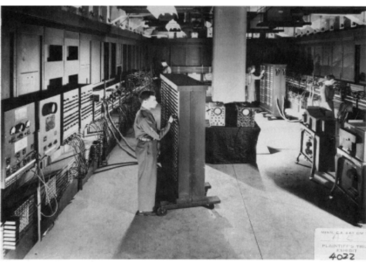
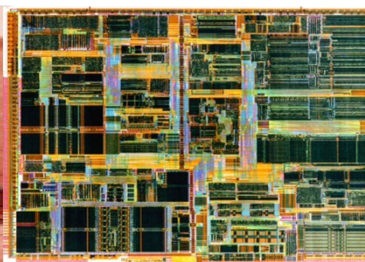
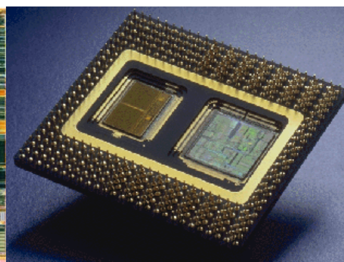
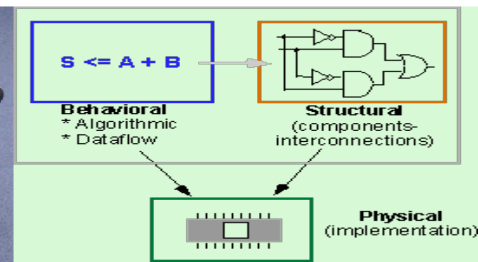
CPE 487: Digital System Design
Spring 2018

Lecture 12
VHDL Synthesis

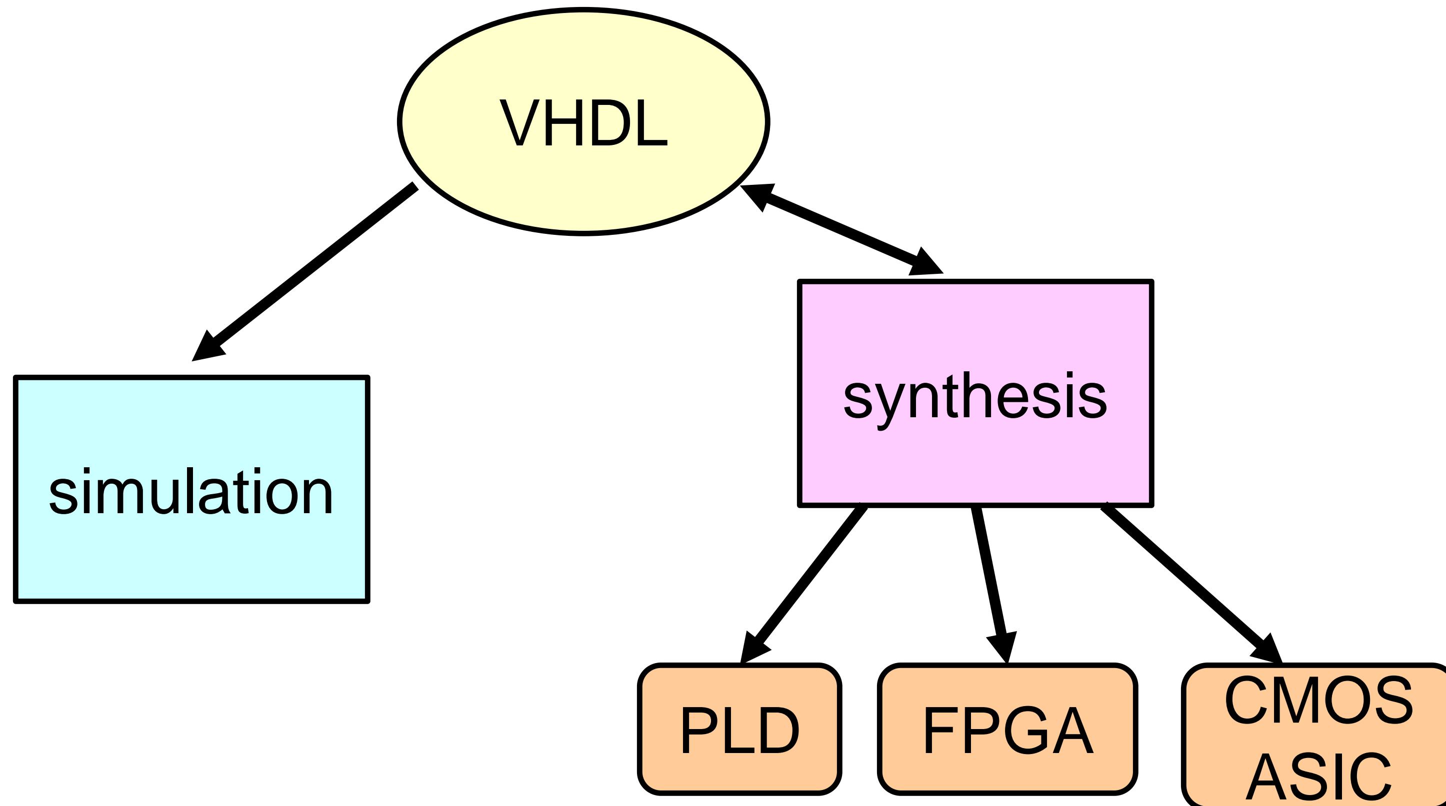
Bryan Ackland
Department of Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, NJ 07030



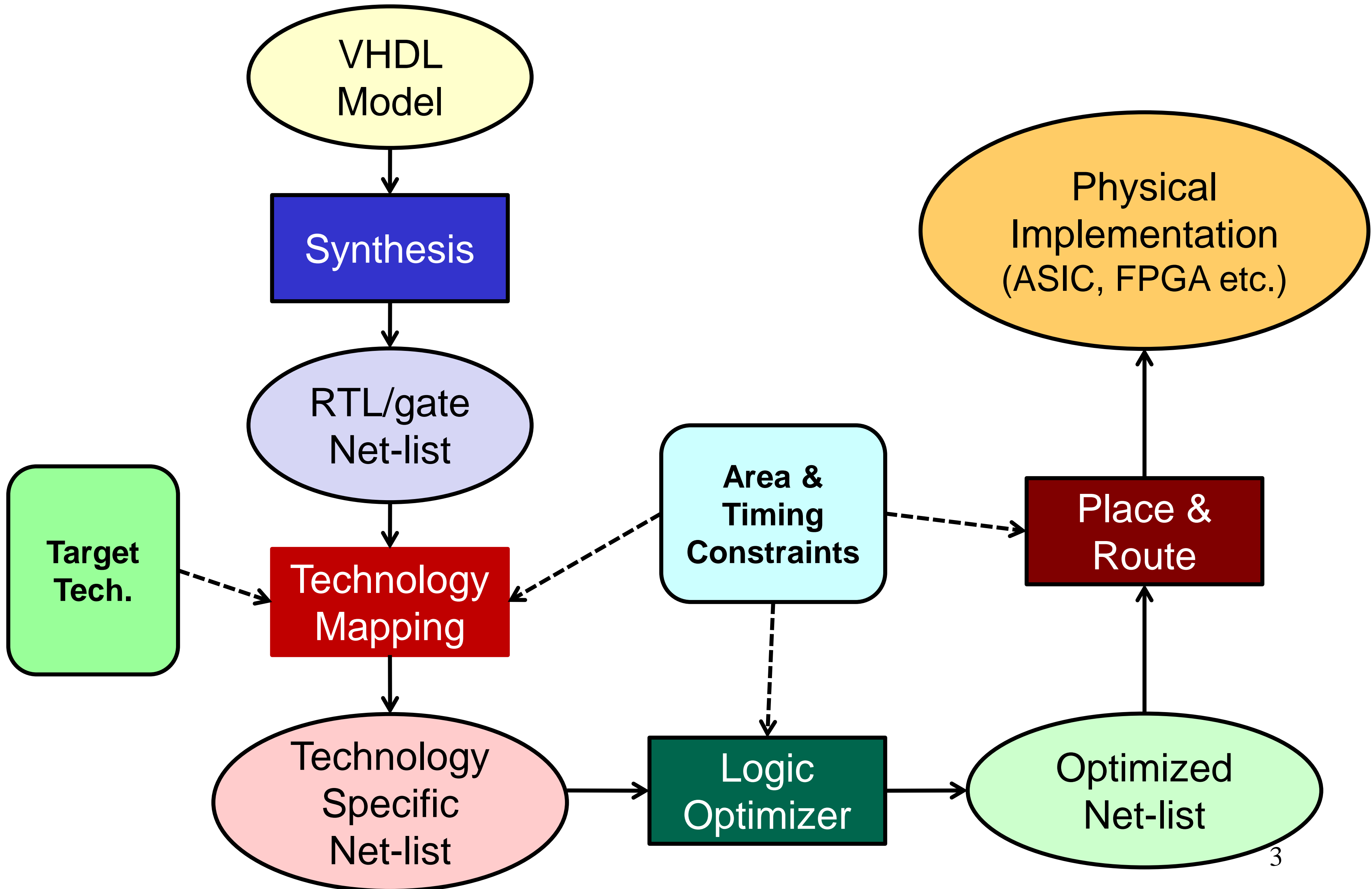




What is Synthesis?

