Digital Design with Verilog

Verilog - Synthesis

Lecture 17: Logic Synthesis with Verilog HDL

Introduction

- Advances in logic synthesis have pushed HDLs into the forefront of digital design technology.
- Logic synthesis tools have cut design cycle times significantly.
- Designers can design at a high level of abstraction and thus reduce design time.

Learning objectives

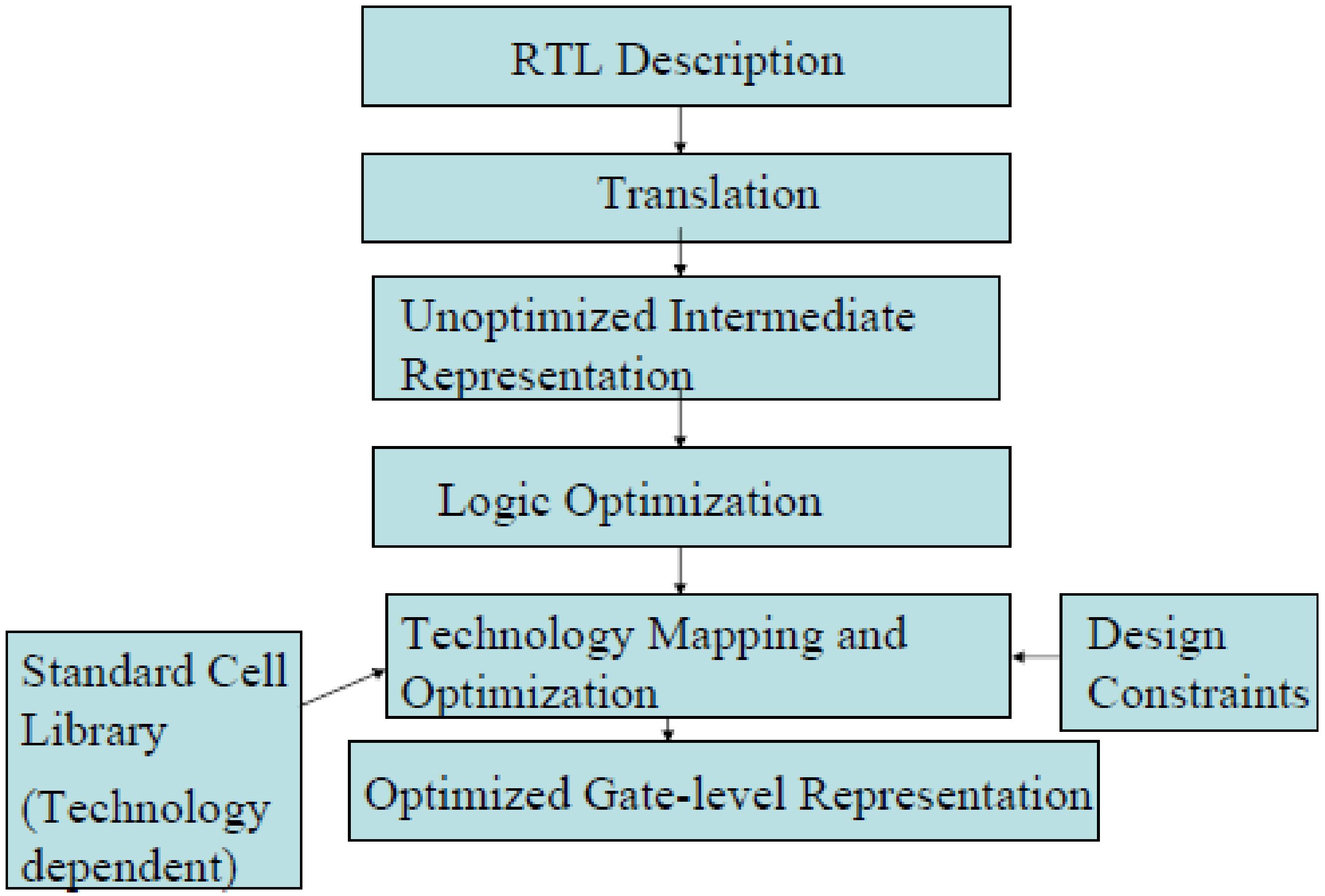
- Define logic synthesis and explain the benefits of logic synthesis.
- Identify Verilog HDL constructs and operators accepted in logic synthesis.
- Understand how the logic synthesis tool interprets these constructs.
- Explain a typical design flow, using logic synthesis.

Learning objectives

- Describe the components in the logic synthesis-based design flow.
- Describe verification of the gate-level netlist produced by logic synthesis.
- Understand techniques for writing efficient RTL descriptions.

What is Synthesis?

- Synthesis is the process of constructing a gate level netlist from a register-transfer level model of a circuit.


RTL Description
Translation
Unoptimized Intermediate Representation
Logic Optimization
Technology Mapping and Optimization
Design Constraints
Standard Cell Library (Technology dependent)
Optimized Gate-level Representation

Technology Library

- Comprises of cells. Each cell must be *characterized*.

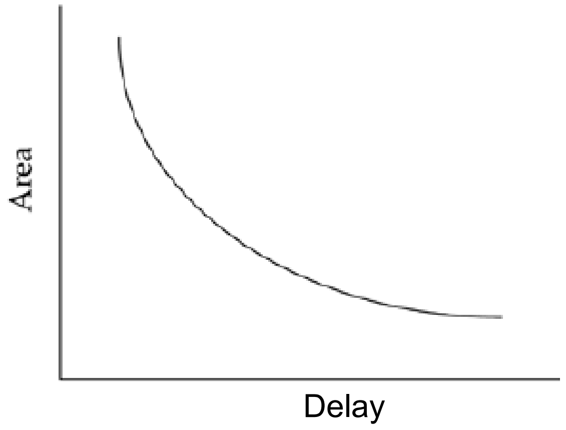
- What is Cell Characterization?

  - Functionality of the cell
  - Area of the cell layout
  - Timing information about the cell
  - Power information about the cell

- Cells are characterized by the fab vendors.

