TSTE12 Design of Digital Systems Lecture 2

Kent Palmkvist

LINKÖPING UNIVERSITY

Agenda

- Practical issues
- Introduction to VHDL
  - Simple design examples

TSTE12 Practical Issues

- Make sure you are registered to the course
  - Check that you have access to the lisam group room for 2023 version of the course
- Sign up for lab group
  - Sign up function in the Lisam course room
  - Select A or B group

TSTE12 Lab info

- Lab open 05-23 each day, 7 days/week
- Lab group defined to guarantee computer access
  - Unused computers available for other group
- MUX2 lab available (starting wednesday)
  - Initially limited to scheduled hours (unlocked lab door)
  - Later access given through LiU card
- MUX1 lab also possible (but used more in other courses)
  - Make sure the check with the schedule server (timeedit) that other course not uses the lab before entering

TSTE12 Deadlines Y,D,ED

- Group definitions Thursday 31 August (afternoon)
  - On web, include supervisor assignment
- Friday 1 September (possible also monday 4/9): First meeting with supervisor
  - Determine project manager (contact person)
  - Questions (short meeting)
- Tuesday 5 September: First version of requirement specification
- Hint: Deadline means ”no later than”, i.e., allowed to complete tasks before these dates

TSTE12 Deadlines MELE, erasmus

- Group definition Wednesday 6 September (afternoon)
  - On web, include supervisor assignment
- Friday 8 September: First meeting with supervisor
  - Determine project manager (contact person)
  - Question (short meeting)
- Tuesday 12 September: First version of requirement specification

Deciding meeting with supervisor

- Supervisors work with multiple courses
- Meeting with supervisor decided by signing up on paper list outside his office (or other method defined by the supervisor)
  - Corridor B, 2nd floor, entrance 27 (towards entrance 25)
  - List shows available timeslots for meetings
- Sign up day before meeting
  - Supervisors needs to know their day in the morning

Documents to be discussed must be submitted at least 24h before the meeting time

Computer system intro

- More info about the computer system at

  https://liuonline.sharepoint.com/sites/student-campus-och-lokaler/SitePages/en/Datorsalar.aspx
  - Require login
- Single password for all computers
  - Same files and folders (home folder) for windows as well as linux
  - We use linux (CentOS 7) that is unique to MUX1 and MUX2 labs. Reason: software not supported under other OS.

Computer system, remote access

- Remote access to general linux machines
  - Require 2-step verification (additional step using app on phone)
  - Use thinlinc protocol software (runs on windows, mac, linux)
  - Use rdp protocol software (choose one linux machine)
  - Use ssh/X11 protocol software and connect to a linux machine
- Graphic interface necessary (X11 protocol)
  - Linux: builtin support
  - Windows: mobaxterm
  - Mac: xquartz

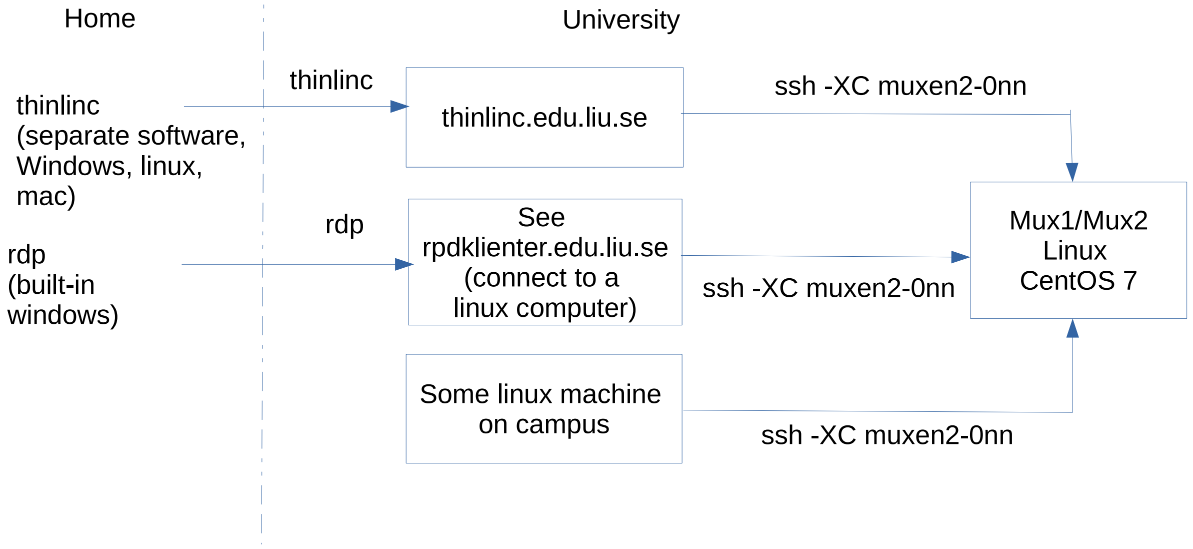
Computer system in MUX lab

- Special computer setup in MUX1 and MUX2
  - CentOS 7 (linux variant)
  - Hardware and software different from other labs, including libreoffice and other software
- Possible to remote login from thinlinc.edu.liu.se
  - Use ssh -XC muxen2-0nn in a terminal window from a linux machine (nn is 01-16)
  - Check if someone already logged in on the computer

```
w
```

- Machines always reboots at night

Summary of remote access to MUX



Home

University

thinlinc (separate software, Windows, linux, mac)

rdp (built-in windows)

thinlinc

rdp

thinlinc.edu.liu.se

See rpdklienter.edu.liu.se (connect to a linux computer)

