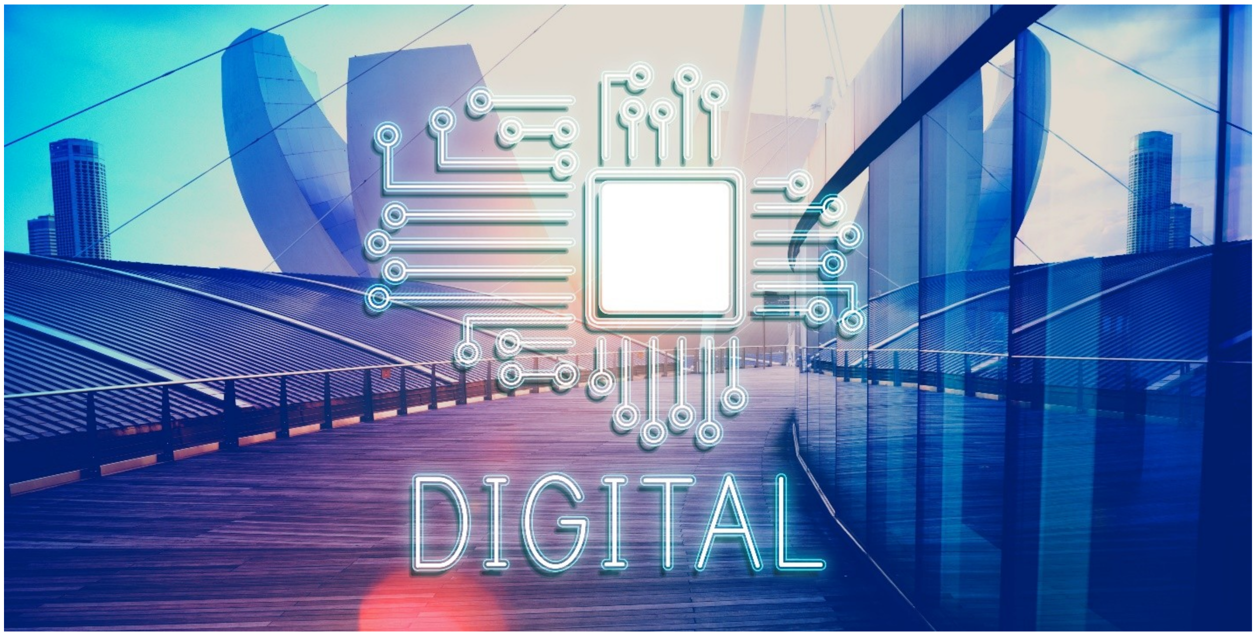
Călin BÎRĂ



===

Digital Electronics by Example

When Hardware greets Hardware

Preface

This book is an educational book and provides examples and exercises for the students in 2nd year of bachelor's degree path, regarding the Digital Electronics topic.

Chapter one provides an introduction into analog and digital signals and systems and is similar to the introduction in [1]

Chapter two starts with exercises regarding 0-loop circuits (CLCs).

Table of Contents

List of Figures

List of Tables

List of Acronyms

AACS	Advanced Access Content System
AC	Alternative Current
ACK	Acknowledge
ADC	Analog to digital converter
AES	Advanced Encryption Standard
AGM	Absorbent Glass Mat
AMD	Advanced Micro Devices
ANSI	American National Standards Institute
ARPA	Advanced Research Projects Agency
ARPANET	Advanced Research Projects Agency Network
ASIC	Application Specific Integrated Circuit
BIOS	Basic Input-Output System
BJT	Bipolar Junction Transistor
BOD	Brown-out detect
CBC	Cipher Block Chaining
CD	Compact Disk
CDIP	Ceramic Dual Inline Package
CERN	Conseil Européen pour la Recherche Nucléaire / European Council for Nuclear Research
CFB	Cipher Feedback Mode
CMOS	Complementary Metal-Oxide Semiconductor
COBOL	Common Business-Oriented Language
CPHA	Clock Phase
CPOL	Clock Polarity
CPU	Central Processing Unit

CRT	Cathode-Ray Tube
CS	Chip Select
CSNET	Computer Science Network
CTFT	Continuous Time Fourier Transform
CTR	Counter
CUDA	Compute Unified Device Architecture
CV	Computer Vision
DAC	Digital to Analog Converter
DDR	Double Data Rate
DES	Data Encryption Standard
DHCP	Dynamic Host Configuration Protocol
DIP	Dual In-line Package
DNS	Domain Name Service
DRAM	Dynamic Random Access Memory
DUT	Device Under Test
EB	Exabyte
ECC	Error Correction Code
EMES	Engineering of Modern Electric Systems
FET	Field-Effect Transistor
FIFO	First-In First-Out
FIPS	Federal Information Processing Standard
FM	Frequency Modulation
FTP	File Transfer Protocol
GB	Gigabyte
GHz	Gigahertz
GND	Ground
GP	General Purpose
GPIO	General Purpose Input/Output
GPU	Graphical Processing Unit
GUI	Graphical User Interface
HD	High Definition
HDD	Hard Disk Drive
HDL	Hardware Description Language
HTML	Hypertext Markup Language
IC	Integrated circuit
IDE	Integrated development environment
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IR	Infrared
ISO	International Organization for Standardization
JEDEC	Joint Electron Device Engineering Council
JS	JavaScript

KB	Kilobyte
kHz	Kilohertz
LED	Light-emitting diode
LIDAR	Light Detection and Ranging
LSB	Least Significant Bit/Byte
MATLAB	Matrix Laboratory
MB	Megabyte
MCU	Microcontroller Unit
MHz	Megahertz
MIL-STD	Military Standard
MIT	Massachusetts Institute of Technology
MOhm	Megohm
MOS	Metal-Oxide Semiconductor
MSB	Most Significand Bit/Byte
MW	Megawatt
NACK	Not Acknowledge
NAND	Not AND
NIST	National Institute of Standards and Technology
NOM	Nominal
NOR	Not OR
NTC	Negative Temperature Coefficient
NVM	Non-Volatile Memory
OOK	On/Off Keying
OP	Operation
OSI	Open Systems Interconnection Model
OTP	One Time Password
OUT	Output
PB	Petabyte
PC	Personal Computer
PCB	Printed Circuit Board
PDIP	Plastic Dual Inline Package
PIR	Passive Infrared
PN	Part Number
PTC	Positive Temperature Coefficient
PTP	Picture Transfer Protocol
QSPI	Quad SPI
R,G,B	Red, Green, Blue
RADAR	Radio Detection and Ranging
RAM	Random Access Memory
RC	Remote Control
RDS(on)	Resistance from drain to source, when in on state
RF	Radio Frequency

RFC	Request for Comments
RFID	Radio Frequency Identification
RGB	Red Green Blue
RH	Relative humidity
RMS	Root Means Square
RX	Receiver or reception
SAR	Successive approximative register
SCK	Serial Clock
SCL	Serial Clock
SD	Secure Digital
SDA	Serial Data
SDRAM	Synchronous Dynamic Random Access Memory
SMD	Surface Mount Device
SOIC	Small Outline Integrated Circuit
SPDT	Single Pole Double Throw
SPI	Serial Peripheral Interface
SPST	Single Pole Single Throw
SQL	Structured Query Language
SRAM	Static Random Access Memory
TB	Terabyte
TCR	Temperature Coefficient of Resistance
THT	Through Hole Technology
TIOBE	The Importance Of Being Earnest
TTL	Transistor-to-transistor logic
TX	Transmitter / Transmission
UART	Universal Asynchronous Receiver/Transmitter
UDIMM	Unbuffered Dual In-Line Memory Module
UDP	User Datagram Protocol
UHF	Ultra-High Frequencies
USART	Universal Synchronous/Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
UV	Ultraviolet
V	Volt
VHDL	VHSIC Hardware Description Language
VHF	Very High Frequency
VI	Input Voltage
VIH	Input Voltage (high, minimum)
VIL	Input Voltage (low, maximum)
VO	Output Voltage
VOH	Output Voltage (high, minimum)
VOL	Output Voltage (low, maximum)
XOR	Exclusive OR

YB	Yottabyte
ZB	Zettabyte

1. Brief Introduction to Digital Circuits and Programming

This chapter is similar to the first chapter in [1] and presents an introduction into digital signals and systems. There are advantages of converting analog signals into digital signals and back, otherwise one would not go through the trouble and cost of the conversion. These reasons will be highlighted at the end of this chapter.

1.1 Analog signals

All the studied systems in high-school's physics classes were composed from analog equipment/devices (voltage supplies, current supplies, resistors, capacitors, inductors, lightbulbs etc.). They are circuits where, for example, the voltage varies continuously within some limits. A default system is an analog audio amplifier, which takes an analogue audio signal, amplifies it (keeps the shape, but delivers more power from the supply) and sends it to the speakers. Analogue signals are hard to store and process, so lately, digital signals are used increasingly.

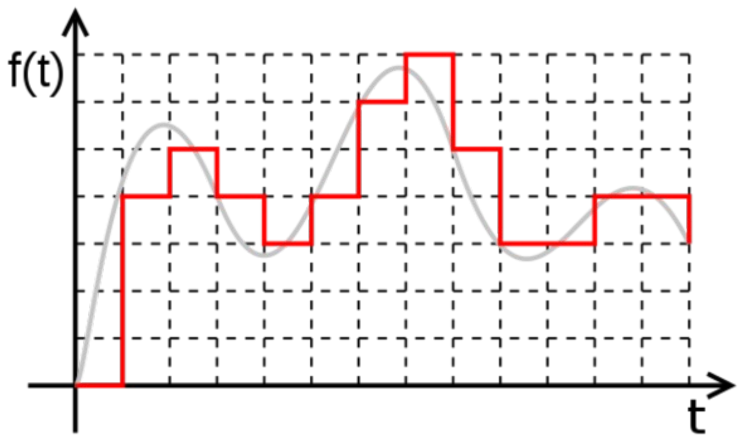


Figure 1. A digital signal (as a result of both sampling and quantization processes)

1.2 Digital signals

A digital signal is a signal which is discrete (as opposed to continuous) in both time and value. To create a time-discrete signal, one samples a continuous one. To create a value-discrete signal, one quantizes a continuous one.

The number of samples taken in a unit of time is called sampling rate (e.g., 44100 Hz == samples per second, CD-quality). The number of bits (0 or 1 symbols) required to express the amplitude is linked directly to the number of quantization steps (e.g., 16-bit for 2 to the power of 16 = 65536 steps, in the case of CD-audio quality)

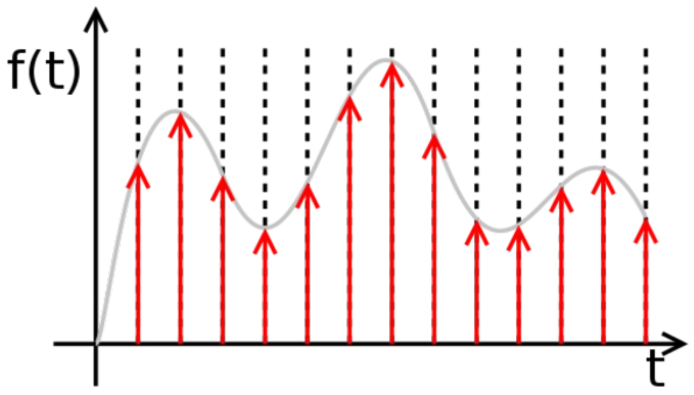


Figure 2. A time-sampled signal (as a result of sampling process)

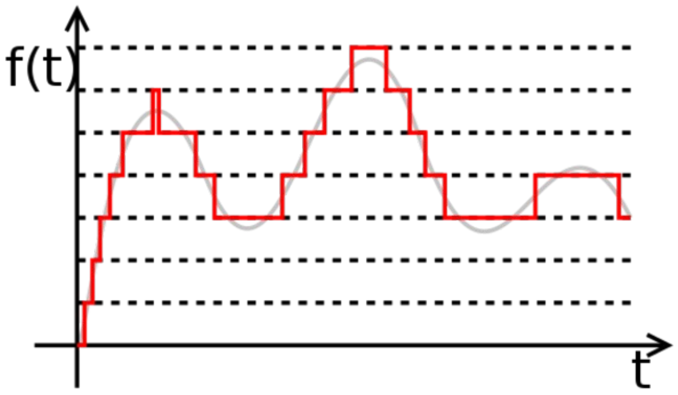


Figure 3. A value-sampled signal (as a result of quantization process)

Digital signals are used because their expression uses numbers, which allows easy
storage, copying (without loss) and processing. Immunity to noise can be obtained using mathematical instruments like error-detection and error-recovery processes. In addition, the quality (how similar it looks to the source

analogue signal) can be chosen as a compromise. The more quantization steps we use (e.g. infinite) the more accurate the value is to the source signal’s value: however, these systems are usually used for the comfort of human life, so the trade-off will take into consideration human hearing or human sight etc. which will not push the quantization step too high (e.g. audio signals are good enough when using 64k steps, that is 16-bits per sample; video signals are good enough when colors are represented with 256 steps of Red, Green and Blue, therefore 3x 8-bits are enough for a pixel). Regarding the recovery of a continuous signal from the time-sampled signal, we have the sampling theorem which demonstrates that we can fully recover the original, as long as the sampling rate is at least twice the maximum frequency contained in the original signal. For example, if one wants to recover up to 22 kHz audio signals (more than what the common human ear can hear), one should use 44 kHz sampling rate.

1.3 Digit, number, and radix

A radix-10 number uses a dictionary of 10 symbols (the ten digits) to express any number. For example, number 123 is made of 1 * 100 + 2 * 10 + 3. The radix of 10 is not the only known radix but is the most used by humans (arguably because we have ten fingers and can count easily using them). However, to use radix of 10, one must distinguish between 10 different symbols (0-9). Digital electronics use radix of 2 because it is easier to distinguish between only two symbols, therefore it is easier to store information in this form. The trade-offs that same number expressed in radix of 10 is around 3.5 times shorter that a radix of 2. For example, 9 is expressed in radix 2 with the sequence 1001, the number 127 is 11111111 etc. The symbols available for the radix of 2 are 0 and 1, and they are called BInary digiTS, in short bits. Using 2 digits we can express numbers from 0 to 99 (that is, 100 different numbers). Using 2 bits we can express numbers from 0 to 3 (that is, 4 different numbers). Most common radices are 2 (binary), 8 (octal), 16 (hex), 10 (dec) and 256. We will mark numbers in radix 2, as prefixed with 0b e.g.: 0b1001 is number 9 in radix 10. The hexadecimal number will be prefixed with 0x e.g.: 0x10 is 16 in radix 10.

Table 1. Multiples of bits / bytes according to JEDEC [3.]. IEC 80000-13 standard changes the name for the power of two, by inserting a “bi” in the name: Kibibyte, Mebibyte, Gibibyte.

Memory Unit (JEDEC)	**Memory unit (IEC)**	**Description**
Bit	Bit	Binary Digit 1 or 0
Kbit	Kibibit	1024 bits
Mbit	Mebibit	1024 Kbits

Byte	Byte	8 bits
KiloByte(KB)	KibiByte (KiB)	1024 Bytes
MegaByte(MB)	MebiByte (MiB)	1024 KB
GigaByte(GB)	Gibibyte (GiB)	1024 MB
TeraByte(TB)	Tebibyte (TiB)	1024 GB
PetaByte(PB)	Pebibyte (PiB)	1024 TB
HexaByte or exaByte (EB)	Exbibyte (EiB)	1024 PB
ZettaByte (ZB)	Zebibyte (ZiB)	1024 EB
YottaByte (YB)	Yobiibyte (YiB)	1024 ZB

1.4 Digital systems

Digital systems are designed to store and process and exchange information in digital form. They are found in a wide range of applications, including process control, communication systems, digital instruments, and consumer products. These systems/circuits may be classified by the number of *appropriate* loops enclosed within [5.]; more loops will mean more autonomy, therefore *smarter* circuits.

0 - loop circuits: contain only combinational circuits (logic gates)

1 - loop circuits: the memory circuits, with behavioral autonomy in their own internal states; they are mainly used for *storing*

2 - loops circuits: the automata, with the behavioral autonomy in their own state space, performing mainly the function of *sequencing*

3 - loops circuits: the processors, with the autonomy in interpreting their own internal states; they perform the function of *controlling*

4 - loops circuits: the computers, which interpret autonomously the programs according to the internal *data*

n-loop circuits: systems in which the information is interpenetrated with the physical structures involved in processing it; the distinction between *data* and *programs* is surpassed and the main novelty is the *self-organizing* behavior.

Any k-loop circuit can do everything any k-1 loop circuit can do.

While 0 – loop circuits (combinational logic circuits) are quite easy to grasp as they are very simple in structure and behavior, the more evolved circuits, containing sequential circuits (with the clock signal driving them) are the ones used to handle complexity.

Some common 0-loop circuits are: logic gates, multiplexers (sends the selected digital input to the output), demultiplexers (send the input to the selected output), decoders (sends logic 1 to the selected output), adders, subtractors, ALUs (arithmetical-logical units), equality comparators, magnitude comparators etc.

Some common sequential circuits are flip-flops (FFs), registers, counters/timers, and FSMs.

1.5 The advantages of using digital systems

The world is analog, therefore, interacting with it, using digital systems, implies conversion between analog to digital and back.

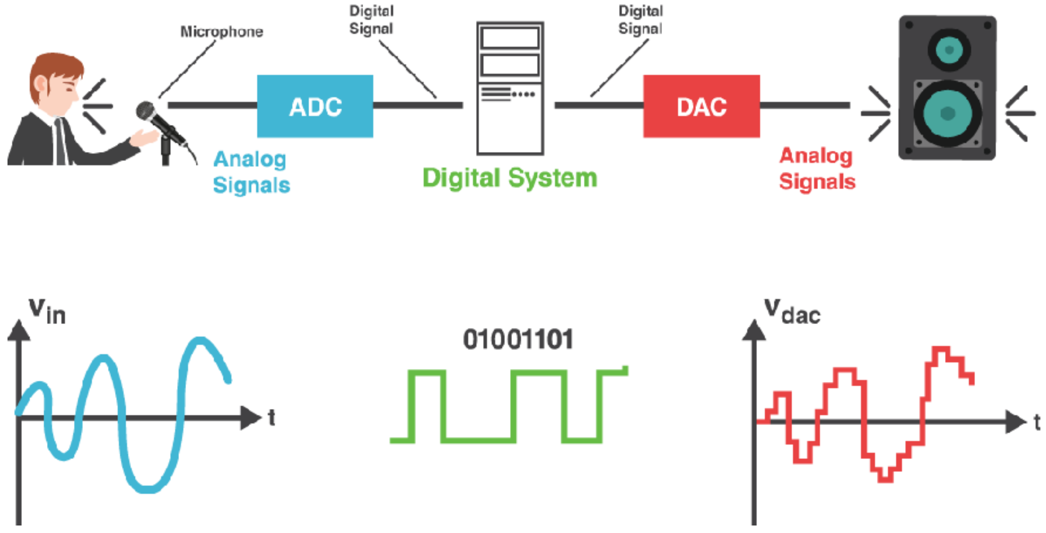


Figure 4. Analog to digital conversion, digital processing and digital to analog conversion example

The main advantages of using a digital system:

- Error correction: math (using numbers) can help a lot to find errors and correct them
- Noise tolerance: copy of a copy may be digitally identical, whereas copy of a copy in analog storage is never the same. Keeping numbers

in base2, allows maximum noise immunity as the digital system only has to discriminate between the two possible different symbols. In analog,

- Compression: crunching numbers is possible when
- Modularity: data transfers between digital modules, imply that data processing in a digital system is modular, therefore, there is a high chance of reusability in both hardware modules and software modules. Modular is good as it allows divide-et-impera method of problem-solving.
- Encryption: it is easy to scramble and obfuscate data in other data
- Repeaters: low-cost repeaters and they only have to amplify two symbols accurately (the noise of the repeater can be high)
- Compromise of space & compute power: it can be done in the field, not in the factory. Digital systems are easily configurable to save space and power with acceptable compromises on quality

2. The FPGA (Field Programmable Gate Array)

The FPGA is an semiconductor device based on a matrix of reconfigurable logic blocks (CLBs) as seen in Figure 5. The advantage of this device is that it can be reconfigured to emulate any digital circuit of a certain complexity, in the field (not in the factory!). The interconnection can relay signals coming from any direction to signals going towards any direction. The I/O cells allow signals to travel between the FPGA and outer world.

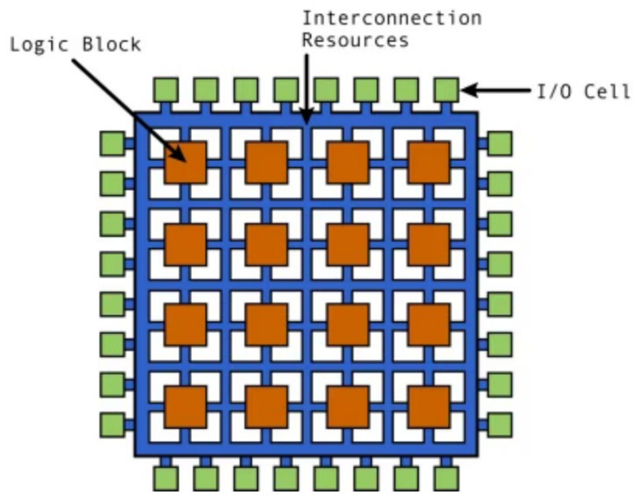


Figure 5. Generic FPGA Architecture Overview

An example of a configurable logic block (CLB) is exemplified below, in Figure 6.

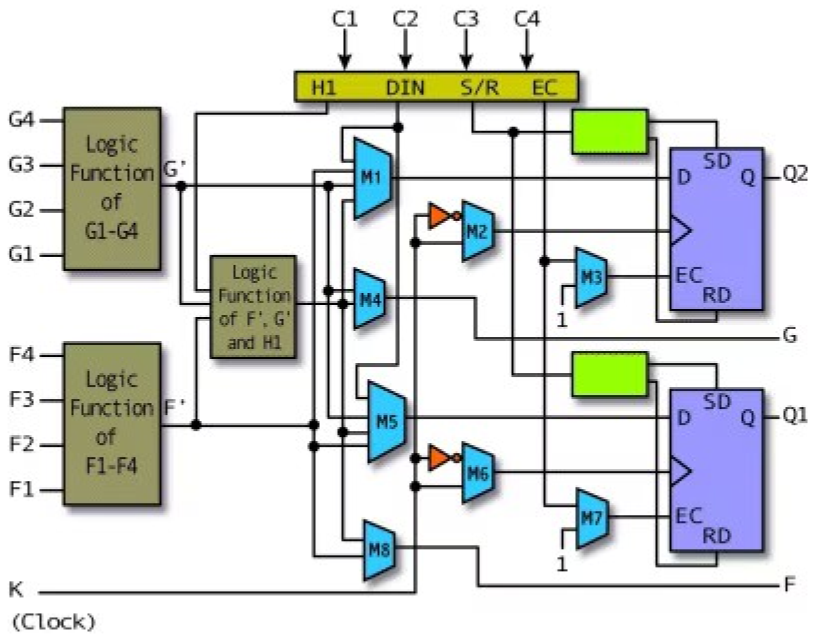


Figure 6. Xilinx CLB. Blue blocks are multiplexers, violet blocks are FFs and dark-green blocks are LUTs (look-up tables)

The software tools first convert code into RTL (register-transfer level, a design abstraction which models the synchronous circuit into flow of digital signals between registers and the logical operations performed on them), then from RTL into gates (during synthesis), and then it infers what resources of the FPGA should it use and how to link them in the physical

FPGA device (during implementation). The implementation step is where it matters what specific FPGA chip will be used.

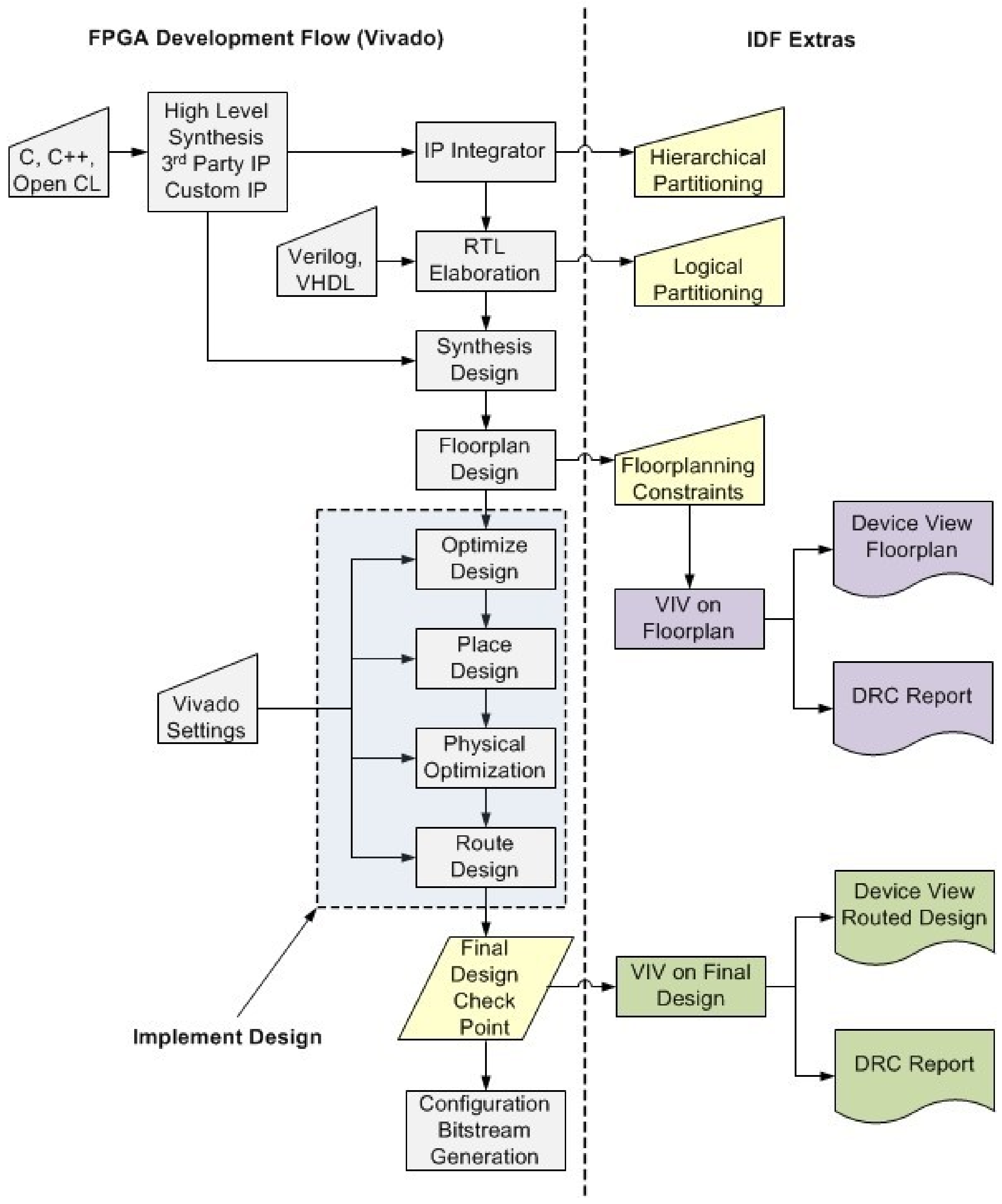


Figure 7. FPGA development flow

Below, there is an FPGA board we will use further, to exemplify digital circuits.

