# Overview

- Algorithm development is usually done in MATLAB, C, or C++
- Code must be structured such that H/W and S/W designers can correlate various components
- H/W is written in Verilog (or any other HDL).
- SystemVerilog enhances some of the features of Verilog for design, but more importantly for verification
Verilog – An Overview

• “Though Verilog is C like in syntax but has distinct character and interpretation. A programmer must set his perception right before coding in Verilog. He must visualize hardware in his mind while structuring Verilog modules consisting of procedural blocks and assignments.”

• Some constructs in Verilog is for supporting verification, modeling, and simulation and do not synthesize (RTL Verilog is synthesizable).

System Level Design

Algorithm Development

Design Specification  →  Floating Point Behavioral Description

S/W  →  Fixed Point Implementation S/W

S/W – H/W Partitioning

S/W – H/W Co-verification

Software development and testing

System level design verification and testing

System Integration & Testing

Hardware Development and Testing

H/W  →  Fixed point conversion

RTL Verilog Implementation

Functional Verification

Synthesis

Gate level netlist

Timing & Functional Verification

Layout
Verilog

- A Verilog code has a top level module which may instantiate many other modules.
- The module is the basic building block in Verilog
- A module when synthesized infers digital logic.
- The designer conceives the design as hierarchically interconnected lower level modules forming higher level modules.
- Each module starts with the keyword `module`, followed by the module name and parameters list. The module is ended with the keyword `endmodule`.

Modules

```verilog
module FA (<port declaration>);

assign {c_out,sum} = a+b+c_in;
endmodule
```
RTL Design

- At RTL level the designer must know all the registers in the design
- The computations performed are modeled by a combinational cloud
- Gate level details are not important
- HDLs Verilog/VHDL are used to implement a design at RTL level
- Verilog resembles with C and is usually preferred in industry
Verilog Standards


Design Partitioning

- If possible partition such that boundaries reside at register outputs.
- No combination cloud crosses module boundaries
- Gives synthesis tool more leverage to generate optimized logic
System Level Design Tips

- Algorithms are designed in MATLAB, C, ...
- Must meet the Requirements and specifications
- System designer must define all the components (input, output, and functionality). and the data flow among the components.
- The components are mapped to ASIC, FPGA, or embedded DSP’s.
- It is preferred that the designers in the subsequent phases to stick to the same components and variables names as far as possible to ease going back and forth in the design cycle

Hierarchical Design

![Hierarchical Design Diagram]
Hierarchical Design

• Another way to do it without the need to order the parameters

module RCA (  
  input [2:0] a,b,  
  input c_in,  
  output [2:0] sum,  
  output c_out);  
wire carry[1:0];  
//Module instantiation  
FA fa0(a[0],b[0],c_in, sum[0],carry[0]);  
FA fa1(a[1],b[1],carry[0], sum[1],carry[1]);  
FA fa2(a[2],b[2],carry[1], sum[2],c_out);  
endmodule
Synthesis Guidelines

• Avoid glue logic that connects 2 modules
• Individual modules may satisfy timing conditions, but top level module may not.
• Such a logic should be made as a part of the combinational logic in one or the two modules.
• This may arise in debugging or after changing/correcting one of the interfaces of a module.
Synthesis Guidelines

- Design modules with common design objectives
- Try to avoid putting time-critical and non-time-critical logic in the same modules.
- Time critical logic will be synthesized for best timing, while non-time-critical logic may be synthesized for minimum area.
- Split them into two modules
Verilog Syntax

- 0 zero, logic low, false, or ground
- 1 one, logic high, or power
- x unknown
- z high impedance, unconnected, or tri-state port
- A number in Verilog may contain all four possible values:
  - Example: 20"b 0011_1010_101x_x0z0_011z
  - The underscore (_) is ignored by simulation and synthesis tools. It is meant to give a better visualization of the number
Data Types

• Nets
  – Nets are physical connections between components
  – Nets always show the logic value of the driving components
  – Many types of nets, we use wire in RTL
  – Usually is the output of a combinational logic

