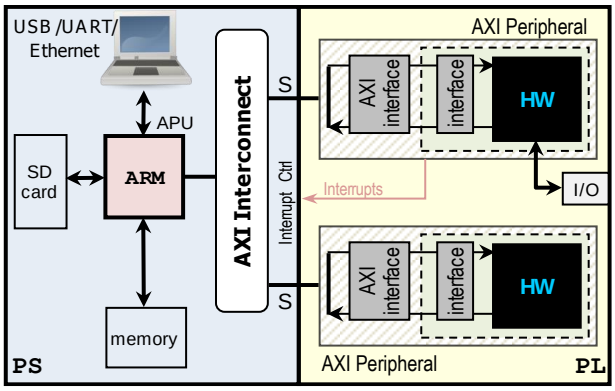
# Unit 5 – Embedded Systems in PSoC

## Embedded System on PSoC

- Zynq-7000 PSoC devices include:
  - ✓ PS (Processing System): It includes an ARM microprocessor as well as many peripherals.
  - ✓ PL (Programmable Logic): This is the reconfigurable fabric.
- These two units can be interconnected via the AXI (Advanced eXtensible Interface) Bus.
- When an architecture is implemented in the PL, an AXI interface must be included for interconnection to the AXI Bus. An AXI Peripheral refers to the architecture and its AXI interface.



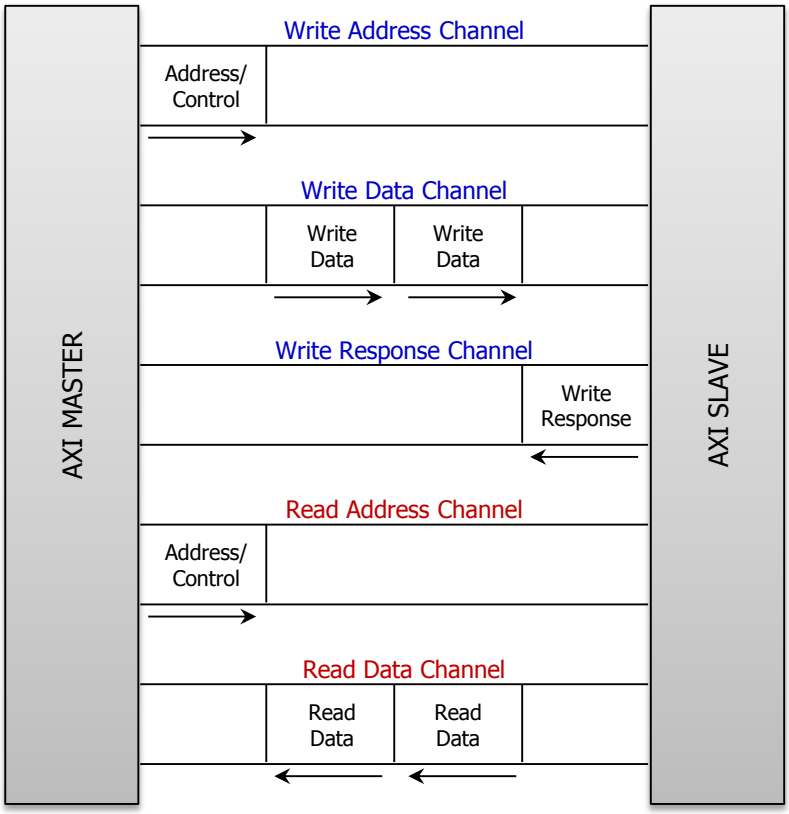

## AXI Bus

**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

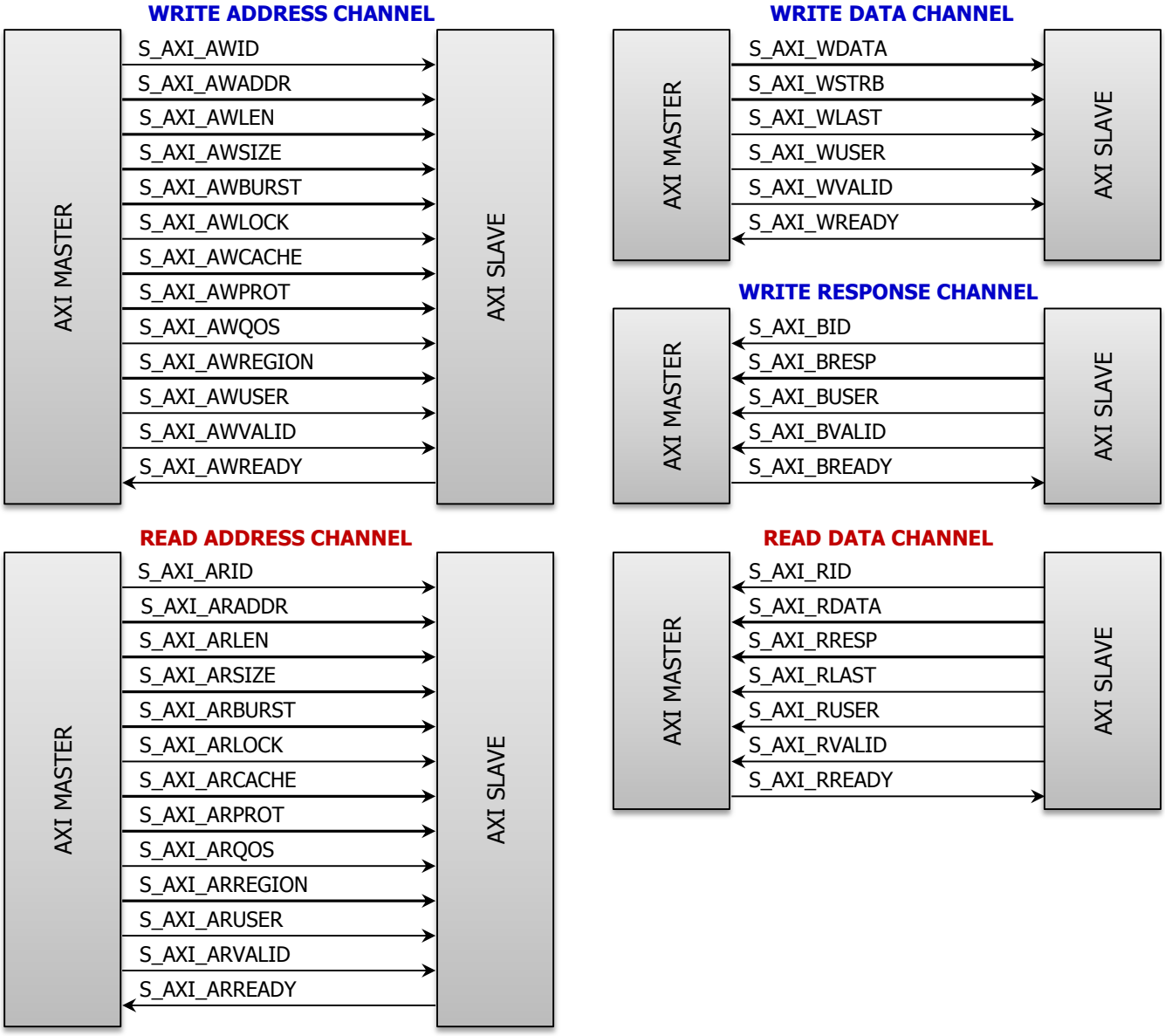
- The AXI protocol is burst-based and defines five independent transaction channels.
- Write Channel Architecture: Address and Control data is transmitted to the slave before a burst of data is transmitted, and a Write Response signaled following completion:
  - ✓ Write Address Channel
  - ✓ Write Data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

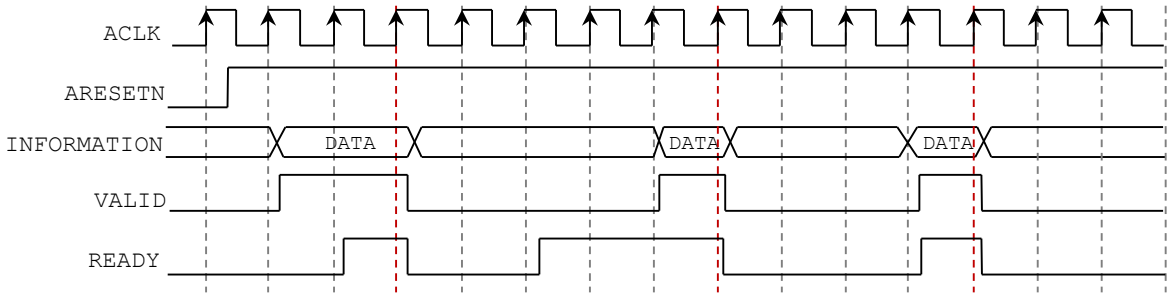
- **Signals:**
  Global System Signals:
  - ✓ S_AXI_CLK: AXI4 clock
  - ✓ S_AXI_ARESETN: AXI4 active-low reset.

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



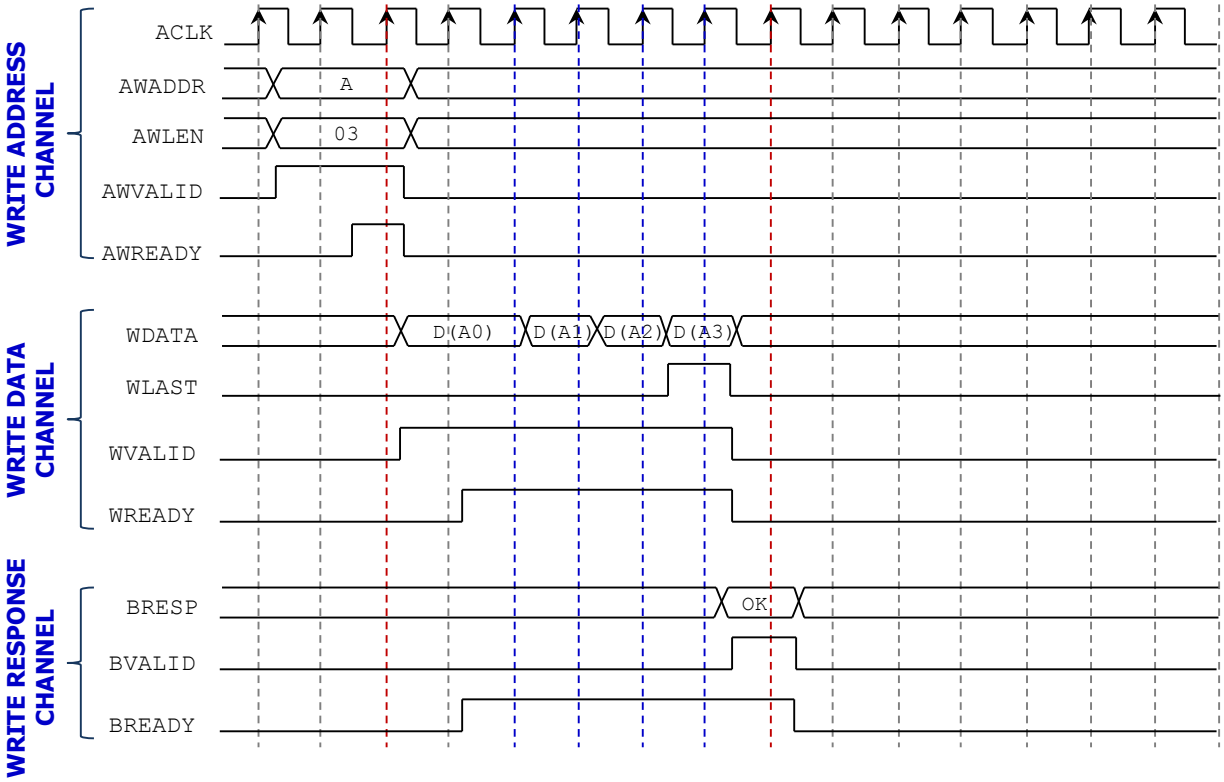

**AXI4-FULL PROTOCOL**

- 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.



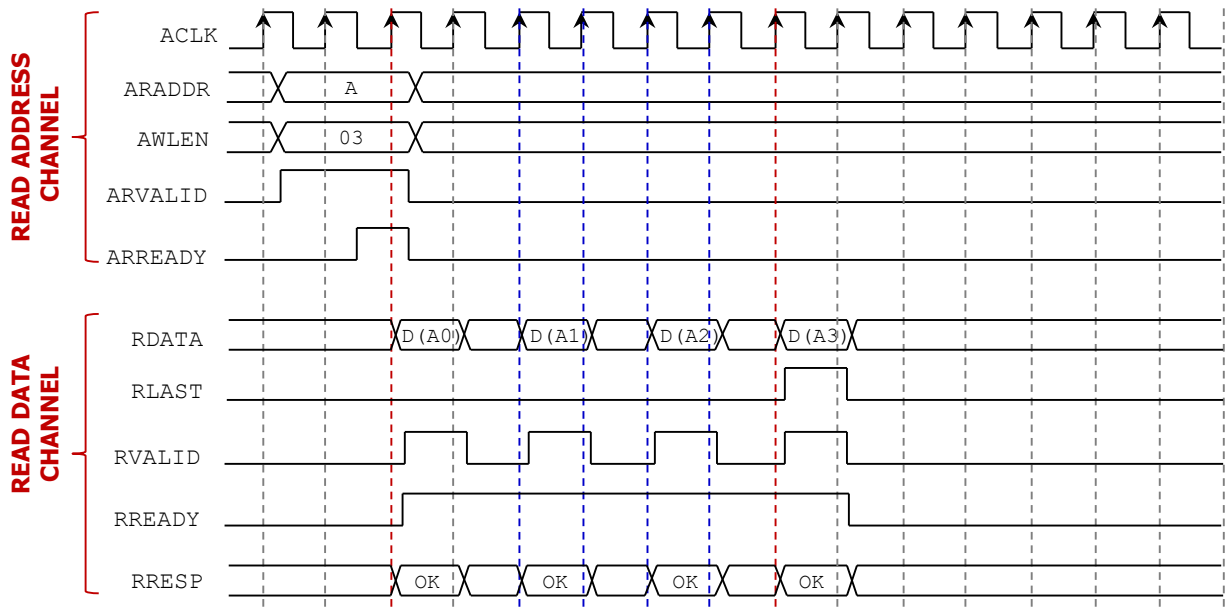
**Writing Transaction – Simple Memory:**

- 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).