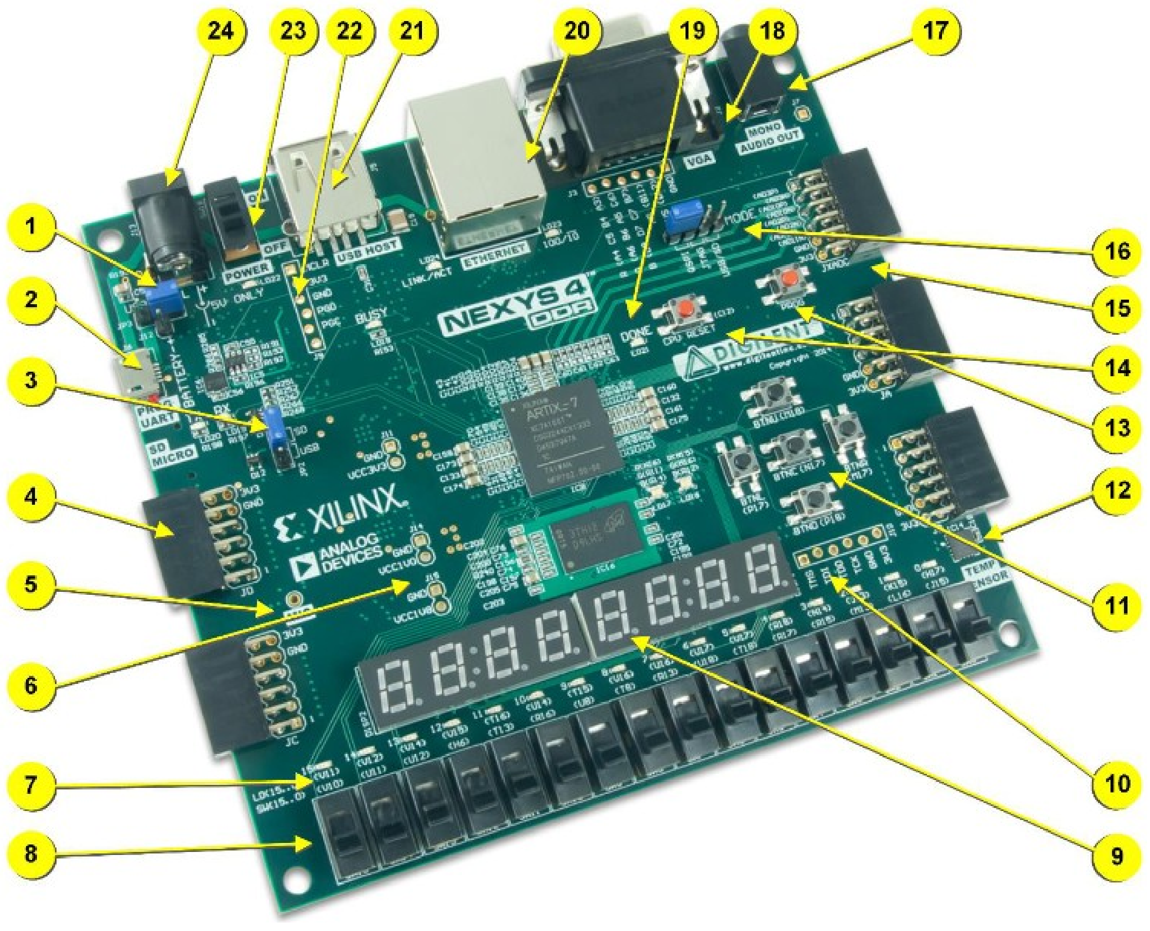


Figure 8. The Nexys 4 DDR FPGA board

Callout	Component Description	Callout	Component Description
1	Power select jumper and battery header	13	FPGA configuration reset button
2	Shared UART/ JTAG USB port	14	CPU reset button (for soft cores)
3	External configuration jumper (SD / USB)	15	Analog signal Pmod port (XADC)
4	Pmod port(s)	16	Programming mode jumper
5	Microphone	17	Audio connector
6	Power supply test point(s)	18	VGA connector
7	LEDs (16)	19	FPGA programming done LED
8	Slide switches	20	Ethernet connector
9	Eight digit 7-seg display	21	USB host connector
10	JTAG port for (optional) external cable	22	PIC24 programming port (factory use)
11	Five pushbuttons	23	Power switch
12	Temperature sensor	24	Power jack

Figure 9. Nexys 4 DDR board features

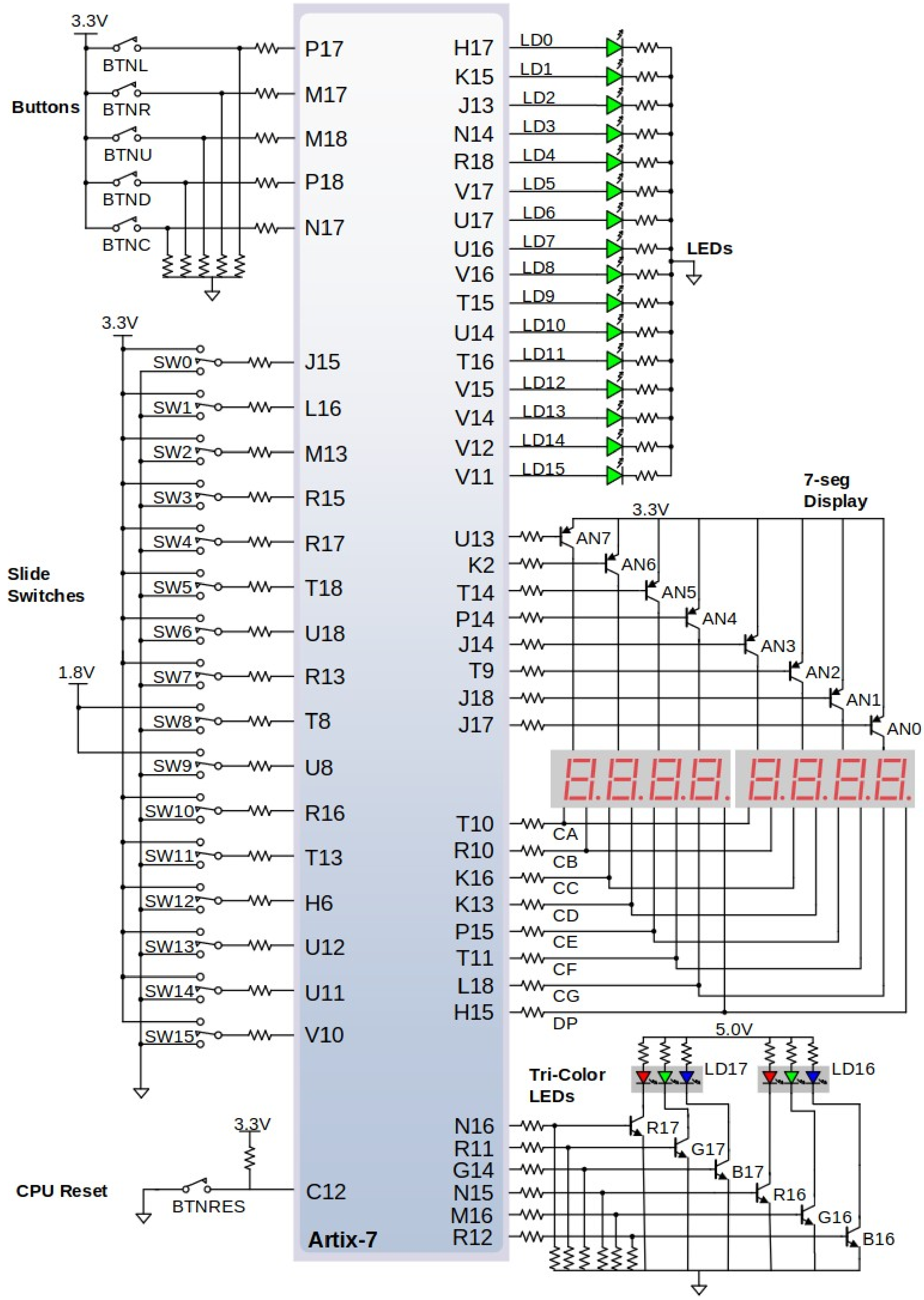

3.3V
Buttons
BTNL
BTNR
BTNU
BTND
BTNC
P17
M17
M18
P18
N17
Slide Switches
SW0
SW1
SW2
SW3
SW4
SW5
SW6
SW7
SW8
SW9
SW10
SW11
SW12
SW13
SW14
SW15
1.8V
J15
L16
M13
R15
R17
T18
U18
R13
T8
U8
R16
T13
H6
U12
U11
V10
CPU Reset
BTNRES
C12
Artix-7
H17
K15
J13
N14
R18
V17
U17
U16
V16
T15
U14
T16
V15
V14
V12
V11
LD0
LD1
LD2
LD3
LD4
LD5
LD6
LD7
LD8
LD9
LD10
LD11
LD12
LD13
LD14
LD15
LEDs
7-seg Display
U13
K2
T14
P14
J14
T9
J18
J17
AN7
AN6
AN5
AN4
AN3
AN2
AN1
AN0
T10
R10
K16
K13
P15
T11
L18
H15
CA
CB
CC
CD
CE
CF
CG
DP
5.0V
Tri-Color LEDs
LD17
LD16
N16
R11
G14
N15
M16
R12
R17
G17
B17
R16
G16
B16

Figure 10. GPIO devices on the Nexys4 DDR FPGA board

2. Zero-loop systems

2.1 Prerequisites

2.1.1 Binary and Boolean Logic

Boolean algebra is a branch of algebra, where the values of the variables are truth values (true and false) usually coded as 1 and 0 and uses logical operators such as AND (conjunction), OR (disjunction), NOT (negation). It was introduced by English mathematician George Boole in the book "The Mathematical Analysis of Logic" in 1847.

A logical operation is a function of two variables and may be expressed using a truth table as below:

Table 2. A (general) logic operation's truth table

A	B	OP (A, B)
FALSE	FALSE	?
FALSE	TRUE	??
TRUE	FALSE	???
TRUE	TRUE	????

There are 16 dual-input single-output logical operations, 3 of which are most used (hence named). The 16 number comes from the output: there are 4 bits of 0 or 1 (one for every combination of A and B), therefore, there are 16 different output configurations => 16 different gates. Their truth table is as below:

Table 3. Truth tables for AND, OR and NOT operations

A	B	A and B	A or B	Not A
FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE	FALSE
TRU	TRUE	TRUE	TRUE	FALSE

E				

Table 4. Truth tables for AND, OR and NOT operations (seen as 1-bit operations)

A	B	A and B	A or B	Not A
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

2.1.2 Digital Gates

The electronic circuits used to implement Boolean logic are the logical gates. Nowadays, all gates are made of transistors (a semiconductor device used to amplify or switch electrical signals and power). For example, the NOT gate is made of two CMOS transistors (a p-channel and n-channel MOS transistor), as seen in Figure 4 below.

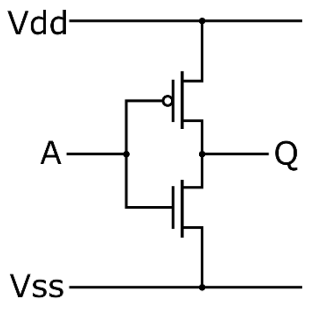
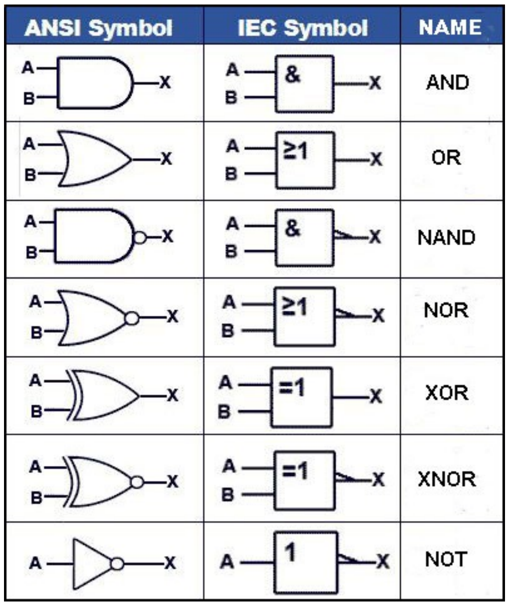


Figure 11. NOT gate ("inverter") made of one pMOS (top) and one nMOS (bottom) transistor.

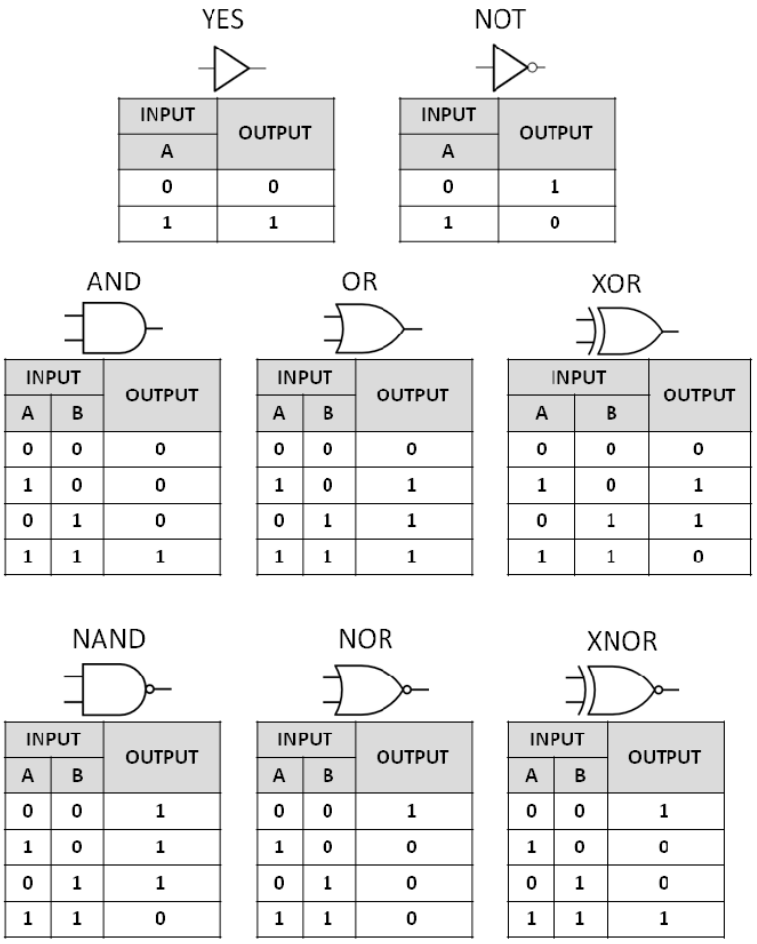
Vdd is usually at least 1.8V over Vss, and Vss is usually ground. When A is "low," pMOS transistor conducts current and draws Q close to Vdd ("high") whereas nMOS is not conducting. When A is "high" level, nMOS conducts and ties Q to the Vss level ("low").

Engineers use the symbols of such gates, in logic schematics; these symbols, are ratified by international standards as seen in Figure 5.



ANSI Symbol	IEC Symbol	NAME
A, B — X	A, B — & — X	AND
A, B — X	A, B — ≥1 — X	OR
A, B — X	A, B — & — X	NAND
A, B — X	A, B — ≥1 — X	NOR
A, B — X	A, B — =1 — X	XOR
A, B — X	A, B — =1 — X	XNOR
A — X	A — 1 — X	NOT

Figure 12. ANSI / IEC [6.] (right) and MIL-STD-806B [7.] (left) symbols foremost common 7/16 dual-input logic gates (elementary), with their names.



YES

INPUT	OUTPUT
A	
0	0
1	1

NOT

INPUT	OUTPUT
A	
0	1
1	0

AND

INPUT		OUTPUT
A	B	
0	0	0
1	0	0
0	1	0
1	1	1

OR

INPUT		OUTPUT
A	B	
0	0	0
1	0	1
0	1	1
1	1	1

XOR

INPUT		OUTPUT
A	B	
0	0	0
1	0	1
0	1	1
1	1	0

NAND

INPUT		OUTPUT
A	B	
0	0	1
1	0	1
0	1	1
1	1	0

NOR

INPUT		OUTPUT
A	B	
0	0	1
1	0	0
0	1	0
1	1	0

XNOR

INPUT		OUTPUT
A	B	
0	0	1
1	0	0
0	1	0
1	1	1

Figure 13. The truth tables for the most commonly used logic gates.

All the 2-input gates are called elementary (or basic) gates. Similarly, for all circuits that are expressed in a iterative or recursive way, the very first instance is called “elementary”.

NOR and NAND gates are called universal gates, because all other 2-input gates (and therefore, N-input gates) may be implemented using any one of them. For example:

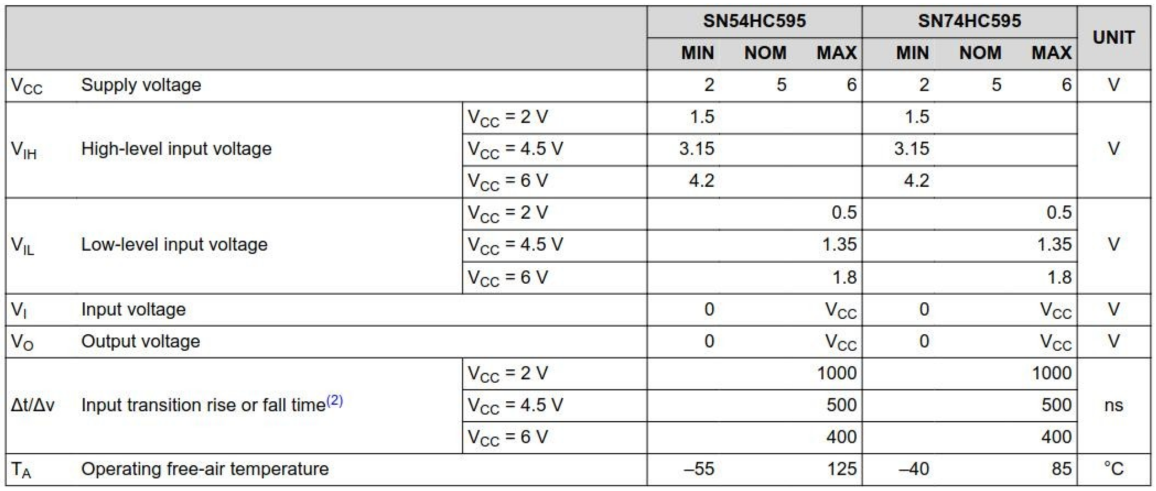
- One NOT gate may be made of one NOR or NAND where both inputs are tied from the same NOT input.
- One AND gate is made of NAND + AND, so three NAND gates are enough.
- One OR gate is made of NOR + NOT, so three NOR gates are enough.
- XOR gate is made of one NAND, one NOT and one AND gate.

Last, a NAND gate may be transformed into NOR and vice versa, by using DeMorgan:

NOT (A OR B) = NOT(A) AND NOT(B)
NOT (A AND B) = NOT(A) OR NOT(B)

2.1.3 Voltage levels (VI / VO)

Table 5. Recommended operating conditions for 54HC595 / 74HC595 ICs (a common 74HCxx circuit)



			SN54HC595			SN74HC595			UNIT
			MIN	NOM	MAX	MIN	NOM	MAX	
V_{CC}	Supply voltage		2	5	6	2	5	6	V
V_{IH}	High-level input voltage	V_{CC} = 2 V	1.5			1.5			V
		V_{CC} = 4.5 V	3.15			3.15			
		V_{CC} = 6 V	4.2			4.2			
V_{IL}	Low-level input voltage	V_{CC} = 2 V			0.5			0.5	V
		V_{CC} = 4.5 V			1.35			1.35	
		V_{CC} = 6 V			1.8			1.8	
V_I	Input voltage		0		V_{CC}	0		V_{CC}	V
V_O	Output voltage		0		V_{CC}	0		V_{CC}	V
$\Delta t/\Delta v$	Input transition rise or fall time[2]	V_{CC} = 2 V			1000			1000	ns
		V_{CC} = 4.5 V			500			500	
		V_{CC} = 6 V			400			400	
T_A	Operating free-air temperature		–55		125	–40		85	°C

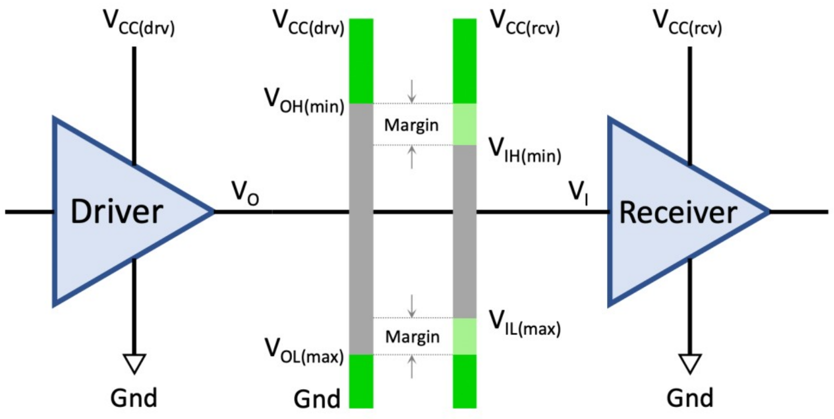


Figure 14. Voltage levels: output requirements vs. input requirements.

- High state/Logic 1: $V_O > V_{OH}$ (V_{OH} is minimum output voltage for state HIGH or logic 1)
- Low state/Logic 0: $V_O < V_{OL}$ (V_{OL} is maximum output voltage for low/0 state)
- High state/Logic 1: $V_I > V_{IH}$ (V_{IH} is minimum voltage interpreted as logic high/1)
- Low state/Logic 0: $V_I < V_{IL}$ (V_{IL} is maximum voltage interpreted as logic low/0)
- “Noise Margin”- allows logic level to be correct even if some voltage variation occurs
- High state/Logic 1: $V_{OH} = V_{IH}$ + Vnoise (noise does not pull voltage below threshold)
- Low state/Logic 0: $V_{OL} = V_{IL}$ – Vnoise (noise does not pull voltage above threshold)

In digital design vocabulary, a driver is similar in function to an electrical driver: it just amplifies the current (keeps the voltage the same, therefore, keeps the logic value on the output, similar to input)

2.1.4 Logic gates components

The logic gates may be composed of various devices depending on the field used in. First ones were mechanical, then electro-mechanical, and nowadays electronic. One common implementation in electronics is with MOSFET transistors.

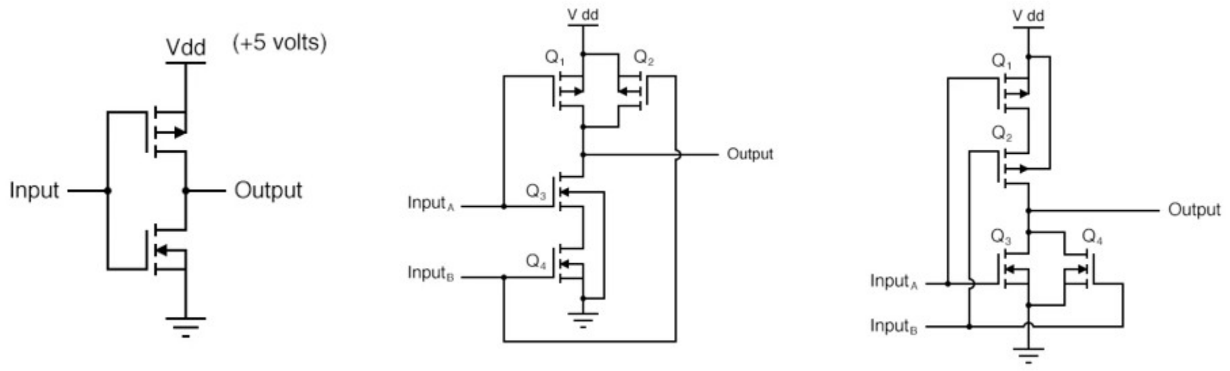


Figure 15. MOSFET implementations of common logic gates. Left (NOT), middle (NAND), and right (NOR)

Regarding the size in MOS transistors we have:

- NOT gate, made of 2 transistors
- NAND, NOR gates made of 4 transistors
- AND, OR gates made of 6 transistors (NAND/NOR + NOT)
- XOR gate made of 8 transistors

Logic gates are still available as integrated circuits using the 74xx family, for example the 74HC00 implements 6x NOT gates in one DIP14 or SOIC14 package (6 inputs, 6 outputs, one Vcc and one GND). There are circuits for all common 2-input gates (74HC00 for NAND, 74HC02 for NOR) and common 3-input gates (74HC10 for NAND, 74HC11 for AND).

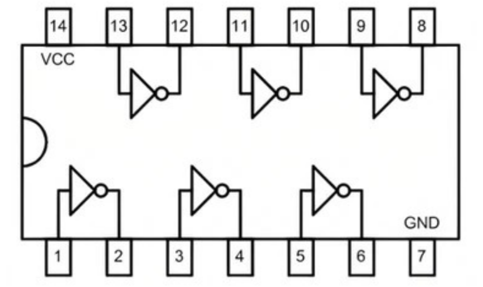


Figure 16. The 74HC04 hex-inverter circuit

2.1.5 Complexity in digital circuits

A complex circuit is a circuit that has spatial complexity (structure) or behavior complexity or a combination of the two.

To express spatial complexity, two metrics are used:

- the SIZE (S) of the circuit (the number of elementary gates)
- the DEPTH (D) of the circuit (the largest number of elementary gates passed through, when the signal goes from any input to any output)

An elementary gate (2-input, 1-output) has the S = 1 and D = 1.

2.1.6 Complexity classes

A more in-depth talk on complexity for algorithms may be found in [941.], below is an extract of how complexity applies to circuits. Complexity is usually expressed as Big-O notation. The complexity of a function f is decided by finding another function g, which asymptotically bounds the f function.

Mathematically, this is expressed as: if f(n) has the same complexity as g(n), then from n = k onwards, the c*g(n) >= f(n), where k is a point < infinity and c is a constant

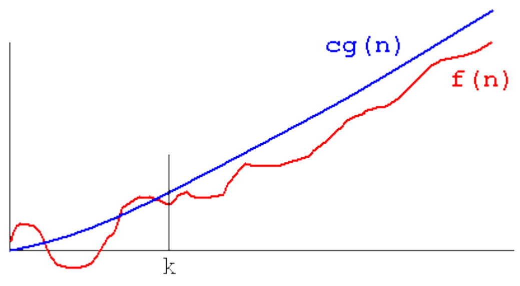


Figure 17. Representing a g(n) function which asymptotically bound f(n) function.

To learn more about Big-Oh, Big-Theta and Big-Omega and their small variants, see [10.10.]. A list of common complexity classes is listed below. O(1) means constant time, and is the best one can hope for: the algorithm runtime will not increase with data increase, the circuit will not increase its size no matter the number of inputs etc.

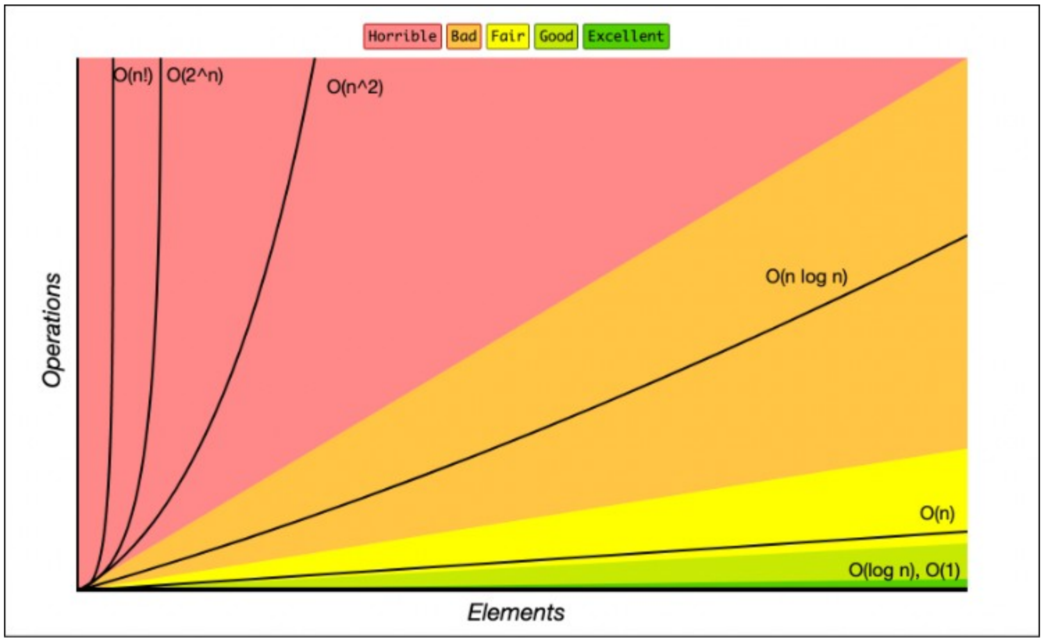


*Figure 18. The O-notation complexity increases with the number of elements processed. O(1) and O(logn) are usually excellent complexities, O(n) is fair, and O(n*logn) is usually considered almost decent in algorithms and circuits*

To better understand the complexity difference between O(logN) and O(N) we propose the next game: one thinks at a number from 0 to 100. Assuming another one tries to guess the number, with hints of “my number is higher” or “my number is lower”:

- in O(N) algorithm (brute forcing all values) one will guess in at most 100 steps

- whereas in O(logN) algorithm, same task will take at most 7 steps (assuming log2).

This is not very impressive, but as one goes further (towards infinity), the advantage will become obvious. Assume the same game, but with numbers from 0 to 4 billion:

- in O(N) algorithm it will take at most 4 billion steps
- whereas in O(logN) it will take at most 32 steps!

This is the power of the logarithm: it goes to infinity with N going to infinity, but much slower.

