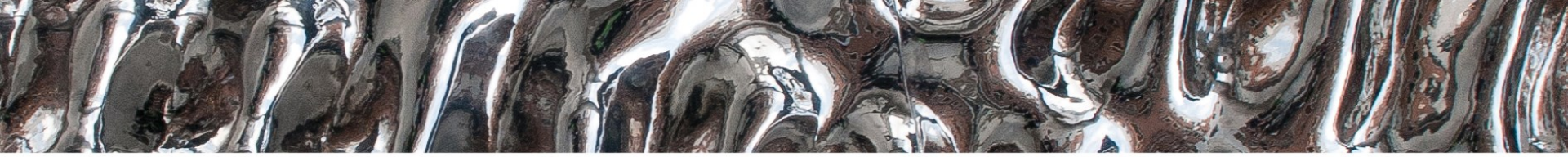

UiO : Department of Informatics
University of Oslo

IN 3160, IN4160

Organizing testbenches

Yngve Hafting

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:

- Know how to define well-structured test-benches

Why is verification *more* important *than ever*?

- Why do we make digital designs in the first place?
  - Other options aren't capable
    - Timing
      - Real time requirements
      - Parallellism
    - Power
  - Practicality
    - IO –reconfigurability
    - Maintaining options throughout design process
- Small simple, designs with few requirements
  - can almost always be solved using microcontroller boards.
- Using CPU architectures makes real time requirements tough to warrant

Overall questions...

- How should we verify designs?
- How do we plan verification?
- What defines quality in verification?
- What defines quality in test bench code?
  - What should i do / not do...

- Do we have decent examples showcasing this?

Overview

- Verification plan
  - Unit test
  - System wide tests

- Organizing a test bench
  - Structure
    - Test Sequencer
    - Stimuli generator
    - DUT-monitor
    - Fault-injector
    - Report module

  - Other modules

- Example:
  - Oblig 8 Testbench
    - Parts along the way

Verification plan

- A plan for how and when which type of tests should be applied.
- *Can be made once the design specification is made*

- Each module should have their own tests:
  - Can all input/output combinations be tested?
  - What cases do we need to test to verify function?
- System tests
  - Which modules need to be connected?

- Physical testing is *normally* not a part of the verification plan
  - *Test jigs*
  - *Built in self test*

Unit test (Module test bench)

- Planned tests
  - Typical cases (*As the module will be used*)
    - Every instruction
    - All states and all state transitions in FSMs
    - Every register
  - Corner cases
    - Unusual data, examples:
      - overflow
      - division by zero
      - Unused addresses etc.
    - Counting up past max
    - Counting down below zero
  - Corrupted data
    - Illegal states – response on unused states
  - Randomized data
    - *How much* random data is required to verify a solution..?

System tests

- Uses several modules
  - Full or partial module hierarchy
  - Simulation models
  - Bus functional models (BFM)
    - = "IP" for creating bus test data (auto generates handshake etc)

- Cocotb
  - A single top layer is required for each testbench
    - A top layer wrapper can always be created in VHDL
      - Enables the possibility of utilizing both co-simulation and standarized packages (UVVM, OVM, UVM etc)
  - *Independent test benches can be launched together and*
    - *communicate via network (TCP/UDP etc). (On a single PC)*

Example: Testbench for PWM module in Oblig 8

- Verification plan:
  - Reset test
    - PWM enable deasserted
  - Short circuit testing
    - en low when dir changes
    - en switches not too often
  - Response test
    - Not doing PWM when called for = fail (timing can vary)
  - Duty cycle
    - Actual PWM values should correspond to input
      - Not too fast
      - Not too slow
  - Data used:
    - Several duty cycles, both directions, fixed and random values.
    - reset at beginning and once at arbitrary time

```
entity pulse_width_modulator is
  port(
      mclk       : in std_logic;
      reset      : in std_logic;
      duty_cycle: in std_logic_vector(7 downto 0);
      dir        : out std_logic;
      en        : out std_logic
    );
end entity pulse_width_modulator;
```

Example

- Libraries, constants and dictionaries

```
# tb_pwm.py V4 : Testbench for PWM module with fault injection.
# By Yngve Hafting 04-11 2022

import cocotb
from cocotb import start_soon
from cocotb.clock import Clock
from cocotb.handle import Force, Freeze, Release
from cocotb.triggers import ClockCycles, Edge, First, FallingEdge, RisingEdge
from cocotb.triggers import ReadOnly, ReadWrite, Timer, with_timeout
from cocotb.utils import get_sim_time

import random
import numpy as np

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

#design constants
PERIOD_NS = 10
PWM_TIMEOUT_MS = 12
TOO_FAST_PWM_US= 160
```

Organizing test benches...

