ECE 742 - Digital System-on-Chip Design

Course Summary Report - Day 2

By: **Gouthami Channakeshavamurthy**

A20445435

Instructor: Dr. Jafar Saniie

Class Date: 09-21-2019

Due Date: 09-27-2019

Acknowledgment: I acknowledge all of the work (including figures and codes) belongs to me and/or persons who are referenced. I did not receive any assistance from anyone or copy other sources and internet for completing this assignment.

Signature : Gouthami · C

***Abstract** - This paper presents the summary of what was taught on Day-2 of Digital System-On-Chip Design course. It includes equivalent states and reduction of state table, Sequential circuit timing, critical path and clock gating, along with other references and examples that were shown in the class. We then continue with a brief introduction to the basics of VHDL programming.*

1. INTRODUCTION:

As the continuation of past summary report, this report begins with the techniques required to reduce equivalent states was discussed. The theme of consecutive circuits closes with the dialogs on set-up and hold time. A small topic on tristate logic and its application on buses were discussed. In the second half, the design methodology of integrated circuits with the help of Gajski's Y chart and the most commonly used hardware description language, VHDL were introduced. Different type of adders and flip-flops are designed using VHDL coding both in behavioral and structural domain. The basic difference between the entity and component, concurrent and sequential statements are provided. .

2. TECHNICAL CONTENTS:

2.1 EQUIVALENT STATES AND REDUCTION:

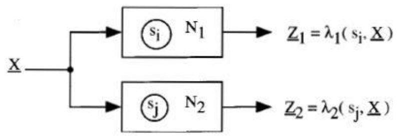
The concept of equivalent states is important for the design and testing of sequential circuits. It helps to reduce the hardware consumed by circuits. Two states in a sequential circuit are said to be *equivalent* if we cannot tell them apart by observing input and output sequences[1]. For this, we need to choose two states, say S2 and S3 that starts from the same state S1. Now different input patterns are applied to the circuit and the outputs are determined. If both the outputs are the same for all input patterns, then the states s2 and s3 are equivalent. This way of equivalence testing is practically not possible, since it requires input sequences of infinite length.

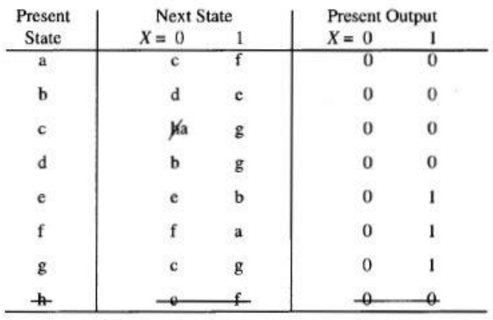
Here s_i and s_j are the two states taken for equivalence checking. When the input sequence X is applied, the two outputs are Z1 and Z2. When Z1=Z1 for all possible input sequences then $\mathbf{s_i \equiv s_j}$. In the more practical way, state equivalence theorem is used. Here we should consider both output and next state, but only one input to test is sufficient. Another method is the implication table.

(i) Each pair of rows in the state table are compared. If the outputs are different for states a and c, then the states are not equivalent, and put X. For same outputs, put implied pairs as f-g . If the outputs are same, put tick mark. a≡c.

(ii) Go to the implied pairs and check whether it contains X or tick. If the value of the implied pairs is X then the same should be placed in the square a-c.

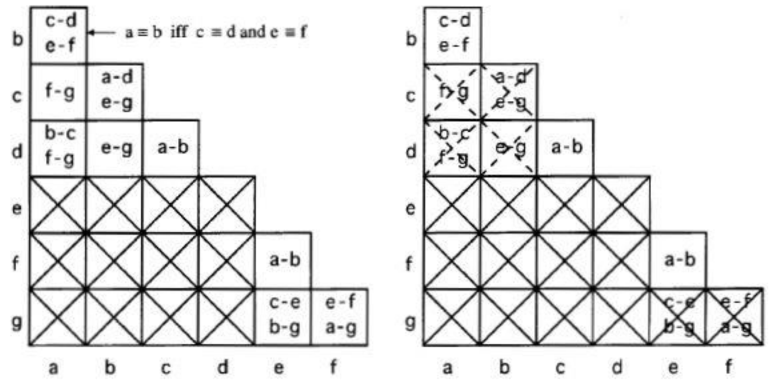
(iii) Repeat the same process until no more X can be added to the table.





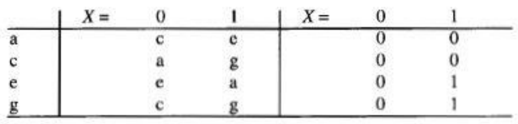
Present State	Next State X = 0	1	Present Output X = 0	1
a	c	f	0	0
b	d	c	0	0
c	~~h~~ a	g	0	0
d	b	g	0	0
e	e	b	0	1
f	f	a	0	1
g	c	g	0	1
~~h~~	~~c~~	~~f~~	~~0~~	~~0~~

State table reduction by row matching



Implication chart (first pass)

After second and third passes



	X = 0	1	X = 0	1
a	c	e	0	0
c	a	g	0	0
e	e	a	0	1
g	c	g	0	1

Final reduced table

Figure 1: Sequential Circuits and State Table Reduction

2.2 SETUP AND HOLD TIME:

In sequential circuits, the input must be stable for a specified amount of time before the positive edge of the clock (provided it is positive edge triggered circuit). If the circuit is negative edge triggered, then negative edge considered as active edge. This is called setup time. On the other hand, hold time is the amount time for which the input must be stable after the active edge of the clock. The set-up and hold time of the D flipflop is provided below.

