100-1 Under-Graduate Project
Synthesis of Combinational Logic

Speaker: Kuanyu Ho
Adviser: Prof. An-Yeu Wu
Date: 2011/10/18

Outline

- What is synthesis?
- Behavior Description for Synthesis
- Write Efficient HDL Code

What is logic synthesis

- Logic synthesis is the process of converting a high-level description of design into an optimized gate-level representation
- Logic synthesis uses standard cell library which have simple cells, such as basic logic gates like **`and`**, **`or`**, and **`nor`**, or macro cells, such as adder, muxes, memory, and special flip-flops
- The designer would first understand the architectural description. Then he/she would consider design constraints such as timing, area, testability, and power

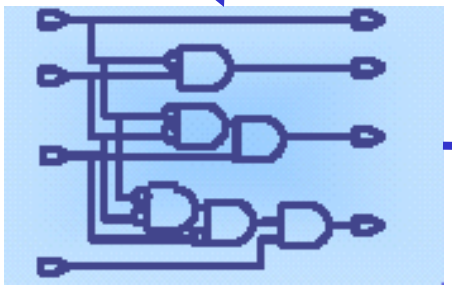
What is logic synthesis

- Synthesis = translation + optimization + mapping

```
residue = 16'h0000;
if ( high_bits == 2'b10)
  residue = state_table[index];
else state_table[index] =16'h0000;
```

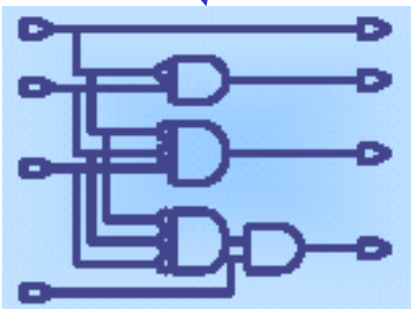
HDL Source

Translate



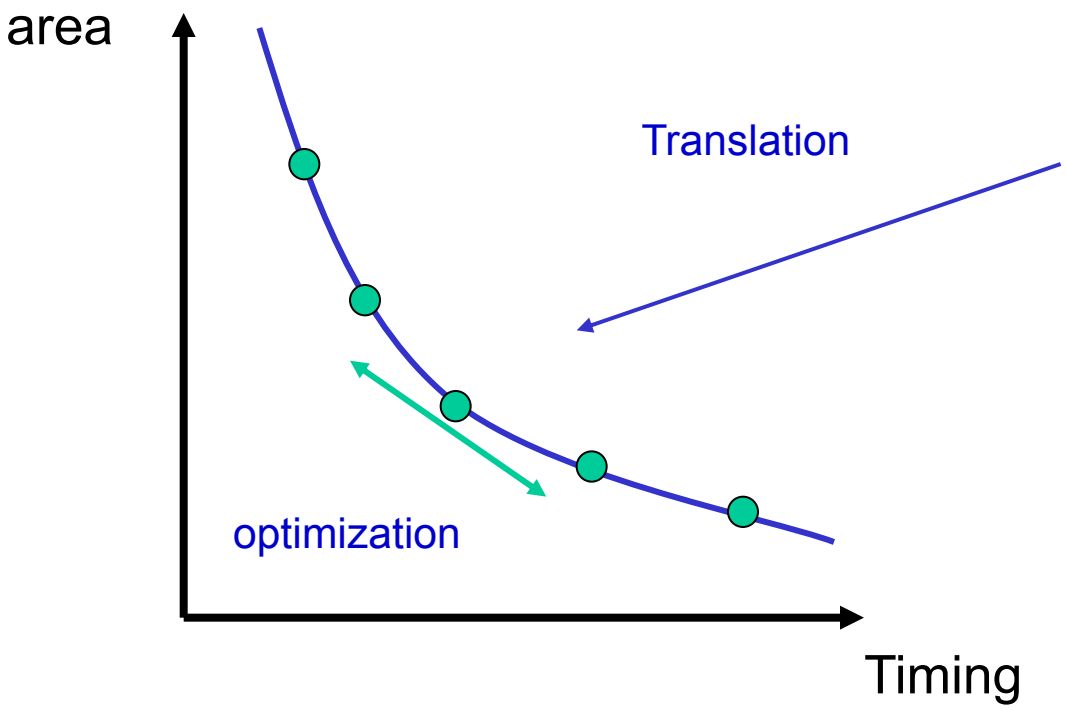
Generic Boolean (GTECH)