- Process of creating a RTL or gate level net-list from a VHDL model
- Net-list can be used to create a physical (hardware) implementation



Synthesis Process



Limitations of Synthesis

- VHDL can model a hardware system at different levels of abstraction from gate level up algorithmic level
- Synthesis tools cannot, in general, map arbitrary high level behavioral models into hardware.
- Many VHDL constructs have no hardware equivalent:

```
wait for 10 ns;
signal a1: real;
y<= x'last_value;
assert a=b report "mismatch" severity error;
```

- Even if VHDL code is synthesizable, it may lead to very inefficient (in terms of area, speed) implementation
- In moving from VHDL high level behavioral description to synthesizable VHDL hardware description, designer needs to know:
  - What subset of VHDL is synthesizable
  - What hardware in inferred by various VHDL constructs

Example of inefficient synthesis

- 8 x 8 matrix multiply:

```
mpr:    process (A,B) is
variable sum: integer;
begin
    for i in 1 to 8 loop
        for j in 1 to 8 loop
            sum:=0;
            for k in 1 to 8 loop
                sum := sum + A(i)(k)*B(k)(j);
            end loop;
            C(i)(j) <= sum;
        end loop;
    end loop;
end process;
```

Uses: 512 multipliers
448 adders

Inference

- Synthesis is the process of hardware inference followed by optimization
- The synthesis compiler infers hardware structures from your VHDL code
- Those hardware structures are subsequently optimized to meet your area and/or speed constraints.

- Part of being a good synthesis designer is being able to put yourself in the place of the compiler and understand what hardware constructs are likely to be inferred from your code.

Inferring Hardware from VHDL Code

```
entity synth is
port(A,B,C,D: in integer;
        sel: in std_logic_vector(1 downto 0);
        Z:out integer);

architecture behavioral of synth is
begin
        with sel select
        Z <= A+B when "00"
        C+D when "10"
        0 when others;
        end architecture behavioral;
```

- We need an adder (and a multiplexer).
  - How large (how many bits) should the adder be?
  - Integers can be up to 32 bits
  - What if A,B,C and D are in the range 0-20?
- Do we need two adders, or can one adder be shared?
  - Are we more concerned with speed or area?

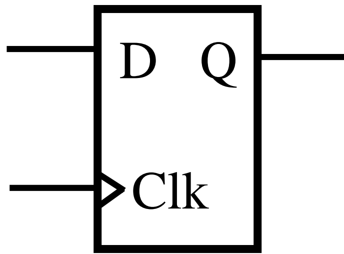
Inferring Signals

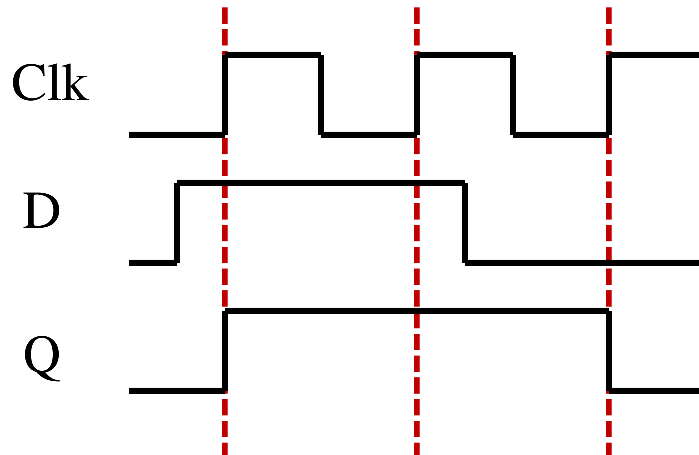
- In regular programming languages, declared variables & constants are used to allocate storage.
- In VHDL, signals are used to represent timed information flow between subsystems.
  - How are signals represented in hardware?
- When synthesizing from VHDL, the basic hardware implementations of signals are:
  - Wires
  - Latches (level sensitive)
  - Flip-flops (edge triggered)
- Signals generated by combinational expressions (where signal value depends only on the current value of the inputs) will infer wires
- Signals generated by sequential expressions (where signal value depends on current and previous value of inputs) will infer latches or flip-flops

Flip-flop vs. Latch

Flip-flop

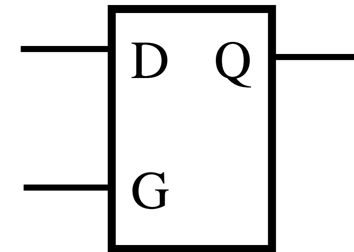
stores data when clock rises

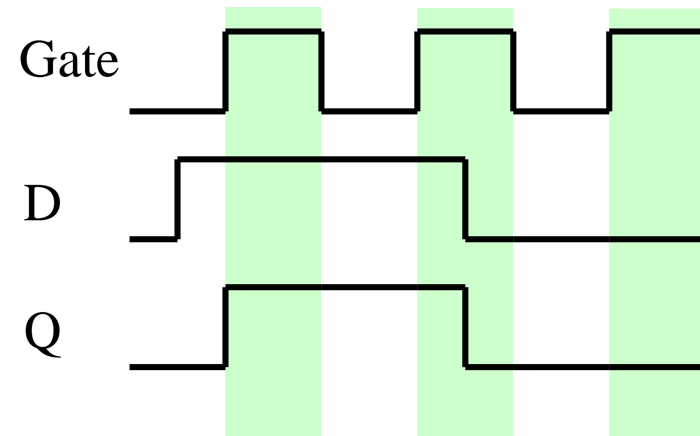




Latch

passes data when G is high
stores data when G is low





Inference from Declarations

- When a signal is declared, there needs to be sufficient information to determine the correct number of bits

```
signal result: std_logic_vector (12 downto 0);
signal count: integer;
signal index: integer range 0 to 18:= 0;

type state_type is (state0, state1, state2, state3);
signal next_state: state_type;
```

- How many bits are required to represent *count* ?
  - Compilers are conservative: they will only perform transformations that are guaranteed not to produce incorrect answers
- Initializations are often ignored
  - Need to include run-time initialization in hardware (e.g. reset)
- How many bits are required to represent *next_state* ?

Inference from Simple CSA's

target_signal <= *expression*;

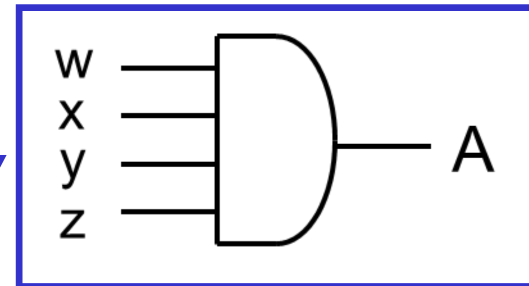
e.g. A <= (B **nand** C) **or** Y **after** 12 **ns**;

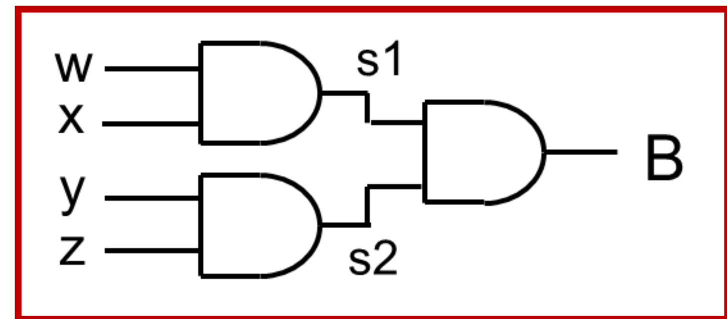
- A simple CSA assigns a value to a target signal that is a function of the present value of the signals in the RHS expression.
  - Whenever an event occurs on any signal in *expression*, *target_signal* is re-evaluated.
  - no memory of previous values
- The synthesis compiler will infer a combinational circuit
- Delay information (e.g. **after** 12 **ns**) will be ignored.
- You can control inferred structure by appropriately grouping logic operations into assignment statements

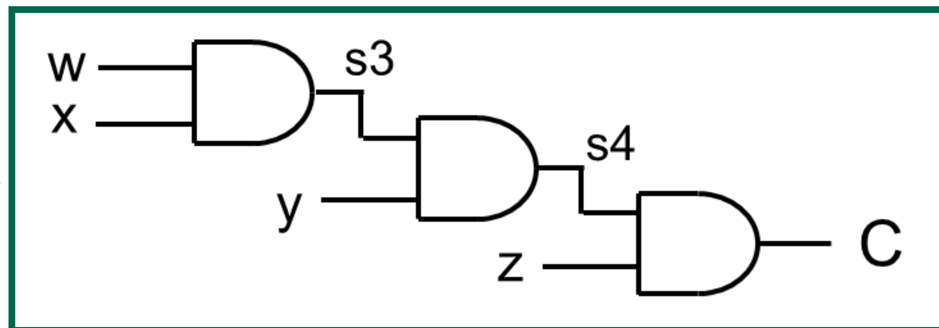
Logical Grouping

- Use of intermediate signals can be used to infer different logical (RTL) implementations:

```
entity abc is
port(w, x, y, z: in std_logic;
     A, B, C: out std_logic);
end entity abc;

architecture dataflow of abc is
signal s1, s2: std_logic;
begin
A <= w and x and y and z;
s1 <= w and x;
s2 <= y and z;
B <= s1 and s2;
s3 <= w and x;
s4 <= s3 and y;
C <= s4 and z;
end architecture;
```

