

ET5080E

Digital Design Using Verilog HDL

Fall '21

For Loops & Synthesis
Generate Statements
Use of X in Synthesis
Synthesis Pitfalls
Coding for Synthesis

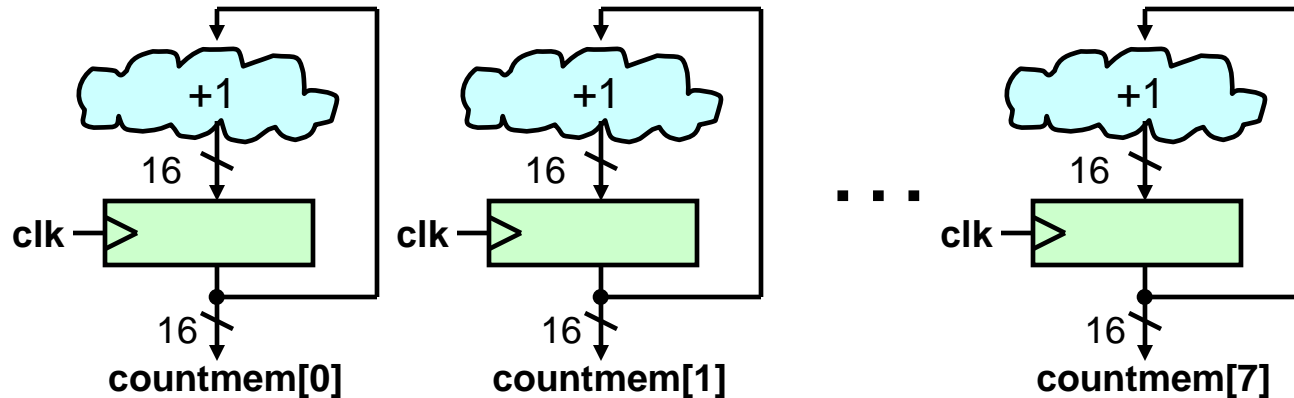
For Loops & Synthesis

- Can a For Loop be synthesized?
 - Yes, if it is fixed length
 - The loop is “*unrolled*”

```
reg [15:0] countmem [0:7];  
integer x;  
always @(posedge clk) begin  
    for (x = 0; x < 8; x = x + 1) begin  
        countmem[x] <= countmem[x] + 1;  
    end  
end
```

How do you think this code would synthesize?

For Loops & Synthesis



- These loops are *unrolled* when synthesized
 - That's why they must be fixed in length!
 - Loop index is type **integer** but it is not actually synthesized
 - Example creates eight 16-bit incrementers.
- What if loop upper limit was a parameter?

Unnecessary Calculations

- Expressions that are fixed in a **for loop** are replicated due to “loop unrolling.”
- Solution: Move fixed (unchanging) expressions outside of all loops.

```
for (x = 0; x < 8; x = x + 1) begin  
  for (y = 0; y < 8; y = y + 1) begin  
    index = x*8 + y;  
    value = (a + b)*c;  
    mem[index] = value;  
  end  
end
```

This is just basic common sense, and applies to any language (in a programming language you would be wasting time, not hardware).

Yet this is a common mistake

- Which expressions should be moved?

More on Loops & Synthesis

- A loop is static (data-independent) if the number of iterations is fixed at compile-time
- Loop Types
 - Static without internal timing control
 - ✓ Combinational logic
 - Static with internal timing control (i.e. @(posedge clk))
 - ✓ Sequential logic
 - Non-static without internal timing control
 - ✓ Not synthesizable
 - Non-static with internal timing control (i.e. @(posedge clk))
 - ✓ Sometimes synthesizable, Sequential logic

Static Loops w/o Internal Timing

- Combinational logic results from “loop unrolling”
- Example

```
always @(a) begin  
    andval[0] = 1;  
    for (i = 0; i < 4; i = i + 1)  
        andval[i + 1] = andval[i] & a[i];  
end
```

- What would this look like?
- For registered outputs:
 - Change sensitivity list ‘a’ with ‘**posedge** clk’

Static Loops with Internal Timing

- If a static loop contains an internal edge-sensitive event control expression, then activity distributed over multiple cycles of the clock

```
always begin
```

```
    for (i = 0; i < 4; i = i + 1)
```

```
        @(posedge clk) sum <= sum + i;
```

```
end
```

- What does this loop do?
- Does it synthesize?...Yes, but...

Non-Static Loops w/o Internal Timing

- Number of iterations is variable
 - Not known at compile time
- Can be simulated, but not synthesized!
- Essentially an iterative combinational circuit of data dependent size!

```
always@(a, n) begin  
    andval[0] = 1;  
    for (i = 0; i < n; i = i + 1)  
        andval[i + 1] = andval[i] & a[i];  
end
```

- What if n is a parameter?

Non-Static Loops with Internal Timing

- Number of iterations determined by
 - Variable modified within the loop
 - Variable that can't be determined at compile time
- Due to internal timing control—
 - Distributed over multiple cycles
 - Number of cycles determined by variable above
- Variable must still be bounded

always begin

continue = 1'b1;

for (; continue;) **begin**

@(**posedge** clk) sum = sum + in;

if (sum > 8'd42) continue = 1'b0;

end

end

Can this be synthesized?

What does it synthesize to?

Who really cares!
This is a stupid way to do it!
Use a SM.

Any loop with internal timing can be done as a SM

```
module sum_till (clk,rst_n,sum,in);  
  
input clk,rst_n;  
input [5:0] in;  
output [5:0] sum;  
  
always @(posedge clk or negedge rst_n)  
    if (~rst_n)  
        sum <= 6'h00;  
    else if (en_sum)  
        sum <= sum + in  
  
assign en_sum = (sum<6'd43) ? 1'b1 : 1'b0;  
endmodule
```

Previous example didn't really even require a SM.

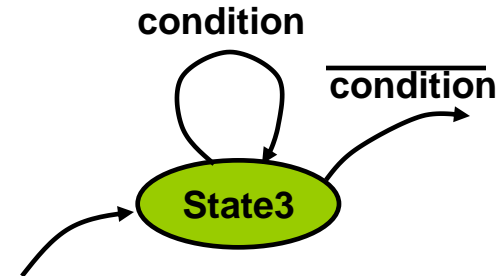
This code leaves no question how it would synthesize.

RULE OF THUMB:

If it takes you more than 15 seconds to conceptualize how a piece of code will synthesize, then it will probably confuse Synopsys too.

FSM Replacement for Loops

- Not all loop structures supported by vendors
- Can always implement a loop with internal timing using an FSM
 - Can make a “while” loop easily
 - Often use counters along with the FSM
- All synthesizers support FSMs!
- Synopsys supports for-loops with a static number of iterations



Generated Instantiation

- Generate statements → control over the instantiation/creation of:
 - ✓ Modules
 - ✓ UDPs & gate primitives
 - ✓ continuous assignments
 - ✓ **initial** blocks & **always** blocks
- Generate instantiations resolved during “elaboration” (compile time)
 - ✓ When module instantiations are linked to module definitions
 - ✓ **Before** the design is simulated or synthesized – this is NOT dynamically created hardware

Generate-Loop

- A generate-loop permits making one or more instantiations (pre-synthesis) using a for-loop.

```
module gray2bin1 (bin, gray);  
  parameter SIZE = 8;  // this module is parameterizable  
  output [SIZE-1:0] bin; input [SIZE-1:0] gray;  
  genvar i;  
  generate  
    for (i=0; i<SIZE; i=i+1) begin: bit  
      assign bin[i] = ^gray[SIZE-1:i];  \\ Data Flow replication  
    end  
  endgenerate  
endmodule
```

Does not exist during simulation of design

Typically name the generate as reference

Generate Loop

- Is really just a code replication method. So it can be used with any style of coding. Gets expanded prior to simulation.

```
module replication_struct(i0, i1, out);  
parameter N=32;  
input [N-1:0] i1,i0;  
output [N-1:0] out;  
genvar j;  
generate  
for (j=0; j<N; j=j+1)  
  begin : xor_loop  
    xor g1 (out[j],in0[j],in1[j]);  
  end  
endgenerate  
endmodule
```

