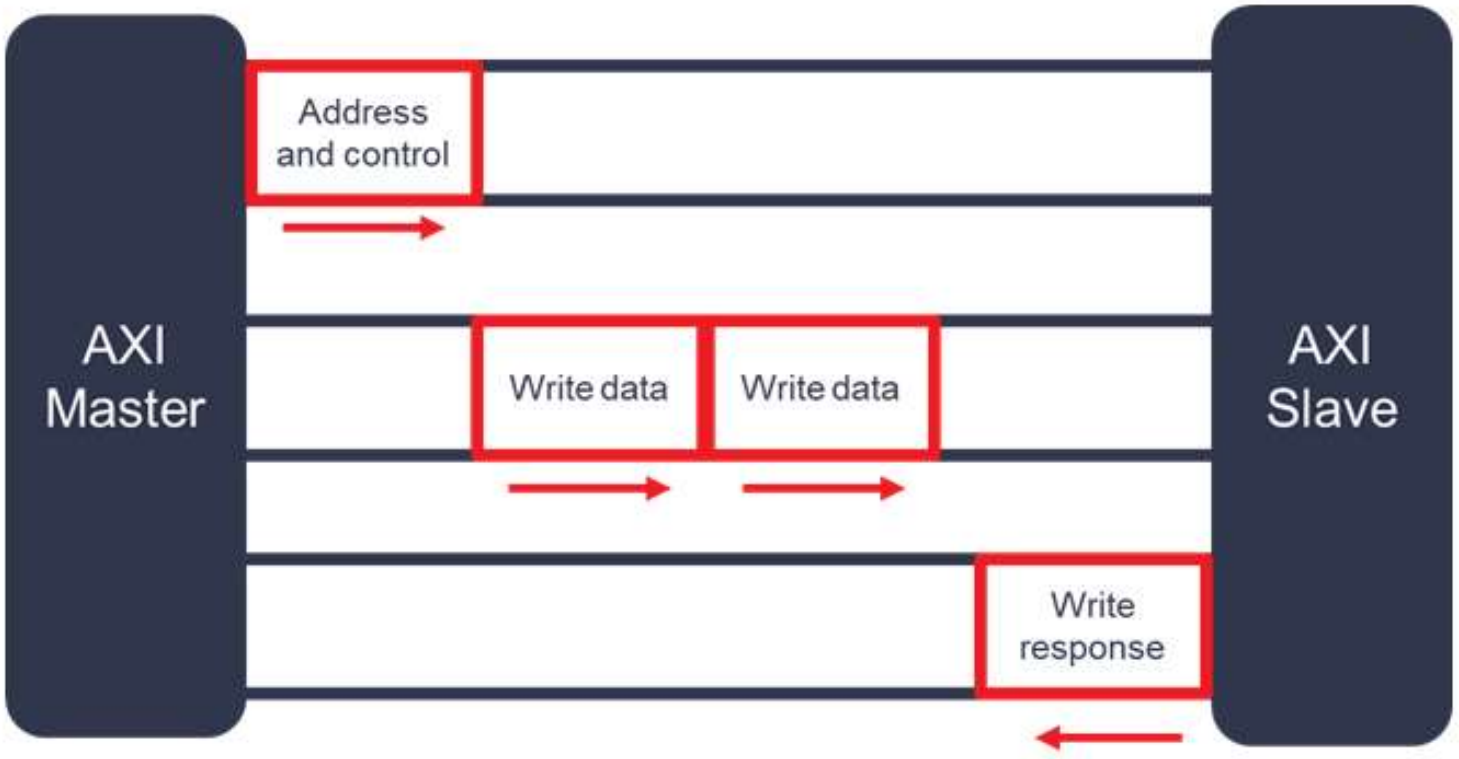
AXI

Vince

AMD
together we advance_

modified on 2023/11/24

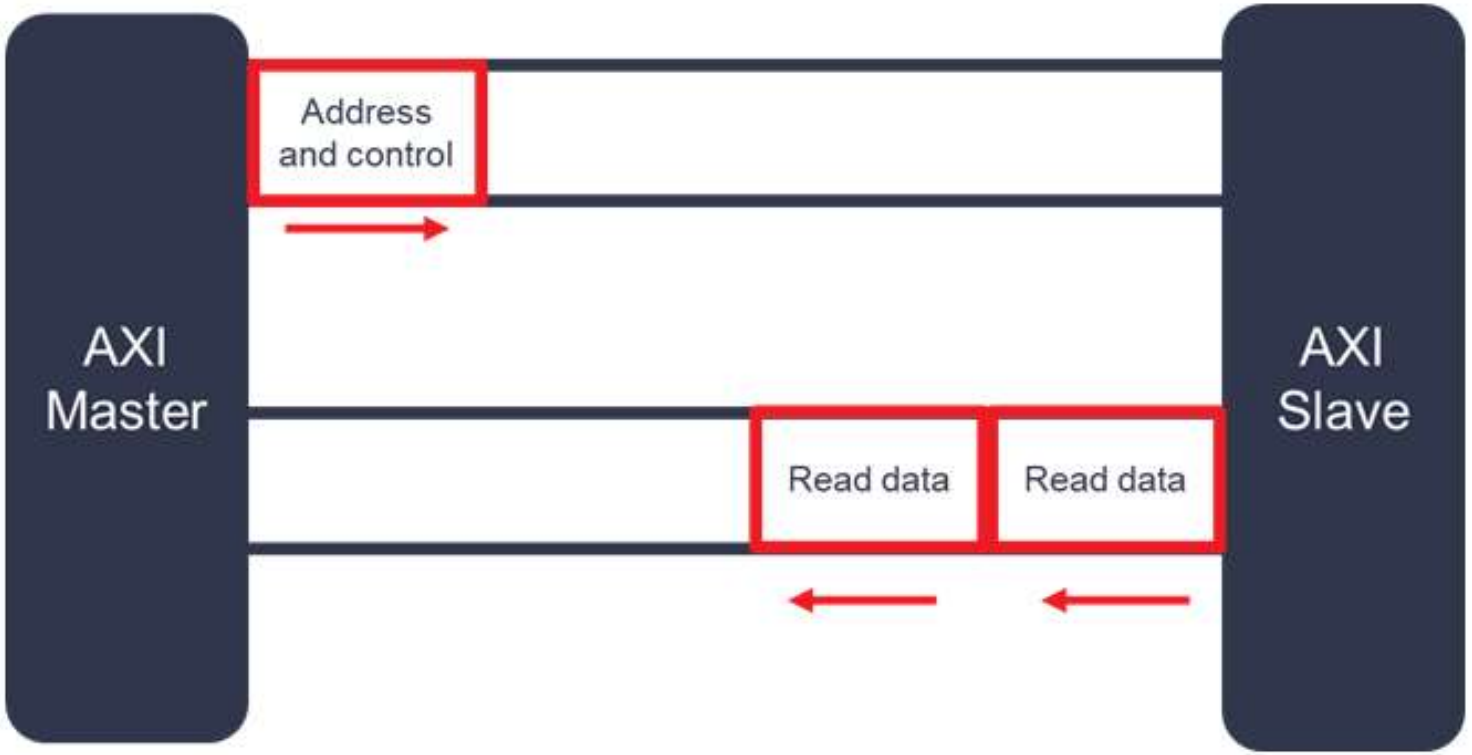
AXI Write Transactions



An AXI Write transactions requires multiple transfers on the 3 Read channels.

- **First**, the Address Write Channel is sent Master to the Slave to set the address and some control signals.
- **Then** the data for this address is transmitted Master to the Slave on the Write data channel.
- **Finally** the write response is sent from the Slave to the Master on the Write Response Channel to indicate if the transfer was successful

AXI Read Transactions



An AXI Read transactions requires multiple transfers on the 2 Read channels.

- **First**, the Address Read Channel is sent from the Master to the Slave to set the address and some control signals.
- **Then** the data for this address is transmitted from the Slave to the Master on the Read data channel.

Note, as per the figure below, there can be multiple data transfers per address. This type of transaction is called a burst.

AXI4 Interface requirements

- When a VALID (AxVALID/xVALID) signal is asserted, it must remain asserted until the rising clock edge after the slave asserts AxREADY/xREADY.
- the VALID signal of the AXI interface sending information must not be dependent on the READY signal of the AXI interface receiving that information
- However, the state of the READY signal can depend on the VALID signal
- A write response must always follow the last write transfer in the write transaction of which it is a part
- Read data must always follow the address to which the data relates
- A slave must wait for both ARVALID and ARREADY to be asserted before it asserts RVALID to indicate that valid data is available

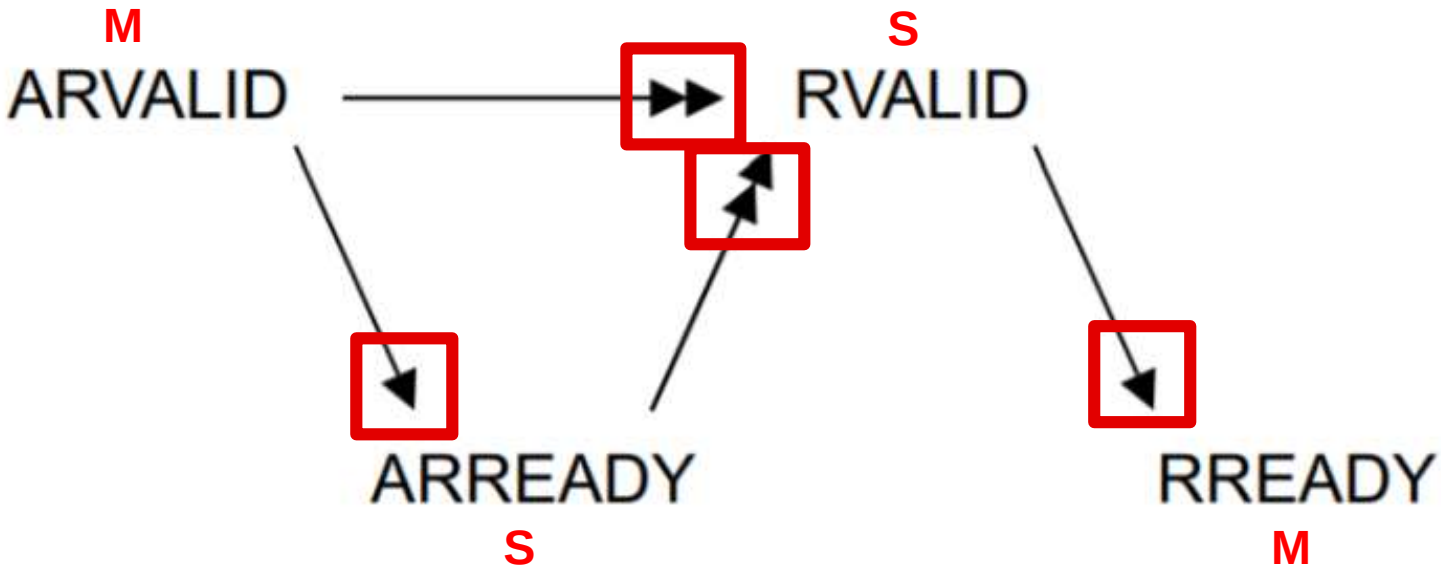
Channel Handshake Process

Write:
Addr **&** data **&** respond

Read:
Addr **&** data

Transaction channel	Handshake pair
Write address channel	AWVALID, AWREADY
Write data channel	WVALID, WREADY
Write response channel	BVALID, BREADY
Read address channel	ARVALID, ARREADY
Read data channel	RVALID, RREADY

Read(Read address & Read data Channel)

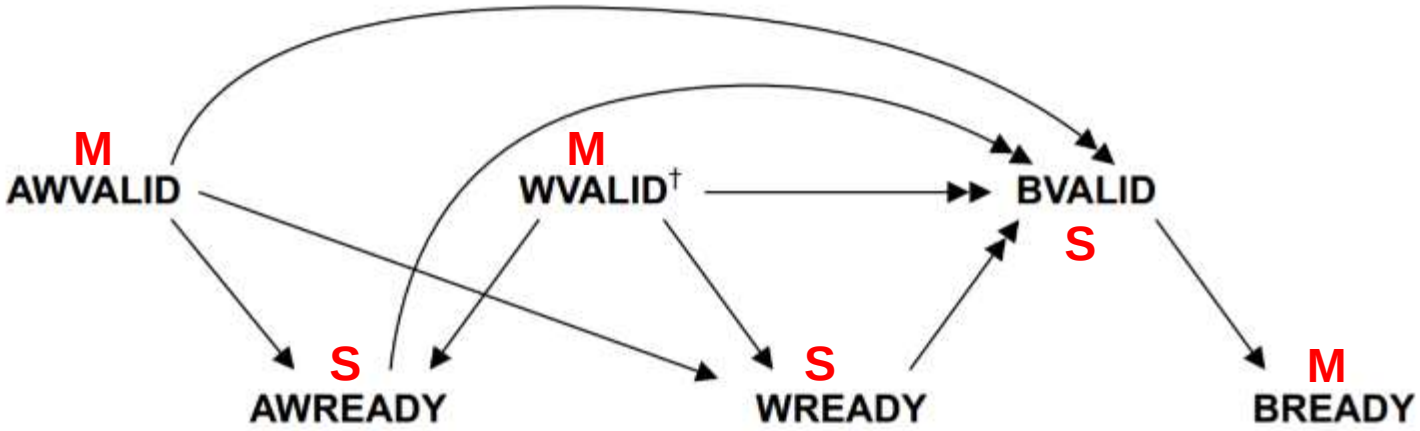


Read address channel	ARVALID, ARREADY
Read data channel	RVALID, RREADY

Valid : data is ready
Ready : can receive.

- The **slave device** can wait for the **ARVALID** signal to be acknowledged before asserting the **ARRENDY** signal.
- The **slave device** must wait for all **ARVALID** and **ARREADY** signals to be acknowledged before it can begin returning data by asserting the **RVALID** signal.

Write(Write address & Write data Channel)



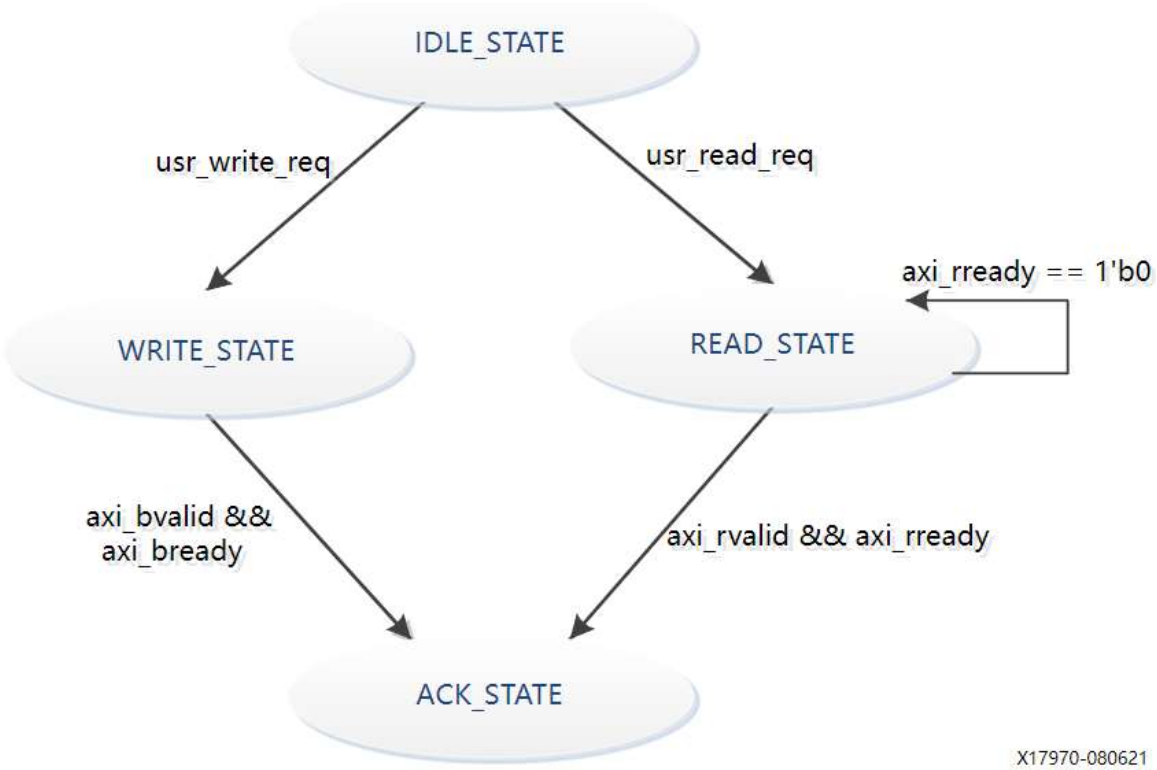
† Dependencies on the assertion of **WVALID** also require the assertion of **WLAST**

Write address channel	AWVALID, AWREADY
Write data channel	WVALID, WREADY
Write response channel	BVALID, BREADY

Valid : data is ready
Ready : can receive.

