Digital System Design

Synthesizable Verilog Coding

Lecturer: 王景平

Advisor: Prof. An-Yeu Wu

Date: 2025.03.27

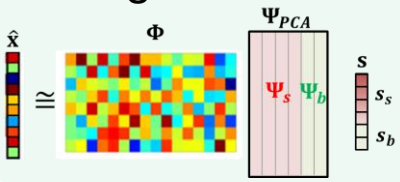
Outline

- Introduction to Synthesis
- Code for Synthesis
- Circuit-Level Coding Skills
- Coding Tips
- Check for Synthesizability

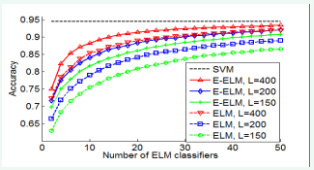
Cell-Based IC Design Flow

Algorithm

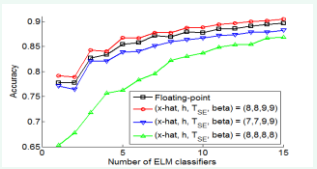
Algorithm



Floating-point Analysis



Fixed-point Analysis

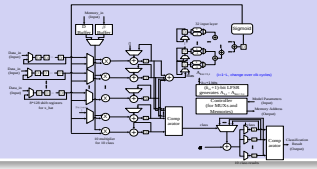


RTL

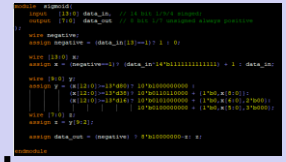
Spec

Parameter	Range	Descript ion
Dimension of input data (D_x)	128, 256, 512, 1024	MNIST Databas e
Number of hidden neurons (h_n)	1-256	

Architecture

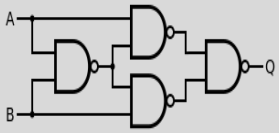


Verilog

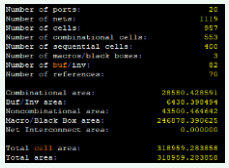


RTL Simulation

Synthesis



Gate-level Netlist

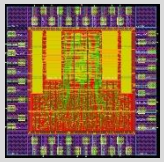


Gate-level Simulation

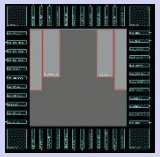
Power Optimization

Not cover

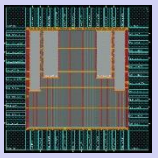
Layout



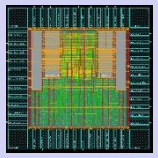
Floorplan



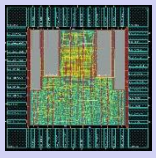
Powerplan



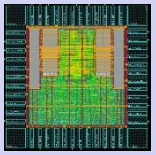
Placement



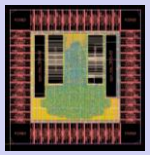
CTS



Route



DRC/LVS

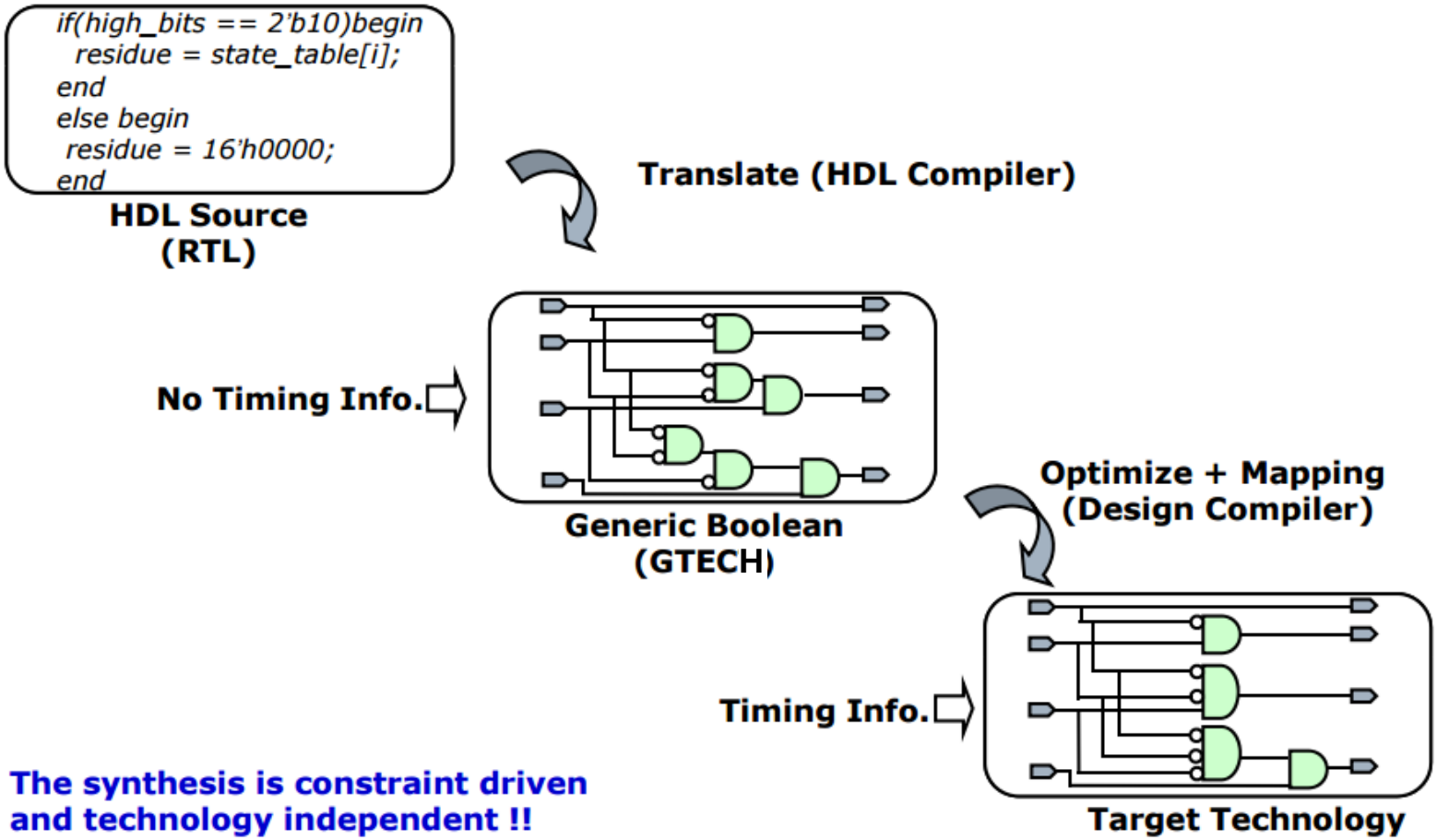


Introduction to Logic Synthesis (1/2)

- Process of converting a high-level description of design into an optimized gate-level representation.
- Logic synthesis uses standard cell library
  - Basic logic gates like **`and`**, **`or`**, and **`nor`**
  - Macro cells like adder, multiplexers, memory, and special flip-flops.
- Constrained driven
  - Area, timing, and power.

