

# VHDL PROJECT

## **Mini-Router**

Marie Giannoni

UNIVERSITY OF PISA



School of Engineering  
Master of Science in Computer Engineering  
**Electronics Systems**

ACADEMIC YEAR 2019/2020

## SUMMARY

---

1	Introduction .....	4
1.1	Description of Inputs and Outputs .....	4
1.1.1	Clock and Reset .....	4
1.1.2	Links Sources .....	4
1.1.3	Output Link .....	5
1.2	Levels of Priority .....	5
1.3	Round Robin Algorithm .....	5
1.4	Possible architectures .....	6
2	Architecture description .....	6
2.1	Mini-Router .....	6
2.2	Registered Inputs .....	7
2.3	Decoders .....	7
2.4	Round Robin and LUT .....	9
2.5	Multiplexer .....	9
2.6	Registered output .....	7
2.7	Adapter .....	9
2.8	PISO (Parallel In Serial Out) and SIPO (Serial In Parallel Out) .....	10
3	VHDL code .....	11
3.1	D Flip Flop .....	11
3.1.1	Entity description .....	11
3.1.2	Code .....	11
3.2	D Flip Flop with N Bits .....	11
3.2.1	Entity description .....	11
3.2.2	Code .....	12
3.3	PISO : Parallel In Serial Out .....	12
3.3.1	Entity description .....	12
3.3.2	Signals .....	12
3.3.3	Code .....	12
3.4	SIPO: Serial In Parallel Out .....	12
3.4.1	Entity description .....	12
3.4.2	Signals .....	13
3.4.3	Code .....	13
3.5	Adapter .....	13
3.5.1	Entity description .....	13
3.5.2	Components .....	13
3.5.3	Signals .....	13
3.5.4	Code .....	14
3.6	Mini-Router .....	15

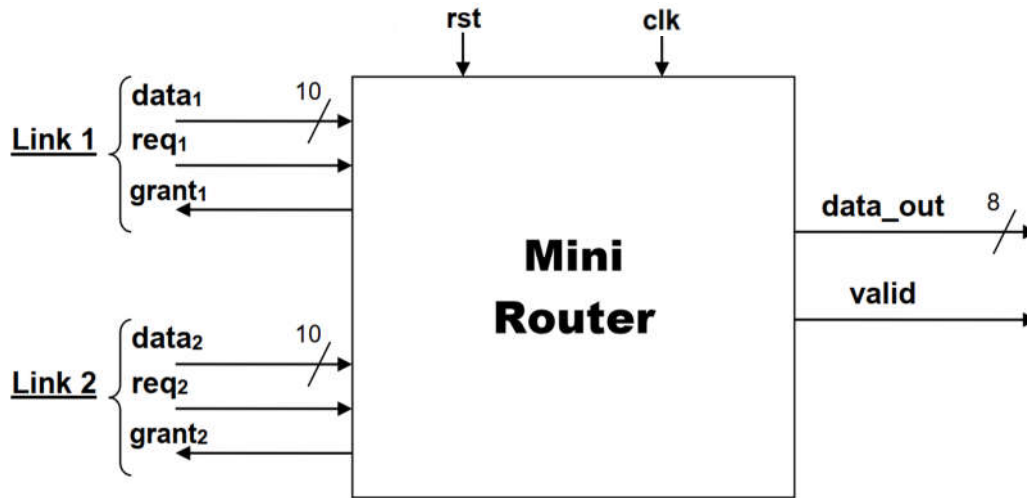
3.6.1	Entity description .....	15
3.6.2	Components.....	16
3.6.3	Signals .....	16
3.6.4	Code .....	17
3.7	Decoder (for a Link).....	18
3.7.1	Entity description .....	18
3.7.2	Code .....	19
3.8	Round Robin Arbiter .....	19
3.8.1	Entity description .....	19
3.8.2	Components.....	20
3.8.3	Signals .....	20
3.8.4	Code .....	20
3.9	LUT (for the Round-Robin Algorithm).....	21
3.9.1	Entity description .....	21
3.9.2	Signals .....	21
3.9.3	Code .....	22
3.10	Multiplexer (for directing the selected input Link to the output Link).....	22
3.10.1	Entity description.....	22
3.10.2	Signal .....	23
3.10.3	Code.....	23
4	Test Plan.....	23
5	Testbench .....	24
5.1	SIPO .....	24
5.2	PISO .....	25
5.3	Decoder (for a Link).....	25
5.4	Multiplexer (for directing the selected Input link to the output Link) .....	26
5.5	Round Robin Arbiter .....	26
5.6	Mini-Router .....	30
5.7	Adapter .....	33
6	Synthesis Logic results .....	34
6.1	Constraints .....	34
6.2	Synthesis Warnings .....	35
6.3	Device Utilization.....	35
6.4	Maximum Frequency and Relevant Critical Paths .....	36
7	Conclusion.....	39

# 1 INTRODUCTION

The main goal of this project is to design and implement, using VHDL and a Zynq FPGA, of a synchronous mini-router. The mini-router has two input links and one output direction. Its role is to direct one of the two links to the output direction. The selection of the link to be transferred is made according to the following rules:

- A request is emitted through the input link to be routed;
- When both links have made a request, the selected link is the one with the highest priority level;
- When the two links have the same priority level, a Round Robin algorithm is used to play the role of the arbiter and decide between them.

The schema of the mini-router is the following:



*Schema of the Mini-Router with all inputs and outputs*

## 1.1 DESCRIPTION OF INPUTS AND OUTPUTS

### 1.1.1 Clock and Reset

The present mini-router is a synchronous circuit. The clock signal provides a common time reference in order to all input and output signals and all internal blocks and nodes to be stable in high or low state on the active rising edge of the clock.

The clock is then represented in the project by:

- clk: in std\_logic;

The reset is an asynchronous reset and is active low:

- rst: in std\_logic;

### 1.1.2 Links Sources

A link source is composed by three signals, two inputs and one output. For  $i=\{1,2\}$ , we have:

- data<sub>i</sub>: in std\_logic\_vector(9 downto 0);

It is the bus of 10 bits to be transferred to the external link. It is composed by 8 bits of data and 2 bits of priority, the 0<sup>th</sup> (minor priority) and the 3<sup>rd</sup> (major priority).

- req<sub>i</sub>: in std\_logic;

This signal represents the request. When a data transfer request is made, the signal is high. Otherwise, the signal is low.

- grant<sub>i</sub>: out std\_logic;

When the system grants the link to be transferred, the signal grant is set high. Otherwise, it is set low.

### 1.1.3 Output Link

The external output link is composed by two signals: the data and a signal that said if data are transferred.

- data\_out: out std\_logic\_vector(7 downto 0);

data\_out takes the value of the data<sub>i</sub> selected without the two bits of priority. It is then a bus of 8 bits.

- valid: out std\_logic;

valid is a signal that informs if data\_out is feed by a link source or not. It is set high when a transfer occurs and low otherwise.

## 1.2 LEVELS OF PRIORITY

When both links request to be selected, we must decide between them which one is chosen. Therefore, we use two bits of priority: the number 0 and the number 3 on the 10-length bus of data<sub>i</sub>.

The bit number 3 is the bit of major priority. So, we first check it. If one of the bit number 3 of the two links is high, then we select the link with it. Otherwise, if both number 3 bits are high or low, we need to use the bit number 0 in order to decide between them.

The bit number 0 is the bit of minor priority. We check it if the bit number 3 of both signals are high or low at the same time. If only one of the bit number 0 of the two links is high, then we select the link with it. Otherwise, we have a data conflict and we need to use an arbiter.

*First example:*

Link	data <sub>i</sub>	Bit number 3	Bit number 0
1	“101101 <b>1</b> 01 <b>1</b> ”	1	1
2	“100100 <b>0</b> 11 <b>1</b> ”	0	1

We select link 1 because its bit number 3 is high when the bit number 3 of link 2 is low.

*Second example:*

Link	data <sub>i</sub>	Bit number 3	Bit number 0
1	“101101 <b>0</b> 01 <b>0</b> ”	0	0
2	“100100 <b>0</b> 11 <b>1</b> ”	0	1

We select link 2 because its bit number 1 is high when the bit number 1 of link 1 is low.

*Third example:*

Link	data <sub>i</sub>	Bit number 3	Bit number 0
1	“101101 <b>1</b> 01 <b>0</b> ”	1	0
2	“100100 <b>1</b> 11 <b>0</b> ”	1	0

There is a data conflict.

## 1.3 ROUND ROBIN ALGORITHM

When there is a data conflict, the links must be decided between. A Round Robin algorithm is then used. After the reset, at the first conflict the link 1 is selected. At the second conflict, the link 2 is selected. At the third conflict, the link 1 is selected again and so on. Then, we choose link 1 for odd-numbered conflict and link 2 for even-numbered conflict.

## 1.4 POSSIBLE ARCHITECTURES

The possible architectures for a mini-router are :

- A sequential logic circuit:

Since our system is synchronous, we can perfectly describe a sequential architecture, i.e. with the help of a time-dependent function and thus using processes.

- A combinatorial logic circuit:

We can also describe functionalities in separate blocks using competitive logic. Input and output registers allow us to synchronize this combinatorial architecture and protect the inputs and outputs during code execution.

- A combination of sequential and combinatorial logic:

Finally, it is possible to combine combinatorial and sequential logics. Thus, processes can coexist with multiplexers and other combinatorial functions.

## 2 ARCHITECTURE DESCRIPTION

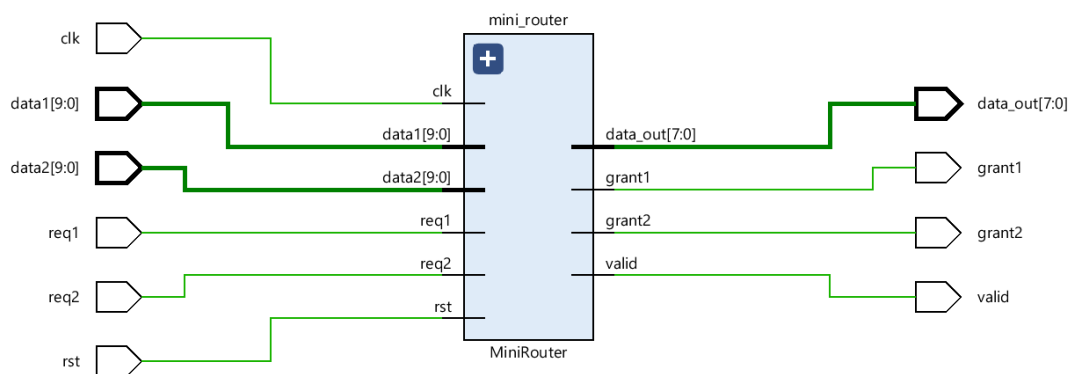
I first started on a mix between combinatorial and sequential logic by writing a sequential process for the Round-Robin algorithm. Finally, I found a more optimized solution, so I don't have sequential process anymore in the mini-router itself. The architecture chosen for the implementation of this mini-router is then a combinatorial architecture, with d flip flop registers to synchronize it.

I also use SIPO and PISO for respectively parallelize and serialize data, which are sequential process. Indeed, I did a version of the whole architecture only with parallel inputs and output. And then, I realize that the Zynq 7000 FPGA board has not enough input and output pins to test my design. Then, I had two possibilities (I have no doubt that there are more but I have only thought about two of them) to solve this problem: using features like seven segment display, VGA monitor, etc. or using a serial output and input that I would parallelize with a faster clock.

I decide not doing the first possibility i.e. using other features. Indeed, these features could be difficult to manage in my design and, most of all, could be really time consuming. Then, I opt to implement the second solution: convert serial input to parallel output for the data of both input links and convert parallel input to data output for the data of the output link. Such a conversion strategy can be used when we need to minimize the connection wires. The clock must run at least 10 times faster since there are ten bits to be parallelized for the inputs. Using a serial connection is also a good way to minimize the skew problem on the connection i.e. when the same sourced clock arrives at different components at different times.

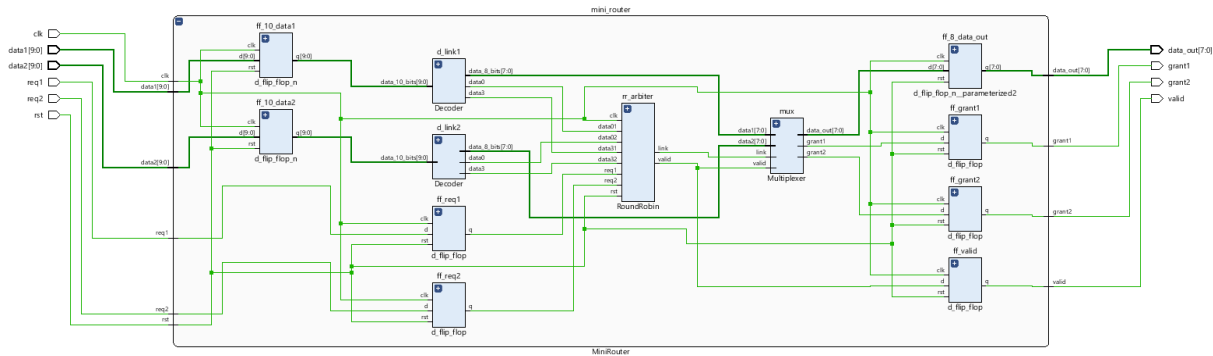
The global architecture is then a combination of sequential and combinatorial logic, even if the mini-router, the core of the project, is only combinatorial.

### 2.1 MINI-ROUTER



*Architecture of the Mini-Router*

As we have seen in the introduction, the mini-router is a block with six input and four output signals. It has twelve blocks in its architecture. These components can be seen on the following schema:

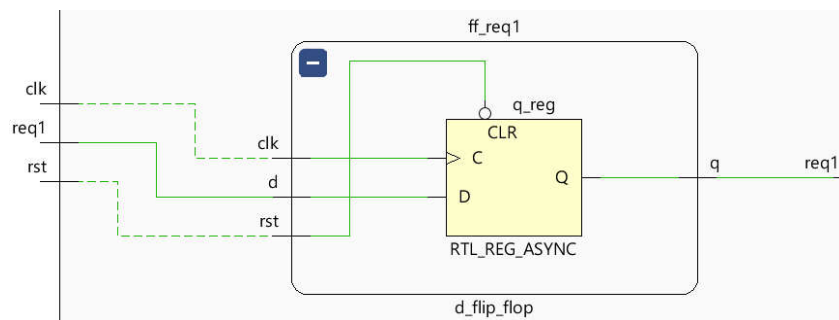


*More detailed architecture of the Mini-Router*

It is composed with registers for the input and the output (8 blocks), with two decoder blocks for decode the data of link 1 and link 2 (2 blocks), with the round-robin arbiter component (1 block) and with a multiplexer that drive the correct link to the output link (1 block).

