POLITECNICO
MILANO 1863

Final Project of Logical Networks

Giuliano Crescimbeni
Person Code: 10712403
Registration Number: 960690

Academic Year 2022 - 2023

Contents

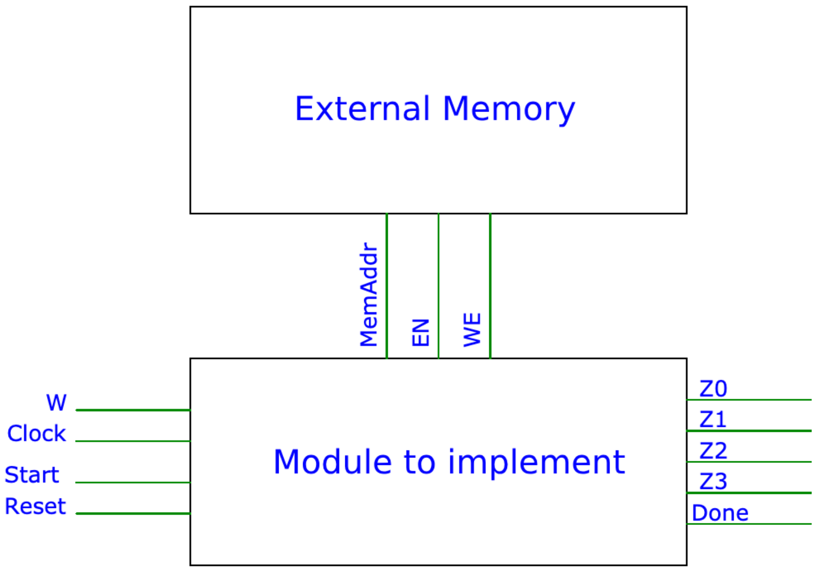
1 Introduction

The problem considered in the final exam of the course involves the implementation of a hardware component defined by the following specification:

1.1 Project Specification

1.1.1 General Description

The module to be implemented has two 1-bit inputs **(W and START)** with 4 8-bit outputs **(Z0, Z1, Z2, Z3)** and a 1-bit end-of-computation signal **(Done)**. Additionally, it is expected that the hardware interfaces with an external memory component via a 16-bit signal **(MemoryAddress)** and two 1-bit signals **(EN, WE)**. The following diagram describes the defined interface:



1.1.2 Operation

Upon receiving the reset signal, the machine must return to the initial state:

- Z0, Z1, Z2, Z3 = **"0000 0000"**
- Done = **’0’**

The input data is distributed to the module through a bit sequence on the **W** signal:

- The first two input bits represent the output channel address:

  - "00" corresponds to Z0
  - "01" corresponds to Z1
  - "10" corresponds to Z2
  - "11" corresponds to Z3
- The remaining bits correspond to the memory address **(MemoryAddress)** for the memory component request.

All bits are sampled on the rising edge of the clock cycle.

The **W** signal can vary from 2 to 18 bits. The channel bits are guaranteed, the remaining portion of the input string has no specified length. If a memory address is not provided in its entirety, it must be extended with ’0’s on the most significant bits. The sequence is valid only when the **START** signal is equal to ’1’.

Once the data is acquired from the memory, it must be registered on the corresponding output channel. All outputs with previously registered values must be shown only when the **Done** bit is set to ’1’, which occurs for a single clock cycle whenever a request to the module has been successfully made.

The **START** signal is guaranteed to remain ’0’ until the next request (i.e., when **Done** is ’1’). The maximum time for computing the result, from when **START** returns to ’0’ until the channel output, must be 20 clock cycles.