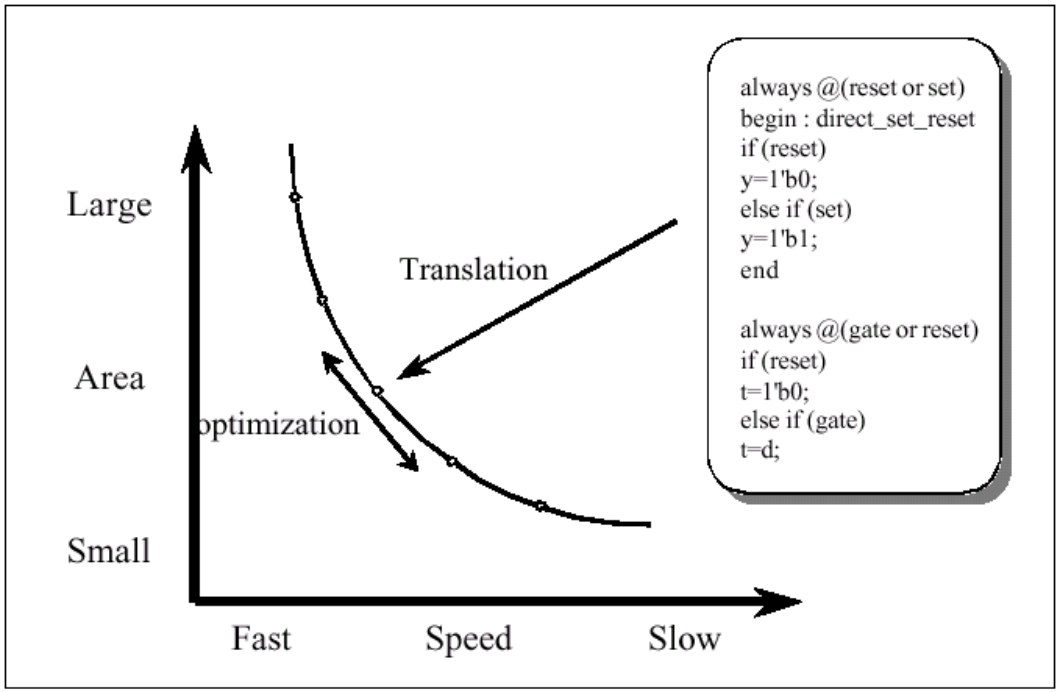
Introduction to Logic Synthesis (2/2)

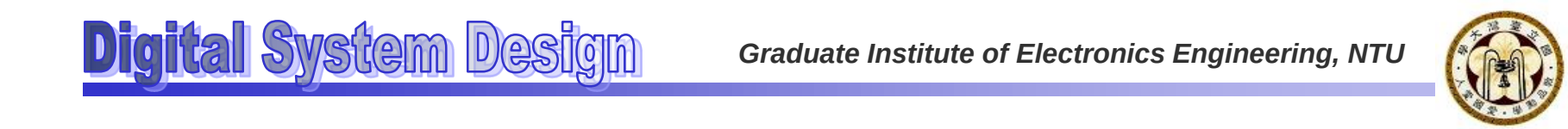
- Synthesis = translation + optimization + mapping



