TSTE12 Design of Digital Systems Lecture 3

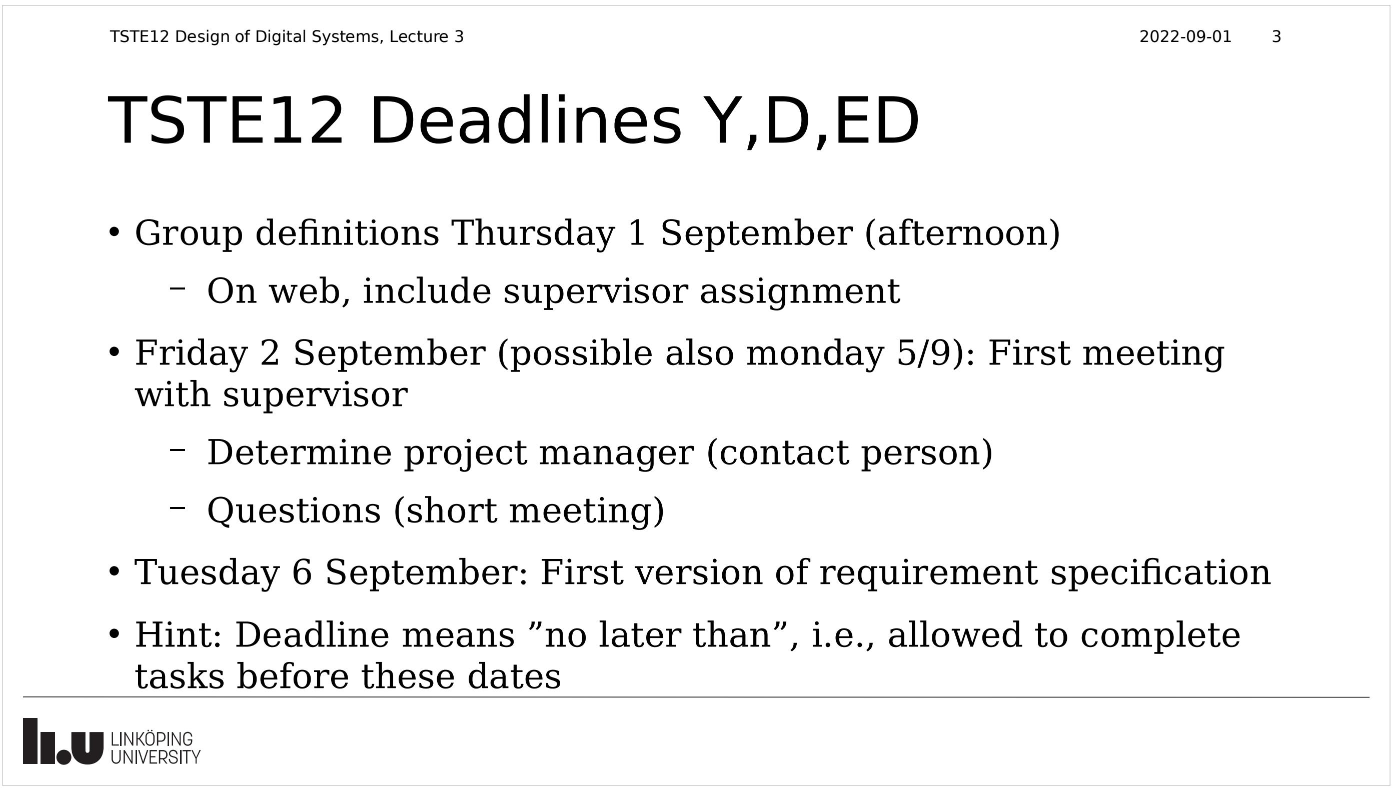
Kent Palmkvist

LINKÖPING UNIVERSITY

Agenda

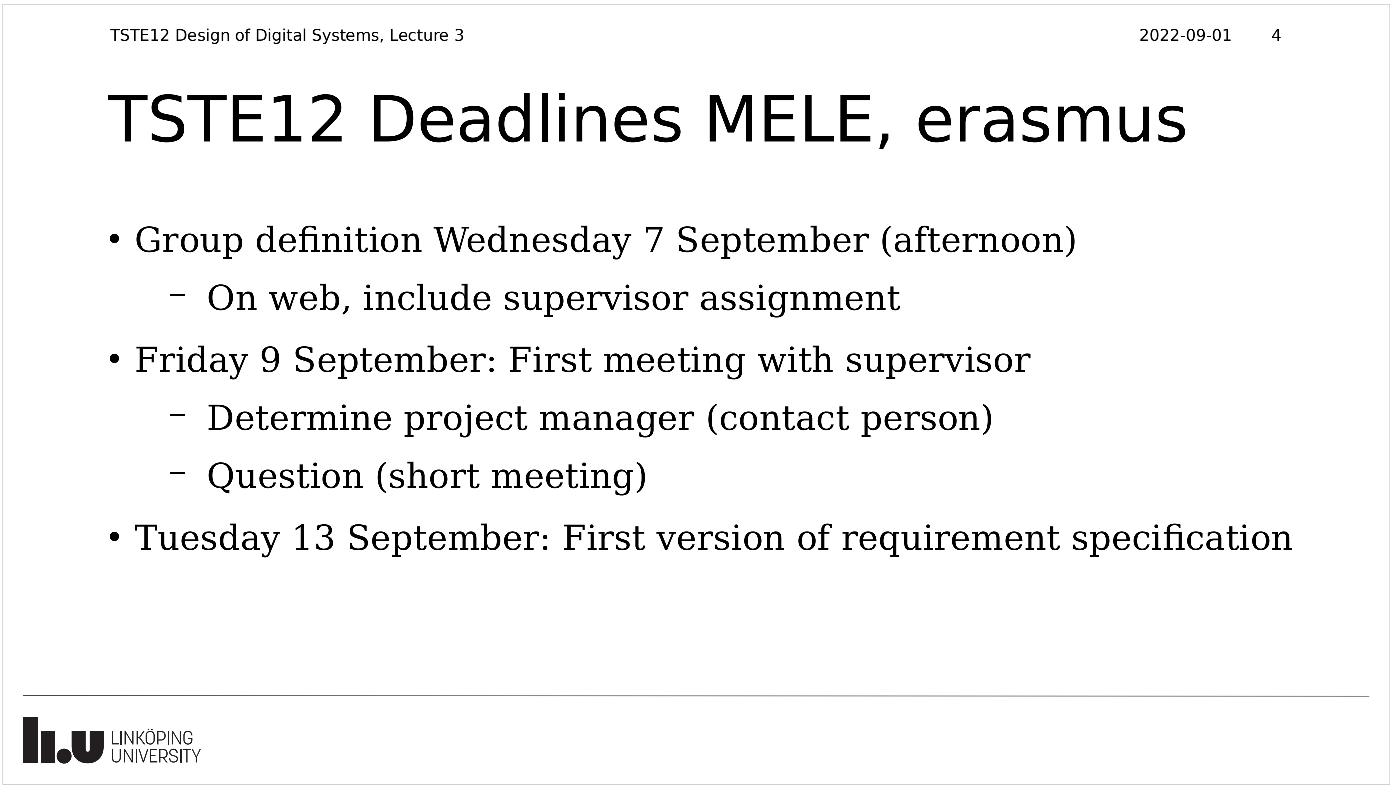
- Practical issues
  - Material
  - Lab 1 requirements
- Project task intro
- Introduction to VHDL, continued
  - Timing
  - Testbench

