Enabling Productivity

RTL Coding Guidelines

DEC, 1999

SYNOPSYS CONFIDENTIAL ☐ REGISTERED

Professional Service Group
Synopsys, Inc. Asia Pacific Operations

Disclosure of information shared to other than those authorized, is strictly prohibited.

SYNOPSYS®
Professional Service Group

Introductions

- **A common set of problems facing everyone who is designing design-reuse ASICs:**
  - Time-to-market pressures demand rapid development
  - Quality of results, in performance and area, are keys to market success
  - Increasing chip complexity makes verification more difficult
  - The development team has different levels and areas of expertise
  - Design team members may have worked on similar designs in the past, but cannot reuse these designs because the design flow, tools, and guidelines have changed
- **The "Design Reuse Coding Style" offers design team members a collection of coding rules and guidelines.**
- A high quality HDL code is a prerequisite for a high quality product.

Agenda

Presentation:

HDL Coding Style Guidelines

- ❑ General HDL Code Structure
- ❑ Partitioning
- ❑ Implying Logic Structure
- ❑ Safe Coding & Avoiding Problems
- ❑ Source Code Readability
- ❑ Coding Style for Design Reuse
- ❑ Design for Testability
- ❑ Practices

HDL for Synthesis Guidelines

Enabling Productivity

Presentation:

HDL Coding Style Guidelines

- ■ **General HDL Code Structure**
- ❑ **Partitioning**
- ❑ **Implying Logic Structure**
- ❑ **Safe Coding & Avoiding Problems**
- ❑ **Source Code Readability**
- ❑ **Coding Style for Design Reuse**
- ❑ **Design for Testability**
- ❑ **Practices**

General HDL Code Structure: Checklist Items

- ❑ Standard File Headers (101)
- ❑ File Naming Conventions (102)
- ❑ Architecture Naming Conventions (103)
- ❑ Signal Naming Conventions (104)
- ❑ Use of Labels (105)
- ❑ Linking in Verilog (106)
- ❑ Clear & Meaningful Comments (107)

Standard File Headers(101)

Enabling Productivity

- ❑ **Make sure the code look familiar, no matter who writes the module.**
- ❑ **Make sure every file has a file header containing information on**
  - ☞ **file name or module function, author, creation date, abstract or summary, modification history**
  - ☞ **copyright, licensing agreement (if need)**

```
///////////////////////////////////////////////////////////
// FILE:  design.v
// AUTHOR:  Brooke Tioga
// $Id$
// ABSTRACT:  Description of the design object
// KEYWORDS:  dsp, telecom, graphics
// MODIFICATION HISTORY:
// $Log$
//  Brooke                 11/9/97  original
//      Susie              3/3/98           revised as follows...
//
// (C) Copyright 1997 Synopsys Inc.  All rights reserved
//////////////////////////////////////////////////////////
```

Used by RCS!

Headers for Major Constructs (101)

```
////////////////////////////////////////////////////////////
// FUNCTION:  double_trouble
// AUTHOR:  Ornithal Shapiro
// $Id$
// ABSTRACT:  to double throughput of filter
// MODIFICATION HISTORY:
// $Log$
//     Ornithal      12/9/97      original
//     Brooke        4/4/98       revised as follows...
//      This function performs the interpolation of data ...
////////////////////////////////////////////////////////////
```

- **Use for each function and task**
- **Use for each major section of code**

File Naming Conventions (102)

- A consistent approach to naming files greatly improves communication among designers.
- Create individual files for each module:

Convention	Object	Example
design.v	Module	arbiter.v
tb_design.v	Verilog Testbench	tb_arbiter.v

- Use module/function name as part of file name.

Architecture Naming Conventions (103)

Enabling Productivity

- **While the term "architecture" is a VHDL construct, it is used to categorize VHDL modules based on their level of abstraction.**

- **Keep the same file names for all architectures, and manage the design data with different file directories for each architecture.**

Signal Naming Conventions (104)

- **Verilog reserved words (module, endmodule, wire, reg, always, begin, end, if, else, case, endcase, ...)**
  - must use lower case (Verilog requirement)
- **Names (module names, function names, block names, wires, regs, integers, ...)**
  - use lower case
- **Names (macro …)**
  - use upper case
- **Names (Clock Signal)**
  - use clk1, clk2, or clk_interface
  - Use the same name for all clock signals that are driven from the same source.
- **Names (Reset Signal)**
  - use rst for reset signal

Signal Naming Conventions (104)

- ❑ **Names (active low signal)**
  - ☞ end the signal name with an underscore followed by a lowercase character
  - ☞ example_b, example_n
- ❑ **Names (multibit-buses)**
  - ☞ use a consistent ordering of bits
  - ☞ for VHDL (y downto x) or (x to y)
  - ☞ for Verilog [x:0] or [0:x]
- ❑ **Names (meaningful)**
  - ☞ don't use ra for a RAM address, instead, use ram_addr
- ❑ **Check with your vendor for their name restrictions**
  - ☞ (e.g. case, length)
- ❑ **Noun/verb paradigm**
  - ☞ spot_run not run_spot
  - ☞ processor_interrupt not interrupt_processor

Signal Naming Conventions (104)

- For net names, use the same name throughout the hierarchy
- Consider dc_shell commands when choosing names:
  - ☞ `set_input_delay 7.0 find(pin, "xi_pci*")`

Signal Naming Conventions (104)

- Use naming conventions to indicate type of signal:
  - input, output, register output, etc.
- Examples:

`clk_*`	Clock signal
`rst`	Reset signal
`*_n`	Negative logic (active low)
`*_r`	Output of a register
`*_a`	Asynchronous signal
`*_#n`	Signal used in the "n" phase
`*_nxt`	Data before being registered
`*_z`	Three-state internal signal
`xi_*`	Primary chip input
`xo_*`	Primary chip output
`xz_*`	Primary chip three state
`xb_*`	Primary chip bidirectional

Use of Labels (105)

Enabling Productivity

- Labels improve readability & debugging
- If labels are not specified, arbitrary labels are generated internal to simulation/synthesis tools
- Labeled ***always@*** facilitate repartitioning with the ***group*** command.

```
always@(posedge CLK)
    begin: CHT2BIT
        if (RESET == 1'b1)
              QOUT <= 2'b00;
        else
              QOUT <= QOUT + 1'b1;
    end
```

Use Labels at ***begin***

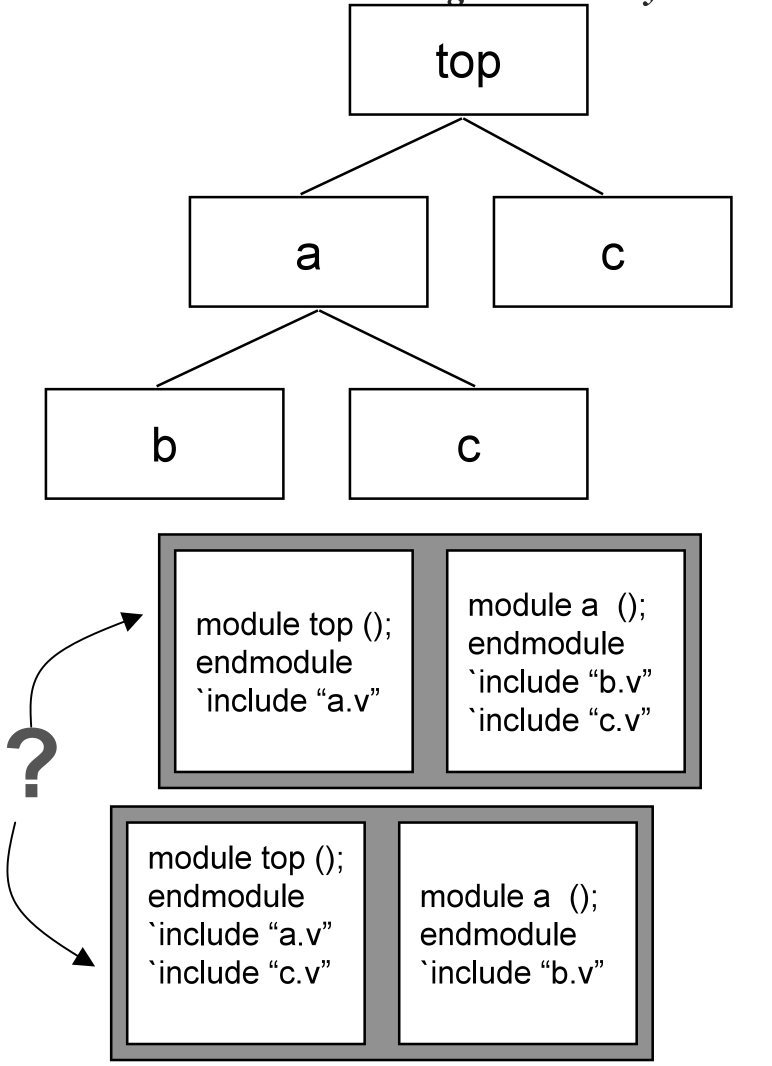
```
dc_shell_script:
group -hdl_block CHT2BIT
```

Use Labels on always@, & function constructs

Linking Modules(106): Avoid Modules linked with `include

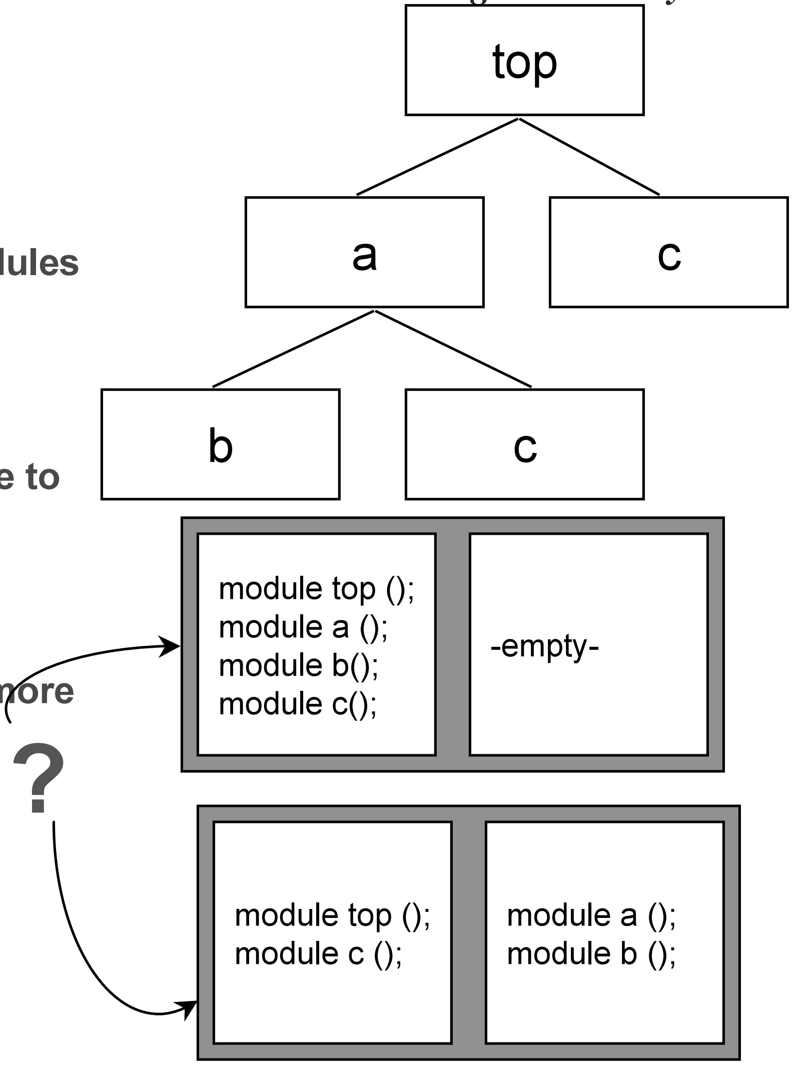
Enabling Productivity

- ❑ **Avoid modules linked with ‘include’**
- ❑ **Reasons:**
  - ☞ Locating the file
    - ⇨ The included file has to reside in the same directory from which the tool(simulation or synthesis) is invoked, or a path to the file must be specified in the source code.
  - ☞ Compilation
    - ⇨ The included file may complicate the design partitioning and may result in greater effort when developing a bottoms-up compile strategy.


top
a
c
b
c
module top (); endmodule `include “a.v”
module a (); endmodule `include “b.v” `include “c.v”
?
module top (); endmodule `include “a.v” `include “c.v”
module a (); endmodule `include “b.v”

Linking Modules (106): Avoid Multiple Modules in a Single File

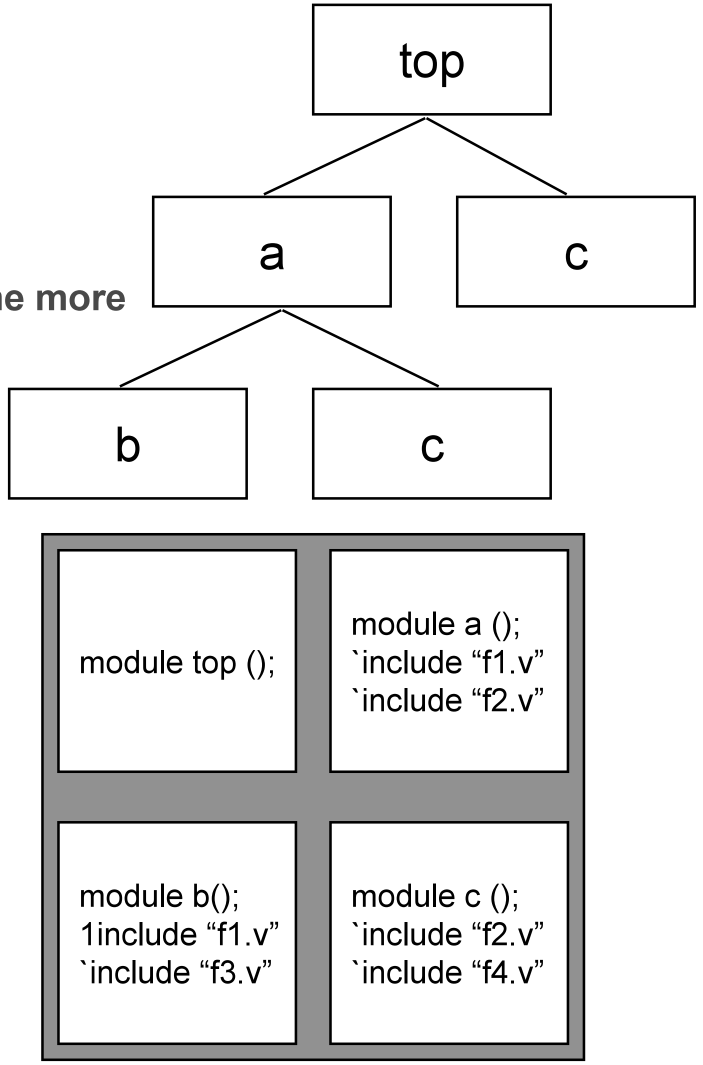
- **Avoid multiple modules in single file.**
- **Reasons:**
  - Locating the file
    - Determining which files contain which modules cannot be inferred using the file name conventions.
  - Compilation
    - Performing incremental compiles in DC due to small changes become more complex and timing consuming.
  - Revision Control (RCS)
    - Revision control and bug tracing become more complicated.



Linking Functions (106): Avoid Multiple Functions in a Single File

Enabling Productivity

- ❑ **Avoid: Multiple functions in single file**
- ❑ **Reasons:**
  - ☞ Revision Control (RCS)
    - ⇨ **Revision control and bug tracing become more complicated.**



Clear Meaningful Comments (107)

- **Improve readability, maintainability, ability to reuse, easy review, traceability**
- **Typical engineers response ...**
  - ☞ ***"I don't have the time to comment now"***
  - ☞ ***When complete do you go back & comment or do you move on to the next assignment ?***

HDL for Synthesis Guidelines

Enabling Productivity

Presentation:

HDL for Synthesis Guidelines

- ❑ **General HDL Code Structure**
- ■ **Partitioning**
- ❑ **Implying Logic Structure**
- ❑ **Safe Coding & Avoiding Problems**
- ❑ **Source Code Readability**
- ❑ **Coding Style for Design Reuse**
- ❑ **Design for Testability**
- ❑ **Practices**

Partitioning Effects

- Partitioning is not just a functional issue. It can significantly affect the following process:
  - Synthesis Quality-of-Result (QOR)
  - Synthesis constraints
  - Synthesis scripts
  - Synthesis compile time
  - Static timing analysis
  - Floorplanning
  - Layout

Partitioning Aims

- ❑ **Physical Implementation Issues**
  - ☞ **Keep related combinational logic together**
  - ☞ **Combine shareable resource**
  - ☞ **Merge user-defined resources and driven logic**
  - ☞ **Partition based on design goals**
- ❑ **Partitioning to Speed Up the Compile Process**
  - ☞ **Eliminate glue logic**
  - ☞ **Maintain a reasonable gate size**
  - ☞ **Maintain a reasonable number of levels**
  - ☞ **Isolate point-to-point exceptions in the same module**

Partition Aims

- **Partitioning to Simplify Scripts and Constraint Files**
  - ☞ **Register all outputs**
  - ☞ **At chip-level create core logic, pad ring, and test hierarchy**
- **Commands that Manipulate Hierarchy**
  - ☞ **If artificial and suboptimal barriers exist in critical combinational logic path, you can rearrange the hierarchy to eliminate the suboptimal interface.**
    - ⇨ **DC command: group**
    - ⇨ **DC command: ungroup**

Partitioning: Checklist Items

Enabling Productivity

❑ **Physical Implementation Issues**

- ☞ No snake paths in critical paths (201)
- ☞ Combine sharable resources (202)
- ☞ Merge User-Defined Resources with the logic they drive (203)
- ☞ Separate logic with different synthesis goals
  - ⇨ area vs. speed sensitive (204)
  - ⇨ random vs. structured (205)
- ☞ Separate Clock Generation Module (206)
- ☞ Separate Asynchronous Logic (207)
- ☞ Separate Finite State Machines (208)

Partitioning: Checklist Items

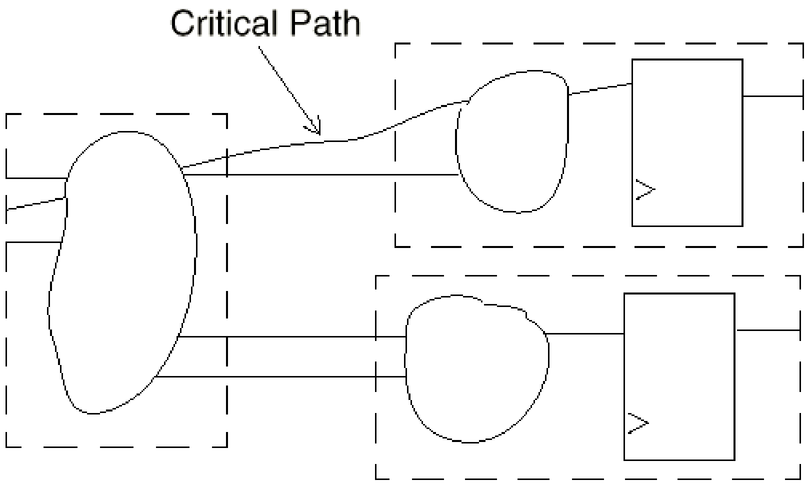
Enabling Productivity

- ❑ **Partitioning to Speed Up the Compile Process**
  - ☞ Eliminate glue logic (209)
  - ☞ Reasonable design size (210)
  - ☞ Reasonable hierarchy (211)
  - ☞ Isolate Point-to-Point Exceptions (212)
- ❑ **Partitioning to Simplify Scripts and Constraints Files**
  - ☞ Register the outputs (213)
  - ☞ Chip-Level Partitioning (214)
- ❑ **Commands that Manipulate Hierarchy**
  - ☞ Ungroup
  - ☞ Group

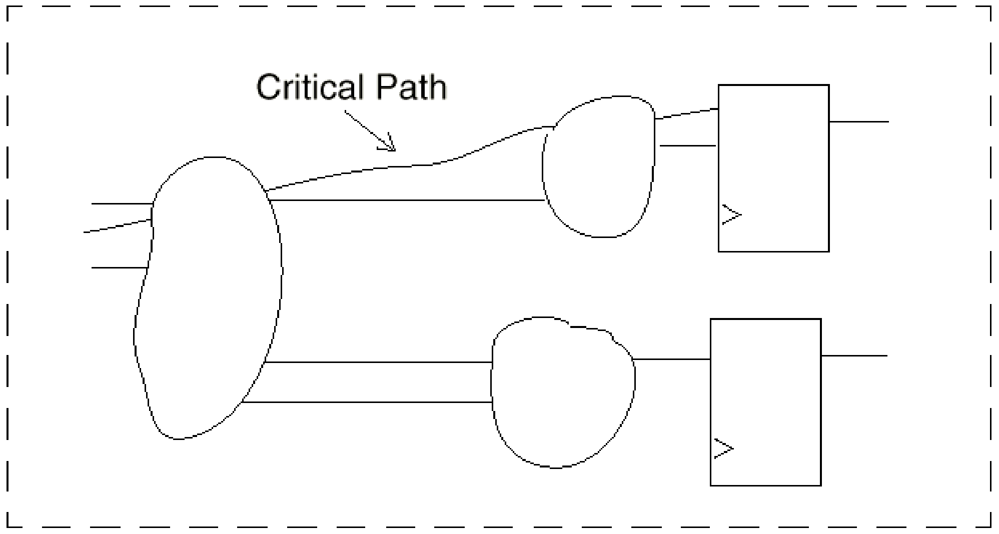
Avoid Snake Paths *(in critical path)* *(201)*

Enabling Productivity

- ❑ **Design Compiler cannot move logic across hierarchical boundaries.**
- ❑ **Dividing related combinational logic into separate modules introduces artificial barriers restrict logic optimization**



Poor Partitioning of Related Logic

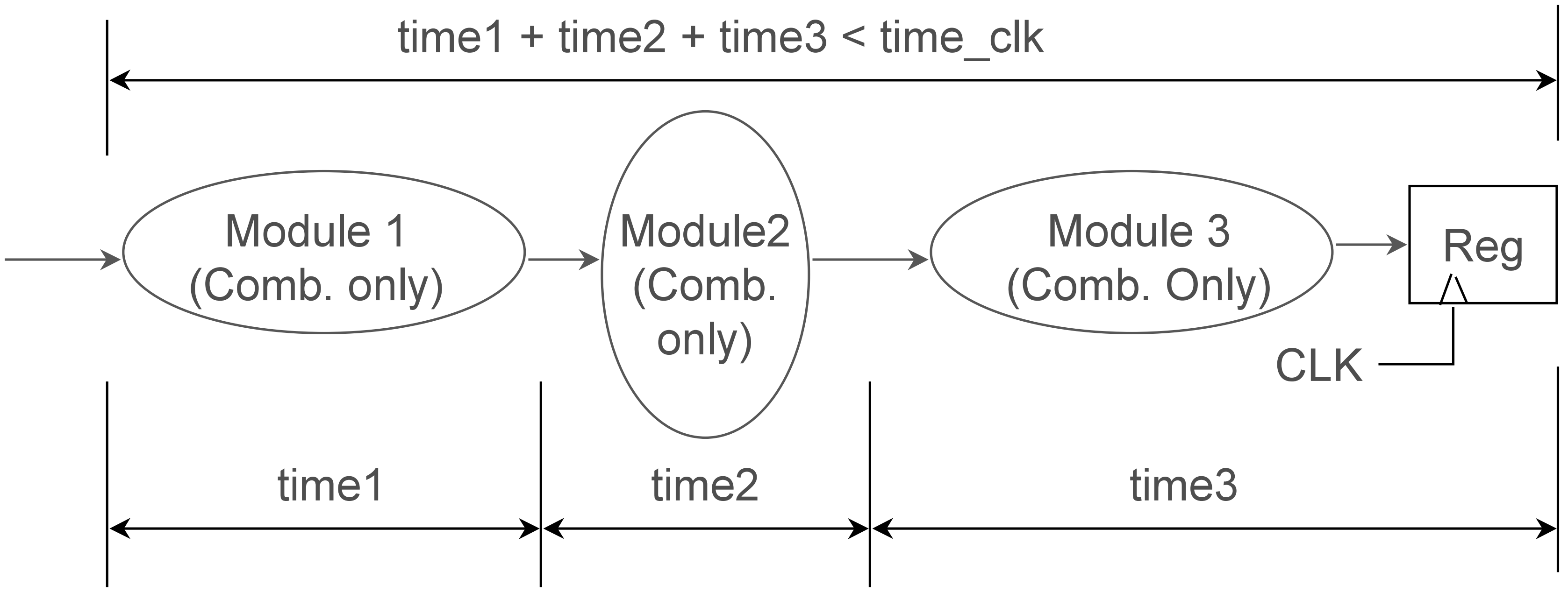


Keep Related Logic in the Same Module

Avoid Snake Paths *(in critical path)* *(201)*

Enabling Productivity

- ❑ **Snake Path - combinational logic path distributed over multiple modules**
- ❑ **DC does not allow cross boundary optimization & requires time budgeting**

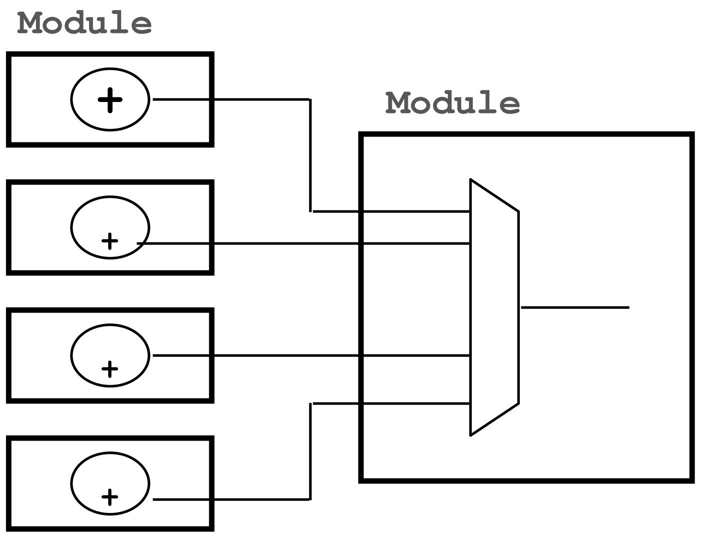


Combine Sharable Resources (202)

Enabling Productivity

- Resources (e.g. adders) can be shared if they are never used ***at the same time.***
- For HDL Compiler to determine this, the resources ***MUST*** be in the same module and always@ block

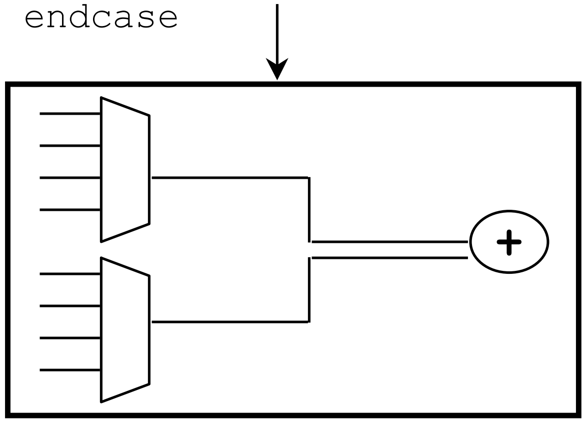
BAD



(Cannot Share
Across Modules)

GOOD

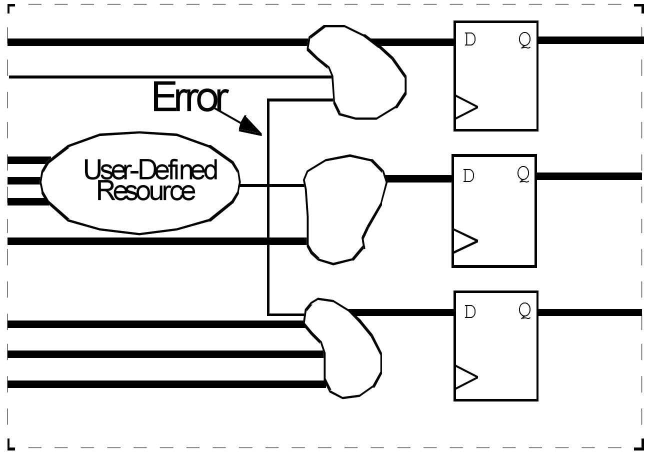
```
case (sel)
  2'b00 :Y = a + b;
  2'b01 :Y = c + d;
  2'b10 :Y = e + f;
  2'b11 :Y = g + h;
  default:Y = 8'bx;
endcase
```

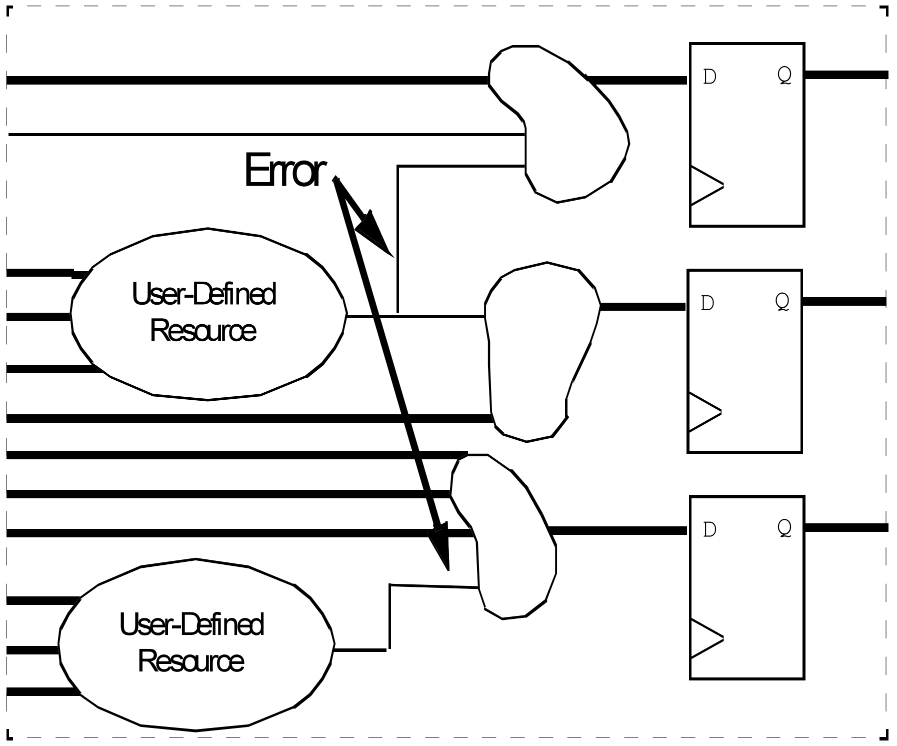


Merge User-Defined Resources with the Logic They Drive(203)

