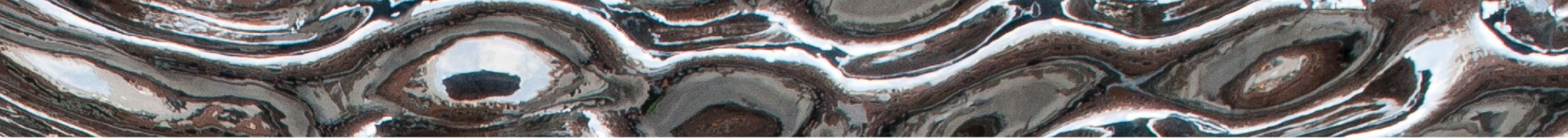


UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

IN3160, IN4160 Digital system design

Introduction
HDL, PL and Design flow

Yngve Hafting

Overview

- General information
  - Course management
  - Schedule
  - Course Goals
  - Curriculum
  - Lab assignments
  - Who are we
- Motivation
  - Why Digital Design?
  - Why HDL?
- Intro to programmable Logic
  - What is programmable logic?
  - Why choose programmable logic?
- Design Flow for digital designs
- Intro to our hardware…:
  - Zedboard
    - Architecture
    - Documentation
  - «Our» HDL: VHDL

- Assignments and suggested reading for this week

Course Management

- Lecturers:
  - **Alexander** Wold (II'er)
  - **Yngve** Hafting (Universitetslektor)
- Lab supervisors / teachers:
  - **Elias** Ringkjøb (student)
  - **Jonas** Wenberg (student)
  - **Øystein** Øverbø (student)

Lectures

Tuesday 10:15 -12:00, OJD Caml

Friday 10:15-12:00, OJD Caml

Lab

LISP (2428): TBD- lab will be manned certain time slots- poll next slide

https://www.mn.uio.no/ifi/om/finn-fram/apningstider/

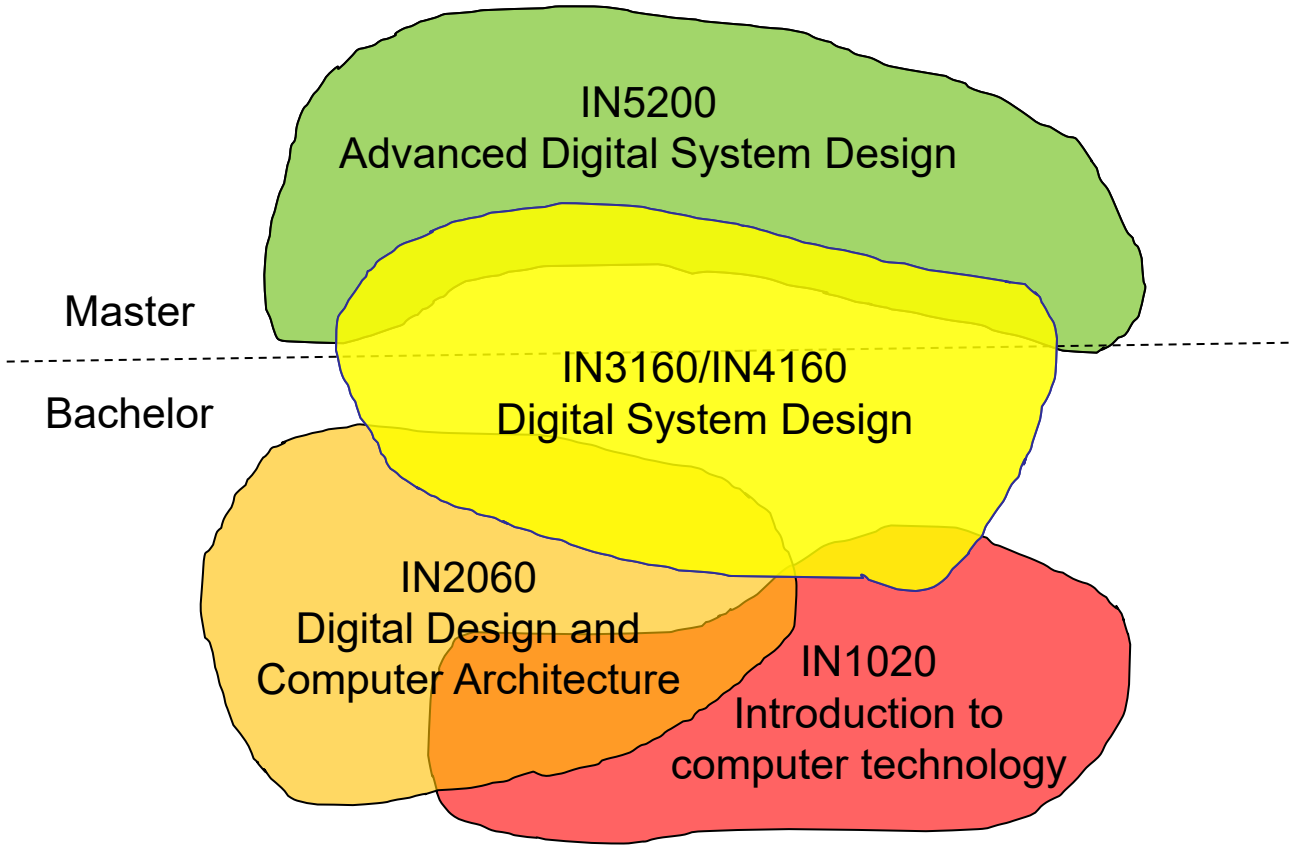
Tid (.15)	Monday	Tuesday	Wednesday	Thursday	Friday
8					
10		Lecture 10-12			Lecture 10-12
12				IN3050 Lecture	
14		Øystein		Elias	Jonas
16					

http://www.uio.no/studier/emner/matnat/ifi/IN3160/
(covers also INF4160)

Where do we stand + lab supervision poll?

- www.menti.com
- Code 2536 2149

(ROBIN) Study program connections



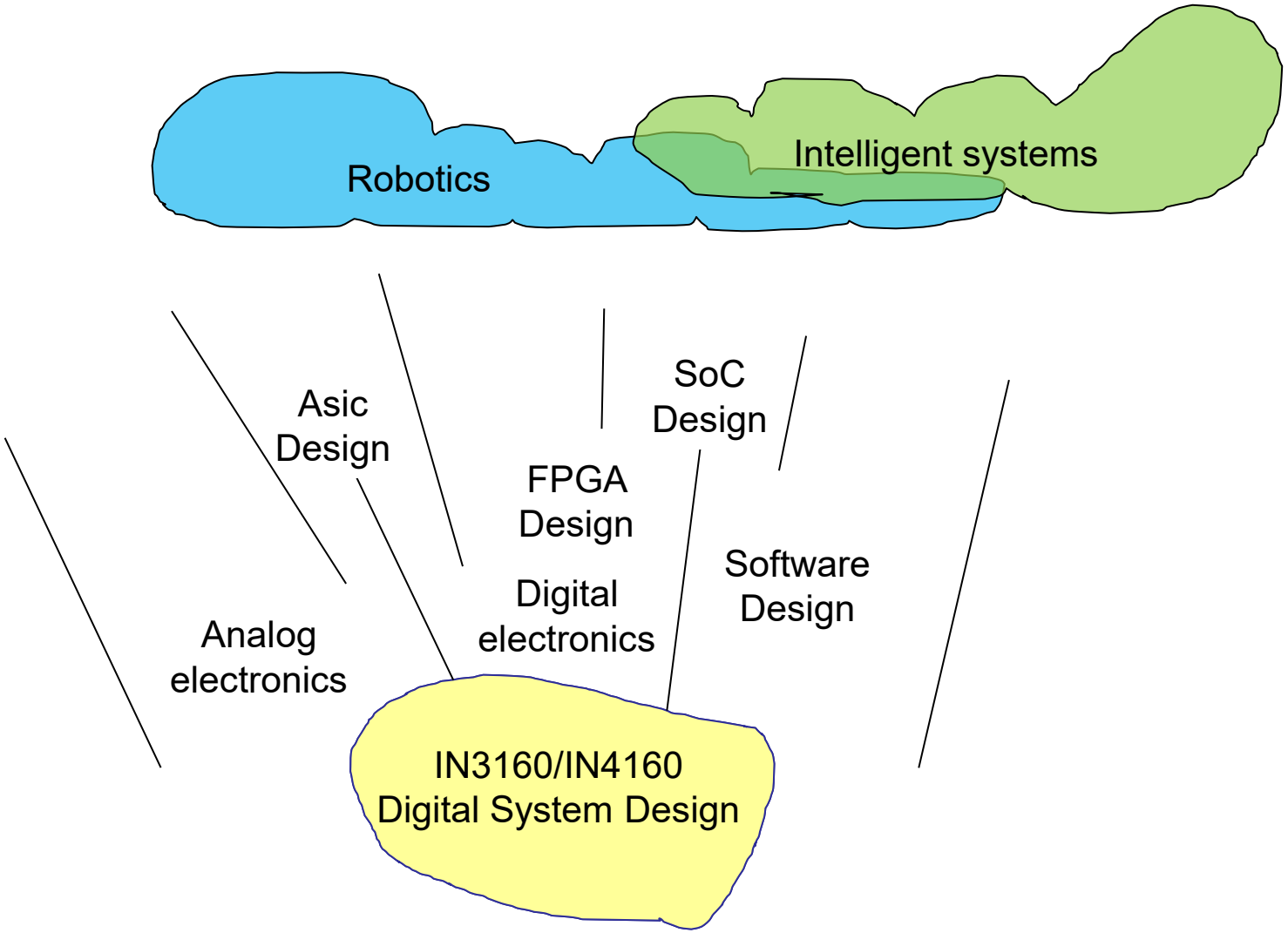
Courses in electronics and circuit theory, such as

FYS1210 - Electronics
FYS3220 - Linear circuit theory
FYS3240 - Data acquisition and control

provide a complementary background that may help understanding digital systems in general.

FYS4220 - Real time and Embedded Data Systems overlaps with 6p

Relevancy



Finishing this course you will be able to do work within the field of digital design.

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'll:...

... IN3160 vs IN4160 ...

IN3160

- After completion of the course you'll:
  - 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.

IN4160

- After completion of the course you'll:
  - 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 **advanced** simulation and synthesis of digital systems
  - **be able to perform advanced implementation and analysis techniques**

NOTE: these are MINIMUM requirements for passing an exam.

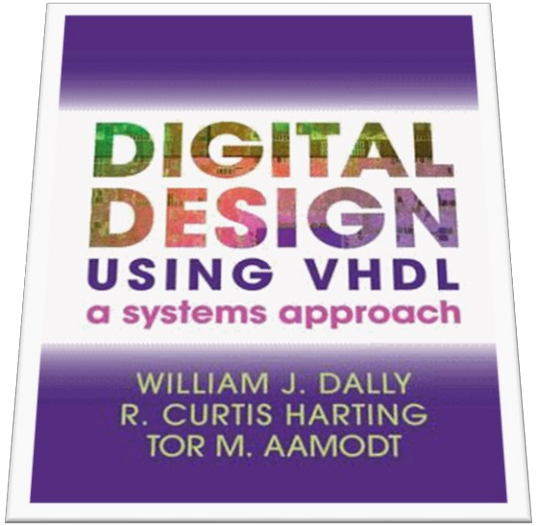
- You will be given the same opportunities to learn, and the curriculum is the same.
- *Grading will be (slightly) stricter for IN4160 due to added minimum requirements*
- Otherwise, this course will be held as one.

Our approach (*compared to other studies*)

- Focus on relevancy for students with programming skills
  - less physics and maths oriented
    - less transistor and PCB technology
    - less focus on mathematical proofs and methods
    - less focus on tweaking (IN2060 covered this)
- Focus on design strategies
  - High level code (New -24: Testbenches in python)
  - Schematics / Diagrams
  - Verifiability
  - State machines
- Focus on physical realization on FPGA
  - Full tool chain used in industry

Syllabus

- Dally, William J. - Harting, R. Curtis - Aamodt, Tor M.
  Digital Design Using VHDL A Systems Approach
  Cambridge University Press 2016
  ISBN9781107098862



- Lectures and lecture slides
- Mandatory assignments
- Handouts – Will be made available digitally on semester page
  (Link from 2022 can be used until the 2023 link is ready)
  - Cookbook
  - Articles (Reset Circuits, Steve Kilts)

Lab assignments

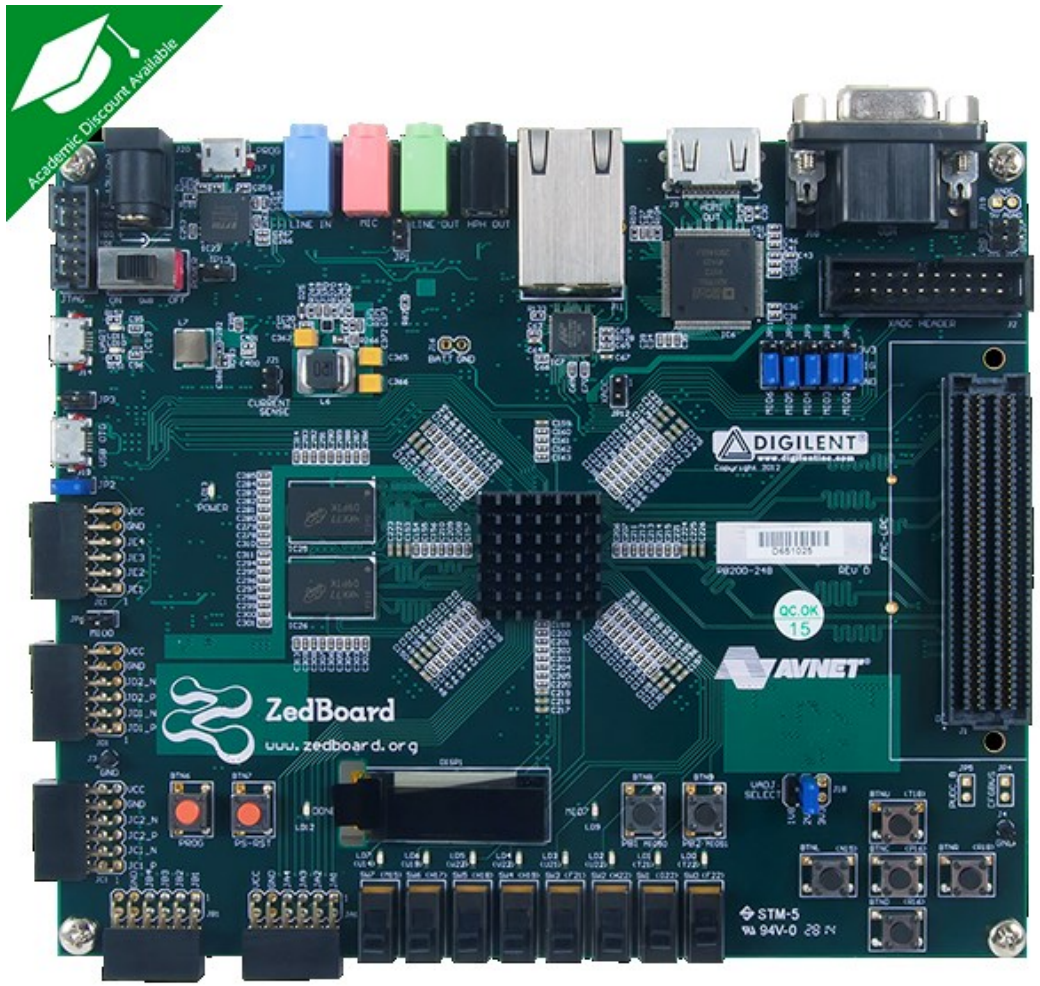
- There are 10 obligatory lab assignments.
  - *The book has chapter-exercises that can be used for self-study.*
- All assignments must be completed to take the exam.
  - *Lab workload and complexity increases* through the semester

- Lectures are prerequisite for some assignments
  - Lectures most intensive in the beginning

- The lab assignments utilises the digilent Zedboard, featuring a Xilinx Zynq 7020 device that includes both a hardcoded ARM processor and FPGA fabric.

- You will be introduced to tools and board first.

- By the end of this course you will design a system, using both processor and FPGA fabric, that will both regulate, read and display the speed of an electric motor connected to the board.



https://store.digilentinc.com/zedboard-zynq-7000-arm-fpga-soc-development-board/

LAB

- Lab starts now!
  - **Assignments are available in Canvas!**
  - Assignments are individual.
    - WHAT ABOUT...
      - Collaboration?
      - Previously approved assignments?
      - ChatGPT?
  - There will be one assignment (3) using peer review only.
    - Your reviews are mandatory for your approval!
  - Some assignments may require that you show your setup to the lab supervisor.
    - ***Labs can be done entirely remote, but on-site is strongly adviced.***
- **LISP (2428) is the LAB.**
  - Both hardware and software will be available in LISP.
    - 4 boards with camera will be available online for those in quarantine/ isolation / specieal needs.
- Questions..?

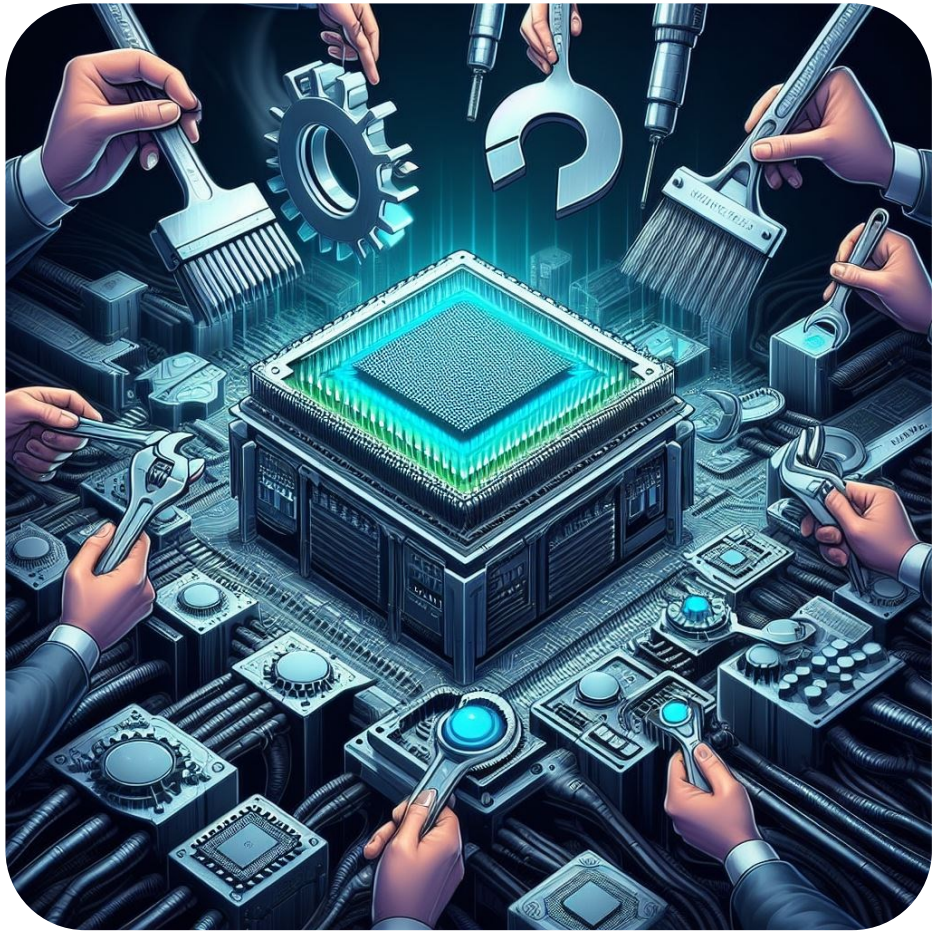
Software

- Vivado, Vitis
  - Floorplanning and Programming FPGA boards
  - https://www.xilinx.com/products/design-tools/vivado.html
    - *Standard edition is free and should be sufficient up to assignment 9.*

- Tool chain for Python Testbenches = Vivado + these https://robin.wiki.ifi.uio.no/Cookbook (Not complete, but has installation guide)
  - GHDL
    - Open source VHDL simulation, used together with CocoTB below.
  - GTKWave
    - Open source waveform viewer
  - CocoTB
    - Cosimulation framework for Python testbenches
    - This is invoked when using "make" when using python based testbenches.

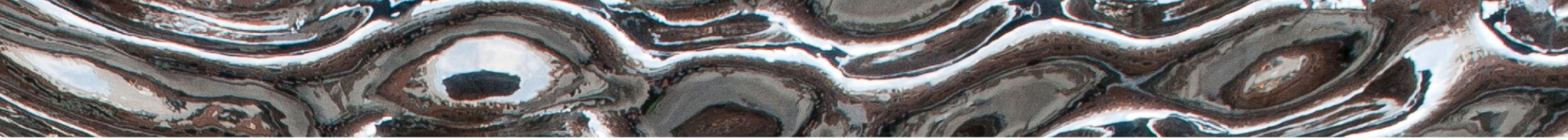
- Questa=Modelsim (Fallback solution if GHDL fails)
  - Compilation, Simulation, waveform viewer
  - "industry standard"
  - Not open source. Access may be limited

- All software can be accessed from Linux machines on IFI, and IFI-digital-electronics
- GHDL+ GTKWave and Cocotb



Resources

- Semester page/ course web "Vortex"
  - Course information
    - Exam date, lecture schedule, etc.
- Canvas (link from semester page)
  - Assignment-files, delivery and -feedback
  - Some links to external content
- in3160-discourse
  - Communication and discussion:
    - Requires login, allowes anonymous posting.
    - Both students and staff can answer ☺
  - Note: Use the manned lab hours as much as possible for questions.
- https://robin.wiki.ifi.uio.no/Cookbook



UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

HDL & PL

Hardware **D**escription **L**anguage & **P**rogrammable **L**ogic

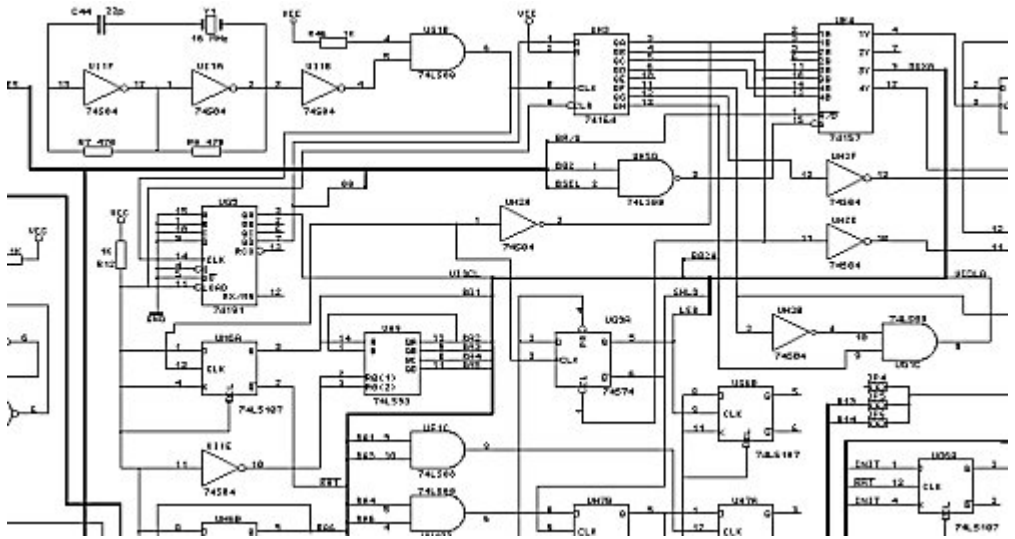
Yngve Hafting

Overview

- How to implement digital designs?
- HDL vs schematics
- **Why use HDL**...
- HDL vs Software

- Hardware
  - ASICs vs PL
  - Some types of PL

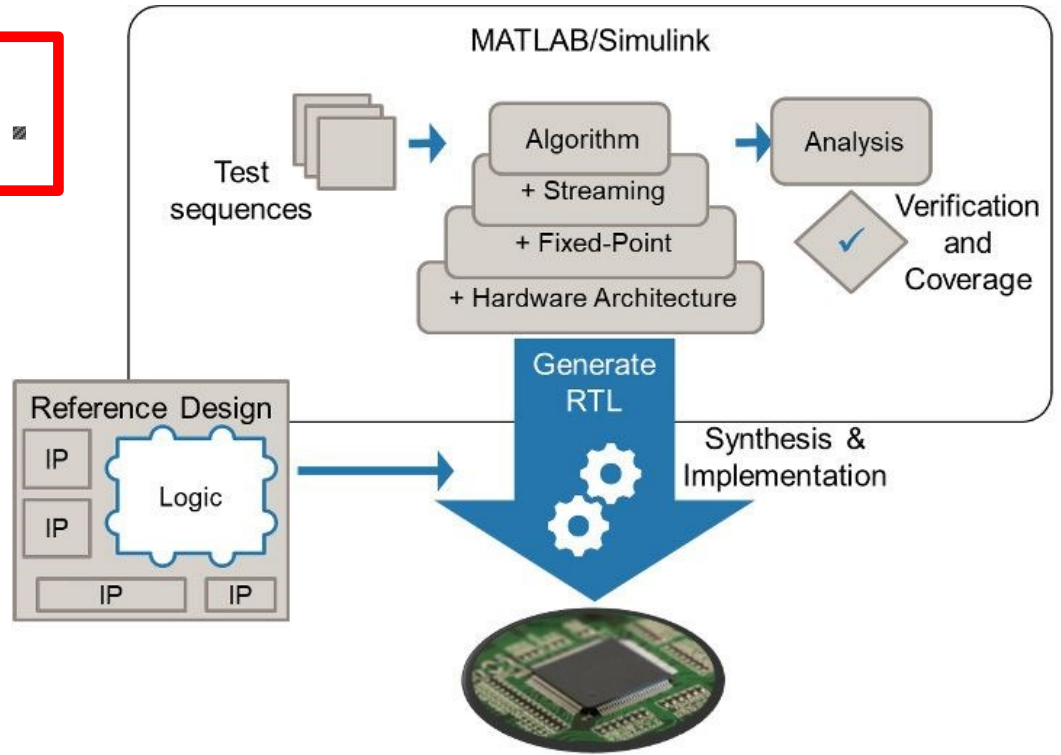
How to implement digital designs..?



- Low Level
  - Netlists
  - Schematic diagrams
  - Programming using hardware description languages (HDL programming)

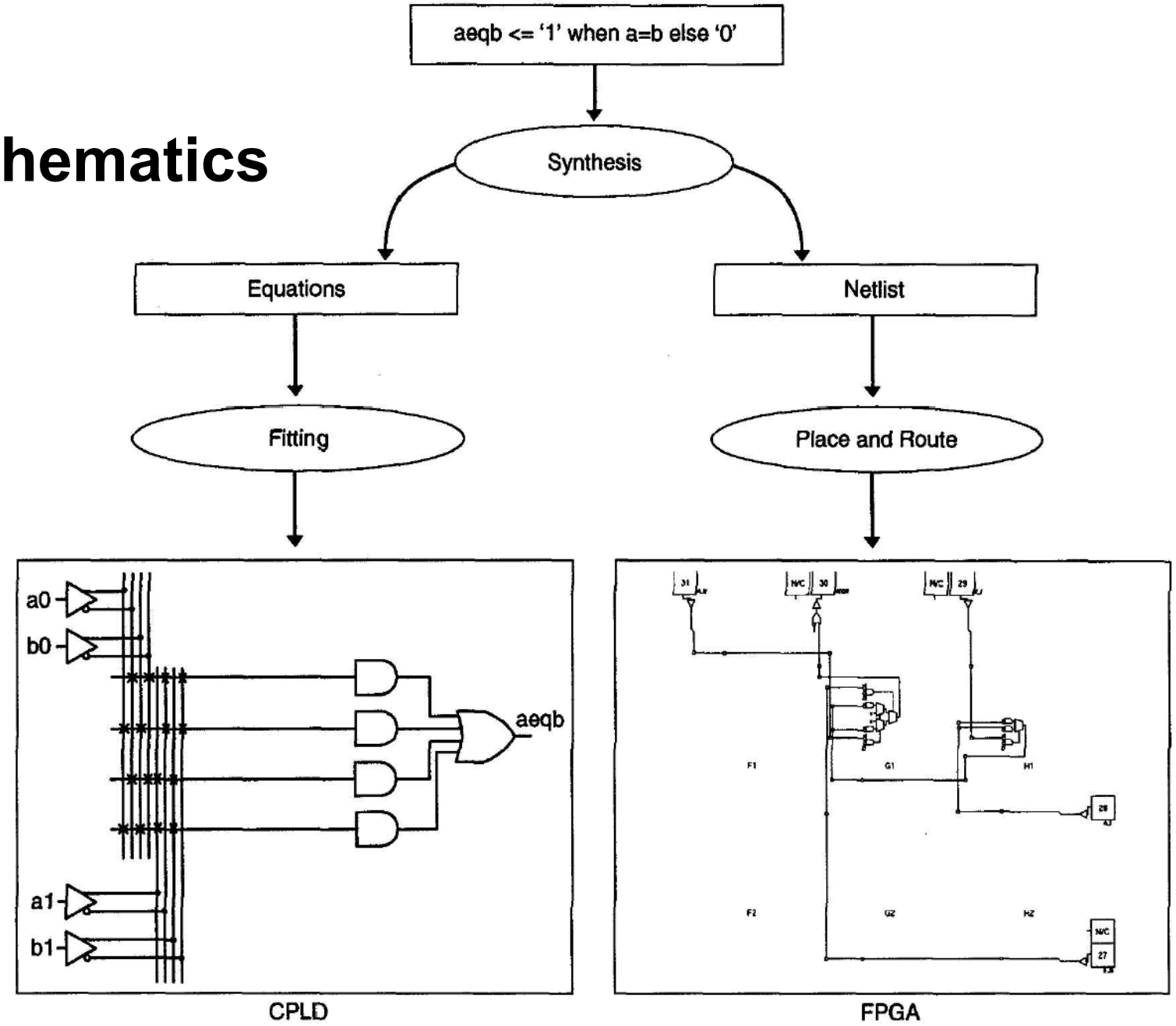
IN3160...

- High Level
  - HDL programming (RTL)
  - Block diagrams
    - Connecting premade models (IP's)
  - High level synthesis...
    - Code Generators (Matlab/Simulink)
      - Uses IP's
      - Generates HDL/ Netlists



HDL vs netlists/ Schematics

- Syntesis enables
  - One design
    - Several physical implementations



Why HDL?

