SAN JOSÉ STATE UNIVERSITY

EE178 Spring 2017
Lecture Module 2

Eric Crabill

Goals

- Review Verilog-HDL
  - Overview of language constructs
  - Modeling finite state machines
  - Synthesis considerations
  - FPGA considerations

- We will use Verilog-HDL as a tool to describe and model digital circuits for implementation in FPGA devices

Reference

- Popular hardware description languages
  - Verilog-HDL (not to be confused with Verilog-XL, a logic simulator program sold by Cadence)
  - VHDL (VHSIC-HDL, where VHSIC is Very High Speed Integrated Circuit)

- HDL Chip Design, by Douglas J. Smith
  - Covers Verilog-HDL and VHDL side by side
  - Shows many schematic implementations
  - Out of print…

Reference

- Verilog-HDL history
  - 1983 Gateway Design Automation released the language and a simulator product for it
  - 1989 Cadence acquired Gateway
  - 1990 Cadence separated the Verilog-HDL from their simulator product, Verilog-XL, releasing the language into the public domain
  - 1995 IEEE adopted 1364-1995 (Verilog-95)
  - 2001 IEEE ratified 1364-2001 (Verilog-2001)
  - 2005 IEEE ratified 1365-2005 (Verilog-2005)

Reference

- VHDL history
  - 1983 VHDL was developed under the VHSIC program of the Department of Defense
  - 1987 IEEE adopted 1076-1987 (VHDL-87)
  - 1993 IEEE ratified 1076-1993 (VHDL-93)
  - 2009 IEEE ratified 1076-2008 (VHDL-2008)

Let’s Fight!

- Which is “better” Verilog or VHDL?
  - Both are adequate for our purposes…
  - VHDL may be more powerful but very rigid
  - Verilog may be easier, but you can hang yourself if you are not careful...

- What you use may be dictated by company preference or government requirement

Usage

- Verilog may be used to model circuits and behaviors at various levels of abstraction:
  - Transistor LOW
  - Gate
  - Logic
  - Behavioral
  - Algorithmic HIGH

- For design with FPGA devices, transistor and gate level modeling is not appropriate

Data Values

- For our logic design purposes, we'll consider Verilog to have four different bit values:
  - 0, logic zero
  - 1, logic one
  - x, unknown
  - z, high impedance

Data Values

- When specifying values, whether single bit or multi-bit, use explicit syntax for clarity:
  - 4’d14 // 4-bit value, specified in decimal
  - 4’he // 4-bit value, specified in hex
  - 4’b1110 // 4-bit value, specified in binary
  - 4’b10xz // 4-bit value, with x and z, in binary

- The general syntax is:
  - {bit width}’{base}{value}

Data Types

- There are two main data types in Verilog
  - Wires
  - Regs

- Wires and regs may be declared as single bit or multi-bit (including two dimensional)

- Wires and regs may be declared as signed
  - A matter of interpretation, and helpful

Data Types

- ❖ Wires are “continuously assigned” values and do not “remember”, or store, information
- ❖ Wires may have zero, one, or more drivers assigning values
  - With 0 drivers, the wire is high impedance
  - With 1 driver, the wire takes the driven value
  - With multiple drivers, the simulator resolves the driven values into a single value for the wire
- ❖ When a driver changes the value it drives, the values for wires driven by it are re-evaluated
- ❖ This behaves much like an electrical wire...

Data Types

- Regs are “procedurally assigned” values and “remember”, or store, information until the next value assignment is made
  - This can be used to model combinational logic
  - This can be used to model sequential logic

- The name reg does **not** mean it is a register!
- A reg may be assigned by multiple processes

- Related data types include integer, real, time

Modules and Ports

- Consider a top level module declaration:

```
module testbench;
  // Top level modules do not have ports.
endmodule
```

- Consider a module declaration with ports:

```
module two_input_xor (
  input  wire in1,
  input  wire in2,
  output wire out
  );
  // We’ll add more later…
endmodule
```

Modules and Ports

❖ Ports may be of type {input, output, inout} and can also be multiple bits wide

```
module some_random_design (
  input  wire       fred,  // 1-bit input port
  input  wire [7:0] bob,   // 8-bit input port
  output wire       joe,   // 1-bit output port
  output wire [1:0] sam,   // 2-bit output port
  inout  wire       tom,   // 1-bit bidirectional
  inout  wire [3:0] ky     // 4-bit bidirectional
  );

  // Some design description would be here…

endmodule
```

Port and Data Types

- An input port can be driven from outside the module by a wire or a reg, but inside the module it can only drive a wire (implicit wire)
- An output port can be driven from inside the module by a wire or a reg, but outside the module it can only drive a wire (implicit wire)
- An inout port, on both sides of a module, may be driven by a wire, and drive a wire

Ports and Data Types

❖ Data type declaration syntax and examples:

```
module some_random_design (
  input  wire       fred,   // 1-bit input port
  input  wire [7:0] bob,    // 8-bit input port
  output wire       joe,    // 1-bit output port
  output wire [1:0] sam,    // 2-bit output port
  inout  wire       tom,    // 1-bit bidirectional
  inout  wire [3:0] ky      // 4-bit bidirectional
  );

  // In the above port declarations, note that you
  // can declare output ports to be of type reg.
  // Below are declarations of more regs and wires.

  wire       dork;          // 1-bit wire declaration
  wire [7:0] hoser;         // 8-bit wire declaration
  reg        state;         // 1-bit reg declaration
  reg  [7:0] lame;          // 8-bit reg declaration

endmodule
```

Instantiation

❖ Here is how you do it:

```
module testbench;
  wire    sig3;                  // wire driven by submodule
  reg     sig1, sig2;            // regs assigned by testbench

  two_input_xor my_xor (.in1(sig1), .in2(sig2), .out(sig3));

  // The format is <module> <instance_name> <port mapping>;
  // Inside the port mapping, there is a comma separated
  // list of .submodule_port(wire_or_reg_in_this_module) and
  // you include all ports of the submodule in this list.
endmodule

module two_input_xor (
  input  wire in1,
  input  wire in2,
  output wire out
  );
  // We'll add more later…
endmodule
```