To better understand how this applies to circuits, imagine an N-input AND gate, implemented with N-1 elementary gates, arranged in logN layers as below:

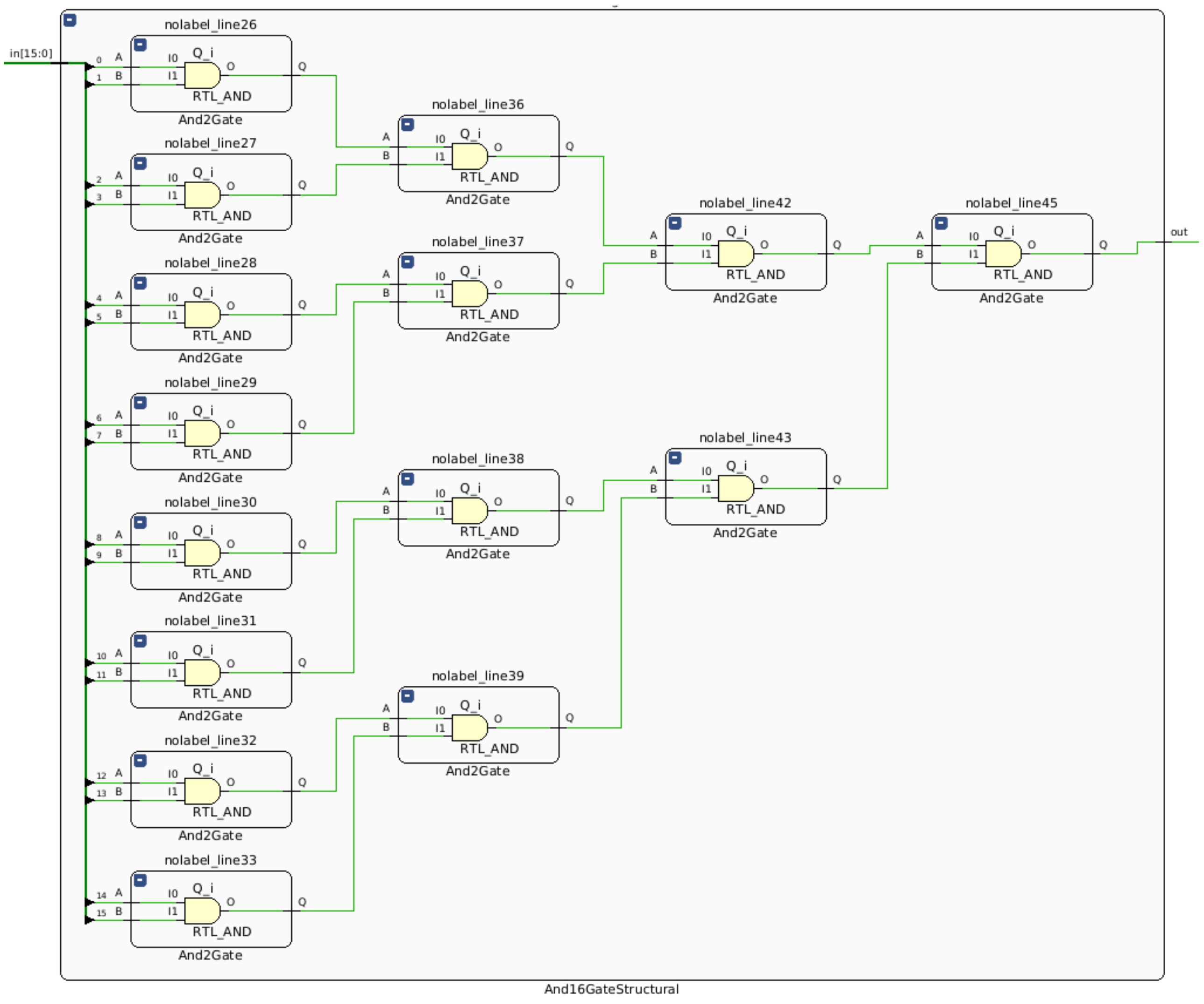
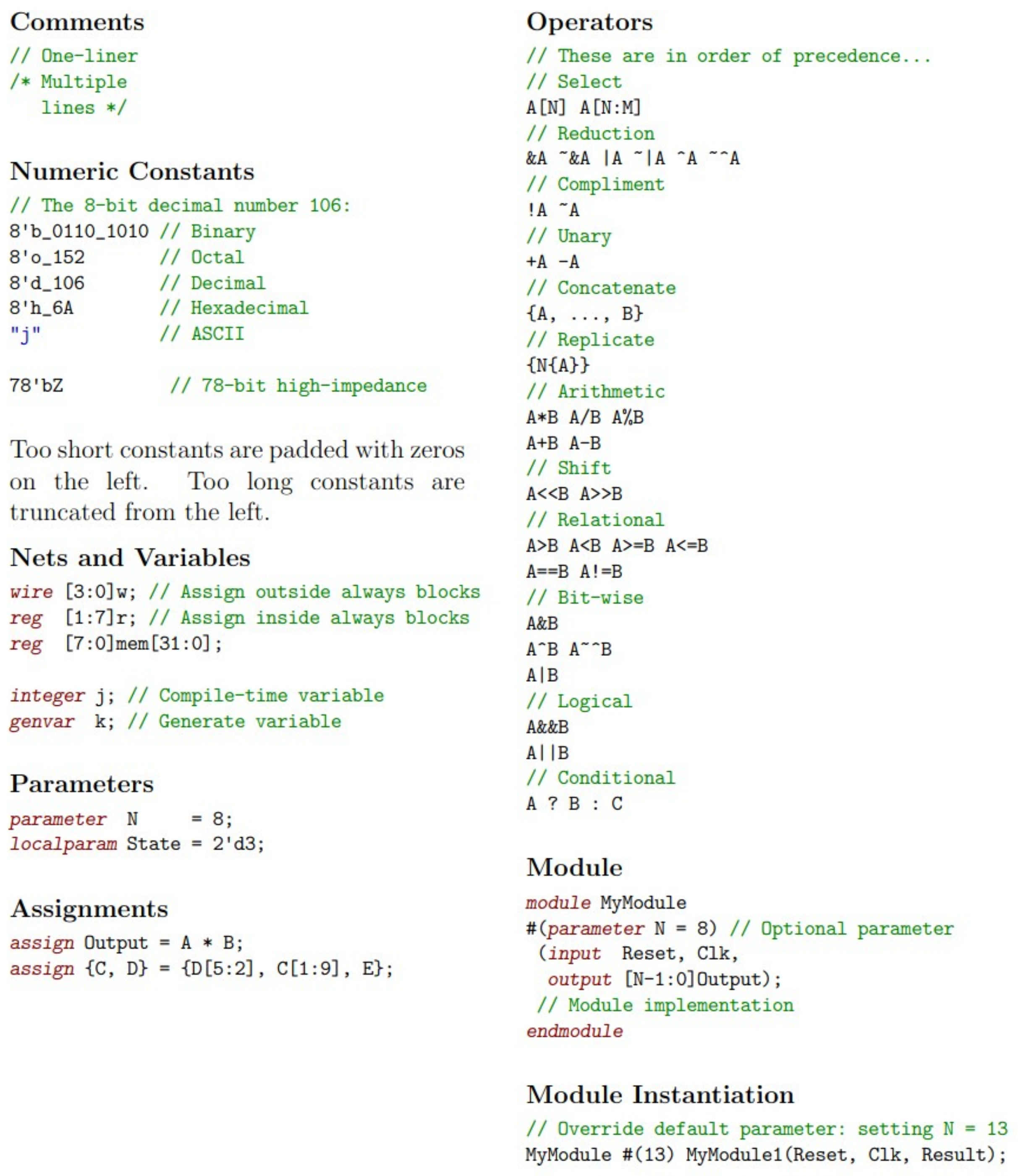


Figure 19. A 16-input AND gate built from 2-input AND gates.

It is quite common to have a logN depth (and acceptable); a size of O(N) is also fair. There are circuits that can implement any logic function (eg. a decoder to output all minterms + specific AND gates to combine the specific minters) however, the price paid is 2**N size or even worse, N*2**N size: this is why the process of minimization (expressing the same logic function with as few number of elementary gates, as possible) is a mandatory study in this field. Fortunately, for most common circuits, the synthesis toolchain will take care of that, but applying an optimization step after inferring the logic function from the behavioral description.

2.1.7 Verilog syntax

S. Winberg and J. Taylor [9.] summarized the most important syntax features of Verilog language in their cheat sheet:



Comments

```
// One-liner
/* Multiple
   lines */
```

Numeric Constants

```
// The 8-bit decimal number 106:
8'b_0110_1010 // Binary
8'o_152       // Octal
8'd_106       // Decimal
8'h_6A        // Hexadecimal
"j"           // ASCII

78'bZ          // 78-bit high-impedance
```

Too short constants are padded with zeros on the left. Too long constants are truncated from the left.

Nets and Variables

```
wire [3:0]w; // Assign outside always blocks
reg  [1:7]r; // Assign inside always blocks
reg  [7:0]mem[31:0];

integer j; // Compile-time variable
genvar  k; // Generate variable
```

Parameters

```
parameter  N     = 8;
localparam State = 2'd3;
```

Assignments

```
assign Output = A * B;
assign {C, D} = {D[5:2], C[1:9], E};
```

Operators

```
// These are in order of precedence...
// Select
A[N] A[N:M]
// Reduction
&A ~&A |A ~|A ^A ~^A
// Compliment
!A ~A
// Unary
+A -A
// Concatenate
{A, ..., B}
// Replicate
{N{A}}
// Arithmetic
A*B A/B A%B
A+B A-B
// Shift
A<<B A>>B
// Relational
A>B A<B A>=B A<=B
A==B A!=B
// Bit-wise
A&B
A^B A~^B
A|B
// Logical
A&&B
A||B
// Conditional
A ? B : C
```

Module

```
module MyModule
#(parameter N = 8) // Optional parameter
 (input  Reset, Clk,
  output [N-1:0]Output);
 // Module implementation
endmodule
```

Module Instantiation

```
// Override default parameter: setting N = 13
MyModule #(13) MyModule1(Reset, Clk, Result);
```

Figure 20. Verilog syntax cheat sheet (1/2)

Case

```
always @(*) begin
 case(Mux)
  2'd0: A = 8'd9;
  2'd1,
  2'd3: A = 8'd103;
  2'd2: A = 8'd2;
  default:;
 endcase
end

always @(*) begin
 casex(Decoded)
  4'b1xxx: Encoded = 2'd0;
  4'b01xx: Encoded = 2'd1;
  4'b001x: Encoded = 2'd2;
  4'b0001: Encoded = 2'd3;
  default: Encoded = 2'd0;
 endcase
end
```

Synchronous

```
always @(posedge Clk) begin
 if(Reset) B <= 0;
 else      B <= B + 1'b1;
end
```

Loop

```
always @(*) begin
 Count = 0;
 for(j = 0; j < 8; j = j+1)
  Count = Count + Input[j];
end
```

Function

```
function [6:0]F;
 input [3:0]A;
 input [2:0]B;
 begin
  F = {A+1'b1, B+2'd2};
 end
endfunction
```

Generate

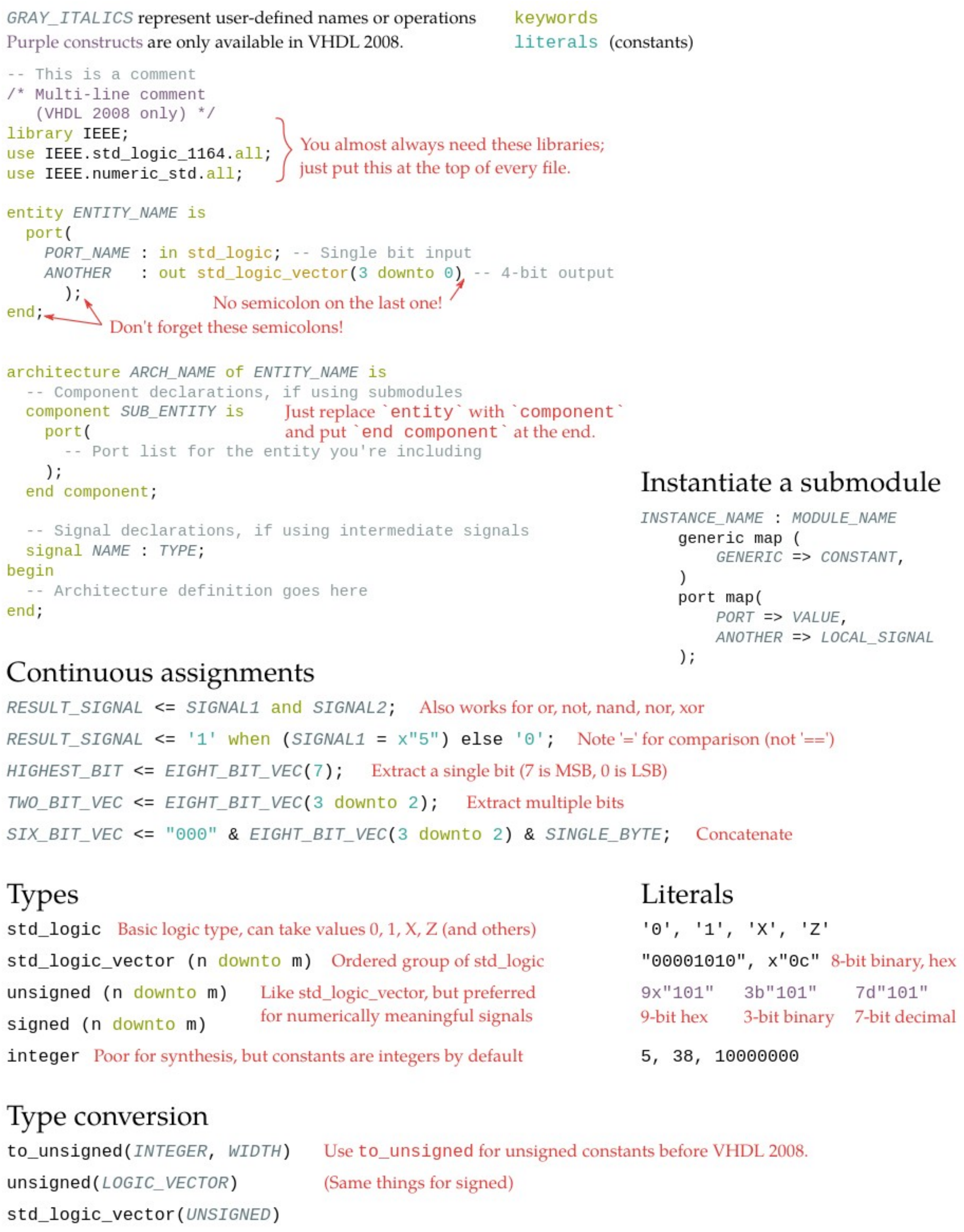
```
genvar j;
wire [12:0]Output[19:0];

generate
 for(j = 0; j < 20; j = j+1)
 begin: Gen_Modules
  MyModule #(13) MyModule_Instance(
   Reset, Clk,
   Output[j]
  );
 end
endgenerate
```

State Machine

```
reg   [1:0]State;
localparam Start = 2'b00;
localparam Idle  = 2'b01;
localparam Work  = 2'b11;
localparam Done  = 2'b10;

reg tReset;

always @(posedge Clk) begin
 tReset <= Reset;

 if(tReset) begin
  State <= Start;

 end else begin
  case(State)
   Start: begin
    State <= Idle;
   end
   Idle: begin
    State <= Work;
   end
   Work: begin
    State <= Done;
   end
   Done: begin
    State <= Idle;
   end
   default:;
  endcase
 end
end
```

Figure 21. Verilog syntax cheat sheet (2/2)

2.1.7 VHDL syntax



GRAY_ITALICS represent user-defined names or operations   keywords
Purple constructs are only available in VHDL 2008.   literals (constants)

```
-- This is a comment
/* Multi-line comment
   (VHDL 2008 only) */
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
```

You almost always need these libraries; just put this at the top of every file.

```
entity ENTITY_NAME is
  port(
    PORT_NAME : in std_logic; -- Single bit input
    ANOTHER   : out std_logic_vector(3 downto 0) -- 4-bit output
      );
end;
```

No semicolon on the last one!

Don't forget these semicolons!

```
architecture ARCH_NAME of ENTITY_NAME is
  -- Component declarations, if using submodules
  component SUB_ENTITY is
    port(
      -- Port list for the entity you're including
    );
  end component;

  -- Signal declarations, if using intermediate signals
  signal NAME : TYPE;
begin
  -- Architecture definition goes here
end;
```

Just replace `entity` with `component` and put `end component` at the end.

Instantiate a submodule

```
INSTANCE_NAME : MODULE_NAME
    generic map (
        GENERIC => CONSTANT,
    )
    port map(
        PORT => VALUE,
        ANOTHER => LOCAL_SIGNAL
    );
```

Continuous assignments

`RESULT_SIGNAL <= SIGNAL1 and SIGNAL2;`	Also works for or, not, nand, nor, xor
`RESULT_SIGNAL <= '1' when (SIGNAL1 = x"5") else '0';`	Note '=' for comparison (not '==')
`HIGHEST_BIT <= EIGHT_BIT_VEC(7);`	Extract a single bit (7 is MSB, 0 is LSB)
`TWO_BIT_VEC <= EIGHT_BIT_VEC(3 downto 2);`	Extract multiple bits
`SIX_BIT_VEC <= "000" & EIGHT_BIT_VEC(3 downto 2) & SINGLE_BYTE;`	Concatenate

Types

`std_logic`	Basic logic type, can take values 0, 1, X, Z (and others)
`std_logic_vector (n downto m)`	Ordered group of std_logic
`unsigned (n downto m)`	Like std_logic_vector, but preferred for numerically meaningful signals
`signed (n downto m)`	
`integer`	Poor for synthesis, but constants are integers by default

Literals

`'0', '1', 'X', 'Z'`

`"00001010", x"0c"` 8-bit binary, hex

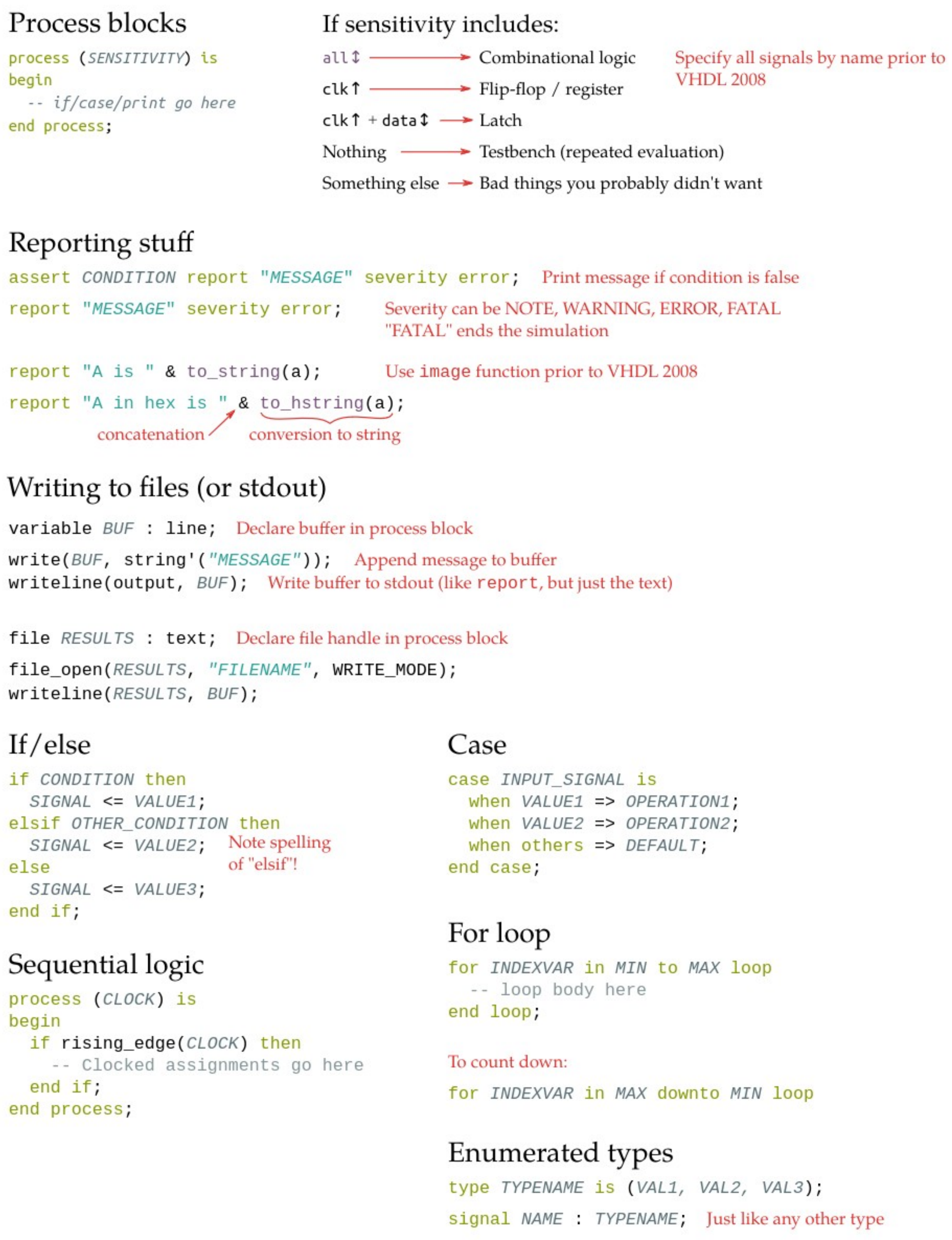
`9x"101"`	`3b"101"`	`7d"101"`
9-bit hex	3-bit binary	7-bit decimal

`5, 38, 10000000`

Type conversion

`to_unsigned(INTEGER, WIDTH)`	Use `to_unsigned` for unsigned constants before VHDL 2008.
`unsigned(LOGIC_VECTOR)`	(Same things for signed)
`std_logic_vector(UNSIGNED)`	

Figure 22. VHDL cheat sheet (1/2)



Process blocks

```
process (SENSITIVITY) is
begin
  -- if/case/print go here
end process;
```

If sensitivity includes:

all↕ → Combinational logic

clk↑ → Flip-flop / register

clk↑ + data↕ → Latch

Nothing → Testbench (repeated evaluation)

Something else → Bad things you probably didn't want

Specify all signals by name prior to VHDL 2008

Reporting stuff

`assert CONDITION report "MESSAGE" severity error;` Print message if condition is false

`report "MESSAGE" severity error;` Severity can be NOTE, WARNING, ERROR, FATAL "FATAL" ends the simulation

`report "A is " & to_string(a);` Use `image` function prior to VHDL 2008

`report "A in hex is " & to_hstring(a);`

concatenation — conversion to string

Writing to files (or stdout)

`variable BUF : line;` Declare buffer in process block

`write(BUF, string'("MESSAGE"));` Append message to buffer

`writeline(output, BUF);` Write buffer to stdout (like `report`, but just the text)

`file RESULTS : text;` Declare file handle in process block

```
file_open(RESULTS, "FILENAME", WRITE_MODE);
writeline(RESULTS, BUF);
```

If/else

```
if CONDITION then
  SIGNAL <= VALUE1;
elsif OTHER_CONDITION then
  SIGNAL <= VALUE2;
else
  SIGNAL <= VALUE3;
end if;
```

Note spelling of "elsif"!

Case

```
case INPUT_SIGNAL is
  when VALUE1 => OPERATION1;
  when VALUE2 => OPERATION2;
  when others => DEFAULT;
end case;
```

Sequential logic

```
process (CLOCK) is
begin
  if rising_edge(CLOCK) then
    -- Clocked assignments go here
  end if;
end process;
```

For loop

```
for INDEXVAR in MIN to MAX loop
  -- loop body here
end loop;
```

To count down:

```
for INDEXVAR in MAX downto MIN loop
```

Enumerated types

```
type TYPENAME is (VAL1, VAL2, VAL3);
```

`signal NAME : TYPENAME;` Just like any other type

Figure 23. VHDL cheat sheet (2/2)

2.1.8 Simulation

The simulation is

2.1.9 Our first digital circuit – the switch tester

Usually, dev-board switches are of poor quality, therefore, before using them, one might check the state of the switches (whether they are always on or off, hence malfunctioning). The circuit would be driver at the input, driver at the output, such that the on or off state of each switch can be monitored on a correspondent led:

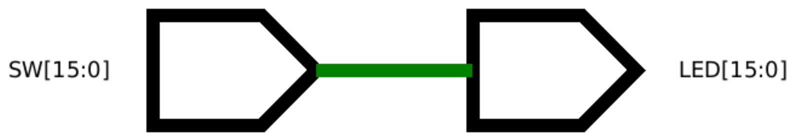
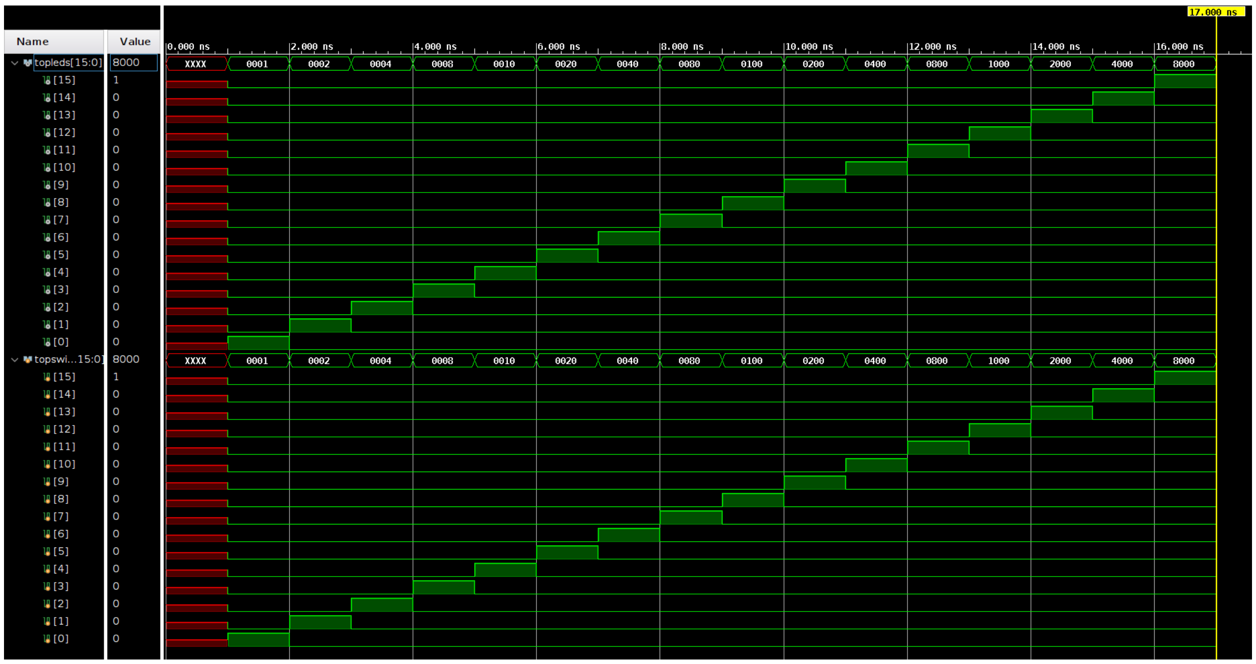
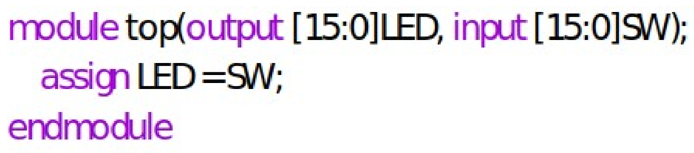


Figure 24. The schematic of a switch checker

A simulation where each switch is switched on then off, should look like the figure below:

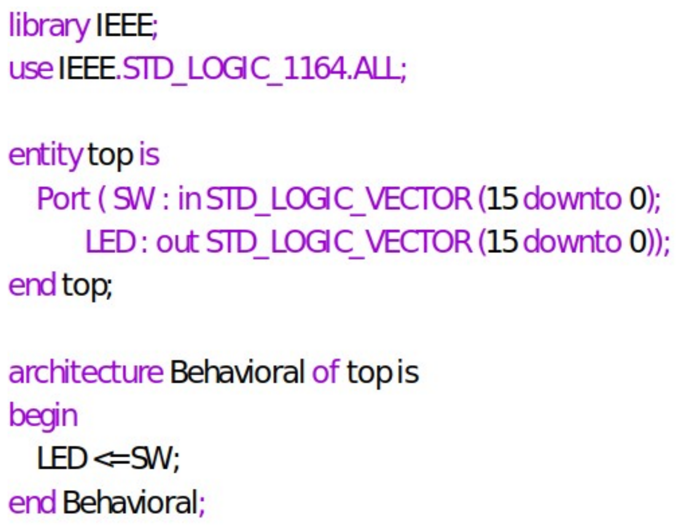


The Verilog and VHDL implementation are as follows:



```verilog
module top(output [15:0]LED, input [15:0]SW);
   assign LED =SW;
endmodule
```

Figure 25. Verilog description of switch-checker



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

entity top is
   Port ( SW : in STD_LOGIC_VECTOR (15 downto 0);
        LED : out STD_LOGIC_VECTOR (15 downto 0));
end top;

architecture Behavioral of top is
begin
   LED <=SW;
end Behavioral;
```

Figure 26. VHDL description of switch-checker

// TODO: Verilog simulation / VHDL?

To express the AND gate behavior in Figure 27 one has to write in Verilog the code or VHDL code below.

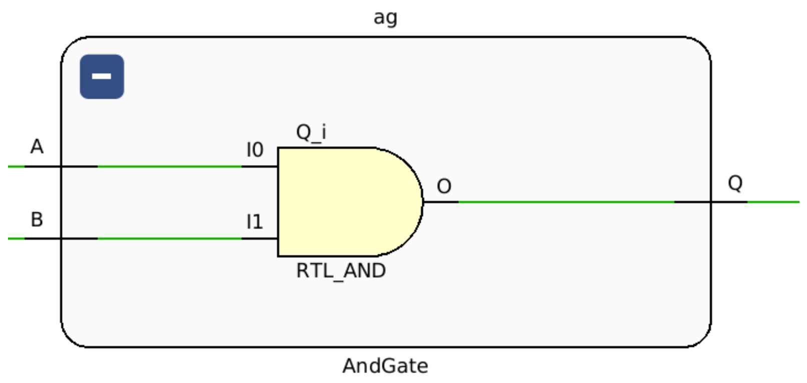


Figure 27. 2-input AND gate

```
module AndGate(input A, input B, output Q);

  assign Q = A & B;

endmodule
```

Figure 28. Verilog code (dataflow) for a 2-input AND gate

```
library IEEE;
use IEEE.std_logic_1164.all;

– Entity declaration
entity andGate is
  port(A : in std_logic;     – AND gate input
       B : in std_logic;     – AND gate input
       Q : out std_logic);   – AND gate output
end andGate;