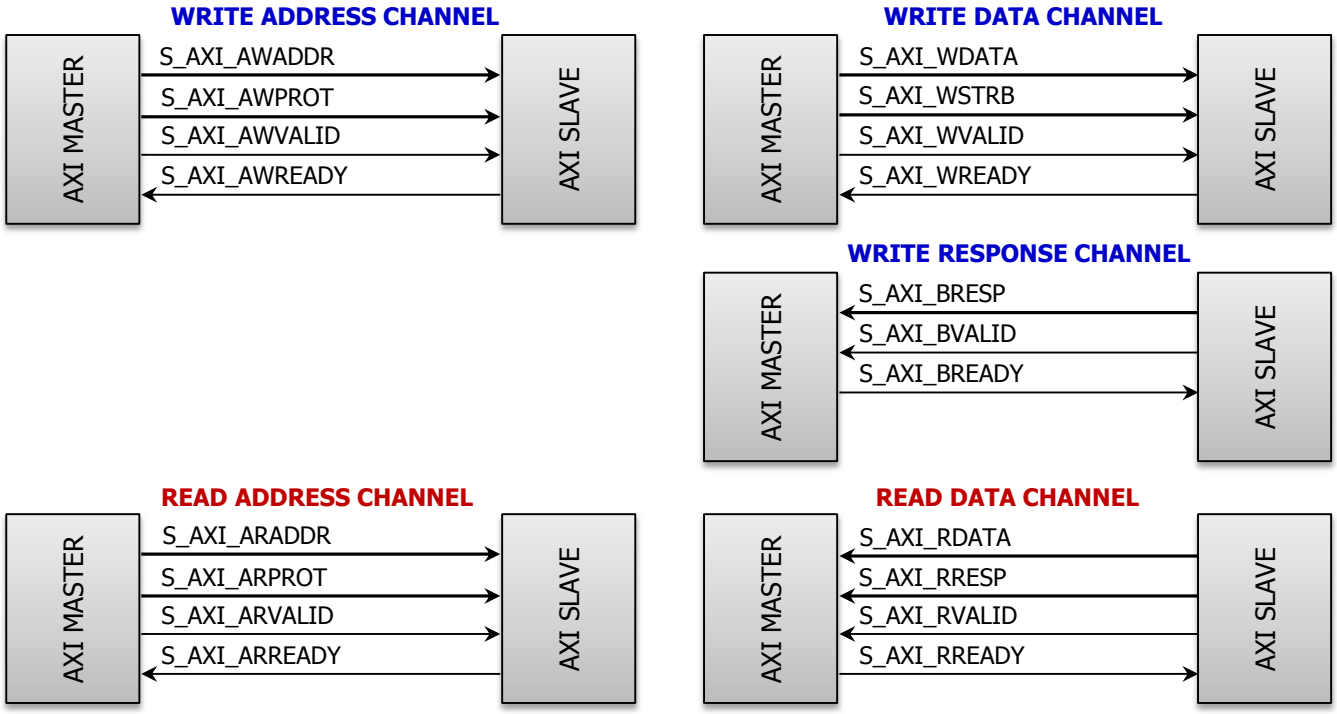

**Reading Transaction – Simple Memory:**

- 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

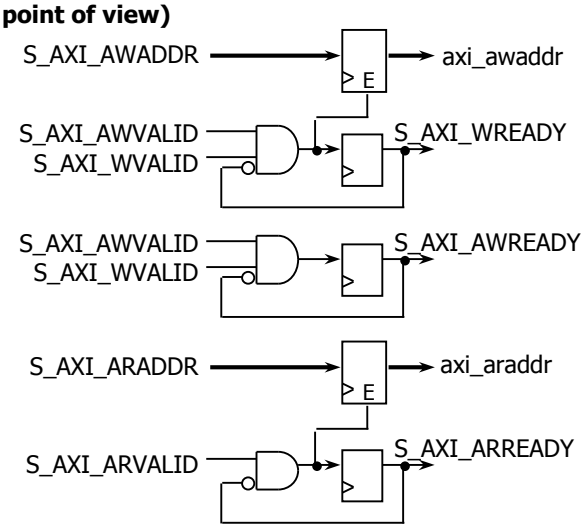
- This is a reduced version of the AXI4-Full. It does not support bursts, i.e., we only have one transaction at a time.
- Data bus: 32 or 64 bits.

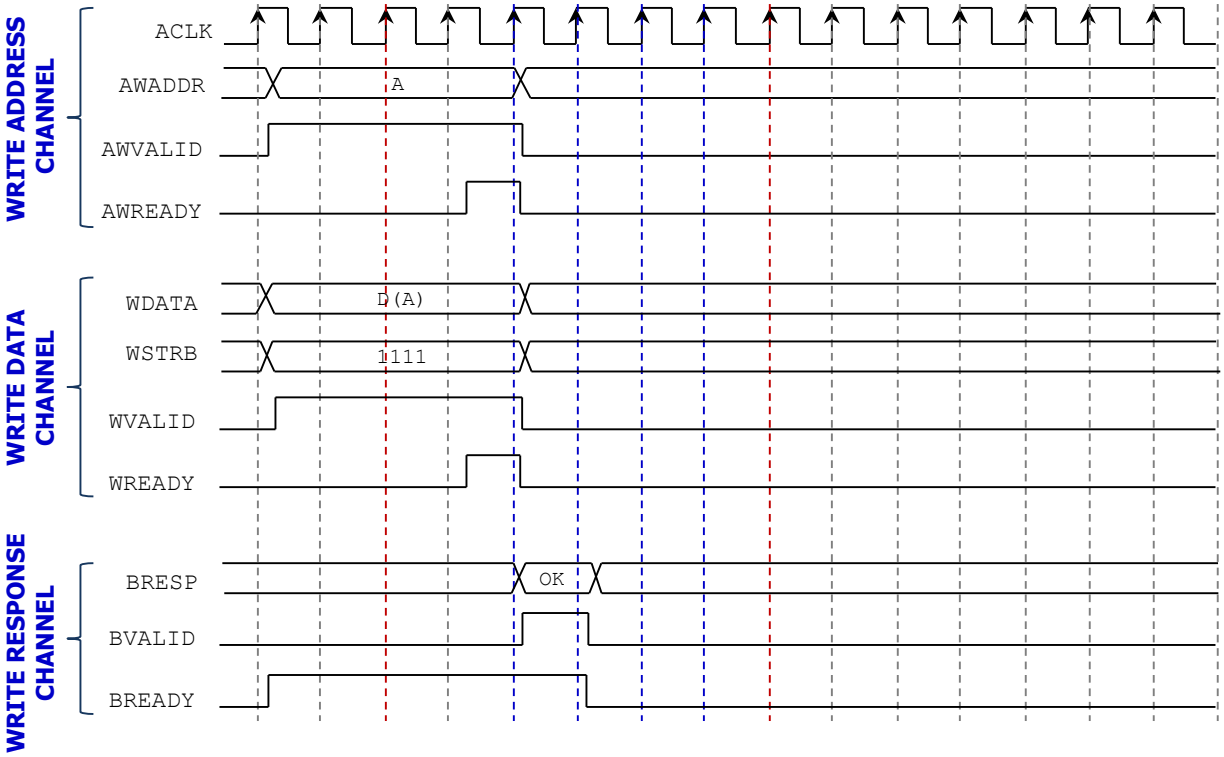


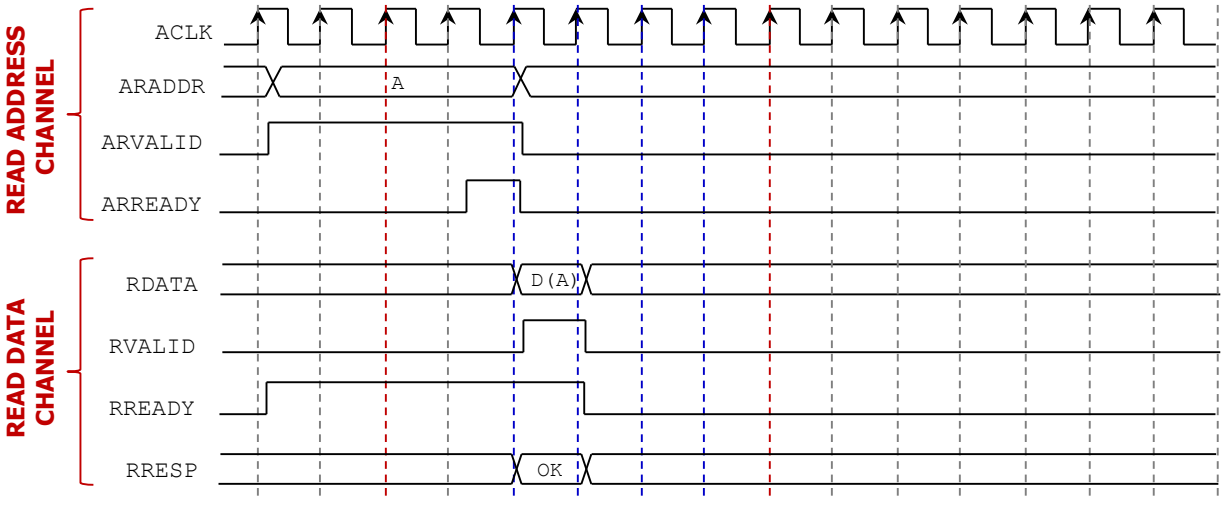

**AXI4-LITE PROTOCOL**

- The AXI Master Interface provided by Zynq-7000 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 Templates

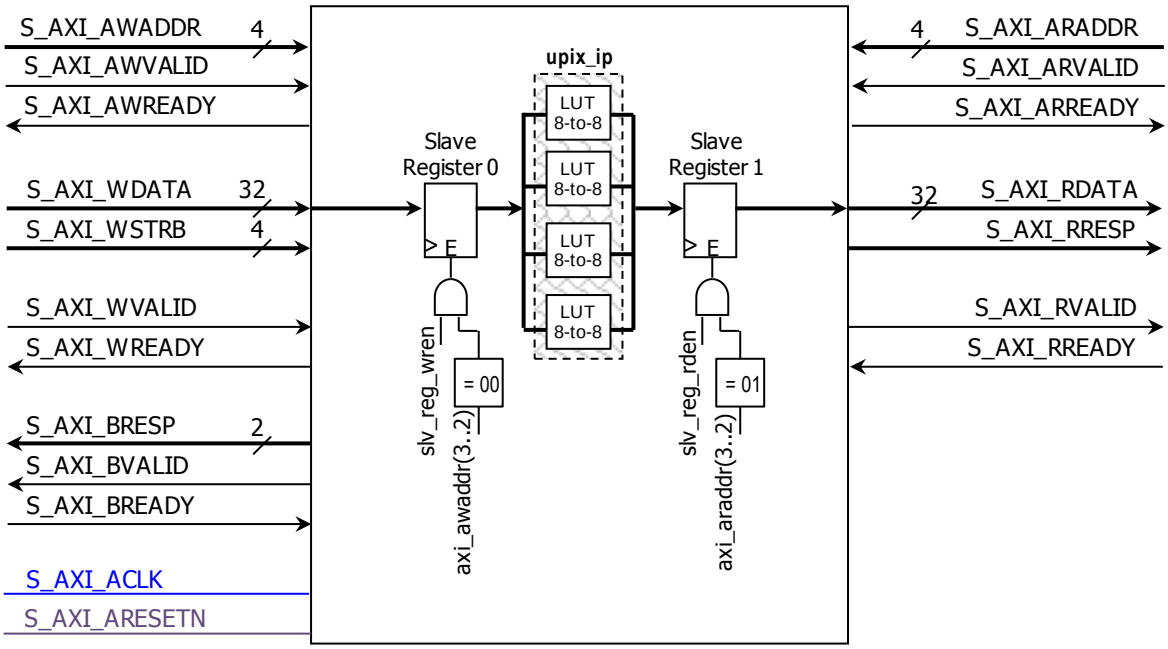
- AXI4-Lite Interface (Slave): Vivado 2019.1 provides a template based on the number of Slave Registers that the user specifies (4 by default). The template on its own can be used to write data on Slave Registers and read data from them in order to verify the functioning of the embedded system. In our examples, we need to modify the template to include our hardware.
- AXI4-Full Interface (Slave): Vivado 2019.1 provides a template based on the number of bytes selected (64 by default). The template includes an AXI4-Full Interface for a 64-bytes memory where we can read and write data using bursts. This interface needs to be modified by including our hardware and/or modifying the interface itself.
- The source files of the examples provided here can be downloaded at: Tutorial: Embedded System Design for Zynq PSoC.

# AXI4-Lite Interface - Examples

## AXI4-Lite: Pixel Processor

- **Custom Hardware Peripheral**: Pixel Processor (NI=8, NO=8, NC=4).
  - ✓ Operation: This is a combinational circuit that that generates output data as soon as input data is present.

- **AXI4-Lite Interface**:
  - ✓ Simple interface with two slave registers (for reading and writing):
    - Slave Register 0: Master writes data on the Slave Peripheral. Here, $\text{axi\_awaddr}(3..2) = 00$.
    - Slave Register 1: Master reads data from the Slave Peripheral. Here, $\text{axi\_awaddr}(3..2) = 01$.
  - ✓ $slv\_reg\_wren$: It indicates that new data is available on a Slave Register. $slv\_reg\_wren = S\_AXI\_WREADY\ and\ S\_AXI\_WVALID\ and\ S\_AXI\_AWREADY\ and\ S\_AXI\_AWVALID$. This signal is pulse with a duration of one clock cycle. It indicates that data is available on the input that will be captured on a Slave Register.
  - ✓ $slv\_reg\_rden$: 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 $slv\_reg\_rden = 1$.



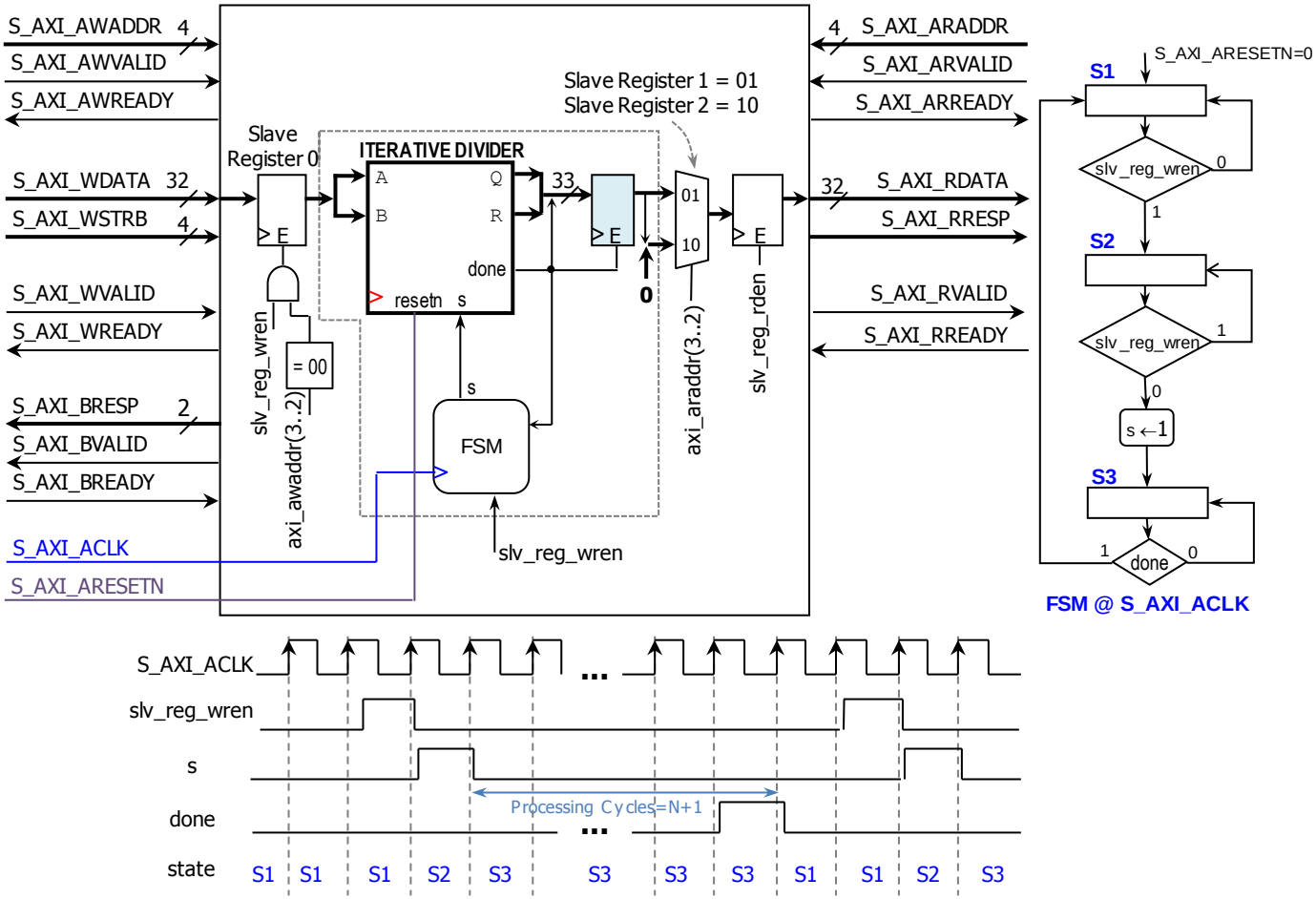

  - ✓ Address ($S\_AXI\_AWADDR, S\_AXI\_ARADDR$): In this example, we selected only two registers, but Vivado 2019.1 creates a template with a minimum of four 32-bit registers. So, we have 16 bytes, hence the 4 bit addresses, from which we only use the 2 MSBs to identify the 32-bit Slave Registers: Register 0 is given the 00 code, and Register 1 the 01 code.