- How do we organize a module test bench?
  - What are the main components of a test bench?

Test bench structure

- Main classes *or modules*
  (VHDL does not organize by the use of classes, but similar structures may be applied)
  - ***Test sequencer***
    - Launches stimuli, fault injection, monitoring tasks and reporting at appropriate times
  - **Stimuli generator**
    - Applies all data input to the module
  - **DUT monitor**
    - Contains all tests performed on DUT signals
  - ***Fault injector***
    - Creates error conditions to test the DUT monitor
    - To be disabled once the DUT monitor is verified
  - ***Report module***
    - *Implements scoreboard functionality*
    - *Presents data in log or screen*
  - **Other** (sub)classes... as needed
    - Signal monitoring classes:
      - To provide (extended) VHDL attribute style testing
        - » Ex: 'stable 'stable_interval, etc

Test Sequencer

- Launches
  - stimuli
  - monitoring tasks and tests
  - *fault injection* (if any)
  - reporting
- All at appropriate times

Example: Main test

- = *test sequencer (here)*
- creates an object of
  - stimuli_generator
  - DUT monitor
  - *and fault injector*
- *Runs the necessary methods to perform all tests*

```
@cocotb.test()
async def main_test(dut):
    ''' Starts monitoring tasks and stimuli generators '''
    stimuli = StimuliGenerator(dut)

    # Wait for Uninitialized (U) inputs to get a resolvable (numeric) value
    await Timer(1, 'ns')
    monitor = Monitor(dut)
    await stimuli.run()

    # Inject Faults to check that the testbench responds to faults
    #trip = FaultInjector(dut)
    #await trip.run()
    #dut._log.info("*** Done testing ***")
```

Stimuli generator

- Fetches or creates data to stimulate DUT
  - Test data may be fetched from files or network
    - Cosimulation (cocotb etc)
      - allows for retrieving network packages created by other sources (Matlab etc)
      - Allowes for using software packages for image analysis etc

- Ideally stimuli is created independently of tests
  - Not always possible
  - Stimuli *may* be in response to model behaviour

- Makes sure all data required by the test plan is generated
  - Separate class for stimuli makes the plan easier to verify

- Stimulate module inputs only (`dut.<signal>.value = <...>`)
  - *Avoid overriding internal values*
    - *using "`force`" may cause inconsistencies and mask erroneous behavior*

Example: Stimuli Generator

- Feeds DUT with all data for the test
  - Here:
    - Reset performed on creation
    - run-method provides all data sequentially
      - *"data sequencer"*
    - Separate methods for each way of providing data

```python
class StimuliGenerator():
    ''' Generates all stimuli used in the normal tests '''
    def __init__(self, dut):
        self.dut = dut
        self.dut._log.info("Starting clock")
        start_soon(Clock(self.dut.mclk, PERIOD_NS, 'ns').start())
        self.dut.duty_cycle.value = 0
        start_soon(self.reset_module())

    async def reset_module(self):
        self.dut._log.info("Resetting module... ")
        self.dut.reset.value = 1
        await Timer(15, 'ns')
        self.dut.reset.value = 0

    async def run(self):
        self.dut._log.info("Starting duty cycle tests ")
        await Timer(20, 'ns')
        await self.seqential_duty_tests()
        self.dut._log.info("Sequential duty tests complete ")
        await self.random_duties(7)
        self.dut._log.info("Random duty tests 1/2 complete ")
        await self.reset_module()
        self.dut._log.info("Reset between duties complete ")
        await self.random_duties(3)
        self.dut._log.info("Random duty tests 2/2 complete ")

    def set_duty(self, duty_cycle):
        self.dut.duty_cycle.value= int((duty_cycle*128)/100)

    async def seqential_duty_tests(self):
        self.set_duty(50)
        for i in range(3):
            await RisingEdge(self.dut.en)
        self.set_duty(-50)
        for i in range(2):
            await RisingEdge(self.dut.en)

    async def random_duties(self, tests):
        duties = list(range(-90+1,-10)) + list(range(10+1,90))
        for x in range(tests):
            random_duty = random.choice(duties)
            duties.remove(random_duty)
            self.set_duty(random_duty)
            for i in range(2):
                await RisingEdge(self.dut.en)
        interval = random.randint(1,300)
        await Timer(interval, units='us')
```

DUT monitor

- Contains all checks applied to the DUT
  - Both output and internal states *can* be read
    - Checking internal signals (whitebox-testing) makes the test less robust
      - Ie changing the design changes the tests.