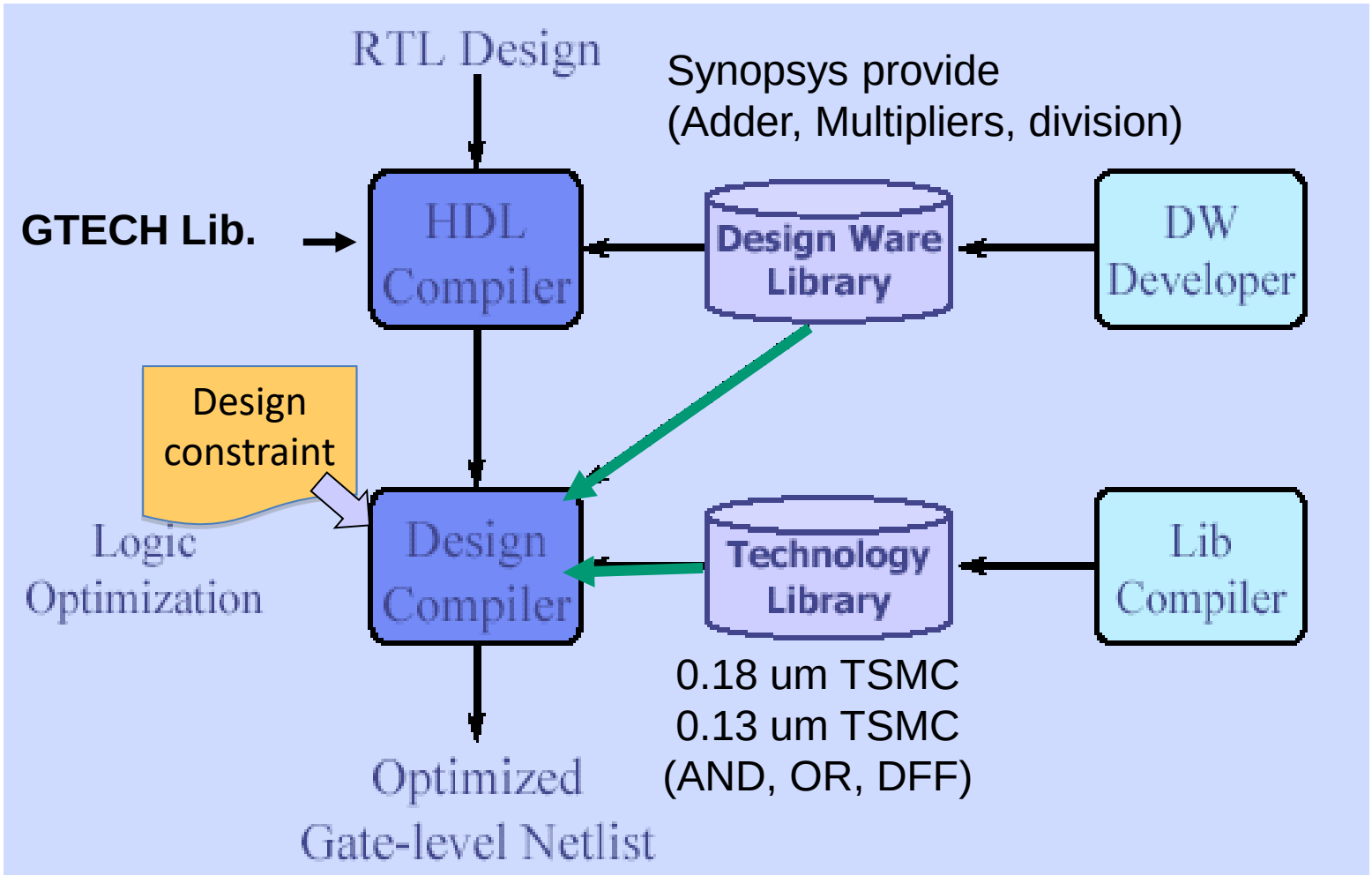
Trade-off between Speed and Area

- Synthesis is Constraint-Driven
- Technology Independent





Logic Synthesis Overview



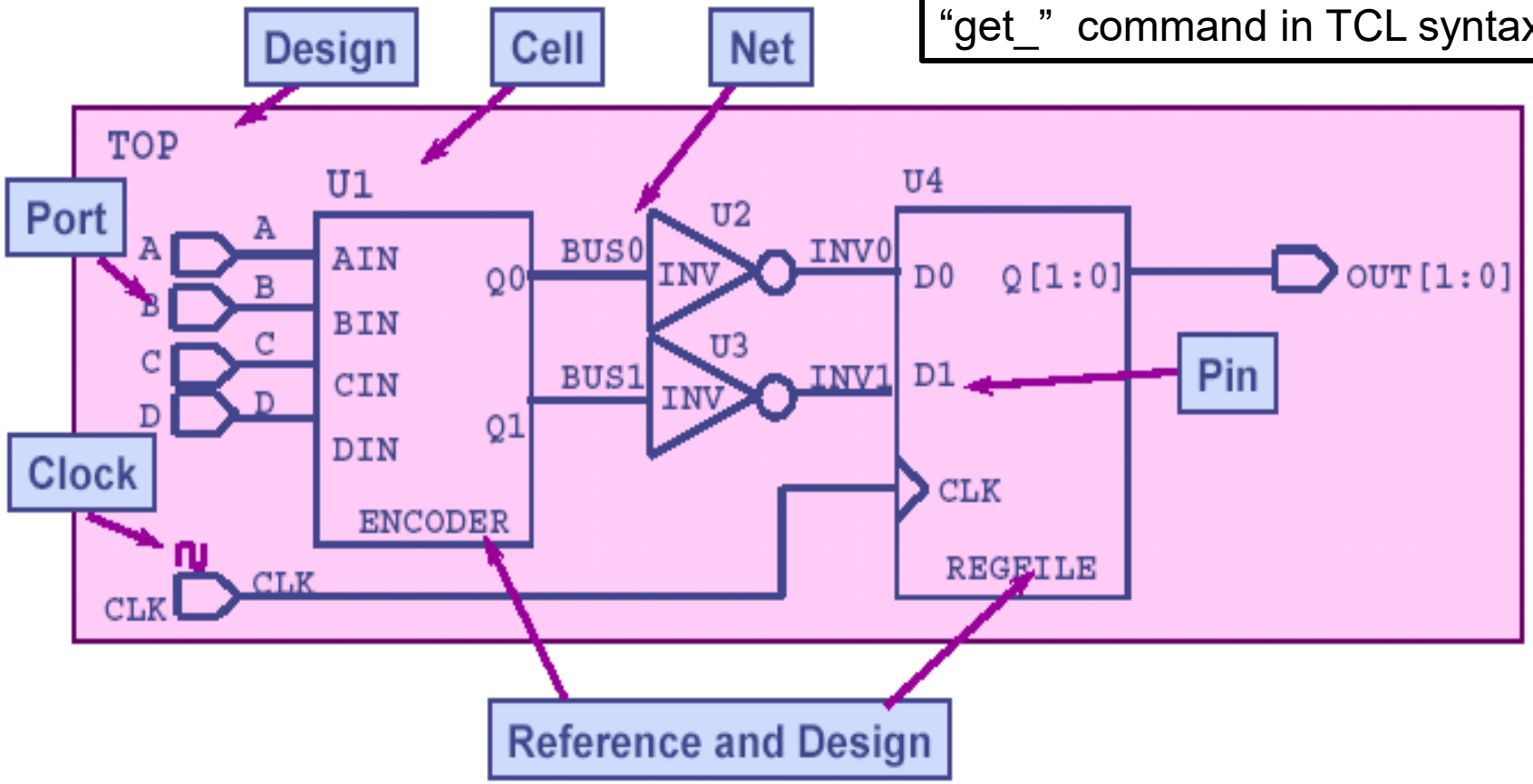
Design Objects

- Seven Types of Design Objects:

- **Design**: A circuit that performs one or more logical functions (top module)
- **Cell**: An instance of a design or library primitive within a design (instance)
- **Reference**: The name of the original design that a cell instance "points to"
- **Port**: The input or output of **a design**
- **Pin**: The input or output of **a cell**
- **Net**: The wire that connects ports to pins and/or pins to each other
- **Clock**: A timing reference object in DC memory which describes a waveform for timing analysis

Design Objects (Schematic Perspective)

Know this and correctly use the "get_" command in TCL syntax!



Design Objects (Verilog Perspective)

```
module TOP (A,B,C,D,CLK,OUT1);
  input A, B, C, D, CLK;
  output [1:0] OUT1;

  wire INV1,INV0,bus1,bus0;

  ENCODER U1 (.AIN (A), . . . .Q1 (bus1));

  INV  U2 (.A (BUS0), .Z( INV0)),
       U3 (.A( BUS1), .Z( INV1));

  REGFILE U4 (.D0 (INV0), .D1 (INV1), .CLK (CLK) );

endmodule
```

Design

Clock

Port

Net

Reference

Pin

Cell

Outline

- Introduction to Synthesis
- Code for Synthesis
- Circuit-Level Coding Skills
- Coding Tips
- Check for Synthesizability

Synthesizable Verilog Codes

- Verilog HDL is not only for synthesizable designs
- Not all kinds of Verilog constructs can be synthesized
- Only a subset of Verilog constructs can be synthesized and codes containing only this subset is synthesizable

Unsupported Syntax

- delay (#)
- initial
- repeat
- wait
- fork … join
- 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

Supported Syntax

- Verilog basis
  - Parameter declarations
  - Wire, reg declarations
  - Input, output declarations
  - Module instantiations
  - Gate instantiations

  - Continuous assignments
  - Always blocks
  - Condition statement (case, if…else...)

  - Task statements (*partially synthesizable*)
  - Function definitions (*partially synthesizable*)
  - For loop (*partially synthesizable*)

Conditions on X or Z

- Conditions on X or Z are treated as false

```
module compare(A, B);
  input  wire A;
  output 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 line 7 in file "compare.v" this may cause simulation to disagree with synthesis. (HDL-170)

Supported Operators

Bit-wise	`~, &, |, ^, ~^`
Unary reduction	`&, |, ^`
Logical	`!, &&, ||`
2's complement arithmetic	`+, -, *, /`
Relational	`>, <, >=, <=`
Equality	`==, !=`
Shift	`>>, <<, >>>, <<<`
Conditional	`?:`

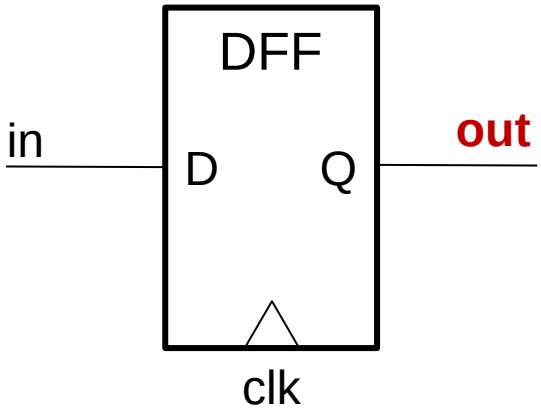
Outline

- Introduction to Synthesis
- Code for Synthesis
- Circuit-Level Coding Skills
- Coding Tips
- Check for Synthesizability

Mapping Sequential Circuits

- Sequential circuits are mapped to flip-flops
- Signals are mapped to flip-flops' output ports

```
module DFF(
  input  wire clk,
  input  wire in,
  output reg  out
);
  always @(posedge clk)
    out <= in;
endmodule
```

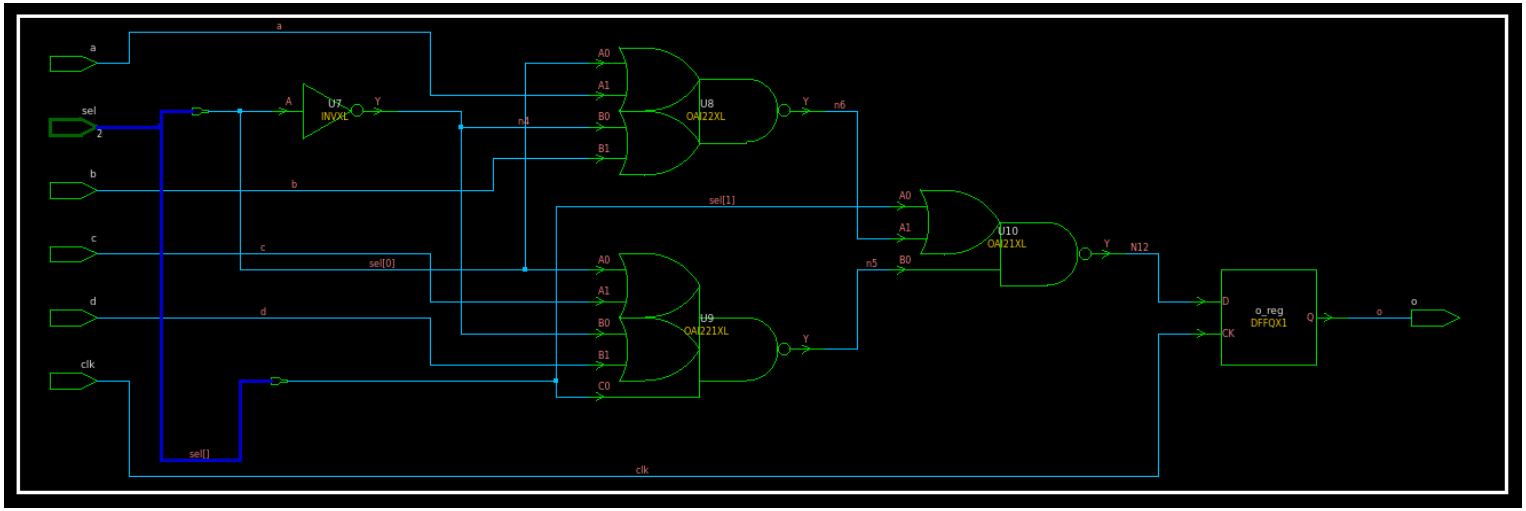


Condition: Mux vs. Priority Encoder

- if…else and case…endcase both imply priority

```
if      (sel == 2'b00) o = a;
else if (sel == 2'b01) o = b;
else if (sel == 2'b10) o = c;
else                   o = d;
```

