EECS 151/251A (Verilog Tutorial)

Authors: Kevin Anderson and Vignesh Iyer (Spring 2024)
Rahul Kumar Yukio Miyasaka (Fall 2023)

Berkeley
UNIVERSITY OF CALIFORNIA

Contents

- HDL (Verilog vs System Verilog vs VHDL)
- How should think when writing Verilog?
- Wire vs Reg
  - Must declare signal before use
  - Inferring registers
  - Define multiple signals of the same type
- Literals
- Structural vs. Behavioral
- Case vs. If
- Continuous Statements vs. Always Block
- Modules + Instantiations
- Parameters (Generators)
- Named Ports
- Code Structure
- Finite State Machine
- Comments
- Synthesizable Constructs
- Testbenches
- Advice + Tips
  - Signal naming
  - Each signal definition on new line

Acknowledgement: Materials borrowed from previous semesters

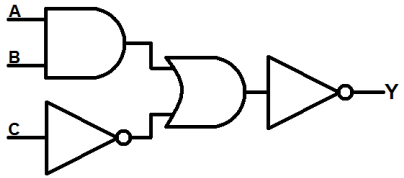
Hardware Description Language (HDL)

- What is HDL?
  - Originally designed to describe how digital circuits work
  - Morphed into a language which can be used to generate digital circuits
- HDLs: VHDL (1980's), Verilog (1984), System Verilog (2002)
  - **VHDL:** verbose, very explicit syntax, largely supported by CAD tools, used in Europe and government
  - **Verilog:** simpler, widely supported by tools, used in industry
  - **System Verilog:** built off Verilog, objects, structs, academia and industry
  - **We use Verilog in this class!**

Structural vs Behavioral

- Verilog written with explicit instantiations of primitives such as gates and transistors
- Mirrors a true digital circuit. Create gates and logic. Wire logic together

```
AND(a, b, ab_out);
NOT(c, c_not);
OR(ab_out, c_out, or_out);
NOT(c_out, y)
```



Digital circuit borrowed from: http://tinyurl.com/42zpe5yk

- Verilog written to describe the behavior of a digital circuit without explicit instantiation of gates
- CAD tools will infer the circuit which implements the logic described

```
y = ~((a&b) + ~c);
```

Think in Hardware!

- Writing HDL is **not** equivalent writing high-level programming language
- Hardware executes in parallel, so think in parallel
- Organize your code! And put comments!
- Readability over succinctness, verbosity for clarity
  - One liners look great, but if they obfuscate the intent then you have failed as a digital designer

The Basics

Nets

- Signals are called **nets** in Verilog
- Nets must be declared before use! Can declare multiple nets of same width simultaneously
- All nets have a data type and bit width
    - **If bit width not explicitly given, then assumed to be single bit**
- Two main net data types: **`wire`** and **`reg`**
- Specific rule for write net (to avoid multidriven net error) and read from limitlessly
- Can have values: 1, 0, X (undefined), Z (high-Z)

wire

- **Used in continuous assignment only**
- Conceptually, is always considered to be a wire connecting two logic elements

reg

- **Used in procedural assignments only** (i.e. always blocks, initial blocks, function, task)
- *Can* represent a register *or* a wire. It all depends on the context!

Nets (Examples)

wire

```
wire internal_1;          // 1-bit signal
wire internal_2;          // 1-bit signal
wire [2:0] internal_3;       // 3-bit signal

// Drive signal low
assign internal_1 = 1'b0;
assign internal_3 = 3'b101;

// Provide internal 1 as input
// and internal2 as output of NOT gate
// (Structural Verilog)
NOT(internal1, internal2);
```

reg

```
reg reg0;          // 1-bit signal
reg reg1, reg2;    // 1-bit signals; *multiple nets declared
reg [1:0] reg3;    // 2-bit signal

// This wont form registers!!
always @(reg0, reg1, reg2) begin
  reg2 <= reg0 & reg1;
end

// This will create registers!
always @(posedge clk) begin
  if (reg1 == 1'b1) begin
    reg2 <= 2'b10;
  end else begin
    reg2 <= 2'b11;
  end
end
```

Multi-Bit Nets

- Define width with *MSb* and *LSb;* width = (*MSb* – *LSb*) + 1
- Syntax (two options):
  1. *<net_type>* [*MSb* :*LSb*] *<net_name>*;
  2. *<net_type>* [*LSb* : *MSb*] *<net_name>*;
- Conventions
  - **Lowest index should be 0**
  - **Most digital designers prefer [*MSb* : *LSb*] style**
- **Index** extract a single bit (ex. tmp[0])
- Perform a **slice** to extract a range of bits a (ex. tmp[i:j]);
- When connecting multi-bit nets (esp. inputs or outputs), the bit-widths of the nets must match!