- Technology independent code
- Different abstraction layers

Netlist:

```
U1: xor2 port map(a(0), b(0), x(0));
U2: xor2 port map(a(1), b(1), x(1));
U3: nor2 port map(x(0), x(1), aeqb);
```

Boolean equations:

```
aeqb <=      (a(0) xor b(0))
         nor (a(1) xor b(1));
```

Concurrent statements:

```
aeqb <= '1' when a=b else '0';
```

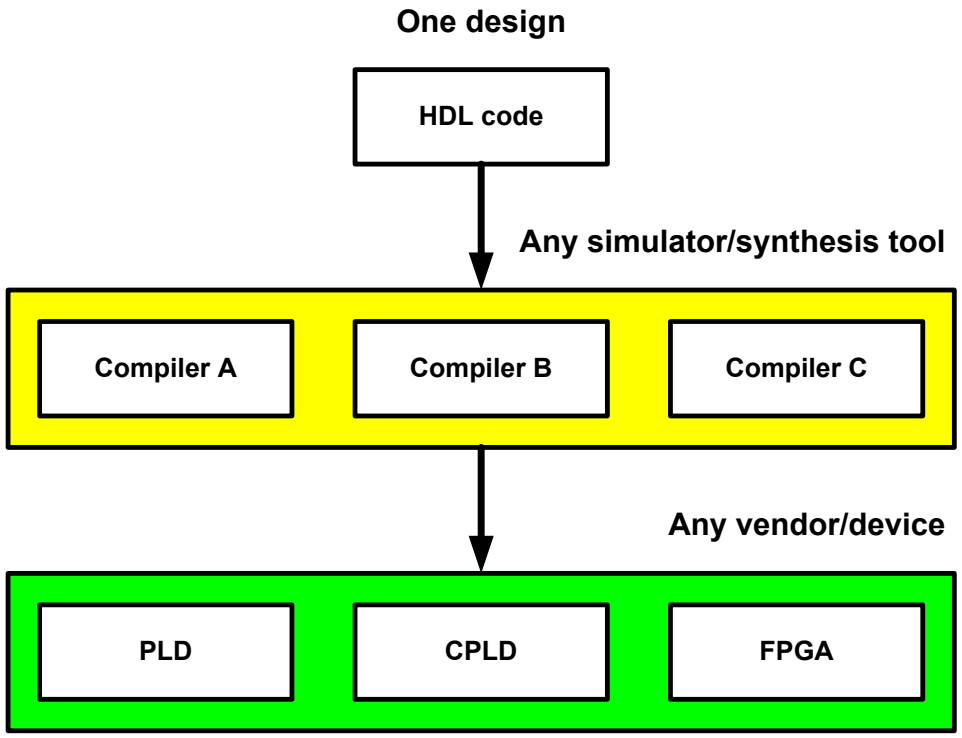
Sequential statements:

```
if a=b then
  aeqb <= '1';
else
  aeqb <= '0';
end if;
```

Why HDL?

- Portability
  - Tool and device independency

- IEEE Standards
  - VHDL and System Verilog
    - Both IEEE standards
      (Institute of Electrical and Electronics Engineers)
    - VHDL - IEEE 1076
    - System Verilog - IEEE 1364



Why HDL?

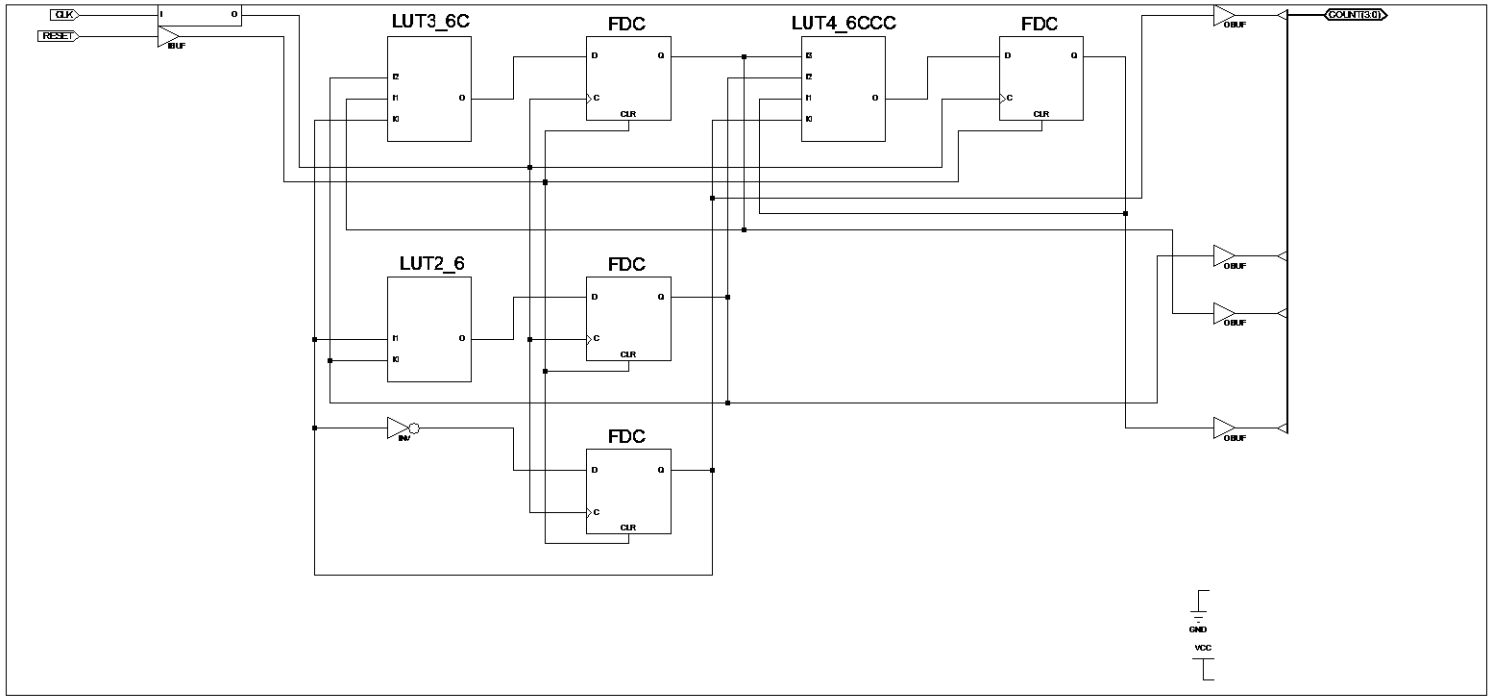
- Example :
  Simple mainteenance/expansion of a counter

```
signal r_count, next_count : unsigned(N-1 downto 0);
---
next_count <= r_count+1;

REGISTER_UPDATE: process(clk) is
begin
  if rising_edge(clk) then
    r_count => (others => '0') when reset else next_count;
  end if;
end process;
```

Why HDL?

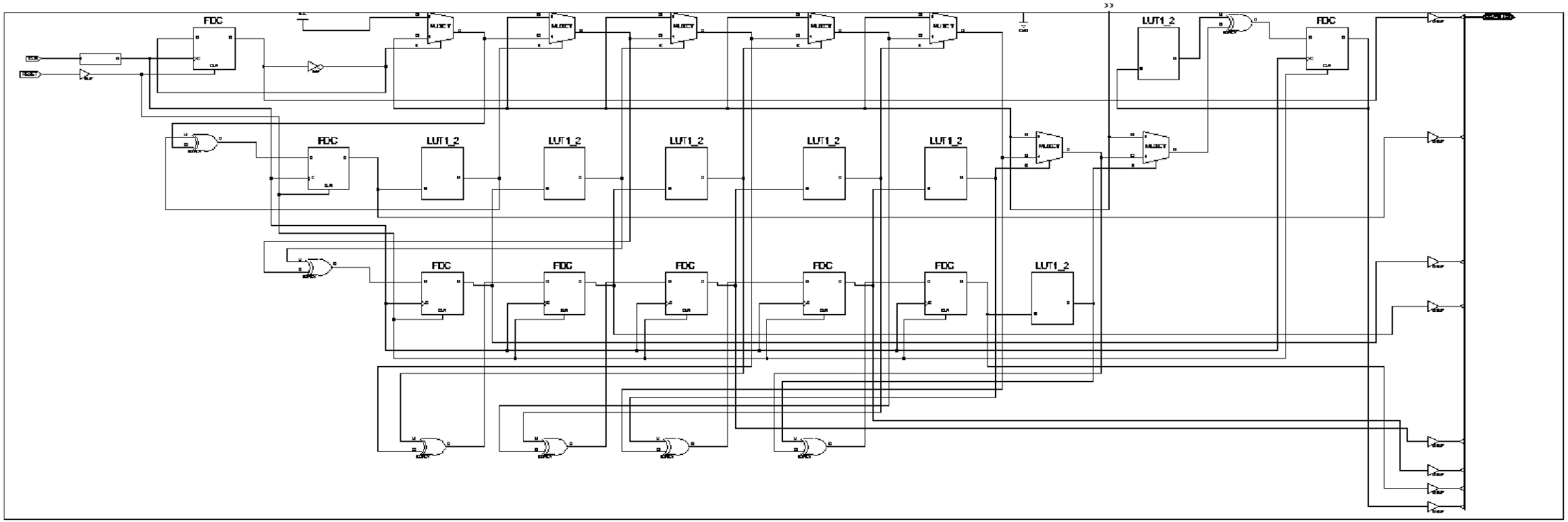
4 bit counter:

```
signal r_count, next_count : unsigned(3 downto 0);
```



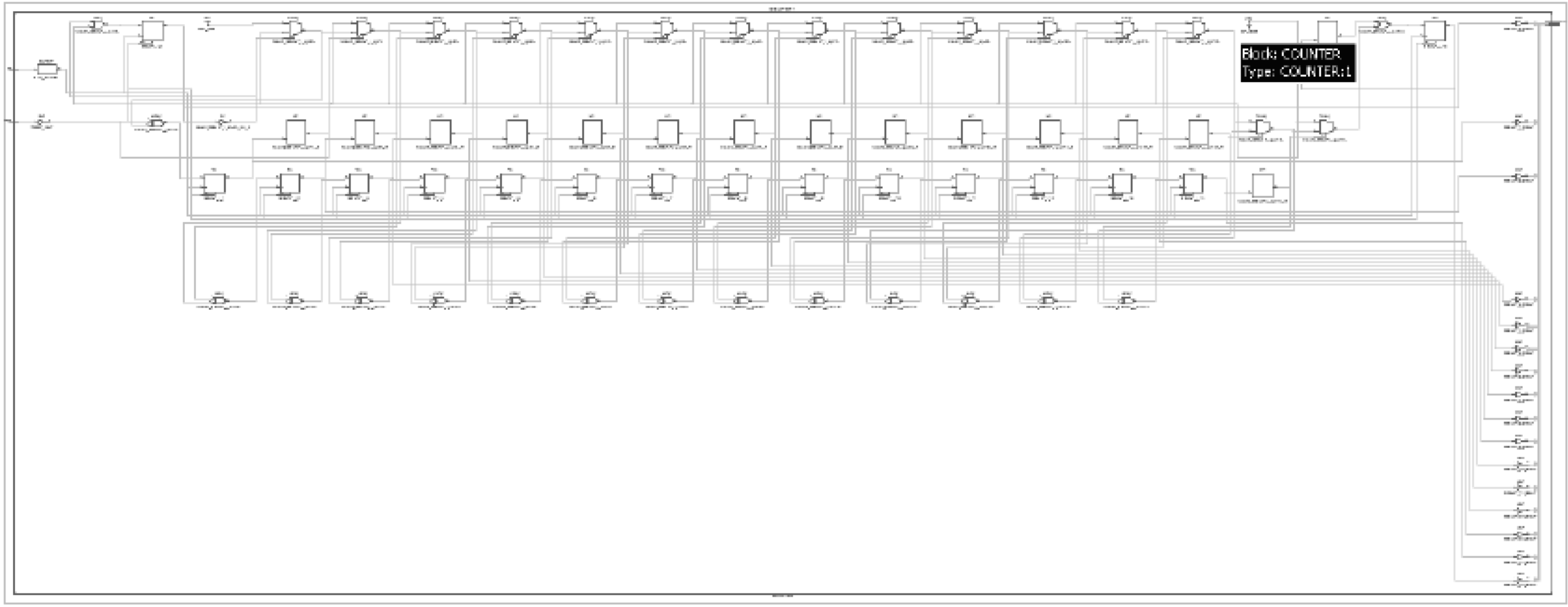
8 bit …

```
signal r_count, next_count : unsigned(7 downto 0);
```



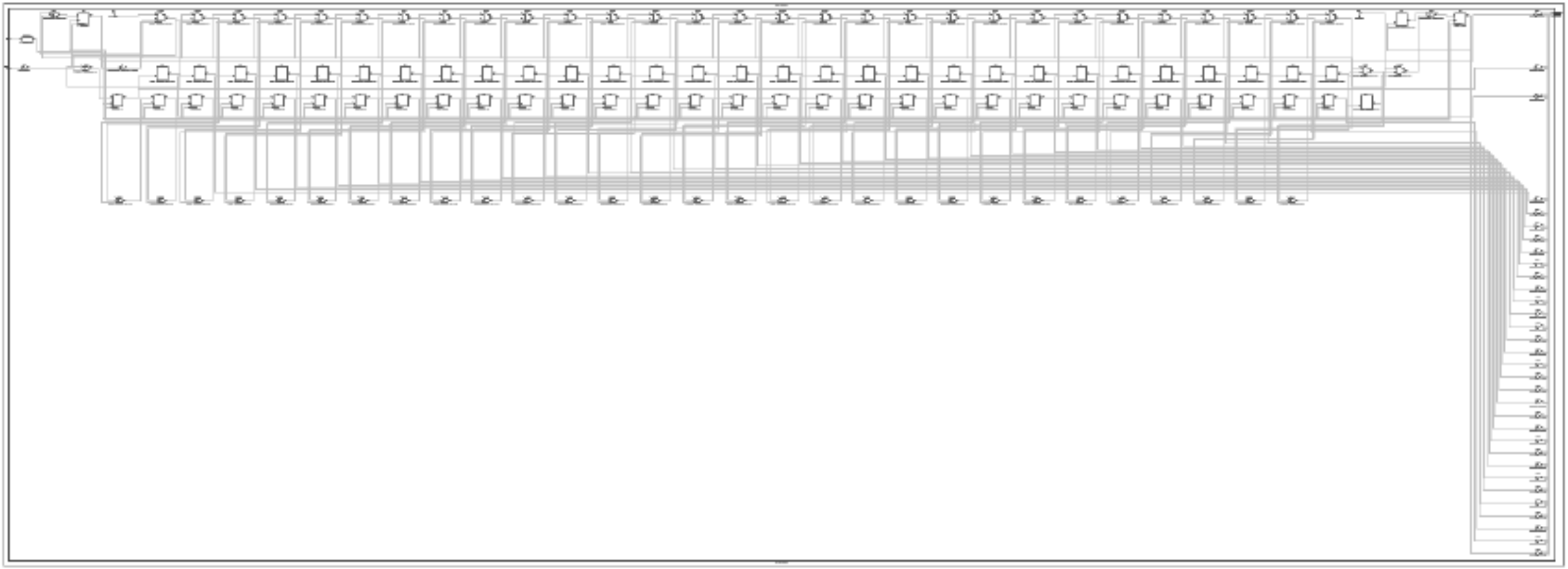
16 bit …

```
signal r_count, next_count : unsigned(15 downto 0);
```



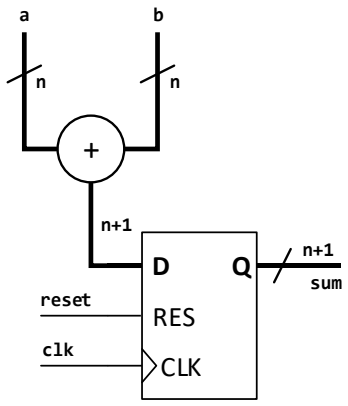
32 bit...

```
signal r_count, next_count : unsigned(31 downto 0);
```



HDL vs software

```
next_sum <= a+b;
r_sum <=
  (others => '0') when reset else
  next_sum when rising_edge(clk);
```



Hardware description languages	Software languages
Defines the logic function of a circuit	**Defines the sequence of instructions and which data shall be used** for one or more processors or processor cores
CAD tools *syntetizises* designs to enable realization using physical gates.	***A compiler translates program code to machine code instructions*** that the processor can read sequentially from memory
*Implemented in **Programmable Logic*** *(*FPGA, CPLD, PLD, PAL, PLA, …) or ***ASICs*** (application specific circuits) (“ASICs”, processors, ..-chips,.. etc.)	**Is stored in computer memory**
Verilog (SystemVerilog) **VHDL (VHDL 2008)** (System C m. fl.)	C, C++, C#, Python, Java, assemblere (ARM, MIPS, x86, …) Fortran, LISP, Simula, Pascal, osv…

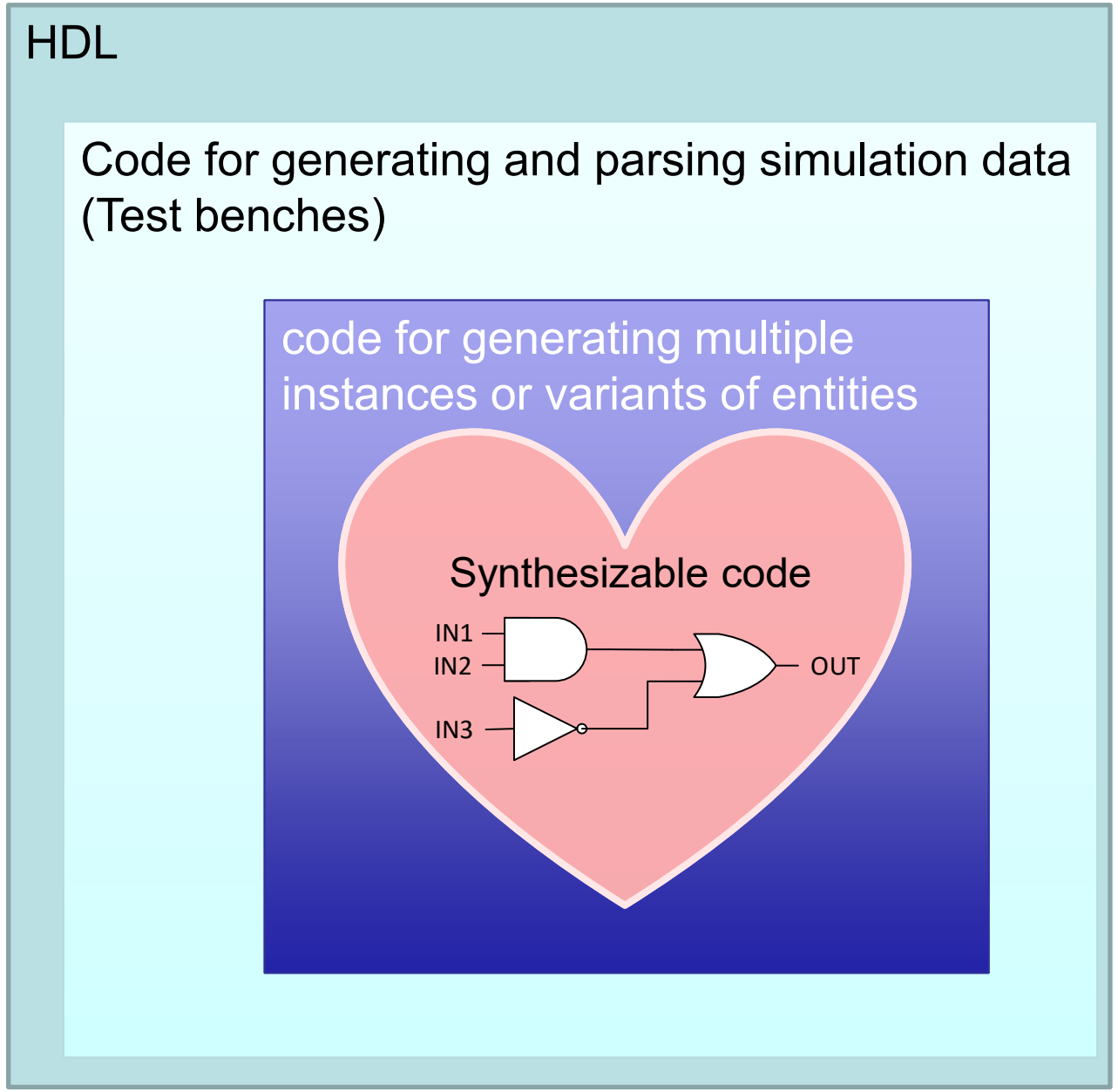
```
int sum(int a, int b){
  int s;
  s = a + b;
  return s;
}
```

```
sum(int, int):
 push   rbp
 mov    rbp,rsp
 mov    DWORD PTR [rbp-0x14],edi
 mov    DWORD PTR [rbp-0x18],esi
 mov    edx,DWORD PTR [rbp-0x14]
 mov    eax,DWORD PTR [rbp-0x18]
 add    eax,edx
 mov    DWORD PTR [rbp-0x4],eax
 mov    eax,DWORD PTR [rbp-0x4]
 pop    rbp
 ret
```

```
55
48 89 e5
89 7d ec
89 75 e8
8b 55 ec
8b 45 e8
01 d0
89 45 fc
8b 45 fc
5d
c3
```

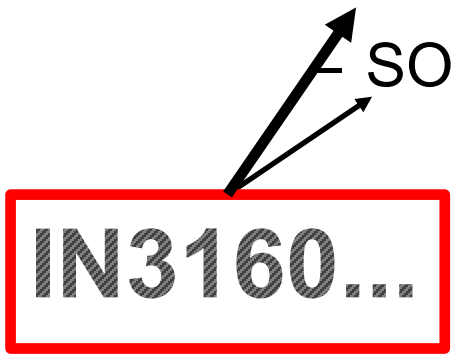
What is HDL

- VHDL = VHSIC HDL:
  - Very High Speed Integrated Circuit **Hardware Description Language**
  - The purpose is to generate circuits, and verify their function through simulation.
  - **Synthesizable** (realizable) **code work concurrently** (in parallel, always on).
  - Code for simulation include things such as file I/O which cannot be synthesized.
  - Testbenches can use synthesizable elements, but will use sequential statements, and is only run as software.
    This may be confusing at times...



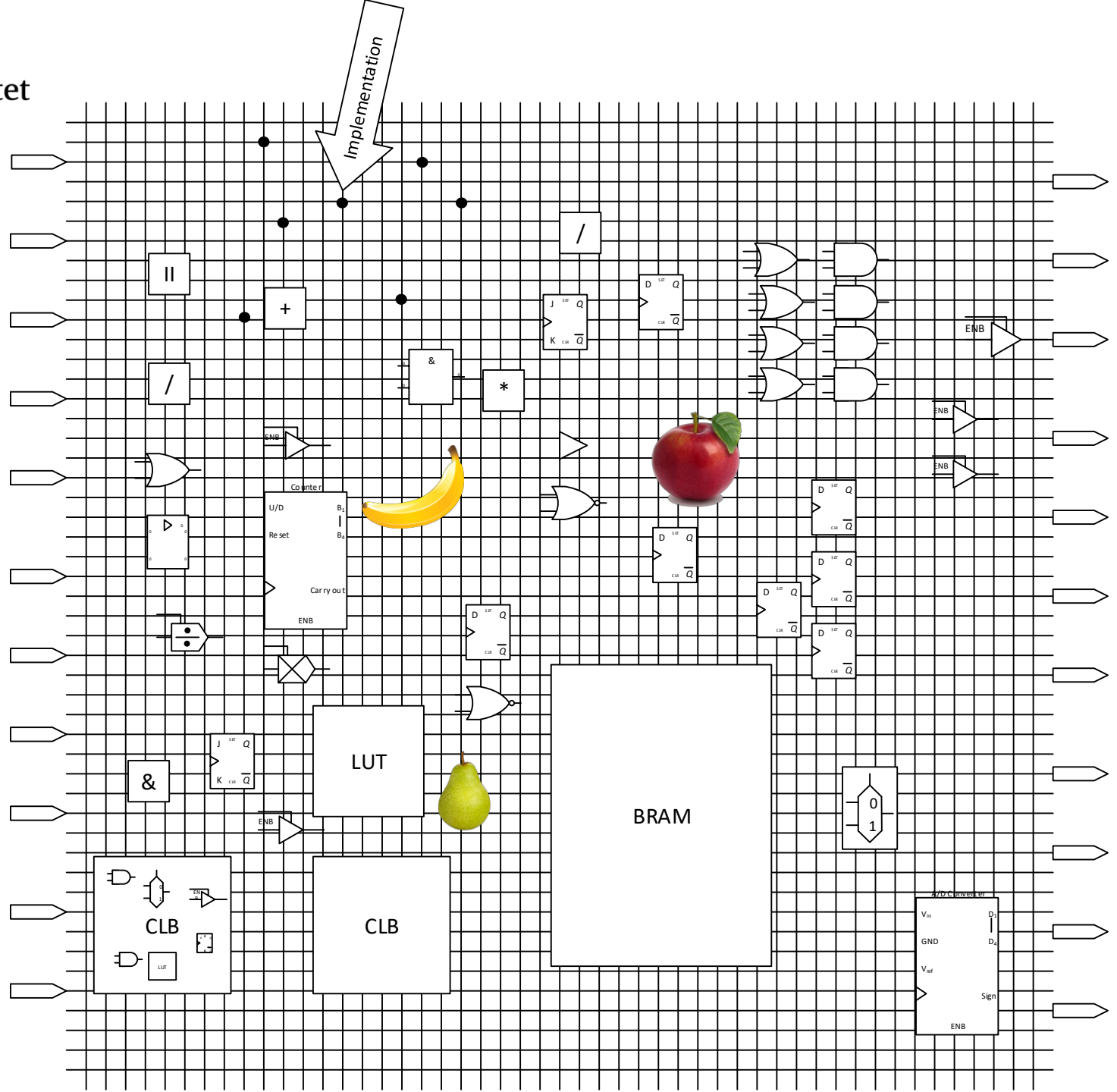
What hardware..?

- ***ASIC*** : Application Specific Integrated Circuit
  - Processors, microcontrollers, GPUs (x86, ARM, GeForce)
  - Customized chips
- ***PL*** : Programmable logic
  - PLA, PAL "Programmable Logic Array" / .."Array Logic"
  - CPLD "Complex Programmable Logic Device" = Several PAL/PLAs, FFs
  - FPGA "Field Programmable Gate Array" = More complex array of primitives
  - SOC's "System On Chip" = FPGA + micro –prosessor/-controller



What is PL and FPGA ? (Programmable Logic)

- PL = FPGA, CPLD, PLA…
  (Field Programmable Gate Array, Complex Programmable logic Device)
- PL vs processor (FPGA vs CPU, MCU) ?
- PL vs ASIC (Application Specific Integrated circuit)?

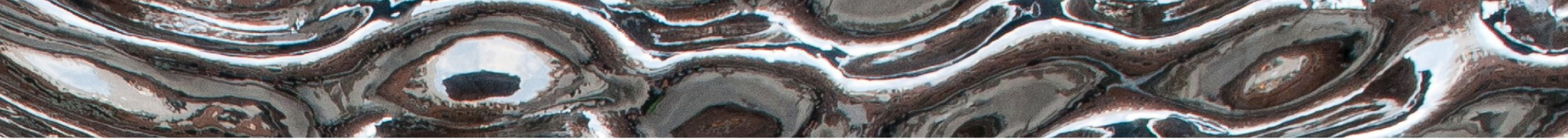


When or why choose programmable logic?

- (Verify behavior of ASIC)
- Prototyping flexibility
  - Lots of multi purpose IO
  - Reprogrammable
- Small batch production
- Parallellism
- Custom / fast
- Runtime reconfigurability
- …

When to avoid programmable logic?

