Department of Electrical Engineering,
National Tsing Hua University

Advance SoC

Lab 4 – Caravel-FSIC FPGA

110590022 陳冠晰

110061217 王彥智

110000107 陳柏翰

GitHub Link:

https://github.com/vic9112/Advance_SOC

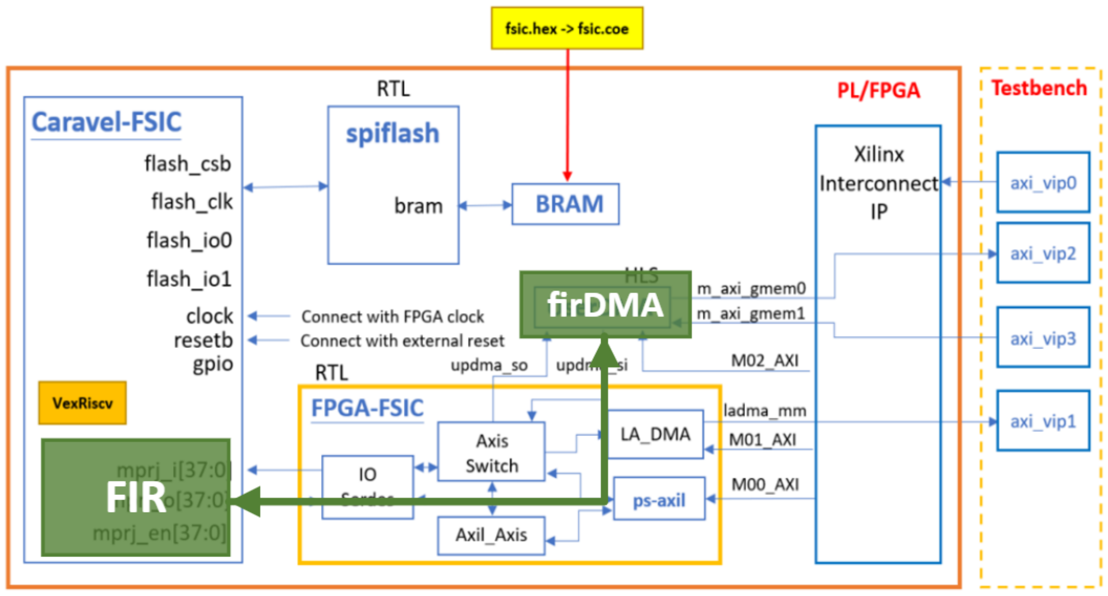
https://github.com/lukaschen1010

https://github.com/kenny0915

Table of Contents

1. Introduction

The purpose of this lab is to familiarize ourselves with Caravel-FSIC and FPGA simulation and validation. We integrated a user FIR into the User Project in Caravel SoC side, generating AXI-Lite transactions to initialize the FIR from the FPGA side. Then, we developed a DMA located at the FPGA side specifically for the FIR. This DMA uses AXI-Master to retrieve data (x_in) from memory and then streams them downstream to the FIR. After the FIR processes the data, it streams the results (y_out) upstream back to the DMA, which then stores them back into memory.



Where we have 4 'axi_vip' in the testbench

- axi_vip_0:
  - PS master cycle generation
  - Config module in the Caravel and ps_axi
- axi_vip_1
  - Memory slave module
  - Receive data from LA in caravel side
  - Flesh to the memory slave module from ps_axi
- axi_vip_2, 3
  - User DMA target
  - One is to read data pattern; the other one is to write data pattern.

2. Caravel-FSIC FPGA Simulation

2-1. DMA for FIR

The initial design of the DMA was for EdgeDetect, so we need to change the DMA's design to transmit and receive data for the FIR. The definitions of each register in DMA are shown in the following table:

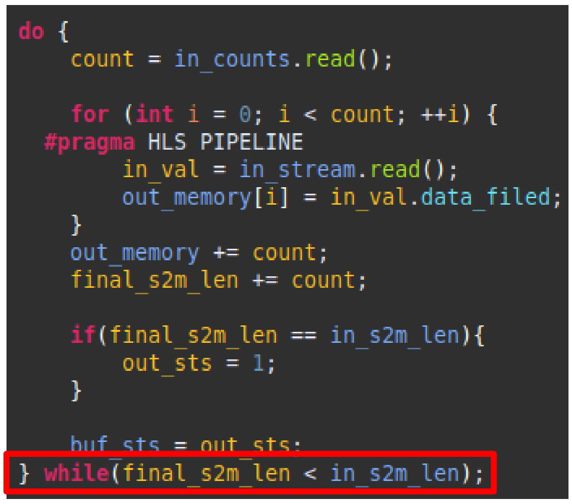
Table 1

DMA register table

Register Name	Offset Address	Description
Control signals	0x00	Control signals Definition bit 0 - ap_start (Read/Write/COH) Set 1 to start Axis DMA function bit 1 - ap_done (Read/COR) bit 2 - ap_idle (Read) bit 3 - ap_ready (Read/COR)
s2m Buffer transfer done status register	0x10	bit 0 – s2m buffer transfer done status (Read) Set this register to 1 if stream data has written to memory and data length is equal to Buffer Length(640*480/4 DW)
s2m Buffer transfer done status clear register	0x20	bit 0 – clear s2m buffer transfer done status (Read/Write) Set 1 to clear s2m buffer done status/s2m error status and reset internal state, then set 0 if finish to clear buffer done status Note: Before set this register to 1 to clear s2m buffer transfer done status/s2m error status, Clear status control register must set to 1. After buffer transfer done status is clear, this register needs to be cleared for next operation.
s2m Buffer Length	0x28	Set s2m buffer length, must set to 640*480/4.
s2m Clear Status Control register	0x30	bit 0 –Enable to clear s2m buffer transfer done status (Read/Write)
s2m Buffer Lower base address register	0x38	bit 31~0 – The memory base address [31:0] of s2m buffer (Read/Write)
s2m Buffer Upper base address register	0x3C	bit 31~0 – The memory base address [63:32] of s2m buffer (Read/Write)
s2m err status register	0x44	bit 0 – s2m err status when sof(start of frame) signal or eol(end of line) signal of side band of user project is incorrect (Read)
Image width	0x54	bit 31~0 – Image_width[31:0] (Read/Write) Note: The value of this register is DW unit, so the value should be real width divided by 4. This value must be 160, due to Image is 640(width)*480(height)
m2s Buffer Lower base address register	0x5C	bit 31~0 – The memory base address [31:0] of m2s buffer (Read/Write)
m2s Buffer Upper base address register	0x60	bit 31~0 – The memory base address [63:32] of m2s buffer (Read/Write)
m2s Buffer transfer done status register	0x68	bit 0 – m2s buffer transfer done status (Read) Set this register to 1 if data has fetched from memory and data length is equal to Buffer Length(640*480/4 DW)
m2s Buffer transfer done status clear register	0x78	bit 0 – clear m2s buffer transfer done status (Read/Write) Set 1 to clear m2s buffer done status/s2m error status and reset internal state, then set 0 if finish to clear buffer done status Note: Before set this register to 1 to clear m2s buffer transfer done status/ m2s error status, Clear status control register must set to 1. After buffer transfer done status is clear, this register needs to be cleared for next operation.
m2s Buffer Length	0x80	Set m2s buffer length, must set to 640*480/4.
m2s Clear Status Control register	0x88	bit 0 –Enable to clear m2s buffer transfer done status (Read/Write)

2.2.1. Define the Length of Transmission

In the initial design, the length of data transmission was determined using a hardware constant (e.g. BUF_LEN). However, if we want to change the size of the data length, it will require altering the hardware design. To improve this aspect, I use the user input data length as looping boundary:



```
do {
    count = in_counts.read();

    for (int i = 0; i < count; ++i) {
  #pragma HLS PIPELINE
        in_val = in_stream.read();
        out_memory[i] = in_val.data_filed;
    }
    out_memory += count;
    final_s2m_len += count;

    if(final_s2m_len == in_s2m_len){
        out_sts = 1;
    }

    buf_sts = out_sts;
} while(final_s2m_len < in_s2m_len);
```

Therefore, it can handle different data length without crashing the system.

2.2.2. Error Detection

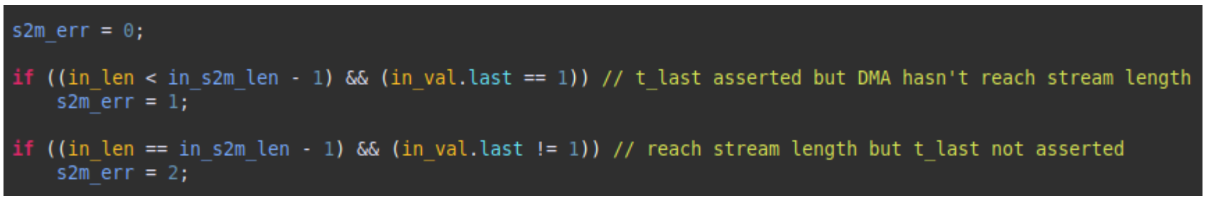
There are two scenarios where a DMA transaction can end:

1. After transmitting the specified amount of data:
    - For stream-in, it does not matter whether `tlast` is asserted.
    - For stream-out, `tlast` must be asserted.
2. For stream-in, when `tlast` is received, regardless of whether the specified transmission amount has been reached.

