Module library

Eggen, Jonas Agentoft
jonasae@stud.ntnu.no

Gjermundnes, Øystein
oystein.gjermundnes@iet.ntnu.no

September 5, 2017

Abstract

This document contains the documentation of a digital module library. The digital module library contains basic digital building blocks. These building blocks will be used as a basis for lectures and assignments in TFE4141 Design of digital systems 1.

Contents

1 Introduction

This document contains a digital module library. The modules described in the document will be used as a part of assignments and lectures in TFE4141 Design of Digital Systems 1.

Each module is described in a separate chapter. The documentation includes a functional description of the design, description of the interface of the design, microarchitecture description, PPA (power, performance, area) data and netlists.

More complex designs also has a section describing the verification methodology.

1.1 Functional description

Each module in the library must have a functional description. The focus is to describe the functionality of the block without going into any implementation details.

Implementation details, such as diagrams illustrating the microarchitecture will instead be treated in separate sections.

1.2 Interface

Description of the interface. This should include listing of the entity declaration of the module. It should also contain a table for each main interface of the design with a description of each signal in the interface. The documentation of each interface should be further enhanced through waveform diagrams using the wavedrom tool `http://wavedrom.com/`

1.3 Microarchitecture

This section describes the microarchitecture of the module. Some information should also be given that explains why the microarchitecture is a good implementation of the functional specification.

1.4 Microarchitecture characteristics

Section where interesting microarchitecture characteristics are explained. This includes information related to:

- Whether or not the design is a combinational or a sequential circuit.
- The number of flip-flops in the design.
- The critical path of the design.

1.5 PPA

PPA is a three letter acronym that stands for Power, Performance and Area. These are physical characteristics of the implementation of the design. When engineers discuss PPA of a particular design, they will typically relate the characteristics to a technology process. Examples of such processes could be:

- TSMC28HPM (ASIC): `http://www.tsmc.com/english/dedicatedFoundry/technology/28nm.htm`
- TSMC16FF (ASIC): `http://www.tsmc.com/english/dedicatedFoundry/technology/16nm.htm`
- Xilinx Zync XC7Z020 (FPGA): `http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf`

Traditionally PPA refers to power, performance and area, but it is not limited to that, and engineers are now typically associating PPA with other physical aspects as well such as energy efficiency or area utilization.

Some terms commonly used to describe PPA:

- *Max clock frequency.* The longest combinatorial bath between two flip-flops in the design usually determines the max clock frequency the design can run at and still produce correct output. If the clock frequency is increased above this treshold, then the design is likely to produce the wrong output. The max clock frequency must be accompanied with information about operating conditions as well as target technology.
- *Area.* An important physical aspect of the design is the area of the design. The manufacturing cost of the design is often proportional to the area the design occupies, and for cost sensitive products it is often important to have a low area. The area is often given in units such as mm^2. That number alone can often lead to misunderstandings unless it is accompanied with information about the process. Relevant information includes:
  - Are scan chains inserted or not?
  - Is the area an estimate after synthesis or is the area reported after place and route?
  - Is the area associated with routing taken into account?
  - Has the silicon utilization been factored in?
- *Power.* Power is usually given relative to a process, clock frequency and supply voltage. It will also depend on which workload or benchmark that is run on the device.
- *Performance.* The performance of the device is often given in terms such as frames per second, messages per second, instructions per section, FLOPS (floating point operations per second), bytes per second. The performance data must be accompanied by a clock frequency.
- *Energy efficiency.* More important than power these days is often that the energy consumed in order to run a particular workload is as low as possible. In order words, the energy efficiency must be high. This is often measured in Joules per frame or Joules per benchmark. The benchmark/test/workload must of course be specified as well.
- *Performance density.* For many applications, its very important that the design strikes a good balance between performance and area. One way to measure this is to ensure that the design has as high a performance to area ratio as possible.

1.6 RTL schematic and synthesized netlist

This section contains RTL schematic and netlists generated by the synthesis tool. The RTL schematic can help the designer understand how the VHDL code translates into actual logic. By spending some time studing the RTL schematic and the netlist, it is easier to understand how to write the RTL code in order to e.g. increase the clock frequency or reduce the area.

For some designs this chapter includes observations conclusions drawn by the designer when comparing the RTL code with the produced netlist before and after synthesis.

2 Technology specific optimizations

The max frequency of logic increases more than the max frequency of RAM when moving from a 28nm process to a 16nm fin-fet process. The optimal logic depth between two registers might therefore be different in two different CMOS processes.

When comparing standard-cell ASICs with FPGAs the technology differences are even more pronounced. The primitives in standard-cell technologies are gates and flip-flops, and FPGAs contains primitives such as LUTs, register slices, block rams and multipliers.

IP companies will usually have to write technology-independent RTL code that maps well to a range of different technologies. If you, on the other hand, can optimize for only one target technology, then it is often possible to achieve a better result than what would be possible when forced to write generic code.

The next sections lists up some optimization guidelines that are valid for Xilinx FPGAs.

Note though, that the modules in this document are written in a technology-independent style. The term project should also be written in technology-independent VHDL.

2.1 Xilinx FPGA

In `http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/xst_v6s6.pdf`, Xilinx gives an overview of best practice for their FPGAs. Some interesting points are repeated below:

Flip-Flops and Registers Initialization

To initialize the content of a Register at circuit power-up, specify a default value for the signal modeling it.

```
signal example1 : std_logic := '1';
signal example2 : std_logic_vector(3 downto 0) := (others => '0');
signal example3 : std_logic_vector(3 downto 0) := "1101";
```

Reset

Avoid operational set/reset logic whenever possible. There may be other, less expensive, ways to achieve the desired effect, such as taking advantage of the circuit global reset by defining initial contents.

Active high

Always describe the clock enable, set, and reset control inputs of Flip-Flop primitives as active-High. If they are described as active-Low, the resulting inverter logic will penalize circuit performance.

2.2 ASIC

Flip-Flops and Registers Initialization

Flip-flops without reset are smaller than flip-flops with reset. It is therefore possible to save some area by avoiding to use flip-flops with reset.

Always use reset on flip-flops driving control signals and control logic. Flip-flops for data-registers, on the other hand, is often not necessary.

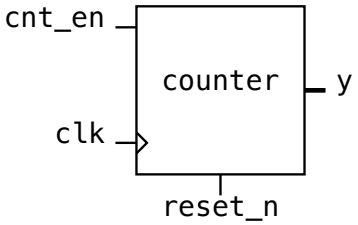
The standard cells for flip-flops using active low reset are smaller than the standard cells for flip-flops using active high reset. It is therefore common to use active low reset for CMOS standard-cell technologies.

3 Counter

A counter increments its output every clock cycle given that a control signal, `cnt_en`, is asserted.

3.1 Interface

```
entity counter is
  generic (
    COUNTER_WIDTH : natural := 8);
  port (
    clk      : in  std_logic;
    reset_n  : in  std_logic;
    cnt_en   : in  std_logic;
    y        : out std_logic_vector(COUNTER_WIDTH-1 downto 0));
end counter;
```



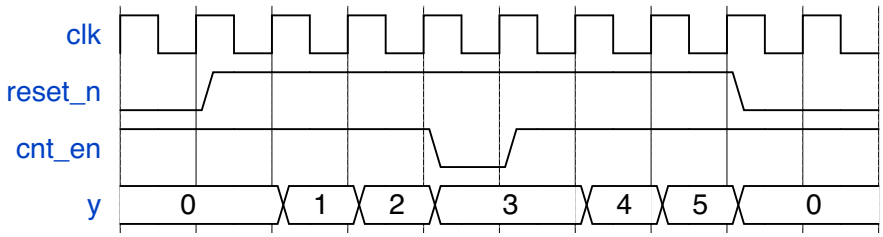


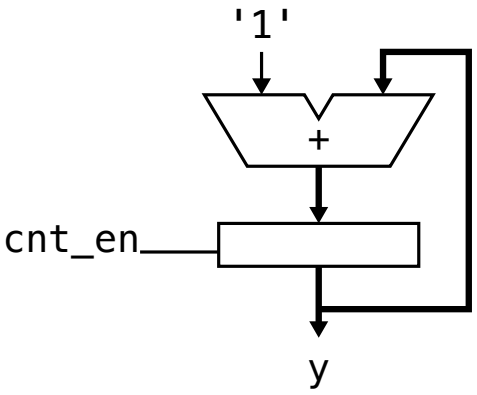
Figure 1: Example waveform

3.2 Microarchitecture

This circuit can be implemented in many ways. Here, two possible implementations are discussed. One could either implement what is shown in the figure, or swap '1' and `cnt_en`. Reasons for choosing the former include:

Energy consumption
: By connecting `cnt_en` to the enable-input of the register, toggling is reduced.

Timing
: Imagine that `cnt_en` is the result of an intricate calculation. This would cause `cnt_en` to get its value quite some time after the clock edge. If `cnt_en` was connected to the adder, the resulting critical path would be even longer. By connecting `cnt_en` directly to the enable-input of the register this is avoided.



3.3 Microarchitecture characteristics

Combinatorial	Sequential	Number of flip flops
✓	✓	COUNTER_WIDTH

3.4 Source

```
1  -- ********************************************************************************
2  -- Name:       counter.vhd
3  -- Created:    03.02.16 @ NTNU
4  -- Author:     Jonas Eggen
5  -- Purpose:    A simple counter.
6  -- ********************************************************************************
7
8  library ieee;
9  use ieee.std_logic_1164.all;
10 use ieee.numeric_std.all;
11
12 entity counter is
13   generic (
14     COUNTER_WIDTH : natural := 8);
15   port (
16     clk     : in  std_logic;
17     reset_n : in  std_logic;
18     cnt_en  : in  std_logic;
19     y       : out std_logic_vector(COUNTER_WIDTH-1 downto 0));
20 end counter;
21
22 architecture rtl of counter is
23   signal value : unsigned(COUNTER_WIDTH-1 downto 0);
24 begin
25   process(clk, reset_n)
26   begin
27     if(reset_n = '0') then
28       value <= (others => '0');
29     elsif (clk'event and clk='1') then
30       if (cnt_en = '1') then
31         value <= value + 1;
32       end if;
33     end if;
34   end process;
35   y <= std_logic_vector(value);
36 end rtl;
```

3.5 PPA

Table 1: Resource consumption for target device Xilinx Zync XC7Z020-CLG484-1

Generic	COUNTER_WIDTH = 8	COUNTER_WIDTH = 16	Available Resources
f_{max}	484 MHz	435 MHz	
FF	8	16	106400
LUT	2	2	53200
I/O	11	19	200
BUFG	1	1	32

3.6 Netlist

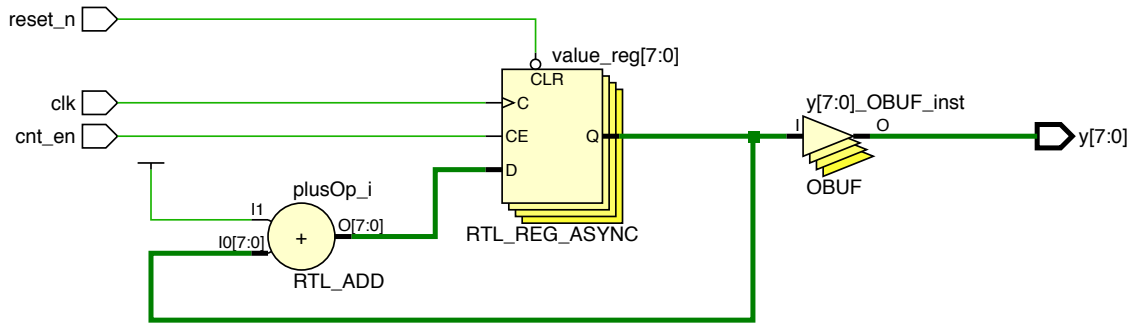


Figure 2: RTL schematic for COUNTER_WIDTH = 8

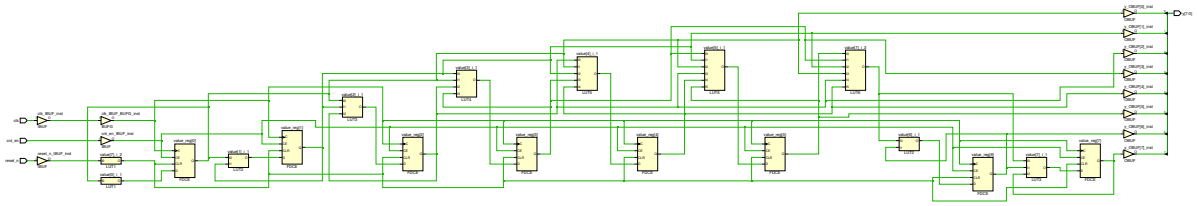


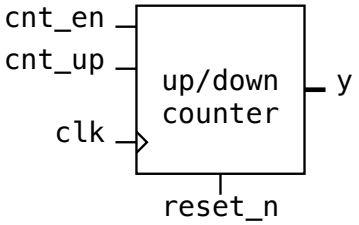
Figure 3: Synthesised schematic for COUNTER_WIDTH = 8

4 Up/down counter

A counter that counts every clock cycle given that a control signal, `cnt_en`, is asserted. The direction is controlled by the control signal `cnt_up`.

4.1 Interface

```
entity up_down_counter is
  generic (
    COUNTER_WIDTH : natural := 8);
  port (
    clk      : in  std_logic;
    reset_n  : in  std_logic;
    cnt_en   : in  std_logic;
    cnt_up   : in  std_logic;
    y        : out std_logic_vector(COUNTER_WIDTH-1 downto 0));
end up_down_counter;
```



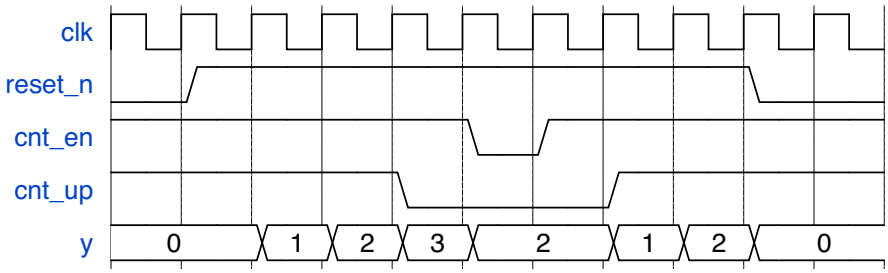
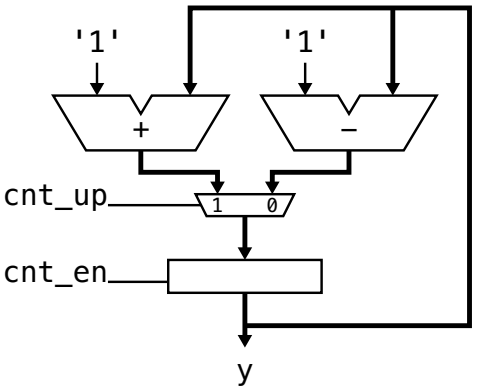


Figure 4: Example waveform

4.2 Microarchitecture

When implementing this circuit, the goals for the regular counter in Section 3 are reused. That means that timing and energy consumption are still in focus.
Depending on the application, the implementation should also change. If area is important, functionality sharing might help reduce area (i.e. using only one adder, and muxing in different constants depending on `cnt_up`).



4.3 Microarchitecture characteristics

Combinatorial	Sequential	Number of flip flops
✓	✓	COUNTER_WIDTH

4.4 Source

```
1  -- ********************************************************************************
2  -- Name:       up_down_counter.vhd
3  -- Created:    13.02.16 @ NTNU
4  -- Author:     Jonas Eggen
5  -- Purpose:    An up/down counter. Counts if cnt_en is asserted. Direction is
6  --             controlled by cnt_up.
7  -- ********************************************************************************
8
9  library ieee;
10 use ieee.std_logic_1164.all;
11 use ieee.numeric_std.all;
12
13 entity up_down_counter is
14   generic (
15     COUNTER_WIDTH : natural := 8);
16   port (
17     clk     : in  std_logic;
18     reset_n : in  std_logic;
19     cnt_en  : in  std_logic;
20     cnt_up  : in  std_logic;
21     y       : out std_logic_vector(COUNTER_WIDTH-1 downto 0));
22 end up_down_counter;
23
24 -- Naive implementation
25 architecture rtl_alt1 of up_down_counter is
26   signal value : unsigned(COUNTER_WIDTH-1 downto 0);
27 begin
28   process(clk, reset_n)
29   begin
30     if(reset_n = '0') then
31       value <= (others => '0');
32     elsif (clk'event and clk='1') then
33       if (cnt_en = '1') then
34         if (cnt_up = '1') then
35           value <= value + 1;
36         else
37           value <= value - 1;
38         end if;
39       end if;
40     end if;
41   end process;
42   y <= std_logic_vector(value);
43 end rtl_alt1;
44
45
46 -- Logic separation
47 architecture rtl_alt2 of up_down_counter is
48   signal value_r   : unsigned(COUNTER_WIDTH-1 downto 0);
49   signal value_nxt : unsigned(COUNTER_WIDTH-1 downto 0);
50 begin
51   process(clk, reset_n)
52   begin
53     if(reset_n = '0') then
54       value_r <= (others => '0');
55     elsif (clk'event and clk='1') then
56       if (cnt_en = '1') then
57         value_r <= value_nxt;
58       end if;
59     end if;
60   end process;
61
62   process(cnt_up, value_r)
63   begin
64     if (cnt_up = '1') then
65       value_nxt <= value_r + 1;
66     else
67       value_nxt <= value_r - 1;
68     end if;
69   end process;
70
71   y <= std_logic_vector(value_r);
72 end rtl_alt2;
73
74
75 -- Functionality sharing and logic separation
76 architecture rtl_alt3 of up_down_counter is
77   signal value_r   : unsigned(COUNTER_WIDTH-1 downto 0);
78   signal value_nxt : unsigned(COUNTER_WIDTH-1 downto 0);
79   signal delta     : integer range -1 to 1;
80 begin
81   process(clk, reset_n)
```

```
82    begin
83      if(reset_n = '0') then
84        value_r <= (others => '0');
85      elsif (clk'event and clk='1') then
86        if (cnt_en = '1') then
87          value_r <= value_nxt;
88        end if;
89      end if;
90    end process;
91
92    value_nxt <= value_r + delta;
93
94    process(cnt_up, value_r)
95    begin
96      if (cnt_up = '1') then
97        delta <= 1;
98      else
99        delta <= -1;
100     end if;
101   end process;
102
103   y <= std_logic_vector(value_r);
104 end rtl_alt3;
```

4.5 PPA

Table 2: Resource consumption for target device Xilinx Zync XC7Z020-CLG484-1

Generic	COUNTER_WIDTH = 8	COUNTER_WIDTH = 16	Available Resources
f_{max}	470 MHz	372 MHz	
FF	8	16	106400
LUT	10	16	53200
I/O	12	20	200
BUFG	1	1	32

4.6 Netlist

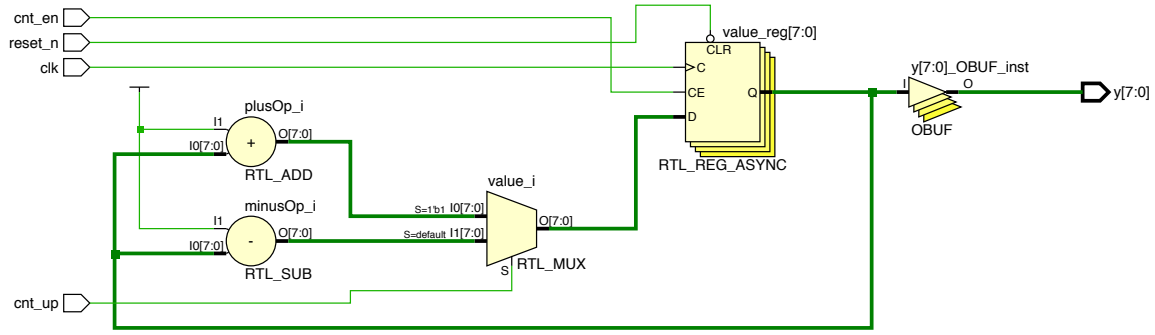


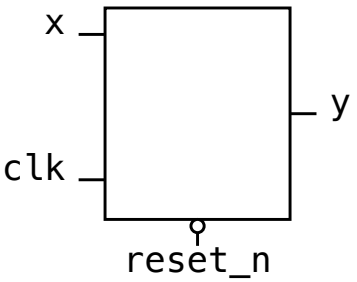
Figure 5: RTL schematic for COUNTER_WIDTH = 8

5 Positive edge detector

A positive edge detector asserts its output, y, for up to one clock cycle after a rising edge of the input, x, has been seen.

