University Bremerhaven

Course Documentation

Digital Systems / VHDL
[ET–DTV]

- Part 1: Digital Systems Fundamentals
- Part 2: Design Principles
- Part 3: Simulation of Digital Sytems
- Part 4: Design of Digital Systems Using VHDL
- Part 5: Practical Lab Experiments

Revision: V1.2 (minor)

Release: March 2020

Prof. Dr.-Ing. Kai Mueller
University of Applied Sciences Bremerhaven
Institute for Automation and Electrical Engineering
An der Karlstadt 8

D–27568 Bremerhaven

Phone: +49 471 48 23 – 415
FAX: +49 471 48 23 – 555

Email: kmueller@hs–bremerhaven.de

I Introduction

I.I Course Documentation

See < http://www1.hs – bremerhaven.de/kmueller/> for updates.

I.II Digital Systems / VHDL

In billions of devices digital systems are almost everywhere. They have become part of many consumer products, industrial and scientific devices. Since the prices for digital hardware dropped dramatically over the years todays engineers can create almost everything that can be imagined.

This course teaches to design digital systems using VHDL.

Bremerhaven, March 2011

Kai Müller
<kmueller@hs – bremerhaven.de>
Tel: (0471) 4823 – 415

II Contents

1 Digital Systems Fundamentals

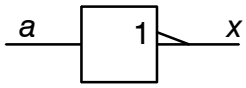
1.1 Combinational Logic

In digital systems physical states are reduced to several discrete values (in VHDL `std_logic` type we have 9 different values). The boolean algebra knows only two values: '0' and '1'.

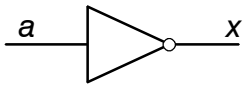
Boolean algebra, thus, can be used to describe the functional operation of a system.

1.1.1 Unary Operators

symbol



intern. symbol



truth table

a	x
0	1
1	0

boolean function

german (DIN):	$x = \overline{y}$	($x =$ nicht a)
intern. form:	$x = a'$	($x =$ not a)

1.1.2 Binary Operators

inputs	
$a =$	1 0 1 0
$b =$	1 1 0 0

outputs x	boolean symbol	notation	elementary logic
0 0 0 0	$x = 0$	constant 0	
0 0 0 1	$x = (a + b)'$	NOR	•
0 0 1 0	$x = a \cdot b'$	inhibit	
0 0 1 1	$x = b'$	negate (b)	
0 1 0 0	$x = a' \cdot b$	Inhibit	
0 1 0 1	$x = a'$	negate (a)	
0 1 1 0	$x = a \oplus b$	XOR	•
0 1 1 1	$x = (a \cdot b)'$	NAND	•
1 0 0 0	$x = a \cdot b$	AND	•
1 0 0 1	$x = a \equiv b$	equivalence	
1 0 1 0	$x = a$	identity (a)	
1 0 1 1	$x = a + b'$	implication	
1 1 0 0	$x = b$	identity (b)	
1 1 0 1	$x = a' + b$	implication	
1 1 1 0	$x = a + b$	OR	•
1 1 1 1	$x = 1$	constant 1	

1.1.3 Boolean Theorems for One Variable and Constants

The proof is straight forward by using the truth table.

$$x + 0 = x \tag{2.1}$$

$$x + 1 = 1 \tag{2.2}$$

$$x \cdot 0 = 0 \tag{2.3}$$

$$x \cdot 1 = x \tag{2.4}$$

$$x + x' = 1 \tag{2.5}$$

$$x \cdot x' = 0 \tag{2.6}$$

$$x + x = x \tag{2.7}$$

$$x \cdot x = x \tag{2.8}$$

$$x'' = x \tag{2.9}$$

1.1.4 Boolean Theorems for Several Variables

The proof is also possible by using the truth table since the number of rows to check is limited. Also 1.1.3 can proof some results.

Commutativity

$$x + y = y + x\,, \quad x \cdot y = y \cdot x\,. \tag{2.10}$$

Associativity

$$(x + y) + z = x + (y + z)\,, \quad (x \cdot y) \cdot z = x \cdot (y \cdot z)\,. \tag{2.11}$$

Distributivity

$$x \cdot y + x \cdot z = x \cdot (y + z)\,, \quad (x + y) \cdot (x + z) = x + y \cdot z\,. \tag{2.12}$$

Covering

$$x + x \cdot y = x\,, \quad x \cdot (x + y) = x\,. \tag{2.13}$$

Combining

$$x \cdot y + x \cdot y' = x\,, \quad (x + y) \cdot (x + y') = x\,. \tag{2.14}$$

DeMorgan 1 (inverted AND)

$$(x_1 \cdot x_2 \cdot x_3 \cdot \cdots)' = x_1' + x_2' + x_3' + \cdots\,. \tag{2.15}$$

DeMorgan 2 (inverted OR)

$$(x_1 + x_2 + x_3 + \cdots)' = x_1' \cdot x_2' \cdot x_3' \cdot \cdots . \tag{2.16}$$

Duality (Metatheorem) non-trivial and important

Any theorem or identity remains valid if '0' and '1' are swapped and '+ ' and '' are swapped also. The '+ ' and '· ' operators exchange precedence after applying duality theorem as well. Example:

$$x + x \cdot y \cdot 1 = x \cdot (x + y + 0) = x . \tag{2.17}$$

1.1.5 Boolean Expression Types

The following types are just definitions. No boolean mathematics is involved with it.

Literal

Variable or complement of a variable: x, x'.

Product term

Logical product of one or more literals: x · y · z'.

Sum term

Logical sum of one or more literals: x + y' + z.

Sum of products

Logical sum product terms: x · y' · z' + x' · y · z + x' · y' · z.

Product of sums

Logical product of sum terms: (x + y' + z') · (x' + y + z) · (x' + y' + z).

Normal term

Logical term where no variable occurs more than once: x + y' · z'.
Any non-normal term can always converted to a normal term applying the previous theorems.

Minterm

Logical product as a normal term: x · y' · z'.

Maxterm

Logical sum as a normal term: x + y' + z'.

Canonical sum

Logical sum of minterms (corresponding to truth table rows which produce a '1'). This is denoted as $\Sigma_{x,y,z}$ for a three variable truth table.

Canonical product

Logical product of maxterms (corresponding to truth table rows which produce a '0').
This is denoted as $\Pi_{x,y,z}$ for a three variable truth table.

1.1.6 Circuit Minimization

Circuits based on canonical sums or canonical products are often expensive because they contain unnecessary terms. Circuit minimization is based on the combining theorem. If the number of variables is not greater than 4 the minimization can be carried out graphically by **Karnaugh maps**.

x2	x1	y
0	0	0
0	1	1
1	0	0
1	1	1

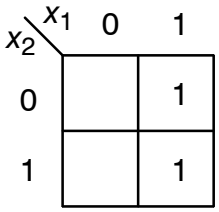


Figure 1.1: 2-variable truth table and Karnaugh map

x3	x2	x1	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

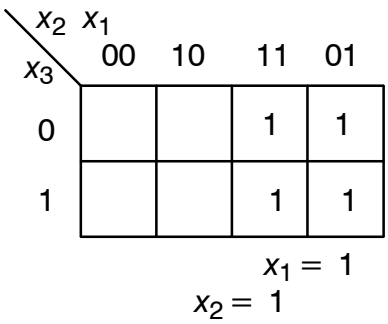


Figure 1.2: 3-variable truth table and Karnaugh map

For any adjacent cell only one input variable changes it's value. This is the reason for the ordering of rows and columns in the Karnaugh map.

Karnaugh maps have no start and end column or row. It can be any column or row.

x4	x3	x2	x1	y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

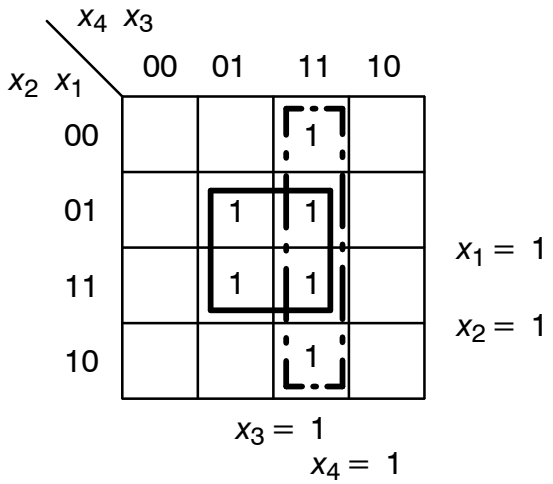


Figure 1.3: 4-variable truth table and Karnaugh map

Any set of adjacent ’0’ belongs to a minimal maxterm; a set of adjacent ’1’ form a minimal minterm. A large “area” corresponds to a simple term and vice versa. Thus, it is desirable to find the largest possible areas.

We will focus the theory to ’1’. The same theory applies to ’0’ by the duality theorem.

Prime Implicant

A prime implicant a maximum area of ’1’ (see areas in fig. 1.3). Of course smaller areas exist but they are no prime implicants.

A minimal sum is a sum of prime implicants.

The sum of all prime implicants is the *complete sum*. This is not necessarily minimal.

Distinguished 1-cells

A cell which is covered by only *one* prime implicant.

Essential prime implicant

A prime implicant that covers at least *one* distinguished 1-cell.

Every essential prime implicant must be included in a minimal logic function.

After removing all ’1’ from the essential prime implicants the remaining ’1’ need to be covered by additional prime implicants. If there is no distinguished 1-cell, any single cell can be treated as a distinguished 1-cell. Finding the truly minimized logical function is sometimes a non-trivial task.

Lab #01:

1.2 Prime Number Detector Design and Simulation

A detector for 4 bit prime numbers is required. It should be designed in a minimal form. It should be functionally verified by simulation.

- Find all prime numbers in the range of 4 bit unsigned integers.
- The input numbers are denoted as `x3...x0`. The output (= prime number detected) is denoted as `prnum`.
- Write down the truth table for the above logical function.
- Draw the Karnaugh map. Write the corresponding decimal numbers and the boolean '1' into the appropriate cells.
- Derive the minimal logical function for `prnum`.
- Draw the corresponding schematic diagram for the above boolean equation.
- Start a new Xilinx ISE project ("pridetct") and code the boolean equations as a VHDL top module.
- Simulate your design with Xilinx ISim (ISE simulator). Write another VHDL module for providing the stimulus data to test the design.

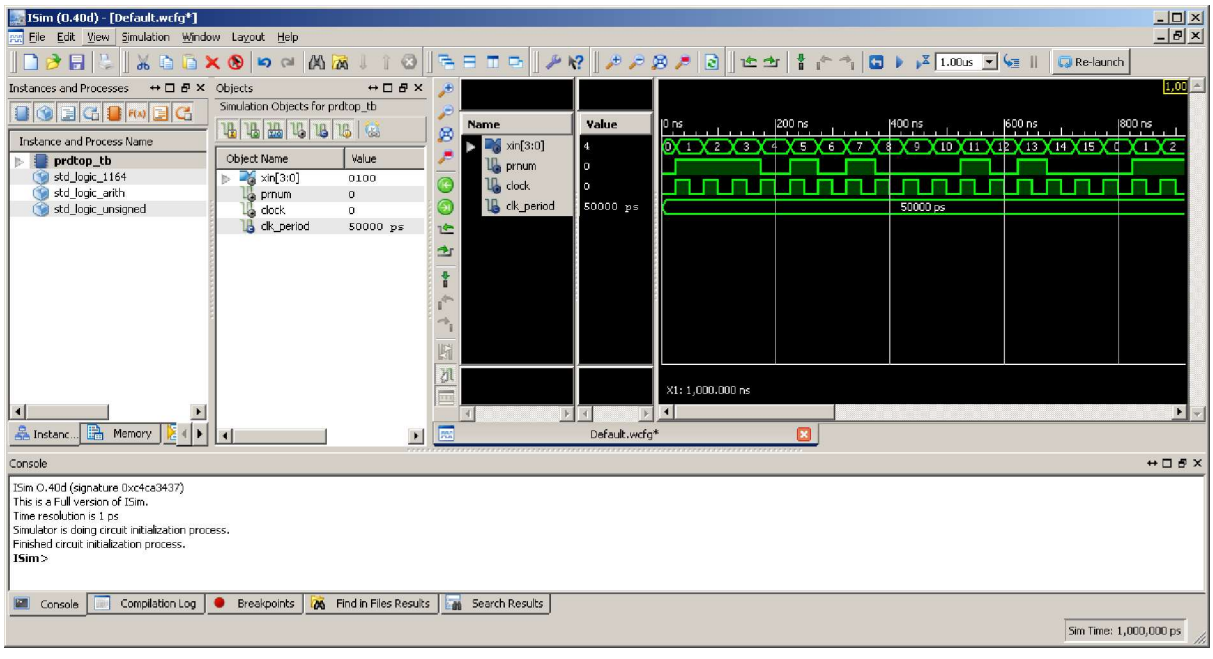


Figure 1.4: Xilinx ISim result

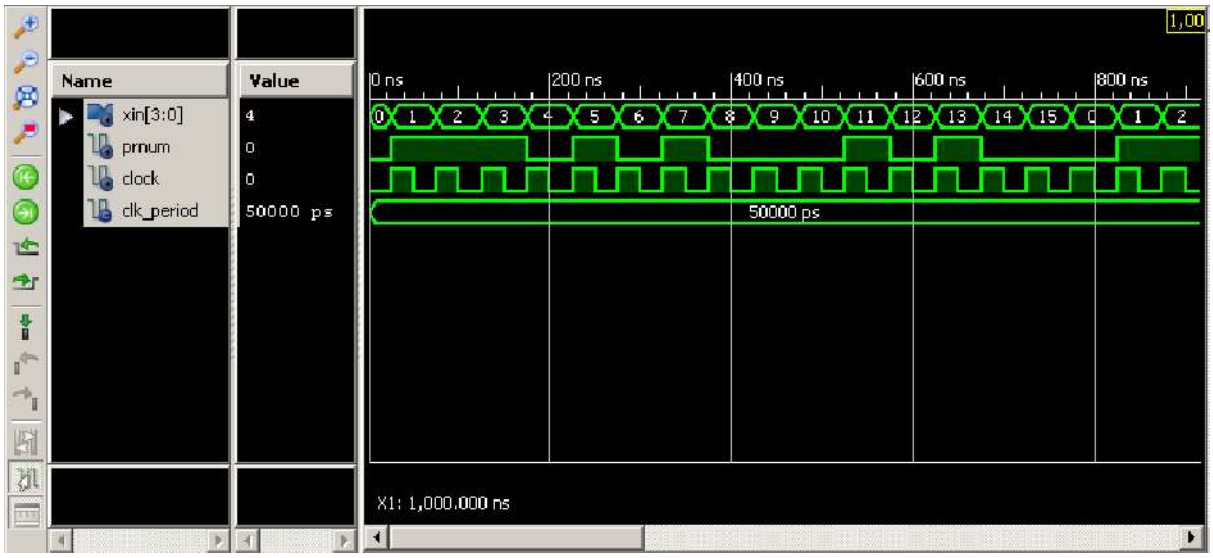


Figure 1.5: Xilinx ISim simulation details (yes, it works!)

HINT: instead of coding low level logic the program can be written in a truly behavioral manner. VHDL will compile most likely to the same RTL (register transfer level). Here is an example (processes will be discussed later):

```
WITH xin SELECT
    isprime <= '1' WHEN x"1" | x"2" | x"3" | x"5" | x"7" | x"b" | x"d",
               '0' WHEN OTHERS;
```

2 VHDL

VHDL stands for **V**ery **H**igh Speed Integrated Circuit **H**ardware **D**escription **L**anguage. The language was sponsored by the U.S. Department of Defense (DoD) in the 1980s. The standards are maintained by the IEEE. Standards:

VHDL–87	(first standard)
VHDL–93	(extension, quasi-standard for current designs)
VHDL–2006	(subset of PSL = Property Specification Language, extension of operators
VHDL–2008	(proposal for object orientation, PSL integration =)

Similar language is **Verilog**. It was developed in parallel to VHDL and is also widely used. Modern development tools support both VHDL and Verilog.

VHDL has a good structure to design large scale systems.

Properties of VHDL:

- Well suited for simulation and synthesis (only a subset of VHDL is synthesizable)

- hierarchical structure (Entity – Architecture)
- not case-sensitive
- strong data type concept
- modeling of concurrency, timing and clocking
- extendable by libraries, strong good library concept

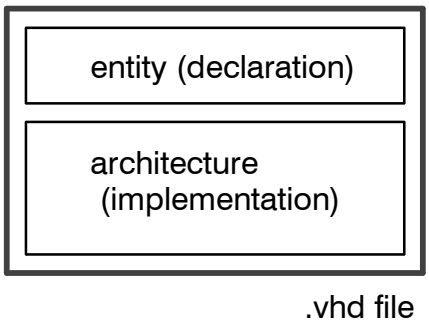


Figure 1.6: VHDL module structure

The entity block (can be regarded as a "header") contain the port definition with the signals that enter or leave the module (without internal signals!). Example:

```
entity prdtop is
    Port ( xin : in  STD_LOGIC_VECTOR(3 downto 0);
           prnum : out  STD_LOGIC);
end prdtop;
```

The above example requires the use of the following library

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

The types `STD_LOGIC` and `STD_LOGIC_VECTOR` are no VHDL predefined data types.

The data type has to be preceded by the data type direction:

- in (input)
- out (output)
- inout (input and output signal, typically threestate IOs as top level signals)
- buffer (outputs, but readable by VHDL instructions, a variable of direction type `out` is not readable!)

The predefined types in VHDL are:

- bit, bit_vector
- boolean
- character
- integer, real (integer range from -2^{31} ... $+2^{31}$ -1)

- string
- time

New types can be declared like `type my_color is (red, green, blue).` Types are fully synthesizable; the VHDL synthesis process will create automatically the required number of bits for a SIGNAL or VARIABLE. Moreover an optimal encoding scheme will be used (binary, gray, one-hot) for synthesis.

The range of a type can be restricted (subtype):

```
subtype my_int is integer range -4 to 12;
```

Subtypes can detect possible errors at an early stage. The following code will give an error during compilation (with the above definition of my_int):

```
VARIABLE  k : my_int;
begin
      k := 42;
```

Arrays as ordered sets of the same element type can be declared with the array keyword:

```
TYPE opcode IS ARRAY (1 to 64) OF integer;
```

CONSTANT declarations increase the readability of the source code. It is also very useful for writing reusable code:

```
CONSTANT  N_Regs : integer := 4;
```

2.1 Data Type STD_LOGIC and STD_LOGIC_VECTOR

These data types are defined and implemented in the IEEE ieee.std_logic_1164 library. Physical signals, inputs and outputs can be modeled with this type, since it well suited for “real” circuits. Any STD_LOGIC signal or variable can have one of the following 9 values:

’U’	–	uninitialized. This signal hasn’t been set yet (for simulation).
’X’	–	unknown. Impossible to determine this value/result
’0’	–	logic ’0’ (same as in VHDL builtin BIT)
’1’	–	logic ’1’ (same as in VHDL builtin BIT)
’Z’	–	High Impedance
’W’	–	Weak signal, not known if ’0’ or ’1’
’L’	–	Weak signal will go to ’0’ if not driven by a strong signal
’H’	–	Weak signal will go to ’1’ if not driven by a strong signal
’–’	–	Don’t care, useful for the design of minimal logic

Note that ’0’ or ’1’ is required instead of 0 or 1 to distinguish between logic and numbers.

Several convenience methods exist to set arrays, especially STD_LOGIC_VECTORS. If the variable x is of type STD_LOGIC_VECTOR (0 to 7) is declared, the following constructs are legal (and identical!):

- x := ('1', '0', '1', '1', '1', '0', '1', '1');
- x := (1 => '0', 5 => '0', others => '1'); -- useful for large arrays
- x := "10111011";
- x := "1011" & "1011";

2.1.1 Operator Overloading for STD_LOGIC_VECTOR

IEEE libraries take advantage of the operator overloading possibilities of VHDL. Algebraic operations therefore possible with STD_LOGIC_VECTOR values.

If ieee.std_logic_unsigned is included, the following operation is valid:

```
x := y + z;        -- all vectors must have the same length
if (z = y) then ...
```

In the above case the sum of *y* and *z* is calculated. Note that this results in a very high implementation complexity and requires many resources.

Lab #02:

2.2 Decoder (Synthesis and Simulation)

In the past when digital systems where designed with discrete ICs, 74xx138 was used to decode several devices (for instance ADCs or ports) which could be connected to a common bus (common data lines).

Function Tables
LS138

Inputs					Outputs							
Enable		Select										
G1	G2 (Note 8)	C	B	A	YO	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	H	H	H	H	H	L	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	L	H	H	H	H	H	H	L	H	H
H	L	H	H	L	H	H	H	H	H	H	L	H
H	L	H	H	H	H	H	H	H	H	H	H	L

H = High Level, L = Low Level, X = Don't Care

Note 8: G2 = G2A + G2B

Figure 1.7: Truth table (© Fairchild)

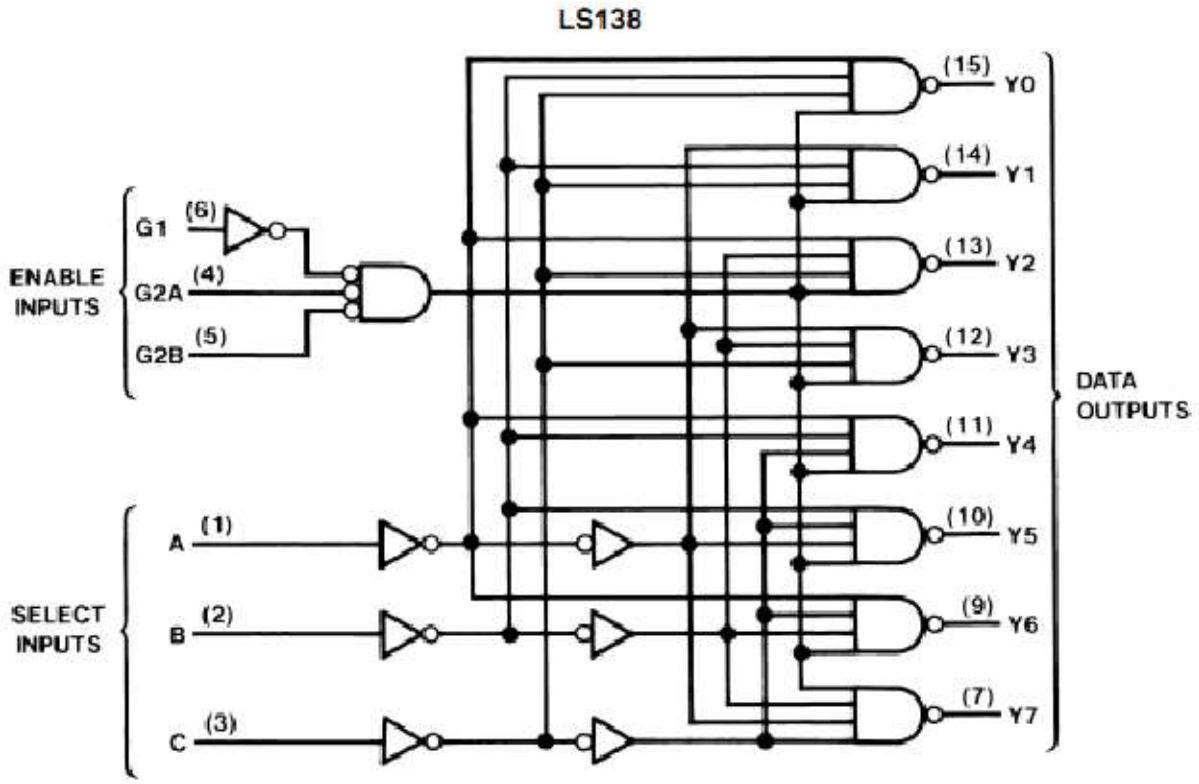


Figure 1.8: Equivalent logic diagram (© Fairchild)

- Start a new Xilinx ISE project ("ls138") and write a VHDL top level module which declares I/O and implements the LS138 logic. It is not required that this is done by logic gates as in fig. 1.8.
  HINT: use "if" and "case" for a true behavioral description of the logic.

- Simulate your design with Xilinx ISim (ISE simulator). Write another VHDL module for providing the stimulus data to test the design.

