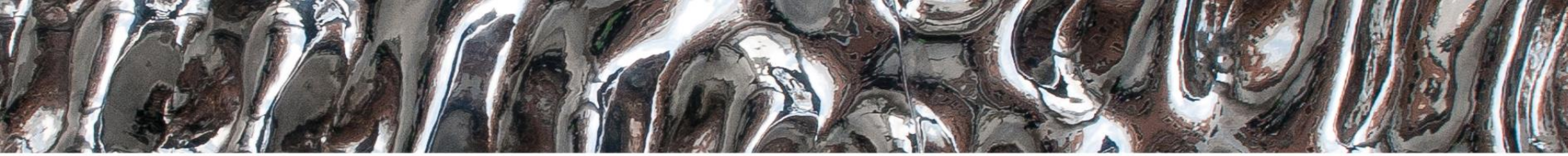


UiO : Department of Informatics
University of Oslo

IN3160, IN4160

1: Subroutines, packages and libraries

2: Clocked statements

Messages

- Remember to **log out** after using the lab machines
  - Reduces the need for reboot (USB port locked to user)

- Lisp has been nice and tidy! ☺
  - (not due to lack of access i hope...)

Goal

- Learn how to create subroutines using VHDL
- Learn good practice for writing subroutines
- Learn which packages are most used in VHDL
- Learn how to use and create libraries and packages in VHDL

Overview

- Subroutine types
  - Functions
  - Procedures
- Functions and operators
- Procedures
- Overloading in VHDL
- Libraries
  - Package/package body
- Standard libraries
- **Clocked statements**

Next: Verification & file IO

Why Subroutines

- *avoid duplicating code*
  - Make the code more readable
  - Reduce code complexity
  - Make the code easier to maintain
  - Make code easier to test or verify

VHDL Subroutine types and practice:

- Two types:
  - Functions – *returns* one *value*
  - Procedures – *a group of statements*

- General good practice:
  - **use functions!**
  - Limit the use of procedures to testbenches
    - Consider functions, entites or processes before using procedures
      - *Using procedures to create HW is sometimes used to generate structure*
        - » *Ie. not RTL code.*

Functions-

parameter(s)

single return value

```
function sum_function(vect: integer_vector) return integer is
  variable sum: integer := 0;
begin
  for i in vect’range loop
    sum := sum + vect(i);
  end loop;
  return sum;
end;
```

```
b <= std_logic_vector(to_signed(
  sum_function((
    to_integer(signed(a2)),
    to_integer(signed(a1)),
    to_integer(signed(a0)) )),
  b'high - b'low + 1) );
```

The «extra» paranthesis is needed to make one vector out of the three integers. Without, they will be interpreted as three separate parameters of wrong type

- take one or more parameters
  - Parameters in functions
    - Can not be changed/manipulated
      - always mode “in”
    - (only) *constant*, *signal* or *file*
      - *constant is default*
  - Parameters are separated by ‘;’
    `(a: bit; b: my_type;… )`
- return only a <u>single *value*</u>
  - The value can be of any *type*
    - *Including vectors and custom types*
- *pure* functions use only their input parameters => CL
- *Impure* functions make use of data visible where they are declared (as parameters)
  - *Ex: File IO (next lecture)*
- Cannot have wait- statements. (execution within a single simulation cycle)
- Cannot have internal signals (no storage)

Function usage example:

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

entity subprogram is
  generic( k: positive := 4 );
  port(
    a2, a1, a0: in std_logic_vector(k-1 downto 0);
    b : out std_logic_vector(k-1 downto 0) );
end subprogram;

architecture example of subprogram is
  function sum_function(vect: integer_vector)
    return integer is
    variable sum: integer := 0;
  begin
    for i in vect'range loop
      sum := sum + vect(i);
    end loop;
    return sum;
  end;
```

```
begin
  local: process(all) is
     variable v: integer_vector(2 downto 0);
     variable sum: integer;
     variable s: signed(b'high-b'low downto b'low);
  begin
    v:=(
      to_integer(signed(a2)),
      to_integer(signed(a1)),
      to_integer(signed(a0)) );

    sum := sum_function(v);

    s := to_signed(sum, b'high - b'low + 1);
    b <= std_logic_vector(s);
  end process local;

end architecture example;
```

More Functions…

- Can be used for both synthesis and simulation
- Can (also) be *overloaded* (two or more functions having same name)
  - different parameters and or return type
- Are declared in the declarative region of
  - architectures
  - processes
  - packages (declaration and body – example later)
- Are frequently used for
  - Computation
  - Type converting
- Packages in libraries we use typically define functions
  - IEEE (library)
    - std_logic_1164 (package)
    - numeric_std
    - …
  - *We use these all the time…*

