Verilog Coding Style and Synthesizable Verilog

Overview

In this lesson we will

- ✓ Introduce the embedded development cycle
- ✓ Compare and contrast traditional development cycle with current needs
- ✓ Introduce modeling approach to design
- ✓ Review and motivate Verilog language
- ✓ Examine hardware modeling and synthesis using Verilog.
- ✓ Review Verilog design flow.
- ✓ Present some good Verilog coding style and practices.
- ✓ Examine some of the non-synthesizable Verilog constructs.

Introduction

Increasing complexities of contemporary systems
- Demand the use of increasingly powerful tools
- Pencil and paper methods
  - No longer reasonable in large sense

Tools today increasingly computer based
- Collection and practice
  - Called *electronic design automation* - EDA

Today hardware portion of design
- Follows design flow similar to software development
  - Many of the same methodologies apply
- Designs developed using *hardware design languages* – HDLs
- Managed same way as software developments
- Following a formal and disciplined development process
  - Can lead high quality reliable safe products at its end

Major focus in ensuing discussions will be on hardware side of the development cycle
- None-the-less software plays major role in development of today's embedded systems

Will use Verilog HDL as major tool in developing then synthesizing hardware models
- Should be quite familiar with basic Verilog and structure of Verilog program

Over next several lessons will take study of language and modeling to next level

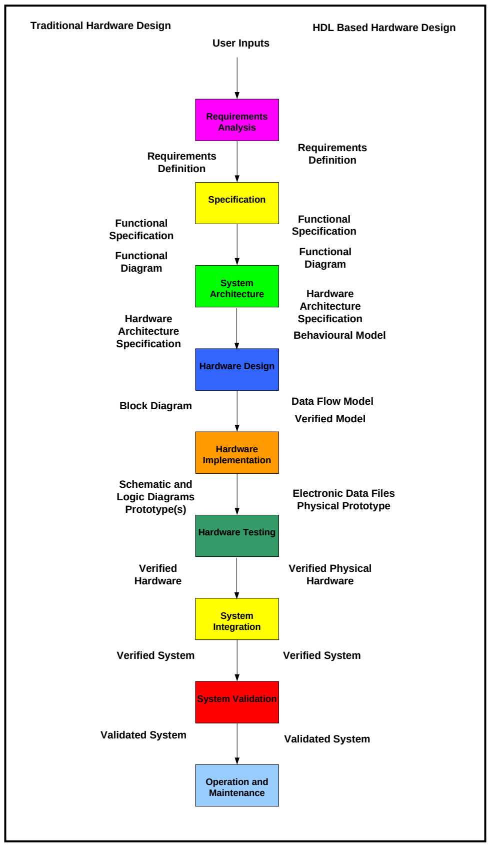
- Will begin with review of purpose of Verilog
- Introduce embedded development cycle
- Examine some good design and coding practices
- Examine how to utilize different modeling levels
- Incorporate real-world effects into Verilog models
- Then identify some Verilog constructs that cannot be synthesized into hardware
- How to model real-world effects

Important considerations

- ✓ End goal of Verilog program
    - A solid robust reliable system
- ✓ Verilog is a hardware design language not a software programming language
- ✓ Many of tools and techniques appropriate to good software development
    - Also appropriate to developing good Verilog designs
- ✓ Modeling design using Verilog is intermediate step
- ✓ End goal of Verilog program is synthesis into hardware

Birefly examine traditional approach to hardware design

Traditional Hardware Design



Traditional approach

- ✓ Identify requirements and formulate specification
- ✓ Functional decomposition
- ✓ Formulate architecture
- ✓ Map modules to architecture
- ✓ Design comprising modules
    - At gate level
    - Draw logic diagram or schematic
- ✓ Build modules
    - Test modules to verify functionality
    - Make necessary modifications
        - Based upon testing
- ✓ Integrate modules into subsystems
    - Test modules to verify functionality
    - Make necessary modifications
        - Based upon testing

- ✓ Integrate modules into systems
    - Test modules to verify functionality
    - Make necessary modifications
        - Based upon testing
- ✓ Formulate and confirm
    - Timing and operational requirements
    - Other constraints

For contemporary designs
- Serial path of traditional approach no longer feasible
- Driven by
    - Complexity
    - Physical constraints
        - Circuit will operate differently
            - When spread out on bench
            - Reduced to IC or PLD
- Most contemporary designs
    - Mix of hardware and software
- Don't have luxury of approaching in serial manner
    - First hardware then software

Models and Modeling

Based upon limitations of traditional approach to hardware design
- Need to consider alternate approaches

Today modeling and HDLs are an essential part of design

Let's look briefly at
- Motivation for modeling
- What we are modeling
- Essential characteristics for modeling method

Why are We Modeling?