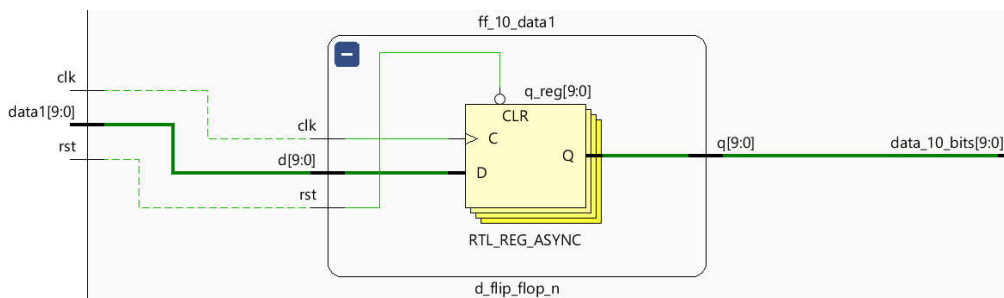
## 2.2 REGISTERED INPUTS

The inputs are registered in d flip flop for the std\_logic signals and in d flip flop of n bits for the std\_logic\_vector(N-1 downto 0) signals. I use a d flip flop for req1 and req2:



*Example of register for req1*

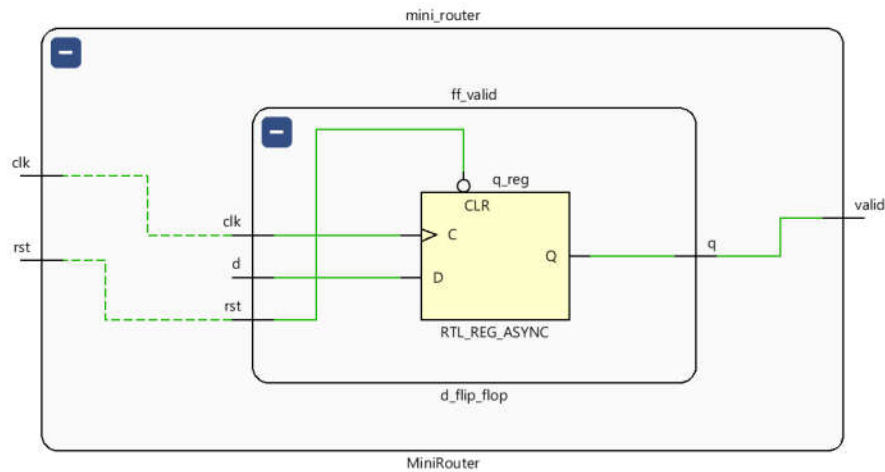
I also use a d flip flop with 10 bits for data1 and data2:



*Example of register for data2*

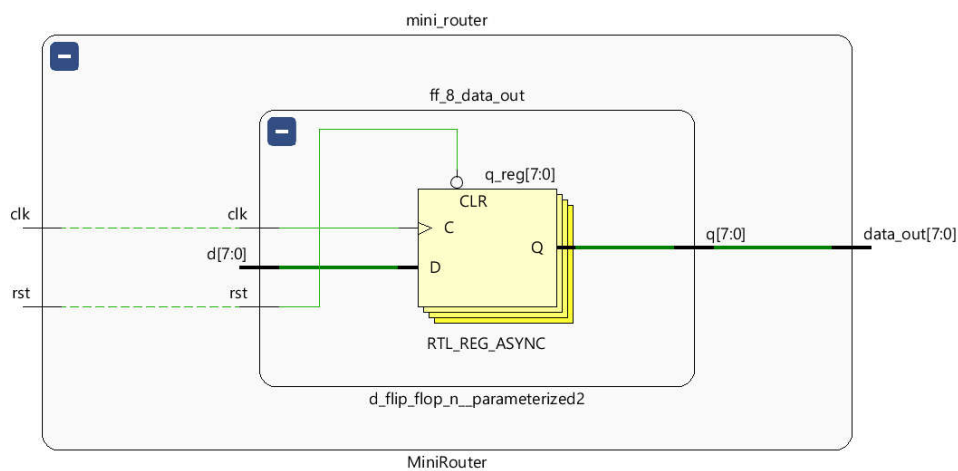
## 2.3 REGISTERED OUTPUT

The output signals of the Mini-Router are registered. The registered output and input are necessary to synchronize the design and are implemented using d flip flop.



*Example of a register for valid*

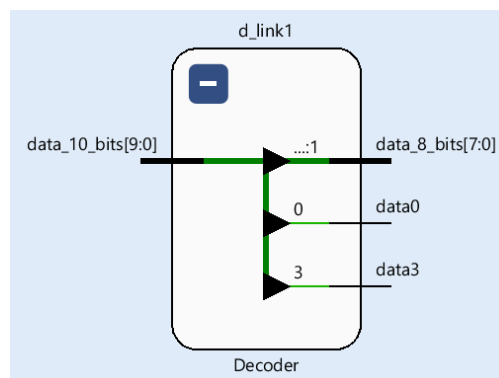
I also use a d flip flop register with eight bits of input and output for `data_out`:



*Example of a register for data\_out*

## 2.4 DECODERS

In the Mini-Router, there are two blocks for decoding the data input of both links 1 and 2. A Decoder component has one input and three outputs signals. The input comes directly from a register.



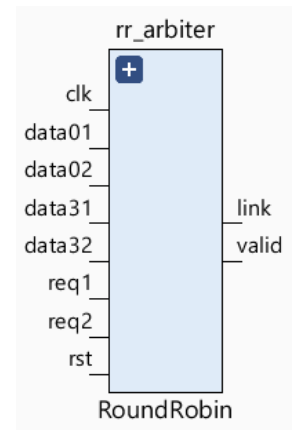
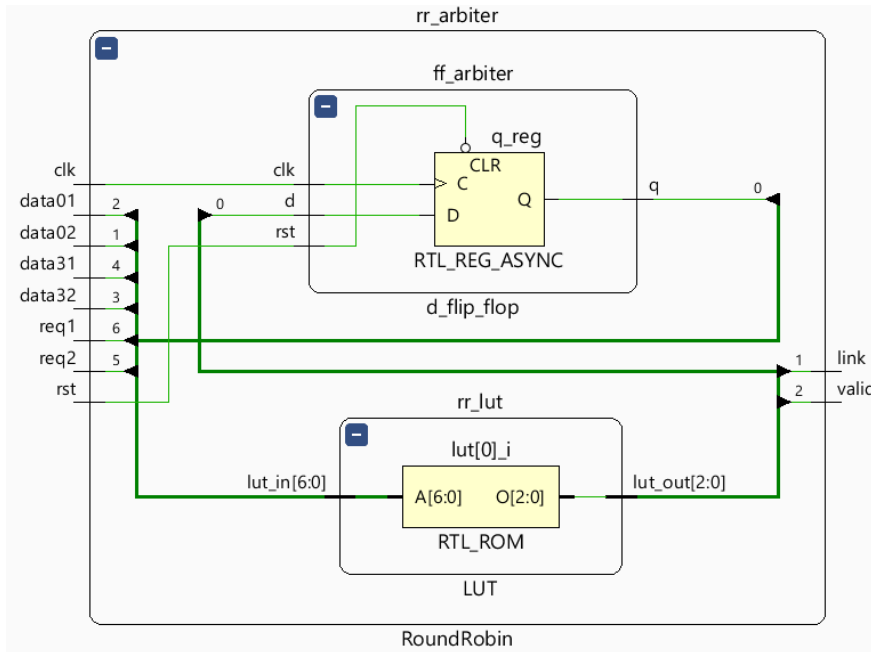
*Example of decoder for link1*



## 2.5 ROUND ROBIN AND LUT

The RoundRobin component has 8 1-bit-length inputs signals and two 1-bit-length output signals.

It is composed with two sub-components: a d flip flop, storing the state of the arbiter, and a LUT.



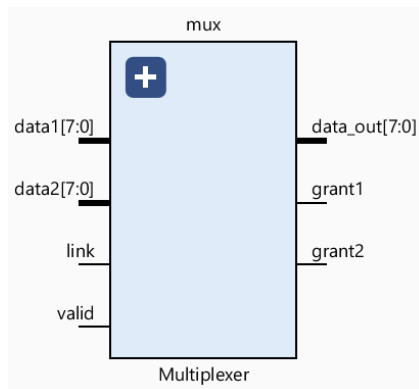
*Architecture of the Round-Robin component*

*Architecture of Round-Robin with its components*

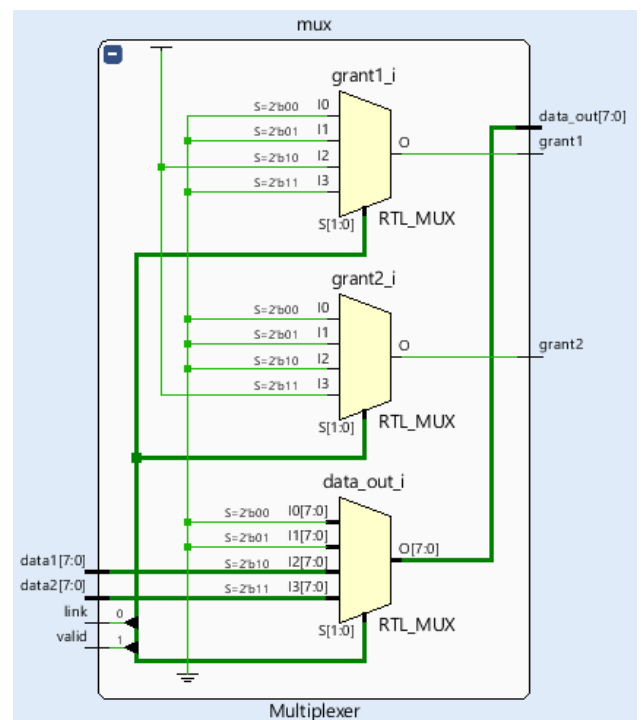
## 2.6 MULTIPLEXER

The multiplexer is a component with four input and three output signals. It has no subcomponent.

The multiplexer is composed by three little multiplexer logic functions: one for grant1, one for grant2 and one for data\_out.



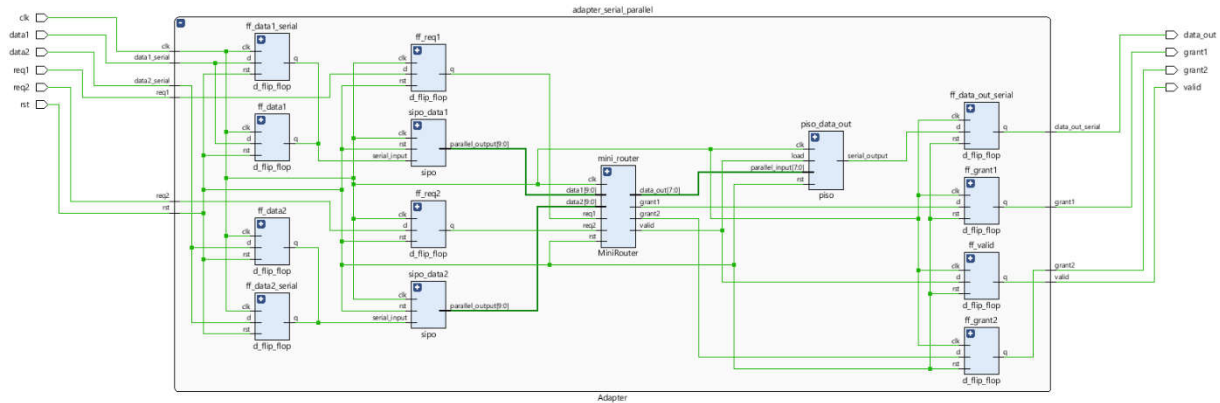
*Architecture of the Multiplexer component*



*Internal composition of the Multiplexer*

## 2.7 ADAPTER

I use an adapter that would parallelize the input and serialized output using PISO and SIPO registers.

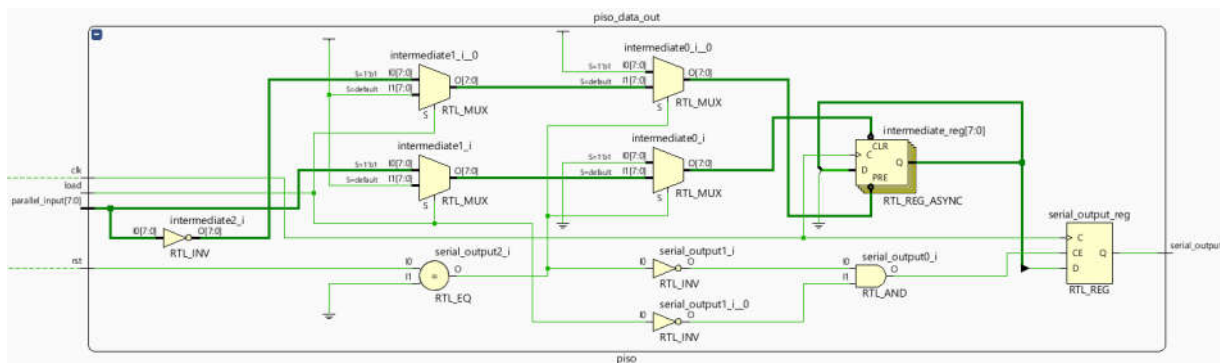


Adapter's architecture

The input and output signals are registered (data1, data2, data\_out, grant1, grant2, req1, req2, valid). data1 and data2 are also parallelized using SIPOs and data\_out is serialized using a PISO register. Like this, I don't have to rethink all the architecture of the Mini-Router itself and I can still use the FPGA board with less pins.

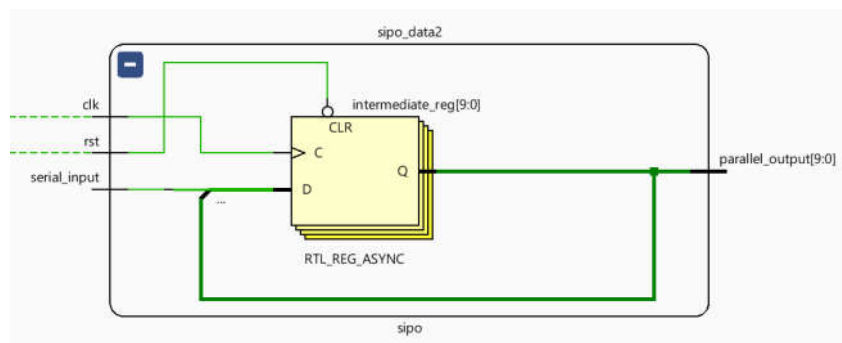
## 2.8 PISO (PARALLEL IN SERIAL OUT) AND SIPO (SERIAL IN PARALLEL OUT)

PISO and SIPO are then two shift registers. PISO converts N-bits data input to a one-bit data output. I use it for converting the data\_out of the Mini-Router into a serial signal.



Architecture of the PISO register for the data output

SIPO converts one-bit data input to N-bits data output. I use it for converting the data of link 1 and link 2 into parallelized input.



Architecture of the SIPO register for data of link 2

## 3 VHDL CODE

The VHDL code is composed of several entities: the Adapter, which allows to use serial input and output instead of parallelized ones, the MiniRouter which receives two links as inputs and returns one link as an output, the Decoder which removes the priority bits from the data of the link, the RoundRobin which is an arbiter following a Round-Robin algorithm and determines which link should be transferred to the output, and the Multiplexer, which is the entity that leads the right data to the output according to the RoundRobin indications. There are also registers: d flip flop, d flip flop parallel, parallel-in serial-out and serial-in parallel-out.

### 3.1 D FLIP FLOP

#### 3.1.1 Entity description

```
entity d_flip_flop is
  port(
    clk: in std_logic;
    rst: in std_logic;
    d: in std_logic;
    q: out std_logic
  );
end d_flip_flop;
```

The entity of a d flip flop has:

- one input: d: in std\_logic
- one output: q: out std\_logic

It is synchronous thanks to the use of the clock and it has an asynchronous active low reset. There are important to ensure that the design is synchronized and to improve its performances. Indeed, it improves the data delay.

#### 3.1.2 Code

```
architecture rtl of d_flip_flop is
begin
  d_flip_flop_process: process (clk, rst)
  begin
    if (rst = '0') then -- asynchronous low reset
      q <= '0';
    elsif (rising_edge(clk)) then -- when the clk is high and on a rising edge,
      -- we give to q the value of d
      q <= d;
    end if;
  end process;
end rtl;
```

When the clock is active on a rising edge, the output q is set with the state of d. Otherwise, q is set to '0'.

### 3.2 D FLIP FLOP WITH N BITS

It is really similar to the D Flip Flop. The only difference is that the input and the output are bus of N bits. This entity is generic in order to use D Flip Flop of N bits for various values of N.

#### 3.2.1 Entity description

```
entity d_flip_flop_n is
  generic(N: integer);
  Port(
    clk: in std_logic;
    rst: in std_logic;
    d: in std_logic_vector(N-1 downto 0);
    q: out std_logic_vector(N-1 downto 0)
  );
end d_flip_flop_n;
```

As you can see on the code, we have a generic entity with:

- one input: d: in std\_logic\_vector(N-1 downto 0)
- one output: q: out std\_logic\_vector(N-1 downto 0)
- the clock and the reset (asynchronous active low)

### 3.2.2 Code

```
architecture rtl of d_flip_flop_n is
begin
    d_flip_flop_n_process: process (clk, rst)
    begin
        if (rst = '0') then -- asynchronous low reset
            q <= (others => '0');
        elsif (rising_edge(clk)) then -- when the clk is high and on a rising edge,
            -- we give to q the value of d for all the bits
            q(N-1 downto 0) <= d(N-1 downto 0);
        end if;
    end process;
end rtl;
```

When the clock is active on a rising edge, the output q is set to the state of d. Otherwise, each bit of q is set to '0'.

## 3.3 PISO : PARALLEL IN SERIAL OUT

### 3.3.1 Entity description

The PISO is a generic register with four input and one output signals. The inputs are the clock clk and the reset rst and the parallel\_input which is a std\_logic\_vector(N-1 downto 0) and load, a std\_logic which is in fact the value of valid. The output is a one-bit signal names serial\_output.

```
entity piso is
    generic(N: integer);
    port(
        clk: in std_logic;
        rst: in std_logic;
        load: in std_logic;
        parallel_input: in std_logic_vector(N-1 downto 0);
        serial_output: out std_logic
    );
end piso;
```

### 3.3.2 Signals

I use an intermediate N-bits signal. This signal allows to left shift the parallel input and then send out one by one each bit.

```
signal intermediate: std_logic_vector(N-1 downto 0);
```

### 3.3.3 Code

```
piso_process: process (clk, rst, load, parallel_input, intermediate)
begin
    if (rst = '0') then -- asynchronous low reset
        intermediate <= (others => '0');
    elsif (load = '1') then
        -- we give to intermediate the value of parallel_input
        intermediate <= parallel_input;
    elsif (rising_edge(clk)) then -- when the clk is high and on a rising edge
        -- we give the the output the value of the LSB
        serial_output <= intermediate(N-1);
        -- we shift by one bit the signal
        intermediate(N-1 downto 0) <= intermediate(N-2 downto 0) & '0';
    end if;
end process;
```

When valid is high, we give to intermediate the state of the input parallel\_input. And when the clock is active and on a rising edge, we send to the output the value of the most significant bit and we left shift the data. When the reset is active (low), we connect intermediate to the ground.

