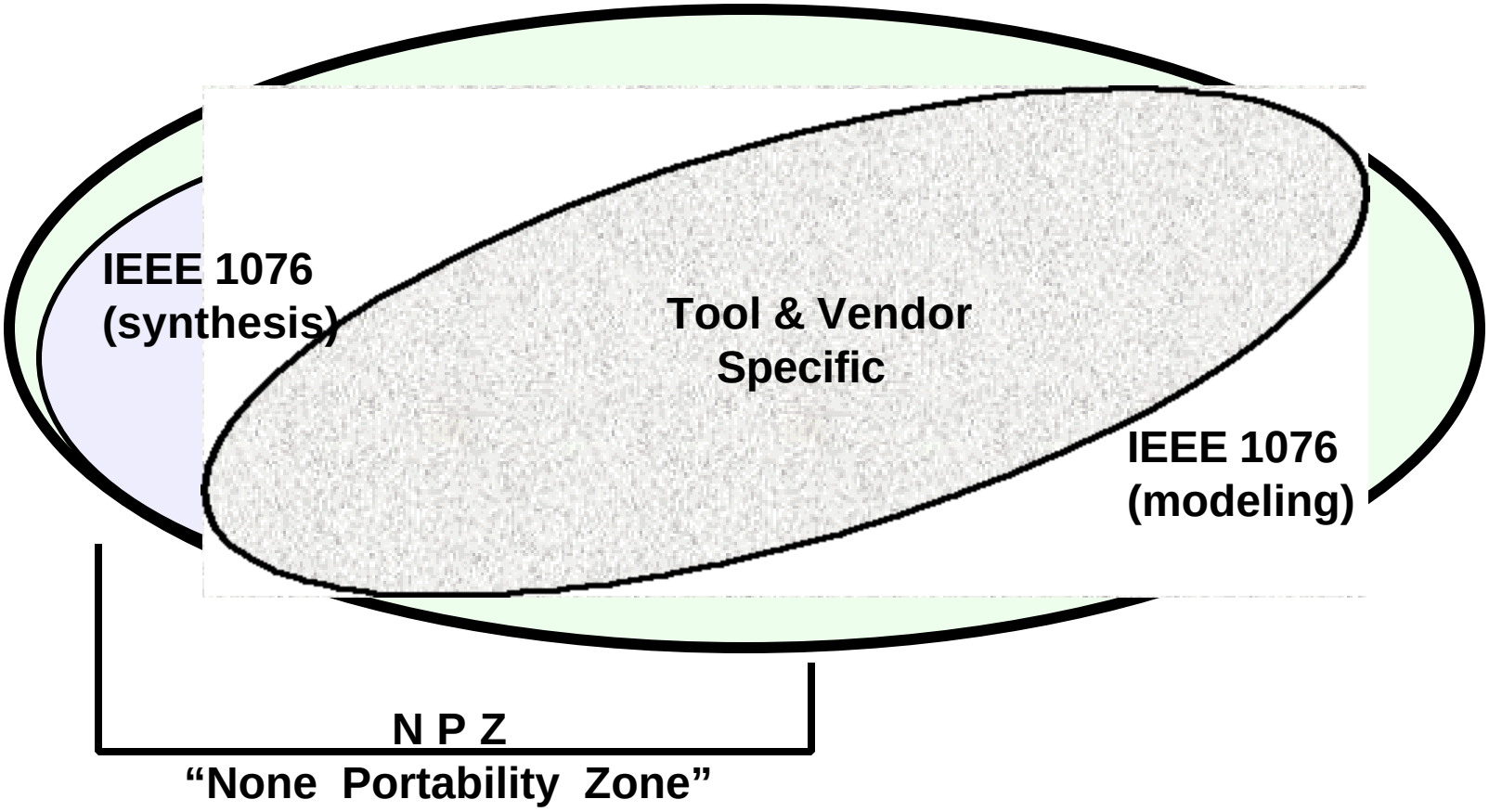
Chapter 4

HDL Coding Style

Outline

- **HDL and Synthesis Concept**
- Hierarchy of HDL design
- Latch inference and registers
- Instantiation and Black box
- Synchronous and Asynchronous Design
- Combinatorial and Sequential Logic
- Case V.S. If - elsif
- Others

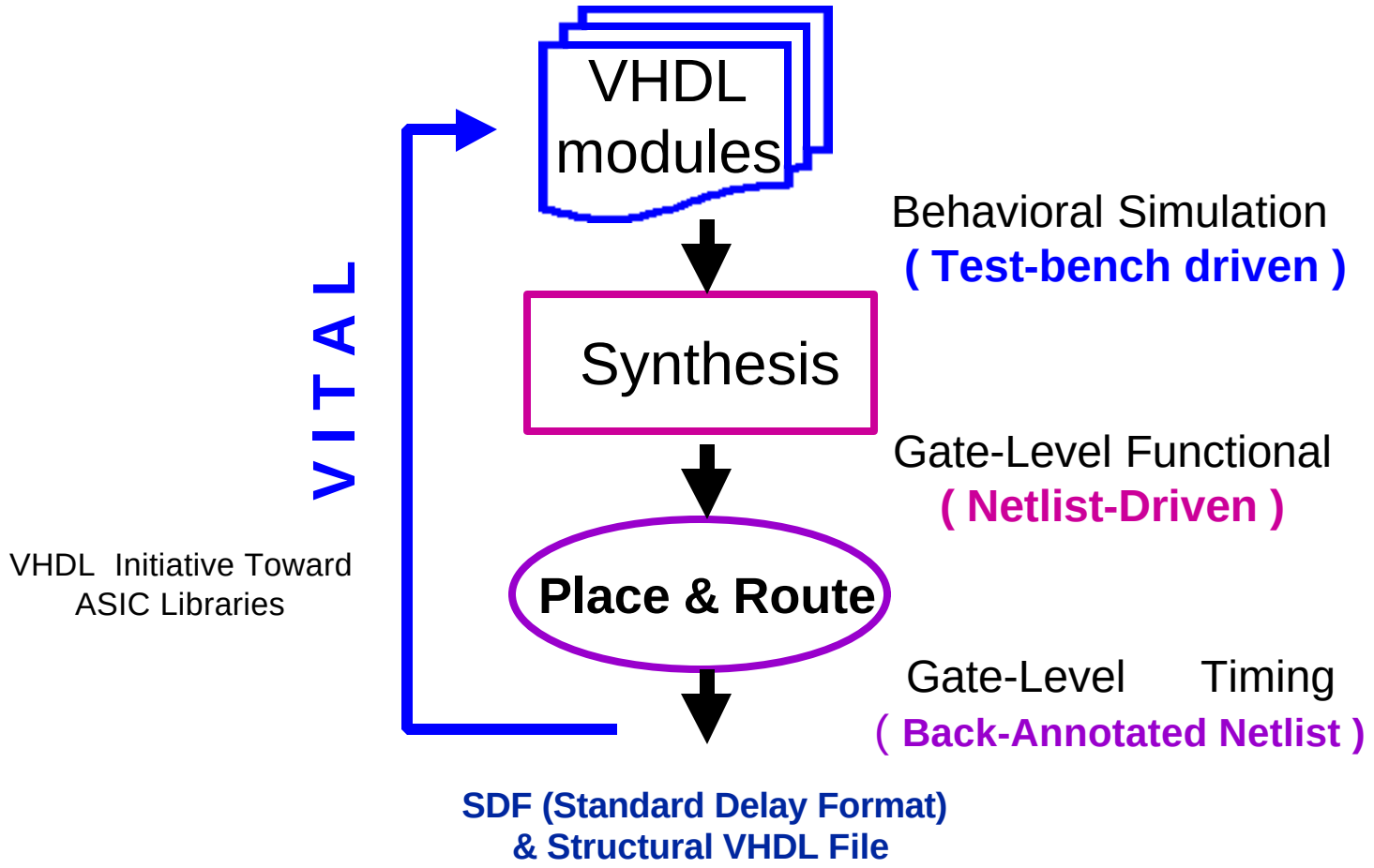
Language Subsets



Avoid extensive use of tool specific constructs that are outside of standard VHDL

