VHDL Synthesis



- Synthesizable / Non-Synthesizable Code
- Translation to Hardware
- Simulation vs. Synthesis
- Synthesis: Hints, Guidelines
- Style Guidelines.

References

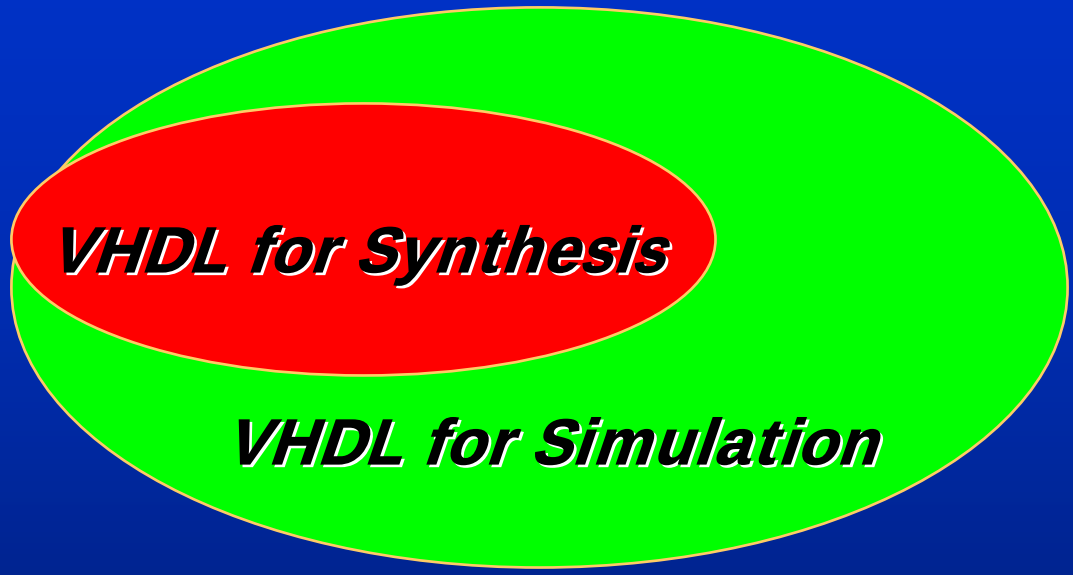
- Peter Ashenden's "The designer's guide to VHDL, 2nd edition", Morgan Kaufmann publishers, 2002
- Douglas Perry, "VHDL", 3rd Edition, McGraw Hill.
- Introductory VHDL: From Simulation to Synthesis, Sudhakar Yalamanchili, Prentice Hall, 2001.
- "VHDL Guidelines for Synthesis", Siemens Semiconductor Group

Execution Models for VHDL Programs

- Two classes of execution models govern the application of VHDL programs

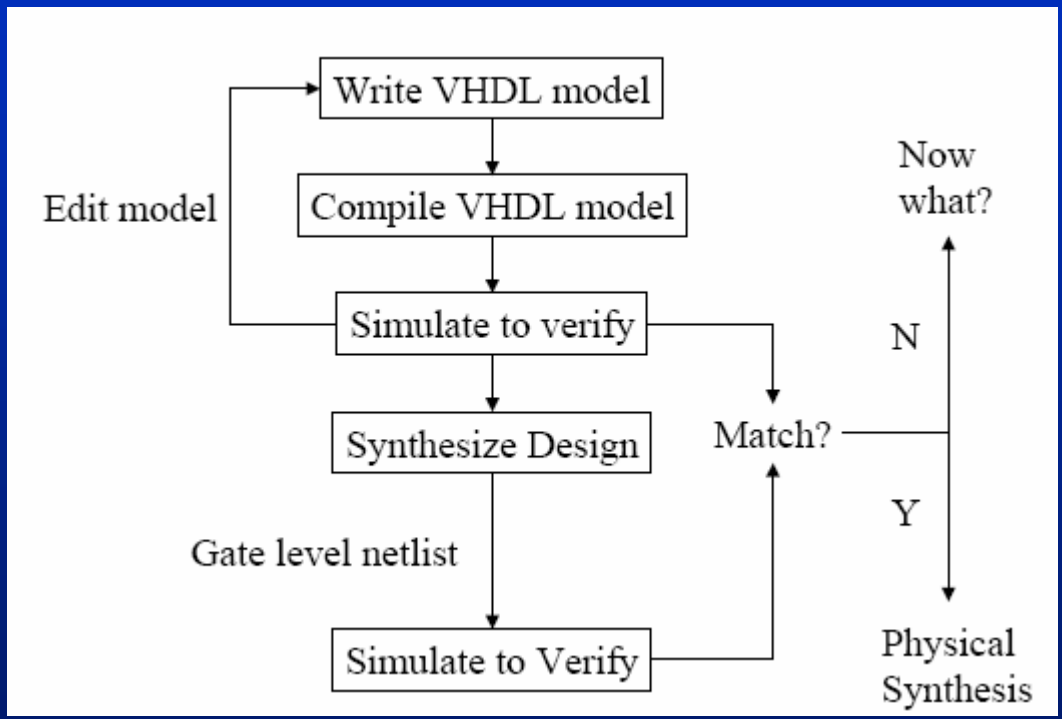
- For ***Simulation***
  - Discrete event simulation
  - Understanding is invaluable in debugging programs
- For ***Synthesis***
  - Hardware inference
  - The resulting circuit is a function of the building blocks used for implementation
    - » Primitives: NAND vs. NOR
    - » Cost/performance

VHDL for Synthesis

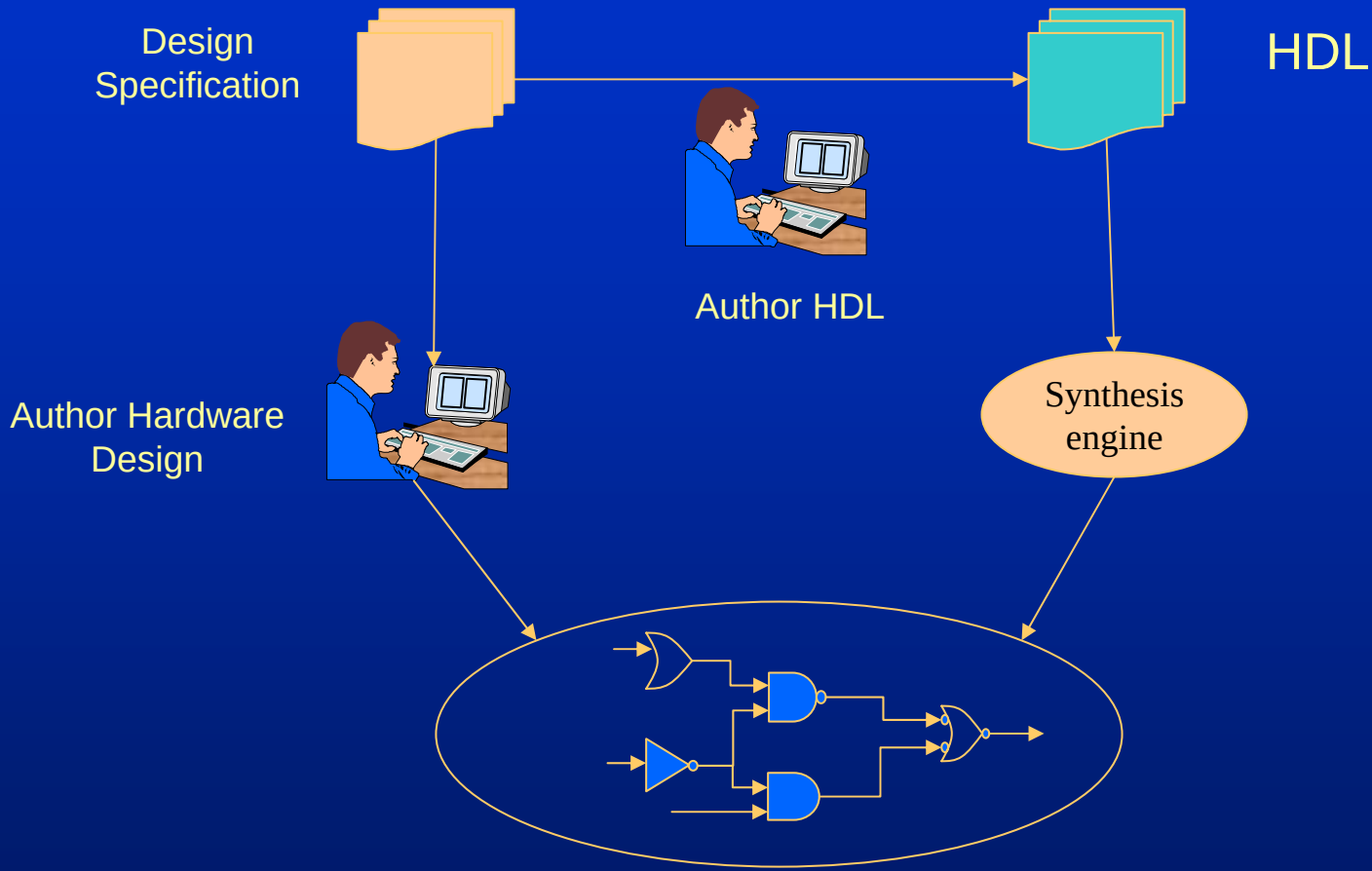


- Only a subset of the VHDL language is synthesizable
- The VHDL subset that is synthesizable is tool specific!
  - Do not expect your VHDL description to be synthesizable with another tool.
- Understand what is meant for simulation versus what is meant for producing hardware (i.e. synthesis)

Design Flow for VHDL Synthesis



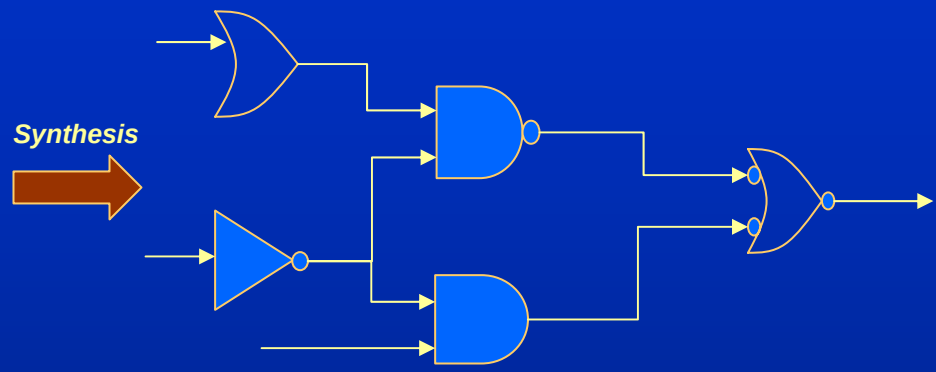
Synthesis and Hardware Inference



- Both processes can produce ***very different*** results! Why?

VHDL Synthesis

```
entity my_ckt is
port(x, y:in bit;
        z: out bit)
end entity my_ckt;
architecture behavioral of my_ckt is
begin
--
-- process code here
--
end architecture behavioral;
```



- Inferring hardware from sequential code in processes
  - Multiple alternative implementations
  - Metrics for selecting the right implementation
- Coding style can “help” the compiler

Synthesis

- Hardware implementation of VHDL code depends
  - Coding convention
  - Fitter technology
  - Optimization option
  - Nature of application
- Not all designs are synthesizable: several constructs defined in the language cannot be synthesized!

Synthesis: Hardware Translation

- How does a data type translate to hardware?
  - Integer, Real, Boolean, ...
- How are literals mapped to logic values?
- How do data objects map to hardware?
  - Signals, variables, constants, ...
- How are statements translated to hardware?
  - Assignments, operators, ...
- How do some constructs differ when translating code to hardware?
  - If-Else, Case, For, While, ...