## 3.4 SIPO: SERIAL IN PARALLEL OUT

### 3.4.1 Entity description

The SIPO is a generic shift register with three input and one output signals. The inputs are the clock, clk the reset, rst, and the serial input, serial\_input, which is a std\_logic. The output is a std\_logic\_vector of N bits.

```
entity sipo is
    generic(N: integer);
    port(
        clk: in std_logic;
        rst: in std_logic;
        serial_input: in std_logic;
        parallel_output: out std_logic_vector(N-1 downto 0)
    );
end sipo;
```

### 3.4.2 Signals

```
signal intermediate: std_logic_vector(N-1 downto 0);
```

An intermediate signal is used to shift properly the N bits of the signal to the right.

### 3.4.3 Code

```
sipo_process: process (clk, rst)
begin
    if (rst = '0') then -- asynchronous low reset
        intermediate <= (others => '0');
    elsif (rising_edge(clk)) then -- when the clk is high and on a rising edge
        -- we shift by one bit the signal
        intermediate(N-1 downto 1) <= intermediate(N-2 downto 0);
        -- we insert the new bit
        intermediate(0) <= serial_input;
    end if;
end process;

parallel_output <= intermediate;
```

When the clock is active and on a rising edge, we right shift the signal in intermediate and we give to the less significant bit (since we send the value of the most significant bit in the PISO) of the intermediate signal the value of the input serial\_input. Then, we set the state of the parallel-output to the state of intermediate. When the reset is active (low), we set low the signal intermediate.

## 3.5 ADAPTER

### 3.5.1 Entity description

The Adapter is a component that parallelize the data of link 1 and link 2 using SIPO and parallelize the data of the data output link using PISO. It connects the parallelized signal to the mini-router.

```
entity Adapter is
Port(
    -- INPUT --
    -- CLK MANAGEMENT --
    clk : in std_logic;
    rst : in std_logic;

    -- LINK 1 --
    data1_serial: in std_logic;
    req1: in std_logic;

    -- LINK 2 --
    data2_serial: in std_logic;
    req2: in std_logic;

    -- OUTPUT --
    -- LINK 1 --
    grant1: out std_logic;

    -- LINK 2 --
    grant2: out std_logic;

    -- OUTLINK --
    valid: out std_logic;
    data_out_serial: out std_logic
);
end Adapter;
```

### 3.5.2 Components

It is composed with eight sub-components:

- registers for the serial input and output signals (3)
- registers for the inputs and outputs: req1, req2, grant1, grant2 (4)
- a PISO for the output link (1)
- SIPOs for the input link (2)
- a register for valid in order to use it as a load in the PISO (1)
- a mini-wrapper (1)

### 3.5.3 Signals

Signals are used for the parallelized signal and for the serial intermediate of the serial inputs and outputs.

An intermediate signal is also use for valid because we want to use its state as input for the load of the PISO register. Intermediate signals are also used for the inputs and outputs to be registered.

```
-- LINK 1 --
signal data1_parallel: std_logic_vector(9 downto 0);
signal data1_serial_intermediate: std_logic;
signal req1_intermediate: std_logic;
signal grant1_intermediate : std_logic;

-- LINK 2 --
signal data2_parallel: std_logic_vector(9 downto 0);
signal data2_serial_intermediate: std_logic;
signal req2_intermediate: std_logic;
signal grant2_intermediate : std_logic;

-- VALID --
signal valid_intermediate : std_logic;

-- DATA OUTPUT --
signal data_out_parallel: std_logic_vector(7 downto 0);
signal data_out_intermediate: std_logic;
```

## 3.5.4 Code

```

-- REGISTERS FOR SERIAL INPUT --
ff_data1_serial: d_flip_flop
  port map (
    clk => clk,
    rst => rst,
    d => data1_serial,
    q => data1_serial_intermediate
  );

ff_data2_serial: d_flip_flop
  port map (
    clk => clk,
    rst => rst,
    d => data2_serial,
    q => data2_serial_intermediate
  );

-- REGISTER FOR SERIAL OUTPUT --
ff_data_out_serial: d_flip_flop
  port map (
    clk => clk,
    rst => rst,
    d => data_out_intermediate,
    q => data_out_serial
  );

-- REGISTER FOR VALID --
ff_valid: d_flip_flop
  port map (
    clk => clk,
    rst => rst,
    d => valid_intermediate,
    q => valid
  );

-- REGISTERED INPUT --
-- of req1 --
ff_req1: d_flip_flop
  port map (
    clk => clk,
    rst => rst,
    d => req1,
    q => req1_intermediate
  );

-- of req2 --
ff_req2: d_flip_flop
  port map (
    clk => clk,
    rst => rst,
    d => req2,
    q => req2_intermediate
  );

-- of data1 --
ff_data1: d_flip_flop
  port map (
    clk => clk,
    rst => rst,
    d => data1_serial,
    q => data1_serial_intermediate
  );

-- of data2 --
ff_data2: d_flip_flop
  port map (
    clk => clk,
    rst => rst,
    d => data2_serial,
    q => data2_serial_intermediate
  );

-- REGISTERED OUTPUT --
-- of grant1 --
ff_grant1: d_flip_flop
  port map (
    clk => clk,
    rst => rst,
    d => grant1_intermediate,
    q => grant1
  );

-- of grant2 --
ff_grant2: d_flip_flop
  port map (
    clk => clk,
    rst => rst,
    d => grant2_intermediate,
    q => grant2
  );

```

The serial input and output signals are registered in order to improve the data delay and ensure the synchronicity of the whole design.

The inputs req1, req2, data1\_serial, data2\_serial and the output grant1 and grant2 are registered in de flip flop for the same reason.



```

-- SIPO --
sipo_data1: sipo
generic map ( N => 10)
port map (
    clk => clk,
    rst => rst,
    serial_input => data1_serial_intermediate,
    parallel_output => data1_parallel
);

sipo_data2: sipo
generic map ( N => 10)
port map (
    clk => clk,
    rst => rst,
    serial_input => data2_serial_intermediate,
    parallel_output => data2_parallel
);

-- PISO --
piso_data_out: piso
generic map ( N => 8)
port map (
    clk => clk,
    rst => rst,
    load => valid_intermediate,
    parallel_input => data_out_parallel,
    serial_output => data_out_intermediate
);

-- Mini-Router --
mini_router: MiniRouter
port map(
    clk => clk,
    rst => rst,
    data1 => data1_parallel,
    req1 => req1_intermediate,
    data2 => data2_parallel,
    req2 => req2_intermediate,
    grant1 => grant1_intermediate,
    grant2 => grant2_intermediate,
    valid => valid_intermediate,
    data_out => data_out_parallel
);

```

The serial data inputs from link 1 and link 2 are converting to parallel signal using two SIPO components.

A PISO component is used to convert the parallel data\_out signal into a serial signal. We use valid\_intermediate as a load. The data is sent out only when valid\_intermediate is set high. Valid must be set high only when all the bits are set inside the PISO component.

The last component of the adapter is MiniRouter, the core of this project. As we can see, we connect the intermediate signals and the parallel data input and output to its ports.

## 3.6 MINI-ROUTER

### 3.6.1 Entity description

The entity MiniRouter has the inputs:

- clk and rst for synchronized the block;
- req1: in std\_logic, so the request for link 1;
- req2: in std\_logic, so the request for link2;
- data1: in std\_logic\_vector(9 downto 0), so the data and the priority bit of the link;
- data2: in std\_logic\_vector(9 downto 0), so the data and the priority bit of the link;

It has the outputs:

- valid: out std\_logic, that says if the output link receives data from an input link or not
- grant1: out std\_logic, that says if the link 1 was selected
- data\_out: out std\_logic\_vector(7 downto 0), the data of the output link
- grant2: out std\_logic, that says if the link2 was selected

```

entity MiniRouter is
    Port(
        --- INPUT ---
        -- CLK MANAGEMENT --
        clk : in std_logic;
        rst : in std_logic;

        -- LINK 1 --
        data1: in std_logic_vector(9 downto 0);
        req1: in std_logic;

        -- LINK 2 --
        data2: in std_logic_vector(9 downto 0);
        req2: in std_logic;

        --- OUTPUT ---
        -- LINK 1 --
        grant1: out std_logic;

        -- LINK 2 --
        grant2: out std_logic;

        -- OUTLINK --
        valid: out std_logic;
        data_out: out std_logic_vector(7 downto 0)
    );
end MiniRouter;

```

### 3.6.2 Components

It also has five types of components.

```

component Decoder is
  Port(
    --- INPUT ---
    data10_bits: in std_logic_vector(9 downto 0);

    --- OUTPUT ---
    data8_bits: out std_logic_vector(7 downto 0);
    data0: out std_logic;
    data3: out std_logic
  );
end component;

component Multiplexer is
  Port(
    --- INPUT ---
    -- LINK 1 --
    data1: in std_logic_vector(7 downto 0);

    -- LINK 2 --
    data2: in std_logic_vector(7 downto 0);

    -- RR ARBITER --
    valid: in std_logic;
    link: in std_logic;

    --- OUTPUT ---
    -- LINK 1 --
    grant1: out std_logic;

    -- LINK 2 --
    grant2: out std_logic;

    -- DATA --
    data_out: out std_logic_vector(7 downto 0)
  );
end component;

component RoundRobin is
  Port(
    --- INPUT ---
    -- CLOCK MANAGEMENT --
    clk: in std_logic;
    rst: in std_logic;
    -- LINK 1 --
    req1: in std_logic;
    data01: in std_logic;
    data31: in std_logic;
    -- LINK 2 --
    req2: in std_logic;
    data02: in std_logic;
    data32: in std_logic;

    --- OUTPUT ---
    -- LINK --
    valid: out std_logic;
    link: out std_logic
  );
end component;

```

The Decoder component is used to retrieve the priority bits and the data from data1 and data2.

The Multiplexer component is used to direct an input link to the output link.

The RoundRobin component is used to decide between link 1 and link 2 which link must be selected or not to be transferred.

We also use d flip flop and d flip flop with n bits to have registered outputs and inputs.

### 3.6.3 Signals

We use an intermediate signal for each input and output. They have the same name of the in/output with \_intermediate.

We also have other signals:

- signal data1\_8\_bits, data2\_8\_bits:  
std\_logic\_vector(7 downto 0);

These signals are the data from data1 and data2 without the two priority bits.

- signal data01, data02, data31, data32:  
std\_logic;

These signals are the priority bits of both links.

```

-- LINK 1 --
signal data1_intermediate: std_logic_vector(9 downto 0);
signal req1_intermediate: std_logic;
signal data1_8_bits : std_logic_vector(7 downto 0);
signal data01 : std_logic;
signal data31 : std_logic;
signal grant1_intermediate : std_logic;

-- LINK 2 --
signal data2_intermediate: std_logic_vector(9 downto 0);
signal req2_intermediate: std_logic;
signal data2_8_bits : std_logic_vector(7 downto 0);
signal data02 : std_logic;
signal data32 : std_logic;
signal grant2_intermediate : std_logic;

-- ARBITER --
signal valid_intermediate : std_logic;
signal link : std_logic;

-- DATA OUTPUT --
signal data_out_intermediate : std_logic_vector(7 downto 0);

```



signal link: std\_logic;

This signal is used to know which link is selected. If link is high, we select link 2, otherwise we select link 1.

### 3.6.4 Code

```
-- REGISTERED INPUT --
-- of req1 --
ff_req1: d_flip_flop
  port map (
    clk => clk,
    rst => rst,
    d => req1,
    q => req1_intermediate
  );

-- of data1 --
ff_10_data1: d_flip_flop_n
  generic map ( N => 10)
  port map (
    clk => clk,
    rst => rst,
    d => data1,
    q => data1_intermediate
  );

-- of req 2 --
ff_req2: d_flip_flop
  port map (
    clk => clk,
    rst => rst,
    d => req2,
    q => req2_intermediate
  );

-- of data2 --
ff_10_data2: d_flip_flop_n
  generic map ( N => 10)
  port map (
    clk => clk,
    rst => rst,
    d => data2,
    q => data2_intermediate
  );
```

We register the inputs. This step is essential for synchronize the design and improve the data delay and then the performance. We do it for req1, req2, data1 and data2.

We decode each input link in order to retrieve the data and the priority bits.

```
-- DECODER --
-- we will retrieve the priority bits (0 and 3) and the data without the priority section for both links
-- Link 1
d_link1: Decoder
  port map (
    data_10_bits => data1_intermediate, -- input data for link 1
    data_8_bits => data1_8_bits, -- output data for link 1 if link 1 is chosen
    data0 => data01, -- second priority level
    data3 => data31 -- first priority level
  );

-- Link 2
d_link2: Decoder
  port map (
    data_10_bits => data2_intermediate, -- input data for link 2
    data_8_bits => data2_8_bits, -- output data for link 2 if link 2 is chosen
    data0 => data02, -- second priority level
    data3 => data32 -- first priority level
  );
```

We route the correct data link to the output using the Multiplexer. When valid is equal to '1', we take the data of the link indicated by the signal link ('1' is link 2, '0' is link 1).

```
-- we route the correct data input to data output (if we have an input link to route, otherwise data_out is set to "00000000")
mux: Multiplexer
  port map (
    data1 => data1_8_bits, -- data input for link 1
    data2 => data2_8_bits, -- data input for link 2
    valid => valid_intermediate, -- validity : '1' if output data is present, '0' otherwise
    link => link, -- chosen link : '0' if link 1, '1' if link 2
    grant1 => grant1_intermediate, -- '1' if link 1 is chosen
    grant2 => grant2_intermediate, -- '1' if link 2 is chosen
    data_out => data_out_intermediate -- data output => equal to the data1 or to data2 or null
  );
```

We use the request signals req1 and req2, and the priority bits in order to see which link must be selected. We use a RoundRobin algorithm to let the arbiter who decide between both link in case of data conflict.

```
-- PRIORITY AND ARBITER --
-- we use a Round Robin Arbiter with two priority levels
rr_arbiter: RoundRobin
  port map (
    clk => clk,
    rst => rst,
    req1 => req1_intermediate, -- request for link 1
    data01 => data01, -- second priority level of data1
    data31 => data31, -- first priority level of data1
    req2 => req2_intermediate, -- request for link 2
    data02 => data02, -- second priority level of data2
    data32 => data32, -- first priority level of data2
    valid => valid_intermediate, -- validity : '1' if output data is present, '0' otherwise
    link => link -- chosen link : '0' if link 1, '1' if link 2
  );

-- REGISTERED OUTPUT --
-- of valid --
ff_valid: d_flip_flop
  port map (
    clk => clk,
    rst => rst,
    d => valid_intermediate,
    q => valid
  );

-- of grant1 --
ff_grant1: d_flip_flop
  port map (
    clk => clk,
    rst => rst,
    d => grant1_intermediate,
    q => grant1
  );

-- of grant2 --
ff_grant2: d_flip_flop
  port map (
    clk => clk,
    rst => rst,
    d => grant2_intermediate,
    q => grant2
  );

-- of data_out --
ff_8_data_out: d_flip_flop_n
  generic map ( N => 8)
  port map (
    clk => clk,
    rst => rst,
    d => data_out_intermediate,
    q => data_out
  );
```

And we register the outputs for improving the design performance, as we did for the inputs, with d flip flop or with d flip flop of n bits. The registered outputs are valid, grant1, grant2 and data\_out.

### 3.7 DECODER (FOR A LINK)

The Decoder is used to retrieve the priority bits and discard them from the data of each input links.

#### 3.7.1 Entity description

The entity Decoder has one input, data\_10\_bits: in std\_logic\_vector(9 downto 0);

This input is the data link. It is composed by 8 bits of data and 2 priority bits.

