



UiO : **Department of Informatics**
University of Oslo

IN 3160, IN4160

Verification part 2

Yngve Hafting



Messages

- The «unfair» challenge— how did it go?
- NOTE: Do not delay work until last week in later assignments...
- Video lectures by Alexander and Roar will be posted
 - It is OK to watch early
 - QnA session will be held at...

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.**

Course Goals and Learning Outcome

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

Goals for this lesson:

- To write self-testing testbenches
 - What is self-testing test benches
 - File IO in VHDL
 - VHDL attributes used in test benches
 - Assertions
- To understand set-up and hold-time
 - Be able to check for violations
- To generate test-bench clocks that emulate real world clocks

Next lesson

- Finite state machines (FSM's)

Outline

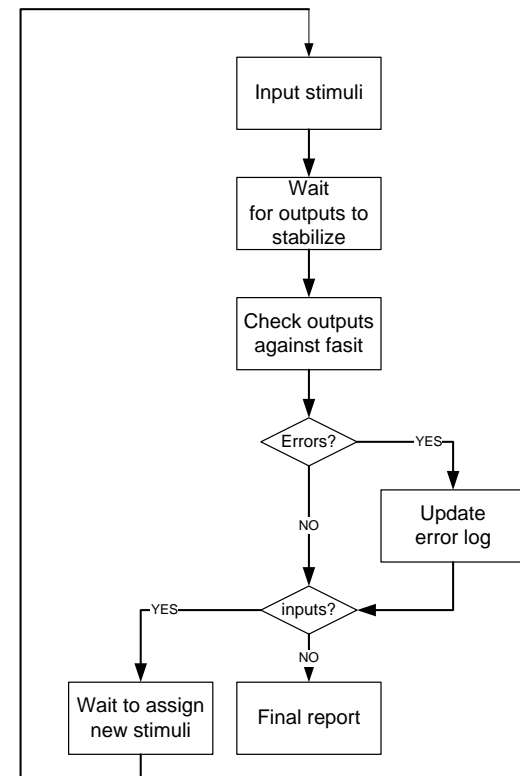
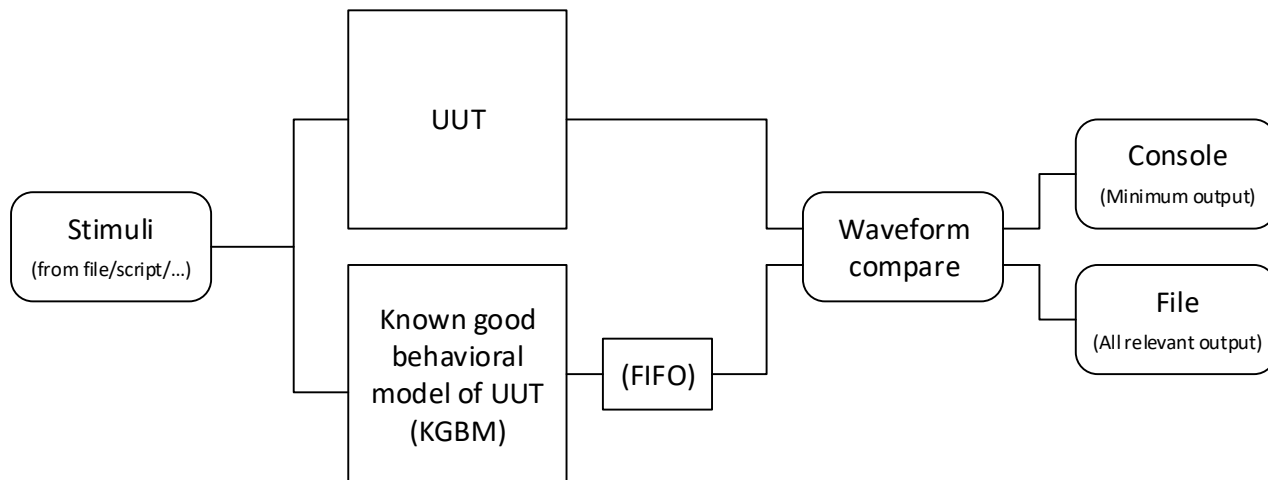
- Self checking testbenches Concept
- Assertions
- File IO

