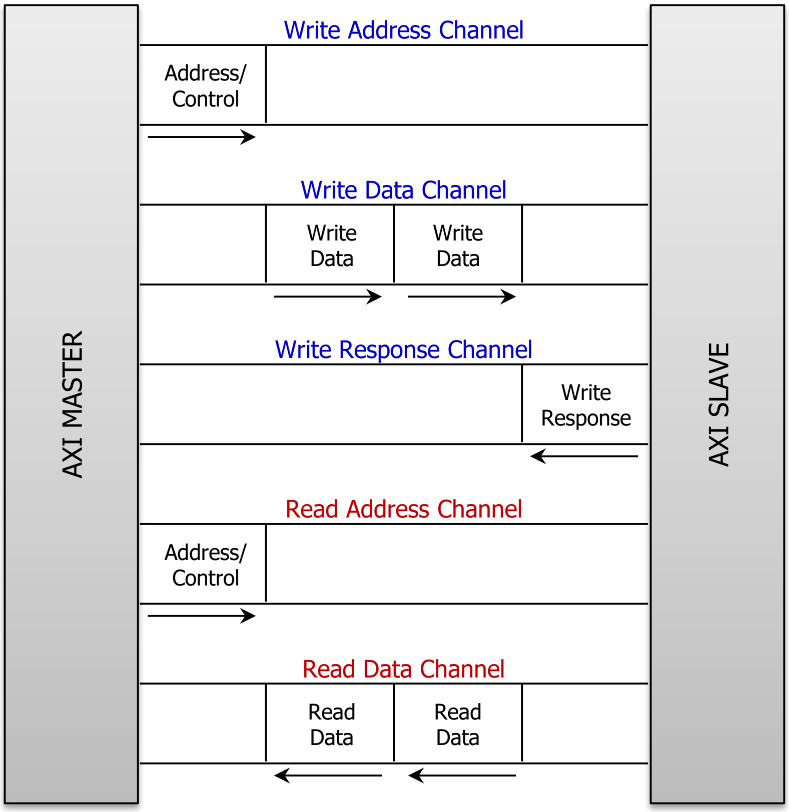
# Unit 5 – Embedded Systems in SoC

## AXI Bus

**References**:
- Zynq$^{TM}$ 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



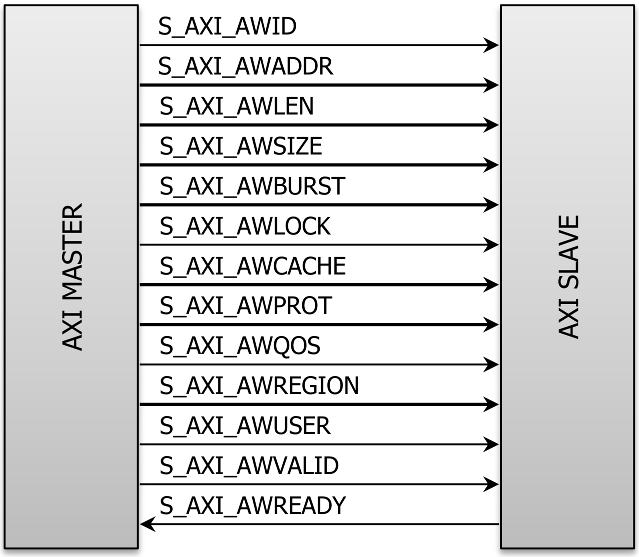
- The AXI protocol is burst-based and defines five independent transaction channels.
- Write Channel Architecture: Address and Control data is transmitted to the slave before a burst of data is transmitted, and a Write Response signaled following completion:
  - ✓ Write Address Channel
  - ✓ Write Data 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.)

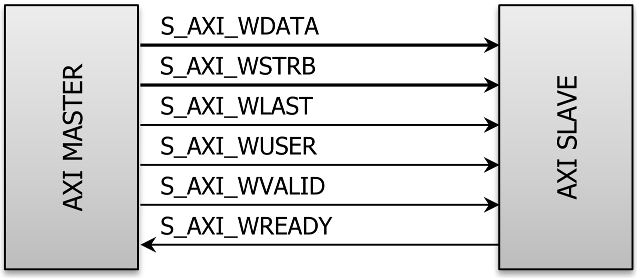
- **Signals:**
  Global System Signals:
  ✓ S_AXI_CLK: AXI4 clock
  ✓ S_AXI_ARESETN: AXI4 active-low reset.

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

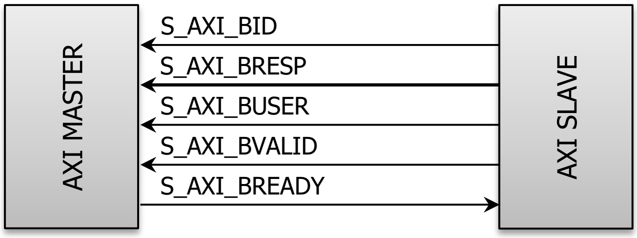
**WRITE ADDRESS CHANNEL**



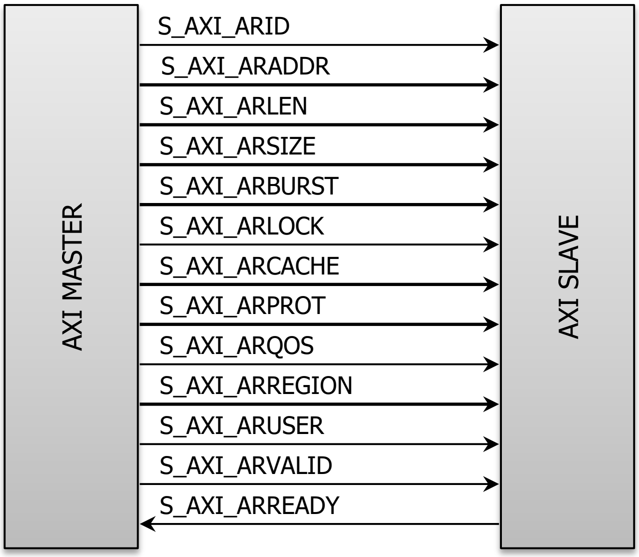
**WRITE DATA CHANNEL**



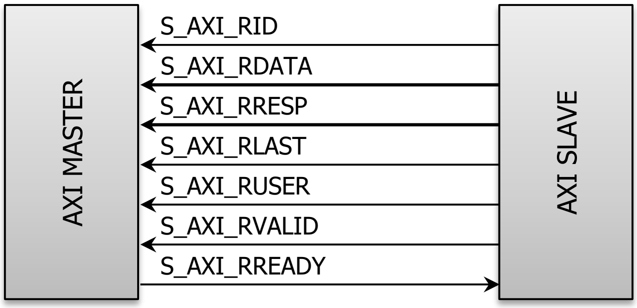
**WRITE RESPONSE CHANNEL**



**READ ADDRESS CHANNEL**

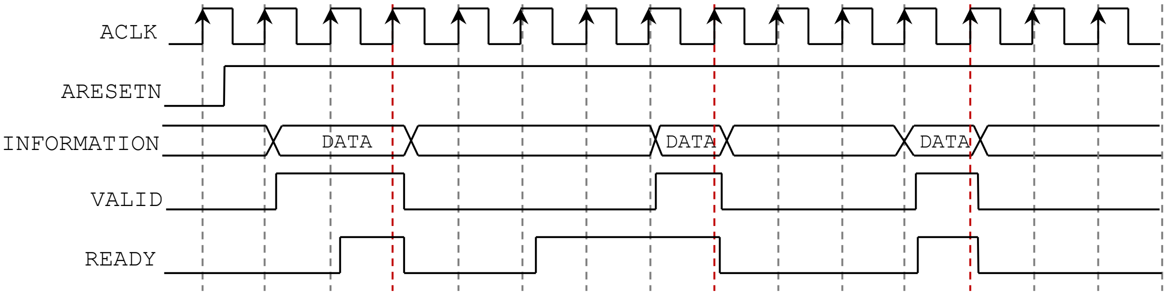


**READ DATA CHANNEL**



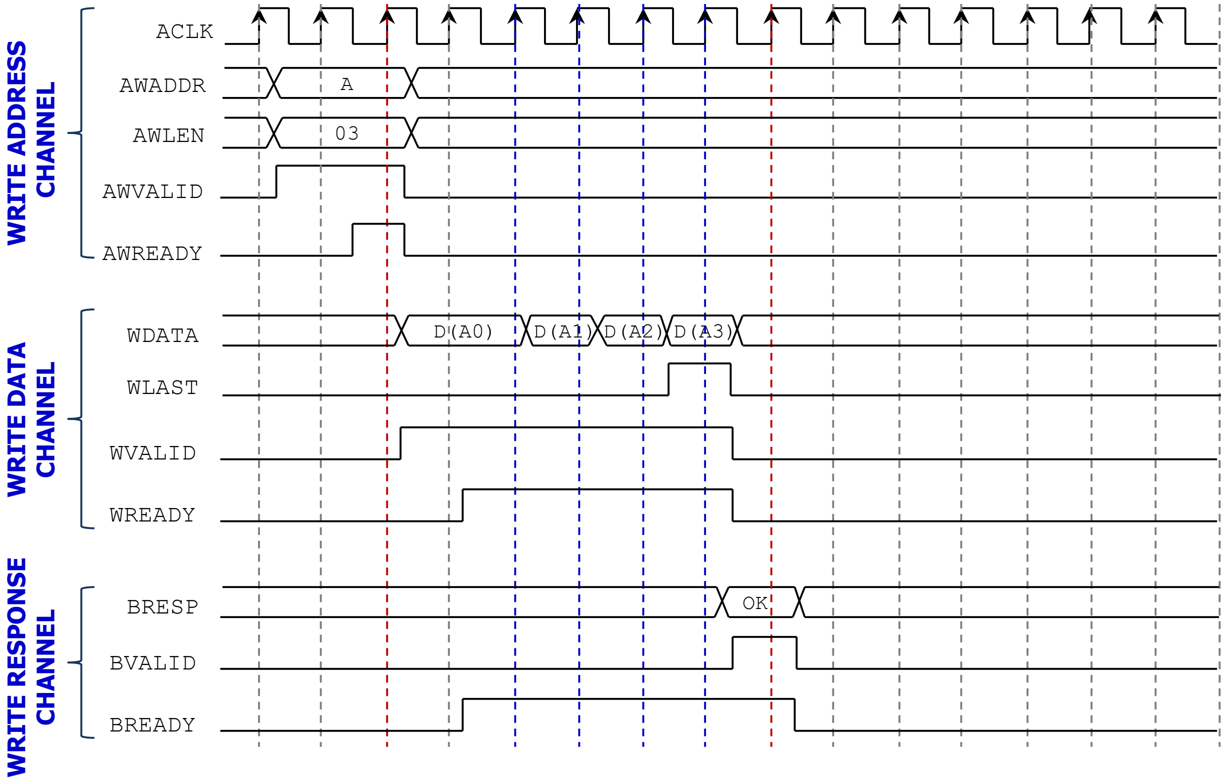
## AXI4-FULL PROTOCOL

- 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.



