UiO : Department of Informatics
University of Oslo

IN 3160, IN4160

Verification part 2 File IO

Yngve Hafting

Messages

- Watch videos posted by Alexander Wold before session monday (see timeplan). (=Flipped classroom)

- Roar will run ordinary lectures this year.

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:

- 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)

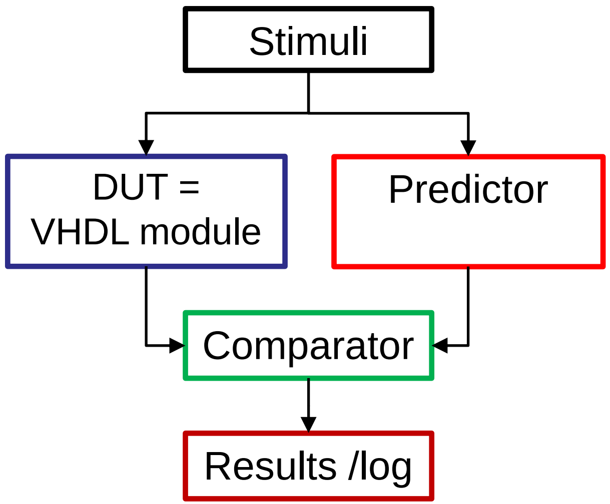
Outline

- Testbench
  - General layout (repetition)
  - Self checking testbenches concept
  - What should a good TB do
  - TB output
- Assertions
- File IO

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

- Set-up / hold time for FFs
- Timing checks

General testbench layout (R)