- Compares data from DUT and the model behavior
  - Model-behavior can be calculated or fetched from other sources
  - Should ideally respond to DUT in- and output.
    - *Avoid creating stimuli in the monitor class*
      - *Mixing makes both less robust to changes*
      - *Mixing will easily result in complex -ad hoc type testing (test this then that, then...)*

- Is *verified* when
  - all the tests in the test plan is applied
  - each test can be confirmed triggered...
    - *Fault injection can be used for this (description follows)*

Example: DUT-Monitor 1/2

- Here:
  - Each check has its own triggers
  - Each check run throughout all stimuli
  - all checks are launched on creation (__init__)
  - `while True`:
    - "True" could be replaced with <condition> for toggling usage
  - One or more assertions for each checker

- Not implemented here: *reporting*
  - *More on reporting later...*

```
class Monitor:
    """ Contains all tests checking signals in and from DUT """
    def __init__(self, dut):
        self.dut = dut
        self.dut._log.info("Starting monitoring events")
        self.en_mon  = SignalEventMonitor(self.dut.en)
        self.duty_mon = SignalEventMonitor(self.dut.duty_cycle)
        self.reset_mon = SignalEventMonitor(self.dut.reset)
        start_soon(self.reset())
        start_soon(self.dir_stability())
        start_soon(self.timeout())
        start_soon(self.duty_dir_cohesion())
        start_soon(self.duty_checks())

    async def reset(self):
        ''' Checks that PWM pulse (en) is deasserted when reset is applied '''
        while True:
            await FallingEdge(self.dut.reset)
            assert self.dut.en.value == 0,
                   "PWM enable has not been deasserted during reset"
            self.dut._log.info("Passed: Reset test")

    async def dir_stability(self):
        ''' Checks that we are not short-circuiting the
            half-bridge by switching direction while pulsing '''
        while True:
            await Edge(self.dut.dir)
            if self.dut.reset.value == 0:
                assert self.dut.en.value == 0, (
                  "HALF-BRIDGE SHORT CIRCUITED: en active when changing direction")
                assert self.en_mon.stable_interval('ns') > PERIOD_NS-1, (
                  "SHORT CIRCUIT DANGER:
                   en deactivated less than one cycle before dir change")
                wait_task = Timer(PERIOD_NS-1, 'ns')
                event_task = Edge(self.dut.en)
                result = await First(wait_task, event_task)
                assert result == wait_task, (
                  "SHORT CICUIT DANGER: En was not stable for {per} {uni}"
                  .format(per=PERIOD_NS, uni='ns'))

    async def timeout(self):
        ''' Checks that the PWM signal is actually driven
            within a reasonable timeframe'''
        while True:
            if self.dut.duty_cycle.value == 0 : await Edge(self.dut.duty_cycle)
            await with_timeout(Edge(self.dut.en), PWM_TIMEOUT_MS, 'ms')
```

Example: DUT-Monitor 2/2

Here:

- `duty_checks` has dual purpose
  - Ideally these would be separate..
  - *Uses same calculation and triggers*

```python
async def duty_dir_cohesion(self):
    ''' Checks that the pwm drives the motor in the correct direction'''
    while 1:
        await Edge(self.dut.duty_cycle)
        await ClockCycles(self.dut.mclk, 2)   # Trigger two clock edges after duty cycle was changed
        await ReadOnly()                      # Wait for all signals to settle (all delta delays)
        duty = int(self.dut.duty_cycle.value) # Check if sign and dir matches
        if np.int8(duty) > 0:
            assert self.dut.dir.value == 1, (
              "DIR is not '1' within 2 clock cycles of positive duty cycle: {DU} = {D}"
              .format(DU=np.int8(duty), D=self.dut.duty_cycle.value))
        if np.int8(duty) < 0:
            assert self.dut.dir.value == 0, (
              "DIR is not '0' within 2 clock cycles of negative duty cycle: {DU} = {D}"
              .format(DU=np.int8(duty), D=self.dut.duty_cycle.value))

async def duty_checks(self):
    ''' Checks that pwm pulses are not happening too fast for the PMOD module '''
    await RisingEdge(self.dut.en)
    while 1:
        # Wait until we have a full period after reset
        if self.dut.reset.value == 1:
             await FallingEdge(self.dut.reset)
             await RisingEdge(self.dut.en)
        await RisingEdge(self.dut.en)

        # Find the interval/period
        start = self.en_mon.last_rise/ps_conv['us']
        interval =  get_sim_time('us') - start

        # Trigger only when duty cycle has been stable for the last period
        if self.duty_mon.stable_interval('us') > interval:
            assert interval > TOO_FAST_PWM_US, (
              "PWM period too short!: {iv:.2f}us, f={f:.3f}kHz   Minimum period: {per} us, f<{maxf:.2f}kHz "
              .format(iv=interval, f=(1000/interval), per=TOO_FAST_PWM_US, maxf=(1000/TOO_FAST_PWM_US)))

            # Calculate duty cycle
            mid = self.en_mon.last_fall/ps_conv['us']
            high = mid-start
            measured_duty = np.int8((high*100)/interval)
            set_duty = np.int8(self.dut.duty_cycle.value.integer)*100/128

            # Report duty cycle and check correspondence between input and output
            sign = "-" if self.dut.dir.value == 0 else " "
            self.dut._log.info(
              "Duty cycles: Set dc: {S:.1f}%, Measured dc: {Sig}{M:.1f}%, period = {P:.1f}us, f = {F:.2f}kHz"
              .format(S=set_duty, Sig = sign, M = measured_duty, P = interval, F = 1000/interval))
            abs_duty = abs(set_duty)
            deviation = np.int8(abs(abs_duty - measured_duty))
            assert deviation < 5, (
              "Set and measured duty cycle deviates by more than 5% ({D}%) "
              .format(D=deviation))
```

Test triggers

- Tests can be performed when called for or when triggered
  - Called tests run manually, inserted at certain timeslots
    - May miss important failure conditions because it was not performed at correct time.
    - May cause the code to become a long list of *applying data -> test output -> apply data -> test output -> ...->...*
      - Such a list in itself is hard to verify
        - » Did we cover all necessary situations?
- Better: Use triggers
  - Make the test latently wait for their trigger while testing any data.
    - Trigger conditions can be fixed (in one spot)
    - Test can be fixed in one spot
    - Data can be generated once.

Fault injector (*note: own work*)

- Verifies that the DUT-monitor responds to errors
  - Create at least one fault injection algorithm that triggers each test
- Forces (overrides) DUT output to erroneous behaviour
  - *Can* be done independent of other stimuli
  - Can be used before the design module is ready
- Disable after use
  - Forced values may cause malfunction in other tests
  - May "flood" reports / *ordinary check reports becomes hard to spot*
  - Inactive fault injection can be revoked when making changes to tests
    - *Keep the code as a testimony of the checks capability to uncover errors*
- **Fault injections that *does not* trigger an error should fail the test!**