Hierarchical reference to these instantiated gates will be:

...xor_loop[0].g1
...xor_loop[1].g1
.
.
.
...xor_loop[31].g1

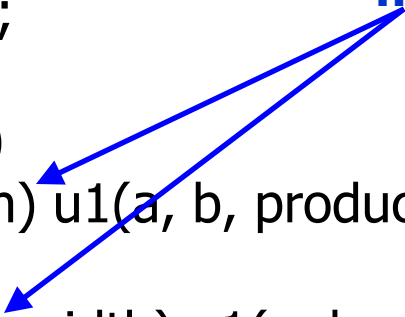
Structural replication

Generate-Conditional

- A generate-conditional allows conditional (pre-synthesis) instantiation using **if-else-if** constructs

```
module multiplier(a ,b ,product);  
  parameter a_width = 8, b_width = 8;  
  localparam product_width = a_width+b_width;  
  input [a_width-1:0] a; input [b_width-1:0] b;  
  output [product_width-1:0] product;  
  generate  
    if ((a_width < 8) || (b_width < 8))  
      CLA_multiplier #(a_width,b_width) u1(a, b, product);  
    else  
      WALLACE_multiplier #(a_width,b_width) u1(a, b, product);  
  endgenerate  
endmodule
```

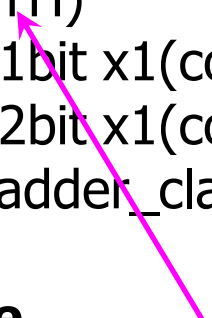
These are parameters, not variables!



Generate-Case

- A generate-case allows conditional (pre-synthesis) instantiation using case constructs
- See Standard 12.1.3 for more details

```
module adder (output co, sum, input a, b, ci);  
  parameter WIDTH = 8;  
  generate  
    case (WIDTH)  
      1: adder_1bit x1(co, sum, a, b, ci); // 1-bit adder implementation  
      2: adder_2bit x1(co, sum, a, b, ci); // 2-bit adder implementation  
      default: adder_cla #(WIDTH) x1(co, sum, a, b, ci);  
    endcase  
  endgenerate  
endmodule
```



Of course case selector has to be deterministic at elaborate time, can not be a variable. Usually a parameter.

Synthesis Of **x** And **z**

- Only allowable uses of **x** is as “don’t care”, since **x** cannot actually exist in hardware
 - in **case x**
 - in defaults of conditionals such as :
 - ✓ The **else** clause of an **if** statement
 - ✓ The **default** selection of a **case** statement
- Only allowable use of **z**:
 - Constructs implying a 3-state output
 - ✓ Of course it is helpful if your library supports this!

Don't Cares

- **x**, **?**, or **z** within case item expression in **case**
 - Does not actually output “don’t cares”!
 - Values for which input comparison to be ignored
 - Simplifies the case selection logic for the synthesis tool

```
case (state)
  3'b0??: out = 1'b1;
  3'b10?: out = 1'b0;
  3'b11?: out = 1'b1;
endcase
```

		state[1:0]			
		00	01	11	10
state[2]	0	1	1	1	1
	1	0	0	1	1

$$\text{out} = \overline{\text{state}[0]} + \text{state}[1]$$

Use of Don't Care in Outputs

- Can really reduce area

```
case (state)
  3'b001: out = 1'b1;
  3'b100: out = 1'b0;
  3'b110: out = 1'b1;
  default: out = 1'b0;
endcase
```

		state[1:0]			
		00	01	11	10
state[2]	0	0	1	0	0
	1	0	0	0	1

```
case (state)
  3'b001: out = 1'b1;
  3'b100: out = 1'b0;
  3'b110: out = 1'b1;
  default: out = 1'bx;
endcase
```

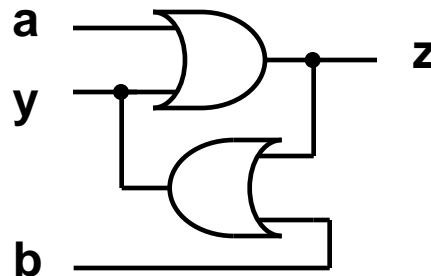
		state[1:0]			
		00	01	11	10
state[2]	0	X	1	X	X
	1	0	X	X	1

Unintentional Latches

- Avoid structural feedback in continuous assignments, combinational **always**

```
assign z = a | y;
```

```
assign y = b | z;
```



- Avoid incomplete sensitivity lists in combinational **always**
- For conditional assignments, either:
 - Set default values before statement
 - Make sure LHS has value in every branch/condition
- For warning, set **hdlin_check_no_latch** true before compiling

Synthesis Example [1]

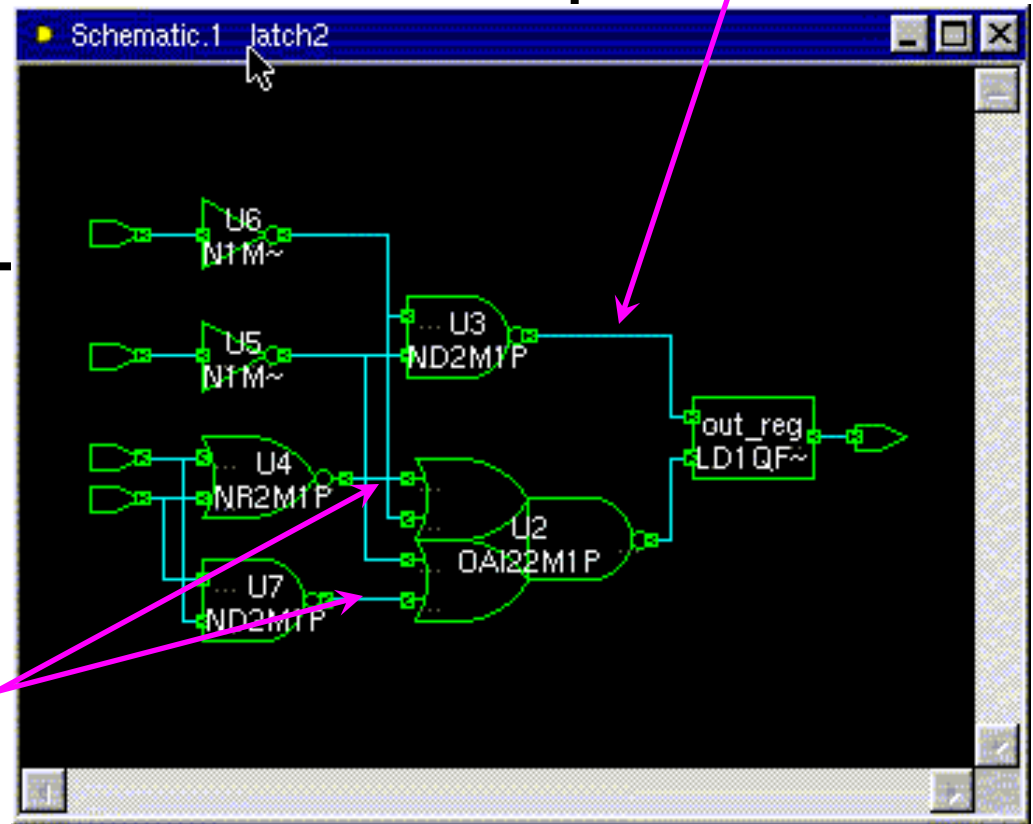
```
module Hmmm(input a, b, c, d, output reg out);  
always @(a, b, c, d) begin  
    if (a) out = c | d;  
    else if (b) out = c & d;  
end  
endmodule
```

a | b enables latch

How will this synthesize?