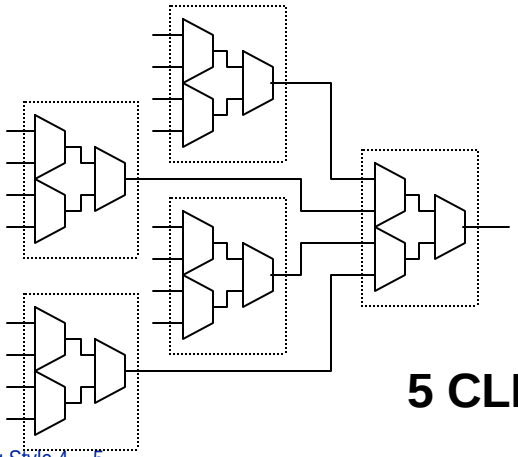
Design Verification

- When using an HDL entry method, there is an additional level of design verification available.

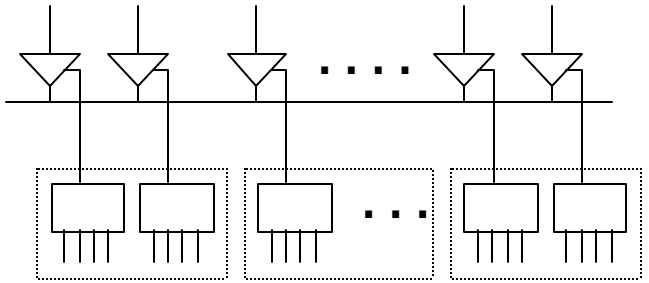


Technology Independent Synthesis

- The synthesis process consists of two steps:
  - ❶ **Synthesis** technology & constraint independent
  - ❷ **Optimization** technology & constraint driven
- Decisions made at the synthesis stage have effect on implementation
  - — this is where your HDL coding style has an impact
- Take a simple example: a 16-to-1 multiplexer...



5 CLBs



8 CLBs

```
-- 4 to 1 multiplexer design with case construct
--          SEL: in STD_LOGIC_VECTOR(1 downto 0);
--          A, B, C, D:in STD_LOGIC;
--          MUX_OUT: out STD_LOGIC;

process (SEL, A, B, C, D)
begin
   case SEL is
      when "00" => MUX_OUT <= A;
      when "01" => MUX_OUT <= B;
      when "10" => MUX_OUT <= C;
      when "11" => MUX_OUT <= D;
   end case;
end process;
```

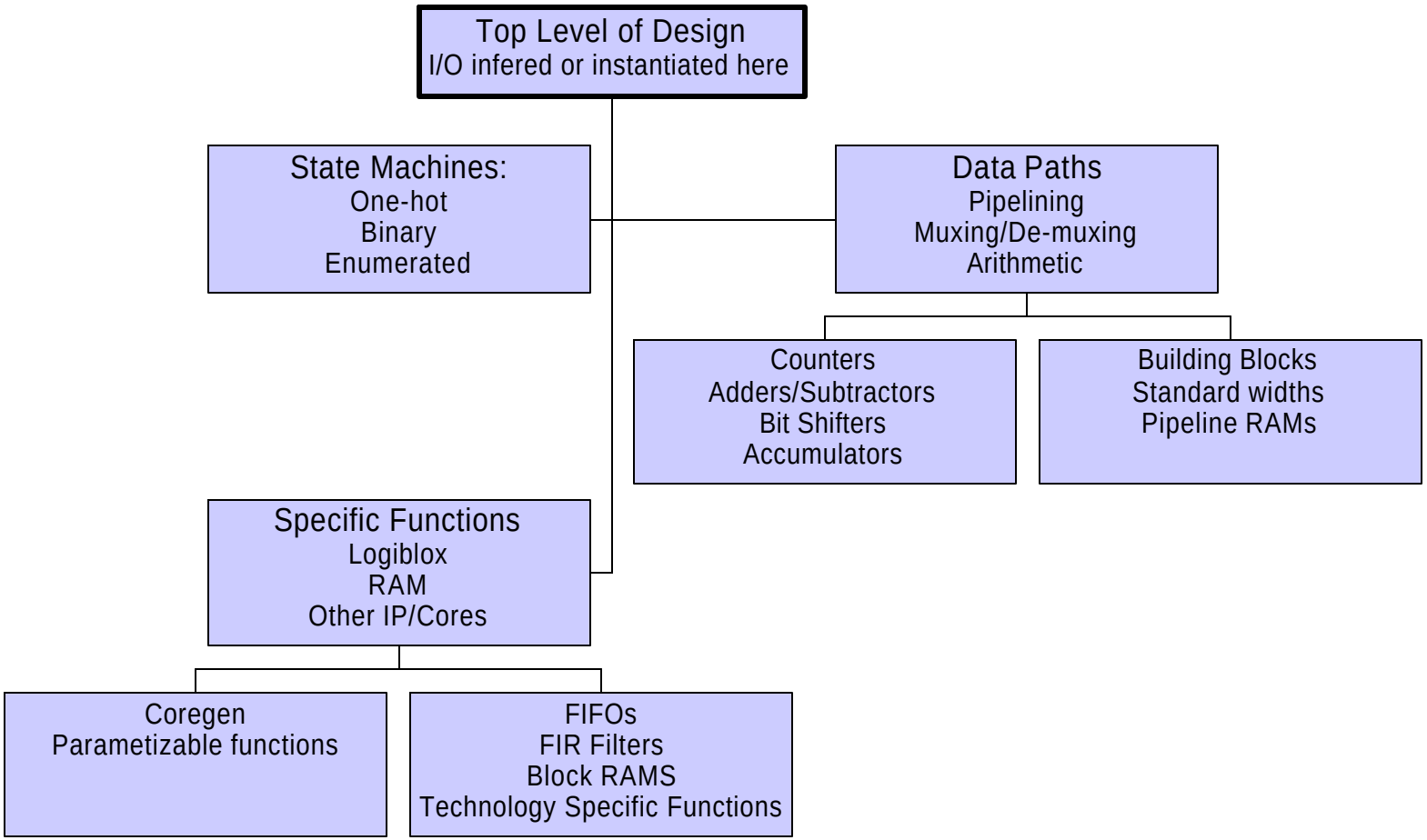
```
-- 4 to 1 multiplexer design with tri-state construct
--          SEL: in STD_LOGIC_VECTOR(3 downto 0);
--          A, B, C, D:in STD_LOGIC;
--          MUX_OUT: out STD_LOGIC;

   MUX_OUT <= A when (SEL(0)='0') else 'Z';
   MUX_OUT <= B when (SEL(1)='0') else 'Z';
   MUX_OUT <= C when (SEL(2)='0') else 'Z';
   MUX_OUT <= D when (SEL(3)='0') else 'Z';
```

Outline

- HDL and Synthesis Concept
- **Hierarchy of HDL design**
- Latch inference and registers
- Instantiation and Black box
- Synchronous and Asynchronous Design
- Combinatorial and Sequential Logic
- Case V.S. If - elsif
- Others