Some linux machine on campus

ssh -XC muxen2-0nn

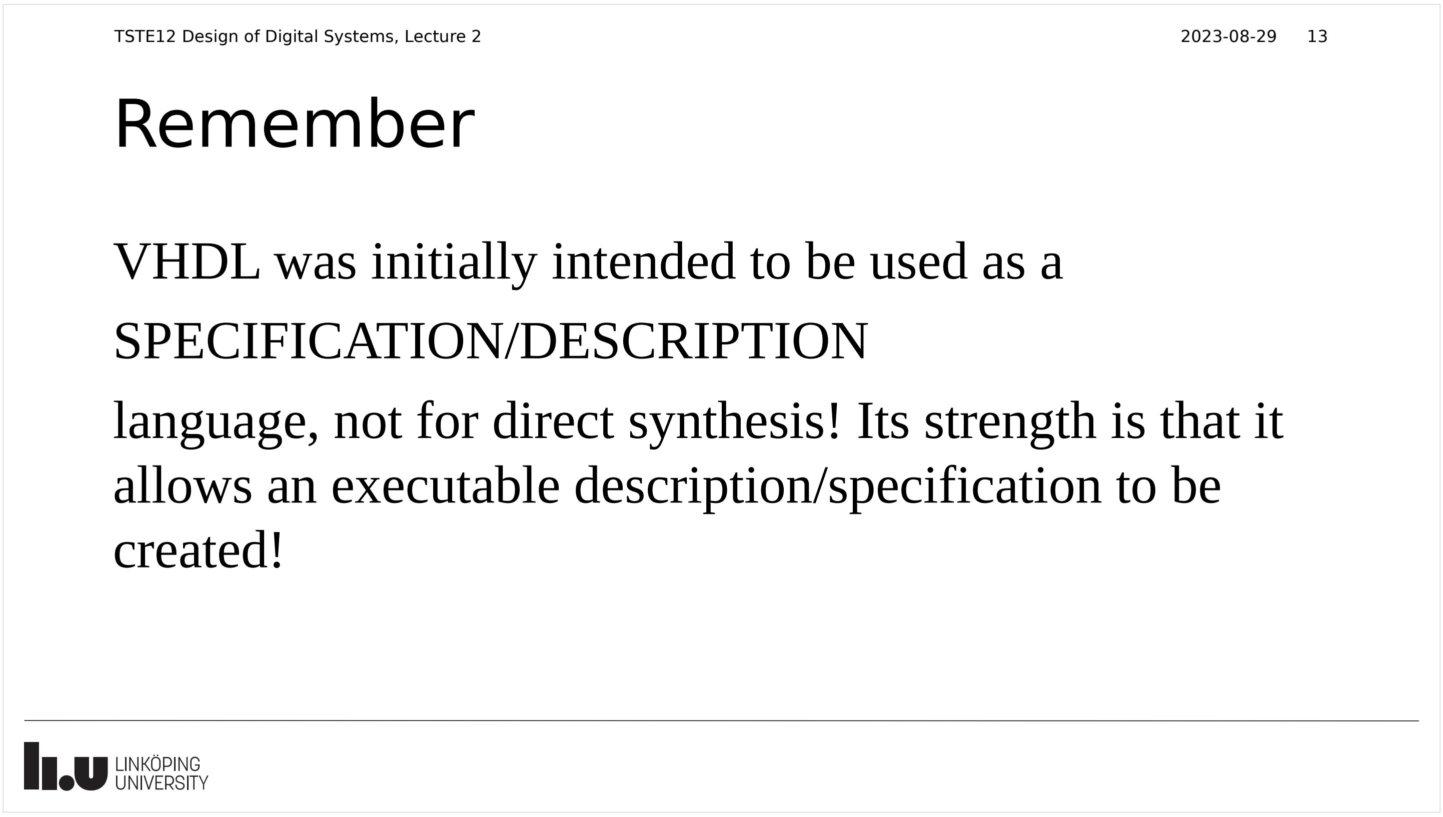
ssh -XC muxen2-0nn

ssh -XC muxen2-0nn

Mux1/Mux2 Linux CentOS 7

VHDL Introduction

- VHSIC Hardware Description Language
  - Very High Speed Integrated Circuits
- Developed on contract from US Dept. of Defense
- First standard created 1987. Major revisions 1993, 2008 and 2019 (minor revision in 2000/2002, 2008, and 2019).
  - Will cover 87 version first
- Additional standards has been adopted (e.g. std-logic data types, math library etc.)
- Most popular in Europe

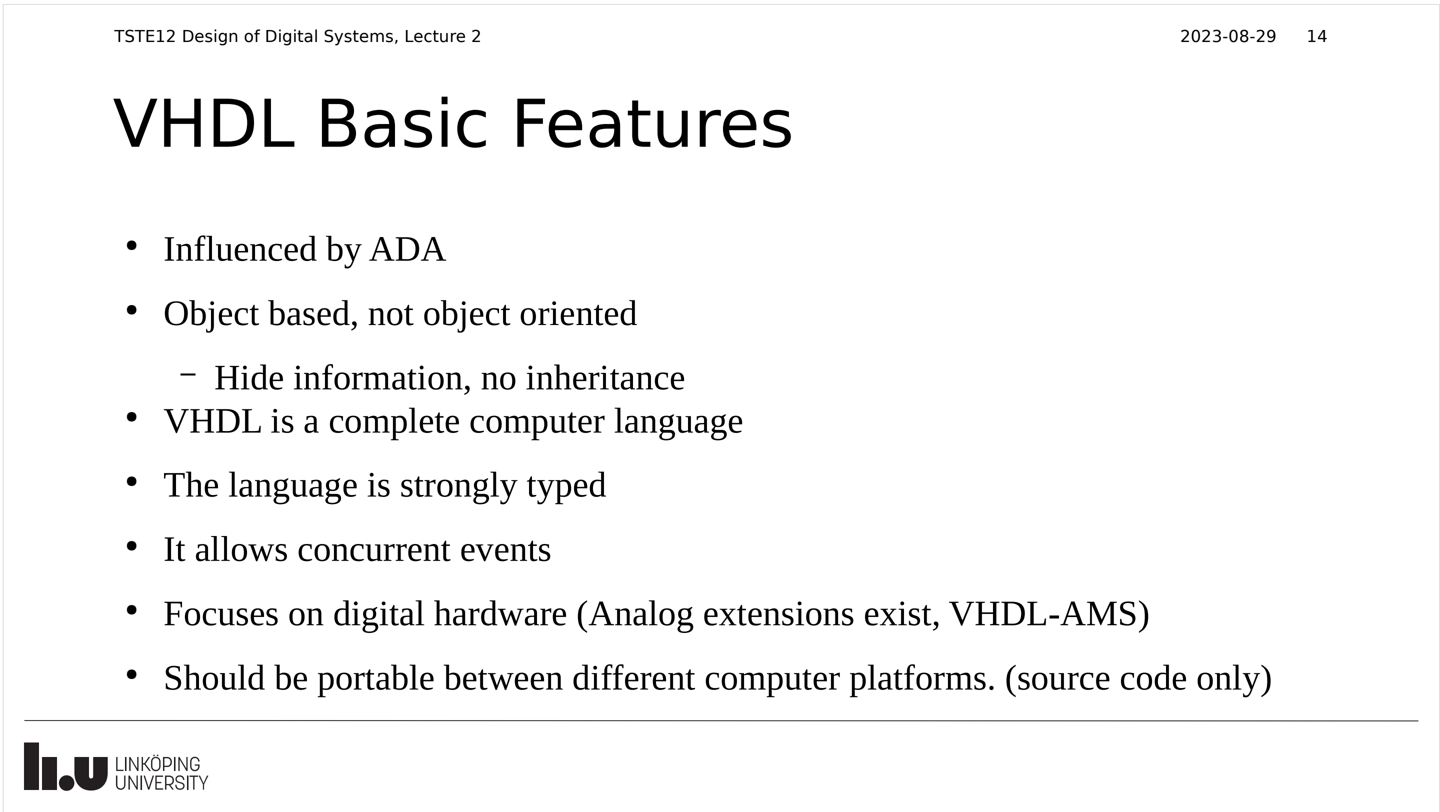


Remember

VHDL was initially intended to be used as a

SPECIFICATION/DESCRIPTION

language, not for direct synthesis! Its strength is that it allows an executable description/specification to be created!



VHDL Basic Features

- Influenced by ADA
- Object based, not object oriented
  - Hide information, no inheritance
- VHDL is a complete computer language
- The language is strongly typed
- It allows concurrent events
- Focuses on digital hardware (Analog extensions exist, VHDL-AMS)
- Should be portable between different computer platforms. (source code only)

VHDL Comments

- Remember to add comments

```
-- comments starts with double dashes
-- each comment continues to the end of line
```

- Use comments to document your design
- Special form know as Pragmas
  - Control simulation and synthesis tools
  - Vendor depend (no defined standard)
  - Example: `--pragma translate_off`

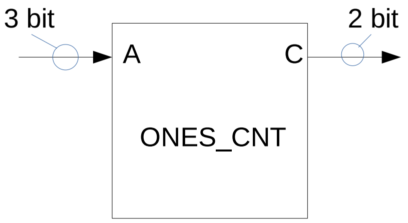
VHDL Basic Building Block

- The common building block is called an entity
- Design entities consists of two parts
  - Entity definition describing the interface
  - Architecture describing internals
    - possible to have multiple architectures for a single entity definition
    - Internals not accessible from the outside
- Common to divide these two parts into separate files
- Hierarchy allows reuse of entities and hiding of detail

VHDL Entity definition

- Example

```
entity ONES_CNT is
port (A : in BIT_VECTOR(2 downto 0);
      C : out BIT_VECTOR(1 downto 0));
end ONES_CNT;
```



- Fully specified interface
  - Datatype
  - Direction
  - Names

NO information about how it works!

NO information about how it is implemented!