Area = 44.02

Either **c | d** or **c&d** are passed through an inverting mux depending on state of **a / b**



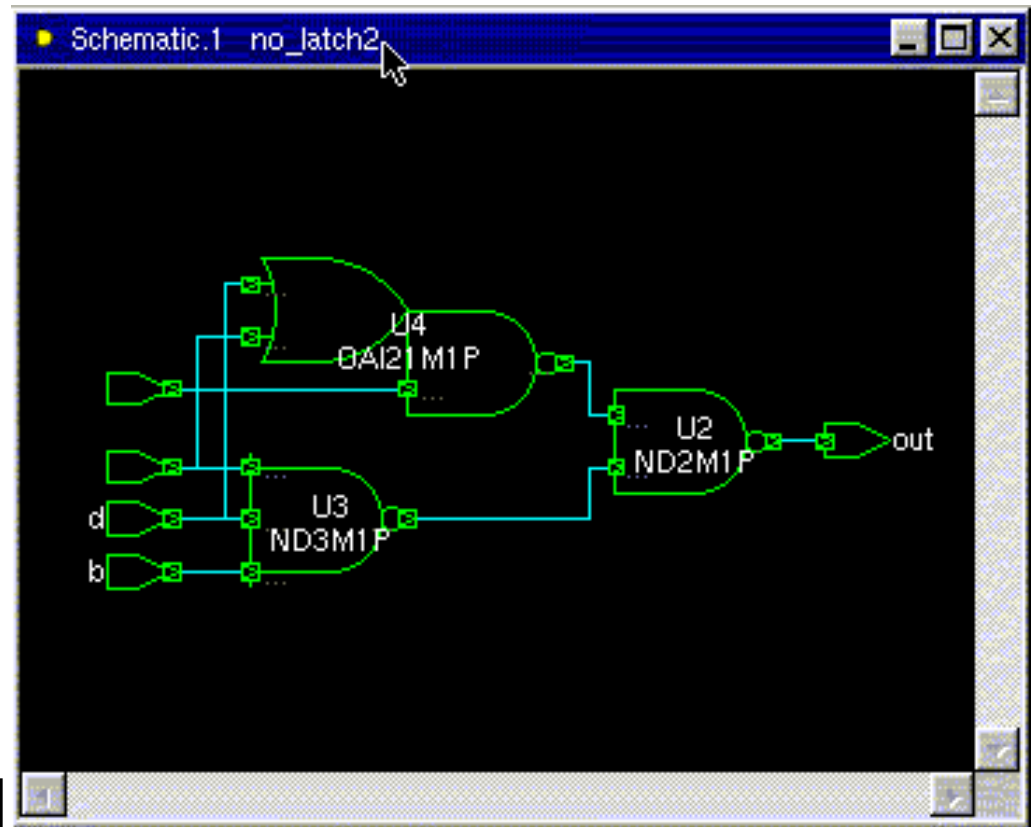
Synthesis Example [2]

```
module Better(input a, b, c, d, output reg out);  
always @(a, b, c, d) begin  
  if (a) out = c | d;  
  else if (b) out = c & d;  
  else out = 1'b0;  
end  
endmodule
```

Perhaps what you meant
was that if not **a** or **b** then
out should be zero??

Area = 16.08

Does synthesize better...no latch!



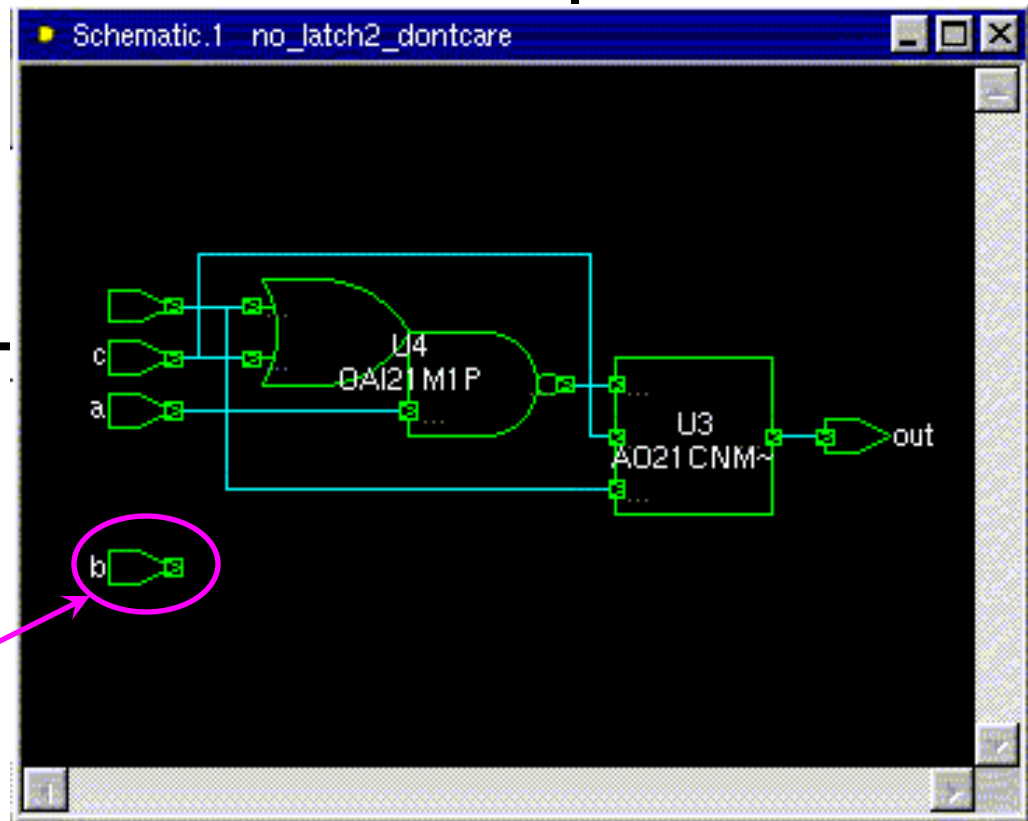
Synthesis Example [3]

```
module BetterYet(input a, b, c, d, output reg out);  
always @(a, b, c, d) begin  
  if (a) out = c | d;  
  else if (b) out = c & d;  
  else out = 1'bx;  
end  
endmodule
```

Area = 12.99

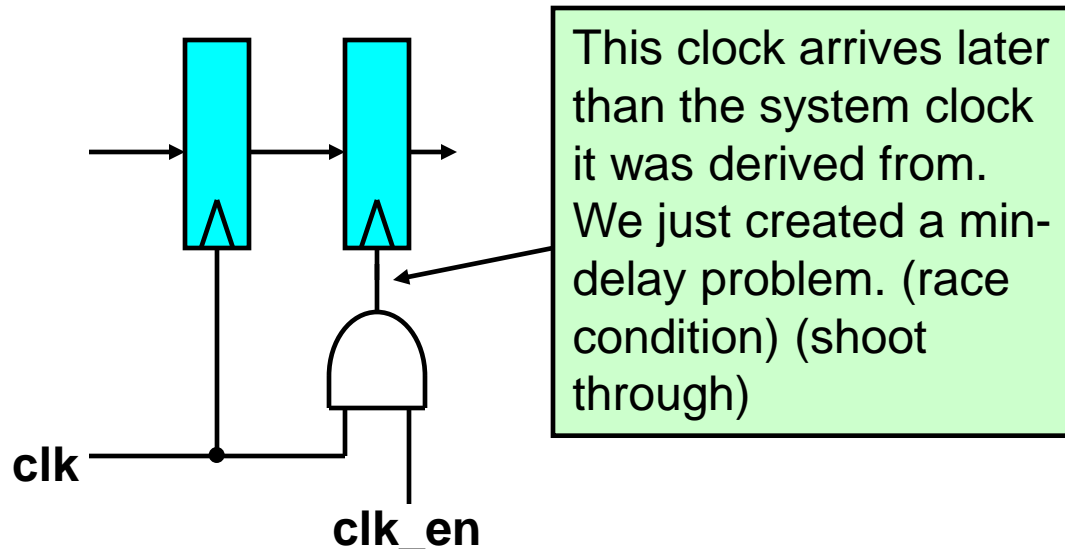
But perhaps what you meant
was if neither **a** nor **b** then you
really don't care what **out** is.

Hey!, Why is **b** not used?