Operators

- Operators in Verilog are similar to operators you find in other programming languages
- Operators may be used in both procedural and continuous assignments
- The following pages present them in order of evaluation precedence

Operators

- { } is used for concatenation

  Say you have two 1-bit data objects, sam and bob

  {sam, bob} is a 2-bit value from concatenation

- {{ }} is used for replication

  Say you have a 1-bit data object, ted

  {4{ted}} is a 4-bit value, ted replicated four times

Operators

- Unary operators:
  - ! Performs logical negation (test for non-zero)
  - ~ Performs bit-wise negation (complements)
  - & Performs unary reduction of logical AND
  - | Performs unary reduction of logical OR
  - ^ Performs unary reduction of logical XOR

Operators

- Dyadic arithmetic operators:
  - * Multiplication
  - / Division
  - % Modulus
  - + Addition
  - - Subtraction

Operators

- Dyadic logical shift operators:
  - << Shift left
  - >> Shift right

- Dyadic relational operators:
  - > Greater than
  - < Less than
  - >= Greater than or equal
  - <= Less than or equal

Operators

- Dyadic comparison operators:
  - == Equality operator (compares to z, x are invalid)
  - === Identity operator (compares to z, x are valid)
  - != Not equal
  - !== Not identical

Operators

- Dyadic binary bit-wise operators:
  - & Bit-wise logical AND
  - | Bit-wise logical OR
  - ^ Bit-wise logical XOR
  - ~^ Bit-wise logical XNOR

- Dyadic binary logical operators:
  - && Binary logical AND
  - || Binary logical OR

Operators

- Ternary operator for conditional selection:

  ? :

- May be used for description of multiplexers and three state logic

  sel ? value_if_sel_is_one : value_if_sel_is_zero

  oe ? driven_value : 1’bz

Continuous Assignment

- Continuous assignments are made with the assign statement

- assign LHS = RHS;
  - The left hand side, LHS, must be a wire
  - The right hand side, RHS, may be a wire, a reg, a constant, or expressions with operators using one or more wires, regs, and constants

- You model combinational logic with assign

Continuous Assignment

❖ Two examples to complete two_input_xor:

```
// example 1
assign out = in1 ^ in2;

// example 2
wire product1, product2;
assign product1 = in1 && !in2;      // could have done in
assign product2 = !in1 && in2;      // single assignment
assign out = product1 || product2;  // statement…
```

Continuous Assignment

- Two more examples to complete two_input_xor:

```
// example 3
assign out = (in1 != in2);

// example 4
assign out = in1 ? (!in2) : (in2);
```

- Please note – there are often many ways to do the same thing!

Continuous Assignment

- The value of the RHS is continuously driven onto the wire of the LHS
- Whenever elements in the RHS change, the simulator re-evaluates the result and updates the value driven on the LHS wire
- Values x and z are allowed and processed
- All assign statements operate concurrently

Procedural Assignment

- Procedural assignments operate concurrently and are preceded by event control
- Procedural assignments are done in block statements which start with “begin” and end with “end”
- Single assignments can omit begin and end

Procedural Assignment

❖ Syntax examples:

```
initial
begin
    // These procedural assignments are executed
    // one time at the beginning of the simulation.
end

always @(sensitivity list)
begin
    // These procedural assignments are executed
    // whenever the events in the sensitivity list
    // occur.
end
```

Procedural Assignment

- A sensitivity list is used to qualify when the block should be executed by providing a list of events which cause the execution to begin:
  - always @(a or b)   // any changes in a or b
  - always @(posedge a)   // a transitions from 0 to 1
  - always @(negedge a)   // a transitions from 1 to 0
  - always @*   // any changes in “inputs”

- You can model combinational and sequential logic using procedural assignments

Procedural Assignment

- Inside a block, two types of assignments exist:
  - LHS = RHS; // blocking assignment
  - LHS <=RHS; // non-blocking assignment
  - Do not mix them in a given block

- Assignment rules:
  - The left hand side, LHS, must be a reg
  - The right hand side, RHS, may be a wire, a reg, a constant, or expressions with operators using one or more wires, regs, and constants

Procedural Assignment

- From *HDL Chip Design*:

  A blocking procedural assignment must be executed before the procedural flow can pass to the subsequent statement. This means that any timing delay associated with such statements is related to the time at which the previous statements in the particular procedural block are executed.

  A non-blocking procedural assignment is scheduled to occur without blocking the procedural flow to subsequent statements. This means the timing in an assignment is relative to the absolute time at which the procedural block was triggered.

Procedural Assignment

- Do I use blocking or non-blocking assignments?
  - Blocking assignments are especially useful when describing combinational logic
    - You can “build up” complex logic expressions
    - Blocking assignments make your description evaluate in the order you described it
  - Non-blocking assignments are useful when describing sequential logic
    - At a clock or reset event, all flops change state at the same time, best modeled by non-blocking assignments

Procedural Assignment

- ❖ In procedural assignments, you may also use if-else and various types of case statements for conditional assignments
- ❖ You also can make use of additional timing control like wait, #delay, repeat, while, etc…
- ❖ While powerful and flexible, this grows confusing so let's look at some simple examples of using procedural assignments

Procedural Assignment

- Examples to complete two_input_xor using procedural assignment -- this assumes the module output, “out” is declared as type reg

```
// example 1
always @(in1 or in2)   // Note that all input terms
begin                  // are in sensitivity list!
    out = in1 ^ in2;   // Or equivalent expression...
end

// example 2
always @(in1 or in2) out = in1 ^ in2;
// no begin-end pair needed for single statements
```

Procedural Assignment

❖ More examples:

```
// example 3
always @* out = in1 ^ in2;
// use of wildcard prevents sensitivity list errors

// example 4
always @*
begin
    if (in1 == in2) out = 1’b0;   // very complex decision
    else out = 1’b1;              // logic is possible
end
```