Enabling Productivity

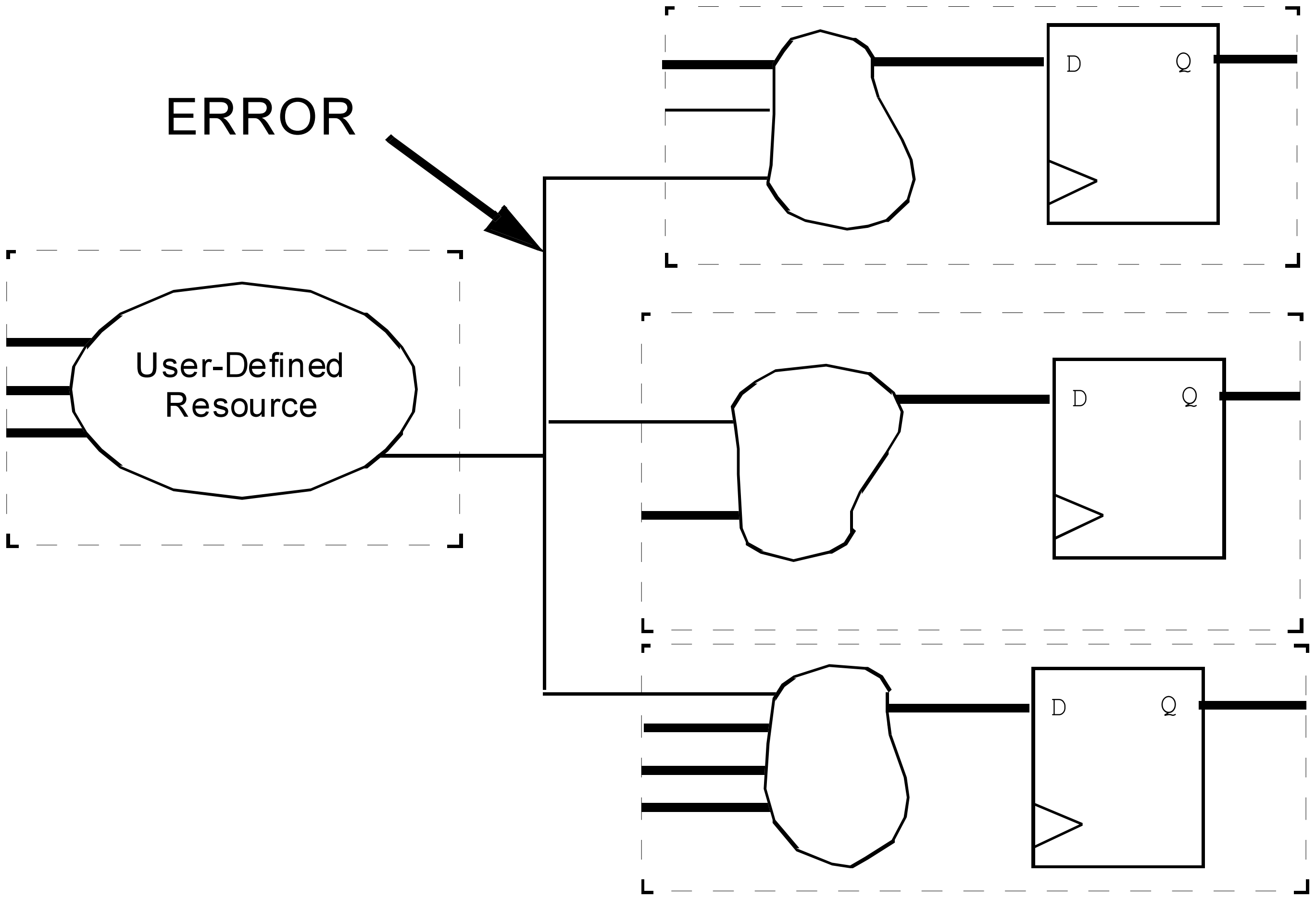
- A user defined resource is any logic that drives a large fanout .
  - (e.g. mux-select for 100 muxes)
- You may want to replicate user defined resources to balance the load.
  - (e.g. 10 mux-selects to drive 10 muxes)





Merge User-Defined Resources with the Logic They Drive(203)

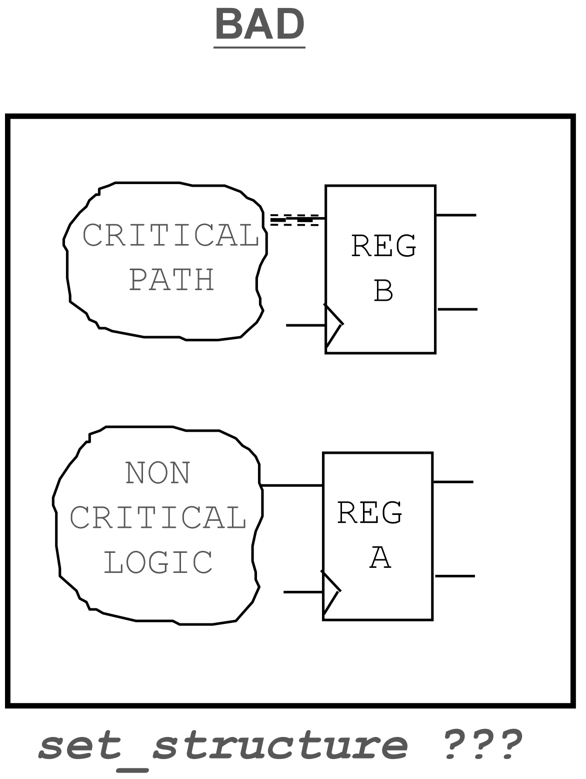
- ❑ **A poor partitioning might bring more synthesis and timing analysis problems.**

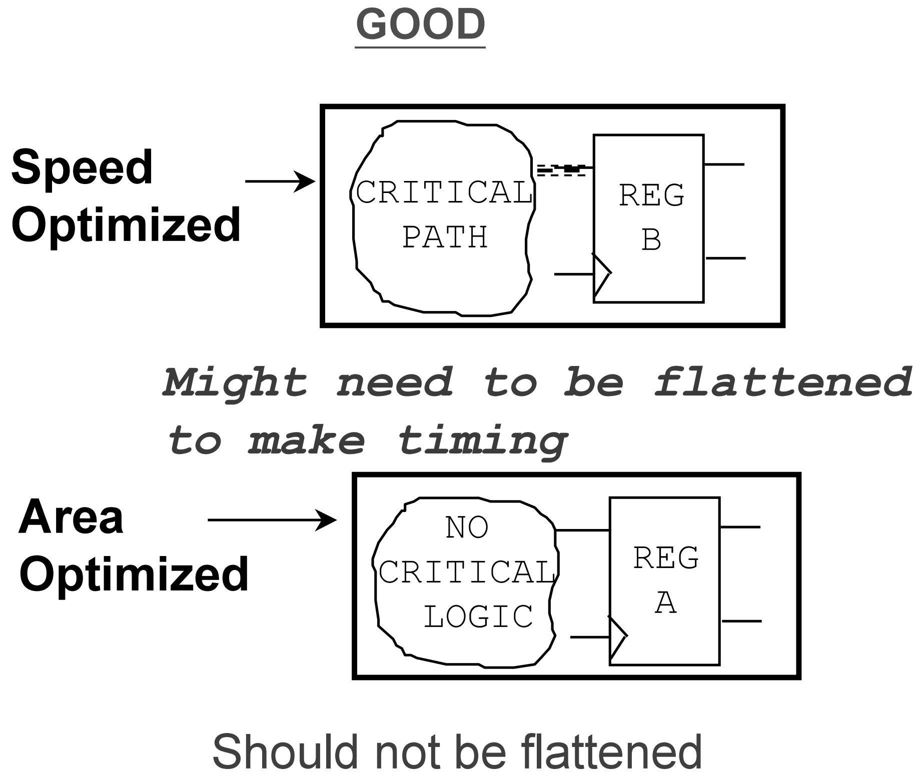


Separate Area & Speed Logic (204)

Enabling Productivity

- Area and Speed Critical Logic are best optimized with different compile strategies
- Separate the logic so you can apply these strategies individually

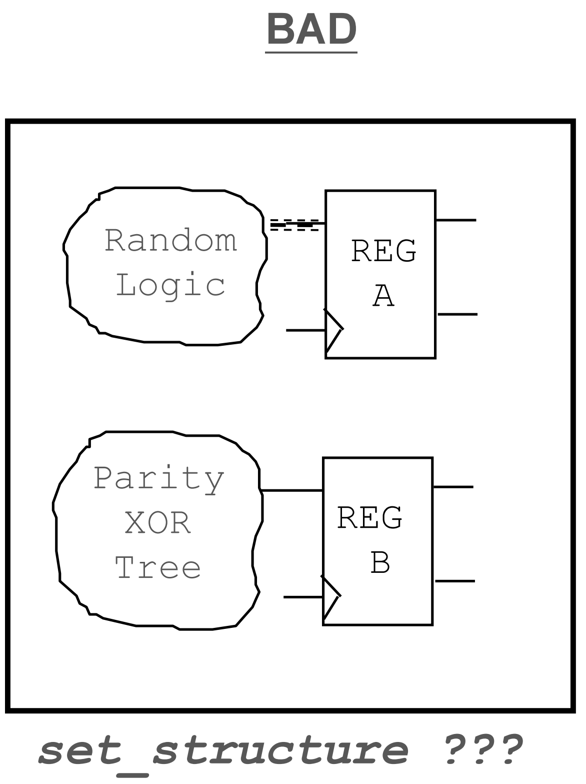


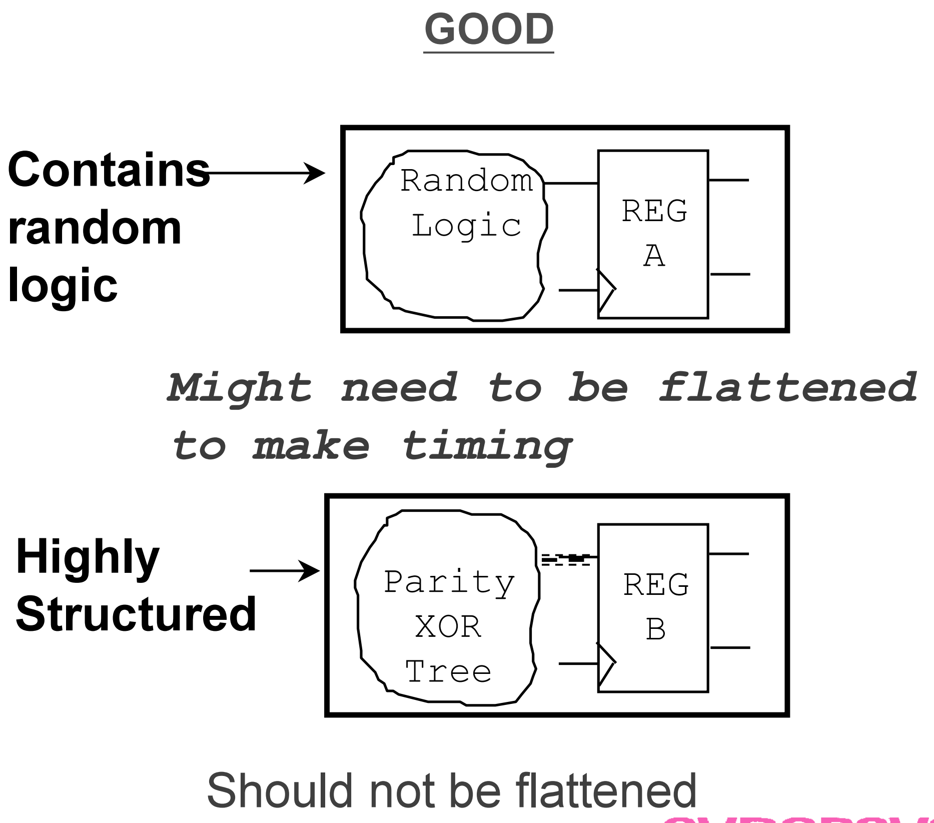


Separate Random & Structured Logic (205)

Enabling Productivity

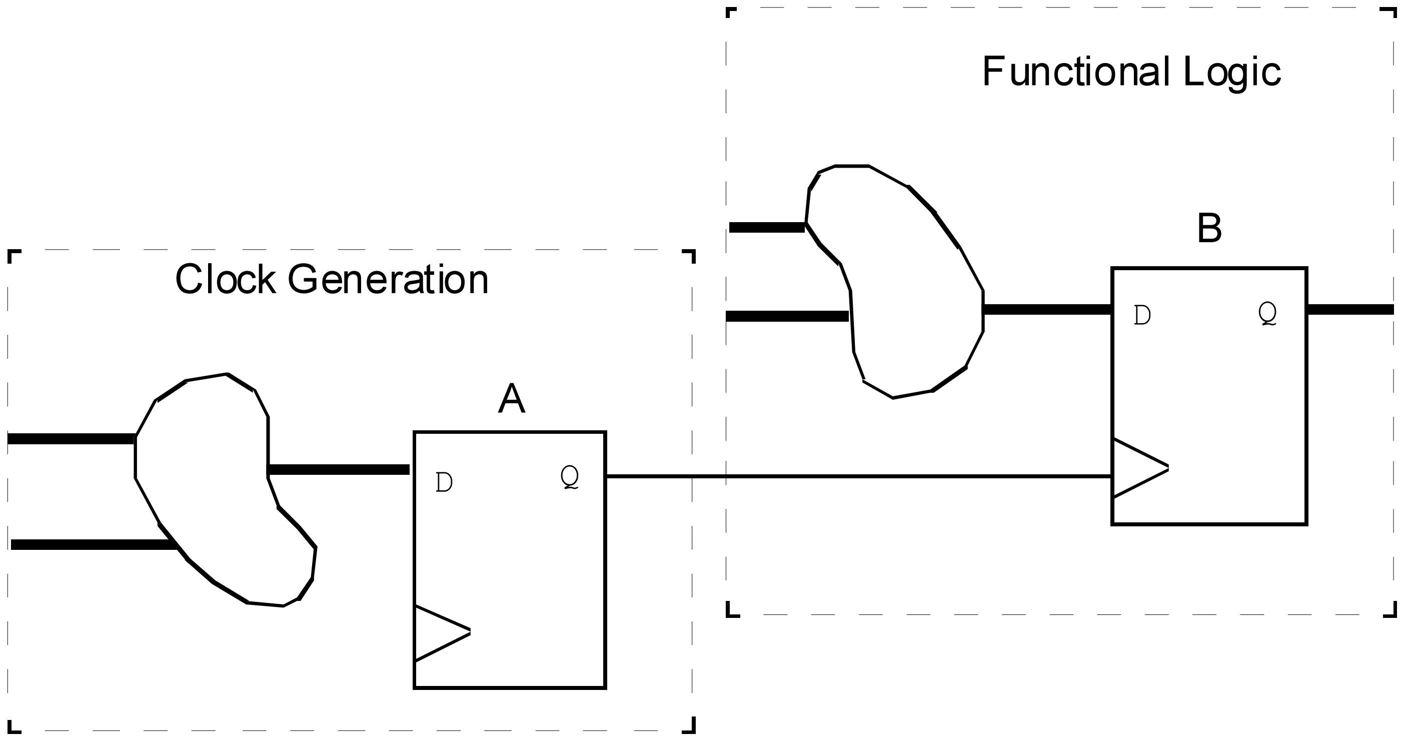
- ❑ Random and Structured Logic are best optimized with different compile strategies.
- ❑ Separate logic so you can apply these strategies individually.





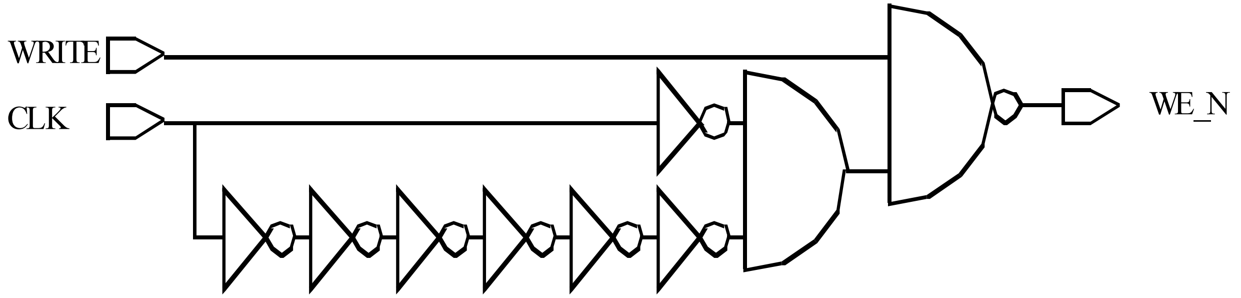
Separate Clock Generation Module (206)

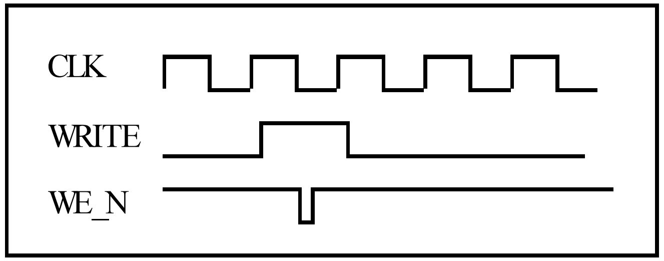
- Clock generation logic is typically handcrafted and often requires special timing analysis.
- It is often recommended that clock generation logic be put into its own module.



Separate Asynchronous Logic (207)

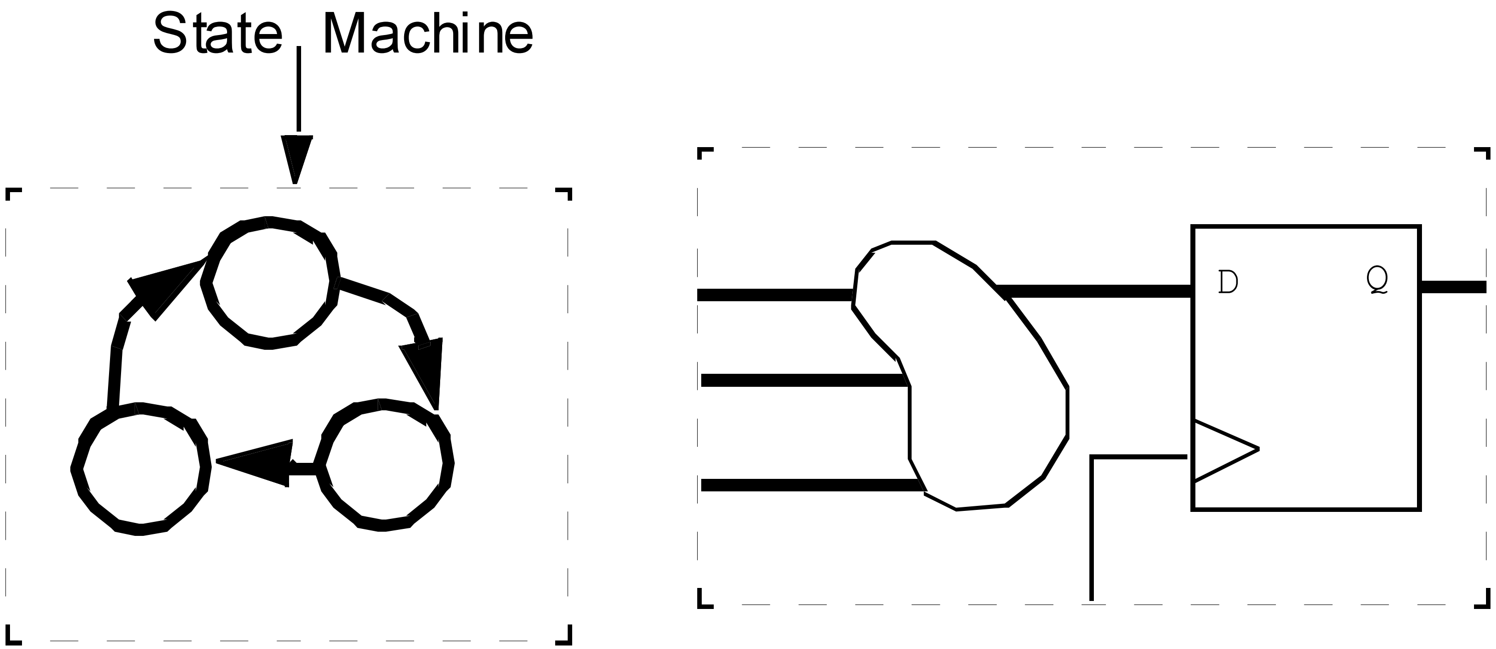
- Asynchronous logic is sometimes technology-dependent, and typically requires gate-level instantiation and a special synthesis methodology.
- Asynchronous logic typically requires special test considerations and verification strategies.





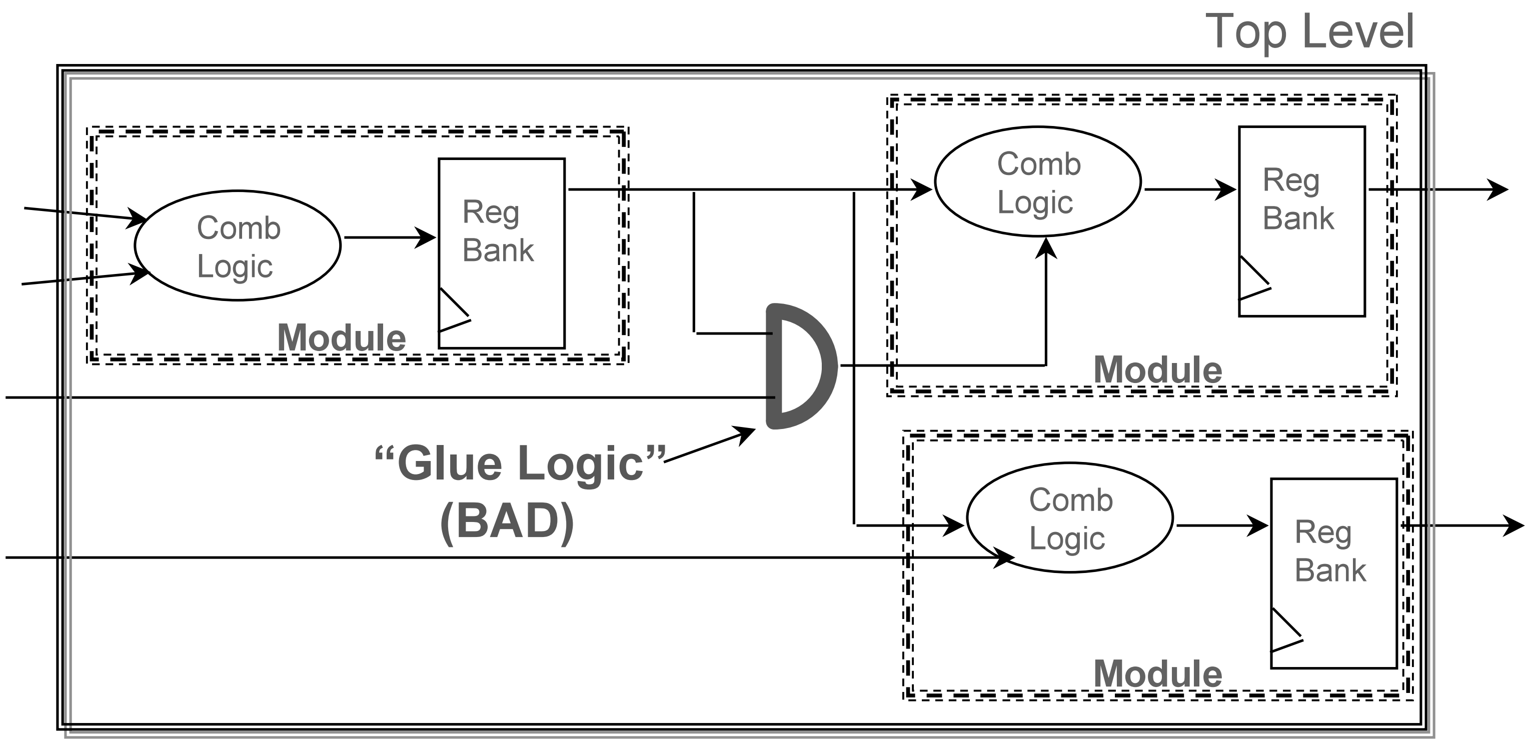
Separate Finite State Machines (208)

- A state machine may benefit from the state machine compiler or from a flattening optimization strategy.
- Modules that contain only state machines simplify the state extraction and optimization process.



Eliminate Glue Logic (209)

- **Design should only contain gates at the leaf level of the hierarchy tree**
  - **Reduces CPU time to compile small amounts of logic for glue.**
  - **Synthesis compile scripts are simplified when glue logic is removed.**



Reasonable Design Size (210)

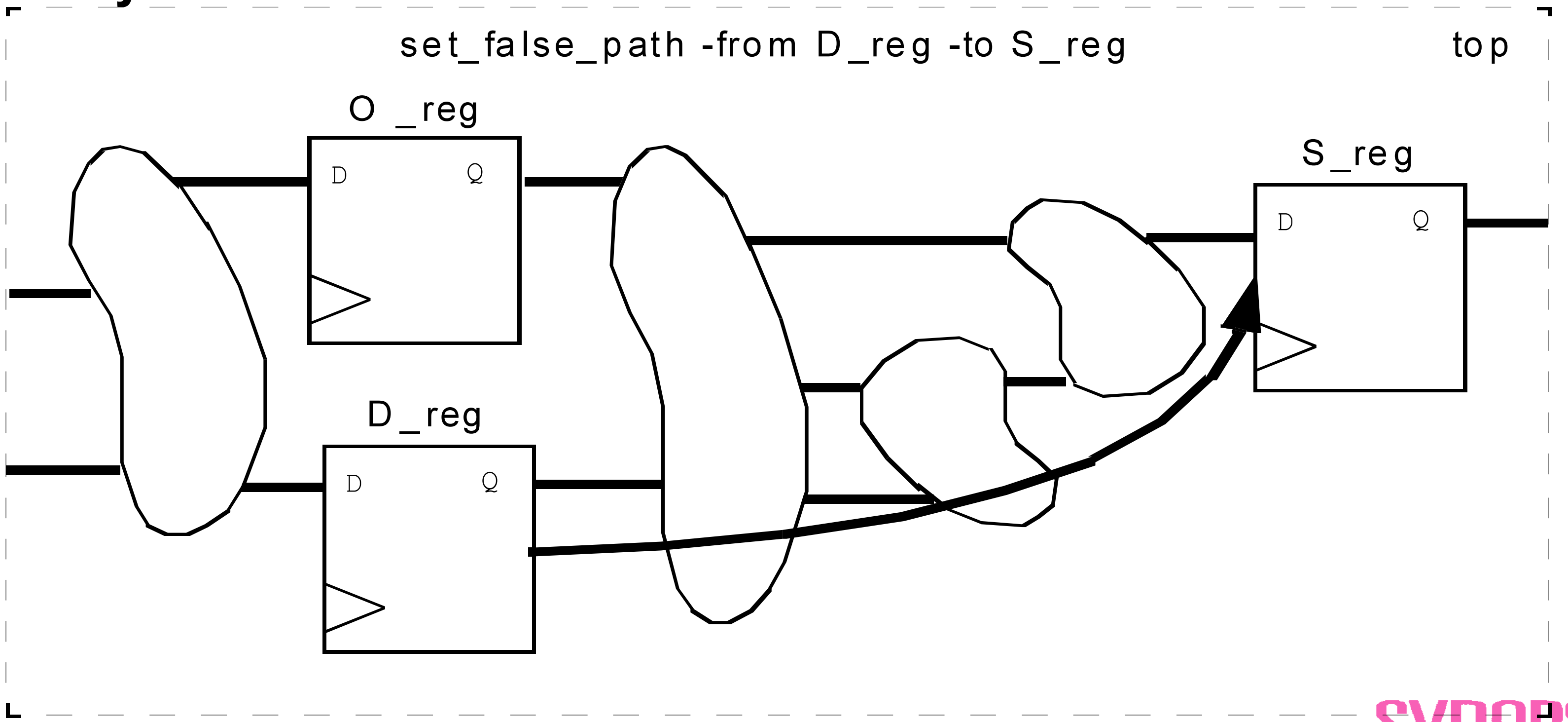
- **Symptom: Too many lines of code in block**
- **Pitfalls:**
  - analyze / elaborate steps are slow
  - code is difficult to read / inspect
- **Recommendation:**
  - blocks should contain only clock
  - blocks should have few timing exceptions
  - add a new level of hierarchy

Reasonable Hierarchy (211)

- **Use a reasonable number of levels in the hierarchy**
- **Pitfalls:**
  - reduced readability
  - longer compile times
  - more error prone (more places for error)
- **Recommend:**
  - Use 2-3 levels per major function block or algorithm,
  - Use no more than 8 levels per design.
  - A new level is introduced via DesignWare
    (can be eliminated by ungroup)

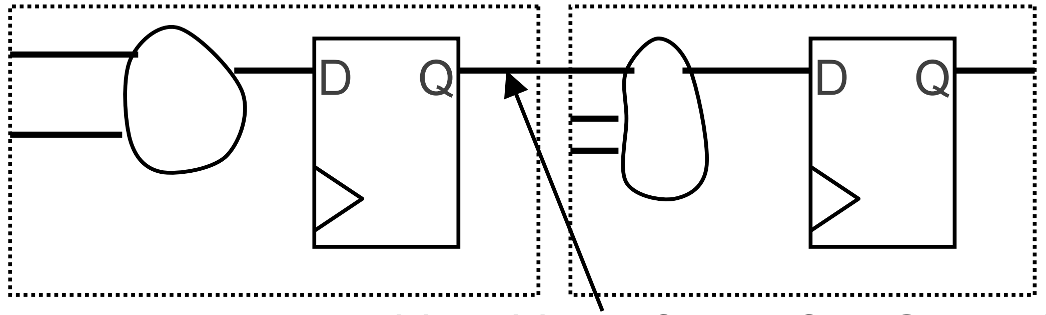
Isolate Point-to-Point Exceptions (212)

- If a design contains point-to-point exceptions (false path/multiple cycles), keep those exceptions within a module.
- By containing the point-to-point exception within one module, execution-time in static timing analysis or synthesis is minimized.



Register the Outputs (213)

- ❑ **To simplify the constraints and scripts process, register all outputs of a block.**



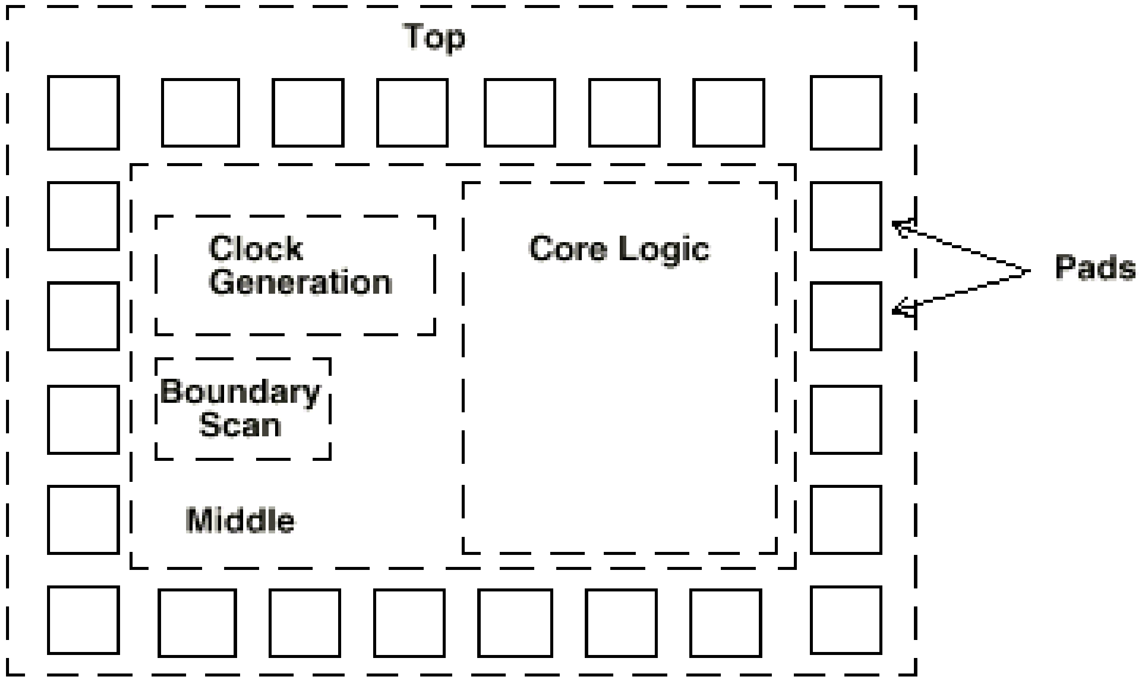
set_drive drive_of <my_flop/Q> all_inputs
set_input_delay 2 -clock CLK <input_port>

- ❑ **The drive strength of the inputs is predictable.**
- ❑ **The input delays from the previous block are predictable.**
- ❑ **It speeds up simulation, since the process activate only once per clock cycle.**

Chip-Level Partitioning (214)

Enabling Productivity

❑ The partitioning recommendation for the top of an ASIC

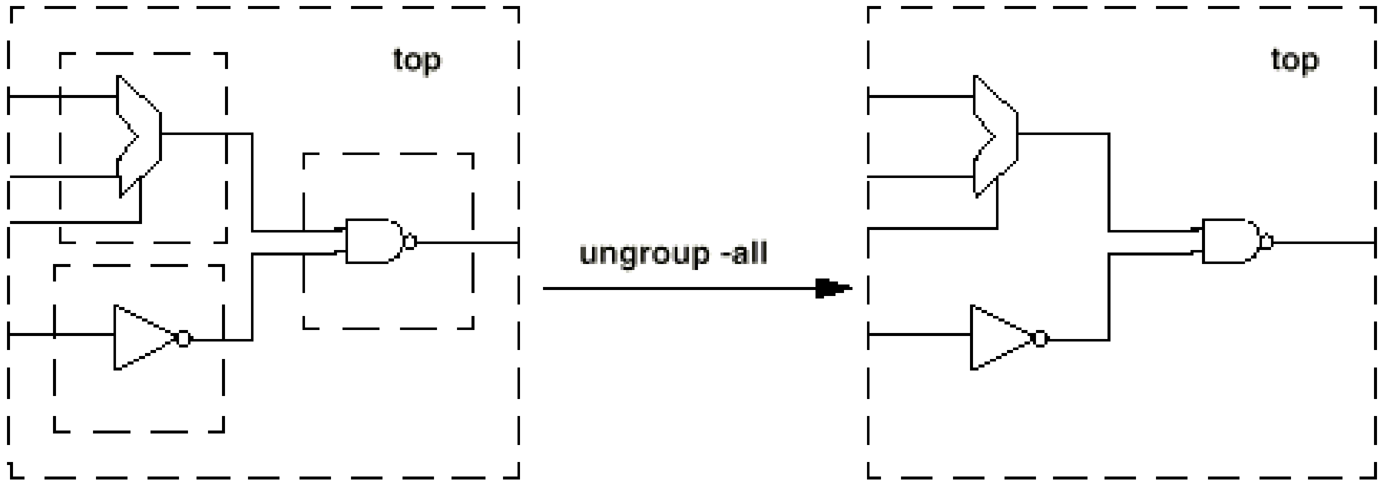


❑ The clock generation circuitry is isolated from the rest of the design, since typically it is handcrafted and carefully simulated.