```
case (sel)
  2'b00: o = a;
  2'b01: o = b;
  2'b10: o = c;
  2'b11: o = d;
endcase
```

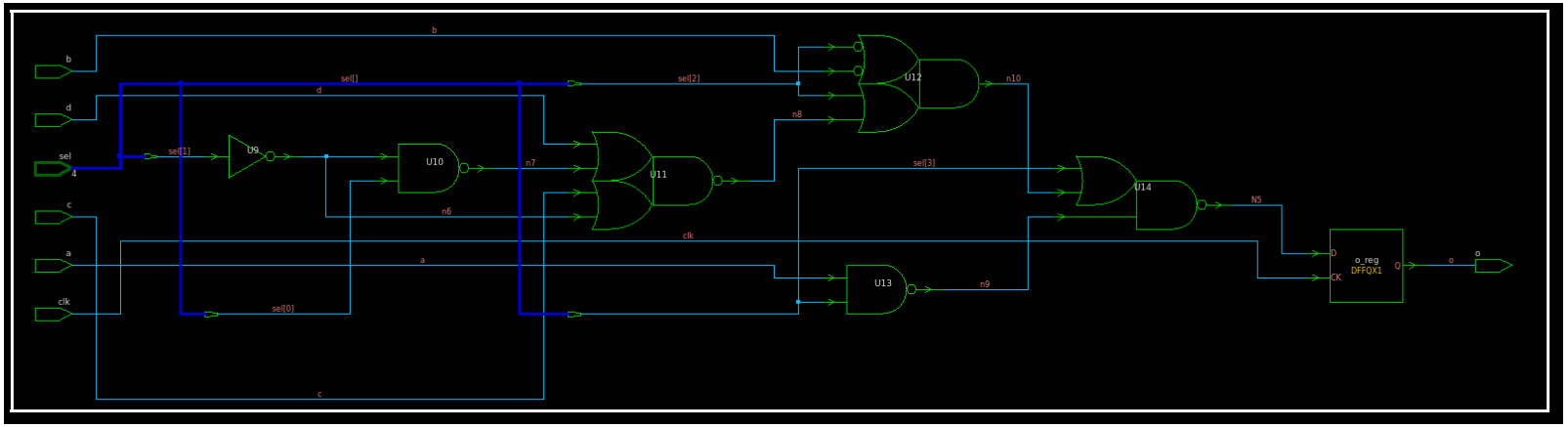


Case Study: One-Hot Decoder

- Input: 4 bits(1000/0100/0010/0001)

```
if      (sel[3]) o = a;
else if (sel[2]) o = b;
else if (sel[1]) o = c;
else if (sel[0]) o = d;
else             o = 1;
```

```
case (1'b1)
  sel[3]:  o = a;
  sel[2]:  o = b;
  sel[1]:  o = c;
  sel[0]:  o = d;
  default: o = 1;
endcase
```



Case Modeling

- case/casez/casex

```
case (1'b1)
  sel[3]:  o = a;
  sel[2]:  o = b;
  sel[1]:  o = c;
  sel[0]:  o = d;
  default: o = 1;
endcase
```

```
case (sel)
  4'b1???: o = a;
  4'b?1??: o = b;
  4'b??1?: o = c;
  4'b???1: o = d;
  default: o = 1;
endcase
```

```
casez (sel)
  4'b1zzz: o = a;
  4'bz1zz: o = b;
  4'bzz1z: o = c;
  4'bzzz1: o = d;
  default: o = 1;
endcase
```

case		0	1	x	z
	0	1	0	0	0
	1	0	1	0	0
	x	0	0	1	0
	z	0	0	0	1

casez		0	1	x	z
	0	1	0	0	1
	1	0	1	0	1
	x	0	0	1	1
	z	1	1	1	1

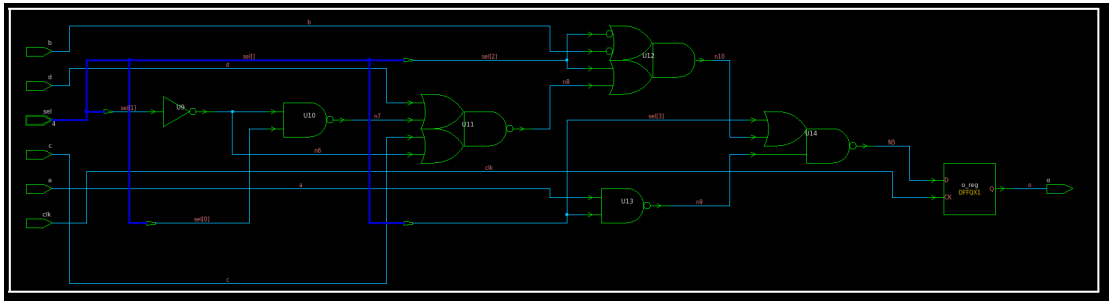
casex		0	1	x	z
	0	1	0	1	1
	1	0	1	1	1
	x	1	1	1	1
	z	1	1	1	1

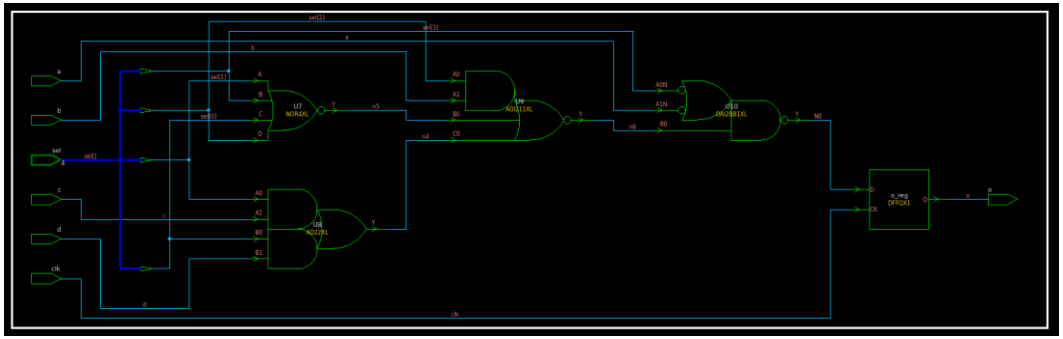
- Be careful with the mismatch between synthesis and simulation

Synopsys Synthesis Attributes

- parallel_case

```
case (1'b1) // synopsys parallel_case
  sel[3]:  o = a;
  sel[2]:  o = b;
  sel[1]:  o = c;
  sel[0]:  o = d;
  default: o = 1;
endcase
```

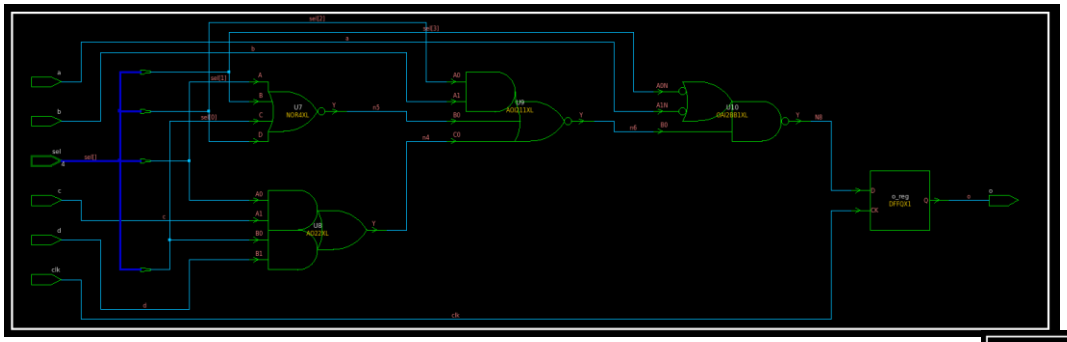


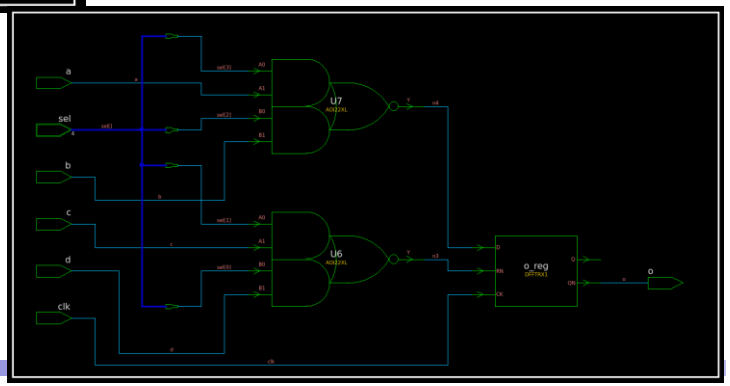


Synopsys Synthesis Attributes

- full_case

