



UiO • **Institutt for informatikk**
Det matematisk-naturvitenskapelige fakultet

IN3160

Combinational logic design (+ Verification)



Messages:

- Python Beta for oblig 1-4 is posted on vortex: <https://www.uio.no/studier/emner/matnat/ifi/IN3160/v23/python-beta/>
 - (some textual benefits not related to python also).
- Implement?
 - Oblig 1, 2
 - yes (follow assignment instructions)
- Demonstrate / show lab supervisor?
 - Yes:
 - Simplifies approval and feedback process
 - If not possible...
 - Video -> filesender. <https://filesender.uio.no/>
 - Can be used to show the same
 - *Feedback in canvas only*
 - *...remember to check...*

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 behavior 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 CL part:

- Know how to create combinational logic (CL)
- What is CL and non combinational logic?
- What is hazards in CL
- How to manage hazards in CL

- Verification

Overview

- What is combinational logic circuits
- CL vs Sequential logic
- What is and how to deal with hazards

Combinational vs Combinatorial

Combinational



combinational logic circuit
combines inputs to generate
an output

Combinatorial



mathematics of counting

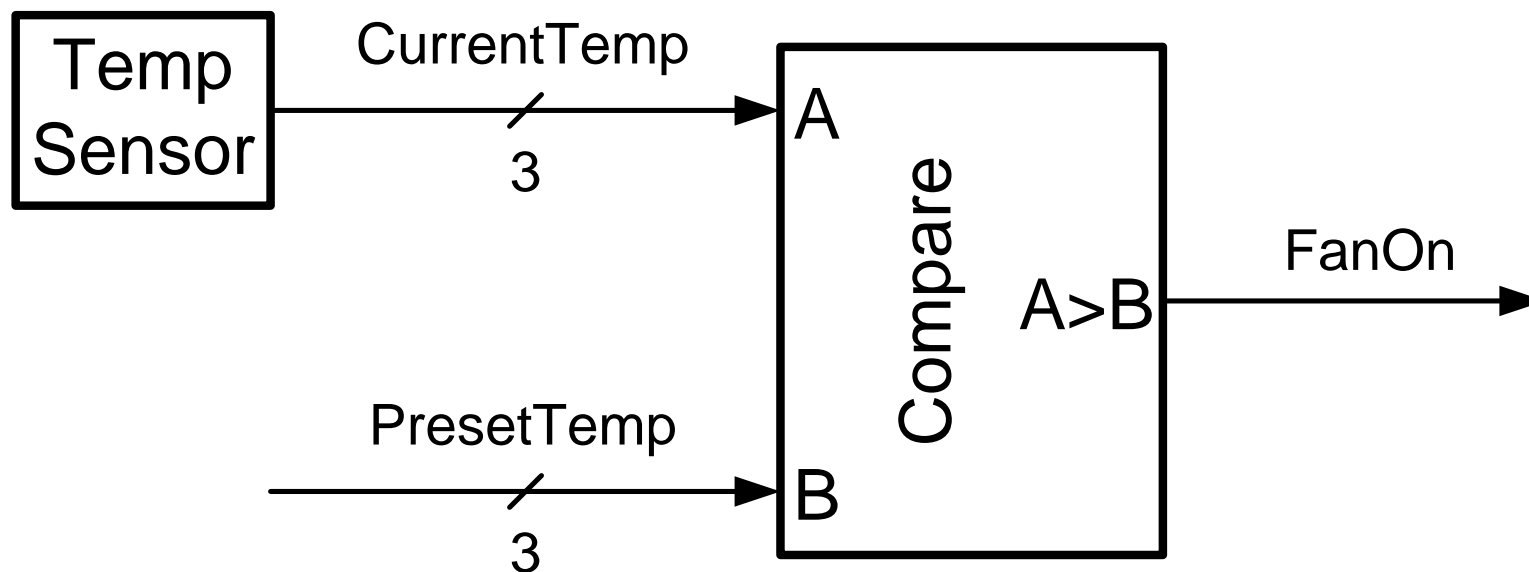
Norsk: Kombinasjonslogikk / kombinatorisk logikk)

Varierende bruk forekommer...

I all hovedsak brukes “kombinatorisk” på norsk..

Combinational Logic Circuit

- Output is a function of current input
- Example – digital thermostat

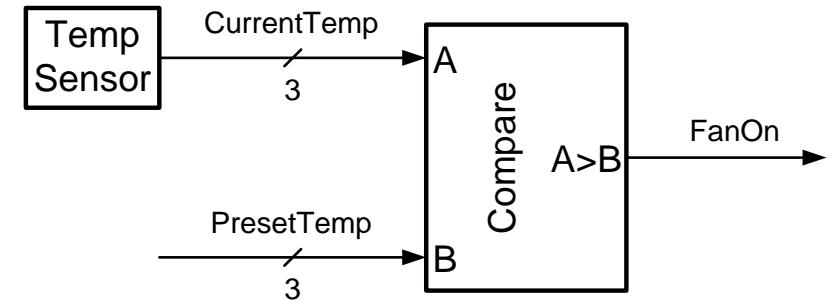


VHDL code

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

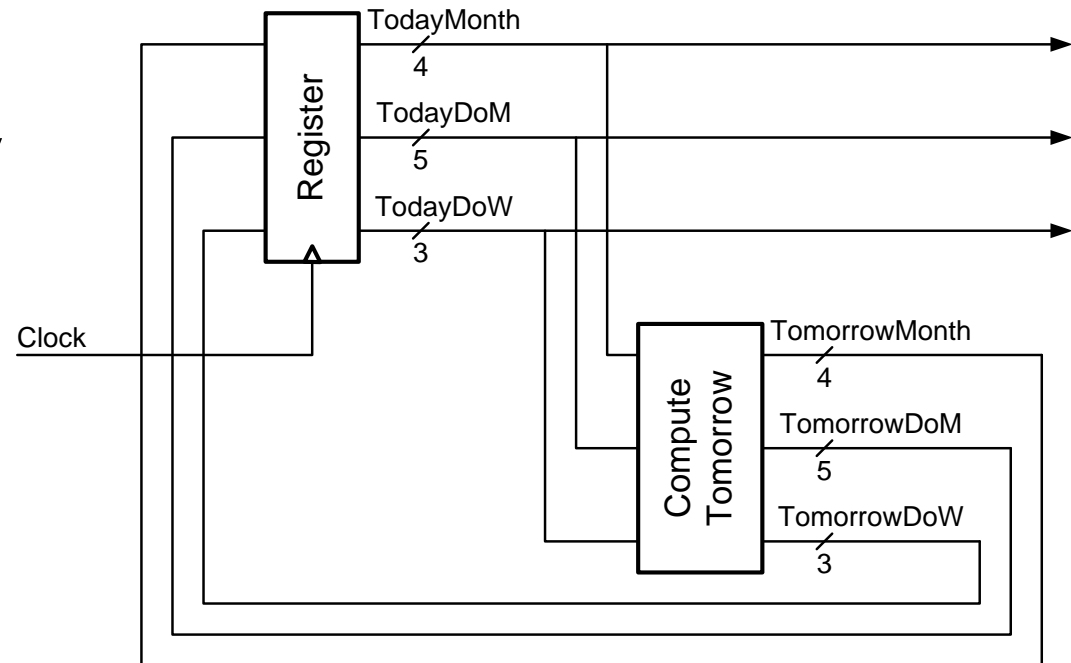
entity compare is
  port(
    current_temp : in  std_logic_vector(2 downto 0);
    preset_temp  : in  std_logic_vector(2 downto 0);
    fan_on       : out std_logic
  );
end entity compare;

architecture combinational of compare is
  -- declarations (none)
begin
  -- statements
  fan_on <= '1' when (current_temp > preset_temp) else '0';
end architecture;
```



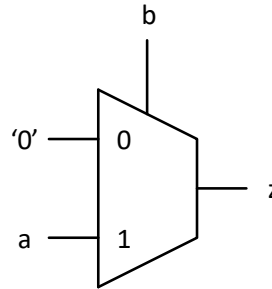
Sequential logic circuit

- Includes state (memory, storage)
- Makes output a function of history as well as current inputs
- Synchronous sequential logic uses a *clock*
- Example: calendar circuit
 - (1 clock / day...)
 - "Compute Tomorrow" is CL
 - Register stores state

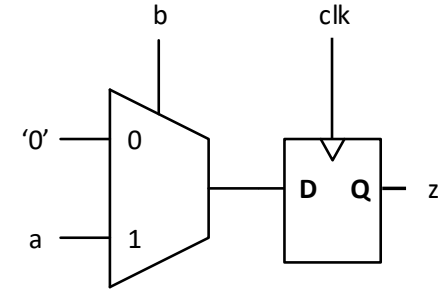


Combinational vs sequential code example

```
COMBINATIONAL: process (all) is  
begin  
  z <= '0';  
  if b then  
    z <= a;  
  end if;  
end process;
```



```
SEQUENTIAL: process (clk) is  
begin  
  if rising_edge(clk) then  
    z <= '0';  
    if b then  
      z <= a;  
    end if;  
  end if;  
end process;
```



-- «all» can be replaced by b here

NOTE:

Using IF, we get latches unless all options are covered.

Here: `z <= '0'` (default value) solves this issue.

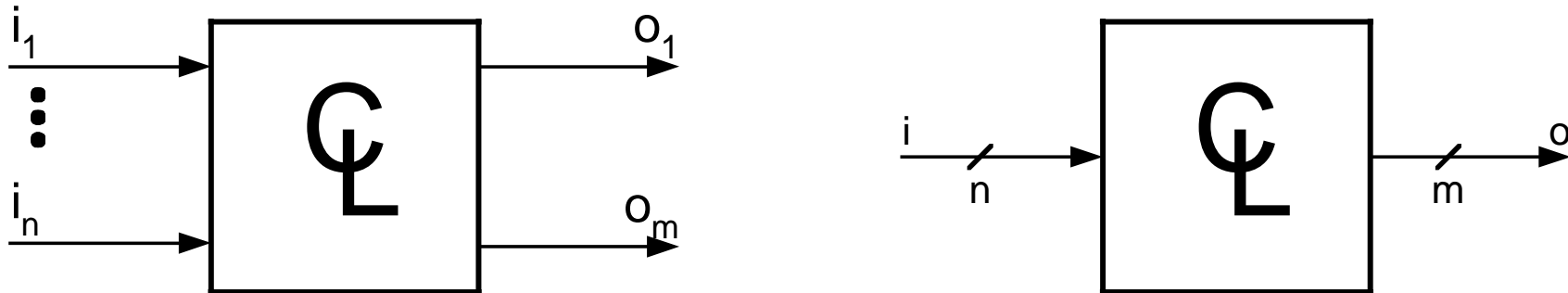
Using `'when-else'` is another option

-- concurrent statement is more compact...

