Custom Peripheral for the AXI4-Full Interface

Objectives

- Create a Hardware/Software system using the ZYBO Board or the ZYBO Z7-10 Board
- Create custom AXI4-Full peripherals in VHDL and integrate them into a Block Based Design in Vivado 2019.1.
- Create a software application in SDK that can transfer data from/to custom peripheral.

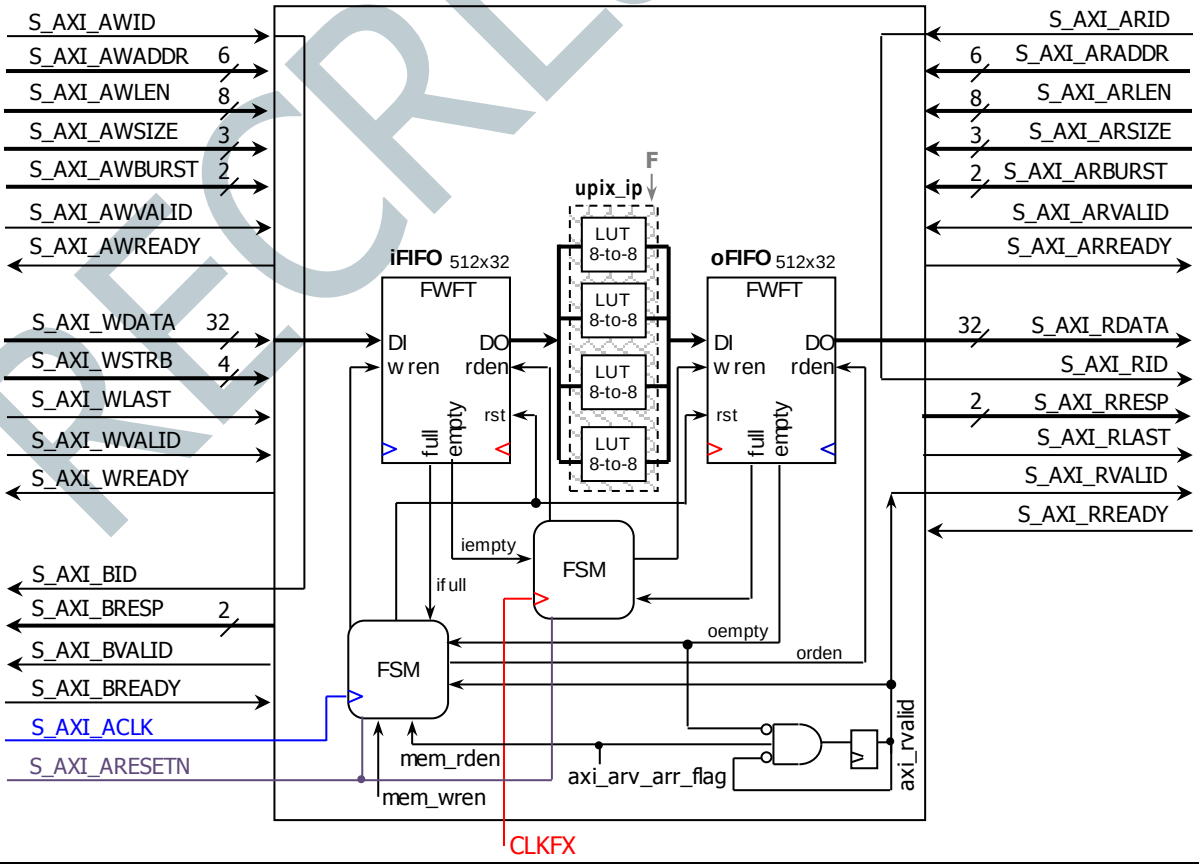
ZYBO/ZYBO Z7-10 Board Setup for Hardware/Software Co-Design

- It is assumed that the definition files (available in `vivado-boards-mastes.zip`) have been installed in Vivado.
- ZYBO: PS_CLK input: 50 MHz. PL_CLK input: 125 MHz. By default, a 100 MHz is generated for the PL fabric.
- ZYBO Z7-10: PS_CLK input: 33.33 MHz. PL_CLK input: 125 MHz. By default, a 50 MHz clock is generated for the PL fabric.