Handling methods:

- If data is sent beyond what is set by DMA (`tlast` exceeds the set `length`), the excess data is dropped, but this should be recorded in the status so that the software is aware of what occurred.
- If less data is transmitted (i.e. `tlast` is asserted prematurely), the DMA should end normally, but an error flag should be recorded in the status to inform the software.

In the function `GetInStream`:

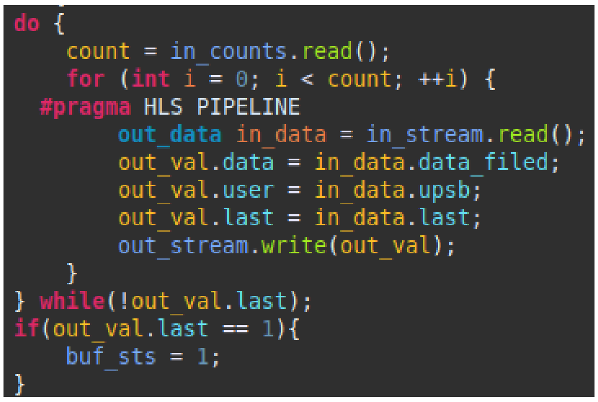


```
s2m_err = 0;

if ((in_len < in_s2m_len - 1) && (in_val.last == 1)) // t_last asserted but DMA hasn't reach stream length
    s2m_err = 1;

if ((in_len == in_s2m_len - 1) && (in_val.last != 1)) // reach stream length but t_last not asserted
    s2m_err = 2;
```

Here I defined 3 statuses:

- 0: Passed
- 1: Stream-in less than Length
- 2: Stream-in more than Length

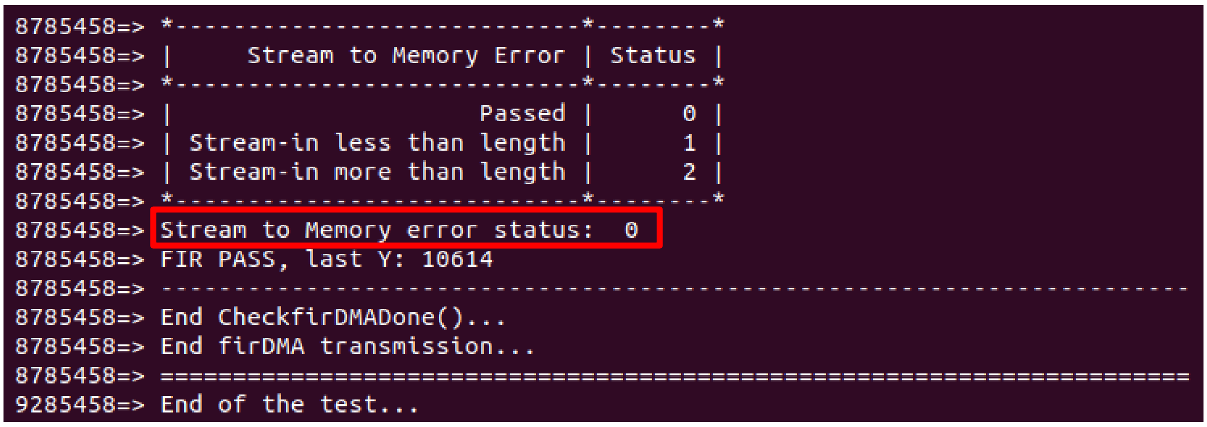
In the function `SendOutStream`:



```
do {
    count = in_counts.read();
    for (int i = 0; i < count; ++i) {
  #pragma HLS PIPELINE
        out_data in_data = in_stream.read();
        out_val.data = in_data.data_filed;
        out_val.user = in_data.upsb;
        out_val.last = in_data.last;
        out_stream.write(out_val);
    }
} while(!out_val.last);
if(out_val.last == 1){
    buf_sts = 1;
}
```

The DMA will assert `tlast` when finished streaming-out.

Simulation result:

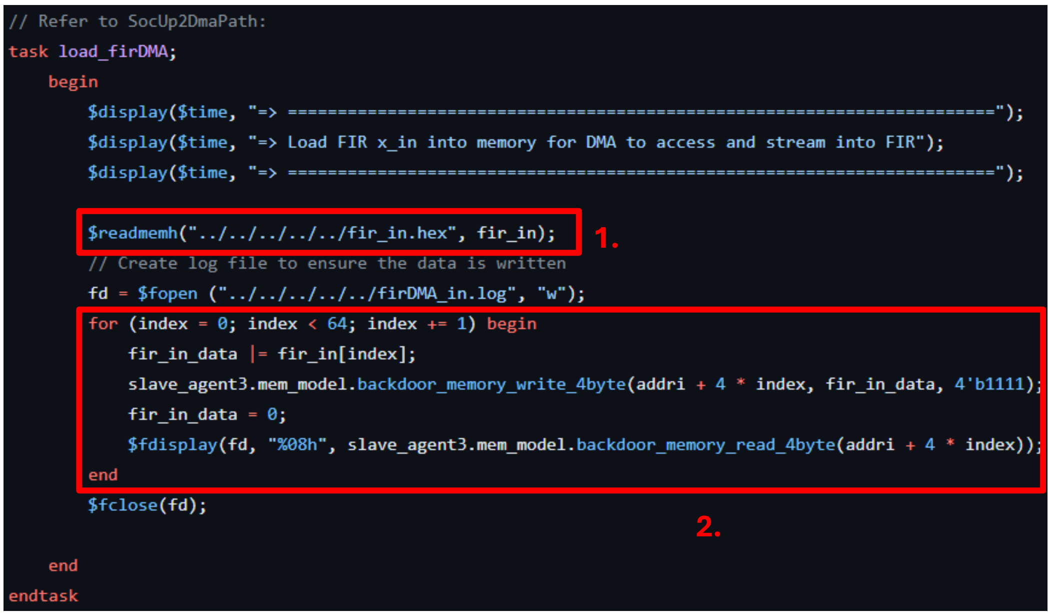


```
8785458=> *----------------------------*--------*
8785458=> |     Stream to Memory Error | Status |
8785458=> *----------------------------*--------*
8785458=> |                     Passed |      0 |
8785458=> | Stream-in less than length |      1 |
8785458=> | Stream-in more than length |      2 |
8785458=> *----------------------------*--------*
8785458=> Stream to Memory error status:  0
8785458=> FIR PASS, last Y: 10614
8785458=> -----------------------------------------------------------------------
8785458=> End CheckfirDMADone()...
8785458=> End firDMA transmission...
8785458=> =======================================================================
9285458=> End of the test...
```

2-2. Testbench

2.2.1. Load Data into Memory

To let DMA retrieve data from memory, we need to load data into memory, following code shows how we implemented it:



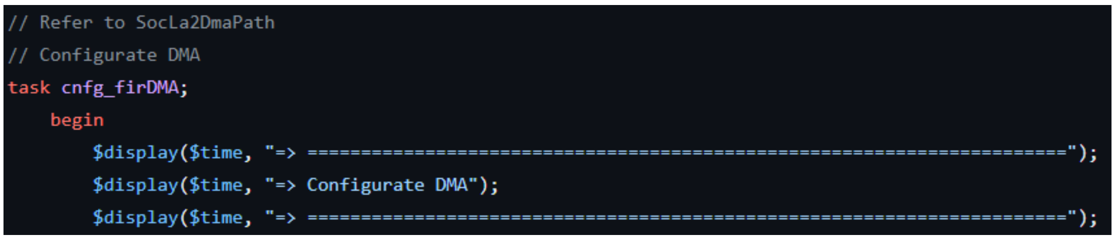
```
// Refer to SocUp2DmaPath:
task load_firDMA;
    begin
        $display($time, "=> ===================================================================");
        $display($time, "=> Load FIR x_in into memory for DMA to access and stream into FIR");
        $display($time, "=> ===================================================================");

        $readmemh("../../../../../fir_in.hex", fir_in);
        // Create log file to ensure the data is written
        fd = $fopen ("../../../../../firDMA_in.log", "w");
        for (index = 0; index < 64; index += 1) begin
            fir_in_data |= fir_in[index];
            slave_agent3.mem_model.backdoor_memory_write_4byte(addri + 4 * index, fir_in_data, 4'b1111);
            fir_in_data = 0;
            $fdisplay(fd, "%08h", slave_agent3.mem_model.backdoor_memory_read_4byte(addri + 4 * index));
        end
        $fclose(fd);

    end
endtask
```

(https://github.com/vic9112/Advance_SOC/blob/main/lab04%20-%20fsic_fpga/vivado/fsic_tb.sv)

1. Load input from .hex file and store into register 'fir_in' which has the length of 64
2. Load input from register 'fir_in' into memory.