• Registers
  – Implicit storage – unless variable of this type is modified it retains previously assigned value
  – Does not necessarily imply a hardware register
  – Register type is denoted by reg

Variable Declaration

• Declaring a net, signed or unsigned
  – wire [<signed>] [<range>] <net_name> [<net_name>*];
  – Range is specified as [MSB:LSB]. Default is one bit wide

• Declaring a register
  – reg [<signed>] [<range>] <reg_name> [<reg_name>*];

• Declaring memory
  – reg [range] <memory_name> [start_addr : end_addr];

• Examples
  – reg r; // 1-bit reg variable
  – wire x1, x2; // 2 1-bit wire variable
  – reg signed [7:0] y_reg; // 8-bit sign register
  – reg [7:0] ram_mem [0:1023]; // a 1 KB memory
Constants

• Constants can be written in
  • decimal (default)
    – 13, ’d13
  • binary
    – 4'b1101
  • octal
    – 4’o15
  • hexadecimal
    – 4’hd

Levels of Abstraction

• The HW can be described at several levels of details
• To capture these details Verilog provides four levels of abstraction
  1. Switch level
  2. Gate level
  3. Dataflow level
  4. Behavioral or algorithmic level
Levels of Abstraction

• **Switch Level**: The lowest level of abstraction is the switch or transistor level modeling.
• **Gate Level**: Synthesis tools compile high level code and generate code at gate level.
• **Dataflow Level**: The level of abstraction higher than the gate level.
• **Behavioral Level**: In more complex digital designs, priority is given to the performance and behaviour at algorithmic level.

Switch Level

• The lowest level of abstraction (switch and transistor level modeling).
• Rarely used at the level we are talking about in this course.
Gate level– Structural Modeling

• Are built from gate primitives
• Verilog has built-in gate-level primitives NAND, NOR, AND, OR, XOR, BUF, NOT, and some others
• Describe the circuit using logic gates—much as you have seen in an implementation of a circuit in basic logic design course
• The delay can also be modeled
• Typical gate instantiation is

    and #delay instance-name (out, in1, in2, in3, ...)

    module mux (out, in1, in2, sel);
    output out;
    input in1, in2, sel;
    wire out1, out2, sel_n;
    and #5 a1(out1, in1, sel_n);
    and #5 a2(out2, in2, sel);
    or #5 o1(out, out1, out2);
    not n1(sel_n, sel);
    endmodule
Data Flow Modeling

• Expressions, operands and operators form the basis of data flow modeling

Operators for Dataflow Modeling

<table>
<thead>
<tr>
<th>Type</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>+ - = * / % **</td>
</tr>
<tr>
<td>Binary bitwise</td>
<td>~ &amp; ~&amp;</td>
</tr>
<tr>
<td>Unary reduction</td>
<td>&amp; ~&amp;</td>
</tr>
<tr>
<td>Logical</td>
<td></td>
</tr>
<tr>
<td>Relational</td>
<td>&lt; &gt; &lt;= &gt;=</td>
</tr>
<tr>
<td>Logical shift</td>
<td>&gt;&gt; &lt;&lt;</td>
</tr>
<tr>
<td>Arithmetic shift</td>
<td>&gt;&gt;&gt; &lt;&lt;&lt;</td>
</tr>
<tr>
<td>Conditional</td>
<td>?:</td>
</tr>
<tr>
<td>Concatenation</td>
<td>0</td>
</tr>
<tr>
<td>Replication</td>
<td>{{}}}</td>
</tr>
</tbody>
</table>
Conditional Operator

```plaintext
out = sel ? a : b;
```

This statement is equivalent to

```plaintext
if(sel)
    out = a;
else
    out = b;
```

Conditional operator can also be used to infer higher order multiplexers. The following code infers a 4:1 mux

```plaintext
out = sel[1] ? ( sel[0] ? in3 : in2 ) : ( sel[0] ? in1 : in0 );
```

Concatenation and Replication

- `{ }` means concatenation of what is inside the curly braces

```plaintext
p={a[3:0], b[2:0], 3'b111, c[2:0]};
```

```
    p
   /   |
  MSB  v  LSB
   |
   v
   13 bits
```
Replication

• \{n\}\{\} means replication

A = 2'b01;
B = \{4(A)\} // the replication operator

The operator replicates A four times and assigns the replicated value to B.