LINKÖPING UNIVERSITY



TSTE12 Deadlines Y,D,ED

- Group definitions Thursday 1 September (afternoon)
  - On web, include supervisor assignment
- Friday 2 September (possible also monday 5/9): First meeting with supervisor
  - Determine project manager (contact person)
  - Questions (short meeting)
- Tuesday 6 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 7 September (afternoon)
  - On web, include supervisor assignment
- Friday 9 September: First meeting with supervisor
  - Determine project manager (contact person)
  - Question (short meeting)
- Tuesday 13 September: First version of requirement specification

Shared folder problem

- Currently a configuration problem with the use of shared lab folders
- Temporary solution: Use initially a folder in your home folder
  - Drawback: Limit the access of the file to the student who logged in
  - Not a big problem if you work on the tutorial (the files are not needed later on)

Useful resources

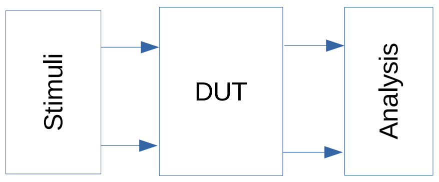
- Lab hardware description (on the linux machines)
  - /courses/TSTE12/material/DE2-115_SystemCD_v3.0.6
  - Lab board datasheets, schematic of hardware

Lab 1 requirements

- Working design (both simulation and hardware)
- Only use sys_clk as (the only) clock signal
  - This is the 100 MHz sys_clk on the board
- Testbench testing two different key sequences
  - Should test after each key scan code input has been generated
  - A text message in the simulator window should be displayed after each scan code test telling the user if test was ok or failed.
- Hint: No need to count the number of received bits, or verify parity and stop bits. Is OK to have intermediate flickering on the display while data is received

VHDL model testingComments

- Testbench
  - Standard approach
  - Test of design will be independent of simulation tool
- Testbench contents
  - Stimuli generation
  - Design under test (DUT)
  - Analysis of design output



Programming hardware in the lab

- FPGA can be configured from any of the computers
  - Start Quartus Programmer
  - Select hardware setup
  - Select “Add server”
  - Specify machine name to which the board is connected, use the password described on the whiteboard at the door
    - muxen2-009, muxen2-015
  - Make sure the correct machine and board is defined as hardware
- Remember to check that attached computer is on

Useful linux commands

- Remember to always work from the “mentorskal” terminal window (except for handins)
- Sets the access rights correctly to support sharing of files within the groups (but not between the groups)

  ls -la file_or_directory
  shows owner and access rights of files

  chmod g+rwx filename_or_directory

  let all members of the group have full access to the file or directory (allow others to change files)

Project information

- Project directory will be available for all groups
  - /courses/TSTE12/proj/projgrpXX
  - Access only to members of the group
  - This directory must be used
- Project document templates
  - /courses/TSTE12/material/project/LIPS-templates
- Project functionality
  - Project directive (description) on the web
  - Different priority (mandatory, selective, optional)

Project directive, high priority

- Volume & balance control (atleast 10 levels each)
  - Volume is logarithmic
  - Simplifications are often possible
    - Multiplication is generally expensive
  - Amplification (factor > 1) may lead to overflow!
- A/D and D/A implemented in separate codec chip
  - Supports volume control (not necessary to feed audio into FPGA)
  - Still need to initialize the codec chip (implement I2C communication)

Optional requirement, medium priority

- Each project group make their own choice
  - No problem if more than one group selects the same project
  - Chose one or at most two medium priority requirements
- Oscilloscope
  - Vertical vs horizontal
  - Zoom, color coding, average
  - FFT
    - Complex algorithm
    - Use available designs if possible

Optional requirement, medium priority

- Echo
  - Long echo requires large memory (2MByte SRAM)
  - Number representation
    - 44 Khz, 20 bit, stereo => 220KByte/s
    - Dynamic range, Accuracy
    - 2's complement/Floating point
  - Controls?
    - On/off, Length, Strength