- High Volume + low cost
- When dedicated HW is well suited.
- When extreme speed is required => ASIC
- …



UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

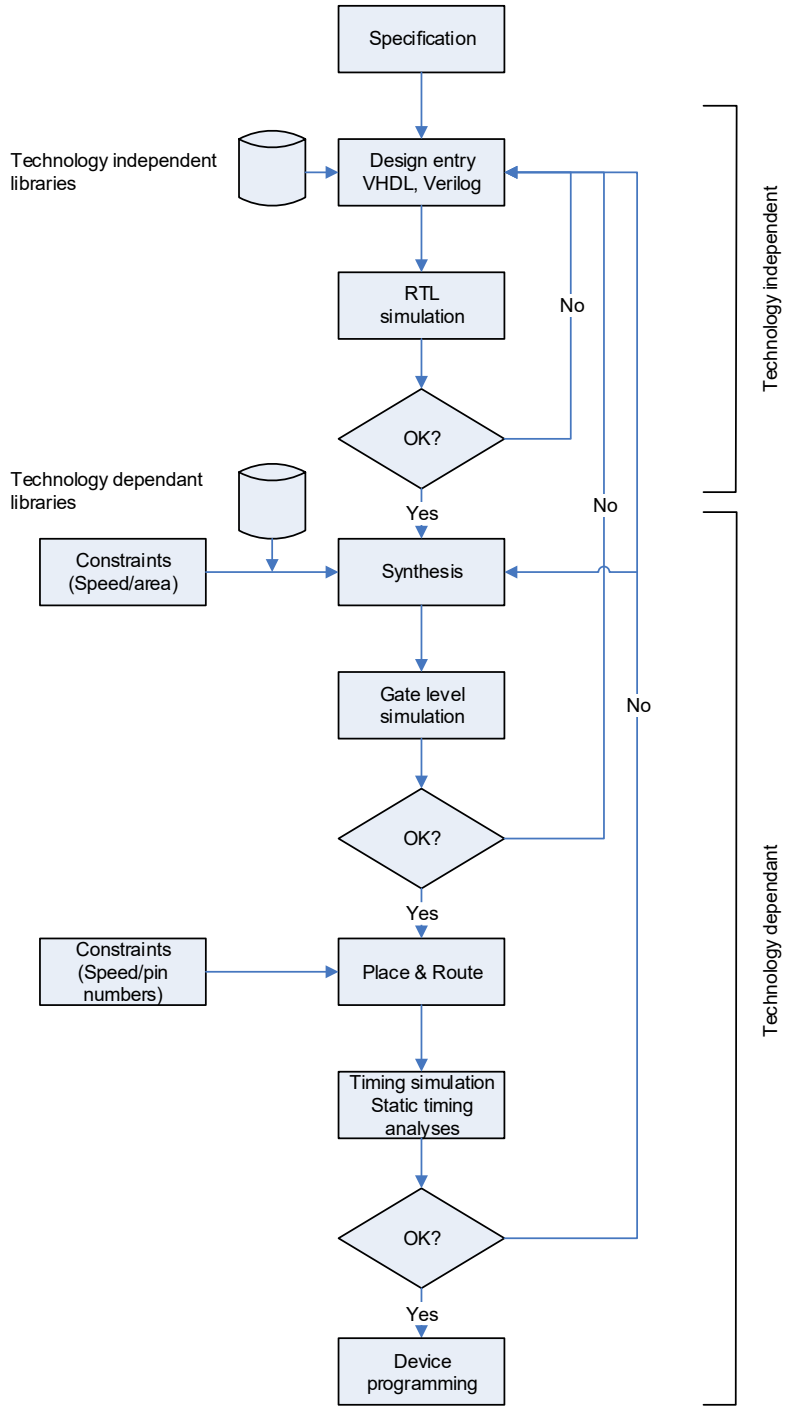
IN3160

Digital Design Flow

Yngve Hafting

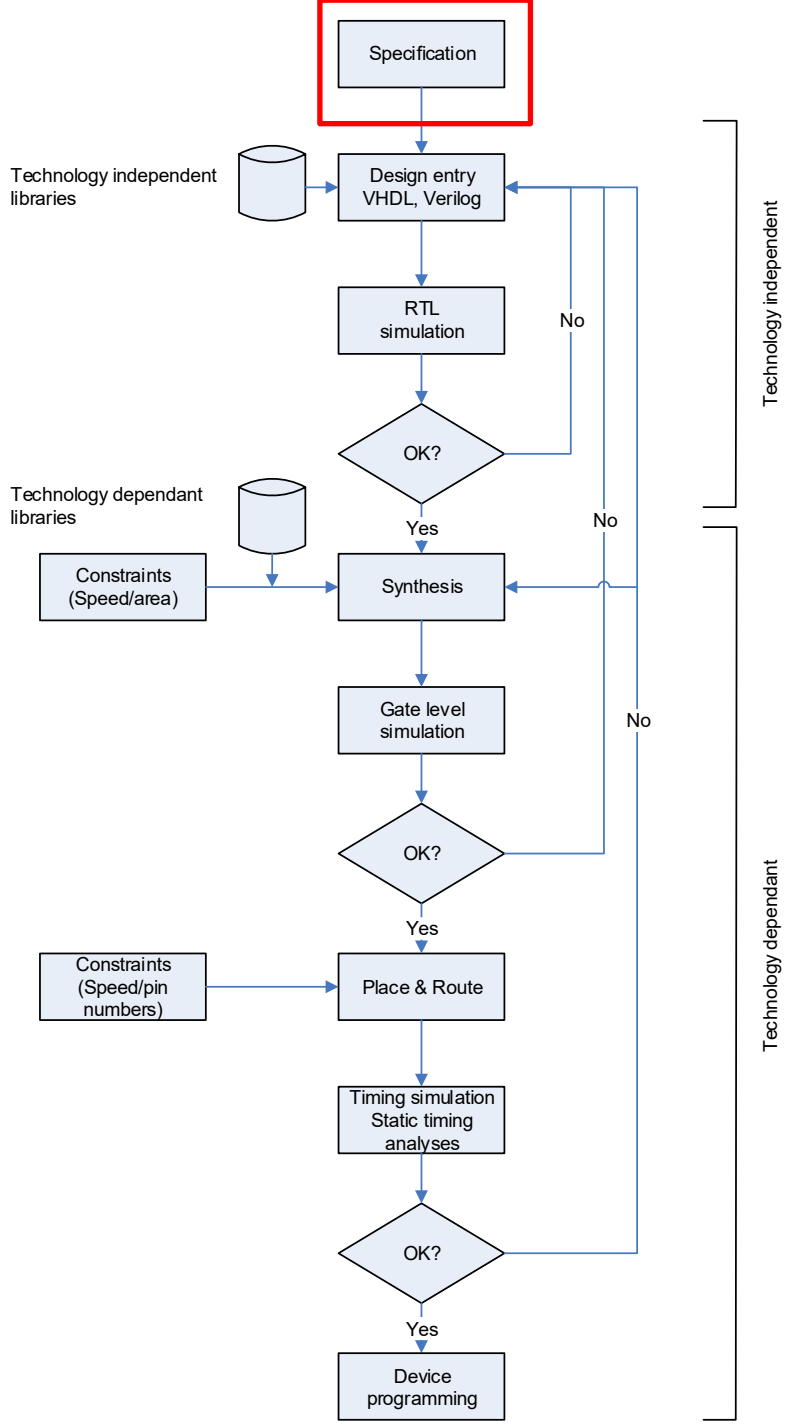
Overview

- Digital design tools.
- Specification
- Design entry, synthesis and PAR
- Timing analysis
- Timing simulation
- Testing



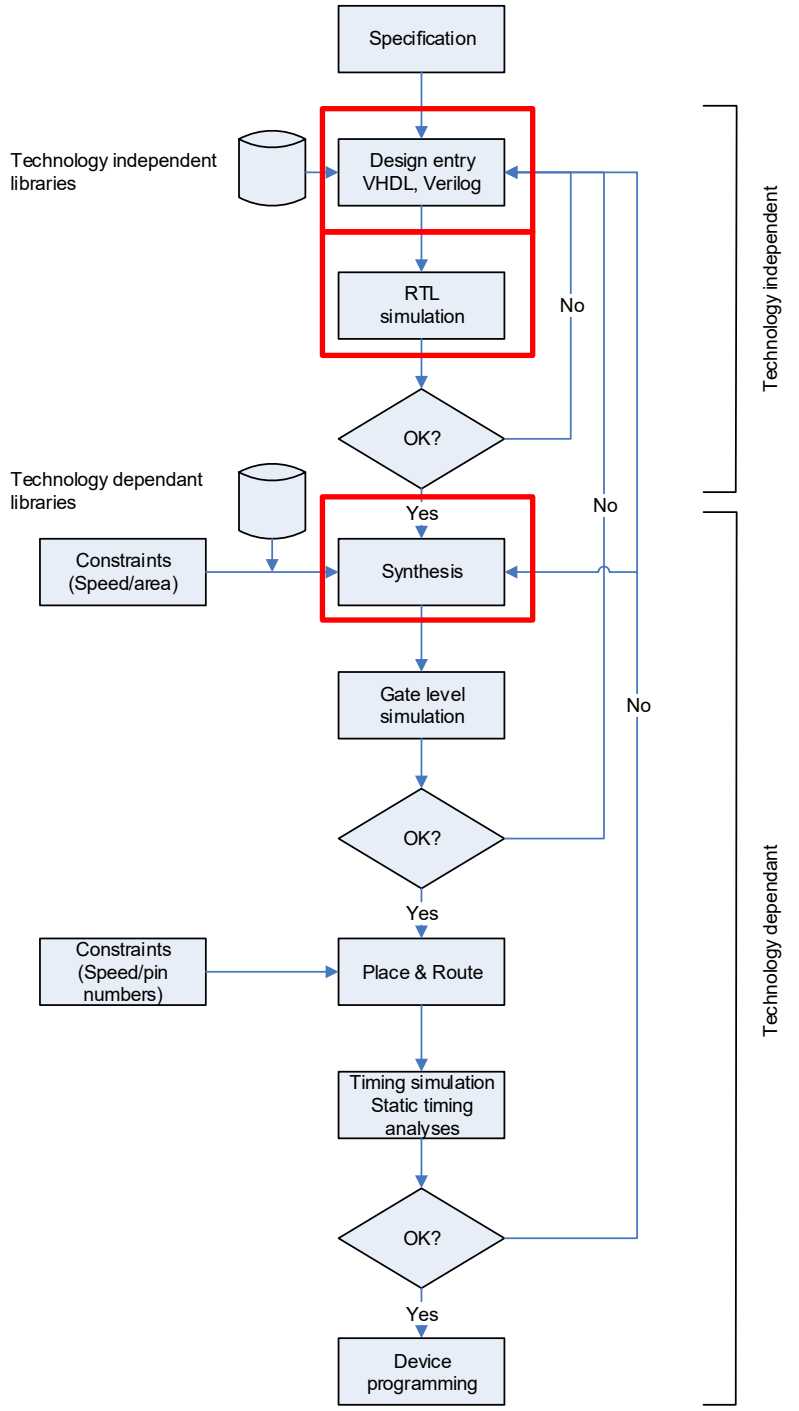
Digital Design Flow: Specification

1. Define the problem
2. Draw a functional diagram
   - block diagram with major components and connections
3. Identify IO requirements
4. Identify necessary interface circuits
5. Decide on HDL (VHDL, Verilog, System C,…)
6. *Draw a program flowchart (ASM diagram)*
   - Defines how the design shall work logically.
   - By hand or using tools such as:
     - *Visio, Draw.io, Lucid chart, etc.*



Digital Design tools…

- Design entry:
  - Use your favourite HDL text editor
    VScode, Notepad++, Emacs, ...
- Simulation (RTL)
  - Here: Typically using GHDL
- Synthesis, Implementation, Programming
  - Vendor specific tools,
    - Here: Vivado by Xilinx
  - Also possible: Digilent tools for programming.

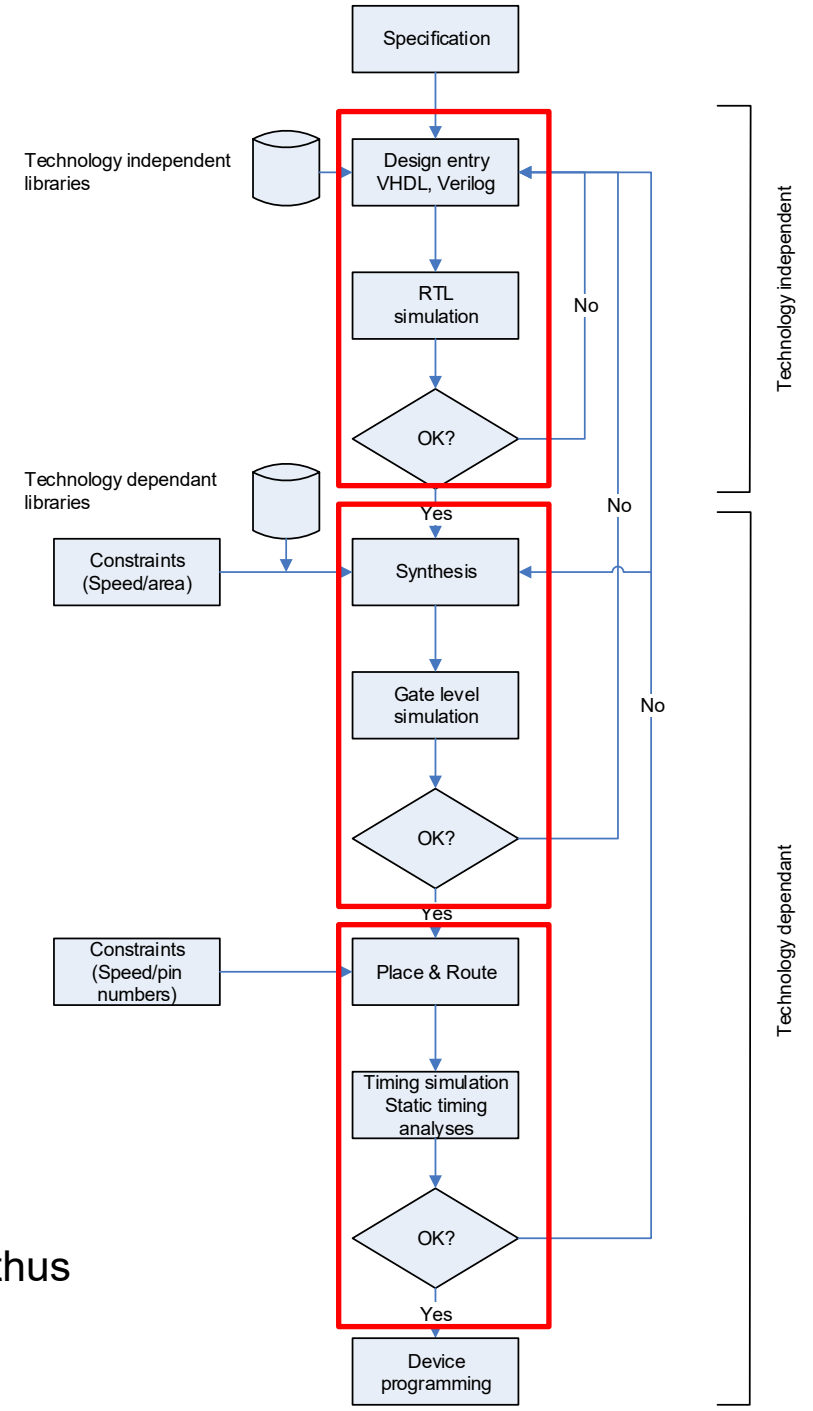


Design entry, synthesis and PAR

- RTL = Register Transfer Level
  - RTL does not use specific gates or technology
  - Designs are *mostly* done in RTL
  - RTL simulation can be used to verify logic function.

- Gate level synthesis
  - Technology specific gates are selected for all components in the design.
    - Typically a synthesizer will pick gates specific for the (FPGA) chip family we use.
  - Once we have a gate level design we can
    - calculate gate-, but not propagation delays
    - Simulate using gate delays.

- Place and route
  - After synthesis gates can be placed within a specific (FPGA) chip.
  - When place and route is performed propagation delays may also be simulated thus
  - We can do all timing simulation, including propagation delays.



Static timing analysis

- Performed by EDA tools on synthesized or routed designs
- Will attempt to
  - find critical path(s) and
  - check if timing requirements (constraints) can be met.

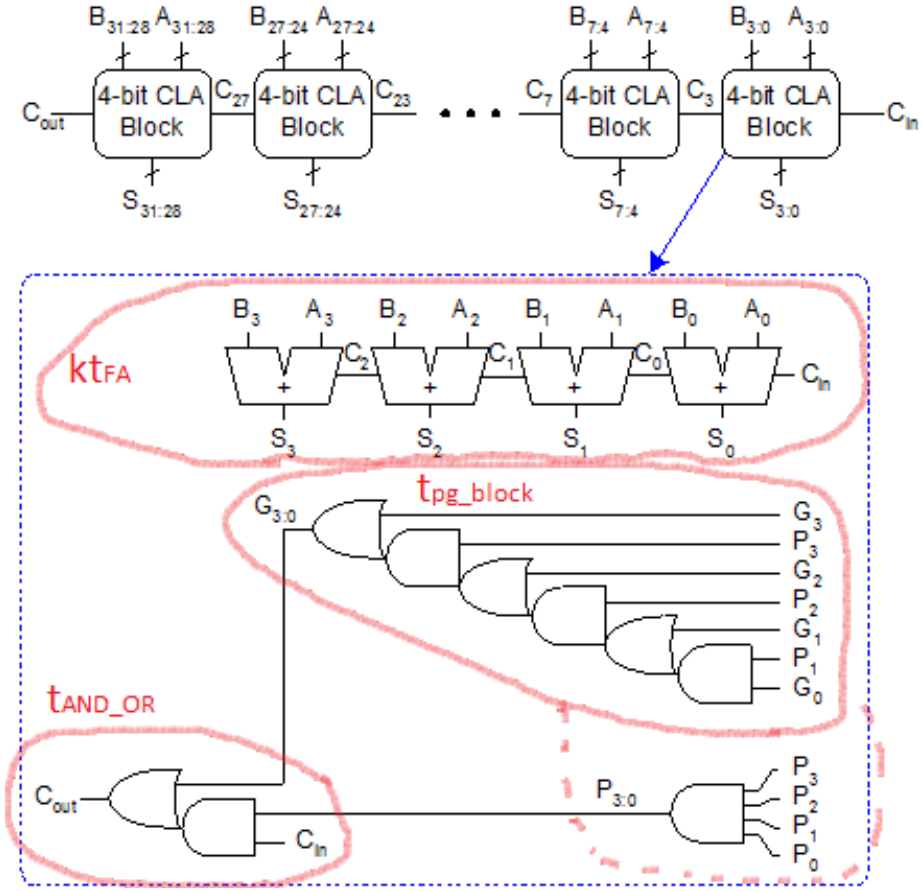
IN2060: Carry-Lookahead Adder Delay

For *N*-bit CLA with *k*-bit blocks:

$$t_{CLA} = t_{pg} + t_{pg\_block} + (N/k - 1)t_{AND\_OR} + kt_{FA}$$

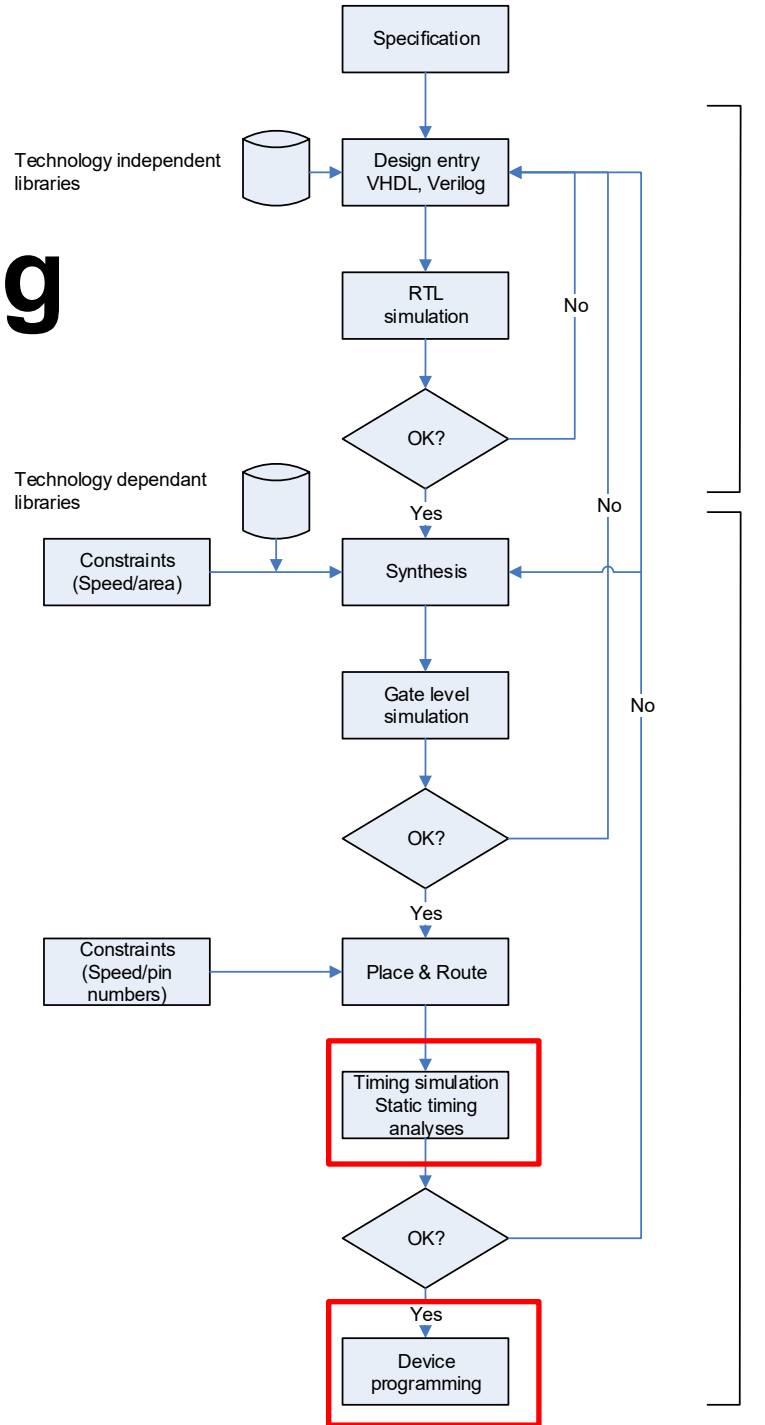
- t_{pg} : delay to generate all P_i, G_i
- t_{pg_block} : delay to generate all $P_{i:j}$, $G_{i:j}$
- t_{AND_OR} : delay from C_{in} to C_{out} of final AND/OR gate in *k*-bit CLA block

An *N*-bit carry-lookahead adder is generally much faster than a ripple-carry adder for $N > 16$



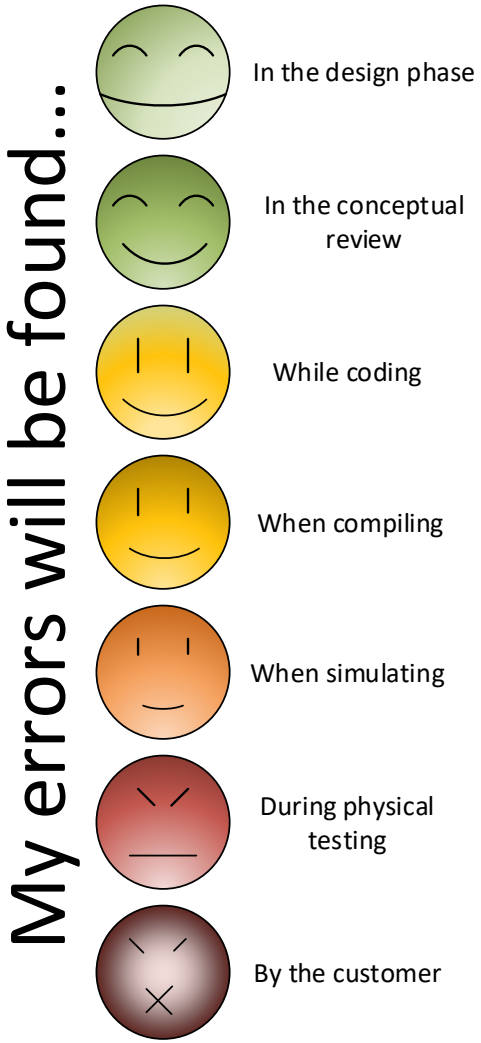
Timing simulation, programming

- Simulating synthesized or routed designs
  - Not common for FPGA design, but for ASIC and analog
    - *IN3160 does not use timing simulation:*
      Static timing analysis is mostly sufficient
  - Uses timing information for every component in use.
    - Requires much more resources than RTL simulation.
    - Can be slow for complex designs
      - Hence the option to simulate at gate level, before performing PAR.
- Device programming…
  - (Usually done from vivado, but third part tools *may* be used).
  - Download bit stream to FPGA



Testing and verification

- «*Testing*» is to find *physical errors* in a *device*.
  - Built in self-tests
    - Ex: Memory tests in BIOS
  - Design for testability
    - Means that we design for physical testing.
    - We *may* touch this later in the course.
- «*Verification*» is to check the *design*
  - Reading the code...
  - Simulation
    - Testbenches
      - HDL
      - Scripts
      - Co-simulation using normal programming languages
  - Analysis:
    - Compilation
    - Timing Analysis
    - Implementation reports
- *Spend more time in early phases!*



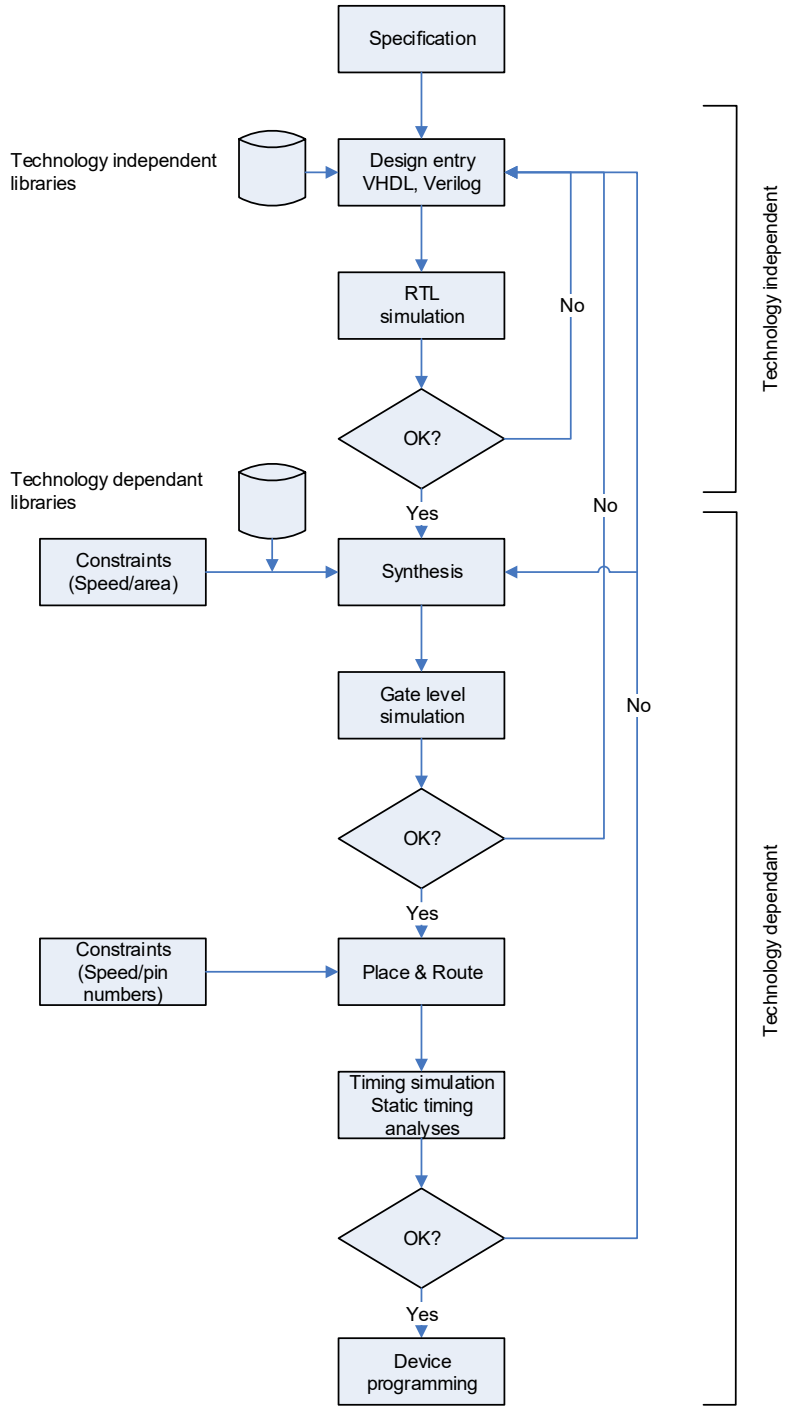
Introduction to course hardware and software tools

- Zedboard
- Questa
- Vivado

- ROBIN wiki:
  https://robin.wiki.ifi.uio.no/Hovedside
  - Software
    - FPGA tools
      https://robin.wiki.ifi.uio.no/FPGA_tools (See VSCode)
    - https://robin.wiki.ifi.uio.no/Cocotb,_GHDL_and_GTKWave
- Cook book and ZedBoard documentation
  - Canvas – IN3160
    - Cookbook_v3_5.pdf
    - ZedBoard HW UG vX_X.pdf
  - Zynq intro video:
    https://www.xilinx.com/video/soc/zedboard-overview-featuring-zynq.html

Digital Design tools…

- Design entry:
  - Use your favourite HDL text editor
    Vscode Notepad++, Emacs, Vivado.
- Simulation (RTL, Gate Level, Timing)
  - GHDL
  - (Questa= Modelsim)
- Synthesis, Implementation, Programming
  - Vendor specific tools…
    - Here: Vivado by AMD/Xilinx, (*Vitis for SoC designs*)



Simulation and test benches

Simulation can be run several ways:

1. Manually setting inputs and specifying time intervals in the GUI or console
   - Tedious and not really practical at all
   - *Normally this is only done only initially.*

2. To make scripts (tcl for Questa) in a separate (.do) file.
   - The *script commands will be added to the console during manual use, and can be copied as text into a .do file.*
   - setting up the simulation windows can be done reusing script commands.

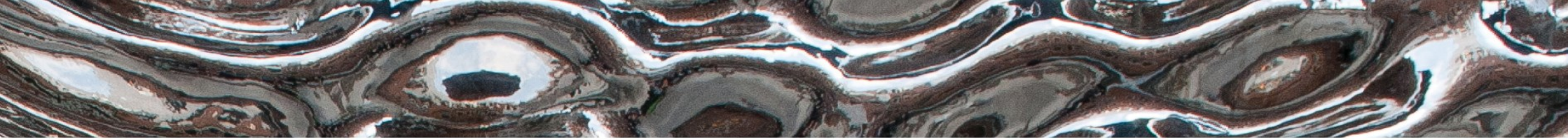
3. Using test benches written in HDL
   - possible in combination with running scripts
   - VHDL can be used to generate code for applying test vectors sequentially to the inputs of an entity for simulating.
   - *Test bench code is SW even if it is written in an HDL* (not synthesizable)
   - VHDL has built in test-specific attributes
4. Using Co-Simulation (cocotb)
   - runs simulation and coroutines in parallell
     - Environment switches back and forth between coroutines and simulation
   - Test vectors are generated in software coroutines
   - Checks and reporting is done in coroutines
   - Python can be used for test-benches
     - not built for hardware testing initially
     - scale better with large design and complex testing
     - non-HDL-specific issues has (way) better support

Suggested reading, Mandatory assignments

- D&H:
  - 1.4 p 11-13
  - 1.5 p 13-16
  - 1.6 p 16-17
  - 2.1 p 22-28
  - 2.2 p 28-30
  - 2.3 p 30-34
  - 3.1-3.5 p 43-51 = repetition (known from previous courses)

- Oblig 1: «Design Flow»
  - See canvas for further instruction.