Gated Clocks

- Use only if necessary (e.g., for low-power)
 - Becoming more necessary with demand for many low power products



$$\text{Min_delay_slack} = \text{clk2q} - T_{\text{Hold}} - \text{Skew_Between_clks}$$

Gated Clocks

Gated clock domains can't be treated lightly:

- 1.) Skew between domains
- 2.) Loading of each domain. How much capacitance is on it? What is its rise/fall times
- 3.) RC in route. Is it routed in APR like any old signal, or does it have priority?

Clocks are not signals...don't treat them as if they were.

- 1.) Use clock tree synthesis (CTS) within the APR tool to balance clock network (usually the job of a trained APR expert)
- 2.) Paranoid control freaks (like me) like to generate the gated domains in custom logic (like clock reset unit). Then let CTS do a balanced distribution in the APR tool.

Our guest lecturer will cover some of this material....

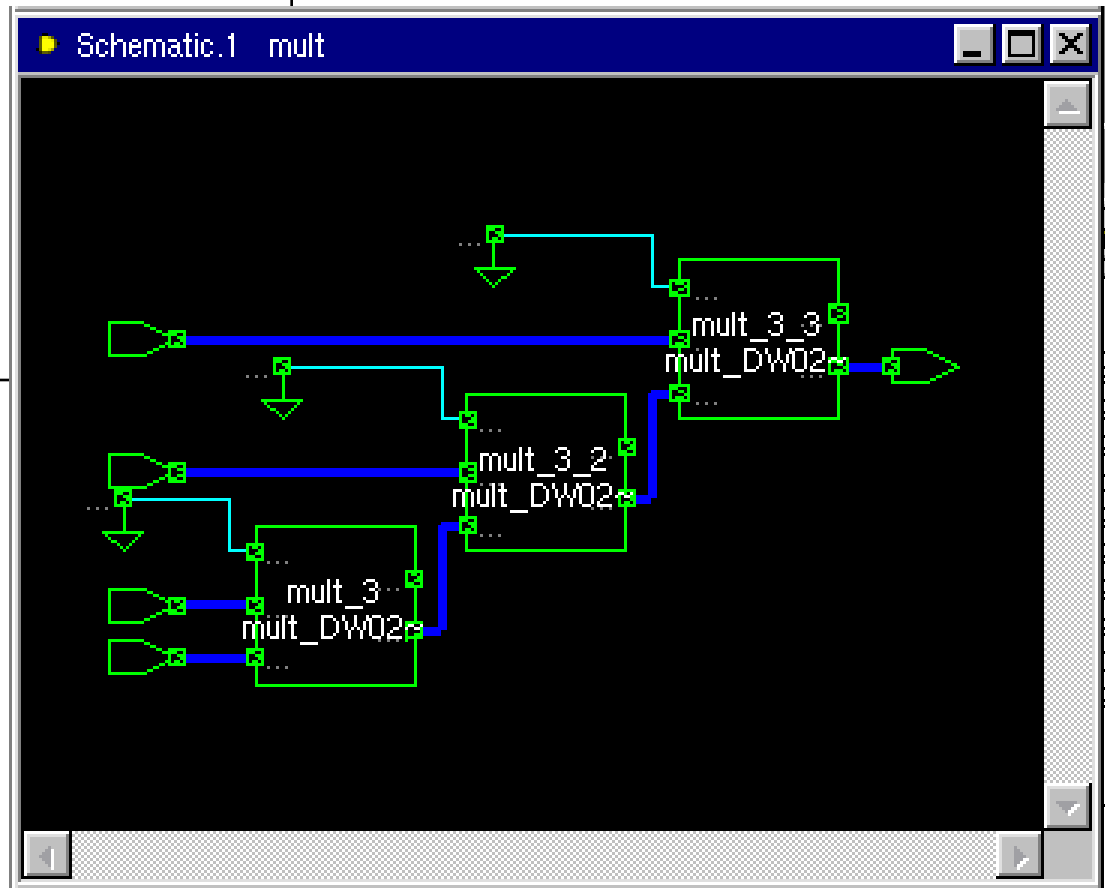
Chain Multiplier

```
module mult(output reg [31:0] out,  
            input [31:0] a, b, c, d);
```

```
    always@(*) begin  
        out = ((a * b) * c) * d;  
    end
```

```
endmodule
```

Area: 47381
Delay: 8.37



Tree Multiplier

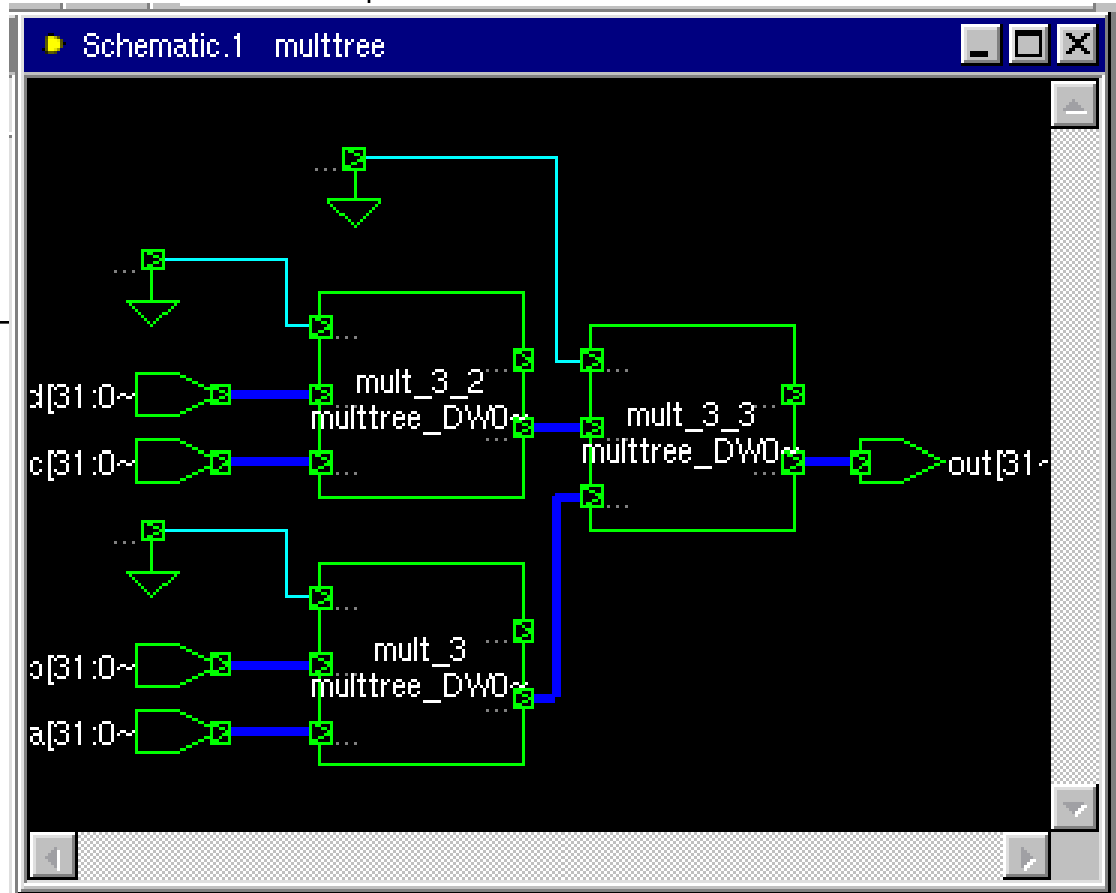
```
module multtree(output reg [31:0] out,  
               input [31:0] a, b, c, d);
```

```
  always@(*) begin  
    out = (a * b) * (c * d);  
  end
```

```
endmodule
```