Optional requirement, medium priority

- Signal level indication (logarithmic scale)
  - Peak, average
  - Screen update ~ 50Hz
    - Do not want a flickery output
    - Large number of samples to average on
- Peak indication
  - move slowly towards zero
  - quickly updated when large peak found

Optional requirement, medium priority

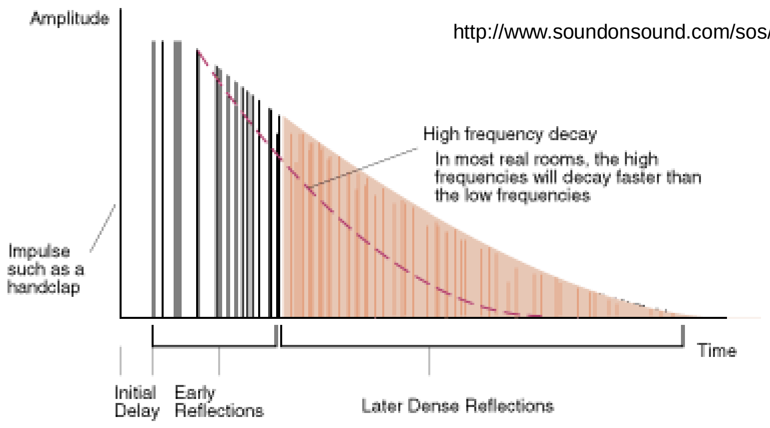
- Loudness, supression of mono
  - Loudness control => attenuate medium frequencies
- Equalizer
  - Requires understanding of digital filters
- Test filter algorithms using Matlab
- Other sound modification algorithms possible
  - Reverb, Flange
  - Pitch shifts
  - ....

2022-08-31 16:09

Big hall simulation (reverb)

- Principle: Echoes from walls and objects

http://www.soundonsound.com/sos/oct01/articles/advancedreverb1.asp



Reverb, cont.

- Three parts of the sound
  - Direct path
  - Early reflections
  - Decaying mix of reflexions
- Time in the echoes are short
  - Example: 5, 10, 20, 30, 45, 60, 75 ms versions added
  - Create attenuated feedback

Optional requirement, low priority

- Try to think of a lot of optional features
  - Color coding, background image
  - Animations, different VGA screen backgrounds
  - Special keys to use (e.g., arrow keys)
  - Change parameters at run time
  - Additional volume/balance steps
  - ....
- Try to optimize the size of the design
- Many clock cycles between input samples

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)

Libraries

- All design units are stored in libraries. Accessing a library requires definition.

  library LIBRARY_NAME;

  - How this is done in the file system is not defined in VHDL
- Libraries have only logical names
  - Simulator and synthesis uses tool-dependent mappings from logical to physical directories (possible including binary format file)
  - Allows VHDL source code to be run on different platforms without rewrite
- Two predefined libraries: WORK STD

Packages

- Used in similar way to header files or libraries in other languages
  - Repeat declarations and definitions
- Divided into the declaration and body
  - Declaration part consists of visible declarations when using the package
    - Constants, types, subprograms
  - Declaratation parts contains subprogram names and parameter list, but no code
  - Body is visible only within the package. Includes the code for the subprograms
  - Only one possible body definition

Package example

```
package HANDY is
  subtype BITVECT3 is BIT_VECTOR(0 to 2);
  subtype BITVECT2 is BIT_VECTOR(0 to 1);
  function MAJ3(X: BIT_VECTOR(0 to 2) return BIT;

   --- Other declarations ----------

end HANDY

package body HANDY is
  function MAJ3(X: BIT_VECTOR(0 to 2))
    return BIT is
  begin
    return (X(0) and X(1)) or (X(0) and X(2)) or (X(1) and X(2));
  end MAJ3;

  --- Other subprogram declarations --------

end HANDY;
```

VHDL cont., Packages

- Packages included in a design by the use clause
    - use LIBRARY_NAME.PACKAGE_NAME.ELEMENT_NAME;
  - Element name ALL matches complete body declaration
- Standard packages: std.standard
  - boolean, bit, character, severity level, string, bit_vector, time
  - Automatically included
- Packages and libraries used to add new functionality to the language
  - Additional datatypes to support e.g., tristate levels, mathematical functions, etc.

