# Streaming Data Processing. Design of AXI4-Stream Compliant Modules

## 1 Purpose

This laboratory work introduces basic information about the behaviour of AXI4-Stream compliant hardware modules. The major objective is to describe the general mechanism based on which AXI4-Stream modules exchange data and to show how it can be implemented in order to develop such hardware components.

## 2 AXI4-Stream Internal Behaviour

As described in the previous laboratory work, AXI4-Stream protocol is used to exchange data between various hardware modules and it is extremly useful in processing data in a streaming fashion. The I/O interfaces of such modules must comply with certain rules defining a ready-valid handshake.

In case of AXI4-Stream modules which perform simple operations in just one clock cycle, the internal behaviour can be synthetically represented as a two-state Finite State Machine (FSM). In one state, the module accepts data and performs the actual operation, while in the other state it provides the result at the output.

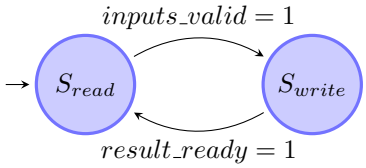


Figure 1: Internal behaviour of AXI4-Stream modules represented as a FSM

Figure 1 shows the state diagram which describs, in a simplified way, the operating mechanism of an AXI4-Stream module that is capable of performing the actual processing in a single clock cycle.

Table 1: FSM states

| State | Description | Transition condition | Next state |
|---|---|---|---|
| $S_{read}$ | The module is ready to retrieve the values received at the input and performs the actual operation. | All inputs are valid. | $S_{write}$ |
| $S_{write}$ | The module provides the result at the output and indicates that the output is valid on every output interface | All receivers are ready to accept the results provided by the module. | $S_{read}$ |

Table 1 shows the description and the transition condition for the two states of the FSM. The transition conditions can be described reffering the actual I/O signals as follows:

- $S_{read} \rightarrow S_{write}$ occurs when all **TVALID** signals of all the **input** interfaces are HIGH and the current state is $S_{read}$

- $S_{write} \rightarrow S_{read}$ occurs when all **TREADY** signals of all the **output** interfaces are HIGH and the current state is $S_{write}$

# 3 Example of AXI4-Stream Compliant Module

This section provides an example of AXI4-Stream compliant adder/subtractor, implemented in VHDL using the approach described in the previous section. The meaning of the port names is the one explained in the previous laboratory work.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

entity int_adder_subtractor is
  Port (
    aclk : IN STD_LOGIC;
    s_axis_a_tvalid : IN STD_LOGIC;
    s_axis_a_tready : OUT STD_LOGIC;
    s_axis_a_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    s_axis_b_tvalid : IN STD_LOGIC;
    s_axis_b_tready : OUT STD_LOGIC;
    s_axis_b_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    s_axis_operation_tvalid : IN STD_LOGIC;
    s_axis_operation_tready : OUT STD_LOGIC;
    s_axis_operation_tdata : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    m_axis_result_tvalid : OUT STD_LOGIC;
    m_axis_result_tready : IN STD_LOGIC;
    m_axis_result_tdata : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
  );
end int_adder_subtractor;

architecture Behavioral of int_adder_subtractor is

type state_type is (S_READ, S_WRITE);
signal state : state_type := S_READ;

signal res_valid : STD_LOGIC := '0';
signal result : STD_LOGIC_VECTOR (31 downto 0) := (others => '0');

signal a_ready, b_ready, op_ready : STD_LOGIC := '0';
signal internal_ready, external_ready, inputs_valid : STD_LOGIC := '0';

begin
    s_axis_a_tready <= external_ready;
    s_axis_b_tready <= external_ready;
    s_axis_operation_tready <= external_ready;

    internal_ready <= '1' when state = S_READ else '0';
    inputs_valid <= s_axis_a_tvalid and s_axis_b_tvalid  and s_axis_operation_tvalid;
    external_ready <= internal_ready and inputs_valid;

    m_axis_result_tvalid <= '1' when state = S_WRITE else '0';
    m_axis_result_tdata <= result;

    process(aclk)
    begin
```

```
        if rising_edge(aclk) then
            case state is
                when S_READ =>
                    if external_ready = '1' and inputs_valid = '1' then
                        if s_axis_operation_tdata = "00000000" then
                            result <= s_axis_a_tdata + s_axis_b_tdata;
                        else
                            result <= s_axis_a_tdata - s_axis_b_tdata;
                        end if;