Procedural Assignment

❖ More examples:

```
// example 5
always @*
begin
    case ({in2, in1})  // Concatenated 2-bit selector
      2'b01:   out = 1'b1;
      2'b10:   out = 1'b1;
      default: out = 1'b0;
    endcase
end
endmodule
```

Procedural Assignment

- Sequential logic example -- what is this?

```
module counter (
  input  wire       clk,
  input  wire       rst,
  input  wire       ce,
  output reg  [7:0] out
  );

  always @(posedge clk)
  begin
    if (rst) out <= 0;
    else if (ce) out <= out + 1;
  end

endmodule
```

Delay Control

- ❖ You can add delay to continuous assignments
- ❖ assign #delay LHS = RHS;
  - The #delay indicates a time delay in simulation time units; for example, #5 is a delay of five
  - This can be used to model physical delays of combinational logic
- ❖ The simulation time unit can be changed by the Verilog `timescale directive

Delay Control

- You can control the timing of assignments in procedural blocks in several ways:
  - Simple delays
    - #10;
    - #10 a = b;
  - Edge triggered timing control
    - @(a or b);
    - @(a or b) c = d;
    - @(posedge clk);
    - @(negedge clk) a <= b;

Delay Control

- You can control the timing of assignments in procedural blocks in several ways:
  - Level triggered timing control
    - wait (!reset);
    - wait (!reset) a = b;

Delay Control

❖ Generation of clock and resets in testbench:

```
reg rst, clk;
initial         // this happens once at time zero
begin
    rst = 1’b1; // starts off as asserted at time zero
    #100;       // wait for 100 time units
    rst = 1’b0; // deassert the rst signal
end
always          // this repeats forever
begin
    clk = 1’b0; // starts off as low at time zero
    #25;        // wait for half period
    clk = 1’b1; // clock goes high
    #25;        // wait for half period
end
```

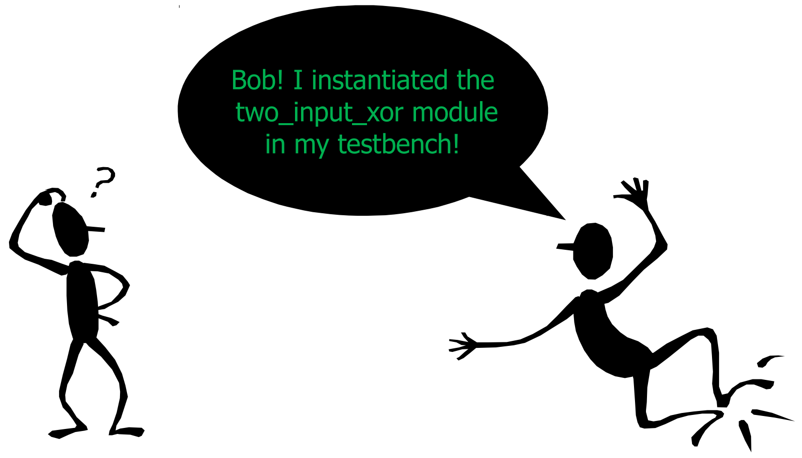
System Tasks

- The $ sign denotes Verilog system tasks, there are a large number of these, most useful being:
  - $display(“The value of a is %b”, a);
    - Used in procedural blocks for text output
    - The %b is the value format (binary, in this case…)
  - $finish;
    - Used to finish the simulation
    - Use when your stimulus and response testing is done
  - $stop;
    - Similar to $finish, but doesn’t exit simulation

Application

- The act of using a module within another module (as a sub-module) is called instantiation

- This permits hierarchical design and the re-use of modules


Bob! I instantiated the two_input_xor module in my testbench!
?

Application

- Useful hierarchy
  - A mux is something you can describe in one line
    - Usually don’t need a submodule for that, it’s basic
    - Doing so could detract from readability – why?
  - Just because it’s possible doesn’t mean it’s good

- Opinion on useful hierarchy
  - Driven by clean cuts of functionality or behavior
  - A module of only a few lines of code is a net loss
  - A module more than a few pages is difficult to comprehend for simulation and debug

Application

- For synthesis purposes, you will want to keep the circuit description in a separate module from the stimulus and response checking
  - The circuit can consist of any number of sub-modules
  - When you synthesize the circuit, you will only provide the synthesis tool with the modules that correspond to the actual design

Finite State Machines

- In *Real-time Object-oriented Modeling*, Bran Selic and Garth Gullekson view a state machine as:
  - A set of input events
  - A set of output events
  - A set of states
  - A function that maps states and input to output
  - A function that maps states and inputs to states
  - A description of the initial state

Finite State Machines

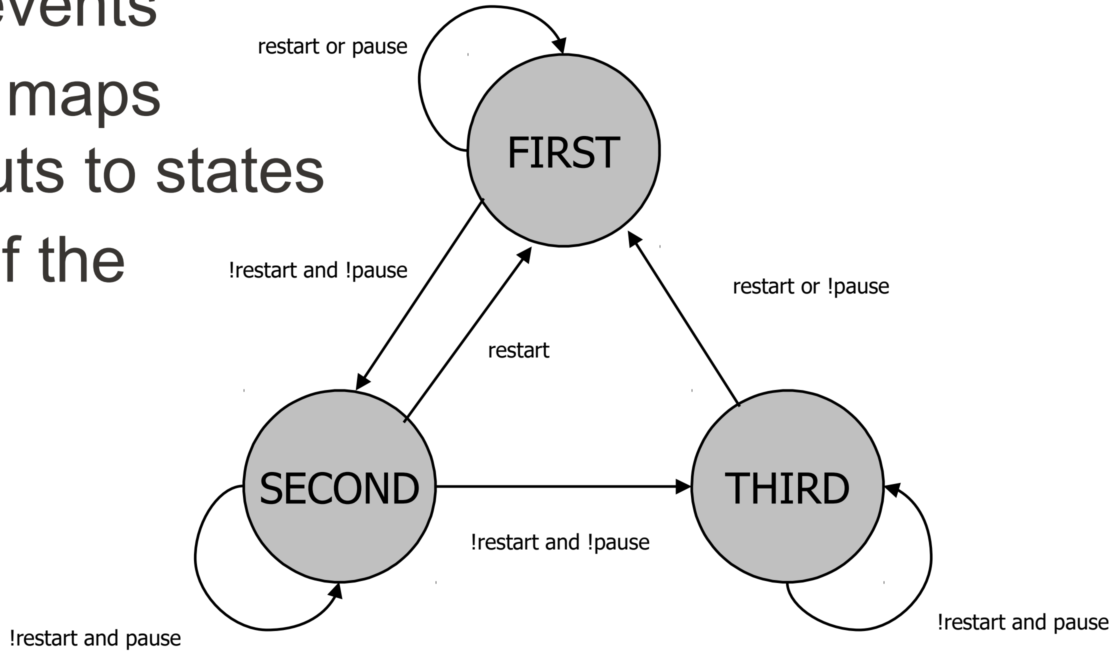
- A finite state machine is one that has a limited, or *finite*, number of states
- The machine state is described by a collection of state variables
- A finite state machine is an abstract concept, and may be implemented using a variety of techniques, including digital logic