- Example synthesizable File IO
- Example- self checking test bench

- Set-up / hold time for FFs
- Timing checks
 - Relevant attributes for assertions
- Clock generator for testbench

Self checking test benches

- Two perspectives
 - As a system of modules
 - As a finite state machine



Useful libraries in test benches

- `std.textio` from IEEE contains procedures for reading from and writing to file
- (see next page for package declaration)
- Standard VHDL package declarations can be found by searching the web (if you do know their name)

L (**line**) is the **access** (pointer) to the «current» position in a text

Note: L is **inout** since it is both read and set by the procedure

```
package TEXTIO is
```

```
  type LINE is access string;  
  type TEXT is file of string;  
  type SIDE is (right, left);  
  subtype WIDTH is natural;
```

```
  file input : TEXT open READ_MODE is "STD_INPUT";  
  file output : TEXT open WRITE_MODE is "STD_OUTPUT";
```

```
  procedure READLINE (file F: TEXT; L: inout LINE);
```

```
  procedure READ (L:inout LINE; VALUE: out bit; GOOD : out BOOLEAN);  
  procedure READ (L:inout LINE; VALUE: out bit);
```

```
  procedure READ (L:inout LINE; VALUE: out bit_vector; GOOD : out BOOLEAN);  
  procedure READ (L:inout LINE; VALUE: out bit_vector);
```

```
  procedure READ (L:inout LINE; VALUE: out BOOLEAN; GOOD : out BOOLEAN);  
  procedure READ (L:inout LINE; VALUE: out BOOLEAN);
```

```
  procedure READ (L:inout LINE; VALUE: out character; GOOD : out BOOLEAN);  
  procedure READ (L:inout LINE; VALUE: out character);
```

```
  procedure READ (L:inout LINE; VALUE: out integer; GOOD : out BOOLEAN);  
  procedure READ (L:inout LINE; VALUE: out integer);
```

```
  procedure READ (L:inout LINE; VALUE: out real; GOOD : out BOOLEAN);  
  procedure READ (L:inout LINE; VALUE: out real);
```

```
  procedure READ (L:inout LINE; VALUE: out string; GOOD : out BOOLEAN);  
  procedure READ (L:inout LINE; VALUE: out string);
```

```
  procedure READ (L:inout LINE; VALUE: out time; GOOD : out BOOLEAN);  
  procedure READ (L:inout LINE; VALUE: out time);
```

```
  procedure WRITELINE (file F : TEXT; L : inout LINE);
```

```
  procedure WRITE (L :inout LINE; VALUE : in bit;  
  JUSTIFIED: in SIDE := right;  
  FIELD: in WIDTH := 0);
```

```
  procedure WRITE (L : inout LINE; VALUE : in bit_vector;  
  JUSTIFIED: in SIDE := right;  
  FIELD: in WIDTH := 0);
```

```
  procedure WRITE (L : inout LINE; VALUE : in BOOLEAN;  
  JUSTIFIED: in SIDE := right;  
  FIELD: in WIDTH := 0);
```

```
  procedure WRITE (L : inout LINE; VALUE : in character;  
  JUSTIFIED: in SIDE := right;  
  FIELD: in WIDTH := 0);
```

```
  procedure WRITE (L : inout LINE; VALUE : in integer;  
  JUSTIFIED: in SIDE := right;  
  FIELD: in WIDTH := 0);
```

```
  procedure WRITE (L : inout LINE; VALUE : in real;  
  JUSTIFIED: in SIDE := right;  
  FIELD: in WIDTH := 0;  
  DIGITS: in NATURAL := 0);
```

```
  procedure WRITE (L : inout LINE; VALUE : in string;  
  JUSTIFIED: in SIDE := right;  
  FIELD: in WIDTH := 0);
```

```
  procedure WRITE (L : inout LINE; VALUE : in time;  
  JUSTIFIED: in SIDE := right;  
  FIELD: in WIDTH := 0;  
  UNIT: in TIME := ns);
```

```
end TEXTIO;
```