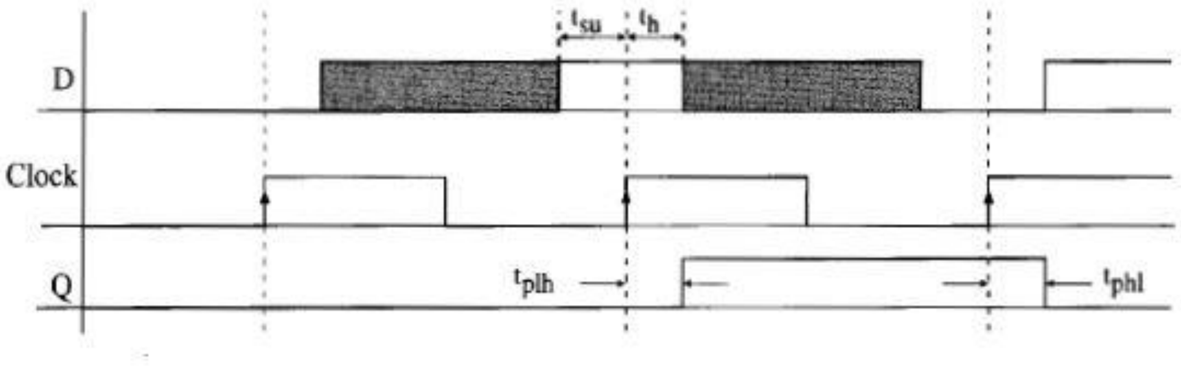


Figure 2: setup and hold time of D flipflop

Here the D input transition can occur anywhere in the shaded region. But it must be stable during t_{su} and t_h. Then only the output will be relevant to the input and prevents malfunction of flipflop.

The condition for maximum clock time t_{ck} depends on maximum time of propagation for combinational circuit t_{cmax} and maximum time for the output of the flipflop to change after the active edge of the clock, t_{pmax}. The input D of the flipflop should be stable for t_{su} (setup time) before the active edge of the clock. Thus,

$$t_{ck} \geq t_{pmax} + t_{cmax} + t_{su}$$

Using t_{ck} we can compute maximum frequency of the clock by reciprocating. The hold time is satisfied if

$$t_{pmin} + t_{cmin} \geq t_h$$

The above condition should be satisfied to avoid hold time violations in the case of feeding the output Q back to the input D through combinational network.

2.3 SYNCHRONOUS DIGITAL SYSTEM:

Most of the digital circuits contains clocks which synchronizes all the operations in the circuit. All the transitions and output changes occurs only at the active edge of the clock. The clock must be stable for a certain period of time, until all the transitions settle down.

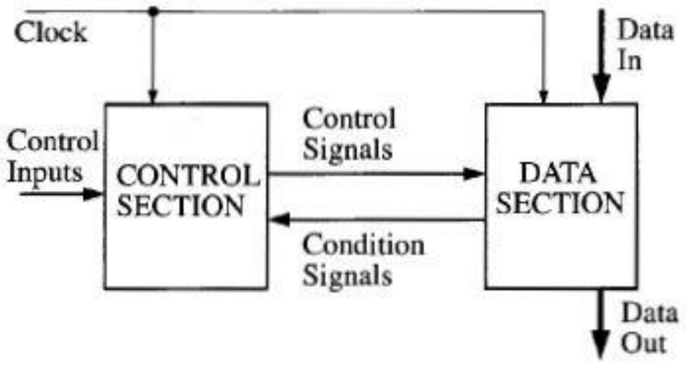
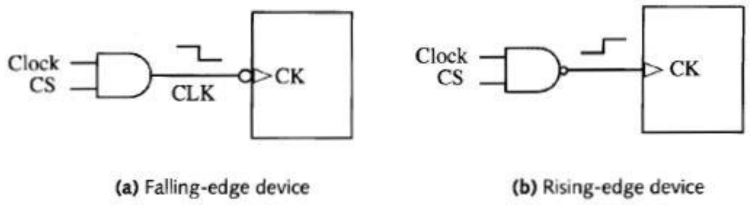


Figure 3: Synchronous digital system

The above digital system containing control and data section is synchronized by the clock. The control section gives the signals to the data section for operations. For eg.: The control section instructs the data section to add or sub the data, depending upon the opcode of the instruction. In order to avoid the glitches and hazards in the control signal after the active edge of the clock, the control signal is AND with the clock signal. Then the clock signal is pure, because, while the glitches occurring in control signal, the clock is low and the output of AND gate is also low.



The above design for the rising edge device is wrong, because, the device is rising edge but trigger at falling edge of the clock. To avoid this we have an alternate design.

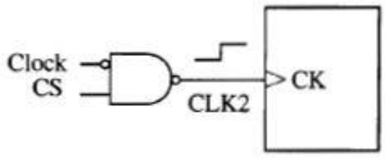
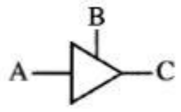


Figure 4: correct design for clock gating(rising edge device) CLK2=CS'+clock

2.4 TRISTATE LOGIC AND BUSSES:

Normally, the digital circuits have two state ‘0’ and ‘1’. But there is an another state introduced in the tristate logic called Hi- impedance Hi-Z. This tri-state logic is very much advantageous in the busses where the devices, which needs to use the bus is only enabled. Other devices are disconnected completely from the bus by Hi-Z. And also, the tristate buffers are used to introduce delay in the circuits.