***

2.3 Implementing Logic (VHDL Architecture)

Every programing language describes the way information is processed. Since VHDL does not know objects and methods of objects (object orientation is probably totally inadequate for a hardware description language) all instructions refer to data processing and conversion. Of course, VHDL comprises all modern flow control mechanisms.

2.3.1 Concurrency

Unlike other programing languages the instructions in VHDL are executed in parallel. The particular sequence of instructions has no impact on the meaning of the program. However, if a sequence of instructions is required for an algorithm VHDL has the concept of a `PROCESS`. All processes are executed in parallel.

The concept of concurrency reveals the true nature of hardware – parallel information processing. This is the benefit of a hardware solution instead of a software solution: hardware is magnitudes faster than software execution. Since embedded systems often requires real-time responses, a realization in hardware is often required to meet timing specifications.

The following two examples are equivalent:

```
x1 <= a AND not b;
x2 <= a XNOR c;

x2 <= a XNOR c;
x1 <= a AND not b;
```

Because of concurrency the following code might be misleading and is not recommended (although it is legal):

```
x1 <= a AND not b;
x2 <= x1 xor c;
```

It seams that the second equation depends on the first which is not true. Later in this chapter will be shown how the `PROCESS` can be used to solve this issue.

A clear contradiction to concurrency is:

```
x1 <= a;
x1 <= NOT c;
```

Synthesis will become impossible since there is no solution to this contradiction. This is called a “multiple driver error” for x_1.

2.3.2 Constants

As outlined in the `ENTITY` section a constant consists of a type and a value:

```
CONSTANT  SMALL_REG_SIZE : integer := 8;
```

2.3.3 Ports

The port definitions of the ENTITY section can be used (and must be used!) in the architecture section. Depending on the data direction type it can be written, read or both. The assignment operator is <= . Example:

```
led(2:4) <= "011";            -- light leds 3 and 4, switch led 2 off
```

2.3.4 (Shared) Variables and Signals

Signals and variables are used to hold values. They seem to serve the same purpose but it is important to choose the correct type. Ports, however, are signals by default.

SIGNALS

Signals are used to pass values in and out of a circuit and between components of a circuit. They are what was a "wire" in physical circuits or copper line on an printed circuit board. Signals are defined in declarative part of an ARCHITECTURE or PACKAGE . Example:

```
signal  q0 : BIT;
```

Signals can be read and written, for example the following code is valid:

```
q0 <= a XOR b;
b <= q0 AND c;
```

The assignment operator is <= .

VARIABLES

A variable has a strictly local scope. It is restricted to PROCESS , FUNCTION , PROCEDURE , or PACKAGE BODY . Therefore it is very useful in *sequential* code (see following section). Since variables are part of sequential code they are similar to variables of software programing languages like C++ or Java. Example:

```
VARIABLE  state_no : integer;


state_nr := 0;
if (start = '1') then
          state_nr := state_nr + 1;
end if;
```

The assignment operator is := .

SHARED VARIABLES

A shared variable – as the name indicates – has a scope in more than one process (a global variable). It must be declared in the declaration section of an ARCHITECTURE .

```
ARCHITECTURE  shared_example OF my_entity IS
          SHARED VARIABLE  state_count : integer RANGE 0 to 6;
begin
   .
   .
```

The variable state_count has a global scope inside of the architecture `shared_example` .

Comparison of SIGNALS and VARIABLES

SIGNALS cannot be declared in sequential code; VARIABLES (except SHARED VARIABLES cannot declared outside sequential units (= process, subprograms).

Only one effective assignment in the whole code is allowed for SIGNALS. VARIABLES may have *multiple* assignments.

SIGNAL update: The new value of a signal is only available if the process of subprogram has concluded.
VARIABLE update: The update occurs immediately. The new value can be used in the next line of code or in subsequent lines.

REGISTER INFERENCE (will be discussed later in detail):
D-flip-flops (DFF) are created whenever an assignment occurs on the transition of a signal. No difference exists between VARIABLES and SIGNALS in this case. However the corresponding VARIABLE must affect a SIGNAL's value eventually. EXAMPLE:

```
ARCHITECTURE behavoiral OF my_circuit IS
SIGNAL  buf4 : std_logic_vector(3 downto 0);
begin
          PROCESS(clk)
                    VARIABLE tmp : integer RANGE 0 to 31;
                    if rising_edge(clk) then
                              buf4 <= x;
                              tmp := y;
                    end if;
          end PROCESS;
          .
          .
```

The above example infers 9 DFFs (4 for `buf4`, 5 for `tmp`).

Flip-Flop inference depends always in the transition of a **signal**!

2.3.5 Structural Design Elements (Accessing Lower-Level-Modules)

The prime number detector in section 1.2 may have the following `ENTITY` declaration:

```
entity prdtop is
    Port ( xin : in  STD_LOGIC_VECTOR(3 downto 0);
```

```
            prnum : out  STD_LOGIC);
end prdtop;
```

This component (circuit) can be used in other modules if they contain a `COMPONENT` section inside of their `ARCHITECTURE`:

```
ARCHITECTURE structured_arch of big_appl IS
COMPONENT prdtop is
          Port ( xin : in  STD_LOGIC_VECTOR(3 downto 0);
                 prnum : out  STD_LOGIC);
end prdtop;

begin
          pdetect_1: pdrtop port map (
                    xin => local_xin,
                    prnum => prnum_1);
```

The signals `local_xin` and `prnum_1` must be local signals of appropriate type. The component can be used any number of times. This is the explicit form.

An implicit form exists also, the port names can be omitted. However, the sequence of signals from the component declaration needs to be preserved:

```
ARCHITECTURE structured_arch of big_appl IS
COMPONENT prdtop is
          Port ( xin : in  STD_LOGIC_VECTOR(3 downto 0);
                 prnum : out  STD_LOGIC);
end prdtop;

begin
          pdetect_1: pdrtop port map(local_xin, prnum_1);
```

There is no need to define signals with different names for the local variables. It is common practise to use the following code:

```
ARCHITECTURE structured_arch of big_appl IS

COMPONENT prdtop is
          Port ( xin : in  STD_LOGIC_VECTOR(3 downto 0);
                 prnum : out  STD_LOGIC);
end prdtop;

begin
          pdetect_1: pdrtop port map (
                    xin => xin,
                    prnum => prnum);
```

Lab #03a:

2.4 Structural Design (PORT MAP Usage Example 1)

The following structural design should be carried out using three VHDL modules.

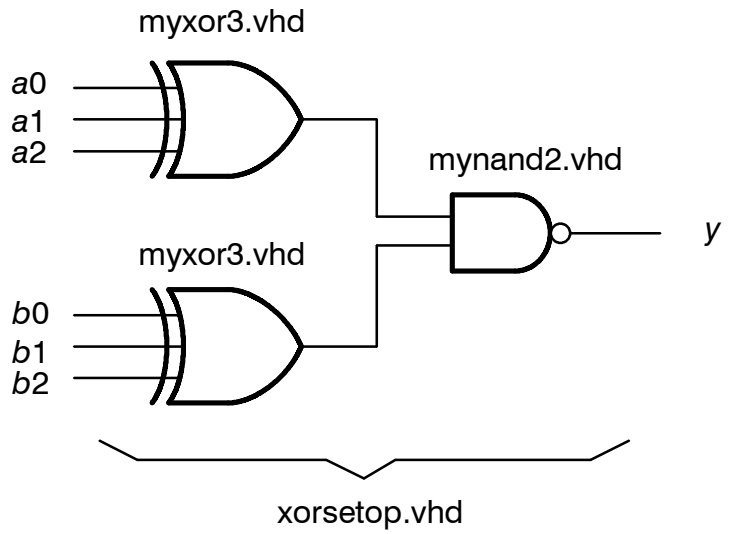


Figure 1.9: Structural design

- Create a new project "xorse"
- Create a VHDL module "xorsetop.vhd" with ENTITY

```
entity xorsetop is
    Port ( a0 : in STD_LOGIC;
           a1 : in STD_LOGIC;
           a2 : in STD_LOGIC;
           b0 : in STD_LOGIC;
           b1 : in STD_LOGIC;
           b2 : in STD_LOGIC;
           y : out STD_LOGIC);
end xorsetop;
```

- Instantiate "myxor3" and "mynand2" in "xorsetop" and make all connections (using signals).
- Note that "myxor3" and "mynand2" are unresolved. Add design sources "myxor3.vhd" and "mynand2.vhd" so resolve the dependencies.
- Simulate the design with ""xorse_tb.vhd" test bench file (supplied file). The results should look as below:

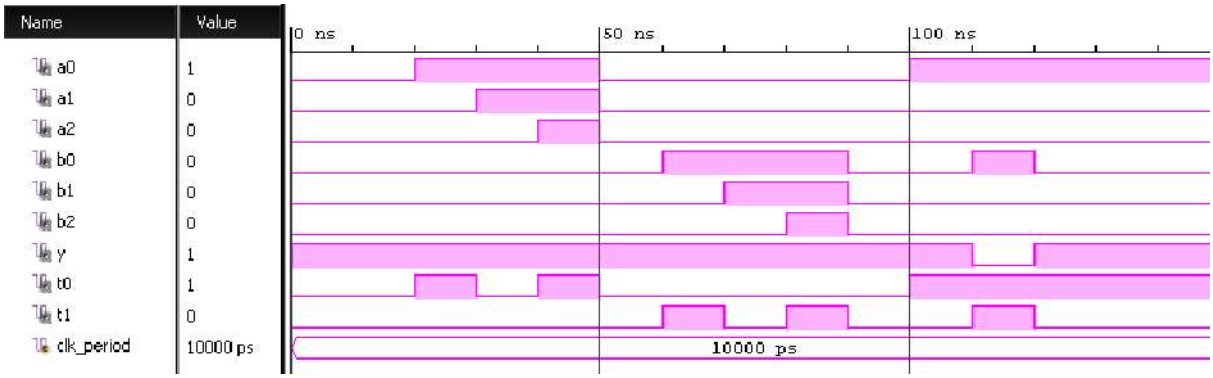


Figure 1.10: Simulation of structural design

Lab #03b:

2.5 Device Primitive Inference (PORT MAP Usage Example 2)

The best possible performance can be obtained by using FPGA device primitives (optimized elements in an FPGA for special purposes). This comprises (not complete):

- RAM (block ram, dual port RAM
- DSP (multiplier, accumulator)
- Buffers (IO)
- Clock manager
- JTAG Logic
- Intellectual Property protection
- PCI Express interface
- build-in microcontrollers
- Shift registers

It is not guaranteed that a shift register is always implemented by a device primitive. It this lab the `SLR16E` 16 bit shift register should be used in a VHDL module. It can be configured by it's A input from a 1-bit (A = "0000") to a 16-bit (A = "1111") shift register. The simulation in fig 1.10 shows a 6-bit shift register (A = "0101").

▶ Create a new project "sr16pr"

▶ Create a VHDL module "sr16top.vhd". Instantiate the device primitive SRL16E. The VHDL module header must include the following library:

```
Library UNISIM;
use UNISIM.vcomponents.all;
```

▶ No component declaration is necessary in the declaration part of the ARCHITECTURE, since this is defined in the above library. Next instantiate the shift register with the following template (the parameters are explained by comments):

```
SRL16E_inst : SRL16E
generic map (
   INIT => X"0000")
port map (
   Q => Q,       -- SRL data output
```

```
      A0 => A0,      -- Select[0] input
      A1 => A1,      -- Select[1] input
      A2 => A2,      -- Select[2] input
      A3 => A3,      -- Select[3] input
      CE => CE,      -- Clock enable input
      CLK => CLK,    -- Clock input
      D => D         -- SRL data input
   );
```

- The INIT generic parameter is the initial value of the register flip-flops. The CE (clock enable) port can be hardwired to '1', i.e. it can be written

```
CE => '1',
```

- Create simulation data to verify proper operation of the shift register (see below).

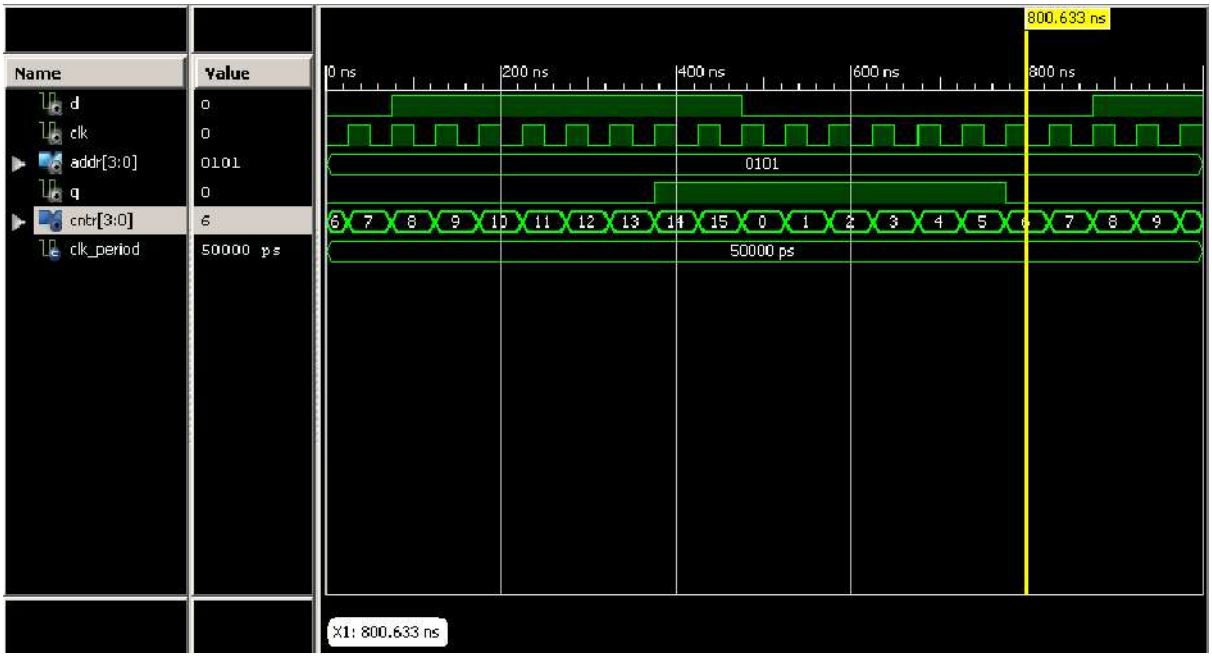


Figure 1.11: Simulation of the 6 bit shift register

- Declare the logic of a different module in your `ARCHITECTURE` .

```
COMPONENT MyLogic PORT ( a: in std_logic; b : out std_logic);
end COMPONENT;
```

- Instantiate this logic component in your `ARCHITECTURE` implementation:

```
ml1: MyLogic PORT map (a => D, b => open);
```

- How does that affect the process window in the implementation flow?

2.6 Predefined Signal Attributes

Attributes for all kinds of data types can be defined to increase readability of code and to write compact code. Of great importance are the predefine attributes for signals, since they are required for for FF (Flip-Flop) instantiation as well as for simulation purposes. It is obvious that the `'event` attribute is very useful for synthesis. The attributes used for synthesis are marked in grey.

Attributes are appended to the a signal's name by the prime " ' " character. if for instance the the signal

```
signal  x : std_logic;
```

exists, than the following *predefined* attributes are available.

attribute	return type	result
x'delayed[(t)]	signal (the base type of x)	same signal but delayed t units of time
x'stable[(t)]	boolean	TRUE if the signal hasn't changed for t units of time, FALSE otherwise
x'quiet[(t)]	boolean	TRUE if no transaction has been scheduled for t units of time, FALSE otherwise.
x'transaction	BIT	Becomes '1' is a transaction is scheduled in the simulation cycle, '0' otherwise.
x'event	boolean	TRUE if a signal change has just occured (low to high or high to low), FALSER otherwise.
x'active	boolean	TRUE is a transition is scheduled, FALSE otherwise
x'last_event	TIME	Elapsed time since last event on x
x'last_active	TIME	Elapsed time since a transaction was scheduled for x.
x'last_value	signal (base type of x)	Value of x before last event, current value if no event occured yet.
x'driving	boolean	TRUE if the process is driving x, FALSE if x disconnected from driver.
x'driving_value	signal (base type of x)	Value of the driver for x ion the current process.

[(t)] denote optional arguments of a time unit. If not specified $t = 0$ is assumed.

It is not required that the signal is of type `std_logic`.

2.7 Abstract Concurrent Control Structures

The benefit of high level programing languages is an abstract description of algorithms. Therefore, complex logic can be described much easier as with hardware related logic. It is up to the VHDL compiler to optimize these structures to an efficient register transfer level.

The control elements in this section can be used in a normal concurrent environment, i.e. between `BEGIN` and `END` in an `ARCHITECTURE`.

These elements cannot be used in sequential code (see section 3).

> In VHDL 2008 the control structures may be used in sequential code also.

2.7.1 WHEN

```
< assignment expression>  WHEN  < condition(s)>  ELSE
     < assignment expression>  WHEN  < condition(s)>  ELSE
     ...;
```

It it not necessary that all input values are assigned. Example:

```
x <= '0'  WHEN  inita = '0'  ELSE
     '1'  WHEN  a XOR b  ELSE
     '-';                    -- don't care
```

The last line of code prevents the generation of unnecessary (or unwanted) logic.

If pure combinational logic is intended (i.e. truth table logic) all input options should be covered. This prevents inference of latches. The keyword `OTHERS` is suited to cover all input combinations. Not doing so may **result in a latch** (probably unwanted).

According to [3] a D-latch is created with both code fragments:

```
q <= '0' WHEN rst = '1'  ELSE
     d  WHEN  clk = '1';

q <= '0' WHEN rst = '1'  ELSE
     d  WHEN  clk = '1'  ELSE
     UNAFFECTED;
```

The `UNAFFECTED` keyword explicitly expressed that a memory element is wanted. The first example should synthesize in the same way.

2.7.2 SELECT

SELECT is the concurrent equivalent to the sequential CASE (see below). It uses multiple values (*not conditions*) to select different outputs. Example:

```
WITH  intval  SELECT
    x <= '0'  WHEN  0 | 1 | 4,
         '1'  WHEN  6 TO 8,
         'Z'  WHEN OTHERS;
```

The vertical bar "| " has the meaning of "or"; with "to" a range can be specified. **All** possible input values have to be covered, therefore the OTHERS keyword is helpful.

With keyword UNAFFECTED memory (latches) are created:

```
WITH  intval  SELECT
    y <= "001"  WHEN  0 | 1 | 4,
         "010"  WHEN  6 TO 8,
         "100"  WHEN  10,
         UNAFFECTED  WHEN OTHERS;
```

2.7.3 GENERATE

GENERATE is the concurrent counterpart of the LOOP statement in sequential code. It is very useful to repeat code a number of times. It often reduces the number of code lines.

Unconditional GENERATE (also called FOR-GENERATE):

```
flip:  FOR  k in 0 to 31 GENERATE
    x(k) <= b(31-k);
end GENERATE;
```

Conditional GENERATE (also called IF-GENERATE):

```
cxor:  FOR  k in 0 to 7 GENERATE
    x(k) <= '1'  WHEN (b(k) XOR c(k))='1';  ELSE  '0';
end GENERATE;
```

If *x* was define as `signal  x : BIT_VECTOR(0 TO 7);` then a more elegant and flexible code would be:

```
cxor:  FOR  k in x'RANGE  GENERATE
    x(k) <= '1'  WHEN (b(k) XOR c(k))='1';  ELSE  '0';
end GENERATE;
```

In this example x'RANGE is expanded to x'LEFT to x'RIGHT which becomes 0 TO 7 . Of course, 'RANGE only applies to arrays.

***

3 Sequential Logic and Sequential Code

A note on labels: Labels are optional and can be used for all control structures, for instance

```
wait_Label_1:  WAIT  FOR  15ns;
```

Since label are rarely used in VHDL programing it will be omitted for the description of commands.

All sequential code examples assume the declaration of appropriate `SIGNALS` of `VARIABLES`. These declarations are omitted.

Concurrent code (see 1.1 and subsequent sections) is mainly used for combinational logic. Storage elements like latches can be created with concurrent code:

```
q <= d WHEN clk = '1' ELSE  UNAFFECTED;
```

The result is a D-latch which stores data when clk= '0'.

Remember that concurrent code uses control elements `WHEN`, `SELECT`, and `GENERATE`.

More sophisticated combinational logic and sequential logic (like state machines) require **sequential code**. This means that concurrent code can only create combinational code (+ latches). Sequential code will generate combinational logic *and* sequential logic (using FFs).

Sequential code occurs in a `PROCESS`, a `FUNCTION`, or in a `PROCEDURE`. The control flow instructions in sequential code are `IF`, `WAIT`, `LOOP`, and `CASE`.

Sequential code allows the use of `VARIABLES`.

Control elements for sequential and for concurrent code sections can only be used exclusively in sequential or concurrent code.

While latches can be inferred from concurrent code, flip-flops can generated easily from sequential code. Therefore, state-machines are programmed in sequential code. The commonly used latches (DL) and flip-flops (DFF) are summarized below.

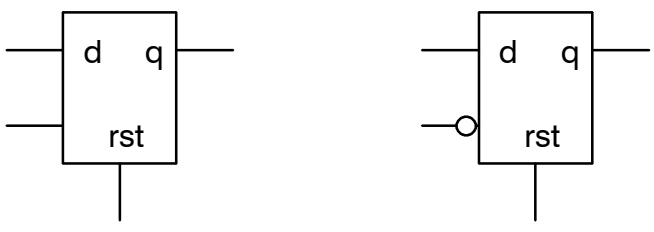


Figure 1.12: Positive and negative level DL (D-latch) with reset

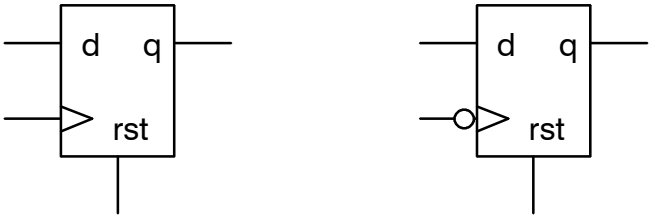


Figure 1.13: Positive and negative edge DFF (D-flip-flop) with reset (asynchronous)

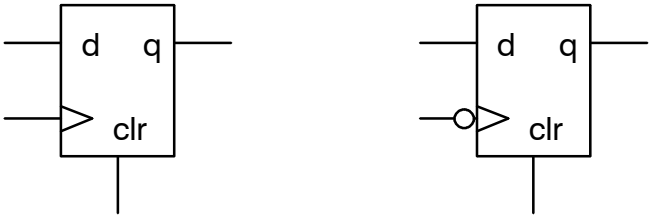


Figure 1.14: Positive and negative edge DFF (D-flip-flop) with clear (synchronous)

> Using concurrent code where possible is recommended, since this will result in less complex logic in the synthesis stage.

The VHDL synthesizer will probably optimize unnecessary logic away, but it is not recommended to always use sequential logic especially with `IF - ELSEIF - ELSE` which will likely create a so-called *priority-decoder*.

3.1 IF (Similar to WHEN in Concurrent Code)

The IF command is widely used since it allows the inference of latches and flip-flops.

```
IF (condition) THEN
     <assignments>;
ELSIF (condition) THEN
     <assignments>;
ELSE
     <assignments>;
END IF;
```

```
IF (x = '0' AND k > 0) THEN
          result := 42;
ELSEIF (k = 0) THEN
          qs <= "010111";
ELSE
          qs <= (others => '0');
END IF;
```

Note that this creates a latch of appropriate size for `result` and `qs`. If this is not intended implement the full truth table probably by using don't cares ('–').

3.2 WAIT (no concurrent counterpart)

The WAIT command is only useful for simulations. In this case the command is very helpful. In any kind it is not synthesizable.