- The **master device** does not need to wait for the **slave device** to acknowledge the **AWREADY** or **WREADY** signal before asserting the **AWVALID** and **WVALID** signals.
- After acknowledging **AWERADY**, the **slave device** can wait for the **AWVALID** or **WVALID** signal, or both signals.
- After acknowledging **WERADY**, the **slave device** can wait for the **WVALID** or **WVALID** signal, or both signals.
- Before confirming **BVALID**, the **slave device** does not need to wait for the **master device** to confirm the **BREADY** signal.
- The **master device** can wait for the **BVAILD** signal before confirming **BREADY**.

AXI Lite FSM



A functional description of each state is described as below:

IDLE_STATE

By default, the FSM will be in IDLE_STATE. When the user_read_req signal becomes High, then it moves to READ_STATE else if user_write_req signal is High, **it moves to WRITE_STATE else it remains in IDLE_STATE**.

WRITE_STATE

You provide **S_AXI_AWVALID, S_AXI_AWADDR, S_AXI_WVALID**, S_AXI_WDATA, and S_AXI_WSTRB in this state to write to the register map through AXI. When S_AXI_BVALID and S_AXI_BREADY from AXI slave are High then it moves to ACK_STATE. If any write operation happens in any illegal addresses, the S_AXI_BRESP[1:0] indicates 2'b10 that asserts the write error signal.

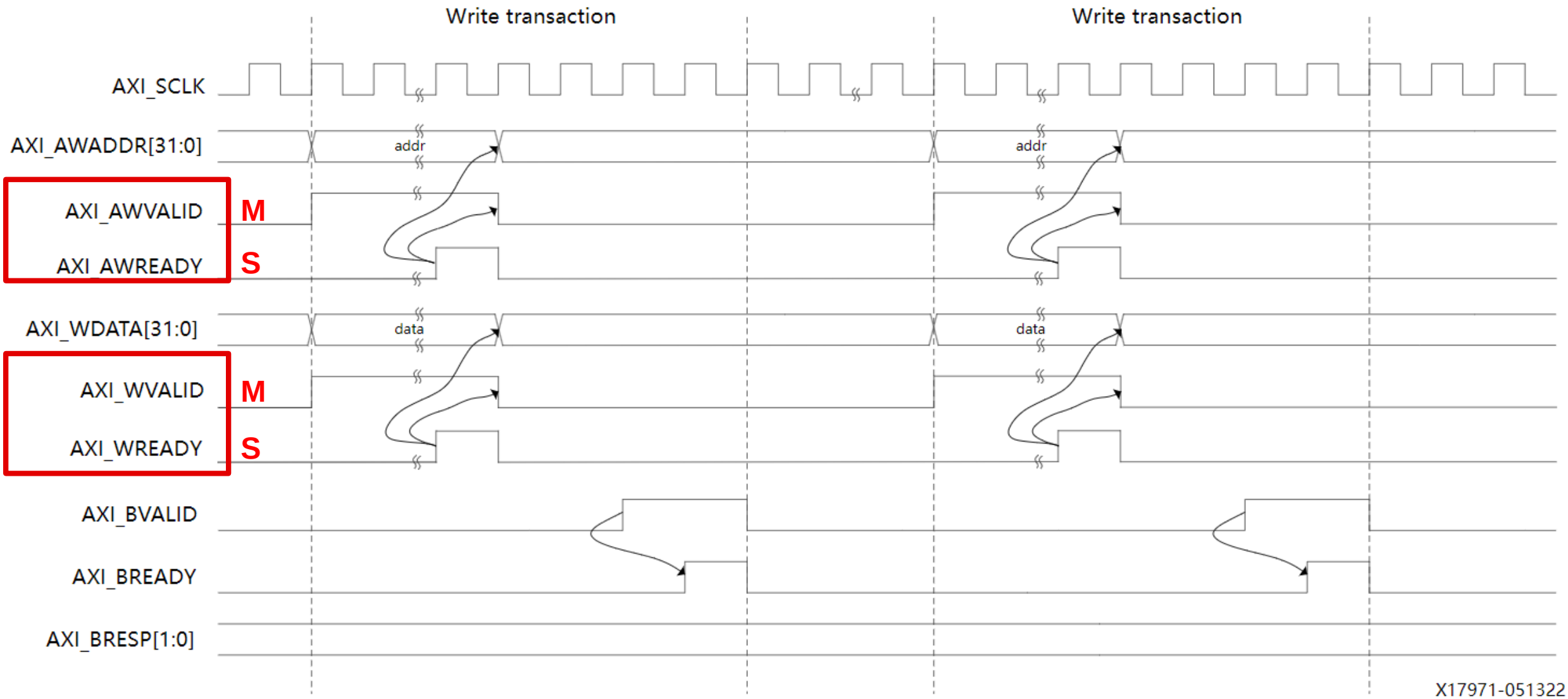
READ_STATE

You provide S_AXI_ARVALID and S_AXI_ARADDR in this state to read from the register map through AXI. When S_AXI_RVALID and S_AXI_RREADY are High then it moves to ACK_STATE. If any read operation happens from any illegal addresses, the S_AXI_RRESP[1:0] indicates 2'b10 that asserts the read error signal.

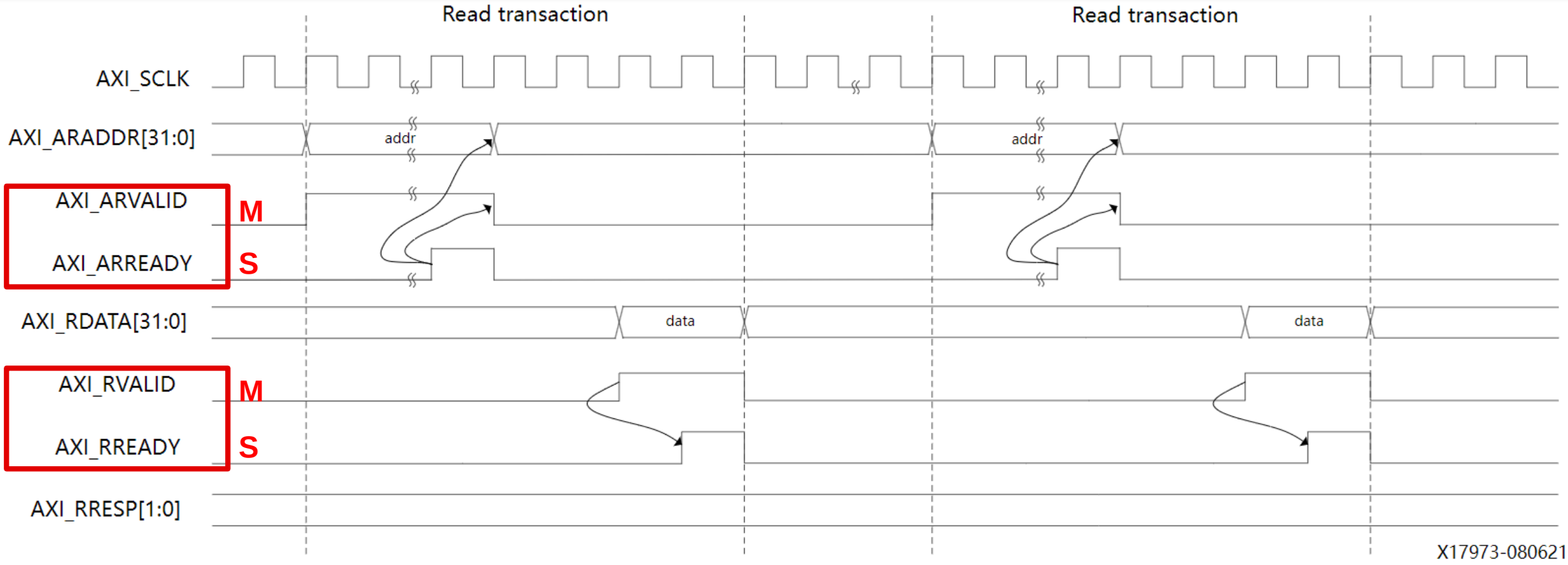
ACK_STATE

The state moves to IDLE_STATE

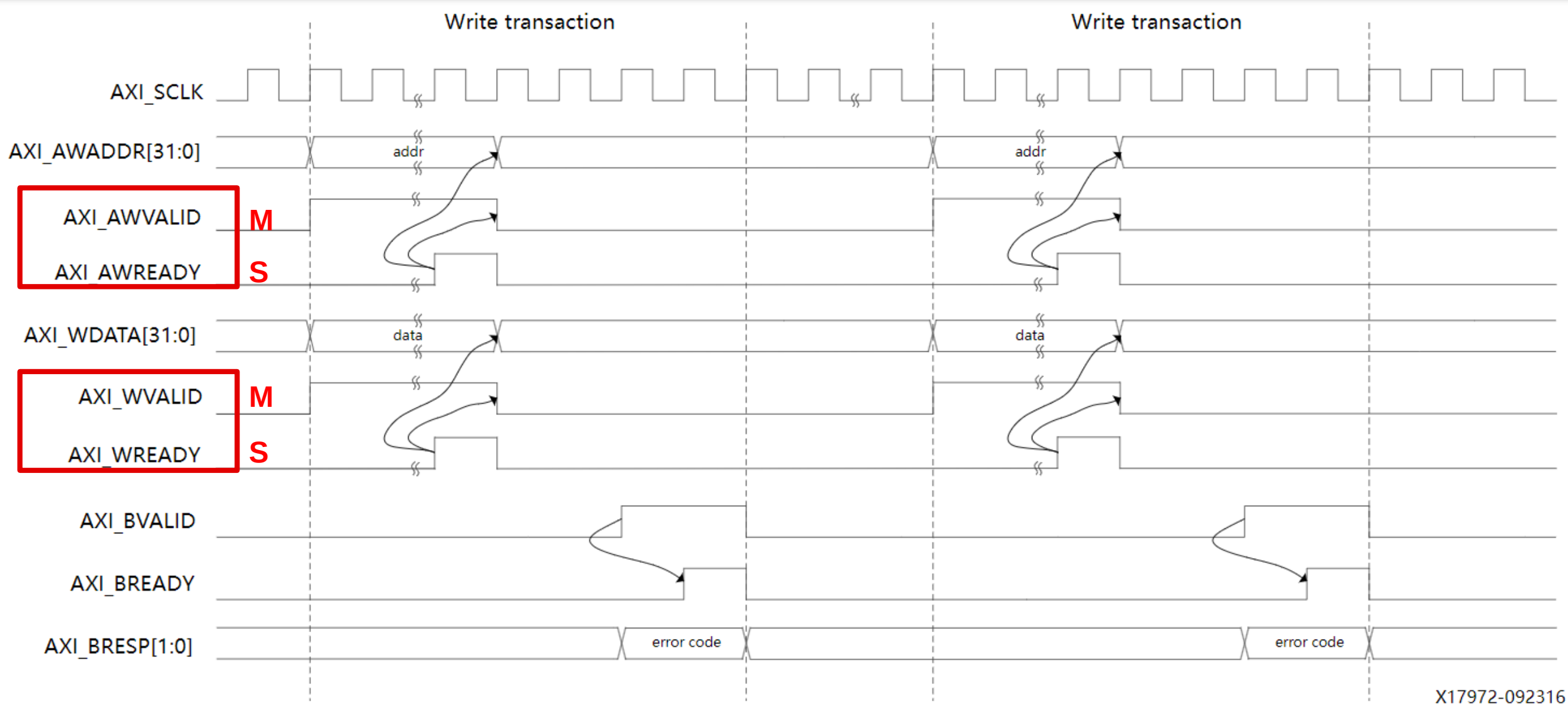
Valid Write Transactions
AXI4-Lite User Side Write Transaction



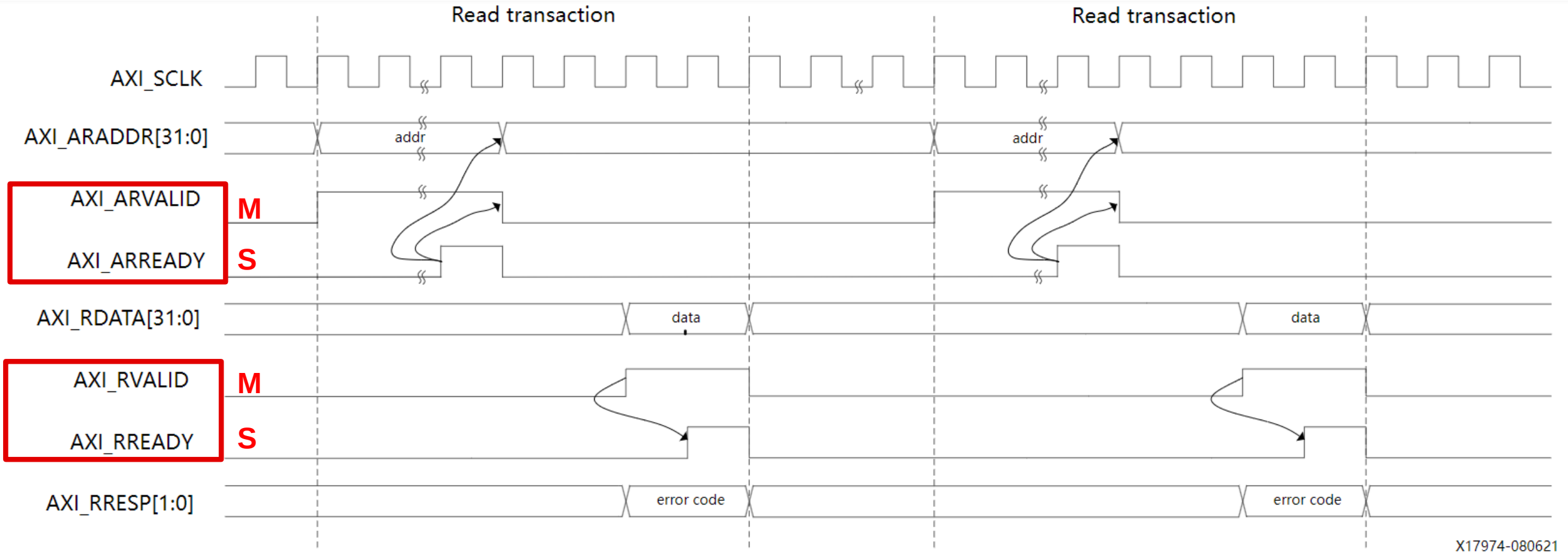
Valid Read Transactions
AXI4-Lite User Side Read Transaction



