Sequential Statements

This slide set covers the *sequential statements* and the VHDL *process* (do NOT confuse with *sequential circuits*)

Sequential statements are executed in *sequence* and allow a circuit to be described in more abstract terms

A **process** is used to encapsulate them because they are not compatible with the concurrent execution model of VHDL

Unlike concurrent statements, there is NO clear mapping to hardware components

Some sequences and coding styles are **difficult** or **impossible** to synthesize

 To use them for synthesis, coding must be done in a disciplined matter

A VHDL *process* contains a set of sequential statements that describe a circuit's behavior

 The *process* itself is a **concurrent** statement and should be thought of as a *circuit part* enclosed inside a black box

VHDL Process Statement

The sequential statements that can be included in a process include

- *wait* stmt
- *sequential signal assignment* stmt
- *if* stmt
- *case* stmt
- simple *for loop* stmt

There are other sequential stmts, including more sophisticated loop stmts, the *next* and *exit* statements, that are useful in simulations to be discussed later

Two basic forms of the *process* stmt

- A process with a sensitivity list
- A process with wait statement

The second form has one or more wait stmts but no sensitivity list

Commonly used in test benches for simulations

The first form is better for describing hardware

Process with Sensitivity List

Syntax

```
 process(sensitivity_list)
    declarations;
    begin
    sequential statement;
    sequential statement;
   ...
 end process;
```

The *sensitivity list* is a list of signals to which the process responds and *declarations* are local to the process

A *process* is NOT invoked (as in prog. lang) but is either

- Active (known as activated)
- Inactive (known as suspended)

A process is activated when a signal in the sensitivity list **changes its value**

Its statements will be executed *sequentially* until the end of the process

Process with Sensitivity List

It then suspends again, waiting on another signal in *sensitivity list* to change

```
signal a, b, c, y: std_logic;
process(a, b, c)
   begin
   y <= a and b and c;
end process;
```

This process simply describes a *3-input AND* gate

```
process(a)
   begin
   y <= a and b and c;
end process;
```

This process has an **incomplete** sensitivity list, i.e., executes when *a* changes but remain *inactive* for changes in *b* and *c*

This implies *memory* (*y* maintains its value when *b* and *c* change) and it describes a circuit that is sensitive to the rising and falling edge on *a* (not realizable)

Although incorrect here, we will see other uses later for *sequential circuits*

Process with *wait* Statement

So for combinational circuits, ALL inputs MUST be included in *sensitivity list*

Process with **wait** statement(s) has no sensitivity list

Process continues the execution until a *wait* statement is reached and is then suspended

There are several forms of the *wait* statement

```
wait on signals;
wait until boolean_expression;
wait for time_expression;
```

For example

```
process
   begin
   y <= a and b and c;
   wait on a, b, c;
end process;
```

Sequential Signal Assignment Statement

This process immediately executes and computes the output for *y*

It then waits for a change on *a*, *b* or *c* -- on a change it continues and resets the output *y* to a new value based on the input signal change, and suspends again

Note this describes the *3-input AND* gate as well, however, the process with the *sensitivity list* is preferred for synthesis

A process can has *multiple* wait statements

Enables the modeling of complex timing behavior and sequential events

However, for synthesis, restrictions apply, e.g., only one *wait* stmt

Syntax of the *sequential signal assignment* statement

```
signal_name <= value_expression;
```

Syntax is identical to the simple concurrent signal assignment, however, inside a process, a signal can be assigned **multiple times**

But only the **last** assignment takes effect

Sequential Signal Assignment Statement

For example

```
process(a, b, c, d)
    begin                      -- yentry := y
    y <= a or c;               -- yexit := a or c;
    y <= a and b;              -- yexit := a and b;
    y <= c and d;              -- yexit := c and d;
end process;                   -- y <= yexit
```

It is same as

```
process(a, b, c, d)
    begin
    y <= c and d;
end process;
```

What happens if the 3 statements are concurrent statements (outside a process)?

