Notes - Unit 5

Embedded Systems in All-Programmable SoC

AXI Bus

References:
- Zynq™ Book
- AXI4 Specification
- Connecting User Logic to AXI Interfaces of High-Performance Communication Blocks in the SmartFusion2 Devices – Libero SoC v11.4.

AXI4-Full Interface

- The AXI protocol is burst-based and defines five independent transaction channels.
- Write Channel Architecture: Address and Control data is transmitted to the slave before a burst of data is transmitted, and a Write Response signaled following completion:
  - Write Address Channel
  - Write Data Channel
  - Write Response Channel
- Read Channel Architecture: Address and Control data transmitted to the slave before a burst of read data is transmitted to the master:
  - Read Address Channel
  - Read Data Channel
- Data can move in both directions simultaneously.
- Data transfer size: up to 256 data transfers (burst transactions).
- AXI4-Lite: One data transfer per transaction. Burst is not supported
- AXI4-Stream: One single channel for transmission of streaming data. It can burst an unlimited amount of data.

- Write/Read Data Channel: The data bus can be: 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide.
- Burst Size: This is defined by the signals S_AXI_AWSIZE and S_AXI_ARSIZE. They can have the values 000 (1 byte), 001 (2 bytes), 010 (4 bytes), 011 (8 bytes), and 100 (16 bytes = 128 bits).
  The Burst Size must not exceed the Data Bus Width. If the AXI Width is greater than the Burst size, the AXI interface must determine from the transfer address which byte lanes of data bus to use for each transfer (when writing, this can be done using the WSTRB signal).
  As a good rule of thumb, make the Burst Size the same as the Write/Read Data Channel.
- Burst type: Defined by S_AXI_AWBURST and S_AXI_ARBURST. 00: FIXED (address remains constant during transaction), 01: INCR (address increments depending on the transaction size), 10: WRAP. This is for the address inside the peripheral where data should be placed. It is up to the recipient of the data to implement this feature.
- Burst Length: This is defined by the S_AXI_AWLEN and S_AXI_ARLEN signals. It provides the exact number of transfers in a burst. 1-256 (0x00 – 0xFF) for the INCR burst type. For all the other burst types, only 1-16 are supported. (It seems that in Zynq, burst can only be up to 16 words.)
• **Signals:**
  - Global System Signals:
    - S_AXI_CLK: AXI4 clock
    - S_AXI_ARESETN: AXI4 active-low reset.

Each of the five channels has their own set of respective signals:

**WRITE ADDRESS CHANNEL**

- AXI MASTER
  - S_AXI_AWID
  - S_AXI_AWADDR
  - S_AXI_AWLEN
  - S_AXI_AWSIZE
  - S_AXI_AWBURST
  - S_AXI_AWLOCK
  - S_AXI_AWCACHE
  - S_AXI_AWPORT
  - S_AXI_AWQS
  - S_AXI_AWREGION
  - S_AXI_AWUSER
  - S_AXI_AWVALID
  - S_AXI_AWREADY

- AXI SLAVE

**WRITE DATA CHANNEL**

- AXI MASTER
  - S_AXI_WDATA
  - S_AXIWSTRB
  - S_AXI_WLAST
  - S_AXI_WUSER
  - S_AXI_WVALID
  - S_AXI_WREAY

- AXI SLAVE

**WRITE RESPONSE CHANNEL**

- AXI MASTER
  - S_AXI_BID
  - S_AXI_BRESP
  - S_AXI_BUSER
  - S_AXI_BVALID
  - S_AXI_BREADY

- AXI SLAVE

**READ ADDRESS CHANNEL**

- AXI MASTER
  - S_AXI_ARID
  - S_AXI_ARADDR
  - S_AXI_ARLEN
  - S_AXI_ARSIZE
  - S_AXI_ARBURST
  - S_AXI_ARLOCK
  - S_AXI_ARCACHE
  - S_AXI_ARPORT
  - S_AXI_AQOS
  - S_AXI_ARREGION
  - S_AXI_ARUSER
  - S_AXI_ARVALID
  - S_AXI_ARREADY

