FIR Workbook (lab_3)

SOC Design

Revise: 7-8-2024

- Elaborate more on testbench function
- Write one to clear ap_done

Function specification

- Same as course-lab_2(FIRN11stream)
- $y[t] = \Sigma (h[i] * x[t - i])$

Design specification

- Data_Width 32
- Tape_Num 11
- Data_Num TBD – Based on size of data file
- Interface
  - data_in stream （ Xn ）
  - data_out: stream (Yn)
  - coef[Tape_Num-1:0] axilite
  - len: axilite
  - ap_start: axilite
  - ap_done: axilite
- Using one Multiplier and one Adder
- Shift register implemented with SRAM (Shift_RAM, size = 10 DW) – size = 10 DW
- Tap coefficient implemented with SRAM (Tap_RAM = 11 DW) and initialized by axilite write
- Operation
  - ap_start to initiate FIR engine (ap_start valid for one clock cycle)
  - Stream-in Xn. The rate is depending on the FIR processing speed. Use axi-stream valid/ready for flow control
  - Stream out Yn, the output rate depends on FIR processing speed.

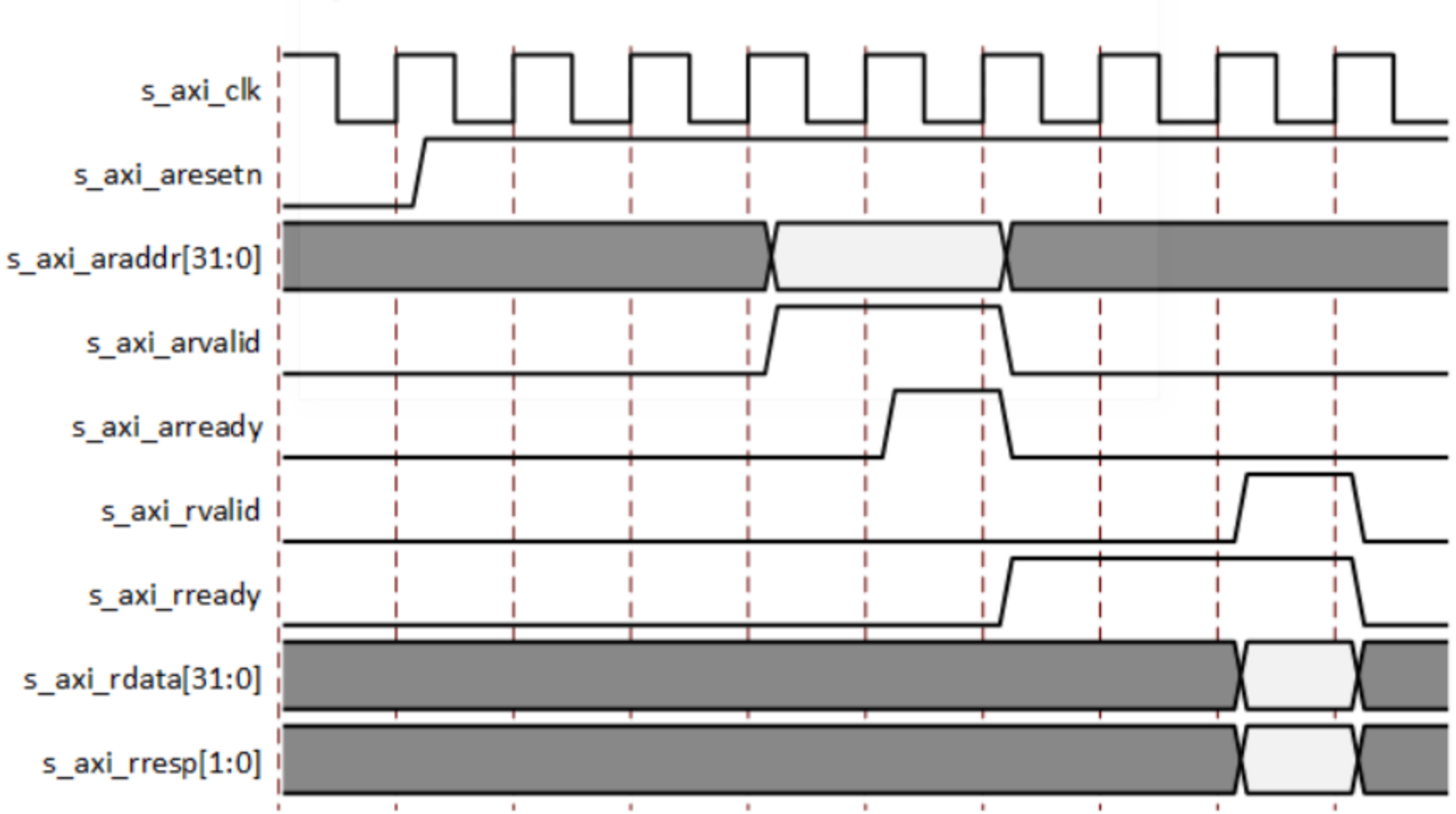
You will implement
- **fir.v**
- **fir_tb.v** (testbench – you can reference and modify from Github fir_tb.v)