Area: 47590

Delay: 5.75 vs 8.37



Multi-Cycle Shared Multiplier

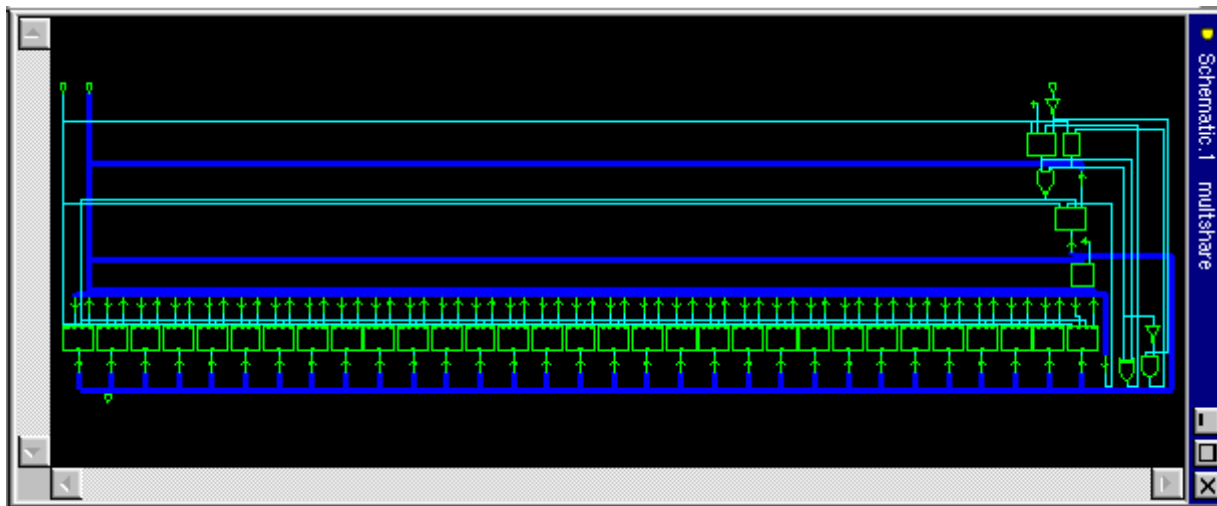
```
module multshare(output reg [31:0] out,  
                 input [31:0] in, input clk, rst);  
  
  reg [31:0] multval;  
  reg [1:0] cycle;  
  always @(posedge clk) begin  
    if (rst) cycle <= 0;  
    else cycle <= cycle + 1;  
    out <= multval;  
  end  
  
  always @(*) begin  
    if (cycle == 2'b0) multval = in;  
    else multval = in * out;  
  end  
endmodule
```

Multi-Cycle Shared Multiplier (results)

Area: 15990 vs 47500

Delay: 4×3.14

4 clocks, minimum period 3.14



Shared Conditional Multiplier

```
module multcond1(output reg [31:0] out,  
                 input [31:0] a, b, c, d, input sel);
```

```
always @(*) begin
```

```
  if (sel) out = a * b;
```

```
  else out = c * d;
```

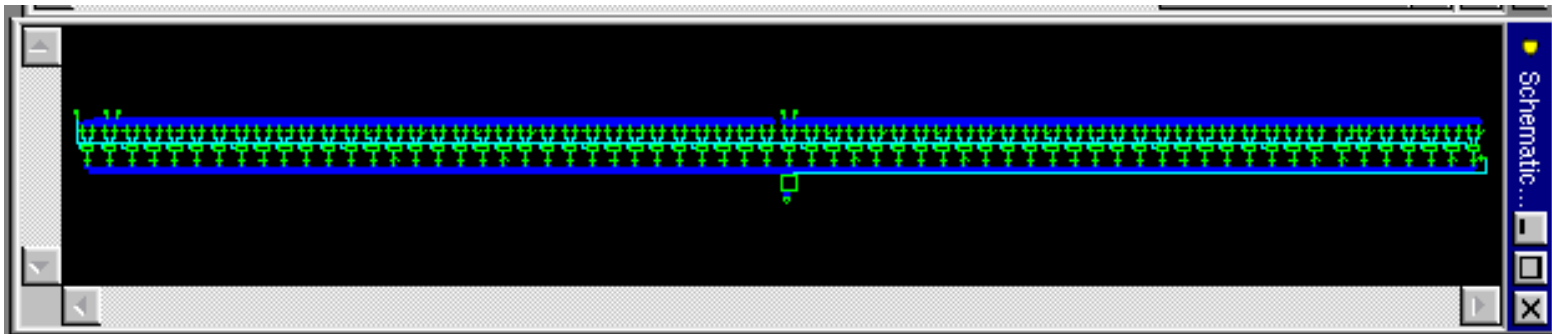
Mutually exclusive use of the multiply

```
end
```

```
endmodule
```

Area: 15565

Delay: 3.14

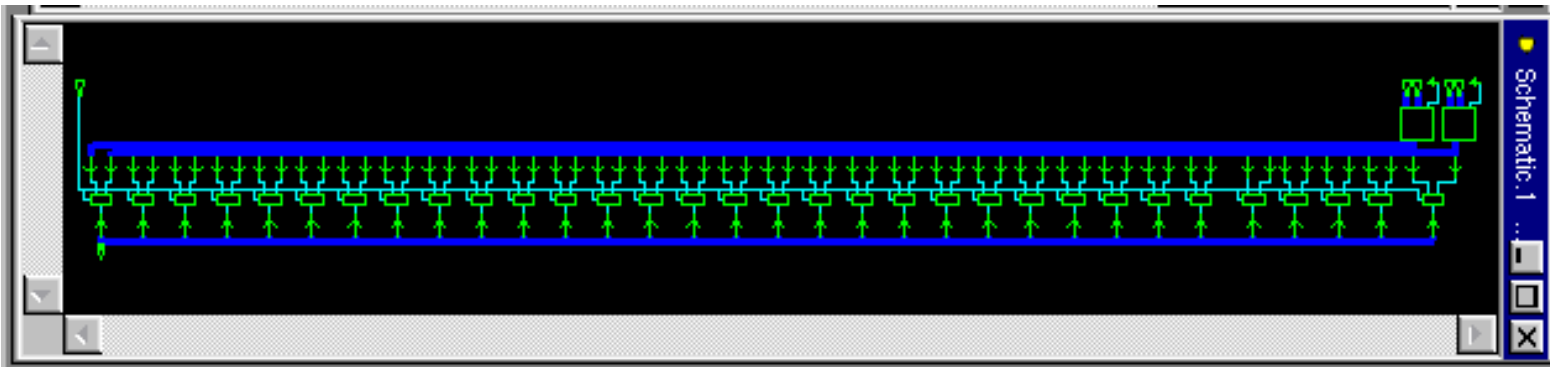


Selected Conditional Multiplier [1]

```
module multcond2(output reg [31:0] out,  
                 input [31:0] a, b, c, d, input sel);  
  
wire [31:0] m1, m2;  
assign m1 = a * b;  
assign m2 = c * d;  
  
always @(*) begin  
    if (sel) out = m1;  
    else out = m2;  
end  
  
endmodule
```

Selected Conditional Multiplier [1]

- Area: 30764 vs. 15565
- Delay: 3.02 vs. 3.14
- Why is the area larger and delay lower?



Decoder Using Indexing

Example 3-1 Verilog for Decoder Using Indexing

```
module decoder_index (in1, out1);  
  parameter N = 8;  
  parameter log2N = 3;  
  input [log2N-1:0] in1;  
  output [N-1:0] out1;  
  reg [N-1:0] out1;  
  always @(in1)  
  begin  
    out1 = 0;  
    out1[in1] = 1'b1;  
  end  
endmodule
```

What does synthesis do?

Think of Karnaugh Map

© 2001 Synopsys, Inc.

Decoder Using Loop

Example 3-3 Verilog for Decoder Using Loop

```
module decoder38_loop (in1, out1);  
  parameter N = 8;  
  parameter log2N = 3;  
  input [log2N-1:0] in1;  
  output [N-1:0] out1;  
  reg [N-1:0] out1;  
  integer i;  
  
  always @(in1)  
  begin  
    for(i=0;i<N;i=i+1)  
      out1[i] = (in1 == i);  
  end  
endmodule
```

For this implementation how are we directing synthesis to think?