Examples:

```
wire [0:1] a;    // 2-bit wire; LSb style
reg [3:0]  b;    // 4-bit wire; MSb style

assign b[3] = 1'b1;    // Indexing
assign a = b[2:1];     // Slicing b and assigning to a
```

Note only one bit is assigned

Net Data Type

- Nets also have data type keyword: **`signed`** and **`unsigned`**
- **`unsigned`** signifies the net represents an unsigned value
- **`signed`** signifies the net represents a signed 2's complement value
- Use system task **`$signed`** and **`$unsigned`** to convert back and forth respectively
  - The underlying bit representation to not change, only the interpretation during arithmetic operations

Wire vs. Reg

- Rules for picking a **wire** or **reg** net type:
  - If a signal needs to be assigned inside an always block, it must be declared as a **reg**.
  - If a signal is used in a continuous assignment statement, it must be declared as a **wire**.
  - Module input and output ports are implicitly given **wire** type; if any output ports are assigned in an **`always`** block, they must be explicitly declared as reg

- How to know if a net represents a register or a wire?
  - A **wire** net always represents a combinational link
  - A **reg** net represents a wire *if* it is assigned in an `always @(*)` block
  - A **reg** net *typically* represents a register *if* it is assigned in an `always @(posedge/negedge clock)` block (tool synthesis process determines if **reg** net becomes a register)

Multidimensional Arrays

- Multidimensional arrays are possible, but **should be used sparingly if at all**
- **Ports should not be 2D arrays!**
- **Most common is 2D array**
  - If type **reg**, usually means a memory
  - If type **wire**, usually means a collection of buses
- Syntax: (**reg/wire)** [*ELEMENT_WIDTH-1:0*] <name> [*DEPTH*-1:0];
  - Example: `reg [3:0] mem [1023:0];  // 1024 entry memory with 4-bit entries`

EECS 151/251A Specific (John W Style)

- The **reg** net example from slide 5 uses "inferred registers"; the synthesis tool *infers* whether the net should be a register.
- Synthesis tools define specific constructions to explicitly specify register creation
- In this class, we provide a register library *EECS151.v* follow the convention for our synthesis tool
- Whenever you need a register, you have to instantiate one from the library

Specification:

- Stores an *N*-bit value
- Outputs the value as `q`
- At each positive edge of `clk`,
  - Resets to INIT if `rst`
  - Updates to `d` if `!rst & ce` (clock enable)
  - Hold the value otherwise

```
module REGISTER_R_CE(q, d, rst, ce, clk);
  parameter N = 1;
  parameter INIT = {N{1'b0}};
  output reg [N-1:0] q;
  input [N-1:0]           d;
  input                   rst, ce, clk;
  initial q = INIT;
  always @(posedge clk)
    if (rst) q <= INIT;
    else if (ce) q <= d;
endmodule
```

Literals

- Verilog defines a particular way of specifying literals
  - Syntax: [*bit width*]'[*radix*][*literal*]
- Radix can be
  - b (binary)
  - d (decimal)
  - h (hexadecimal)
  - o (octal)
- Examples: 1’b1, 3’b101, 6’d6, 4’hF, 5’o2
- It is critical to match bit widths for operators and module connections, do not ignore these warnings from the tools

Literals

- Literals *can* be defined without specifying width and radix
  - **This is frowned upon**; verbosity for clarity
  - Tool will assume decimal radix and infer width (truncate or extend as necessary)
  - **Ex:** `assign a = 1;`
- A convenient shorthand is to not specify the width
  - Tool will infer width of signal, truncate and extend as necessary
  - This is **very** convenient for certain cases like zeroing out a register
  - **Ex:**

```
wire [3:0] b;
assign b = 'b0;  // Set all bits to zero
```

If-Else

- Syntax: if <*condition*> begin end else begin end
- **if-else** statements represent a mux in hardware
- Be careful nesting! This creates long delay in chain of muxes

Example: if without else

```
if (b == 1'b1) begin
  c = 1'b1;
end
```

Example: if-else

```
if (a == 1'b1) begin
  b = 1'b1;
else
  b = 1'b0;
end
```

Example: if-else if with nested if

```
if (c === 1'b1) begin
  d = 2'b00;
end else if (b == 1'b0) begin
  d = 2'b10;
end else begin
  if(a == 1'b1) begin
    d = 2'b10
  end

  d = 2'b11;
end
```

Case

- Syntax: case (<*condition*>) begin endcase
- **case** statements represent a decoder
- Select a condition by patterning matching value of an input,
- Ensure every input combination has an assignment
  - Having a default case ensures this

```
case (sel)
  2'b00: out = 2'b10;
  2'b01: out = 2'b11;
  2'b10: out = 2'b00;
  2'b11: out = 2'b01;
  default : out = 2'b00;
endcase
```

If-Else vs Case

- **`if-else`** statements with many conditions or nested **if-else** statements will likley synthesize to a chain of multiplexers
  - This leads to bad timing. Signal must propagate through series of muxes
- A **`case`** statement inherently represents a decoder
- Decoders are more efficient as far as timing, therefore favor the use of **`case`** statements over long chains of **`if-else`** statements
  - Plus it looks nicer ☺

Procedural Assignment

- Procedural assignments are for “triggered” events (that happens if this event occurs)
  - **`always`** blocks
  - **`initial`** blocks
  - **`function`**
  - **`tasks`**
  - Multiline blocks have enclosures **`begin`** and **`end`**
- Only **`reg`** type signals can be used in procedural assignment
- **You cannot assign the same `reg` net in different procedural blocks!**
- Use procedural assignments for FSMs, counters, or succinct combinational logic.

always @(*) Block

- Syntax: **always** @(<*sensitivity list*>)
- Can represent *both* sequential and combinational logic
- Must include begin and end keyword for multi-line blocks
- The sensitivity list ”trigger” execution of the always block when any of the included nets change. Sensitivity list must include all nets on RHS of assignments
  - **Must include all nets involve in logical operation or assignments**
  - **Best practice is to use wildcard ‘*’ instead of listing each individual signal.** Tools interpret this as all nets involved in operations or assignments
  - Another common sensitivity list is **always** *@(posedge clk)*

