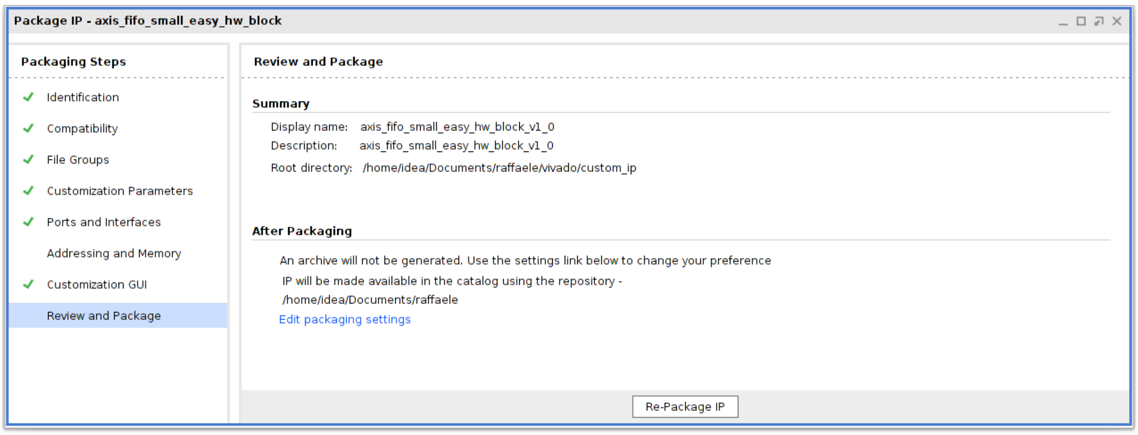
List Options: Filter Incompatible Physical Ports; Hide Mapped Ports

Mapped Ports Summary

Logical Port	Physical Port
TDATA	m00_axis_tdata
TLAST	m00_axis_tlast
TVALID	m00_axis_tvalid
TREADY	m00_axis_tready

Figure 6: Package IP - Choose interface definition and map ports

Finally, package the IP.



Package IP - axis_fifo_small_easy_hw_block

Packaging Steps

- ✓ Identification
- ✓ Compatibility
- ✓ File Groups
- ✓ Customization Parameters
- ✓ Ports and Interfaces
- Addressing and Memory
- ✓ Customization GUI
- Review and Package

Review and Package

Summary

Display name: axis_fifo_small_easy_hw_block_v1_0
Description: axis_fifo_small_easy_hw_block_v1_0
Root directory: /home/idea/Documents/raffaele/vivado/custom_ip

After Packaging

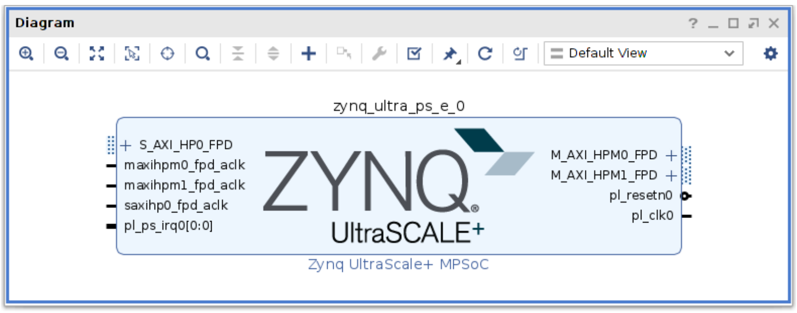
An archive will not be generated. Use the settings link below to change your preference
IP will be made available in the catalog using the repository - /home/idea/Documents/raffaele
Edit packaging settings