Pixel Processor: Custom Peripheral for AXI4-Full Interface

CONSIDERATIONS

- We will use the Pixel Processor with NC=4, NI=NO=8, F=1.
- AXI4-Full Peripheral: the design allows for 3 cases:
  - ✓ MEMO: Xilinx® example just to test a simple 16-word (32 bits) memory architecture (no pixel processor).
  - ✓ PIXO: Modified "MEMO". Between input data and the 16-word memory, we inserted the pixel processor and a register.
  - ✓ **FIFO**: Custom FIFO-based interface that we will be using. All writes/reads to any 16-word memory position is treated the same (writing/reading on the FIFO).
- List of files (hardware description) to use:
  - ✓ `mypixfull_v1_0.vhd`: AXI4-Full Peripheral (top file, Vivado template).
  - ✓ `mypixfull_v1_0_S00_AXI.vhd`: AXI4-Full Interface description (edited Vivado template, select the "**FIFO**" option)
  - ✓ `myAXI_IP.vhd`, `my_AXI_fifo.vhd`, `my_AXImem.vhd`, `my_genpulse_sclr.vhd`: Ancillary files for the AXI4-Full Peripheral.
  - ✓ `static_ip.vhd`: Pixel Processor IP with glue logic to the FIFOs. Pixel Processor parameters (`F(1..5)`) are specified here.
  - ✓ `LUT_group.vhd`: Top file of the Pixel Processor IP
  - ✓ `LUT_NItoNO.vhd, LUTNIto1.vhd, pack_xtras.vhd`: Other files that make up the Pixel Processor
  - ✓ `LUT_values8to8.txt`: LUT values in a text file.
  - ✓ `tb_mypixAXI4Full.vhd`: Testbench for AXI4-Full peripheral. This is very useful as it emulates the AXI signals resulting from the execution of the software in the PS. This allows to debug the peripheral before it is integrated in the PSoC.
- Pixel Processor AXI4-Full Peripheral (FIFO mode) with AXI signals (we make `S_AXI_CLK=CLK_FX`).



RECRLAB@OU

IP GENERATION

- Create new project in Vivado: `myaxifullpix`
  - ✓ Make sure the default language is VHDL, so that the system wrapper and the template files are created in VHDL
  - ✓ At Default Part, go to Boards, and select the **Zybo** (or **Zybo Z7-10**) Board.
- From the menu bar, select **Tools → Create and Package New IP**.
  - ✓ Create a new AXI4 Peripheral. Name: `mypixfull`. Location `/ip_repo`.
    Peripheral Repositories tip: To add a previously-generated IP into a new project, go to: Project Settings → IP → IP repositories and point to the associated repository folder.
  - ✓ Add Interface: Full, 32 bits. Interface Mode: Slave. Memory Size: 64 bytes (16 32-bit words)
  - ✓ Select Edit IP. A New project appears, open it and look for the `<peripheral name>_S00_AXI.vhd` file (in this case it will be `mypixfull_v1_0_S00_AXI.vhd`). Modify the project:
    - Replace `mypixfull_v1_0_s00_AXI.vhd` with our edited file `mypixfull_v1_0_S00_AXI.vhd`. Same for the file `mypixfull_v1_0.vhd` (this is not strictly necessary).
    - Add the extra files to the folder `/hdl` in `/ip_repo/mypixfull_1.0` and add these source files (including the .txt file) to the Vivado project. * Vivado 2019.1: by default, the files will be added to the folder `/src`.
  - ✓ There is no need to add ports as our peripheral does not include external I/Os.
  - ✓ Synthesize your circuit (just to double-check everything is ok): You should've simulated the code in a different project.
  - ✓ Go to Package IP – mypixfull → File Groups (Merge changes). Then Review and Package → Re-Package IP.
- Your custom IP is now ready to be used as an AXI4-Full Peripheral.
- You will return to the original Vivado Project.

CREATING A BLOCK DESIGN PROJECT IN VIVADO

- Click on Create Block Design and instantiate the Zynq PS and the AXI MYPIXFULL peripheral.
- Click on Run Block Automation and Run Connection Automation. Then 'Validate Design'
- There is no need to add an `.xdc` file as our peripheral does not use external ports.
- Create the VHDL wrapper (Sources Window → right click on the top-level system design → Create HDL Wrapper)
- Synthesize, implement, and generate the bitstream.
  - ✓ An error will be reported when Synthesizing. Vivado only copies VHDL files from the IP folder to the embedded project folder (located inside the `/<peripheral name>.srcs/…/ipshared` folder). As a result, the `LUT_NItoNO.vhd` file cannot find the `LUT_values.txt`. We need to place this text file in the same folder as the `LUT_NItoNO.vhd` file.
  - ✓ This folder location is available by opening the `LUT_NItoNO.vhd` file. You need to find this file in the design structure or via the Vivado error which will point to the `LUT_NItoNO.vhd` file. After copying the .txt file, you can Synthesize again.
  - ✓ In general, this procedure is to be followed for any ancillary file (e.g. text file) used by the VHDL files.
- Export hardware (with bitstream) and launch SDK.

SDK

- See Tutorial Unit 2 for details on how to create and test a software application on SDK.
- Go to Xilinx Tools → Repositories, click on 'New' and then browse to the folder `\ip_repo\mypixfull_1.0` and click ok.
- Create a new SDK application: pixtest. Then, copy the following file into the `/src` folder: `test_pixi.c`
- The software routine controls the AXI4-Full Pixel Processor peripheral by writing and reading from memory positions. Note that the instructions to write on the AXI4-Full peripheral are different than in the case of AXI4-Lite.
- **Testing strategy**: write four 32-bit words and read four 32-bit words. Each output word corresponds to an input word.
  - ✓ `MYPIXFULL_mWriteMemory(Base Address, 32-bit word)`: Use this function (created when generating the AXI-4 Full IP) to write a 32-bit word onto the AXI4-Full Peripheral.
  - ✓ `MYPIXFULL_mReadMemory(Base Address)`: Use this function (created when generating the AXI4-Full IP) to read a 32-bit word from the AXI4-Full Peripheral.
  - ✓ `Base Address`: In the FIFO case, any address in the 64-byte range (the one allowed for the AXI4-Full Pixel Processor peripheral) works since we only write/read to/from the FIFOs.
  - ✓ For example, with the given Pixel Processor Parameters (with parameter F = 1), we are supposed to get:

Input	Output
0xF00DBEEF	0xF83ADDF7
0xBEBEDEAF	0xDDDDEED4
0xFADEDEAD	0xFDEEEED2
0xCAFEC0C0	0xE3FFDEDE

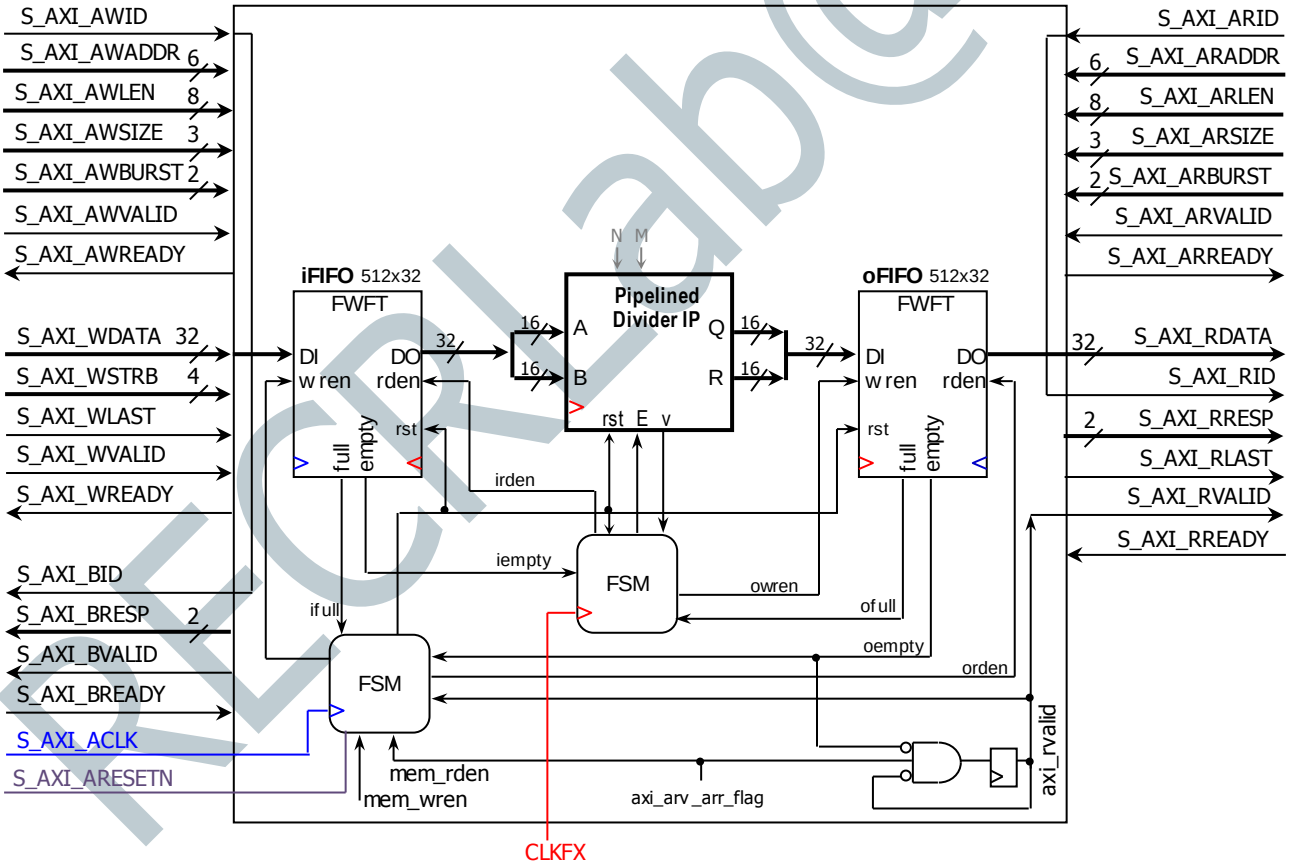
RECRLab@OU

PIPELINED DIVIDER: CUSTOM PERIPHERAL FOR AXI4-FULL INTERFACE

CONSIDERATIONS

- We will use the Pipelined Divider with N=16, M=16.
- AXI4-Full Peripheral: the VHDL files allow only for the case:
  - ✓ **FIFO**: Custom FIFO-based interface that we will be using. All writes/reads to any 16-word memory position is treated the same (writing/reading on the FIFO).