- AXI SLAVE

**READ DATA CHANNEL**

- AXI MASTER
  - S_AXI_RDATA
  - S_AXI_RRESP
  - S_AXI_RLAST
  - S_AXI_RUSER
  - S_AXI_RVALID
  - S_AXI_RREADY

- AXI SLAVE

**AXI4-FULL PROTOCOL**

- The VALID/READY handshake process is used by all five transaction channels (‘Assert and Wait’ Rule)
- VALID: Generated by the source only when information (address, data, and control) is available.
- READY: Generated by the destination to indicate it can accept information.
- Transfer occurs on the rising clock edge when VALID=READY=1. At that moment, VALID becomes 0 followed by READY becoming 0. * A source is not permitted to wait until READY is asserted before asserting VALID.
Writing Transaction – Simple Memory:

- The AXI master sends the write address (along with burst information) via the Write Address Channel. Then, it writes data via the Write Data Channel. Finally, the Slave sends the response via the Write Response Channel.

- **Write Address Channel Handshake**: The AXI Master asserts the AWVALID signal only when it drives valid Address and Control information. The signals remain asserted until the AXI Slave accepts the Address and Control information and asserts the associated AWREADY signal (at this moment, it captures the Address and Control).

- **Write Data Channel Handshake**: The AXI Master asserts the WVALID signal only when it drives valid write data. The WVALID signal remains asserted until the AXI Slave accepts the write data by asserting the WREADY signal (this is when data is captured). If the burst is greater than 1, when WREADY is asserted, the AXI Master must place another data on the bus, assert WVALID and wait until WREADY is asserted. The process continues until all the bursts are completed (the last burst is signaled by WLAST). Notice that the AXI Master controls when to assert WVALID in a burst. The figure shows that after the first data (D(A0)), the next three data (Burst Length = 4) are issued every clock cycle.

- **Write Response Channel Handshake**: The AXI Slave asserts the BVALID signal only when it drives the valid response BRESP. This happens when the bursts have been completed. The BVALID signal remains asserted until the AXI Master asserts BREADY (here, the Master captures BRESP). Note that the master can assert BREADY before the slave asserts BVALID. This helps the completion of the operation in one cycle, as BVALID cannot be waiting on BREADY.

- The figure below shows the case for a simple memory system: Data is written starting from the address provided on S_AXI_AWADDR. The internal circuitry is in charge of incrementing the address (if in INCR or WRAP mode).

Reading Transaction – Simple Memory:

- The AXI master sends the read address (along with burst information) via the Read Address Channel. Then, the Slave sends Read Data Back via the Read Data Channel.

- **Read Address Channel Handshake**: The AXI Master asserts ARVALID only when it drives valid address and control information. It remains asserted until the AXI slave accepts the address and control information and asserts the associated ARREADY signal (here is when address and control are captured).

- **Read Data Channel Handshake**: The AXI Master asserts RVALID only when it drives the valid read data. The RVALID signal remains asserted until the AXI Master accepts the data by asserting the RREADY signal (here data is captured). If the burst is greater than 1, when RREADY is asserted, the AXI Slave must place another data on the bus, assert RVALID and wait until RREADY is asserted. The process continues until all the bursts are completed (the last burst is signaled by RLAST). Notice that the AXI Slave controls when to assert RVALID in a burst. The figure shows that after the each data, we wait one cycle before issuing the next data.

- The figure below shows the case for a simple memory system: Data is written starting from the address provided on S_AXI_ARADDR. The internal circuitry is in charge of incrementing the address (if in INCR or WRAP mode).
AXI4-LITE INTERFACE
- This is a reduced version of the AXI4-Full. It does not support bursts, i.e., we only have one transaction at a time.
- Data bus: 32 or 64 bits.

