ECSE 325 Notes

ELK

January 19, 2021

Contents

1 Design Flow **2**
- 1.1 Design Flow 2
- 1.2 Verification and Validation 2

2 HDL Technology Mapping **3**

3 VHDL Part 1 **3**
- 3.1 Design Entity 3
- 3.2 Signals 4

4 VHDL Part 2 **4**
- 4.1 Signal Assignment Statements 4
- 4.2 Components 5

5 VHDL Part 3 - Process blocks **6**
- 5.1 Process blocks 6
- 5.2 Wait statement 7
- 5.3 Types of Sequential Statements 7

6 Synchronous Circuits **8**
- 6.1 Synchronous Digital Systems 8
- 6.2 Glitches 8
- 6.3 VHDL for Synchronous Sequantial Circuits 9

7 Arithmetic VHDL and FSMs **9**
- 7.1 Arightmetic Signal Types 9
- 7.2 Arithmetic Operations 10
- 7.3 VHDL Description of Moore FSMs 10

8 FSM and Datapath/Controller Architectures **11**
- 8.1 Datapath/Controller System Design 12

9 Functional verification **12**
- 9.1 Motivation for Functional Verification 12
- 9.2 Formal Verification 13
- 9.3 Assertion-Based Verification 13
- 9.4 Test 13

10 Simulation **13**
10.1 Functional Simulation . . . 13
10.2 Event-Driven Simulation . . . 14
10.3 Cycle-Based Simulation . . . 14
10.4 Testbench . . . 15
10.5 Delays in VHDL . . . 15
10.6 Inertial Delays . . . 15
10.7 Transport Delays . . . 16
10.8 WAIT Statement Syntax . . . 16
10.9 Errors . . . 16

11 Timing **17**

12 Timing Analysis **17**
12.1 Techniques to Avoid Timing Violations . . . 17

1 Design Flow

1.1 Design Flow

Definition 1.1: Design Flow

Design Flow effectively refers to the all parts involved in transforming a concept into an actual physical implementation. In other words, it is the explicit combination of electronic design automation tools to accomplish the design of the integrated circuit.

A typical PLD (programmable logic device) design flow would follow the following:

Prompt: **Design Specification** This initial portion refers to the high-level representation of the system, including concerns regarding chip size, performance, functionality, etc., as well as any required compromises that need to be met in order to meet market and technological requirements and constraints. This process ensures feasibility.

1.2 Verification and Validation

This design needs to be verified, documented, and implemented (or synthesized). Ultimately, is our design correct?

Definition 1.2: Verification

There are two types of verification. **Dynamic Verification** is concerned with simulating the design using some description language to verify if specified behaviour is obtained. This is in contrast with **Static/-formal Verification** which encompass tool for analysis of written logic implementation. These two types go hand in hand, and complement each other.

This verification process requires a way to describe the implementation, and a way to simulate the implementation based on the description. Finally, we will need a way to compare the simulation results with our specifications to **validate** the implementation. Conveniently, we can use the implementation description for documentation as well, conveying appropriate information to relevant parties.

Using the description of our hardware, we can now map, or **synthesize**, our design onto the desired target hardware.The aforementioned description language are available in various forms, and are referred to as **hardware description languages** or **HDL**. Two of the most common languages are VHDL, and Verilog.

2 HDL Technology Mapping

The logic cells of the Altera's top-end FPGAs can be configured into many different arrangements of LUTs.

Definition 2.1: Technology Mapping

LUT technology mapping describes the acts of converting our logic gates, and logic functions into equivalent lookup tables or LUTs. More specifically, the goal is to map our netlist to lookup tables.

This mapping is to be done in such a way so as to minimize the **area** (number of LUTs), and/or the **delay** (number of levels of LUTs bewteen registers).

Reminder: The gate-equivalent **area** of a LUT is $3(2^N - 1)$ where N is the number of inputs. However, if an FPGA is comprised of LUTs of a single size, e.g. 4-input LUTs, then the **area** is the number of LUTs times the **area** of the individual LUT.

Given the above, we can decompose our logic function, or replicate gates to increase the area of our LUTs, or merge the gates instead, so as to reduce the area.

Note: How to convert circuit into N-input LUTs?

Definition 2.2: Placement

Placement refers to assigning a logical LUT to a physical location on the hardware.

Definition 2.3: Routing

Routing refers to the selection of wire segments and switches for interconnection between LUTS to achieve the desired output.

3 VHDL Part 1

3.1 Design Entity

Let us focus on the Hardware in VHDL.

