Verilog Synthesis

• Synthesis vs. Compilation
• Descriptions mapped to hardware
• Verilog design patterns for best synthesis

Why Perform Logic Synthesis?
1. Automatically manages many details of the design process:
   • Fewer bugs
   • Improves productivity
2. Abstractions the design data (HDL description) from any particular implementation technology:
   • Designs can be re-synthesized targeting different chip technologies:
     - Re-implement in FPGA then later in ASIC
3. In some cases, leads to a more optimal design than could be achieved by manual means (e.g.: logic optimization)

Why Not Logic Synthesis?
1. May lead to less than optimal designs in some cases

Operators
• 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

Synthesis vs. Compilation

• Compiler
  - Recognizes all possible constructs in a formally defined program language
  - Translates them to a machine language representation of execution process
• Synthesis
  - Recognizes a target dependent subset of a hardware description language
  - Maps to collection of concrete hardware resources
  - Iterative tool in the design flow

Logic Synthesis

• Verilog and VHDL started out as simulation languages, but soon programs were written to automatically convert Verilog code into low-level circuit descriptions (netlists):
  - Synthesis converts Verilog (or other HDL) descriptions to an implementation using technology-specific primitives:
    - For FPGAs: LUTs, flip-flops, and RAM blocks
    - For ASICs: standard cell gate and flip-flop libraries, and memory blocks

Verilog HDL Synthesis Tool circuit netlist
Simple Example

module foo (a,b,s0,s1,f);
input [3:0] a;
input [3:0] b;
output [3:0] f;
reg f;
always @ (a or b or s0 or s1)
if ((a & b) || ~a) f = a; else f = b;
endmodule

should expand if else into 4-bit wide multiplexer (a, b, f are 4-bit vectors) and optimize/minimize the control logic:

Procedural Assignments

- Verilog has two types of assignments within always blocks:
  - Blocking procedural assignment "=
    - RHS is executed and assignment is completed before the next statement is executed: e.g.,
      Assume A holds the value 1, A[2]; B=A: A is left with 2, B with 2.
  - Non-blocking procedural assignment "="
    - RHS is executed and assignment takes place at the end of the current time step (not clock cycle); e.g.,
      Assume A holds the value 1, A[2]; B=A: A is left with 2, B with 1.
- Notion of "current time step" is tricky in synthesis, so to guarantee that your simulation matches the behavior of the synthesized circuit, follow these rules:
  1. Use blocking assignments to model combinational logic within an always block
  2. Use non-blocking assignments to implement sequential logic
  3. Do not mix blocking and non-blocking assignments in the same always block
  4. Do not make assignments to the same variable from more than one always block

Unsupported Language Constructs

- Net types: trireg, war, trior, wand, triand, tri0, ... to system tasks and system functions (they are only for simulation)
- Generate error and halt synthesis

Simply ignored

- Delay, delay control, and drive strength
- Scanned, vectorized
- 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)

Unsupported Language Constructs

Generate error and halt synthesis

Combinational Logic

Supports Verilog Constructs

- Net types: wire, tri, supply0, supply1;
- 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 inputs to a module
- All operators (+, *, /, %, <<, >>, &,, |, &., |., ^, ~, <<, >>, <, >, 2, 1, {1}, {{1}}) (Note: / and % are supported for compile-time constants and constant powers of 2)
- Procedural statements: if else: if case, case sel, 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
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:
- $\text{Sel case value } 2'd2 \text{ omitted}$
- $\text{Out is not updated when select line has } 2'd2$
- $\text{Latch is added by tool to hold the last value of out under this condition}$

\begin{verbatim}
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'd2: out = d;
    default: out = 2'bxx;
    endcase
  end
endmodule
\end{verbatim}

Latch Rule

- If a variable is not assigned in all possible executions of an always statement then a latch is inferred
  - E.g., when not assigned in all branches of an if or case
  - Even a variable declared locally within an always is inferred as a latch if incompletely assigned in a conditional statement

Encoder Example

- Nested IF-ELSE might lead to "priority logic"
  - Example: 4-to-2 encoder

\begin{verbatim}
always @(x)
  begin : encode
    if (x == 4'b0001) y = 2'b01;
    else if (x == 4'b0010) y = 2'b11;
    else if (x == 4'b1010) y = 2'b10;
    else if (x == 4'b1000) y = 2'b11;
    else y = 2'bxx;
  end
\end{verbatim}

Encoder Example (cont.)

- To avoid "priority logic" use the case construct:

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

- All cases are matched in parallel
  - Note, you don't need the "parallel case" directive (except under special circumstances, described later)

Encoder Example (cont.)

- 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 (one hot encoding), then simpler logic can be generated:

```verilog
always @ (x) begin : encode
  case (x)
    4’b1001: y = 2’b00;
    4’b1010: y = 2’b01;
    default: y = 2’bx;
  endcase
end
```

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

Encoder Example (cont.)

- Parallel version of "priority encoder":

```verilog
module priority_encoder(clk, rst, enable, data_in, data_out);
input clk, rst, enable;
input data_in;
output data_out;

// Define state encoding: 
// 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 for sequential logic */
always @ (rst) state <= idle;
else state <= next_state;
endmodule
```

Sequential Logic

- Example: 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 (set) q <= 1;
else if (reset) q <= 0;
else q <= d;
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 needed for set/reset

We prefer synchronous set/reset, but how would you specify an asynchronous reset/clear?

Finite State Machines

```verilog
module FSM(clk, rst, enable, data_in, data_out);
input clk, rst, enable;
input data_in;
output data_out;

// always block for CL */
always @ (state or enable or data_in)
begin
  case (state)
    // For each state def output and next */
    idle : begin
      data_out = 1’b0;
      if (enable)
        next_state = read;
      else next_state = idle;
    end
    read : begin
      data_out = 1’b1;
      if (enable)
        next_state = idle;
    end
    endcase
end
```

- Use CASE statement in an always block to implement next state and output logic
- Always use default case and assert the state variable and output to 'x'
- Avoids implied latches
- Allows use of don’t cares leading to simplified logic
- "FSM compiler" within synthesis tool can re-encode your states; Process is controlled by using a synthesis attribute (passed in a comment);
- Details in Synplify guide
More Help

- Online documentation for Synplify Synthesis Tool:
  - Under Documents/General Documentation, see Synplify Web Site/Literature:
  - Online examples from Synplicity
- Bhasker is a good synthesis reference
- 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

Bottom-line

- Have the hardware design clear in your mind when you write the verilog
- Write the verilog to describe that HW
  - It is a Hardware Description Language not a Hardware Imagination Language
- If you are very clear, the synthesis tools are likely to figure it out