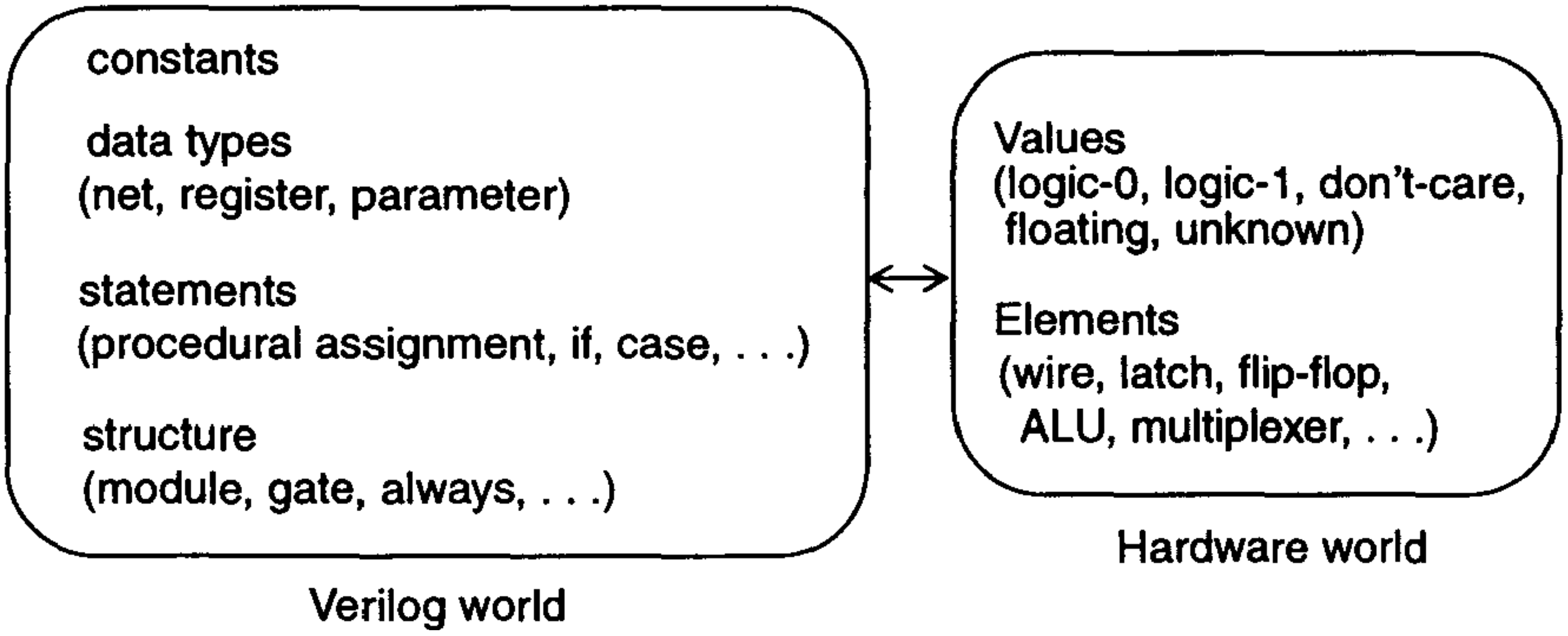
Design Constraints

- Timing
  - The circuit must meet certain timing requirements –static timing analyzer.
- Area
  - The area of the final layout must not exceed a limit
- Power
  - The power dissipation in the circuit must not exceed a threshold.
- Area Vs Timing
  - The design has to be parallelized for better timing and hence more area.



What is Synthesis?

- Figure shows the basic elements of Verilog HDL and the elements used in hardware.


constants
data types
(net, register, parameter)
statements
(procedural assignment, if, case, . . .)
structure
(module, gate, always, . . .)
Verilog world
Values
(logic-0, logic-1, don't-care, floating, unknown)
Elements
(wire, latch, flip-flop, ALU, multiplexer, . . .)
Hardware world

The two worlds of synthesis.

What is Synthesis?

A mapping mechanism or a construction mechanism has to be provided that translates the Verilog HDL elements into their corresponding hardware elements.

Questions to ask are:

- How does a data type translate to hardware?
- How are constants mapped to logic values?
- How are statements translated to hardware?

Synthesis in a Design Process

- Verilog HDL is a hardware description language that allows a designer to model a circuit at different levels of abstraction, ranging from the gate level, register-transfer level, behavioral level to the algorithmic level.
- Thus, a circuit can be described in many different ways, not all of which may be synthesizable.
- Compounding this is the fact that Verilog HDL was designed primarily as a simulation language and not as a language for synthesis.

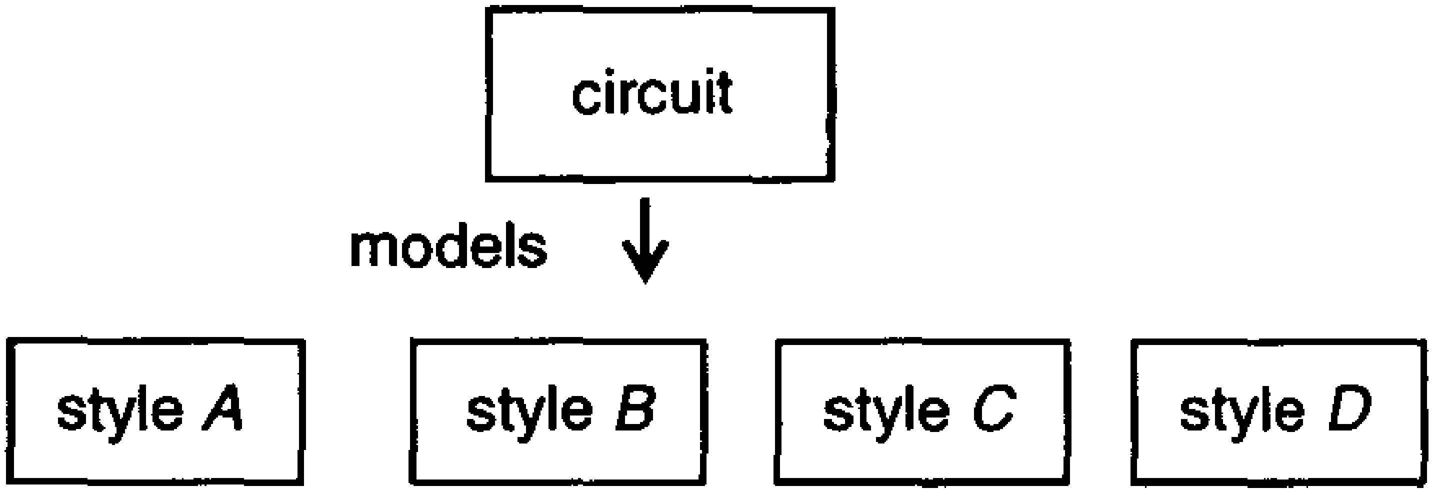
Synthesis in a Design Process

- Consequently, there are many constructs in Verilog HDL that have no hardware counterpart, for example, the $display system call.

- Also there is no standardized subset of Verilog HDL for register-transfer level synthesis.

- Because of these problems, different synthesis systems support different Verilog HDL subsets for synthesis.

