Digital System Design

Synthesizable Coding of Verilog

Lecturer: Chihhao Chao

Date: 2009.03.18

Outline

- Basic concepts of logic synthesis
- Synthesizable Verilog coding subset
- Verilog coding practices
  - Coding for readability
  - Coding for synthesis
  - Partitioning for synthesis

Outline

- Basic concepts of logic synthesis
- Synthesizable Verilog coding subset
- Verilog coding practices
  - Coding for readability
  - Coding for synthesis
  - Partitioning for synthesis

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, multiplexers, memory, and special flip-flops.
- Use *Design Compiler* to synthesize the circuit in order to meet 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 (RTL)

no timing info.

Translate (HDL Compiler)

Generic Boolean (GTECH)

Optimize + Map (Design Compiler)

timing info.

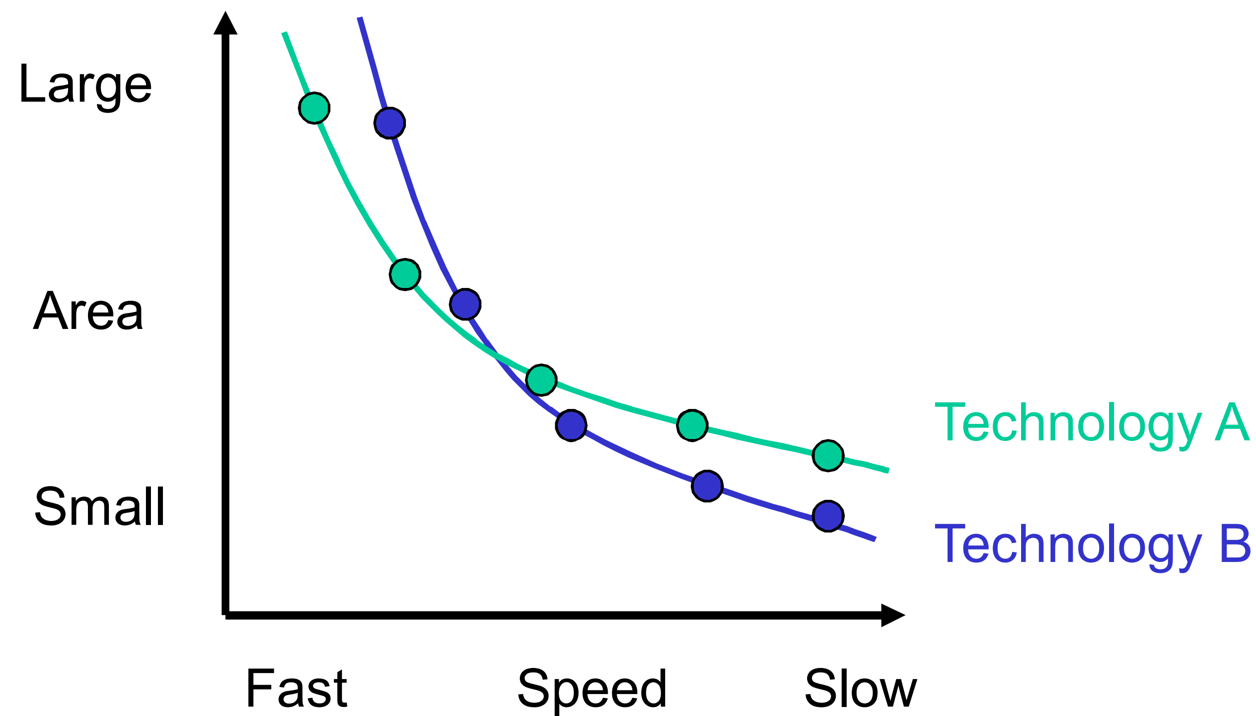
Target Technology

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

- Through synthesis, design can be translated to any technology.



Limitation on Manual Design

- **Error-Prone**: for large designs, manual conversion was prone human error, such as a small gate missed somewhere
- **Hard to Verify**: the designer could never be sure that the design constraints were going to be met until the gate-level implementation is complete and tested
- **Time-Consuming**: A significant portion of the design cycle was dominated by the time taken to convert a high-level design into gates
- **Hard to reuse**: design reuse was not possible
- **Impossible to optimize globally**: 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

- Automated Logic synthesis tools addressed these problems as follows
  - Fewer 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 can optimize the design globally.
  - Logic synthesis tools allow technology-independent design
  - Design reuse is possible (reuse the higher-level description)

Logic Synthesis

- Takes place in two stages:

1. **Translation** of Verilog (or VHDL) source to a netlist
   - Performs architectural optimizations and then creates an internal representation of the design.
   - Usually this is automatically done while design is imported to the synthesis tool.

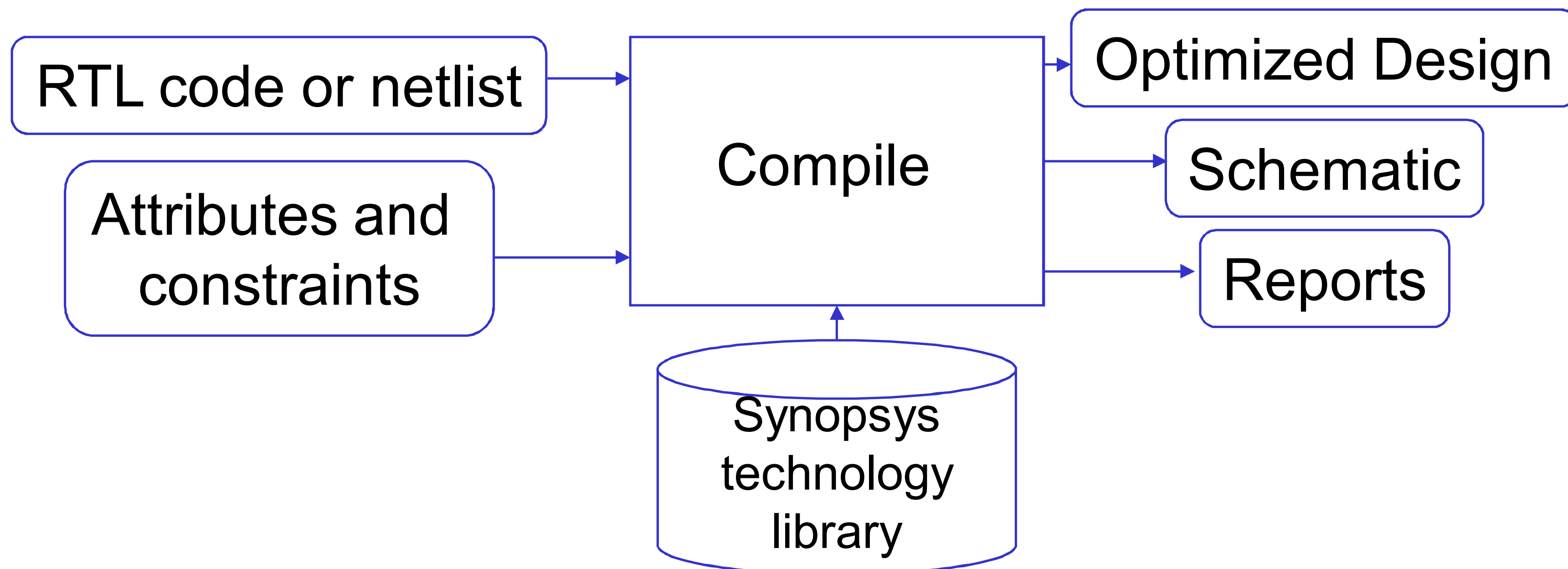
2. **Optimization** of the resulting netlist to fit constraints on speed (timing constraint) and area (area constraint)
   - Most critical part of the process
   - Logic optimization + Gate optimization