- Primarily we use models to represent a description of
    - Real system or one that will become real
        - When it is designed
- Models give us different views of our system
    - External, internal, abstract, behavioural, structural…
- Model gives us means to describe characteristics of system to be designed
    - Provides basis for later verification

- Models are cheaper than building complete system
  - To test design concept
- Models allow us to execute test that may be too hazardous to run
  - During preliminary development

In design process
- Model precedes actual design
- Provides opportunity to quickly explore variety of alternative approaches
  - Cheaply
  - Quickly

What are We Modeling?

To effectively formulate a good model
- Must understand what we are modeling

Our target is embedded applications

We know that embedded systems are
- Reactive
  - System runs continuously
  - Responds or reacts to signals from external environment
- Often real-time
  - Time constraints imposed on behaviour
- Heterogeneous
  - Composed of hardware and software pieces
    - Hardware can be PLD, ASIC, custom IC, microprocessor, combination
- Supported by different development environments

We need to distinguish
- ✓ Model
- ✓ Language used to express the model
  - Can very easily develop models using
    - C, C++, Java
    - Matlab, PSPICE
    - Etc.
- ✓ Final hardware implementation

Qualifying the Model

Restating – the model expresses an abstraction of the real world
- Intended to give an abstract representation
  - Portion of real world
- Allows us to temporarily ignore certain details
  - As we gain understanding of problem

To be useful
- We can hypothesize some essential general capabilities
  - Abstraction
    Must allow us to express and examine behaviour
    - Of complete system
      - Unburdened by details of sub-components
  - Refinement
    Must allow us to express and decompose behaviour of system
    - At different levels of granularity
  - Structure
    Must be able to express system as set of interconnected modules
  - Communication
    Must support inter module communication method
  - Should support synchronization method
  - Easy to interpret
    Must express anticipated behaviour or aspect being modeled
    - In comprehensible format

Two classifications of model are particularly useful
- Conceptual and analytic

Conceptual

Precedes analytic

Allow us to work at high level of abstraction
- Uses a symbolic means
  - To capture qualitative aspects of problem

Useful during early stages of design
- ✓ Formulating specification
- ✓ High-level architecture
- ✓ Early stages of partitioning the system
- ✓ Allow us to grasp and work with complexities of a design
  - To focus on essential details while ignoring others

Are behavioural in nature

Analytic

Permits analysis at lower levels of detail

Use mathematical or logical relations

Express quantitative physical behaviour

Useful during middle and later stages of design

- ✓ Later stages of partitioning
- ✓ Modeling and analyzing detailed architectures
- ✓ Verifying detailed performances
- ✓ Making performance trade-offs

Are more structural in nature

Important Characteristics of Models

To be effective

Models should give us ability to express

1. Modularity and hierarchy

   Should be able to express

   Static and dynamic behaviour

   Structural and functional construction
2. Relationships among subsystems

   Should be able to express

   Sequential and concurrent flow of control

   Inter subsystem synchronization and communication

   Temporal behaviour
3. Communication amongst tools
4. Use of legacy designs or behaviours
5. Affects of real-world physics on circuit and signal behaviour
6. Ideally models should be executable

Later this is how we verify the system throughout design process

Major focus of discussions will be on hardware side of the cycle

None-the-less software plays major role in development of today's embedded systems

Will use Verilog HDL as major tool in developing then synthesizing hardware models

Should be quite familiar with basic Verilog and structure of Verilog program

What is Verilog?

Verilog is a hardware description language - HDL
- Provides a means of specifying a digital system
  - At a wide range of levels of abstraction

Language supports
- Early conceptual stages of design
  - With its *behavioral* level of abstraction

- Later implementation stages
  - Data and control flow with *dataflow* level of abstraction

- Detailed device level model
  - With its structural level of abstraction

Language provides hierarchical constructs
- Allows the designer to control the complexity of a description

- Note: this description is an excerpt from the book
  - Verilog Hardware Description Language, by Thomas and Moorby.

