Bridge of Life Education

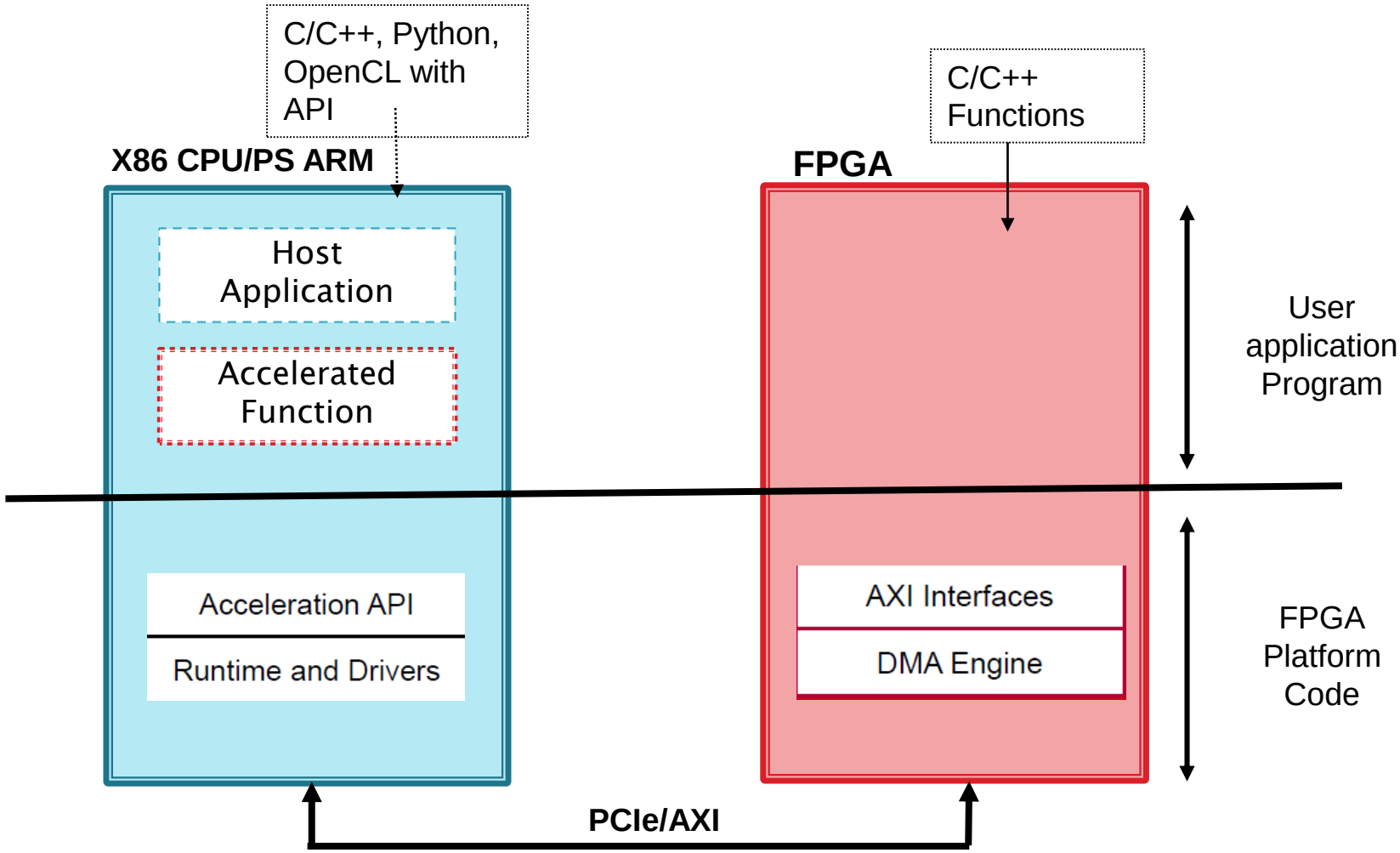
SOC Design

PYNQ/Lab2

Software & Hardware Co-design

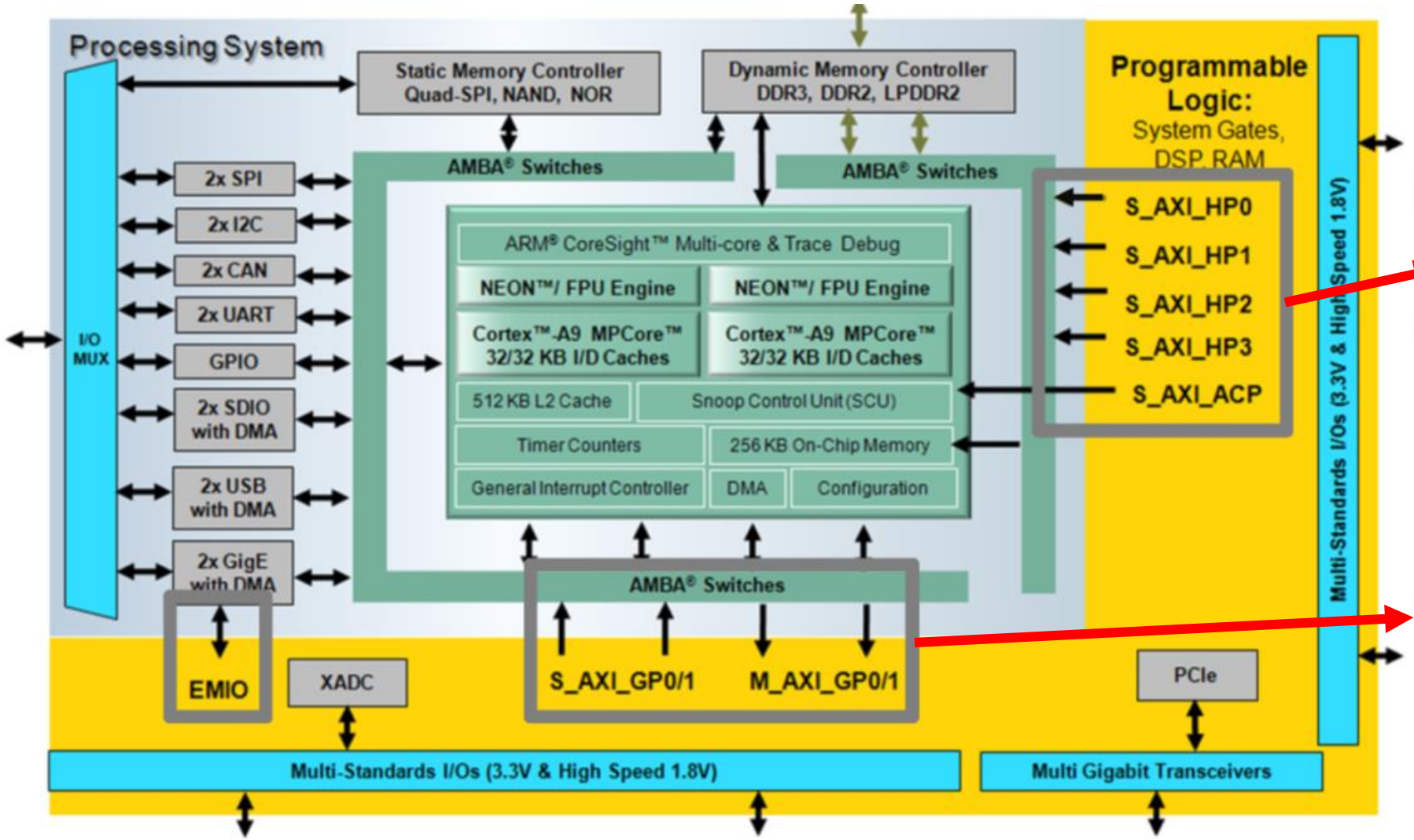
PYNQ Workshop: https://github.com/Xilinx/PYNQ_Workshop

Software Interacts with FPGA



Moving function to FPGA creates a lot of overhead.
Can we really gain performance and reduce power?