2.2.2. DMA Configuration

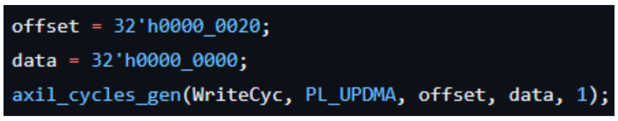
As shown in Table 1, we need to configurate those registers in order to ensure the DMA operates correctly and get some information from them. In the following task:



```
// Refer to SocLa2DmaPath
// Configurate DMA
task cnfg_firDMA;
    begin
        $display($time, "=> ===================================================================");
        $display($time, "=> Configurate DMA");
        $display($time, "=> ===================================================================");
```

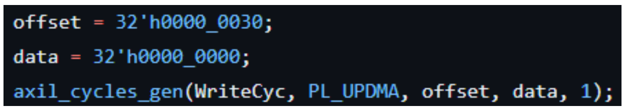
(https://github.com/vic9112/Advance_SOC/blob/main/lab04%20-%20fsic_fpga/vivado/fsic_tb.sv)

We initialized the following:



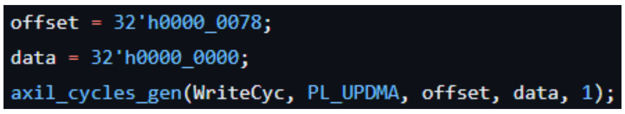
```
offset = 32'h0000_0020;
data = 32'h0000_0000;
axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);
```

- 0x20: s2m buffer transfer done status clear



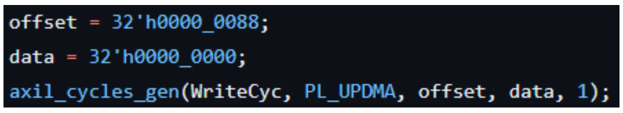
```
offset = 32'h0000_0030;
data = 32'h0000_0000;
axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);
```

- 0x30: s2m clear status control



```
offset = 32'h0000_0078;
data = 32'h0000_0000;
axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);
```

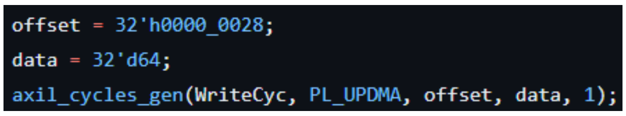
- 0x78: m2s buffer transfer done status clear



```
offset = 32'h0000_0088;
data = 32'h0000_0000;
axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);
```

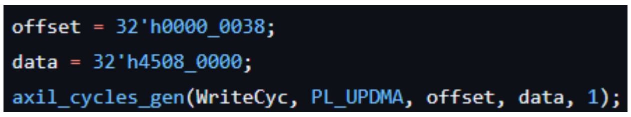
- 0x88: m2s clear status control

And programmed:



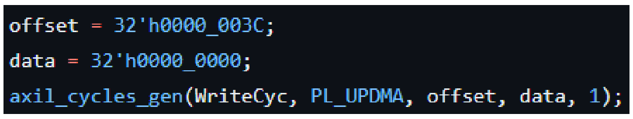
```
offset = 32'h0000_0028;
data = 32'd64;
axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);
```

- 0x28: buffer length => 64



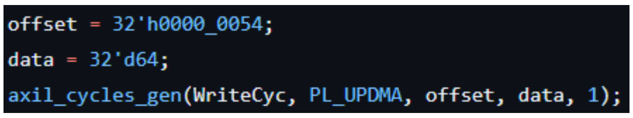
```
offset = 32'h0000_0038;
data = 32'h4508_0000;
axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);
```

- 0x38: s2m buffer lower base address => 0x4508_0000



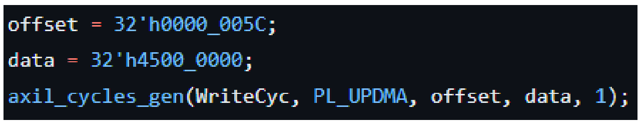
```
offset = 32'h0000_003C;
data = 32'h0000_0000;
axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);
```

- 0x3C: s2m buffer upper base address => 0



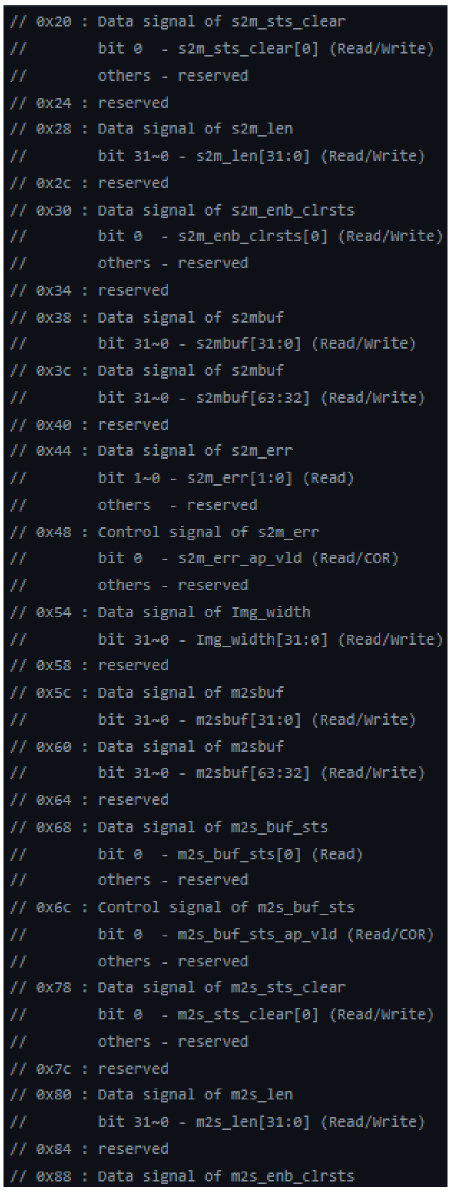
```
offset = 32'h0000_0054;
data = 32'd64;
axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);
```

- 0x54: image width, which will not be used in FIR



```
offset = 32'h0000_005C;
data = 32'h4500_0000;
axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);
```

- 0x5C: m2s buffer lower base address => 0x4500_0000



```
// 0x20 : Data signal of s2m_sts_clear
//        bit 0  - s2m_sts_clear[0] (Read/Write)
//        others - reserved
// 0x24 : reserved
// 0x28 : Data signal of s2m_len
//        bit 31~0 - s2m_len[31:0] (Read/Write)
// 0x2c : reserved
// 0x30 : Data signal of s2m_enb_clrsts
//        bit 0  - s2m_enb_clrsts[0] (Read/Write)
//        others - reserved
// 0x34 : reserved
// 0x38 : Data signal of s2mbuf
//        bit 31~0 - s2mbuf[31:0] (Read/Write)
// 0x3c : Data signal of s2mbuf
//        bit 31~0 - s2mbuf[63:32] (Read/Write)
// 0x40 : reserved
// 0x44 : Data signal of s2m_err
//        bit 1~0 - s2m_err[1:0] (Read)
//        others  - reserved
// 0x48 : Control signal of s2m_err
//        bit 0  - s2m_err_ap_vld (Read/COR)
//        others - reserved
// 0x54 : Data signal of Img_width
//        bit 31~0 - Img_width[31:0] (Read/Write)
// 0x58 : reserved
// 0x5c : Data signal of m2sbuf
//        bit 31~0 - m2sbuf[31:0] (Read/Write)
// 0x60 : Data signal of m2sbuf
//        bit 31~0 - m2sbuf[63:32] (Read/Write)
// 0x64 : reserved
// 0x68 : Data signal of m2s_buf_sts
//        bit 0  - m2s_buf_sts[0] (Read)
//        others - reserved
// 0x6c : Control signal of m2s_buf_sts
//        bit 0  - m2s_buf_sts_ap_vld (Read/COR)
//        others - reserved
// 0x78 : Data signal of m2s_sts_clear
//        bit 0  - m2s_sts_clear[0] (Read/Write)
//        others - reserved
// 0x7c : reserved
// 0x80 : Data signal of m2s_len
//        bit 31~0 - m2s_len[31:0] (Read/Write)
// 0x84 : reserved
// 0x88 : Data signal of m2s_enb_clrsts
```

```
offset = 32'h0000_0060;
data = 32'h0000_0000;
axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);
```

- 0x60: m2s buffer upper base address

```
offset = 32'h0000_0080;
data = 32'd64;
axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);
```

- 0x80: m2s buffer length => 64

2.2.3. FIR Initialization

Before we start streaming data from/to DMA, we need to initialize FIR using AXI-Lite transmissions from FPGA. In the following task:

```
task start_fir;
    int i;
    begin
        $display($time, "=> =======================================================================");
        $display($time, "=> Initialize FIR and program ap_start");
        $display($time, "=> =======================================================================");
```

(https://github.com/vic9112/Advance_SOC/blob/main/lab04%20-%20fsic_fpga/vivado/fsic_tb.sv)

First, we choose User Project 1, where our FIR locate at, as target by configurating CC module:

```
offset = 0;
data = 32'h0000_0001;
axil_cycles_gen(WriteCyc, SOC_CC, offset, data, 1);
```

Then, program data length into FIR:

```
offset = 12'h10;
data = 32'd64;
axil_cycles_gen(WriteCyc, SOC_UP, offset, data, 1);
```

And write coefficients into FIR:

```
for(i = 0; i < 11; i = i + 1) begin
    $display($time, "=> Fpga2Soc_Write: SOC_UP");
    offset = 12'h20 + 4 * i;
    data = coef[i];
    axil_cycles_gen(WriteCyc, SOC_UP, offset, data, 1);
```

Finally, program ap_start:

```
offset = 12'h00;
data = 32'h0000_0001;
axil_cycles_gen(WriteCyc, SOC_UP, offset, data, 1);
```

2.2.4. DMA Start

After the above processes have been set up, we can program DMA's ap_start to start streaming data to/from FIR and read/write data from/into memory.

```
task start_firDMA;
    begin
        $display($time, "=> ==================================================================");
        $display($time, "=> Start DMA streaming x in, program ap_start to DMA");
        $display($time, "=> ==================================================================");
        offset = 32'h0000_0000;
        data   = 32'h0000_0001;
        axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);

        fork
            CheckfirDMADone();
        join_none

        @(userdma_done);

        $display($time, "=> End firDMA transmission...");
        $display($time, "=> ==================================================================");
    end
endtask
```

(https://github.com/vic9112/Advance_SOC/blob/main/lab04%20-%20fsic_fpga/vivado/fsic_tb.sv)

Above start the task `CheckfirDMADone` & wait until the event `userdma_done` raised.