- **Software routine**: It writes one 32-bit word, and then reads one 32-bit word.

## AXI4-LITE: SEQUENTIAL (OR ITERATIVE) DIVIDER

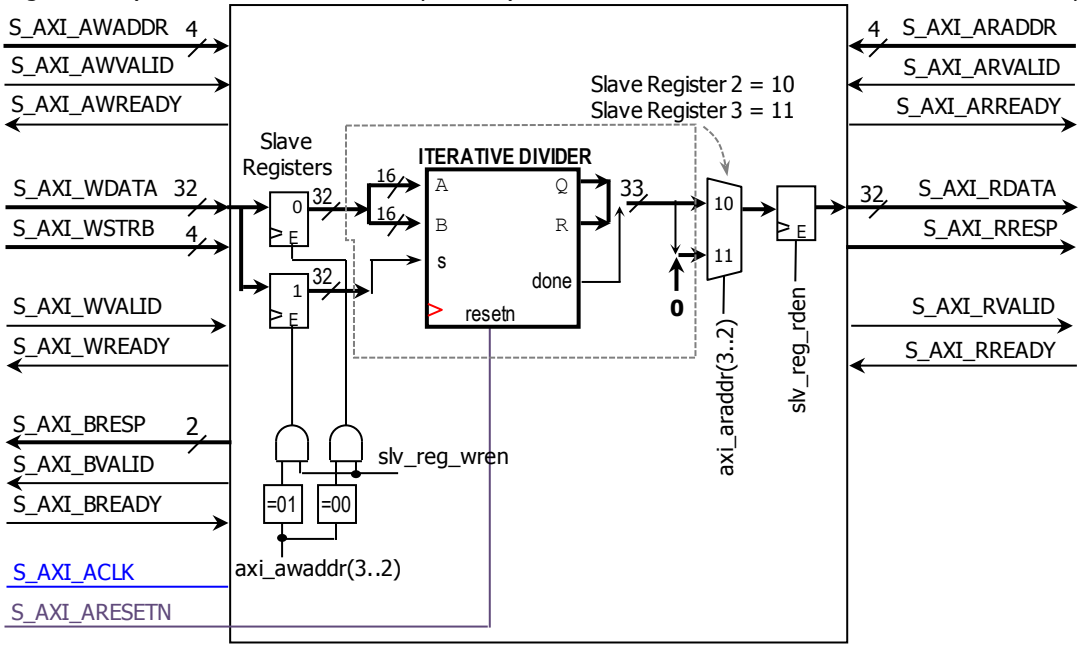
- **Custom Hardware Peripheral**: Iterative Integer Divider (N=16, M=16).
  - ✓ Operation: The circuit reads input data (16-bit A, 16-bit B) when the s signal (usually a one-cycle pulse) is asserted. When the result (16-bit Q, 16-bit R) is ready (after N+1=17 cycles), the signal done is asserted. Only after this, we can feed a new input data set (with s = 1). Note that this is how iterative circuits work.

- **AXI4-Lite Interface**:
  - ✓ Simple interface with 3 Slave Registers (for reading and writing), a Register, and an FSM.
    - 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. $axi\_araddr\ (3..2) = 10$
  - ✓ When using Slave Registers we need to consider $axi\_awaddr$ and $axi\_araddr$ to identify the registers to/from we write/read. We use the signal $slv\_reg\_wren$ to determine whether data is present on Slave Register 0. But note that data is present on Slave Register 0 one cycle after $slv\_reg\_wren$.
  - ✓ Slave Registers 1 and 2: No need for a physical register for each Slave Register. A multiplexor along with a 32-bit register suffice in this case. The inputs of the multiplexor will be the corresponding Slave Register signals: slv_reg1, slv_reg2.
  - ✓ The extra Register is a buffer that stores the output results (when done=1). Digital systems with an iterative behavior usually keep the output values until a new input data set is captured, but others do not. Thus, it is always good practice to store the output results in a buffer until they are read by the AXI4-Lite interface.
  - ✓ Input Data: 32 bits (A, B). Slave Register 0 contains A&B.
  - ✓ Output Data: 33 bits (Q, R, done). Though not necessary, the signal done is included to verify that i) the output data was captured when done=1, and ii) the circuit is working (done is being issued). Output data is captured in the buffer when done=1. Since the output interface is 32-bits wide, the 33 bits have to be multiplexed into two 32-bit words.
  - ✓ FSM: It issues a one-cycle pulse for s after a pulse on $slv\_reg\_wren$. It then waits for done=1 before returning to S1. This means that the software must retrieve output data and verify that done=1 before trying to process a new input data set.




- **Software routine**: It writes one 32-bit word, and then reads two 32-bit words. One of these output words includes the signal done. If done=1, the software routine can start a new computation (write a 32-bit word, read two 32-bit words).
  * In most cases, by the time we read data, the circuit has already computed its result, but it is good practice to verify this.

- **AXI4-Lite Interface (alternative – software-controlled)**:
  - ✓ 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 Reads Data from the Slave Peripheral. $axi\_araddr\ (3..2) = 10$.
    - Slave Register 3: Master Reads Data from the Slave Peripheral. $axi\_araddr\ (3..2) = 11$
  - ✓ Slaves Registers 2 and 3: These are implemented with a multiplexor and a 32-bit register. The inputs of the MUX will be the corresponding Slave Register signals: slv_reg2, slv_reg3.
  - ✓ Input Data: 32 bits (A, B). Slave Register 0 contains A&B.
  - ✓ Output Data: 33 bits (Q, R, done). The signal done is included to verify that the Q&R is valid, and the circuit is working (done asserted). Since we have a 32-bit interface output, the 33 bits have to be multiplexed into two 32-bit words.
  - ✓ Note: In this interface the connections between the Slave Registers and the Iterative Divider are direct (no extra control). This means that data feeding, retrieving, and verification are carried out exclusively by the software routine.
  - ✓ Output data: Not captured in a buffer when done=1. But it must be kept until the AXI4-Lite interface reads it. It all depends on the Iterate Divider I/O mechanism: the circuit keeps its data if the input s is kept at 1, and the data is gone when s goes to 0 (so we can start a new computation). The software routine must issue s=1 and s=0 at the proper times.




- **Software routine**: The procedure is slightly more complex that in the previous case:
  - ✓ Write one 32-bit word (A&B), then another 32-bit word (with s=1). Note that s is being kept at 1. So, we know that the output data will be kept until we specifically set s = 0.
  - ✓ Read two 32-bit words (Q&R, and done). If done=1, output data is valid, and a new computation can be started.
    * By the time we read data, the circuit might have already computed its result, but it is good practice to verify this.
  - ✓ Before a new computation can be started, set s=0 (i.e., so, write a word on Slave Register 1: 0x00000000)
  - ✓ Now, we can start a new computation.

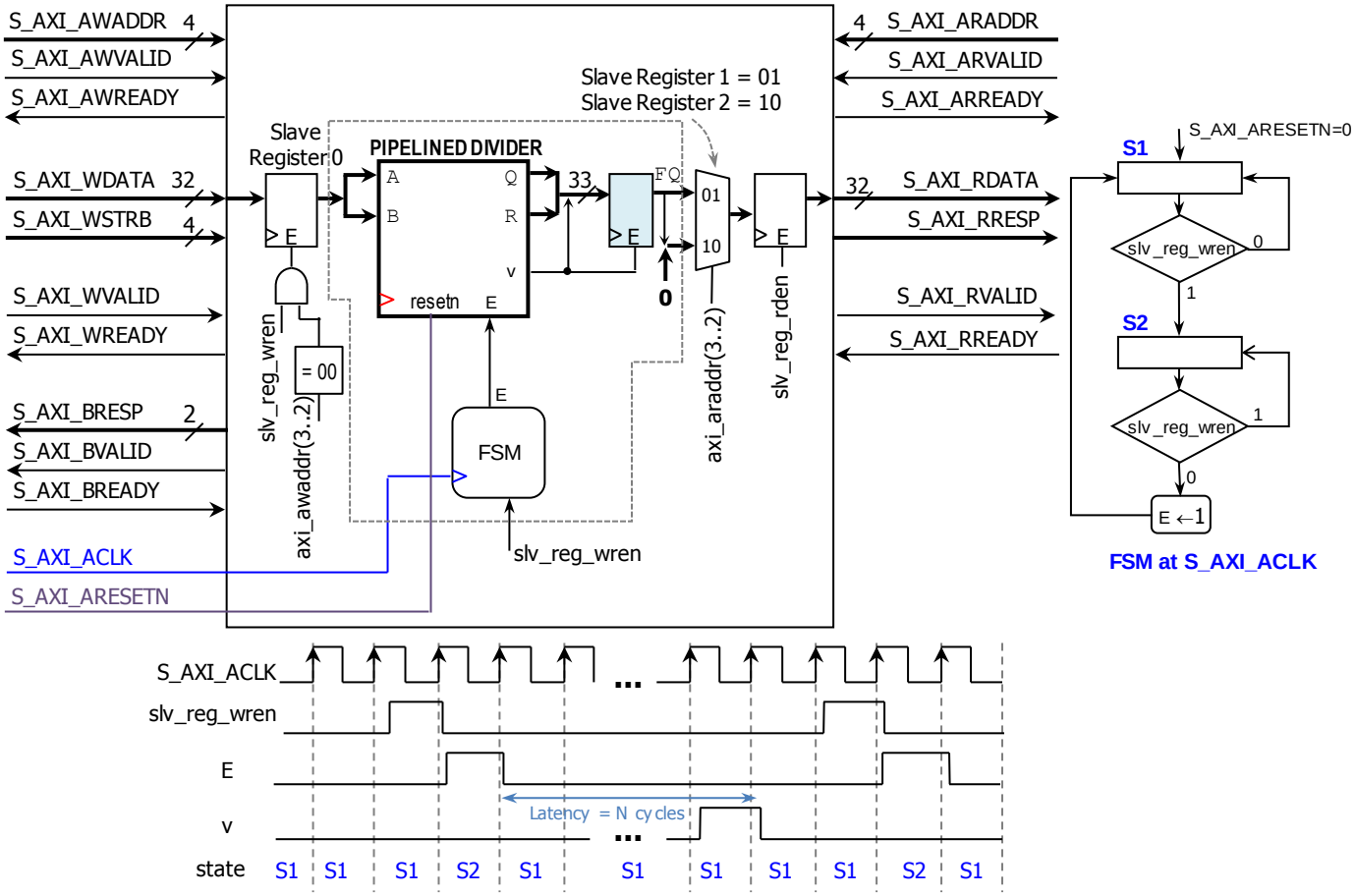
**TIPS: AXI4-Lite interface files**: If you call your interface 'my_intf', Vivado 2019.1 creates the following template files:

- <my intf>_v1_0.vhd: top file of the interface. No need to edit unless you plan to include extra I/Os in the interface.
- <my intf>_v1_0_S00.AXI.vhd: This file implements the AXI4-Lite interfacing and includes the Slave Registers. Input registers are actual 32-bit registers. Output registers: implemented as a MUX and a 32-bit register. Edit this file by only using the Slave Registers that you need and by connecting the Slave registers signals to I/Os in your design (let's call it 'my_core').
  - ✓ **Suggestion**: create a file on top of 'my_core', called 'my_core_ip' where you will include 'my_core' and the glue logic (e.g.: buffer register, FSM) required to connect 'my_core' to the slave register signals.
  - ✓ **Example**: AXI4-Lite interface called myaxidiv for Iterative Divider (my_diviter) design. The file hierarchy is as follows:
    myaxidiv_v1_0.vhd: Top file. No need to edit it.
    myaxidiv_v1_0_S00_AXI.vhd: Edit this file: Use only the required slave registers. Instantiate my_diviter_ip and connect the 3 slave register signals (slv_reg0, slv_reg1, slv_reg2), and signals slv_reg_wren, resetn, clock.
    my_diviter_ip.vhd: Build this circuit: instantiate my_diviter and include the glue logic (FSM, buffer register).
    my_diviter.vhd: This is your circuit and it includes any other components (.vhd) and ancillary files.
    ... (*extra files required for* mydiviter.vhd)

## AXI4-LITE: PIPELINED DIVIDER

- **Custom Hardware Peripheral**: Pipelined Integer Divider (N=16, M=16).
  - ✓ Operation: The circuit reads input data (16-bit A, 16-bit B) when the E (enable) signal is asserted. After a processing delay (N=16 cycles), the result (16-bit Q, 16-bit R) appears and it is signaled by v=1. As this circuit is pipelined, we can continuously feed and retrieve data (no need to wait until v=1 to feed a new input data set). This also means that valid output data is guaranteed to appear only for one clock cycle. Note that this is how pipelined circuits work.

- **AXI4-Lite Interface**:
  - ✓ Simple interface with 3 Slave Registers (for reading and writing), a Register, and an FSM.
    - 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. $axi\_araddr\ (3..2) = 10$
  - ✓ Extra Register: buffer that stores the output results when v=1. Pipelined circuits update the output when a new input data set is captured. Thus, you must store the output results in a buffer until they are read by the AXI4-Lite interface.
  - ✓ Input Data: 32 bits (A, B). Slave Register 0 contains A&B.
  - ✓ Output Data: 33 bits (Q, R, v). The signal v is included to verify that i) the output data was captured when v=1, and ii) the circuit is working (v was issued). Output data is captured in the buffer when v=1. Since the output interface is 32-bits wide, the 33 bits have to be multiplexed into two 32-bit words.
  - ✓ FSM: It issues a one-cycle pulse for E (after a pulse on *slv_reg_wren*). After S2, we could have gone to S3, where we would wait for v=1 before returning to S1. This is a slightly different approach, where this is handled via software: new input data can only be fed only when output data (with v=1) has been retrieved.