Definition 3.1: Design entity

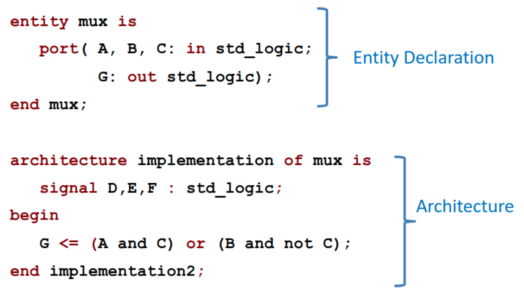
Note that VHDL descriptions consist of two parts: the **entity** declaration, and the **architecture**. The combination of these two parts is referred to as the design entity.

We can observe the following figure for an example of what a typical design entity would look like.

More specifically, the entity declaration describes the circuit as it would appear from a black box. We can further imagine an entity declaration to resemble a block symbol on a schematic (*Figure 2*). That is to say, we are only concerned with the inputs, and outputs of the circuit. We do this by labeling the ports, identifying them as either inputs, or outputs, and finally defining their types. The actual operation of the circuit is not defined here.

The description of what happens inside, is what the **Architecture** body is for. This block is comprised of an optional **Declarations area**, in which non-port signals, constants, and component types can be declared, and the **Concurrent statements area** where signals are assigned, and handled. More specifically, this is where the functionality of the circuit is described, and is marked by a *begin* and *end* statements.

Given that VHDL is meant to describe physical circuits, there is no natural or sequential ordering of operations as with higher-level programming languages. Instead, these operations or events can happen simultaneously, and so can be written in any order.



```
entity mux is
   port( A, B, C: in std_logic;
         G: out std_logic);
end mux;

architecture implementation of mux is
   signal D,E,F : std_logic;
begin
   G <= (A and C) or (B and not C);
end implementation2;
```

Figure 1: Design Entity

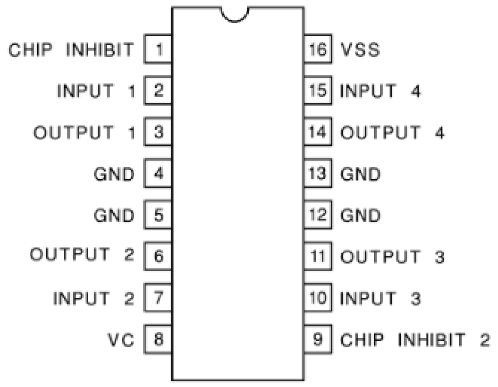


Figure 2: Schematic

3.2 Signals

Definition 3.2: Signals

Signals in VHDL are objects whose values may be changes, and have a time dimension. They are implemented as lists of time-stamped events which represent digital waveforms on wires in a circuit.

Finally, these signal can have types. Namely:

- *BIT*: can take on either 0, or 1 as a value.
- *BIT_VECTOR*: can be arrays of values, as shown below.

```
-- Used libraries must always be declared before the design entity
library IEEE;
use std_logic_1164.all;
-- Design Entity
entity example is
    port( A, B: in bit_vector(7 downto 0);
        EQ: out bit);
end example;
```

Notice how the vector is defined as an 8-element arrays with each array being of type bit. In this particular situation, the bits are described from most significant bit, to least significant bit.

Other signal types include boolean, character, string, integer, and real. The integer signal type, in particular, is very useful for implementing counters. All of the above signals are built into VHDL, and defined by the VHDL standard, however, other signal types can be user-defined. For instance, the **STD_LOGIC** type is specifically designed to model electrical signals on single wires. Additionally, a **STD_LOGIC_VECTOR** exists and is similar to a bit vector, but for standard-logic values.

4 VHDL Part 2

4.1 Signal Assignment Statements

Definition 4.1: Signal Assignment Statements

These statements are used to described the functionality of some circuit. More specifically, the **signal assignments** specify how events on some signals are created in response to events on other signals.

These statements, or concurrent statements, come in various forms. We shall observe some of the following.

Simple Signal Assignment Statement is of the following form.

```
-- Basic form
signal <= expression;
-- Example
architecture implementation of example is
    signal I1: std_logic;
begin
    C <= A or I1;
    I1 <= not B;
end implementation;
```

Selected Signal Assignment Statement behaves much like switch cases in other programming languages. When a selected signal is a given value, then the attributed circuit functionality must be used. Otherwise, the default can be used, and is defined by *others*. However, note that all options must be defined, and made explicit.

```
-- Example
architecture implementation of mux is
begin
    with C select
        G <= A when'0',
        B when others;
end implementation;
```

Conditional Signal Assignment Statement behaves like *if, then, else* statements in other programming languages. A signal is assigned a waveform if the boolean value holds true.

```
-- Example
architecture implementation of mux is
begin
    G <= A when C ='0' else B;
end implementation;
```

4.2 Components

Definition 4.2: Components

Components are used to connect multiple VHDL design units, that is to say entity/architecture pairs or design entities, together to form larger hierarchical designs. This allows for proper design partitioning. We can effectively reduce a design entity into a single component for use in another design entity.