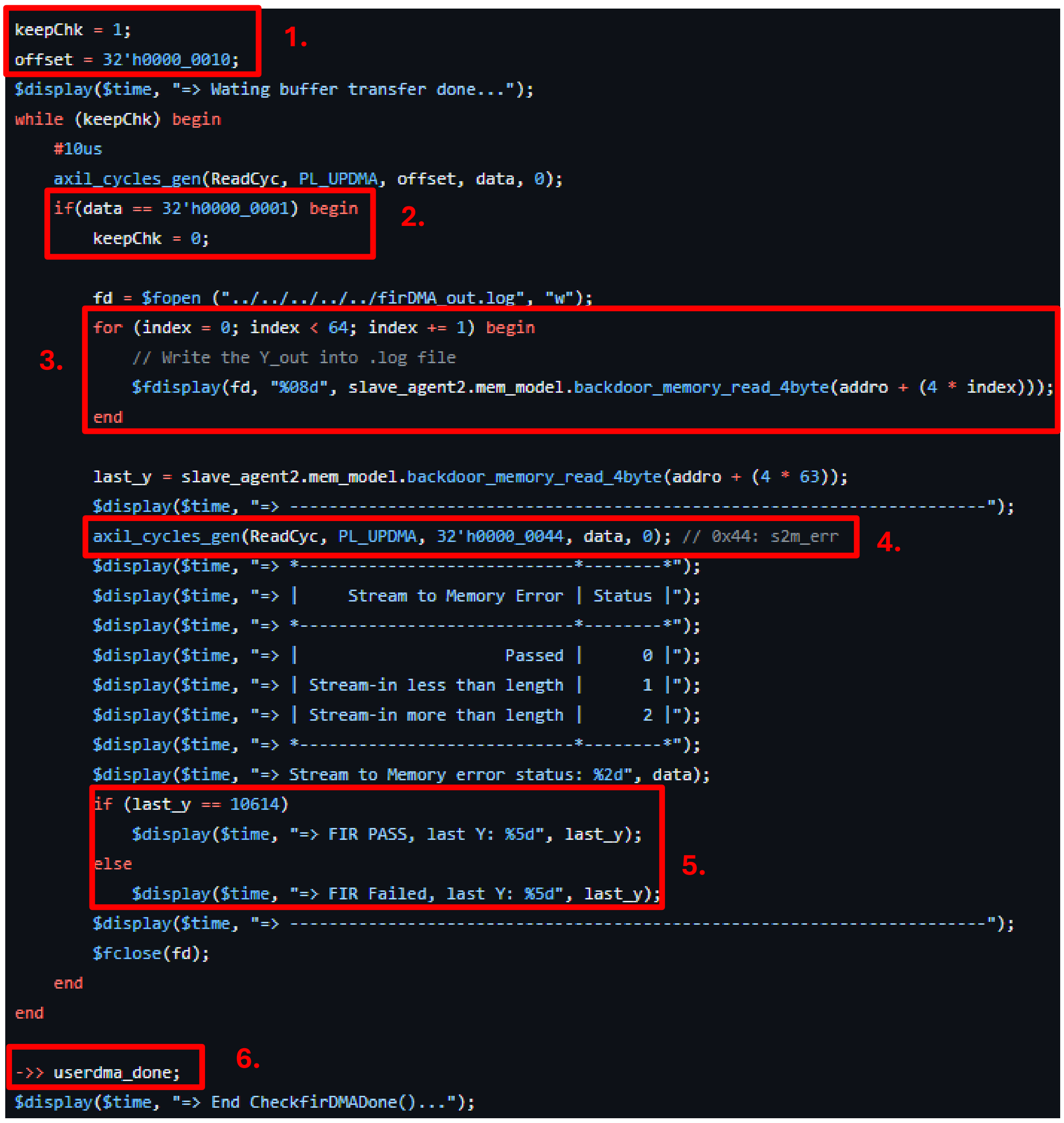
2.2.5. CheckfirDMADone

To check whether firDMA has completed its transactions, we continuously loop and check the address 0x10, which is the configuration address of s2m buffer transfer done status register defined in firDMA (Table 1). In the following task, we write the output from firDMA into a .log file after its transactions has completed. (Notice that addro(0x4508_0000) is the base address of firDMA out)

```
task CheckfirDMADone;
    reg [31:0] last_y;
    begin
        $display($time, "=> ==================================================================");
        $display($time, "=> Starting Check fir result written back by DMA");
        $display($time, "=> ==================================================================");
```

(https://github.com/vic9112/Advance_SOC/blob/main/lab04%20-%20fsic_fpga/vivado/fsic_tb.sv)

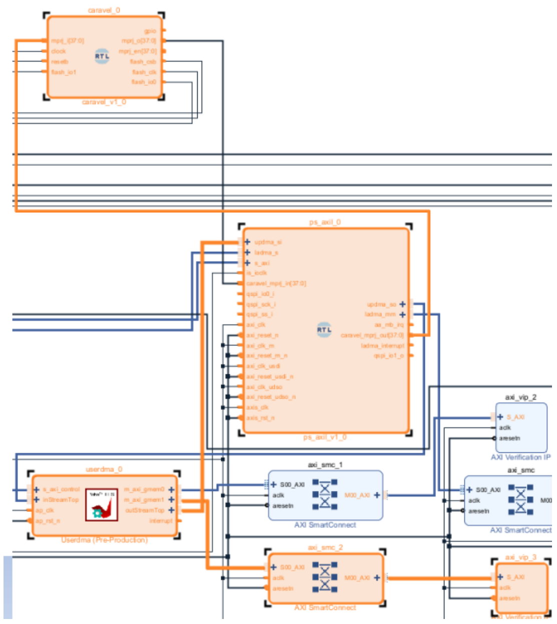
Inside the task:

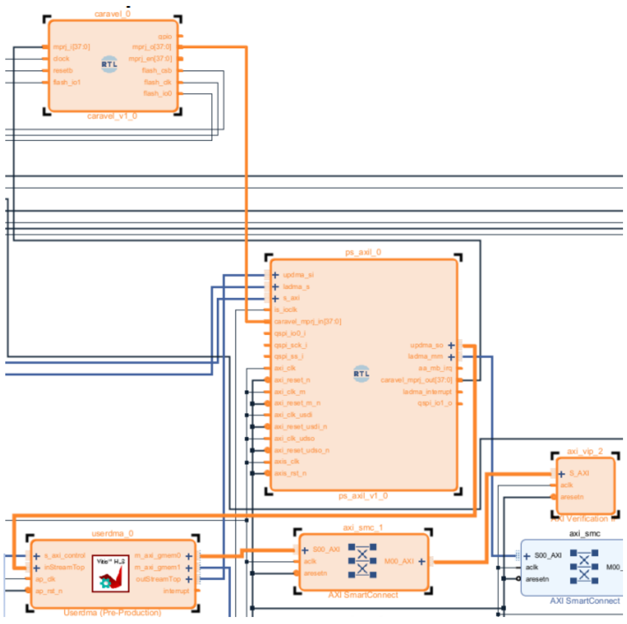


```
keepChk = 1;
offset = 32'h0000_0010;
$display($time, "=> Wating buffer transfer done...");
while (keepChk) begin
    #10us
    axil_cycles_gen(ReadCyc, PL_UPDMA, offset, data, 0);
    if(data == 32'h0000_0001) begin
        keepChk = 0;

        fd = $fopen ("../../../../../firDMA_out.log", "w");
        for (index = 0; index < 64; index += 1) begin
            // Write the Y_out into .log file
            $fdisplay(fd, "%08d", slave_agent2.mem_model.backdoor_memory_read_4byte(addro + (4 * index)));
        end

        last_y = slave_agent2.mem_model.backdoor_memory_read_4byte(addro + (4 * 63));
        $display($time, "=> -----------------------------------------------------------------------");
        axil_cycles_gen(ReadCyc, PL_UPDMA, 32'h0000_0044, data, 0); // 0x44: s2m_err
        $display($time, "=> *---------------------------*--------*");
        $display($time, "=> |     Stream to Memory Error | Status |");
        $display($time, "=> *---------------------------*--------*");
        $display($time, "=> |                     Passed |      0 |");
        $display($time, "=> | Stream-in less than length |      1 |");
        $display($time, "=> | Stream-in more than length |      2 |");
        $display($time, "=> *---------------------------*--------*");
        $display($time, "=> Stream to Memory error status: %2d", data);
        if (last_y == 10614)
            $display($time, "=> FIR PASS, last Y: %5d", last_y);
        else
            $display($time, "=> FIR Failed, last Y: %5d", last_y);
        $display($time, "=> -----------------------------------------------------------------------");
        $fclose(fd);
    end
end

->> userdma_done;
$display($time, "=> End CheckfirDMADone()...");
```