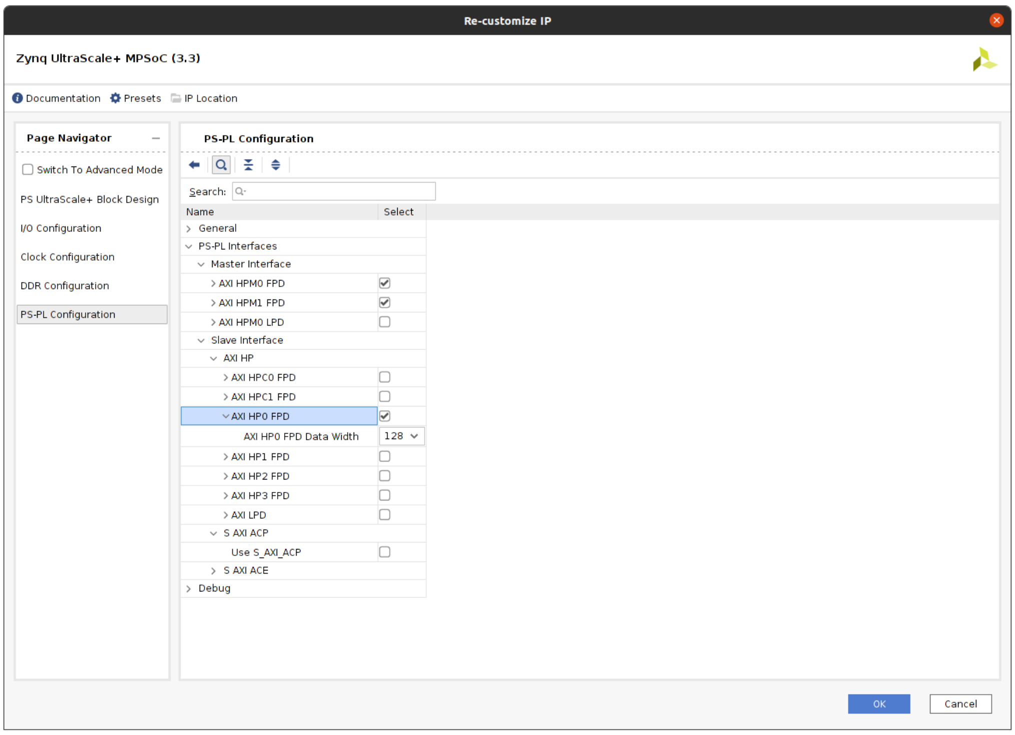
Figure 7: Package IP - Review and package

2.2 Connect DMA, custom IP and Processing System blocks

Once the custom IP has been created and added to the repository IP Catalog, open a new project, create a new Block Design and connect all components.

Add **Zynq UltraScale+ MPSoc** IP block for the PS side, click run block automation (Apply Board Preset) and edit 'PS-PL Configuration' checking 'AXI HP0 FPD'.




Re-customize IP

Zynq UltraScale+ MPSoC (3.3)

Documentation Presets IP Location

Page Navigator
- ☐ Switch To Advanced Mode
- PS UltraScale+ Block Design
- I/O Configuration
- Clock Configuration
- DDR Configuration
- PS-PL Configuration