```
z <= a when b else '0';
```

-- quite often we have both reset and clk

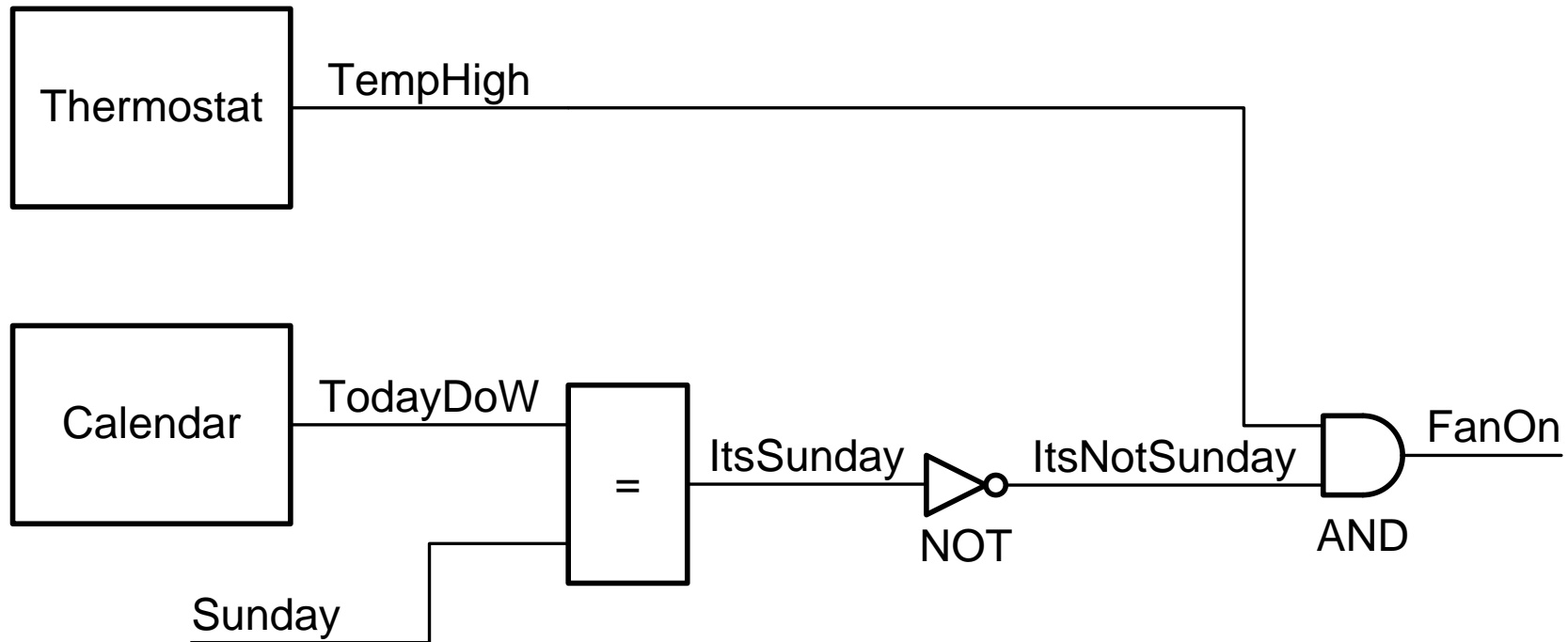
Combinational logic is memoryless



$$\mathbf{o = f(i)}$$

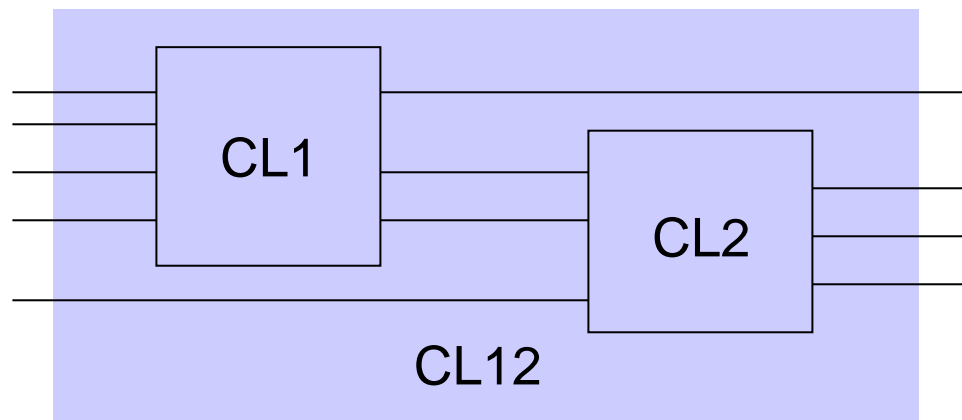
Input determines output

Can compose digital circuits

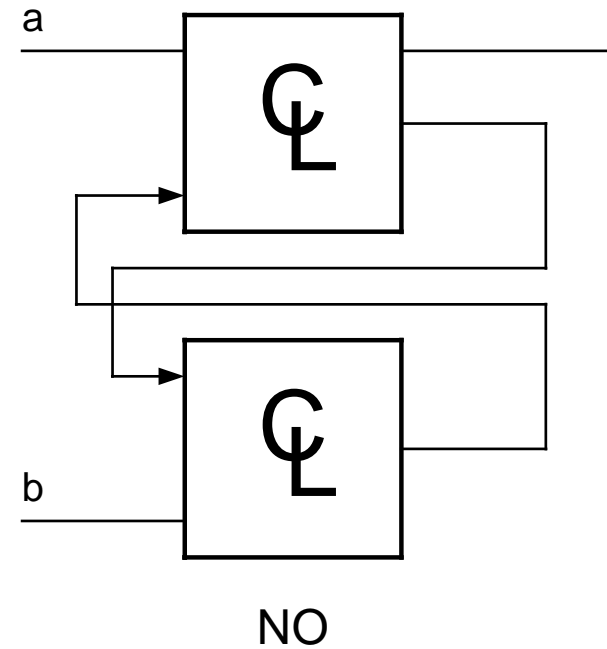
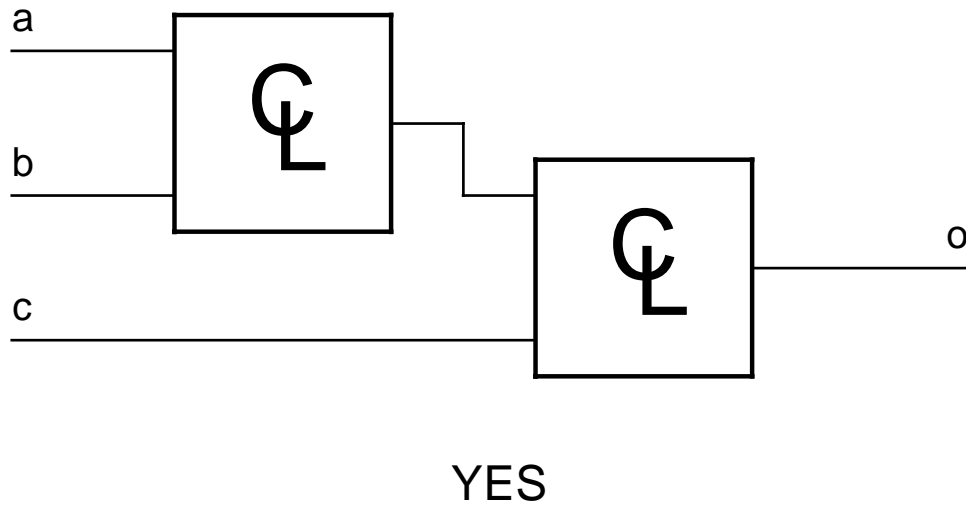


Closure

- "Combinational logic circuits are closed under acyclic composition".



- I.e. As long as there are **no loops**:
 - A module of modules *of combinational logic* is combinational



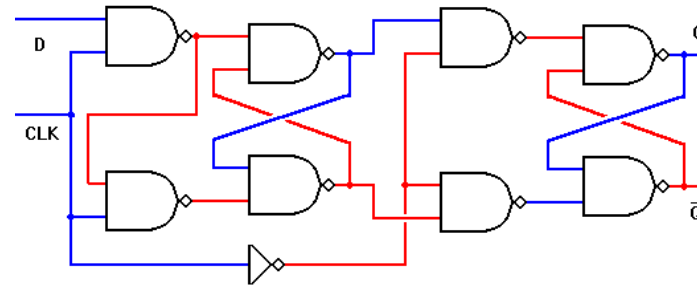
Code that refers back to itself infers latches and is not combinational.

Inferring latches (*not intended as RAM/ROM*) is bad practice, and should be shunned at all costs unless strictly necessary.

If you think you need latches (*rather than FFs*) you most likely should rethink the design...

Non CL example :

- Can be hard to spot in dataflow code
- => Use high level code for *readability (& modifiability)*

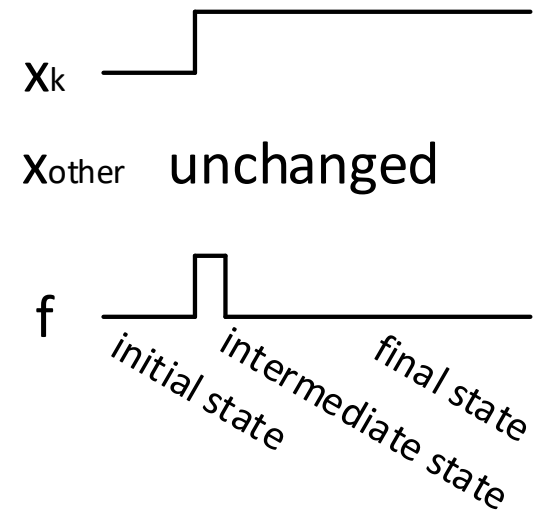
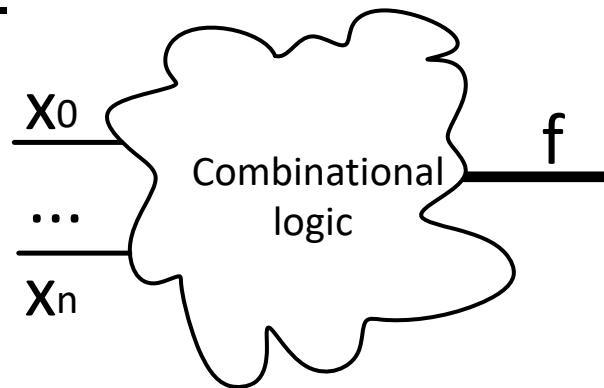


