AMBA AXI

Narendiran

June 4, 2021

AXI - Advacned eXtensible Interface

0.0.1 Objectives

- high-performance, high frequency system for high-speed submicron interconnect
- high-bandwidth, low-latency
- flexibility in implementation of interconnect architecture
- backward-compatible with AHB and APB Interface

0.0.2 Features

- seperate address/control and data phases
- unaligned data transfers using byte strobes
- burst-based transactions
- seperate read and write data channels to enabled low-cost DMA
- issue multiple outstanding addressess
- out-of-order transaction completion
- easy addition of register stages to provide timing closure

1 Architecture

- AXI protocol is *burst-based*.
- Every transaction has address and control information on the address channel which describes the nature of data transfers
- Data is transferred between master and slave
  - Using a write data channel to the slave
  - Using a read data channel to the master
- In write transaction, all data flows from master to slave
- In write response channel, the slave signals the completion of write transaction to master.

The read transaction can be seen below which uses *Read address* and *Read data* channel

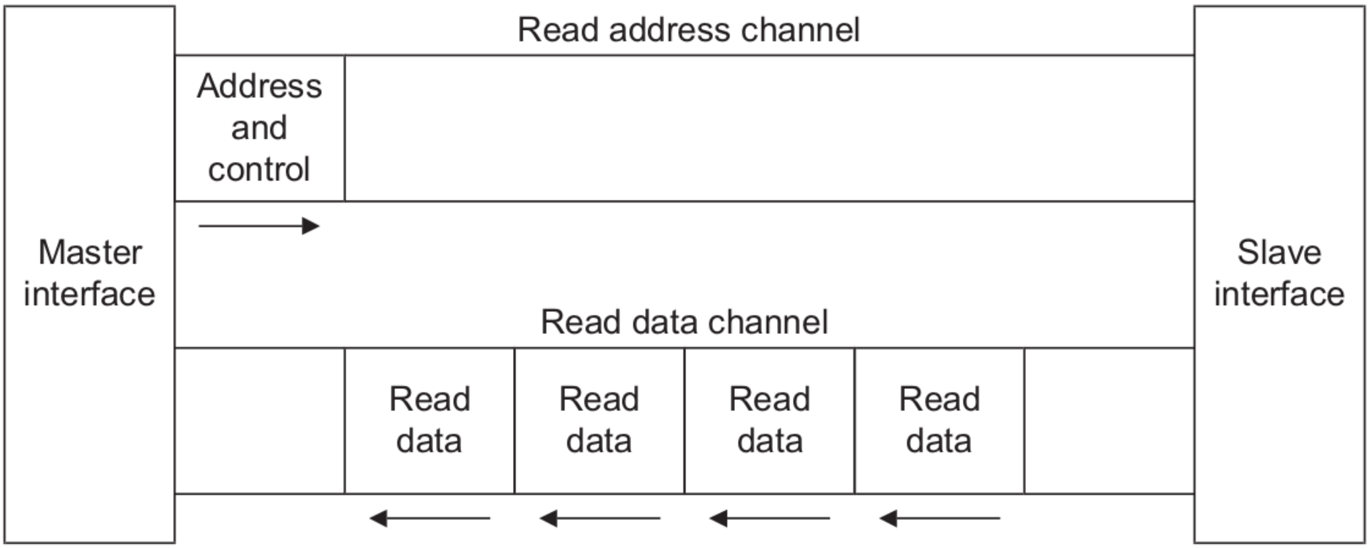


Figure 1: Read transaction

The write transaction can be seen below which uses *Write address*, *Write Data* and *Write Response* channel

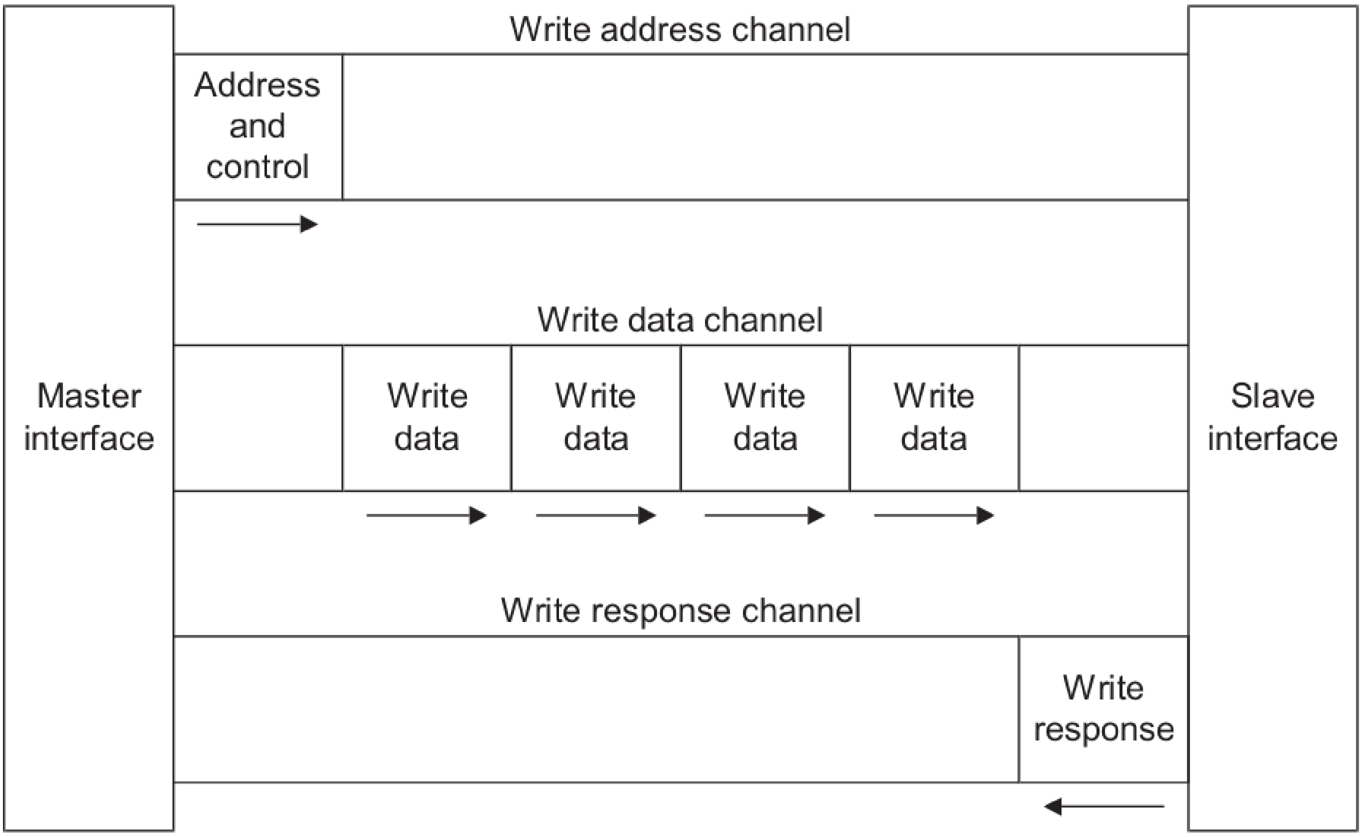


Figure 2: Write transaction

1.1 Channel Definition

- There are five channels namely *Read Address* channel, *Read Data* channel, *Write Address* channel, *Write Data* channel and *Write Response* channel.
- All the five independent channels uses a two-way **VALID** and **READY** handshake mechanism.
  - The information Source uses the **VALID** signal to shown when valid data or control information is available on the channel.

- The Destination uses the **READY** signal to show it can accept the data.

- The *Read data* and *Write data* channel uses the **LAST** signal to indicate the transfer of final data item withing a transaction.

1.1.1 Supported Mechanisms

- Variable-length bursts from 1 to 16 data transfers per burst.
- Each data transfer can have a transfer size 8 to 1024 bits.
- Gives ID tag to every transaction across interface.
- wrapping, incrementing and non-incrementing burts.
- atopic operations using exclusive or locked accesses
- system-level caching and buffering control
- secure and priviledged access.

1.1.2 Read Address Channel

- Read Tranaction has it's own address channel.
- Carries the required address and control information for the read transaction.

1.1.3 Read Data Channel

- *Read Data* channel conveys both read data and read response information from slave back to master.
- includes a data data bus which can be 8, 16, 32, 64, 128, 256, 512, 1024 bit width
- read response indicating the completion of read transaction.

1.1.4 Write Address Channel

- Write Tranaction has it's own address channel.
- Carries the required address and control information for the write transaction.