1. Set the flag ‘keepChk = 1’, offset = 0x10: s2m buffer transfer (upstream) done status
2. If s2m buffer transfer done status = 1, stop looping
3. Write all output Y to a .log file
4. Check the error status.
5. Check the correctness of Y (we only check the last Y here since all the answers are coherent)
6. Raise the event ‘userdma_done’

2-3. DMA Traffic in Block Diagram



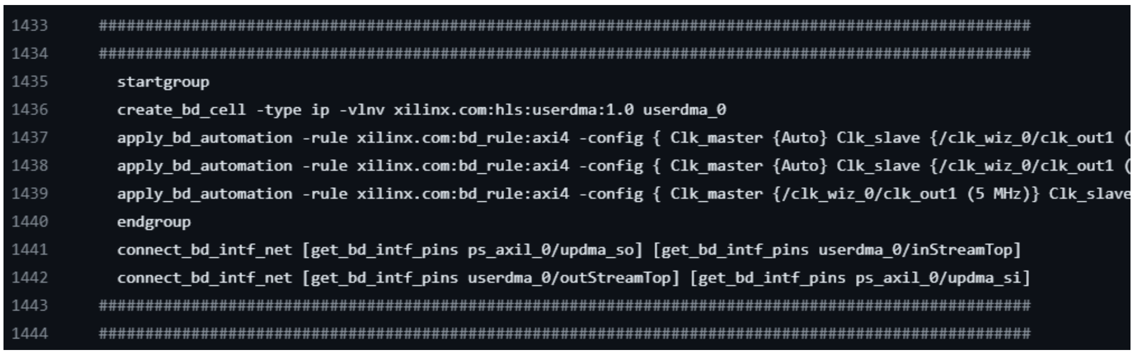


Downstream (FPGA to SoC)　　　　Upstream (SoC to FPGA)

3. Caravel-FSIC Validation

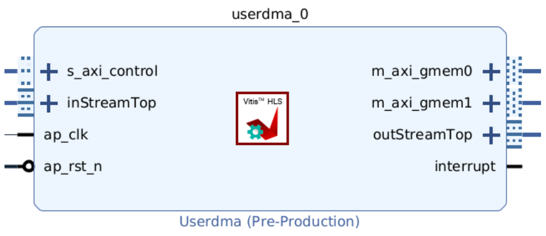
3-1. DMA Connection

I edited the .tcl file (run_vivado_fsic) so that our firDMA can be connected during the block diagram generation phase.

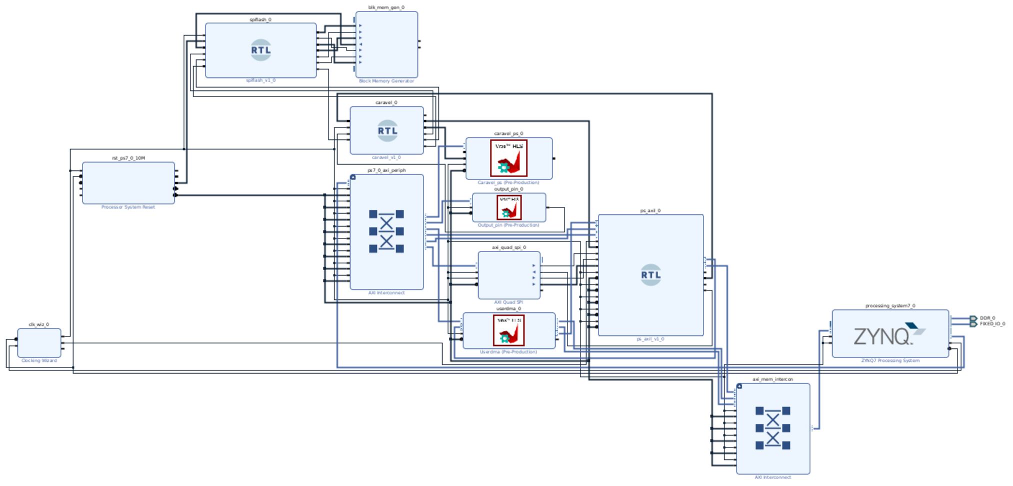


```
###########################################################################################
###########################################################################################
  startgroup
  create_bd_cell -type ip -vlnv xilinx.com:hls:userdma:1.0 userdma_0
  apply_bd_automation -rule xilinx.com:bd_rule:axi4 -config { Clk_master {Auto} Clk_slave {/clk_wiz_0/clk_out1 (
  apply_bd_automation -rule xilinx.com:bd_rule:axi4 -config { Clk_master {Auto} Clk_slave {/clk_wiz_0/clk_out1 (
  apply_bd_automation -rule xilinx.com:bd_rule:axi4 -config { Clk_master {/clk_wiz_0/clk_out1 (5 MHz)} Clk_slave
  endgroup
  connect_bd_intf_net [get_bd_intf_pins ps_axil_0/updma_so] [get_bd_intf_pins userdma_0/inStreamTop]
  connect_bd_intf_net [get_bd_intf_pins userdma_0/outStreamTop] [get_bd_intf_pins ps_axil_0/updma_si]
###########################################################################################
###########################################################################################
```

(https://github.com/vic9112/Advance_SOC/blob/main/lab04%20-%20fsic_fpga/vivado/vvd_caravel_fpga_fsic.tcl)

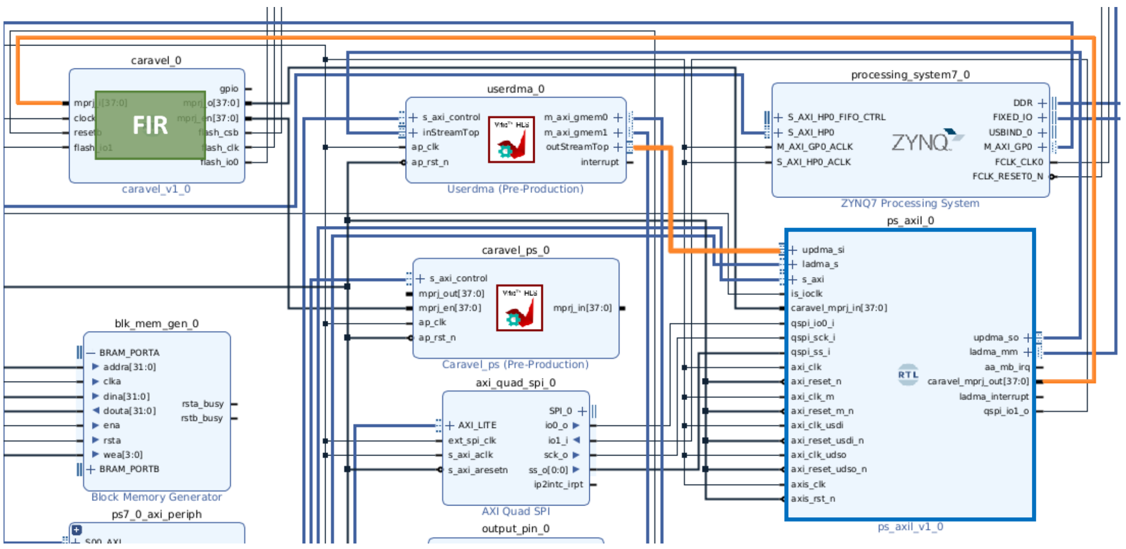


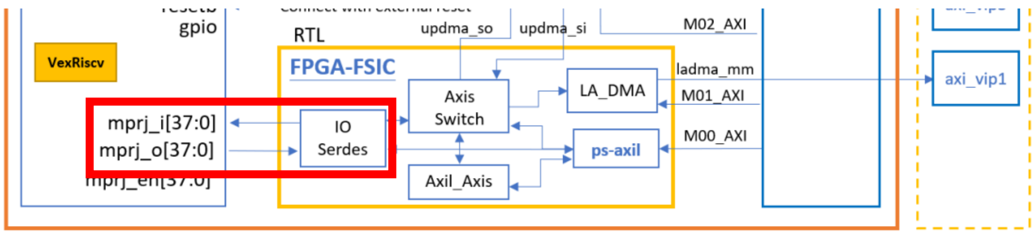
3-2. Block Diagram



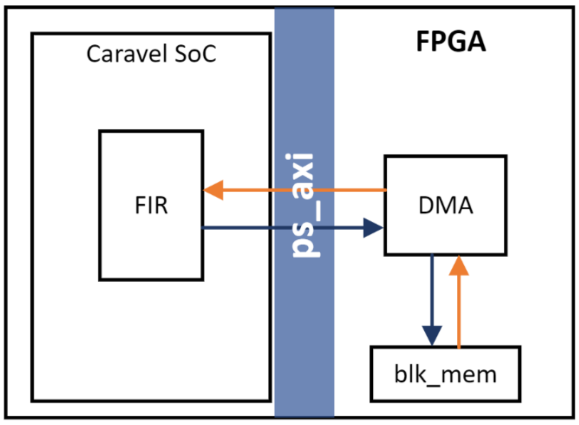
3.2.1. DMA Downstream (FPGA to Caravel SoC)

The orange line in the diagram below shows the path of the DMA streaming x to the FIR located in the Caravel User Project. First, on the DMA interface, we can see 2 `m_axi` ports (m_axi_gmem0, m_axi_gmem1). It retrieves data from memory through the AXI-Master and then stream it out. It goes through the module `**ps_axi**` to forward the DMA request transaction, then it is input into the Caravel SoC via **IO_SERDES** (mprj pin).