Packages, 1164 standard

- IEEE.STD_LOGIC_1164
  - std_logic data types, overloaded boolean functions, conversion functions, edge detection functions
- IEEE.NUMERIC_STD
  - Overloaded arithmetic functions for std_logic_vector and conversion functions to/from integer
  - Defines UNSIGNED and SIGNED datatypes
    - Can be converted to/from std_logic_vector through typecast, e.g., std_logic_vector_signal <= std_logic_vector(unsigned_vectorsignal)
- IEEE.NUMERIC_BIT
  - Allows numeric processing of bitvectors

Configurations

- Defines which entity in a library should be used in the structural code
- Structural example: How to know which gate is used?
  - Component declaration name does not need to exist
  - Assume names are XOR2 and NAND2
- Configuration maps between entity and architecture in library and component name in structure

```
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;
```

Configuration example

- Add configuration statement to select exact instance and architecture for each instance.
- Drawback: require recompilation if library entity mapping is changed

```
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

    for U1 : XOR_GATE use entity work.xor2(behav);
    for U2 : XOR_GATE use entity work.xor2(behav);
    for U3 : NAND_GATE use entity work.nand2(behav);
    for U4 : NAND_GATE use entity work.nand2(behav);
    for U5 : NAND_GATE use entity work.nand2(behav);

    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;
```

Configurations, cont.

- Possible to create a configuration as a separate unit (here named STRUCT_BEHAV_XOR)

```
configuration STRUCT_BEHAV_XOR of ONES_COUNT is
    for BEHAV
        for U1:XOR_GATE use entity work.EXORGATE(BEHAV);
        end for;
        for U2:XOR_GATE use entity work.EXORGATE(BEHAV);
    end for;
end STRUCT_BEHAV_XOR;
```

- Multiple configurations possible for the same entity (similar to having multiple architectures)
  - Useful for testbench where both behavoral model and final netlist version of the device should be tested

Visibility

- Same name may be used for multiple purposes
  - Doing this may hide some units depending on the situation

```
package SIG is
  signal X: INTEGER:= 1;
end SIG;

use work.SIG.all;
entity Y is
  signal X: INTEGER:= 2;
end Y;
```

```
architecture Z of Y is
  signal Z1,Z2,Z3,Z4,Z5: INTEGER:= 0;
  function R return INTEGER is
    variable X: INTEGER := 3;
  begin
    return X; -- Returns value of 3.
  end R;
begin
  B: block
    signal X: INTEGER := 4;
    signal Z6: INTEGER := 0;
  begin
    Z6 <= X + Y.X;  -- Z6 = 6
  end block B;

P1: process
     variable X: INTEGER :=5;
  begin
    Z5 <= X;        -- Z5=5
    wait;
  end process;

  Z1 <= work.SIG.X; -- Z1=1
  Z2 <= X;          -- Z2=2
  Z3 <= R;          -- Z3=3
  Z4 <= B.X;        -- Z4=4
end Z;
```

Advanced feature: Overloading

- Can change meaning of literals, names of operators, functions, and procedures
- Correct version found by matching both name and parameter types
- If multiple identical declarations => latest visible one is used
- Useful feature for extending the language to support new datatypes or functions
  - E.g. extend to support complex valued real numbers
- IEEE standard 1164 is an example of this
  - Defines new datatypes as well as how comparison, assignments, arithmetic is done using these datatypes

