EECS150 - Digital Design
Lecture 10 – Logic Synthesis

September 26, 2002
John Wawrzynek
Logic Synthesis

• Verilog and VHDL stated 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

• Converts Verilog (or other HDL) description 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.
How does it work?

• A variety of general and ad-hoc (special case) methods:
  – **Instantiation**: maintains a library of primitive modules (and, or, etc.) and adds to this user defined modules.
  – **“macro expansion” / substitution**: a large set of language operators (+, -, Boolean operators, etc.) and constructs (if-else, case) expand into special circuits.
  – **Inference**: special patterns are detected in the language description and treated specially (ex: inferring RAM blocks from variable declaration and read/write statements, FSM detection and generation from always blocks).
  – **Logic optimization**: Boolean operations are grouped and optimized with logic minimization techniques.
  – **Structural reorganization**: advanced techniques including sharing of operators, and retiming of circuits, and others?
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 @ (a or b or s0 or s1)
    if ('s0 && s1 || s0) f=a; else f=b;
endmodule

• Should expand if-else into 4-bit wide mux and optimize the control logic:
module <top_module_name>(<port list>);
/* Port declarations. followed by wire, reg, integer, task and function declarations */
/* Describe hardware with one or more continuous assignments, always blocks, 
module instantiations and gate instantiations */

// Continuous assignment
wire <result_signal_name>;
assign <result_signal_name> = <expression>;

// always block
always @( <event expression> )
begin
    // Procedural assignments
    // if statements
    // case, casex, and casez statements
    // while, repeat and for loops
    // user task and user function calls
end

// Module instantiation
<module_name> <instance_name> (<port list>);
// Instantiation of built-in gate primitive
gate_type_keyword (<port list>);
endmodule

• The order of these statements is irrelevant, all execute concurrently.
• The statements between the begin and end in an always block execute sequentially from top to bottom. (However, beware of blocking versus non-blocking assignment)
• Statements within a fork-join statement in an always block execute concurrently.
Procedural Assignments

- Verilog has two types of assignments within always blocks:
  - **Blocking** procedural assignment “=“
    - The RHS is executed and the assignment is completed before the next statement is executed. Example:
      \[ A=1; \ldots A=2; \ B=A; \]  \( A \) is left with 2, \( B \) with 2.
  - **Non-blocking** procedural assignment “<=“
    - The RHS is executed and assignment takes place at the end of the current time step (not clock cycle). Example:
      \[ A=1; \ldots A<=2; \ B<=A; \]  \( A \) is left with 2, \( B \) with 1.

- The notion of the “current time step” is tricky in synthesis, so to guarantee that your simulation matches the behavior of the synthesized circuit, follow these rules:
  - Use blocking assignments to model combinational logic within an always block.
  - Use non-blocking assignments to implement sequential logic.
  - Do not mix blocking and non-blocking assignments in the same always block.
  - Do not make assignments to the same variable from more than one always block.
Supported Verilog Constructs

- Net types: wire, tri, supply1, supply0; register types: reg, integer, time (64 bit reg); arrays of reg.
- Continuous assignments.
- Gate primitive and module instantiations.
- always blocks, user tasks, user functions.
- inputs, outputs, and inouts to a module.
- All operators (+, -, *, /, %, <, >, <=, >=, ==, !=, ||, &&, ~, &,
  ^, [, |, ^, ^, <=, >=, ==, !=, &&, |, ^, ^, <=, >=, ==, !=
  ], { }, {{ } [Note: / and % are supported for compile-time constants and constant
  powers of 2.]
- Procedural statements: if-else-if, case, casex, casez, for, repeat, while,
  forever, begin, end, fork, join.
- Procedural assignments: blocking assignments =, nonblocking assignments <= (Note: <= cannot be mixed with = for the same register).
- Compiler directives: `define, `ifdef, `else, `endif, `include, `undef
- Miscellaneous:
  - Integer ranges and parameter ranges.
  - Local declarations to begin-end block.
  - Variable indexing of bit vectors on the left and right sides of assignments.
Unsupported Language Constructs

**Generate error and halt synthesis**

- Net types: `trireg`, `wor`, `trior`, `wand`, `triand`, `tri0`, `tri1`, and charge strength;
- register type: `real`.
- Built-in unidirectional and bidirectional switches, and `pull-up`, `pull-down`.
- Procedural statements: `assign`, `deassign`, `wait`.
- Named events and event triggers.
- UDPs (user defined primitives) and `specify` blocks.
- `force`, `release`, and hierarchical net names (for simulation only).