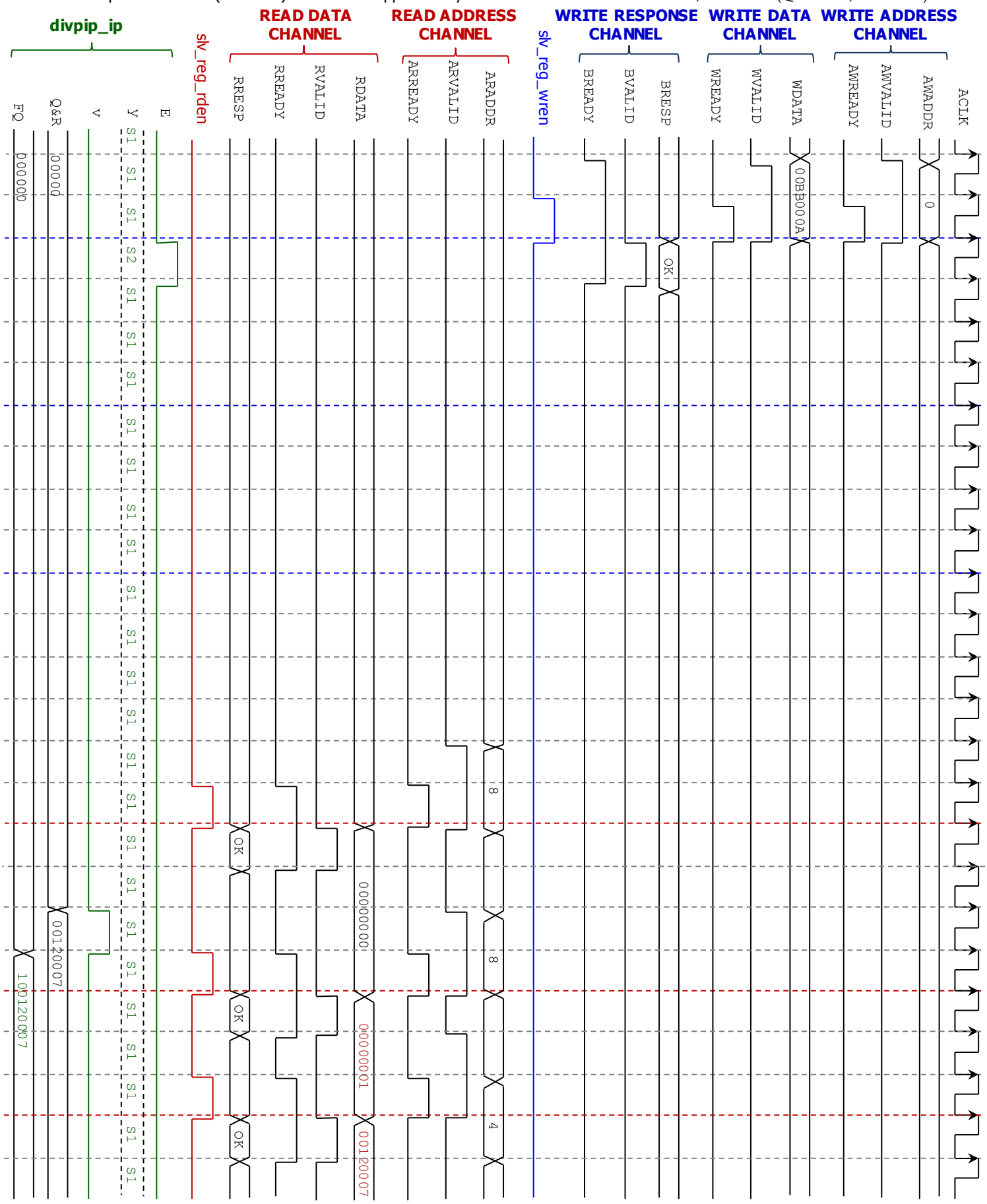




- **Software routine**: It writes one 32-bit word, and then reads two 32-bit words. One of these output words includes the signal v. If v=1, the software routine can start a new computation (write a 32-bit word, read two 32-bit words).
  * In most cases, by the time we read data, the circuit has already computed its result, but it is good practice to verify this.

- **Pipelined Circuits and AXI4-Lite**: The AXI4-Lite interface can only write/read a 32-bit word per bus transaction. As a result, we cannot take advantage of the pipelined behavior (E is always a one-cycle pulse).
  If we were to include a FIFO instead of the Register, we could write a chunk of 32-bit words first, then retrieve a chunk of output data. But there is no advantage over using a Register as AXI4-Lite does not support writing 32-bit words continuously.

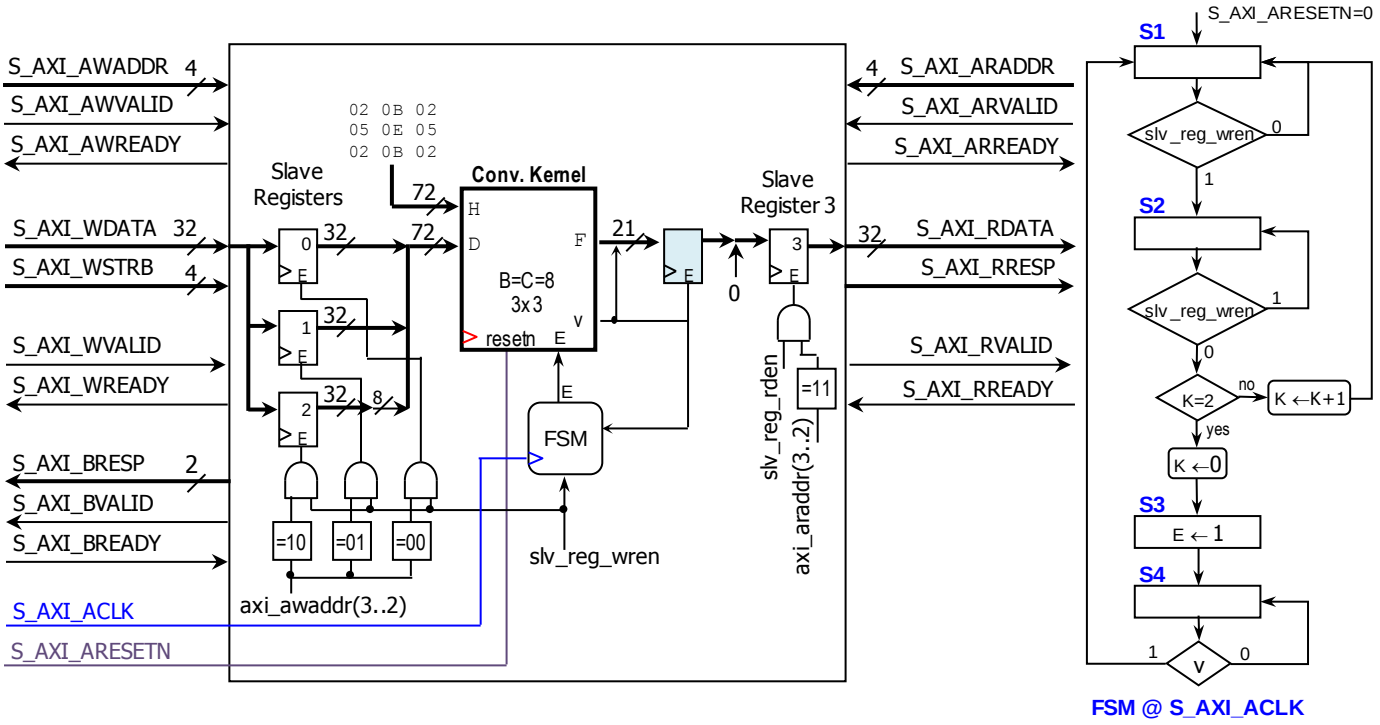
- **AXI4-Lite Interface – Timing Diagram**: (we first read from register 2 first to find out done=1, then from register1)
  - ✓ Pipelined Divider (M=N=16). The results appear 16 cycles after E is asserted. A=0x00BB, B=0x000A (Q=0x0012, R=0x0007).



## AXI4-LITE: PIPELINED 2D CONVOLUTION KERNEL

- **Custom Hardware Peripheral:** Pipelined 2D Convolution Kernel (N=3, B=C=8)
  - ✓ Operation: This circuits captures reads input data (72-bit D) when the E (enable) signal is asserted. After a processing delay, the result (20-bit F, v) appears and it is signaled by v=1. As this circuit is pipelined, we can continuously feed and retrieve data (no need to wait until v=1 to feed a new input data set).
  - ✓ Output data is valid only when v=1. v is a delayed version of E. If E is only asserted for one cycle, then when the division operation completes, v will only be asserted for one cycle, i.e., data will only be valid for one clock cycle.

- **AXI4-Lite Interface:**
  - ✓ Simple interface with 4 Slave Registers for reading and writing, a Register, and an FSM.
    - 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. $axi\_araddr\ (3..2) = 11$.
  - ✓ Extra Register: Buffer that stores the output results (when v=1). Pipelined circuits update the output when a new input data set is captured. Thus, the output results must be stored in the buffer until they are read by the AXI4-Lite interface.
  - ✓ Input Data: 72 bits (D). Slave Register 0, Slave Register 1, Slave Register 2 contain D[71..40], D[39..8], D[7..0] respectively
  - ✓ Output Data: 21 bits (F, v). The signal v is included to verify that i) the output data was captured when v=1, and ii) the circuit is working (v was issued). Output data is captured in the buffer when v=1. As the output interface is 32-bits wide, the 21 bits need to be zero-padded to 32 bits.
  - ✓ FSM: It captures 72 bits (three 32-bit words) on the Slave Registers (it detects 3 pulses of $slv\_reg\_wren$). Once the 72 bits are present on D ready for processing, the FSM issues E=1 for a clock cycle. It then waits for v=1 before returning to S1 to start over a new computation.




- **Software routine**: It writes three 32-bit words, and then reads a 32-bit word. The output word includes the signal v. If v=1, the software routine can start a new computation.
  * In most cases, by the time we read data, the circuit has already computed its result, but it is good practice to verify this.
- **Pipelined Circuits and AXI4-Lite**: The AXI4-Lite interface can only write/read a 32-bit word per bus transaction. As a result, we cannot take advantage of the pipelined behavior.

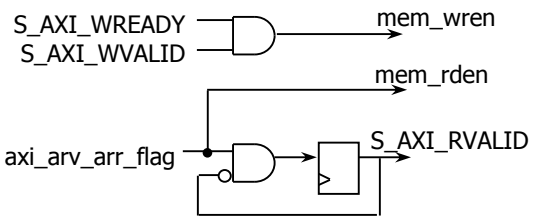
### PIPELINED VS. ITERATIVE CIRCUITS

- Bear in mind how their behavior differs (iterative: s and done, pipelined: E and v). In an iterative circuit, we cannot feed new data until the previous operation has completed (done=1). In a pipelined circuit, we can feed new data continuously.
- In all the examples shown, we mentioned that it is good practice to include a Register buffer. This is especially required for the interfaces for the Pipelined circuits. For the iterative circuits, it is strongly recommended.
- Pipelined circuits: the AXI4-Lite interface cannot feed data continuously. Thus, they are better suited for AXI4-Full interfaces.

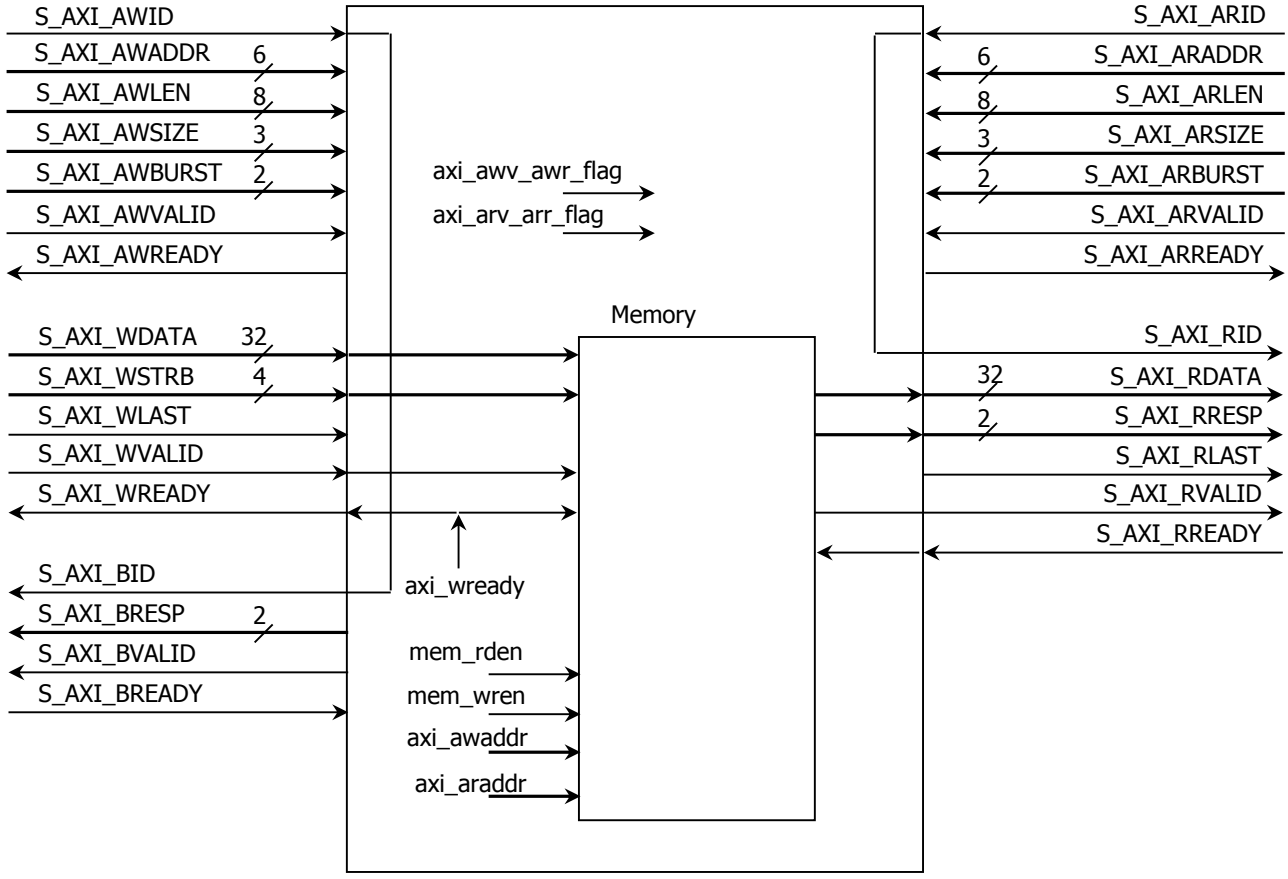
# AXI4-FULL INTERFACE - EXAMPLES

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

- Hardware Peripheral: 64-byte memory (or 16 32-bit word memory). Data Width: 32 bits.
- Address (S_AXI_AWADDR, S_AXI_ARADDR): These signals are different from the latched addresses $axi\_awaddr$, $axi\_araddr$ in AXI4-Lite. Vivado 2019.1 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 $S\_AXI\_AWADDR, S\_AXI\_ARADDR$.
- The Xilinx template provides the following signals (depicted in the figure below):
  - ✓ $axi\_awv\_awr\_flag$: 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. $mem\_rden = axi\_arv\_arr\_flag$.