Typical introduction to Verilog focuses on modeling ability
- Generally little emphasis on mapping of model into physical hardware
  - Let's examine the difference

Why use Verilog?

Why use Verilog indeed
- Why use any modeling language for that matter

As we know circuits and systems we are developing today
- Growing in capability and complexity every day

As noted: yesterday a sketch on a piece of paper and a handful of parts
- Sufficient to try out a design idea

Today that is no longer possible
- Why not viable approach today
  - Speed
  - Parasitics
  - What else

Idea is

- Modeled using computer based tools and languages
- Synthesized into the desired hardware implementation
- Test and verify the design

We use two key words here *model* and *synthesize*

  While test is important

    It applies no matter what approach is used

Looking at two components of development process

- ➢ Objectives of Verilog HDL *modeling*
  Capture and verify behaviour of design prior to committing to physical hardware
  Process entails
  - ✓ Mapping the system requirements into design that meets those requirements
    Sentence makes very strong and important statement
  - ✓ Understanding real-world physical constraints and limitations of
    Modeled parts and environment
  - ✓ Incorporating effects of identified constraints and limitations into model
    Includes variations on such constraints and limitations
  - ✓ Verifying that modeled design meets or exceeds specified requirements
    Subject to real-world physical constraints and limitations

- ➢ Objectives of Verilog HDL *synthesis*
  Map modeled design to physical hardware and verify behaviour of design
  Process entails
  - ✓ Removing all modeled constraints and limitations from model
    Real-world constraints and limitations already exist in target environment
  - ✓ Mapping design to physical hardware as programmable logic device or IC
  - ✓ Verifying that design meets or exceeds specified requirements
    Subject to actual real-world physical constraints and limitations
  - ✓ Revisiting modeling process as necessary if design does not meet specs

A number of languages that support such a design approach
- Verilog and VHDL
  - Two of the more common
- SystemC
  - For modeling both the hardware and software components
  - Finding its way into an increasing number of designs in the embedded world

Working with the Verilog HDL – HDL Based Design

Design executed using HDL
- Must still meet many of same initial requirements as traditional approaches
- Whether using
  - Pencil and paper or computer based tools
    - Must still work from sound and solid
      - Set of requirements
      - Specification

Today as we progress through formal design process we

- Identify all system requirements and specifications
- Decompose required system into major functional blocks
- Identify data and control flow between and among blocks
- Formulate a system architecture
- Model the design and target environment
  Compile design into working HDL implementation in modeled environment
  Typical implementation
  - Behavioural or dataflow model
- Test
  - First coarse grained functionality and behaviour
  - Then fine grained details
  - Then stress design subject to environmental constraints and limitations
  - Iterate
    - Until we are satisfied
    - This is engineering
  - Synthesize the model
    - Map modeled HDL design
    - Into switch level real-world hardware implementation
      - Target may be
        - Programmable logic device

FPGA, CPLD, Memory

Integrated circuit

ASIC, full custom design

Some other combination of digital hardware

- Test

  First coarse grained functionality

  Then fine grained details

- Iterate

  Until we are satisfied

  Once again this is engineering

Models and Verilog HDL – A First Look

We'll now look at how Verilog

Supports each of the desired characteristics of models

Levels of Abstraction

We noted when executing a design

Must be able to work a several different levels of abstraction

Ultimately hardware design will be implemented

At transistor level

Complexity of today's systems

Make detailed design at transistor level difficult at best

Today generally work at higher levels

Verilog HDL supports design at several levels

Switch level

Gate level

Dataflow

Behavioural

Mix of types

Switch Level

Switch level model

Views system to be designed

At the transistor level

Can be viewed as analogous to working

At machine code level

Modeling at switch level

- Provides full control over design
- Not reasonable for large complex systems

Structural – Gate Level

- Structural or gate level model
  - Views system to be designed
    - As collection or components or gates
  - Gates
    - Typically gate level primitives
      - AND, OR, NOT
  - Components
    - RTL level devices

- Approach intuitive for those with
  - Basic knowledge of digital design
- There is a one to one correspondence
  - Between logic diagram and Verilog code
- All signal-assignment
  - Concurrent

- Modeling at gate level
  - Can be viewed as analogous to working
    - At assembly code level
  - Relinquishes some control over design
    - Enables more rapid development
      - Than switch level modeling
  - Not reasonable for large complex systems
  - Sometimes necessary
    - For specific parts of the design
      - That must be optimized to meet
        - Size, speed, or architectural constraints