Example:

```
wire a, b, s;
reg out;

always @(*) begin
  if (s) begin
    out = a;
  end else begin
    out = b;
  end
end
```

always @(*) Block: Blocking vs Non-blocking Assignments

- **This is fundamental to Verilog!**
- There are two types of assignments within procedural blocks:
  - Blocking Assignments (‘=‘) – assignments are evaluated sequential (i.e. assignments happen sequentially)
  - Non-blocking Assignments (‘<=‘) – assignments are evaluated in parallel (i.e. assignments occur simultaneously)
- DO NOT mix non-blocking and blocking assignment in **always** block. Error prone!
- Which to use? Depends on the intent
  - **Must use non-blocking for combinational logic**
  - For sequential logic, it depends on the code, but often non-blocking

initial Block

- Syntax: **`initial`**
- Code first executed, runs until timing event
  - Timing events use # (<*delay(integer)*>) or @(<*event*>)
- **Not synthesizable! Simulation only!**
- Used to set initial values of nets and sequence events in testbenches
- Multiple **`initial`** blocks allowed, but often confusing

Example:

```
initial begin
  rst = 1'b1;
  a = 1'b0;
  b = 1'b0;
  #4;
  rst = 1'b0;
  a = 1'b1;
end
```

function and task

- **functions** and **tasks** are NOT necessary for EECS151/251A, but common in Verilog
- Both **functions** and **tasks** represent callable repetitive logic, but have significant differences (some listed below):
  1. Tasks can have timing delay
  2. Tasks can have multiple outputs, functions can only have one output

```
task convert;
    input [7:0] adc_in;
    output [7:0] out;

    begin
        #4;
        out = (9/5) *( adc_in + 32)
    end
endtask
```

```
function  myfunction;
    input a, b, c, d;

    begin
        myfunction = ((a+b) + (c-d));
    end
endfunction
```

Examples borrowed (inspired) from here

Continuous Assignment

- Continuous assignments use the **assign** statements
  - Example: `assign internal_1 = 1'b0;`
- Nets with continuous assignments always driven by the given or expression
- Preferred/primary way to represent combinational logic
- Only **wire** type signals can be used in continuous assignments

Assign Statement

- **`assign` statements cannot be used inside procedural blocks**
- Wires can be assigned to logic equations, other wires, or operations on other wires
- The LHS of the assign statement must be a wire, and cannot be an input wire
- The RHS of the assign statement can be any expression created from Verilog operators and wires

```
wire [0:1] a;    // 2-bit wire; LSb style
reg [3:0]  b;    // 4-bit wire; MSb style

assign b[3] = 1'b1;    // Indexing
assign a = b[2:1];     // Slicing b and assigning to a
```

Conditional (Ternary) Operator

- Syntax: <*condition*> ? <*expression if condition is true*> : <*expression if condition is false*>
- Can be used in procedural and continuous assignments (more often in assign statements)
- Can be nested to create chain of if-else logic
- Example 1: a circuit that saturates at 10

```
assign out = a > 10 ? 10 : a;
```

- Example 2: comparator circuit which outputs: *0* if (*a == b)*, *1* if input (*a < b*), or *2* if input (*a > b*).

```
assign c = (a == b) ? 2'd0 : ((a < b) ? 2'd1 : 2'd2);
```

Operators

- Verilog contains operators that can be used to perform arithmetic, form logic expression, perform reductions/shifts, and check equality between signals.

Operator Type	Symbol	Operation Performed
Arithmetic	+	Add
	-	Subtract
	*	Multiply
	/	Divide
	%	Modulus
Logical	!	Logical negation
	&&	Logical and
	||	Logical or

Operator Type	Symbol	Operation Performed
Shift	<<	Shift left logical
	>>	Shift right logical
	<<<	Arithmetic left shift
	>>>	Arithmetic right shift
Concatenation	{}	Join bits
Replication	{{}}	Duplicate bits
Indexing/Slicing	[MSB:LSB]	Select bits

Operators (Examples)

- An example using multiple operators

```
wire [7:0] d;
wire [31:0] e;
wire [31:0] f;
wire [31:0] cnt, nxt_cnt;

// A counter
REGISTER_R_CE cnt (.clk(clk), .rst(rst), .ce(1'b1), .d(nxt_cnt), .q(cnt))

assign nxt_cnt = cnt + 1;          // Addition
assign f = {d, e[23:0]};           // Concatenation + Slicing
assign f = { 32{d[5]} };           // Replication + Indexing
assign e[31:24] = e[15:8] << 1;    // Left shift
assign d[7] = (e[1] == 1) && (d[4] == 1) ? 1'b1 : 1'b0; // Logical operators
```

Verilog Modules

- Modules are units of hardware that perform a specific function
- Modules form a design hierarchy
- A module declaration includes parameter declaration, port declaration, and code to implement the desired functionality.
- **Module ≠ Class. Module ≠ Function**
- **Good Convention: One module per file and the module name must match the file name!**

Verilog Modules

- Signals declared inside module are **internal only**. Only values on ports are visible to the outside world
- Parameters are module-level variables that can be overwritten during instantiation
  - This provides flexibility, most common modules are parameterized; parameters are the basis of generators
  - Usually **integer** types
  - Example: `parameter RAM_DEPTH = 32;`
- Local parameters are constants; like parameters but **cannot be overwritten**
  - **Often used to declare FSM states**
  - Example: `localparam STATE_0 = 0;`
- Instantiation is creating a unique instance of a module
  - Modules can be instantiated in another module
  - **Named ports** are used during instantiation so order during instantiation does not matter