**AXI4-LITE PROTOCOL**
- The AXI Master Interface provided by Zynq in Vivado sends both the Write Address and Write Data at the same time. When Reading, the Master first requests to read an address and the AXI Slave responds with data.
Write cycle and Read Cycle (Xilinx AXI4-Lite, from Master’s point of view)

- S_AXI_AWREADY: Registered signal asserted for one clock cycle when S_AXI_AWVALID=S_AXI_WVALID='1' (this can happen immediately or after a few cycles).
- S_AXI_WREADY: Registered signal that is asserted for one clock cycle when S_AXI_AWVALID=S_AXI_WVALID='1' (this can happen immediately or after a few cycles).
- S_AXI_AWADDR: It is captured into axi_awaddr when S_AXI_AWVALID=S_AXI_WVALID='1', S_AXI_AWREADY='0'.
- S_AXI_ARREADY: It is asserted for one clock cycle when S_AXI_ARVALID is asserted (it can happen immediately or after a few cycles).
- S_AXI_ARADDR: It is captured into the axi_araddr signal when S_AXI_ARVALID = '1' and S_AXI_ARREADY = '0'.
- S_AXI_RVALID: It is asserted for one clock cycle right after both S_AXI_ARVALID and S_AXI_ARREADY are detected to be '1'. During that clock cycle, S_AXI_RREADY is still '1' (due to the AXI specification), so when S_AXI_RVALID becomes zero, S_AXI_RREADY follows suit and becomes zero.
**AXI4 INTERFACE - EXAMPLES**

- **AXI4-Lite Interface (Slave):** Vivado 2015.3 provides a template based on the number of Slave Registers that the user specifies (4 by default). The template on its own can be used to write data on Slave Registers and read data from them in order to verify the functioning of the embedded system. In our case example, we have to modify the template to include our hardware.

- **AXI4-Full Interface (Slave):** Vivado 2015.3 provides a template based on the number of bytes selected (64 by default). The template is a 64-bytes memory where we can read and write data using bursts. We need to modify this circuit by including our hardware.

### AXI4-LITE: PIXEL PROCESSOR

- Simple interface with two slave registers for reading and writing on the Pixel Processor:
  - `slv_reg_wren`: It indicates that new data is available on a Slave Register.
    
    \[
    \text{slv}\_\text{reg}\_\text{wren} = S\_\text{AXI}\_\text{WREADY}\text{ and } S\_\text{AXI}\_\text{VALID}\text{ and } S\_\text{AXI}\_\text{AWREADY}\text{ and } S\_\text{AXI}\_\text{AWVALID}.
    \]
  - `slv_reg_rden`: It indicates that data is being read from a Slave Register.
    
    \[
    \text{slv}\_\text{reg}\_\text{rden} = S\_\text{AXI}\_\text{ARREADY}\text{ and } S\_\text{AXI}\_\text{ARVALID}\text{ and } (\text{not } S\_\text{AXI}\_\text{RVALID}).
    \]
  - `axi_aw_addr`: Latched address (from `S_AXI_AWADDR`) that specifies a Slave Register. In the example, we have 4-bit addresses, where each address specifies a particular byte. This is, the 2 LSBs indicate individual bytes within a 32-bit word. As a Slave Register is 32-bits wide, we only need `axi_aw_addr(3..2)` to specify a particular slave register.
  - `axi_ar_addr`: Latched address (from `S_AXI_ARADDR`) that specifies a Slave Register. In the example, we have 4-bit addresses, where each address specifies a particular byte. This is, the 2 LSBs indicate individual bytes within a 32-bit word. As a Slave Register is 32-bits wide, we only need `axi_ar_addr(3..2)` to specify a particular slave register.
  - Data is written (from processor to our peripheral) on a Slave Register specified by `axi_aw_addr(3..2)` when `slv_reg_wren = 1`. Also, data is read from a Slave register specified by `axi_ar_addr(3..2)` when `slv_reg_rden = 1`.

![Diagram of AXI4-Lite Interface](image)

- **Address (S_AXI_AWADDR, S_AXI_ARADDR):** In this example, we selected only two registers, but Vivado 2015.3 creates a template with a minimum of four 32-bit registers. So, we have 16 bytes, hence the 4 bit addresses, from which we only use the 2 MSBs to identify the Slave Registers: Register 0 is given the 00 code, and Register 1 the 01 code.
AXI4-LITE: PIPELINED DIVIDER

- Simple interface with 3 Slave Registers for reading and writing:
  - Slave Register 0: Master Writes data on the Slave Peripheral. When this happens, \( axi\_awaddr \) \( (3..2) = 00 \).
  - Slave Register 1: Master Reads Data from the Slave Peripheral. \( axi\_araddr \) \( (3..2) = 01 \).
  - Slave Register 2: Master Reads Data from the Slave Peripheral. \( axi\_araddr \) \( (3..2) = 10 \)

- Note that for Slave Registers 1 and 2, we do not need a physical register for both so-called Slave Registers. A multiplexor suffices in this case.