```
architecture func_arch of functest is

-- declarations
function bool2bit(a: boolean) return bit is
begin
  if a then
    return ‘1’;
  else
    return ‘0’;
  end if;
end bool2bit;

-- statements
begin
  …
end func_arch;
```

Procedures…

- do not have a return value
- can have
  - **in** and **out** *parameters*
    - in is default
    - out parameters (must be set)
  - **wait** statements
  - signals
  - file access
- cannot be used in a statement
  - *Only standalone «calls»*
- Are typically used in test benches
  - Reading test vectors from file
  - Applying test vectors
  - Writing test results to file

parameter(s)

no return value

```
procedure sum_proc(vect: in integer_vector;
                   sum: out integer) is
begin
  for i in vect'range loop
    sum := sum + vect(i);
  end loop;
end;
-- «DO NOT DO THIS AT HOME»
-- (can and should be solved using function)
```

!

- *Can manipulate both* **out**-*parameters and other* **signals** declared in the same (underlying) region...

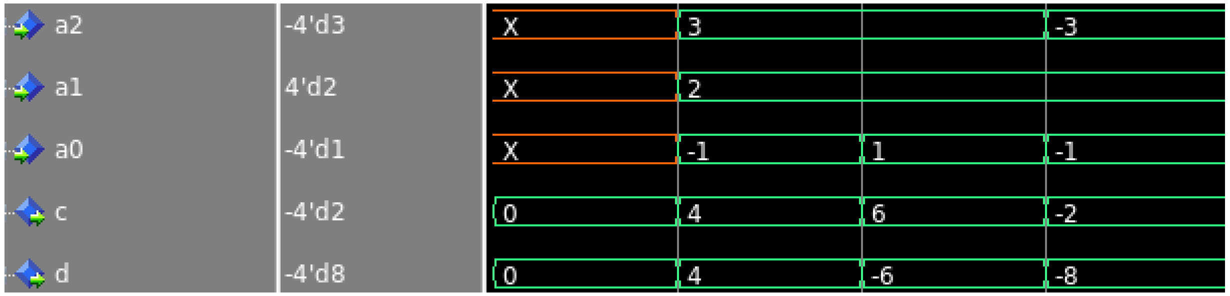
!

Example- when not to use procedure:

```
process(all) is
     variable v: integer_vector(2 downto 0);
     variable sum: integer;
     variable s: signed(b'high-b'low downto b'low);
  begin
    v:=(
      to_integer(signed(a2)),
      to_integer(signed(a1)),
      to_integer(signed(a0)) );

    sum := sum_function(v);

    s := to_signed(sum, b'high - b'low + 1);
    c <= std_logic_vector(s);
  end process local;
```

Only difference apart from declarations (previous slides)

```
process(all) is
    variable v: integer_vector(2 downto 0);
    variable sum: integer;
    variable s: signed(b'high-b'low downto b'low);
  begin
    v:=(
      to_integer(signed(a2)),
      to_integer(signed(a1)),
      to_integer(signed(a0)) );

    sum_proc(v,sum);

    s := to_signed(sum, b'high - b'low + 1);
    d <= std_logic_vector(s);
  end process;
```



```
a2   -4'd3   X   3        -3
a1   4'd2    X   2
a0   -4'd1   X   -1   1   -1
c    -4'd2   0   4    6   -2
d    -4'd8   0   4    -6  -8
```

- **the procedure is not CL.**
  - *sum accumulates*
  - *process variable -> single instantiation*
- In HW, d would be unstable due to this feedback loop, since the process is not clocked.
- *-6 and -8 is the result of the 4 digit two-complement representation.*

- Why aren't c and d equal?

Functions vs Procedures

Functions	Procedures
Returns a value (*can be vector*) of any type	A collection of statements (Sets signals)
Can only use variables, no signals.	Can contain both signals and variables that will be hidden from the outside. May use signals from the underlying structure.
Cannot replace procedures fully	*Can* replace functions (DON'T DO THAT!)
Much used in conversions (from bit to STD_LOGIC, from some_type to my_type, etc).	Much used for repetitive tasks- particularly in test benches.
Typically found in libraries and packages	*Mostly used for simulation/ test benches.*
Always "instant" (CL), never time based	Can use "wait" and timing information.
Can be used in statements… `a <= my_func(…);`	Can only be used standalone… `my_procedure(..);`

Neither can store internal values between calls.

Parameters for subprograms

Parameters or «interface objects» have up to five parts

1. Class : **constant** (default), **variable**, **signal**, **file**
2. Identifier: the name you decide must be defined
3. Mode: **in** (default) or **out**
4. Type: **std_logic**, **integer**, **bit**, **text**, … must be defined
5. Default value := optional

Ex:

```
procedure apply_vectors(
  file vector_input : text;
  addend : integer := 42;
  signal valid : out boolean;
  file vector_output : text);
```

Good practice

- When considering to create a subprogram:
  - Is it possible to do this using a function?
    - Yes: **use function**
    - No:.. Is it for creating HW?
      - If yes: consider a process, or a separate entity + architecture
  - Is it for simulation only, and a function will not do:
    - Use a procedure

- Subprograms generally should have a single purpose.
  - Try see if the purpose can be said in one sentence without use of “and” or “or”…

Good practice

- Use functions when you can
  - Limitations in functions makes it easier to achieve well structured code
    - readable
    - maintainable
    - short
- Limit procedures to test bench code
  - It is easy to create messy code using procedures since they allow
    - multiple in and out parameters
    - to use signals and create storage elements

Packages

- In a package declarative region you can add:
  - Component declarations
  - Data type definitions
  - Constants
  - Subprogram declarations
    - Functions
    - Procedures

- The declarative region is publicly visible
  - similar to header files in C

- Package body-
  - declarations is not publicly visible
  - typically contains content of-
    - subprograms
    - components

```
package my_pkg is
  -- publicly visible declarations
  type imb_vec is record
    re: bit_vector;
    im: bit_vector;
  end record;

  constant IMB_VEC1: imb_vec := (re => "010", im => "001");
  function bool2bit(a: boolean) return bit;
  …
end;

Package body my_pkg is
  -- non visible, internal declarations
  function bool2bit(a: boolean) return bit is
  begin
    …
  end bool2bit;
  …
end my_pkg;
```

Packages

Save and compile your package in the work folder

To use package contents, include these two lines:

```vhdl
library work;
use work.my_package.all;
```

```vhdl
-- Package Declaration Section
package my_package is

  constant c_PIXELS : integer := 65536;

  type t_my_rec is record
    full: std_logic;
    empty: std_logic;
  end record t_my_rec;

  component my_component is
    port (i_data  : in  std_logic; o_res : out std_logic);
  end component my_component;

  function Bit_OR (i_vec : in std_logic_vector(3 downto 0))
      return std_logic;

end package my_package;

-- Package Body Section
package body my_package is

  function Bitwise_OR (i_vec : in std_logic_vector(3 downto 0))
    return std_logic is
  begin
    return (i_vec (0) or i_vec (1) or i_vec (2) or i_vec (3));
  end;

end package body my_package;
```

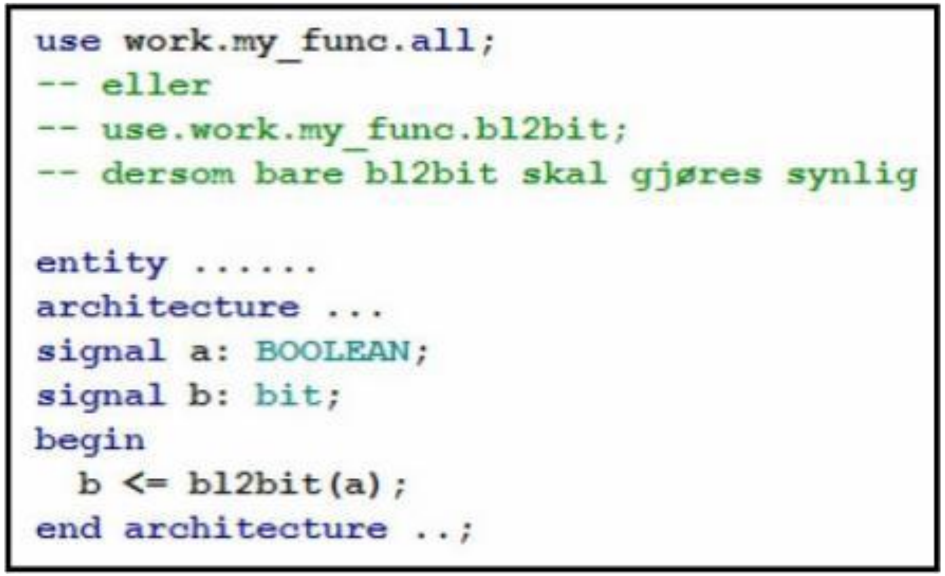
Typical use of packages

- Typically packages is organized such that it contains
  - custom or abstract data types
    - Ex: I have a project that will incorporate calendar data
      - => lets make a package for all types used
  - functions that work on these abstract data types.

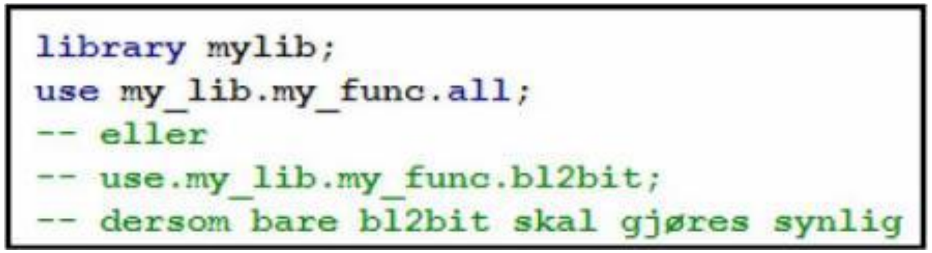
- Create packages when you have
  - function(s) that may be used by more than one design unit.
  - Types that may be used in several design units
  - Components that may be used in several designs
  - Simulation -models, -procedures and -functions that can be re-used

Use of functions library

- A “package/body” pair can be compiled to work or to another library.
  - This needs to have a logic name. Here: mylib
- The logic name is given in the current tool and is reflected in the directory structure of the tool



