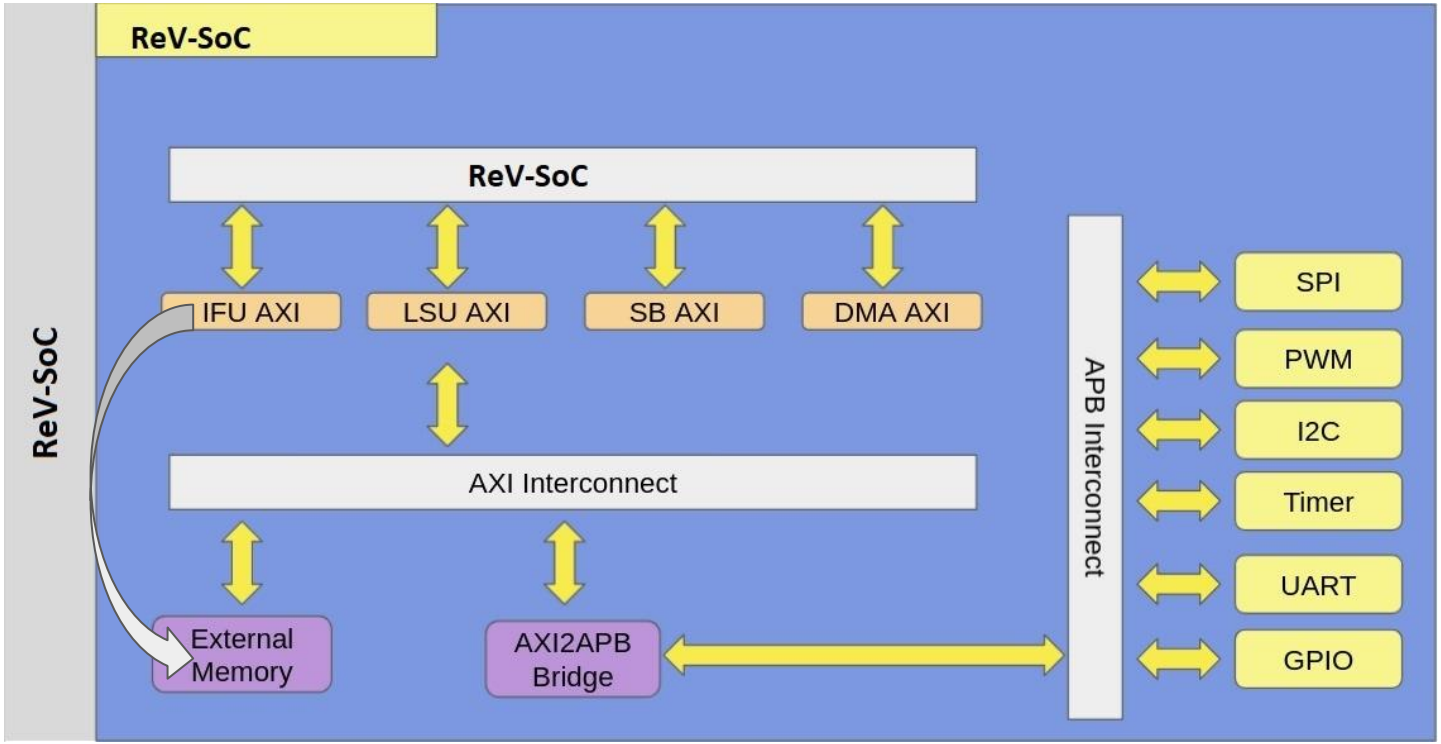
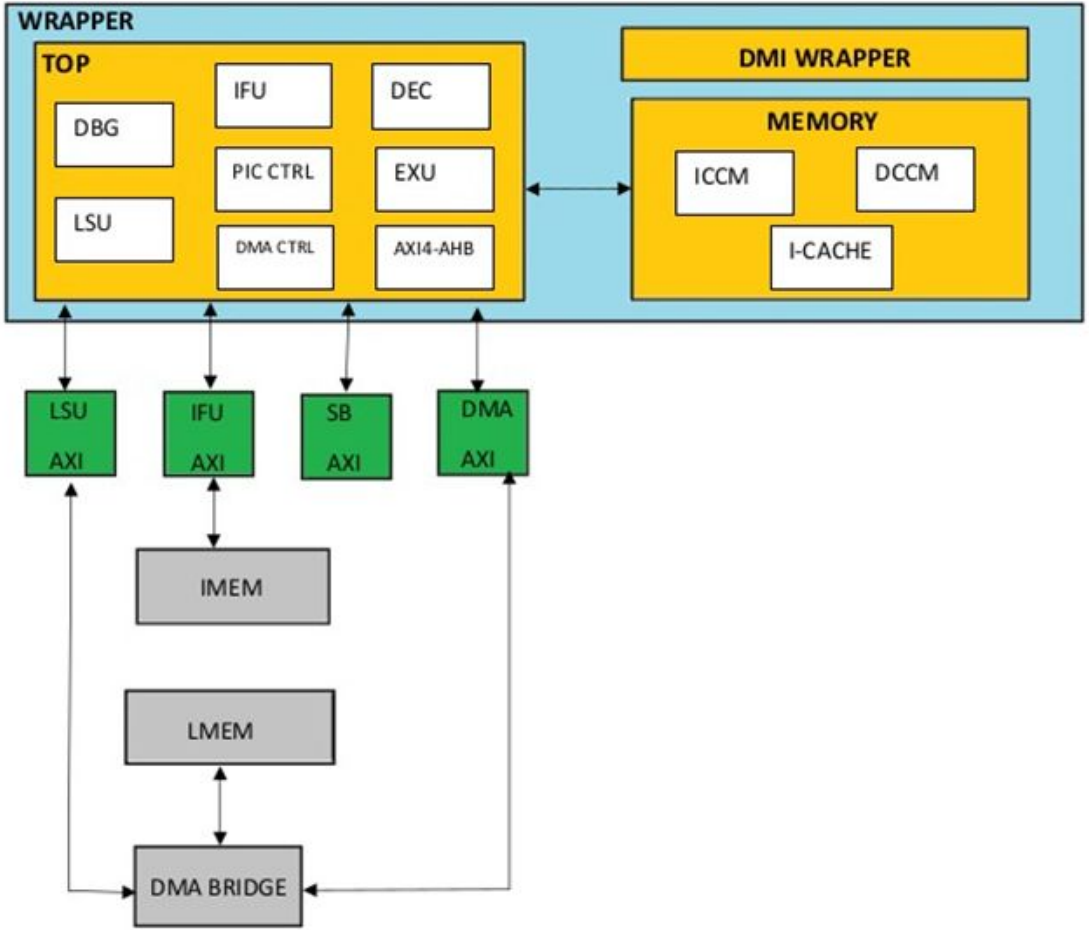
ReV-SoC


ReV-SoC
ReV-SoC
ReV-SoC
IFU AXI
LSU AXI
SB AXI
DMA AXI
AXI Interconnect
External Memory
AXI2APB Bridge
APB Interconnect
SPI
PWM
I2C
Timer
UART
GPIO

Architecture Of SweRV-EL2



•Instruction closely-coupled memory ICCM, Data closely-coupled memory DCCM are optional internal memories.

•Four system bus interfaces for instruction fetch, data accesses, debug accesses, and external DMA accesses to closely-coupled memories (configurable as 64-bit AXI4 or AHB-Lite).

•**ADDRESSES:**

• 0x0008_0000 - DCCM start address to load.

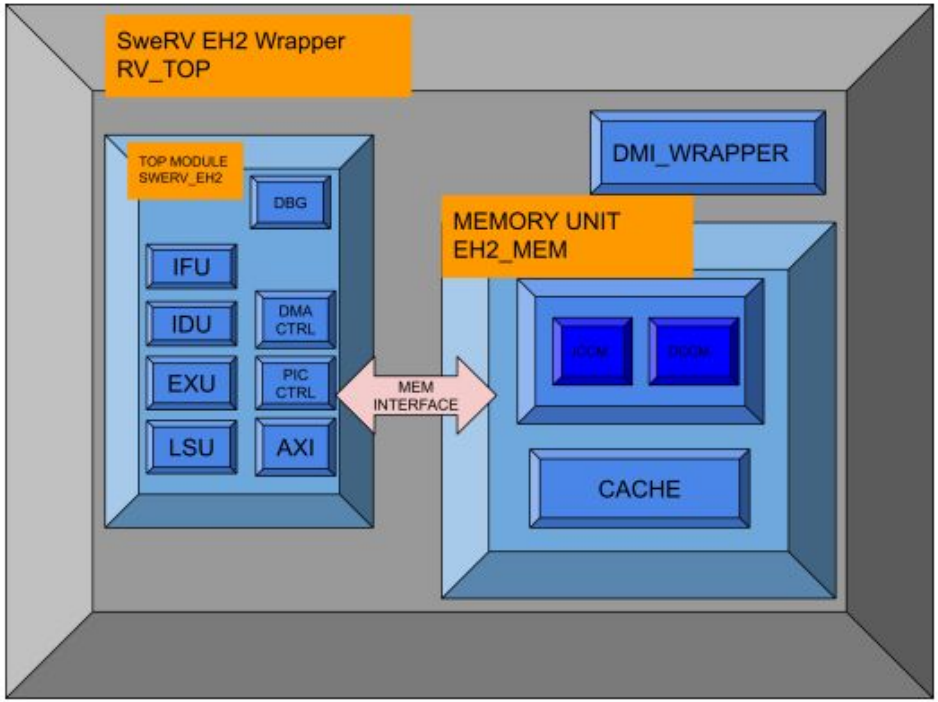
•0x0009_FFFF - DCCM end address to load.

• 0x0004_000 - ICCM start address to load.

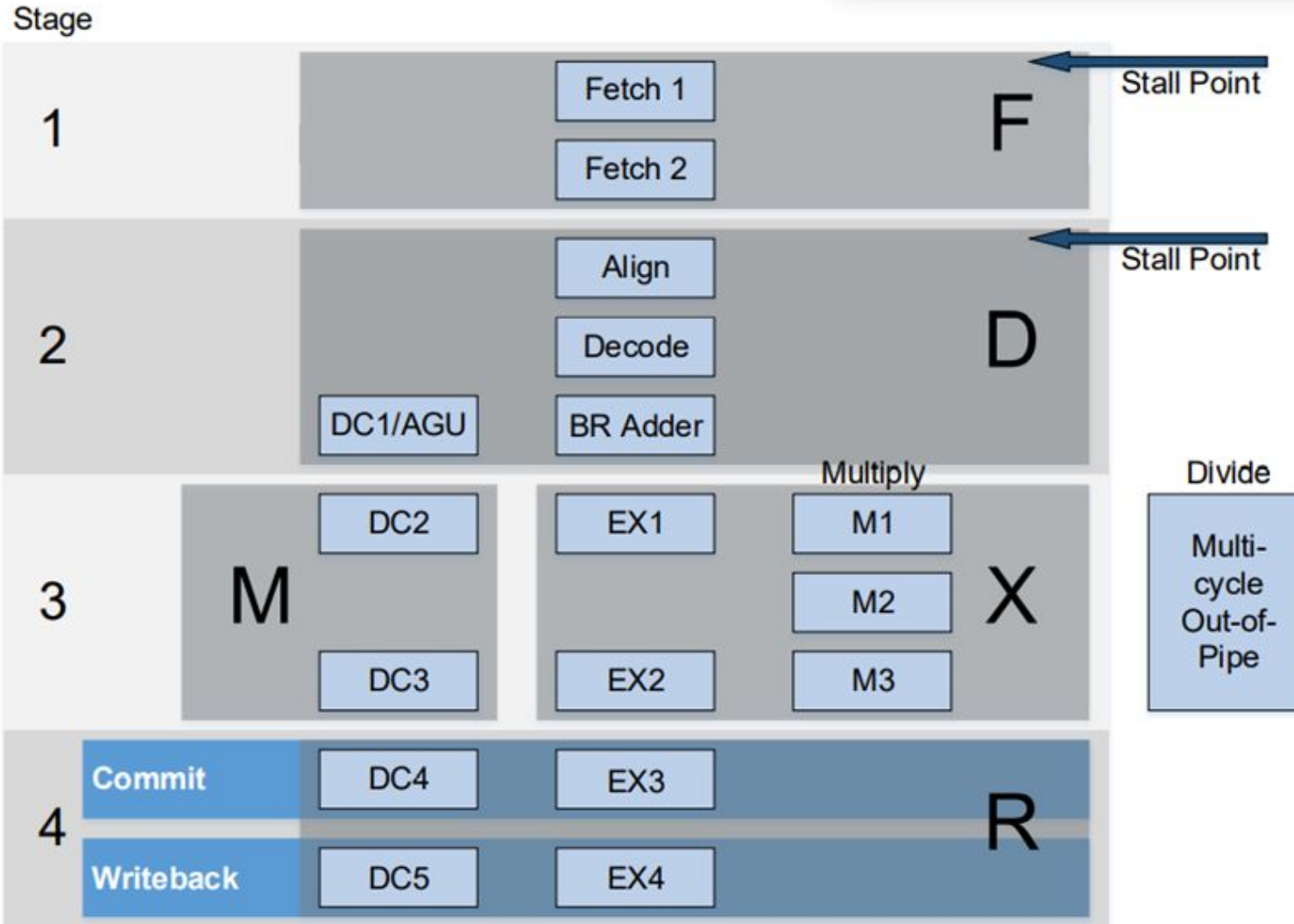
•0x0005_FFFF - ICCM end address to load.