Using Hierarchy in HDL

- Using Hierarchy leads to easy design readability, re-use, and debug



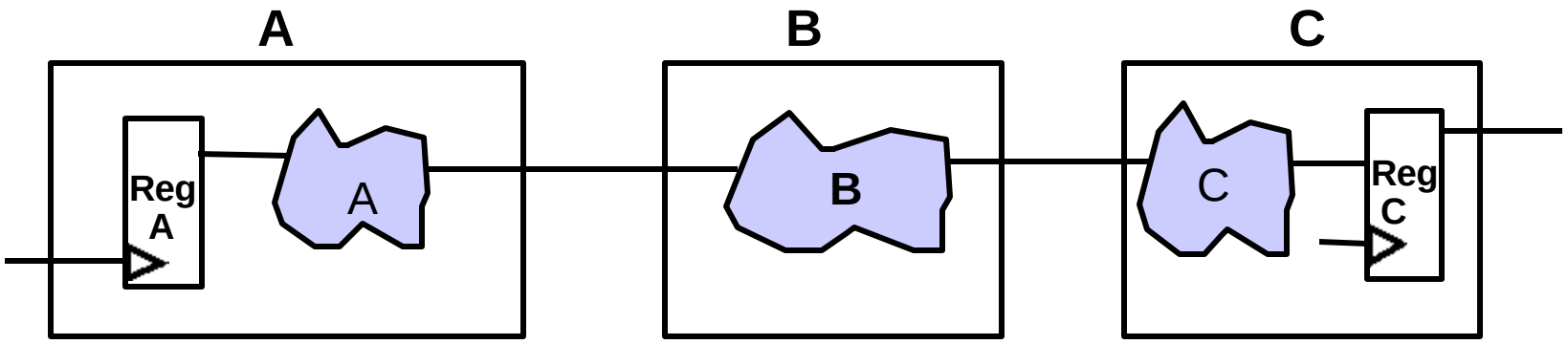
Guidelines for Choosing Hierarchy

- Consider the following points when arranging your hierarchy
  - all **arithmetic operators** should be evaluated for resource sharing and combined within the same hierarchy/process
  - keep distinct **logic-types** (such as state machines, random logic, data paths, etc.) separate, so the appropriate optimizations can be applied to each
- Choose modules that have
  - a minimum of routing between modules
  - a logical data flow between modules
- Some synthesis tools flatten the design before optimization
- There are commands to remove hierarchy
  - in general, leave hierarchy in the design for later visibility into the design after place and route

Keep Related Logic Together (1)

No Hierarchy in Combinational Path

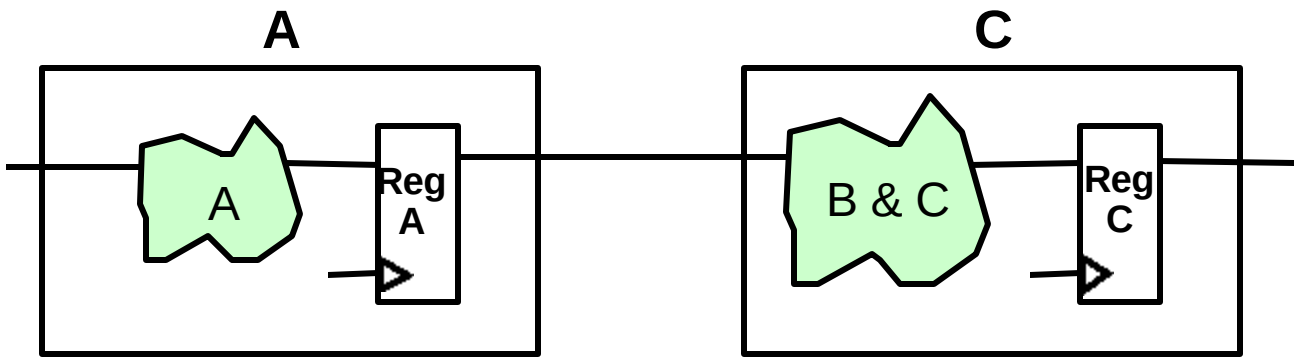
- **The path from Reg A to Reg C is divided between <u>three different block</u> descriptions**



- **Example - Optimization is limited because hierarchical boundaries prevent sharing of common terms**

Keep Related Logic Together (2)

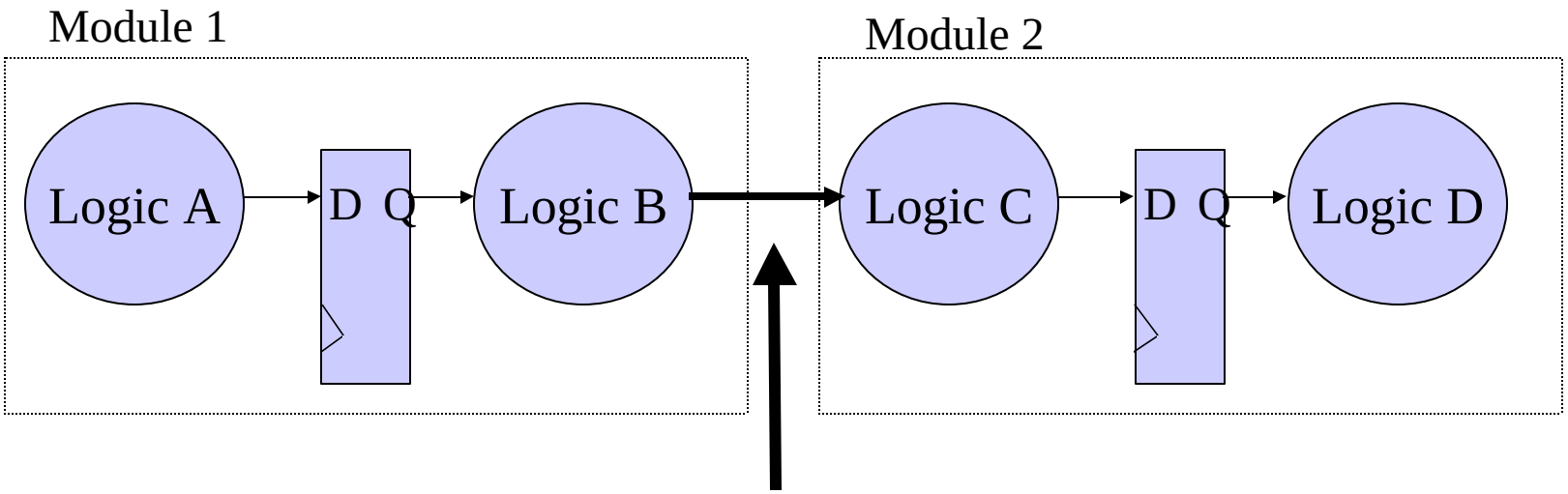
Good Example



- Related combinational **logic drive registers in the same block**
- **No hierarchical boundaries between combinational logic and registers**
  - **Allows for improved sequential mapping**

