Introduction to Verilog I

CMPE 415 Programmable Logic Devices

Prof. Ryan Robucci

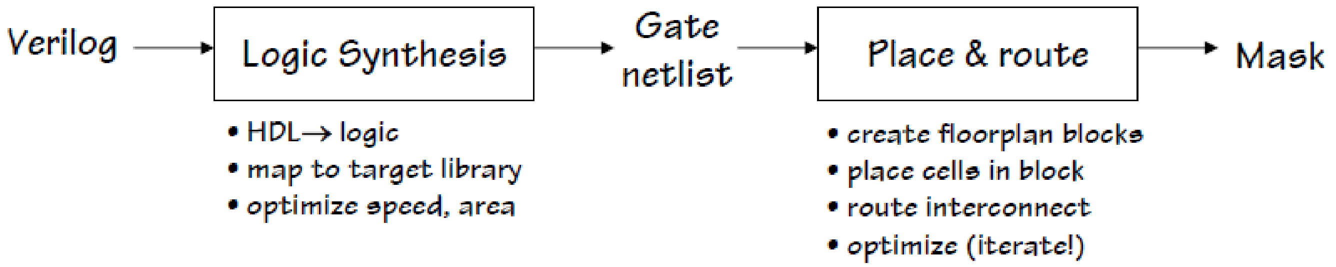
UMBC
DEPARTMENT OF COMPUTER SCIENCE AND ELECTRICAL ENGINEERING

References

† http://6004.csail.mit.edu/6.371/handouts/L0{2,3}.pdf
marked in Red Outline Shape

Why use an HDL?

- Want an executable functional specification
  - Document exact behavior of all the modules and their Interfaces
  - Executable models can be tested & refined until they do what you want
- Too much detail at the transistor and mask levels
  - Can't debug 1M transistors as individual analog components
  - Abstract away "unnecessary" details
  - Play by the rules: don't break abstraction with clever hacks
- HDL description is first step in a mostly automated process to build an implementation directly from the behavioral model



http://6004.csail.mit.edu/6.371/L03.pdf

Abstraction

- Abstraction is a cornerstone of digital design.
- HDLs allow us to model hardware with varying levels of abstraction. They allow us to flexibly describe and represent not only functionality, but also implementation and structure at varying degrees. For the purpose of simulation, the most significant difference from functional modeling in software is the level of support for representing timing (delays) and concurrent execution.

A Tale of Two HDLs

VHDL	Verilog
ADA-like verbose syntax, lots of redundancy (which can be good!)	C-like concise syntax
Extensible types and simulation engine. Logic representations are not built in and have evolved with time (IEEE-1164).	Built-in types and logic representations. Oddly, this led to slightly incompatible simulators from different vendors.
Design is composed of entities each of which can have multiple architectures. A configuration chooses what architecture is used for a given instance of an entity.	Design is composed of modules.
Behavioral, dataflow and structural modeling. Synthesizable subset...	Behavioral, dataflow and structural modeling. Synthesizable subset...
Harder to learn and use, not technology-specific, DoD mandate	Easy to learn and use, fast simulation, good for hardware design

Important Verilog Coding Styles

- **Structural models**: basically a hierarchical netlist starting with “primitives” and modules built using other styles.
- **Dataflow models**: combinational logic described using expressions
- **Behavioral models**: This level describes a system by concurrent “algorithms” (Behavioral). Each algorithm itself is sequential, that means it consists of a set of instructions that are executed one after the other. There is no regard to the structural realization of the design.
- **Register-Transfer Level** (RTL) register-focused design.
  - Registers are identified, and the movement of data between them at specific specified timing events like clock edges logic is described. Modern RTL code definition is "Any code that is synthesizable is called RTL code".

Structural

- Structural models: basically a hierarchical netlist with “primitives” (built-in Verilog logic gates, or instances of library modules).
- Instantiation of Modules
- Use of Gate Level Primitives
  - Within the logic level the characteristics of a system are described by logical links and their timing properties. All signals are discrete signals. They can only have definite logical values (`0', `1', `X', `Z`). The usable operations are predefined logic primitives (AND, OR, NOT etc gates). Using gate level modeling might not be a good idea for any level of logic design. Gate level code is generated by tools like synthesis tools and this netlist is used for gate level simulation and for backend. http://www.asic-world.com/verilog/intro1.html
- Use of Switch Level Primitives
  - Switch Level modeling allows you to construct transistor-level schematic model of a design from transistor and supply primitives
  - nmos, pmos, supply1, supply0, etc...

