Arnav Goyal

Digital Design

Self Study

Preface

The purpose of this document is to act as a comprehensive note for my understanding on the subject matter. I may also use references aside from the lecture material to further develop my understanding, and these references will be listed here.

This document should eventually serve as a standalone reference for learning or review of the subject matter. There is also a lot of organization within these documents, please refer to the table of contents within your PDF viewer for ease of navigation.

References

- Provided Course Materials from ECE 2277, ECE 3375
- Digital Design with an Introduction to the Verilog HDL - 5e - M. Mano, D. Ciletti
- Verilog Complete Tutorial - VLSI Point (YouTube Link)

Part I Digital Design

Finite State Machines

A **finite state machine** or[1] (**FSM**) is a system that can be in a finite number of **states**, accept a finite set of **inputs**, producing a finite state of **outputs**.

- Listing these states, the possible transitions between states based on the input, and the conditions required for each possible output, provides a complete function description of an FSM.

A FSM is a general concept, a mathematical concept. However, when dealing with a specific subset of then called **deterministic FSMs**, we can actually implement them with sequential circuits.

- An FSM is **deterministic** if every combination of current state and input results in only one transition (no probabilistic transitions).

[1] Sometimes an FSM is called a Finite State Automaton

Mathematical Formalism

There are two types of FSMs. A **Mealy** and a **Moore** FSM.

- The output of a Moore FSM depends on only the state it is currently in.
- The output of a Mealy FSM depends on transitions between states.

Lets make things more organized by introducing some basic mathematical notation to this.

- Let the set of all possible states in an FSM be denoted by Q, lets also call the starting state Q_0
- Let the set of all possible inputs be called I
- Let the transition function be denoted $\delta(Q, I)$, it determines the next state as a function of the current state and next input, we say: $\delta : Q \times I \mapsto Q$
- Let the output function be given by $f(\cdot)$, it determines the output as a function of something.

This means that any FSM M can be described by the following basic description[2]

$$\mathrm{M} = (Q, I, Q_0, \delta, f)$$

We can differentiate mathematically between Mealy and Moore FSMs through the description of their output functions. Let the set O be the set of all possible outputs.

- A Moore FSMs output function is defined[3] as $f : Q \mapsto O$.
- A Mealy FSMs output function is defined[4] as $f : \delta \mapsto O$

[2] Here the functions are written as variables for clarity, we also probably cant define these functions in any way other than a truth/state table. For now just assume that they can.

[3] A Moore FSMs output depends ONLY on its current state

[4] A Mealy FSMs current output depends on the SPECIFIC transition it is experiencing

Clock Domain Crossing

Clock Domains & Metastability

So far we've only discussed **synchronous** circuits[5]. In the real world, it is extremely common to have **asynchronous** circuits[6]. Consider some circuit C with two design blocks $C_1, C_2 \subseteq C$

- we say that the circuit is synchronous if the only clock within C is f_1 or derivatives[7] of f_1
- we say that the circuit is asynchronous if there are multiple clocks - if C_1 is clocked by f_1, and C_2 clocked by f_2

A problem occurs[8] when trying to send data from C_1 to C_2, we are trying to send data at some frequency f_1, and trying to sample it on some frequency f_2. Sending data through different clock domains like this is called **clock domain crossing** or (**CDC**)

Whenever this occurs, the data we are trying to send has the possibility of becoming **metastable**.

- When a signal is metastable, it means that it isn't a well defined 0 or 1.

[5] circuits that share the same clock.

[6] circuits with multiple clocks

[7] Any clocks with constant phase relationships, such as $f_1/2, f_1/4$, etc.

[8] This is the problem with Clock Domain Crossing (CDC)

- Metastability arises from violating the **setup and hold times**[9] of a flip-flop.

[9] Setup time t_s refers to the amount of time the input must be stable before the clock edge arrives. Hold time t_h refers to the amount of time the input must be stable after the clock edge arrives

Crossing Clock Domains

When crossing clock domains, metastability arises, it is not possible to completely remove metastability from a design, but with the proper procedures we can reduce the chances of it happening.

- A measure of metastability can be approximated through the notion of **mean time between failures** also (**MBTF**)[10]
- We *deal* with metastability by increasing the MBTF as much as we can through the use of some **CDC techniques**.
- A metastable signal entering a flip flop will either produce a metastable output, or a well defined 0 or 1. There is no way to tell, as this is probabilistic in nature.