Components must first be **instantiated**. This process occurs in the concurrent statements section of the Architecture body.

Reminder: Special care must be taken with respect to the naming scheme, and port declarations for a component, as it must exactly match the relevant entity.

As can be seen from the figure above, a component can be thought of as a piece of reusable VHDL module which can be declared within another digital logic circuit. During the instantiation process, a given architecture will make instances of an entity via component instantiations.

```
-- Typical format
Instance_label: component_name
    port map(
        component_port1 => signal1,
        component_port2 => signal2,
```

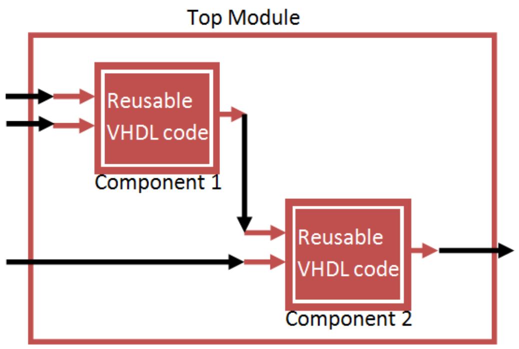


Figure 3: Component

```
        -- ...
        component_portN => signalN);
-- Example
architecture Behavioral of example is
-- Component Declaration
component sub_module
    port(x,y : in STD_LOGIC;
        s: in STD_LOGIC;
        z: out STD_LOGIC);
end component;
signal m1, m2: std_logic;
-- Component Instantiation
begin
    c1: sub_module port map(A => x, B => y , S0  => s,  m1 => z);
    c2: sub_module port map(C => x, D => y, S0 => s, m2 => z);
    c3: sub_module port map(m1 => x, m2 => y, S1 => s, Z => z);
end Behavioral;
```

Reminder: Include library of parametrized modules (LPM). This is a set of design entities which implement various logic blocks such as gates, nor gates, adders, multiplexers, counters, etc.

5 VHDL Part 3 - Process blocks

5.1 Process blocks

Definition 5.1: Process Block

The process block, or process statement, is what ultimately allows us to simulate our designs. Furthermore, operations described in the process block are executed sequentially. However, the process block, as a whole, is concurrent with the rest of the architecture.

Observe the 2 different forms a process statement can take.

```
-- Form 1
process_label: process(sensitivity list)
--declarations
begin
    --sequential statements
end process;

--Form 2
```

```
 process_label: process
--declarations
begin
    --sequential statements
    --wait statement
end process;
```

A process with a **sensitivity list** is evaluated during simulation only when an event occurs on any of the signals in the sensitivity list. Otherwise, the process is evaluated on any signal. However, a similar effect can be achieved by using a *wait* statement on signals inside the block.

Note: Outputs do not show up in the sensitivity list.

For simulation purposes, all concurrent signal assignment statements are effectively converted to process blocks.

5.2 Wait statement

Various forms of the wait statement can be used to manage a process' simulation. For instance, combinations of the different types can be used to achieve desired results.

- Wait on signal_name: suspends execution until an event occurs on either signals defined here.
- Wait until condition
- Wait for time expression

Reminder: Process blocks with a sensitivity list cannot contain wait statements, as they already depend on signals from said list.

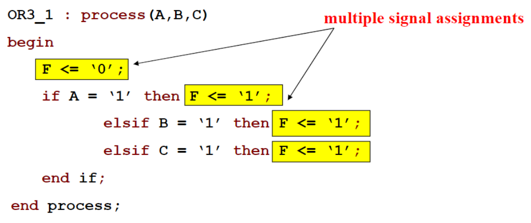
5.3 Types of Sequential Statements

Unlike the architecture body, we cannot have component instantiations, selected signal assignments, or conditional signal assignments inside of our process block. Only **simple signal assignment statements**, and conditional statements, such as *if, then, else*, work. In fact, multiple signal assignments are permitted within a process block, with only the last one taking effect.

```
-- if/then/else statements
 if first_condition then
    --statements
 elsif second_condition then
    --statements
else
    --statements
end if;
```

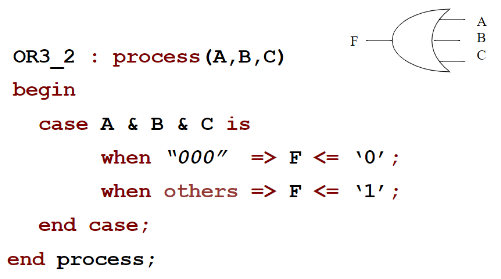
For the above, condition must evaluate to a boolean signal type. Additionally, nested conditionals are valid constructs. Case statements are also valid alternative control statements, though all test expressions must be mutually exclusive.

```
-- case statements
case control_expression is
when test_expression1=>
    --statements
when test_expression2=>
    --statements
when others=>
    --statements
end case;
```



```
OR3_1 : process(A,B,C)
begin
   F <= '0';
   if A = '1' then F <= '1';
         elsif B = '1' then F <= '1';
         elsif C = '1' then F <= '1';
   end if;
end process;
```