Translating Verilog into Gates

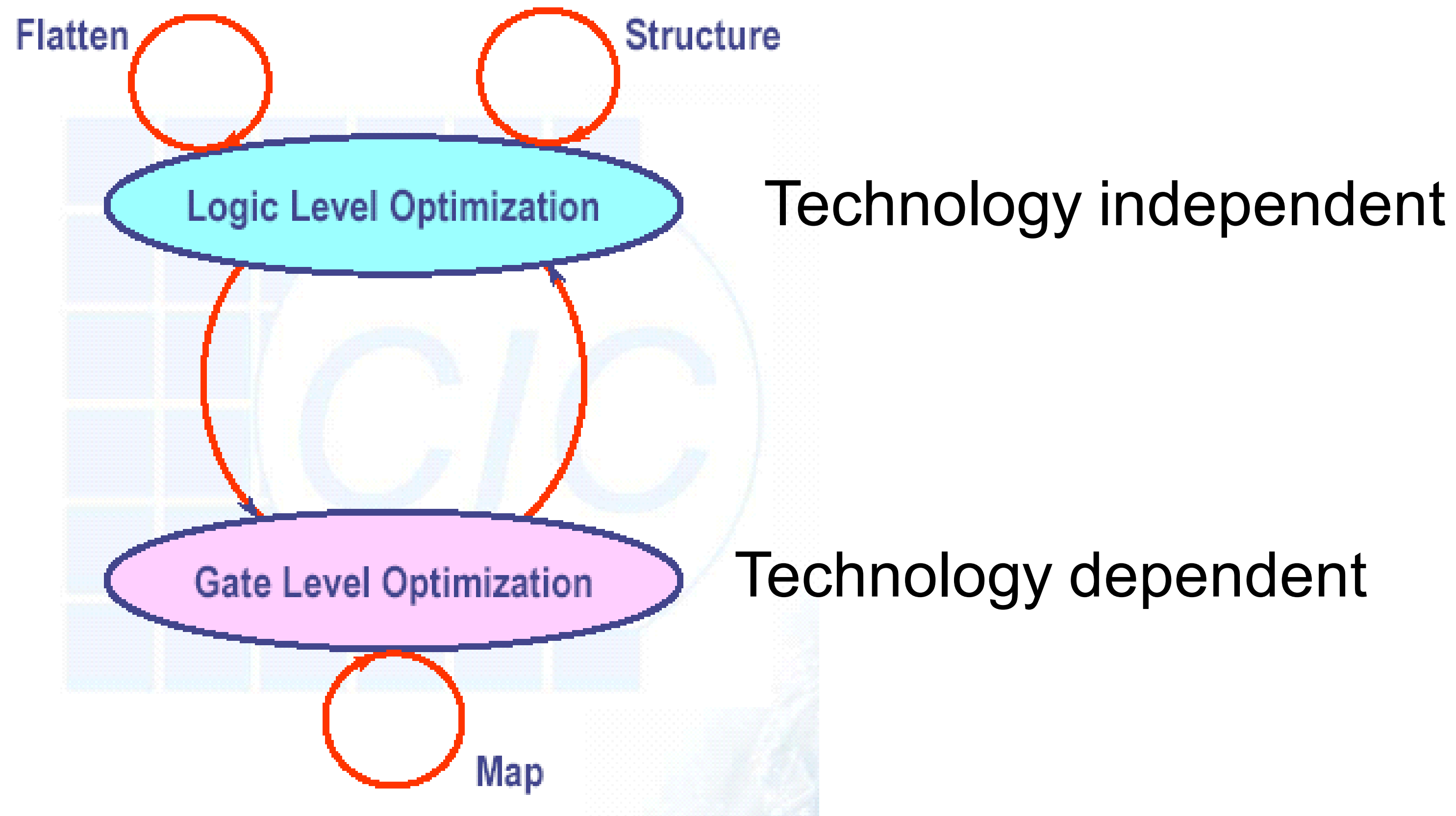
- Parts of the language easy to translate
  - Structural descriptions with primitive gates
    - Already a netlist
  - Continuous assignment
    - Expressions turn into little datapaths
- Behavioral statements
  - Synthesizable coding subset

Optimization: the “Art” of Synthesis

- **compile** command drives synthesis tool to optimize the design



Compile



Technology independent

Technology dependent

Logic Level Optimization

- Operate with Boolean representation of a circuit
- Has a *global effect* on the overall area/speed characteristic of a design
- Strategy
  - Structure
  - Flatten
  - If both are true, the design is first flattened and then structured

Structure

- Factors out common sub-expression as intermediate variable
- Useful for speed optimization as well as area optimization
- The default logic-level optimization strategy; suitable for structured circuits (e.g. adders and ALU's)
- Example:

Before Structuring

f = acd + bcd + e
g = ae' + be'
h = cde

After Structuring

f = xy + e
g = xe'
h = ye
x = a + b
y =cd

Flatten

- Remove all intermediate variable
- Result a two-level sum-of-product (SOP) form
  - Note: it doesn't mean that you will have a 2-level hardware due to library limitations
- Flatten is default OFF; Use when you have a timing goal and have don’t cares(x) in your HDL code.
- Example:

Before Flattening

f0 = at
f1 = d + t
f2 = t’e
t = b + c

After Flattening

f0 = ab + ac
f1 = b + c + d
f2 = b’c’e

Gate Level Optimization

- Select components to meet timing, design rule & area goals specified for the circuit
- Has a *local effect* on the area/speed characteristics of a design
- Strategy
  - Mapping
    - Combinational mapping
    - Sequential mapping