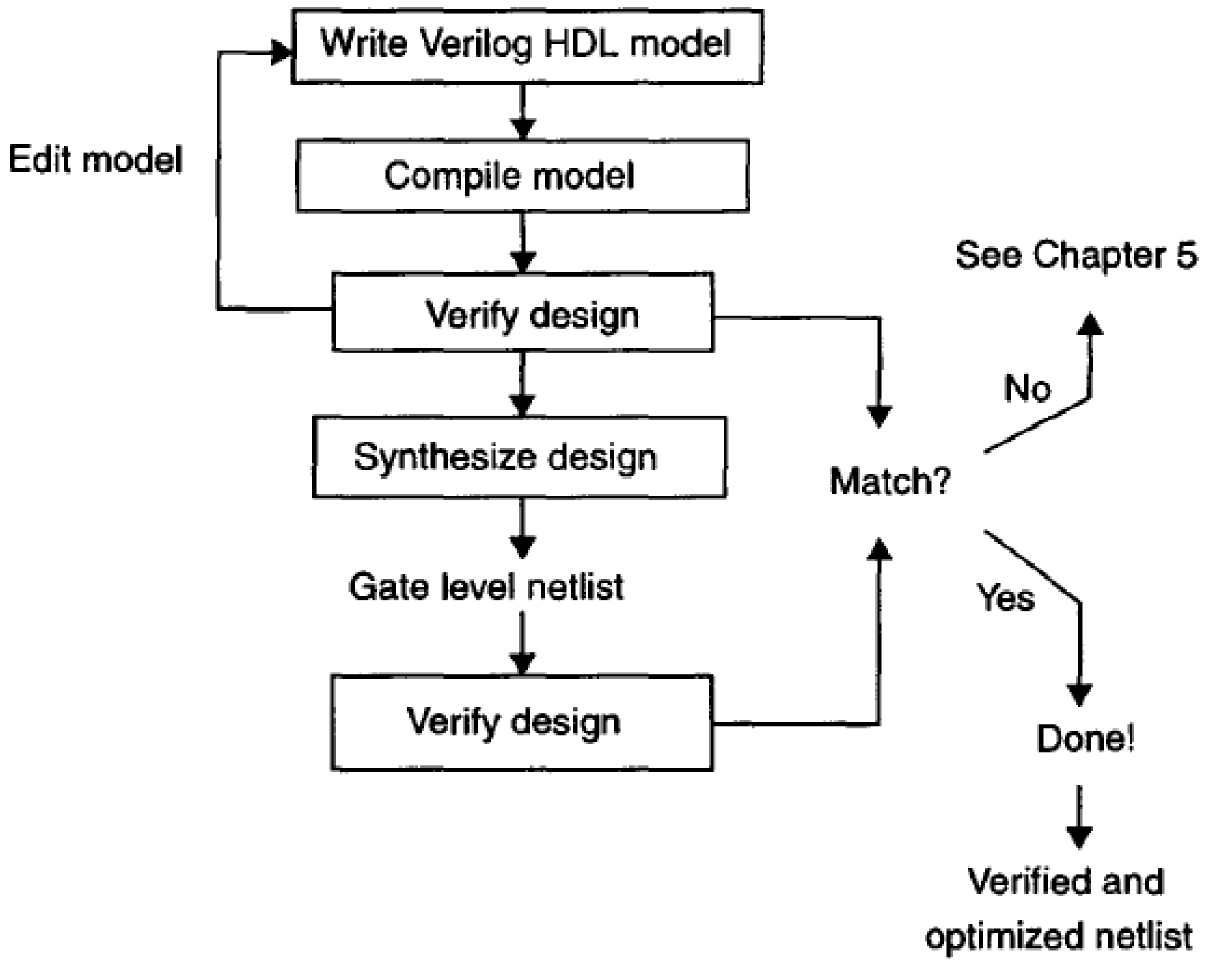
Synthesis in a Design Process



Same behavior, different styles

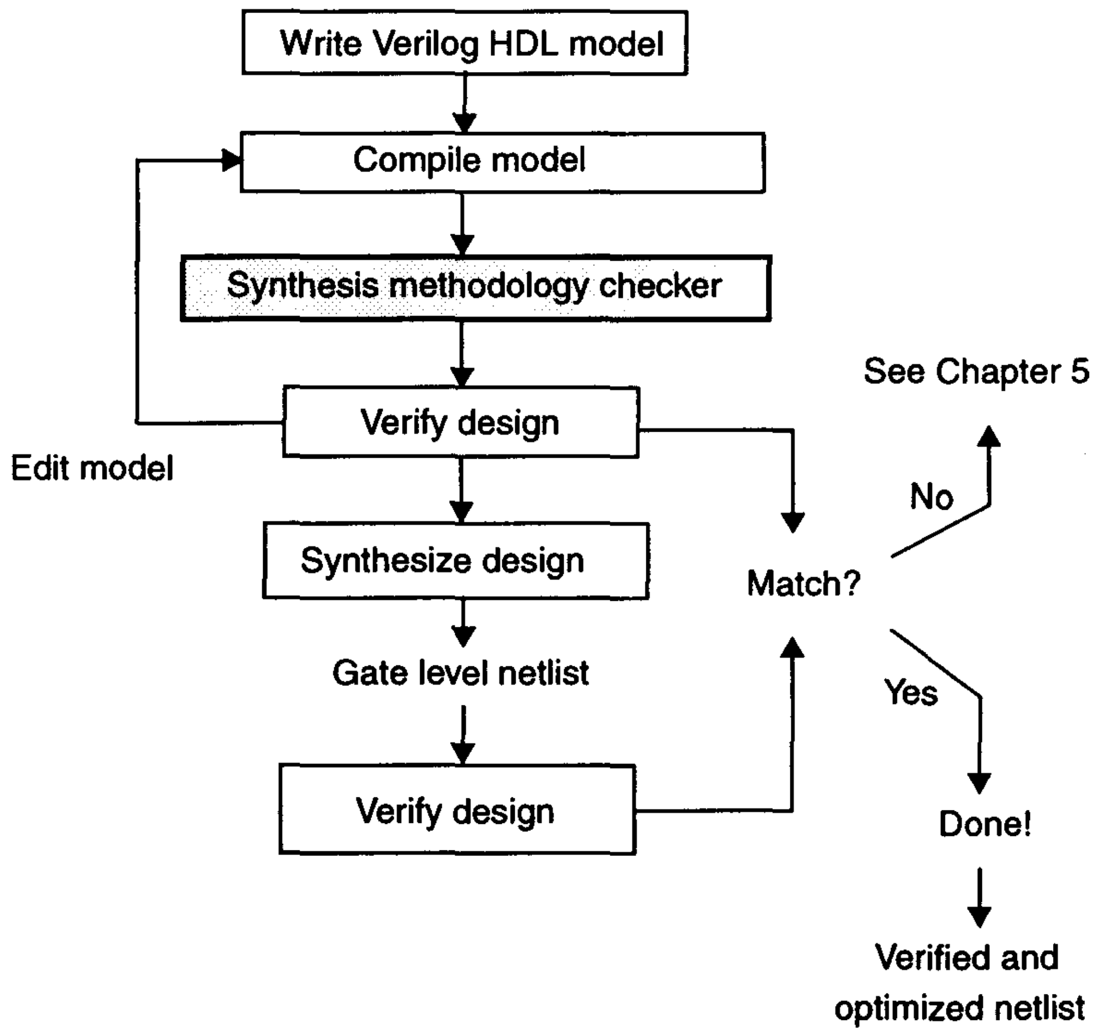
- A synthesis system that supports synthesis of styles *A* and *B* may not support that of style C.
- This implies that typically synthesis models are non-portable across different synthesis systems. Style *D* may not be synthesizable at all.

Typical Design Process



Typical design process.

New Design Process



New design process.

Verilog HDL Synthesis (VHS)

- For the purpose of logic synthesis, designs are written in an HDL at a register transfer level (RTL).
- The term RTL is used for an HDL description style that utilizes a combination of data flow and behavioral constructs.
- Logic synthesis tools take the register transfer-level HDL description and convert it to an optimized gate-level netlist.

VHS: Verilog Constructs

- Not all constructs can be used when writing a description for a logic synthesis tool.

- In general, any construct that is used to define a cycle-by-cycle RTL description is acceptable to the logic synthesis tool.

- Logic synthesis ignores all timing delays specified by #<delay> construct. Therefore, pre- and post-synthesis Verilog simulation results may not match.

VHS: Verilog Constructs

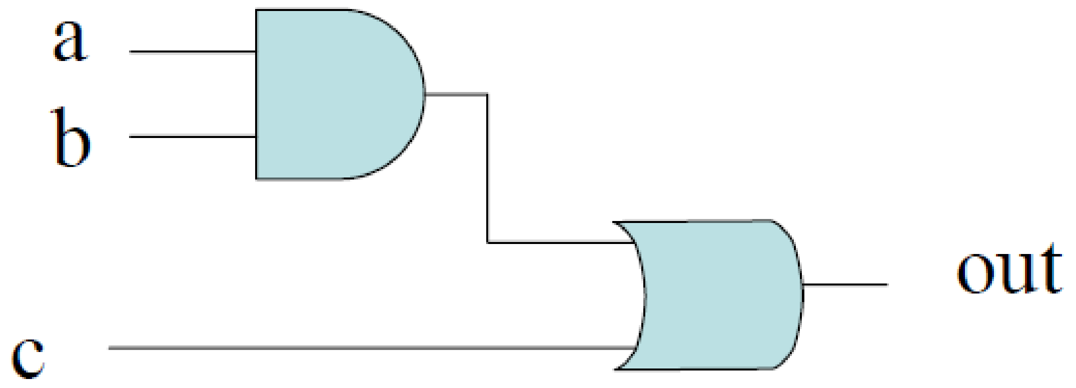
- initial construct is not supported by logic synthesis tools.
    - Instead, the designer must use a reset mechanism to initialize the signals in the circuit.
- It is recommended that all signal widths and variable widths be explicitly specified.
    - Defining un-sized variables can result in large, gate-level netlists because synthesis tools can infer unnecessary logic based on the variable definition.