Non Synthesizable Subset

- There are a number of constructs that cannot be synthesized into hardware
  - File operations including text I/O
  - Assertion statements
  - Loops with no bound (infinite while loop)
  - Functions with many constant parameters
  - Real data types are not supported

VHDL Hardware Correspondence

VHDL Construct	Hardware
Variables and Signals	Flip-flops, latches, wires
Arithmetic operators (+,-,*)	Adder, ALU, multiplier..
Logic operators	Gates
Relational operators	Comparator, ALU
Control Constructs (for,if-then, case,.....)	Decoders, Multiplexers, priority encoder, ..
Hierarchy Description	Hierarchical Hardware
Resolution Functions	Tri-state logic, wire-and, wire-or

Simple Assignment Statements

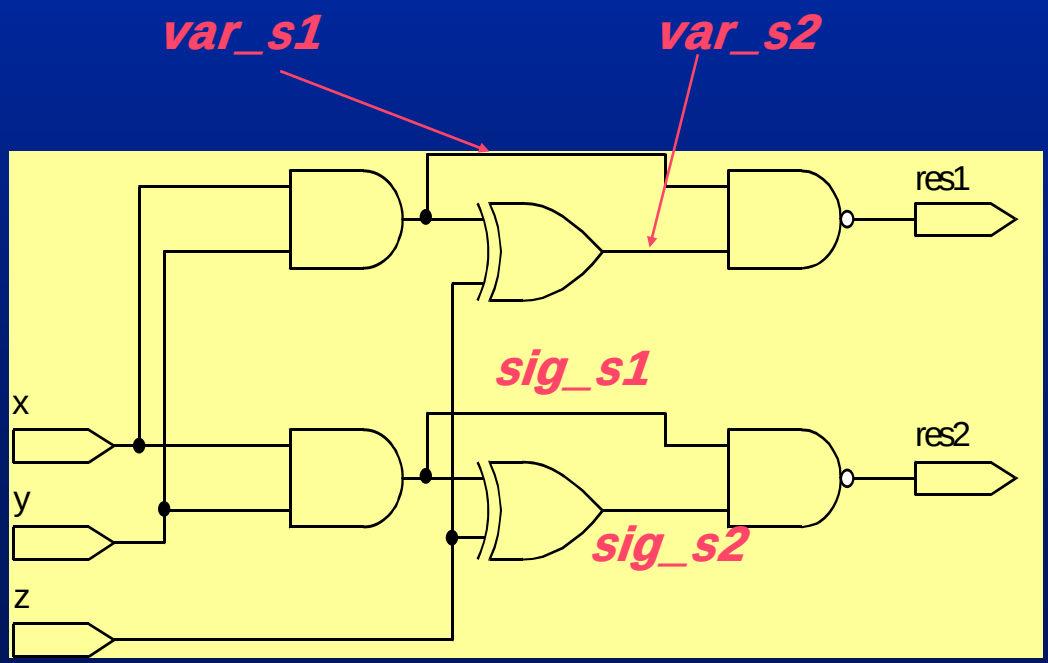
```
architecture behavior of stover is
signal sig_s1, sig_s2: std_logic;
begin
proc1: process (x, y, z) is
        -- Process 1 using variables
variable var_s1, var_s2: std_logic;
begin
   var_s1:= x and y;
   var_s2:= var_s1 xor z;
   res1 <= var_s1 and var_s2;
end process proc1;
```

```
proc2: process (x, y, z)
        -- Process 2 using signals

begin
   sig_s1 <= x and y;
   sig_s2 <= sig_s1 xor z;
   res2 <= sig_s1 and sig_s2;
end process proc2;

end architecture behavior;
```

- Simulation mismatch
  - synthesis collapses multiple simulation cycles
- Dependence analysis



Chain of Dependencies

```
proc2: process (x, y, z)
        -- Process 2 using signals

begin
   sig_s1 <= x and y;
   sig_s2 <= sig_s1 xor z;
   sig_s3 <= sig_s2 and sig_s0;
   sig_s4 <= sig_s3 and y;
end process proc2;

end architecture behavior;
```

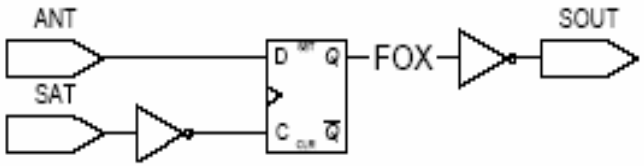
- The computation of the value of sig_s4 uses the value of signals s1 through the computation of several intermediate signals.
- The result is long signal paths in the synthesized logic!!

IF: Inferring a Latch

```
signal SAT, SOUT, ANT : BIT;

LATCH: process(SAT,ANT)
  variable FOX : BIT;
begin
  if SAT = '0' then
    FOX := ANT;
  end if;

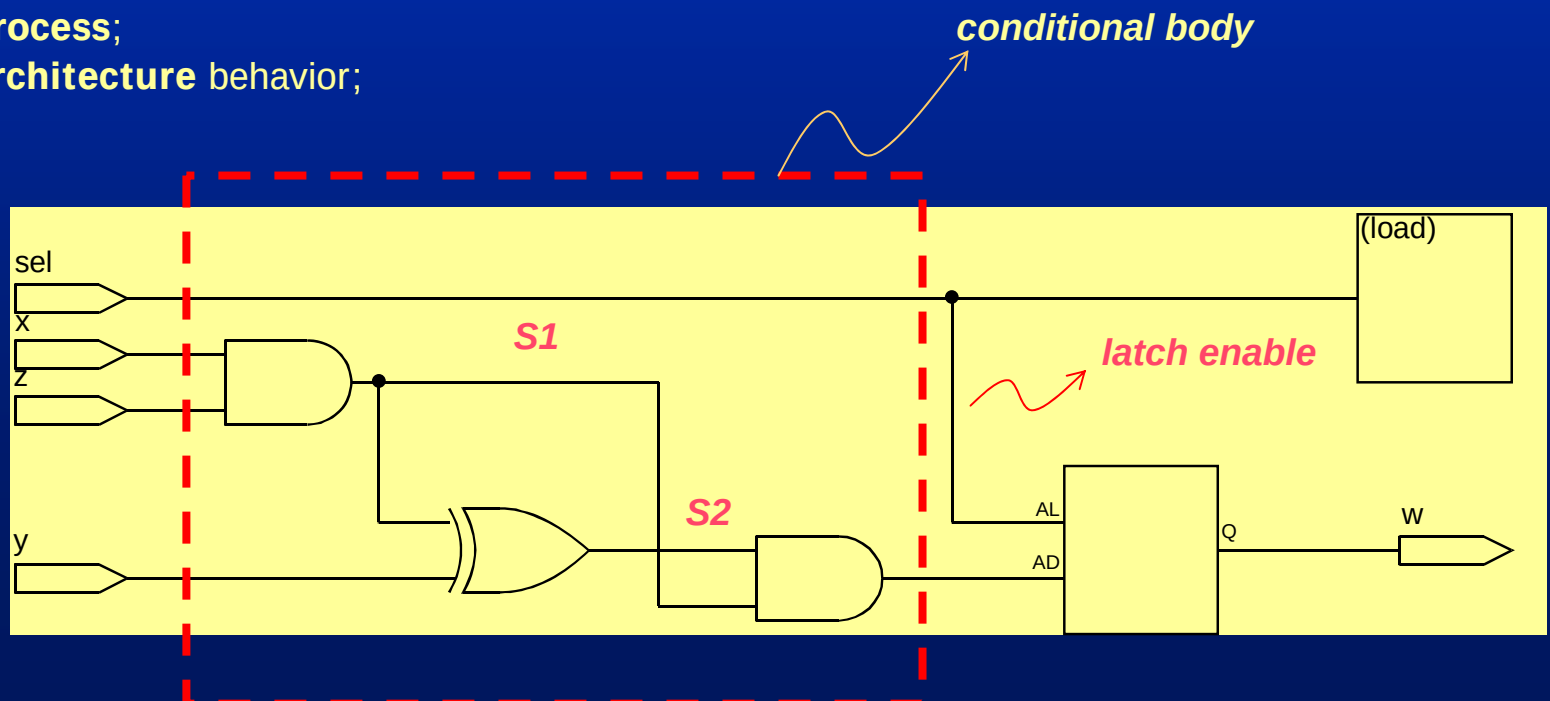
  SOUT <= not FOX;
end process;
```



- Wait for event on SAT or ANT
- Latch inferred due to if statement
  - Clk is if condition (SAT = '0')
  - Value is changed in then part
  - Value is held for else part

Conditional Statements

```
architecture behavior of inflate is
begin
process (x, y, z, sel) is
variable s1, s2: std_logic;
begin
if (sel = '1')then
    s1:= x and z;
    s2:= s1 xor y;
    w <= s2 and s1;    -- w gets a value only conditionally
end if;                -- hence a latch is inferred
end process;
end architecture behavior;
```



Avoiding Latch Inference

- Ensure every path computes a value for every signal
  - Presence of the else branch
  - Nested structures
  - if-then-elsif structure
- Use initial values
- Basic principle: ensure that every combination of input signal values leads to a computation of a value for every output signal



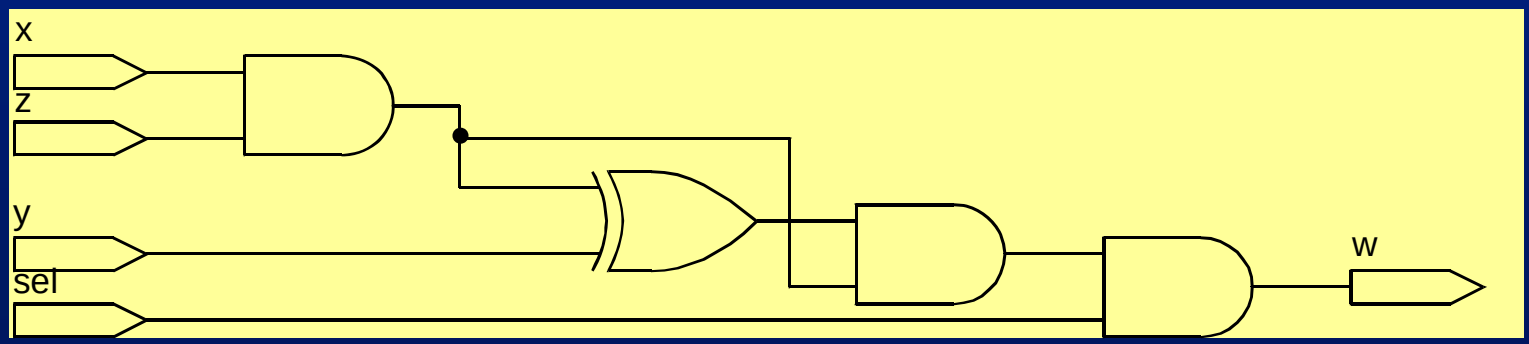
combinational logic

Revisit the Example

```
architecture behavior of inflate is
begin
process (x, y, z, sel) is
variable s1, s2: std_logic;
begin
   w <= '0';              -- output signal set to a default value to avoid latch inference
  if (sel = '1') then
      s1:= x and z;       -- body generates combinational logic
      s2:= s1 xor y;
     w <= s2 and s1;
 end if;

end process;
end architecture behavior;
```

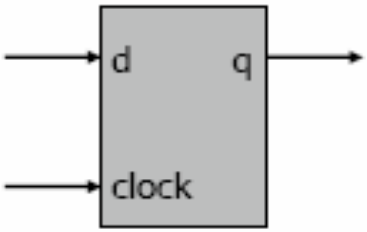
No Latches



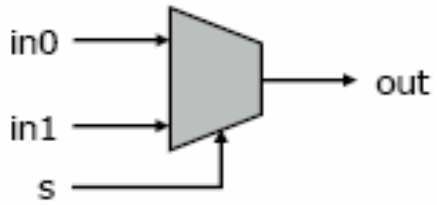
Muxes vs. Latches

- **Both** latches and multiplexers are described through `IF` statements

```
latch : PROCESS ( clock , d )
BEGIN
   IF (clock = '1') THEN
      q <= d;
   END IF;
END PROCESS latch;
```




```
mux : PROCESS ( s , in0 , in1 )
BEGIN
   IF (s = '1') THEN
      out <= in1;
   ELSE
      out <= in0;
   END IF;
END PROCESS mux;
```

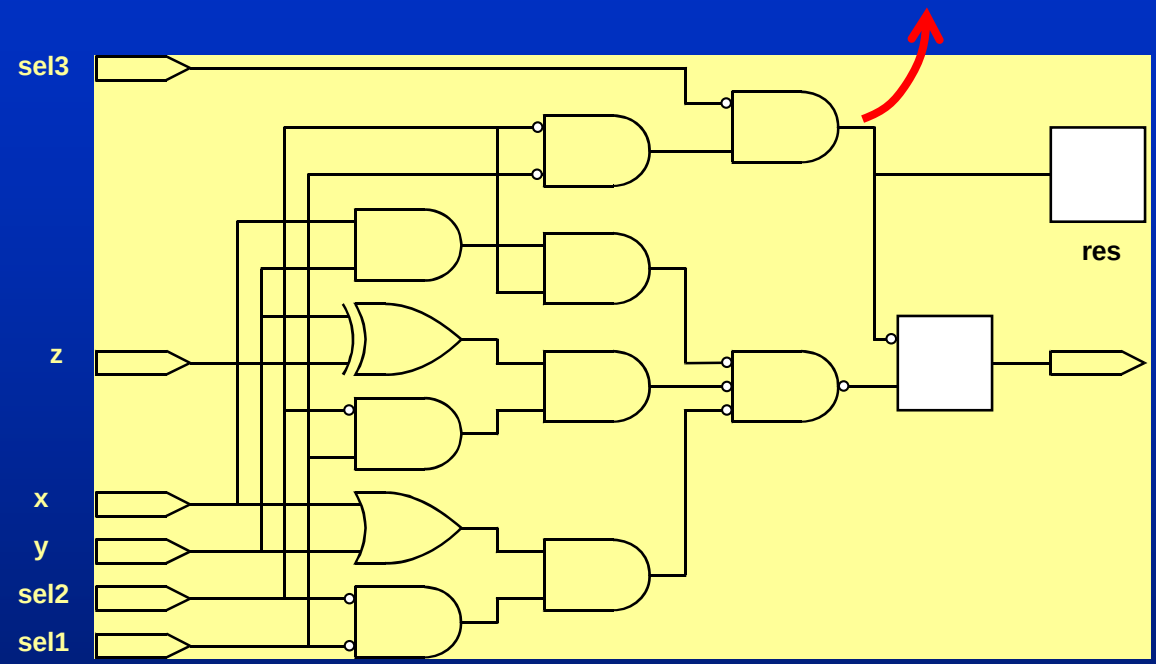


- The important difference is that in muxes there is **always a new value assigned** to signals or variables, whereas in latches sometimes nothing is assigned and they **keep their old value** (→ memory!)

