EECS150 - Digital Design

Lecture 6 - Computer Aided Design (CAD) - Part I (Logic Synthesis)

Feb 4, 2010

John Wawrzynek

State Elements

Always blocks are the only way to specify the "behavior" of state elements. Synthesis tools will turn state element behaviors into state element instances.

D-flip-flop with synchronous set and reset example:

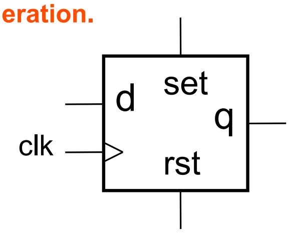
```
module dff(q, d, clk, set, rst);
  input d, clk, set, rst;
  output q;
  reg q;

  always @(posedge clk)
    if (rst)
      q <= 1'b0;
    else if (set)
      q <= 1'b1;
    else
      q <= d;
endmodule
```

keyword

"always @ (posedge clk)" is key to flip-flop generation.

This gives priority to reset over set and set over d.

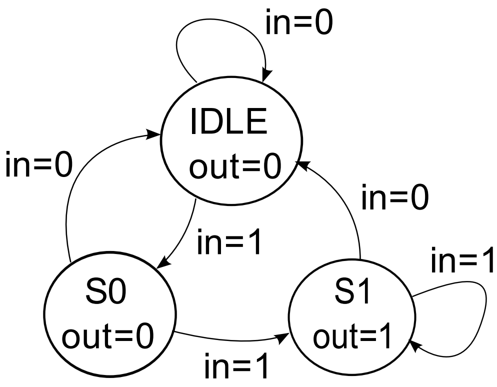


On FPGAs, maps to native flip-flop.

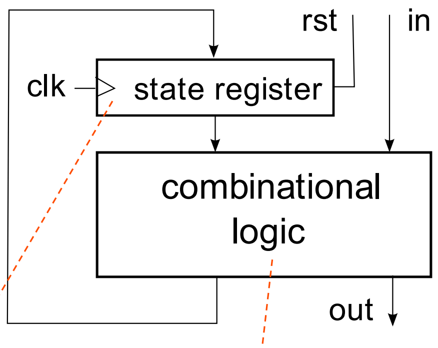
How would you add an CE (clock enable) input?

Finite State Machines

State Transition Diagram



Implementation Circuit Diagram



Holds a symbol to keep track of which bubble the FSM is in.

CL functions to determine output value and next state based on input and current state.

out = f(in, current state)

next state = f(in, current state)

What does this one do?

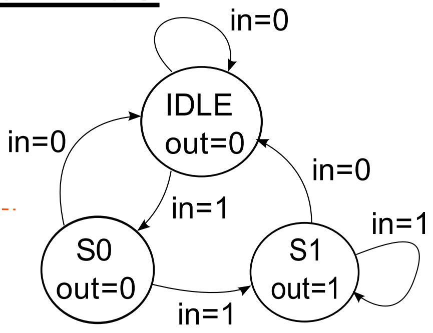
Did you know that every SDS is a FSM?

Finite State Machines

```
module FSM1(clk, rst, in, out);
input clk, rst;
input in;
output out;
```

Must use reset to force to initial state.

reset not always shown in STD



```
// Defined state encoding:
parameter IDLE = 2'b00;
parameter S0 = 2'b01;
parameter S1 = 2'b10;
reg out;
reg [1:0] state, next_state;
```

Constants local to this module.

out not a register, but assigned in always block

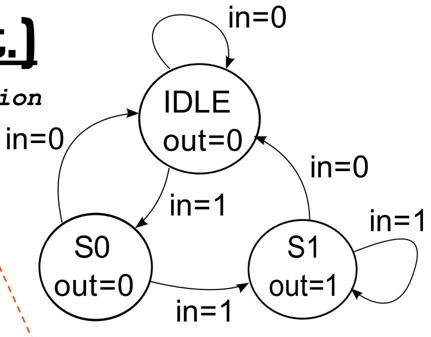
Combinational logic signals for transition.