Lab Github:
https://github.com/bol-edu/caravel-soc_fpga-lab/tree/main/lab-fir

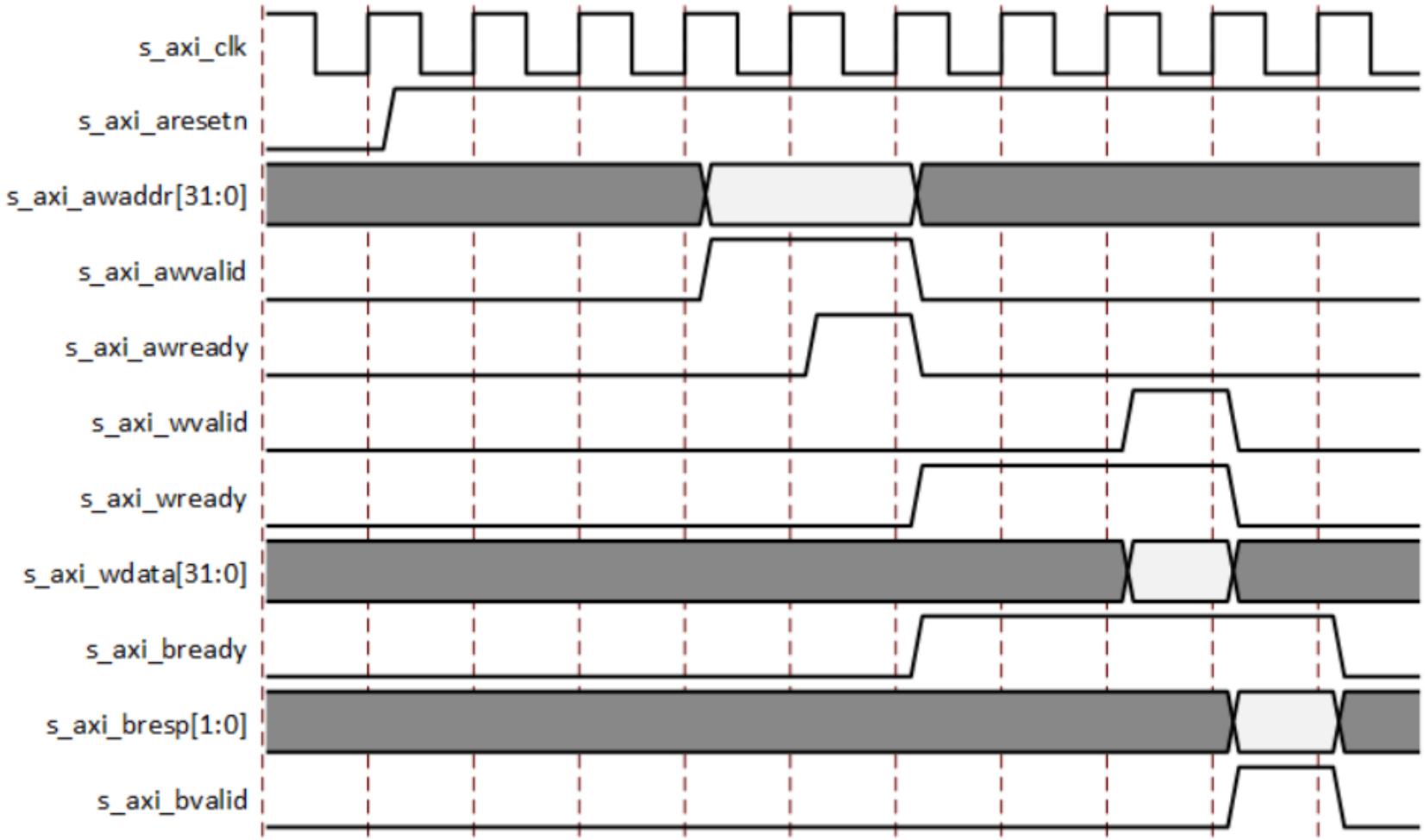
FIR module interface (AXI-Lite, AXI-Stream)

- AXI-lite:
  - https://www.realdigital.org/doc/a9fee931f7a172423e1ba73f66ca4081
  - https://docs.xilinx.com/r/en-US/pg202-mipi-dphy/AXI4-Lite-Interface
- AXI-stream:
  - https://developer.arm.com/documentation/ihi0051/latest/
  - https://docs.xilinx.com/r/en-US/pg256-sdfec-integrated-block/AXI4-Stream-Interface
- BRAM Interface: Synchronous read/write