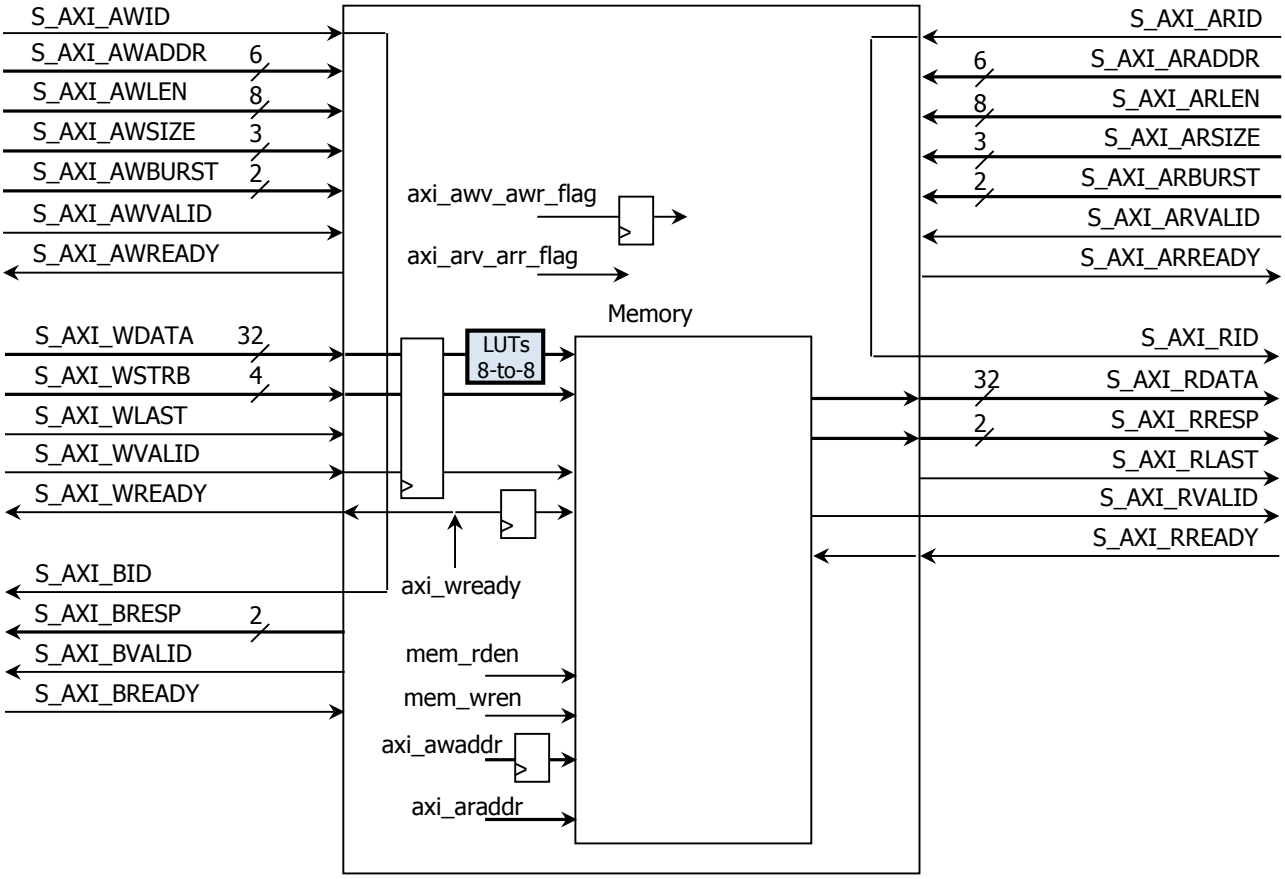



- Reading bursts (according to timing diagram obtained by simulating Vivado template): this particular circuit can only output one word every two cycles.
- **Burst:** This is configured by: i) $S\_AXI\_AWSIZE$ and $S\_AXI\_ARSIZE$ (Data width per burst), ii) $S\_AXI\_AWBURST$ and $S\_AXI\_ARBURST$ (Burst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



## AXI4-FULL: XILINX TEMPLATE (MEMORY) WITH PIXEL PROCESSOR

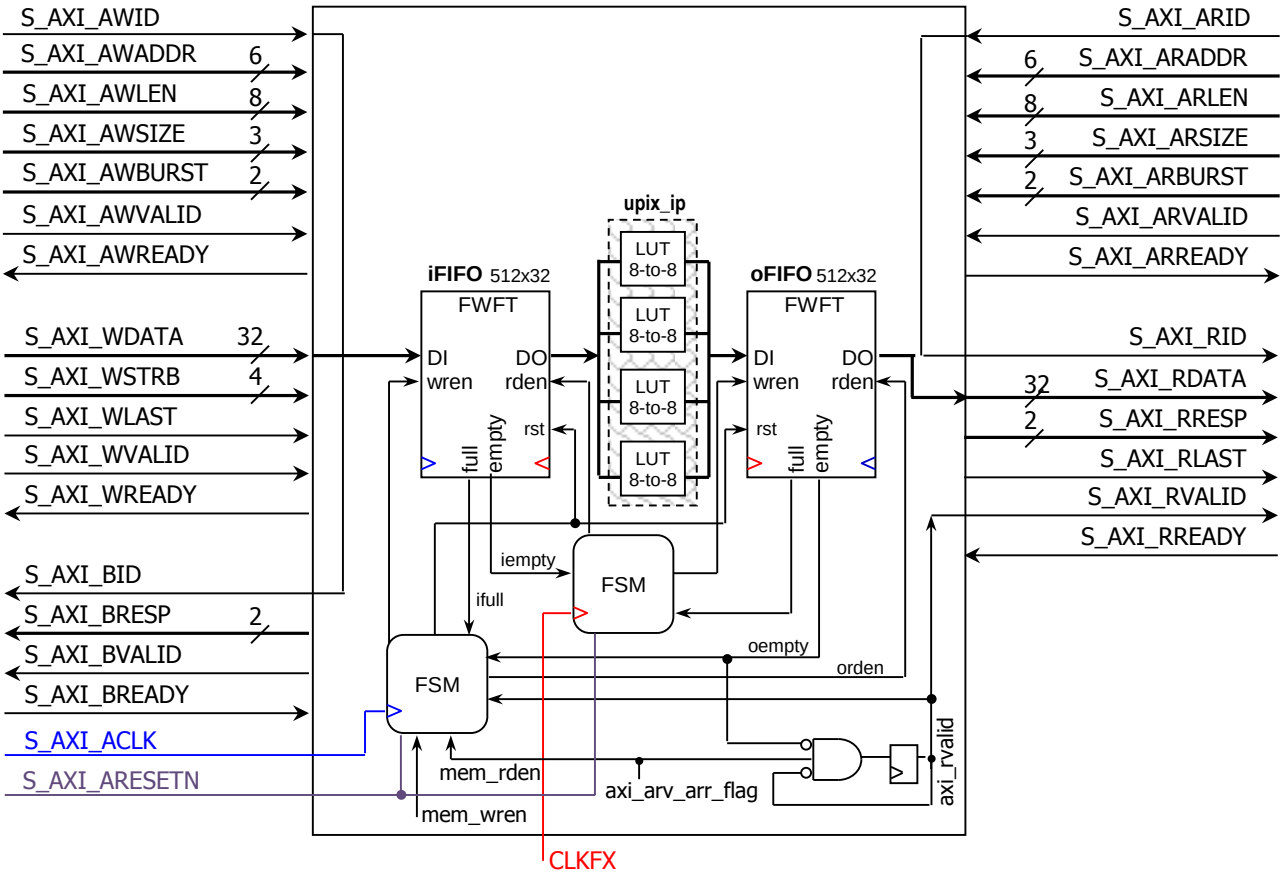
- We use the previous 64-byte memory, but we add a pixel processor unit of 32 bits (four LUT 8-to-8). Due to the LUT delay most incoming signals to the Write Address and Write Channel (as well as some internal signals) are delayed using a register.




- This approach requires significant modification to work with more complex hardware components that take several cycles to compute data. As a result, we modified the memory-based approach and instead we use FIFOs to communicate with the AXI4-Full I/O signals. We still keep most of the AXI4-Full interface signals generation though.

## 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 direct control to tell the software that the iFIFO is full: the AXI Peripheral will behave 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 oFIFO is empty (oempty=1), the signal RVALID is forced to 0. So, the Master will wait until data is available in oFIFO. This might lead to software deadlock. A more sophisticated approach might require the software 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.



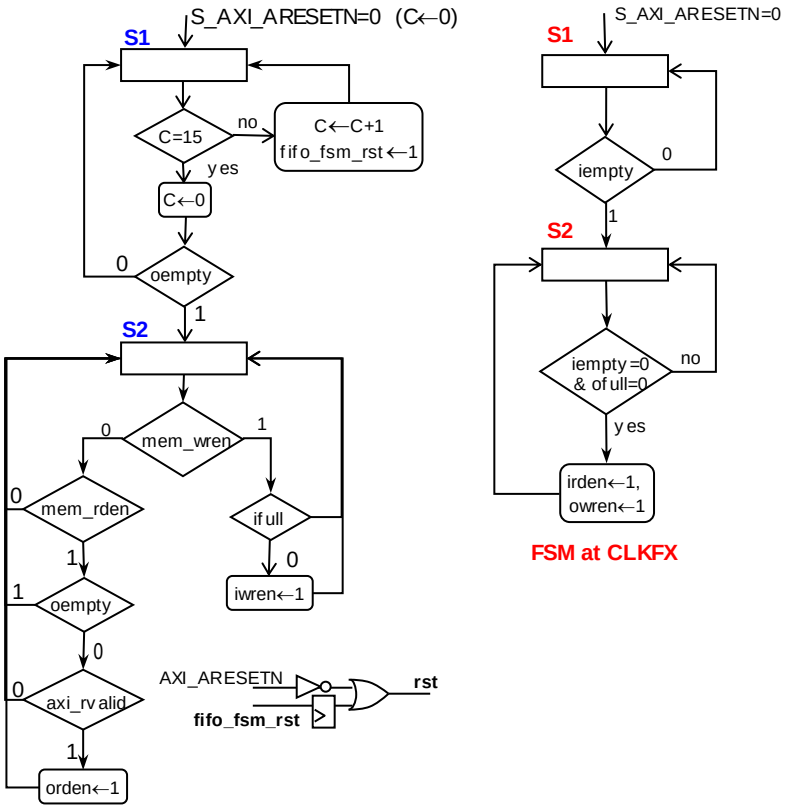

- **Input/Output Example:** If we input one 32-bit word, we get one 32-bit output word.

| 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.
  - ✓ $reset$: This active-high signal is generated in order to fulfill the requirements of the FIFOs' reset.
    - FIFOs must 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.

- $resetn$ signal of the FSM @ CLKFX: We connect it to the active-low AXI bus reset.



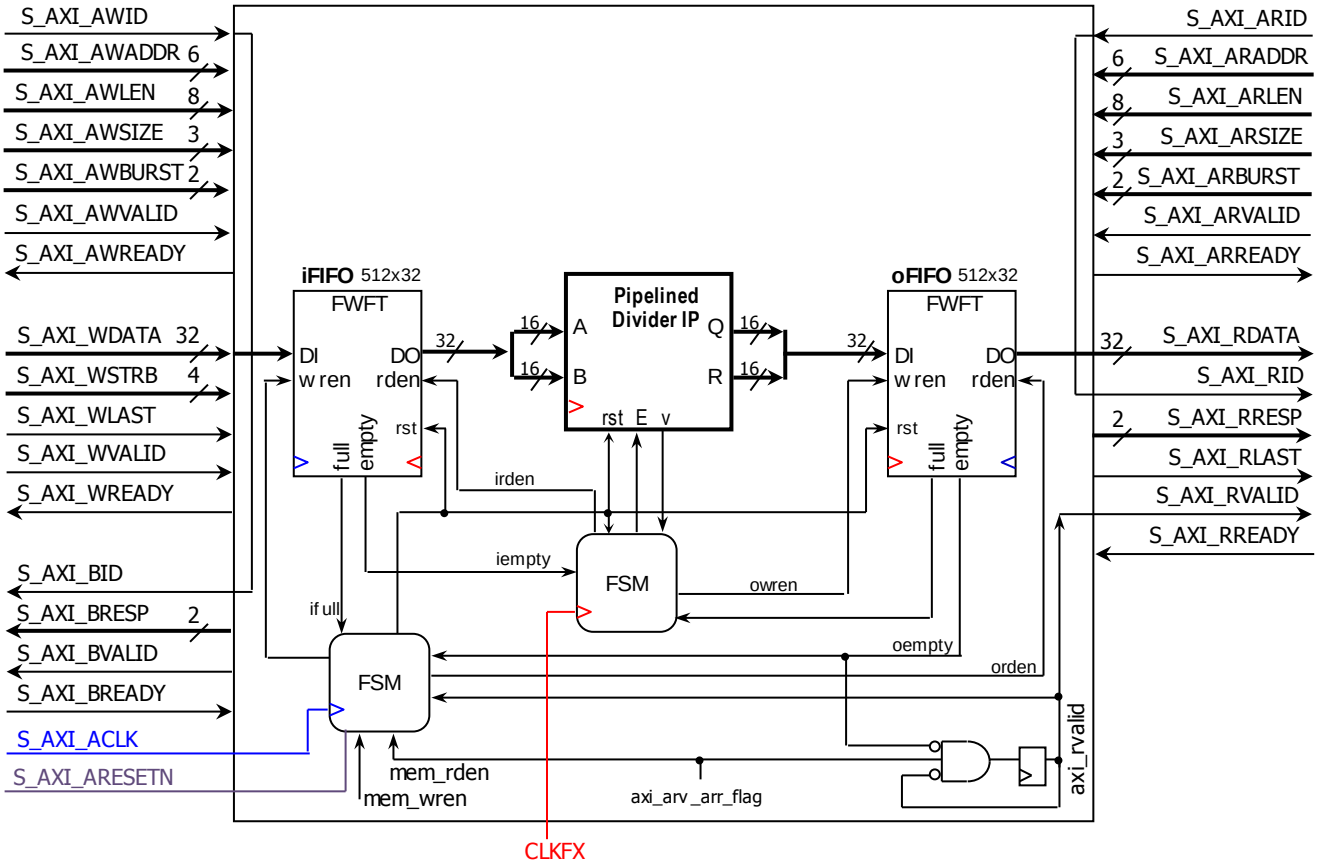

FSM at S_AXI_ACLK

FSM at CLKFX

- **Template:** You might 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. However, unlike this Pixel Processor, circuits are usually synchronous (requiring a clock) and/or usually require glue logic between the hardware architecture and iFIFO output and oFIFO input. The Pipelined Divider and the Pipelined 2D Convolution Kernel show such cases.