Ungrouping a Design Hierarchy

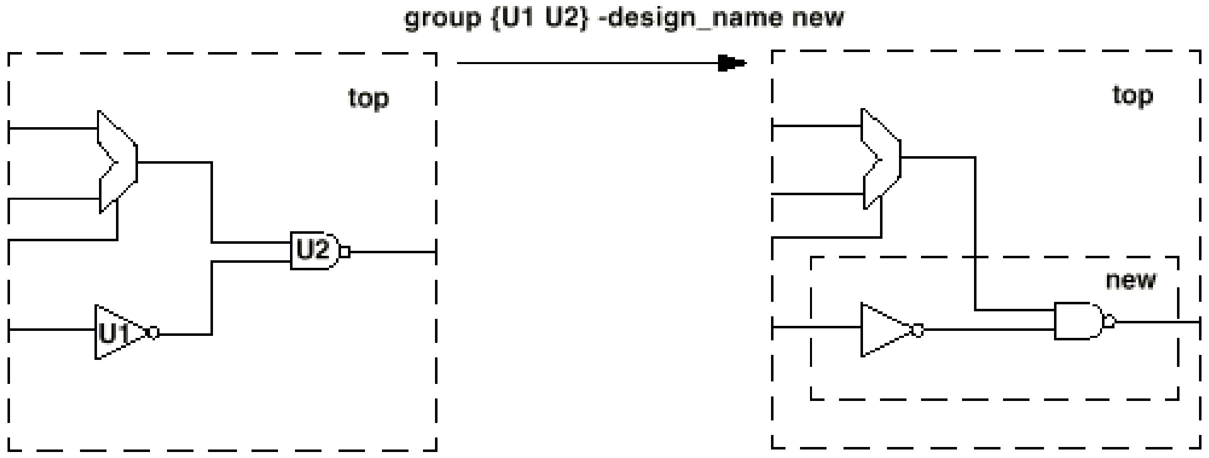
Enabling Productivity

- **The ungroup command collapses hierarchy.**



Group Cells into a Module

- The group command allows you to create new levels of hierarchy from the objects at this level.



Group Cells into a Module

Enabling Productivity

- **You can group individual HDL blocks with the hdl_block option of the group command.** -

```
Block label examples for Verilog source code
            // This is a named always block
            always @(A or B or C) begin: My_Process
            .......
            end
```

group

```
group -hdl_block My_Process -design My_Block
```

HDL for Synthesis Guidelines

Enabling Productivity

Presentation:

HDL for Synthesis Guidelines

- ❑ General HDL Code Structure
- ❑ Partitioning
- ■ Implying Logic Structure
- ❑ Safe Coding & Avoiding Problems
- ❑ Source Code Readability
- ❑ Coding Style for Design Reuse
- ❑ Design for Testability
- ❑ Practices

Implying Logic Structure: Checklist Items

- ❑ Unintentional latches (301)
- ❑ If vs. Case statement (302)
- ❑ Code organization & optimization (303)
- ❑ Resource sharing (304)
- ❑ Finite Statement Machines (305)
- ❑ Don't care inference (306)
- ❑ Coding of repetitive structures (307)
- ❑ Sharing Common Subexpression (308)
- ❑ Avoid Redundant Logic and Subexpress. (309)
- ❑ Inferring the Correct Register (310)
- ❑ Structure for Minimum Delay (311)
- ❑ Inferring Tri-State Drivers (312)

Imply Structure

Enabling Productivity

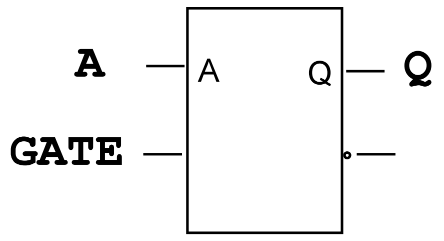
- ❑ Poor structure may never converge on the right results
- ❑ Poor structure usually means at least an increase in synthesis run times
- ❑ Designers imply lots of structure!

You get what you write!

Unintentional Latches (301)

An IF statement with outputs not fully specified synthesizes to a latch. Example:

```
module oops_latch ...

always @(GATE or A)
  begin
    if (GATE == 1)
      Q = A;
  end

endmodule
```



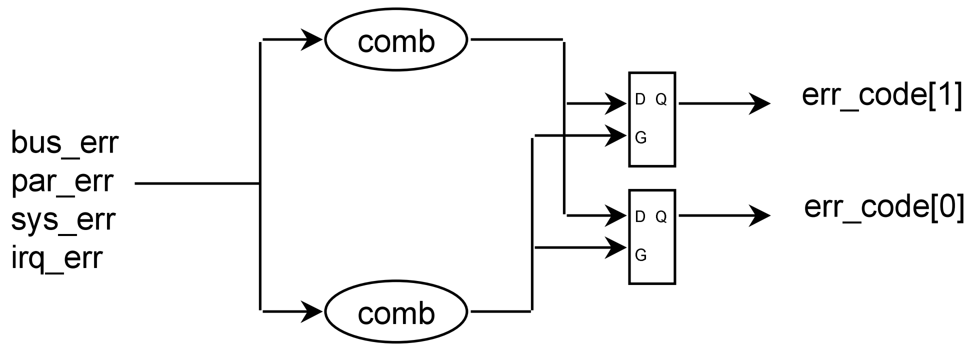
❑ **hdlin_check_no_latch = 'false' is default   If set to "true" HDL Compiler will issue a warning if a latch is synthesized.**

Unintentional Latches (301)

Enabling Productivity

A ‘case’ statement with paths that bypass reg assignment synthesizes to a latch. Example:

```
module oops_latch (bus_err, par_err, sys_err, irq_err, err_code);
input bus_err, par_err, sys_err, irq_err;
output [1:0] err_code;
reg    [1:0] err_code;

always @(bus_err or par_err or sys_err or irq_err)
  begin
    case ({bus_err, par_err, sys_err, irq_err})
      4'b1000: err_code = 0;
      4'b0100: err_code = 1;
      4'b0010: err_code = 2;
      4'b0001: err_code = 3;
    endcase;
  end

endmodule
```



Assign a value under all conditions. Assign value to all variables. Use a ‘default’ clause. Use ‘full_case’ directive.

If vs. Case Statements (302)

- **Priority Encoder :**

```
if (sel[0])
  z = dat[0];
else if (sel[1])
  z = dat[1];
else if (sel[2])
  z = dat[2];
else
  z = dat[3];
```

If HDL compiler cannot statically determine that branches are parallel, it synthesizes hardware that include a priority encode.

- **Simple "One Hot"Encoder:**

```
case (1'b1) // synopsys parallel_case
  sel[0] : z = dat[0];
  sel[1] : z = dat[1];
  sel[2] : z = dat[2];
  sel[3] : z = dat[3];
endcase;
```

Parallel_case: no cases overlap

Is synopsys directive needed ?

Code Organization & Optimization (303)

Enabling Productivity

- Organize code such that the latest arriving (design speed) or most frequent (simulation speed) event is evaluated first:

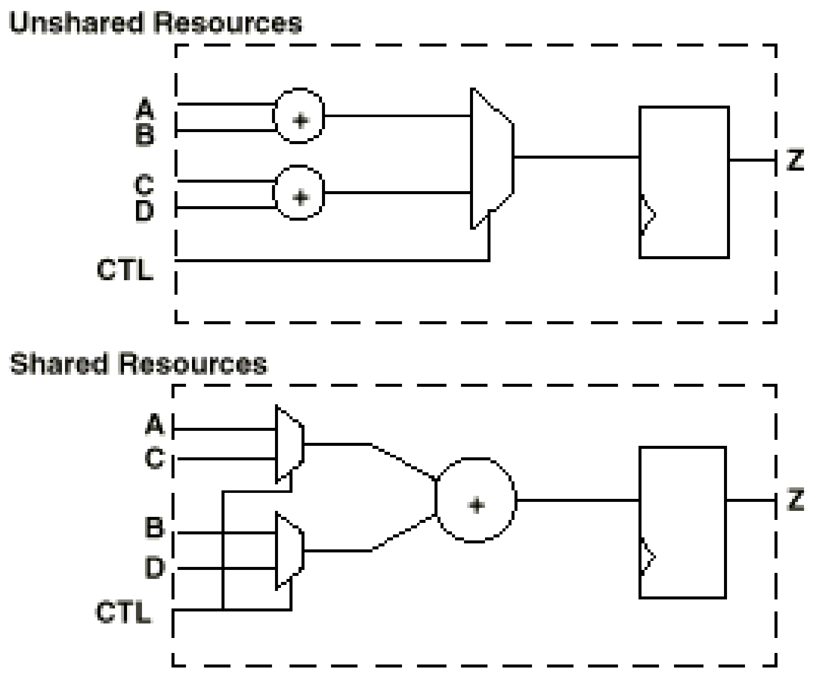
```
if (often)
   ....
else if (rare)

case (state)
     often: ...
     lessoften: ...
     rare: ....
```

- This approach speeds up design speed since the latest arriving signal is further down the logic cone
- This approach speeds simulation since the first condition evaluated is usually true eliminating the need for further processing.

Resource Sharing (304)

Enabling Productivity

- Design Compiler can share resources like adders or multipliers.
- Resource sharing can only occur if the resource allocation do not violate the limitations of scope and restrictions.



Resource Sharing (304)

Resource Sharing =

Resource Allocation

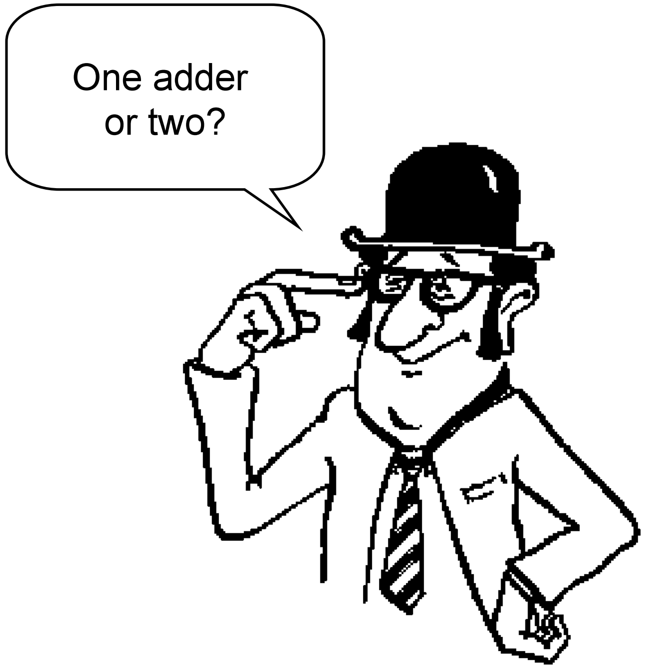
+

Implementation Selection

Resource Allocation (304)

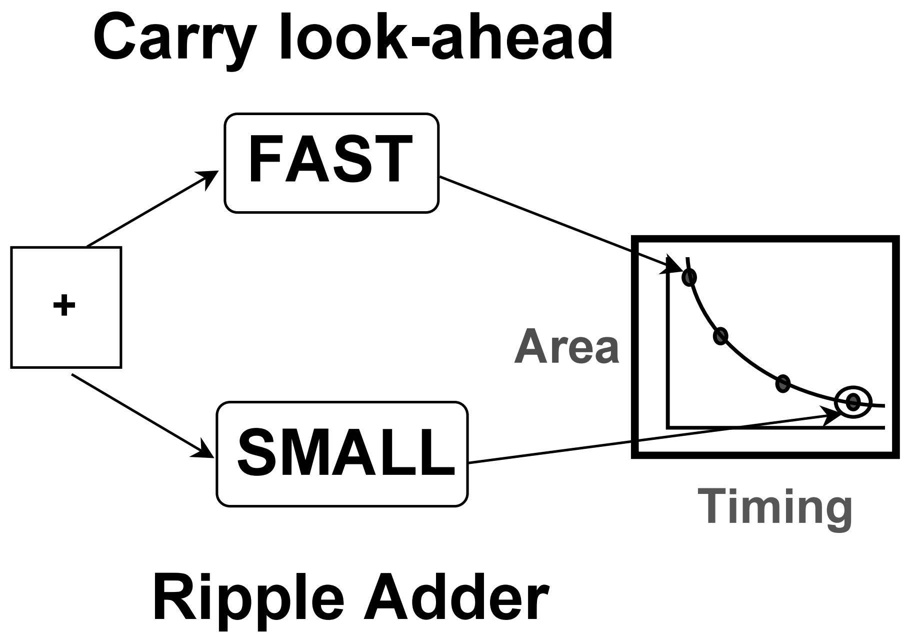
❑ **Resource Allocation is the process of determining the number of resources in your design.**

```
// Depending upon adder_control,
// select correct inputs.

if (adder_control)
  adder_output = busa + busb;
else
  adder_output = busc + busd;
```



Implementation Selection (304)

- ❑ **Implementation selection is the process of choosing the correct DesignWare architecture according to your constraints.**



Limitations of Resource Sharing (304)

- ❑ **Not all operations in your design can be shared. The following operators can be shared.**
  - ☞ ***, +, and -**
  - ☞ **>, >=, <, and <=**
- ❑ **Operations can be shared only if they lie in the same always block.**
- ❑ **Two operations can be shared only if no execution path exists from the start of the block to the end of the block that reaches both operations. (Control Flow Conflicts)**
- ❑ **Operations cannot be shared if doing so cause a combinational feedback loop. (Data Flow Conflicts)**

Resource Sharing (304): Scope

Enabling Productivity

```
always @(A1 or  B1 or C1 or D1 or COND_1)
  begin
    if (COND_1)
      Z1 = A1 + B1;
    else
      Z1 = C1 + D1;
  end
```

```
always @(A2 or B2 or C2 or D2 or COND_2)
  begin
    if (COND_2)
      Z2 = A2 + B2;
    else
      Z2 = C2 + D2;
  end process P2;
```

Allowed & Disallowed Sharing

	A1+B1	C1+D1	A2+B2	C2+D2
A1+B1	-	yes	no	no
C1+D1	yes	-	no	no
A2+B2	no	no	-	yes
C2+D2	no	no	yes	-

Only Operators in the same always@ block can be SHARED !

Resource Sharing (304): Control Flow Conflicts

```
always @(A or B or C or D or E or F or G
         or H or I or J or OP)
  begin: ADDER_SELECT
     Z1 = A + B;
     case (OP)
       2'b00 : Z2 = C + D;
       2'b01 : Z2 = E + F;
       2'b10 : Z2 = G + H;
       2'b11 : Z2 = I + J;
     endcase;
  end
```

Allowed & Disallowed Sharings

	A+B	C+D	E+F	G+H	I+J
A+B	-	no	no	no	no
C+D	no	-	yes	yes	yes
E+F	no	yes	-	yes	yes
G+H	no	yes	yes	-	yes
I+J	no	yes	yes	yes	-

Disable resource sharing only if logic is in CRITICAL PATH

Resource Sharing (304): Control Flow Conflicts

Enabling Productivity

- **Operations in separate branches of a ?: (conditional) construct cannot share the same hardware.**
- **Consider the following line of code where expressions_n represents any expressions.**
  - ☞ **z = expression_1 ? expression_2 : expression_3;**

HDL Compiler interprets this code as

```
temp_1 = expression_1;
temp_2 = expression_2;
temp_3 = expression_3;

z = temp_1 ? temp_2 : temp_3;
```

HDL Compiler evaluates both expression_2 and expression_3, regardless of the value of the conditional.

Therefore, operations in expression_2 cannot share the same resource as operations expression_3.

Resource Sharing (304): Data Flow Conflicts

- **To understand how sharing can cause a feedback loop, consider the following example.**

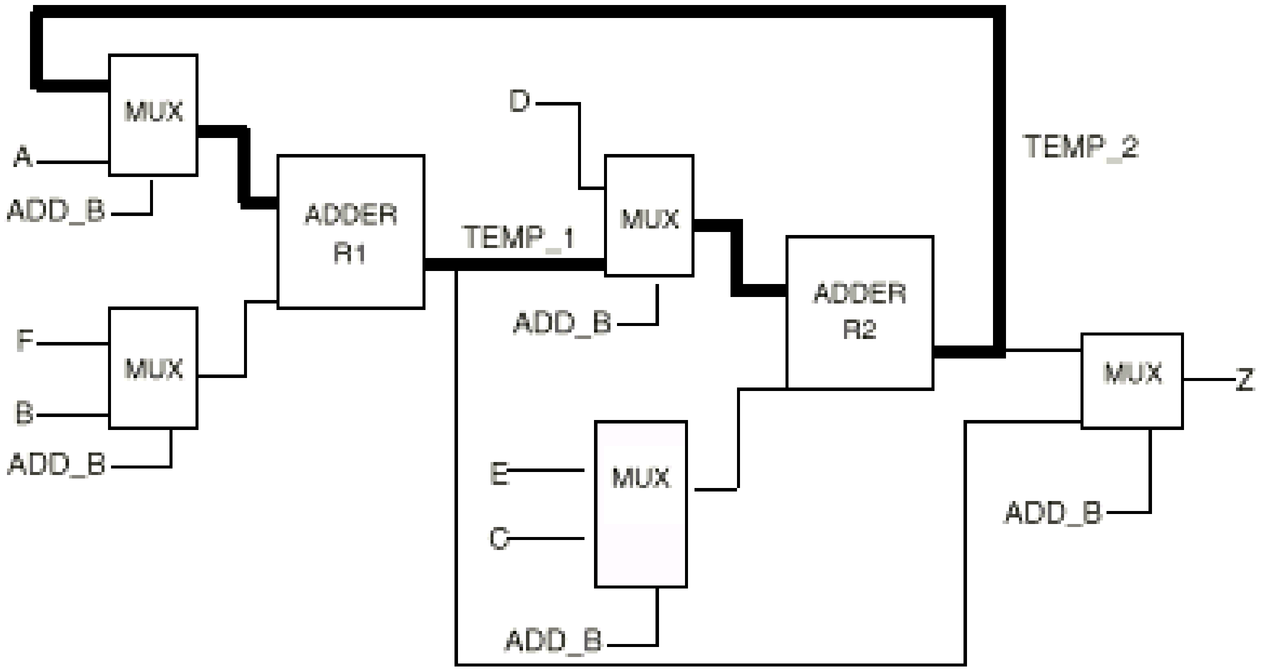
```
//Data Flow Conflict
always @(A or B or C or D or E or F or Z or ADD_B)
begin
    if(ADD_B) begin
        TEMP_1 = A + B;
         Z = TEMP_1 + C;
    end
    else begin
        TEMP_2 = D + E;
        Z = TEMP_2 + F;
    end
end
```

When the A+B addition is shared with the TEMP_2+F addition on an adder call R1 and the D+E addition is shared with the TEMP_1+C addition on an adder called R2, a feedback loop results

Resource Sharing (304): Data Flow Conflicts

Enabling Productivity

- Feedback Loop For the previous example.



HDL Compiler resource sharing mechanism does not allow combinational feedback paths to be created because most timing verifiers cannot handle them properly.

Critical Path Considerations...(304)

- ❑ **To enable automatic sharing for all designs, set the dc_shell variable as shown before you execute the compile command.**

```
dc_shell> hlo_resource_allocation = constraint_driven
```

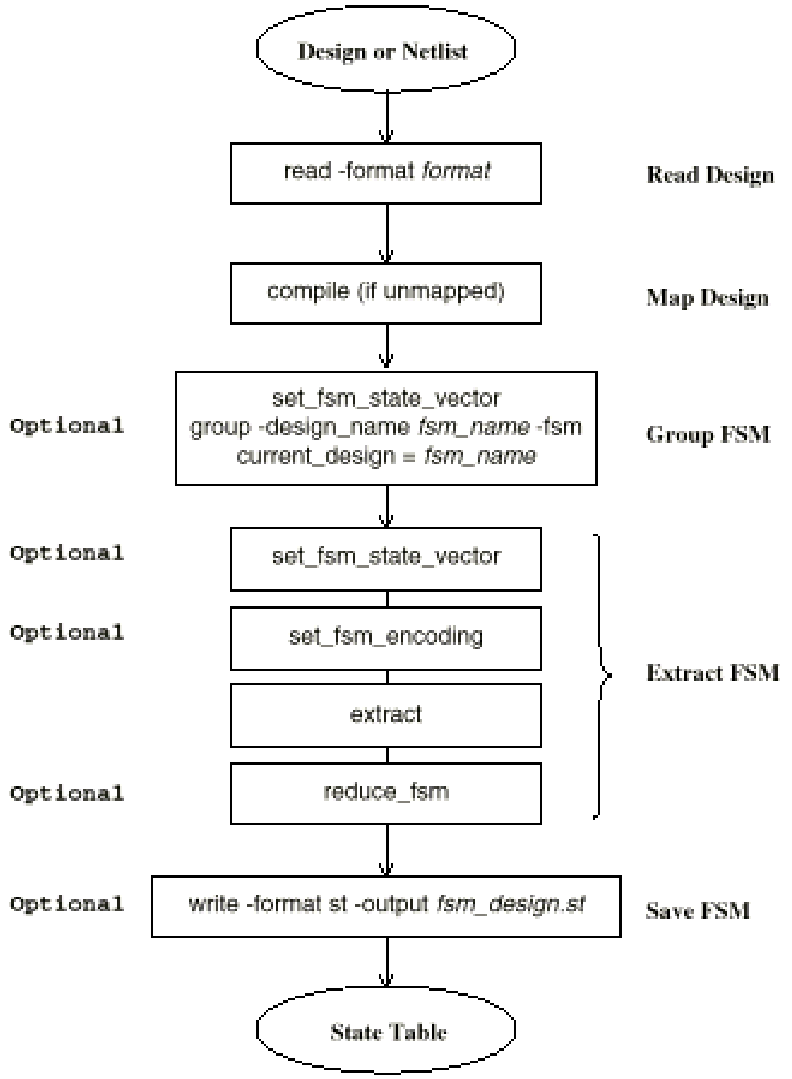
- ❑ **The default value for this variable is `constraint_driven`.**

- ❑ **To disable automatic sharing for uncompiled designs, and enable resource sharing only for selected designs, enter the following commands:**

```
dc_shell> hlo_resource_allocation = none
dc_shell> current_design = MY_DESIGN
dc_shell> set_resource_allocation constraint_driven
```

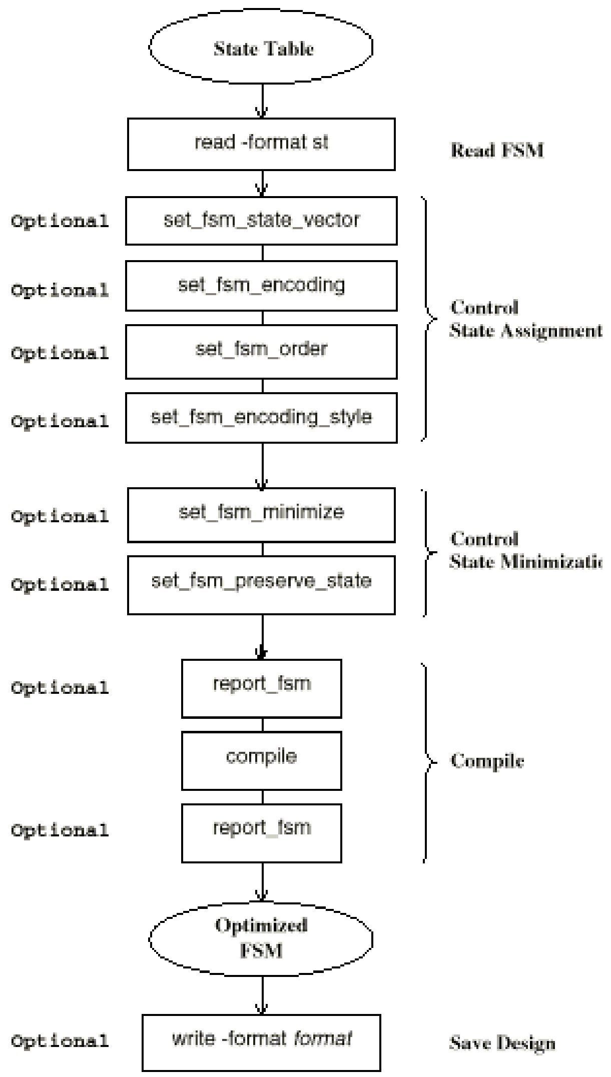
Finite State Machines (305): Extracting

Enabling Productivity



Finite State Machines (305): Compile

Enabling Productivity



Finite State Machines (305)

- ❑ Design Compiler uses logic and gate-level optimization techniques for synthesis.
- ❑ Two additional techniques are available for FSMs:
  - ☞ *State Minimization*

    removal of redundant states
  - ☞ *State Assignment*

    encoding styles (e.g. binary, gray, one-hot)
- ❑ Technique: (1) read in design, (2) map to gates, and (3) extract FSM

State Machine Extraction (305)

Enabling Productivity

❑ **Extract out the state machine when:**

- ☞ Number of states is from 15-32

  AND
- ☞ States and surrounding logic are random

  AND
- ☞ Best state ordering and number of bits is unpredictable

  AND
- ☞ Constraints are a mix of timing and area

State Machine Syntax (305)

- **Use Synopsys' directives and style if you wish to extract out a state machine.**

- **Synopsys Style:**
  - ☞ **Separate state machine into two processes**
  - ☞ **Create an enumerated type for the state vector**
  - ☞ **Drive FSM with embedded Synopsys directives**
  - ☞ **Read HDL Compiler for Verilog manual for detailed information**

After FSM extraction and optimization, back-annotate states into HDL code - but do not routinely flow through the extraction process.

FSM Verilog Example (305)

```
// This finite-state machine (Mealy type) reads 1 bit
// per cycle and detects 3 or more consecutive 1s.

module enum2_V(signal, clock, detect);
input signal, clock;
output detect;
reg detect;

// Declare the symbolic names for states
parameter [1:0]//synopsys enum state_info
     NO_ONES = 2'h0,
     ONE_ONE = 2'h1,
     TWO_ONES = 2'h2,
     AT_LEAST_THREE_ONES = 2'h3;

// Declare current state and next state variables.
reg [1:0] /* synopsys enum state_info */ cs;
reg [1:0] /* synopsys enum state_info */ ns;
```

```
// synopsys state_vector cs
always @ (cs or signal)
     begin
           detect = 0;// default values
           if (signal == 0)
               ns = NO_ONES;
           else
               case (cs) // synopsys full_case
                     NO_ONES: ns = ONE_ONE;
                     ONE_ONE: ns = TWO_ONES;
                     TWO_ONES,
                     AT_LEAST_THREE_ONES:
                           begin
                                 ns =
AT_LEAST_THREE_ONES;
                                 detect = 1;
                           end
               endcase
      end
always @ (posedge clock) begin
      cs = ns;
end
endmodule
```

Example Synopsys (Non Verilog) FSM Code

Enabling Productivity

```
# Soft drink machine -- Price is 15 cents
.design      soft_drink_machine
# Inputs:  clock and reset signals;
#          nickel, dime, and quarter input signals
.inputnames  clk reset nickel_in dime_in quarter_in
# Outputs:  nickel change, dime change, dispense drink
.outputnames nickel_out dime_out dispense
# Clock signal name and type
.clock clk rising_edge
# Asynchronous reset signal, type, and reset state
.asynchronous_reset reset rising IDLE
# State table
100 IDLE      FIVE      000
010 IDLE      TEN       000
...
%%% OWE_DIME   IDLE      010
# Wait in current state until money is deposited
000 IDLE      IDLE      000
000 FIVE      FIVE      000
000 TEN       TEN       000
```

Refer to the Design Compiler Family Reference Manual 's appendix for more information regarding Synopys FSM Code

Don't Care Inference (306)

Enabling Productivity

❑ **You can greatly reduce circuit area with don't-cares in your design.**
  - ☞ **Use x, z, ? in case items of the casex statement to infer don't-care conditions.**
  - ☞ **Use z, ? In case items of the casez statement to infer don't-care conditions.**

```
module decoder8_c(A, Z);
parameter N = 8;
parameter log2N = 3;
input  [N-1:0]     A;
output [log2N-1:0] Z;
reg    [log2N-1:0] Z;

always @(A) begin: encode
   casex (A) // synopsys full_case parallel_case
      8    00000001 : Z = 3    000;
      8    0000001x : Z = 3    001;
      8    000001xx : Z = 3    010;
      8    00001xxx : Z = 3    011;
      8    0001xxxx : Z = 3    100;
      8    001xxxxx : Z = 3    101;
      8    01xxxxxx : Z = 3    110;
      8    1xxxxxxx : Z = 3    111;
   endcase
end
endmodule
```

Coding of Repetitive Structures (307)

Enabling Productivity

```
module foo (j, k, l, z);
input  [3:0] j, k, l;
output [3:0] z;
reg    [3:0] z;

integer i;

always @(j or k or l)
  begin
    z[0] = (j[0] + k[0]) ^ l[0];
    z[1] = (j[1] + k[1]) ^ l[1];
    z[2] = (j[2] + k[2]) ^ l[2];
    z[3] = (j[3] + k[3]) ^ l[3];
    z[4] = (j[4] + k[4]) ^ l[4];
    z[5] = (j[5] + k[5]) ^ l[5];
    z[6] = (j[6] + k[6]) ^ l[6];
    z[7] = (j[7] + k[7]) ^ l[7];
end

endmodule
```

```
module foo (j, k, l, z);
input  [7:0] j, k, l;
output [7:0] z;
reg    [7:0] z;

integer i;

always @(j or k or l)
  begin
    for (i=0; i <=7; i=i+1)
      z[i] = (j[i] + k[i]) ^ l[i];
  end

endmodule
```

Sharing Common subexpressions (308)

- **Sharing common subexpressions might reduce the area of your circuit.**
- **You can manually force common subexpressions to be shared by declaring a temporary variable to store the subexpression, then use the temporary variable where you want to repeat the subexpressions.**

```
// Simple Additions with a Common Subexpression

temp = a + b;
x = temp;
y = temp + c;
```

Sharing Common subexpressions (308)

Enabling Productivity

- **You can let Design Compiler automatically determine whether sharing common subexpressions improves your circuit; however, there are some limitations that you should know.**

```
// Unidentified Command Subexpression
Y = A + B + C;
Z = D + A + B
```

The parser does not recognize A+B as a common subexpression, because the second equation as (D+A).