Nested Constructs

```
architecture behavior of nested_if is
begin
process (x,y,z,sel1,sel2,sel3)
begin
  if (sel1 = '1')then
        res <= x and y;
  elsif (sel2 ='1') then
        res <= y xor z;
  elsif (sel3 ='1') then
       res <= x or y;
  end if;
end process;
```

Condition under which a latch is inferred
When sel1,sel2,sel3 are all zero

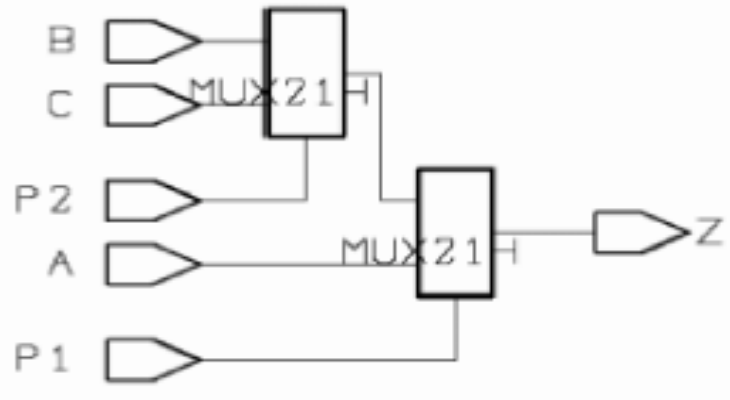


Add an else to avoid latch inferring

- Latch inference due to the absence of the last “else”
- Latch inference due to placement in higher level blocks
- Priority ordering of the computations

Synthesis of an IF Statement

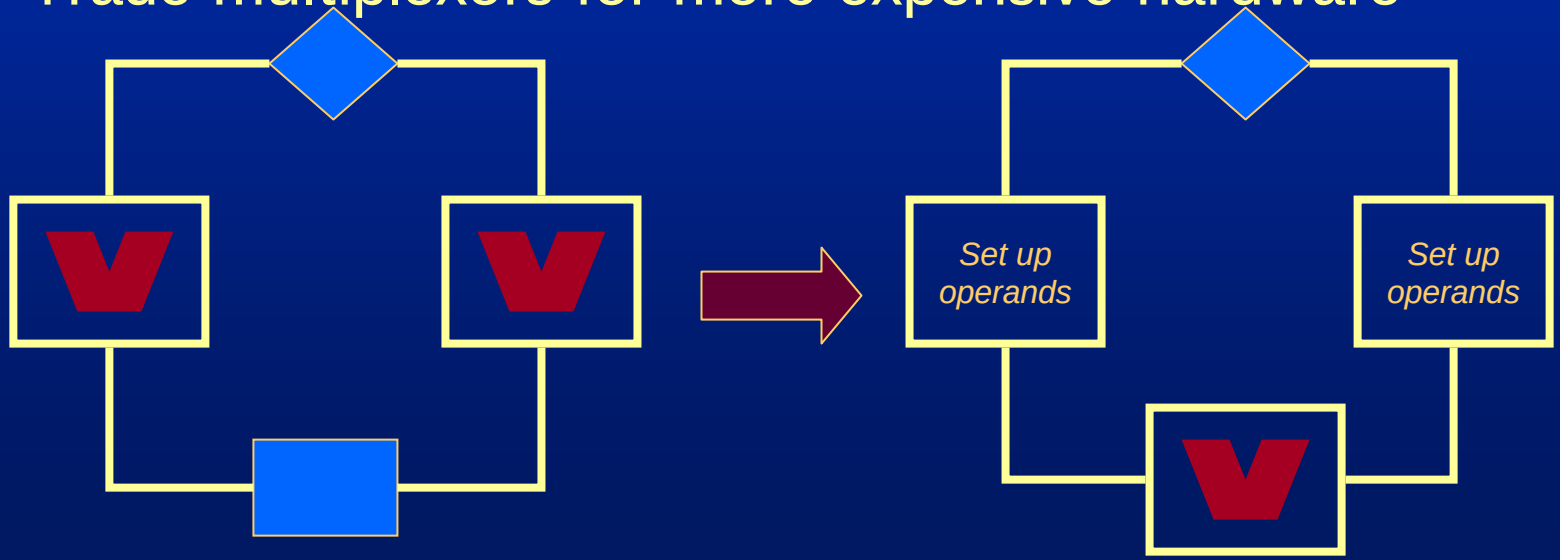
```
signal A, B, C, P1, P2, Z: BIT;
if (P1 = '1') then
   Z <= A;
elsif (P2 = '0') then
   Z <= B;
else
   Z <= C;
end if;
```



The IF-statement implies a priority!

Efficiency Considerations

- Now that we can control latch inferencing what about circuit size and speed?
- Move common operations (hardware) out of the branches
  - Good programming practice in general
  - Trade multiplexors for more expensive hardware



Efficiency Considerations

```
entity inference is
port (sel : in std_logic;
x, y, z: in std_logic;
w: out std_logic );
end entity inference;

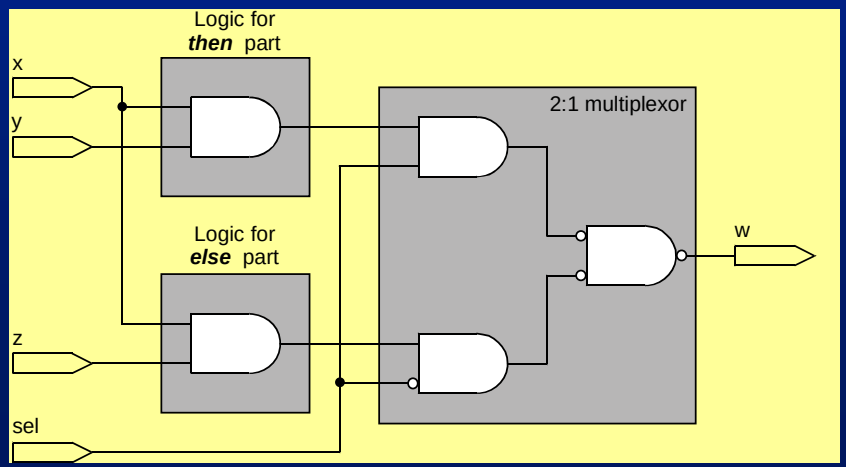
architecture behavior of inference is
begin
process (x, y, z, sel) is
begin
   if (sel = '1')then
       w <= x and y;
  else
      w <= x and z;
 end if;
end process;
end architecture behavior;
```

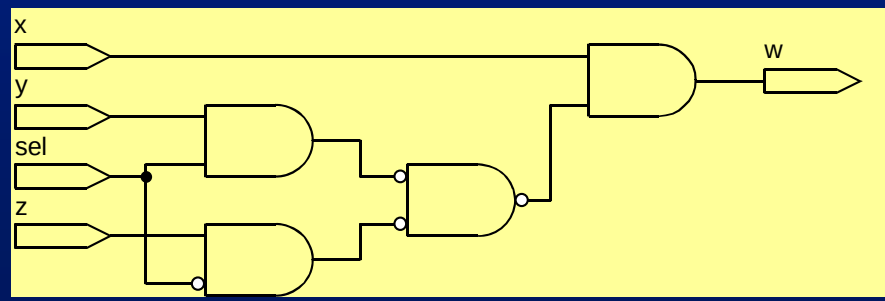
- Eliminated the latches
- Computation (logical AND) preceded by operand selection (mux)

```
entity inference is
port (sel : in std_logic;
x, y, z: in std_logic;
w: out std_logic );
end inference;

architecture behavior of inference is
begin
process (x, y, z, sel)
variable right: std_logic;
begin
   if (sel = '1')then
       right:= y;
  else
      right:= z;
 end if;
w <= x and right;

end process;
end behavior;
```





Signal Selection in VHDL

- **Signal selection in VHDL for some FPGA targets could be different from that used for ASICs, as shown below:**

General signal selection in VHDL

```
if (sel ='0') then
        output <= signal_0;
else
        output <= signal_1;
end if;
```

Signal selection is implemented here using muxes

In some FPGAs, muxes can be expensive, and tri-state buffers are cheaper

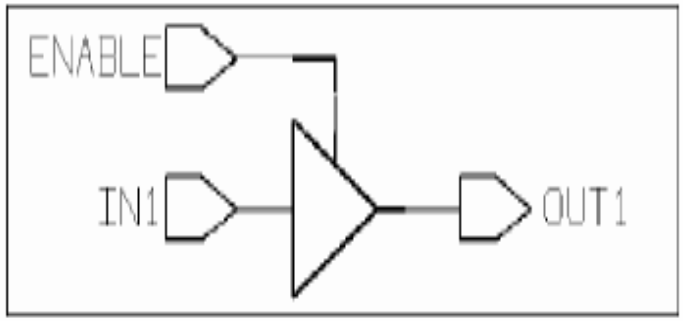
Preferred signal selection for FPGAs using tri-state busses

```
output <= signal_0 when (EnA = '1') else 'Z';
output <= signal_1 when (EnB = '1') else 'Z';
```

Synthesis of 3-state Driver

```
entity three_state is
   port(IN1, ENABLE : in std_logic;
        OUT1 : out std_logic );
   end;
architecture rtl of three_state is
begin
   process (IN1, ENABLE) begin
     if (ENABLE = '1') then
       OUT1 <= IN1;
     else
       OUT1 <= 'Z'; -- assigns high-impedance state
     end if;
   end process;
end rtl;
```





Three-State Device Name	Type	MB
OUT1_tri	Three-State Buffer	N

Comparison Logic and Effects

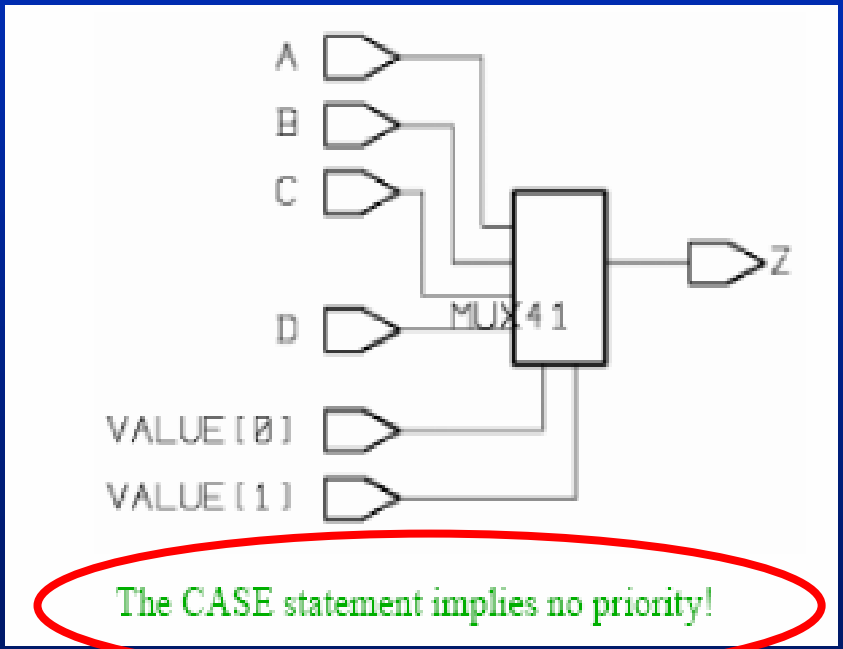
```
architecture behavior of stover is
begin
process (x, y, z) is
begin
   if (sel = "– 0") then        -- the symbol "–" represents the don't care
        w <= x nor y;           -- for std_logic types
  else
       w <= x nor z;
end if;

end process;
end architecture behavior;
```

- Comparison always returns false - the "then" branch is never taken
- Synthesis returns the combinational logic for the "else" branch – a single NOR gate