Assign each bit to digital comparator result

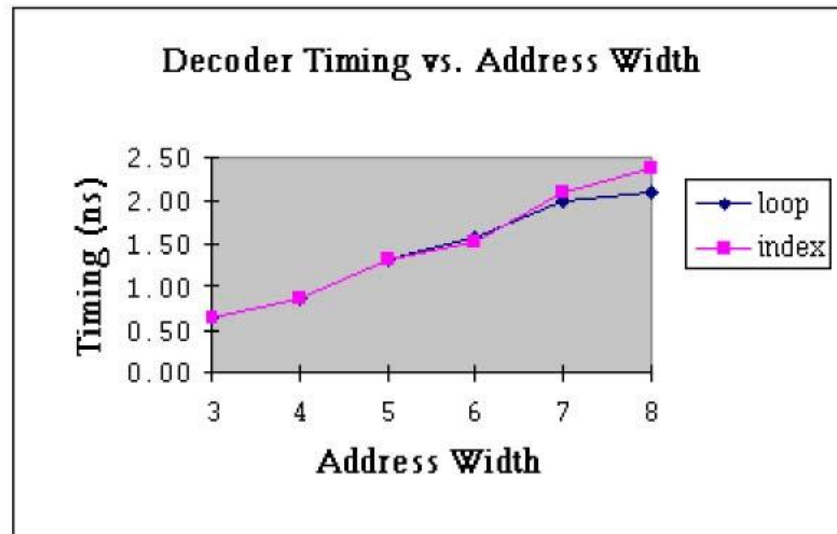
Decoder Verilog: Timing Comparison

Table 3-1 Timing Results for Decoder Coding Styles

Input Address Width	3	4	5	6	7	8
Index	0.64	0.86	1.33	1.52	2.11	2.37
Loop	0.64	0.86	1.33	1.57	1.98	2.10

© 2001 Synopsys, Inc.

Figure 3-1 Decoder Timing Results Versus Address Width



Loop method
Starts to look
advantageous

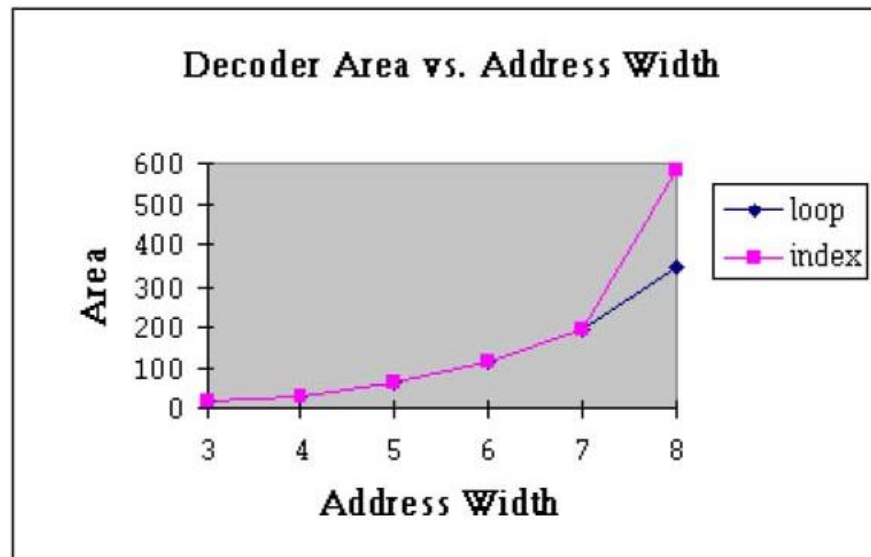
Decoder Verilog: Area Comparison

Table 3-2 Area Results for Decoder Coding Styles

Input Address Width	3	4	5	6	7	8
Index	18	29	61	115	195	583
Loop	18	30	61	116	195	346

© 2001 Synopsys, Inc.

Figure 3-2 Decoder Area Versus Address Width



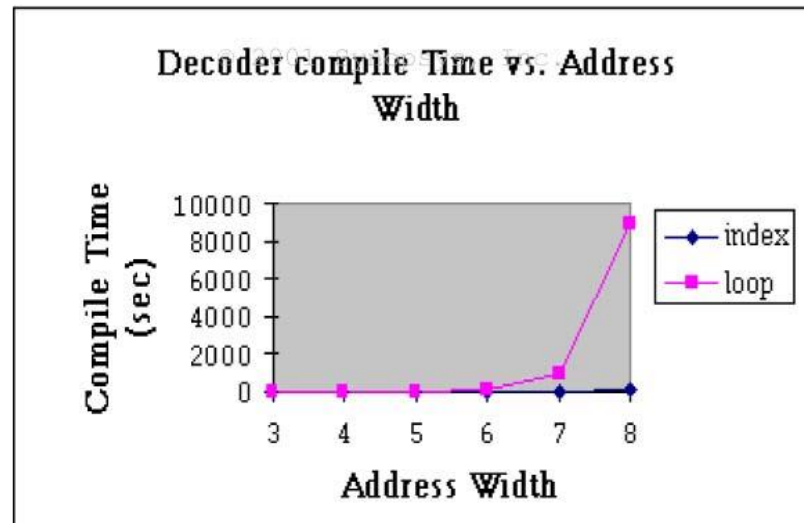
Loop method
Starts to look
advantageous

Decoder Verilog: Compile Time Comparison

Table 3-3 Compile Time (Seconds) for Decoder Coding Styles

Input Address Width	3	4	5	6	7	8
Index	2	3	11	18	58	132
Loop	16	13	42	163	946	9000

© 2001 Synopsys, Inc.



Holy Mackerel Batman!

What is synthesis doing?

Why is it working so long and hard?

Looking for shared terms

Late-Arriving Signals

- After synthesis, we will identify the critical path(s) that are limiting the overall circuit speed.
- It is often that one signal to a datapath block is late arriving.
- This signal causes the critical path...how to mitigate?:
 - Circuit reorganization
 - ✓ Rewrite the code to restructure the circuit in a way that minimizes the delay with respect to the late arriving signal
 - Logic duplication
 - ✓ This is the classic speed-area trade-off. By duplicating logic, we can move signal dependencies ahead in the logic chain.

Logic Reorganization Example [1]

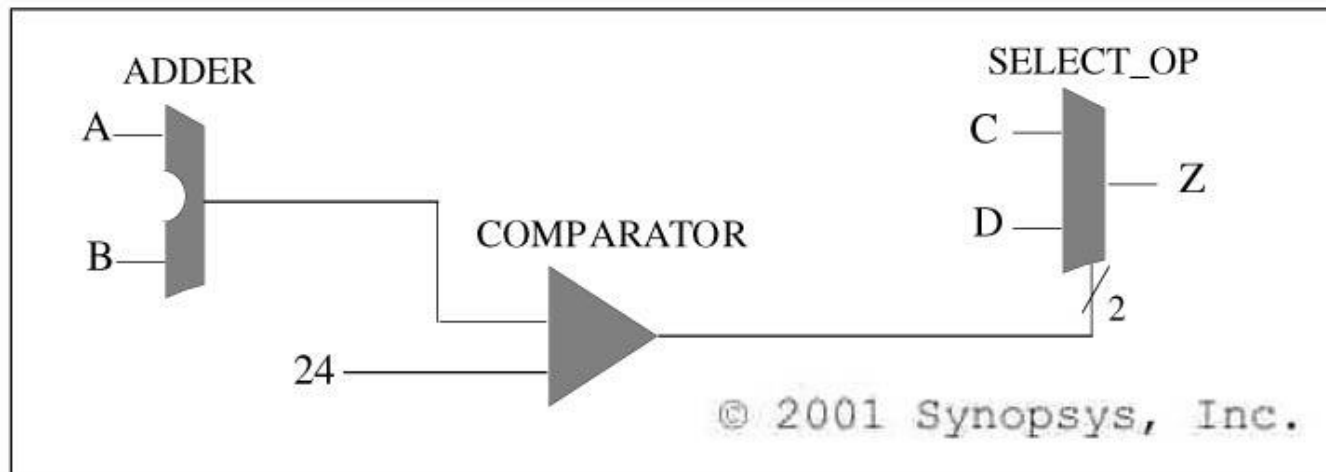
Example 4-5 Original Verilog With Operator in Conditional Expression

```
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
```

© 2001 Synopsys, Inc.