```
use work.my_func.all;
-- eller
-- use.work.my_func.bl2bit;
-- dersom bare bl2bit skal gjøres synlig

entity .......
architecture ...
signal a: BOOLEAN;
signal b: bit;
begin
  b <= bl2bit(a);
end architecture ..;
```



```
library mylib;
use my_lib.my_func.all;
-- eller
-- use.my_lib.my_func.bl2bit;
-- dersom bare bl2bit skal gjøres synlig
```

Operators

- Operators are defined in the same way as functions, but by “<operator name>”
- Operators are being used differently from functions
- You can create overloaded operators (ie ‘+’ for my_type),
  - *but not create new*

```
-- package declaration (overload)
 function "+" (a,b :std_logic_vector) return std_logic_vector;

 ...
 -- usage
 sum <= a + b;


 -- package declaration (non overload)
 function add (a,b :std_logic_vector) return std_logic_vector;

 ...
 -- usage
 sum <= add(a, b);
```

Overloading

- **Overloading** means defining the same operator-, function- or procedure-name for different data types or a mix of data types.
- Overloaded subprograms (operators, functions and procedures) may have different number of parameters
- Synthesis tools separates the usage of overloaded subprograms by comparing actual parameters (those in use) with formal parameters (in the subprogram declaration)

Overloading

- There are a lot of standard libraries with overload operators, functions and procedures in IEEE 1164 and IEEE 1076.3
  - IEEE *1164*
    - ***Package std_logic_1164***
    - Synopsis libraries (compiled to IEEE, but not standard)
      - Package std_logic_unsigned
      - Package std_logic_signed
      - Package std_logic_arith
      - Package std_logic_textio
      - *Don't use these in this course.*
        - » *Std libraries covers the usage and there are some differences.*
  - IEEE *1076.3*
    - ***Package numeric_std***
    - Use the package `IEEE.numeric_std` for integer arithmetics with the use of the data types **signed** and **unsigned**

UiO : Department of Informatics
University of Oslo

IN 3160, IN4160

Clocked processes and statements

Yngve Hafting

Course Goals and Learning Outcome

https://www.uio.no/studier/emner/matnat/ifi/IN3160/index-eng.html