Figure 4: If-else conditional



```
OR3_2 : process(A,B,C)
begin
  case A & B & C is
       when "000"  => F <= '0';
       when others => F <= '1';
  end case;
end process;
```

Figure 5: Case conditional

Observe the figures below. Notice that they both implement a 3 input OR gate, however one is more efficient than the other.
The For loop is another sequential statement that fits into a process block. This is mainly used to avoid a lot of typing, and to make the code more readable. It takes the following form:

```
 -- For loops
 label : for parameter in range loop
       --sequential statements
end loop label;
```

6 Synchronous Circuits

6.1 Synchronous Digital Systems

Definition 6.1: Synchronous

In a **synchronous** sequential circuit, outputs change only at times specified by events on a small subset of input signals. These signals usually involved *clocks* or asynchronous control signals. The outputs are therefore synchronized to the clocks or control signals. Therefore, the values of the inputs only matter around the times of the synchronizing events.

Reminder: A periodic, repetitive synchronizing event is referred to as a **clock**.

These synchronous systems are described as **Register-Transfer** systems, where data flows between registers through combinational logic blocks.

6.2 Glitches

Given the overlap between inputs and synchronizing events, any glitches that may arise at any other time will have no effect on the circuit's functionality, assuming the glitch is small enough. However, because these glitches occur during a clock edge, one must wait for the glitches to completely die out.
Synchronization is then typically done via edge-triggered flip-flops as per the above figure.

Definition 6.2: Synchronous control signals

Situations in which a control input is only applicable during a clock edge. This should be the case for most sequential circuit control inputs.

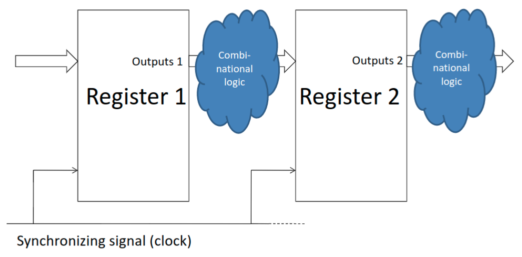


Figure 6: Register transfer system

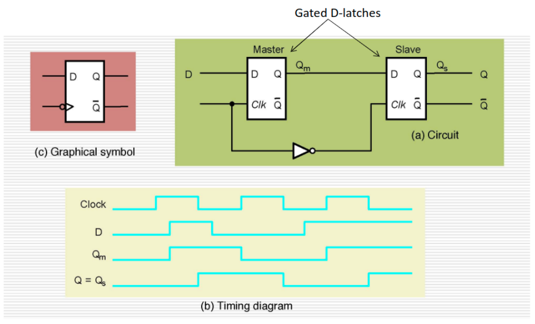


Figure 7: Master-Slave D flip-flop

6.3 VHDL for Synchronous Sequantial Circuits

Only the presence of a **clock signal** differentiates a synchronous sequential circuit from an asynchronous/combinational circuit.

Note: The occurrence of a clock edge or clock transition event is represented by a **TRUE** value of the **EVENT** attribute of the clock signal.

Consider the following:

```
Clk'event
```

The *apostrophe* followed by a name indicates a signal attribute. This is effectively a function on a signal's list of events. The actual **event** attribute can only take on a value of 1 or 0 depending on the signal.

Edge-triggered clocking can then be implemented as follows:

```
--Here, cocking can only occur during a positive clock edge
elsif Clk='1' and Clk'event then

--Using rising_edge instead
elsif rising_edge(clk) then
```

Alternatively, one can use the **rising_edge** function.

7 Arithmetic VHDL and FSMs

7.1 Arightmetic Signal Types

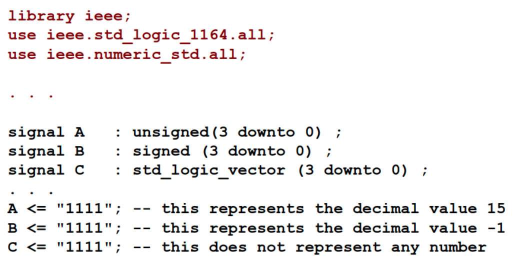
Reminder: the std_logic_vector signal type just described a collection of bits, and has no arithmetic operations defined for it. This must be defined.

The way to represent a signed or unsigned number in VHDL is to defined two new signal types. Using the **ieee.numeric_std.all** package, we much explicitly say what type of signal we are dealing with.