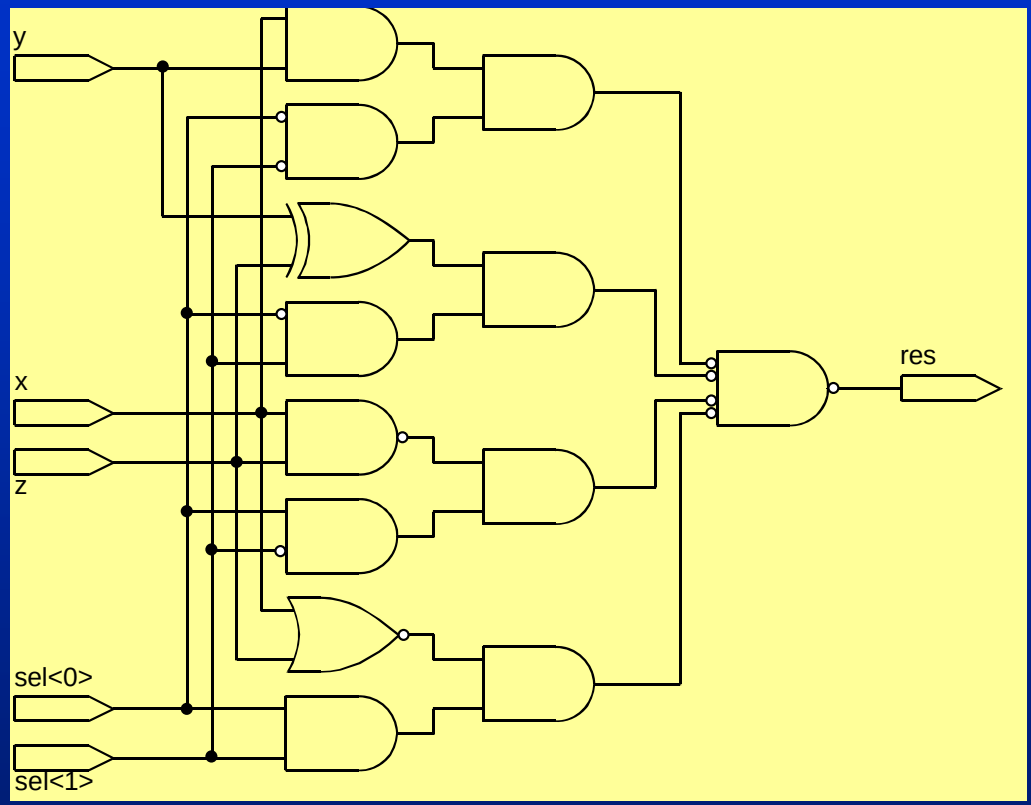
Synthesis of Case Statement

```
type ENUM is (PICK_A, PICK_B, PICK_C, PICK_D);
signal VALUE: ENUM;
signal A, B, C, D, Z: BIT;
...
case VALUE is
  when PICK_A => Z <= A;
  when PICK_B => Z <= B;
  when PICK_C => Z <= C;
  when PICK_D => Z <= D;
end case;
```



Case Statements

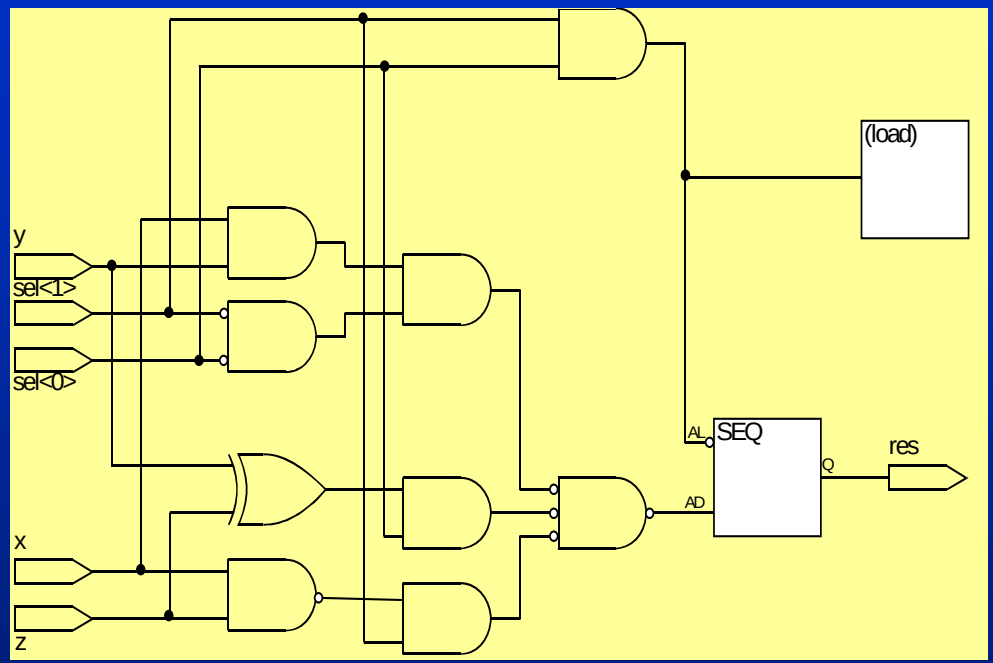
```
architecture behavior of case_st is
begin
process (x,y,z,sel) is
begin
case sel is
    when 0 => res <= x and y;
    when 1 => res <= y xor z;
    when 2 => res <= x nand z;
    when others => res <= x nor z;
end case;
end process;
end architecture behavior;
```



- Synthesis of a multiplexor
- Latch inference intuition applies
- Difference with nested if-then-elsif → latter synthesizes priority logic

CASE Statements (cont.)

```
architecture behavior of case_ex is
begin
process (x,y,z,sel) is
begin
case sel is
  when 0 => res <= x and y;
  when 1 => res <= y xor z;
  when 2 => res <= x nand z;
  when others => null;
    -- in this case the value of res
    -- remains unaltered
end case;
end process;
end architecture behavior;
```



- Use of the “null” statement and latch inference
- How do we avoid a latch in this case?

If/Case Sequential Statements

- Incomplete **if** statement implies a flip-flop or latch primitive.
- Incomplete assignments within **case** statements can also result in latches using combinational feedback rather than latch primitives.
- Complete **if** statement specifies (using an **else** clause) a combinational function.

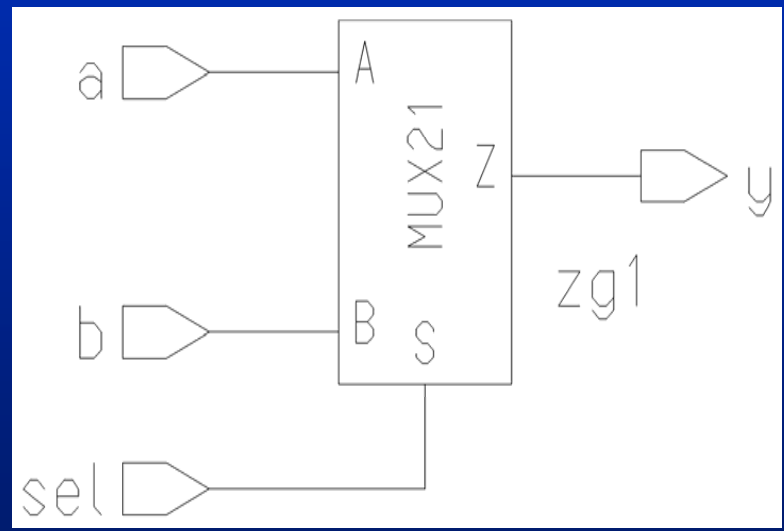
Concurrent When Statement

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

ENTITY mux2 is
  PORT(a   : IN  std_logic;
       b   : IN  std_logic;
       sel : IN std_logic;
       y   : OUT std_logic);
END mux2;

ARCHITECTURE behavior OF mux2 IS

  BEGIN
   y <= a WHEN (sel = '0') ELSE
        b WHEN (sel = '1') ELSE
        'X';
END behavior;
```

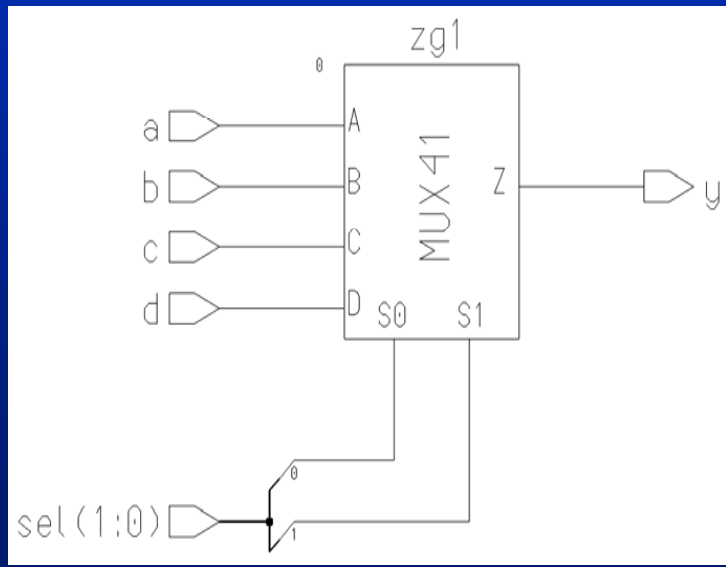


Concurrent With Statements

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

ENTITY mux4 is
  PORT(a   : IN  std_logic;
       b   : IN  std_logic;
       c   : IN  std_logic;
       d   : IN  std_logic;
       sel : IN std_logic_vector(1 DOWNTO 0);
       y   : OUT std_logic);
END mux4;

ARCHITECTURE behavior OF mux4 IS
  BEGIN
    WITH sel SELECT
      y <= a WHEN "00",
           b WHEN "01",
           c WHEN "10",
           d WHEN "11",
         'X' WHEN OTHERS;
END behavior;
```



Case vs. If Statements

- Any **case** statement does have an equivalent (functionally) formulation in an **if-then-elsif** form.
- However the **if-then-else** must produce ***priority*** logic to maintain the priority order implicit in the nesting of the branches.
- This is not the situation for the **case** statement.
- As a result we would expect that the **case** statement is ***a more efficient*** alternative when there is no priority ordering among the alternatives.

Loop Statements

- What/How much hardware should be generated?
- Most commonly supported is the *for loop*
  - Number of iterations is known a priori
  - **Loop is unrolled** and optimized as a sequence of sequential statements

```
for N in 3 downto 1 loop
  shift_reg (n) <= shift_reg (n-1);
end loop;
```



```
shift_reg (3) <= shift_reg (2);
shift_reg (2) <= shift_reg (1);
shift_reg (1) <= shift_reg (0);
```

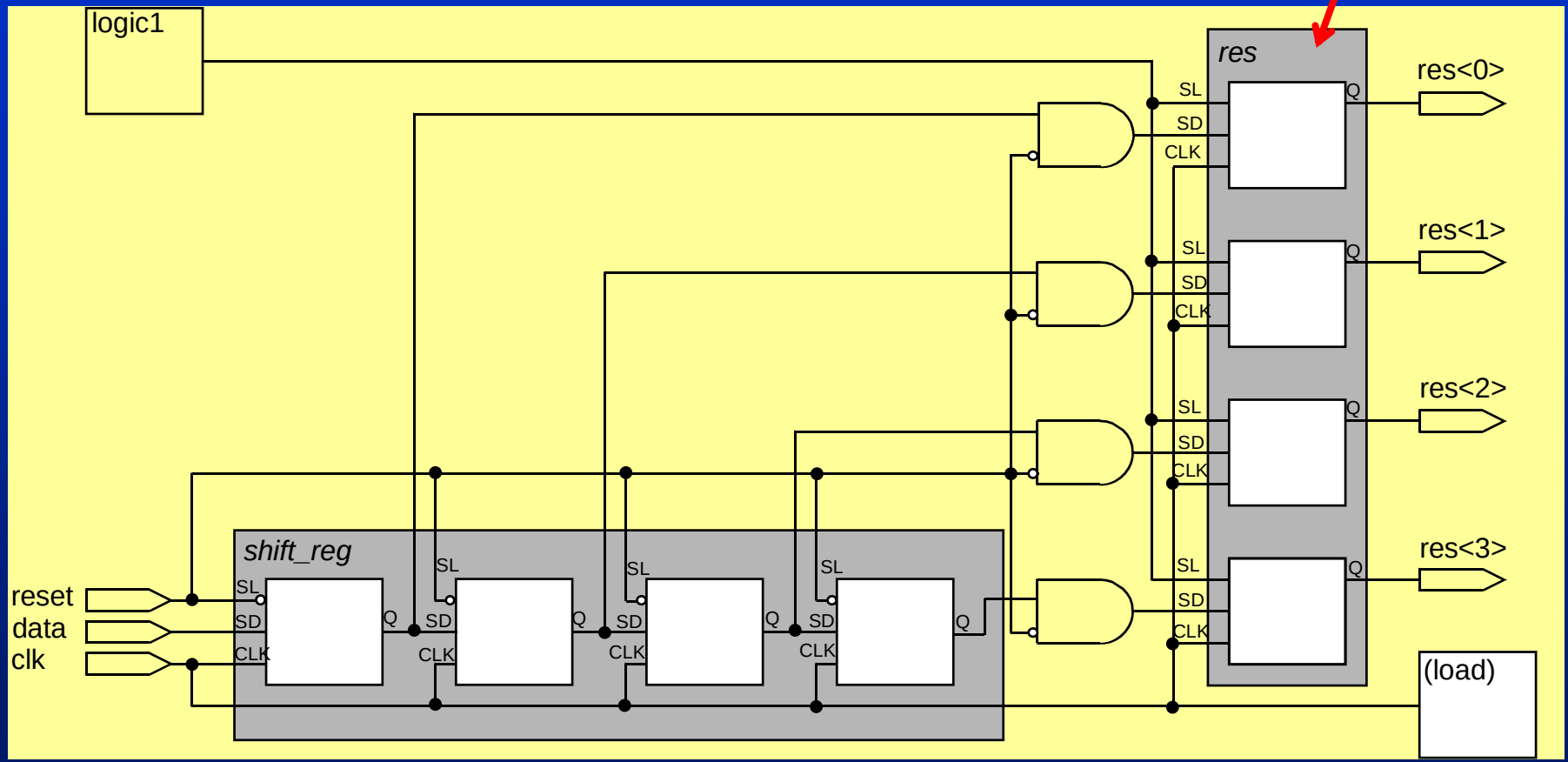
- Dependencies within and across iterations
  - Cross iteration dependencies synthesize to long signal paths