- **Important**: The pipelined divider captures input data when \( E = 1 \). We use the signal \( slv\_reg\_wren \) to determine whether data is present on Slave Register 0. However, data is present on Slave Register 0 on the cycle after \( slv\_reg\_wren = 1 \). That is why \( E \) is asserted on the next state (S2). This is an important consideration when designing more complex systems.

- **Software application**: The software routine writes a 32-bit word (A and B) and the divider starts processing. The software routine must write another 32-bit word (a dummy) to restart the process. An improvement would be to let the software online write 32-bit words of actual data (A and B), while a more complex FSM would take care of asserting E and then de-asserting E (when the processor requests reading via \( slv\_reg\_rden \)). When using more Slave Registers we need to consider \( axi\_awaddr \) and \( axi\_araddr \) to identify the registers to/from we write/read.
AXI4-FULL: MEMORY (XILINX® EXAMPLE)

- Data Width: 32 bits.
- Address (S_AXI_AWADDR, S_AXI_ARADDR): These signals are different from the latched addresses axi_awaddr, axi_araddr. Vivado 2015.3 creates a memory with 64 bytes (by default), hence the 6 bit addresses. The memory has 16 32-bit words.
- In order to point to a 32-bit word, we just use the four MSBs of S_AXI_AWADDR, S_AXI_ARADDR.
- In the figure below, the circuitry generates the following signals:
  - axi_awaddr, axi_araddr: On the Write Address/Read Address cycle, these addresses capture the value of S_AXI_AWADDR, S_AXI_ARADDR. Burst Transfers: these addresses are incremented by the interface following the burst rules set in S_AXI_AWBURST, S_AXI_ARBURST (FIXED, INCR, WRAP).
  - mem_wren: It indicates that new data is available on S_AXI_WDATA.
  - mem_rden: It indicates that we are ready to read data from the Memory. \( \text{mem\_wren} = \text{axi\_arv\_arr\_flag} \).

- Reading bursts (according to timing diagram obtained by simulating Vivado template), this particular circuit can only output one word every two cycles.
- Burst: This is configured by: i) S_AXI_AWSIZE and S_AXI_ARSIZE (Data width per burst), ii) S_AXI_AWBURST and S_AXI_ARBURST (Burst type), and iii) S_AXI_AWLEN and S_AXI_ARLEN (transfer per bursts).
AXI4-FULL: MEMORY WITH PIXEL PROCESSOR

- We use the same memory as before, but we add a pixel processor unit of 32 bits (four LUT 8-to-8). Due to the LUT delay most incoming signals to the Write Address and Write Channel (as well as some internal signals) are delayed using a register.
AXI4-FULL: PIXEL PROCESSOR WITH FIFO INTERFACE

- This design illustrates how to integrate a hardware architecture into the AXI Interface. We use the Pixel Processor as our first example, even though it does not require this complex interfacing.

- **Components:**
  - Input FIFO (iFIFO), Output FIFO (oFIFO). The FIFOs are asynchronous.
  - FSM @ S_AXI_ACLK, FSM @ AXI_CLKFX.