1.1.5 Write Data Channel

- *Write data* channels conveys the write data from master to slave.
- include a data data bus which can be 8, 16, 32, 64, 128, 256, 512, 1024 bit width
- include one byte(8-bit) lane strobe for every eight data bits indicate which bytes of data bus are valid.
- *Write data* channel information is buffered, so master can perfrom wirte transaction without slave acknowledgemnt of previous write transaction.

1.1.6 Write Response Channel

- *Write Response* channel proives a way for the slave to respond to write transactions
- All write transaction uses completion signalling.
- The completion signalling occurs once for each burst and not for each individual data transfer within a burst.

1.2 Interface and Interconnect

Consist of master and slave devices connected together fthrough some interconnect.

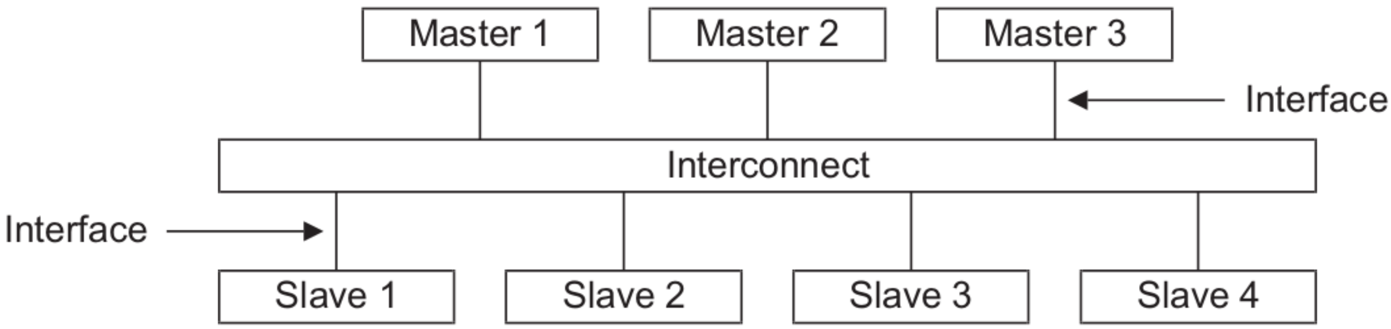


Figure 3: Interconnects

1.3 Register Slices

- Each AXI channel transfer information in only one direction.
- Enables insertion of register slices in any channel at the cost of additional cycle of latency.
- Register slices can be used at any point within a interconnect.

1.4 Read Burst example

THe following describes the read burst of four transfer.

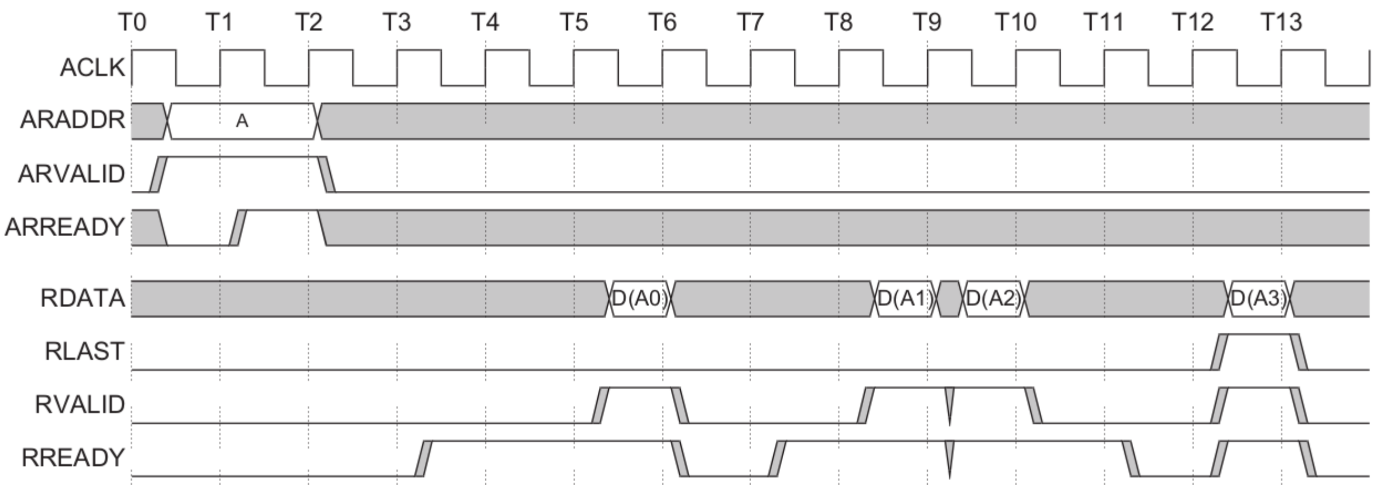


Figure 4: Read Burst Example

- The *read address* and *read data* channels are used for this read transaction.
- Master drives the address and slave accept in one cycle.
  - Master places the address on **ARADDR** bus in the *read address* channel.
  - Master makes the **ARVALID** HIGH to indicate that read address is available and its valid.
  - Then, the slave makes the **ARREADY** signal HIGH to indicate its ready to accept the address
- The master on receiving the **ARREADY** signal of the *read address* channel from the slave, makes the **RREADY** signal of the *Read data* channel HIGH.
- Now, the slave sends the data throught the *read data* channels.

- Slave places the data on to the **RDATA** bus and makes the **RVALID** signal HIGH to start the data transfer.
- Now, the master on receives the first data by checking the **RVALID** signal form the slave.
- On receiving the data, the master makes the **RREADY** signal LOW untill it can process the recieved data from slave.
- WHen master is ready to receive, the master makes the **RREADY** singal HIGH to indicate it can receive further data.

- Now, the slave places the second data onto the **RDATA** bus and makes the **RVALID** signal HIGH.
- The above process repeates.
- When the last data is to be sent, the slave placess the last data onto the **RDATA** bus and makes the **RLAST** and **RVALID** HIGH.

1.5 Overlapping read example

- Master drives another burst address after slave accpets the first address.
- this enables the slave to begin processing data for second burst in parallel with completion of first bus.
- The following describes the Overlapping read burst example.

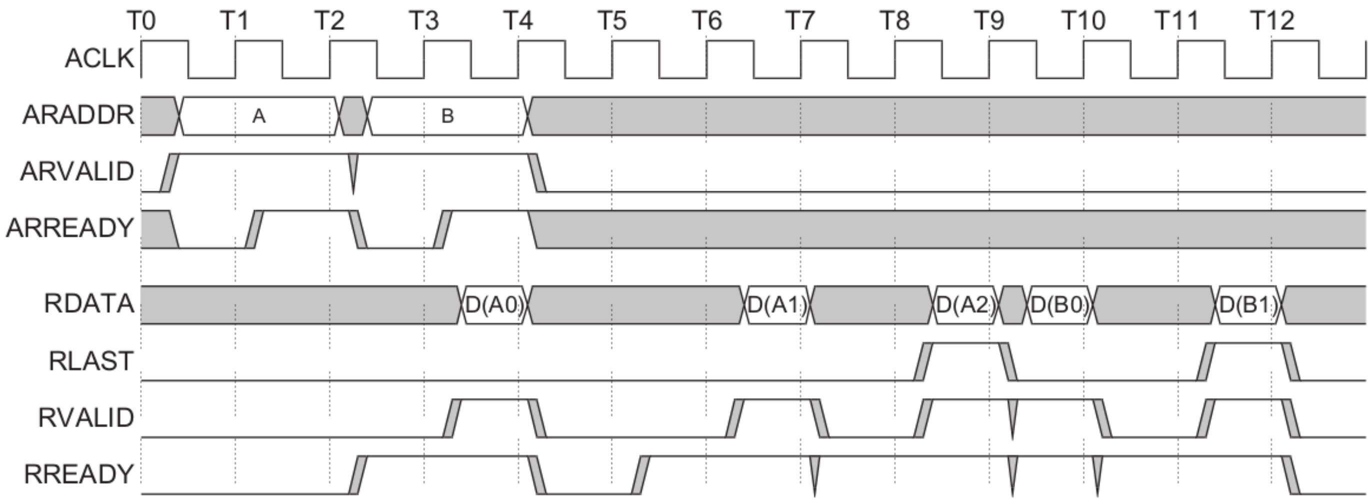


Figure 5: Overlapping Burst Example

- The *read address* and *read data* channels are used for this read transaction.
- Master drives the address and slave accept in one cycle.
  - Master places the address on **ARADDR** bus in the *read address* channel.
  - Master makes the **ARVALID** HIGH to indicate that read address is available and its valid.
  - Then, the slave makes the **ARREADY** signal HIGH to indicate its ready to accept the address
- The master on receiving the **ARREADY** signal of the *read address* channel from the slave, makes the **RREADY** signal of the *Read data* channel HIGH.
- Now, the slave sends the data throught the *read data* channels.
- During the above process, the master places the next address to be read for next read burst onto the **ARADDR** bus and makes the **ARVALID** singal.