SweRV EL2 Core RV 32 IMC

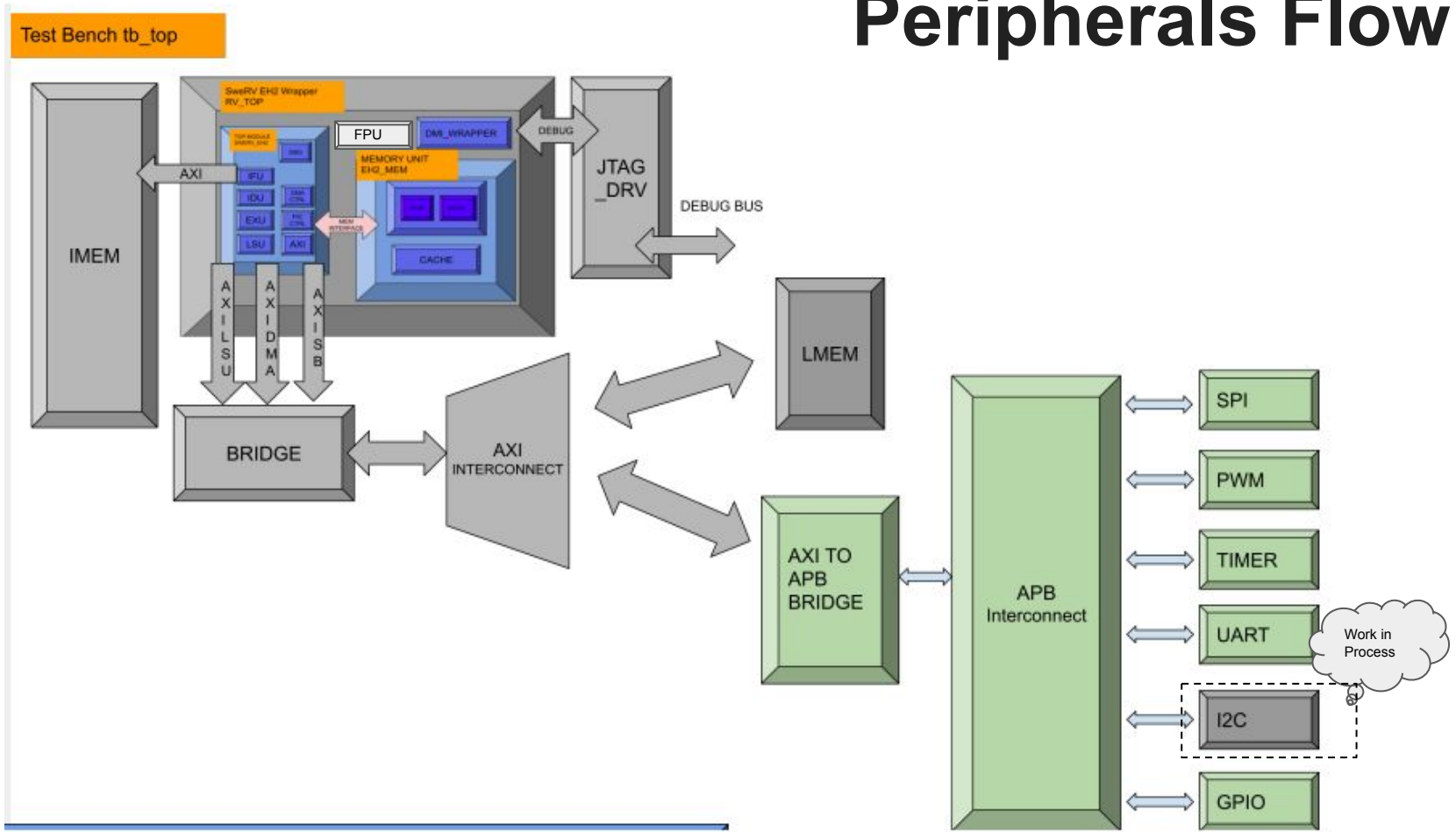
Closer View of Wrapper

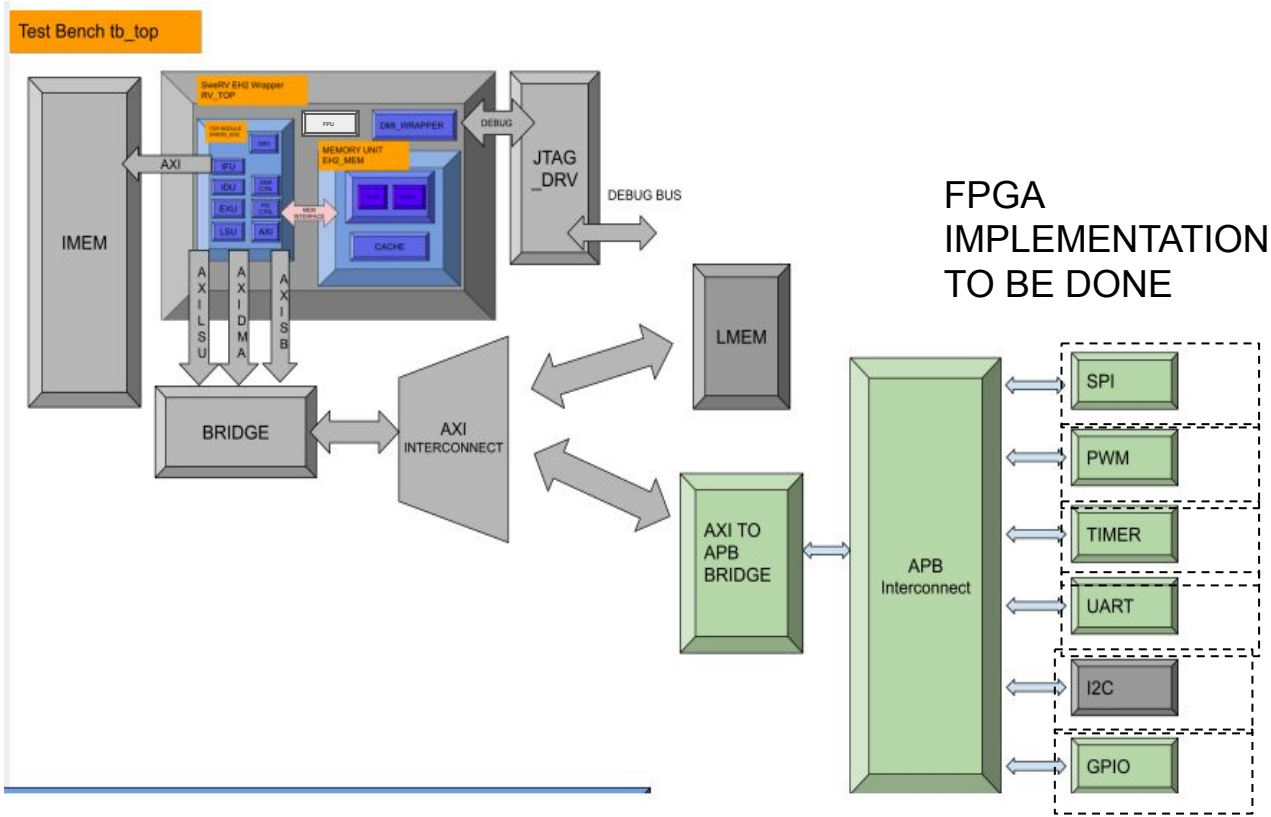


Pipeline Flow



Peripherals Flow

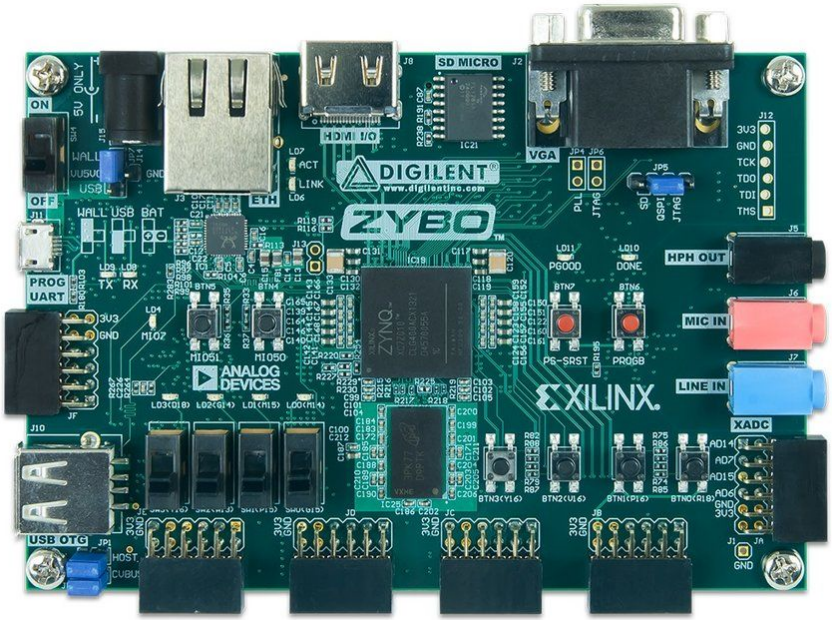




FPGA
IMPLEMENTATION HAS
TO BE DONE

FPGA Implementation

Removing Testing elements from testbench and converting it to top so that design can be synthesized



DETAILS

TIMER Stand alone verified. integrated with SweRV. Verification With SweRV done FPGA implementation is left.

SPI Stand alone verified. integrated with SweRV. Verification With SweRV done FPGA implementation is left.

GPIO Stand alone verified. integrated with SweRV. Verification With SweRV and FPGA implementation is left.

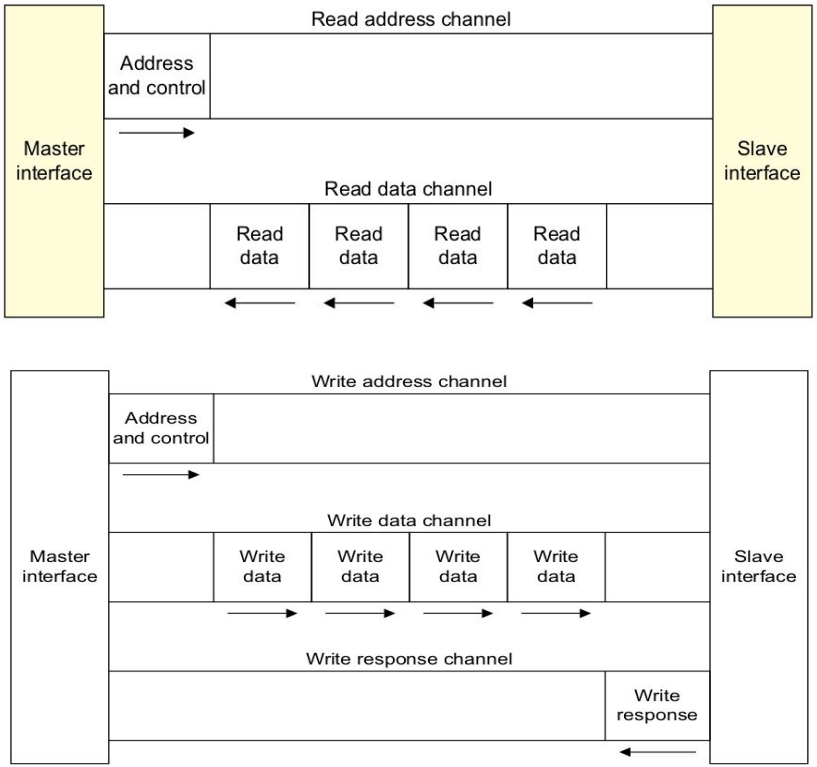
UART Stand alone verified. integrated with SweRV. Verification With SweRV and FPGA implementation is left.

PWM Stand alone verified. integrated with SweRV. Verification With SweRV and FPGA implementation is left.

I2C Stand alone verification in process.

FPU Literature Review Plan Implementation Stand alone Verification, Integration and Final Verification all left.

AXI BUS PROTOCOL



The AXI protocol is burst-based and defines the following independent transaction channels:

- read address
- read data
- write address
- write data
- write response.

AXI BUS PROTOCOL

CONVENTIONAL PREFIXES
For Example:

- **AWADDR** in this signal **AW** is indicating that it belongs to address write channel. Lastly the ADDR is the name of the signal which is basically the address.

- **WDATA** in this signal **W** stands for Write channel and **DATA** is the name of signal.

- **BRESP** here **B** represents write response channel and **RESP** is the name of the signal.

- **ARADDR** in this signal **AR** stands for address read channel. Lastly the **ADDR** is the name of the signal which is basically the address.

- **RDATA** in this signal **R** stands for read channel DATA is the name of signal.

AXI BUS PROTOCOL

GLOBAL SIGNALS
ACLK Clock source
ARESETn Reset source active low

AXI BUS PROTOCOL

- ## Write address channel
- **AWID** Master Write address ID. This signal is the identification tag for the write address group
- **AWADDR** Master Write address. The write address gives the address of the first transfer in a write
- **AWLEN** Master Burst length. The burst length gives the exact number of transfers in a burst. This information determines the number of data transfers associated with the address.This changes between AXI3 and AXI4. See Burst length on page
- **AWSIZE** Master Burst size. This signal indicates the size of each transfer in the burst.
- **AWBURST** Master Burst type. The burst type and the size information, determine how the address for each transfer within the burst is calculated.
- **AWLOCK** Master Lock type. Provides additional information about the atomic characteristics of the transfer. This changes between AXI3 and AXI4.
- **AWCACHE** Master Memory type. This signal indicates how transactions are required to progress through a system.
- **AWPROT** Master Protection type. This signal indicates the privilege and security level of the transaction, and whether the transaction is a data access or an instruction access.
- **AWQOS** Master Quality of Service, QoS. The QoS identifier sent for each write transaction. Implemented only in AXI4.
- **AWREGION** Master Region identifier. Permits a single physical interface on a slave to be used for multiple logical interfaces. Implemented only in AXI4.
- **AWUSER** Master User signal. Optional User-defined signal in the write address channel.Supported only in AXI4.
- **AWVALID** Master Write address valid. This signal indicates that the channel is signaling valid write address and control information.
- **AWREADY** Slave Write address ready. This signal indicates that the slave is ready to accept an address and associated control signals.

AXI BUS PROTOCOL

- ## Write data channel
- **WID** Master Write ID tag. This signal is the ID tag of the write data transfer. Supported only in AXI3.
- **WDATA** Master Write data.
- **WSTRB** Master Write strobes. This signal indicates which byte lanes hold valid data. There is one write
- strobe bit for each eight bits of the write data bus.
- **WLAST** Master Write last. This signal indicates the last transfer in a write burst.
- **WUSER** Master User signal. Optional User-defined signal in the write data channel.
- Supported only in AXI4.
- **WVALID** Master Write valid. This signal indicates that valid write data and strobes are available.
- **WREADY** Slave Write ready. This signal indicates that the slave can accept the write data.Write address ID. This signal is the identification tag for the write address group

AXI BUS PROTOCOL

- **Write response channel**
- **BID** Slave Response ID tag. This signal is the ID tag of the write response.
- **BRESP** Slave Write response. This signal indicates the status of the write transaction.
- **BUSER** Slave User signal. Optional User-defined signal in the write response channel. Supported only in AXI4.
- **BVALID** Slave Write response valid. This signal indicates that the channel is signaling a valid write response.
- **BREADY** Master Response ready. This signal indicates that the master can accept a write response.

AXI BUS PROTOCOL

- ## Read Address channel
- **ARID** Master Read address ID. This signal is the identification tag for the read address group of signals.
- **ARADDR** Master Read address. The read address gives the address of the first transfer in a read burst transaction.
- **ARLEN** Master Burst length. This signal indicates the exact number of transfers in a burst. This changes between AXI3 and AXI4.
- **ARSIZE** Master Burst size. This signal indicates the size of each transfer in the burst.
- **ARBURST** Master Burst type. The burst type and the size information determine how the address for each transfer within the burst is calculated.
- **ARLOCK** Master Lock type. This signal provides additional information about the atomic characteristics of the transfer. This changes between AXI3 and AXI4.
- **ARCACHE** Master Memory type. This signal indicates how transactions are required to progress through a system.
- **ARPROT** Master Protection type. This signal indicates the privilege and security level of the transaction, and whether the transaction is a data access or an instruction access.
- **ARQOS** Master Quality of Service, QoS. QoS identifier sent for each read transaction. Implemented only in AXI4.
- **ARREGION** Master Region identifier. Permits a single physical interface on a slave to be used for multiple logical interfaces. Implemented only in AXI4.
- **ARUSER** Master User signal. Optional User-defined signal in the read address channel.Supported only in AXI4.
- **ARVALID** Master Read address valid. This signal indicates that the channel is signaling valid read address and control information.
- **ARREADY** Slave Read address ready. This signal indicates that the slave is ready to accept an address and associated control signals.

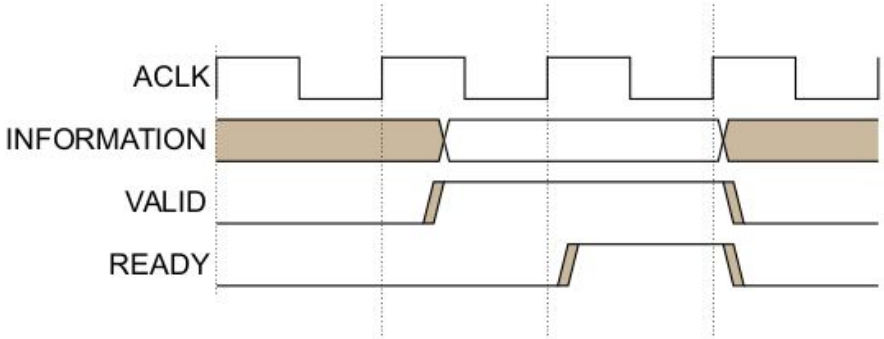
AXI BUS PROTOCOL

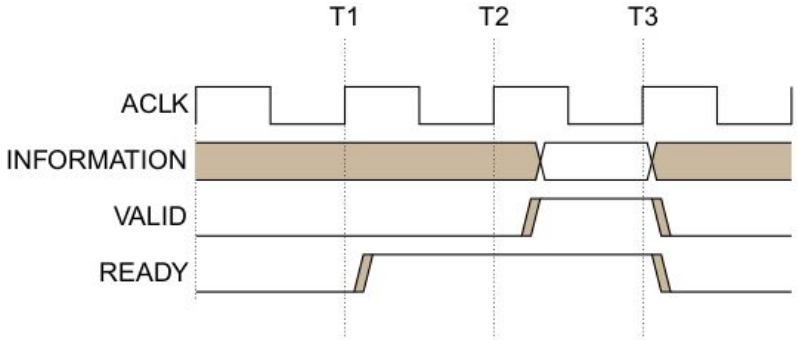
- **Read Data channel**
- **RID** Slave Read ID tag. This signal is the identification tag for the read data group of signals generated by the slave.
- **RDATA** Slave Read data.
- **RRESP** Slave Read response. This signal indicates the status of the read transfer.
- **RLAST** Slave Read last. This signal indicates the last transfer in a read burst.
- **RUSER** Slave User signal. Optional User-defined signal in the read data channel.Supported only in AXI4.
- **RVALID** Slave Read valid. This signal indicates that the channel is signaling the required read data.
- **RREADY** Master Read ready. This signal indicates that the master can accept the read data and response information.

AXI BUS PROTOCOL

Handshake process

All five transaction channels use the same **VALID/READY** handshake process to transfer address, data, and control information. This two-way flow control mechanism means both the master and slave can control the rate at which the information moves between master and slave. The *source* generates the **VALID** signal to indicate when the address, data or control information is available. The *destination* generates the **READY** signal to indicate that it can accept the information. Transfer occurs only when *both* the **VALID** and **READY** signals are HIGH.



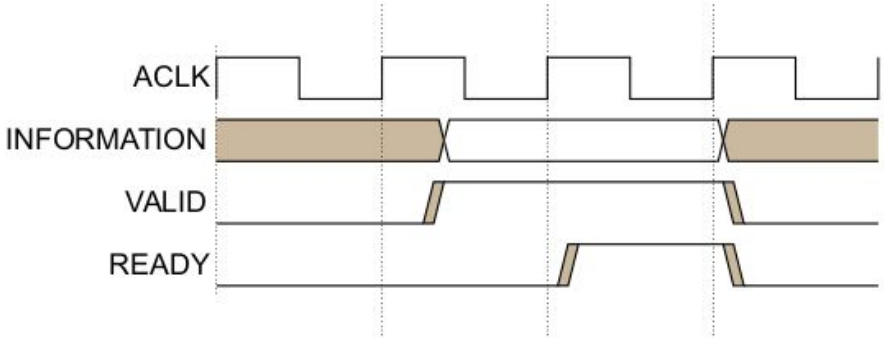


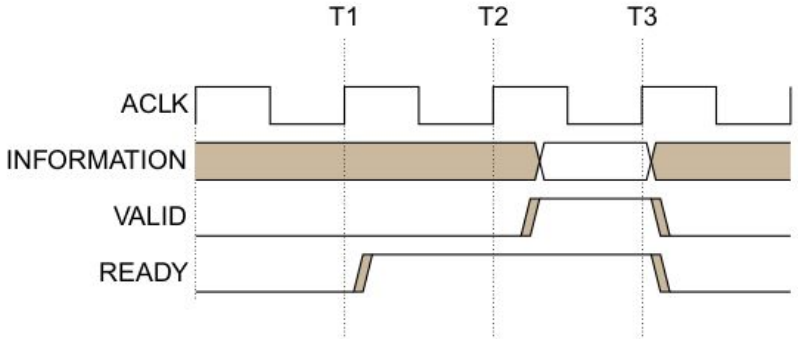
AXI BUS PROTOCOL

Write address channel

The master can assert the **AWVALID** signal only when it drives valid address and control information. **When asserted, AWVALID** must remain asserted until the rising clock edge after the slave asserts **AWREADY**.

The default state of **AWREADY** can be either HIGH or LOW. This specification recommends a default state of HIGH. When **AWREADY** is HIGH the slave must be able to accept any valid address that is presented to it.





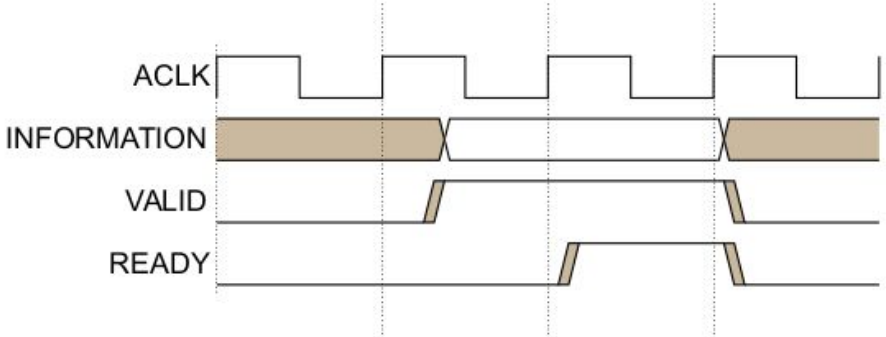
AXI BUS PROTOCOL

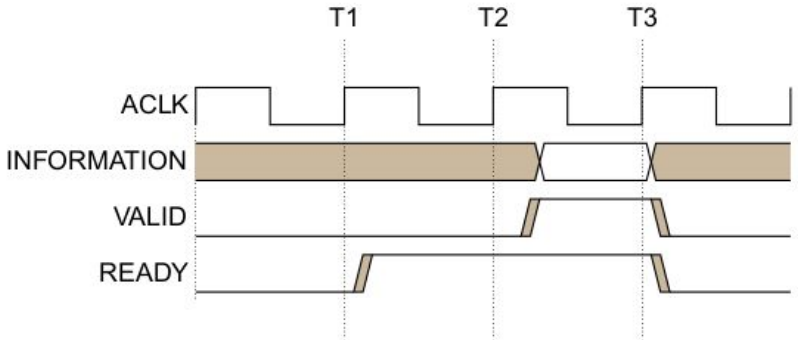
Write data channel

During a write burst, the master can assert the **WVALID** signal only when it drives valid write data. When asserted, **WVALID** must remain asserted until the rising clock edge after the slave asserts **WREADY**.

The default state of **WREADY** can be HIGH, but only if the slave can always accept write data in a single cycle.

The master must assert the **WLAST** signal while it is driving the final write transfer in the burst.





AXI BUS PROTOCOL

- the **VALID** signal of the AXI interface sending information must not be dependent on the **READY** signal of the AXI interface receiving that information
- an AXI interface that is receiving information can wait until it detects a **VALID** signal before it asserts its corresponding **READY** signal.

While it is acceptable to wait for **VALID** to be asserted before asserting **READY**, it is also acceptable to assert **READY** before detecting the corresponding **VALID**. This can result in a more efficient design.