It has three outputs:

- data\_8\_bits: out std\_logic\_vector(7 downto 0); so, the data without the priority bits.
- data0, data3: out std\_logic; so, the two priority bits (number 0 and number 3)

```
entity Decoder is
  Port(
    --- INPUT ---
    data_10_bits: in std_logic_vector(9 downto 0);

    --- OUTPUT ---
    data_8_bits: out std_logic_vector(7 downto 0);
    data0: out std_logic;
    data3: out std_logic
  );
end Decoder;
```

### 3.7.2 Code

We take the value of the bit number 0 and of the bit number 3 of the input to set data0 and data3. We also composed the output data with the concatenation of the bits of the input from the most significative bit to the fourth and from the second to the less significative bit.

```
architecture rtl of Decoder is
begin
    -- We retrieve the low-priority bit
    data0 <= data_10_bits(0);
    -- We retrieve the high-priority bit
    data3 <= data_10_bits(3);
    -- The priority bits (0 and 3) are removed from the data bus.
    data_8_bits <= data_10_bits(9 downto 4) & data_10_bits(2 downto 1);
end rtl;
```

## 3.8 ROUND ROBIN ARBITER

I firstly use a process in order to implement the Round-Robin Algorithm. It was the following:

```
round_robin_process: process (req1, req2, data31, data32, data01, data02, arbiter_in)
begin
    if (req1 = '1') and (req2 = '0') then
        -- req signal of link 1 is high
        valid <= '1';
        link <= '0'; -- link 1 is choosen
    elsif (req1 = '0') and (req2 = '1') then
        -- req signal of link 2 is high
        valid <= '1';
        link <= '1'; -- link 2 is choosen
    elsif (req1 = '1') and (req2 = '1') then
        -- both req signals of link 1 and link 2 are high
        -- so we need to decide which one will be selected
        -- FIRST LEVEL OF PRIORITY
        valid <= '1';
        if (data31 = '1') and (data32 = '0') then
            -- link 1 has a higher priority than link 2
            link <= '0'; -- link 1 is choosen
        elsif (data31 = '0') and (data32 = '1') then
            -- link 2 has a higher priority than link 1
            link <= '1'; -- link 2 is choosen
        else
            -- SECOND LEVEL OF PRIORITY
            if (data01 = '1') and (data02 = '0') then
                -- link 1 has a higher priority than link 2
                link <= '0'; -- link 1 is selected
            elsif (data01 = '0') and (data02 = '1') then
                -- link 2 has a higher priority than link 1
                link <= '1'; -- link 2 is selected
            else
                -- DATA CONFLICT
                -- link 1 and link 2 have the same priority
                link <= arbiter_in;
                arbiter <= '1';
            end if;
        end if;
    else
        valid <= '0';
    end if;

    -- when arbiter_in = '0', link 1 is selected
    -- when arbiter_in = '1', link 2 is selected
    -- we change the next link to be chosen for the next conflict
    if (arbiter = '1') then arbiter_out <= not(arbiter_in);
    else arbiter_out <= arbiter_in;
    end if;
end process;
```

This code was then composed with three levels of 'if'. It was decided, in order to optimize it to create a Look-Up Table that resume all the possible choices. The advantage of a LUT is that computing the function only takes a single memory lookup regardless of the complexity of the function. It is then very fast. The disadvantage is that it takes memory.

### 3.8.1 Entity description

The entity RoundRobin has the inputs:

- clk and rst for synchronized the block;
- req1: in std\_logic, so the request for link 1;
- req2: in std\_logic, so the request for link2;

- data01, data31: in std\_logic, so the priority bits of link1 (bit number 0 and bit number 3).
- data02, data32: in std\_logic, so the priority bits of link2

It has the outputs:

- valid: out std\_logic, that says if the output link receives data from an input link or not
- link: out std\_logic, that informs which link is selected

```
entity RoundRobin is
  Port(
    -- INPUT --
    -- CLOCK MANAGEMENT --
    clk: in std_logic;
    rst: in std_logic;
    -- LINK 1 --
    req1: in std_logic;
    data01: in std_logic;
    data31: in std_logic;
    -- LINK 2 --
    req2: in std_logic;
    data02: in std_logic;
    data32: in std_logic;

    -- OUTPUT --
    -- LINK --
    valid: out std_logic;
    link: out std_logic
  );
end RoundRobin;
```

### 3.8.2 Components

I use two components in RoundRobin. Firstly, the LUT and secondly, a d flip flop. The d flip flop is used to keep during a clock cycle the value of the arbiter, which is a signal of RoundRobin.

```
component LUT is
  Port(
    lut_in : in std_logic_vector(6 downto 0);
    lut_out : out std_logic_vector(2 downto 0)
  );
end component;
```

### 3.8.3 Signals

```
signal arbiter: std_logic;
signal lut_in: std_logic_vector(6 downto 0);
signal lut_out: std_logic_vector(2 downto 0);
```

We use three different signals. One for the arbiter and two for using the LUT of the arbiter. We feed the LUT component with these two signals:

- lut\_in: in std\_logic\_vector(6 downto 0)

It is composed by 7 bits. These bits are the followings :

6 (MSB)	5	4	3	2	1	0 (LSB)
req1	req2	data31	data32	data01	data02	arbiter

- lut\_out: out std\_logic\_vector(2 downto 0)

It is only composed by 3 bits which are the followings:

2	1	0 (LSB)
Valid	Link	arbiter

The arbiter is set to '0' after the reset. Its value is storing into a d flip flop during a clock cycle. It is used to form lut\_in and is then used to calculate the link to be selected. If no conflict occurs, the arbiter keeps its state. Otherwise, the arbiter changes its state thanks to the LUT result.

### 3.8.4 Code

As you can see in the code, we concatenate req1, req2, data31, data32, data01, data02 and the arbiter to form lut\_in. We also retrieve the value of valid and link using lut\_out.

```
-- lut_in is the concatenation of req1, req2, data31, data32, data01, data02 and arbiter
lut_in <= req1 & req2 & data31 & data32 & data01 & data02 & arbiter;

-- lut_out give us valid, link and arbiter
valid <= lut_out(2);
link <= lut_out(1);
```

We find the state of lut\_out using lut\_in and the LUT component.

```
-- we retrieve lut_out using lut_in
rr_lut : LUT
  port map (
    lut_in => lut_in,
    lut_out => lut_out
  );
```

And we store the arbiter state ('0' if the next link to be selected is link1 and '1' otherwise) in a d\_flip\_flop, updating its state with the result of the LUT retrieval.

```
-- we keep the arbiter in a d flip flop for the next time
ff_arbiter : d_flip_flop
  port map (
    clk => clk,
    rst => rst,
    d => lut_out(0),
    q => arbiter
  );
```

### 3.9 LUT (FOR THE ROUND-ROBIN ALGORITHM)

#### 3.9.1 Entity description

```
entity LUT is
  Port(
    lut_in : in std_logic_vector(6 downto 0);
    lut_out : out std_logic_vector(2 downto 0)
  );
end LUT;
```

The LUT has only one input and one output signals:

- lut\_in: in std\_logic\_vector(6 downto 0)
- lut\_out: out std\_logic\_vector(2 downto 0)

#### 3.9.2 Signals

```
signal lut_in_integer : integer range 0 to 127;
type lut_t is array (natural range<>) of std_logic_vector(2 downto 0);
```

In order to do the conversion between the table and the input value, we use an integer. The signal lut\_in\_integer is an integer that can take the value from 0 to 127 ( $2^7-1$ ). This value is the range of the array lut\_t which contain the corresponding output of 3 bits. lut\_t contains all the possible output that lut\_out can take.

Input								Output			
req1	req2	data31	data32	data01	data02	arbiter	input	valid	link	arbiter2	output
0	0	0	0	0	0	0	"0000000"	0	0	0	"000"
0	0	0	0	0	0	1	"0000001"	0	0	1	"001"
0	0	0	0	0	1	0	"0000010"	0	0	0	"000"
0	0	0	0	0	1	1	"0000011"	0	0	1	"001"
0	0	0	0	1	0	0	"0000100"	0	0	0	"000"
0	0	0	0	1	0	1	"0000101"	0	0	1	"001"
0	0	0	0	1	1	0	"0000110"	0	0	0	"000"
0	0	0	0	1	1	1	"0000111"	0	0	1	"001"
0	0	0	1	0	0	0	"0001000"	0	0	0	"000"
0	0	0	1	0	0	1	"0001001"	0	0	1	"001"
0	0	0	1	0	1	0	"0001010"	0	0	0	"000"
0	0	0	1	0	1	1	"0001011"	0	0	1	"001"
0	0	0	1	1	0	0	"0001100"	0	0	0	"000"
0	0	0	1	1	0	1	"0001101"	0	0	1	"001"
0	0	0	1	1	1	0	"0001110"	0	0	0	"000"
0	0	0	1	1	1	1	"0001111"	0	0	1	"001"
0	0	1	0	0	0	0	"0010000"	0	0	0	"000"
0	0	1	0	0	0	1	"0010001"	0	0	1	"001"
0	0	1	0	0	1	0	"0010010"	0	0	0	"000"
0	0	1	0	0	1	1	"0010011"	0	0	1	"001"
0	0	1	0	1	0	0	"0010100"	0	0	0	"000"
0	0	1	0	1	0	1	"0010101"	0	0	1	"001"
0	0	1	0	1	1	0	"0010110"	0	0	0	"000"
0	0	1	0	1	1	1	"0010111"	0	0	1	"001"
0	0	1	1	0	0	0	"0011000"	0	0	0	"000"
0	0	1	1	0	0	1	"0011001"	0	0	1	"001"
0	0	1	1	0	1	0	"0011010"	0	0	0	"000"
0	0	1	1	0	1	1	"0011011"	0	0	1	"001"
0	0	1	1	1	0	0	"0011100"	0	0	0	"000"
0	0	1	1	1	0	1	"0011101"	0	0	1	"001"
0	0	1	1	1	1	0	"0011110"	0	0	0	"000"
0	0	1	1	1	1	1	"0011111"	0	0	1	"001"
0	1	0	0	0	0	0	"0100000"	1	1	0	"110"
0	1	0	0	0	0	1	"0100001"	1	1	1	"111"
0	1	0	0	0	1	0	"0100001"	1	1	0	"110"

The LUT was firstly written on a Excel file to be sure that no possibility was forgotten. The Excel file is in the zip and it is called lut.xlsx.

### 3.9.3 Code

```
constant lut : lut_t (0 to 127) := (
    "000",
    "001",
    "000",
    "001",
    "000",
    "000",
    "001",
    "000",
    "001",
    "000",
    "001",
    "000",
    "001",
    "000",
    "001",
    "000",
    ...
    "101",
    "100",
    "101",
    "100",
    "101",
    "101",
    "110",
    "110",
    "111",
    "100",
    "101",
    "101",
    "110"
);

begin
    -- we convert the 7 bits input in an integer from 0 to 127.
    -- 0 to 127 is the range of the 3 bits value in the look-up table.
    lut_in_integer <= TO_INTEGER(unsigned(lut_in));
    -- we retrieve in the look-up table the element with the range previously computed
    lut_out <= lut(lut_in_integer);
end rtl;
```

We convert the 7-bits input into an integer and use this integer to retrieve the output value. We use the function TO\_INTEGER() in order to convert the input (we precise that it is unsigned using the unsigned() function).

## 3.10 MULTIPLEXER (FOR DIRECTING THE SELECTED INPUT LINK TO THE OUTPUT LINK)

### 3.10.1 Entity description

```
entity Multiplexer is
    Port(
        --- INPUT ---
        -- LINK 1 --
        data1: in std_logic_vector(7 downto 0);

        -- LINK 2 --
        data2: in std_logic_vector(7 downto 0);

        -- RR ARBITER --
        valid: in std_logic;
        link: in std_logic;

        --- OUTPUT ---
        -- LINK 1 --
        grant1: out std_logic;

        -- LINK 2 --
        grant2: out std_logic;

        -- DATA --
        data_out: out std_logic_vector(7 downto 0)
    );
end Multiplexer;
```

- grant1, grant2: out std\_logic;

When the link1 is granted, we set grant1 to '1' and grant2 to '0'. Otherwise, we do the opposite. We inform which link was selected by the system with these two outputs.

- data\_out: out std\_logic\_vector(7 downto 0);

The multiplexer is an entity that direct the selected input link to the output link. It uses the results of the RoundRobin block: valid and link. The value of these later allows us to set the output.

The inputs are:

- data1, data2: in std\_logic\_vector(7 downto 0);

They are the data of link 1 and link 2 without the priority bits.

- valid, link: in std\_logic;

valid represents the fact that a link is directed to the output or not. It is set to '1' if we have a link to be transferred. link gives which link was selected ('1' is link 2 and '0' is link 1).



This is the data output. It takes the state of data1 or data2 according to the state of valid and link. If valid is not high, it is set to '00000000'.

### 3.10.2 Signal

```
signal valid_link: std_logic_vector(1 downto 0);
```

We use an intermediate signal to concatenate valid and link in order to simplify the condition in the process of the multiplexer.

### 3.10.3 Code

```
begin
  valid_link <= valid & link; -- we concatenate valid and link for simplicity
  multiplexer_process: process(valid_link, data1, data2)
  begin
    case valid_link is
      when "00" => -- no link is selected
        data_out <= "00000000";
        grant1 <= '0';
        grant2 <= '0';
      when "01" => -- no link is selected
        data_out <= "00000000";
        grant1 <= '0';
        grant2 <= '0';
      when "10" => -- link 1 is selected
        data_out <= data1;
        grant1 <= '1';
        grant2 <= '0';
      when "11" => -- link 2 is selected
        data_out <= data2;
        grant1 <= '0';
        grant2 <= '1';
      when others => null;
    end case;
  end process;
end rtl;
```

We use a process to set the outputs. We put valid\_link, data1 and data2 in the sensitivity list because their states are necessary for the evaluation of the outputs. When valid is set low, the outputs are set low too. When valid is set high, the state of grant1, grant2 and data\_out depends on the state of link.

## 4 TEST PLAN

In order to be sure that the mini-router works well, we need to check the following parameters:

- The request of both signals: req1 and req2. We must check them when req1 = '0' and req2 = '1', when req1 = '1' and req2 = '0', when they are both high and when they are both low.
- The 10-bits length data inputs. We are obviously not taking into consideration the 8 bits of data since their values have no impact on the mini-router. Indeed, 8 bits of data means  $2^8$  possible combinations and confront the both 8 bits of data would mean doing  $2^{16}$  experiments, which is impossible and irrelevant. We have then the two remaining bits of priority, that can have, if both req are high, an incidence.

Therefore, the following cases are tested in order to check that the mini-router answer to all the mandatory requirements:

#### A. A link is selected because there is only one request:

Stimuli		Expected Output			
req1	req2	data output	valid	grant1	grant2
'1'	'0'	data1	'1'	'1'	'0'
'0'	'1'	data2	'1'	'0'	'1'

#### B. No link is selected because there is no request:

Stimuli		Expected Output			
req1	req2	data output	valid	grant1	grant2
'0'	'0'	"00000000"	'0'	'0'	'0'

- C. Both requests are emitted. We need to check the priority bit and decide between links which one is forwarded.

- a. First level of priority: bit number 3

Stimuli		Expected Output			
Bit number 3 data 1	Bit number 3 data 2	data_output	valid	grant1	grant2
'1'	'0'	data1	'1'	'1'	'0'
'0'	'1'	data2	'1'	'0'	'1'

- b. Second level of priority: bit number

Stimuli				Expected Output			
Bit number 3 data 1	Bit number 3 data 2	Bit number 0 data 1	Bit number 0 data 2	data_output	valid	grant1	grant2
'0'	'0'	'1'	'0'	data1	'1'	'1'	'0'
'0'	'0'	'0'	'1'	data2	'1'	'0'	'1'
'1'	'1'	'1'	'0'	data1	'1'	'1'	'0'
'1'	'1'	'0'	'1'	data2	'1'	'0'	'1'

- c. Data conflict: arbiter

Stimuli				Arbiter			Expected Output			
Bit number 3 data 1	Bit number 3 data 2	Bit number 0 data 1	Bit number 0 data 2	Number of the conflict	Value of arbiter		data_output	valid	grant1	grant2
					Before	After				
'0'	'0'	'0'	'0'	1 <sup>st</sup>	'0'	'1'	data1	'1'	'1'	'0'
'0'	'0'	'1'	'1'	2 <sup>nd</sup>	'1'	'0'	data2	'1'	'0'	'1'
'1'	'1'	'0'	'0'	3 <sup>rd</sup>	'0'	'1'	data1	'1'	'1'	'0'
'1'	'1'	'1'	'1'	4 <sup>th</sup>	'1'	'0'	data2	'1'	'0'	'1'

All these tests are realized in Modelsimn, using testbenches, for the following entities:

- The Round-Robin entity;
- The Mini-Router entity;

The adapter is also tested but since the input must be serial, doing all these tests would be too loud. Thus, I only test that a link is correctly forwarding when the request is high, if the mini-router works for all case and if the adapter works for this situation, then, it would work for every situation. Since I use register inside the Mini-Router, a delay is observed for the output. I need then to take that into consideration and send the request only when the 10 bits of the input are emitted. Other testbenches are done to verify that each block functions well.