                        state <= S_WRITE;
                    end if;

                when S_WRITE =>
                    if m_axis_result_tready = '1' then
                        state <= S_READ;
                    end if;
            end case;
        end if;
    end process;

end Behavioral;
```

The performed operation is selected based on the `s_axis_operaion_tdata` input, as described in Table 2.

Table 2: Operation selection

| `s_axis_operaion_tdata` | **Operation** |
|---|---|
| 00000000 | + |
| $\neq$ 00000000 | - |

When the value received on the input port `s_axis_operaion_tdata` is equal to `00000000`, the module performs the addition of the values received on the input ports `s_axis_a_tdata` and `s_axis_b_tdata` and when the value of the `s_axis_operaion_tdata` input signal is not equal to `00000000`, it performs the subtraction of the two values.

The most important signals are described in Table 3.

Table 3: Description of the main signals

| **Signal** | **Description** |
|---|---|
| `state` | Signal to represent the current state of the FSM |
| `inputs_valid` | Control signal which indicates that all inputs are valid |
| `internal_ready` | Control signal which indicates that the module is ready to accept data given as input (HIGH when the current state is `READ_OPERANDS`) |
| `external_ready` | Control signal which ensures that all inputs are consumed at once (HIGH when all inputs are valid and the module is ready to accept input data) |

The signals which depend only on the current state of the finite state machine are `internal_ready` and `m_axis_result_tvalid`. These signals are not influenced by the values of other signals.

- `internal_ready` is HIGH in the `S_READ` state, to indicate that the module is internally ready to accept the input data. The `external_ready` is the control signal which, additionally, ensures

that all the inputs are consumed at once. Therefore, `external_ready` is HIGH only when both `internal_ready` and `inputs_valid` are HIGH.

- `m_axis_result_tvalid` is HIGH only during the `S_WRITE` state to indicate that the output is valid.

These considerations are synthetically described in Table 4,

Table 4: State-dependent signals

| **State** | `internal_ready` | `m_axis_result_tvalid` |
|---|---|---|
| `S_READ` | HIGH | LOW |
| `S_WRITE` | LOW | HIGH |

# 4 Exercises

1. **Implement an AXI4-Stream compliant module which adjusts the value given as input so that it fits in a given range. The range limits are also provided as inputs. This module has to ensure that the output is in the given range by saturating the input value if necessary.**

   **Hints**

   - The module should be implemented as a FSM, in a similar way to the adder shown in Section 3.
   - The actual operation performed by the module can be described as follows:

     **procedure** SATURATE($val, min, max$)
       **if** $val > max$ **then**
         $res \leftarrow max$
       **else**
         **if** $val < min$ **then**
           $res \leftarrow min$
         **else**
           $res \leftarrow val$
         **end if**
       **end if**
       **return** $res$
     **end procedure**

   - The structure of the entity is shown below.

```
entity saturator is
    Port (
      aclk : IN STD_LOGIC;
      s_axis_val_tvalid : IN STD_LOGIC;
      s_axis_val_tready : OUT STD_LOGIC;
      s_axis_val_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
      s_axis_max_tvalid : IN STD_LOGIC;
      s_axis_max_tready : OUT STD_LOGIC;
      s_axis_max_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
      s_axis_min_tvalid : IN STD_LOGIC;
      s_axis_min_tready : OUT STD_LOGIC;
      s_axis_min_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
      m_axis_result_tvalid : OUT STD_LOGIC;
      m_axis_result_tready : IN STD_LOGIC;
      m_axis_result_tdata : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
```