Optimize + Map



Target Technology

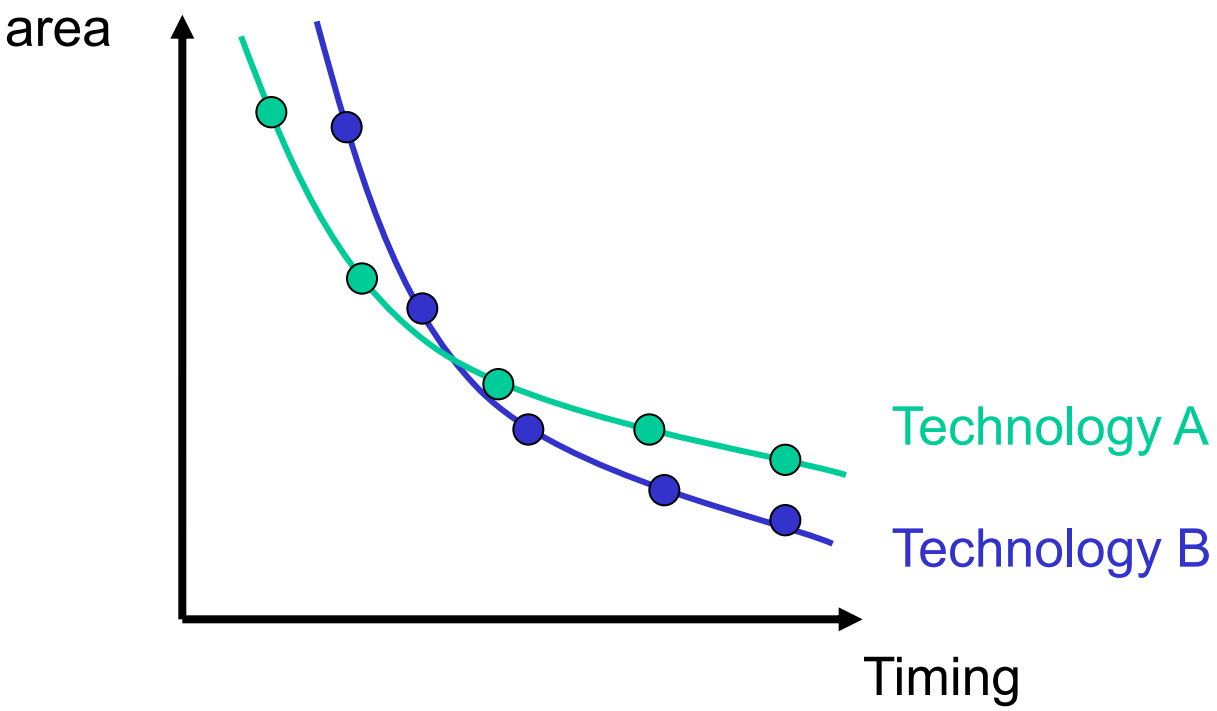
Synthesis is Constraint Driven



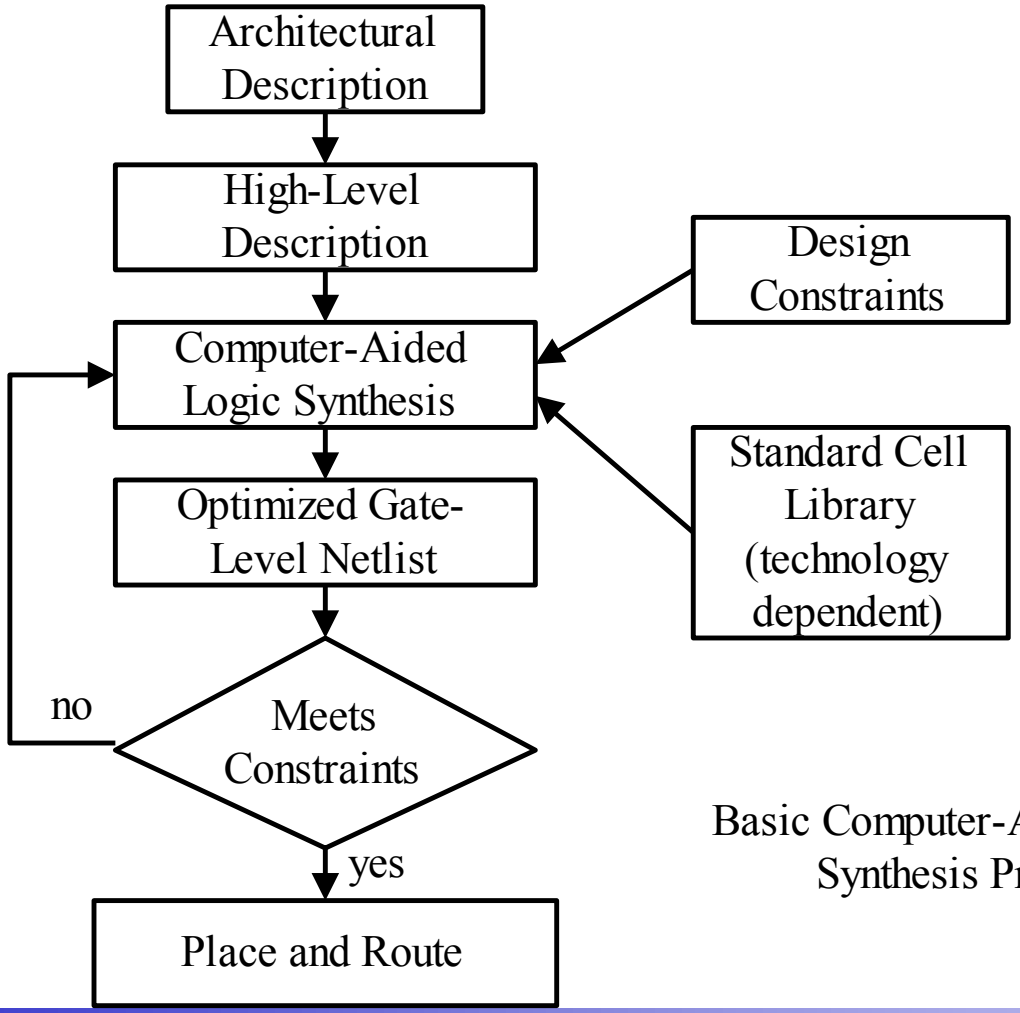

```
always @(reset or set)
begin : direct_set_reset
if (reset)
    y=1'b0;
else if (set)
    y=1'b1;
end
always @(gate or reset)
if (reset)
    t=1'b0;
else if (gate)
    t=d;
```

Technology Independent

- Design can be transferred to any technology



What is logic synthesis(cont.)



Basic Computer-Aided Logic Synthesis Process

Impact of Logic Synthesis

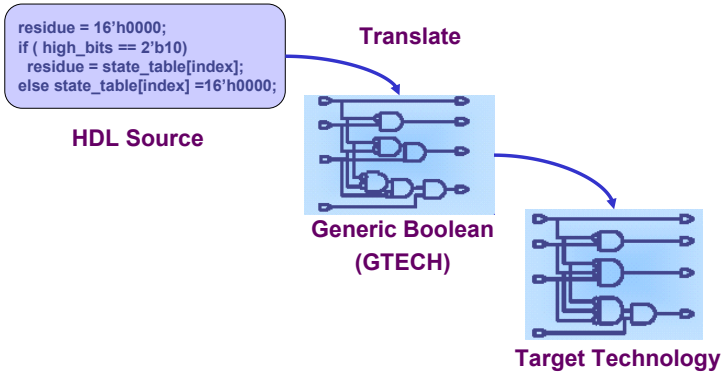
- Limitation on manual design
  - For large designs, manual conversion was prone human error, such as a small gate missed somewhere
  - The designer could never be sure that the design constraints were going to be met until the gate-level implementation is complete and tested
  - A significant portion of the design cycle was dominated by the time taken to convert a high-level design into gates
  - Design reuse was not possible
  - Each designer would implement design blocks differently. For large designs, this could mean that smaller blocks were optimized but the overall design was not optimal

Impact of Logic Synthesis(cont.)

- Automated Logic synthesis tools addressed these problems as follows
  - High-level design is less prone to human error because designs are described at a higher level of abstraction
  - High-level design is done without significant concern about design constraints
  - Conversion from high-level design to gates is fast
  - Logic synthesis tools optimize the design as a whole. This removes the problem with varied designer styles for the different blocks in the design and suboptimal designs
  - Logic synthesis tools allow technology-independent design
  - Design reuse is possible for technology-independent descriptions.