5.1 Interface

```
entity pos_edge_detector is
  port (
    clk     : in  std_logic;
    reset_n : in  std_logic;
    x       : in  std_logic;
    y       : out std_logic);
end pos_edge_detector;
```



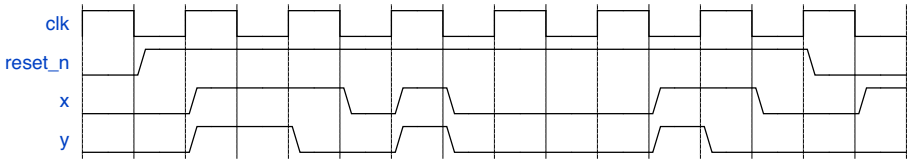
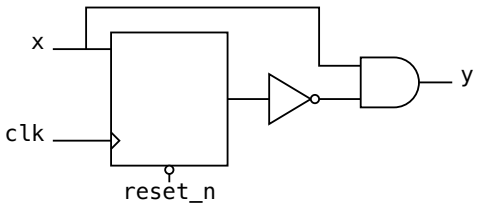


Figure 6: Example waveform

5.2 Microarchitecture

Resetting the flip-flop to '0' will generate a glitch given that x is high. Therefore, using a preset, we 'reset' the flip-flop to '1'.



5.3 Microarchitecture characteristics

Combinatorial	Sequential	Number of flip flops
✓	✓	1

5.4 Source

```
1  -- ******************************************************************************
2  -- Name:        pos_edge_detector.vhd
3  -- Created:     13.02.16 @ NTNU
4  -- Author:      Jonas Eggen
5  -- Purpose:     A positive edge detector. y goes high at the rising edge of x,
6  --              before going low at the first rising edge of the clock.
7  -- ******************************************************************************
8
9  library ieee;
10 use ieee.std_logic_1164.all;
11 use ieee.numeric_std.all;
12
13 entity pos_edge_detector is
14   port (
15     clk     : in  std_logic;
16     reset_n : in  std_logic;
17     x       : in  std_logic;
```

```
18      y       : out std_logic);
19  end pos_edge_detector;
20
21  architecture rtl of pos_edge_detector is
22    signal x_d1 : std_logic;
23  begin
24    process(clk, reset_n)
25    begin
26      if(reset_n = '0') then
27        x_d1 <= '1';
28      elsif (clk'event and clk='1') then
29        x_d1 <= x;
30      end if;
31    end process;
32    y <= x and (not x_d1);
33  end rtl;
```

5.5 PPA

Table 3: Resource consumption for target device Xilinx Zync XC7Z020-CLG484-1

	Synthesised	Available Resources
Speed	555 555 555 555 555 MHz	
FF	1	106400
LUT	2	53200
I/O	4	200
BUFG	1	32

5.6 Netlist

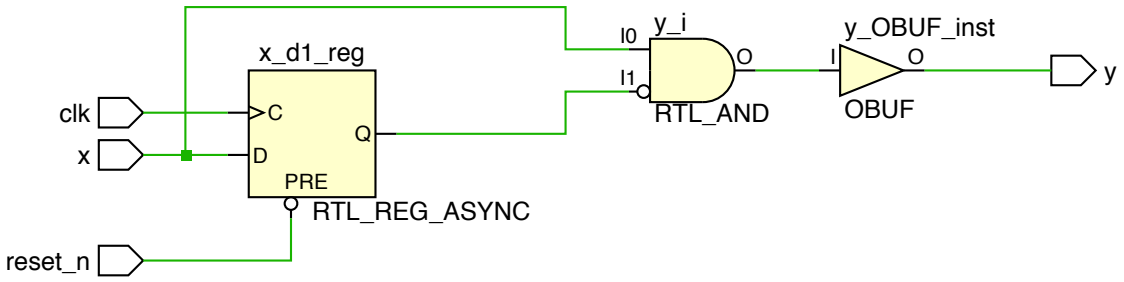


Figure 7: RTL schematic

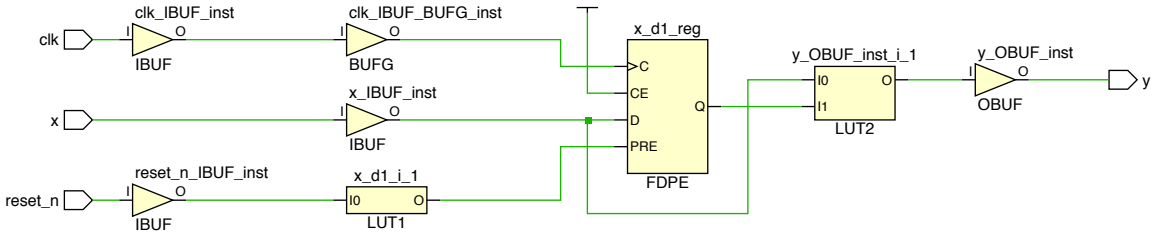


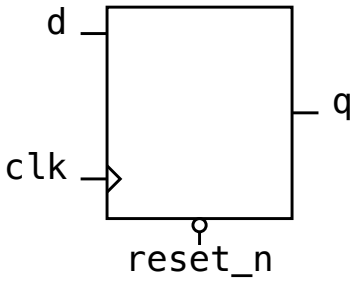
Figure 8: Synthesised schematic

6 DFF with reset_n

Single positive edge triggered D flip-flop with asynchronous reset, active low.

6.1 Interface

```
entity dff_reset_n is
  port (
    clk     : in  std_logic;
    reset_n : in  std_logic;
    d       : in  std_logic;
    q       : out std_logic);
end dff_reset_n;
```



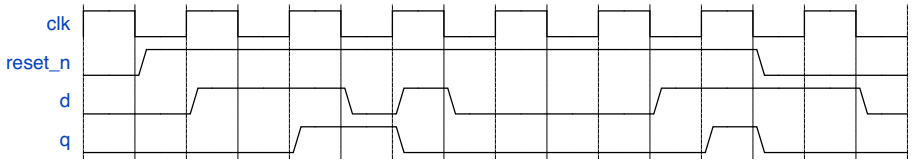


Figure 9: Example waveform

6.2 Microarchitecture characteristics

Combinatorial	Sequential	Number of flip flops
	✓	1

6.3 Source

```
1  -- ******************************************************************************
2  -- Name:       dff_reset_n.vhd
3  -- Created:    13.02.16 @ NTNU
4  -- Author:     Jonas Eggen
5  -- Purpose:    Single positive edge triggered flip flop with asynchronous reset,
6  --             active low.
7  -- ******************************************************************************
8
9  library ieee;
10 use ieee.std_logic_1164.all;
11 use ieee.numeric_std.all;
12
13 entity dff_reset_n is
14   port (
15     clk     : in  std_logic;
16     reset_n : in  std_logic;
17     d       : in  std_logic;
18     q       : out std_logic);
19 end dff_reset_n;
20
21 architecture rtl of dff_reset_n is
22 begin
23   process(clk, reset_n)
24   begin
25     if(reset_n = '0') then
26       q <= '0';
27     elsif (clk'event and clk='1') then
28       q <= d;
29     end if;
30   end process;
31 end rtl;
```

6.4 PPA

Table 4: Resource consumption for target device Xilinx Zync XC7Z020-CLG484-1

	Synthesised	Available Resources
Speed	555 555 555 555 555 MHz	
FF	1	106400
LUT	1	53200
I/O	4	200
BUFG	1	32

Note that a LUT is being used. This could be avoided by using active-high reset.

6.5 Netlist

Note that in Figure 11, `reset_n` is inverted (in LUT1) in order to match the interface of the FDCE. Please see Section 2.1 for more info about this.

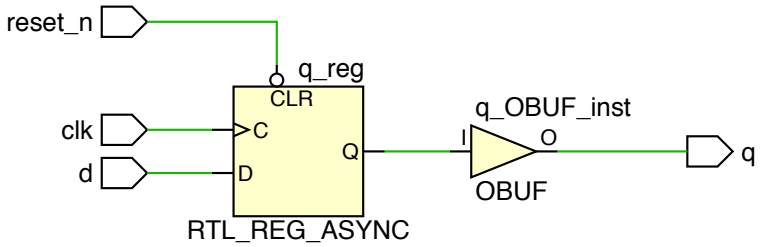


Figure 10: RTL schematic

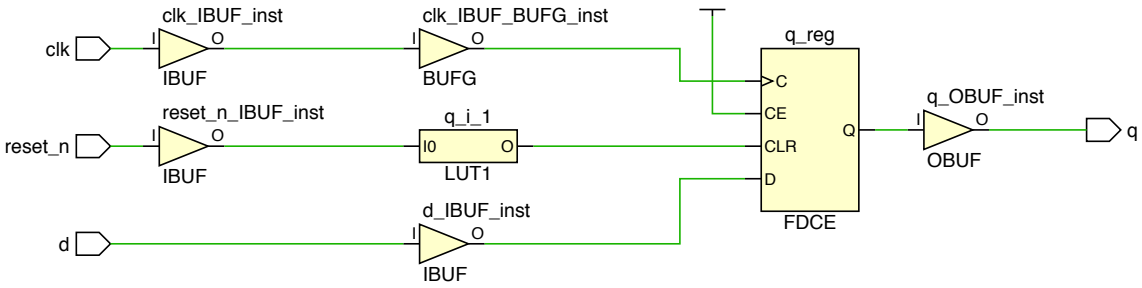


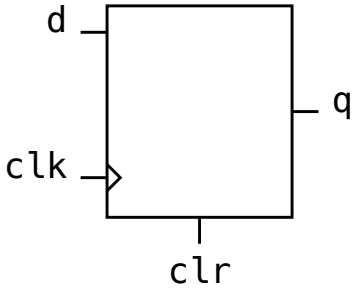
Figure 11: Synthesised schematic

7 DFF with clr

Single positive edge triggered D flip-flop with synchronous reset.

7.1 Interface

```
entity dff_clr is
  port (
    clk     : in  std_logic;
    reset_n : in  std_logic;
    d       : in  std_logic;
    q       : out std_logic);
end dff_clr;
```



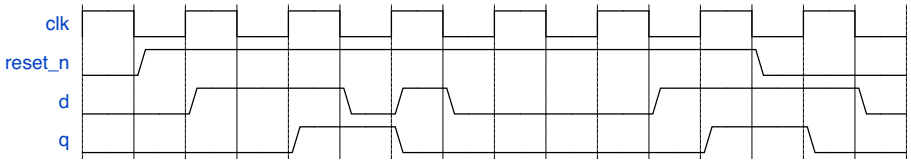


Figure 12: Example waveform

7.2 Microarchitecture characteristics

Combinatorial	Sequential	Number of flip flops
	✓	1

7.3 Source

```
1  -- ******************************************************************************
2  -- Name:       dff_clr.vhd
3  -- Created:    13.02.16 @ NTNU
4  -- Author:     Jonas Eggen
5  -- Purpose:    Single positive edge triggered flip flop with synchronous reset.
6  -- ******************************************************************************
7
8  library ieee;
9  use ieee.std_logic_1164.all;
10 use ieee.numeric_std.all;
11
12 entity dff_clr is
13   port (
14     clk     : in  std_logic;
15     reset_n : in  std_logic;
16     d       : in  std_logic;
17     q       : out std_logic);
18 end dff_clr;
19
20 architecture rtl of dff_clr is
21 begin
22   process(clk, reset_n)
23   begin
24     if (clk'event and clk='1') then
25       if(reset_n = '0') then
26         q <= '0';
27       else
28         q <= d;
29       end if;
30     end if;
31   end process;
32 end rtl;
```

7.4 PPA

Table 5: Resource consumption for target device Xilinx Zync XC7Z020-CLG484-1

	Synthesised	Available Resources
Speed	555 555 555 555 555 MHz	
FF	1	106400
LUT	1	53200
I/O	4	200
BUFG	1	32

Note that a LUT is being used. This could be avoided by using active-high reset.

7.5 Netlist

Note that in Figure 14, `reset_n` is combined with d (in LUT2) in order to match the interface of the FDRE. Please see Section 2.1 for more info about this. By using an active-high reset, a DFF with an asynchronous reset would be inferred. The effect is also seen in Figure 13.

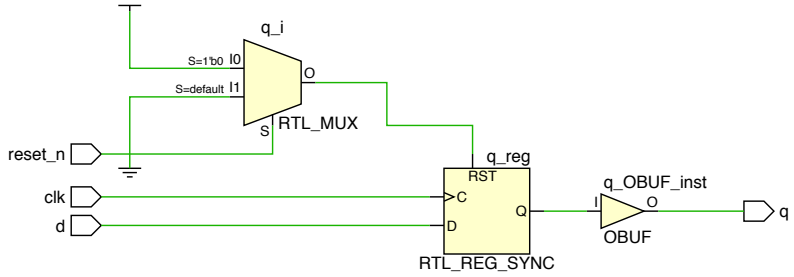


Figure 13: RTL schematic

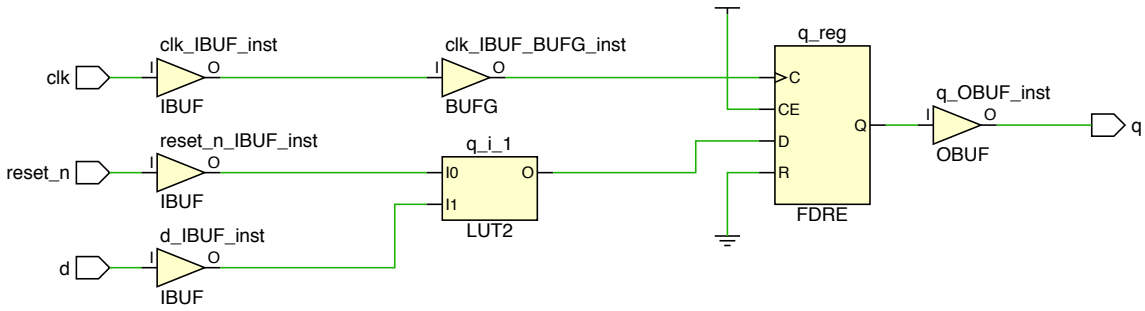


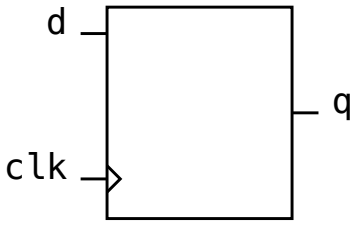
Figure 14: Synthesised schematic

8 DFF

Single positive edge triggered D flip-flop.