VHDL Architecture dataflow example

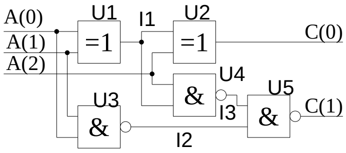
- Architecture describes internal function of an entity
- Here, boolean equations are used to describe the expected behaviour
- Both equations are evaluated at the same time
- Note parenthesis around expressions!
  - AND and OR have equal precedence

Name of corresponding entity

```
architecture DATA_FLOW of ONES_CNT is
  begin
    C(1) <= (A(1) and A(0)) or (A(2) and A(0))
         or (A(2) and A(1));
    C(0) <= (A(2) and not A(1) and not A(0))
         or (not A(2) and not A(1) and A(0))
         or (A(2) and A(1) and A(0))
         or (not A(2) and A(1) and not A(0));
end DATA_FLOW;
```

VHDL Architecture structure example

- Build function using other building blocks (entities)
- Reuse of basic gates
- Hierarchy
  - First declare components (interfaces)
  - Then instantiate them (connect them together)

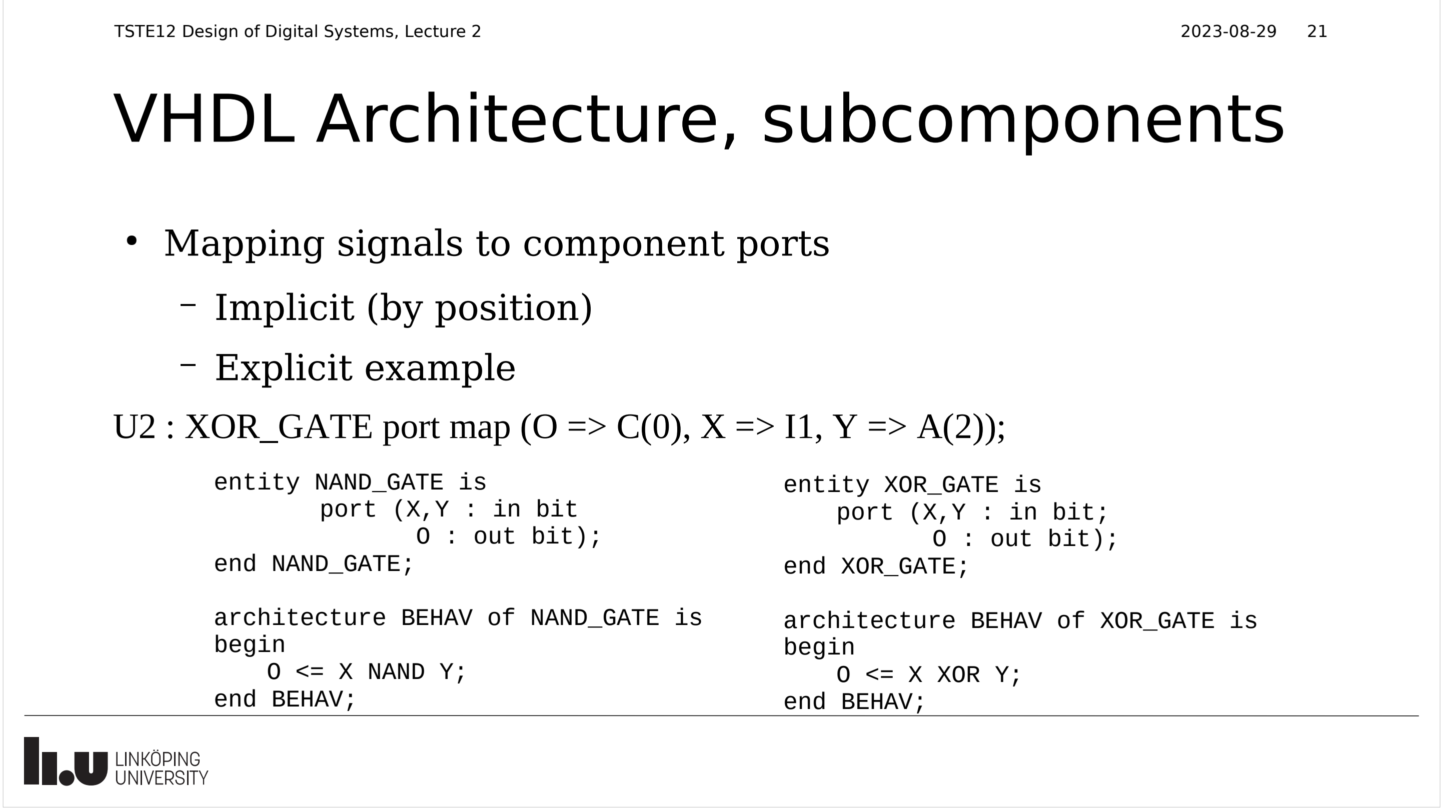


```
architecture STRUCTURAL of ONES_COUNT is

    component XOR_GATE
        port (X,Y : in bit; O : out bit);
    end component;

    component NAND_GATE
        port (X,Y : in bit; O : out bit);
    end component

    signal I1, I2, I3 : bit;

begin

    U1 : XOR_GATE port map(A(0),A(1),I1);
    U2 : XOR_GATE port map(I1,A(2),C(0));
    U3 : NAND_GATE port map(A(0),A(1),I2);
    U4 : NAND_GATE port map(A(2),I1,I3);
    U5 : NAND_GATE port map(I2,I3,C(1));

end STRUCTURAL;
```

VHDL Architecture declaration

- Describes whats inside the entity

```
    architecture ARCHITECTURE_NAME of ENTITY_NAME is
     -- Architecture declaration section
     -- types, subtypes, constants, subprograms, components
     -- Signals declared here (NO variables)
  begin
     -- concurrent statements
end ARCHITECTURE_NAME;
```