THE register to hold the “state” of the FSM.

```
// always block for state register
always @(posedge clk)
      if (rst) state <= IDLE;
      else state <= next_state;
```

A separate always block should be used for combination logic part of FSM. Next state and output generation. (Always blocks in a design work in parallel.)

FSMs (cont.)



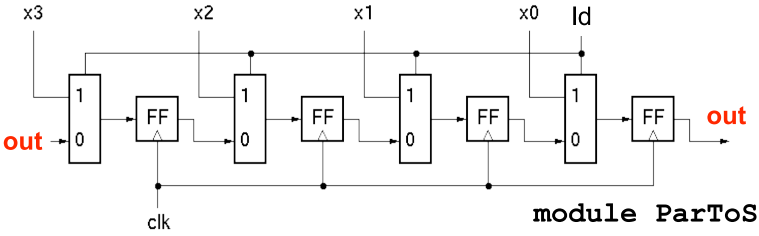
```
// always block for combinational logic portion
always @(state or in)
case (state)
// For each state def output and next
  IDLE    : begin
              out = 1'b0;
              if (in == 1'b1) next_state = S0;
              else next_state = IDLE;
            end
  S0      : begin
              out = 1'b0;
              if (in == 1'b1) next_state = S1;
              else next_state = IDLE;
            end
  S1      : begin
              out = 1'b1;
              if (in == 1'b1) next_state = S1;
              else next_state = IDLE;
             end
  default: begin
               next_state = IDLE;
               out = 1'b0;
             end
endcase
endmodule
```

Each state becomes a case clause.

For each state define:
Output value(s)
State transition

Use “default” to cover unassigned state.
Usually unconditionally transition to reset state.

Example - Parallel to Serial Converter



```
module ParToSer(ld, X, out, clk);
   input [3:0] X;
   input ld, clk;
   output out;

   reg [3:0] Q;
   wire [3:0] NS;

   assign NS =
     (ld) ? X : {Q[0], Q[3:1]};

   always @ (posedge clk)
      Q <= NS;

   assign out = Q[0];
endmodule
```

Specifies the muxing with “rotation”

forces Q register (flip-flops) to be rewritten every cycle

connect output

Parameterized Version

Parameters give us a way to generalize our designs. A module becomes a "generator" for different variations. Enables design/module reuse. Can simplify testing.

```
parameter N = 4;
```

Declare a parameter with default value.

Note: this is not a port. Acts like a "synthesis-time" constant.

```
ParToSer #(.N(8))
   ps8 ( ... );

ParToSer #(.N(64))
  ps64 ( ... );
```

Overwrite parameter N at instantiation.

```
module ParToSer(ld, X, out, CLK);
   input [N-1:0] X;
   input ld, clk;
   output out;
   reg out;
   reg [N-1:0] Q;
   wire [N-1:0] NS;

   assign NS =
     (ld) ? X : {Q[0], Q[N-1:1]};

   always @ (posedge clk)
      Q <= NS;

   assign out = Q[0];
endmodule
```

Replace all occurrences of "3" with "N-1".

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 denotes synthesis-time operations

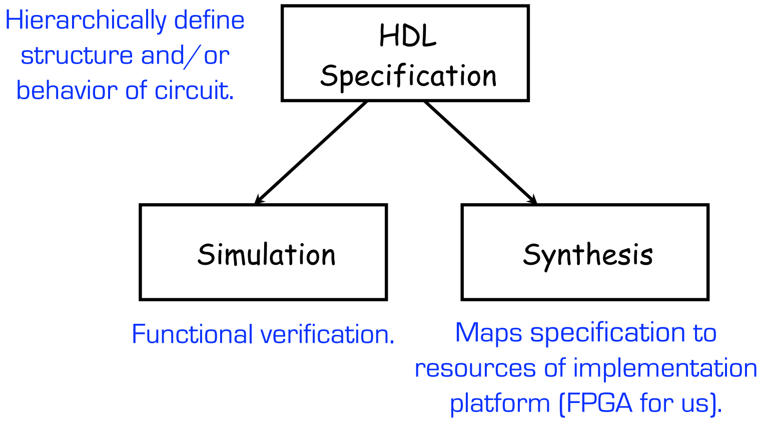
For-loop creates instances of assignments

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.