Note:

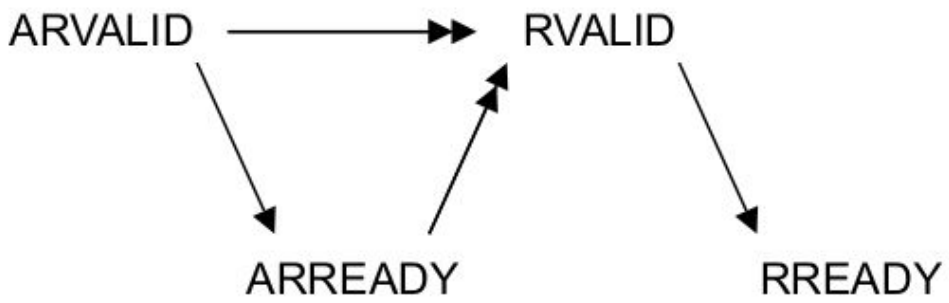
Dependencies between channel handshake signals

To prevent a deadlock situation, the dependency rules that exist between the handshake signals must be observed.

AXI BUS PROTOCOL

Deadlock Condition:
If a slave is waiting for WVALID before asserting AWREADY. Then the deadlock will occur.
Thus dependency rules should be followed and VALID must always be independent of ready.

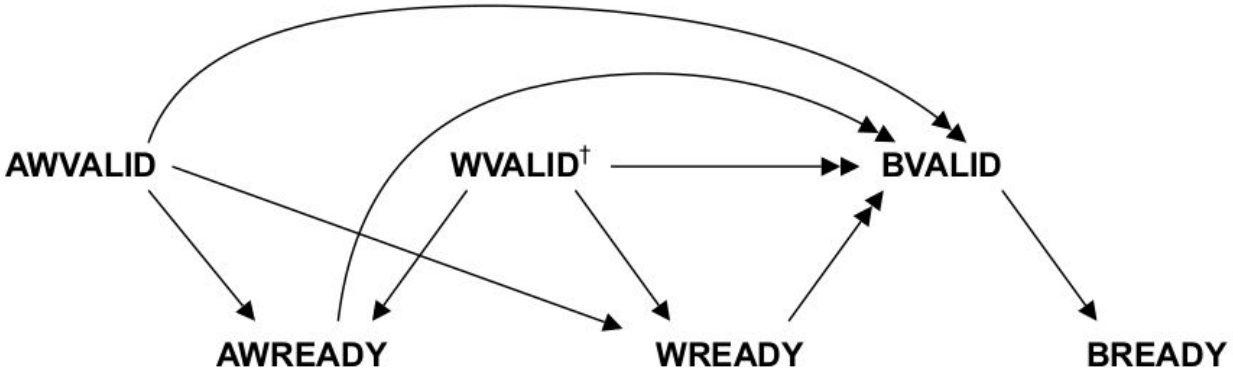
Read transaction dependencies



AXI BUS PROTOCOL

Deadlock Condition:
If a slave is waiting for WVALID before asserting AWREADY. Then the deadlock will occur.
Thus dependency rules should be followed and VALID must always be independent of ready.

Write transaction dependencies



† Dependencies on the assertion of **WVALID** also require the assertion of **WLAST**

AXI BUS PROTOCOL

Address structure

The AXI protocol is burst-based. The master begins each burst by driving control information and the address of the first byte in the transaction to the slave. As the burst progresses, the slave must calculate the addresses of subsequent transfers in the burst.

A burst must not cross a 4KB address boundary.

Burst Length

- **ARLEN[7:0]**, for read transfers
- **AWLEN[7:0]**, for write transfers.

AXI3 supports burst lengths of 1 to 16 transfers, for all burst types.

AXI4 extends burst length support for the INCR burst type to 1 to 256 transfers. Support for all other burst types in AXI4 remains at 1 to 16 transfers.

Burst_Length = **AxLEN[3:0]** + 1

The burst length for AXI4 is defined as,

Burst_Length = **AxLEN[7:0]** + 1, to accommodate the extended burst length of the INCR burst type in AXI4.

AXI BUS PROTOCOL

Burst size

The maximum number of bytes to transfer in each data transfer, or beat, in a burst, is specified by:

- **ARSIZE[2:0]**, for read transfers
- **AWSIZE[2:0]**, for write transfers.

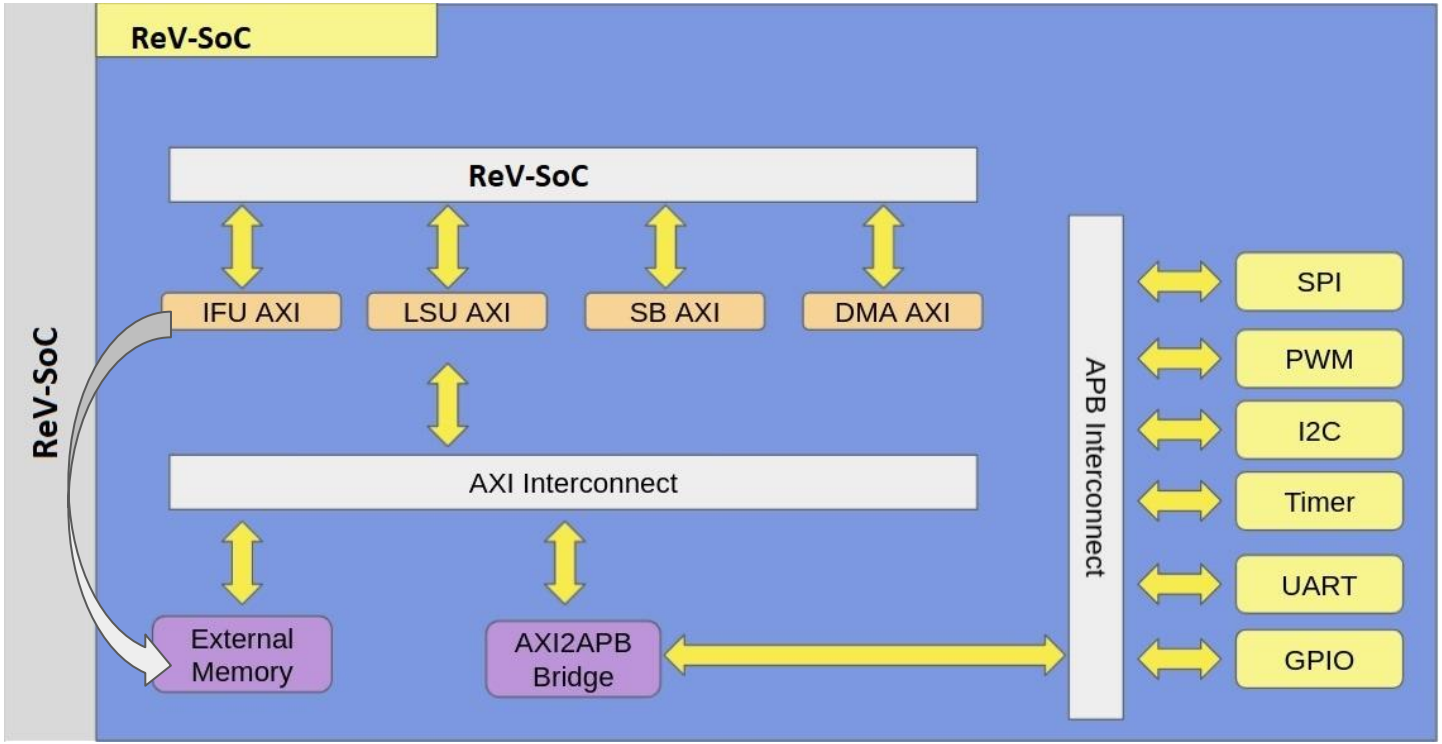
AxSIZE[2:0]	Bytes in transfer
0b000	1
0b001	2
0b010	4
0b011	8
0b100	16
0b101	32
0b110	64
0b111	128

AXI BUS PROTOCOL

Burst type

FIXED	Address remains the same.
INCR	Address increments according to the burst size bytes.

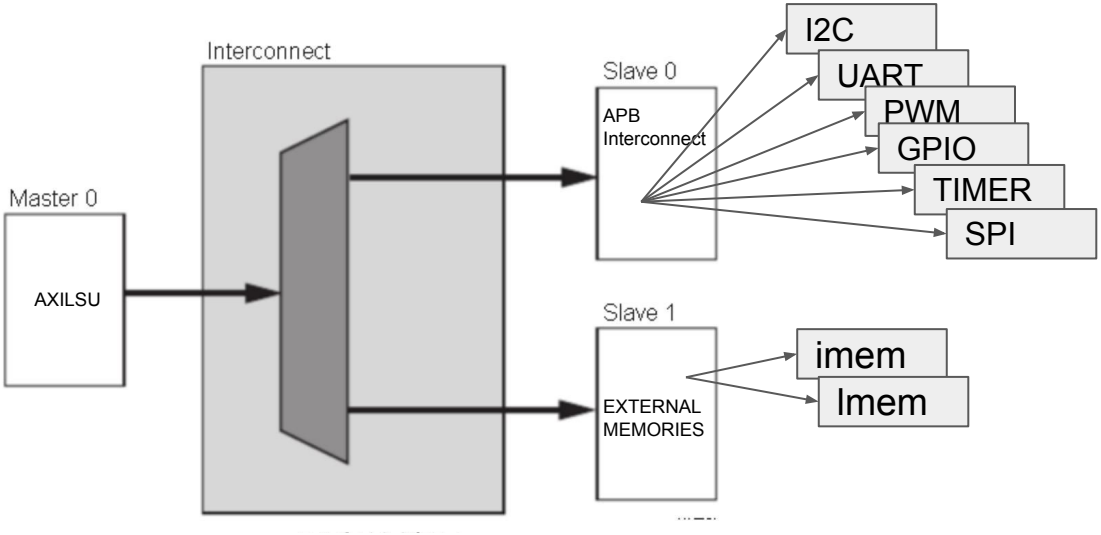
AXI Interconnect



AXI Interconnect

- If Address is between 0x20000000 to 0x80000000 AXI interconnect will allow transactions between core and peripherals

- If Address is between 0x80000000 to 0xFFFFFFFF AXI interconnect will allow transactions from core to external memories



Timer

Registers

- Timer register
- Compare register
- Prescaler / Control Register

Timer Register

- It is basically a counter register which is provided the clk generated by prescaler. if prescaler is enabled with some value otherwise direct PCLK is connected to it.

Compare register

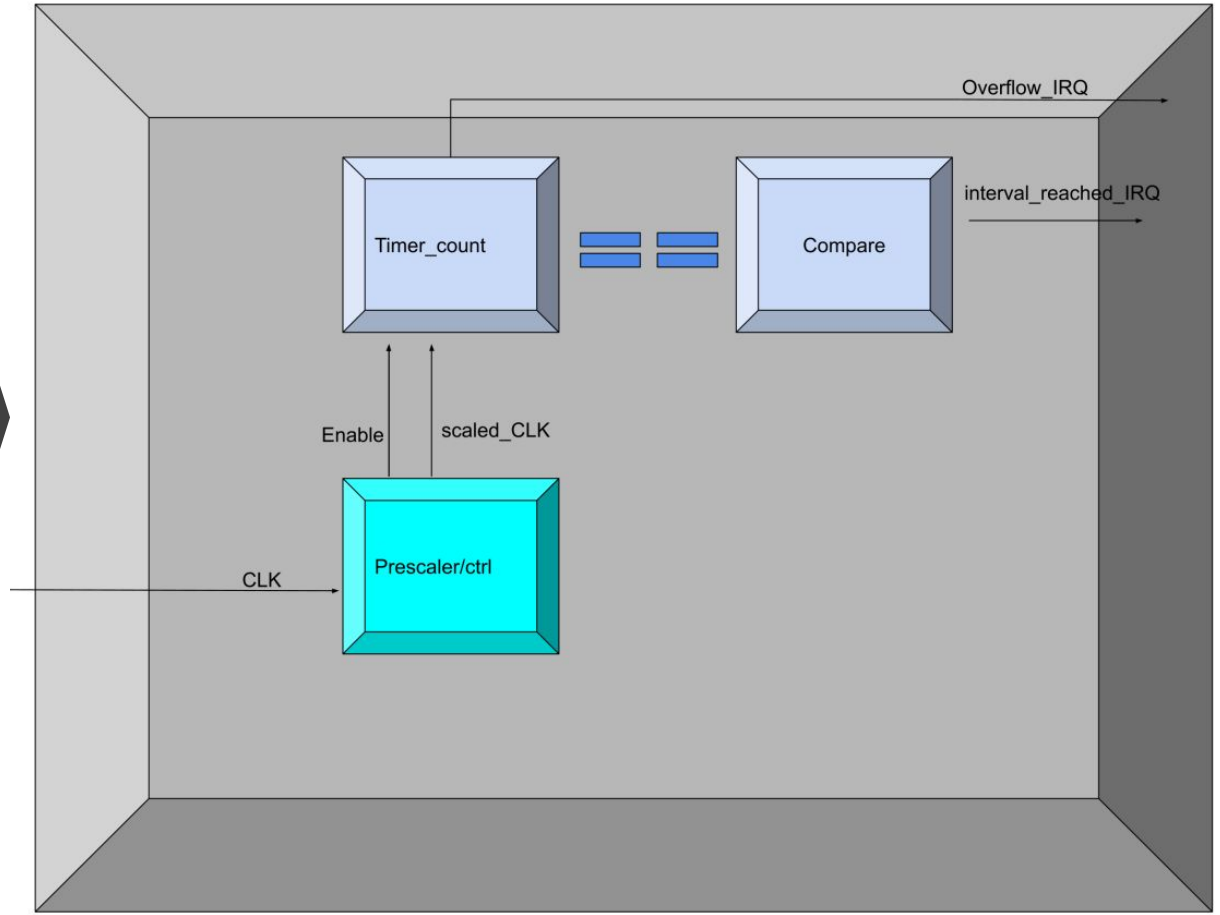
- It contains the value which will be compared with timer register if both are equal then an interrupt will be generated.

Prescaler / Control register

- It is a register which contains some prescaler bits as well as an enable bit. It simply enable / disable the timer based on the value of zeroth bit. ctrl_reg[0] = 0 or ctrl_reg[0] = 1. 0 = disable , 1 = enable.

- In this timer the prescaler_value = ctrl_reg[5:3].
- This value is used to divide the clk frequency.
- For this timer if prescaler is 1 and enable bit is also 1 so the clk provided to timer register will be 10 times slower than the PCLK.

BLOCK DIAGRAM



Address and prescaler format

- REG_TIMER 0
- REG_TIMER_CTRL /Prescaler 1
- REG_CMP 2
- format of :PADDR = {7'd0,index[2:0],2’d0}
- addr => ctrl = 12'd4
- addr => cmp = 12'd8
- addr => timer = 12’d0
- PRESCALER_STARTBIT 3 of reg_timer_ctrl
- PRESCALER_STOPBIT 5 of reg_timer_ctrl
- prescaler format: WDATA = {26'd0,presc_bit[5:3],3’d0}

Prescaler Values

- ctrl = 32'd0 disabled timer no prescaler
- ctrl = 32'd1 enabled timer no prescaler
- ctrl = 32'd8 disabled timer prescaler 1 i.e 10 times larger delay
- ctrl = 32'd9 enabled timer prescaler 1 i.e 10 times lager delay
- ctrl = 32'd16 disabled timer prescaler 2 i.e 20 times greater delay
- ctrl = 32'd17 enabled timer prescaler 2 i.e 20 times greater delay
- ctrl = 32'd24 disabled timer prescaler 3 i.e 30 times greater delay
- ctrl = 32'd25 enabled timer prescaler 3 i.e 30 times greater delay
- ctrl = 32'd32 disabled timer prescaler 4 i.e 40 times greater delay
- ctrl = 32'd33 enabled timer prescaler 4 i.e 40 times greater delay
- ctrl = 32'd40 disabled timer prescaler 5 i.e 50 times greater delay
- ctrl = 32'd41 enabled timer prescaler 5 i.e 50 times greater delay
- ctrl = 32'd48 disabled timer prescaler 6 i.e 60 times greater delay
- ctrl = 32'd49 enabled timer prescaler 6 i.e 60 times greater delay
- ctrl = 32'd56 disabled timer prescaler 7 i.e 70 times greater delay
- ctrl = 32'd57 enabled timer prescaler 7 i.e 70 times greater delay

Specs

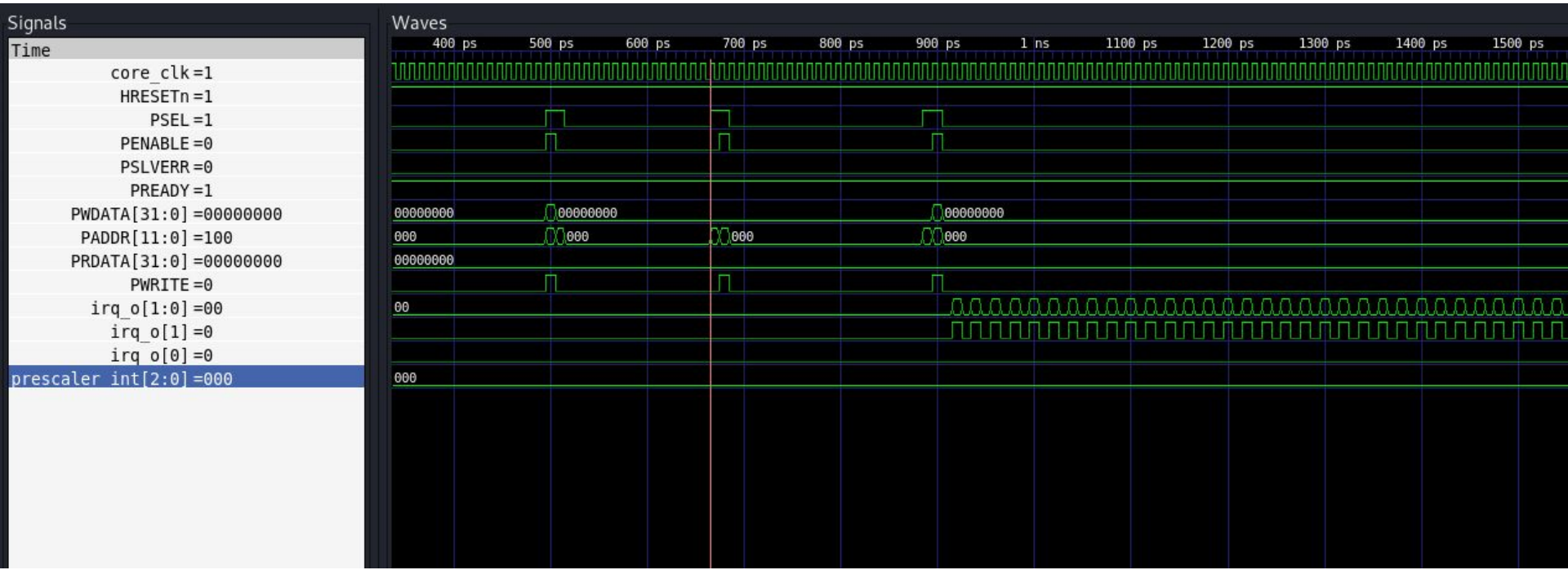
- to set prescaler we have to write on REG_TIMER_CTRL
- to set timer we have to write on REG_CMP
- irq_o[0] == overflow interrupt when REG_Timer == 32’hFFFF_FFFF
- irq_o[1] == time reached interrupt
- zeroth bit of reg timer ctrl is ENABLE BIT
- if you set value 1 in cmp register then you will receive interrupt after every clock cycle
- if you set value 1 in cmp register and 1 value in prescaler bits then you will receive interrupt after every 10 clock cycles
- Conclusion prescaler value 1 will slow down the clock ten times and normal number of cycle-based delay will be given by value of compare register you set.