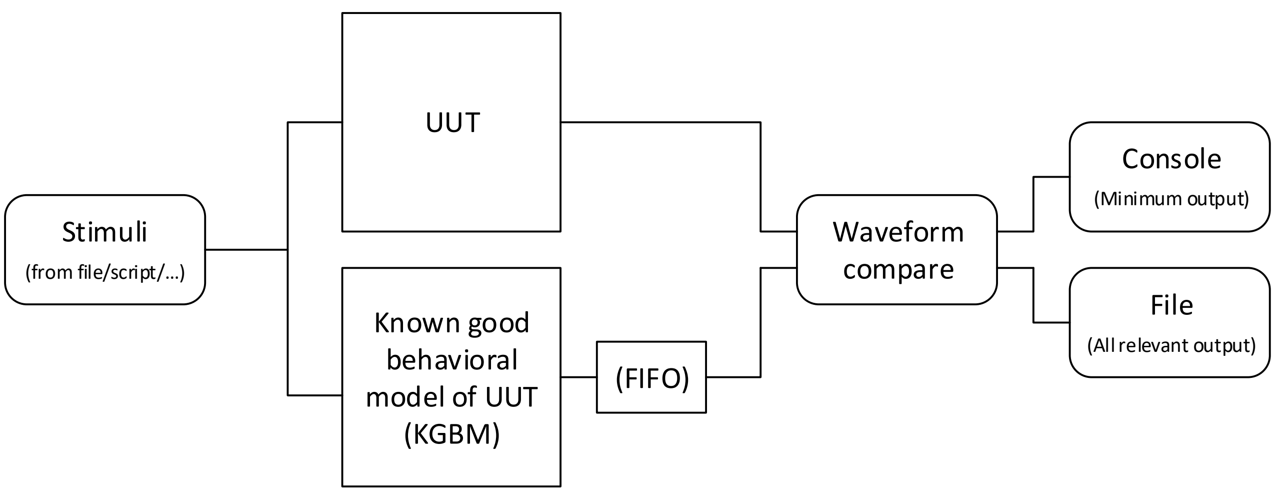


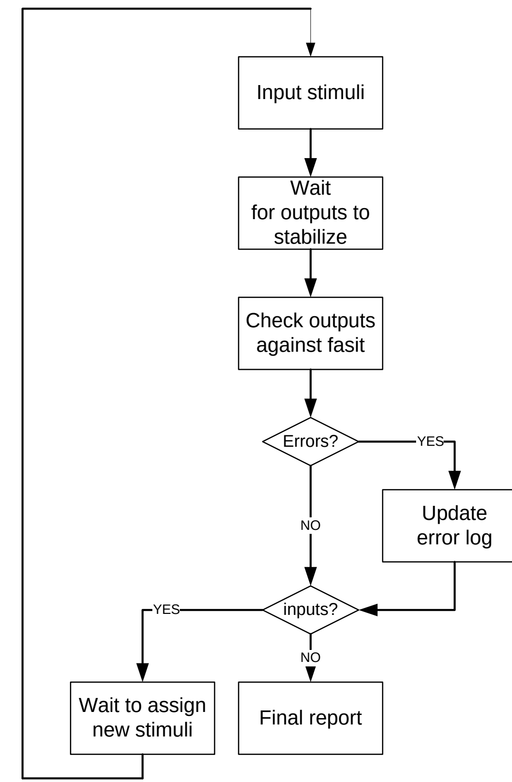
- **Stimuli**
  - Generate or read stimuli from a file
  - Use procedures rather than repeating lines
- **DUT**
  - Device under test (Device, Module, ...)
  - Connect DUT input to stimuli to create simulation results
- **Predictor**
  - Predicts what the output should be
    - Calculates from input or reads from file
- **Comparator**
  - Compares simulation result with predicted result and reports to screen or file.

Self checking test benches

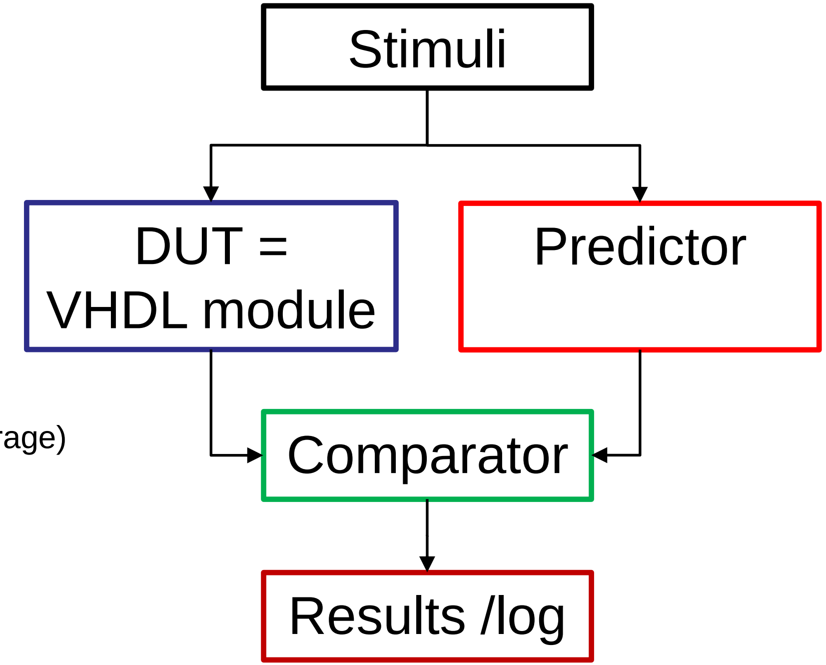
- -performs tests and reports to screen or file (*timing diagram is only used when debugging*)

- Two perspectives
  - As a system of modules
  - As a finite state machine
    - *Does not reflect stimuli independent testing*





A good testbench:



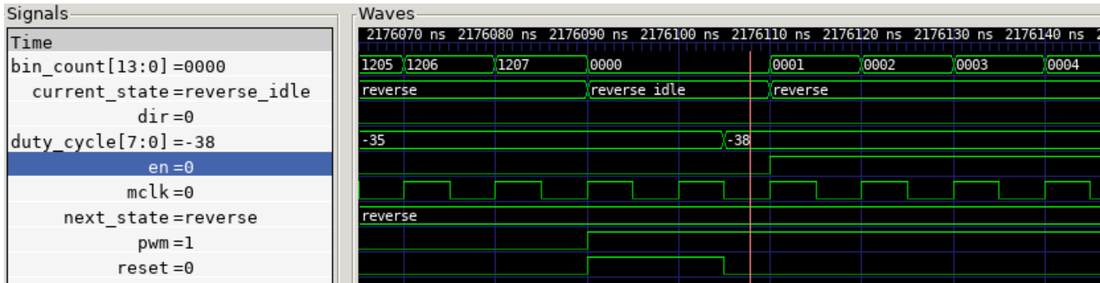
- Tests should
  - run independently of stimuli
  - run throughout simulation
  - cover all (100%) specified behaviour
    - Catch all deviations from known good behavior
- Stimuli
  - Should cover all types of behavior
    - All design (VHDL) code should be run (100% code coverage)
    - All corner cases or
      - Formal verification : All possible input
        - » = not possible in most cases
        - » (eg 32 bit adder = $(2^{32})^2$ combinations)
  - Normally sets DUT-inputs only
    - never overwrite signals inside DUT
      - => create and test submodules separately if this seems needed...
- Fault injection
  - Proves that the each test will catch errors
    - Fault injection can use "Force" to overwrite signals from or in DUT