Interpretation of Few Verilog Constructs

The assign statement

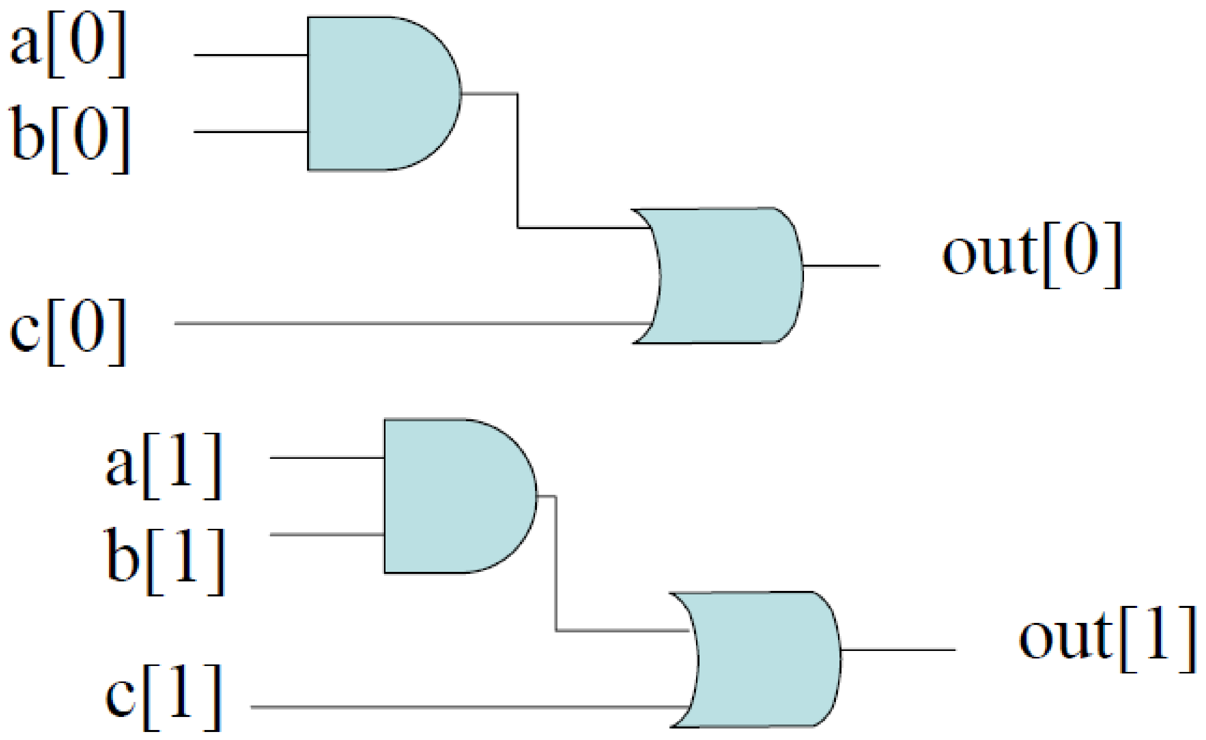
- The assign construct is the most fundamental construct used to describe combinational logic at an RTL level.

- Example: assign out = (a & b) | c ;



The assign statement

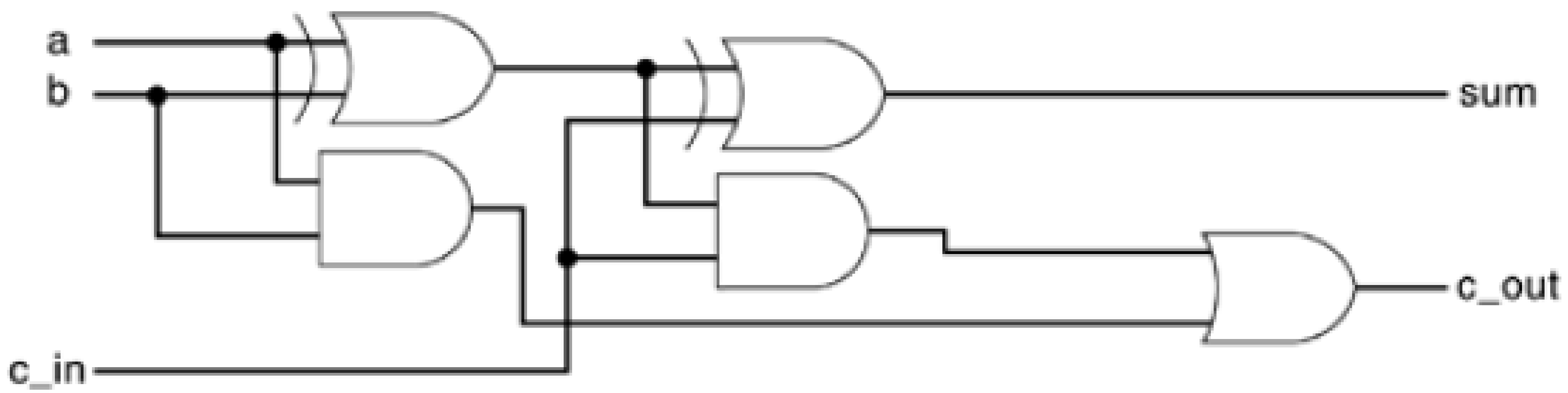
- If a, b, c, and out are 2-bit vectors [1:0]; then the above assign statement will frequently translate to two identical circuits for each bit.



The assign statement (cont'd)

- A 1 bit-full adder

```
assign {c_out, sum} = a + b + c_in;
```

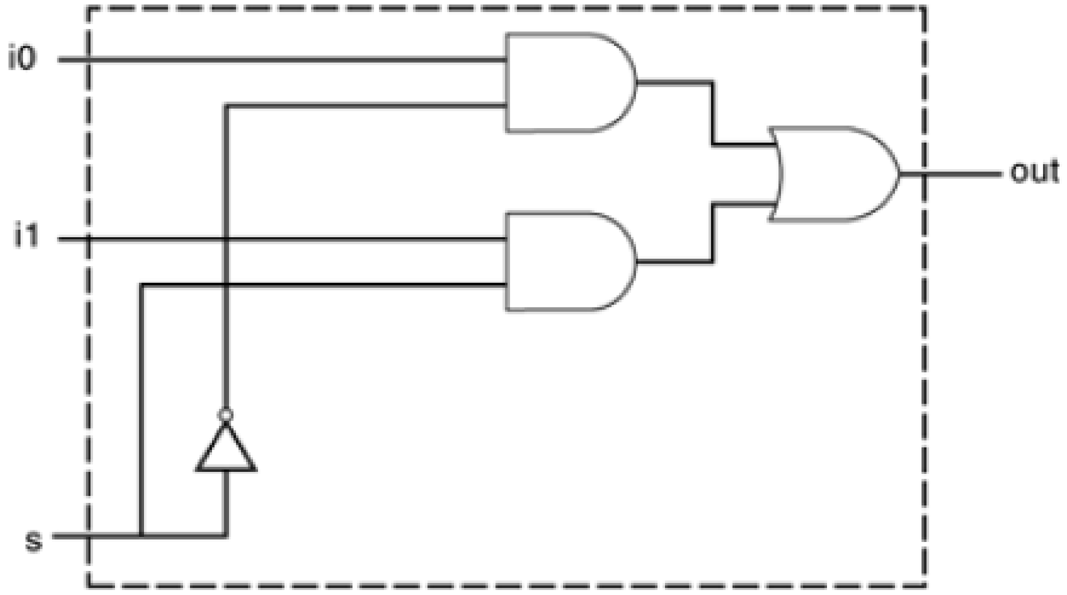


The assign statement (cont'd)

- Multiplexer

```
assign out = (s) ? i1: i0
```

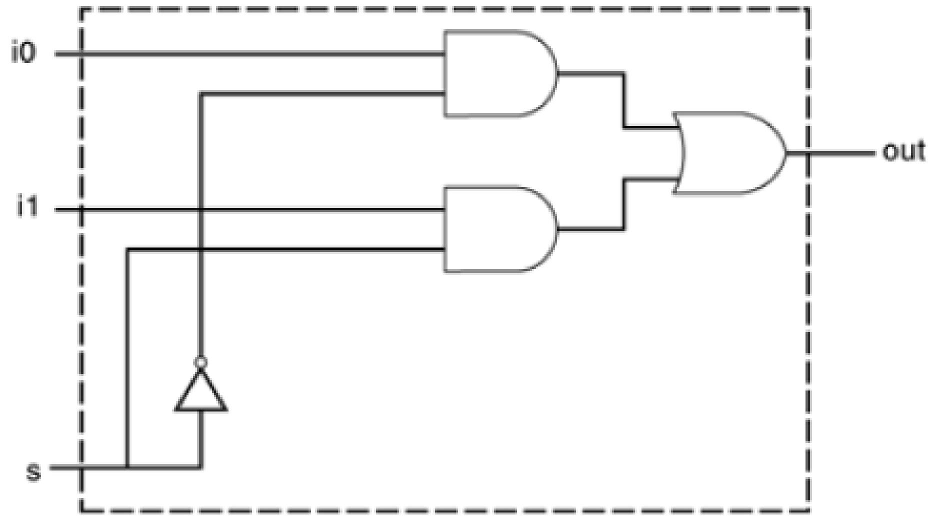
// yields a 2-to-1 multiplexer. Always the '?'construct yields a multiplexer.