Note: Some of this content will be covered in depth in later lectures.
- *Read this to familiarize yourself with content, form and language.*



UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

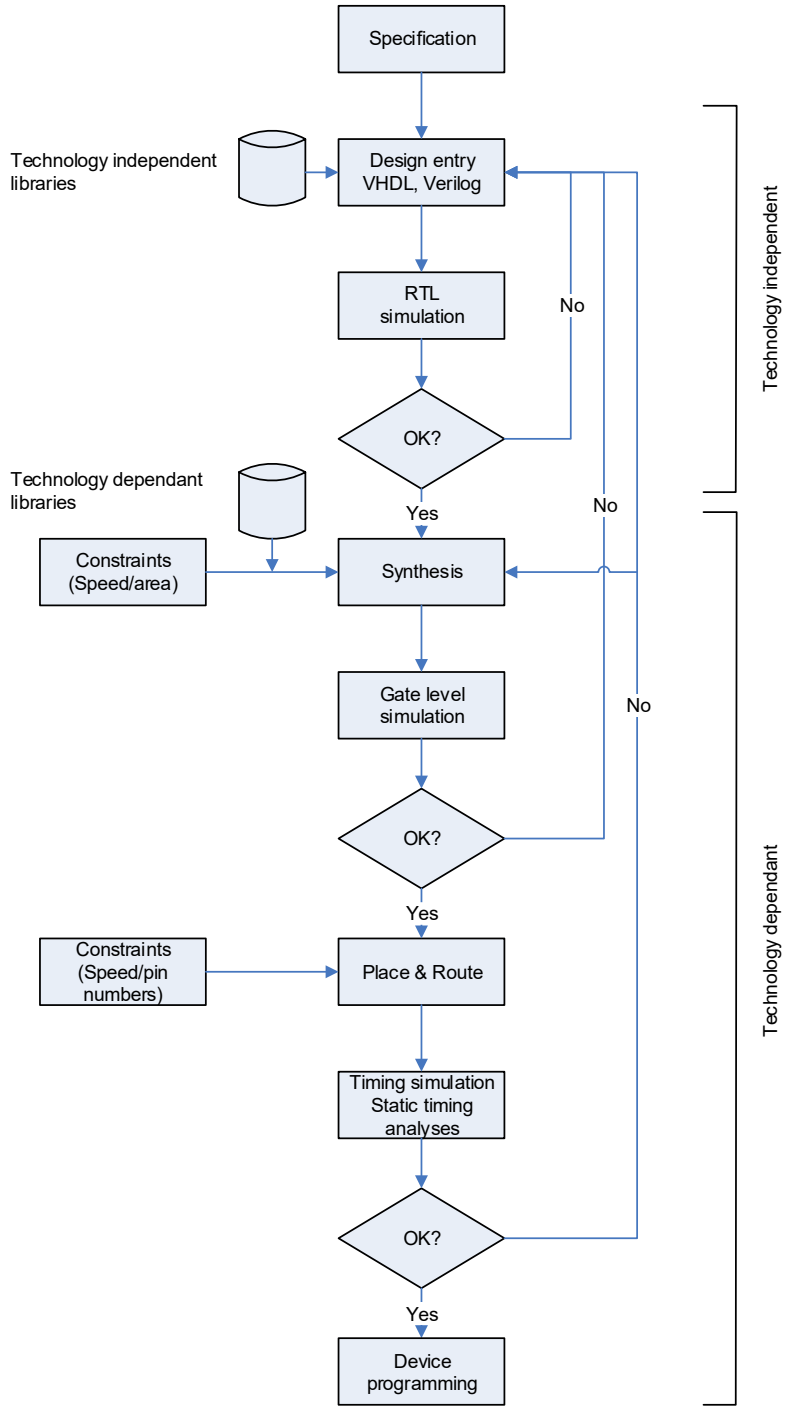
IN3160

Digital Design Flow

Yngve Hafting

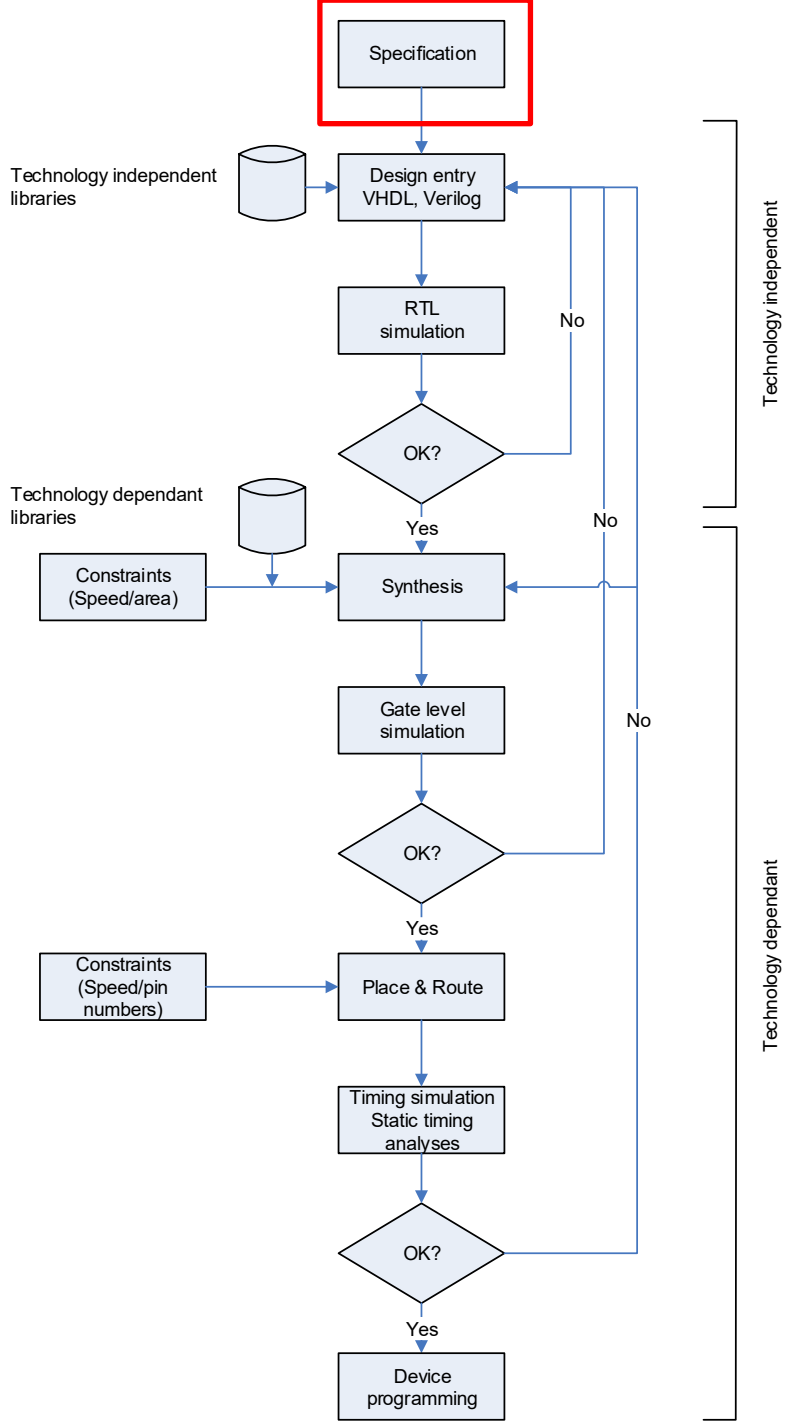
Overview

- Digital design tools.
- Specification
- Design entry, synthesis and PAR
- Timing analysis
- Timing simulation
- Testing



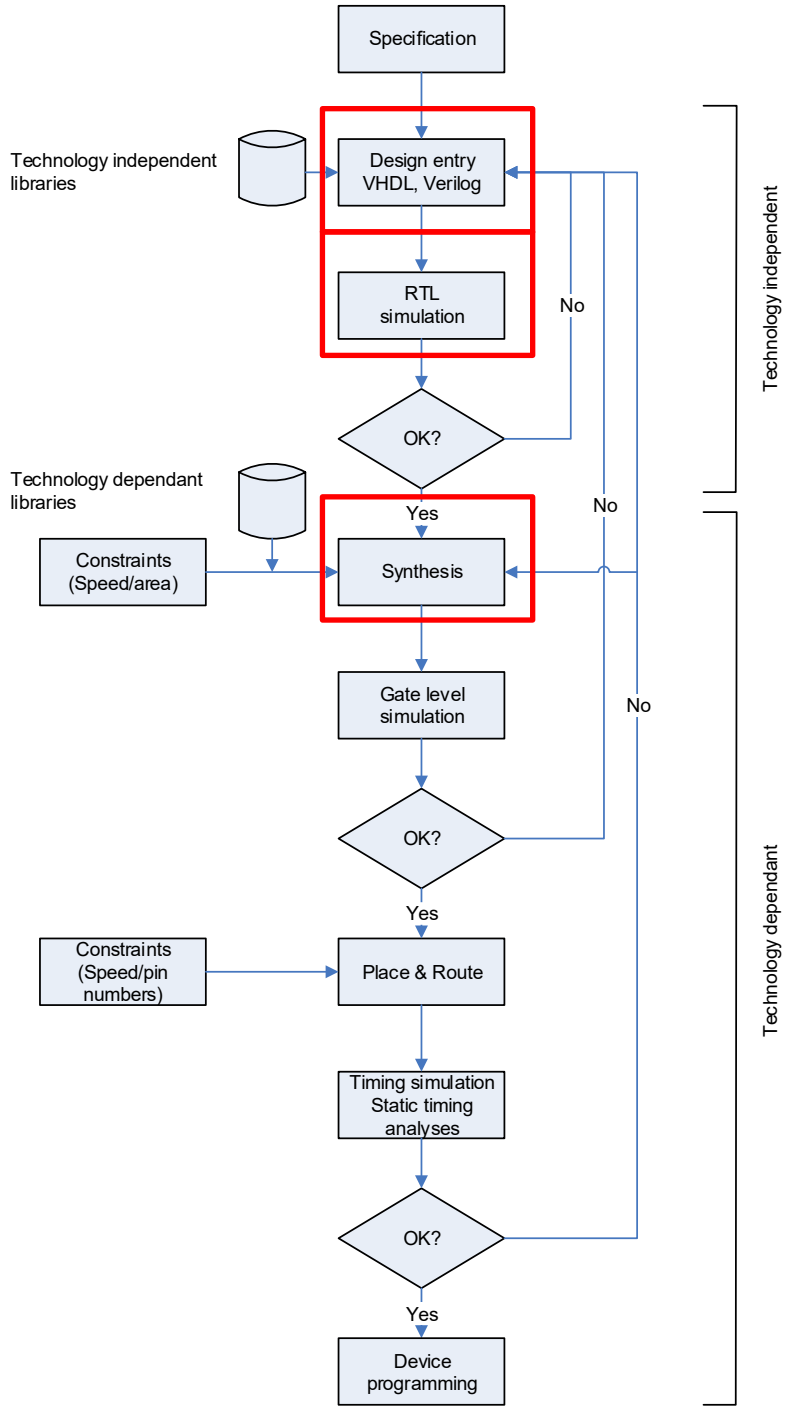
Digital Design Flow: Specification

1. Define the problem
2. Draw a functional diagram
   - block diagram with major components and connections
3. Identify IO requirements
4. Identify necessary interface circuits
5. Decide on HDL (VHDL, Verilog, System C,…)
6. *Draw a program flowchart (ASM diagram)*
   - Defines how the design shall work logically.
   - By hand or using tools such as:
     - *Visio, Draw.io, Lucid chart, etc.*



Digital Design tools…

- Design entry:
  - Use your favourite HDL text editor
    VScode, Notepad++, Emacs, ...
- Simulation (RTL)
  - Here: Typically using GHDL
- Synthesis, Implementation, Programming
  - Vendor specific tools,
    - Here: Vivado by Xilinx
  - Also possible: Digilent tools for programming.

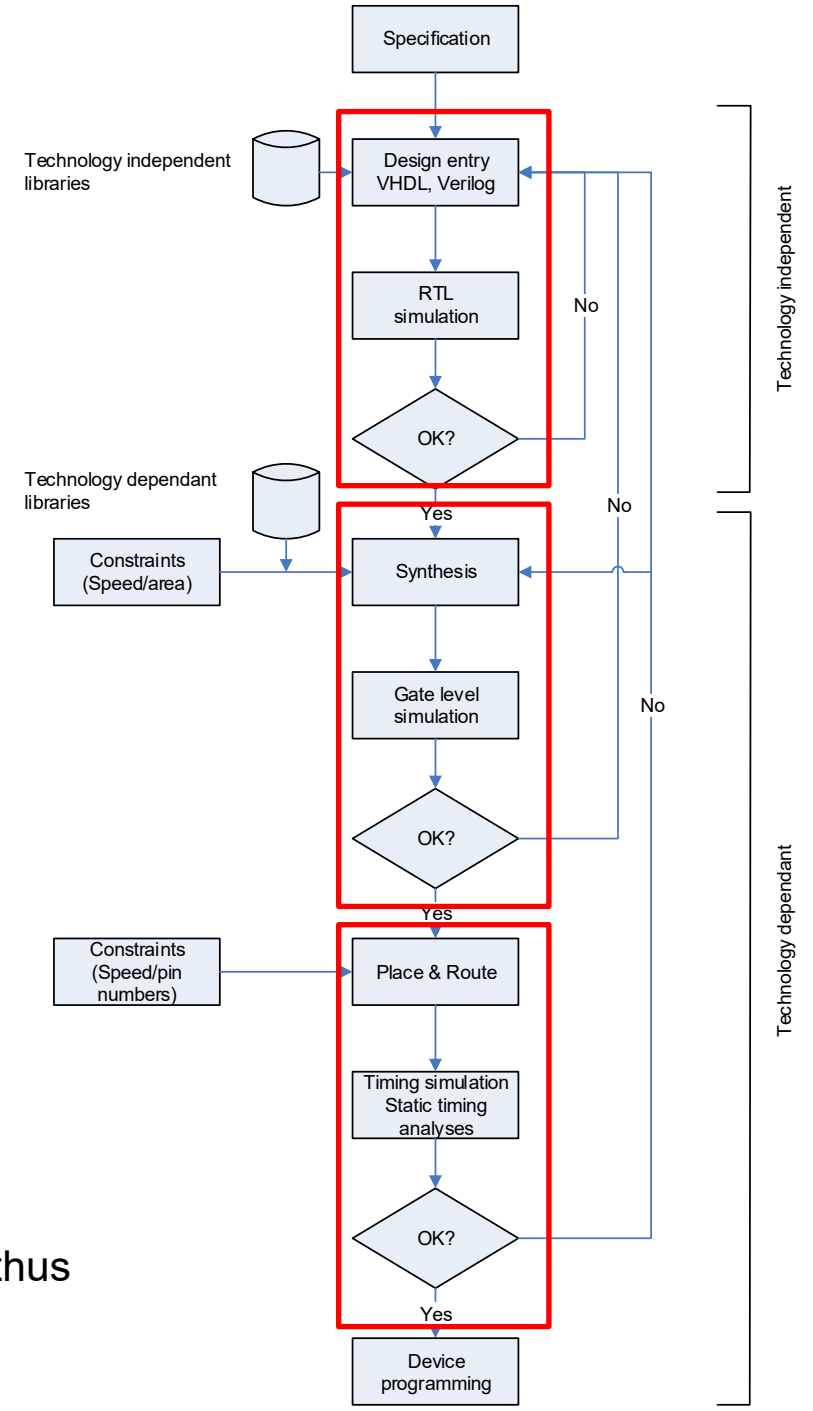


Design entry, synthesis and PAR

- RTL = Register Transfer Level
  - RTL does not use specific gates or technology
  - Designs are *mostly* done in RTL
  - RTL simulation can be used to verify logic function.

- Gate level synthesis
  - Technology specific gates are selected for all components in the design.
    - Typically a synthesizer will pick gates specific for the (FPGA) chip family we use.
  - Once we have a gate level design we can
    - calculate gate-, but not propagation delays
    - Simulate using gate delays.

- Place and route
  - After synthesis gates can be placed within a specific (FPGA) chip.
  - When place and route is performed propagation delays may also be simulated thus
  - We can do all timing simulation, including propagation delays.



Static timing analysis

- Performed by EDA tools on synthesized or routed designs
- Will attempt to
  - find critical path(s) and
  - check if timing requirements (constraints) can be met.

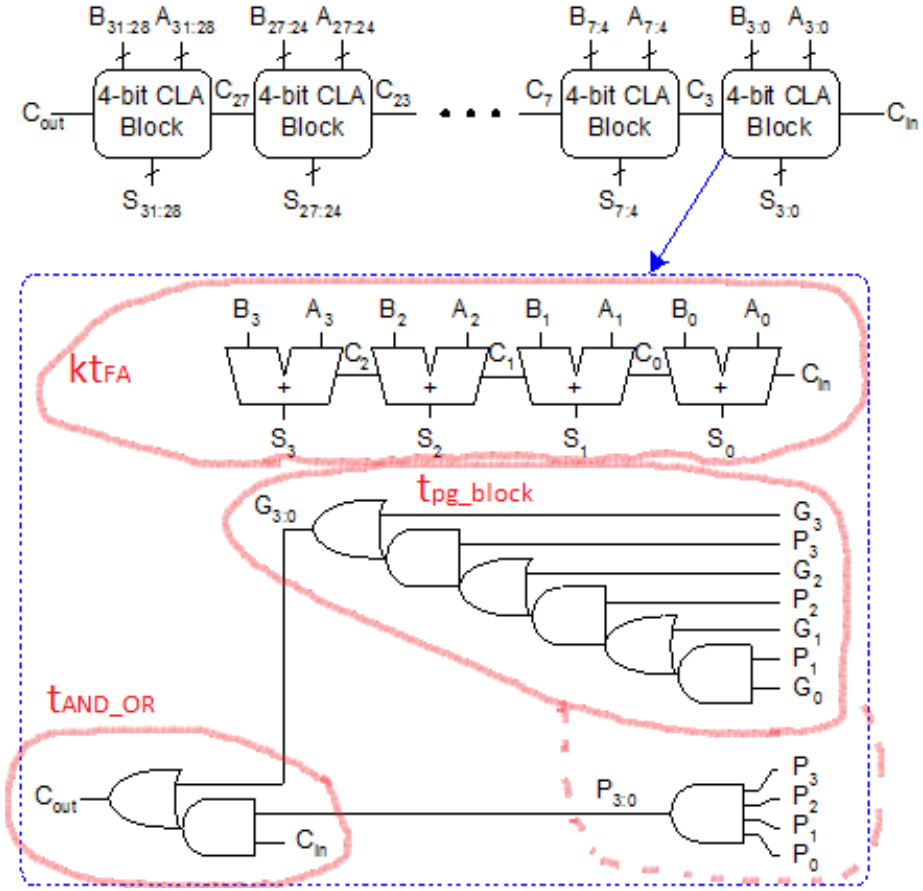
IN2060: Carry-Lookahead Adder Delay

For *N*-bit CLA with *k*-bit blocks:

$$t_{CLA} = t_{pg} + t_{pg\_block} + (N/k - 1)t_{AND\_OR} + kt_{FA}$$

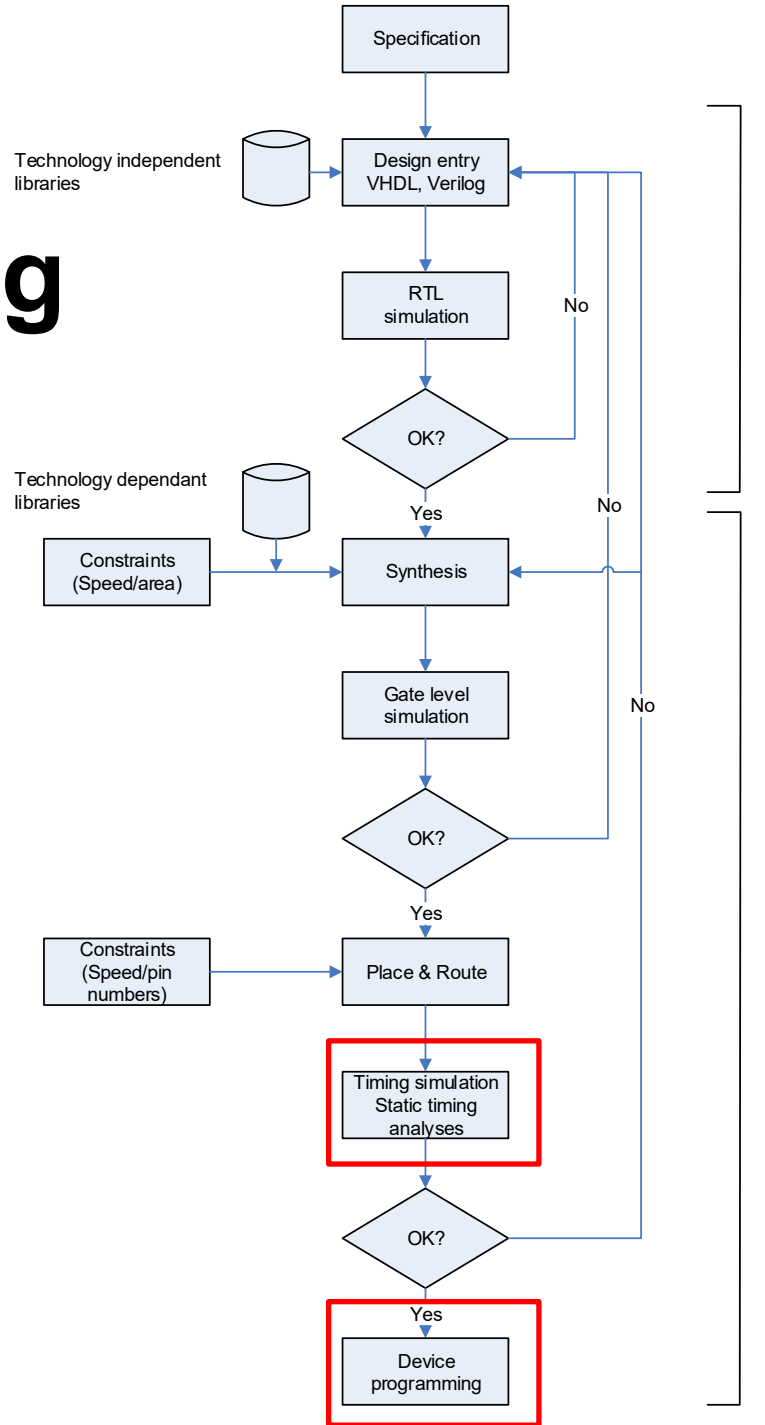
- t_{pg} : delay to generate all P_i, G_i
- t_{pg_block} : delay to generate all $P_{i:j}$, $G_{i:j}$
- t_{AND_OR} : delay from C_{in} to C_{out} of final AND/OR gate in *k*-bit CLA block

An *N*-bit carry-lookahead adder is generally much faster than a ripple-carry adder for $N > 16$



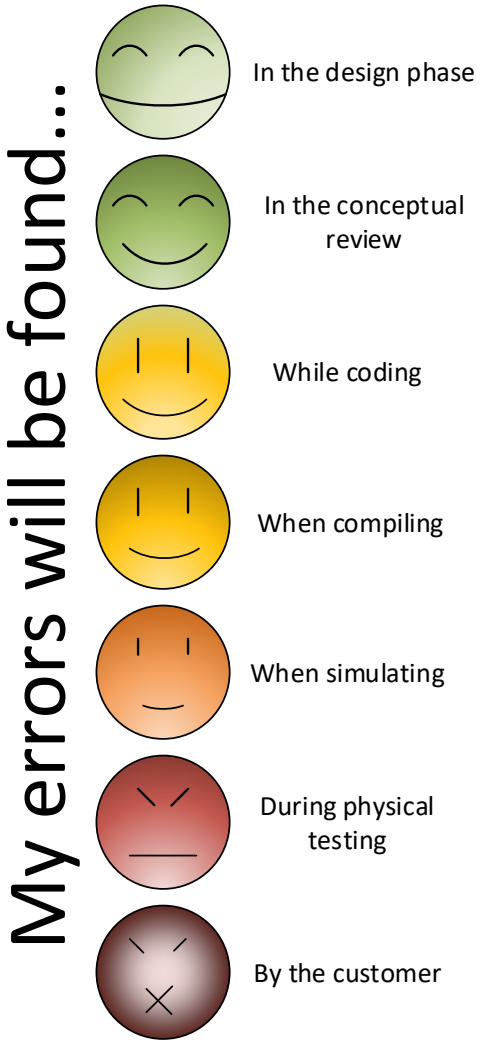
Timing simulation, programming

- Simulating synthesized or routed designs
  - Not common for FPGA design, but for ASIC and analog
    - *IN3160 does not use timing simulation:*
      Static timing analysis is mostly sufficient
  - Uses timing information for every component in use.
    - Requires much more resources than RTL simulation.
    - Can be slow for complex designs
      - Hence the option to simulate at gate level, before performing PAR.
- Device programming…
  - (Usually done from vivado, but third part tools *may* be used).
  - Download bit stream to FPGA



Testing and verification

- «*Testing*» is to find *physical errors* in a *device*.
  - Built in self-tests
    - Ex: Memory tests in BIOS
  - Design for testability
    - Means that we design for physical testing.
    - We *may* touch this later in the course.
- «*Verification*» is to check the *design*
  - Reading the code...
  - Simulation
    - Testbenches
      - HDL
      - Scripts
      - Co-simulation using normal programming languages
  - Analysis:
    - Compilation
    - Timing Analysis
    - Implementation reports
- *Spend more time in early phases!*



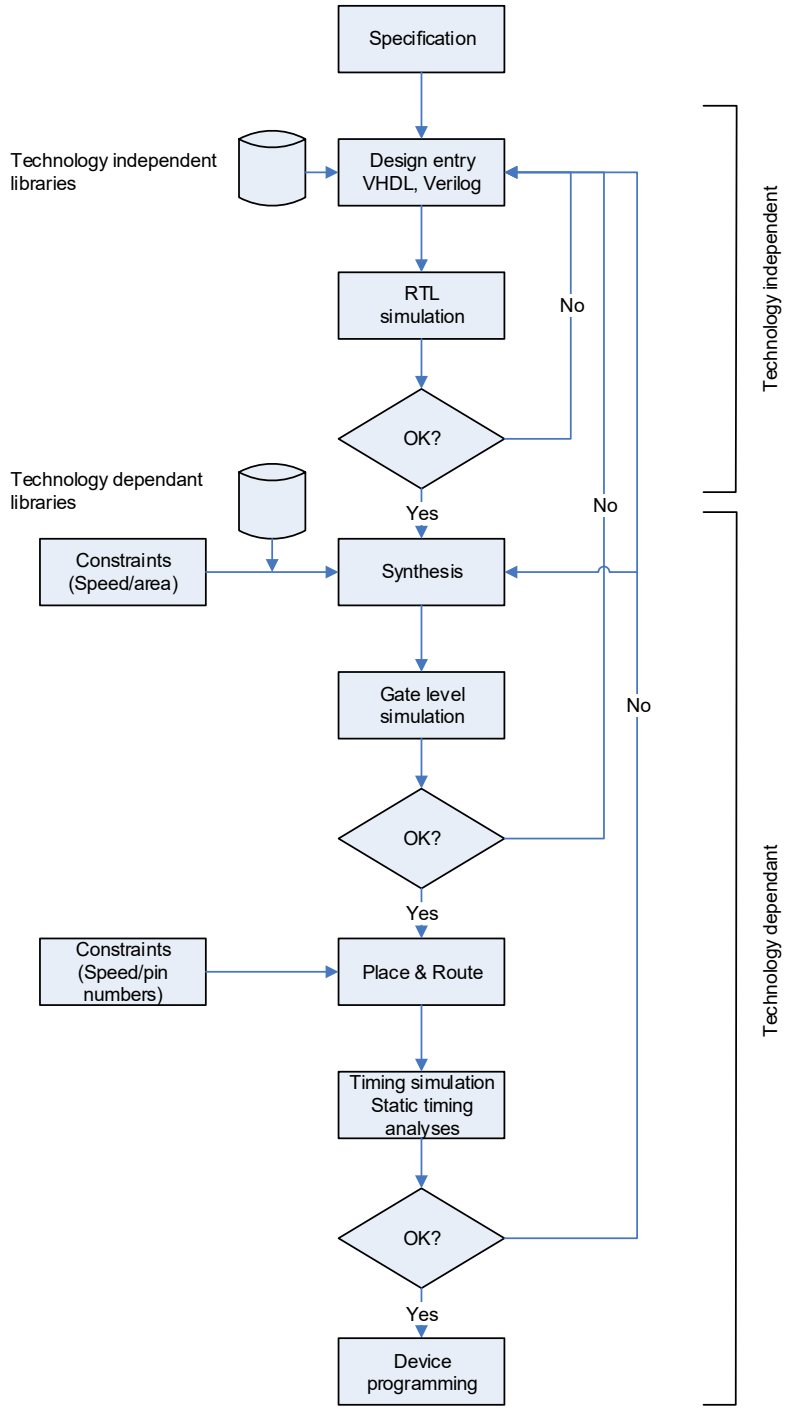
Introduction to course hardware and software tools

- Zedboard
- Questa
- Vivado

- ROBIN wiki:
  https://robin.wiki.ifi.uio.no/Hovedside
  - Software
    - FPGA tools
      https://robin.wiki.ifi.uio.no/FPGA_tools (See VSCode)
    - https://robin.wiki.ifi.uio.no/Cocotb,_GHDL_and_GTKWave
- Cook book and ZedBoard documentation
  - Canvas – IN3160
    - Cookbook_v3_5.pdf
    - ZedBoard HW UG vX_X.pdf
  - Zynq intro video:
    https://www.xilinx.com/video/soc/zedboard-overview-featuring-zynq.html

Digital Design tools…

- Design entry:
  - Use your favourite HDL text editor
    Vscode Notepad++, Emacs, Vivado.
- Simulation (RTL, Gate Level, Timing)
  - GHDL
  - (Questa= Modelsim)
- Synthesis, Implementation, Programming
  - Vendor specific tools…
    - Here: Vivado by AMD/Xilinx, (*Vitis for SoC designs*)



Simulation and test benches

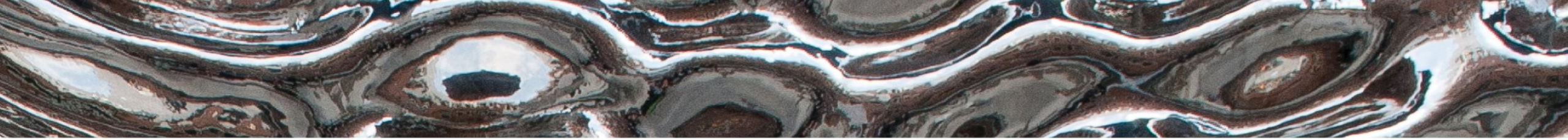
- Four different approaches:
  1. Manually setting inputs and specifying time intervals in the GUI or console
     - This way is tedious if much testing is to be done. Normally this is only done initially.
  2. Run the simulator using scripts (tcl) {Tikl}
     - Automating 1.
     - Can be combined with other solutions
  3. Create a test bench in VHDL
     - *This was the preferred method for IN3160 up to 2023*
     - *VHDL 2008 is created with testing in mind*
       - *Test benches and simulation code is software..!*
         - » *Can be confusing*
       - *Uses simulator only = runs fast*
  4. *Co simulating using or python)*
     (possible in combination with running scripts)
     - This is the preferred method
     - VHDL can be used to generate code for applying test vectors sequentially to the inputs of an entity for simulating.
     - Test bench code is not synthesizable
     - easy to read and use test data for each particular design,
     - Can be used both prior and post synthesis or implementation

Suggested reading, Mandatory assignments

- D&H:
  - 1.4 p 11-13
  - 1.5 p 13-16
  - 1.6 p 16-17
  - 2.1 p 22-28
  - 2.2 p 28-30
  - 2.3 p 30-34
  - 3.1-3.5 p 43-51 = repetition (known from previous courses)

- Oblig 1: «Design Flow»
  - See canvas for further instruction.