Register Hierarchical Boundaries

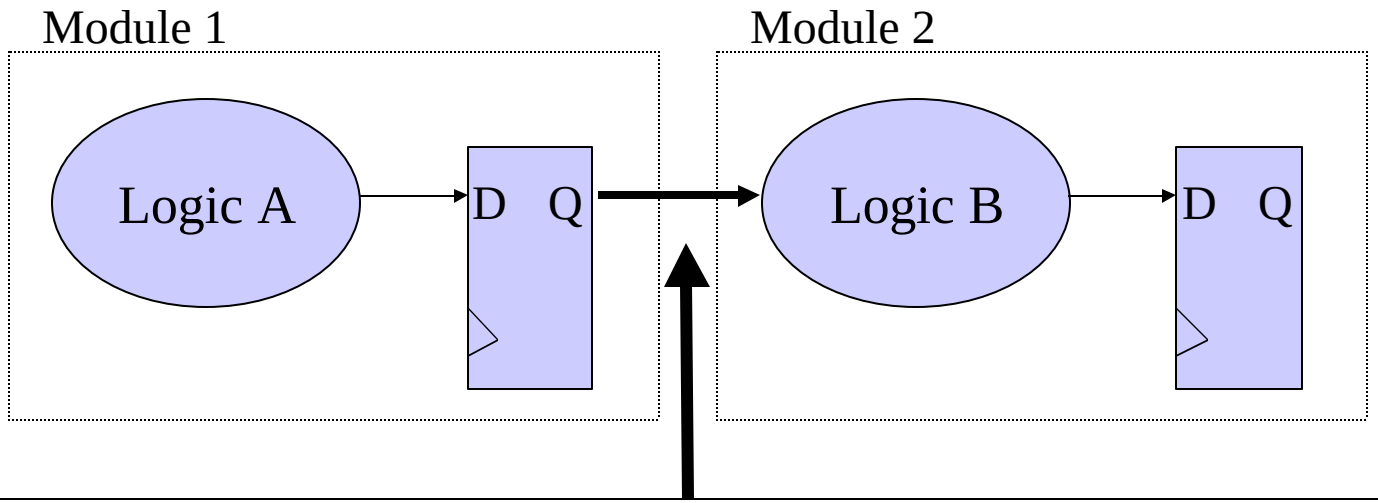
- This reduces the amount of cross boundary optimization the synthesizer must do
- **(X) Bad Module Boundary**:



At this boundary, the Synthesis tool must decide how much to optimize Logic B and C together? It is un-wise to let the synthesis tool make a judgement call about your design. This type of design leads to bigger, slower, and less likely to work designs

Register Hierarchical Boundaries

- Register Boundary at the output of each module provides a more stabile, re-useable, and synthesizable design
- **Good Module Boundary:**



At this boundary, the Synthesis tool has no decisions to make. Logic B is going to be synthesized the same every time, regardless of which module is attached to its inputs or outputs. Additionally, Module 2 can be synthesized and tested on its own, to insure performance as expected.

Use Hierarchy to Isolate Technology

- **Put technology specific cores in their own hierarchical blocks to allow for maximum design re-use**
  - Xilinx block RAMs, distributed RAMs, DLLs, I/Os, clock buffers, global resets, Coregen modules
- **Keep clock domains separated by using hierarchy**
  - this makes the interaction between clocks very clear in the design
  - reduces un-wanted clock confusion
  - easier to add timing constraints later
  - allows different design section to be synthesized individually, and tested before they are part of the larger design
- **Keep the number of lines of code per module below 400**
  - the modules are easier to read
  - the modules are easier to debug and synthesize.

Use Hierarchy to make Design Building Blocks

- Build yourself a standard set of functions you can re-use throughout your design
  - muxes, register banks, FIFOs, adders, counters, and other standard functions
- Compile from the bottom up
  - synthesize each low level module on it own the first time
  - run these lower levels into the Xilinx tools to get a resource usage estimate for each module and design sub-section
  - be sure each sub-block can meet your requirements before adding it into the main design: Is it the right size? Is it fast enough?

Outline

- HDL and Synthesis Concept
- Hierarchy of HDL design
- **Latch inference and registers**
- Instantiation and Black box
- Synchronous and Asynchronous Design
- Combinatorial and Sequential Logic
- Case V.S. If - elsif
- Others

Watch for Unintentional Latches

- Completely specify all clauses for every case and if statement
- Completely specify all outputs for every case or if statement: Unspecified outputs are required to retain their old values
- Elaboration will report all generated registers

What's wrong with these example coding sections?

```
process (A, B) begin
    if (A = '1') then
     Q <= B;
    end if;
end process;
```

(Latch Inferred)

```
process (C) begin
    case C is
     when '0' => Q <='1';
     Z <='0';
     when others => Q <= '0';
    end case;
end process;
```

VHDL

(Missing Z Output)

(Missing Outputs)

(Missing Case)

```
always @ (D) begin
    case (D)
     2'b00: Z = 1'b1;
     2'b01: Z = 1'b0;
     2'b10: S = 1'b1;
    endcase
end
```

Verilog

Implementing Registers (VHDL)

- **D Flip Flop**

```
FF: process (CLK)
begin
  if (CLK'event and CLK='1') then
    Q <= D_IN;
  end if;
end process
```

- **Flip-Flop with async. reset**

```
FF_AR: process(RESET,CLOCK)
begin
 if (RESET = '1') then
   Q <= '0'
 elsif(CLK'event and CLK='1') then
   Q <= D_IN;
 end if;
end process
```

- **Flip-Flop with async. set**

```
FF_AS: process(RESET,CLOCK)
begin
 if (RESET = '1') then
   Q <= '1'
 elsif (CLK'event and CLK='1') then
   Q <= D_IN;
 end if;
end process
```

- **Flip-Flop with sync. set**

```
FF_SS: process(CLOCK)
begin
 if (CLOCK'event and CLOCK='1') then
   if (RESET = '1') then
     Q <= '0'
   else
     Q <= D_IN;
   end if;
 end if;
end process
```

Instantiation

- Instantiation is how you dictate to a synthesis tool that you want to use a specific component from the Xilinx library
- **Instantiation makes your HDL code vendor specific**, and can make behavioral simulation difficult
- Certain Xilinx functions can only be activated by instantiation
  - Clock buffers/DLL
  - Boundary scan
  - Start-up block
  - Unbonded pads
  - All types of RAM modules
  - Virtex Select I/O
- Instantiated components might need a *don't touch* or *black box* attribute to prevent them from being changed or removed by the synthesis tool

Using Black Boxes

- Sometimes Black Boxes Need to be Instantiated:
  - RAM and ROM
  - IP Cores (PCI, DSP, etc.)
  - Other Hard Macros
- Black Boxes are Empty Placeholders in the Design Hierarchy
- Black Box Functions are Linked by the Place & Route Tool