```
    );
end saturator;
```

2. **Implement an AXI4-Stream compliant module which computes the sum of a window of values. The module gets a single value at once, then updates the internally stored window of values and provides at the output the sum of all the values in the window.**

   **Hints**

   - The window is represented internally as an array of `std_logic_vector(31 downto 0)`.
   - The size of the window should be a generic parameter of the entity.
   - The position where the new value is placed is indicated by an pointer, which is updated when a new value is read, as follows:

     **if** $ptr < window.size - 1$ **then**
     $ptr \leftarrow ptr + 1$
     **else**
     $ptr \leftarrow 0$
     **end if**

     Figure 2 shows how the pointer indicates the current position where the newest value is placed.

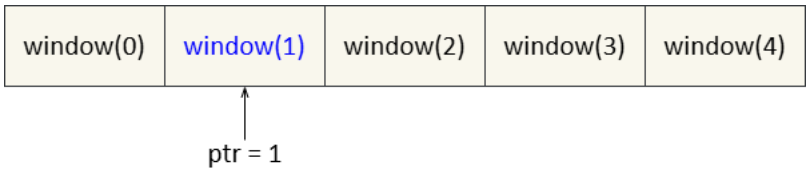


Figure 2: Internal representation of the window

   - All the values in the window are initially set to 0 and the pointer is initialized to 0.
   - Remember that signals are updated only when the process is suspended. This means that the *ptr* pointer will indicate the position in the window where the new input *val* will be placed. This is actually the position of the oldest value in the window. Thus, this value has to be subtracted from the current running sum while adding the input value *val*. After that, the input *val* has to be placed in the window at the same position and the pointer has to be updated accordingly.
   - The steps performed by the module when a new value is provided at the input can be described as follows:

     **procedure** SLIDING-WINDOW-SUM(*val*)
     $sum \leftarrow sum + val - window(ptr)$
     $window(ptr) \leftarrow val$
     **if** $ptr < window.size - 1$ **then**
     $ptr \leftarrow ptr + 1$
     **else**
     $ptr \leftarrow 0$
     **end if**
     **return** *sum*
     **end procedure**

   - The structure of the entity is given below:

```
entity sliding_window_adder is
    Generic (
      WINDOW_SIZE : integer := 5
    );
    Port (
      aclk : IN STD_LOGIC;
```

```
        s_axis_val_tvalid : IN STD_LOGIC;
        s_axis_val_tready : OUT STD_LOGIC;
        s_axis_val_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        m_axis_sum_tvalid : OUT STD_LOGIC;
        m_axis_sum_tready : IN STD_LOGIC;
        m_axis_sum_tdata : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
end sliding_window_adder;
```

3. **Connect the previously implemented modules to get a module which computes the sum of a window of values which are pre-adjusted to fit in a given range. Create a testbench to simulate your design. The block design of the module is shown in Figure 3.**

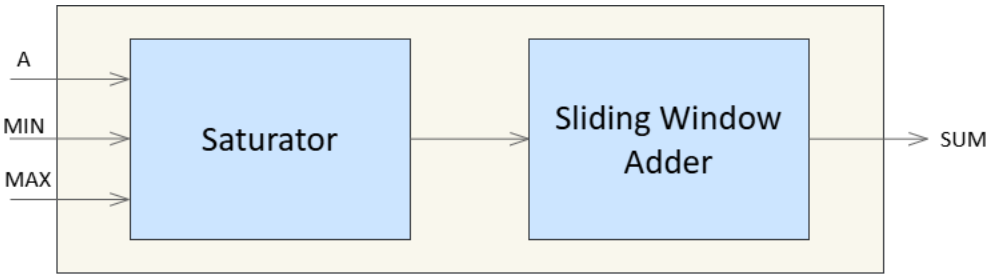


Figure 3: The block design of the module to be implemented

# References


[1] AMBA 4 AXI4-Stream Protocol Specification - Arm [accessed Oct. 2024], https://documentation-service.arm.com/static/642583d7314e245d086bc8c9?token=

[2] How the axi-style ready/valid handshake works [accessed Oct. 2024], https://vhdlwhiz.com/how-the-axi-style-ready-valid-handshake-works/

[3] Stimulus file read in testbench using TEXTIO [accessed Oct. 2024], https://vhdlwhiz.com/stimulus-file/