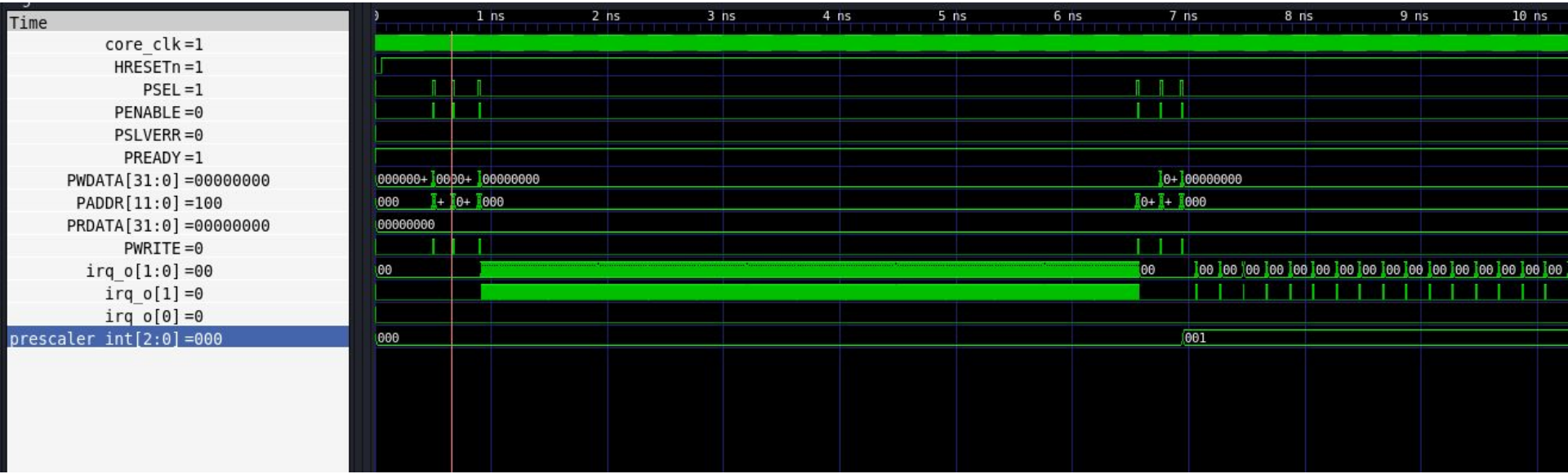
Delay time calculation

- max value through cmp register = 4294967295 clks
- delay_time = time_of_one_clk*(cmp value * prescaler_value)
- Max delay_time = time_of_one_clk*(4294967295*70)
- = time_of_one_clk*(300647710650)

Ports

- input HCLK 1 bit
- input HRESETn 1 bit
- input PADDR 12 bits
- input PWDATA 32 bits
- input PWRITE 1 bit
- input PSEL 1 bit
- input PENABLE 1 bit
- output PRDATA 32 bits
- output PREADY 1 bit
- output PSLVERR 1 bit
- output irq_o 2 bits


Signals
Waves
Time
core_clk =1
HRESETn =1
PSEL =1
PENABLE =0
PSLVERR =0
PREADY =1
PWDATA[31:0] =00000000
PADDR[11:0] =100
PRDATA[31:0] =00000000
PWRITE =0
irq_o[1:0] =00
irq_o[1] =0
irq_o[0] =0
prescaler_int[2:0] =000
400 ps
500 ps
600 ps
700 ps
800 ps
900 ps
1 ns
1100 ps
1200 ps
1300 ps
1400 ps
1500 ps
00000000
00000000
00000000
000
000
000
00000000
00
000


Time
core_clk =1
HRESETn =1
PSEL =1
PENABLE =0
PSLVERR =0
PREADY =1
PWDATA[31:0] =00000000
PADDR[11:0] =100
PRDATA[31:0] =00000000
PWRITE =0
irq_o[1:0] =00
irq_o[1] =0
irq_o[0] =0
prescaler int[2:0] =000
1 ns
2 ns
3 ns
4 ns
5 ns
6 ns
7 ns
8 ns
9 ns
10 ns

WHY SPI

Serial Peripheral Interface

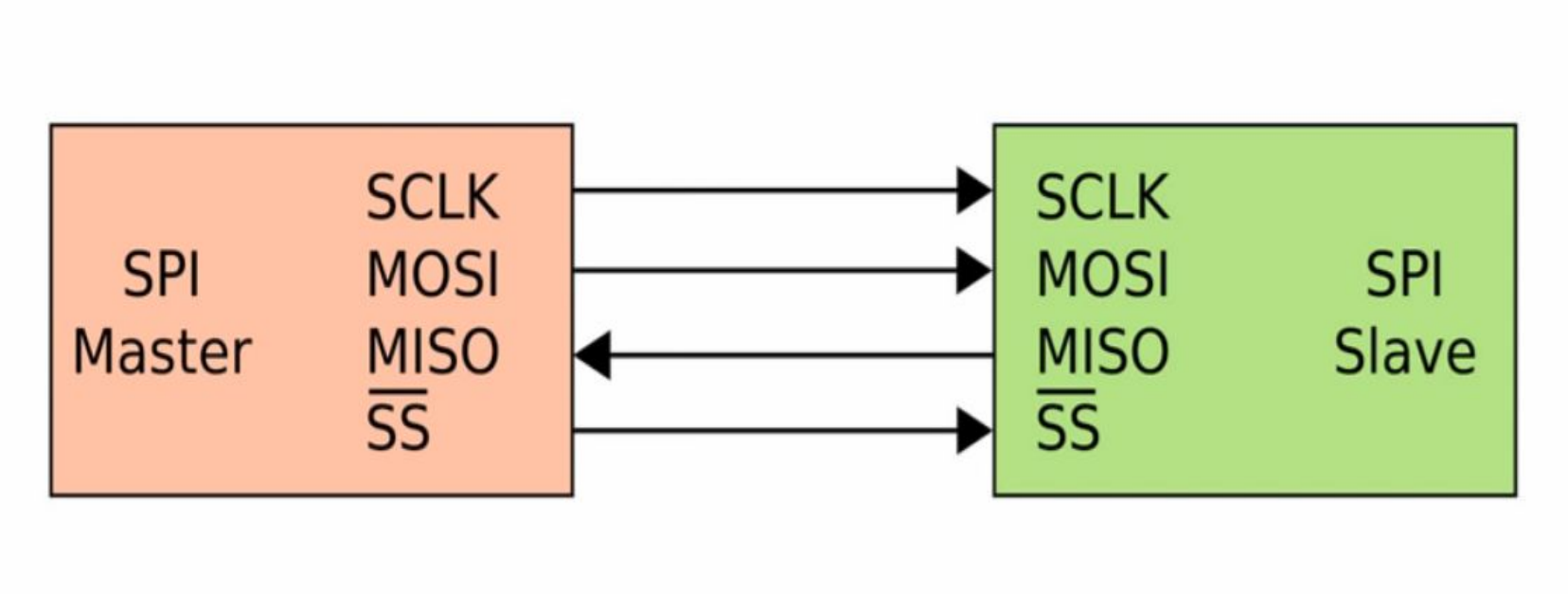
1. SPI has clk whereas UART doesn't
2. SPI is much faster than UART
3. SPI max speed = fPCLK/2 bps
4. Full duplexed that is you can send and receive on the same time]
5. Higher speed than uart or I2C no addresses just chip select

SPI

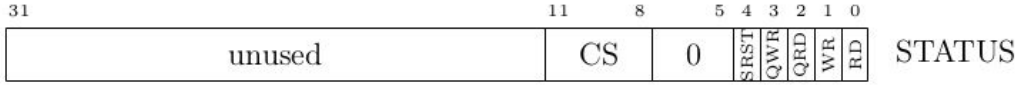
Serial Peripheral Interface

1. CPOL = clock polarity
2. if CPOL = 0; then clk will idle on zero level
3. if CPOL = 1; then clk will idle on high level
4. CPHA = clk phase
5. if CPHA = 0 then data will be sampled on rising edge
6. if CPHA = 1 then data will be sampled on falling edge

SPI



Status Register



`Bit 11:8` **CS**: Chip Select.
Specify the chip select signal that should be used for the next transfer.

`Bit 4` **SRST**: Software Reset.
Clear FIFOs and abort active transfers.

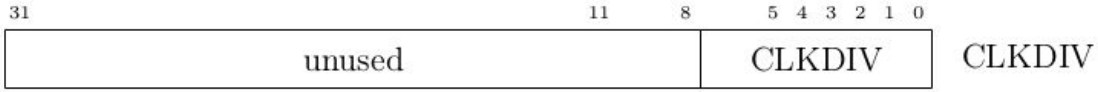
`Bit 3` **QWR**: Quad Write Command.
Perform a write using Quad SPI mode.

`Bit 2` **QRD**: Quad Read Command.
Perform a read using Quad SPI mode.

`Bit 1` **WR**: Write Command.
Perform a write using standard SPI mode.

`Bit 0` **RD**: Read Command.
Perform a read using standard SPI mode.

CLK DIVIDER REGISTER



Bit 7:0 **CLKDIV**: Clock Divider.
Clock divider value used to divide the SoC clock for the SPI transfers. This register should not be modified while a transfer is in progress.

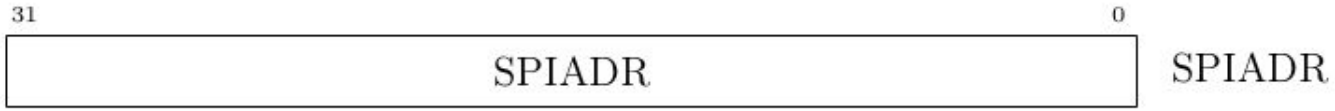
5.3.3 SPICMD (SPI Command)

Address: 0x1A10_2008
Reset Value: 0x0000_0000

SPI COMMAND

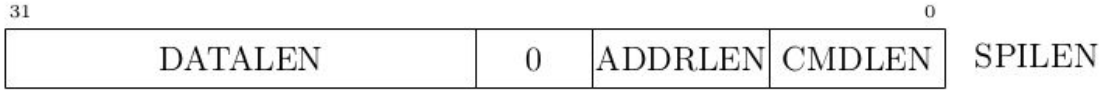
Bit 31:0 **SPICMD**: SPI Command.
When performing a read or write transfer the SPI command is sent first before any data is read or written. The length of the SPI command can be controlled with the SPILEN register.

SPI ADDRESS



Bit 31:0 **SPIADR**: SPI Address.
When performing a read or write transfer the SPI command is sent first before any data is read or written, after this the SPI address is sent. The length of the SPI address can be controlled with the `SPILEN` register.

LENGTH REGISTER

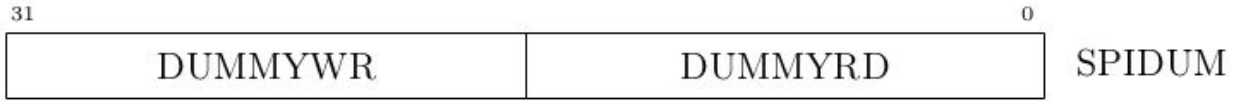


Bit 31:16 **DATALEN**: SPI Data Length.
The number of bits read or written. Note that first the SPI command and address are written to an SPI slave device.

Bit 13:8 **ADDRLEN**: SPI Address Length.
The number of bits of the SPI address that should be sent.

Bit 5:0 **CMDLEN**: SPI Command Length.
The number of bits of the SPI command that should be sent.

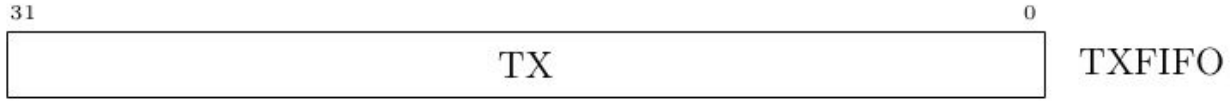
Dummy Cycle



Bit 31:16 **DUMMYWR**: Write Dummy Cycles.
Dummy cycles (nothing being written or read) between sending the SPI command + SPI address and writing the data.

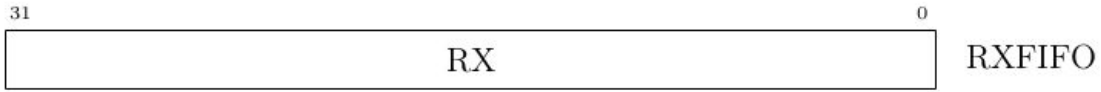
Bit 15:0 **DUMMYRD**: Read Dummy Cycles.
Dummy cycles (nothing being written or read) between sending the SPI command + SPI address and reading the data.

TX FIFO or Data Register (Data to transmit)



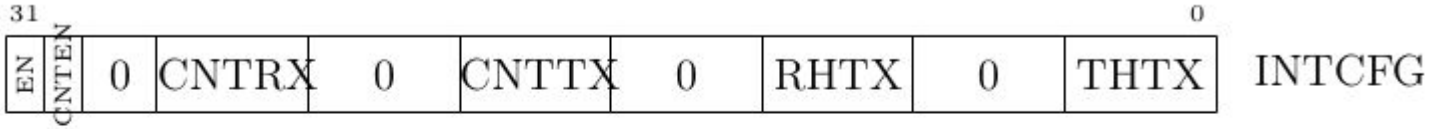
Bit 31:0 **TX**: Transmit Data.
Write data into the FIFO.

RX FIFO or Data to receive

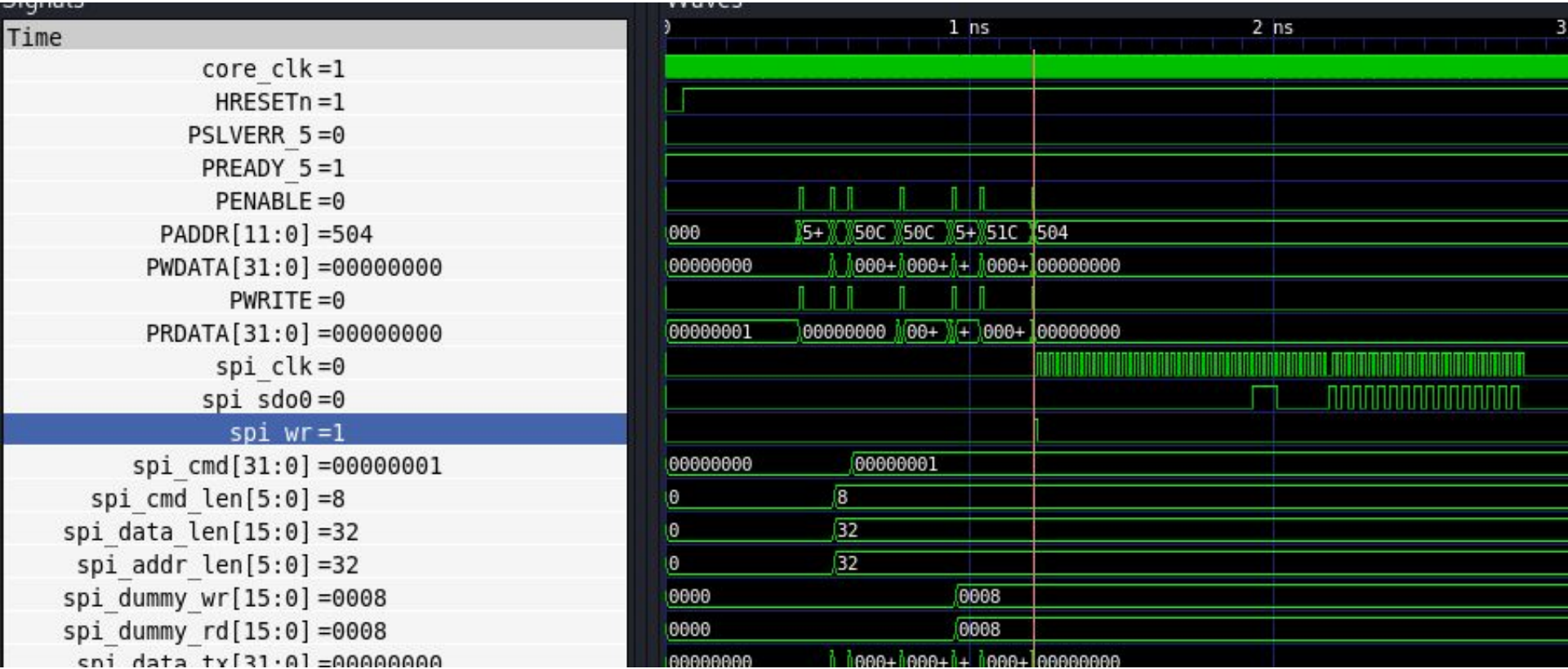


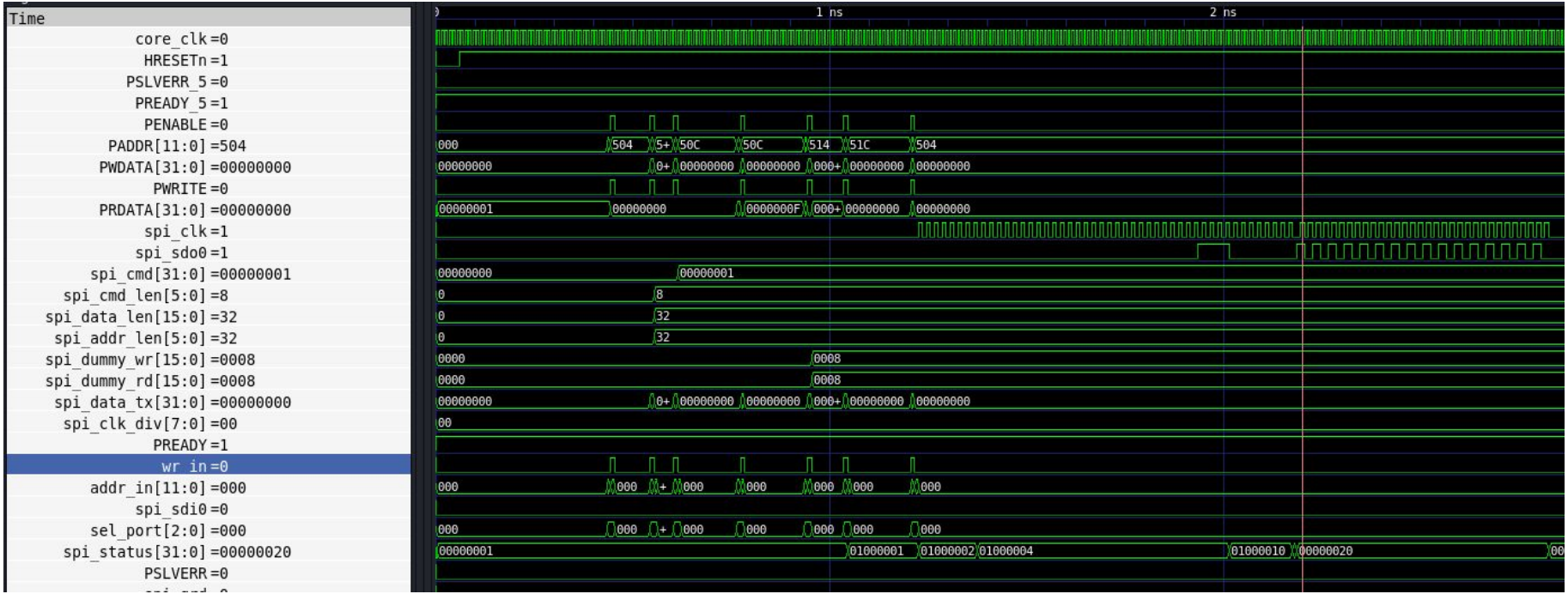
Bit 31:0 **RX**: Receive Data.
Read data from the FIFO.