Hint: the result is very different and is not likely something you would want to build

Variable Assignment Statement

Syntax

```
variable_name := value_expression;
```

Note the use of ':=' instead of '<=', which indicates **immediate assignment** (no propagation delay)

This behavior is similar to variables in C

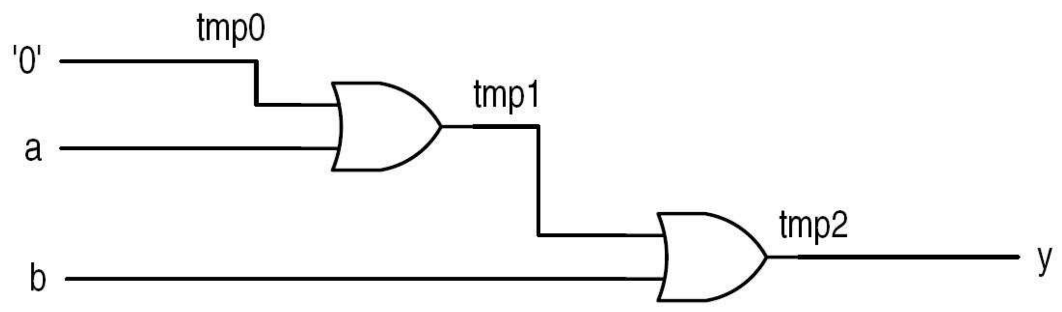
```
process(a, b, c)
    variable tmp: std_logic;
    begin
    tmp := '0';
    tmp := tmp or a;
    tmp := tmp or b;
    y <= tmp;
end process;
```

Although easy to understand, this is difficult to map to hardware

Variable Assignment Statement

In order to realize the previous process in hardware, we need to re-code as

```
process(a, b, c)
    variable tmp0, tmp1, tmp2: std_logic;
    begin
    tmp0 := '0';
    tmp1 := tmp0 or a;
    tmp2 := tmp1 or b;
    y <= tmp2;
end process;
```



This re-coding allows us to interpret the variables as *signals* or *nets*.

What happens if we replace the *variables* with *signals*?

Variable Assignment Statement

```
signal a, b, y, tmp: std_logic; -- 'globally' declared
...
process(a, b, c, tmp)
   begin                     -- tmp_entry := tmp
   tmp <= '0';               -- tmp_exit := '0';
   tmp <= tmp or a;          -- tmp_exit := tmp_entry or a;
   tmp <= tmp or b;          -- tmp_exit := tmp_entry or b;
end process;                 -- tmp <= tmp_exit
```

Same as:

```
process(a, b, c, tmp)
   begin
   tmp <= tmp or b;
end process;
```

This specifies a combinational loop, i.e., the output of an **or** gate is connected to one of its inputs!

If Statement

Syntax

```
if boolean_expr_1 then
   sequential_statements;
elsif boolean_expr_2 then
   sequential_statements;
elsif boolean_expr_3 then
   sequential_statements;
...
else
   sequential_statements;
end if;
```

Consider an *if* stmt description of the MUX, decoder, priority decoder and simple ALU from concurrent signal assignment chapter

```
architecture if_arch of mux4 is
   begin
   process(a, b, c, d, s)
      begin
```

If Statement

```
            if (s="00") then
               x <= a;
            elsif (s="01")then
               x <= b;
            elsif (s="10")then
               x <= c;
            else
               x <= d;
            end if;
         end process;

      architecture if_arch of decoder4 is
         begin
         process(s)
            begin
            if (s="00") then
               x <= "0001";
            elsif (s="01")then
```

If Statement

```
                x <= "0010";
            elsif (s="10")then
                x <= "0100";
            else
                x <= "1000";
            end if;
        end process;
    end if_arch;

    architecture if_arch of prio_encoder42 is
        begin
        process(r)
            begin
            if (r(3)='1') then
                code <= "11";
            elsif (r(2)='1')then
                code <= "10";
            elsif (r(1)='1')then
```

If Statement

```
            code <= "01";
         else
            code <= "00";
         end if;
      end process;
      active <= r(3) or r(2) or r(1) or r(0);
   end if_arch;

   architecture if_arch of simple_alu is
      signal src0s, src1s: signed(7 downto 0);
      begin
      src0s <= signed(src0);
      src1s <= signed(src1);
      process(ctrl, src0, src1, src0s, src1s)
         begin
         if (ctrl(2)='0') then
            result <= std_logic_vector(src0s + 1);
         elsif (ctrl(1 downto 0)="00")then
```

If Statement

```
            result <=  std_logic_vector(src0s + src1s);
         elsif (ctrl(1 downto 0)="01")then
            result <= std_logic_vector(src0s - src1s);
         elsif (ctrl(1 downto 0)="10")then
            result <= src0 and src1;
         else
            result <= src0 or src1;
         end if;
      end process;
   end if_arch;
```

