KYUNGPOOK NATIONAL UNIVERSITY

COMP311-1 Logic Circuit Design Chap. 14

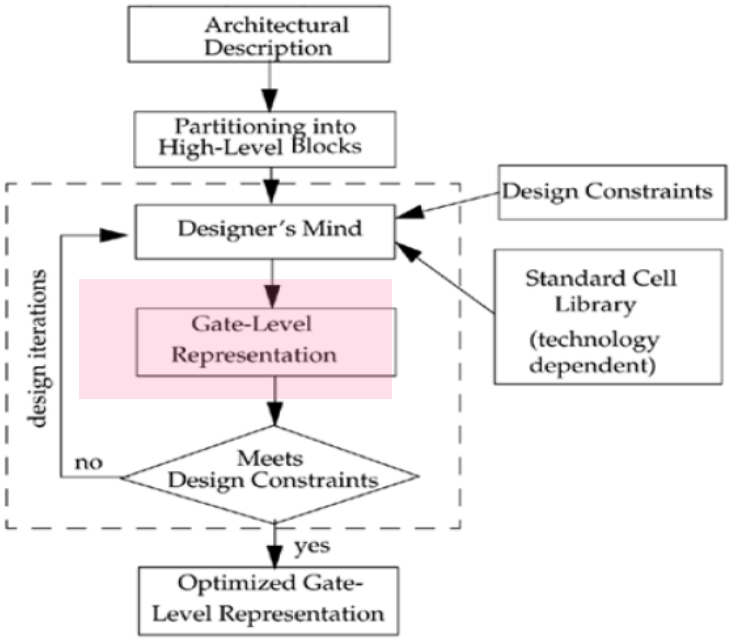
Instructor: Taigon Song (송대건)

2019 Fall

14.1 What is Logic Synthesis?

- The process of converting a high-level description of the design into an optimized gate-level representation
  - w/ given std. cell library
  - w/ given design constraints
- Previously done inside the designer's mind
- In our class, it would be FPGA perhaps

Figure 14-1. Designer's Mind as the Logic Synthesis Tool

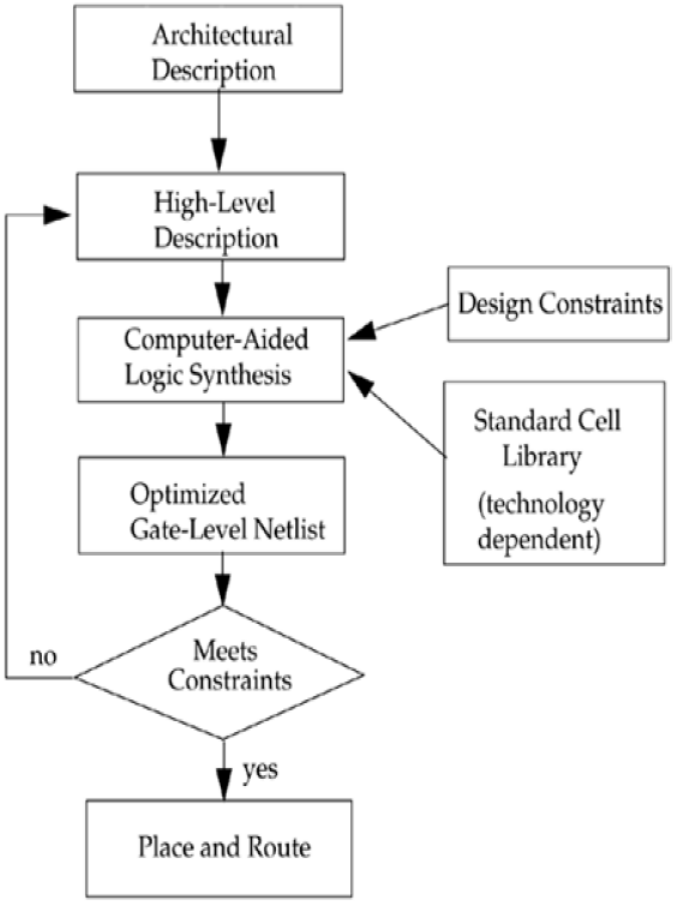


14.1 What is Logic Synthesis?

- These days, every process is automated

- Don’t forget, these processes were done by hand in the early days
  - http://www.intel4004.com/4004_original_schematics.htm