8.1 Interface

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



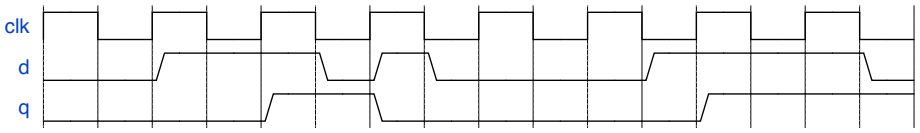


Figure 15: Example waveform

8.2 Microarchitecture characteristics

Combinatorial	Sequential	Number of flip flops
	✓	1

8.3 Source

```
1  -- ******************************************************************************
2  -- Name:       dff.vhd
3  -- Created:    13.02.16 @ NTNU
4  -- Author:     Jonas Eggen
5  -- Purpose:    Single positive edge triggered flip flop.
6  -- ******************************************************************************
7
8  library ieee;
9  use ieee.std_logic_1164.all;
10 use ieee.numeric_std.all;
11
12 entity dff is
13   port (
14     clk     : in  std_logic;
15     d       : in  std_logic;
16     q       : out std_logic);
17 end dff;
18
19 architecture rtl of dff is
20 begin
21   process(clk)
22   begin
23     if (clk'event and clk='1') then
24       q <= d;
25     end if;
26   end process;
27 end rtl;
```

8.4 PPA

8.5 Netlist

Table 6: Resource consumption for target device Xilinx Zync XC7Z020-CLG484-1

	Synthesised	Available Resources
Speed	555 555 555 555 555 MHz	
FF	1	106400
LUT	0	53200
I/O	3	200
BUFG	1	32

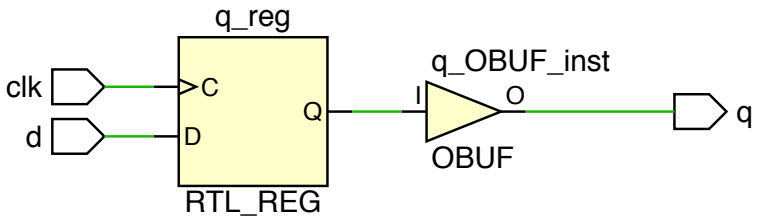


Figure 16: RTL schematic

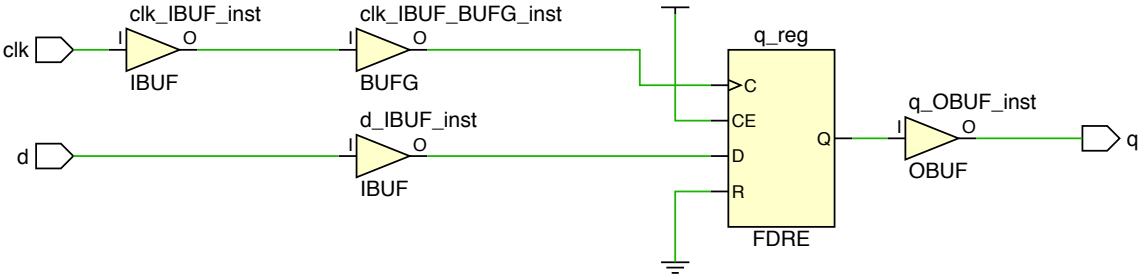


Figure 17: Synthesised schematic

9 Binary decoder

A binary decoder converts a binary input value into an associated pattern of output bits. This binary decoder is an n-to-2^n line decoder. This decoder asserts the output bit position referred to by the binary interpretation of the input. In this implementation, $n =$ INPUT_WIDTH.

9.1 Interface

```
entity bin_decoder is
  generic(
    INPUT_WIDTH : natural := 2);
  port (
    x           : in  std_logic_vector(INPUT_WIDTH-1 downto 0);
    y           : out std_logic_vector((INPUT_WIDTH**2)-1 downto 0));
end bin_decoder;
```

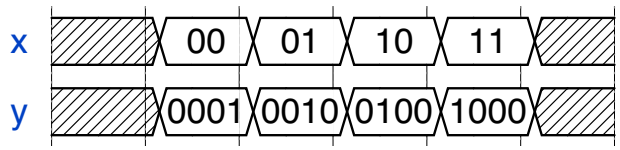


Figure 18: Example waveform for INPUT_WIDTH = 2

9.2 Microarchitecture

The RTL code for the binary decoder is written at a very high level. The synthesis tool should be able to break the description down to an optimal implementation for the target technology, but if you are in doubt, it is always good to check whether the synthesized netlist matches the result you had in mind.

A block diagram illustrating the logic depth for the binary decoder is given in Figure 19. Compare the hand-synthesized version with the netlist generated by the synthesis tool in Figure 20 and Figure 21.

9.3 Microarchitecture characteristics

Combinatorial	Sequential	Number of flip flops
✓		0

9.4 Source

```
1  -- ******************************************************************************
2  -- Name:       bin_decoder.vhd
3  -- Created:    20.02.16 @ NTNU
4  -- Author:     Jonas Eggen
5  -- Purpose:    A parameterisable binary decoder.
6  -- Example:    INPUT_WIDTH of 2:
7  --             x1  x0  |  y3  y2  y1  y0
8  --             ------------------------
9  --              0   0  |   0   0   0   1
10 --              0   1  |   0   0   1   0
11 --              1   0  |   0   1   0   0
12 --              1   1  |   1   0   0   0
13 -- ******************************************************************************
14
15 library ieee;
16 use ieee.std_logic_1164.all;
17 use ieee.numeric_std.all;
18
19 entity bin_decoder is
20   generic(
21     INPUT_WIDTH : natural := 2);
```

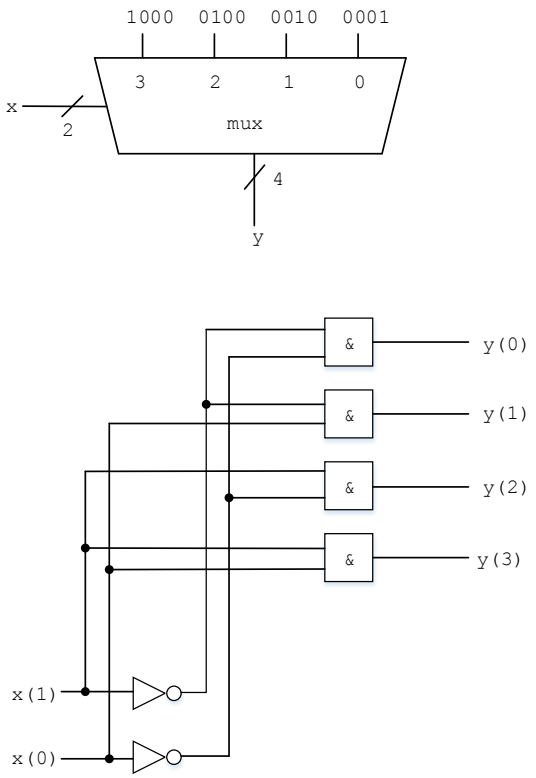


Figure 19: Binary decoder synthesized by hand for INPUT_WIDTH = 2

```
22    port (
23      x           : in  std_logic_vector(INPUT_WIDTH-1 downto 0);
24      y           : out std_logic_vector((INPUT_WIDTH**2)-1 downto 0));
25  end bin_decoder;
26
27  architecture rtl of bin_decoder is
28  begin
29    gen : for i in y'range generate
30      y(i) <= '1' when unsigned(x)=to_unsigned(i,x'length) else '0';
31    end generate;
32  end rtl;
```

9.5 PPA

Table 7: Resource consumption for target device Xilinx Zync XC7Z020-CLG484-1

Generic	INPUT_WIDTH = 2	INPUT_WIDTH = 4	Available Resources
LUT	2	8	53200
I/O	6	20	200

9.6 Netlist

As mentioned, this circuit is perfect LUT food (Figure 21).

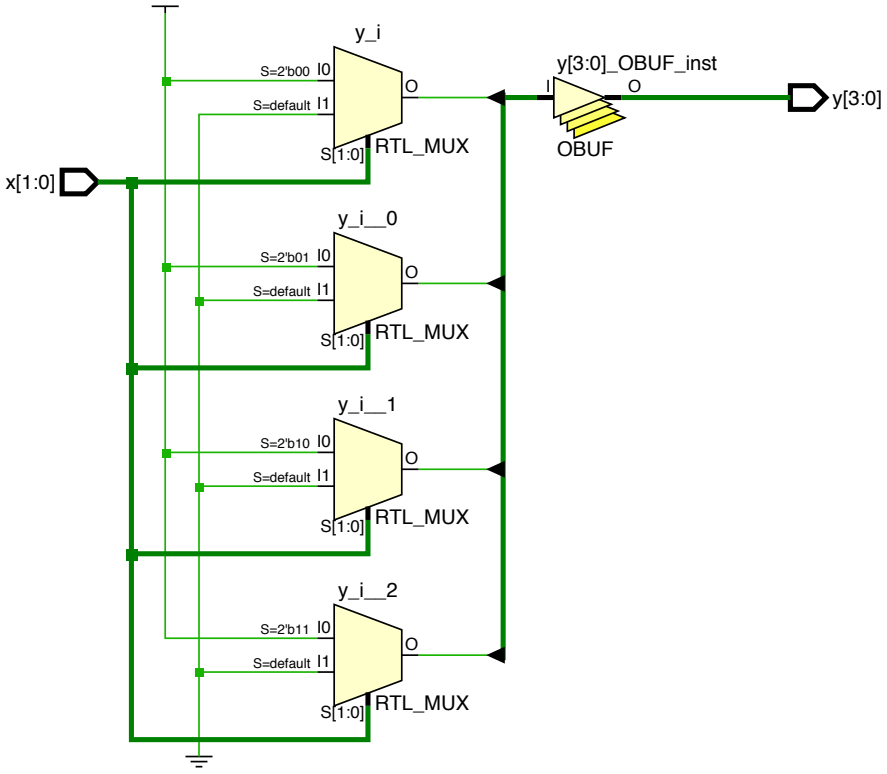


Figure 20: RTL schematic for INPUT_WIDTH = 2

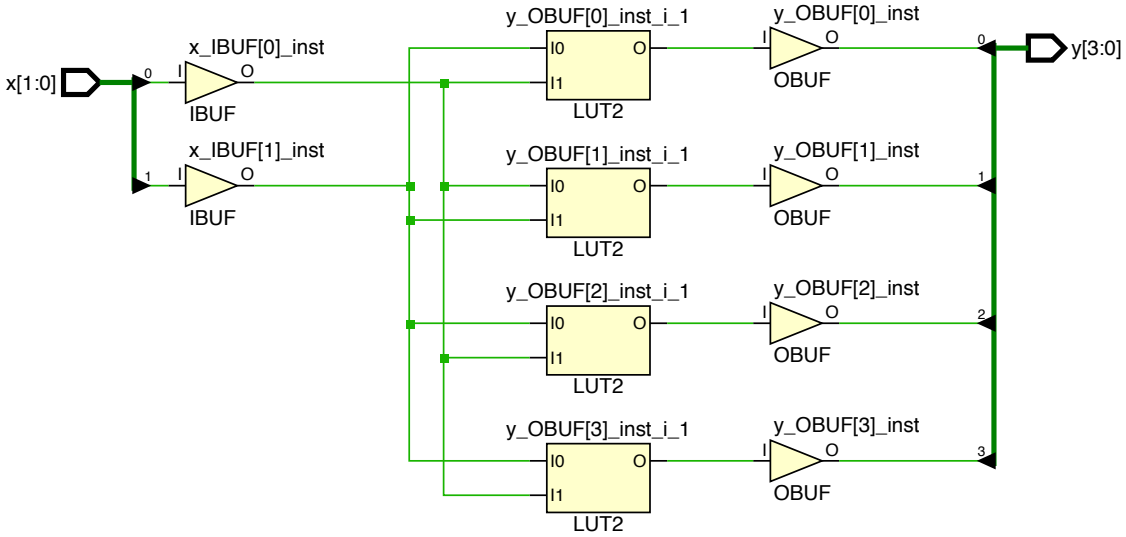


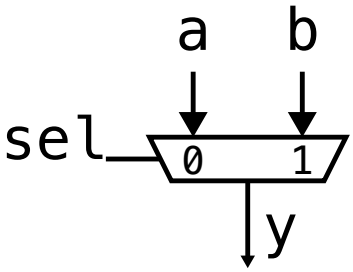
Figure 21: Synthesised schematic for INPUT_WIDTH = 2

10 2-to-1 multiplexer

A two input multiplexer with an input bit with of 1.

10.1 Interface

```
entity mux2x1 is
  port (
    a       : in  std_logic;
    b       : in  std_logic;
    sel     : in  std_logic;
    y       : out std_logic);
end mux2x1;
```



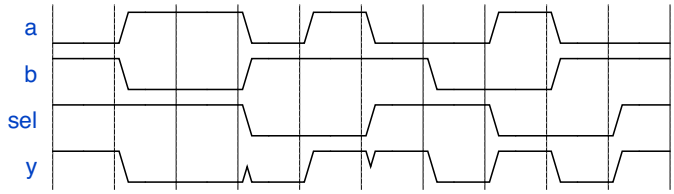


Figure 22: Example waveform

10.2 Microarchitecture

A 2x1 mux is a primitive in standard-cell technologies. For FPGAs a 1-bit 2x1 mux will fit into a single LUT. It is interesting though to try to synthesize the mux down to and/or/not gates. This is an exercise left for the reader.

10.3 Microarchitecture characteristics

Combinatorial	Sequential	Number of flip flops
✓		0

10.4 Source

```
1  -- ******************************************************************************
2  -- Name:       mux2x1.vhd
3  -- Created:    20.02.16 @ NTNU
4  -- Author:     Jonas Eggen
5  -- Purpose:    A two input mux with an input bit width of 1.
6  -- ******************************************************************************
7
8  library ieee;
9  use ieee.std_logic_1164.all;
10 use ieee.numeric_std.all;
11
12 entity mux2x1 is
13   port (
14     a       : in  std_logic;
15     b       : in  std_logic;
16     sel     : in  std_logic;
17     y       : out std_logic);
18 end mux2x1;
19
20 architecture rtl of mux2x1 is
21 begin
22   y <= a when sel='0' else b;
23 end rtl;
```

10.5 PPA

Table 8: Resource consumption for target device Xilinx Zync XC7Z020-CLG484-1

	Synthesised	Available Resources
LUT	1	53200
I/O	4	200

10.6 Netlist

As mentioned, this circuit is perfect LUT food (Figure 24).

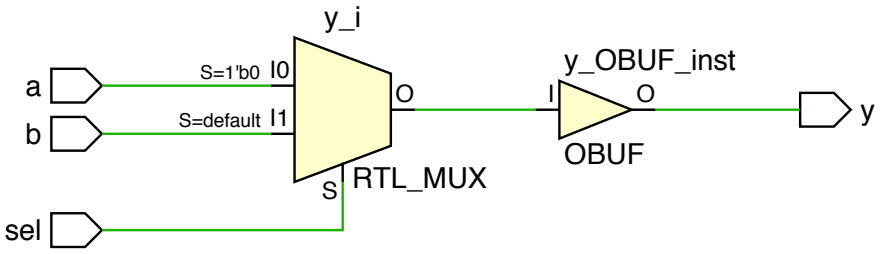


Figure 23: RTL schematic

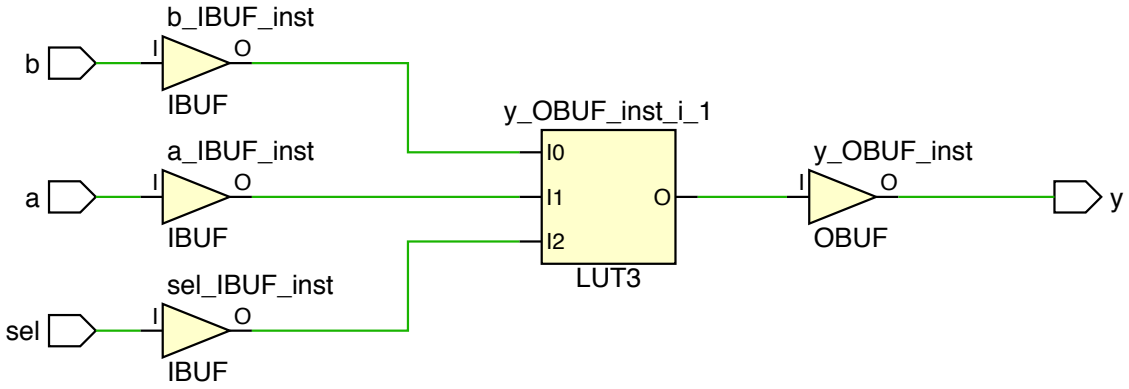


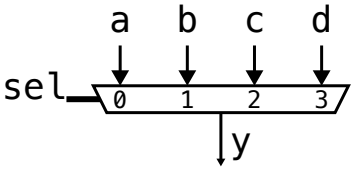
Figure 24: Synthesised schematic

11 4-to-1 multiplexer

A four input multiplexer with an input bit with of 1.

11.1 Interface

```
entity mux4x1 is
  port (
    a       : in  std_logic;
    b       : in  std_logic;
    c       : in  std_logic;
    d       : in  std_logic;
    sel     : in  std_logic_vector(1 downto 0);
    y       : out std_logic);
end mux4x1;
```



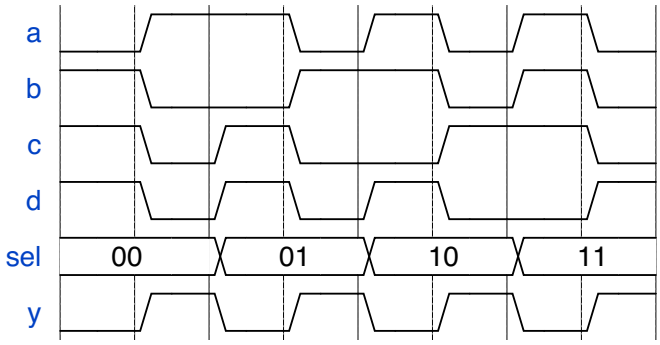


Figure 25: Example waveform

11.2 Microarchitecture

A 4x1 mux can be created from 2x1 mux-es. This is an exercise left for the reader.

11.3 Microarchitecture characteristics

Combinatorial	Sequential	Number of flip flops
✓		0

11.4 Source

```
1  -- ******************************************************************************
2  -- Name:       mux4x1.vhd
3  -- Created:    20.02.16 @ NTNU
4  -- Author:     Jonas Eggen
5  -- Purpose:    A four input mux with an input bit width of 1.
6  -- ******************************************************************************
7
8  library ieee;
9  use ieee.std_logic_1164.all;
10 use ieee.numeric_std.all;
11
12 entity mux4x1 is
13   port (
14     a       : in  std_logic;
15     b       : in  std_logic;
16     c       : in  std_logic;
17     d       : in  std_logic;
18     sel     : in  std_logic_vector(1 downto 0);
19     y       : out std_logic);
```

```
20  end mux4x1;
21
22  architecture rtl of mux4x1 is
23  begin
24    y <= a when sel="00" else
25         b when sel="01" else
26         c when sel="10" else
27         d;
28  end rtl;
```

11.5 PPA

Table 9: Resource consumption for target device Xilinx Zync XC7Z020-CLG484-1

	Synthesised	Available Resources
LUT	1	53200
I/O	7	200

11.6 Netlist

As mentioned, this circuit is perfect LUT food (Figure 27).