B	A	C
0	0	Hi-Z
0	1	Hi-Z
1	0	0
1	1	1

The above is one of the types of tristate buffers. Other buffers can be created by just adding the bubbles wherever necessary. Here B is the enable signal. When B is 1, A propagates to C. When B=0, the output is Hi-Z.

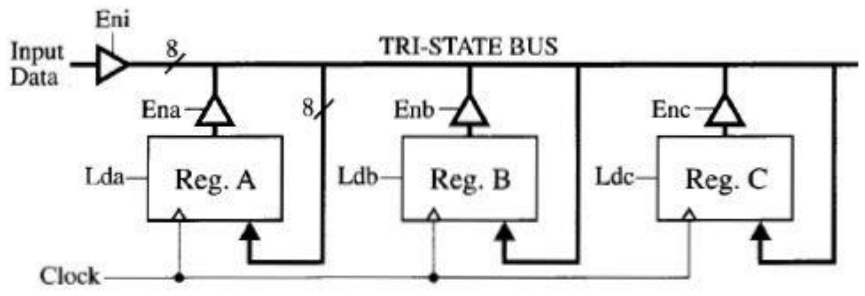


Figure 5: Bus with tristate buffer

For example:

(i)When `Ena=1, Lda=1` and `Eni=1,` the input data is copied to Reg A.

(ii)When `Ena=1, Ldc=1` and `Enc=1,` the data in A is loaded in Reg C.

Here `Enb` and `Eni` should be ‘`0`’.

2.5 ULTRASONIC TARGET DETECTION:

The ultrasonic signals to detect objects have wide variety of applications in medical, submarines, manufacturing etc. The basic principle is sending the ultrasonic signal towards the target and get information on its size, shape and distance. By measuring the time between sending the signal and receiving the echoes, the above parameters are determined. However, the echo signal is disrupted by noise very much. To avoid this digital SOC design is adopted as they are immune to noise.

The split-spectrum processing is used for ultra sonic imaging and it is implemented in a small portable device. This portable device can be easily carried to different terrains for target detection[2].

2.6 ABSTRACT LEVELS OF DESIGN:

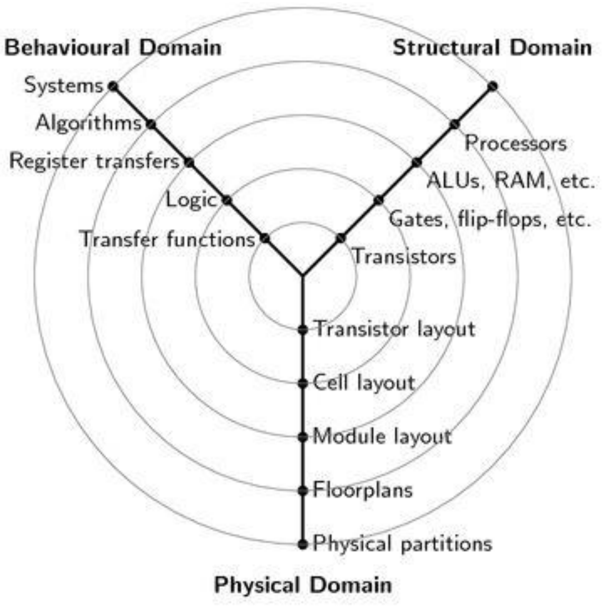


Figure 6: Gajski's Y chart

The design methodology consists of three domains, behavioral, structural and physical domain.

- The behavioral level is the highest level of abstraction where the design is explained in terms of statements like C statements. It provides the relationship between the inputs and outputs without any details of the hardware realization. By logical synthesis, we can transfer the design to structural domain.

- The structural domain describes the network in register transfer level (RTL). The connection between the gates, flipflops, modules are discussed. Simply, it tells about the hardware structure.
- The physical domain is essential for manufacturing of chips. From the structural domain, physical synthesis is done to determine layout of chip, telling the location of the transistors and the interconnection between them. Finally, the floorplan is obtained for manufacturing.

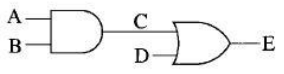
While designing we go for top-down approach in each domain. The errors in design at a certain level, should be rectified there itself do avoid propagation of errors to lower level of abstraction. Optimization is carried out at each level.

2.7 VHDL LANGUAGE:

VHDL is a hardware description language (HDL) which is most widely used. In hardware description language, by default all the statements are concurrent i.e. All the instructions will start executing in parallel, unlike in high level languages like C which has sequential execution. Using VHDL any complex digital systems can be designed and simulated in both behavioral and structural domain. Let us see the design of basic gates, adders and flipflops.

2.8 VHDL DESCRIPTION OF COMBINATIONAL CIRCUITS

(i) *SIMPLE GATE CIRCUITS:*

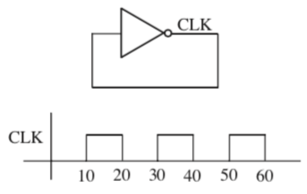


The above is the simple AND OR circuit which is described in VHDL as follows ,

```
C<=A and B after 5 ns;
E<=C or D after 5 ns;
```

Here and, or, ns are keywords which cannot be used as input or output variables. “after 5 ns” is added because, the statements are concurrent. But the second line of code cannot be executed until first one is completed. So we introduce a wait time here.