Using Black Boxes

VHDL Requires a Component Declaration for the Port Directions

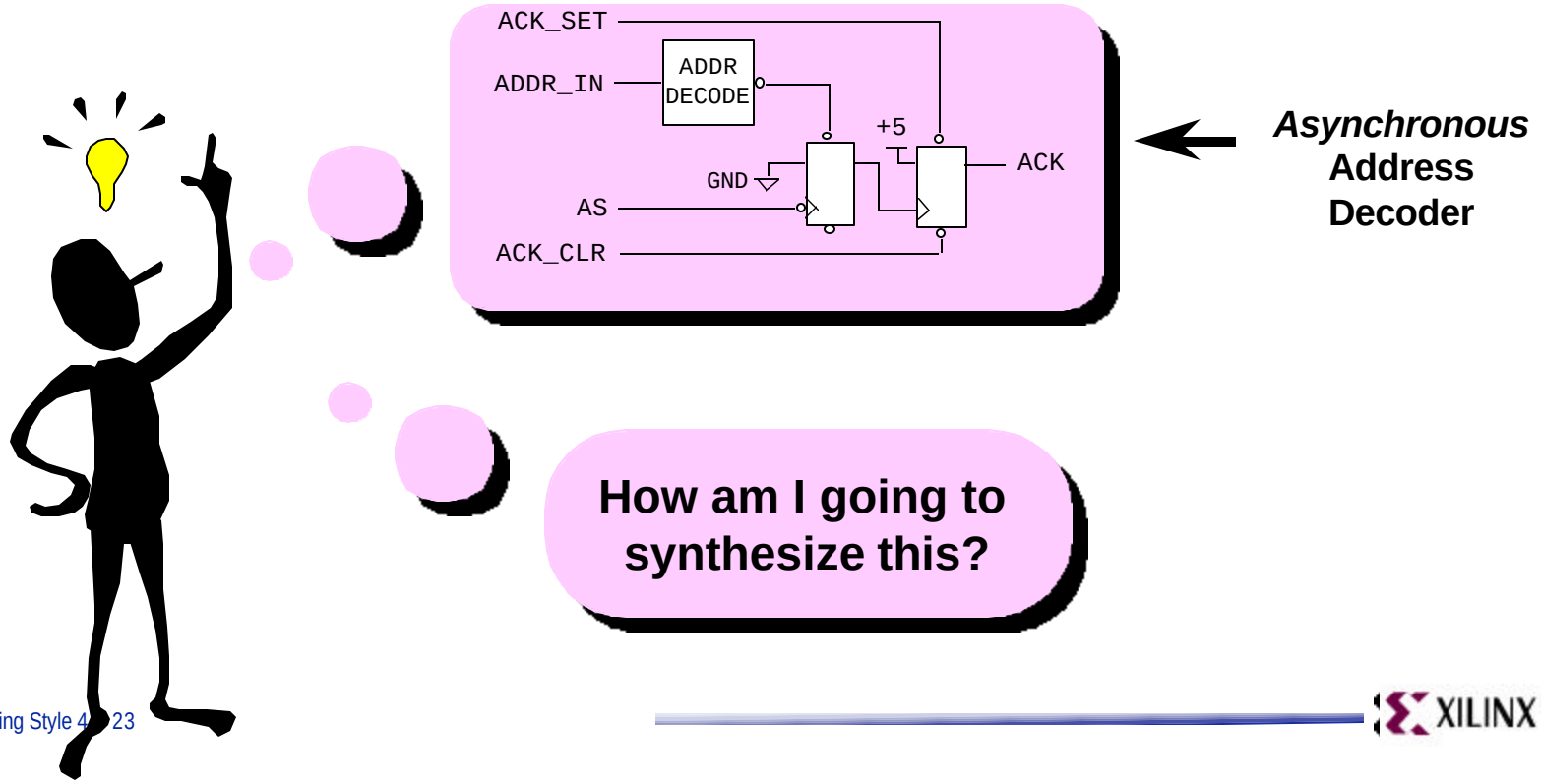
```
component MY_BLACK_BOX
  port(a, b: in std_logic; y, z: out std_logic);
end component;
...
u1: MY_BLACK_BOX
  port map(a => a_int, b => b_int,
    y => y_int, z => z_int);
```

Verilog Requires an Empty Module Definition for the Port Directions

```
module top(
...
MY_BLACK_BOX u1(.a(a_int), .b(b_int),
  .y(y_int), .z(z_int));
...
endmodule

module MY_BLACK_BOX(a, b, y, z);
input a, b;
output y, z;
endmodule
```

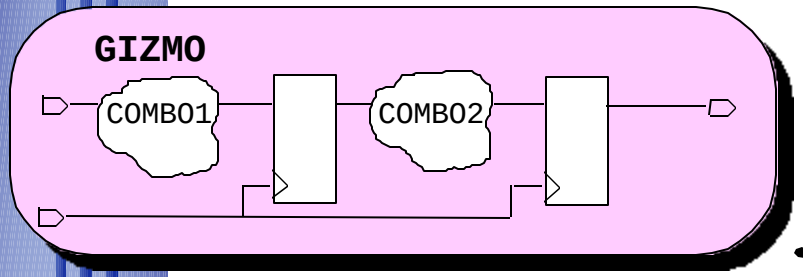
Think Synchronous Hardware

- *Synchronous* designs run smoothly through synthesis, simulation and place & route.
- *Asynchronous* designs may require instantiation and placement to verify. If asynchronous logic is necessary, isolate into separate blocks.

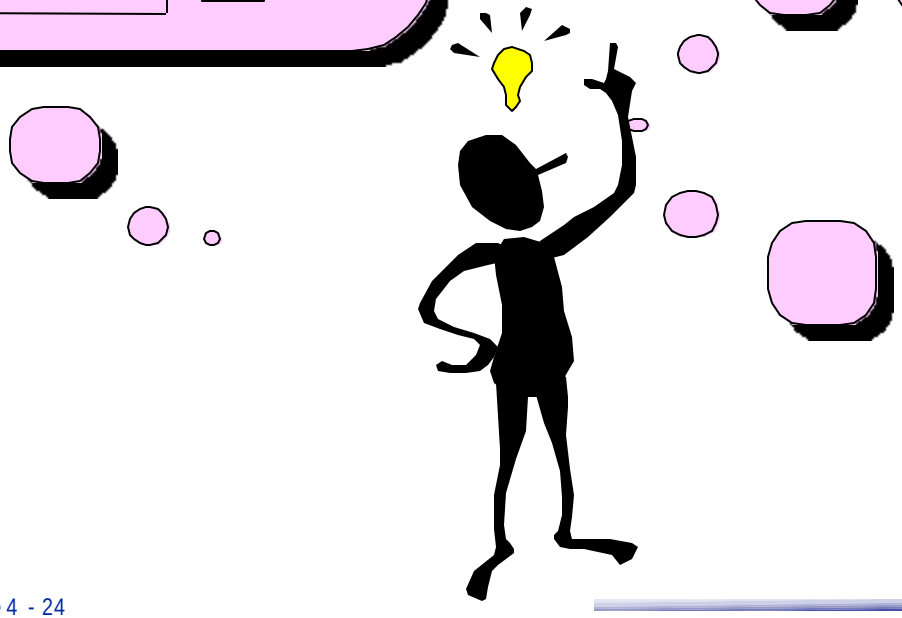


Think RTL Description of Synchronous Hardware

- Describe the register-to-register functionality of the design (i.e., describe the function of the combinational logic *between* registers).




```
module GIZMO (A, CLK, Z);
...
always@ (A) begin : COMBO1...
always@ (posedge CLK)...
always@ (B) begin : COMBO2...
always@ (posedge CLK) ...
end module;
```



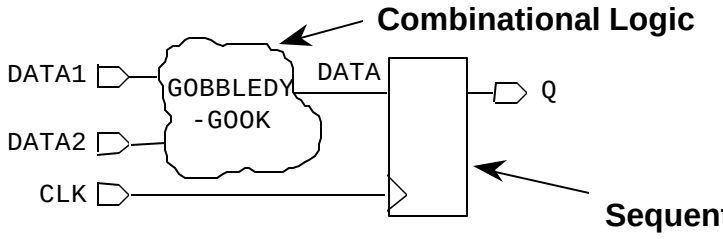
```
entity GIZMO is
     ...
architecture RTL of GIZMO is
begin
  COMBO1 : process (A) ...
  REG1 :  process (CLK) ...
  COMBO2 : process (B) ...
  REG2 : process (CLK) ...
end RTL;
```

Separate Combinational from Sequential

- Easy to read and “self-documenting”
- Follows RTL coding style.

```
module EXAMPLE (DATA1,DATA2,CLK,Q)
input DATA1,DATA2,CLK;
output Q;
reg DATA, Q;

always @ (DATA1 or DATA2)
  begin: COMBO
    DATA = GOBBLEDYGOOK (DATA1,DATA2);
  end

always @ (posedge CLK)
  begin: SEQUENTIAL
    Q <= DATA;
  end
endmodule
```

```
library IEEE;
use IEEE.std_logic_1164.all;

entity EXAMPLE is
port (DATA1,DATA2,CLK: in STD_LOGIC;
      Q: out STD_LOGIC);
end EXAMPLE;

architecture SEPARATE of EXAMPLE is
signal DATA: STD_LOGIC_VECTOR(7 downto
0);
begin
  COMBO: process (DATA1, DATA2)  begin
    DATA <= GOBBLEDYGOOK (DATA1, DATA2);
  end process COMBO;

  SEQUENTIAL: process (CLK) begin
    if (CLK'EVENT and CLK = '1') then
      Q <= DATA;
    end if;
  end process SEQUENTIAL;
end SEPARATE
```