Finite State Machines

- For an edge-triggered, synchronous FSM implemented in digital logic, consider:
  - A set of input events (input signals, including clock)
  - A set of output events (output signals)
  - A set of states (state variables are flip flops)
  - A function that maps states and input to output (this is the output logic)
  - A function that maps states and inputs to states (this is the next-state logic)
  - A description of the initial state (initial flip flop value)

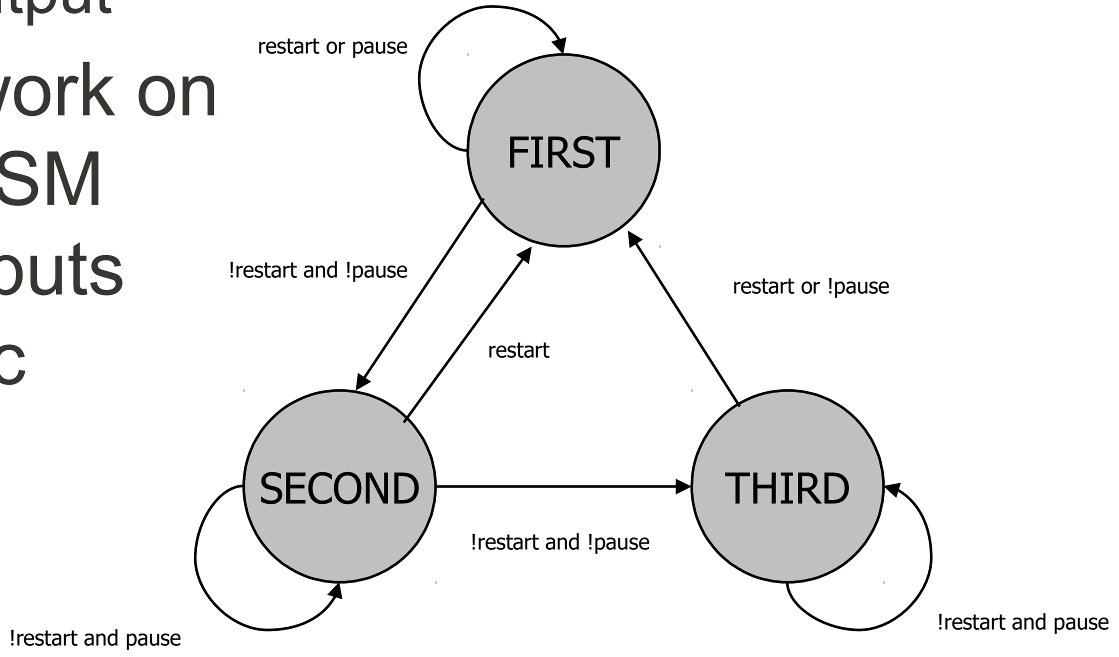
Finite State Machines

- Consider this edge-triggered, synchronous FSM to be implemented in digital logic:
  - A set of states
  - A set of input events
  - A function that maps states and inputs to states
  - A description of the initial state



Finite State Machines

- Things that are not shown (yet):
  - A set of output events
  - A function that maps states and input to output
- For now, let's work on modeling the FSM without the outputs and output logic



Finite State Machines

- The state variables must be able to represent at least three unique states for this FSM
  - A flip flop has two unique states
  - N flip flops can represent up 2^N unique states
  - How many flip flops are required for three states?
    - One flip flop is not enough
    - Two flip flops are minimally sufficient
    - More flip flops may be used, if desired

Finite State Machines

- Select a state encoding method:
  - Binary
  - Gray
  - Johnson
  - One Hot
  - Custom

State	Binary	Gray	Johnson	One Hot
0	3'b000	3'b000	4'b0000	8'b00000001
1	3'b001	3'b001	4'b0001	8'b00000010
2	3'b010	3'b011	4'b0011	8'b00000100
3	3'b011	3'b010	4'b0111	8'b00001000
4	3'b100	3'b110	4'b1111	8'b00010000
5	3'b101	3'b111	4'b1110	8'b00100000
6	3'b110	3'b101	4'b1100	8'b01000000
7	3'b111	3'b100	4'b1000	8'b10000000