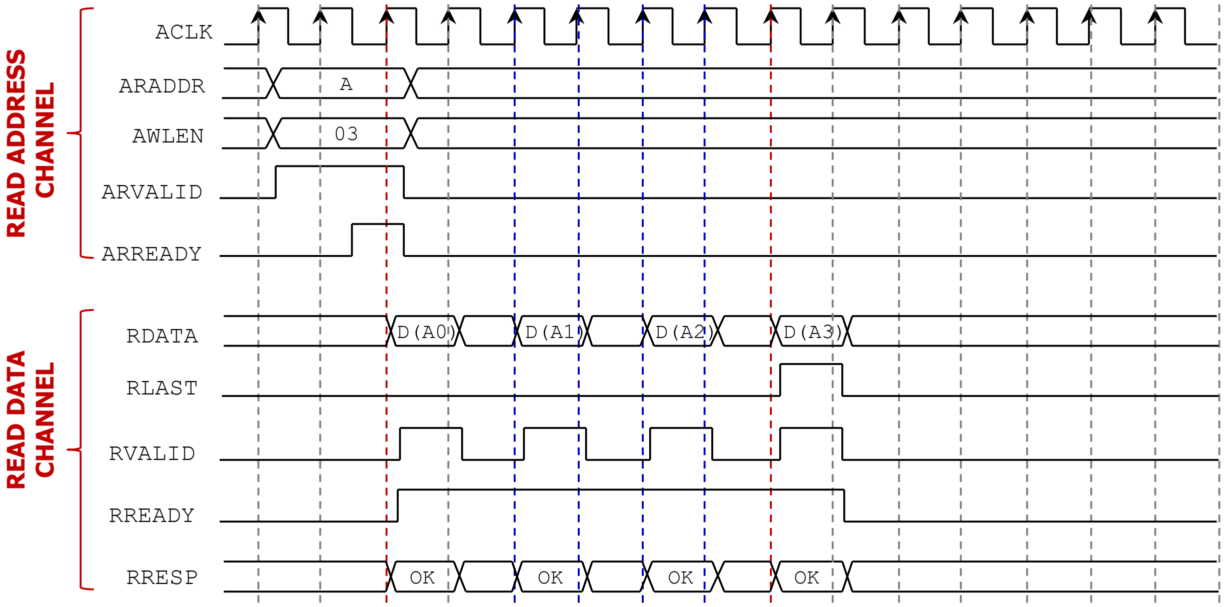
**Writing Transaction – Simple Memory:**

- 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 send 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 one 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:**

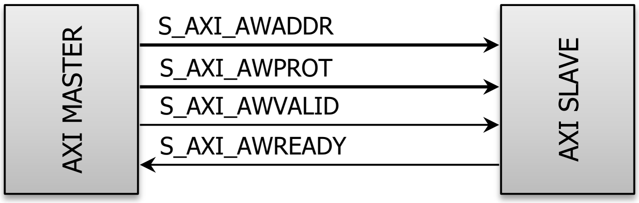
- The AXI master sends the read address (along with burst information) via the Read Address Channel. Then, the Slave sends Read Data Back via the Read Data Channel.
- *Read Address Channel Handshake*: The AXI Master asserts ARVALID only when it drives valid address and control information. It remains 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 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).



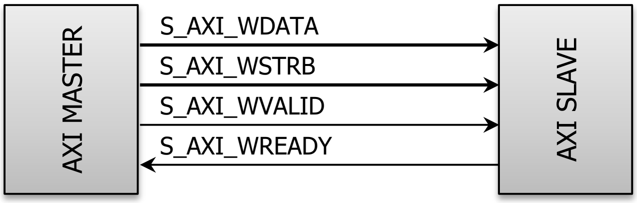
## AXI4-Lite Interface

- 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.

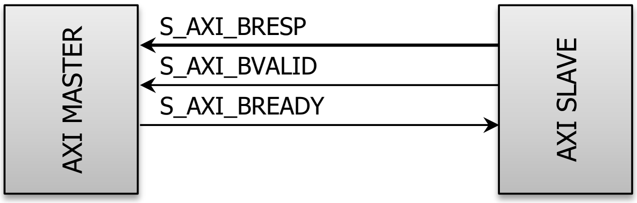
**WRITE ADDRESS CHANNEL**



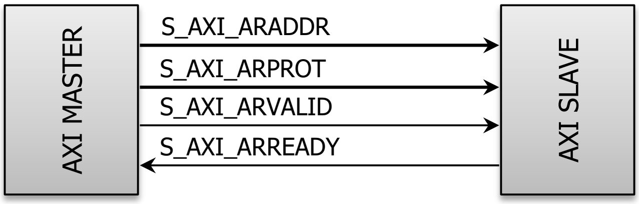
**WRITE DATA CHANNEL**



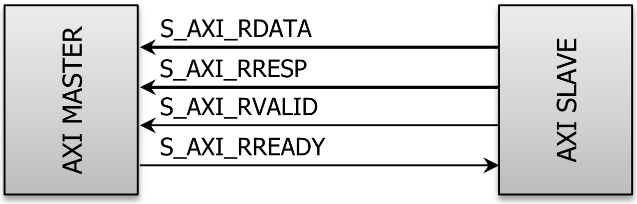
**WRITE RESPONSE CHANNEL**



**READ ADDRESS CHANNEL**



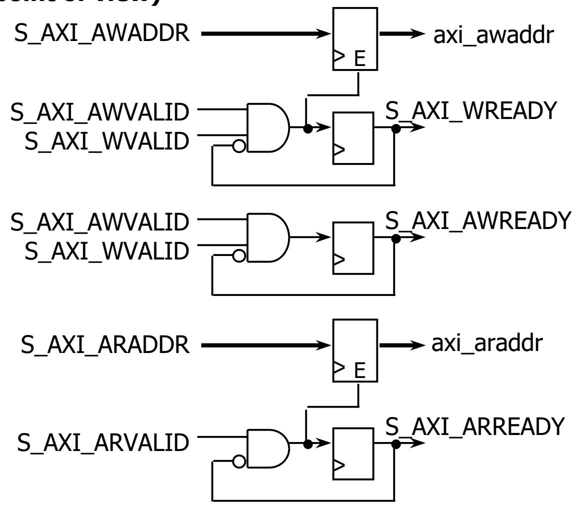
**READ DATA CHANNEL**

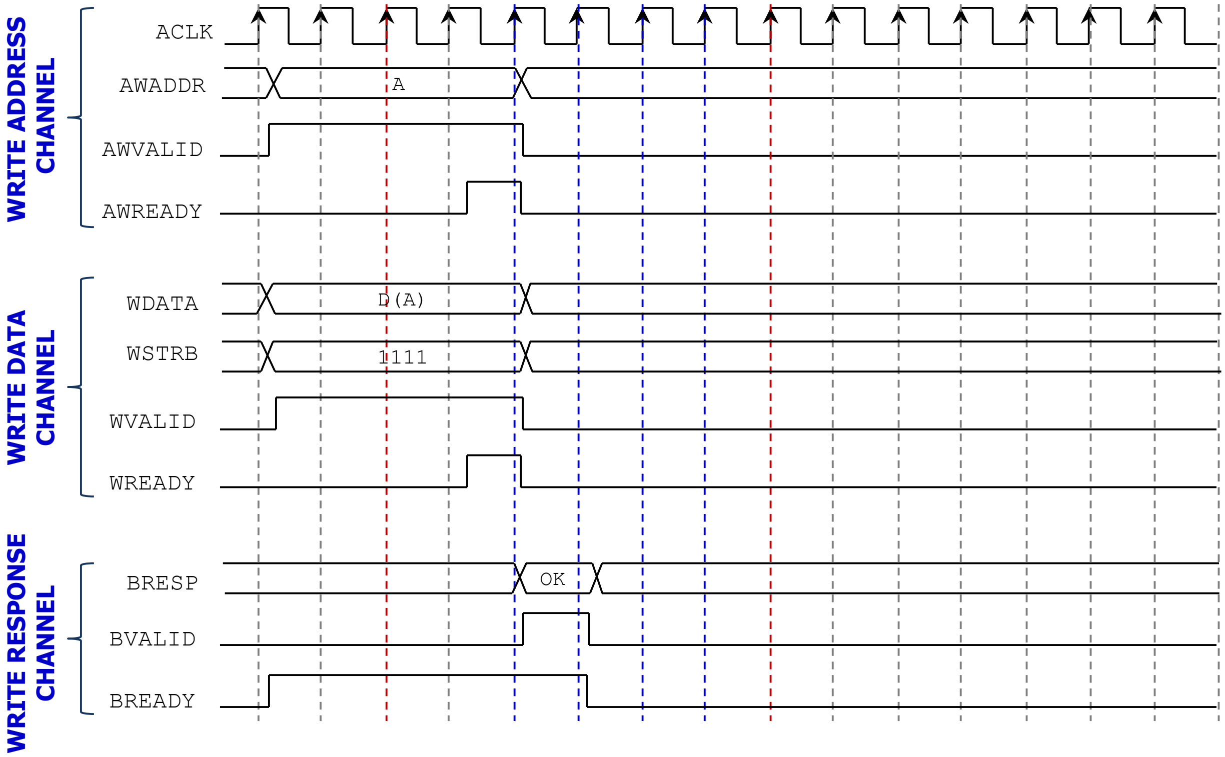


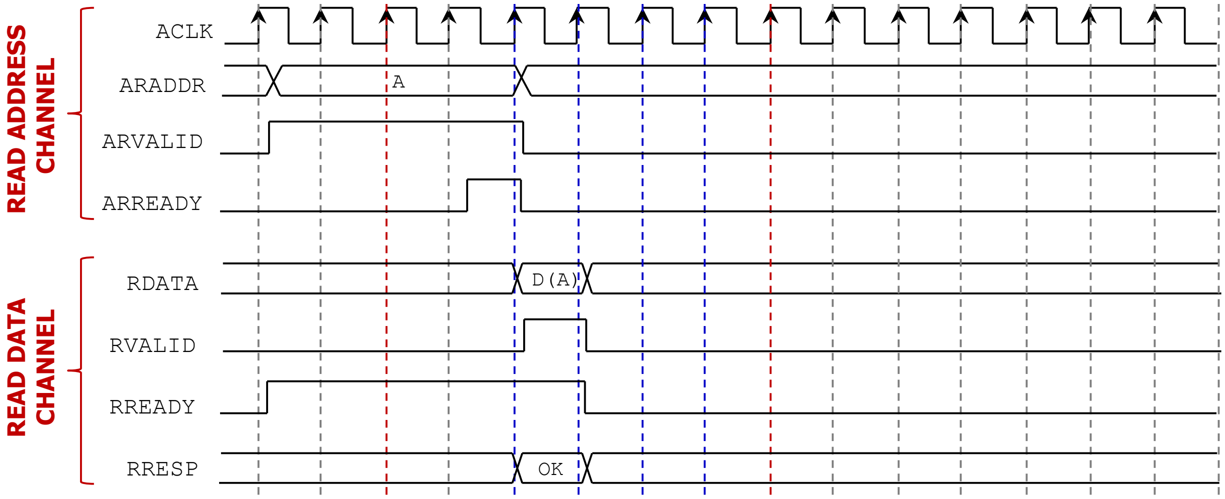
## AXI4-LITE PROTOCOL

- 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_WVALID='1', S_AXI_AWREADY='0'.
  - ✓ S_AXI_ARREADY: It is asserted for one clock cycle when S_AXI_RVALID 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.





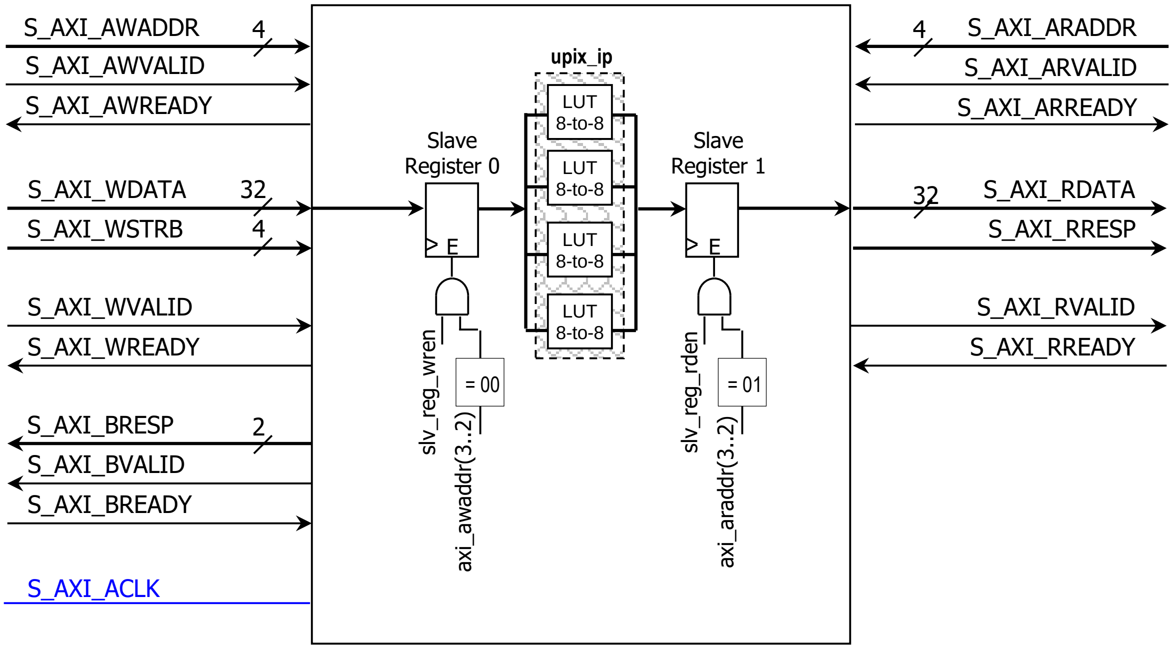


# AXI4 INTERFACE - EXAMPLES

- AXI4-Lite Interface (Slave): Vivado 2016.2 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 2016.2 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.
- The source files of the examples provided here can be downloaded at: Tutorial: Embedded System Design for Zynq SoC.