```
architecture data_flow of D_flipflop is
    signal e, f, g, h, i, j, k, l: std_logic;
begin
    -- concurrent statements
    e <= not (D and clk);
    f <= not (e and clk);
    g <= not (e and h);
    h <= not (f and g);
    i <= not (g and not clk);
    j <= not (h and not clk);
    k <= not (l and i);
    l <= not (k and j);
    Q <= k;
end architecture data_flow;
```

Hazards (glitches) in combinatorial circuits

- Definition of hazard in a combinatorial circuit:
 - Output goes through an (unwanted) intermediate state when input changes

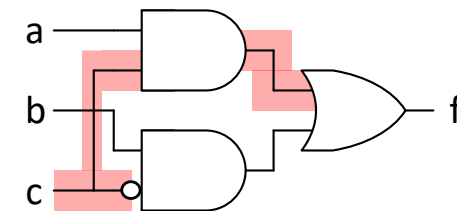
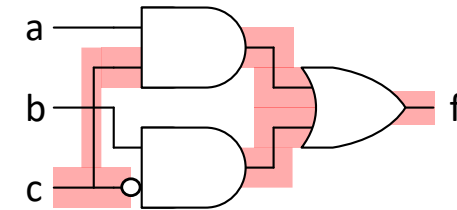
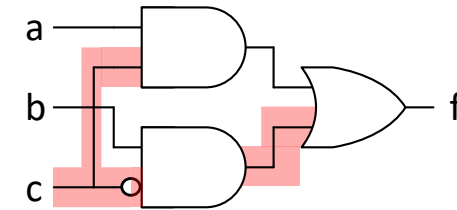
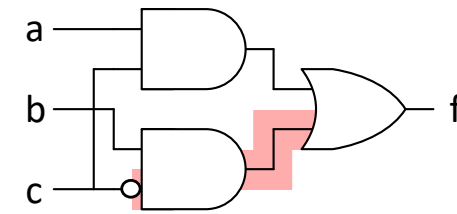
– e.g.



- With several inputs, there can be several unwanted transitions
 - It doesn't have to be $0 \ 1 \ 0$, it can be $\mathbf{X} \rightarrow \mathbf{Y} \rightarrow \mathbf{Z}$ or $\mathbf{X} \rightarrow \mathbf{Y}_1 \rightarrow \dots \rightarrow \mathbf{Y}_n \rightarrow \mathbf{Z}$

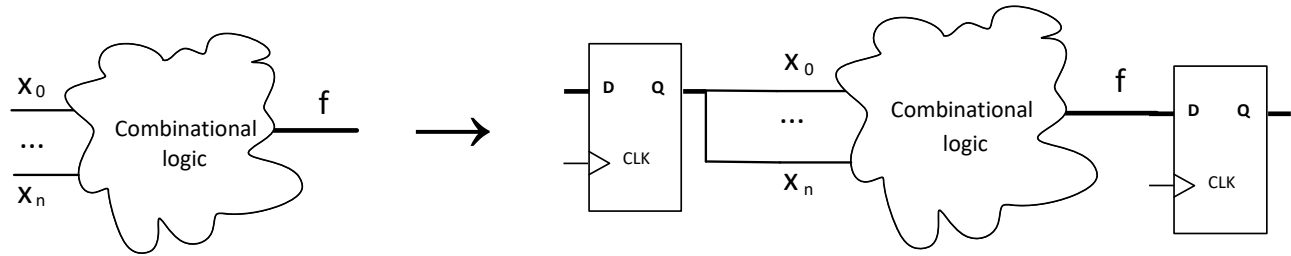
Hazards in combinatorial design

- Ex: $f(a,b,c) = (a \wedge c) \vee (b \wedge c')$
 $f \leq (a \text{ and } c) \text{ or } (b \text{ and not } c);$
- $a = '1', b = '1', c$ changes from '1' to '0'
- f goes from 1 to 0 to 1
 (the inverted input of the second and-gate..)
- Possible solutions: (next page)



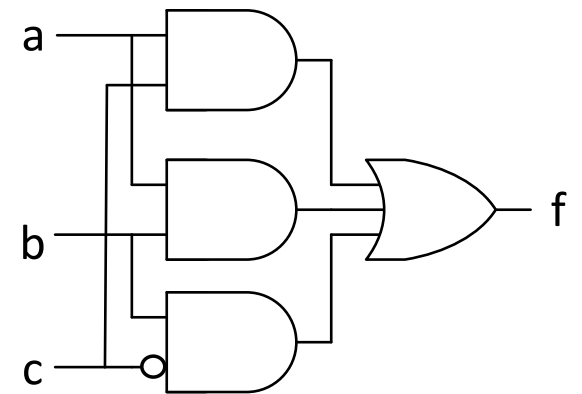
Solutions

- 1: add registers... (we'll get back to this one in oblig 8)
 - This is what we normally do..
 - Left => stable input
 - Right => stable output



- 2: Manually design a solution
 - D&H goes through that process
 - Laborious process: not a topic for this course

- 3: (Use high level code!)
 - **may not solve every possible issue, but**
 - will not induce issues if the synthesizer is capable
 - Synthesizing for FPGAs, = LUTs (problem occurs first at > 4 inputs)



Designer vs tool- example

- $F(d,c,b,a)$ is true if input d,c,b,a is prime

$$f = \sum_{dcba} m(1,2,3,5,7,11,13)$$

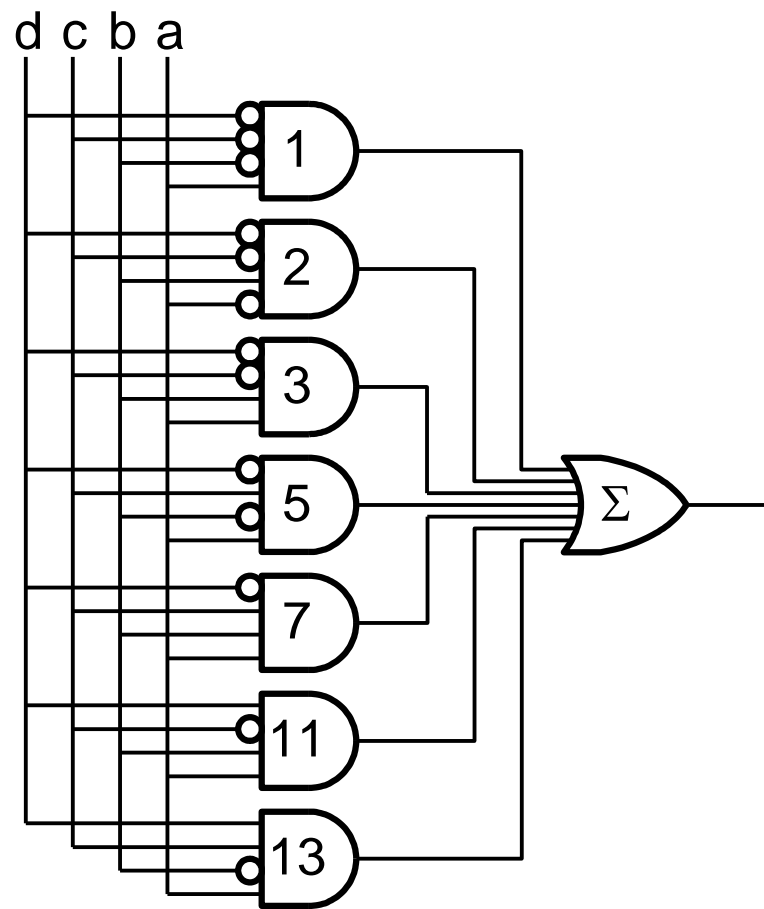
No	dcba	q
0	0000	0
1	0001	1
2	0010	1
3	0011	1
4	0100	0
5	0101	1
6	0110	0
7	0111	1
8	1000	0
9	1001	0
10	1010	0
11	1011	1
12	1100	0
13	1101	1
14	1110	0
15	1111	0

Schematic Logic Diagram

Equation:

$$f = \sum_{dcba} m(1,2,3,5,7,11,13)$$

Schematic Logic Diagram:

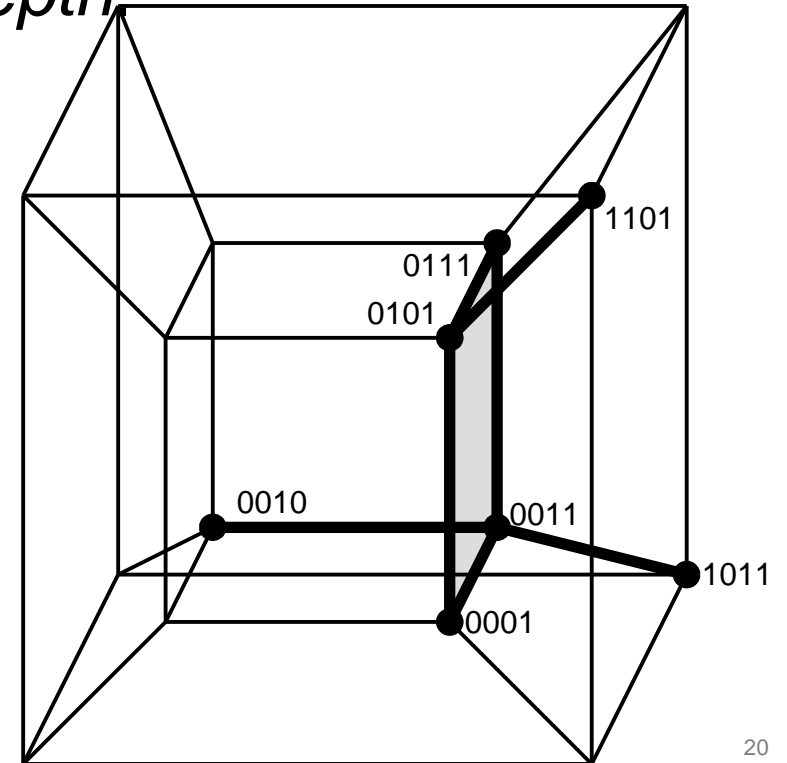
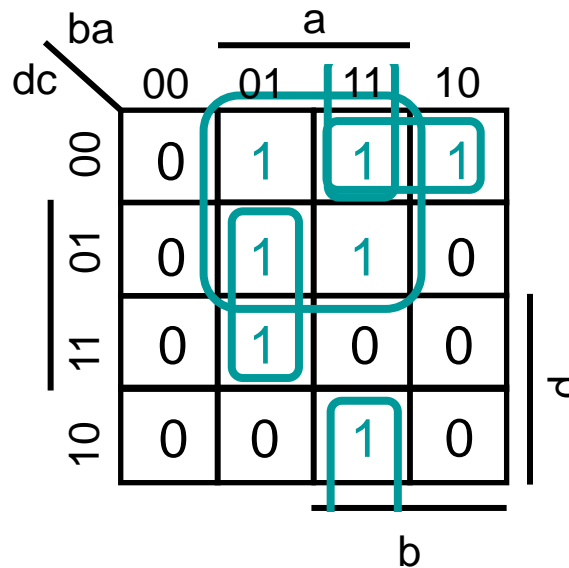


Manual optimization

- Minimalistic and Hazard free implementations can be found using implicant cubes and Karnaugh diagrams
- *Method is laborious and can normally be skipped entirely.*
- D&H covers this in 6.4-6.9, *we will not go in-depth*

$$f = \sum_{dcba} m(1,2,3,5,7,11,13)$$

Code will have to be written as a dataflow or structural design.



VHDL that implement the prime function (F)

- Note that these do not address any hazard issue.

VHDL Solution using case

$$f = \sum_{dcba} m(1,2,3,5,7,11,13)$$