Thus B = 8'b 01010101

Relational Operators

<table>
<thead>
<tr>
<th>Relational Operator</th>
<th>Operator Symbol</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal</td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal</td>
<td></td>
</tr>
</tbody>
</table>
### Reduction Operators

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operator Symbol</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduction</td>
<td>&amp;</td>
<td>Reduction and</td>
</tr>
<tr>
<td></td>
<td>~&amp;</td>
<td>Reduction nand</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>~</td>
<td></td>
</tr>
<tr>
<td></td>
<td>^</td>
<td>Reduction xor</td>
</tr>
<tr>
<td></td>
<td>^</td>
<td>or ^</td>
</tr>
</tbody>
</table>

### Bitwise Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operator Symbol</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitwise</td>
<td>~</td>
<td>Bitwise negation</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td></td>
<td>~&amp;</td>
<td>Bitwise NAND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>~</td>
<td></td>
</tr>
<tr>
<td></td>
<td>^</td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td></td>
<td>^</td>
<td>or ^</td>
</tr>
</tbody>
</table>
### Equality Operators

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operator Symbol</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equality</td>
<td>==</td>
<td>Equality</td>
</tr>
<tr>
<td></td>
<td>!=</td>
<td>Inequality</td>
</tr>
<tr>
<td></td>
<td>===</td>
<td>Case Equality</td>
</tr>
<tr>
<td></td>
<td>!==</td>
<td>Case Inequality</td>
</tr>
</tbody>
</table>

### Logical Operators

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operator Symbol</th>
<th>Operation Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical</td>
<td>!</td>
<td>Logical Negation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;&amp;</td>
<td>Logical AND</td>
</tr>
</tbody>
</table>
Shift Operators

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operator Symbol</th>
<th>Operation Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Shift</td>
<td>&gt;&gt;</td>
<td>Unsigned Right Shift</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;</td>
<td>Unsigned Left Shift</td>
</tr>
<tr>
<td>Arithmetic Shift</td>
<td>&gt;&gt;&gt;</td>
<td>Signed Right Shift</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;&lt;</td>
<td>Signed Left Shift</td>
</tr>
</tbody>
</table>

Examples

- **Shift an unsigned reg** A = 6'b101111 by 2
  - B = A >> 2;
  - drops 2 LSBs and appends two zeros at MSBs position, thus
  - B = 6'b001011
- **Arithmetic shift right a wire** A = 6'b101111 by 2
  - B = A >>> 2;
  - This operation will drop 2 LSBs and appends the sign bit to 2 MSBs locations.
  - Thus B is 6'b111011.
Examples

• Apply & reduction operator on a 4-bit number  \( A=4'b1011 \)
• assign out = &A;
• This operation is equivalent to performing a bitwise & operation on all the bits of A i.e.

Dataflow Level

• At this level every expression is modeled with the assign keyword
• assign c=a+b
• The value on the wire c is continuously driven by the result of the arithmetic operation
• RHS must be a wire
• Operands my be wire or reg
Example Dataflow Modeling

module adder_4 (a, b, ci, s, co);
input [3:0] a, b;
input ci;
output [3:0] s;
output co;
assign (co, s) = a + b + ci;
endmodule

Example 2;1 MUX

module stimulus;
reg IN1, IN2, SEL;
wire OUT;
mux2_1 MUX(IN1, IN2, SEL, OUT);
initial
begin
IN1 = 1; IN2 = 0; SEL = 0;
#5 SEL = 1;
#5 IN1 = 0;
end
initial
$monitor($time, ": IN1=%b, IN2=%b, SEL=%b, OUT=%b\n", IN1, IN2, SEL, OUT);
endmodule
Behavioral Modeling

