Verilog Synthesis

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

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).

\[
\text{Verilog HDL} \rightarrow \text{Synthesis Tool} \rightarrow \text{circuit netlist}
\]

- 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
Why Perform Logic Synthesis?
1. Automatically manages many details of the design process:
   • Fewer bugs
   • Improves productivity
2. Abstracts the design data (HDL description) from any particular implementation technology
   • Designs can be re-synthesized targeting different chip technologies; E.g.: 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 (e.g.: logic optimization)

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

How Does It Work?
• Variety of general and ad-hoc (special case) methods:
  - Instantiation: maintains a library of primitive modules (AND, OR, etc.) and 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 (e.g.: inferring memory blocks from variable declaration and read/write statements, FSM detection and generation from "always @ (posedge clk)" 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 (moving FFs), and others
Operators

- Logical operators map into primitive logic gates
- Arithmetic operators map into adders, subtractors, ...
  - Unsigned 2s complement
  - Model carry: target is one-bit wider than 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

\[ Y = \sim X << 2 \]

\[
\begin{align*}
X[0] & \quad Y[2] \\
& \quad Y[1] \\
& \quad Y[0]
\end{align*}
\]

Synthesis vs. Compilation

Levels of Representation

- **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
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 multiplexer (a, b, f are 4-bit vectors) and optimize/minimize the control logic:

Module Template

Synthesis tools expects to find modules in this format.

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, casez, 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

• Order of these statements is irrelevant, all execute concurrently
• 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** “=”
    - 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:
  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

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, 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 (different from the "continuous assignment"), 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
- Scaraed, 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!

```verilog
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
```

- 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

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

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

Or, in general, use the “default” case:

```verilog
  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 assign the value “x” to the variable; E.g.:

```verilog
  default: out = 1’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.)
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

Midterm #1 Results

<table>
<thead>
<tr>
<th>Score</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
</tr>
<tr>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>34</td>
<td>1</td>
</tr>
<tr>
<td>35</td>
<td>1</td>
</tr>
<tr>
<td>36</td>
<td>1</td>
</tr>
<tr>
<td>37</td>
<td>1</td>
</tr>
<tr>
<td>38</td>
<td>1</td>
</tr>
<tr>
<td>39</td>
<td>1</td>
</tr>
<tr>
<td>40</td>
<td>1</td>
</tr>
<tr>
<td>41</td>
<td>1</td>
</tr>
<tr>
<td>42</td>
<td>1</td>
</tr>
<tr>
<td>43</td>
<td>1</td>
</tr>
<tr>
<td>44</td>
<td>1</td>
</tr>
<tr>
<td>45</td>
<td>1</td>
</tr>
<tr>
<td>46</td>
<td>1</td>
</tr>
<tr>
<td>47</td>
<td>1</td>
</tr>
<tr>
<td>48</td>
<td>1</td>
</tr>
<tr>
<td>49</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>1</td>
</tr>
</tbody>
</table>

Mean GPA = 3.43
Encoder Example

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

```verilog
always @ (x)
begin
    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:

```verilog
always @ (x)
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
```

- 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:

```
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" -- least significant 1 gets priority (the more significant 1's are ignored); the circuit will be simplified when possible
Encoder Example (cont.)

- Parallel version, assuming we can guarantee only one 1 in the input:

```verilog
always @(x)
begin : encode
case(x) // synthesis parallel_case
  4'bxxxx: y = 2'b00;
  4'bxxx: y = 2'b01;
  4'bxxx: y = 2'b10;
  4'bxxxx: y = 2'b11;
default: y = 2'bxx;
endcase
end

• Note now more than one case might match the input
• Therefore use "parallel case" directive: without it, synthesis adds appropriate matching logic to force priority
  - Semantics of case construct says that the cases are evaluated from top to bottom
  - Only an issue for synthesis when more than one case could match input
```

Encoder Example (cont.)

- Parallel version of "priority encoder":

```verilog
always @(x)
begin : encode
case(x)
  4'bxxxx: y = 2'b00;
  4'bxxx: y = 2'b01;
  4'bxxxx: y = 2'b10;
  4'bxxxxx: y = 2'b11;
default: y = 2'bxx;
endcase
end
```

- Note: "parallel case" directive is not used, synthesis adds appropriate matching logic to force priority
  - Just what we want for a priority encoder
- Behavior matches that of the if-else version presented earlier
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 (reset)
    q <= 0;
else if (set)
    q <= 1;
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 asynchronous preset/clear?

Finite State Machines

```verilog
module FSM1(clk, rst, enable, data_in, data_out);
input clk, rst, enable;
input data_in;
output data_out;
/* Defined 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 @(posedge clk)
    if (rst) state <= idle;
    else state <= next_state;
endmodule
```

- 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
FSMs (cont.)

```verilog
/* always block for CL */
always @(state or enable or data_in)
begin
    case (state)
        idle : begin
            data_out = 1'b0;
            if (enable)
                next_state = read;
            else next_state = idle;
        end
        read : begin ... end
        write : begin ... end
        default : begin
            next_state = default;
            data_out = 1'bx;
        end
    endcase
endmodule
```

- Use CASE statement in an always to implement next state and output logic
- Always use default case and assert the state variable and output to 'bx:
  - 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:**
  - 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