Zynq Block Diagram

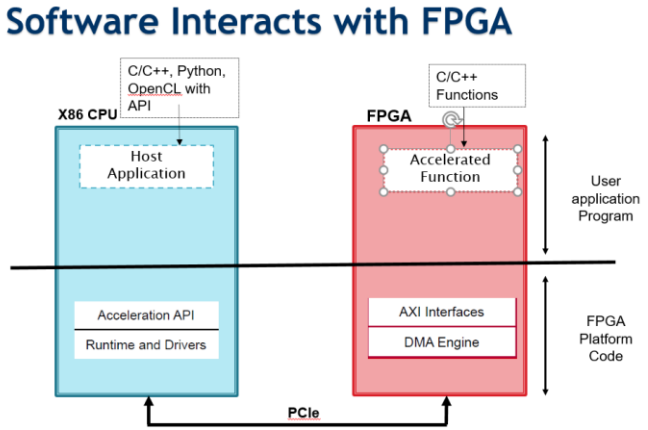


- **AXI High Performance**
  - 4x Slave (64-bit)
    - 1K FIFOs
- **ACP**
  - 1x Slave (64-bit)
  - Cache access

- **AXI General Purpose ports**
  - 2x Master (32-bit)
  - 2x Slave (32-bit)

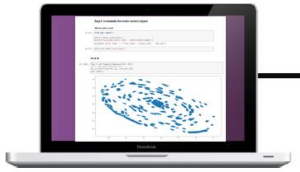
PYNQ = Embedded Jupyter Notebook

+ Pythonic integration of FPGA & Hard IPs



Jupyter notebooks, browser-based interface

Most competitive, open and advanced GUI



WAN

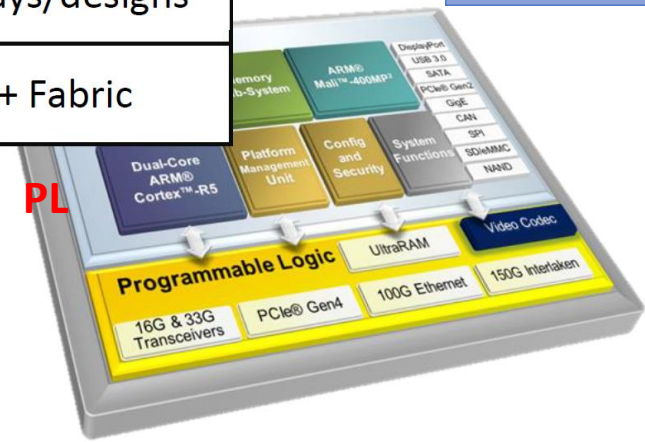
PYNQ enables JupyterLab on Zynq and ZU+

Jupyter web server	
IPython kernel	
Ubuntu-based Linux	Overlays/designs
ARM A9 / A53	ZU+ Fabric

Hardware C-drivers wrapped in Python packages

PS

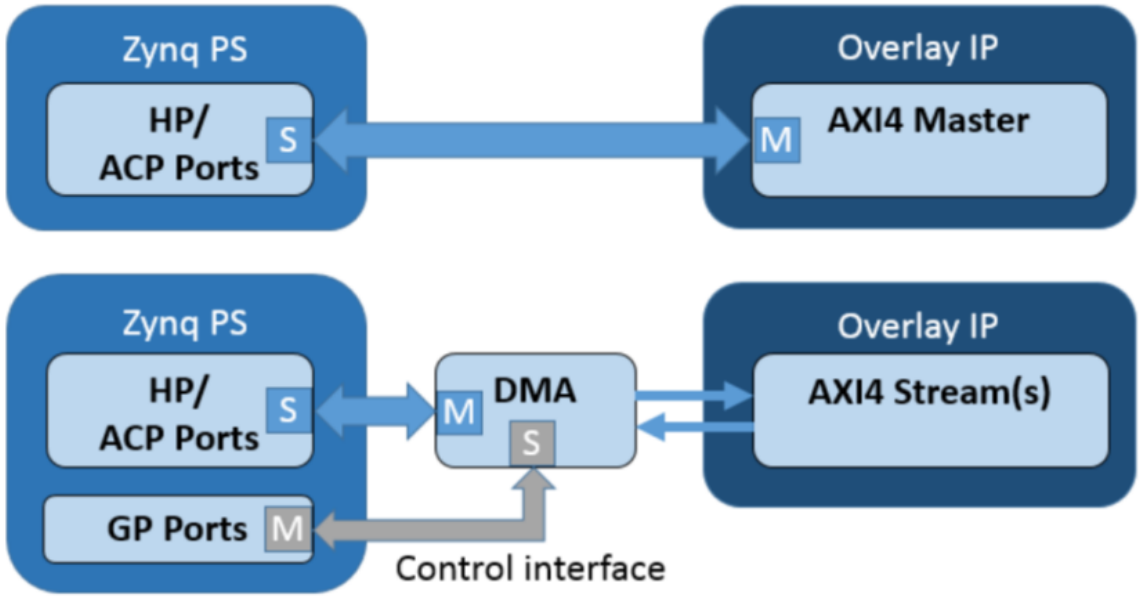
PL



Overlay – package of
- Design Bitstream
- Design metadata file (hwh)
- C Driver

Zynq PS-PL Interface

- **AXI High Performance**
  - 4x Slave (64-bit)
    - 1K FIFOs
- **ACP**
  - 1x Slave (64-bit)
  - Cache access



- **AXI Master**
  - AXI HP/ACP
  - Typically higher performance IP
- **AXI Stream**
  - Via DMA
  - HP/ACP ports for data path
  - GP slave for control

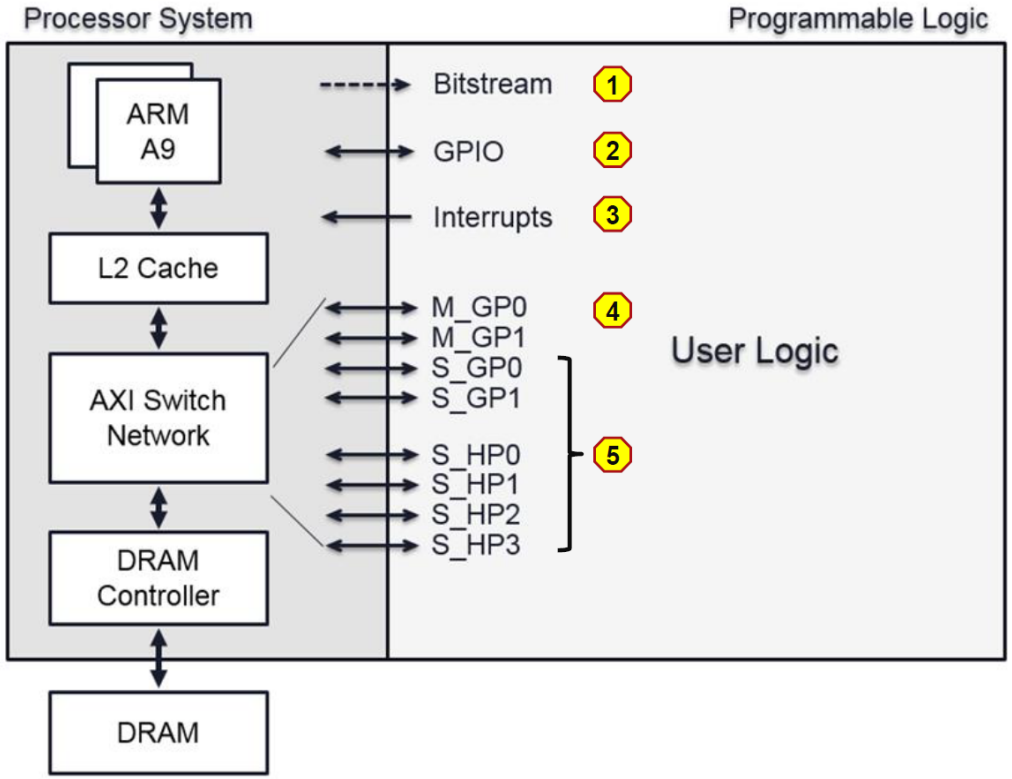
- **AXI General Purpose ports**
  - 2x Master (32-bit)
  - 2x Slave (32-bit)