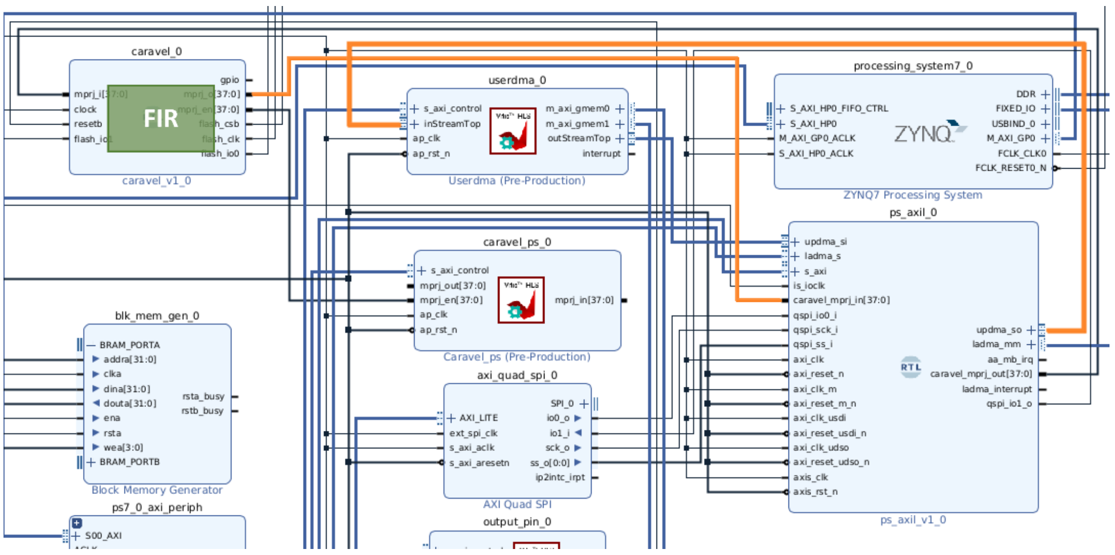


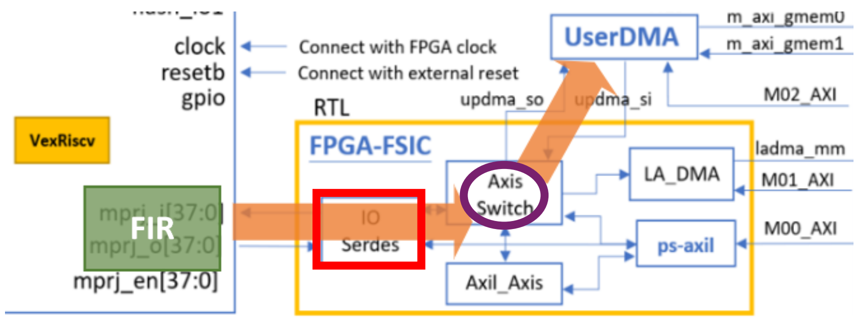
The following figure illustrates the path of the DMA downstream in a simpler way:



3.2.2. DMA Upstream (Caravel SoC to FPGA)

The orange line in the diagram below shows the path of the DMA get the output Y streaming from the FIR located in the Caravel User Project. After FIR completes its calculations, it will upstream the output Y through the **IO_SERDES** to the FSIC at the FPGA side, where it is transmitted to the DMA in a streaming manner via the **AXIS_SWITCH**



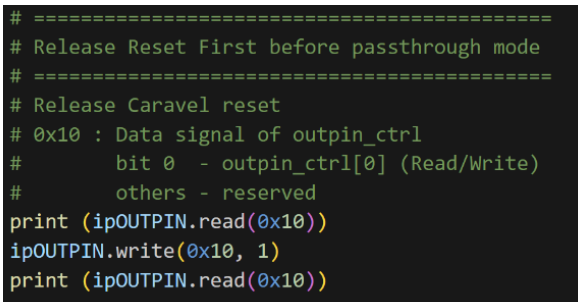


3-3. Simulation on Board

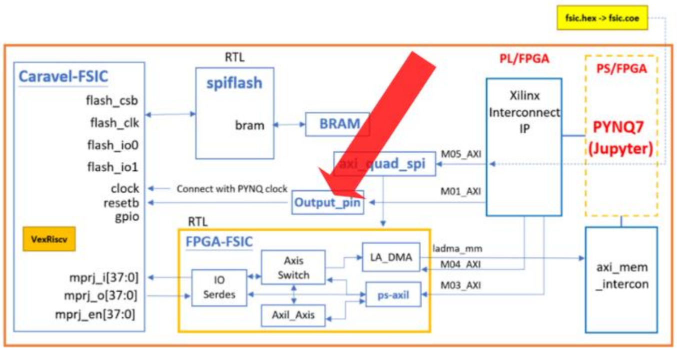
3.3.1. Load firmware into SPIROM

- ➩ Release reset for passthrough.
- ➩ Load firmware (fsic.hex) to memory npROM
- ➩ Enabling passthrough mode
- ➩ Load firmware (fsic.hex) to memory npROM
- ➩ Writing firmware into SPIROM
- ➩ Exit passthrough mode, FW will be fetched.

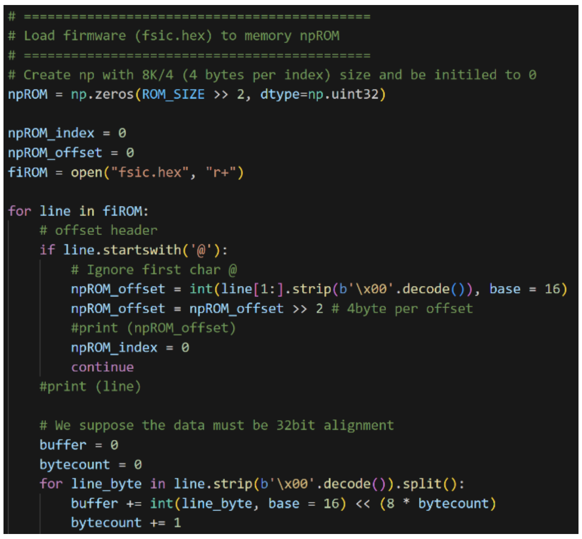
Release reset for passthrough. (output_pin)



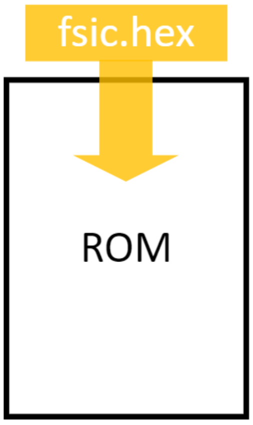
```
# ==========================================
# Release Reset First before passthrough mode
# ==========================================
# Release Caravel reset
# 0x10 : Data signal of outpin_ctrl
#        bit 0  - outpin_ctrl[0] (Read/Write)
#        others - reserved
print (ipOUTPIN.read(0x10))
ipOUTPIN.write(0x10, 1)
print (ipOUTPIN.read(0x10))
```

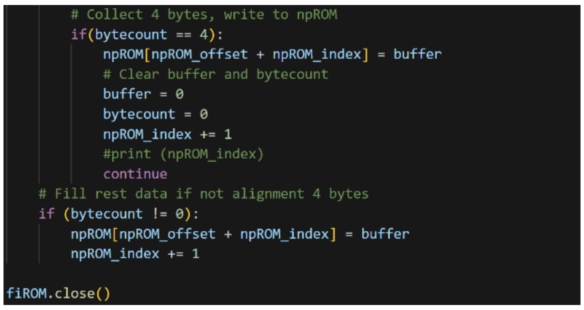


Load firmware to npROM

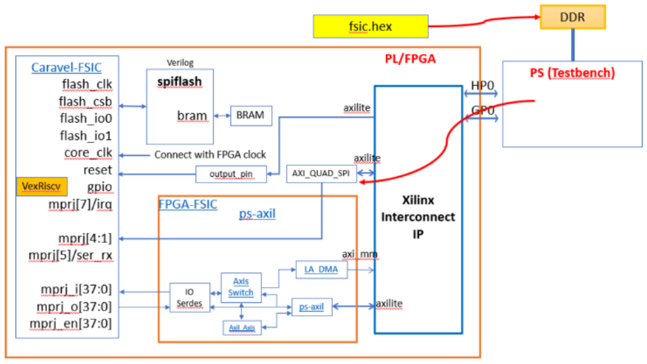


```
# ==========================================
# Load firmware (fsic.hex) to memory npROM
# ==========================================
# Create np with 8K/4 (4 bytes per index) size and be initiled to 0
npROM = np.zeros(ROM_SIZE >> 2, dtype=np.uint32)

npROM_index = 0
npROM_offset = 0
fiROM = open("fsic.hex", "r+")

for line in fiROM:
    # offset header
    if line.startswith('@'):
        # Ignore first char @
        npROM_offset = int(line[1:].strip(b'\x00'.decode()), base = 16)
        npROM_offset = npROM_offset >> 2 # 4byte per offset
        #print (npROM_offset)
        npROM_index = 0
        continue
    #print (line)

    # We suppose the data must be 32bit alignment
    buffer = 0
    bytecount = 0
    for line_byte in line.strip(b'\x00'.decode()).split():
        buffer += int(line_byte, base = 16) << (8 * bytecount)
        bytecount += 1
```