Data-Flow – RTL

- Data-flow or Register transfer level model
  - Describes how system's signals
    - Flow from inputs to outputs
  - Design usually written as
    - Boolean function on inputs to produce outputs
    - That is describes input and output operations
      - In terms of dataflow operations

On signals and register values
If registers involved
Operations or transfers
May be synchronous

RTL descriptions use language operators
To determine values of signals in circuit
They characterize flow of data through
Computational units
Data-flow descriptions
Simulate design by showing how signals flow
From inputs to outputs
Data-flow signal-assignment statements are concurrent
Execution controlled by events

Combinational logic descriptions
Use *continuous assignment*
To create implicit structural model of such logic

Assignment
Declares Boolean relationship
Between operands on left and right hand sides

Under simulation
Simulator monitors operands
Determines if and when to schedule activity
On target variable
All signal assignment statements
That have an event
Executed concurrently

Modeling at data-flow level
Can be viewed as analogous to working
At C code level
Relinquishes more control over design than gate level
Enables more rapid development
Than switch or gate level modeling
Reasonable for large complex systems
Sometimes maybe necessary to handcraft certain modules
For specific parts of the design

                That must be optimized to meet
                    Size, speed, or architectural constraints

Behavioural – Algorithmic

- Increasing design complexity
    - Demands important design decisions
        - Made early in project
    - Need to be able to evaluate and trade-off alternative
        - Architectures
        - Algorithms
    - Before deciding on optimum implementation scheme

- Such decisions
    - Made at algorithmic level
        - Rather than in terms of gates

- Focus is on behavior rather than performance
- Following high-level analysis
    - Implementation and optimization
        - Of selected approach executed

- Behavioural or algorithmic model
    - Describes system by showing
        - How outputs behave based upon changes on inputs
    - Describes input and output operations
        - In terms that may not conform to a dataflow
            - May not be synthesizable
    - Do not need to know underlying
        - Detailed logic implementation
    - Must know how output behaves
        - In response to changes in inputs

- Design expressed as collection of procedural statements
    - Determine or specify
        - Relationship between input and output signals of module
        - Without reference to hardware or structure

- Algorithmic description of behavior
  - Assigns value to register storage variable
    - By executing procedural statement
      - Utilizing operators and flow of control constructs
  - All statements within body of module
    - Executed sequentially
  - Those within *always* block
    - Treated as concurrently

- Implemented behavior
  - May be combinational or sequential

- Modeling at behavioural level
  - Can be viewed as analogous to working
    - At C++, Java, or C# code levels
  - Relinquishes more control over design than data-flow level
    - Enables more rapid development
      - Than other modeling levels
  - Reasonable for large complex systems
    - Real strength is modeling
      - During early part of design
        - Details not necessarily know
        - Functionality must be verified

  - When used for full system design
    - Sometimes maybe necessary to handcraft certain modules
      - For specific parts of the design
        - That must be optimized to meet
          - Size, speed, or architectural constraints

Design by Composition

- Verilog HDL supported by
  - Library of built in primitives

- Language is extensible
  - Permitting user defined types

- New modules
  - Can be created by process called *composition*
  - From
    - Primitives
    - Other user defined types
  - Thus architecture of composite module
    - Can be flat
    - Hierarchical

- Composing modules can be
  - Gate level, dataflow, or behavioural implementations
  - Contained modules
    - Can be of any type of implementation
    - Do not all have to be of the same type

- User defined module name
  - Becomes type specifier
- The ability to incorporate
  - Primitives
  - Other user defined modules
- Naturally implies support for legacy models/modules

- Put in different terms
  - Verilog does not preclude reuse
  - However to facilitate reuse
    - Component needs to
      - Have well-defined public interface
      - Be well-defined
      - Be properly modularized
      - Ideally conform to some interchange standard

Inter Module Relationships

- Supported by design
- As with support for legacy modules
  - Inter-module relationships not precluded

Looking at a Verilog Source File

Case-sensitivity

- Verilog is case sensitive
  - When you encounter an error while compiling a Verilog source file
    - Look for case-errors

Modules

General format

- Verilog utilizes the concept of modules
  - Think of a module as a "black box"

- Module in Verilog
  - Describes
    - Functionality or structure of
      - Entire design
      - Piece of a design
    - Ports through which it communicates
      - With
        - Outside world
        - Other modules
  - Implements a software encapsulation of
    - Structure
    - Behaviour
    - Other properties
  - Of a piece of functionality
  - Names
    - Port signals for the module