Verilog Modules

- Ports allow inter-module communication or I/O to outside world
- Ports for a module are declared in the port declaration
- Each port has a name, type, and width (**input**, **output**, or **inout)**
  - Names cannot be duplicated
  - Ports without explicit type are assumed to be **wires**
  - Ports without explicit width are assumed to be a **single bit**
  - Ordering of port list does not matter; conceptually related ports often grouped together (i.e. ports connecting to the same module are consecutively listed)

Verilog Modules

- General instantiation structure:

  module *<module_name> <instance_name> [<parameters (optional)>] <port_list>*

  <internal signals & logic>

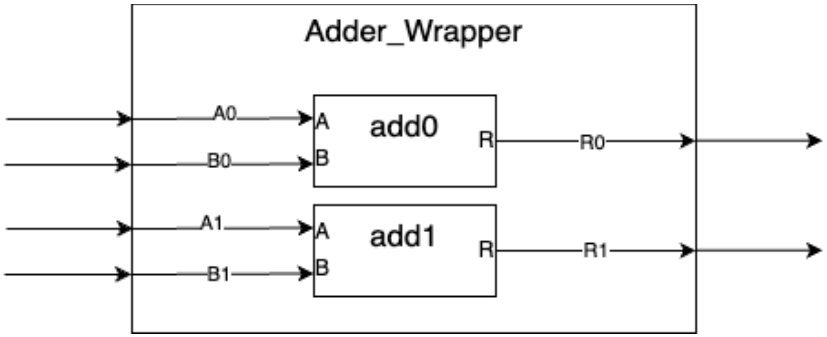
  endmodule

```
module Adder_Wrapper(
  input [N-1:0] A,
  input [N-1:0] B,
  output[N:0]   R);

  parameter N;

  Adder #(8) add0 ( .A(A),
                    .B(B),
                    .R(R));

endmodule
```

```
// Instantiation without named ports
Adder_Wrapper add_wrap1 (A ,B ,R);

// Instantiation with named ports
Adder_Wrapper add_wrap0 (.A(A), .B(B),.R(R));
```

Verilog Modules

- Every Verilog design has a top-level module which sits at the highest level of the design hierarchy
- The top-level module defines the I/O for the entire digital system
- All the modules in your design reside inside the top-level module




```
module Adder_Wrapper2(
  input  [N-1:0] A0,
  input  [N-1:0] B0,
  output [N:0]   R0,
  input  [N-1:0] A1,
  input  [N-1:0] B1,
  output [N:0]   R1);


  parameter N;

  Adder #(8) add0 ( .A(A0),
                    .B(B0),
                    .R(R0));

  Adder #(8) add1 ( .A(A1),
                    .B(B1),
                    .R(R1));

endmodule
```

Implementation 1:

```
module Adder(A, B, R);

  // Parameter
  parameter N = 4;

  // Ports
  input [N-1:0] A;
  input [N-1:0] B;
  output [N:0] R;

  // Signals
  wire [N:0] C;


  genvar i;
  generate
    for (i=0; i<N; i=i+1) begin
      FullAdder add(.a(A[i],
                    .b(B[i]),
                    .ci(C[i]),
                    .co(C[i+1]),
                    .r(R[i])));
    end
  end generate

  // Signal Assignment
  assign C[0] = 1'b0;
  assign R[N] = C[N];

endmodule
```

Port List

Parameter Declaration

Port Type & Width

Internal Signals

Module Instantiation

Continuous Signal Assignments

Implementation 2:

```
module Adder
  #( parameter N = 4)
  ( input [N-1:0] A,
    input [N-1:0] B,
    output[N:0]   R);

  // Signals
  wire [N:0] C;

  genvar i;
  generate
    for (i=0; i<N; i=i+1) begin
      FullAdder add(.a(A[i],
                    .b(B[i]),
                    .ci(C[i]),
                    .co(C[i+1]),
                    .r(R[i])));
    end
  end generate

  // Signal Assignment
  assign C[0] = 1'b0;
  assign R[N] = C[N];

endmodule
```

Both implementations are valid and equivalent!

Important Topics

Finite State Machines (FSM)

- FSMs orchestrate a sequence of events throughout *time*
  - Must use registers (at minimum a register to track the state)
- State diagrams visualize describe FSMs
  - States are typically gray coded
  - Arrows between state represents transition
    - Transitions without conditions are taken regardless
    - Transitions with conditions are only taken if condition is met
- Two defined types (difference is when the output changes)
  - Moore: the output is dependent solely on the state
  - Mealy: the output is dependent the input and the state
  - **In practice, many FSMs are a mixture of both**

Finite State Machines (FSM)

- States are created as **`localparam`** or with **`define`** statements
  - Some engineers like define statements because they use a single header file with constants used in the design
- Two styles for FSM in HDL (both are functional equivalent and reduce to similar logic)
  1. Two process (one sequential, one combinational)
  2. One process containing both sequential and combinational logic