File IO

- Synthesis
 - Mostly used for reading ROM content
 - Strictly not supported by VHDL-> vendor specific solutions
 - *Vivado synthesis can only use std_logic or bit, no integers*
- Simulation
 - Stimuli (input)
 - Response (logging)
 - Data output
 - Errors and other messages

File IO

- Binary files
 - Can output whole types (custom types, records / anything)
 - Only one type per file
 - *Tool specific* (non portable code)
- Text files
 - Can contain anything
 - Human readable
 - A bit trickier to use (text to type conversions...)
- *We will use text files*

Example: File IO for synthesis of ROM 1/2

```
library IEEE;
  use IEEE.STD_LOGIC_1164.all;
  use IEEE.numeric_std.all;
  use STD.textio.all;

entity ROM is
  generic(
    data_width: natural := 8;
    addr_width: natural := 2;
    filename: string := "ROM data bits.txt"
  );
  port(
    address: in std_logic_vector(addr_width-1 downto 0);
    data: out std_logic_vector(data_width-1 downto 0));
end entity;
```

- 4 byte ROM example
 - 8 bit data
 - 2 bit address
- Libraries
 - Remember std.textio
- File name
 - Assuming project (work) directory

Example: File IO for synthesis of ROM 1/2

```
type memory_array is array(2**addr_width-1 downto 0) of
std_logic_vector(data_width-1 downto 0);

impure function initialize_ROM(file_name: string)
return memory_array is
file init_file: text open read mode is file_name;
variable current_line: line;
variable result: memory_array;
begin
for i in result'range loop
readline(init_file, current_line);
read(current_line, result(i));
end loop;
return result;
end function;

--initialize rom:
constant ROM_DATA: memory_array := initialize_ROM(filename);

begin
data <= ROM_DATA(to_integer(unsigned(address)));
end;
```

Combinational implementation

- Tool specific: Vivado won't allow for integers being read from file or strings
 - Integer data will have to be converted to '1' and '0' (without '_').
- Impure:
 - Does not always return the same result using same input parameters (due to file usage)
- *File is a text we open in read mode*
- Line is "access" type which means
 - A pointer to a position in the file
- Readline
 - Sets the line pointer to the beginning of the (first or) next line
- Read
 - Sets the data parameter
 - Sets the line pointer to the next data (or end of line)
 - Whitespace is delimiter
- What do we get if we set ROM_DATA to a signal?

Assertions - «To ensure a model is working with valid inputs»*

- Syntax

```
assert <boolean condition> -- report when false
report <string>
severity <note, warning, error, failure>;
```
- Compilation
 - Can be used to check for size mismatches at compile time.
- RTL Simulation
 - Compare simulated and expected outcome values (behavior)
- Post Synthesis simulation
 - Checks on signal timing attributes in addition to behavior
- Severity levels
 - Failure means «simulation should be stopped»
 - Usually when a module cant be initiated correctly, something doesn't compile...
 - Error – when the model provides wrong output or goes into wrong state
 - Warning – «*unexpected conditions that do not affect the state of the model*»
 - Note – to report when everything went well (default for report)

Example Self-checking test bench 1/3

```
library IEEE;  
  use IEEE.STD_LOGIC_1164.all;  
  use IEEE.numeric_std.all;  
  use STD.textio.all;
```

```
entity tb_ROM is  
end entity;
```

```
architecture behavioral of tb_ROM is  
  constant data_width: natural := 8;  
  constant addr_width: natural := 2;
```

```
  component ROM is  
    generic(  
      data_width: natural := 8;  
      addr_width: natural := 2;  
      filename: string := "ROM data bits.txt");  
    port(  
      address: in std_logic_vector(addr_width-1 downto 0);  
      data: out std_logic_vector(data_width-1 downto 0));  
  end component;
```

```
  signal tb_data : std_logic_vector(data_width-1 downto 0);  
  signal tb_address: std_logic_vector(addr_width-1 downto 0) := "00";
```

```
begin  
  DUT: ROM  
  port map(  
    address => tb_address,  
    data => tb_data);
```