Note: Some of this content will be covered in depth in later lectures.
- *Read this to familiarize yourself with content, form and language.*



UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

IN3160, IN4160

Infrastructure and tool introduction

Yngve Hafting 2022

Overview

- Demo
- VSCode setup info
- Remote access solution
  - Vmware / VDI
    - Ifi Digital Electronics

- Assignments and suggested reading for this week

Demos (Own PC / VmWare)

- Try :
  - Live-demo for cocotb + gtkwave
    - Show how to
      - Run
      - Display Waveform in gtkwave (+ analog step)
      - Modify vhdl / errors
      - Modify python to get weird behavior
  - Vivado + vhdl-eksempel or live-demo
    - elaborate and create schematic
    - Synthesize
    - New project?

VSCode with VHDL?

- VHDL using VSCode
  - https://robin.wiki.ifi.uio.no/FPGA_tools#VHDL_using_VSCode

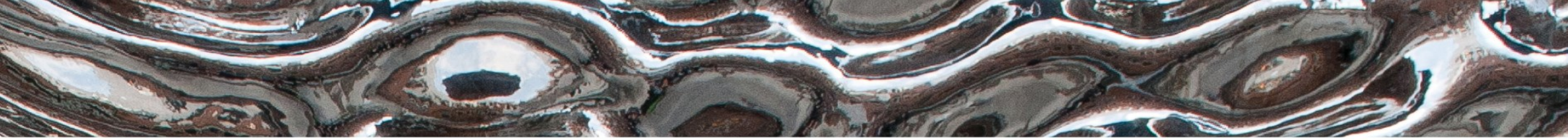
- Notepad ++
  - https://notepad-plus-plus.org

Access through VDI

- https://www.mn.uio.no/ifi/tjenester/it/hjelp/
  - => Linux->Virtuell arbeidsstasjon (VDI) Virtual Desktop Interface(?)
    - If you don't have the Vmware client already, check
      - https://www.uio.no/tjenester/it/maskin/vdi/hjelp/vdi-installer-og-bruk.html
  - Start Vmware horizon client
    - Enter view.uio.no
    - Use ifi-Digital-Electronics
      - *Run vivado, make og gtkwave from Xterm / command line*
  - Ifi digital electronics can be used for everything... *until programming*

Remote access using Xwin?

- login.ifi.uio.no
  - Ssh -Y
  - X-win



UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

IN3160, IN4160

Introduction to VHDL
+Basic layout for VHDL

Yngve Hafting

Messages:

- Timeplan -> Im working on it...
- Questions before we start?
- Today:
  - VHDL structure
  - Layout (only in slides) <- Hvis tid– også egnet for selvstudium.
- Friday: Basic verification and test-benches

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 this lesson:

- Know the basic structure of VHDL
  - Know which design entities there are
  - Know how assignment and statements works
  - Know the basic functionality of processes
  - Be able to create designs using VHDL
  - Know the relation between phsyical signals and their declaration.
  - Know the difference between basic coding styles
- Know basic layout principles
  - Guidelines for capital letters
  - Basic layout types
  - Principles for indentation, commenting, naming, punctuations

Overview

- Repetition
- VHDL Structure (*Partly repetition from IN2060*)
  - Design entities
  - Statements
  - Signals, variables, vectors
  - Processes
  - Libraries
  - STD_LOGIC
  - Operators
  - String literals
- Code layout principles
- Next lesson: Combinational logic

- Assignments and suggested reading for this week

HDL

- VHDL = VHSIC HDL:
  - Very High Speed Integrated Circuit **Hardware Description Language**
  - **The purpose is to generate digital circuits, and verify their function through simulation.**
  - **Synthesizable (realizable) code generates circuits that are always on = work concurrently (in parallel)**.
  - Code for simulation include things such as file I/O which cannot be synthesized.
  - Testbenches can and will use some synthesizable elements, but will in general look more like other sequential languages, and use sequential statements. *This may be confusing at times...*

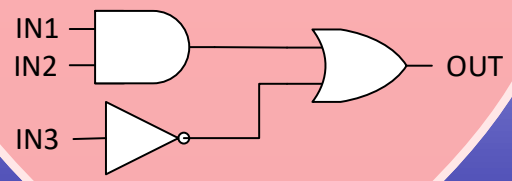
HDL

REPETITION

Code for generating and parsing simulation data (Test benches)

code for generating multiple instances or variants of entities

Synthesizable code



VHDL structure

- Design entities
- Architecture styles
- Ports and signals
- Vectors
- Assignment
- Libraries
- STD_LOGIC data type
- Operators

Design entities in VHDL

- 5 types of design entities
  - Entity
  - Architecture *body*
  - Package
  - Package body
  - Configuration
- Each entity can have its own file...

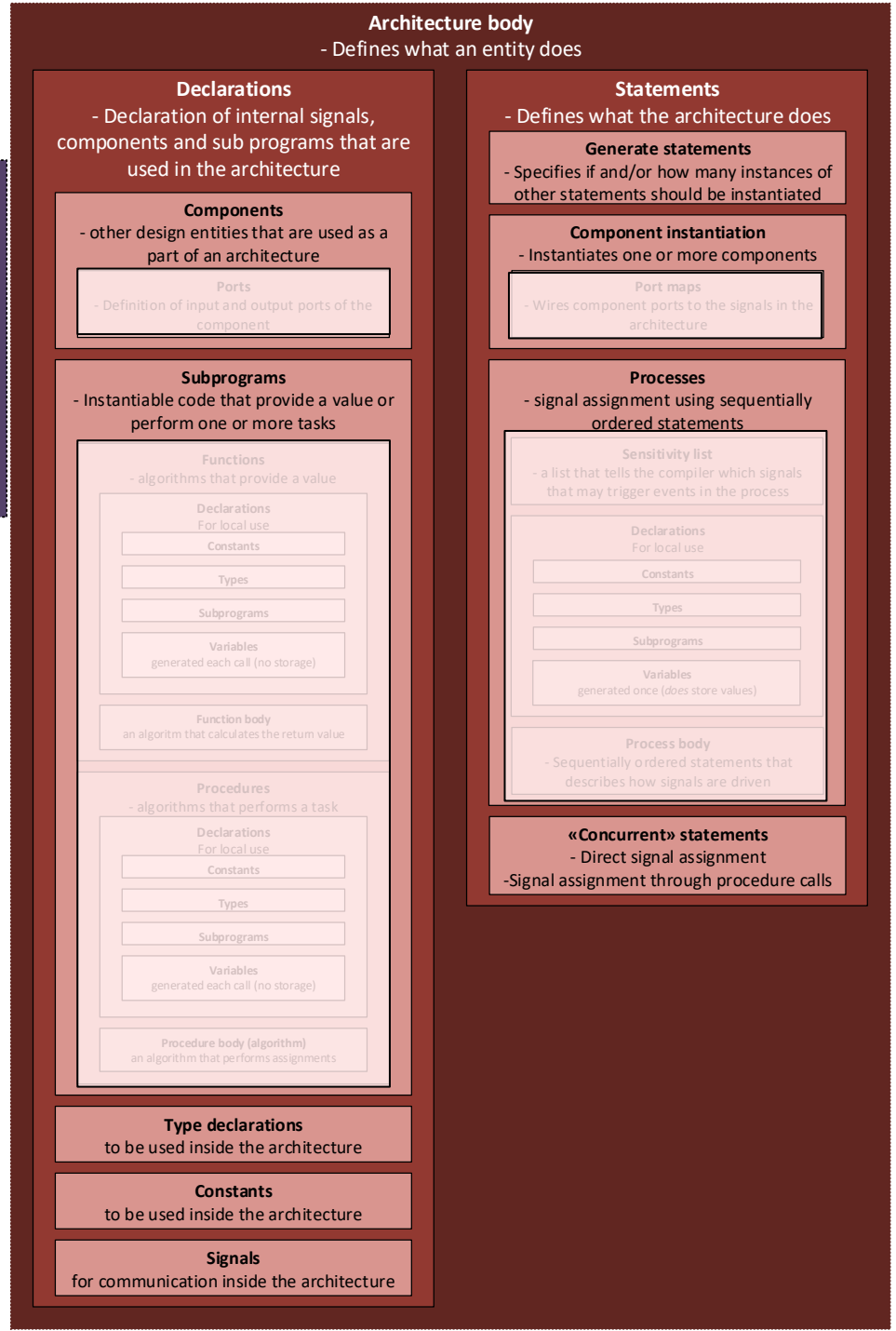
Design-entities: Entity and architecture

Entity
- Defines the interface to the surronding environment

Generics
-Values that can be used inside the entity as constants

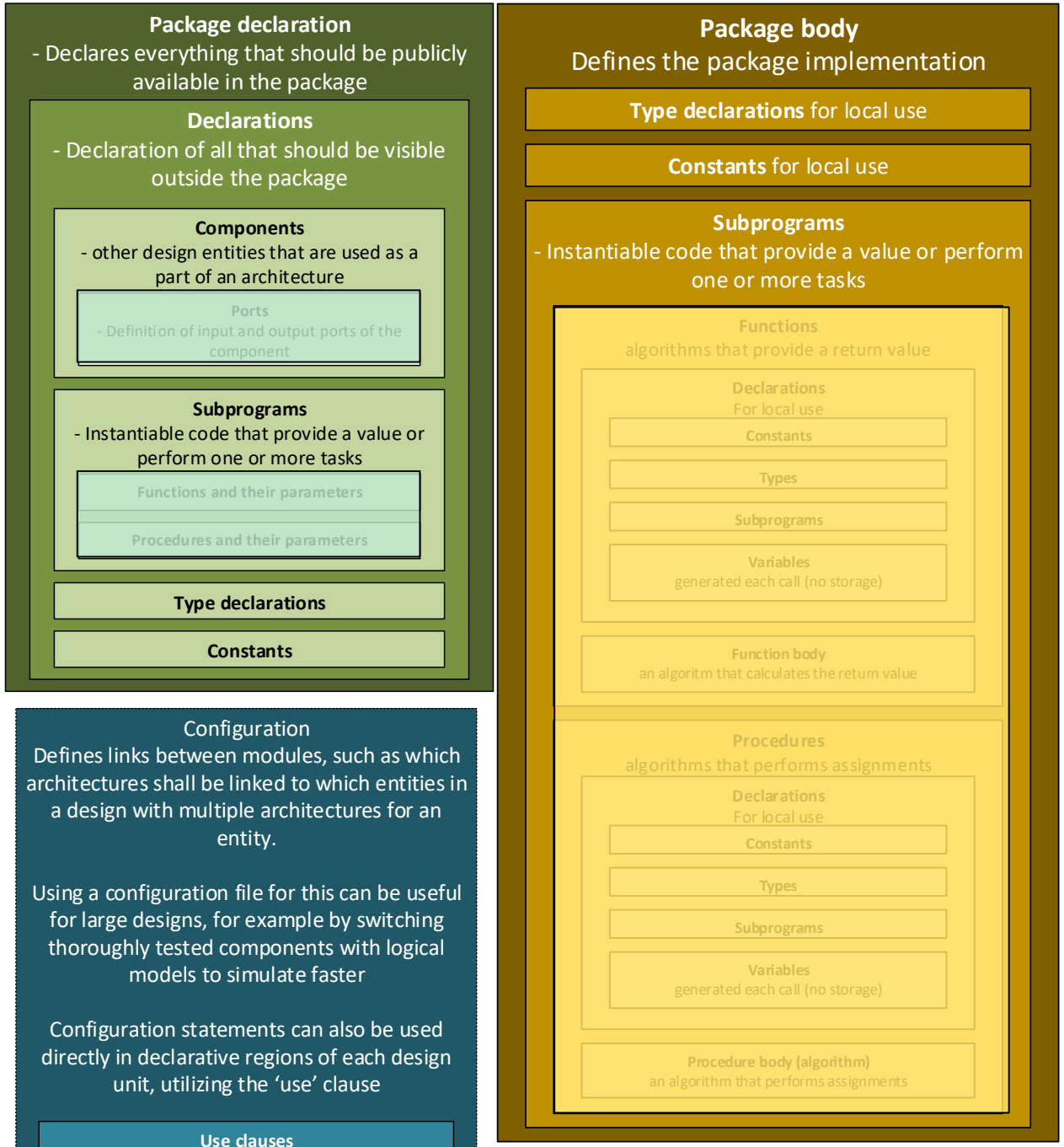
IO ports
- Defines the entity interface

- E. and A. is often put into the same file
- In larger designs these may be separated, several architectures can be used for one entity.
  - Simulation vs modules for synthesis
- *Details will be revealed later..*

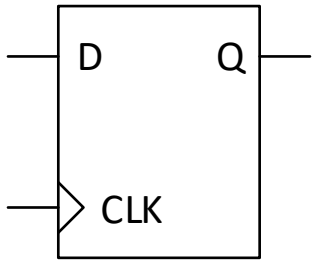


Design entities:
Package, P.body, Configuration

- VHDL uses and can be used to create packages
- We will almost always use packages in precompiled libraries.
- *Configuration* files can be used to specify which components or architectures that shall be used in (large) designs
  - (Not a primary concern for in3160)



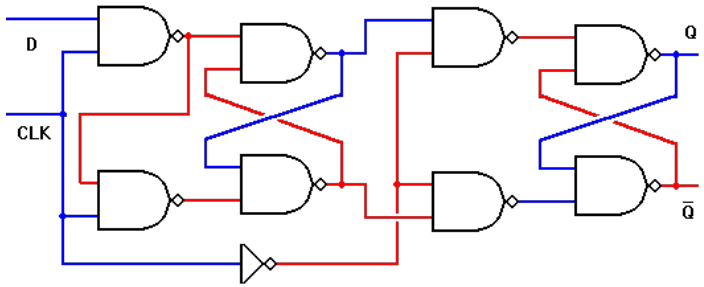
VHDL Entity and Architecture

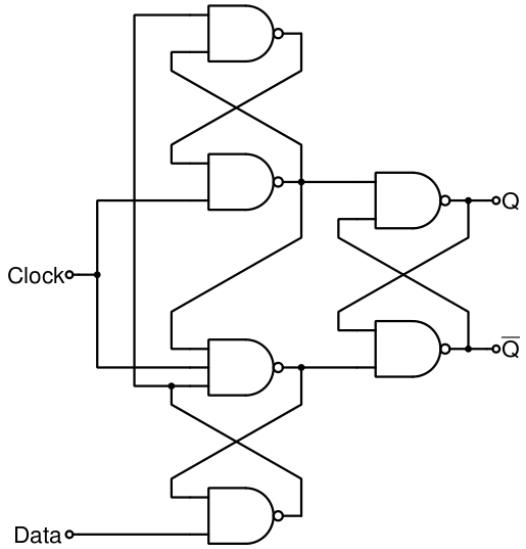


- Entity defines Input and Output ports in the design
  - There is only one entity in a vhdl file..
- Architecture defines what the design does.
  - There can be several architectures for an entity
  - Architectures, may be defined using different styles (next slide)
    - "RTL" and "Dataflow" are names providing information;
      - changing these names would not change function.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity D_flipflop is
  port(
    clk: in std_logic;
    D  : in std_logic;
    Q  : out std_logic
  );
end entity D_FLIPFLOP;
```

```
architecture RTL of D_flipflop is
begin
  process (clk) is
  begin
    if rising_edge(clk) then
      Q <= 'D';
    end if;
  end process;
end architecture RTL;
```






```
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;
```

4 main abstraction levels

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity D_flipflop is
  port(
    clk: in std_logic;
    D: in std_logic;
    Q: out std_logic
  );
end entity D_flipflop;
```

```vhdl
architecture RTL of D_flipflop is
begin
  process (clk) is
  begin
    if rising_edge(clk) then
      Q <= 'D';
    end if;
  end process;
end architecture RTL;
```

RTL, register transfer level

- high level
- easy to read
- describes registers and what happens between them
- «default» for sequential logic

Data Flow

- typically used for optimization
- will easily become unreadable if used extensively.
- *Use higher level code unless absolutely necessary*

```vhdl
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;
```

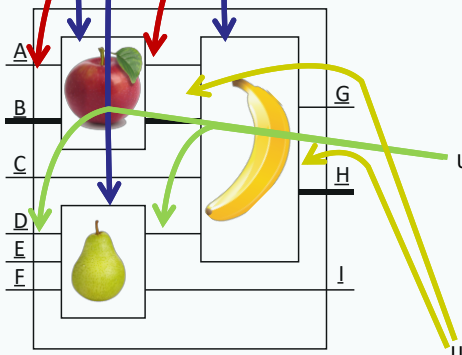
Behavioral

- *Simulation models only = Software*
- Not for synthesis or implementation

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity my_thing is
  port (
    A: in std_logic;
    B: in std_logic_vector(5 downto 0);
    C, D, E, F: in std_logic;
    G: out std_logic;
    H: out std_logic_vector(64 downto 0);
    I: out std_logic
  );
end entity my_thing;
```

Structural

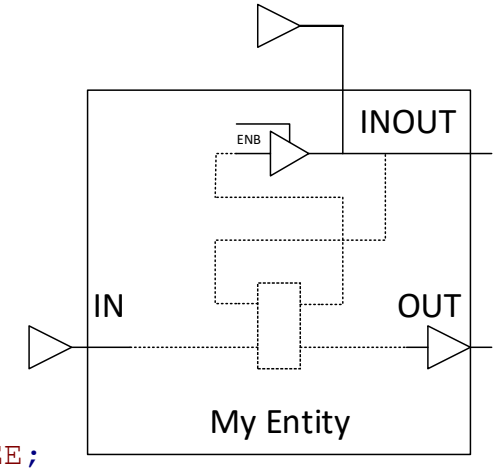
- Ties components together
- Typically used in test benches, and when using predefined components




```vhdl
architecture structural of my_thing is
  signal js: std_logic;
  signal ks: std_logic_vector(64 downto 0);
  signal ls: std_logic;

  component apple is
    port (
      A: in std_logic;
      B: in std_logic_vector(5 downto 0);
      C: out std_logic;
      D: out std_logic_vector(64 downto 0)
    );
  end component;

  component pear is
    port (
      A, B, C: in std_logic;
      D, E: out std_logic
    );
  end component;

  component banana is
    port (
      smurf: in std_logic_vector(64 downto 0);
      cat, dog, donkey: in std_logic;
      horse: out std_logic;
      monkey: out std_logic_vector(64 downto 0)
    );
  end component;

begin -- port map (component => My_thing)
  U1: apple port map(
    A => A,
    B => B,
    C => js,
    D => ks
  );

  U2: pear port map(
    A => D,
    B => E,
    C => F,
    D => ls,
    E => I
  );

  U3: banana port map(
    smurf  => ks,
    cat    => js,
    dog    => C,
    donkey => ls,
    horse  => G,
    monkey => H
  );

end architecture structural;
```

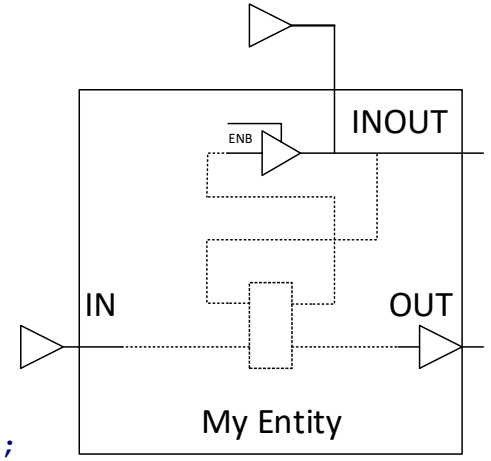
Ports and signals

- Ports define the entity interface
  - IN:
    - *can only be read*
    - cannot be driven or assigned internally
  - OUT:
    - *should be driven from the architecture*
      - *It is bad practice not to do so.*
      - *prior to VHDL2008 output ports could not be read*
  - *INOUT*
    - *Can be both driven and read (typical use is for buses)*
- Signals are internal
  - For connecting internal modules, subprograms and processes.




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

entity my_entity is
  port(
    Ain    : in    std_logic;
    Binout : inout std_logic;
    Cout   : out   std_logic
  );
end entity my_entity;

architecture dataflow of my_entity is
  signal enb, s1, s2, s3, s4 : std_logic;

begin
  -- ...

  s1 <= Ain;
  Cout <= s2;
  -- reading INOUT is OK
  s3 <= Binout;
  -- setting INOUT should implement tri-state
  Binout <= s4 when enb else 'Z';

end architecture dataflow;
```

Ports continued



INOUT is for tying input and output to the same pin

- ***should* implement tristate functionality**.
- 'Z' means it is not driven (tristate)
- Typically to be used when connecting a bus with multiple drivers.

DO NOT use INOUT for convenience!

- The compiler will not alert you if you are driving from two sources simultaneously.
  - *May cause undetected electrical faults*
- INOUT may infer inferior structures (long delays)

Even if it is used safely, it will require special resources, usually only found near the package boundaries.

(Can create timing or availability issues, and other..)

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

entity my_entity is
  port(
    Ain    : in    std_logic;
    Binout : inout std_logic;
    Cout   : out   std_logic
  );
end entity my_entity;

architecture dataflow of my_entity is
  signal enb, s1, s2, s3, s4 : std_logic;

begin
  -- ...

  s1 <= Ain;
  Cout <= s2;
  -- reading INOUT is OK
  s3 <= Binout;
  -- setting INOUT should implement tri-state
  Binout <= s4 when enb else 'Z';

end architecture dataflow;
```

Statements and assignments

- Statements and processes
  - defines how a digital circuit works
  - assign drivers to signals
  - All statements must be valid at all times
    - A circuit may have sequential behavior...
      - But all logic is present and functional at all times
  - Processes are complex statements
    - Assigns values to one or more signals
    - Can have local variables.. (more on those later)
  - A signal can only be assigned once in an architecture
    - We do not want two circuits to apply voltage on the same wire...

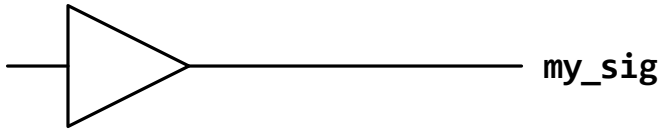
Examples:

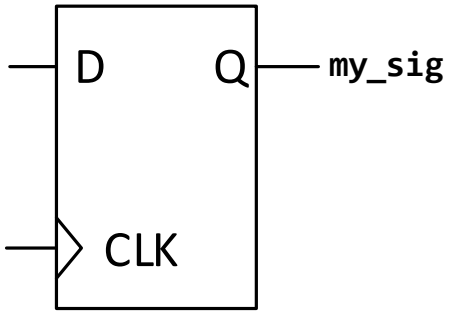
```
with input select isprime <=
    '1' when x"1" | x"2" | x"3" | x"5" | x"7" | d"11" | x"d",
    '0' when others;
```

```
process(Reset, Clock) is
    variable Q : Unsigned(7 downto 0);
  begin
    if Reset = '0' then
      Count <= zero_byte;
    elsif rising_edge(Clock) then
      Q :=
        unsigned(Count)     when  Enable  else
        unsigned(Data)      when not Load else
        unsigned(Count) + 1 when not Mode else
        dec_count(unsigned(Count));
      Count <= std_logic_vector(Q);
    end if;
  end process;
```

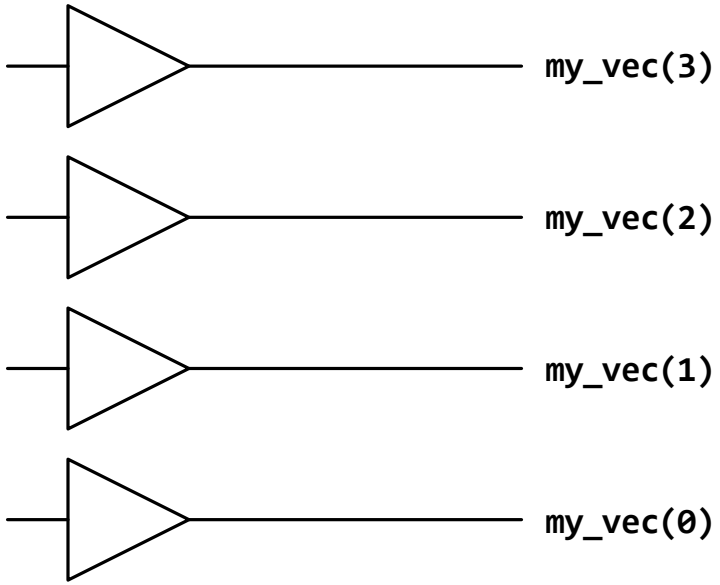
«Vectors»

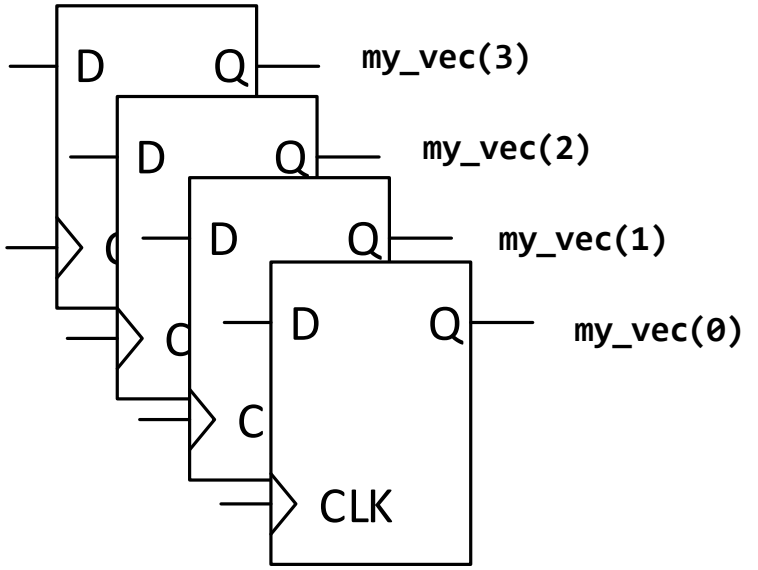
- `signal my_sig std_logic;`





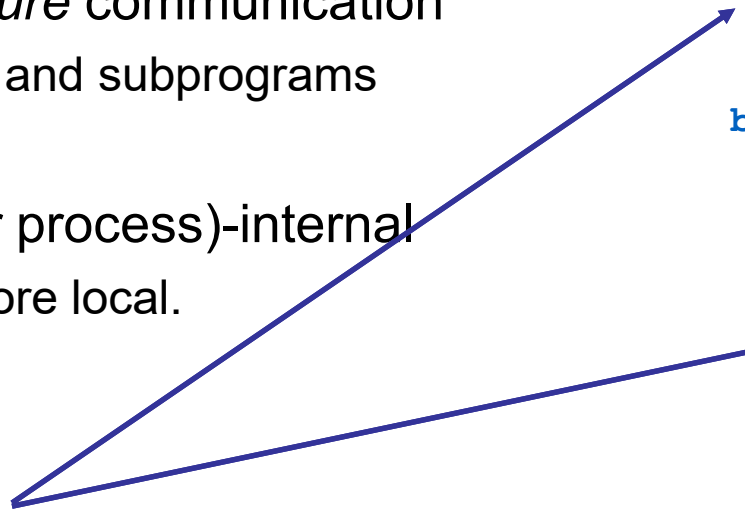
- `signal my_vec std_logic_vector(3 downto 0);`





Signals and variables

- **signal**s are for *inter-architecture* communication
  - Between processes, modules and subprograms
- **variable**s are subprogram(or process)-internal
  - To make code clearer, and more local.
- Example note:
  - placement of s&v declarations
  - Signal assignment order is irrelevant outside processes



```
architecture example of sigvar is
  -- (signal) declarations
  signal S, T : std_logic;

begin
  -- statements
  S <= A and B;

  process (A,B) is
    -- (variable) declarations
    variable V : std_logic;
  begin
    -- process body
    V := '0';
    -- …

  end process;

  X <= S XOR T;
end architecture;
```

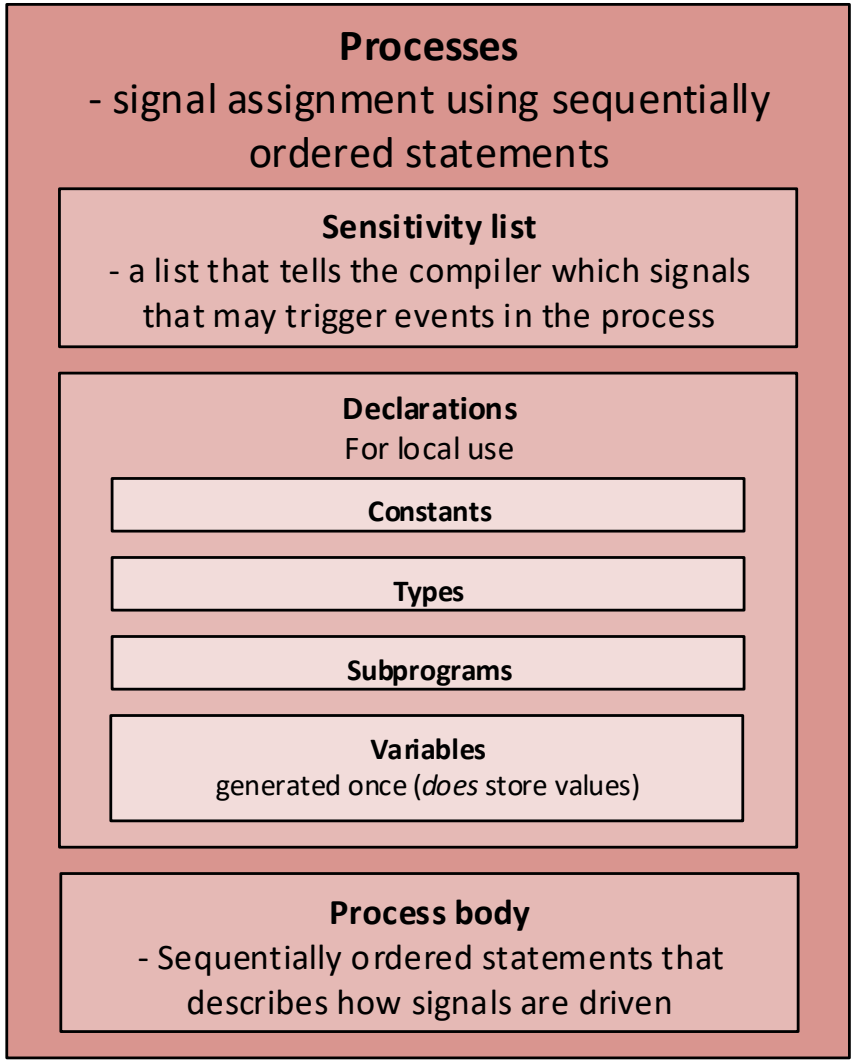
Signal and variable assignment

- Signals are assigned concurrently in statements
  - both in and outside processes
  - Signals are assigned using <=
  - Signals uses event based updates
    - ie after a process is complete.
- Variables can only be used inside processes and subprograms
  - Variables are assigned using :=
  - Variables are updated immediately in simulation
  - *Processes can have variables store values*
    - *Initialized at the beginning of simulation*
  - *Subprograms (procedure, function) can not have variables store values*
    - *initialized on every call*