- List of files (hardware design) to use:
  - ✓ `mypipdivfull_v1_0.vhd`: AXI4-Full Peripheral (top file, Vivado template).
  - ✓ `mypipdivfull_v1_0_S00_AXI.vhd`: AXI4-Full Interface description (edited Vivado template)
  - ✓ `myAXI_IP.vhd`, `my_AXI_fifo.vhd`, `my_genpulse_sclr.vhd`: Ancillary files for the AXI4-Full Peripheral.
  - ✓ `pipdiv_ip.vhd`: Pipelined Divider IP with glue logic to the FIFOs. Parameters (N, M) are specified here.
  - ✓ `res_div_pip.vhd`: Top file of the Pipelined Divider IP
  - ✓ `unit_proc.vhd, my_pashiftreg.vhd, fulladd.vhd, dffe.vhd`: Other files that make up the Pipelined Divider
  - ✓ `tb_mypipdivAXI4Full.vhd`: Testbench for AXI4-Full peripheral. This is very useful as it emulates the AXI signals resulting from the execution of the software in the PS. We can then carefully verify the functionality of the peripheral before it is integrated in the PSoC.
- Pipelined Divider AXI4-Full Peripheral (FIFO mode) with AXI signals (we make `S_AXI_CLK=CLK_FX`).
  - ✓ Active-high reset: This is required by the FIFOs.
  - ✓ Note how we also feed the active-high reset to the Pipelined Divider and to the FSM@CLKFX. Though we could have used the active-low AXI bus reset (S_AXI_ARESETN), we preferred the high-level reset as this configuration might be useful if we decide to make the create a run-time alterable region inside this AXI4-Fulll peripheral.



IP GENERATION

- Create new project in Vivado: `myaxifullpipdiv`
  - ✓ Make sure the default language is VHDL, so that the system wrapper and the template files are created in VHDL
  - ✓ At Default Part, go to Boards, and select the **Zybo** (or **Zybo Z7-10**) Board.
- From the menu bar, select **Tools → Create and Package New IP**.
  - ✓ Create a new AXI4 Peripheral. Name: `mypipdivfull`. Location `/ip_repo`.
    Peripheral Repositories tip: To add a previously-generated IP into a new project, go to: Project Settings → IP → IP repositories and point to the associated repository folder.

RECRLab@OU

- ✓ Add Interface: Full, 32 bits. Interface Mode: Slave. Memory Size: 64 bytes (16 32-bit words)
- ✓ Select Edit IP. A New project appears, open it and look for the `<peripheral name>_S00_AXI.vhd` file (in this case it will be `mypipdivfull_v1_0_S00_AXI.vhd`). Modify the project:
  - Replace `mypipdivfull_v1_0_s00_AXI.vhd` with our edited file `mypipdivfull_v1_0_S00_AXI.vhd`. Same for the file `mypipdivfull_v1_0.vhd` (this is not strictly necessary).
  - Add the extra files to the folder `/hdl` in `/ip_repo/mypipdivfull_1.0` and add these source files to the Vivado project. * Vivado 2019.1: by default, the files will be added to the folder `/src`.
- ✓ There is no need to add ports as our peripheral does not include external I/Os.
- ✓ Synthesize your circuit (just to double-check everything is ok): You should've simulated the code in a different project.
- ✓ Go to Package IP – mypipdivfull → File Groups (Merge changes). Then Review and Package → Re-Package IP.

- Your custom IP is now ready to be used as an AXI4-Full Peripheral.
- You will return to the original Vivado Project.

CREATING A BLOCK DESIGN PROJECT IN VIVADO

- Click on Create Block Design and instantiate the Zynq PS and the AXI MYPIPDIVFULL peripheral.
- Click on Run Block Automation and Run Connection Automation. Then 'Validate Design'
- There is no need to add an `.xdc` file as our peripheral does not use external ports.
- Create the VHDL wrapper (Sources Window → right click on the top-level system design → Create HDL Wrapper)
- Synthesize, implement, and generate the bitstream.
- Export hardware (with bitstream) and launch SDK.

SDK

- See Tutorial Unit 2 for details on how to create and test a software application on SDK.
- Go to Xilinx Tools → Repositories, click on 'New' and then browse to the folder `\ip_repo\mypipdivfull_1.0` and click ok.
- Create a new SDK application: pipdivtest. Then, copy the following file into the `/src` folder: `test_pipdiv.c`
- The software routine controls the AXI4-Full Pipelined Divider peripheral by writing and reading from memory positions.
- **Testing strategy**: write four 32-bit words and read four 32-bit words. Each output word corresponds to an input word.
  - ✓ `MYPIPDIVFULL_mWriteMemory(Base Address, 32-bit word)`: Use this function (created when generating the AXI-4 Full IP) to write a 32-bit word onto the AXI4-Full Peripheral.
  - ✓ `MYPIPDIVFULL_mReadMemory(Base Address)`: Use this function (created when generating the AXI4-Full IP) to read a 32-bit word from the AXI4-Full Peripheral.
  - ✓ `Base Address`: In the FIFO case, any address in the 64-byte range (the one allowed for the AXI4-Full Pipelined Divider peripheral) works since we only write/read to/from the FIFOs.
  - ✓ For example, for the given with the given Pipelined Divider parameters (N=M=16), we are expected to get:

Input (A|B)	Output (Q|R)
0x008C0009	0x000F0005
0x00BB000A	0x00120007
0x0FEA0371	0x00040226
0x09640037	0x002B0027