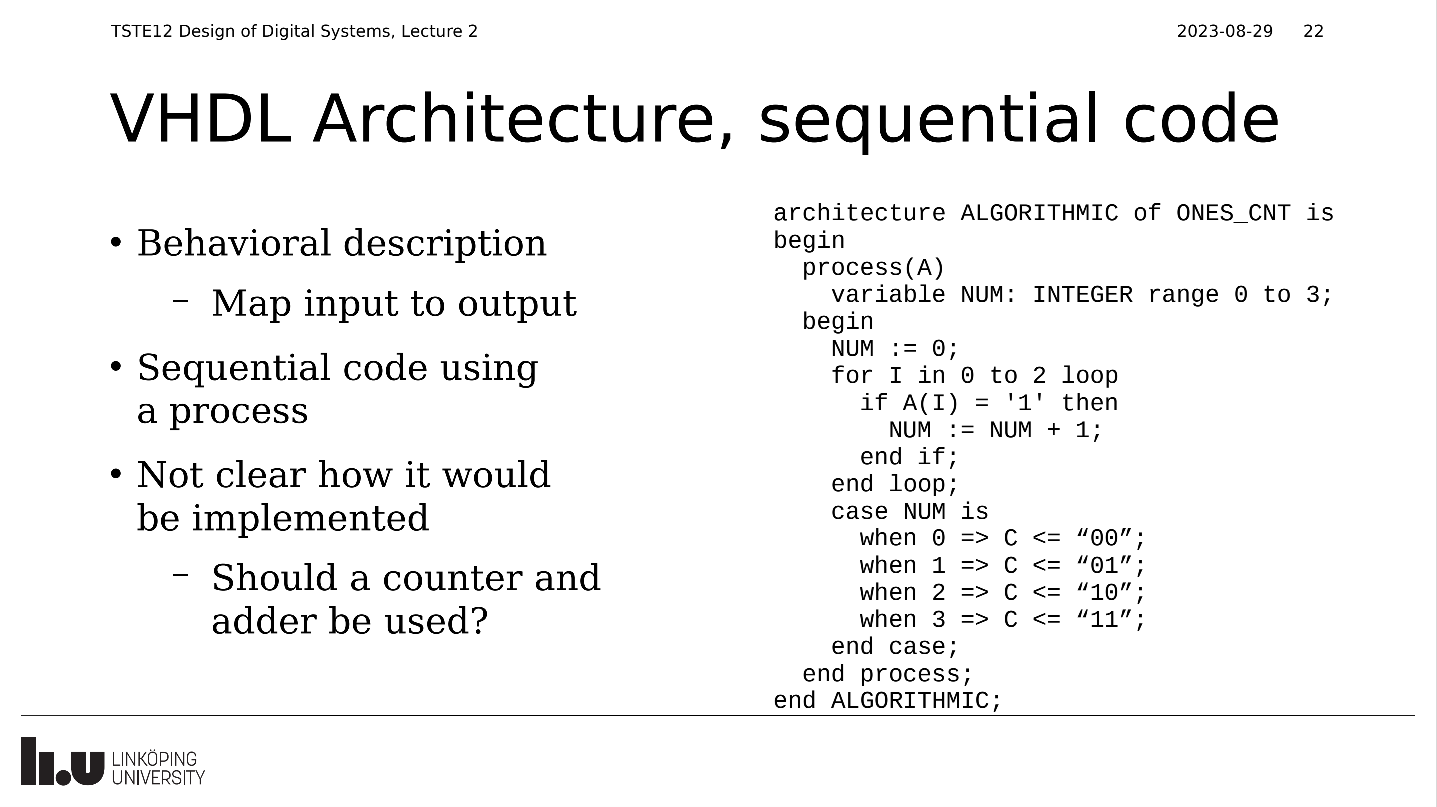
VHDL Architecture, subcomponents

- Mapping signals to component ports
  - Implicit (by position)
  - Explicit example

U2 : XOR_GATE port map (O => C(0), X => I1, Y => A(2));

```
entity NAND_GATE is
        port (X,Y : in bit
               O : out bit);
end NAND_GATE;

architecture BEHAV of NAND_GATE is
begin
    O <= X NAND Y;
end BEHAV;
```

```
entity XOR_GATE is
    port (X,Y : in bit;
           O : out bit);
end XOR_GATE;

architecture BEHAV of XOR_GATE is
begin
    O <= X XOR Y;
end BEHAV;
```



VHDL Architecture, sequential code

- Behavioral description
  - Map input to output
- Sequential code using a process
- Not clear how it would be implemented
  - Should a counter and adder be used?

```
architecture ALGORITHMIC of ONES_CNT is
begin
  process(A)
    variable NUM: INTEGER range 0 to 3;
  begin
    NUM := 0;
    for I in 0 to 2 loop
      if A(I) = '1' then
        NUM := NUM + 1;
      end if;
    end loop;
    case NUM is
      when 0 => C <= “00”;
      when 1 => C <= “01”;
      when 2 => C <= “10”;
      when 3 => C <= “11”;
    end case;
  end process;
end ALGORITHMIC;
```

Examples so far

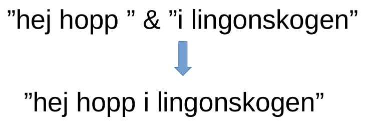
- Three types of examples so far
  - Basic logic gates (boolean equations)
  - Flipflop (very simple small process) previous lecture
  - Interconnect gates and flipflops using structure
- Should now be able to create small designs
  - Still manual steps (create Karnoughmaps, state graphs)
  - Not using the power of the language and synthesis tools

VHDL Basics

- Character set: 7-bit ASCII (extended in 1993)
  - Avoid strange character (e.g. å, ä, ö etc.)
- Terminate statements with ;
  - It is not separating statements, it is ending a statement
- Identifiers (names)
  - Start with a letter
  - Include only letters, digits and isolated _
  - Last character must be a digit or a letter
  - No case sensitivity

VHDL characters and strings

- Character literals
  - One character between two apostrophe delimiter
    - Example: ’a’, ’ ’
- String literal
  - Characters between ”
  - Length equal to number of characters
  - Not possible to span multiple lines
    - Use concatenation using & operator instead



VHDL Bit strings

- Special case of strings
  - Base specifier can be used
    - B (binary), O (octal), X (hexadecimal)
  - Examples (different values)
    - B”101101101”, ”11101011101”, X”DE”
- Viewed as string of bits, has no associated value
  - Example: X”C” is viewed as ”1100”

VHDL Numeric Values

- Integers
  - Does not have a base point
  - Examples: 5, 27, 23E5
- Real
  - Has always a base point
  - Examples: 5.0, 0.0, 2.3E-5
- Based literals
  - Has a base specification [2, 16]
  - Examples: 16#FfF#, 4#3.33333#e5

VHDL Data Types

- Strictly enforced data types (strongly typed language)
  - No automatic translation between types
  - Not allowed to mix datatypes in expressions
  - Helps avoid programming errors
  - Can create your own data types
- Subtypes
  - Type plus constraint
  - Limit the set of allowed values
  - Example: Natural is a subtype of integers

VHDL Data Types, cont.

- Scalar Data types
  - Simple, single values
  - Enumerations, integers, physical, real
- Composite
  - Array (includes vectors)
  - Record
- Access
  - Pointers
- File

Predefined Data Types

- Defined enumerations

```
type boolean is (FALSE, TRUE);
type bit is ('0', '1');
type character is (
  NUL,SOH,STX,ETX,EOT,ENQ,ACK,BEL,BS,HT,LF,VT,FF,CR,SO,SI,
  DLE,DC1,DC2,DC3,DC4,NAK,SYN,ETB,CAN,EM,SUB,ESC,FSP,GSP,RSP,USP,
  ' ','!','″','#','$','%','&',''','(',')','*','+',',','-','.','/',
  '0','1','2','3','4','5','6','7','8','9',':',';','<','=','>','?',
  '@','A','B','C','D','E','F','G','H','I','J','K','L','M','N','O',
  'P','Q','R','S','T','U','V','W','X','Y','Z','[','\',']','^','_',
  '`','a','b','c','d','e','f','g','h','i','j','k','l','m','n','o',
  'p','q','r','s','t','u','v','w','x','y','z','{','|','}','~',DEL);
type severity_level is (NOTE, WARNING, ERROR, FAILURE);
```

