The Verilog Hardware Description Language - A Structural View

Overview

In this lesson we will

✓ Introduce Verilog Modeling and the Hardware Design Language.
✓ Introduce some of the basics of the Verilog language and structure of a Verilog program.
✓ Begin to explore some of the Verilog terminology and primitives.
✓ Introduce concepts of Verilog variables, nets, and multibit signals.
✓ Introduce Verilog numbers and number specifications.
✓ Introduce some of the data types that are part of the Verilog language.
    These are known as the intrinsic (or built-in) types.
✓ Study and develop Verilog structural level models of combinational and sequential circuits.
✓ Introduce real-world affects into the models.

Introduction

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

Following up on these words
Today as we progress through formal design process we
• Model the design
    Transform requirements into working HDL implementation
• Synthesize the model
    Transform modeled functionality
    Into switch level hardware implementation
• Test
  First coarse grained functionality
  Then fine grained details
• Iterate
  Until we are satisfied
• Then transform switch level implementation into
  Target is programmable logic device
  ASIC, FPGA, CPLD, Memory
  Some other combination of digital hardware
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

Verilog is a hardware design language
  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
  Mid level modeling of data and control flow
    With dataflow or Register Transfer (RTL) level modeling
Later implementation stages
  With its structural level of abstraction
Ultimately
  switch level for implementation

Language provides hierarchical constructs
  Allows the designer manage the complexity of contemporary designs
We will

Begin with some useful information on
Several different Verilog data types
Follow with quick review of
Basic components and organization of a Verilog program
Review gate-level or *structural* modeling
Combinational logic circuits
Sequential circuits
Introduce *dataflow* and *behavioral* models
Examine some important tools and capabilities of language
Facilitate fine grained modeling and test of a design

Emphasizing last point important to recognize
Design is only one aspect of the product development
Each design must also be tested
Confirm that it meets specified requirements and the objectives
Of the modeling process
To do so must have a specification
Level of formality varies
To that end will also discuss how one can formulate test suites
To verify the intended operation.
Material on testing will lay the foundation
To enable developer to build test cases that will support testing to desired level

**Variables and Nets**

Verilog language defines several different kinds of *variables* and *nets*
Variables and nets are different
These used to
Hold logical signals
Interconnect various components or modules – functional blocks
That make up a Verilog program
Ultimately the system being modeled
Variable
Verilog variable like a variable in C, C++, or Java
Can be assigned a value – they can be an lvalue
Can appear on left hand side of an assignment operator
Will hold that value until a subsequent assignment replaces the value

Net
Net represents a class of primitive data types
Used to model a node or electrical connection in a circuit
Net is an rvalue
Cannot
Be assigned to hold a value
Appear on left hand side of assignment operator
Value results from being continuously driven by output of a logical device
Important in dataflow model
If a net is not driven
Takes on the default value of ‘z’
Meaning high impedance or floating.

Wire
A wire type is a kind of net
Abstraction of real world wires which are used to
Connect output of one logic element to
Input(s) of other logical elements
Real-world wires can be si, cu, fiber, wireless connections
Because it is a net
Value of a wire can only be changed
As result of a gate or a behavioral or dataflow statement driving it

Reg
A reg is a kind of variable
Value of a reg or register
Can be changed directly by an assignment
One should not confuse the Verilog reg with the hardware register
The reg is simply an entity that can hold a value
Default value of a reg data type is ‘x’, or unknown
Like real-world storage devices
The syntax for the reg and wire declarations is given as

Syntax
```verilog
reg regList;
wire wireList;
```
Declaring Multi-Bit Signals

Often necessary to represent multi-bit wires

Formally such sets called vectors

A 3-bit wire that can carry digital signals representing the values 0..7

Called a 3 bit vector

Types reg and wire can also be formed into a bus such as

```
Syntax
Big Endian
reg [msb:lsb] reg_list
wire [msb:lsb] wire_list;

Little Endian
reg [lsb:msb] reg_list
wire [lsb:msb] wire_list;
```

msb is the bit index of the most significant bit

lsb is the bit index of the least significant bit

The value of the lsb index must be zero

Since bit position 0 conventionally denotes the least-significant bit

Such statements configure a set of individual wires

So that they can now be treated as a group

```
wire [2:0] myWires;   // a 3-bit signal (a bus)
reg [15:0] aState;    // a 16-bit state holding value
```

The declaration myWires

Declares a 3-bit signal that has

- MSB (the 2²’s place) as myWires[2]
- Middle bit of myWires[1].
- LSB (the 2⁰’s place) as myWires[0]