- Now, the slave along with responding to the previous read transaction, it makes the **ARREAD** HIGHT to indicate it can receive data.
- As soon as the slave completes the first data transaction, the slave begins the next transaction by placing the data onto **RDATA** bus and making the **RVALID** singal HIGH.

1.6 Write Burst example

The following describes the write burst example.

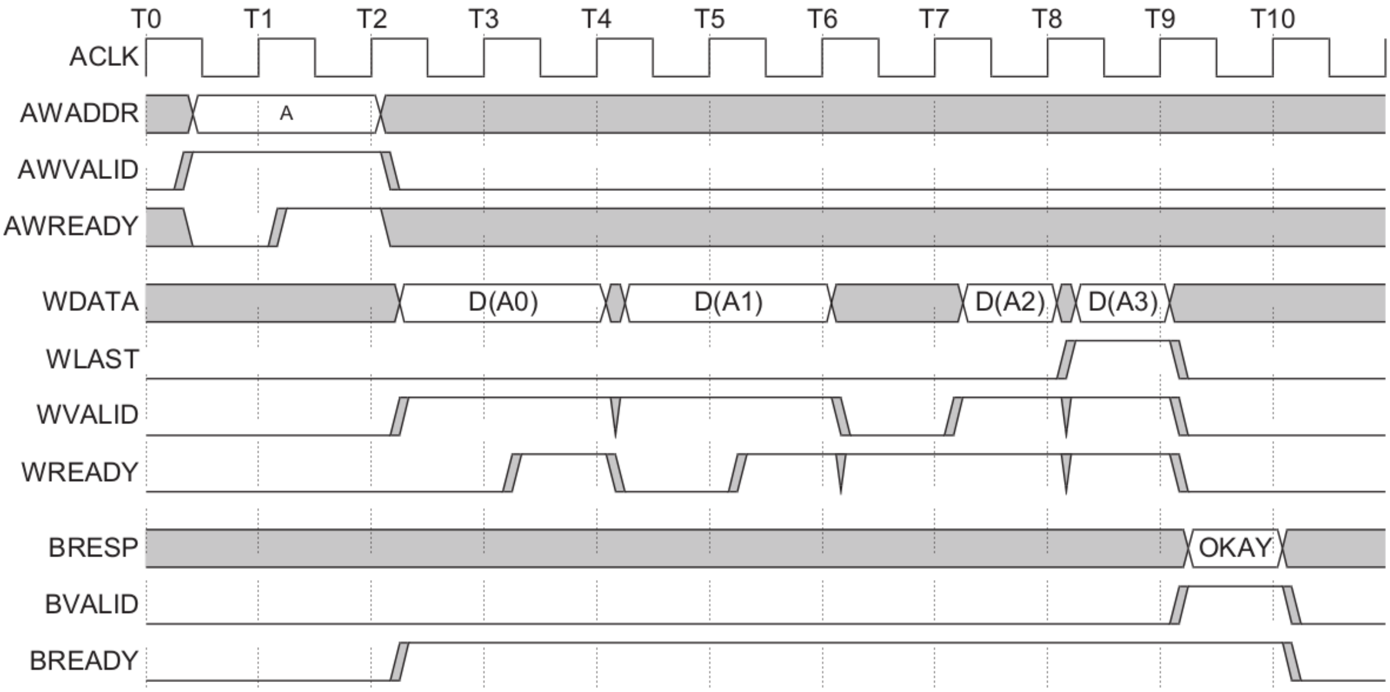


Figure 6: Write Burst Example

- The *Write address*, *Write Data* and *Write Response* channel are used for this write transaction.
- The process is started by master placing the address and control information onto the *write address* channel.
  - Master places the address onto the **AWADDR** bus and mkes the **AWVALID** singal HIGH.
  - Now, the slaves accepts the address from master after one cycle by making the **AWREAD** signal HIGH.
- Also, the master makes the **BREADY** singal HIGH in the *write response* channel for getting the response of this write transaction.
- After the slave responds by making the **AWREADY** HIGHT, the master starts sending the data and control information in the *write data* channel.
  - THe master places the first data to be sent onto to the **WDATA** bus and makes the **WVALID** signal HIGH.
  - After one cycle, the slave accepts the data from master by making the **WREADY** signal HIGH.
  - After accepting the data, the slave mkaes the **WREADY** singal LOW for processing.
- Now, the master sends the next data onto the **WDATA** bus and makes the **WVALID** singal HIGH.
- The above process is repeated for further data.
- When sending the last data, the master places the data onto the **WDATA** bus and makes the **WVALID** singal and **WLAST** singal HIGH.

- The slave makes the **WREADY** for the last time for that write transaction and accepts the data.
- Now, the slave responds to master by driving the *Write Response* channel to indicate the completion of write transaction.
  - The slave makes the **BRESP** signal to be *OKAY* to indicate the success and makes the **BVALID** singal HIGH.

2 Signal Description

2.1 Global Signals

- **ACLK** - Global AXI clock - All signals are sampled at rising edge of this Global clock.
- **ARESETn** - GLobal Reset - Active LOW global reset.

2.2 Write Address channel

Signals	Source	Name	Description
AWID[3:0]	Master	Write Address ID	ID tag for write address group of signal.
AWADDR[31:0]	Master	Write Address	The actual write address for first transfer in a write burst transaction.
AWLEN[3:0]	Master	Burst length	Number of data transfers in a burst associated with address.
AWSIZE[2:0]	Master	Burst size	Size of each data transfer in a burst.
AWBURST[1:0]	Master	Burst Type	Burst type along with size information calculates addresses for each transfer withing burst.
AWLOCK[1:0]	Master	Lock Type	Atomic Characteristics of transfer.
AWCACHE[3:0]	Master	Cache Type	indicates the bufferable, cacheable, write-through, write-back and allocate attributes of the transaction.
AWPROT[2:0]	Master	Projection Type	indicates the normal, privileged, or secure protection level of the transaction and whether the transaction is a data access or an instruction access.
AWVALID	Master	Write address valid	1 - address and control information are available and valid. Address and control information should be stable until address acknowledgemnt signal **AWREADY** goes HIGH.
AWREADY	Slave	Write address ready	1 - salve ready to accept an address and associated control signal.

2.3 Write Data channel

Signals	Source	Name	Description
WID[3:0]	Master	Write ID	ID tag for write data group of signal. **WID** must be equal to **AWID** for write transaction.
WDATA[31:0]	Master	Write Data	The actual write data. Can be 8, 16, 32, 64, 128, 256, 512 or 1024 bits wide
WSTRB[3:0]	Master	Write Strobes	indicates which byte lanes to update in memory. One write strobe for each eight bits of write data bus. $$WSTRB[n] = WDATA[(8*n)+7:(8*n)]$$
WLAST	Master	Write last	indicate the last data transfer in a write burst.
WVALID	Master	Write valid	1 - write data and stobes are available and valid.
WREADY	Slave	Write ready	1 - salve ready to accept write data.

2.4 Write Response channel

Signals	Source	Name	Description
BID[3:0]	Slave	Response ID	ID tag for write response group of signal. **BID** must be equal to **AWID** for write transaction.
BRESP[1:0]	Slave	Write Response	indicates the status of write transaction and can take *OKAY*, *EXOKAY*, *SLVERR* and *DECERR*.
BVALID	Slave	Write responds valid	1 - write response is available and valid.
BREADY	Master	Write response ready	1 - master ready to accept response information.

2.5 Read Address channel

Signals	Source	Name	Description
ARID[3:0]	Master	Read address ID	ID tag for read address group of signal.
ARADDR[31:0]	Master	Read Address	The actual read address for first transfer in a read burst transaction.
ARLEN[3:0]	Master	Burst length	Number of data transfers in a burst associated with address.
ARSIZE[2:0]	Master	Burst size	Size of each data transfer in a burst.
ARBURST[1:0]	Master	Burst Type	Burst type along with size information calculates addresses for each transfer within burst.
ARLOCK[1:0]	Master	Lock Type	Atomic Characteristics of transfer.
ARCACHE[3:0]	Master	Cache Type	indicates cacheable attributes of the transaction.
ARPROT[2:0]	Master	Projection Type	indicates the protection unit information transaction.
ARVALID	Master	Read address valid	1 - address and control information are available and valid. Address and control information should be stable until address acknowledgemnt signal **ARREADY** goes HIGH.
ARREADY	Slave	Read address ready	1 - salve ready to accept an address and associated control signal.