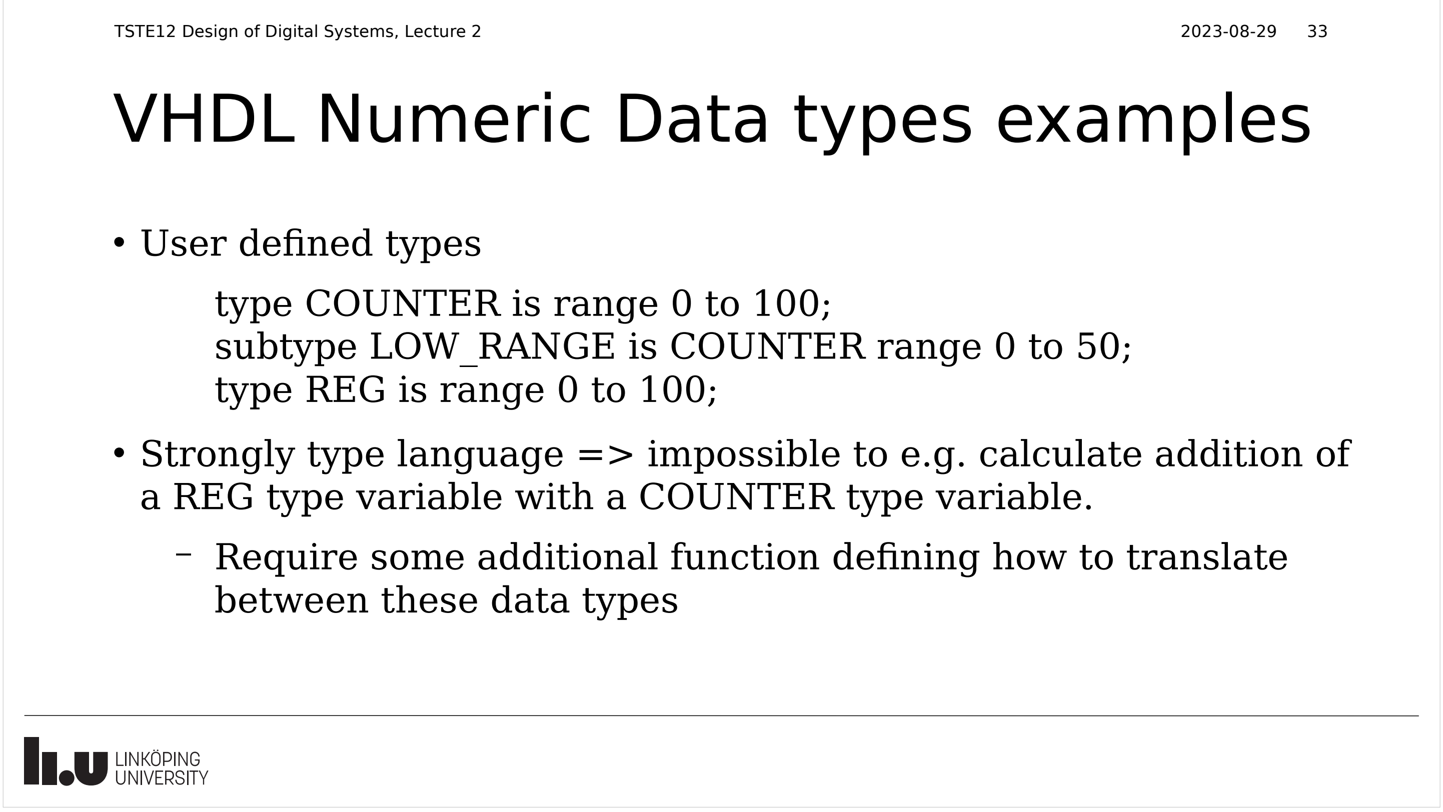
- Predefined operations (and, or, etc.) exist for bit and boolean

VHDL Enumerations and Attributes

- Example: Type COLOR is (red, orange, green);
- Each element has a position (0 to the left, integer increment)
  - Initial values of enumeration (if not specified) is always 'left value
    - Example: Bit variables and signals default to '0'
- Attributes can be give properties of a type or variable
  - Based on the position of the element in the enumeration
  - 'pos, 'val, 'left, 'right, 'high, 'low, 'succ, 'pred
    - Example: COLOR'pos(GREEN) = 2
  - Possible to create user-defined attributes

Numeric Data types

- Integers
  - Range is implementation dependent
    - Minimum 32 bits (-2147483647 to 21483647)
    - Subtypes usually used to catch errors and help synthesis
- Real
  - Range is implementation dependent (at least 32 bits)
  - This range is to small for many simulation purposes (e.g. communication systems)

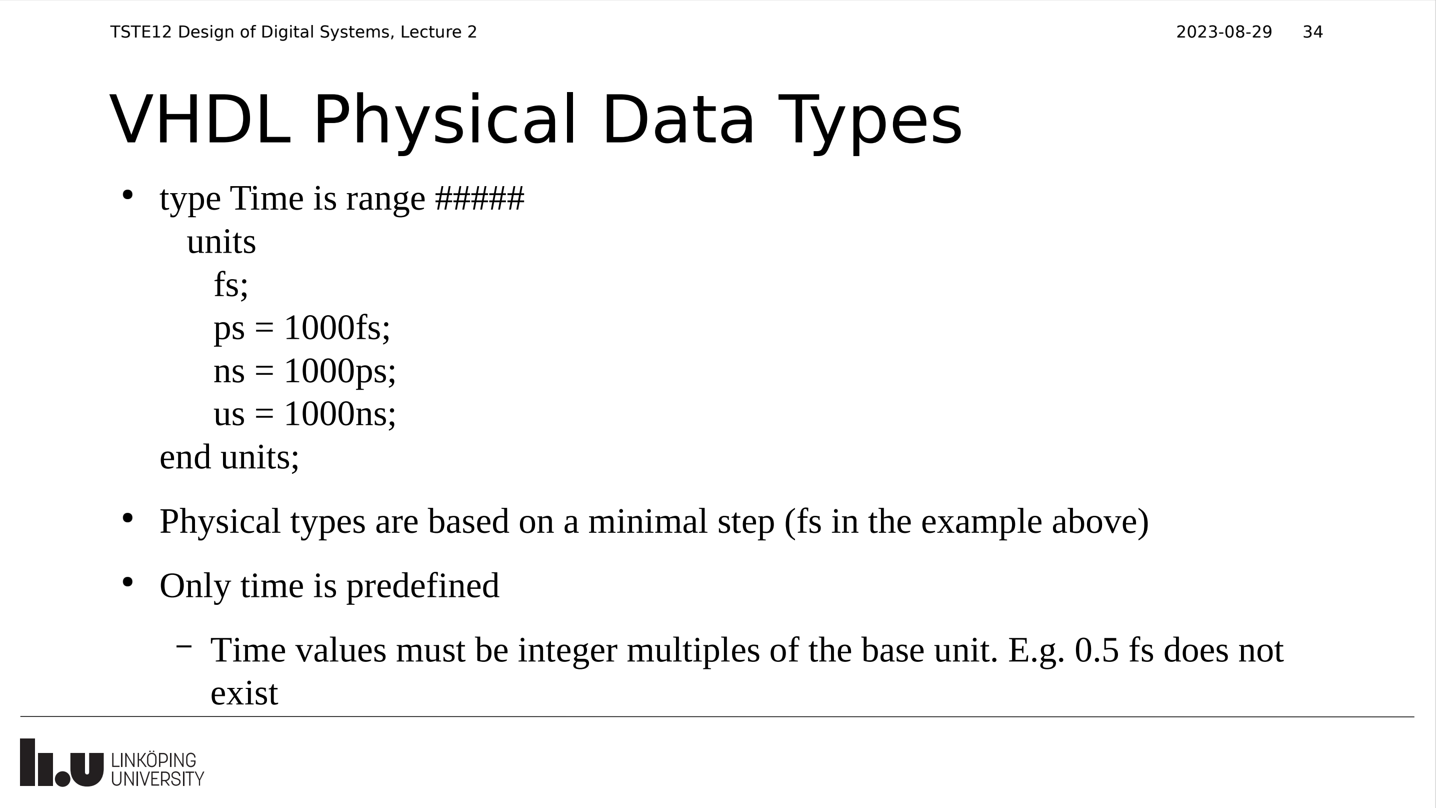


VHDL Numeric Data types examples

- User defined types

```
type COUNTER is range 0 to 100;
subtype LOW_RANGE is COUNTER range 0 to 50;
type REG is range 0 to 100;
```

- Strongly type language => impossible to e.g. calculate addition of a REG type variable with a COUNTER type variable.
  - Require some additional function defining how to translate between these data types



VHDL Physical Data Types

-
  ```
  type Time is range #####
    units
      fs;
      ps = 1000fs;
      ns = 1000ps;
      us = 1000ns;
  end units;
  ```
- Physical types are based on a minimal step (fs in the example above)
- Only time is predefined
  - Time values must be integer multiples of the base unit. E.g. 0.5 fs does not exist

VHDL Composite Data Types

- Arrays (predefined)

```
Type String is array (positive range <>) of Characters;
Type bit_vector is array (natural range <>) of bit;
```

  - <> means unconstrained range (not specified yet)

- More complex version
  Type ROM_TYPE is array (natural range <>) of bit_vector(31 downto 0);
- Array attributes
  - ’right, ’left, ’low, ’high, ’length
- Allows for generic subroutines and designs without hardcoded dimensions