```

library IEEE;
use IEEE.std_logic_1164.all;

entity prime is
port(
input:  in  std_logic_vector(3 downto 0);
isprime: out std_logic
);
end entity prime;

architecture case_impl of prime is begin
process(input) begin
case input is
when x"1" | x"2" | x"3" | x"5" | x"7" | x"b" | x"d" => isprime <= '1';
when others => isprime <= '0';
end case;
end process;
end case_impl;

```

The vertical bar '|' can be used to list multiple choices

No	dcba	q
0	0000	0
1	0001	1
2	0010	1
3	0011	1
4	0100	0
5	0101	1
6	0110	0
7	0111	1
8	1000	0
9	1001	0
10	1010	0
11	1011	1
12	1100	0
13	1101	1
14	1110	0
15	1111	0

Solution using «Matching case» = case?

$$f = \sum_{dcba} m(1,2,3,5,7,11,13)$$

```
library ... (same as previous slide)
```

```
entity ...
```

```
architecture mcase_impl of prime is
```

```
begin
```

```
  process(all) begin
```

```
    case? input is
```

```
      when "0--1" => isprime <= '1';
```

```
      when "0010" => isprime <= '1';
```

```
      when "1011" => isprime <= '1';
```

```
      when "1101" => isprime <= '1';
```

```
      when others => isprime <= '0';
```

```
    end case?;
```

```
  end process;
```

```
end mcase_impl;
```

Matching case can be used with '-'
('-' = don't-care bit)

Note:

Each option should only be listed once

No	dcba	q
0	0000	0
1	0001	1
2	0010	1
3	0011	1
4	0100	0
5	0101	1
6	0110	0
7	0111	1
8	1000	0
9	1001	0
10	1010	0
11	1011	1
12	1100	0
13	1101	1
14	1110	0
15	1111	0

More on case and case? «matching case»

- Case requires all possible outcomes to be defined
 - «when others»
 - Will cover other outcomes, but may also cover errors
 - I.e: we added a subtype to a type, and should extend a case...
- If you have many options that do the same
 - Use «matching case» case?
 - Allows for don't cares to cover options with the same outcome.

```
type holiday is (Xmas, easter, summer);  
signal min_holiday: holiday;  
type temperature is (freezing, cold, mild, warm);  
signal weather : temperature;
```

```
...
```

```
case my_holiday is  
  when Xmas    => weather <= freezing;  
  when easter  => weather <= cold;  
  when others => weather <= warm;  
end case;
```

```
-- add 'autumn' to holiday type..  
-- will compilation bug you?  
-- should autumn be considered warm..?
```


$$f = \sum_{dcba} m(1,2,3,5,7,11,13)$$

Solutions using concurrent signal assignment

```
architecture dataflow of prime is
begin
  isprime <=
    (input(0) and (not input(3))) or
    (input(1) and (not input(2)) and (not input(3))) or
    (input(0) and (not input(1)) and input(2)) or
    (input(0) and input(1) and not input(2));
end architecture dataflow;
```