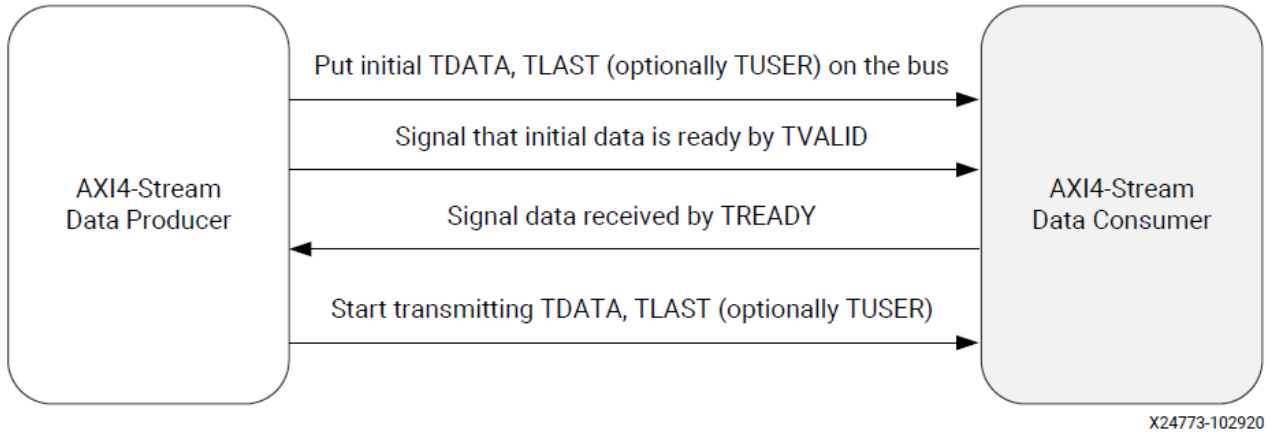
AXI4-Lite Read Transaction

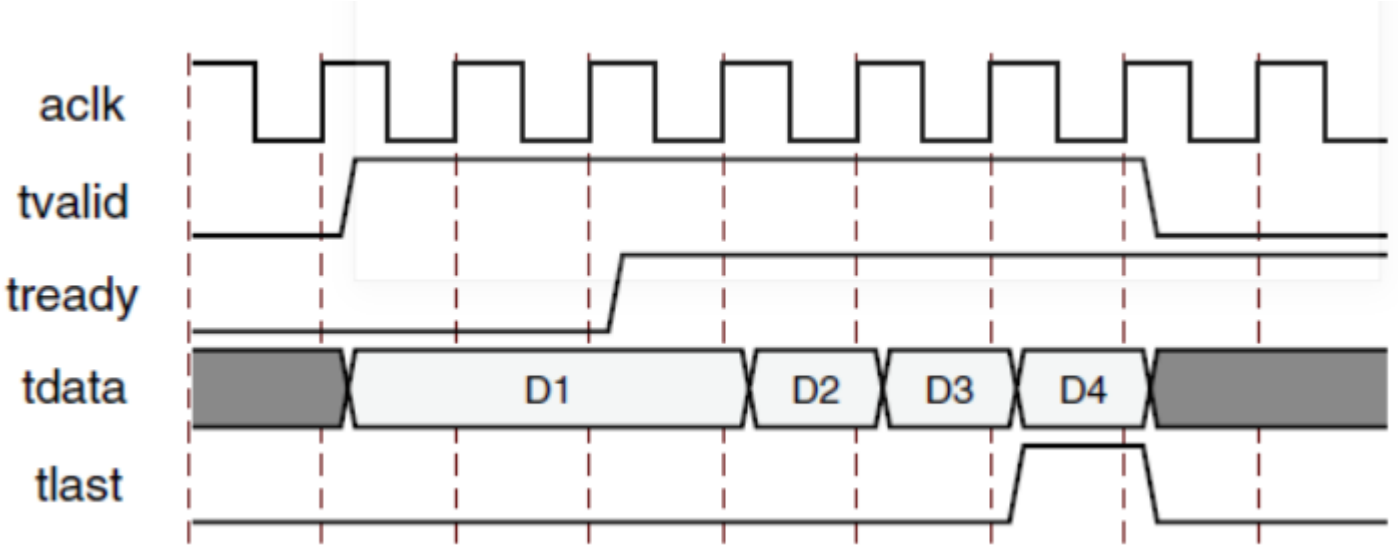


AXI4-Lite Write Transaction



AXI4-Stream Transfer Protocol

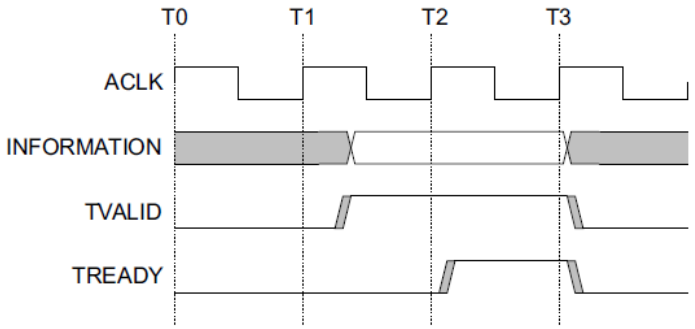




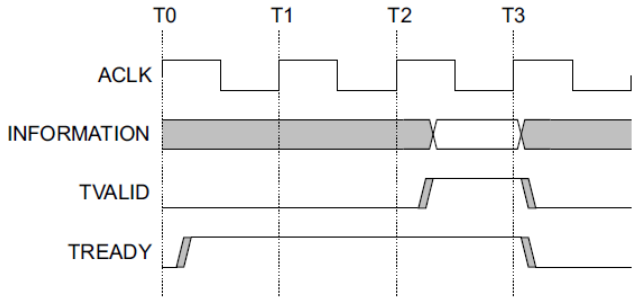
Data Transfer Handshake : TVALID, TREADY

- For a transfer to occur, both **TVALID** and **TREADY** must be asserted
- A Transmitter is not permitted to wait until **TREADY** is asserted before asserting **TVALID**
- Once **TVALID** is asserted, it must remain asserted until the handshake occurs
- A Receiver is permitted to wait for **TVALID** to be asserted before asserting **TREADY**