[10] This is an estimate due to the probabilistic nature of metastability. The below equation is provided by cadence's CDC paper

$$\text{MBTF} = \frac{\exp\left(k_1 t_{\text{meta}}\right)}{k_2 \cdot f_{\text{clk}} \cdot f_{\text{data}}}$$

Now we will learn the most common techniques to deal with CDC

- The n-flop synchronizer
- Handshake Synchronizer[11]
- Asynchronous FIFO (Queue)[12]

[11] works through a 'request' and 'acknowledge' mechanism

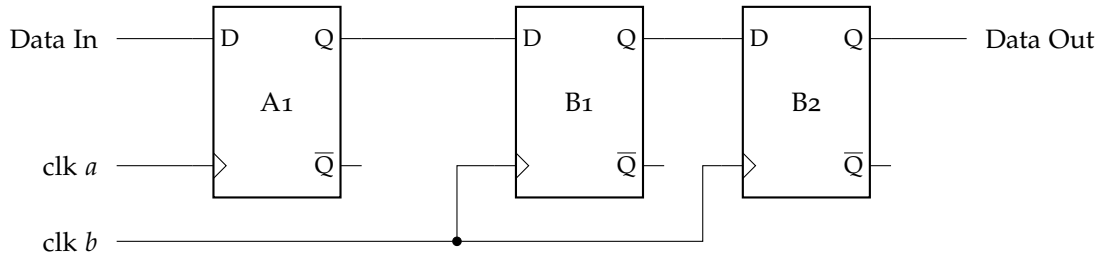
[12] tx writes into the queue, and rx reads from the queue.

These are extremely complex circuits, so we will only be learning about the first method.

n-Flop Synchronizer

The most common way to deal with CDC is to construct an ***n*-flop synchronizer**[13]. The circuit of 2-flop synchronizer looks like this.

[13] usually n is 2 or 3



It's easy to see how this circuit can be arbitrarily extended to contain n-flops. Adding a flop flop to the chain increased MBTF, but also adds latency to our design.

- This design works because even if the data becomes metastable after propagating through B1, it has another clock cycle to settle and become a well defined 0 or 1 before propagating through B2.
- Each additional flip-flop in the chain gives the (possibly) metastable signal more time to settle before being sent out.

Reset Domain Crossing

Reset Domains & Metastability

Consider a circuit with many different flip-flops. We define a **reset domain** as the set of any flip flops that share the same reset signal *rstn*.

- The reset signal guarantees that the data output is 0 during its **assertion**.
- Metastability can only occur during the **deassertion** of a reset signal.

There are a couple methods to actually fix this, and the first one is the same as its CDC counterpart

- *n*-flop synchronizer
- reset ordering
- clock gating

RDC Techniques

Once again the first method is to construct an *n*-flop synchronizer on the rx end that allows the (potentially) metastable signal the settle before propagating

through our design. We do this by tying the resets of all the synchronizer flops together.

Reset ordering is done by only ensuring some *rst*1 be fully deasserted before deasserting *rst*2. This works because even if *rst*1 is deasserted on a clock edge (causing a violation of setup/hold time) the metastable signal will not be propagated further as the rx flip flop is in the reset state, only after some time has passed (and the tx output is stable) we deassert *rst*2.

Clock gating is done in a similar fashion, We essentially gate off (cut off) the clock to the rx flop during the *rst*1 assertion so it doesnt sample the metastable signal right away, after the deassertion of *rst*1, we give it some time to settle before un gating the clock, and sampling it at the rx end.

Synthesis & Timing Closure Flow

Synthesis

Synthesis is the process of converting Verilog or any code written in an HDL to equivalent logic, described by a **netlist**. An EDA tool[14] is responsible for doing this and mapping it onto an FPGA device.

[14] such as ModelSim

- We can also provide constraints to guide this process: setup/hold time constraints, power or area constraints, etc.

Timing Closure Flow

Timing closure flow is the process of ensuring that a design meets timing requirements. Digital circuits must meet their timing requirements in order to function.

- If the propagation delay of some logic in between two flip-flops is longer than the clock period, the data will not arrive in time to be sampled at the clock pulse, rendering the circuit useless.

In order to avoid this, we employ the timing closure flow.

- We start by specifying timing constraints, such as the desired clock frequency

and setup/hold times.

- Next we test our design and see if it violates any timing constraints.
- If it does we examine the **critical paths** [15] and try to optimize them by tweaking logic or some other modifications.
- We then repeat those two steps until we meet our timing constraints while ensuring that any tweaks we made still retain functional integrity of the design.