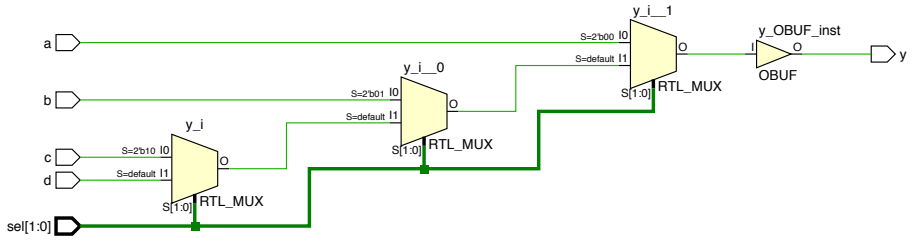


Figure 26: RTL schematic

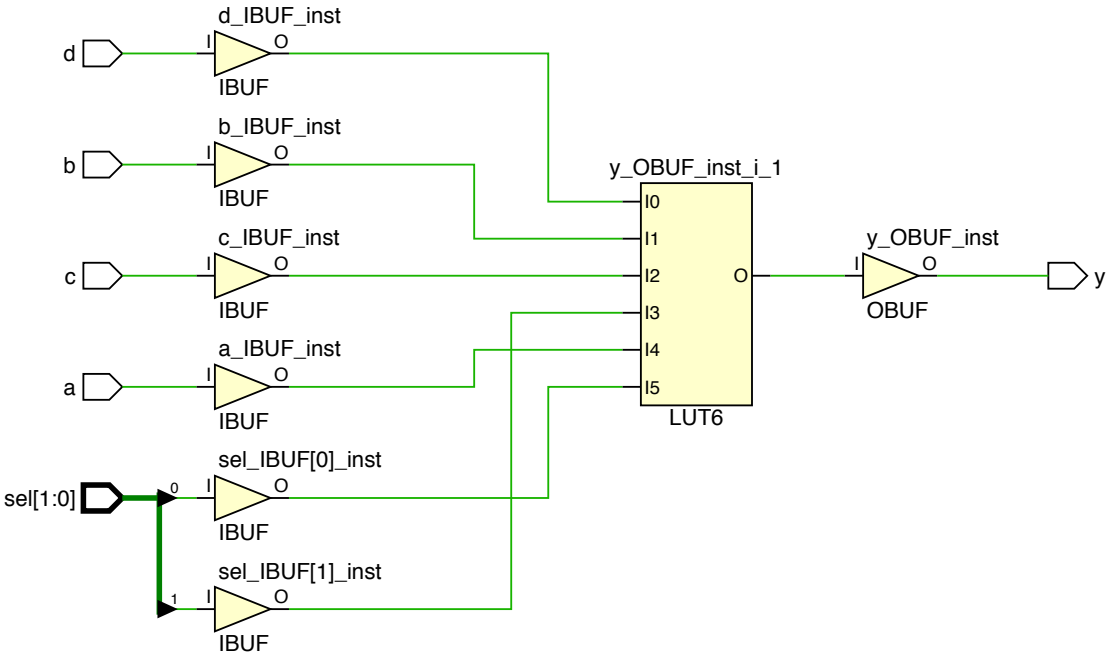


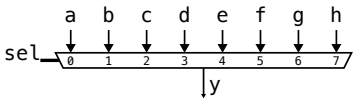
Figure 27: Synthesised schematic

12 8-to-1 multiplexer

An eight input multiplexer with an input bit with of 1.

12.1 Interface

```
entity mux8x1 is
  port (
    a   : in  std_logic;
    b   : in  std_logic;
    c   : in  std_logic;
    d   : in  std_logic;
    e   : in  std_logic;
    f   : in  std_logic;
    g   : in  std_logic;
    h   : in  std_logic;
    sel : in  std_logic_vector(2 downto 0);
    y   : out std_logic);
end mux8x1;
```



12.2 Microarchitecture

A 8x1 mux can be created from 2x1 mux-es. This is an exercise left for the reader.

12.3 Microarchitecture characteristics

Combinatorial	Sequential	Number of flip flops
✓		0

12.4 Source

```
1  -- ******************************************************************************
2  -- Name:       mux8x1.vhd
3  -- Created:    20.02.16 @ NTNU
4  -- Author:     Jonas Eggen
5  -- Purpose:    An eight input mux with an input bit width of 1.
6  -- ******************************************************************************
7
8  library ieee;
9  use ieee.std_logic_1164.all;
10 use ieee.numeric_std.all;
11
12 entity mux8x1 is
13   port (
14     a   : in  std_logic;
15     b   : in  std_logic;
16     c   : in  std_logic;
17     d   : in  std_logic;
18     e   : in  std_logic;
19     f   : in  std_logic;
20     g   : in  std_logic;
21     h   : in  std_logic;
22     sel : in  std_logic_vector(2 downto 0);
23     y   : out std_logic);
24 end mux8x1;
25
26 architecture rtl of mux8x1 is
27 begin
28   y <= a when sel="000" else
29        b when sel="001" else
30        c when sel="010" else
31        d when sel="011" else
32        e when sel="100" else
33        f when sel="101" else
34        g when sel="110" else
35        h;
36 end rtl;
```

12.5 PPA

Table 10: Resource consumption for target device Xilinx Zync XC7Z020-CLG484-1

	Synthesised	Available Resources
LUT	2	53200
I/O	12	200

12.6 Netlist

As mentioned, this circuit is perfect LUT food (Figure 29). Note that a MUXF7 has been inserted in Figure 29. This is expected and described on page 308 in `http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/7series_hdl.pdf`

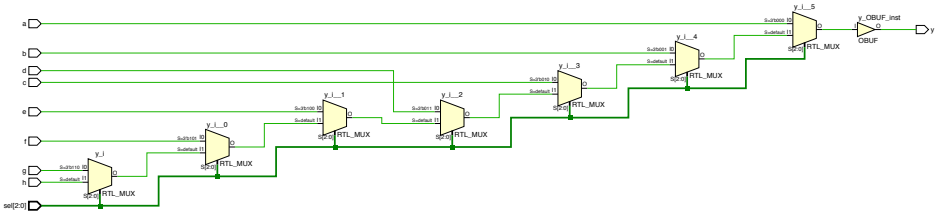


Figure 28: RTL schematic

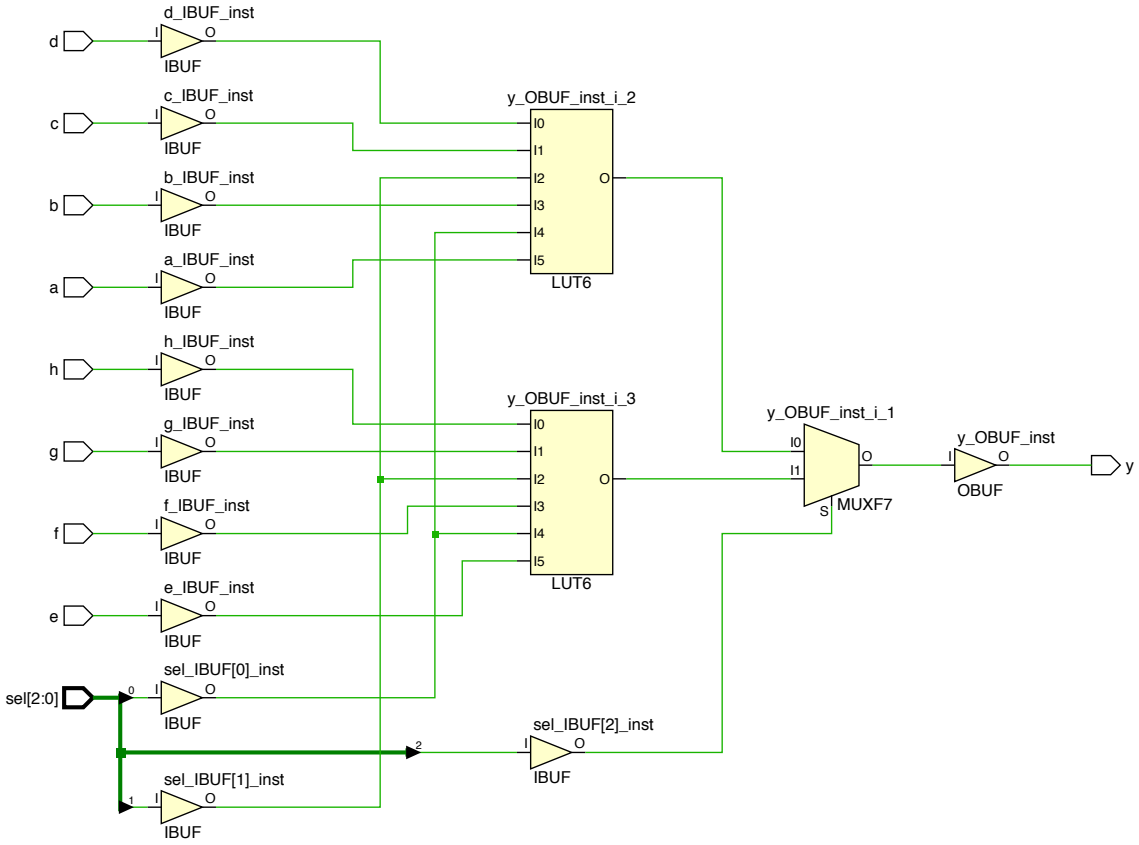


Figure 29: Synthesised schematic

13 Priority encoder

A priority encoder takes in a vector of bits and computes the index of the most significant bit that is set in the input vector. Such circuits can be used as building blocks in for example arbiters. The output is only valid if at least one of the input bits are set.

13.1 Interface

```
entity priority_encoder is
  generic(
    OUTPUT_WIDTH : natural := 2);
  port (
    x            : in  std_logic_vector(OUTPUT_WIDTH**2-1 downto 0);
    y            : out std_logic_vector(OUTPUT_WIDTH-1 downto 0);
    valid        : out std_logic);
end priority_encoder;
```

Note that OUTPUT_WIDTH is specified instead of INPUT_WIDTH. This could be changed by using `log2c` from Section 23.1.

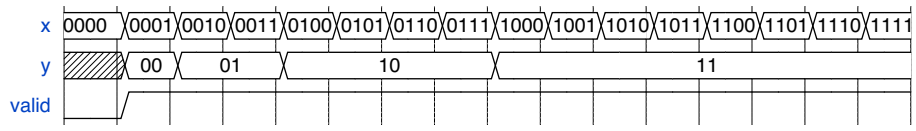


Figure 30: Example waveform for OUTPUT_WIDTH = 2

13.2 Microarchitecture

A block diagram illustrating the logic depth for the priority encoder is given in Figure 31.

13.3 Microarchitecture characteristics

Combinatorial	Sequential	Number of flip flops
✓		0

13.4 Source

```
1  -- ******************************************************************************
2  -- Name:       priority_encoder.vhd
3  -- Created:    20.02.16 @ NTNU
4  -- Author:     Jonas Eggen
5  -- Purpose:    A parameterisable priority encoder.
6  -- Example:    OUTPUT_WIDTH of 2:
7  --             x3  x2  x1  x0  |  y1  y0  valid
8  --             ----------------+--------------
9  --              0   0   0   0  |   x   x      0
10 --              0   0   0   1  |   0   0      1
11 --              0   0   1   x  |   0   1      1
12 --              0   1   x   x  |   1   0      1
13 --              1   x   x   x  |   1   1      1
14 -- ******************************************************************************
15
16 library ieee;
17 use ieee.std_logic_1164.all;
18 use ieee.numeric_std.all;
19 use ieee.std_logic_misc.all;
20
21 entity priority_encoder is
22   generic(
23     OUTPUT_WIDTH : natural := 2);
24   port (
25     x            : in  std_logic_vector(OUTPUT_WIDTH**2-1 downto 0);
```

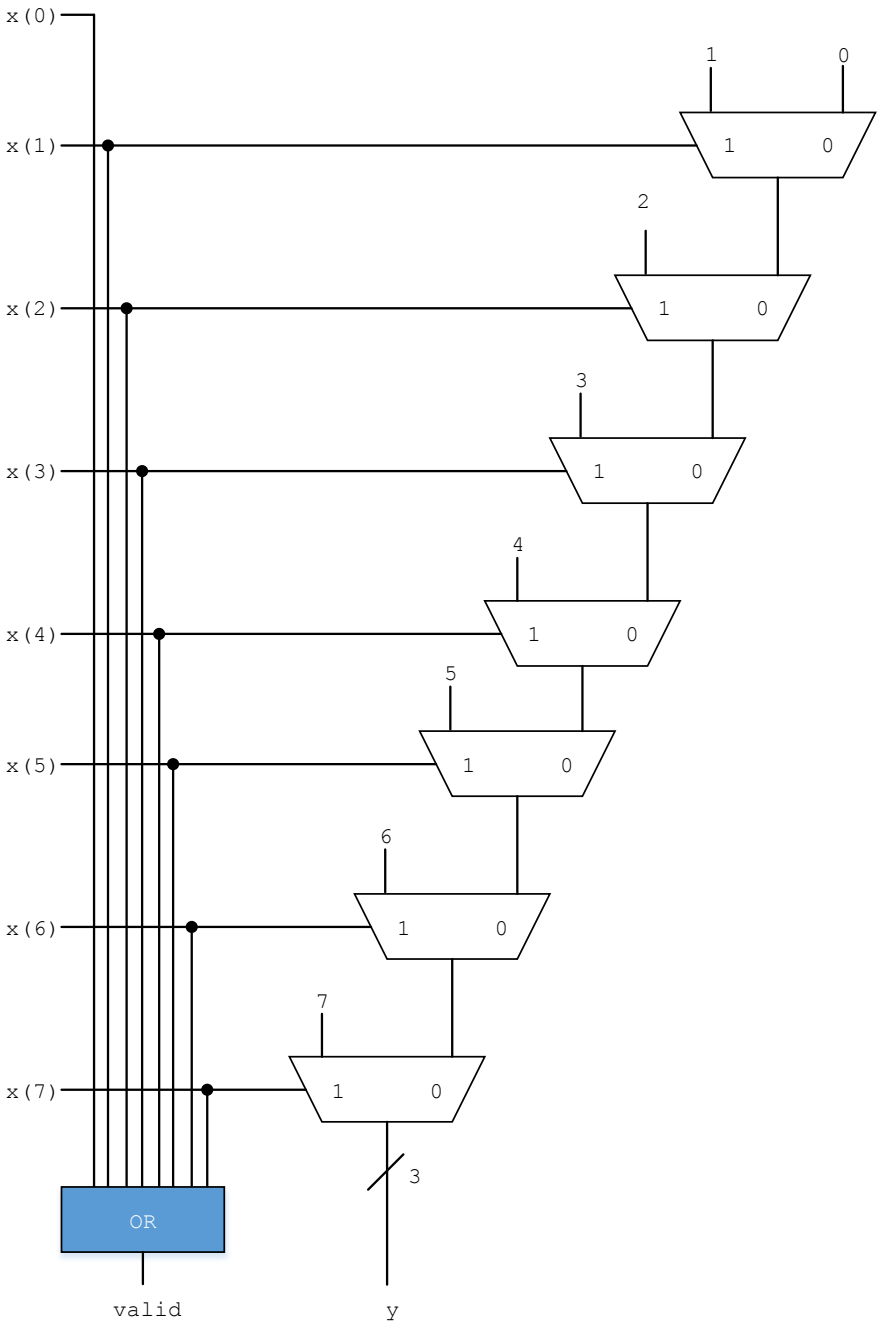


Figure 31: Microarchitecture INPUT_WIDTH = 8

```
26      y             : out std_logic_vector(OUTPUT_WIDTH-1 downto 0);
27      valid         : out std_logic);
28  end priority_encoder;
29
30  architecture rtl of priority_encoder is
31    function pri_encode (x: std_logic_vector) return std_logic_vector is
32      variable dontcare: std_logic_vector(OUTPUT_WIDTH-1 downto 0) := (others => '-');
33    begin
34      for i in x'range loop
35        if (x(i)='1') then
36          return std_logic_vector(to_unsigned(i,OUTPUT_WIDTH));
37        end if;
38      end loop;
39      return dontcare;
40    end function;
41  begin
42    y     <= pri_encode(x);
43    valid <= or_reduce(x);
44  end rtl;
```

The function or_reduce is part of the package std_logic_misc.

13.5 PPA

Table 11: Resource consumption for target device Xilinx Zync XC7Z020-CLG484-1

Generic	OUTPUT_WIDTH = 2	OUTPUT_WIDTH = 4	Available Resources
LUT	2	12	53200
I/O	7	21	200

13.6 Netlist

As mentioned, this circuit is perfect LUT food (Figure 33).

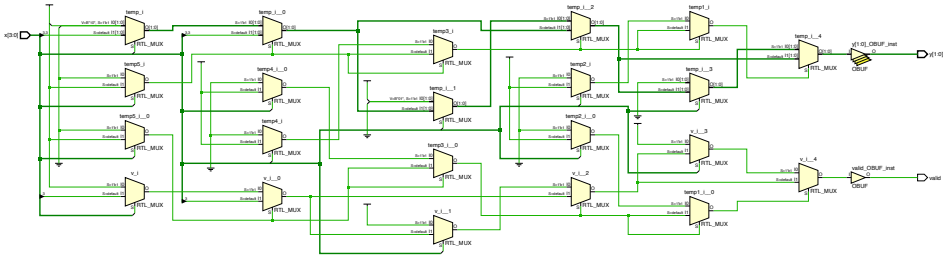


Figure 32: RTL schematic for OUTPUT_WIDTH = 2

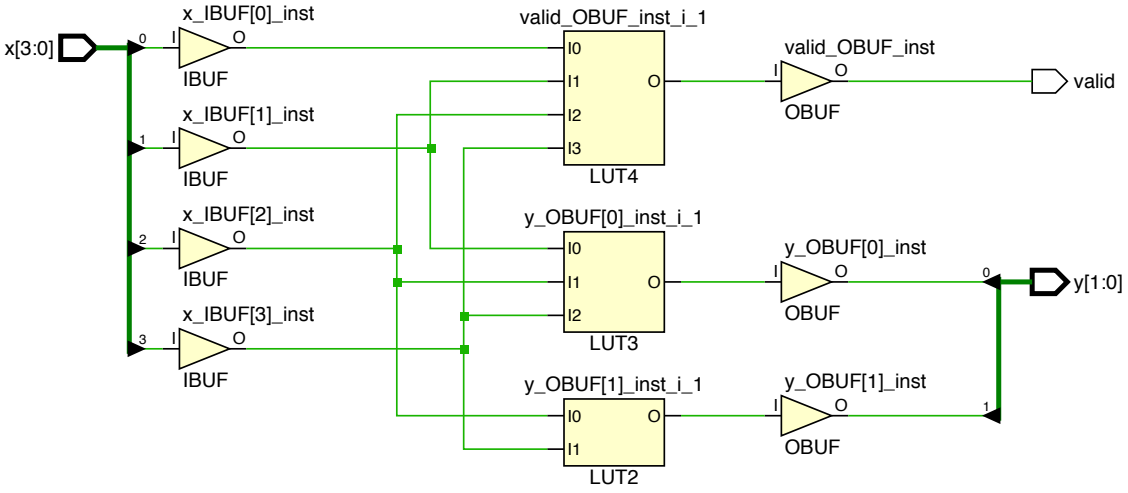


Figure 33: Synthesised schematic for OUTPUT_WIDTH = 2

14 Register with reset_n

One of the most important basic building blocks is the n-bit register.