```
case (1'b1) // synopsys parallel_case full_case
  sel[3]:  o = a;
  sel[2]:  o = b;
  sel[1]:  o = c;
  sel[0]:  o = d;
endcase
```

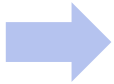




Mapping of *for* Loop

- Provide a shorter way to express a series of statements.
- Loop index variables must be integer type
- Step, start & end value must be constant
- For synthesis tools, for loops are "***unrolled***", and then synthesized.

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



```
always @(*) begin
  c[0] = a[0] & b[0];
  c[1] = a[1] & b[1];
  c[2] = a[2] & b[2];
  c[3] = a[3] & b[3];
end
```

Using *generate* Statement

- Generalize form of *for loop* expression
- Loop index variables must be *genvar* type (int)
- Step, start & end value must be constant
- Use keyward "*generate-endgenerate*" out of for loop

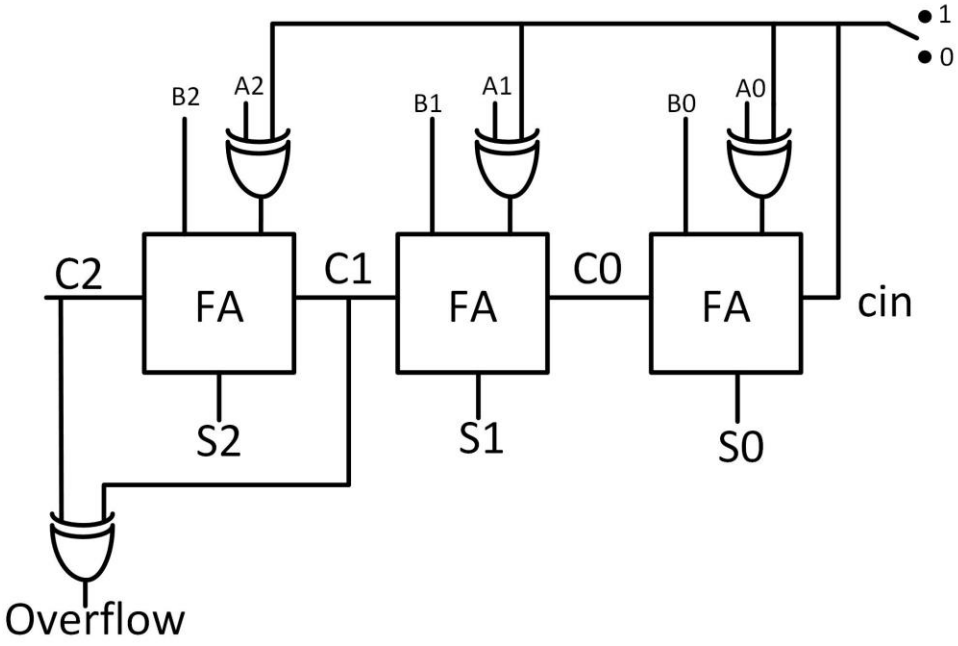
```
parameter p;
genvar gi;
generate
  if (p == 1) begin
    for (i = 0; i < 8; i = i+1) begin: Module_Array
      Module module(…);
    end
  end
endgenerate
```

Mapping of Logical Operators

- Binary Logical Operators (&,|,^,~^)
  - Mapped to logic gates directly
- Unary Logical Operators (&,|,^,~^,~,!)
  - Bit-wisely mapped to a logic gate
- Comparison Operators (>,<,>=,<=)
  - Mapped to a comparator
- Equality Operators (==, !=)
  - Mapped to a sequence of logic gate

Mapping of Arithmetic Operators (1/2)

- Addition (+)
  - Full adder
- Subtraction (-)
  - Full adder with 2's complement inverter
- Multiplication (*)
  - Full adder array
- Division & Modulo (/, %)
  - May need to instantiate DesignWare's modules
  - No direct mapping to any simple elements



Mapping of Arithmetic Operators (2/2)

- Multiplication & Division of Radix-2
  - Simplified as shift operations
  - Left shift by 1 bit: Multiply by 2
  - Right shift by 1 bit: Divide by 2
- Logic Shift operations (<<, >>)
  - Shift by constant: Simply wire assignment

```
// c is the same as b
assign b = a[7:0] >> 2;
assign c = {2'b0, a[7:2]};
```

  - Shift by variable: Shifter (HW1), e.g., a>>var
- Arithmetic Shift operations (<<<, >>>)

```
1100 >>> 1 = 0110
$signed(1100) >>> 1 = 1110
```

Bit Length of Arithmetic Operations

- (Signed or Unsigned) addition bit length
  - A(8 bits) + B(8 bits) → C(8+1 bits)
  - A(M bits) + B (N bits) → C(M+1 bits, if M>N)

```
wire [7:0] A, B;
wire [8:0] C;
assign C = {A[7], A} + {B[7], B};
assign C = $signed(A) + $signed(B);
```

```
wire signed [7:0] A, B;
wire signed [8:0] C;
assign C = A+B;
```

- (Signed or Unsigned) multiplication bit length
  - A(3 bits) x B(5 bits) → C(3 + 5 bits)

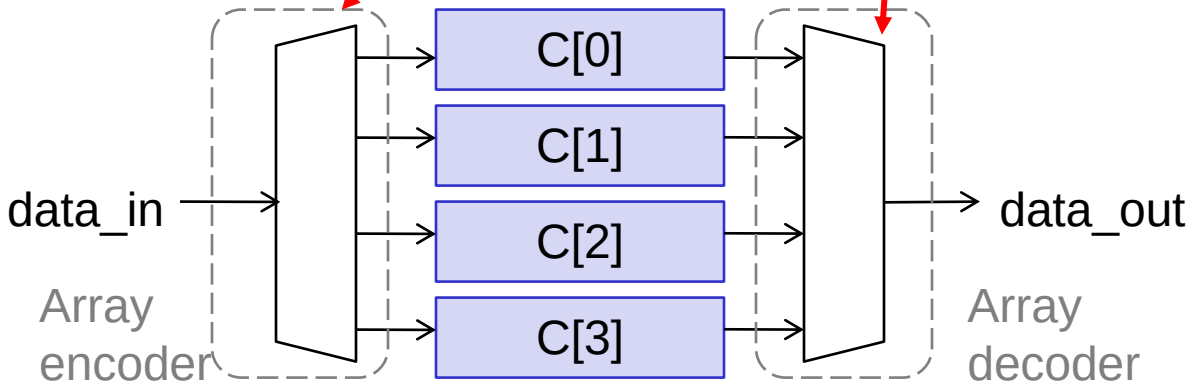
```
wire signed [2:0];
wire signed [4:0];
wire signed [7:0];
assign C = A * B;
```

Vector Array

- Vector array
  - Declaration and usage of vector array (4 vectors of 8 bits)

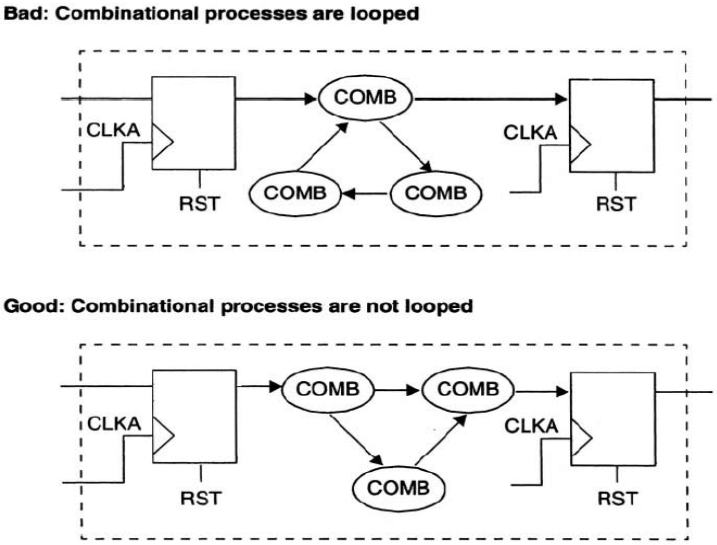
```
reg[7:0] C[0:3];
assign data_out = C[index_o];
always @(posedge clock) begin
    C[index_i] <= data_in;
end
```

  - Hardware translation



Combinational Loop

- An output of a combinational block feeds back to an input of the same block
- Should be avoided!