– Dataflow Modelling Style
– Architecture definition
architecture andLogic of andGate is
 begin
  Q <= A AND B;
end andLogic;
```

Figure 29. Verilog code (dataflow) for a 2-input AND gate

2.2 Theory & Exercises

2.2.1 Multiple-input gates

Using only elementary (2-input) gates, create a 4-input AND gate. Write code in both dataflow and structural manner. Describe the circuit in both in Verilog and VHDL language.

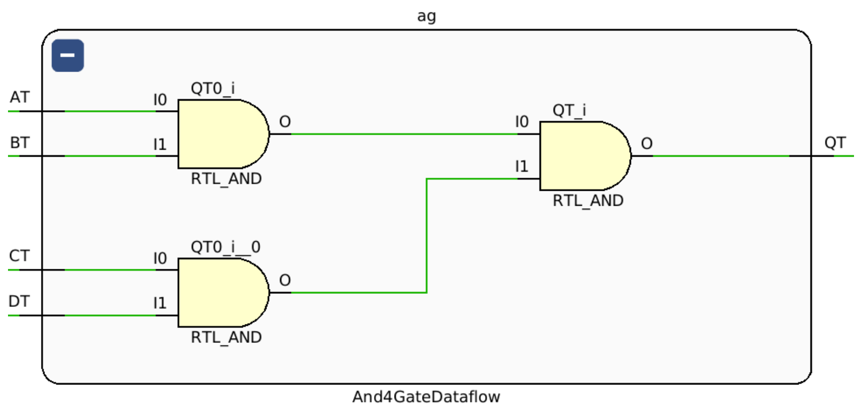
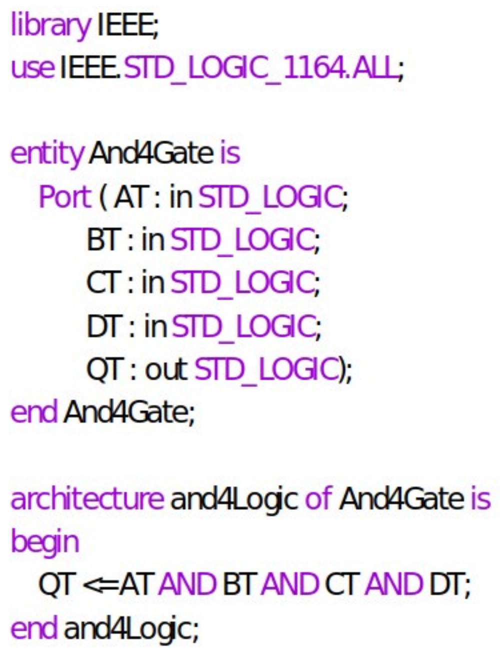


Figure 30. 4-input AND gate from 2-input AND gates (dataflow)

```
module And4GateDataflow(input AT, input BT, input CT, input DT, output QT);
  assign QT =AT & BT & CT & DT;
endmodule
```

Figure 31. Verilog description for 4-input AND (dataflow)



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

entity And4Gate is
   Port ( AT : in STD_LOGIC;
          BT : in STD_LOGIC;
          CT : in STD_LOGIC;
          DT : in STD_LOGIC;
          QT : out STD_LOGIC);
end And4Gate;

architecture and4Logic of And4Gate is
begin
   QT <= AT AND BT AND CT AND DT;
end and4Logic;
```

Figure 32. VHDL description for 4-input AND (dataflow)

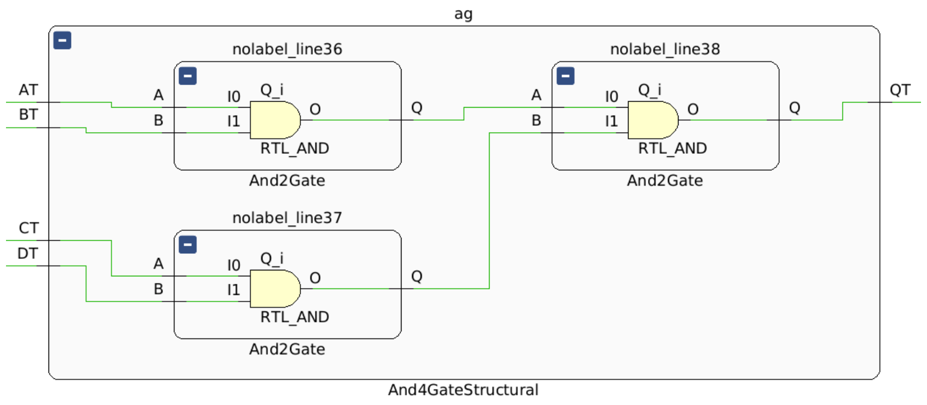
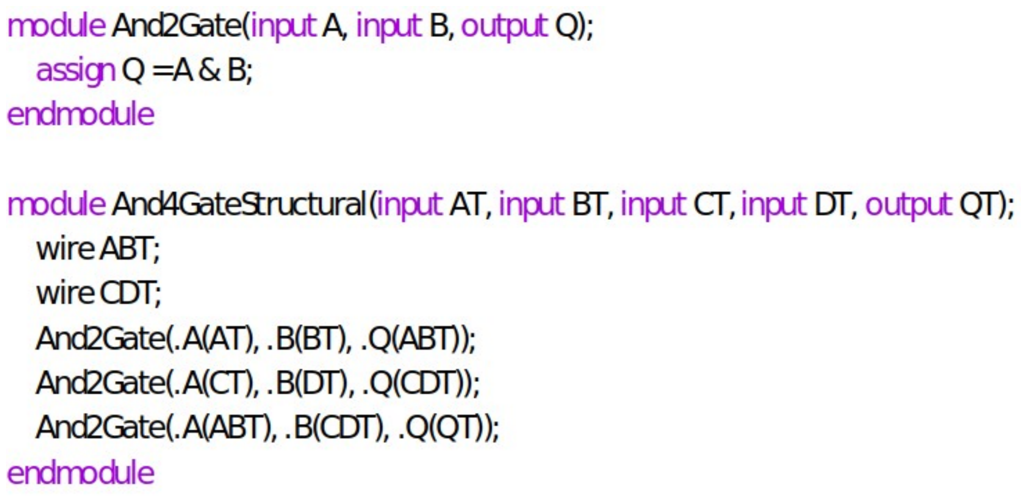


Figure 33. 4-input AND gate from 2-input AND gates (structural)



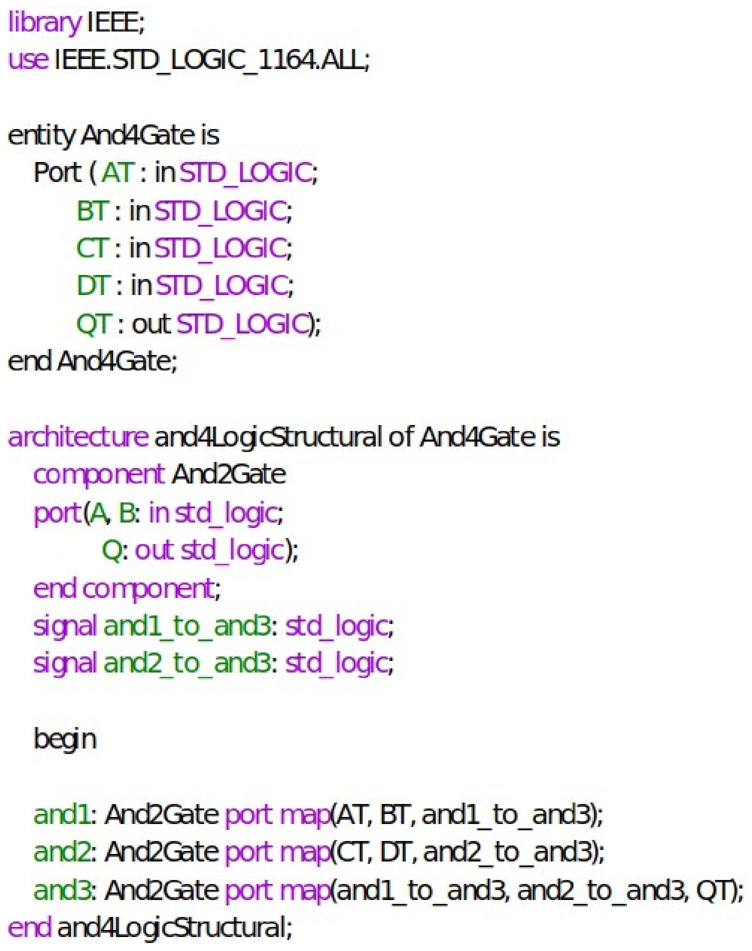
```
module And2Gate(input A, input B, output Q);
   assign Q =A & B;
endmodule

module And4GateStructural(input AT, input BT, input CT, input DT, output QT);
   wire ABT;
   wire CDT;
   And2Gate(.A(AT), .B(BT), .Q(ABT));
   And2Gate(.A(CT), .B(DT), .Q(CDT));
   And2Gate(.A(ABT), .B(CDT), .Q(QT));
endmodule
```

Figure 34.Verilog description of 4-input AND gate from 2-input AND gates (structural)

A few notes here: assuming N is power of 2, the number of elementary gates for an N-input AND or OR gate is a having log2 N layers each with half of the number of gates or previous layer. For example, an 16-input AND gate requires 4 layers having the N-1 gates, distributed as follows:

- Layer 1: 8 elementary gates (all 16 inputs go into the inputs of the first layer)
- Layer 2: 4 elementary gates
- Layer 3: 2 elementary gates
- Layer 4: 1 elementary gate (which gives the output of the circuit)

Therefore, the SIZE of the circuit is O(N), and the DEPTH of the circuit is O(logN)



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

entity And4Gate is
   Port ( AT : in STD_LOGIC;
          BT : in STD_LOGIC;
          CT : in STD_LOGIC;
          DT : in STD_LOGIC;
          QT : out STD_LOGIC);
end And4Gate;

architecture and4LogicStructural of And4Gate is
   component And2Gate
   port(A, B: in std_logic;
            Q: out std_logic);
   end component;
   signal and1_to_and3: std_logic;
   signal and2_to_and3: std_logic;

   begin

   and1: And2Gate port map(AT, BT, and1_to_and3);
   and2: And2Gate port map(CT, DT, and2_to_and3);
   and3: And2Gate port map(and1_to_and3, and2_to_and3, QT);
end and4LogicStructural;
```

Figure 35. VHDL description of 4-input AND gate from 2-input AND gates (structural)

2.2.2 Elementary multiplexer

A multiplexer is a circuit that outputs one of its inputs, depending on the selection. The elementary multiplexer is draws as follows:

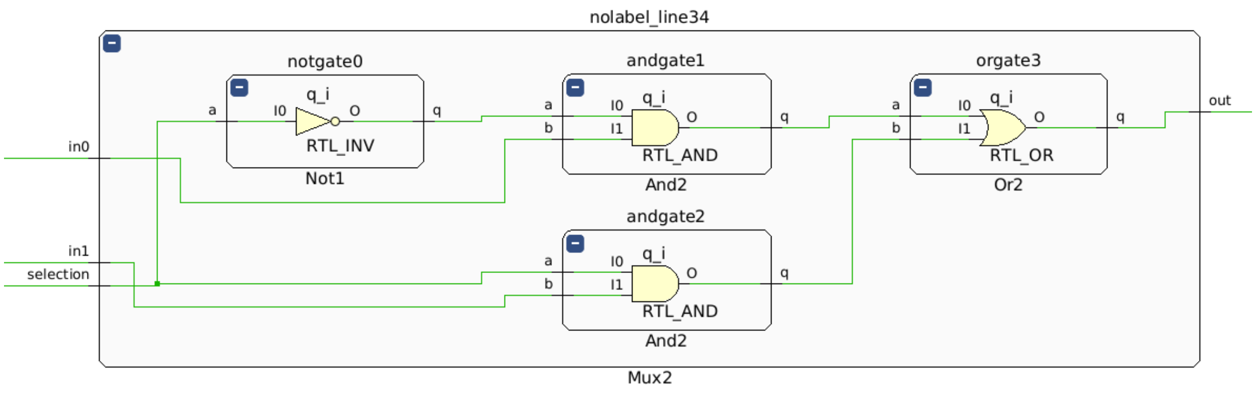
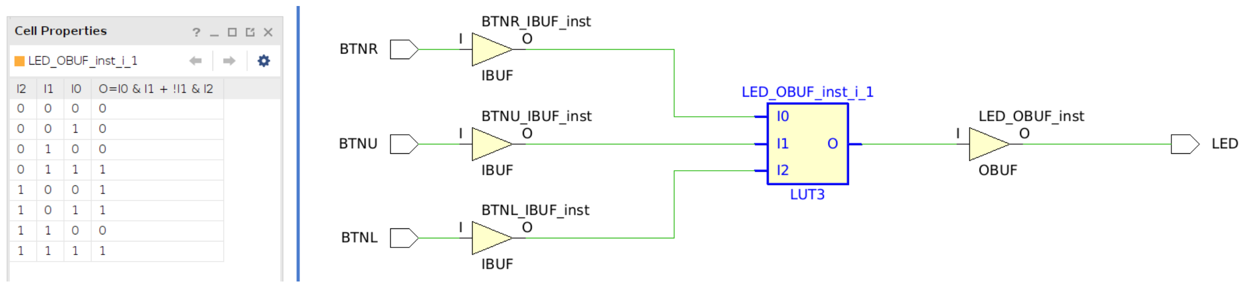


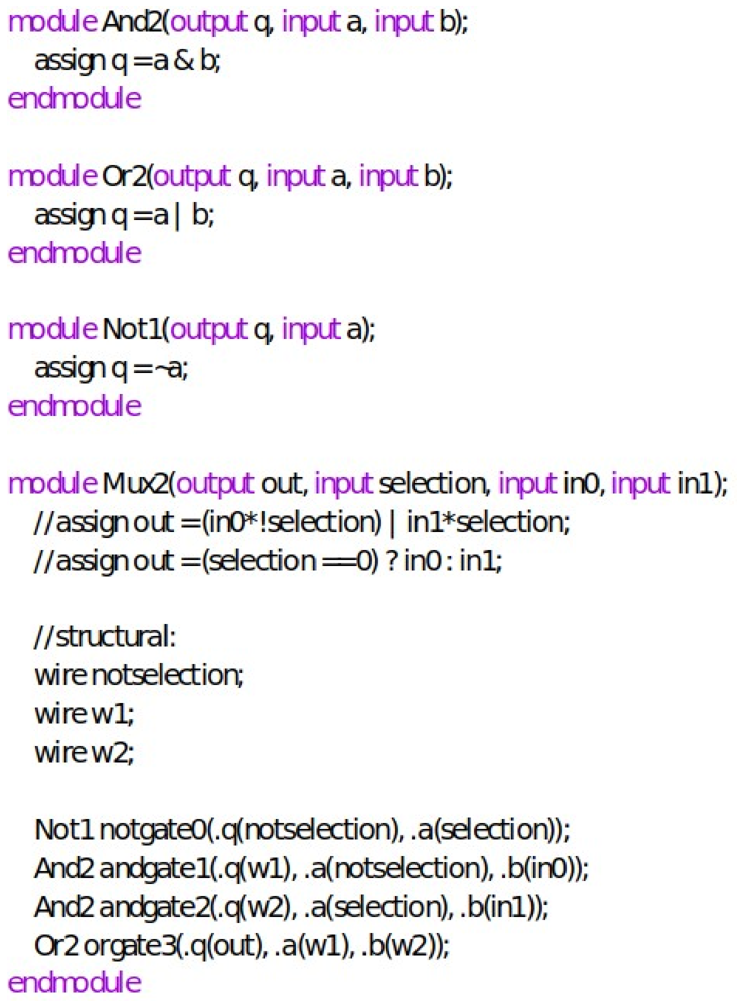
Figure 36. MUX2 implemented with one NOT, two AND, one OR gate(s)

The implementation post-synthesis is done with one LUT, as shown in the



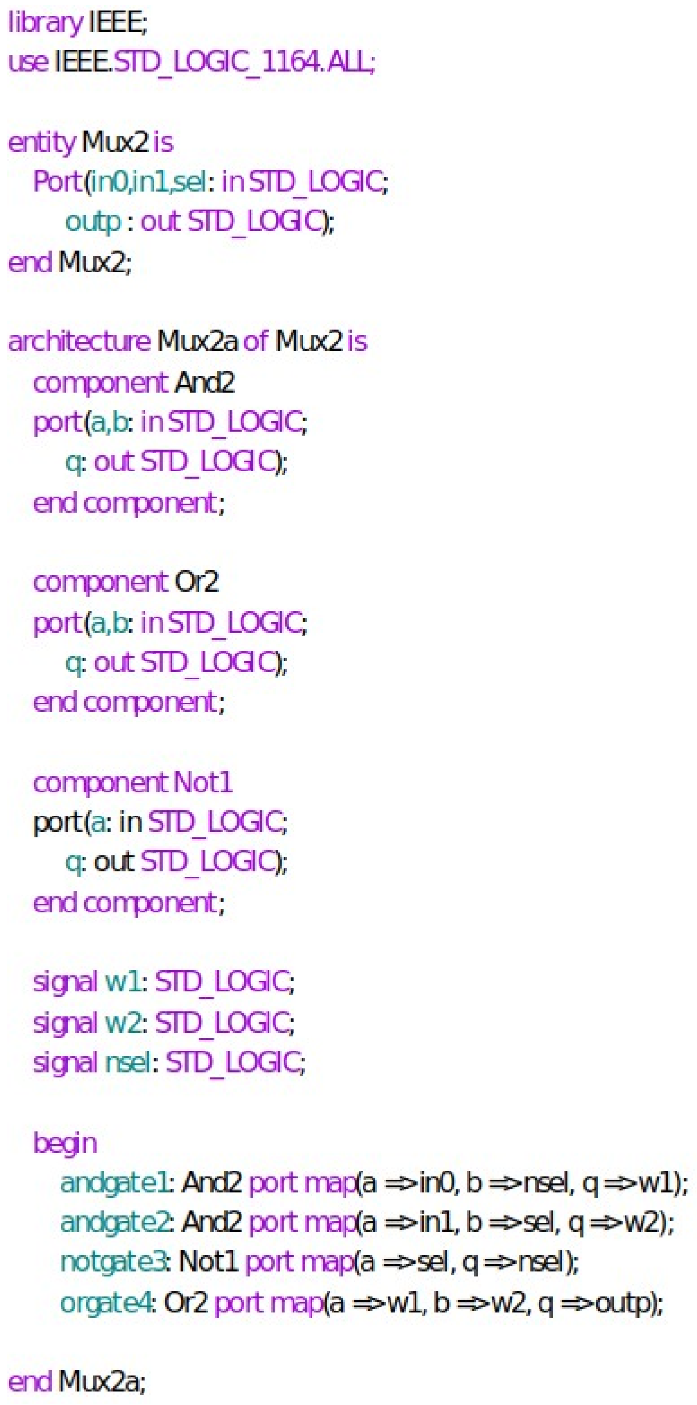
I2	I1	I0	O=I0 & I1 + !I1 & I2
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Figure 37. LUT-based implementation of elementary mux, in FPGA



```
module And2(output q, input a, input b);
   assign q = a & b;
endmodule

module Or2(output q, input a, input b);
   assign q = a | b;
endmodule

module Not1(output q, input a);
   assign q = ~a;
endmodule

module Mux2(output out, input selection, input in0, input in1);
   //assign out = (in0*!selection) | in1*selection;
   //assign out = (selection == 0) ? in0 : in1;

   //structural:
   wire notselection;
   wire w1;
   wire w2;

   Not1 notgate0(.q(notselection), .a(selection));
   And2 andgate1(.q(w1), .a(notselection), .b(in0));
   And2 andgate2(.q(w2), .a(selection), .b(in1));
   Or2 orgate3(.q(out), .a(w1), .b(w2));
endmodule
```

Figure 38. Verilog description of an elementary MUX



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

entity Mux2 is
   Port(in0,in1,sel : in STD_LOGIC;
       outp : out STD_LOGIC);
end Mux2;

architecture Mux2a of Mux2 is
   component And2
   port(a,b: in STD_LOGIC;
       q: out STD_LOGIC);
   end component;

   component Or2
   port(a,b: in STD_LOGIC;
       q: out STD_LOGIC);
   end component;

   component Not1
   port(a: in STD_LOGIC;
       q: out STD_LOGIC);
   end component;

   signal w1: STD_LOGIC;
   signal w2: STD_LOGIC;
   signal nsel: STD_LOGIC;

   begin
      andgate1: And2 port map(a =>in0, b =>nsel, q =>w1);
      andgate2: And2 port map(a =>in1, b =>sel, q =>w2);
      notgate3: Not1 port map(a =>sel, q =>nsel);
      orgate4: Or2 port map(a =>w1, b =>w2, q =>outp);

end Mux2a;
```

Figure 39. VHDL description of an elementary MUX

Exercise: Using elementary MUX (2-input, 1-selection), describe a 4-input MUX. All data is 1-bit wide.
Solution: A hierarchical implementation will be made, with layered MUXes: first layer will have 2x MUXes, and the next layer just 1.

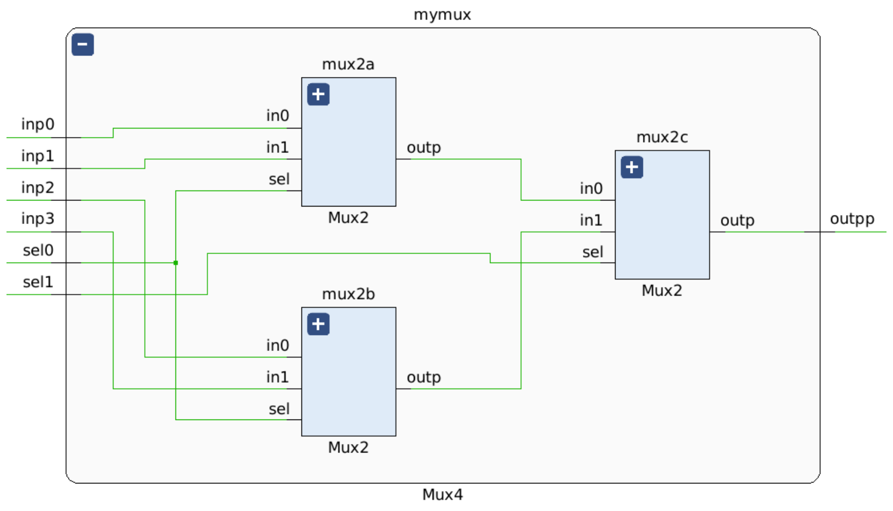
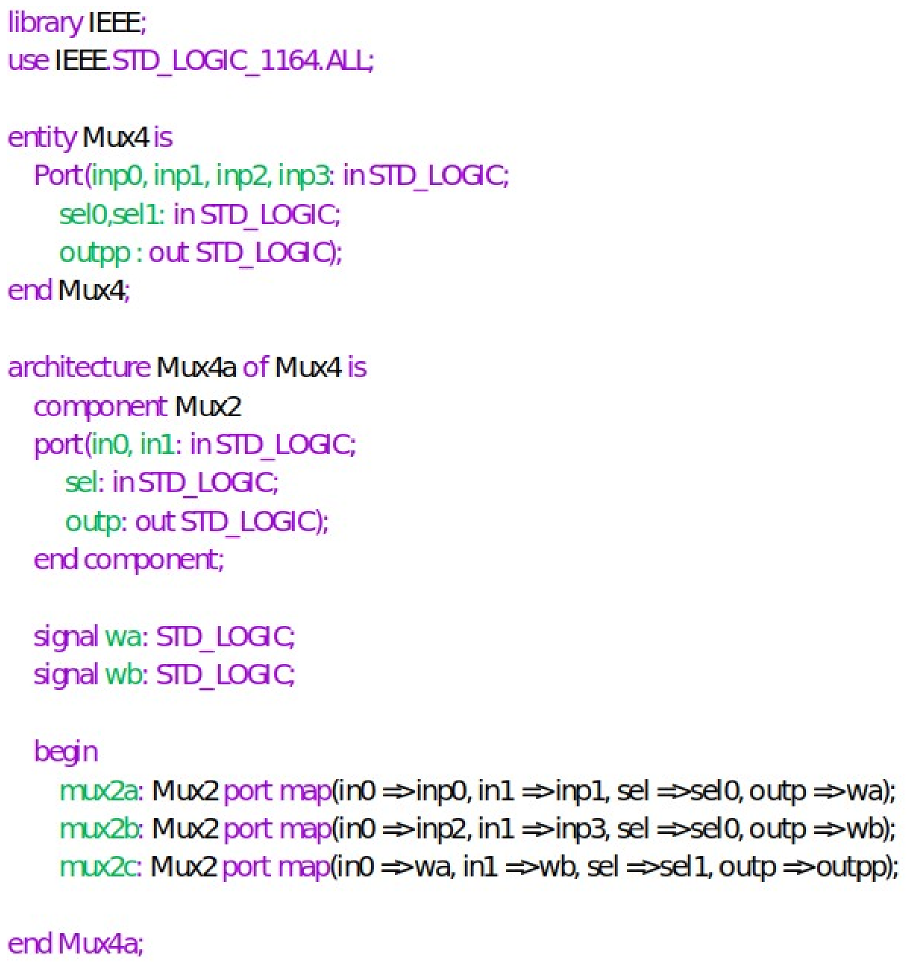


Figure 40. Block schematic for MUX4 made of 3x MUX2

As one may see, the number of layers is $\log_2 N$ => depth is O(logN), and the number of elementary circuits is N-1 => size is O(N).

```
module MUX2(input in0, input in1, input sel, output outp);
  assign outp =((sel ==0) ? in0 : in1);
endmodule

module MUX4(input inp0, input inp1, input inp2, input inp3,
          input sel0, input sel1, output outpp);
  wire mux1out;
  wire mux2out;
  MUX2 mux1(.in0(inp0), .in1(inp1), .sel(sel0), .outp(mux1out));
  MUX2 mux2(.in0(inp2), .in1(inp3), .sel(sel0), .outp(mux2out));
  MUX2 mux3(.in0(mux1out), .in1(mux2out), .sel(sel1), .outp(outpp));
endmodule

module top(input [5:0]SW, output LED);
  MUX4 mux4(.inp0(SW[0]), .inp1(SW[1]), .inp2(SW[2]), .inp3(SW[3]),
          .sel0(SW[4]), .sel1(SW[5]), .outpp(LED));
endmodule
```

Figure 41. Verilog description of MUX4 made of MUX2 circuits



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

entity Mux4 is
   Port(inp0, inp1, inp2, inp3: in STD_LOGIC;
      sel0,sel1: in STD_LOGIC;
      outpp : out STD_LOGIC);
end Mux4;

architecture Mux4a of Mux4 is
   component Mux2
   port(in0, in1: in STD_LOGIC;
       sel: in STD_LOGIC;
       outp: out STD_LOGIC);
   end component;

   signal wa: STD_LOGIC;
   signal wb: STD_LOGIC;

   begin
      mux2a: Mux2 port map(in0 =>inp0, in1 =>inp1, sel =>sel0, outp =>wa);
      mux2b: Mux2 port map(in0 =>inp2, in1 =>inp3, sel =>sel0, outp =>wb);
      mux2c: Mux2 port map(in0 =>wa, in1 =>wb, sel =>sel1, outp =>outpp);