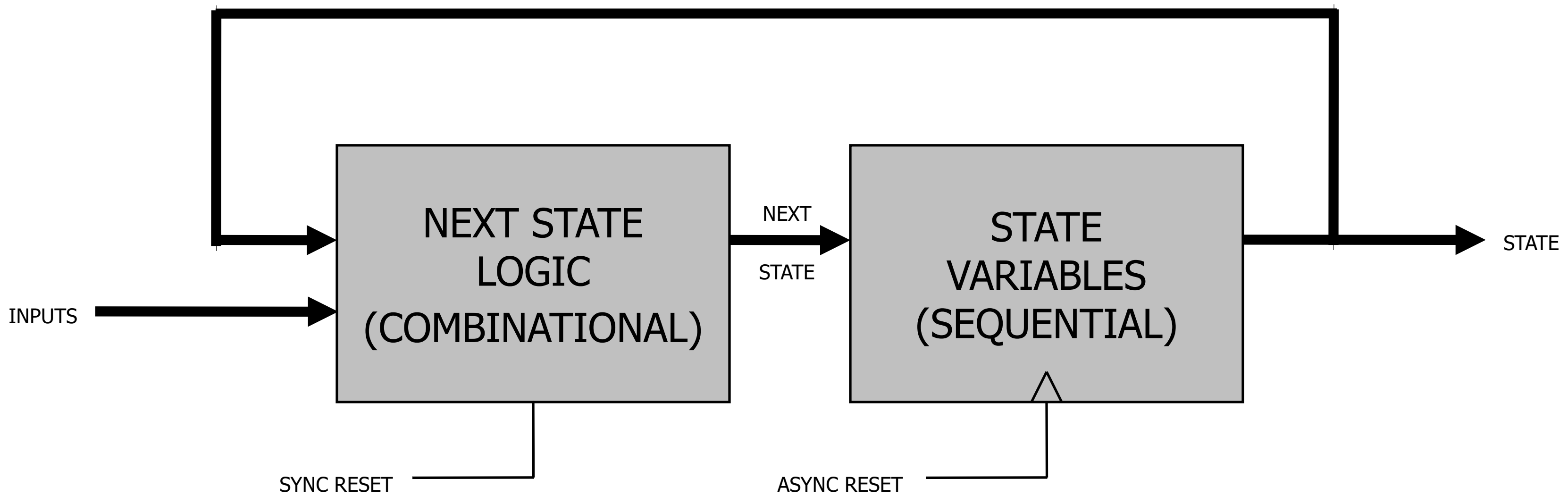
- Your encoding selection may require more than the minimally sufficient number of flip flops

Finite State Machines

- Describe the state variables in Verilog
- Provide a mechanism to force an initial state
- Describe a function that maps inputs and current state to a new, or next state
  - Literal transcription of excitation equations
  - Behavioral description using case, if-else, etc…
- Some additional things to consider:
  - Resets, synchronous or asynchronous?
  - Unused states (error, or no resets) and recovery

Finite State Machines

- Describe it in Verilog just like the diagram!
- I have selected a custom state encoding


NEXT STATE LOGIC (COMBINATIONAL)
NEXT STATE
STATE VARIABLES (SEQUENTIAL)
STATE
INPUTS
SYNC RESET
ASYNC RESET

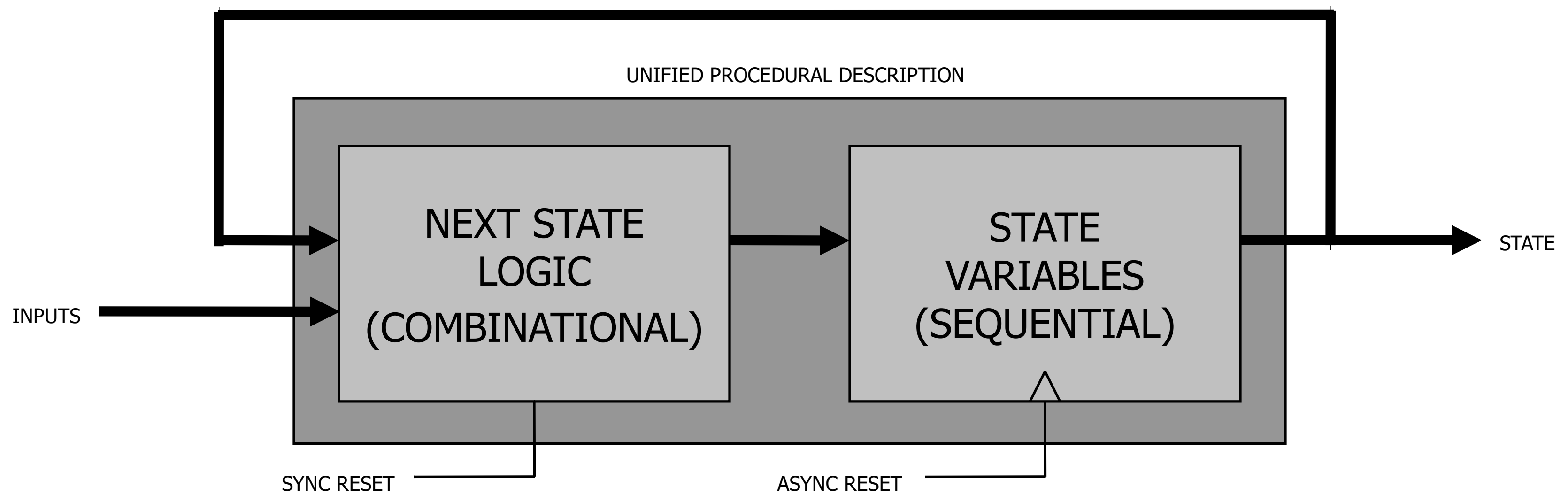
Finite State Machines

```
module fsm (
  input wire pause,
  input wire restart,
  input wire clk,
  input wire rst,
  output reg [1:0] state
  );

  reg [1:0] next_state;

  parameter [1:0] FIRST  = 2'b11;
  parameter [1:0] SECOND = 2'b01;
  parameter [1:0] THIRD  = 2'b10;

  always @(posedge clk or posedge rst) // sequential
  begin
    if (rst) state <= FIRST;
    else state <= next_state;
  end

  always @* // combinational
  begin
    case(state)
      FIRST:   if (restart || pause) next_state = FIRST;
               else next_state = SECOND;
      SECOND:  if (restart) next_state = FIRST;
               else if (pause) next_state = SECOND;
               else next_state = THIRD;
      THIRD:   if (!restart && pause) next_state = THIRD;
               else next_state = FIRST;
      default: next_state = FIRST;
    endcase
  end

endmodule
```

- Note use of parameters; easy encoding changes
- Asynchronous reset is implemented with state
- Synchronous reset is implemented with logic
- Default clause covers the one unused state
- Explicit next state signal

Finite State Machines

- Can also describe it in one procedural block
  - No access to “next state” signal (important?)
  - More compact...