The individual signals can be used

Just like any other binary value in Verilog

```
and a1(myWires[2], myWires[0], C);
```
Statement
AND’s together \( C \) and the LSB of \( myWires \)
Puts the result in the MSB of \( myWires \)

✓ Note again
We are not assigning conjunction to \( myWires[2] \)
The gate \( a1 \) is driving that signal
Only way \( myWires[2] \) can change
If output of gate changes because input changed

This bus specification
Can be extended to input and output lists as well
Multi-bit signals can be passed together to a module

```
module random(bus1, bus2);
    output [31:0] bus1;
    input [19:0]   bus2;
    wire c;
    anotherRandom ar1(C, bus2, bus1);
endmodule
```

Subsets of Multi-Bit Expressions
On occasion it’s necessary to break apart multi-bit values
Can do that by selecting a subset of a value

```
reg [31:0] myReg;
initial myReg[3:1] = 'b101;
```

This would set
\( myReg[3] = 1 \)
\( myReg[2] = 0 \)
\( myReg[1] = 1 \)
All other bits of \( myReg \) will not be altered
One can also use the same form to take a subset of a multi-bit wire
Pass it as an input to another module

wire[31:0] myWires;
output myWires[3:1];

Numbers
Verilog supports two types of number specification
Sized
Unsized

Sized Numbers
Sized numbers declaration comprises
Size, base, value

Syntax
size 'base value
size specifies number of bits in number
base identifies the base
legal bases: 'd or 'D – decimal
'o or O – octal
'h or 'H - hexadecimal
value numeric value in specified base

Examples
4'b1010 // a 4 bit binary number
8'd35 // an 8 bit (2 digit) decimal number
16'hface // a 16 bit (4 digit) hex number

Unsized Numbers
Unsized numbers
Without a base specification
Decimal by default
Without a size specification
Have simulator/machine default number of bits
Must be at least 32

Examples
1010 // a 32 bit decimal number by default
'o35 // a 32 bit octal number
'hface // a 32 bit hex number
Unknown or High Impedance Values

Verilog supports numeric specification
For numbers with unknown or high impedance bits/digits
Symbols used for specification
x – unknown value
z – high impedance value

Examples

4'b101x // a bit binary number with lsb unknown
8’dz5 // an 8 bit (2 digit) decimal number
       // with high impedance ms digit
16’hfzxe // a 16 bit (4 digit) hex number with high impedance and
       // unknown digits

The Verilog Models

Will now examine models and modeling tools
At each of the levels

Three Models – The Gate-Level, the Dataflow, and the Behavioral

Verilog language supports the development of models
At three different primary levels of abstraction

Gate level model
Gives most detailed expression

Behavioral level
Gives most abstract

Gate level
Modules are implemented by interconnecting the various logic gates
Similar to working with SSI and MSI components
Also known as a structural model

Dataflow level
Module is implemented by specifying the movement of the data
Amongst the comprising hardware registers
Model is analogous to the RTL (Register Transfer Level) level
Used in specifying a microprocessor / computer architecture
Behavioral level

Modeling is based upon an algorithmic description of the problem
Without regard for the underlying hardware.

Language does support modeling at the transistor level
Work at that level will not be discussed here

Model Development

Will begin at the gate level and work up
Path will be to use the three different levels
To introduce the core aspects of the language
Because working at the gate level is the most familiar
Will begin / review at that level then move up to higher levels of abstraction
As we do so we will also introduce several aspects of the language
Apply to all levels of abstraction
Will utilize the same combinational and sequential designs
To illustrate how a model is developed at each of the different levels

Combinational circuits
Will use an AND and an OR gate
Extended to implement a NAND and a NOR circuit

Sequential circuits
Will progress from
Basic latch
Gated latch
Flip-flop
Two bit binary counter

Structural / Gate Level Development

At the gate level
Working with the basic logic gates and flip-flops
Typically found in any detailed digital logic diagram
Devices model the behavior of the parts
We can buy from any electronics store
Might design into an ASIC or use in FPGA

Verilog supports logic gate primitives identified in adjacent figure
As predefined intrinsic modules