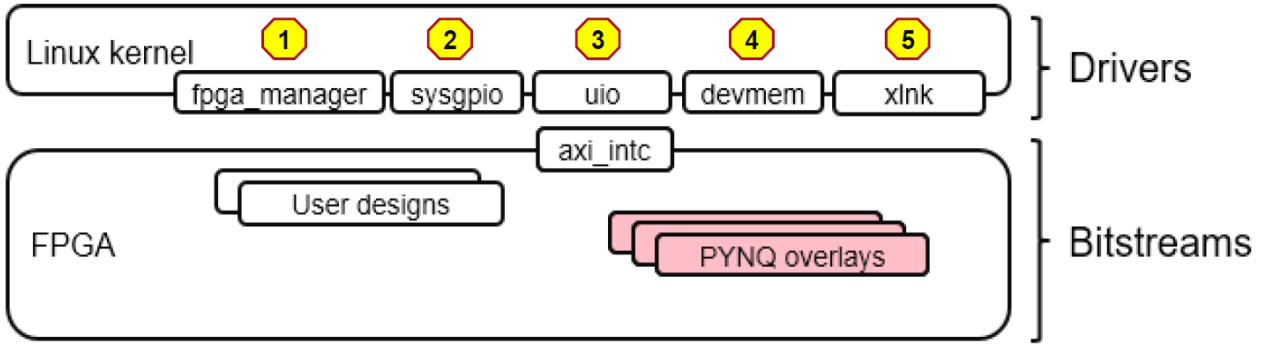
- **AXI (Lite) Slave IP**
  - General Purpose ports
  - Typically lower performance IP

AXI4-Stream Protocol: https://developer.arm.com/documentation/ihi0051/a/Introduction/About-the-AXI4-Stream-protocol
AMBA® AXI and ACE Protocol Specification: https://developer.arm.com/documentation/ihi0022/hc/?lang=en

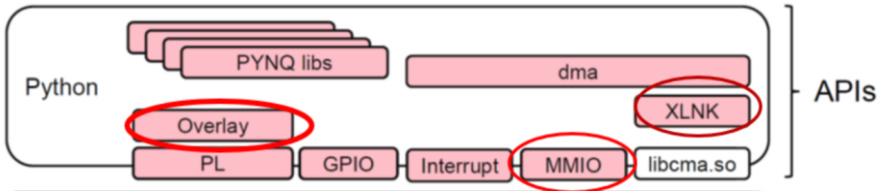
PYNQ provides Linux Drivers for PS-PL interface Wrapped in Python Libraries

Zynq

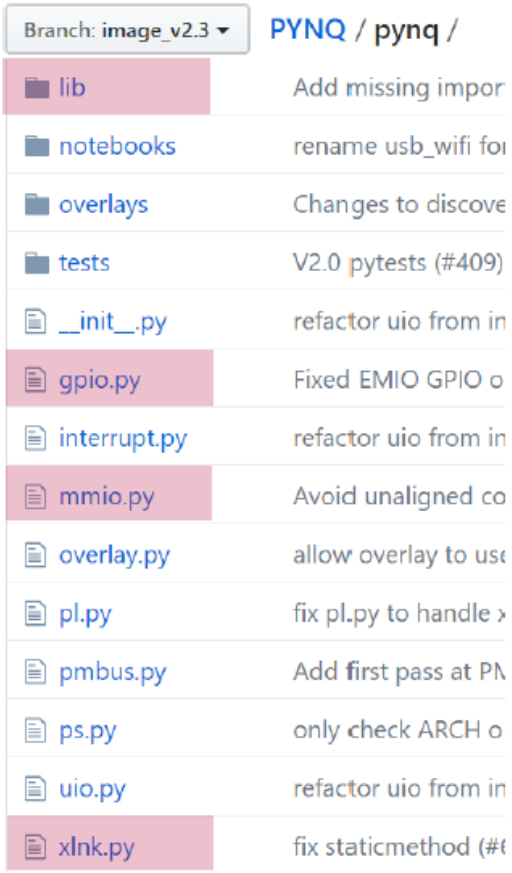


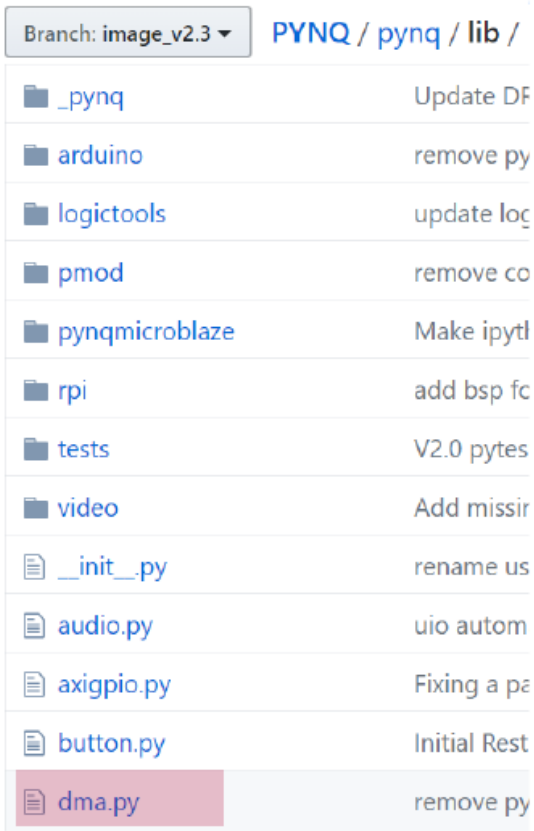


Pynq interface classes – run on PS



- **MMIO (pynq.mmio)**
  - Memory Mapped Input Output
  - Register based memory mapped transactions
- **Xlnk (pynq.xlnk)**
  - Memory allocation (used in DMA or for AXI Master peripheral)
- **DMA (pynq.lib.dma)**
  - Direct Memory Access
  - Offload memory transfers from main CPU
- **GPIO (pynq.gpio)**
  - Read/Write GPIO wires

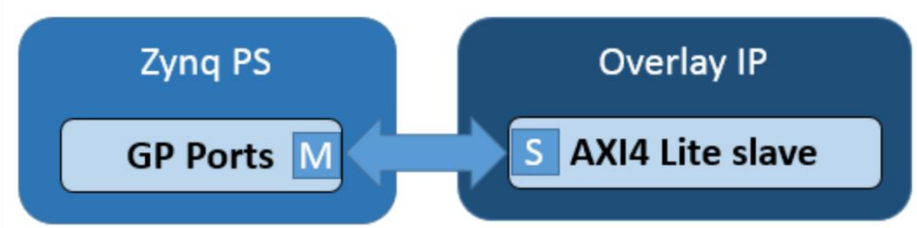




MMIO Class – memory mapped IO

MMIO is used to access and control peripherals

- Import MMIO
- Define memory mapped region
  - BASE_ADDRESS: starting location
  - ARRAY_SIZE: length of accessible memory (optional, default 4 bytes)
- **Read and Write 32-bit values**
  - ADDRESS_OFFSET: offset from BASE_ADDRESS, should be multiple of 4
  - Need to ensure the memory can be read/written