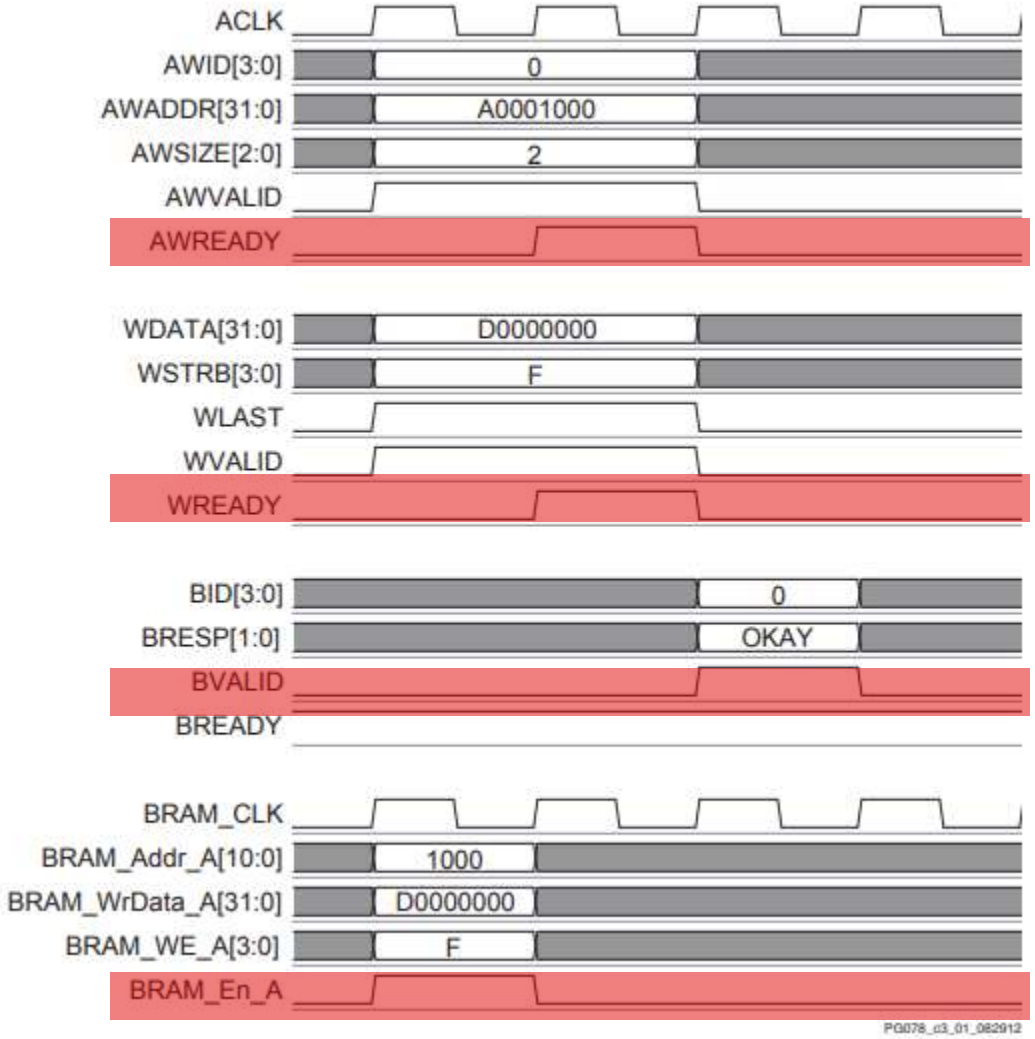
Invalid Write Transactions
AXI4-Lite User Side Write Transaction with Invalid Write



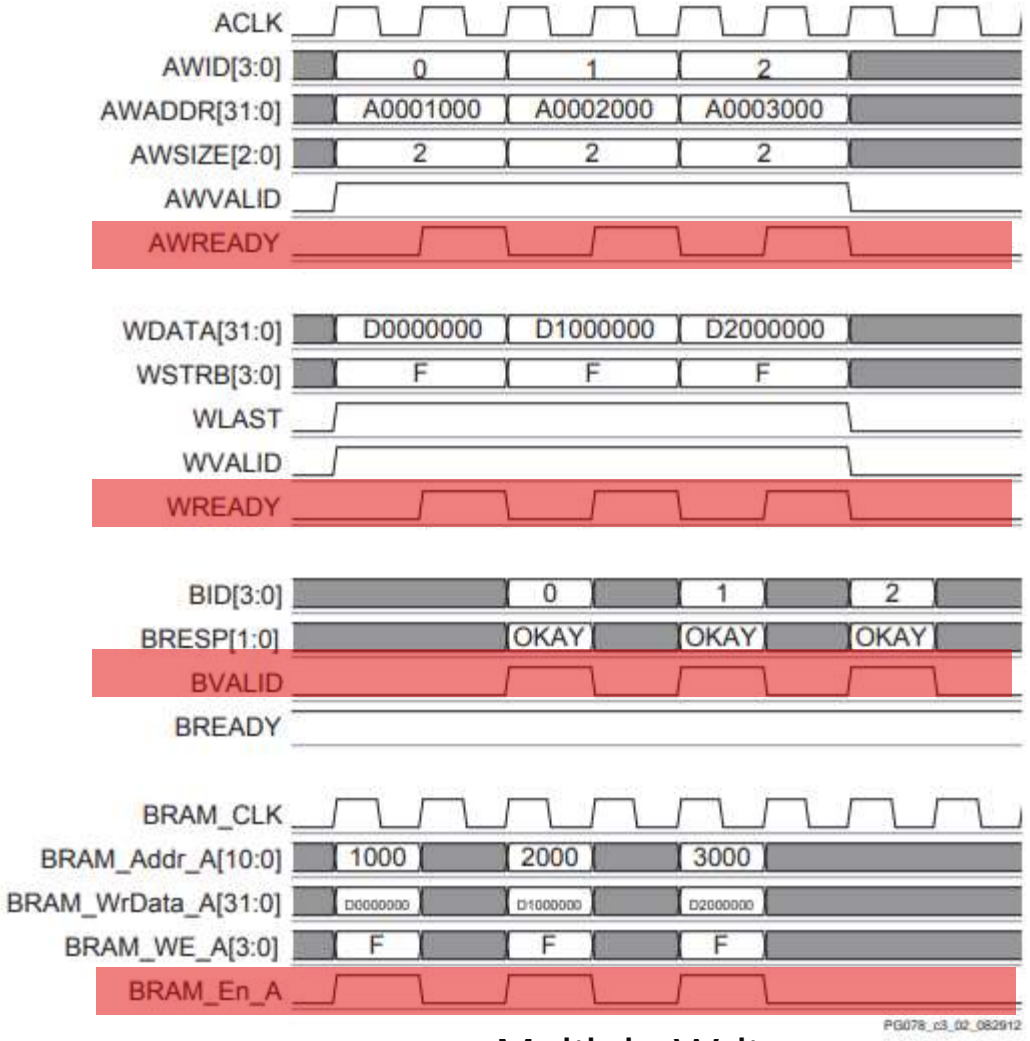
Invalid Read Transactions
AXI4-Lite User Side Read Transaction with Invalid Read Address



AXI4 – Lite ,Write

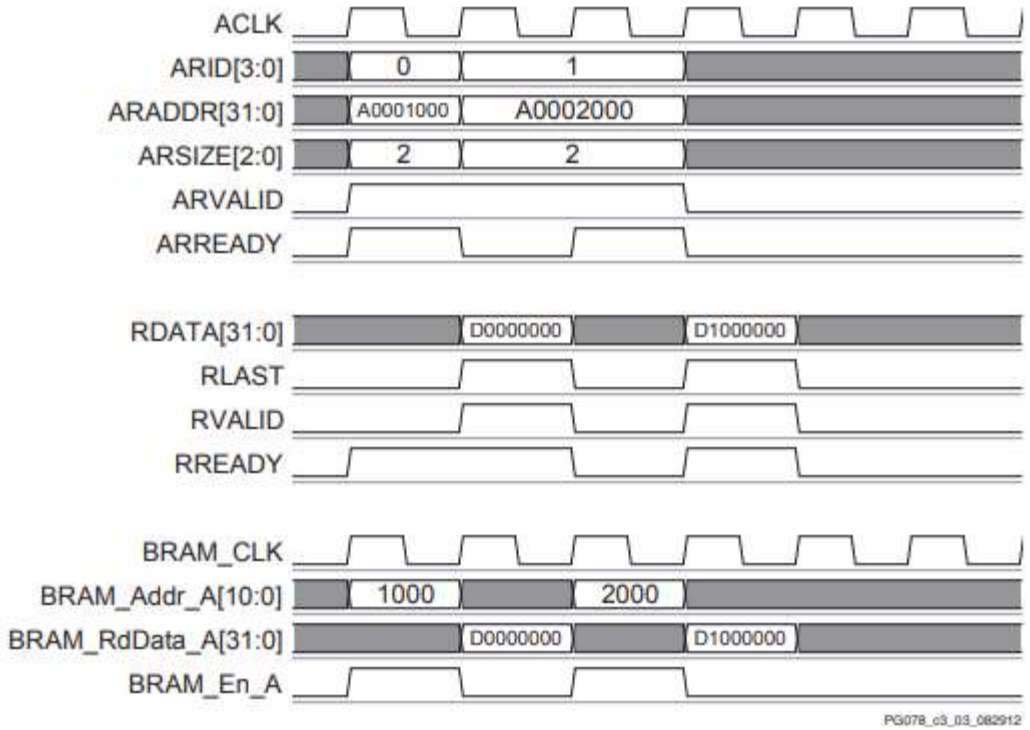


Single Write



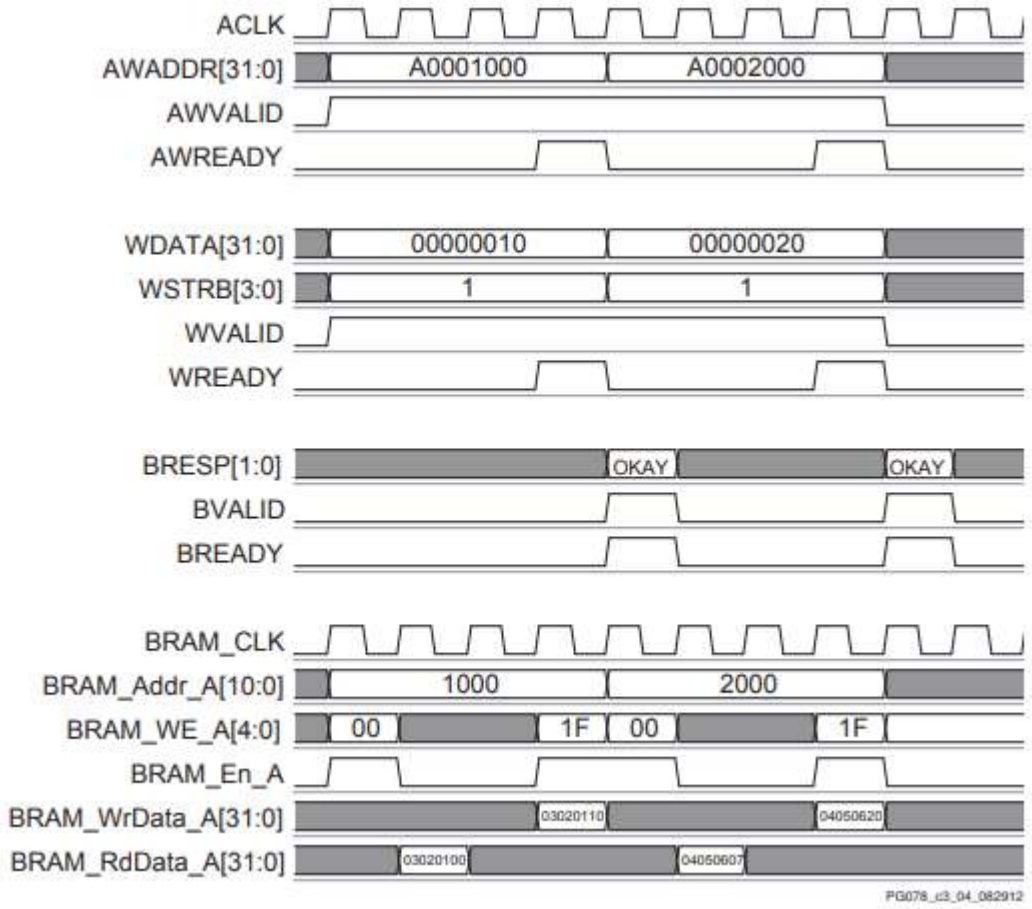
Multiple Write

AXI – Lite, Read



Single Read

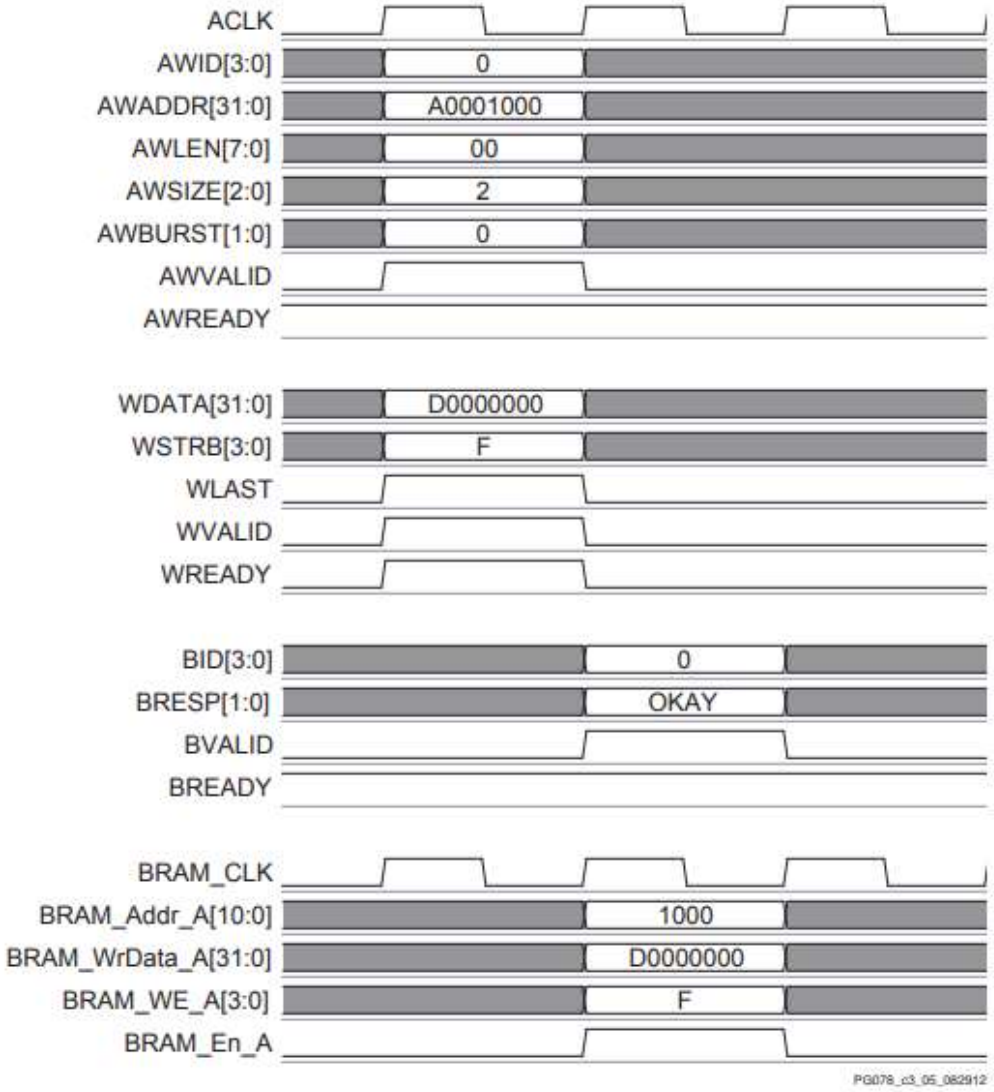
AXI4 – Lite Write with ECC



illustrates the timing of the read-modify-write (RMW) sequence when C_ECC = 1 for write transfers on the AXI4-Lite interface. To save resources the AWADDR and WDATA are not registered in the AXI4-Lite BRAM controller, so the AWREADY and WREADY output flag assertions are delayed until the RMW sequence is completed. Also depicted is the capability of the AXI4-Lite controller to handle back-to-back write request operations.