2.6 Read Data channel

Signals	Source	Name	Description
RID[3:0]	Slave	Read ID	ID tag for read data group of signal. **RID** must be equal to **ARID** for read transaction.
RDATA[31:0]	Slave	Read Data	The actual read data. Can be 8, 16, 32, 64, 128, 256, 512 or 1024 bits wide
RRESP[1:0]	Slave	Read Response	indicate the status of read transfer and can be *OKAY*, *EXOKAY*, *SLVERR* and *DECERR*.
RLAST	Slave	Read last	indicate the last data transfer in a read burst.
RVALID	Slave	Read valid	1 - read data are available and valid. read transfer is complete
RREADY	Master	Read ready	1 - master ready to accept read data and response information.

3 Channel Handshake

3.1 Handshake Procedure

- All five channels use the **VALID** and **READY** singal for handshaking.
- Control the rate at which data and control information moves.
- The source generates the **VALID** signal indicating the data or control information is available.
- The Destination generates the **READY** singal indicating that it can accept the data or control information.
- **Transfer occurs only when both VALID and READY signals are HIGH.**

3.1.1 Example Handshake sequence

VALID before READY

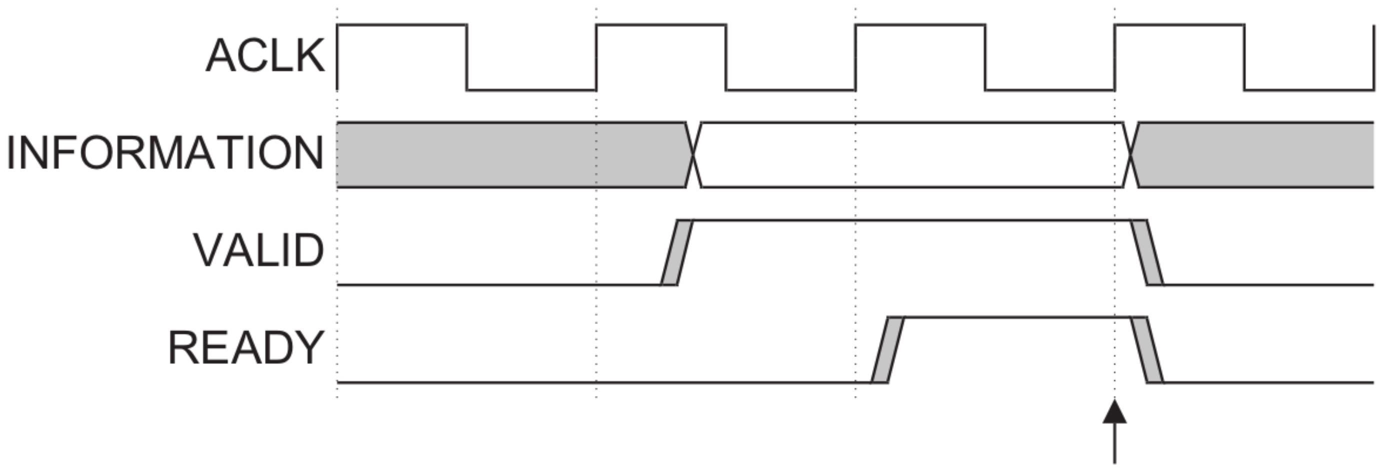


Figure 7: VALID before READY

- The source presents the data or control information and drives the **VALID** signal HIGH.
- The data or control information from the source remains stable until the destination drives the READY signal HIGH, indicating that it accepts the data or control information.
- The arrow shows when the transfer occurs.

READY before VALID

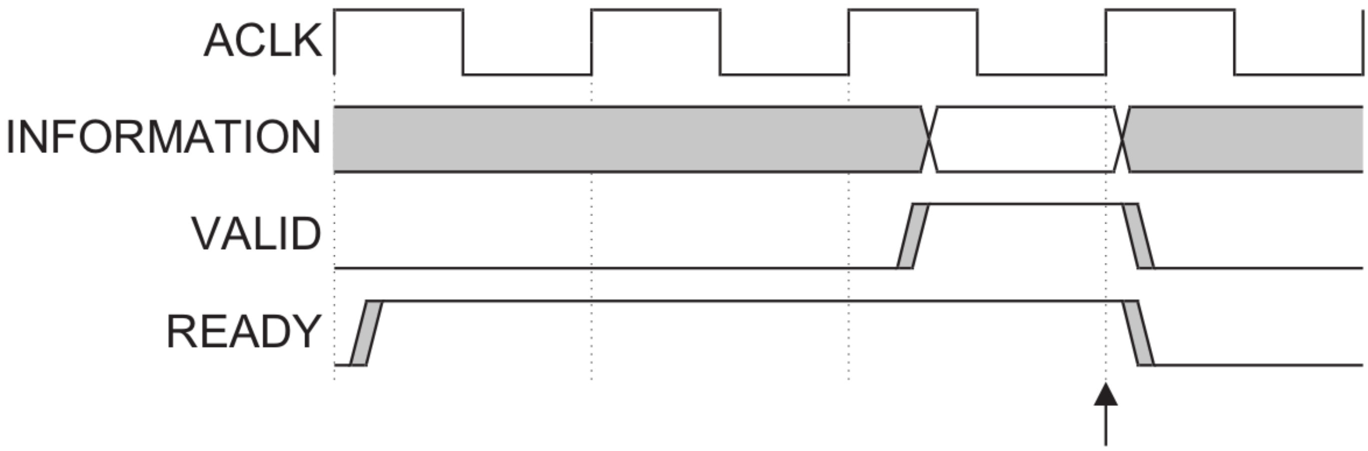


Figure 8: VALID before READY

- The destination drives **READY** HIGH before the data or control information is valid.
- This indicates that destination can accept data or control information in a sigle cyle as soon as it becomes valid.
- The arrow shows when the transfer occurs.

VALID with READY

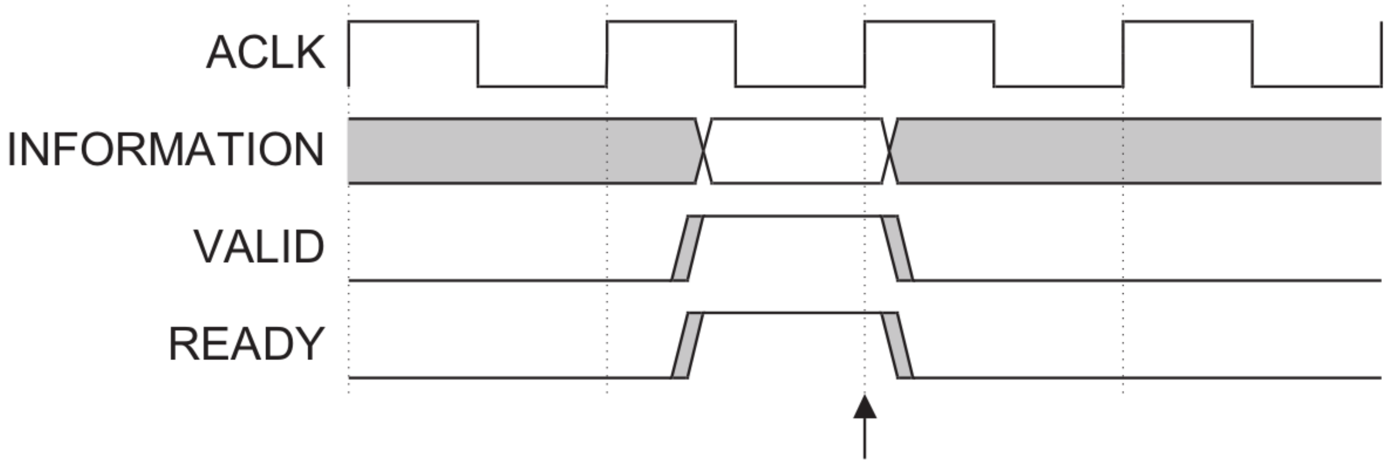


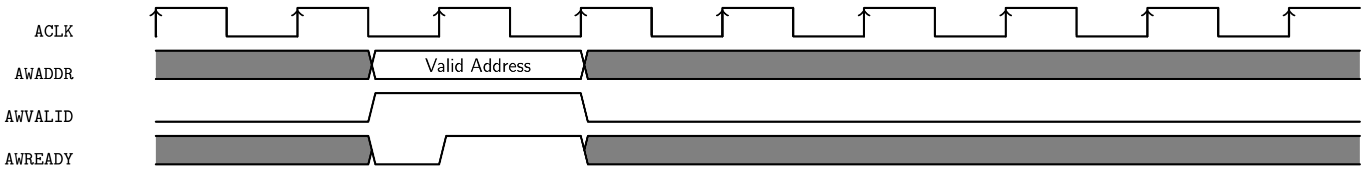
Figure 9: VALID before READY

- Both the source and destination happns to indicate in same cycle that they can transfer data or control information.
- The transfer occurs immediately.
- The arrow shows when the transfer occurs.