## 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.
  $slv\_reg\_wren = S\_AXI\_WREADY\ and\ S\_AXI\_VALID\ and\ S\_AXI\_AWREADY\ and\ S\_AXI\_AWVALID$. This signal is pulse with a duration of one clock cycle.
- $slv\_reg\_rden$: It indicates that the Master (e.g.: the processor) is requesting to read from a Slave Register.
  $slv\_reg\_rden = S\_AXI\_ARREADY\ and\ S\_AXI\_ARVALID\ and\ (not\ S\_AXI\_RVALID)$.
- $axi\_aw\_addr$: 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 $slv\_reg\_rden = 1$.



✓ Address ($S\_AXI\_AWADDR, S\_AXI\_ARADDR$): In this example, we selected only two registers, but Vivado 2016.2 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.

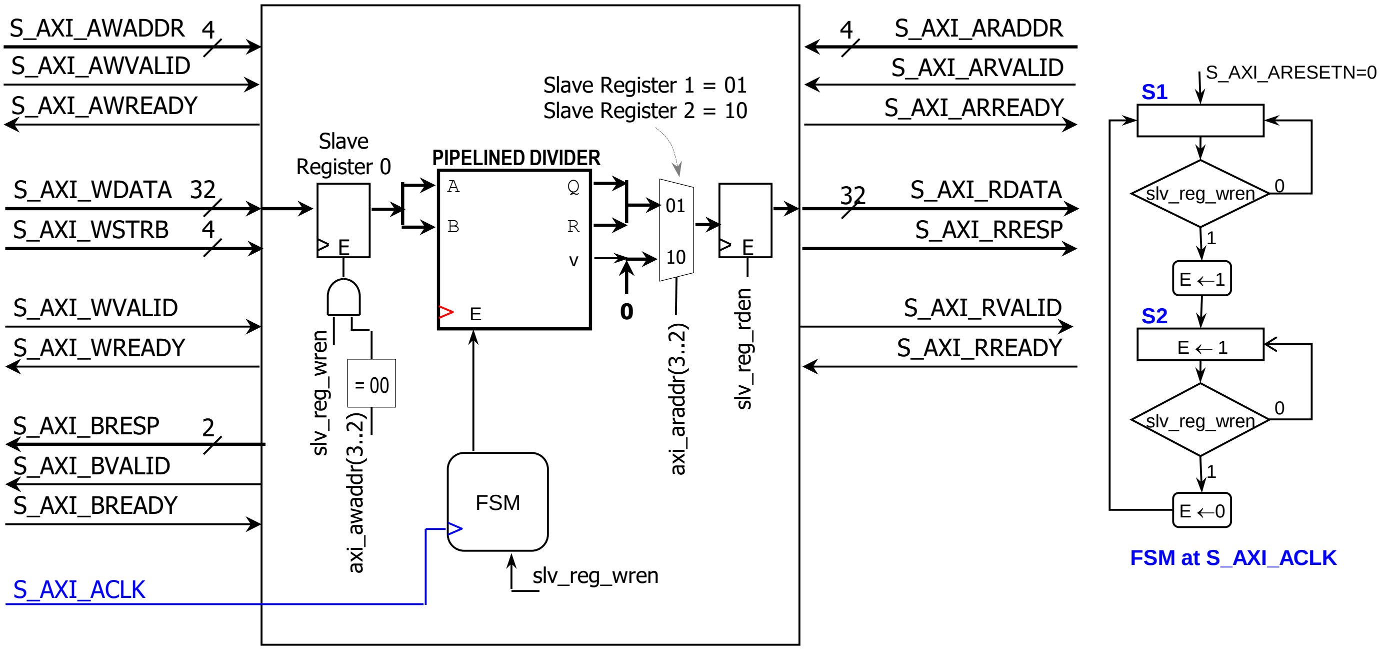
## AXI4-LITE: PIPELINED DIVIDER

- 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. $axi\_araddr\ (3..2) = 10$

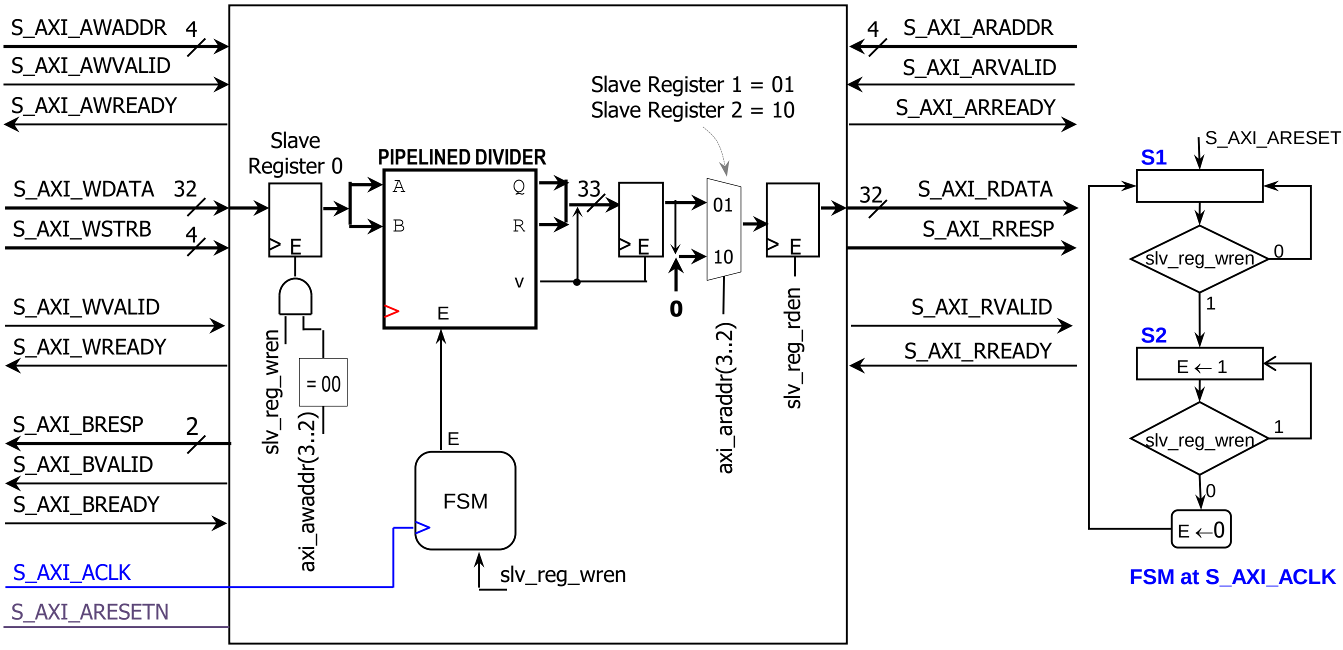
- Note that for Slave Registers 1 and 2, we do not need a physical register for both so-called Slave Registers. A multiplexor suffices in this case.
- When using more Slave Registers we need to consider $axi\_awaddr$ and $axi\_araddr$ to identify the registers to/from we write/read.

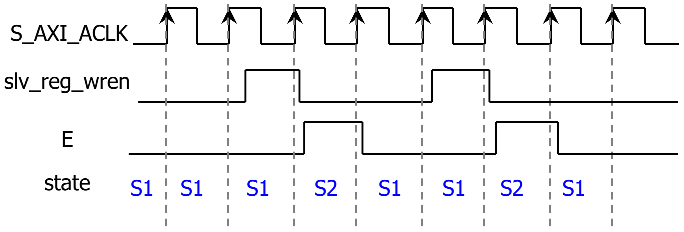
- **Pipelined Divider operation:**
  - ✓ The pipelined divider captures input data $(A, B)$ when $E = 1$. After a processing delay, output data $(Q, R)$ appears and it is signaled by $v = 1$.
  - ✓ Output data is valid only when $v = 1$. $v$ is a delayed version of $E$: if $E$ was only asserted for one cycle, then when the division operation completes, $v$ will only be asserted for one cycle.
  - ✓ 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 one cycle after $slv\_reg\_wren = 1$. This is an important consideration when designing these systems.

- **Interface - First Version:** This works by asserting $E$ as soon as there is data on the bus ($slv\_reg\_wren = 1$). Note that the first time $E = 1$, the pipelined divider does not capture the proper data. But on the next cycles, $E = 1$ and proper data is captured by the pipelined divider many times in a row. This means that the pipelined divider produces the same results every clock cycle after the first $v = 1$. Thus, we can capture output results at any time. We can stop this (set $E = 0$) by writing another word (that is not captured, see FSM). At this point, we can restart the process by writing a new word.
  - ✓ Software Routine: It writes a 32-bit word (A and B) and the divider starts processing. We wait until we detect $v = 1$ via software; at this point, we capture the data (this works because $E = 1$ this entire time). The software routine must write another 32-bit word (a dummy) to restart the process (to set $E = 0$ first).
  - ✓ This is a simple hardware. However, we require to write dummy words in software, making the hardware design look inconsistent.
  - ✓ Improvements:
    - ▫ Use $slv\_reg\_rden$ to set E=0. Here, we can set E=1 one cycle after $slv\_reg\_wren = 1$.
    - ▫ Assert $E$ only when data is available. Requires a buffer to capture output data.



- **Interface - Second Version (recommended):** This works by asserting $E$ only when required (one cycle after $slv\_reg\_wren = 1$). Here, $E$ is asserted for one clock cycle. Unlike the previous interface version, the pipelined divider only captures the proper data.
  - ✓ In this interface, $E$ is asserted one cycle after $slv\_reg\_wren$ is asserted. $E$ lasts one clock cycle, and so does $v$ and the valid output data. In order to capture data at this moment, we must use a buffer that retrieves data when $v = 1$ (this holds true even if we used a sequential divider that keeps the data until the next input word is written). With the data stored in the buffer, we can request a read from a Slave Register. After this, we are ready to write another word.
    * Note: This interface works assuming that a data arrive one per $slv\_reg\_wren$ pulse, i.e. the interface does not support data continuously arriving at every clock cycle (which in effect, AXI4-Lite does not support).
  - ✓ Note that instead of the buffer registers, we can have a FIFO. This would allow us to write a chunk of data first, and then retrieve a chunk of output results. We would need to detect the last $v = 1$ (which occurs after we write the last word) in order to retrieve the data.
  - ✓ Software Routine: It writes a 32-bit word (A and B) and the divider starts processing. We wait until we detect $v = 1$ (on the buffer) via software; at this point, we capture the data. Then, we are ready to write a new data word.
  - ✓ This software-hardware design looks more consistent with the way the pipelined divider works.