AXI Operations

Single **Write**

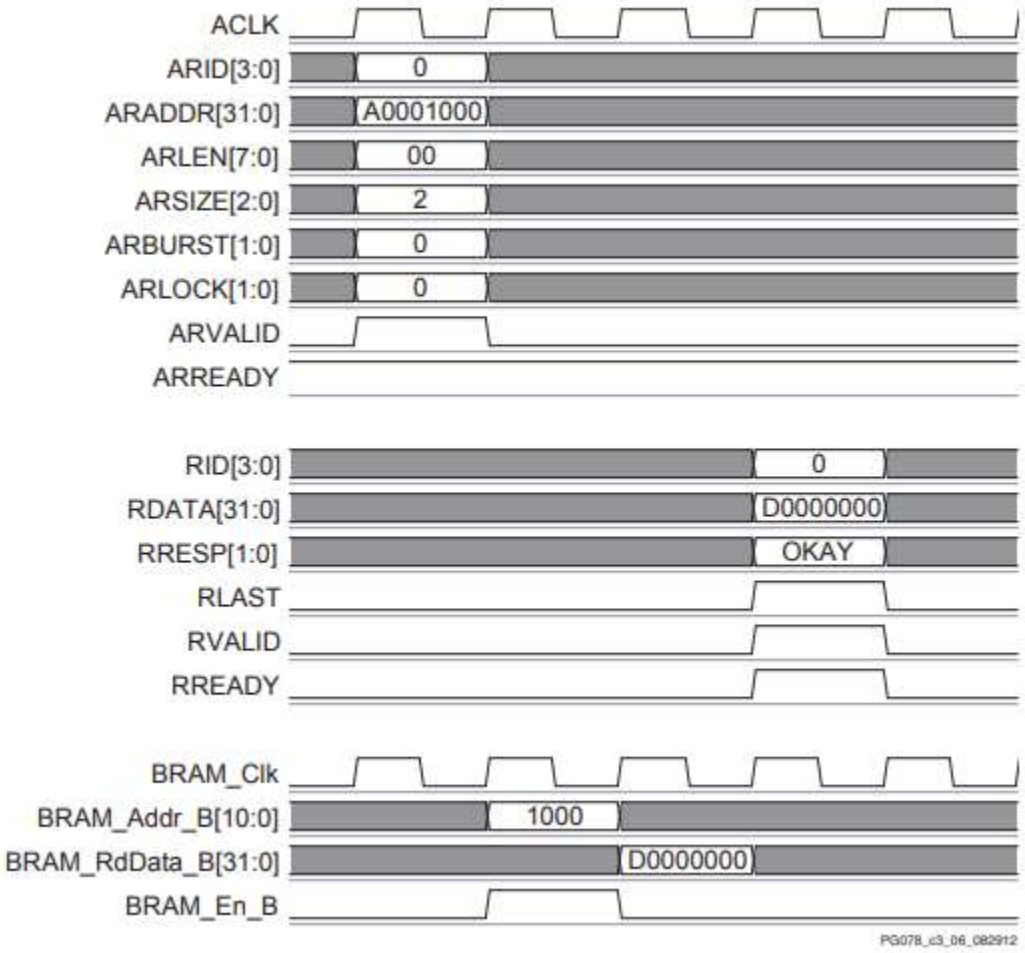


illustrates an example for the timing of an AXI single 32-bit write operation to a 32-bit wide BRAM. This example illustrates the single write to BRAM address 0x1000h, provided that C_S_AXI_BASEADDR is set to 0xA000 0000 and the C_S_AXI_HIGHADDR allows space for more than 4k of addressable BRAM. As recommended, the AXI BRAM Controller keeps the AWREADY signal asserted on the bus, as the address can be captured in the clock cycle when the S_AXI_AWVALID and S_AXI_AWREADY signals are both asserted. Once the write address pipeline (two deep) is full, then the slave AXI BRAM Controller negates the AWREADY registered output signal. The same principle applies to the write data channel, WVALID and WREADY signals. The AXI BRAM Controller negates the WREADY signal when the data pipeline writing to block RAM is full. This condition may occur when the AXI BRAM Controller is processing a prior burst write data operation. When ECC is enabled on full data width BRAM write transfers, the timing of the transaction is identical to when C_ECC = 0.

It is possible on the write data channel for the data to be presented to the AXI BRAM Controller prior to the write address channel (AWVALID). In this case, the AXI BRAM Controller does not initiate the write transactions (the write data is ignored), until the write address channel has valid information for the AXI BRAM Controller to accept.

AXI Operations

Single **Read**



illustrates an example of the timing for an AXI single read operation from a 32-bit BRAM. The registered ARREADY signal output on the AXI Read Address Channel interface defaults to a high assertion. The AXI BRAM Controller can accept the read address in the same clock
cycle as the ARVALID signal is first valid. The AXI BRAM Controller registers in the read address when the ARVALID and the ARREADY signals are both asserted. When the AXI BRAM Controller read address pipeline is full (two-deep), it clears the ARREADY signal until
the pipeline is in a non full condition. The AXI BRAM Controller can accept the same clock cycle assertion of the RREADY by the
master, if the master can accept data immediately. When the RREADY signal is asserted on the AXI bus by the master, the AXI BRAM Controller negates the RVALID signal. For single read transactions, the RLAST is asserted with the RVALID by the AXI BRAM Controller.

On AXI read transactions, the read data always follows the read address handshaking. The AXI BRAM Controller does not assert RVALID until both the ARVALID and ARREADY signals are asserted in the same clock cycle. In other words, there is no ability for early access to the AXI BRAM Controller nor internal caching ability in the AXI BRAM Controller.

AXI Burst Operations

Write Burst

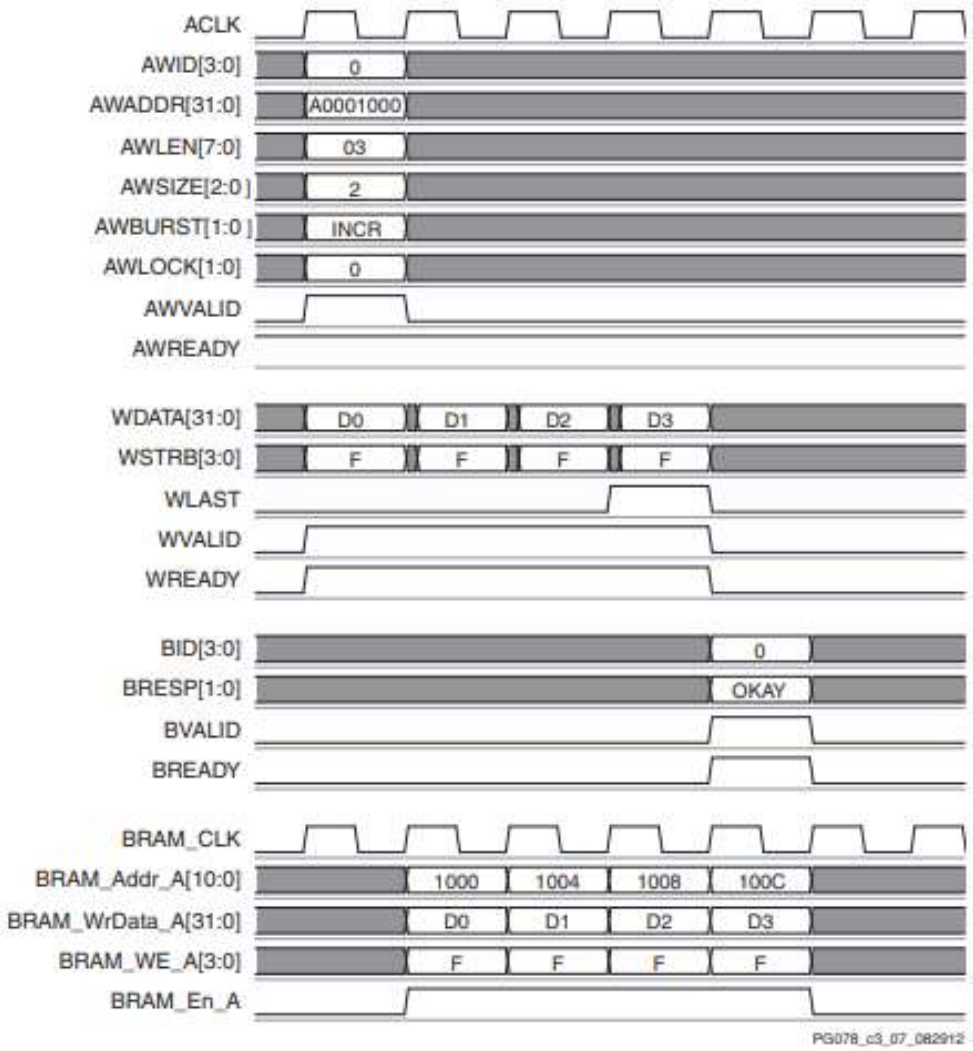


Figure illustrates an example of the timing for an AXI write burst of four words to a 32-bit BRAM. The address write channel handshaking stage communicates the burst type as INCR, the burst length of 4 data transfers (AWLEN = 0011b). The write burst utilizes all byte lanes of the AXI data bus to the block RAM (AWSIZE = 010b). The write burst shown in Figure is set to start at BRAM address 0x1000h, provided that the C_S_AXI_BASEADDR design parameter is set to 0xA000 0000 and the C_S_AXI_HIGHADDR allows space for more than 4k of addressable block RAM. On the AXI write transactions, the slave does not wait for the write data channel, WVALID signal to be asserted prior to the assertion of the write address channel signal, AWREADY. This could potentially cause a deadlock condition and is not allowed.

Narrow Write Bursting

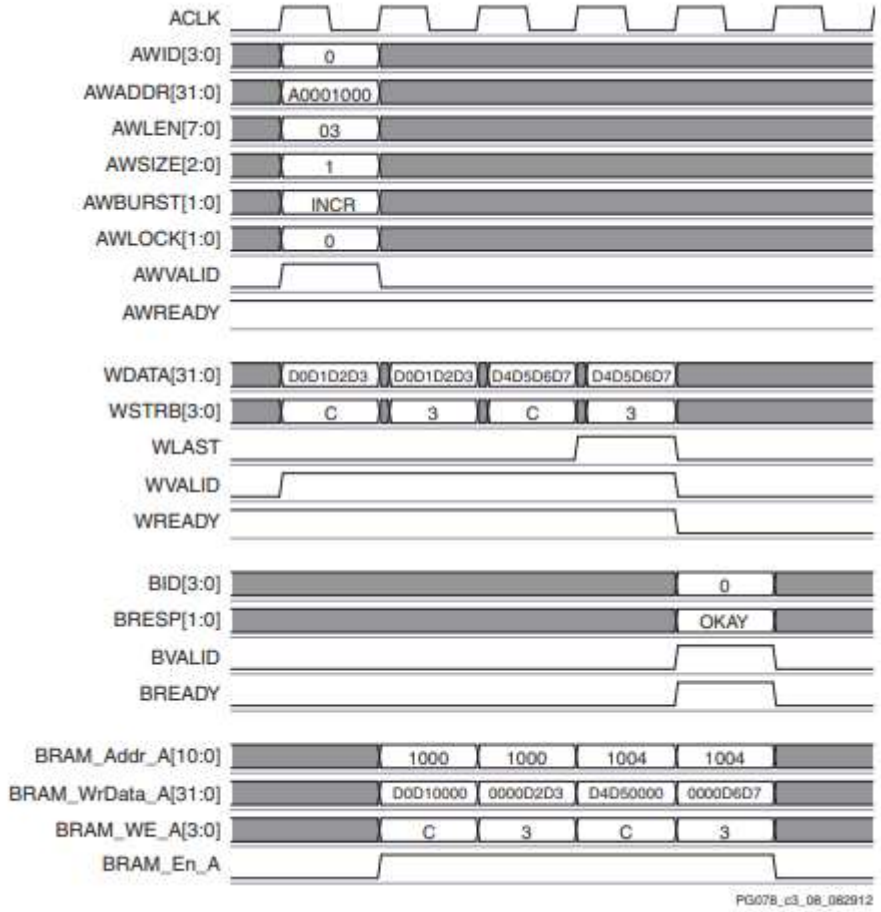
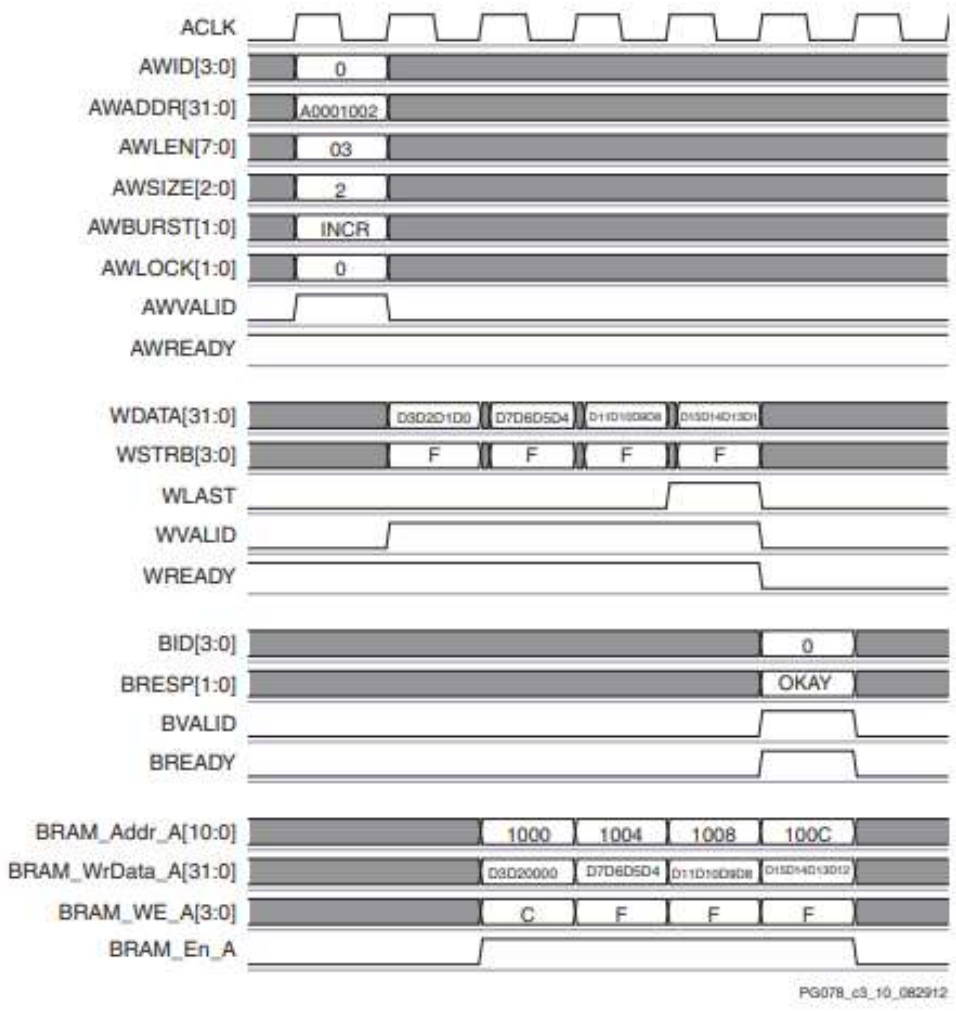


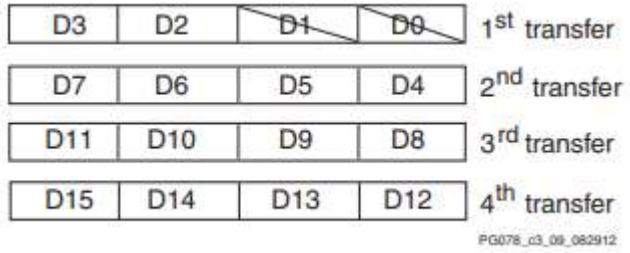
Figure illustrates an example of the AXI BRAM Controller supporting a narrow burst operation. A narrow burst is defined as a master bursting a data size smaller than the BRAM data width. If the burst type (AWBURST) is set to INCR or WRAP, then the valid data on the BRAM interface to the AXI bus rotates for each data beat. The AXI BRAM Controller handles each data beat on the AXI as a corresponding data beat to the block RAM, regardless of the smaller valid byte lanes. In this scenario, the AXI WSTRB is translated to the BRAM write enable signals. The BRAM address only increments when the full address (data) width boundary is met with the narrow write to block RAM.