- Libraries
 - std.textio ++
- Generics for RTL simulation only
 - Post synthesis want these removed or commented out
- Default values only for stimuli generated by testbench
 - *Do not set signals for component outputs!*

```

process is
file stimuli_file: text open read_mode is "ROM_stimuli.txt";
variable stimuli_line: line;
variable stimuli_address: integer;
variable stimuli_data: integer;

procedure set_stimuli is
begin
readline(stimuli_file, stimuli_line);
read(stimuli_line, stimuli_address);
read(stimuli_line, stimuli_data);
tb_address <= std_logic_vector(to_unsigned(stimuli_address, addr_width));
end procedure;

file log_file: text open write_mode is "ROM_results_and_log.txt";
variable log_line: line;

procedure check_output is
constant ADR_DIGITS : integer := 2; -- size address as base 10 number
constant DAT_DIGITS : integer := 4; -- size data as base 10 number
constant SPACER: integer := 1;
begin
--report errors to console
assert (tb_data = std_logic_vector(to_signed(stimuli_data, data_width)))
report ("DATA MISMATCH for address: ", integer'image(stimuli_address))
severity error;
-- report to file
write(log_line, stimuli_address, field => ADR_DIGITS);
write(log_line, stimuli_data, field => DAT_DIGITS + SPACER);
write(log_line, tb_data, field => tb_data'length + SPACER);
writeline(log_file, log_line);
end procedure;
begin
while not endfile(stimuli_file) loop
set_stimuli;
wait for 1 ns;
check_output;
end loop;
file_close(stimuli_file);
file_close(log_file);
report ("Testing finished!");
std.env.stop;
end process;
end architecture;

```

- Why do we put our procedures in process, not architecture declaration?

ROM_DATA_bits.txt

```
00000011
00001100
00010111
10000010
```

ROM_stimuli.txt

```
000 -126
001 23
002 10
003 3
```

ROM_results_and_log.txt

- **Synthesizable**
 - '1' and '0' stored as text
 - Only partial VHDL implementation
 - No integers or other types
 - No underscores
 - Different tool = different issues
- **Simulation only**
 - Any type stored as text
 - Full VHDL implementation
 - Whitespace >1 = OK
 - Good practice:
 - Use human readable values
 - integers or hex values > binary
- **Our output data**
 - We decide format
 - Try to make output that
 - is readable and
 - understandable
 - can be used to check data

```
package TEXTIO is
  type LINE is access string;
  type TEXT is file of string;
  type SIDE is (right, left);
  subtype WIDTH is natural;

  file input : TEXT open READ_MODE is "STD_INPUT";
  file output : TEXT open WRITE_MODE is "STD_OUTPUT";

  procedure READLINE(file F: TEXT; L: inout LINE);

  procedure READ(L:inout LINE; VALUE: out bit; GOOD : out BOOLEAN);
  procedure READ(L:inout LINE; VALUE: out bit);

  procedure READ(L:inout LINE; VALUE: out bit_vector; GOOD : out BOOLEAN);
  procedure READ(L:inout LINE; VALUE: out bit_vector);

  procedure READ(L:inout LINE; VALUE: out BOOLEAN; GOOD : out BOOLEAN);
  procedure READ(L:inout LINE; VALUE: out BOOLEAN);

  procedure READ(L:inout LINE; VALUE: out character; GOOD : out BOOLEAN);
  procedure READ(L:inout LINE; VALUE: out character);

  procedure READ(L:inout LINE; VALUE: out integer; GOOD : out BOOLEAN);
  procedure READ(L:inout LINE; VALUE: out integer);

  procedure READ(L:inout LINE; VALUE: out real; GOOD : out BOOLEAN);
  procedure READ(L:inout LINE; VALUE: out real);

  procedure READ(L:inout LINE; VALUE: out string; GOOD : out BOOLEAN);
  procedure READ(L:inout LINE; VALUE: out string);

  procedure READ(L:inout LINE; VALUE: out time; GOOD : out BOOLEAN);
  procedure READ(L:inout LINE; VALUE: out time);

  procedure WRITELINE(file F : TEXT; L : inout LINE);

  procedure WRITE(L :inout LINE; VALUE : in bit;
    JUSTIFIED: in SIDE := right;
    FIELD: in WIDTH := 0);

  procedure WRITE(L : inout LINE; VALUE : in bit_vector;
    JUSTIFIED: in SIDE := right;
    FIELD: in WIDTH := 0);

  procedure WRITE(L : inout LINE; VALUE : in BOOLEAN;
    JUSTIFIED: in SIDE := right;
    FIELD: in WIDTH := 0);

  procedure WRITE(L : inout LINE; VALUE : in character;
    JUSTIFIED: in SIDE := right;
    FIELD: in WIDTH := 0);

  procedure WRITE(L : inout LINE; VALUE : in integer;
    JUSTIFIED: in SIDE := right;
    FIELD: in WIDTH := 0);

  procedure WRITE(L : inout LINE; VALUE : in real;
    JUSTIFIED: in SIDE := right;
    FIELD: in WIDTH := 0;
    DIGITS: in NATURAL := 0);

  procedure WRITE(L : inout LINE; VALUE : in string;
    JUSTIFIED: in SIDE := right;
    FIELD: in WIDTH := 0);

  procedure WRITE(L : inout LINE; VALUE : in time;
    JUSTIFIED: in SIDE := right;
    FIELD: in WIDTH := 0;
    UNIT: in TIME := ns);

end TEXTIO;
```