**Simply ignored**

- `delay`, `delay control`, and `drive strength`.
- `scalared`, `vectored`.
- `initial` block.
- Compiler directives (except for `define`, `ifdef`, `else`, `endif`, `include`, and `undef`, which are supported).
- Calls to system tasks and system functions (they are only for simulation).
Combinational Logic

**CL can be generated using:**

1. primitive gate instantiation:
   - **AND, OR**, etc.

2. continuous assignment (assign keyword), example:
   ```
   Module adder_8 (cout, sum, a, b, cin);
   output cout;
   output [7:0] sum;
   input cin;
   input [7:0] a, b;
   assign {cout, sum} = a + b + cin;
   endmodule
   ```

3. Always block:
   ```
   always @ (event_expression)
   begin
     // procedural assignment statements, if statements,
     // case statements, while, repeat, and for loops.
     // Task and function calls
   end
   ```
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.

- Example:
  - 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.

```verilog
template
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 (cont.)

- To avoid synthesizing a latch in this case, add the missing select line:
  \[
  2'd2: \text{out} = c;
  \]
- Or, in general, use the “default” case:
  \[
  \text{default: out} = \text{foo};
  \]
- If you don’t care about the assignment in a case (for instance you know that it will never come up) then assign the value “x” to the variable. Example:
  \[
  \text{default: out} = 1`\text{bx};
  \]
  The x is treated as a “don’t care” for synthesis and will simplify the logic.
  (The synthesis directive “full_case” will accomplish the same, but can lead to differences between simulation and synthesis.)
Combinational Logic (cont.)

• Be careful with nested IF-ELSE. Can lead to “priority logic”
  – Example: 4-to-2 encoder

```verilog
always @(x)
begin : encode
  if (x[0] == 1'b1) y = 2'b00;
  else if (x[1] == 1'b1) y = 2'b01;
  else if (x[2] == 1'b1) y = 2'b10;
  else if (x[3] == 1'b1) y = 2'b11;
  else y = 2'bxx;
end
```

```verilog
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
```

```verilog
always @(x)
begin : encode
  begin 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
```
Sequential Logic

• D flip-flop with synchronous set/reset:

```verilog
module dff(q, d, clk, set, rst);
input d, clk, set, rst;
output q;
reg q;
always @(posedge clk)
  if (reset)
    q <= 0;
  else if (set) begin
    q <= 1;
  else begin
    q <= d;
  end
endmodule
```

• “@ (posedge clk)” key to flip-flop generation.

• Note in this case, priority logic is appropriate.

• For Xilinx Virtex FPGAs, the tool infers a native flip-flop (no extra logic is needed for the set/reset.)
module FSM1(clk, rst, enable, data_in, data_out);
input clk, rst, enable;
input [2:0] data_in;
output data_out;

/* Defined state labels; this style preferred over 'defines*/
parameter default=2'bxx;
parameter idle=2'b00;
parameter read=2'b01;
parameter write=2'b10;
reg data_out;
reg [1:0] state, next_state;

/* always block with sequential logic*/
always @(posedge clk)
  if (!rst) state <= idle;
  else state <= next_state;

• Style guidelines (some of these are to get the right result, and some just for readability)
  – Must have reset.
  – Use separate always blocks for sequential and combination logic parts.
  – Represent states with defined labels or enumerated types.
/* always block for CL */
always @(state or enable or data_in)
begin
/* Default values for FSM outputs*/
data_out <= 1'b0;
case (state)
  idle : begin
    data_out <= 1'b0;
    if (enable && data_in)
      next_state <= read;
    else next_state <= idle;
  end
read : begin

write : begin

/* Default assignment for simulation */
default : next_state <= default;
endcase
end
endmodule

• Use a CASE statement in an always to implement next state and output logic.

• Always use a default assignment and set the state variable to ‘bx.
  – This handles the unused state codes and allows the use of don’t cares leading to simplified logic.

• The FSM compiler within the synthesis tool can re-encode your states. This process is controlled by using a synthesis attribute (passed in a comment).
  • See the Synplify guide for details.
More Help

• Online documentation for Synplicity Synthesis Tool:
  – Under “refs/links” and linked to today’s lecture on calendar
  – Online examples from Synplicity.

• Bhasker (same author as Verilog reference book) on reserve in the Engineering library.

• Trial and error with the synthesis tool.
  – Synplify will display the output of synthesis in schematic form for your inspection. Try different input and see what it produces.