end Mux4a;
```

Figure 42. VHDL description of MUX4 made of MUX2 circuits

2.2.3 Elementary decoder and 2-to-4 decoder

A decoder is a circuit that provides a one-hot representation of a binary number.

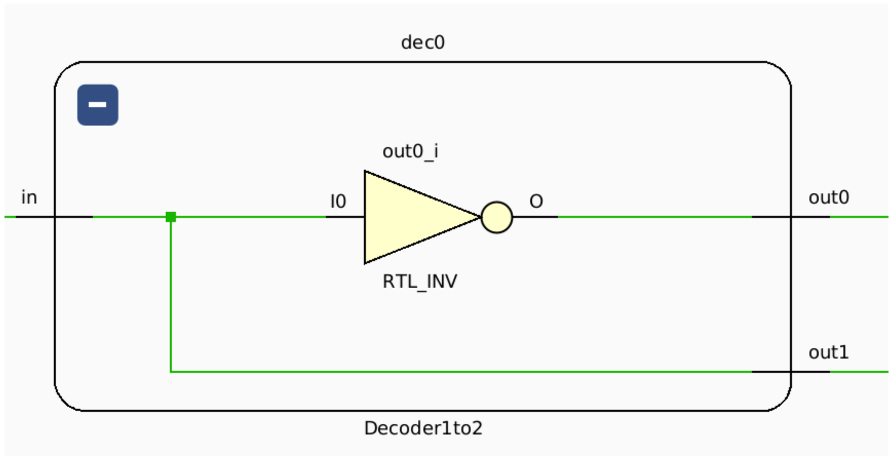


Figure 43. Schematic of an elementary decoder

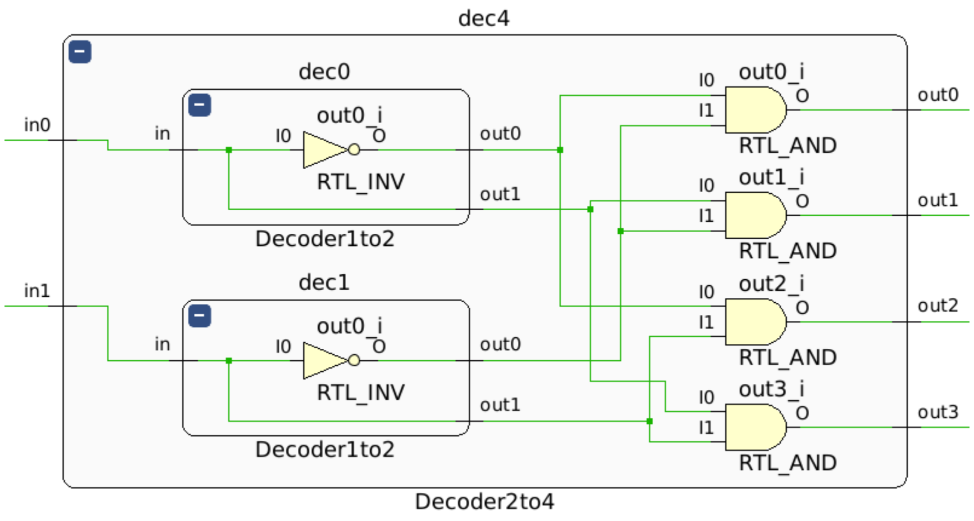


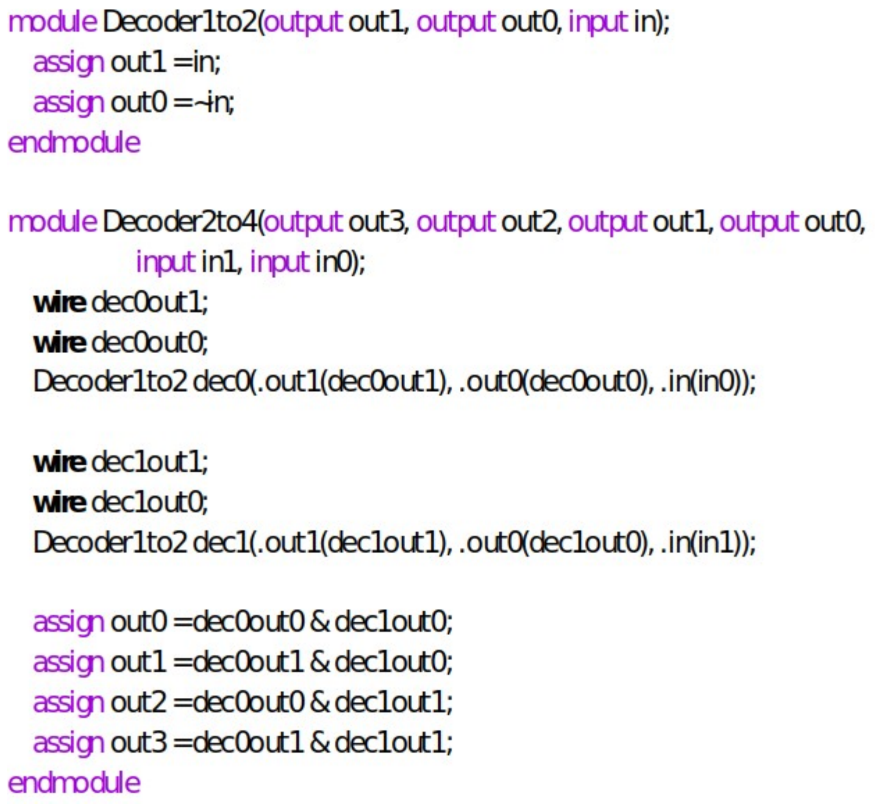
Figure 44. Schematic of 2-to-4 decoder made of 1-to-2

From this definition of 2-to-4 decoder a pattern emerges: assuming N inputs, a N-to-2**N decoder has the size S and depth D of:

S = N*sizeof(NOT) + 2**N*sizeof (N-input AND) = O(N) + 2**N * O(N) = O (N * 2**N)

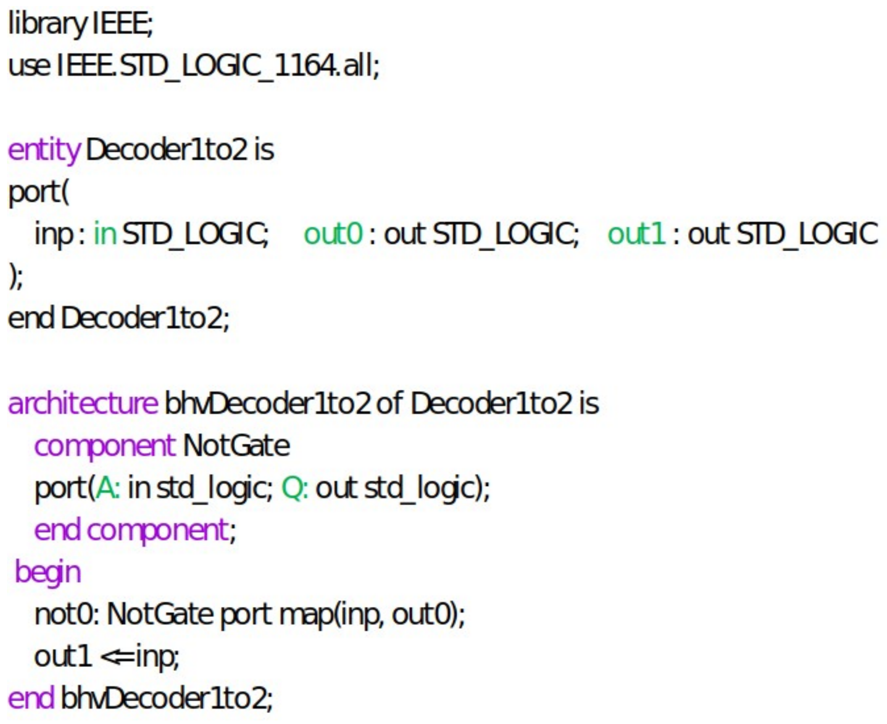
D = 1 + D (N-input AND) = O(logN)

There is another recursive definition of N-input decoder, based on two N/2 input decoders which reduce the size S to O(2**N).



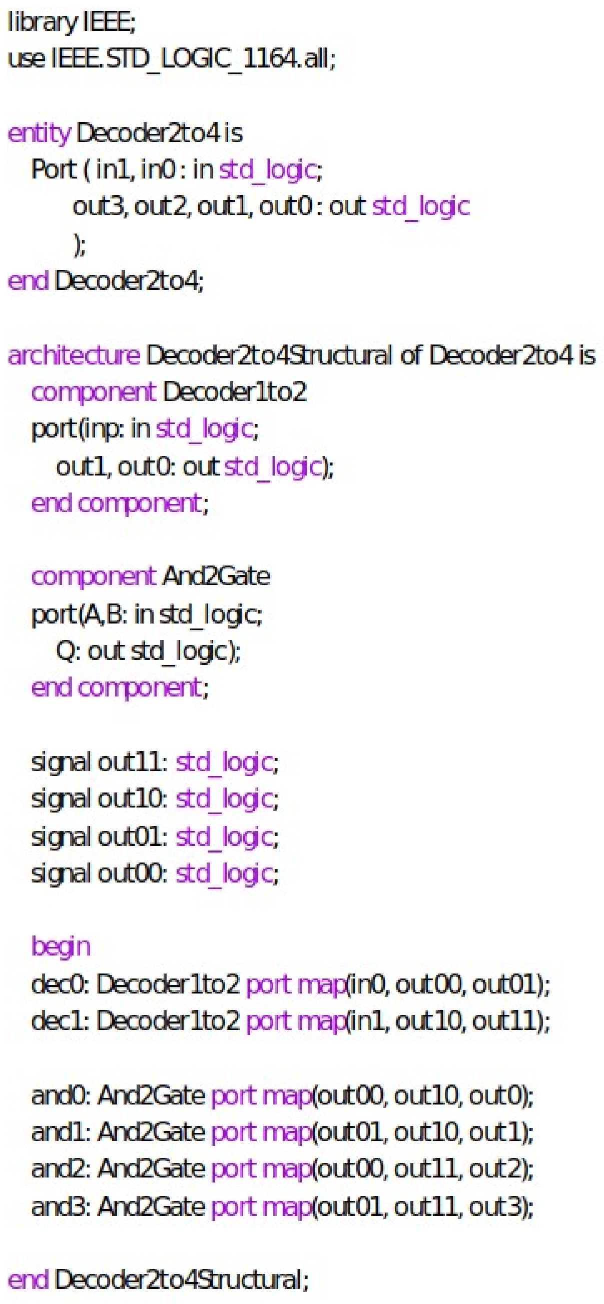
```
module Decoder1to2(output out1, output out0, input in);
   assign out1 = in;
   assign out0 = ~in;
endmodule

module Decoder2to4(output out3, output out2, output out1, output out0,
                   input in1, input in0);
   wire dec0out1;
   wire dec0out0;
   Decoder1to2 dec0(.out1(dec0out1), .out0(dec0out0), .in(in0));

   wire dec1out1;
   wire dec1out0;
   Decoder1to2 dec1(.out1(dec1out1), .out0(dec1out0), .in(in1));

   assign out0 = dec0out0 & dec1out0;
   assign out1 = dec0out1 & dec1out0;
   assign out2 = dec0out0 & dec1out1;
   assign out3 = dec0out1 & dec1out1;
endmodule
```

Figure 45. Verilog description of elementary and 2-to-4 decoder



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

entity Decoder1to2 is
port(
   inp : in STD_LOGIC;     out0 : out STD_LOGIC;    out1 : out STD_LOGIC
);
end Decoder1to2;

architecture bhvDecoder1to2 of Decoder1to2 is
   component NotGate
   port(A: in std_logic; Q: out std_logic);
   end component;
 begin
   not0: NotGate port map(inp, out0);
   out1 <=inp;
end bhvDecoder1to2;
```

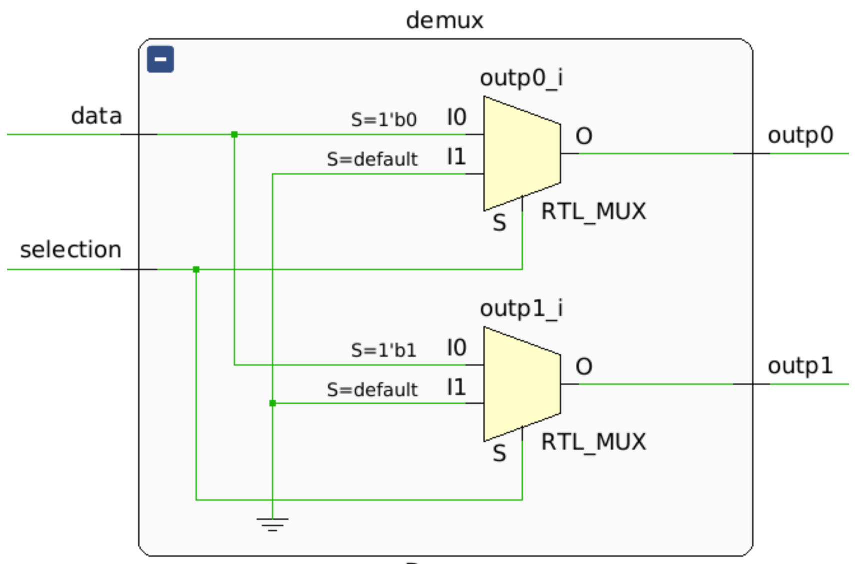
Figure 46. VHDL description of elementary decoder

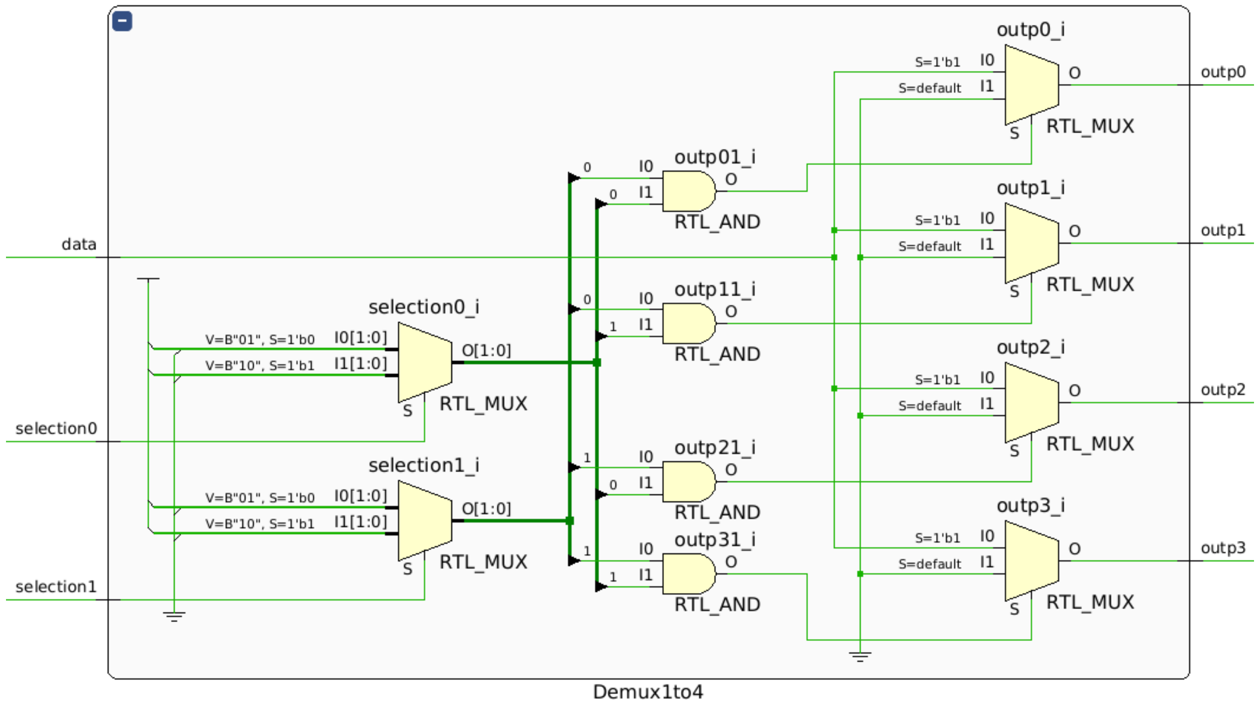


```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity Decoder2to4 is
   Port ( in1, in0 : in std_logic;
          out3, out2, out1, out0 : out std_logic
          );
end Decoder2to4;

architecture Decoder2to4Structural of Decoder2to4 is
   component Decoder1to2
   port(inp: in std_logic;
      out1, out0: out std_logic);
   end component;

   component And2Gate
   port(A,B: in std_logic;
      Q: out std_logic);
   end component;

   signal out11: std_logic;
   signal out10: std_logic;
   signal out01: std_logic;
   signal out00: std_logic;

   begin
   dec0: Decoder1to2 port map(in0, out00, out01);
   dec1: Decoder1to2 port map(in1, out10, out11);

   and0: And2Gate port map(out00, out10, out0);
   and1: And2Gate port map(out01, out10, out1);
   and2: And2Gate port map(out00, out11, out2);
   and3: And2Gate port map(out01, out11, out3);

end Decoder2to4Structural;
```

Figure 47. VHDL description of 2-to-4 decoder

2.2.4 Elementary demultiplexer and 1-to-4 demux



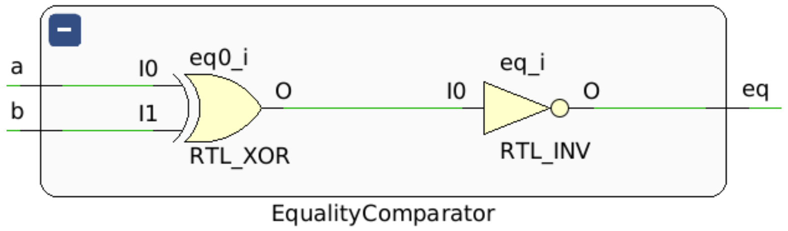


```
module Demux1to2(input data, input selection, output outp1, output outp0);
   assign outp0 =(selection ==0) ? data : 0;
   assign outp1 =(selection ==1) ? data : 0;
endmodule

module Demux1to4(input data,
             input selection1, input selection0,
             output outp3, output outp2, output outp1, output outp0);
   assign outp0 =((selection1 ==0) && (selection0 ==0)) ? data : 0;
   assign outp1 =((selection1 ==0) && (selection0 ==1)) ? data : 0;
   assign outp2 =((selection1 ==1) && (selection0 ==0)) ? data : 0;
   assign outp3 =((selection1 ==1) && (selection0 ==1)) ? data : 0;
endmodule
```

Figure 48. Verilog description of elementary demux and 1-to-4 demux

2.2.5 Elementary equality comparator and N-bit equality comparator

The equality comparator outputs 1 if both input numbers are equal, 0 otherwise.



```
module EqualityComparator(input a, input b, output eq);
   //assign eq =(a ==b);
   assign eq =~(a ^b);
endmodule
```

Figure 49. Verilog description of elementary equality comparator

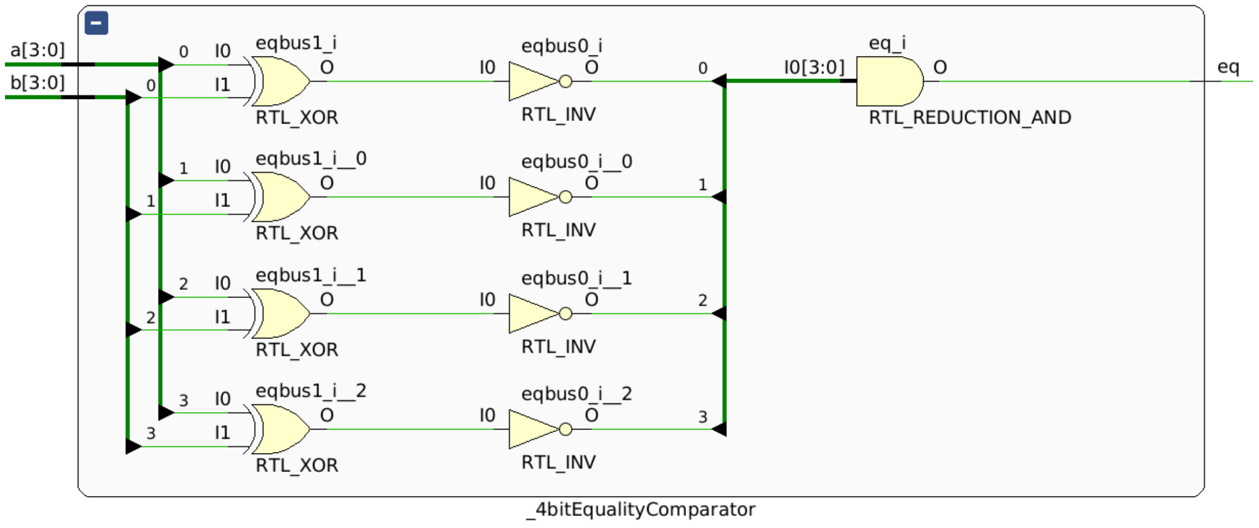
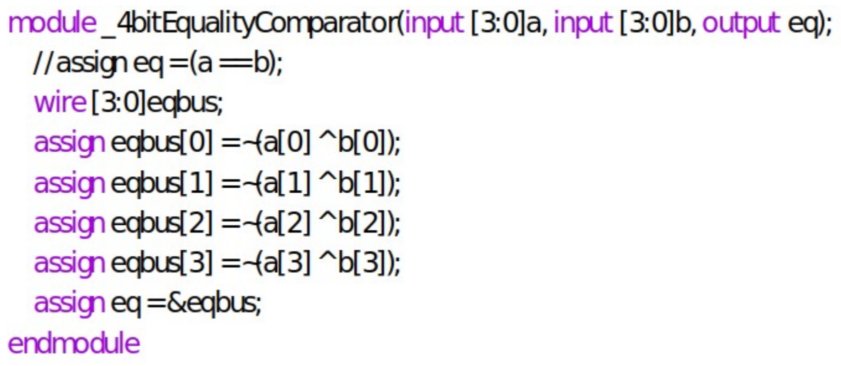


Figure 50. 4-bit equality comparator

As one may see, an N-bit equality comparator is made of N 2-input XOR gates, N inverters and one N-input AND gate. The N-input AND gate is made of $\log_2 N$ levels each having power-of-two 2-input AND gates: in total N-1 gates.
Therefore, the size S of this circuit is N + N + (N-1) = O(N)
The depth of the circuit is D = 1 + 1 + $\log_2 N$ = O (logN)



```
module _4bitEqualityComparator(input [3:0]a, input [3:0]b, output eq);
  //assign eq =(a ==b);
  wire [3:0]eqbus;
  assign eqbus[0] =~(a[0] ^b[0]);
  assign eqbus[1] =~(a[1] ^b[1]);
  assign eqbus[2] =~(a[2] ^b[2]);
  assign eqbus[3] =~(a[3] ^b[3]);
  assign eq =&eqbus;
endmodule
```

Figure 51. Verilog description of 4-bit equality comparator

2.2.6 Magnitude comparator

The magnitude comparator compares two numbers a and b, of equal size and output three signals:

- eq (equal to) which is 1 if both numbers are equal, 0 otherwise
- lt (less than) which is 1 if a < b, and 0 otherwise
- gt (greater than) which is 1 if a > b, and 0 otherwise

Below are, two implementations of 1-bit magnitude comparator:

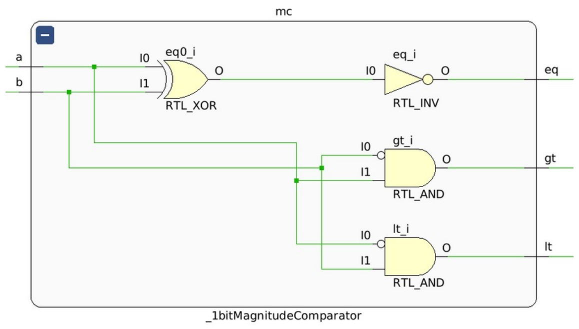


Figure 52. One-bit magnitude comparator (1)

Assuming AND, NOT and NXOR gates are used, the size of the above circuit is 5 and has depth of 2 (one NOT and one AND). Assuming AND, NOT and NOR gates are used, the size of the below circuit is also 5, and its depth is 3 (one NOT, one AND, one NOR)

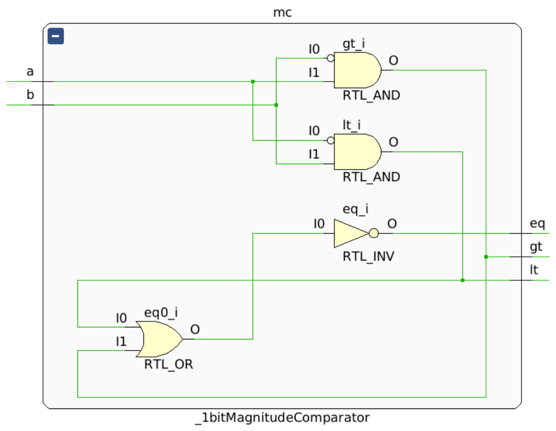
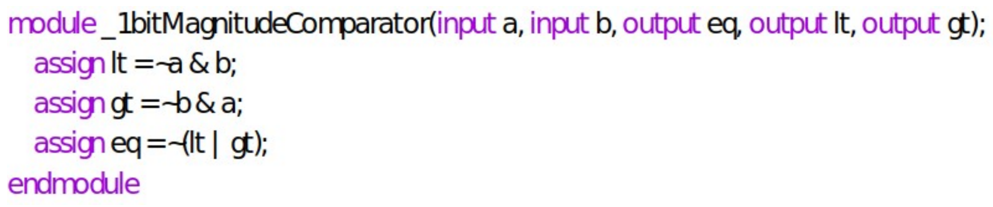
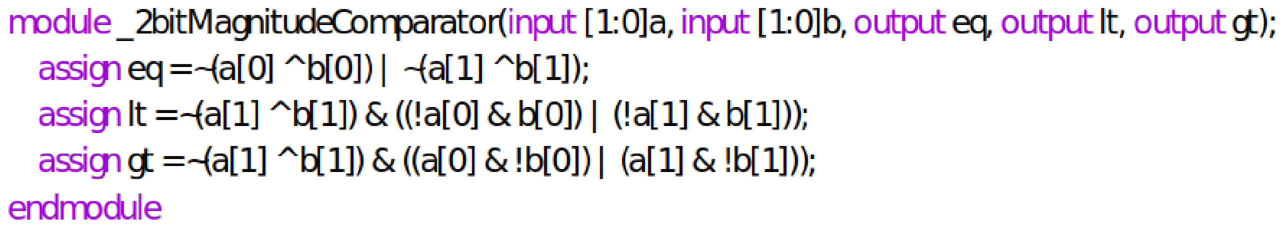


Figure 53. One-bit magnitude comparator (2)



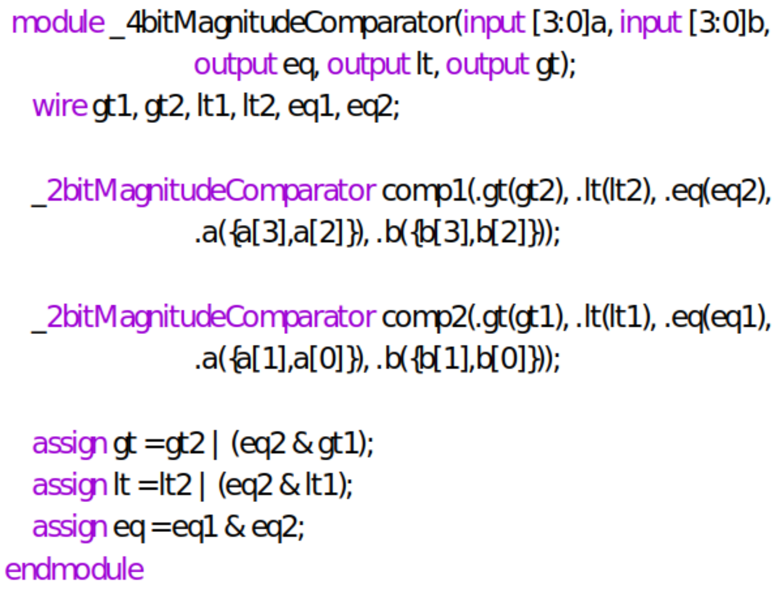
```
module _1bitMagnitudeComparator(input a, input b, output eq, output lt, output gt);
  assign lt =~a & b;
  assign gt =~b & a;
  assign eq =~(lt | gt);
endmodule
```

Figure 54. Verilog description of 1-bit magnitude comparator (with NOR gate)



```
module _2bitMagnitudeComparator(input [1:0]a, input [1:0]b, output eq, output lt, output gt);
  assign eq =~(a[0] ^b[0]) | ~(a[1] ^b[1]);
  assign lt =~(a[1] ^b[1]) & ((!a[0] & b[0]) | (!a[1] & b[1]));
  assign gt =~(a[1] ^b[1]) & ((a[0] & !b[0]) | (a[1] & !b[1]));
endmodule
```

Figure 55. Verilog description of 2-bit magnitude comparator



```
module _4bitMagnitudeComparator(input [3:0]a, input [3:0]b,
                    output eq, output lt, output gt);
  wire gt1, gt2, lt1, lt2, eq1, eq2;

  _2bitMagnitudeComparator comp1(.gt(gt2), .lt(lt2), .eq(eq2),
                    .a({a[3],a[2]}), .b({b[3],b[2]}));

  _2bitMagnitudeComparator comp2(.gt(gt1), .lt(lt1), .eq(eq1),
                    .a({a[1],a[0]}), .b({b[1],b[0]}));

  assign gt =gt2 | (eq2 & gt1);
  assign lt =lt2 | (eq2 & lt1);
  assign eq =eq1 & eq2;
endmodule
```

Figure 56. Verilog description of 4-bit magnitude comparator

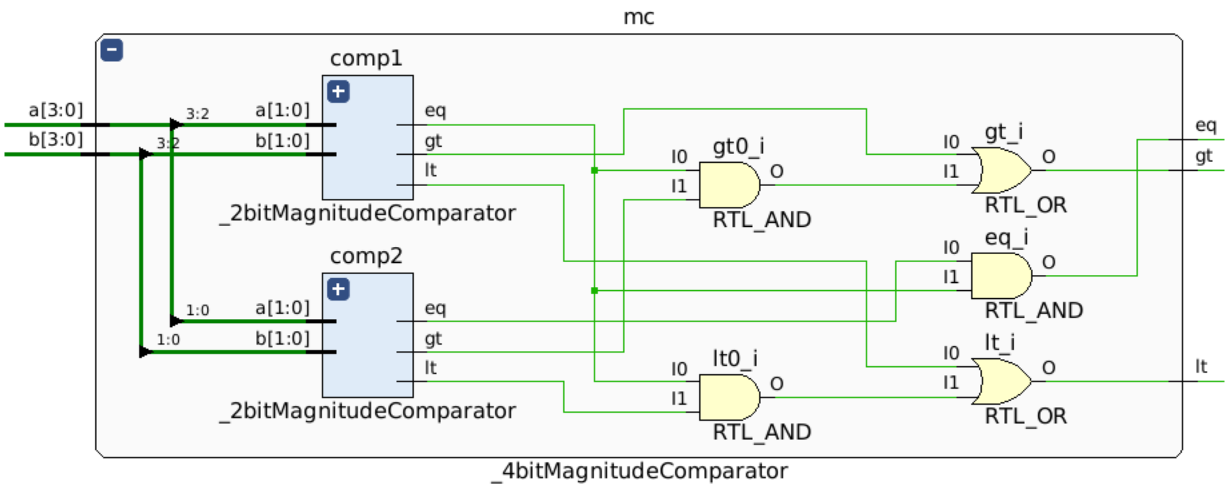
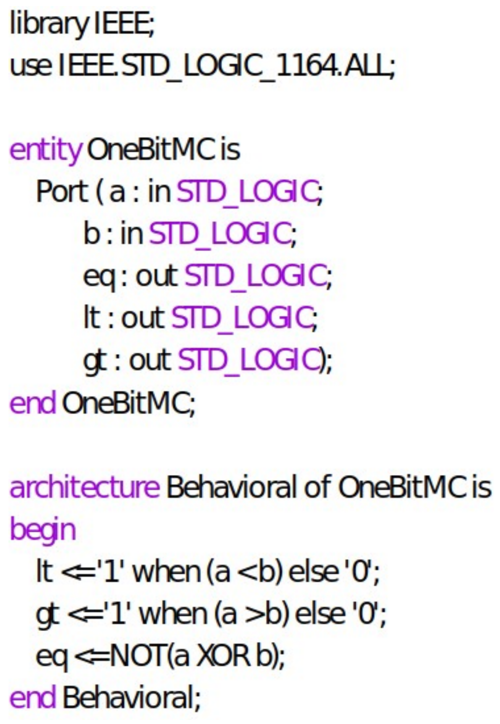
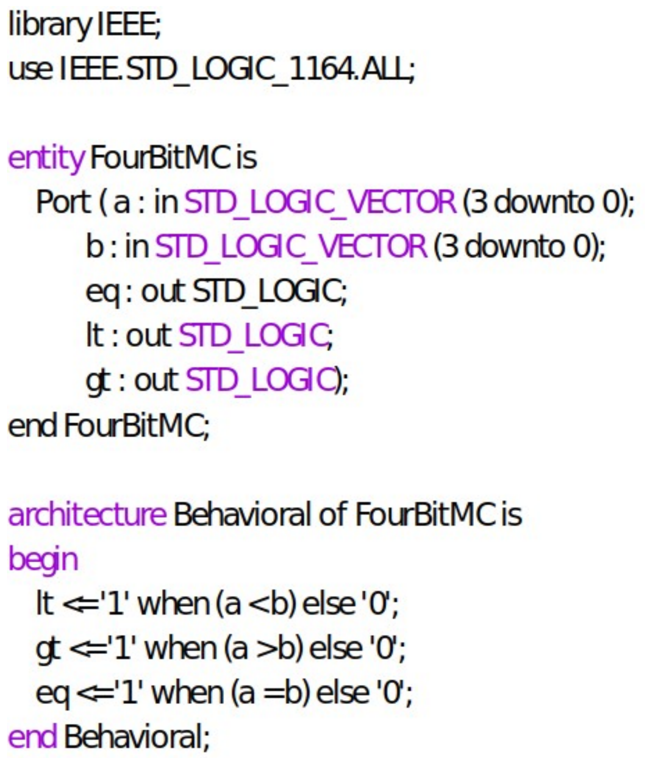


Figure 57. Schematic of 4-bit magnitude comparator made of 2-bit magnitude comparator



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

entity OneBitMC is
   Port ( a : in STD_LOGIC;
          b : in STD_LOGIC;
          eq : out STD_LOGIC;
          lt : out STD_LOGIC;
          gt : out STD_LOGIC);
end OneBitMC;

architecture Behavioral of OneBitMC is
begin
   lt <='1' when (a <b) else '0';
   gt <='1' when (a >b) else '0';
   eq <=NOT(a XOR b);
end Behavioral;
```

Figure 58. VHDL description of 1-bit magnitude comparator



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

entity FourBitMC is
  Port ( a : in STD_LOGIC_VECTOR (3 downto 0);
       b : in STD_LOGIC_VECTOR (3 downto 0);
       eq : out STD_LOGIC;
       lt : out STD_LOGIC;
       gt : out STD_LOGIC);
end FourBitMC;

architecture Behavioral of FourBitMC is
begin
  lt <='1' when (a <b) else '0';
  gt <='1' when (a >b) else '0';
  eq <='1' when (a =b) else '0';
end Behavioral;
```

Figure 59. VHDL description of 4-bit magnitude comparator

2.2.7 The 7-segment display transcoder

The 7-segment display is a type of LED-based display, requiring very few data. It can easily show all 0-9 symbols, and A-F (A b C d E F) which makes it very suitable for showing radix-16 numbers. A few letters may also be shown, but not all of them. As seen in Figure 10., the Nexys4 DDR FPGA board contains 2x 4xDigits of 7-segments each.