Post synthesis testbenches

- Post synthesis, post route
 - Tools may vary =
 - Adaptations may be necessary to compile
 - Generics may be gone
 - Assertions in synthesized code will be gone
 - Timing information will be there
 - Much more to test on...
 - Signal attributes next slide
 - Don't replace static timing analysis and constraints
 - Timing constraints are used for synthesis...

Signal Attributes 1/2

These attributes are predefined for any signal X:

Name	Definition
X'event	True when X changes (boolean)
X'active	True when X assigned to (boolean)
X'last_event	When X last changed (time)
X'last_active	When X was last assigned to (time)
X'last_value	Previous value of X (same type as X)

- These are signal only!
 - Each signal maintains these throughout simulation
 - *Variables don't have these*
 - => v. faster in simulation
- 'event used in **rising_edge()**
 - (other use not intended for synthesis)
- 'last...
 - Can be useful in testbenches
 - Example (oblig 8):

```
assert en'last_event < LONG_PWM_CYCLE/2
report "PWM is not happening,.."
severity error;
```

Signal Attributes 2/2

These attributes create a **new signal**, based on signal X:

Name	Definition
X'delayed(T)	X, delayed by T (same type as X)
X'stable(T)	True if X unaltered for time T (boolean)
X'quiet(T)	True if X unassigned for time T (boolean)
X'transaction	"Toggles" when X is assigned (bit)

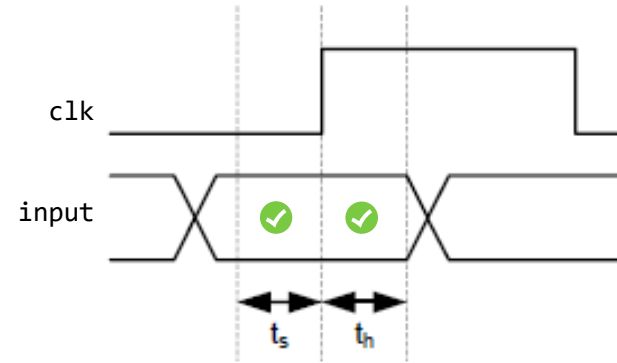
- May be used to create simulation logic and tests
 - (not synthesizable)