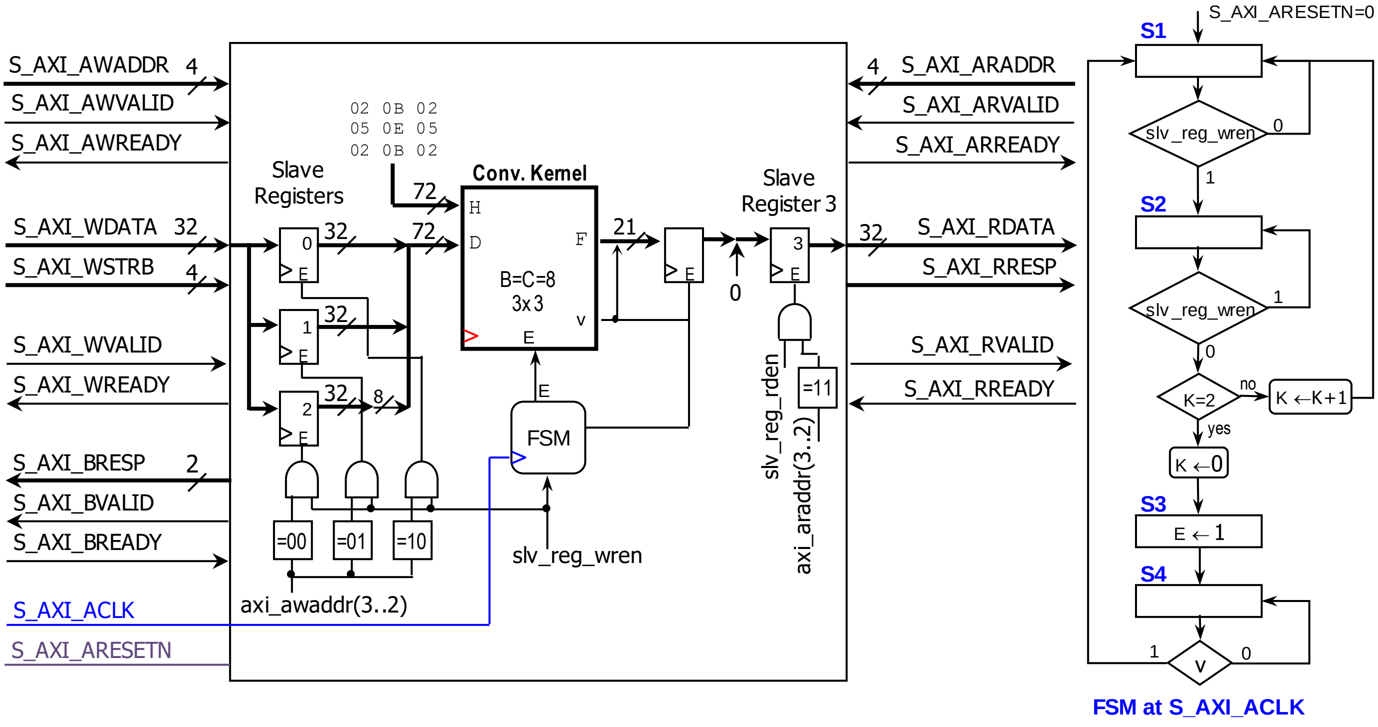
## AXI4-LITE: PIPELINED 2D CONVOLUTION KERNEL

- Simple interface with 4 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 Writes data on the Slave Peripheral. When this happens, $axi\_awaddr\ (3..2) = 01$.
  - ✓ Slave Register 2: Master Writes data on the Slave Peripheral. When this happens, $axi\_awaddr\ (3..2) = 10$.
  - ✓ Slave Register 3: Master Reads Data from the Slave Peripheral. $axi\_araddr\ (3..2) = 11$.

- When using more Slave Registers we need to consider $axi\_awaddr$ and $axi\_araddr$ to identify the registers to/from we write/read.

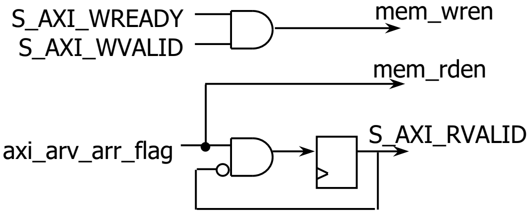
- **Pipelined 2D Convolution Kernel operation:**
  - ✓ We use the 2D Convolution Kernel where N=3, B=C=8.
  - ✓ This core captures $D$ when $E = 1$. After a processing delay, output data ($F$) appears and it is signaled by $v = 1$.
  - ✓ Output data is valid only when $v = 1$. $v$ is a delayed version of $E$: if $E$ was only asserted for one cycle, then when the division operation completes, $v$ will only be asserted for one cycle.
  - ✓ 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 one cycle after $slv\_reg\_wren = 1$. This is an important consideration when designing these systems.

- **Interface:**
  - ✓ $E$ is asserted only when three 32-bit words are present on the input D of the 2D Conv. Kernel. The FSM detects when 3 $slv\_reg\_wren$ pulses are present. Then it asserts $E$ for one clock cycle. Finally, the FSM waits until $v = 1$ to allow for new incoming data.
  - ✓ Software: It writes three 32-bit words, and then reads a 32-bit word. This output word includes the signal $v = 1$. If the signal $v$ is 1, the software routine can repeat the procedure (write 3 32-bit words, read a 32-bit word).

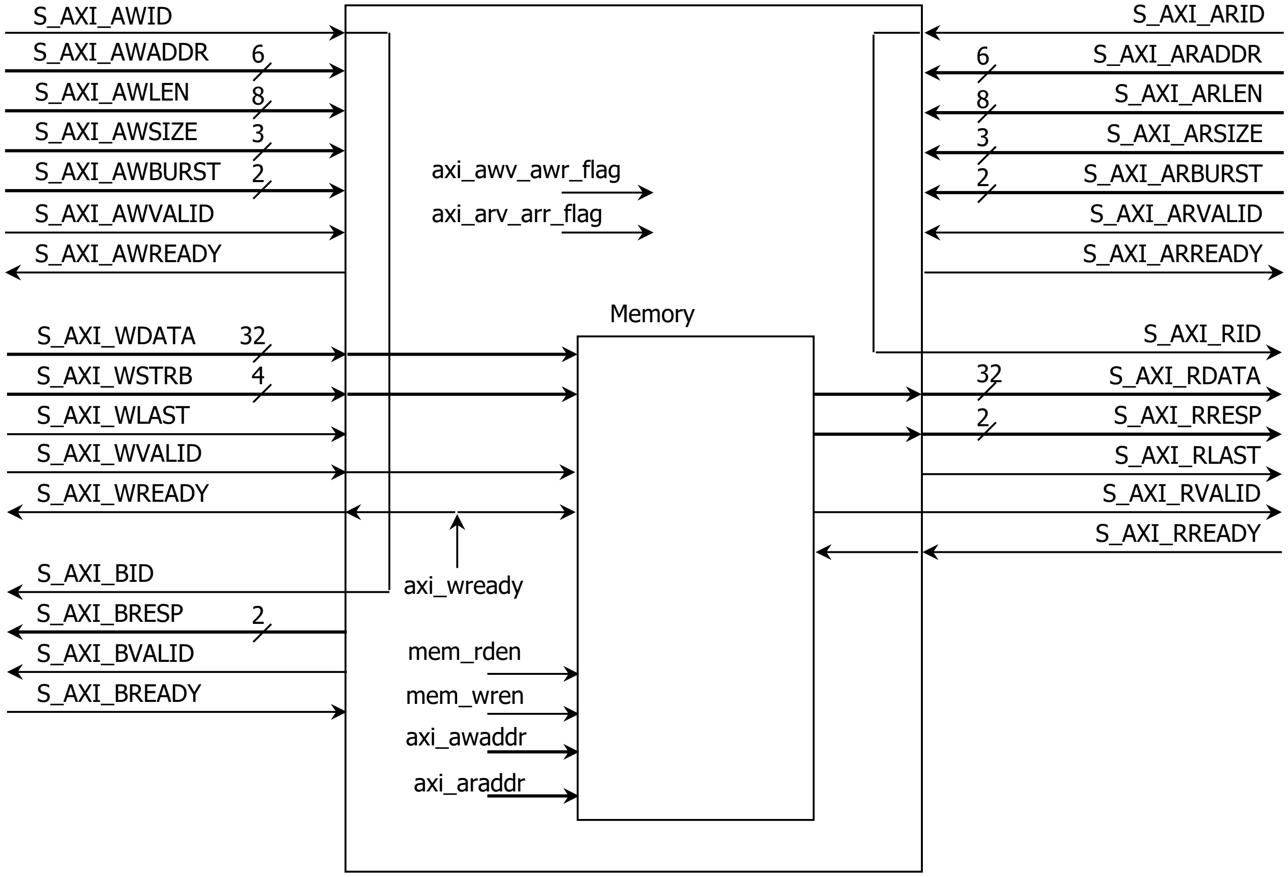


## AXI4-FULL: MEMORY (XILINX® EXAMPLE)

- 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:
  - ✓ $axi\_awv\_awr\_flag$: This registered signal marks the presence of a write address valid (i.e., we are ready to write). It is asserted when $S\_AXI\_AWVALID = 1, S\_AXI\_AWREADY = 0$ (and $axi\_arv\_arr\_flag = 0$). It is de-asserted when $S\_AXI\_WREADY = S\_AXI\_WLAST = 1$.
  - ✓ $axi\_arv\_arr\_flag$: This registered signal marks the presence of a read address valid (i.e., we are ready to read). It is asserted as soon as $S\_AXI\_ARVALID = 1, S\_AXI\_ARREADY = 0$ (and $axi\_awv\_awr\_flag = 0$). It is de-asserted when $S\_AXI\_RVALID = S\_AXI\_RREADY = S\_AXI\_RLAST = 1$.
  - ✓ $axi\_awaddr, axi\_araddr$: 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. $mem\_rden = axi\_arv\_arr\_flag$.

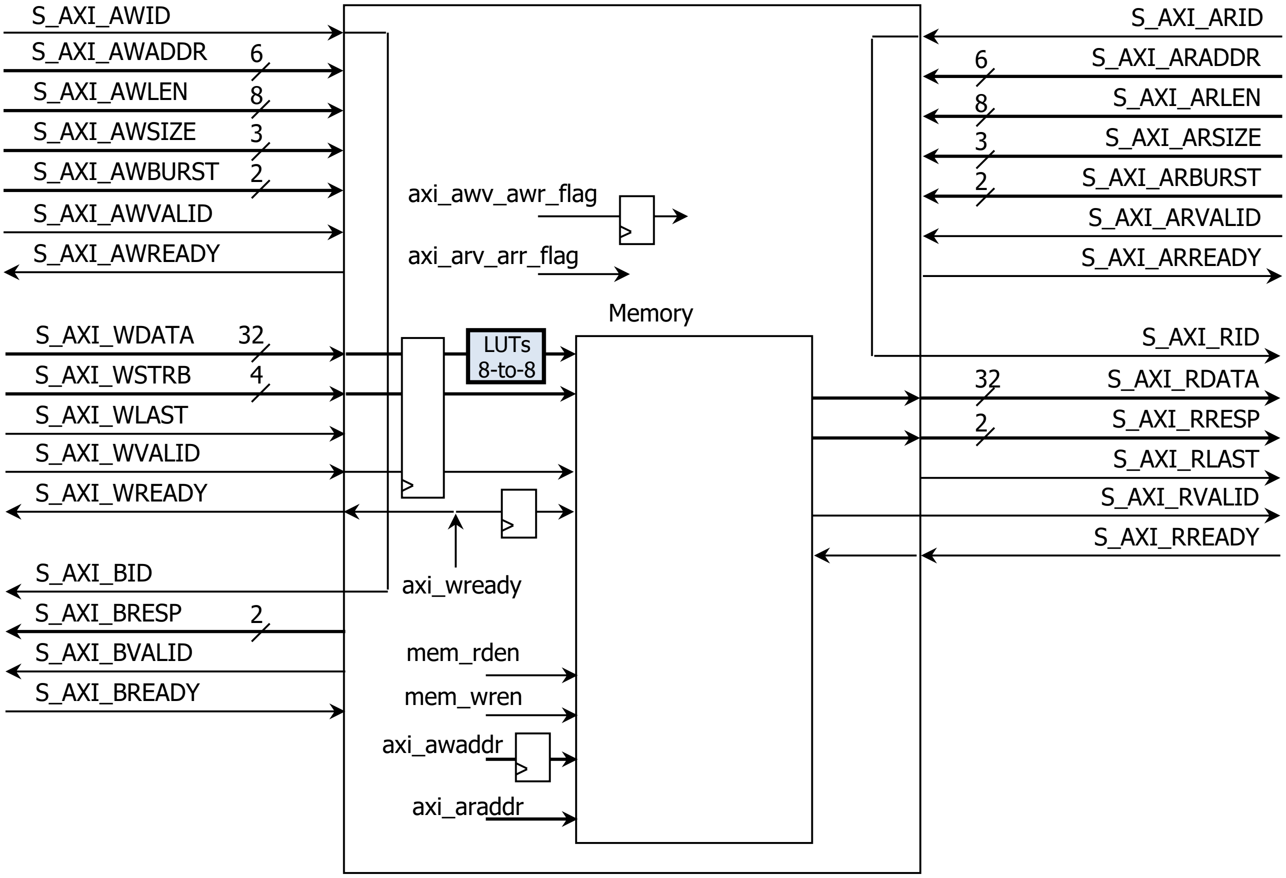


- Reading bursts (according to timing diagram obtained by simulating Vivado template), this particular circuit can only output one word every two cycles.
- **Burst:** This is configured by: i) $S\_AXI\_AWSIZE$ and $S\_AXI\_ARSIZE$ (Data width per burst), ii) $S\_AXI\_AWBURST$ and $S\_AXI\_ARBURST$ (Burst type), and iii) $S\_AXI\_AWLEN$ and $S\_AXI\_ARLEN$ (transfer per bursts).