## 5 TESTBENCH

### 5.1 SIPO

```
case(t) is -- OUTPUT : "11011010"
  when 0 => si_tb <= '1';
  when 1 => si_tb <= '1';
  when 2 => si_tb <= '0';
  when 3 => si_tb <= '1';
  when 4 => si_tb <= '1';
  when 5 => si_tb <= '0';
  when 6 => si_tb <= '1';
  when 7 => si_tb <= '0';
  when others => null;
end case;
t := t+1;
```

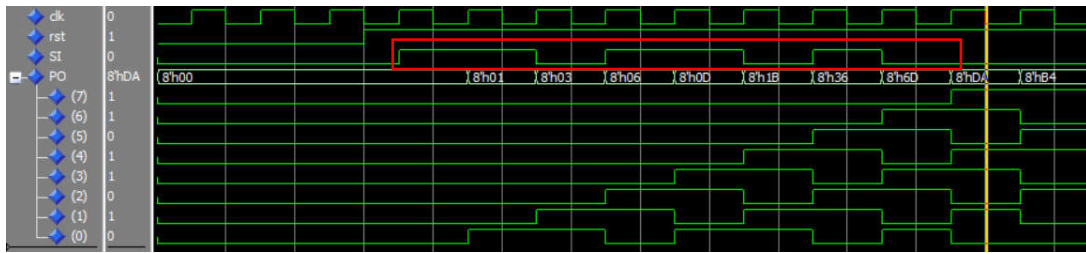
In order to verify that the SIPO is working, we forward a serial signal and we visualize if the output is correctly parallelized. To ensure that the order of the data is correct, a non-symmetrical signal is used ("11011010").

Each clock cycle, we send a new bit, the MSB first and the LSB last.

The red rectangular framed shows the serialized input. On the yellow bar, we can see the correct parallelized output on the next clock cycle.



In the following graph could be seen the result of the test:



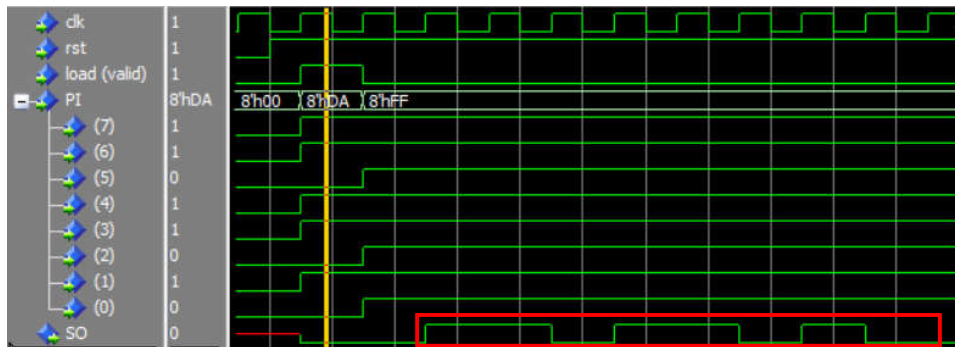
## 5.2 PISO

```
case(t) is
  when 0 =>
    pi_tb <= "11011010";
    load_tb <= '1';
  when 1 =>
    load_tb <= '0';
    pi_tb <= "11111111";
  when others => null;
end case;
t := t+1;
```

To test the PISO, we forward a parallel signal and we visualize if the output is correctly serialized. To ensure that the order of the data is correct, a non-symmetrical signal is used (“11011010”, the same that for the SIPO).

The red rectangular framed shows the serialized input. On the yellow bar, we can see the correct parallelized output on the next clock cycle. The yellow bar is placed on the parallel signal.

In the following graph, made on Modelsim, we can see the result of this test:



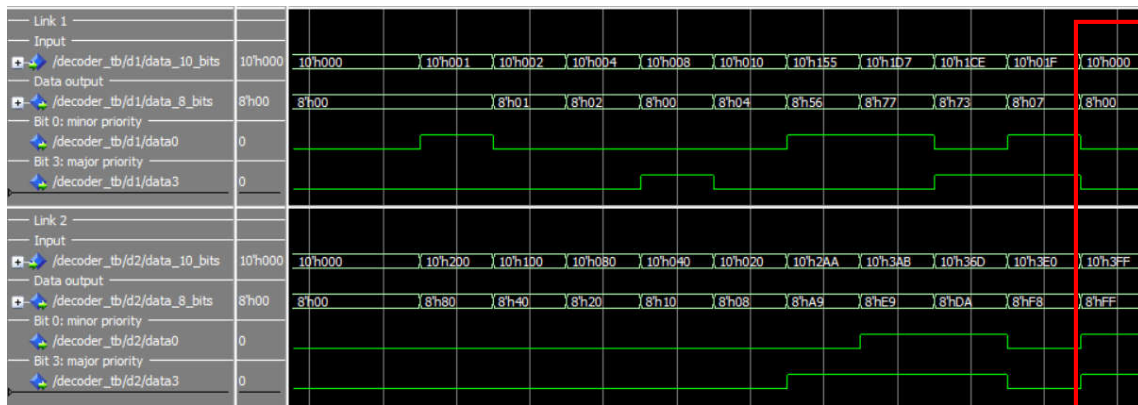
## 5.3 DECODER (FOR A LINK)

For verifying the decoder, I make more experiments than necessary. I use different data inputs and the objective is to retrieve the correct 8-bits-length output and the good priorities bits.

The testbench has the following events:

```
case(t) is
  when 0 =>
    data1_10_bits_tb <= "0000000001";
    data2_10_bits_tb <= "1000000000";
    -- RESULT EXPECTED : data1_8_bits_tb <= "00000000" ;
    --                      data2_8_bits_tb <= "10000000"
  when 1 =>
    data1_10_bits_tb <= "0000000010";
    data2_10_bits_tb <= "0100000000";
    -- RESULT EXPECTED : data1_8_bits_tb <= "00000001" ;
    --                      data2_8_bits_tb <= "01000000"
  when 2 =>
    data1_10_bits_tb <= "0000000100";
    data2_10_bits_tb <= "0010000000";
    -- RESULT EXPECTED : data1_8_bits_tb <= "00000010" ;
    --                      data2_8_bits_tb <= "00100000"
  when 3 =>
    data1_10_bits_tb <= "0000001000";
    data2_10_bits_tb <= "0001000000";
    -- RESULT EXPECTED : data1_8_bits_tb <= "00000000" ;
    --                      data2_8_bits_tb <= "00010000"
  when 4 =>
    data1_10_bits_tb <= "0000001000";
    data2_10_bits_tb <= "0000100000";
    -- RESULT EXPECTED : data1_8_bits_tb <= "00000100" ;
    --                      data2_8_bits_tb <= "00001000"
  when 5 =>
    data1_10_bits_tb <= "0101010101";
    data2_10_bits_tb <= "1010101010";
    -- RESULT EXPECTED : data1_8_bits_tb <= "01010110" ;
    --                      data2_8_bits_tb <= "10101001"
  when 6 =>
    data1_10_bits_tb <= "0111010111";
    data2_10_bits_tb <= "1110101011";
    -- RESULT EXPECTED : data1_8_bits_tb <= "01110111" ;
    --                      data2_8_bits_tb <= "11101001"
  when 7 =>
    data1_10_bits_tb <= "0111001110";
    data2_10_bits_tb <= "1101101101";
    -- RESULT EXPECTED : data1_8_bits_tb <= "01110011" ;
    --                      data2_8_bits_tb <= "11011010"
  when 8 =>
    data1_10_bits_tb <= "0000011111";
    data2_10_bits_tb <= "1111100000";
    -- RESULT EXPECTED : data1_8_bits_tb <= "00000111" ;
    --                      data2_8_bits_tb <= "11111000"
  when 9 =>
    data1_10_bits_tb <= "0000000000";
    data2_10_bits_tb <= "1111111111";
    -- RESULT EXPECTED : data1_8_bits_tb <= "00000000" ;
    --                      data2_8_bits_tb <= "11111111"
  when others => null;
```

The results of the testbench can be seen on the following graph, obtained thanks to Modelsim. As we can see, we retrieve correctly the value that we want to retrieve. For instance, for the last period, we obtain from “0000000000” and from “1111111111” respectively “00000000” and two priority bits equal to ‘0’ and “11111111” and two priority bits equal to ‘1’ (red rectangular framed). This example is framed in red.



#### 5.4 MULTIPLEXER (FOR DIRECTING THE SELECTED INPUT LINK TO THE OUTPUT LINK)

There are four possibilities with the multiplexer:

```

elseif (rising_edge(clk_tb)) then
  case(t) is
    when 0 =>
      data1_tb <= "10101010";
      data2_tb <= "01010101";
      valid_tb <= '1';
      link_tb <= '0';
      -- RESULT EXPECTED : data_out_tb <= "10101010" ; grant1_tb <= '1' ; grant2_tb <= '0'

    when 1 =>
      data1_tb <= "10101010";
      data2_tb <= "01010101";
      valid_tb <= '1';
      link_tb <= '1';
      -- RESULT EXPECTED : data_out_tb <= "01010101" ; grant1_tb <= '0' ; grant2_tb <= '1'

    when 2 =>
      data1_tb <= "10101010";
      data2_tb <= "01010101";
      valid_tb <= '0';
      link_tb <= '0';
      -- RESULT EXPECTED : data_out_tb <= "00000000" ; grant1_tb <= '0' ; grant2_tb <= '0'

    when 3 =>
      data1_tb <= "10101010";
      data2_tb <= "01010101";
      valid_tb <= '0';
      link_tb <= '0';
      -- RESULT EXPECTED : data_out_tb <= "00000000" ; grant1_tb <= '0' ; grant2_tb <= '0'

    when others => null;
  end case;
  t := t+1;
end if;
end process;

```

When valid = '1' and link = '0'. We expect to see the data\_output set with data1, grant2 in low-state and grant1 in high-state.

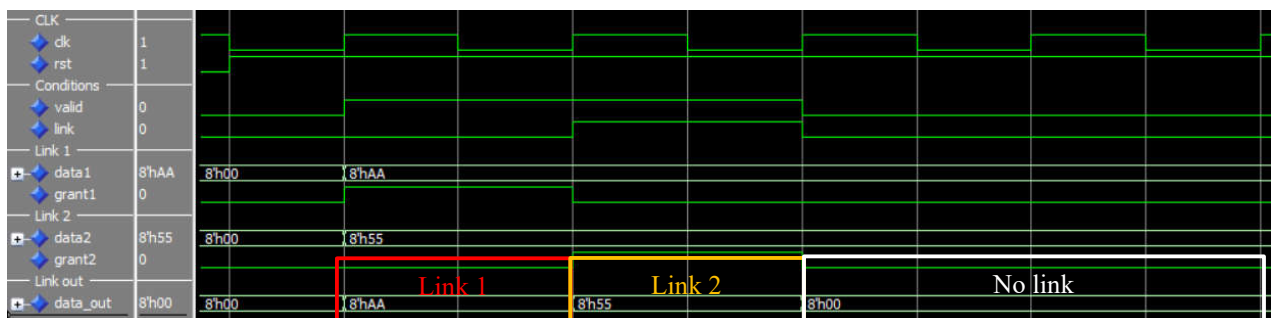
When `valid = '1'` and `link = '1'`. We expect to see the `data_output` set with `data2`, `grant1` in low-state and `grant2` in high-state.

When `valid = '0'` and `link = '1'`. We expect to see the `data_output` set to a low signal, and `grant1` and `grant2` equal to `'0'`.

- When valid = '0' and link = '0'. We expect the same as the previous case.

These possibilities are implemented in the testbench as follows and we can visualize if the results are the one that we are expected or not in the graph obtained from Modelsim.

We can see that data-out takes the correct value:



## 5.5 ROUND ROBIN ARBITER

In order to see if the Round Robin Arbiter works, I test the thirteen cases expressed in the part 4, Test Plan. For the recall, there are :

## A. A link is selected because there is only one request

The results of these cases are on the same graph as the case 3. We can see that, according to what we expected, the link selected when req1 = '1' and req2 = '0' is the first link (link = '0') in the first case. When we set req1 = '0' and req2 = '1', we obtain the contrary. valid is set high since a link can be forwarded.

```

--- REQ IS HIGH FOR ONLY ONE LINK-
-- 1ST CASE : req1 = '1' and req2 = '0'
when 0 =>
    req1_tb <= '1';
    req2_tb <= '0';
    -- RESULT EXPECTED : link_tb = '0' and valid <= '1'
-- 2ND CASE : req1 = '0' and req2 = '1'
when 1 =>
    req1_tb <= '0';
    req2_tb <= '1';
    -- RESULT EXPECTED : link_tb = '1' and valid <= '1'

```

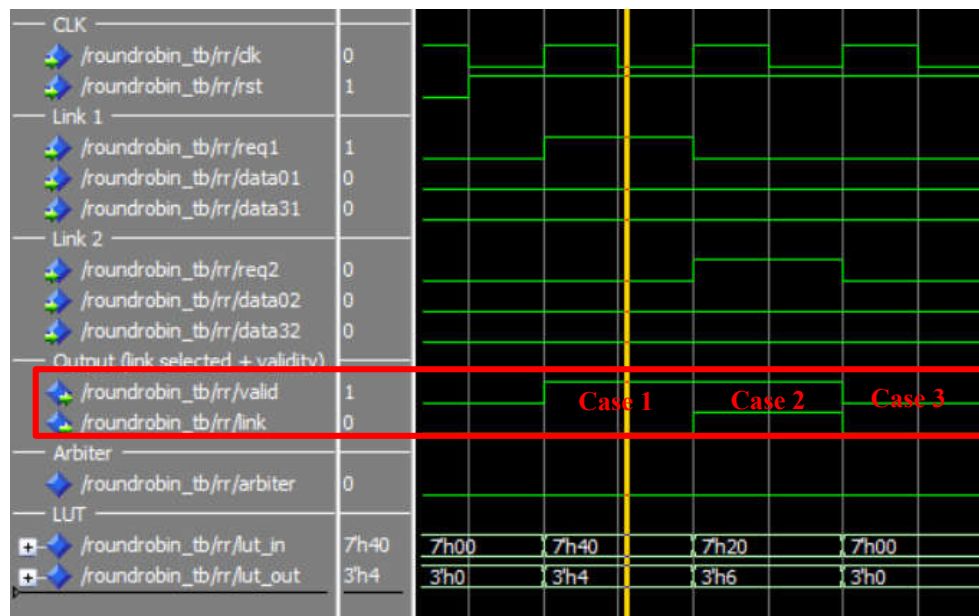
## B. No link is selected because there is no request

When both req are low, no request is done and then no link must be driven to the output. Therefore, we have valid and link set to '0'. This is the only case where valid can be set low.

```

--- REQ IS LOW FOR BOTH SIGNALS
-- 3RD CASE : req1 = '0' and req2 = '0'
when 2 =>
    req1_tb <= '0';
    req2_tb <= '0';
    -- RESULT EXPECTED : link_tb = '0' and valid <= '0'

```



## C. Both requests are emitted. We need to check the priority bit and decide between links which one is forwarded.

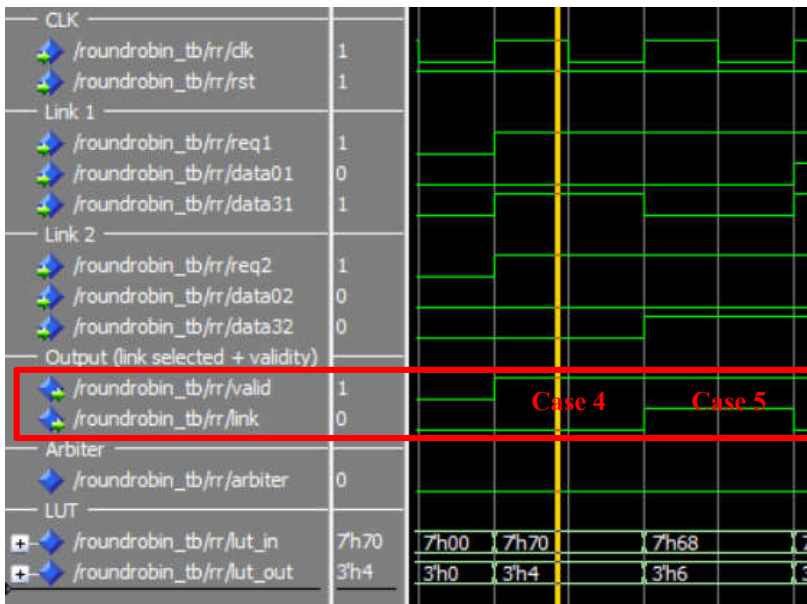
## a. First level of priority: bit number 3

When both req are high, we need to decide between the links which one will be selected. The first thing to do is to check the first level of priority. If only one of the links have the number 3 set to '1', we choose this link.

```

--- REQ IS HIGH FOR BOTH SIGNALS
-- FIRST LEVEL OF PRIORITY
-- 4TH CASE : req1 = '1' and req2 = '1'
--             data31 <= '1' and data32 <= '0'
when 3 =>
    req1_tb <= '1';
    req2_tb <= '1';
    data31_tb <= '1';
    data32_tb <= '0';
    -- RESULT EXPECTED : link_tb = '0' and valid <= '1'
-- 5TH CASE : req1 = '1' and req2 = '1'
--             data31 <= '0' and data32 <= '1'
when 4 =>
    req1_tb <= '1';
    req2_tb <= '1';
    data31_tb <= '0';
    data32_tb <= '1';
    -- RESULT EXPECTED : link_tb = '1' and valid <= '1'

```



In the graph above, we see in the 4<sup>th</sup> case that the link number 1 is selected (link = '0') since the bit number 3 of data1 is high and in the 5<sup>th</sup> case that the link number 2 is selected (link = '1') since the bit number 3 of data2 is high.