VHDL Data Types

- Record: combines elements of different types

```
Type date is record
   Day : integer range 1 to 31;
   Month : month_name;
   Year : integer range 0 to 3000;
End record;
```

- Access: dynamic storage (linked lists, tress etc.)
  - Not covered in this course
  - Only for simulation, no possible direct translation to hardware

VHDL Type Marks

- Overloading of values
  - Same symbol is used in multiple types
    - Example: ’1’ is available both in Bit and as a character
    - Use type marking to remove ambiguity
      - Helps tools to understand the type of the value
    - Example: bit’(’1’)

VHDL Data Objects

- Constants
  - Specified at compile time, never change value during simulation
  - Both value and type must be specified
- Variables
  - Current value can be changed, used in sequential code
- Signals
  - Objects with time dimension. Assignments does not affect the current value, so current value can not be changed

VHDL Signals and Variables

- Declared in different places
  - Signals: ports on entitys, in architecture declaration
  - Variables: in processes and subprograms (functions and procedures)
- Both start with the leftmost value specified otherwise.
- Examples
  - Variable REG1: BIT_vector(15 downto 0) := X”F5A2”;
  - Signal Value: bit_vector(5 to 7) := ”011”;

VHDL intial value, advanced version

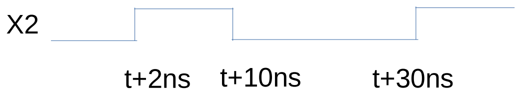
- Example

```
Variable ROM_A : ROM_TYPE(0 to 7) :=
                    (       0 => X”FFFF_FFFF”,
                             5=> X”2222_CCCC”,
                      others=> X”0000_0000”);
```

- Example shows matrix initialization with an aggregate
  - All rows in the ROM_A has value ”00000000” except 0 and 5
  - Efficient way to enter large number of values to vector/array elements.
- Example: Set a bit_vector to all 0: REG1 <= (others => ’0’);

VHDL Assignments

- Variable assignment uses :=
  - Example: A := 5;
- Signal assignment uses <=
  - Assign a new value sometime in the future (never change current value)
  - Examples:
    - X1 <= ’1’ after 10 ns;
    - X2 <= ’1’ after 2 ns,
      ’0’ after 10 ns,
      ’1’ after 30 ns;



VHDL Signal Attributes

'active – transaction in current simulation cycle (update, may be same value as previous value)

'event – event in current simulation cycle (new value, different from previous value)

- Commonly used to detect clock edges (see flipflop model)

'stable(tval) – no events (last tval time units)

'quit(tval) – no transaction for tval time units

'last_active – how long time since last change

'delayed – value of signal delayed

VHDL Operators

** abs not	(highest precedence)
* / mod rem	
+ -	(signing)
+ - &	
= /= < <= > >=	
and or nand nor xor	(lowest precedence)

=	Equal
/=	Not equal to
>	Greater then
>=	Greater than or equal
<	Less than
<=	Less than or equal

- Equal precedence
  - A OR B AND C ≠ A OR (B AND C)
- All associative except nand, nor
  - (X1 nand X2) nand X3 ≠ X1 nand (X2 nand X3) ≠ not (X1 and X2 and X3)

VHDL Operators, cont.

- Comparisons and +, -, & must have same base type for both objects
  - No type casting is done
- Operator & is concatenating one-dimensional arrays
- Mod and rem only works with integers
- Physical data can be multiplied by real or integer
  - E.g. double a delay by time*2
- ** is the exponential operator, abs is the absolute value operator
- Logic not operator only works on bit and boolean

Sequential vs Concurrent code

- Sequential code is the common programmers view on programs
  - Single point of control, executing one statement after another
- Concurrent code
  - All statements computed at the same time
  - No way to know in which order a sequential computer executes statements
- The architecture body contains only concurrent code
- The process, functions and procedures contains only sequential code

Concurrent assignment

```
LABEL:  SIGNAL_NAME <= [transport]
   WAVEFORM1 when CONDITION1 else
   WAVEFORM2 when CONDITION2 else
         :
   WAVEFORMn when CONDITIONn else
   WAVEFORMq;
```

  - Can also be described by sequential signal assignment in a process
  - Example: C <= A or B;
  - Transport will be discussed later

Concurrent signal assignment: Select

```
LABEL: with EXPRESSION select
SIGNAL_NAME <= [transport]
  WAVEFORM1 when CHOICES1,
  WAVEFORM2 when CHOICES2,
      :
  WAVEFORMn when CHOICESn,
  WAVEFORMq when others;
```

- All possible result of expression must must be included exactly ones in a choice.
- Choices may be a list (e.g., when 0|1|2,)

Assert statement

- Only way to get a text message to the user of the code
  - Used mainly for error detection

```
Assert Boolean_EXPRESSION
Report “Message_string”
Severity SEVERITY_LEVEL
```

- If expression is false then report. Severity levels are note, warning, error, failure
  - Simulation may stop depending on settings
- Concurrent version allows a label in front

```
LABEL: assert Boolean_EXPRESSION
```

Process statement

```
[LABEL:] process (SENSITIVITY_SIGNAL_LIST)
     -- constants, types, subtypes, subprograms
     -- variables (NO signals)
  begin
     -- sequential statements
  end process;
```

- Process always executed once at simulation start. Then whenever events occur on signals in the sensitivity list
- Variables are static
  - Initialized at simulation start
  - Keep their values between process activations

Process statements, cont.

- Can combine process statements with structural code in the same architecture
- Processes without a sensitivity list will automatically restart at the end of a process
  - Must have some way to stop simulation/wait some time to avoid an infinite loop (simulation appear to be hanged)
- Processes can not both have a sensitivy list and wait statements in the same process

Sequential code

- Processes and subprograms have sequential code
  - One statement after another is executed in order
  - Most similar to ”ordinary” computer code
  - Simulation time does not increase while executing the statements
    - Exception is the wait statement

Sequential control statements: wait

- Used in processes and subprograms
- Examples

```
wait on x,y until z=0 for 100 ns;
    -- wait until event on x or y while z≠0 or max 100 ns)
wait for 100 ns;
wait on a,b,c;  -- wait for at least one event on a, b or c
wait until z=0;
wait;   -- infinite wait
```