You can force the parser to recognize the common subexpression by rewriting the second assignment statements.

```
Z = A + B + D;
or
Z = D + (A + B)
```

Avoid Redundant Logic and subexpressions (309)

Enabling Productivity

- **Minimize redundant recalculation**

```
// Bad - Will synthesize four adders
X = A + B + C;
Y = D + C + A;

// Better - Will only synthesize three adders
T = A + C;
X = T + B;
Y = T + D;

// Bad - Will synthesize 4 multipliers and 3 adders
Z = A*C + A*D + B*C + B*D;

// Better - Will synthesize 1 multiplier and 2
adders
Z = (A + B) * (C + D);
```

Avoid Redundant Logic and subexpressions (309)

Enabling Productivity

- ❑ **Don't include statements in loops when their values don't change!**

```
for (i=0; i<=7; i=i+1)
  begin
    proc_data = pci_data;
    fifo(i) = cache(i-1);
  end
```

Better $\longrightarrow$

```
proc_data = pci_data;
for (i=0; i<=7; i=i+1)
  begin
    fifo(i) = cache(i-1);
  end
```

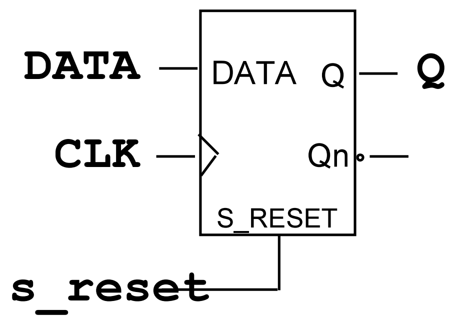
```
if (flag)
  begin
    proc_data = pci_data;
    proc_add = cache_add;
  end
else
begin
    proc_data = pci_data;
    proc_add = int_add;
  end
```

```
proc_data = pci_data;
if (flag)
  begin
    proc_add = cache_add;
  end
else
begin
    proc_add = int_add;
  end
```

Inferring the Correct Register - Synchronous Resets (310)

❑ An attribute is needed to guide DC in inferring synchronous sets or resets in a design.

```
module dff_sync_reset (data, clk, s_reset, q);
input data, clk, s_reset;
output q;
reg    q;

// synopsys sync_set_reset "s_reset"

always @(posedge clk)
  if (s_reset)
    q = 1'b0;
  else
    q = data;

endmodule
```

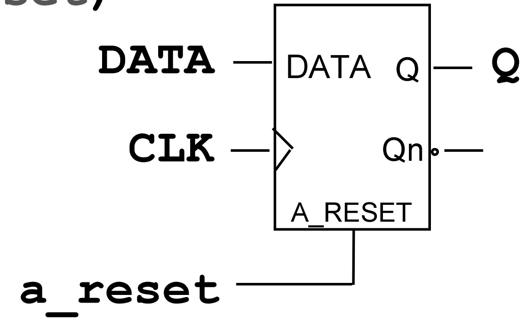


Check Your Technology Library For Sync Set/Reset Cells

Inferring the Correct Register - Asynchronous Resets (310)

❑ No attribute is needed to guide DC in inferring asynchronous sets or resets in a design. (Guidance is via coding style)

```
module dff_async_reset (data, clk, a_reset, q);
input data, clk, a_reset;
output q;
reg    q;

always @(posedge clk or posedge a_reset)
  if (a_reset)
    q = 1'b0;
  else
    q = data;

endmodule
```



Check Your Technology Library For Async. Set/Reset Cells

Check Inference Report

❑ **During Elaboration an Inference report is produced:**

```
dc_shell> elaborate top_level -arch rtl -lib WORK > elaborate.output
                           or
dc_shell> read -format verilog top.v > elaborate.output
```

Sample Inference Report:

```
Inferred memory devices in process 'synchronizer_reg'  in routine m68k line 334 in file
        '/home/design/syn/try8/hdl/m68k.v'.

==============================================================
|     Register Name      |  Type    | Width | Bus | AR | AS | SR | SS | ST |
==============================================================
|       asn_d_reg        | Flip-flop |   1     | - |  N   | N | N   | Y | N |
|       asn_s_reg        | Flip-flop |   1     | - |  N   | N | N   | Y | N |
|       ldsn_d_reg       | Flip-flop |   1     | - |  N   | N | N   | Y | N |
|       ldsn_s_reg       | Flip-flop |   1     | - |  N   | N | N   | Y | N |
|       mrwn_d_reg       | Flip-flop |   1     | - |  N   | N | N   | Y | N |
|       mrwn_s_reg       | Flip-flop |   1     | - |  N   | N | N   | Y | N |
|       udsn_d_reg       | Flip-flop |   1     | - |  N   | N | N   | Y | N |
|       udsn_s_reg       | Flip-flop |   1     | - |  N   | N | N   | Y | N |
==============================================================
```

Environment Var's for FF Inference

Enabling Productivity

- ❑ *`hdlin_ff_always_sync_set_reset = 'true'`*

  Each object in the reference design is interpreted as if sync_set_reset attribute/directive is present

- ❑ *`hdlin_check_no_latch = 'true'`*

  Used to generate a warning message during elaborate if a memory element is inferred in reference design.

See Synthesis Reference Manual for Others

Structure for Minimum Delay (311)

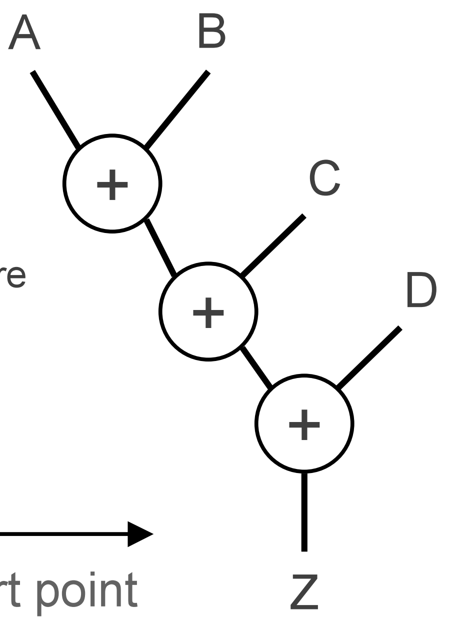
Enabling Productivity

- You can define the synthesis structure starting point and possibly obtains better quality of results.

Default Tree Structure
Z=A+B+C+D
or
Z=(A+B)+C+D;

yields

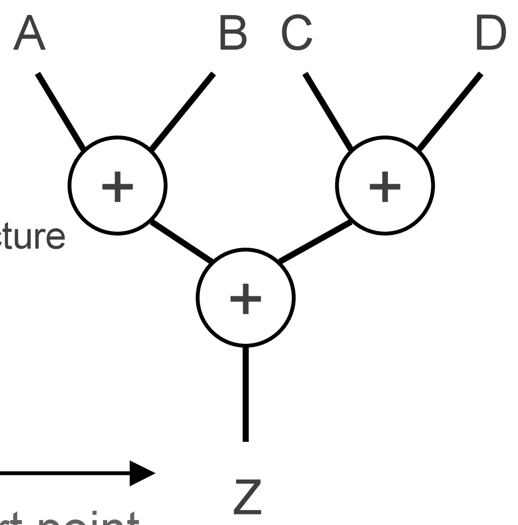
Synthesis start point



Balanced Tree Structure
Z=(A+B)+(C+D)

yields

Synthesis start point

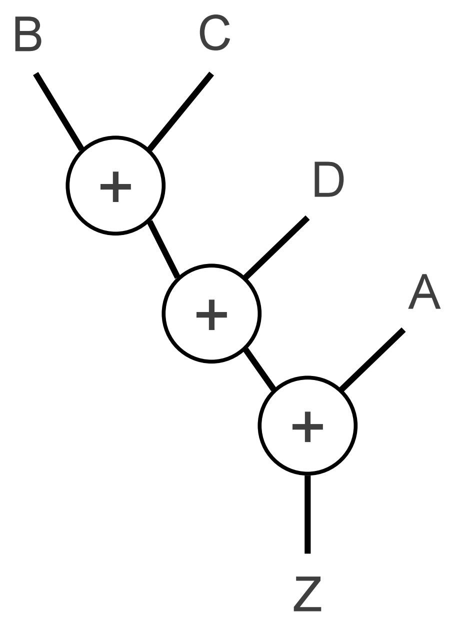


Structure for Minimum Delay (311)

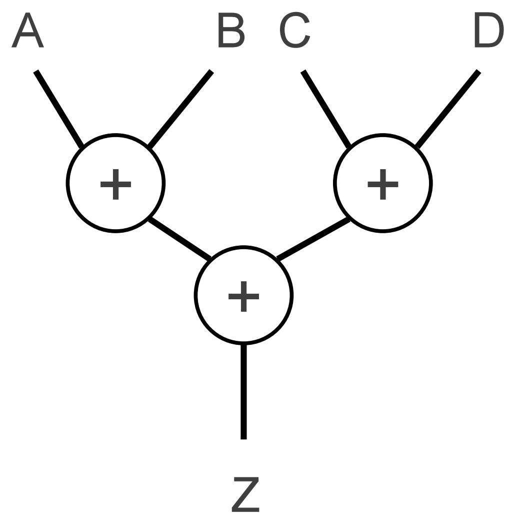
Enabling Productivity

- Consider the signal arrival times.

```
// Expression Tree with Minimum Delay
// Signal A arrives Last

Z= ((B+C) + D) + A;
```



```
// Expression Tree with Minimum Delay
// Same Arrival Times for All Signals

Z= (A + B) + (C + D);
```



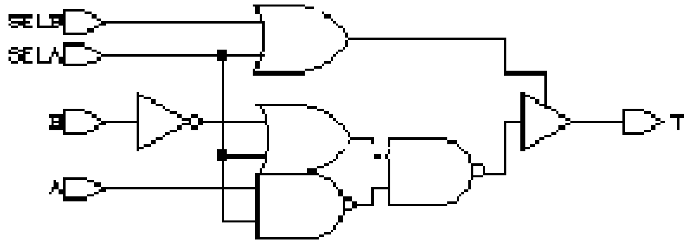
Inferring Tri-State Drivers (312)

Enabling Productivity

- **Tri-state logic is synthesized on the output driver when the output signal is assigned "Z".**
- **It's important to know that each always block can generate only one Tri-state buffer as an output driver.**

```
always @(from_table or enable)
begin: DRIVE_OUTPUT
    if (enable)
        to_bus = from_table;
    else
        to_bus = 8'bz;
end
```

```
always @(SELA or SELB or A or B) begin:
    T=1'bz;
    if (SELA)
        T = A;
    if(SELB)
        T = B;
end
```



SYNOPSYS®
Professional Service Group

Inferring Tri-State Drivers - Multiple Tri-state Drive Inference (312)

```
module tristate_a ( a, b, sela, selb, out1);
input a, b, sela, selb;
output out1;
reg out1;
    always @(slea or a)
        out1 = (sela) ? a : 1'bz;

    always @(selb or b) begin
        if (selb)
            out1=b;
        else
            out1=1'bz;
    end
endmodule
```

```
module tristate_a ( a, b, sela, selb, out1);
input a, b, sela, selb;
output out1;
wire out1;

    assign out1= (sela) ? a : 1'bz;
    assign out1= (selb) ? b : 1'bz;

endmodule
```

HDL for Synthesis Guidelines

Enabling Productivity

Presentation:

HDL for Synthesis Guidelines

Safe Coding & Problems: Checklist Items

- ❑ One clock per module (401)
- ❑ Separate Sequential & Combination Processes (402)
- ❑ Proper sensitivity lists (403)
- ❑ Blocking Statement vs. Non-blocking Statement (404)
- ❑ Named association (405)
- ❑ Instantiation of Sensitive or Asynch Circuits (406)
- ❑ Avoid Continuous Signal Assignments (407)
- ❑ Reset Strategy Consistency and properly coded (408)
- ❑ Instantiation of black-box (no timing) cells (409)
- ❑ Avoid Initialization (410)
- ❑ Avoid Mixed-Edge Sensitivity (411)
- ❑ Constant Propagation (412)

One Clock per Module (401)

- ❑ Synthesis was designed to optimize combinational logic clocked by a register driven from a single clock source
- ❑ Synthesis script development becomes much more complex with multiple clocks
- ❑ Asynchronous logic is often introduced as a result of logic with clock interfaces
- ❑ However .... it’s sometimes unavoidable. If more than one clock in a module then:
  - ☞ Estimate impact on testability
  - ☞ Estimate impact on synthesis
  - ☞ Estimate impact on mixed clocks timing analysis

Separate Sequential & Combination Processes (402)

```
module count (CLOCK, RESET, RESULT);
                input CLOCK, RESET;
                output RESULT;
                reg RESULT, AND_BITS, OR_BITS, XOR_BITS;
                reg [2:0] COUNT;

always @(posedge CLOCK) begin : BAD_EXAMPLE
if (RESET) begin
    COUNT <= 0;
    RESULT <= 0;
end
else begin
    COUNT <= COUNT + 1;
    AND_BITS <= & COUNT; // AND_BITS gets a Flip Flop
    OR_BITS  <= | COUNT; // OR_BITS gets Flip Flop
    XOR_BITS <= ^ COUNT; // XOR_BITS get a Flip Flop
    RESULT <= AND_BITS & OR_BITS & XOR_BITS;
end
end // BAD_EXAMPLE

endmodule
```

Code That Implies Extra Unwanted Registers

Separate Sequential & Combination Processes (402)

Enabling Productivity

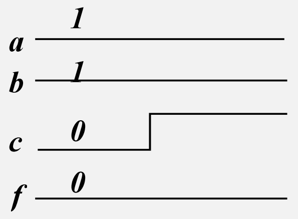
```
module count (CLOCK, RESET, RESULT);
input CLOCK, RESET;
output RESULT;
reg RESULT, AND_BITS, OR_BITS, XOR_BITS;
reg [2:0] COUNT;

always @(posedge CLOCK) begin : SEQ_BLK
   if (RESET) begin
                  COUNT <= 0;
                  RESULT <= 0;
   end
   else begin
                  COUNT <= COUNT + 1;
                  RESULT <= AND_BITS & OR_BITS & XOR_BITS;
                  end
end // SEQ_BLK

always @(COUNT) begin : COMB_BLK
   AND_BITS = & COUNT;
   OR_BITS  = | COUNT;
   XOR_BITS = ^ COUNT;
end // COMB_BLK

endmodule
```

Code Without Implying Extra Registers

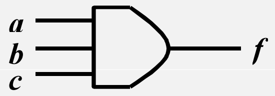
Proper Sensitivity Lists (403)

- Pitfall: gate-level simulation mismatch
- Symptoms:
  - ☞ Warnings during DC read/elaborate
  - ☞ Inconsistent behavior with slight change in stimulus

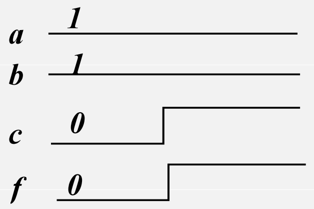
```
always @(a or b)
    begin
        f = a & b & c;
    end
```



Presynthesis Simulation



Synthesis Result



Postsynthesis Simulation

Incomplete Sensitivity List

Blocking vs. Non-blocking Assignments (404)

❑ Blocking procedural **assignments are more like S/W**
- ☞ reg changes immediately
- ☞ Sensitive to dependence
- ☞ Sensitive to assignment order
- ☞ Simulation speed improvement

❑ Non-blocking procedural **assignments are more like H/W**
- ☞ reg changes scheduled
- ☞ Insensitive to dependence
- ☞ Insensitive to assignment order

Blocking vs. Non-blocking Assignments (404)

- **Example of Blocking vs. Non-blocking**

```
always @(a or b or sel) begin : My_Mux
     if ( sel == 1'b0)
          out = a;
     else
          out = b;
end
```

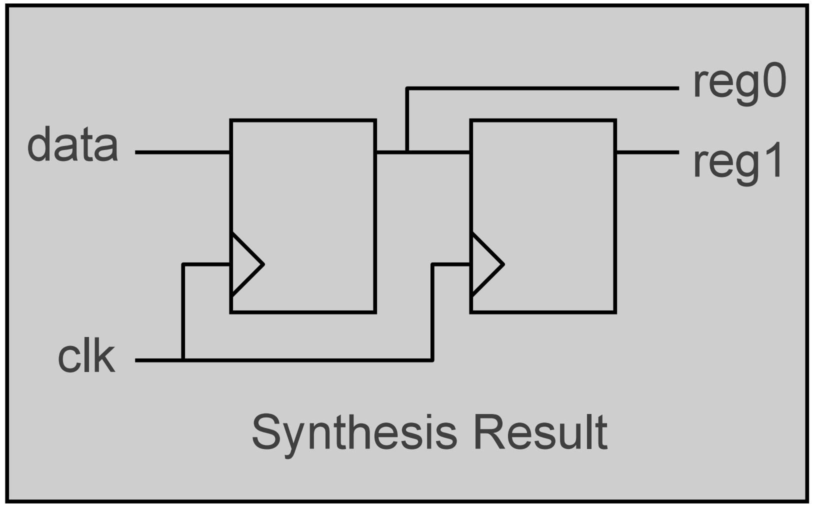
Blocking Assignment

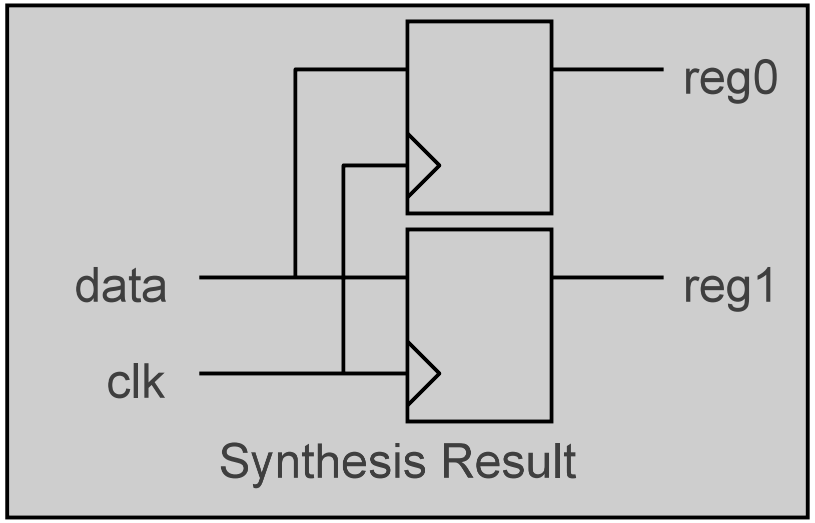
```
always @(posedge CLK) begin :
Shift_reg
     st1_reg <= data_in;
     st2_reg <= st1_reg;
     out_reg <= st2_reg;
end
```

Non-blocking Assignment

Blocking vs. Non-blocking Assignments (404)

❑ **Proper Use of Blocking and Non-blocking assignment in the sequential always block.**





```
//Proper use of Non-blocking assignment
always @(posedge clk) begin
      reg0 <= data;
      reg1 <= reg0;
end
```

```
//Improper use of Blocking assignment
always @(posedge clk) begin
      reg0 = data;
      reg1 = reg0;
end
```

Blocking vs. Non-blocking Assignments (404)

- **Improper use of blocking assignment might cause race condition.**
- **Potential Race**
  - The intention here is that ***a*** is shifted to ***b*** and ***b*** is shifted to ***c*** on the positive edge of the clock. However, since Verilog HDL does not specify the order where the always blocks are scheduled, the simulator may schedule that statement ***b=a*** before the statement ***c=b***.

```
//Potential Race
always @(posedge clk)
        c=b;
always @(posedge clk)
        b=a;
```

Named Association (405)

- **Instantiation port connection via order (implicit)**

```
my_adder U1 (base, offset, eff);
    - is not equivalent to -
my_adder U1 (base, eff, offset);
```

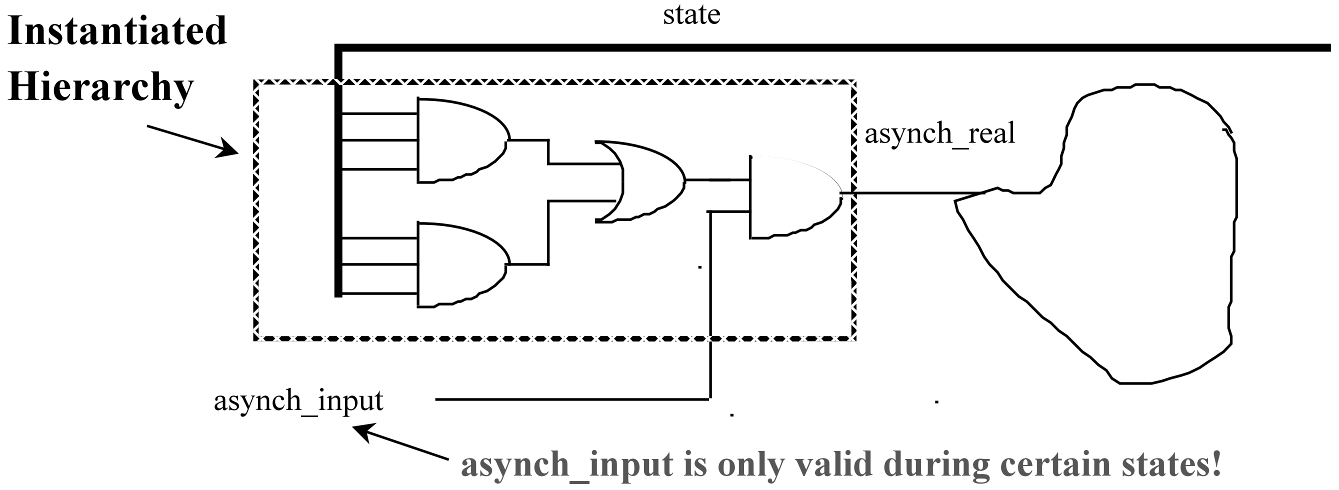
- **Instantiation port connection via name (explicit)**

```
my_adder U1 (.A(base), .B(offset), .SUM(eff));
                - is equivalent to -
my_adder U1 (.SUM(eff),.B(offset), .A(base));
```

Recommend: Always use name based port association.

Instantiation of Sensitive or Asynchronous Circuits (406)

- ❑ **Asynchronous logic is difficult to describe in an HDL or to time accurately through static timing analysis.**



INSTANTIATION and GATE SIMULATION is the correct methodology for asynchronous logic.

Avoid Continuous Signal Assignment (407)

- **Continuous assignment are executed in no defined order and synthesize to combinational logic.**
- **To facilitate the option to repartition in synthesis, and to improve code readability, you should place the logic in a combinational always block instead.**

```
// Continuous Assignment Example -- Not Recommended
assign sum = a_in ^ b_in ^ c_in;
assign c_out = (a_in & b_in) | (b_in & c_in) | (a_in & c_in);
```

```
// Combinational process -- Recommended
always @(a_in or b_in or c_in) begin: Full_Adder
    sum = a_in ^ b_in ^ c_in;
    c_out = (a_in & b_in) | (b_in & c_in) | (a_in & c_in);
end // Full_Adder
```

Reset Strategy Consistency (408)

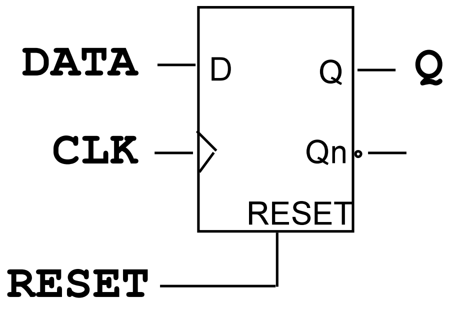
- ❑ **Synchronous or asynchronous external reset ?**
- ❑ **What FF cells are available in the library ?**
  - ☞ **Sync set/reset**
  - ☞ **Async set/reset**
- ❑ **Which strategy does the ASIC vendor prefer ?**
- ❑ **What attributes should I set in source code?**
- ❑ **How can I audit proper inference of registers ?**

Plan a Reset Strategy & Then Use it Consistently

Reset Strategy Consistency (408)

❑ **Example: Synchronous Reset**

```
// synopsys sync_set_reset "RESET"
always @(posedge CLK)
  begin
    if (RESET)
      Q = 1'b0;
    else
      Q = DATA;
  end
```

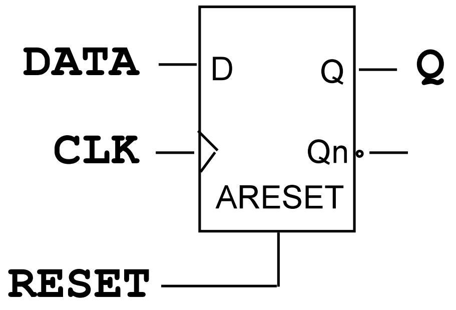


Reset Strategy Consistency (408)

Enabling Productivity

- ❑ **Example: Asynchronous Reset**

```
always @(posedge CLK or posedge RESET)
  begin
    if (RESET)
      Q <= 1'b0;
    else
      Q <= DATA;
  end
```



Reset Strategy Consistency (408)

- ❑ Apply a consistent reset strategy, synchronous or asynchronous.
  - ☞ Simplify Synthesis, DFT…efforts.
- ❑ Infer minimum area D flip-flop cells only when the designer is 100 percent certain that the circuit will self-initialize with no ambiguity.
- ❑ Make sure that the circuit will self-initialize.
  - ☞ Simulate the gate level design before logic implementation.

Instantiation of Black-Box Cells (409)

- ❑ **Timing-Driven Synthesis Requires Timing to be Defined for all Components**
- ❑ **Static-Timing Analysis is also Dependent on Full Timing**
- ❑ **Synthesis Timing Models:**
  - ☞ **Vendor Supplied (ie. LSI RAM Model)**
  - ☞ **Designer Created with Library Compiler Constructs**
    - ⇨ **No Special Library License Required (cell’s don’t have function statements - thus cannot be inferred)**

Example Synthesis-Timing Model (409)

```
library (RAM_LIBRARY) {

cell(RAM_64x8) {
  area : 0;
  pin(WE) {
   direction : input;
   capacitance : 1;
  }
  bus (A) {
   bus_type : BUS6 ;
   direction : input;
   capacitance : 1;
  }
```

Ex. Synthesis-Timing Model-cntd (409)

Enabling Productivity

```
bus (D_IN) {
  bus_type : BUS8 ;
  direction : input;
  capacitance : 1;
 }
 bus (D_OUT) {
  bus_type : BUS8 ;
  direction : output ;
  pin(D_OUT[0]) {
   timing () {
    intrinsic_rise : 25.0;
    intrinsic_fall : 25.0;
    related_pin : "A[5] A[4] A[3] A[2] A[1] A[0] D_IN[0] WE" ;}
  }
```

Avoid Initialization (410)

- Do not initialize; synthesis will ignore !

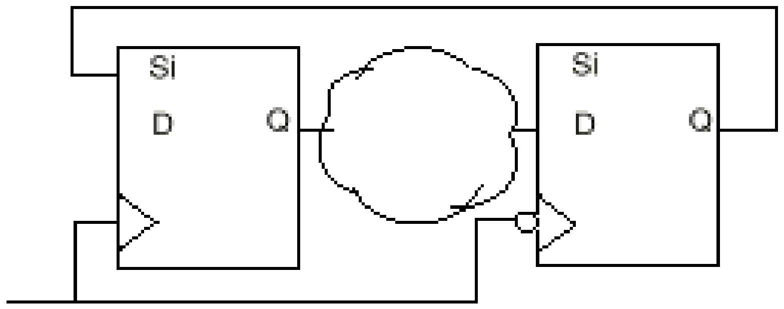
```
initial
  begin
    count = 0;
  end

always @(posedge CLK)
  begin
    count = count + 1;
  end
```

What will synthesis produce?

Avoid Mixed-Edge Sensitivity (411)

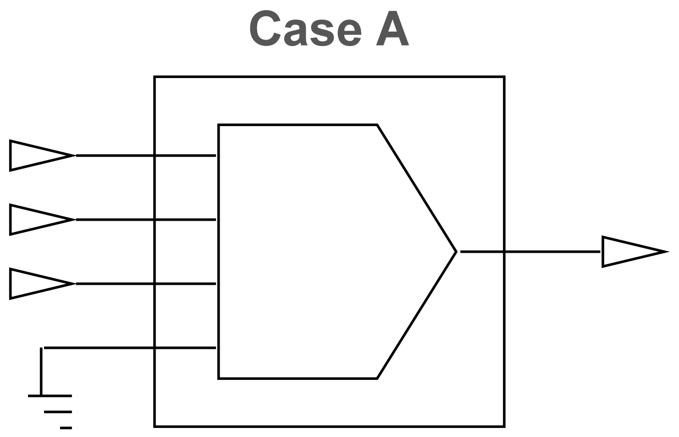
```
always @(posedge CLK or negedge CLK)
  begin
    if (CLK)
      countA = countA + 1;
    else if (!CLK)
      countA = countA + 2;
  end

always
  begin
    @(posedge CLK);
    countB = countB + 1;
    @(negedge CLK);
    countB = countB + 2;
  end
```

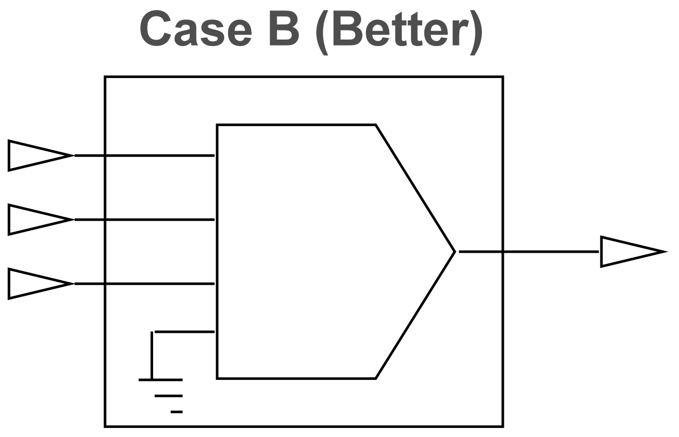
Mixing Clock Edges Example



1. The duty cycle of the clock becomes a critical issue in timing analysis, in addition to the clock frequency itself.
2. Most scan-based testing methodologies requires separate handling of positive and negative edge triggered flops.

Constant Propagation -1 (412)





- **Tie-off pins on subdesigns at the lowest level and don’t propagate as a primary port if not necessary**
- **This will help avoid:**
  - **- problems with constant propagation**
  - **- possible unconnected port issues**
  - **- netlist translation issues (VHDL, Verilog, EDIF, etc.)**

DC can’t eliminate redundant logic across boundaries when connected to ports

HDL for Synthesis Guidelines

Enabling Productivity

Presentation:

HDL for Synthesis Guidelines

- ❑ General HDL Code Structure
- ❑ Partitioning
- ❑ Implying Logic Structure
- ❑ Safe Coding & Avoiding Problems
- ■ **Source Code Readability**
- ❑ Coding Style for Design Reuse
- ❑ Design for Testability
- ❑ Practices

Readability: Checklist Items

Enabling Productivity

- ❑ **Meaningful embedded comments (501)**
- ❑ **Use of Loops & Arrays (502)**
- ❑ **Use of Constants (503)**
- ❑ **Reduction Operators (504)**
- ❑ **Proper use of ‘define & parameter’ (505)**

Meaningful Embedded Comments (501)

- ❑ Improve readability, maintainability, ability to reuse, easy review, trace-ability to spec, etc.

- ❑ Typical engineers response ...

  "I don't have the time to comment now"

- ❑ When complete do you go back & comment or do you move on to the next assignment ?