PS-PL Configuration

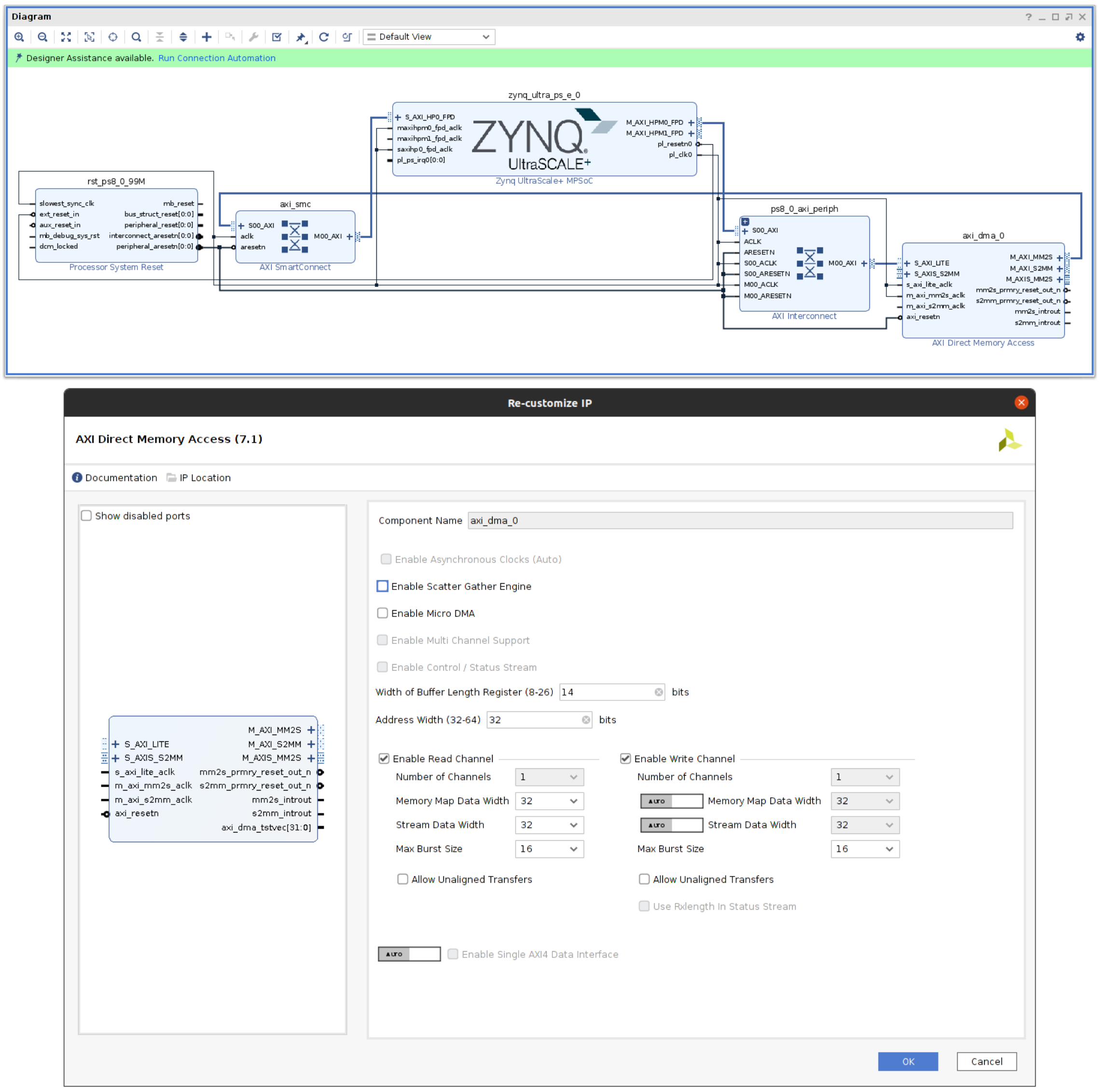
Search:

Name	Select
General	
PS-PL Interfaces	
Master Interface	
AXI HPM0 FPD	✓
AXI HPM1 FPD	✓
AXI HPM0 LPD	☐
Slave Interface	
AXI HP	
AXI HPC0 FPD	☐
AXI HPC1 FPD	☐
AXI HP0 FPD	✓
AXI HP0 FPD Data Width	128
AXI HP1 FPD	☐
AXI HP2 FPD	☐
AXI HP3 FPD	☐
AXI LPD	☐
S AXI ACP	
Use S_AXI_ACP	☐
S AXI ACE	
Debug	

OK Cancel

Figure 8: Add Zynq UltraScale+ MPSoc IP block

Add the **AXI Direct Memory Access** IP block, disable 'Enable Scatter Gather Engine' (leave the remaining options as default) and click run block automation again (check 'All Automation'). The Figure 9 shows a system with only one DMA, thus, because `axis_aes256.v` has three axis slaves and one axis master, add three "input MM2S DMA" ('Enable write channel' disabled), one for each axis slave and a "output S2MM DMA" ('Enable read channel' disabled) for axis master.