More on attributes

- There are attributes for
 - Signals
 - (previous slides)
 - Types
 - Notable:
 - ‘image(v) returns a string ex : `report("current value is: ", integer'image(my_int));`
 - ‘value(s) returns a value (opposite of ‘image) `integer'value(my_str);`
 - Array types/objects (vectors)
 - ‘left, ‘right, ‘low, ‘high, ‘range, ‘reverse_range, ‘length, ‘ascending (= false when «downto»), ‘element (== subtype of the vector)
 - Entities
 - attributes to get compiled name hierarchy- as seen in questa when selecting signals

Testcase: Set-up/hold time in flipflops

- To avoid metastability (neither 0 nor 1), inputs must be stable some time before (set-up) and after (hold) clock edge
- Output will return to 0 or 1 after being in the metastable state, but it's not given which one.
 - This means; the system is no longer deterministic.



Timing and logic check

```
27: entity D_FF is
28:
29:
30:
31: begin
32:
33:
34:
35:
36:
37:
38:
39: end entity D_FF;
```

- The stable attribute can be used to check set-up- and hold times
 - Returns true if a signal has been stable \geq time given as input parameter
- Assert in an entity \Rightarrow checking is being done for all architectures that belongs to this entity.

CAUTION! Care should be taken using asserts. Vivado can only support static asserts that do not create, or are created by, behavior. For example, performing an assert on a value of a constant or a operator/generic works; however, an assert on the value of a signal inside an if statement will not work.

Clock generator

- Asymmetric low and high time (dutycycle)

```
26 entity clock_gen is
27 |   generic (Freq : REAL := 10.0;  -- MHz
28 |           |   Mark : REAL := 0.3);  -- Mark length (0-1.0)
29   end entity clock_gen;
30
31 architecture cg of clock_gen is
32 |   -- Mark time in us
33   constant ClockHigh :TIME := (Mark/Freq)*us;
34 |   -- Space time in us
35   constant ClockLow :TIME := ((1.0-Mark)/Freq)*us;
36   signal clock : std_logic := '0';
37 begin
38 |   process is
39 |   |   begin
40 |   |   |   wait for ClockLow;
41 |   |   |   |   clock <= '1';
42 |   |   |   |   wait for ClockHigh;
43 |   |   |   |   clock <= '0';
44 |   |   |   end process;
45 end architecture cg;
```

Example: Clock with jitter

- Jitter:
 - (random) variable delay
 - Occurs naturally in all digital electronic
- math_real.uniform:

```
procedure UNIFORM(  
  variable SEED1, SEED2 : inout POSITIVE;  
  variable X : out REAL);
```

 - pseud-random number generator procedure
 - uniform distribution
 - alters seed values and sets rnd number

```
1: library IEEE;  
2: use IEEE.std_logic_1164.all;  
3: use IEEE.math_real.all;  
4:  
5: Entity RAND_CLOCK is  
6: -- generic parameters  
7:   generic (delay : DELAY_LENGTH := 100 ns);  
8:   port(clock : out std_logic);  
9: end entity RAND_CLOCK;  
10:  
11: architecture RTL_RAND_CLOCK of RAND_CLOCK is  
12:  
13: begin  
14:  
15: RAND_CLK:  
16: process  
17:   variable seed1, seed2 : INTEGER := 42;  
18:   variable rnd : REAL;  
19: begin  
20:   loop  
21:     clock <= '0';  
22:     uniform (seed1, seed2, rnd);  
23:     wait for delay + (rnd - 0.5) * (10 NS);  
24:     clock <= '1';  
25:     uniform (seed1, seed2, rnd);  
26:     wait for delay + (rnd - 0.5) * (10 NS);  
27:   end loop;  
28: end process;  
29:  
30: end architecture RTL_RAND_CLOCK;
```


Suggested reading

- D&H
 - File access, ROM
 - 8.8 p184-189
 - Attributes
 - B.8 p 638-640
 - Timing constraints:
 - 15.1-3 p 328 - 334
 - 15.4-6 p 334- 340