```
architecture selected of prime is
begin
  with input select isprime <=
    '1' when x"1" | x"2" | x"3" | x"5" |x"7" | d"11" | x"d",
    '0' when others;
end architecture selected;
```



Avoid pure dataflow unless strictly necessary



Selected statements will not infer latches unless explicitly designed for that purpose

No	dcba	q
0	0000	0
1	0001	1
2	0010	1
3	0011	1
4	0100	0
5	0101	1
6	0110	0
7	0111	1
8	1000	0
9	1001	0
10	1010	0
11	1011	1
12	1100	0
13	1101	1
14	1110	0
15	1111	0

Which one would you prefer reading?



UiO • **Institutt for informatikk**
Det matematisk-naturvitenskapelige fakultet

IN3160, IN4160

Verification part 1

Yngve Hafting



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

After completion of the course you will:

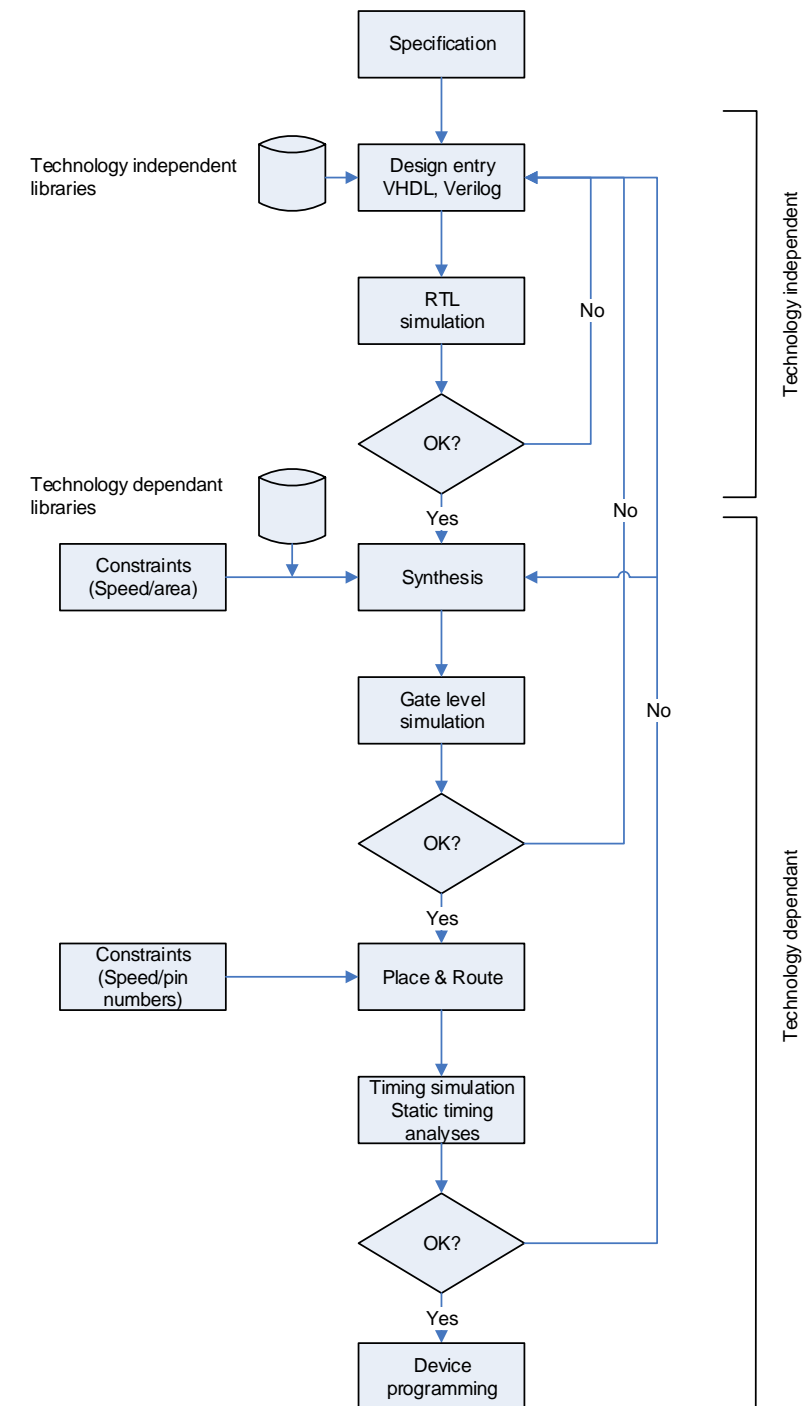
- understand important **principles for design and testing** of digital systems
- understand the relationship between behaviour and different construction criteria
- be able to describe advanced digital systems at different levels of detail
- **be able to perform simulation and synthesis of digital systems.**

Goals for this lesson:

- Know what we mean by ‘verification’ and ‘test’
 - Functional verification
 - Formal verification
 - Compilation
 - Simulation
 - Coverage
- Know how to perform verification
 - Manual stimuli
 - Script based stimuli
 - **Test benches**
- Know basic simulation principles for digital systems
 - Know how event based simulation works
 - Know the difference between event based and cycle based simulation.
- Know how basic VHDL structures will be simulated
 - Compilation steps
 - Process sensitivity list

Overview

- What is- & Why verification
- Verification vs Testing
- Coverage
- Compilation
- Simulation
 - Types
 - RTL (functional)
 - Timing
 - Static timing analysis
 - dynamic
 - Execution
 - Cycle based
 - Event based
- Assignments and suggested reading



Verification vs. Testing..

Verification	Testing
Ensures that the <i>design</i> meets the specification	Ensures that <i>one particular device</i> actually works- has no faulty gates or other production errors
Is done <i>throughout</i> the design process	Is performed after manufacturing process - Before shipping the product to the customer
Consists of tests that ensure that the design works <ul style="list-style-type: none">• Logically<ul style="list-style-type: none">• Compilation• RTL simulation• Timing<ul style="list-style-type: none">• Static timing analysis• Timing simulation	Consists of tests that each product works <ul style="list-style-type: none">• Electrical tests• Built in self tests<ul style="list-style-type: none">• At start up or user initiated• RAM tests etc.

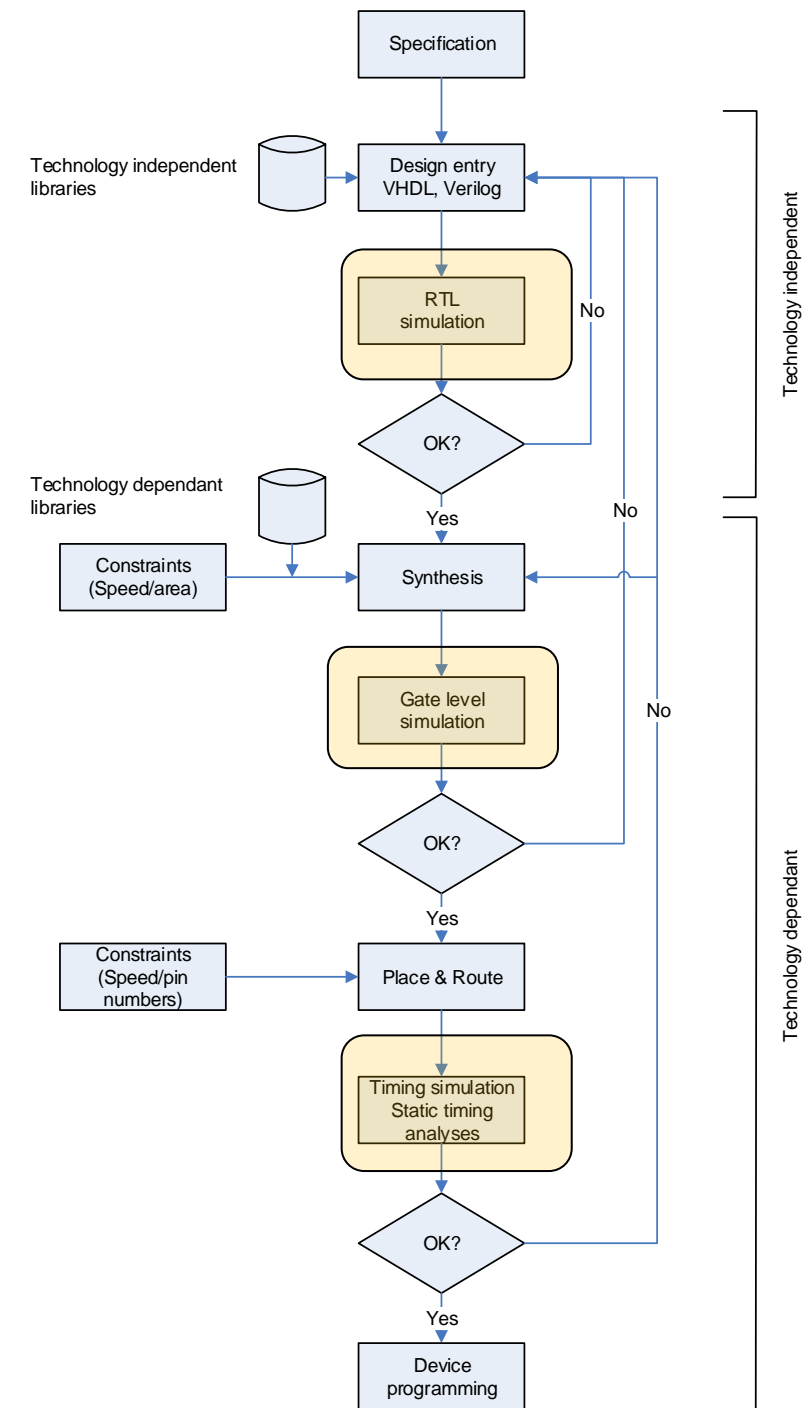
We can *design for testing*-

but when we start *testing*,

- we should already know that the *design* works

What is verification?

- **Compilation**
 - Ensures that the code can be implemented.
- **Formal verification**
 - To *prove* that a module has a certain function
 - Mathematical equivalence (requires mathematical proof)
 - Model check: Checks the state space of a system to test if certain assertions are true
 - = Check if we can set the system in invalid states...
- **Functional verification**
 - To verify that the code behaves as intended
 - *RTL simulation*
 - Checks that the code provides correct output
 - *Gate level simulation* (Post synthesis simulation)
 - *Simple timing tests (fast)*
 - *Timing simulation* (Post PAR simulation)
 - Static timing analysis
 - Critical path analysis
 - Check that we meet setup and hold requirements from interacting devices.
 - Bus functional models, BFM
 - Check that our device can communicate correctly on a bus
 - Can provide both input and provide response for “other devices” on a bus
 - IP or self made...



Coverage

- Will we be able to cover all states and possibilities for our design?
 - Likely not.
 - A formal verification process may ensure this, but is usually not possible.
- *Specification coverage*
 - What is the proportion of the specification that we test?
 - 100% of features usually is the goal
- *Code coverage*
 - Usually all our code lines should be tested.
 - If not- do you really need that code...?
- Test patterns (*Data coverage*)
 - 100% **never possible** (e.g. 64 bit adder 2^{128} possible patterns...)
 - Special cases
 - Randomized tests

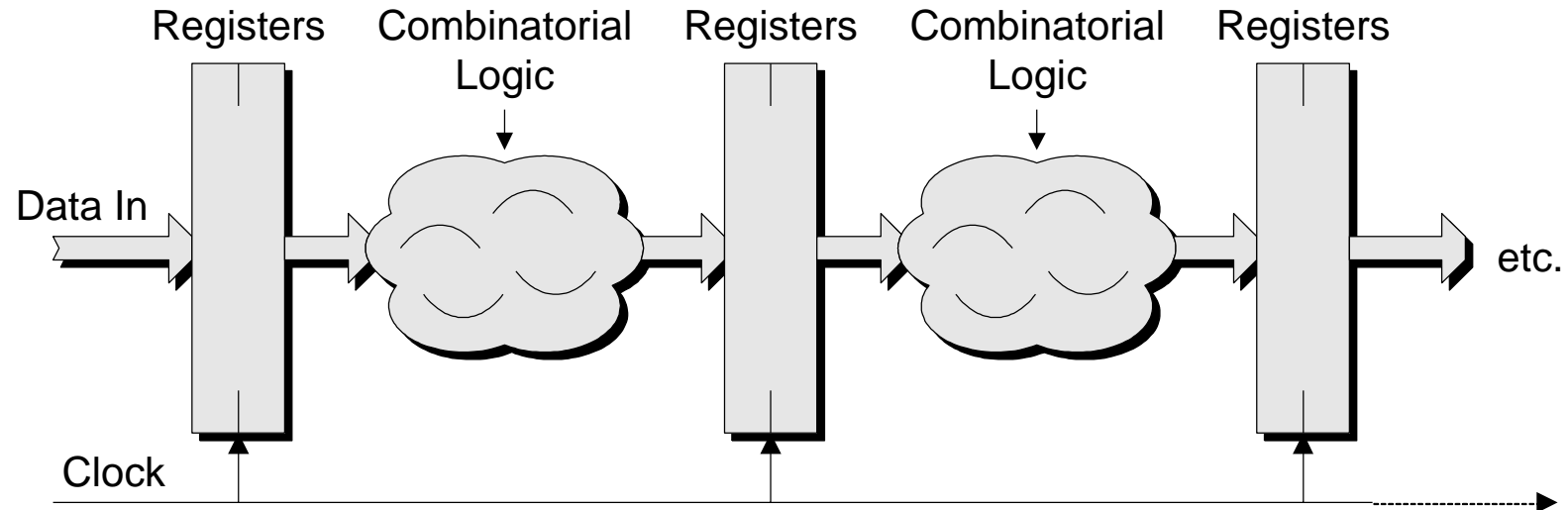
Compilation: Analysis and Elaboration

- Analysis
 - **The compiler reads all the files, check syntax, semantics**
 - Compiled result is
 - translated to an intermediate representation
 - stored in (work) library
- Elaboration (requires error free analysis)
 - **Creates design hierarchy**
 - Instantiates entities
 - Creates connections
 - Checks that types does match for connected signals

Simulation

- The Simulator
 - knows all signal dependencies
 - *Unless there are errors in signal sensitivity lists.*
 - keeps track of all signal values
 - Both external and internal to the design
 - has an event queue
 - Every change in a signal is an event scheduled at a specific time step.
 - Events may be queued to happen at the same simulation step
 - But they are resolved one by one.
- Simulation types (upcoming slides)
 - Cycle based
 - Event based

Cycle based simulation



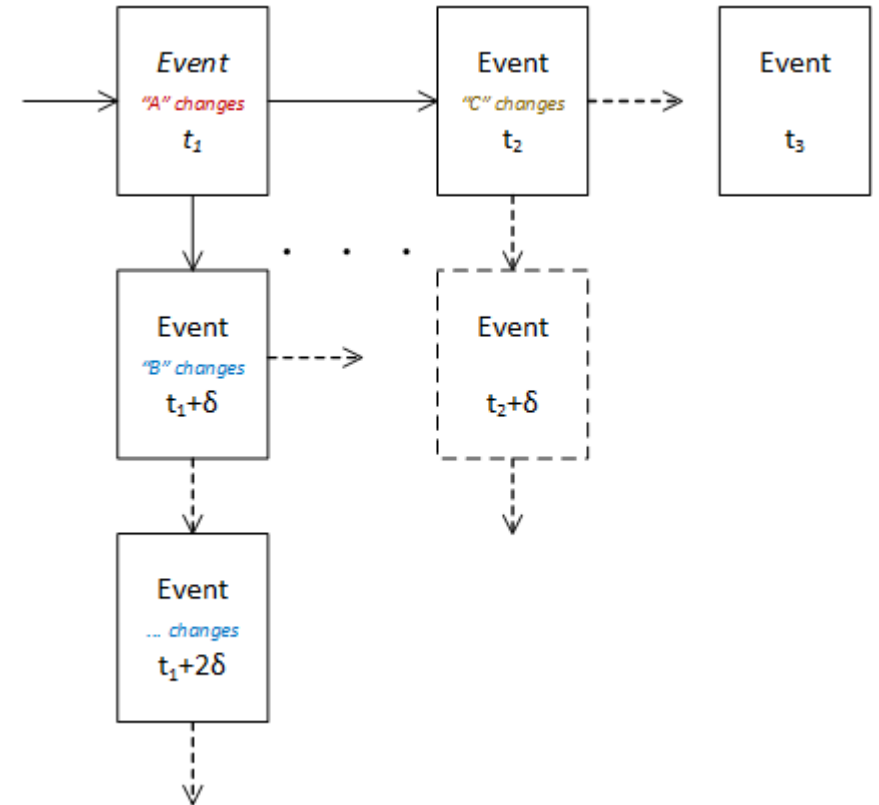
- No notion of time within a single clock cycle
- Only evaluate boolean functions for each component once
- Very fast, but *can produce simulation errors*
- Not widely used, but can be used in some parts of designs with high simulation times
 - This is *almost* what we get when using "clk" as sensitivity in a process

Event driven simulation

- «event driven» => time is driven by events
 - Change in inputs (stimuli)
 - Change in outputs that propagate to other changes
- All signal drivers are modelled with a delay called «delta delay»
 - A delta delay is a delay of 0 time-
 - a mechanism for queuing of events

Event driven simulation

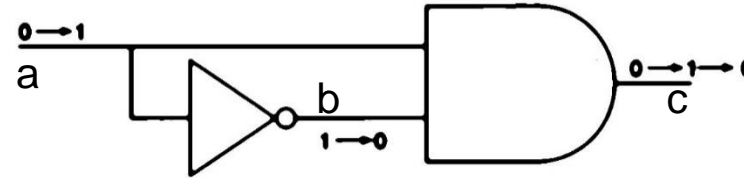
- An event occurs in a time-step when a signal changes (Ex. an input is set from the testbench)
 - All signals that depend on the signal is evaluated
 - Changes are put in the event queue
 - When timing information is provided (post synthesis)
 - » Timing delays are used to schedule events
 - With no timing information (RTL-simulation)
 - » Output is queued at the same time-step
 - » A delay of 0 time is called a delta delay (δ)
 - All events in the queue for a time step is evaluated
 - Until there are no more changes left
 - The simulation proceeds to the next time-step when there are no more events to be evaluated
 - In RTL simulation, FF outputs should only be evaluated when the clock edge occurs
 - Event driven simulation ensures all simulators get the same result.



Event queues and delta delay example