1.1.3 Interface Description

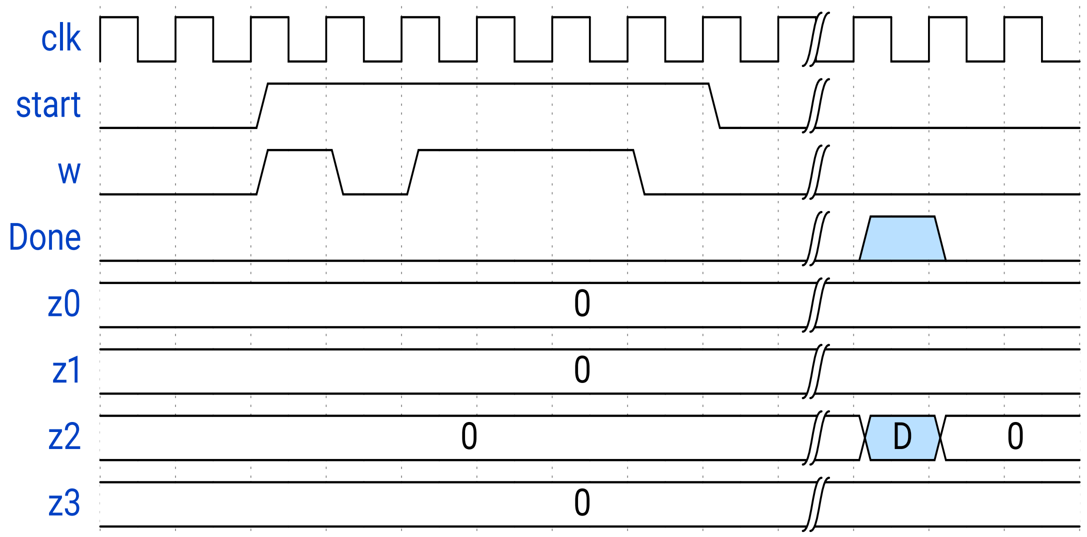
The interface, as specified, must be described as follows:

```
entity project_reti_logiche is
         port (
             i_clk    : in std_logic;
             i_rst    : in std_logic;
             i_start  : in std_logic;
             i_w      : in std_logic;
             o_z0     : out std_logic_vector(7 downto 0);
             o_z1     : out std_logic_vector(7 downto 0);
             o_z2     : out std_logic_vector(7 downto 0);
             o_z3     : out std_logic_vector(7 downto 0);
             o_done   : out std_logic;
             o_mem_addr : out std_logic_vector(15 downto 0);
             i_mem_data : in std_logic_vector(7 downto 0);
             o_mem_we   : out std_logic;
             o_mem_en   : out std_logic
         );
end project_reti_logiche;
```

- **i_clk** is the clock signal, unique to the entire machine **(Input)**
- **i_rst** is the reset signal **(Input)**
- **i_start** is the input sequence start signal **(Input)**
- **i_w** is the input bit that sequences the machine input **(Input)**
- **o_z0** is output channel number 0 **(Output)**
- **o_z1** is output channel number 1 **(Output)**
- **o_z2** is output channel number 2 **(Output)**
- **o_z3** is output channel number 3 **Output)**
- **o_done** is the signal that communicates the end of a request **(Output)**
- **o_mem_addr** is the address communicated to the memory for data retrieval **(Output)**
- **i_mem_data** is the vector of signals representing the data retrieved from the memory **(Input)**

- **o_mem_we** is the memory Write Enable signal, for our purpose it must be set to '0' **(Output)**
- **o_mem_en** is the memory Enable signal, it must be set to '1' when a request is to be made with it **(Output)**

1.1.4 Example of Operation



As can be seen from the example above, the bits corresponding to the channel are **"10"** which correspond to channel **z2**, while the address bits are **"1110"** which will be extended with zeros in the most significant positions. After an unspecified number of clock cycles, the output (represented as **"D"**) is displayed on channel **z2** along with the done signal, which lasts for only one cycle, then returning the channel output to "0".

