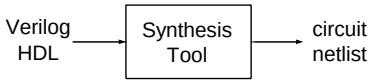
Synthesis of Digital Circuits

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?

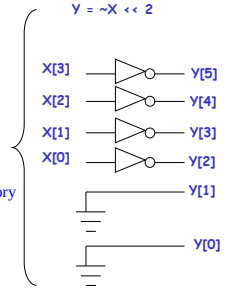
1. May lead to less than optimal designs in some cases

How Does It Work?

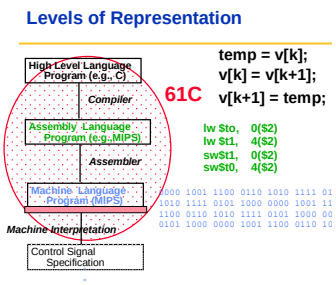
- Variety of general and ad-hoc (special case) methods:
  - **Instantiation**: maintains a library of primitive modules (AND, OR, etc.) and user defined modules
  - **"Macro expansion"/substitution**: a large set of language operators (+, -, Boolean operators, etc.) and constructs (if-else, case) expand 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



Synthesis vs. Compilation

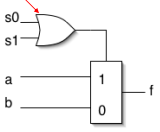


- Compiler
  - Recognizes all possible constructs in a formally defined program language
  - Translates them to a machine language representation of execution process
- Synthesis
  - Recognizes a target dependent subset of a hardware description language
  - Maps to collection of concrete hardware resources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

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, 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
```

- 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**

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
```


VERILOG: Synthesis - Combinational Logic

- Combination logic function can be expressed as:

logic_output(t) = f(logic_inputs(t))

- Rules
  - Avoid technology dependent modeling; i.e. implement functionality, not timing.
  - The combinational logic must not have feedback.
  - Specify the output of a combinational behavior for all possible cases of its inputs.
  - Logic that is not combinational will be synthesized as sequential.

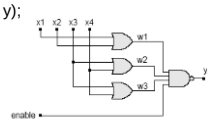
Styles for Synthesizable Combinational Logic

- Synthesizable combinational can have following styles
  - Netlist of gate instances and Verilog primitives (Fully structural)
  - Combinational UDP (Some tools)
  - Functions
  - Continuous Assignments
  - Behavioral statements
  - Tasks without event or delay control
  - Interconnected modules of the above

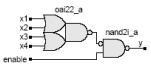
Synthesis of Combinational Logic – Gate Netlist

- Synthesis tools further optimize a gate netlist specified in terms of Verilog primitives
- Example:

```
module or_nand_1 (enable, x1, x2, x3, x4, y);
  input enable, x1, x2, x3, x4;
  output y;
  wire w1, w2, w3;
  or (w1, x1, x2);
  or (w2, x3, x4);
  or (w3, x3, x4); // redundant
  nand (y, w1, w2, w3, enable);
endmodule
```



Pre-synthesis



Post-synthesis

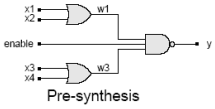
Synthesis of Combinational Logic – Gate Netlist (cont.)

- General Steps:
  - Logic gates are translated to Boolean equations.
  - The Boolean equations are optimized.
  - Optimized Boolean equations are covered by library gates.
  - Complex behavior that is modeled by gates is not mapped to complex library cells (e.g. adder, multiplier)
  - The user interface allows gate-level models to be preserved in synthesis.

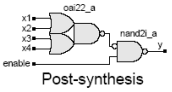
Synthesis of Combinational Logic – Continuous Assignments

Example:

```
module or_nand_2 (enable, x1, x2, x3, x4, y);
  input enable, x1, x2, x3, x4;
  output y;
  assign y = !(enable & (x1 | x2) & (x3 | x4));
endmodule
```



Pre-synthesis



Post-synthesis

Synthesis of Combinational Logic – Behavioral Style

Example:

```
module or_nand_3 (enable, x1, x2, x3, x4, y);
  input enable, x1, x2, x3, x4;
  output y;
  reg y;
  always @ (enable or x1 or x2 or x3 or x4)
    if (enable)
      y = !((x1 | x2) & (x3 | x4));
    else
      y = 1; // operand is a constant.
endmodule
```

Note: *Inputs to the behavior must be included in the event control expression, otherwise a latch will be inferred.*

Synthesis of Combinational Logic – Functions

Example:

```
module or_nand_4 (enable, x1, x2, x3, x4, y);
  input enable, x1, x2, x3, x4;
  output y;
  assign y = or_nand(enable, x1, x2, x3, x4);

  function or_nand;
    input enable, x1, x2, x3, x4;
    begin
      or_nand = ~(enable & (x1 | x2) & (x3 | x4));
    end
  endfunction
endmodule
```