```

15:  STIMULI:
16:  process
17:  begin
18:    loop
19:      a <= '0';
20:      wait for 100 ns;
21:      a <= '1';
22:      wait for 100 ns;
23:    end loop;
24:  end process;
25:
26:  EKS1:
27:  process (a,b)
28:  begin
29:    b <= not a;
30:    c <= b and a;
31:  end process;
    
```



IN a physical circuit
 We *may* see such glitches

Glitches can be hidden in waveform diagrams

Post synthesis- will much more likely show these type of effects than RTL-simulation

ps	δ	a	b	c
	0	0	0	0
	0	0	1	0
100000	+1	1	1	0
100000	+2	1	0	1
100000	+3	1	0	0
200000	+1	0	0	0
200000	+2	0	1	0
300000	+1	1	1	0
300000	+2	1	0	1
300000	+3	1	0	0
400000	+1	0	0	0
400000	+2	0	1	0

RTL simulation practical example

- Modelsim / Questa:
 - Creating test benches
 - **Example (tb_xor.vhd)**
 - **See ../verification...**

Testbench example

```

library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.numeric_std.all;

entity tb_xor is
  -- Empty entity of the testbench
end entity tb_xor;

architecture behavioral of tb_xor is
  component X_OR is
    generic( WIDTH : natural);
    port(
      input : in std_logic_vector(width-1 downto 0);
      output : out std_logic
    );
  end component;

  signal a, b, c : std_logic;
  signal d : std_logic_vector(4 downto 0);
  signal e : std_logic;

begin
  TEST_UNIT_1 : x_or
  generic map(
    WIDTH => 2 -- 2 port XOR
  ) port map(
    input(0) => a,
    input(1) => b,
    output => c
  );

```

```

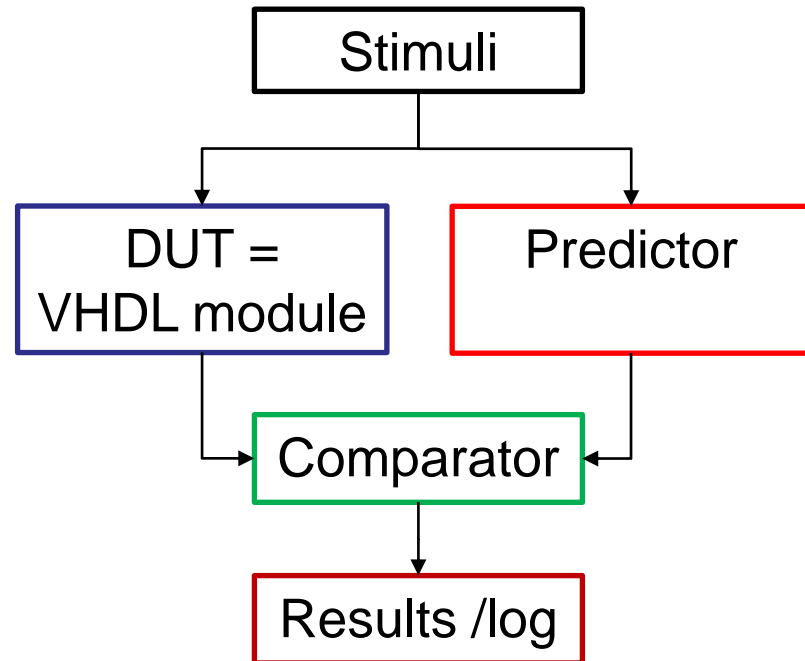
TEST_UNIT_2 : x_or
generic map(
  WIDTH => 5 -- 5 port XOR
) port map(
  input => d,
  output => e
);