2 Architecture

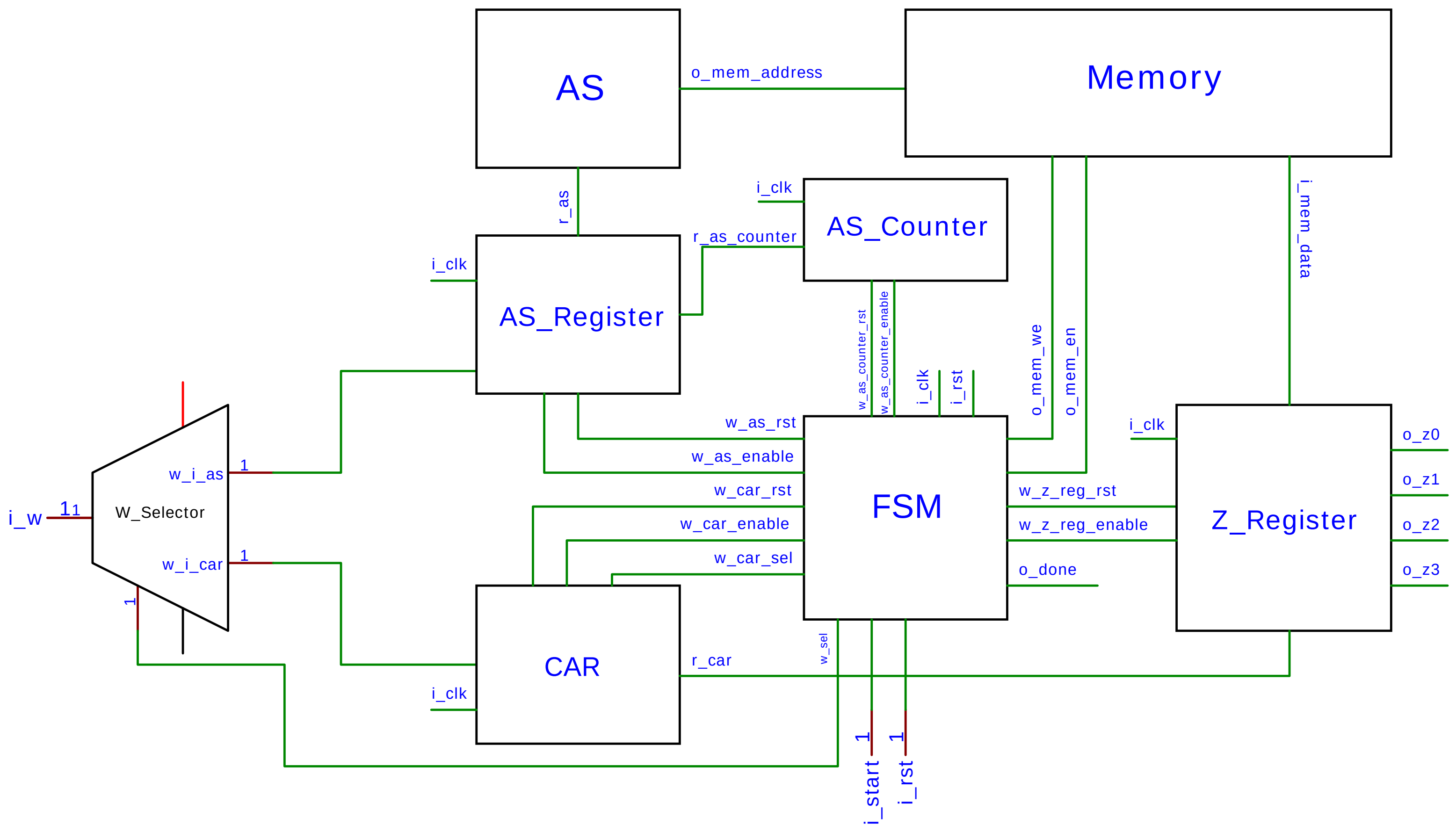
2.1 Datapath

The architecture developed for the correct functioning of the component can be represented, with the appropriate abstract blocks, in the diagram shown on the next page.