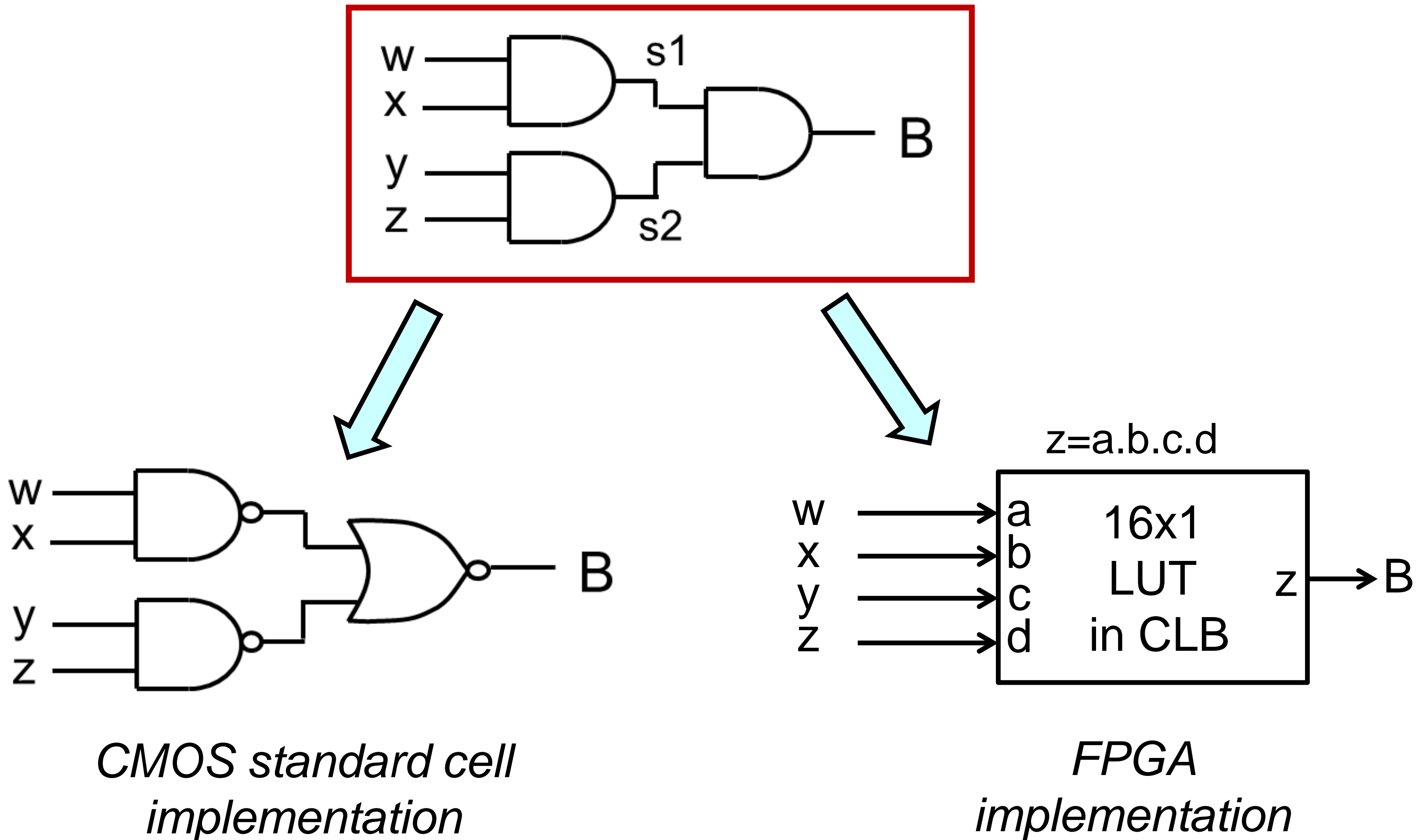






Logical Grouping Modified by Mapping

- RTL structure may be considerably modified by technology mapping and optimization:

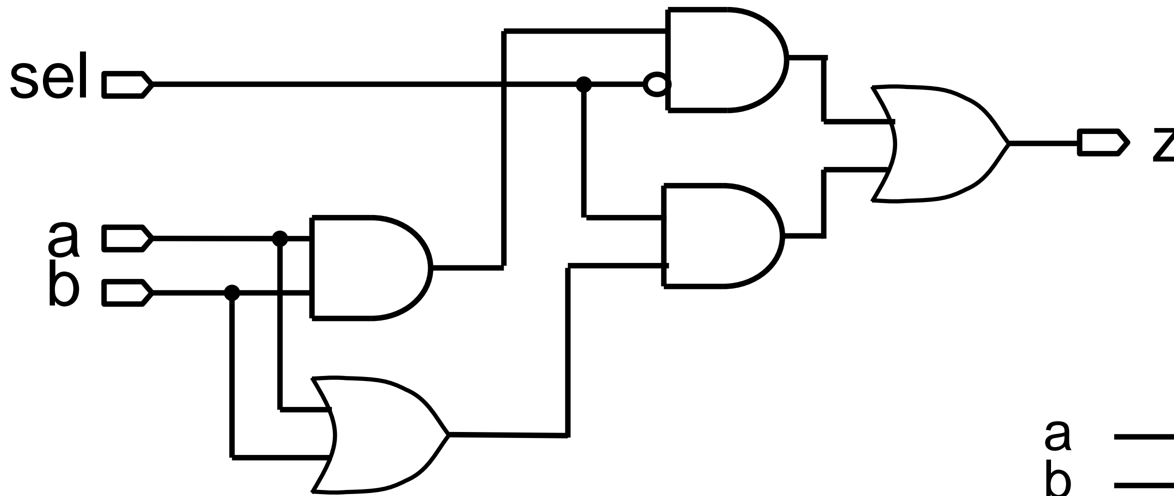


CMOS standard cell implementation

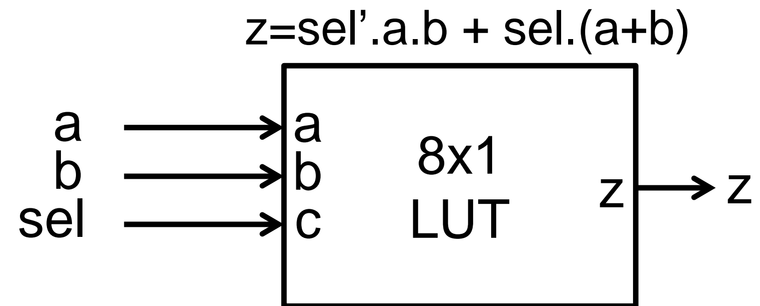
FPGA implementation

Inference from Conditional Signal Assignment

```
architecture cond_sa of mux is
begin
    z <=  a and b when sel='0' else
              a or b;
end cond_sa;
```



gate-level RTL

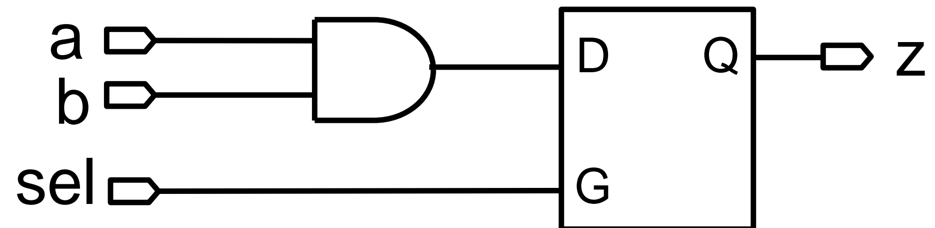


FPGA implementation

Incomplete Assignment Implies Latch

- What happens if there are combinations of inputs that do not result in an assignment?

architecture cond_sa **of** mux **is**
begin
    z <= a **and** b **when** sel='1';
end cond_sa;