Test bench output

- Report
  - which test are performed
  - what stimuli are applied
  - successful completion of
    - Stimuli series
    - Tests
  - Errors
    - Expected vs simulation result
    - Timing
- Create relevant waveforms
  (we do not automate this)
  - Before errors or deviations occur
  - Relevant input, output
  - Internal states (preferably with names)

Example (not perfect):

```
     1.00ns INFO   cctb.pwm   Starting monitoring events
     1.00ns INFO   cctb.pwm   Starting duty cycle tests
    15.00ns INFO   cctb.pwm   Passed: Reset test
163850.00ns INFO   cctb.pwm   Duty cycles: Set dc: 50.0%, Measured dc:  49.0%, period = 163.8us, f = 6.11kHz
327690.00ns INFO   cctb.pwm   Duty cycles: Set dc: -50.0%, Measured dc: -49.0%, period = 163.8us, f = 6.10kHz
327690.00ns INFO   cctb.pwm   Sequential duty tests complete
491530.00ns INFO   cctb.pwm   Duty cycles: Set dc: 27.3%, Measured dc:  27.0%, period = 163.8us, f = 6.10kHz
819210.00ns INFO   cctb.pwm   Duty cycles: Set dc: 50.0%, Measured dc:  49.0%, period = 163.8us, f = 6.10kHz
983050.00ns INFO   cctb.pwm   Duty cycles: Set dc: -84.4%, Measured dc: -84.0%, period = 163.8us, f = 6.10kHz
1310730.00ns INFO  cctb.pwm   Duty cycles: Set dc: -33.6%, Measured dc: -33.0%, period = 163.8us, f = 6.10kHz
1474570.00ns INFO  cctb.pwm   Duty cycles: Set dc: 41.4%, Measured dc:  41.0%, period = 163.8us, f = 6.10kHz
1638410.00ns INFO  cctb.pwm   Duty cycles: Set dc: -23.4%, Measured dc: -23.0%, period = 163.8us, f = 6.10kHz
1966090.00ns INFO  cctb.pwm   Duty cycles: Set dc: -27.3%, Measured dc: -27.0%, period = 163.8us, f = 6.10kHz
2129930.00ns INFO  cctb.pwm   Duty cycles: Set dc: -27.3%, Measured dc: -27.0%, period = 163.8us, f = 6.10kHz
2176090.00ns INFO  cctb.pwm   Random duty tests 1/2 complete
2176090.00ns INFO  cctb.pwm   Resetting module...
2176105.00ns INFO  cctb.pwm   Reset between duties complete
2176105.00ns INFO  cctb.pwm   Passed: Reset test
2339940.00ns INFO  cctb.pwm   Duty cycles: Set dc: -29.7%, Measured dc: -29.0%, period = 163.8us, f = 6.10kHz
2667620.00ns INFO  cctb.pwm   Duty cycles: Set dc: -69.5%, Measured dc: -69.0%, period = 163.8us, f = 6.10kHz
2831460.00ns INFO  cctb.pwm   Duty cycles: Set dc: 27.3%, Measured dc:  27.0%, period = 163.8us, f = 6.10kHz
2943460.00ns INFO  cctb.pwm   Random duty tests 2/2 complete
2943460.00ns INFO  cctb.regr   main_test passed
  ******************************************************************************************
  ** TEST                              STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
  ******************************************************************************************
  ** tb_pwm.main_test                   PASS      2943460.00          11.54     255164.53  **
  ******************************************************************************************
  ** TESTS=1 PASS=1 FAIL=0 SKIP=0               2943460.00          11.66     252500.41  **
  ******************************************************************************************
```



How to organize a medium sized testbench

- Create separate classes for
  - Generic signal monitoring
    - replaces use of 'attributes in VHDL
      - can be much more extensive than attributes
  - DUT Monitoring
    - Contains all tests, their trigger and reports
  - Stimuli generation
    - Creates all input sequentially
      - Either timed or based on response
    - Uses DUT-inputs
  - Fault injection
    - can be run with DUT being an empty entity.
    - Forces/overrides signals used in tests