Interrupt configuration Register



Bit 31 **EN**: Interrupt Enable.
Enable interrupts


Time
core_clk =1
HRESETn =1
PSLVERR_5 =0
PREADY_5 =1
PENABLE =0
PADDR[11:0] =504
PWDATA[31:0] =00000000
PWRITE =0
PRDATA[31:0] =00000000
spi_clk =0
spi_sdo0 =0
spi_wr =1
spi_cmd[31:0] =00000001
spi_cmd_len[5:0] =8
spi_data_len[15:0] =32
spi_addr_len[5:0] =32
spi_dummy_wr[15:0] =0008
spi_dummy_rd[15:0] =0008
1 ns
2 ns
00000001
00000000
0
8
32
0000
0008


Time
core_clk =0
HRESETn =1
PSLVERR_5 =0
PREADY_5 =1
PENABLE =0
PADDR[11:0] =504
PWDATA[31:0] =00000000
PWRITE =0
PRDATA[31:0] =00000000
spi_clk =1
spi_sdo0 =1
spi_cmd[31:0] =00000001
spi_cmd_len[5:0] =8
spi_data_len[15:0] =32
spi_addr_len[5:0] =32
spi_dummy_wr[15:0] =0008
spi_dummy_rd[15:0] =0008
spi_data_tx[31:0] =00000000
spi_clk_div[7:0] =00
PREADY =1
wr_in =0
addr_in[11:0] =000
spi_sdi0 =0
sel_port[2:0] =000
spi_status[31:0] =00000020
PSLVERR =0
1 ns
2 ns

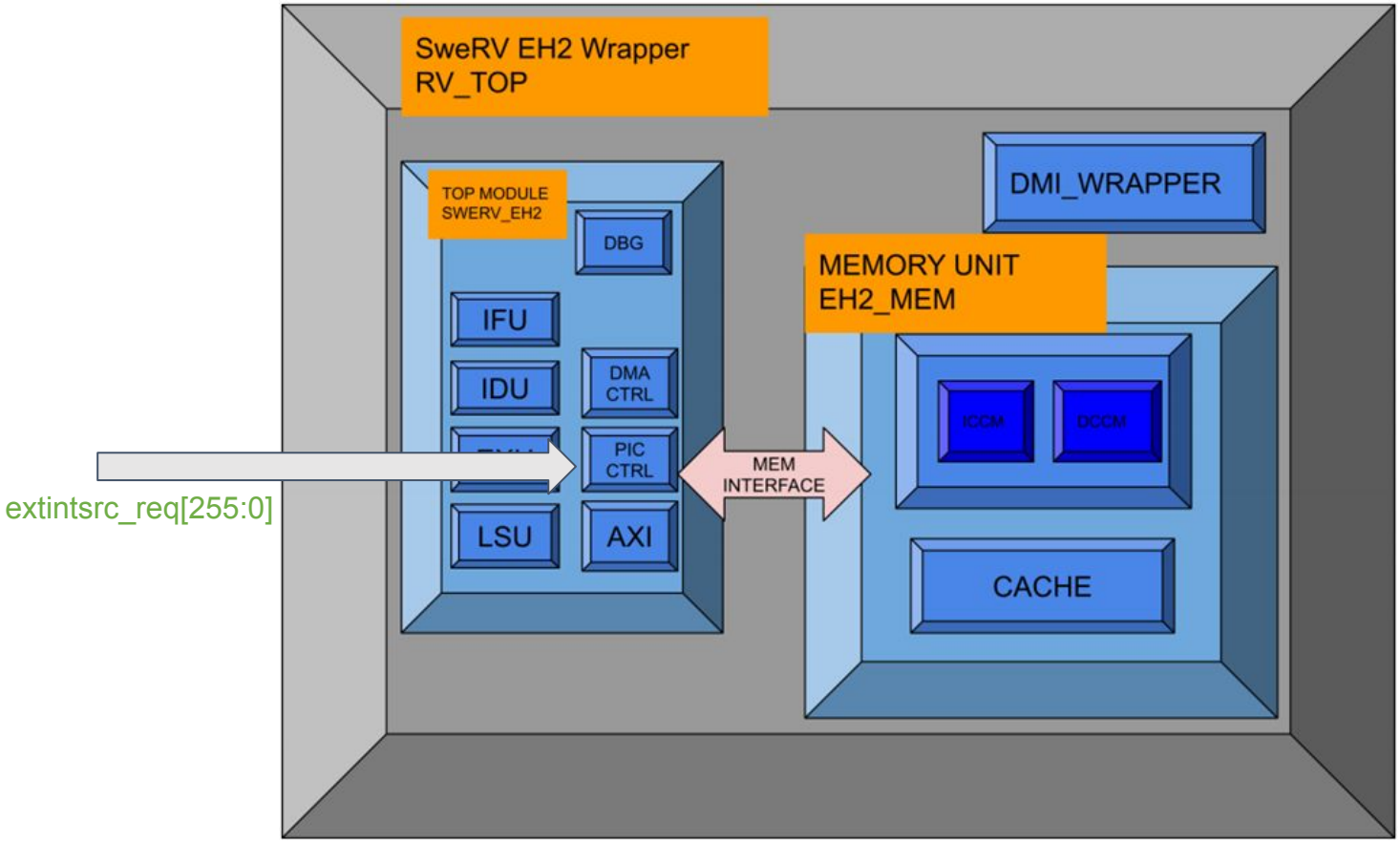
SweRV Programmable Interrupt Controller

Configurations

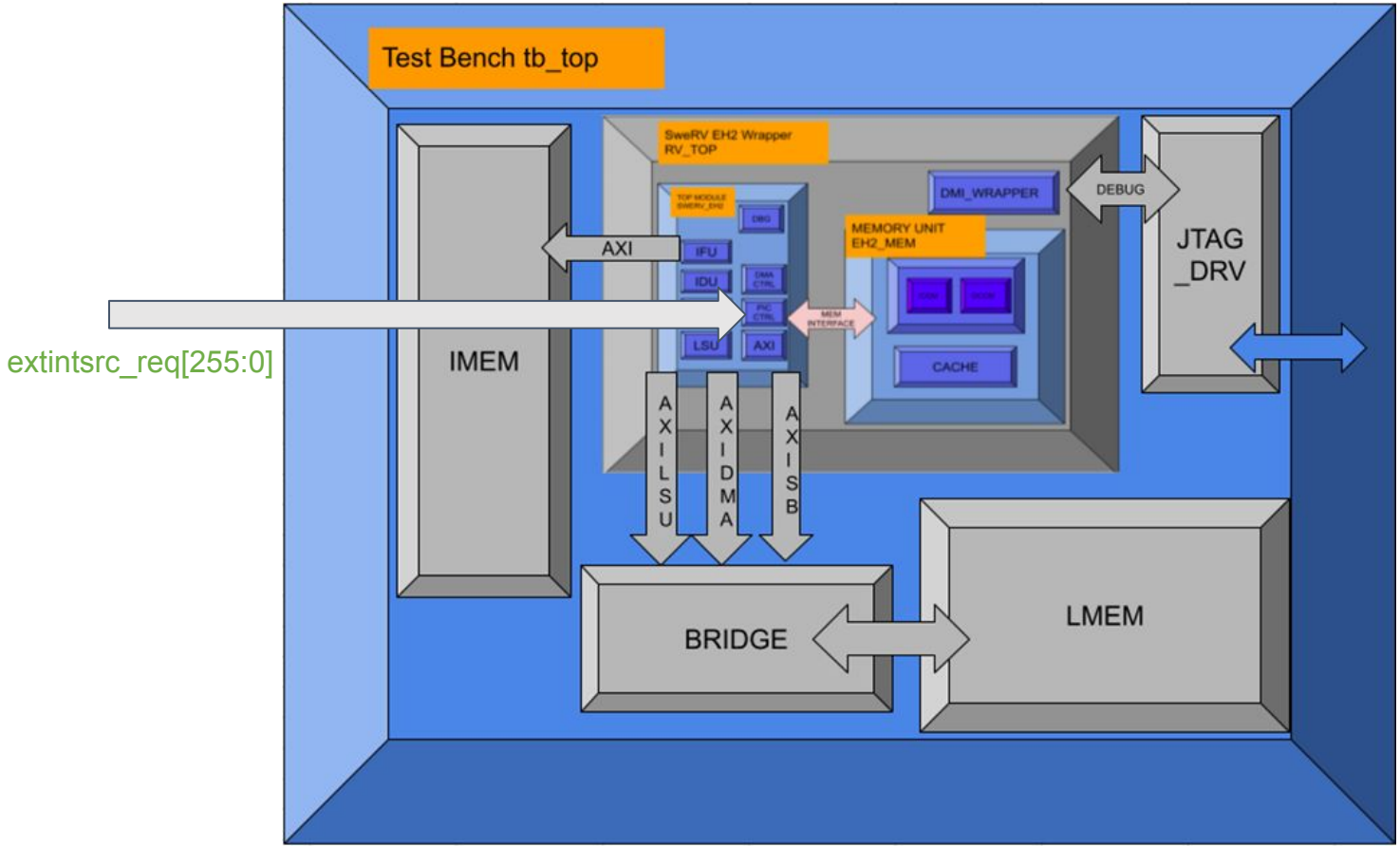
CSRs And Base Addresses

RV_PIC_BASE_ADDR 0xF00C0000
RV_PIC_OFFSET 0xC0000
RV_VECTOR_TABLE_BASE 0x70000000
meivt 0xBC8
meipt 0xBC9
meicpct 0xBCA
meicidpl 0xBCB
meicurpl 0xBCC
meihap 0xFC8
mie 0x304
RV_PIC_MEIGWCTRL_OFFSET 0x4000 //0x4000 + 4*S
RV_PIC_MEIGWCLR_OFFSET 0x5000 //0x5000 + 4*S
RV_PIC_MEIPL_OFFSET 0x0
RV_PIC_MEIP_OFFSET 0x1000
RV_PIC_MEIE_OFFSET 0x2000
RV_PIC_MPICCFG_OFFSET 0x3000
RV_PIC_MEIPT_OFFSET 0x3004

PIC EXPLANATION



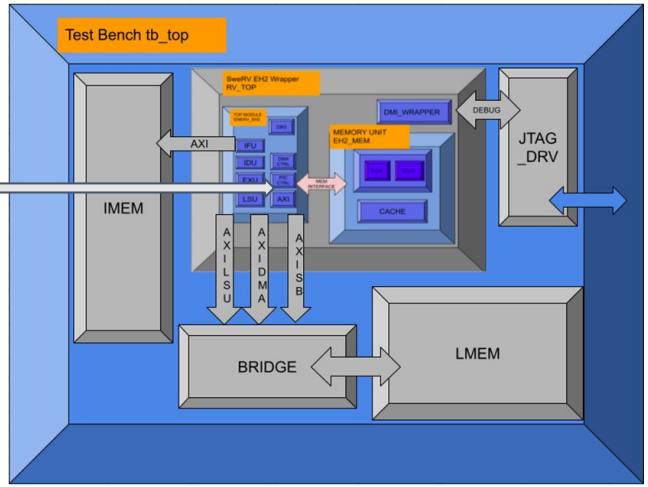
PIC EXPLANATION



PIC EXPLANATION

```
module Interrupt_generator(
   output interrupt_1,
   input core_clk

);
logic [31:0] clk_cnt;
logic  interrupt;
always @(posedge core_clk) begin

  if (clk_cnt < 100)
  interrupt <= 1'b0;
  else if (clk_cnt >= 100 && clk_cnt < 150  )
  interrupt <= 1'b1;
  else if(clk_cnt >= 300 && clk_cnt < 10000 )
   interrupt <= 1'b0;
 // else if(clk_cnt >= 200 && clk_cnt < 250 )
 // interrupt <= 1'b1;

  clk_cnt++;

end
 assign  interrupt_1 = interrupt;
   endmodule
```

extintsrc_req[1]



PIC TASK DETAILS

➔ PIC dummy interrupt generation done through SystemVerilog module and connected to the core interrupt signal extintsrc_req in tb_top.sv.

File name = interrupt_generator.sv

```
1  module Interrupt_generator(
2     output interrupt_1,
3     input core_clk
4
5  );
6  logic [31:0] clk_cnt;
7  logic  interrupt;
8  always @(posedge core_clk) begin
9
10   if (clk_cnt < 100)
11   interrupt <= 1'b0;
12   else if (clk_cnt >= 100 && clk_cnt < 150  )
13   interrupt <= 1'b1;
14   else if(clk_cnt >= 300 && clk_cnt < 10000 )
15    interrupt <= 1'b0;
16  // else if(clk_cnt >= 200 && clk_cnt < 250 )
17  // interrupt <= 1'b1;
18
19   clk_cnt++;
20
21 end
22  assign  interrupt_1 = interrupt;
23    endmodule
```

File name = tb_top.sv

```
logic interrupt_1;
// Temporary inerrupt generator
Interrupt_generator intrupt_gen (.*);
```

```
.extintsrc_req          ( {253'd0,interrupt_1,interrupt_1}
```

PIC TASK PROBLEMS SOLVED

- Interrupt Handling done
- Vector table Base address set
- Interrupt handler set
- Interrupt service routine set

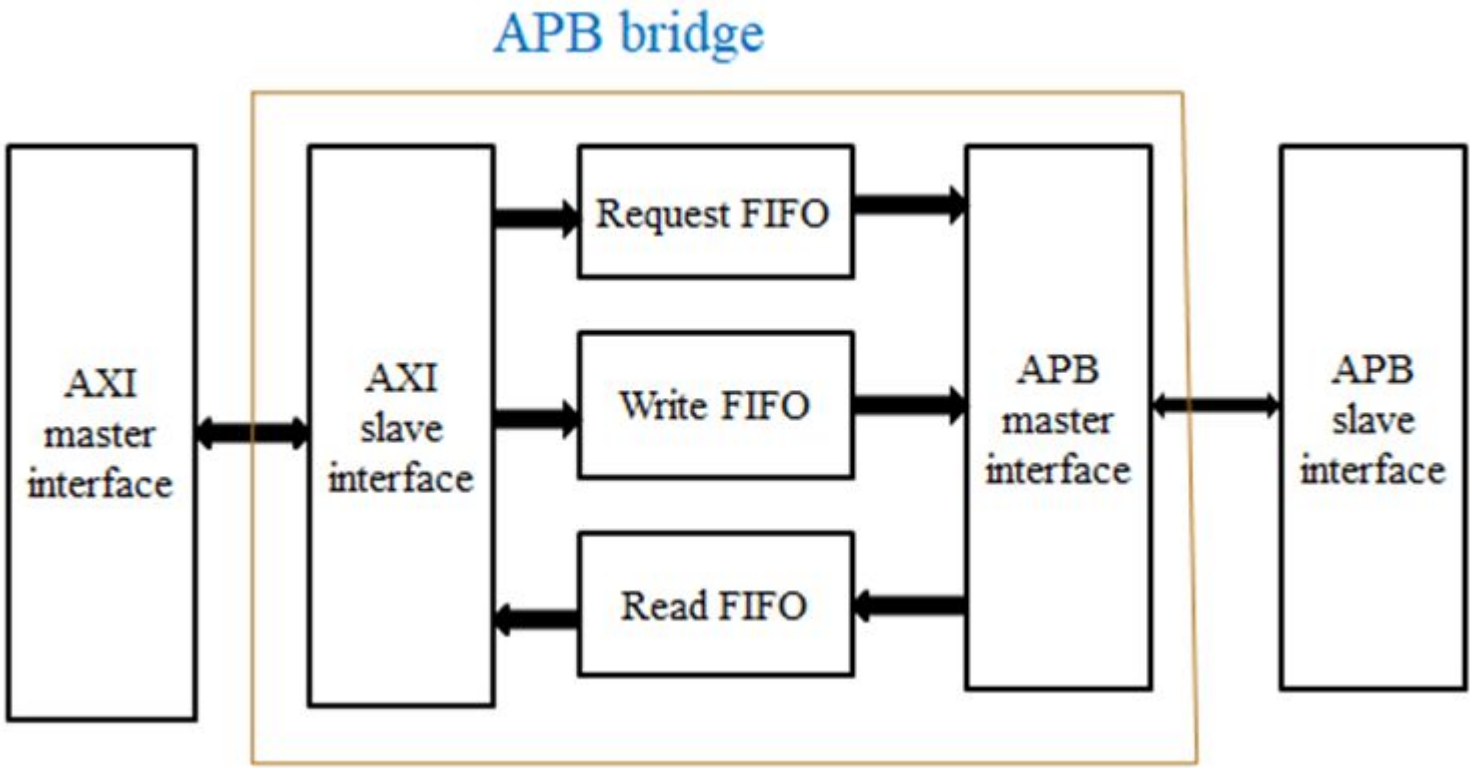
PROBLEMS STILL REMAIN

- Program jums after 6 instructions in interrupt handler for now added 6 nops to cover this issue