Prototypes for each of the gates given in the following figure

```
buf <name> (OUT1, IN1);    // Sets output equal to input
not <name> (OUT1, IN1);    // Sets output to opposite of input
and <name> (OUT, IN1, IN2); // Sets output to AND of inputs
or <name> (OUT, IN1, IN2); // Sets output to OR of inputs
nand <name> (OUT, IN1, IN2); // Sets output to NAND of inputs
nor <name> (OUT, IN1, IN2); // Sets output to NOR of inputs
xor <name> (OUT, IN1, IN2); // Sets output to XOR of inputs
xnor <name> (OUT, IN1, IN2); // Sets output to XNOR of inputs
bufif1<name> (out, in, cntrl)   // Sets output to input if ctrl is 1 tristate otherwise
bufif0<name> (out, in, cntrl)   // Sets output to not input if ctrl is 1 tristate otherwise
notif1<name> (out, in, cntrl)   // Sets output to input if ctrl is 0 tristate otherwise
notif0<name> (out, in, cntrl)   // Sets output to not input if ctrl is 0 tristate otherwise
```

Prototypes appear as for C or C++ function or procedure

The `<name>` for a gate instance

Must begin with a letter

Thereafter can be any combination of

letters,

numbers

underscore ‘_’

‘$’.

Gates with more than two inputs

Created by simply including additional inputs in the declaration

Output list appears first followed by the input list

**Example**

A five-input and gate is declared as

```
and <name> (OUT, IN1, IN2, IN3, IN4, IN5); // 5-input AND
```

Creating Combinational Logic Modules

At the gate level

Verilog module really is a collection of logic gates

Each time we declare and define a module

We are creating that set of gates
Structural or gate level model of a combinational circuit

Reflects the physical gates used to implement the design

To illustrate the basic process of

Creating a Verilog program

Modeling combinational logic at the gate level

Will begin with the following simple circuit.

Example of a simple module begins with following logic diagram

Module requires a name

Call it AndOr

Can analyze the module line by line

```
// Compute the logical AND and OR of inputs A and B.
module AndOr(AandB, AorB, A, B);
output AandB, AorB;
input A, B;
and myAnd (AandB, A, B);
or myOr (AorB, A, B);
endmodule
```

First line is a comment designated by the //

Everything on a line after a // is ignored

Comments can appear

On separate lines or at the end of lines of code

Top of module begins with keyword module

```
module AndOr(AandB, AorB, A, B);
output AandB, AorB;
input A, B;
```

Indicates

Start of module

Name of the module

AndOr

List of signals connected to that module
Subsequent lines
Declare
  First two binary values generated by module are outputs
  Next two (A, B) are inputs to the module
The next lines

```verilog
and myAnd (AandB, A, B);
or myOr (AorB, A, B);
```

Create instances of two gates
  AND gate called `myAnd` with output `AandB` and inputs A and B
  OR gate called `myOr` with output `orOut` and inputs A and B
We declare such intrinsic components
  Same as we did in C, C++ or Java with int, float, or char
The final line declares the end of the module

```verilog
endmodule
```

All modules must end with an `endmodule` statement
  Observe `endmodule` statement
  Is the only one that is not terminated by a semicolon

**Using Combinational Modules**
We build up a complex traditional software program by
  Having procedures call sub procedures
  Composing or aggregating classes into larger and more powerful structures
Verilog builds up complex circuits and systems from modules
  Using a design approach similar to composition or aggregation
To illustrate the process,
  Will use the previous `AndOr` module to build `NandNor` circuit
  Begin with the logic diagram and Verilog module in following figure
The *NandNor* module declares instance of the *AndOr* module

As it would any of the intrinsic types

One can declare multiple instances of a submodule

Another instance of the *AndOr* module

Could be added to the *NandNor* module

Each instance of the submodule

Creates a new set of gates

Three instance of *AndOr* would create a total of $2 \times 3 = 6$ gates

The *wire* Statement

```
wire XandY, XorY;
```

Used to connect the outputs of the *AndOr* module

To the two not gates
These wires comprise a net that carries the signals
From the output of the AndOr module
To the inverters

The Real-World Affects – Part 1
Let’s take a first look at incorporating real-world affects
Into HDL models

Timing and Delays – A First Look
In perfect world
Parts are ideal
Signals flow through wires and parts with no delay
In real world
Parts are not perfect
Signals are delayed by varying amounts
Verilog can model signal propagation delay through basic gates
Using the # operator

Device Delays
Basic syntax is given as

```verilog
Syntax
#delay device;
```

We modify the AndOr module
To incorporate delays into the design
To model real world behavior

```verilog
// Compute the logical AND and OR of inputs A and B.
module AndOr(AandB, AorB, A, B);
    output AandB, AorB;
    input A, B;

    and #5 myAnd (AandB, A, B);
    or #10 myOr (AorB, A, B);
endmodule
```
The following line of code states

```plaintext
and #5 myAnd (AandB, A, B);
```