-- generate test vectors --
main: process is
  variable abd: std_logic_vector(d'range);
begin
  wait for 1 ns;
  for i in 0 to 24 loop
    abd := std_logic_vector( to_unsigned(i, abd'left+1) );
    d <= abd;
    a <= abd(0);
    b <= abd(1);
    wait for 1 ns;
    assert (e = c) report
      ( "e differs from c for input = " & integer'image(i) )
      severity error;
    -- assert (e = xor(d)) report( "e is not the even parity
of d for " & integer'image(i) ) severity error;
  end loop;
  report ("TESTING FINISHED!");
  std.env.stop;
end process;

end architecture behavioral;

```

General testbench layout



- **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.

Cocotb testbench

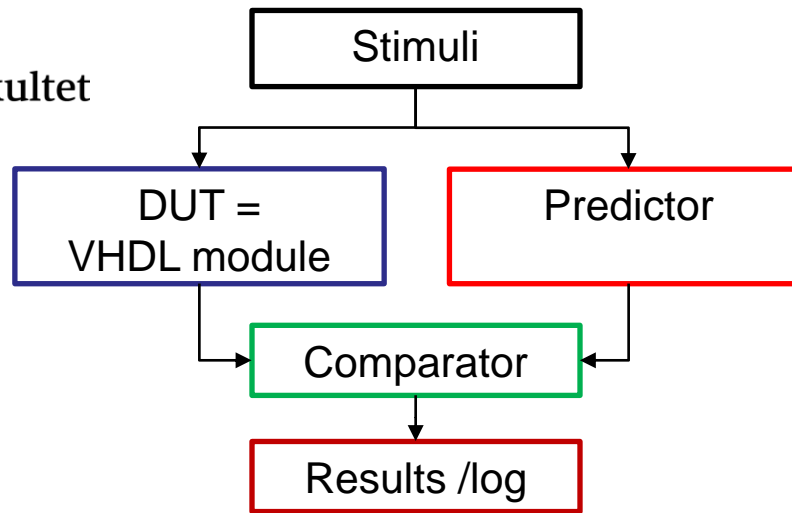
```
import cocotb
from cocotb import start_soon
from cocotb.triggers import Edge, Timer, ReadOnly
import random

# bitwise XOR of input (a subroutine for predictor)
def xor(input):
    result = 0
    for i in range(input.n_bits):
        result = result^(input & 1)
        input = input >> 1
    return result

# Predicts or calculates what the output should be
def predictor(dut):
    return xor(dut.input.value)

# Compares simulated output with predicted output
async def compare(dut):
    while 1:
        # Test on new input, then let output settle.
        await Edge(dut.input)
        await ReadOnly()
        assert dut.output == predictor(dut), (
            "output ({out}) is not as predicted: XOR({inp})"
            .format(out=dut.output.value, inp=dut.input.value))

    dut._log.info("output ({out}) is XOR({inp})"
        .format(out=dut.output.value, inp=dut.input.value))
```



```
# Sets stimuli-data in DUT
async def set_stimuli(dut, vector):
    dut.input.value = vector
    await Timer(1, units= 'ns')

# Generate all data
async def stimuli_generator(dut):
    #for i in range( 2**dut.input.value.n_bits):
    for i in range( 2**len(dut.input)):
        await start_soon(set_stimuli(dut, i))

@cocotb.test()
async def main_test(dut):

    """Try accessing the design."""
    dut._log.info("Running test...")
    start_soon(compare(dut))
    await start_soon(stimuli_generator(dut))

    dut._log.info("Running test...done")
```

```
# Makefile

# defaults
SIM ?= ghdl
TOPLEVEL_LANG ?= vhdl

# VHDL 2008
EXTRA_ARGS +=--std=08

# TOPLEVEL is the name of the
# toplevel module in your VHDL file
TOPLEVEL ?= x_or

#VHDL_SOURCES +=
# $(PWD)/../src/$(TOPLEVEL).vhd
VHDL_SOURCES += $(PWD)/../src/*.vhd*

# SIM_ARGS is Simulation arguments.
# --wave determines name and type of
# waveform
SIM_ARGS +=--wave=$(TOPLEVEL).ghw

# -g<GENERIC> is used to set generics
# defined in the toplevel entity
SIM_ARGS +=-gWIDTH=5

# MODULE is the basename of the
# Python test file
MODULE ?= tb_xor

# include cocotb's make rules to
# take care of the simulator setup
include $(shell cocotb-config --
makefiles)/Makefile.sim

# removing generated binary of top
# entity and .o-file on make clean
clean::
    -@rm -f $(TOPLEVEL)
    -@rm -f e~$(TOPLEVEL).o
```

VHDL Testbench

- Stimuli generator and test environment in VHDL
- Powerful possibilities for simulation
 - File I/O
 - Reading test patterns from file
 - Writing results to file and compare it to a file with the correct answers
 - The file with the correct answers can also be read, and comparison between the result and the blueprint can be executed in the testbench
 - Can build in modules for surrounding circuits
 - Especially important if we have a two-way communication between UUT (Unit Under Test) and surrounding circuits (Handshake signals)
- Gives simulator independent testbench.

My errors will be found...



In the design phase



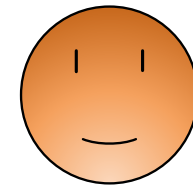
In the conceptual review



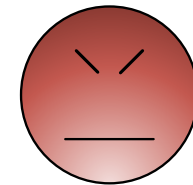
While coding



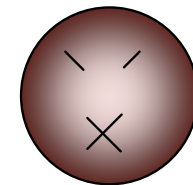
When compiling



When simulating



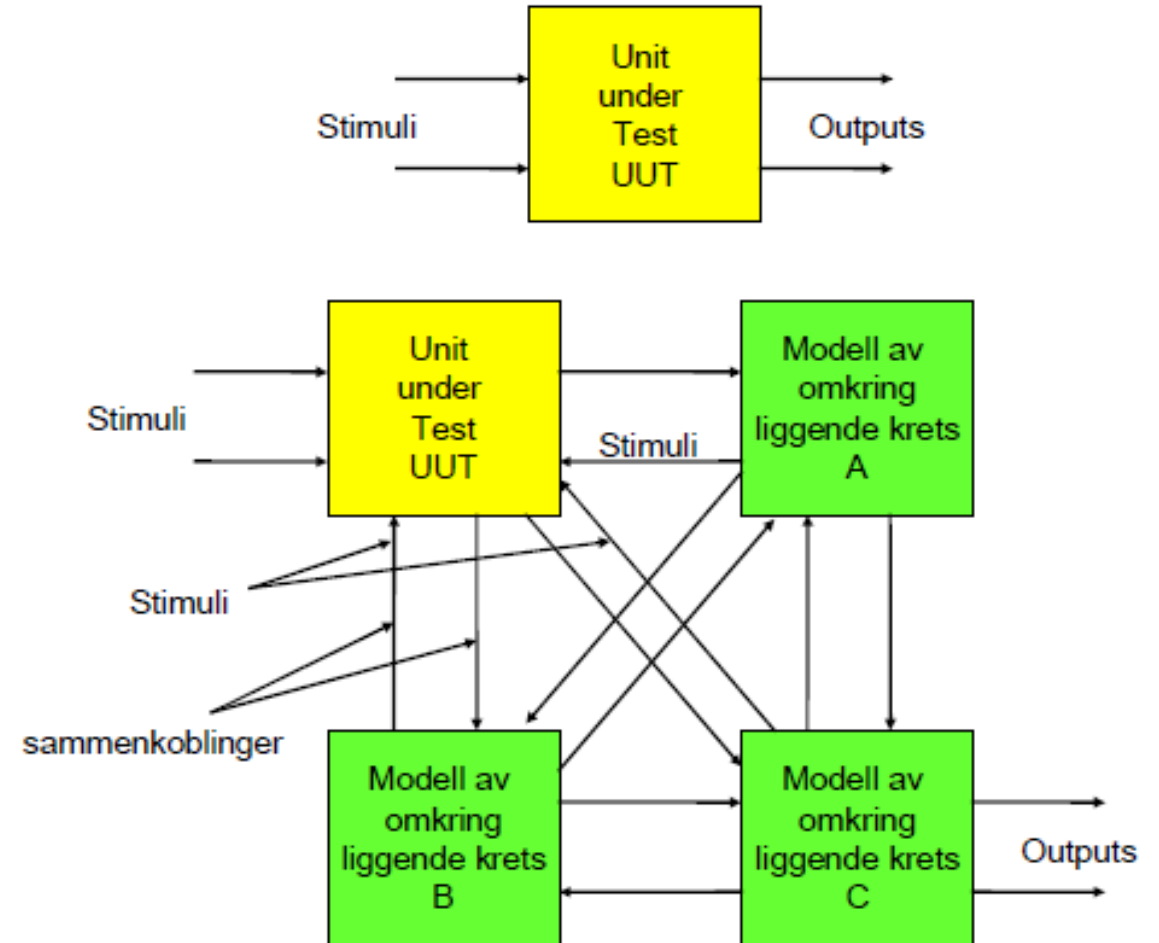
During testing



By the customer

Testbench

- In a test bench you can
 - include other modules (that are more or less verified),
 - or provide data for one unit at a time.
- Often we rely on simulation models created by others.
 - Bus functional models etc.
 - *Can be required to achieve certifications in industry..*



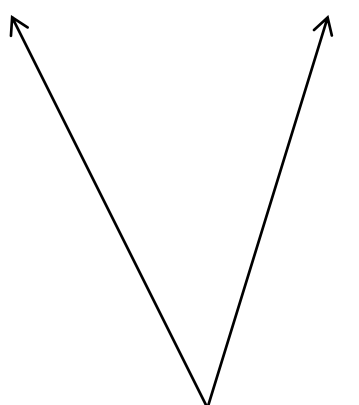
VHDL Testbenches

- Libraries
- Empty entity (can have generics, but no IO)
- Component declaration for each entity that is a part of the test
- Signals for all ports we would like to manipulate
- Component instantiation
 - («DUT» is likely the module we would like to test)
- One or more processes that
 - set input test vectors at specific time intervals
 - evaluates output vectors
 - reports findings and results to screen or file.

Simulation of VHDL models

- The *time* datatype is defined in std.vhd
- The function *now* returns current simulation time