The if-else statement

- Single if-else statements translates to multiplexers where the control signal is the signal or variable in the if clause.

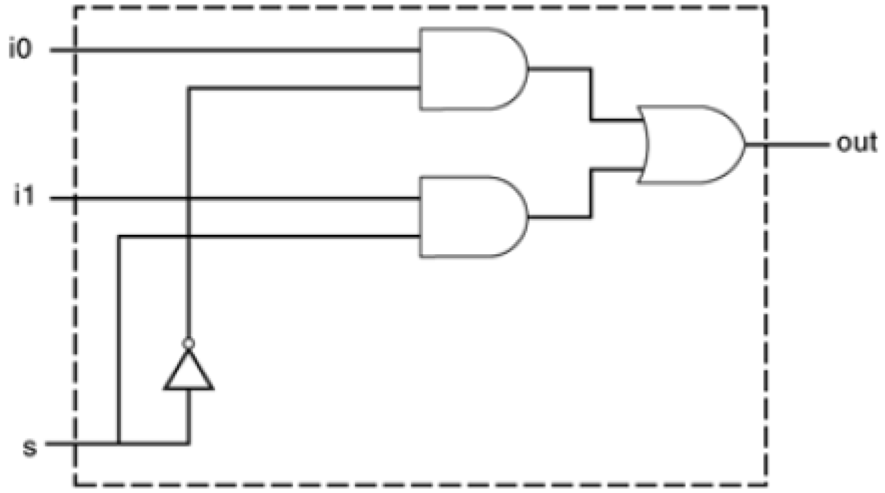
```
If (S)
        out = i1;
else
        out =i0;
```



The case statement

- The case statement also can used to infer multiplexers.

```
case (s)
        1'b0 : out = i0;
        1'b1 : out = i1;
endcase
```



- Large case statements may be used to infer large multiplexers.

for loops

- The for loops can be used to build cascaded combinational logic.

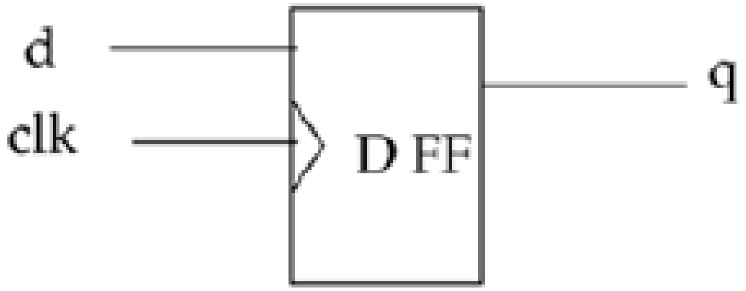
```
// the following for loop builds an 8-bit full adder

c = c_in;

for(i=0; i <=7; i = i + 1)
        {c, sum[i]} = a[i] + b[i] + c; // builds an 8-bit ripple adder

c_out = c;
```

The always statement

- The always statement can be used to infer sequential and combinational logic.

```
always @(posedge clk)
        q <= d;
```

```
always @(clk or d)
    if (clk)
        q = d;
```



- For combinational logic, the always statement must be triggered by a signal other than the clk, reset, or preset.

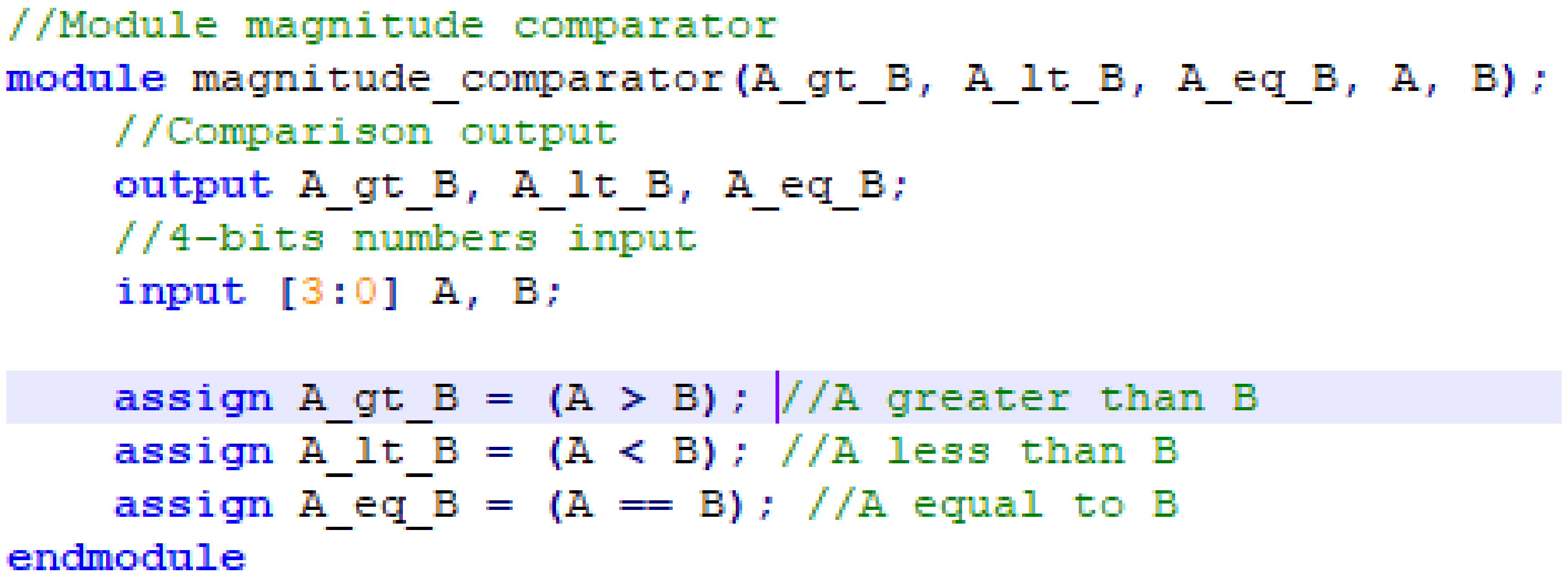
```
always @(a or b or c_in)
      {c_out, sum} = a + b + c_in;
```

An Example of RTL-to-Gates

Design Specification

- Name of module "**magnitude_comparator**"
    - Input ports –two 4-bit inputs
    - Output ports –three ports
        - A_gt_B is true if A > B
        - A_lt_B is true if A < B
        - A_eq_B is true if A = B
- The magnitude comparator circuit must be as fast as possible. Area can be compromised for speed.

RTL Description



```
//Module magnitude comparator
module magnitude_comparator(A_gt_B, A_lt_B, A_eq_B, A, B);
    //Comparison output
    output A_gt_B, A_lt_B, A_eq_B;
    //4-bits numbers input
    input [3:0] A, B;

    assign A_gt_B = (A > B); //A greater than B
    assign A_lt_B = (A < B); //A less than B
    assign A_eq_B = (A == B); //A equal to B
endmodule
```

Technology Library

//Library cells for abc_100 technology

VNAND//2-input nand gate
VAND//2-input and gate
VNOR//2-input nor gate
VOR//2-input or gate
VNOT//not gate
VBUF//buffer
NDFF//Negative edge triggered D flip-flop
PDFF//Positive edge triggered D flip-flop

Functionality, timing, area, and power dissipation information of each library cell are specified in the technology library.