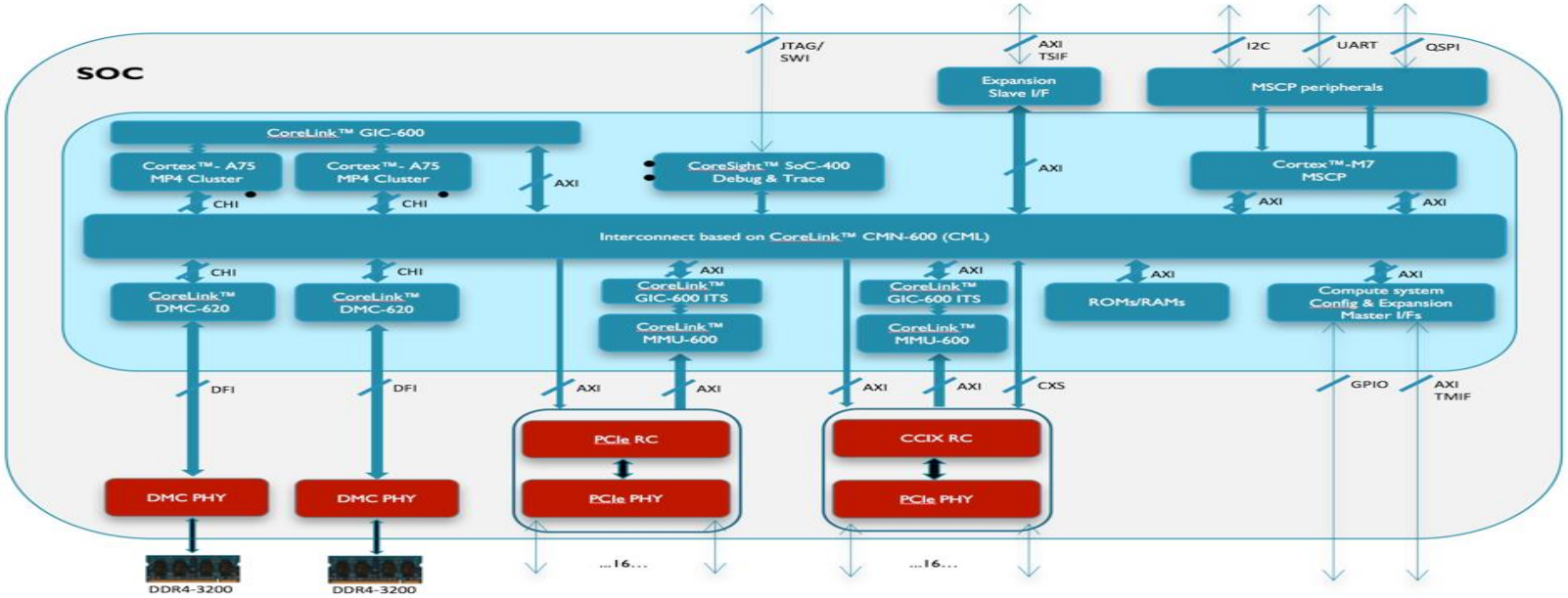

```
IP_BASE_ADDRESS = 0x40000000
ADDRESS_RANGE = 0x1000
ADDRESS_OFFSET = 0x10

from pynq import MMIO
mmio = MMIO(IP_BASE_ADDRESS, ADDRESS_RANGE)

data = 0xdeadbeef
mmio.write(ADDRESS_OFFSET, data)
result = mmio.read(ADDRESS_OFFSET)
```

System memory-map explained

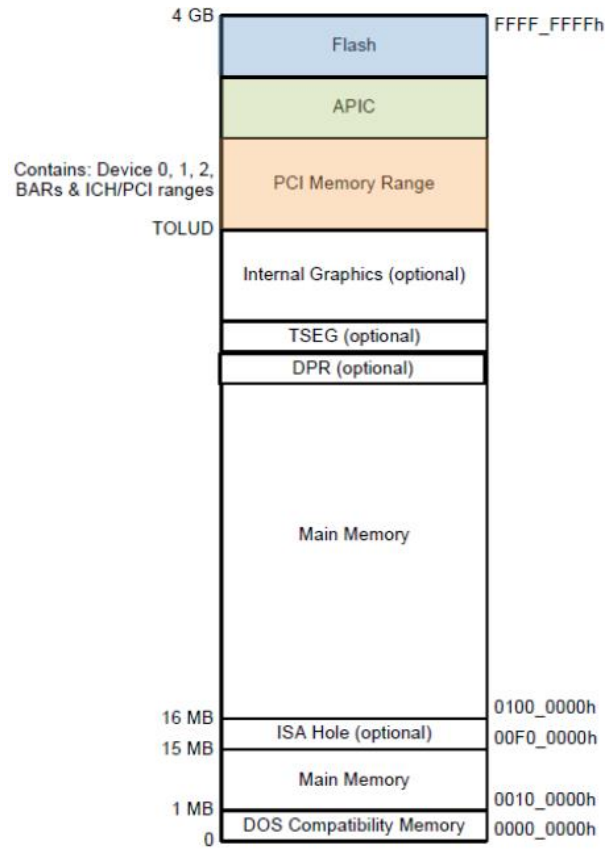
What if the configuration register is a byte?

Memory-Mapped IO Explained

- X86 Address Space: Memory, IO, SMM
- Use Memory Semantics for IO access
- Accesses to these memory ranges are decoded, transaction sent to the device
- Memory holes ?



X86 System Memory/IO



https://opensecuritytraining.info/IntroBIOS_files/Day1_04_Advanced%20x86%20-%20BIOS%20and%20SMM%20Internals%20-%20IO.pdf

Allocate : Class for Memory Allocation

Before IP in the PL accesses PS Memory, memory must first be allocated

- Import allocate
- Allocate physically contiguous memory Buffer object
- Buffer is a sub-class of numpy.ndarray
- Once buffer is allocated a DMA can be used to transfer data between PS/PL

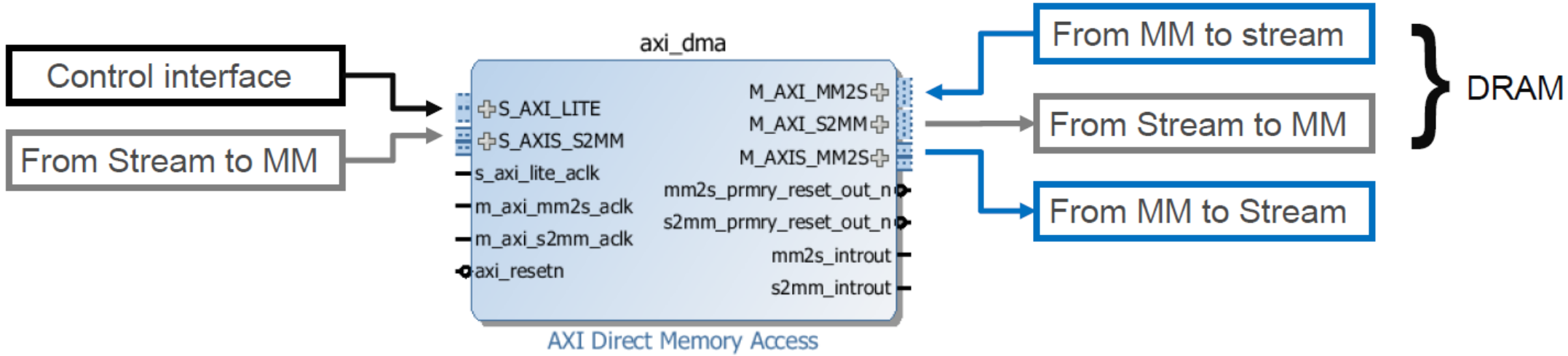
```
from pynq import allocate
ol = Overlay("/home/xilinx/IPBitFile/FIRN11Stream.bit")
ipFIRN11 = ol.fir_n11_strm_0
ipDMAIn = ol.axi_dma_in_0
ipDMAOut = ol.axi_dma_out_0

inBuffer0 = allocate(shape=(N,) dtype=np.int32)
outBuffer0 = allocate(shape=(N,), dtype=np.int32)

for i in range(N):
    line = fiSamples.readline()
    inBuffer0[i] = int(line)

ipFIRN11.write(0x80, len(inBuffer0) * 4)
ipFIRN11.write(0x00, 0x01)
ipDMAIn.sendchannel.transfer(inBuffer0)
ipDMAOut.recvchannel.transfer(outBuffer0)
```

DMA Class

- **Direct memory access**
  - Transfer data between memories directly
    - PS - PL
  - Bypasses CPU
    - Doesn't waste CPU cycles on data transfer
  - Speed up memory transfers with burst transactions
- **Xilinx *AXI Direct Memory Access* IP block supported in PYNQ**
  - Read and Write Paths from PL to DDR and DDR to PL
  - Memory Mapped to Stream
  - Stream to Memory Mapped
- **Needs to stream to/from an allocated memory buffer**
  - PYNQ DMA class inherits from xlnk for memory allocation

AXI Direct Memory Access IP

- **AXI Lite control interface (AXI GP port)**
- **Memory mapped interface (AXI interface, HP/ACP ports)**
- **AXI Stream interface (AXI stream accelerator)**
- **Transfer between streams and memory mapped locations**
  - Paths from PL to DRAM and DRAM to PL
  - Memory Mapped to Stream (MM2S)/Stream to Memory Mapped (S2MM)