Notes: In all the above cases, we can see the **VALID** is made high only when data is valide.

3.2 Write Address Channel

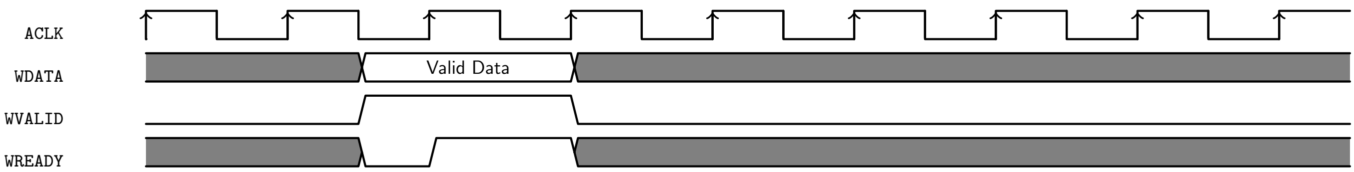
- Master places the valid addresses and control information and makes the **AWVALID** singal HIGH.
- The address and control information and **AWVALID** remains in that statue until slave accepts the address and control information.
- Now the salve makes the **AWREADY** singal HIGH.



Note: The default value for **AWREADY** should be HIGH.

3.3 Write Data channel

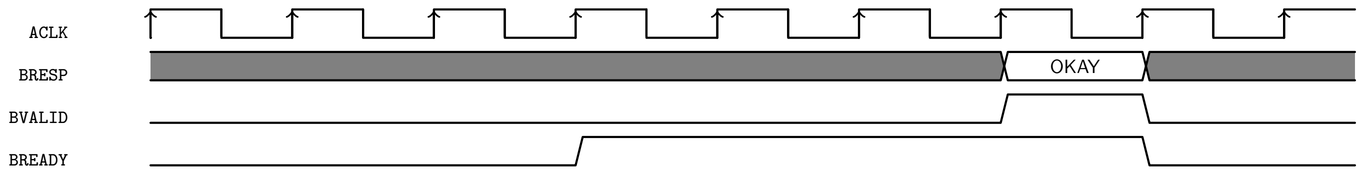
- Master places the valid write data and makes the **WVALID** signal HIGH.
- This **WVALID** singal must remain in that state untill the slave accepts the write data and makes the **WREADY** singal HIGH.
- The master must drive **WLAST** singal HIGH when writing the final write data transfer in a burst.
- When the **WVALID** is LOW, **WSTRB[3:0]** must be LOW or held at previous value.



Note: The default value for **WREADY** should be HIGH only if the slave can always accept write data in a single cycle.

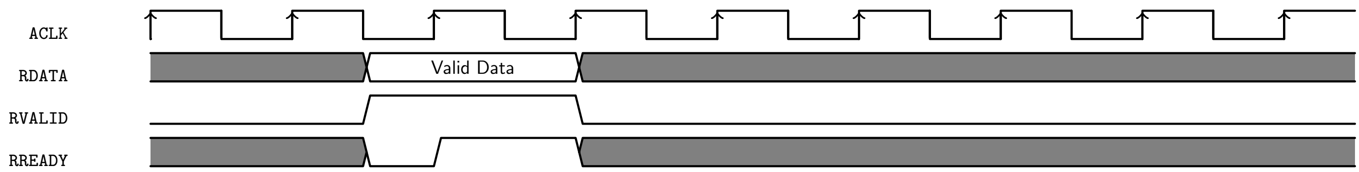
3.4 Write Response channel

- Slave makes the **BVALID** singal HIGH when it drives the valid write response.
- **BVALID** must be HIGH until the master accpets the response and asserts the **BREADY** singal.
- The default value of **BREADY** is HIGH only if master can always accept the write response in a single cycle.



3.5 Read Data channel

- Slave places the valid read data and makes the **RVALID** signal HIGH.
- This **RVALID** singal must remain in that state untill the master accepts the read data and makes the **RREADY** singal HIGH.
- The slave must drive **RLAST** singal HIGH indicating that its transferig the last data transfer in the read burst.



Note: The default value for **RREADY** should be HIGH only if the master can always accept the read data immediately..

3.6 Relationship of handshake signals among channels

- Make **READY** signal HIGH before **VALID** for better efficiency.
- The single-headed arrow point to signal that can be asserted before or after the previous signal.
- The double-headed arrow point to signal that must be asserted only after the previous signal.
- For read transaction,

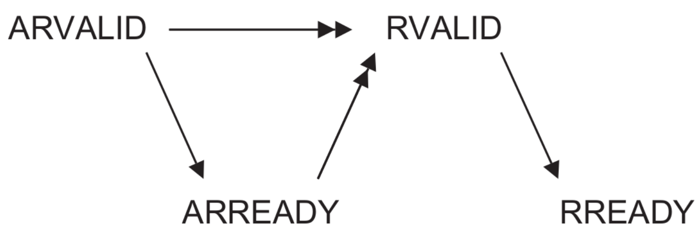


Figure 10: Read transaction handshake dependencies

- the slave can wait for **ARVALID** to be asserted before it asserts **ARREADY** singal.
- the slave must wait for both **ARVALID** and **ARREADY** to be asserted befoere it starts to return data by asserting **RVALID**.
- For Write transaction,

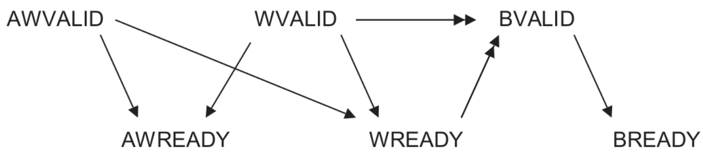


Figure 11: Write transaction handshake dependencies

- the master must not wait for the slave to assert **AWREADY** or **WREADY** before asserting **AWVALID** or **WVALID**
- the slave can wait for **AWVALID** or **WVALID**, or both, before asserting **AWREADY** and/or **WREADY**
- the slave must wait for both **WVALID** and **WREADY** to be asserted before asserting **BVALID**

4 Addressing Options

- AXI protocol is burst-based.
- THe master begin each burst by driving transfer control information and address of the first byte in the transfer.
- As the burst progresses, the slave must calculate the address of subsequent transfers in the burst.
- Burst must not cross 4KB boundaries.

4.1 Burst Length

- The **AWLEN** or **ARLEN** singal specifies the Number of data transfer withing each burst.
- The **AWLEN** or **ARLEN** singal can take values between 1(4'b0000) to 16(4'b1111).

4.2 Burst Size

- The **AWSIZE** or **ARSIZE** singal specifies the Number of data bytes in each beat or data transfer.
- The **AWSIZE** or **ARSIZE** singal can take values as shown

ARSIZE[2:0] or **AWSIZE[2:0]**	**Bytes in each transfer**
3'b000	1
3'b001	2
3'b010	4
3'b011	8
3'b100	16
3'b101	32
3'b110	64
3'b111	128

- AXI determines the trandfer address and also which byte lane to use.

4.3 Burst Type

ARBURST[1:0] or AWBURST[1:0]	**Burst type**	**Access**	**Descritpion**
2'b00	FIXED	FIFO-type	Fixed-address burst – address remains same for every data transfer in the burst; Used in case of repeated acesses to same location such as when loading or emptying peripheral FIFO.
2'b01	INCR	Normal sequential memory	incrementing-address burst – the address for each transfer in the burst in an increment of the previous transfer address. The incrementing value depends on the size of the transfer. If the burst size is 4, then the address of each transfer is previous address plus 4
2'b10	WRAP	Cache line	incrementing-address burst that wraps to a lower address at wratp boundaries – wrap boundarry is the size of each transfer in burst multiplied by total number of transfer in the burst.