The *if stmt* and the *conditional signal assignment stmt* are identical if only one signal assignment statement is present in each *if* branch

The *if stmt* is more flexible, however, because sequential statements can be used in **then**, **elsif** and **else** branches:

Multiple statements

Nested *if stmts*

If Statement

For example, to find the max of *a*, *b* and *c*

```
process(a, b, c)
   begin
   if (a > b) then
      if (a > c) then
         max <= a;
      else
         max <= c;
      end if;
   else
      if (b > c) then
         max <= b;
      else
         max <= c;
      end if;
   end if;
end process;
```

If Statement

We need three conditional signal assignments to accomplish the same task

```
signal ac_max, bc_max: std_logic;
...
ac_max <= a when (a > c) else c;
bc_max <= b when (b > c) else c;
max <= ac_max when (a > b) else bc_max;
```

It can also be written as one conditional signal assignment stmt if we 'flatten' the Boolean conditions

```
max <= a when ((a > b) and (a > c)) else
       c when (a > b) else
       b when (b > c) else
       c;
```

Although shorter, it is more difficult to understand

Another situation that *if stmts* are good for is when many operations are controlled by the same Boolean conditions

If Statement

```
process(a, b)
   begin
   if (a > b and op="00") then
      y <= a - b;
      z <= a - 1;
      status <= '0';
   else
      y <= b - a;
      z <= b - 1;
      status <= '1';
   end if;
end process;
```

We would need to repeat the Boolean expression in the *if stmt* in all three of the equivalent conditional signal assignment stmts

What happens when there is no *elsif* or *else* stmt or one or more signals are not assigned to within an *if*, *elsif* or *else* branch?

If Statement

The signal that is unassigned **keeps** the *previous value* (implying memory)

```
process(a, b)
    begin
    if (a = b) then
        eq <= '1';
    end if;
end process;
```

No *else*, no action is taken when *a* does not equal *b* -- is equivalent to

```
process(a, b)
    begin
    if (a = b) then
        eq <= '1';
    else
        eq <= eq;
    end if;
end process;
```

If Statement

For combo logic, the *else* branch MUST be included as shown below to avoid unwanted memory or a latch

```
process(a, b)
   begin
   if (a = b) then
      eq <= '1';
   else
      eq <= '0';
   end if;
end process;
```

A similar situation occurs when a signal is assigned in some branches but not others

```
process(a, b)
   begin
   if (a > b) then
      gt <= '1';
   elsif (a = b) then
      eq <= '1';
```

If Statement

```
        else
           lt <= '1';
        end if;
     end process;
```

VHDL semantics indicate a signal will **keep its previous value** if it is not assigned, so ALL three assignments MUST be made in each branch, or

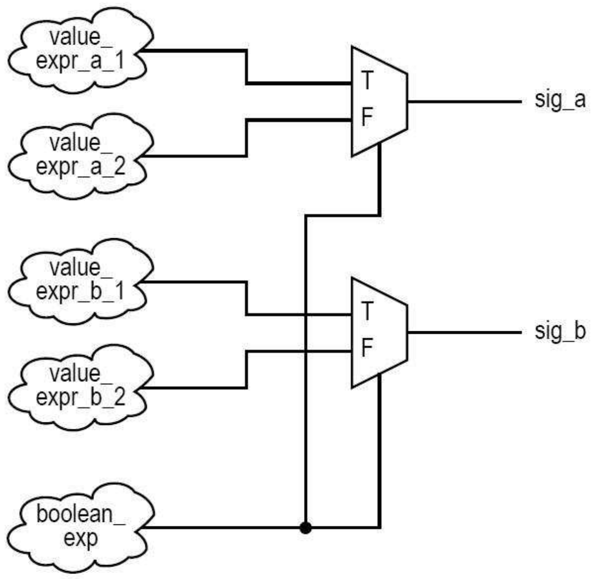
```
     process(a, b)
        begin
        gt <= '0';
        eq <= '0';
        lt <= '0';
        if (a > b) then
           gt <= '1'; -- last assignment takes precedence
        elsif (a = b) then
           eq <= '1'; -- last assignment takes precedence
        else
           lt <= '1'; -- last assignment takes precedence
        end if; end process;
```