Logic Reorganization Example [2]

Figure 4-3 Structure Implied by Original HDL With Late Arriving A Signal



What can we do if **A** is the late-arriving signal?

Logic Reorganization Example [3]

Example 4-7 Improved Verilog With Operator in Conditional Expression

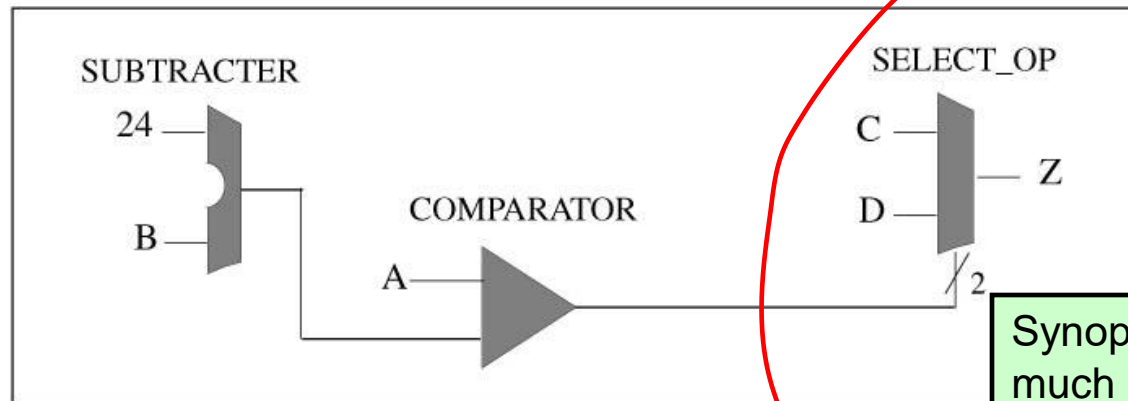
```
module cond_oper_improved (A, B, C, D, Z);  
  parameter N = 8;  
  input [N-1:0] A, B, C, D; // A is late arriving  
  output [N-1:0] Z;  
  
  reg [N-1:0] Z;  
  
  always @(A or B or C or D)  
  begin  
    if (A < 24 - B)  
      Z <= C;  
    else  
      Z <= D;  
    end  
  endmodule
```

That's right! We have to do the math, and re-arrange the equation so the comparison does not involve an arithmetic operation on the late arriving signal.

© 2001 Synopsys, Inc.

Logic Reorganization Example [4]

Figure 4-4 Structure Implied by Improved HDL With A as Input to Comparator



Why the area improvement?

Synopsys didn't spend so much effort upsizing gates to try to make transistors faster. This new design is **faster**, **lower area**, and **lower power**

Table 4-2 Timing and Area Results for Conditional Op

	Data Arrival Time	Area
Original Design	4.33	411.1
Improved Design	3.89	271.0

Logic Duplication Example [1]

Example 4-1 Original Verilog Before Logic Duplication

```
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;

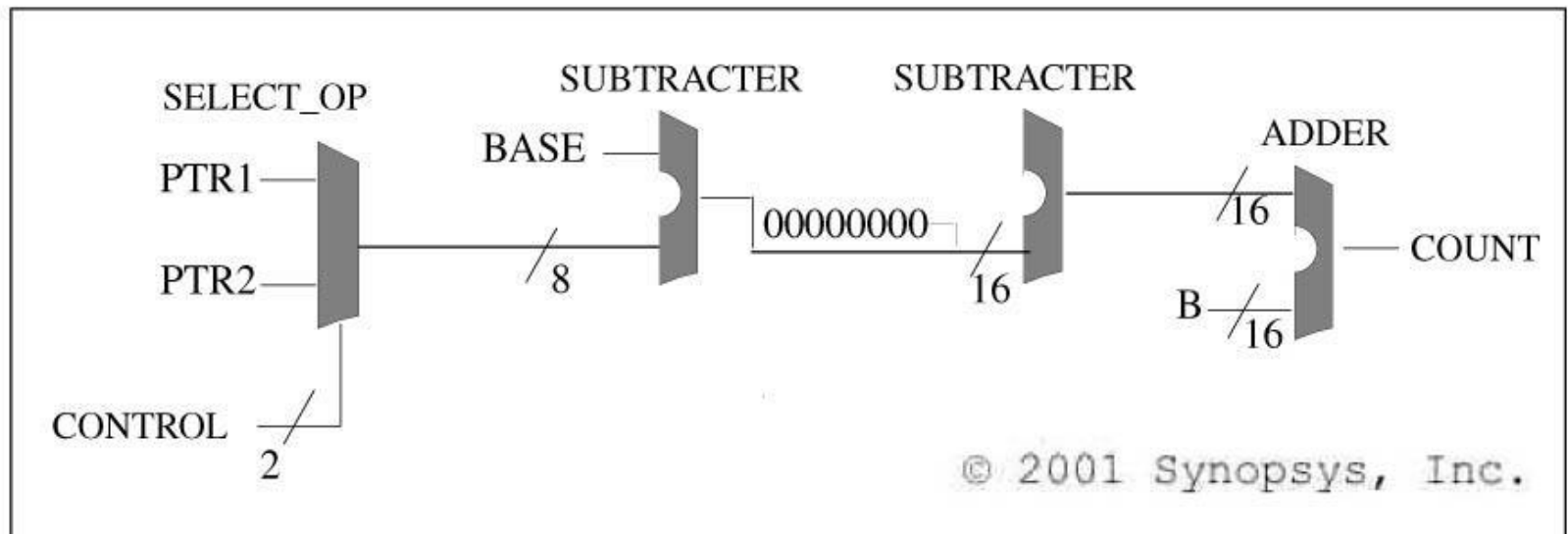
parameter [7:0] BASE = 8'b10000000;
wire [7:0] PTR, OFFSET;
wire [15:0] ADDR;

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
```

Logic Duplication Example [2]

Figure 4-1 Structure Implied by Original HDL Before Logic Duplication



What if *control* is the late arriving signal?

Logic Duplication Example [3]

Example 4-3 Improved Verilog With Data Path Duplicated

```
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
```

Logic Duplication Example [4]