Figure 14-2. Basic Computer-Aided Logic Synthesis Process



14.2 Impact of Logic Synthesis

Problems of "hand-automated designs"

- Revolutionized the digital design industry by significantly improving productivity and by reducing design cycle time
- Limitations of non-automated (hand-written) design process
  - Prone to human error (one bug could mean thousands of gates for redesign)
  - Designer is never sure that the constraints are met until the gate-lv implementation is completed and tested
  - Very time consuming high-lv design to gates
  - Significant turnaround time
  - What-if scenarios hard to verify (specific scenarios hard to verify)
  - Each designer implement design blocks differently
    - Smaller blocks can be optimal, but it may not in the overall design
  - Timing/area/power are fabrication-technology specific
    - An update in library would ask redesign of the entire circuit
  - Designs are technology-specific

14.2 Impact of Logic Synthesis

Solutions by automated logic synthesis tools

- Solutions provided by logic synthesis tools
  - High-level design less prone to human error
  - High-level design done w/o significant concern about design constraints
  - Fast turnaround time
  - What-if scenarios (specific scenarios) easy to verify
  - A common design style (compared to designer specific designs)
  - Bug fixes relatively less critical to total design
  - Technology-independent design
  - Design reuse possible in technology-independent descriptions

14.3 Verilog HDL Synthesis

14.3.1 Verilog Constructs

- For logic synthesis, designs should be written in HDL at a register transfer level (RTL)
  - Either in VHDL or in Verilog
- In general, any construct that is used to define a cycle-by-cycle RTL description is acceptable for logic synthesis
  - “initial” ignored during synthesis
  - Timing-related statements all ignored during synthesis
  - Loops not recommended for synthesis
  - Signal/variable widths should be explicitly specified
    - Unnecessary logic could be synthesized when not clearly defined

14.3 Verilog HDL Synthesis

14.3.1 Verilog Constructs

- Verilog HDL constructs for Logic Synthesis

Table 14-1. Verilog HDL Constructs for Logic Synthesis

Construct Type	Keyword or Description	Notes
ports	input, inout, output	
parameters	parameter	
module definition	module	
signals and variables	wire, reg, tri	Vectors are allowed
instantiation	module instances, primitive gate instances	E.g., mymux m1(out, i0, i1, s); E.g., nand (out, a, b);
functions and tasks	function, task	Timing constructs ignored
procedural	always, if, then, else, case, casex, casez	initial is not supported
procedural blocks	begin, end, named blocks, disable	Disabling of named blocks allowed
data flow	assign	Delay information is ignored
loops	for, while, forever,	while and forever loops must contain @(posedge clk) or @(negedge clk)

14.3 Verilog HDL Synthesis

14.3.2 Verilog Operators

- Almost all operators in Verilog supported for logic synthesis
  - "===" and "!==" are not allowed
    - ('x' and 'z' are not so much meaningful in actual designs)
  - Always recommended to use parentheses to group logic

Table 14-2. Verilog HDL Operators for Logic Synthesis

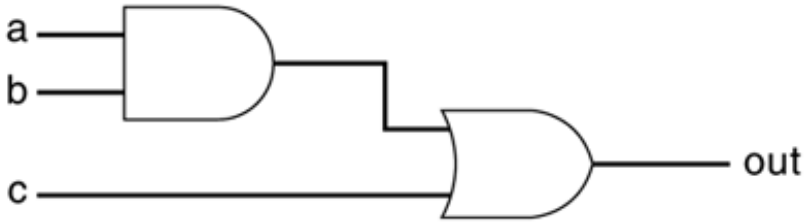
Operator Type	Operator Symbol	Operation Performed
Arithmetic	*	multiply
	/	divide
	+	add
	-	subtract
	%	modulus
	+	unary plus
	-	unary minus
Logical	!	logical negation
	&&	logical and
	||	logical or
Relational	>	greater than
	<	less than
	>=	greater than or equal
	<=	less than or equal
Equality	==	equality
	!=	inequality
Bit-wise	~	bitwise negation
	&	bitwise and
	|	bitwise or
	^	bitwise ex-or
	^~ or ~^	bitwise ex-nor
Reduction	&	reduction and
	~&	reduction nand
	|	reduction or
	~|	reduction nor
	^	reduction ex-or
	^~ or ~^	reduction ex-nor
Shift	>>	right shift
	<<	left shift
	>>>	arithmetic right shift
	<<<	arithmetic left shift
Concatenation	{ }	concatenation
Conditional	?:	conditional