UNIFIED PROCEDURAL DESCRIPTION
NEXT STATE LOGIC (COMBINATIONAL)
STATE VARIABLES (SEQUENTIAL)
INPUTS
STATE
SYNC RESET
ASYNC RESET

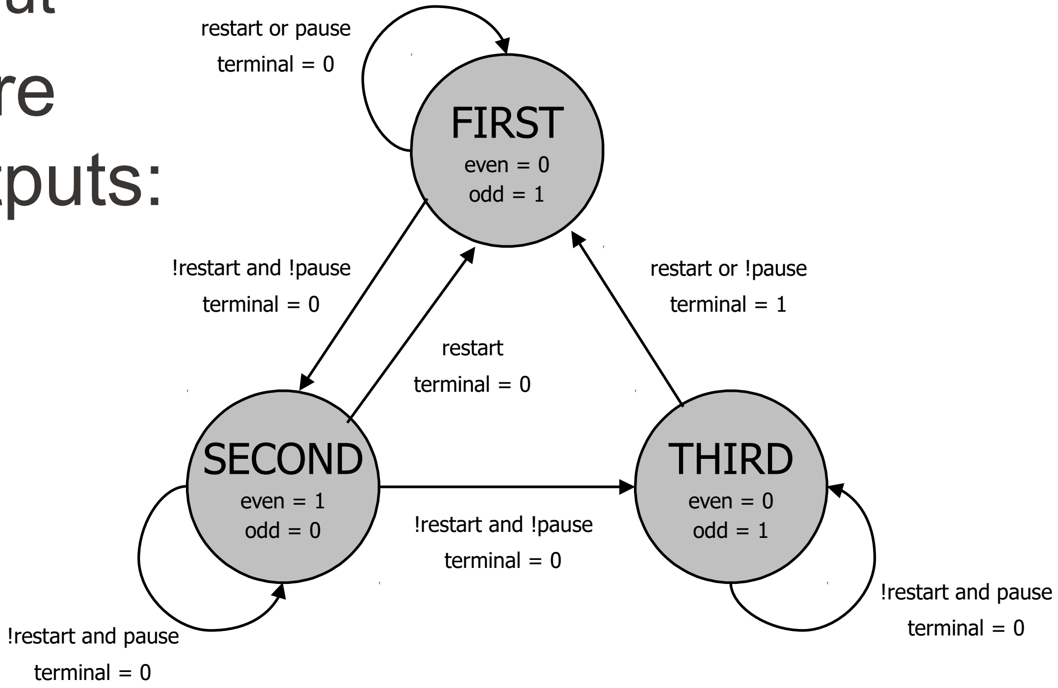
Finite State Machines

```
module fsm (
  input wire pause,
  input wire restart,
  input wire clk,
  input wire rst,
  output reg [1:0] state
  );

  parameter [1:0] FIRST  = 2'b11;
  parameter [1:0] SECOND = 2'b01;
  parameter [1:0] THIRD  = 2'b10;

  always @(posedge clk or posedge rst) // sequential
  begin
    if (rst) state <= FIRST;
    else
    begin
      case(state)
        FIRST:   if (restart || pause) state <= FIRST;
                 else state <= SECOND;
        SECOND:  if (restart) state <= FIRST;
                 else if (pause) state <= SECOND;
                 else state <= THIRD;
        THIRD:   if (!restart && pause) state <= THIRD;
                 else state <= FIRST;
        default: state <= FIRST;
      endcase
    end
  end

endmodule
```

- Note use of parameters; easy encoding changes
- Asynchronous reset and synchronous reset both implemented; distinction is made by sensitivity list
- Default clause covers the one unused state
- Implicit next state signal

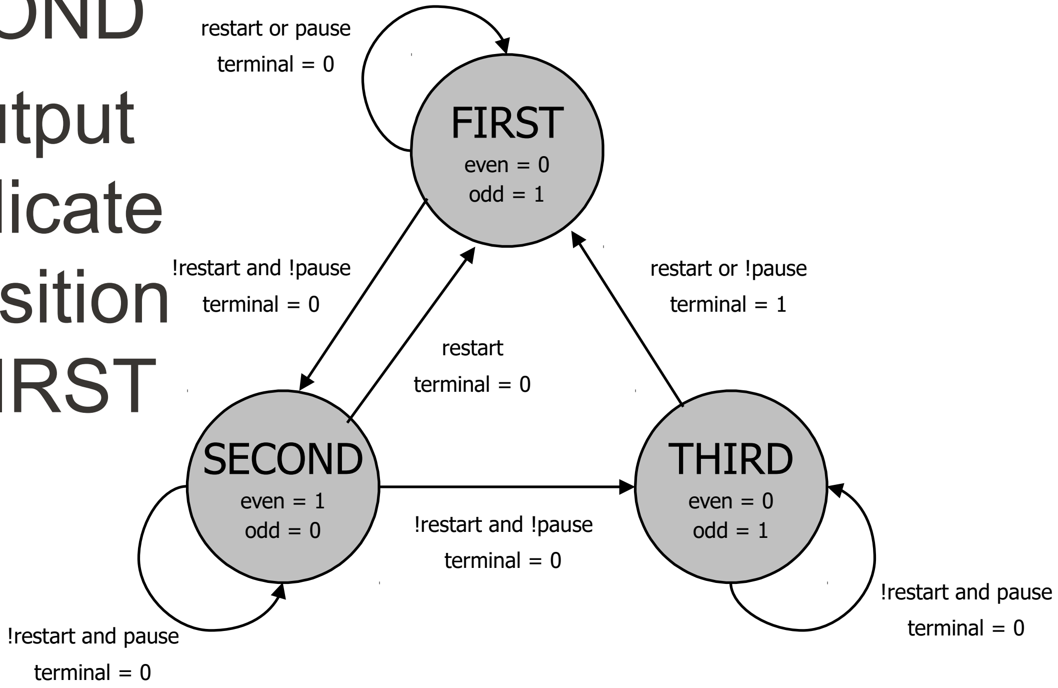
Finite State Machines

- Now, let's consider the following:
  - A set of output events
  - A function that maps states and input to output
- Suppose there are three desired outputs:
  - odd
  - even
  - terminal