Finite State Ma

- Moore FSM
- Two process style

```
/*
 * This module is a Moore FSM which style where sequential
 * and combinational logic are keep separate. A single register
 * represents the sequential logic. The always block is completely
 * combinational using blocking statements. This was common back in
 * the day for clarity and to reduce ambiguity for the synthesis tool.
 * CAD tools are much improved and there is no advantage to
 * writing FSMs like this, it is purely a stylistic decision.
 */
module power_on_seq_moore(
  input clk,
  input rst,
  //
  input pwr_on,
  input pwr_off,
  //
  output rail_1,
  output rail_2,
  output rail_3);

  // Constants
  // State Variables
  localparam IDLE=0;
  localparam WARM_UP=1;
  localparam VDD_ON=3;
  localparam PWR_DOWN=2;


  // Signals
  reg [1:0] nxt_state;
  wire [1:0] state;

  // Instantiations
  REGISTER_R #(2) state_reg(.clk(clk),
                            .rst(rst),
                            .ce(1'b1),
                            .d(nxt_state),
                            .q(state));


  always @(*) begin
    rail_1 = 1'b0;
    rail_2 = 1'b0;
    rail_3 = 1'b0;
    nxt_state = IDLE;

    case (state)
      // Idle state
      IDLE: begin
              if (pwr_on == 1'b1) begin
                nxt_state = WARM_UP;
              end
```

```
            end

      // One cycle wait state
      WARM_UP: nxt_state = VDD_ON;

      // Bring up Rails
      VDD_ON: begin
                if (pwr_off == 1'b1) begin
                  nxt_state = PWR_DOWN;
                end

              rail_1 = 1'b1;
              rail_2 = 1'b1;
              rail_3 = 1'b1;
            end

      // Bring down rails
      PWR_DOWN: begin
                  nxt_state = IDLE
                  rail_1 = 1'b0;
                  rail_2 = 1'b0;
                  rail_3 = 1'b0;
                end

      default : begin
                  nxt_state = IDLE;
                  rail_1 = 1'b0;
                  rail_2 = 1'b0;
                  rail_3 = 1'b0;
                end
    endcase
  end
endmodule
```

Finite State Ma

- Mealy FSM
- Single process style

```
/*
 * This module is a mealy FSM which style where there is a
 * single clocked process containing both sequential and
 * combinational. Note this style uses inferred registers
 * which is not allowed in the course. This is for
 * educational purposes only.
 */
module power_on_seq_mealy(
  input clk,
  input rst,
  //
  input pwr_on,
  input pwr_off,
  //
  output rail_1,
  output rail_2,
  output rail_3);

  // Constants
  // State Variables
  localparam IDLE=0;
  localparam WARM_UP=1;
  localparam VDD_ON=3;


  // Signals
  reg [1:0] state;


  always @(posedge clk) begin
    if (rst == 1'b1) begin
      state  = IDLE
      rail_1 = 1'b0;
      rail_2 = 1'b0;
      rail_3 = 1'b0;
    end else begin

      case (state)
        // Idle state
        IDLE: begin
                if (pwr_on == 1'b1) begin
                  state = WARM_UP;
                end
              end

        // One cycle wait state
        WARM_UP:  begin
                    state = VDD_ON;
                    rail_1 = 1'b1;
                    rail_2 = 1'b1;
                    rail_3 = 1'b1;
                  end

        // Bring up Rails
        VDD_ON: begin
                  if (pwr_off == 1'b1) begin
                    state = IDLE;
                  end

                  // Bring down rails
                  rail_1 = 1'b0;
                  rail_2 = 1'b0;
                  rail_3 = 1'b0;
                end


        default : begin
                    state = IDLE;
                    rail_1 = 1'b0;
                    rail_2 = 1'b0;
                    rail_3 = 1'b0;
                  end
      endcase
    end
  end
endmodule
```

Multiple Net Assignments

- Cannot have multiple continuous assignments for same **wire** net

```
assign a = b;
assign a = c;
```

- Can assign different values to **reg** net at different points in procedural block. Last assignment taken.

```
// a will be set to c
always @(*) begin
  a = b;
  a = c;
end
```

- Useful to handle complex conditions

```
// out if overwritten to a if s == 1'b1
always @(*) begin
  out = b;
  if (s) begin
    out = a;
  end
end
```

Loops

- **`for`** and **`while`** loops exist but only to be used in testbenches
  - Loops don't exist in hardware
  - Other ways to iterate in hardware (e.g. FSM or counter)
- **`for`** loops more common (and safer) than **`while`** loops

```
for (int i = 0; i < 10; i=i+1) begin
  $display("Count: %d", i);
  #4;
end
```

```
while (valid) begin
  $display("Valid is asserted");
  #1;
end
```

Generate Statements

- **generate** statements allows you to create copies the same hardware
- Can you use **generate** on **for** loops, **if** statement, and **case** statement
- **generate** statements are not loops
- Must create a **genvar** variable to use **generate for**

generate for

```
genvar i;

generate
  for (i=0; I < 16; i = i + 1) begin
    adder full_add_x( .a(a[i]),
                      .b(b[i]),
                      .c(c[i]));
  end
endgenerate
```

generate if

```
generate
  if (HALF_ADDER) begin
    adder hadd_x( .a(a[i]),
                  .b(b[i]),
                  .c(c[i]));
  else
    adder full_add_x( .a(a[i]),
                      .b(b[i]),
                      .c(c[i]));
  end
endgenerate
```

generate case

```
wire [2:0] a;

generate
  case (sel) begin
    0: a <= 3'b001;
    1: a <= 3'b011;
    2: a <= 3'b010;
    3: a <= 3'b101;
    default: a <= 3'b0;
  end
endgenerate
```