Figure illustrates an example of AXI narrow bursting with a 32-bit BRAM and the AXI master request is a halfword burst of 4 data beats. AWSIZE is set to 001b.

Unaligned Write Bursting

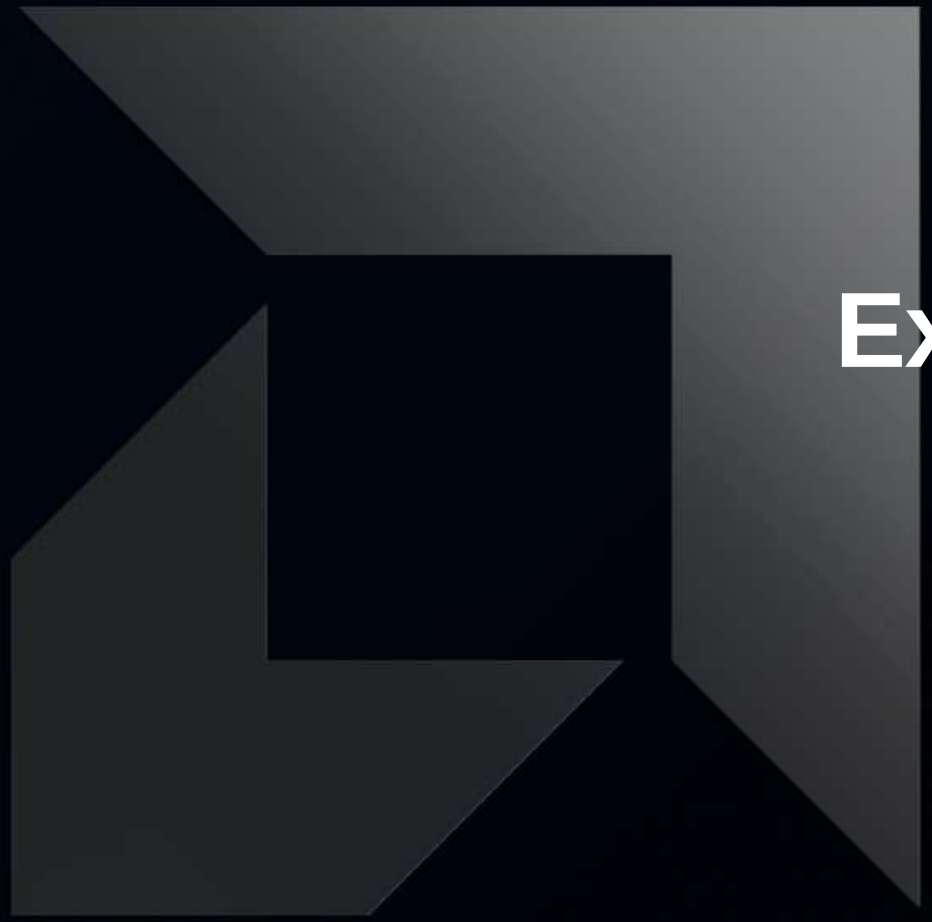


The AXI BRAM Controller supports unaligned burst transfers. Unaligned burst transfers for example, occur when a 32-bit word burst size does not start on an address boundary that matches a word memory location. The starting memory address is permitted to be something other than 0x0h, 0x4h, 0x8h, etc. The example shown in Figure 3-9, illustrates an unaligned word burst transaction of 4 data beats, that starts at address offset, 0x1002h (provided that C_S_AXI_BASEADDR is set to 0xA000 0000 and C_S_AXI_HIGHADDR allows more than 4k of addressable memory). The associated timing relationship is shown is Figure 3-10.



TIP: Note the unaligned address corresponds to the BRAM_WE signals on the write port to reflect the valid byte lanes.

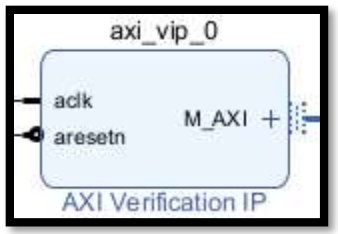
PG078

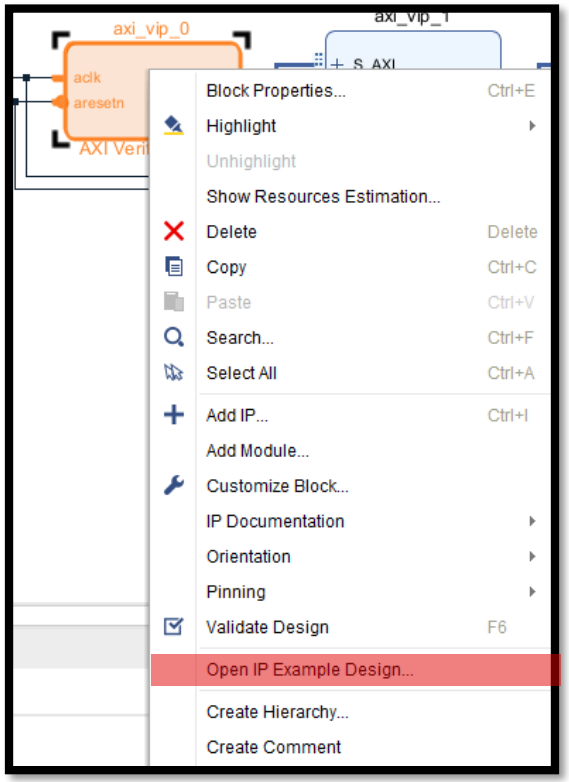


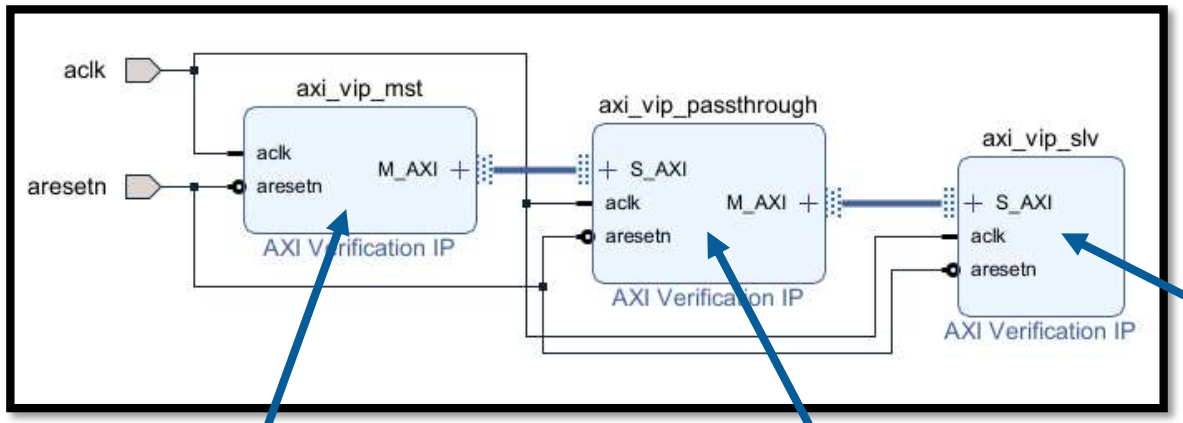
Example code

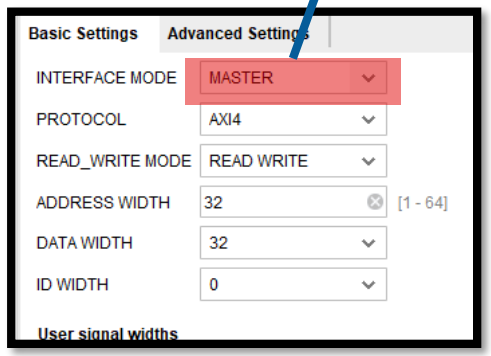
AMD
together we advance_

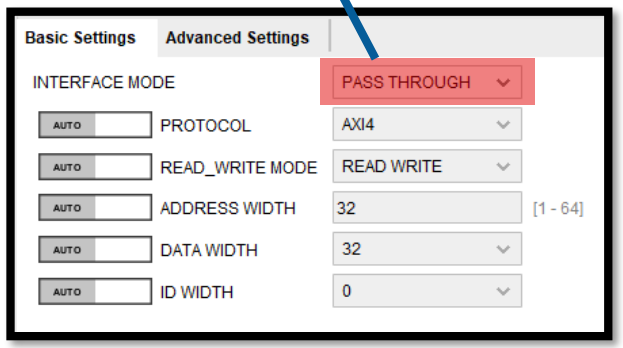
VIP AXI master to AXI passer than AXI slaver (master active)

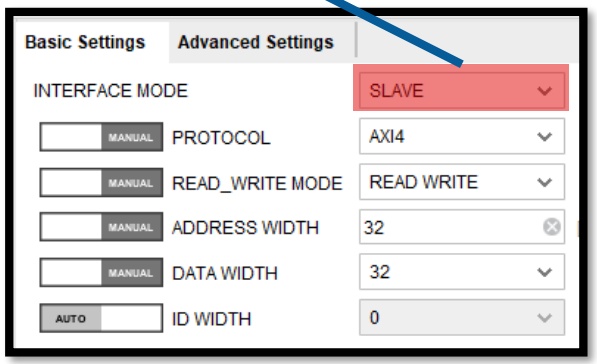






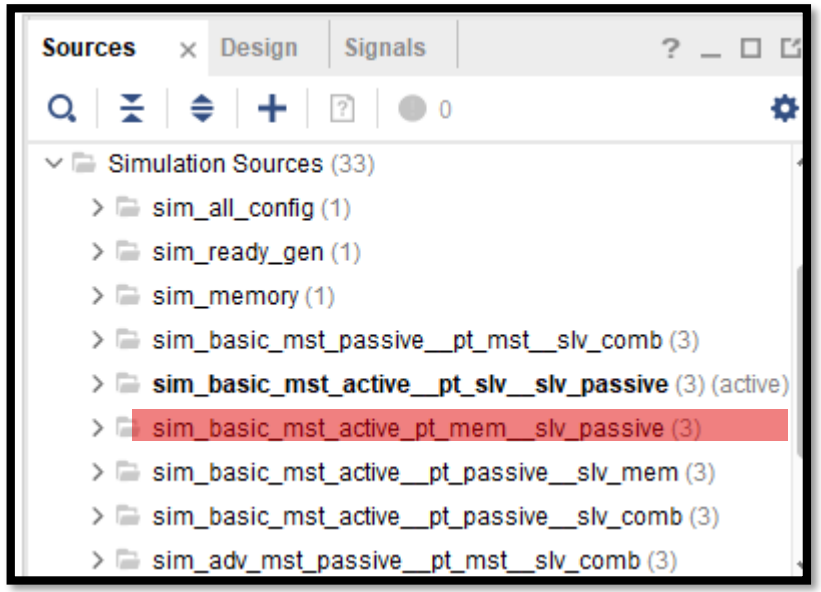


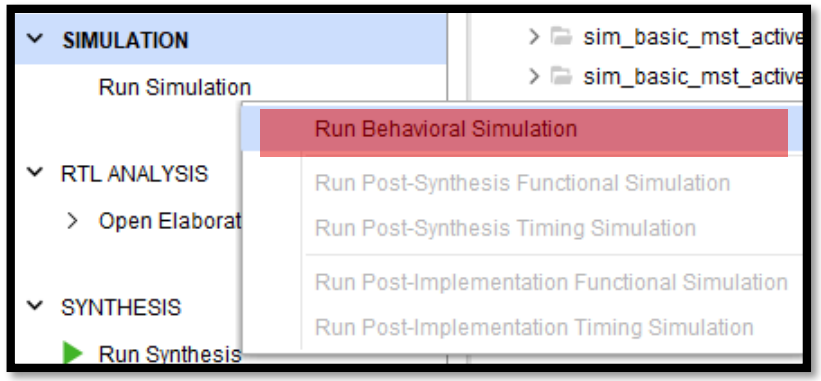




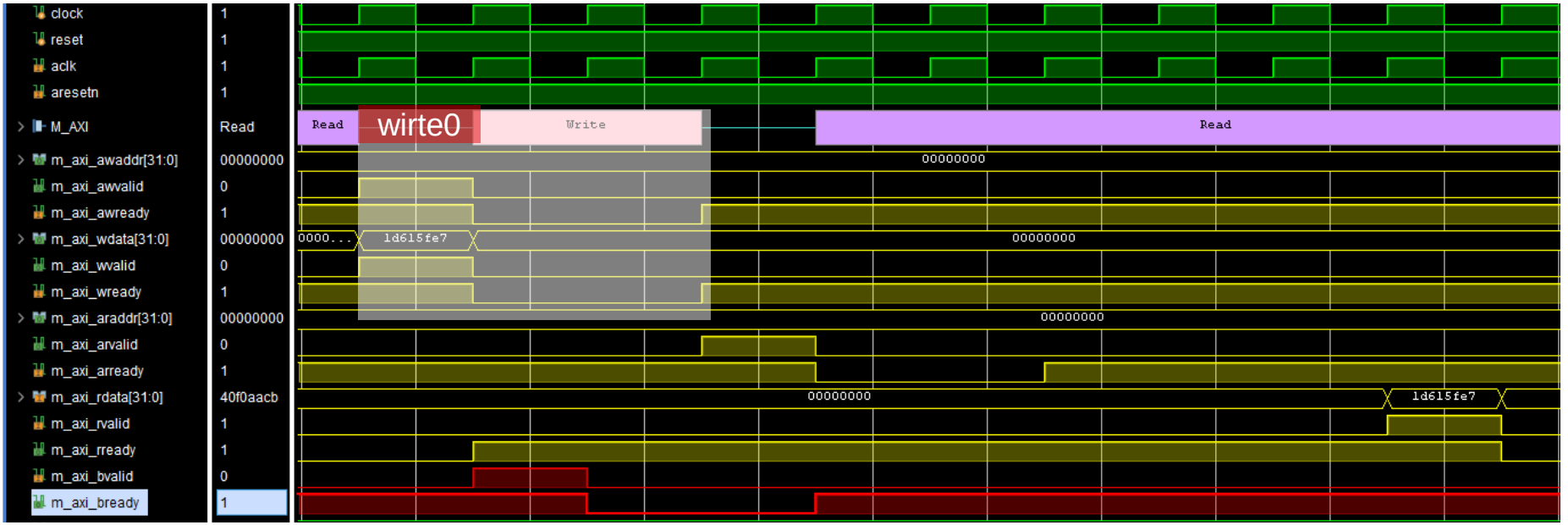
1.Create Block design, add the axi_vip, than right click >> open IP example Design.
2. Setting the interface ace mode in IP option.