Loop Statements: Example

```
architecture behavior of iteration is
signal shift_reg : std_logic_vector(3 downto 0);
begin
process (clk, reset, data) is
        begin
        if (rising_edge(clk)) then
            if (reset = '1') then
                    res <= "0000";
            else
                    for n in 3 downto 1 loop
                            shift_reg(n) <= shift_reg(n-1);
                    end loop;
                    shift_reg(0) <= data;
                    res <= shift_reg;
            end if;
        end if;
end process;
end architecture behavior;
```

- Call to rising_edge(clk) produces edge triggered flip flops
- Loop leads to sequence of dependent assignment statements

Loop Statements: Example



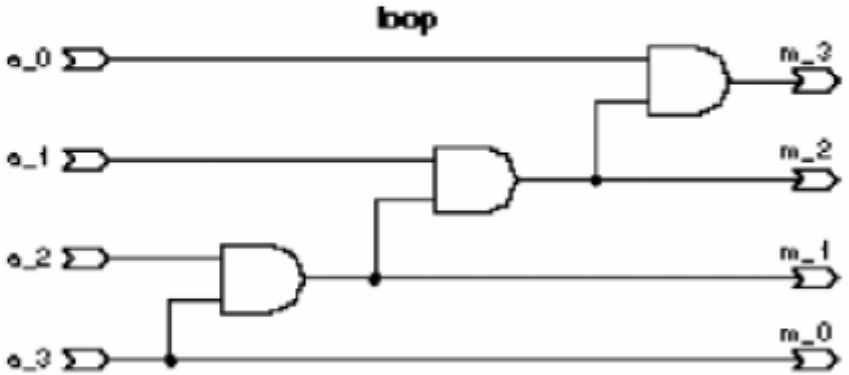
While Statements

- While statement
  - Supported by the synthesizer only when the termination can be determined at the time of synthesis.
  - Unconditioned loops are not synthesizable.

While loop: unrolling

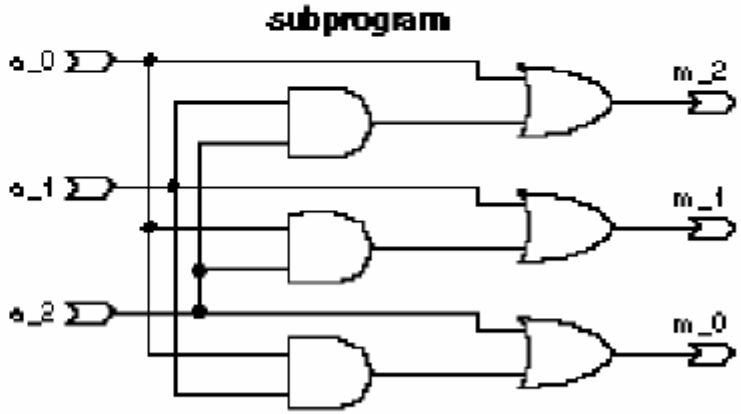
```
entity while_stmt is
    port (a: in bit_vector (0 to 3);
          m: out bit_vector (0 to 3));
end while_stmt;

architecture example of while_stmt is
begin
    process (a)
        variable b: bit;
        variable i: integer;
    begin
        i := 0;
        while i < 4 loop
            b := a(3-i) and b;
            m(i) <= b;
        end loop;
    end process;
end example;
```



Replicated Logic: Procedures

```
entity proc is
    port (a: in bit_vector (0 to 2);
          m: out bit_vector (0 to 2));
end proc;
architecture example of subprograms is
    procedure simple (w, x, y: in bit; z: out bit) is
    begin
        z <= (w and x) or y;
    end;
begin
    process (a)
    begin
        simple(a(0), a(1), a(2), m(0));
        simple(a(2), a(0), a(1), m(1));
        simple(a(1), a(2), a(0), m(2));
    end process;
end example;
```



Infering latches/Flipflops

- VHDL has two kinds of data storage
  - Signals
  - Variables
- Hardware value containers
  - Wires (direct connection)
  - Flip-flops (edge-triggered)
  - Latches (level-sensitive)
- Synthesis infers hardware elements from VHDL description.

Inference Using Signals/Variables

- *Signals* correspond to hardware objects.
  - Thus signals are synthesized into wires or storage devices such as latches and flip-flops.
- *Variables* are programming objects that are used (typically) to compute the values of signals.
  - The dependencies between variables are used for the purpose of propagating values in the computation of some signal values.
  - Thus you do not expect all variables to be synthesized into hardware, but in some instance they may be represented in hardware.

Inference Using Signals vs. Variables

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

entity sig_var is
port (sel : in std_logic;
x, y, z: in std_logic;
v, w: out std_logic );
end entity sig_var;

architecture behavior of sig_var is
signal sig_s1 : std_logic;
begin
process (x, y, z, sel) is
variable var_s1: std_logic;
```

Latch inferred

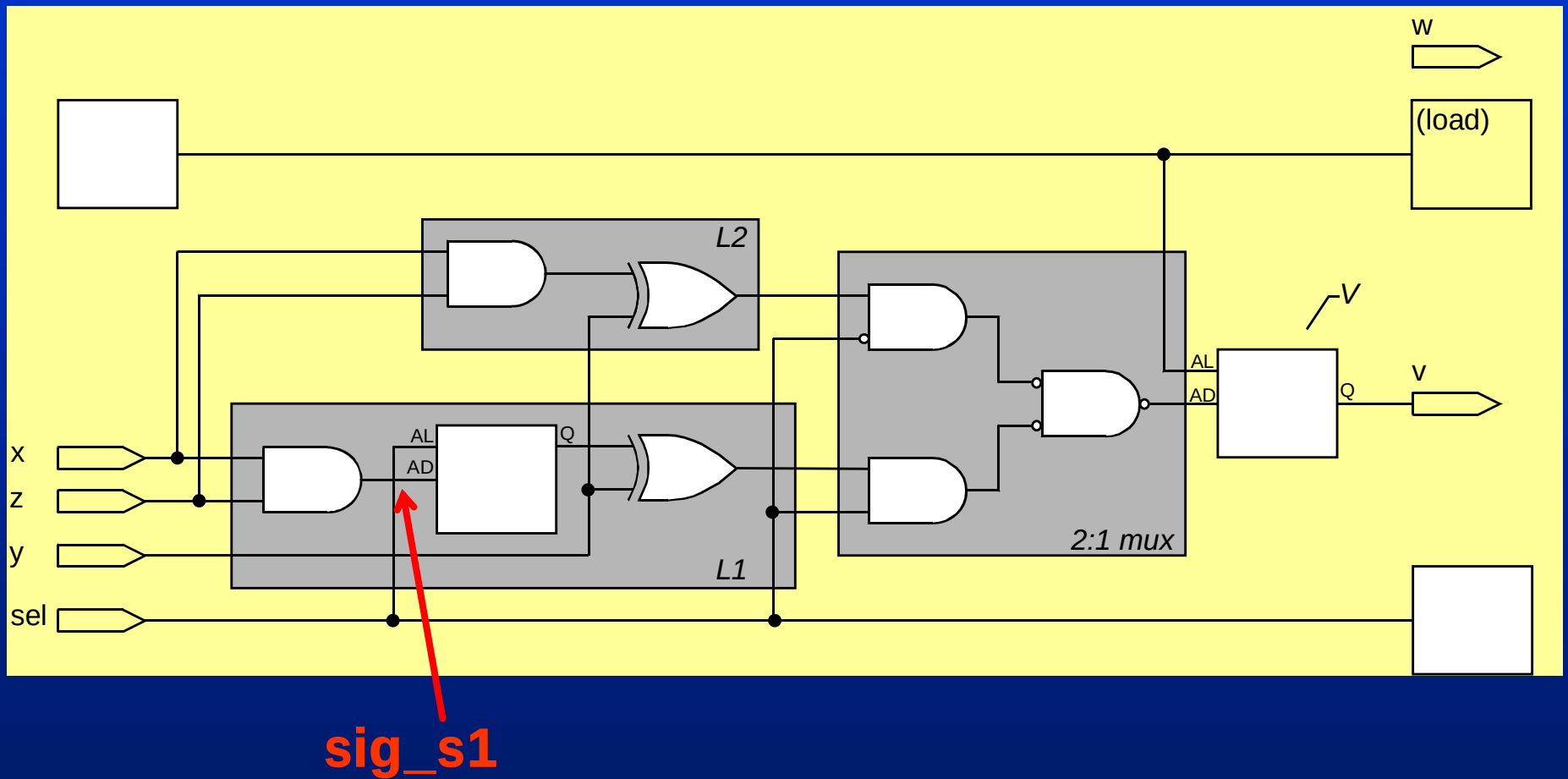
```
L1: if (sel = '1') then
sig_s1   <= x and z;
v        <= sig_s1 xor y;
end if;
```

No Latch inferred

```
L2: if (sel = '0') then
var_s1   := x and z;
v        <= var_s1 xor y;
end if;
end process;
end architecture behavior;
```

- Variable synthesized to a wire vs. signal synthesized to a latch
- Why is a latch inferred at all since all execution paths are covered? (sel = 0, 1, x, ...)

Inference Using Signals vs. Variables

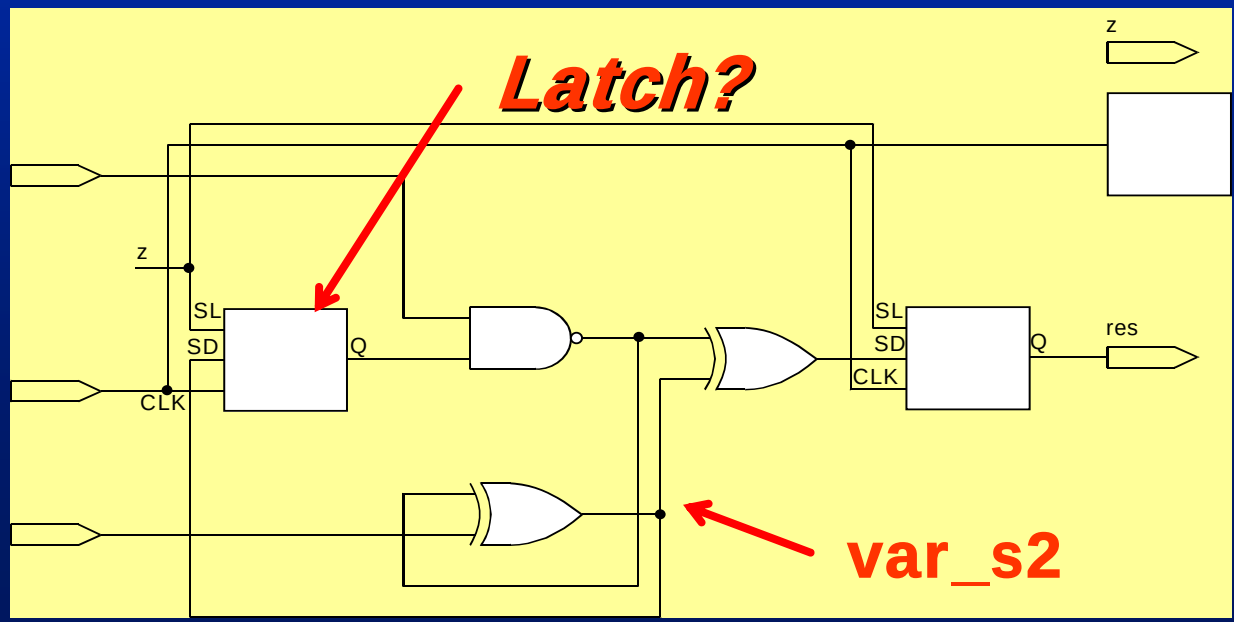


- Current value (variable) vs. previous value (signal)

Inferring Storage for Variables

```
architecture behavior of sig_var is
begin
process
variable var_s1, var_s2 :std_logic;
begin
wait until (rising_edge(clk));
   var_s1 := x nand var_s2;
   var_s2 := var_s1 xor y;
   res <= var_s1 xor var_s2;
end process;
end behavior;
```

- Variables used before it is defined
- There exists an execution sequence (first) where use precedes definition

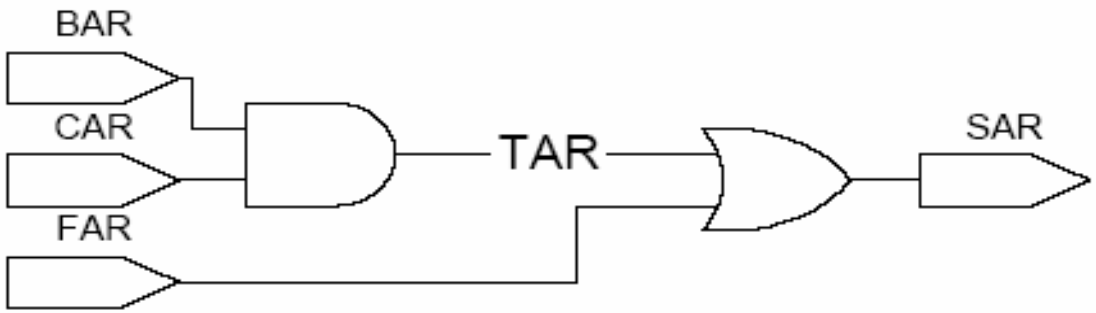


Inferring a Wire