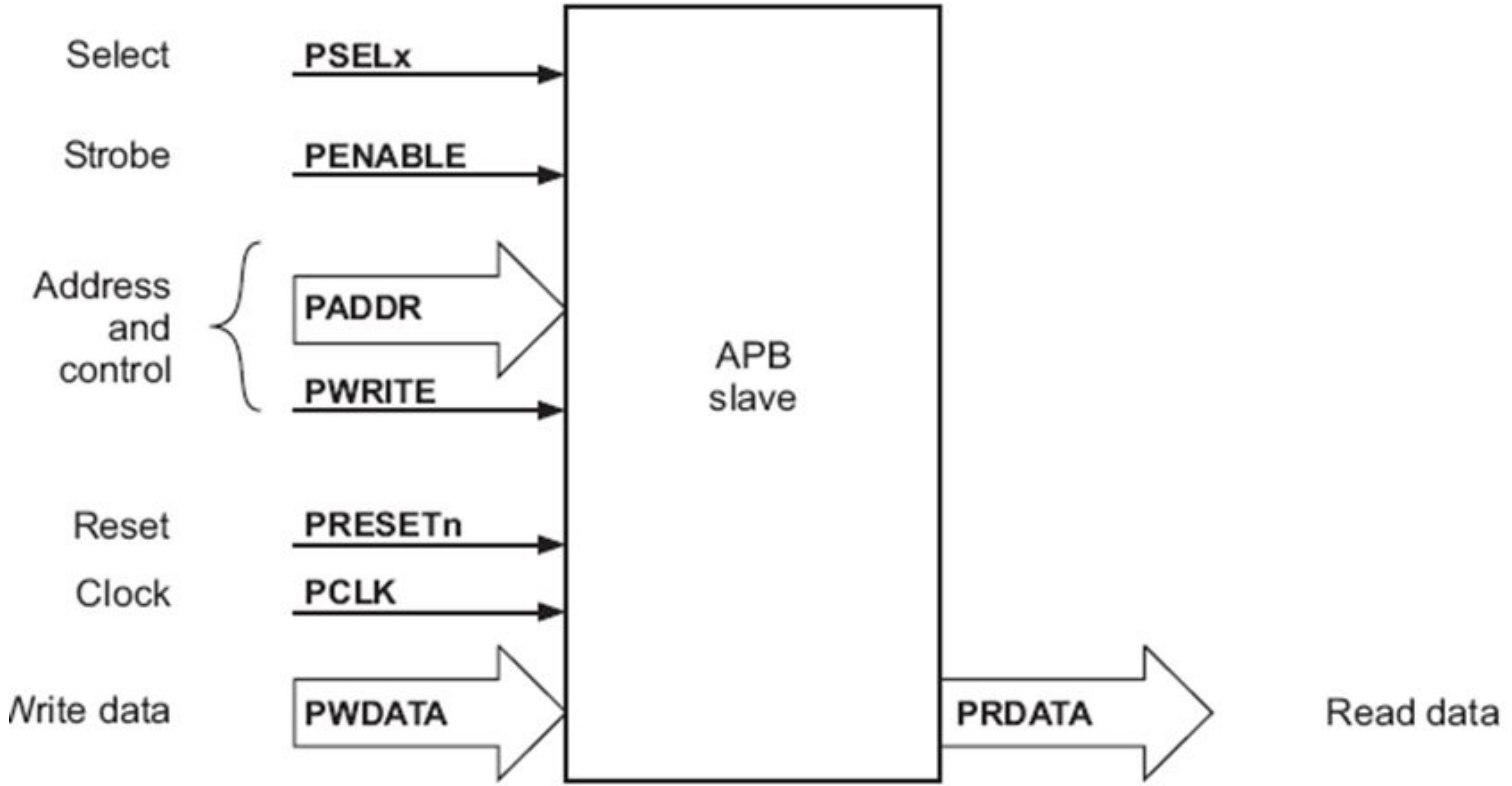
How to configure PIC

- First of all you need to disable the external interrupts by writing

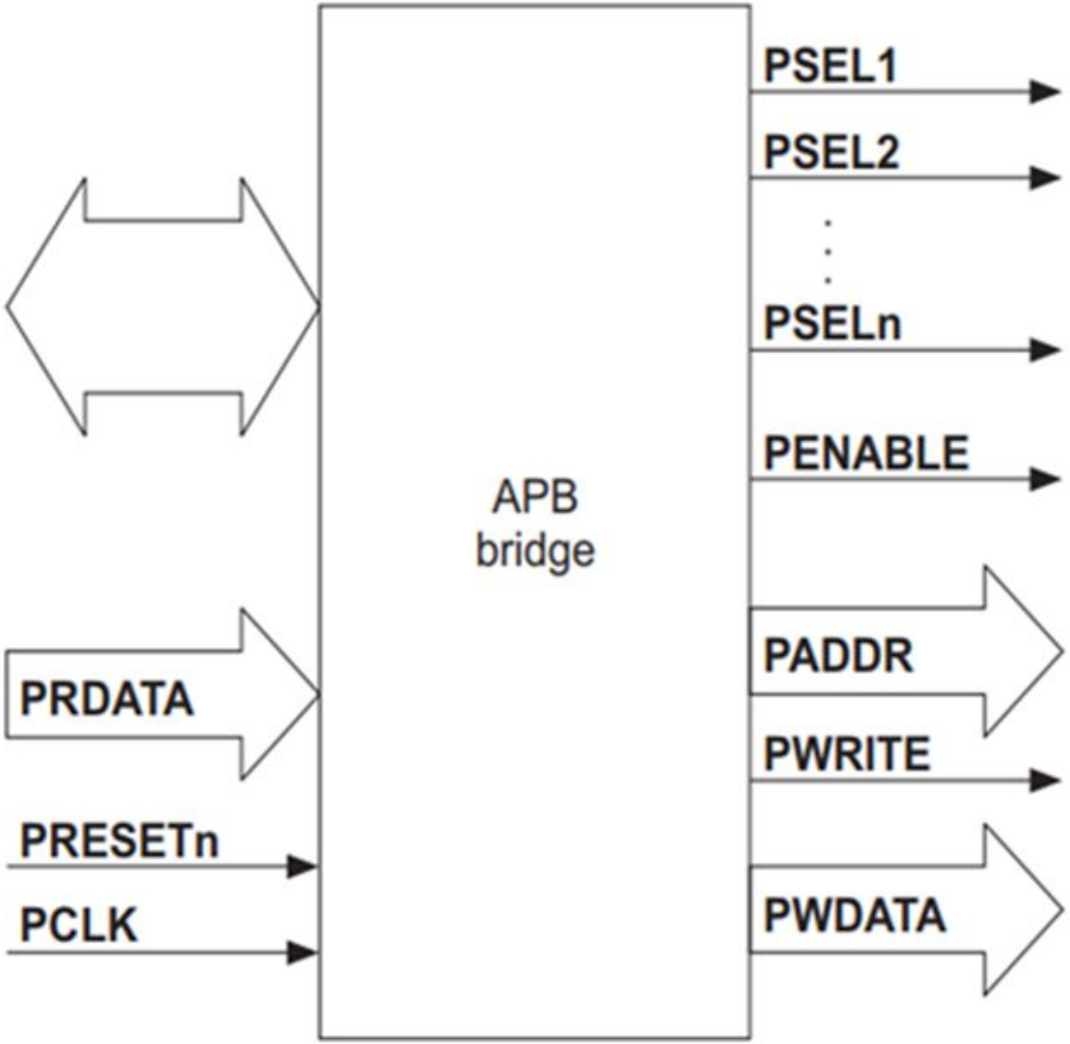
APB Interconnect



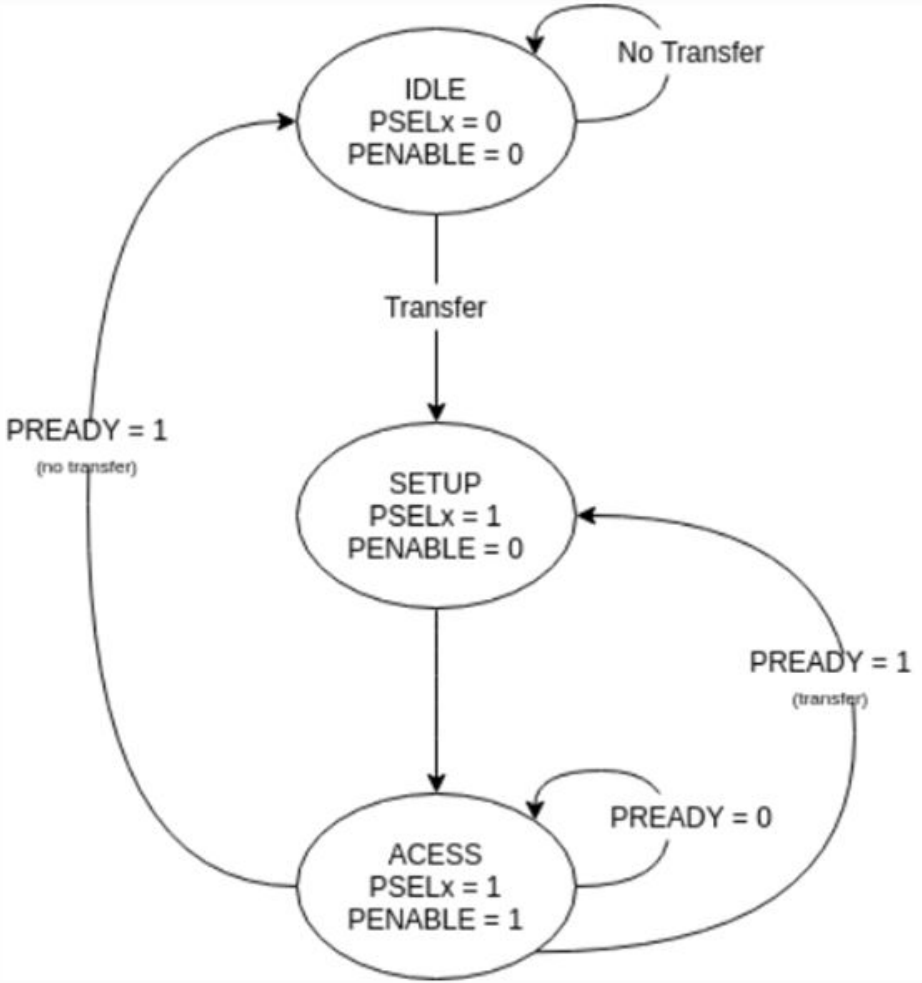
APB Slave


Select
PSELx
Strobe
PENABLE
Address
and
control
PADDR
PWRITE
APB
slave
Reset
PRESETn
Clock
PCLK
Write data
PWDATA
PRDATA
Read data

APB Master


PSEL1
PSEL2
PSELn
PENABLE
APB
bridge
PADDR
PRDATA
PWRITE
PRESETn
PWDATA
PCLK

Operational Mode Of Master Transactor



Operational Mode Of Slave Transactor

Reset = 0	State = IDLE, Port = Initial, Ready = 0
Reset = 1, Enable = 0	State = IDLE, Ready = 0
Reset = 1, Enable = 1	State = SET, Ready = 1
In Set State If ready = 1	New State = Access (in this state select port)
Ready = 0, Enable = 1	Wait State

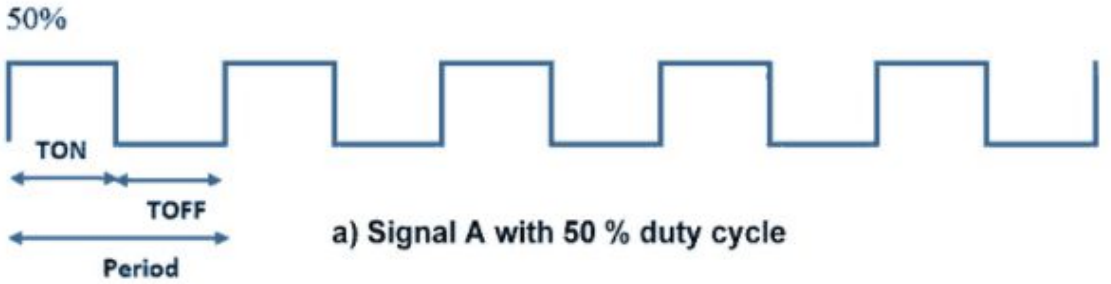
APB Signals Of PWM

- PSEL, if psel = 1 then the peripheral is selected
- PENABLE, if penable = 1, it enables the peripheral
- WRITE , if write= 1 , data is written to the peripheral and if write = 0 the it reads the data from the peripheral.
- ADDR , Address carrier
- PREADY, transfers ready input
- PSLVERR , error detection

Pulse Width Modulation

- **Pulse Width Modulation (PWM)** is a technique by which width of a pulse is varied while keeping the frequency constant
- A period of a pulse consists of an **ON** cycle (HIGH) and an **OFF** cycle (LOW). The fraction for which the signal is ON over a period is known as **duty cycle**.

$$\text{Duty Cycle (In \%)} = \frac{Ton}{Ton + Toff} \text{ x } 100$$



a) Signal A with 50 % duty cycle

b) Signal B with 10% duty cycle

c) Signal C with 30% duty cycle

d) Signal D with 70% duty cycle

PWM Duty Cycle Waveforms

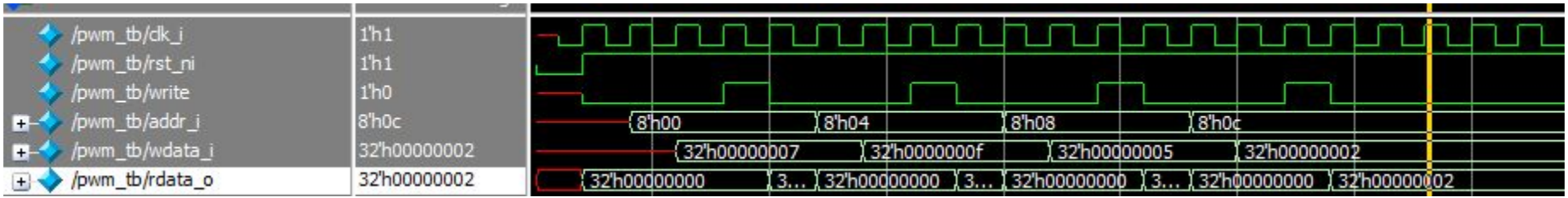
PWM Register Map

```
//////////////control logic///////
parameter   adr_ctrl_1    =   0,
            adr_divisor_1=   4,
            adr_period_1 =   8,
            adr_DC_1     =   12;

parameter   adr_ctrl_2    =   16,
            adr_divisor_2=   20,
            adr_period_2 =   24,
            adr_DC_2     =   28;
```

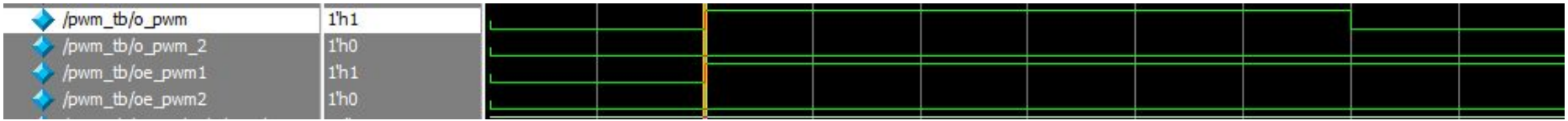
- Address 0x0, writes **wdata[2:0]** to ctrl.
- Address 0x16, writes **wdata[2:0]** to ctrl2.
- Address 0x4, writes **wdata[15:0]** to divisor.
- Address 0x20, writes **wdata[15:0]** to divisor2.
- Address 0x8, writes **wdata[15:0]** to period.
- Address 0x24, writes **wdata[15:0]** to period2.
- Address 0x12,writes **wdata[15:0]** to DC_1.
- Address 0x28,writes **wdata[15:0]** to DC_2.