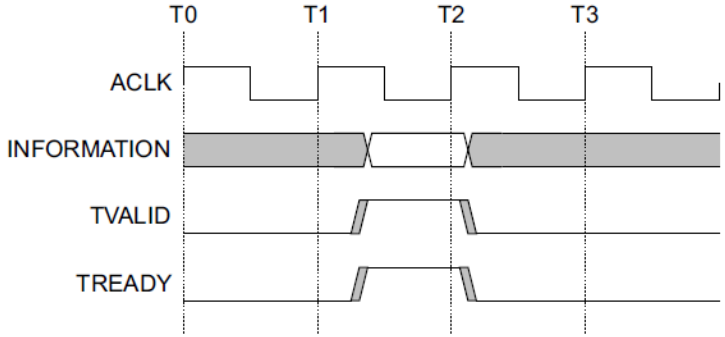
TVALID asserted before TREADY



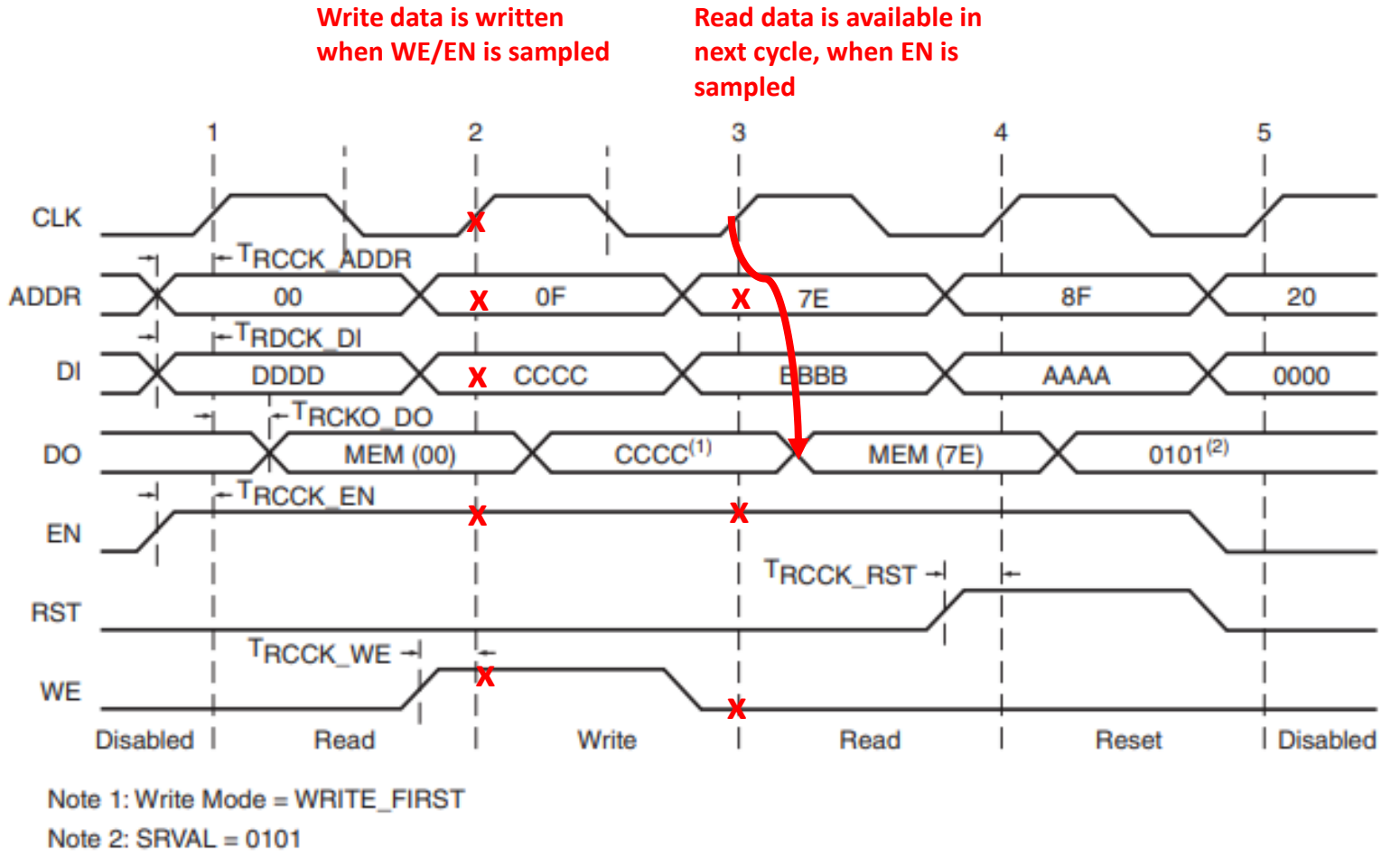
TREADY asserted before TVALID



TVALID and TREADY asserted simultaneously



SRAM Access Timing



SRAM access has different modes, refer to https://docs.xilinx.com/r/en-US/am007-versal-memory/Read-Operation?tocId=VRYu0HURA1U147fufYDMNQ

Deliver module header

- The I/O signals are listed in fir.v.
- You are requested to use simplified AXI-lite and AXI-stream protocol.

Configuration Register Access Protocol

Configuration Register Address map

Address

0x00 – [0] - ap_start (r/w)

set when ap_start signal assert

reset, when start data transfer, i.e. 1st axi-stream data come in

[1] – ap_done (ro) -> when FIR process all the dataset, i.e. receive tlast, and last Y is generated and transferred

[2] – ap_idle (ro) -> indicate FIR is actively processing data

0x10-14 - data-length

0x20-FF – Tap parameters, (e.g., 0x20-23 Tap0, 0x24-0x27 Tap1 .. in sequence …)

Note: Tap parameters set at 0x80 can use lower address bit directly for SRAM address.