Clock Enable Coding

- Coding Style will determine if clock enables are used
- Makes timing constraints easier to control

• VHDL

```
FF_AR_CE: process(RESET,CLK)
begin
  if (RESET = '1') then
    Q <= '0'
  elsif (CLK'event and CLK='1') then
    if (ENABLE = '1') then
      Q <= D_IN;
    end if;
  end if;
end process
```

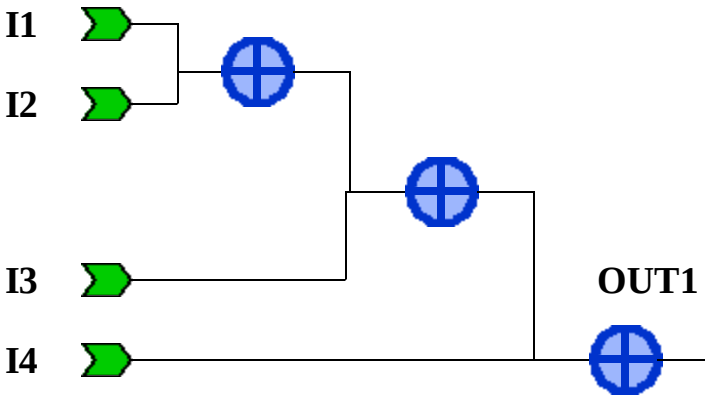
• Verilog

```
always @(posedge CLOCK or posedge RESET)
  if (RESET)
    Q = 0;
  else if (ENABLE)
    Q = D_IN;
```

Combinatorial Logic: Say what you mean

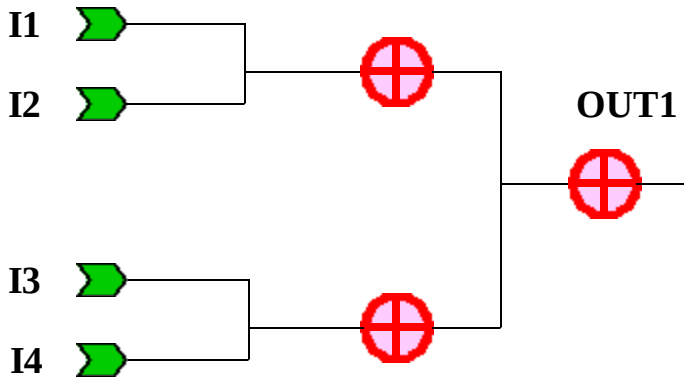
- Parentheses to control logical structure

```
-- No parentheses
OUT1 <= I1 + I2 + I3 + I4
```



3 Layers of Logic

```
-- With parentheses
OUT1 <= (I1 + I2) + (I3 + I4)
```



2 Layers of Logic

Combinatorial Logic: Say what you mean

- Don't use a process (VHDL) or always block (Verilog) when a **concurrent assignment** can be used.

Verilog

```
Always @ (A or B or C)
  begin
    if(A)
      Y = C;
    else
      Y = B;
  end
```

VHDL

```
Process(A,B,C)
  begin
    if (A = '1') then
      Y <= C;
    else
      Y <= B;
    end if;
end process;
```

A Better way...

```
Y <= A ? C : B;
```

```
Y <= C when A = '1' else B;
```

Synthesis of if-then-elsif Statement

- if-then-elseif statements *imply* priority-encoded MUXs.

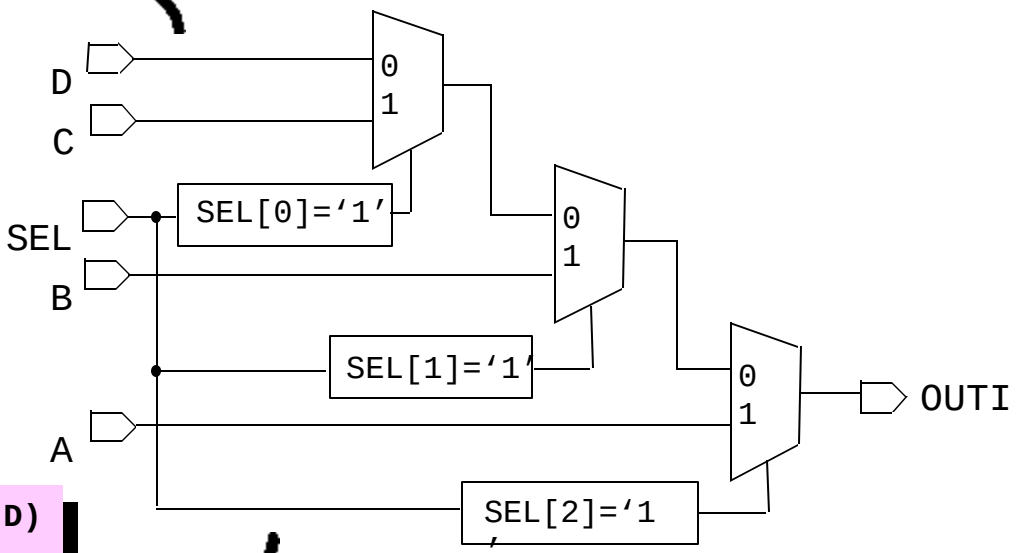
```
process (SEL, A,B,C,D) begin
   if (SEL(2) = '1') then
      OUTI <= A;
   elsif (SEL(1) = '1') then
      OUTI <= B;
   elsif (SEL(0) = '1') then
      OUTI <= C;
   else
      OUTI <= D;
   end if;
end process;
```

VHDL Code

```
always@ (SEL or A or B or C or D)
   if (SEL[2] == 1'b1)
      OUTI = A;
   else if (SEL[1] == 1'b1)
      OUTI = B;
   else if (SEL[0] == 1'b1)
      OUTI = C;
   else
      OUTI = D;
```