#### b. Second level of priority: bit number

However, both bit number 3 can have the same value. We then need to use the second level of priority to decide between link 1 and link 2 the link to drive out.

```
-- SECOND LEVEL OF PRIORITY
-- 6TH CASE : req1 = '1' and req2 = '1'
--             data31 <= '1' and data32 <= '1'
--             data01 <= '1' and data02 <= '0'
when 5 =>
    req1_tb <= '1';
    req2_tb <= '1';
    data31_tb <= '1';
    data32_tb <= '1';
    data01_tb <= '1';
    data02_tb <= '0';
    -- RESULT EXPECTED : link_tb = '0' and valid <= '1'

-- 7TH CASE : req1 = '1' and req2 = '1'
--             data31 <= '1' and data32 <= '1'
--             data01 <= '0' and data02 <= '1'
when 6 =>
    req1_tb <= '1';
    req2_tb <= '1';
    data31_tb <= '1';
    data32_tb <= '1';
    data01_tb <= '0';
    data02_tb <= '1';
    -- RESULT EXPECTED : link_tb = '1' and valid <= '1'

-- 8TH CASE : req1 = '1' and req2 = '1'
--             data31 <= '0' and data32 <= '0'
--             data01 <= '1' and data02 <= '0'
when 7 =>
    req1_tb <= '1';
    req2_tb <= '1';
    data31_tb <= '0';
    data32_tb <= '0';
    data01_tb <= '1';
    data02_tb <= '0';
    -- RESULT EXPECTED : link_tb = '0' and valid <= '1'

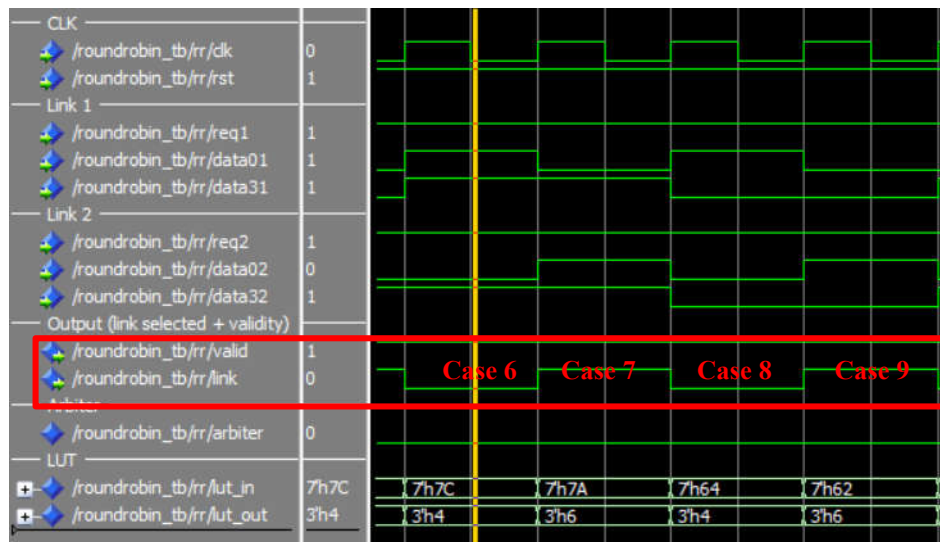
-- 9TH CASE : req1 = '1' and req2 = '1'
--             data31 <= '0' and data32 <= '0'
--             data01 <= '0' and data02 <= '1'
when 8 =>
    req1_tb <= '1';
    req2_tb <= '1';
    data31_tb <= '0';
    data32_tb <= '0';
    data01_tb <= '0';
    data02_tb <= '1';
    -- RESULT EXPECTED : link_tb = '1' and valid <= '1'
```

In the graph that follows the testbench, we can see that in the 6<sup>th</sup> case the link 1 is selected (link = '0'). We have, indeed, a minor priority bit set high for link 1 and set low for link 2.

We can also observe that in the 7<sup>th</sup> case, the link 2 is selected (link = '1'). Indeed, it is now the turn to link 2 to have a high minor priority bit when the bit number 0 of link 1 is set low.

The two following cases are the same that the two previous one but with the major priority bits of link 1 and link 2 equal to '0' instead of '1'. The results are the same because when the first level of priority is the same, only the second level has an impact on the selection.





c. Data conflict: arbiter

When both links have the same level of priority, an arbiter decide which link will be selected according to a round-robin algorithm. When the number of the conflict is even, the link selected is the second one. Otherwise, we choose the first one. That means that for the first conflict, we expect to observe link = '0', for the second conflict link = '1' and so one.

```
-- ROUND ROBIN ALGORITHM : CONFLICT
-- 10TH CASE : req1 = '1' and req2 = '1'
--             data31 <= '1' and data32 <= '1'
--             data01 <= '0' and data02 <= '0'
when 9 =>
    req1_tb <= '1';
    req2_tb <= '1';
    data31_tb <= '1';
    data32_tb <= '1';
    data01_tb <= '0';
    data02_tb <= '0';
    -- RESULT EXPECTED : link_tb = '0' and valid <= '1'
-- 11TH CASE : req1 = '1' and req2 = '1'
--             data31 <= '1' and data32 <= '1'
--             data01 <= '1' and data02 <= '1'
when 10 =>
    req1_tb <= '1';
    req2_tb <= '1';
    data31_tb <= '1';
    data32_tb <= '1';
    data01_tb <= '1';
    data02_tb <= '1';
    -- RESULT EXPECTED : link_tb = '1' and valid <= '1'
-- 12TH CASE : req1 = '1' and req2 = '1'
--             data31 <= '0' and data32 <= '0'
--             data01 <= '0' and data02 <= '0'
when 11 =>
    req1_tb <= '1';
    req2_tb <= '1';
    data31_tb <= '0';
    data32_tb <= '0';
    data01_tb <= '0';
    data02_tb <= '0';
    -- RESULT EXPECTED : link_tb = '0' and valid <= '1'
-- 13TH CASE : req1 = '1' and req2 = '1'
--             data31 <= '0' and data32 <= '0'
--             data01 <= '1' and data02 <= '1'
when 12 =>
    req1_tb <= '1';
    req2_tb <= '1';
    data31_tb <= '0';
    data32_tb <= '0';
    data01_tb <= '1';
    data02_tb <= '1';
    -- RESULT EXPECTED : link_tb = '1' and valid <= '1'
when others => null;
```

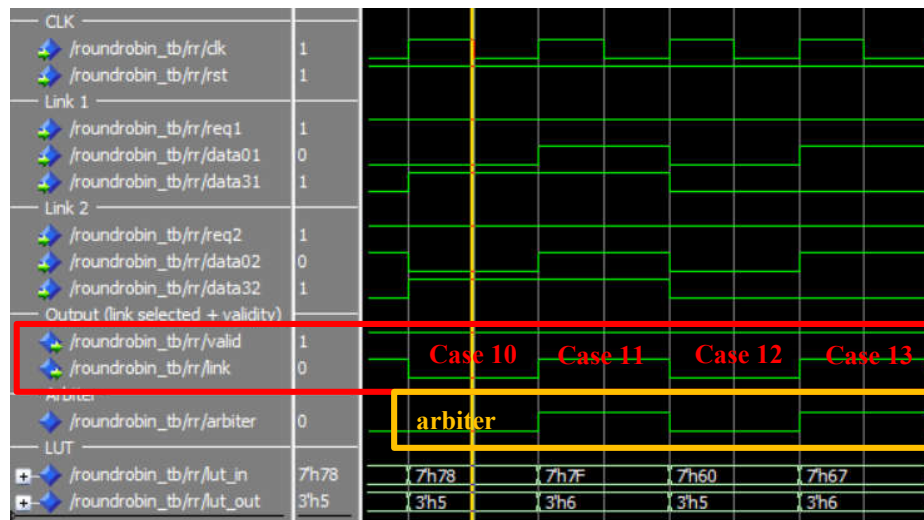
First conflict. The arbiter must be equal to '0'. Link will take the value of the arbiter and must then be equal to '0' too.

Second conflict. The arbiter and link are both set high.

In the third conflict, we invert again the value of the arbiter and therefore we select the link 1.

In the fourth conflict, the link 2 is selected since the arbiter is equal to '1'.

We can observe the result of the conflicts in the following graph:



The arbiter in yellow changes its state at each conflict.

## 5.6 MINI-ROUTER

The exact same test plan is used for the Mini-Router.

- A. A link is selected because there is only one request

The first two case are the one when only one link realizes a request. Obviously, the link that has done the request is the one that is driven out. In the first case, we expect to see `data_out <= data1` and in the second case, `data-out` must be equal to `data2`.

```

--- REQ IS HIGH FOR ONLY ONE LINK-
-- 1ST CASE : req1 = '1' and req2 = '0'
when 0 =>
    req1_tb <= '1';
    req2_tb <= '0';
    data1_tb <= "0000000010";
    data2_tb <= "0100000000";
    -- RESULT EXPECTED : link_tb = '0' and valid <= '1'
    -- and data_out <= "00000001"
-- 2ND CASE : req1 = '0' and req2 = '1'
when 1 =>
    req1_tb <= '0';
    req2_tb <= '1';
    data1_tb <= "0000000010";
    data2_tb <= "0100000000";
    -- RESULT EXPECTED : link_tb = '1' and valid <= '1'
    -- and data_out <= "01000000"

```

The results are present in the same graph as the third case.

- B. No link is selected because there is no request

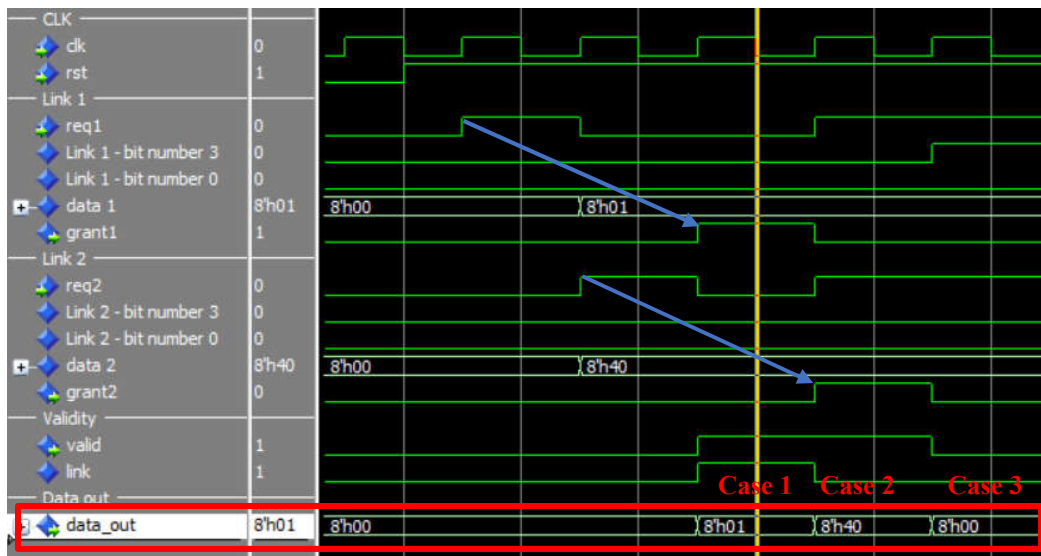
This case is the case where both `req` are low. Then, since no link does a request, no link is selected.

```

--- REQ IS LOW FOR BOTH SIGNALS
-- 3RD CASE : req1 = '0' and req2 = '0'
when 2 =>
    req1_tb <= '0';
    req2_tb <= '0';
    data1_tb <= "0000000010";
    data2_tb <= "0100000000";
    -- RESULT EXPECTED : link_tb = '0' and valid <= '0'
    -- and data_out <= "00000000"

```

The results of the simulation of case 1, 2 and 3 are in the following graph:



The blue arrows show that there is a shift due to the use of registers. We can also observe the influence of req on grant. The result are the expected ones.

- C. Both requests are emitted. We need to check the priority bit and decide between links which one is forwarded.

a. First level of priority: bit number 3

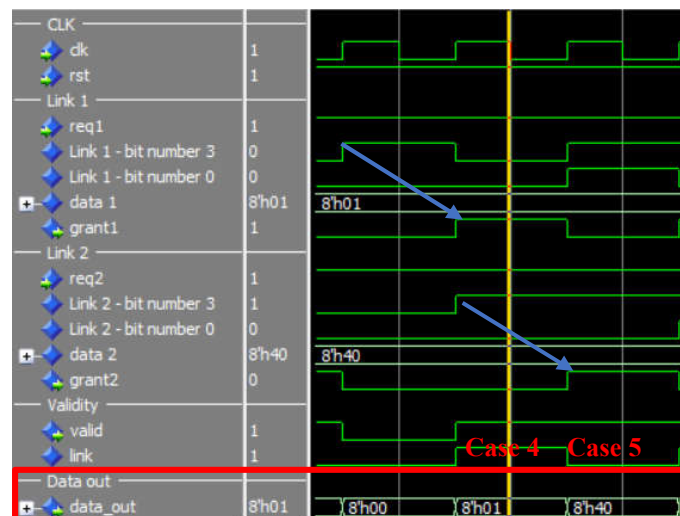
In these cases, we use the major priority bit (bit number 3) to decide between both signals since they both emit a request. We first give to link 1 the highest priority and secondly give it to link 2.

```

--- REQ IS HIGH FOR BOTH SIGNALS
-- FIRST LEVEL OF PRIORITY
-- 4TH CASE : req1 = '1' and req2 = '1'
--             data31 <= '1' and data32 <= '0'
when 3 =>
    req1_tb <= '1';
    req2_tb <= '1';
    data1_tb <= "0000001010";
    data2_tb <= "0100000000";
    -- RESULT EXPECTED : link_tb = '0' and valid <= '1'
    -- and data_out <= "00000001"
-- 5TH CASE : req1 = '1' and req2 = '1'
--             data31 <= '0' and data32 <= '1'
when 4 =>
    req1_tb <= '1';
    req2_tb <= '1';
    data1_tb <= "0000000010";
    data2_tb <= "0100001000";
    -- RESULT EXPECTED : link_tb = '1' and valid <= '1'
    -- and data_out <= "01000000"

```

We see the results in the graph as follows:



The blue arrows show the impact of the bit number on grant.

## b. Second level of priority: bit number

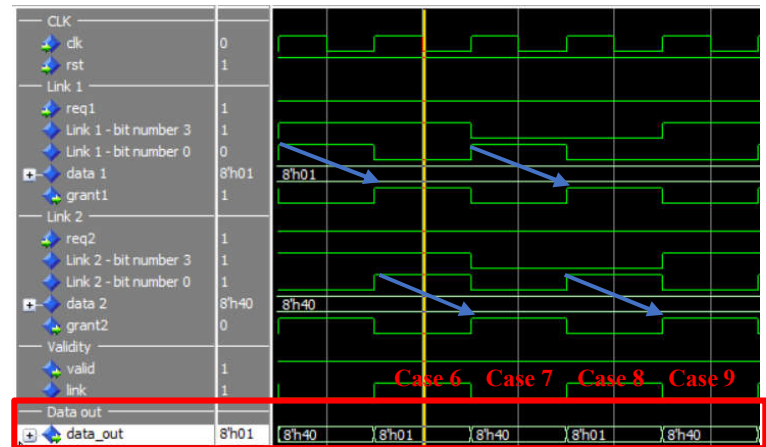
```

-- SECOND LEVEL OF PRIORITY
-- 6TH CASE : req1 = '1' and req2 = '1'
--             data31 <= '1' and data32 <= '1'
--             data01 <= '1' and data02 <= '0'
when 5 =>
    req1_tb <= '1';
    req2_tb <= '1';
    data1_tb <= "0000001011";
    data2_tb <= "0100001000";
    -- RESULT EXPECTED : link_tb = '0' and valid <= '1'
    -- and data_out <= "00000001"
-- 7TH CASE : req1 = '1' and req2 = '1'
--             data31 <= '1' and data32 <= '1'
--             data01 <= '0' and data02 <= '1'
when 6 =>
    req1_tb <= '1';
    req2_tb <= '1';
    data1_tb <= "0000001010";
    data2_tb <= "0100001001";
    -- RESULT EXPECTED : link_tb = '1' and valid <= '1'
    -- and data_out <= "01000000"
-- 8TH CASE : req1 = '1' and req2 = '1'
--             data31 <= '0' and data32 <= '0'
--             data01 <= '1' and data02 <= '0'
when 7 =>
    req1_tb <= '1';
    req2_tb <= '1';
    data1_tb <= "0000000011";
    data2_tb <= "0100000000";
    -- RESULT EXPECTED : link_tb = '0' and valid <= '1'
    -- and data_out <= "00000001"
-- 9TH CASE : req1 = '1' and req2 = '1'
--             data31 <= '0' and data32 <= '0'
--             data01 <= '0' and data02 <= '1'
when 8 =>
    req1_tb <= '1';
    req2_tb <= '1';
    data1_tb <= "0000000010";
    data2_tb <= "0100000001";
    -- RESULT EXPECTED : link_tb = '1' and valid <= '1'

```

The selection of the link is now doing on the second level of priority and then uses the bit number 0 to decide between the links the one to choose. In the 6<sup>th</sup> and 8<sup>th</sup> cases, we prioritize the link 1 and in the 7<sup>th</sup> and 9<sup>th</sup>, we prioritize the link 2.