```
signal CAR, BAR, FAR, SAR : BIT;

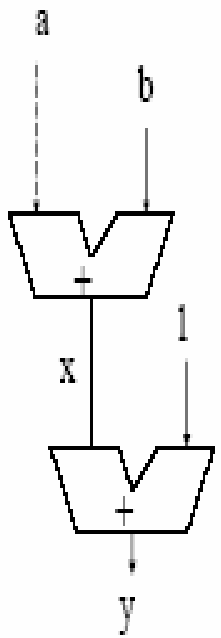
 NO_MEMORY: process(BAR, CAR, FAR)
   variable TAR : std_logic;
 begin
   TAR := BAR and CAR;
   SAR <= TAR or FAR;
 end process;
```

- Wait for event on BAR, CAR, or FAR
- Variable TAR does not require a latch since it is used immediately
- TAR implemented as a wire

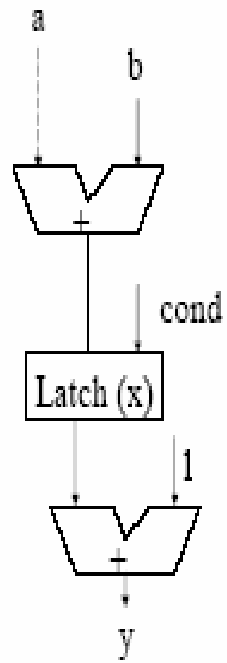


Wires vs. Latches

```
...
end process;
process(a,b)
  variable x,y;
begin
   ...
   x := a + b;
   y := x + 1;
   ...
end process;
...
```




```
process(a,b)
   variable x,y,z;
begin
   ...
   if cond = '1' then
      x := a + b;
   end if;
   y := x + 1;
   ...
end process;
```

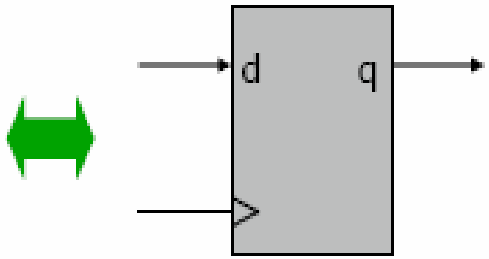


Latch vs. Flip Flop Inference

- Predicates in conditional expressions lead to latch inference
  - *if (sel = '1') then...*
- Edge detection expressions lead to flip flop inference
  - *if (rising_edge(clk)) then...*
  - *if (clk'event and clk = '0') then..*
  - if (clk'lastvalue = '0' and clk = '1' and clk'event) ..
- Semantics must be consistent with available parts
  - Latches vs. flip flops
  - Trigger condition

Inferring: Flip Flops

```
flipflop : PROCESS ( clock , d )
BEGIN
   IF (clock'EVENT and clock = '1') THEN
      q <= d;
   END IF;
END PROCESS flipflop;
```

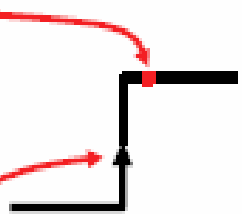


- If **there is a transition** and after it `clock` equals `1`, `q` will be scheduled to be the value of `d`, and at the end of the `PROCESS`, `q` will be assigned the value of `d`
- If there is no transition or `clock` equals `0`, `q` will **not** be scheduled any value because the `IF` statement is skipped. Therefore at the end of the `PROCESS`, `q` will **keep** its previous value!
- We also described a **memory** element, but this time it is a flip-flop

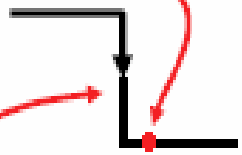
Inferring Flip Flops

- A signal with an `'EVENT` modifier evaluates to true if there has been a high to low or low to high transition

```
IF (<signal_name>'EVENT AND <signal_name> = '1') THEN
    …
END IF;
```



```
IF (<signal_name>'EVENT AND <signal_name> = '0') THEN
    …
END IF;
```



- An `EVENT` should **always** be used in combination with a test for the **current level**, because there exists **no hardware component** which is triggered by both transitions

Example: Counter

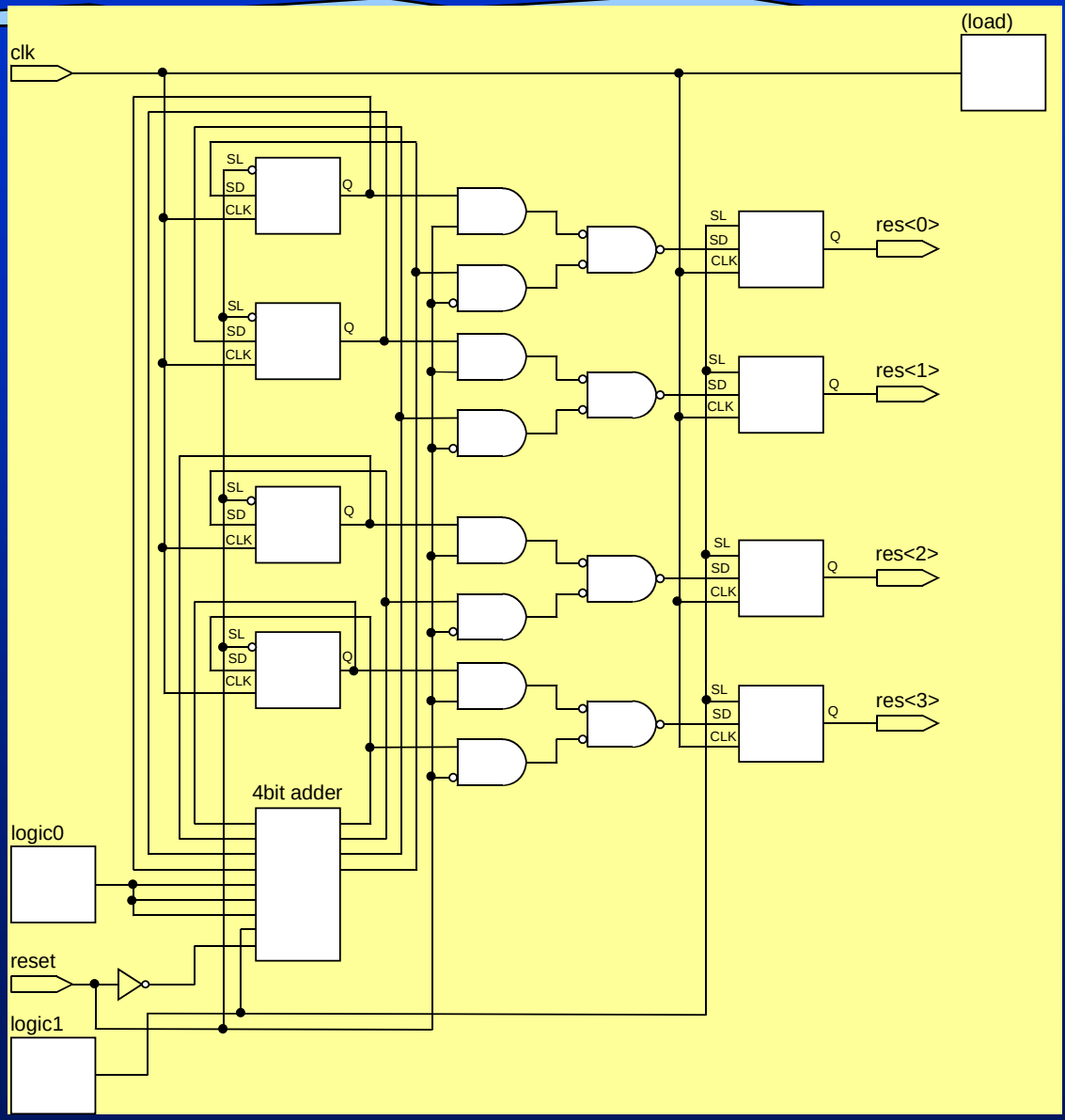
```
entity counter is
port (clk, reset : in std_logic;
res : out unsigned (3 downto 0));
end entity counter;

architecture behavior of counter is
begin
process (clk, reset) is
variable var_count : unsigned (3 downto 0);
begin
if (rising_edge (clk)) then
  if (reset = '1') then
       res <= "0000";
   else
      var_count := var_count + 1;
  end if;
  res <= var_count;
end if;
end process;
end architecture behavior;
```

edge trigger

Note variable is used before it is defined!

Example: Counter

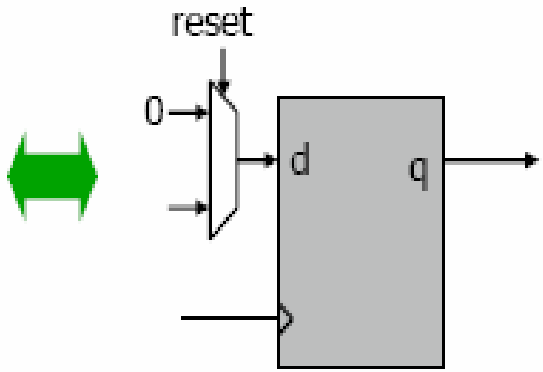


Two Types of Resets

- A register does not have an initial state, and therefore needs a reset
- Two types of reset
  - **Synchronous** reset

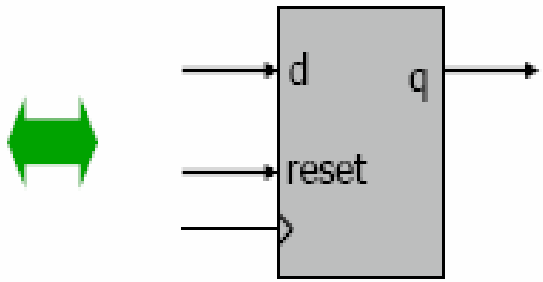
Note: We will always use registers with a reset, either synchronous or asynchronous

```
flipflop : PROCESS ( clock , reset , d )
BEGIN
   IF (clock'EVENT and clock = '1') THEN
      IF (reset = '1') THEN q <= '0';
                       ELSE q <= d;
      END IF;
   END IF;
END PROCESS flipflop;
```



  - **Asynchronous** reset

```
flipflop : PROCESS ( clock , reset , d )
BEGIN
   IF (reset = '1') THEN q <= '0';
   ELSIF (clock'EVENT and clock = '1') THEN
      q <= d;
   END IF;
END PROCESS flipflop;
```



Registers, Signals and Ordering

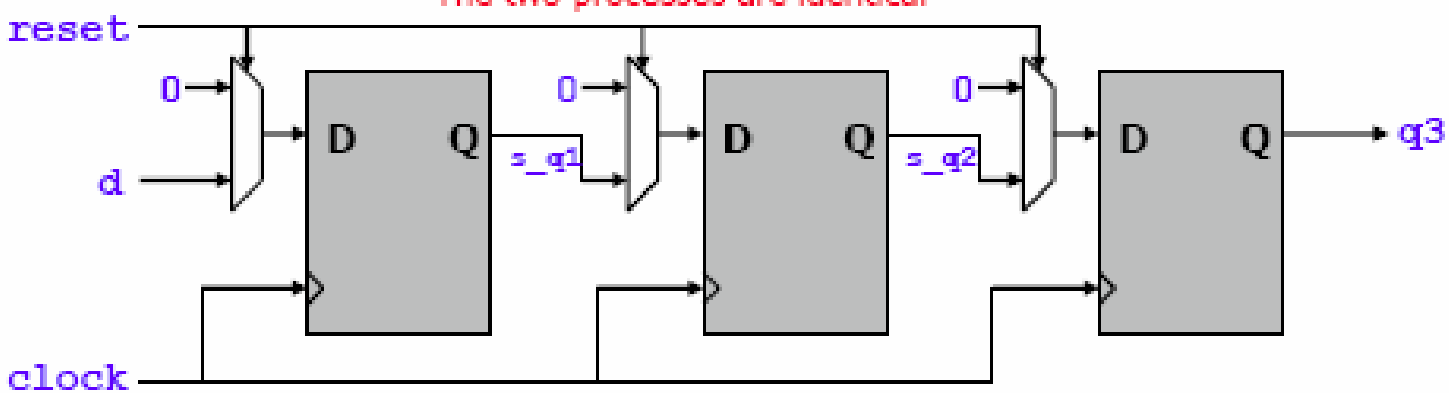
```
Delay_line : PROCESS ( clock , reset , d ,
                       s_q1 , s_q2 )
BEGIN
   IF (clock'EVENT AND clock = '1') THEN
      IF (reset = '1') THEN s_q1 <= '0';
                            s_q2 <= '0';
                            q3   <= '0';
                       ELSE q3   <= s_q2;
                            s_q2 <= s_q1;
                            s_q1 <= d;
      END IF;
   END IF;
END PROCESS Delay_line;
```

=