EECS150 Design Methodology



Hierarchically define structure and/or behavior of circuit.

HDL Specification

Simulation

Synthesis

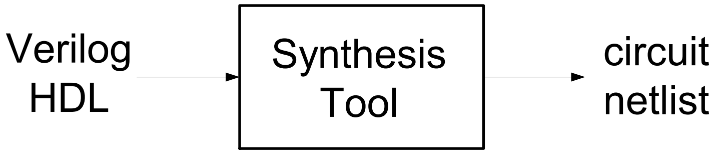
Functional verification.

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

Note: This in not the entire story. Other tools are often used analyze HDL specifications and synthesis results. More on this later.

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).



Verilog HDL

Synthesis Tool

circuit netlist

- 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.

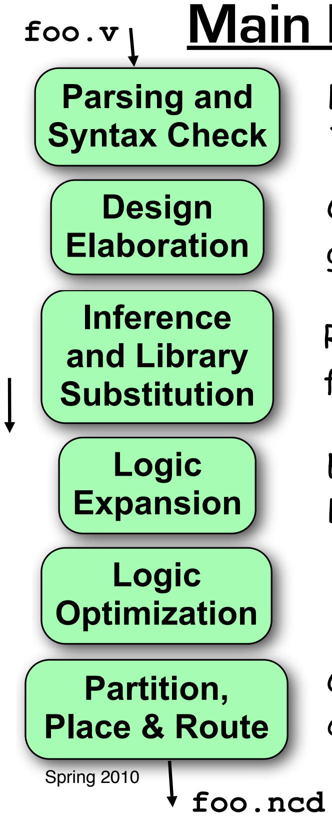
Why Logic Synthesis?

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



foo.v

Parsing and Syntax Check — Load in HDL file, run macro preprocessor for `define, `include, etc..

Design Elaboration — Compute parameter expressions, process generates, create instances, connect ports.

Inference and Library Substitution — Recognize and insert special blocks (memory, flip-flops, arithmetic structures, ...)

Logic Expansion — Expand combinational logic to primitive Boolean representation.

Logic Optimization — Apply Boolean algebra and heuristics to simplify and optimize under constraints.

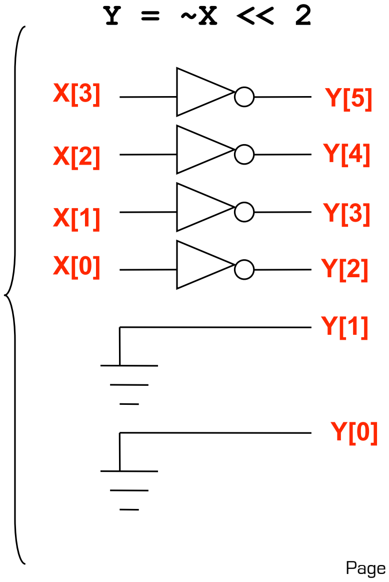
Partition, Place & Route — CL to LUTs, map memory and state elements to chip, assign physical locations, route connections.

foo.ncd

Operators and Synthesis

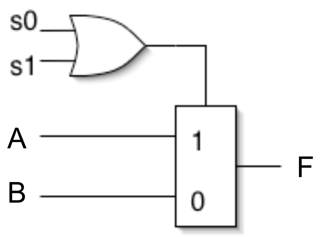
- Logical operators map into primitive logic gates
- Arithmetic operators map into adders, subtractors, …
  - Unsigned 2s complement
  - Model carry: target is one-bit wider that source
  - Watch out for *, %, and /
- Relational operators generate comparators
- Shifts by constant amount are just wire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