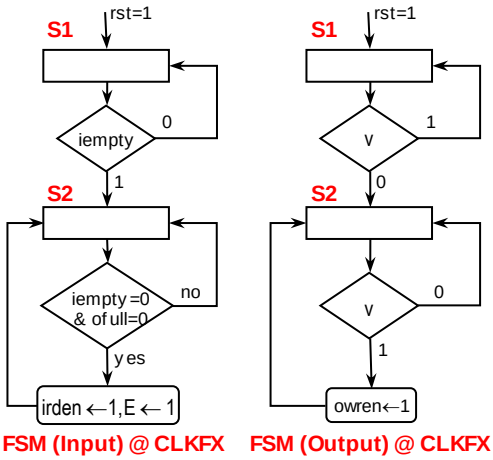
## AXI4-FULL: PIPELINED DIVIDER WITH FIFO INTERFACE

- This design illustrates how to integrate a Pipelined Divider (N=16, M=16) into the AXI4-Full interface.
- **Pipelined Divider IP**: The figure depicts the I/Os: data signals and the control signals (reset, enable, and valid).
  - ✓ Inputs: 16-bit data inputs (A, B), and enable input. Outputs: 16-bit data outputs (Q, R) and valid output.
  - ✓ The pipelined divider captures data (A, B) when E=1. After a processing delay, output (Q, R) appears and it is signaled by v=1. As this circuit is pipelined, we can continuously feed and retrieve data. The AXI4-Full Interface can handle this.
- The interface varies slightly from the one for the pixel processor. In addition to the FSM @ CLKFX, the pipelined divider (a synchronous circuit) circuit also runs at CLKFX. Finally, the FSM @ CLKFX is different.



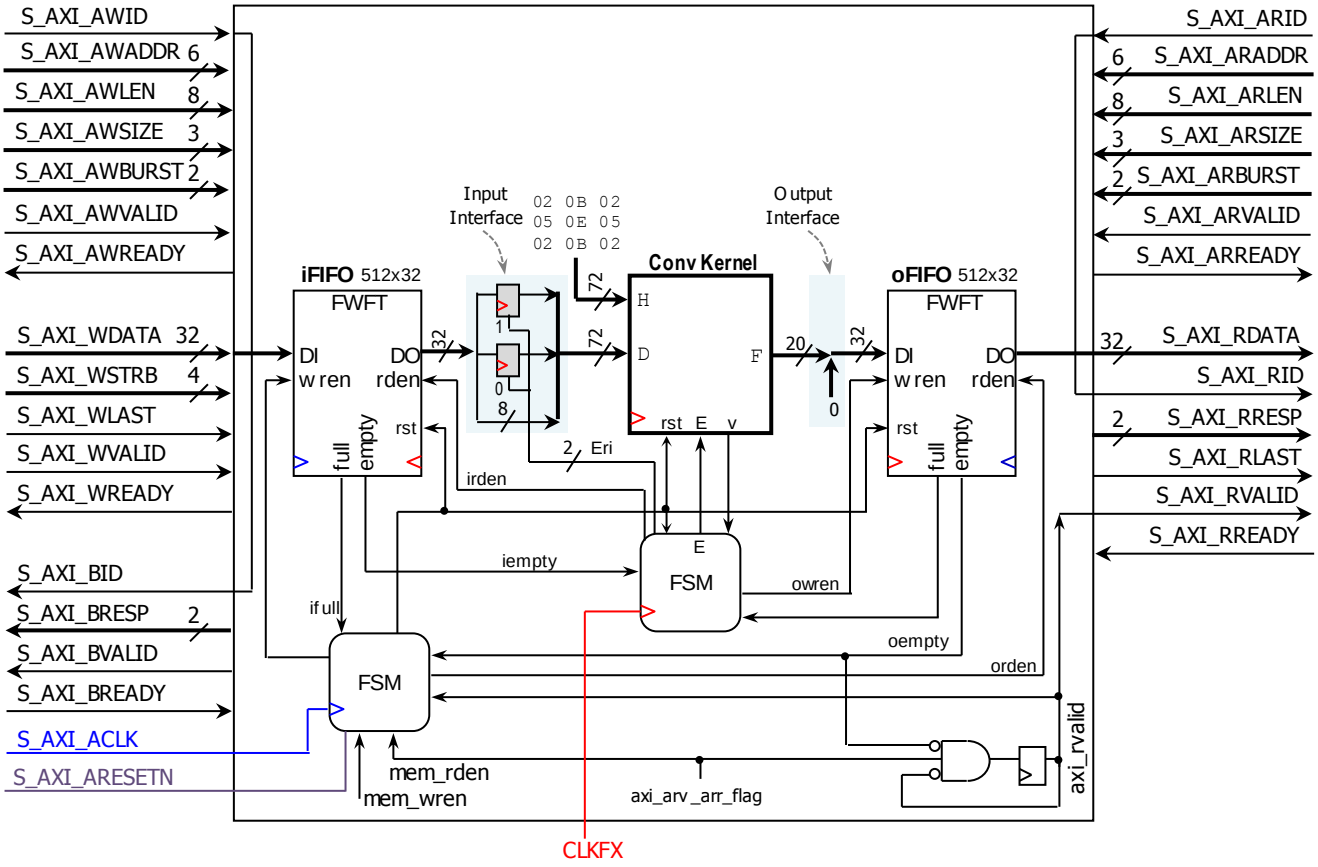

- **FSM @ S_AXI_ACLK**
  - ✓ This is the same FSM as the one for the Pixel Processor. It generates the active-high signal *reset* for the FIFOs.
  - ✓ *reset*: Note that it is also fed to the FSM @ CLKFX, Pipelined Divider IP. Though we could have used the active-low AXI bus reset (S_AXI_ARESETN) for these components, we instead connected them to this active-high reset. This configuration will prove very useful if we want to later perform Partial Reconfiguration (Unit 6).

- **FSM @ CLKFX**:
  - ✓ This FSM handles:
    - The inner side of the FIFOs (iFIFO: *iempty, irden*, oFIFO: *ofull, owren*). The FSM checks whether iFIFO is not empty and oFIFO is not full, before attempting to write data on the Pipelined Divider.
    - Control signals of: i) the Pipelined Divider IP (E, v), and ii) control signals of the interface between the FIFOs and the Pipelined Divider (not required, as there are direct connections).
  - ✓ To be able to continuously feed and retrieve data in the Pipelined Divider, we need to control the input and output sides simultaneously. Thus, we partition the FSM @ CLKFX into:
    - Input FSM: It controls the iFIFO and the input E. It resembles the FSM @ CLKFX of the Pixel processor, but it also asserts E=1 (Divider grabs data) when there is data on iFIFO and oFIFO is not full.
    - Output FSM: It controls oFIFO and the output v. Note that *owren* only depends on v. Also, we could have *owren* = v, but just in case at power-up we check that v in fact transitions from 0 to 1.



## AXI4-FULL: PIPELINED 2D CONVOLUTION KERNEL WITH FIFO INTERFACE

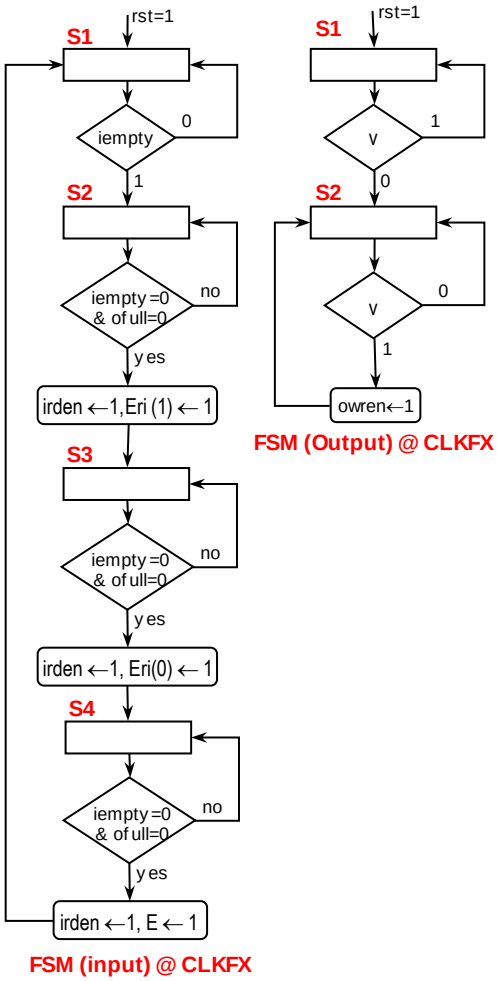
- This design illustrates how to integrate a Pipelined 2D Convolution Kernel (N=3, B=C=8) into the AXI4-Full interface.
- **Pipelined 2D Convolution Kernel IP**: The figure depicts the I/Os: data signals, control signals (reset, enable, and valid).
  - ✓ Inputs: 72-bit data input (D) and enable input. Outputs: 20-bit data outputs (F) and valid output.
  - ✓ The pipelined divider captures data (D) when E=1. After a processing delay, output (F) appears and it is signaled by v=1. As this circuit is pipelined, we can continuously feed and retrieve data. The AXI4-Full Interface can handle this.
- The interface varies from the one for the pipelined divider. In addition to the FSM @ CLKFX being more complex, we have an Input Interface and Output Interface (trivial) between the 2D Conv Kernel and the FIFOs. They all run at CLKFX.




- **FSM @ S_AXI_ACLK**
  - ✓ This is the same FSM as the one for the Pixel Processor and Pipelined Divider. It controls the outer side of the FIFOs and some AXI signals. It also generates the active-high signal *reset* for the FIFOs.
  - ✓ *reset*: Note that it is also fed to the FSM @ CLKFX and Pipelined 2D Convolution Kernel IP. Though we could have used the active-low AXI bus reset (S_AXI_ARESETN) for these components, we instead connected them to this active-high reset. This configuration will prove very useful if we want to later perform Partial Reconfiguration (Unit 6).

- **FSM @ CLKFX**:
  - ✓ This FSM handles:
    - The inner side of the FIFOs (iFIFO: *iempty, irden*, oFIFO: *ofull, owren*). It checks whether iFIFO is not empty and oFIFO is not full, before attempting to write data on the 2D Convolution Kernel.
    - Control signals of the 2D Conv Kernelr IP (E, v), and the interface between the FIFOs and the 2D Conv. Kernel (Eri).
  - ✓ To be able to continuously feed and retrieve data in the Pipelined Divider, we need to control the input and output sides simultaneously. Thus, we partition the FSM @ CLKFX into:
    - Input FSM: It controls the iFIFO and the input E. It is more complex than that of the Pipelined Divider: it loads 3 32-bit words, from which 72 bits are fed into the input D. When the 72 bits are ready, it asserts E=1 when there is data on iFIFO and oFIFO is not full.
    - Output FSM: It controls oFIFO and the output v. Note that *owren* only depends on v. Also, we could have *owren* = v, but just in case at power-up we check that v in fact transitions from 0 to 1.

- **IMPORTANT DIFFERENCE WITH AXI4-LITE**: Note that in the 'pipelined divider' and 'pipelined 2D Convolution kernel', we did not include the signal v in the output. Data is written on oFIFO only when v=1, otherwise oFIFO is empty. When we request a read, only valid data will be retrieved. This is a powerful feature of the FIFO-based approach for AXI4-Full.




**TIPS: AXI4-Full interface files**: If you call your interface 'my_intf', Vivado 2019.1 creates the following template files:

- <my intf>_v1_0.vhd: top file of the interface. No need to edit unless you plan to include extra I/Os in the interface.
- <my intf>_v1_0_S00.AXI.vhd: This file implements most of the AXI4-Full interfacing and includes a 64-byte memory example. In our examples, we made some significant modifications: we removed the 64-byte memory and instantiate the files myAXI_IP.vhd (wrapper file) and my_AXIfifo.vhd: this file implements the 2 FIFOs so that your hardware design (called 'my core') connects to them; it also slightly modifies AXI_RVALID generation (compared to the original Xilinx template).

  ✓ **Suggestion**: If your circuit is called 'my core', create a file on top of it, called 'my_core_ip' where you will include 'my_core' and the glue logic (e.g.: registers, FSM) required to connect 'my_core' to my_AXIfifo.vhd.

  ✓ **Example**: AXI4-Full interface called mypipdivfull for Pip. Divider (res_div_pip.vhd) design. The file hierarchy is:
  mypipdivfull_v1_0.vhd: Top file. No need to edit it.
  mypipdivfull_v1_0_S00_AXI.vhd: Edited template without the 64-byte memory example.
  myAXI_IP.vhd: Wrapper file for the circuit that implements the FIFOs and their connections.
  myAXIfifo.vhd: File that implements the FIFOs. It also interfaces with some Write Channel and Read Channel signals (AXI_RVALID generation is slightly different). It implements the FSM@S_AXI_ACLK. Here, you need to instantiate pipdiv_ip.vhd and connect it to the FIFO I/Os and its control signals (ofull, iempty, owren, irden).
  pipdiv_ip.vhd: This is the circuit you need to build. Instantiate res_div_pip and include the glue logic (FSM@CLKFX, buffer register) to the FIFOs (inner side and the control signals).
  res_div_pip.vhd: This is your circuit and it includes any other components (.vhd) and ancillary files.
  … (*extra files required for* res_div_pip.vhd)

  - **Note**: This is the file structure you should follow for any design, i.e., use this Pipelined Divider as a template. All of the examples here follow this structure. The only exception is the Pixel Processor where the FSM@CLKFX was included in myAXIfifo.vhd (for legacy reasons).

## GENERAL PROCEDURE FOR INTERFACE DEVELOPMENT (AXI4-LITE, AXI4-FULL) FOR CUSTOM HARDWARE

- Design your hardware architecture, called 'Core IP' (e.g. Pipelined Convolution Kernel, iterative CORDIC).
  - ✓ Perform thorough simulation.
  - ✓ The I/O mechanism should be clearly specified. For example: s and done for iterative circuits, E and v for pipelined circuits. Note that there can be other I/O mechanisms: a more complex variation of the examples shown so far, or very different ones. This I/O mechanism is what really matters when designing the AXI interface.
- **Design the AXI Peripheral**: Build an AXI interface around your 'Core IP'.
  - ✓ AXI Peripheral: 'Core IP' + the AXI interface around it. Vivado provides template files:
    - AXI4-Lite: The Vivado template is very helpful as it already includes the circuitry to interface to AXI signals, and it provides Slave Registers and extra control signals (e.g.: slv_reg_wren, slv_reg_rden) to which your 'Core IP' can connect to. Your job is to connect your 'Core IP' to these Slave Registers and to the extra control signals.
    - AXI4-Full: We made significant modifications to the Vivado template and turned it into a FIFO-based interface. Your job is to connect your 'Core IP' to two FIFOs (input FIFO: iFIFO, and output FIFO: oFIFO) and their control signals.
  - ✓ This approach of only 'interfacing' the 'Core IP' to Slave Register signals (AXI4-Lite) or FIFO signals (AXI4-Full) is well recommended as it avoids (for the most part) having to deal with the AXI interface circuitry. Eventually, the AXI bus protocol might be updated or replaced. By following this approach, we might keep our design and their interface to Slave registers (AXI4-Lite) or FIFOs (AXI4-Full) while only replacing the bus interface circuitry. Several examples shown here were initially built for the old PLB Bus; minor changes were made in order to turn the old peripherals into AXI peripherals.
  - ✓ In some applications, you might need extra features in your AXI Peripheral (required by the 'Core IP' or the AXI interface):
    - Connection to external I/Os to the PL (e.g.: switches, leds). You need to route these signals as AXI peripheral I/Os.
    - Generate interrupt signals that connect to the PS. Once you design the hardware interrupts, you need to router these interrupt signals as AXI peripheral I/Os that will connect to the PS.
  - ✓ It is strongly recommended that you create the AXI Peripheral in a separate Vivado project. To get the template files, you can Create and Package a New IP in Vivado and retrieve the template files. Or you can get the files from the examples shown (AXI4-Full/AXI4-Lite pixel processor, pipelined divider, pipelined 2D convolution kernel) and edit them.
  - ✓ Perform thorough simulation. When building the peripheral, it is not uncommon to make a design or coding mistake. These problems can be found and corrected via proper simulation (in some cases unintended issues might be discovered via simulation). Try to emulate (to the best extent) the writes/reads of your software application.
    - To facilitate this process, testbenches are included for the AXI4-Lite and AXI4-Full interfaces of these designs: pixel processor, pipelined divider, pipelined 2D Convolution kernel. These testbenches include procedures to:
      - Write/Read on Slave Registers (AXI4-Lite): `WRITE_REG(Data, Slave Reg #)`, `READ_REG(Slave Reg #)`.
      - Write/Read on memory addresses (AXI4-Full): `WRITE_DATA(Mem. Address, # of words, Data)`, `READ_DATA (Mem. Address, # of words)`. In our FIFO-based approach, the memory address you write/read from is usually not relevant, but it might be in some special circumstances (e.g.: see PL interrupt example later).
    - You can re-use any of these testbenches in your design with minor modifications:
      - Update the name of the testbench.
      - Update the name of the Unit under test (UUT) component declaration and port mapping.
      - Update writes/reads to Slave Registers (AXI4-Lite) or memory positions (AXI4-Full) as needed by your application.