```
A <= B; -- A reads B, or A is assigned to B's ouput,
        -- (A is a signal)

D(6 downto 0) <= E(3 downto 1) & (others => '0');
              -- D is a vector having 7 input signals
              -- D(6) <= E(3)
              -- D(5) <= E(2)
              -- D(4) <= E(1)
              -- D(3 downto 0) <= "0000"

C := B; -- C is given B's value, C is a variable
        -- variables are used internally in processes.
```

Processes

- A process is one (concurrent) statement
  - Ensures one driver for each signal by using priority.
    - "Signals are only updated only once"..
  - The process body has sequential *priority*
    - Last assignment takes precedence over previous.
    - Variables *can* be assigned multiple times within a process body(!)
      - They *can* act as several signals
        - » Generally this should be avoided
  - sensitivity list
    - determines when the process body is invoked *during simulation*
    - *Event triggered*
  - *Can* be used to make sequential logic
    - Clocked events infers flipflops (or latches)



Process example
(avoid this style)

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

entity sigvar is
  port(
    A, B : in  std_logic;
    X    : out std_logic
  );
end entity sigvar;
```

- The use of V would not be allowed a signal outside a process.
- A variable can be used in place of multiple physical signals within a process. (= bad practice...)
- Signals will be assigned one driver in a process.

```
architecture bad of sigvar is
  -- declarations
  signal S, T : std_logic;

begin
  -- statements
  S <= A and B;

  process (A,B) is
    -- decalarations
     variable V : std_logic;

  begin
    -- process body
    V := '0';
    if (A = '1') then
      V := '1';
    end if;
    if (B = '1') then
      V:= '1';
    end if;
    T <= V;
  end process;

  X <= S XOR T;
end architecture;
```

Process example (template code = can be used)

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

entity adder is
  port(
    clk, reset : in std_logic_vector;
    a, b : in  std_logic_vector(7 downto 0);
    sum  : out std_logic_vector(8 downto 0)
  );
end entity adder;
```

- Process for
  - Registry update
  - Complex combinational assignments
- Variables
  - Only used for clarifying

```
architecture RTL of adder is
  signal r_sum, next_sum : unsigned(sum'range);
begin
  -- concurrent statements
  SUM <= std_logic_vector(r_sum);

  REG_UPDATE: process(clk)
  begin
    if rising_edge(clk) then
      if reset then
        r_sum <= (others => '0');
      else
        r_sum <= next_sum;
      end if;
    end if;
  end process;

  COMB_PROCESS: process(all)
    variable va, vb : unsigned(sum'range);
  begin
    -- variables used for clarification
    va <= unsigned("0" & a);
    vb <= unsigned("0" & b);
    next_sum <= va + vb;
  end process;
end architecture;
```

Libraries and Data types

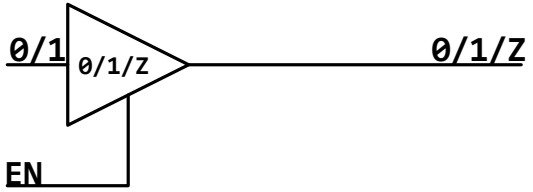
- VHDL is built upon use of libraries and packages.
- You can both use existing ones, and create your own.
- The `library IEEE`, contains the most used data types and functions
  - *The built-in standard (std) package*, containing:
    - `bit`, `integer`, `natural`, `positive`, `boolean`, `string`, `character`, `real`, `time`, `delay_length`
  - std_logic_1164
    - defines the `std_logic` type
  - numeric_std
    - numeric operations for *«std_logic_vectors»*: `unsigned`, `signed`
  - std_logic_textio
    - to provide IO during simulation (ie VHDL testbenches)
    - *Can* be used with *synthesizable VHDL (=unlikely)*
  - numeric_bit
    - numeric operations for bit vectors
  - etc.

STD_LOGIC *TYPE* (requires std_logic_1164 package from IEEE library)

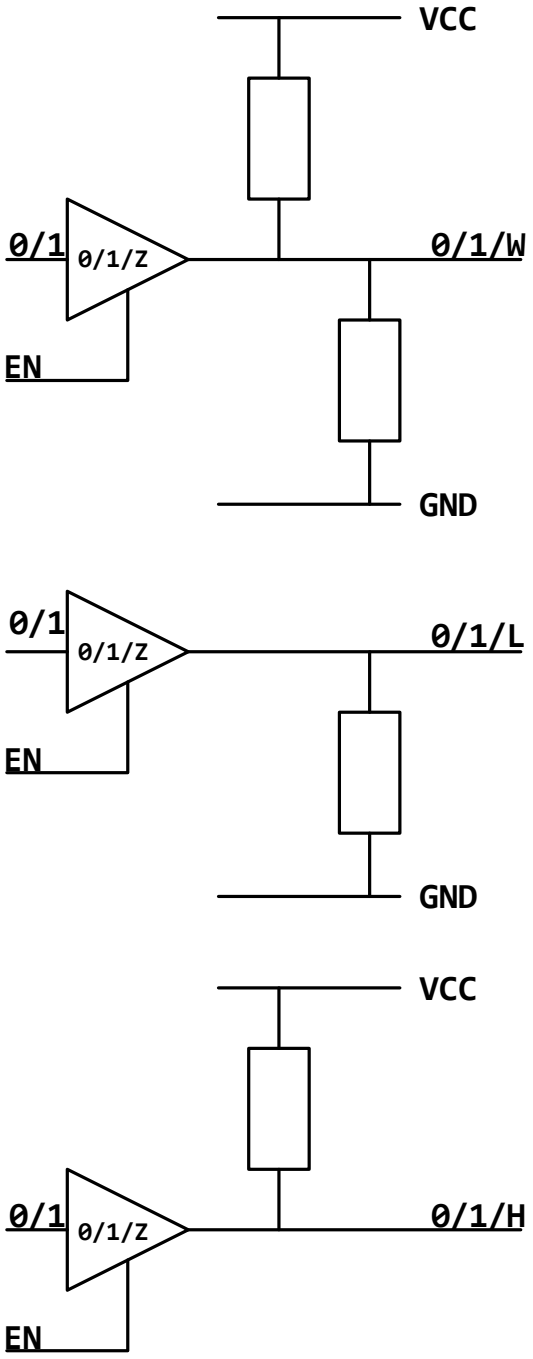
- STD_LOGIC is a **type** that has the following possible values
  - `'U'` Uninitialized (Typically seen in simulation before initializing values)
  - `'X'` Unknown (typically when a signal is driven to both 0 and 1 simultaneously)
  - **`'0'` Driven low**
  - **`'1'` Driven High**
  - **`'Z'` Tristate**
  - `'W'` Weak unknown (when driven by two different weak drivers)
  - `'L'` Weak '0' (Typically for simulating a pulldown resistor)
  - `'H'` Weak '1' (Typically for simulating a pullup resistor)
  - `'-'` Don't care (Typically for assessing results in simulator).
- **You will only *assign*** synthesizable **signals to '0', 1' and 'Z'**

- Type STD_LOGIC_VECTOR is array (NATURAL range <>) og STD_LOGIC
  - STD logic vector is used for hardware. For simulation, other types (such as integer) may be faster. Thus we use STD_LOGIC for hardware interactions, and other types when possible for test bench code.

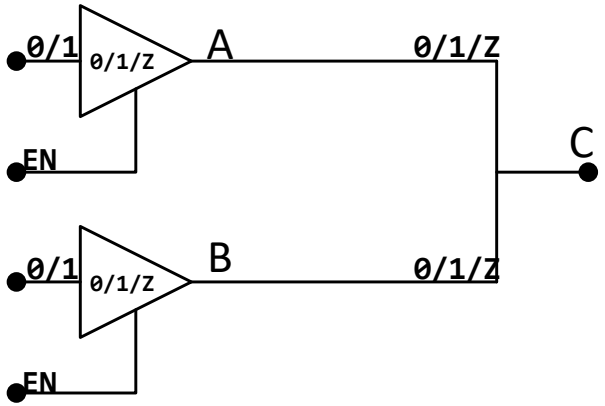
std_logic -- values



Value	Name	Usage
‘U’	Uninitialized state	Used as a default value
'X'	Forcing unknown	Bus contentions, error conditions, etc.
‘0’	Forcing zero	Transistor driven to GND
‘1’	Forcing one	Transistor driven to VCC
‘Z’	High impedance	3-state buffer outputs
‘W’	Weak unknown	Bus terminators
‘L’	Weak zero	Pull down resistors
‘H’	Weak one	Pull up resistors
‘-’	Don’t care	Used for synthesis and advanced modeling

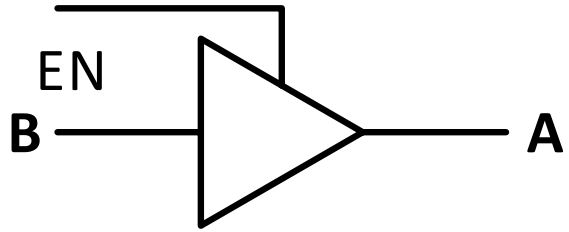


```
-- multiple drivers
signal c : std_logic;
```



Signal A/B	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'–'
'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
'X'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'0'	'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'
'1'	'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'
'Z'	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'
'W'	'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'
'L'	'U'	'X'	'0'	'1'	'L'	'W'	'L'	'W'	'X'
'H'	'U'	'X'	'0'	'1'	'H'	'W'	'W'	'H'	'X'
'–'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'

Tri-state buffer



- Digital logic *reads only either '0' or '1'*
- We can set tristate (high impedance) to allow other sources to drive a bus.
- Simulation tools can use all possible `STD_LOGIC` values.

```
A <= B when EN = '1' else 'Z';

-- ekvivalent med
TRISTATE:
process (B,EN)
begin
  if EN = '1' then
    A <= B;
  else
    A <= 'Z';
  end if;
end process;
```

VHDL operator priority

- Functions are interpreted from left to right (in reading order).
- **Use paranthesis to govern priority!**

Prioritet	Operator klasse	Operatorer
1 (first)	miscellaneous	`**, abs, not`
2	multiplying	`*, /, mod, rem`
3	sign	`+, -`
4	adding	`+, -, &`
5	Shift	`sll, srl, sla, sra, rol, ror`
6	relational	`=, /=, <, <=, >, >=, ?=, ?/=, ?<, ?<=, ?>, ?>=`
7	logical	`And, or, nand, nor, xor, xnor`
8 (last)	condition	`??`

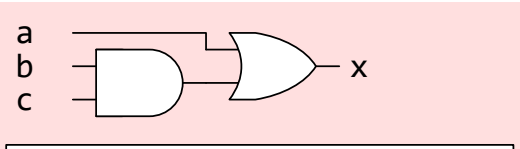
Examples (elaboration on next page):

```
a <= a or b and c;   ==   a <= (a or b) and c;

z <= a and not b and c; == z <= a and (not b) and c; == z <= c and (a and (not b));

y <= a and not (b and c); -- z=1 kun for a=1, b=0, c=1.   y=1 for a=1 og (b eller c)=0.
```

VHDL operator priority

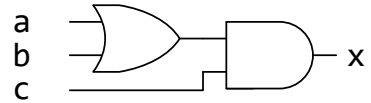
Examples:

```
x <= a or b and c;   ==   x <= (a or b) and c;
```




```
x <= a or (b and c);
```

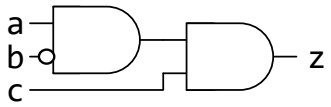
(What you might want)



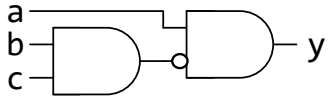

```
x <= a or b and c;
```

(what you actually will get)

Try
a := '1'
b := '1'
c := '0'

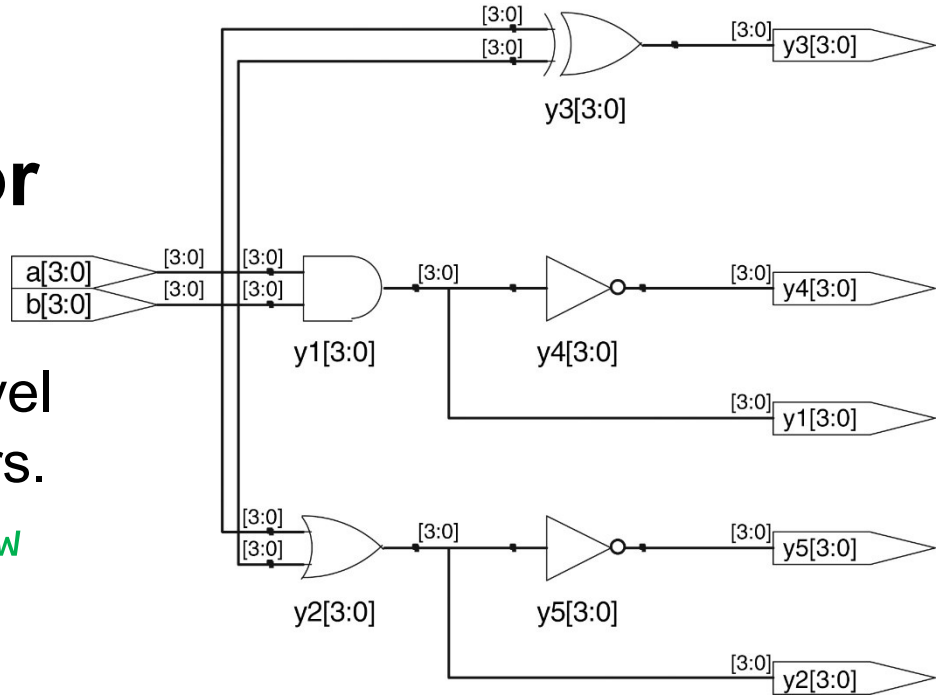



```
-- z=1 only when a=1, b=0, c=1.
z <= a and not b and c; == z <= a and (not b) and c; == z <= c and (a and (not b));
y <= a and not (b and c);
```




```
--  y=1 when a=1, b=0 or when a=1, c=0.
```

Bit operators and reduction operator

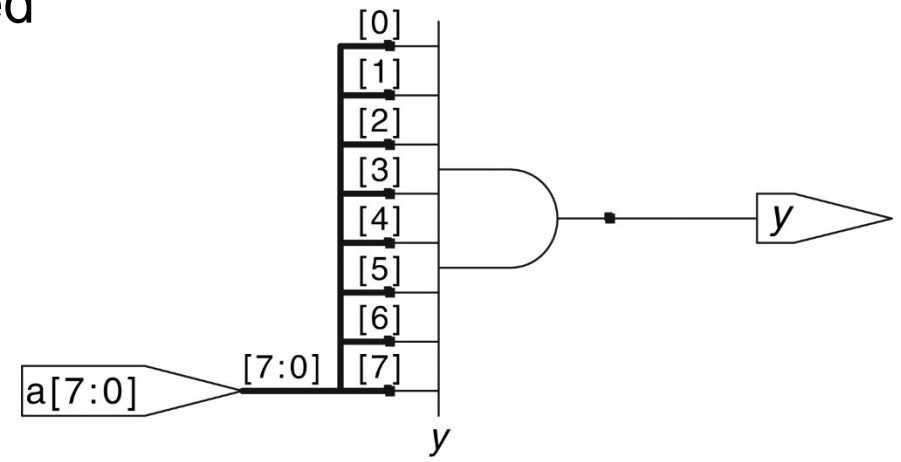


- `and`, `or`, `not`, `xor`, `xnor` operators will work at bit level when they are placed between two signals or vectors.
  - `y1 <= a and b; -- is equal to the lines below`

```
y1(3) <= a(3) and b(3);
y1(2) <= a(2) and b(2);
y1(1) <= a(1) and b(1);
y1(0) <= a(0) and b(0);
```

- I VHDL2008 (not earlier) these operators can be used for reduction

```
y <= and a ; -- is equal to the figure ->
```



- `xor` can be used this way to generate (even) parity for a signal.

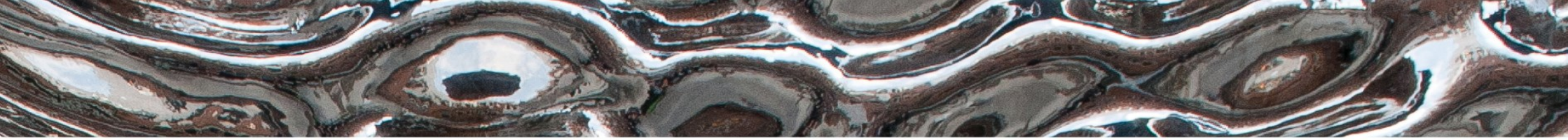
VHDL Bit String Literals

Binary, **D**ecimal, he**X**adecimal, **O**ctal, **U**nsigned, **S**igned

<ant bit><U/S><B/D/O/X><numbers of type B/D/O/X >

```
B"1111_1111_1111" -- Equivalent to the string literal
"111111111111".
X"FFF" -- Equivalent to B"1111_1111_1111".
O"777" -- Equivalent to B"111_111_111".
X"777" -- Equivalent to B"0111_0111_0111".
B"XXXX_01LH" -- Equivalent to the string literal
"XXXX01LH"
UO"27" -- Equivalent to B"010_111"
UO"2X" -- Equivalent to B"011_XXX"
SX"3W" -- Equivalent to B"0011_WWWW"
D"35" -- Equivalent to B"100011"
```

```
12UB"X1" -- Equivalent to B"0000_0000_00X1"
12SB"X1" -- Equivalent to B"XXXX_XXXX_XXX1"
12UX"F-" -- Equivalent to B"0000_1111_----"
12SX"F-" -- Equivalent to B"1111_1111_----"
12D"13" -- Equivalent to B"0000_0000_1101"
12UX"000WWW" -- Equivalent to B"WWWW_WWWW_WWWW"
12SX"FFFC00" -- Equivalent to B"1100_0000_0000"
12SX"XXXX00" -- Equivalent to B"XXXX_0000_0000"
8D"511" – Error (> 2^8)
8UO"477" – Error (>2^8)
8SX"0FF" – Error (cannot have 255 using 8 bit signed)
8SX"FXX" – Error (cannot extend beyond 8 bit)
```



UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

IN3160

Code Layout (15 min)

Kilde: Ricardo Jasinski: Effective Coding with VHDL, Chapter 18

Overview

- Why bother thinking about layout?
- What constitutes a good layout scheme?
- Basic layout types
- Indentation
- Paragraphs and spaces

Why bother thinking of layout?

```
pRoCeSS(clock,reset) bEGIn iF resET then oUTpuT <="0000"; elSE IF RISING_edge
(ClOck) tHEN cASE s Is When 1=>outPUT<= "0001"; wHEn 2046=> oUTpuT <="0010";WheN
31=>OutPut<="0100";when OTHERs=>OUTput <= «1111"; end CASe; END if;END proCESS; --
Q.E.D.
```

A good layout scheme…

1. …accurately matches the structure of the code
2. …improves readability
3. …affords changes
4. …is consistent (few exceptions)
5. …is simple (few rules)
6. …is easy to use
7. …is economic

Basic layout types

Block layout

Endline layout

Column layout

Block layout (What you should use *most of the time*)

- Accurately matches structure
- relatively tidy
- readable,
- easy to maintain, etc.

```
process (clock, reset)
begin
  if reset then
    output <= "0000";
  else if rising_edge(clock) then
    case s is
      when 1 => output <= "0001";
      when 2046 => output <= "0010";
      when 31 => output <= "0100;
      when others => output <= "1111";
    end case;
  end if;
end process;
```

Endline layout (*avoid this*)

- Harder to maintain when code changes.
- Looks tidier, but isn't faster than pure block
- Will get messy- poor match of code hierarchy
- Long lines..!

```
process (clock, reset)
begin
  if reset then output <= "0000";
  else if rising_edge(clock) then case s is
                                  when 1 => output <= "0001";
                                  when 2046 => output <= "0010";
                                  when 31 => output <= "0100";
                                  when others => output <= "1111";
                                  end case;
  end if;
end process;
```

Column layout (use sparingly)

- Can be easier to read than pure block layout (scanning vertically)
- Harder to maintain.
- Best to use when columns are unlikely to change, and statements are related.
  - Typically used for 2D arrays.

```
process (clock, reset)
begin
  if reset then
    output <= "0000";
  else if rising_edge(clock) then
    case s is
      when      1 => output <= "0001";
      when      2 => output <= "0010";
      when    333 => output <= "0100";
      when others => output <= "1111";
    end case;
  end if;
end process;
```

Indentation

- *Use indentation to match code hierarchy*
- 2-4 spaces has been proven most efficient
  - *Along with a monospace font, such as* `courier`, `consolas`..
- Use space rather than tabulator sign.
  - Tabulator spaces may be interpreted differently in different editors.
  - Most editors can be set up for this.
- Example:

```
entity ent_name is
  generic (
    generic_declaration_1;
    generic_declaration_2;
  );
  port(
    port_declaration_1;
    port_declaration_2;
  );
end entity ent_name;
.
.
.
process (sensitivity_list)
  declaration_1;
  declaration_2;
begin
  statement_1;
  statement_2;
end process;
```

Paragraphs and comments

- Paragraphs should be used to separate chunks that does not need to be read all at once.
- Paired with comments that this make for good readability
- Comments should be indented as according to the code it is referring to.

```
-- Find character in text RAM corresponding to x, y
text_ram_x := pixel_x / FONT_WIDTH;
text_ram_y := pixel_y / FONT_HEIGHT;
display_char := text_ram(text_ram_x, text_ram_y);

-- Get character bitmap from ROM
ascii_code := character'pos(display_char);
char_bitmap := FONT_ROM(ascii_code);

-- Get pixel value from character bitmap
x_offset := pixel_x mod FONT_WIDTH;
y_offset := pixel_y mod FONT_HEIGHT;
pixel := char_bitmap(x_offset)(y_offset);
```

Line length and wrapping

- Try to keep line length within reasonable limits
  - 80, 100 and 120 characters is widely used.
- When wrapping lines:
  - break at a point that clearly shows it is incomplete, such as
    - after opening paranthesis
    - after operators or commas (`&, +, -, *, /`)
    - after keywords such as «`and`» or «`or`»
  - consider one item per line…

```
-- one item/line + named association
Paddle <= update_sprite(
  sprite => paddle,
  sprite_x => paddle_position.x + paddle_increment.x,
  sprite_y => paddle_position.x + paddle_increment.y,
  raster_x => vga_raster_x,
  raster_y => vga_raster_y,
  sprite_enabled => true
);
```

```
-- several items/line
paddle <= update_sprite(paddle, paddle_position.x + paddle_increment.y,
  paddle_position.y + paddle_increment.y, vga_raster_x, vga_raster_y, true);
```

Spaces

- Punctuation symbols (comma, colon, semicolon)
  - use spaces as you would in regular prose:
    - Never add space before punctuation symbols
    - Always add space after punctuation symbols
  - *no exceptions*

```
-- too much
function add ( addend : signed ; augend : signed ) return signed ;

-- better
function add(addend: signed; augend: signed) return signed;
```

- Parantheses
  - Add a space before opening paranthesis.
  - Add a space or punctuation symbol after closing paranthesis
  - Except:
    - array indices and routine parameters;
    - *expressions*.

```
-- consider this expression:
a + b mod c sll d;

-- better
(a + (b mod c)) sll d;
```

```
-- consider
(-b+sqrt(b**2-4*a*c))/2*a;

-- better
(-b + sqrt(b**2 - 4*a*c)) / 2*a;

-- too much
( - b + sqrt( b ** 2 - 4 * a * c ) ) / 2 * a;
```

Naming conventions - Letter case and underscores

```
noconventionnaming -- don't go there
UpperCamelCase -- OK used consequently
lowerCamelCase -- OK used consequently
snake_case or underscore_case
SCREAMING_SNAKE_CASE or ALL_CAPS
```

- Do not use ALL_CAPS too frequently.
- Use editor colors/ **bold** for higlighting **keywords**

- Try to avoid mixing snakes_andCamels.
- Treat acronyms/ abbreviations as words
  - `"UDPHDRFromIPPacket"` ***vs***
    `"UdpHdrFromIpPacket"` ***vs***
    `"udp_hdr_from_ip_packet"`

- VHDL packages tend to favour snake_case and ALL_CAPS

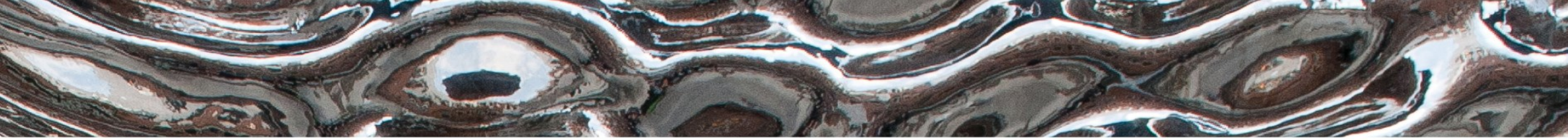
- *Suggestion:*
  - Use snake_case for all **names** except **constants** and **generics** that use ALL_CAPS

Suggested reading, Mandatory assignments

- D&H:
  - 1.5 p 13-16
  - 3.6 p 51-54
  - 6.1 p 105-106

- Oblig 1: «Design Flow»
- Oblig 2: «VHDL»
  - See canvas for further instruction.

Layout is slides only



UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

IN3160

Combinational logic design

Verification 1:

Compilation and Simulation

Basic Test benches.

Messages:

- https://github.uio.no/in3160
  - Intend to bring more content there
- 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

Combinat**ional vs** Combinat**orial**

Combinational

↓

combinational logic circuit
combines inputs to generate
an output

Combinatorial

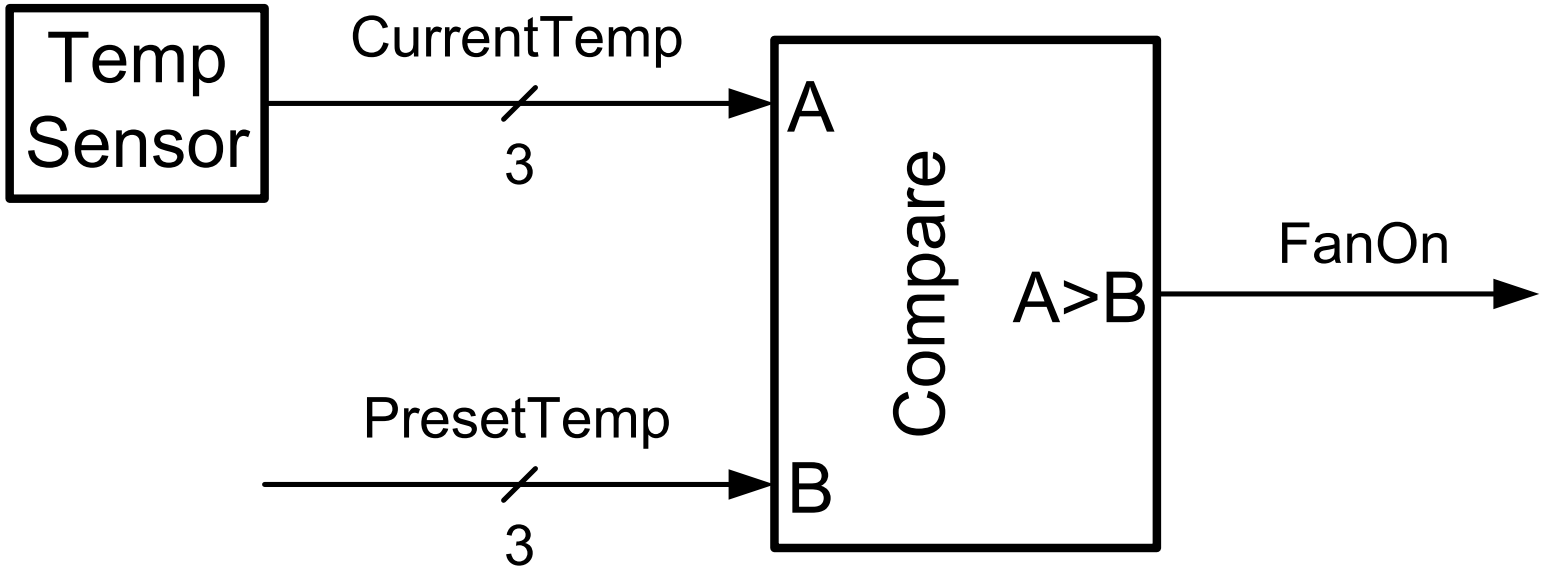
↓

mathematics of counting

Norsk: Kombinasjonslogikk / kombinatorisk logikk)
Varierende bruk forekommer…
I all hovedsak brukes “kombinatorisk” på norsk..

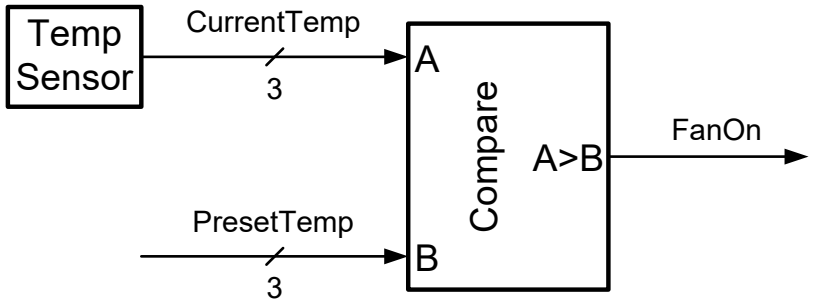
Combinational Logic Circuit

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



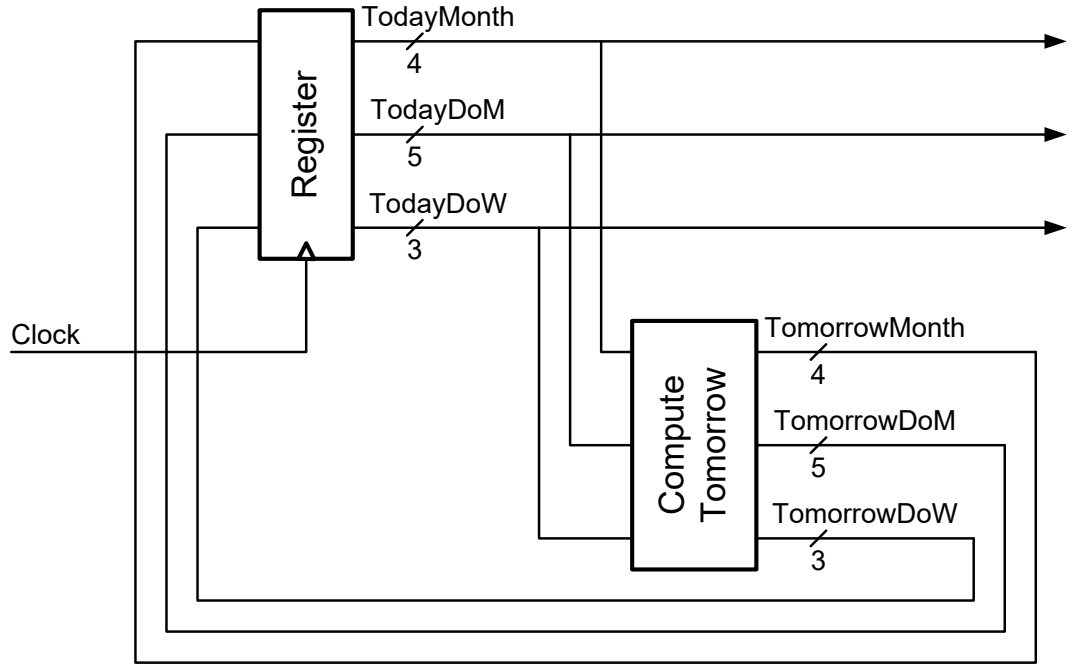
VHDL code