```
Delay_line : PROCESS ( clock , reset , d ,
                       s_q1 , s_q2 )
BEGIN
   IF (clock'EVENT AND clock = '1') THEN
      IF (reset = '1') THEN s_q1 <= '0';
                            s_q2 <= '0';
                            q3   <= '0';
                       ELSE s_q1 <= d;
                            s_q2 <= s_q1;
                            q3   <= s_q2;
      END IF;
   END IF;
END PROCESS Delay_line;
```

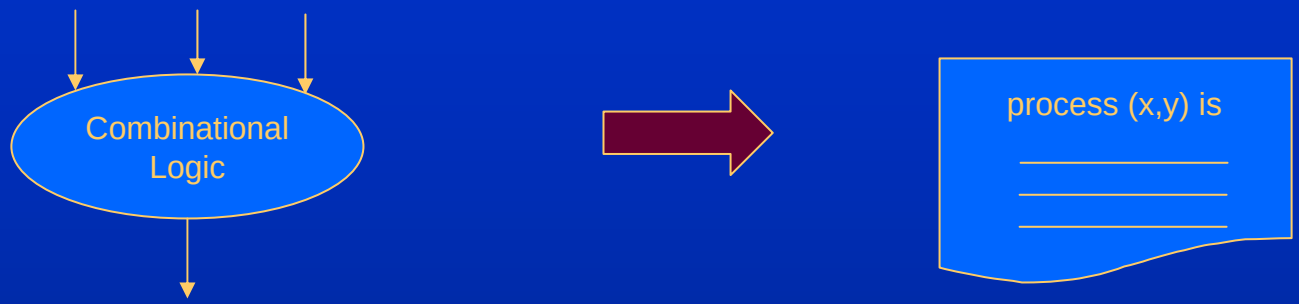
YES!

The two processes are identical



Miscellaneous Issues

- Use of sensitivity lists



  - Pre-synthesis and post-synthesis simulation mismatches
- Optimizations involving variables
- Architecture level interpretation of inference rules
  - Latch inference occurs at outermost level of inferencing
- Use of *buffer* and *inout* modes.

Synthesis Modeling Issues

- **Ensuring compatibility between simulation and synthesis. Both VHDL descriptions below generate the same synthesis output, but differ in their simulation.**

Incomplete sensitivity list

```
Process (enable, in1, in2)
begin
if (enable ='0') then
 output <= (in1 and in2) or in3;
end if  ;
end process;
```

Full sensitivity list

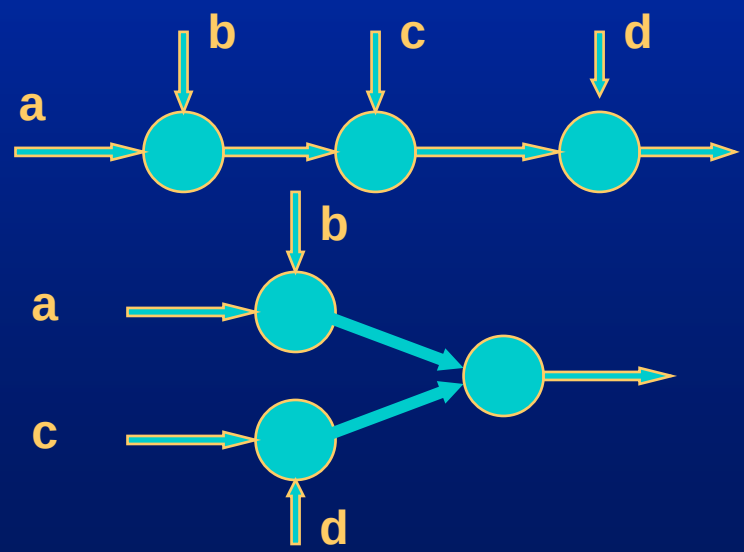
```
Process (enable, in1, in2, in3)
begin
if (enable ='0') then
output <= (in1 and in2) or in3;
end if;
 end process;
```

Note: most synthesis tools ignore the sensitivity list, but IEEE RTL standard recommends using a full sensitivity list to ensure compatibility with the simulation.

Synthesis Modeling Issues

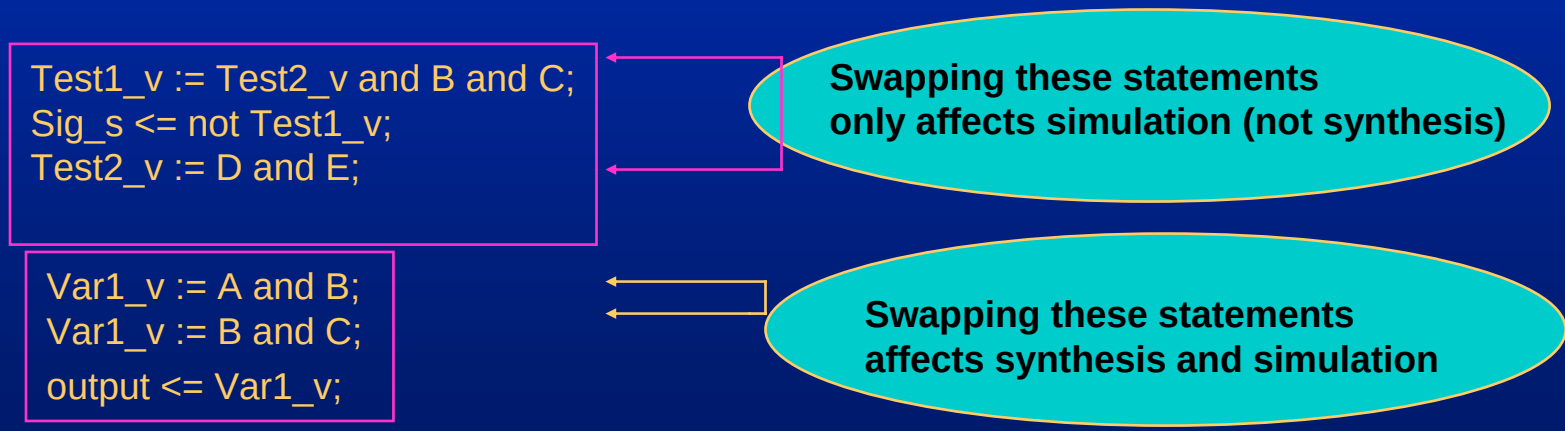
- Using parentheses in arithmetic expressions facilitates optimization through resource sharing, and specification of concurrency, and improves critical path.
- The second statement below has a shorter critical path, for instance,

```
Output <=  a + b + c + d;
Output <=  (a + b) + (c + d)
```



Synthesis Modeling Issues

- **Order of signal assignments does not affect synthesis and simulation as long as the sensitivity list is complete. However, if using signals and variables the order may affect simulation and synthesis**



Wait statements

- The wait statement in VHDL is motivated by the need to model circuits wherein events happen at specific points in time such as
  - **wait on** a, b, c;
  - **wait for** 100 ns;
  - **wait until** x = 1;
  - **wait until** (clk'event and clk = '1')
  - **wait until** (rising_edge(clk)
- From a synthesis perspective a wait statement implies synchronous logic.
- All signals whose value are controlled by the wait statement will be synthesized into flop-flops.

Controlling Inference: Example

```
architecture behavioral of edge is
begin
process
begin
wait until (rising_edge(clk));
if reset = ’1’ then
    Aout <= 1;
else
   Aout <= 3;
end if;
Bout <= 0;
end process;
end architecture behavioral;
```

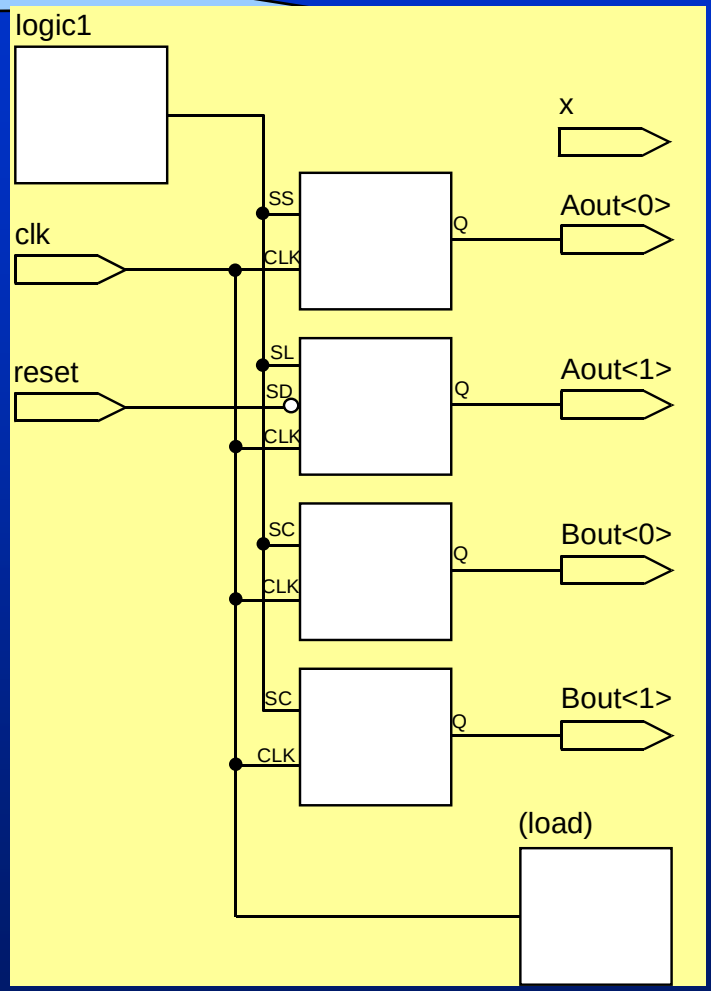
```
architecture behavioral of edge is
begin
process (reset, clk) is
begin
if reset = ‘1’ then
    Aout <= 1;
else
   Aout <= 3;
end if;
if (rising_edge(clk)) then
   Bout <= 0;
end if;
end process;
end architecture behavioral;
```

- Control inference by use of the wait statement

Controlling Inference: Example

```
architecture behavioral of edge is
begin
process
begin
wait until (rising_edge(clk));
if reset = '1' then
    Aout <= 1;
else
   Aout <= 3;
end if;
Bout <= 0;
end process;
end architecture behavioral;
```

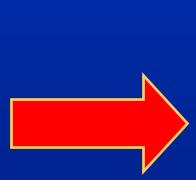


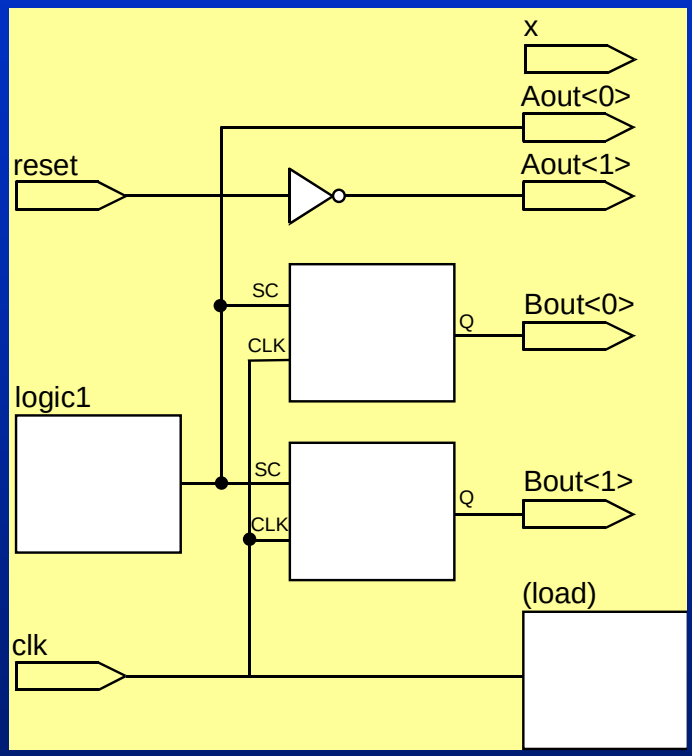


- Control inference by use of the wait statement

Controlling Inference: Example