- HDLs support higher level of abstraction
  - Than can be described using
    - Schematic or logic diagram

- Like object centered languages
  - New proprietary modules can be defined
    - Using process of composition
      - Primitives
    - Other proprietary modules

To make a system consisting of modules
- We link up the individual black box with wires
- The concept of module permits the building of complex systems
  - By linking lower-level designs
  - Provides a structure for the design process

The more modules we have
- The more complicated the design becomes
- In such cases, it is convenient to be able to verify functionality
  - Module by module

  Here is a sample module declared in Verilog

```
module module_name[(outputs_list, inputs_list)];
    outputs              //  outputs from the module
    inputs               //  inputs to the module

    reg                  //  local storage in the module
    wire                 //  conduction paths in the module

    code                 //  your code
    ...
    ...
endmodule
```

Signal-Assignment

Signal-assignment used to assign value to signal
- Left hand side of statement
  - Declared as signal
- Right hand side of statement
  - Permitted to be
    - Signal, variable, constant

Execution

Execution of signal-assignment
- Comprises two phases
  - Calculation
  - Assignment

For execution to start
Event on right hand side of statement must occur
In no event
Statement is inactive
If event or multiple events occur - regardless of order
All statements executed simultaneously

Calculation

Value of right hand sides *calculated*
Using current values of all operands
Comprising the expression
Represents value of signal at time T_i

Assignment

Calculated values *assigned* to left hand side signal
After some delay Δ
If delay specified as 0
Value of Δ infinitesimally small

Execution of signal assignment may be

- Concurrent
  All appropriate assignments within module made
  At same time
- Sequential
  Calculation phase
  Does not wait until preceding statement assigned
  Commences after calculation complete

Timing

- Using Hardware based approach
  Can build prototype
  Timing and time delays
  Restricted to those associated with parts
  Used to build the specific prototype

  Can build number of copies
  Using different parts
  Different vendors or date codes

- Using HDL based approach
  Can model
    Range of delay values
    Different types of delays
  Association between
    Time unit and physical time
      Can be set as compiler directive

Testing

Testing design under HDL
  Has same motivation and goals as in hardware based approach
Under either approach
  Testing occurs at several different levels
  - High level – functional
    Looking at the overall behaviour of system
    Does it satisfy the functional specification
    Typically an external view
  - Verification – detailed test
    Does design meet
      Boundary conditions
      Nominal operation
      Cases beyond boundary
    External view
  - Detailed timing and signal flow
    Internal view

Under
  Hardware based approach
    Test vectors or stimuli and monitoring
      Coming from physical instruments
  HDL based approach
    Test vectors or stimuli and monitoring
Implemented utilizing sw routines

Let's now look at some points that help to ensure quality Verilog model

Coding Style

- ✓ Begin with some key high-level points
- ✓ Move to examining good coding practices
- ✓ Defining and examining synthesizable Verilog

High Level Points

Fundamental points

Make sure that your code is

Readable

Easy to modify

Reusable

Well documented

Good coding style helps to achieve better results

- ✓ Modeling
- ✓ Simulation
- ✓ Synthesis

Not all Verilog constructs can be synthesized

Only a subset can be synthesized

Code containing only this subset can be synthesized

Good Coding Practices

Naming Restrictions

Identifiers

Give an object a name to allow later reference

Identifier may contain

- Alphabetic characters
- Numeric characters
- Underscore
- Dollar sign

Alphanumeric name followed by '-numeric' sometimes not allowed

myName-0

Must begin with alphabetic character or underscore

Can be up to 1024 characters in length

Naming Conventions and Styles

Several common formats for writing identifier name

- addressBuss
- address-buss
- address_buss

All are legal – choose one and stay with it

Don’t mix formats within a program

Searching common operation when designing or debugging program

Searching for identifiers such as

- i1, i2, i3 in large program
- ModuleThatComputesTheSumofTwoNumbersandOutputsanInteger

Challenging at best

Use meaningful names

i1, i2, i3 valid identifier names but meaningless – convey no information

Want code to be self-documenting

Conventions

Use uppercase letters for all

Constants

Use leading uppercase

User defined modules

Use leading lowercase letters for all

- ✓ Signal names
- ✓ Port names
- ✓ Device names

Convey active state of signal in identifier name

- Active low

  nReset

  reset_n

- Active high

  reset

Comments

Two forms of comment

- `//` single line comment
  - Can appear as starting character on each line of block of commented text
- `/* */` multiple line comment
  - Can be used to mark single line comment