Logic Synthesis

- Takes place in two stages:



- Translation of Verilog (or VHDL) source to a netlist
  - Register inference
- Optimization of the resulting netlist to improve speed and area
  - Most critical part of the process
  - Algorithms very complicated and beyond the scope of this class

Combinational vs. Sequential Mapping

Combinational Mapping

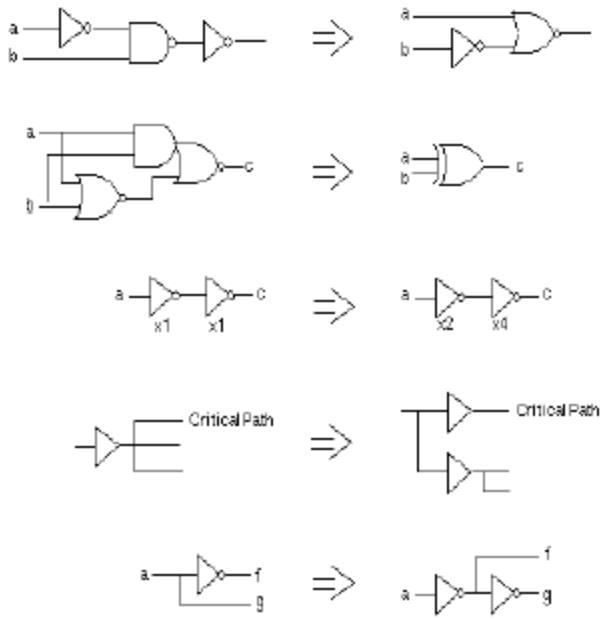
- Mapping rearranges components, combining and re-combining logic into different components
- May use different algorithms such as cloning, resizing or buffering
- Try to meet the design rule constraints and timing/area goals

Sequential Mapping

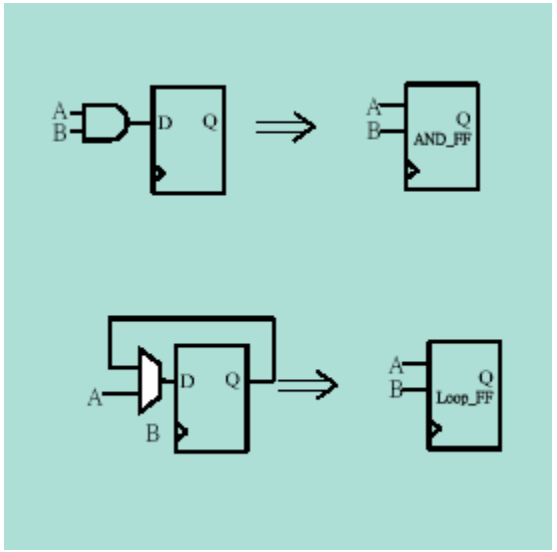
- Optimize the mapping to sequential cells from technology library
- Analyze combinational surrounding a sequential cell to see if it can absorb the logic attribute with HDL
- Try to save speed and area by using a more complex sequential cell

Mapping

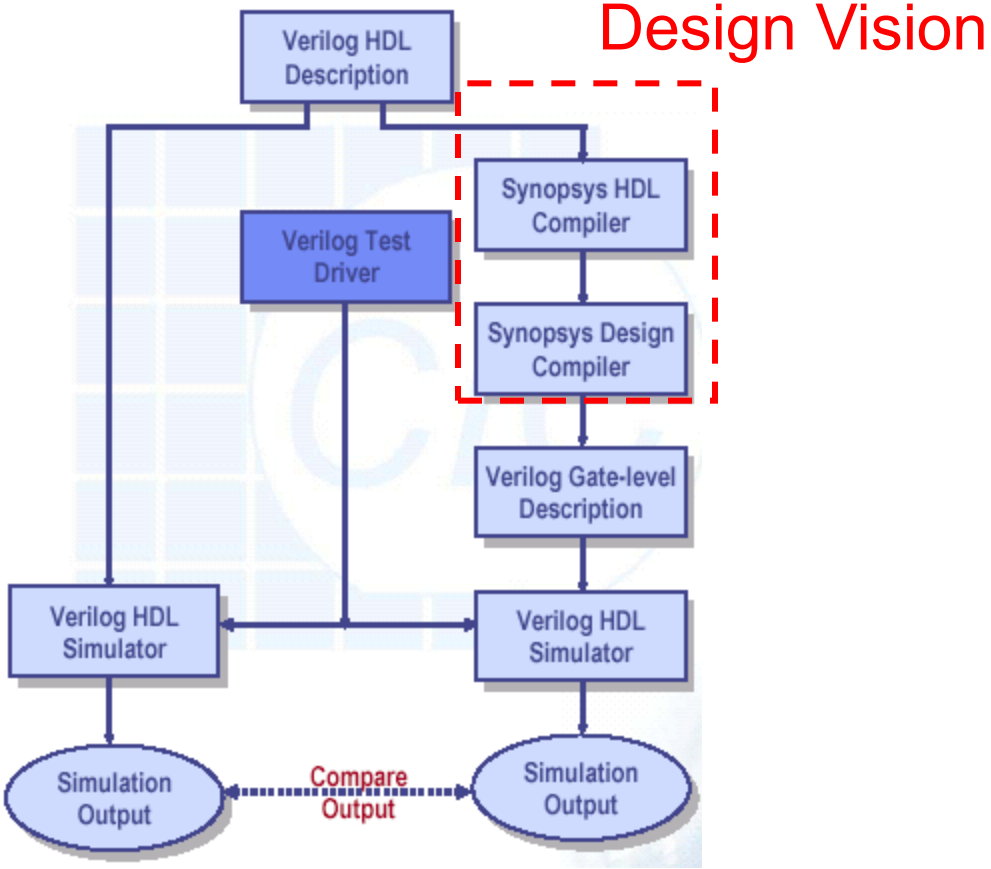
Combinational mapping



Sequential mapping



Design Methodology



Verilog and Logic Synthesis

- Verilog is used in two ways
  - Model for discrete-event simulation
  - Specification for a logic synthesis system

- Logic synthesis converts a subset of the Verilog language into an efficient netlist
- One of the major breakthroughs in designing logic chips in the last 20 years
- Most chips are designed using at least some logic synthesis

Verilog Primitive Cell

- Verilog primitive cells build basic combinational circuit
- Synthesizable Verilog primitives cells
  - and, nand, or, nor, xor, xnor, not
  - bufif0, bufif1, notif0, notif1

What Isn't Translated

- Initial blocks
  - Used to set up initial state or describe finite testbench stimuli
  - Don't have obvious hardware component
- Delays
  - May be in the Verilog source, but are simply ignored
- A variety of other obscure language features
  - In general, things heavily dependent on discrete-event simulation semantics
  - Certain "disable" statements
  - Pure events