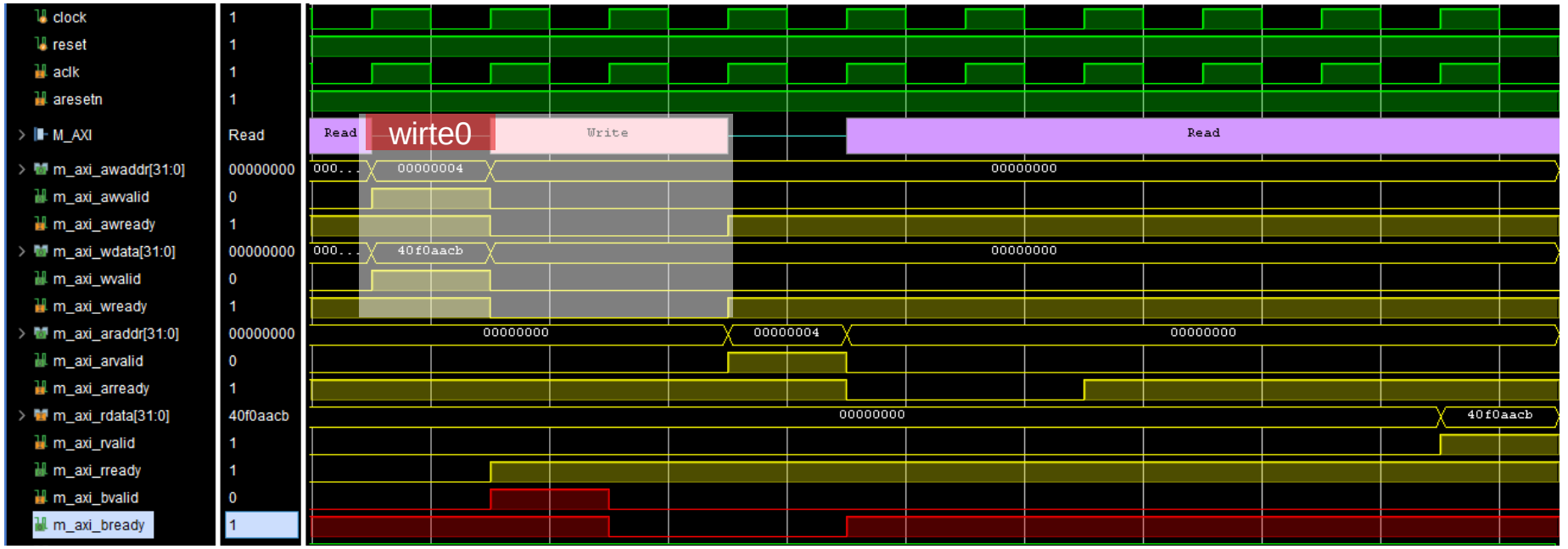
VIP AXI master to AXI passer than AXI slaver (master active)





VIP AXI master to AXI passer than AXI slaver (only do write)





AWVALID AW is address write.
WVALID W is write.
ARVALID AR is address read.
RVALID R is read.

VALID all from **Master**
Ready all from **Slaver**

Previous state

AWVAILD == 1 & AREADY == 1

Write0 State:
Addr_in = awaddr
Data_in = wdata

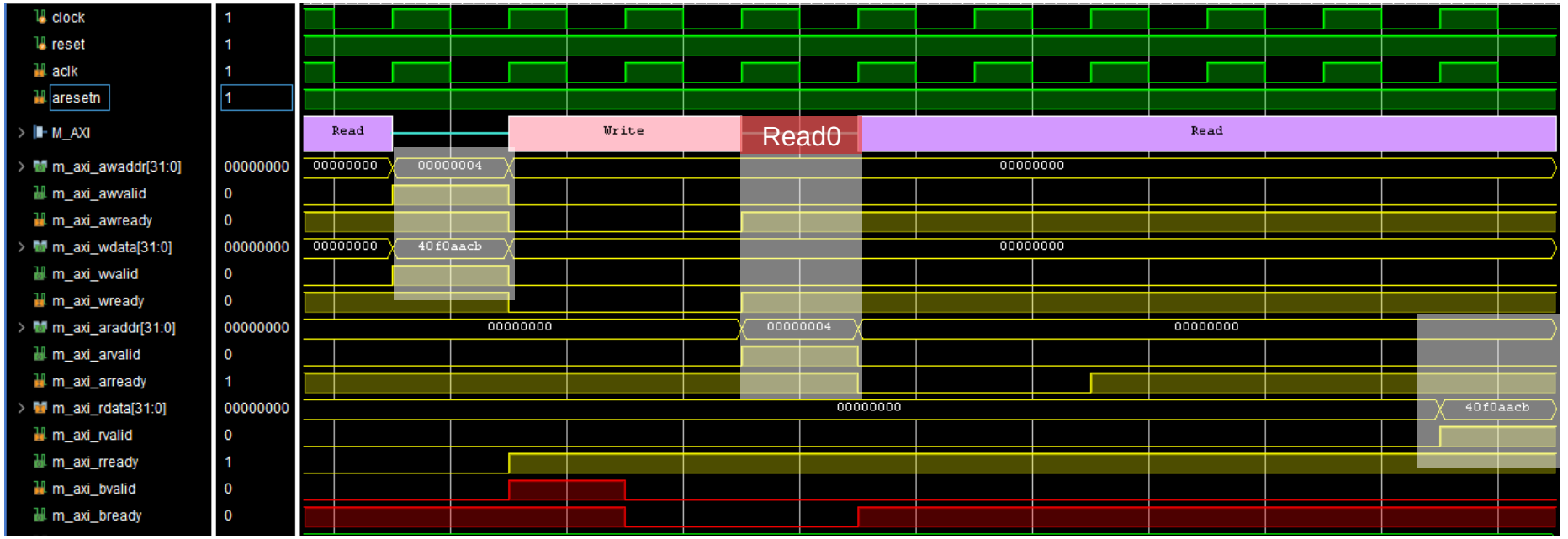
Write State
(Check bvaild and bready)?.

Next state

Look at waveform in PG.9 ~10.

AMD
together we advance_

VIP AXI master to AXI passer than AXI slaver (only do read)



AWVALID AW is address write.
WVALID W is write.
ARVALID AR is address read.
RVALID R is read.

VALID all from **Master**
Ready all from **Slaver**

Write state

ARVAILD == 1 & AREADY == 1

Read0 State:
awaddr = araddr
Awvaild = 0

ARVAILD == 0 & AREADY == 0

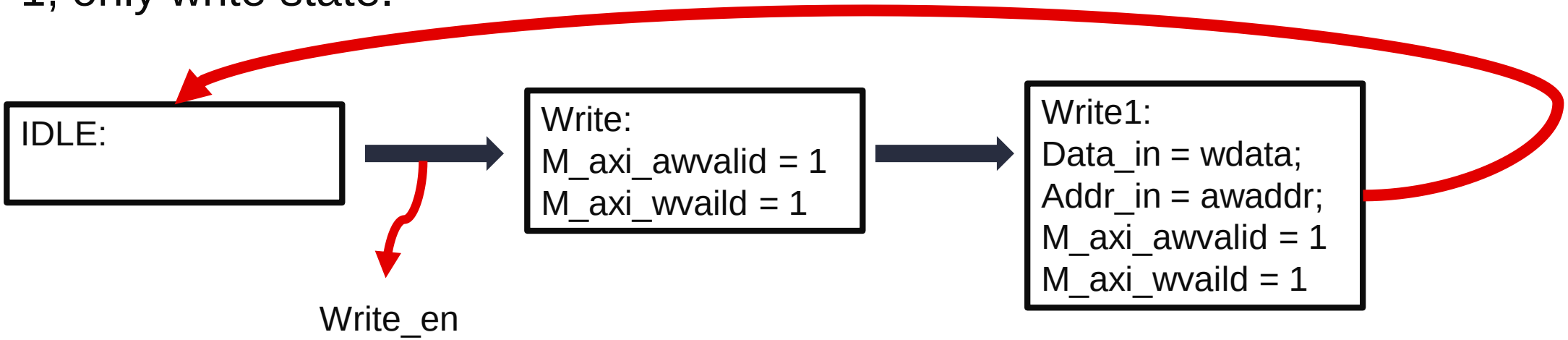
Read State:
Wdata = rdata;

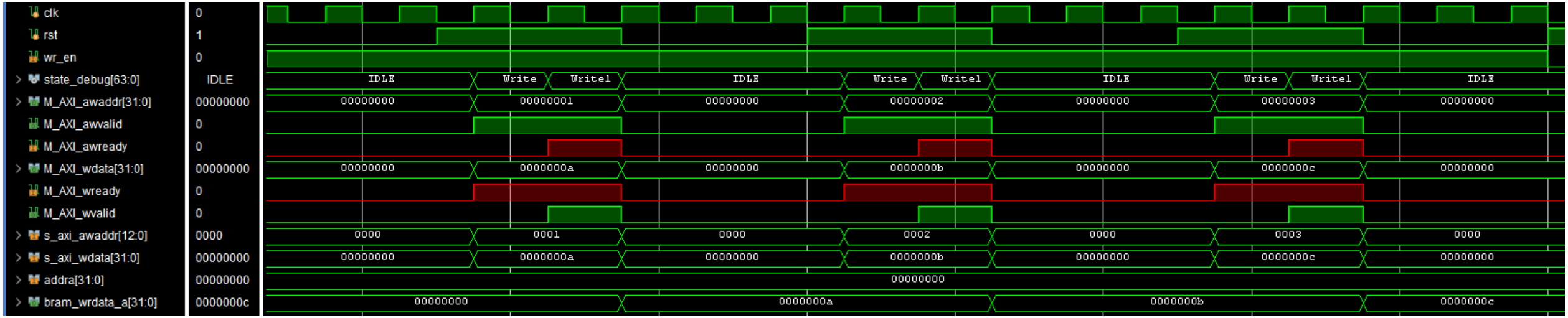
Next state

Look at waveform in PG.9 ~10.

Lab, write the AXI master reference example simulation waveform.

1, only write state.

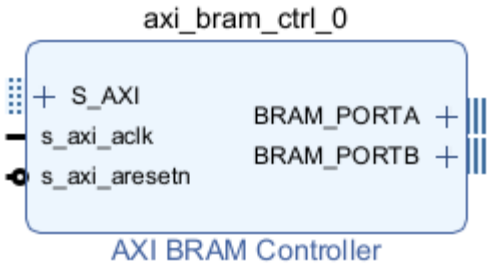




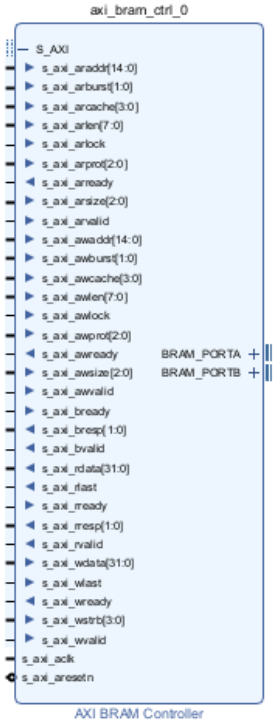
PS, Axi slaver from the Vivado's Ip catalog. This simulation are using same clk and rst, the **ready single(from S_AXI)** refer to the **RST**.

Write AXI master of RTL code, move to IP diagram.

1,using IP integrator, call out the “AXI Bram controller”.



Expend the S_AXi



Axi_bram_ctrl is AXI slaver .

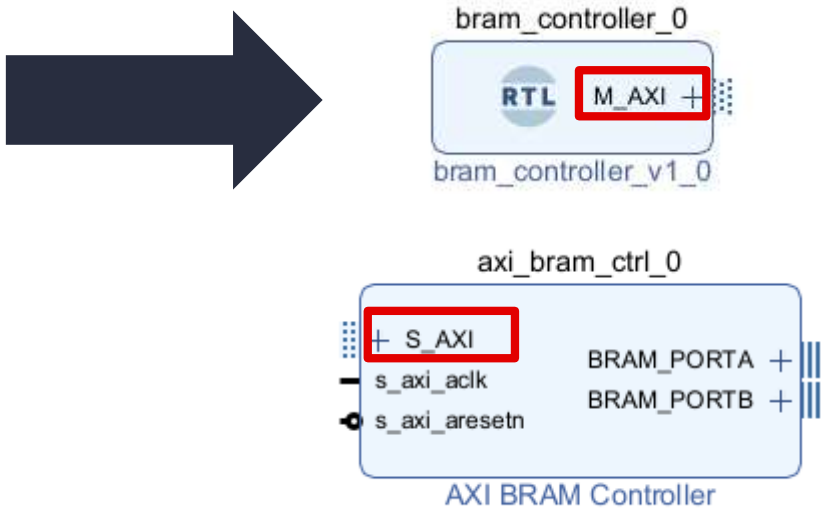
How we write the master , first we look at the expend of the S_AXI, we can see that the pins on slaver have a unified format like S_AXI_XXXX. Next , we refer to the pin defined by IP, than modify S_AXI to M_AXI. Following next page.

Write AXI master of RTL code, move to IP diagram.

1,write the RTL code, define the M_AXI pin, when you done, put it in IP diagram.

```
output reg [31:0] M_AXI_araddr,
output     [1:0]  M_AXI_arburst,
output     [3:0]  M_AXI_arcache,
output reg [7:0]  M_AXI_arlen,
output            M_AXI_arlock,
output     [2:0]  M_AXI_arprot,
input             M_AXI_arready,
output     [2:0]  M_AXI_arsize,
output reg        M_AXI_arvalid,
output reg [31:0] M_AXI_awaddr,
output     [1:0]  M_AXI_awburst,
output     [3:0]  M_AXI_awcache,
output reg [7:0]  M_AXI_awlen,
output            M_AXI_awlock,
output     [2:0]  M_AXI_awprot,
input             M_AXI_awready,
output     [2:0]  M_AXI_awsize,
output reg        M_AXI_awvalid,
output reg        M_AXI_bready,
input      [1:0]  M_AXI_bresp,
input             M_AXI_bvalid,
input      [31:0] M_AXI_rdata,
input             M_AXI_rlast,
output reg        M_AXI_rready,
input      [1:0]  M_AXI_rresp,
input             M_AXI_rvalid,
output reg [31:0] M_AXI_wdata,
output reg        M_AXI_wlast,
input             M_AXI_wready,
output reg [3:0]  M_AXI_wstrb,
output reg        M_AXI_wvalid,
```

IP diagram



- Write the RTL code, define the M_AXI pin, put source code to the IP diagram. we can see system will automatically help us convert the source code we wrote into **M_AXI**, if the PIN is written incorrectly or less any one, the system will not automatically help us to set an M_AXI. This is why we need to refer **S_AXI** to define **M_AXI.**