Structural Verilog

```
// 2-to-4 demultiplexer with active-low outputs
// structural model
module demux1(a,b,enable,z);
  input a,b,enable;
  output [3:0] z;

  wire abar,bbar; // local signals

  not v0(abar,a)
  not v1(bbar,b);
  nand n0(z[0],enable,abar,bbar);
  nand n1(z[1],enable,a,bbar);
  nand n2(z[2],enable,abar,b);
  nand n3(z[3],enable,a,b);

endmodule
```

http://6004.csail.mit.edu/6.371/L02.pdf

Remember that statements "run" concurrently so order in code isn't significant!

Dataflow modeling

- Dataflow models: combinational logic described using expressions
  - assign target = expression
  - Arithmetic operators: +, -, *, /, %, >>, <<
  - Relational operators: <, <=, ==, !=, >=, >, ===, !==
  - Logical operators: &&, ||, !, ?:
  - Bit-wise operators: ~, &, |, ^, ~^, ^~
  - Reduction operators: &, ~&, |, ~|, ^, ~^
  - Concatenation, replication: {sigs…} {number{…}}

http://6004.csail.mit.edu/6.371/handouts/L02.pdf

http://6004.csail.mit.edu/6.371/L02.pdf

- Structural Verilog may include many of the dataflow operations that map directly to built-in logic primitives and specifications of net connections.
  - The following are the same in many contexts
    - x = a & b & c;
    - nand n0(x,a,b,c);
  - The exact implementation and structure implied in the following is less certain unless we explicitly know the exact module that addition would map to with our synthesizer and library
    - x = a+b

Dataflow Verilog

```
// 2-to-4 demultiplexer with active-low outputs
// dataflow model
module demux2(a,b,enable,z);
  input a,b,enable;
  output [3:0] z;

  assign z[0] = | {~enable,a,b}; //reduction operator
  assign z[1] = ~(enable & a & ~b);
  assign z[2] = ~(enable & ~a & b);
  assign z[3] = enable ? ~(a & b) : 1'b1; //conditional
                                          // expression like C
endmodule
```

http://6004.csail.mit.edu/6.371/L02.pdf

Behavioral Code and RTL Code

- Behavioral code is implemented in <u>procedural blocks</u> that include one or several statements that describe an algorithm to define the behavior of a block of logic in a simulation or in hardware
- A procedural block may include <u>sequential statements</u> from which the algorithm may be understood by beginning interpretation of statements one at a time (similar to traditional software coding languages) or <u>parallel statements</u> intended to be interpreted in parallel.
  - **begin**…**end** block include code with sequential statements
  - **fork**…**join** blocks include code with parallel statements
- The creation of behavioral code is **sometimes** characterized by a lack of regard for hardware realization
- **Synthesizable Behavioral Code** is code that a given synthesizer can map to a hardware implementation
  - The definition of synthesizable is synthesizer dependant- some simple prodecural code constructs are universally synthesizable by every synthesizer, while more complex code blocks and certain operators are not considered synthesizable by many
    - Example:
      - **x = myUINT8 >> 2;**
        - This is a shift by a constant implemented by a simple routing of bits. It is generally regarded as synthesizable
      - **x = myUINT8 >> varShift;**
        - This is a variable shift with many possible implementations. It will simulate just fine, but at the synthesis step many synthesizers will   throw an error saying that this is not synthesizable though it is ex
  - Behavioral code may indeed describe behavior in such a way that is not directly synthesizable by almost any synthesizer (such as reading waveforms from a .txt file) – though what is “synthesizable” is always defined by the synthesizer tool being used
- Procedural code implemented with regard for hardware implementation, from which the registers, the combinatorial logic between, and control signals like clocks may be inferred is called <u>Register Transfer Level</u> (RTL) code
  - Sometimes the terms “behavioral code” and “RTL code” are used in proximity to refer to synthesizable and non-synthesizable code, though even this separation is dependent on the synthesizer tool being used

Initial and Always Blocks

- Initial and Always blocks will be the first two types of blocks we will discuss (tasks and functions will be discussed later)
- Initial blocks are triggered once at the start of a simulation or in the case of some synthesis tools may be used to describe the power-up state of registers or may be used to describe the initial default value of an intermediate variable
- Always blocks are triggered with every change in one or more signals as provided in a sensitivity list.
  - When describing combinatorial logic the sensitivity list should include every input to the logic
  - For coding sequential logic the sensitivity list should include only the control signals that trigger updates to sequential logic
    - Example Control signals to include in a sensitivity list for seq. logic blocks:
      - *enable* for latches
      - clock for more traditional registers or flip-flops
      - any additional asynchronous controls like an asynchronous set and asynchronous reset