```
architecture behavioral of edge is
begin
process (reset, clk) is
begin
if reset = '1' then
    Aout <= 1;
else
   Aout <= 3;
end if;
if (rising_edge(clk)) then
   Bout <= 0;
end if;
end process;
end architecture behavioral;
```



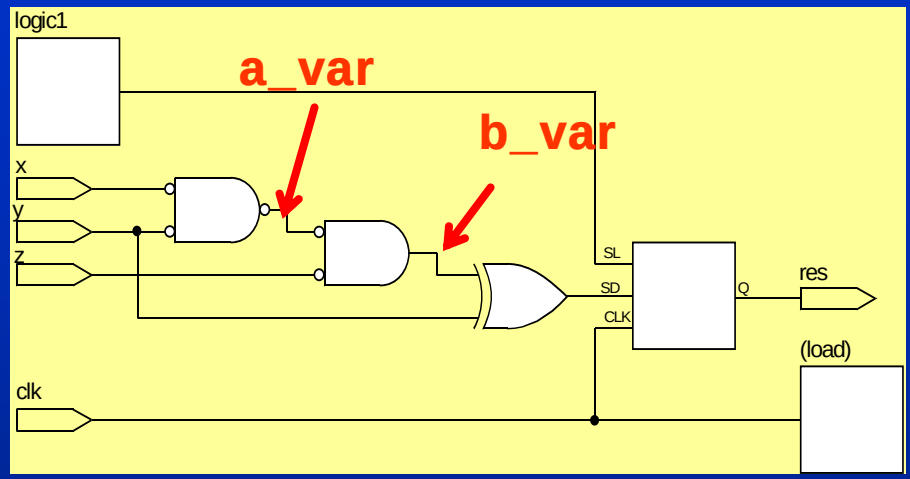


- Using an if-then-endif statement with an edge detection expression limits its flip-flop inference to signals assigned in the body of the then statement.

Variables Controlled by Wait Statement

```
library IEEE;
use IEEE.std_logic_1164.all;
entity sigvar is
port(y, z, x, clk : in std_logic;
     res: out std_logic);
end entity sigvar;

architecture behavioral of sigvar is
begin
 process is
variable a_var, b_var : std_logic;
begin
wait until (rising_edge(clk));
a_var := x or y;
b_var := a_var nor z;
res <= b_var xor y;
end process;
end architecture behavioral;
```



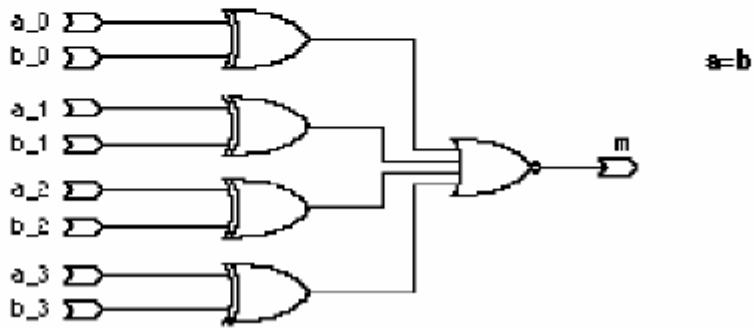
- Variable dependencies collapsed
- If variables are “used before they are defined” then storage is inferred
- Signal dependencies are not collapsed, but rather storage is inferred for each signal

Wait Statements: Remember

- Wait statements imply synchronous logic
- Only one wait statement permitted in a process and it must be the first statement in the process
- All signals in the body of the wait cause storage to be inferred
- Use *if-then-elsif-endif* constructs to trade-off combinational vs. sequential circuit inference

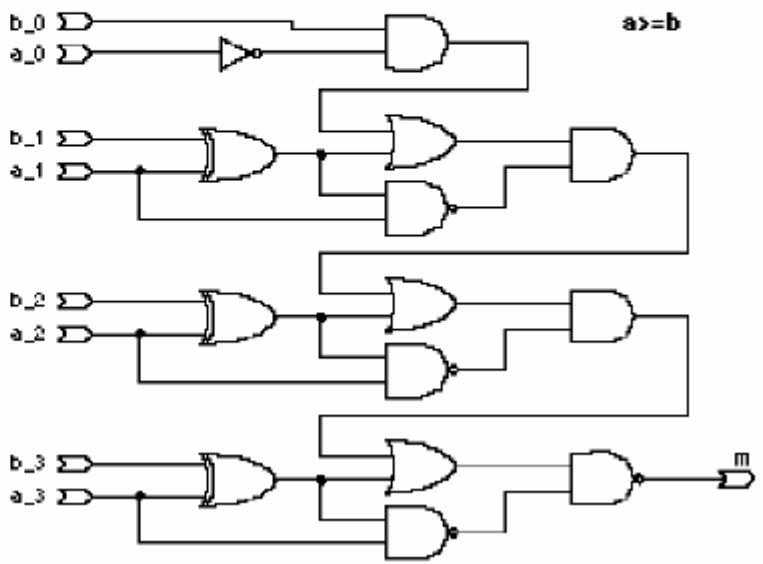
Relational Operations

```
entity relational_ops_1 is
    port (a, b: in bit_vector (0 to 3); m: out Boolean);
end relational_ops_1;
architecture example of relational_ops_1 is
begin
    m <= a = b;
end example;
```



Relational Operations

```
entity relational_ops_2 is
    port (a, b: in integer range 0 to 3; m: out Boolean);
end relational_ops_2;
architecture example of relational_ops_2 is
begin
    m <= a >= b;
end example;
```



Synthesis of Arithmetic Operators

- Only a subset of VHDL operators supported by synthesis tools
  - Supported operations: usually +, -, =, >, <, <>, abs, *
  - The designer must write VHDL description (behavioral or structural) of all un-supported operators: /, N**x
- Specifying a range of integer is important for efficient synthesis
  - Suppose integer x takes only two values 100,101
    - It will be assigned 7 bits
    - It may be better to declare x as constrained integer

Synthesis Hints: Initialization

- Do not specify initial values in your declaration of signals. Most synthesis compilers will ignore them.
- Specify the number of bits necessary for a signal explicitly in the declaration. This will avoid the allocation of much larger number of bits than necessary and therefore lead to less hardware in the form of the widths of signal paths, the number of gates necessary to process these signals, and the number of latches or flip-flops necessary to store signal values.

Synthesis Hints: Inferring Storage

- Generally, edge-detection expressions will cause flip-flops rather than latches to be inferred. Otherwise latches will be inferred.
- If you wish to avoid having a latch inferred for a signal in a process then every execution path through the process must assign a value for that signal.
- If you use variables in a process before they are defined a latch will be inferred for that variable.
- To avoid the inference of latches, make sure that default values are assigned to signals before a conditional block of code.
- For variables or signals assigned within a for-loop a default value must be assigned before the for-loop to avoid latch inference.
- Use If-Then statements to infer Flip-Flops rather than WAIT statements
  - Cannot mix combinational and sequential portions

Synthesis Hints: Optimization-1

- **Avoid** programming in **C or Java** where we try to exploit the sequentiality of the code. This will lead to long signal paths. Attempt to minimize dependencies between statements and try and promote concurrency.
- Move common complex operations out of the branches of **if-then-else** statements and place them after the conditional code. This will generally lead to less hardware.
- Using don't care values to cover the when others case in a **case statement** can enable the synthesis compiler to optimize the logic and create a smaller circuit than if all remaining options were set to values.
- Using a **case statement** rather than an **if-then-else** construct will produce less logic since priority logic will have to be generated for if-..

Synthesis Hints: Optimization-2

- Using don’t care values to cover the when others case in a **case statement** can enable the synthesis compiler to optimize the logic and create a smaller circuit than if all remaining options were set to values.
- Use parentheses in simple CSA statements to control the depth of the circuit. Thus you can exercise some control over operator concurrency in, and therefore speed of, the synthesized circuit.
- The for-loop is synthesized by first unrolling the loop. Loop carried dependencies, where computation in one iteration are dependent on computation in another iteration, can lead to long signal paths.

Synthesis Hints: General

- The **while-loop** statement is generally not supported for synthesis because the loop range must be statically determined in order to generate a fixed amount of logic.
- All **for loop** indices must have statically determinable loop ranges
- Do not use real data types since they are not supported.
- Check vendor specific constraints on the permitted types and range of the for-loop index.
- The choice of level-sensitive conditional expression vs. edge-detection expressions in your VHDL code should be guided by the parts available in our target library. For example, if latches are not available then the synthesis tool may try to create a latch by synthesizing the gate-level equivalent.

Consistency with Simulation

- In a simulation model delays can be specified using the after clause in the signal assignment statements. During synthesis the delay value of the operations are derived from the synthesized implementation. This may differ from the values that the designer specified for simulation.
- Include all signals in a process in the sensitivity list of the process to avoid pre-synthesis and post-synthesis simulation mismatches.
- Use VARIABLEs instead of SIGNALs; Variables may be synthesized the same as signals but simulation is faster.
- Minimize the number of processes. This avoids the overhead of multiple processes suspending and restarting.
- Part of the difficulty in developing synthesizable models is in getting out of the habit of writing models for simulation where some behaviors can be modeled but not necessarily implemented. The process of synthesis is hardware design.

Synthesis - Coding Guidelines

- When in doubt, use hierarchy to allow synthesis control at the level required; It can always be removed later
- Verify small parts of the code independently with gate counts and schematics
- Use processes for sequential and complex combinational
- Do not use "WAIT FOR.." or any timing delays except to solve simulation flash-thru problems;
- Avoid loops except where repetitive code can be reduced; Use FOR loops and avoid EXIT and NEXT where possible

Performance Guidelines

- Place the most likely condition first in an IF/THEN/ELSE or CASE.
- Convert small valued multiplies and divides into adds and shifts.
- A PROCESS uses time when it is activated. If a WAIT condition keeps a PROCESS from activating, it saves time. Converting an IF condition to a WAIT condition may help.
- Use high-level constructs such as “+”, “-”, “*”, etc., instead of low-level logic implementation.
- Limit stimulus to that being used for the portion of the model under test.
  - Example: If the object for a range of stimulus is to verify combinational circuitry, consider turning off the clock which is causing clocked processes to activate periodically.

Style Guidelines

- Follow commas and colons with one or more spaces
- Be consistent in bit numbering (recommended: n-1 DOWNTO 0)
- Use indentation, spacing, and lining up similar elements for ease in reading; Limit indent to 2-3 spaces to avoid pushing out the right margin
- USE COMMENTS!! Explain anything that someone else (or even yourself at a later date) would need to know to understand the model
- Making your code readable and understandable; Be consistent!

Style Guidelines

- Use constants for fixed values especially if they need explanation. For arrays, set a generic or constant to length and set another constant to length-1
- List PORT signals vertically and use comments next to them
- Use STD_LOGIC types (in port interfaces especially)
- Do not use aliases
- Label all concurrent statements
- Repeat names of all constructs at the construct end
- Place only one design unit in a file

Style Guidelines

- Use configurations at each level in your hierarchy to avoid defaults; This would prevent a long simulation with the wrong design unit
- Use named association in port map and procedure calls
- For STD_LOGIC, generally use only the values ’0’, ’1’, ’Z’, and ’X’
- Avoid unconventional coding styles (records, guarded blocks,..)

Synthesis: RTL Scalar Types

- Enumeration
  - Enumeration types shall be supported, and BIT, BOOLEAN, and STD_ULOGIC will be mapped to single bits. Severity_level is ignored
  - The user may override default mapping of enumerated types through use of enum_encoding attribute. File_open_kind, file_open_status are not supported.

- Integer
  - integer types are supported and subtypes NATURAL and POSITIVE are supported
  - Recommendation that tool should convert integer type to bit_vector during implementation. If range has only positive values, use UNSIGNED, if negative/positive use SIGNED representation in NUMERIC_BIT.
  - Integer ranges are synthesized as if 0 is included in range.

Example: One-hot coding
attribute **enum_encoding**: string;
type snack is (donut, biscuit, idli, cake);
attribute **enum_encoding** of snack:
type is "1000 0100 0010 0001" ;

Example: Integer range
integer range 8 to 10;
-- is synthesized the same way (4 bits) as
integer range 0 to 15;

Synthesis: RTL Scalar Types

- Physical Types
  - Physical types, other than TIME, shall not be supported.
  - Declarations of objects (signals and variables) of type TIME shall be ignored.
  - Reference to objects of type TIME may only occur within constructs such as the after clause in VHDL.
- Floating Point Types
  - Floating point types declarations shall be ignored.
  - Reference to objects of type floating point may only occur within constructs such as the after clause in VHDL

Synthesis: Composite Types

- Array
  - Array type definition is supported
  - constrained and unconstrained arrays are supported
  - index of array shall contain exactly one discrete range --- bounds of the index range shall be specified directly or indirectly as static values of integer type
  - null ranges are not supported, and range shall return integer values
  - predefined array types shall be supported.
- Record
  - Record type shall be supported

Synthesis: Type Declarations

- Declarations of type and subtype shall be supported
- Incomplete type declarations shall not be supported
- User-defined resolution functions shall be ignored.
- Full access type declarations or file type definition shall not be supported.

Synthesis: Object Declarations

- Constant declarations
  - Constant declaration shall be supported
  - Deferred constant declaration shall not be supported
- Signal declarations
  - Signal declarations shall be supported
  - signal_kind (register, bus) shall not be supported
  - initial value expressions shall be ignored
  - if signal is declared in package, it must have an initial value
  - assignment to signal declared in package shall not be supported
- Variable declarations
  - Variable declaration shall be supported
  - initial values shall be ignored
  - shared variables shall not be supported
- Interface (port) declarations
  - interface signal, constant and variable declarations supported
  - interface file declarations ignored
  - linkage and bus modes not supported
- Alias declarations
  - the following attribute declarations will be supported