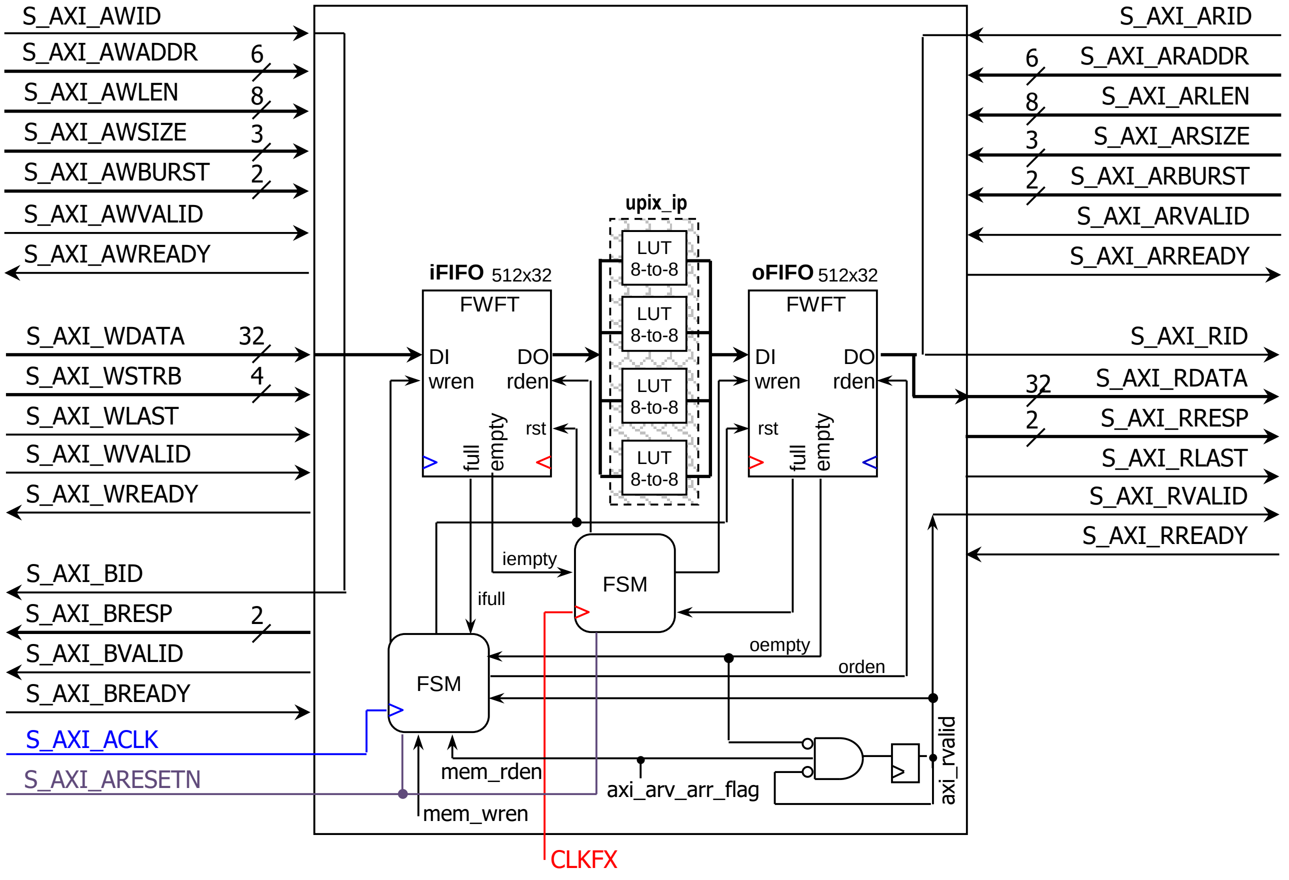
## AXI4-FULL: MEMORY WITH PIXEL PROCESSOR

- 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. Also, they are configured as First Word Fall Through (FWFT), this is by default the first written word always appears on the output.
  - ✓ 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.



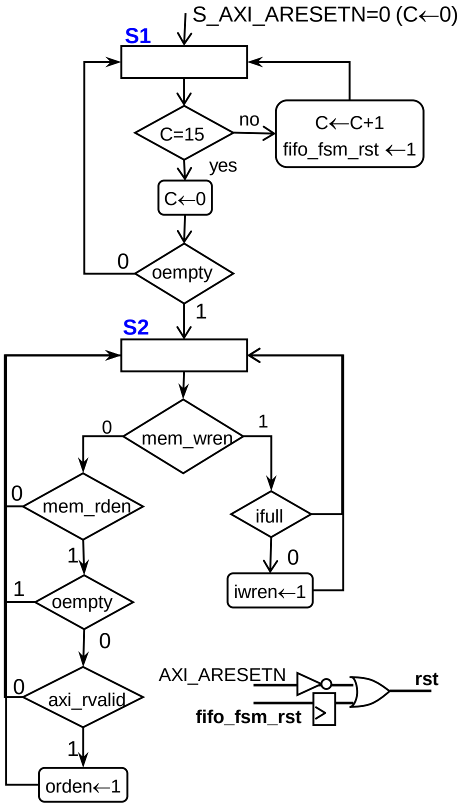
- **Input/Output Example:** If we input one 32-bit word, we get one 32-bit output word.

| Input | Output |
|---|---|
| 0xDEADBEEF | 0xEED2DDF7 |
| 0xBEBEDEAD | 0xDDDDEED2 |
| 0xFADEBEAD | 0xFDEEDDD2 |
| 0xCAFEBEDF | 0xE3FFDDEF |

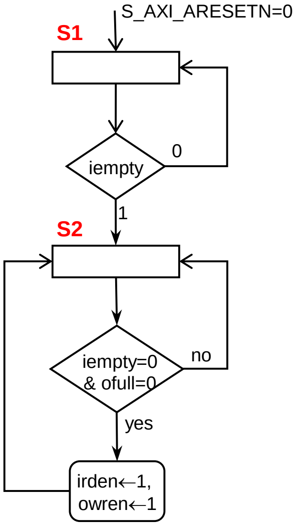
- **FSM @ S_AXI_ACLK**
  - ✓ This FSM does not need to change if we modify the Pixel Processor by another circuit.
  - ✓ This FSM controls the outer side of the FIFOs and some AXI signals.
  - ✓ FIFOs have to be reset prior to usage for at least 5 read/write clock cycles. If we use 16 cycles @ 100 MHz, the minimum clkfx is 16x10ns/5 = 32 ns → 31.25 MHz. For now, we are making S_AXI_ACLK = CLK_FX.
  - ✓ $fifo\_fsm\_rst$: 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 $oempty = 0$ (oFIFO not empty) and that $S\_AXI\_RVALID = 1$ before it issues $orden = 1$ (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 $iempty$, $irden$; for OFIFO, this is: $ofull$, $owren$. 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.



FSM at S_AXI_ACLK



FSM at CLKFX

- **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 FIFO INTERFACE

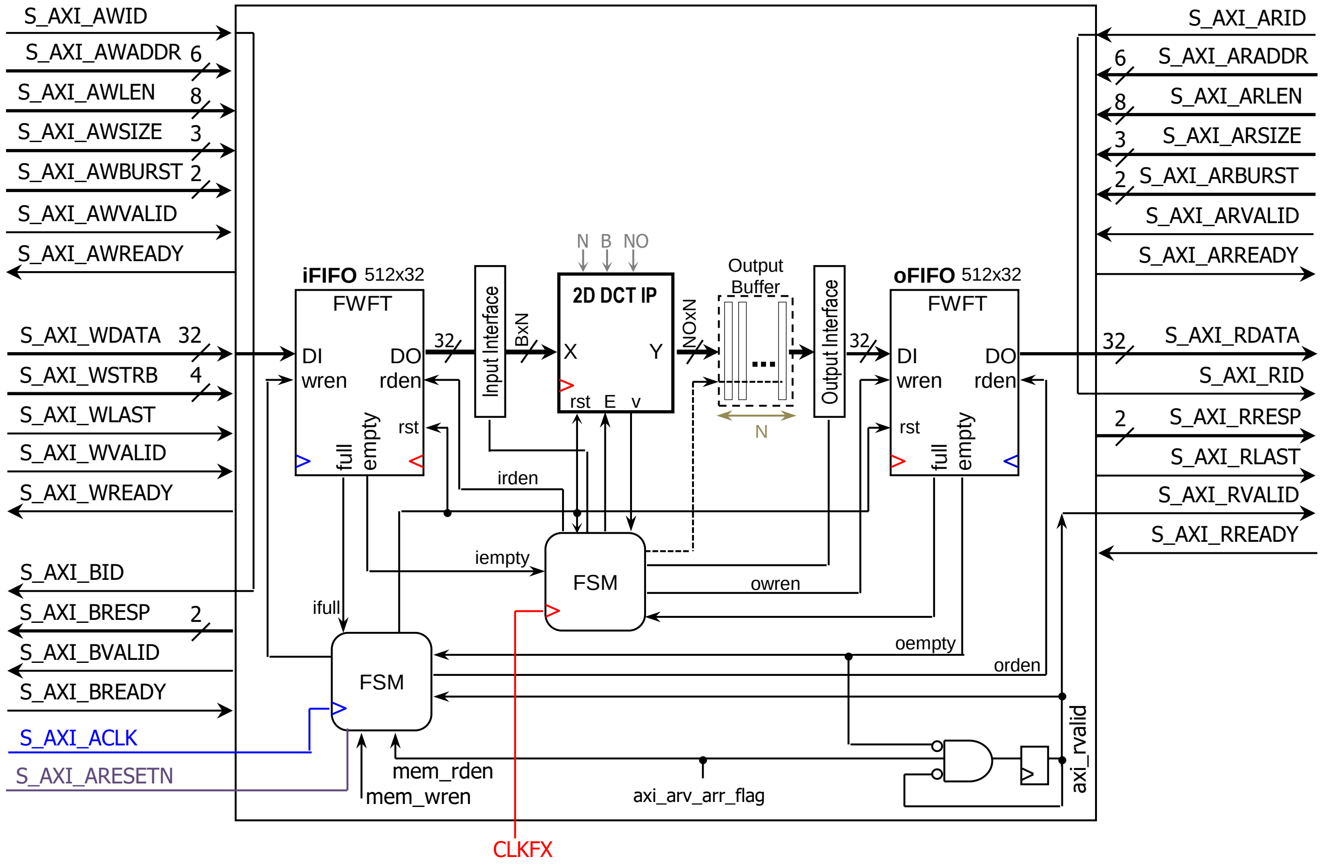
- This design illustrates how to integrate a complex system (2D DCT IP) into the AXI4-Full interface.