Combinational CKTvs. Sequential CKT Mapping

Combinational Circuit Mapping

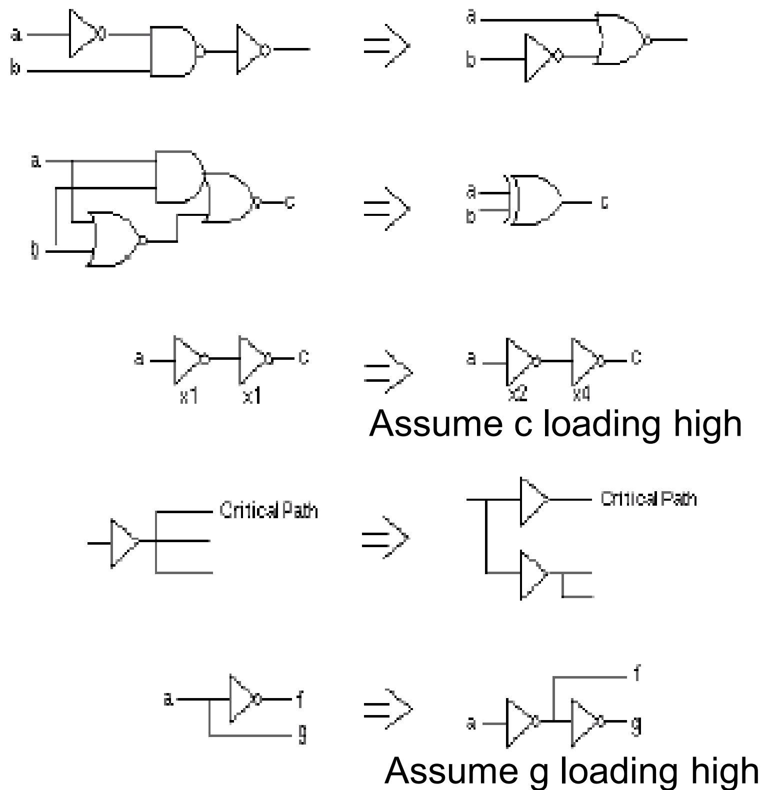
- Mapping rearranges components, combining and re-combining logic into different components
- May use different algorithms such as cloning, resizing or buffering
- Try to meet the design rule constraints and timing/area goals

Sequential Circuit Mapping

- Optimize the mapping to sequential cells from technology library
- Analyze combinational circuit 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

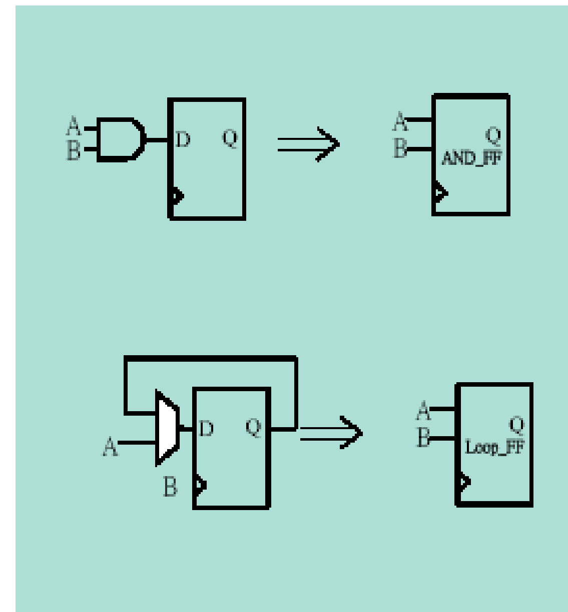
Combinational mapping



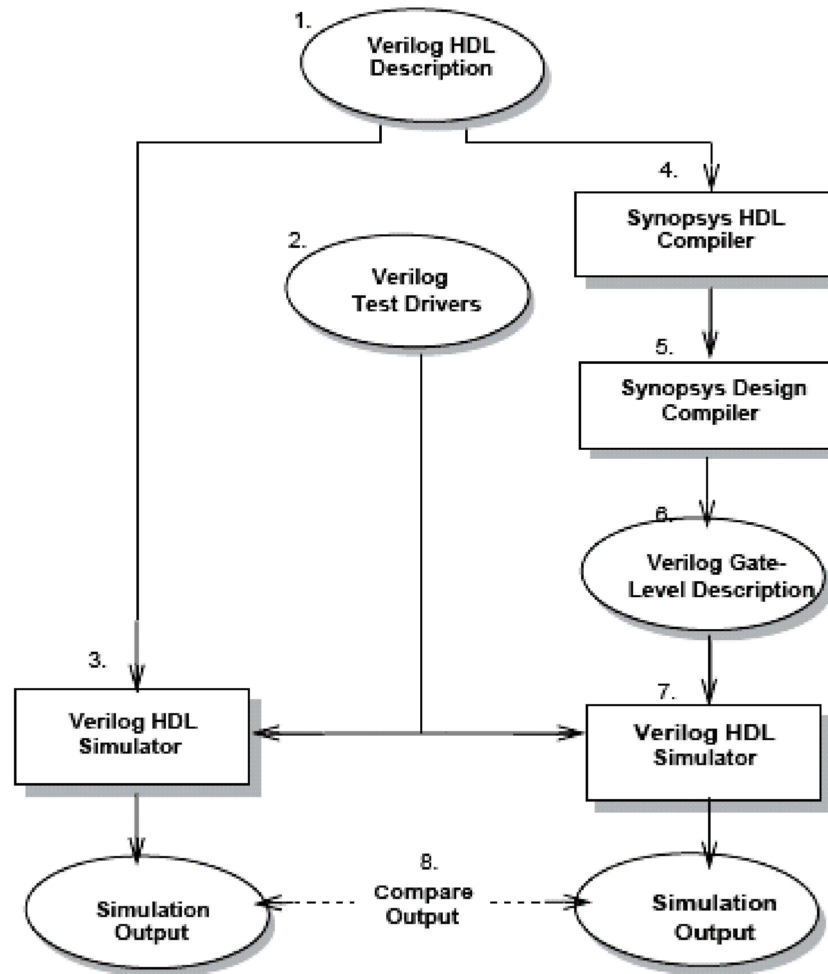
Assume c loading high

Assume g loading high

Sequential mapping



Synthesis Design Flow



Outline

- Basic concepts of logic synthesis
- Synthesizable Verilog coding subset
- Verilog coding practices
  - Coding for readability
  - Coding for synthesis
  - Partitioning for synthesis

Synthesizable Verilog Code

- Not all kinds of Verilog constructs can be synthesized.
- Only a subset of Verilog constructs can be synthesized and the code containing only this subset is synthesizable.

