# EECS150 - Digital Design
## Lecture 5 – Verilog & Synthesis

September 8, 2011

Elad Alon
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www-inst.eecs.berkeley.edu/~cs150



## Announcements

- Homework #1 due today
  - Drop box in 240 Cory
- Homework #2 out tonight
  - Due next Thurs.

## Example: Variable Counter



## Parameterized Version

# Generate Loop

Permits variable declarations, modules, user defined primitives, gate primitives, continuous assignments, initial blocks and always blocks to be instantiated multiple times using a for-loop.

```
// Gray-code to binary-code converter
module gray2bin1 (bin, gray);
    parameter SIZE = 8;
    output [SIZE-1:0] bin;
    input  [SIZE-1:0] gray;

  genvar i;

  generate for (i=0; i<SIZE; i=i+1) begin:bit
      assign bin[i] = ^gray[SIZE-1:i];
    end endgenerate
endmodule
```

genvar exists only in the specification - not in the final circuit.

Keywords that denote synthesis-time operation

For-loop creates instances of assignments

Loop must have constant bounds

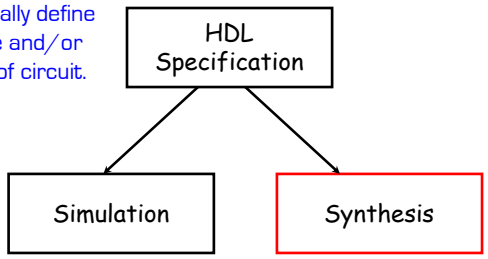
**generate if-else-if** based on an expression that is deterministic at the time the design is synthesized.

**generate case** : selecting case expression must be deterministic at the time the design is synthesized.



# Logic Synthesis

Hierarchically define structure and/or behavior of circuit.

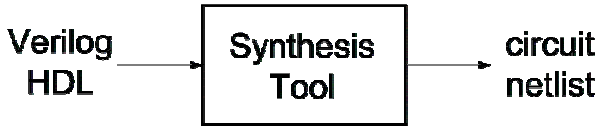



Functional verification.

Maps specification to resources of implementation platform (FPGA for us).

## Logic Synthesis

- Verilog and VHDL started out as simulation languages, but quickly people wrote programs to automatically convert Verilog code into low-level circuit descriptions (netlists).




- Synthesis converts Verilog (or other HDL) descriptions to implementation technology specific primitives:
  - For FPGAs: LUTs, flip-flops, and RAM blocks
  - For ASICs: standard cell gate and flip-flop libraries, and memory blocks.



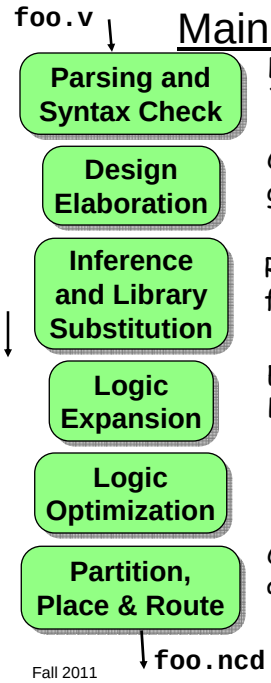
## Why Logic Synthesis?

1. Automatically manages many details of the design process:
   → Fewer bugs
   → Improved productivity
2. Abstracts the design data (HDL description) from any particular implementation technology.
   - Designs can be re-synthesized targeting different chip technologies. Ex: first implement in FPGA then later in ASIC.
3. In some cases, leads to a more optimal design than could be achieved by manual means (ex: logic optimization)

## Why Not Logic Synthesis?

1. May lead to non-optimal designs in some cases.

# Main Logic Synthesis Steps




Load in HDL file, run macro preprocessor for `define, `include, etc..

Compute parameter expressions, process generates, create instances, connect ports.

Recognize and insert special blocks (memory, flip-flops, arithmetic structures, ...)

Expand combinational logic to primitive Boolean representation.