Python overlay API

- **PYNQ Overlay class supports basic overlay functionality**
  - Requires Tcl/HWH
  - Allows download, and overlay discovery (ip_dict)
  - Assigns default drivers to IP
    - Read() and write() access to IP address space

```
from pynq import Overlay
overlay = Overlay("pynqtutorial.bit")
```

```
help(overlay)
```

```
class Overlay(pynq.pl.Bitstream)
 |  Default documentation for overlay pynqtutorial.bit.
 |  attributes are available on this overlay:
 |
 |  IP Blocks
 |  ----------
 |  axi_dma_from_pl_to_ps : pynq.lib.dma.DMA
 |  axi_dma_from_ps_to_pl : pynq.lib.dma.DMA
 |  btns_gpio             : pynq.lib.axigpio.AxiGPIO
 |  mb_bram_ctrl_1        : pynq.overlay.DefaultIP
 |  mb_bram_ctrl_2        : pynq.overlay.DefaultIP
 |  rgbleds_gpio          : pynq.lib.axigpio.AxiGPIO
 |  swsleds_gpio          : pynq.lib.axigpio.AxiGPIO
 |  system_interrupts     : pynq.overlay.DefaultIP
```

```
overlay.rgbleds_gpio.write(0, 3)
overlay.swsleds_gpio.read()
```

Lab#1, 2 – PYNQ

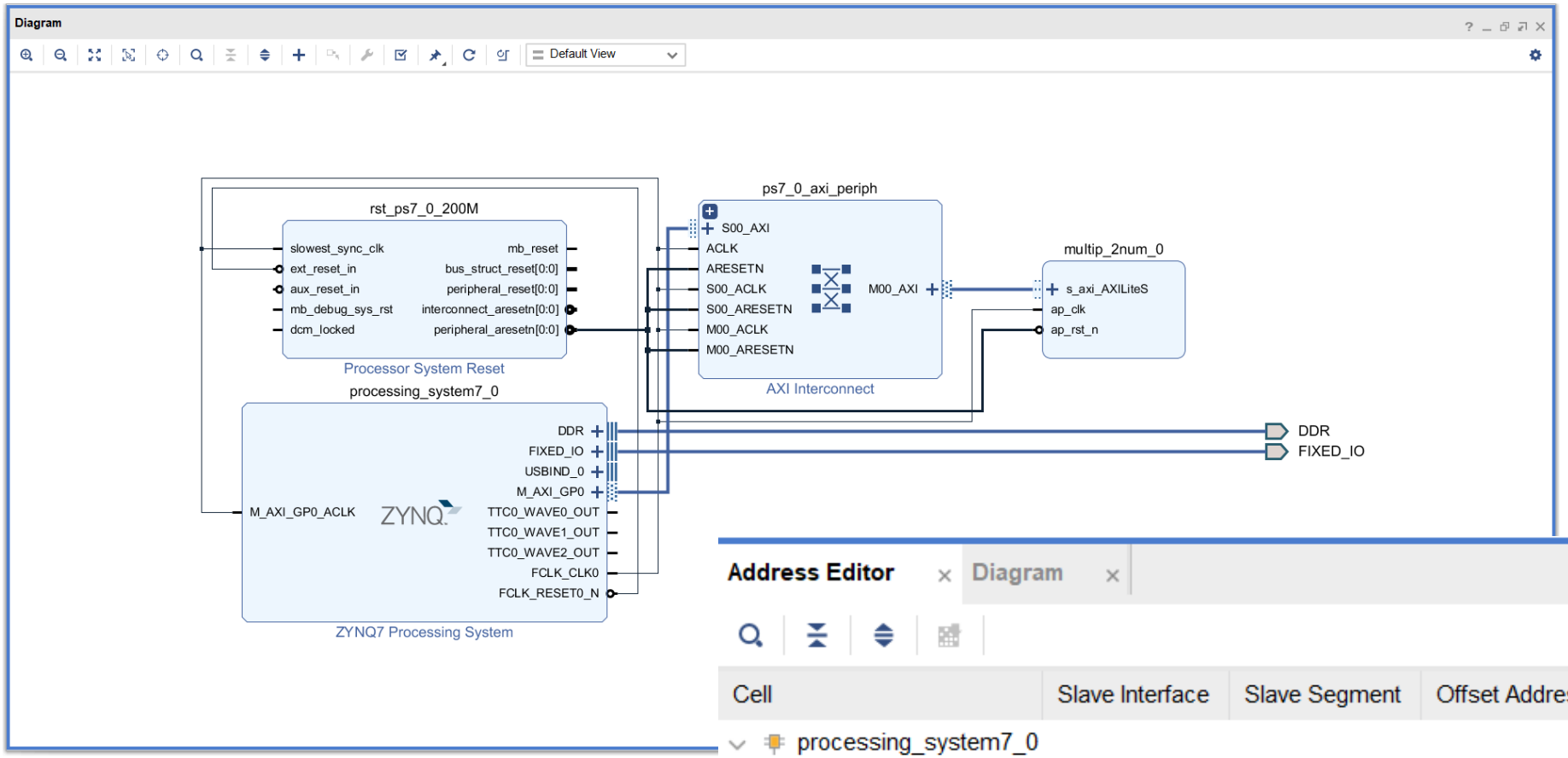
1. axi-lite – Multiplication
2. axi-m – FIR11
3. axi-s – FIR11

PYNQ Workshop: https://github.com/Xilinx/PYNQ_Workshop

Lab#1 - Multiplication

```
void multip_2num(int32_t n32In1, int32_t n32In2, int32_t* pn32ResOut)
{
#pragma HLS INTERFACE s_axilite port=pn32ResOut
#pragma HLS INTERFACE s_axilite port=n32In2
#pragma HLS INTERFACE s_axilite port=n32In1
#pragma HLS TOP name=multip_2num

    *pn32ResOut = n32In1 * n32In2;

    return;
}
```

BD for Multiplication AXI-Lite



Address Editor | Diagram

Cell	Slave Interface	Slave Segment	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
multip_2num_0	s_axi_AXILiteS	Reg	0x43C0_0000	64K	0x43C0_FFFF

HWH/HLS Register Map for Multiplication

```
<MODULES>
  <MODULE ...| FULLNAME="/multip_2num_0" ...>
      <PARAMETER NAME="C_S_AXI_AXILITES_BASEADDR" VALUE="0x43C00000"/>
      <PARAMETER NAME="C_S_AXI_AXILITES_HIGHADDR" VALUE="0x43C0FFFF"/>
    </PARAMETERS>
```

```
void multip_2num(int32_t n32In1, int32_t n32In2, int32_t* pn32ResOut)
{
#pragma HLS INTERFACE s_axilite port=pn32ResOut
#pragma HLS INTERFACE s_axilite port=n32In2
#pragma HLS INTERFACE s_axilite port=n32In1
#pragma HLS TOP name=multip_2num

    *pn32ResOut = n32In1 * n32In2;

    return;
}
```

Synthesis(solution1)(multip_2num_csynth.rpt) | xmultip_2num_hw.h

```
1  // ==========================================================
2  // Vivado(TM) HLS - High-Level Synthesis from C, C++ and SystemC v2019.2 (64-bit)
3  // Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.
4  // ==========================================================
5  // AXILiteS
6  // 0x00 : reserved
7  // 0x04 : reserved
8  // 0x08 : reserved
9  // 0x0c : reserved
10 // 0x10 : Data signal of n32In1
11 //        bit 31~0 - n32In1[31:0] (Read/Write)
12 // 0x14 : reserved
13 // 0x18 : Data signal of n32In2
14 //        bit 31~0 - n32In2[31:0] (Read/Write)
15 // 0x1c : reserved
16 // 0x20 : Data signal of pn32ResOut
17 //        bit 31~0 - pn32ResOut[31:0] (Read)
18 // 0x24 : Control signal of pn32ResOut
19 //        bit 0  - pn32ResOut_ap_vld (Read/COR)
20 //        others - reserved
21 // (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)
22
23 #define XMULTIP_2NUM_AXILITES_ADDR_N32IN1_DATA     0x10
24 #define XMULTIP_2NUM_AXILITES_BITS_N32IN1_DATA     32
25 #define XMULTIP_2NUM_AXILITES_ADDR_N32IN2_DATA     0x18
26 #define XMULTIP_2NUM_AXILITES_BITS_N32IN2_DATA     32
27 #define XMULTIP_2NUM_AXILITES_ADDR_PN32RESOUT_DATA 0x20
28 #define XMULTIP_2NUM_AXILITES_BITS_PN32RESOUT_DATA 32
29 #define XMULTIP_2NUM_AXILITES_ADDR_PN32RESOUT_CTRL 0x24
30
```