Verilog Code

Hardware Result



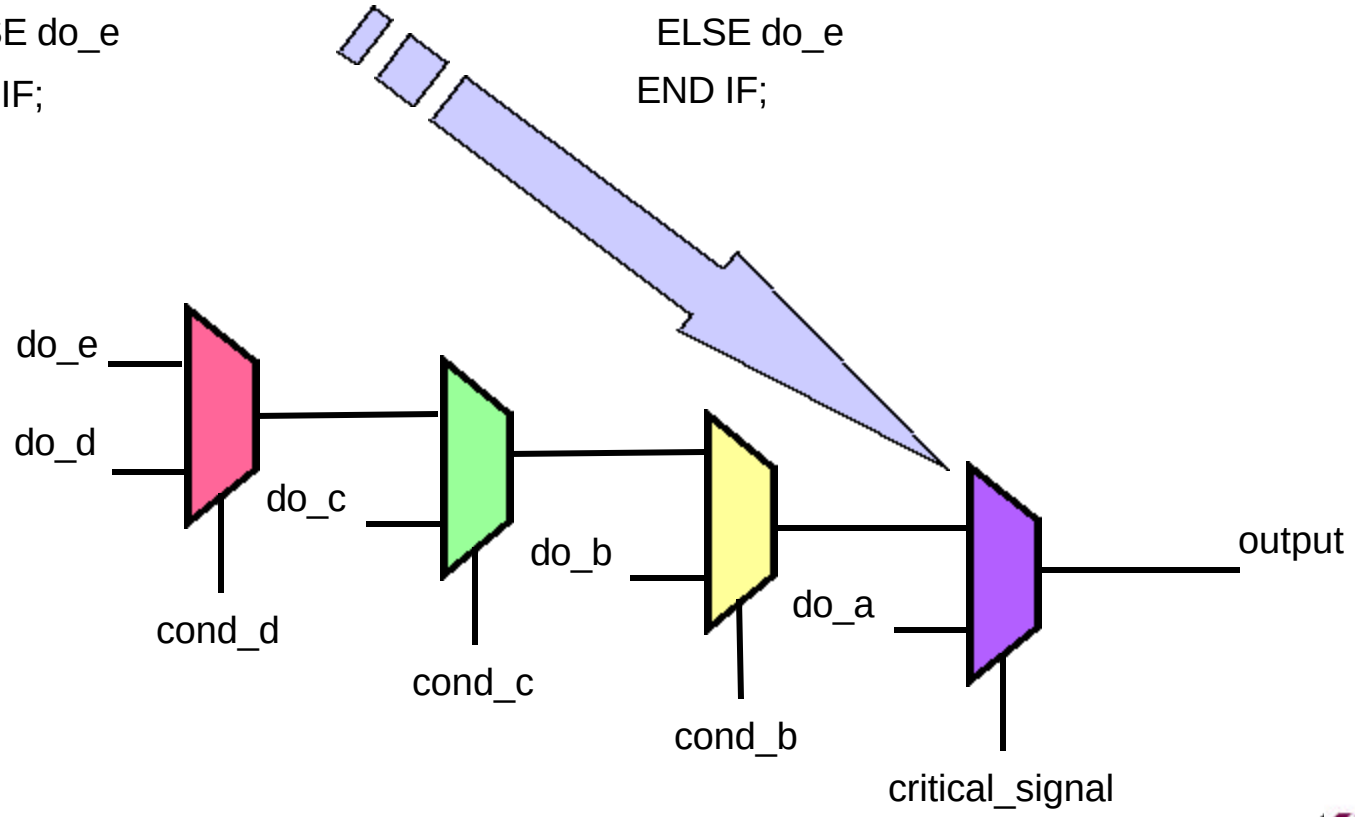
Critical Inputs in IF/ELSE Statements

- Fast critical signals

```
IF critical_signal THEN do_a
  ELSIF cond_b THEN do_b
  ELSIF cond_c THEN do_c
  ELSIF cond_d THEN do_d
  ELSE do_e
END IF;
```

- Slows down critical signals

```
IF cond_b THEN do_b
  ELSIF cond_c THEN do_c
  ELSIF cond_d THEN do_d
  ELSIF critical_signal THEN do_a
  ELSE do_e
END IF;
```



Case Statements I

- All branches of a case statement must be defined
  - enumerated states can all be defined
  - std_logic_vector has several more values than '0' or '1'
- If there are conditions that have "don't care", set output values to:
  - VHDL: Pick '1' or '0' based on logic reduction or **'-' (don't care)** for std_logic
  - Verilog: use don't care value ex: 1'bX
- Avoid using ranges in your HDL code - creates comparitors

```
case COUNTER is
  when 0 to 9 =>
    display <= SEVEN_SEG(COUNTER);
```

- Use explicit numbers - creates a decoder

```
case COUNTER is
  when 0 ! 1 ! 2 ! 3 ! 4 ! 5 ! 6 ! 7 ! 8 ! 9 =>
    DISPLAY <= SEVEN_SEG(COUNTER)
```

Case Statements II

- Case statements in combinatorial process(VHDL) or always statement (Verilog):
  - **all outputs must be defined in all branches** of the case statement to **prevent latches**.
  - use a default statement before case statement to prevent latches
- Case statements in sequential process(VHDL) or always statement (Verilog):
  - clock enables generated if outputs are not defined in all branches
  - this is not “wrong”, but might generate a long clock enable equation
  - use a default statement before case statement to prevent clock enables

Synthesis of case Statement

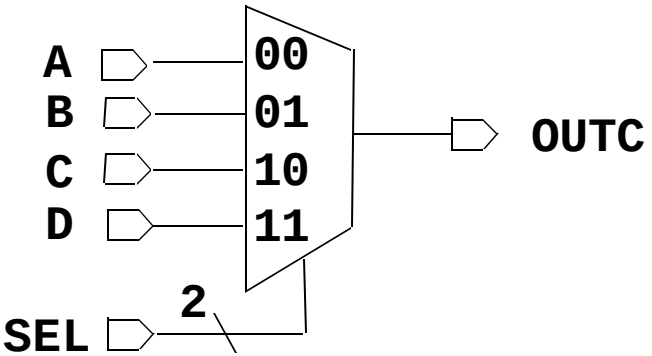
- Case statements *imply* parallel mux function.

```
process (SEL,A,B,C,D) begin
  case SEL is
    when “00” => OUTC <= A;
    when “01” => OUTC <= B;
    when “10” => OUTC <= C;
    when others => OUTC <= D;
  end case;
end process;
```

VHDL Code

```
always@(SEL or A or B or C or D)
begin
  case (SEL)
    2’b00 : OUTC = A;
    2’b01 : OUTC = B;
    2’b10 : OUTC = C;
    default : OUTC = D;
  endcase
end
```

Verilog Code



Arithmetic Operators

- Operators Inferred from HDL
  - Adder, Subtractor, AddSub (+, -)
  - Multiplier (*)
  - Comparators (>, >=, <, <=, =, /=)
  - Incrementer, Decrementer, Incdec (+1, -1)
  - Counters

Magnitude Compare

- This can be done as subtractor with sign extension on the inputs. Below is A > B, reverse the subtraction to get A < B
- The subtractor will be created using the carry chain inside the FPGA, **writing A > B in your code will be Bigger & Slower**

Verilog:

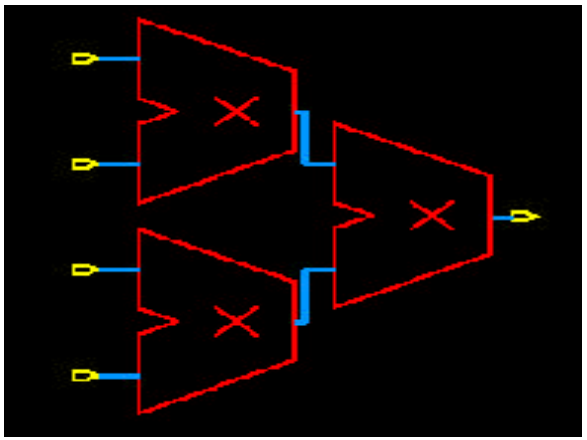
```
wire  [7:0]   A, B;
reg   [8:0]   A_ext, B_ext;
reg   [8:0]   sub;
reg           mag_comp;

always@(A or B)
   begin
      A_ext <= {A(7),A};
      B_ext <= {B(7),B};
      sub   <= A_ext - B_ext;
      mag_comp <= sub(8);
   end
```

VHDL:

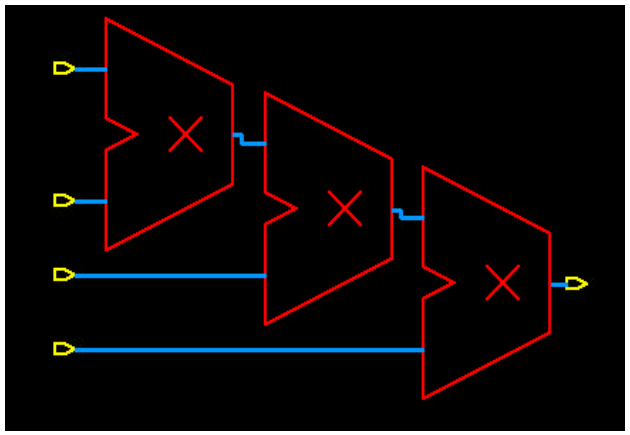
```
signal  A:       std_logic_vector (7 downto 0);
signal  B:       std_logic_vector (7 downto 0);
signal  A_ext: std_logic_vector (8 downto 0);
signal  B_ext: std_logic_vector (8 downto 0);
signal  sub:    std_logic_vector (8 downto 0);
signal  mag_comp:    std_logic;
process (A, B) begin
      A_ext <= (A(7)&A);
      B_ext <= (B(7)&B);
      sub   <= A_ext - B_ext;
      mag_comp <= sub(8);
end process;
```