If WAIT is used no sensitivity list is allowed.

```
WAIT UNTIL (condition);
```

```
WAIT ON <sensitivity list>;
```

```
WAIT FOR <time expression>;
```

For simulations the following examples are equivalent:

```
PROCESS(clk)    -- depreciated
BEGIN
          IF (clk'event AND clk='0') THEN
                    .
                    .
                    .
          END IF;
END PROCESS;

PROCESS(clk)
BEGIN
          IF rising_edge(clk) THEN
                    .
                    .
                    .
          END IF;
END PROCESS;

PROCESS                        -- no sensitivity list
BEGIN
          WAIT UNTIL (clk'event AND clk='0') THEN
                    .
                    .
                    .
          END IF;
END PROCESS;
```

3.3 LOOP (Similar to GENERATE in Concurrent Code)

The LOOP command allows the instantiation of code for several times. Complex programs can thus be written in a compact form.

```
LOOP
     <sequential statements>;
END LOOP;
```

```
FOR  <identifier>  IN  <range>  LOOP
     <sequential statements>;
END LOOP;
```

```
WHILE  (condition)  LOOP
     <sequential statements>;
END LOOP;
```

```
FOR  <identifier>  IN  <range>  LOOP
     <sequential statements>;
     EXIT  WHEN  (condition);
     <sequential statements>;
END LOOP;
```

```
FOR  <identifier>  IN  <range>  LOOP
     <sequential statements>;
     NEXT  WHEN  (condition);
     <sequential statements>;
END LOOP;
```

```
  FOR  k  IN  k´RANGE LOOP
           b(k) <= a(a´RIGHT - k);
           NEXT WHEN i=7;
           y(k) := 2*k;
  END LOOP;
```

3.4 CASE (Similar to SELECT in Concurrent Code)

The `CASE` command is quite similar to `SELECT` , however, the syntax is different.

```
CASE  (expression)  IS
     WHEN  <value> => assignments;
     WHEN  <value> => assignments;
     WHEN  <value> => assignments;
     WHEN  others => assignments;
END CASE;
```

For an example see 3.5.1.

3.5 Design Guidelines for Sequential Code

When FFs are required sequential code is necessary. Combinational logic can be created by concurrent of sequential code.

> If combinational logic can be expressed by concurrent code – use concurrent code.

If sequential logic is needed a process becomes necessary.

> Include all input signals that are read inside of a process in it's sensitivity list. Do not ignore compiler warnings in this case.

3.5.1 Avoidance of Unnecessary Latches

Creating combinational logic in a `PROCESS` may easily create unnecessary logic (latches).

> The truth table should always be *completely* specified.

Failure to do so will infer (probably many) unwanted latches.

Example for a bad design (only `ARCHITECTURE` is shown) according to [3]:
sel is declared as

```
sel : integer RANGE 0 to 3;

ARCHITECTURE behavioral of my_design IS
BEGIN
          PROCESS(a,b,c,d,sel)
          BEGIN
                    CASE sel IS
                              WHEN 0 => x <= a;  y <= '0';
                              WHEN 1 => x <= b;  y <= '1';
                              WHEN 2 => x <= c;
                              WHEN OTHERS => x <= d;
          END PROCESS;
END ARCHITECTURE;
```

Latches are avoided by specifying "don't care" values '–' in the design:

```
ARCHITECTURE behavioral of my_design IS
BEGIN
          PROCESS(a,b,c,d,sel)
          BEGIN
```

```
                    CASE sel IS
                              WHEN 0 => x <= a;  y <= '0';
                              WHEN 1 => x <= b;  y <= '1';
                              WHEN 2 => x <= c;  y <= '-';
                              WHEN OTHERS => x <= d; y <= '-';
          END PROCESS;
END ARCHITECTURE;
```

A good solution is the use of two (concurrent) SELECT statements.

```
ARCHITECTURE behavioral of my_design IS
BEGIN
          WITH sel SELECT
                    x <= a WHEN 0,
                          b WHEN 1,
                          c WHEN 2,
                          d WHEN others;
          WITH sel SELECT
                    y <= '0' WHEN 0,
                          '1' WHEN 1,
                          '-' others;
END ARCHITECTURE;
```

The last line could be omitted due to concurrent code but should be included for safety reasons and probably new versions of VHDL.

Lab #04:

3.6 Moving Light (Finite State Machine)

The LEDs on the Spartan6 board should light in a "moving" manner. Therefore, PROCESSES are required to implement this design (using IFs).

- The following user constraints file (UCF) is required. Generate this file by PlanAhead (preferred) of as an ASCII text file (works also).

```
NET "clk" LOC = L15;
NET "led[0]" LOC = U18;
NET "led[1]" LOC = M14;
NET "led[2]" LOC = N14;
NET "led[3]" LOC = L14;
NET "led[4]" LOC = M13;
NET "led[5]" LOC = D4;
NET "led[6]" LOC = P16;
NET "led[7]" LOC = N12;
NET "sw0" LOC = A10;
NET "sw1" LOC = D14;
NET "sw2" LOC = C14;

NET "led[0]" IOSTANDARD = LVTTL;
NET "led[1]" IOSTANDARD = LVTTL;
NET "led[2]" IOSTANDARD = LVTTL;
NET "led[3]" IOSTANDARD = LVTTL;
NET "led[4]" IOSTANDARD = LVTTL;
NET "led[5]" IOSTANDARD = LVTTL;
NET "led[6]" IOSTANDARD = LVTTL;
NET "led[7]" IOSTANDARD = LVTTL;
NET "led[0]" DRIVE = 6;
NET "led[1]" DRIVE = 6;
NET "led[2]" DRIVE = 6;
NET "led[3]" DRIVE = 6;
NET "led[4]" DRIVE = 6;
NET "led[5]" DRIVE = 6;
NET "led[6]" DRIVE = 6;
NET "led[7]" DRIVE = 6;
```

- Note that the clk signal is 100MHz. You need to divide the signal by a binary counter of 24 Bits. The MSB toggles at a rate of 100 MHz / 2^ 24 ≈ 6 Hz. So the FPGA is slowed down by a factor of 16,777,216.

- The switches should serve the following purposes:
  SW0: start/stop moving light (stop count)

SW1: direction of moving
SW2: Reset LEDs to “00000011”

- Download the design and Good luck with lighting up the LEDs!

***

4 System Level Programing

The design of larger systems becomes easier if the code is partitioned. Another important issue is the reuse of code. These aspects are covered by the following sections on packages, functions and procedures.

Since the code in packages are not directly included in source files programing packages is called system level programing.

4.1 Generic

The GENERIC statement provides a way to define constants during instantiation of components. It is part of an ENTITY declaration. It is possible to define a default if the generic parameter is not set. It must be placed before the PORT declaration. Example:

```
ENTITY special_gate IS
          GENERIC (Gate_Inputs : integer := 8);
          PORT ( x : in std_logic_vector(0 to Gate_Inputs-1);
                  y : out std_:logic);
END ENTITY;
```

When instantiating this component the ARCHITECTURE section is as follows

```
ARCHITECTURE  main IS
COMPONENT special_gate IS
          GENERIC (Gate_Inputs : integer := 8);
          PORT ( x : in std_logic_vector(0 to Gate_Inputs-1);
                  y : out std_:logic);
END COMPONENT;
signal  y_main : std_logic;
         x_main : std_logic_vector(0 to 2);

BEGIN
          .
          .
          spg1: special_gate
                    GENERIC MAP (Gate_Inputs => 3)
                    PORT MAP (x => x_main,
                              y => y_main);
          .
          .
END main;
```

During component instantiation the number of elements of the input vector has been changed to three.

4.2 Packages

Packages are accessed by declaring the corresponding library in the header area of a VHDL module. The elements can then be used inside of the ARCHITECTURE section (ENTITY is not involved here). See lab #03 in section 2.4 for a package usage example.

More general a package is a "container" for the definition of types, subtypes, constants, functions, procedures, and – widely used – for COMPONENT declarations.

In a package component declaration are possible. This avoids the component declarations in the ARCHITECTURE section of a VHDL module. This mechanism is very useful for the reuse of code.

The general structure of a package is:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package my_package is

          CONSTANT, FUNCTION, GENERIC, PORT definitions

end my_package;

package body my_package is

          implementation section (similar to ARCHITECTURE in VHDL modules)


end my_package;
```

4.3 Libraries

A library is a collection of compiled VHDL sources. Since PACKAGES are also VHDL files they are often part of libraries.

> The VHDL sources of a design are compiled into the default library "work".

It is not necessary to include the library "work" in VHDL files since this is done be default. If packages are referenced they have to included by a "use" instruction, e.g.

```
use work.my_package.all;
```

When accessing VHDL modules from a library the use of library elements is as shown in fig. 1.15. With a library containing a PACKAGE VHDL module the main code VHDL source become simpler and easier to handle.

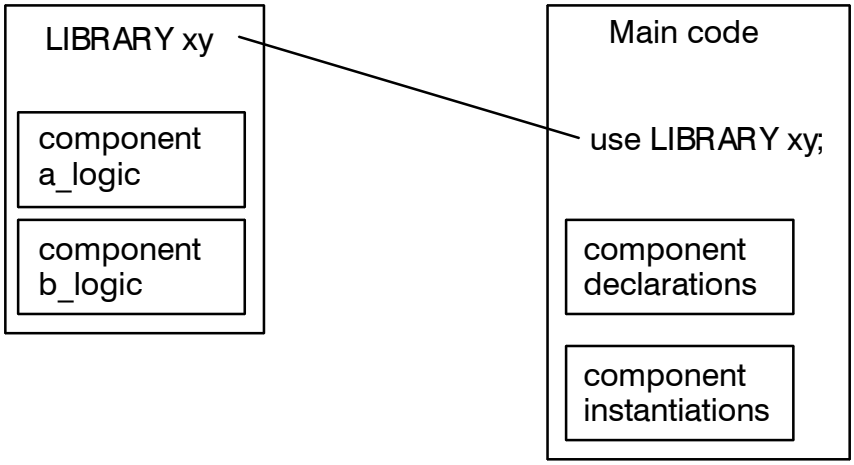


Figure 1.15: Instantiating components from a VHDL library (without package)

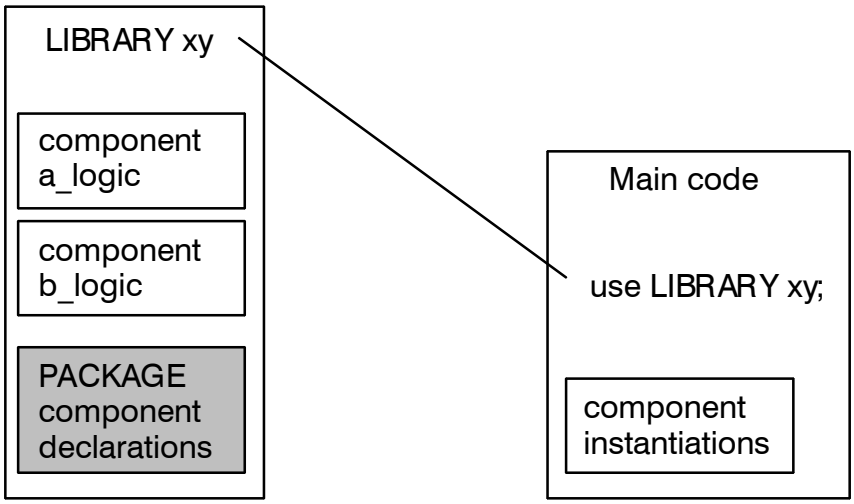


Figure 1.16: Instantiating components from a VHDL library (using package)

4.4 Functions and Procedures

FUNCTIONS and PROCEDURES are similar to functions and subroutines of other high level languages. A PROCEDURE can return several arguments while a (simpler) FUNCTION can return only only one argument, but is easier to use.

> FUNCTIONS are often used for explicit type conversions and allows the definition of overloading operators like "+" or "–".

A good example for the use of FUNCTIONs and PROCEDUREs are mnemonics conversions for an embedded microprocessor (see Lab #06 in section 4.6).

Functions and procedures can appear almost everywhere in VHDL code. The most common place is in a `PACKAGE`. This approach will be taken throughout the documentation.

4.4.1 Parameter Checking With ASSERT

The compiler can check only obvious errors like boundary exceptions of range violations. A much better test for proper use of parameter can be achieved with the ASSERT statement:

```
ASSERT < boolean expression>
REPORT < string>
SEVERITY < severity level> ;
```

Whenever the boolean expression becomes FALSE , the report string is displayed and an action according to the severity level is taken. The severity level can be any of the following levels:

NOTE	(information level)
WARNING	
ERROR	(unusual exception)
FAILURE	(unacceptable)

Example:

```
ASSERT  (x'LENGTH <= 16)
REPORT "Max. dim. of signal x (" & INTEGER'IMAGE(x'LENGTH) & ") exceeded."
SEVERITY FAILURE;
```

The output uses the 'IMAGE attribute to generate a readable output.

If an asset should be executed unconditionally the form is:

```
ASSERT  false
REPORT "Assert at position xyz has executed."
SEVERITY WARNING;
```

The ASSERT statement is very useful for simulation, since it can capture a lot of errors.

Lab #05:

4.5 Packages and Libraries

- Start a new project with name "pgktest" and create a new top level VHDL module.
- Create a PACKAGE which contains a CONSTANT definition

```
CONSTANT  BusWidth : integer := 32;
```

and a FUNCTION definition

```
FUNCTION neg_edge(signal s: std_logic) return BOOLEAN;
```

- Implement the neg_edge FUNCTION in the PACKAGE BODY section

```
FUNCTION neg_edge(signal s: std_logic) return BOOLEAN IS
BEGIN
          RETURN (s'event AND s='0');
END neg_edge;
```

- In the main (top) module instantiate a flip-flop (DFF) by using the above FUNCTION.
- Create a new library "my_registers" (empty) and add a VHDL module "reg8.vhd", which contains an 8 bit register (DFF). The size of the register should be parametrized by a GENERIC statement similar to

```
entity reg8 is
           GENERIC (Reg_Size : integer := 8);
    Port ( clk : in  STD_LOGIC;
           clr : in std_logic;
            din : in  STD_LOGIC_VECTOR (Reg_Size-1 downto 0);
            qout : out  STD_LOGIC_VECTOR (Reg_Size-1 downto 0));
end reg8;
```

- Insert into the library "my_registers" a PACKAGE which contains the COMPONENT declaration of "reg8".
- Instantiate the register "reg8" but change the GENERIC register size to 10 bits similar to

```
my_reg1: reg8 GENERIC MAP (Reg_Size => 10)
          PORT MAP(clk => clk, ...
```

- Verify the structural approach by the "Check Syntax" command.

Lab #06:

4.6 Microprocessor Opcode

For an embedded microprocessor is an initial assembler program required. The simplified microprocessor has a 4 bit instruction word. The command is coded as 2 bits; the operand (register) consists also of 2 bits (4 registers s0–s3).

Instructions:

LOAD	=	“00”
STORE	=	“01”
ADD	=	“10”
SUB	=	“11”

Registers:

s0	=	“00”
s1	=	“01”
s2	=	“10”
s3	=	“11”

All instructions are compatible with all registers (symmetric instruction set). For instance

```
LOAD    s0
STORE   s2
ADD     s3
SUB     s1
STORE   s0
```

are all legal instructions. The assembler instruction “store s2” is thus coded as “0110”.

- Start a new project with name “mcprom” and create a new top level VHDL module with the following content:

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

library my_microproc;
use my_microproc.opcode.all;


entity prom is
    Port ( adr : in  integer RANGE 0 to 3;
           data : out  STD_LOGIC_VECTOR (3 downto 0));
end prom;
```

```
architecture Behavioral of prom is

type register_file is array(0 to 3) of std_logic_vector(3 downto 0);
signal  regfile : register_file;

begin
          regfile(0) <= AluCode(load, s0);
          regfile(1) <= AluCode(add, s2);
          regfile(2) <= AluCode(store, s1);
          regfile(3) <= (others => '0');

          data <= regfile(adr);
end Behavioral;
```

- Create a new library "my_microproc" with a new package "opcode". This package should contain all declarations and the FUNCTION "AluCode" to program initial assembly program.

Lab #07:

4.7 Shift Register Library and Package

Libraries are very useful for efficient code development and for writing reusable code. This is illustrated by a simple library for shift registers.

- Create a new project "rotshift".
- Create a top level VHDL module with the same name "rotshift.vhd" that uses a loadable shift register which can shift left and shift right:

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

library rotshiftlib;
use rotshiftlib.rsutils.ALL;

entity rotshift is
    Port ( clk : in  STD_LOGIC;
           load : STD_LOGIC;
           sleft : STD_LOGIC;
           din : in  srvector;
           dout : out  srvector);
end rotshift;

architecture Behavioral of rotshift is

begin
          sreg_0: srreg PORT MAP (
                    clk => clk,
                    load => load,
                    mode => sleft,
                    d_in => din,
                    d_out => dout);
end Behavioral;
```

As you see there is not implementation code required; everything is contained in the library "rotshiftlib".

The shift register should operate as follows:

clk	clock signal for shift and load operations
load	'0' = shift / rotate operations
	'1' = load "d_in" into shift register
mode	'0' = shift left
	'1' = shift right

- Create a new library “rotshiftlib”
- Add a package with the name “rsutils.vhd” to this library. The package header should look like this:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package rsutils is

          CONSTANT  REG_SIZE : integer := 8;

          SUBTYPE  srvector IS std_logic_vector(REG_SIZE-1 downto 0);

          COMPONENT srreg IS
                    PORT ( clk : in std_logic;
                           load : in std_logic;
                           mode : in std_logic;
                           d_in : in srvector;
                           d_out : out srvector);
          END COMPONENT;

          FUNCTION  shiftleft(signal xv : srvector) RETURN srvector;
          FUNCTION  shiftright(signal xv : srvector) RETURN srvector;

end rsutils;
```

- Implement the package body which contain the function implementations.
- Create a new VHDL module inside the library “rotshiftlib”. The name should be “srreg.vhd”. Implement the shift register logic which is required in the lop level file.
- Simulate the top level module “rotshift.vhd”. The result should look similar to the following diagram.

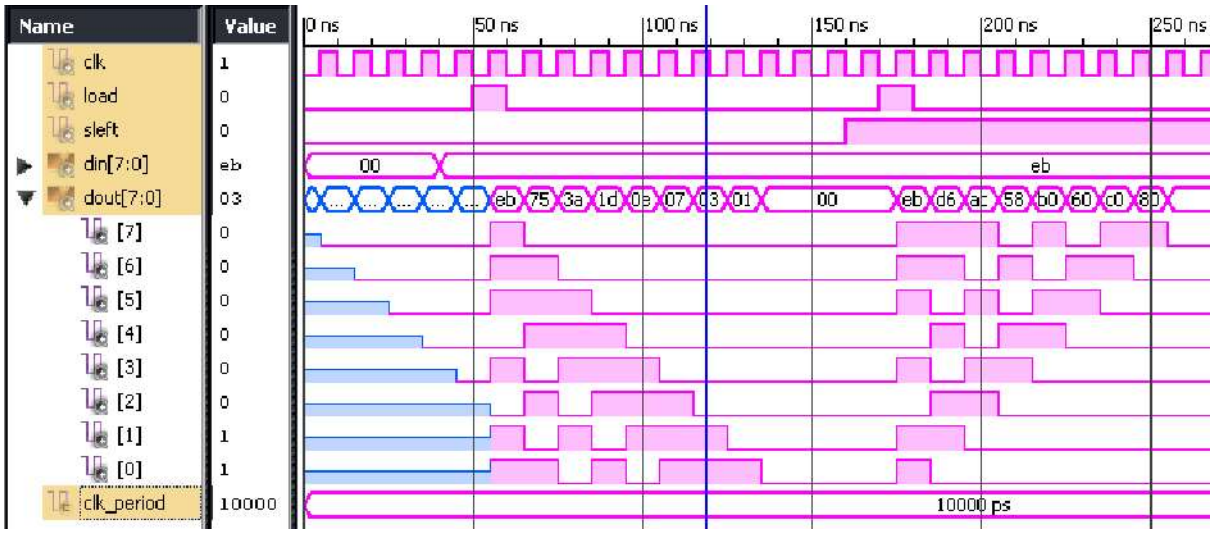


Figure 1.17: Shift register simulation

5 Advanced Simulation of Digital Systems

Simulating a design is recommended to ensure proper operation. If a system is safety-critical simulation is mandatory. Simulation takes place in all stages of design:

- **Functional test**
  The simulation should verify the functional operation of a design. If the simulation fails the other simulations make no sense. This simulation stage is device independent. Any simulator can be used.
- **Post-synthesis function simulation**
  This is the simulation after the design has been mapped into the resources of an FPGA or CPLD. It can fail it timing conditions are violated. If it fails the design requires modifications or requires another (probably faster) device.
- **Post-fitting functional simulation**
  After all previous simulations, and if the design was successfully fitted into the logical device the post-fitting simulation can verify the operation of the final device. The real device may perform better than the simulation but this is, of course, not guaranteed.

For synthesis a design usually is a top level VHDL module. In simulation mode the design becomes UUT (unit under test) or DUT (device under test). A module becomes a UUT with a VHDL wrapper. The simulation file then replaces the top level design file.

If for example the design file has the entity

```
entity bgtop is
    Port ( switch3 : in  STD_LOGIC_VECTOR (2 downto 0);
           led : out  STD_LOGIC_VECTOR (7 downto 0));
end bgtop;
```

the simulation file (new top level) instantiates

```
COMPONENT bgtop
PORT(switch3 : IN  std_logic_vector(2 downto 0);
      led : OUT  std_logic_vector(7 downto 0));
END COMPONENT;
```

The entity section of the simulation top file is empty, since the simulation module creates all signals (stimuli) using SIGNALS and/or VARIABLES .

In VHDL simulation files are language elements can be used. It is not required that the statements are synthesizable (as seen below).

A test signal that satisfies some timing conditions is shown here:

```
xtst <= '1' after 100ns, '0' after 120ns, '1' after 150ns, '0' after 200ns;
```

Alternatively it can be programmed using a process (without sensitivity list):

```
x_tst2p: process
begin
```

```
        wait for 100ns;
        xtst2 <= '1';
        wait for 20ns;
        xtst2 <= '0';
        wait for 30ns;
        xtst2 <= '1';
        wait for 50ns;
        xtst2 <= '0';
        wait;
    end process;
```

The results are exactly the same. Another possibility is the use of a FOR LOOP , which is a compact solution even for complex signals. Again, the result is exactly the same as for the above examples.

```
x_tst3p: process
VARIABLE  tmp : std_logic_vector(0 to 10) := "11000111110";
begin
    wait for 100ns;
    FOR k IN tmp'RANGE LOOP
          xtst3 <= tmp(k);
          wait for 10ns;
    END LOOP;
    wait;
end process;
```

5.1 Simulation Using File Data

Reading and writing to files and standard IO is part of the IEEE library. It is required to include the following line to the test bench module:

```
use std.textio.all;
```

Before read or write becomes possible a FILE descriptor has to be defined. The access type should be set to READ_MODE or WRITE_MODE . Opening a file for reading is achieved by:

```
FILE  f_in  : TEXT OPEN READ_MODE is "my_data.txt";
```

The file descriptor f_in allows read access to file text (ASCII) file “my_data.txt”.

The access of a line of text requires a VARIABLE of the type LINE .

```
VARIABLE  txtline : LINE;
```

A line of text is accessed with the PROCEDURE

```
READLINE(f_in, txtline);
```

Reading and writing is based on the LINE data type. This data type represents a line of text which is limited by an end-of-line character (usually CR, LF or CR-LF, depending of the operating systems). Once a line has been read, all data can read from the LINE variable with the PROCEDURE READ :

```
READ(txtline, var1);  READ(txtline, var2),  READ(txtline, var3); ...
```

The LINE variable `txtline` will be destroyed by the PROCEDURE `READ`.

Although not defined in the IEEE library all simulators have the BOOLEAN FUNCTION `ENDFILE(<filevar>)` defined. A fragment to read a textfile might look like this:

```
library IEEE;
...
use std.textio.all;


FILE  f_in : TEXT OPEN READ_MODE is "my_file.txt";
...
VARIABLE  li : LINE;
VARIABLE  x1, x2, x3 : INTEGER RANGE 0 to 31;
...
WHILE not ENDFILE(f_in) LOOP
    READLINE(f_in, li);
    READ(li, x1);  READ(li, x2);  READ(li, x3);
    ...
    portx <= std_logic_vector(to_unsigned(var2));
    ...
END LOOP;
-- the file was processed
```

Writing to a file is similar:

```
library IEEE;
...
use std.textio.all;


FILE  f_out : TEXT OPEN WRITE_MODE is "logfile.txt";
...
VARIABLE  li : LINE;
VARIABLE  var1 : INTEGER RANGE 0 to 31;
...
WRITE(li, "some text ");  WRITE(li, var1);
WRITELINE(f_out, li);
```

When writing to the simulator console (standard output) the special file name `"STD_OUTPUT"` has been defined:

```
FILE  f_stdout : TEXT OPEN WRITE_MODE is "STD_OUTPUT";
...
VARIABLE  li : LINE;
...
WRITE(li, "Hello, World! ");
WRITELINE(f_stdout, li);
```

Of course, all text IO make only sense when used in conjunction with a simulator.