- Separate location for classes will ensure better readability

- Allow the simulator to run as much as possible...
  - Test only on appropriate triggers
    - everything on every clock => slow test.
    - Do not store values that can be easily calculated

VHDL libraries for test benches/ file IO

- 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 (2020) 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?*
- By initializing as default value this memory can be synthesized with file usage

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
  - Cheks 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)

* Richardo Jasinski «Effective coding with VHDL»

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
  - For post synthesis simulation:
    - Synthesis will already have used any generic for creating sizes, unless you are working with a behavioral model of a component.
- Default values for stimuli generated by testbench
  - *Do not set default values for component outputs!*
    - *May hide initialization errors*

Example Self-checking test bench 2/3

```
STIMULI: 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 adress 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?

Example Self-checking test bench 3/3

ROM_DATA_bits.txt

```
00000011
00001100
00010111
10000010
```

- Synthesizable
  - ‘1’ and ‘0’ stored as text
  - Only partial VHDL implementation
    - No integers or other types
    - No underscores
  - Different tool = different issues

ROM_stimuli.txt

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

- Simulation only
  - Any type stored as text
  - Full VHDL implementation
    - Whitespace >1 = OK
  - Good practice:
    - Use human readable values
    - integers or hex values > binary

ROM_results_and_log.txt

- Our output data
  - We decide format
  - Try to make output that
    - is readable and
    - understandable
    - can be used to check data

We used std_logic_vector similar to bit_vector

```
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, post implementation simulation and testbenches

- Mostly relevant for ASIC design

- Post synthesis, post route
  - Using the same testbench may be difficult
    - Simulation information in design files will normally be stripped during synthesis.
      - Assertions will be gone
    - Adaptations may be necessary to compile
      - Generics may be frozen/ not generic
  - Timing information will be there
    - Much more to test on…
      - Signal attributes next slide
    - Does not replace static timing analysis and constraints
      - Timing constraints are used for synthesis…

https://docs.xilinx.com/r/en-US/ug900-vivado-logic-simulation/Post-Synthesis-Simulation

Performing a thorough timing simulation ensures that the completed design is free of defects that could otherwise be missed, such as:

- functionality changes that are caused by:
  - Synthesis properties or constraints that create mismatches
  - UNISIM properties applied in the Xilinx Design Constraints (XDC) file
  - The interpretation of language during simulation by different simulators
- Dual port RAM collisions
- Missing, or improperly applied timing constraints
- Operation of asynchronous paths
  - Functional issues due to optimization techniques

Signal Attributes for simulation 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 for simulation 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)

Attributes in cocotb?

- Cocotb and GHDL does not (Jan. 2023) have built in support for VHDL signal attributes.
  - Other simulators may have an API for this...
- Solution: create a signal monitor class
  - It will run slightly slower than a pure VHDL simulation due to added context switching
- Alternative: build a new top layer in VHDL with signal attributes as outputs along with the other signals.
  - Downside for this is spreading testbench code into multiple modules and languages

```
import cocotb
from cocotb import start_soon
from cocotb.triggers import Edge, ReadOnly
from cocotb.utils import get_sim_time

# Conversion to pico-seconds using dictionary
ps_conv = {'fs': 0.001, 'ps': 1, 'ns': 1000, 'us': 1e6, 'ms':1e9}

class SignalEventMonitor():
    """ Tracks a signals last event.  """
    def __init__(self, signal):
        self.signal = signal
        self.last_event = get_sim_time('ps')
        self.last_rise = self.last_event
        self.last_fall = self.last_event
        start_soon(self.update())

    async def update(self):
        while 1:
            await Edge(self.signal)
            # Avoid multiple triggers on a single event
            await ReadOnly()
            self.last_event = get_sim_time('ps')
            if self.signal == 1: self.last_rise = self.last_event
            else: self.last_fall = self.last_event

    def stable_interval(self, units='ps'):
        # convert last_event to the prefix in use
        last_event_c = self.last_event/ps_conv[units]
        # calculate stable interval
        stable = get_sim_time(units) - last_event_c
        return stable


#... Monitoring a signal ...
en_mon  = SignalEventMonitor(self.dut.en)
PERIOD_NS = 10
#...
assert en_mon.stable_interval('ns') > PERIOD_NS-1, (
        "not stable long enough!")
```

core "attributes"