Operator Balancing

- Depends on parenthesis



(A*B)*(C*D)



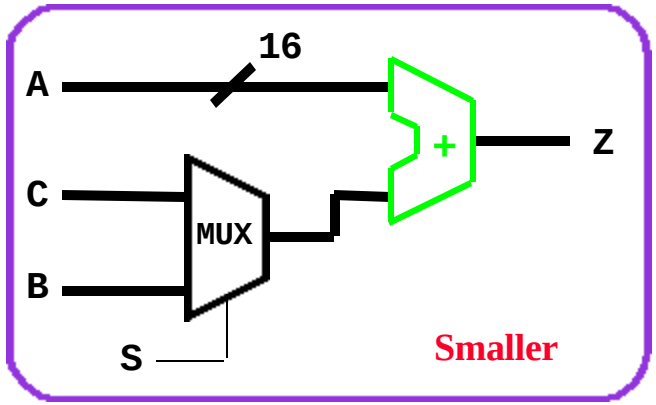
A*B*C*D

Sharing of Arithmetic Operators

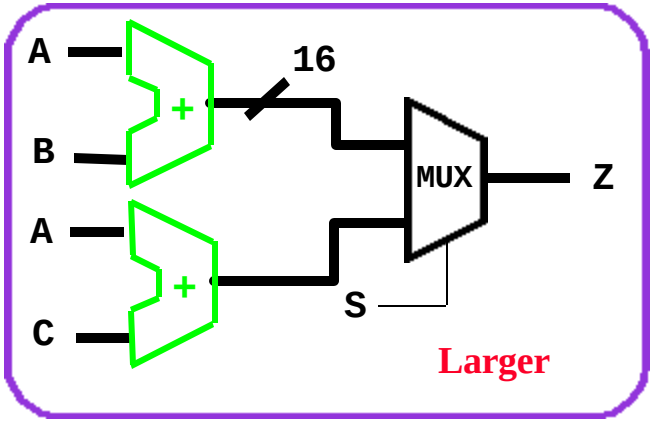
- Operators can be Shared Within:
  - — - A Process (VHDL)
  - — - An *always* Block (Verilog)

```
process (S,A,B,C)
begin
    if (S) then
          Z <= A+B;
    else
          Z <= A+C;
    end if;
end process;
```

With Sharing

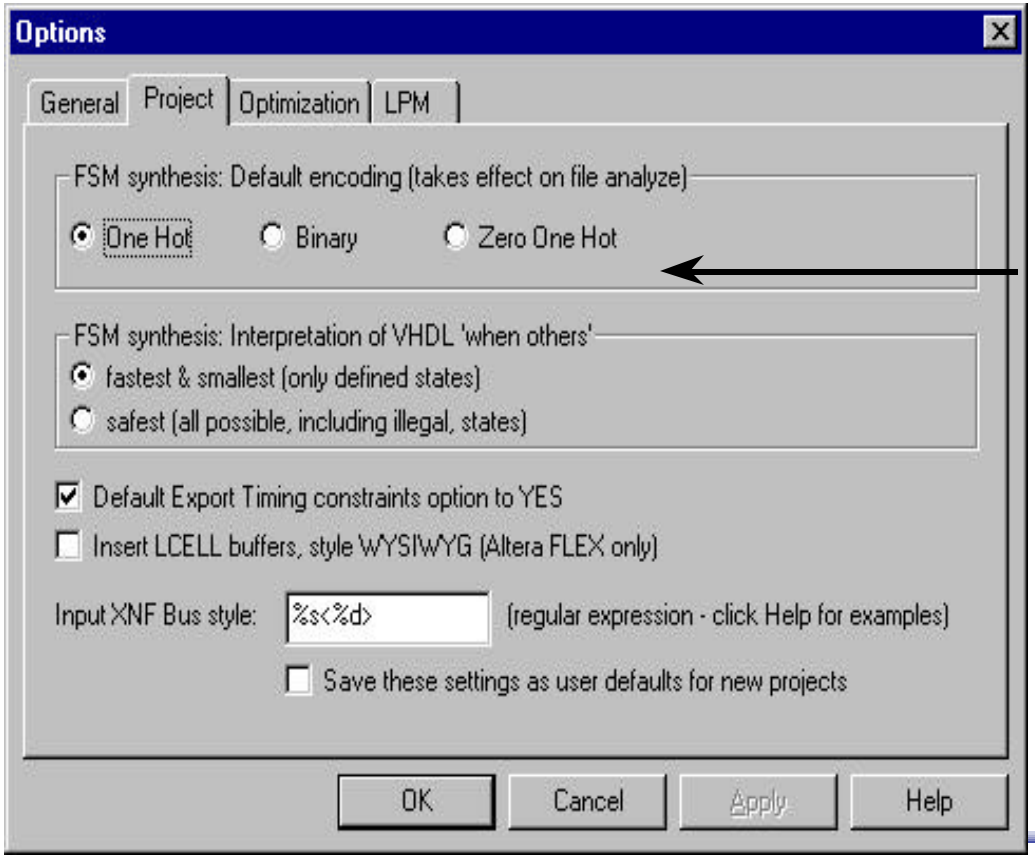


Without Sharing



FSM Encoding

- FPGA Express: manual extraction and re-encoding using FSM Compiler



You can choose one-hot, binary, or zero one-hot encoding

VHDL: Avoid Integers

- Integers are 32 bits wide
- Integer Types Default to Signed
- Use constrained std_logic_vector instead
  - e.g. std_logic_vector(7 downto 0)
  - include arithmetic packages:
  - ieee.std_logic_unsigned.all
  - ieee.std_logic_signed.all

Synopsys Translation Directives

- Useful for ignoring simulation constructs
- VHDL
  - -- pragma translate_off or -- synopsys translate_off to start
  - -- pragma translate_on or -- synopsys translate_on to finish
  - use synthesis_off/synthesis_on to check ignored code for syntax
- Verilog
  - // synopsys translate_off to start
  - // synopsys translate_on to finish
  - Turn Verilog Pre-Processor (VPP) ON to use ‘ifdef

How to implement a Synchronous Reset

- This must be declared at the top of the VHDL file as shown:

```
library synopsys;
use synopsys.attributes.all;
```

- Then attach the "sync_set_reset" attribute to the reset signal.
- Here is the section of VHDL code that will infer the synchronous reset:

```
attribute sync_set_reset of RESET: signal is "true";
```

VHDL Example

```
library synopsys;
use synopsys.attributes.all;
....
architecture COUNT_ARCH of COUNTER is
        signal COUNT: STD_LOGIC_VECTOR (7 downto 0);
        attribute sync_set_reset of RESET: signal is "true";
begin
process (CLK, RESET)
begin
if (CLK'event and CLK='1') then
  if (RESET='1') then
    COUNT <= "00000000";
  else
    COUNT <= COUNT + 1;
  end if;
end if;
end process;
```

Verilog Example

- No library needs to be defined for Verilog.

```
//synopsys sync_set_reset "RESET"

always @(posedge CLK)
  if (RESET)
    COUNT = 8'b00000000;
  else
    COUNT = COUNT + 1;
```