Design constraints

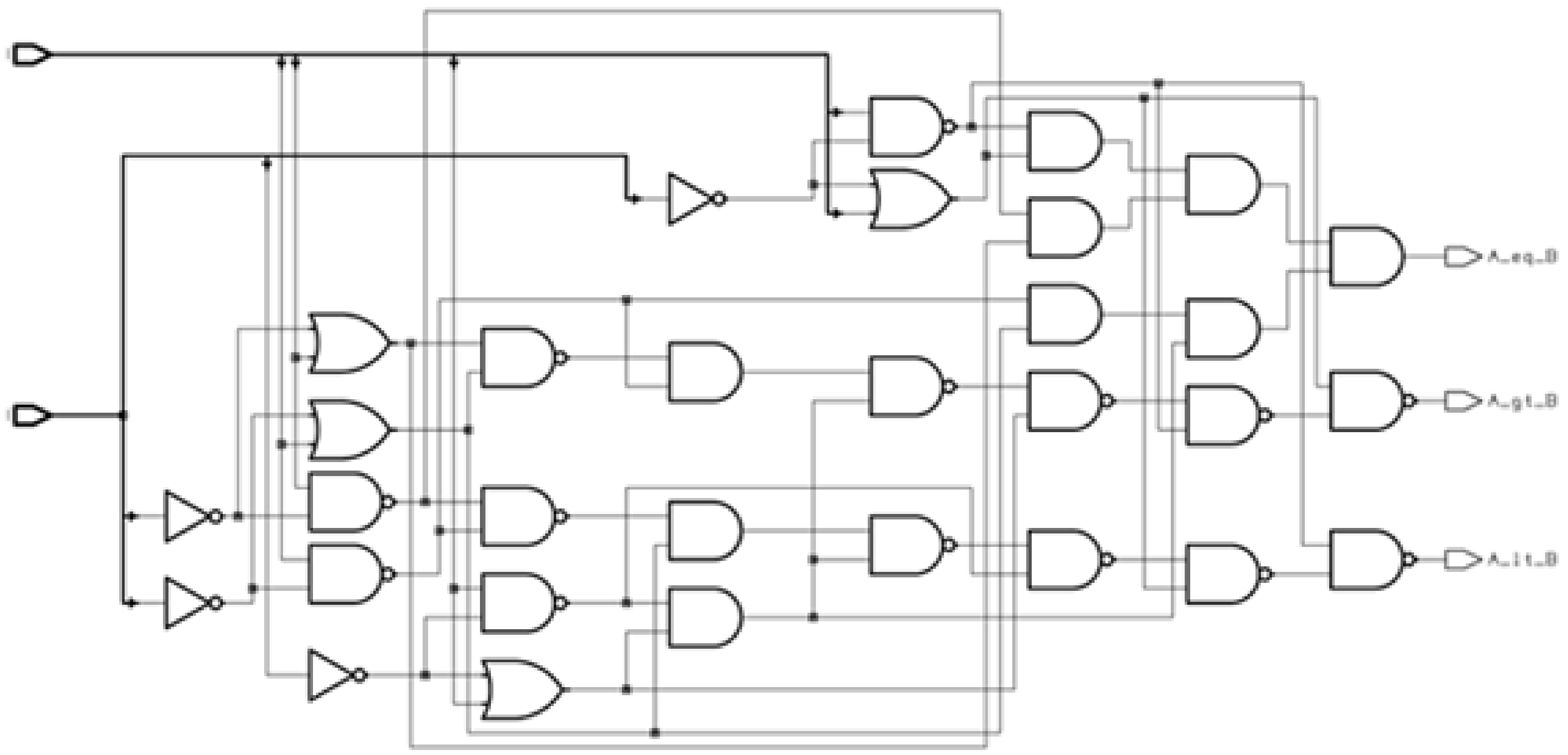
- The design should be as fast as possible for the target technology, abc_100. There are no area constraints. Thus, there is only one design constraint.

  Optimize the final circuit for fastest timing

- Logic Synthesis
  - The RTL description of the magnitude comparator is read by the logic synthesis tool.
  - The design constraints and technology library for abc_100 are provided to the logic synthesis tool.
  - The logic synthesis tool performs the necessary optimizations and produces a gate-level description optimized for abc_100 technology.

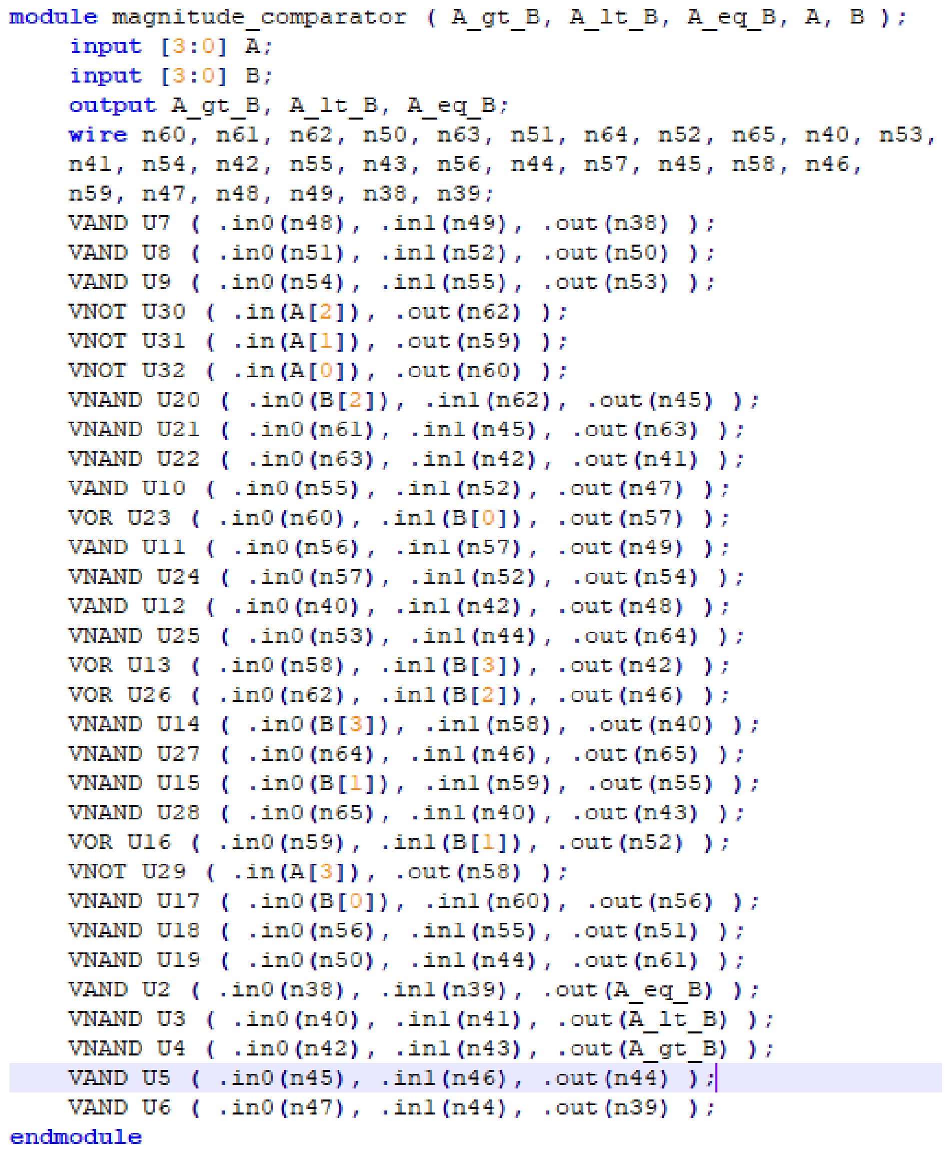
Final, Optimized, Gate-Level Description

- The logic synthesis tool produces a final, gate-level description.