ap_start protocol and implementation

1. ap_start is a read/write registers
2. When ap_start is programmed one, the FIR engine starts.
3. Host Software or testbench can program ap_start
    1. When ap_idle is one.
    2. After data-length, tap parameters are programmed
    3. If ap_start is programmed one when ap_idle is zero (i.e. engine is running), the ap_start is not effective
4. ap_start is set by software/testbench, and reset by engine
5. Engine resets ap_start when engine is not idle, i.e. engine starts processing data

ap_done protocol and implementation

1. It is read/write-one-clear register
2. ap_done is reset in the following condition
    1. Reset signal is asserted
    2. After a task complete, the ap_done is cleared by
        1. When ap_done is read, i.e. address 0 is read
        2. Write one to ap_done register bit to clear

        => Choose one implementation
3. ap_done is asserted when engine completes last data processing and data is transferred

ap_idle protocol and implementation

1. ap_idle is set to 1 when reset
2. ap_idle is set to 0 when ap_start is sampled
3. ap_idle is set to 1 when FIR engine processes the last data and last data is transferred

Handle Configuration read/write while engine is active

- It is illegal operation, but we still need to handle it.
- Cfg read: TapRAM read return ‘hffffffff (invalid value, software can check), other address, return valid value
- Cfg write: ignore, illegal action

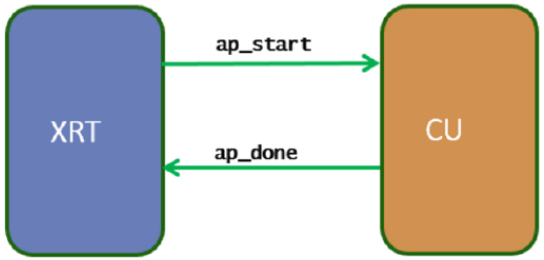
Testbench Specification

Testbench

- fir_tb(Please name your top module same as **fir** for simulation)
- The testbench should keep in the same directory with Makefile
- Block-level Protocol (ap_start, ap_done):

AP_CTRL_HS (Sequential Executed Kernel)

- Host and Kernel Synchronization by
  - ap_start
  - ap_done
- Kernel can only be restarted (ap_start), after it completes the current execution (ap_done)
- Serving one execution request a time



Host software / Testbench Programming Sequence

Host Software / Testbench

1. Check FIR is idle, if not, wait until FIR is idle
2. Program length, and tap parameters
3. Program ap_start -> 1
4. Fork
    1. Transmit Xn,
    2. Receive Yn
    3. Polling ap_done
5. When ap_done is sampled, compare Yn with golden data

FIR Engine

Wait for ap_start
Set ap_idle = 0

Process data

If reach data-length, set ap_done

Note: Transmit Xn (stream-in), Receive Yn (stream-out) and Polling ap_done (axilite) are running concurrently. They are using different interface and do not interfere each other

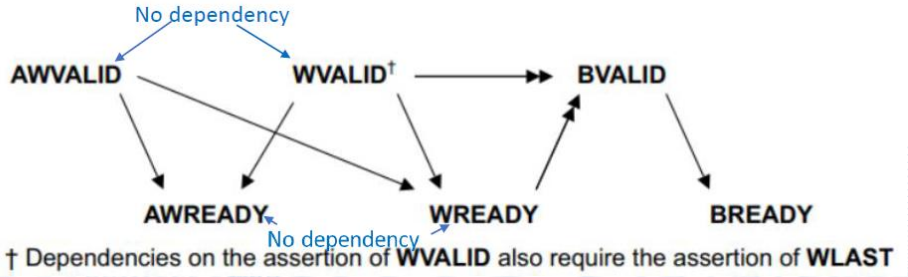
Testbench – Develop your own testbench

1. **Setup phase**
    1. Load datafile, and count # of data = data_length
    2. Program tap_parameters and data_length, read back and check it is correctly programmed
    3. Compute Yn expected value, or load golden data into Yn buffer
    4. Read and check ap_start, ap_idle, ap_done are in proper state
2. **Execution phase**
    1. Program ap_start
    2. Start latency timer
    3. Fork the following operations, run concurrently
        1. Task1(axis-in): Stream_in_Xn
        2. Task2(axis_out): Stream_out_Yn and save into Yn buffer
        3. Task3(axilite):
            1. Polling ap_done, when ap_done is sampled, disable tasks (stream_in_Xn, stream_out_Yn, and Polling)
            2. Read/write tap_parameters: make sure it does not corrupt fir computation
                1. Read return invalid value, e..g. 'hffffffff
                2. Write ignored
3. **Checking Phase**
    1. Report latency
    2. Compare Yn buffer with golden data
4. **Repeat 2 – 3 for three times**

Note: You may print message to assist debugging

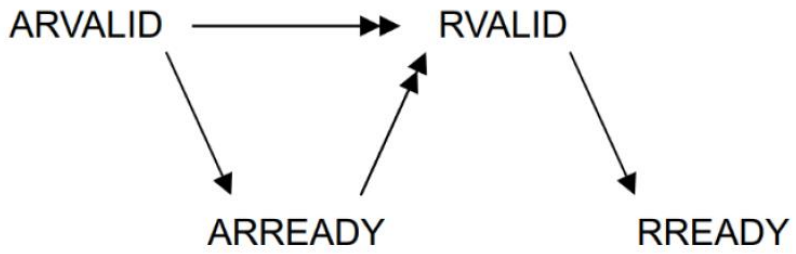
TestBench – Axilite write

- Axilite write
  - Spawn two processes: aw_proc, w_proc
  - Each process has random initial delay 0 – 5T
    - Asserts awaddr (wdata) when awvalid (waddr) asserts
    - Wait for awready (wready)
    - Once awready (wready) is sampled active, invalidate awaddr (wdata)
    - Optionally wait for bvalid (in our case, no bvalid)