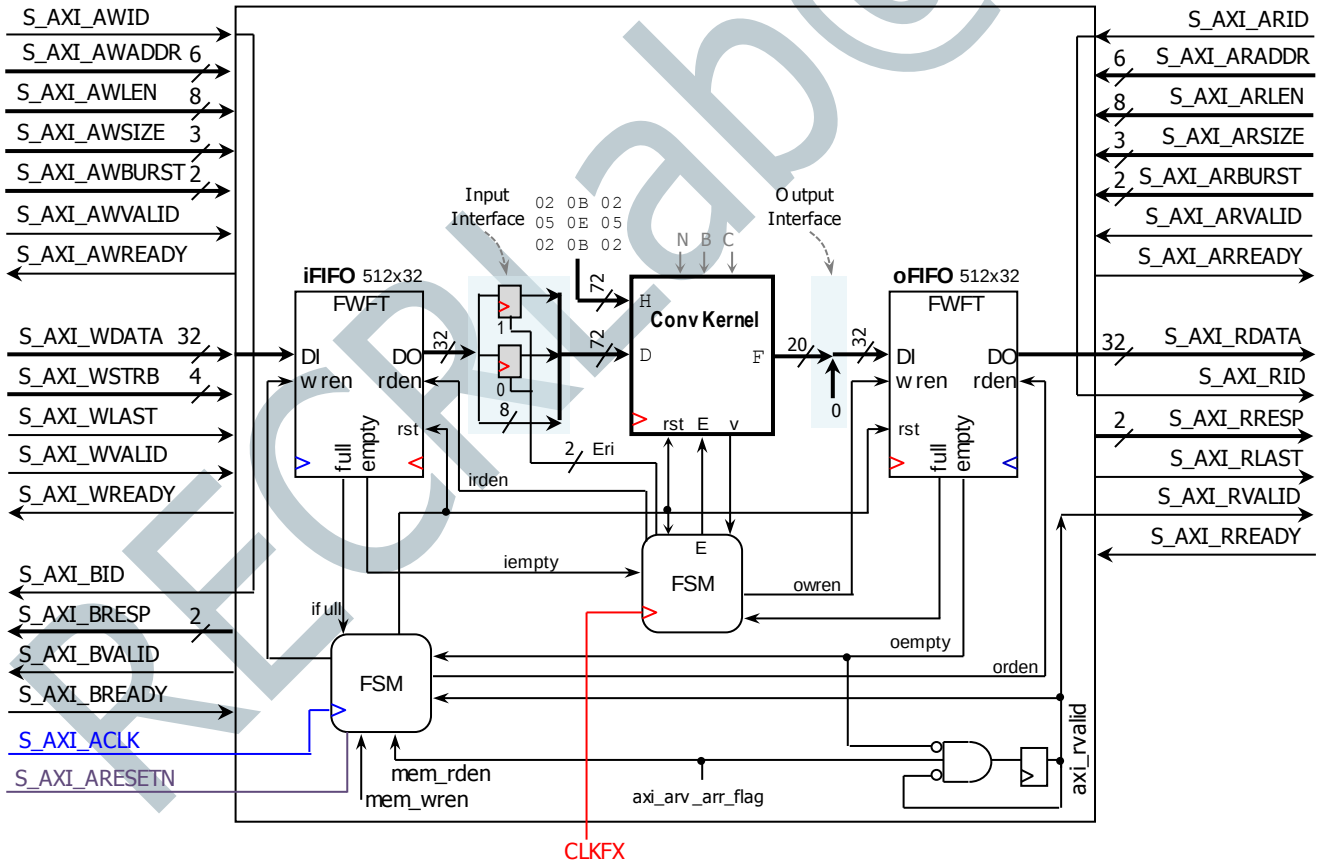
RECRLAB@OU

PIPELINED 2D CONVOLUTION KERNEL: CUSTOM PERIPHERAL FOR AXI4-FULL INTERFACE

CONSIDERATIONS

- We will use the Pipelined 2D Convolution Kernel with N=3, B=C=8.
- AXI4-Full Peripheral: the VHDL files allow only for the case:
  - ✓ **FIFO**: Custom FIFO-based interface that we will be using. All writes/reads to any 16-word memory position is treated the same (writing/reading on the FIFO).

- List of files (hardware design) to use:
  - ✓ `myconv2full_v1_0.vhd`: AXI4-Full Peripheral (top file, Vivado template).
  - ✓ `myconv2full_v1_0_S00_AXI.vhd`: AXI4-Full Interface description (edited Vivado template)
  - ✓ `myAXI_IP.vhd`, `my_AXI_fifo.vhd`, `my_genpulse_sclr.vhd`: Ancillary files for the AXI4-Full Peripheral.
  - ✓ `myconv2_ip.vhd`: Pipelined 2D Conv Kernel IP with glue logic to the FIFOs. Parameters (N, B, C) are specified here.
  - ✓ `myconv2.vhd`: Top file of the Pipelined 2D Convolution Kernel IP
  - ✓ `adder_tree.vhd, pack_xtras.vhd, my_pashiftreg.vhd, my_rege.vhd, my_addsub.vhd, fulladd.vhd, dffe.vhd`: Other files that make up the Pipelined 2D Convolution Kernel IP.
  - ✓ `tb_myconv2AXI4Full.vhd`: Testbench for AXI4-Full peripheral. This is very useful as it emulates the AXI signals resulting from the execution of the software in the PS. We can then carefully verify the functionality of the peripheral before it is integrated in the PSoC.
- Pipelined 2D Convolution Kernel AXI4-Full Peripheral (FIFO mode) with AXI signals (we make `S_AXI_CLK=CLK_FX`).
  - ✓ Active-high reset: This is required by the FIFOs.
  - ✓ Note how we also feed the active-high reset to the Pipelined 2D Conv Kernel and to the FSM@CLKFX. Though we could have used the active-low AXI bus reset (S_AXI_ARESETN), we preferred the high-level reset as this configuration might be useful if we decide to make the create a run-time alterable region inside this AXI4-Fulll peripheral.