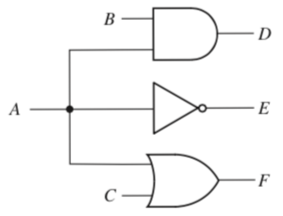
(ii) *INVERTER WITH FEEDBACK:*



The above is a Inverter feedback circuit which is described in VHDL as follows,

```
CLK <= not CLK after 10 ns;
```

(iii) *THREE GATES WITH A COMMON INPUT AND DIFFERENT DELAYS*

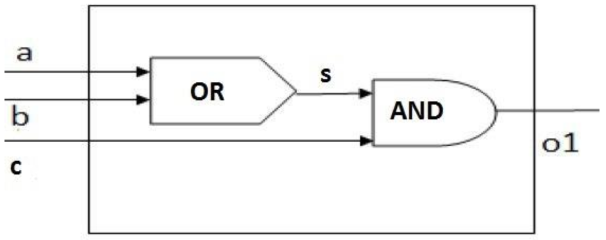


when A changes, these concurrent statements all execute at the same time

```
D <= A and B after 2 ns;
E <= not A after 1 ns;
F <= A or C after 3 ns;
```

<u>2.9 VHDL MODULE:</u>

A module is a collection of gates which are defined as entity in VHDL. They can be used in semi-custom design to ease the designing process.



```
entity two_gates is
     port(A, B, D: in bit; E: out bit);
end two_gates;
architecture gates of two_gates is signal C: bit;
begin
      C <= A and B;                                    -- concurrent
      E <= C or D;                                     -- statements
end gates;
```

The above VHDL module gives the output (a+b).c . So wherever in the design we need this function, we can add this entity as component and port map them. This is the main concept of structural coding.

Example: VHDL code for Full adder as entity

```
entity FullAdder is
     port(X, Y, Cin: in bit;                                --Inputs
          Cout, Sum: out bit);                              --Outputs
end FullAdder;
architecture Equations of FullAdder is
begin                              -- concurrent assignment statements
      Sum <= X xor Y xor Cin after 10ns;
      Cout <=(X and Y) or (X and Cin) or (Y and Cin) after 10 ns;
end Equations;
```

Using this full adder module for 1 bit addition, we can do 4-bit addition by connecting the full adder one after the other.

2.10 ENTITY AND COMPONENT:

The entity which is the VHDL module defined in a program can be added as the component in another program to make coding easier. The entity and component has more or less same syntax.

```
entity fulladder is
port( x, y, Cin: in bit;
Cout,sum : out bit; );
end fulladder;
component fulladder
port( x, y, Cin: in bit;
      Cout,sum : out bit; );
end component;
```

The entity and the component should have same inputs and outputs in same order.

2.11 FOUR BIT FULL ADDER:

With the sufficient understanding about VHDL modules, entity and component, let us design a 4 bit full adder using full adder module described earlier.

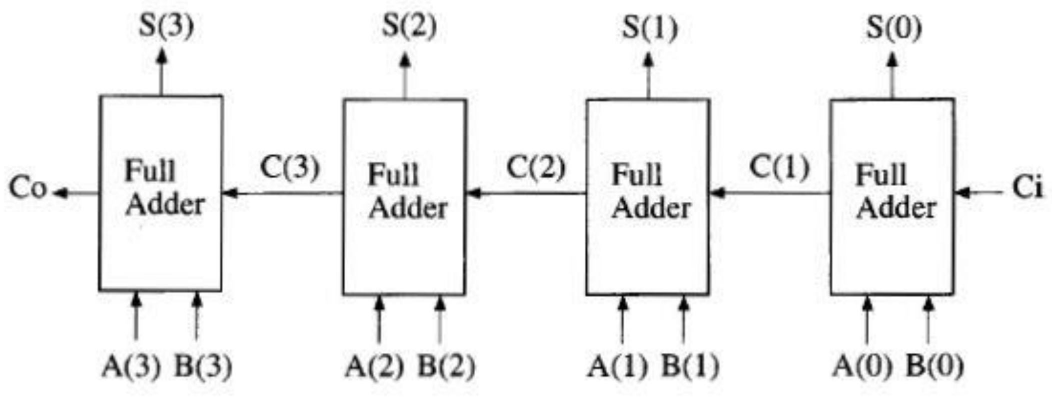


Figure 7: bit full adder

Here the fulladder is added as the component in the 4adder entity.

```
entity Adder4 is
     port(A, B: in bit_vector(3 downto 0); Ci: in bit;       -- Inputs
          S: out bit_vector(3 downto 0); Co: out bit);      -- Outputs
end Adder4;
architecture Structure of Adder4 is
component FullAdder
     port (X, Y, Cin: in bit; -- Inputs Cout, Sum: out bit); --Outputs
end component;
signal C: bit_vector(3 downto 1);            -- C is an internal signal
begin                          — instantiate four copies of the FullAdder
     FA0: FullAdder port map (A(0), B(0), Ci, C(1), S(0));
     FA1: FullAdder port map (A(1), B(1), C(1), C(2), S(1));
     FA2: FullAdder port map (A(2), B(2), C(2), C(3), S(2));
     FA3: FullAdder port map (A(3), B(3), C(3), Co, S(3));
end Structure;
```

In the program, port mapping is the main work. The order of the inputs and outputs should be strictly followed while port mapping.

2.12 ERROR CODES THAT CANNOT BE SIMULATED:

```
entity gates is
     port(A, B, C: in bit; D, E: out bit);
end gates;
architecture example of gates is
begin
     D<=A or B after 5ns;                            — -statement1
     E <= C or D after 5 ns;                        -- statement 2
end example;
```

Here d is declared as an output bit. But in 2nd statement, d is used as an input. This is an error. We can declare d as inout.

2.13 VHDL PROCESSES:

The VHDL process is used to define a sequential circuit. The process statements has a set of inputs in their parentheses. They are called sensitivity list. Whenever there is a change in sensitivity list inputs, the process execution starts.