Lab #08:

5.2 Simulation with Stimulus Data from File

For large variations of stimulus data a file based simulation is recommended. A "bargraph" application should be tested by a input data at variable times.

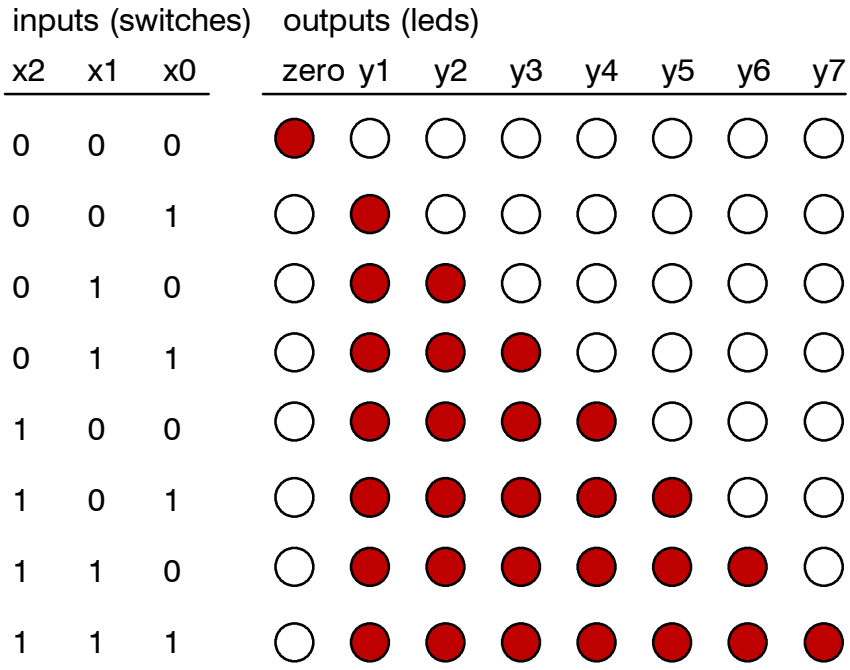


Figure 1.18: bargraph logic

- Create a new project "bargraph" and a new top level main code VHDL module.
- Code the bargraph logic as concurrent code using "SELECT". Fully code all input combinations to avoid "unwanted latches".
- Create a VHDL test bench file. This test bench file reads the text file "bgsim.txt" which contains stimulus data:

```
-- bgsim.txt: simulation data for bargraph design
--
-- time  switch_value
   20ns  3
   20ns  5
   50ns  2
   60ns  1
   30ns  7
```

```
   20ns  6
   20ns  5
   40ns  4
   50ns  3
   30ns  2
   20ns  1
   20ns  0
   30ns  4
  100ns  0
-- End of file
```

The first column is the duration time (compatible with the VHDL TIME data type) and the second column consists of the data (integer representation of the three switches). If a line starts with "--" it must be ignored, since the whole line is a comment.

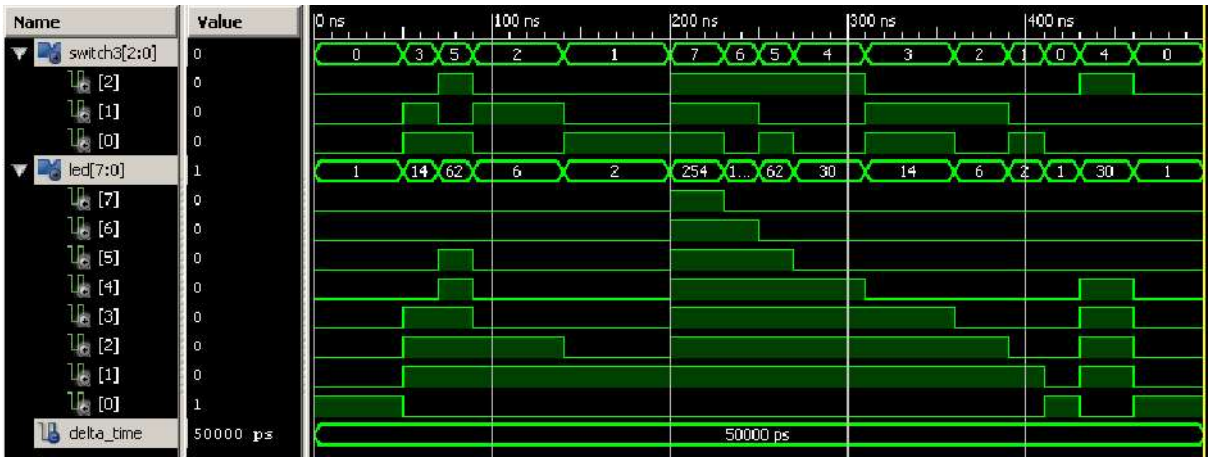


Figure 1.19: Simulation of the bargraph logic with input data from "bgsim.txt"

- Verify proper operation by the graphical output (like fig. 1.19).
- Write the simulation input data to standard output to verify that the input file is properly read. The output might look like this:

```
Simulator is doing circuit initialization process.
Finished circuit initialization process.
1     20 ns     3
2     20 ns     5
3     50 ns     2
4     60 ns     1
5     30 ns     7
6     20 ns     6
7     20 ns     5
8     40 ns     4
9     50 ns     3
10     30 ns     2
11     20 ns     1
12     20 ns     0
```

```
13    30 ns    4
14    100 ns    0
ISim>
```

***

6 Design of Finite State Machines (FSM)

Finite state machines are the base of every practical digital system. It is a characteristic of a FSM that it contains memory to store the state. While this can be any type of flip-flop (DFF, JKFF, TFF etc.), in most cases DFFs are used.

Even the most complex digital system can be regarded as a collection of state machines. A microprocessor, for instance, is a state machine that takes opcodes (the instructions) for input.

6.1 FSM Architectures

All state machines have a state memory (FFs), an input logic (F) and an output logic (G). Two variants exist, the MOORE and the MEALY machine. They differ only in the output logic G.

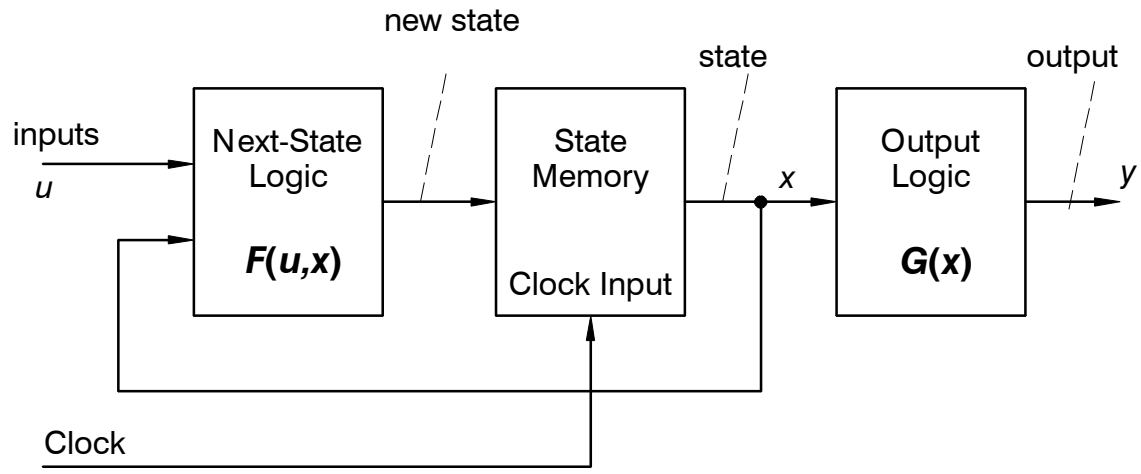


Figure 1.20: Moore machine

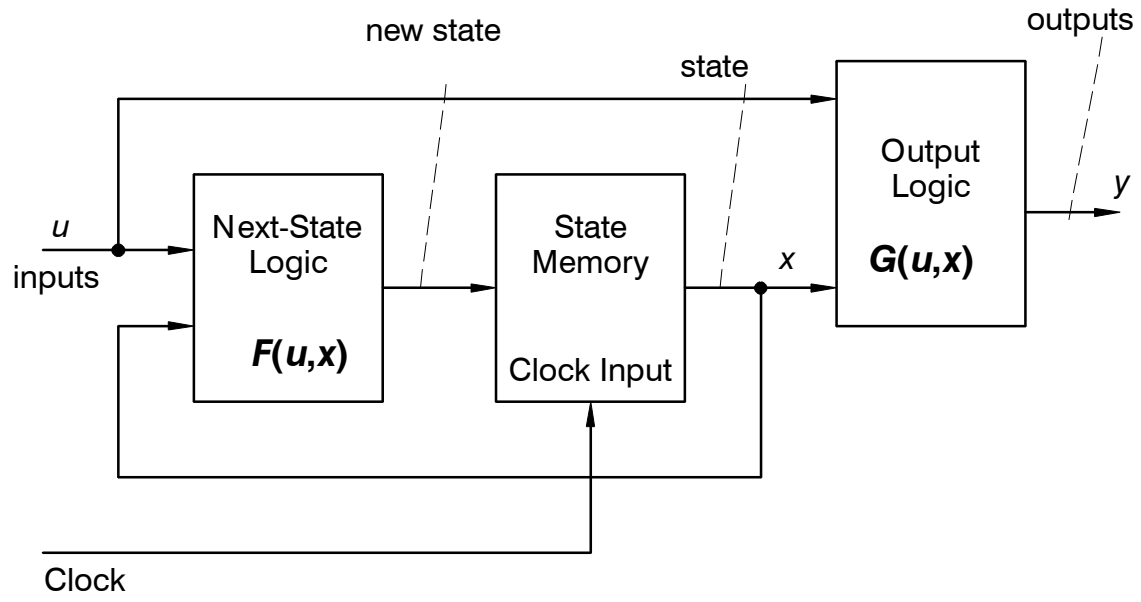


Figure 1.21: Mealy machine

Both designs may cause *glitches* (hazards) due to physicals delays in the output logic. This avoids a pipelined design as shown in fig. 1.22. Of course, the pipelined design is "slower" (on clock cycle) and consumes more flip-flops. It depends on the application with design can be used.

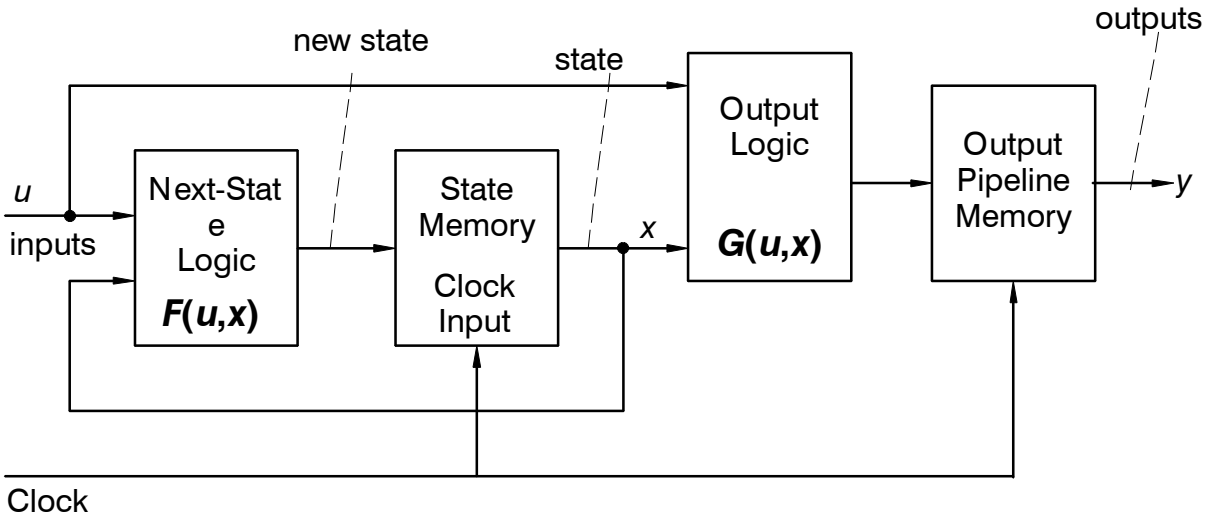


Figure 1.22: Mealy machine with pipelined outputs (glitch-free version of fig. 1.21)

6.1.1 Hazard and Hazard Avoidance

Hazards are wrong output caused by the change of input signals. Static-0 and static-1 hazards denote false ouputs for a signal that should be '0' or '1' respectively.

A static-0 Hazard creates a ’1’, although it should stay ’0’.

A static-1 Hazard creates a ’0’, although it should stay ’1’.

Hazards occur due to different propagation delays in combinational logic. Thus, they are also called timing hazards. The following example illustrates a hazard occurrence.

x3	x2	x1	y	
0	0	0	0	
0	0	1	1	
0	1	0	0	
0	1	1	0	
1	0	0	0	
1	0	1	1	•
1	1	0	1	
1	1	1	1	•

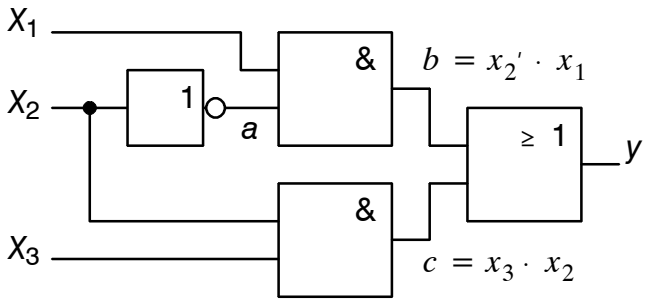


Figure 1.23: Hazard occurrence

According to the 6th and the 8th row in the truth table (marked) the output y should stay equal to ’1’ independent of the signal x_2. A detailed analysis with constant propagation delays for all gates reveal the existence of a static-1 hazard.

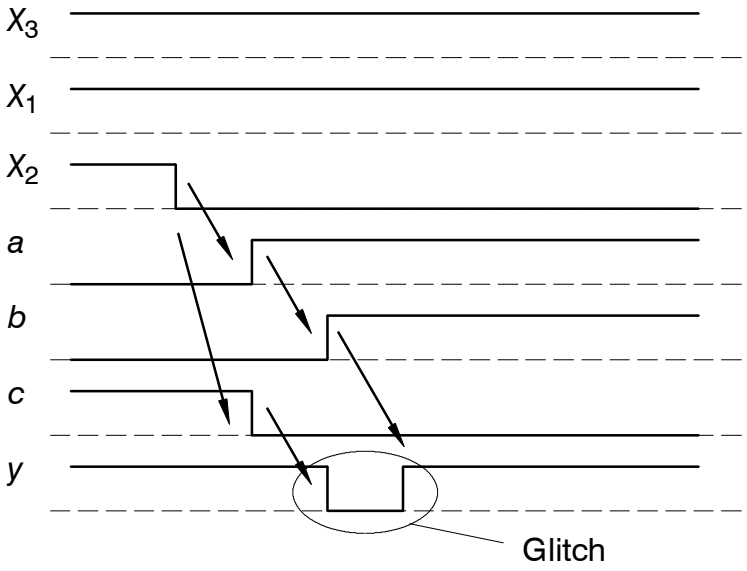


Figure 1.24: Hazard analysis

The Karnaugh map in fig. 1.25 show the minterms (solid lines) which correspond to the circuit’s gates.

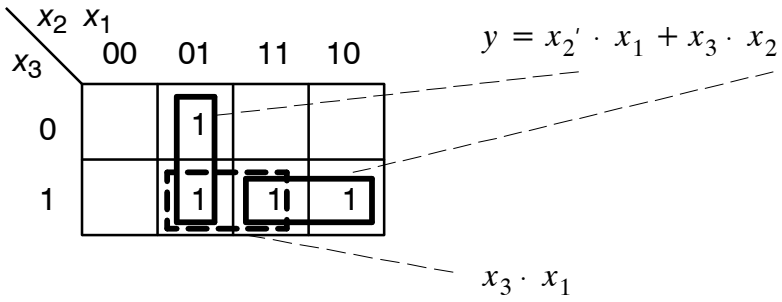


Figure 1.25: Karnaugh map of truth table in fig. 1.23

The hazard happens when x_2: changes $1 \rightarrow 0$. When using the additional minterm (dashed lines) no hazard can occur. However, the realization is not minimal.

The boolean expression then becomes

$$y = x_2' \cdot x_1 + x_3 \cdot x_2 + x_3 \cdot x_1 . \tag{1.1}$$

Fig. 1.26 shows the glitch-free realization.

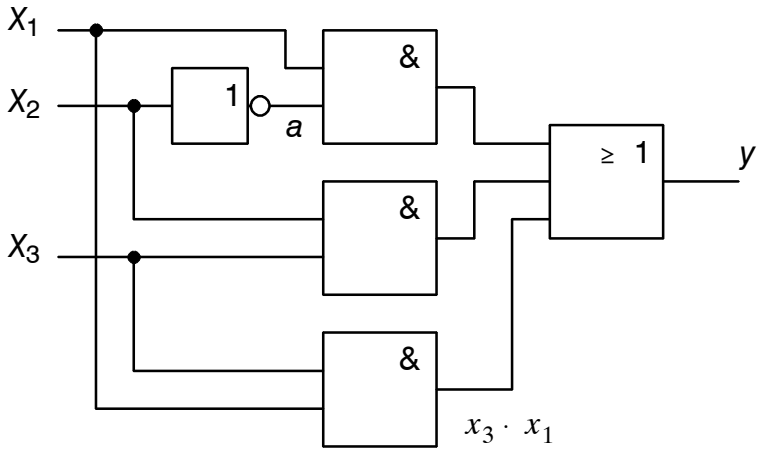


Figure 1.26: Static-1 hazard-free circuit

6.1.2 State Encoding

With n DFFs we can code $N = 2^n$ states. On the other hand, if the number of states N are known the number of required DFFs is

$$n = \text{ceil}(\text{ld}\, N) = \text{ceil}\left(\frac{\log N}{\log 2}\right) . \tag{1.2}$$

Here ld is the dual logarithm and ceil means rounding towards the next higher integer. For example if $n = 22$ states are required, the number of DFFs becomes

$$n = \text{ceil}(\text{ld}\, 22) = \text{ceil}\left(\frac{\log 22}{\log 2}\right) = \text{ceil}\left(\frac{1.3423}{0.301}\right) = \text{ceil}(4.459) = 5 . \tag{1.3}$$

With $n = 5$ we can code 32 states which is, of course, more than the required 22 states.

The states can be encoded in an arbitrary code (binary, gray etc.). The type of encoding has an impact on the complexity of the required logic. For small FSMs the **one-hot-encoding** scheme leads to the overall simplest solution.

In **one-hot-encoding** every state has it's own DFF (n = N).

This means that the required number of DFFs is identical to the number of states. It is a good idea to let the design software choose the type of encoding, when the state machine is described in a behavioral manner.

6.2 FSM Templates in VHDL (Moore Machine)

The design of the Mealy machine is very similar since it differs only on the output logic $G(x)$ or $G(x, u)$, respectively. The following VHDL template is a starting point for building FSMs. It is an example not a technical regulation.

```
----------------------------------------------------------------------------
-- Moore machine template
-- Engineer:       Kai Mueller
--
-- Create Date:    20:19:17 04/13/2011
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mooret is
    Port(clk : in STD_LOGIC;
         mreset : in  STD_LOGIC;
         my_ouputs : out  STD_LOGIC_VECTOR (3 downto 0));
end mooret;

architecture Behavioral of mooret is
   TYPE my_state_type is (st_0, st_1, st_2, ..., st_err);
   SIGNAL state, next_state : my_state_type;
   -- state machine outputs as local signals (if needed)
   signal a, b, c : std_logic;

begin
   -- state memory update (state machine FFs)
   -- let the compiler select the number of required FFs
   UPD_PROC: process (clk, tlreset)
   begin
      if rising_edge(clk) then
         if (mreset = '1') then
            state <= st_x;
```

```
         else
            state <= next_state;
         end if;
      end if;
   end process;

   -- input logic F(x)
   FX_PROC: process (state)
   begin
      --declare default state for next_state to avoid latches
      next_state <= state;  -- possible in a sequential evironment
      CASE (state) IS
         WHEN st_st0 =>     next_state <= st_1;
         WHEN st_1 =>
            if (<condition>) then
          next_state <= st_2;
            end if;
         WHEN st_2 =>    ...
         WHEN others =>     next_state <= st_0;
      END CASE;
   end process;

   -- G(x) Output logic as a process (not required but possible)
   GX_PROC: process (state)
   begin
      a <= '0';  b <= '0';  c <= '0';
      CASE (state) IS
         WHEN st_0 =>
          a <= '1';
         WHEN st_1 =>
          a <= '1';  c <= '1';
         WHEN ...
         WHEN others =>
          a <= '-';  b <= '-';  c <= '-';
         END CASE;
   end process;

   -- more outputs (instead of G(x) process or additional assignments)
   WITH state SELECT
      my_outp <= "00" WHEN st_0,
          "01" WHEN st_1,
          "10" WHEN st_2,
          "11" WHEN st_3,
          "--" WHEN others;  -- This is good practice!
end Behavioral;
```

Lab #09:

6.3 Three Bit Down Counter

Before creating state machines in programmable logic a state machine should be build in hardware. Every clock cycle decrements a 3 bit counter in the following way:

```
000 - 111 - 110 - 101 - 100 - 011 - 010 - 001 - 000 - 111 - ...
```

The state diagram shows the operation.

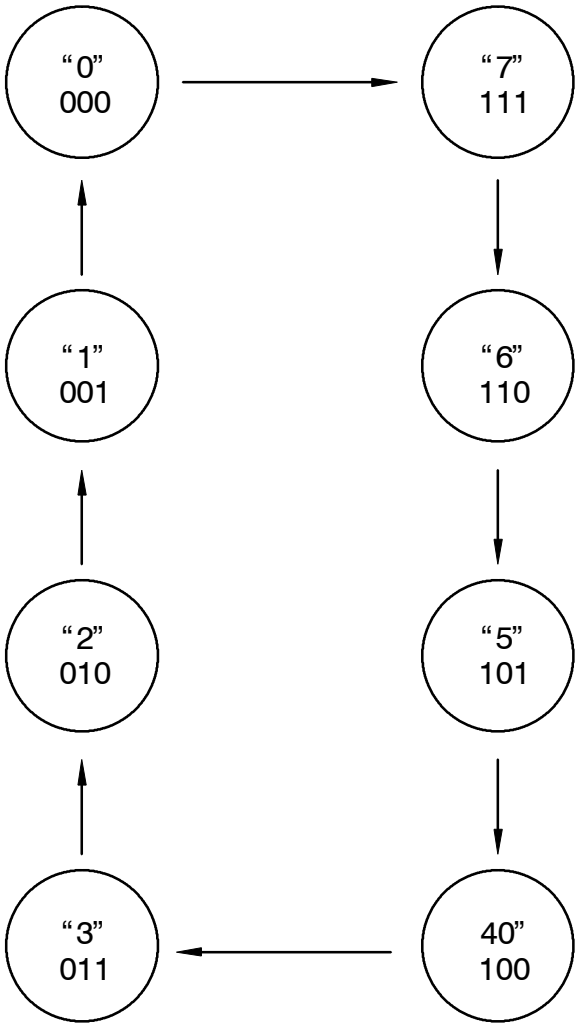


Figure 1.27: State diagram for a 3 bit down counter

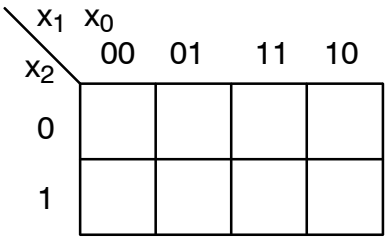
No output logic is required since the state encoding is identical to the outputs. Therefore only the next state logic $f(x)$ is required. Note, that the counter has no input.