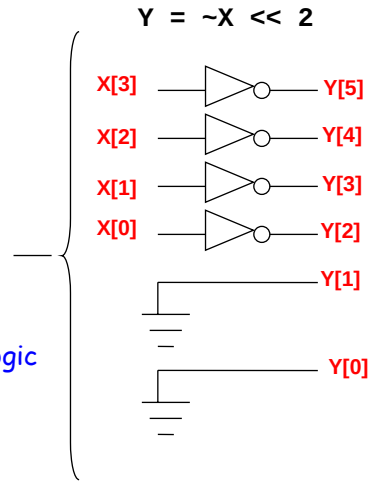
Apply Boolean algebra and heuristics to simplify and optimize under constraints.

CL to LUTs, map memory and state elements to chip, assign physical locations, route connections.



# Operators and Synthesis

- Logical operators map into primitive logic gates
- Arithmetic operators map into adders, subtractors, …
  - Unsigned 2s complement
  - Model carry: target is one-bit wider that source
  - Watch out for *, %, and /
- Relational operators generate comparators
- Shifts by constant amount are just wire connections
  - No logic involved
- Variable shift amounts a whole different story --- shifter
- Conditional expression generates logic or MUX



## Simple Example

```
module foo (A, B, s0, s1, F);
  input [3:0] A;
  input [3:0] B;
  input s0,s1;
  output [3:0] F;
  reg F;
  always @ (*)
    if (!s0 && s1 || s0) F=A; else F=B;
endmodule
```



## Some More Interesting Examples

```
module mux4to1 (out, a, b, c, d, sel);
output out;
input a, b, c, d;
input [1:0] sel;
reg out;
always @(sel or a or b or c or d)
begin
  case (sel)
    2'd0: out = a;
    2'd1: out = b;
    2'd3: out = d;
  endcase
end
endmodule
```

## Fix (Rule #1 for CL Always Blocks)

To avoid synthesizing a latch in this case, add the missing select line:

```
2'd2: out = c;
```

Or, in general, use the "default" case:

```
default:  out = foo;
```

If you don't care about the assignment in a case (for instance you know that it will never come up) then you can assign the value "x" to the variable. Example:

```
default:  out = 1'bx;
```

The x is treated as a "don't care" for synthesis and will simplify the logic.

Be careful when assigning x (don't care). If this case were to come up, then the synthesized circuit and simulation may differ.



## Another Example

```
module and_gate (out, in1, in2);
  input    in1, in2;
  output       out;
  reg          out;

  always @(in1) begin
     out = in1 & in2;
  end

endmodule
```

## Incomplete Triggers

Leaving out an input trigger usually results in latch generation for the missing trigger.

```
module and_gate (out, in1, in2);
  input    in1, in2;
  output       out;
  reg          out;

  always @(in1) begin
     out = in1 & in2;
  end

endmodule
```

Easy way to avoid incomplete triggers for combinational logic is with: **`always @*`**



## Procedural Assignments

Verilog has two types of assignments within always blocks:

- **Blocking** procedural assignment "="
  - **In simulation** the RHS is executed and the assignment is completed before the next statement is executed.
- **Non-blocking** procedural assignment "<="
  - **In simulation** the RHS is executed and all assignment take place at the same time (end of the current time step - not clock cycle).

## Synthesized Procedural Assignments

```
always @ (posedge clk) begin
 a = in;    b = a;
end
```

```
always @ (posedge clk) begin
 a <= in;   b<= a;
end
```

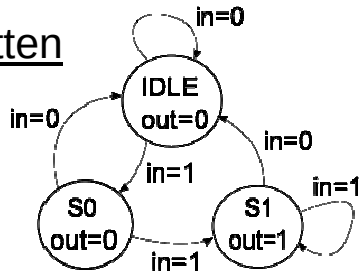


## Procedural Assignments

The sequential semantics of the blocking assignment allows variables to be multiply assigned within a single always block. Unexpected behavior can result from mixing these assignments in a single block. Standard rules:

i. **Use blocking assignments to model combinational logic within an always block ( "=").**
ii. **Use non-blocking assignments to implement sequential logic ("<=").**
iii. **Do not mix blocking and non-blocking assignments in the same always block.**
iv. **Do not make assignments to the same variable from more than one always block.**

# FSM CL block rewritten




```
always @*
  begin
   next_state = IDLE;
   out = 1'b0;
   case (state)
    IDLE   : if (in == 1'b1) next_state = S0;
    S0     : if (in == 1'b1) next_state = S1;
    S1     : begin
               out = 1'b1;
               if (in == 1'b1) next_state = S1;
             end
    default: ;
   endcase
  end
endmodule
```

* for sensitivity list

Nominal values: used unless specified below.