Note: the **ieee.numeric_std.all** package should come after the **iee.std.logic_1164.all** package.

```
signal X : SIGNED; -- this is 2's complement
signal Y : UNSIGNED;
```

Observe the following as an example:
Furthermore, VHDL **concatenation** operator can be used to combine two vectors into a single longer vector as follows:

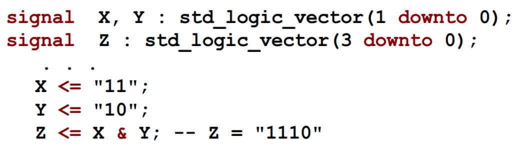


```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

. . .

signal A   : unsigned(3 downto 0) ;
signal B   : signed (3 downto 0) ;
signal C   : std_logic_vector (3 downto 0) ;
. . .
A <= "1111"; -- this represents the decimal value 15
B <= "1111"; -- this represents the decimal value -1
C <= "1111"; -- this does not represent any number
```

Figure 8: (un)signed



```
signal  X, Y : std_logic_vector(1 downto 0);
signal  Z : std_logic_vector(3 downto 0);
    . . .
  X <= "11";
  Y <= "10";
  Z <= X & Y; -- Z = "1110"
```

Figure 9: concatenation

Note: Various type casting can be applied to signals for different effects, namely conversion between (un)signed, and std_logic_vector. We can perform the same type casting between (un)signed, and integer. However, note that the conversion between std_logic_vector to an integer is not possible with one step, and requires an intermediary step where the std_logic_vector is first converted to an (un)signed representation.

Consider now some array of std_logic_vector. The individual bits of said array can be manual written out via **positional association**. However, this is a lot of writing. we can label all bits of the same value at once, with the others assignment, referred to as **named association**.

```
--positional association
DATA4 <= ('1','0','1','1');
DATA4 <= "1011"

--named association
DATA4 <= (3 => '1', 2 => '0', others => '1');
```

7.2 Arithmetic Operations

If a given unsigned signal X, from N down to 0, we say that this is a $N+1$ bit unsigned binary number. where N is the most significant bit, and 0 the least. An addition of two of this type, refers to an N+1 bit adder, and will have no carry in or carry out. The final sum is also $N+1$ bits.

Note: at least one operand on the RHS must have the same bit length as that of the LHS.

To add a carry input and output, we can simply add **Cin** to our summation or adder statement.

7.3 VHDL Description of Moore FSMs

Finite state machines should be described using 2 process blocks: one for the state update and state storage, and another for the output logic. Moreover, We can define a state signal as follows:

```
--where State_type is the name for the new signal type, and list of values are the values the signal
    type can have. These can be symbolic state names, or have numerical values
TYPE State_type IS (list of signal values);

--Example
```

```
 architecture behavioural of FSM is
TYPE state_signal IS
     (RESET_STATE, S1, S2, S3, DONE);
SIGNAL state : state_signal;
 begin
 --...
```

As shown in the above example, the signal type declaration must be placed in the declarations area of the architectures, much like components. Observe *Figure 10* for an example:

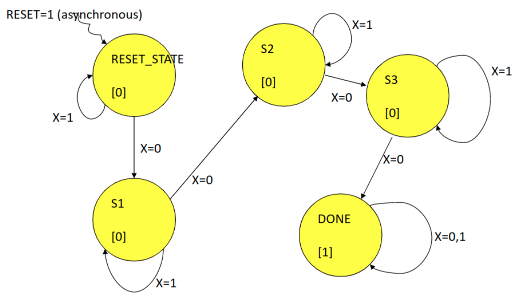


Figure 10: Moore Machine

Note: Case statements are very useful for describing finite state machines. Use one case for each possible state value.

Definition 7.1: State Assignment

When implementing an FSM in digital hardware, we need to map the abstract state labels to digital values. Think about mapping out *Figure 10* onto bit vectors.

This case assignment can be done in various ways, but generally we want the minimum number of bits, while avoid to change more than one bit at a time.

Definition 7.2: One-Hot State Assignment

For every state, only one bit at a time is set to high. This makes decoding the state simple, and makes detection of illegal or invalid states easy.

In general, when designing circuits with large number of states, state assignment will have substantial effects on the cost of the final implementation, and so an exhaustive approach to state assignment is not practical. Instead, we should design the state diagram with state labels, and let the CAD tool do the state assignment.

Note: Make the first state in the state type list to be the reset state (or some other initial state).

8 FSM and Datapath/Controller Architectures

Synchronous digital systems are often specified by describing the transfer of data between registers, and the processing applied to the data.

Definition 8.1: Register-Transfer Level Design

RTL design focuses on describing digital systems at the register level, where the system consists of a set of registers interconnected through logic blocks that apply data operations of various types. Hardware descriptions expressed at the register transfer level are deemed **synthesizable**.