- Assume that the states are decoded as a binary number $x_2\ x_1\ x_0$.
  Write down the truth table

x2	x1	x0	x2n	x1n	x0n
0	0	0			
0	0	1			
...					

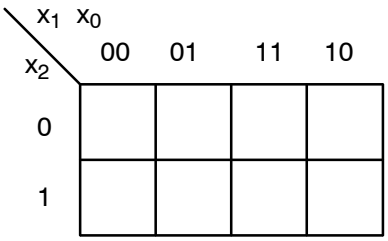
  where $x_{2n}\ x_{1n}\ x_{0n}$ are the state variables for the next state.

- Find minimal solutions for canonical sums or canonical products (whatever is the simpler solution). You can use the following Karnaugh map templates.



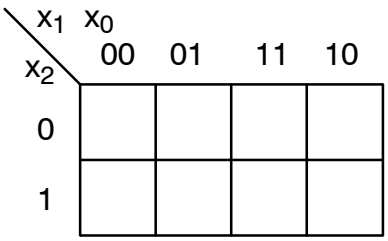
$x_{2n} =$

Figure 1.28: Karnaugh map for x_{2n}



$x_{1n} =$

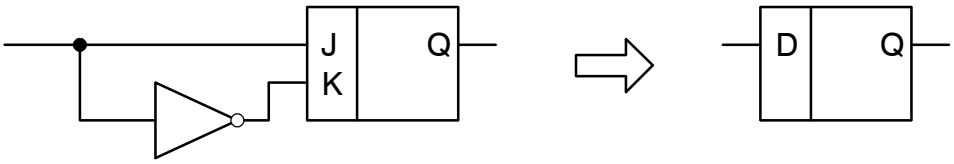
Figure 1.29: Karnaugh map for x_{1n}



$x_{0n} =$

Figure 1.30: Karnaugh map for x_{0n}

- Draw the complete circuit diagram.
- Make the required connections *in hardware*.
  NOTE: D-FFs are not available in lab hardware. You can convert the JK-FF into a D-FF is shown below:



***

Lab #10:

6.4 Traffic Light Controller (Basic)

A traffic light controller should generate a standard cycle for one road direction.

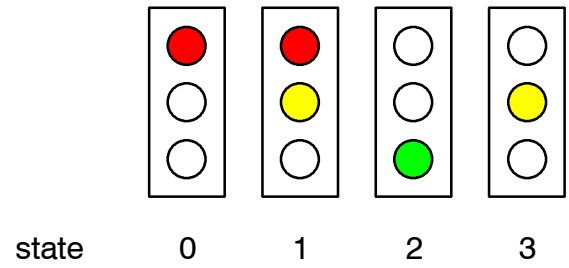


Figure 1.31: Traffic light cycle

- Create a new project "tflight1" with a VHDL main code module. The Entity section should be similar to

```
entity tlmain is
    Port(clk : in STD_LOGIC;
         tlreset : in  STD_LOGIC;
         led : out  STD_LOGIC_VECTOR (7 downto 0));
end tlmain;
```

clk is the CLOCK source,
tlreset should reset the state machine into the first state (red light)
led holds the traffic lights (led(7)= red, led(6)= yellow, led(5)= green.

- Define all states in a TYPE instruction (inside your ARCHITECTURE)
- In order to keep the design simple the states should change with every rising edge of the CLOCK signal.
- Code the "update" process as well as input logic F(x) and output logic G(X). The output logic can be a process or concurrent code (possible since hazards are not a problem here).
- Simulate the design.

***

Lab #11:

6.5 Traffic Light Controller (with Timer, see Chapter 6.9)

A practical implementation requires a timer to extend the states of the traffic light controller.

- Create a new project "tflight2" and create a main code VHDL module.
- Create a loadable down-counter as a timer (yes, this is another state machine!). If the counter reaches zero a signal should indicate this. Provide a 4-bit – counter.
- The number of states from lab #08 have to be doubled. We need one state to load the timer with a value. Another state waits until the timer has expired (i.e. it reached zero) in order to advance to the next state.
- Simulate the design.
- If required load the design into the Spartan6 board.

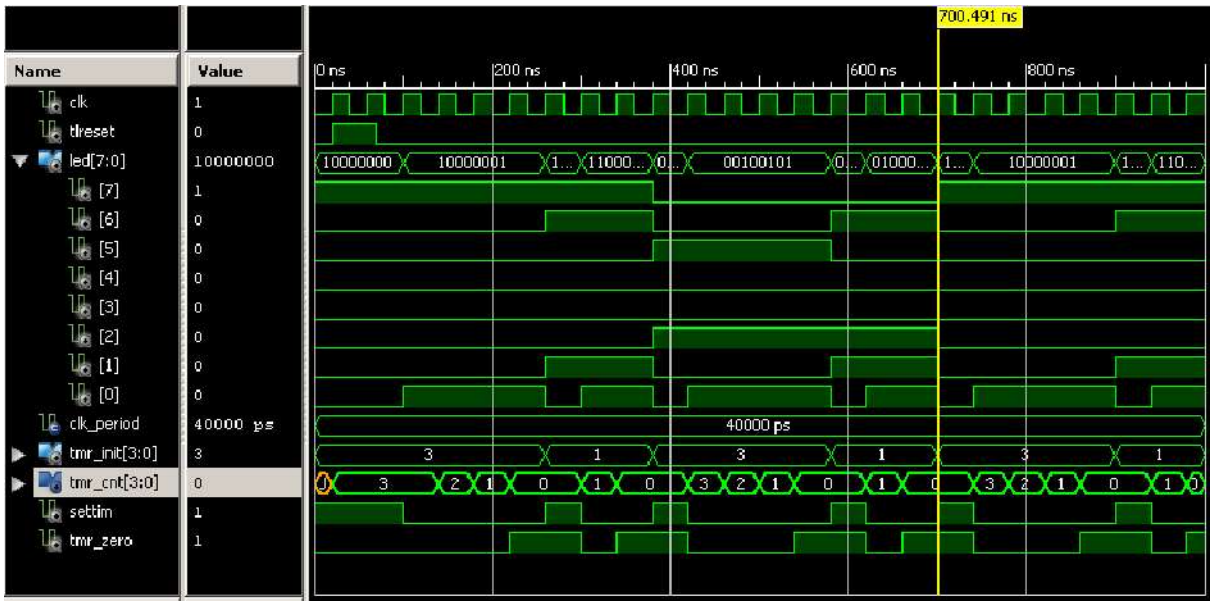


Figure 1.32: Traffic light controller with timer (led(7)= red, led(6)= yellow, led(5)= green)

***

6.6 State (Transition) Diagrams

Every design of a state machine should be preceded by drawing a state diagram. Several software tools support the code generation from the graphical drawing of the diagram (like Matlab's Stateflow™). Xilinx has removed it's StateCAD™ tool since ISE version 11.x without further explanations. The code created from StateCAD was probably inefficient.

Here is an example from a StateCAD generated VHDL code of a traffic light controller:

```
PROCESS (AMBER,GREEN,RED,REDAMB,RESET,TIMER0,TIMER1,TIMER2,TIMER3)
BEGIN

    IF (( TIMER0='1' AND RESET='0' AND  (AMBER='1')) OR ( TIMER1='1' AND
          RESET='0' AND  (AMBER='1')) OR ( TIMER2='1' AND RESET='0' AND
          (AMBER='1')) OR ( TIMER3='1' AND RESET='0' AND  (AMBER='1'))
          OR ( RESET='0' AND TIMER0='1' AND TIMER1='1' AND TIMER2='0'
          AND TIMER3='0' AND  (GREEN='1'))) THEN
              next_AMBER<='1';
          ELSE next_AMBER<='0';
              END IF;
```

An efficient implementation of state machines can easily be carried out by VHDL coding of state diagrams. Below is the state diagram of a traffic light sequencer. Every state has its unique name (in this case "A", "B", "C", and "D"). These states require unique encoding (one-hot, binary, gray, or integer for example). The best encoding method depends on the complexity of the state machine and on the available resources of the FPGA used.

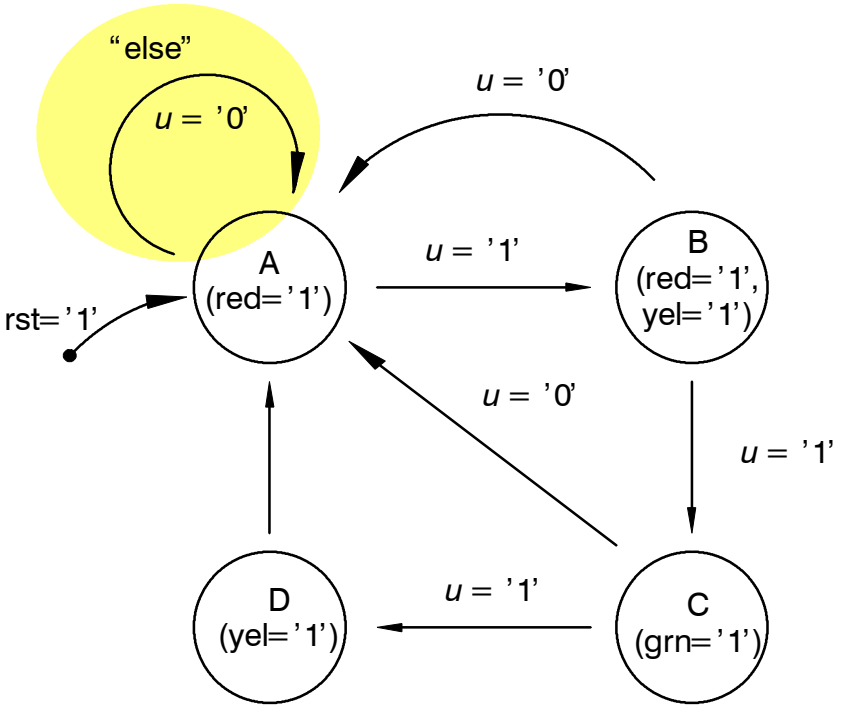


Figure 1.33: Complete state diagram of the simplified traffic light controller

Inside of the state “bubbles” the active outputs are write in brackets (). The conditions for the change of a state are written beside the arrowhead lines. The conditions must be mutually exclusive, otherwise the state machine cannot be implemented. In the above example the state “A” contains an “ELSE” clause, which indicates that this state is maintained whenever the signal *u* is ’0’. Since this is always implicit it can (and should) be omitted. The resulting state diagram is shown in fig. 1.34.

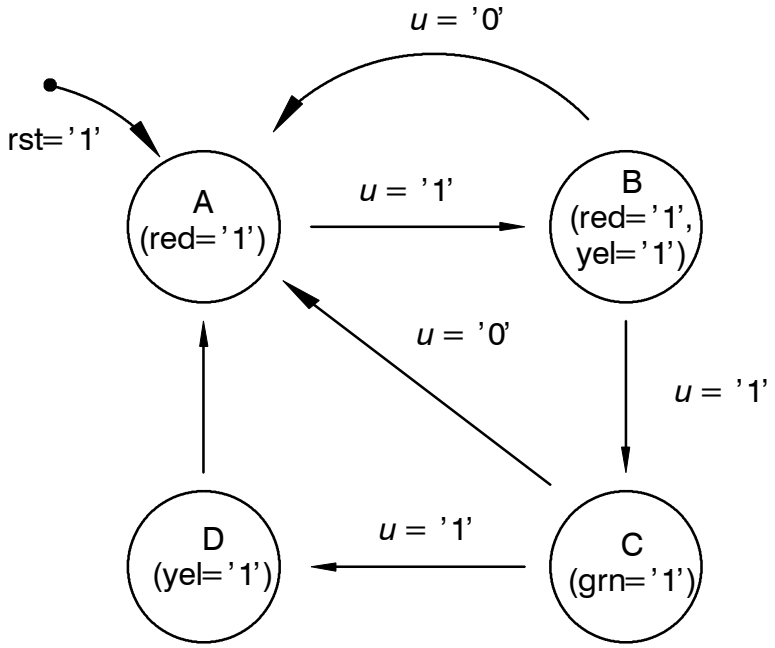


Figure 1.34: Complete state diagram of the simplified traffic light controller

The leftmost transition with “rst=’1’” has a special meaning: it resets the state machine (synchronously or asynchronously) into state “A”. The transition form “D” to “A” has no condition. Therefore, it is taken by the next clock cycle independent from the signal *u*.

6.7 State Machines of State Machines

State machines should be kept as simple as possible. Instead of creating one complex state machine a “sub state machine” could be used. If a “timer” is needed for the condition to take a transition the following state machine acts a delay that waits for *tmr_init* clocks:

```
-- down counter as timer
tmr_zero <= '1' WHEN tmr_cnt = "0000" ELSE
            '0' WHEN tmr_cnt /= "0000" ELSE
             -';

DWNC_PROC: process (clk, settim)
    begin
          if rising_edge(clk) then
              if settim='1' then
```

```
            tmr_cnt <= tmr_init;
          elsif tmr_cnt /= "0000" then
            tmr_cnt <= tmr_cnt - 1;
          end if;
      end if;
  end process;
```

The signal *tmr_zero* can be used in the transition from a state. Of course in a state before the time has to started with the signal *settim*.

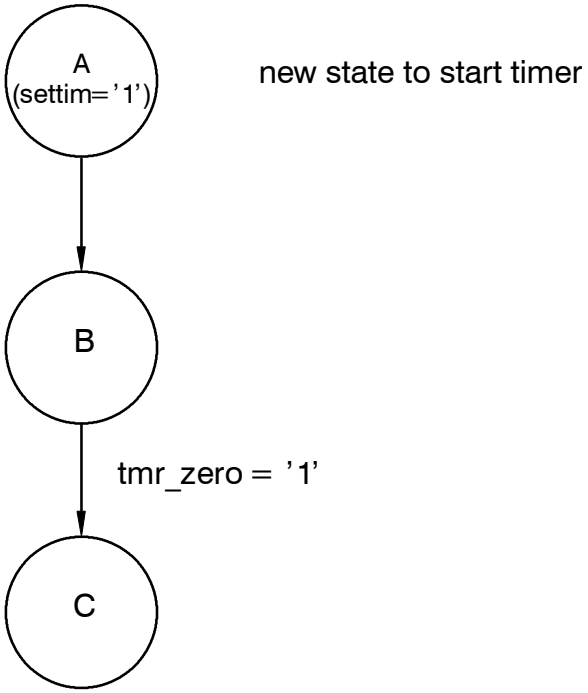


Figure 1.35: Using a sub state machine to implement a “timer”

6.8 Efficient Programming of Small/Mid State Machines

In many cases an efficient programming model differs from the standard model from fig. 1.20 or fig. 1.21. This is the case whenever the input (next-state) logic and the output logic requires similar or redundant logic.

```
-- input logic F(x)
FX_PROC: process (state, u)
begin
   next_state <= state;  -- possible in a sequential evironment
   CASE (state) IS
      WHEN st_st0 =>
        if (condition) then
            next_state <= state_x;
        end if;
```

```
      WHEN st_st1 =>
        if (condition) then
            next_state <= state_y
        end if;
      .
      .
      .
      WHEN others =>     next_state <= st_0;
   END CASE;
end process;

-- G(x) Output logic as a process
GX_PROC: process (state, u)
begin
   a <= '0';  b <= '0';  c <= '0';
   CASE (state) IS
      WHEN st_0 =>
       a <= '1';
      WHEN st_1 =>
       a <= '1';  c <= '1';
      WHEN ...
      WHEN others =>
       a <= '-';  b <= '-';  c <= '-';
      END CASE;
end process;
```

$F(x)$ and $G(x)$ repeat almost the same logic which lead in a poor design. A different structure lead in these cases to a better design.

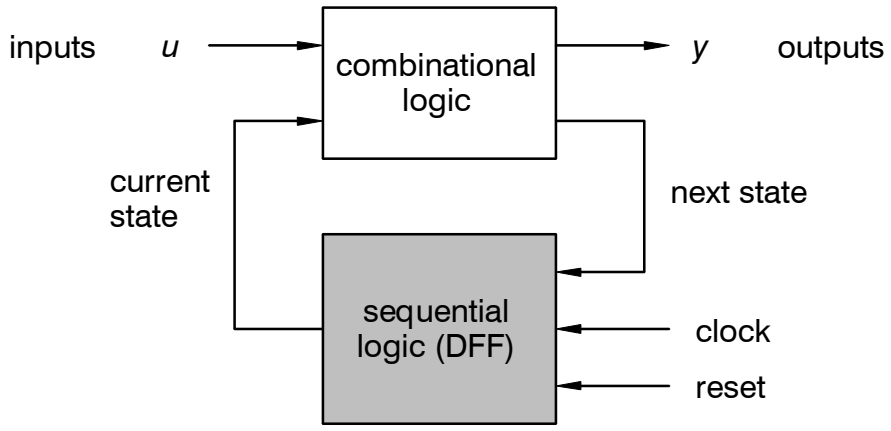


Figure 1.36: State machine with combinational and sequential logic blocks

The combinational block (upper block) comprises input and output logic. This structure has a clear separation of combinational and sequential logic. Both blocks are usually implemented as processes. The above example could be coded as follows (better!):

```
 -- sequential block (lower block)
SEQ_BLK: process (clk, sreset)
```

```
begin
   if rising_edge(clk) then
      if (sreset = '1') then
         state <= st_0;
      else
         state <= next_state;
      end if;
   end if;
end process;

-- combinational block (upper block)
COM_BLK: process (state, u, ...)
begin
   next_state <= state;  -- possible in a sequential evironment
   CASE (state) IS
      WHEN st_st0 =>
        if (condition) then
            next_state <= state_x;
        end if;
        a <= '1';
      WHEN st_st1 =>
        if (condition) then
            next_state <= state_y
        end if;
        a <= '1';  c <= '1';
      .
      .
      .
      WHEN others =>
        next_state <= st_0;
        a <= '-';  b <= '-';  c <= '-';
   END CASE;
end process;
```

6.9 Timed State Machines

In many cases timer or counter conditions are an issue. In these cases the timer functionality can be build directly into the sequential block of the state machine. The timer is coded as a counter which counts up until a specified value is achieved or counts down until the counter reaches zero. Resetting the counter and storing of the next state occurs simultaneously. This is the key technique to facilitate the design.

The sequential part might look like:

```
ARCHITECTURE ...
SIGNAL state_type IS (state_A, state_B, ...);
SIGNAL state, next_state : state;
SIGNAL timer : POSITIVE RANGE 0 to 31;
```

```
BEGIN
    -- sequential block
    PROCESS (clk, reset)
    VARIABLE count : POSITIVE RANGE 0 to 31;
    BEGIN
          IF reset='1' THEN
              state <= state_A;
              count := 0;
          ELSIF rising_edge(clk) THEN
              count := count + 1;
              IF count >= timer THEN
                    state <= next_state;
                    count := 0;
              END IF;
          END IF;
    END PROCESS;
```

In this template an up-counter is used. When *timer* is reached, the state update occurs. This differs from the conventional state machine sequential block. At the same time the counter is reset to zero.

> If no timer is required for a transition, the value for *timer* must be set to 1.

Consider the following simple state machine in fig. 1.37.

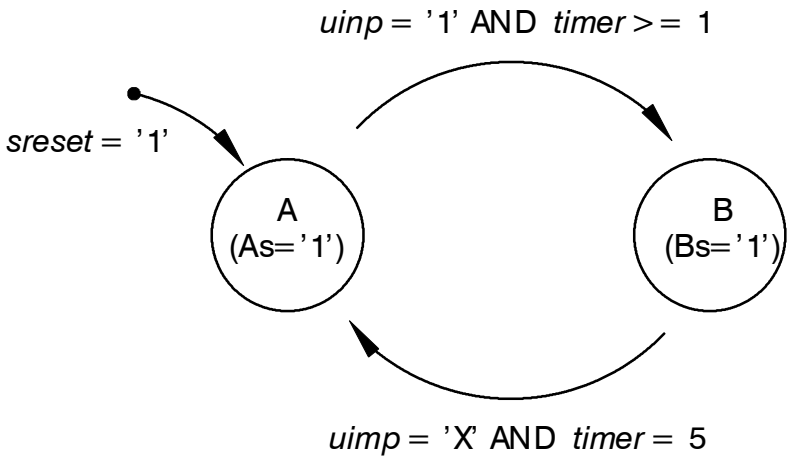


Figure 1.37: Complete state diagram

The upper transition is a pure condition because the *timer* condition has no effect. The lower transition is only a timer condition. The effective state diagram is shown below.

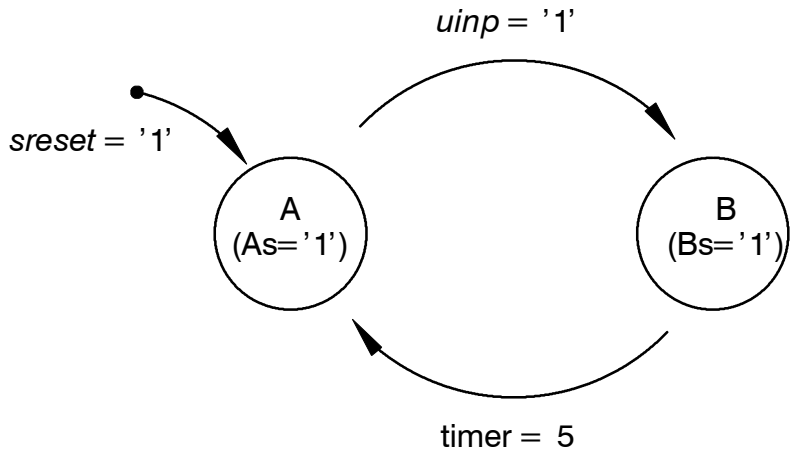


Figure 1.38: Effective state diagram

The corresponding combinational block is shown below.

```
-- combinational logic block
PROCESS (state, uinp)
BEGIN
    As <= '0';  Bs <= '0';
    next_state <= state;
    timer <= 1;
    CASE state IS
          WHEN state_A =>
              IF uinp='1' THEN
                    next_state <= state_B;
              END IF;
              As <= '1';
          WHEN state_B =>
              timer <= 5;              -- override the default
              next_state <= state_A;
              Bs <= '1';
    END CASE;
END PROCESS;
```

State *state_B* is reached immediately after *uinp* goes to '1'. After 5 clock cycles the state machines switches back to *state_A*.

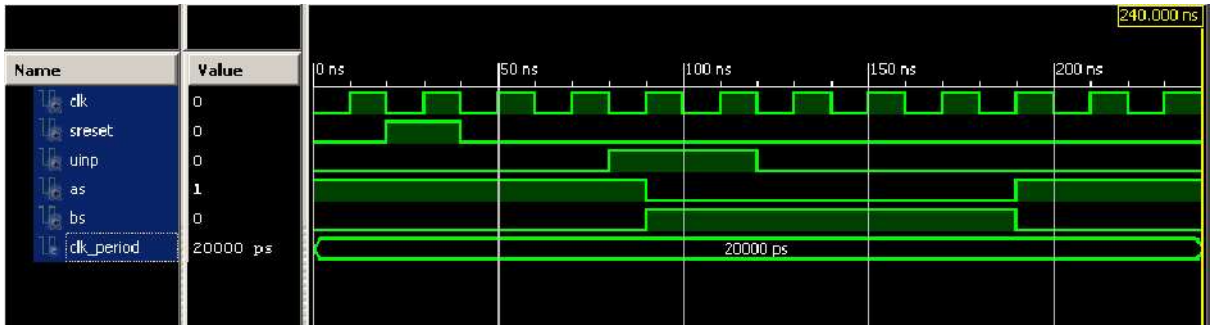


Figure 1.39: Simulation of a timed stated machine

Lab #12: [Mandatory lab assignment]

6.10 Traffic Light Controller with Timed State Machine

The traffic light controller should be programmed using a timed state machine. The design must be synthesizable.

THIS LAB MUST BE PASSED BY SUBMITTING A REPORT BY EVERY INDIVIDUAL – GROUP SOLUTION ARE NOT ACCEPTED.

- Create a new project "tltimsm".
- Provide the following inputs and outputs:

```
Port ( sreset : in  STD_LOGIC;
       clk : in  STD_LOGIC;
       red : out  STD_LOGIC;
       yel : out  STD_LOGIC;
       green : out  STD_LOGIC);
```

- Program the traffic light controller with a timed state machine. The code must contain one sequential process and one combinational process.
  AVOID unwanted latches of unwanted FFs.
- The states should last for the following clock cycles:

red	:	4
red / yellow	:	1
green	:	4
yellow	:	1

- Add the log for the synthesis process to verify that the system is synthesizable. Explain any warnings you receive for the synthesis stage.
- Simulate the design and add the simulation results to the report.