`Y = ~X << 2`



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
```



Should expand if-else into 4-bit wide multiplexor and optimize the control logic and ultimately to a single LUT on an FPGA:

More about Always blocks

Combinational logic always blocks

Make sure all signals assigned in a combinational always block are explicitly assigned values every time that the always block executes. Otherwise latches will be generated to hold the last value for the signals not assigned values.

Sel case value 2’d2 omitted.

Out is not updated when select line has 2’d2.

Latch is added by tool to hold the last value of out under this condition.

Similar problem with if-else statements.

```
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
```

Combinational logic 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.

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
```

in2 not in always sensitivity list.

A latched version of in2 is synthesized and used as input to the and-gate, so that the and-gate output is not always sensitive to in2.

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. Example:

    Assume A holds the value 1 ... A=2; B=A; A is left with 2, B with 2.
- **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). Example:

    Assume A holds the value 1 ... A<=2; B<=A; A is left with 2, B with 1.
- **In synthesis the difference shows up primarily when inferring state elements:**

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

b stores in

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

b stores the old a

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:

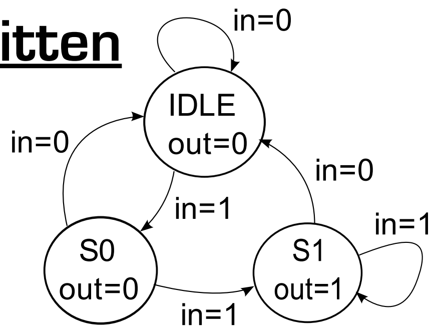
i. **Use blocking assignments to model combinational logic within an always block ("=").**

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

Normal values: used unless specified below.