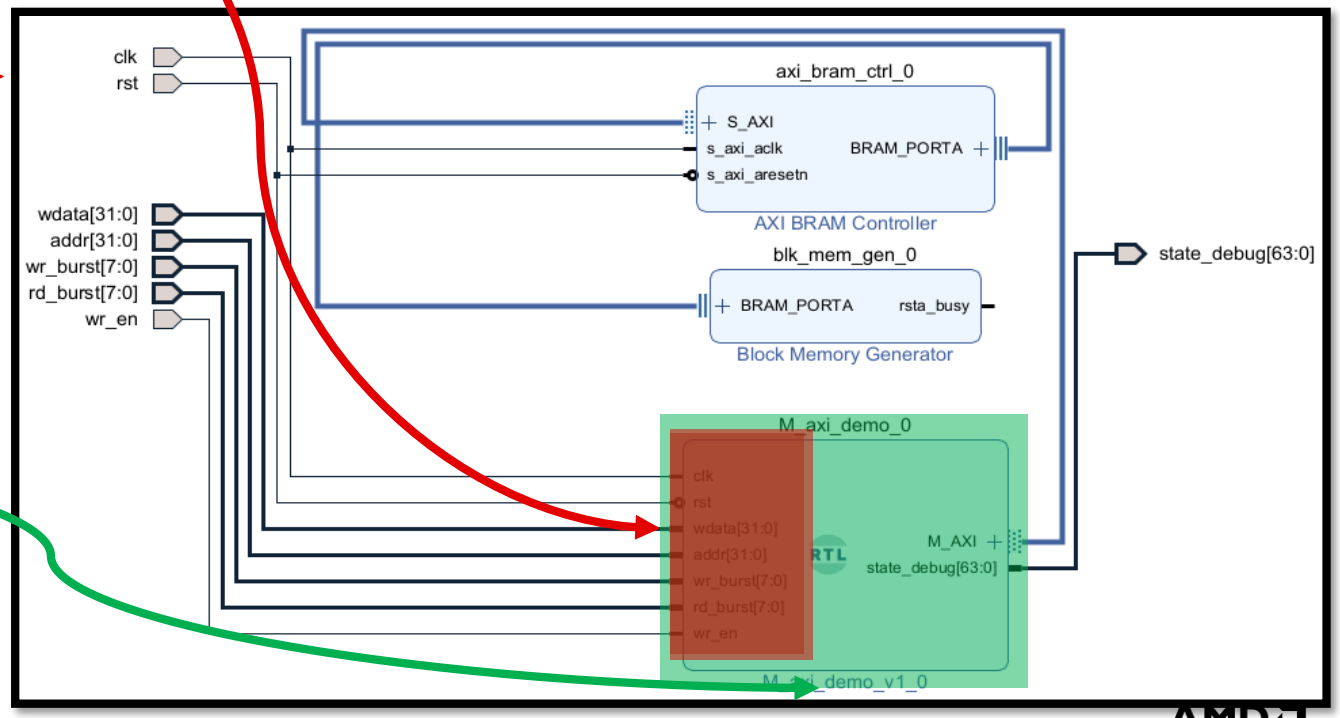
Create Block Design

```
    input               clk,        // Clock Signal
    input               rst,        // Reset Signal, active low
//    output     [7:0]  state,      // Debug Signal
    input      [31:0] wdata,       // Write Data Port
    input      [31:0] addr,        // DRAM Read/Write Address
    input      [7:0]  wr_burst,    // Write Burst Length, Max 256
    input      [7:0]  rd_burst,    // Read Burst Length, Max 256
//    input             rd,         // Read Enable Signal
    input              wr_en,       // Write Enable Signal
    output  [63:0] state_debug // debug the state_debug
    );
    //////write state
```

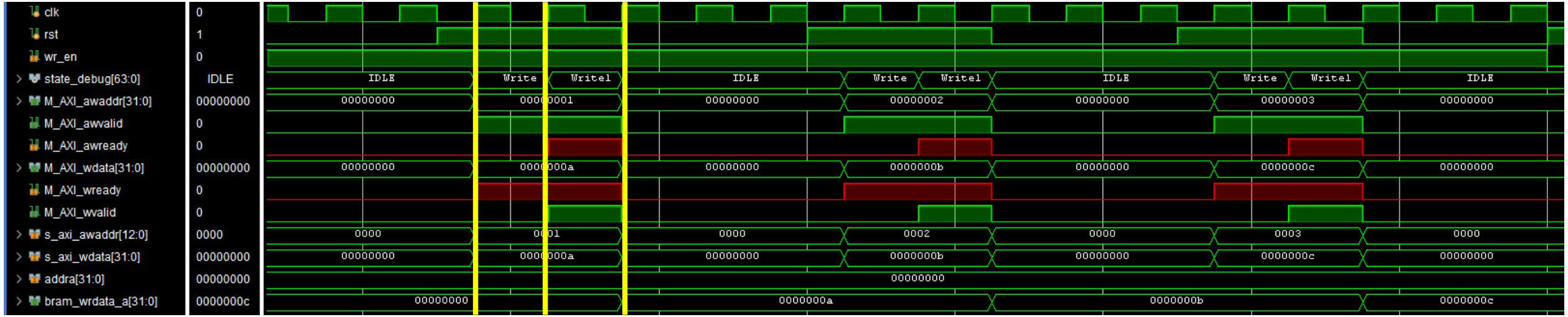
1. Define the input pin in the RTL code.
2. simulate Source.
3. Only do the write state, so we just define the wr_en.

1. Block Design will like the JPG on right side

```
output reg [31:0] M_AXI_araddr,
output     [1:0]  M_AXI_arburst,
output     [3:0]  M_AXI_arcache,
output reg [7:0]  M_AXI_arlen,
output            M_AXI_arlock,
output     [2:0]  M_AXI_arprot,
input             M_AXI_arready,
output     [2:0]  M_AXI_arsize,
output reg        M_AXI_arvalid,
output reg [31:0] M_AXI_awaddr,
output     [1:0]  M_AXI_awburst,
output     [3:0]  M_AXI_awcache,
output reg [7:0]  M_AXI_awlen,
output            M_AXI_awlock,
output     [2:0]  M_AXI_awprot,
input             M_AXI_awready,
output     [2:0]  M_AXI_awsize,
output reg        M_AXI_awvalid,
output reg        M_AXI_bready,
input      [1:0]  M_AXI_bresp,
input             M_AXI_bvalid,
input      [31:0] M_AXI_rdata,
input             M_AXI_rlast,
output reg        M_AXI_rready,
input      [1:0]  M_AXI_rresp,
input             M_AXI_rvalid,
output reg [31:0] M_AXI_wdata,
output reg        M_AXI_wlast,
input             M_AXI_wready,
output reg [3:0]  M_AXI_wstrb,
output reg        M_AXI_wvalid,
```



State in waveform

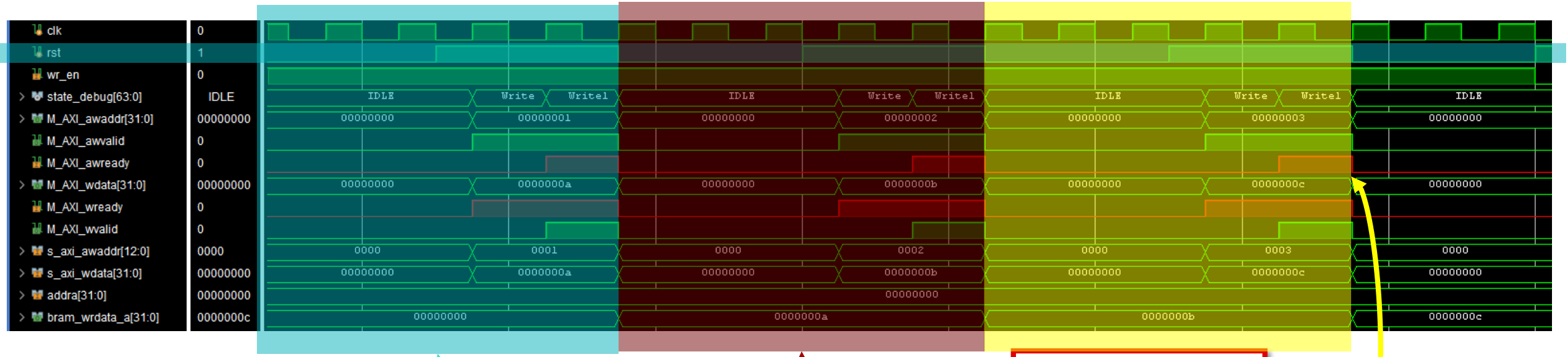



```
IDLE:
    begin
    M_AXI_araddr  = 32'h0;
    M_AXI_arlen   = 8'd0;
    M_AXI_arvalid = 1'b0;
    M_AXI_awaddr  = 32'h0;
    M_AXI_awlen   = 8'd0;
    M_AXI_awvalid = 1'b0;
    M_AXI_bready  = 1'b0;
    M_AXI_rready  = 1'b0;
    M_AXI_wdata   = 32'h0;
    M_AXI_wlast   = 1'b0;
    M_AXI_wstrb   = 4'h0;
    M_AXI_wvalid  = 1'b0;
    end
```

```
Write:
    begin
    M_AXI_araddr  = 'b0;
    M_AXI_arlen   = 8'd0;
    M_AXI_arvalid = 1'b0;
    M_AXI_awaddr  = addr;
    M_AXI_awlen   = 8'd0;
    M_AXI_awvalid = 1'b1;
    M_AXI_bready  = 1'b0;
    M_AXI_rready  = 1'b0;
    M_AXI_wdata   = wdata;
    M_AXI_wlast   = 1'b0;
    M_AXI_wstrb   = 4'h0;
    M_AXI_wvalid  = 1'b0;
    end
```

```
Writel:
    begin
    M_AXI_araddr  = 'b0;
    M_AXI_arlen   = 8'd0;
    M_AXI_arvalid = 1'b0;
    M_AXI_awaddr  = addr;
    M_AXI_awlen   = 8'd0;
    M_AXI_awvalid = 1'b1;
    M_AXI_bready  = 1'b0;
    M_AXI_rready  = 1'b0;
    M_AXI_wdata   = wdata;
    M_AXI_wlast   = 1'b0;
    M_AXI_wstrb   = 4'h0;
    M_AXI_wvalid  = 1'b1;
    end
```

Test beach




```
module demo_2_wrapper();
  reg [31:0]addr;
  reg clk;
  reg [7:0]rd_burst;
  reg rst;
  wire [63:0]state_debug;
  reg [31:0]wdata;
  reg [7:0]wr_burst;
  reg wr_en;

  demo_2 demo_2_i
       (.addr(addr),
        .clk(clk),
        .rd_burst(rd_burst),
        .rst(rst),
        .state_debug(state_debug),
        .wdata(wdata),
        .wr_burst(wr_burst),
        .wr_en(wr_en));
```

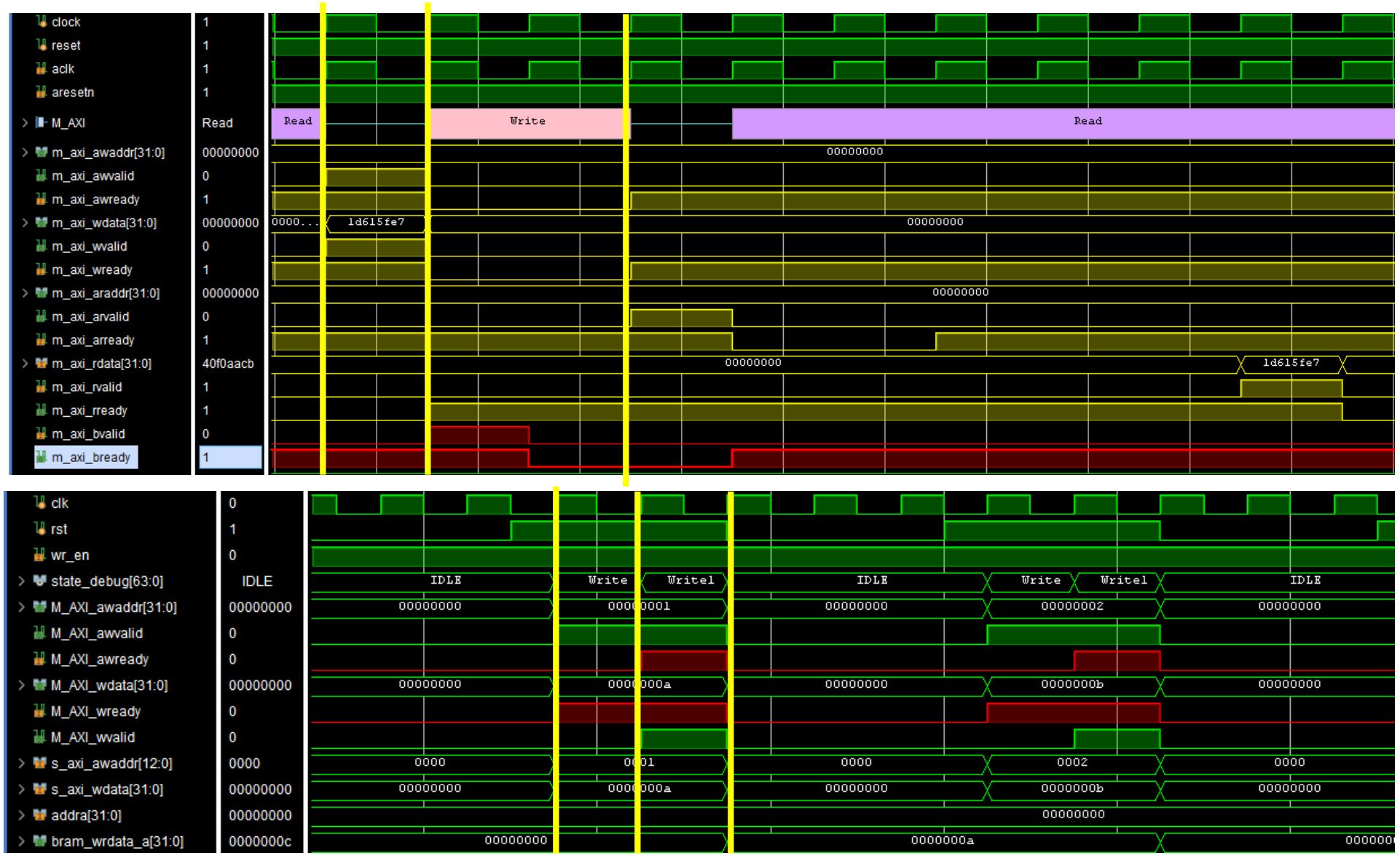
```
always #5 clk = ~clk;
//always #50 wr_en = ~wr_en;
always #25 rst = ~rst;
initial
begin
  clk = 'b0;
  addr ='b1;
  rd_burst ='b0;
  rst ='b1;
  wdata ='ha;
  wr_burst ='b0;
  wr_en ='b1;

  #100
  addr ='d2;
  wdata ='hb;

  #50
  addr ='d3;
  wdata ='hc;

  #50
  wr_en = 'b0;
```

Waveform compare example code



Need to confirm:

1,AWready single is different to example.