Generators

- "Generator" = parameterized Verilog module
- Typically, uses **generate** statements
- **generate for** and **generate if** are most common
- Example: A shift register with a `DEPTH` and `BIT_WIDTH` parameters

Example:

```
module naryand (in, out);
  parameter N = 1;

  input [N-1:0] in;
  output out;

  wire [N-1:0] tmp;
  buf(tmp[0], in[0]);
  buf(out, tmp[N-1]);
  genvar i;
  generate
    for(i = 1; i < N; i = i + 1) begin : ands
      and(tmp[i], in[i], tmp[i-1]);
    end
  endgenerate
endmodule

// Instantiation of generator
wire [4:0] f;
wire g;
naryand #(.N(5)) and5(.in(f), .out(g));
```

Inferred Latches

- A latch is a combinational circuit that does not change with clock
- Latches are created two ways:
  1. **no default case**
  2. **missing net in sensitivity list**
- Your design should not contain latches unless intended
- Unless for specific cases, latches are frowned upon
  - No sense of timing
  - Prone to bugs
  - The realized hardware might not even work consistently

Example:

```
module infer_latch (
  input a,
  input b,
  output reg out);

  always @(*) begin
    case ({a, b})
      2'b01: out = 1;
      2'b10: out = 1;
    endcase
  end
endmodule
```

Explanation: If input `2'b10` or `2'b01` happen, then the output is 1 forever

Interfaces

- Definition: ports at boundary of module to communicate to another module
- Technically, the ports of another module are an “interface”
- However, “interface” typically implies an industry standard protocol
    - Examples: Ready-Valid, AXI-Interface APB-Interface

Common Structures

- FIFO*
- Edge-Detector
- Shift Register*
- Counter*
- Accumulator*
- Pipeline*
- Barrel Shifter
- Carry-Save Adder (CSA)
- Booth Multiplier

* means know how to create this structure and how they are used

Synthesizable Verilog

- Synthesis is the process of transforming user written Verilog into an optimized Verilog **netlist** (structural) performed by a CAD tool (ex Cadence *Genus*)
- **Not all Verilog is synthesizable!**

Structure	Use	Alternative
Initial value of nets (ex. wire clk = 1'b0)	Sets initial value of nets	Use a reset condition in a procedural block
Initial block	Testbenches and simulation	Use a reset condition in a procedural block
Timing Delays and Events	Testbenches and behavioral simulations	Generate a clock and count cycles
Loops	Testbenches and behavioral simulations	A FSM
(Some) System Tasks ($finish, $display, $fork, $join, etc)	Ease of use	N/A

Tips/Advice

- Think before you code: *“How would I implement this in hardware?”*
- Need help? Ask for it
- If it looks crazy, then there is typical a better way
- Every register should have a reset
- Always define multibit nets in [MSb:0] format
- DO NOT add long chains of ternary operators
- **`generate`** statements are overrated. Use only when necessary
- Waveforms are better than many `$display` calls
- Active high reset for FPGAs, active low for ASIC (power consumption)

Tips/Advice

- One port per line
- One net declaration per line
- Timing, timing, timing…timing*
    - If I present a value to a flip-flop input at the clock rising edge, then the flip flop output will not be that value *until the next cycle*

Code Organization

- Consistent indents or else…
- Put block comments
  - At the top of the file list a description of the module or logic in the file
  - Some structures are common, others are not. For structures which are not put a block comment explaining the intent (ex. accessing a range of bits in a multibit net which represents a field, pipeline stages)
- Verilog has little constraint on placement of declarations and definitions. Therefore, it is your responsibility to enforce code structure. **Code must be logically organized!**

Common HDL Organization Styles

- Group everything related to perform subfunction (net declaration co-located to procedural block or continuous assign)

```
wire wire0;
wire wire1;
reg [1:0] reg0;
wire wire2;
wire [3:0] wire3;
```

```
always @(wire0, wire1) begin
    reg0 <= {~wire1, wire0};
end
```

...

```
wire wire2;
wire [3:0] wire3;
```

```
assign wire2 = 1'b1;
assign wire3 = 4'd5'
```

- Group code by kind: nets, function, instantiations, etc

```
wire wire0;
wire wire1;
reg [1:0] reg0;
wire wire2;
wire [3:0] wire3;
```

```
always @(wire0, wire1) begin
    reg0 <= {~wire1, wire0};
end
```

```
assign wire2 = 1'b1;
assign wire3 = 4'd5'
```

Kevin's Biased Opinion

- Group code by kind: nets, function, instantiations, etc

```
wire wire0;
wire wire1;
reg [1:0] reg0;
wire wire2;
wire [3:0] wire3;
```

```
always @(wire0, wire1) begin
    reg0 <= {~wire1, wire0};
end
```

```
assign wire2 = 1'b1;
assign wire3 = 4'd5'
```

```
module template(
  input clk,
  input rst,
  ...);

  // Signals Declaration

  // Functions

  // Module Instantiation

  // FSM and Procedural

  // Signal Assignment


endmodule
```