Seven modules can be recognized within the diagram:
FSM - W_Selector - CAR - AS_Register - AS_Counter - AS - Z_Register

The implemented memory module, as specified, is a *Single-Port Block RAM Write-First Mode*, a standard memory block from the *Xilinx Design Suite*, and is provided as follows:

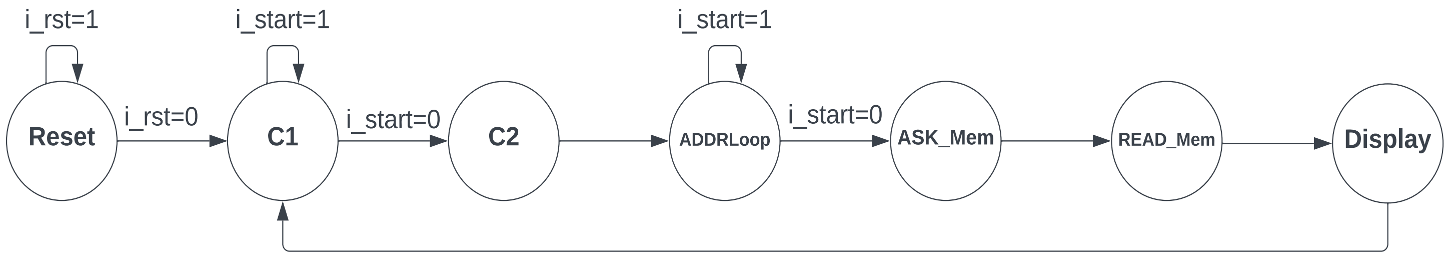
```
-- Single-Port RAM with Asynchronous Read (Distributed RAM)
  -- File: rams_dist.vhd
  library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;
  entity rams_dist is
   port(
    clk : in  std_logic;
    we  : in  std_logic;
    a   : in  std_logic_vector(5 downto 0);
    di  : in  std_logic_vector(15 downto 0);
    do  : out std_logic_vector(15 downto 0)
   );
  end rams_dist;
  architecture syn of rams_dist is
   type ram_type is array (63 downto 0) of std_logic_vector(15 downto 0);
   signal RAM : ram_type;
  begin
   process(clk)
   begin
    if (clk'event and clk = '1') then
     if (we = '1') then
      RAM(to_integer(unsigned(a))) <= di;
     end if;
    end if;
   end process;
   do <= RAM(to_integer(unsigned(a)));
  end syn;
```


AS
o_mem_address
Memory
r_as
i_clk
AS_Counter
r_as_counter
i_clk
AS_Register
i_mem_data
w_as_counter_rst
w_as_counter_enable
i_clk
i_rst
o_mem_we
o_mem_en
i_clk
w_as_rst
w_as_enable
w_car_rst
w_car_enable
w_car_sel
FSM
w_z_reg_rst
w_z_reg_enable
o_done
Z_Register
o_z0
o_z1
o_z2
o_z3
W_Selector
i_w
w_i_as
w_i_car
CAR
r_car
w_sel
i_clk
i_start
i_rst

2.2 Modules

2.2.1 FSM

The **FSM** is the main module and allows the state machine to operate. It consists of 7 states (**Reset, C1, C2, ADDR Loop, ASK_Mem, READ_Mem, Display**) connected as shown in the following diagram:



- **Reset**: Initial state, the machine remains in this state until the reset signal is deasserted.
- **C1**: Idle state, the machine uses this state to wait for the **start** signal and to store the first channel bit, then moves to the next state.
- **C2**: Stores the second channel bit, moves to the next state without condition.
- **ADDRLoop**: State for storing the channel address. The machine cycles within it as long as the **start** signal is high.
- **ASK_Mem**: State used to query the memory device, sets the **Enable** and **WriteEnable** signals.
- **READ_Mem**: State used for receiving and saving data from the memory.
- **Display**: State for outputting the recorded data on the channels, then returns to **C1** to repeat the execution cycle.

The state table and the implementation of the behavior described by the previous diagram are shown on the following pages:

States	o_done	o_mem_we	o_mem_en	w_as_rst	w_as_enable	w_as_counter_rst	w_as_counter_enable	w_car_rst	w_car_enable	w_car_sel	w_z_reg_rst	w_z_reg_enable	w_sel	w_done
Reset	0	0	0	1	0	1	0	1	0	0	1	0	0	0
C1	0	0	0	1	0	1	0	0	1	0	0	0	0	0
C2	0	0	0	0	0	0	0	0	1	1	0	0	0	0
ADDRLoop	0	0	0	0	1	0	1	0	0	0	0	0	1	0
ASK_Mem	0	0	1	0	0	0	0	0	0	0	0	0	0	0
READ_Mem	0	0	0	0	0	0	0	0	0	0	0	1	0	0
Display	1	0	0	0	0	0	0	0	0	0	0	0	0	1

```
StateProcess : process(i_clk, i_rst) is
--Sequential Process
--State cycle management
begin
    if i_rst = '1' then
        State <= Reset;
    elsif rising_edge(i_clk) then
        case State is
            when Reset      =>
                State <= C1;
            when C1         =>
                if i_start = '1' then
                    State <= C2;
                end if;
            when C2         =>
                State <= ADDRLoop;
            when ADDRLoop   =>
                if i_start = '0' then
                    State <= ASK_Mem;
                end if;
            when ASK_Mem    =>
                State <= READ_Mem;
            when READ_Mem   =>
                State <= Display;
            when Display    =>
                State <= C1;
        end case;
    end if;
end process StateProcess;
```

```
ControlProcess : process(State) is
--Combinational Process
--Control signal management
begin
    o_done              <= '0';
    o_mem_we            <= '0';
    o_mem_en            <= '0';
    w_as_rst            <= '0';
    w_as_enable         <= '0';
    w_as_counter_rst    <= '0';
    w_as_counter_enable <= '0';
    w_car_rst           <= '0';
    w_car_enable        <= '0';
    w_car_sel           <= '0';
    w_z_reg_rst         <= '0';
    w_z_reg_enable      <= '0';
    w_sel               <= '0';
    w_done              <= '0';
    case state is
        when Reset      =>
            w_as_rst            <= '1';
            w_as_counter_rst    <= '1';
            w_car_rst           <= '1';
            w_z_reg_rst         <= '1';
        when C1         =>
            w_as_rst            <= '1';
            w_as_counter_rst    <= '1';
            w_car_enable        <= '1';
        when C2         =>
            w_car_enable        <= '1';
            w_car_sel           <= '1';
        when ADDRLoop   =>
            w_as_enable         <= '1';
            w_as_counter_enable <= '1';
            w_sel               <= '1';
        when ASK_Mem    =>
            o_mem_en            <= '1';
        when READ_Mem   =>
            w_z_reg_enable      <= '1';
        when Display    =>
            o_done              <= '1';
            w_done              <= '1';
    end case;
end process ControlProcess;
```

The sequential process on the left (**StateProcess**) is the state management process, it creates the execution cycle of the machine.

The combinational process on the right (**ControlProcess**) manages the control signals in the various states.

2.2.2 W_Selector

This module is the first component that the input data **w** encounters when entering the device. It is a **DEMUX** and is responsible for directing the input bit to the correct module:

```
W_Selector : process(i_w, w_sel) is
--Combinational Process
--W routing
begin
    if w_sel = '0' then
        w_i_car <= i_w;
        w_i_as  <= '0';
    else
        w_i_car <= '0';
        w_i_as  <= i_w;
    end if;
end process W_Selector;
```

The module is combinational, depending on the input (**i_w**) and the command from the FSM (**w_sel**). In states **C0** and **C1**, when **i_w** is '0', it directs **i_w** to the **CAR**, otherwise to the **AS_Register**.

2.2.3 CAR

The **CAR (Channel Address Register)** is the component responsible for saving the channel bits. It can be schematized as a **DEMUX** connected to two one-bit registers.

```
CAR_Register : process(i_clk, w_car_rst) is
--Sequential Process
--Saving channel address
begin
    if w_car_rst = '1' then
        r_car(0) <= '0';
        r_car(1) <= '0';
    elsif rising_edge(i_clk) then
        if w_car_enable = '1' then
            case w_car_sel is
                when '0' => r_car(0) <= w_i_car;
                when '1' => r_car(1) <= w_i_car;
                when others =>
            end case;
        end if;
    end if;
end process CAR_Register;
```

The process is sequential, depending on **i_clk** and the reset signal (**w_car_rst**) sent by the **FSM**. In case of reset, the module returns to the default value

"00". The channel bits are saved only if there is a high signal from the state machine (**w_car_enable**), sampled from the **W_Selector** output.