IEEE std_logic_1164 package

- Define a new datatype and support functions
  - the std_logic data type
  - Overloads all common functions used for bits and bitvectors
    - AND, OR, NOT, XOR
    - <, >, = etc.
  - Conversion functions between std_logic and bit
- Netlists generated by synthesis use std_logic for all signals

```
type std_ulogic is ('u',  -- uninitialized
                    'x',  -- forcing  unknown
                    '0',  -- forcing  0
                    '1',  -- forcing  1
                    'z',  -- high impedance
                    'w',  -- weak unknown
                    'l',  -- weak 0
                    'h',  -- weak 1
                    '-'   -- don't care
                    );
```

Overloading example

- Example: Want to support new data type
  - F <= (A and B) or
            (C and D);
- new version of and
  - Uses table lookup with non-numeric indexes
  - Index number is based on position in definition enumeration of MVL4

```
type MVL4 is ('X', '0', '1', 'Z');

function “and” (L, R: MVL4) return MVL4 is
  -- Declare a two-dimensional table type.
  type MVL4_TABLE is array (MVL4,MVL4) of MVL4;
  -- truth table for “and” function
  constant table_AND: MVL4_TABLE :=
  -- ---------------------------------
  -- |  X     0     1      Z
  -- ---------------------------------
    (('X', '0', 'X', 'X'),    -- | X |
     ('0', '0', '0', '0'),    -- | 0 |
     ('X', '0', '1', 'X'),    -- | 1 |
     ('X', '0', 'X', 'X'));   -- | Z |
begin
  return table_AND(L,R);
end “and”;
```

File I/O

- Possible to read or write a file (1993 allow both on same file)
- Formatted IO
  - Not generally human readable (platform dependent)
- TEXT IO
  - Human readable
- Special package includes definitions
  - STD.TEXTIO
  - Functions for open file, read a complete line, and read individual data from the line

Later revisions

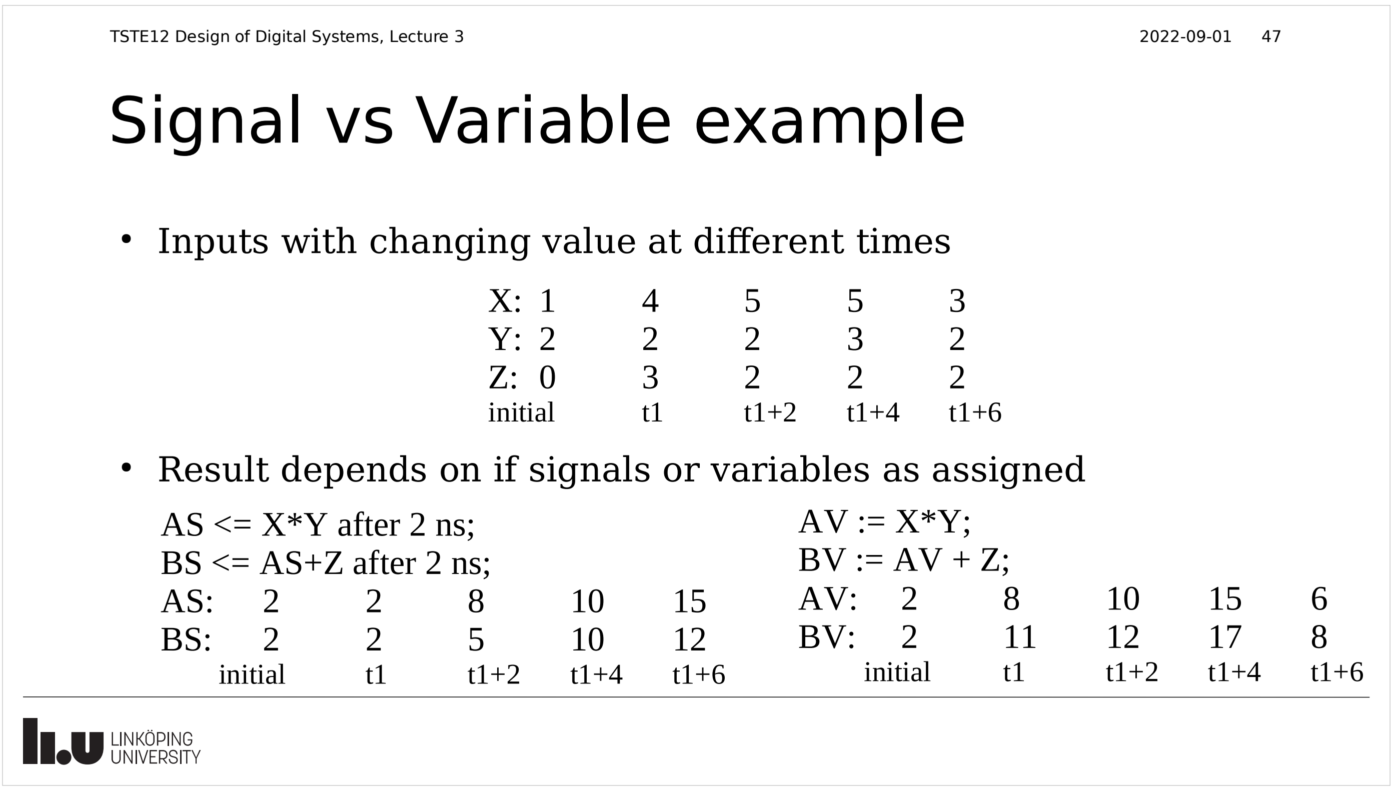
- Mostly simplifications and additional function support
- 1993:
  - 8-bit ASCII, identifier restrictions relaxed, declarations simplifications
  - Shared variables (global variables outside processes).
  - Improved reporting in assert statements
- 2008:
  - Simplified sensitivity lists (keyword all to include all signals used)
  - Simplified conditions, allow bit and std_logic values as result of condition
  - Read of output ports on entity