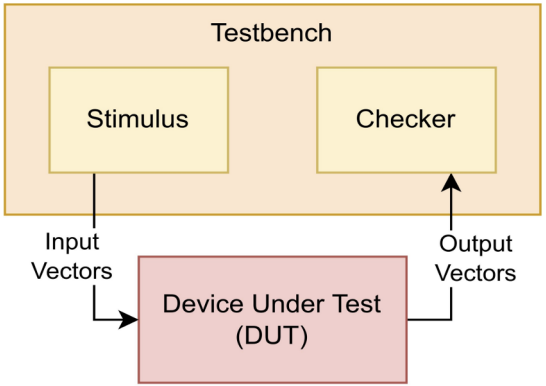
Simulation

Simulation

- Simulation is more important in hardware than in software because once it's made you can't change it
- There are commercial and open-source tools to perform Verilog simulation
- Different forms of simulation: Unit Testing, Full Design test, Verification
    - In this course, we focus on unit and full design testing. This are not the exhaustive test, but give some confidence in the design and therefore a necessary skill

Testbench

- Testbenches are Verilog modules which instantiate your DUT to drive its inputs and verify its outputs (functional testing)
- **Testbenches have no inputs or outputs**
- Things to have:
  1. Simulated clock
  2. **initial** block for driving input for given test cases
  3. Instantiated DUT
  4. Process to verify outputs or print to console



Testbench

- Clock:
  - Define **localparam** for clock period

```
reg clk = 0; // Create clock; initialize as zero
always #(CLK_PERIOD/2) clk = ~clk;  // Toggle clock
```

- Nets:
  - Create **reg** nets for DUT inputs, connect to DUT, and drive them in **initial** block
  - Create **wire** nets for DUT output, connect to DUT, and verify outputs in process
- Initial block:
  - Always assert reset first
  - Begin with system calls (if necessary) to record waveforms (ex. **$dumpfile**("dump.vcd"); **$dumpvars;)**
  - If DUT is synchronous, driving inputs of negative edge ensures DUT captures data of rising edge
  - Have test cases separated by a delay or timing event
  - End with **$finish** to end simulation

Testbench

- Verify Outputs
  - Create **`always`** block (clocked, or triggered off DUT output valid)
  - Ensure check is synchronized with DUT
    - Method 1: Verify output on negative following output present at port
    - Method 2: Use combinational **`always`** block with DUT valid signal
    - Method 3: Check output immediate next cycle (output will be present at rising edge)
  - Either hardcode expected result, generate expected results dynamically, or read from file (**`$readmemh`**)

Testbench Template

- No ports
- Timescale declared at the top of the file
- `clk` is simulated with **`localparam`** and is initialized
- **`initial`** block to execute sequence of test cases
  - `rst` is asserted first for a couple of cycles
  - Ends with a **$finish** system call
  - Timing event: #(4*CLK_PERIOD)
  - Separate test cases with timing events for synchronization
- Clocked process (`always` block) to verify DUT outputs (can be combinational if DUT outputs signal indicate output is valid)

```
`timescale 1ns/10ps

module template_tb();

  ////////////////////////////////
  // Constants
  ////////////////////////////////
  localparam CLK_PERIOD = 10;


  ////////////////////////////////
  // Signals
  ////////////////////////////////
  reg clk;
  reg rst;


  ////////////////////////////////
  // DUT Instantiation
  ////////////////////////////////


  ////////////////////////////////
  // Process for inputs
  ////////////////////////////////
  initial begin
    clk <= 1'b0;
    rst <= 1'b1;
    #(4*CLK_PERIOD);    // Delay for 4 clock periods
    rst <= 1'b0;

    // TODO: DRIVE INPUTS HERE

    $finish;            // End simulation
  end


  ////////////////////////////////
  // Process to verify outputs
  ////////////////////////////////
  always @(posedge clk) begin
    // TODO: FILL IN
  end


  ////////////////////////////////
  // Signal Assignment
  ////////////////////////////////
  always #(CLK_PERIOD/2) clk = ~clk

endmodule
```

Printing Signals

- Use the **`$display`** system task: **`$display`**("format string", values);
  - Example: **`$display`**("in: %b, out: %b, expected: %b", in, out, expected);

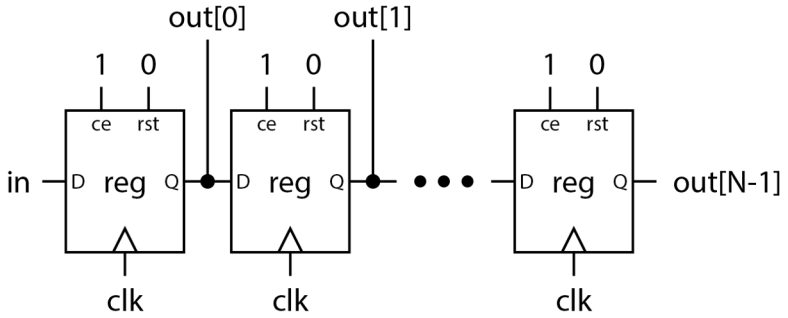
%d or %D	Decimal format
%b or %B	Binary format
%h or %H	Hexadecimal format
%o or %O	Octal format
%c or %C	ASCII character format
%v or %V	Net signal strength
%m or %M	Hierarchical name
%s or %S	As a string
%t or %T	Current time format

Examples

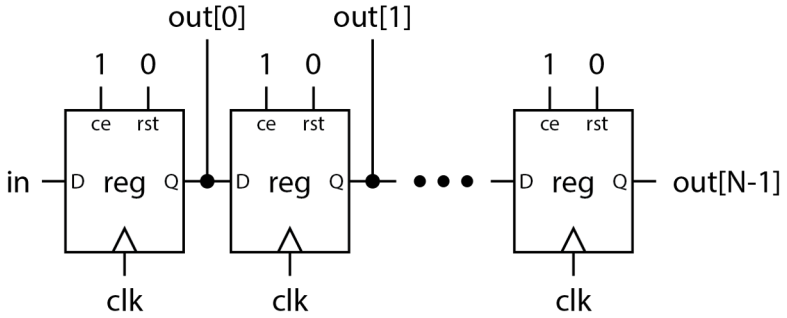
Example: 1-bit Shift-Register

Specification:

- A shift-register for single bit values
- Parameterized to be N registers deep
- Output is the value of each register



Example: 1-bit Shift-Register



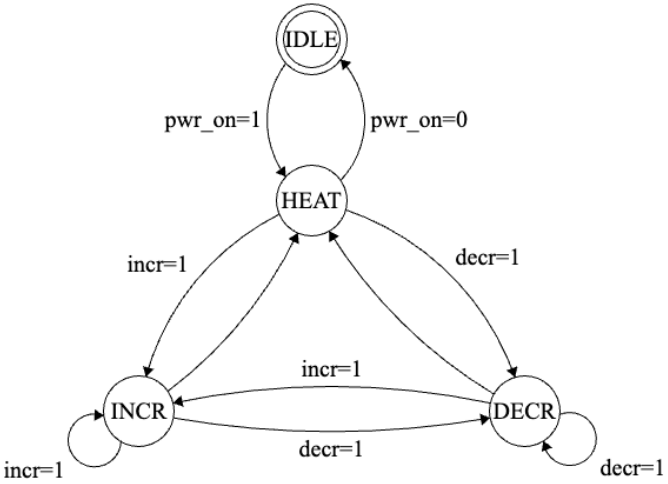
Verilog Implementation:

```
module shift_register # (parameter N = 1)
                        (input clk,
                         input in,
                         output [N-1:0] out);

   wire [N-1:0] tmp;

   REGISTER_R_CE #(.N(N))
      r(.q(out), .d(tmp), .rst(0), .ce(1), .clk(clk));

   assign tmp = {out, in};
endmodule
```

Example: Heater Moore FSM

Specification:

- A heater which blows hot air when turned on. The heater starts off, no hot air. It has three buttons which control the actions of the heater. These buttons are inputs to our FSM. A single bit represents whether the heater is blowing hot air is the output.
- Inputs:
  - *pwr_on* – turn heater on
  - *incr* – to increase temperature
  - *decr* – to decrease temperature
- Outputs
  - *air* – asserted if heater is on
- States:
  - *IDLE* – heater is off, not air is blowing
  - *HEAT* – heater is on, but not changing temperature
  - *INCR* – heater is on and increasing temperature by 1˚F
  - *DECR* – heater is on and decreasing temperature by 1˚F

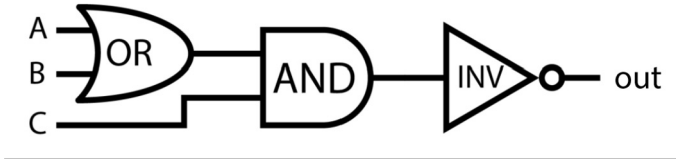


State diagram for our example FSM

Example: Heater Moore FSM

```
module heater(
  input clk,
  input rst,
  input pwr_on,
  input incr,
  input decr,
  output reg air);

  // Constants
  // States
  localparam IDLE = 0;
  localparam HEAT = 1;
  localparam INCR = 2;
  localparam DECR = 3;

  // Signals Declaration
  reg [1:0] nxt_state;
  wire [1:0] state;


  // Functions
  // N/A

  // Module Instantiation
  REGISTER_R #(2) fsm_st (.clk(clk),  // 2-bit state variable for FSM
                          .rst(rst),
                          .d(nxt_state),
                          .q(state));

  // FSM and Procedural
  always @(posedge clk) begin
    if(rst) begin
       nxt_state <= 'b0;
       air       <= 1'b0;
    end else begin
      case (fsm_st)

        IDLE: begin
                if (pwr_on == 1'b1) begin
                  nxt_state <= HEAT;
                end

                air <= 1'b0;
              end

        HEAT: begin
                if (pwr_on == 1'b0) begin
                  nxt_state <= IDLE;
                end else if (incr == 1'b1) begin
                  nxt_state <= INCR;
                end else if (decr == 1'b1) begin
                  nxt_state <= DECR;
                end else begin
                  nxt_state <= HEAT;
                end

                air <= 1'b1;
              end

        INCR: begin
                if (incr == 1'b1) begin
                  nxt_state <= INCR;
                end else if (decr == 1'b1) begin
                  nxt_state <= DECR;
                end else begin
                  nxt_state <= HEAT;
                end

                air <= 1'b1;
              end

        DECR: begin
                if (decr == 1'b1) begin
                  nxt_state <= DECR;
                end else if (incr == 1'b1) begin
                  nxt_state <= INCR;
                end else
                  nxt_state <= HEAT;
                end

                air <= 1'b1;
              end

        default:  begin
                    nxt_state <= IDLE;
                    air       <= 1'b0;
                  end
      endcase
    end
  end

  // Signal Assignment
  // N/A

endmodule
```

Example: OAI



EDA Playground (https://www.edaplayground.com/)

- Free web-based simulator
- Built-in waveform viewer
- HIGHLY recommended for homework 2!
- (Select Icarus Verilog 0.9.7 for simulator)

OAI design and testbench:

https://www.edaplayground.com/x/Egiz

Resources

- IEEE Standard for Verilog Hardware Description Language
- Finite State Machines
- EDA Playground