IP GENERATION

- Create new project in Vivado: `myaxifullconv2`
  - ✓ Make sure the default language is VHDL, so that the system wrapper and the template files are created in VHDL
  - ✓ At Default Part, go to Boards, and select the **Zybo** (or **Zybo Z7-10**) Board.
- From the menu bar, select **Tools → Create and Package New IP**.
  - ✓ Create a new AXI4 Peripheral. Name: `myconv2full`. Location `/ip_repo`.

Peripheral Repositories tip: To add a previously-generated IP into a new project, go to: Project Settings → IP → IP repositories and point to the associated repository folder.

- ✓ Add Interface: Full, 32 bits. Interface Mode: Slave. Memory Size: 64 bytes (16 32-bit words)
- ✓ Select Edit IP. A New project appears, open it and look for the `<peripheral name>_S00_AXI.vhd` file (in this case it will be `myconv2full_v1_0_S00_AXI.vhd`). Modify the project:
  - Replace `myconv2full_v1_0_s00_AXI.vhd` with our edited file `myconv2full_v1_0_S00_AXI.vhd`. Same for the file `myconv2full_v1_0.vhd` (this is not strictly necessary).
  - Add the extra files to the folder `/hdl` in `/ip_repo/myconv2full_1.0` and add these source files to the Vivado project. * Vivado 2019.1: by default, the files will be added to the folder `/src`.
- ✓ There is no need to add ports as our peripheral does not include external I/Os.
- ✓ Synthesize your circuit (just to double-check everything is ok): You should've simulated the code in a different project.
- ✓ Go to Package IP – myconv2full → File Groups (Merge changes). Then Review and Package → Re-Package IP.

- Your custom IP is now ready to be used as an AXI4-Full Peripheral.
- You will return to the original Vivado Project.

CREATING A BLOCK DESIGN PROJECT IN VIVADO

- Click on Create Block Design and instantiate the Zynq PS and the AXI MYCONV2FULL peripheral.
- Click on Run Block Automation and Run Connection Automation. Then 'Validate Design'
- There is no need to add an `.xdc` file as our peripheral does not use external ports.
- Create the VHDL wrapper (Sources Window → right click on the top-level system design → Create HDL Wrapper)
- Synthesize, implement, and generate the bitstream.
- Export hardware (with bitstream) and launch SDK.

SDK

- See Tutorial Unit 2 for details on how to create and test a software application on SDK.
- Go to Xilinx Tools → Repositories, click on 'New' and then browse to the folder `\ip_repo\myconv2full_1.0` and click ok.
- Create a new SDK application: conv2test. Then, copy the following file into the `/src` folder: `test_conv2.c`
- The software routine controls the AXI4-Full Pip. 2D Conv Kernel peripheral by writing and reading from memory positions.
- **Testing strategy**: write three 32-bit words and read one 32-bit words. Each output word corresponds to 3 input words.
  - ✓ `MYCONV2FULL_mWriteMemory(Base Address, 32-bit word)`: Use this function (created when generating the AXI-4 Full IP) to write a 32-bit word onto the AXI4-Full Peripheral.
  - ✓ `MYCONV2FULL_mReadMemory(Base Address)`: Use this function (created when generating the AXI4-Full IP) to read a 32-bit word from the AXI4-Full Peripheral.
  - ✓ `Base Address`: In the FIFO case, any address in the 64-byte range (the one allowed for the AXI4-Full Pipelined 2D Convolution Kernel peripheral) works since we only write/read to/from the FIFOs.
  - ✓ For example, for the given with the given Pipelined 2D Conv Kernel parameters (N=3, B=C=8), we are expected to get:

Input (D)	Output (F)
0xA1B2C3D4 0xF0E1D2C3 0x000000B3	0x00002B82
0xF109050A 0xC302A1F0 0x0000001C	0x000018FF