Finite State Machines

- The “odd” output is asserted in FIRST and THIRD
- The “even” output is asserted in SECOND
- The “terminal” output is asserted to indicate the FSM will transition from THIRD to FIRST

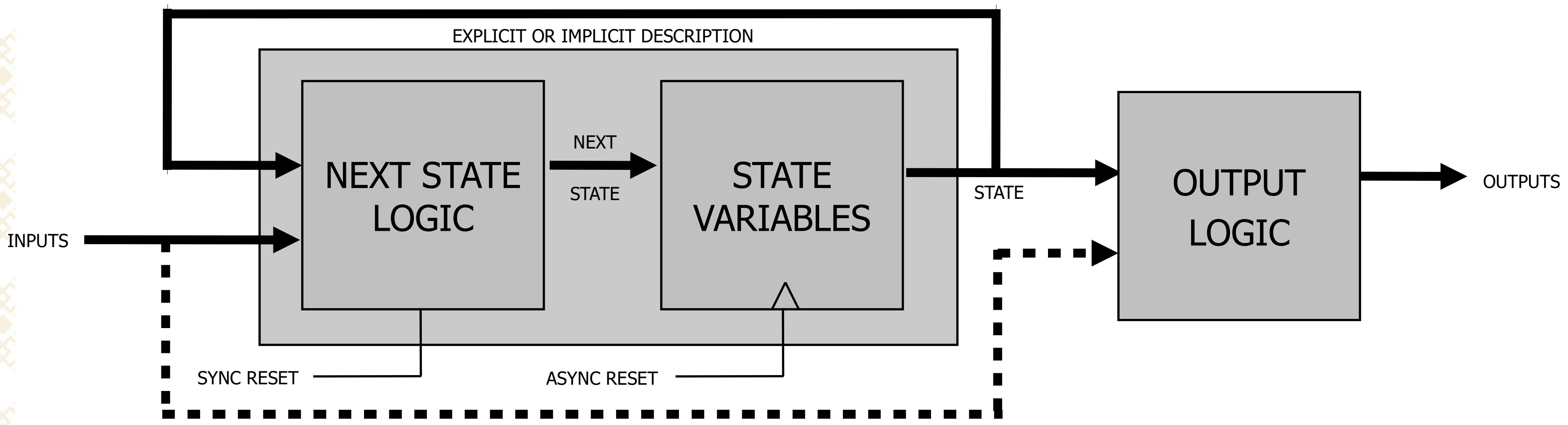


Finite State Machines

- ❖ Outputs that require functions of only the current state are Moore type outputs
  - This includes using state bits directly
  - Outputs “odd” and “even” are Moore outputs
- ❖ Outputs that require functions of the current state and the inputs are Mealy type outputs
  - Output “terminal” is a Mealy output
- ❖ Consider the latency and cycle time tradeoffs

Finite State Machines

- Describe the output functions in Verilog, just as shown in the diagram...



Finite State Machines

```
module fsm (
  input wire pause,
  input wire restart,
  input wire clk,
  input wire rst,
  output reg [1:0] state,
  output wire odd,
  output wire even,
  output wire terminal
  );

  reg [1:0] next_state;

  parameter [1:0] FIRST  = 2'b11;
  parameter [1:0] SECOND = 2'b01;
  parameter [1:0] THIRD  = 2'b10;

  always @(posedge clk or posedge rst) // sequential
  begin
    if (rst) state <= FIRST;
    else state <= next_state;
  end

  always @* // combinational
  begin
    case(state)
      FIRST:   if (restart || pause) next_state = FIRST;
               else next_state = SECOND;
      SECOND:  if (restart) next_state = FIRST;
               else if (pause) next_state = SECOND;
               else next_state = THIRD;
      THIRD:   if (!restart && pause) next_state = THIRD;
               else next_state = FIRST;
      default: next_state = FIRST;
    endcase
  end

  // output logic described using continuous assignment
  assign odd = (state == FIRST) || (state == THIRD);
  assign even = (state == SECOND);
  assign terminal = (state == THIRD) && (restart || !
pause);

endmodule
```

- Started with the FSM described using explicit next state logic, but could have used the other one
- Added three assignment statements to create the output functions

Finite State Machines

```
module fsm (
  input wire pause,
  input wire restart,
  input wire clk,
  input wire rst,
  output reg [1:0] state,
  output reg odd,
  output reg even,
  output reg terminal
  );

  parameter [1:0] FIRST  = 2'b11;
  parameter [1:0] SECOND = 2'b01;
  parameter [1:0] THIRD  = 2'b10;

  always @(posedge clk or posedge rst) // sequential
  begin
    if (rst) state <= FIRST;
    else
    begin
      case(state)
        FIRST:   if (restart || pause) state <= FIRST;
                 else state <= SECOND;
        SECOND:  if (restart) state <= FIRST;
                 else if (pause) state <= SECOND;
                 else state <= THIRD;
        THIRD:   if (!restart && pause) state <= THIRD;
                 else state <= FIRST;
        default: state <= FIRST;
      endcase
    end
  end

  // output logic described using procedural assignment
  always @* // combinational
  begin
    odd = (state == FIRST) || (state == THIRD);
    even = (state == SECOND);
    terminal = (state == THIRD) && (restart || !pause);
  end

endmodule
```

- Started with the FSM described using implicit next state logic, but could have used the other one
- This describes the same output logic as before, but uses a procedural block to create the outputs
  - Could have used case…
  - Could have used if-else...