2.2.4 AS_Counter

The **AS_Counter (Address Shifter Counter)** module is a support component for the **AS_Register** and **AS** modules. Its implementation was chosen to keep track of the unknown number of memory address bits, and thus for correct saving and shifting:

```
AS_Counter : process(i_clk, w_as_counter_rst) is
--Sequential Process
--Incoming bit counter
begin
    if w_as_counter_rst = '1' then
        r_as_counter <= "0000";
    elsif rising_edge(i_clk) then
        if w_as_counter_enable = '1' then
            r_as_counter <= r_as_counter + 1;
        end if;
    end if;
end process AS_Counter;
```

The process that composes the module is sequential. Essentially, it represents a standard 4-bit counter, with module **"16"**, **"A minimum number of bits"** code, and **"Natural Binary"** encoding. It depends on **i_clk** and the reset signal (**w_as_counter_rst**) with reset state "0000". It is enabled by the **w_as_counter_enable** command in the **ADDRLoop** state and samples the previously saved number + 1.

2.2.5 AS_Register

The **AS_Register (Address Shifter Register)** is used to save the address bits.

```
AS_Register : process(i_clk, w_as_rst) is
--Sequential Process
--Saving address bits
begin
    if w_as_rst = '1' then
        r_as <= "0000000000000000";
    elsif rising_edge(i_clk) then
        if i_start = '1' then
            case r_as_counter is
```

```
                when "0000" => r_as(0)  <= w_i_as;
                when "0001" => r_as(1)  <= w_i_as;
                when "0010" => r_as(2)  <= w_i_as;
                when "0011" => r_as(3)  <= w_i_as;
                when "0100" => r_as(4)  <= w_i_as;
                when "0101" => r_as(5)  <= w_i_as;
                when "0110" => r_as(6)  <= w_i_as;
                when "0111" => r_as(7)  <= w_i_as;
                when "1000" => r_as(8)  <= w_i_as;
                when "1001" => r_as(9)  <= w_i_as;
                when "1010" => r_as(10) <= w_i_as;
                when "1011" => r_as(11) <= w_i_as;
                when "1100" => r_as(12) <= w_i_as;
                when "1101" => r_as(13) <= w_i_as;
                when "1110" => r_as(14) <= w_i_as;
                when "1111" => r_as(15) <= w_i_as;
                when others =>
            end case;
        end if;
    end if;
end process AS_Register;
```

It is very similar to the **CAR** with the difference that the **DEMUX** for choosing the register to save the input bit is driven by the counter (**AS_Counter**). The process is sequential, depending on **i_clk** and the reset signal (**w_as_reset**) but does not have any enable signal as its data does not need to be preserved for long, only during the **ADDRLoop** cycle period.

2.2.6 AS

The **AS (Address Shifter)** is a component dependent on the **AS_Register** and the **AS_Counter**. Its operation is dynamic as its output changes during the acquisition of the memory address bits. The process that implements it is combinational and depends on **r_as** and **r_as_counter**, respectively the register containing the address and the one containing the counter. Based on the number of iterations ***i***, the shifter appends a number ***i* - 1** of **'0'** to the left of the memory address bits saved in the **AS_Register**, swapping the least significant bit position with the most significant, and so on. The output of the **AS** is directly directed to the output signal to the memory (**o_mem_addr**) to make the request:

```
AS_Shift : process(r_as, r_as_counter) is
--Combinational Process
--Address bit shifting
begin
    case r_as_counter is
        when "0001" => o_mem_addr <=  r_as(0) & r_as(1) . . .
        when "0010" => o_mem_addr <= "000000000000000" & r_as(0);
        when "0011" => o_mem_addr <= "00000000000000" & r_as(0) & r_as(1);
        .
        .
        .
        when others => o_mem_addr <= "XXXXXXXXXXXXXXXX";
    end case;
end process AS_Shift;
```

The process is not fully reported due to its size.