RECRLab@OU

2D DCT (DISCRETE COSINE TRANSFORM): CUSTOM PERIPHERAL FOR AXI4-FULL INTERFACE

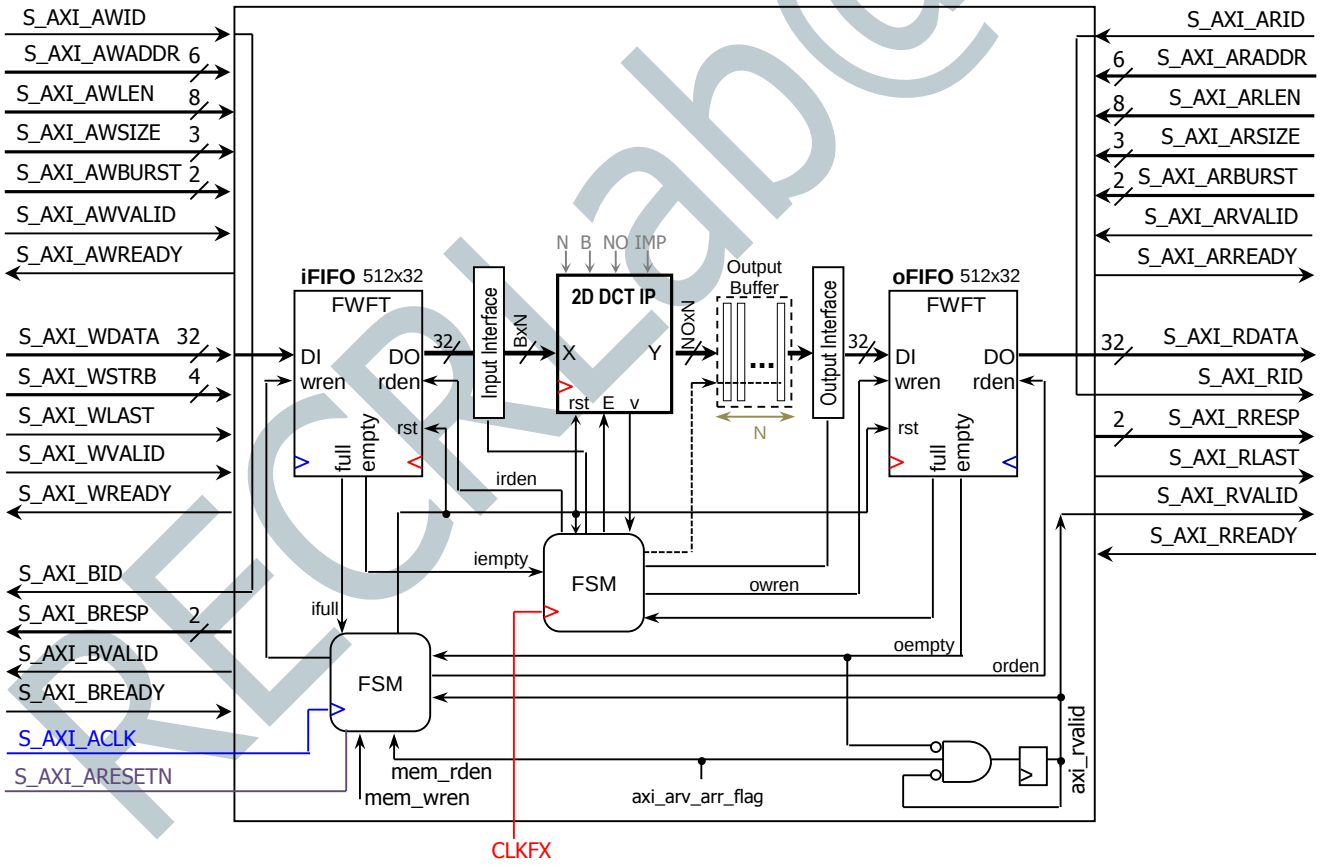
CONSIDERATIONS

- We will use the 2D DCT hardware that allows for transforms of sizes 4, 8, and 16.
- As for the AXI4-Full Peripheral, the VHDL files allows only for the case:
  - ✓ **FIFO**: Custom FIFO-based interface that we will be using. All writes/read to any of the 16-word memory positions is treated the same (writing/reading on the FIFO).

- Description of the hardware files (the most important ones) we are using:
  - ✓ `mydctfull_v1_0.vhd`: AXI4-Full Peripheral (top file, Vivado template).
  - ✓ `mydctfull_v1_0_S00_AXI.vhd`: AXI4-Full Interface description (edited Vivado template)
  - ✓ `myAXI_IP.vhd, my_AXI_fifo.vhd, my_genpulse_sclr.vhd`: Ancillary files for the AXI4-Full Peripheral.
  - ✓ `dct_ip.vhd`: top file of the hardware that runs at CLKFX. This includes the 2D DCT IP, the input/output interfaces to the FIFOs, the Output Buffer, and the FSM@CLKFX.
  - ✓ `DCT_2d.vhd`: Top file of the 2D DCT hardware architecture
  - ✓ `fsm_fullypip.vhd, fsm_onetrans.vhd`: Hardware description of the FSM @ CLKFX.
  - ✓ `DCT4_NH16_LUT_values[1..4], DCTeo8_NH16_LUT_values[1..8], DCTeo16_NH16LUT_values[1..16].txt`: LUT values for the DCT Transform coefficients specified in text files.

- 2D DCT parameters: In `my_AXI_fifo.vhd`, we can modify: N (transform size: 4, 8, 16), B (input pixel bitwidth: 8, 16), NO (output pixel bitwidth: 8,16), IMP (implementation type: folded, pipelined), NH (coefficient bit-width). We fix NH=16.