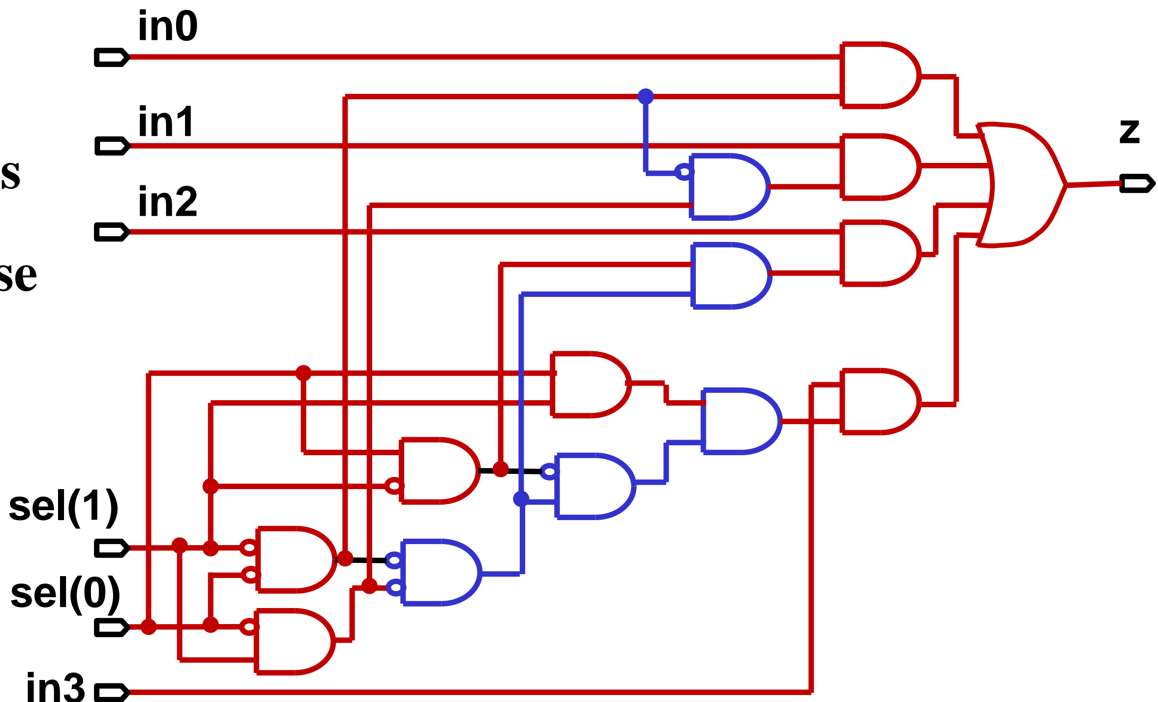


- A latch is inferred to cover the case *sel='0'*
  - This is now a sequential circuit – is that what we intended?
- We may not care what the result is when *sel='0'*
- Result is unnecessary, hazardous hardware
  - If we really wanted sequential operation, better to use flip-flop
- If combinational circuit is intention, make sure output is always assigned a value by using a final *else* clause

Conditional Maintains Priority Order

- This is great for priority encoder, but what about multiplexer where all conditions are mutually exclusive?

```
architecture behave of mux4 is
begin
    z <=  in0 when sel=“00” else
          in1 when sel=“01” else
          in2 when sel=“10” else
          in3 when sel=“11” else
          ‘0’;
end behave;
```

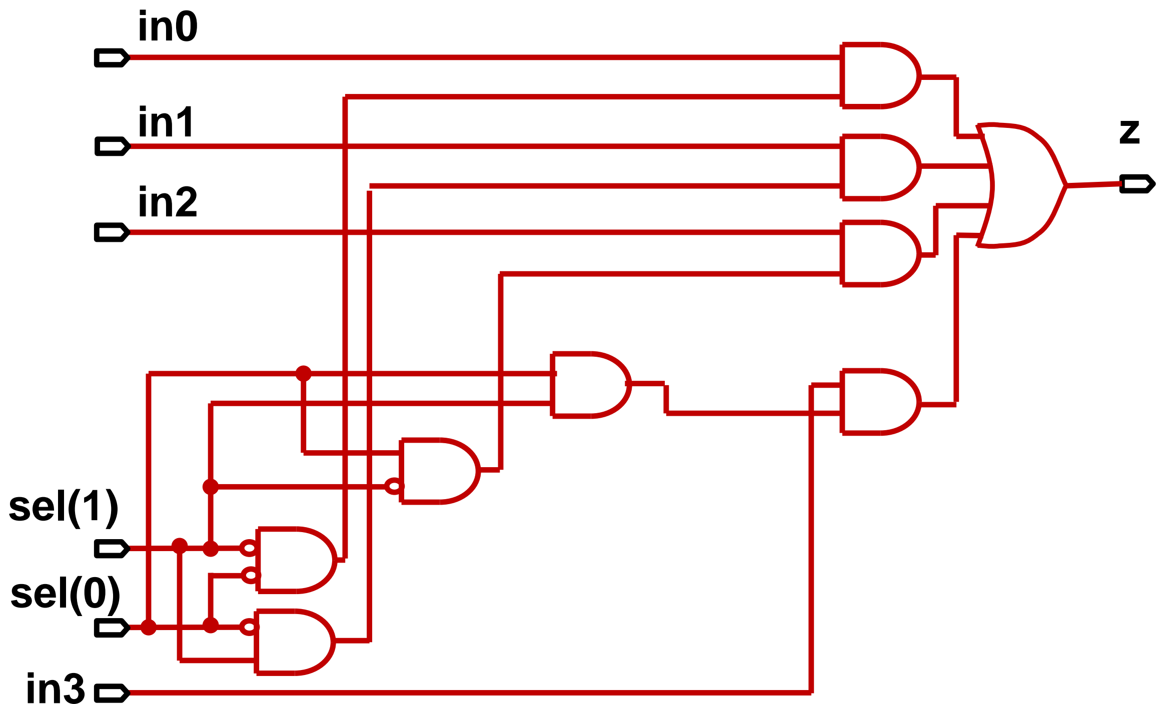


- Only red gates are needed to implement multiplexer
- Blue gates are inferred to maintain priority coding
- Not needed because the clauses are mutually exclusive

Selected Signal Assignment

- SSA is better construct for building a muliplexer
  - No longer implied priority order
  - Clauses are required to be mutually exclusive
  - No redundant gates

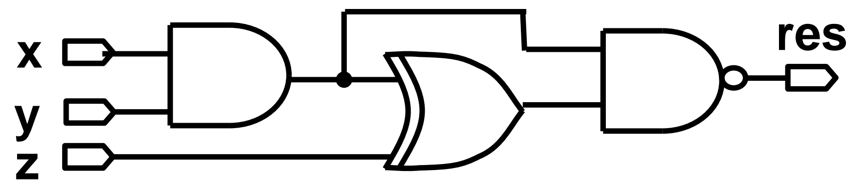
```
architecture SSA of mux4 is
begin
    with sel select
    z <=  in0 when "00" ,
          in1 when "01",
          in2 when "10" ,
          in3 when "11"
          'X' when others;
end SSA;
```



Inferring Logic from Processes and Variables

- Simple variable assignment statements generate combinational logic
- Sensitivity list is usually ignored by synthesis compiler

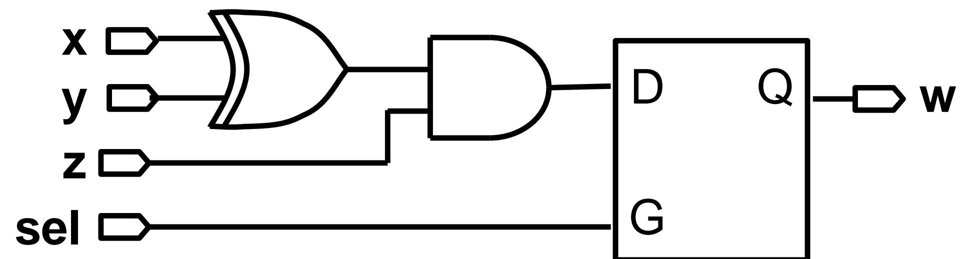
```
entity syn_y is
port ( x,y,z: in std_logic;
    res: out std_logic);
end entity syn_v;

architecture behav of syn_v is
begin
pr1: process (x,y)
    variable v1,v2: std_logic;
    begin
        v1 := x and y;
        v2 := v1 xor z;
        res <= v1 nand v2;
    end process;
end behav;
```



If-then-else Statements

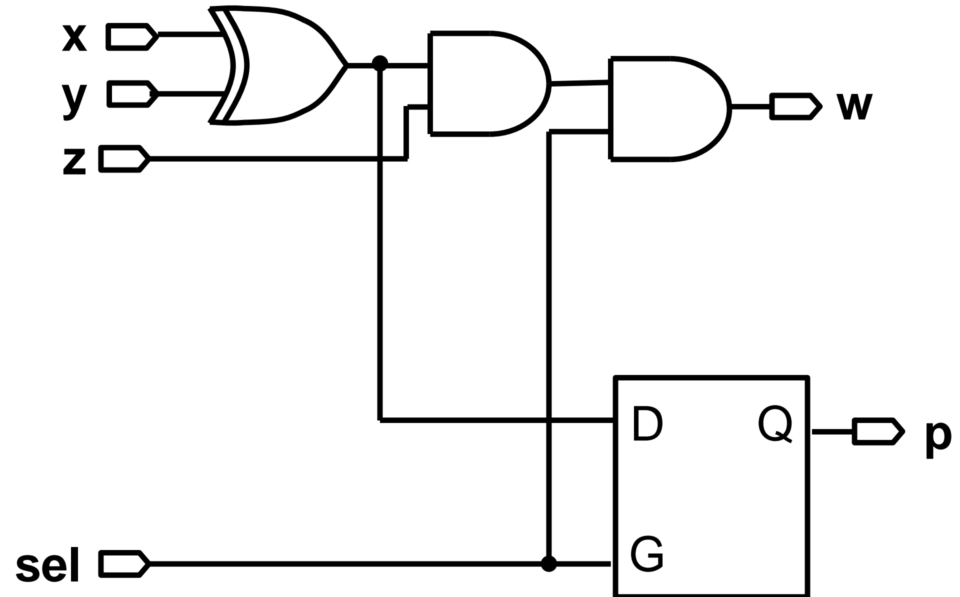
- Like conditional assignment statements, if-then-else statements will generate latches *unless every "output signal" of the statement is assigned a value each time the process executes*
  - (a) One branch of the if-then-else clause must always be taken AND
  - (b) A signal assigned a value in one branch must be assigned a value in all branches.

```
architecture behav of syn_if is
begin
pr1: process (x,y,z,sel)
    variable v1: std_logic;
    begin
        if sel='1' then
        v1 := x xor y;
        w <= v1 and  z;
        end if;
    end process;
end behav;
```