HDL Compiler Unsupported

- delay
- initial
- repeat
- wait
- fork … join
- event
- deassign
- force
- release
- primitive -- User defined primitive (Sequential)
- time
- triand, trior, tri1, tri0, trireg
- nmos, pmos, cmos, rnmos, rpmos, rcmos
- pullup, pulldown
- rtran, tranif0, tranif1, rtranif0, rtranif1
- case identity and not identity operators
- Division and modulus operators
  - division can be done using DesignWare instantiation

Verilog Operators Supported

- Binary bit-wise (~,&,|,^,~^)
- Unary reduction (&,~&,|,~|,^,~^)
- Logical (!,&&,||)
- 2's complement arithmetic (+,-,*,/,%)
- Relational (>,<,>=,<=)
- Equality (==,!=)
- Logical shift (>>,<<)
- Conditional (?:)

always Block

- Example

  always @(event-expression)
    begin
      statements
    end
- If event-expression contains posedge or negedge, flip-flop will be synthesized.
- In all other cases, combinational logic will be synthesized.
- A variable assigned within an always @ block that is not fully specified will result in latches synthesized.

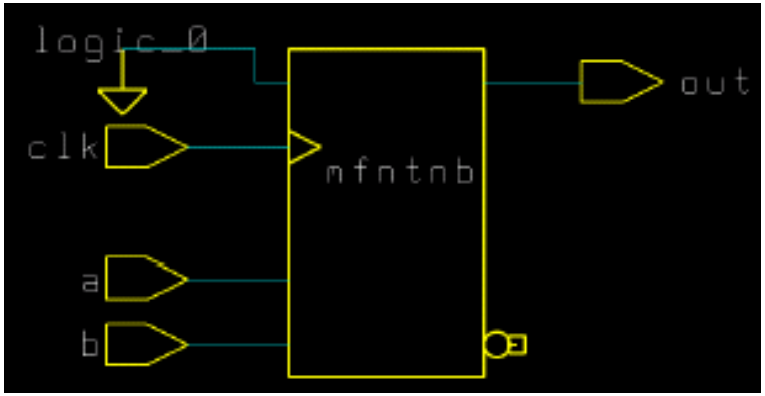
Latch Inference

- A variable assigned within an always block that is not fully specified.
- If a case statement is not a full case, it will infer a latch.

Register Inference

- A register (flip-flop) is implied when you use the @(posedge clk) or @(negedge clk) in an always block.
- Any variable that is assigned a value in this always block is synthesized as a D-type edge-triggered flip-flop.

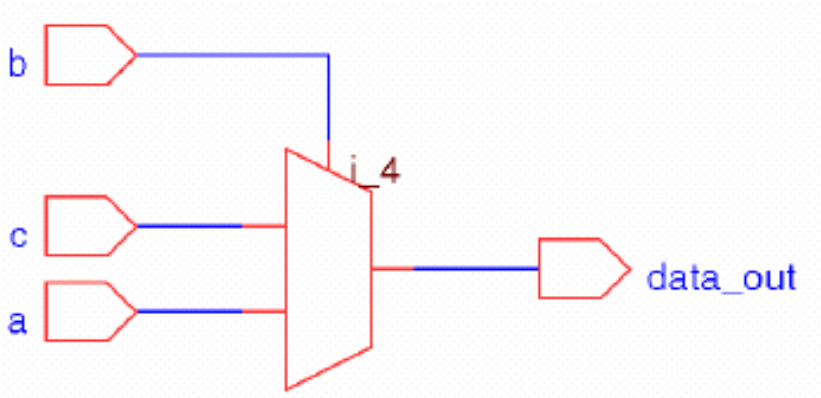
```
always @(posedeg clk)
  out <= a & b;
```



Combinational Logic Inference

- Not all variables declared as reg data type need a flip-flop or a latch for logic implementation.

```
reg data_out;
always @(a or b or c)
if(b)
    data_out = a;
else
    data_out = c;
```



Comparisons to X or Z

- Comparisons to an X or Z are always ignored.
- Comparison is always evaluated to false, which may cause simulation & synthesis mismatch.

```
module compare_x(A,B);
input A;
output B;
reg B;
always begin
if (A== 1'bx)
  B=0;
else
  B=1;
end
endmodule
```

Warning: Comparisons to a "don't care" are treated as always being false in routine compare_x line 7 in file "compare_x.v" this may cause simulation to disagree with synthesis. (HDL-170)

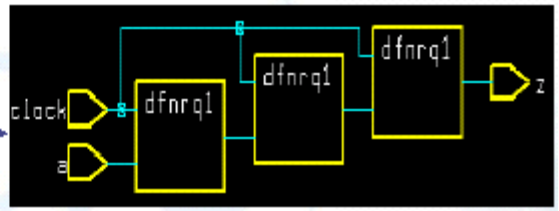
Always false

Non-Blocking and Blocking

- For the most straightforward code:
  (you get what you expect)
  - Use ***non-blocking*** assignments within sequential always block.
  - Example:

```
always @(posedge clock) begin
     x <= a;
     y <= x;
     z <= y;
end
```

Usually you expect




```
always @(posedge clock) begin
     x = a;
     y = x;
     z = y;
end
```

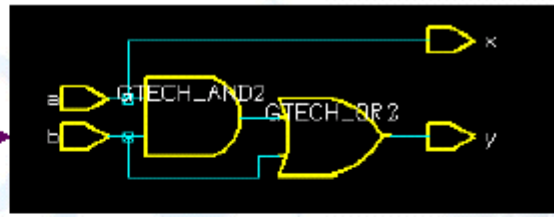
May not you expect

Non-Blocking and Blocking

- For the most straightforward code:
  (you get what you expect)
  - Use ***blocking*** assignments within combinational always block.
  - Example:

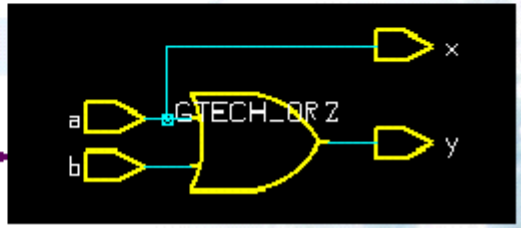
```
always @(a or b or x) begin
    x = a & b;
    y = x | b;
    x = a;
end
```

Usually you expect




```
always @(a or b or x) begin
    x <= a & b;
    y <= x | b;
    x <= a;
end
```