The results are presented in the following graph, obtained on Modelsim:



The blue arrows represent the impact of the minor priority bit on grant.

## c. Data conflict: arbiter

When the links bear the same priority level, we need to use the arbiter of the round-robin algorithm to decide between them. We expect to see link 1 selected when the first conflict occurs, link 2 when the second occurs, link 1 again for the third and so on.

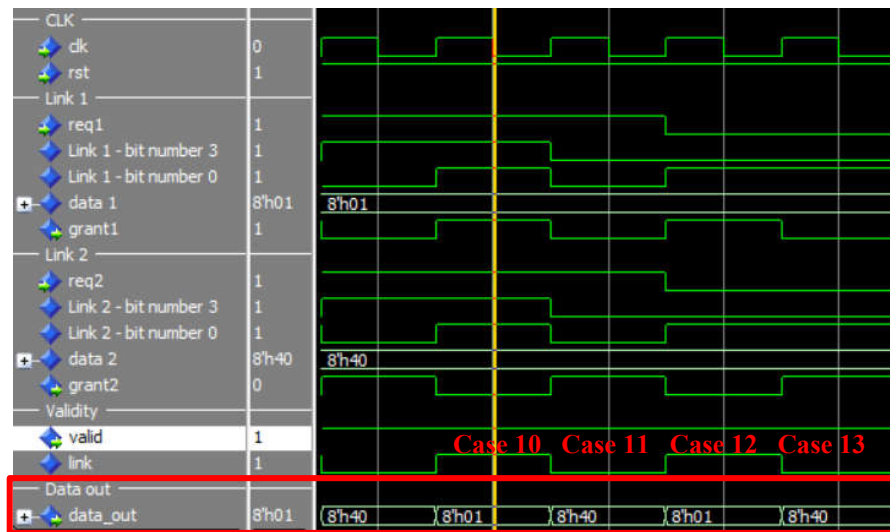
```

-- ROUND ROBIN ALGORITHM : CONFLICT
-- 10TH CASE : req1 = '1' and req2 = '1'
--             data31 <= '1' and data32 <= '1'
--             data01 <= '0' and data02 <= '0'
when 9 =>
    req1_tb <= '1';
    req2_tb <= '1';
    data1_tb <= "0000001010";
    data2_tb <= "0100001000";
    -- RESULT EXPECTED : link_tb = '0' and valid <= '1'
    -- and data_out <= "00000001"
-- 11TH CASE : req1 = '1' and req2 = '1'
--             data31 <= '1' and data32 <= '1'
--             data01 <= '1' and data02 <= '1'
when 10 =>
    req1_tb <= '1';
    req2_tb <= '1';
    data1_tb <= "0000001011";
    data2_tb <= "0100001001";
    -- RESULT EXPECTED : link_tb = '1' and valid <= '1'
    -- and data_out <= "01000000"
-- 12TH CASE : req1 = '1' and req2 = '1'
--             data31 <= '0' and data32 <= '0'
--             data01 <= '0' and data02 <= '0'
when 11 =>
    req1_tb <= '1';
    req2_tb <= '1';
    data1_tb <= "0000000010";
    data2_tb <= "0100000001";
    -- RESULT EXPECTED : link_tb = '0' and valid <= '1'
    -- and data_out <= "00000001"
-- 13TH CASE : req1 = '1' and req2 = '1'
--             data31 <= '0' and data32 <= '0'
--             data01 <= '1' and data02 <= '1'
when 12 =>
    req1_tb <= '1';
    req2_tb <= '1';
    data1_tb <= "0000000011";
    data2_tb <= "0100000000";
    -- RESULT EXPECTED : link_tb = '1' and valid <= '1'
    -- and data_out <= "01000000"

```



The results are in the following graph that comes from Modelsim:



## 5.7 ADAPTER

The adapter was only instantiated in order to accommodate the mini-router to the FPGA Zynq-7000 so we don't do a testbench with all the different cases but only with one.

```
-- We send sequentially the data : 011111111 to retrieve 01111111
when 0 =>
    req1_tb <= '0';
    req2_tb <= '0';
    data1_serial_tb <= '0';
    data2_serial_tb <= '0';

when 1 =>
    req1_tb <= '0';
    req2_tb <= '0';
    data1_serial_tb <= '1';
    data2_serial_tb <= '0';

when 2 =>
    req1_tb <= '0';
    req2_tb <= '0';
    data1_serial_tb <= '1';
    data2_serial_tb <= '1';

when 3 =>
    req1_tb <= '0';
    req2_tb <= '0';
    data1_serial_tb <= '1';
    data2_serial_tb <= '0';

when 4 =>
    req1_tb <= '0';
    req2_tb <= '0';
    data1_serial_tb <= '1';
    data2_serial_tb <= '1';

when 5 =>
    req1_tb <= '0';
    req2_tb <= '0';
    data1_serial_tb <= '1';
    data2_serial_tb <= '1';

when 6 =>
    req1_tb <= '0';
    req2_tb <= '0';
    data1_serial_tb <= '1';
    data2_serial_tb <= '0';

when 7 =>
    req1_tb <= '0';
    req2_tb <= '0';
    data1_serial_tb <= '1';
    data2_serial_tb <= '0';

when 8 =>
    req1_tb <= '0';
    req2_tb <= '0';
    data1_serial_tb <= '1';
    data2_serial_tb <= '0';

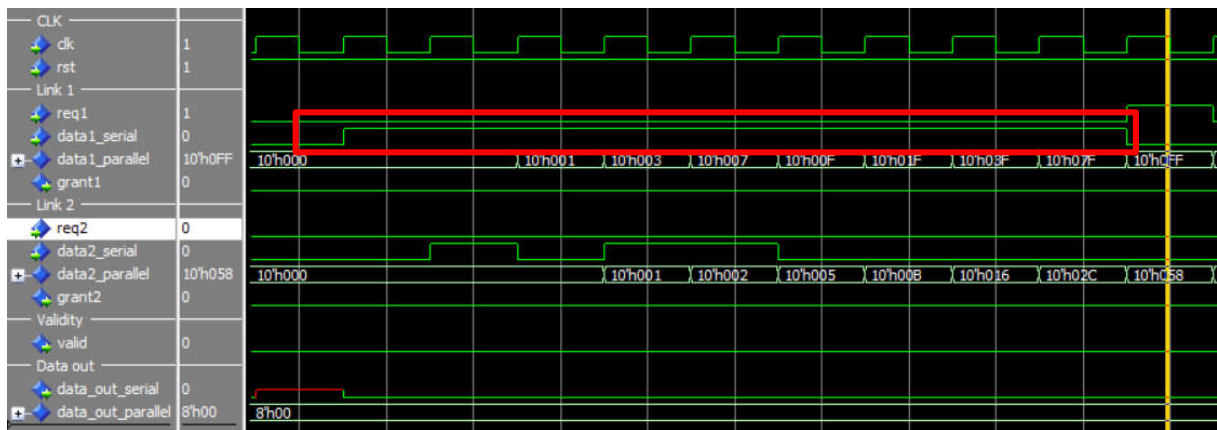
when 9 =>
    req1_tb <= '0';
    req2_tb <= '0';
    data1_serial_tb <= '1';
    data2_serial_tb <= '0';

when 10 =>
    req1_tb <= '1'; -- the signal is complete: send the request
    req2_tb <= '0';
    data1_serial_tb <= '0';
    data2_serial_tb <= '0';

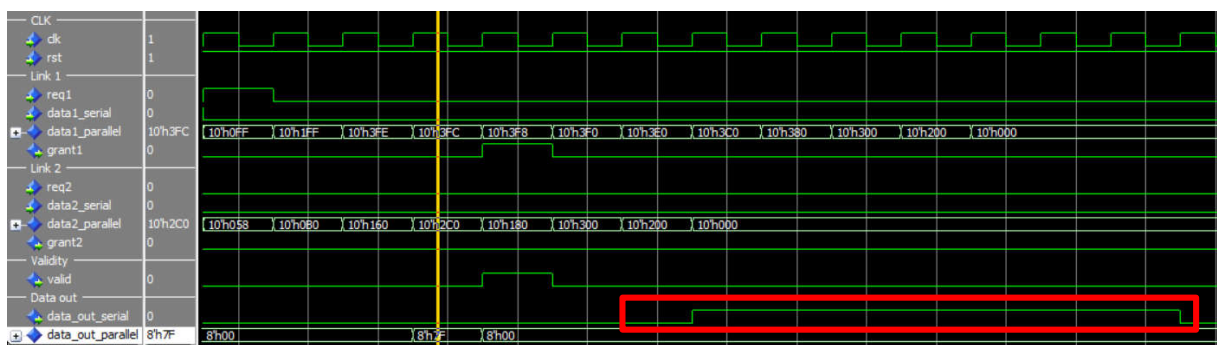
-- RESULT EXPECTED : data_out_parallel <= "01111111"
-- and the bits will be sent each rising edge in this order : 0,1,1,1,1,1,1,1
```

Each clock cycle we send a new bits from the MSB to the LSB on data1. Since req1 is high and req2 is low, we expect to see data\_out with the data of data1.

The result of this last testbench are presented in this graph:



In red, we can see the serial input. When it is parallelized, it is equal to “01111111”. We expect then “01111111” for the output.



The parallelized output is “01111111” and we retrieve then the right value at the end. The link is forward correctly. The clock must be multiplied by 10 in order to compensate the delays due to the serial input and output.

## 6 SYNTHESIS LOGIC RESULTS

### 6.1 CONSTRAINTS

I select the following pin for the inputs and outputs:

```

3 | set_property PACKAGE_PIN L16 [get_ports clk]
4 | set_property PACKAGE_PIN T16 [get_ports data1]
5 | set_property PACKAGE_PIN W13 [get_ports data2]
6 | set_property PACKAGE_PIN D18 [get_ports data_out]
7 | set_property PACKAGE_PIN G14 [get_ports grant1]
8 | set_property PACKAGE_PIN M15 [get_ports grant2]
9 | set_property PACKAGE_PIN M14 [get_ports valid]
10 | set_property PACKAGE_PIN R18 [get_ports rst]
11 | set_property PACKAGE_PIN P15 [get_ports req1]
12 | set_property PACKAGE_PIN G15 [get_ports req2]

```

We can see these pins (except for the clock, which is not visible on this picture) on the figure 14, taken from the Zynq 7000 documentation.

I decide to use a clock frequency of 125 MHz and I create then it in the constraints file:

```
create_clock -period 8.000 -name clk125 -waveform {0.000 4.000} -add [get_ports clk]
```

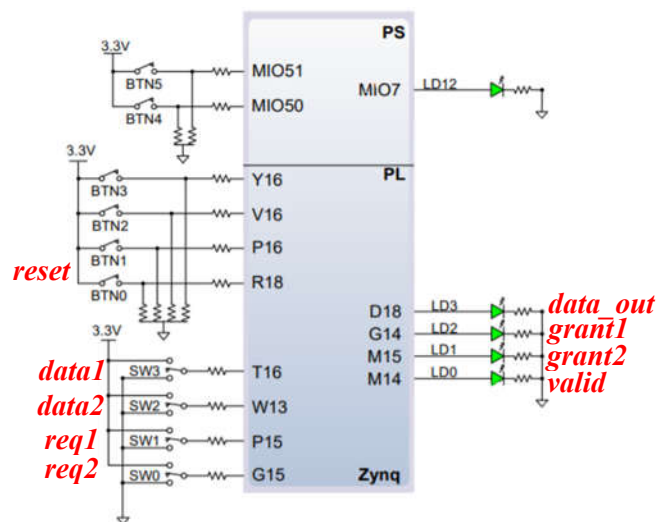


Figure 14. ZYBO GPIO.

## 6.2 SYNTHESIS WARNINGS

There are two synthesis warning that occurred after the synthesis.


1. [Synth 8-3332] Sequential element (adapter\_serial\_parallel/piso\_data\_out/intermediate\_reg[0]\_C) is unused and will be removed from module MINI\_ROUTER\_WRAPPER.

This warning is present because Vivado optimizes the PISO. I didn't get exactly why (and so I don't realize myself the optimization) but after looking for an answer on Internet, I saw that it could be not taken into consideration because sometimes it happens only when two registers are merged and share the same data.

2. [Constraints 18-5210] No constraints selected for write.

Resolution: This message can indicate that there are no constraints for the design, or it can indicate that the used\_in flags are set such that the constraints are ignored. This later case is used when running synth\_design to not write synthesis constraints to the resulting checkpoint. Instead, project constraints are read when the synthesized design is opened.

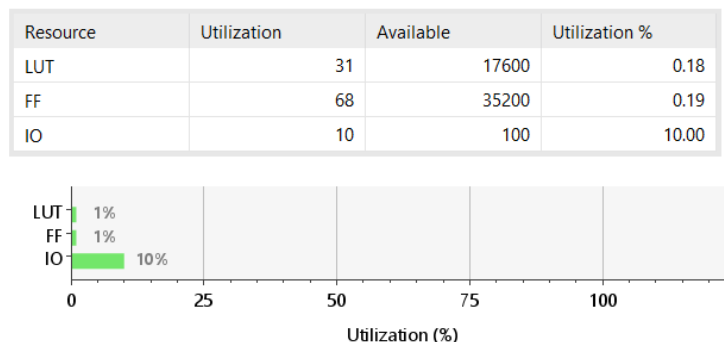
I firstly thought that I forget to set my constraints file as a target. I checked it but the file is showed as following:

 MINI\_ROUTER\_CONSTRAINTS.xdc (target)

I already have this message during the project in class and since it works correctly at this moment and because my constraints are targeted, I assume that I could ignore this warning.

## 6.3 DEVICE UTILIZATION

Here we can see the utilization of FPGA resources. This summary is provided by Vivado and allows us to visualize how the design uses the resources of the board:



*Utilization of the resource of the FPGA device*

Inputs and outputs represent 10% of the resources' utilization. The design actually has 10 input (5) and output (4) ports. If 10 ports cause 10% of the resources' utilization, then using parallel inputs and outputs rather than serial as originally intended would have caused high device utilization. The choice to serialize the ports therefore minimized device utilization.

We also have 68 flip flop registers and we use 31 LUT. We use then less than 0.2% of the flip flop and LUT presented on the board.