```
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;
```



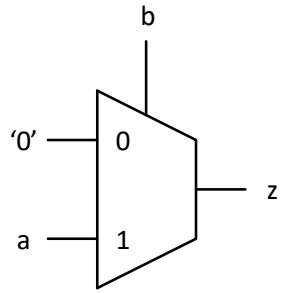
Sequential logic circuit

- 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;
```

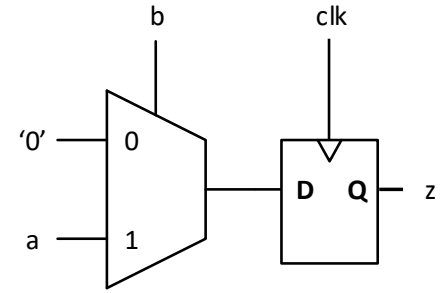



```
-- «all» can be replaced by b here
```

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

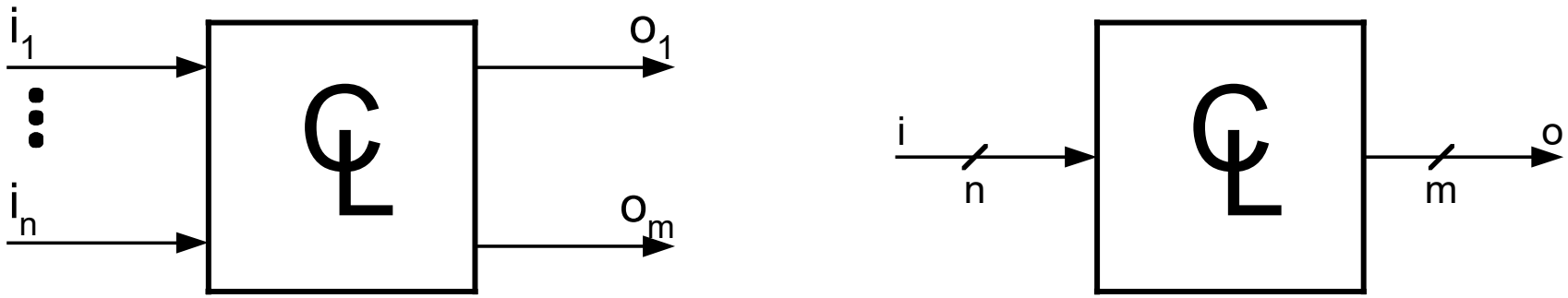
```
-- concurrent statement is more compact...
z<= a when b else '0';
```

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




```
-- quite often we have both reset and clk
```

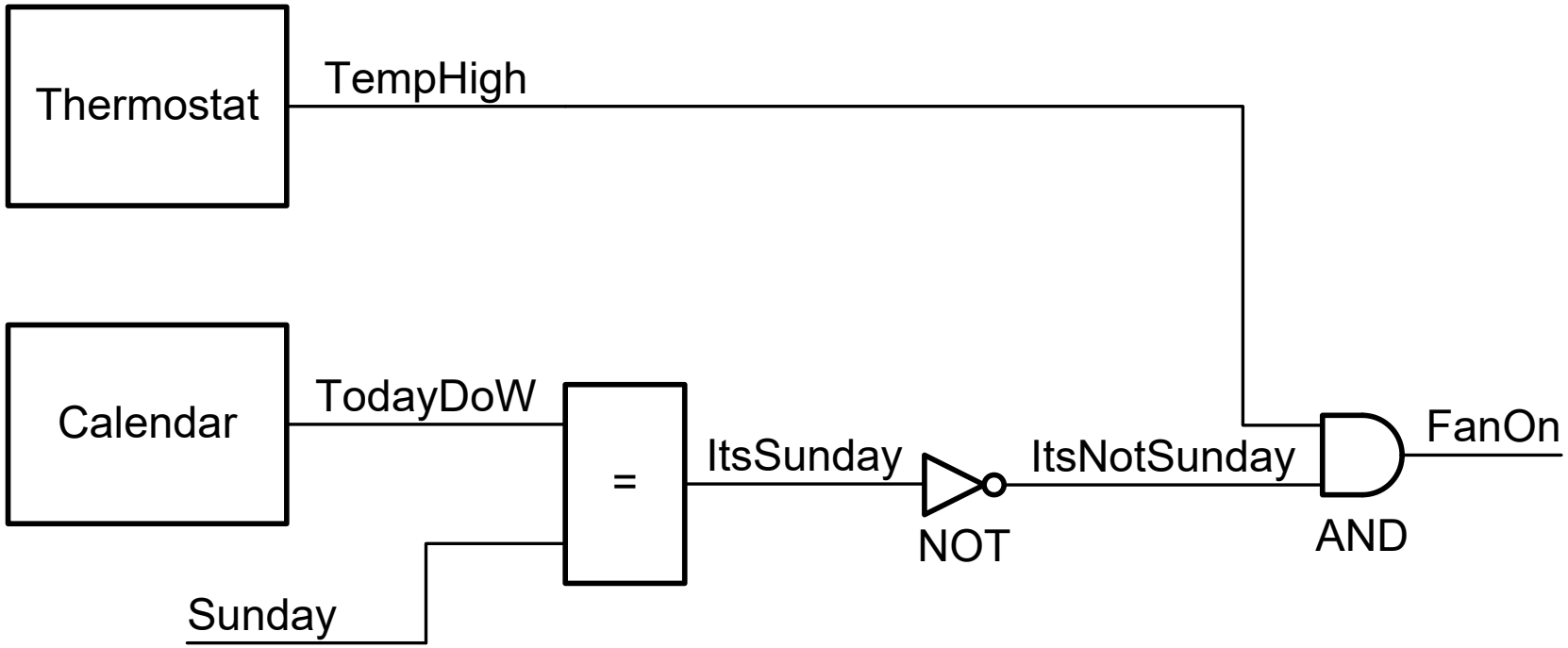
Combinational logic is memoryless



o = f(i)

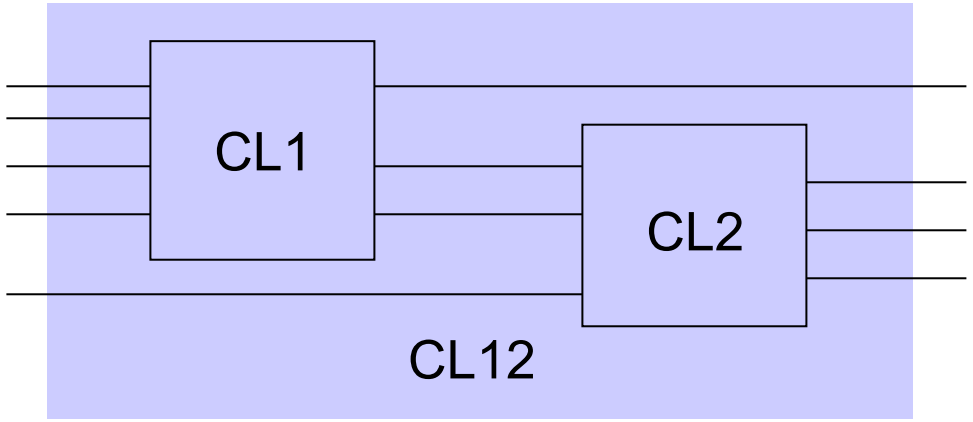
Input determines output

Can compose digital circuits

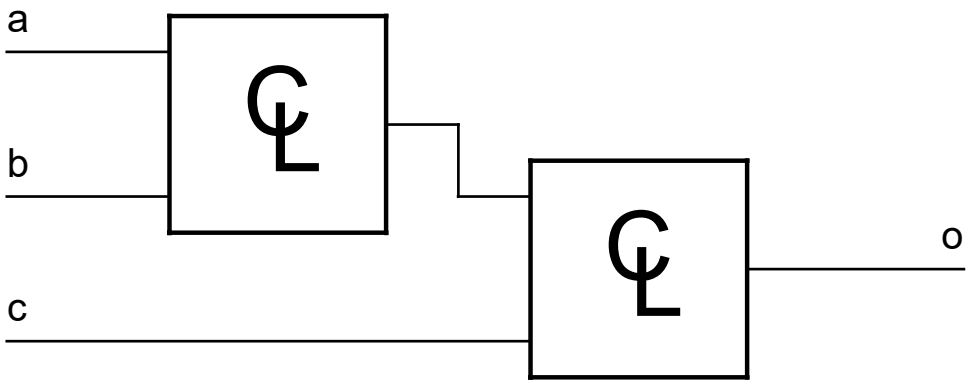


Closure

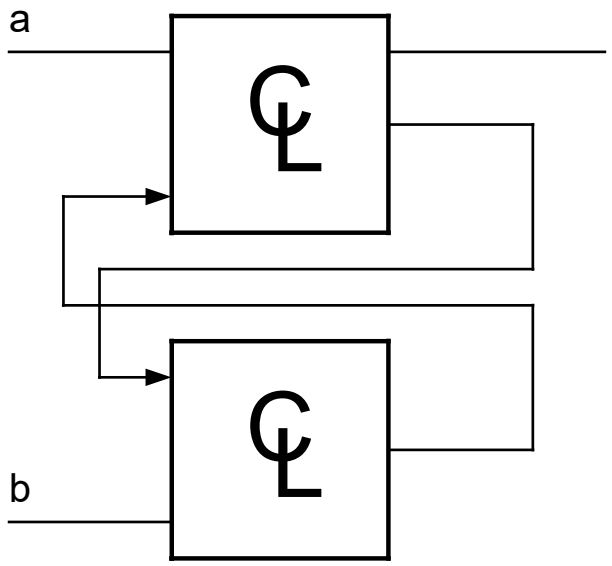
- "Combinational logic circuits are closed under acyclic composition".
  - Norsk: Vi får CL når vi kobler CL etter CL (uten tilbakekobling)



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



YES



NO

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

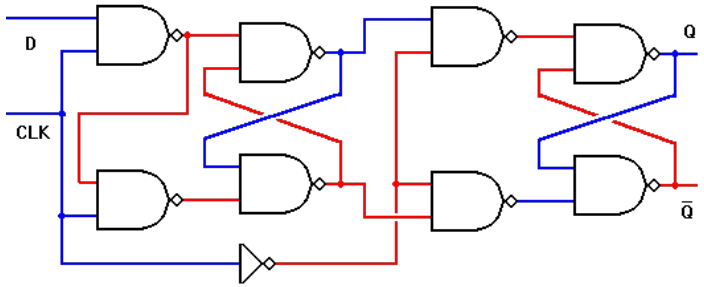
Inferring latches *(not intended as RAM/ROM)* **is bad practice,**
and should be shunned at all costs unless strictly necessary.

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

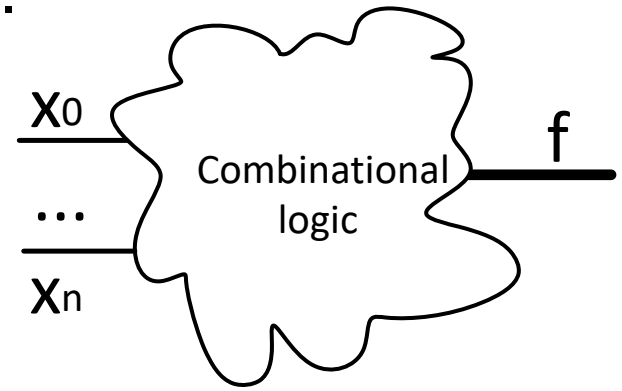
- => Use high level code
  - Unless strictly necessary
  - for *readability (& modifiability)*

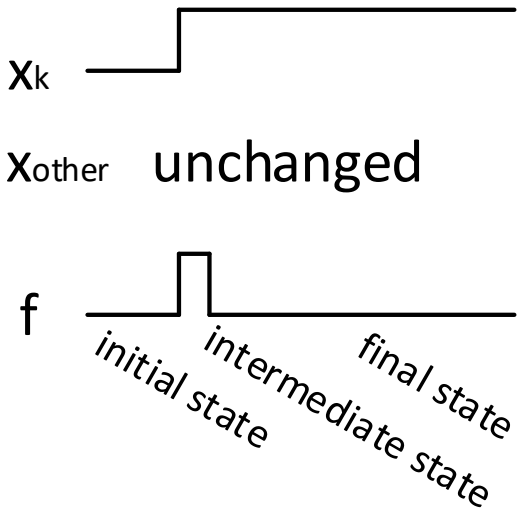



```
architecture data_flow of D_flipflop is
  signal e, f, g, h, i, j, k, l: std_logic;
begin
  -- concurrent statements
  e <= not (D and clk);
  f <= not (e and clk);
  g <= not (e and h);
  h <= not (f and g);
  i <= not (g and not clk);
  j <= not (h and not clk);
  k <= not (l and i);
  l <= not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

Hazards (glitches) in combinational circuits

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

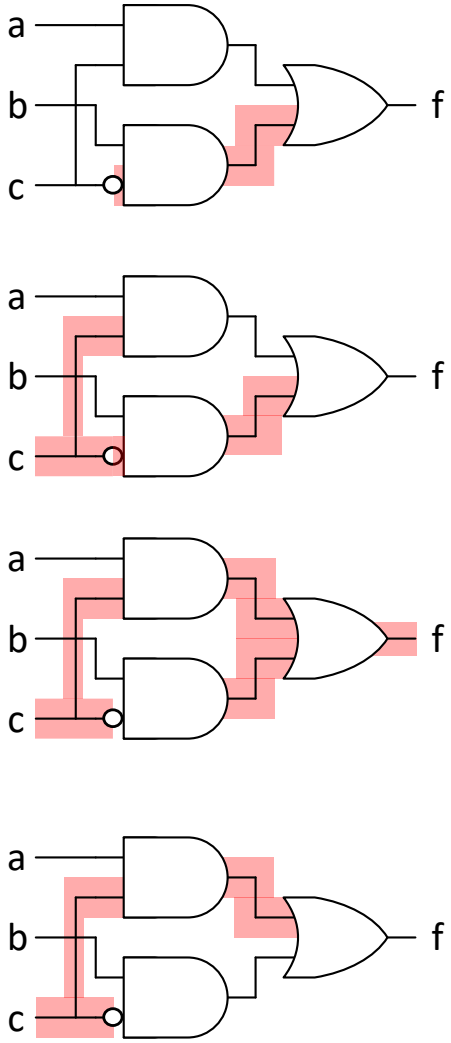




  - With several inputs, there can be several unwanted transitions
    - It doesn't have to be 0 1 0, it can be **X->Y->Z** or **X->Y_1->…-> Y_n-> Z**

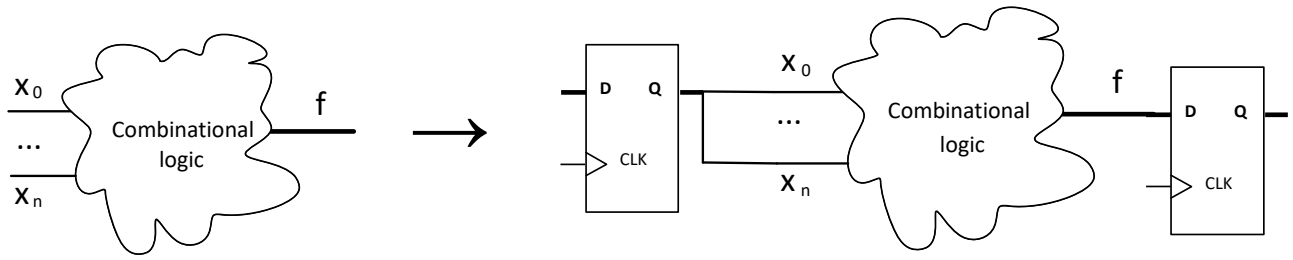
Hazards in combinational design

- Ex: f(a,b,c) = (aΛc) V (bΛc')
  ```
  f <= (a and c) or (b 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)

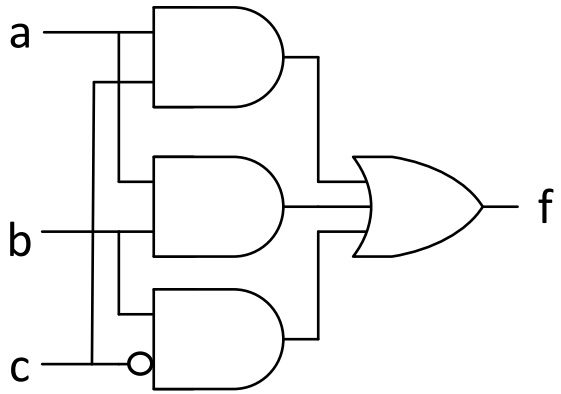


Solutions

- 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
  - Laboriuos process:
    - Typically last resort (ASIC design)
    - Technology dependant solutions
    - *Know it is there:* We do not use this method in this course.



- 3: (Use high level code!)
  - **may not solve every possible issue, *but***
    - Allows synthesizer to decide
  - Synthesizing for FPGAs, = LUTs (problem occurs first at > 4 inputs)

Designer vs tool- example:

Create this circuit:

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

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

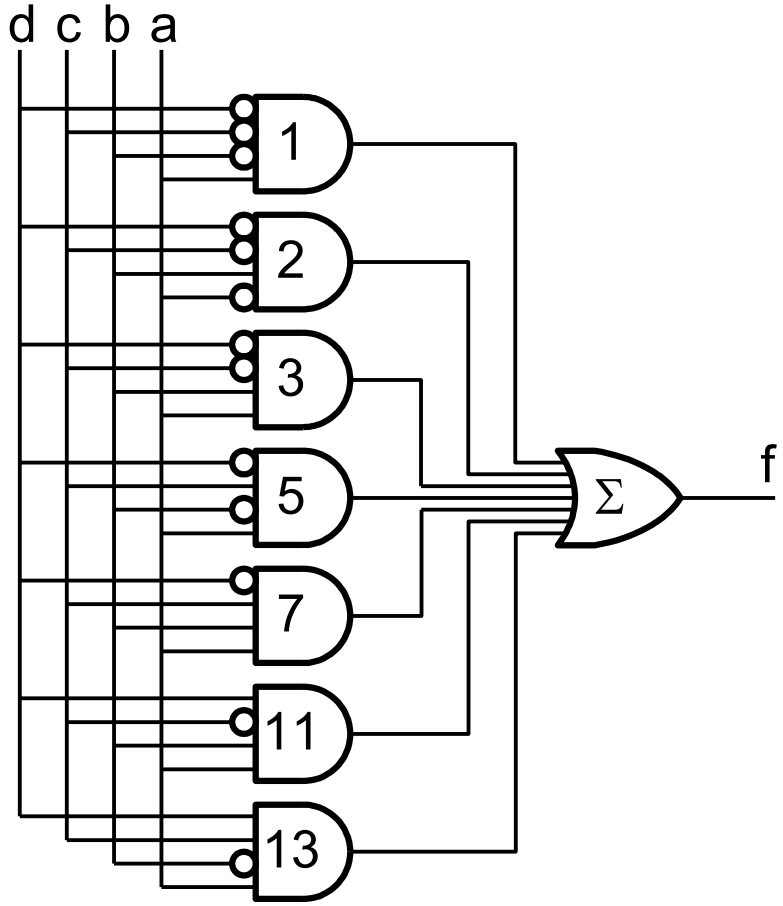
No	dcba	q
0	0000	0
1	0001	1
2	0010	1
3	0011	1
4	0100	0
5	0101	1
6	0110	0
7	0111	1
8	1000	0
9	1001	0
10	1010	0
11	1011	1
12	1100	0
13	1101	1
14	1110	0
15	1111	0

Schematic Logic Diagram

Equation:

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

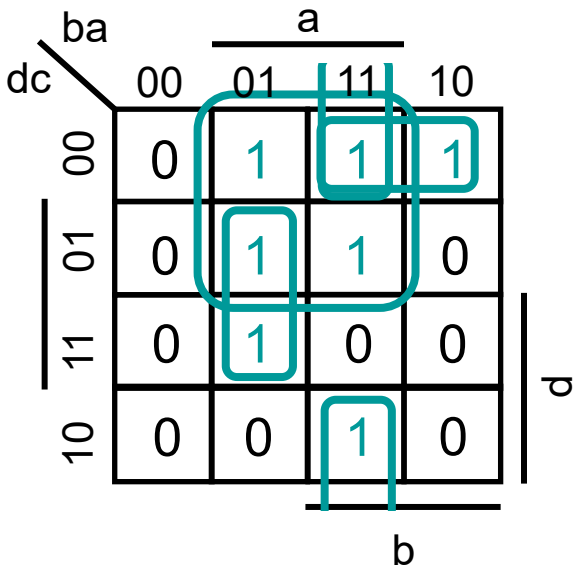
Schematic Logic Diagram:

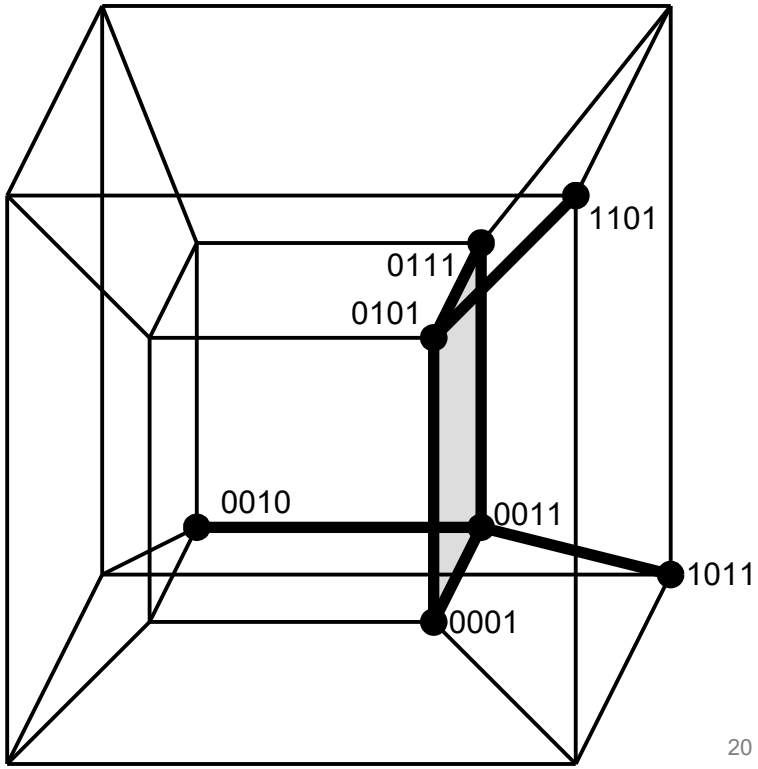


Manual optimization

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

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





Leads to a dataflow (or structural) design.

High level (RTL)
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

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

Solution using «Matching case» = case?

```
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

Solutions using concurrent signal assignment

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

```
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

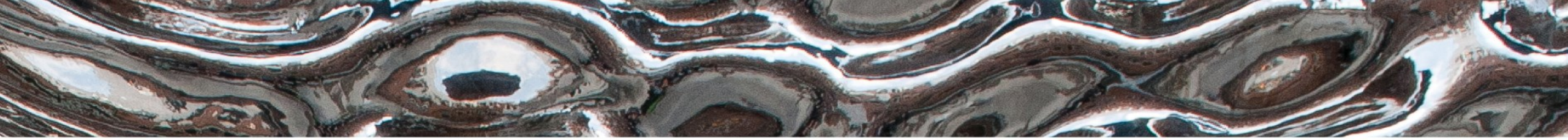
No	dcba	q
0	0000	0
1	0001	1
2	0010	1
3	0011	1
4	0100	0
5	0101	1
6	0110	0
7	0111	1
8	1000	0
9	1001	0
10	1010	0
11	1011	1
12	1100	0
13	1101	1
14	1110	0
15	1111	0

Which one would you prefer reading?