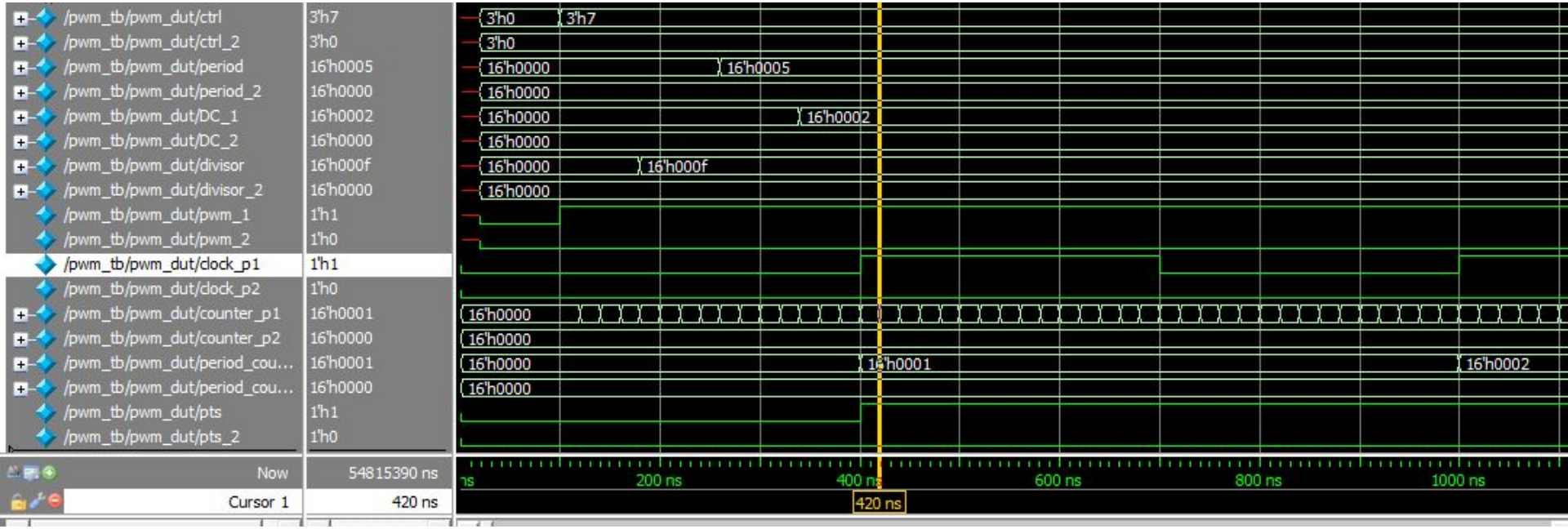
PWM Waveform

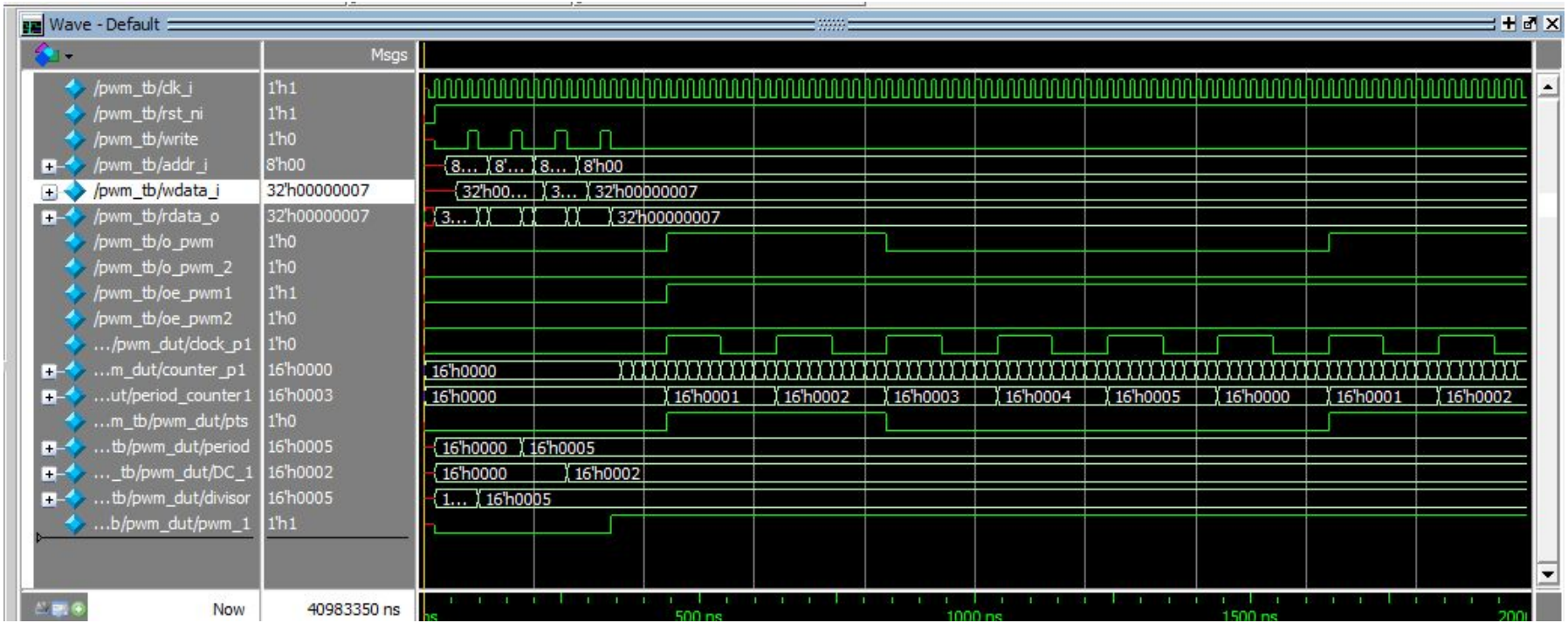
Inputs



Output



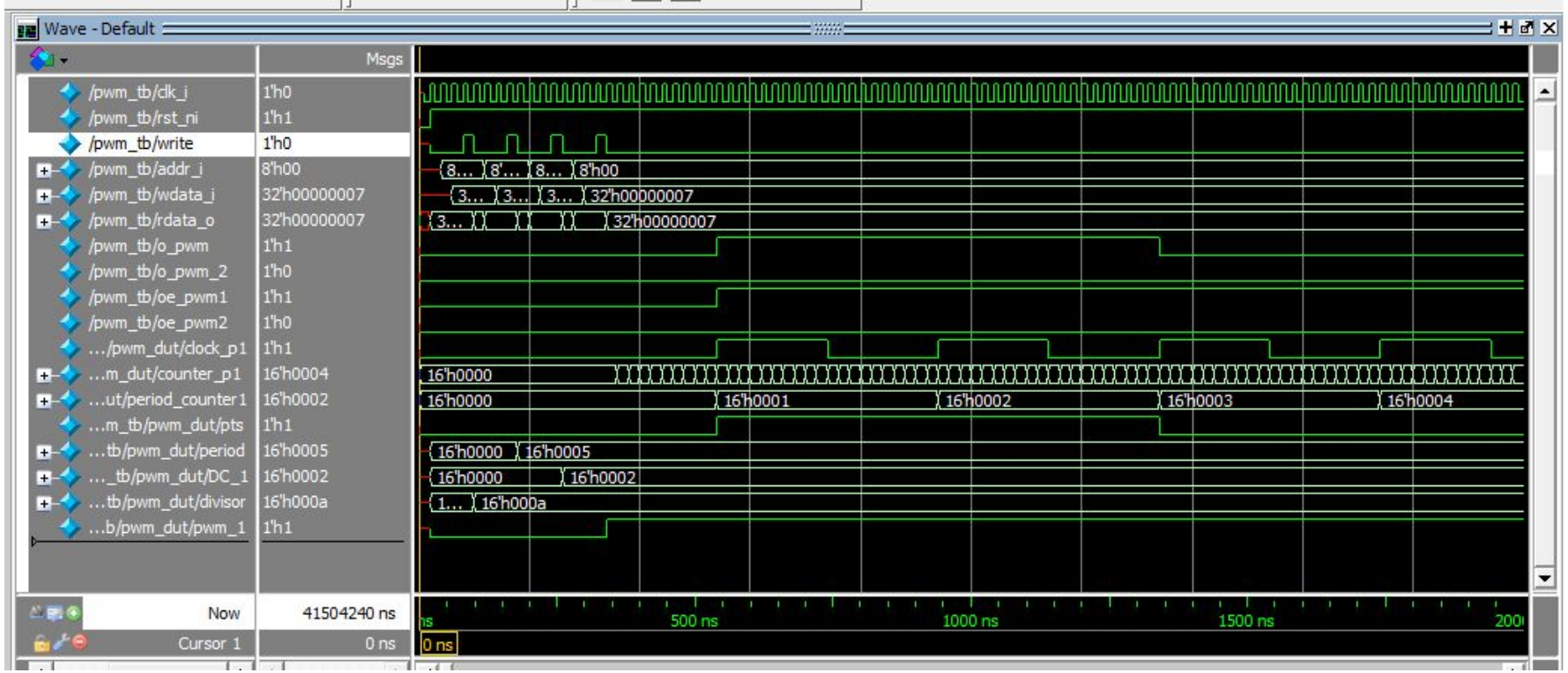

/pwm_tb/pwm_dut/ctrl
/pwm_tb/pwm_dut/ctrl_2
/pwm_tb/pwm_dut/period
/pwm_tb/pwm_dut/period_2
/pwm_tb/pwm_dut/DC_1
/pwm_tb/pwm_dut/DC_2
/pwm_tb/pwm_dut/divisor
/pwm_tb/pwm_dut/divisor_2
/pwm_tb/pwm_dut/pwm_1
/pwm_tb/pwm_dut/pwm_2
/pwm_tb/pwm_dut/clock_p1
/pwm_tb/pwm_dut/clock_p2
/pwm_tb/pwm_dut/counter_p1
/pwm_tb/pwm_dut/counter_p2
/pwm_tb/pwm_dut/period_cou...
/pwm_tb/pwm_dut/period_cou...
/pwm_tb/pwm_dut/pts
/pwm_tb/pwm_dut/pts_2
3'h7
3'h0
16'h0005
16'h0000
16'h0002
16'h000f
1'h1
1'h0
16'h0001
Now
54815390 ns
Cursor 1
420 ns
200 ns
400 ns
600 ns
800 ns
1000 ns



Divisor = 5

Period = 5

DC = 2



Divisor = 10

Period = 5

DC = 2

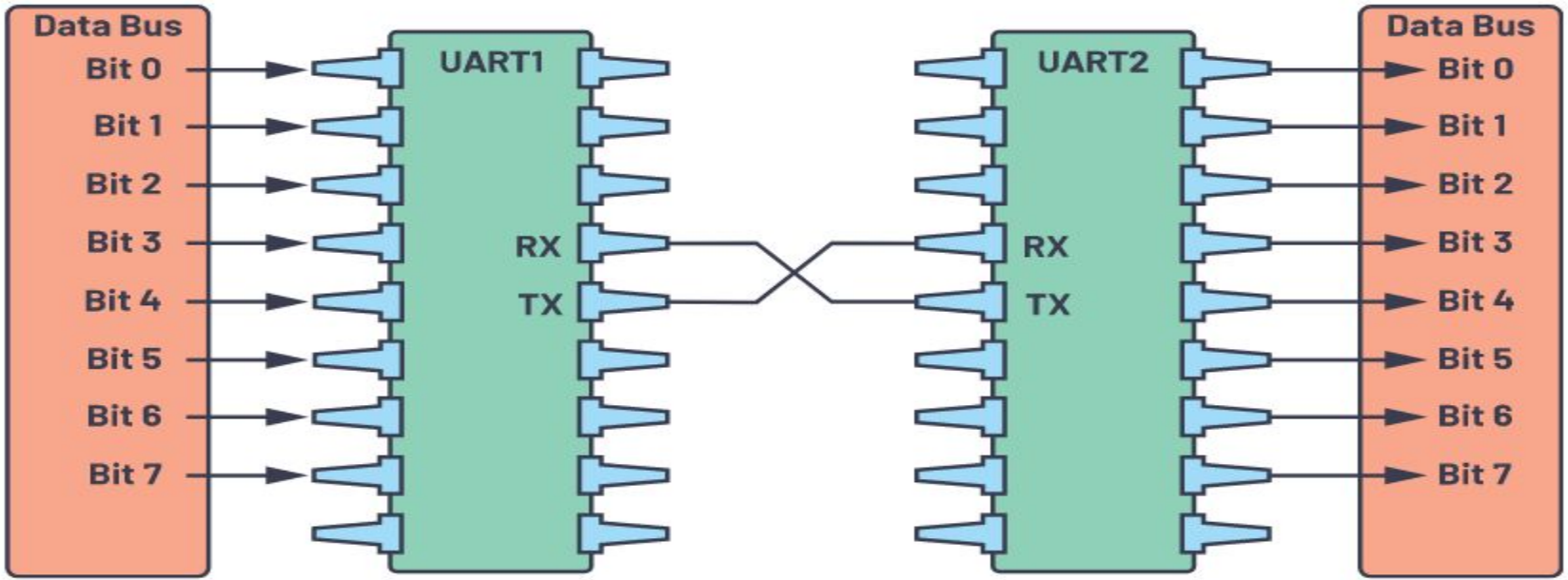
UART

- UART is one of the most simple and most commonly used Serial Communication techniques.
- The main reason for integrating the UART hardware in to microcontrollers is that it is a serial communication and requires only two wires for communication.

contd.

- UART is a serial communication device that performs parallel – to – serial data conversion at the transmitter side
- And serial – to – parallel data conversion at the receiver side.
- It is universal because the parameters like transfer speed, data speed, etc. are configurable.
- Synchronized on baud clock

UART With Data Bus



UART Application

- Bluetooth module
- USB
- GPS

UART REGISTERS

```
localparam ADDR_CTRL = 0;
localparam ADDR_TX   = 4;
localparam RX_EN     = 12;
localparam TX_EN     = 16;
localparam RX_STATUS = 20;
localparam RX_SC     = 24;
```

These are the registers that are used
To set the clks_per_bit ADDR_CTRL will be used and the address will be 0 in this case. To set the clks_per_bits wdata[15:0] will be assigned the value.

To set the value to transfer address will be 4 i.e. ADDR_TX. We will assign the 8 bits of data to wdata[7:0] that needs to be transferred.

To enable the transfer address will be 16 i.e. TX_EN. We will assign 1 bit to wdata[0].

To enable the transfer address will be 12 i.e. RX_EN. We will assign 1 bit to wdata[0].

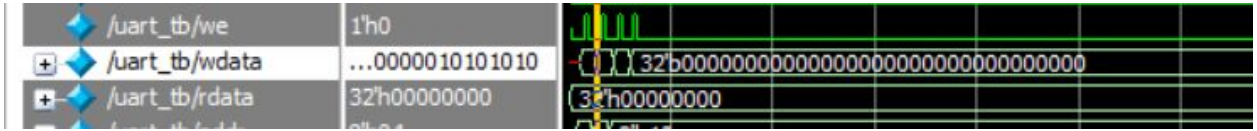
When the address is 20 i.e RX_STATUS, done signal is transferred to the output.

When the address is 24 i.e RX_SC,We will assign 0 to wdata[0] to clear the status register.
To set the value to transfer address will be 4 i.e. ADDR_TX. We will assign the 8 bits of data to wdata[7:0] that needs to be transferred.

TRANSFERRING DATA IN UART

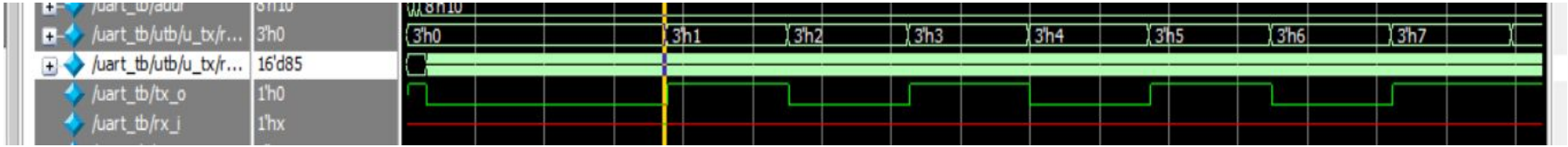
Data to be transferred is 10101010.

Data will be transferred in LSB order which will be 01010101

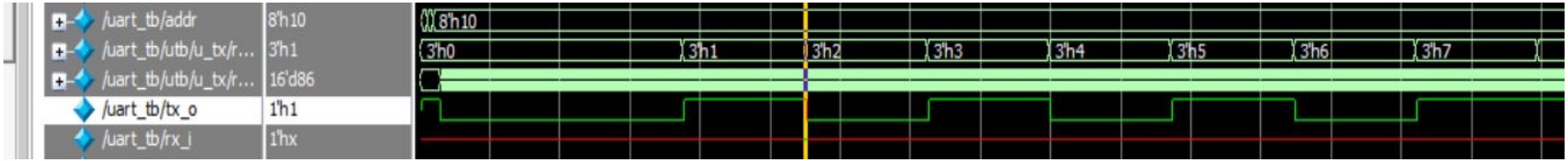


First bit transferred i.e. 0 which is the start bit and is transferred in 87 clocks as the CLK_PER_BIT = 87 then clock_count resets at index = 0

Second bit is transferred that is first bit of data i.e.0 (87 cycles) at index = 1



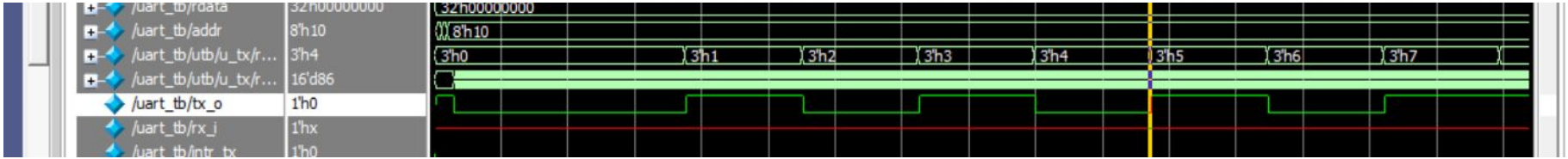
Then after 87 cycles third bit is transferred i.e. 1 at index = 1 (second bit of data)



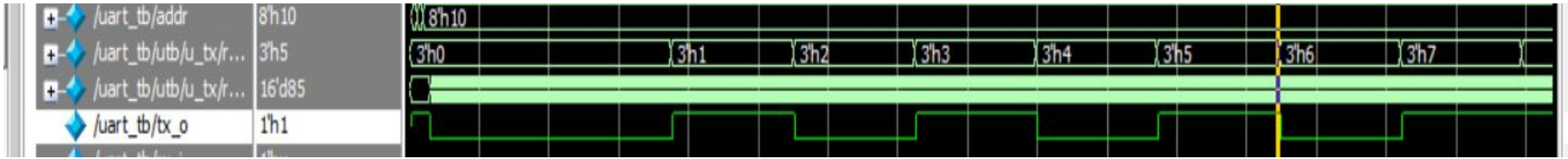
Then after 87 cycles fourth bit is transferred i.e. 0 at index = 2 (third bit of data)

Then after 87 cycles fifth bit is transferred i.e. 1 at index = 3 (fourth bit of data)

Then after 87 cycles sixth bit is transferred i.e. 0 at index = 4 (fifth bit of data)

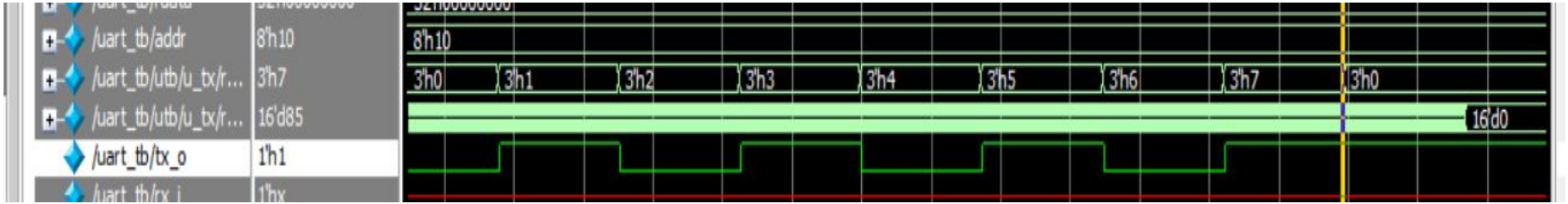


Then after 87 cycles seventh bit is transferred i.e. 1 at index = 5 (sixth bit of data)

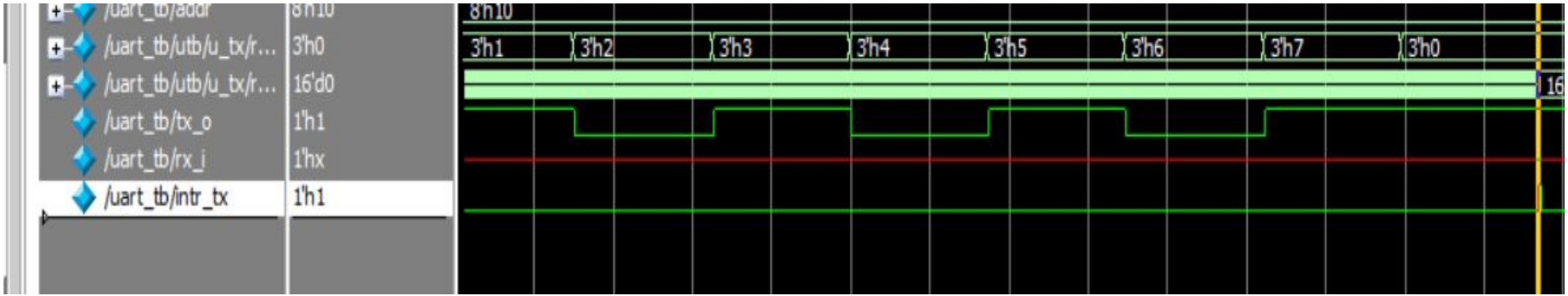


Then after 87 cycles eight bit is transferred i.e. 0 at index = 6 (seventh bit of data)

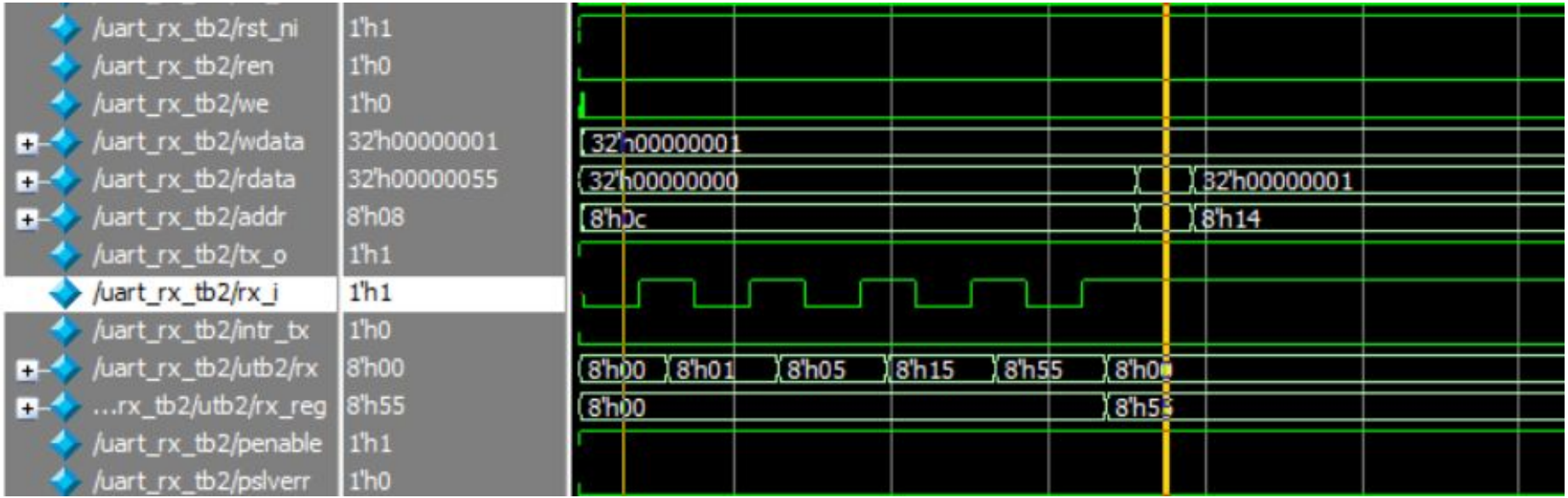
Then after 87 cycles ninth bit is transferred i.e. 1 at index = 7 (eight bit of data) and index reset as all the data is transferred.