Synthesis of Combinational Logic – Tasks

Example:

```
module or_nand_5 (enable, x1, x2, x3, x4, y);
  input enable, x1, x2, x3, x4;
  output y;
  reg y;
  always @ (enable or x1 or x2 or x3 or x4)
    or_nand (enable, x1, x2, x3c, x4);

  task or_nand;
    input enable, x1, x2, x3, x4;
    output y;
    begin
      y = !(enable & (x1 | x2) & (x3 | x4));
    end
  endtask
endmodule
```

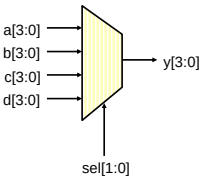
Construct to Avoid for Combinational Synthesis

- Edge-dependent event control
- Multiple event controls within the same behavior
- Named events
- Feedback loops
- Procedural-continuous assignment containing event or delay control
- **fork** ... **join** blocks
- **wait** statements
- External **disable** statements
- Procedural loops with timing
- Data dependent loops
- Tasks with timing controls
- Sequential UDPs

Synthesis of Multiplexors

- **Conditional Operator**

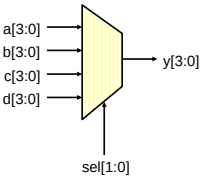
```
module mux_4bits(y, a, b, c, d, sel);
  input [3:0] a, b, c, d;
  input [1:0] sel;
  output [3:0] y;
  assign y =
    (sel == 0) ? a :
    (sel == 1) ? b :
    (sel == 2) ? c :
    (sel == 3) ? d : 4'bx;
endmodule
```



Synthesis of Multiplexors (cont.)

- **CASE Statement**

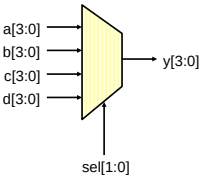
```
module mux_4bits (y, a, b, c, d, sel);
  input [3:0] a, b, c, d;
  input [1:0] sel
  output [3:0] y;
  reg [3:0] y;
  always @ (a or b or c or d or sel)
    case (sel)
      0: y = a;
      1: y = b;
      2: y = c;
      3: y = d;
      default: y = 4'bx;
    endcase
endmodule
```



Synthesis of Multiplexors (cont.)

- **if .. else Statement**

```
module mux_4bits (y, a, b, c, d, sel);
  input [3:0] a, b, c, d;
  input [1:0] sel
  output [3:0] y;
  reg [3:0] y;
  always @ (a or b or c or d or sel)
    if (sel == 0) y = a; else
    if (sel == 1) y = b; else
    if (sel == 2) y = c; else
    if (sel == 3) y = d;
    else y = 4'bx;
endmodule
```

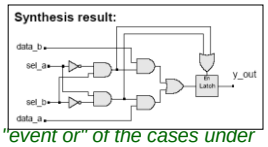


Note: *CASE* statement and *if/else* statements are more preferred and recommended styles for inferring MUX

Unwanted Latches

- Unintentional latches generally result from incomplete case statement or conditional branch
- Example: case statement

```
always @ (sel_a or sel_b or data_a or data_b)
  case ({sel_a, sel_b})
    2'b10: y_out = data_a;
    2'b01: y_out = data_b;
  endcase
```

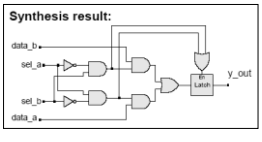


The latch is enabled by the "event or" of the cases under which assignment is explicitly made. e.g. ({sel_a, sel_b} == 2'b10) or ({sel_a, sel_b} == 2'b01)

Unwanted Latches (cont.)

- Example: if .. else statement

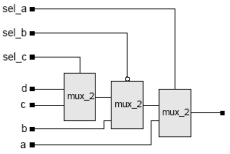
```
always @ (sel_a or sel_b or data_a or data_b)
  if ({sel_a, sel_b} == 2'b10)
    y_out = data_a;
  else if ({sel_a, sel_b} == 2'b01)
    y_out = data_b;
```



Priority Logic

- When the branching of a conditional (if) is not mutually exclusive, or when the branches of a case statement are not mutually exclusive, the synthesis tool will create a priority structure.
- Example:

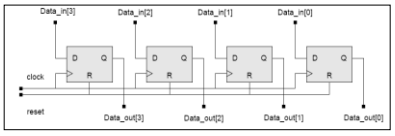
```
module mux_4pri (y, a, b, c, d, sel_a, sel_b, sel_c);
  input a, b, c, d, sel_a, sel_b, sel_c;
  output y;
  reg y;
  always @ (sel_a or sel_b or sel_c or a or b or c or d)
  begin
    if (sel_a == 1) y = a; else
    if (sel_b == 0) y = b; else
    if (sel_c == 1) y = c; else
      y = d;
  end
endmodule
```