14.3 Verilog HDL Synthesis

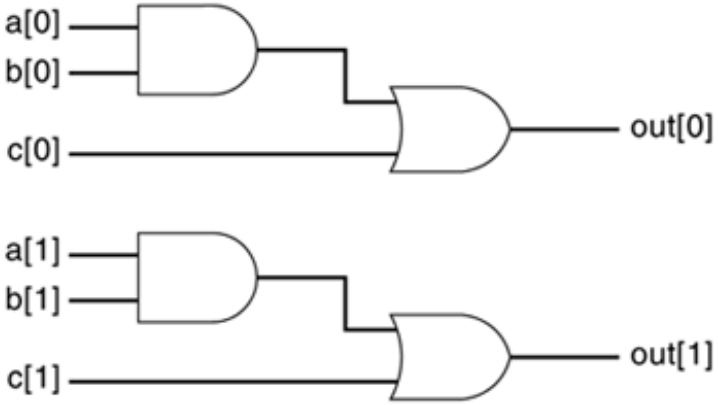
14.3.3 Interpretation of a Few Verilog Constructs

- The assign statement

```
assign out = (a & b) | c;
```



- If [1:0] out = (a & b) | c;

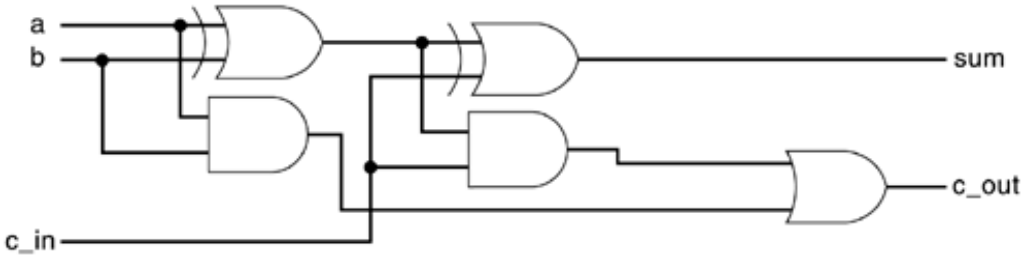


14.3 Verilog HDL Synthesis

14.3.3 Interpretation of a Few Verilog Constructs

- The assign statement

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



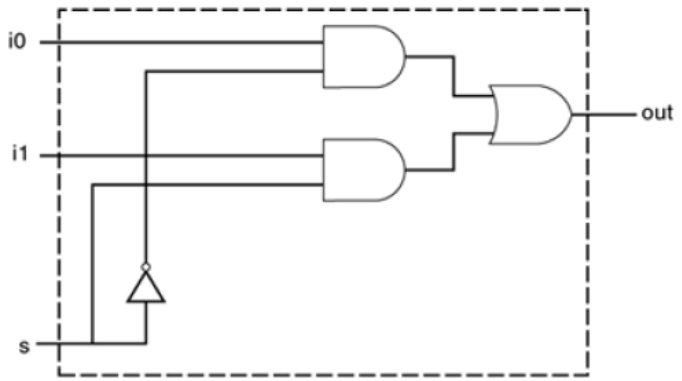
  - E.g., adders are some common/frequently used blocks in HDL
- Conditional operator, if/else statement, case statement

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

```
if(s)
    out = i1;
else
   out = i0;
```

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

Figure 14-3. Multiplexer Description



14.3 Verilog HDL Synthesis

14.3.3 Interpretation of a Few Verilog Constructs

- “for” loops
  - Can be used to build cascaded combinational logic

```
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;
```