```
always @(*) begin
  b = a+1;
end
always @(*) begin
  a = b+1;
end
```

```
always @(*) begin
    a = a+2;
end
```

```
always@(posedge clk) begin
  a <= a+2;
end
```

```
always @(*)
  nxt_a <= a+2;
```

```
always @(posedge clk)
  a <= nxt_a;
```

Circuit-Level Refinement

- Be aware of the translation between circuits and codes
  - Operators means computation units
  - Datapath controllers mean FSM or multiplexers
- Plan a design using the block diagram instead of a pseudo code of data flow
  - Easy to understand your design cost (area/timing/critical path)

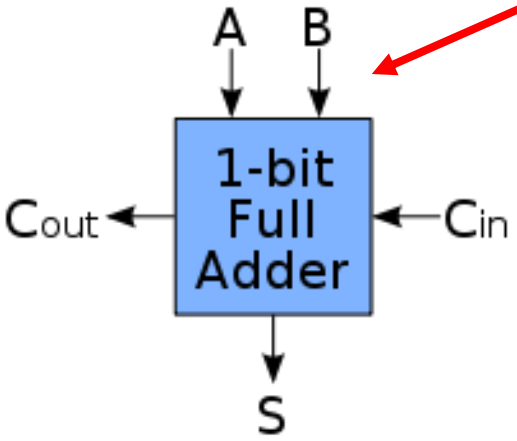
Use Operator Bit-Width Efficiently

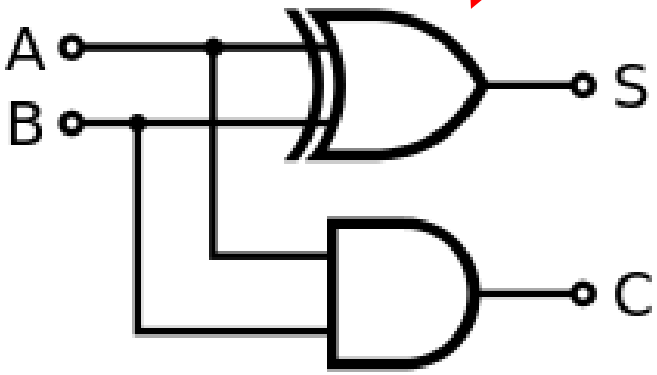
```
module test(a,b,out);
input [7:0] a,b;
output [8:0] out;
assign out=add_lt_10(a,b);

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
```

Redundant bits not involved

Propagate Constant Value

```
parameter size = 8;
wire  [3:0] a,b,c,d,e;
assign c = size + 2;   // constant
assign d = a + 1;      // incrementer
assign e = a + b;      // adder
```





Data-Path Duplication (1/2)

No_dulpicated

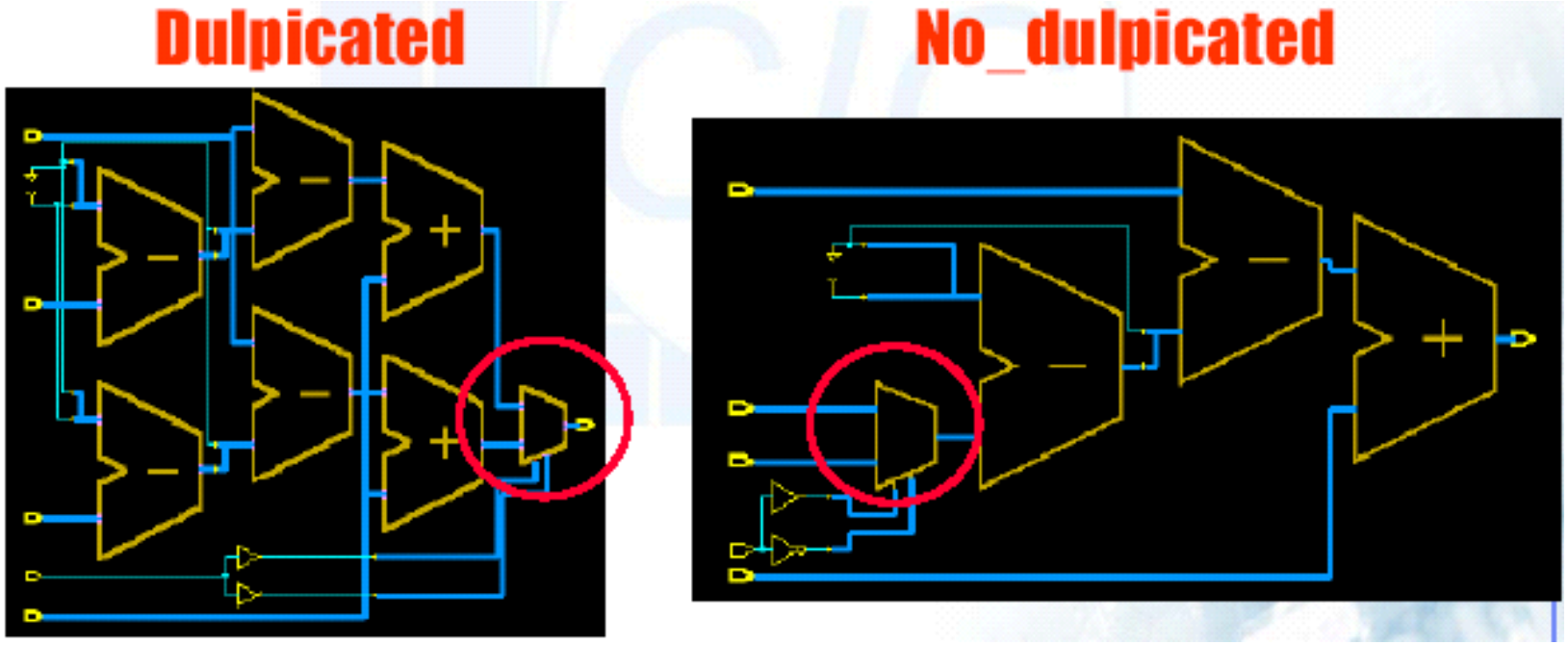
```
module BEFORE (ADDRESS, PTR1, PTR2, B, CONTROL, COUNT);
input [7:0] PTR1,PTR2;
input [15:0] ADDRESS, B;
input CONTROL;          // CONTROL is late arriving
output [15:0] COUNT;
parameter [7:0] BASE = 8'b10000000;
wire [7:0] PTR, OFFSET;
wire [15:0] ADDR;
assign PTR = (CONTROL == 1'b1) ? PTR1 : PTR2;
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
```

Data-Path Duplication (2/2)

- We assume that signal “CONTROL” is the latest arrival pin.
- Sacrifice area to gain latency reduction



Comparison Refinement (1/2)

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

Before_improved

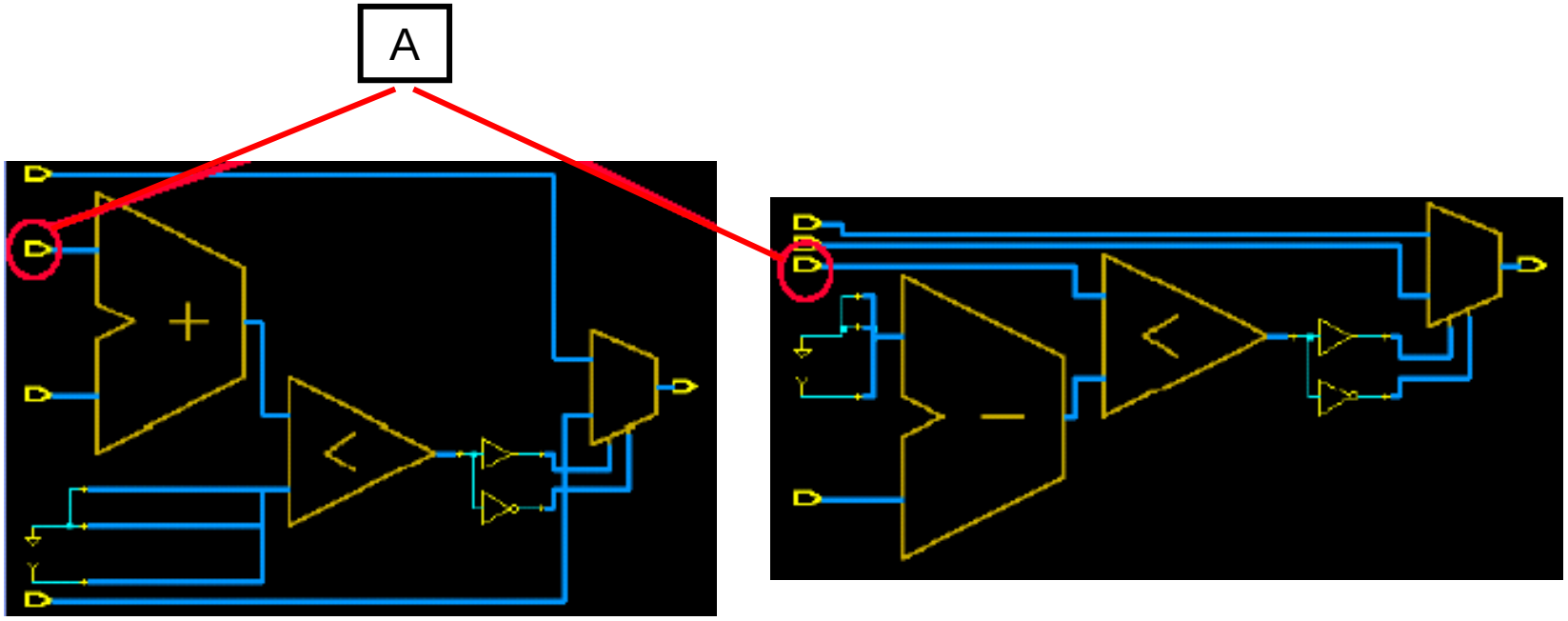
```
module cond_oper(A, B, C, D, Z);
parameter N = 8;
input [N-1:0] A, B, C, D;
//A is late arriving
output [N-1:0] Z;
reg [N-1:0] Z;