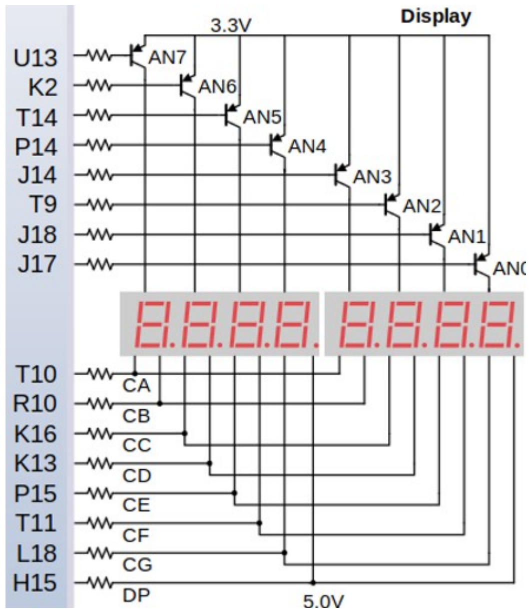


Figure 60. 7-segment displays present on Nexys4 DDR (schematic)

The 7-segment display is called this was, since it has 7 vertical or horizontal LED-based segments, composing one digit. For example, the Kingbright SA39-11CGKWA is an one-digit 7-segment display, with the following dimensions and layout:

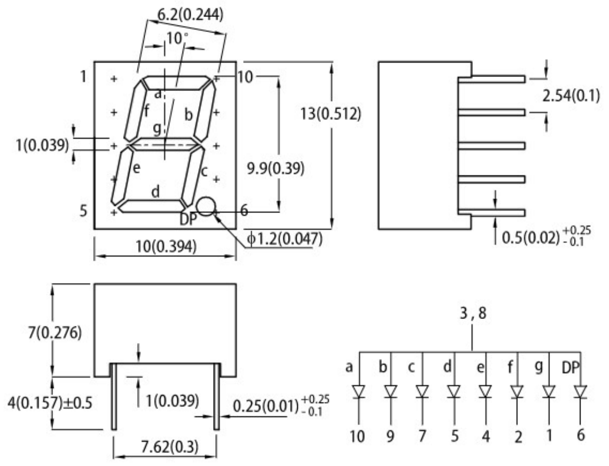


Figure 61. A Kingbright 1-digit common anode, 7-segment display

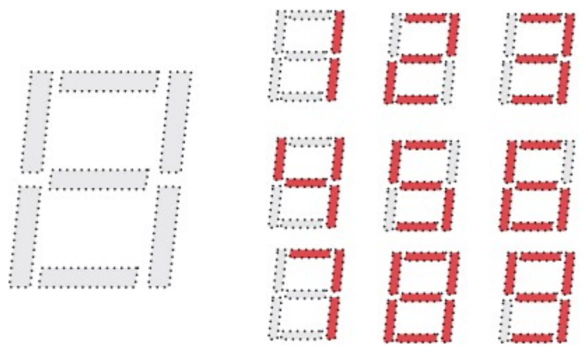


Figure 62. Font used for 0-9 characters in 7-segment displays

One may notice the 7segments, called a,b,c,d,e,f,g starting from top and going clockwise, with g being the middle horizontal segment. An 8-th segment is sometimes present, and is usually called decimal point. A 4-digit display may add additional segments to express time or to mark the passing of one second as in the figure below:

Figure 63. The TDCG1060M clock display from Vishay

Regarding the pinning, having 7 LEDs in a digit might require 2x 7 pins. However, in order to save some I/O on the driving microcontroller, one cathode and 7 anode pins (common cathode / CC) or one anode and 7 cathodes (common anode / CA).

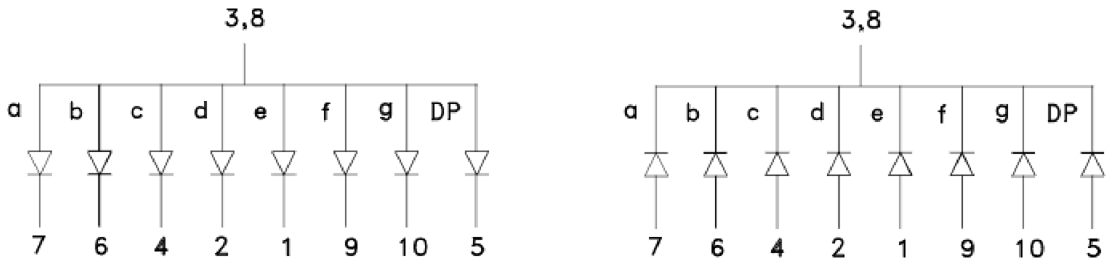


Figure 64. 7-Segment display as common anode (+) in the left side, and common cathode (-) in the right side

For 4-digit displays, even with this CC/CA saving, one should use 4x (7 + 1) = 32 pins. However, another improvement may be done, by using time-multiplexing. There is no need to show each digit in the same moment of time: digits may be powered on and off periodically, at a high enough rate to trick the human eye (>100Hz or even in the KHz range)

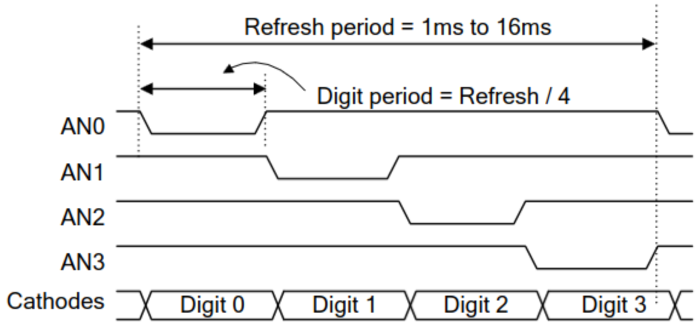


Figure 65. Time-multiplexing for the 4x 7-segment displays

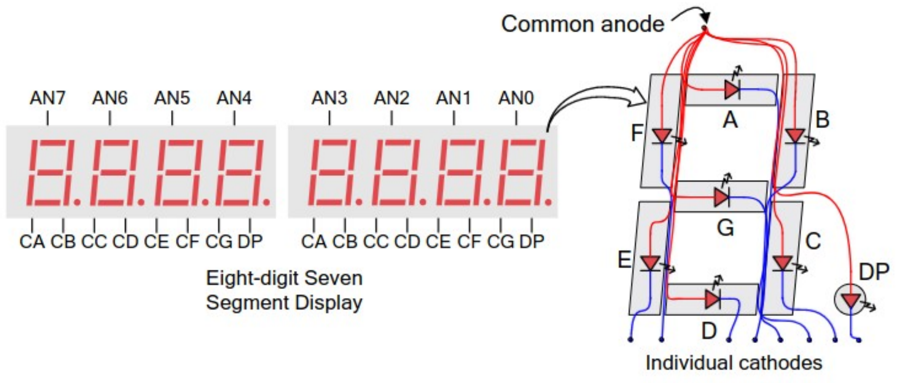


Figure 66. 2x 4 x digit comon anode circuit found in Nexys4 DDR

Another optimization may be performed to multiplex LEDs, if a low number of pins is available. This is called charlieplexing and relies on the fact that all I/Os may be powered to high (source) or low voltages (sink). In the image below, using only 5 I/Os allows to independently address 16 LEDs.

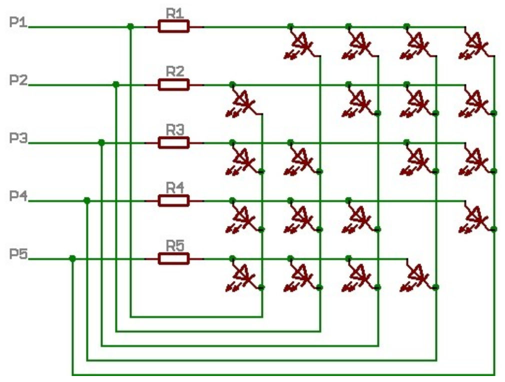


Figure 67. Charlieplexing 5 I/Os to power 16 LEDs

Exercise: Using the inputs from SW0-SW3 forming a 4-bit number, drive a 7-segment display representing that number. The schematic of the circuit is presented below:

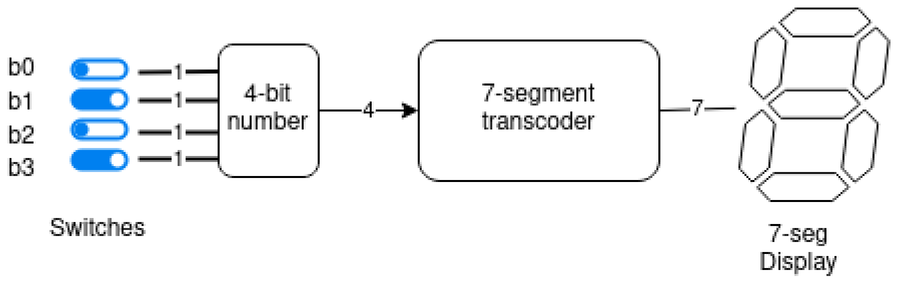
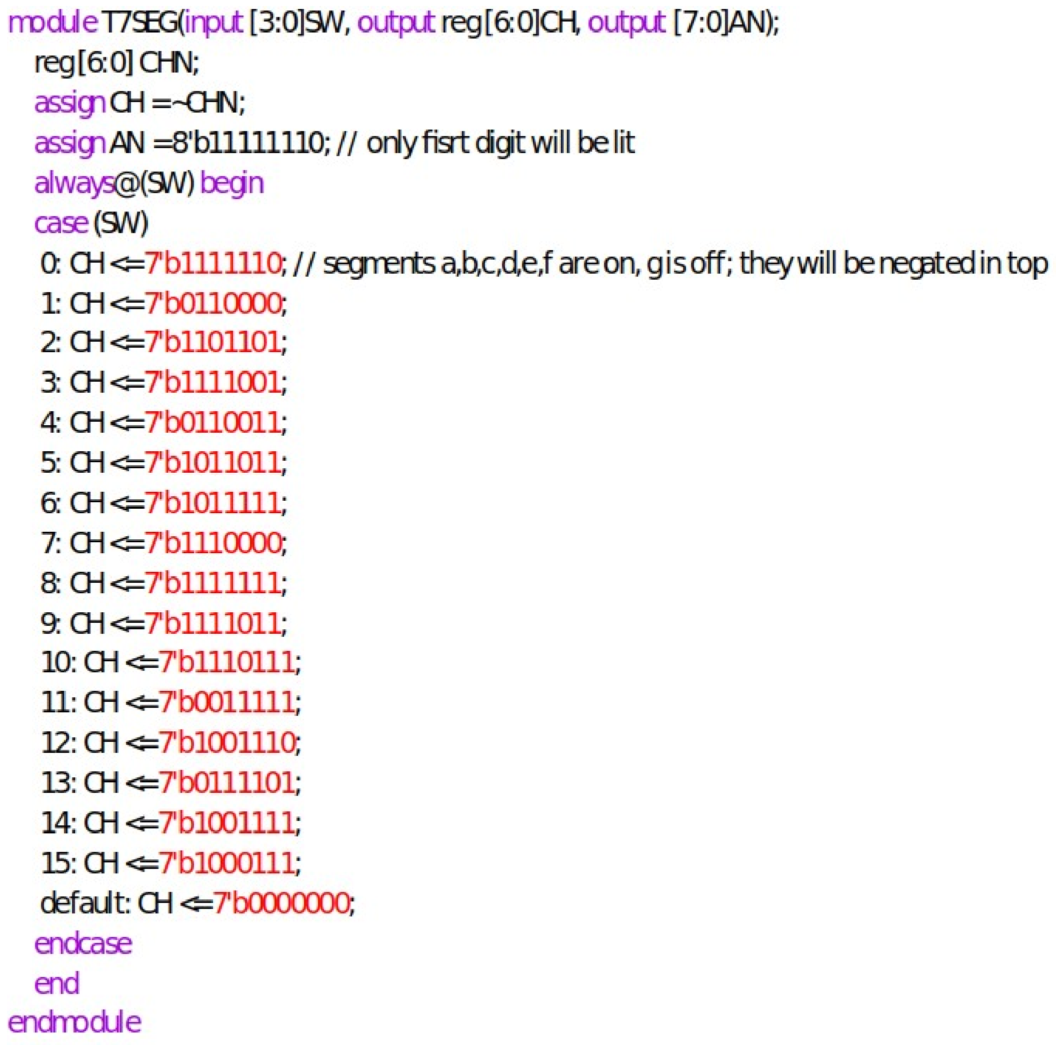
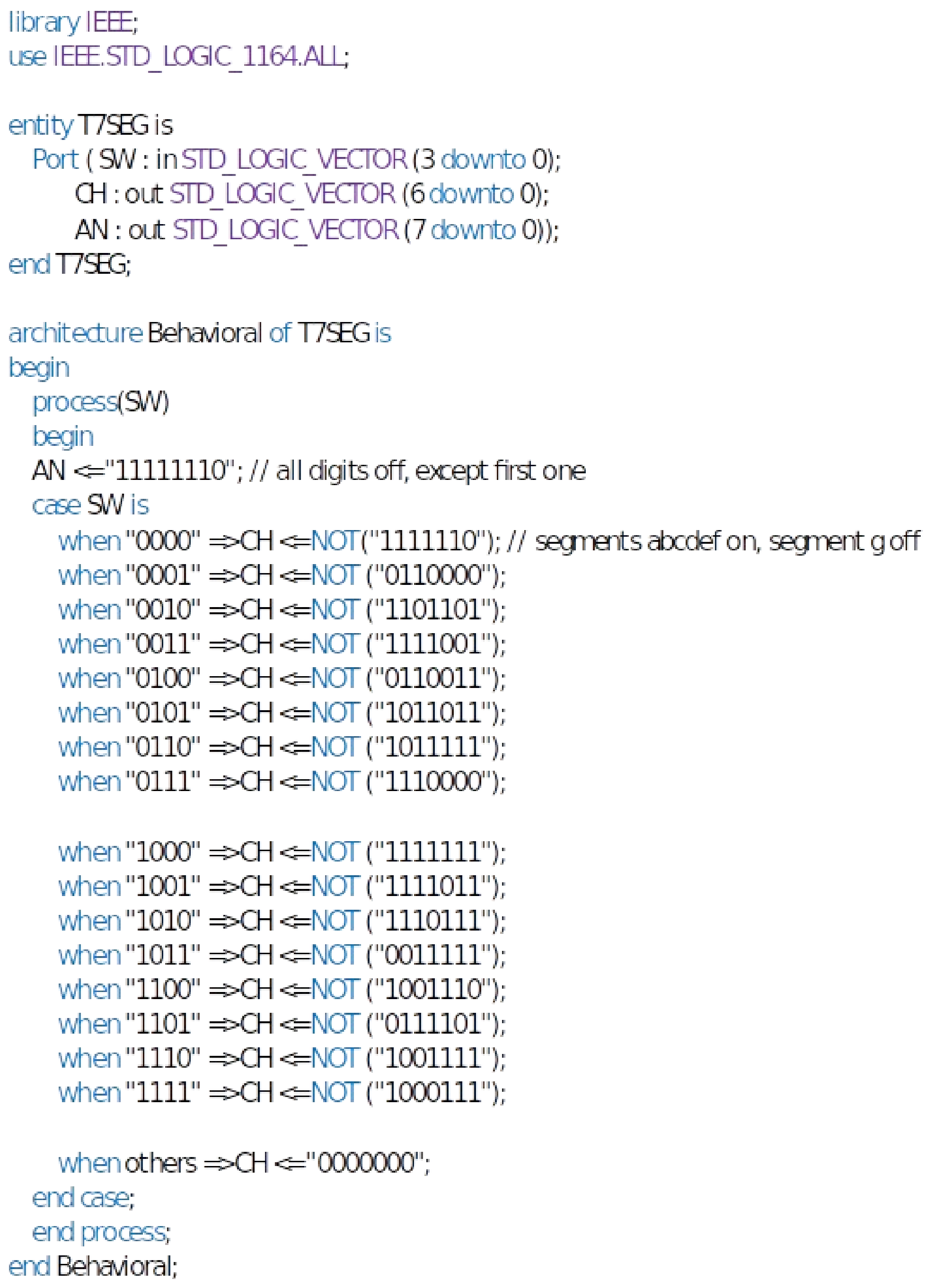


Figure 68. A 7-segment display data-driven by 4 on/off switches



```
module T7SEG(input [3:0]SW, output reg[6:0]CH, output [7:0]AN);
   reg[6:0] CHN;
   assign CH =~CHN;
   assign AN =8'b11111110; // only fisrt digit will be lit
   always@(SW) begin
   case (SW)
    0: CH <=7'b1111110; // segments a,b,c,d,e,f are on, g is off; they will be negated in top
    1: CH <=7'b0110000;
    2: CH <=7'b1101101;
    3: CH <=7'b1111001;
    4: CH <=7'b0110011;
    5: CH <=7'b1011011;
    6: CH <=7'b1011111;
    7: CH <=7'b1110000;
    8: CH <=7'b1111111;
    9: CH <=7'b1111011;
    10: CH <=7'b1110111;
    11: CH <=7'b0011111;
    12: CH <=7'b1001110;
    13: CH <=7'b0111101;
    14: CH <=7'b1001111;
    15: CH <=7'b1000111;
    default: CH <=7'b0000000;
   endcase
   end
endmodule
```

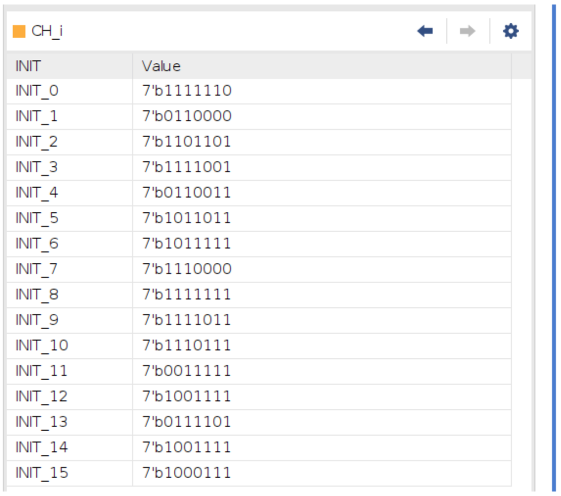
Figure 69. Verilog description for 7-segment transcoder circuit



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

entity T7SEG is
   Port ( SW : in STD_LOGIC_VECTOR (3 downto 0);
         CH : out STD_LOGIC_VECTOR (6 downto 0);
         AN : out STD_LOGIC_VECTOR (7 downto 0));
end T7SEG;

architecture Behavioral of T7SEG is
begin
   process(SW)
   begin
   AN <="11111110"; // all digits off, except first one
   case SW is
      when "0000" =>CH <=NOT("1111110"); // segments abcdef on, segment g off
      when "0001" =>CH <=NOT ("0110000");
      when "0010" =>CH <=NOT ("1101101");
      when "0011" =>CH <=NOT ("1111001");
      when "0100" =>CH <=NOT ("0110011");
      when "0101" =>CH <=NOT ("1011011");
      when "0110" =>CH <=NOT ("1011111");
      when "0111" =>CH <=NOT ("1110000");

      when "1000" =>CH <=NOT ("1111111");
      when "1001" =>CH <=NOT ("1111011");
      when "1010" =>CH <=NOT ("1110111");
      when "1011" =>CH <=NOT ("0011111");
      when "1100" =>CH <=NOT ("1001110");
      when "1101" =>CH <=NOT ("0111101");
      when "1110" =>CH <=NOT ("1001111");
      when "1111" =>CH <=NOT ("1000111");

      when others =>CH <="0000000";
   end case;
   end process;
end Behavioral;
```

Figure 70. VHDL description of 7-segment display transcoder

NB: the VHDL implementation does not have a top module (where AN/CH were inverted), therefore AN and CH should be active-low.



CH_i

INIT	Value
INIT_0	7'b1111110
INIT_1	7'b0110000
INIT_2	7'b1101101
INIT_3	7'b1111001
INIT_4	7'b0110011
INIT_5	7'b1011011
INIT_6	7'b1011111
INIT_7	7'b1110000
INIT_8	7'b1111111
INIT_9	7'b1111011
INIT_10	7'b1110111
INIT_11	7'b0011111
INIT_12	7'b1001111
INIT_13	7'b0111101
INIT_14	7'b1001111
INIT_15	7'b1000111

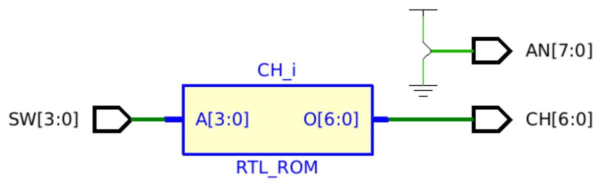


Figure 71. RTL implementation of transcoder with ROM

Lastly, the .xdc constraints file will be written as follows:

```
set_property -dict {PACKAGE_PIN J15  IOSTANDARD LVCMOS33 }[get_ports {SW[0] }];
set_property -dict {PACKAGE_PIN L16  IOSTANDARD LVCMOS33 }[get_ports {SW[1] }];
set_property -dict {PACKAGE_PIN M13  IOSTANDARD LVCMOS33 }[get_ports {SW[2] }]
set_property -dict {PACKAGE_PIN R15  IOSTANDARD LVCMOS33 }[get_ports {SW[3] }];

set_property -dict {PACKAGE_PIN T10  IOSTANDARD LVCMOS33 }[get_ports {CH[6] }];
set_property -dict {PACKAGE_PIN R10  IOSTANDARD LVCMOS33 }[get_ports {CH[5] }];
set_property -dict {PACKAGE_PIN K16  IOSTANDARD LVCMOS33 }[get_ports {CH[4] }];
set_property -dict {PACKAGE_PIN K13  IOSTANDARD LVCMOS33 }[get_ports {CH[3] }];
set_property -dict {PACKAGE_PIN P15  IOSTANDARD LVCMOS33 }[get_ports {CH[2] }];
set_property -dict {PACKAGE_PIN T11  IOSTANDARD LVCMOS33 }[get_ports {CH[1] }];
set_property -dict {PACKAGE_PIN L18  IOSTANDARD LVCMOS33 }[get_ports {CH[0] }];

set_property -dict {PACKAGE_PIN J17  IOSTANDARD LVCMOS33 }[get_ports {AN[0] }];
set_property -dict {PACKAGE_PIN J18  IOSTANDARD LVCMOS33 }[get_ports {AN[1] }];
set_property -dict {PACKAGE_PIN T9   IOSTANDARD LVCMOS33 }[get_ports {AN[2] }];
set_property -dict {PACKAGE_PIN J14  IOSTANDARD LVCMOS33 }[get_ports {AN[3] }];
set_property -dict {PACKAGE_PIN P14  IOSTANDARD LVCMOS33 }[get_ports {AN[4] }];
set_property -dict {PACKAGE_PIN T14  IOSTANDARD LVCMOS33 }[get_ports {AN[5] }];
set_property -dict {PACKAGE_PIN K2   IOSTANDARD LVCMOS33 }[get_ports {AN[6] }];
set_property -dict {PACKAGE_PIN U13  IOSTANDARD LVCMOS33 }[get_ports {AN[7] }];
```

Figure 72. Constraints file for 7-segment transcoder circuit

2.2.8 The adder

While there are multiple architectures for the adder (a circuit that computes a sum of at least two numbers), we will explore the ripple-carry adder. As an elementary unit, we will first present the **half-adder(no carry-in)** and the **full adder (with carry-in)**

2.2.8.1 The half-adder

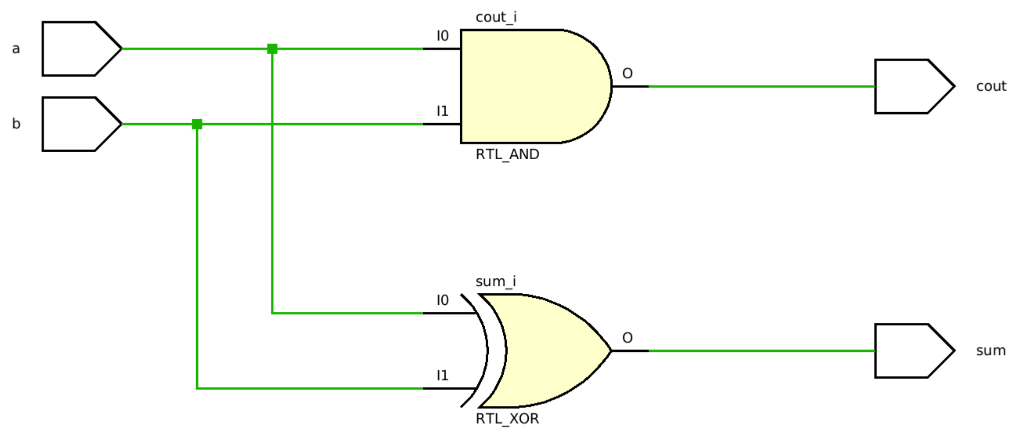


Figure 73. Schematic of half adder

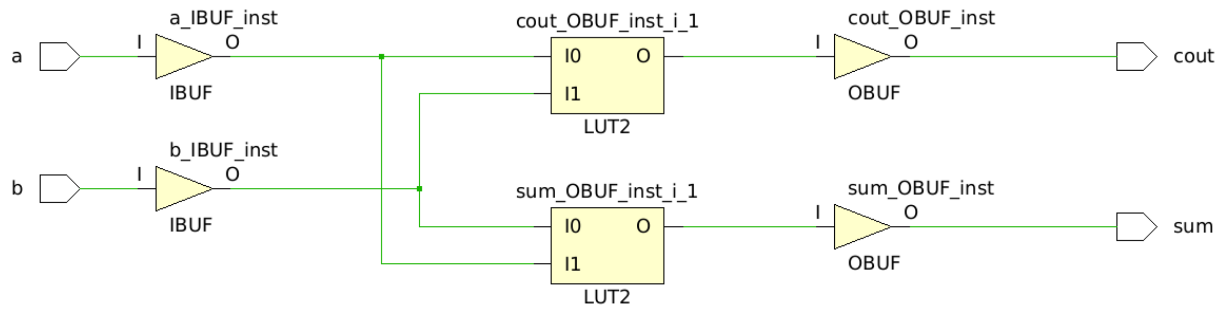
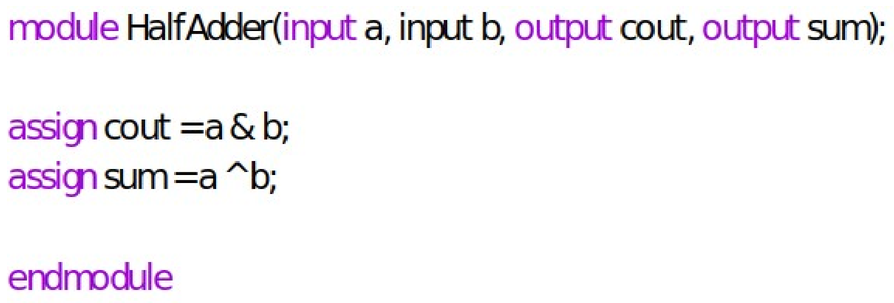


Figure 74. Synthesis result for the half-adder: LUT2 is enough

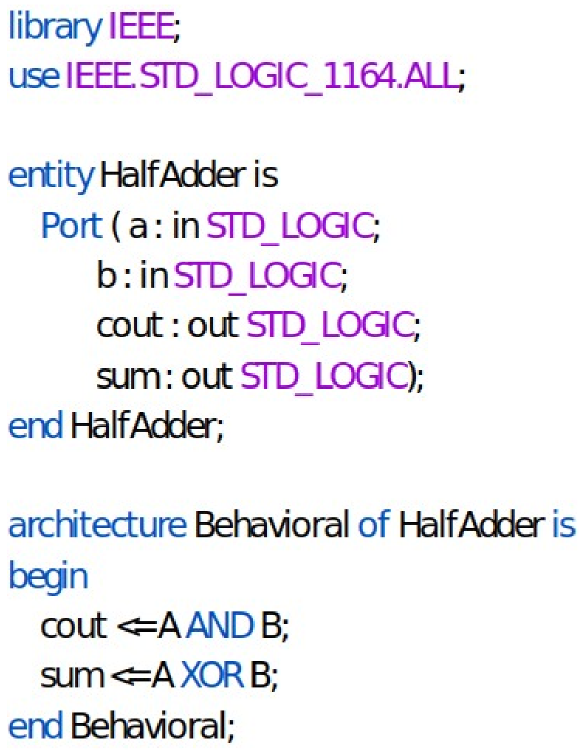
Table 6. LUT2 content

I0	**I1**	**Result**	
0	**0**	**0**	**AND with eq:** O=I0 & I1
0	**1**	**0**	
1	**0**	**0**	
1	**1**	**1**	
0	**0**	**0**	**XOR with eq:** O=I0 & !I1 + !I0 & I1
0	**1**	**1**	
1	**0**	**1**	
1	**1**	**0**	



```
module HalfAdder(input a, input b, output cout, output sum);

assign cout =a & b;
assign sum =a ^b;

endmodule
```

Figure 75. Verilog description of half-adder



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

entity HalfAdder is
   Port ( a : in STD_LOGIC;
          b : in STD_LOGIC;
          cout : out STD_LOGIC;
          sum : out STD_LOGIC);
end HalfAdder;

architecture Behavioral of HalfAdder is
begin
   cout <=A AND B;
   sum <=A XOR B;
end Behavioral;
```

Figure 76. VHDL description of half-adder

The size of a half-adder is, S = 2

The depth of a half-adder is, D = 1

2.2.8.2 The full-adder

The full-adder is a circuit that adds three bits (a, b, carry-in) and output two bits (sum and carry-out). This is possible without overflow since 1+1+1 = 3, just enough to be expressed on 2 bits.