Monitoring service

Secondary attribute calculation on demand

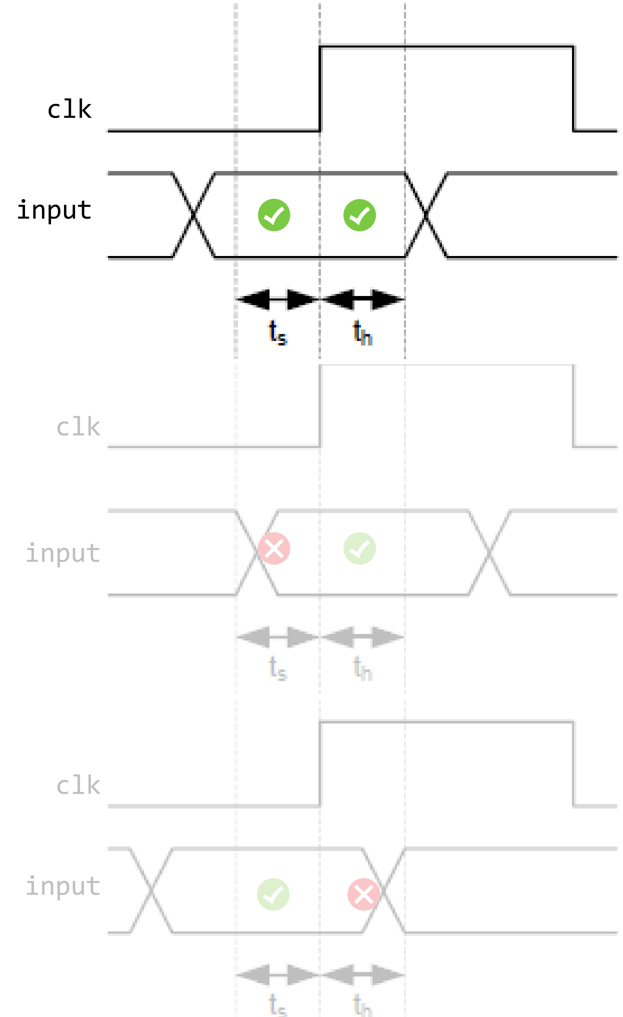
More on VHDL attributes

- 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 simulator 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 (not shown) 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

```
entity D_FF is

   port (D, Clk, Set, Reset: in std_logic;
         Q : out std_logic);
begin
   assert (not(Clk ='1' and Clk'EVENT and not D'STABLE(Setup)))
   report "Setup time violation" severity WARNING;
   assert (not(Clk ='1' and D'EVENT and not Clk'STABLE(Hold)))
   report "Hold time violation" severity WARNING;
   assert (not(Set ='0' and Reset = '0'))
   report "Set and Reset are both asserted"
   severity ERROR;
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 >= time given as input parameter
- Assert in an entity =>
  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 as assert on a value of a constant or a operator/generic works; however, as asset on the value of a signal inside an* `if` *statement will not work.*

Clock generator

- Asymmetric low and high time (dutycycle)

```
entity clock_gen is
  generic (Freq : REAL := 10.0;  -- MHz
           Mark : REAL := 0.3);  -- Mark length (0-1.0)
end entity clock_gen;

architecture cg of clock_gen is
  -- Mark time in us
  constant ClockHigh :TIME := (Mark/Freq)*us;
  -- Space time in us
  constant ClockLow :TIME := ((1.0-Mark)/Freq)*us;
  signal clock : std_logic := '0';
begin
    process is
      begin
      wait for ClockLow;
      clock <= '1';
      wait for ClockHigh;
      clock <= '0';
    end process;
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

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.math_real.all;

Entity RAND_CLOCK is
-- generic parameters
  generic (delay : DELAY_LENGTH := 100 ns);
  port(clock : out std_logic);
end entity RAND_CLOCK;

architecture RTL_RAND_CLOCK of RAND_CLOCK is

begin

RAND_CLK:
process
  variable seed1, seed2 : INTEGER := 42;
  variable rnd : REAL;
begin
  loop
    clock <= '0';
    uniform (seed1, seed2, rnd);
    wait for delay + (rnd - 0.5) * (10 NS);
    clock <= '1';
    uniform (seed1, seed2, rnd);
    wait for delay + (rnd - 0.5) * (10 NS);
  end loop;
end process;

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