AXI Direct Memory Access (7.1)

Component Name: axi_dma_0

- [ ] Enable Asynchronous Clocks (Auto)
- [ ] Enable Scatter Gather Engine
- [ ] Enable Micro DMA
- [ ] Enable Multi Channel Support
- [ ] Enable Control / Status Stream

Width of Buffer Length Register (8-26): 14 bits

Address Width (32-64): 32 bits

[✓] Enable Read Channel	[✓] Enable Write Channel
Number of Channels: 1	Number of Channels: 1
Memory Map Data Width: 32	AUTO Memory Map Data Width: 32
Stream Data Width: 32	AUTO Stream Data Width: 32
Max Burst Size: 16	Max Burst Size: 16
[ ] Allow Unaligned Transfers	[ ] Allow Unaligned Transfers
	[ ] Use Rxlength In Status Stream

AUTO [ ] Enable Single AXI4 Data Interface

Figure 9: Add AXI Direct Memory Access IP block

Add the input and output **AXI4-Stream Data FIFOs** (one for each slave/-master interface respectively), and the custom IP with AXIS interface.

- Connect the Master interface of DMA to Slave interface of input FIFO (M_AXIS_MM2S and S_AXIS).
- Connect the Master interface of input FIFO to Slave interface of custom IP (M_AXIS and s00_axis).
- Connect the Master interface of custom IP to Slave interface of output FIFO (m00_axis and S_AXIS).
- Connect the Master interface of output FIFO to Slave interface of DMA (M_AXIS and S_AXIS_S2MM).
- Run connection automation.

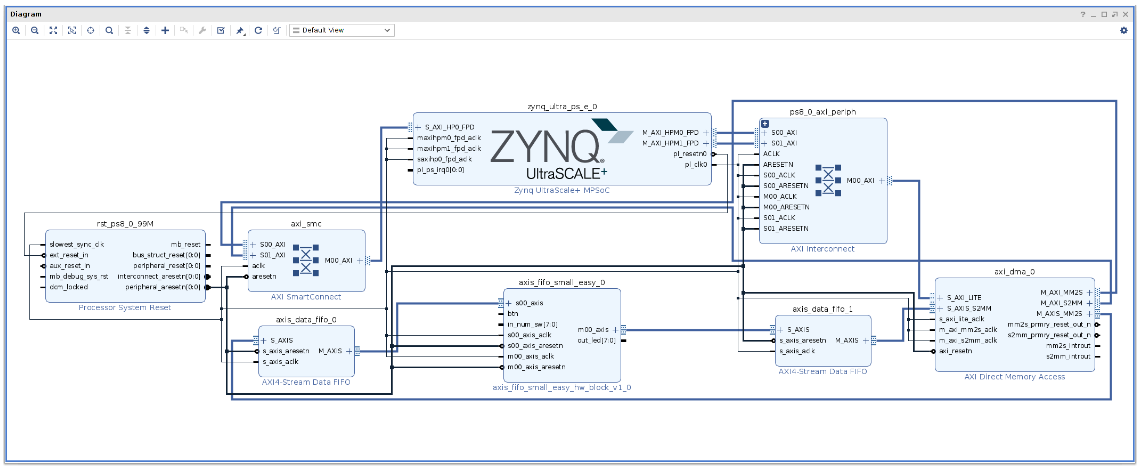


Figure 10: Add AXI4-Stream Data FIFOs and custom IP

If the custom IP has external input/output ports, right click and then 'Make External'.