Example: Fault injector

```python
class FaultInjector():
    """ Contain tests to verify that each assertion will trigger """
    def __init__(self, dut):
        self.dut = dut
        self.dut._log.info("*** TRIP-BENCH INITIALIZED ***")

    async def run(self):
        # TESTS run, comment out when working
        #await self.reset()
        #await self.dir_stability_1()
        #await self.dir_stability_2()
        #await self.dir_stability_3()
        #await self.timeout()
        #await self.duty_dir_cohesion()
        #await self.too_fast_pwm()
        #await self.duty()
        assert False, "Trip run came to an end without tripping anything else!"

    def release(self):
        ''' Releases all Forced values. '''
        self.dut.reset.value = Release()
        self.dut.en.value = Release()
        self.dut.dir.value = Release()
        self.dut.duty_cycle.value = Release()

    async def disable_reset(self):
        self.dut.reset.value = Force(0)
        await Timer(20, 'ns')

    async def reset(self):
        ''' Enable asserted while reset is deasserted'''
        self.dut.reset.value = Force(1)
        self.dut.en.value = Force(1)
        await RisingEdge(self.dut.mclk)
        self.dut.reset.value = Force(0)
        await RisingEdge(self.dut.mclk)
        self.release()
```

```python
    async def dir_stability_1(self):
        ''' Enable is not deasserted when dir changes'''
        await self.disable_reset()
        self.dut.dir.value = Force(0)
        self.dut.en.value = Force(1)
        await Timer(1, 'ns')
        self.dut.dir.value = Force(1)
        await Timer(30, 'ns')
        self.release()

    async def dir_stability_2(self):
        ''' Enable is deasserted within one clock cycle of dir changing'''
        await self.disable_reset()
        self.dut.en.value = Force(1)
        await Timer(1, 'ns')
        self.dut.en.value = Force(0)
        await Timer(8, 'ns')
        self.dut.dir.value = Force(not self.dut.dir.value)
        await Timer(10, 'ns')
        self.release()

    async def dir_stability_3(self):
        ''' Enable asserted within one clock cycle after dir changing '''
        await self.disable_reset()
        self.dut.en.value = Force(0)
        await Timer(20, 'ns')
        self.dut.dir.value = Force(not self.dut.dir.value)
        await Timer(8, 'ns')
        self.dut.en.value = Force(1)
        await Timer(1, 'ns')
        self.release()

    async def timeout(self):
        ''' Prevents pulsing although a nonzero duty cycle'''
        await self.disable_reset()
        self.dut._log.info("*** timeout test started: this may take minutes ***")
        self.dut.duty_cycle.value = Force(0x50)
        # Stopping clock might be useful to reduce time spent here.
        # Requires a pointer/handle to the process.
        self.dut.en.value = Force(0)
        await Timer(PWM_TIMEOUT_MS+1, 'ms')
        self.release()
```

```python
    async def duty_dir_cohesion(self):
        ''' To not have dir correspond to duty cycle within two clock cycles'''
        await self.disable_reset()
        self.dut.dir.value = Freeze()
        self.dut.duty_cycle.value = Force(0xEE)
        await ClockCycles(self.dut.mclk, 3)
        self.dut.duty_cycle.value = Force(0x11)
        await ClockCycles(self.dut.mclk, 3)
        self.release()

    async def too_fast_pwm(self):
        ''' Runs PWM signal (en) faster than allowed by TB'''
        await self.disable_reset()
        for i in range(4):
            self.dut.en.value = Force(1)
            await RisingEdge(self.dut.mclk)
            self.dut.en.value = Force(0);
            await RisingEdge(self.dut.mclk)
        self.release()

    async def duty(self):
        ''' Asserts one duty cycle, and pulses another'''
        await self.disable_reset()
        self.dut.en.value = Force(0)
        await ClockCycles(self.dut.mclk, 10)
        self.dut.duty_cycle.value = Force(-32) # 25% as hex (128 = 100%)
        await ClockCycles(self.dut.mclk, 10)
        for i in range(4):
            self.dut.en.value = Force(1)
            await ClockCycles(self.dut.mclk, 8001)
            self.dut.en.value = Force(0);
            await ClockCycles(self.dut.mclk, 8001)
        self.release()
```

- All tests still present although disabled.
- NOTE: GHDL bug using force

Creating testbench *with fault injection*

- Create each check and corresponding fault injection
  - One at a time
  - Reiterate and run the checks until the test properly triggers on faulty conditions
    - Then switch off that fault injection
    - When checks are changed, reuse fault injection

- Fault injection algoritms (FIA) *may* be created independent of DUT
  - Necessary triggers (for starting the checks) must be injected
    - Ie simulating DUT up to the point where the error is injected.

Report module

- Typically used for scoreboard
  - What checks are performed
  - What stimuli is applied
  - What worked and what went wrong
- Turn on/off reporting
- *Queues in python can be suited for reporting*
- Used in industry standard test environments
  - UVVM (VHDL)
  - UVM, OVM (Verilog //...)
- *Cocotb (Python) has built in log module*
  - `dut._log.info("<...>")`

Example: Report module

- Uses cocotb built-in logging and assertions to report to screen.
  - -- ER Mulig å gjøre mer ut av – ie rapportere pdf excel osv.

Other modules: Example: Signal monitoring subclass

- VHDL attribute-like functionality
  - 'stable
  - 'last_event
  - 'last_rise 'last_fall *(doesn't exist in VHDL)*

```
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)
            await ReadOnly()          # RO.-allows edge-edge measurment
            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
```

Summary

- Exam relevancy:
  - General structure and meaning
    - Ie. what does a fault injector do..?
  - *You will not be asked to create a full test bench with this structure..*
    - *= too much for exam*

- Sources:
  - Effective coding with VHDL, Richardo Jasinski
  - Own work, ...

Bing Image Creator: The final testbench, *steampunk art*