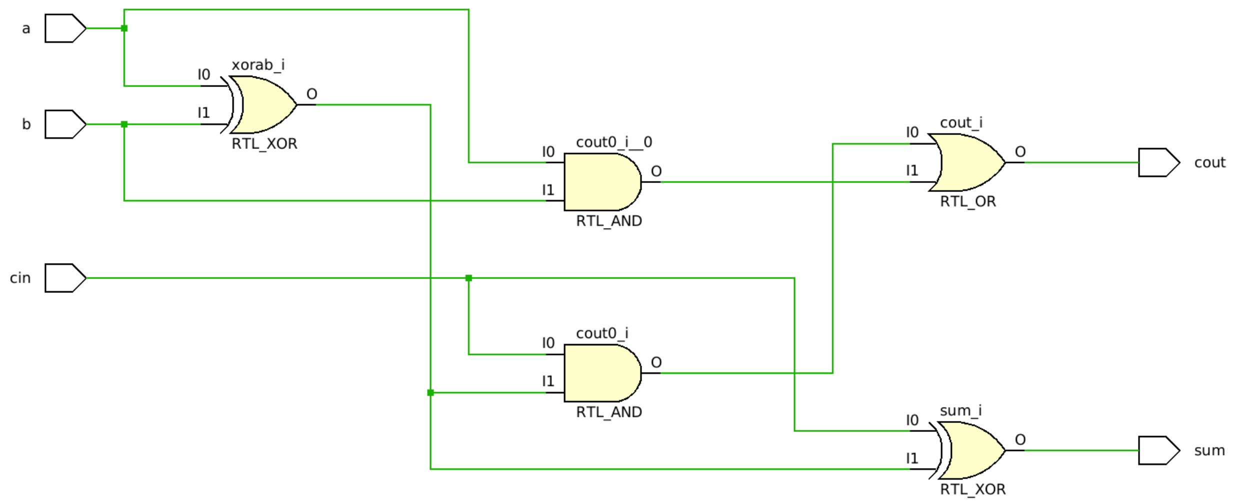


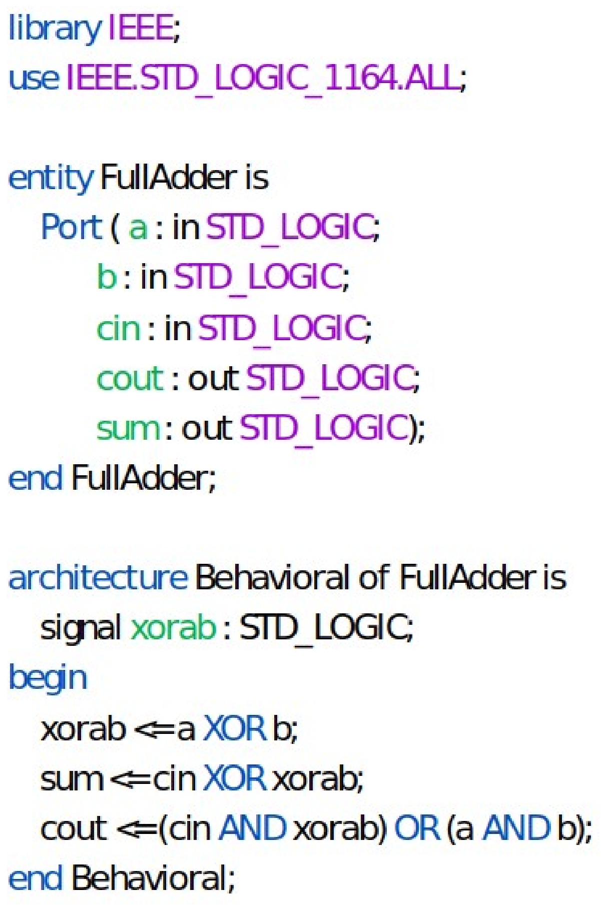
Figure 77. Full adder schematic

```
module FullAdder(input a, input b, input cin,
            output cout, output sum);
  //assign {cout, sum}={a +b +cin};
  wire xorab;
  assign xorab =a ^b;
  assign sum=cin ^xorab;
  assign cout =(cin & xorab) | (a & b);
endmodule
```

Figure 78. Verilog description of a full adder

```
set_property -dict {PACKAGE_PIN J15  IOSTANDARD LVCMOS33 }[get_ports {a }];
set_property -dict {PACKAGE_PIN L16  IOSTANDARD LVCMOS33 }[get_ports {b }];
set_property -dict {PACKAGE_PIN M13  IOSTANDARD LVCMOS33 }[get_ports {cin}];
set_property -dict {PACKAGE_PIN H17  IOSTANDARD LVCMOS33 }[get_ports {sum }];
set_property -dict {PACKAGE_PIN K15  IOSTANDARD LVCMOS33 }[get_ports {cout }];
```

Figure 79. XDC file (with pin constraints) for full adder



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

entity FullAdder is
   Port ( a : in STD_LOGIC;
          b : in STD_LOGIC;
          cin : in STD_LOGIC;
          cout : out STD_LOGIC;
          sum : out STD_LOGIC);
end FullAdder;

architecture Behavioral of FullAdder is
   signal xorab : STD_LOGIC;
begin
   xorab <= a XOR b;
   sum <= cin XOR xorab;
   cout <= (cin AND xorab) OR (a AND b);
end Behavioral;
```

Figure 80. VHDL description of a full adder

The size S of a full-adder is 5. The depth D of a full-adder is 3.

Exercise. Given the 1-bit full adder above, create an adder for two 4-bit numbers.

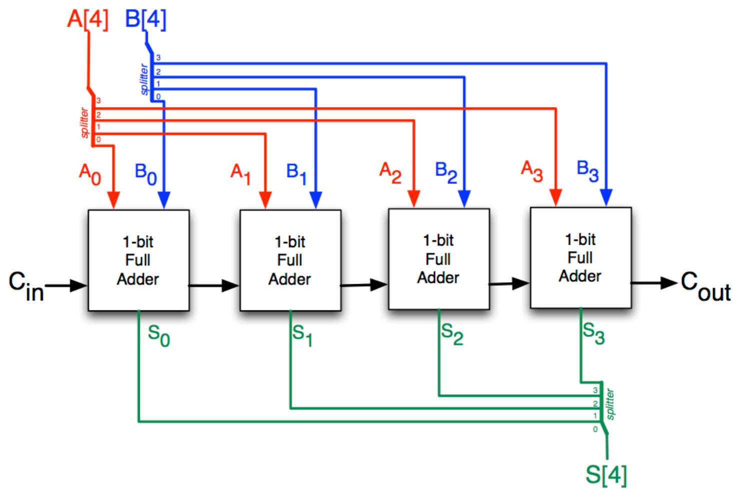


Figure 81. 4-bit ripple carry adder

```
module FourBitRippleCarryAdder(input [3:0]a, input [3:0]b,
                        output cout, output [3:0]sum);
  wire [2:0]carryio;
  FullAdder fa0(a[0], b[0], 0, carryio[0], sum[0]);
  FullAdder fa1(a[1], b[1], carryio[0], carryio[1], sum[1]);
  FullAdder fa2(a[2], b[2], carryio[1], carryio[2], sum[2]);
  FullAdder fa3(a[3], b[3], carryio[2], cout, sum[3]);
endmodule

module FullAdder(input a, input b, input cin,
            output cout, output sum);
  //assign {cout, sum}={a +b +cin};
  wire xorab;
  assign xorab =a ^b;
  assign sum =cin ^xorab;
  assign cout =(cin & xorab) | (a & b);
endmodule
```

Figure 82. Verilog description of 4-bit ripple-carry adder

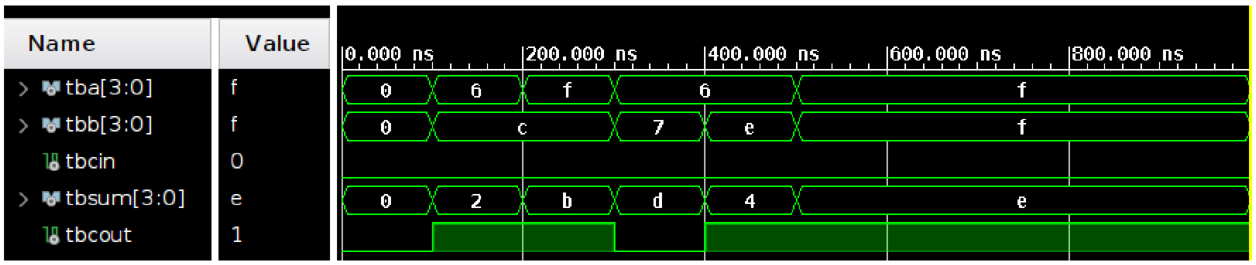
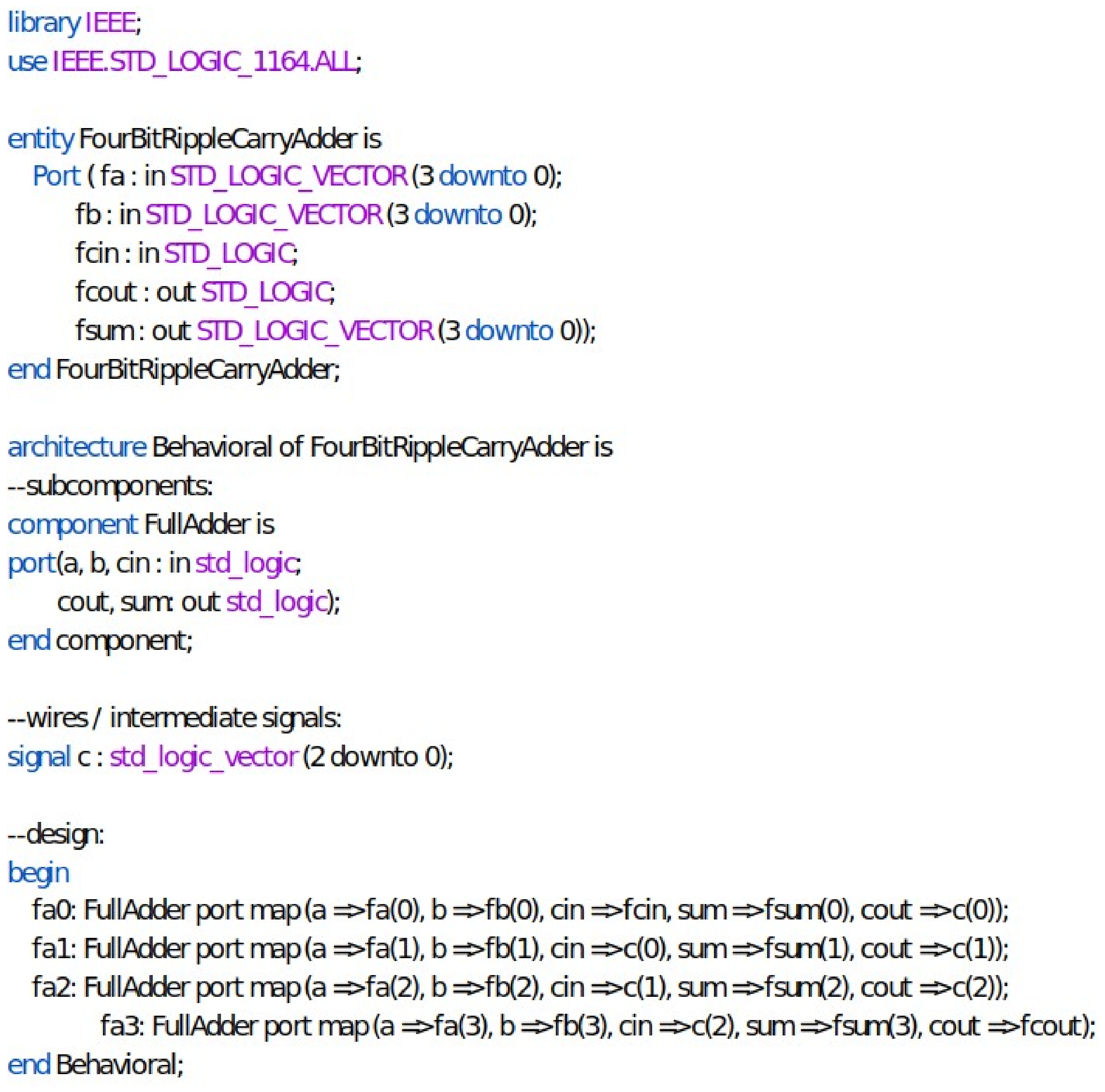
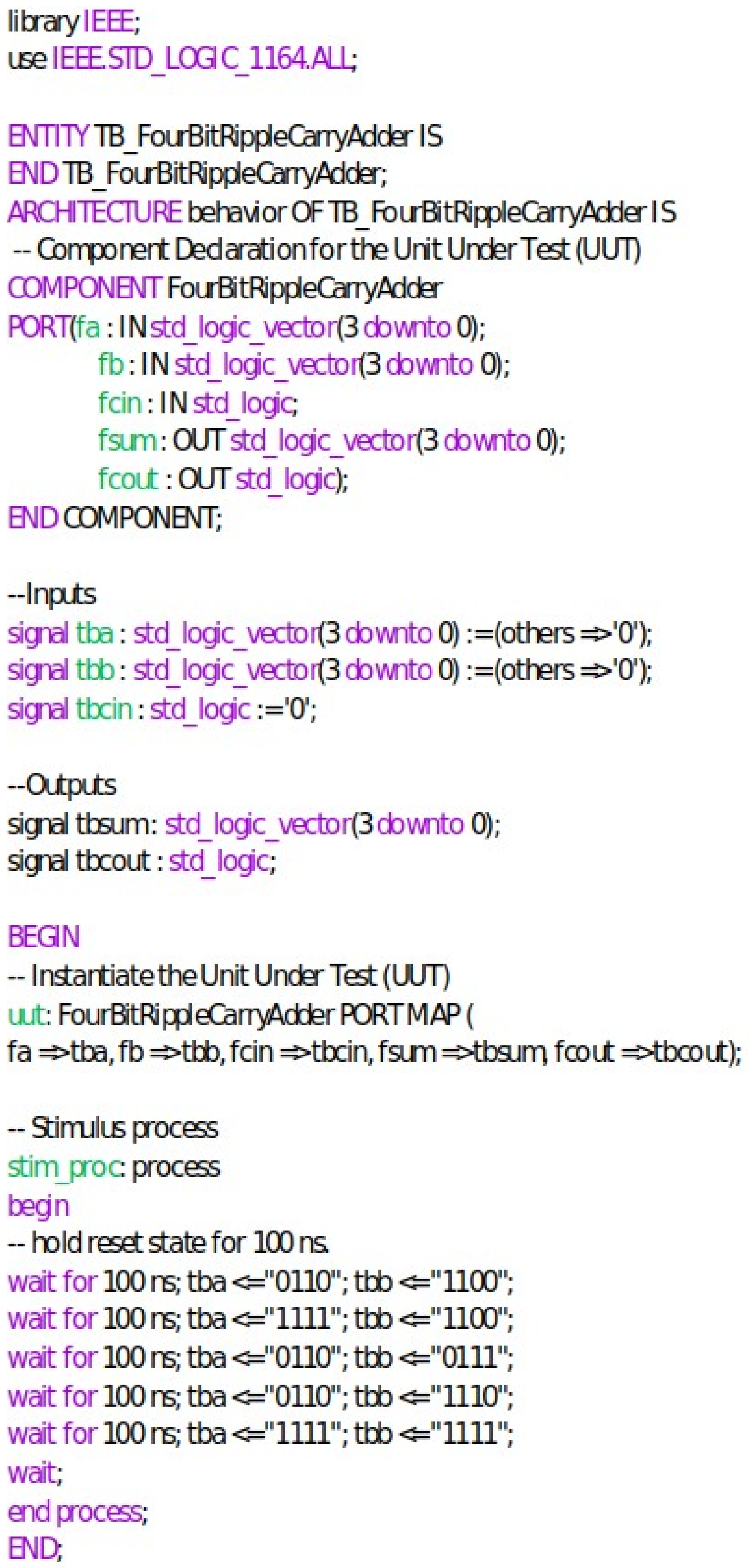


Figure 83. Simulation results for 4-bit full adder



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

entity FourBitRippleCarryAdder is
   Port ( fa : in STD_LOGIC_VECTOR (3 downto 0);
          fb : in STD_LOGIC_VECTOR (3 downto 0);
          fcin : in STD_LOGIC;
          fcout : out STD_LOGIC;
          fsum : out STD_LOGIC_VECTOR (3 downto 0));
end FourBitRippleCarryAdder;

architecture Behavioral of FourBitRippleCarryAdder is
--subcomponents:
component FullAdder is
port(a, b, cin : in std_logic;
       cout, sum: out std_logic);
end component;

--wires / intermediate signals:
signal c : std_logic_vector (2 downto 0);

--design:
begin
   fa0: FullAdder port map (a => fa(0), b => fb(0), cin => fcin, sum => fsum(0), cout => c(0));
   fa1: FullAdder port map (a => fa(1), b => fb(1), cin => c(0), sum => fsum(1), cout => c(1));
   fa2: FullAdder port map (a => fa(2), b => fb(2), cin => c(1), sum => fsum(2), cout => c(2));
            fa3: FullAdder port map (a => fa(3), b => fb(3), cin => c(2), sum => fsum(3), cout => fcout);
end Behavioral;
```

Figure 84. VHDL description of 4-bit adder



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

ENTITY TB_FourBitRippleCarryAdder IS
END TB_FourBitRippleCarryAdder;
ARCHITECTURE behavior OF TB_FourBitRippleCarryAdder IS
 -- Component Declaration for the Unit Under Test (UUT)
COMPONENT FourBitRippleCarryAdder
PORT(fa : IN std_logic_vector(3 downto 0);
        fb : IN std_logic_vector(3 downto 0);
        fcin : IN std_logic;
        fsum : OUT std_logic_vector(3 downto 0);
        fcout : OUT std_logic);
END COMPONENT;

--Inputs
signal tba : std_logic_vector(3 downto 0) :=(others =>'0');
signal tbb : std_logic_vector(3 downto 0) :=(others =>'0');
signal tbcin : std_logic :='0';

--Outputs
signal tbsum: std_logic_vector(3 downto 0);
signal tbcout : std_logic;

BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: FourBitRippleCarryAdder PORT MAP (
fa =>tba, fb =>tbb, fcin =>tbcin, fsum =>tbsum, fcout =>tbcout);