Use of Arrays and Loops (502)

- **Use Higher-Level Looping Constructs**
  - **For Loop & While Loop**
- **For Verilog: use ‘defines**
- **Use Arrays instead of group of bits (see example next)**

Example of Register Bank (502)

```
module RGBANK (CLK, WE, ADDR, DATA_IN, DATA_OUT);
input        CLK, WE;
input  [1:0] ADDR:
input  [7:0] DATA_IN;
output [7:0] DATA_OUT;
reg    [7:0] DATA_OUT;

reg [7:0] RG_0, RG_1, RG_2, RG_3;

always @(ADDR or RG_0 or RG_1 or RG_2 or RG_3)
  begin
    DATA_OUT = 0;
    case (ADDR)
      0: DATA_OUT = RG_0;
      1: DATA_OUT = RG_1;
      2: DATA_OUT = RG_2;
      3: DATA_OUT = RG_3;
    endcase;
  end

always @(posedge CLK)
  begin
    if (WE)
      case (ADDR)
        0: RG_0 = DATA_IN;
        1: RG_1 = DATA_IN;
        2: RG_2 = DATA_IN;
        3: RG_3 = DATA_IN;
      endcase;
  end
endmodule // RGBANK
```

Register Bank Using Arrays (502)

```
module RGBANK (CLK, WE, ADDR, DATA_IN, DATA_OUT);
input        CLK, WE;
input  [3:0] ADDR:
input  [7:0] DATA_IN;
output [7:0] DATA_OUT;

reg [7:0] RG[3:0];

assign DATA_OUT = RG[ADDR];

always @(posedge CLK)
  begin
    if (WE)
      RG[ADDR] = DATA_IN;
  end
endmodule // RGBANK
```

Use of Constants (503)

- Constants are a very simple way of improving Verilog source code readability and code quality by eliminating typographical errors.
- Sometimes, if the architecture changes, only the constants need to be updated.

```
// in a header file, declare all constants shared by more than
// one module
`define INTBUS_WIDTH 16
`define EXTBUS_WIDTH 32
…
`define DEVICE_ID 16’h0007
`define REVISION_ID 16’h0002
```

Use of Constants (503)

Enabling Productivity

```
// within module files, declare all local constants
`define CONTROL_OFFSET = 3'b000
`define STATUS_OFFSET = 3'b001
`define MASK_OFFSET = 3'b010
`define INT_OFFSET = 3'b011
…
// Register Read Mux
case (addr)
        `CONTROL_OFFSET : data_out = control_reg;
        `STATUS_OFFSET   : data_out = status_reg;
        `MASK_OFFSET     : data_out = mask_reg;
        `INT_OFFSET      : data_out = int_reg;
```

Use of Reduction Operators (504)

Enabling Productivity

```
These verbosity...
     AND          z = a[0]  & a[1]  & a[2];
     OR  z = a[0]  | a[1]  | a[2];
     XOR          z = a[0]  ^ a[1]  ^ a[2];
     NAND         z = ~(a[0] & a[1] & a[2]);
     NOR          z = ~(a[0] | a[1] | a[2]);
     XNOR         z = ~(a[0] ^ a[1] ^ a[2]);
```

```
... become these concise statements.
     AND          z = &a;
     OR           z = |a;
     XOR          z = ^a;
     NAND         z = ~&a;
     NOR          z = ~|a;
     XNOR         z = ~^a;
```

Use of Reduction Operators (506)

Enabling Productivity

- ❑ **Parity Logic:**

```
EVEN_PARITY = ^DATA[7:0];

ODD_PARITY = ~^DATA[7:0];
```

-or-

```
EVEN_PARITY = DATA[7] ^ DATA[6] ^ DATA[5] ^ DATA[4]
            ^ DATA[3] ^ DATA[2] ^ DATA[1] ^ DATA[0];

ODD_PARITY = DATA[7] ~^ DATA[6] ~^ DATA[5] ~^ DATA[4]
          ~^ DATA[3] ~^ DATA[2] ~^ DATA[1] ~^ DATA[0];
```

Proper Use of `defines & parameters (505)

- **`define**
  - Text substitution
  - Typical uses include
    - constants
    - readability improvement
- **parameter**
  - Represents constants
  - Can be modified at compile time
  - Modified via
    - defparam statement
    - module instance statement
  - Typical uses include
    - delay specification
    - width of variables

Proper Use of `defines & parameters (505)

Enabling Productivity

pc_defines.v

```
`define SERIAL_CS   16'h1050
`define PARALLEL_CS 16'h23ff
`define FLOPPY_CS   16'h4b80
```

Do you have a preference?

io_control.v

```
`include "pc_defines.v"
...
...

if (ADDR == `SERIAL_CS)
  ...
else if (ADDR == `PARALLEL_CS)
  ...
else if (ADDR == `FLOPPY_CS)
  ...
```

```
...
...
...

if (ADDR == 16'h1050)
  ...
else if (ADDR == 16'h23ff)
  ...
else if (ADDR == 16'h4b80)
  ...
```

Proper Use of `defines & parameters (505)

Enabling Productivity

```
module regbank (clk, data_in, data_out);
parameter size = 8, delay = 1;
input  [size-1:0] data_in;
output [size-1:0] data_out;
reg    [size-1:0] data_out;

always @(posedge clk)
  data_out = #delay data_in;

endmodule
```

```
module top;
reg clk;
reg [15:0] inA;
reg [3:0]  inB;
wire [15:0] outA; // need delay of 3
wire [3:0]  outB; // need delay of 2

regbank #(16, 3) U1 (clk, inA, outA);
regbank U2 (clk, inB, outB);

endmodule
```

```
module annotate;

defparam
  top.U2.size = 4,
  top.U2.delay = 2;

endmodule
```

HDL for Synthesis Guidelines

Enabling Productivity

Presentation:

HDL for Synthesis Guidelines

Code Reuse: Checklist Items

Enabling Productivity

- ❑ **Don’t Embed Synthesis scripts in source code (601)**
- ❑ **Maintain technology independence (602)**
- ❑ **Use GTECH for simple cell instantiation (603)**
- ❑ **Databook Quality Description (604)**
- ❑ **Parameterize modules (605)**

Code Reuse Principal

- **Design reuse is the action of utilizing objects in the form of macros, subsystems, and systems in the development of new systems**
- **Design object - with it's associated views (interface, functional spec, etc.) is intended for use in an "object oriented" way**
- **For example, an implementation of CCITT H.261 (Video Compression Std) should be implemented such that it can be reused in other systems with minimal effort**
- **Other examples: PCI bus, ADPCM, MPEG decoder, JPEG, etc.**

Levels of Code Reuse

- ❑ **Reuse by the individual**
  - ☞ Commonly done, but limited
  - ☞ For example: Counter, Mux, RAM Model, etc.
- ❑ **Reuse within a group**
  - ☞ Short lifetime, but improved
  - ☞ For example: Adaptive Equalizer
- ❑ **Reuse by department/lab**
  - ☞ Reasonable lifetime, significant productivity benefits
  - ☞ For example: MPEG Decoder
- ❑ **Reuse across enterprise Highest level of reuse**
  - ☞ Significant competitive advantage

Don't Embed Synthesis Scripts in Source Code(601)

- Example of embedded dc_shell script:

```
// synopsys dc_script_begin
// set_max_area 2500.0
// set_drive -rise 1 port_b
// synopsys dc_script_end
```

- Or hiding simulation constructs or other (e.g. FPGA) from synthesis compiler

```
-- translate_off
initial
.  .  .
-- translate_on

-- then set hdlin_translate_off_skip_text = false
   to have DC analyze and elaborate the module
```

Maintain Technology Independence (602)

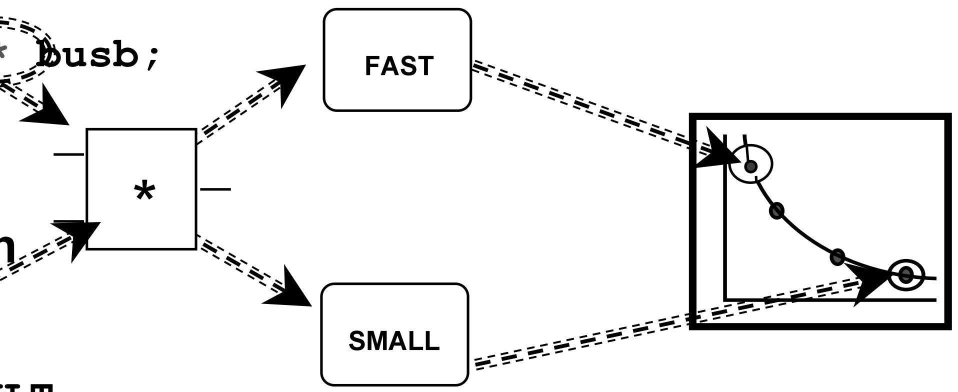
Enabling Productivity

- ❑ **Use DesignWare components**
  - ☞ DesignWare components are pre-verified for synthesis and can save you time coding and testing your design.
  - ☞ Using DesignWare components can also improve your quality of results.
- ❑ **The DesignWare Library is extensive and is broken down into five families:**
  - ☞ Standard Family(adder, subtractor, multiplier, comparator, etc)
  - ☞ ALU Family(barrel, shifter, incrementer/decrementer, etc)
  - ☞ Advanced Math Family(advanced multiplier, vector add/subtract, etc)
  - ☞ Sequential Family(FIFOs, Gray-Scale counters, stack, etc)
  - ☞ Fault Tolerant Family(parity checker, CRC generator, etc)
  - ☞ Refer to DesignWare Library documentation for detailed information.

Synthetic Parts: Using DesignWare (602)

Through HDL inference

```
...
if (OPCODE_1) then
  mult_output <= busa * busb;
```



Through HDL instantiation

```
 // instantiate DW02_MULT
DW02_MULT #(wordlength1, wordlength2)
   U1 (in1, in2, control, product)
```

Reference Design

Synthesis Binds to Synthetic Module

Synthesis Selects Proper Implementation

Implementation Optimized for Context

Use GTECH for simple cell instantiation (603)

❑ **When necessary to instantiate ... use GTECH !**

- ☞ **Provides technology independence**
- ☞ **Typical GTECH cells (AND, NAND, OR, NOR, XOR, FA, HA, FF, LATCH, AOI, MUX, etc.)**
- ☞ **Use *`map_only`* attribute to prevent DC from *`ungrouping`* or**
- ☞ **In dc_shell:**

```
set_map_only { find(reference “my_gtech_cell” }
```

Using GTECH - verilog (603)

```
`include "<SYNOPSYS_ROOT>/packages/gtech/src_ver/gtech_lib.v"

module top (...);
...
...

GTECH_AND2  U1 (.A(in1), .B(in2), .Z(out1));
GTECH_NAND2 U2 (in3, in4, out2);

...
...
endmodule
```

GTECH instantiation allows a technology independent HDL description.

Databook Quality Description (604)

Enabling Productivity

- Databook-like quality implies published quality documentation.
- It's worth spending the effort to produce databook-like comments and consider the following characteristics:
  - Readable Documentation
  - Traceability to Specification
    - Block diagrams
    - Functional specification
    - Description of parameters and their use
    - Interface signal descriptions
    - Timing diagrams and requirements
    - Verification strategy
    - Synthesis constraints

Databook Quality Description (604)

Enabling Productivity

- Continue...
  - Useful Examples of How To Use the Module
  - A complete Testbench for the Module
    - Verification reports (what was tested)
    - Technology used

Parameterize Modules (605)

- **The use of the parameter construct improve the ability to reuse this module because of the parameterization provided by the parameter statement.**

```
module FIFO (CLK, WRITE_ENABLE, WRITE_SELECT, READ_SELECT,
                                        DATA_IN, DATA_OUT);

parameter SELECT_WIDTH = 3;
parameter DATA_WIDTH = 8;
parameter FIFO_DEPTH = 8;

input CLK, WRITE_ENABLE;
input [SELECT_WIDTH-1:0] READ_SELECT, WRITE_SELECT;
input [DATA_WIDTH-1:0] DATA_IN;
```

You can change the parameter value in a module
during instantiation or elaborating designs in synthesis.
module_name #(parameter_value,…..) instance_name(port list)
or
elaborate design_name -parameters parameter_list

HDL for Synthesis Guidelines

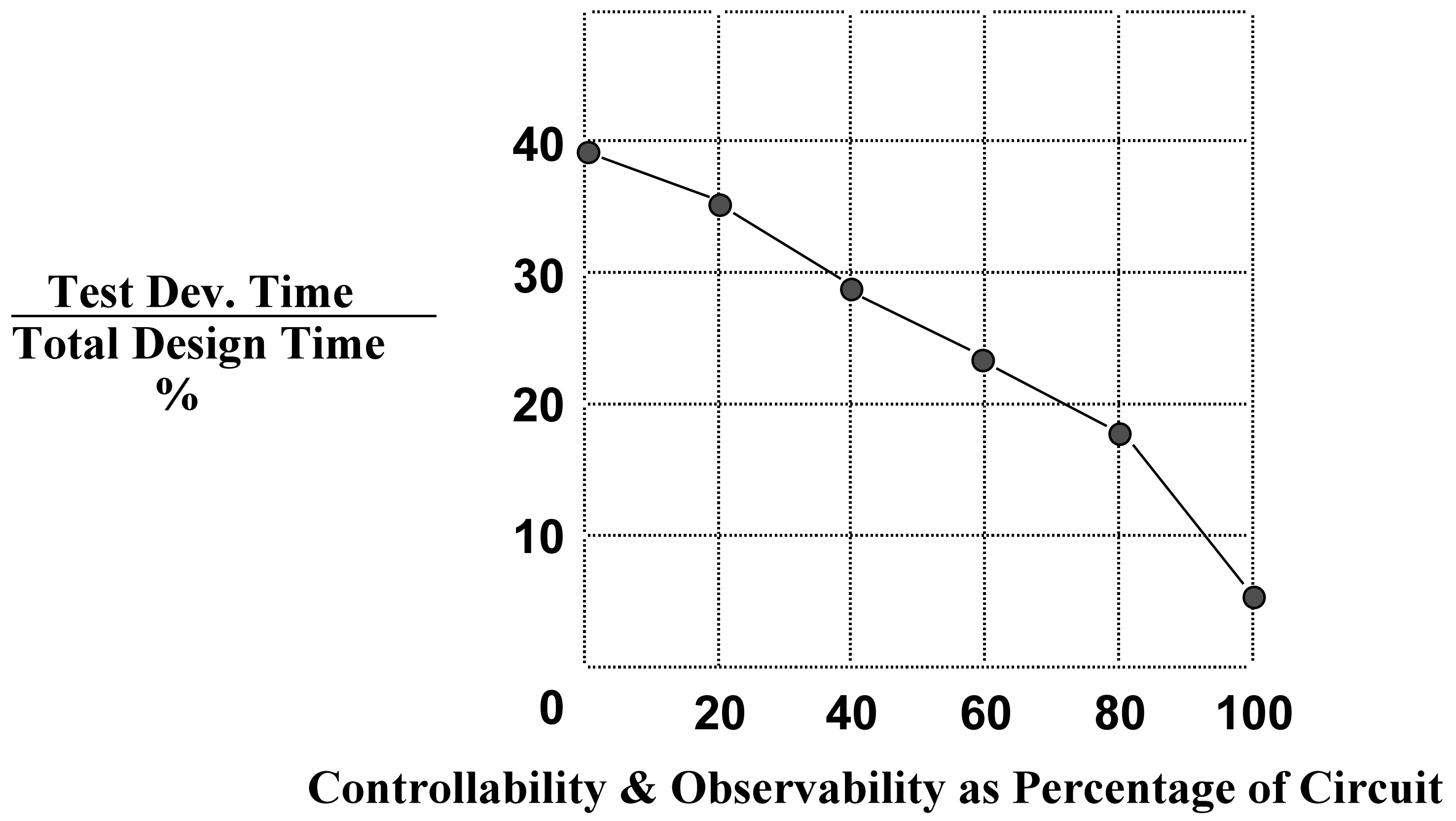
Enabling Productivity

Presentation:

HDL for Synthesis Guidelines

Why Design for Test?

Enabling Productivity



Source of Graph: “ASIC Testing Upgraded”, by Marc Levitt, IEEE Spectrum, May 1992, pp26-29

Use Synchronous Design Style

Enabling Productivity

- Avoid One Shots
- Avoid Asynchronous State Machines
- Isolate Asynchronous Logic

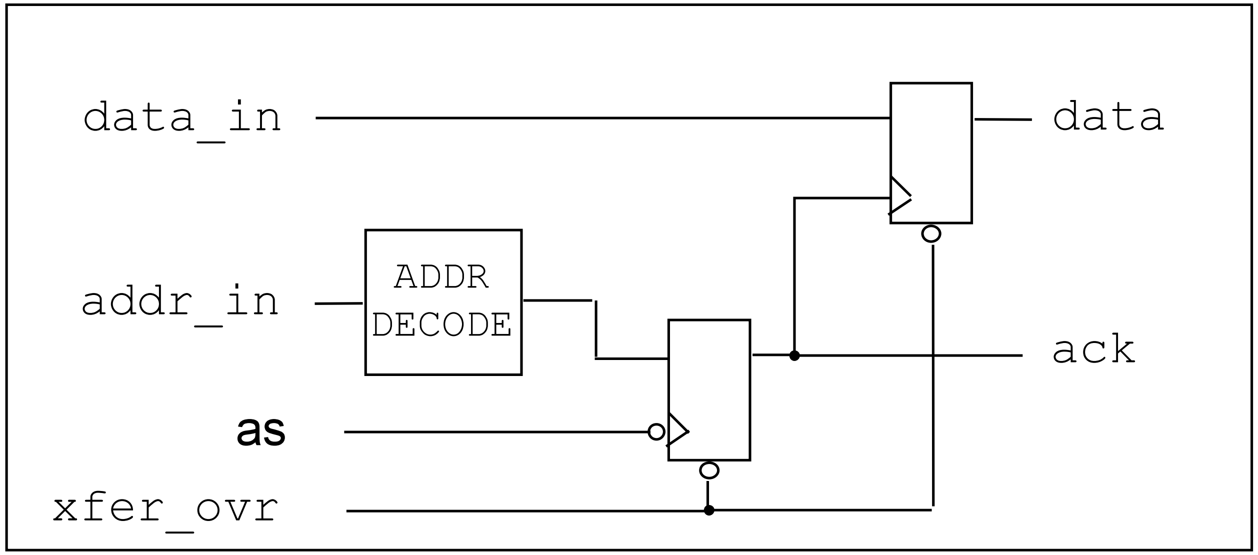
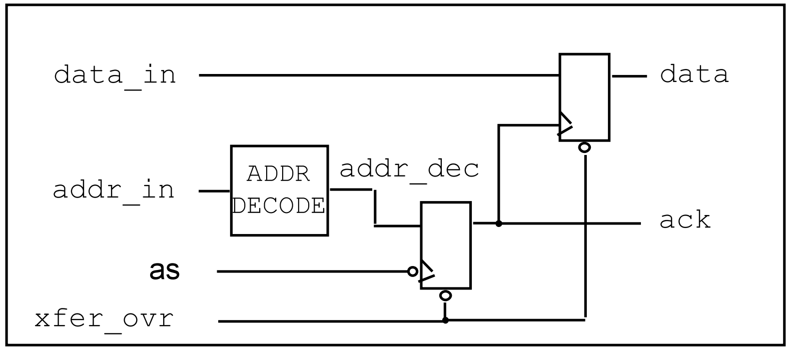


Fig. 1 Example Asynchronous Bus Interface

Exercise: Make Asyn. Xface Testable

```
module bus_xface (data_in, addr_in, as, xfer_ovr, ack,data);
input [3:0] data_in;
input [3:0] addr_in;
input as, xfer_ovr;
output [3:0] data;
output ack;

reg [3:0] data;
reg ack;

always @ (negedge as or negedge xfer_over)
  if(~xfer_over)
    ack <= 1'b0;
  else
    ack <= addr_dec;

ADDR_DECODE U1(addr_in, addr_dec);

always @ (posedge ack or negedge xfer_over)
  if(~xfer_over)
    data <= 4'b0000;
  else
    data <= data_in;
endmodule
```



Bypass Internally Created Clock

Enabling Productivity

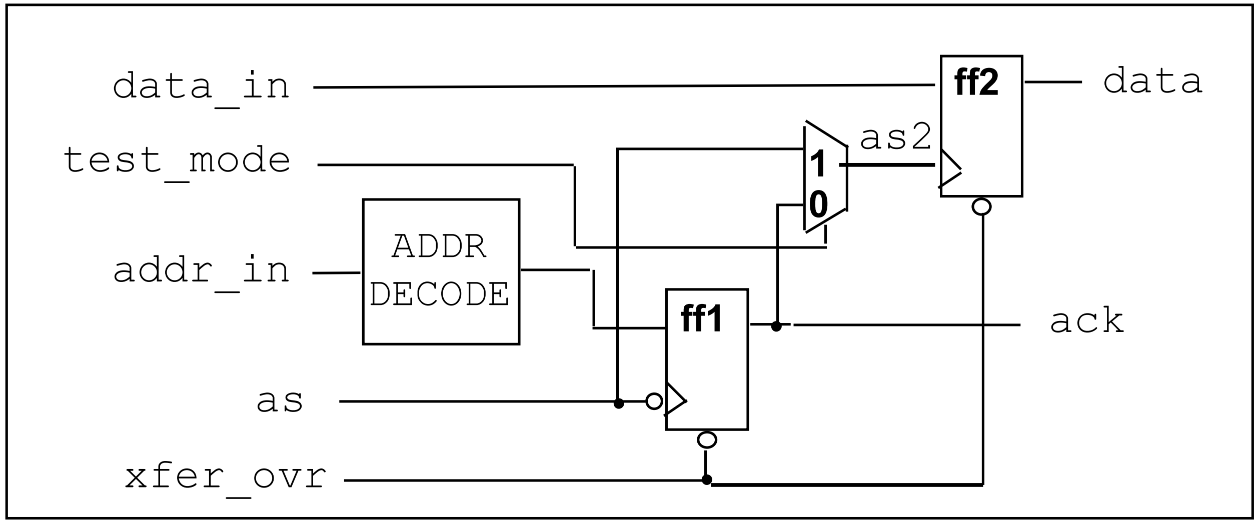


Fig 2. Testable "Asynchronous" Xface with Controlled Clock Ckt.

- Internal clock is not controllable: bypass it during test.
- Two phases of clock used: route scan chain ff1 to ff2 to prevent "shoot thru".
- Assume "`as`" is available at chip I/O and synchronous relative to other clocks during test.

Verilog Code for Bypassed Asyn Ckt

Enabling Productivity

```
module bus_xface (data_in, addr_in, as, xfer_ovr, ack,data,test_mode);
input [3:0] data_in;
input [3:0] addr_in;
input as, xfer_ovr; test_mode;
output [3:0] data;
output ack;

reg addr_dec;
reg [3:0] data;
reg ack;
wire as2;

assign as2 = test_mode ? as : ack;

always @ (negedge as or negedge xfer_over)
  if(~xfer_over)
    ack <= 1'b0;
  else
    ack <= addr_dec;

ADDR_DECODE U1(addr_in, addr_dec);

always @ (posedge a or negedge xfer_over)
  if(~xfer_over)
    data <= 4'b0000;
  else
    data <= data_in;
endmodule
```

Avoid Internal Three State Buses

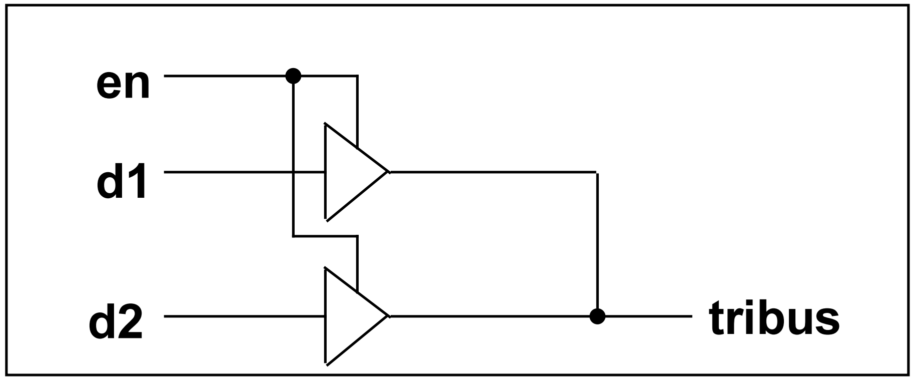


Fig. 3 Example Three-State Circuit

Rule: Cannot have multiple drivers active at the same time.

- Can potentially cause bus contention and a power sink, if the values driven by the drivers are different.

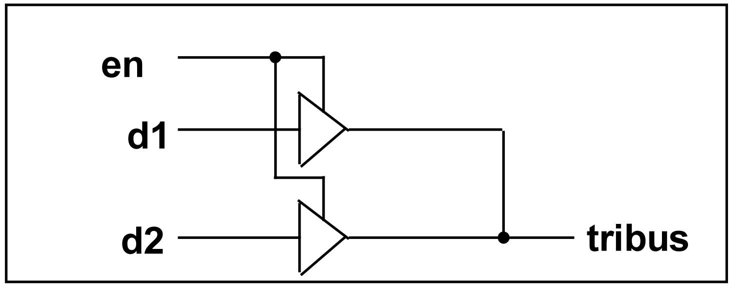
Rule: Must have at least one driver active at all times.

- Cannot test enable signal, if disabling drivers causes bus to float.

Exercise: Make 3 States Testable

Enabling Productivity

```
module tri_state (en,d1,d2,tribus);
input d1, d2;
input [1:0] en;
output tribus;
reg tribus;

    always @(d1 or en)
        if (en)
            tribus = d1;
        else
            tribus = 1'bz;
    always @(d2 or en)
        if (en)
            tribus = d2;
        else
            tribus = 1'bz
endmodule
```



Use Pull-Ups & Muxed Enables

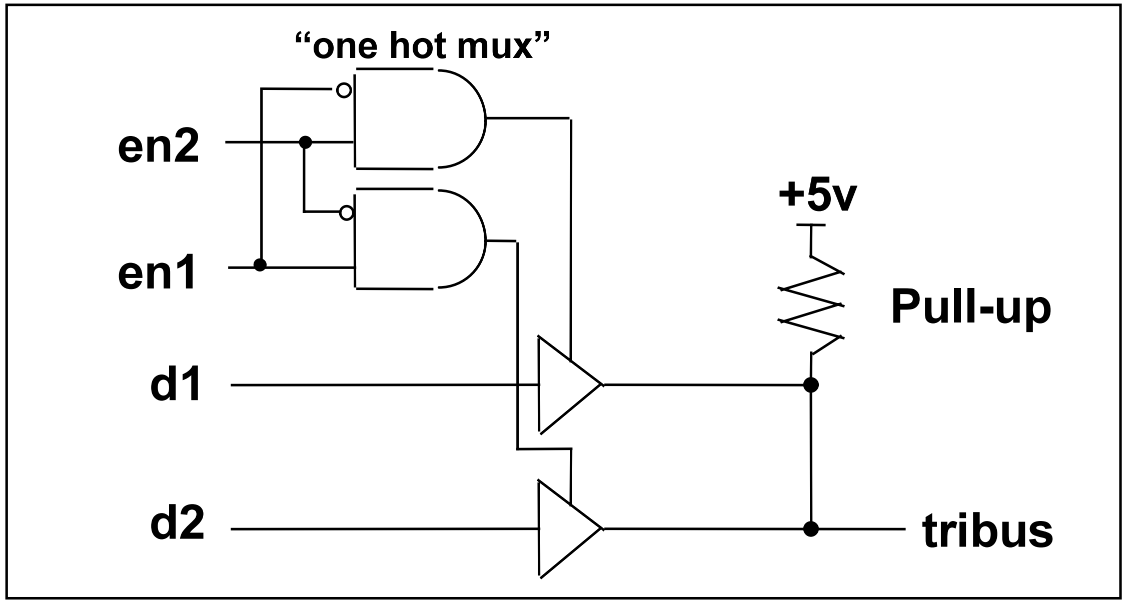


Fig. 4 Example Three-State w/ Pull-ups & Multiplexed Enables

- All tri-state controls are preferably fully decoded to ensure one active driver

Verilog Code for Pull-ups & 3-States

```
module tri_state (en1, en2,d1,d2,tribus);
input d1, d2;
input en1,en2;
output tribus;
reg tribus;

always @(d1 or en1 or en2)
      if (en1 & ~en2)
         tribus = d2;
      else
         tribus = 1’bz;

always @(d2 or en1 or en2)
     if (~en1 & en2)
        tribus = d1;
     else
        tribus = 1’bz;

pullup (tribus); // not synthesizable

endmodule
```

- Pull-ups can ONLY be instantiated, not inferred in Verilog code.

Use Mux Instead of Three -State

Enabling Productivity

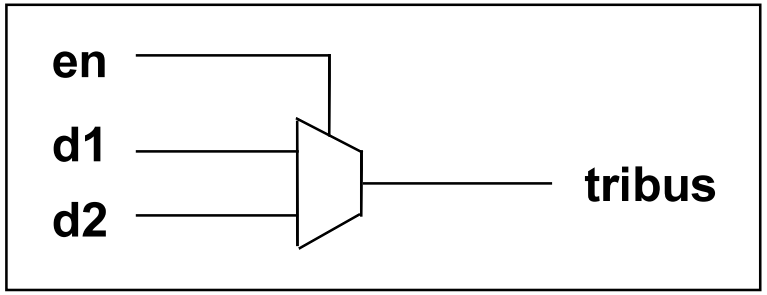


Fig. 5 Example Multiplexed Bus

- Simpler to code.
- No possibility of bus contention

```
module mux_example (en,d1,d2,tribus);
input d1, d2;
input en;
output tribus;
reg tribus;

always @(d1 or d1 or en)
      if (en)
         tribus = d1;
      else
         tribus = d2;
endmodule
```

Avoid Uncontrollable Clocks

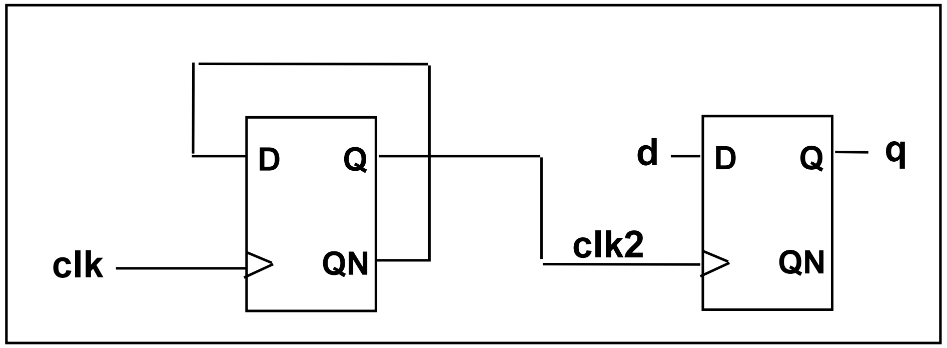


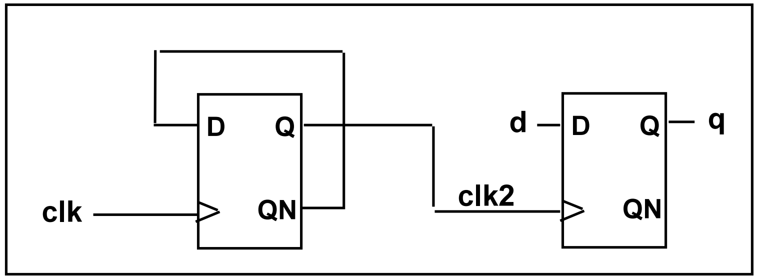
Fig. 6 Example Clock Divider Circuit

Rule: All clocks must be controllable and accessible from top level ports.

- Chip tester would require multiple tester cycles per serial scan chain data item.
- Clock dividers inherently untestable, belong to asynchronous circuit category.

Exercise: Make Clk2 Controllable

```
module clk_gen (d, clk, q);
input clk, d;
output q;

reg clk2;

always @ (posedge clk)
  clk2 <= ~clk2;

always @ (posedge clk2)
  q <= d;

endmodule
```



Bypass Bad Clocks During Test

Enabling Productivity

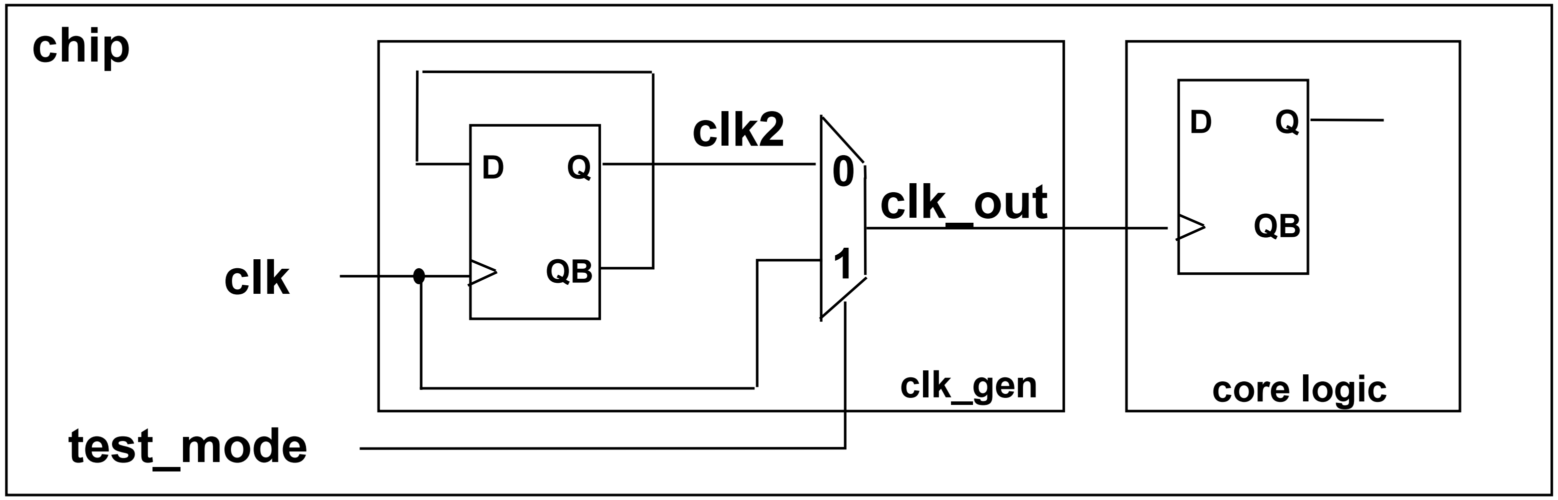


Fig. 7 Example Asynchronous Clock Generator Bypass

- Bypass circuitry added to source code by <u>designer</u>.
- During test: use chip level clock, during regular operation: use derived clock.
- TEST_MODE signal is active high during test, and requires a dedicated port.