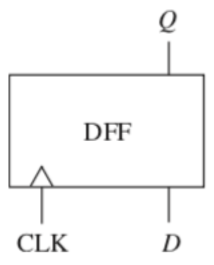
```
process(A, B, C, D)
begin
      C <= A and B;                                 -- sequential
      E <= C or D;                                  -- statements
end process;
```

The order of the sensitivity list and the order of the inputs in statements should match to avoid errors. This is somewhat like loop statements in C. Similarly, VHDl also has if-else statements.

2.14 FLIP-FLOPS:

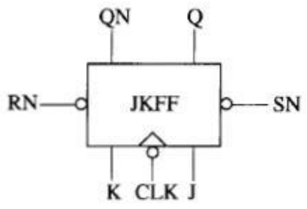
Since, flipflops are the sequential circuits, they are modeled using process statements.

VHDL Code for a Simple D Flip-Flop: By the definition of D flipflop, the input propagates to the output during rising edge of the clock.



```
process(CLK)
begin
     if CLK'event and CLK = '1'          - -  rising edge of CLK
          then Q <= D;
     end if;
end process;
```

JK flipflop: The JK flipflop had output Q and complement of the output QN. The reset RN is used to clear the flipflop and set SN is used to set the flipflop. The characteristic equation is given by $Q^+ = JQ' + K'Q$



```
entity JKFF is
     port(SN, RN, J, K, CLK: in bit;
          Q, QN: out bit);
end JKFF;
architecture JKFF1 of JKFF is
signal Qint: bit;     -- Qint can be used as input or output
begin
     Q <= Qint;                                -- output Q and QN to port
     QN <= not Qint;           -- combinational output outside process
process(SN, RN, CLK)
begin
     if RN = '0' then Qint <= '0' after 8 ns; -- RN = '0' will clear the FF
     elsif SN = '0' then Qint <= '1' after 8 ns; -- SN='0' will set the FF
     elsif CLK'event and CLK = '0' then            -- falling edge of CLK
          Qint <= (J and not Qint) or (not K and Qint) after 10 ns;
     end if;
end process;
end JKFF1;
```

Here Q_{int} is the signal which will be copied to Q after execution.

2.15 PROCESSES USING WAIT STATEMENTS:

An alternative form for a process uses wait statements instead of a sensitivity list. A process cannot have both wait statements and a sensitivity list. This process will execute the sequential-statements until a wait statement is encountered. Then it will wait until the specified wait condition is satisfied. It will then execute the next set of sequential-statements until another wait is encountered. It will continue in this manner until the end of the process is reached. Then it will start over again at the beginning of the process. Wait statements can be of three different forms:

- **wait on** sensitivity-list;
- **wait for** time-expression;
- **wait until** Boolean-expression;

Types of VHDL Delays: *Transport and Inertial Delays*

VHDL provides two types of delays—transport delays and inertial delays. The default delay is inertial delay; hence, the after clause in the preceding statement rep- resents an **inertial delay**. Inertial delays are slightly different from simple delays that readers normally assume. The second type of VHDL delay is **transport delay**, which is intended to model the delay introduced by wiring, simply delays an input signal by the specified delay time. In order to model this delay, the key word **transport** must be specified in the code.

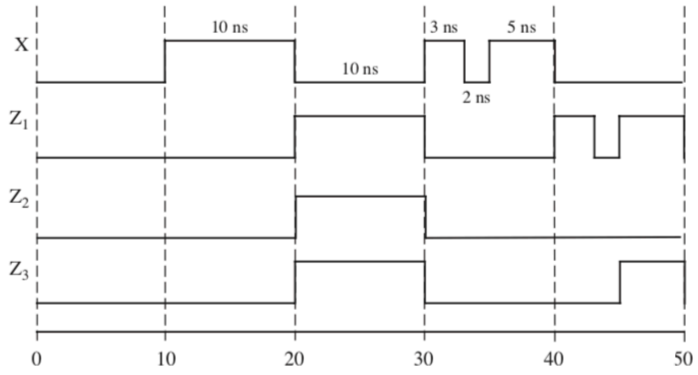
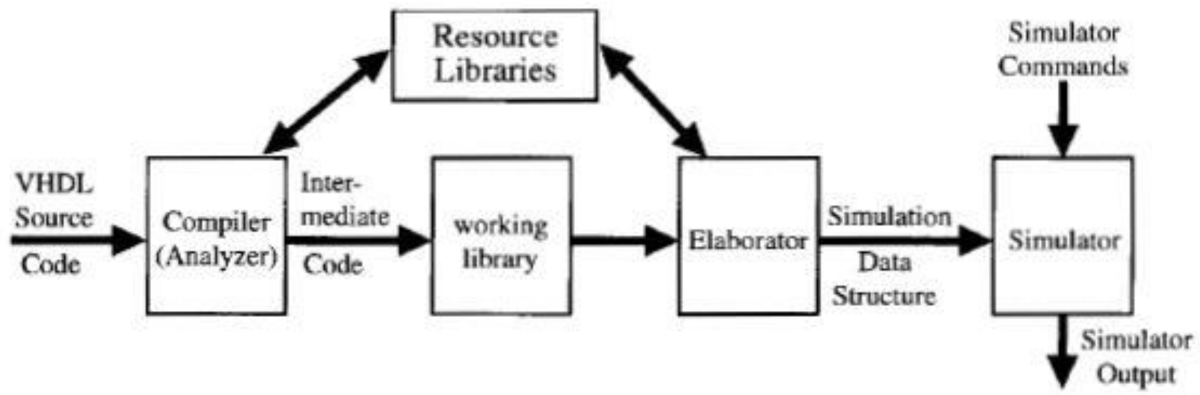