-- Stimulus process
stim_proc: process
begin
-- hold reset state for 100 ns.
wait for 100 ns; tba <="0110"; tbb <="1100";
wait for 100 ns; tba <="1111"; tbb <="1100";
wait for 100 ns; tba <="0110"; tbb <="0111";
wait for 100 ns; tba <="0110"; tbb <="1110";
wait for 100 ns; tba <="1111"; tbb <="1111";
wait;
end process;
END;
```

Figure 85. VHDL description of simulation testbench for 4-bit adder

Exercise: Given the above T7SEG module, create a design of an ALU (calculator) having the operations: addition, subtraction, multiplication and division, modulo, AND, OR, XOR between two 2-bit numbers. The numbers are intentionally kept on 2-bits to have the largest result (of multiplication) in 4-bit range.

Solution: A simple schematic would be:

2x 2-bit Numbers -> ALU with 4-bit result -> T7SEG to display values 0 to 15.

3. One-loop systems

Memory unit of 1 bit...

4. Two-loop systems (automata)

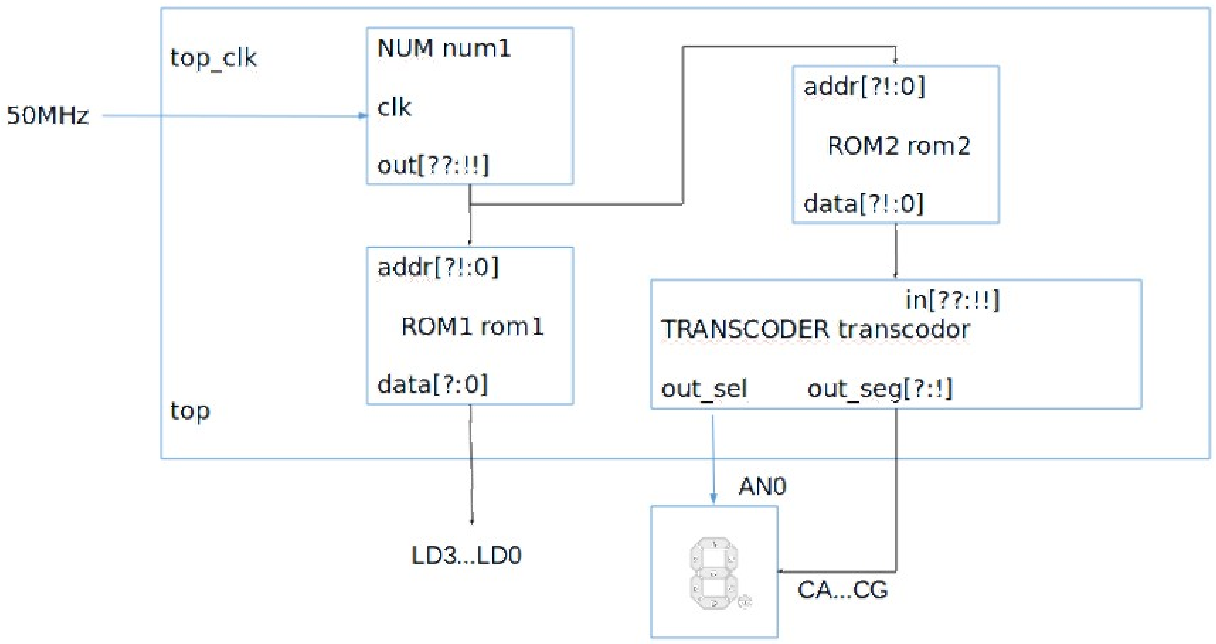
4.1 Prerequisites

4.2 Theory

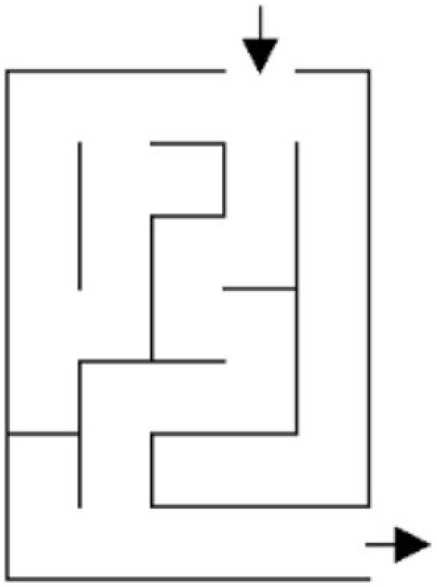
4.3 TODO: other exercises

Exercises.

Famous Chip & Dale squirrels want to go to Beautiful Almond Trees. For this, Chip hired you to design and implement an electronic map, for Dale, with the following block schematic:



The map is as follows:



To walk through the maze, the squirrel should go: 3 steps forward, 1 step to the right, 1 step to the left...

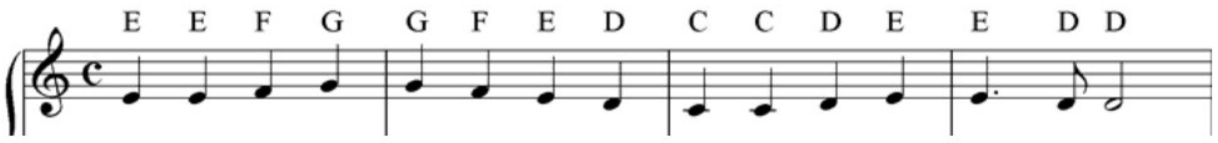
ROM1 memory contains data regarding the direction where the squirrel should go:
1 = Forward, 2 = Backwards, 4 = Left, 8 = Right

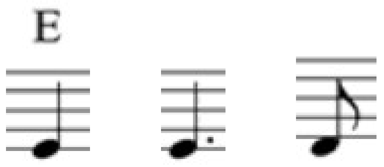
ROM2 memory contains data regarding how many steps the squirrel go in that direction

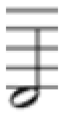
Iesirea out a numaratorului, isi schimba valoarea la fiecare FIX 1 secunda (veverita isi poate schimba directia doar 1 data pe secunda)

Your task is to write Verilog code for the electronic map.
Punctaj (din 50 de puncte): 6 + 7 + 7 + 4 p, 12p pentru top, 10p pentru "design" si 4p coding style.
Numaratorul de timp NUM (6p):
(1p) numaratorul are dimensiunea minima necesara a registrului de stare interna
(1p) numaratorul are dimensiunea minima necesara a registrului de iesire
(1p) registrul de stare interna se incrementeaza in ritmul corespunzator
(1p) registrul de iesire se incrementeaza in ritmul corespunzator (1 secunda)
(1p) conditia de numarare este corecta
(1p) modulul verilog are denumirea ceruta (NUM)
Memoria de tip ROM1 (7p):
(1p) are numarul minim de adrese / locatii, numele cerut al intrarii
(1p) are numarul minim de biti de iesire, numele cerut al iesirii
(1p) conditia de citire e corecta
(Xp) memoria are continutul corect in proportie de X*25%, (X e maxim 4)
Memoria de tip ROM2 (7p)
(1p) are numarul minim de adrese / locatii, numele cerut al intrarii
(1p) are numarul minim de biti de iesire, numele cerut al iesirii
(1p) conditia de citire e corecta
(Xp) memoria are continutul corect in proportie de X*25%, (X e maxim 4)
Transocodorul de tip TRANSCODER (4p)
(1p) in are dimensiunea, tipul corect, numele cerut
(1p) out_sel are dimensiunea si tipul corect, numele cerut
(2p) out_seg are dimensiunea si tipul corect, numele cerut
top (12p):
(1p) memoria ROM1 este instantiata corect (tip, nume semnale)
(1p) memoria ROM2 este instantiata corect (tip, nume semnale)
(1p) numaratorul este instantiat corect (tip, nume semnale)
(1p) transcodorul este instantiat corect (tip, nume semnale)
(1p) legaturile NUM-ROM1 sunt corecte
(1p) legaturile NUM-ROM2 sunt corecte

(1p) legaturile ROM2-TRANSCODER sunt corecte
(1p) top_clk se leaga corect in modulul de top
(1p) ROM1 se leaga corect in exterior
(1p) TRANSCODER se leaga corect in exterior (top: out_sel)
(2p) TRANSCODER se leaga corect in exterior (top: out_seg)
design (10p):
(3p) design-ul este complet (ca numar / tip de componente) si nu are erori de sintaxa / sinteza /
implementare
(3p) design-ul functioneaza pe FPGA asa cum s-a cerut (intre stari trec fix. X milisecunde si starile
se succed in ordinea ceruta)
(1p) corespondenta semnalului de ceas <> pin e corecta
(1p) corespondenta biti de iesire <> pini e corecta (selectie digit)
(2p) corespondenta biti de iesire <> pini e corecta (segmente)
coding_style(4p): codul este usor de citit (indentat si spatiat similar cu exercitiul din laborator5)

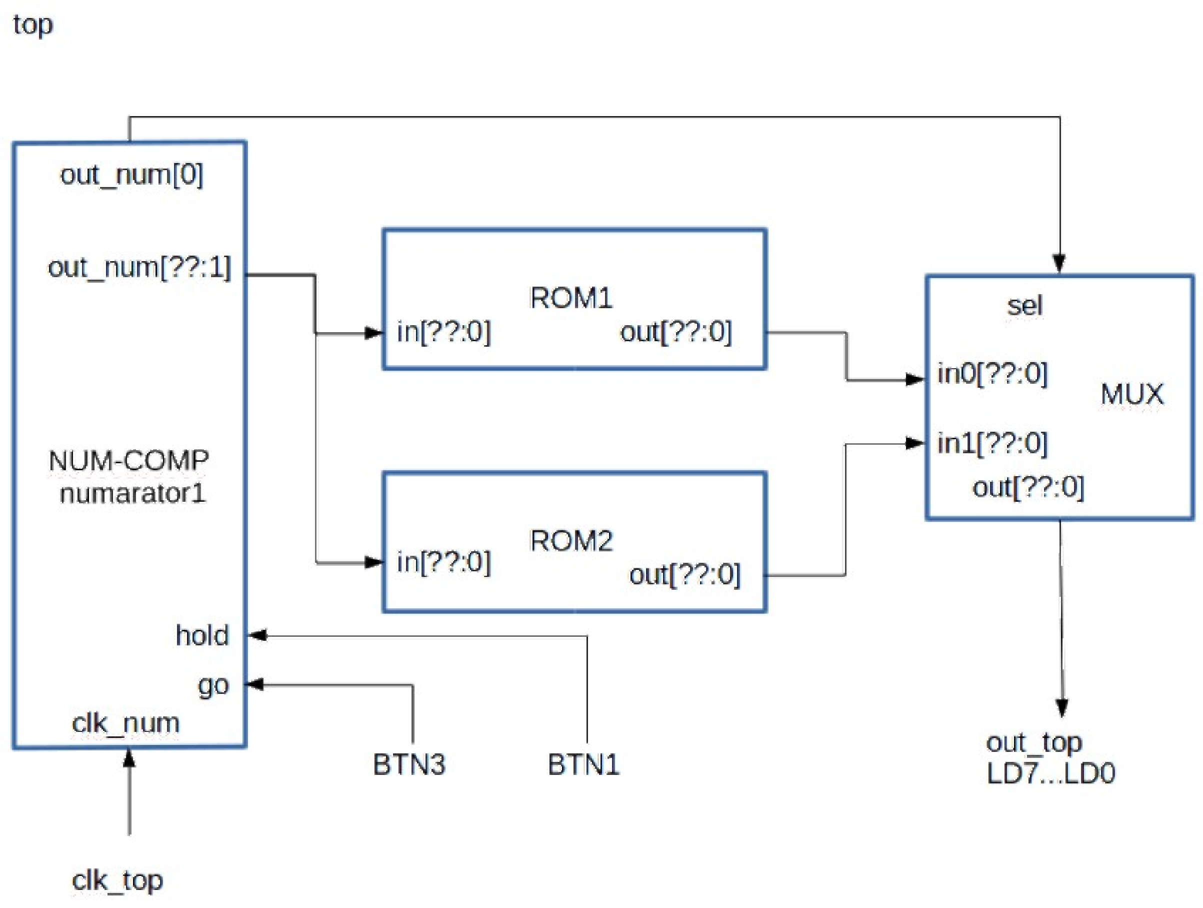






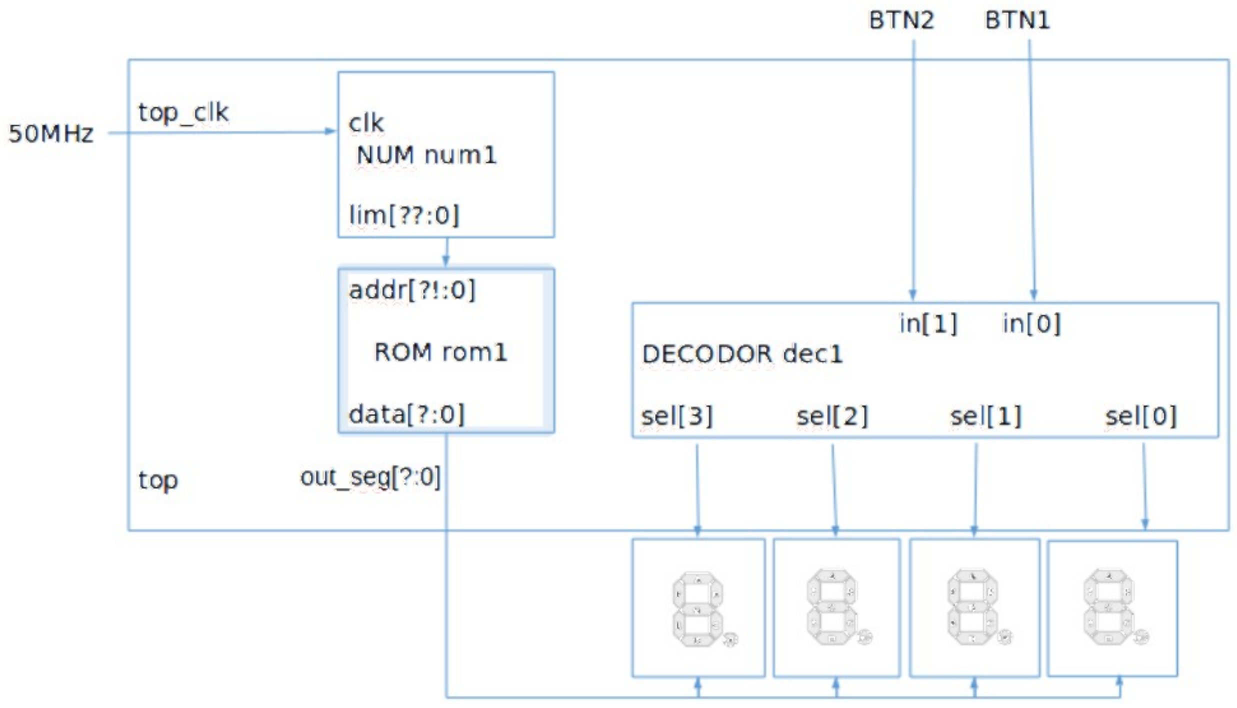
Timp de efectiv de lucru: 50 de minute. SUBIECT_5_FARA_RAM
Extraterestrii din sistemul solar luminat de Betelgeuse, au fost de acord sa ne imprumute un
generator ZPM. Drept multumire, oamenii au decis sa tina o serata pentru a le delecta
“ochiurechea” (ochiurechea este un organ de simt al extraterestrilor, care transforma impulsurile
luminoase in semnale electrice interpretate de creierul lor ca “sunete”)

Este minisunea ta, sa reproduci o parte din Simfonia a 9-a pentru ochiurechile extraterestrilor. La
sfarsitul melodiei, se poate introduce o pauza (niciun led aprins) convenabil de lunga apoi se repeta
melodia.
Obs1.
• pentru a produce sunetul A, trebuie aprins LD0
• penrtu a produce sunetul B, trebuie aprins LD0 si LD1
• pentru a produce sunetul C, trebuie aprins LD0, LD1, LD2
• pentru a produce sunetul D, trebuie aprinse LD0, LD1, LD2, LD3
• si tot asa.
Obs2. Durata notelor se considera:
• "2 T" pentru notele care arata ca prima nota E (cea mai din stanga)
• "3 T" pentru nota E cu punct (ultima nota E)
• "1 T" pentru notele care arata ca penultima nota D
• "4 T" pentru notele care arata ca ultima nota D (cea mai din dreapta)
unde 1 T inseamna FIX 1 secunda.
Obs3. Se considera echivalenta o nota de 4 T cu 2 de 2 T sau 4 de 1 T.
Obs4.
Numaratorul NUM-COMP:
• contine doua registre:
◦ unul pentru starea interna (care se incrementeaza la fiecare perioada de ceas)
◦ unul pentru iesire "out_num" (care se incrementeaza cand registrul de stare interna a
numarat o secunda)
• la fiecare incrementare a iesirii, registrul de stare interna se duce in 0
• NUM-COMP numara cand "go" este 1
• NUM-COMP isi mentine valoarea cand "hold" este 1 si "go" este 0
"ROM1" este o memorie de tip ROM de dimensiune corespunzatoare (minima)
"ROM2" este o memorie de tip ROM de dimensiune corespunzatoare (minima)
"MUX" este un multiplexor
Ceasul din sistem este ceasul generat de oscilatorul de 50 MHz de pe placa cu FPGA.
Implementati in Verilog modulele din circuitul din figura, RESPECTAND numele semnalelor si ale
modulelor / instantelor



Timp de efectiv de lucru: 90 de minute. SUBIECT_3_FARA_RAM
"Predator" vrea sa ajute METROREX sa construiasca o linie metrou pana la Aeroportul "Henri
Coanda". In acest scop, vrea sa doneze regiei, un ceas cu timer care la terminarea timpului indicat,
spulbera roca dura din drumul liniei de metrou.
Indicatorul de timp de pe ceas afiseaza initial toate segmentele aprinse; el apoi marcheaza trecerea
timpului prin stingerea a cate unuia din segmente aprinse, la fiecare FIX 500 milisecunde
La momentul T=0: 7 segmente aprinse
La momentul T=1: 6 segmente aprinse
La momentul T=2: 5 segmente aprinse
La momentul T=3: 4 segmente aprinse
La momentul T=4: 3 segmente aprinse
La momentul T=5: 2 segmente aprinse
La momentul T=6: 1 segment aprins
La momentul T=7: 0 segmente aprinse
"num1" este un numarator de tip NUM care numara in sus, din 1 in 1.
"rom1" este o memorie de tip ROM de dimensiune minima.

"dec1" este un decodor care produce mereu un singur bit de 0 pe iesire (arata care 1 digit este
aprins, restul de 3 fiind stinsi)
Ceasul din sistem este ceasul generat de oscilatorul de 50 MHz de pe placa cu FPGA.
Implementati in Verilog modulele din circuitul din figura, RESPECTAND numele semnalelor si ale
modulelor / instantelor



Punctaj (din 50 de puncte): 6 + 9 + 3 + 12p pentru top, 16p pentru "design" si 4p coding style.
Detaliu punctaj:
numarator1 (6p):
(1p) numaratorul are dimensiunea minima necesara
(1p) iesirea numaratorului are dimensiunea corecta
(3p) iesirea numaratorului se schimba exact in ritmul specificat (timp)
(1p) iesirea numaratorului se schimba in maniera necesara (valori)
rom1 (9p):
(1p) conditia de citire a memoriei este corecta
(1p) dimensiunea memoriei e corecta (numar adrese)
(1p) dimensiunea iesirii memoriei e corecta

(1p) continutul memoriei este corespunzator
(1p) intrarile sunt declarate ca intrari
(1p) iesirile sunt declarate ca iesiri
(3p) memoria e implementata corect si complet
dec1 (3p):
(1p) conditia de decodare este corecta
(2p) decodorul e implementata corect si complet
top (12 p):
(2p) memoria rom1 este instantiata corect (denumire, tip, dimensiune)
(2p) numaratorul num1 este instantiat corect (denumire, tip, dimensiune)
(2p) decodorul dec1 este instantiat corect (denumire, tip, dimensiune)
(2p) toate legaturile rom1 sunt corecte (denumire, tip, dimensiune)
(2p) toate legaturile dec1 sunt corecte (denumire, tip, dimensiune)
(2p) toate legaturile din exterior se duc spre blocurile corecte (denumire, tip, dimensiune)
design (16p):
(5p) design-ul este complet (ca numar / tip de componente) si nu are erori de sintaxa
(6p) design-ul functioneaza pe FPGA asa cum s-a cerut (intre stari trec fix. 500 milisecunde si
starile se succed in ordinea ceruta)
(1p) corespondenta semnalului de ceas <> pin e corecta
(2p) corespondenta butoane <> pini e corecta
(2p) corespondenta biti de iesire <> pini e corecta
coding_style(4p)
(4p) codul este usor de citit (indentat si spatiat similar cu exercitiul din laborator5

The basic choice reaction time is a measurement for the time it takes a human to choose between two stimuli and react to them. This time may be correlated with neural diseases according to the papers [] TODO. The logic diagram is as depicted in Figure 86

Necessary hardware:

- Machine capable of processing data and measuring passing of time
- Two LEDs usually of different colors (green and red, for example)
- Two buttons

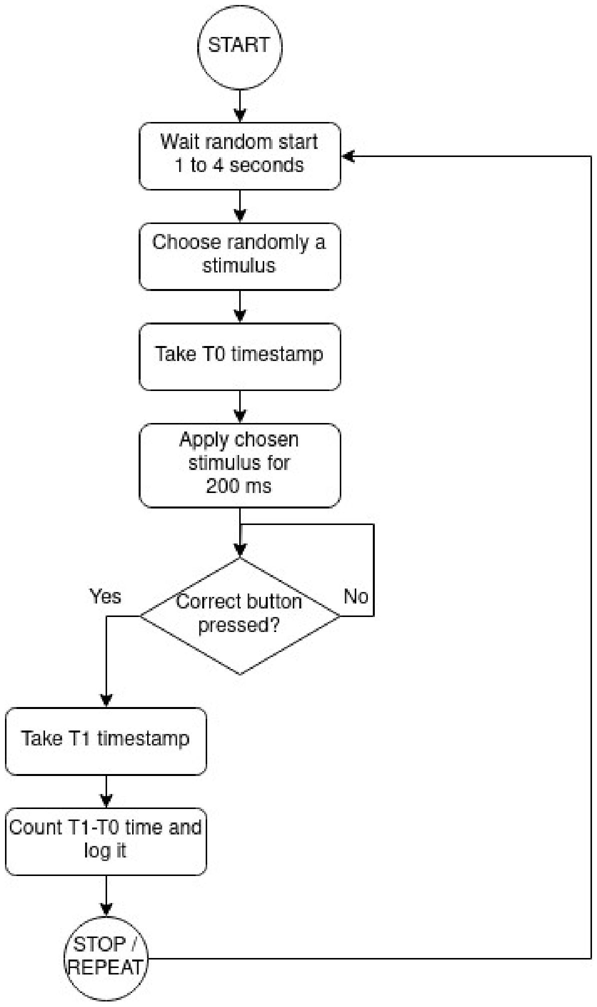
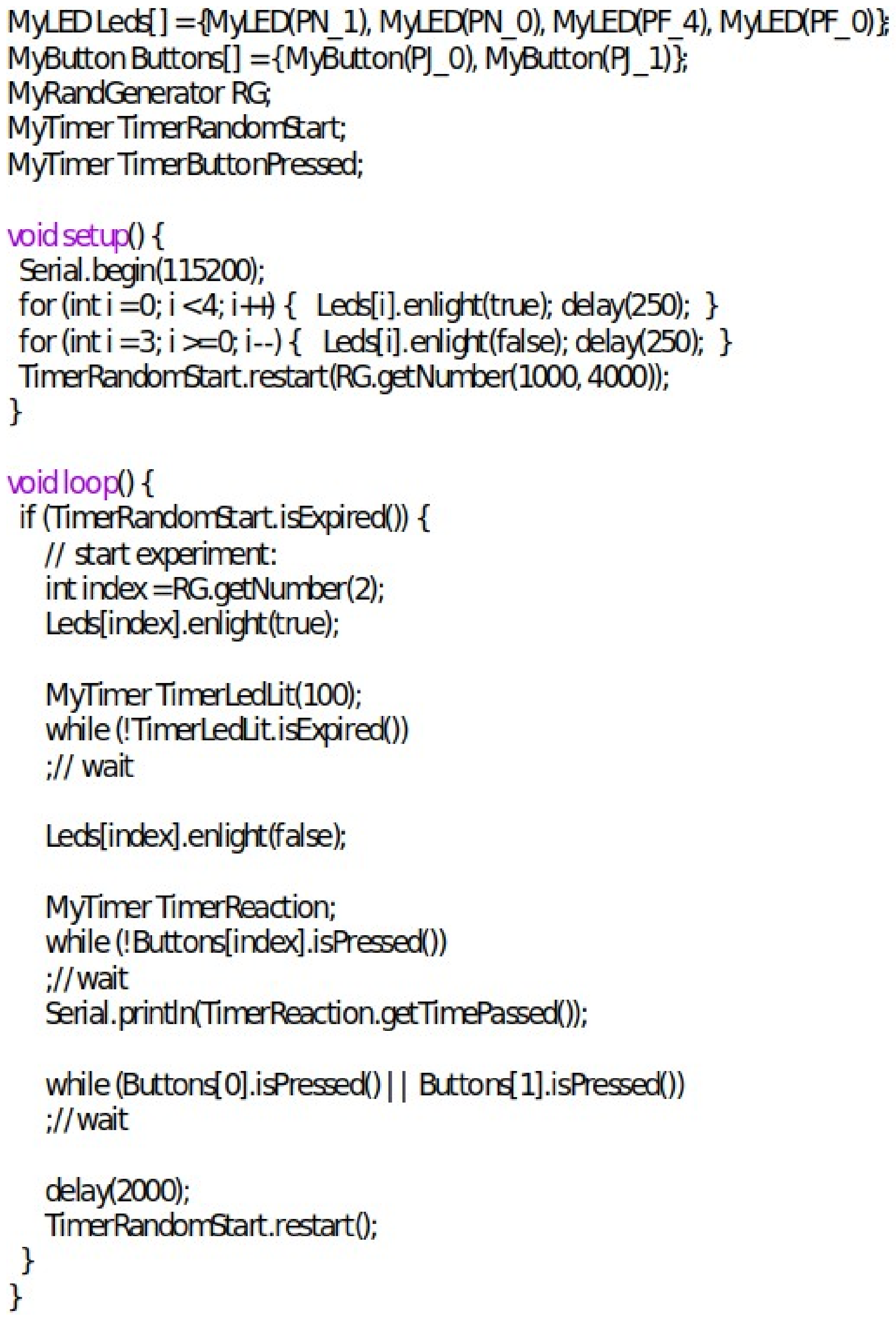

START
Wait random start 1 to 4 seconds
Choose randomly a stimulus
Take T0 timestamp
Apply chosen stimulus for 200 ms
Correct button pressed?
Yes
No
Take T1 timestamp
Count T1-T0 time and log it
STOP / REPEAT

Figure 86. Logic diagram of BCRT game

A C++ software implementation made in Arduino IDE for TI's Connected Launchpad board (XM4129C) would be the one below:

```
class MyLED {
 int mPin;
 bool mHighIsOn;

 public:
  MyLED(int pin, bool highIsOn =true) {
    mPin =pin;
    mHighIsOn =highIsOn;
    pinMode(mPin, OUTPUT); enlight(false);
  }
  void enlight(bool b) {
        digitalWrite(mPin,  b?(mHighIsOn ? HIGH : LOW) : (mHighIsOn ? LOW : HIGH));
  }
};

class MyButton {
 int mPin;
 bool mHighIsPressed;

 public:
  MyButton(int pin, bool highIsPressed =false) {
    mPin =pin;
    mHighIsPressed =highIsPressed;
    if (mHighIsPressed ==false) pinMode(mPin, INPUT_PULLUP);
    else pinMode(mPin, INPUT);
  }
  bool isPressed() {return digitalRead(mPin) ==(mHighIsPressed ==HIGH)? HIGH : LOW; }
};

class MyRandGenerator {
 public:
 MyRandGenerator() { randomSeed(analogRead(0));  }
 int getNumber(int posLimitHigh) {return random(posLimitHigh); }
 int getNumber(int posLimitLow, int posLimitHigh) {return random(posLimitLow, posLimitHigh);  }
};

class MyTimer{
 long mExpireTimestampMs;
 long mExpirePeriodMs;
 public:
 MyTimer(long ExpirePeriodMs =0) {
    mExpirePeriodMs =ExpirePeriodMs;
    restart();
 }
 void restart(long ExpirePeriodMs =0) {
   mExpireTimestampMs =millis() +((ExpirePeriodMs==0)? mExpirePeriodMs : ExpirePeriodMs);
 }
 bool isExpired() {return mExpireTimestampMs <=millis();  }
 long getTimePassed() {return millis() - (mExpireTimestampMs - mExpirePeriodMs);  }
};
```

Figure 87. BCRT's class definitions



```
MyLED Leds[] ={MyLED(PN_1), MyLED(PN_0), MyLED(PF_4), MyLED(PF_0)};
MyButton Buttons[] ={MyButton(PJ_0), MyButton(PJ_1)};
MyRandGenerator RG;
MyTimer TimerRandomStart;
MyTimer TimerButtonPressed;

void setup() {
 Serial.begin(115200);
 for (int i =0; i <4; i++) {   Leds[i].enlight(true); delay(250);  }
 for (int i =3; i >=0; i--) {   Leds[i].enlight(false); delay(250);  }
 TimerRandomStart.restart(RG.getNumber(1000, 4000));
}

void loop() {
 if (TimerRandomStart.isExpired()) {
    // start experiment:
    int index =RG.getNumber(2);
    Leds[index].enlight(true);

    MyTimer TimerLedLit(100);
    while (!TimerLedLit.isExpired())
    ;// wait

    Leds[index].enlight(false);

    MyTimer TimerReaction;
    while (!Buttons[index].isPressed())
    ;// wait
    Serial.println(TimerReaction.getTimePassed());

    while (Buttons[0].isPressed() || Buttons[1].isPressed())
    ;// wait

    delay(2000);
    TimerRandomStart.restart();
  }
}
```

Figure 88. BCRT's main function

The digital implementation as a circuit would consist of similar modules (as the software implementation):

- Timers
- RNG
- Automata (that decides what to do when)

Figure References

1. Figure 1. A digital signal (as a result of both sampling and quantization processes). Digital Integrated Circuits https://wiki.dcae.pub.ro/index.php/Introducere._Verilog_HDL
2. Figure 2. A time-sampled signal (as a result of sampling process). Digital Integrated Circuits https://wiki.dcae.pub.ro/index.php/Introducere._Verilog_HDL
3. Figure 3. A value-sampled signal (as a result of quantization process). Digital Integrated Circuits. https://wiki.dcae.pub.ro/index.php/Introducere._Verilog_HDL
4. Figure 4. Analog to digital conversion, digital processing and digital to analog conversion example. Dornelas, Helga. "Low power SAR analog-to-digital converter for internet-of-things RF receivers." (2018).
5. Figure 5. Generic FPGA Architecture Overview. https://www.eetimes.com/all-about-fpgas/
6. Figure 6. Xilinx CLB. Blue blocks are multiplexers, violet blocks are FFs and dark-green blocks are LUTs (look-up tables) https://www.eetimes.com/all-about-fpgas/
7. Figure 7. FPGA development flow. https://www.xilinx.com/applications/isolation-design-flow.html
8. Figure 8. The Nexys 4 DDR FPGA board, https://digilent.com/reference/_media/reference/programmable-logic/nexys-4-ddr/nexys4ddr_rm.pdf
9. Figure 9. Nexys 4 DDR board features, https://digilent.com/reference/_media/reference/programmable-logic/nexys-4-ddr/nexys4ddr_rm.pdf
10. Figure 10. GPIO devices on the Nexys4 DDR FPGA board. https://digilent.com/reference/_media/reference/programmable-logic/nexys-4-ddr/nexys4ddr_rm.pdf
11. Figure 11. NOT gate ("inverter") made of one pMOS (top) and one nMOS (bottom) transistor. Why do CMOS NOT gate designs differ from BJT NOT gate designs? https://electronics.stackexchange.com/questions/570389/why-do-cmos-not-gate-designs-differ-from-bjt-not-gate-designs
12. Figure 12. ANSI / IEC [6.] (right) and MIL-STD-806B [7.] (left) symbols foremost common 7/16 dual-input logic gates (elementary), with their names. Logic Gates. https://learnabout-electronics.org/Digital/dig21.php

13. Figure 13. The truth tables for the most commonly used logic gates. Nucleic Acid Computing and its Potential to Transform Silicon0Based Technology. https://www.researchgate.net/publication/291418819_Nucleic_Acid_Computing_and_its_Potential_to_Transform_Silicon-Based_Technology
14. Figure 14. Voltage levels: output requirements vs. input requirements. Logic Level Shifters for Driving LED Strips – Electric Fire Design. https://electricfiredesign.com/2021/03/12/logic-level-shifters-for-driving-led-strips/
15. Figure 15. MOSFET implementations of common logic gates. Left (NOT), middle (NAND), and right (NOR). Accessed on 24.10.2023. https://www.allaboutcircuits.com/textbook/digital/chpt-3/cmos-gate-circuitry/
16. Figure 16. The 74HC04 hex-inverter circuit, accessed on 24.10.2023, https://robocraze.com/products/hex-inverter-ic-74hc04
17. Figure 17. Representing a g(n) function which asymptotically bound f(n) function. Big-O notation. https://xlinux.nist.gov/dads/HTML/bigOnotation.html
18. Figure 18. The O-notation complexity increases with the number of elements processed. O(1) and O(logn) are usually excellent complexities, O(n) is fair, and O(n*logn) is usually considered almost decent in algorithms and circuits. Big-O Algorithm Complexity Cheat Sheet (Know Thy Complexities!) @ericdrowell. https://www.bigocheatsheet.com/
19. Figure 19. A 16-input AND gate built from 2-input AND gates.
20. Figure 20. Verilog syntax cheat sheet (1/2) accessed online on 12.10.2023 https://marceluda.github.io/rp_dummy/EEOF2018/Verilog_Cheat_Sheet.pdf
21. Figure 21. Verilog syntax cheat sheet (2/2) accessed online on 12.10.2023 https://marceluda.github.io/rp_dummy/EEOF2018/Verilog_Cheat_Sheet.pdf
22. Figure 22. VHDL cheat sheet (1/2) accessed online on 08.10.2023: https://vhdlweb.com/static/vhdl_cheatsheet.pdf, and www.ece.tufts.edu/es/4
23. Figure 23. VHDL cheat sheet (2/2) accessed online on 08.10.2023: https://vhdlweb.com/static/vhdl_cheatsheet.pdf, and www.ece.tufts.edu/es/4
24. Figure 24. The schematic of a switch checker
25. Figure 25. Verilog description of switch-checker
26. Figure 26. VHDL description of switch-checker

https://digilent.com/reference/_media/reference/programmable-logic/nexys-4-ddr/nexys4ddr_rm.pdf

61. Figure 61. A Kingbright 1-digit common anode, 7-segment display, accessed on 28.10.2023, https://www.kingbrightusa.com/images/catalog/SPEC/SC52-11EWA.pdf
62. Figure 63. The TDCG1060M clock display from Vishay, accessed on 28.10.2023, https://www.tme.eu/Document/18a20c562b919309a112ae29f2efb3bd/TDCG1050M.pdf
63. Figure 64. 7-Segment display as common anode (+) in the left side, and common cathode (-) in the right side, accessed on 28.10.2023, https://www.farnell.com/datasheets/1864135.pdf
64. Figure 67. Charlieplexing 5 I/Os to power 16 LEDs. Accessed on 28.10.2023, https://www.elektronik-labor.de/AVR/Charlieplexing.html
65. Figure 65. Time-multiplexing for the 4x 7-segment displays
66. Figure 66. 2x 4 x digit comon anode circuit found in Nexys4 DDR
67. Figure 67. Charlieplexing 5 I/Os to power 16 LEDs
68. Figure 68. A 7-segment display data-driven by 4 on/off switches
69. Figure 69. Verilog description for 7-segment transcoder circuit
70. Figure 70. VHDL description of 7-segment display transcoder
71. Figure 71. RTL implementation of transcoder with ROM
72. Figure 72. Constraints file for 7-segment transcoder circuit
73. Figure 86. Logic diagram of BCRT game
74. Figure 87. BCRT's class definitions
75. Figure 88. BCRT's main function

References

1. C.Bira, “[Digital] Electronics by Example When Hardware Greets Software”, MATRIX ROM, October 2023, ISBN 978-606-25-0845-6, online available: https://www.researchgate.net/publication/374388893_Digital_Electronics_by_Example_When_Hardware_Greets_Software
2. git@gitlab.upb.ro:Teaching/CID/atm-de.git accessed on 19.10.2023
3. https://www.iso.org/standard/31898.html accessed on 10.08.2023
4. https://www.jedec.org accessed on 10.08.2023
5. Loops & Complexity in DIGITAL SYSTEMS (Lecture notes on Digital Design in Ten Giga-Gate/Chip Era) by Gheorghe M. Stefan, link: http://users.dcae.pub.ro/~gstefan/2ndLevelteachingMaterials/0-BOOK.pdf accessed on 25.07.2023
6. "IEEE Standard Graphic Symbols for Logic Functions (Including and incorporating IEEE Std 91a-1991, Supplement to IEEE Standard Graphic Symbols for Logic Functions)," in *IEEE Std 91a-1991 & IEEE Std 91-1984,* vol., no., pp.1-160, 13 July 1984, doi: 10.1109/IEEESTD.1984.7896954.
7. MIL-STD-806B, https://bitsavers.org/pdf/mil-std/MIL-STD-806B_Graphical_Symbols_For_Logic_Diagrams_19620226.pdf accessed on 28.07.2023
8. Abels, Seth & Khisamutdinov, Emil. (2015). Nucleic Acid Computing and its Potential to Transform Silicon-Based Technology. DNA and RNA Nanotechnology. 2. 10.1515/rnan-2015-0003.
9. S.Winbers, J.Taylor, Verilog Cheat Sheet, https://marceluda.github.io/rp_dummy/EEOF2018/Verilog_Cheat_Sheet.pdf accessed on 08.10.2023
10. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd. ed.). The MIT Press.
11. C99, ISO/IEC 9899:1999 standard, https://www.iso.org/standard/29237.html accessed on 30.09.2023

Călin Bîră, PhD., is Associate Professor at the Politehnica University of Bucharest, where he teaches undergraduate and graduate courses related to programming, microcontrollers, digital design, signal acquisition and processing. He obtained his PhD in 2013 from the Politehnica University of Bucharest, with a thesis on programming environment for (energy-efficient embedded) accelerators, thus gradually shifting to an academic career. His work with the Faculty of Electronics, Telecommunications and Information Technology aims to bridge academia (research and teaching) with business and industry and their ways of designing research and development projects, thus addressing the gap between university and businesses. Călin Bîră's commitment to academic research and strengthening international collaboration and teamwork is made visible by multiple research grants and projects (DocInvest, SAVE, DEXTER, ATLAS) as well as by publications in the domain of electronics, programming and signal processing. Prior to his pursuing an academic career, Călin Bîră worked for 8 years as an engineer for global companies, specializing in low-level software and hardware design. His current interests include energy-efficient computation and embedded systems for research and commercial projects.