- 2D DCT AXI4-Full Peripheral (FIFO-based) with AXI signals (we make `S_AXI_CLK=CLK_FX`).



IP GENERATION

- Create new project in Vivado: `myaxifulldct`
  - ✓ Make sure the default language is VHDL, so that the system wrapper and the template files are created in VHDL
  - ✓ At Default Part, go to Boards, and select the **Zybo** (or **Zybo Z7-10**) Board.
- From the menu bar, select **Tools → Create and Package New IP**.
  - ✓ Create a new AXI4 Peripheral. Name: `mydctfull`. Location: `/ip_repo`.
    Peripheral Repositories tip: To add a previously-generated IP into a new project, go to: Project Settings → IP → IP repositories and point to the associated repository folder.

RECRLab@OU

- ✓ Add Interface: Full, 32 bits. Interface Mode: Slave. Memory Size: 64 bytes (16 32-bit words)
- ✓ Select Edit IP. A New project appears, open it and look for the `<peripheral name>_S00_AXI.vhd` file (in this case it will be `mydctfull_v1_0_S00_AXI.vhd`). Modify the project:
  - Replace `mydctfull_v1_0_s00_AXI.vhd` with our edited file `mydctfull_v1_0_S00_AXI.vhd`. Same for the file `mydctfull_v1_0.vhd` (this is not strictly necessary).
  - Add the extra files to the folder `/hdl` in `/ip_repo/mydctfull_1.0` and add these source files (including ALL the `.txt` files) to the Vivado project. * Vivado 2019.1: by default, the files will be added to the folder `/src`.
- ✓ There is no need to add ports as our peripheral does not include external I/Os.
- ✓ Synthesize your circuit (just to double-check everything is ok): You should've simulated the code in a different project.
- ✓ Go to Package IP – mydctfull → File Groups (Merge changes). Then Review and Package → Re-Package IP.

- Your custom IP is now ready to be used as an AXI4-Full Peripheral.
- You will return to the original Vivado Project.

CREATING A BLOCK DESIGN PROJECT IN VIVADO

- Click on Create Block Design and instantiate the Zynq PS and the AXI MYDCTFULL peripheral.
- Click on Run Block Automation and Run Connection Automation. Then 'Validate Design'
- There is no need to add an `.xdc` file as our peripheral does not use external ports.
- Create the VHDL wrapper (Sources Window → right click on the top-level system design → Create HDL Wrapper)
- Synthesize, implement, and generate the bitstream.
  - ✓ An error will be reported when Synthesizing. Vivado only copies VHDL files from the IP folder to the embedded project folder (located inside the `/<peripheral name>.srcs/…/ipshared` folder). As a result, the DCT design files cannot find the `.txt` files. We need to place these text files in the same folder as the rest of the design files.
  - ✓ This folder location is available by opening the `.vhd` file(s) that require these `.txt` files. Though you can find these files in the design structure, it is better to let the Vivado error point to the `.vhd` file(s) that "cannot open" the `.txt` files. After copying ALL of the `.txt` files, you can Synthesize again.
  - ✓ In general, this procedure is to be followed for any ancillary files (e.g. text files) used by the VHDL files.
- Export hardware (with bitstream) and launch SDK.

SDK

- See Tutorial Unit 2 for details on how to create and test software application on SDK.
- Go to Xilinx Tools → Repositories, click on 'New' and then browse to the folder `\ip_repo\mydctfull_1.0` and click ok.
- Create a new SDK application: dcttest. Then, copy the following file into the `/src` folder: `dct_tst.c`
- The software routine controls the AXI4-Full 2D DCT peripheral by writing and reading from memory positions.

- **Testing strategy**: For $N = 4,\ NO = 16,\ B = 8$. For every 2D Transform, we write 4 input columns (four 32-bit words), and we get 4 output columns (eight 32-bit words).
  - ✓ `MYDCTFULL_mWriteMemory(Base Address, 32-bit word)`: Use this function (created when generating the AXI-4 Full IP) to write a 32-bit word onto the AXI4-Full Peripheral.
  - ✓ `MYDCTFULL_mReadMemory(Base Address)`: Use this function (created when generating the AXI4-Full IP) to read a 32-bit word from the AXI4-Full Peripheral.
  - ✓ `Base Address`: In the FIFO case, any address in the 64-byte range (the one allowed for the AXI4-Full 2D DCT peripheral) works since we only write/read to/from the FIFOs.
  - ✓ For the given 2D DCT parameters, we provide two data sets. For the input values, each 32-bit word is a column; for the output values, each two 32-bit words is a row.

Input (columns)	Output (rows)
0xDEADBEEF	0x8000E92E
0xBEBEDEAD	0x14C00D82
0xFADEBEAD	0x18A6E418
0xCAFEBEDF	0xDB3E1FB2
	0x0A401E19
	0x1D40236D
	0xF8382A32
	0xDEC9FDE7

Input (columns)	Output (rows)
0xCFC7C9C7	0x80000CF4
0xCAC4C6C3	0xFF0003D5
0xC6C3C7C3	0x0471045F
0xBEBDC2BD	0xFF89FF65
	0x010003CE
	0x0000000B
	0x06D0FFE5
	0x00310020

RECRLABOU