HDL Compiler Unsupported

- delay
- initial
- repeat
- wait
- fork … join
- event
- deassign
- force
- release
- primitive -- User defined primitive
- 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 Basis & Primitive Cell Supported

- Verilog basis
  - Parameter declarations
  - Wire, wand, wor declarations
  - Reg declarations
  - Input, output, inout declarations
  - Continuous assignments
  - Module instantiations
  - Gate instantiations
  - Always blocks
  - Task statements
  - Function definitions
  - For, while loop
- Synthesizable Verilog primitive cells
  - And, or, not, nand, nor, xor, xnor
  - Bufif0, bufif1, notif0, notif1

Verilog Operators Supported

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

Comparisons to X or Z

- A comparison to an X or Z is always evaluated to false.
  - 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)

Outline

- Basic concepts of logic synthesis
- Synthesizable Verilog coding subset
- Verilog coding practices
  - Coding for readability
  - Coding for synthesis
  - Partitioning for synthesis

Pre-RTL Preparation Checklist

- Communicate design issues with your team
  - Naming conventions, revision control, directory trees and other design organizations
- Have a specification for your design
  - Everyone should have a specification BEFORE they start coding
- Design partition
  - Follow the specification's recommendations for partition
  - Break the design into major functional blocks

RTL Coding Style

- Create a block level drawing of your design before you begin coding.
  - Draw a block diagram of the functions and sub-functions of your design.
- Always think of the poor guy who has to read your RTL code.
  - Correlate “top to bottom” in the RTL description with ”left to right” in the block diagram.
  - Comments and headers.
- Hierarchy design

General Naming Conventions(1/3)

- Use lowercase letters for all signal names, variable names, and port names.
- Use uppercase letters for names of constants and user-defined types.
- User meaningful names for signals, ports, functions, and parameters
  - Do not use *ra* for a RAM address bus, use *ram_addr*
- Use the same name or similar names for ports and signals that are connected
- Use short but descriptive names for parameters
  - Avoid excessively long names during elaboration

General Naming Conventions(2/3)

- Use *clk* for the clock signal
  - If there is more than one clock, use *clk* as the prefix for all clock signals (*clk1, clk2, clk_interface*)
- Use the same name for all clock signals that are driven from the same source
- For active low signals, end the signal name with an underscore followed by a lowercase character (e.g. _*n*).
  - Use the same lowercase character consistently to indicate active-low signals throughout the design
- Use *rst* for the reset signal. If the reset signal is active low, use *rst_n*

General Naming Conventions(3/3)

- Use *[x:0]* when describing multibit buses
  - A somewhat arbitrary suggested "standard" by the Author
- Use a distinctive suffix for state variable names.
  - <name>_cs for the current state
  - <name>_ns for the next state

Signal Naming Conventions

Convention	Use
*_r	Output of a register
*_a	Asynchronous signal
*_pn	Signal used in the *n*th phase
*_nxt	Data before being registered into a register with the same name
*_z	Tristate internal signal

File Headers

- Include informational header at the top of every source file, including scripts.
  - Legal statement: confidentiality, copyright, restrictions on reproduction
  - Filename
  - Author
  - Description of function and list of key features of the module
  - Available parameters
  - Reset scheme and clock domain
  - Date the file was created
  - Modification history including date, name of modifier, and description of the change
  - Critical timing and asynchronous interface
  - Test Structures

File Header Example: ALU.v (1/2)

```
 1 //+FHDR************************************************************************
 2 // ACCESS Laboratory, Graduate Institute of Electronics Engineering, NTU
 3 // ---------------------------------------------------------------------------
 4 // FILE NAME   : ALU.v
 5 // AUTHER      : Jih-Chiang Yeo
 6 // DATE        : 2004-03-01
 7 // VERSION     : 1.0
 8 // PURPOSE     : A ALU with Six Kind of Instructions
 9 // ---------------------------------------------------------------------------
10 // ABSTRACT
11 //
12 //     This is a design of ALU with six kind of instructions. We define the -
13 // instruction set as follows:
14 //
15 //     instruction[2:0]  operation
16 //     ==============================================
17 //     000               two's complement addition
18 //     001               two's complement subtraction
19 //     010               bit-wise NOT
20 //     011               bit-wise AND
21 //     100               bit-wise OR
22 //     101               bit-wise XOR
23 //     others            no operation
24 //
25 //     The default wordlength is eight bits, and the wordlength can be redef-
26 // ined by parameter DATA_WIDTH.
27 // ---------------------------------------------------------------------------
```

File Header Example: ALU.v (2/2)

```
27 // ------------------------------------------------------------------------
28 // REVISION HISTORY
29 //
30 //     VERSION    DATE          AUTHER            DESCRIPTION
31 //     ====================================================================
32 //     1.0        2004-02-26    Jih-Chiang Yeo    Original
33 //     1.1        2004-03-01    Jih-Chiang Yeo    Add some comments
34 // ------------------------------------------------------------------------
35 // PARAMETERS
36 //
37 //     PARA_NAME       RANGE       DESCRIPTION                 DEFAULT
38 //     ====================================================================
39 //     DATA_WIDTH      [4,32]      width of data               8
40 // ------------------------------------------------------------------------
41 // REUSE ISSUES
42 //
43 // RESET STRATEGY  : asynchronous reset
44 // CLOCK DOMAINS   : posedge trigger clock
45 // OTHER           :
46 //-FHDR********************************************************************
```

Use comments

- Use comments appropriately to explain blocks, functions, ports, signals, and variables, or groups of signals or variables.
  - Logically, near the code that they describe
  - Brief, concise, explanatory
  - Avoid “comment clutter”– obvious functionality does not need to be commented
- Describe the intent behind the section of code

More on Readability

- Use a separate line for each HDL statement
- Keep the line length to 72 characters or less
- Use indentation to improve the readability of continued code lines and nested loops
  - Use indentation of 2 spaces, avoid large indentation
  - Avoid using tabs
- Do not use Verilog reserved words for names of any elements

Ports

- Declare one port per line, with a comment following it on the same line
- Use comments to describe groups of ports
- Declare the ports in the following order, first input then output ports:
  - Inputs
    - Clocks
    - Resets
    - Enables
    - Other control signals
    - Data and address lines
  - Outputs
    - Clocks
    - Resets
    - Enables
    - Other control signals
    - data

Port Mapping

- Always use explicit mapping for ports, use named association rather than positional association

```
DW_ram_r_w_s_dff
  #((`ram_data_width+`ram_be_data_width),
    (`fifo_depth),1)
    U_int_txf_ram (
      .clk           (refclk),
      .rst_n         (txfifo_ram_reset_n),
      .cs_n          (1'b0),
      .wr_n          (txfifo_wr_en_n),
      .rd_addr       (txfifo_rd_addr),
      .wr_addr       (txfifo_wr_addr),
      .data_in       (txfifo_wr_data),
      .data_out      (txf_ram_data_out)
    );
```

Use functions

- Use functions when possible
  - Avoid repeating the same sections of code
  - If possible, generalize the function to make it reusable

```
// This function converts the incoming address to the
// corresponding relative address.

function  [`BUS_WIDTH-1:0]  convert_address;
  input input_address, offset;
  integer input_address, offset;

begin
  // ... function body goes here ...
end
endfunction // convert_address
```

Do Not Use Hard-Coded Numeric Values

Poor coding style:

```
wire    [7:0]    my_in_bus;
reg     [7:0]    my_out_bus;
```

Recommended coding style:

```
`define MY_BUS_SIZE 8
wire    [`MY_BUS_SIZE-1:0]    my_in_bus;
reg     [`MY_BUS_SIZE-1:0]    my_out_bus;
```

- Constants associate a design intention with the value.
- Constant values can be changed in one place.
- Compilers can spot typos in constants but not in hard-coded values.