Comments should be

Meaningful informative suggestive of their intended purpose

Don't state the obvious

Vertically align left hand sides of all comments

Example

```
Bad
    parameter halfPeriod = 100;      //  set halfPeriod to 100
Good
    parameter halfPeriod = 100;      //  set halfPeriod to minimum legal value

Bad
    parameter fullPeriod = 200;      //  set fullPeriod maximum legal value
    parameter halfPeriod = 100; //  set halfPeriod to minimum legal value
Good
    parameter fullPeriod = 200;      //  set fullPeriod maximum legal value
    parameter halfPeriod = 100;      //  set halfPeriod to minimum legal value
```

Formatting Conventions and Styles

The following are recommended formatting preferences

Goal is to enhance readability of code

- Preferred - place each part of begin-end pairs on a line by itself
- Vertically left align begin and end
- Indent and align the body of compound statements from the opening and closing delimiters
- Declare each variable on a separate line (with a trailing comment)
- Place a blank line before a declaration that follows executable code.
- Place spaces on either side of a binary operator
- No more than one statement per line
- Maximum line length of 100 characters

Declarations, Definitions, and Modules….

Constants

Declare all parameter constants at the top of the module written in all upper case letters.

Example

```
parameter HALFPERIOD = 100;
```

Modules

The first module should be the test bench or the top-level module
Followed by the remaining modules

Each module including the top-level
Should have a header listing
Name
Inputs
Outputs
Description
Author(s)
Date written
Date and description of each revision

Block Comments at Start of Files

```
//-----------------------------------------------------------
// File name:
//     MyFile
//
// Description:
//    Implements high-speed SerialIO system.
//       Provides coms link between data collection system and
//       remote peripheral devices.
//       Data stream sent with Manchester Phase Encoded
Clock
//
// Author:
//    Iman Engineer
//
```

Block Comments at the Start of modules

```
//-------------------------------------------------------------
// Module name:
//     MyModule
//
// Description:
//    Module implemented as part of high-speed SerialIO system.
//       Counts number of characters transmitted
//       Computes running parity
//       Transmits parity and EOM
//
// Author:
//    Youran Engineer
//
//-------------------------------------------------------------
```

Block Statements

Coding convention for all block statements shall be either of the following:

```
if( expression ) begin
          statement1;
          statement2;

          if( expression ) begin
               statement1a;
               statement 2a;
        end

     end

     else begin
          statement3;
          statement4;
     end
```

or

Choose one style on the other – don’t mix styles

Note:
Indentation of statements
Relative to "if" and "else" for each if-else statement

```
if( expression )
    begin
        statement1;
        statement2;

        if( expression )
        begin
            statement1a;
            statement 2a;
        end
    end

else
    begin
        statement3;
        statement4;
    end
```

Synthesizing the Design

Introduce concept of synthesizable Verilog
Term often dependent upon tool used to synthesize Verilog code into hardware

In current context
Synthesizing design means mapping design onto physical hardware
Subject to vagaries of real-world

That said - important to note
Richard Stallman and colleagues at Free Software Foundation
Have and continue to develop excellent *compilers* for software languages
Essentially their target environment is benign

With Verilog

- *Compile* code
  Into target form for modeling, simulation, and testing
- *Synthesize* code
  Into basic logical devices in hardware

Issues

- ✓ Function or operation of software *compiled* into target code
  - Not dependent upon fundamental physics of real world

- ✓ Function of operation of modeled design *synthesized* into target code
  - Totally dependent upon fundamental physics of real world

In the current context term synthesizable Verilog means
- Code can be synthesized into hardware
- Target hardware performs intended function
- Target hardware optimized
  - Size
  - Execution speed

Ensuing discussion examines
- Some good design practices
- What and what not typically possible to synthesize into hardware

- Does not address
  - Problem of optimization of synthesized hardware
  - Fundamental design details

Clocks and Reset Signals

- Utilize synchronous logic – specify clocked behaviour rather than strobed
  - Master resets can be an exception
- Don't mix clock edges
  - Rising edge for some devices falling edge for others
  - Use multiphased clocks if necessary to move in time
- Don't put combinational logic on clock or reset signals
  - Differing prop delays for signals coming into combinational logic block
    - Can create races leading to hazards