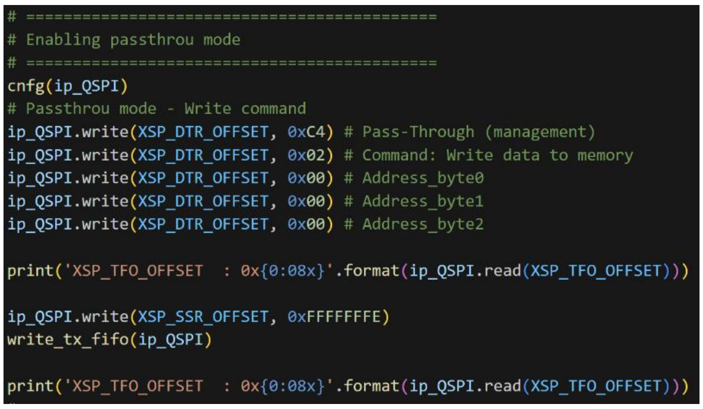




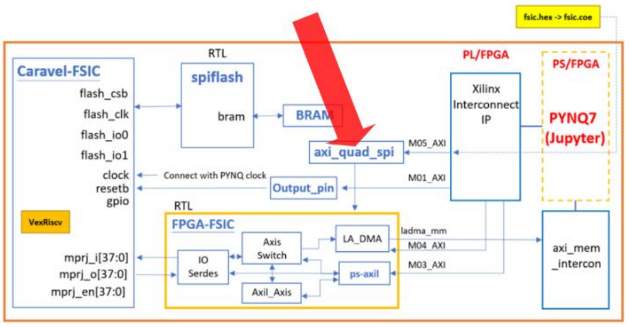
```
        # Collect 4 bytes, write to npROM
        if(bytecount == 4):
            npROM[npROM_offset + npROM_index] = buffer
            # Clear buffer and bytecount
            buffer = 0
            bytecount = 0
            npROM_index += 1
            #print (npROM_index)
            continue
    # Fill rest data if not alignment 4 bytes
    if (bytecount != 0):
        npROM[npROM_offset + npROM_index] = buffer
        npROM_index += 1

fiROM.close()
```



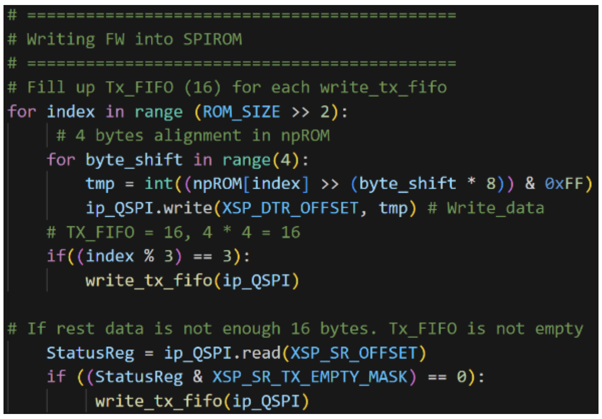
Enabling passthrough mode



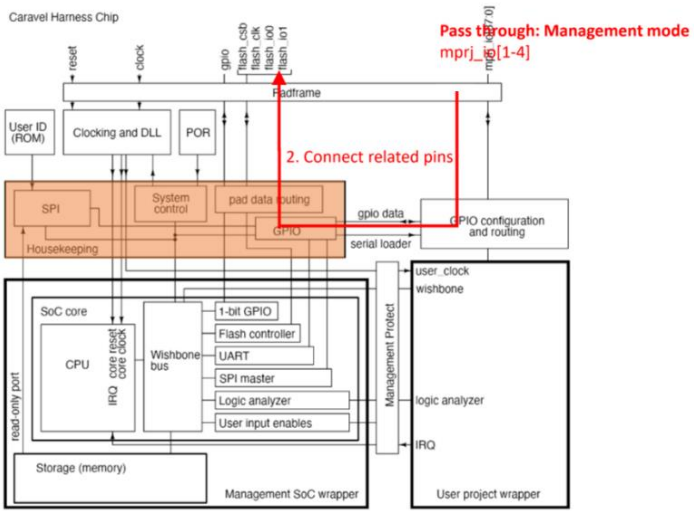
```
# ============================================
# Enabling passthrou mode
# ============================================
cnfg(ip_QSPI)
# Passthrou mode - Write command
ip_QSPI.write(XSP_DTR_OFFSET, 0xC4) # Pass-Through (management)
ip_QSPI.write(XSP_DTR_OFFSET, 0x02) # Command: Write data to memory
ip_QSPI.write(XSP_DTR_OFFSET, 0x00) # Address_byte0
ip_QSPI.write(XSP_DTR_OFFSET, 0x00) # Address_byte1
ip_QSPI.write(XSP_DTR_OFFSET, 0x00) # Address_byte2

print('XSP_TFO_OFFSET  : 0x{0:08x}'.format(ip_QSPI.read(XSP_TFO_OFFSET)))

ip_QSPI.write(XSP_SSR_OFFSET, 0xFFFFFFFE)
write_tx_fifo(ip_QSPI)

print('XSP_TFO_OFFSET  : 0x{0:08x}'.format(ip_QSPI.read(XSP_TFO_OFFSET)))
```



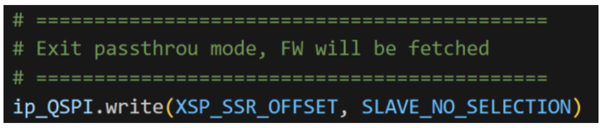
Writing FW into SPIROM



```
# ============================================
# Writing FW into SPIROM
# ============================================
# Fill up Tx_FIFO (16) for each write_tx_fifo
for index in range (ROM_SIZE >> 2):
    # 4 bytes alignment in npROM
    for byte_shift in range(4):
        tmp = int((npROM[index] >> (byte_shift * 8)) & 0xFF)
        ip_QSPI.write(XSP_DTR_OFFSET, tmp) # Write_data
    # TX_FIFO = 16, 4 * 4 = 16
    if((index % 3) == 3):
        write_tx_fifo(ip_QSPI)

# If rest data is not enough 16 bytes. Tx_FIFO is not empty
    StatusReg = ip_QSPI.read(XSP_SR_OFFSET)
    if ((StatusReg & XSP_SR_TX_EMPTY_MASK) == 0):
        write_tx_fifo(ip_QSPI)
```



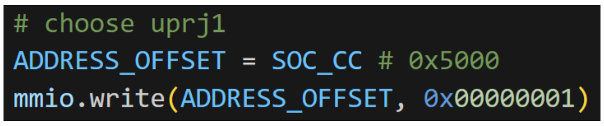
Exit passthrough mode.



```
# ============================================
# Exit passthrou mode, FW will be fetched
# ============================================
ip_QSPI.write(XSP_SSR_OFFSET, SLAVE_NO_SELECTION)
```

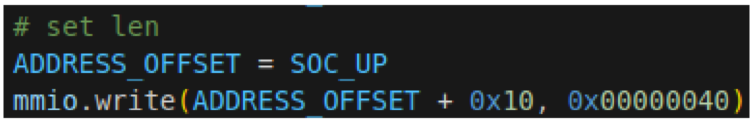
3.3.2. FIR Initialization and DMA Configuration

In FIR Initialization, we first select uprj1, which is the place we put our FIR in.



```
# choose uprj1
ADDRESS_OFFSET = SOC_CC # 0x5000
mmio.write(ADDRESS_OFFSET, 0x00000001)
```

Second, we programmed the length of data 64 into SOC_UP + 0x0010 (0x0010) in hex presentation.