In this course you will learn about the design of advanced digital systems. This includes programmable logic circuits, a hardware design language and system-on-chip design (processor, memory and logic on a chip). Lab assignments provide practical experience in how real design can be made.

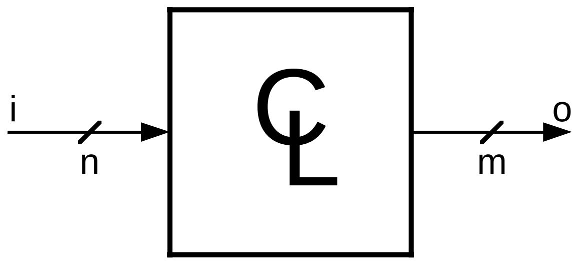
After completion of the course you will:

- understand important principles for design and testing of digital systems
- **understand the relationship between behaviour and different construction criteria**
- **be able to describe advanced digital systems at different levels of detail**
- be able to perform simulation and synthesis of digital systems.

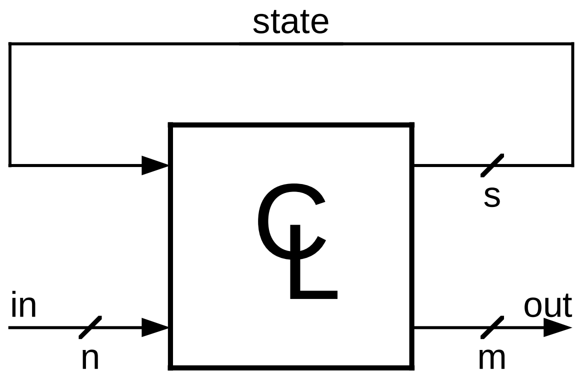
Goals for this lesson:

- Know different approaches to achieve clocked logic in VHDL
  - Why they exists
  - Benefits and pitfalls
  - ...

Sequential logic has state



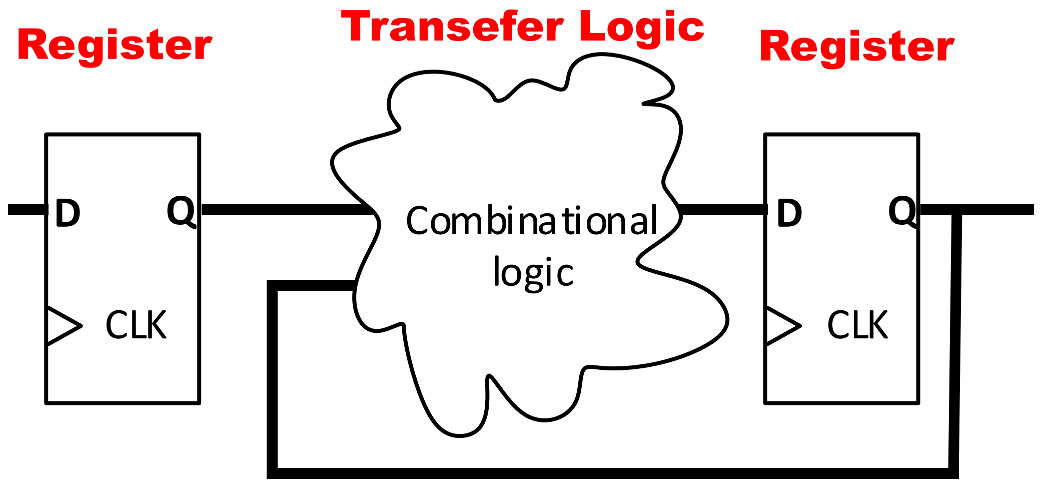
Combinational Logic



Sequential Logic

Do not use feedback into CL without using flipflops!

Sequential design = CL + FFs



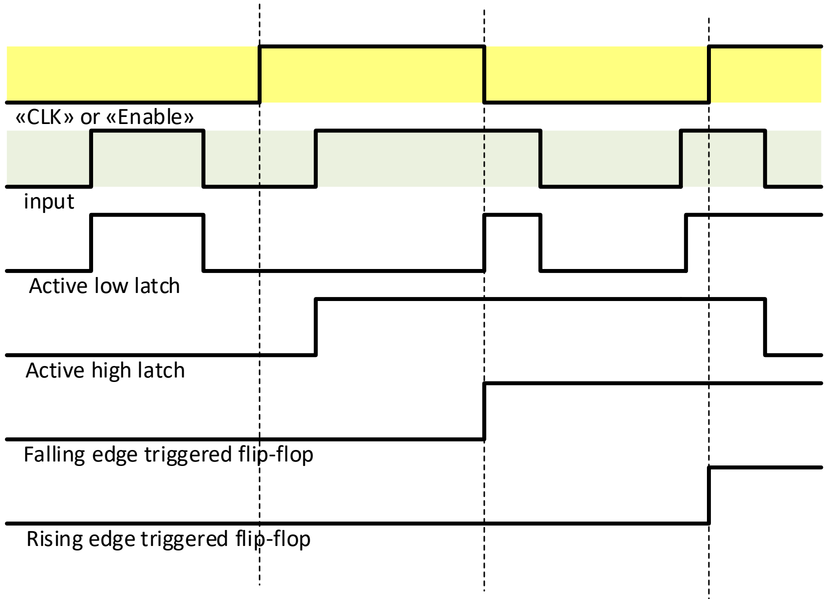
=> RTL design

- Good practice:
  - Keep CL and register storage separate
  - *Know* when you infer storage elements!

- Sequential designs *are* state machines
  - *Sometimes* they have other names
    - *Counters*
    - *Shift Registers*
    - *LFSR – Linear Feedback shift Registers*
    - *...*

Latch vs Flip-flop

- The functions `rising_edge` and `falling_edge` gives a true (0->1 or 1->0) edge detection
  - `if CLK'event and CLK = '1' then` reacts on all transitions to '1', for example U->1 *(simulation)*
- NB! *An incomplete* conditional statement will be synthesized to a latch* (implied memory)
  - *complete defines all outputs for all conditions of the input variable(s).


«CLK» or «Enable»
input
Active low latch
Active high latch
Falling edge triggered flip-flop
Rising edge triggered flip-flop

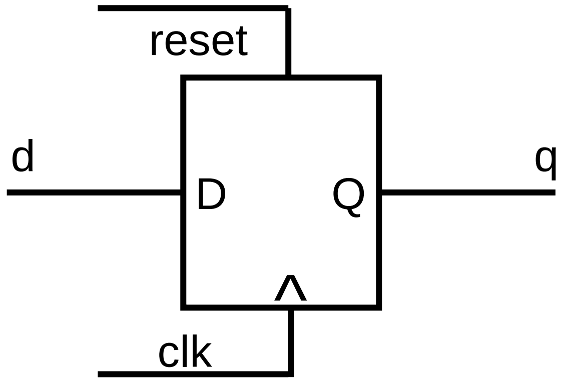
```
architecture RTL_DFF of DFF is
begin
process(CLK)
begin
  if rising_edge(CLK) then
    Q <= D;
  end if;
end process;
end architecture RTL_DFF;
```

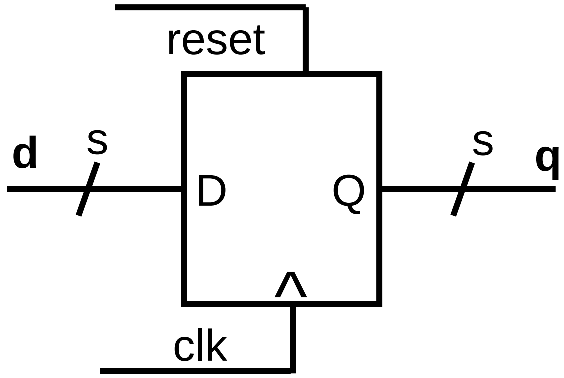
```
architecture RTL_DL of DL is