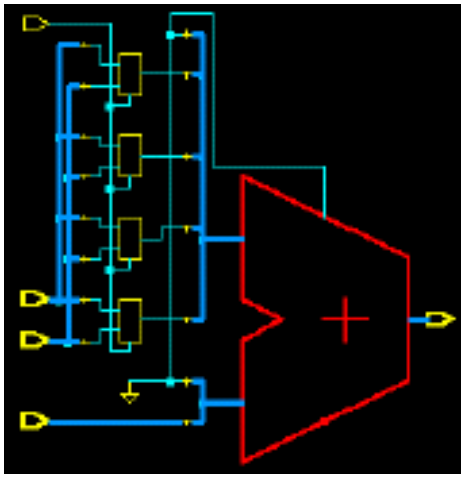
May not you expect



Resource Sharing

- Operations can be shared if they lie in the same always block.

```
always @(a or b or c or sel)
    if (sel)
        z=a+b;
    else
        z=a+c;
```

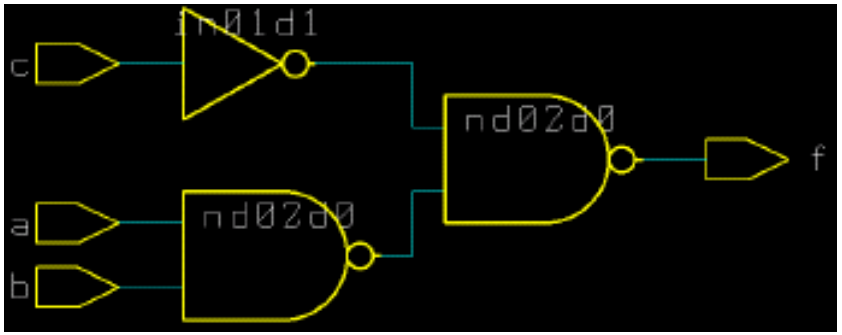


Combinational Always Block

- Sensitivity list must be specified completely, otherwise synthesis may mismatch with simulation

```
always @(a or b or c)
  f=a&b|c;
```

```
always @(a or b)
  f=a&b|c;
```



Warning: Variable 'c' is being read
in routine train line 6 in file '/ccyang/abc/train1.v',
but does not occur in the timing control of the block which begins
there. (HDL-180)

if Statement

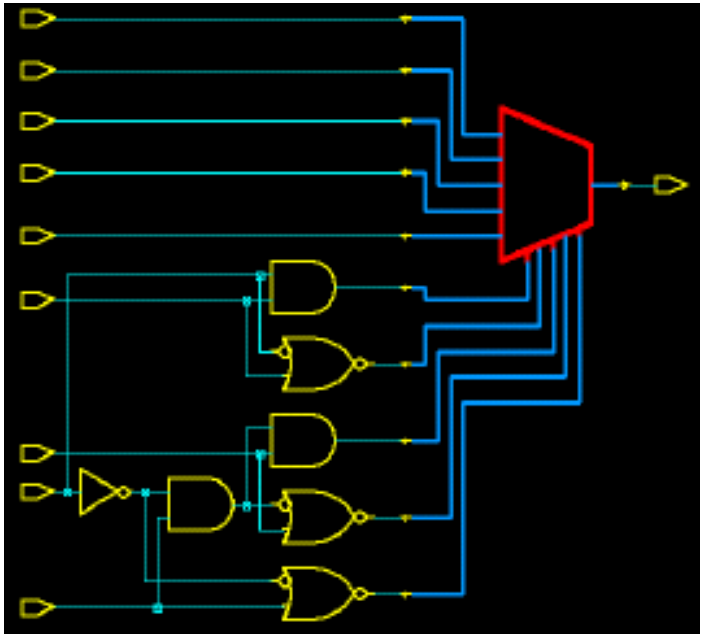
- Provide for more complex conditional actions, each condition expression controls a multiplexer legal only in function & always construct
- Syntax

```
if (expr )
  begin
  ... statements ...
  end
else
  begin
  ... statements ...
  end
```

if Statement

- if statement can be nested

```
always @(sel1 or sel2 or sel3 or sel4 or in1
or in2 or in3 or in4 or in5)
  begin
    if (sel1) begin
      if (sel2) out=in1;
      else out=in2;
    end
    else if (sel3) begin
      if (sel4) out=in3;
      else out=in4;
    end
    else out=in5;
  end
```



if Statement

- What's the difference between these two coding styles?

```
module mult_if(a, b, c, d, e, sel, z);
input a, b, c, d, e;
input [3:0] sel;
output z;
reg z;
always @(a or b or c or d or e or sel)
begin
z = e;
if (sel[0]) z = a;
if (sel[1]) z = b;
if (sel[2]) z = c;
if (sel[3]) z = d;
end
endmodule
```

```
module single_if(a, b, c, d, e, sel, z);
input a, b, c, d, e;
input [3:0] sel;
output z;
reg z;
always @(a or b or c or d or e or sel)
begin
z = e;
if (sel[3])
  z = d;
else if (sel[2])
  z = c;
else if (sel[1])
  z = b;
else if(sel[0])
  z = a;
end
endmodule
```

Priority of sel[3] is highest

case Statement

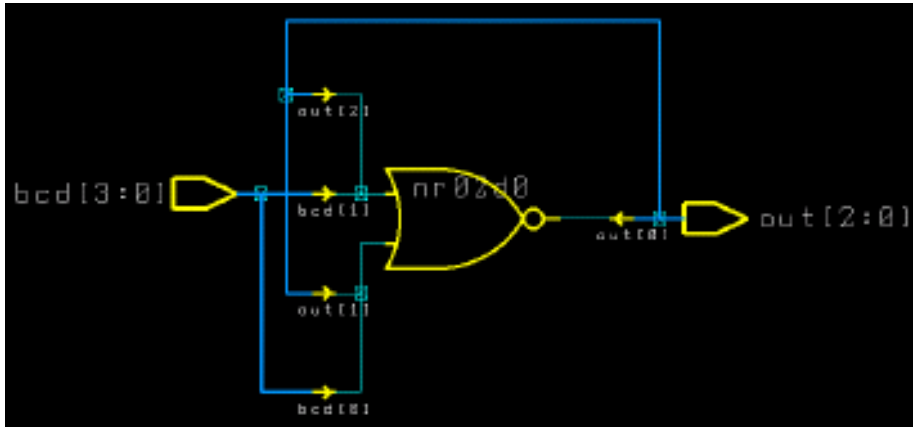
- Legal only in the function & always construct
- syntax

```
case ( expr )
     case_item1: begin
     ... statements ...
     end
     case_item2: begin
     ... statements ...
     end
     default: begin
     ... statements ...
     end
endcase
```

case Statement

- A case statement is called a full case if all possible branches are specified.