Address read/write single definition

Write address channel and definition			Read address channel and definition		
PIN	**SOURCE**		**PIN**	**SOURCE**	
Awid[3:0]	Master	Write ID of address	Arid[3:0]	Master	
Awaddr[31:0]	Maeter	Write the address	Araddr[31:0]	Maeter	
Awled[7:0]	Master	Number of data	Arled[7:0]	Master	
Awsize[2:0]	Master	Width of data	Arsize[2:0]	Master	
Awburst[1:0]	Master	Type of burst,	Arburst[1:0]	Master	
Awlock	Master	Type of lock.	Arlock	Master	
Awcache[3:0]	Master	Type of cache	Arcache[3:0]	Master	
Awprot[2:0]	Master	Type of protection	Arprot[2:0]	Master	
Awvalid	Master	Address valid	Arvalid	Master	
Awready	**Slaver**	Ready to write address	**Arready**	**Slaver**	
Awqos[3:0]	Master		Arqos[3:0]	Master	
Awregion[3:0]	Master		Arregion[3:0]	Master	

Awburst [1:0]/ Arburst[1:0]			
[1:0]	**Tpye of burst**	**Describe**	**target**
00	固定長度突發	地址固定的突發	FIFO
01	增量突發(INCR)	地址遞增的突發	正常序列 memory
10	回還突發(WRAP)	地址遞增的突發，在邊界會回到較低的地址	高速緩存(cache)
11	-	-	-

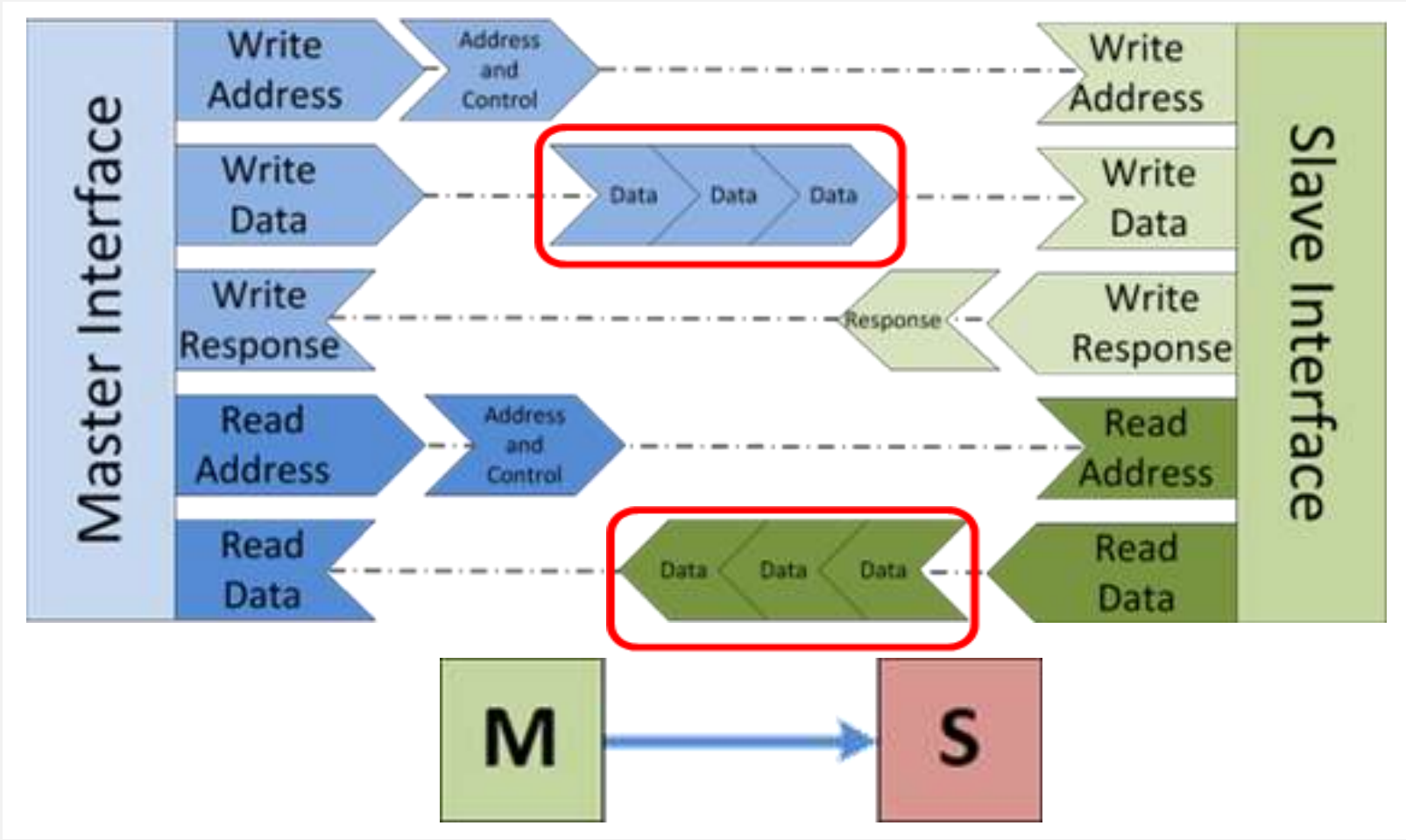
Data read/write single definition

Write data channel and definition		
PIN	**SOURCE**	
Rid[3:0]	slaver	
Rdata[31:0]	slaver	
Rres[1:0]	slaver	
Rlast[2:0]	slaver	
Rvalid[1:0]	slaver	
Rready	Master	

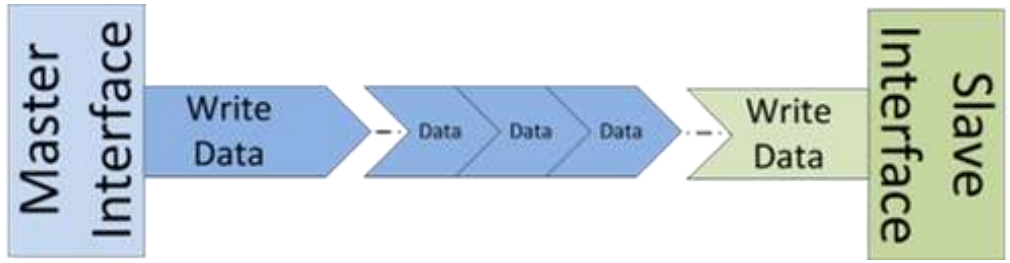
Read data channel and definition		
PIN	**SOURCE**	
Wdata[31:0]	Master	
Wstrb[1:0]	Master	
Wlast[2:0]	Master	
Wvalid[1:0]	Master	
Wready	slaver	

ACK data channel and definition		
PIN	**SOURCE**	
Bid[3:0]	slaver	
Bresp[1:0]	slaver	
Bvalid[1:0]	slaver	
Bready	Master	

AXI-4 Memory Map

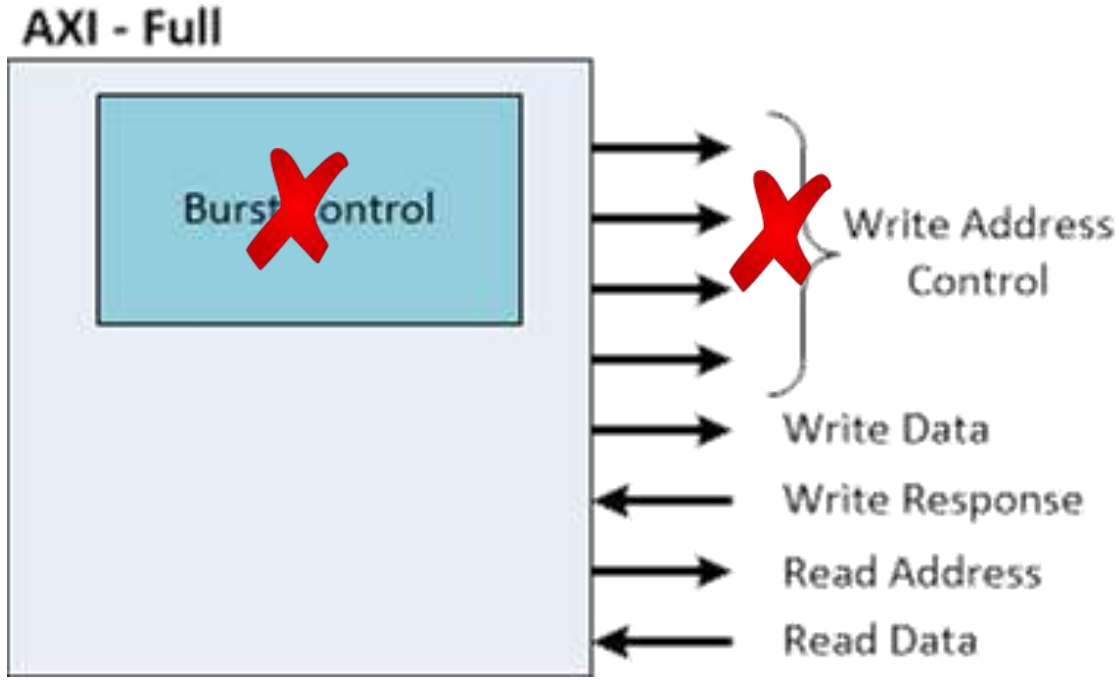


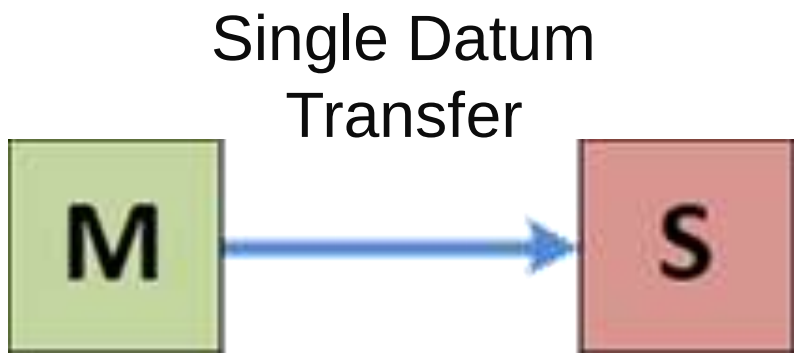
AXI-4 Stream



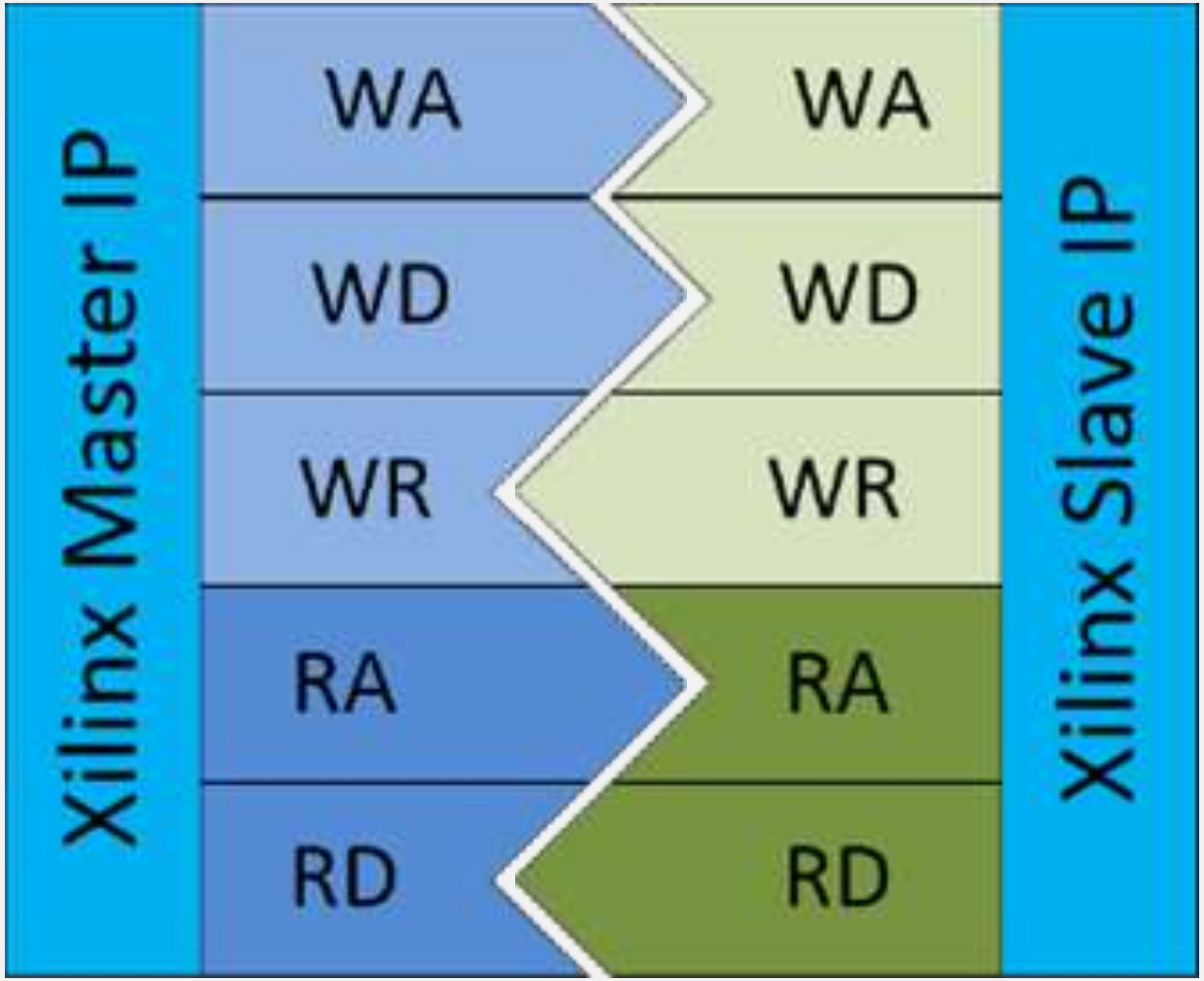
Data Only Moves from Master to Slave

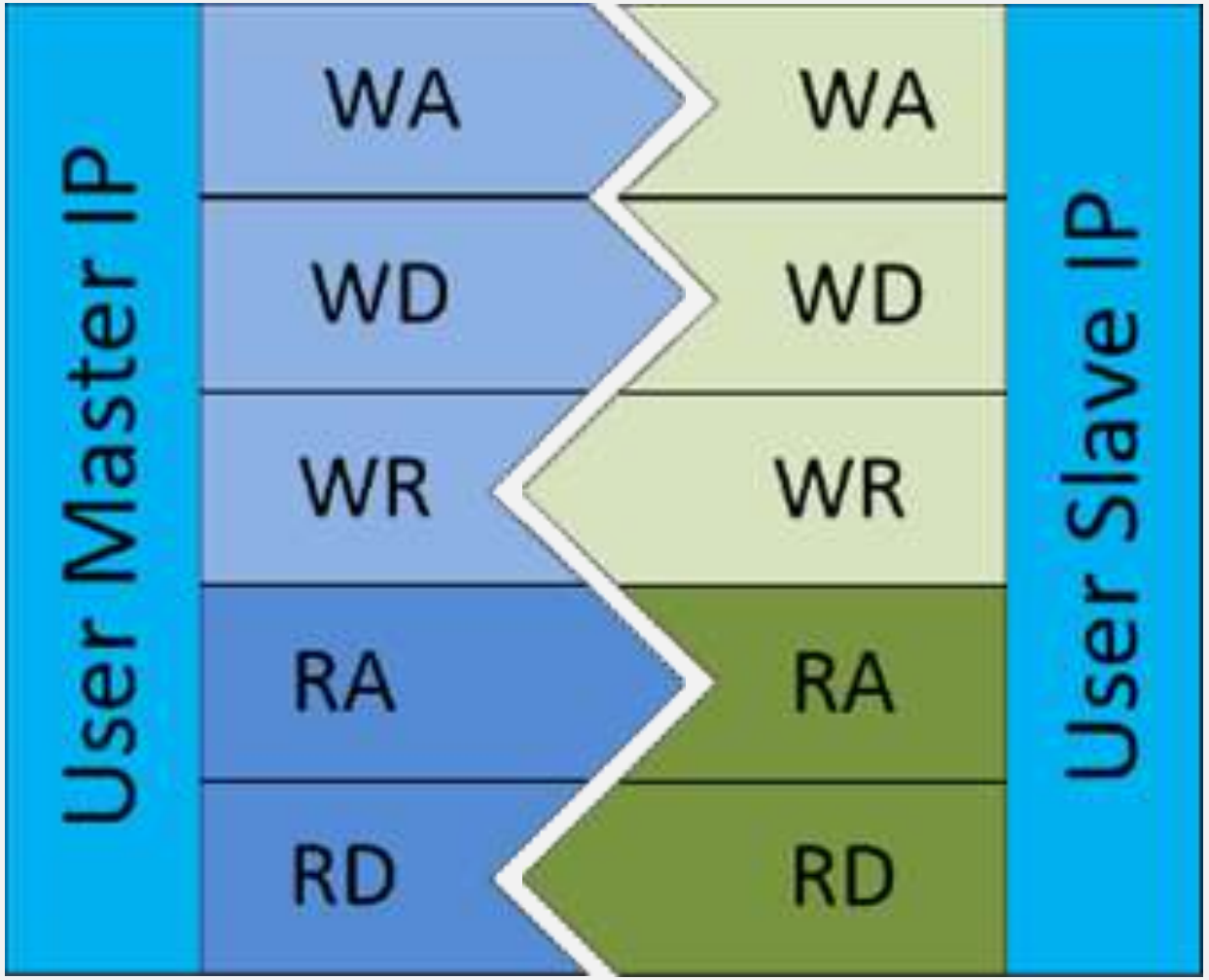
AXI-4 Lite





Goals of AXI






AMD