VHDL timing and concurrency

- Simulation of concurrent events (hardware) on a sequential computer
- Must have the same result from simulation independent of execution order of individual event
- Delay is an important property of hardware that must be simulated

Signals vs Variables

- Electronic signals can not change values in 0 seconds
  - Always slopes on voltages going from 0 to 1
- Common sequential code assumes variables are updated before next statement is executed
- Expect different result depending on if variables or signals are used
- Both variables and signals can be used in synthesized code



Signal vs Variable example

- Inputs with changing value at different times

X:	1	4	5	5	3
Y:	2	2	2	3	2
Z:	0	3	2	2	2
	initial	t1	t1+2	t1+4	t1+6

- Result depends on if signals or variables as assigned

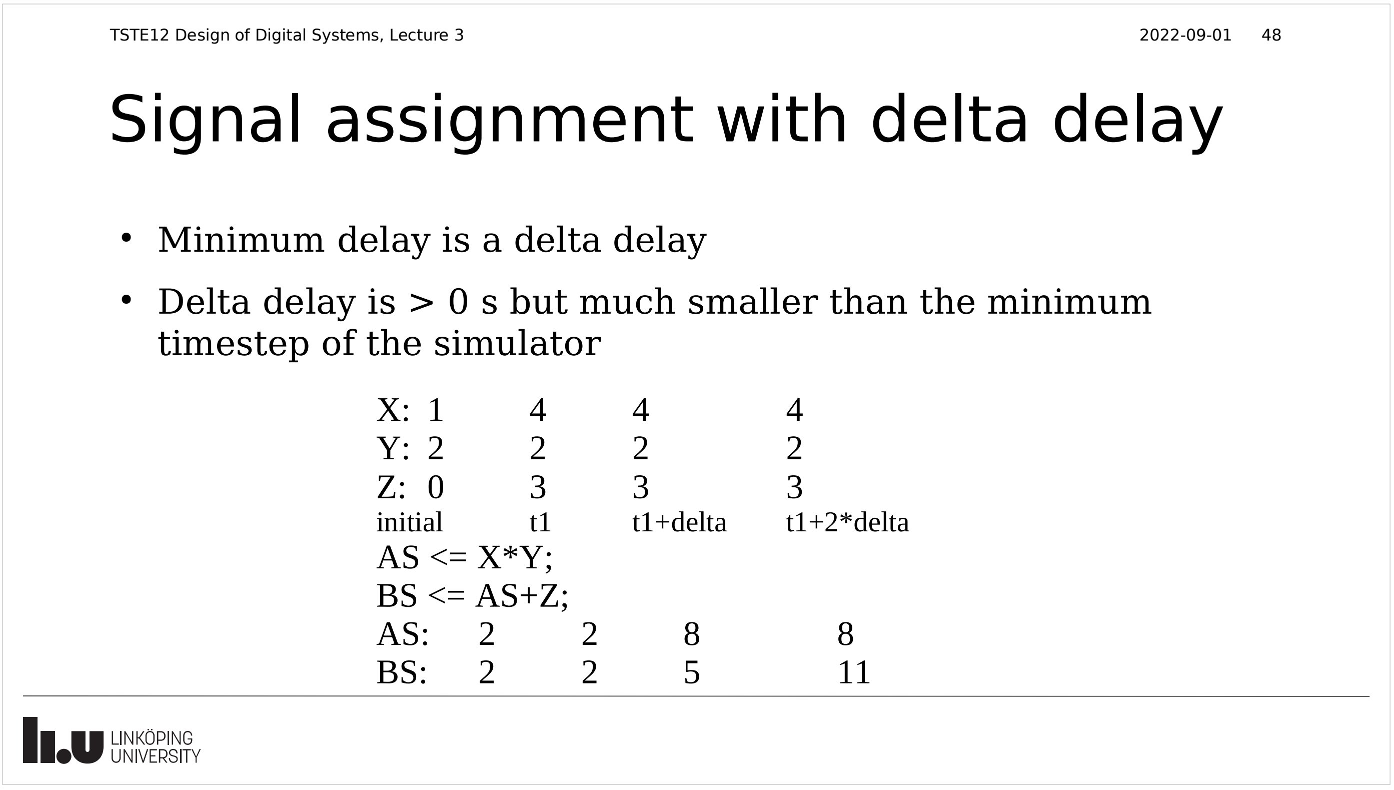
```
AS <= X*Y after 2 ns;
BS <= AS+Z after 2 ns;
```

AS:	2	2	8	10	15
BS:	2	2	5	10	12
	initial	t1	t1+2	t1+4	t1+6

```
AV := X*Y;
BV := AV + Z;
```

AV:	2	8	10	15	6
BV:	2	11	12	17	8
	initial	t1	t1+2	t1+4	t1+6

LINKÖPING UNIVERSITY



Signal assignment with delta delay

- Minimum delay is a delta delay
- Delta delay is > 0 s but much smaller than the minimum timestep of the simulator

X:	1	4	4	4
Y:	2	2	2	2
Z:	0	3	3	3
	initial	t1	t1+delta	t1+2*delta

```
AS <= X*Y;
BS <= AS+Z;
```

AS:	2	2	8	8
BS:	2	2	5	11

LINKÖPING UNIVERSITY

Delta delay

- Can not be explicitly specified
- Delta delays will never add up to a simulation delay in seconds (standard time)
- Sometimes referred to as Macro (simulation time) and micro (delta delays) timing.
- Time may stand still in simulation by continuous signal updates
  - Example: process triggered by a signal that it is updating
  - Combinatorial loops without macro delay in assignments
  - Delta delay is increasing but not the simulation time

Simulation models

- Delta delay only
  - Functional verification of models
- Standard time unit delay only
  - Validate system timing
- Mixed
  - Delta delay where delay is not important
  - Standard time unit delay where delay is significant
  - Study system timing

LINKÖPING
UNIVERSITY