Lab #13:

6.11 Serial Communication Circuit (SPI Transmitter)

The SPI (serial peripheral interface) protocol is a fast and simple protocol for the serial transfer of data. The number of bits to be transferred is unlimited. Many device can share the same serial line. The transfer occurs synchronously to a given serial clock (sclk).

The SPI protocol is widely used to transfer data to and from ADCs and DACs. Some vendors use SPI for programming their devices. While several variants exist a typical transfer (only transmit) is shown in fig. 1.40.

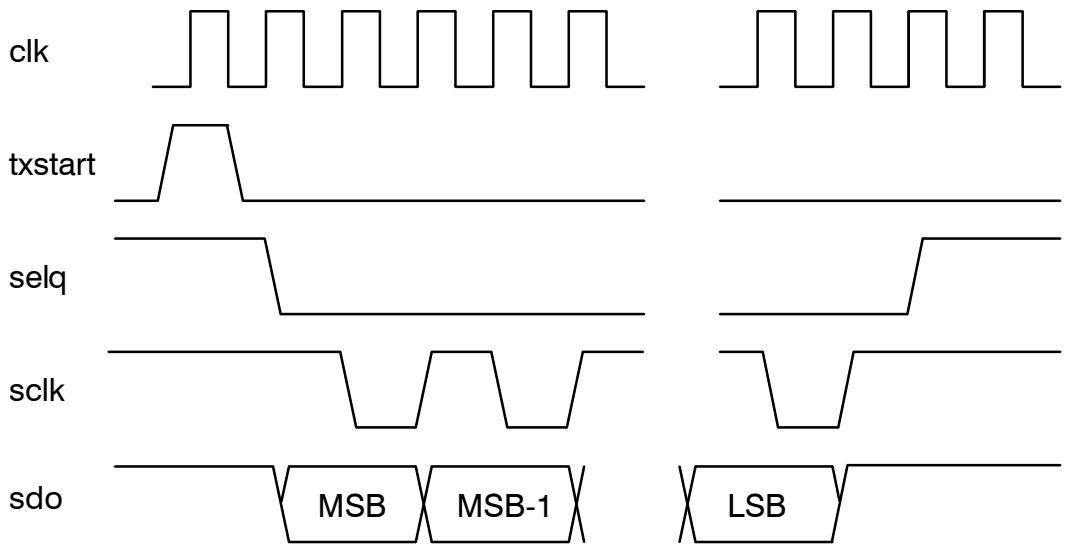
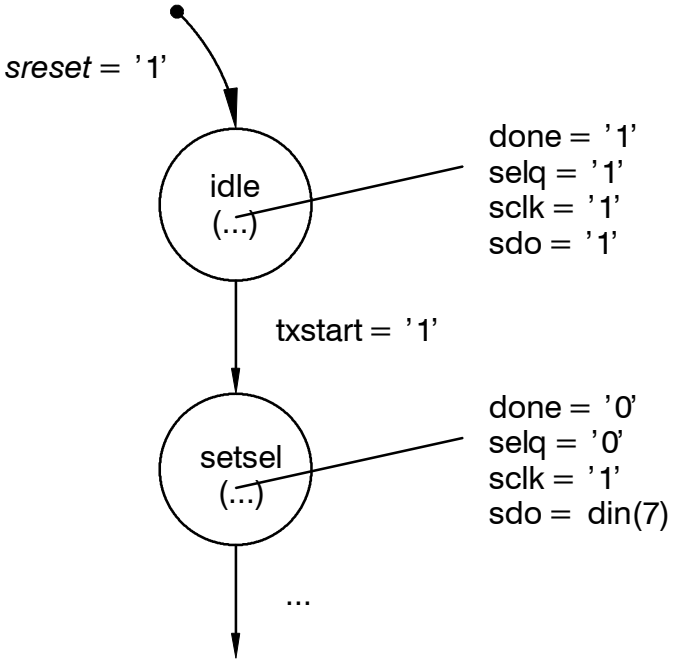


Figure 1.40: SPI signals for transmitting of a word

- Create a new project "spitransmit" and create a main code VHDL module.
- Draw a complete state diagram (omit implicit else clauses). It should start and and in the "idle" state.



- The ENTITY section should contain:

```
entity spitx is
    Port ( clk : in  STD_LOGIC;
           sreset : in  STD_LOGIC;
           din : in  STD_LOGIC_VECTOR (7 downto 0);
           selq : out  STD_LOGIC;
           sclk : out  STD_LOGIC;
           sdo : out  STD_LOGIC;
           txstart : in  STD_LOGIC;
           done : out  STD_LOGIC);
end spitx;
```

Input signal description:

clk	sequential block clock signal
sreset	state machine reset signal (to idle state)
din	data to be sent (one byte)
	☞ The number of bits should be declared as a CONSTANT
txstart	initiate transfer (start signal)

Output signal description:

selq	slave select signal (see diagram)
sclk	serial data clock
sdo	serial data out
done	signal that indicates that all bits have been sent

- A signal of suitable type (or subtype) should contain the current bit number that is transferred.

- Simulate the design by providing the following stimulus data:
  clk
  sreset
  din
  txstart

The following simulation shows the signals involved with the transfer of 8 bits.

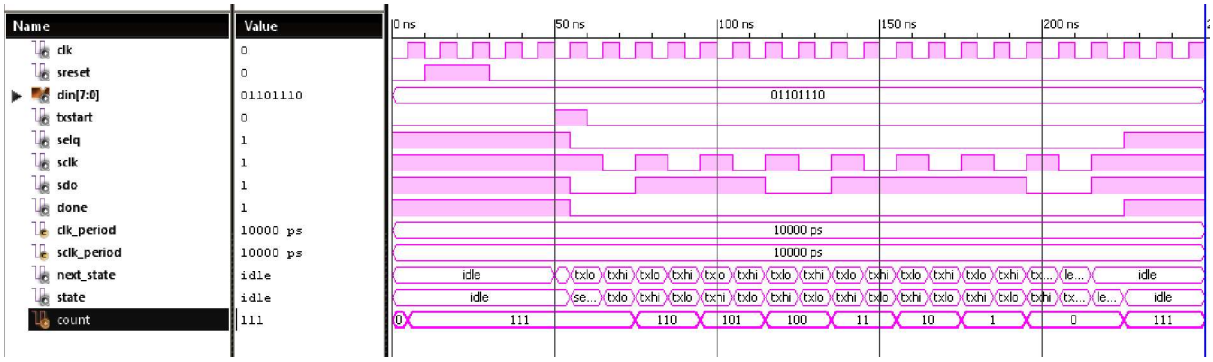


Figure 1.41: SPI transmit of the byte 0x6e

***

Lab #14:

6.12 Complete SPI Transmitter/Receiver (Master/Slave)

The complete SPI transmitter (from 6.11) and the receiver should be coded in separate modules (top level module, transmitter, and receiver). The transmitter and receiver modules can be used later for transmitting data to and from DACs and ADCs.

The complete system is outlined in fig. 1.42.

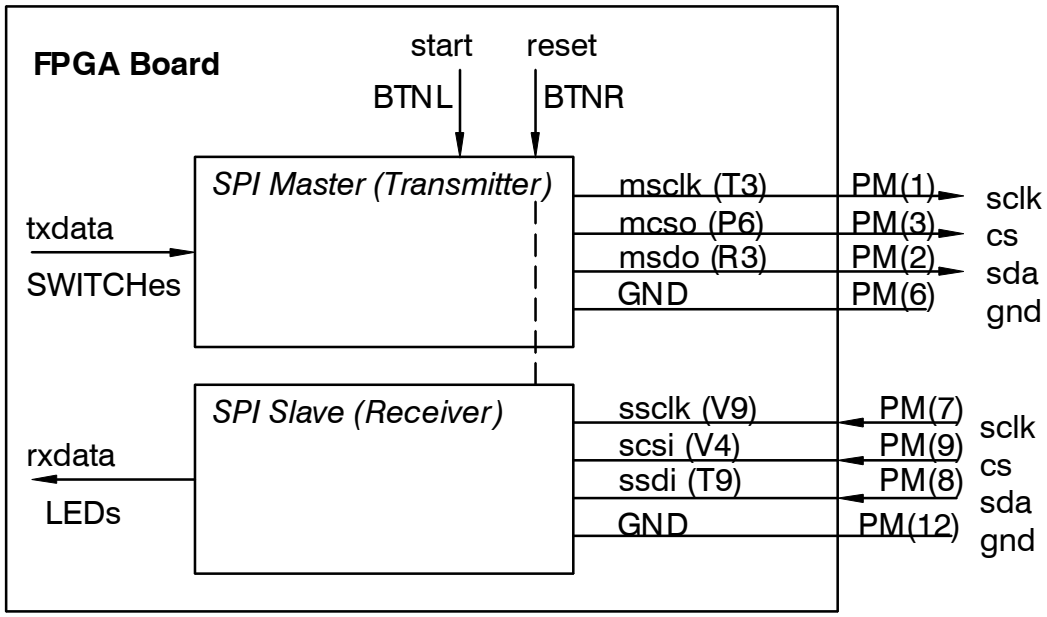


Figure 1.42: Master SPI transmitter and slave SPI receiver in one system

For Simulation (mandatory) the external connectors are not used, instead the simulation test bench VHDL module provide internal connections according to fig. 1.43.

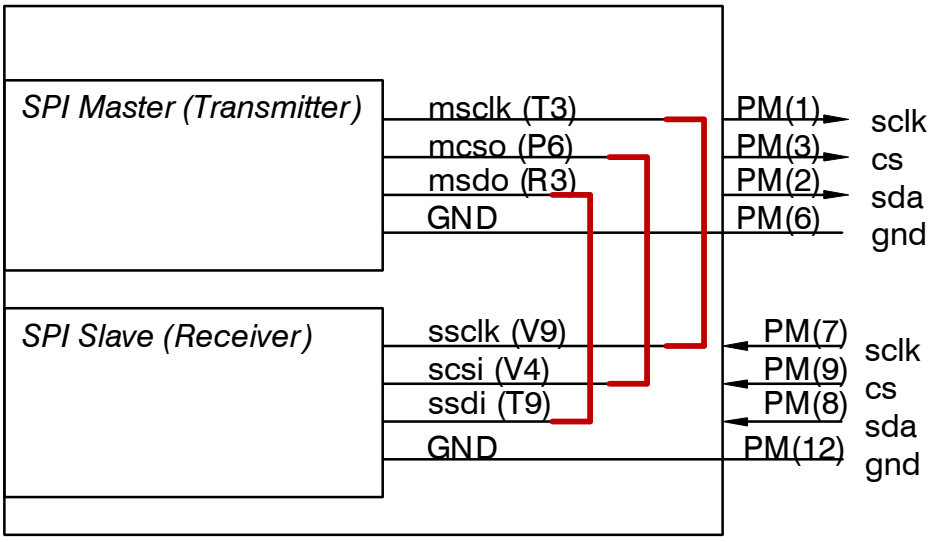


Figure 1.43: Internal connections by simulator

The final hardware test requires external connection on the standard “PMOD” connector.

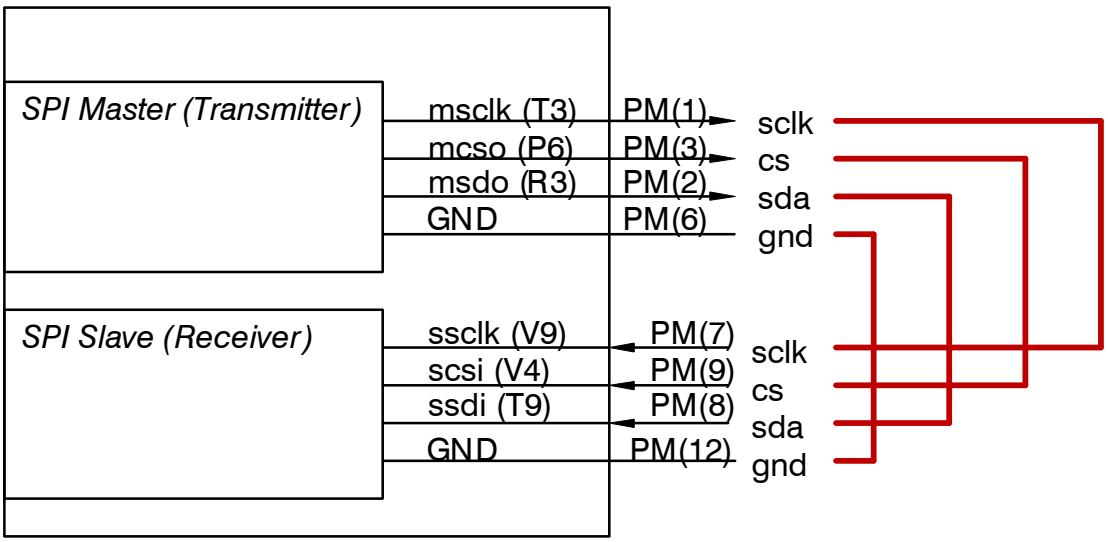


Figure 1.44: External connections for hardware test

The external connection allows communication between two boards by a “*crossover*” cable. Sending an receiving is possible in *full duplex* mode.

The external connector is a “PMOD” type connector with the following layout.

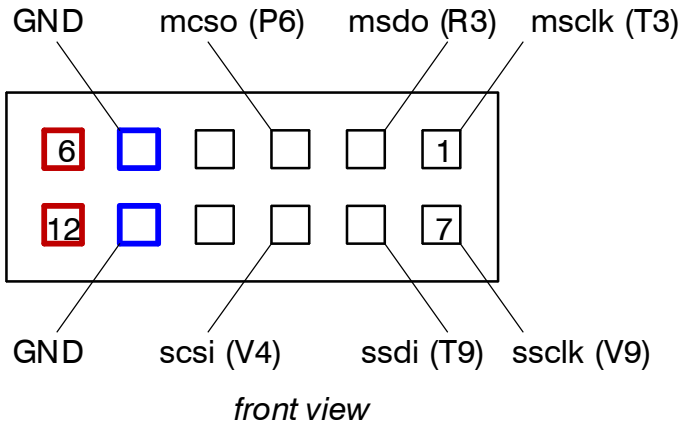


Figure 1.45: PMOD connector layout Atlys Spartan-6

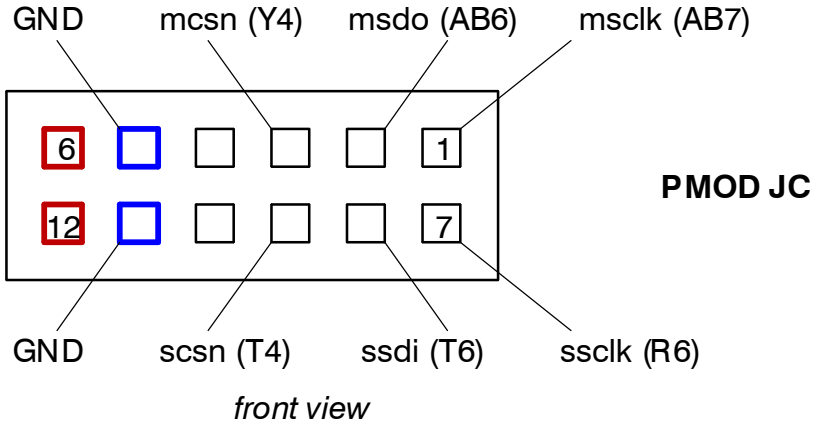


Figure 1.46: PMOD connector layout ZedBoard Zynq-7000

It is very important to choose the 2.5 V voltage supply for the PMOD connector signals, since the FPGA outputs of this section support only 2.5 V (see fig. 1.47).

Figure 1.47: 2.5 V selection for PMOD signals

- Start a new project "spirxtxng"
- Create a main code module with the following *ENTITY*:

```
entity spirxtx is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;       -- button right: F6
           start : in  STD_LOGIC;       -- button left:  P4
           msclk : out  STD_LOGIC;      -- PIN 1: T3  ->-+
           ssclk : in  STD_LOGIC;       -- PIN 7: V9  -<-+
           msdo : out  STD_LOGIC;       -- PIN 2: R3  -->--+
           mcso : out  STD_LOGIC;       -- PIN 3: P6  .... |  ..>..+
           ssdi : in  STD_LOGIC;        -- PIN 8: T9  --<--+       |
           scsi : in  STD_LOGIC;        -- PIN 9: V4  .........<..+
           txdata : in  STD_LOGIC_VECTOR (7 downto 0);     -- switches
           rxdata : out  STD_LOGIC_VECTOR (7 downto 0));   -- leds
end spirxtx;
```

The signal are as follows:

clk	state machine clock
reset	reset signal for *all* state machines (button right)
start	start signal for transmission (button left)
msclk	master serial clock (out)
ssclk	slave serial clock (in)
msdo	master serial data out
mcso	master chip select out
ssdi	slave serial data in
scsi	slave chip select in
txdata	data to be send (8 bits, from switches, in)
rxdata	data received (8 bits, to LEDs, out)

- You might use an optional *CONSTANT* definition in your main code:

```
CONSTANT NBITS : NATURAL := 8;
```

- Create a component declaration for the transmitter (file similar to previous lab):

```
    COMPONENT spitx IS
    GENERIC ( TXDATA_SIZE : INTEGER := 16 );
    PORT ( clk : in  STD_LOGIC;
           txclk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           start : in  STD_LOGIC;
           din : in  STD_LOGIC_VECTOR (7 downto 0);
           csq : out  STD_LOGIC;
           sclk : out  STD_LOGIC;
           sdo : out  STD_LOGIC;
           done : out  STD_LOGIC);
    END COMPONENT;
```

The transmitter requires the txclk signal which determines the transmission speed. The transmission speed should be 10 Mbit/s, since it is intended to be routed outside of the board by external cables. This signal has to be derived from the 100 MHz clk source by a PROCESS in the top level module.

- Create a component declaration for the receiver (file similar to previous lab):

```
COMPONENT spirx IS
GENERIC ( RXDATA_SIZE : INTEGER := 16 );
PORT ( clk : in  STD_LOGIC;
       reset : in  STD_LOGIC;
       csq : in  STD_LOGIC;
       sclk : in  STD_LOGIC;
       sdi : in  STD_LOGIC;
       dout : out  STD_LOGIC_VECTOR (7 downto 0);
       done : out  STD_LOGIC);
END COMPONENT;
```

Of course, the *GENERIC* default values needs to be overwritten by *NBITS* (= 8).

The only purpose of the top level VHDL module is the instantiation of the transmitter and the receiver. TO TEST THE DESIGN SIMPLY CONNECT THE SIGNAL *sdi* WITH *sdo*!

- Simulate the top level VHDL module with an arbitrary bit pattern. Then result should look similar to fig. 1.48.

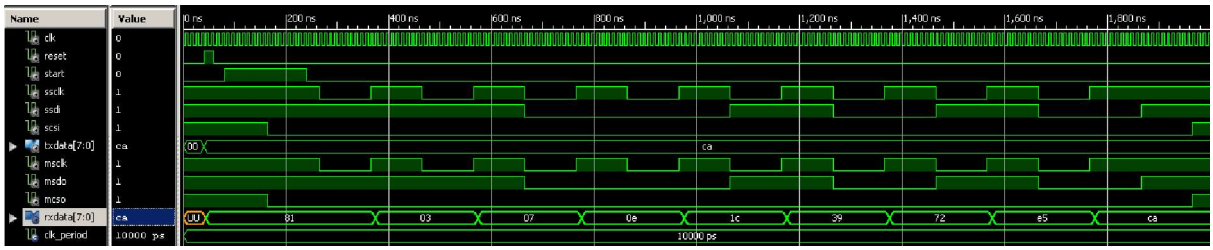


Figure 1.48: Simulation of SPI receiver and transmitter

- Create an user constraints file for pin definitions (PlanAhead). The resulting UCF file should look like this:

```
NET "clk" LOC = L15;
NET "mcso" LOC = P6;
NET "msclk" LOC = T3;
NET "msdo" LOC = R3;
NET "reset" LOC = F6;
NET "rxdata[0]" LOC = U18;
NET "rxdata[1]" LOC = M14;
NET "rxdata[2]" LOC = N14;
NET "rxdata[3]" LOC = L14;
NET "rxdata[4]" LOC = M13;
```

```
NET "rxdata[5]" LOC = D4;
NET "rxdata[6]" LOC = P16;
NET "rxdata[7]" LOC = N12;
NET "scsi" LOC = V4;
NET "ssclk" LOC = V9;
NET "ssdi" LOC = T9;
NET "start" LOC = P4;
NET "txdata[0]" LOC = A10;
NET "txdata[1]" LOC = D14;
NET "txdata[2]" LOC = C14;
NET "txdata[3]" LOC = P15;
NET "txdata[4]" LOC = P12;
NET "txdata[5]" LOC = R5;
NET "txdata[6]" LOC = T5;
NET "txdata[7]" LOC = E4;
```

- Make external connection on your board and verify that the data from the switches are properly transferred to the LEDs.
- Make crossover connection to another board and verify proper operation. **GND connections are necessary in this case!**

Lab #14a: (Zynq)

6.13 Complete SPI Transmitter/Receiver (Master/Slave)

The complete SPI transmitter (from 6.11) and the receiver should be coded in separate modules (top level module, transmitter, and receiver). The transmitter and receiver modules can be used later for transmitting data to and from DACs and ADCs.

The complete system is outlined in fig. 1.49.

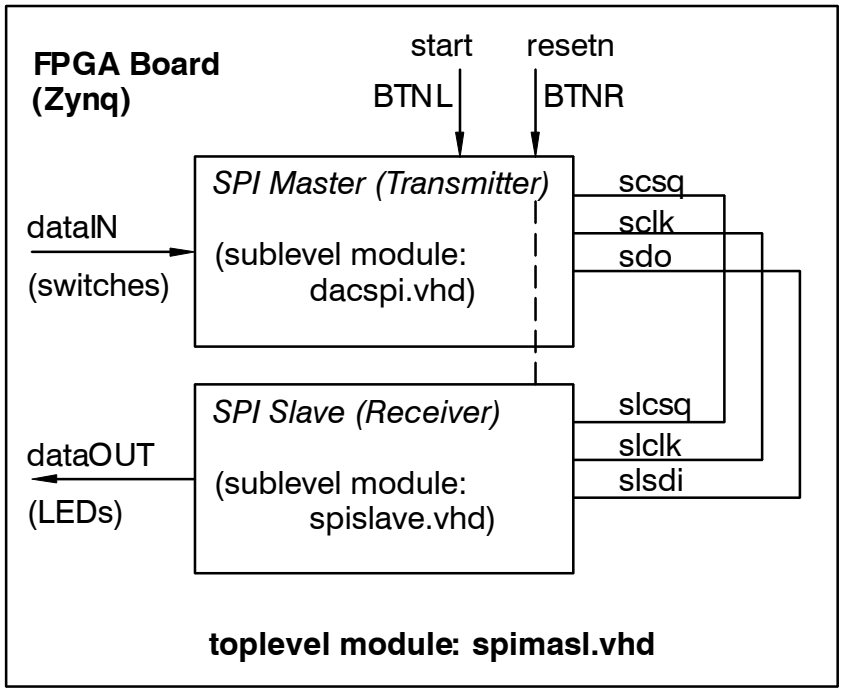


Figure 1.49: Master SPI transmitter and slave SPI receiver in one system, connections between master and slave made in toplevel module

For Simulation (mandatory) the external connections (switches/LEDs) are not used, instead the simulation test bench VHDL module provide the test data.