- “always” statement

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

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

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

- “function” statement

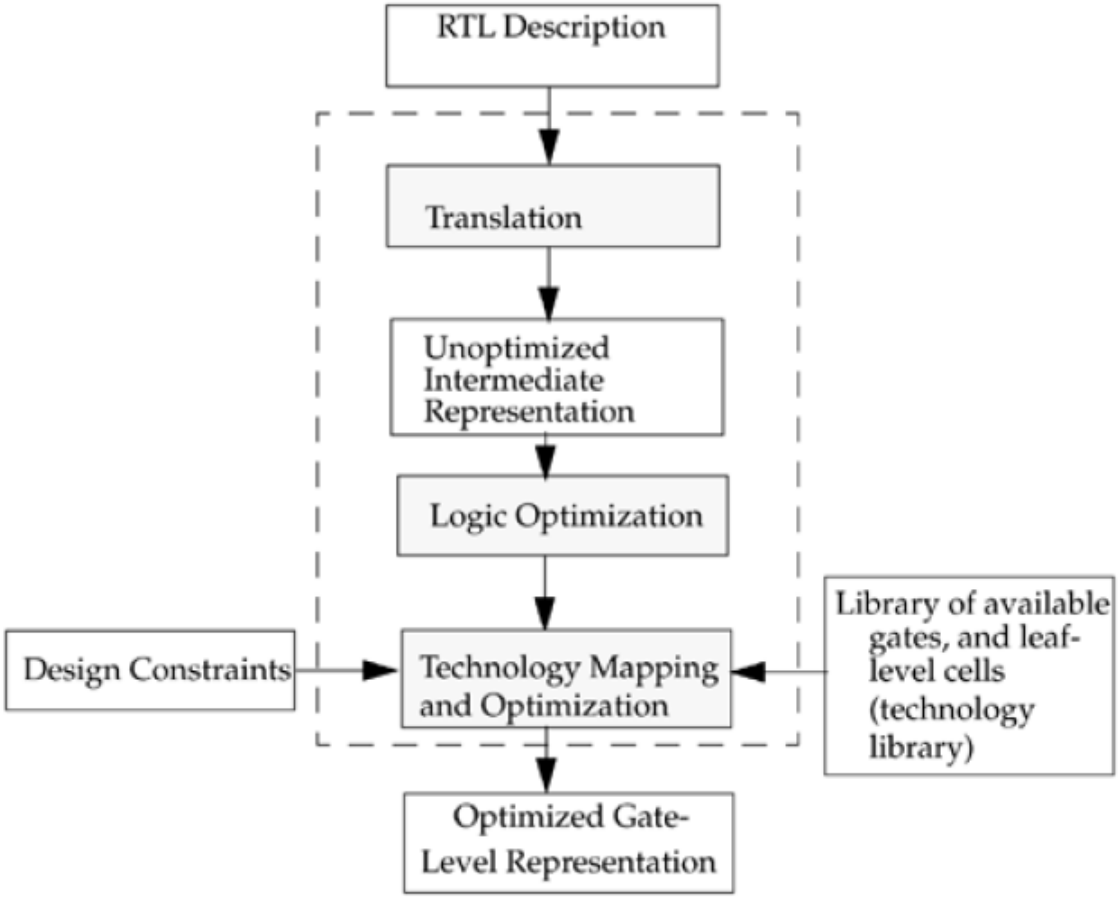
```
function [4:0] fulladd;
input [3:0] a, b;
input c_in;
begin
    fulladd = a + b + c_in; //bit 4 of fulladd for carry, bits[3:0] for
sum.
end
endfunction
```

14.4 Synthesis Design Flow

시험 안나옴 PASS

14.4.1 RTL to Gates

Figure 14-4. Logic Synthesis Flow from RTL to Gates



14.4 Synthesis Design Flow

14.4.1 RTL to Gates

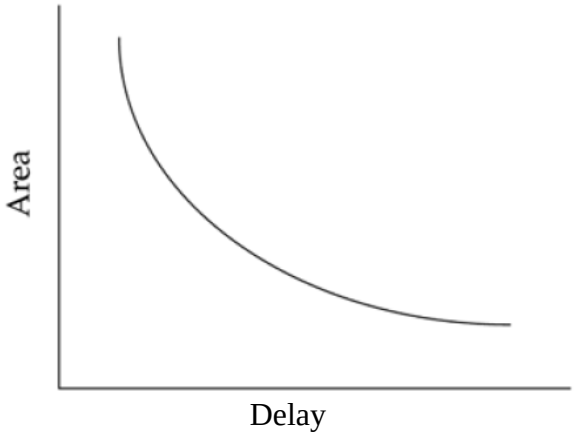
- RTL description
  - Designer describes the design at a high-level by using RTL constructs
- Translation
  - RTL converted by logic synthesis tool to an unoptimized intermediate, internal representation
- Unoptimized intermediate representation
  - Design represented internally by the synthesis tool in terms of internal data structures
- Logic optimization
  - Logic optimized to remove redundant logic
  - Using technology-independent Boolean logic
- Technology mapping and optimization
  - Design mapped to the desired target technology
  - so-called “technology mapping”