- High level language constructs are used
  - for loop
  - if else
  - while etc …
- All statements come in a procedural block
  - always
  - initial
- A subset of constructs are synthesizable and called RTL Verilog
- The designer must know the HW consequences of the code (for loop …)

Behavioral Models

- All variables on the LHS must be of type “reg”.
- RHS may be reg or wire
Initial Block

- This block starts with initial keyword
- This is non synthesizable
- Non RTL
- This block is used only in stimulus
- All initial blocks execute concurrently in an arbitrary order
- They execute until they come to a #delay operator
- Then they suspend, putting themselves in the event list delay time units in the future
- At delay units, they resume executing where they left off

Procedural Assignments

- Blocking assignment
  - Represented as “=” (for combinational circuits)
  - Assignment takes place immediately
  - Blocks, i.e. next statement will wait (blocked) until the assignment is completed

```
always
begin
  A=B;
  B=A;
end
```

A+B, B+B
Procedural Assignments

- **Nonblocking assignment**
  - Represented as “<=“ (for synchronous logic)
  - Assignment takes place at the end of the block
  - LHS are calculated first, then assigned to RHS

```
always
begin
    A<=B;
    B<=A;
end
```

Swaps A and B

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>req sum, carry; always @(x or y) begin sum = x^y; carry = x&amp;y; end</td>
<td>reg sum, carry; always @(x, y) begin sum = x^y; carry = x&amp;y; end</td>
<td>reg sum, carry; always @(*) begin sum = x^y; carry = x&amp;y; end</td>
</tr>
</tbody>
</table>

(a) Verilog-95 style  (b) Verilog-2001 support of comma separated sensitivity list  
(c) Verilog-2001 style that only writes * in the list

```
reg sum_reg, carry_reg;
always @(posedge clk)
begin
    sum_reg <=x^y;
    carry_reg <=x&y;
end
```
Multiple Procedural Assignments

• All procedural blocks start execution at \( t=0 \).
• The simulator schedules their execution in an arbitrary order
• All assignments to the same variables must be in the same block, otherwise pre and post synthesis simulation could be different (if synthesized at all).

Time Control

• \$\text{time} \quad$
  – A built-in variable that represents simulated time
  – A unitless integer
  – \$\text{display}($\text{time}$, “\text{a}=%d”, a);
• \# Time Control
  – \#<\text{number}> statement
  – statement is not executed until \(<\text{number}> \) time units have passed
  – control is released so that other processes can execute
  – used in test code
  – used to model propagation delay in combinational logic 60
  – \text{xor} \#2 x2(c,a,b)
@ Time Control

- @(*)
- @(expression)
- @(expression or expression or …)
- @(posedge onebit)
- @(negedge onebit)
- do not execute statement until event occurs
- @(clk) is same as @(posedge clk or negedge clk)

RTL Coding Guidelines

- Avoid Combinational feedback. It does not make sense
- If logic requires it, a register must be placed in the feedback loop

```vhdl
reg [15:0] acc;
always @(acc)
  acc = acc + 1;
```
Registers

//register with asynchronous active low reset
always @(posedge clk or negedge rst_n)
begin
  if(!rst_n)
    r_reg <= 'b'0;
  else
    r_reg <= data;
end

//register with asynchronous active high reset
always @(posedge clk or negedge rst_n)
begin
  if(rst_n)
    r_reg <= 'b'0;
  else
    r_reg <= data;
end

//register with synchronous active low reset
always @(posedge clk)
begin
  if(!rst_n)
    r_reg <= 'b'0;
  else
    r_reg <= data;
end

//register with synchronous active high reset
always @(posedge clk)
begin
  if(rst_n)
    r_reg <= 'b'0;
  else
    r_reg <= data;
end