Verilog Code for Clock Bypass Logic

Enabling Productivity

- General Rule: Isolate clock generation circuit into its own level of hierarchy.

```
module clk_gen (d, clk, test_mode, q,clk_out);
input clk, d, test_mode;
output q,clk_out;

reg clk2,clk_out;

always @ (posedge clk)
  clk2 <= ~clk2;

always @ (posedge clk_out)
  q <= d;

always @ (test_mode or clk or clk2)
  if (test_mode)
   clk_out = clk;
  else
   clk_out = clk2;

endmodule
```

```
module chip (clk, test_mode, data_in, instr_in, data_out);
input clk, test_mode;
input [31:0] data_in;
input[7:0] instr_in;
output [15:0] data_out;

wire [15:0] data_out;
wire clk_inner;

clk_gen clk_gen_0(clk,test_mode,clk_inner);

core_logic core_logic_0(clk_inner, data_in, instr_in,data_out);

endmodule
```

Avoid Using Clocks as Data Inputs

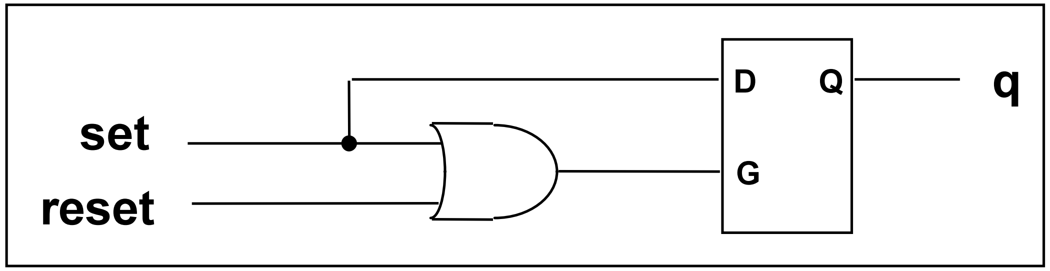


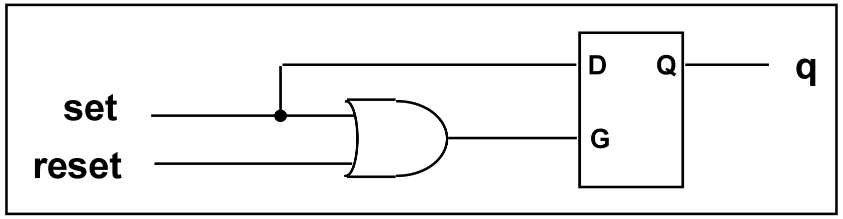
Fig. 8 Example Set - Reset Latch

Rule: DO NOT Use Clocks as Data Inputs.

- Race condition could exist between the enable and data of the latch.
- Even if race condition fixed - it’s very difficult to detect and correct such problems.
- Falls under category of asynchronous logic.

Exercise: Make Clock Testable

```
module s_r_latch(set, reset, q);
input set, reset;
output q

reg q;

always @ (set or reset)
  if(set |reset)
    q <= set;

endmodule
```



Use a S-R Flip-flop

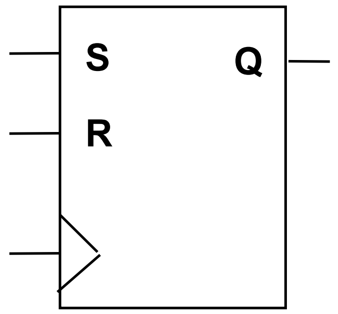


Fig. 9 Example S-R FF

```
module s_r_ff (  set, reset, clk, q);
input set, reset, clk;
output q;

reg q;

// synopsys sync_set_reset “set, reset”
always @ (posedge clk)
  if (set)
      q <= 1’b1;
  else if (reset)
      q <= 1’b0;

endmodule
```

Uncontrollable Asynchronous Resets

Enabling Productivity

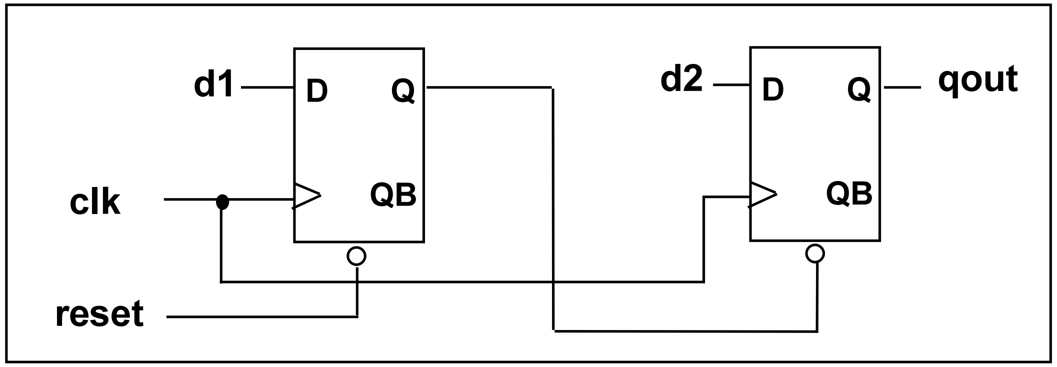


Fig. 10 Example Uncontrollable Reset During Test

Rule: All asynchronous reset / set signals should be controllable through a chip level port.

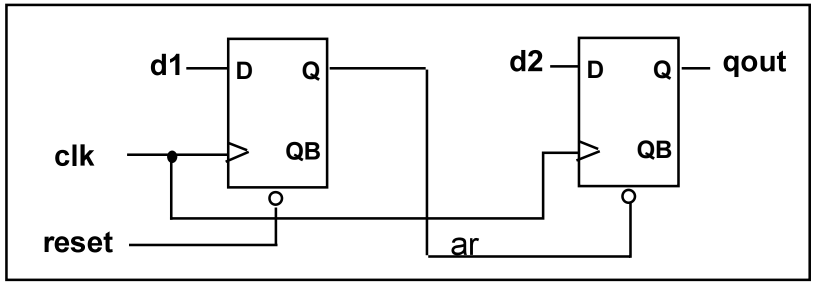
- The integrity of data scanned through the register during scan shifting must be upheld.

- Uncontrolled reset/set signals could overwrite/erase parts of the scan chain data

Exercise: Fix Uncontrollable Reset

Enabling Productivity

```
module asyn_reset (clk,d1,d2, reset,qout);
input d1, d2, clk, reset;
output qout;

reg ar;

always @ (posedge clk or negedge reset)
if (~reset)
  ar <= 1'b0;
else
  ar <= d1;

always @ (posedge clk or negedge ar)
if (~ar)
   qout <= 1'b0;
else
   qout <= d2;

endmodule
```



Controlling Asynchronous Resets

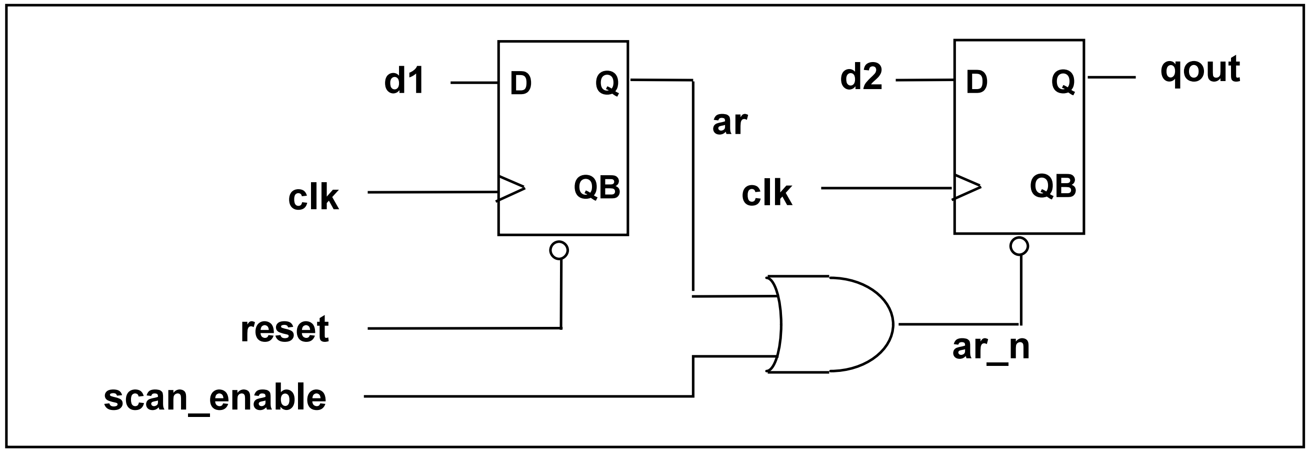


Fig. 11 Example Test Override of Asynchronous Reset

- During scan shift operation, reset held inactive, scan data integrity upheld.
- During capture cycle of test, scan_enable is low and thus asynchronous reset signal can be tested.

Vevilog Code to Control Asynch. Reset

Enabling Productivity

- Internal reset signal “gated” within asynchronous reset description

```
module asyn_reset (clk, d1, d2, reset, scan_enable, qout);
input d1, d2, clk, reset, scan_enable;
output qout;

reg ar;

always @ (posedge clk or negedge reset)
if (~reset)
  ar <= 1’b0;
else
  ar <= d1;

wire ar_n = ar | scan_enable;

always @ (posedge clk or negedge ar_n)
if (~ar_n)
    qout <= 1’b0;
else
    qout <= d2;

endmodule
```

Testable Reset Synchronizer

Enabling Productivity

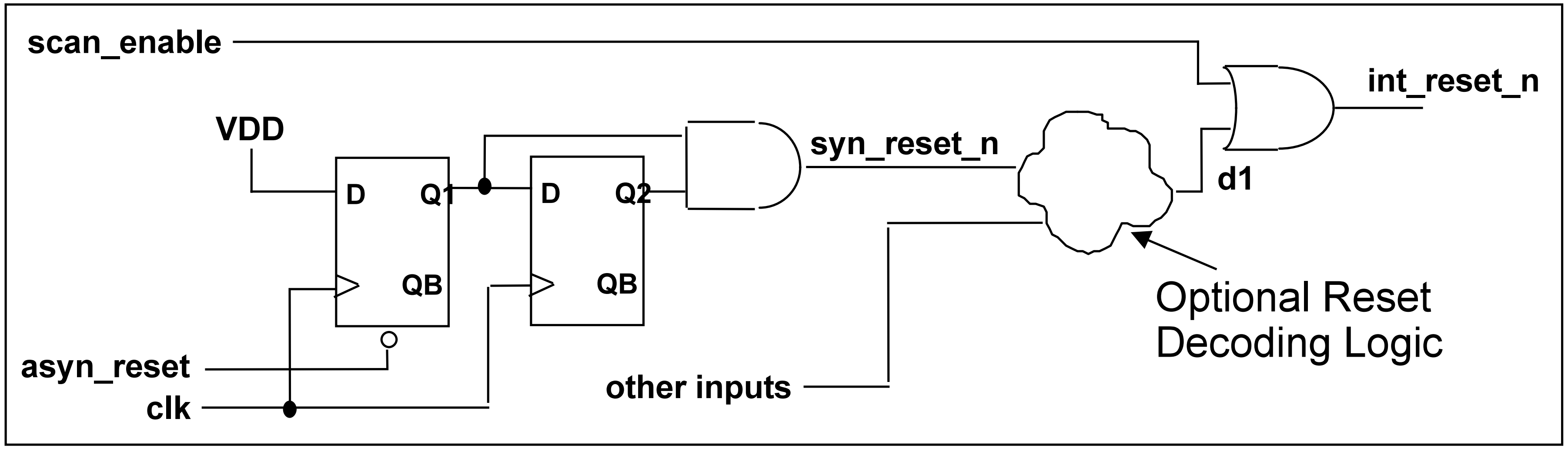


Fig. 12 Example Testable Reset Synchronizer Circuit

- Extension of controlling asynchronous signals scheme.
- The synchronized reset signal, *syn_reset_n* can be combined with other inputs if necessary, then gated with *scan_enable* .
- Don't forget to identify the scan_enable port as part of your test circuitry.

Verilog Code for Reset Synchronizer

Enabling Productivity

```
Module syn_reset(clk,asyn_reset,scan_enable, int_reset_n);
input clk, asyn_reset, scan_enable;
output int_reset_n;

reg q1, q2;

always @ (posedge clk or negedge asyn_reset)
  if (~asyn_reset)
    q1 <= 1'b0;
  else
    q1 <= 1'b1;

….
….
….

always @ (posedge clk)
  q2 <= q1;

wire int_reset_n = scan_enable | d1;

endmodule
```

Avoid Combinational Feedback Loops

Enabling Productivity

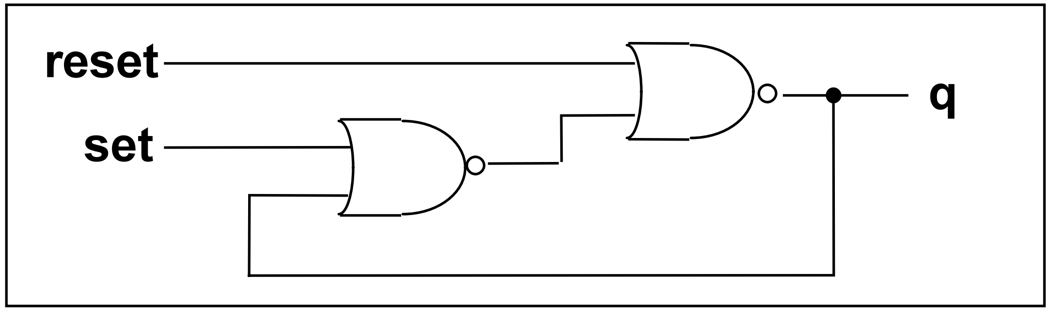


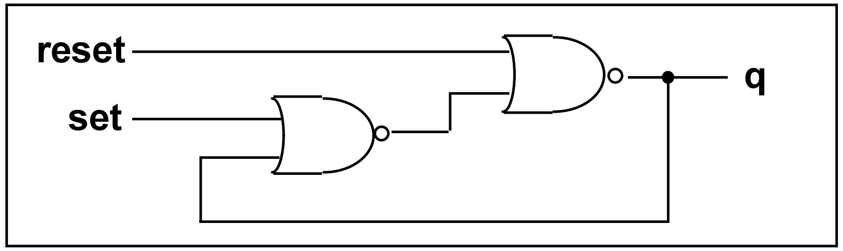
Fig. 13 Example S-R Latch

Rule: Do not have combinational feedback loops in design.

- Introduces states in the design which cannot be synchronously controlled.
- Faults within the logic of the combinational feedback loop may not be testable.
- Asynchronous feedback loops cause problems with synthesis.

Exercise: Make S-R Latch Testable

Enabling Productivity



```
module s_r_latch (  set, reset, q);
input set, reset;
output q;

wire tmp;

nor (q, reset, tmp);
nor (tmp, set, q);

endmodule
```

Model S-R Latch as a Leaf Cell

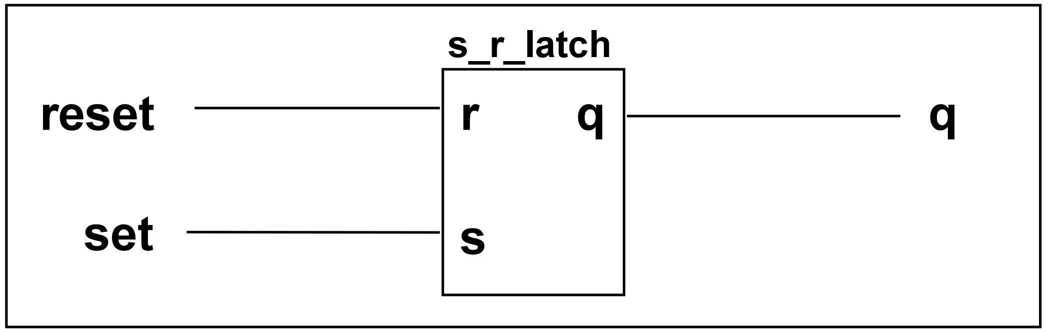


Fig.14 Example s_r_latch as a leaf cell

- If SR Latch is modelled as a leaf cell in the technology library, then internal feedback loop not visible to the test generation software.

- Should not be a problem IF a scannable equivalent of the cell exists in the library.

- Otherwise must treat latch as you treat other latches in your design (latches discussed in Test Schemes).

Bi-di Pads Introduce Feedback Loops

Enabling Productivity

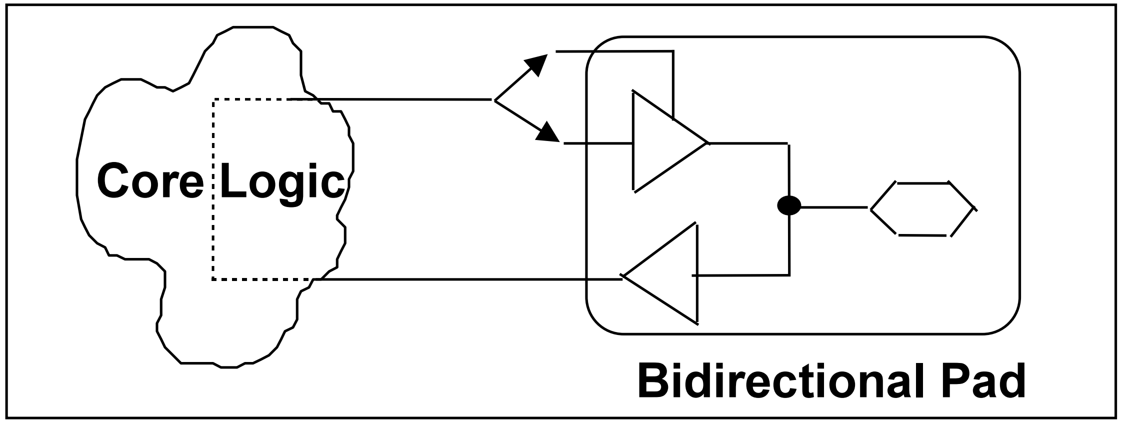


Fig. 15 Example of Feedback Loop w/Bi-Directional Pad

- Combinational feedback path might exist from the input driver thru the internal core logic to the enable or data pin of the output driver.
- Feedback is evident only when the pad is in “output mode”.
- Feedback may or may not be detected by test design rule checker.
- Feedback loop might have to be explicitly broken. Best practice: eliminate loops altogether from design.

Design for Test Rules Summary

Enabling Productivity

- Use Synchronous design styles.
- Avoid Asynchronous designs
- Asynchronous: Isolate Asynchronous logic
- Avoid Three-State Drivers
- 3States: Cannot have more than one driver active at a time
- 3States: Must have at least one driver active at all times (or use a pullup)
- All clocks must be controllable and accessible from top level ports.
- Do NOT use clocks as data inputs.
- All asynchronous reset/set signals must be controllable thru top level port
- Do not have combinational feedback loops in the design.

Test Schemes

Enabling Productivity

- Latches in Flip-flop Based Designs
- Improving Control & Observability
- Techniques for Testing RAMs

Latches in Flip-flop Based Designs

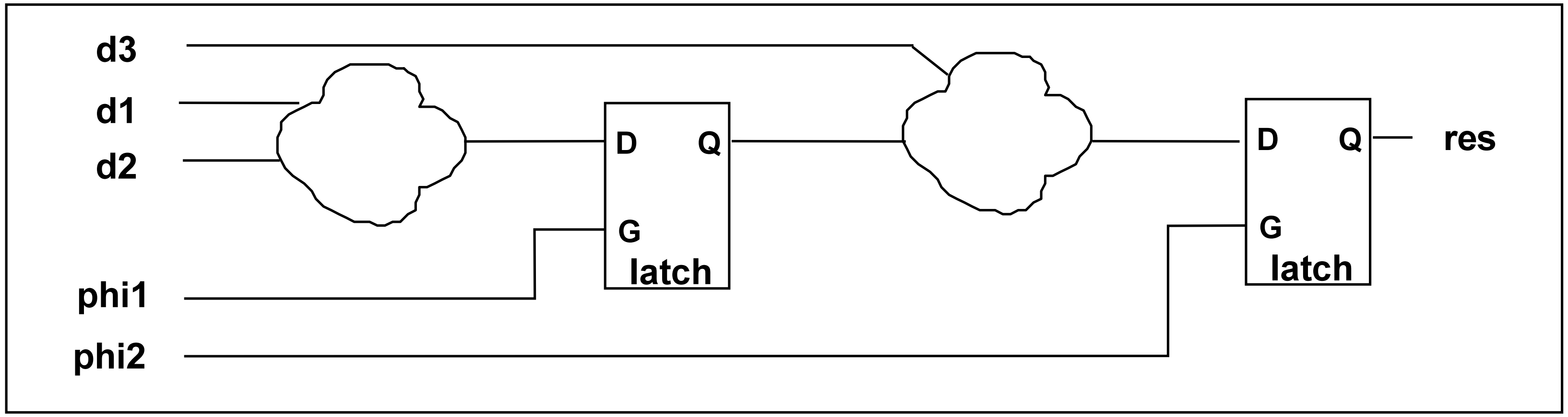


Fig. 16 Example Latches in a Design

You can:

- Leave them as is - let test tool deal with them.
- Replace them with a scannable equivalent (for example use a LSSD cell).
- Model them as black boxes, with the resultant loss of fault coverage
- Hold them transparent during test (watch out for combinational feedback loops!)

Making Latches Transparent

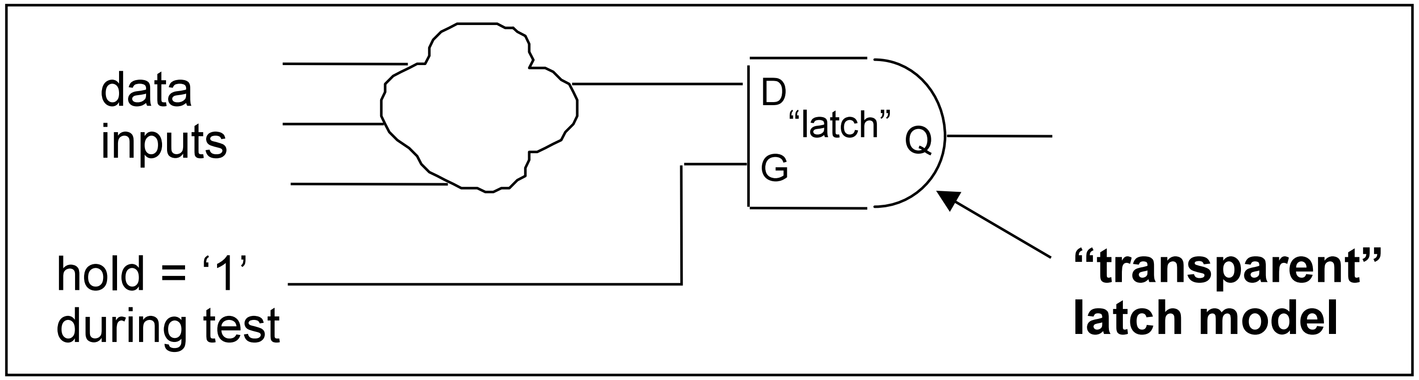


Fig. 17 Example Transparent Model for a Latch

- Latch is modelled as a combinational circuit which represents a latch in active (pass thru) mode.
- Lose some fault coverage on enable pin of latch.
- Treat enable pin as data, do not "hook" it up to clock source.
- This treatment of latches is tool specific (ie. beyond Verilog coding style).

Improving Control & Observability

Enabling Productivity

- Given: Scan chains improve testability by providing access to internal registers.
- You can make a design more observable and/or controllable and thus better for test by adding flip-flops at crucial points.

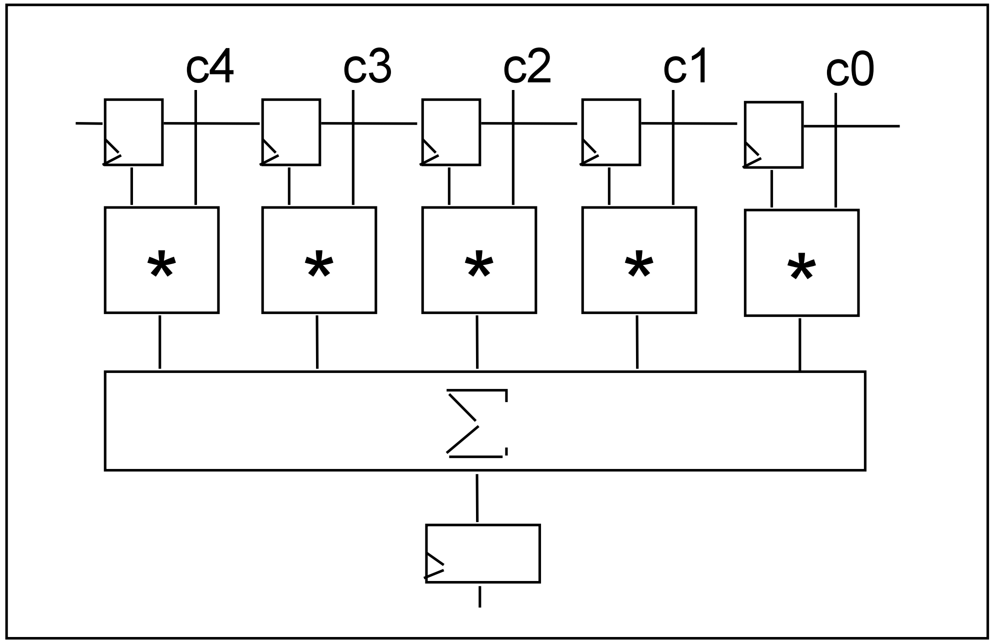


Fig.18 Example “Difficult to Test” Circuit

Adding FF's to Improve Controllability

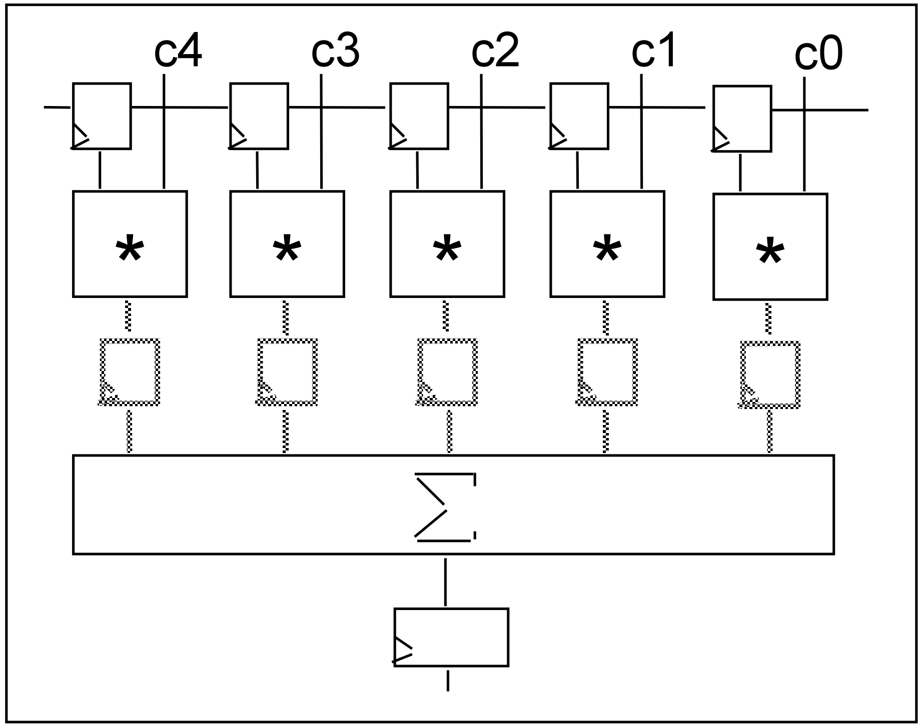


Fig.19 Example Design with Improved Testability

- Simplifies the timing constraints of the design (increases latency, decreases critical paths).

- Can test adder without adding primary input/outputs to the multipliers.

Partitioning to Improve Testability

Rule: Do not allow hierarchical boundaries in combinational paths.

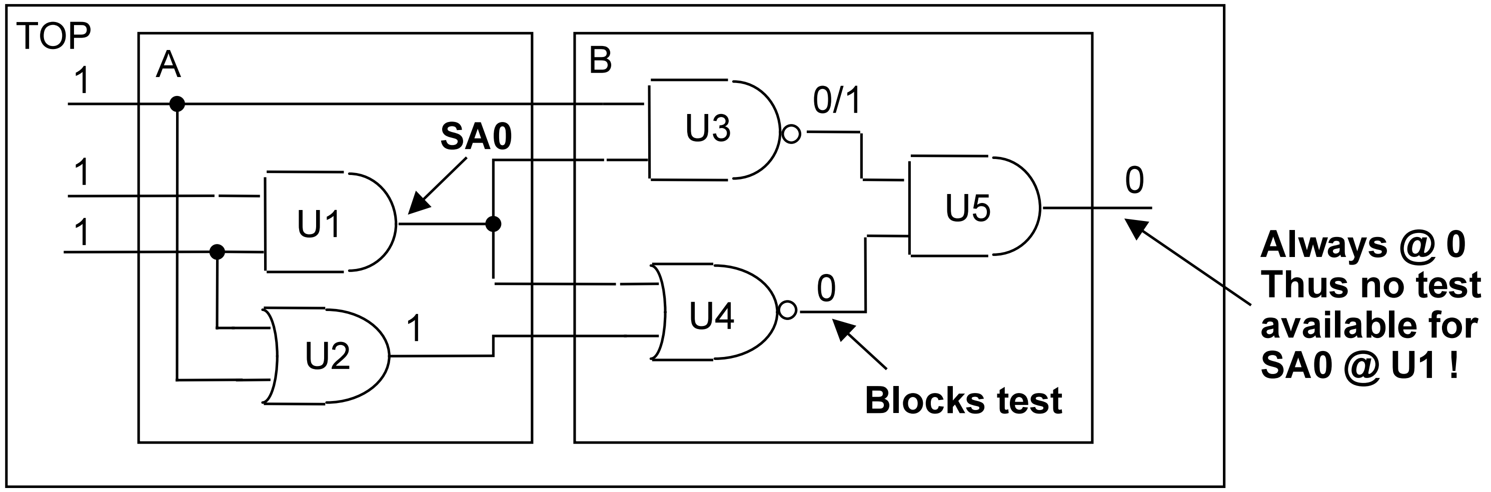


Fig. 20 Testing for a Stuck-At-0 Fault with a Reconvergent Fanout Design

Reconvergent fanout : different paths from the same signal converge again at the same component downstream in the logic.

Techniques for Testing RAMs

- Multiplexed I/O
- Register Bounding
- Transparent RAMs
- Built-In-Self-Test (BIST)

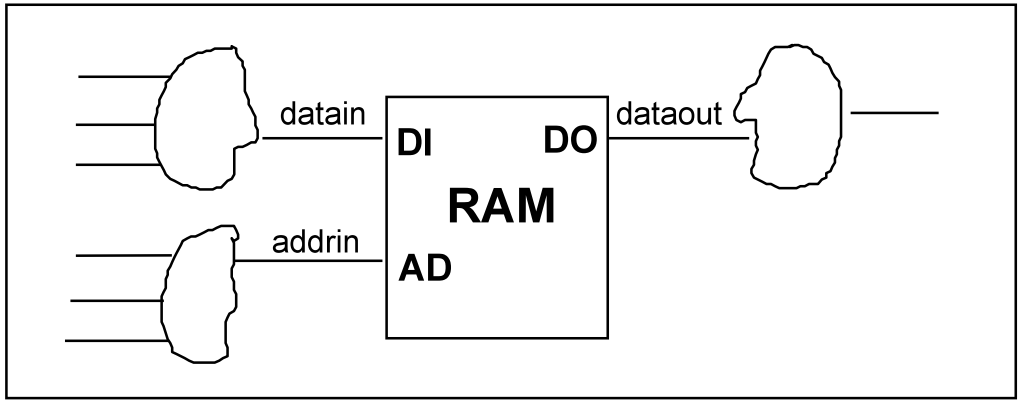


Fig.21 Example RAM with surrounding logic

Multiplexed I/O for RAMs

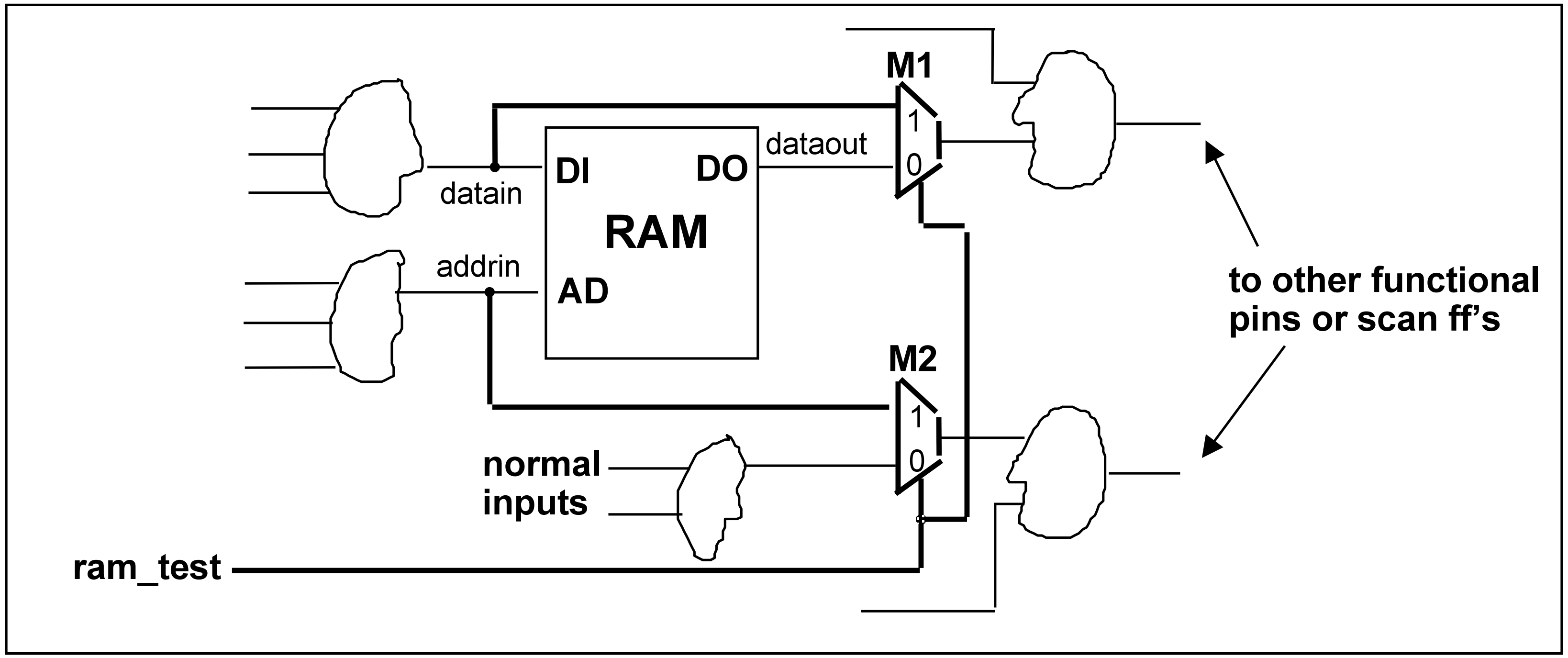


Fig.22 Example Multiplexed I/O Test Scheme applied

Multiplexed I/O Test Scheme

- Increases the observability of the data input to the RAM.
- Increases the controllability of the data output by the RAM.
- Increases the observability of the address driving the RAM.
- If the RAM has a known state during test, then output mux is not needed.