VERILOG: Synthesis - Sequential Logic

- General Rule: A variable will be synthesized as a flip-flop when its value is assigned synchronously with an edge of a signal.
- Example:

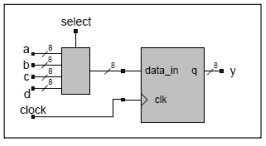
```
module D_reg4a (Data_in, clock, reset, Data_out);
  input [3:0] Data_in;
  input clock, reset;
  output [3:0] Data_out;
  reg [3:0] Data_out;
  always @ (posedge reset or posedge clock)
    if (reset == 1'b1) Data_out <= 4'b0;
    else Data_out <= Data_in;
endmodule
```



Registered Combinational Logic

- Combinational logic that is included in a synchronous behavior will be synthesized with registered output.
- Example:

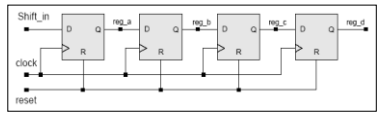
```
module mux_reg (a, b, c, d, y, select, clock);
  input [7:0] a, b, c, d;
  output [7:0] y;
  input [1:0] select;
  reg [7:0] y;
  always @ (posedge clock)
    case (select)
      0: y <= a; // non-blocking
      1: y <= b; // same result with =
      2: y <= c;
      3: y <= d;
      default y <= 8'bx;
    endcase
endmodule
```



Verilog Shift Register

- Shift register can be implemented knowing how the flip-flops are connected

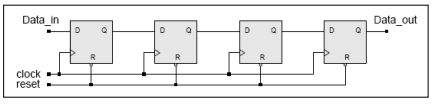
```
always @ (posedge clock) begin
  if (reset == 1'b1)  begin
    reg_a <= 1'b0;
    reg_b <= 1'b0;
    reg_c <= 1'b0;
    reg_d <= 1'b0;
  end
  else begin
    reg_a <= Shift_in;
    reg_b <= reg_a;
    reg_c <= reg_b;
    reg_d <= reg_c;
  end
end
```



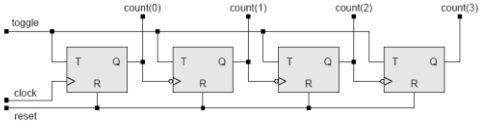
Verilog Shift Register

- Shift register can be implemented using concatenation operation referencing the register outputs

```
module Shift_reg4 (Data_out, Data_in, clock, reset);
  input Data_in, clock, reset;
  output Data_out;
  reg [3:0] Data_reg;
  assign Data_out = Data_reg[0];
  always @ (negedge reset or posedge clock) begin
    if (reset == 1'b0) Data_reg <= 4'b0;
    else Data_reg <= {Data_in, Data_reg[3:1]};
  end
endmodule
```



Verilog Ripple Counter

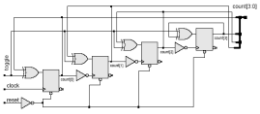


4-bit Ripple Counter

Verilog Ripple Counter

```
module ripple_counter (clock, toggle, reset, count);
  input clock, toggle, reset;
  output [3:0] count;
  reg [3:0] count;
  wire c0, c1, c2;
  assign c0 = count[0], c1 = count[1], c2 = count[2];

   always @ (posedge reset or posedge clock)
     if (reset == 1'b1) count[0] <= 1'b0;
     else if (toggle == 1'b1) count[0] <= ~count[0];

   always @ (posedge reset or negedge c0)
     if (reset == 1'b1) count[1] <= 1'b0;
     else if (toggle == 1'b1) count[1] <= ~count[1];

   always @ (posedge reset or negedge c1)
     if (reset == 1'b1) count[2] <= 1'b0;
     else if (toggle == 1'b1) count[2] <= ~count[2];

   always @ (posedge reset or negedge c2)
     if (reset == 1'b1) count[3] <= 1'b0;
     else if (toggle == 1'b1) count[3] <= ~count[3];
endmodule
```

Synthesis Result



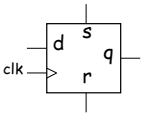
Sequential Logic

- Example: D flip-flop with synchronous set/reset:

```
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

```
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;
```

- 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.)

```
/* 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'bx;
            end
endcase
end
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

Bottom-line

- Have the hardware design clear in your mind when you write the verilog
- Write the verilog to describe that HW
- If you are very clear, the synthesis tools are likely to figure it out

Reference