begin
  process(ENABLE,D)
  begin
    if ENABLE = '1' then
      Q <= D;
    end if;
  end process;
end architecture RTL_DL;
```

D Flip-Flop

- Input: D
- Output: Q
- Clock

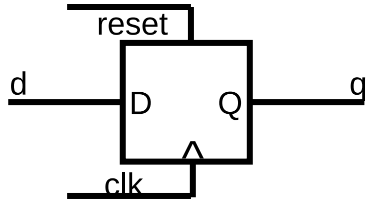
- Q outputs a steady value
- Edge on Clock changes Q to be D
- Flip-flop stores state
- Allows sequential circuits to iterate





D-flip-flop with asynchronous reset

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

entity DFF is
  port(
    clk, reset, d : in std_logic;
    q : out std_logic);
end entity DFF;
```



one-liner... (VHDL 2008 style)
- Compact & readable for simple structures

```
architecture oneliner of DFF is
begin
  q <= '0' when reset else d when rising_edge(clk);
end architecture;
```

- reset has priority
- Both clk and reset in sensitivity lists
  - **NEVER use 'all' for clocked sensitivity lists**

```
architecture signal_and_process of DFF is
begin
  process(clk, reset) is
  begin
    if reset then
      q <= '0';
    elsif rising_edge(clk) then
      q <= d;
    end if;
  end process;
end architecture;
```

Old style...
This style is mostly what you will find from old code (internet) and ChatGPT
This can ensure tests for reset and clock edge is in one place only

Variables *can* be used for FF instantiation, but...

No FF is created unless a signal is assigned to the state variable

```
architecture variabled of DFF is
begin
  process(clk, reset) is
    variable state;
  begin
    if reset then
      state := '0';
    elsif rising_edge(clk) then
      state := d;
    end if;
    Q <= state;
  end process;
end architecture;
```

Synchronous D-flip-flop

- Clock has priority
- Only clk in sensitivity list
  - (not reset, and *definetely not 'all'*)

```
architecture synchronous_reset of DFF is
begin
  process(clk) is
  begin
    if rising_edge(clk) then
      if reset then
        q <= '0';
      else
        q <= d;
      end if;
    end if;
  end process;
end architecture;
```

- Old style...
- This style is mostly what you will find from old code (internet)
- *Consider* other styles to avoid making messy code...