[15] The paths that violate the timing constraints

Part II The Verilog HDL

Data Types & Signals

Values & Signal Strength

Verilog supports 4 possible values, and 8 strength levels to model the functionality of real hardware. Most data types in verilog support these 4 values:

- `0` - Logic Low
- `1` - Logic High
- `X` - Unknown
- `Z` - High Impedance

There are also 8 measures of signal strength[16] but i have never used them before. There are some things to keep in mind though:

- When dealing with two shorted signals of the *same value*, but different strengths, the highest strength level will be carried forward.
- When dealing with two *different values* of the same strength, the compiler outputs an X (unknown state)

[16] From strongest to weakest: `supply`, `strong`, `pull`, `large`, `weak`, `medium`, `small`, `highz`

Basic Data Types

This section will cover the different **data types** [17] present within the Verilog HDL.

- The two classes of data in Verilog are `net` and `reg`

A `reg` is a type that represents storage elements[18]. You should think of it as a variable (wire) that can hold a value.

- The default value of a `reg` is X (unknown).
- Some sub-types of reg are: `integer`, `real`, and `time`.

A `net` represents connections between hardware elements [19]. We cannot use them to store values, and all nets should be **continuously driven**.

- The default value of a net is Z (high impedance)
- Some sub-types of net are: `wire`, `wand`, `wor`, `tri`, `trior`, `triand`, `trireg`, etc...

[17] A data type is the 'type' of a variable, it also defines what operations can be applied to the variable

[18] A variable of type `reg` does not *always* mean it is a register!

[19] It should be thought of as a wire. Conveniently it is also most commonly declared with `wire`

Port Assignments

There are three types of **ports** in verilog: the `input`, `output` and `inout`. They are used to interact with modules in a very intuitive way. Now that we know about basic datatypes we can learn about the following rules that apply **inside a module**.

- An `input` should always be a wire (can be of reg outside the module).
- An `output` should always be net or reg (can be of net outside the module).
- An `inout` should always be of net (also must be a net outside the module).

Numbers

Numbers in verilog are pretty different from other languages. There is a syntax to declare them[20]. Here its easiest to learn through example by reading snippet 1.

[20] <size>'<base><number>

```
1 1'b1 // 1 bit of binary 1
2 8'h21 // 8 bits of hexadecimal 21 -> 0010 0001
3 12'ha35 // 12 bits of hex A35 -> 1010 0110 0101
4 12'o42xx // 12 bits of 42xx -> 100 010 XXX XXX
5 4'd1 // 4 bits of decimal 1 -> 0001
6 35 // signed number, default width is 32 bits
```

Code Snippet 1: Numbers in Verilog

Net-Data Types

Various net data types are supported for synthesis[21]

- `wire` and `tri` are equivalent - This also includes the and/or versions[22].

[21] `wire`, `wand`, `wor`, `tri`, `trior`, `triand`, `trireg`, `supply0`, `supply1` etc...

[22] This means `trior` and `wor` are equivalent, the same with `wand` and `triand`

- Types such as `wand` and `trior` aim to resolve short circuits by inserting an & or | condition to avoid unknown states (see snippet 2)

```
wand Y;
assign Y = A & B; // A and B
assign Y = C | D; // C or D
/*
wand:
output of Y = (A & B) & (C | D)

wor:
output of Y = Y = (A & B) | (C | D)
*/
```

Code Snippet 2: wand and wor

Reg-Data Types

Aside from `reg`, there exists the `integer`, `real` and `time` reg-data types. They all have their unique purposes

- `integer` is used for counting in for-loops[23]
- `real` is used to store decimal (floating point) numbers
- `time` is used to store the current simulation time

[23] finally a paradigm to regular programming

Some notes:

- When assinging a `real` to a `reg`, we round to the nearest integer value (0.5 rounds up here).
- The `time` datatype is not synthesizable (is ignored by syntehsizer), but it is are extremely useful for testbenches.

Vectors & Arrays

We can declare datatypes as **vectors** and/or **arrays**, which are exactly what they sounds like. See snippet 3 for syntax.

```
// VECTORS
reg [3:0] q1;
// A 4 bit register where index 3
     is MSb and index 0 is LSb

// ARRAYS
reg q2 [1023:0]
// 1024, 1-bit registers

// BOTH
reg [7:0] q3 [1023:0] //  1024, 8-
     bit registers (A 1K x 8 RAM)
```

Code Snippet 3: Vectors & Arrays

Parameters & Strings

Parameters cannot be used as variables, instead they are used as replacements for writing values (constants). Similar to HIGH and LOW in Arduino code. we declare them as shown in snippet 4.