Outline

- Basic concepts of logic synthesis
- Synthesizable Verilog coding subset
- Verilog coding practices
  - Coding for readability
  - Coding for synthesis
  - Partitioning for synthesis

if Statement

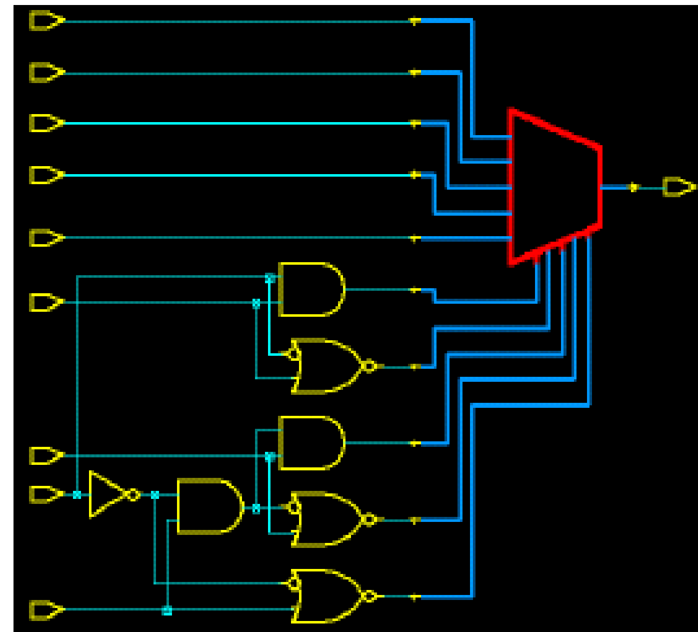
- Provide for more complex conditional actions, each condition expression controls a multiplexer legal only in function & always construct
- Syntax

```
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?

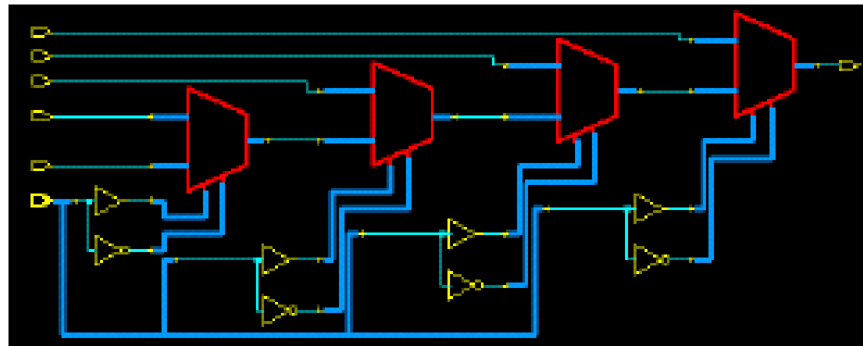
```
module mult_if(a, b, c, d, e, sel, z);
input a, b, c, d, e;
input [3:0] sel;
output z;
reg z;
always @(a or b or c or d or e or sel)
begin
z = e;
if (sel[0]) z = a;
if (sel[1]) z = b;
if (sel[2]) z = c;
if (sel[3]) z = d;
end
endmodule
```

if sel==4'b1001
  z = d;

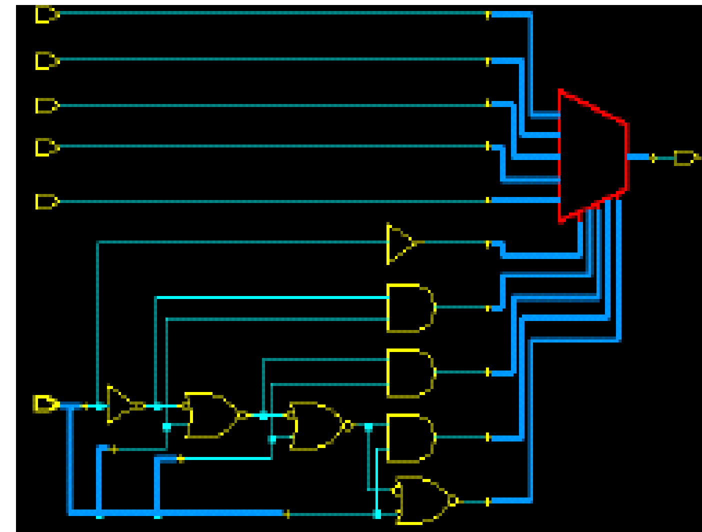
```
module single_if(a, b, c, d, e, sel, z);
input a, b, c, d, e;
input [3:0] sel;
output z;
reg z;
always @(a or b or c or d or e or sel)
begin
z = e;
if (sel[3])
  z = d;
else if (sel[2])
  z = c;
else if (sel[1])
  z = b;
else if(sel[0])
  z = a;
end
endmodule
```

if sel==4'b1001
  z = d;

if Statement



longer critical path,
smaller area



shorter critical path,
larger area

case Statement

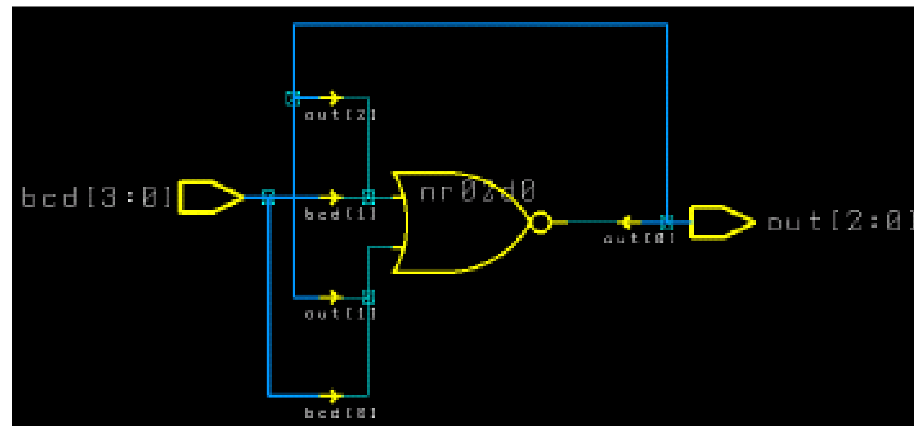
- Legal only in the function & always construct
- syntax