- Compact and easy to read
- Slightly more work to maintain if you change name of the reset signal

```
architecture sync_compact of DFF is
begin
  process(clk) is
  begin
    if rising_edge(clk) then
      q <= '0' when reset else d;
    end if;
  end process;
end architecture;
```

```
architecture separate_CL of DFF is
  signal next_state : std_logic;
begin
  next_state <= '0' when reset else q;
  q <= next_state when rising_edge(clk);
end architecture;
```

- Relatively compact and easy to read
  - *Require next_state-signals*
- Separates combinational logic from sequential storage.
- Can also be achieved using processes and variables

- Forces you to **know your storage elements**...

Generally when writing RTL code:

- Separate use of CL and clocked processes
  - Keep (all) register assignment in one place
  - Use one process for each purpose : ex
    - process for state assignment
    - process for state output
    - process for registers
- Avoid unintentional states by not combining CL and registers(!)

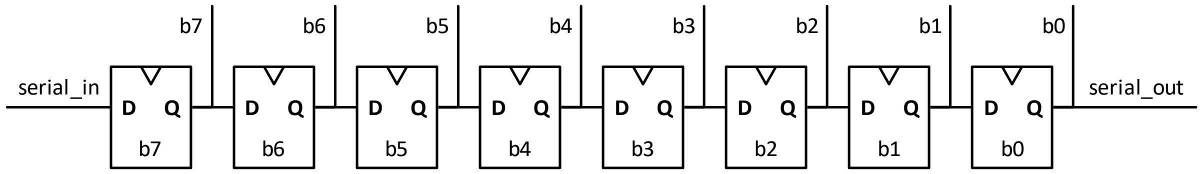
- Use synchronous reset unless specific reasons for async reset.
  - (reset circuits will be covered later)

- Simple statements can be written concurrently

```
architecture processed of RTL-module is
  signal next_x, next_y, ... : std_logic;
begin
  CL: process(all)is
  begin
     -- Create next based on input and registers>
     next_x <= ...
     next_y <= ...
  end process;

  registers: process(clk) is
  begin
     if rising_edge(clk) then
        -- registers based on next signal alone on clk
        x <= next_x;
        y <= next_y;
     end if;
  end process;

end architecture;
```

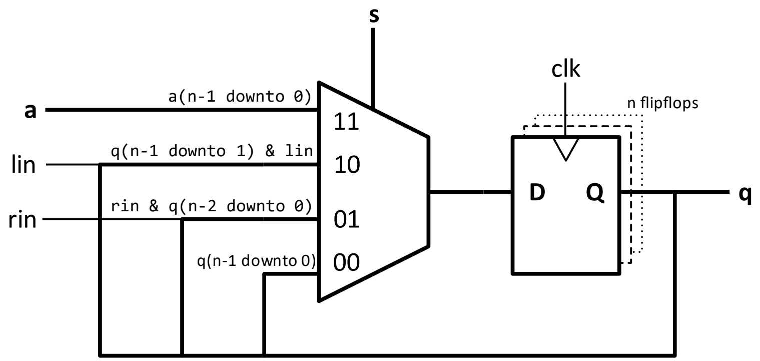
```
architecture concurrent of RTL-module is
  signal next_x, next_y, ... : std_logic;
begin
 -- concurrent statements (CL)
 next_x <= ...
 next_y <= ...

 registers:
 y <= next_y when rising_edge(clk);
 x <= next_y when rising_edge(clk);

end architecture;
```

Shift registers (not shifters)



- Shifter = CL
- Shift register = Flipflops connected in series
  - Used to parallelize serial input
    - Serial data transfer is used for high speed IO over distance
      - Ie largest parallel high speed io is memory buses on a PC main board
  - Can sometimes be used both ways
    - Ie both serial and parallel in/out

  - Assignment 4 lets you attempt this using structural code.



Parity calculation in VHDL 2008

- VHDL-2008 adds Unary Reduction Operators of the form:

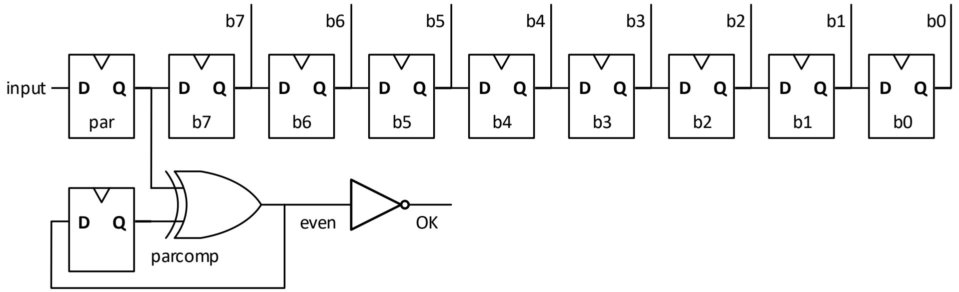
```
function "xor" ( anonymous: BIT_VECTOR) return BIT;
```