The AND gate takes 5 time units to propagate
Change on the input to the output

The delay through the OR gate is twice as long
Taking 10 time units

```plaintext
or #10 myOr (AorB, A, B);
```

Units of time can be whatever we want
As long as we use consistent values

Net Delays
The delay operator
Can also be applied to a net
Why might this be important

When delay specified on a net
Any state change on input to net
Delayed accordingly
Syntax follows that of device delay

```plaintext
Syntax
#delay wire;
```

Using Symbolic Constants
One can use what are called *magic numbers*
More robust design
Will use named or symbolic constants
Variables whose value is
Set in one place
Used throughout a piece of code
Learned symbolic constant in Verilog
Called a parameter
Parameter defined and initialized
Using the following syntax

```
Syntax
parameter = aValue;
```

The following code fragment illustrates symbolic constant for
Inclusion of a delay of 2 time units in a part model

```
parameter propagationDelay = 2;
not #propagationDelay myNot (sigOut, sigIn);
```

Let’s modify the previous example to
Reflect more professional approach
Also incorporate the signal rise and fall times

```
// Compute the logical AND and OR of inputs A and B.
module AndOr(AandB, AorB, A, B);
    output AandB, AorB;
    input A, B;

    // specify real-world effects
    parameter delay0 = 5;
    parameter delay1 = 10;
    parameter riseTime = 3;
    parameter fallTime = 4;

    and #(riseTime, fallTime, delay0) myAnd (AandB, A, B);
    or #(riseTime, fallTime, delay1) myOr (AorB, A, B);
endmodule
```

Modified code sets
Gate delays delay0 and delay1
Rise and fall times
To the values specified by remaining two parameters
To speed up either gate
One could simply change the value in the parameter lines
Sequential Logic
Sequential logic modeled at the gate level
First developing the appropriate flip-flop module
Then implementing the design
As a composition of
Instances of that module
Necessary gates
Interconnecting the components with wires

SR Latch
Begin with the basic SR latch

```
module srLatch(q, qnot, s, r);
    input s, r;
    output q, qnot;
    parameter delay0 = 2;
    // implement the latch
    nor #delay0 n0(q, r, qnot);
    nor #delay0 n1(qnot, s, q);
endmodule
```
SR Latch with Enable

Basic design can be extended to include Enable as an additional level of control

```verilog
module gsrLatch(q, qnot, sg, rg, clr, enab);
  input sg, rg, clr, enab;
  output q, qnot;
  parameter delay0 = 2;

  // Build the gating logic
  not n0(nclr, clr);
  and and0(rL, rg, clr, enab);
  and and1(sL, sg, clr, enab);

  // Build the basic RS latch
  nor #delay0 n0(q, rL, nclr, qnot);
  nor #delay0 n1(qnot, sL, q);
endmodule
```

Master Slave Configuration

The master slave implementation using the gated latch follows naturally

```verilog
module srmsff(q, qnot, s, r, clr, clk);
  input s, r, clk, clr;
  output q, qnot;

  not n0(nclk, clk);
  gsrLatch master(qm, qnotm, s, r, clr, clk);
  gsrLatch slave(q, qnot, qm, qnotm, clr, nclk);
endmodule
```
Binary Counter

Can use the SR flip-flop to build

Simple two-bit synchronous binary up counter

// Build a two bit binary up counter using master slave SR flip-flops

/*
 tn  tn+1
 A B A B
 0 0   0 1
 0 1   1 0
 1 0   1 1
 1 1   0 0
*/

module TwoBitCntr(qA, qB, clr, clk);
    input clr, clk;
    output qA, qB;
    and a1(sA, qAnot, qB);
    and a2(rA, qA, qB);
    srmsff FFB(qB, qBnot, qBnot, qB, clr, clk);
    srmsff FFA(qA, qAnot, sA, rA, clr, clk);
endmodule

Summary

In this lesson we

✓ Introduced Verilog Modeling and the Hardware Design Language.
✓ Introduced some of the basics of the Verilog language and structure of a Verilog program.
✓ Began to explore some of the Verilog terminology and primitives.
✓ Introduced concepts of Verilog variables, nets, and multibit signals.
✓ Introduced Verilog numbers and number specifications.
✓ Introduced some of the intrinsic data types that are part of the Verilog language.
✓ Studied and developed Verilog structural level models of combinational and sequential circuits.
✓ Introduced real-world affects into the models.