If-then-else Statements

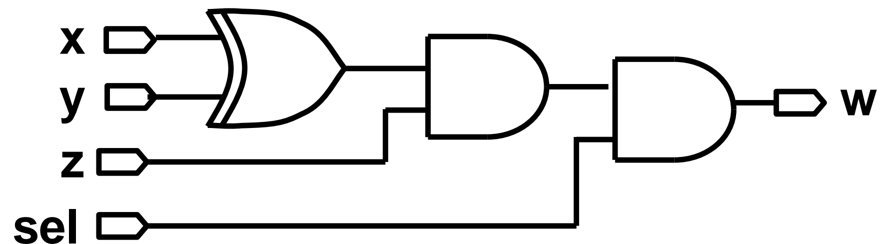
```
architecture behav of syn_if  is
begin
pr1: process (x,y,z,sel)
    variable v1: std_logic;
    begin
        if sel=‘1’ then
            v1 := x xor y;
            w <= v1 and  z;
            p <= V1;
        else
            w<=0;
        end if;
    end process;
end behav;
```



Avoiding latches by “initialization”

- Can ensure that a signal is always assigned in an if-then-else clause by including a final else clause that assigns a default value to all “output signals” of the statement
- Another alternative is to set a signal to a default value within the process but before the if-then-else statement:

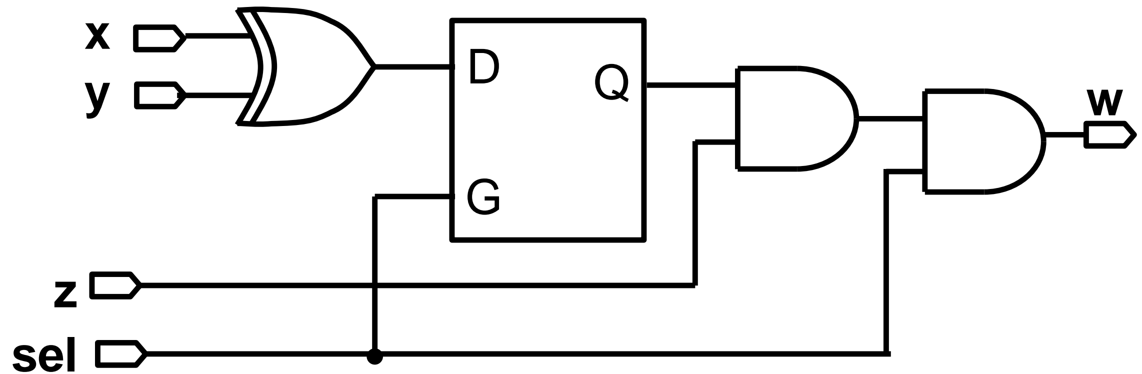
```
architecture behav of syn_if is
begin
pr1: process (x,y,z,sel)
    variable v1: std_logic;
    begin
        w <= '0';
        if sel='1' then
        v1 := x xor y;
        w <= v1 and  z;
        end if;
    end process;
end behav;
```



Variables and Inferred Latches

- Can a variable be used in a manner that implies a latch?
- Yes, if in a single active pass through the process a variable is used before it is assigned:

```
architecture behav2 of syn_if is
begin
pr1: process (x,y,z,sel)
    variable v1: std_logic;
    begin
        w <= '0';
        if sel='1' then
        w <= v1 and  z;
        v1 := x xor y;
        end if;
    end process;
end behav2;
```



Case Statement

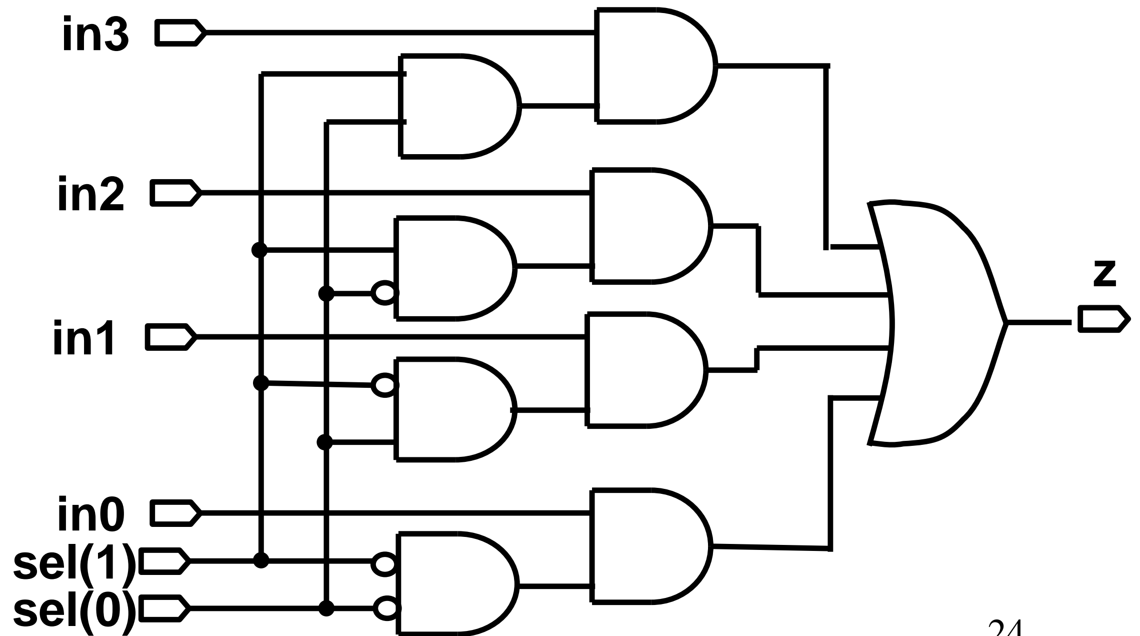
- Case statement is ideally suited for implementing multiplexers
  - all clauses must be mutually exclusive
  - no implied priority between clauses (unlike if-then-else)
  - no redundant logic
- Need to obey same rules to avoid latches

  (a) One branch of the case statement must always be true AND

  (b) If a signal is assigned a value in one branch of a case statement, it must be assigned a value no matter which branch is taken.

4 input multiplexer

```
architecture behav of syn_mux is
begin
pr1: process (in0,in1,in2,in3,sel)
    begin
        case sel is
        when “00” => z <= in0;
        when “01” => z <= in1;
        when “10” => z <= in2;
        when “11” => z <= in3;
        end case;
    end process;
end behav2;
```