14.1 Interface

```
entity register_reset_n is
  generic (
    REGISTER_WIDTH : natural := 8);
  port (
    clk     : in  std_logic;
    reset_n : in  std_logic;
    d       : in  std_logic_vector(REGISTER_WIDTH-1 downto 0);
    q       : out std_logic_vector(REGISTER_WIDTH-1 downto 0));
end register_reset_n;
```

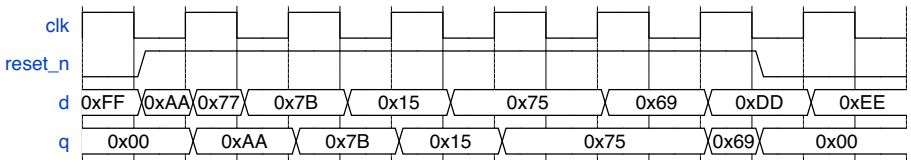


Figure 34: Example waveform for REGISTER_WIDTH = 8

14.2 Microarchitecture

An n-bit register, is just n flip-flops bundled together into a register.

14.3 Microarchitecture characteristics

Combinatorial	Sequential	Number of flip flops
	✓	REGISTER_WIDTH

14.4 Source

```
1  -- ******************************************************************************
2  -- Name:        register_reset_n.vhd
3  -- Created:     20.02.16 @ NTNU
4  -- Author:      Jonas Eggen
5  -- Purpose:     Positive edge triggered register with asynchronous reset,
6  --              active low.
7  -- ******************************************************************************
8
9  library ieee;
10 use ieee.std_logic_1164.all;
11 use ieee.numeric_std.all;
12
13 entity register_reset_n is
14   generic (
15     REGISTER_WIDTH : natural := 8);
16   port (
17     clk     : in  std_logic;
18     reset_n : in  std_logic;
19     d       : in  std_logic_vector(REGISTER_WIDTH-1 downto 0);
20     q       : out std_logic_vector(REGISTER_WIDTH-1 downto 0));
21 end register_reset_n;
22
23 architecture rtl of register_reset_n is
24 begin
25   process(clk, reset_n)
26   begin
27     if(reset_n = '0') then
28       q <= (others => '0');
```

```
29      elsif (clk'event and clk='1') then
30        q <= d;
31      end if;
32    end process;
33  end rtl;
```

14.5 PPA

Table 12: Resource consumption for target device Xilinx Zync XC7Z020-CLG484-1

Generic	REGISTER_WIDTH = 8	Available Resources
FF	8	53200
LUT	1	53200
I/O	18	200
BUFG	1	200

Note that a LUT is being used. This could be avoided by using active-high reset.

14.6 Netlist

Note that in Figure 36, `reset_n` is inverted (in LUT1) in order to match the interface of the FDCE. Please see Section 2.1 for more info about this.

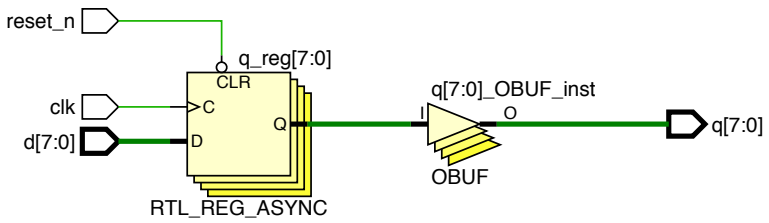


Figure 35: RTL schematic for REGISTER_WIDTH = 8

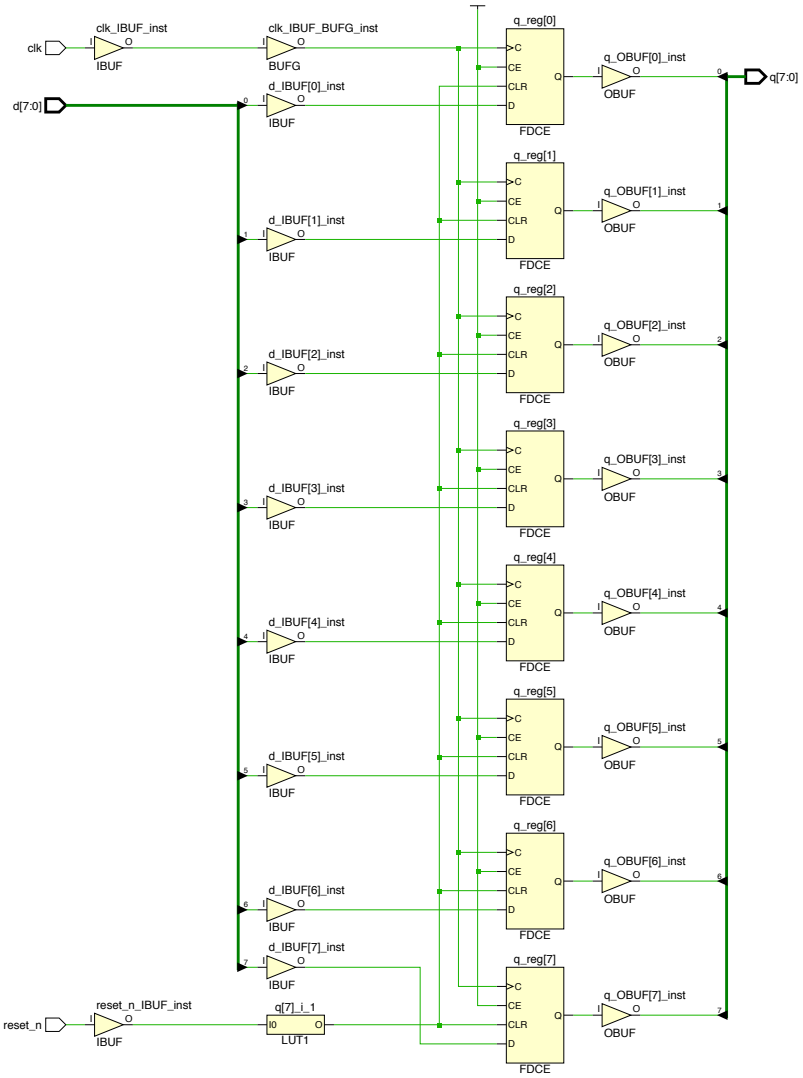


Figure 36: Synthesised schematic for REGISTER_WIDTH = 8

15 Register

One of the most important basic building blocks is the n-bit register.

15.1 Interface

```
entity reg is
  generic (
    REGISTER_WIDTH : natural := 8);
  port (
    clk      : in  std_logic;
    d        : in  std_logic_vector(REGISTER_WIDTH-1 downto 0);
    q        : out std_logic_vector(REGISTER_WIDTH-1 downto 0));
end reg;
```

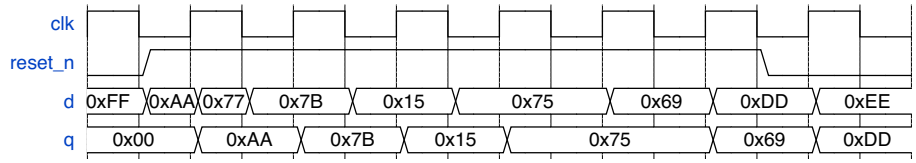


Figure 37: Example waveform for REGISTER_WIDTH = 8

15.2 Microarchitecture

An n-bit register, is just n flip-flops bundled together into a register.

15.3 Microarchitecture characteristics

Combinatorial	Sequential	Number of flip flops
	✓	REGISTER_WIDTH

15.4 Source

```
1  -- ******************************************************************************
2  -- Name:       reg.vhd
3  -- Created:    20.02.16 @ NTNU
4  -- Author:     Jonas Eggen
5  -- Purpose:    Positive edge triggered register.
6  -- ******************************************************************************
7
8  library ieee;
9  use ieee.std_logic_1164.all;
10 use ieee.numeric_std.all;
11
12 entity reg is
13   generic (
14     REGISTER_WIDTH : natural := 8);
15   port (
16     clk      : in  std_logic;
17     d        : in  std_logic_vector(REGISTER_WIDTH-1 downto 0);
18     q        : out std_logic_vector(REGISTER_WIDTH-1 downto 0));
19 end reg;
20
21 architecture rtl of reg is
22 begin
23   process(clk)
24   begin
25     if (clk'event and clk='1') then
26       q <= d;
27     end if;
28   end process;
29 end rtl;
```

15.5 PPA

Table 13: Resource consumption for target device Xilinx Zync XC7Z020-CLG484-1

Generic	REGISTER_WIDTH = 8	Available Resources
FF	8	53200
I/O	18	200
BUFG	1	200

15.6 Netlist

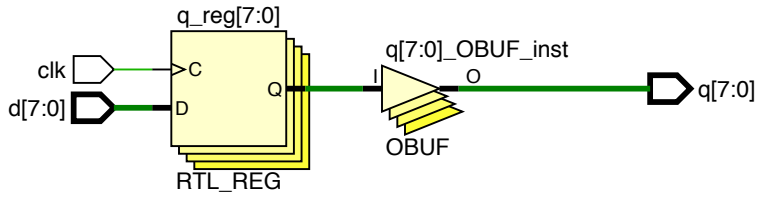


Figure 38: RTL schematic for REGISTER_WIDTH = 8

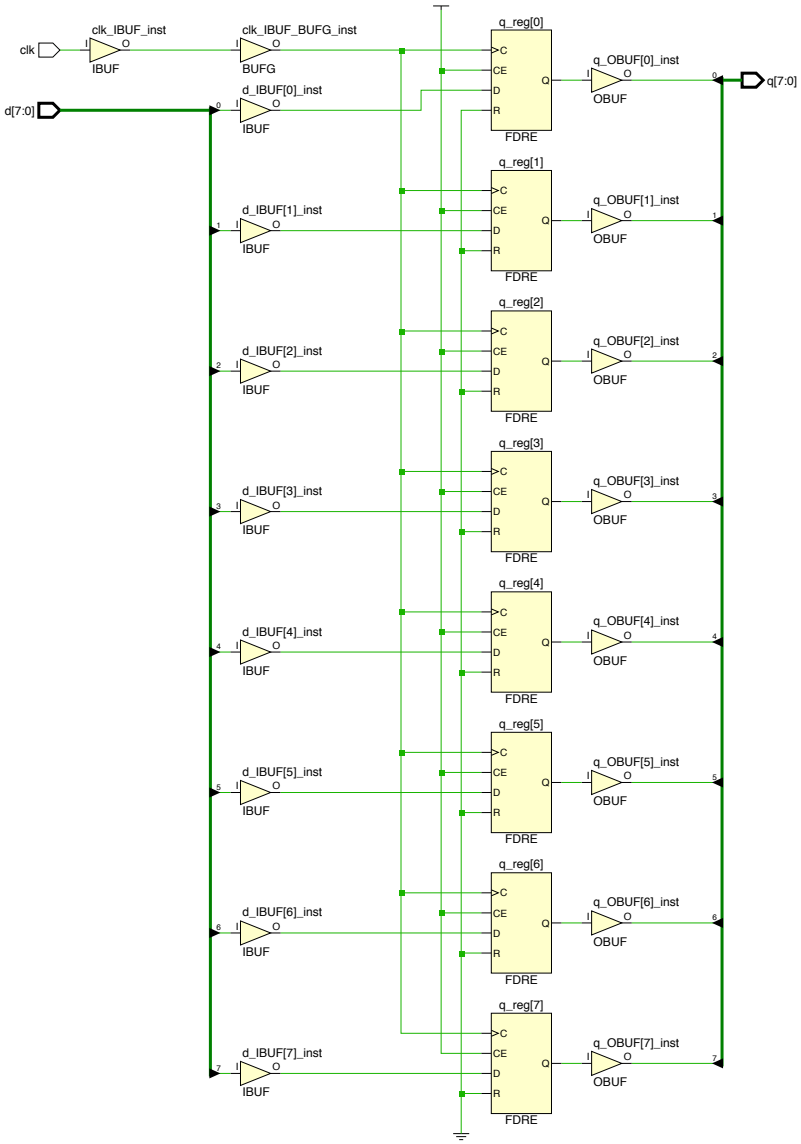


Figure 39: Synthesised schematic for REGISTER_WIDTH = 8

16 Shift register

A shift register is a circuit where data can be shifted into a register. This particular implementation is of a Serial-in, Parallel-out type of shift register. This means that serial data is shifted in one bit at a time, while the whole register can be read at the same time.

16.1 Interface

```
entity shift_register is
  generic (
    REGISTER_WIDTH : natural := 8);
  port (
    clk     : in  std_logic;
    reset_n : in  std_logic;
    d       : in  std_logic;
    q       : out std_logic_vector(REGISTER_WIDTH-1 downto 0));
end shift_register;
```

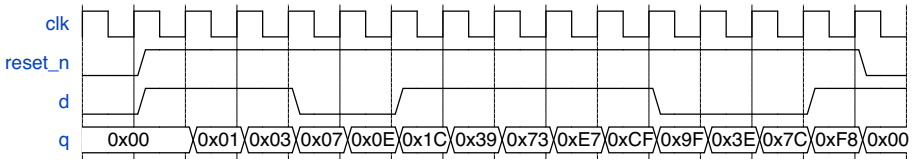


Figure 40: Example waveform for REGISTER_WIDTH = 8

16.2 Microarchitecture

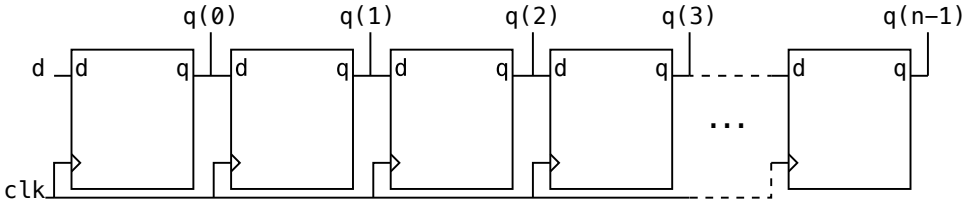


Figure 41: Shift register microarchitecture, where n = REGISTER_WIDTH.

16.3 Microarchitecture characteristics

Combinatorial	Sequential	Number of flip flops
Nja	✓	REGISTER_WIDTH

16.4 Source

```
1  -- ********************************************************************************
2  -- Name:       shift_register.vhd
3  -- Created:    20.02.16 @ NTNU
4  -- Author:     Jonas Eggen
5  -- Purpose:    Positive edge triggered shift register with asynchronous reset,
6  --             active low. Shift in MSB first.
7  -- ********************************************************************************
8
9  library ieee;
10 use ieee.std_logic_1164.all;
11 use ieee.numeric_std.all;
12
13 entity shift_register is
14   generic (
15     REGISTER_WIDTH : natural := 8);
16   port (
17     clk      : in  std_logic;
18     reset_n  : in  std_logic;
19     d        : in  std_logic;
20     q        : out std_logic_vector(REGISTER_WIDTH-1 downto 0));
21 end shift_register;
22
23 architecture rtl of shift_register is
24   signal q_i : std_logic_vector(REGISTER_WIDTH-1 downto 0);
25 begin
26   process(clk, reset_n)
27   begin
28     if(reset_n = '0') then
29       q_i <= (others => '0');
30     elsif (clk'event and clk='1') then
31       q_i <= q_i(REGISTER_WIDTH-2 downto 0) & d;
32     end if;
33   end process;
34   q <= q_i;
35 end rtl;
```

16.5 PPA

Table 14: Resource consumption for target device Xilinx Zync XC7Z020-CLG484-1

Generic	REGISTER_WIDTH = 8	Available Resources
FF	8	53200
LUT	1	53200
I/O	11	200
BUFG	1	200

Note that a LUT is being used. This could be avoided by using active-high reset.

16.6 Netlist

Note that in Figure 43, `reset_n` is inverted (in LUT1) in order to match the interface of the FDCE. Please see Section 2.1 for more info about this.

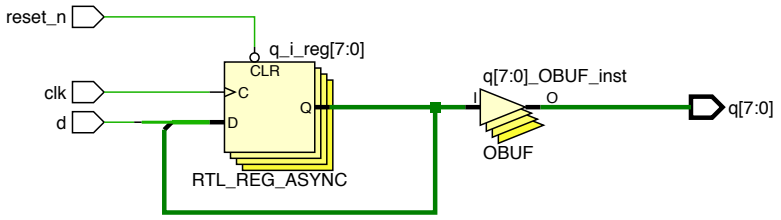


Figure 42: RTL schematic for REGISTER_WIDTH = 8

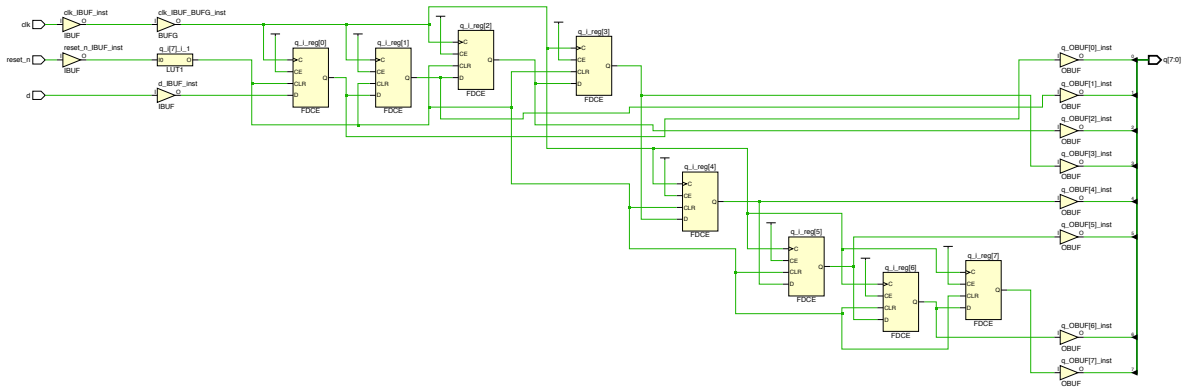


Figure 43: Synthesised schematic for REGISTER_WIDTH = 8

17 Register file

A register file is a central component in a general purpose processor. Register files consists of addressable registers. This particular register file has one write port, and two read ports.