More in details, we can see which block use the slice, LUTs, registers, etc. in the following table, provide by Vivado:

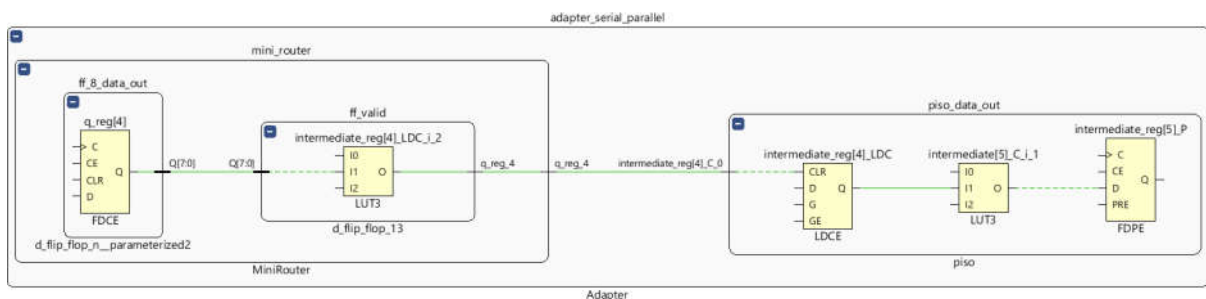
Name	^1	Slice LUTs (17600)	Slice Registers (35200)	Slice (4400)	LUT as Logic (17600)	Bonded IOB (100)	BUFGCTRL (32)
MINI_ROUTER_WRAPPER		31	68	32	31	10	1
adapter_serial_parallel (Adapter)		31	68	32	31	0	0
ff_data1_serial (d_flip_flop)		0	1	1	0	0	0
ff_data2_serial (d_flip_flop_0)		0	1	1	0	0	0
ff_data_out_serial (d_flip_flop_1)		0	1	1	0	0	0
ff_grant1 (d_flip_flop_2)		0	1	1	0	0	0
ff_grant2 (d_flip_flop_3)		0	1	1	0	0	0
ff_req1 (d_flip_flop_4)		0	1	1	0	0	0
ff_req2 (d_flip_flop_5)		0	1	1	0	0	0
ff_valid (d_flip_flop_6)		0	1	1	0	0	0
mini_router (MiniRouter)		23	30	12	23	0	0
ff_8_data_out (d_flip_flop_n_)		0	8	4	0	0	0
ff_10_data1 (d_flip_flop_n)		0	8	6	0	0	0
ff_10_data2 (d_flip_flop_n_8)		1	8	6	1	0	0
ff_grant1 (d_flip_flop_9)		0	1	1	0	0	0
ff_grant2 (d_flip_flop_10)		0	1	1	0	0	0
ff_req1 (d_flip_flop_11)		0	1	1	0	0	0
ff_req2 (d_flip_flop_12)		0	1	1	0	0	0
ff_valid (d_flip_flop_13)		8	1	7	8	0	0
mux (Multiplexer)		8	0	4	8	0	0
rr_arbiter (RoundRobin)		6	1	2	6	0	0
ff_arbiter (d_flip_flop_14)		1	1	2	1	0	0
rr_lut (LUT)		5	0	2	5	0	0
piso_data_out (piso)		8	24	24	8	0	0
sipo_data1 (sipo)		0	3	3	0	0	0
sipo_data2 (sipo_7)		0	3	3	0	0	0

The PISO register seems to use a lot of slices (75%), LUTs (25%) and registers (35%) comparing to the other block and according to what it is supposedly doing. Then, it is obvious that it will be an important actor of the critical path. If something must be optimized, it would be it.

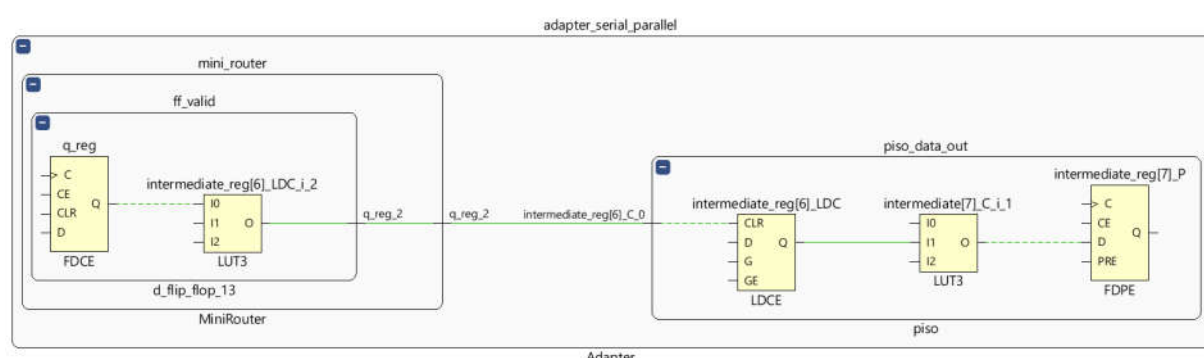
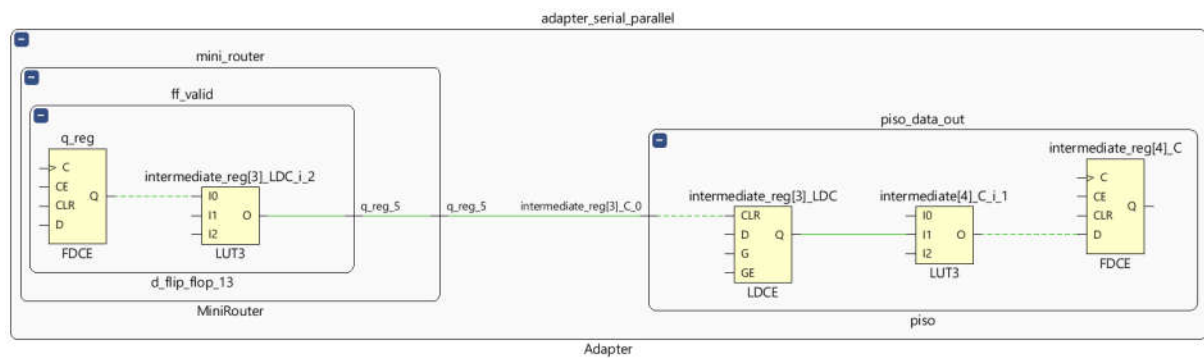
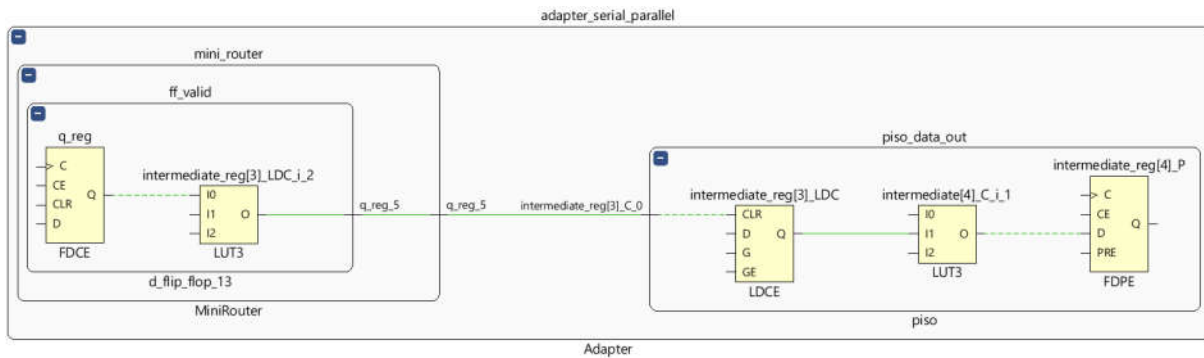
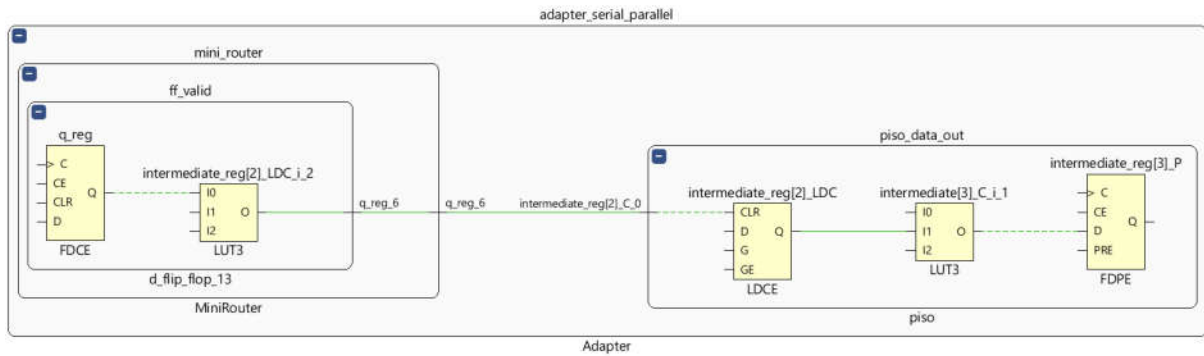
Most of the registers are found in the Mini-Router entity (44% of them) since we register the parallelized input and output here. The multiplexer and the arbiter are, with the flip flop of the signal valid, the principal user of LUT of the design.

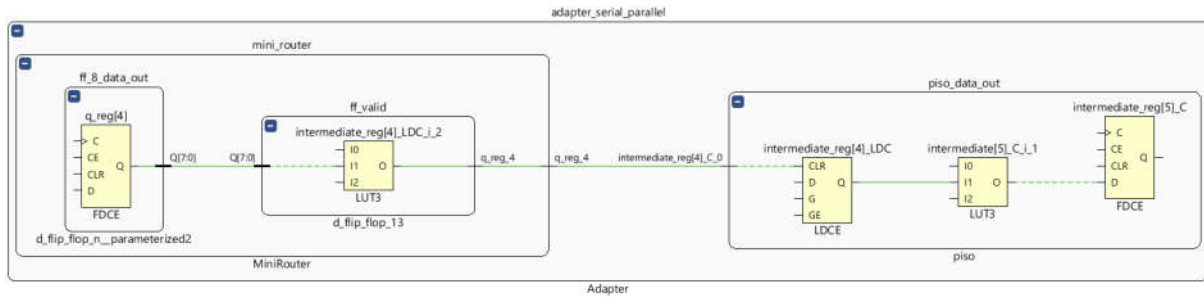
## 6.4 MAXIMUM FREQUENCY AND RELEVANT CRITICAL PATHS

The Report Timing Summary reports 10 relevant critical paths. There are as follows, from the worst critical path to the less bad:

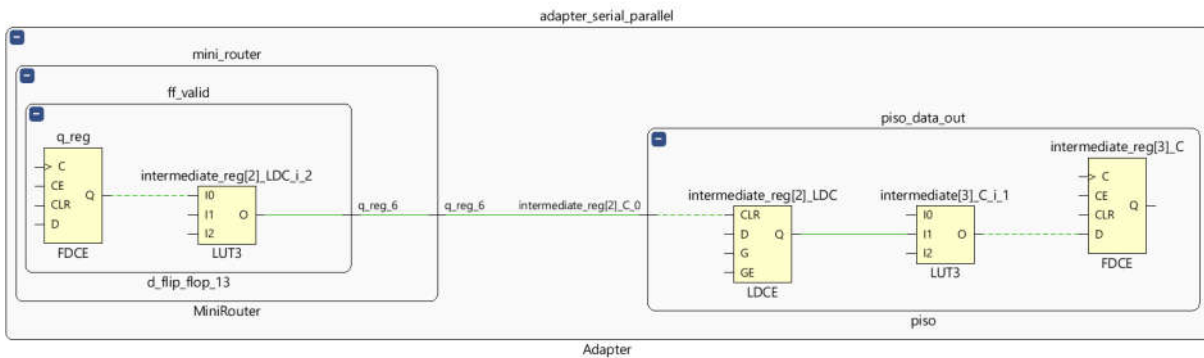


Path 10

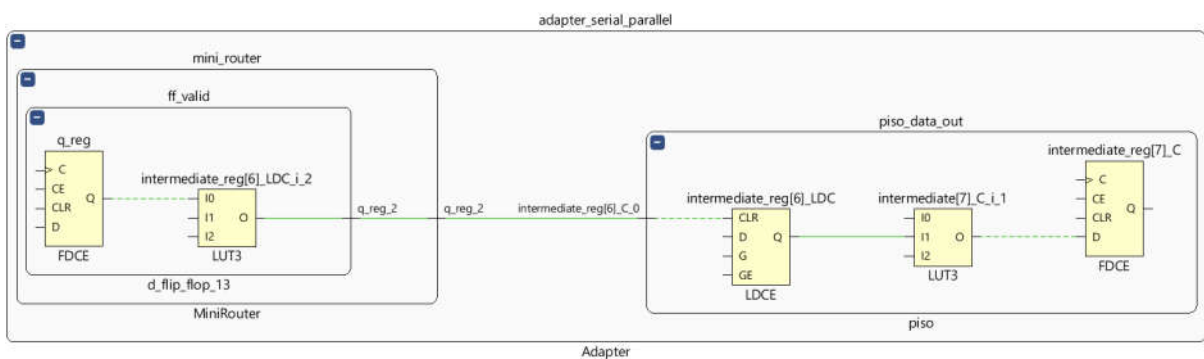




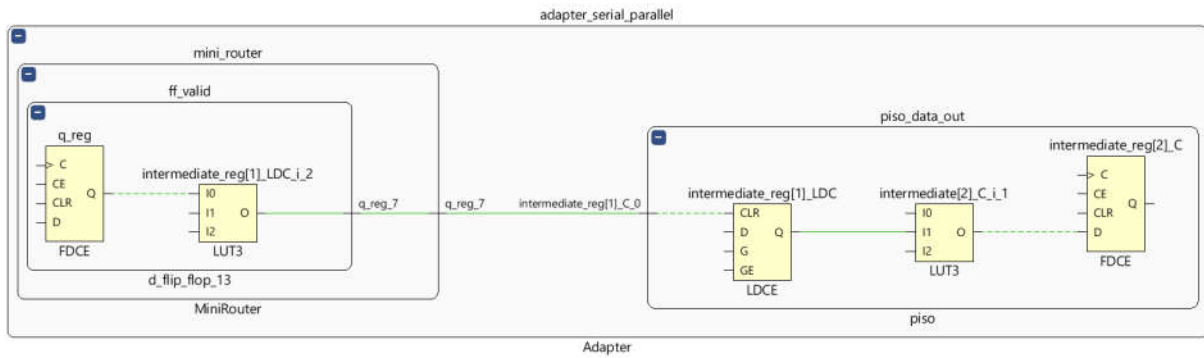
Path 5



Path 4

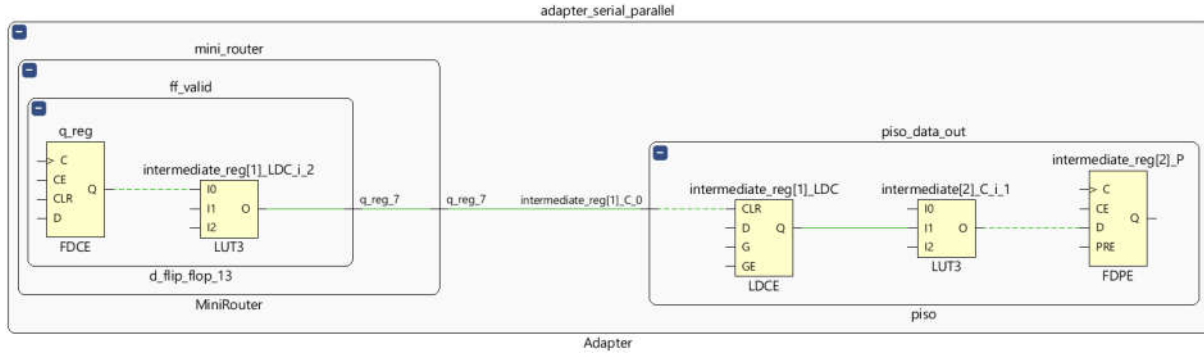


Path 3



Path 2





Path 1

In all this critical path, we retrieve the same main actors: the PISO and the flip flop valid in the MiniRouter entity. We also find the flip flop for the data\_out in the MiniRouter entity sometimes.

The maximum clock frequency has the following formula:

$$f_{clk_{MAX}} = \frac{1}{T_{CLK} - T_{SLACK}}$$

We compute the maximum clock frequency for each path, using the Slack given in the Report Timing Summary in the table here:

Critical Path	$\overline{T_{CLK}}$	$\overline{T_{SLACK}}$	$f_{clk_{MAX}}$
Path 10	8 ns	4.014 ns	250.9 MHz
Path 9	8 ns	3.893 ns	243.5 MHz
Path 8	8 ns	3.823 ns	239.4 MHz
Path 7	8 ns	3.817 ns	239.1 MHz
Path 6	8 ns	3.773 ns	236.6 MHz
Path 5	8 ns	3.753 ns	235.5 MHz
Path 4	8 ns	3.698 ns	232.5 MHz
Path 3	8 ns	3.628 ns	228.7 MHz
Path 2	8 ns	3.584 ns	226.4 MHz
Path 1	8 ns	3.236 ns	209.9 MHz

## 7 CONCLUSION

The main goal of this project was to design and implement a synchronous mini-router working with an arbiter using a Round-Robin algorithm and using VHDL and a Zynq FPGA. The test results of the Mini-Router itself seem conclusive since all the identified cases of link routing have been presented and all work correctly.

It was interesting to look for a solution to optimize the use of the ports on the FPGA board. This search seems to have been rewarded by a rather low use of the board resources.

However, the PISO element could be improved since it causes a warning first (Vivado having decided to optimize one of its elements), since it is one of the most resource-intensive components of the whole design and since it is one of the important elements of the identified critical paths.