- **2D DCT IP**: The figure depicts the input and output data signals, and the control signals (reset, enable, and valid), and the three most important parameters (N, B, NO).
  - ✓ 2D DCT input: NxN pixels, each pixel of $B$ bits. Data is input column-wise. To feed a column, enable must be asserted.
  - ✓ 2D DCT output: NxN pixels, each pixel of $NO$ bits. Data is generated row-wise. When a row is ready, $v$ is asserted. The N rows are generated one cycle after another.
  - ✓ 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.

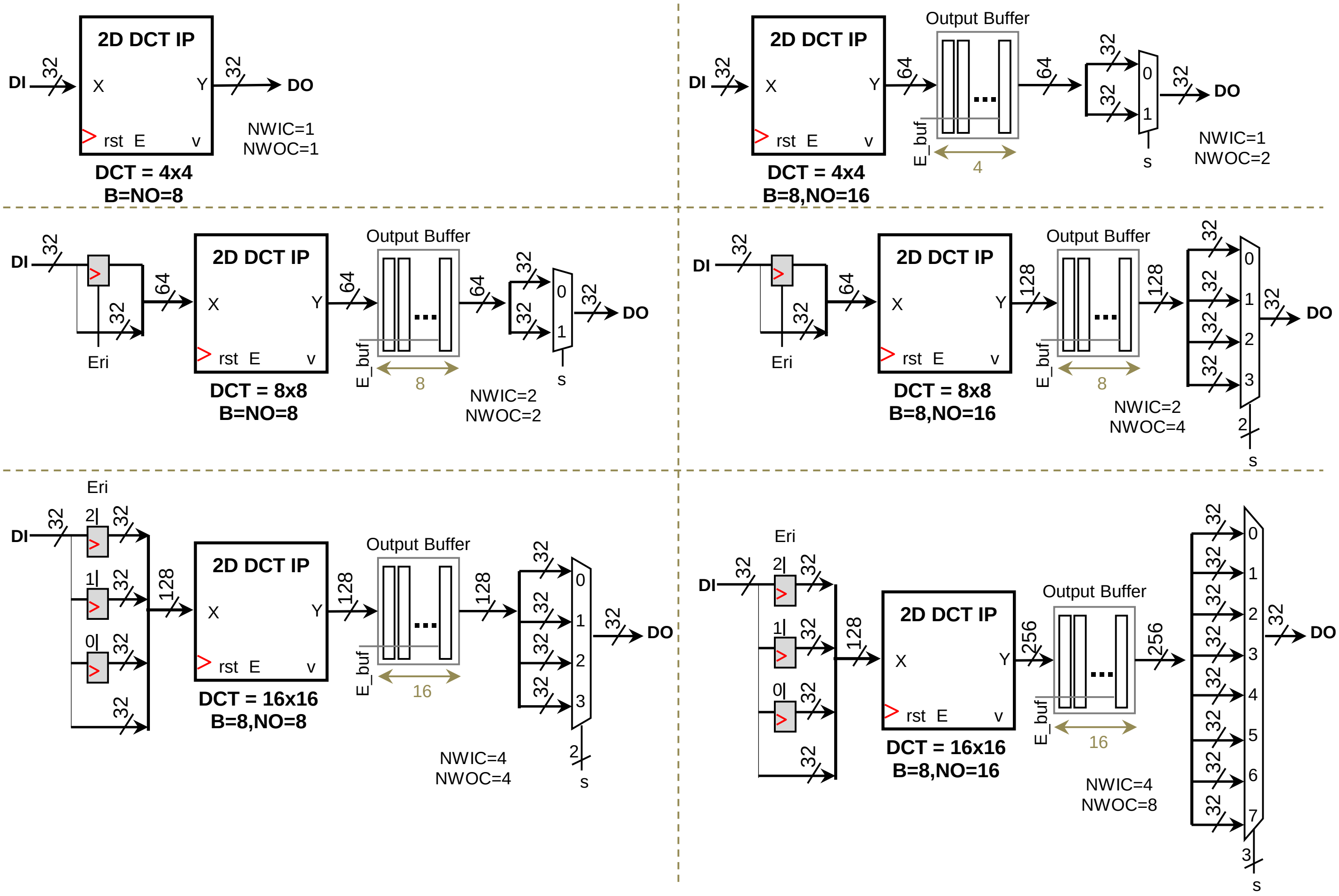
- **Glue logic between 2D-DCT and FIFOs**:
  - ✓ Input Interface: This is just a bunch of registers that capture data 32 bits at a time. The 2D DCT data input is more than 32 bits (usually B=8, N=4, 8, 16): we input N groups of $N \times B$ bits.
  - ✓ Output Buffer: The 2D DCT generates $N$ groups of $N \times NO$ bits in successive cycles. As we usually cannot place this amount of data fast enough on oFIFO, 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 $N \times NO$ bits.
  - ✓ $NWIC = \frac{N}{\lceil 32/B \rceil}$. This is the number of words per input column
  - ✓ $NWOC = NWIC \times \left\lceil \frac{NO}{B} \right\rceil$. This is the number of words per output column (or row).
  - ✓ The following table displays the values of NWIC and NWOC for common DCT sizes and values of B and NO:

| DCT | B=8 | NO=8 | NO=16 |
|---|---|---|---|
| 4x4 | NWIC=1 | NWOC = 1 | NWOC = 2 |
| 8x8 | NWIC = 2 | NWOC = 2 | NWOC = 4 |
| 16x16 | NWIC = 4 | NWOC = 4 | NWOC = 8 |

- $reset$ 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 will be more helpful if we want to later perform Partial Reconfiguration.



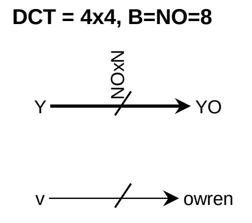
- **Glue Logic examples**: The figure depicts different input/output interfaces to the 2D DCT IP core along with the Output buffer. They depend on the parameter N, B, and NO.
  Note that when DCT=4x4 and B=NO=8, there is no need for the extra buffer or for any glue logic. In all the other cases, we do need an output buffer as the oFIFO is only 32-bits wide.



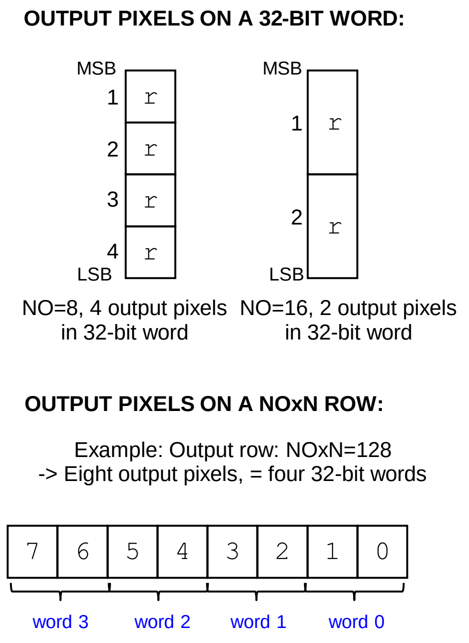
- **FSM @ S_AXI_ACLK**
  - ✓ This is the same FSM as the one for the Pixel Processor.

- **FSM @ CLKFX**:
  - ✓ This FSM handles:
    - ▫ The inner side of the FIFOs. For iFIFO, this is $iempty$, $irden$; for OFIFO, this is: $ofull$, $owren$. For the 2D DCT IP, the FSM checks whether iFIFO is not empty and oFIFO is not full, before attempting to write data on the 2D DCT IP.
    - ▫ Control signals to the 2D DCT IP $(E, v)$.
    - ▫ Control signals to the interface between the FIFOs and the 2D DCT input/output data signals: $s, Eri, E\_buf$.
  - ✓ For simplicity's sake, we divide this FSM @ CLKFX in two parts: (i) Output FSM: It controls the output interface, output buffer, and oFIFO, and (ii) Input FSM: It controls the input interface and iFIFO.

  - ✓ **Output FSM**
    The figure depicts the output interface control and $owren$ generation. (a) $owren = v$ and no output buffer when DCT=4x4, B=BO=8. (b) Output buffer and FSM that generates $E\_buf$ and $owren$. (c) Order of output pixels in a 32-bit word (same for input pixels) and on a $NO \times N$ output row.

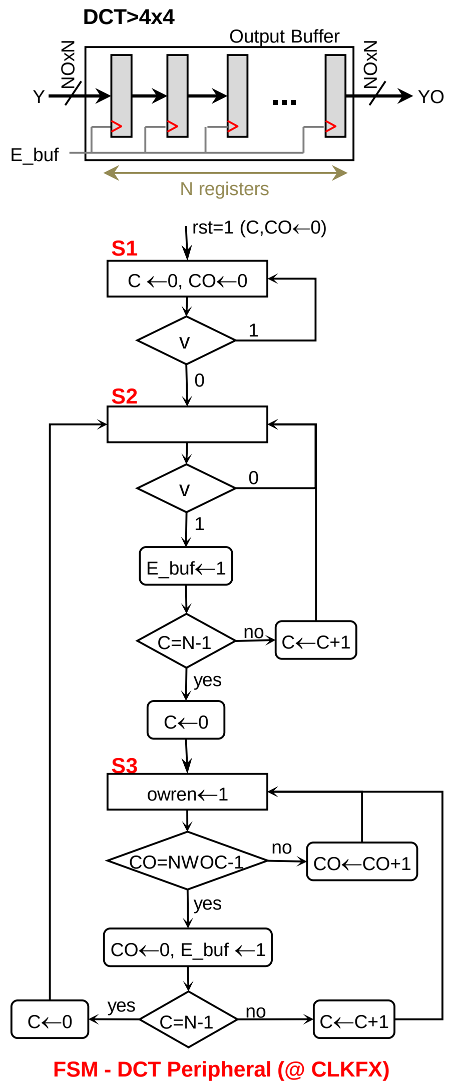


(a)


OUTPUT PIXELS ON A 32-BIT WORD:
MSB
1
2
3
4
LSB
r
r
r
r
MSB
1
2
LSB
r
r
NO=8, 4 output pixels in 32-bit word
NO=16, 2 output pixels in 32-bit word
OUTPUT PIXELS ON A NOxN ROW:
Example: Output row: NOxN=128
-> Eight output pixels, = four 32-bit words
7
6
5
4
3
2
1
0
word 3
word 2
word 1
word 0

When storing on oFIFO, the order is :
*word 3, word 2, word 1, word 0*
Thus s = NWOC-1-CO

(c)



FSM - DCT Peripheral (@ CLKFX)

(b)