Designing the synchronous digital systems can be done via the Datapath/Controller framework, in which the **Datapath Modules** process and manipulate data, and the **Control Modules** generate control signals that modify the processing of the datapath modules. The latter is also controlled by status signals sent from the datapath modules. This entire approach is referred to as the **FSMD**, or FSM with Datapath.

The basic workflow of this framework is as follows:

- Control Signals: Tell the datapath what to do, when to do it, how how to carry out various actions.
- Status Signals: Comes from the datapath to tell the controller how it's doing, such as when it's ready for new data, or busy processing data.
- Control Inputs: Comes from outside the system to start things up, stop processes, or change the mode of operation.
- Control Outputs: Sent to the outside to indicate the state of the system.

8.1 Datapath/Controller System Design

To design a datapath-controller system that implements a given function, we must do the following:

1. Describe the function to be performed at a high level. This is easiest to describe via pseudo-code. For instance, we can define the new values to be stored in the registers at each clock event, given that certain conditions are met.

   Reminder: RTL describes how the values stored in registers are transformed and combined.

2. Determine what datapath elements are needed. Reading off our pseudo-code, we should be able to deduce the resources we will need (i.e. registers, comparators, multiplxers for input control, etc.)
3. Specify the interconnections of the datapath elements. What data sources will we need? Think of the various inputs, and intermediary inputs to the various components.
4. Identify the controller input and output signals. For the inputs, we can gather this information from the control signals of the datapath elements (i.e. resets, start, clock, status signals, etc.). For the outputs, consider what we need and want to derive from our circuit.
5. Sketch the sequence of control signal values needed to carry out the desired function. For instance, when should a load signal be asserted, or cleared. For counters, when should their count enables be asserted? We can think of this as the logic flow.
6. Design a finite state machine that will implement the required sequence. The simplest way to do this is to specify a different state for each line, or clock period in the RTL description.
7. Simulate the complete system to verify the proper execution of the desired function.
8. Implement and test complete system. Go back to step 2 for any implementation issues.

Note: Go over code near the end of Lecture 18

9 Functional verification

9.1 Motivation for Functional Verification

Given some new circuit implementation, how do we know that a design is correct,the design behaves as expected, of that the hardware is correct? To that end we must do **Functional Verification**. This would lower product time-to-market by reducing hardware turn-around time, and the volume of bugs. Moreover, it'll also lower development costs via **Early User Hardware**.

Essentially, It is crucial to reduce bugs, as over time, it becomes more expensive to fix. For instant, initially, a bug found in the early designer stages has little cost. A bug during chip or system simulation has moderate cost, as more debug time is required. A bug in system test requires new hardware. But worst of all, a bug in the customer's environment can cost hundreds of millions in hardware, and tarnished brand image.

Verification will be based on **test pattern generation**, **reference models**, and **result checking**.

9.2 Formal Verification

However, simulation and emulation cannot account for all cases, as corner, or edge, cases may be missed. Although this remains a solid method for initial debugging, we need more. This is where formal verification comes into play. Where as formal verification requires some type of initial user stimuli to test, formal verification starts from the desired result, and attempts to trace its way back to a counter-example, or a situation where some test fails. **Perfect Verification** would entail all of the above, as well as all remaining combination and permutations of inputs. However, this is not practical on large pieces of design. Furthermore, verification can only show presence of errors, and not their absence. Given enough time, errors will be uncovered.

Moreover, Formal Verification can be referred to as **Symbolic Simulation**, as the goal is to transform the formula for a circuit to the one for specification by mathematical reasoning using axioms, and theorems. This becomes more a mathematical proof of the correctness of the design, which is then simulated.

With this said, verification is a time consuming process, and costs a decent amount. Despite this, it is indispensable to create revenue, and design functionally correct systems to the benefit of all parties.

9.3 Assertion-Based Verification

Fusing simulation and formal verification is not easy given that simulation patterns cannot be used for formal verification, as properties cannot be simulated as it is. This is where **Assertion-based verification** comes into play. In fact, grants must always occur within 6 cycles of request, except when reset occurs.

9.4 Test

The purpose of tests is to verify that the design was manufactured properly. In other words, we are trying to detect imperfection in patterning, impurities in doping, dust, etc. This is different from verification, where we seek to ensure that the design meets the functionality intent.

10 Simulation

10.1 Functional Simulation

Definition 10.1: Functional Simulation

The purpose of simulation is to determine if the circuit performs correctly, as well as to determine if timing constraints are met. However, **functional simulation** is only concerned with the first half, and so we do not care about propagation delays, and other timing issues.

Note: Mapping a design to some target hardware is not of concern in this type of simulation.

Definition 10.2: Timing Simulation

This type of simulation is concerned with the second half, that is to say timing constraints. This requires that the design be mapped onto some target device.

Note: No information is provided in terms of propagation delays or other timing information for the function simulation. It is more for checking correctness of our behavioural descriptions only.