Sequential control: if

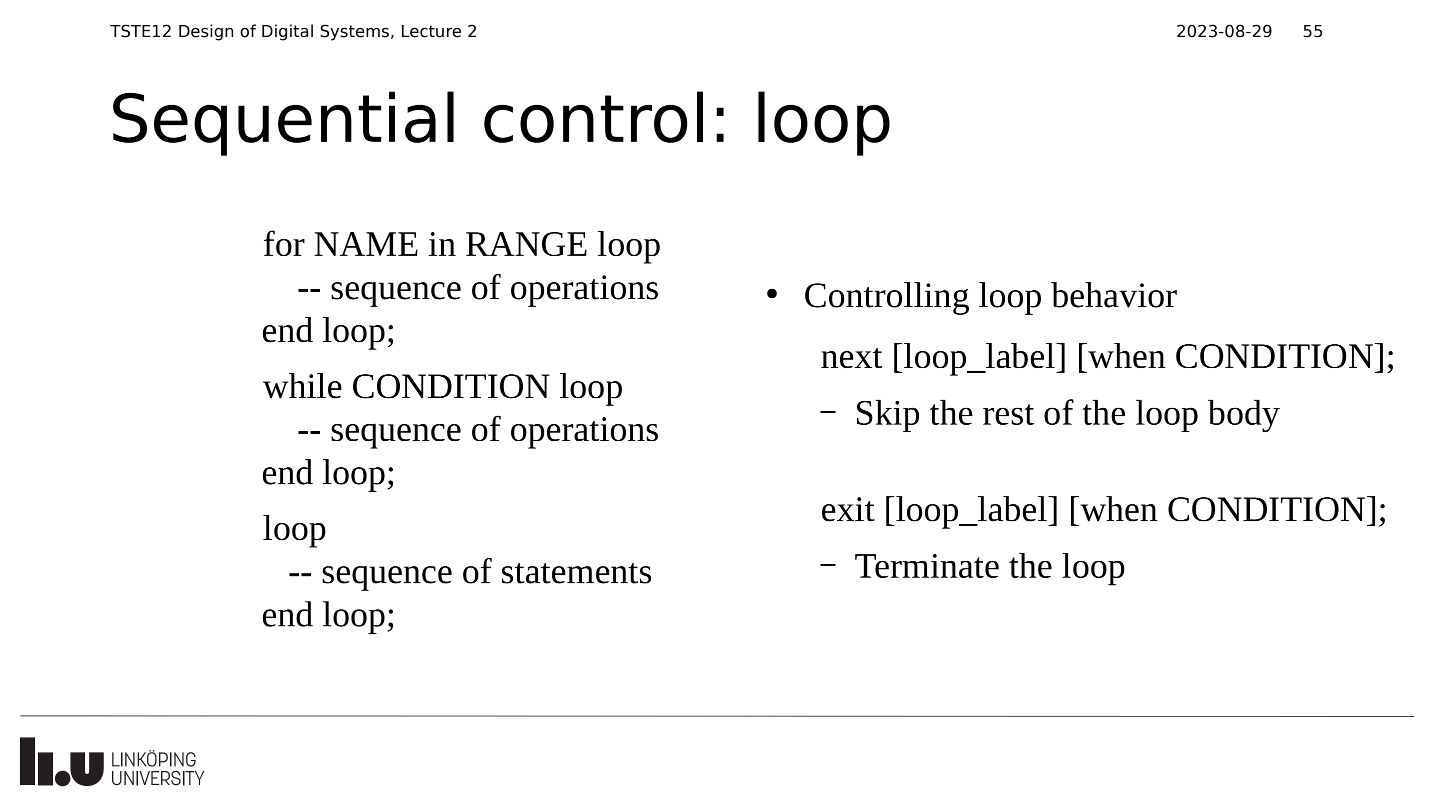
```
if CONDITION1 then
    -- sequence of statements 1
elsif CONDITION2 then
    -- sequence of statements 2
    -- any number of elsif clauses
else
    -- last sequence of statements
end if;
```

- Indentation not important (not like python)
- CONDITION must return boolean (not enough with a bit)

Sequential control: case

```
case EXPRESSION is
    when CHOICE1 => -- sequence of statements1
    when CHOICE2 => -- sequence of statements 2
    when others => -- last sequence of statements
end case;
```

- All possible choices must be covered once
  - Others catch all choices not covered earlier
- Choices may be a list (e.g., when 0|1|2 =>)



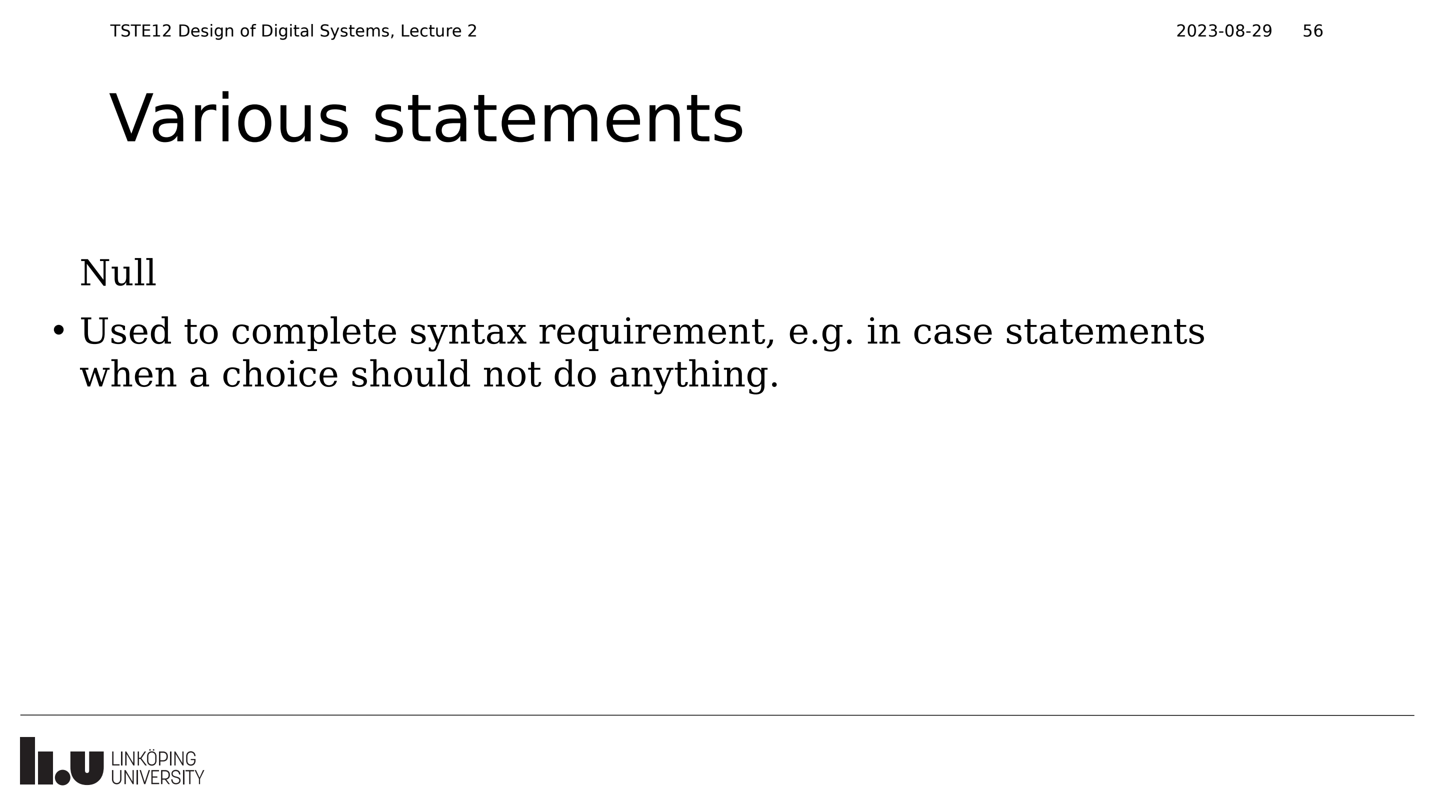
Sequential control: loop

```
for NAME in RANGE loop
   -- sequence of operations
end loop;
while CONDITION loop
   -- sequence of operations
end loop;
loop
  -- sequence of statements
end loop;
```

- Controlling loop behavior

  next [loop_label] [when CONDITION];
  - Skip the rest of the loop body

  exit [loop_label] [when CONDITION];
  - Terminate the loop



Various statements

Null

- Used to complete syntax requirement, e.g. in case statements when a choice should not do anything.