4.4 Burst address

The Variables used are

- *Start_Address* - Start address issued by the master.
- *Numbe_Bytes* - Maximum Number of bytes in each data transfer.
- *Data_Bus_Bytes* - Number of byte lanes in the data bus.
- *Aligned_Addres* - Aligned version of start address.
- *Burst_Length* - total Number of data transfers within a burst.
- *Address_N* - address of transfer N within a burst.
- *Wrap_Boundary* - Lowest address withing a wrapping burst.
- *Lower_Byte_Lane* - Byte lane of lowest addressed byte of a transfer.
- *Upper_Byte_Lane* - Byte lane of highest addressed byte of a transfer.
- *INT(x)* - Rounded-down integer value of x.
- Addresses of tranfers within a burst
  - $Start\_Address = ADDR$
  - $Number\_Bytes = 2^{SIZE}$
  - $Burst\_Lenght = LEN + 1$
  - $Aligned\_Address = INT(\frac{Start\_Address}{Number\_Bytes}) * Number\_Bytes$
- First Address
  - $Address\_1 = Start\_Address$
- Address of any tranfer after the first transfer in a burst
  - $Address\_N = Aligned\_Address + (N - 1) * Number\_Bytes$
- The wrapping boundary in wrapping burst
  - $Wrap\_Boundary = INT(\frac{Start\_Address}{Number\_Bytes*Burst\_Length}) * Number\_Bytes * Burst\_Length$
  - $Address\_N = Wrap\_Boundary.$
- Data is transferred on
  - $DATA[(8 * Upper\_Byte\_Lane) + 7 : (8 * Lower\_Byte\_Lane)]$

5 Basic Interface for Master

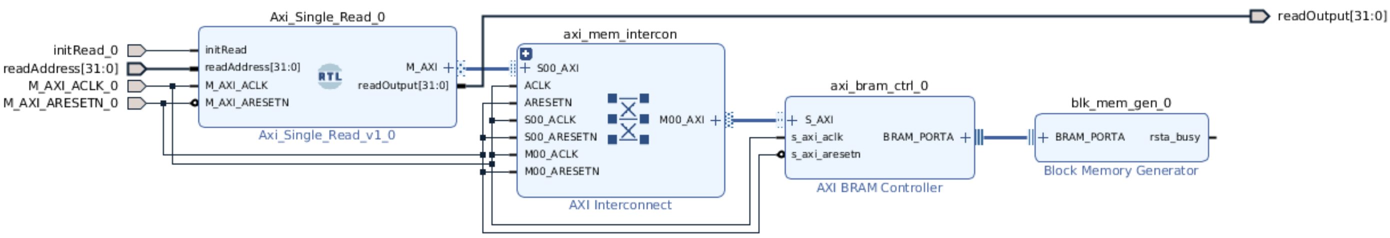
```
//Global Signals
input wire  M_AXI_ACLK,
input wire  M_AXI_ARESETN,
// Write address interface (issued by master)
output wire [32-1 : 0] M_AXI_AWADDR,
output wire [2 : 0] M_AXI_AWPROT,
output wire  M_AXI_AWVALID,
input wire  M_AXI_AWREADY,
// Write Data interface (issued by master)
output wire [32-1 : 0] M_AXI_WDATA,
output wire [32/8-1 : 0] M_AXI_WSTRB,
output wire  M_AXI_WVALID,
input wire  M_AXI_WREADY,
// Write Response interface (issued by slave)
input wire [1 : 0] M_AXI_BRESP,
input wire  M_AXI_BVALID,
```

```
output wire  M_AXI_BREADY,
// Read Address interface (issued by master)
output wire [32-1 : 0] M_AXI_ARADDR,
output wire [2 : 0] M_AXI_ARPROT,
output wire  M_AXI_ARVALID,
input wire  M_AXI_ARREADY,
// Read Data interface (issued by slave)
input wire [32-1 : 0] M_AXI_RDATA,
input wire [1 : 0] M_AXI_RRESP,
input wire  M_AXI_RVALID,
output wire  M_AXI_RREADY
```

6 AXI Master Single Read

- A single read using AXI protocol.
- Uses just the *read address* and *read data* channel.
- Manipulating the **ARADDR**, **ARVALID** and **RREADY** signals only.
- We read data from BRAM using AXI interface.
- Initial content of BRAM is the Sample Mem coe file.

6.1 Block Diagram

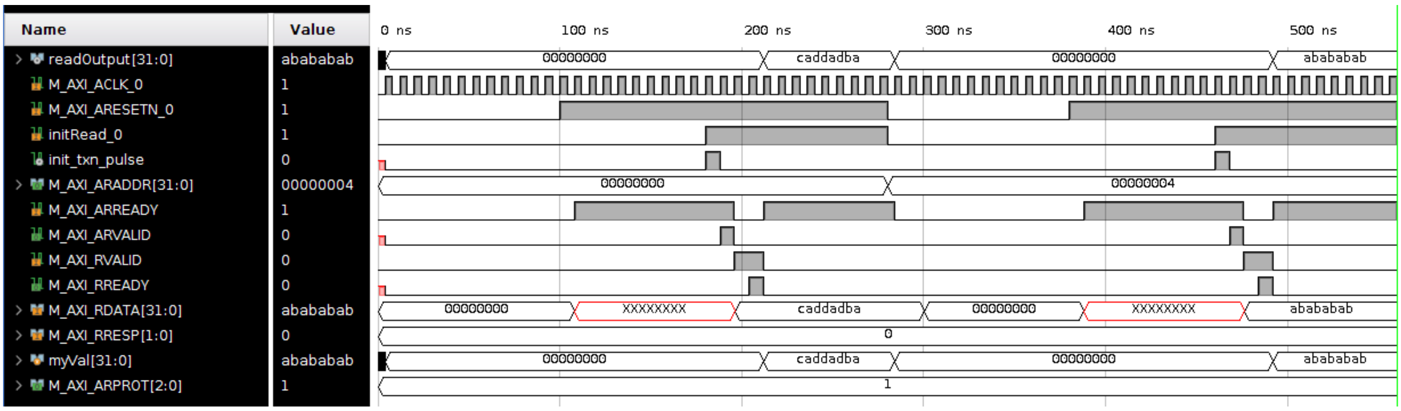


6.2 Code

```
// for ARVALID
always @(posedge M_AXI_ACLK)
    begin
        if (M_AXI_ARESETN == 0)
          begin
            axi_arvalid <= 1'b0;
          end
        else if (init_txn_pulse == 1'b1)
          begin
            axi_arvalid <= 1'b1;
          end
        else if (M_AXI_ARREADY==1 && axi_arvalid==1)
          begin
            axi_arvalid <= 1'b0;
          end
    end

// for RREADY
always @(posedge M_AXI_ACLK)
    begin
        if (M_AXI_ARESETN == 0)
```

```
        begin
          axi_rready <= 1'b0;
        end
      else if (M_AXI_RVALID==1 && axi_rready==0)
        begin
          axi_rready <= 1'b1;
        end
      else if (axi_rready==1)
        begin
          axi_rready <= 1'b0;
        end
  end

// Reading the data
reg [31:0]myVal;
always @(posedge M_AXI_ACLK)
  begin
    if (M_AXI_ARESETN == 0)
        myVal <= 1'b0;
    else if (M_AXI_RVALID==1 && axi_rready==1)
      myVal <= M_AXI_RDATA;
    else
      myVal <= myVal;
  end
```

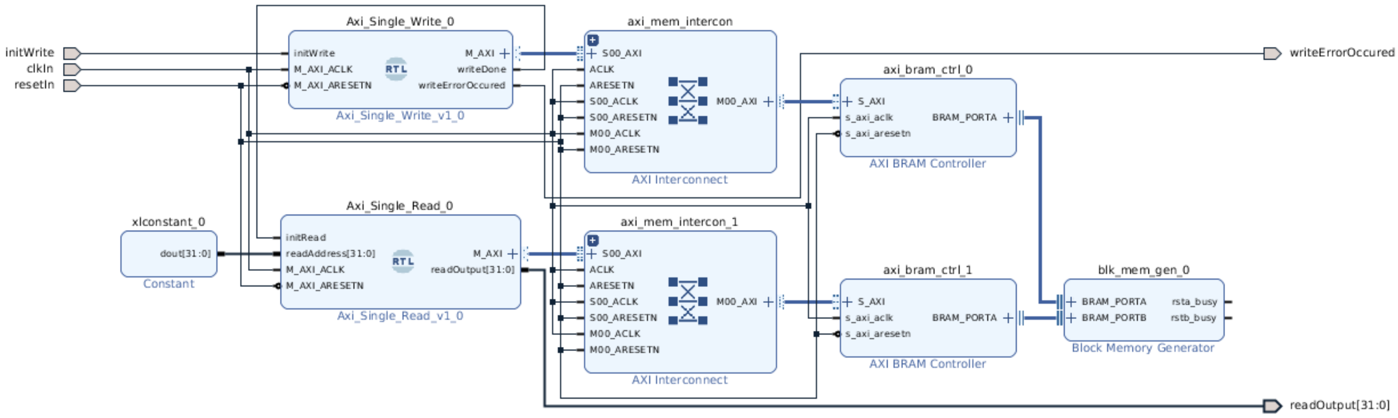
6.3 Output



7 AXI Master Single Write

- A single write using AXI protocol.
- Uses just the *write address*, *write data* and *write response* channel.
- Manipulating the **AWADDR**, **AWVALID**, **WDATA**, **WVALID** and **BREADY** signals only.
- We write data from BRAM using AXI interface and also read from the BRAM.
- Output can be seen in readOutput.