```
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'bxxx;
  endcase
end
```

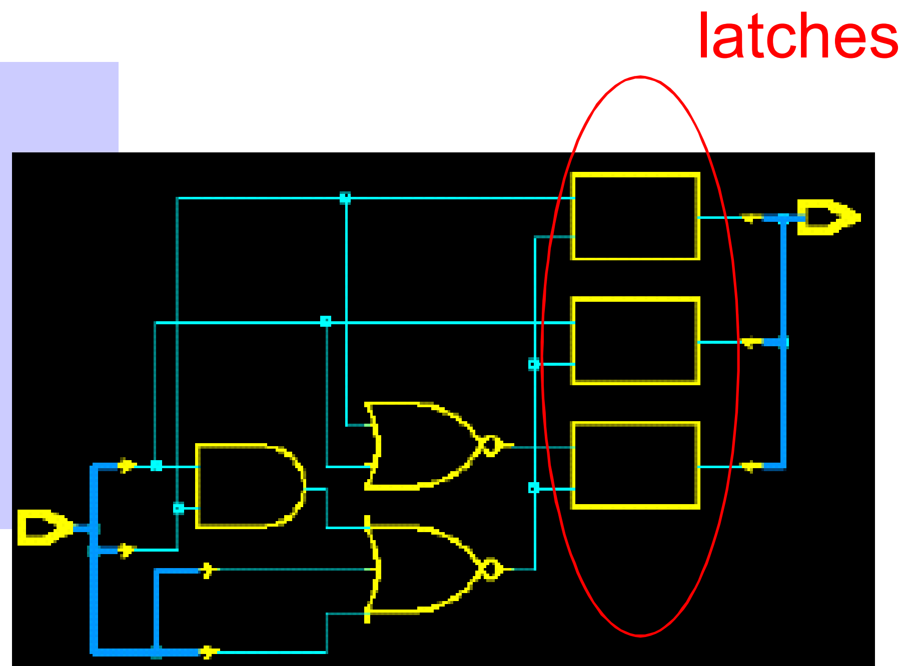


case Statement

- 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
```

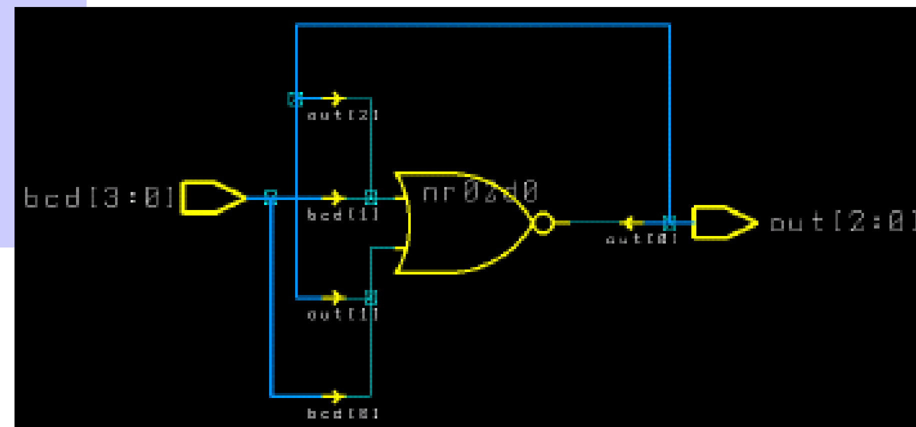
latches



case Statement

- If you do not specify all possible branches, but you know the other branches will never occur, you can use "//synopsys full_case" directive to specify full case

```
always @(bcd) begin
  case (bcd) //synopsys full_case
    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
```

case Statement

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

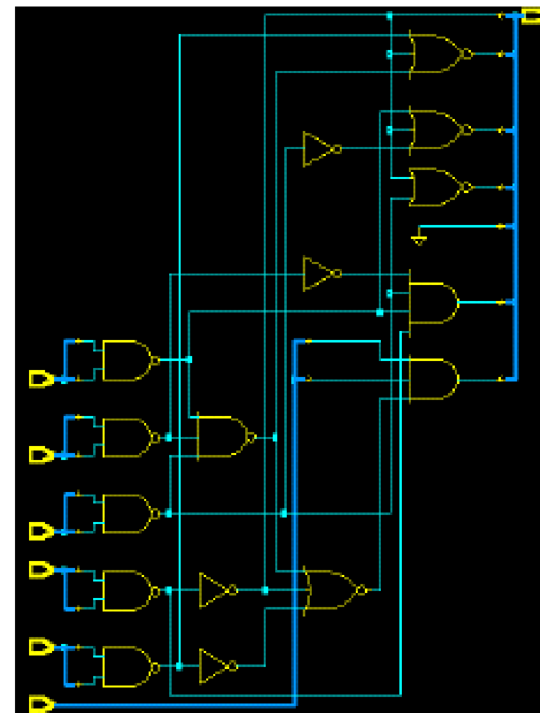
(1)
```
out = 3'b000 ; // this is called unconditional assignment
case (condition)
…
endcase
```

(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.

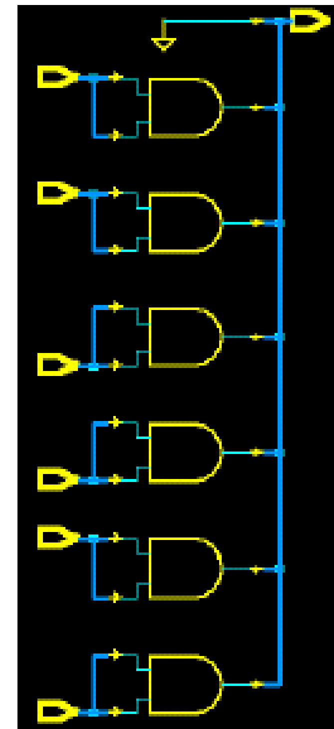
```
always @(u or v or w or x or y or z)
begin
case (2'b11)
  u:out=10'b0000000001;
  v:out=10'b0000000010;
  w:out=10'b0000000100;
  x:out=10'b0000001000;
  y:out=10'b0000010000;
  z:out=10'b0000100000;
  default:out=10'b0000000000;
endcase
end
```



case Statement

- 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
```



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.
- In synthesis, for loops loops are "unrolled", and then synthesized.

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

```
always@( a or b )
  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
```

always Block