If Statement: Conceptual Implementation

When the *if stmt* consists of a single assignment, the hardware is identical to the conditional signal assignment stmt

When there are multiple assignments, the implementation can be constructed recursively

```
if (boolean_expr) then
   sig_a <= value_expr_a_1;
   sig_b <= value_expr_b_1;
else
   sig_a <= value_expr_a_2;
   sig_b <= value_expr_b_2;
end if;
```

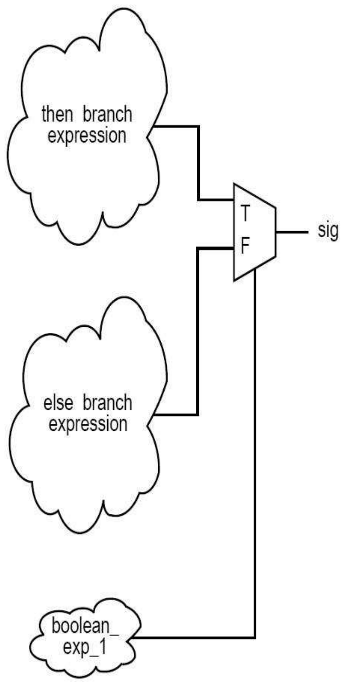


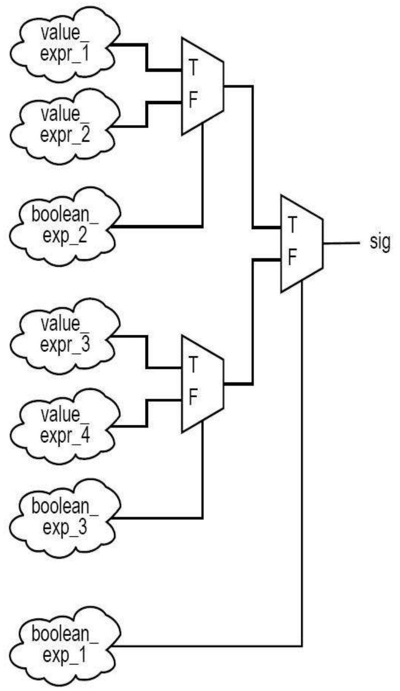
For nested *if stmts*, the conceptual diagram is constructed in a hierarchal manner

If Statement: Conceptual Implementation

First derive the routing structure for the outer *if stmt*

```
if bool_expr_1 then
   if bool_expr_2 then
      sig_a <= value_expr_1;
   else
      sig_a <= value_expr_2;
   end if;
else
   if bool_expr_3 then
      sig_a <= value_expr_3;
   else
      sig_a <= value_expr_4;
   end if;
end if;
```





A priority structure can also be constructed using a default assignment and a sequence of '**if** ... **end if**' stmts (see text for alternate version of priority encoder)

Case Statement

Syntax

```
case case_expression is
   when choice_1 =>
      sequential statements;
   when choice_2 =>
      sequential statements;
   ...
   when choice_n =>
      sequential statements;
end case;
```

The *case_expression* term functions just like the *select_expression* term in a selected signal assignment stmt

Its data type MUST be a discrete tpe or 1-D array

As was true for selected signal assignment, *choice_i* terms must be **mutually exclusive** and **all inclusive** (keyword **others** may be used to cover all unused values)

Case Statement

The *case stmt* applied to the MUX, decoder, priority decoder and simple ALU

```
architecture case_arch of mux4 is
   begin
   process(a, b, c, d, s)
      begin
      case s is
         when "00" =>
            x <= a;
         when "01" =>
             x <= b;
         when "10" =>
             x <= c;
         when others =>
             x <= d;
      end case;
   end process;
end case_arch;
```

Case Statement

```
architecture case_arch of decoder4 is
   begin
   process(s)
      begin
      case s is
         when "00" =>
            x <= "0001";
         when "01" =>
            x <= "0010";
         when "10" =>
            x <= "0100";
         when others =>
            x <= "1000";
      end case;
   end process;
end case_arch;
```

Case Statement

```
architecture case_arch of prio_encoder42 is
   begin
   process(r)
      begin
      case r is
         when "1000"|"1001"|"1010"|"1011"|
              "1100"|"1101"|"1110"|"1111" =>
            code <= "11";
         when "0100"|"0101"|"0110"|"0111" =>
            code <= "10";
         when "0010"|"0011" =>
            code <= "01";
         when others =>
            code <= "00";
      end case;
   end process;
   active <= r(3) or r(2) or r(1) or r(0);
end case_arch;
```