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
```

Comparison Refinement (2/2)

- Latency is reduced

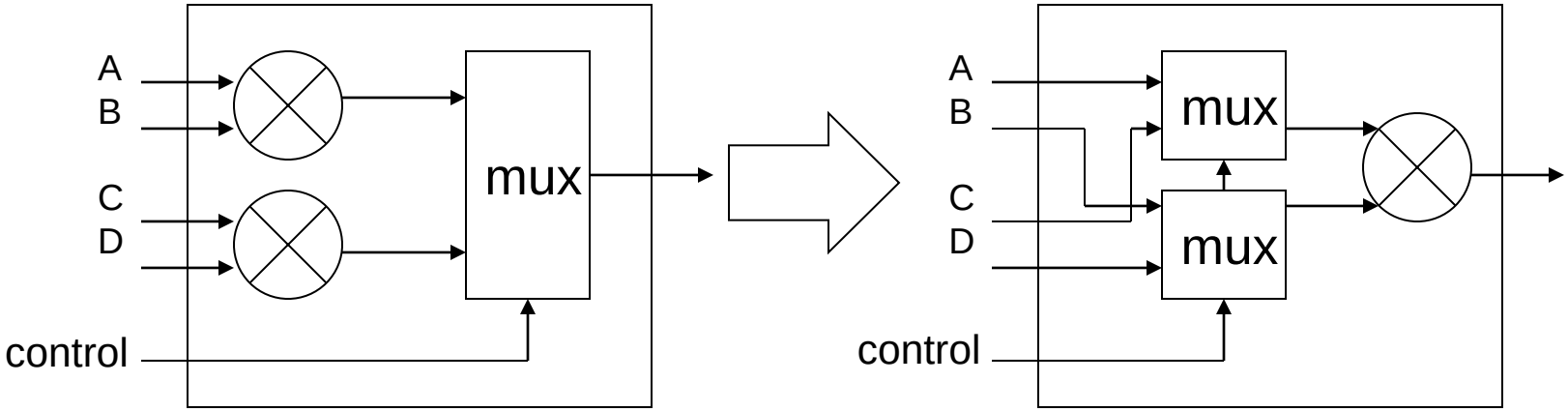


Before_improved

Improved

Resource Reusing

- Keep sharable resources in the same block




