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).
- 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

Operators

- Logical operators map into primitive logic gates
- Arithmetic operators map into adders, subtractors, ...
  - Signed/2’s complement
  - Trap 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

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
Unsupported Language Constructs

Generate error and halt synthesis

- Net types: trist, war, trw, trd, 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) design, 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;
- 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);

Unpredictable

- all other

Procedural assignments 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.

- The 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 always blocks.
  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.

Supported Verilog Constructs

- Net types: wire, tri, supply,
- 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 (x, ^, /, &, ~, ||, &&, &&, ^, |, &,
    - ==, !=, <=, >=, <=, =, =) (Note: /= and % are supported for compile-time constraints and constant powers of 2)
- Procedural statements: if-else-if, case, casez, for, repeat, while, forever, begin, end, fork, join

Module Template

Synthesis tools expects to find modules in this format.

```
module simple_module (result_signal);
// instantiate components, e.g., +, *
// instantiations of separate components
// always <block1>
// block2
// if-else-if
// case,
// end
// for, while
// forever
// fork, join,
// user blocks, user tasks, user functions
// `ifdef, `else, `endif, `include,
// `define, `undef
// `assert, `assume
// `time, `stop, `wait
// synthesize block
// module simple_module (result_signal);
```

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;
     - input (1:0) a;
     - input (1:0) 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

    ```
    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

  ```
  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;
      default: out = foo;
    endcase
  end
  endmodule
  ```

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

    ```
    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

Midterm #1 Results

- GPA=3.43

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
  - Note, you don’t need the “parallel case” directive (except under special circumstances, described later)
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;
  // @posedge clk) key to flip-flop generation
  always @(posedge clk)
    if (reset) q <= 0;
    else if (set) q <= 1;
    else q <= d;
  endmodule
  ```

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

Finite State Machines

- Example (cont.):

```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 "define*/
// parameter default=0'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/param
always @(posedge clk)
  if (rst) state <= idle;
  else state <= next_state;
```
FSMs (cont.)

```verilog
/* 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
            ...
        end
        write : begin
            ...
        end
        default : begin
            next_state = default;
            data_out = 1'b0;
        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 'b0
- 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 Synplify
  - 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