17.1 Interface

```
entity register_file is
  generic (
    REGISTER_WIDTH : natural := 8;
    REGISTER_DEPTH : natural := 8); -- Change to Address width?
  port (
    clk        : in  std_logic;
    reset_n    : in  std_logic;
    wr_en      : in  std_logic;

    wr_addr    : in  std_logic_vector(log2c(REGISTER_DEPTH)-1 downto 0);
    d_in       : in  std_logic_vector(REGISTER_WIDTH-1 downto 0);

    rd_addr_a  : in  std_logic_vector(log2c(REGISTER_DEPTH)-1 downto 0);
    rd_addr_b  : in  std_logic_vector(log2c(REGISTER_DEPTH)-1 downto 0);
    d_out_a    : out std_logic_vector(REGISTER_WIDTH-1 downto 0);
    d_out_b    : out std_logic_vector(REGISTER_WIDTH-1 downto 0));
end register_file;
```

Here, the function `log2c` from Section 23.1 is used.

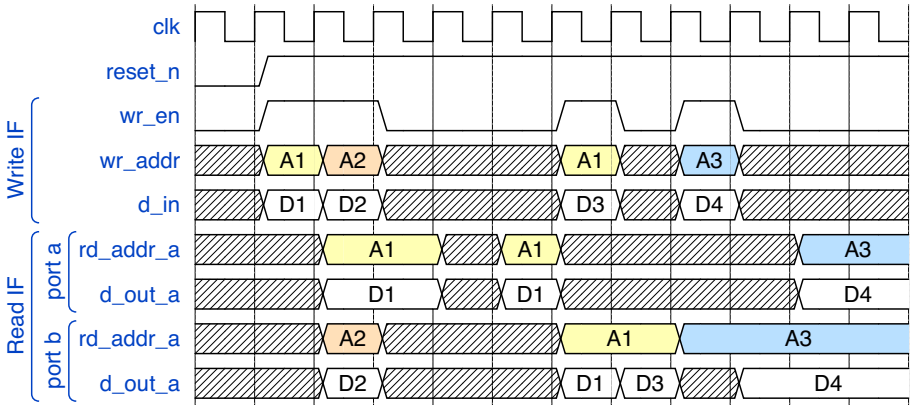


Figure 44: Example waveform for REGISTER_WIDTH = REGISTER_DEPTH = 8

17.2 Microarchitecture

...

17.3 Microarchitecture characteristics

Combinatorial	Sequential	Number of flip flops
✓	✓	REGISTER_WIDTH × REGISTER_DEPTH

17.4 Source

```
1  -- ******************************************************************************
2  -- Name:        register_file.vhd
3  -- Created:     09.03.16 @ NTNU
4  -- Author:      Jonas Eggen
5  -- Purpose:     Positive edge triggered register file with asynchronous reset,
```

```
6  --            active low.
7  -- ******************************************************************************
8
9  library ieee;
10 use ieee.std_logic_1164.all;
11 use ieee.numeric_std.all;
12 use work.math_utilities.all;
13
14 entity register_file is
15   generic (
16     REGISTER_WIDTH : natural := 8;
17     REGISTER_DEPTH : natural := 8); -- Change to Address width?
18   port (
19     clk        : in  std_logic;
20     reset_n    : in  std_logic;
21     wr_en      : in  std_logic;
22
23     wr_addr    : in  std_logic_vector(log2c(REGISTER_DEPTH)-1 downto 0);
24     d_in       : in  std_logic_vector(REGISTER_WIDTH-1 downto 0);
25
26     rd_addr_a  : in  std_logic_vector(log2c(REGISTER_DEPTH)-1 downto 0);
27     rd_addr_b  : in  std_logic_vector(log2c(REGISTER_DEPTH)-1 downto 0);
28     d_out_a    : out std_logic_vector(REGISTER_WIDTH-1 downto 0);
29     d_out_b    : out std_logic_vector(REGISTER_WIDTH-1 downto 0));
30 end register_file;
31
32 architecture rtl of register_file is
33   type reg_file_type is array (0 to REGISTER_DEPTH-1) of std_logic_vector (REGISTER_WIDTH-1
       downto 0);
34   signal reg_file : reg_file_type;
35 begin
36   d_out_a <= reg_file(to_integer(unsigned(rd_addr_a)));
37   d_out_b <= reg_file(to_integer(unsigned(rd_addr_b)));
38
39   process(clk, reset_n)
40   begin
41     if(reset_n = '0') then
42       reg_file <= (others => (others =>'0'));
43     elsif (clk'event and clk='1') then
44       if(wr_en='1') then
45         reg_file(to_integer(unsigned(wr_addr))) <= d_in;
46       end if;
47     end if;
48   end process;
49 end rtl;
```

17.5 PPA

Table 15: Resource consumption for target device Xilinx Zync XC7Z020-CLG484-1

Generic	REGISTER_WIDTH = REGISTER_DEPTH = 8	Available Resources
FF	64	53200
LUT	41	53200
I/O	36	200
BUFG	1	200

17.6 Netlist

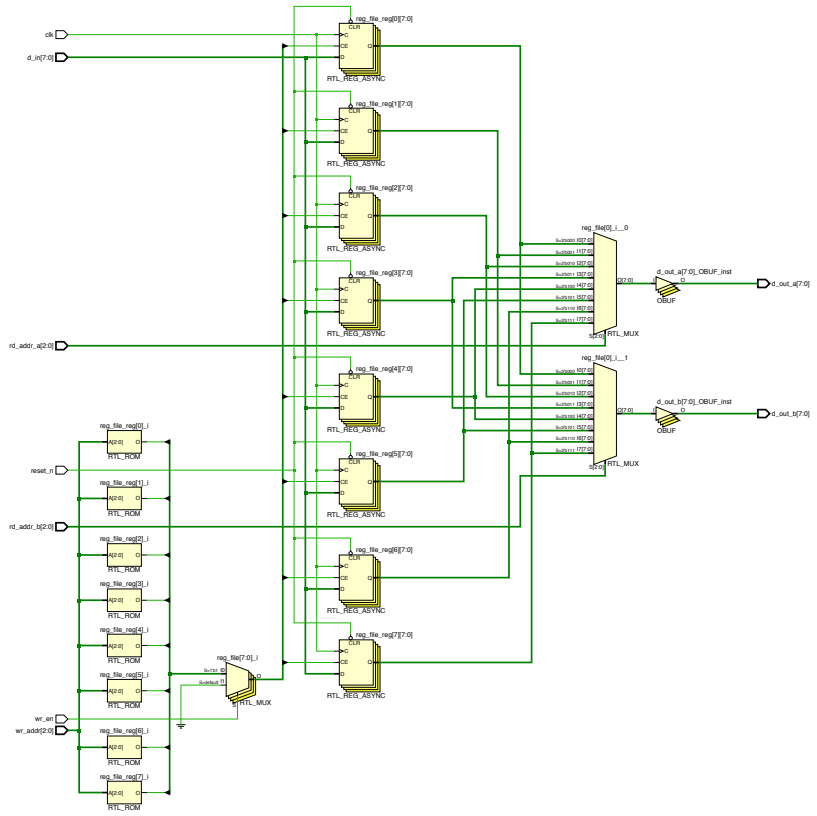


Figure 45: RTL schematic for REGISTER_WIDTH = REGISTER_DEPTH = 8

18 Combinational adder tree

Combinational adder tree

18.1 Interface

```
entity adder_tree_comb is
  port (
    numbers_a   : in  slv16_array(0 to 15);
    sum         : out std_logic_vector(19 downto 0));
end adder_tree_comb;
```

18.2 Microarchitecture

Look at the RTL code and try to create a block diagram for this design yourself.

18.3 Microarchitecture characteristics

Combinatorial	Sequential	Number of flip flops
✓		0

18.4 Source

```
1  -- ******************************************************************************
2  -- Name:       adder_tree_comb.vhd
3  -- Created:    13.03.16 @ NTNU
4  -- Author:     Jonas Eggen
5  -- Purpose:    A combinational adder tree for summarising 16 16-bit numbers.
6  -- ******************************************************************************
7
8  library ieee;
9  use ieee.std_logic_1164.all;
10 use ieee.numeric_std.all;
11 use work.arraypackage.all;
12
13 entity adder_tree_comb is
14   port (
15     numbers_a   : in  slv16_array(0 to 15);
16     sum         : out std_logic_vector(19 downto 0));
17 end adder_tree_comb;
18
19 architecture rtl_parantheses of adder_tree_comb is
20 begin
21   sum <= std_logic_vector(to_unsigned(
22                           (((to_integer(unsigned(numbers_a(0)))+
23                           to_integer(unsigned(numbers_a(1))))+
24                           (to_integer(unsigned(numbers_a(2)))+
25                           to_integer(unsigned(numbers_a(3)))))+
26                           ((to_integer(unsigned(numbers_a(4)))+
27                           to_integer(unsigned(numbers_a(5))))+
28                           (to_integer(unsigned(numbers_a(6)))+
29                           to_integer(unsigned(numbers_a(7))))))+
30                           (((to_integer(unsigned(numbers_a(8)))+
31                           to_integer(unsigned(numbers_a(9))))+
32                           (to_integer(unsigned(numbers_a(10)))+
33                           to_integer(unsigned(numbers_a(11)))))+
34                           ((to_integer(unsigned(numbers_a(12)))+
35                           to_integer(unsigned(numbers_a(13))))+
36                           (to_integer(unsigned(numbers_a(14)))+
37                           to_integer(unsigned(numbers_a(15))))))
38                           , sum'length));
39 end rtl_parantheses;
40
41 architecture rtl_verbose of adder_tree_comb is
42   signal s_0_1, s_2_3, s_4_5, s_6_7, s_8_9, s_10_11, s_12_13, s_14_15 : unsigned(16 downto 0);
43   signal s_0_1_2_3, s_4_5_6_7, s_8_9_10_11, s_12_13_14_15 : unsigned(17 downto 0);
44   signal s_0_1_2_3_4_5_6_7, s_8_9_10_11_12_13_14_15 : unsigned(18 downto 0);
45 begin
46   s_0_1 <= unsigned('0' & numbers_a(0)) + unsigned('0' & numbers_a(1));
47   s_2_3 <= unsigned('0' & numbers_a(2)) + unsigned('0' & numbers_a(3));
48   s_4_5 <= unsigned('0' & numbers_a(4)) + unsigned('0' & numbers_a(5));
```

```
49      s_6_7 <= unsigned('0' & numbers_a(6)) + unsigned('0' & numbers_a(7));
50      s_8_9 <= unsigned('0' & numbers_a(8)) + unsigned('0' & numbers_a(9));
51      s_10_11 <= unsigned('0' & numbers_a(10)) + unsigned('0' & numbers_a(11));
52      s_12_13 <= unsigned('0' & numbers_a(12)) + unsigned('0' & numbers_a(13));
53      s_14_15 <= unsigned('0' & numbers_a(14)) + unsigned('0' & numbers_a(15));
54
55      s_0_1_2_3 <= ('0' & s_0_1) + ('0' & s_2_3);
56      s_4_5_6_7 <= ('0' & s_4_5) + ('0' & s_6_7);
57      s_8_9_10_11 <= ('0' & s_8_9) + ('0' & s_10_11);
58      s_12_13_14_15 <= ('0' & s_12_13) + ('0' & s_14_15);
59
60      s_0_1_2_3_4_5_6_7 <= ('0' & s_0_1_2_3) + ('0' & s_4_5_6_7);
61      s_8_9_10_11_12_13_14_15 <= ('0' & s_8_9_10_11) + ('0' & s_12_13_14_15);
62
63    sum <= std_logic_vector(('0' & s_0_1_2_3_4_5_6_7) + ('0' & s_8_9_10_11_12_13_14_15));
64  end rtl_verbose;
```

18.5 PPA

Table 16: Resource consumption for target device Xilinx Zync XC7Z020-CLG484-1

	Synthesised	Available Resources
LUT	251	53200
I/O	276	200

18.6 Netlist

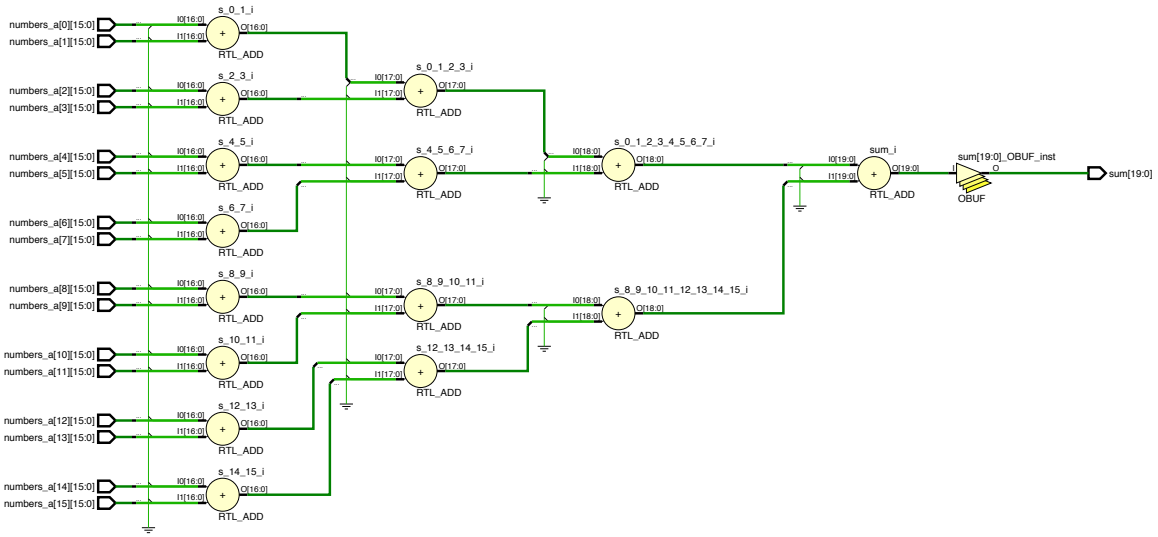


Figure 46: RTL schematic

19 Pipelined adder tree

Pipelined adder tree

19.1 Interface

```
entity adder_tree_pipelined is
  port (
    clk         : in  std_logic;
    reset_n     : in  std_logic;
    numbers_a   : in  slv16_array(0 to 15);
    sum         : out std_logic_vector(19 downto 0));
end adder_tree_pipelined;
```

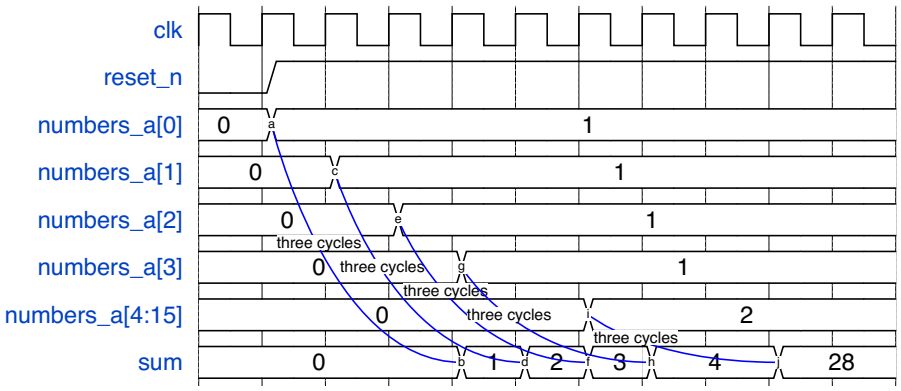


Figure 47: Example waveform

19.2 Microarchitecture

Look at the RTL code and try to create a block diagram for this design yourself.