Function simulations just looks at the RTL descriptions, and has no need for hardware synthesis, whereas timing simulation requires the latter. Propagation delays are then extracted from the hardware mapping. This additional information is added back into the design or model.

Simulation of RTL descriptions usually take on of two forms. **Event-Driven**, or **Cycle-Based**.

10.2 Event-Driven Simulation

Definition 10.3: Even-Drivent

An event refers to some change in logic value at a node, at a certain instant of time. Event-driven the only considers active nodes, where events do occur. This is more efficient.

This method of simulation, in which all nodes are visible, allows for glitch detection.

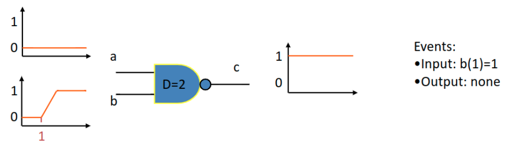


Figure 11: EDS1

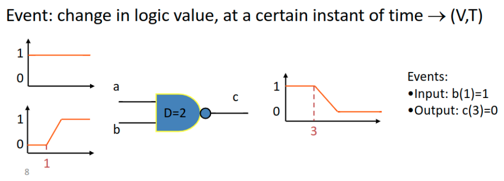

Event: change in logic value, at a certain instant of time → (V,T)
1
0
1
0
1
a
b
D=2
c
1
0
3
Events:
•Input: b(1)=1
•Output: c(3)=0

Figure 12: EDS2

Definition 10.4: Timewheel

A list of all events not processed yet, sorted in time. This timewheel is used to manage the relationship between components. In fact, when an event is generated, it is put in the appropriate point in the timewheel to ensure causality.

The main function of an Event Driven simulator is to detect events, and schedule gate simulations in response to them. If no events occur, implying that there are no net changes, then no gates will be simulated. Consider the following, if an input, or output out of some gate does not change, then there is no need to simulate it. This saves time, as it increases simulation speed.

10.3 Cycle-Based Simulation

Definition 10.5: T

is method takes advantage of the synchronous nature of most digital designs, that is to say, the entirety of the design depends on a clock. More specifically, refers to state elements of some **synchronous circuit** that changes on the active edge of a clock. The simulation can then compute steady-state response of the circuit at each clock cycle.

Event-Driven simulation works in such a way that each internal node needs scheduling, and functions may be evaluated multiple times. Cycle-based simulation does so with no information delay, and only at boundary nodes.

This is only good for synchronous systems. Unfortunately, the latter does not detect glitches, and setup/hold time like the alternative. However, cycle-based simulation tends to be 10x-100x faster than Event-driven simulation.

Note: This class will be based on Modelsim software

10.4 Testbench

To simulate a design, we must first set the inputs to desired patterns. This can be done via a HDL entity known as a **Testbench**. This testbench is never synthesized into real hardware.

Note: The testbench entity is unique in that it has no inputs or outputs

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_adder_combinatorial is
end tb_adder_combinatorial;

architecture test of tb_adder_combinatorial is
signal OP1 : std_logic_vector(3 downto0);
signal OP2 : std_logic_vector(3 downto0);
signal SUM : std_logic_vector(4 downto0);
begin
--Instantiate Design Under Test (DUT)
    dut : entity work.adder_combinatorial --Here we use entity instantiation instead of componenet
        instantaiation. Doing it this way means we don't have to declare any components. Furthermore
        , work refers to the default location for compiled entities.
    port map(OP1 => OP1,
            OP2 => OP2,
            SUM => SUM);
end architecture test;
```

10.5 Delays in VHDL

There are two ways of specifying a delay in VHDL: **Inertial**, and **Transport**.

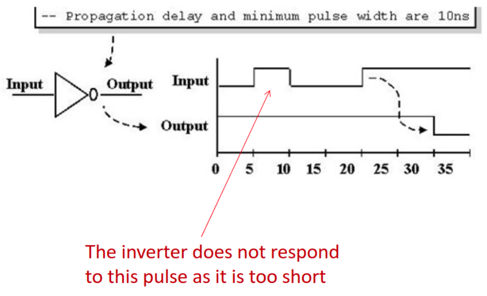
Note: When we assignment values to things, the operation will typically be infinitesimally short time-wise. This delay is called the **Delta Delay**

10.6 Inertial Delays

Inertial delays model the propagation delay and response time of logic gates. That being said, gates have a minimum input pulse width to which they will respond to. By default, we can assume this to be the propagation delay.

Reminder: Propagation delay is the changes in the output of a logic gate lagging behind changes in its input.