Figure 8: Transport and Inertial Delays

```
Z1 <= transport X after 10 ns;                    -- transport delay
Z2 <= X after 10 ns;                              -- inertial delay
Z3 <= reject 4 ns X after 10 ns;    -- delay with specified rejection pulse width
```

2.16 COMPILATION, SIMULATION AND SYNTHESIS:



After writing the VHDL code, we need to do simulation in order to check whether the code represents the required design with its specifications. The three steps in simulation are compilation, elaboration and simulation. The analyzer will compile the code for syntax and semantic errors. In the elaboration step, the code is converted to the form understandable by the simulator. At last, the simulation step goes through initialization phase and actual simulation. In the initialization phase, the values of the signals are initialized. During simulation, the execution of VHDL statements take place and the values are updated.

The synthesis step translates the VHDL code to circuit components and its interconnections between them. After compilation and elaboration, we can either do synthesis or simulation.

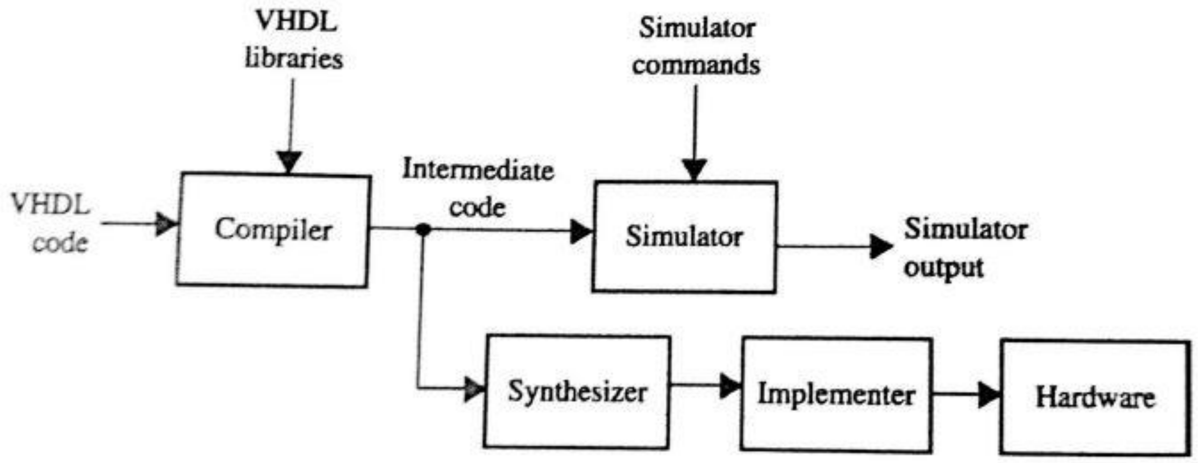


Figure 9: Compilation, simulation and synthesis

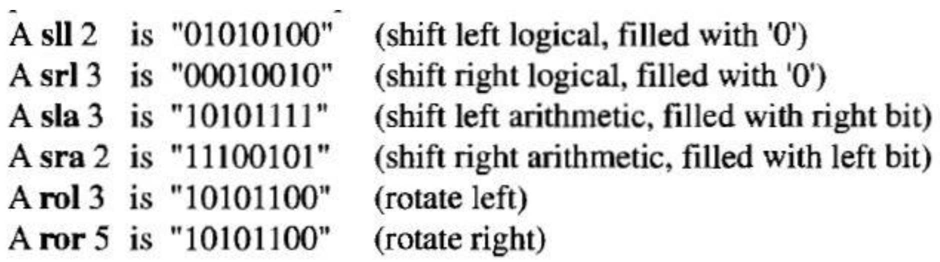
2.17 VHDL OPERATORS:

We have different kinds of VHDL operators with its own precedence. Just like the BODMAS rule, we have to apply precedence in Boolean expressions in VHDL code.

1. Binary logical operators: **and or nand nor xor xnor**
2. Relational operators: = ≠ < ≤ > ≥
3. Shift operators: **sll srl sla sra rol ror**
4. Adding operators: + — & (concatenation)
5. Unary sign operators: + —
6. Multiplying operators: * / **mod rem**
7. Miscellaneous operators: **not abs ****

Example:

```
A nand not B ror 2
Assume A=”0010” and B=”0100” Firstly,
B ror 2 = ”0001”
Secondly, not B ror 2 = ”1110”
Finally, A nand not B ror 2 =”1101” Similarly if A=”00010101” then
```



A **sll** 2	is	"01010100"	(shift left logical, filled with '0')
A **srl** 3	is	"00010010"	(shift right logical, filled with '0')
A **sla** 3	is	"10101111"	(shift left arithmetic, filled with right bit)
A **sra** 2	is	"11100101"	(shift right arithmetic, filled with left bit)
A **rol** 3	is	"10101100"	(rotate left)
A **ror** 5	is	"10101100"	(rotate right)

2.18 MULTIPLEXERS:

A multiplexer called MUX has crucial part in circuit design. A MUX can be implemented in VHDL using logical statements, conditional statements or case statements.