- Build the AXI4 Peripheral (Full or Lite) in Vivado. This mechanical procedure packages your peripheral so it is ready to drag and drop it in a Block Based Design in Vivado. Refer to Embedded System Design for Zynq PSoC Tutorial → Units 3 and 4.
  - ✓ If your peripheral includes I/Os, refer to Zynq Book Tutorial → Creating IP in VHDL for step-by-step instructions.
  - ✓ If your peripheral includes interrupt signals that connect to the PS, refer to Embedded System Design for Zynq PSoC Tutorial → Unit 9.
- Create the embedded system (Block Based Design): Drag and drop the Zynq PS and your custom AXI peripheral. Refer to Embedded System Design for Zynq PSoC Tutorial → Units 3 and 4 for step-by-step instructions.
- **Build software application in SDK**. When you open SDK (after building your embedded system in Vivado), software drivers will be generated for your custom AXI peripheral (among other drivers for PS/PL peripherals).
  - ✓ If the name of your peripheral is 'my_intf', the software drivers are listed in 'my_intf.h'. The available functions are based on low-level functions that write/read in the memory space (Xil_Out32, Xil_In32):
    - AXI4-Lite functions: MY_INTF_mWriteReg(base address, SlvReg #, 32-bit data), MY_INTF_mReadReg (base address)
    - AXI4-Full functions: MY_INTF_mWriteMemory(base address + offset, 32-bit data), MY_INTF_mReadMemory (base address + offset). In our FIFO-based approach, we can write a large chunk of data before we retrieve any results (we have up to 512 32-big input and output words). We will see later how to write/read chunks of data in bursts via DMA.
  - ✓ Our AXI Peripheral has a register-based interface: we write/read onto specific registers (AXI4-Lite) or Memory Addresses (AXI4-Full) in order to communicate with our hardware design.
  - ✓ Note: Each address in a microprocessor addresses a byte. However, each word we write/read is 32 bits. Thus:
    - AXI4-Lite: The Address width of awaddr, araddr is 4 bits if we use 4 Slave Registers (i.e., 4 32-bit words). Thus, Register 1 is address 0100, Register 3 is address 1100 (a slave register is viewed as a memory address).
    - AXI4-Full: If we use the 64-byte (or 16 32-bit words) memory template (that was converted in FIFOs), the address width of awaddr and araddr is 6 bits. So, we have 16 32-bit memory positions to write/read.

## AXI4-FULL: 2D-DCT FIFO INTERFACE

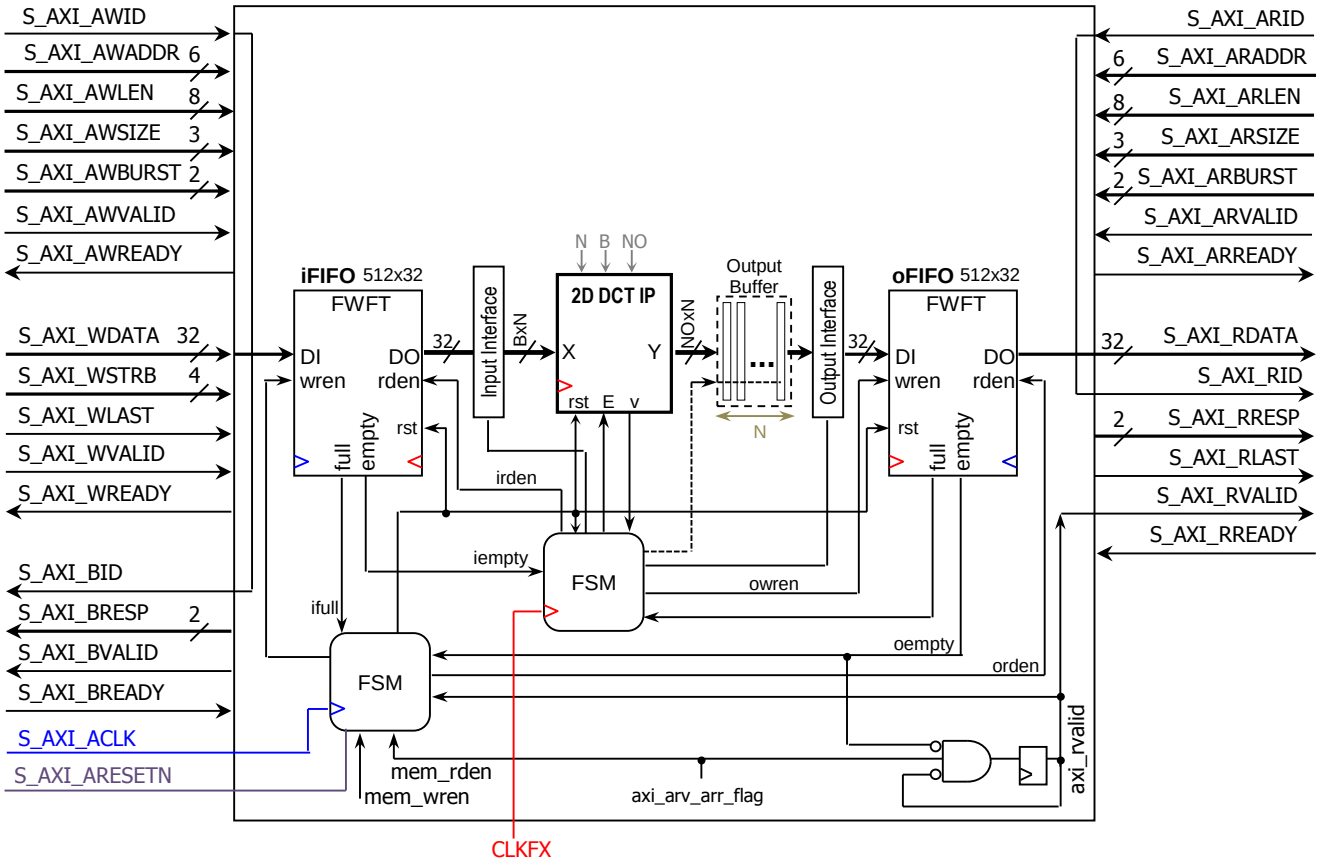
- This design illustrates how to integrate a complex system (2D DCT IP) into the AXI4-Full interface. Unlike the previous examples, here we intend to design a parametric interface that supports different parameters of the 2D-DCT IP.
- **2D DCT IP**: The figure depicts the input and output data signals, and the control signals (reset, enable, and valid), and the three most important parameters (N, B, NO).
  - ✓ Inputs: N×N B-bit pixels. Data is fed column-wise (N B-bit pixels at a time). To feed a column, E must be asserted.
  - ✓ Outputs: N×N pixels, each pixel of NO bits. Data is generated row-wise (N NO-bit pixels at a time). 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).

- This interface is quite different than the previous ones as it is parametric (it depends on N, B, NO). The glue logic between the 2D-DCT and the FIFOs varies according to the parameters. The 2D-DCT architecture and the glue logic runs at CLKFX. Finally, the FSM @ CLKFX is large to handle the complex I/O mechanism.

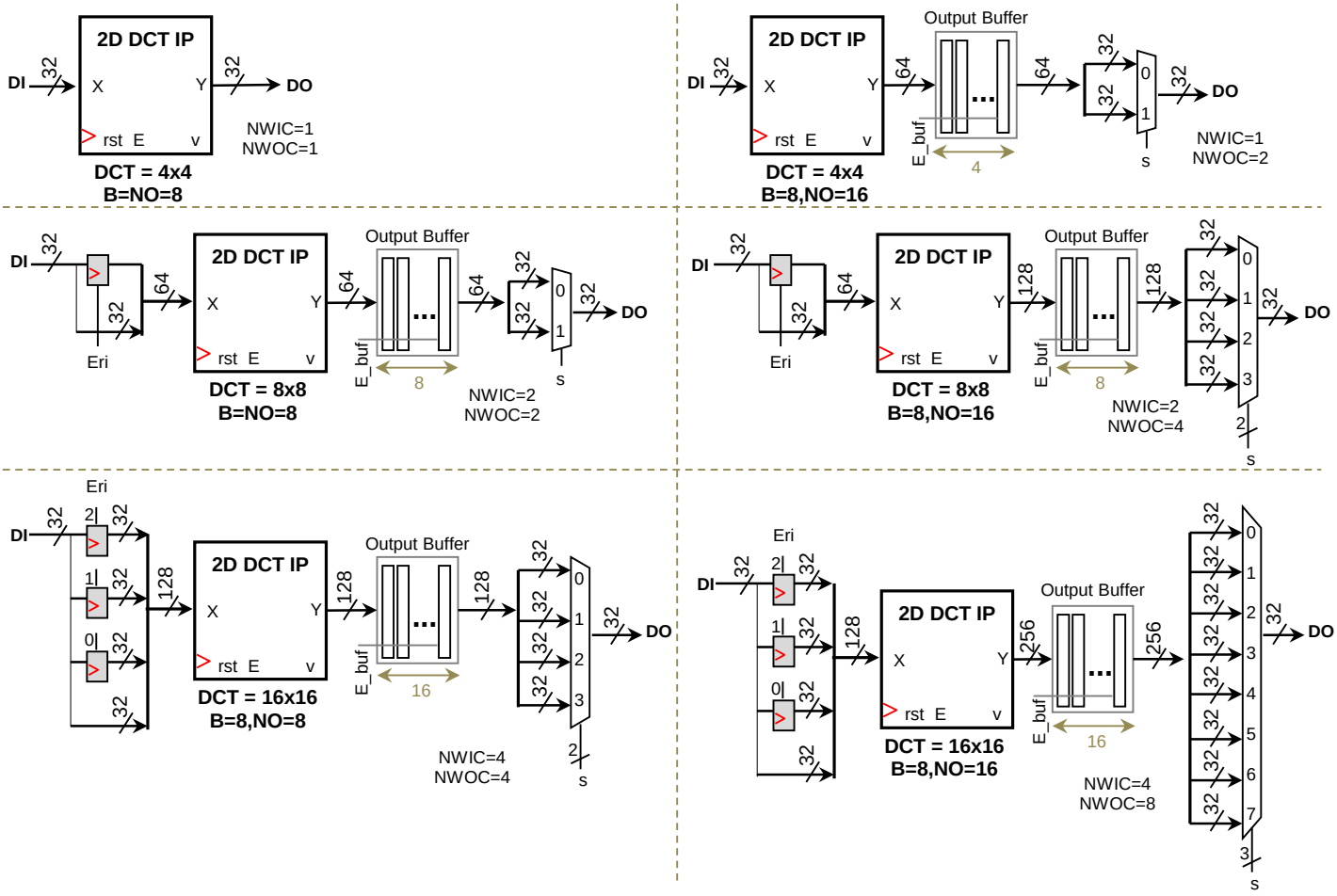
- **Glue logic between 2D-DCT and FIFOs**:
  - ✓ Input Interface: Collection of registers that capture data 32 bits at a time. The 2D DCT data input is N×N×B bits (B=8, N=4, 8, 16). We feed one column at a time, i.e., we have N groups of N×B bits.
  - ✓ Output Buffer: The 2D DCT generates N groups of N×NO bits in successive cycles. If N×NO>32, we cannot place this amount of data fast enough on oFIFO. Here, we need a temporal buffer to store this data.
  - ✓ Output Interface: When N×NO>32, a MUX is needed so we can place data on oFIFO 32 bits at a time.
  - ✓ $NWIC = \frac{N}{\lceil 32/B \rceil}$. This is the number of 32-bit words per input column
  - ✓ $NWOC = NWIC \times \left\lceil \frac{NO}{B} \right\rceil$. This is the number of 32-bit words per output column (or row).
  - ✓ The following table displays the values of NWIC and NWOC for the supported DCT size (N) 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 active-low 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 the different Input and Output Interfaces to the 2D DCT IP core along with the Output buffer (when needed). The particular architecture depends on the parameters N, B, and NO.
  - ✓ Note that when DCT=4x4 (N=4) 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.

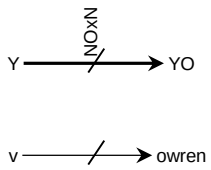



- **FSM @ S_AXI_ACLK**
  - ✓ This is the same FSM as the one for the Pixel Processor.

- **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: $s, Eri, E\_buf$.
  - ✓ To be able to control the input and output sides simultaneously, we partition the FSM @ CLKFX in two parts:
    - Input FSM: It controls the input interface, the input E to the 2D DCT and iFIFO.
    - Output FSM: It controls the output interface, output buffer, oFIFO, and the output v.

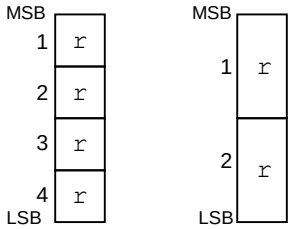
  - ✓ **Output FSM**
    The figure depicts the output interface control and $owren$ generation:
    - (a) N=4, B=NO=8: owren=v and no output buffer. This requires no Output FSM.
    - (b) Cases different N=4, B=NO=8: Output buffer and FSM that generates E_buf and owren.
    - (c) Order of output pixels in a 32-bit word (same for input pixels) and on a NO×N output row.