Host Code for Multiplication

- Import MMIO (Involving in Overlay)
- Define memory mapped region
  - The base/offset is defined in hwh file
- Read and Write 32-bit values
- In read function, it shall check the ap_vld (0x24)

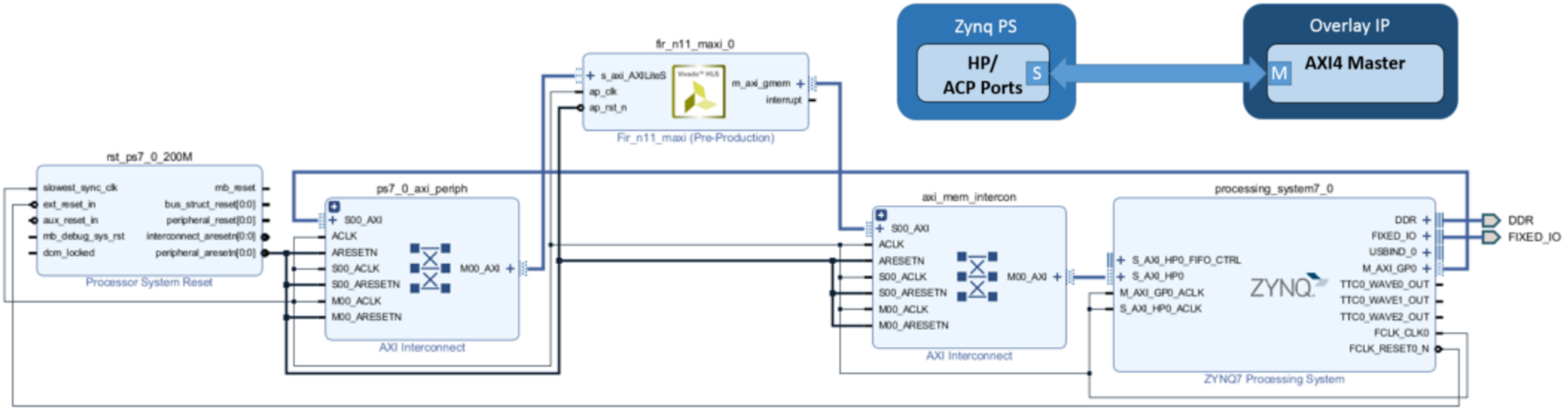
```
ol = Overlay("/home/xilinx/IPBitFile/Multip2Num.bit")
regIP = ol.multip_2num_0

for i in range(9):
    print("============================")
    for j in range(9):
        regIP.write(0x10, i + 1)
        regIP.write(0x18, j + 1)
        Res = regIP.read(0x20)
        print(str(i + 1) + " * " + str(j + 1) + " = " + str(Res))
```

Lab#2 – FIR11 use AXI-Master

```
void fir_n11_maxi(
                volatile int32_t* pn32HPInput,
                volatile int32_t* pn32HPOutput,
                int32_t an32Coef[MAP_ALIGN_4INT],
                reg32_t regXferLeng )
{
```

```
// directive.tcl
set_directive_interface -mode s_axilite "fir_n11_maxi"
set_directive_interface -mode m_axi -depth 128 -offset slave "fir_n11_maxi" pn32HPInput
set_directive_interface -mode m_axi -depth 128 -offset slave "fir_n11_maxi" pn32HPOutput
set_directive_interface -mode s_axilite "fir_n11_maxi" an32Coef
set_directive_interface -mode s_axilite "fir_n11_maxi" regXferLeng
```

BD for FIR AXI Master



Address Editor × **Diagram** ×

Cell	Slave Interface	Slave Segment	Offset Address	Range	High Address
fir_n11_maxi_0					
Data_m_axi_gmem (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
fir_n11_maxi_0	s_axi_AXILiteS	Reg	0x43C0_0000	64K	0x43C0_FFFF

HWH/HLS Register Map for FIR AXI Master

```
<MODULE ... FULLNAME="/fir_n11_maxi_0" ...>
  <PARAMETERS>
    <PARAMETER NAME="C_S_AXI_AXILITES_BASEADDR" VALUE="0x43C00000"/>
    <PARAMETER NAME="C_S_AXI_AXILITES_HIGHADDR" VALUE="0x43C0FFFF"/>
```

```
// 0x00 : Control signals
//        bit 0  - ap_start (Read/Write/COH)
//        bit 1  - ap_done (Read/COR)
//        bit 2  - ap_idle (Read)
//        bit 3  - ap_ready (Read)
//        bit 7  - auto_restart (Read/Write)
//        others - reserved
// 0x04 : Global Interrupt Enable Register
//        bit 0  - Global Interrupt Enable (Read/Write)
//        others - reserved
```

```
void fir_n11_maxi(volatile int32_t* pn32HPInput,
                  volatile int32_t* pn32HPOutput,
                  int32_t an32Coef[MAP_ALIGN_4INT],
                  reg32_t regXferLeng)
{
```

Synthesis(solution1)(fir_n11_maxi_csynth.rpt) | xfir_n11_maxi_hw.h

```
// 0x08 : IP Interrupt Enable Register (Read/Write)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
//        others - reserved
// 0x0c : IP Interrupt Status Register (Read/TOW)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
//        others - reserved
// 0x10 : Data signal of pn32HPInput
//        bit 31~0 - pn32HPInput[31:0] (Read/Write)
// 0x14 : reserved
// 0x18 : Data signal of pn32HPOutput
//        bit 31~0 - pn32HPOutput[31:0] (Read/Write)
// 0x1c : reserved
// 0x80 : Data signal of regXferLeng_V
//        bit 31~0 - regXferLeng_V[31:0] (Read/Write)
// 0x84 : reserved
// 0x40 ~
// 0x7f : Memory 'an32Coef' (12 * 32b)
//        Word n : bit [31:0] - an32Coef[n]
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)

#define XFIR_N11_MAXI_AXILITES_ADDR_AP_CTRL             0x00
#define XFIR_N11_MAXI_AXILITES_ADDR_GIE                 0x04
#define XFIR_N11_MAXI_AXILITES_ADDR_IER                 0x08
#define XFIR_N11_MAXI_AXILITES_ADDR_ISR                 0x0c
#define XFIR_N11_MAXI_AXILITES_ADDR_PN32HPINPUT_DATA    0x10
#define XFIR_N11_MAXI_AXILITES_BITS_PN32HPINPUT_DATA    32
#define XFIR_N11_MAXI_AXILITES_ADDR_PN32HPOUTPUT_DATA   0x18
#define XFIR_N11_MAXI_AXILITES_BITS_PN32HPOUTPUT_DATA   32
#define XFIR_N11_MAXI_AXILITES_ADDR_REGXFERLENG_V_DATA  0x80
#define XFIR_N11_MAXI_AXILITES_BITS_REGXFERLENG_V_DATA  32
#define XFIR_N11_MAXI_AXILITES_ADDR_AN32COEF_BASE       0x40
#define XFIR_N11_MAXI_AXILITES_ADDR_AN32COEF_HIGH       0x7f
#define XFIR_N11_MAXI_AXILITES_WIDTH_AN32COEF           32
```

Host Code for FIR AXI Master

- Open Overlay ”FIRN11MAXI.bit”
- Identify kernel “fir_n11_maxi_0“
- Allocate inBuffer0, outBuffer0 – array physical address and size
- Fill in coefficient location starts at 0x40 (defined in .hwh)
- Specify transfer length at 0x80
- Specify inBuffer0, outBuffer0 physical address
- Start the kernel.