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

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

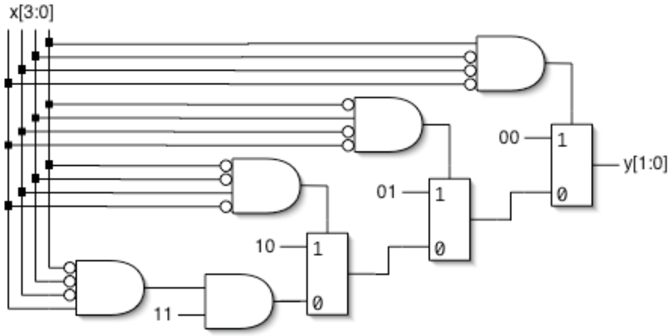
Encoder Example

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
```

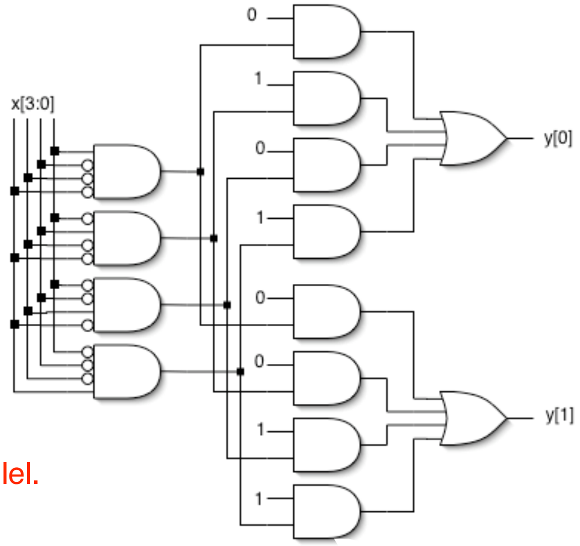
This style of cascaded logic may adversely affect the performance of the circuit.



Encoder Example (cont.)

To avoid “priority logic” use the case construct:

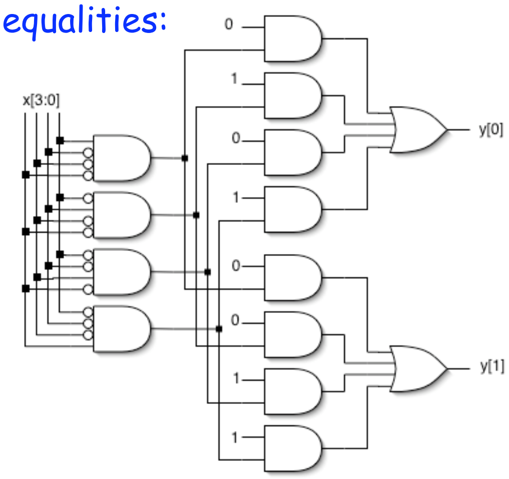
```
always @(x)
begin : encode
case (x)
4’b0001: y = 2'b00;
4’b0010: y = 2'b01;
4'b0100: y = 2'b10;
4'b1000: y = 2'b11;
default: y = 2'bxx;
endcase
end
```

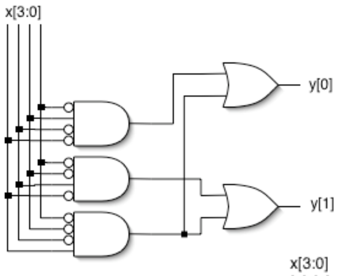


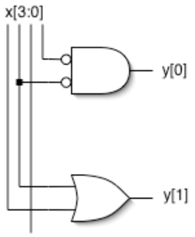
All cases are matched in parallel.

Encoder Example (cont.)

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





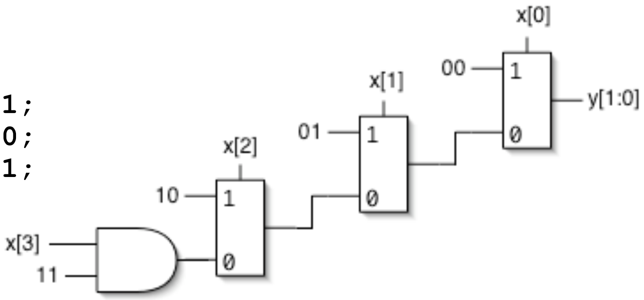


A similar simplification would be applied to the if-else version also.

Encoder Example (cont.)

If you can guarantee that only one 1 appears in the input, then simpler logic can be generated:

```
always @(x)
begin : encode
if (x[0]) y = 2'b00;
else if (x[1]) y = 2'b01;
else if (x[2]) y = 2'b10;
else if (x[3]) y = 2'b11;
else y = 2'bxx;
end
```



If the input applied has more than one 1, then this version functions as a "priority encoder". The least significant 1 gets priority (the more significant 1's are ignored). Again the circuit will be simplified when possible.

Verilog in EECS150

- We use **behavior modeling** along with **instantiation** to 1) build hierarchy and, 2) map to FPGA resources not supported by synthesis.
- Primary Style Guidelines:
  - Favor continuous assign and avoid always blocks unless:
    - no other alternative: ex: state elements, case
    - they help clarity of code & possibly circuit efficiency : ex: case vs, large nested if else
  - Use named ports.
  - Separate CL logic specification from state elements.
  - Follow our rules for procedural assignments.
- Verilog is a big language. This is only an introduction.
  - Our text book is a good source. Read and use chapter 4.
  - Be careful of what you read on the web. Many bad examples out there.
  - We will be introducing more useful constructs throughout the semester. Stay tuned!

Final thoughts on Verilog Examples

Verilog may look like C, but it describes hardware! (Except in simulation "test-benches" - which actually behave like programs.)

Multiple physical elements with parallel activities and temporal relationships.

A large part of digital design is knowing how to write Verilog that gets you the desired circuit. First understand the circuit you want then figure out how to code it in Verilog. If you do one of these activities without the other, you will struggle. These two activities will merge at some point for you.

Be suspicious of the synthesis tools! Check the output of the tools to make sure you get what you want.