7.1 Block Diagram

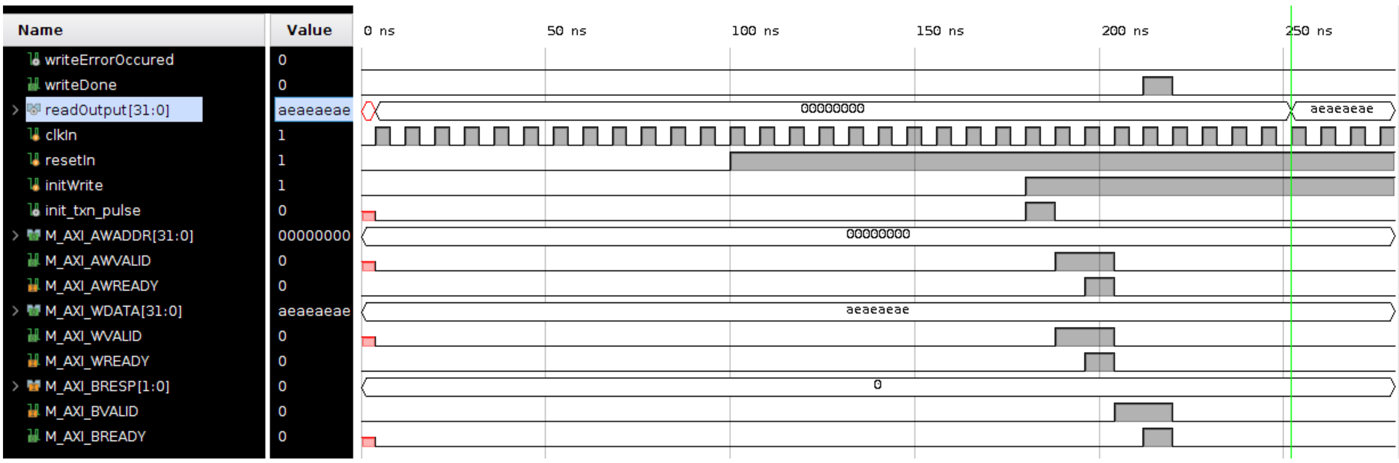


7.2 Code

```
// for AWVALID
always @(posedge M_AXI_ACLK)
    begin
    if (M_AXI_ARESETN == 0)
        begin
        axi_awvalid <= 1'b0;
        end
    else
        begin
        if ( init_txn_pulse == 1'b1)
            begin
            axi_awvalid <= 1'b1;
            end
        else if (M_AXI_AWREADY && axi_awvalid)
            begin
            axi_awvalid <= 1'b0;
            end
        end
    end

// for WVALID
always @(posedge M_AXI_ACLK)
    begin
        if (M_AXI_ARESETN == 0 )
        begin
            axi_wvalid <= 1'b0;
        end
        else if (init_txn_pulse == 1'b1)
        begin
            axi_wvalid <= 1'b1;
        end
        else if (M_AXI_WREADY && axi_wvalid)
        begin
        axi_wvalid <= 1'b0;
        end
    end
```

```
// for BREADY
always @(posedge M_AXI_ACLK)
    begin
    if (M_AXI_ARESETN == 0)
        begin
        axi_bready <= 1'b0;
        end
    else if (M_AXI_BVALID==1 && axi_bready==0)
        begin
        axi_bready <= 1'b1;
        end
    else if (axi_bready==1)
        begin
        axi_bready <= 1'b0;
        end
    else
        axi_bready <= axi_bready;
    end

assign writeErrorOccured = (axi_bready & M_AXI_BVALID & M_AXI_BRESP[1]);
assign writeDone = (axi_bready & M_AXI_BVALID);
```

7.3 Output



8 Basic Interface for Slave

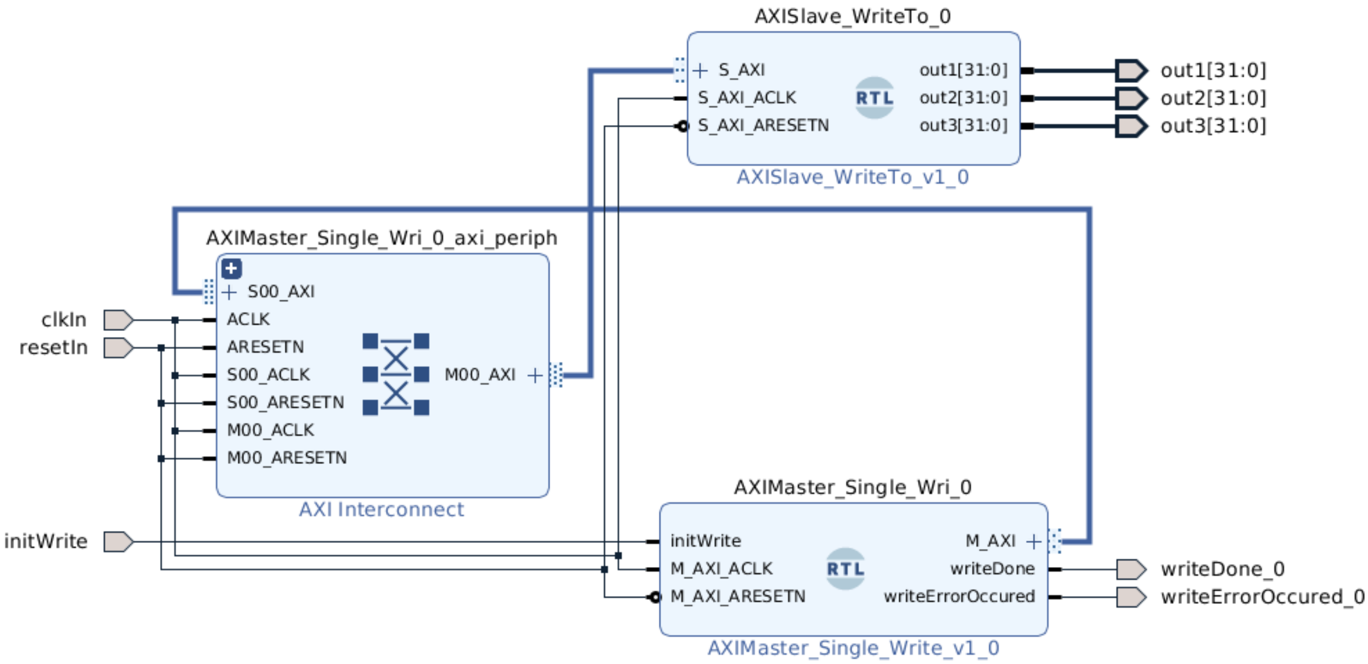
```
//Global Signals
input wire  S_AXI_ACLK,
input wire  S_AXI_ARESETN,
// Write address interface (issued by master)
input wire [32-1 : 0] S_AXI_AWADDR,
input wire [2 : 0] S_AXI_AWPROT,
input wire  S_AXI_AWVALID,
output wire  S_AXI_AWREADY,
// Write Data interface (issued by Master)
input wire [32-1 : 0] S_AXI_WDATA,
input wire [(32/8)-1 : 0] S_AXI_WSTRB,
input wire  S_AXI_WVALID,
output wire  S_AXI_WREADY,
    // Write Response interface (issued by slave)
output wire [1 : 0] S_AXI_BRESP,
```

```
output wire  S_AXI_BVALID,
input wire  S_AXI_BREADY,
// Read Address interface (issued by master)
input wire [32-1 : 0] S_AXI_ARADDR,
input wire [2 : 0] S_AXI_ARPROT,
input wire  S_AXI_ARVALID,
output wire  S_AXI_ARREADY,
// Read Data interface (issued by slave)
output wire [32-1 : 0] S_AXI_RDATA,
output wire [1 : 0] S_AXI_RRESP,
output wire  S_AXI_RVALID,
input wire  S_AXI_RREADY
```

9 AXI Slave Write To

- Master writes onto the slave.
- Uses just the *write address*, *write data* and *write response* channel.
- Manipulating the **AWREADY**, **AWVALID**, **WREADY**, **BVALID** and **BRESP** signals only.
- We write data from Master into the Slave's 3 reg with **AWADDR**.