TestBench – Axilite read

- Axilite read
  - Spawn two processes: ar_proc, r_proc
  - Each process has random initial delay 0-5T
    - Asserts araddr, and rready
    - Once arready is sampled active, invalidate araddr
  - Check rvalid is only asserted after arvalid and arready are asserted



TestBench – axi-stream X input

Protocol

- For a transfer to occur, both TVALID and TREADY must be asserted
- A Transmitter is not permitted to wait until TREADY is asserted before asserting TVALID
- Once TVALID isasserted , it must remain asserted until the handshake occurs
- A Receiver is permitted to wait for TVALID to be asserted before asserting TREADY

Testbench

- Randomize tvalid initial delay
  - Case1: short latency [0-1]
  - Case2: long latency [0-2 x Filter latency]
- Once sampled tready, deasserts tdata

Testbench: axi-stream Y output

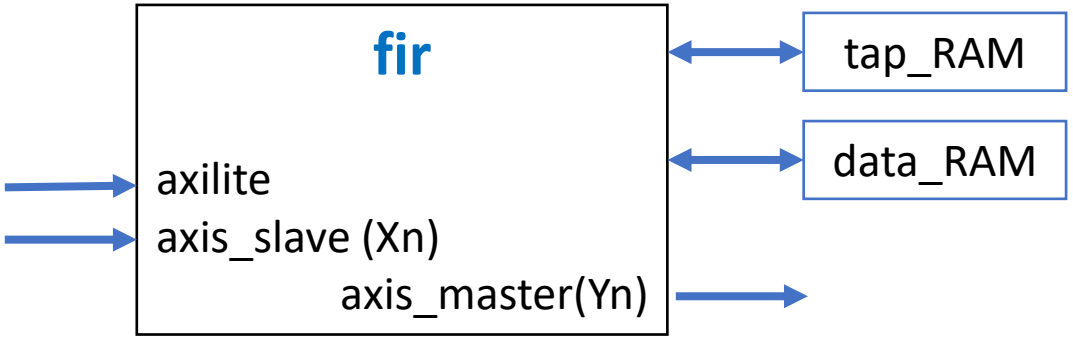
- spawn a task ->task_sm_tready
  - randomize sm_tready delay
    - Short latency: delay [0-5]
    - Long latency: delay [0- 2* filter latency]
  - when both sm_tready and sm_tavlid sampled asserted,
    - Check data
    - finish the task
- Note: sm_tready could be asserted before sm_tvalid asserted

Test dataset

- Samples_triangular_wave.dat
- out_gold.dat

SRAM Interface Implementation

- Refer to verilog-sram.pdf - Use memory for ASIC flow
- Github updated - https://github.com/bol-edu/caravel-soc_fpga-lab/tree/main/lab-fir
- Implement SRAM without .db/.lib
- Use external SRAM (bram.v). fir.v provides ports to interface with the external SRAM. So, the fir.v can be synthesized with BRAM
- Two size of bram.v (you choose either one to fit your design)
  - bram11.v (11 x 32) – depth 11
  - bram12.v (12 x 32) – depth 12



Submission (1/2)

- Hierarchy:
  - StudentID_lab3/
    - Waveform
    - Simulation.log
    - Report.pdf
    - Synthesis report
    - Github link
- Your Github link should attach the file
  - fir.v （the fir design）
  - fir_tb.v (the testbench)
  - Log files including : synthesis, simulation, static timing report
  - Synthesis report – area usage, Including FF, LUT (Note: there should be no BRAM because BRAM is an external model, not in the RTL design)
  - Timing Report, including slack, and max delay path
  - Waveform – show
    - Configuration write
    - ap_start, ap_done (measure # of clock cycles from ap_start to ap_done)
    - Xn stream-in, and Yn stream-out
  - Report
- Location of design (If use vivado to design)
  - hostproject/hostproject.srcs/sources_1/new/

What is included in the report

- Block Diagram
  - Datapath – dataflow
  - Control signals
- Describe operation, e.g.
  - How to receive data-in and tap parameters and place into SRAM
  - How to access shiftram and tapRAM to do computation
  - How ap_done is generated.
  - …..
- Resource usage: including FF, LUT, BRAM
- Timing Report
  - Try to synthesize the design with maximum frequency
  - Report timing on longest path, slack
- Simulation Waveform, show
  - Coefficient program, and read back
  - Data-in stream-in
  - Data-out stream-out
  - RAM access control
  - FSM

Submission (2/2)

- Compress all above files in a single zip file named
  - StudentID_lab3.zip (e.g., 111061545_lab3.zip)
- Submit to NTHU eeclass
- Deadline:

  - 20% off for the late submission penalty within 3 days

Supplement

Use Memory in ASIC Flow

Memory Inference in ASIC

- ASIC Synthesis tool does not infer memories from RTL in the way FPGA synthesis tools do.

- Use Memory Compiler
  - Generate memory block with the specification (bitwidth, depth, # of port)
  - (.lef) – Library Exchange Format – containing placement information
  - Schematic & Netlist (for LVS and functional verification)
  - (.v) Function model (verilog) with timing check – for RTL simulation and gate-level simulation
  - (.lib/.db) – Liberty Timing File – containing Timing delay Dynamic/Static timing analysis, and synthesis
  - (.gds) – Graphical Database System – containing final layout information

- In RTL code, explicitly instantiate the memory, and design its control signals, e.g. Enable, read/write, address, input/output data

Note on ASIC Implementation with SRAM

1. RTL design use memory instance directly. Need to provide sram synthesis library, either .lib, or .db (synospsy design-compiler). There is no particular inference method. (note: Xilinx FPGA can use inference)
2. If there is no sram .lib, or .db, you may put the sram outside using module ports. In this case, you can simulate with post-synthesis gate with sram behavior model. To get sram interface timing optimization, you will need to specify sram interface timing constraints, for example, output delay, input delay.