Synthesis Considerations

- As you may now appreciate, Verilog provides a wide variety of means to express yourself
- Some modules you create, like test benches, are never intended to be synthesized into actual hardware
- In these types of modules, feel free to use the complete and terrible power of Verilog

Synthesis Considerations

- For modules you intend on synthesizing, you should apply a coding style to realize:
  - Efficiency
  - Predictability
  - Synthesizability

Synthesis Considerations

- Efficiency is important, for both performance and cost concerns
  - Use multiplication, division, and modulus operators sparingly
  - Use vectors / arrays to create “memories” only if you are sure the synthesis tool does it properly
  - Case may be better than large if-else trees
  - Keep your mind focused on what hardware you are implying by your description

Synthesis Considerations

- Predictability is important
  - Predictability of what hardware may be created by what you describe
  - Predictability of hardware behavior
    - Hardware behavior will match your description
    - No dependency on event ordering in code
  - Understanding of your description and how it may map into hardware is important when you are debugging the physical implementation

Synthesis Considerations

- Not everything you can write in Verilog can be turned into hardware by a synthesis tool
  - Multiple non-tristateable drivers on a wire
  - Initial blocks have no hardware equivalent
  - Most types of timing control have no equivalent
  - There is no hardware for comparisons to z and x
  - Or assignments to x...

Synthesis Considerations

- Realize that synthesis is machine design
  - Based on various algorithms
  - Do you think they included every one known?
  - Don’t assume, the machine is not a mind reader
  - Garbage in, garbage (or errors) out
- Read the manual for your particular synthesis tool to understand what it can and cannot do
- Carefully read synthesis warnings and errors to identify problems

FPGA Considerations

- There are four major considerations to keep in mind while creating synthesizable code for FPGA devices
  - Understand how to use vendor specific primitives
  - Code for the resources in the FPGA architecture
  - Use static timing analysis with design constraints
  - I/O insertion

FPGA Considerations

- FPGA vendors usually provide a library of primitive modules that can be instantiated in your design

- These primitives usually represent device features that cannot be efficiently described in an HDL, or cannot be inferred by synthesis

FPGA Considerations

- Examples of library components:
  - Large memories, called BlockRAM
  - Smaller memories, called DistributedRAM
  - I/O buffers, global buffers, three-state buffers
  - Clock management circuits
- Consult the Xilinx Library Guide for more info

FPGA Considerations

- FPGA devices typically have look up tables for implementing combinational logic, and D flip flops and D latches for sequential logic, in addition to a few other resource types...

- You won’t find other types of sequential elements, or weird RAMs -- avoid describing things that will be implemented inefficiently or require sequential elements to be built from combinational logic

FPGA Considerations

Subject: Synthesis Error: It's interesting and surprising!

An interesting problem occurred when I used "more than three" signals in always block sensitivity list. Same code is successfully compiled and simulated. What kind of problem is this? I put the code below.

ERROR - dummy.v line 25: Unexpected event in always block sensitivity list.

```
always @(posedge a or posedge b or posedge c or posedge d)
begin
  if (a)      z=~z;
  else if (b) z=~z;
  else if (c) z=~z;
  else if (d) z=~z;
end
```

Subject: Re: Synthesis Error: It's interesting and surprising!

What the hell is this? A quad-clock toggle flip-flop?

Subject: Re: Synthesis Error: It's interesting and surprising!

Your results are not surprising. Although your code is legal Verilog and is able to simulate, it's not synthesizable. What type of flip-flop would you expect to infer from the "always @(posedge a or posedge b or posedge c or posedge d)" construct? You need to re-think your code!

FPGA Considerations

- There are a limited number of global clock buffers in most FPGA devices
  - We will cover why these are important
  - The synthesis tool should recognize clock signals and automatically insert global clock buffers
  - You can manually insert global clock buffers by instantiation from the library

FPGA Considerations

- Where do latches come from?
  - You directly instantiated them or inferred them in your code on purpose
  - You inferred them in your code by accident
    - For procedural blocks modeling combinational logic, all the regs and wires on RHS must appear in sensitivity list
    - For procedural block modeling combinational logic, you have incompletely specified if-else or case statements
    - For modeling sequential logic, you have more than a single clock and asynchronous control in the sensitivity list

FPGA Considerations

- Templates for sensitivity lists:
  - always @(posedge clk)
    - Synchronous logic without asynchronous reset
    - Use of “negedge clk” is legal, but not in this class
  - always @(posedge clk or posedge rst)
    - Synchronous logic with asynchronous reset
    - Use of “negedge clk” or “negedge rst_n” is legal
    - Generally avoid asynchronous resets in this class

FPGA Considerations

- Templates for sensitivity lists:
  - always @(sig_1 or sig_2 or … or sig_n)
    - Combinational logic description; whenever an input changes, the output must be evaluated
    - Every signal used as an “input” in description must appear in the sensitivity list
  - always @*
    - A very welcome Verilog-2001 enhancement!
    - Combinational logic description, process is sensitive to any signal used as an “input” in the description

FPGA Considerations

- I/O insertion is the process where the synthesis tool looks at the ports of the top-level module that is synthesized and decides what FPGA primitives to insert to interface with the outside world…

- FPGA devices typically have:
  - Input buffers (IBUFs)
  - Output buffers (OBUFs)
  - Three-state output buffers (OBUFTs)

FPGA Considerations

- Compare this to the Verilog port types:
  - input ports map into IBUFs
  - output ports map into OBUFs
  - inout ports map into… what?

- How do you create bidirectional functionality at a chip boundary?

Suggested Coding Style

- One module per file, name the file same as the module, partitioning larger designs into modules on meaningful boundaries
- Always use formal port mapping of sub-modules
- Use parameters for commonly used constants

Suggested Coding Style

- Do not write modules for stuff that is easy to express (multiplexers, flip flops)
- Don’t ever just sit down and “code”; think about what hardware you want to build, how to describe it, and how you should test it
- You are not writing a computer program, you are describing hardware… ***Verilog is not C!***

SAN JOSÉ STATE
UNIVERSITY