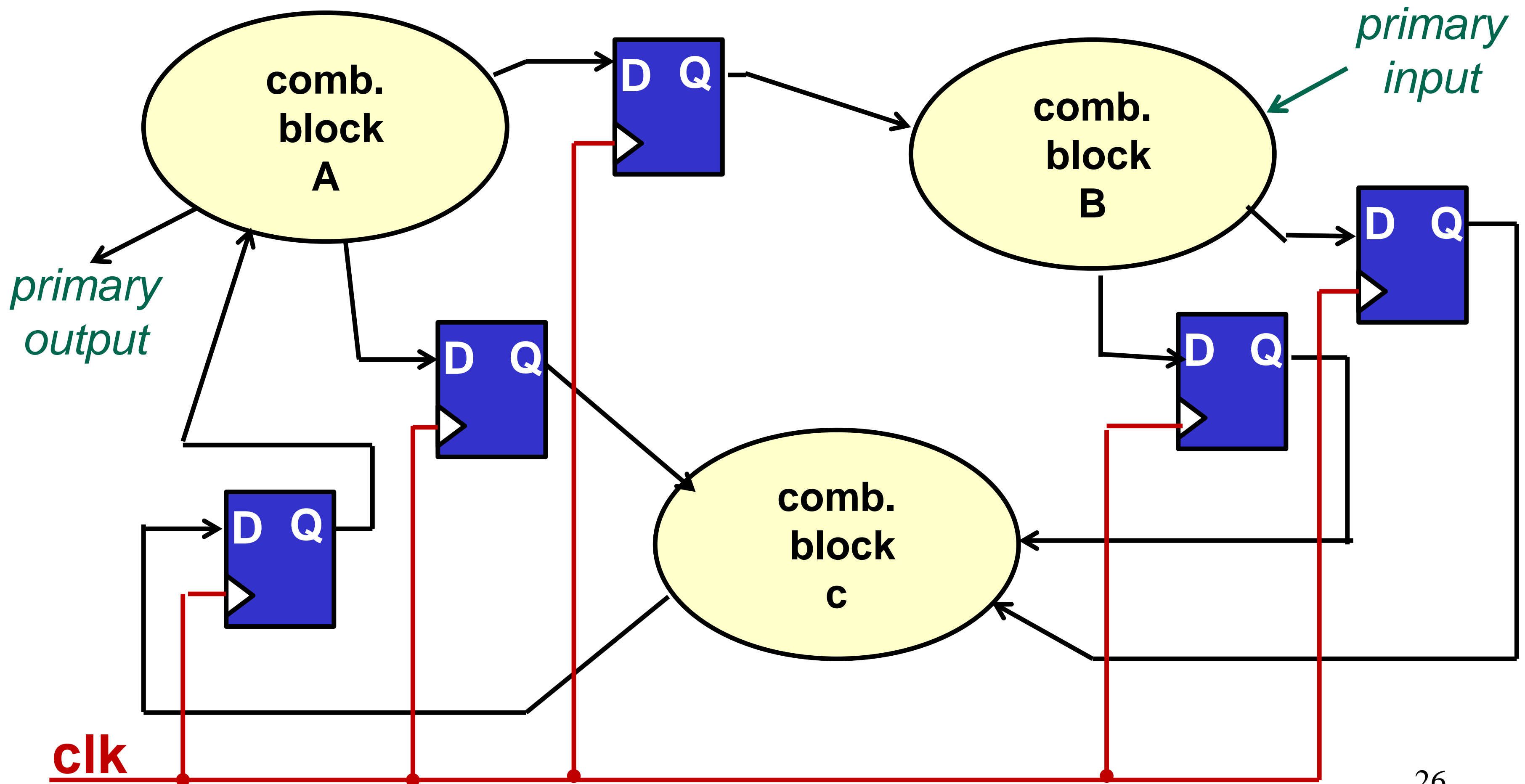


Sequential Circuits

- Any moderately complex digital system requires the use of sequential circuits, e.g. to
  - identify and modify current state of system
  - store and hold data
  - identify sequences of inputs
  - generate sequences of outputs
- If and case statements and conditional signal assignments can be used to infer latches
- Latches are <u>not</u> preferred means of generating sequential circuits.
  - A latch is transparent when LE is high – can lead to unintended asynchronous sequential behavior when a result is fed back to an input
  - Latches are prone to generate race conditions
  - Circuits with latches are difficult to verify timing
  - Circuits with latches are difficult to test

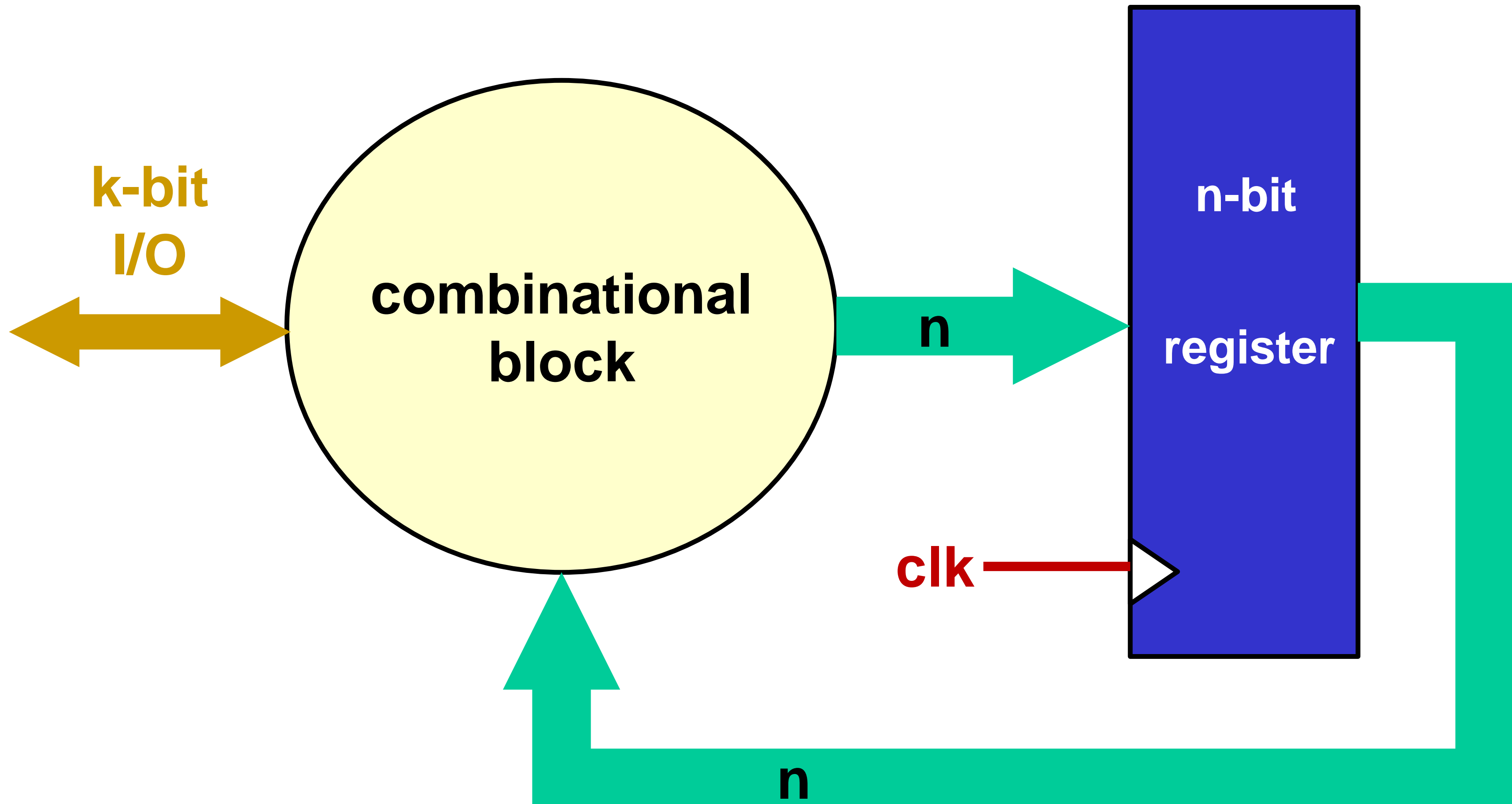
Synchronous (Single Clock) Digital Design

- Preferred design style is combinational circuit modules connected via positive (negative) edge-triggered flip-flops that all use a common clock.



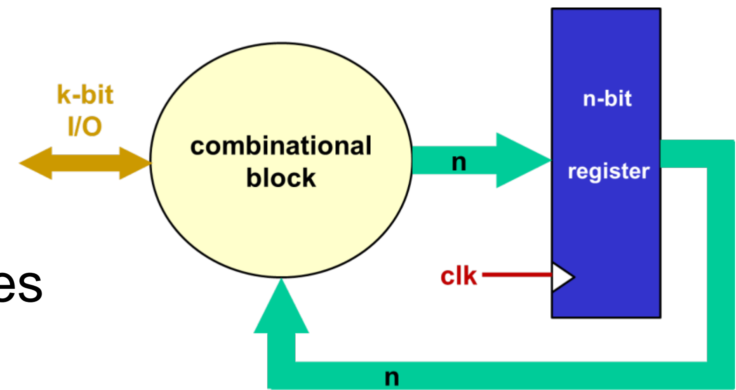
Finite State Machine

- Single clock synchronous system can be modeled as a single combinational block and a multi-bit register



Advantages of single clock synchronous

- Edge triggered D flip-flops are never transparent
  - no unintended asynchronous sequential circuits
- Timing can be analyzed by:
  - determining all combinational delays
  - just add them up
  - checking flip-flop setup and hold times



- No race conditions
  - only time signal values matter is on clock edge
- Easy to test
- Most real systems, however, cannot be designed using a single clock
  - different timing constraints on various I/O interfaces
  - clock delays across chip
  - goal is to make the single clock modules as large as possible

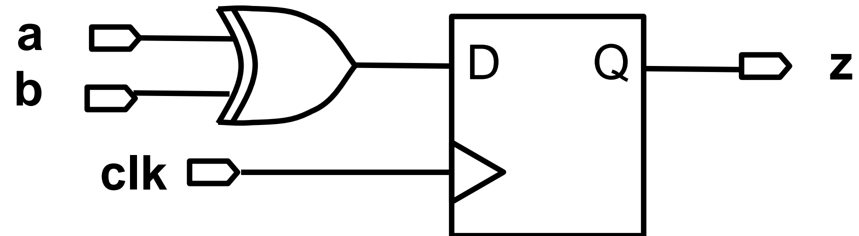
D Flip-flop

- Edge triggered D flip-flops are preferred sequential component
- Within a process, positive edge triggered operation is inferred using:

```
if  rising_edge (clk) then          -- only with std_logic type
if clk'event and clk='1' then
```

- For example:

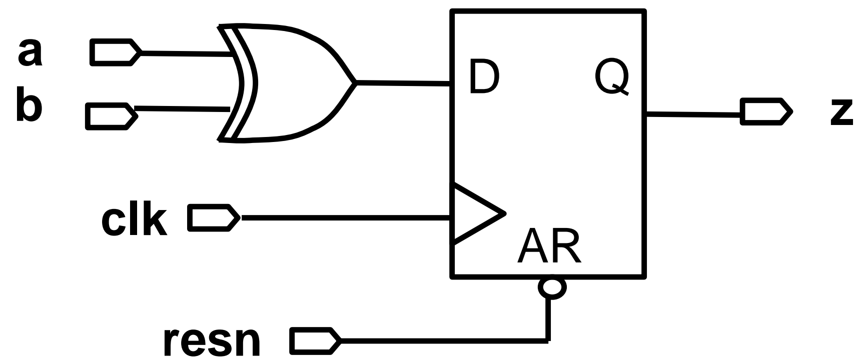
```
architecture RTL of FFS is
begin
p0: process (a,b,clk)
    begin
        if rising_edge (clk) then
            z <= a xor b;
        end if;
    end process;
end RTL;
```