Register Bounding for RAMs

Enabling Productivity

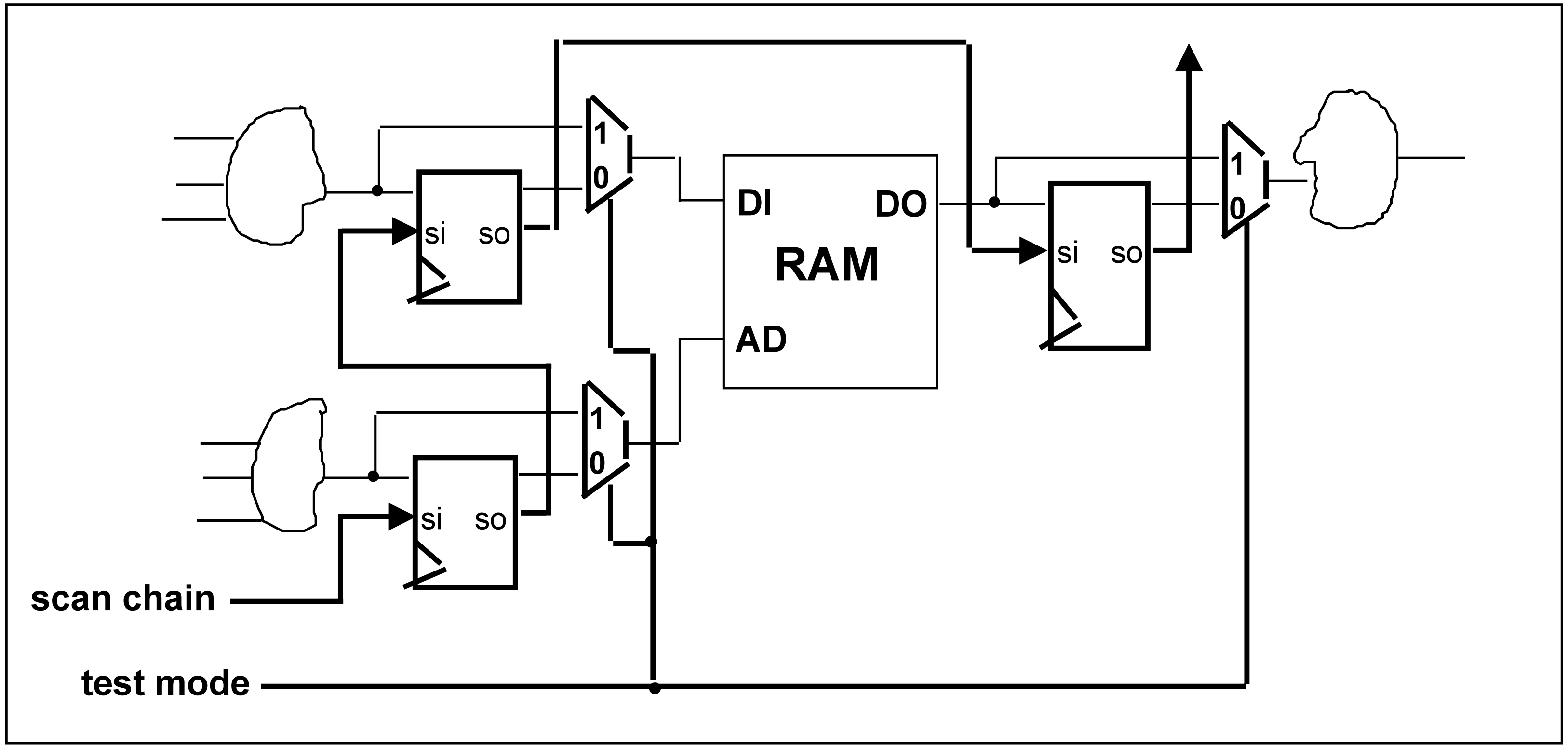


Fig.23 Example Register Bounding Scheme applied to RAM

Register Bounding Scheme

- In normal mode, the bounding registers are bypassed.
- The rest of the ASIC is isolated from the memory and can be tested independently.
- Bounding registers can be used to access & test RAM array.
- All memory arrays to be tested are usually connected into one scan chain.
- Not appropriate for memory arrays more than 1K words.
- Muxes only necessary if combo logic in between internal registers and RAM pins.
- RAM read/write control might need to be controlled by scan_enable signal.

Transparent RAMs

Enabling Productivity

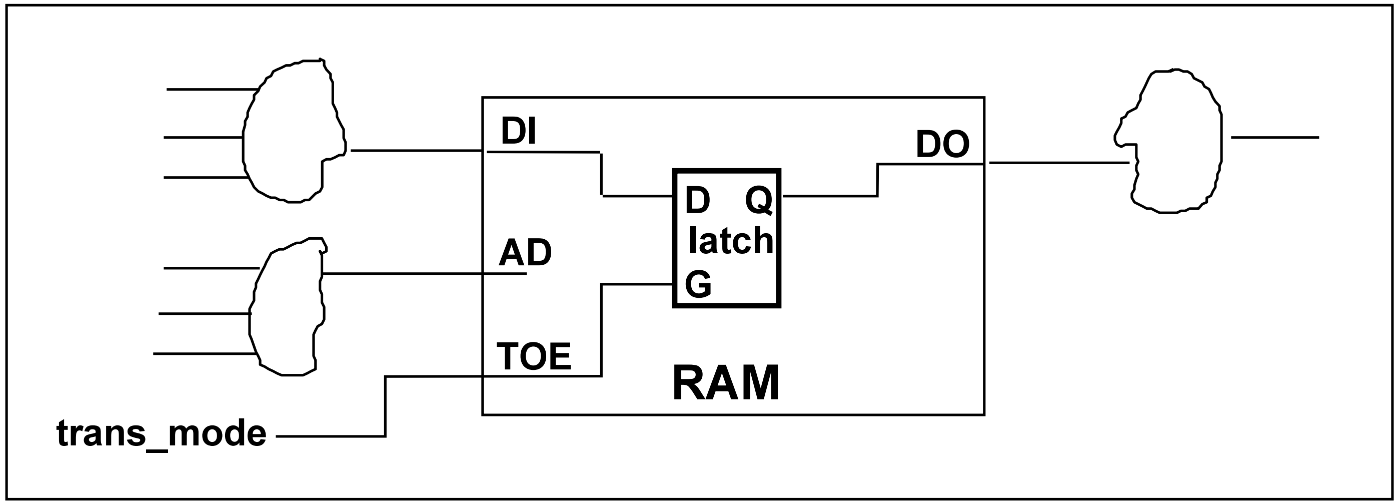


Fig.24 Example Transparent RAM

Transparent RAM Scheme

- Treats the RAM array as if it consists of an array of latches.
- Meant to allow observability of data inputs at data output.
- Must have technology library support from vendor.
- Does not test address inputs or RAM array.

Comparison of RAM Test Schemes

Enabling Productivity

Test Method	Coverage & Ease	Implementation Limitations	Comments
Multiplexed I/O	Easiest Tests Logic Only	Adds mux delay to paths	
Register Bounding	Tests Logic & RAM	Mux if combo logic; Arrays<1K words	
Transparent Mode	No Additional Muxes Tests Logic Only	Additional Test Protocol	Limited ASIC Library Support
BIST	Tests RAM only Comprehensive	Additional real-estate	Limited by available S/W tools

Table 1 Comparison of RAM Test Schemes

Recommended Test Schemes for RAMs

• If the address, datain, and dataout pins of the RAM are connected directly to internal registers with no combinational logic in between, then *use register bounding* (ie. make surrounding registers part of a scan chain).

• If combinational logic exists between internal registers and the address & data pins of the RAM, consider the following in the order given:

Small / Medium Arrays	**Large Arrays**
Register Bounding	BIST
Multiplexed I/O	Register Bounding
BIST	Multiplexed I/O

Table 2: Test Schemes For Combo Logic Surrounding RAM

Test Scheme Summary

Enabling Productivity

- Hold Latches in transparent mode.
- Add flip-flops to a design to increase controllability & observability
- Do not partition a combinational logic path across hierarchical boundaries.
- Use register bounding to test RAMs (as a first choice).

Test Methodology Summary

- Test is a *design methodology*. It has it's own testability rules, created to insure that scan chains can be added to a design, with the ultimate goal of using an Automated Test Pattern Generator (ATPG) to create test patterns for the chip.

- Most problems associated with test can be avoided or anticipated and corrected up front, during the INITIAL synthesis of the source Verilog code to gates.

HDL for Synthesis Guidelines

Presentation:

HDL for Synthesis Guidelines

- ❑ General HDL Code Structure
- ❑ Partitioning
- ❑ Implying Logic Structure
- ❑ Safe Coding & Avoiding Problems
- ❑ Source Code Readability
- ❑ Coding Style for Design Reuse
- ❑ Design for Testability
- ■ **Practices**

Practices

- ❑ Unsupported Verilog Language Constructs
- ❑ Limitations of Blocking and Non-blocking Assignments
- ❑ Limitations of D Flip-Flop Inferences
- ❑ while Loops Limitations
- ❑ forever Loops Limitations
- ❑ Handling Comparisons to X and Z
- ❑ Limitations of Using Delay Specification
- ❑ Limitations of Tri-State Inferences
- ❑ Limitations of Arithmetic Operators
- ❑ Limitations of casex and casez Statement
- ❑ Case Statement usage
- ❑ Register Inferring

Unsupported Verilog Language Constructs

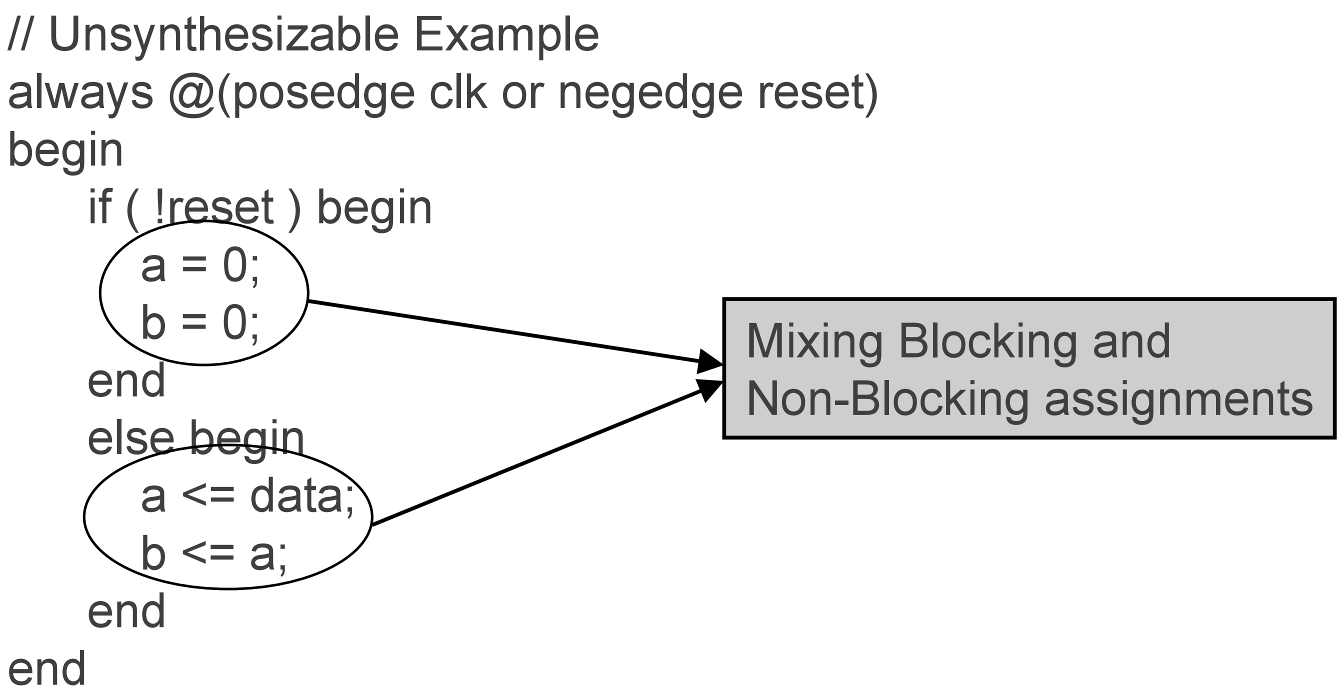
- **Unsupported Definitions and Declarations**
  - time declaration
  - event declaration
  - triand, trior, tri1, tri0, and trireg net types
  - Ranges and arrays for integers
- **Unsupported operators**
  - Case equality and inequality operators (=== and !==)
  - Division and modules operators for variables
- **Unsupported gate-level constructs**
  - nmos, pmos, cmos, rpmos, rcmos, pullup, pulldown, tranif0, tranif1, rtran, rtranif0, and rtranif1 gate types

Unsupported Verilog Language Constructs

- **Unsupported Statements**
  - defparam statement
  - initial statement
  - repeat statement
  - delay control
  - event control
  - wait statement
  - fork statement
  - deassign statement
  - force statement
  - release statement
  - procedural continuous assignment

Limitations of Blocking and Non-blocking Assignments

- **A variable can follow only one assignment method and cannot be the target of both Blocking and Non-blocking assignments.**

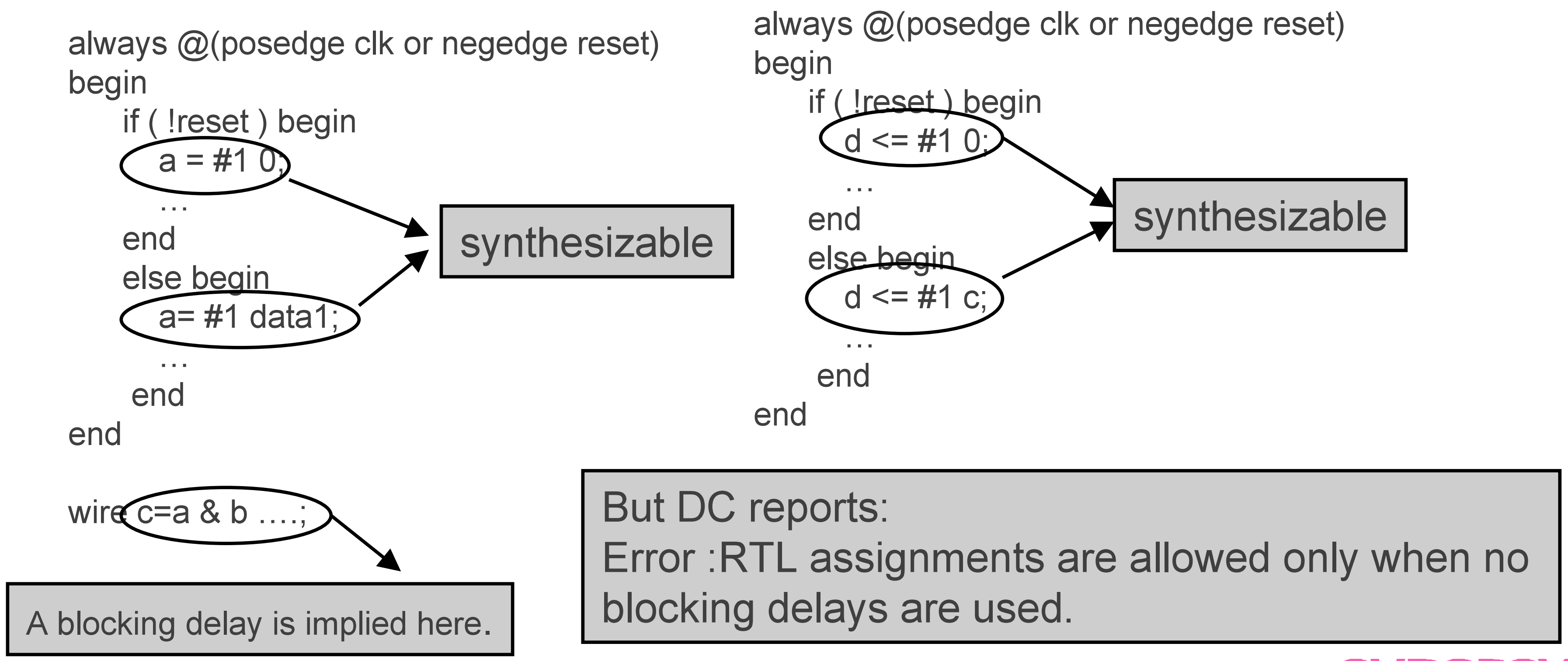


```
// Unsynthesizable Example
always @(posedge clk or negedge reset)
begin
      if ( !reset ) begin
          a = 0;
          b = 0;
      end
      else begin
          a <= data;
          b <= a;
      end
end
```

Limitations of Blocking and Non-blocking Assignments

Enabling Productivity

- RTL assignments are allowed only when no blocking delays are used
  - ☞ #1 a<=b ---- unsythesizable
- If variables in two always blocks having dependencies are used in mixing Blocking and Non-blocking statements, they might case the design unsynthesizable.



```
always @(posedge clk or negedge reset)
begin
      if ( !reset ) begin
         a = #1 0;
         …
      end
      else begin
         a= #1 data1;
         …
       end
end

wire c=a & b ….;
```

synthesizable

A blocking delay is implied here.

```
always @(posedge clk or negedge reset)
begin
      if ( !reset ) begin
         d <= #1 0;
         …
      end
      else begin
         d <= #1 c;
         …
       end
end
```

synthesizable

But DC reports:
Error :RTL assignments are allowed only when no blocking delays are used.

Limitations of D Flip-Flop Inferences

Enabling Productivity

- ❑ **The signal in an edge expression cannot be an indexed expression.**
- ❑ **Set and reset conditions must be single-bit variables.**
- ❑ **Set and reset conditions cannot use complex expressions.**
- ❑ **An if statement must occur at the top level of the always block.**

Limitations of D Flip-Flop Inferences

```
// Unsynthesizable Examples

always @(posedge clk[1])

always @(posedge clk and negedge reset_bus)
     if (!reset_bus[1])

always @(posedge clk and negedge reset)
     if (reset == (1-1))

always @(posedge clk or posedge reset) begin
     # 1;
     if (reset)
```

Invalid: it uses an indexed expression

Invalid: it uses a bused variable

Invalid: it uses a complex expression

Invalid: the if statements does not occur at the top level

while Loops Limitations

- **A while loop creates a conditional branch that must be broken by one of the following statements to prevent combinational feedback.**
  - **@(posedge clock)**
  - **@(negedge clock)**

```
// Unsupported while loop
always
      while ( x < y )
           x = x + z;
```

```
// Supported while loop
always begin
      begin @(posedge clock)
            while ( x < y )
             begin
                 @(posedge clock)
                 x = x + z;
              end
       end
```

forever Loops Limitations

- **Infinite loops in Verilog use the keyword forever.**
- **You must break up an infinite loop with the following statements to prevent combinational feedback.**
  - ☞ **@(posedge clock)**
  - ☞ **@(negedge clock)**

```
//Supported forever Loop
always
    forever
    begin
        @(posedge clock);
        x = x + z;
    end
```

Handling Comparison to X and Z

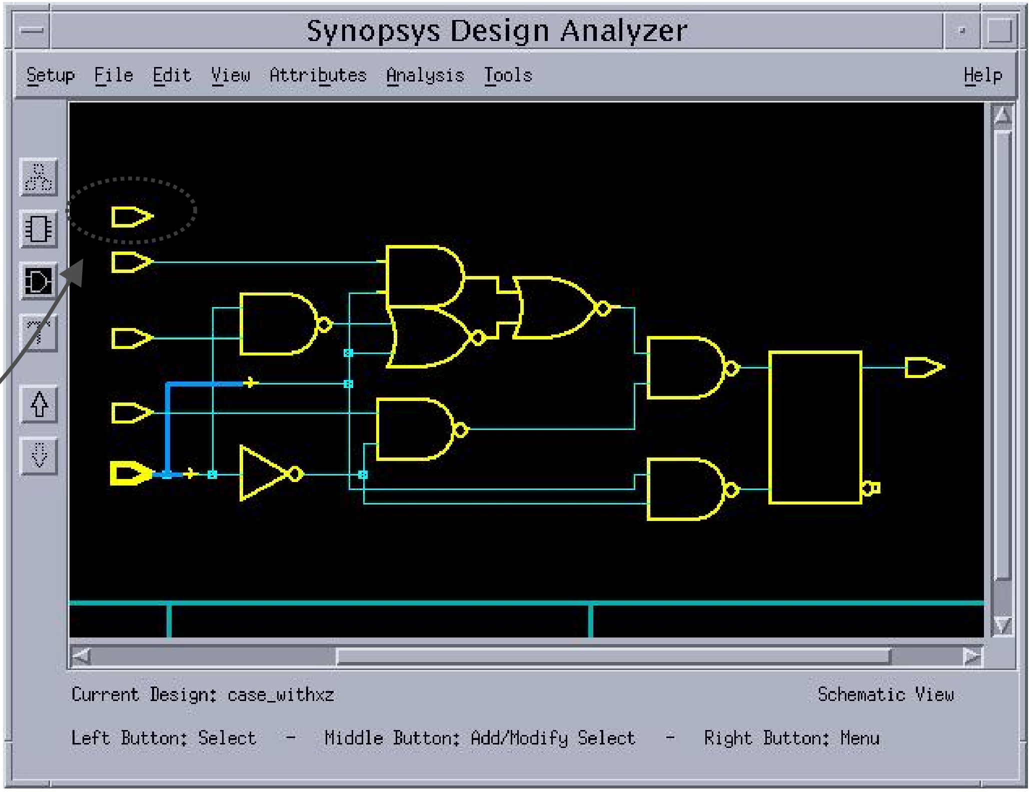
- **Comparison to an X or a Z, a warning message is displayed in DC indicating that the comparison always evaluated to false, which might cause simulation to disagree with synthesis.**

```
// Comparison to X Ignored
always begin
    if ( A === 1'bx )
        B = 0;
    else
        B = 1;
end
```

Handling Comparison to X and Z

- Improper using of case statement might cause synthesis not to agree with simulation.

```
module case_withxz(q, a, b, c, d, sel);
input a, b, c, d;
input [1:0] sel;
output q;
reg    q;

always @(sel or a or b or c or d)
  case(sel)
    2'b00: q=a;
    2'b0x: q=b;
    2'b10: q=c;
    2'b11: q=d;
  endcase

endmodule
```



Synopsys Design Analyzer
Setup File Edit View Attributes Analysis Tools Help
Current Design: case_withxz
Schematic View
Left Button: Select - Middle Button: Add/Modify Select - Right Button: Menu

Limitations of Using Delay Specification

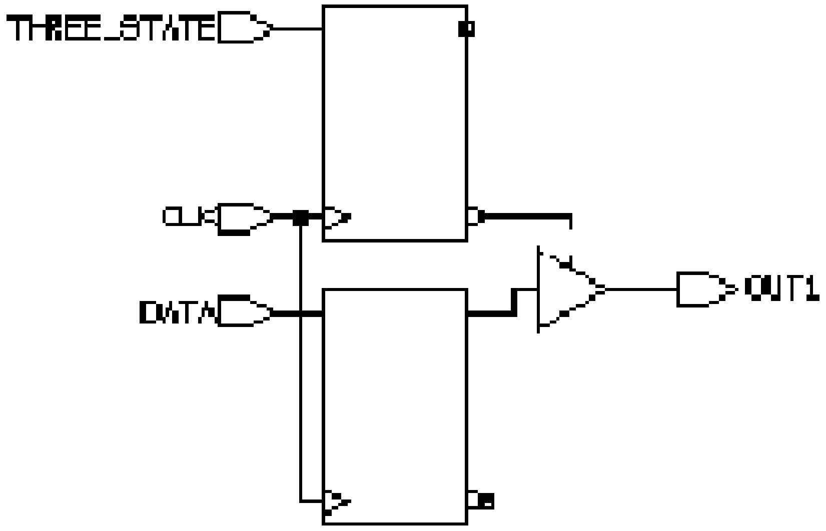
- **You can use delay specification information for modeling, but Design Compiler ignores delay information.**
- **If the functionality of your circuit depends on the delay information, Design Compiler might create logic whose behavior does not agree with the behavior of the simulated circuit.**

```
module top(a, c, d, clk);
reg b;…

flip_flop F1(a, clk, c);
flip_flop F2(b, clk, d);

always @(a or c or d or clk)
begin
    b <= #100 a;
end
endmodule
```

Limitations of Tri-State Inferences

- **When a variable is registered in the same block in which it is three-stated, HDL Compiler also registers the enable of this type of code.**

```
// Three-State Driver with Enable
module ff_3state (DATA, CLK, THREE_STATE, OUT1);
input DATA, CLK, THREE_STATE;
output OUT1;
reg OUT1;
    always @ (posedge CLK) begin
        if (THREE_STATE)
          OUT1 = 1'bz;
        else
           OUT1 = DATA;
    end
endmodule
```

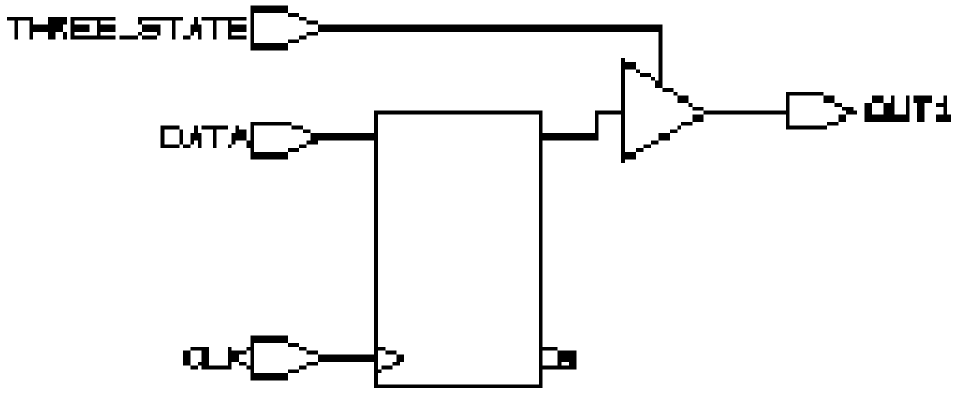


Limitations of Tri-State Inferences

Enabling Productivity

- **An example for Three-State Driver without Registered Enable**

```
// Three-State Driver without Registered Enable
module ff_3state (DATA, CLK, THREE_STATE, OUT1);
input DATA, CLK, THREE_STATE;
output OUT1;
reg OUT1;
reg TEMP;
    always @(posedge CLK)
        TEMP = DATA;

    always @(THREE_STATE or TE
        if (THREE_STATE)
            OUT1 = TEMP;
        else
            OUT1 = 1'bz;
endmodule
```



Limitations of Arithmetic Operators

- ❑ **Arithmetic operators perform simple arithmetic on operands. The Verilog arithmetic operators are**
  - ☞ **Addition (+)**
  - ☞ **Subtraction (-)**
  - ☞ **Multiplication (*)**
  - ☞ **Division (/)**
  - ☞ **Modules (%)**
- ❑ **HDL Compiler requires that / and % operators have constant-valued operands.**

Limitations of casex and casez Statements

- **HDL Compiler allows ?, z, x bits in casex items in a casex statement, but not in casex expressions.**
- **HDL Compiler allows ?, z in casez items in a casez statements, but not in casez expressions.**

```
// Invalid casex Expression
express = 3'bxz?;
…
//illegal testing of an expression
casex (express)
…
Endcase
```

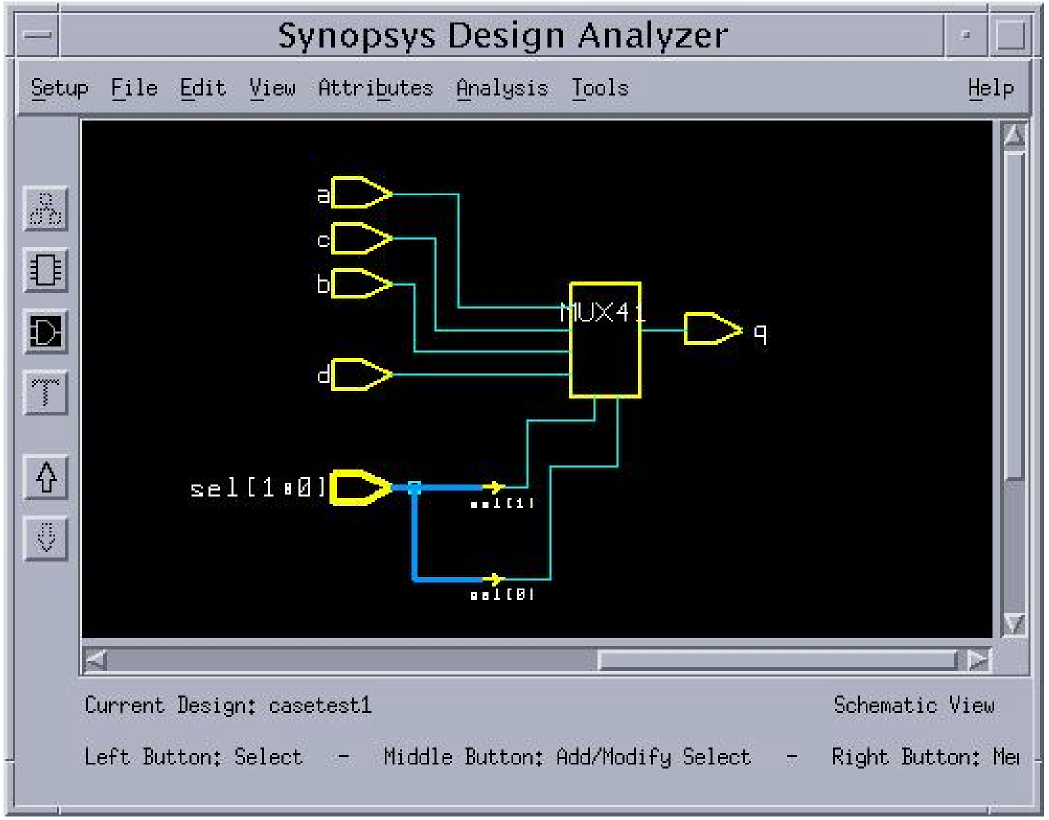
```
// Invalid casez Expression
express = 1'bz;
…
//illegal testing of an expression
casez (express)
…
Endcase
```

Case Statement: Full Case

Enabling Productivity

- ❑ **A case statement is full if all possible branches are specified.**
- ❑ **A full case statement does not infer latches.**

```
module casetest1(q, a, b, c, d, sel);
input a, b, c, d;
input [1:0] sel;
output q;
reg    q;

always @(sel or a or b or c or d)
  case(sel)
    2'b00: q=a;
    2'b01: q=b;
    2'b10: q=c;
    2'b11: q=d;
  endcase

endmodule
```