2.2.7 Z_Register

The module is used for saving data, acquired from the memory, in the respective output channel registers:

```
Z_Register : process(i_clk, w_z_reg_rst) is
--Sequential Process
--Saving memory output
begin
    if w_z_reg_rst = '1' then
        r_z_reg_0 <= "00000000";
        r_z_reg_1 <= "00000000";
        r_z_reg_2 <= "00000000";
        r_z_reg_3 <= "00000000";
    elsif rising_edge(i_clk) then
        if w_z_reg_enable = '1' then
            case r_car is
                when "00" => r_z_reg_0 <= i_mem_data;
                when "01" => r_z_reg_1 <= i_mem_data;
                when "10" => r_z_reg_2 <= i_mem_data;
                when "11" => r_z_reg_3 <= i_mem_data;
                when others =>
            end case;
        end if;
    end if;
end process Z_Register;
```

The process is sequential, depending on **i_clk** and the reset signal (**w_z_reg_rst)**. There are 4 signals (**r_z_reg_n**) representing the individual output channel registers. When the enable command (**w_z_reg_enable**) is sent, via a **DEMUX** that takes the channel address of the **CAR (r_car)** as a control command, the module can choose which register to sample the memory output to save the data.

2.2.8 Output

The following combinational Output process is used to display the registers (**Z_Register**) on the output channels (**o_zN**). It varies based on the parameters **r_z_reg_0, r_z_reg_1, r_z_reg_2, r_z_reg_3, w_done** and is activated only by the **w_done** command signal of the **FSM**, so that it returns the register values only in the end computation clock cycle (**o_done =1**):

```
Output : process(r_z_reg_0, r_z_reg_1, r_z_reg_2, r_z_reg_3, w_done) is
--Combinational Process
--Display of output registers on output channels
begin
    if w_done = '1' then
        o_z0 <= r_z_reg_0;
        o_z1 <= r_z_reg_1;
        o_z2 <= r_z_reg_2;
        o_z3 <= r_z_reg_3;
    elsif w_done = '0' then
        o_z0 <= "00000000";
        o_z1 <= "00000000";
        o_z2 <= "00000000";
        o_z3 <= "00000000";
    end if;
end process Output;
```

3 Experimental Results

3.1 Synthesis Report

Here are the utilization and timing reports of the *post-synthesis* design generated by the Vivado tool:

3.1.1 Utilization Report

```
+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             |   86 |     0 |    134600 |  0.06 |
|   LUT as Logic          |   86 |     0 |    134600 |  0.06 |
|   LUT as Memory         |    0 |     0 |     46200 |  0.00 |
| Slice Registers         |   57 |     0 |    269200 |  0.02 |
|   Register as Flip Flop |   57 |     0 |    269200 |  0.02 |
|   Register as Latch     |    0 |     0 |    269200 |  0.00 |
| F7 Muxes                |    0 |     0 |     67300 |  0.00 |
| F8 Muxes                |    0 |     0 |     33650 |  0.00 |
+-------------------------+------+-------+-----------+-------+
```

3.1.2 Timing Report

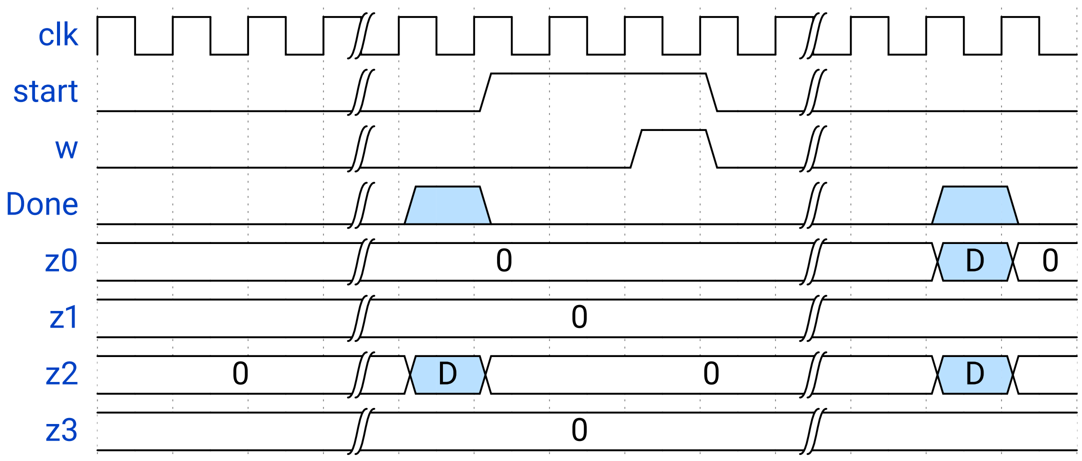
```
Slack (MET) :             96.989ns  (required time - arrival time)
  Path Group:             **async_default**
  Path Type:              Recovery (Max at Slow Process Corner)
  Requirement:            100.000ns(clock rise@100.000ns-clock rise@0.000ns)
  Data Path Delay:        2.422ns(logic 0.751ns (31.007%)
                                  route 1.671ns (68.993%))
  Logic Levels:           1  (LUT3=1)
```