Case Statement

```
    architecture case_arch of simple_alu is
       signal src0s, src1s: signed(7 downto 0);
       begin
       src0s <= signed(src0);
       src1s <= signed(src1);
       process(ctrl, src0, src1, src0s, src1s)
          begin
          case ctrl is
             when "000"|"001"|"010"|"011" =>
                result <= std_logic_vector(src0s + 1);
             when "100" =>
                result <= std_logic_vector(src0s + src1s);
             when "101" =>
                result <= std_logic_vector(src0s - src1s);
             when "110" =>
                result <= src0 and src1;
             when others =>    -- "111"
                result <= src0 or src1;
```

Case Statement

```
          end case;
       end process;
    end case_arch;
```

Comparison to *selected signal assignment* stmt:

Two statements are the same if there is only **one** output signal in case statement

```
with select_expression select
   sig <= value_expr_1 when choice_1,
          value_expr_2 when choice_2,
          value_expr_3 when choice_3,
          ...
          value_expr_n when choice_n;
```

Can be written as

```
case case_expression is
   when choice_1 =>
      sig <= value_expr_1;
```

Comparison of Case Statement with Selected Signal Assignment Statement

```
      when choice_2 =>
         sig <= value_expr_2;
      when choice_3 =>
         sig <= value_expr_3;
      ...
      when choice_n =>
         sig <= value_expr_n;
   end case;
```

Case statement is more flexible because multiple sequential statements can be included in each of the branches

Incomplete Signal Assignment

Any 'incomplete when clause' is a syntax error

However, no such restriction exists for signal assignments, i.e., signals do **not** need to be assigned in every 'choice_i' case

Incomplete Signal Assignment in Case Statement

When a signal is **unassigned**, it keeps the previous value, which implies memory

```
process(a)
   begin
   case a is
      when "100" | "101" | "110" | "111" =>
         high <= '1';
      when "010" | "011" =>
         middle <= '1';
      when others =>
         low <= '1';
   end case;
end process;
```

This does **not** behave as expected, e.g., if *a* is "111", then *high* gets assigned '1' but *middle* and *low* are left unassigned.

This **infers** three unwanted memory elements for *high*, *middle* and *low*

Incomplete Signal Assignment in Case Statement

You can fix by assigning to *high*, *middle* and *low* in EVERY case or

```
process(a)
   begin
   high <= '0';
   middle <= '0';
   low <= '0';
   case a is
      when "100" | "101" | "110" | "111" =>
         high <= '1';
      when "010" | "011" =>
         middle <= '1';
      when others =>
         low <= '1';
   end case;
end process;
```