14.4 Synthesis Design Flow

14.4.1 RTL to Gates

- Technology library
  - Contains cells provided by the foundary
  - From AND, OR, XOR, XNOR gates to MUXes, ALUs, and flip-flops
  - Contains the following information:
    - Functionality
    - Area
    - Timing information
    - Power information
- Design constraints
  - Timing
  - Area
  - Power

Figure 14-5. Area vs. Timing Trade-off (cell level)



14.4 Synthesis Design Flow

14.4.1 RTL to Gates

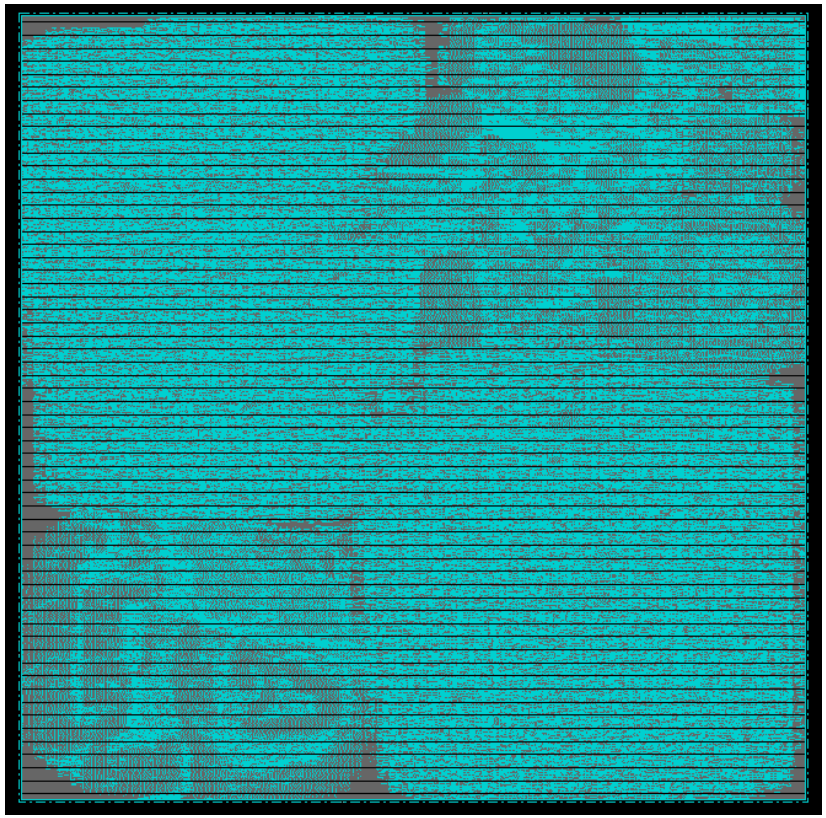
- Optimized gate-level description
  - If the final gate-lv netlist fulfils its design goal, it is handled for final layout

- Three points to note about synthesis flow
  - Technology libraries are normally provided by the foundry
  - Given the technology library, the designer's goal is to provide the best RTL for logic design (+design constraints)
  - For submicron designs, interconnect delays are becoming a dominating factor in the overall delay