Behavioral Verilog

reg $\neq$ "register"

```
//  2-to-4 demultiplexer with active-low outputs
// behavioral model
module demux3(a,b,enable,y);
  input a,b,enable;
  output [3:0] y;
  reg y; // not really a register!
  always @(a, b, enable)
    case ({enable,a,b})
     default: y = 4'b1111;
     3'b100: y = 4'b1110;
     3'b110: y = 4'b1101;
     3'b101: y = 4'b1011;
     3'b111: y = 4'b0111;
  endcase
endmodule
```

http://6004.csail.mit.edu/6.371/L02.pdf

Beginner's note

Here is something to be cleared up right away when learning Verilog "reg" is just a variable. In fact it is called a "variable" as of Verilog 2000 because the name is so confusing. So, don't be confused by it, "reg" is not necessarily register, They are used in behavioral descriptions as variables that may end up being implemented with if sequential logic is generated and are just represent the output net of combinatorial logic otherwise. Wires on the other hand are for structural connections (nets/wires) between modules or outputs of combinatorial expressions.

Thus we shall always refer to a statement "reg y;" as "reg why" not "register why"

Behavioral Verilog

```
// 2-to-4 demultiplexer with active-low
outputs
// behavioral model
module demux3(a,b,enable,y);
  input a,b,enable;
  output [3:0] y;
  reg y; // not really a register!
  always @(a, b, enable)
    case ({enable,a,b})
     default: y = 4'b1111;
     3'b100: y = 4'b1110;
     3'b110: y = 4'b1101;
     3'b101: y = 4'b1011;
     3'b111: y = 4'b0111;
  endcase
endmodule
```

Contents of "sensitivity list" needs careful attention. A bug here is a most common cause for differences between Verilog simulation and synthesized hardware.

Since y is always assigned a value in the body of the always block, it's value doesn't have to be remembered between executions. Therefore no state needs to be saved, i.e., no register needs to be created.

Sequential vs Combinatorial Logic in hardware synthesis using always begin…end blocks (Rules of Thumb)

- if you could <u>ignore the sensitivity list</u> and <u>reevaluate the procedural block at any and every instant of time</u> and there would be no change the overall interpretation of the algorithm then **combinatorial hardware** is described meaning it can be mapped to a set of output input relationships described by a combinatorial truth table and no memory of the past is required
- if the restriction of only allowing revaluation of the block contents when specific changes occur according to the sensitivity list would cause a difference in behavior at any point in time then sequential logic is described
- If results from any execution of the block directly rely on signals/results generated from a previous execution of the block then sequential logic is describe. This does not include the case when results are saved using external sequential logic.
- Draw the truth table for each block with and without considering the sensitivity list, it should include a row for every possible input combination and the output variables should should never occur in the output columns:

```
always @ (a,b,c) begin
  x = (c & a) | (~c & b);
end
```

```
always @ (a,c) begin
  x = (c & a) | (~c & x);
end
```

```
always @ (a,b,c) begin
 if c x = a;
 else x = b;
end
```

```
always @ (c) begin
  if c x = a;
  else x = b;
end
```

```
always @ (a,b,c) begin
 if c x = a;
end
```

Viewing results

```
module main;
  reg a,b,enable;
  wire [3:0] s_z,d_z,b_z;

  demux1 structural(a,b,enable,s_z);
  demux2 dataflow(a,b,enable,d_z);
  demux3 behavioral(a,b,enable,b_z);
  initial begin
    $dumpfile("demux.vcd"); //Specify file for
                //Value Change Dump (VCD) info
    $dumpvars(1,main);
    enable = 0; a = 0; b = 0; //Force one last change
                 //at final time for better display
    #10 enable = 1;
    #10 a = 1;
    #10 a = 0; b = 1;
    #10 a = 1;
    #10 enable = 0;
    $finish;
  end
endmodule
```

Dump variables in module main. First arg is # of levels to dump, eg, “2” would include variables from modules instantiated by main. $dumpvars with no args will dump everything.