```
always @( event-expression )
  begin
    statements
  end
```

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

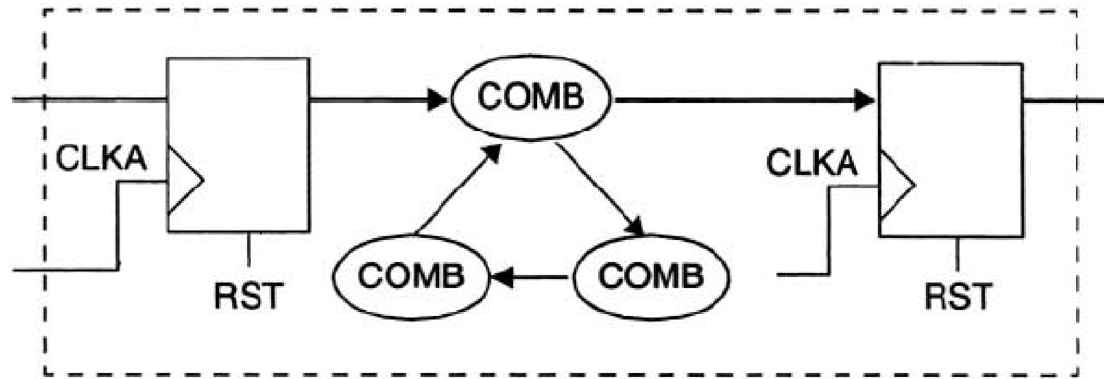
Infer Registers

- Verilog template for sequential processes
  - Use the reset signal to initialize registered signals

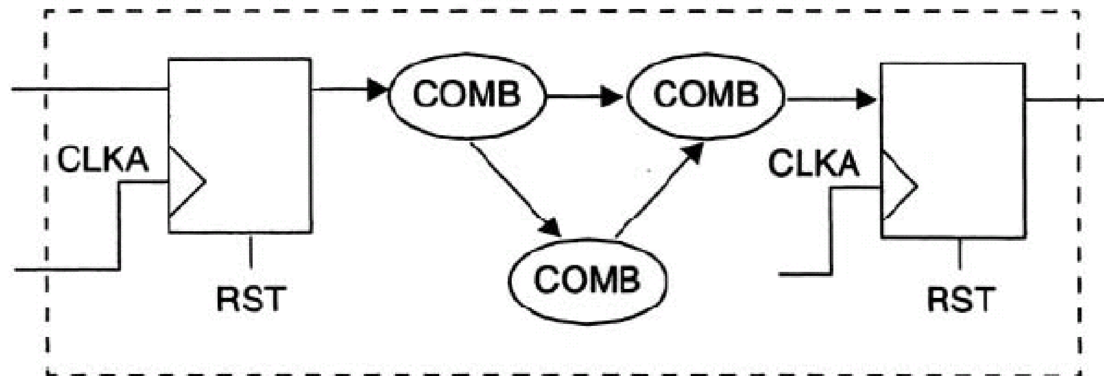
```
// process with asynchronous reset
always @(posedge clk or posedge rst_a)
  begin : ex5-20_proc
    if  (rst_a == 1´b1)
      begin
         ...
      end
    else begin
       ...
    end
  end // ex5-20_proc
```

Avoid Combinational Feedback

Bad: Combinational processes are looped



Good: Combinational processes are not looped



Sensitivity List (1/3)

- For combinational blocks, the sensitivity list must include every signal that is read by the process.
  - Signals that appear on the right side of an assign statement
  - Signals that appear in a conditional expression

```
always @(a or inc_dec)
begin : COMBINATIONAL_PROC
  if (inc_dec == 0)
    sum = a + 1;
  else
    sum = a - 1;
end // COMBINATIONAL_PROC
```

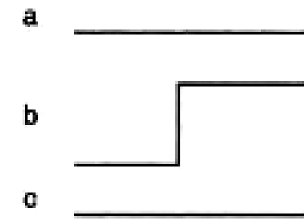
Sensitivity List (2/3)

- Include a complete sensitivity list in each of *always* blocks
  - If not, the behavior of the pre-synthesis design may differ from that of the post-synthesis netlist.

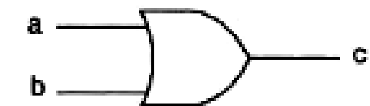
Incomplete sensitivity list

```
always @ (a)
  c <= a or b;
```

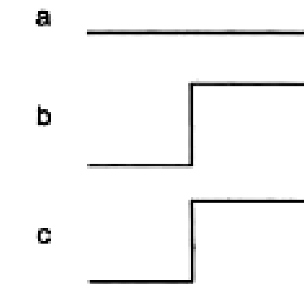
Pre-synthesis Simulation Waveform



Synthesized Netlist



Post-synthesis Simulation Waveform



Sensitivity List (3/3)

- For sequential blocks
  - The sensitive list must include the clock signal.
  - If an asynchronous reset signal is used, include reset in the sensitivity list.

```
always @(posedge clk)
begin : SEQUENTIAL_PROC
  q <= d;
end  // SEQUENTIAL_PROC
```

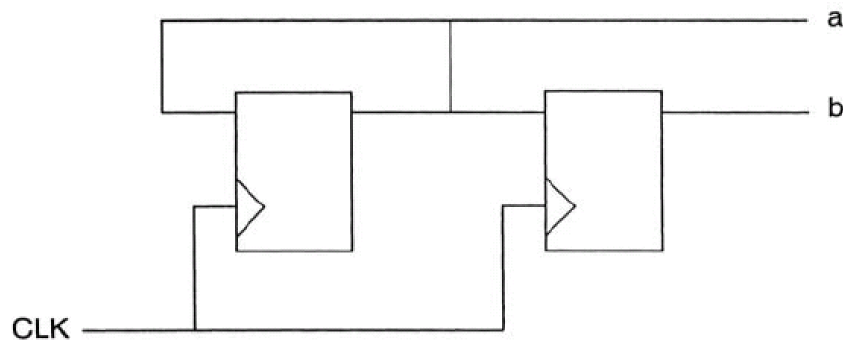
- Use only necessary signals in the sensitivity lists
  - Unnecessary signals in the sensitivity list slow down simulation

Blocking and Nonblocking Assignments(1/3)

- Two types of assignments
  - Blocking assignments execute in sequential order.
  - Nonblocking assignments execute concurrently.
- Always use nonblocking assignments in *always@ (posedge clk)* blocks.
  - Otherwise, the simulation behavior of the RTL and gate-level designs may differ.
  - Specifically, blocking assignments can lead to race conditions and unpredictable behavior in simulations.

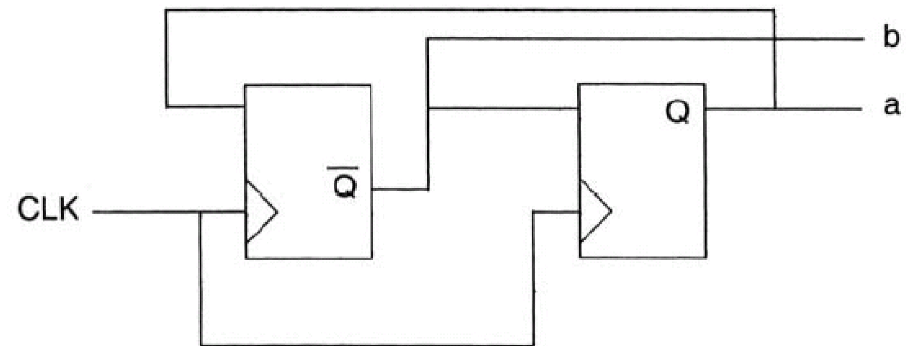
Blocking and Nonblocking Assignments(2/3)

Bad: Circuit built from blocking assignment