- **Considerations:**
  - AXI_RVALID: Compared to the Xilinx®-provided template, we modify the generation of S_AXI_RVALID (and S_AXI_RRESP). Now AXI_RVALID is asserted when axi_arv_arr_flag = 1 and when oFIFO is not empty (oempty = 0).
  - In this design, the memory address is ignored. That is, any 6-bit address will allow for writing and reading from the FIFOs. You can further customize your peripheral by performing address decoding so that only certain 6-bit addresses allow access to the FIFOs. This way you can use the other addresses for control purposes.
  - Notice that there is no control to tell the AXI interface that the iFIFO is full: the AXI Slave will respond as if data was actually written. So, the user software needs to keep track of how much data is being written to iFIFO.
  - When reading, if the oFIFO is empty, the AXI read request will be denied and it might lead to software deadlock. A more sophisticated design might be required here. So, the user software needs to keep track of how much data is present on oFIFO at all times.
  - Asynchronous FIFO: This circuit allows us to partition the peripheral into two different clock regions: one controlled by S_AXI_ACLK and the other controlled by CLKFX. Asynchronous FIFOs usually require a dual-port RAM memory (to write and read at the same time for different addresses) and extra logic to generate the 'empty' and 'full' signals.
  - Dynamic Frequency Control: MMCM (Multi mode Clock Managers) on the Zynq-7000 devices include a dynamic reconfiguration port (DRP). This port is a register-based interface that can adjust the frequency and phase at run-time without loading a new bitstream on the SoC. This circuitry can be connected to an AXI4-Lite peripheral in order to modify CLKFX. If we want to avoid this level of complexity, we can just do CLKFX = S_AXI_ACLK.

- **Input/Output Example:** If we input one 32-bit word, we get one 32-bit output word.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xDEADBEEF</td>
<td>0xEED2DDF7</td>
</tr>
<tr>
<td>0xBBBBBEAD</td>
<td>0xDDDDEED2</td>
</tr>
<tr>
<td>0xFADFBEAD</td>
<td>0xFDEEDDD2</td>
</tr>
<tr>
<td>0xCAFEDFBE</td>
<td>0xE3FFDDEF</td>
</tr>
</tbody>
</table>
• **FSM @ S_AXI_ACLK**
  - This FSM does not need to change if we modify the Pixel Processor by another circuit.
  - This FSM controls the outer side of the FIFOs and some AXI signals.
  - FIFOs have to be reset prior to usage for at least 5 read/write clock cycles. If we use 16 cycles @ 100 MHz, the minimum clkfx is 16x10ns/5 = 32 ns → 31.25 MHz. For now, we are making S_AXI_ACLK = CLK_FX.
  - \texttt{fifo fsm rst}: The register is to avoid glitches (this is to avoid simulation problems as FIFO reset has to glitch-free).
  - When reading: the FSM (@S_AXI_ACLK) requires that \texttt{oempty = 0} (oFIFO not empty) and that \texttt{S AXI_RVALID = 1} before it issues \texttt{orden = 1} (load next data on the output of oFIFO).

• **FSM @ CLKFX:**
  - This FSM needs to change if we modify the Pixel Processor by another circuit. Most circuits include a 'start' and 'done' signals (or ‘enable’ and ‘valid’) to be controlled by this FSM. This way, our only job is to implement an interface to the FIFOs to load or write the required input or output data.
  - This FSM handles:
    - The inner side of the FIFOs. For iFIFO, this is \texttt{iempt}, \texttt{irden}; for OFIFO, this is: \texttt{ofull}, \texttt{owren}. For the Pixel Processor, The FSM checks whether IFIFO is not empty and oFIFO is not full. If so, we push out the next iFIFO word (irden = 1) and we write a word on oFIFO (owren = 1).
    - Control signals to the Pixel Processor (e.g.: start, done, enable, valid signals; they do not exist in this example)
    - Control signals to the interface between the FIFOs and the Pixel Processor input/output data signals. We might require extra glue logic between the output of IFIFO and the Pixel Processor input, and between the Pixel Processor output and the input of oFIFO. In this case, this is not required, as there are direct connections.

• reset signal of the FSM @ CLKFX: We connect it to the AXI bus reset.