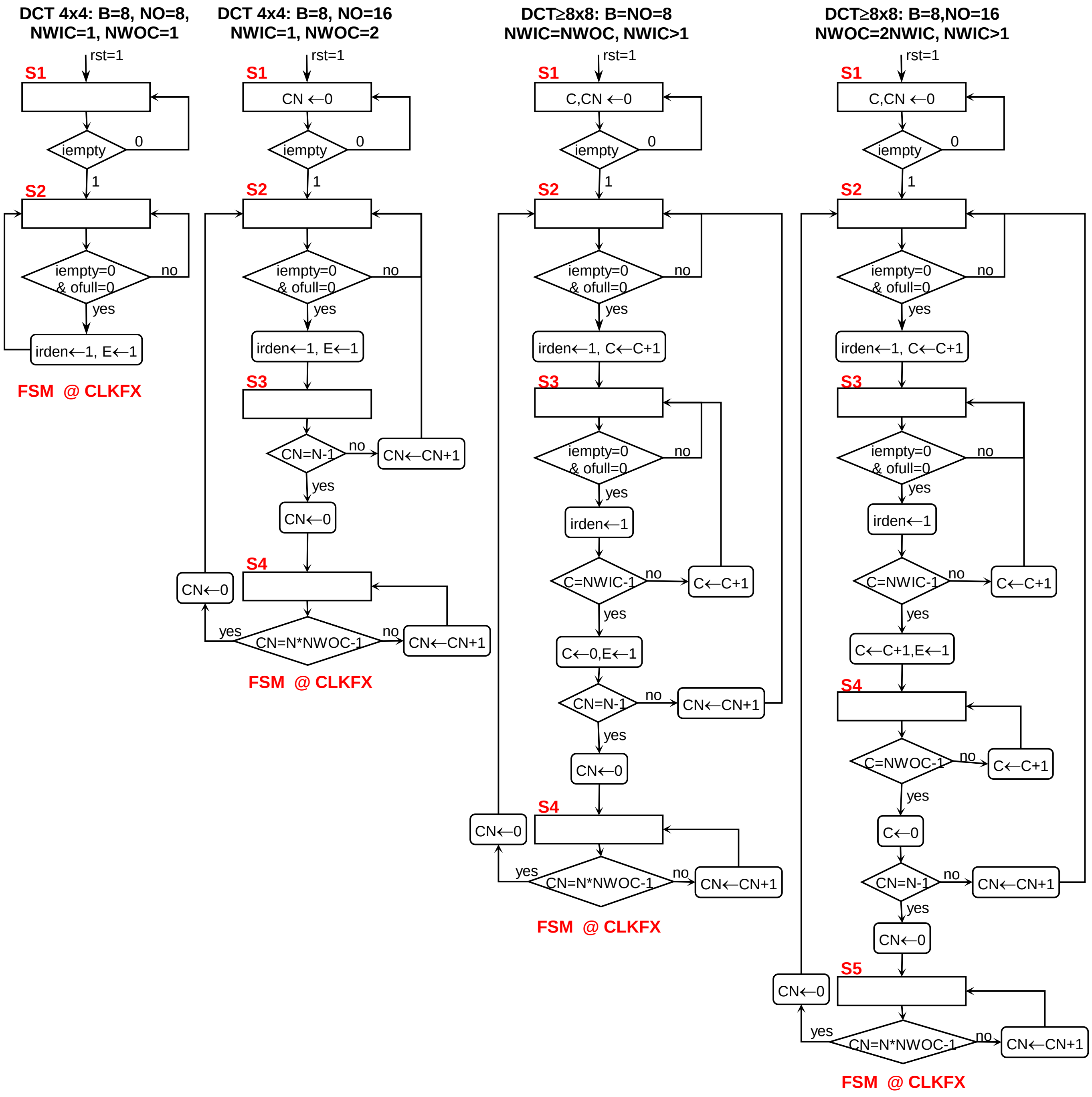
✓ **Input FSM**

- To avoid data in the output buffer to be overrun by a new block, there must be $N \times NWOC$ cycles between the 'v' of the last row of an output block and the 'v' of the first row of the next output block. This is satisfied if we wait $N \times NWOC$ cycles between the assertion of 'E' for the last column of an input block and the assertion of 'E' for the first column of the next input block.
- $Eri$ generation: The table below shows the value of $Eri$ for DCT=8x8 and 16x16. For 4x4, $Eri$ is not required. In general, the formula is: $Eri = \frac{2^{NWIC-1-C}}{2(drop\ LSB)}\ AND\ irden$.

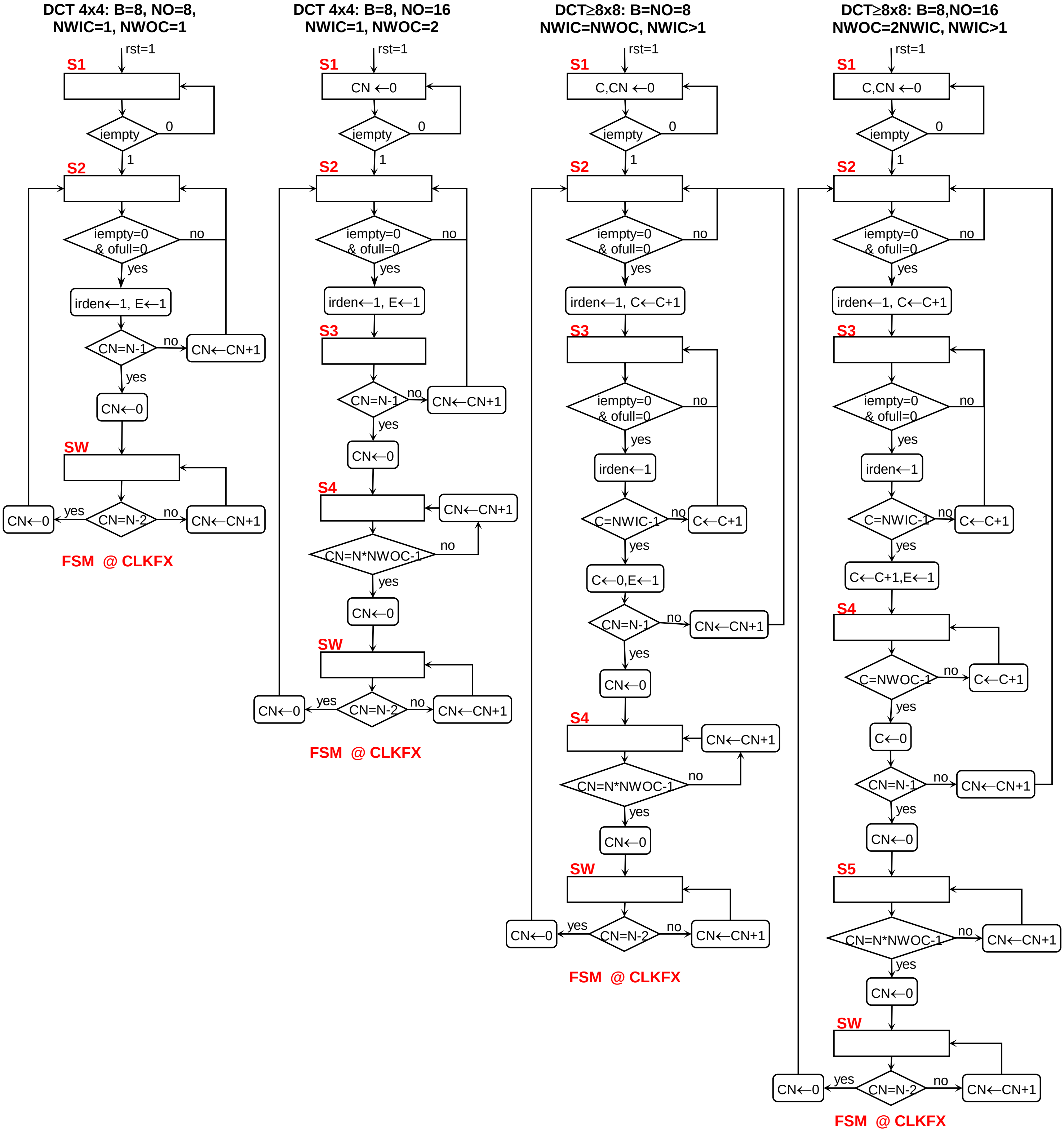
| DCT = 8x8 | | | DCT = 16x16 | | |
|---|---|---|---|---|---|
| C | irden | Eri | C | irden | Eri |
| 0 | 1 | 1 | 0 | 1 | 100 |
| 1 | 1 | 0 | 1 | 1 | 010 |
| X | 0 | 0 | 2 | 1 | 001 |
| | | | 3 | 1 | 000 |
| | | | X | 0 | 000 |

- There are two variations of the DCT 2D IP core:
  - Fully pipelined case: Selected with the parameter `IMPLEMENTATION=fullypip`. Assuming no I/O constraints, we can feed a new block to the 2D DCT IP core right after the previous one. Note that due to the FIFOs, we must wait $NxNWOC$ cycles between input blocks.
  - One Transpose case: Selected with the parameter `IMPLEMENTATION=onetrans`. Assuming no I/O constraints, we have to wait N-1 cycles before feeding a new block to the 2D DCT IP core right after the previous one. Note that due to the FIFOs, we must wait an extra $NxNWOC$ cycles between input blocks.

- Fully Pipelined case: The figure depicts the case for 2D DCTs of different sizes:



- One Transpose case: The figure depicts the case for 2D DCTs of different sizes:

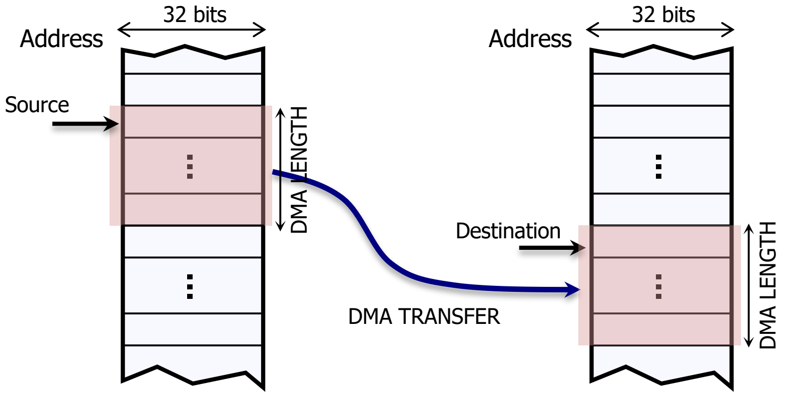


- **Input/Output Example** (N=4, B=8, NO=16)**:** The outputs have been verified (with a MATLAB model) to be correct. For the inputs, each 32 bit word is a column (top to bottom). For the output, each two 32-bit words is a row (left to right).

| Input (columns) | Output (rows) |
|---|---|
| 0xDEADBEEF<br>0xBEBEDEAD<br>0xFADEBEAD<br>0xCAFEBEDF | 0x8000E92E<br>0x14C00D82<br>0x18A6E418<br>0xDB3E1FB2<br>0x0A401E19<br>0x1D40236D<br>0xF8382A32<br>0xDEC9FDE7 |
| 0xCFC7C9C7<br>0xCAC4C6C3<br>0xC6C3C7C3<br>0xBEBDC2BD | 0x80000CF4<br>0xFF0003D5<br>0x0471045F<br>0xFF89FF65<br>0x010003CE<br>0x0000000B<br>0x06D0FFE5<br>0x00310020 |

- **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.

## DIRECT MEMORY ACCESS (DMA)



- The DMA controller (DMAC) is available inside the Processing System (PS). It uses a 64-bit AXI master interface to perform DMA transfers to/from system memories and PL peripherals. The transfers are controlled by the DMA instruction execution engine. The DMAC is able to move large amounts of data without processor intervention, leading to faster data transfers.
- The source or destination memory can be anywhere in the system (PS or PL).
- The user can configure up to eight DMA channels (0-7). Each channel corresponds to a thread running on the DMA's engine processor. We can issue commands for up to eight read and up to eight write AXI transactions.

- The DMA Controller can generate the following Interrupt Signals to the PS Interrupt Controller:

| **Interrupt Name** | **Zynq-7000 SoC – IRQ ID #** |
|---|---|
| DMA Operation Done Channel 0 | 46 |
| DMA Operation Done Channel 1 | 47 |
| DMA Operation Done Channel 2 | 48 |
| DMA Operation Done Channel 3 | 49 |
| DMA Operation Done Channel 4 | 72 |
| DMA Operation Done Channel 5 | 73 |
| DMA Operation Done Channel 6 | 74 |
| DMA Operation Done Channel 7 | 75 |
| DMA Abort | 45 |

- There are other DMA controllers in the system that are local to the I/O peripherals in the PS. These include:
  - ✓ GigE controller.
  - ✓ USB controller.
  - ✓ SDIO controller: for SD (Secure Digital) memory cards, MMC (MultiMedia Cards).
  - ✓ DevC (Device Configuration) Interface: for Device Boot and PL Configuration.

- For more information, refer to the Xilinx® Zynq-7000 AP SoC Technical Reference Manual (UG585) – Chapter 9. For a list of available functions (SDK 2016.2), look into the `xdmaps.h` file in the *bsp*: `/libsrc/dmaps_v2_1/src`.