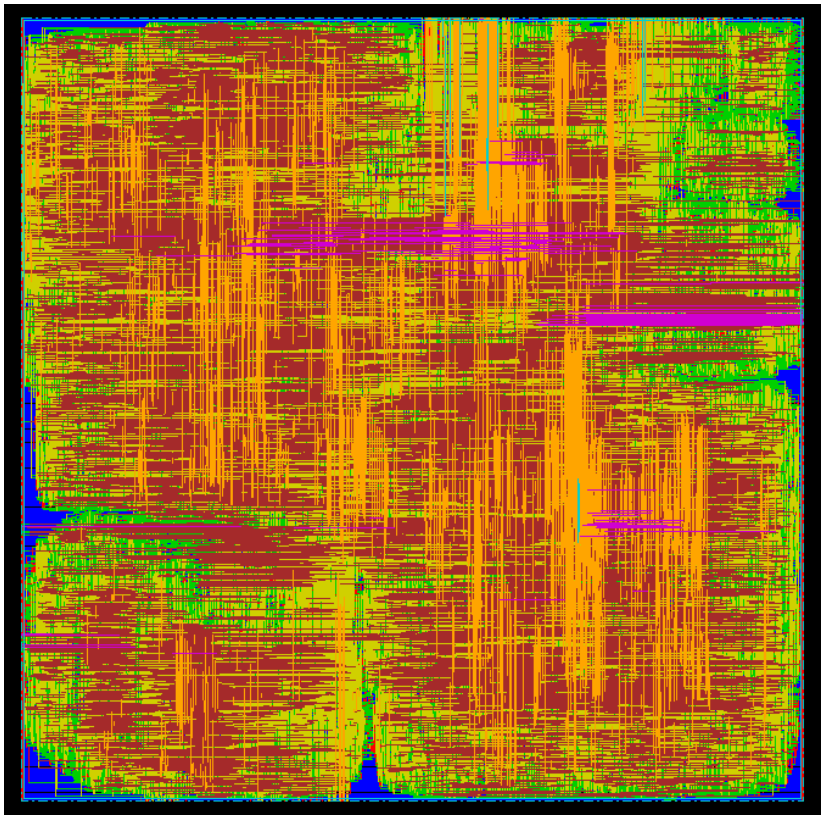
Layout of FGU

A Module in OpenSPARC T1

- In 90nm Technology
  - Cell count: 111k



1676um



14.4 Synthesis Design Flow

14.4.2 An Example of RTL-to-Gates

- Design specification
  - 4-bit magnitude comparator
  - To be designed as fast as possible
    - Two 4-bit inputs A and B
    - Three one-bit outputs
- RTL description
- Technology library

Example 14-1 RTL for Magnitude Comparator

```
//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
```

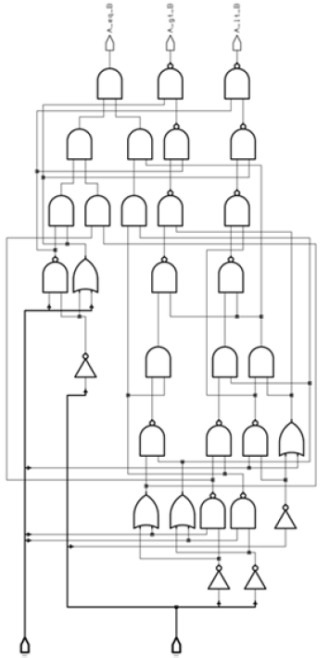
```
//Library cells for abc_100 technology

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
```

14.4 Synthesis Design Flow

14.4.2 An Example of RTL-to-Gates

- Design constraints
  - Optimize the final circuit for fastest timing
- Logic synthesis
- Final, Optimized, Gate-level description

Figure 14-6. Gate-Level Schematic for the Magnitude Comparator



Example 14-2 Gate-Level Description for the Magnitude Comparator

```
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
```

14.5 Verification of Gate-Level Netlist

14.5.1 Functional Verification

- Gate-lv netlist produced by the logic synthesis tool must be verified
  - For functionality
  - For timing
- Perform testbench simulation on both RTL and the synthesized netlist

Example 14-3 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
```

14.5 Verification of Gate-Level Netlist

14.5.1 Functional Verification

- For synthesized netlist, a timing library must be provided for simulation

Example 14-4 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
...
//All library cells will have corresponding module definitions
//in terms of Verilog primitives.
...
```