```
always @ (posedge clk)
begin
  b = a;
  a = b;
end
```

Good: Circuit built from nonblocking assignment




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

Blocking and Nonblocking Assignments(3/3)

- When modeling sequential logic, use nonblocking assignments.
- When modeling combinational logic with an *always* block, use blocking assignments.
- Do not mix blocking and nonblocking assignments in the same *always* block.
- Do not make assignments to the same variable from more than one *always* block.

Outline

- Basic concepts of logic synthesis
- Synthesizable Verilog coding subset
- Verilog coding practices
  - Coding for readability
  - Coding for synthesis
  - Partitioning for synthesis

Register All Outputs

- For each subblock of a hierarchical macro design, register all output signals from the subblock.
  - Makes output drive strengths and input delays

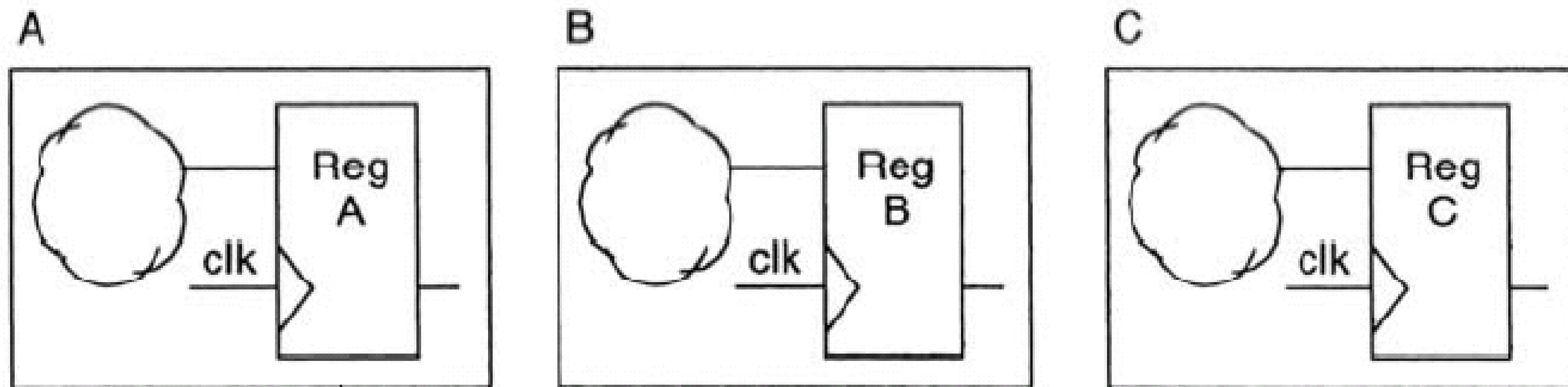
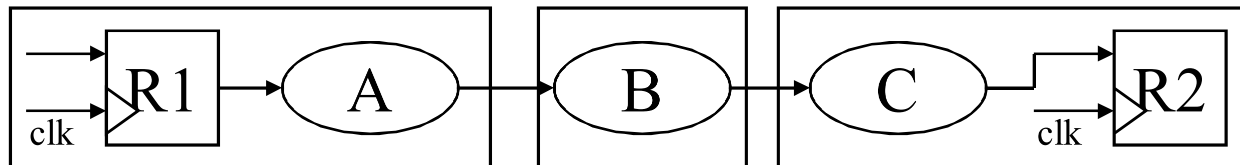


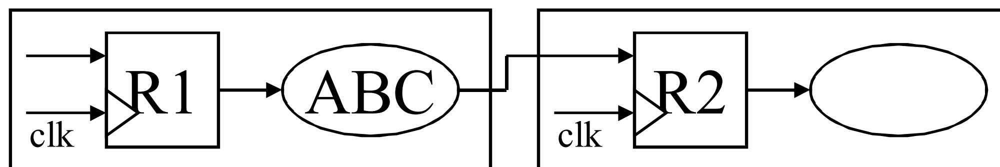
Figure Good example: All output signals are registered

Locate Related Combinational Logic in a Single Module

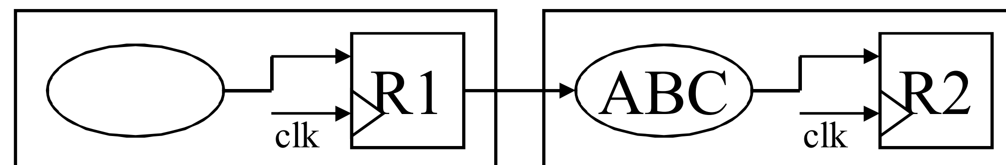
- Keep related combinational logic together in the same module



Bad



Better



Best

Separate Modules that Have Different Design Goals

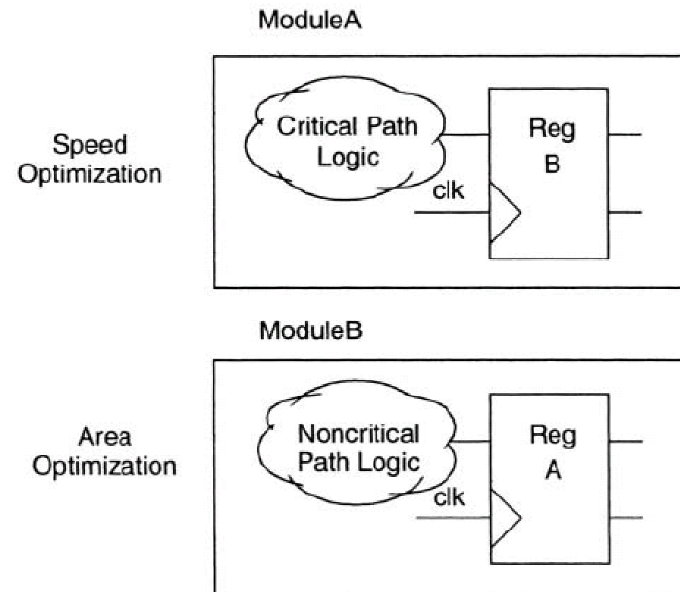


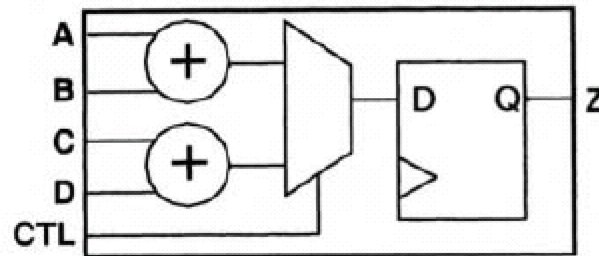
Figure Good example: Critical path logic and noncritical path logic grouped separately

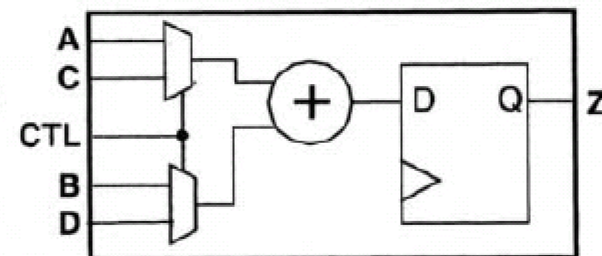
- Synthesis tools can perform speed optimization on the critical path logic, while performing area optimization on the noncritical path logic.

Arithmetic Operators: Merging Resources

- Consider the following code segment:

```
if ctl = 1'b1
   z = a + b;
else
   z = c + d;
```





- Normally two adders will be created.
- However, if only an area constraint exists, synthesis tools are likely to synthesize a single adder and to share it between the two additions.
- For synthesis tools to consider resource sharing, all relevant resources need to be within the same always block

Eliminate Glue Logic at the Top Level

- Do not instantiate gate-level logic at the top level of the macro hierarchy.
  - Merge into the subblocks so that more optimization can be performed