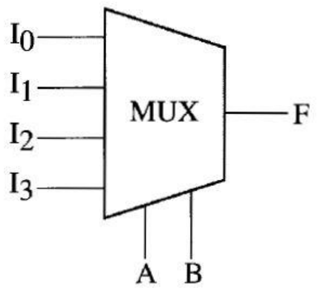


Figure 10: 4 to 1 MUX

In the above MUX, the output F is equal to one of the inputs depending on the select lines A and B. It is “00” for F<=I_0 and “01” for F<=I_1 and so on. In circuit design, whenever there is a need to select one of the many signals, MUX is used.

```
F <= I0 when Sel = 0
       else I1 when Sel = 1
       else I2 when Sel = 2
       else I3;
```

The above code is the implementation of MUX using conditional statements. Thus when sel=2, F<=I2 and similarly goes on. It is easily understandable because they are similar to C statements. Next let us see how MUX is implemented by logical equation obtained from truth table.

The logic equation for the 4-to-1 MUX is

$$F = A'B'I_0 + A'BI_1 + AB'I_2 + ABI_3$$

Thus, one way to model the MUX is with the VHDL statement

```
F <= (not A and not B and I0) or (not A and B and I1) or
     (A and not B and I2) or (A and B and I3);
```

Another way is using the case statements just like switch statements in C.

```
case Sel is
   when 0 => F <= I0;
   when 1 => F <= I1;
   when 2 => F <= I2;
   when 3 => F <= I3;
end case;
```

2.19 INFERRED LATCH:

Inferred latch is the scenario in which in case and nested if statements, if a particular signal is not assigned any value then the previous value will be assumed.

```
entity inferredLatch is
   port (input_1,input_2 : in bit;
         output_1,output_2 : out bit);
end inferredLatch;

architecture behavior of inferredLatch is
     begin
       process (input_1,input_2)
       begin
         if input_1 = '0' then
              output_1 <= '0'; output_2 <= '0';
         else
              if input_2 ='1' then
                 output_1 <= '1'; output_2 <= '1';      Inferred latch on output_2
              else
                 output_1 <= '0';
              end if;
          end if;
        end process;
       end behavior;
```

In the above program, inside second else statement, output_2 is not defined. Thus, VHDL assumes its value from previous iterations.

2.20 VHDL LIBRARIES:

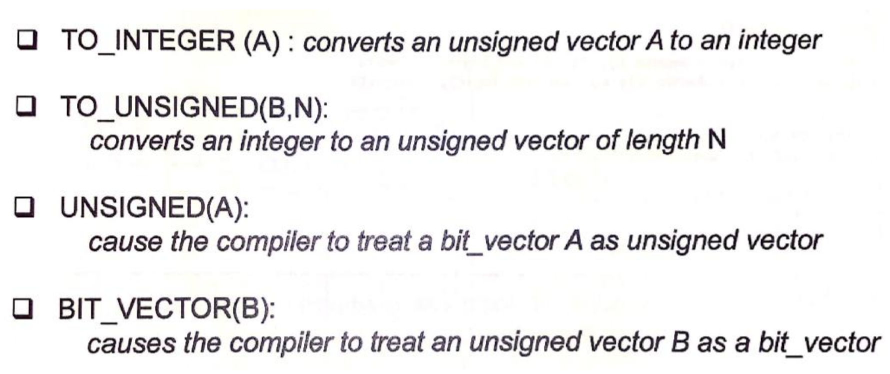
There are different libraries which makes VHDL coding easier. The one which are going to discuss elaborately is use IEEE.numeric_bit.all. This numeric bit package defines signed and unsigned type as unconstrained arrays of bit:

type signed is array (range) of bit;

type unsigned is array (range) of bit;

Overload operators on unsigned and signed numbers:
Arithmetic: + - * / rem mod
Relational: = /= > >= <=
Logical: not and or nand nor xor xnor
Shifting: shift_left, shift_right, rotate_left, rotate_right,sll,srl,rol,ror

Conversion functions:



- ❑ TO_INTEGER (A) : *converts an unsigned vector A to an integer*
- ❑ TO_UNSIGNED(B,N):
  converts an integer to an unsigned vector of length N
- ❑ UNSIGNED(A):
  cause the compiler to treat a bit_vector A as unsigned vector
- ❑ BIT_VECTOR(B):
  causes the compiler to treat an unsigned vector B as a bit_vector

2.21 ZYNQ 7000 FAMILY:

The ZYNQ 7000 family was developed by Xilinx, an extensible processing platform devices. It has about 30000 logic blocks and it is cost efficient. Normally, designers use ARM next to FPGA for flexible solutions. But it has limitations like high power consumption, performance and more board space required. In order to overcome this problem, ZYNQ 7000 is introduced. It has static memory controller with NOR and NAND flash. It also has CAN, Ethernet and USP interfaces.

3. APPLICATIONS:

3.1 Spiking neural network on FPGA:

Abstract: *This project develops a cortical neuron. Four parameters and an input are used to control each neuron. The model is implemented in hardware and allowed to learn which is the basic property of neural networks. Using FPGA we can do complex integration on differential equation of neural networks. v and u are the state variables of each neuron, membrane potential and membrane recovery variable.*

Implementation on FPGA:

The structural model consists of many HDL modules such as

- soma model(cell body and spike generator)
- propagation delay of spike
- Chemical synapse

- Electrical synapse
- STDP learning unit

The connections from the top level module with 4 neurons are given below

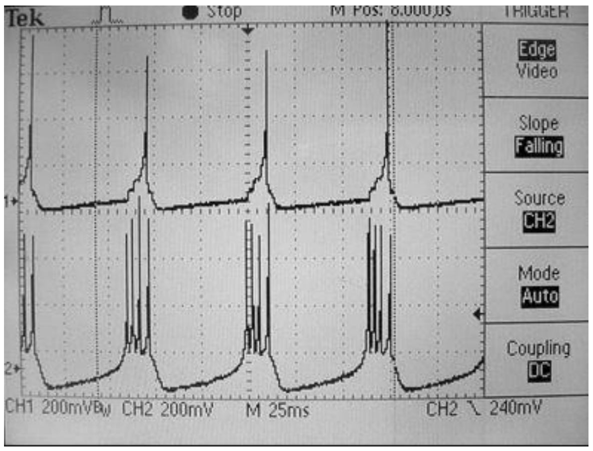
1. neuron 1 output is connected to neuron2 by a synapse(weight=0.016) and connected to neuron 3 by an electrical synapse. By doing this, the current will flow only if v1>v2 and the last output from neuron 1 is connected to LED 1.