Current Design: casetest1   Schematic View

Left Button: Select - Middle Button: Add/Modify Select - Right Button: Me

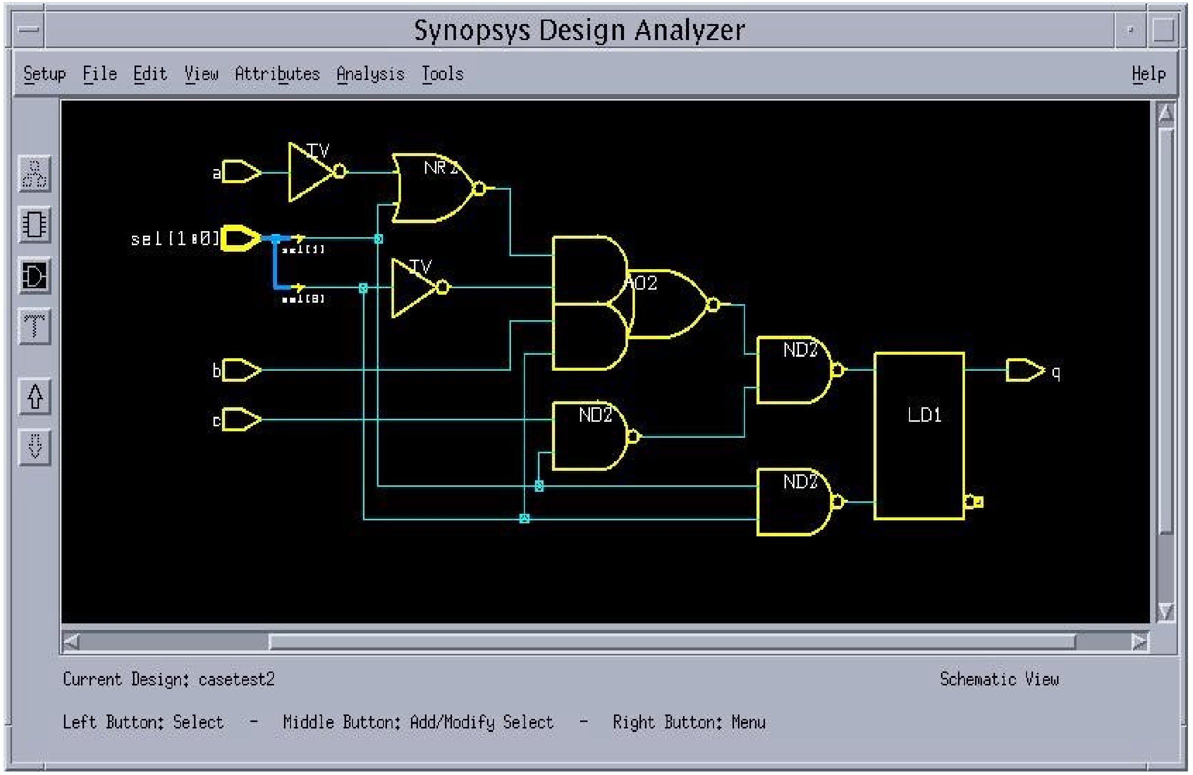
Professional Service Group

Case Statement: No-full Case

Enabling Productivity

- A case statement that is not full case infers latches.

```
module casetest2(q, a, b, c, sel);
input a, b, c;
input [1:0] sel;
output q;
reg    q;

always @(sel or a or b or c)
  case(sel)
    2'b00: q=a;
    2'b01: q=b;
    2'b10: q=c;
  endcase

endmodule
```



Current Design: casetest2

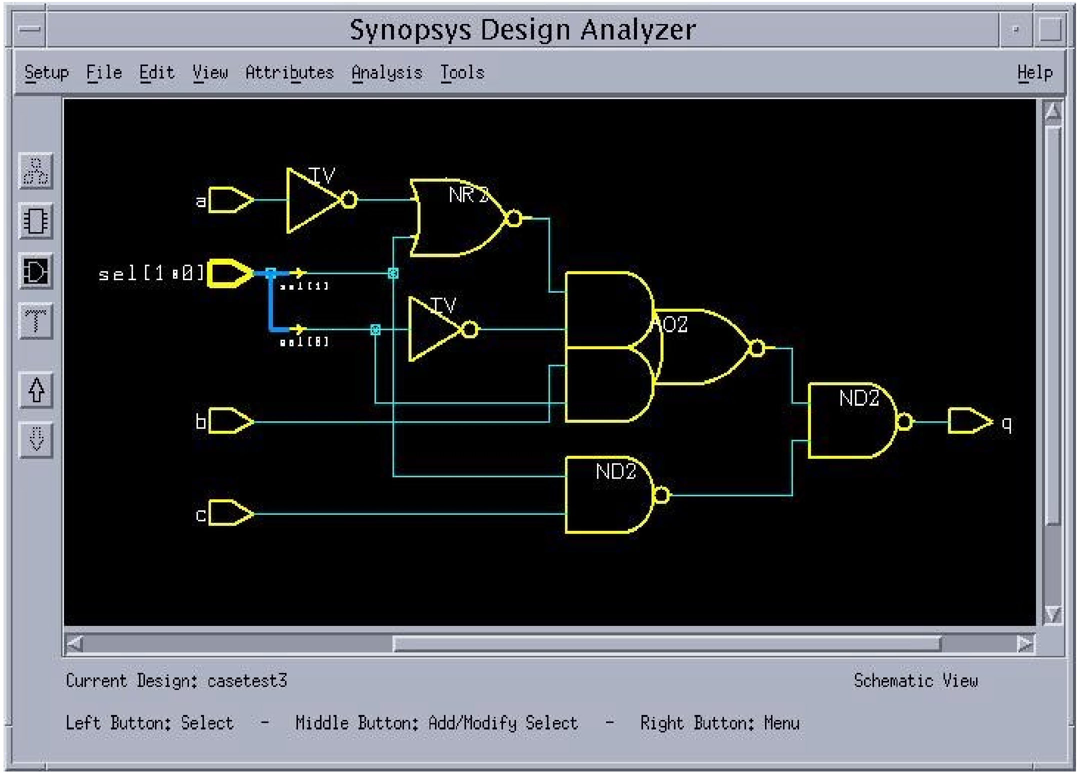
Schematic View

Left Button: Select - Middle Button: Add/Modify Select - Right Button: Menu

Case Statement: Compile Directive

- A compile directive "synopsys full_case" guides Design Compiler not to synthesize latches.

```
module casetest3(q, a, b, c, sel);
input a, b, c;
input [1:0] sel;
output q;
reg    q;

always @(sel or a or b or c)
  case(sel) //synopsys full_case
    2'b00: q=a;
    2'b01: q=b;
    2'b10: q=c;
  endcase

endmodule
```


Synopsys Design Analyzer
Setup File Edit View Attributes Analysis Tools Help
a
sel[1:0]
sel[1]
sel[0]
b
c
IV
NR2
IV
AO2
ND2
ND2
q
Schematic View
Left Button: Select - Middle Button: Add/Modify Select - Right Button: Menu

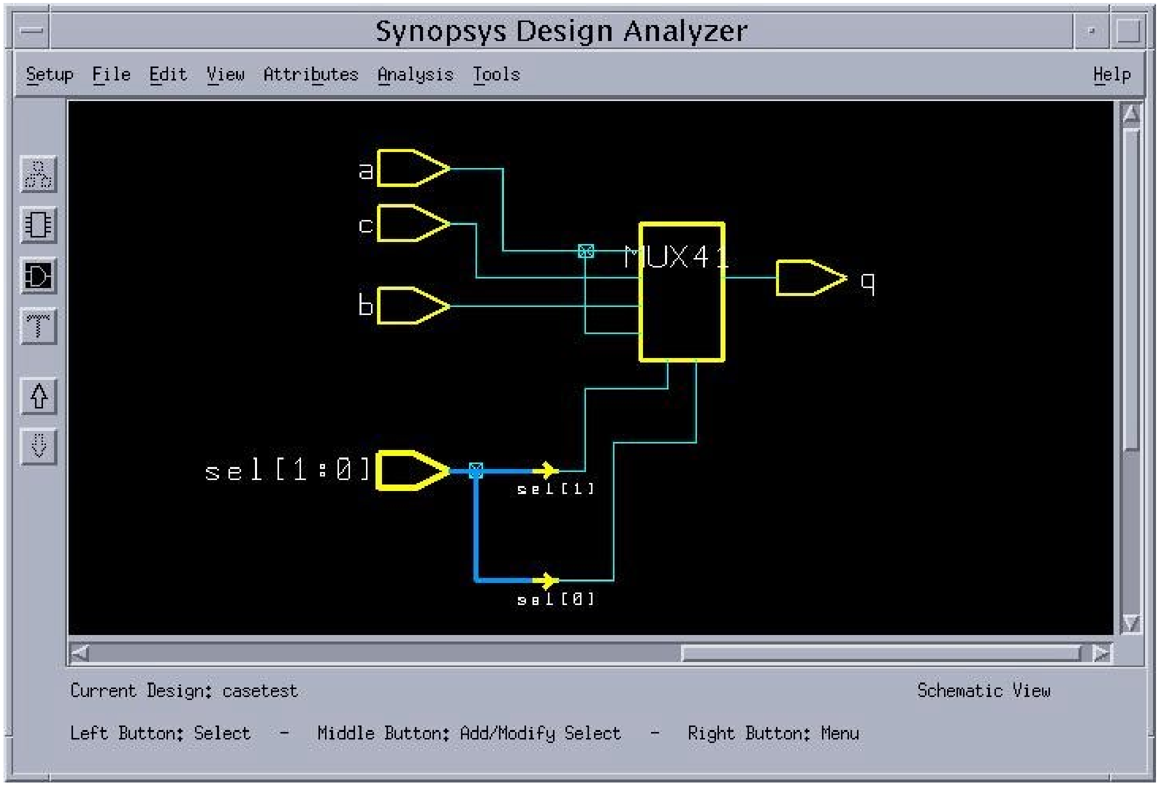
Current Design: casetest3

Case Statement: Default Clause

Enabling Productivity

- A case statement that is not full case with the default clause does not infer latches.

```
module casetest4(q, a, b, c, sel);
input a, b, c;
input [1:0] sel;
output q;
reg    q;

always @(sel or a or b or c)
  case(sel)
    2'b00: q=a;
    2'b01: q=b;
    2'b10: q=c;
    default: q=a;
  endcase

endmodule
```

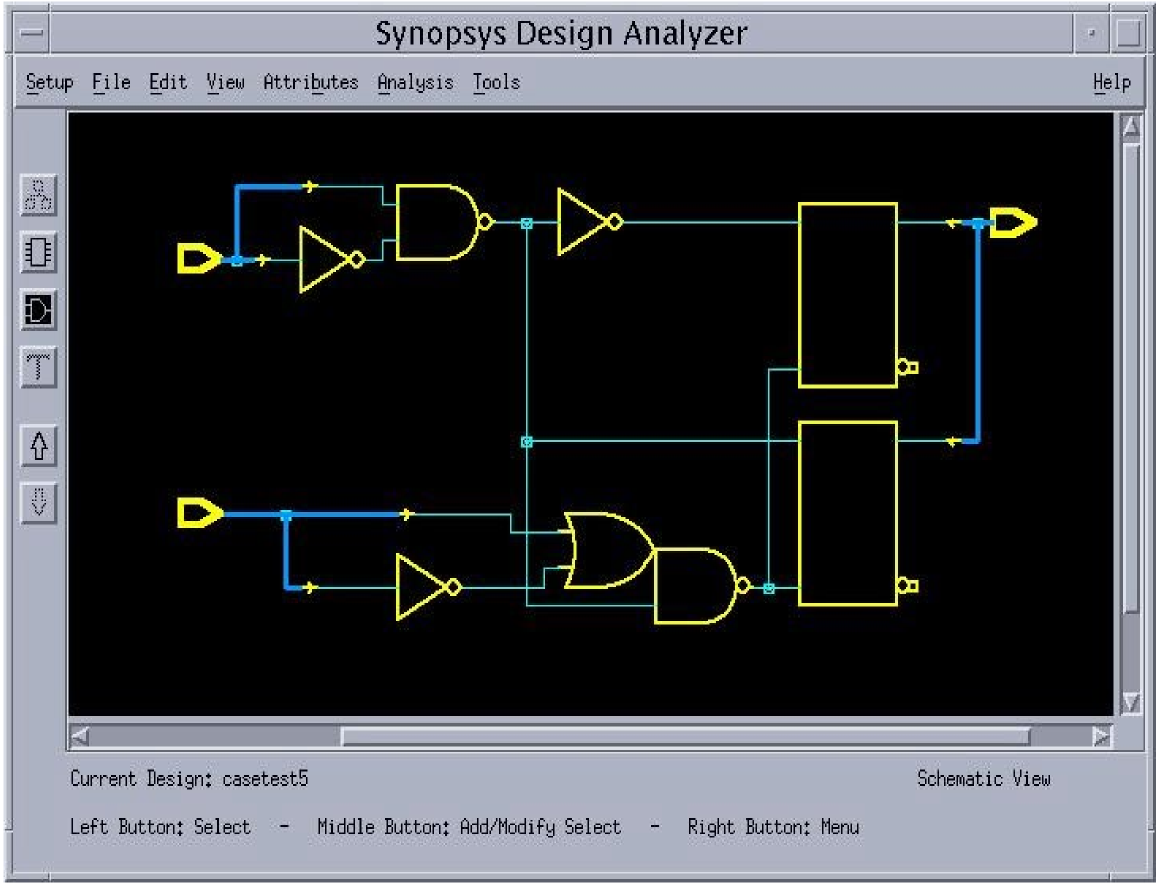


Synopsys Design Analyzer
Setup File Edit View Attributes Analysis Tools
Help
a
c
b
MUX41
q
sel[1:0]
sel[1]
sel[0]
Current Design: casetest
Schematic View
Left Button: Select - Middle Button: Add/Modify Select - Right Button: Menu

Case Statement: with Priority Encoder

- ❑ **A case statement is parallel case if no case items overlap.**
- ❑ **In the following example, a priority encoder is synthesized.**

```
module casetest5(q, a, b);
input  [1:0]  a, b;
output [1:0]  q;
reg    [1:0]  q;

always @(a or b)
  case(2'b10)
    a: q = 2'b10;
    b: q = 2'b01;
  endcase

endmodule
```



Current Design: casetest5 Schematic View

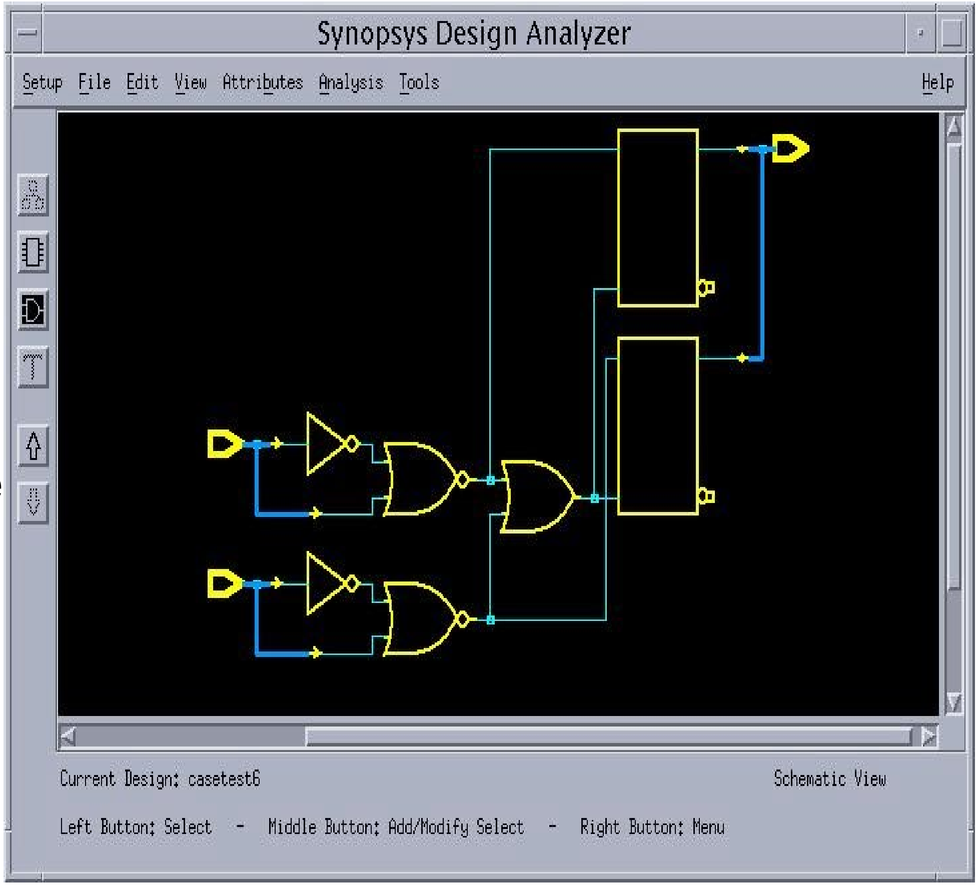
Left Button: Select - Middle Button: Add/Modify Select - Right Button: Menu

Case Statement: without Priority Encoder

Enabling Productivity

- ❑ **A compiler directive "synopsys parallel_case" guides Design Compiler not to synthesize a priority encoder.**

```
module casetest6(q, a, b);
input  [1:0]  a, b;
output [1:0]  q;
reg    [1:0]  q;

always @(a or b)
  case(2'b10) //synopsys parallel_case
     a: q = 2'b10;
     b: q = 2'b01;
  endcase

endmodule
```

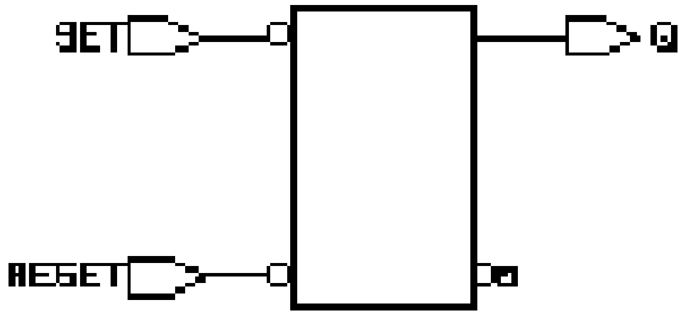

Synopsys Design Analyzer
Setup File Edit View Attributes Analysis Tools Help
Current Design: casetest6
Schematic View
Left Button: Select - Middle Button: Add/Modify Select - Right Button: Menu

SYNOPSYS®
Professional Service Group

SR Latch

Enabling Productivity

```
module sr_latch (SET, RESET, Q);
input SET, RESET;
output Q;
reg Q;

//synopsys async_set_reset     "SET, RESET"
always @(RESET or SET)
     if (~RESET)
          Q = 0;
     else if (~SET)
          Q = 1;

endmodule
```



SYNOPSYS®
Professional Service Group

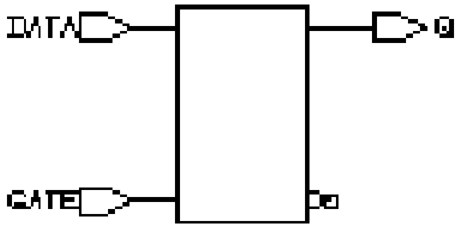
Latch Inference Using an if Statement

Enabling Productivity

```
// Latch Inference Using an if Statement
always @ (DATA or GATE) begin
     if (GATE) begin
          Q = DATA;
     end
```

```
// Avoiding Latch Inference
always @ (DATA, GATE) begin
    Q = 0;
    if (GATE)
        Q = DATA;
    end
```

```
// Another Way to Avoid Latch Inference
always @ (DATA, GATE) begin
    if (GATE)
        Q = DATA;
    else
        Q = 0;
end
```

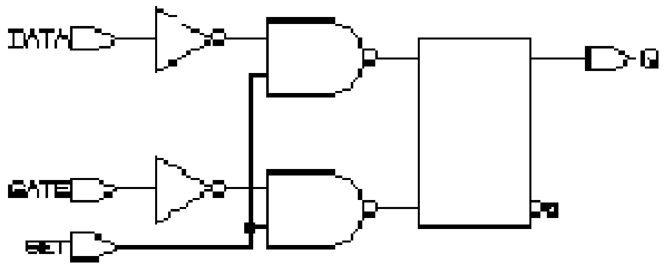


D Latch

SYNOPSYS®
Professional Service Group

D Latch with Asynchronous Set

```
// D Latch with Asynchronous Set
module d_latch_async_set (GATE, DATA, SET, Q);
input GATE, DATA, SET;
output Q;
reg Q;

//synopsys async_set_reset  “SET”
always @(GATE or DATA or SET)
    if (~SET)
        Q = 1’b1;
    else if (GATE)
        Q = DATA;
endmodule
```

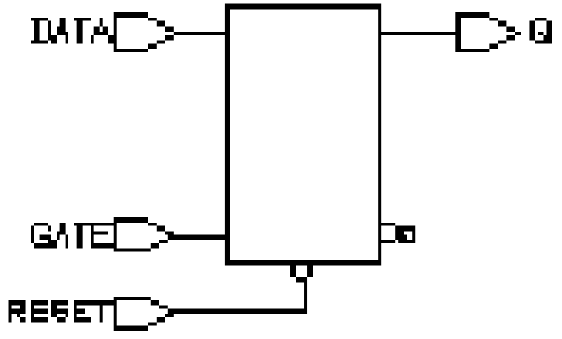


Ps. Because the target technology library does not contain a latch with an asynchronous set, Design Compiler synthesizes the set logic, using combinational logic.

D Latch with Asynchronous Reset

Enabling Productivity

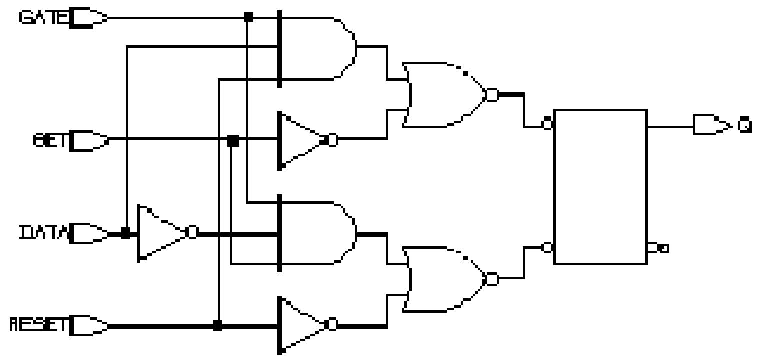
```
// D Latch with Asynchronous Reset
module d_latch_async_reset (RESET, GATE, DATA,
Q);
input RESET, GATE, DATA;
output Q;
reg Q;

//synopsys async_set_reset “RESET”
always @ (RESET or GATE or DATA)
     if (~RESET)
         Q <= 1’b0;
    else if (GATE)
         Q <= DATA;
endmodule
```



D Latch with Asynchronous Set and Reset

Enabling Productivity

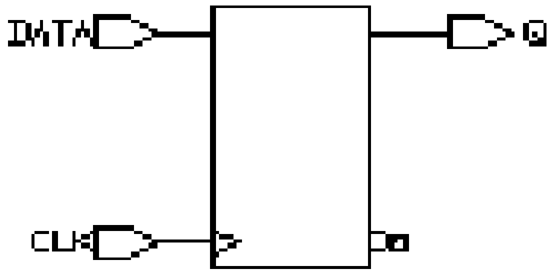
```
// D Latch with Asynchronous Set and Reset
module d_latch_async (GATE, DATA, RESET, SET, Q);
input GATE, DATA, RESET, SET;
output Q;
reg Q;

// synopsys async_set_reset "RESET, SET"
// synopsys one_cold  "RESET, SET"
always @ (GATE or DATA or RESET or SET)
begin : infer
    if (!SET)
        Q <= 1'b1;
    else if (!RESET)
        Q <= 1'b0;
    else if (GATE)
        Q <= DATA;
end
```



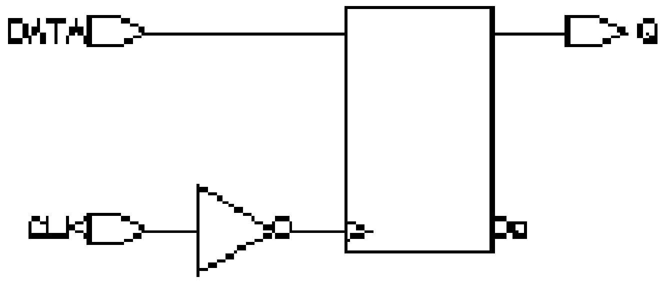
Simple D Flip-Flop

Enabling Productivity

```
// Positive Edge-Triggered D Flip-Flop
module dff_pos (DATA, CLK, Q);
input DATA, CLK;
output Q;
reg Q;

always @(posedge CLK)
    Q <= DATA;
endmodule
```

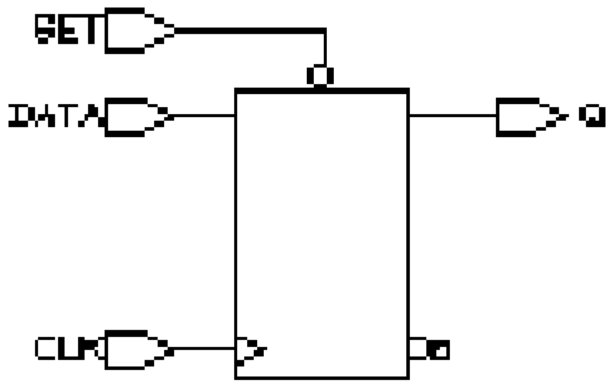


```
// Negative Edge-Triggered D Flip-Flop
module dff_neg (DATA, CLK, Q);
input DATA, CLK;
output Q;
reg Q;

always @(negedge CLK)
    Q <= DATA;
endmodule
```



D Flip-Flop with Asynchronous Set

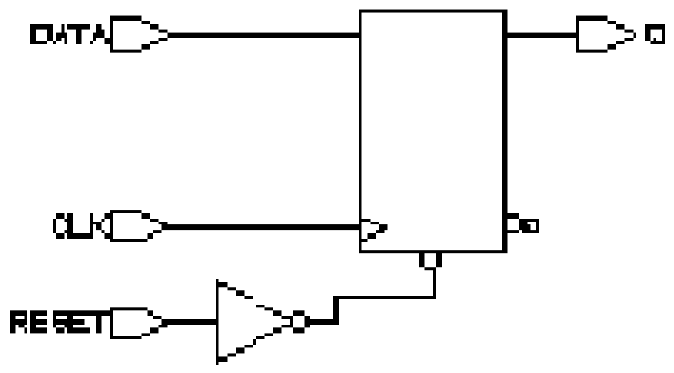
```
// D Flip-Flop with Asynchronous Set
module dff_async_set (DATA, CLK, SET, Q);
input DATA, CLK, SET;
output Q;
reg Q;

always @(posedge CLK or negedge SET)
    if (~SET)
        Q <= 1’b1;
    else
        Q <= DATA;
endmodule
```



D Flip-Flop with Asynchronous Reset

Enabling Productivity

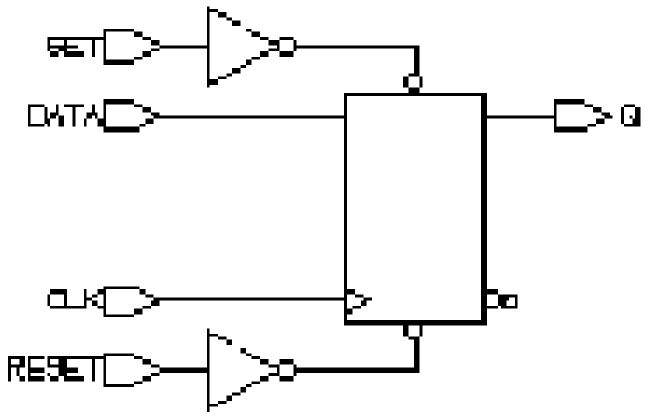
```
//  D Flip-Flop with Asynchronous Reset
module dff_async_reset (DATA, CLK, RESET, Q);
input DATA, CLK, RESET;
output Q;
reg Q;

always @(posedge CLK or posedge RESET)
    if (RESET)
        Q <= 1'b0;
    else
        Q <= DATA;
endmodule
```



D Flip-Flop with Asynchronous Set and Reset

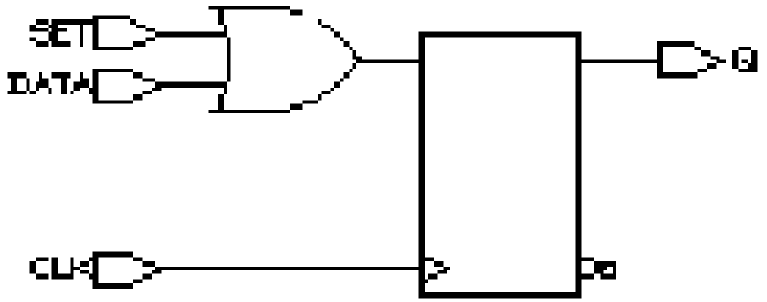
Enabling Productivity

```
// D Flip-Flop with Asynchronous Set and Reset
module dff_async (RESET, SET, DATA, Q, CLK);
input CLK;
input RESET, SET, DATA;
output Q;
reg Q;

// synopsys one_hot "RESET, SET"
always @(posedge CLK or posedge RESET or posedge SET)
    if (RESET)
        Q <= 1'b0;
    else if (SET)
        Q <= 1'b1;
    else
         Q <= DATA;
endmodule
```



D Flip-Flop with Synchronous Set

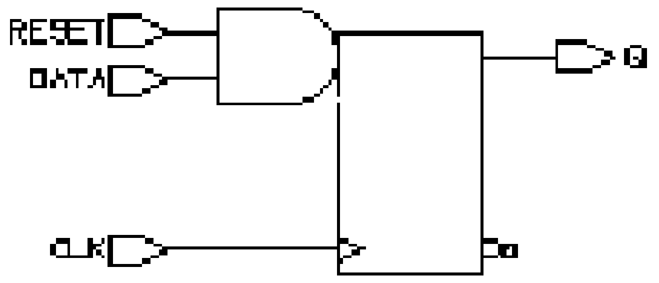
```
// D Flip-Flop with Synchronous Set
module dff_sync_set (DATA, CLK, SET, Q);
input DATA, CLK, SET;
output Q;
reg Q;

//synopsys sync_set_reset  ”SET”
always @(posedge CLK)
    if (SET)
        Q <= 1’b1;
    else
        Q <= DATA;
endmodule
```



D Flip-Flop with Synchronous Reset

Enabling Productivity

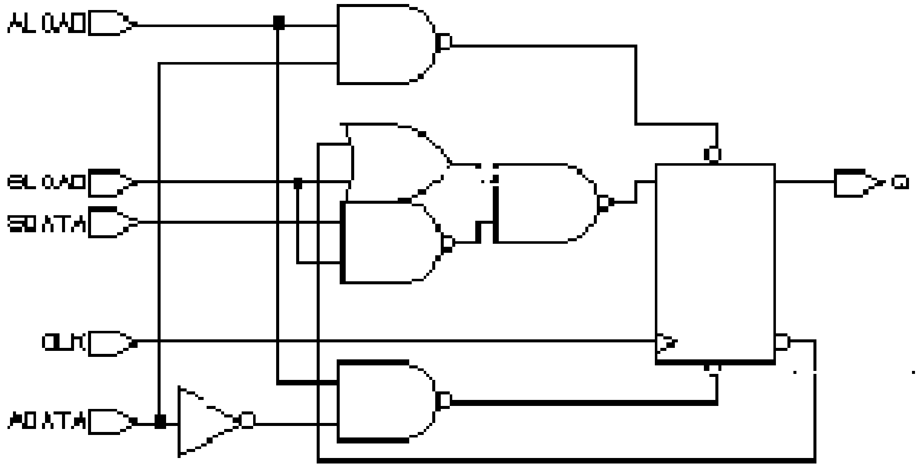
```
// D Flip-Flop with Synchronous Reset
module dff_sync_reset (DATA, CLK, RESET, Q);
input DATA, CLK, RESET;
output Q;
reg Q;

//synopsys sync_set_reset "RESET"
always @(posedge CLK)
    if (~RESET)
        Q <= 1'b0;
    else
        Q <= DATA;
endmodule
```



D Flip-Flop with Synchronous and Asynchronous Load

Enabling Productivity

```
// D Flip-Flop with Synchronous and Asynchronous Load
module dff_a_s_load (ALOAD, SLOAD, ADATA, SDATA, CLK, Q);
input ALOAD, ADATA, SLOAD, SDATA, CLK;
output Q;
reg Q;

always @ (posedge CLK or posedge ALOAD)
   if (ALOAD)
       Q <= ADATA;
   else if (SLOAD)
       Q <= SDATA;
endmodule
```



Professional Service Group