19.3 Microarchitecture characteristics

Combinatorial	Sequential	Number of flip flops
✓		0

19.4 Source

```
1  -- ******************************************************************************
2  -- Name:       adder_tree_pipelined.vhd
3  -- Created:    13.03.16 @ NTNU
4  -- Author:     Jonas Eggen
5  -- Purpose:    A pipelined adder tree for summarising 16 16-bit numbers.
6  -- ******************************************************************************
7
8  library ieee;
9  use ieee.std_logic_1164.all;
10 use ieee.numeric_std.all;
11 use work.arraypackage.all;
12
13 entity adder_tree_pipelined is
14   port (
15     clk         : in  std_logic;
16     reset_n     : in  std_logic;
17     numbers_a   : in  slv16_array(0 to 15);
18     sum         : out std_logic_vector(19 downto 0));
19 end adder_tree_pipelined;
```

```
20
21  architecture rtl of adder_tree_pipelined is
22    signal s_0_1_nxt, s_2_3_nxt, s_4_5_nxt, s_6_7_nxt, s_8_9_nxt, s_10_11_nxt, s_12_13_nxt,
          s_14_15_nxt : unsigned(16 downto 0);
23    signal s_0_1_2_3_nxt, s_4_5_6_7_nxt, s_8_9_10_11_nxt, s_12_13_14_15_nxt : unsigned(17 downto
          0);
24    signal s_0_1_2_3_4_5_6_7_nxt, s_8_9_10_11_12_13_14_15_nxt : unsigned(18 downto 0);
25
26    signal s_0_1_r, s_2_3_r, s_4_5_r, s_6_7_r, s_8_9_r, s_10_11_r, s_12_13_r, s_14_15_r :
          unsigned(16 downto 0);
27    signal s_0_1_2_3_r, s_4_5_6_7_r, s_8_9_10_11_r, s_12_13_14_15_r : unsigned(17 downto 0);
28    signal s_0_1_2_3_4_5_6_7_r, s_8_9_10_11_12_13_14_15_r : unsigned(18 downto 0);
29  begin
30
31    process(clk, reset_n)
32    begin
33      if(reset_n = '0') then
34        s_0_1_r                    <= (others => '0');
35        s_2_3_r                    <= (others => '0');
36        s_4_5_r                    <= (others => '0');
37        s_6_7_r                    <= (others => '0');
38        s_8_9_r                    <= (others => '0');
39        s_10_11_r                  <= (others => '0');
40        s_12_13_r                  <= (others => '0');
41        s_14_15_r                  <= (others => '0');
42
43        s_0_1_2_3_r                <= (others => '0');
44        s_4_5_6_7_r                <= (others => '0');
45        s_8_9_10_11_r              <= (others => '0');
46        s_12_13_14_15_r            <= (others => '0');
47
48
49        s_0_1_2_3_4_5_6_7_r        <= (others => '0');
50        s_8_9_10_11_12_13_14_15_r <= (others => '0');
51      elsif (clk'event and clk='1') then
52        s_0_1_r                    <= s_0_1_nxt;
53        s_2_3_r                    <= s_2_3_nxt;
54        s_4_5_r                    <= s_4_5_nxt;
55        s_6_7_r                    <= s_6_7_nxt;
56        s_8_9_r                    <= s_8_9_nxt;
57        s_10_11_r                  <= s_10_11_nxt;
58        s_12_13_r                  <= s_12_13_nxt;
59        s_14_15_r                  <= s_14_15_nxt;
60
61        s_0_1_2_3_r                <= s_0_1_2_3_nxt;
62        s_4_5_6_7_r                <= s_4_5_6_7_nxt;
63        s_8_9_10_11_r              <= s_8_9_10_11_nxt;
64        s_12_13_14_15_r            <= s_12_13_14_15_nxt;
65
66
67        s_0_1_2_3_4_5_6_7_r        <= s_0_1_2_3_4_5_6_7_nxt;
68        s_8_9_10_11_12_13_14_15_r <= s_8_9_10_11_12_13_14_15_nxt;
69      end if;
70    end process;
71
72    s_0_1_nxt   <= unsigned('0' & numbers_a(0)) + unsigned('0' & numbers_a(1));
73    s_2_3_nxt   <= unsigned('0' & numbers_a(2)) + unsigned('0' & numbers_a(3));
74    s_4_5_nxt   <= unsigned('0' & numbers_a(4)) + unsigned('0' & numbers_a(5));
75    s_6_7_nxt   <= unsigned('0' & numbers_a(6)) + unsigned('0' & numbers_a(7));
76    s_8_9_nxt   <= unsigned('0' & numbers_a(8)) + unsigned('0' & numbers_a(9));
77    s_10_11_nxt <= unsigned('0' & numbers_a(10)) + unsigned('0' & numbers_a(11));
78    s_12_13_nxt <= unsigned('0' & numbers_a(12)) + unsigned('0' & numbers_a(13));
79    s_14_15_nxt <= unsigned('0' & numbers_a(14)) + unsigned('0' & numbers_a(15));
80
81    s_0_1_2_3_nxt     <= ('0' & s_0_1_r) + ('0' & s_2_3_r);
82    s_4_5_6_7_nxt     <= ('0' & s_4_5_r) + ('0' & s_6_7_r);
83    s_8_9_10_11_nxt   <= ('0' & s_8_9_r) + ('0' & s_10_11_r);
84    s_12_13_14_15_nxt <= ('0' & s_12_13_r) + ('0' & s_14_15_r);
85
86    s_0_1_2_3_4_5_6_7_nxt       <= ('0' & s_0_1_2_3_r) + ('0' & s_4_5_6_7_r);
87    s_8_9_10_11_12_13_14_15_nxt <= ('0' & s_8_9_10_11_r) + ('0' & s_12_13_14_15_r);
88
89    sum <= std_logic_vector(('0' & s_0_1_2_3_4_5_6_7_r) + ('0' & s_8_9_10_11_12_13_14_15_r));
90  end rtl;
```

19.5 PPA

Table 17: Resource consumption for target device Xilinx Zync XC7Z020-CLG484-1

	Synthesised	Available Resources
f_{max}	403 MHz	
FF	246	106400
LUT	254	53200
I/O	278	200
BUFG	1	32

19.6 Netlist

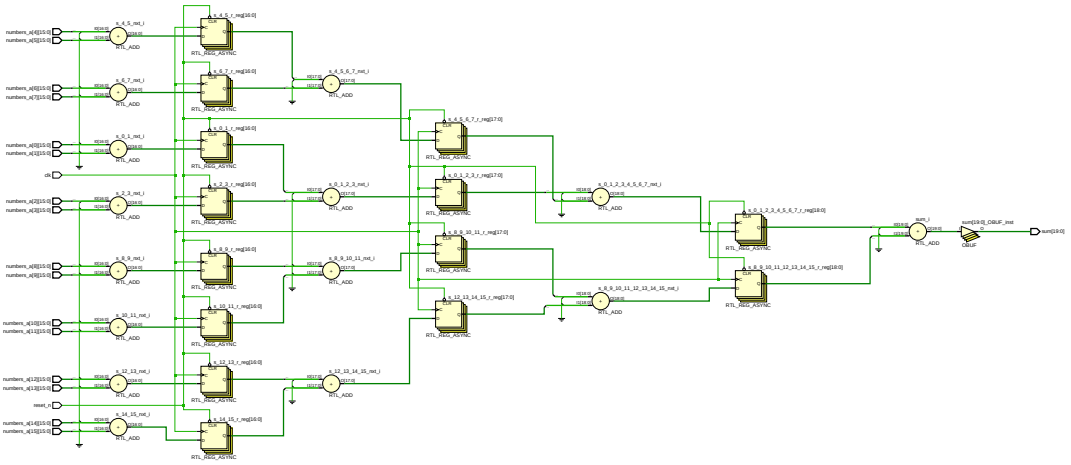


Figure 48: RTL schematic

20 FIFO

FIFO

20.1 Interface

```
entity fifo is
  generic (
    FIFO_DEPTH : natural := 8;
    FIFO_WIDTH : natural := 8);
  port (
    clk      : in  std_logic;
    reset_n  : in  std_logic;
    rd       : in  std_logic;
    wr       : in  std_logic;
    d_in     : in  std_logic_vector(FIFO_WIDTH-1 downto 0);
    full     : out std_logic;
    empty    : out std_logic;
    d_out    : out std_logic_vector(FIFO_WIDTH-1 downto 0));
end fifo;
```

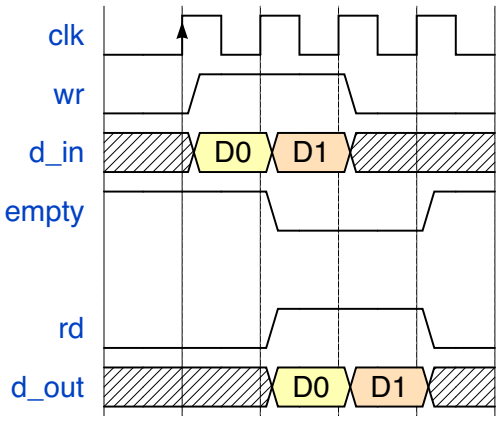


Figure 49: Specification

By looking at the specification in Figure 49, some conclusions can be made:

An asynchronous read operation is needed
In the waveform, `d_out` arrives at the same time as `rd`. This means that the read operation is not clocked, and thus needs to be asynchronous.

The delay is very short
As seen in the waveform, D0 only uses one clock cycle to get through the FIFO. This essentially means that the FIFO needs to operate like a D flip-flop at times when a read directly follows a write to an empty FIFO.

Reading and writing a full FIFO will conflict with the first point
When this FIFO is full, one cannot write even if reading at the same time, as this will be in conflict with the previous point. NOTE: this should be elaborated. Something about the fact that a full and empty FIFO really is the same thing (`rd_addr` equal to `wr_addr`), the only distinction being which pointer caught up with the other.

20.2 Microarchitecture

20.3 Microarchitecture characteristics

20.4 Source

Combinatorial	Sequential	Number of flip flops
✓	✓	Meeeh

```
1  -- ******************************************************************************
2  -- Name:       fifo.vhd
3  -- Created:    02.03.16 @ NTNU
4  -- Author:     Jonas Eggen
5  -- Purpose:    FIFO.
6  -- ******************************************************************************
7
8  library ieee;
9  use ieee.std_logic_1164.all;
10 use ieee.numeric_std.all;
11
12 entity fifo is
13   generic (
14     FIFO_DEPTH : natural := 8;
15     FIFO_WIDTH : natural := 8);
16   port (
17     clk     : in  std_logic;
18     reset_n : in  std_logic;
19     rd      : in  std_logic;
20     wr      : in  std_logic;
21     d_in    : in  std_logic_vector(FIFO_WIDTH-1 downto 0);
22     full    : out std_logic;
23     empty   : out std_logic;
24     d_out   : out std_logic_vector(FIFO_WIDTH-1 downto 0));
25 end fifo;
26
27 -- Using asynchronous read, and thus distributed ram.
28 architecture rtl_lut_ram of fifo is
29   type ram_type is array (0 to FIFO_DEPTH-1) of std_logic_vector (FIFO_WIDTH-1 downto 0);
30   signal RAM          : ram_type;
31   signal wr_addr_r    : integer range 0 to FIFO_DEPTH-1;
32   signal wr_addr_succ: integer range 0 to FIFO_DEPTH-1;
33   signal wr_addr_nxt : integer range 0 to FIFO_DEPTH-1;
34   signal rd_addr_r    : integer range 0 to FIFO_DEPTH-1;
35   signal rd_addr_succ: integer range 0 to FIFO_DEPTH-1;
36   signal rd_addr_nxt : integer range 0 to FIFO_DEPTH-1;
37   signal full_r       : std_logic;
38   signal full_nxt     : std_logic;
39   signal empty_r      : std_logic;
40   signal empty_nxt    : std_logic;
41   signal operation    : std_logic_vector(1 downto 0);
42
43 begin
44   wr_addr_succ <= (wr_addr_r + 1) mod FIFO_DEPTH; -- The mod will be optimized away if
         FIFO_DEPTH is a power of 2.
45   rd_addr_succ <= (rd_addr_r + 1) mod FIFO_DEPTH;
46   full  <= full_r;
47   empty <= empty_r;
48
49   process(clk, reset_n)
50   begin
51     if (reset_n = '0') then
52       full_r <= '0';
53       empty_r <= '1';
54       wr_addr_r <= 0;
55       rd_addr_r <= 0;
56     elsif (clk'event and clk='1') then
57       -- A clock enable here could reduce energy consumption. At least in ASIC.
58       full_r <= full_nxt;
59       empty_r <= empty_nxt;
60       wr_addr_r <= wr_addr_nxt;
61       rd_addr_r <= rd_addr_nxt;
62     end if;
63   end process;
64
65   -- Next state and address logic
66   operation <= wr & rd;
67   process(operation, full_r, empty_r, wr_addr_r, rd_addr_r, wr_addr_succ, rd_addr_succ)
68   begin
69     full_nxt    <= full_r;
70     empty_nxt   <= empty_r;
71     wr_addr_nxt <= wr_addr_r;
72     rd_addr_nxt <= rd_addr_r;
73     case operation is
74       when "00" => -- nop
75       when "10" => -- write
```

```
76          if (full_r='0') then
77            wr_addr_nxt <= wr_addr_succ;
78            empty_nxt   <= '0';
79            if (wr_addr_succ = rd_addr_r) then
80              full_nxt <= '1';
81            end if;
82          end if;
83        when "01" => -- read
84          if (empty_r='0') then
85            rd_addr_nxt <= rd_addr_succ;
86            full_nxt   <= '0';
87            if (rd_addr_succ = wr_addr_r) then
88              empty_nxt <= '1';
89            end if;
90          end if;
91        when others => -- read and write
92          wr_addr_nxt <= wr_addr_succ;
93          rd_addr_nxt <= rd_addr_succ;
94      end case;
95    end process;
96
97    d_out <= RAM(rd_addr_r); -- This asynchronous read forces inference of distributed RAM
          instead of BRAM.
98
99    process(clk)
100   begin
101     if (clk'event and clk='1') then
102       if (wr='1' and (full_r='0' or rd='1')) then
103         RAM(wr_addr_r) <= d_in;
104       end if;
105     end if;
106   end process;
107 end rtl_lut_ram;
108
109 -- Avoids using asynchronous read, and can therefore be implemented using block ram.
110 architecture rtl_block_ram of fifo is
111   type ram_type is array (0 to FIFO_DEPTH-1) of std_logic_vector (FIFO_WIDTH-1 downto 0);
112   signal RAM          : ram_type;
113   signal wr_addr_r    : integer range 0 to FIFO_DEPTH-1;
114   signal wr_addr_succ: integer range 0 to FIFO_DEPTH-1;
115   signal wr_addr_nxt : integer range 0 to FIFO_DEPTH-1;
116   signal rd_addr_r    : integer range 0 to FIFO_DEPTH-1;
117   signal rd_addr_succ: integer range 0 to FIFO_DEPTH-1;
118   signal rd_addr_nxt : integer range 0 to FIFO_DEPTH-1;
119   signal full_r       : std_logic;
120   signal full_nxt     : std_logic;
121   signal empty_r      : std_logic;
122   signal empty_nxt    : std_logic;
123   signal operation    : std_logic_vector(1 downto 0);
124
125   signal ram_out      : std_logic_vector(FIFO_WIDTH-1 downto 0);
126   signal d_in_d1      : std_logic_vector(FIFO_WIDTH-1 downto 0);
127   signal wr_d1        : std_logic;
128
129 begin
130   wr_addr_succ <= (wr_addr_r + 1) mod FIFO_DEPTH; -- The mod will be optimized away if
          FIFO_DEPTH is a power of 2.
131   rd_addr_succ <= (rd_addr_r + 1) mod FIFO_DEPTH;
132   full  <= full_r;
133   empty <= empty_r;
134
135   process(clk, reset_n)
136   begin
137     if (reset_n = '0') then
138       full_r <= '0';
139       empty_r <= '1';
140       wr_addr_r <= 0;
141       rd_addr_r <= 0;
142       wr_d1 <= '0';
143       d_in_d1 <= (others => '0');
144     elsif (clk'event and clk='1') then
145       full_r <= full_nxt;
146       empty_r <= empty_nxt;
147       wr_addr_r <= wr_addr_nxt;
148       rd_addr_r <= rd_addr_nxt;
149       wr_d1 <= wr;
150       d_in_d1 <= d_in;
151     end if;
152   end process;
153
154   operation <= wr & rd;
155   process(operation, full_r, empty_r, wr_addr_r, rd_addr_r, wr_addr_succ, rd_addr_succ)
156   begin
```

```
157       full_nxt    <= full_r;
158       empty_nxt   <= empty_r;
159       wr_addr_nxt <= wr_addr_r;
160       rd_addr_nxt <= rd_addr_r;
161       case operation is
162         when "00" => -- nop
163         when "10" => -- write
164           if (full_r='0') then
165             wr_addr_nxt <= wr_addr_succ;
166             empty_nxt   <= '0';
167             if (wr_addr_succ = rd_addr_r) then
168               full_nxt <= '1';
169             end if;
170           end if;
171         when "01" => -- read
172           if (empty_r='0') then
173             rd_addr_nxt <= rd_addr_succ;
174             full_nxt    <= '0';
175             if (rd_addr_succ = wr_addr_r) then
176               empty_nxt <= '1';
177             end if;
178           end if;
179         when others => -- read and write
180           wr_addr_nxt <= wr_addr_succ;
181           rd_addr_nxt <= rd_addr_succ;
182       end case;
183     end process;
184
185     -- If the fifo is full, and we wrote last clock cycle, use delayed version of d_in instead
          of BRAM output.
186     -- This allows for inference of BRAM while staying in spec.
187     d_out <= d_in_d1 when (rd_addr_succ = wr_addr_r) and wr_d1='1' else ram_out;
188     process(clk)
189     begin
190       if (clk'event and clk='1') then
191         if (wr='1' and (full_r='0' or rd='1')) then
192           RAM(wr_addr_r) <= d_in;
193         end if;
194         ram_out <= RAM(rd_addr_nxt);
195       end if;
196     end process;
197   end rtl_block_ram;
198
199   -- Avoids using asynchronous read, and can therefore be implemented using block ram. Avoid
        using separate register for d_in_d1.
200   architecture rtl_block_ram_no_d1 of fifo is
201     type ram_type is array (0 to FIFO_DEPTH-1) of std_logic_vector (FIFO_WIDTH-1 downto 0);
202     signal RAM          : ram_type;
203     signal wr_addr_r    : integer range 0 to FIFO_DEPTH-1;
204     signal wr_addr_succ: integer range 0 to FIFO_DEPTH-1;
205     signal wr_addr_nxt : integer range 0 to FIFO_DEPTH-1;
206     signal rd_addr_r    : integer range 0 to FIFO_DEPTH-1;
207     signal rd_addr_succ: integer range 0 to FIFO_DEPTH-1;
208     signal rd_addr_nxt : integer range 0 to FIFO_DEPTH-1;
209     signal full_r       : std_logic;
210     signal full_nxt     : std_logic;
211     signal empty_r      : std_logic;
212     signal empty_nxt    : std_logic;
213     signal operation    : std_logic_vector(1 downto 0);
214
215     signal ram_out      : std_logic_vector(FIFO_WIDTH-1 downto 0);
216     signal written_data: std_logic_vector(FIFO_WIDTH-1 downto 0);
217     signal wr_d1        : std_logic;
218
219   begin
220     wr_addr_succ <= (wr_addr_r + 1) mod FIFO_DEPTH; -- The mod will be optimized away if
          FIFO_DEPTH is a power of 2.
221     rd_addr_succ <= (rd_addr_r + 1) mod FIFO_DEPTH;
222     full  <= full_r;
223     empty <= empty_r;
224
225     process(clk, reset_n)
226     begin
227       if (reset_n = '0') then
228         full_r <= '0';
229         empty_r <= '1';
230         wr_addr_r <= 0;
231         rd_addr_r <= 0;
232         wr_d1 <= '0';
233       elsif (clk'event and clk='1') then
234         full_r <= full_nxt;
235         empty_r <= empty_nxt;
236         wr_addr_r <= wr_addr_nxt;
```

```
237         rd_addr_r <= rd_addr_nxt;
238         wr_d1 <= wr;
239       end if;
240     end process;
241
242     operation <= wr & rd;
243     process(operation, full_r, empty_r, wr_addr_r, rd_addr_r, wr_addr_succ, rd_addr_succ)
244     begin
245       full_nxt    <= full_r;
246       empty_nxt   <= empty_r;
247       wr_addr_nxt <= wr_addr_r;
248       rd_addr_nxt <= rd_addr_r;
249       case operation is
250         when "00" => -- nop
251         when "10" => -- write
252           if (full_r='0') then
253             wr_addr_nxt <= wr_addr_succ;
254             empty_nxt   <= '0';
255             if (wr_addr_succ = rd_addr_r) then
256               full_nxt <= '1';
257             end if;
258           end if;
259         when "01" => -- read
260           if (empty_r='0') then
261             rd_addr_nxt <= rd_addr_succ;
262             full_nxt    <= '0';
263             if (rd_addr_succ = wr_addr_r) then
264               empty_nxt <= '1';
265             end if;
266           end if;
267         when others => -- read and write
268           wr_addr_nxt <= wr_addr_succ;
269           rd_addr_nxt <= rd_addr_succ;
270       end case;
271     end process;
272
273     -- If the fifo is full, and we wrote last clock cycle, use delayed version of d_in instead
          of BRAM output.
274     -- This allows for inference of BRAM while staying in spec.
275     d_out <= written_data when (rd_addr_succ = wr_addr_r) and wr_d1='1' else ram_out;
276     process(clk)
277     begin
278       if (clk'event and clk='1') then
279         if (wr='1' and (full_r='0' or rd='1')) then
280           RAM(wr_addr_r) <= d_in;
281           written_data <= d_in;
282         else
283           written_data <= RAM(wr_addr_r);
284         end if;
285         ram_out <= RAM(rd_addr_nxt);
286       end if;
287     end process;
288   end rtl_block_ram_no_d1;
```

20.5 PPA

Table 18: Resource consumption for target device Xilinx Zync XC7Z020-CLG484-1

Generic	FIFO_WIDTH = FIFO_DEPTH = 8	Available Resources
FF	x	53200
LUT	x	53200
I/O	x	200
BUFG	x	200

20.6 Netlist

Figure 50: RTL schematic

21 Combinational bit scanner

The output of a bit scanner is a bit vector of the same width as the input. Given that LSB has the highest priority, a bit scanner scans from LSB towards MSB until it finds the first one. A one is then set in the output at the same position.