SRAM with .db/.lib

- RTL Simulation
  - RTL code with instance of SRAM
  - Simulate with functional model
- Synthesis
  - Refer to .lib for SRAM timing/area information
  - Optimize timing for SRAM interface timing
- Post-Synthesis Gate-level Simulation
  - Post layout gate-level timing simulation
  - Use functional model with timing check (specify/endspecify)

```
// RTL module
module your_design( …) begin

// instantiate SRAM
SRAM32X32 (CLK, WE, EN, ADDR, DI, DO)

endmodule
```

```
// Functional Model with timing check
module SRAM32X32( … ) begin

specify    // timing check
endspecify

end
```

```
// Timing model : .lib
```

SRAM without .lib/.db

- SRAM instance could not be in RTL design for synthesis, instead, provide ports to interface with SRAM
- RTL simulation
  - Simulate with SRAM model
- Synthesis
  - Provide timing constrains (e.g. output delay, input delay) for SRAM interface ports
- Post Synthesis with gate-level simulation
  - Simulate with SRAM functional model with timing check

```
// RTL module
module your_design(

// SRAM interface ports
  SRAM_EN,
  SRAM_ADDR,

…) begin

// No SRAM instance
// SRAM32X32 (CLK, WE, EN, ADDR, DI, DO)

endmodule
```

```
// Functional Model with timing check
module SRAM32X32( … ) begin

specify    // timing check
endspecify

end
```

Timing Check Tasks in Verilog

- Specify block can be used to specify setup and hold times for signals
  - specify and endspecify (Use specparam to define parameters in specify block)
- $setup (data, clock edge, limit)–Displays warning message if setup timing constraint is not met
  - $setup(d, posedge clk, 10)
- $hold (clock edge, data, limit)–Displays warning message if hold timing constraint is not met
  - $hold(posedge clk, d, 2)
- $width (pulse event, limit)–Displays warning message if pulse width is shorter than limit
  - $width(posedge clk, 20) –specify start edge of pulse
- $period (pulse event, limit)–Check if period of signal is sufficiently long
  - $period(posedge clk, 50)

Specify Block Example

```
module d_model(
    input      d,
    input      clr,
    input      clk,
    output reg q
    );

   parameter Tclr = 30;
   parameter Trise = 13;
   parameter Tfall = 25;

   specify
      $setup (d, posedge clk, 10);  // check setup time
      $period (posedge clk, 60);
   endspecify

   always @ (posedge clk, posedge clr)
      if (clr)
         #Tclr q <= 1'b0;         // clear delay
      else
         if (d == 1'b1)
            #Trise q <= 1'b1;     // Tplh delay
         else
            #Tfall q <= 1'b0;     // Tphl delay

endmodule
```

```
specify
   specparam Tsetup = 10,  // minimum setup time before clk
      Tperiod = 60;        // minimum clock period
   $setup (d, posedge clk, Tsetup); // check setup time
   $period (posedge clk, Tperiod);  // check period
endspecify
```

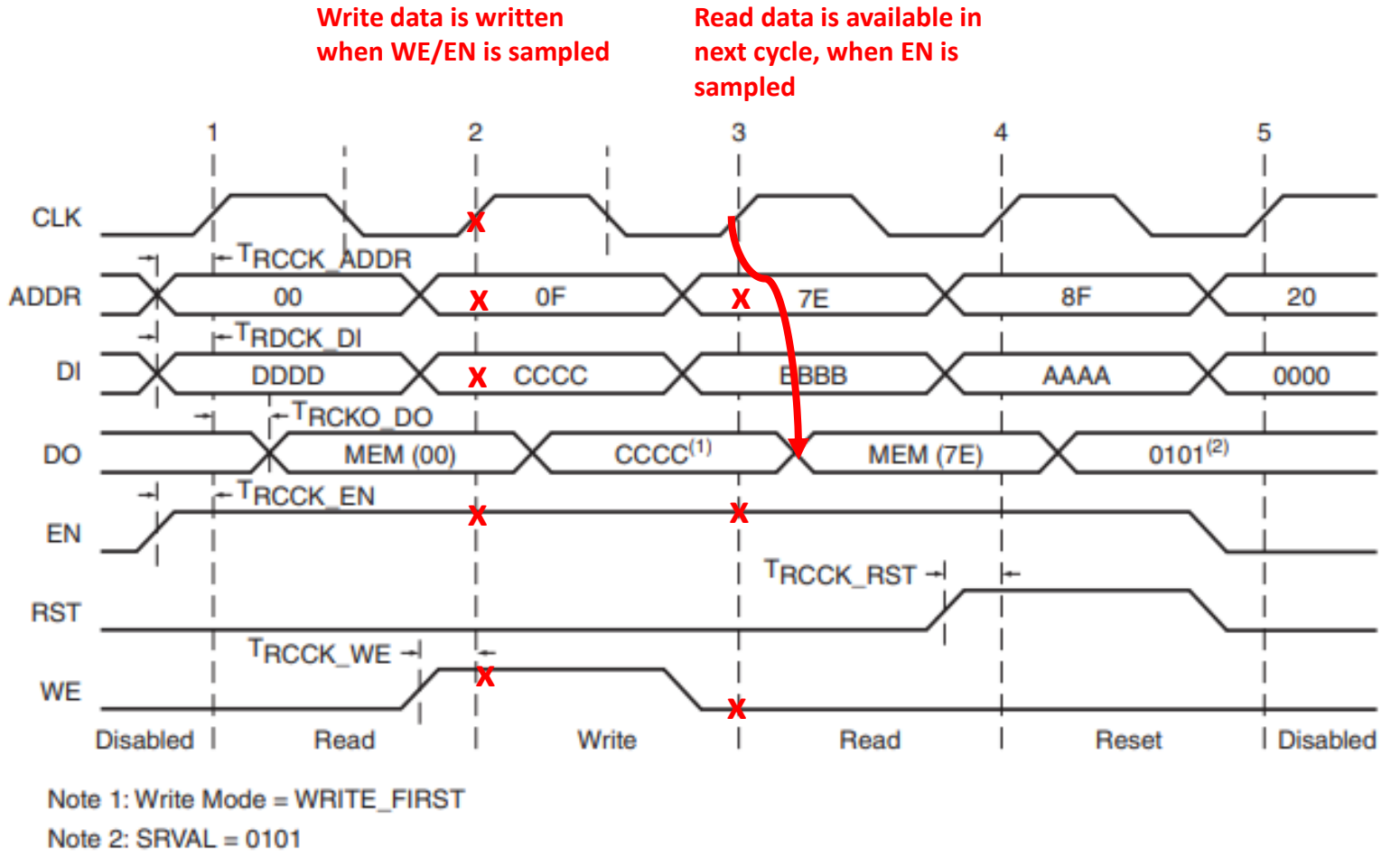
SRAM Access in behavior model and Synthesizable Hardware Design
ref : spiflash-vip.v v.s. spiflash.v