Strings are stored in a reg vector of size 8*length, each char is stored as a byte, thus a string is essentially a reg vector of bytes.

Syntax for both is shown in snippet 4

```
1 // PARAMETERS
2 parameter A = 0, B = 1;
3
4 // STRINGS
5 reg [10*8:1] str;
6 str = "1234567890"
7 // we can store 10 chars in the str
      variable
```

Code Snippet 4: parameters

Operators

Like standard programming languages, Verilog provides a bunch of **operators**:

- Arithmetic, Logical, Bitwise, Equality, Relational, Reduction, Shift, Concatenation, and Conditional Operators are all offered in Verilog![24]

[24] you will *probably* not enjoy learning all of them... unless you are a maniac

Arithmetic Operators

I wonder what these do! They certainly *do not* perform arithmetic operations![25]. The operations listed below are binary operations, but we will learn about bitwise and unary operations soon

[25] Sarcasm

Operator	Operation	Type
+	Addition	Binary
-	Subtraction	Binary
*	Multiplication	Binary
/	Division	Binary
%	Modulus	Binary
**	Exponentiation	Binary

Logical Operators

These are used to evaluate boolean expressions[26]

[26] No sarcasm here

Operator	Operation	Type
!	Logical Negation	Unary
&&	Logical AND	Binary
||	Logical OR	Binary

Bitwise Operators

These operators perform a bit-by-bit operation on two operands. Sort of like an element-wise function applied to a matrix. With these in mind, we can perform a NAND using: ~&

Operator	Operation	Type
~	Bitwise NOT	Unary
&	Bitwise AND	Binary
|	Bitwise OR	Binary
^	Bitwise XOR	Binary

Equality Operators

These are an extension of the logical operators, and they only deal with equality and inequality. The difference between logical and case equalities are that case

equalities can compare with unknowns (X) and high impedance (Z) states, while the regular logical equality cannot.[27]

[27] 4'b1xxz === 4'b1xxz will return 1, if we used logical equality it would have returned X

Operator	Operation	Type
==	Logical equality	Binary
!=	Logical Inequality	Binary
===	Case Equality	Binary
!==	Case Inequality	Binary

Relational Operators

These are the >, < and >=, <= symbols that are used and behave in exactly the same ways as other programming languages[28]

[28] This means I do not have to make another table in LaTeX :)

Reduction Operators

Suppose we have a 100 bit vector! How do we perform a bitwise and on this vector? The correct answer is through **reduction operators**! Given some X = 4'b1010 We have:

- & X - equivalent to 1 & 0 & 1 & 0
- |X - equivalent to 1 | 0 | 1 | 0
- ^X - equivalent to 1 ^ 0 ^ 1 ^ 0

Shift Operators

Bit shift and arithmetic shift by n bits are done with $>> n$ and $>>> n$ symbols respectively (changing the arrows to $<<$ or $<<<$ changes the shift direction).

Concatentation Operator

We can concatenate through angle brackets. A, B concatenates A and B. 3A replicates A 3 times.

Conditional Operator

We can write if else statements easily thorough this ternary operator

$$\text{out} = \text{boolean expression ? value if true : value if false}$$

System Tasks

There are useful tasks and functions that are used to generate input and output during simulation, called **system tasks**. Their names begin with a dollar sign $.

Variable Monitoring

We can use the following system tasks as variable monitors (similar to print statements). These must be in an `always` block. They have syntax similar to printf() in C[29]

[29] $func(format_string, [variables])

- `$display()` - immediately prints values, is like println()
- `$strobe()` - prints values at the end of current time-step.
- `$write()` - same as $display but without the newline character, this is like print()
- `$monitor` - prints values at the end of current time-step if they have changed.

Simulation Control

We can also control simulations with system tasks

- `$reset` - resets the simulation back to time 0
- `$stop` - halts the simulator and puts it in interactive mode[30]
- `$finish` - exits the simulation
- `$time` - holds the current time, can be assigned to a `time` datatype or printed.

[30] the user can enter commands here

Tasks & Functions

You will probably be implementing a lot of things over and over again in verilog. Verilog provides **tasks** and **functions** to make this easier, as well as improve the maintenance of code.

Tasks are declared with `task` and `endtask`. They must be used (instead of functions) if any of the below are true regarding the procedure:

- There are delay, timing or event control constructs within the procedure
- The procedure has zero or more than one output arguments
- The procedure has no input arguments

```
1 task [task_name];
2     input [ports];
3     output [ports];
4     begin
5         // code procedure here
6     end
7 endtask
```

Code Snippet 5: Task Declaration Syntax