Case Statement: Conceptual Implementation

Same as *selected signal assignment stmt* if the *case stmt* consists of

- One output signal
- One sequential signal assignment in each branch

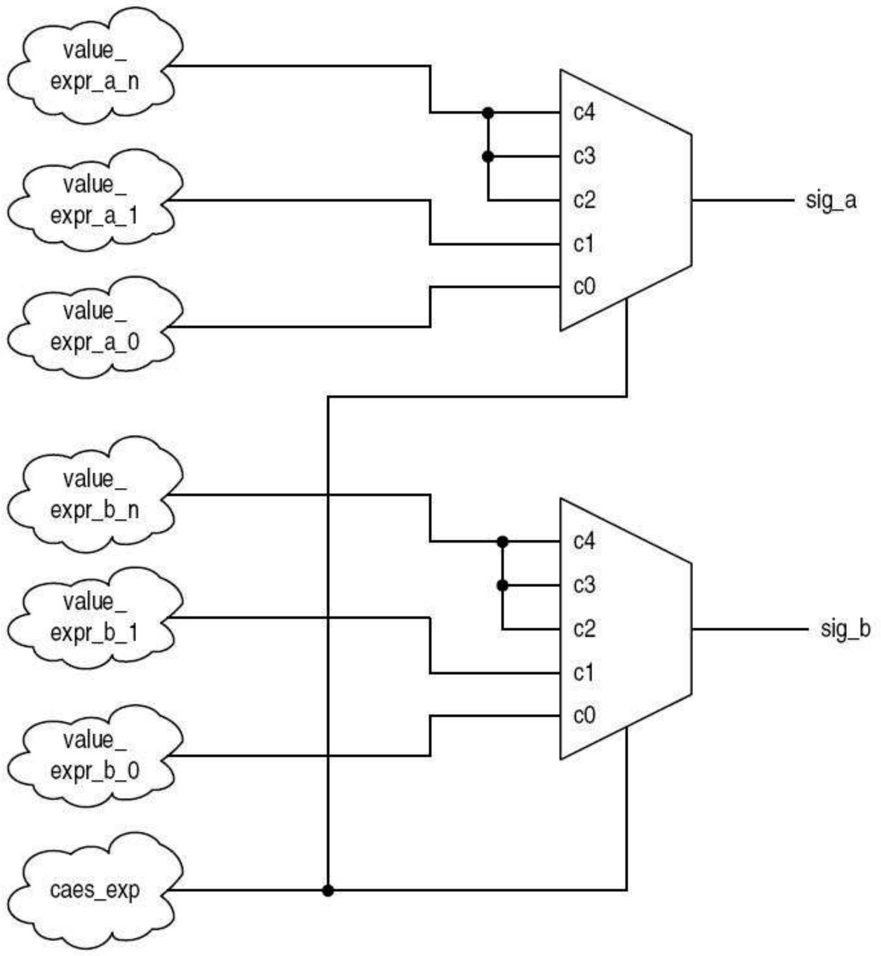
Multiple sequential statements can be constructed recursively

Consider

```
case case_exp is
    when c0 =>
        sig_a <= value_expr_a_0;
        sig_b <= value_expr_b_0;
    when c1 =>
        sig_a <= value_expr_a_1;
        sig_b <= value_expr_b_1;
    when others =>
        sig_a <= value_expr_a_n;
        sig_b <= value_expr_b_n;
end case;
```

Case Statement: Conceptual Implementation

case stmts can include other *case stmts* inside a *when* clause, and therefore a recursive application of the following is required



Simple For Loop Statement

VHDL provides a variety of loop constructs including the *simple infinite loop*, *for loop* and *while loop*, as well as mechanisms to terminate a loop (*exit* and *next*)

However, only a restricted form of a loop can be synthesized

Syntax

```
for index in loop_range loop
    sequential statements;
end loop;
```

loop_range must be static, and *index* assumes value of *loop_range* from left to right

index assumes data type of *loop_range* and does not need to be declared

Flexible and versatile but can be difficult or impossible to synthesize

4-bit xor circuit (NOTE: This is easily accomplished alternatively using 'y <= a xor b;')

```
library ieee;
use ieee.std_logic_1164.all;
```

Simple For Loop Statement

```
entity bit_xor is
   port(
      a, b: in std_logic_vector(3 downto 0);
      y: out std_logic_vector(3 downto 0)
   );
end bit_xor;

architecture demo_arch of bit_xor is
   constant WIDTH: integer := 4;
   begin
   process(a, b)
      begin
      for i in (WIDTH-1) downto 0 loop
         y(i) <= a(i) xor b(i);
      end loop;
   end process;
end demo_arch;
```

Simple For Loop Statement

reduced-xor: performs an xor operation over a group of signals

For example, the reduced-xor of a_3, a_2, a_1, and a_0 is a_3 **xor** a_2 **xor** ... a_0

```
library ieee;
use ieee.std_logic_1164.all;
entity reduced_xor_demo is
   port(
      a: in std_logic_vector(3 downto 0);
      y: out std_logic
   );
end reduced_xor_demo;

architecture demo_arch of reduced_xor_demo is
   constant WIDTH: integer := 4;
   signal tmp: std_logic_vector(WIDTH-1 downto 0);
   begin
```

Simple For Loop Statement

```
      process(a, tmp)
         begin
         tmp(0) <= a(0);    -- boundary bit
         for i in 1 to (WIDTH-1) loop
            tmp(i) <= a(i) xor tmp(i-1);
         end loop;
      end process;
      y <= tmp(WIDTH-1);
   end demo_arch;
```

For a conceptual implementation, **unroll** the loop and replicate the code inside the loop

```
tmp(0) <= a(0);
tmp(1) <= a(1) xor tmp(0);
tmp(2) <= a(2) xor tmp(1);
tmp(3) <= a(3) xor tmp(2);
y <= tmp(3);
```

Will be extremely useful in *parameterized design*