21.1 Interface

```
entity bit_scanner is
  generic(
    N         : natural := 4);
  port (
    req       : in  std_logic_vector(N-1 downto 0);
    gnt       : out std_logic_vector(N-1 downto 0));
end bit_scanner;
```

21.2 Microarchitecture

Two implementations will be explored here. One that works according to Figure 51. Here, the request is ANDed with its two's complement. This will later be implemented as `rtl_signed` and `rtl_twos_comp`.

$$\begin{aligned} \texttt{req:} &\ 110101001100 \\ \overline{\texttt{req}}\texttt{:} &\ 001010110011 \\ \overline{\texttt{req}}\texttt{+1:} &\ 001010110100 \\ \texttt{req \& (}\overline{\texttt{req}}\texttt{+1):} &\ 000000000100 \end{aligned}$$

Figure 51: Step-by-step walkthrough of two's complement method for N = 12

The other implementation is very similar. Here, the request is ANDed with the complement of the request minus 1. This will be implemented as `rtl_sub`. See Figure 52.

$$\begin{aligned} \texttt{req:} &\ 110101001100 \\ \texttt{req-1:} &\ 110101001011 \\ \overline{\texttt{req-1}}\texttt{:} &\ 001010110100 \\ \texttt{req \& (}\overline{\texttt{req-1}}\texttt{):} &\ 000000000100 \end{aligned}$$

Figure 52: Step-by-step walkthrough of subtraction method for N = 12

21.3 Microarchitecture characteristics

Combinatorial	Sequential	Number of flip flops
✓		0

21.4 Source

```
1  -- ******************************************************************************
2  -- Name:       bit_scanner.vhd
3  -- Created:    20.02.16 @ NTNU
4  -- Author:     Jonas Eggen
5  -- Purpose:    A combinational bit scanner. LSB has highest priority.
6  -- ******************************************************************************
7
8  library ieee;
```

```
 9  use ieee.std_logic_1164.all;
10  use ieee.numeric_std.all;
11
12  entity bit_scanner is
13    generic(
14      N         : natural := 4);
15    port (
16      req       : in  std_logic_vector(N-1 downto 0);
17      gnt       : out std_logic_vector(N-1 downto 0));
18  end bit_scanner;
19
20  architecture rtl_signed of bit_scanner is
21  begin
22    gnt <= std_logic_vector(signed(req) and -signed(req));
23  end rtl_signed;
24
25  architecture rtl_twos_comp of bit_scanner is
26  begin
27    gnt <= std_logic_vector(unsigned(req) and ((not unsigned(req)) + 1));
28  end rtl_twos_comp;
29
30  architecture rtl_sub of bit_scanner is
31  begin
32    gnt <= std_logic_vector(unsigned(req) and not (unsigned(req) - 1));
33  end rtl_sub;
```

21.5 PPA

Table 19: Number of LUTs for target device Xilinx Zync XC7Z020-CLG484-1

Architecture	N = 4	N = 64
rtl_signed	2	128
rtl_twos_comp	2	128
rtl_sub	2	126

21.6 Netlist

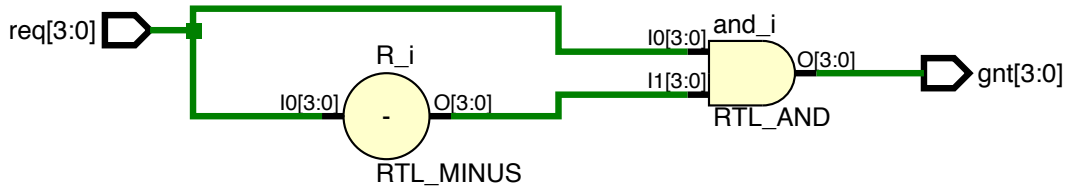


Figure 53: rtl_signed schematic for N = 4

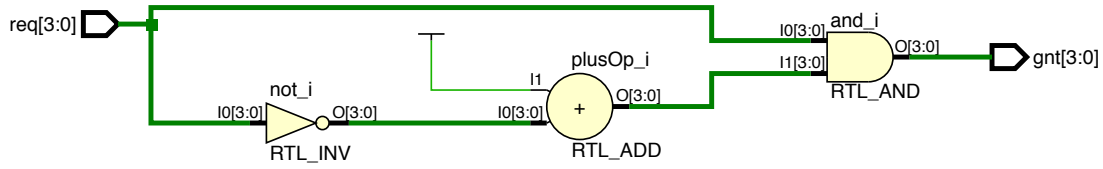


Figure 54: rtl_twos_comp schematic for N = 4

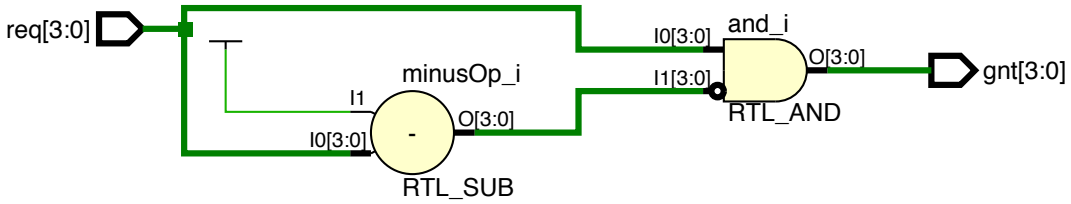


Figure 55: `rtl_sub` schematic for N = 4

22 Round-robin bit scanner

While the combinational bit scanner from Section 21 has a fixed priority, the round-robin bit scanner puts a bit of fairness into the bit scanner. Given that all request lines are high for several clock cycles, grants will be given in a round-robin fashion. The waveform in Figure 56 should give an idea of how it works.

22.1 Interface

```
entity rr_bit_scanner is
  generic(
    N          : natural := 4);
  port (
    clk        : in std_logic;
    reset_n    : in std_logic;
    req        : in  std_logic_vector(N-1 downto 0);
    gnt        : out std_logic_vector(N-1 downto 0));
end rr_bit_scanner;
```

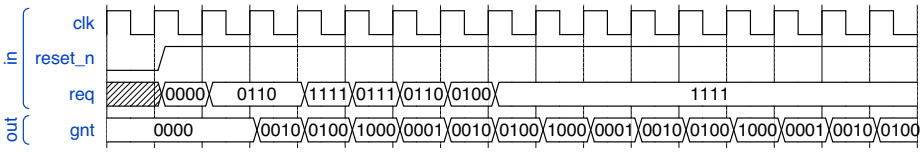


Figure 56: Example waveform for N = 4

22.2 Microarchitecture characteristics

Combinatorial	Sequential	Number of flip flops
✓	✓	N

22.3 Source

```
1  -- ********************************************************************************
2  -- Name:       rr_bit_scanner.vhd
3  -- Created:    20.02.16 @ NTNU
4  -- Author:     Jonas Eggen
5  -- Purpose:    A combinational bit scanner. LSB has highest priority.
6  -- ********************************************************************************
7
8  library ieee;
9  use ieee.std_logic_1164.all;
10 use ieee.numeric_std.all;
11 use ieee.std_logic_misc.all;
12
13 entity rr_bit_scanner is
14   generic(
15     N          : natural := 4);
16   port (
17     clk        : in std_logic;
18     reset_n    : in std_logic;
19     req        : in  std_logic_vector(N-1 downto 0);
20     gnt        : out std_logic_vector(N-1 downto 0));
21 end rr_bit_scanner;
22
23 architecture rtl_mux of rr_bit_scanner is
24   signal gnt_r        : std_logic_vector(N-1 downto 0);
25   signal gnt_nxt      : std_logic_vector(N-1 downto 0);
26
27   signal gnt_umasked  : std_logic_vector(N-1 downto 0);
28   signal mask         : std_logic_vector(N-1 downto 0);
29   signal req_masked   : std_logic_vector(N-1 downto 0);
30   signal gnt_masked   : std_logic_vector(N-1 downto 0);
```

```
31  begin
32    process(clk, reset_n)
33    begin
34      if(reset_n = '0') then
35        gnt_r <= (others => '0');
36      elsif(clk'event and clk='1') then
37        gnt_r <= gnt_nxt;
38      end if;
39    end process;
40
41    gnt <= gnt_r;
42
43    -- Regular bit scanner
44    umasked: entity work.bit_scanner(rtl_signed)
45      generic map (
46        N    => N)
47      port map (
48        req => req,
49        gnt => gnt_umasked);
50
51    -- Mask generation
52    mask(0) <= '0';
53    mask(1) <= gnt_r(0);
54    mask_gen: for i in 2 to N-1 generate
55      mask(i) <= mask(i-1) or gnt_r(i-1);
56    end generate;
57
58    -- Masked version of request
59    req_masked <= req and mask;
60
61    -- Regular bit scanner working on masked request
62    masked: entity work.bit_scanner(rtl_signed)
63      generic map (
64        N    => N)
65      port map (
66        req => req_masked,
67        gnt => gnt_masked);
68
69    -- Choose masked version if any grants given there
70    gnt_nxt <= gnt_masked when or_reduce(req_masked) = '1' else gnt_umasked;
71  end rtl_mux;
72
73  architecture rtl_no_mux of rr_bit_scanner is
74    signal gnt_r        : std_logic_vector(N-1 downto 0);
75    signal gnt_nxt      : std_logic_vector(N-1 downto 0);
76
77    signal mask         : std_logic_vector(N-1 downto 0);
78    signal req_masked   : std_logic_vector(2*N-1 downto 0);
79    signal gnt_masked   : std_logic_vector(2*N-1 downto 0);
80  begin
81    process(clk, reset_n)
82    begin
83      if(reset_n = '0') then
84        gnt_r <= (others => '0');
85      elsif(clk'event and clk='1') then
86        gnt_r <= gnt_nxt;
87      end if;
88    end process;
89
90    gnt <= gnt_r;
91
92    -- Mask generation
93    mask(0) <= '0';
94    mask(1) <= gnt_r(0);
95    mask_gen: for i in 2 to N-1 generate
96      mask(i) <= mask(i-1) or gnt_r(i-1);
97    end generate;
98
99    -- Request concatenated with masked version of request
100   req_masked <= req & (req and mask);
101
102   -- Regular bit scanner working on masked request
103   masked: entity work.bit_scanner(rtl_signed)
104     generic map (
105       N    => 2*N)
106     port map (
107       req => req_masked,
108       gnt => gnt_masked);
109
110   -- Oring upper(unmasked) and lower(masked) part of grant to get n bit wide gnt.
111   gnt_nxt <= gnt_masked(2*N-1 downto N) or gnt_masked(N-1 downto 0);
112 end rtl_no_mux;
```

22.4 PPA

Table 20: Resource consumption for target device Xilinx Zync XC7Z020-CLG484-1 for N = 4

RR architecture	rtl_mux		rtl_no_mux	
BS architecture	rtl_sub	rtl_signed	rtl_sub	rtl_signed
f_{max}	384 MHz	421 MHz	420 MHz	421 MHz
FF	4	4	4	4
LUT	7	7	7	7

Table 21: Resource consumption for target device Xilinx Zync XC7Z020-CLG484-1 for N = 64

RR architecture	rtl_mux		rtl_no_mux	
BS architecture	rtl_sub	rtl_signed	rtl_sub	rtl_signed
f_{max}	124 MHz	122 MHz	108 MHz	108 MHz
FF	64	64	64	64
LUT	378	342	269	248

From Table 21 one can see that depending on what the goal of the design is, different architectures has to be chosen. If speed is important, rtl_mux with rtl_sub should be chosen. If area is key, rtl_no_mux with rtl_signed would be preferable.

22.5 Netlist

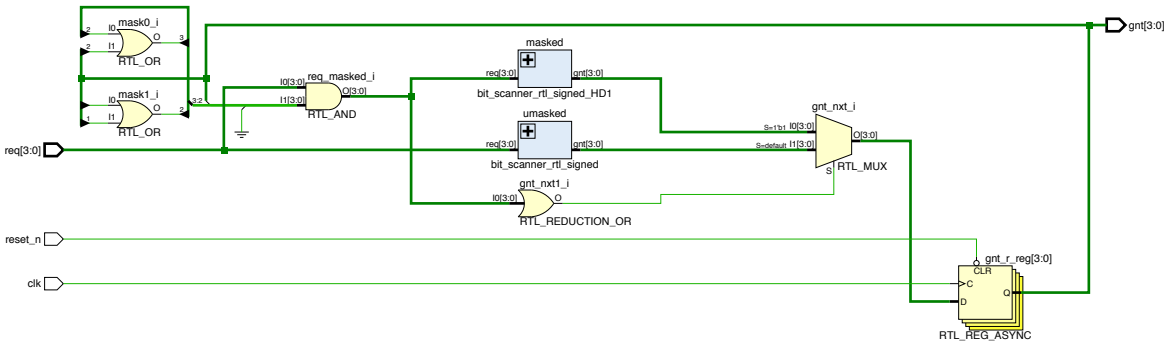


Figure 57: rtl_mux schematic for N = 4

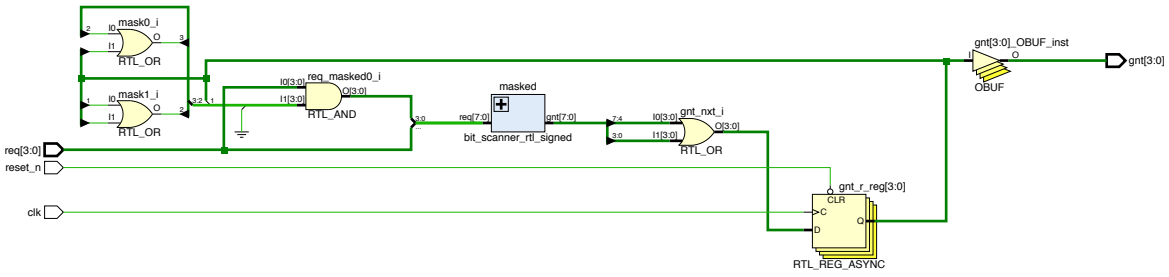


Figure 58: rtl_no_mux schematic for N = 4

23 Packages

23.1 math_utilities.vhd

This is a utility package with useful math functions.

log2c

```
function log2c(constant value : in positive) return natural
```

This function calculates $\lceil \log_2 n \rceil$, where $n =$ `value`, i.e. the binary logarithm of `value`, and returns the result after rounding it towards ∞. This is useful when calculating the required address width from the number of addressable entities.

23.1.1 Source

```
1  -- ******************************************************************************
2  -- Name:       math_utilities.vhd
3  -- Created:    10.03.16 @ NTNU
4  -- Author:     Jonas Eggen
5  -- Purpose:    Utility package with useful math functions.
6  --              - log2c returns ceil(log2(x))
7  -- ******************************************************************************
8
9  library ieee;
10 use ieee.math_real.all;
11
12 package math_utilities is
13   function log2c(constant value : in positive) return natural; -- Try using natural here..
14 end math_utilities;
15
16 package body math_utilities is
17   function log2c(constant value : in positive) return natural is
18   begin
19       return integer(ceil(log2(real(value))));
20   end function;
21 end package body math_utilities;
```

23.2 arraypackage.vhd

This is a utility package with definitions of arrays that can be used as port types. Ideally, this package would only have one type: `type slv_array is array (natural range <>) of std_logic_vector;`. This would allow for both the array and vector length to be unconstrained. However, this is VHDL 2008, and is not widely supported.

slv16_array

```
type slv16_array is array (natural range <>) of std_logic_vector(15 downto 0);
```

This type is an unconstrained array of `std_logic_vector(15 downto 0)`.

23.2.1 Source

```
1  -- ******************************************************************************
2  -- Name:       arraypackage.vhd
3  -- Created:    13.03.16 @ NTNU
4  -- Author:     Jonas Eggen
5  -- Purpose:    Package to allow array as an input type.
6  -- ******************************************************************************
7
8  library ieee;
9  use ieee.std_logic_1164.all;
10
11 package arraypackage is
12   type slv16_array is array (natural range <>) of std_logic_vector(15 downto 0);
13 end arraypackage;
14
15 package body arraypackage is
16 end package body arraypackage;
```