Within case only need to specify exceptions to the nominal values.

Note: The use of "blocking assignments" allow signal values to be "rewritten", simplifying the specification.



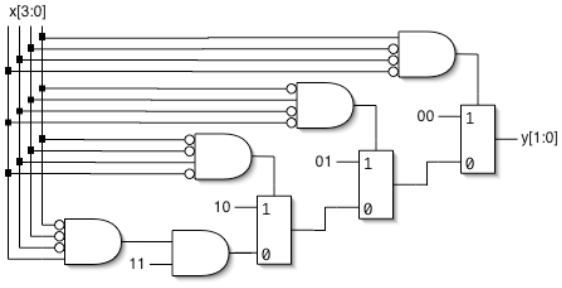
# Encoder Example

Nested IF-ELSE might lead to "priority logic"

Example: 4-to-2 encoder

```
always @(x)
begin : encode
if (x == 4'b0001) y = 2'b00;
else if (x == 4'b0010) y = 2'b01;
else if (x == 4'b0100) y = 2'b10;
else if (x == 4'b1000) y = 2'b11;
else y = 2'bxx;
end
```

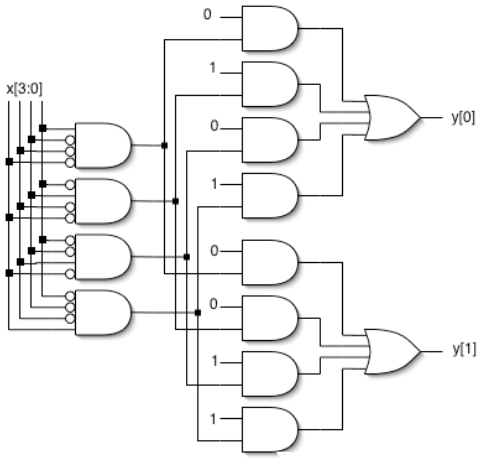
This style of cascaded logic may adversely affect the performance of the circuit.



# Encoder Example (cont.)

To avoid "priority logic" use the case construct:

```
always @(x)
begin : encode
case (x)
4'b0001: y = 2'b00;
4'b0010: y = 2'b01;
4'b0100: y = 2'b10;
4'b1000: y = 2'b11;
default: y = 2'bxx;
endcase
end
```

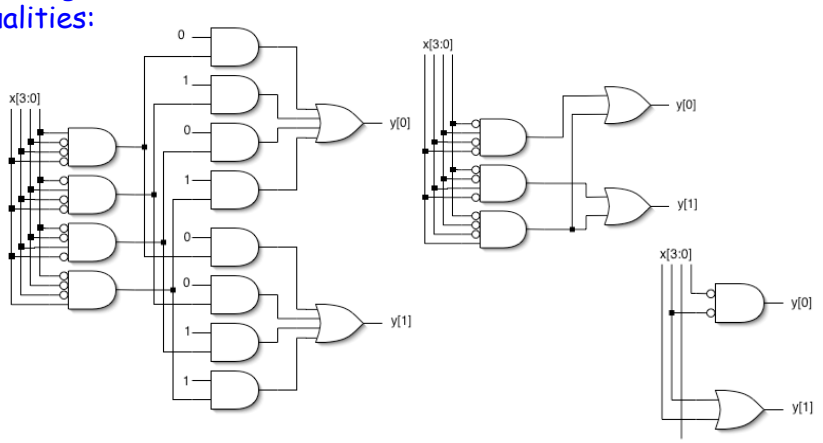



All cases are matched in parallel.



# Encoder Example (cont.)

This circuit would be simplified during synthesis to take advantage of constant values as follows and other Boolean equalities:




*A similar simplification would be applied to the if-else version also.*