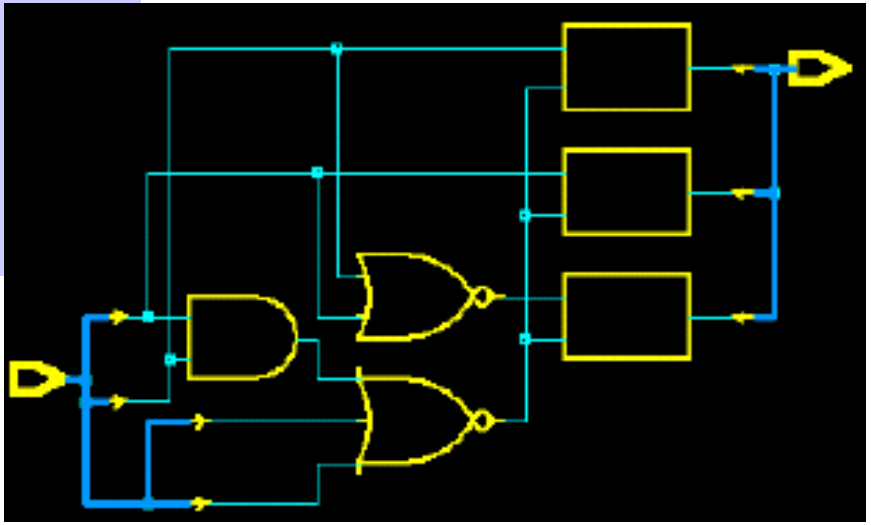
```
always @(bcd) begin
  case (bcd)
    4'd0:out=3'b001;
    4'd1:out=3'b010;
    4'd2:out=3'b100;
    default:out=3'b000;
  endcase
end
```



case Statement

- If a case statement is not a full case, it will infer a latch.

```
always @(bcd) begin
  case (bcd)
    4'd0:out=3'b001;
    4'd1:out=3'b010;
    4'd2:out=3'b100;
  endcase
end
```



case Statement

- **Note:** the second case item does not modify reg2, causing it to be inferred as a latch (to retain last value).

```
case (cntr_sig) // synopsys full_case
2'b00 : begin
        reg1 = 0 ;
        reg2 = v_field ;
        end
2'b01 : reg1 = v_field ; /* latch will be inferred for reg2*/

2'b10 : begin
        reg1 = v_field ;
        reg2 = 0 ;
        end
endcase
```

reg1 -> combinational

reg2 -> latch

case Statement

- Two possible ways we can assign a default value to next_state.

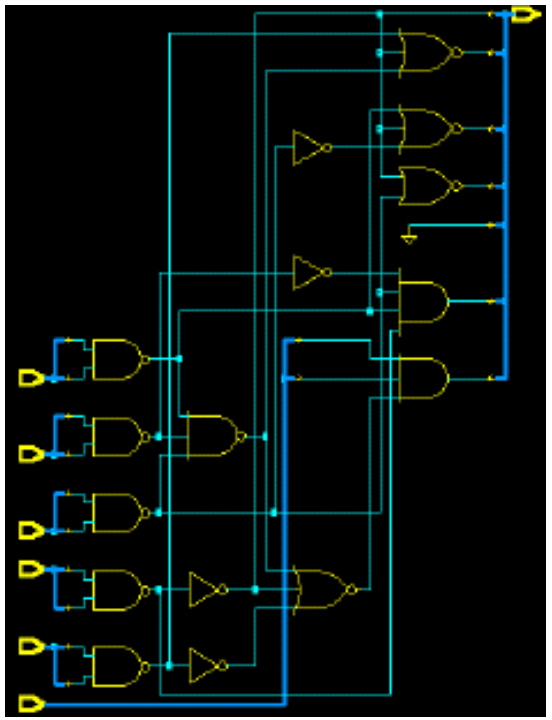
(1)
```
out = 3'b000 ; // this is called unconditional assignment
case (condition)
…
endcase
```

(2)
```
case (condition)
…
default : out = 3'b000 ; // out=0 for all other cases
endcase
```

case Statement

- If HDL Compiler can't determine that case branches are parallel, its synthesized hardware will include a priority decoder.

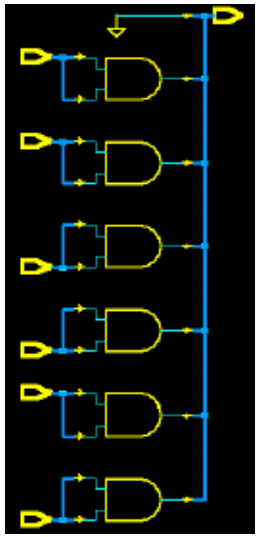
```
always @(u or v or w or x or y or z)
begin
case (2'b11)
  u:out=10'b0000000001;
  v:out=10'b0000000010;
  w:out=10'b0000000100;
  x:out=10'b0000001000;
  y:out=10'b0000010000;
  z:out=10'b0000100000;
  default:out=10'b0000000000;
endcase
end
```



case Statement

- If you can be sure that only one variable will be "11" at one time.
- You can declare a case statement as parallel case with the "`//synopsys parallel_case`" directive.

```
always @(u or v or w or x or y or z)
begin
case (2'b11) //synopsys parallel_case
  u:out=10'b0000000001;
  v:out=10'b0000000010;
  w:out=10'b0000000100;
  x:out=10'b0000001000;
  y:out=10'b0000010000;
  z:out=10'b0000100000;
  default:out=10'b0000000000;
endcase
end
```



if vs. case

- Recommendations:
  - If the “if else” chain is too long, use “case” statement to replace them.
  - If you can know the conditions of “case” statement are mutually exclusive, please use synopsys directive “`//synopsys parallel_case`” in order to let design compiler to create a parallel decoder for you.
  - If you know the conditions of a “case” statement, which is not listed, will never occur, please use “`//synopsys full_case`” directive in order to prevent latches been synthesized.

for Loop

- Provide a shorthand way of writing a series of statements.
- Loop index variables must be integer type.
- Step, start & end value must be constant.
- In synthesis, for loops loops are “unrolled”, and then synthesized.
- Example

```
always @(a or b) begin
  for( k=0; k<=3; k=k+1 ) begin
    out[k]=a[k]^b[k];
    c=(a[k]|b[k])&c;
  end
end
```