Synthesized Netlist

If technology has to be changed, one has to re-execute the synthesis tool with the new Technology library and NEED NOT change the RTL.



```
module magnitude_comparator ( A_gt_B, A_lt_B, A_eq_B, A, B );
    input [3:0] A;
    input [3:0] B;
    output A_gt_B, A_lt_B, A_eq_B;
    wire n60, n61, n62, n50, n63, n51, n64, n52, n65, n40, n53,
    n41, n54, n42, n55, n43, n56, n44, n57, n45, n58, n46,
    n59, n47, n48, n49, n38, n39;
    VAND U7 ( .in0(n48), .in1(n49), .out(n38) );
    VAND U8 ( .in0(n51), .in1(n52), .out(n50) );
    VAND U9 ( .in0(n54), .in1(n55), .out(n53) );
    VNOT U30 ( .in(A[2]), .out(n62) );
    VNOT U31 ( .in(A[1]), .out(n59) );
    VNOT U32 ( .in(A[0]), .out(n60) );
    VNAND U20 ( .in0(B[2]), .in1(n62), .out(n45) );
    VNAND U21 ( .in0(n61), .in1(n45), .out(n63) );
    VNAND U22 ( .in0(n63), .in1(n42), .out(n41) );
    VAND U10 ( .in0(n55), .in1(n52), .out(n47) );
    VOR U23 ( .in0(n60), .in1(B[0]), .out(n57) );
    VAND U11 ( .in0(n56), .in1(n57), .out(n49) );
    VNAND U24 ( .in0(n57), .in1(n52), .out(n54) );
    VAND U12 ( .in0(n40), .in1(n42), .out(n48) );
    VNAND U25 ( .in0(n53), .in1(n44), .out(n64) );
    VOR U13 ( .in0(n58), .in1(B[3]), .out(n42) );
    VOR U26 ( .in0(n62), .in1(B[2]), .out(n46) );
    VNAND U14 ( .in0(B[3]), .in1(n58), .out(n40) );
    VNAND U27 ( .in0(n64), .in1(n46), .out(n65) );
    VNAND U15 ( .in0(B[1]), .in1(n59), .out(n55) );
    VNAND U28 ( .in0(n65), .in1(n40), .out(n43) );
    VOR U16 ( .in0(n59), .in1(B[1]), .out(n52) );
    VNOT U29 ( .in(A[3]), .out(n58) );
    VNAND U17 ( .in0(B[0]), .in1(n60), .out(n56) );
    VNAND U18 ( .in0(n56), .in1(n55), .out(n51) );
    VNAND U19 ( .in0(n50), .in1(n44), .out(n61) );
    VAND U2 ( .in0(n38), .in1(n39), .out(A_eq_B) );
    VNAND U3 ( .in0(n40), .in1(n41), .out(A_lt_B) );
    VNAND U4 ( .in0(n42), .in1(n43), .out(A_gt_B) );
    VAND U5 ( .in0(n45), .in1(n46), .out(n44) );
    VAND U6 ( .in0(n47), .in1(n44), .out(n39) );
endmodule
```

What after Synthesis?

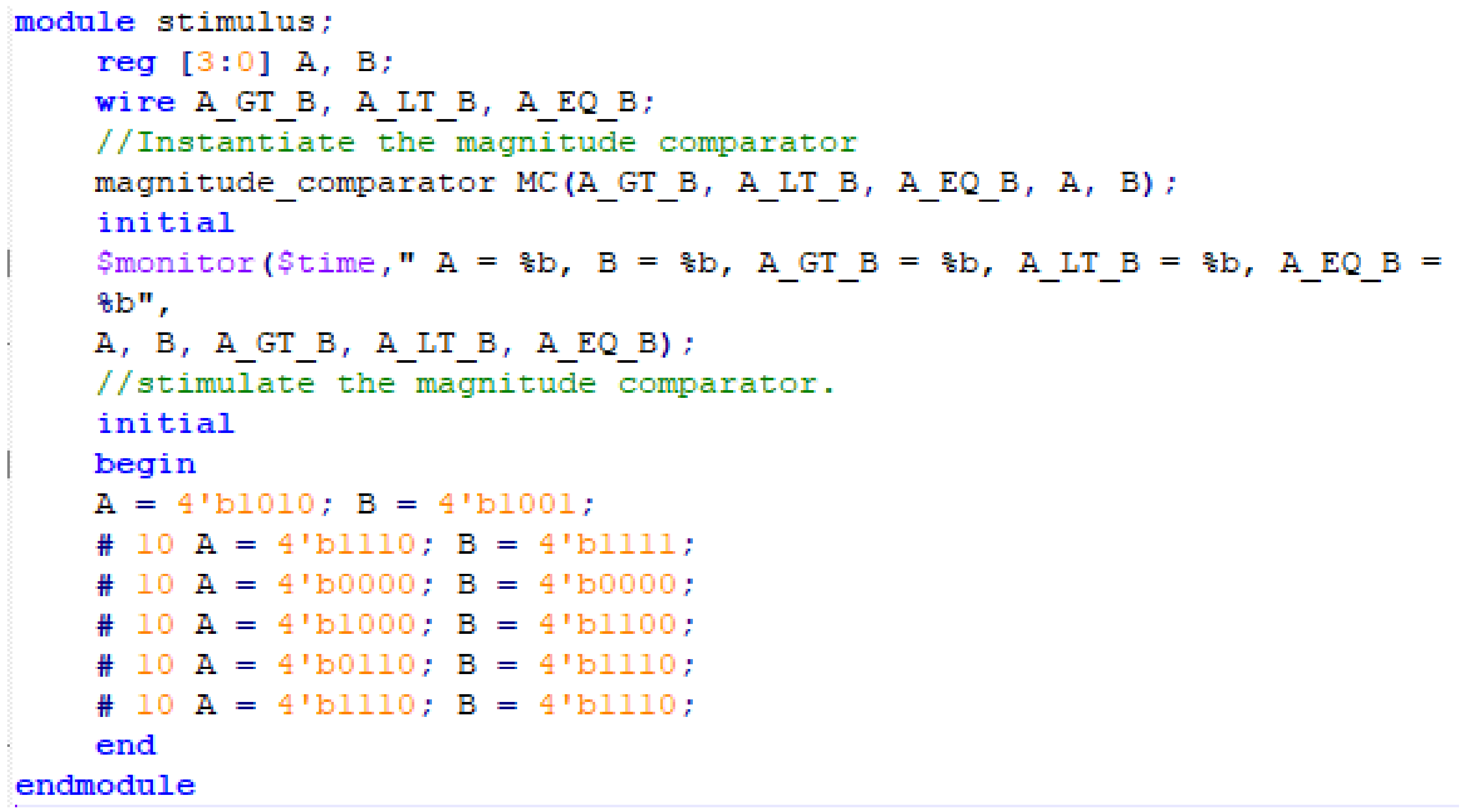
- The gate-level netlist is verified for functionality.
- The synthesis tool may not always be able to meet both timing and area requirements, if they are too stringent.
- Separate timing verification (static timing analysis) done on the gate-level netlist.
- Layout of the net-list is done
- Post-layout circuit should be checked if it meets the timing requirements and functionality.
- Finally, the chip is fabricated ☺

Functional Verification

- The gate-level net list is again a Verilog file in which the modules from the standard cell library are used.