**Template:** You can use this interface as a template to integrate any hardware architecture into an AXI4-Full peripheral. The only part that needs to change is the circuitry running at CLKFX: the hardware architecture and the FSM @ CLKFX. Unlike the Pixel Processor, we also usually require glue logic between the hardware architecture and iFIFO output and oFIFO input. The next example shows such a case.
AXI4-FULL: 2D DCT WITH FIFO INTERFACE

- This design illustrates how to integrate a complex system (2D DCT) into the AXI interface. The 2D-DCT includes the input and output data signals, and the control signals (reset, enable, and valid).
- 2D DCT input: NxN pixels, each pixel of B bits. Data is input column-wise.
- 2D DCT output: NxN pixels, each pixel of 'NO' bits. Data is output row-wise.
- Parameters: N (transform size: 4, 8, 16), B (input pixel bitwidth: 8, 16), NO (output pixel bitwidth: 8, 16).
- As previously mentioned, what changes with respect to the previous system is what is running at CLKFX: the 2D-DCT architecture, the FSM @ CLKFX, and the glue logic between the 2D-DCT and the FIFOs.

- Glue logic between 2D-DCT and FIFOs:
  - Input Interface: This is just a bunch of registers that capture data 32 bits at a time. The 2D DCT data input is more than 32 bits (usually B=8, N=4, 8, 16): we input N groups of NxB bits.
  - Output Buffer: The 2D DCT outputs N groups of NxNO bits in successive cycles. As we cannot place this amount of data fast enough on FIFO, we need a temporal buffer to store this data.
  - Output Interface: This is a multiplexer that outputs data 32 bits at a time. The 2D DCT data output is more than 32 bits (usually NO=16, N=4, 8, 16): we output N groups of NxNO bits.

- reset signal of the 2D DCT IP and FSM @ CLKFX: Though we can connect it to the AXI bus reset (S_AXI_ARESETN), we prefer to connect them to the FIFOs' reset; this active-high signal is generated by the FSM@ S_AXI_ACLK. This configuration is more helpful if we want to later perform Partial Reconfiguration.
- **Glue Logic examples**: The figure depicts different input/output interfaces to the 2D DCT IP core along with the Output buffer. They depend on the parameter N, B, and NO. Note that when DCT=4x4 and B=NO=8, there is no need for the extra buffer or for any glue logic. In all the other cases, we do need an output buffer as the oFIFO is only 32-bits wide. Note the signals that the FSM @ CLKFX needs to control: ‘s’, ‘E_buf’, ‘Eri’.

- **Input/Output Example (N=4, B=8, NO=16)**: The outputs have been verified (with a MATLAB model) to be correct. For the inputs, each 32 bit word is a column (top to bottom). For the output, each two 32-bit words is a row (left to right).

<table>
<thead>
<tr>
<th>Input (columns)</th>
<th>Output (rows)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xDEADBEEF</td>
<td>0x8000E92E</td>
</tr>
<tr>
<td>0xBE8EDDEAD</td>
<td>0x14C00D82</td>
</tr>
<tr>
<td>0xFADEBEAD</td>
<td>0x18A6E418</td>
</tr>
<tr>
<td>0xC6EBEBEDF</td>
<td>0xDB3E1FB2</td>
</tr>
<tr>
<td>0xA401E19</td>
<td>0xF8382A32</td>
</tr>
<tr>
<td>0xD40236D</td>
<td>0xDEC9FDE7</td>
</tr>
<tr>
<td>0xCAFEBEDF</td>
<td>0x80000CF4</td>
</tr>
<tr>
<td>0xCAFEBEDF</td>
<td>0xFF0003D5</td>
</tr>
<tr>
<td>0xC6C3C7C3</td>
<td>0x0471045F</td>
</tr>
<tr>
<td>0xBE8DC2BD</td>
<td>0xFF89FF65</td>
</tr>
<tr>
<td>0x1D40236D</td>
<td>0x010003CE</td>
</tr>
<tr>
<td>0xF8382A32</td>
<td>0x0000000B</td>
</tr>
<tr>
<td>0xDEC9FDE7</td>
<td>0x06D0FFE5</td>
</tr>
<tr>
<td>0x00310020</td>
<td></td>
</tr>
</tbody>
</table>

- **Template**: You can use this interface as a template to integrate any hardware architecture into an AXI4-Full peripheral. The only part that needs to change is the circuitry running at CLKFX: the hardware architecture, the FSM @ CLKFX, and the glue logic between the FIFOs and the DCT 2D.