With a requirement of **100.000ns**, the Slack is **96.989ns**, meaning the longest combinatorial path has a delay of **3.011ns**, well within the design specifications.

3.2 Simulations

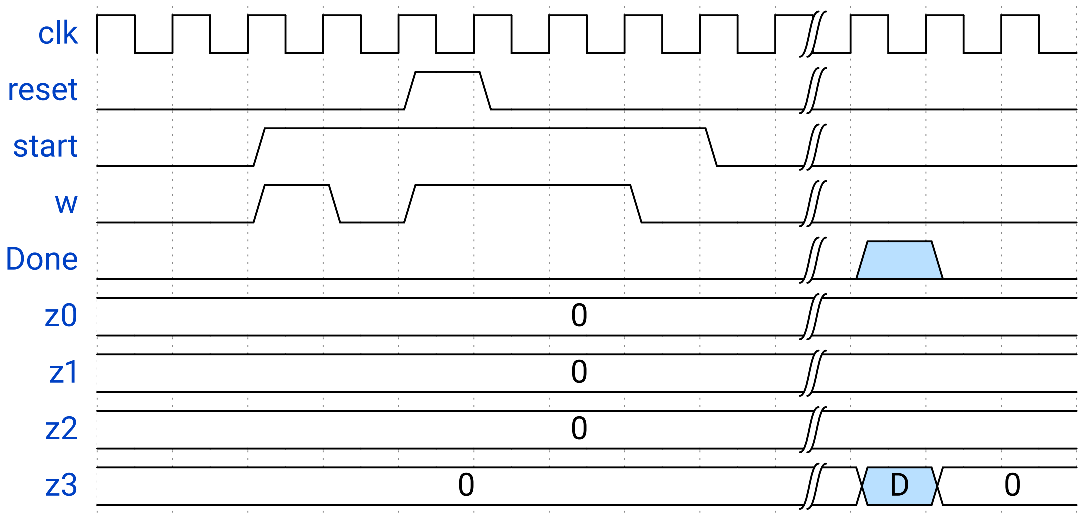
All the provided testbenches are successfully executed by the machine, both pre- and post-synthesis within the required timing terms. The following edge cases were also analyzed, each demonstrating correct operation:

3.2.1 TestBench 1



This testbench verifies the operation condition when the **start** signal is called immediately after the end of a previous completed computation. This serves to verify the possibility of data loss during the machine reset. Testing with an extreme case (on the falling edge of the **done** signal) shows that this does not occur, and thus the machine cannot lose data when transitioning from one request to another.

3.2.2 TestBench 2



The second testbench verifies the correct behavior of the machine in the event of a **reset** during the operation period. In this case, the **reset** occurs in the third **clock** cycle, thereby invalidating the previously acquired channel bits.

The result should be reported in channel **Z3**. This verification was conducted in multiple scenarios, with the **reset** signal appearing at the beginning, middle, and end of the operation process; in each case, the machine's behavior was correct.

4 Conclusions

The component was developed according to the criteria required by the specification. It meets all timing constraints for result creation, clock constraints, and synthesis. I analyzed the problem in as much detail as possible; the testbenches I used are multiple, both provided by various professors and created by myself for edge cases, covering all scenarios I could think of. I tested and reworked the circuit several times to optimize its performance and address potential critical issues as much as possible.