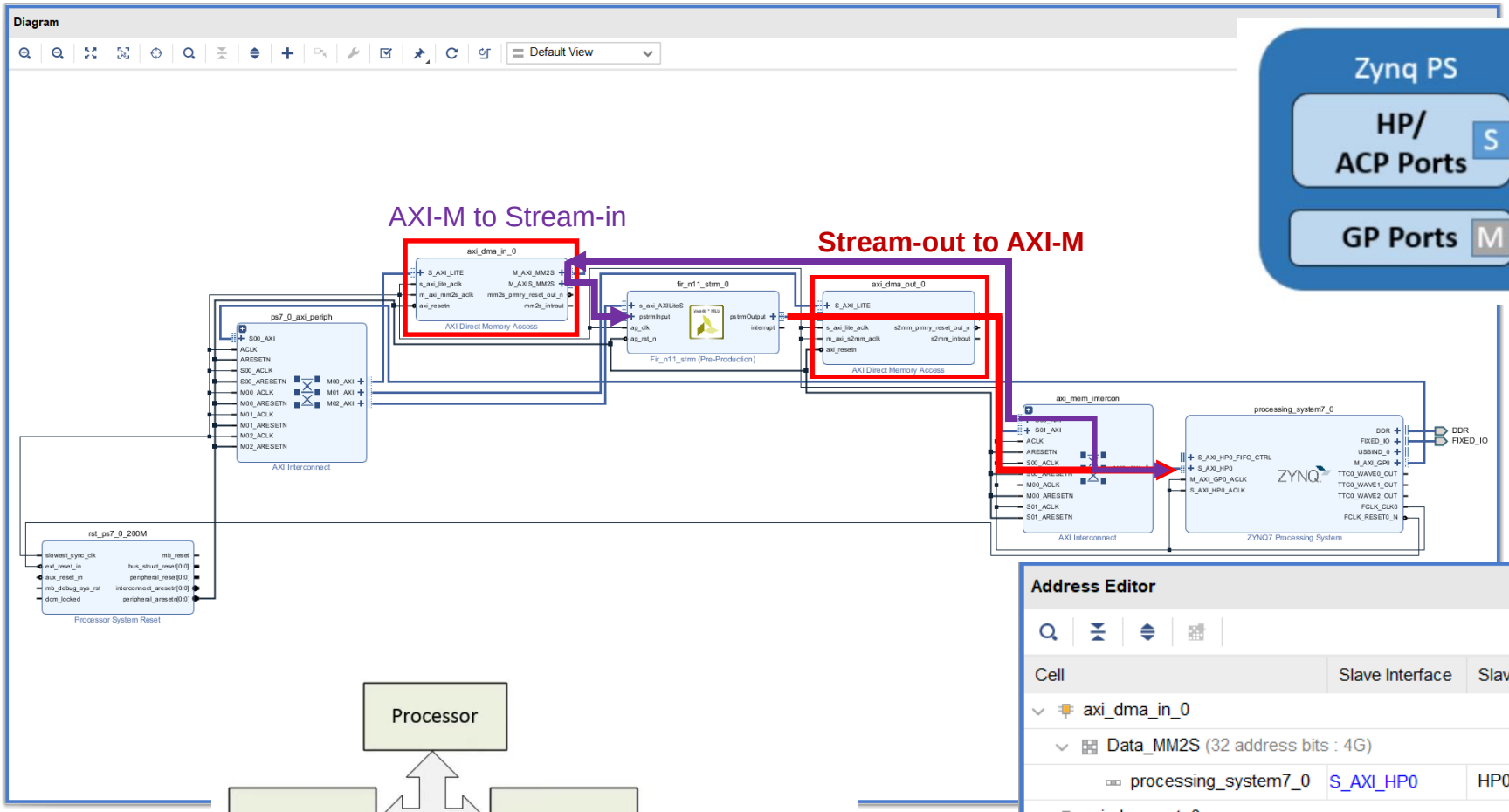
```
ol = Overlay("/home/xilinx/IPBitFile/FIRN11MAXI.bit")
ipFIRN11 = ol.fir_n11_maxi_0

fiSamples = open("samples_triangular_wave.txt", "r+")
numSamples = 0
line = fiSamples.readline()
while line:
    numSamples = numSamples + 1
    line = fiSamples.readline()

inBuffer0 = allocate(shape=(numSamples,), dtype=np.int32)
outBuffer0 = allocate(shape=(numSamples,), dtype=np.int32)
fiSamples.seek(0)
for i in range(numSamples):
    line = fiSamples.readline()
    inBuffer0[i] = int(line)
fiSamples.close()

numTaps = 11
n32Taps = [0, -10, -9, 23, 56, 63, 56, 23, -9, -10, 0]
#n32Taps = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
n32DCGain = 0
timeKernelStart = time()
for i in range(numTaps):
    n32DCGain = n32DCGain + n32Taps[i]
    ipFIRN11.write(0x40 + i * 4, n32Taps[i])
if n32DCGain < 0:
    n32DCGain = 0 - n32DCGain
ipFIRN11.write(0x80, len(inBuffer0) * 4)
ipFIRN11.write(0x10, inBuffer0.device_address)
ipFIRN11.write(0x18, outBuffer0.device_address)
ipFIRN11.write(0x00, 0x01)
while (ipFIRN11.read(0x00) & 0x4) == 0x0:
    continue
timeKernelEnd = time()
```

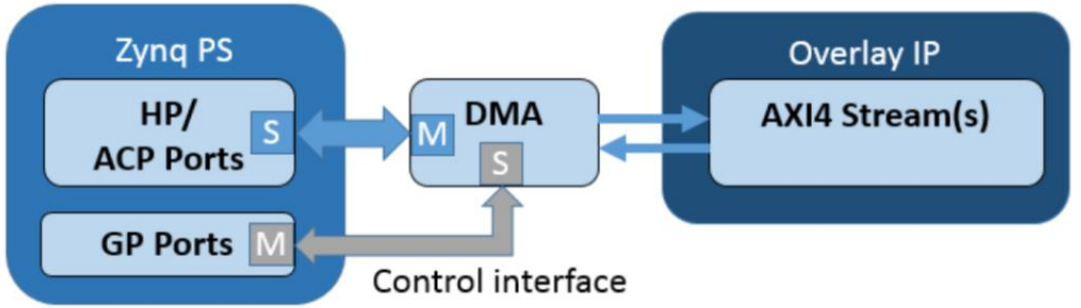
Lab#2 – FIR11 – axi-s

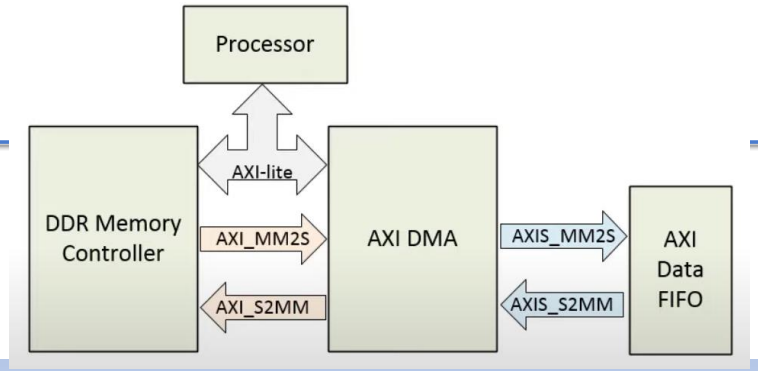
```
typedef ap_axiu<32,1,1,1> value_t;
typedef hls::stream<value_t> stream_t;
typedef ap_uint<32> reg32_t;
typedef signed int int32_t;
typedef unsigned int uint32_t;

void fir_n11_strm(
        stream_t* pstrmInput,
        stream_t* pstrmOutput,
        int32_t an32Coef[MAP_ALIGN_4INT],
        reg32_t regXferLeng) {

#pragma HLS INTERFACE s_axilite       port=regXferLeng
#pragma HLS INTERFACE s_axilite       port=an32Coef
#pragma HLS INTERFACE axis register both  port=pstrmOutput
#pragma HLS INTERFACE axis register both  port=pstrmInput
#pragma HLS INTERFACE s_axilite       port=return
```

```
    for (n32XferCnt = 0; n32XferCnt < n32NumXfer4B; n32XferCnt++) {
        n32Acc = 0;
        value_t valTemp = pstrmInput->read();
        n32Temp = valTemp.data;
SHIFT_ACC_LOOP:
        for (n32Loop = N - 1; n32Loop >= 0; n32Loop--) {
            if (n32Loop == 0) {
                an32ShiftReg[0] = n32Temp;
                n32Data = n32Temp;
            } else {
                an32ShiftReg[n32Loop] = an32ShiftReg[n32Loop - 1];
                n32Data = an32ShiftReg[n32Loop];
            }
            n32Acc += n32Data * an32Coef[n32Loop];
        }
        valTemp.data = n32Acc;
        pstrmOutput->write(valTemp);
        if (valTemp.last) break;
    }
```