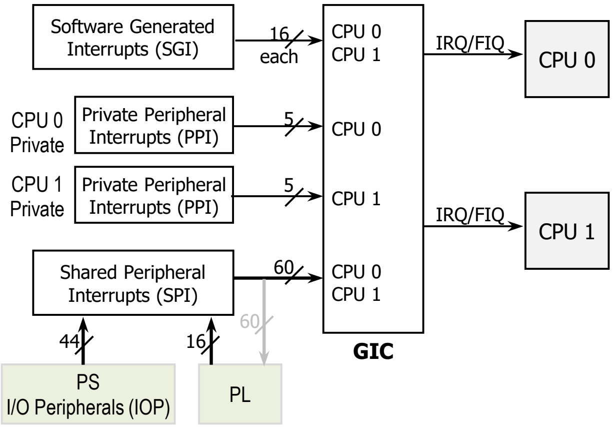
## INTERRUPTS

- In embedded systems, an interrupt is a signal that temporarily pauses the processor's current activities. The processor saves its current state and executes an Interrupt Service Routine (ISR) to address the reason for the interrupt. An interrupt can come from the following places:
  - ✓ Hardware: A signal directly connected to the processor.
  - ✓ Software: A software instruction loaded by the processor.
  - ✓ Exception: An exception generated by the processor when an error or an exceptional event occurs.
- Interrupts can be either maskable or non-maskable. Maskable interrupts can be safely ignored by setting a particular bit in a processor register. Non-maskable interrupts cannot be ignored. Interrupt signals can be edge triggered or level triggered.
- Using interrupts allows the processor to continue processing until an event occurs, at which time the processor can address the event. This interrupt-driven approach also enables a faster response time to events than a polled approach, in which a program actively samples the status of an external device in a synchronous manner.

### ZYNQ-7000 SOC'S INTERRUPT STRUCTURE

- **Generic Interrupt Controller (GIC)**: This is a centralized resource for managing interrupts sent to the CPUs from the PS and PL. The controller enables, disables, masks, and prioritize the interrupt sources and sends them to the selected CPU (or CPUs) in a programmed manner as the CPU interface accepts the next interrupts.
- All of the interrupt requests (PPI, SGI, and SPI) are assigned a unique ID number. The GIC uses the ID number to arbitrate.
- The GIC handles interrupts from the following sources:
  - ✓ ***Software-generated Interrupts (SGI)***: 16 interrupts available (for each CPU). They can interrupt one or both of the CPUs. The sensitivity types for SGIs are fixed and cannot be changed.

| Interrupt Name | IRQ ID # | Type |
|---|---|---|
| Software 0 | 0 | Rising edge |
| Software 1 | 1 | |
| Software 2 | 2 | |
| … | … | |
| Software 15 | 15 | |


Software Generated Interrupts (SGI)
16
each
CPU 0
CPU 1
IRQ/FIQ
CPU 0
CPU 0 Private
Private Peripheral Interrupts (PPI)
5
CPU 0
CPU 1 Private
Private Peripheral Interrupts (PPI)
5
CPU 1
IRQ/FIQ
CPU 1
Shared Peripheral Interrupts (SPI)
60
CPU 0
CPU 1
60
44
16
GIC
PS
I/O Peripherals (IOP)
PL

  - ✓ ***Private peripheral Interrupts (PPI)***: Each CPU connects to a private set of 5 peripheral interrupts. The sensitivity types for PPIs are fixed and cannot be changed. Note that the fast interrupt (FIQ) and the interrupt (IRQ) signals from the PL are inverted and then sent to the interrupt controller (i.e., they are active High at the PS-PL interface, but Active Low when they reach the GIC).

| Interrupt Name | IRQ ID # | Type | Description |
|---|---|---|---|
| Global Timer | 27 | Rising edge | |
| nFIQ | 28 | Active Low level | Fast interrupt signal from PL |
| CPU Private Timer | 29 | Rising edge | |
| AWDT{0,1} | 30 | Rising edge | Private watchdog timer for each CPU |
| nIRQ | 31 | Active Low level | Interrupt signal from PL |

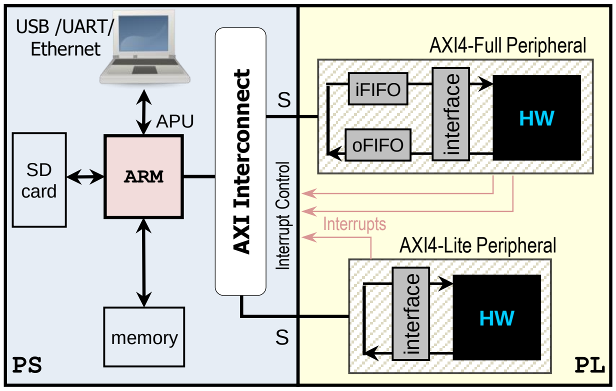
  - ✓ ***Shared peripheral Interrupts (SPI)***: 60 interrupts available. These interrupts can come from the I/O peripherals and various modules (44), or from the programmable logic (PL) side of the device (16). Note that the PL can also accept interrupts coming from the PL. They are shared between the Zynq SoC's two CPUs. Except for interrupts coming from the PL (IRQ #61 through #68 and #84 through #91), all interrupt sensitivity types are fixed and cannot be changed. The table below shows the PL interrupts as well as interrupts coming from common I/O peripherals in the PS.

| Source | Interrupt Name | IRQ ID # | Type |
|---|---|---|---|
| PL | PL [15..8] | 91:84 | Rising edge/High Level |
| | PL [7..0] | 68:61 | Rising edge/High Level |
| DMAC | DMAC [7..4] | 75:72 | High Level |
| | DMAC [3..0] | 49:46 | |
| | DMAC Abort | 45 | |
| Timer | TTC 0 | 44:42 | High Level |
| | TTC 1 | 71:69 | |
| IOP | GPIO | 52 | High Level |
| | USB 0 | 53 | |
| | USB 1 | 76 | |
| | I2C 0 | 57 | |
| | I2C 1 | 80 | |
| | UART 0 | 59 | |
| | UART 1 | 82 | |

- For interrupts coming from the PS, each particular peripheral handles the interrupts in their own way (see DMA controller). Refer to the documentation and examples available for every controller in SDK (see the `/libsrc` folder in the *bsp*).
- For interrupts coming from the PL, we need to create the hardware support and then deal with the software drivers.
- For more information, refer to the Xilinx® Zynq-7000 AP SoC Technical Reference Manual (UG585) – Chapter 7. For a list of available functions (SDK 2016.2), look into the `xscugic.h` file in the *bsp*.

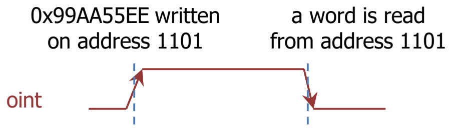
## INTERRUPTS COMING FROM THE PL

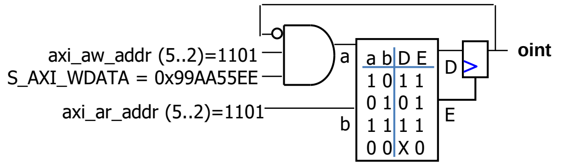
- A circuit inside the PL can generate one or more interrupts that are then connected to the PS. The interrupts can be asserted due to any event that the designer specifies (e.g.: arithmetic overflow, result ready).
- Up to 16 Interrupt signals can be connected.
- The interrupt type can be configured via software to either High Level or Rising Edge.



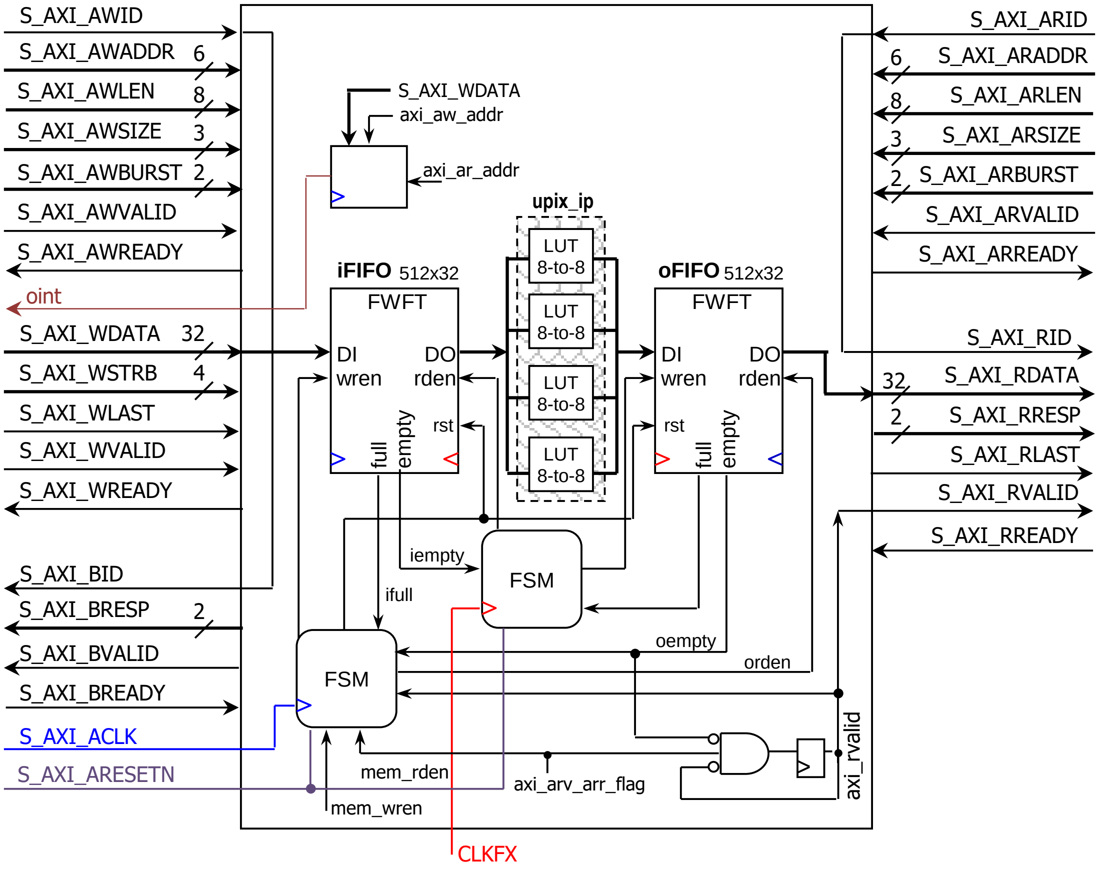
## Case Example: Pixel Processor (PS+PL)

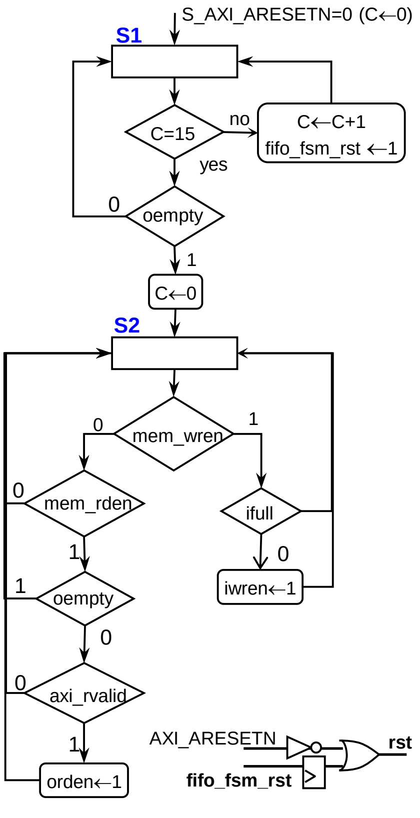
- Here, the Pixel Processor interface generates an interrupt signal $oint$. The figure depicts the block that generates this signal.
- The $oint$ signal is asserted when the PS writes a specific word ($0x99AA55EE$) on address `1101`. This allows us to properly tests interrupts. Note: even though the interrupt is caused via software, this is not a Software Interrupt.
- This interrupt signal is asserted until the PS detects it. At this point, the ISR needs to de-assert the interrupt signal (so that the signal does not continuously interrupt the PS). This is performed by reading from address `1101`.


0x99AA55EE written on address 1101
a word is read from address 1101
oint



| a | b | D | E |
|---|---|---|---|
| 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |
| 0 | 0 | X | 0 |





FSM at S_AXI_ACLK