Code from the demux examples can be found in /mit/6.371/examples/demux.vl

Simulating

- The stimulus (input)
  - Designs can be instantiated and driven by other HDL code, typically called a testbench, that drives test signals
  - Alternatively, some simulators support a scripting language to drive input signals
- The output
  - Use $display $monitor or $strobe statements to print result to screen or file
  - Create a value change dump file (VCD)
    - Can be read and displayed by many tools
  - May Directly use a GUI to select and display signals

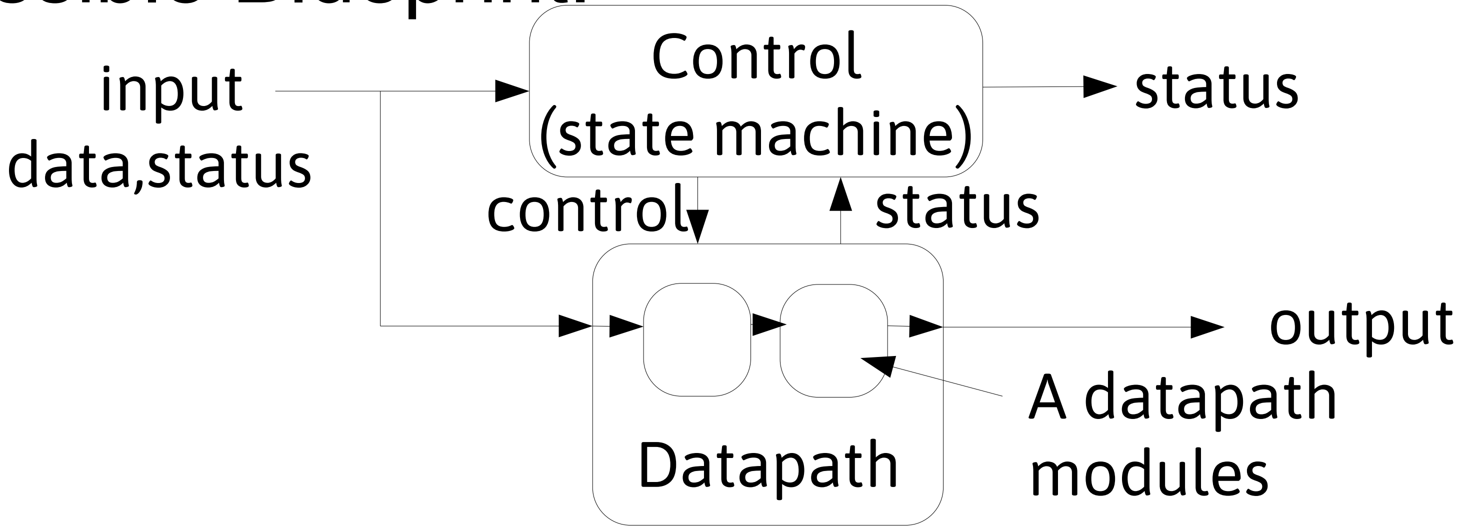
Design Strategies

- For a beginner, treat Verilog as Hardware Description Language, not a software coding language. Start off learning Verilog by describing hardware for which you can design and draw a schematic; then translate this to HDL.
- Plan by partitioning the design into sections and modules and coding styles that should be used.
- Identify existing modules, memory components needed, and data-path logic, as well as the control signals required to make those operate as desired.
- Simulate each part with a testbench before putting system together. Update testbenches and resimulate them as your design evolves.

- Large memory blocks are often provided by the manufacturer to be instantiated. Smaller memory elements may be coded or embedded into other descriptions of the design
- Data-path logic can be embedded coded with data-flow, structural elements, or complex synthesizable behavioral descriptions.
- Some styles explicitly separate Comb. Logic and Seq Logic, but this is up to you.
- **Best practice is to develop a consistent approach to design, as well as a consistent coding style. It makes designing, coding, and debugging easier for you with time. An inconsistent hack-it-together and hack-until-it-works approach is not conducive to becoming more efficient.**

- Typically, complex control is implemented by a synthesizable behavioral case-statement-based state-machine, while simpler control could be implemented with any combinatorial description style. Data-path logic (comb. and sequential) can be integrated into the overall state machine or separated out (better for incremental simulation).

- Possible Blueprint:



Components of a modeling/description language

- Wires and registers with specified precision and sign
- Arithmetic and bitwise operations
- Comparison Operations
- Bitvector Operations (selecting multiple bits from a vector, concatenation)
- Logical Operators
- Selection (muxes)
- Indexed Storage (arrays)
- Organizational syntax elements and Precedence
- Modules (Hardware) Definition and Instantiation

Modules and Ports

*code from http://www.verilogtutorial.info/chapter_2.htm

keyword **module** begins a module

port list: ports must be declared to be input, output, or inout

```
`timescale 1ns / 1ps
//create a NAND gate out of an AND and an Inverter
module some_logic_component (c, a, b);
  // declare port signals
  output c;
  input a, b;

  // declare internal wire
  wire d;

  //instantiate structural logic gates
  and a1(d, a, b); //d is output, a and b are inputs
  not n1(c, d);    //c is output, d is input
endmodule
```

Additional internal nets may be declared using the wire or reg word

keyword endmodule begins a module

nodes can be connected to nested modules or primitives or interact with procedural code