BD for FIR Stream







Address Editor

Cell	Slave Interface	Slave Segment	Offset Address	Range	High Address
axi_dma_in_0					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
axi_dma_out_0					
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_dma_in_0	S_AXI_LITE	Reg	0x4040_0000	64K	0x4040_FFFF
axi_dma_out_0	S_AXI_LITE	Reg	0x4041_0000	64K	0x4041_FFFF
fir_n11_strm_0	s_axi_AXILiteS	Reg	0x43C0_0000	64K	0x43C0_FFFF

HWH/HLS Register Map for FIR Stream

```
<MODULE ... FULLNAME="/fir_n11_strm_0" ...>
    <PARAMETERS>
      <PARAMETER NAME="C_S_AXI_AXILITES_BASEADDR" VALUE="0x43C00000"/>
      <PARAMETER NAME="C_S_AXI_AXILITES_HIGHADDR" VALUE="0x43C0FFFF"/>
```

Synthesis(solution1)(fir_n11_strm_csynth.rpt) | xfir_n11_strm_hw.h

```
// 0x00 : Control signals
//        bit 0  - ap_start (Read/Write/COH)
//        bit 1  - ap_done (Read/COR)
//        bit 2  - ap_idle (Read)
//        bit 3  - ap_ready (Read)
//        bit 7  - auto_restart (Read/Write)
//        others - reserved
// 0x04 : Global Interrupt Enable Register
//        bit 0  - Global Interrupt Enable (Read/Write)
//        others - reserved
// 0x08 : IP Interrupt Enable Register (Read/Write)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
//        others - reserved
// 0x0c : IP Interrupt Status Register (Read/TOW)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
//        others - reserved
// 0x80 : Data signal of regXferLeng_V
//        bit 31~0 - regXferLeng_V[31:0] (Read/Write)
// 0x84 : reserved
// 0x40 ~
// 0x7f : Memory 'an32Coef' (12 * 32b)
//        Word n : bit [31:0] - an32Coef[n]
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)

#define XFIR_N11_STRM_AXILITES_ADDR_AP_CTRL            0x00
#define XFIR_N11_STRM_AXILITES_ADDR_GIE                0x04
#define XFIR_N11_STRM_AXILITES_ADDR_IER                0x08
#define XFIR_N11_STRM_AXILITES_ADDR_ISR                0x0c
#define XFIR_N11_STRM_AXILITES_ADDR_REGXFERLENG_V_DATA 0x80
#define XFIR_N11_STRM_AXILITES_BITS_REGXFERLENG_V_DATA 32
#define XFIR_N11_STRM_AXILITES_ADDR_AN32COEF_BASE      0x40
#define XFIR_N11_STRM_AXILITES_ADDR_AN32COEF_HIGH      0x7f
#define XFIR_N11_STRM_AXILITES_WIDTH_AN32COEF          32
#define XFIR_N11_STRM_AXILITES_DEPTH_AN32COEF          12
```

```
void fir_n11_strm(stream_t* pstrmInput,
                  stream_t* pstrmOutput,
                  int32_t an32Coef[MAP_ALIGN_4INT],
                  reg32_t regXferLeng)
{
```

HWH/Xilinx IP Register Map for Xilinx DMA

```
<MODULES>
  <MODULE ... FULLNAME="/axi_dma_in_0" ...>
    </PARAMETERS>
      <PARAMETER NAME="C_BASEADDR" VALUE="0x40400000"/>
      <PARAMETER NAME="C_HIGHADDR" VALUE="0x4040FFFF"/>
    </PARAMETERS>

  <MODULE ... FULLNAME="/axi_dma_out_0" ...>
    <PARAMETERS>
      <PARAMETER NAME="C_BASEADDR" VALUE="0x40410000"/>
      <PARAMETER NAME="C_HIGHADDR" VALUE="0x4041FFFF"/>
    </PARAMETERS>
```

Table 2-6: **Direct Register Mode Register Address Map**

Address Space Offset[1]	Name	Description
00h	MM2S_DMACR	MM2S DMA Control register
04h	MM2S_DMASR	MM2S DMA Status register
08h – 14h	Reserved	N/A
18h	MM2S_SA	MM2S Source Address. Lower 32 bits of address.
1Ch	MM2S_SA_MSB	MM2S Source Address. Upper 32 bits of address.
28h	MM2S_LENGTH	MM2S Transfer Length (Bytes)
30h	S2MM_DMACR	S2MM DMA Control register
34h	S2MM_DMASR	S2MM DMA Status register
38h – 44h	Reserved	N/A
48h	S2MM_DA	S2MM Destination Address. Lower 32 bit address.
4Ch	S2MM_DA_MSB	S2MM Destination Address. Upper 32 bit address.
58h	S2MM_LENGTH	S2MM Buffer Length (Bytes)

https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf

Host Code for FIR Stream

- Import MMIO (in Overlay)
- Define memory mapped region
  - BASE_ADDRESS
  - ARRAY_SIZE
- Read and Write 32-bit values
  - ADDRESS_OFFSET from BASE_ADDRESS

```
ol = Overlay("/home/xilinx/IPBitFile/FIRN11Stream.bit")
ipFIRN11 = ol.fir_n11_strm_0
ipDMAIn = ol.axi_dma_in_0
ipDMAOut = ol.axi_dma_out_0

fiSamples = open("samples_triangular_wave.txt", "r+")
numSamples = 0
line = fiSamples.readline()
while line:
    numSamples = numSamples + 1
    line = fiSamples.readline()

inBuffer0 = allocate(shape=(numSamples,), dtype=np.int32)
outBuffer0 = allocate(shape=(numSamples,), dtype=np.int32)
fiSamples.seek(0)
for i in range(numSamples):
    line = fiSamples.readline()
    inBuffer0[i] = int(line)
fiSamples.close()

numTaps = 11
n32Taps = [0, -10, -9, 23, 56, 63, 56, 23, -9, -10, 0]
#n32Taps = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
n32DCGain = 0
timeKernelStart = time()
for i in range(numTaps):
    n32DCGain = n32DCGain + n32Taps[i]
    ipFIRN11.write(0x40 + i * 4, n32Taps[i])
if n32DCGain < 0:
    n32DCGain = 0 - n32DCGain
ipFIRN11.write(0x80, len(inBuffer0) * 4)
ipFIRN11.write(0x00, 0x01)
ipDMAIn.sendchannel.transfer(inBuffer0)
ipDMAOut.recvchannel.transfer(outBuffer0)
ipDMAIn.sendchannel.wait()
ipDMAOut.recvchannel.wait()
timeKernelEnd = time()
```

Host Code for Xilinx DMA

- Import MMIO (Involving in Overlay)
- Define memory mapped region
  - BASE_ADDRESS
  - ARRAY_SIZE
- Allocate memory buffer
  - PHYSICAL_ADDRESS
  - BUFFER_LENGTH
- DMA transfer
  - Direct memory access

Experiment:

1. **Move around the code**
2. **Add delay**

```
ol = Overlay("/home/xilinx/IPBitFile/FIRN11Stream.bit")
ipFIRN11 = ol.fir_n11_strm_0
ipDMAIn = ol.axi_dma_in_0
ipDMAOut = ol.axi_dma_out_0

fiSamples = open("samples_triangular_wave.txt", "r+")
numSamples = 0
line = fiSamples.readline()
while line:
    numSamples = numSamples + 1
    line = fiSamples.readline()

inBuffer0 = allocate(shape=(numSamples,), dtype=np.int32)
outBuffer0 = allocate(shape=(numSamples,), dtype=np.int32)
fiSamples.seek(0)
for i in range(numSamples):
    line = fiSamples.readline()
    inBuffer0[i] = int(line)
fiSamples.close()

numTaps = 11
n32Taps = [0, -10, -9, 23, 56, 63, 56, 23, -9, -10, 0]
#n32Taps = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
n32DCGain = 0
timeKernelStart = time()
for i in range(numTaps):
    n32DCGain = n32DCGain + n32Taps[i]
    ipFIRN11.write(0x40 + i * 4, n32Taps[i])
if n32DCGain < 0:
    n32DCGain = 0 - n32DCGain
ipFIRN11.write(0x80, len(inBuffer0) * 4)
ipFIRN11.write(0x00, 0x01)
ipDMAIn.sendchannel.transfer(inBuffer0)
ipDMAOut.recvchannel.transfer(outBuffer0)
ipDMAIn.sendchannel.wait()
ipDMAOut.recvchannel.wait()
timeKernelEnd = time()
```