14.6 Modeling Tips for Logic Synthesis

14.6.1 Verilog Coding Style

- Which is more important?
  - high-lv design abstraction vs. low-lv (gate-lv) design
- Guidelines:
  - Use meaningful names for signal names and variables
  - Avoid mixing positive and negative edge-triggered flip-flops
  - Using basic building blocks vs. continuous assign statements
    - MUXes are technology dependent
    - Statements such as if-else can synthesize unwanted latches
  - Use parentheses to optimize logic structure

```
//translates to 3 adders in series
out = a + b + c + d;

//translates to 2 adders in parallel with one final adder to sum
results
out = (a + b) + (c + d) ;
```

  - Using arithmetic operators (*,/,%) vs. designing building blocks

14.6 Modeling Tips for Logic Synthesis

14.6.1 Verilog Coding Style

- Guidelines:
  - Be careful with multiple assignments to the same variable

```
//two assignments to the same variable
always @(posedge clk)
       if(load1) q <= a1;

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

  - Define if-else or case statements explicitly
    - Branches for all possible conditions must be specified
      - Or latches pop out after synthesis

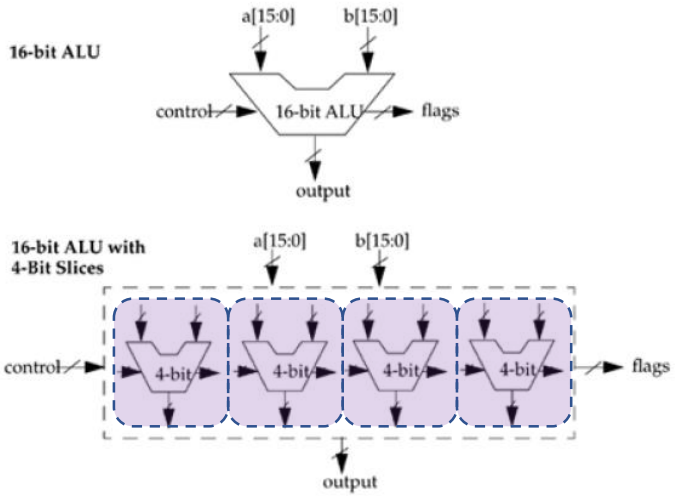
```
//latch is inferred; incomplete specification.
//whenever control = 1, out = a which implies a latch behavior.
//no branch for control = 0
always @(control or a)
    if (control)
       out <= a;

//multiplexer is inferred. complete specification for all values of
//control
always @(control or a or b)
    if (control)
        out = a;
    else
       out = b;
```

14.6 Modeling Tips for Logic Synthesis

14.6.2 Design Partitioning

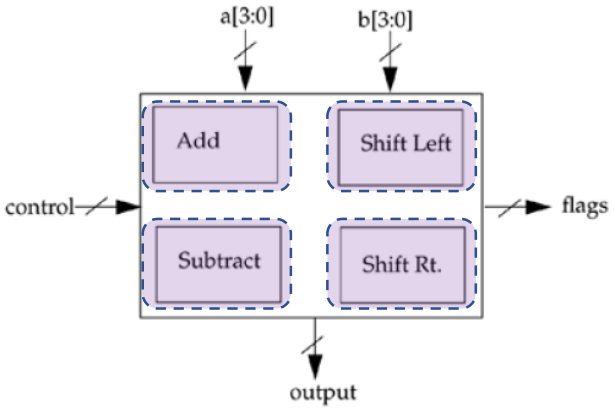
- Horizontal partitioning
  - Based on bit slices

Figure 14-7. Horizontal Partitioning of 16-bit ALU



- Vertical partitioning
  - Based on functionality

Figure 14-8. Vertical Partitioning of 4-bit ALU



14.6 Modeling Tips for Logic Synthesis

14.6.2 Design Partitioning

- Parallelizing design structure
  - For faster design implementation
    - E.g. Carry lookahead adder vs. Ripple carry adder

- Design Constraint Specification
  - Accurate specification of timing, area, power
  - Environmental parameters such as input drive strengths, output loads, input arrival times…etc

In-class Activity

Are all source codes synthesizable?

- Change the source code to make it synthesizable

```
module synpossible1(
  input [3:0] in1,
  output reg [15:0] out1
    );
  reg flag;

  always @(*) begin
    if (in1)
      flag = 1;
    else
      flag = 0;
    while (flag)
      begin
        if(in1 == 4'b0100) begin
          out1 = in1 * 30;
          flag = 0;
        end
    end
  end
endmodule
```