```
--inertial delay specified by after, and inertial keyword is option as it is assumed to be inertial
    by default.
A <= inertial B after 5ns;
--we can change the minimum input pulse width via the reject clause
A <= reject 500ps inertial B after 5 ns;
--The reject value must always be less than the delay time.
```



```
-- Propagation delay and minimum pulse width are 10ns
```

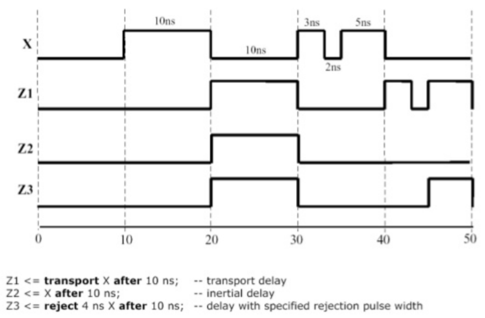
Input
Output
Input
Output
0 5 10 15 20 25 30 35
The inverter does not respond to this pulse as it is too short

Figure 13: Minimum Pulse Width

10.7 Transport Delays

Transport delays are used to model the propagation delays of wires, interconnections, transmission lines, etc. There is no minimum pulse width for these structures, and thus only propagation delay is important.

```
A <= transport B after 5 ns;
```



```
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
```

Figure 14: Delay Comparison

10.8 WAIT Statement Syntax

We can susupend an execution until conditions either become true, or the timer has been reached.

```
-- for 25ns is the timer here, without it, the process would wait forever until conditions have been
    met.
wait until (clock = '1' or enable /= '1') for 25ns;
```

Note: An indefinite wait can be added to a process block to cause the simulator to schedule no more events, when running it.

10.9 Errors

Using an **Assert** statement will allow Modelsim to print out an error, namely to tell us where the code went wrong. However, we won't know much more than that. To find out under which condition the error arised in, we need to use VHDL **IMAGE** attribute. This converts a signal value to a string, so it can be displayed by the **Assert** statement. This will allow us to write more useful assert statements.

11 Timing

Note: Delays would not be explicitly written in the VHDL description. This is because we don't know what the propagation delays will be until after the hardware mapping, or synthesis, is done. Quartus will derive the delay values as part of the synthesis process (.sdo file).

Gate-level timing simulation will keep track of various timings, and delays associated with say routing, and cell delays. This differs from gate-level functional, and behavioral simulation (RTL), that only tracks the logic.

For complex circuits, timing simulations are not done. Instead, functional simulations are done, followed by **static timing analysis**. This simply checks to see if specified timing constraints are met by the synthesized circuit. For example, one constraint might be that the minimum clock frequency be 100MHz.

In synchronous systems the maximum clock rate, or minimum clock period, depends on the propagation delay of the critical path, and the setup time of the registers. As per the above figure, the register input must be stable

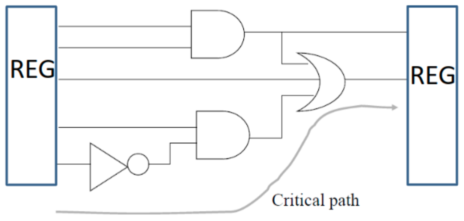


Figure 15: critical path

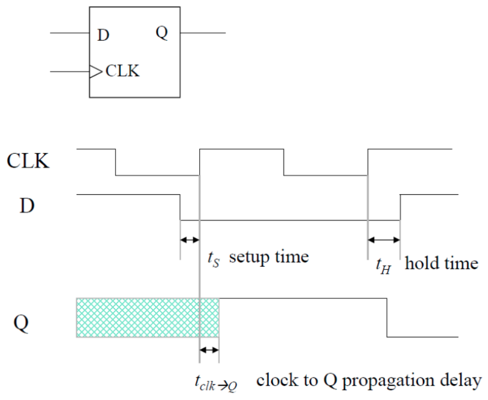


Figure 16: EDS2

for a certain length of time before the clock event. This is the **Setup Time**. Alternatively, the register input must also be stable for a certain length of time after the clock event. This is the **Hold Time**. Violating these constraints would result in a metastable state where the voltage takes a while to settle. Data input to a register must therefore be stable before, and after a clock event. For a synchronous system, this must apply to all registers.

12 Timing Analysis

Definition 12.1: Clock Skew

The relative delay between arrival of a clock at different flip-flops. This can cause timing problems. This propagation delay to various components is caused by interconnects.

The difference between the required setup/hold time and the actual time provided is called the **Slack**. For setup and hold time constraints to be met, the setup, and hold slack must be positive.

12.1 Techniques to Avoid Timing Violations

We can **insert delay elements** on the clock path to avoid setup time violation, or on data path to avoid hold time violations.

Reminder: In order to ensure that setup/hold time constraints are satisfied, a timing simulation is not necessary. As mentioned previously, **static timing analysis** is sufficient, since it checks the timing of each path between registers in the system. There is no need for simulation.

Static Timing Analysis would, given a minimum clock frequency constraint, determine the setup and hold time slacks for each path between registers in the system. Any negative slack values will result in a failed analysis. The design must then be redone.