2. neuron 2 output is connected to neuron1 by a synapse(weight=0.016) and then to LED 2.

3. neuron 3 output is connected to neuron4 by a synapse(weight=0.016) and connected to neuron 1 by an electrical synapse. By doing this, the current will flow only if v1>v2 and the last output from neuron 1 is connected to LED 3.

4. neuron 3 output is connected to neuron1 by a synapse(weight=0.016) and then to LED 4.

In the final equilibrium state, there is a spike in neuron 3 for a burst in neuron 1.

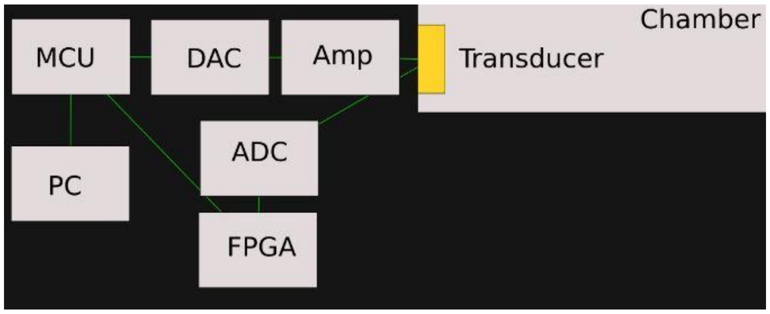


3.2 Zymeter using FPGA:

Abstract*- Zymeter is the device used to measure fermentation progress by pH, specific gravity, temperature and to determine when the beer is ready with appropriate alcohol content.*

Procedure:

Alcohol= original specific gravity-current specific gravity)*131. The specific gravity is found by ultrasonic transducer.



The MCU oscillator micro chip will generate a sine pulse which will be converted to analog by DAC. The amplified sine pulse is applied to the transducer which will penetrate through the beer, reflected by the chamber and travels back to transducer. The analog pulse is converted to digital by ADC and fed into FPGA. The FPGA will calculate the time of travel of pulse to and fro inside the chamber. This will help us to calculate speed and amplitude of pulse. From these details we can compute the density of the beer by the formula

density2=density1.speed of sound1(1+R) / speed of sound2(1+R) where R is the acoustic impedance.

<u>Future Applications of FGPA:</u>

- **Wireless/5G –** No industry is in more flux at the moment that the telecom industry. The 5G specifications are constantly being updated while telecom providers are doing what they can to extract the most out of installed 4G infrastructure equipment. In addition, telecom equipment must meet stringent size, weight, and power requirements. All of these factors argue in favor of SoCs with eFPGAs to provide instant flexibility while reducing equipment size, power, and weight.

- **Fintech/High-Frequency Trading –** As discussed above, eFPGAs reduce latency. In the high-frequency trading world, cutting latency by a microsecond can be worth millions of dollars. That alone is more than enough justification for developing SoCs with on-chip eFPGAs to handle the frequently changed trading algorithms.

- **Artificial Intelligence / Machine Learning (AI/ML) Inference and CNNs –** Convolutional Neural Network (CNN) inference algorithms rely heavily on multiply/ accumulate operations and programmable logic. Speedcore eFPGAs can significantly accelerate such algorithms using the massive parallelism made possible by including a large number of programmable DSP blocks in your eFPGA specification.[3]

4. CONCLUSIONS:

Thus in this report, discussion on sequential circuit timing is done, The necessity of tristate buffers in busses is discussed. The digital design methodology is discussed with various levels of abstraction and the VHDL codes for basic gates is introduced, along with compilation, simulation and synthesis of VHDL code. The summary report ends with the application of the discussed topics in real world.

5. REFERENCES:

1. Digital Systems Design Using VHDL, *Charles H. Roth, Lizy Kurian John*. 2008 Thomson Learning.

2. FPGA-Based Configurable Frequency-Diverse Ultrasonic Target-Detection System, J*oshua Weber, Erdal Oruklu, and Jafar Saniie*. *IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS, VOL. 58, NO. 3, MARCH 2011*

3. https://semiwiki.com/efpga/achronix/7541-when-why-and-how-should-you-use-embedded-fpga-technology/

4. https://en.wikipedia.org/wiki/Wire_recording

5. http://www.xilinx.com/video/software/sdsoc-developing-in-c-and-c-plus-plus.html

6. https://www.xilinx.com/applications/smarter-vision.html

7. https://www.youtube.com/watch?v=u0bW6lQvsVI

8. https://hackaday.io

9. http://ww1.microchip.com/downloads/en/DeviceDoc/31002a.pdf

10. https://www.quora.com/How-do-artificial-neural-networks-learn

11. https://en.wikipedia.org/wiki/Spiking_neural_network

12. https://www.quora.com/How-do-artificial-neural-networks-learn