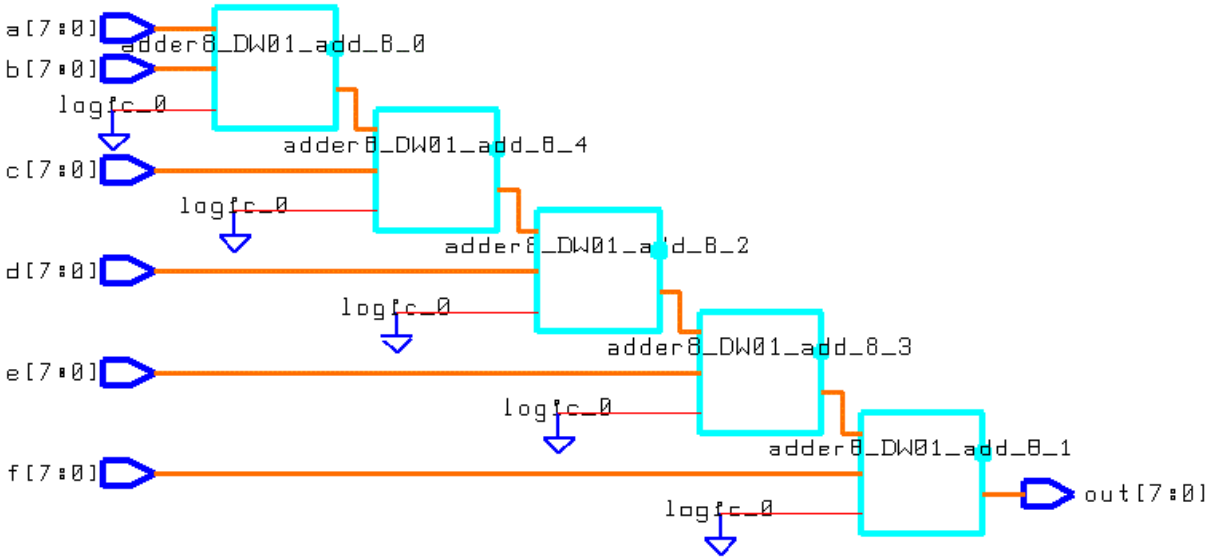
```
out[0] = a[0]^b[0];
out[1] = a[1]^b[1];
out[2] = a[2]^b[2];
out[3] = a[3]^b[3];
c = (a[0] | b[0]) & (a[1] | b[1]) &
     a[2] | b[2]) & (a[3] | b[3]) & c;
```

Write Efficient HDL Code

- Use parentheses control complex structure of a design.
- Use operator bit-width efficiently.
- Data-path Duplication
- Operator in “if” statement

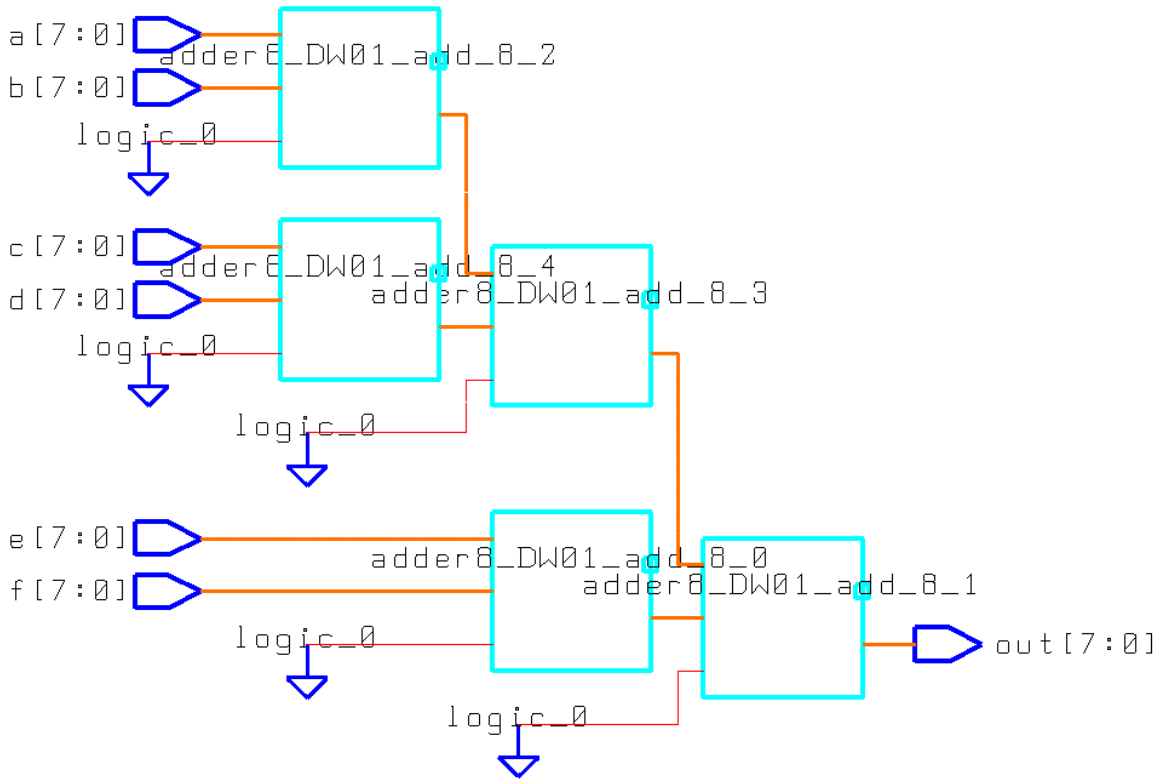
Use Parentheses Properly

- Out=a+b+c+d+e+f;



Use Parentheses Properly(cont.)

- Out=(a+b)+(c+d)+(e+f);



Use Operator Bit-Width Efficiently

```
module test(a,b,out);
input [7:0] a,b;
output [8:0] out;
assign out=add_lt_10(a,b);

function [8:0] add_lt_10;
input [7:0] a,b;
reg [7:0] temp;
begin
  if (b<10) temp=b;
  else temp=10;
  add_lt_10=a+temp[3:0]; //use [3:0] for temp
end
endfunction

endmodule
```