**DCT = 4x4, B=NO=8**




(a)

**OUTPUT PIXELS ON A 32-BIT WORD:**

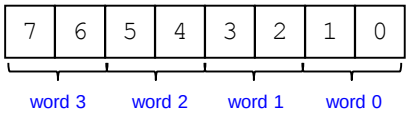



NO=8, 4 output pixels in 32-bit word

NO=16, 2 output pixels in 32-bit word

**OUTPUT PIXELS ON A NOxN ROW:**

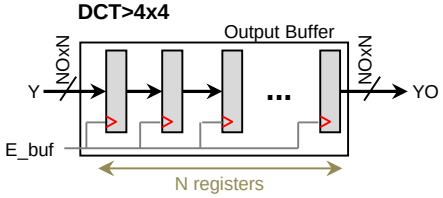
Example: Output row: NOxN=128
-> Eight output pixels, = four 32-bit words

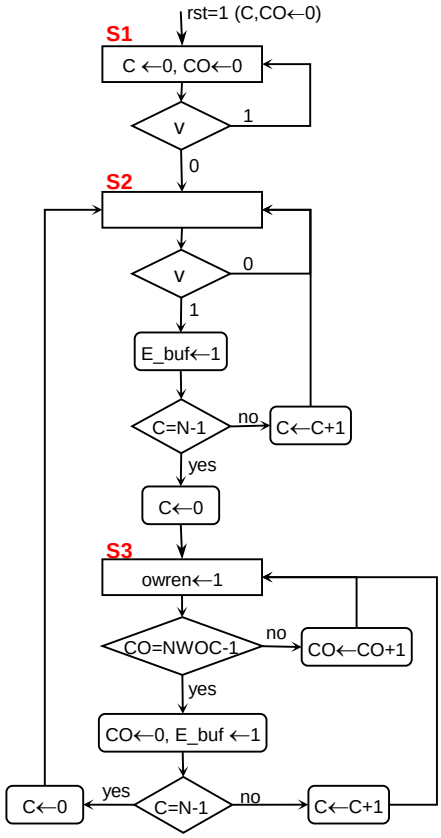



When storing on oFIFO, the order is :
*word 3, word 2, word 1, word 0*
Thus s = NWOC-1-CO

(c)

**DCT>4x4**




**FSM - DCT Peripheral (@ CLKFX)**

(b)

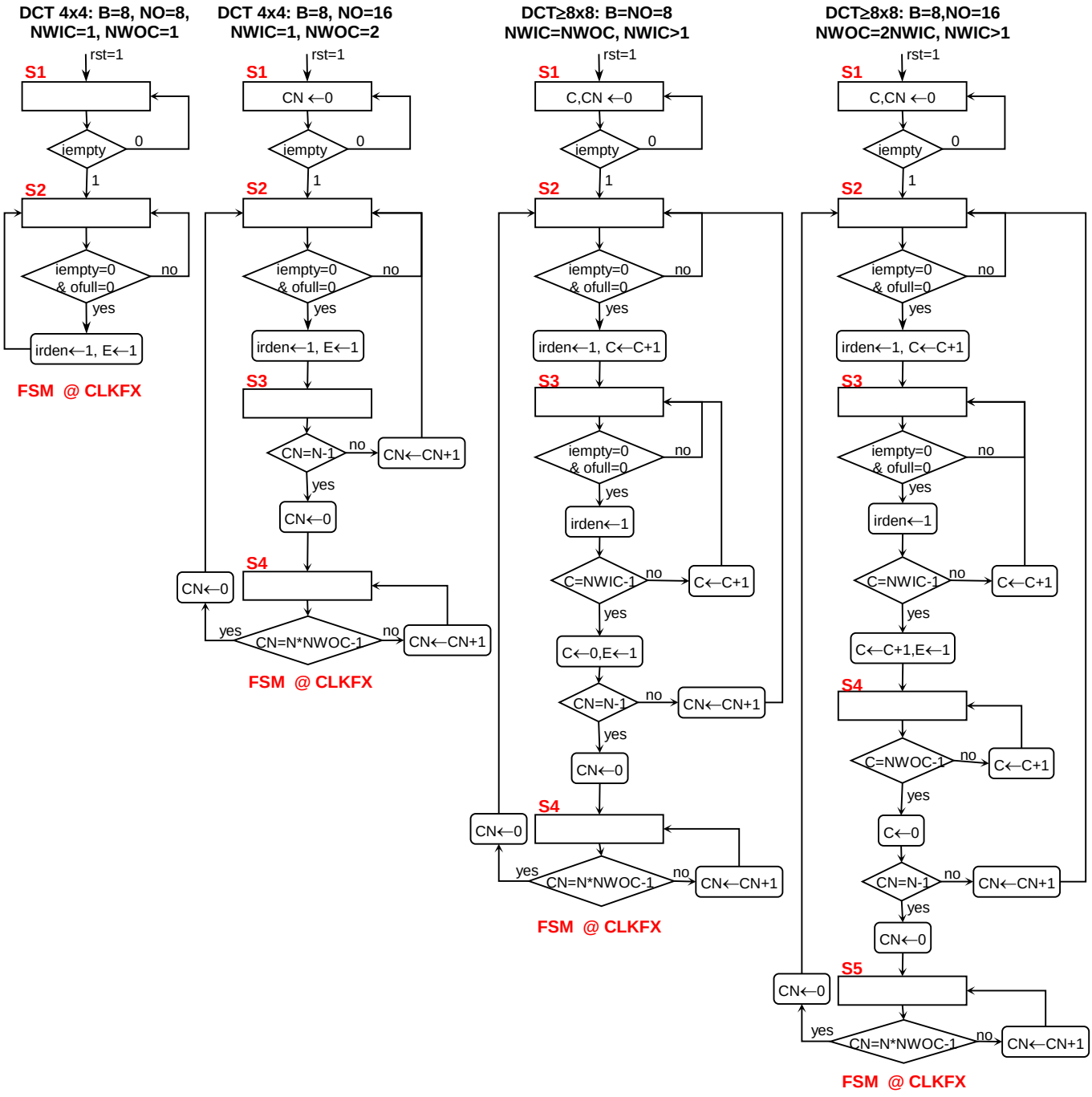
✓ **Input FSM**

- To avoid data in the output buffer to be overrun by a new input 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 |

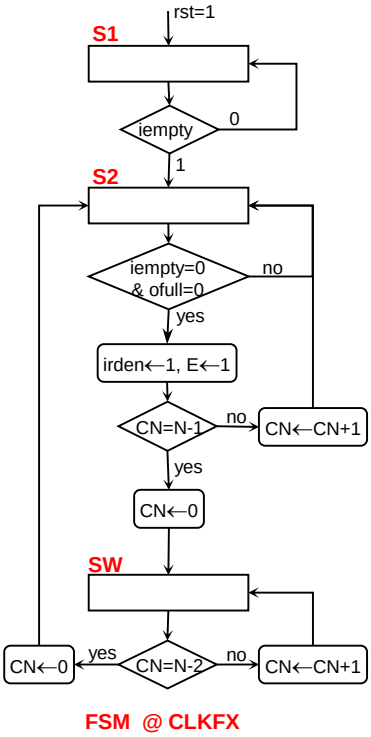
- There are two variations of the DCT 2D IP core:
  - Fully pipelined case: Selected with the parameter `IMPLEMENTATION=fullypip`. Assuming no I/O constraints, we can feed a new block to the 2D DCT IP core right after the previous one. Note that due to the FIFOs, we must wait $NxNWOC$ cycles between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



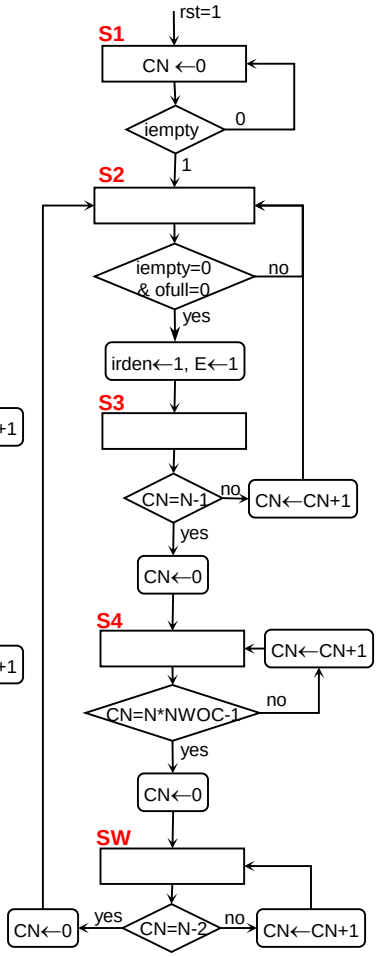
- One Transpose case: The figure depicts the case for 2D DCTs of different sizes:

**DCT 4x4: B=8, NO=8, NWIC=1, NWOC=1**



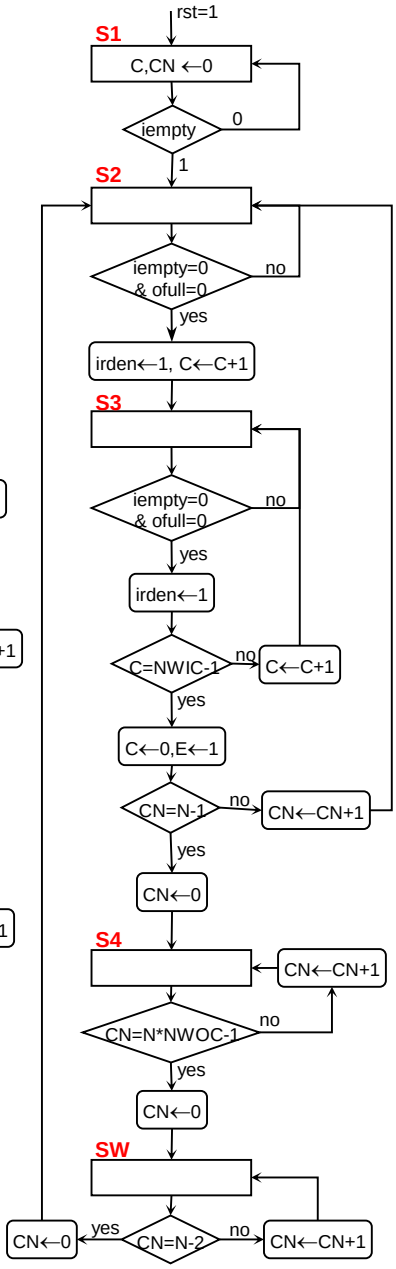

FSM @ CLKFX

**DCT 4x4: B=8, NO=16 NWIC=1, NWOC=2**



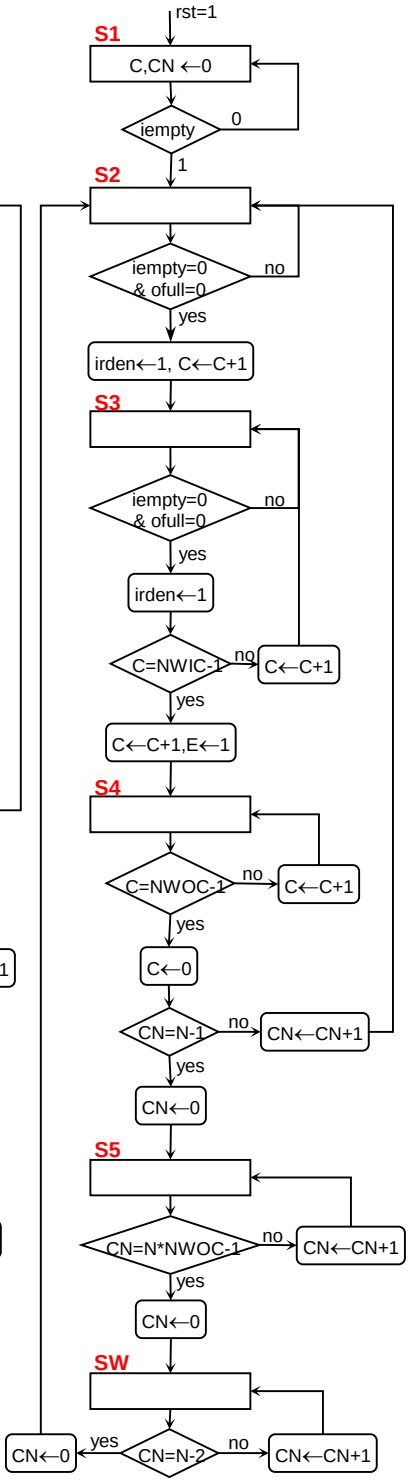

FSM @ CLKFX

**DCT≥8x8: B=NO=8 NWIC=NWOC, NWIC>1**




FSM @ CLKFX

**DCT≥8x8: B=8,NO=16 NWOC=2NWIC, NWIC>1**




FSM @ CLKFX

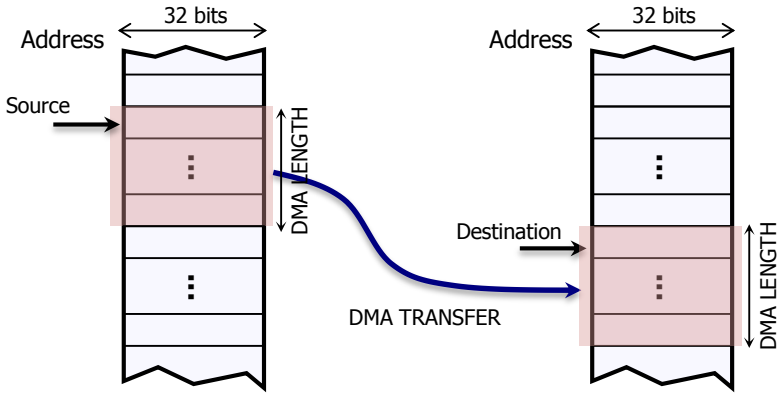
- **Input/Output Example** (N=4, B=8, NO=16)**:** The outputs have been verified (with a MATLAB model) to be correct. For the inputs, each 32 bit word is a column (top to bottom). For the output, each two 32-bit words is a row (left to right).

| 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.

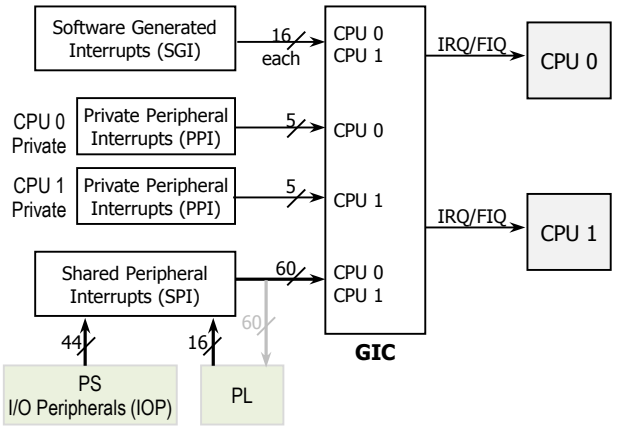
- For more information, refer to the Xilinx® Zynq-7000 AP SoC Technical Reference Manual (UG585) – Chapter 9. For a list of available functions (SDK 2016.2), look into the `xdmaps.h` file in the *bsp*: `/libsrc/dmaps_v2_1/src`.

# 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 | |

  - ✓ ***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 |

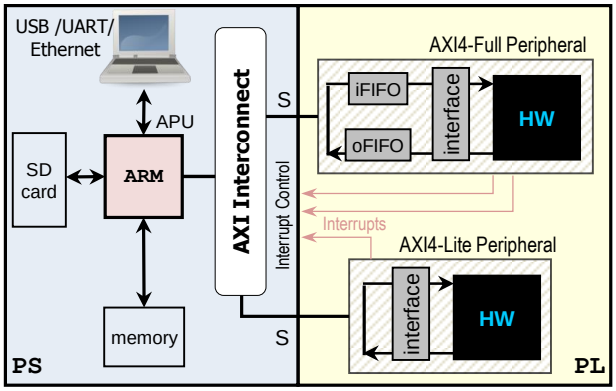
  - ✓ ***Shared peripheral Interrupts (SPI)***: 60 interrupts available. These interrupts can come from the I/O peripherals and various modules (44), or from the programmable logic (PL) side of the device (16). Note that the PL can also accept interrupts coming from the PL. They are shared between the 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`.