- Defined for arrays of bit and std_ulogic
- Defined for all binary logic operators:
  - AND, OR, XOR, NAND, NOR, XNOR

- Simplifies parity calculation

```
signal data : std_logic_vector(7 downto 0) ;
signal parity : std_logic;
. . .
parity <= xor data; -- even parity
```

Serial parity check



- Even parity
  - parity bit is even (‘0’) when there is an even number of bits that are ‘1’
  - Using even parity bit, each byte transmission (*including parity bit*) should always have even parity.
    - OK signal is high when even is ‘0’.

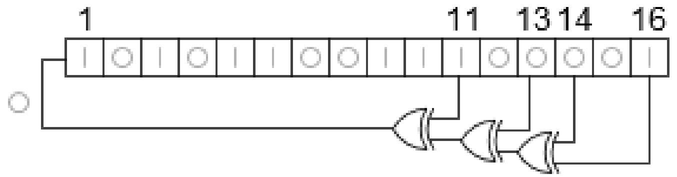
- Odd parity is «**`not`** `even`»

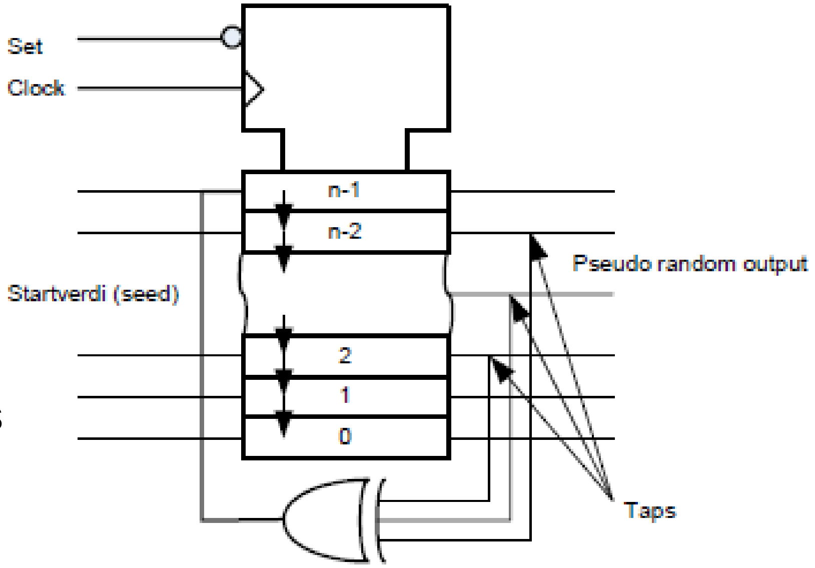
Linear Feedback Shift Register(LFSR)

- Made by xor-ing one and one bit that are connected back to MSB

- Apparently a random counting sequence
  - Nicknamed "Pseudo-random generator" since the counting sequence looks random
- It can be shown that it's not needed more than three xor gates to make a random sequence
- *Some combinations are better* https://web.archive.org/web/20161007061934/http://courses.cse.tamu.edu/csce680/walker/lfsr_table.pdf

- Used in testing of communication lines and buses
- Used in encryption





Oblig 3, Reccomended reading

Challenge next page..

- Oblig 3:
  - Peer review is required for passing
  - 2 peer review will be assigned to each
  - When in trouble, call the lab-assistant.
  - Be polite!

- Subprograms: This lecture
- Clocked processes and statements:
  - D&H:
    - 14.1-2 p 305-309,
    - 16.1-2 p 344-356

(If-15 min...) challenge:

- 5 different architectures...
- Fill in what X is based on the input signals (in the table)
- How many FF's are created here?
- What type of circuit is this / What does it do?
- Raise you hand when finished...
- We will discuss and elaborate after

```
entity XXX is
  port (Clock : in  Std_logic;
  Reset : in  Std_logic;
  Enable: in  Std_logic;
  Load  : in  Std_logic;
  Mode  : in  Std_logic;
  Data  : in  Std_logic_vector(7 downto 0);
  X     : out Std_logic_vector(7 downto 0));
end;
```

Enable	Load	Mode	X
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

https://www.uio.no/studier/emner/matnat/ifi/IN3160/v21/10-min-challenge/

To be revealed in the lecture