- Start a new project "spimasl"
- Create a main code module with the following *ENTITY*:

```
entity spimasl is
    Port ( breset : in STD_LOGIC;
           clk : in STD_LOGIC;
           start : in STD_LOGIC;
```

```
            dataIN : in STD_LOGIC_VECTOR (7 downto 0);
            dataOUT : out STD_LOGIC_VECTOR (7 downto 0));
end spimasl;
```

The signal are as follows:

breset	reset signal for *all* state machines (button right)
clk	state machine clock (system clock)
start	start signal for transmission (button left)
dataIN	data to be send (8 bits, from switches, in, required by master only)
dataOUT	data received (8 bits, to LEDs, out, required by slave only)

- You might use an optional *CONSTANT* definition in your main code:

```
CONSTANT  SPI_BITS : integer := 8;
```

- Create a component declaration for the transmitter (file similar to previous lab):

```
    COMPONENT dacspi is
    GENERIC (
        USPI_SIZE : INTEGER := 16 );
    Port ( resetn : in STD_LOGIC;
           bclk : in STD_LOGIC;
           spi_clkp : in STD_LOGIC;
           start : in STD_LOGIC;
           done : out STD_LOGIC;
           scsq : out STD_LOGIC;
           sclk : out STD_LOGIC;
           sdo : out STD_LOGIC;
           sndData : in STD_LOGIC_VECTOR (USPI_SIZE-1 downto 0));
end COMPONENT dacspi;
```

The transmitter requires the spi_clkp signal which determines the transmission speed. The transmission speed should be 25 Mbit/s, otherwise the slave will not be fiunctional. This signal has to be derived from the 100 MHz clk source by a PROCESS in the top level module. Use a constant declaration for the clock divider:

```
CONSTANT  SPI_CLK_DIV : integer := 4;
```

- Create a component declaration for the receiver (file similar to previous lab):

```
    COMPONENT spislave is
    GENERIC (
        USPI_SIZE : INTEGER := 16 );
    Port ( resetn : in STD_LOGIC;
           bclk : in STD_LOGIC;
           done : out STD_LOGIC;
```

```
            slcsq : in STD_LOGIC;
            slsclk : in STD_LOGIC;
            slsdi : in STD_LOGIC;
            slrcvData : out STD_LOGIC_VECTOR (USPI_SIZE-1 downto 0));
end COMPONENT spislave;
```

Of course, the *GENERIC* default values needs to be overwritten by `SPI_BITS` (= 8).

The only purpose of the top level VHDL module is the instantiation of the transmitter and the receiver. TO TEST THE DESIGN SIMPLY CONNECT THE SIGNAL *sdi* WITH *sdo*!

- Simulate the top level VHDL module with an arbitrary bit pattern.
- Add the supplied contraints file `spimasl.xdc` to the Project Manager:

```
# ZedBoard xdc

# define clk and period
create_clock -period 10.000 -name clk -waveform {0.000 5.000} [get_ports clk]
set_property PACKAGE_PIN Y9 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]

############################
# On-board pushbuttons     #
############################
# btnl (start)
set_property PACKAGE_PIN N15 [get_ports start]
set_property IOSTANDARD LVCMOS18 [get_ports start]
# btnr (breset)
set_property PACKAGE_PIN R18 [get_ports breset]
set_property IOSTANDARD LVCMOS18 [get_ports breset]

############################
# On-board switches        #
############################
# SW0 (dataIN)
set_property PACKAGE_PIN F22 [get_ports dataIN[0]]
set_property IOSTANDARD LVCMOS18 [get_ports dataIN[0]]
# SW1
set_property PACKAGE_PIN G22 [get_ports dataIN[1]]
set_property IOSTANDARD LVCMOS18 [get_ports dataIN[1]]
# SW2
set_property PACKAGE_PIN H22 [get_ports dataIN[2]]
set_property IOSTANDARD LVCMOS18 [get_ports dataIN[2]]
# SW3
set_property PACKAGE_PIN F21 [get_ports dataIN[3]]
set_property IOSTANDARD LVCMOS18 [get_ports dataIN[3]]
# SW4
set_property PACKAGE_PIN H19 [get_ports dataIN[4]]
set_property IOSTANDARD LVCMOS18 [get_ports dataIN[4]]
```

```
# SW5
set_property PACKAGE_PIN H18 [get_ports dataIN[5]]
set_property IOSTANDARD LVCMOS18 [get_ports dataIN[5]]
# SW6
set_property PACKAGE_PIN H17 [get_ports dataIN[6]]
set_property IOSTANDARD LVCMOS18 [get_ports dataIN[6]]
# SW7
set_property PACKAGE_PIN M15 [get_ports dataIN[7]]
set_property IOSTANDARD LVCMOS18 [get_ports dataIN[7]]

############################
# On-board leds            #
############################
set_property PACKAGE_PIN T22 [get_ports dataOUT[0]]
set_property IOSTANDARD LVCMOS33 [get_ports dataOUT[0]]
set_property PACKAGE_PIN T21 [get_ports dataOUT[1]]
set_property IOSTANDARD LVCMOS33 [get_ports dataOUT[1]]
set_property PACKAGE_PIN U22 [get_ports dataOUT[2]]
set_property IOSTANDARD LVCMOS33 [get_ports dataOUT[2]]
set_property PACKAGE_PIN U21 [get_ports dataOUT[3]]
set_property IOSTANDARD LVCMOS33 [get_ports dataOUT[3]]
set_property PACKAGE_PIN V22 [get_ports dataOUT[4]]
set_property IOSTANDARD LVCMOS33 [get_ports dataOUT[4]]
set_property PACKAGE_PIN W22 [get_ports dataOUT[5]]
set_property IOSTANDARD LVCMOS33 [get_ports dataOUT[5]]
set_property PACKAGE_PIN U19 [get_ports dataOUT[6]]
set_property IOSTANDARD LVCMOS33 [get_ports dataOUT[6]]
set_property PACKAGE_PIN U14 [get_ports dataOUT[7]]
set_property IOSTANDARD LVCMOS33 [get_ports dataOUT[7]]
```

- Load the design onto your board and verify that the data from the switches are properly transferred to the LEDs. **No external connections are necessary in this case!**

***

Lab #15: Function Generator

6.14 Function Generator for Triangular and Sine Wave Analog Output

The DAC SPI module (from 6.11) shall be used to generate a 10 KHz triangular wave and sine wave on the output of a two channel DAC as shown in fig. 1.50.

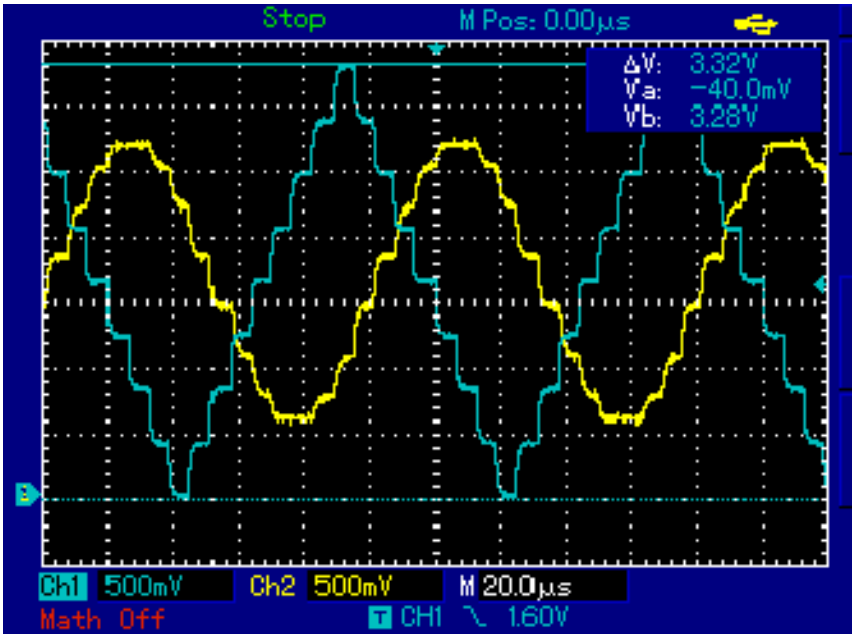


Figure 1.50: Scope measurement of DAC outputs

The DACs will be attached as a PMOD device on JA (pins 1..6, upper row) on the ARTY (Artix-7 FPGA).

The sampling frequency is $f_s = 160$ KHz (mainly limited by the DAC speed, not the FPGA performance). Therefore the sampling time is $T_s = 6.25$ µs.

The DAC is 12 bits, so legal range for data is $0 \le x \le 2^{12}\text{-}1 = 4095$. The analog output with supply voltage $V_{ss} = 3.3$ V is

$$u_{out} = 3.3\text{V}\,\frac{x}{4096}\,. \tag{1.4}$$

The 10 KHz triangular wave is given by

$$u_{tr} = A_{min} + kA_{delta} \,. \tag{1.5}$$

with $A_{min} = 16$, $A_{delta} = 508$ and k = 0, 1, 2, 3, 4, 5, 6, 7, 8, 7, 6, 5, 4, 3, 2, 1, 0, 1, ... This corresponds to a maximum value of 4080 (with k = 8). The index k is the output of an up/down counter. The period of this signal requires 16 samples, so the signal frequency is $f_{sig} = f_s$ / 16 = 160 KHz / 16 = 10 KHz.

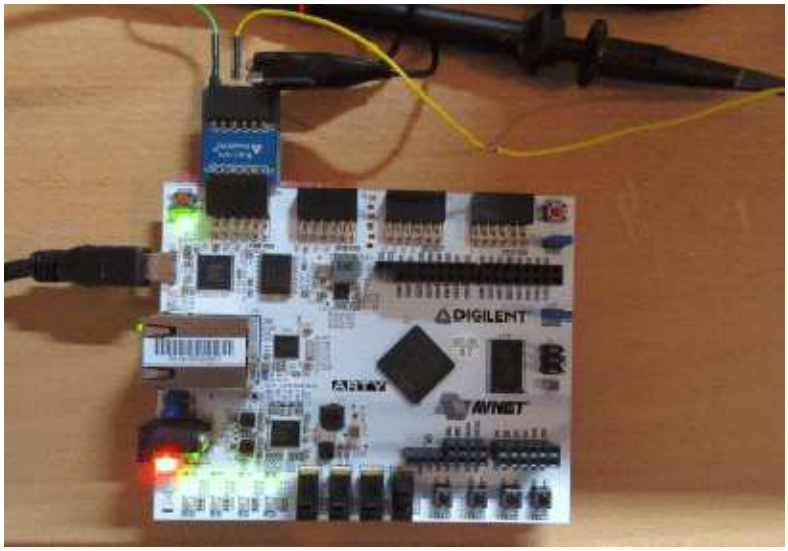


Figure 1.51: PmodDA2 device attached on port JA

The sine wave is generated from the triangular signal by a 9th order FIR filter in transposed form (suitable for hardware computation within 1 clock cycle).

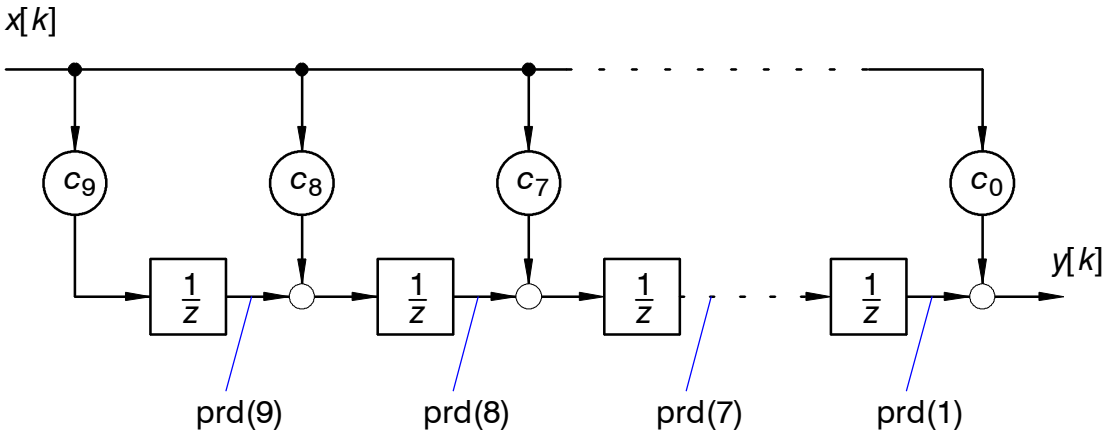


Figure 1.52: FPGA suitable Form of FIR filter

The maximal combinational path is short (product and sum). The values prd(1)...prd(9) can be computed in one clock cycle and in parallel. The vector of filter coefficients is given by fcoef = [786, 2136, 5820, 10422, 13604, 13604, 10422, 5820, 2136, 786] . The format is int_16.16 (16 fractional bits). Corresponding real values for the coefficients c_9 and c_0 are for instance $786 / 2^{16} = 0.011993$.

The the entity section of the top level VHDL module `fctgen.vhd` must be:

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

entity fctgen is
    Port ( reset : in STD_LOGIC;
           clk : in STD_LOGIC;
           start : in STD_LOGIC;
           spi_csq : out STD_LOGIC;
           spi_sclk : out STD_LOGIC;
           spi_sdo_0 : out STD_LOGIC;
           spi_sdo_1 : out STD_LOGIC;
           aLED : out STD_LOGIC_VECTOR (3 downto 0));
end fctgen;
```

Please note that the package `IEEE.NUMERIC_STD.ALL` is required.

Meaning of port names:

reset	active high reset (for state machines and SPI module)
clk	system clk (100 MHz)
start	start signal for signal generation
spi_csq	CSq (SPI)
spi_sclk	SCLK (SPI)
spi_sdo_0	SDO channel 0
spi_sdo_1	SDO channel 1
aLED	4 LEDs (for status indication)

The top level module fctgen.vhd shall use the module `spidacms.vhd` for sending data to the DACs. The component declaration is shown below:

```
COMPONENT spidacms IS
    GENERIC ( USPI_SIZE : INTEGER := 16 );
    PORT ( resetn : in STD_LOGIC;
           bclk : in STD_LOGIC;
           spi_clkp : in STD_LOGIC;
           dstart : in STD_LOGIC;
           ddone : out STD_LOGIC;
           scsq : out STD_LOGIC;
           sclk : out STD_LOGIC;
           sdo_0 : out STD_LOGIC;
           sdo_1 : out STD_LOGIC;
           sndData_0 : in STD_LOGIC_VECTOR (USPI_SIZE-1 downto 0);
           sndData_1 : in STD_LOGIC_VECTOR (USPI_SIZE-1 downto 0) );
END COMPONENT spidacms;
```

Meaning of port names:

resetn	active low reset (requires inverted reset)
bclk	system clk (100 MHz)
spi_clkp	clock pulse every 100 ns (for 5 MHz transmission rate)

dstart	start signal for sending
ddone	indicates send via SPI done
sclk	SPI clk
sdo_0	SPI SDO channel 0
sdo_1	SPI SDO channel 1
sndData_0	12 bit std_logic_vector for channel 0 data
sndData_1	12 bit std_logic_vector for channel 1 data

The algorithm requires explicit conversions from integer to std_logic_vector (defined in the `IEEE.NUMERIC_STD` package. Conversion is done by:

```
<std_logic> = std_logic_vector(to_singed(<intval>, <std_logic>'Length));
```

7 I²C Interface

The I²C interface (inter integrated circuit interface) is a popular and powerful bus system for connecting a peripheral devices (mainly analog components or sensors). Up to 1024 (slave) devices can be connected to just 2 wires (thus 2-wire-interface). A typical transfer to and from the Analog Devices AD5398™ digital-to-analog converter (DAC) used for medical equipment. All transfers are initiated by the master.

The protocol was introduced by Philips in 1982 and has been several times revised. The latest specification is dated from 2007. The maximum clock speed is 1MHz. Writing to a 16 bit DAC requires including start and stop signals 29 clock cycles (= 29µs).

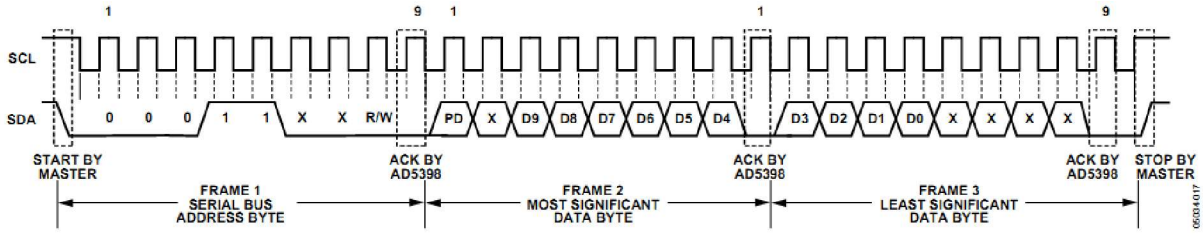


Figure 19. Write Operation

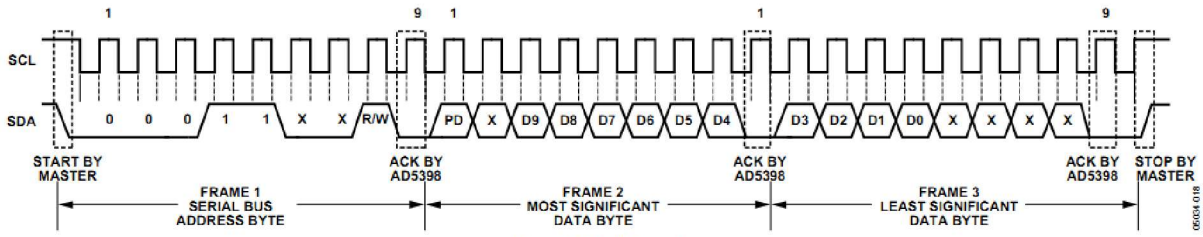


Figure 20. Read Operation

Figure 1.53: Read and write cycles on I²C bus

In fact the protocol asks for three wires:

GND common ground for master and slaves unless isolated

SCL	Serial CLock **provided by the master**
SDA	Serial DAta **bidirectional line driven by master and slave**

The *SDA* line requires pull-up resistors in order to maintain '1' if master and slaves are in 'Z' (high impedance) state.

As shown in the above figure the transfers are organized in bytes (8 bits). This requires a 9-bit-block, since after every byte an acknowledge from slave or master is required.

A master to slave data transfer is shown in fig. 1.54.

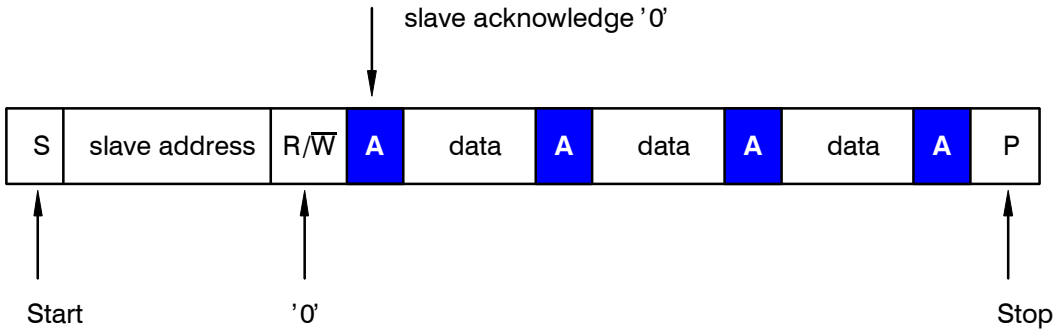


Figure 1.54: Master to slave transfer (3 data bytes) – WRITE (R/$\overline{W}$ = '0')

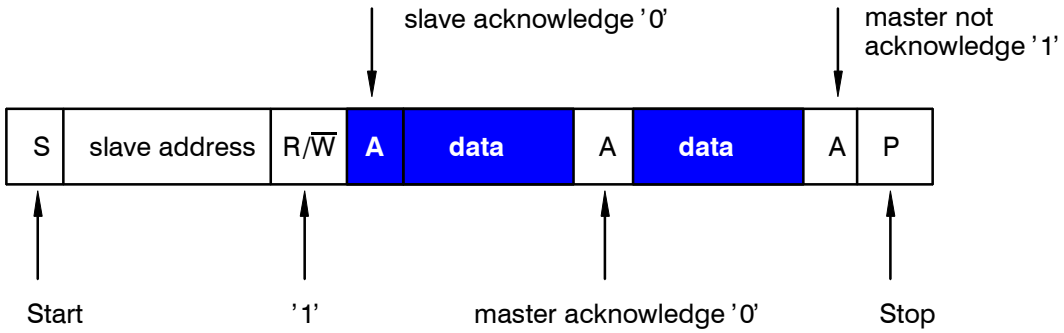


Figure 1.55: Data transfer from slave to master (2 data bytes) – READ (R/$\overline{W}$ = '1')

7.1 Bidirectional IO

On the I^{2}C bus the *sda* line serves for input and output. Since data is transferred in two direction this is called a bidirectional line. The VHDL code should provide three signals which need to combined into a single pad on the chip.

sdaout	data to be send on the I^{2}C bus
sdain	data from the I^{2}C bus (to be read)
inpena	input enable (if data should be read from *sdain*). Otherwise data will be sent to *sdaout* (*inpena*= '0')

If a pin on an FPGA is input and output a special buffer is required. The device primitive for this purpose is shown in fig. 1.56. The input signal *T* (Tristate) switch the upper buffer into a high impedance mode (STD_LOGIC value 'Z').

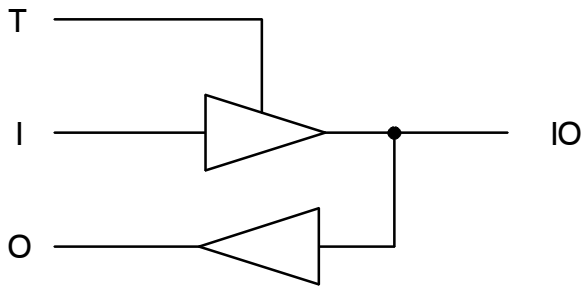


Figure 1.56: IOBUF device primitive

The truth tables for IOBUF show that *T*='0' open the upper buffer while T='1' puts its output into tristate ('Z') mode. Since the behavior depends on *T* it is better to use two truth tables.

T = '0':

I	IO	O
0	0	0
1	1	1

T = '1':

I	IO	O
X	0	0
X	1	1
X	Z	X

Figure 1.57: Truth tables of IOBUF for *T*='0' and *T*='1'

Note that if *T*='1' and *IO*='Z' (high impedance, not driven) then the output *O* is undefined. To avoid the last row in the second truth table a pull-up resistor is recommended. A smaller resistor improves the timing response when the output is switched to high impedance ('Z'). Of course, the resistor must never exceed the current capability of the output buffer.

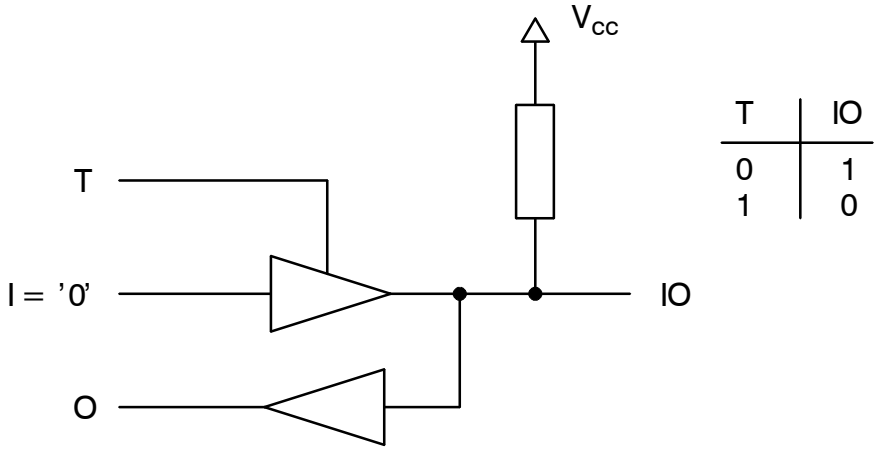


Figure 1.58: IOBUF with pull-up resistor to avoid undefined signal on *O*

The use of pull-up resistors avoid data contention (two active drivers on the same line).

The IOBUF primitive exist in almost all FPGAs. There are two possible methods to use this buffer. The first method is the instantiation of the primitive. On Xilinx FPGAs it is required to include the device library by using

```
library UNISIM;
use UNISIM.VComponents.all;
```

in the header section of the VHDL module. The primitive is then instantiated by the following code:

```
I2Csda_buf : IOBUF
generic map (
    DRIVE => 12,
    IOSTANDARD => "DEFAULT",
    SLEW => "SLOW")
port map ( O => sdain,    -- Buffer output (= sda input)
      IO => sda,            -- Buffer inout port (connect to pin)
      I => sdaout,          -- Buffer input (= sda output)
      T => inpena );        -- 3-state enable input, high=input, low=output
```

However, there is no need to use device primitives directly. The following code samples give the same result. It is assumed that the following port signals exist:

Sequential:

```
sdaiobuf: PROCESS(inpena, sdaout)
BEGIN
    IF inpena='0' THEN        -- send sdaout to sda
          sda <= sdaout;
    ELSE                      -- make bauffer tristate
          sda <= 'Z';
    END IF;
END PROCESS;

sdain <= sda;
```

Concurrent:

```
sda <= sdain WHEN inpena='0'
        ELSE 'Z';

sdain <= sda;
```

The behavioral description is portable on all devices. Using primitives is limited to the specific devices and depend on the vendor.

Since the output is wired-OR (pullup resisitor in fig. 1.58) a '1' is created in high impedance "Z" mode. A simple and good way to achieve this is

```
sda <= '0' WHEN sdaout='0'
        ELSE 'Z';
```

```
sdain <= sda;
```

This means only a '0' is driven by specifying '0' with `sdaout`, otherwise it becomes 'Z'. When reading an external value on `sda`, `sdaout` has to be '1'.

***

Lab #16: I²C

7.2 I²C Master Protocol for One-Byte READ/WRITE

An I²C master can access a large number of slow to medium speed (DACs, ADCs, sensors etc.). According to figures 1.53-1.55 of the I²C protocol the interface needs to implement the following features:

- start condition
- slave address transmission
- read/$\overline{\text{write}}$ transmission
- slave acknowledge capture
- one byte data transmission (in write mode)
- one byte data reception (in read mode)
- master "not acknowledge" signal generation (when reading)
- stop condition

The write (*rdwrq*= '0') and read (*rdwrq*= '1') cycles look as follows.

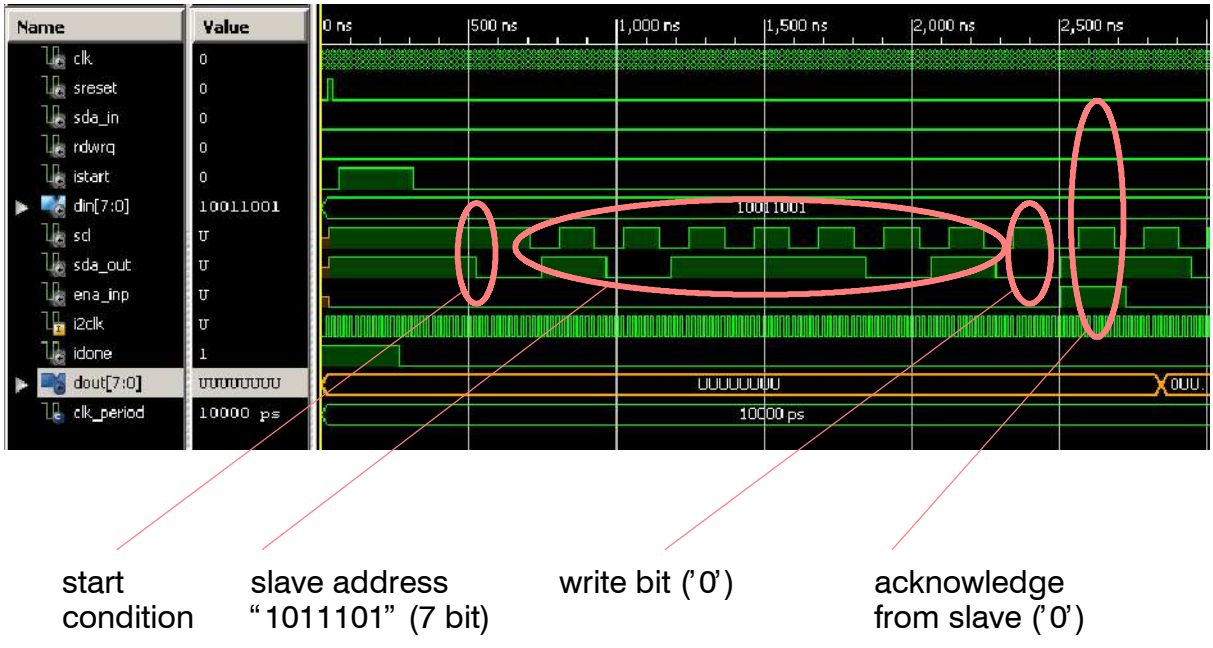


Figure 1.59: I²C write cycle (part #1)

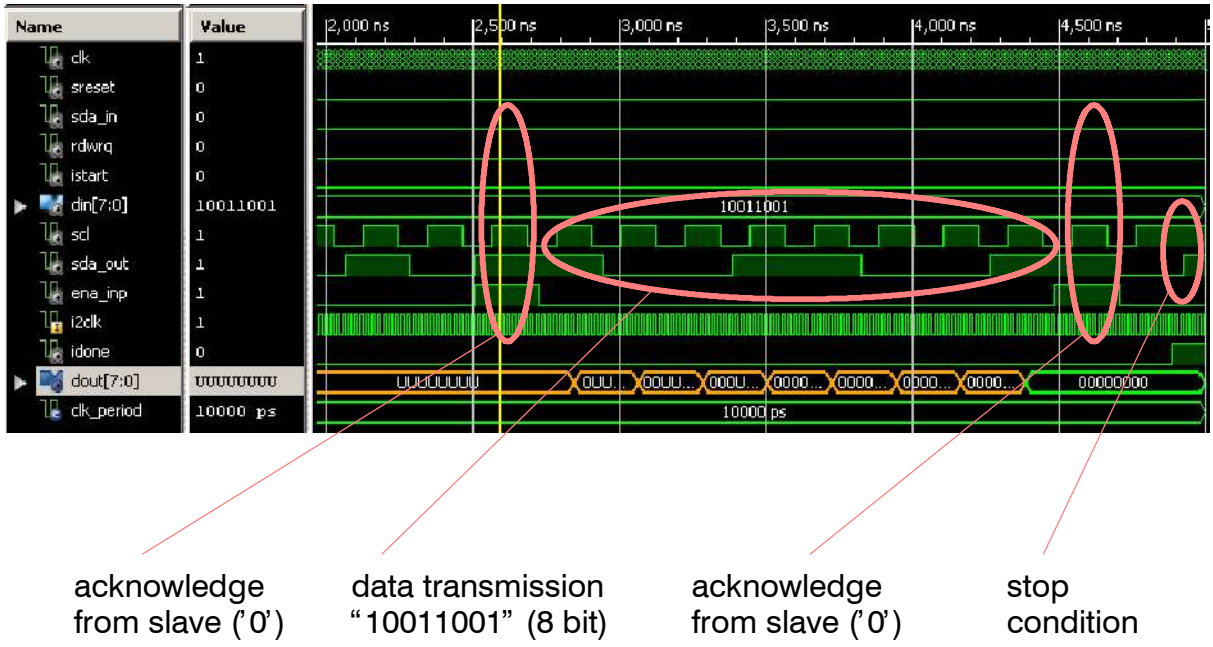


Figure 1.60: I²C write cycle (part #2)

Figure 1.61: I²C read cycle (part #1)

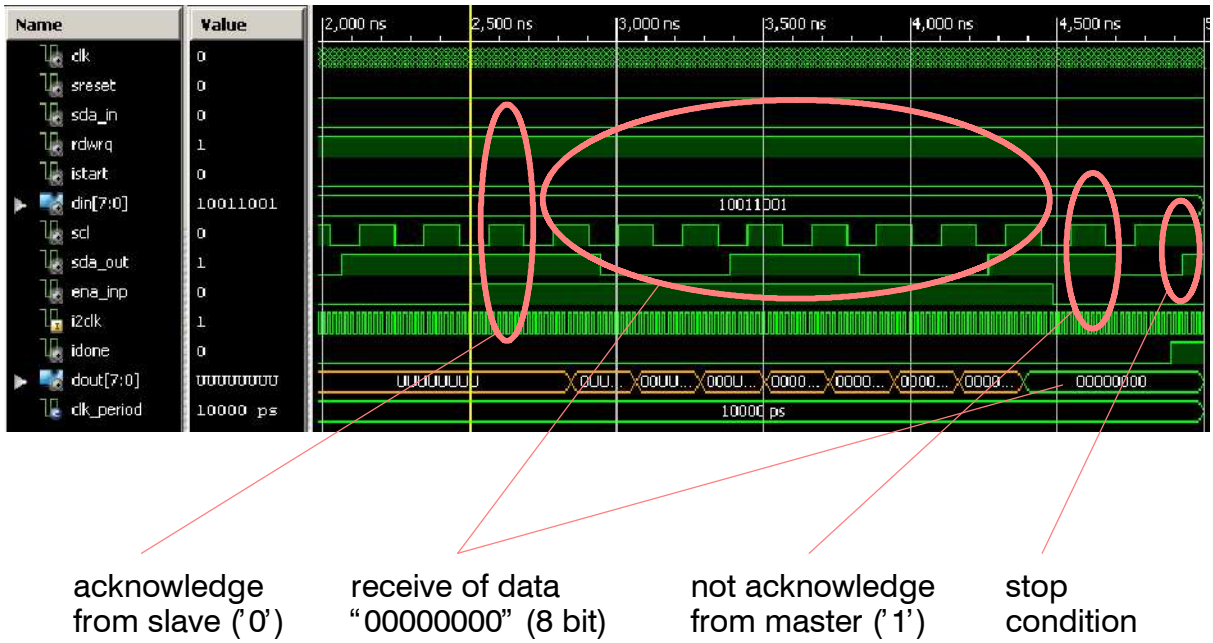


Figure 1.62: I^2C read cycle (part #2)

The bidirectional pin from section 7.1 is not required. This is necessary only if the interface is fully synthesized.

- Create a new project "i2cmaster"
- Create a main (top level) module "i2cmstr" with the following entity section:

```
entity i2cmstr is
    Port ( clk : in  STD_LOGIC;
           sreset : in  STD_LOGIC;
           scl : out  STD_LOGIC;
           mr_read : out  STD_LOGIC;
           sda_in : in  STD_LOGIC;
           sda_out : out  STD_LOGIC;
           rdwrq : in  STD_LOGIC;
           mstart : in  STD_LOGIC;
           mdone : out  STD_LOGIC;
           maddr : in  STD_LOGIC_VECTOR (6 downto 0);
           mdat_tx : in  STD_LOGIC_VECTOR (7 downto 0);
           mdat_rx : out  STD_LOGIC_VECTOR (7 downto 0));
end i2cmstr;
```

Signal explanation

clk	system clock
sreset	reset signal for state machine(s)
scl	serial clock line for I^2C bus
mr_read	enable input signal (read from sda)
sda_in	data read from sda line

sda_out	send data for sda line
rdwrq	'1' = I²C read cycle, '0' = I²C write cycle
mstart	start signal to initiate a transfer (start state machine)
mdone	indicates that a transfer is complete (or has aborted)
maddr	slave address (7 bits)
mdat_tx	byte to be sent over I²C (8 bits)
mdat_rx	byte read over I²C (8 bits)

- Implement the I²C master functionality.
- Since all bit transfers (including start and stop conditions) are similar, a special version of the sequential block of the state machine is recommended (equivalent to the time state machine in section 6.9). Based on the *i2clkp* clock pulses the following structure is be useful.

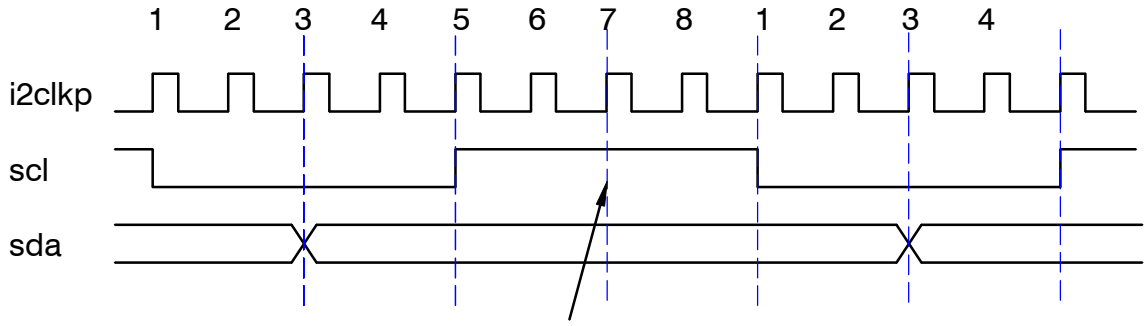


Figure 1.63: Signal generation for *scl* and *sda*

The sequential block contains a (local) variable (e.g. *counter*) from 0...8 to handle the exact generation of *scl* and *sda*(*_out*). Since data is valid during *scl* high time, the location to capture data while reading (data and slave acknowledge) is when the timer value is 7.

- The output values must not show any hazards. Therefore, a "glitch-free" state machine is required. The outputs must be registered, i.e. they must be outputs of DFFs. Since they are generated by the sequential part of the state machine FFs are created anyway.
- In a reference design the following states have been used:

```
TYPE state_type IS (idle, start, saddr, saddr_e, rd_wr, ack_addr, data,
                    data_e, ack_data, addr_err, data_err, stop);
SIGNAL state, next_state : state_type;
```

The "addr_err" and "data_err" states become active, if the slave does not issue an acknowledge after the address or data byte, respectively.

***

Lab #17:

7.3 I²C Master/Slave Protocol for One-Byte READ/WRITE

Adding an I²C slave makes the protocol complete. In order to test the design the signal *sdaout* of the master should be connected to *sdain* of the slave. The signal *sdout* of the slave is connected to *sdain* of the master.

- Create a new project "i2cms" (= I²C master and slave)
- Create a top-level module with the following ports:

```
entity i2ctop is
    Port ( clk : in  STD_LOGIC;
           ireset : in  STD_LOGIC;
           m_scl : out  STD_LOGIC;
           s_scl : in  STD_LOGIC;
           rdwrq : in STD_LOGIC;
           istart : in STD_LOGIC;
           m_read : out  STD_LOGIC;
           s_read : out  STD_LOGIC;
           m_sdaout : out  STD_LOGIC;
           m_sdain : in  STD_LOGIC;
           s_sdaout : out  STD_LOGIC;
           s_sdain : in  STD_LOGIC;
           m_din : in  STD_LOGIC_VECTOR (7 downto 0);
           m_dout : out  STD_LOGIC_VECTOR (7 downto 0);
           s_din : in  STD_LOGIC_VECTOR (7 downto 0);
           s_dout : out  STD_LOGIC_VECTOR (7 downto 0));
end i2ctop;
```

- Create a component for the master (similar to the previous lab)

```
COMPONENT i2cmr IS
    PORT ( clk : in  STD_LOGIC;
           sreset : in  STD_LOGIC;
           i2clk : in STD_LOGIC;
           scl : out  STD_LOGIC;
           mr_read : out  STD_LOGIC;
           sda_in : in  STD_LOGIC;
           sda_out : out  STD_LOGIC;
           rdwrq : in  STD_LOGIC;
           mstart : in  STD_LOGIC;
           mdone : out  STD_LOGIC;
```

```
            maddr : in  STD_LOGIC_VECTOR (6 downto 0);
            mdat_tx : in  STD_LOGIC_VECTOR (7 downto 0);
            mdat_rx : out  STD_LOGIC_VECTOR (7 downto 0));
END COMPONENT i2cmr;
```

- Create a component for the slave

```
COMPONENT i2csl IS
    PORT ( clk : in std_logic;
           sreset : in std_logic;
           sladdr : in std_logic_vector(6 downto 0);
           scl : in std_logic;
           sl_read : out  STD_LOGIC;
           sda_in : in  STD_LOGIC;
           sda_out : out  STD_LOGIC;
           sdone : out  STD_LOGIC;
           sdat_tx : in  STD_LOGIC_VECTOR (7 downto 0);
           sdat_rx : out  STD_LOGIC_VECTOR (7 downto 0));
END COMPONENT i2csl;
```

- Instantiate the components and add the missing slave module. Note that the slave needs only the system clock (clk). sda_out can be set during scl = '0' and data can be captured if scl = '1', see figure 1.64 below.

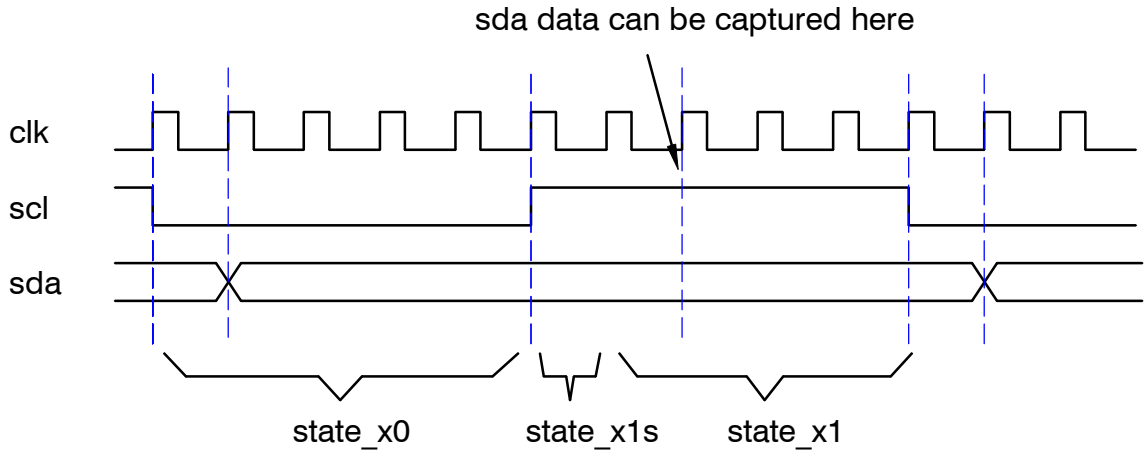


Figure 1.64: Divisions of *scl* clocks for signal generation and capture

The major difference between master and slave is the dependency on *scl*. The slave cannot rely on a fixed *scl* rate since the master may delay the clock cycles as long as the master wants. It is only guaranteed that the clock cycles do never fall below a minimum value.

Therefore, *scl*='0' requires a state as well as *scl*='1'. An additional state can be used to decrement counters for address and data counters.

The figures below show the complete I^2C read and write cycle with master and slave.

- In a reference design the following states for the slave have been used:

```
TYPE state_type IS (idle, start, adr0, adr1_s, adr1,
                    adr0_e, adr1_se, adr1_e, ackn_adr0, ackn_adr1,
                    rdwrq0, rdwrq_s, rdwrq1,
                    data0, data1_s, data1, data0_e, data1_se, data1_e,
                    ackn_dat0, ackn_dat1, stop0, stop1);
SIGNAL state, next_state : state_type;
```

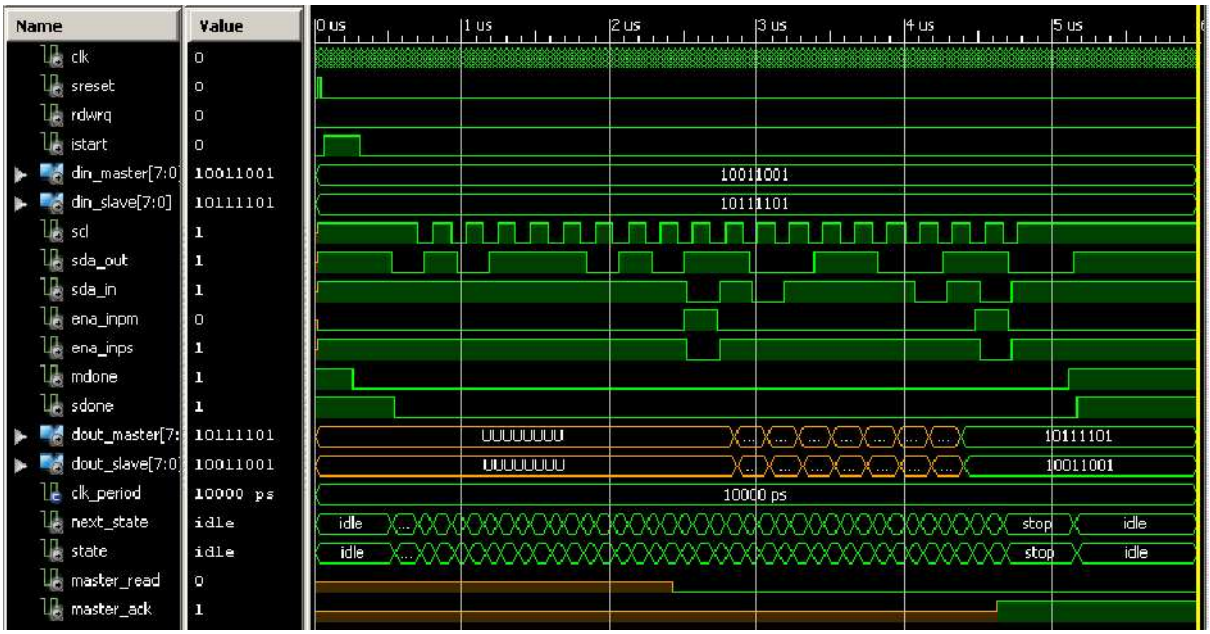


Figure 1.65: I²C master write cycle + slave

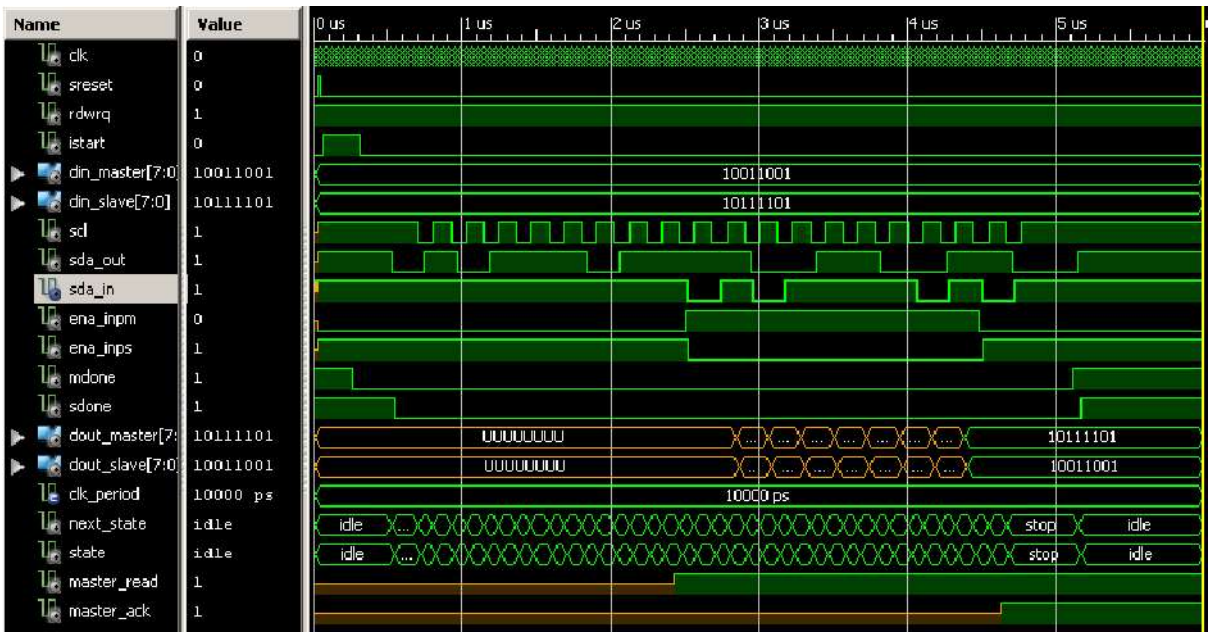


Figure 1.66: I²C master read cycle + slave

***

8 Bibliography

[1] Peter J. Ashenden: The Designer's Guide to VHDL, 3rd. Ed.
Morgan Kaufmann, 2008

[2] John F. Wakerly: Digital Design, Principles & Practices.
Prentice Hall, 2001

[3] Volnei A. Predroni: Circuit Design and Simulation with VHDL, 2nd. Ed.
MIT Press, 2010

[4] Roger Lipsett, Carls Schaefer, Cary Ussery: VHDL Hardware Description and Design.
Kluwer Academic 1990

[5] Sudhakar Yalamanchili: VHDL Starter's Guide.
Prantice Hall, 1998

[6] J. Reichardt, B. Schwarz: VHDL-Synthese.
Oldenbourg, 2001

***