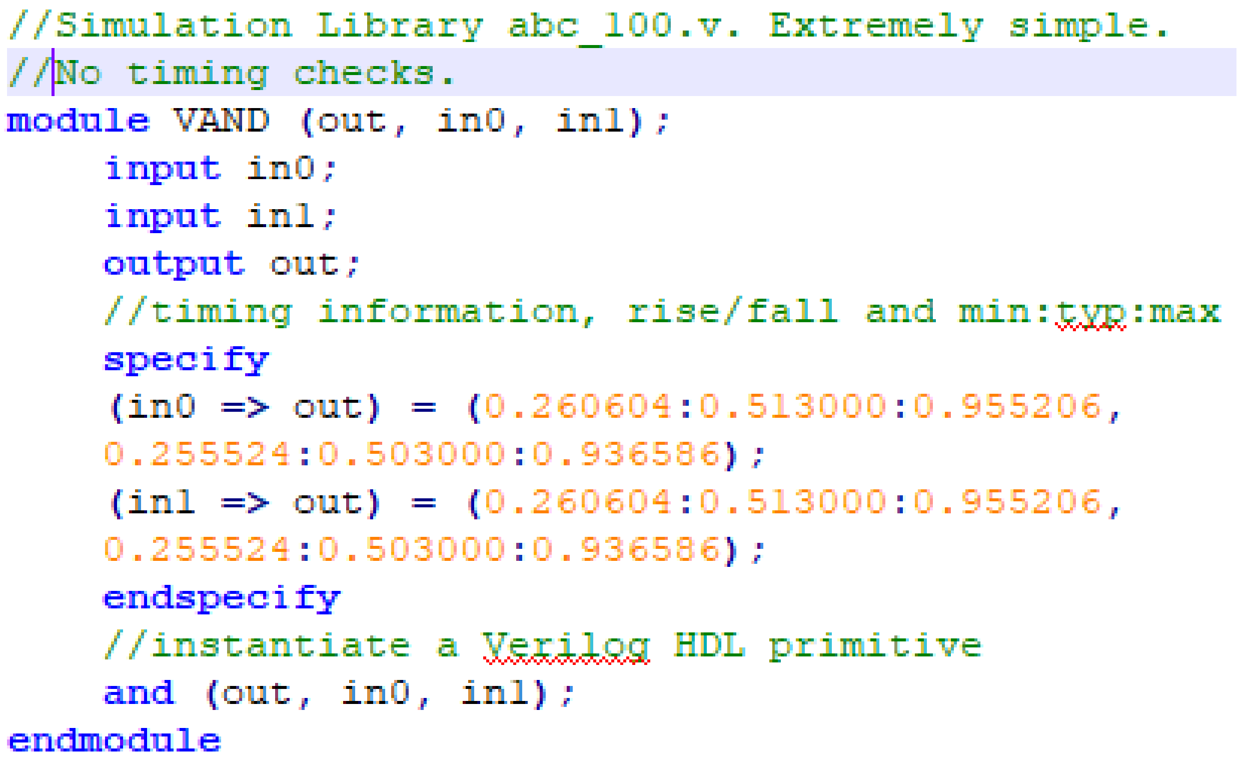
- To simulate the gate-level description, a simulation library has to be provided by the vendor.

Stimulus for Magnitude Comparator

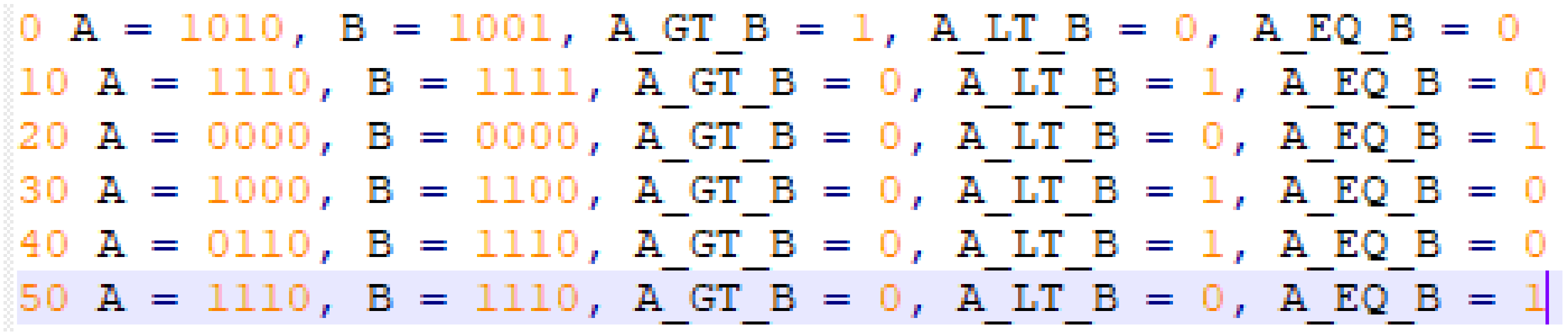


```
module stimulus;
    reg [3:0] A, B;
    wire A_GT_B, A_LT_B, A_EQ_B;
    //Instantiate the magnitude comparator
    magnitude_comparator MC(A_GT_B, A_LT_B, A_EQ_B, A, B);
    initial
    $monitor($time," A = %b, B = %b, A_GT_B = %b, A_LT_B = %b, A_EQ_B =
    %b",
    A, B, A_GT_B, A_LT_B, A_EQ_B);
    //stimulate the magnitude comparator.
    initial
    begin
    A = 4'b1010; B = 4'b1001;
    # 10 A = 4'b1110; B = 4'b1111;
    # 10 A = 4'b0000; B = 4'b0000;
    # 10 A = 4'b1000; B = 4'b1100;
    # 10 A = 4'b0110; B = 4'b1110;
    # 10 A = 4'b1110; B = 4'b1110;
    end
endmodule
```

Simulation Library



```
//Simulation Library abc_100.v. Extremely simple.
//No timing checks.
module VAND (out, in0, in1);
    input in0;
    input in1;
    output out;
    //timing information, rise/fall and min:typ:max
    specify
    (in0 => out) = (0.260604:0.513000:0.955206,
    0.255524:0.503000:0.936586);
    (in1 => out) = (0.260604:0.513000:0.955206,
    0.255524:0.503000:0.936586);
    endspecify
    //instantiate a Verilog HDL primitive
    and (out, in0, in1);
endmodule
```



```
0 A = 1010, B = 1001, A_GT_B = 1, A_LT_B = 0, A_EQ_B = 0
10 A = 1110, B = 1111, A_GT_B = 0, A_LT_B = 1, A_EQ_B = 0
20 A = 0000, B = 0000, A_GT_B = 0, A_LT_B = 0, A_EQ_B = 1
30 A = 1000, B = 1100, A_GT_B = 0, A_LT_B = 1, A_EQ_B = 0
40 A = 0110, B = 1110, A_GT_B = 0, A_LT_B = 1, A_EQ_B = 0
50 A = 1110, B = 1110, A_GT_B = 0, A_LT_B = 0, A_EQ_B = 1
```

Modeling Tips for Logic Synthesis

- The RTL design code influences the final gate-level netlist.
- Trade-off between level of design abstraction and control over the structure of the logic synthesis output.
- **Verilog Coding**
  - Use meaningful names for Signals and Variables
  - Avoid mixing positive and negative edge-triggered flip-flops.
    - May introduce inverters and buffers causing clock skews

Verilog Coding

- Use basic building blocks rather than using continuous assign statements

- More complex the logic, less optimize the netlist is.

Verilog Coding

- Basic building blocks are better optimized by synthesis tools and also gives symmetric design.

- Lesser abstraction and degrades simulator performance

- Multiplexers may be used rather than if-else, case, at possible places

- Use parentheses to optimize logic structures
  - a = b + c + d + e; //May be three adders in series
  - a = (b + c) + (d + e); // Three adders in two stages

Verilog Coding

- Use arithmetic operators *, / and modulo rather than designing building blocks for them.
  - The operators are technology independent and will be substituted by the optimized units by synthesizer.
- These optimized units may take long for the RTL designer to code in a HDL like Verilog.

Verilog Coding

- Multiple assignments to the same variable

```
always @(posedge clk)
    if (load1)
          q <= a1;

always @(posedge clk)
    if (load2)
          q <= a2;
```

- Synthesis tools infers two flip-flops and ‘and’-es them together.

Verilog Coding

- If-else or case statements should be coded explicitly.

```
always @(control or a)
      if (control)
            out <= a; //latch is inferred

always @(control or a)
      if (control)
            out <= a;
      else
            out <= b; //multiplexer is inferred
```

- Similarly, for case statements, all possible branches, including the “default” statement must be specified.

References

- Chapter 14, Verilog HDL by Samir Palnitkar
- Chapter 1, Verilog HDL Synthesis by J. Bhaskar

Thank you