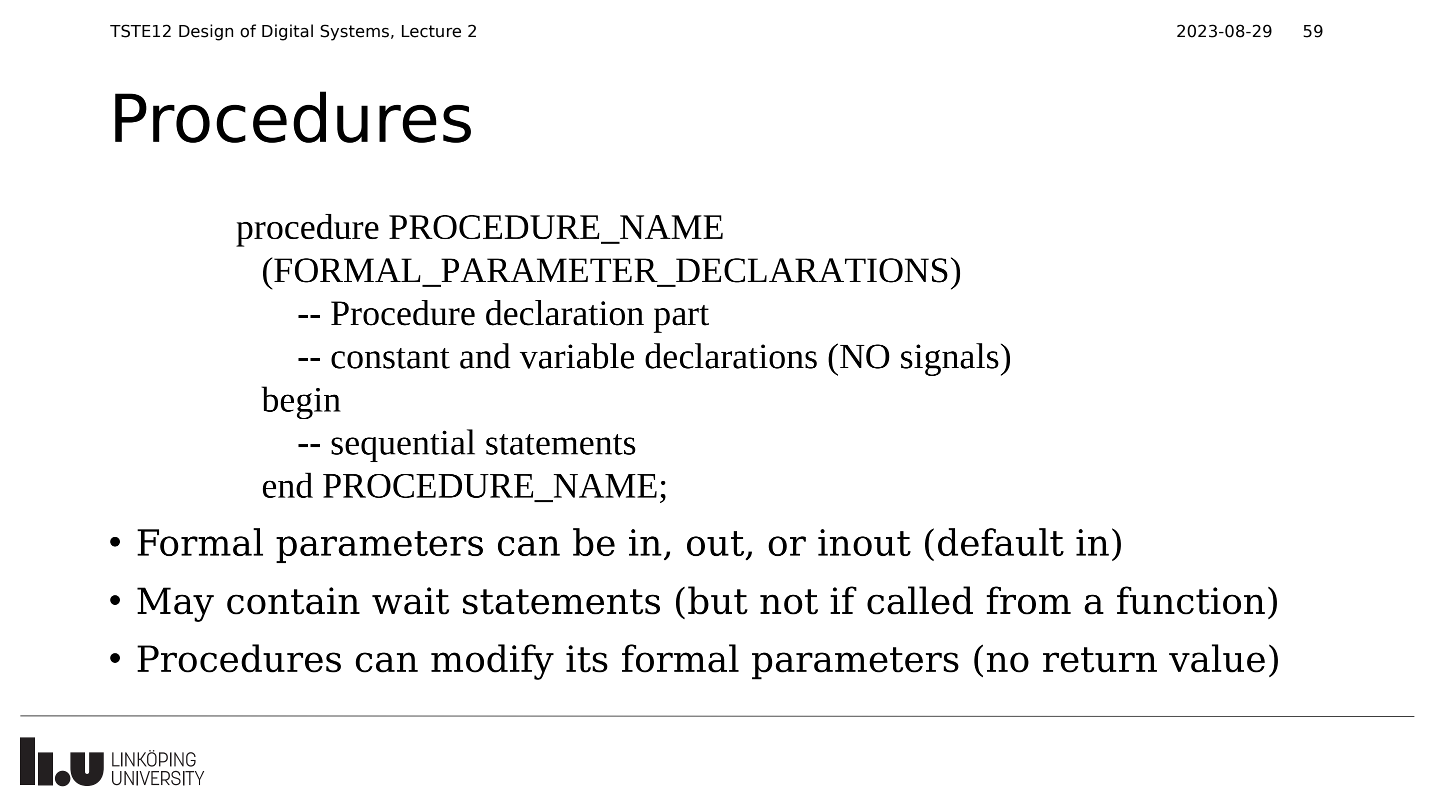
Subprograms

- Functions and procedures
- Declared in declaration region of architecture, process, block, or other subprograms.
- Variables are dynamic (initialized at every call)
- Functions
  - Always returns a value (must be used in an expression)
  - Never modifies its parameters (all parameters are inputs)
  - No side effects allowed
  - Can not contain wait statements

Functions

```
function FUNCTION_NAME (FORMAL_PARAMETER_DECLARATIONS)
   return RETURN_TYPE is
      -- constant and variable declarations (NO signals)
   begin
      -- sequential statements
      return (RETURN_VALUE);
   end FUNCTION_NAME;
```

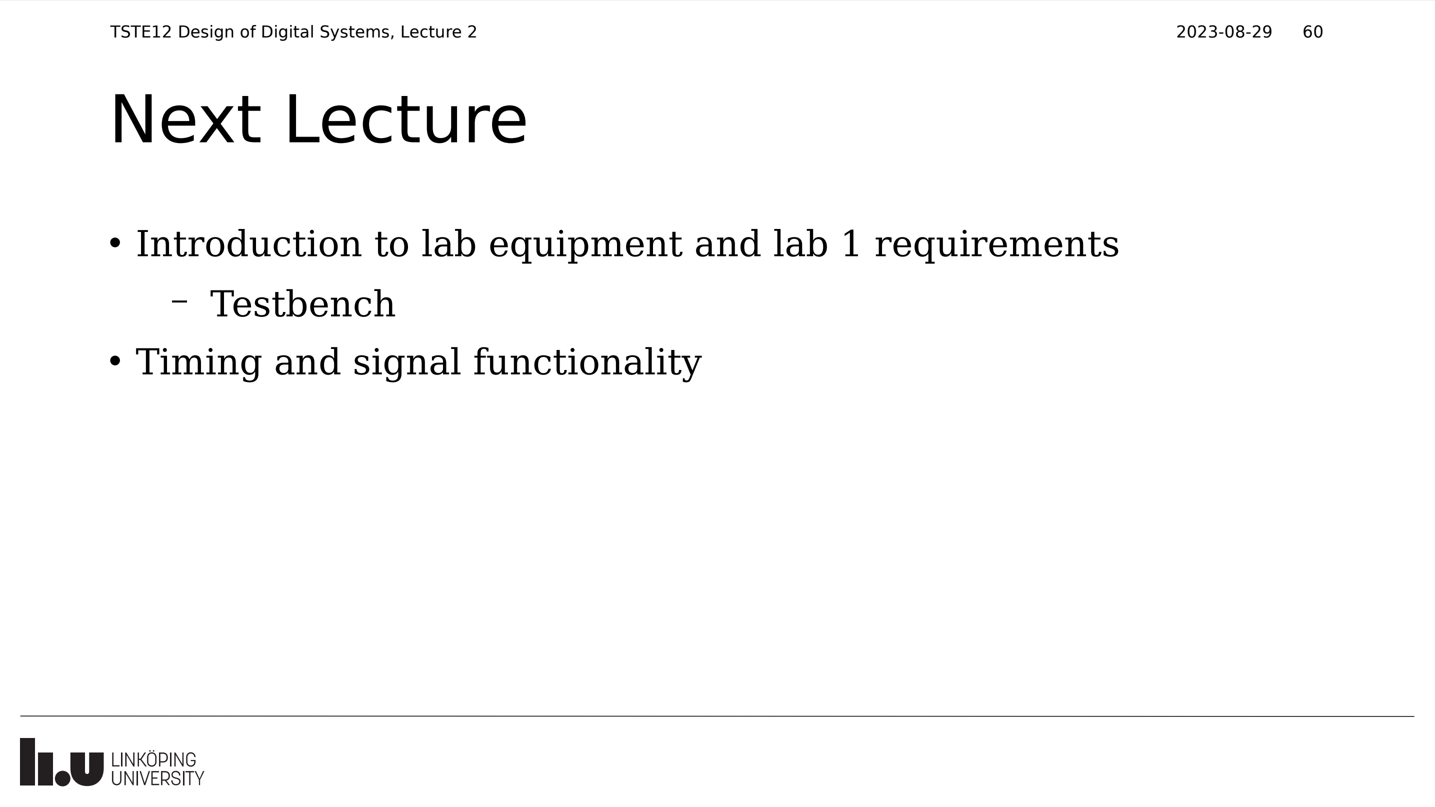
- Must always return a defined value



Procedures

```
procedure PROCEDURE_NAME
  (FORMAL_PARAMETER_DECLARATIONS)
     -- Procedure declaration part
     -- constant and variable declarations (NO signals)
  begin
     -- sequential statements
  end PROCEDURE_NAME;
```

- Formal parameters can be in, out, or inout (default in)
- May contain wait statements (but not if called from a function)
- Procedures can modify its formal parameters (no return value)



Next Lecture

- Introduction to lab equipment and lab 1 requirements
  - Testbench
- Timing and signal functionality

LINKÖPING
UNIVERSITY