Storage Devices and FSM Management

Storage Devices

Three basic types of storage devices
- Latches
- Gated latches
- Flip-flips

Gated latches and flip-flops
- Control behaviour either by gate or clock signal

(Inferred) Latches have no control
- These should be avoided

Example

```
always @ (s0)
begin
    case (s0)
        2'b00:     out0 =1'b1;
        2'b01:     out0 =1'b0;
        default :  out0 =1'b0;
    endcase
end
```

Sequential blocks should use *non-blocking* statements

Power Up and Clock Edges

Consider carefully how system behaves during and after power-up

- As power is coming up all signals take on random values
- Following power-up storage devices are in random state

System should utilize power-on or master reset

✓ Holds system in reset state during power-up
✓ Establishes known initial state following power-up

Consider what happens before and after each clock edge

If setup or hold times are not met

- Possible metastable behaviour
- Incorrect data stored
- Data lost

State Encoding

Number of alternatives for encoding state variables for a FSM
Choice depends upon engineer and the design

- Binary or Sequential (minimal) encoding

  Straight gets states encoded

  S0 = 000

  S1 = 001 etc.

- Gray encoding

  Single state variable change between states

  ✓ If done properly eliminates decoding hazards on outputs based upon combinations of state variables
  ✓ Reduces state transition errors caused by asynchronous inputs changing during flip-flop set-up times
  ✓ Minimizes power consumed in state vector flip-flops

S0 = 000
S1 = 001
S2 = 011 etc.

- One-hot encoding assigns one flip-flop per state
  Single flip-flop per state variable
    - ✓ Eliminates decoding on outputs based upon state variables
    - ✓ Uses large number of flip-flops
- M of N encoding
  Variation on one-hot encoding
    - ✓ State variable values mapped to output signals – do decoding
    - ✓ If done properly eliminates decoding hazards on outputs
- Custom: assignment of state variable values

Synchronous and Asynchronous Resets

Understand

- ✓ Difference between synchronous and asynchronous resets
- ✓ Flow of control

Synchronous (with system clock) reset

```
always@(posedge clock)
begin
    //  reset code will be executed first if reset asserted
    //  synch with system clock
    if (reset)
    begin
        code under reset
    end

    without else block statement
    remaining code will be executed following possible reset
end
```

Asynchronous (to system clock) reset

```
always@(posedge clock or negedge reset)
begin
    //  reset code will be executed first if !reset asserted
    //  asyn to system clock
    if (!reset)
    begin
        code under reset
    end

    else
    begin
        with else block statement
        remaining code will be not be executed following possible reset
    end
end
```

Verilog Primitives

Not all kinds of Verilog constructs can be synthesized
Only a subset of Verilog constructs
Can be synthesized
Code containing only this subset is synthesizable

✓ Synthesizable primitives

- buf, not, and, or, nand, nor, xor, xnor
- bufif0, bufif1, notif0, notif1

Continuous assignment must be used for tri-state devices

✓ Nonsynthesizable primitives

===	!==	
/ (division)	% (modulus)	
#delay	initial	
repeat	forever	wait
fork	join	event
$display	$monitor	

Variable Ranges

Simple code can generate large amounts of circuitry

```
integer a,b,c;
always @(a or b) c = a * b;
```

Will compile but take significant time to synthesize

integer type implies 32 bits

Within the module specify or limit range of values

```
input [2:0] a;
input [2:0] b;
output [2:0] c
```

Operator Precedence

Understand operator precedence

Highest precedence	Concatenation ({ }, {{}})
	Unary reduction (!, ~, &, |, ^)
	2's complement arithmetic (+, -, *)
	Logic shift (>>, <<)
	Relational (>, <, >=, <=)
	Equality (==, !=)
	Binary bit-wise (&, |, ^, ~^)
	Logical (&&, ||)
Lowest precedence	Conditional (?:)

Summary

In this lesson we have

- ✓ Introduced modeling approach to design
- ✓ Reviewed and motivated Verilog language and modeling with Verilog.
- ✓ Examine hardware modeling and synthesis using Verilog.
- ✓ Reviewed Verilog design flow.
- ✓ Presented some good Verilog coding style and practices.
- ✓ Examined some of the non-synthesizable Verilog constructs.
- ✓ Introduce the embedded development cycle
- ✓ Compare and contrast traditional development cycle with current needs