Example: spiflash design

- spiflash-vip.v - spiflash behavior model
  - access sram as an model
- bram.v – BlockRAM behavior model
- spiflash.v
  - Adapt from spiflash-vip.v, synthesizable verilog design
  - Generate bram interface signal to access data

  Reference code:
  https://github.com/bol-edu/caravel-soc_fpga-lab/tree/main/spiflash

SRAM Access Timing



SRAM access has different modes, refer to https://docs.xilinx.com/r/en-US/am007-versal-memory/Read-Operation?tocId=VRYu0HURA1U147fufYDMNQ

BRAM Model

```
module bram #( parameter FILENAME = "firmware.hex")
(
   input   wire    CLK;
   input   wire    [3:0]   WE0;
   input   wire    EN0;
   input   wire    [31:0]  Di0;
   output  reg     [31:0]  Do0;
   input   wire    [31:0]   A0
)
reg [7:0] RAM[0:4*1024*1024-1];   // Declare Memory Storage
```

@(posedge CLK) if EN0 is sampled, output its memory content to Do0

@(posedge CLK) if WE[3:0] is sampled, RAM is written with Di0 per byte

```
always @(posedge CLK)
    if(EN0) begin
      Do0 <= {RAM[{A0[31:2],2'b11}],
              RAM[{A0[31:2],2'b10}],
              RAM[{A0[31:2],2'b01}],
              RAM[{A0[31:2],2'b00}]};
      if(WE0[0]) RAM[{A0[31:2],2'b00}] <= Di0[7:0];
      if(WE0[1]) RAM[{A0[31:2],2'b01}] <= Di0[15:8];
      if(WE0[2]) RAM[{A0[31:2],2'b10}] <= Di0[23:16];
      if(WE0[3]) RAM[{A0[31:2],2'b11}] <= Di0[31:24];
    end
    else
       Do0 <= 32'b0;
```

```
initial begin
                $display("Reading %s",  FILENAME);
                $readmemh(FILENAME, RAM);
                $display("%s loaded into memory", FILENAME);
                $display("Memory 5 bytes = 0x%02x 0x%02x 0x%02x 0x%02x 0x%02x",
                                RAM[0], RAM[1], RAM[2], RAM[3], RAM[4]);
end
```

spiflash-vip – behavior model to access RAM

Memory data is available the same time address is supplied. This is not feasible in the actual memory system.

Memory defined and initialization

```
reg [7:0] memory [0:16*1024*1024-1]; // 16MB
initial begin    // memory content loaded data from file
  $readmemh(FILENAME, memory);
end
```

memory read access

```
buffer = memory[spi_addr]; // memory read
```

memory write access

```
memory[spi_addr] = buffer;
```

spiflash.v – Synthesizable hardware design

- Generate SRAM interface signals
  - Addr, EN, WEN, Din, Dout
  - **The interface signal is generated from internal control logic (FSM)** Interface signals follows the interface timing specification, e.g. read data is available in next clock cycle

BRAM RAM

Addr
EN
WEN
Din
Dout

Clk
Rst

```
// BRAM Interface
  assign romcode_Addr_A = {8'b0, spi_addr};
  assign romcode_Din_A = 32'b0;
  assign romcode_EN_A = (bytecount >= 4);
  assign romcode_WEN_A = 4'b0;
  assign romcode_Clk_A = ap_clk;
  assign romcode_Rst_A = ap_rst;

wire [7:0] memory;
assign memory =
            (spi_addr[1:0] == 2'b00) ? romcode_Dout_A[7:0] :
            (spi_addr[1:0] == 2'b01) ? romcode_Dout_A[15:8] :
            (spi_addr[1:0] == 2'b10) ? romcode_Dout_A[23:16] :
                                       romcode_Dout_A[31:24] ;
```