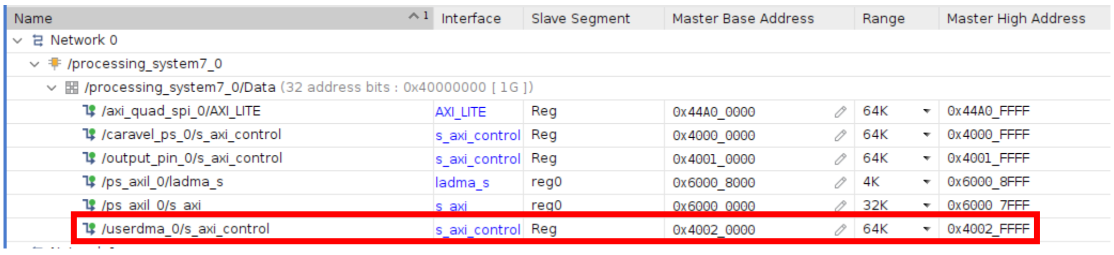
```
# set len
ADDRESS_OFFSET = SOC_UP
mmio.write(ADDRESS_OFFSET + 0x10, 0x00000040)
```

Third, we put 11 coefficients in coef [ ]. Then we write them into 0x20 – 0x60 using mmio.wirte.

```
# set coef
coef = [0x0,0xFFFFFFF6, 0xFFFFFFF7, 0x00000017, 0x00000038, 0x0000003f, 0x00000038, 0x00000017, 0xFFFFFFF7, 0xFFFFFFF6, 0x0]
ADDRESS_OFFSET = SOC_UP
for i in range(11):
    mmio.write(ADDRESS_OFFSET + 0x20 + 4 * i, coef[i])
```

Forth, we started setting DMA. We assigned an MMIO base address with 0x10000 (64k) range.

```
#setting mmio for DMA
firDMA_mmio = MMIO(0x40020000, 0x10000)
```



Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
/processing_system7_0					
/processing_system7_0/Data (32 address bits : 0x40000000 [1G])					
/axi_quad_spi_0/AXI_LITE	AXI_LITE	Reg	0x44A0_0000	64K	0x44A0_FFFF
/caravel_ps_0/s_axi_control	s_axi_control	Reg	0x4000_0000	64K	0x4000_FFFF
/output_pin_0/s_axi_control	s_axi_control	Reg	0x4001_0000	64K	0x4001_FFFF
/ps_axil_0/ladma_s	ladma_s	reg0	0x6000_8000	4K	0x6000_8FFF
/ps_axil_0/s_axi	s_axi	reg0	0x6000_0000	32K	0x6000_7FFF
/userdma_0/s_axi_control	s_axi_control	Reg	0x4002_0000	64K	0x4002_FFFF

Then, set the buffer length 64 (0x40) to address 0x28 (s2m set buffer length), and other registers.

```
firDMA_mmio.write(0x20, 0x00000000) # s2m exit clear
firDMA_mmio.write(0x30, 0x00000000) # s2m disable to clear
firDMA_mmio.write(0x78, 0x00000000) # m2s exit clear
firDMA_mmio.write(0x88, 0x00000000) # m2s disable to clear
firDMA_mmio.write(0x28, 0x00000040) # s2m set buffer len
```

Allocate the buffer for output (size = 1024, datatype = 32-bits integer)

```
firDMA_buf_o = allocate(shape=(1024,), dtype=np.int32)

firDMA_mmio.write(0x38, firDMA_buf_o.device_address)  # output buffer addr low
firDMA_mmio.write(0x3c, 0x00000000) # output buffer addr high
firDMA_mmio.write(0x54, 0x00000040) # width
```

Allocate the buffer for input (size = 1024, datatype = 32-bits integer)

```
firDMA_buf_i = allocate(shape=(1024,), dtype=np.int32)

firDMA_mmio.write(0x5c,firDMA_buf_i.device_address)  # input buffer addr low
firDMA_mmio.write(0x60, 0x00000000) # input buffer addr high
firDMA_mmio.write(0x80, 0x00000040) # m2s set buffer len
```

Data in

```
for i in range(64):
    firDMA_buf_i[i] = i
```

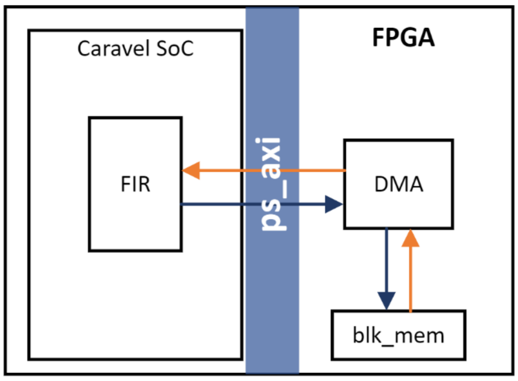
Python Validation result:

```
Check MPRJ_IO input/out/en
0x10 =  0x0
0x14 =  0x0
0x1c =  0x0
0x20 =  0x0
0x34 =  0xfffffff7
0x38 =  0x3f
===========================================
Release Reset First before passthrough mode
===========================================
0
1
===========================================
Load firmware (fsic.hex) to memory npROM
===========================================
Finish loading firmware into npROM
===========================================
===========================================
Enabling passthrough mode
===========================================
Configure device
XSP_TFO_OFFSET  : 0x00000004
XSP_TFO_OFFSET  : 0x00000000
Finish enabling passthrou mode
===========================================
===========================================
Exit passthrou mode, FW will be fetched
===========================================
Check MPRJ_IO input/out/en
0x10 =  0x0
0x14 =  0x0
0x1c =  0x0
0x20 =  0x0
0x34 =  0x3ffff6
0x38 =  0x10
==========================================
DMA AND FIR START
==========================================
mmio.read(PL_IS):  0x0
mmio.read(PL_IS):  0x1
mmio.read(PL_IS):  0x3
```

3.3.4. DMA Start

Program ap_start to start the FIR

```
# fir ap_start
ADDRESS_OFFSET = SOC_UP
mmio.write(ADDRESS_OFFSET, 0x00000001)
```



Program ap_start to start the DMA

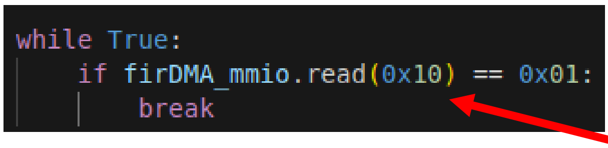
```
# dma_start
firDMA_mmio.write(0x00,0x00000001)
```

Python validation

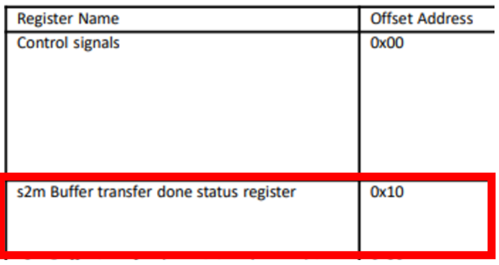
```
==========================================
DMA AND FIR START
==========================================
mmio.read(PL_IS):  0x0
mmio.read(PL_IS):  0x1
mmio.read(PL_IS):  0x3
```

3.3.5. Result Checking

Using while loop to wait for firDMA transfer done:

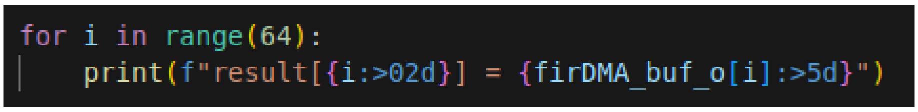


```python
while True:
    if firDMA_mmio.read(0x10) == 0x01:
        break
```



Register Name	Offset Address
Control signals	0x00
s2m Buffer transfer done status register	0x10

Print out FIR calculation result:



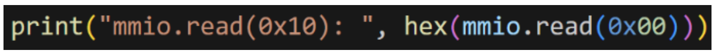
```python
for i in range(64):
    print(f"result[{i:>02d}] = {firDMA_buf_o[i]:>5d}")
```

Result printed on Python

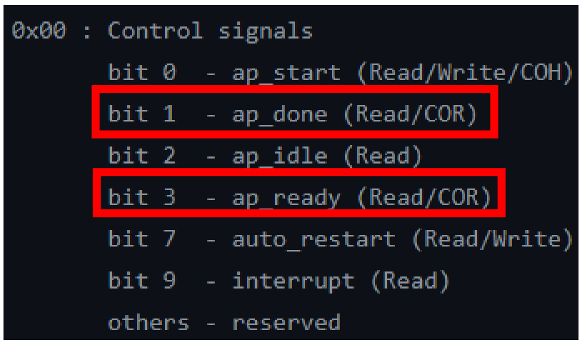
```
result[00] =     0          result[32] =  4941
result[01] =     0          result[33] =  5124
result[02] =   -10          result[34] =  5307
result[03] =   -29          result[35] =  5490
result[04] =   -25          result[36] =  5673
result[05] =    35          result[37] =  5856
result[06] =   158          result[38] =  6039
result[07] =   337          result[39] =  6222
result[08] =   539          result[40] =  6405
result[09] =   732          result[41] =  6588
result[10] =   915          result[42] =  6771
result[11] =  1098          result[43] =  6954
result[12] =  1281          result[44] =  7137
result[13] =  1464          result[45] =  7320
result[14] =  1647          result[46] =  7503
result[15] =  1830          result[47] =  7686
result[16] =  2013          result[48] =  7869
result[17] =  2196          result[49] =  8052
result[18] =  2379          result[50] =  8235
result[19] =  2562          result[51] =  8418
result[20] =  2745          result[52] =  8601
result[21] =  2928          result[53] =  8784
result[22] =  3111          result[54] =  8967
result[23] =  3294          result[55] =  9150
result[24] =  3477          result[56] =  9333
result[25] =  3660          result[57] =  9516
result[26] =  3843          result[58] =  9699
result[27] =  4026          result[59] =  9882
result[28] =  4209          result[60] = 10065
result[29] =  4392          result[61] = 10248
result[30] =  4575          result[62] = 10431
result[31] =  4758          result[63] = 10614
```

Check FIR Done

The value read from 0x10 should be 0x6 (1010: ap_ready, ap_done)



```
print("mmio.read(0x10): ", hex(mmio.read(0x00)))
```



```
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/COR)
       bit 7  - auto_restart (Read/Write)
       bit 9  - interrupt (Read)
       others - reserved
```

Python validation result:

```
mmio.read(0x10):  0x6
```

Indeed it's 6!

ap_done: DMA has finished its task.

ap_ready: DMA is ready for next task (ap_start).

Validation Finish

```
============================================
ON BOARD VALIDATION FINISH!
============================================
```