9.1 Block Diagram



9.2 Code

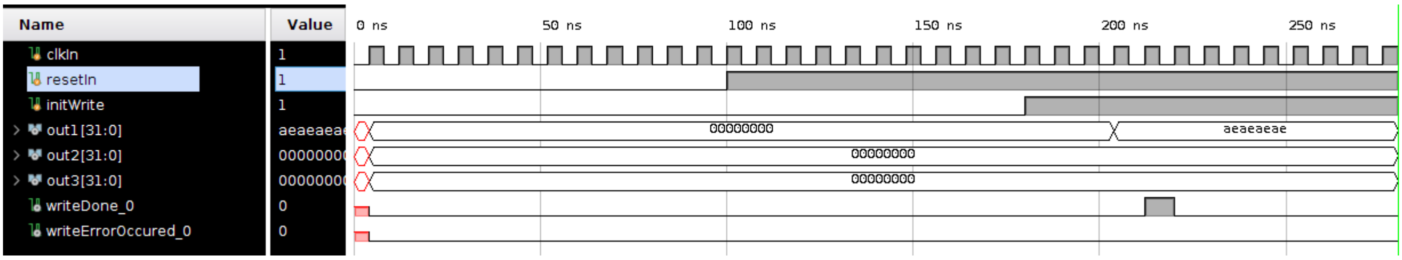
```
// for AWREADY
always @( posedge S_AXI_ACLK )
    begin
        if ( S_AXI_ARESETN == 1'b0 )
            begin
            axi_awready <= 1'b0;
            end
        else
            begin
                if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID)
```

```
                begin
                axi_awready <= 1'b1;
                end
            else if (S_AXI_BREADY && axi_bvalid)
                begin
                axi_awready <= 1'b0;
                end
            else
                begin
                axi_awready <= 1'b0;
                end
        end
    end

// for AWADDR
always @( posedge S_AXI_ACLK )
    begin
        if ( S_AXI_ARESETN == 1'b0 )
            begin
            axi_awaddr <= 0;
            end
        else
            begin
                if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID)
                    begin
                        axi_awaddr <= S_AXI_AWADDR;
                    end
            end
    end

// for WREADY
always @( posedge S_AXI_ACLK )
    begin
        if ( S_AXI_ARESETN == 1'b0 )
            begin
                axi_wready <= 1'b0;
            end
        else
            begin
                if (~axi_wready && S_AXI_WVALID && S_AXI_AWVALID )
                    begin
                        axi_wready <= 1'b1;
                    end
                else
                    begin
                        axi_wready <= 1'b0;
                    end
            end
    end

// for writing WDATA
reg [32-1:0]slv_reg0, slv_reg1, slv_reg2;
integer         byte_index;

always @( posedge S_AXI_ACLK )
```

```
    begin
    if ( S_AXI_ARESETN == 1'b0 )
        begin
        slv_reg0 <= 0;
        slv_reg1 <= 0;
        slv_reg2 <= 0;
        end
    else if (axi_wready && S_AXI_WVALID && axi_awready && S_AXI_AWVALID)
        begin
            case ( axi_awaddr[3:2] )
                2'h0:
                    for ( byte_index = 0; byte_index <= (32/8)-1; byte_index =
                     ↪  byte_index+1 )
                    if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                    // Respective byte enables are asserted as per write strobes
                    // Slave register 0
                    slv_reg0[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +:
                     ↪  8];
                    end
                2'h1:
                    for ( byte_index = 0; byte_index <= (32/8)-1; byte_index =
                     ↪  byte_index+1 )
                    if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                    // Respective byte enables are asserted as per write strobes
                    // Slave register 1
                    slv_reg1[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +:
                     ↪  8];
                    end
                2'h2:
                    for ( byte_index = 0; byte_index <= (32/8)-1; byte_index =
                     ↪  byte_index+1 )
                    if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                    // Respective byte enables are asserted as per write strobes
                    // Slave register 2
                    slv_reg2[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +:
                     ↪  8];
                end
                default : begin
                slv_reg0 <= slv_reg0;
                slv_reg1 <= slv_reg1;
                slv_reg2 <= slv_reg2;
                end
            endcase
        end
    end

// for BVALID and BRESP
always @( posedge S_AXI_ACLK )
    begin
      if ( S_AXI_ARESETN == 1'b0 )
        begin
          axi_bvalid  <= 0;
          axi_bresp   <= 2'b0;
        end
      else
```

```
      begin
        if (axi_awready && S_AXI_AWVALID && ~axi_bvalid && axi_wready &&
         ↪  S_AXI_WVALID)
          begin
            axi_bvalid <= 1'b1;
            axi_bresp  <= 2'b0; // 'OKAY' response
          end
        else
          begin
            if (S_AXI_BREADY && axi_bvalid)
              begin
                axi_bvalid <= 1'b0;
              end
          end
      end
    end

assign out1 = slv_reg0;
assign out2 = slv_reg1;
assign out3 = slv_reg2;
```

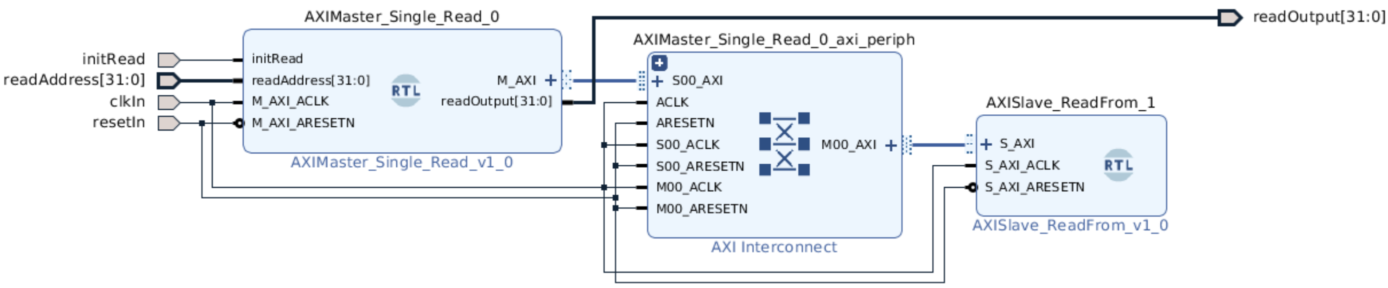
9.3 Output



10 AXI Slave Read From

- Master reads from the slave.
- Uses just the *read address* and *read data* .
- Manipulating the **RDATA**, **AWVALID**, **WREADY**, **BVALID** and **BRESP** signals only.
- We read data from Master into the Slave's 3 reg .

10.1 Block Diagram



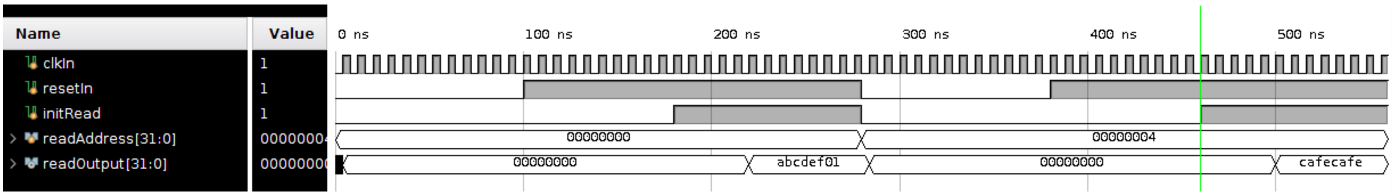
10.2 Code

```
// for ARREADY and ARADDR
always @( posedge S_AXI_ACLK )
```

```
    begin
        if ( S_AXI_ARESETN == 1'b0 )
            begin
            axi_arready <= 1'b0;
            axi_araddr  <= 32'b0;
            end
        else
            begin
                if (~axi_arready && S_AXI_ARVALID)
                    begin
                        // indicates that the slave has acceped the valid read
                        ↪  address
                        axi_arready <= 1'b1;
                        // Read address latching
                        axi_araddr  <= S_AXI_ARADDR;
                    end
                else
                    begin
                        axi_arready <= 1'b0;
                    end
            end
    end

// for RVALID and RRESP
always @( posedge S_AXI_ACLK )
    begin
        if ( S_AXI_ARESETN == 1'b0 )
        begin
            axi_rvalid <= 0;
            axi_rresp  <= 0;
        end
        else
        begin
            if (axi_arready && S_AXI_ARVALID && ~axi_rvalid)
            begin
                axi_rvalid <= 1'b1;
                axi_rresp  <= 2'b0; // 'OKAY' response
            end
            else if (axi_rvalid && S_AXI_RREADY)
            begin
                axi_rvalid <= 1'b0;
            end
        end
    end

// address decoding
reg [31:0]reg_data_out;
always @(*)
    begin
            // Address decoding for reading registers
            case ( axi_araddr[3:1] )
            2'h0   : reg_data_out <= 32'habcdef01;
            2'h1   : reg_data_out <= 32'hcafecafe;
            2'h2   : reg_data_out <= 32'hd00dd00d;
            default : reg_data_out <= 0;
```

```
            endcase
    end

// for RDATA
always @( posedge S_AXI_ACLK )
    begin
        if ( S_AXI_ARESETN == 1'b0 )
            begin
                axi_rdata  <= 0;
            end
        else
            begin
                if (axi_arready & S_AXI_ARVALID & ~axi_rvalid)
                begin
                    axi_rdata <= reg_data_out;     // register read data
                end
        end
    end
```

10.3 Output



11 UPDATED (June 4)