```
type time is range -2147483647 to 2147483647
  units
    fs;
    ps = 1000 fs;
    ns = 1000 ps;
    us = 1000 ns;
    ms = 1000 us;
    sec = 1000 ms;
    min = 60 sec;
    hr = 60 min;
  end units;
```



Implementation specific

VHDL simulation cycle I

- VHDL simulation has two phases:
 - Initialization phase
 - Repeating execution of the simulation cycle
- Simulation starts at time 0
- Current simulation time, T_c
- Next simulation time, T_n
- T_n is calculated from the earliest occurrence of:
 - TIME'high (max simulation time)
 - Next time a driver is activated
 - Next time a process starts (continues)

VHDL simulation cycle II

- It is assumed that all signals have had their value forever at the start of simulation
 - Signals are initialized to 'U' (undefined) unless otherwise specified.
- Each simulation cycle is executed by:
 - T_c is set to T_n
 - All explicit assignments (input stimuli) and all implicit signals are updated. Both of these can cause new events, either a delta delay ahead or at another time.
 - If $T_n = T_c$, then a delta cycle is the next cycle

VHDL processes

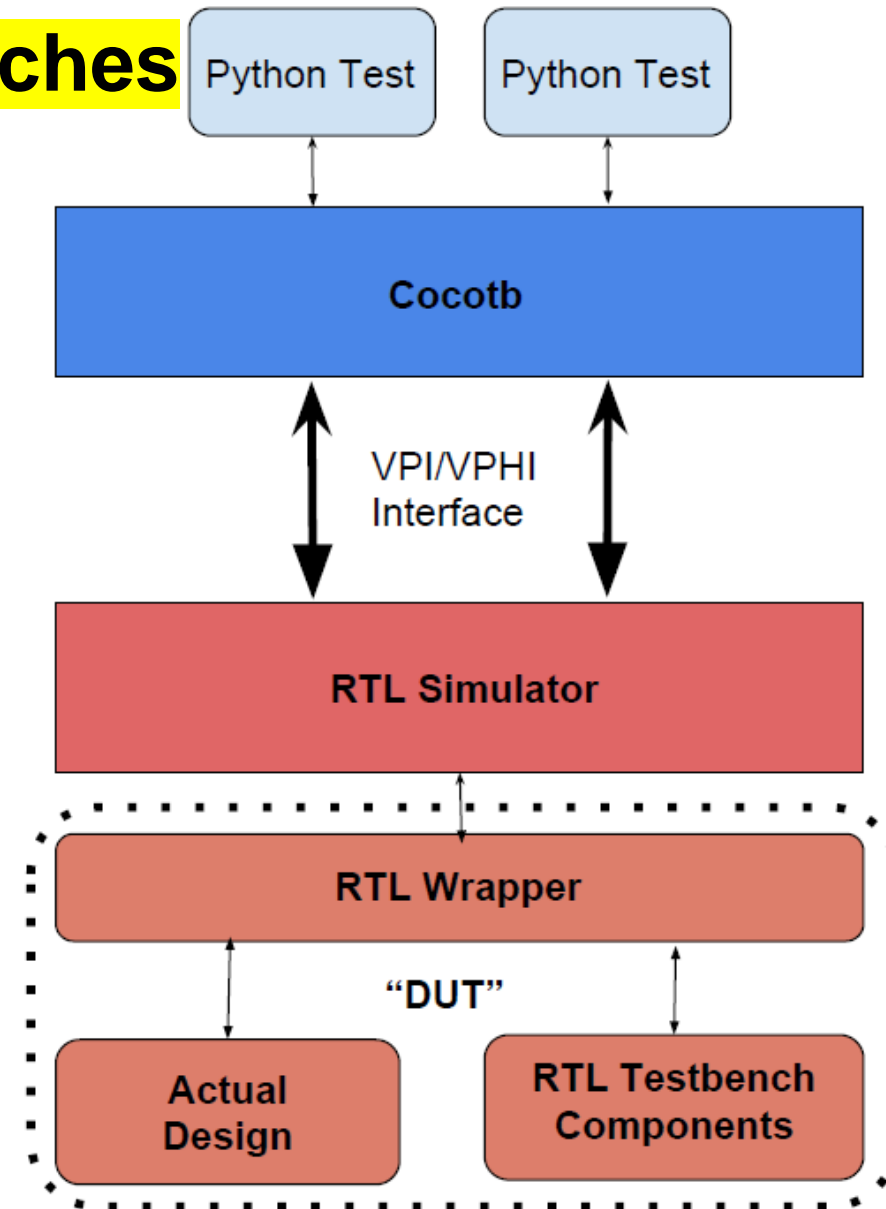
- Process sensitivity
 - Decides when a process is invoked in *simulation*
 - «**SHOULD not**» **interfer with how HW is made...**
 - *Do not trust this..!*
 - Good practice:
 - Use keyword `all` for combinational logic
 - Use clock (and reset when asynchronous) for sequential logic.

Simulation specific code (Non synthesis)

- **Assertions** (*will be ignored by synthesis tools*)
 - Can be used to create error messages and notifications based on results
 - Ex: `assert (e = c) report("e differs from c") severity error`
 - Read «assert» as «if not <boolean expression> then» [report...]
- **Wait for** <time> (*will be ignored by synthesis tools*)
- **Warning...:**
 - «`wait for / wait on` <signal>» *can* be used in synthesizable code
 - Makes sequential (latched or flip-flopped) logic.
 - Better: use the IEEE1164 keyword «`rising_edge`» or «`falling_edge`» to ensure operation

Cosimulation: Cocotb and python testbenches

- *Cosimulation*: Design and testbench simulated independently
- Communication through VPI/VHPI interfaces, represented by cocotb "triggers".
- When the Python code is executing, **simulation time is not advancing**.
- When a trigger is awaited, the testbench waits until the triggered condition is satisfied before resuming execution.
- Available triggers include:
 - Timer(time, unit): waits for a certain amount of simulation time to pass.
 - Edge(signal): waits for a signal to change state (rising or falling edge).
 - RisingEdge(signal): waits for the rising edge of a signal.
 - FallingEdge(signal): waits for the falling edge of a signal.
 - ClockCycles(signal, num): waits for some number of clocks (transitions from 0 to 1).



Cocotb: Coroutines, tasks and triggers

- All signals in the design hierarchy can be probed and set in python
- "async def" is used when defining coroutines
- Multiple triggers can be used
 - enable tests running independently
 - cocotb.start_soon(<coroutine>)
 - Starts the coroutine as soon when "awaiting" the next time
 - Used to start clock generation,
 - await(<task/trigger>) <https://docs.python.org/3/library/asyncio-task.html>
 - Waits until the task is finished or trigger condition occurs
 - await ReadOnly() is used to let signals settle after other triggers such as await Edge(<dut.signal>)
 - » You do not want to read signals before all delta delays are completed...

```
async def stimuli_generator(dut):  
    ''' Generates all data for this tesbench'''  
    for i in range( 2**len(dut.input)):  
        await start_soon(set_stimuli(dut, i))
```

```
async def compare(dut):  
    ''' Compares simulated output with predicted output '''  
    while 1:  
        await Edge(dut.input) # Test on each new input  
        await ReadOnly() # Wait for output to settle  
        assert dut.output == predictor(dut), (  
            "output ({out}) is not as predicted: XOR({inp})"  
            .format(out=dut.output.value, inp=dut.input.value))  
        dut._log.info(  
            "output ({out}) is XOR({inp})"  
            .format(out=dut.output.value, inp=dut.input.value))
```

```
@cocotb.test()  
async def main_test(dut):  
    """ Starts comparator and stimuli """  
    dut._log.info("Running test...")  
    start_soon(compare(dut))  
    await start_soon(stimuli_generator(dut))  
    dut._log.info("Running test...done")
```

Cocotb keywords

- Cocotb documentation: <https://docs.cocotb.org/en/stable/>
- Coroutines generally in python: <https://docs.python.org/3/library/asyncio-task.html>

Suggested reading, corresponding assignments

Combinational logic

- D&H
 - 3.6
 - 6.1, 6.2, 6.3 (p105-109)
 - (6.4-6.9 p110- 120 .. Not syllabus)
 - 6.10 p121-123
 - 7.1 p 129-143
 -

Verification

- D&H:
 - 2.1.4 p 27
 - 7.2 p 143-148
 - 7.3 p 148-153
 - 20.1 p 453 – 456
- Oblig 1: «Design Flow»
 - See canvas for further instruction.
- Oblig 2: «VHDL»