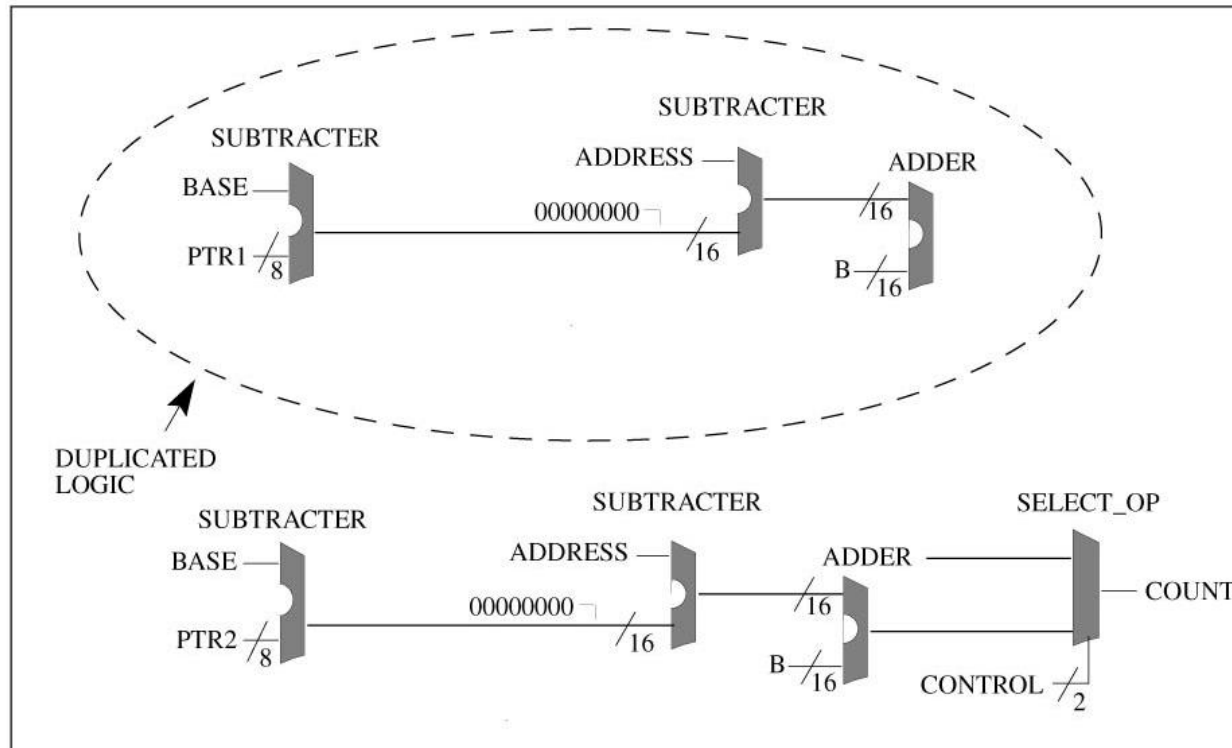


Table 4-1 *Timing and Area Results for Data-Path Duplication*

	Data Arrival Time	Area
Original Design	5.23	1057
Improved Design	2.33	1622

Exercise

- Assume we are implementing the below code, and *cin* is the late arriving signal? How can we optimize the resulting hardware for speed? At what cost?

```
reg [3:0] a, b;  
reg [4:0] y;  
reg cin;  
  
y = a + b + cin;
```

Exercise

- Revise to maximize performance wrt *late*

```
reg [3:0] state;  
reg late, y, x1, x2, x3;  
  
case(state)  
  SOME_STATE:  
    if (late) y = x1;  
    else y = x2;  
  default:  
    if (late) y = x1;  
    else y = x3;  
endcase
```

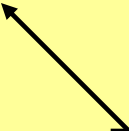
Actually, there is nothing you can really do here. This is simple boolean logic, and synopsys already does a good job optimizing for late arriving.

Coding optimizations often apply more to larger functions (like arithmetic operations, & comparisons). Than to boolean logic.

Mixing Flip-Flop Styles (1)

- What will this synthesize to?

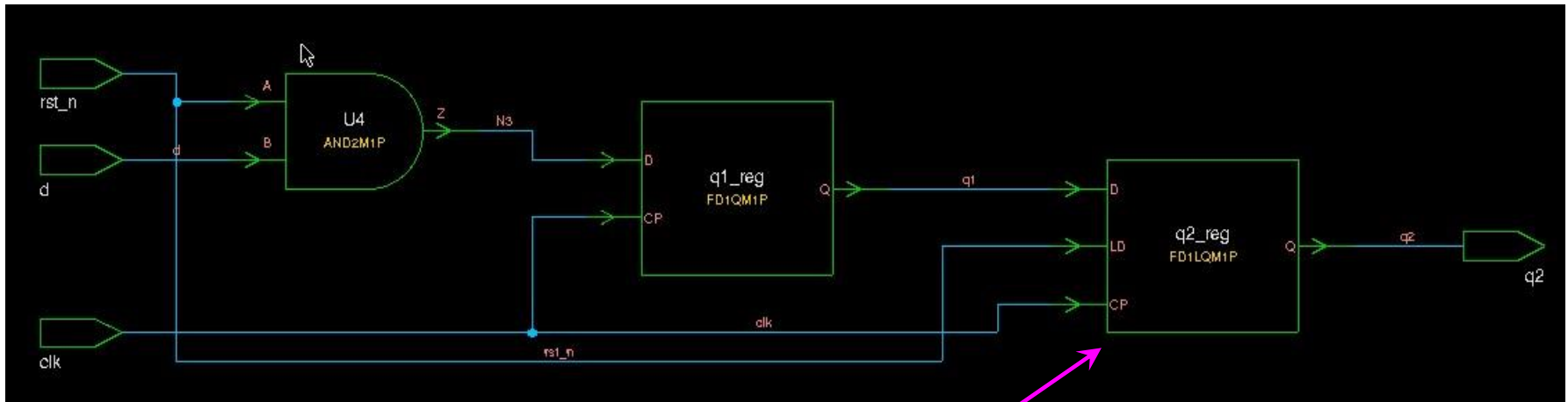
```
module badFFstyle (output reg q2, input d, clk, rst_n);  
  reg q1;  
  
  always @(posedge clk)  
    if (!rst_n)  
      q1 <= 1'b0;  
    else begin  
      q1 <= d;  
      q2 <= q1;  
    end  
endmodule
```



If !rst_n then q2 is not assigned...
It has to keep its prior value

Flip-Flop Synthesis (1)

- Area = 59.0



Note: q2 uses an enable flop (has mux built inside)
enabled off `rst_n`

Mixing Flip-Flop Styles (2)

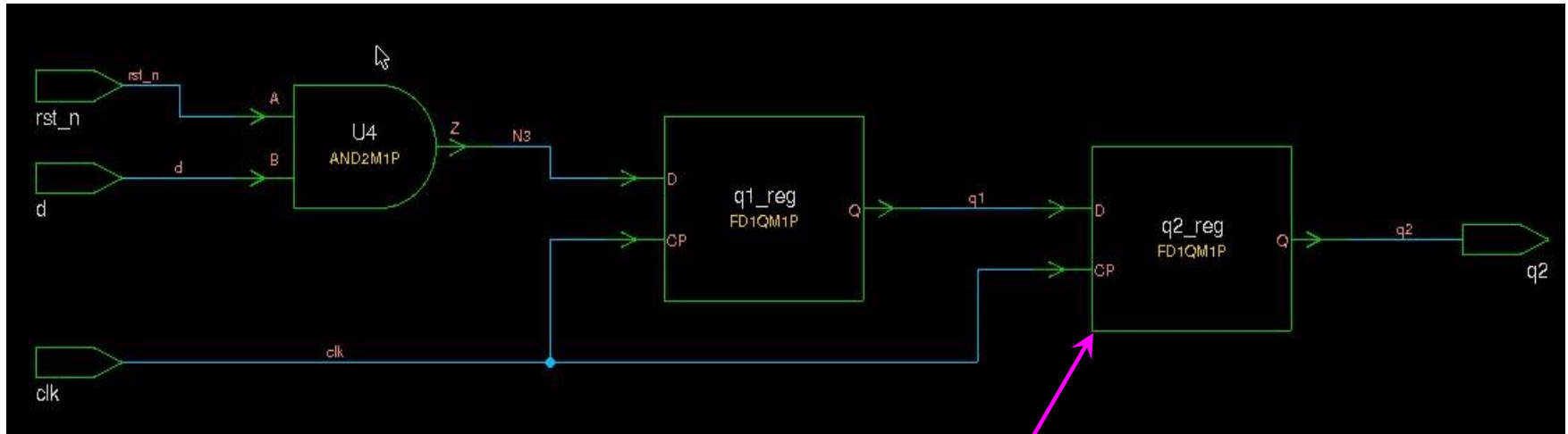
```
module goodFFstyle (output reg q2, input d, clk, rst_n);  
  reg q1;  
  
  always @(posedge clk)  
    if (!rst_n) q1 <= 1'b0;  
    else q1 <= d;  
  
  always @(posedge clk)  
    q2 <= q1;  
  
endmodule
```

Only combine like flops (same reset structure) in a single **always** block.

If their reset structure differs, split into separate **always** blocks as shown here.

Flip-Flop Synthesis (2)

- Area = 50.2 (85% of original area)



Note: q2 is now just a simple flop as intended

Flip-Flop Synthesis (3)

- Using **asynchronous** reset instead
 - Bad (same **always** block): Area = 58.0
 - Good (separate **always** block): Area = 49.1

Note asynch area less than synch, and cell count less (less interconnect)