Inferring D Flip-flop with Asynchronous Reset

- Asynchronous reset takes precedence over clock
- Flip-flop can also include asynchronous set

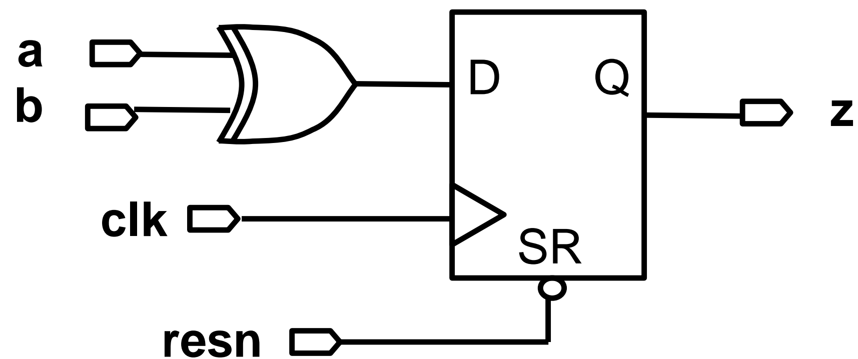
```
architecture RTL of ARFF is
begin
p0: process (a, b, resn, clk)
    begin
        if resn='0'  then
            z <= '0';
        elsif rising_edge (clk) then
            z <= a xor b;
        end if;
    end process;
end RTL;
```



Inferring D Flip-flop with Synchronous Reset

- Synchronous reset waits for clock
- Flip-flop can also include synchronous set

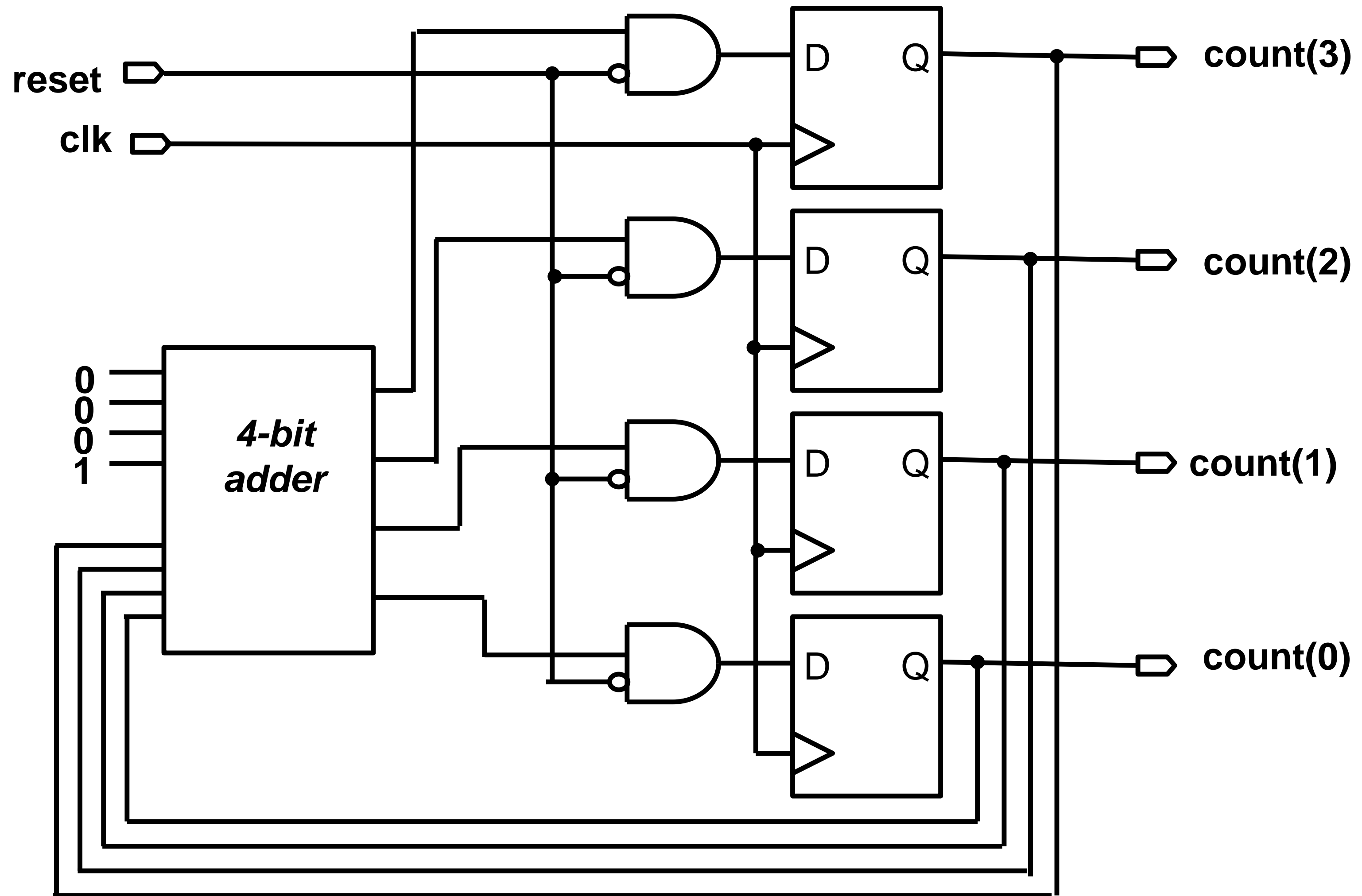
```
architecture RTL of ARFF is
begin
p0: process (a, b, resn, clk)
    begin
        if rising_edge (clk) then
            if resn='0'  then
                z <= '0';
            else
                z <= a xor b;
            end if;
        end if;
    end process;
end RTL;
```



Example: 4-bit synchronous counter

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

entity count4 is
port( clk, reset: in std_logic;
    count: out std_logic_vector (3 downto 0));
end entity count4;

architecture RTL of count4 is
begin
p0: process (clk, reset)
    variable vcount: std_logic_vector (3 downto 0);
    begin
        if rising_edge (clk) then
            if reset='1'  then
                vcount := "0000";
            else
                vcount := vcount+1;
            end if;
        end if;
        count <= vcount;
    end process;
end RTL;
```

Example: 4-bit synchronous counter



Wait Statement

- Only one wait statement permitted per process
- Must be the first statement in the process
- Must be of the **wait until** form
  - cannot use **wait for** or **wait on** constructs

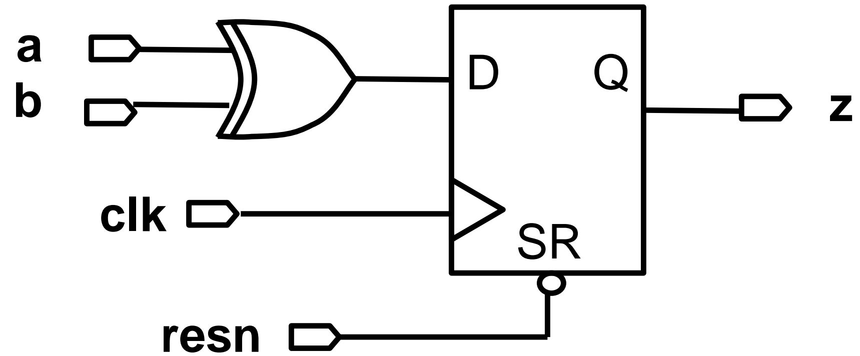
- Either of these will trigger execution on rising clock edge:

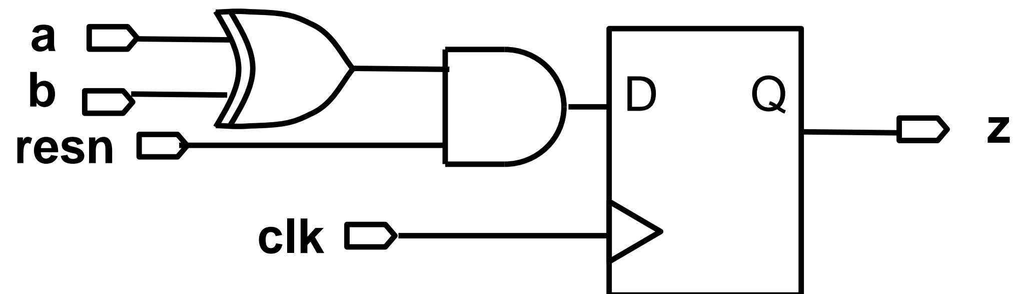
```
wait until clk'event and clk='1';
wait until rising_edge(clk);    -- only with std_logic type
```

- A D flip-flop will be inferred for all signals assigned in the process
  - all signals are synchronously assigned in process

Wait Statement - Example

```
architecture RTL of ARFF is
begin
p0: process
    begin
        wait until rising_edge(clk);
            if resn='0'  then
                z <= '0';
            else
                z <= a xor b;
            end if;
        end if;
    end process;
end RTL;
```





Loop Statements

- For loop is most commonly supported by synthesis compilers
- Iteration range should be known at compile time
- While statements are usually not supported because iteration range depends on a variable that may be data dependent.
- Compiler will unroll the loop, e.g.:

```
for k in 1 to 3 loop
   shift_reg(k) <= shift_reg(k-1);
end loop;
```

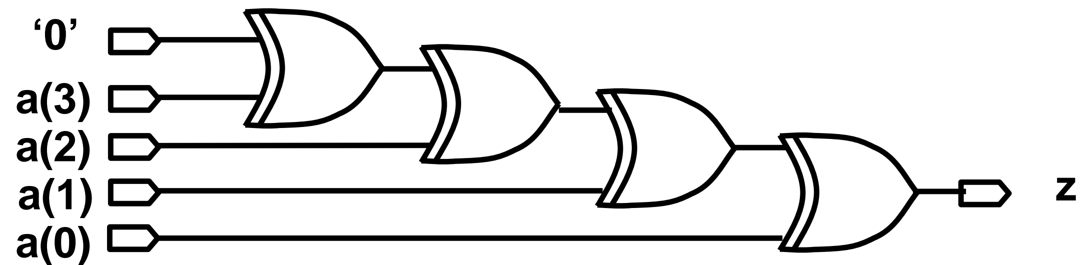
will be replaced by:

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

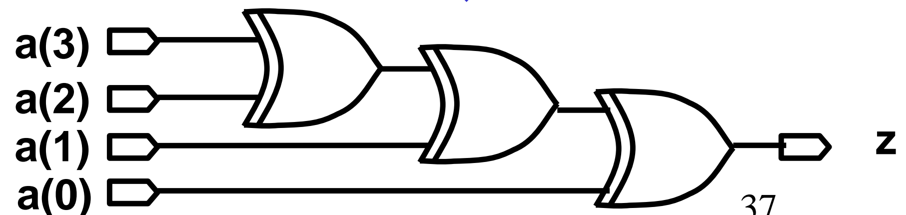
Functions

- Since:
  – functions are executed in zero time and
  – functions do not remember local variable values between calls
- Functions will typically infer combinational logic
- Function call is essentially replaced by in-line code

```
function parity (data: std_logic_vector)
     return std_logic is
variable par: std_logic:= '0';
begin
    for i in data'range loop
        par := par xor data(i);
    end loop;
    return (par);
end function parity;

signal a: std_logic_vector (3 downto 0);
begin
    z <= parity (a);
```



optimized

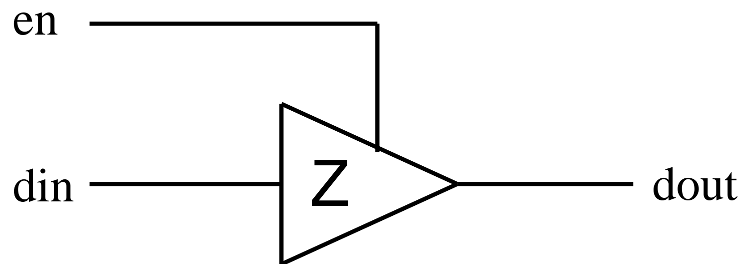


Procedures

- Like functions, procedures do not remember local variable values between calls
- Procedures, however, do not necessarily execute in zero time
- And procedures may assign new events to signals in parameter list
- Like function, think of procedure calls as substituting the procedure as inline code in the calling process or architecture
- Can include a wait statement, but then cannot be called from a process
  - Generally avoid wait statements in a procedure
  - Synchronous behavior can be inferred using if-then-else
- If procedure is called from a process with wait statement, flip-flops will be inferred for all signals assigned in procedure

Tri-state Gates

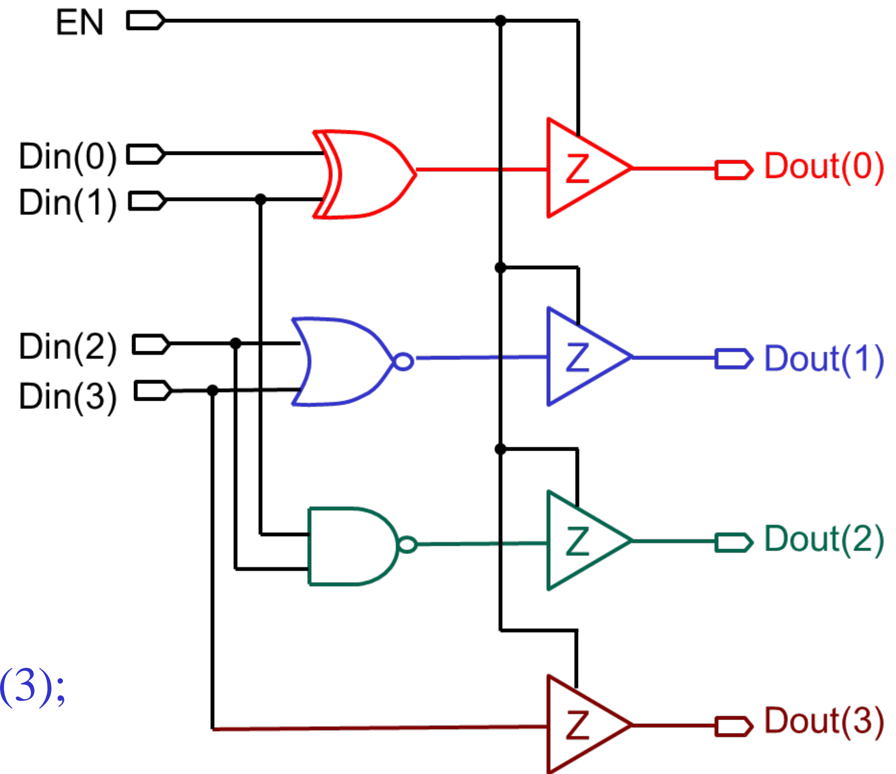
- A tri-state gate is inferred if the output value is conditionally set to high impedance Z



din	en	dout
0	1	0
1	1	1
X	0	Z

Examples of Tri-state Gates

```
architecture RTL of STATE3 is
begin
  seq0 : process (Din, EN)
  begin
    if (EN = '1') then
      Dout(0) <= Din(0) xor Din(1);
    else DOUT(0) <= 'Z';
    end if;
  end process;
  seq1 : process (EN, Din)
  begin
    case EN is
      when '1' => Dout(1) <= Din(2) nor Din(3);
      when others => Dout(1) <= 'Z';
    end case;
  end process;
  Dout(2) <= Din(1) nand Din(2) when EN = '1' else 'Z';
  with EN select
      Dout(3) <= Din (3) when '1',
      'Z'    when others;
end RTL;
```



Think of the Hardware Cost

- Comparison operators like =,>,< require comparator circuits
- Arithmetic operators like +,-, and * imply adder, subtractor, and multiplier
- Multiplication and division by powers of 2 can be implemented using an arithmetic shift
- Multiplication by simple constants can be accomplished with explicit shift & add (e.g. multiply by 3)
- Order of logic and arithmetic operations:

Think of the Hardware Cost

- Compare the following:

```
architecture RTL of  modX  is
signal A1, B1: unsigned(15 downto 0);
begin
  A1 <= A - 1;
  B1 <= B - 1;
seq: process
    begin
      wait until clk;
      case SEL is
          when '0'  => DOUT <= A1;
          when '1'  => DOUT <= B1;
      end case;
    end process;
end RTL;
```

```
architecture RTL of  modX  is
begin
seq: process
variable v1: unsigned (15 downto 0);
    begin
      wait until clk;
      case SEL is
          when '0'  => v1:=A;
          when '1'  => v1:=B;
      end case;
     DOUT <= v1 - 1;
    end process;
end RTL;
```

Example: Inferring Synthesis

- Draw the logic inferred by the following VHDL code:

```
entity abc is
port (
    A,B, clk, reset: IN bit;
    C: OUT bit
    );
end abc;
```

```
architecture RTL of abc is
signal W, X, Y, Z;
begin
    W <= X xor Z;
    p0: process (clk, A, B, reset)
    begin
        X <=  '0';
        if clk='1' then
            X <=  A;
            Z  <=  A and B;
        end if;
        if reset='0' then
            Y  <=  '0';
        elsif clk'event and clk='1' then
            Y  <=  A nor B;
        end if;
    end process;
    C <=  W nand Y;
end RTL;
```

Summary – Guidelines for Synthesis

(1) While loop is generally not supported

(2) All for loops must have statically determined loop ranges

(3) To avoid a latch being inferred in a conditional CSA, every execution of the statement must assign a value to the target signal

(4) To avoid a latch being inferred for a signal in a process, every executable path through the process must assign a value to that signal (assign default values before conditionals)

(5) Using a variable in a process before it is assigned will infer a latch for that variable

(6) Do not specify initial values for signals

(7) Include all signals in process sensitivity list to avoid mismatch between pre- and post-synthesis simulation

(8) All code should infer hardware structure – avoid “algorithmic” descriptions of hardware

Summary – Guidelines for Synthesis

9) If possible, specify data ranges explicitly in declarations
10) Minimize signal assignment within a process – use variables
11) If you wish to use both combinational and sequential logic in a process, use if-then statements to infer flip-flops rather than wait statement.
12) When possible, move common complex operations out of the branches of conditional code and place them after the conditional code. This will generally lead to less hardware
13) Use a case statement rather than if-then-else if clauses are mutually exclusive – avoids redundant priority logic
14) Avoid level sensitive latches if they are not part of the target technology library