Then after 87 cycles last bit is transferred i.e. 1 which is stop bit and at this point intr_tx is set high as it indicates that the data is transferred and stopped.

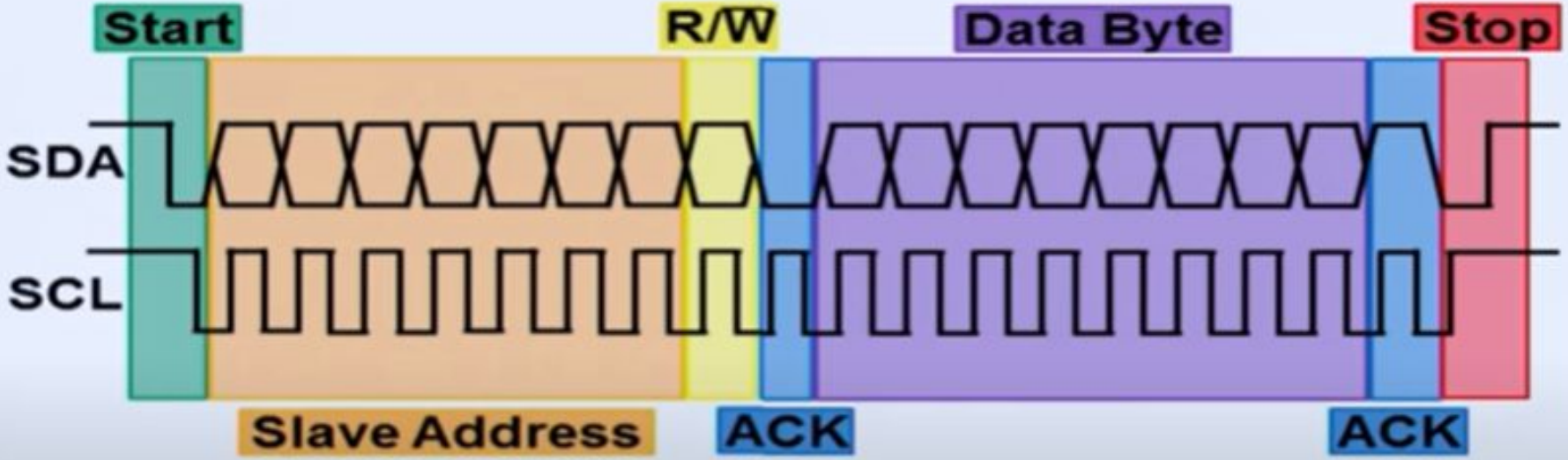


RECEIVING DATA IN UART



MICROCHIP
MASTERs 2016

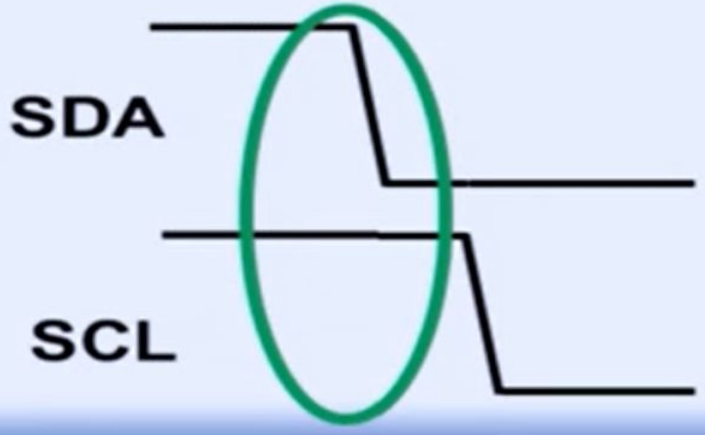
I²C Waveform Example



MICROCHIP
MASTERs 2016

Start Condition

- Initiated from Bus Idle (SDA = 1, SCL = 1)
- SDA goes Low (0) while SCL is High (1)



MICROCHIP
MASTERs 2016

Slave Address

- **Follows Start Condition**
- **7-bits or 10-bits**
  - Fixed Address specified in Datasheet
  - Address Lines (i.e. A2, A1, A0)
- **Each device on the same I^2C bus needs a unique Slave Address**

MICROCHIP
MASTERs 2016

Read/$\overline{\text{Write}}$ (R/$\overline{\text{W}}$) Bit

- Follows the Slave Address (8th Clock Cycle)
- Informs the Slave if the Master wants to Read or Write.
  - R/$\overline{\text{W}}$ bit = 1 means Read; R/$\overline{\text{W}}$ bit = 0 means Write

MICROCHIP
MASTERs 2016

Acknowledge Bit

- **Every 9th Clock Cycle**
  - 0 = ACK (Acknowledged)
  - 1 = NACK (Not Acknowledged)
- **Bi-directional**
  - Slave pulls down SDA to acknowledge the Master
  - Master pulls down SDA to acknowledge the Slave

MICROCHIP
MASTERs 2016

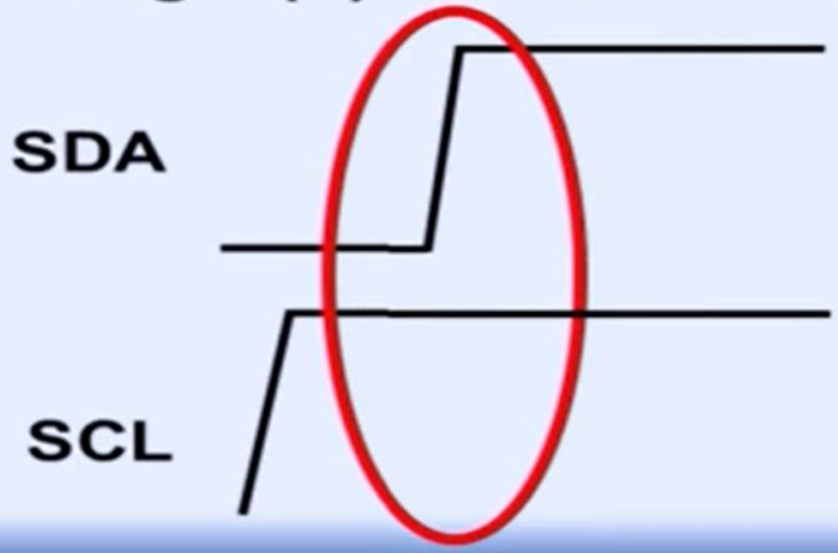
Data Byte

- **Data Byte Contents**
  - EEPROM Address or Register Address
  - Data to be written to the Slave (Write)
  - Data from the Slave (Read)

MICROCHIP
MASTERs 2016

Stop Condition

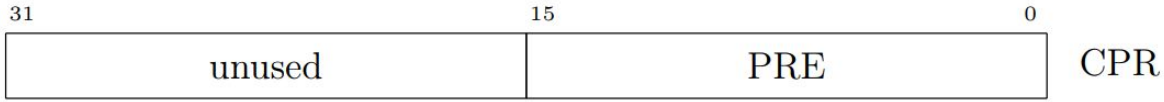
- SDA goes High (1) while SCL is High (1)



I2C SIGNALS

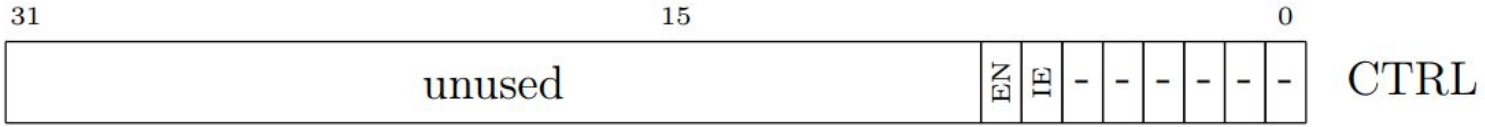
Signal	Direction
scl_pad_i	input
scl_pad_o	output
scl_padoen_o	output
sda_pad_i	input
sda_pad_o	output
sda_padoen_o	output
interrupt_o	output

PRE-SCALER REGISTER



`Bit 15:0` **PRE**: Prescaler.
Sets the clock prescaler by the value in PRE to achieve the desired I^2C clock by dividing the current system clock by the given factor.

CONTROL REGISTER



`Bit 7` **EN**: Enable.
Enable the I^2C peripheral.

`Bit 6` **IE**: Interrupt enable.
Enable interrupts.

`Bit 5:0` **Reserved**: Set to 0.

STATUS REGISTER

`Bit 7` **RXA**: Acknowledge from sent data.

`Bit 6` **BUS**: Bus is busy.

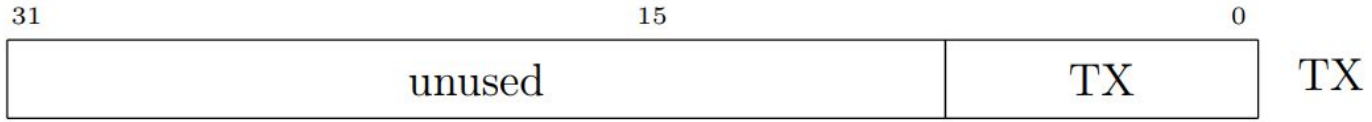
`Bit 5` **AL**: Arbitration lost.

`Bit 4:2` **Reserved**: Set to 0.

`Bit 1` **TIP**: Transfer in progress.

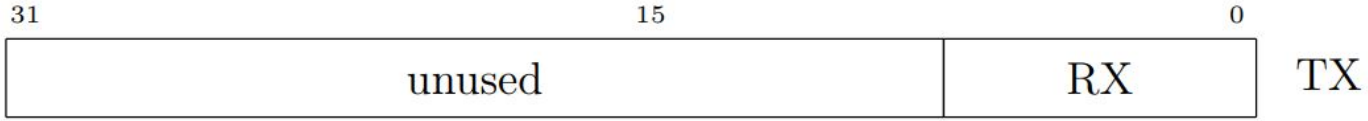
`Bit 0` **IRQ**: Interrupt received.
This flag is always set when transmission has finished or bus arpitration was lostm, regardless of whether interrupts are enabled or not. This flag can possibly polled and is cleared by writing 1 to the IA command register.

TRANSMIT REGISTER



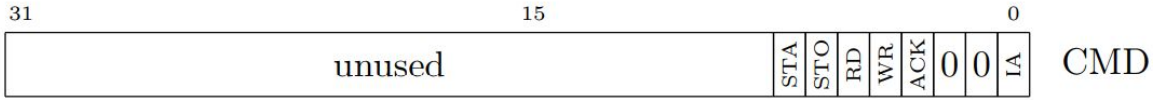
Bit 7:0 **TX**: Transmit Register

RECEIVE REGISTER



Bit 7:0 **RX**: Receive Register

Command Registers



`Bit 7` **STA**: Send start bit.

`Bit 6` **STO**: Send stop bit.

`Bit 5` **RD**: Read from bus.

`Bit 4` **WR**: Write to bus.

`Bit 3` **ACK**: Acknowledge received data.

`Bit 2:1` **Reserved**: Set to 0.

`Bit 0` **IA**: Interrupt Acknowldge.
Set to one to acknowledge interrupt. Cleared when transmission is done or arbitration is lost.

Verification Plan (Kinza) --->(2 months plan)

FIRST STEP:

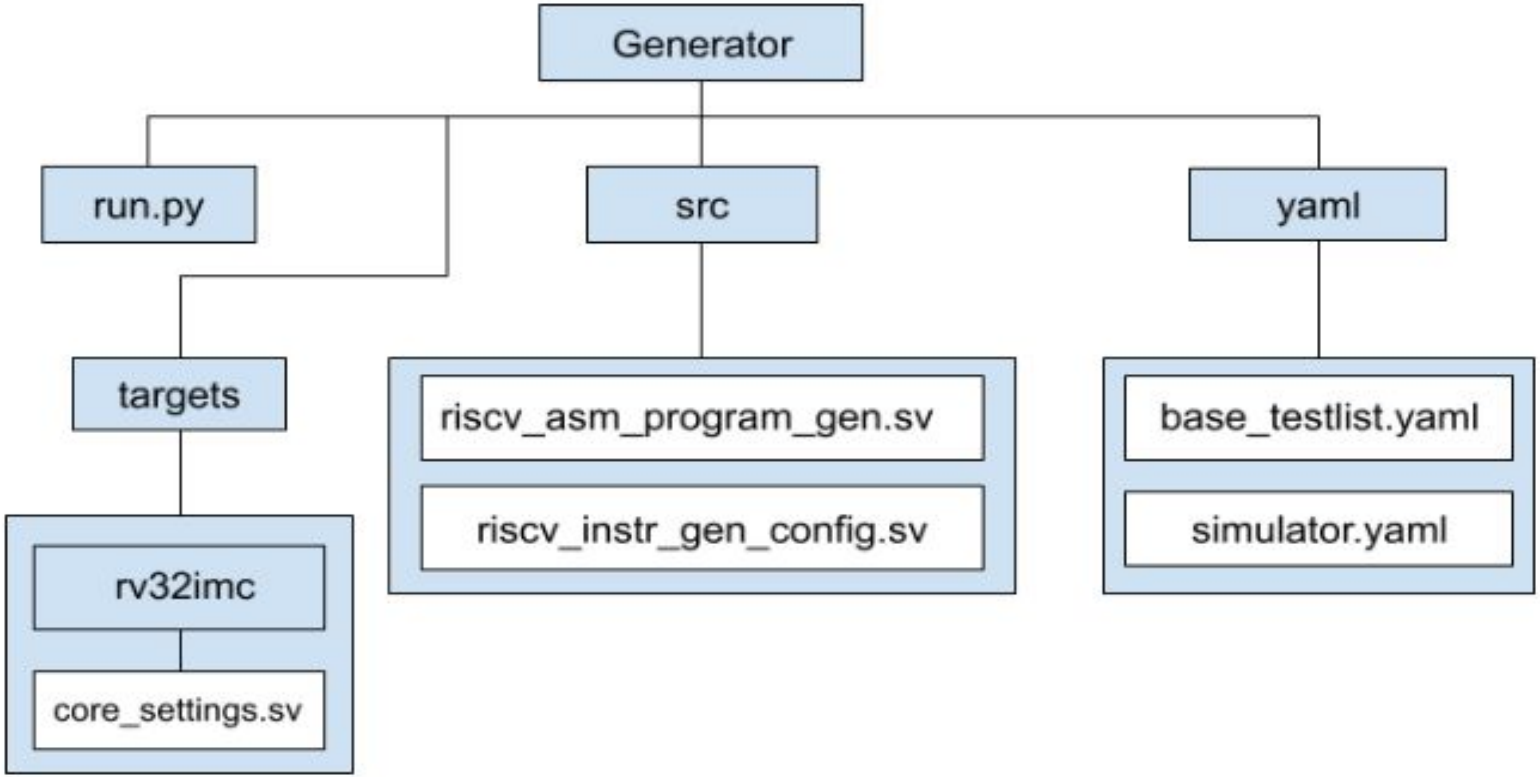
➔ Run the DV on SweRV-EL2

Changes in python script should get done in order to send program.hex from DV to core

SECOND STEP:

➔ Setup UVM for SweRV-EL2 between DV and core.

Directory structure (contd.)



run.py

- ➔ A python script, runs the generator.
- ➔ Consist of functions
- ➔ Functions in this script acts as steps to run the generator.
  - ◆ Compilation of generator
  - ◆ Instruction generation
  - ◆ Run the random test on ISS
  - ◆ Run the directed assembly test on ISS
  - ◆ Run the directed c test on ISS
  - ◆ Seed generation
  - ◆ Compilation of assembly test
  - ◆ Run ISS simulation with the generated test program
  - ◆ Comparison of two ISS logs
  - ◆ Comparison of ISS results
  - ◆ Save the regression report

Yaml directory

testlist.yaml

- ➔ This file include the test name,generator test,RTL test, and generator options corresponding to generator.
- ➔ Separate csr testlist and coverage testlist

```yaml
test: riscv_arithmetic_basic_test
description: >
  Arithmetic instruction test, no load/store/branch instructions
gen_opts: >
  +instr_cnt=10000
  +num_of_sub_program=0
  +directed_instr_0=riscv_int_numeric_corner_stream,4
  +no_fence=1
  +no_data_page=1
  +no_branch_jump=1
  +boot_mode=m
  +no_csr_instr=1
iterations: 2
gen_test: riscv_instr_base_test
rtl_test: core_base_test
```

contd.

Simulator.yaml

➔ This file include the compilation and simulation commands corresponding to the simulator.

ISS.yaml

➔ This is similar to the simulator.yaml file except it contains the command for ISS

Targets directory

➔ The target directory has subdirectories containing core settings and test list based on the extension

riscv_core_settings.sv:

➔ It is the processor features configuration file. The specification of design should be declare here.

src

RISC-V assembly program generator

➔ This is the main class to generate a complete RISC-V program, including the init routine,instruction section, data section, stack section, page table, interrupt and exception handling etc.

Riscv_instr_gen_config

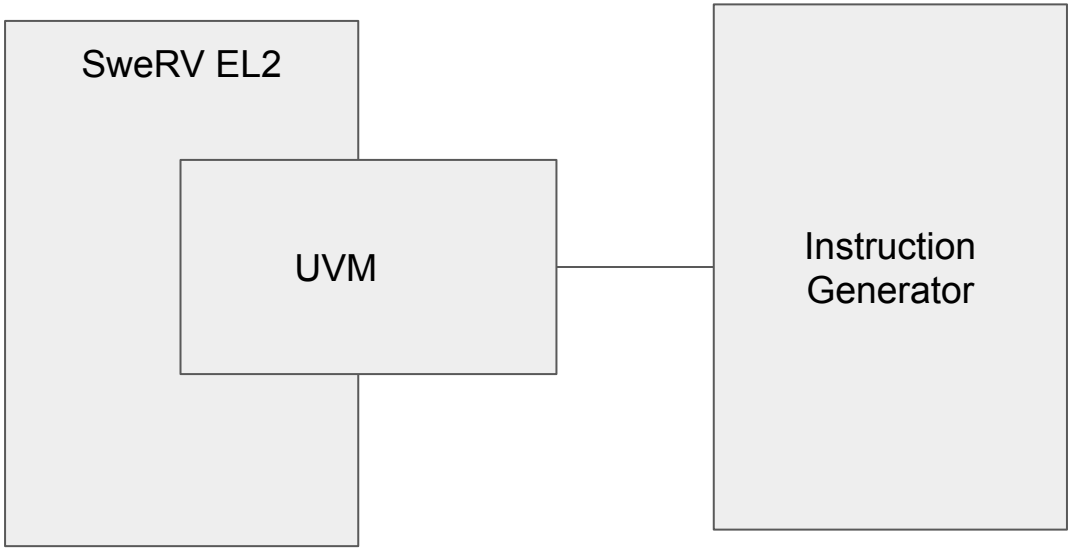
➔ Contains the configurations for assembly program/generator

Goal for verification of SweRV EL2 SoC

- ➔ Integration of SweRV El2 with generator.
  - First step is simple, we will replace the default program.hex with generated hex.
  - For that, changes should get done in run.py script.
  - Generate directed test through generator and execute that on core.
  - Generate random test through generator and execute that on core.

- ➔ Setup a separate UVM testbench and provide that as a RTL test to the generator.

contd.



Complete Program Flow