```
always @(*) begin
  if (control) z = a*b;
  else         z = c*d;
end
```

```
always @(*) begin
  z = (control ? a : c)
    * (control ? b : d);
end
```

Outline

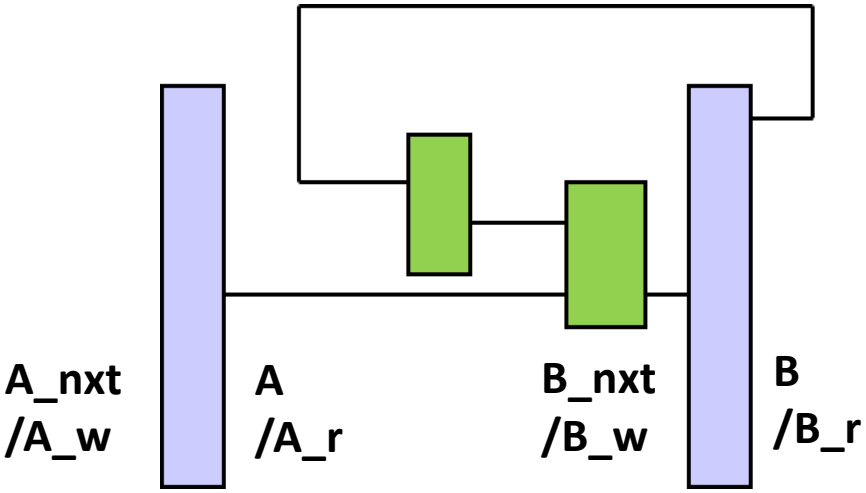
- Introduction to Synthesis
- Code for Synthesis
- Circuit-Level Coding Skills
- Coding Tips
- Check for Synthesizability

Comb./Seq. Partition for Synthesis

- Separate the design into two parts
  - Pure combinational: Logic Propagation
  - Pure sequential: Flip-Flops
- Avoid misunderstanding by synthesis tools
- Easily tracing of next/current state values after synthesis

Separate Combinational and Sequential Part

- Separate combinational circuit and sequential circuit make you keep in mind what your hardware architecture looks like.
- Sequential
  - Only D-flipflop!
  - Simple transition function
- Combinational
  - Use value from flipflop and assign it to variable a_nxt



Register All Outputs

- For each subblock of a hierarchical macro design, register all output signals from the subblock and input signals from testbench.
  - Let each module have enough cycle time to complete calculation without violating setup

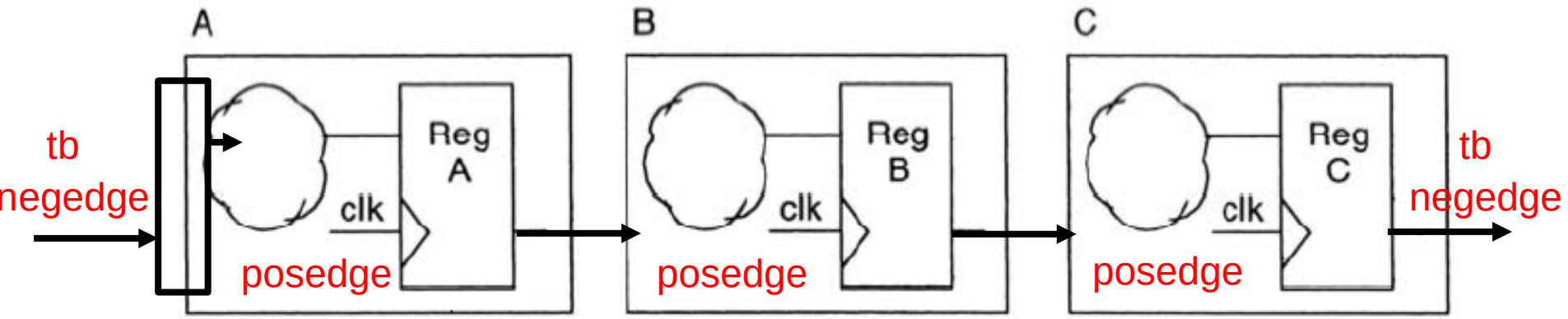
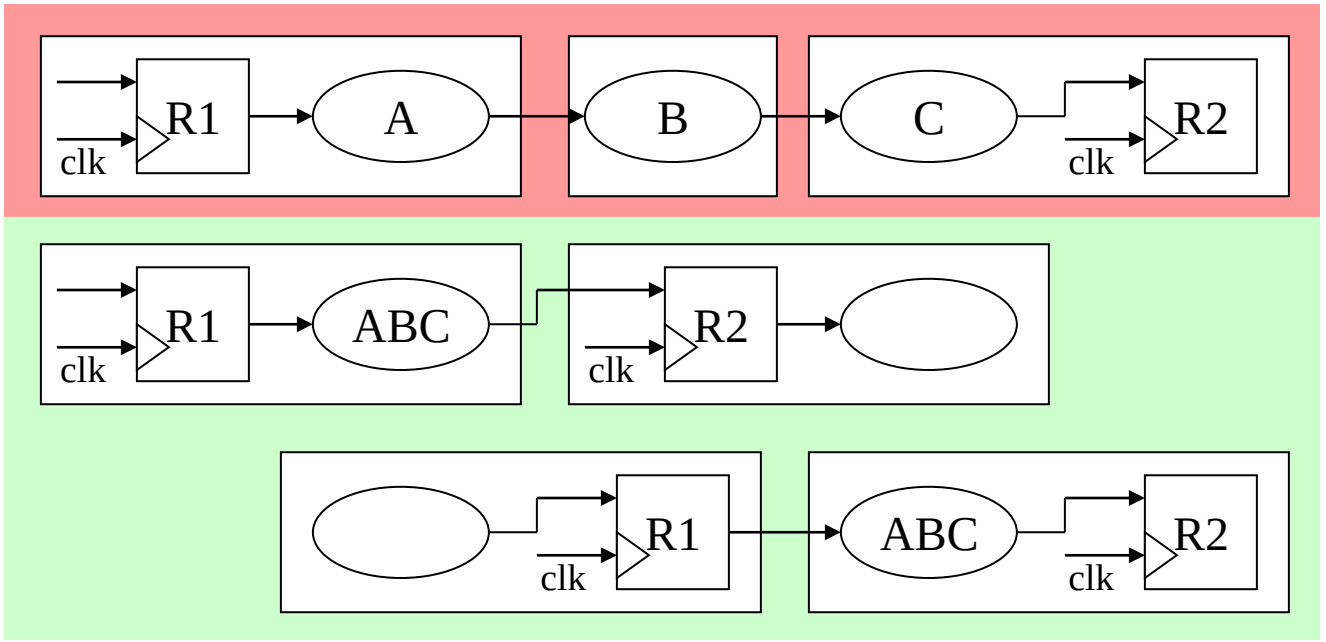


Figure Good example: All output signals are registered

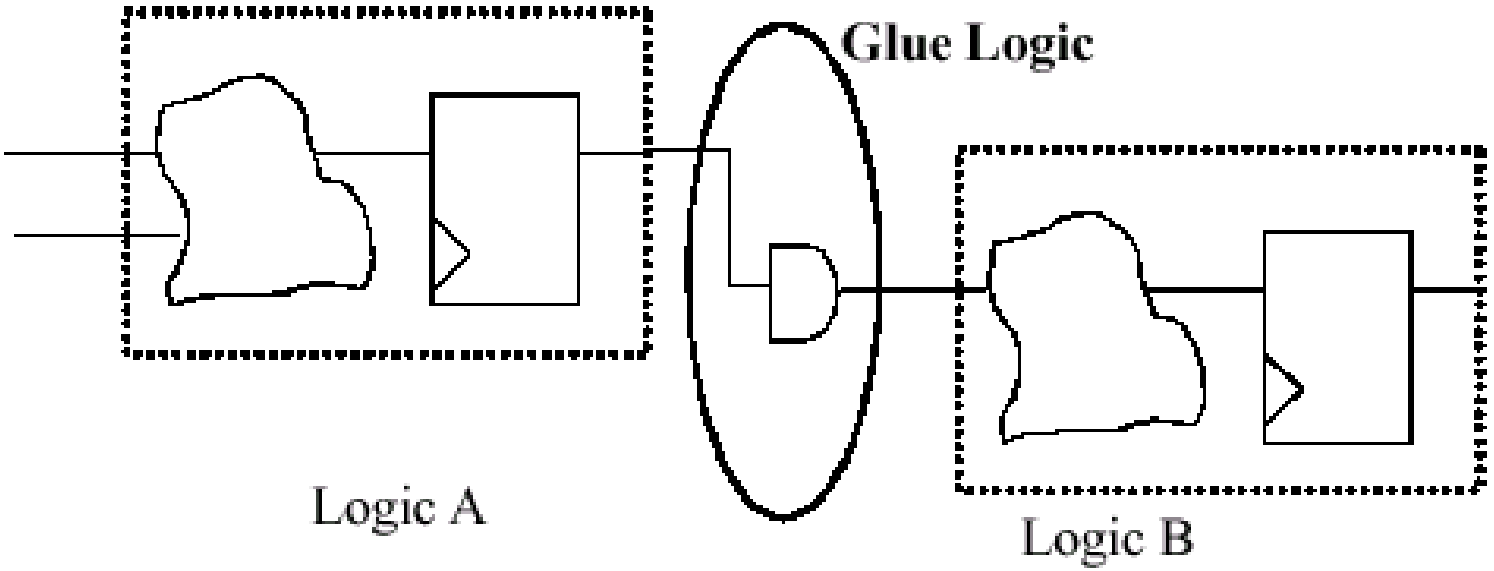
Locate Related Combinational Logic in a Single Module

- Keep related combinational logic together in the same module
  - Synthesis tools cannot optimize logic across hierarchical boundaries



Avoid Glue Logic

- No Cells except at leaf levels of hierarchy
- Any extra gates should be grouped into a sub-design



RTL Coding Cautions: Partitioning for Synthesis

- Separate combinational and sequential part
- Register at hierarchical output
- Avoid the glue logic

Outline

- Introduction to Synthesis
- Code for Synthesis
- Circuit-Level Coding Skills
- Coding Tips
- Check for Synthesizability

Check for Synthesizable (1/2)

- *SpringSoft nLint*
  - Check for correct mapping of your design
  - Not so powerful in detecting latches

- *Synopsys Design Compiler*
  - Synthesis Tool
  - The embedded *Presto Compiler* can list your flip-flops and latches in details

```
> dc_shell
dc_shell > read_verilog your_design.v
```

Check for Synthesizable (2/2)

```
Inferred memory devices in process
        in routine cache line 281 in file
                '/home/m97/gieks/cache/cache.v'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|     block6_reg      | Flip-flop |  155  |  Y  | N  | Y  | N  | N  | N  | N  |
|     block7_reg      | Flip-flop |  155  |  Y  | N  | Y  | N  | N  | N  | N  |
|     block0_reg      | Flip-flop |  155  |  Y  | N  | Y  | N  | N  | N  | N  |
|      state_reg      | Flip-flop |   2   |  Y  | N  | Y  | N  | N  | N  | N  |
|     block1_reg      | Flip-flop |  155  |  Y  | N  | Y  | N  | N  | N  | N  |
|  mem_fetching_reg   | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
|     block3_reg      | Flip-flop |  155  |  Y  | N  | Y  | N  | N  | N  | N  |
|     block5_reg      | Flip-flop |  155  |  Y  | N  | Y  | N  | N  | N  | N  |
|     block2_reg      | Flip-flop |  155  |  Y  | N  | Y  | N  | N  | N  | N  |
|     block4_reg      | Flip-flop |  155  |  Y  | N  | Y  | N  | N  | N  | N  |
===============================================================================
Presto compilation completed successfully.
Current design is now '/home/m97/gieks/cache/cache.db:cache'
Loaded 1 design.
Current design is 'cache'.
cache
design_vision>
```

Checking latches using Design Compiler