Verilog 2000: New Port Decl. Options

Verilog 95 code

```
module memory( read, write, data in, addr, data out);
input  read;
input  write;
input  [7:0] data_in;
input  [3:0] addr;
output  [7:0] data_out;

reg [7:0] data_out;
```

After the port list, port direction must be declared to be **input**, **output**, or **inout** as well as the width **if more than one bit** Type declaration: type is by default a **wire** unless another type is provided

Verilog 2k with direction and data type listed

```
module memory(
  input wire read,
  input wire write,
  input wire [7:0] data_in,
  input wire [3:0] addr,
  output reg [7:0] data_out
);
```

Verilog 2k with no type in port list

```
module memory(
  input  read,
  input  write,
  input  [7:0] data_in,
  input  [3:0] addr,
  output reg [7:0] data_out
);
```

declared as wire by default

Disadvantage to exposing type in port declaration

Verilog 2000 – port y as reg

```
module dff2y(output reg qA,
             output reg qB,
             output reg y,
             input dA,
             input dB,
             input en_n,
             input clk)

  always@(posedge clk)
    if (~en_n) begin
      qA <= dA;
      qB <= dB;
    end

  always @(qA,aB) begin
    y = 1'b0;
    if (a&b) begin
      y = 1;
    end
  end
end
```

Verilog 2000 – port y as wire

```
module dff2y(output reg qA,
             output reg qB,
             output wire y,
             input dA,
             input dB,
             input en_n,
             input clk)

  always@(posedge clk)
    if (~en_n) begin
      qA <= dA;
      qB <= dB;
    end

    assign y = qA&qB;
end
```

y is not the output of a register though declared as reg along with qA and qB in the left module declaration
It is arguable that such an internal coding implementation detail does not belong in the presentation of an “external” interface

Disadvantage to exposing type in port declaration

Verilog 2000 -- hiding reg while exposing port direction

```
module dff2y(  output qA,
               output qB,
               output y,
               input dA,
               input dB,
               input en_n,
               input clk)

  reg qA_int,qB_int,y_int;

  assign qA = qB_int;
  assign qB= qA_int;
  assign y= y_int;
```

```
always@(posedge clk)
    if (~en_n) begin
      qA_int <= dA;
      qB_int <= dB;
    end

  always @(qA,aB) begin
    y_int = 1'b0;
    if (a&b) begin
      y_int = a&b;
    end
  end
end
```

Disadvantage to exposing type in port declaration

- With V'95, the type is not externally exposed in the port definition

Verilog 95

```
module dff2y(qA,qB,   //dff out A B
             y,   //and of qA&qB
             dA,dB,   //dff inputs
             en_n,   //input clk en
             clk) //clk

output reg qA;
output reg qB;
output reg y;
input dA;
input dB;
input en_n;
input en_n;

always@(posedge clk) begin
    if (~en_n) begin
      qA <= dA;
      qB <= dB;
    end
  end

  always @(qA,aB) begin
    y = 1'b0;
    if (a&b) begin
          y = a&b;
    end
  end
end
```

```
output reg qA;
output reg qB;
output y;
input dA;
input dB;
input en_n;
input clk;

always@(posedge clk)
begin
    if (~en_n) begin
      qA <= dA;
      qB <= dB;
    end
  end

  assign y = a&b;

end
```

Hierarchy and Instantiation

- Implicit port mapping uses the order of the ports in the definition to imply connections to local nets

Instantiation with Implicit Port Mapping

```
module dff2y_en(  output qA, qB,
                  output y,
                  input dA, dB,
                  input en,
                  input clk)

  wire en_n = ~en; { wire en_n;
                     assign en_n = ~en;

  dff2y dff2yInstance(
           qA,qB,  //dff out A B
               y,  //and of qA&qB
           dA,dB,  //dff inputs
            en_n,  //input clk en
             clk); //clk
endmodule
```

At this level of the hierarchy y is a wire regardless of the internal code implementation of y in dff2y

Hierarchy and Instantiation

- Explicit port mapping uses the port names prefixed with . and allows **reordering**, **no-connect**, and **omission** of ports

Instantiation with Explicit Port Mapping

```
module dff2y_en(  output y,
                  input dA, dB,
                  input en,
                  input clk)

  dff2y dff2yInstance(
           .dA(dA),
           .dB(dB),
           .qA(), //qA not used (no-connect)
                  //qB omitted
           .y(y),
           .en_n(~en),
           .clk(clk)); //clk
endmodule
```

Implicit net declaration of a net holding the result ~en.
This is NOT allowed if default net declaration is disabled. ie using `` `default_net_type none ``