Example -- Accumulator

// Register with asynchronous active-low reset
always @(posedge clk or negedge rst_n)
begin
  if(!rst_n)
    acc_reg <= 16'b0;
  else
    acc_reg <= data+acc_reg;
end

// Register with asynchronous active-high reset
always @(posedge clk or posedge rst)
begin
  if(rst)
    acc_reg <= 16'b0;
  else
    acc_reg <= data+acc_reg;
end
Generating Clock in Stimulus

```verilog
initial // All the initializations should be in the initial block
begin
  clk = 0; // clock signal must be initialized to 0
  # 5 rst_n = 0; // pull active low reset signal to low
  # 2 rst_n = 1; // pull the signal back to high
end
always // generate clock in an always block
  #10 clk = ~clk;
```

Case Statement

```verilog
module mux4_1(in1, in2, in3, in4, sel, out);
  input [1:0] sel;
  input [15:0] in1, in2, in3, in3;
  output [15:0] out;
  reg [15:0] out;
  always @(*)
    case (sel)
      2'b00: out = in1;
      2'b01: out = in2;
      2'b10: out = in3;
      2'b11: out = in4;
      default: out = 16'bx;
    endcase
  endmodule
```

MUX whose output depends on sel

If multiple matches, the first one is executed only

@(*) automatically populates the sensitivity list with all variables in the RHS

Case is inside an always block
Casex and Casez

• To make comparison with the „don"t care"
  – casez takes z as don"t care
  – casex takes z and x as don"t care

always @(op_code)
begin
  casez (op_code)
    4'b1???: alu_inst(op_code);
    4'b01??: mem_rd(op_code);
    4'b001?: mem_wr(op_code);
  endcase
end

If Statement

if (brach_flag)
  PC = brach_addr
else
  PC = next_addr;

always @(op_code)
begin
  if (op_code == 2'b00)
    cntr_sgn = 4'b1011;
  else if (op_code == 2'b01)
    cntr_sgn = 4'b1110;
  else
    cntr_sgn = 4'b0000;
end
Avoiding latches

- A latch is a storage device that stores a value without the use of a clock.
  - Latches are technology-specific and must be avoided in synchronous designs.
- To avoid latches adhere to coding guidelines
  - fully specify assignments or use a default assignment

```
input [1:0] sel;
reg [1:0] out_a, out_b;
always @ (*)
begin
  if (sel == 2'b00)
    begin
      out_a = 2'b01;
      out_b = 2'b10;
    end
  else
    out_a = 2'b01;
end
```

Out_b is not assigned any value under else, the synthesis tool will infer a latch.

Example

Implement the circuit

```
input [1:0] sel;
reg [1:0] out_a, out_b;
always @ (*)
begin
  if (sel == 2'b00)
    begin
      out_a = 2'b01;
      out_b = 2'b10;
    end
  else
    out_a = 2'b01;
end
```
Avoiding latches

• Use default assignment

```verilog
input [1:0] sel;
reg [1:0] out_a, out_b;
always @ (*)
begin
  out_a = 2'b00;
  out_b = 2'b00;
  if (sel==2'b00)
    begin
      out_a = 2'b01;
      out_b = 2'b10;
    end
  else
    out_a = 2'b01;
end
```

Avoiding latches

• Check all conditions

```verilog
input [1:0] sel;
reg [1:0] out_a, out_b;
always @ (*)
begin
  out_a = 2'b00;
  out_b = 2'b00;
  if (sel==2'b00)
    begin
      out_a = 2'b01;
      out_b = 2'b10;
    end
  else if (sel == 2'b01)
    out_a = 2'b01;
  else
    out_a = 2'b00;
end
```
Avoid latches

```verilog
always @*
begin
    out_a = 2'b00;
    out_b = 2'b00;
    case (sel)
        2'b00:
            begin
                out_a = 2'b01;
                out_b = 2'b10;
            end
        2'b01:
            out_a = 2'b01;
        default:
            out_a = 2'b00;
    endcase
end
```