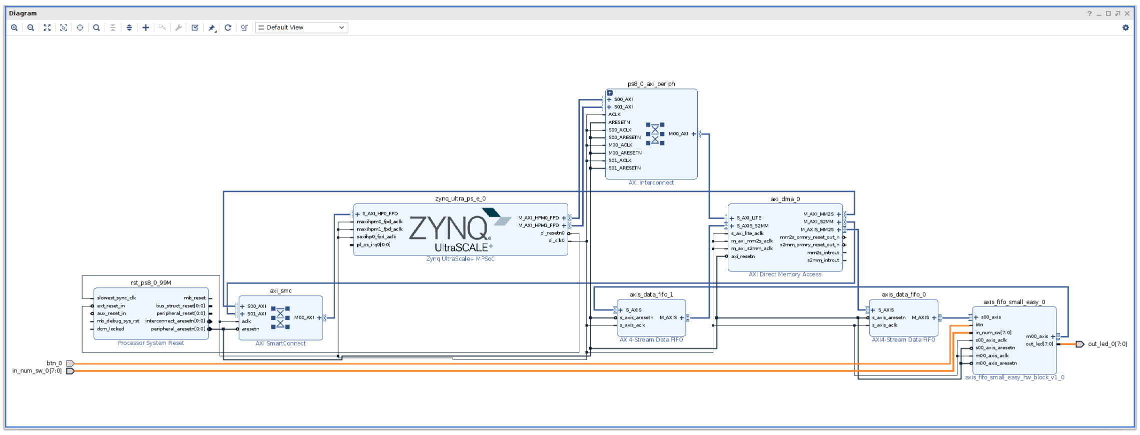


Figure 11: Connect external inputs

Once all components have been connected and the block design has been validated, save it and go to sources, right click on 'design_file_name', and choose 'Create HDL wrapper' (Let Vivado manage wrapper and auto-update) – it will create a *.v version of the Block Design. If you edit the block design, re-run 'Validate block design' before creating the wrapper.

2.3 Generate Bitstream

If the custom IP has external ports, download the XDC from the official site and connect **only** the external ports.

Run synthesis, implementation and generate bitstream[4]. Make sure that neither critical warnings nor errors appear.

[4]Make sure that in project settings `bin_file` generation has been selected

3 Software Application

In order to use the bitstream by the Linux Userspace, as shown in Section 3, please see Yocto FPGA programming [1]. The core is a C application which loads the accelerator and uses the memory map engine to control the DMA for communication, AXI DMA v7.1 - AXI DMA Register Address Map [3]. It manages the communication writing and reading the DMA control registers of the DDR. Source code: `dma_sample_app.c`.

1. Load the accelerator using fpgautil.

   ```
   system("fpgautil -b aes256_dma.bin");
   ```

2. Open the ddr memory.

   ```
   int ddr_memory = open("/dev/mem", O_RDWR | O_SYNC);
   ```

3. Use `mmap()` for mapping the DMA control addresses and data addresses.
4. Write data in source data virtual addresses.
5. Reset, halt the DMAs, and enable all interrupts.
6. Write the source and destination addresses.
7. Run the MM2S and S2MM channels, and the transfer length.
8. Wait for MM2S and S2MM synchronizations.
9. Unmap virtual addresses and close the ddr memory.

`dma_sample_app.c` is a first prototype to test the communication via mmap and DMA, I have planned to implement API to simplify it further.

References

[1] How to create a yocto os on us+ zcu102. https://mdc-suite.github.io/miscellaneous/yoctofpga.

[2] Introduction to using axi dma in embedded linux. https://www.hackster.io/whitney-knitter/introduction-to-using-axi-dma-in-embedded-linux-5264ec. Last visited on 03/03/2022.

[3] Axi dma v7.1. https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf. Last visited on 03/03/2022.

[4] Axi reference guide. https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf. Last visited on 03/03/2022.

[5] Vivado 2021.2 - using ip integrator. https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0009-vivado-using-ip-integrator-hub.html. Last visited on 03/03/2022.

[6] Vivado design suite user guide - designing ip subsystems using ip integrator. https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2021_2/ug994-vivado-ip-subsystems.pdf. Last visited on 03/03/2022.