HHHHFFFH

Coding Skill-Data-path Duplication

No_dulpicated

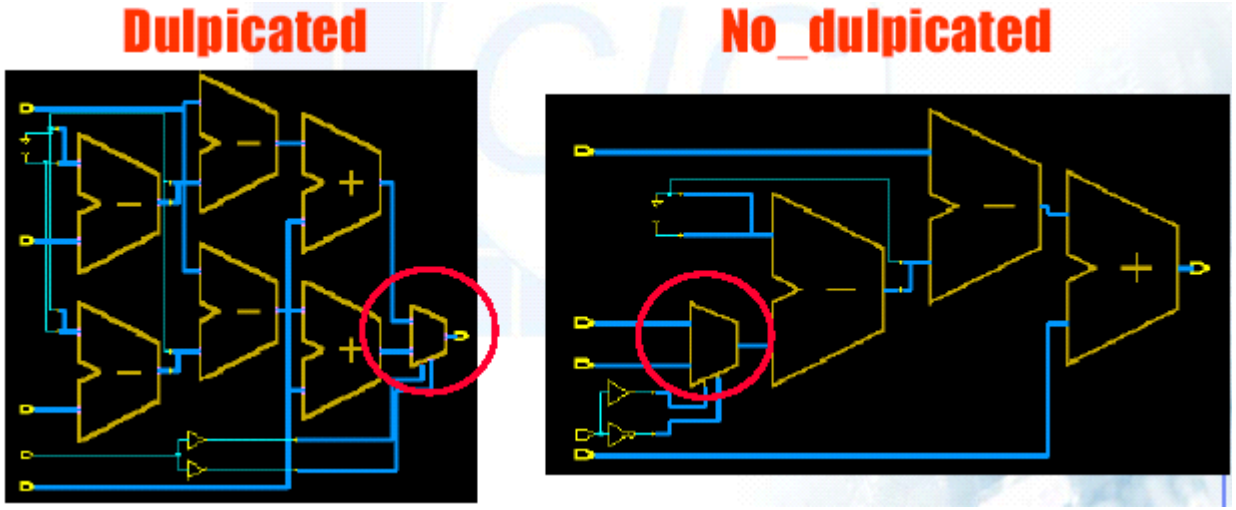
```
module BEFORE (ADDRESS, PTR1, PTR2, B, CONTROL, COUNT);
input [7:0] PTR1,PTR2;
input [15:0] ADDRESS, B;
input CONTROL;        // CONTROL is late arriving
output [15:0] COUNT;
parameter [7:0] BASE = 8'b10000000;
wire [7:0] PTR, OFFSET;
wire [15:0] ADDR;
assign PTR = (CONTROL == 1'b1) ? PTR1 : PTR2;
assign OFFSET = BASE - PTR; //Could be any function f(BASE,PTR)
assign ADDR = ADDRESS - {8'h00, OFFSET};
assign COUNT = ADDR + B;
endmodule
```

Dulpicated

```
module PRECOMPUTED (ADDRESS, PTR1, PTR2, B, CONTROL, COUNT);
input [7:0] PTR1, PTR2;
input [15:0] ADDRESS, B;
input CONTROL;
output [15:0] COUNT;
parameter [7:0] BASE = 8'b10000000;
wire [7:0] OFFSET1,OFFSET2;
wire [15:0] ADDR1,ADDR2,COUNT1,COUNT2;
assign OFFSET1 = BASE - PTR1;  // Could be f(BASE,PTR)
assign OFFSET2 = BASE - PTR2;  // Could be f(BASE,PTR)
assign ADDR1 = ADDRESS - {8'h00 , OFFSET1};
assign ADDR2 = ADDRESS - {8'h00 , OFFSET2};
assign COUNT1 = ADDR1 + B;
assign COUNT2 = ADDR2 + B;
assign COUNT = (CONTROL == 1'b1) ? COUNT1 : COUNT2;
endmodule
```

Coding Skill - Data-path Duplication

- We assume that signal "CONTROL" is the latest arrival pin.
- By this skill, we will reduce latency but we must pay for it, area!



Coding Skill -- operator in if

- We assume that signal “A” is latest arrival signal

Before_improved

```
module cond_oper(A, B, C, D, Z);
parameter N = 8;
input [N-1:0] A, B, C, D;
//A is late arriving
output [N-1:0] Z;
reg [N-1:0] Z;

always @(A or B or C or D) begin
if (A + B < 24)
  Z <= C;
else
  Z <= D;
end
endmodule
```

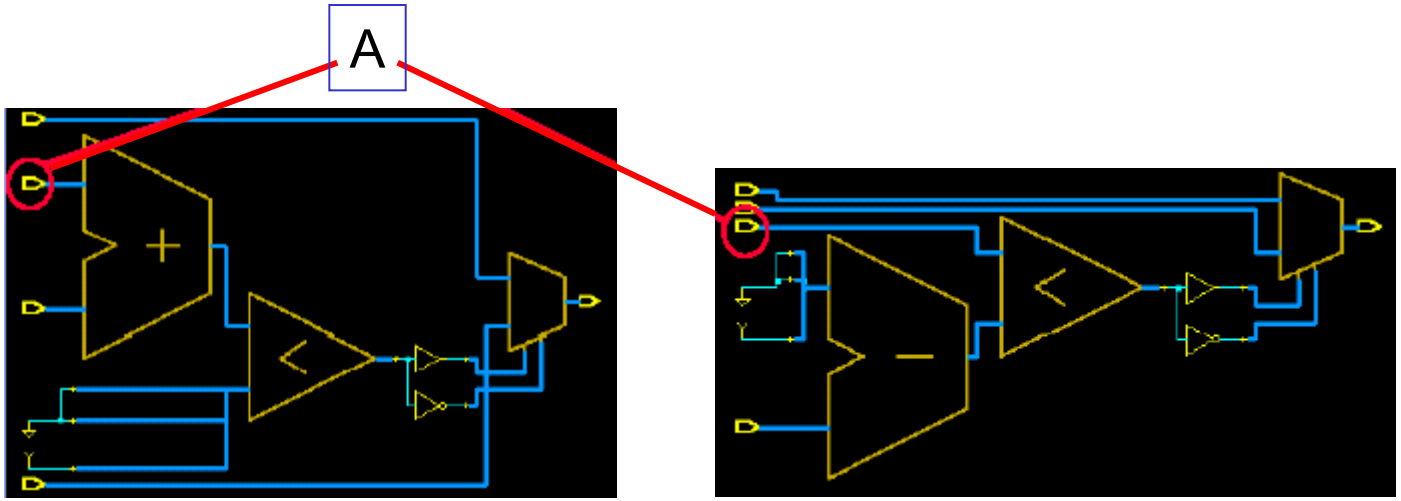
Improved

```
module cond_oper_improved (A, B, C, D, Z);
parameter N = 8;
input [N-1:0] A, B, C, D;
// A is late arriving
output [N-1:0] Z;
reg [N-1:0] Z;

always @(A or B or C or D) begin
if (A < 24 - B)
  Z <= C;
else
  Z <= D;
end
endmodule
```

Coding Skill -- operator in if

- In this example, not only latency reduced, but also area reduced.



Before_improved

Improved