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
      - Ie: 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?
    - Allowes 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..?
```



UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

IN3160, IN4160

Verification part 1 : Simulation

Yngve Hafting

Code examples can be found at
https://github.uio.no/in3160/lectures/

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**.

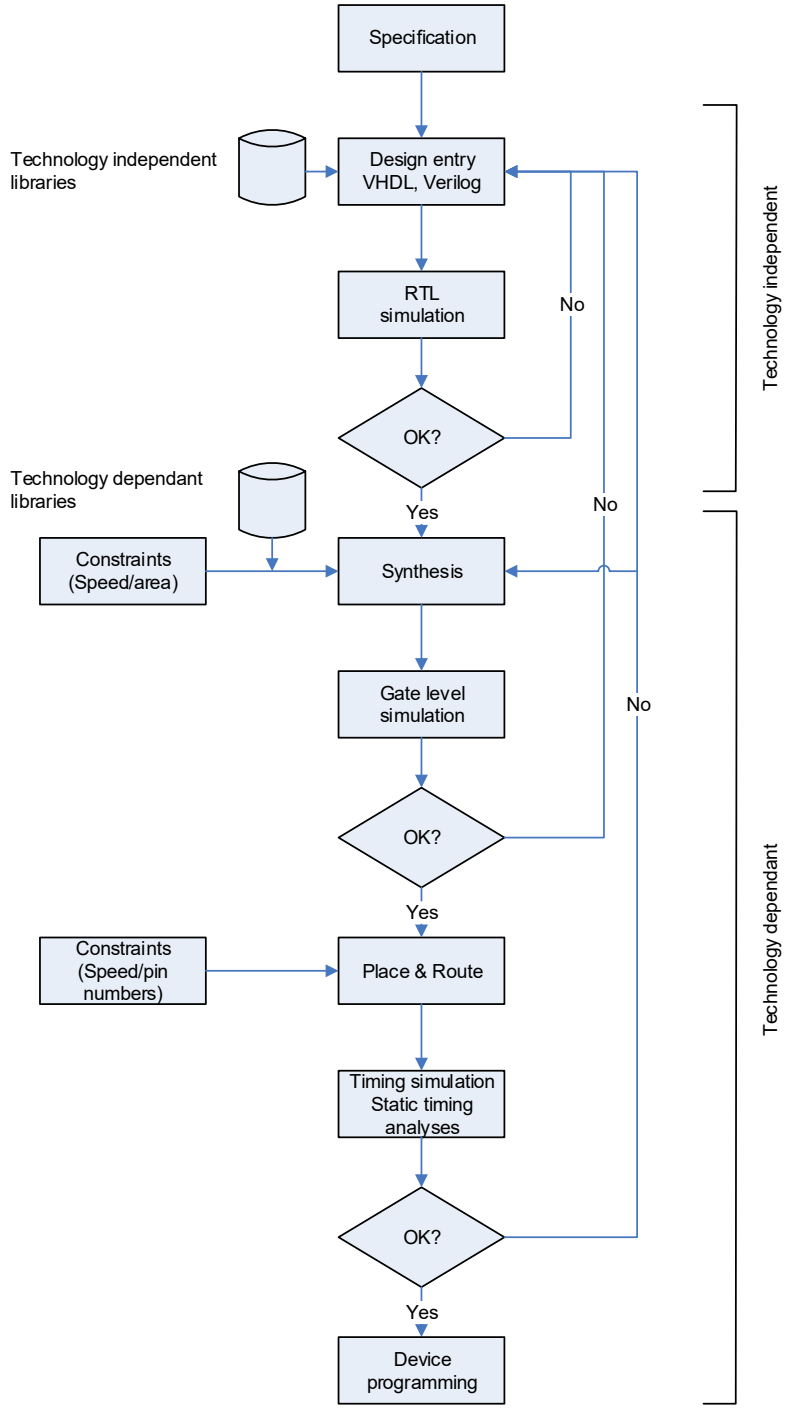
Goals for this lesson:

- Know 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

- Verification part 1:
  - Compilation
  - Simulation
    - Types
      - RTL (functional)
      - Timing
        - » Static timing analysis
        - » dynamic
    - Execution
    - Cycle based
    - Event based

- Assignments and suggested reading



Compilation: Analysis and Elaboration

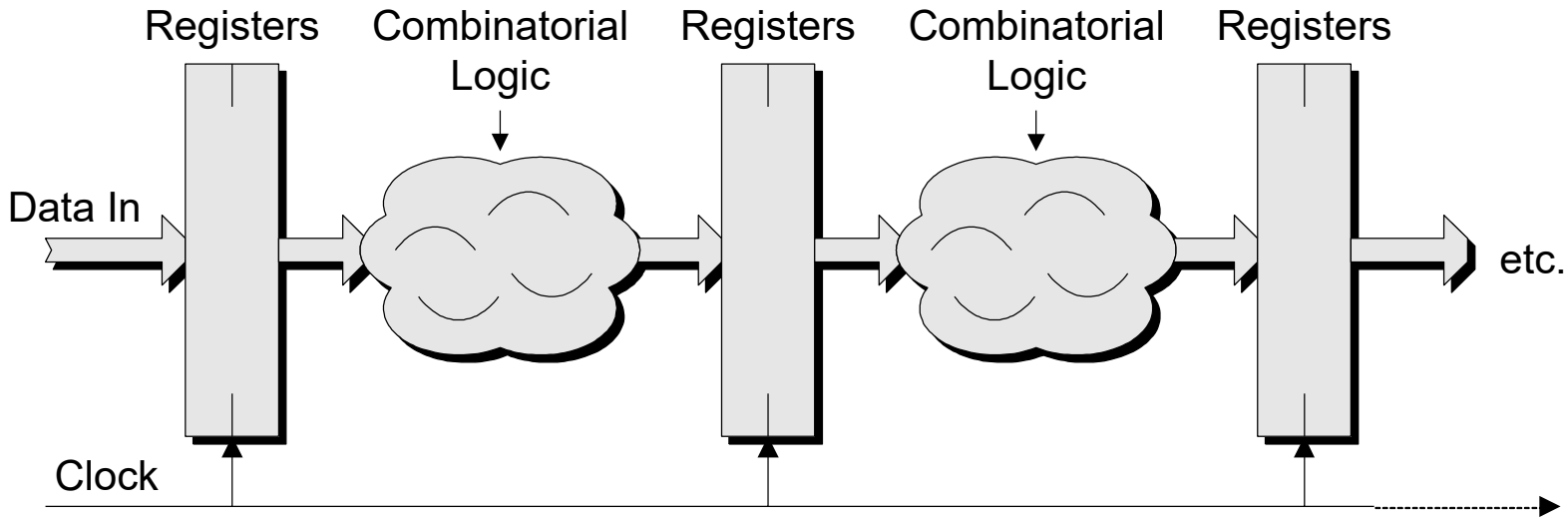
- Analysis
  - **The compiler reads all the files, check syntax, semantics**
  - Compiled result is
    - translated to an intermediate representation
    - stored in (work) library
- Elaboration (requires error free analysis)
  - **Creates design hierarchy**
    - Instantiates entities
    - Creates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

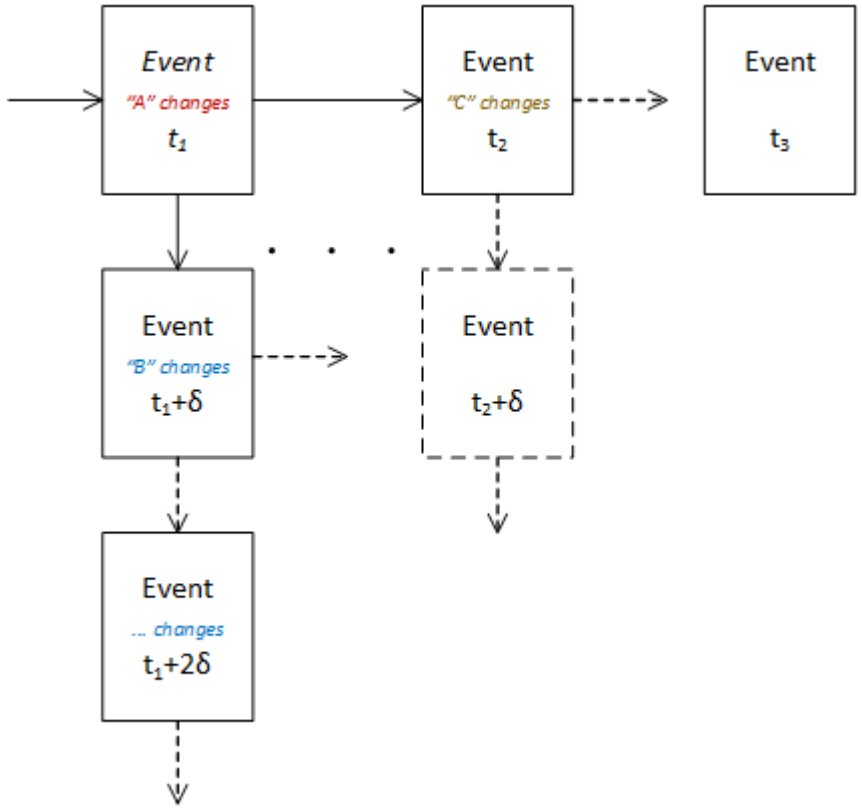
- All signal drivers are modelled with a delay called «delta delay»
  - A delta delay is a delay of 0 time-
    - a mechanism for queuing of events

Event driven simulation

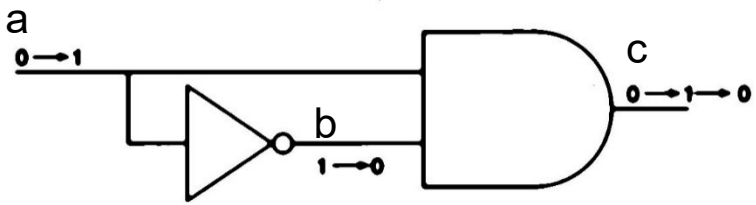
- An event occurs in a time-step when a signal changes (Ex. an input is set from the testbench)
  - All signals that depend on the signal is evaluated
    - Changes are put in the event queue
      - When timing information is provided (post synthesis)
        - » Timing delays are used to schedule events
      - 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

- Event driven simulation ensures all simulators get the same result.



Event queue example : Glitch
Full code (for ref)




```
0ps     a:0     b:U     c:U
0ps     a:0     b:1     c:0
100ps   a:1     b:1     c:0
100ps   a:1     b:0     c:1
100ps   a:1     b:0     c:0
200ps   a:0     b:0     c:0
200ps   a:0     b:1     c:0
300ps   a:1     b:1     c:0
300ps   a:1     b:0     c:1
300ps   a:1     b:0     c:0
400ps   a:0     b:0     c:0
400ps   a:0     b:1     c:0
```

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

entity delta is
  port(
    a : in std_logic;
    c : out std_logic
  );
end entity delta;

architecture dataflow of delta is
  signal b : std_logic;
begin
  b <= not a;
  c <= b and a;
end architecture;
```

```
import cocotb
from cocotb import start_soon
from cocotb.triggers import Edge, Timer, First
from cocotb.utils import get_sim_time

async def monitor(dut):
    while True:
        ta = Edge(dut.a)
        tb = Edge(dut.b)
        tc = Edge(dut.c)
        await First(ta, tb, tc)
        print(f"{get_sim_time(units='ps'):{9}.0f}ps    ",
              f"a:{(dut.a.value)}    ",
              f"b:{(dut.b.value)}    ",
              f"c:{(dut.c.value)}")

@cocotb.test()
async def main_test(dut):
    dut.a.value = 0
    start_soon(monitor(dut))
    for i in range(5):
        await Timer(100, units='ps')
        dut.a.value = not dut.a.value
```

Waiting for ReadOnly()

```
  0ps     a:0     b:1     c:0
100ps     a:1     b:0     c:0
200ps     a:0     b:1     c:0
300ps     a:1     b:0     c:0
400ps     a:0     b:1     c:0
```

IN a physical circuit
We *may* see all glitches

Glitches can be hidden in simulation results and waveform diagrams

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

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

async def monitor(dut):
    while True:
        ta = Edge(dut.a)
        tb = Edge(dut.b)
        tc = Edge(dut.c)
        await First(ta, tb, tc)
        await ReadOnly()
        print(f"{get_sim_time(units='ps'):{9}.0f}ps    ",
              f"a:{(dut.a.value)}    ",
              f"b:{(dut.b.value)}    ",
              f"c:{(dut.c.value)}")

@cocotb.test()
async def main_test(dut):
    dut.a.value = 0
    start_soon(monitor(dut))
    for i in range(5):
        await Timer(100, units='ps')
        dut.a.value = not dut.a.value
```

Source https://docs.cocotb.org/en/stable/triggers.html#

Cocotb Triggers -- `await <trigger>`

`ReadOnly``()`

Waits until all delta delays has settled

useful for *most* checks

`ReadWrite``()`

get out of ReadOnly before next value are set

`Edge``(`*`signal`*`)`

Waits for any change in the signal

`RisingEdge``(`*`signal`*`),` **`FallingEdge`**`(`*`signal`*`)`

`Timer``(`*`value`*`,` *`unit`*`="ps")`

Waits the exact simulation time

`ClockCycles``(`*`signal`*`,` *`num_cycles`*`,` *`rising=True`*`)`

Waits a number of (rising) edges

`First``(`*`*triggers`*`)`

Waits for the first trigger in the list

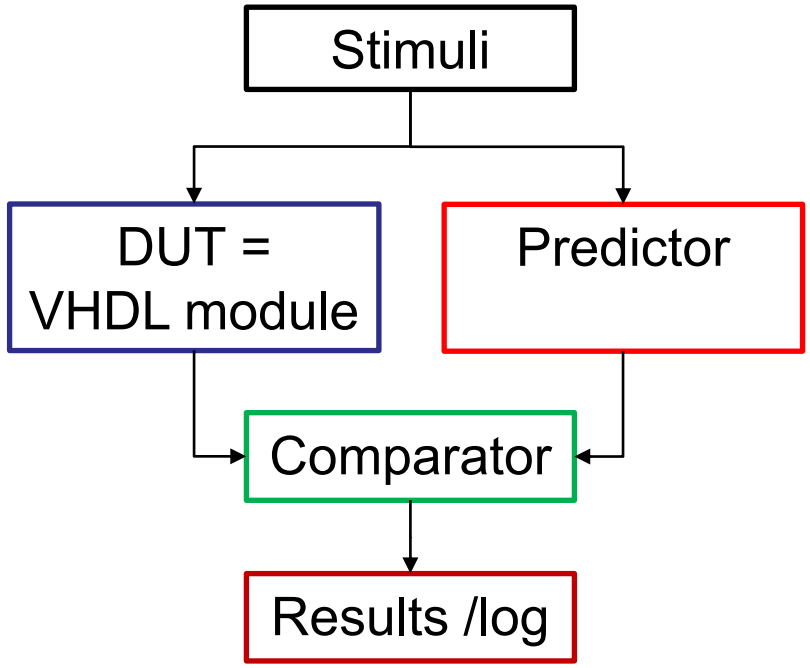
`Combine``(`*`*triggers`*`)`

Waits for all triggers in the list

RTL simulation practical example

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

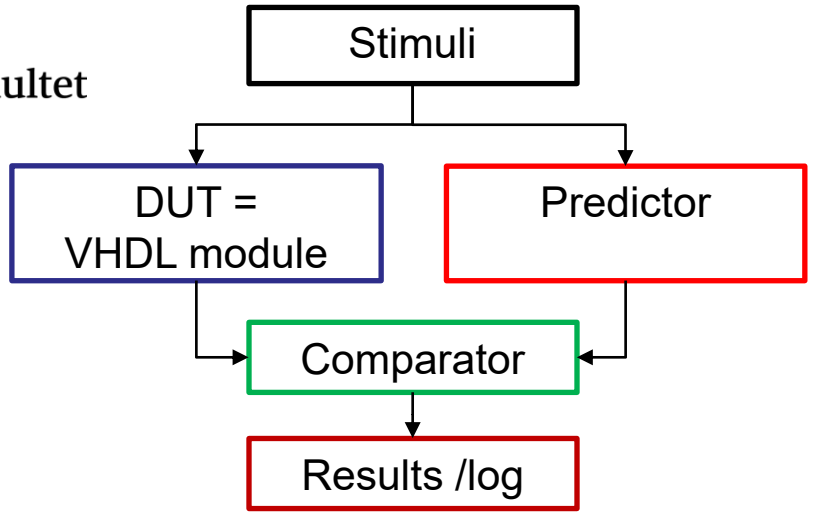
General testbench layout



- **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.

UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Cocotb testbench



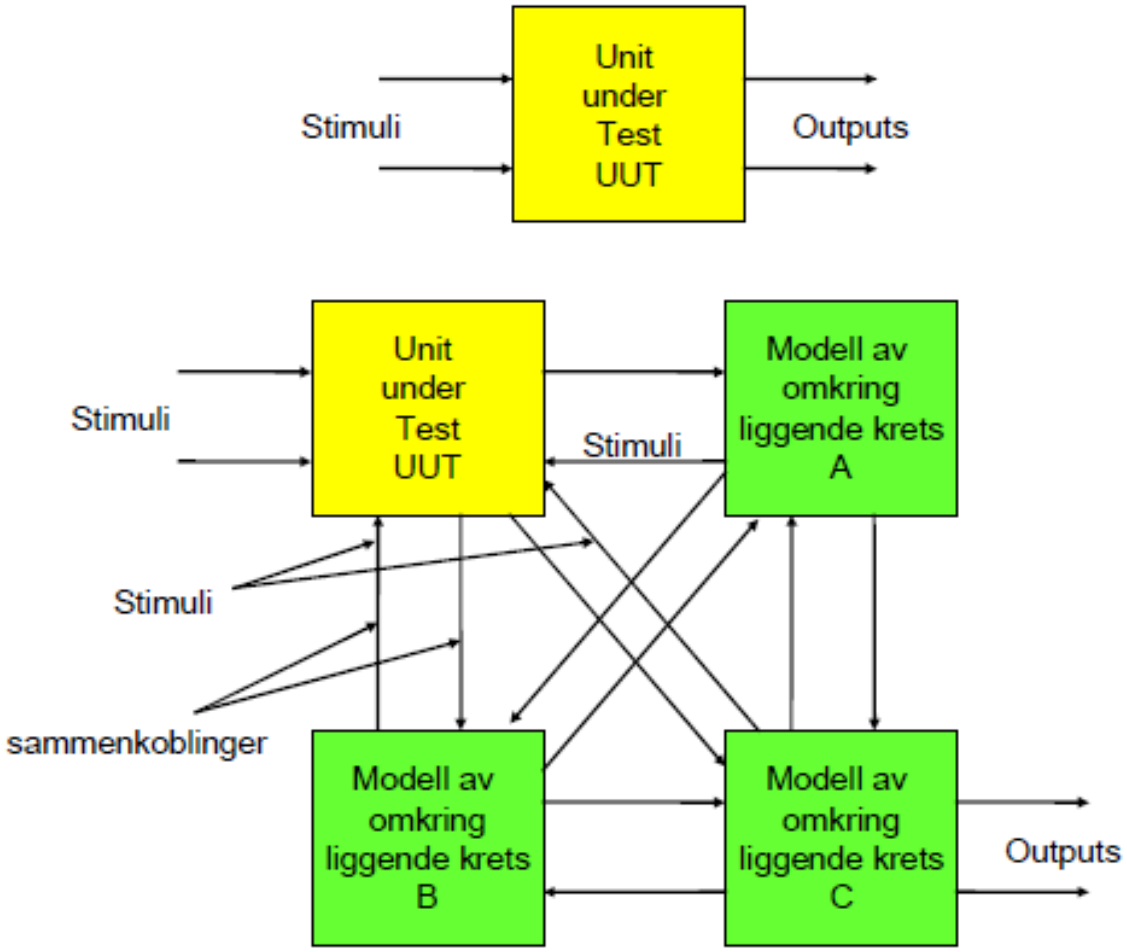

```python
import cocotb
from cocotb import start_soon
from cocotb.triggers import Edge, Timer, ReadOnly
import random

# 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 True:
    # 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))
```

```python
# Generate all data
async def stimuli_generator(dut):
   for i in range( 2**len(dut.input)):
     dut.input.value = vector
     await Timer(1, units= 'ns')

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

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

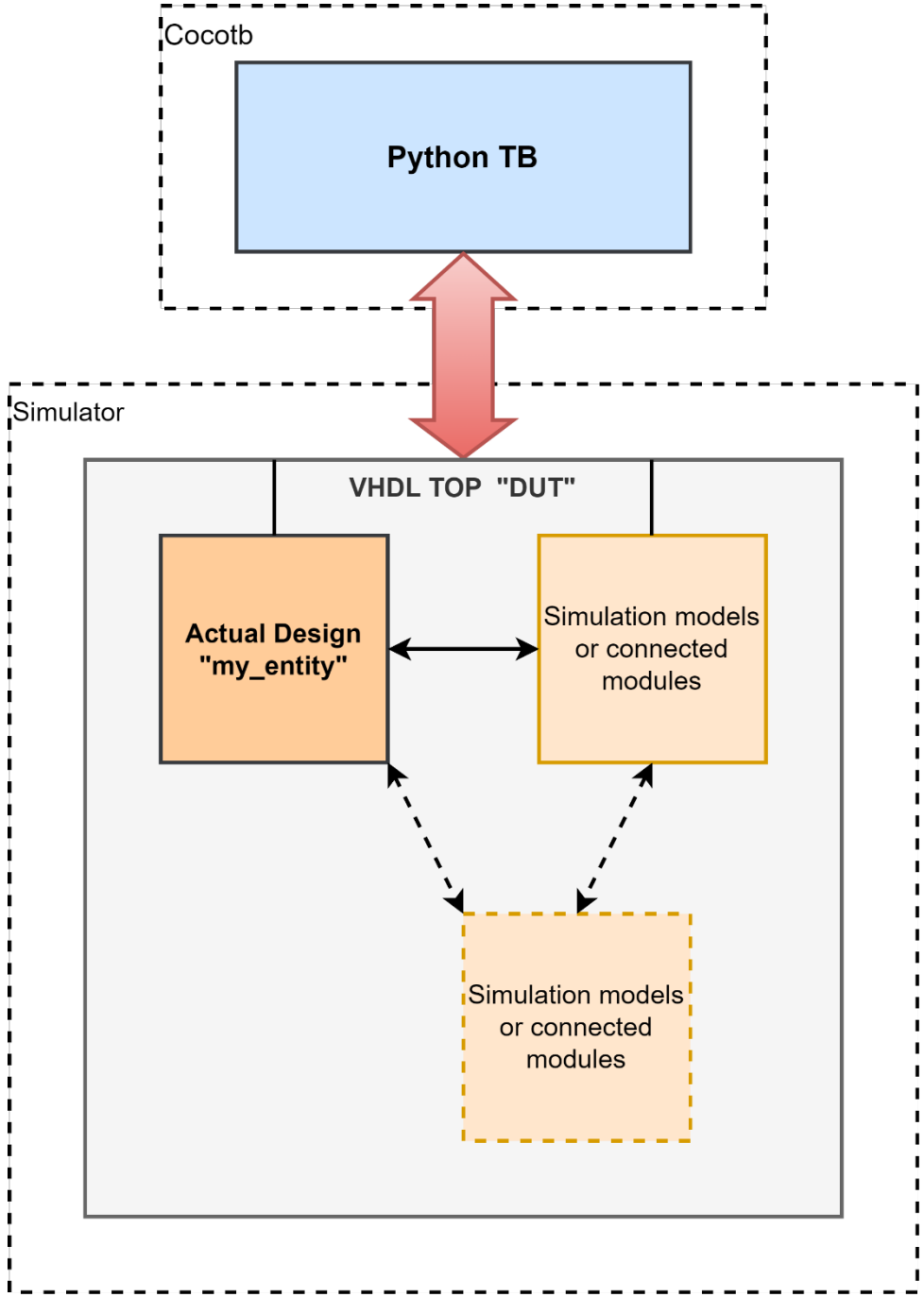
Testbench structures

- Multiple modules are often required to be tested together
  - Other design modules
  - Premade modules (Ips)
    - Bus functional models etc.
  - *Can be required to achieve certifications in industry..*



Multiple modules in cocotb

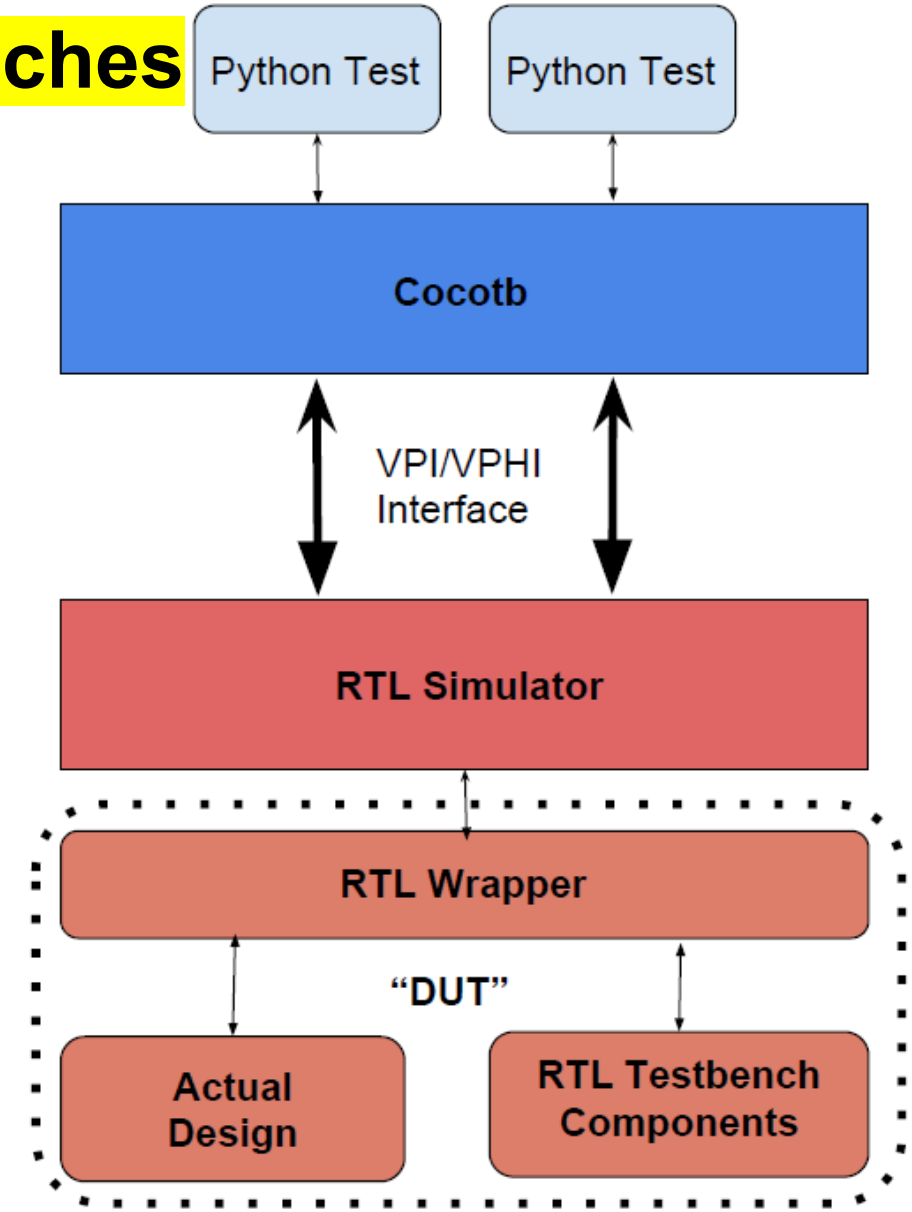
- Only one toplevel can be used
  - Multiple modules =>
    - Create a top entity connecting all modules.
- Python TB can access all hierarchy
  - `"Dut.my_entity.my_signal.value = ..."`
    - Use dot notation to go deeper in hierarchy



Kilde: Ben Rosser (Penn) Cocotb for Cern Micorelectronics 2018 P14,16

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
    - 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>)
        - » Normally all delta delays should finish before reading

```
async def stimuli_generator(dut):
    ''' Generates all data for this tesbench'''
    for i in range( 2**len(dut.input)):
      dut.input.value = i
      await Timer(1, units= 'ns')


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

More Cocotb...

- 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

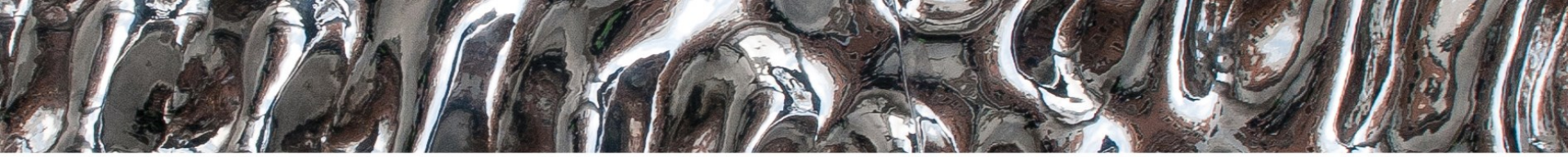
Combinational logic

- 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»



UiO : Department of Informatics
University of Oslo

IN 3160, IN4160

More VHDL:
Processes, signals and variables
conditional statements
Structural design

Yngve Hafting

Messages

Content

- How VHDL processes work
  - Example with
    - Sensitivity
    - Signals
    - Variables
  - Compared to cocotb testbench code

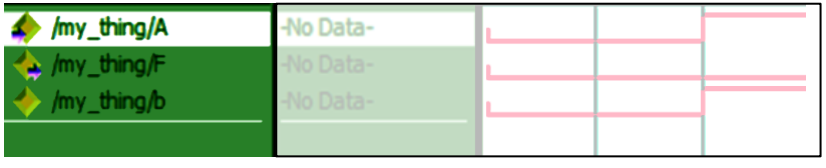
VHDL processes

- A process must work in a predictable, deterministic way for both creating and simulating circuits

- 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: `process(all) is...`
    - Use clock for sequential logic: `process(clk)...`
      - or (clk, reset) when asynchronous reset is desired

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity My_thing is
    port(A: in STD_LOGIC;
         F: out STD_LOGIC
    );
end entity My_thing;


architecture Behavioral of My_thing is
    signal b : STD_LOGIC;
begin
    signal_update: process(a)
    begin
        if A = '1' then b <= '1';
        else b <= '0';
        end if;

        if b = '1' then F <= '1';
        else F <= '0';
        end if;

    end process;
end architecture Behavioral;
```

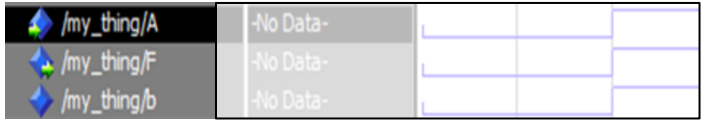


Sensitivity list

What happens with F?

Assume a changes from ‘0’ to ‘1’

```
signal_update: process(a,b)
begin
  if a = '1' then
    b <= '1';
  else
    b <= '0';
  end if;
  if b = '1' then
    f <= '1';
  else
    f <= '0';
  end if;
end process;
```



Signals and variables

- Signals
  - A signal can be used within the **whole architecture**
  - *Changes value when simulation exits a process* (or statement)
- Variables
  - Can only be **used locally**
    - within a process, function or procedure
  - Assigned using ":=" (Ex: `var := '1';`)
  - Changes value **immediately** in simulation
    - Immediately = based on position, *read from top to bottom*.
    - *can have multiple values* within one process.
  - Variables are useful to keep intermediate results in algorithms
    - Subprograms initialize variables every run.
    - Process variables initialize once, *when simulation starts*
- Both signals and variables can be used for storage
  - Both FFs and latches.
  - Variables that are read «before» written will accomplish this = BAD PRACTICE!…

Signals vs. variables (sequential logic example)

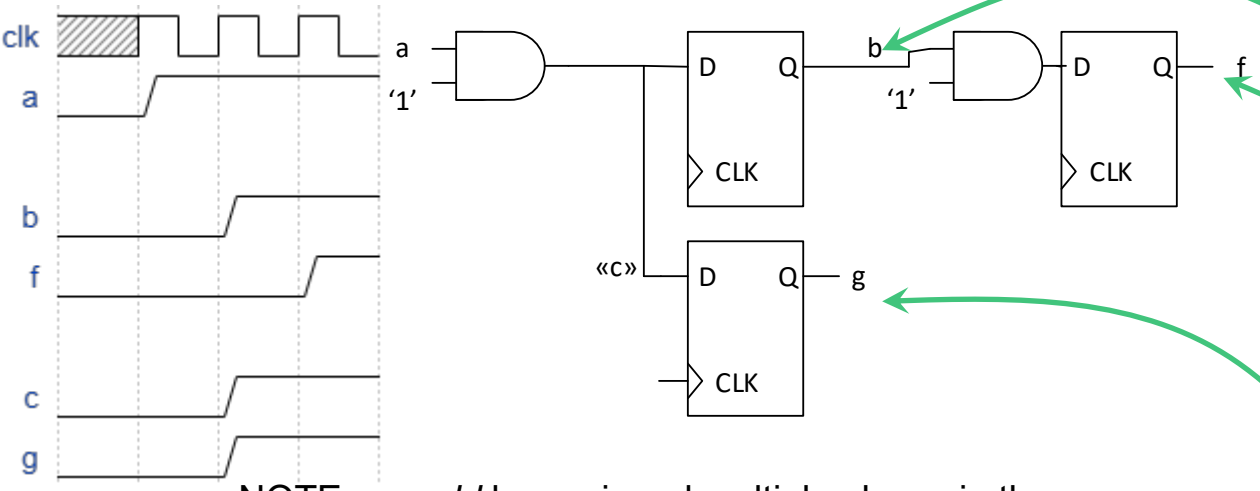
- Exercise:
- Assume all signals are 0, then
  - signal a changes from 0 to 1.

- On which clock cycles does f and g change value; first, second, third?

Try for 1 minute:

Time's up...

```
signal_var_update : process(clk)
  variable c : std_logic;
begin
   if rising_edge(clk) then
      if a = '1' then
         b <= '1';
         c := '1';
      else
         b <= '0';
         c := '0';
      end if;
      if b = '1' then
         f <= '1';
      else
         f <= '0';
      end if;
      if c = '1' then
         g <= '1';
      else
         g <= '0';
      end if;
   end if;
end process;
```




```
signal_var_update :
process(clk)
  variable c : std_logic;
begin
  if rising_edge(clk) then
    if a = '1' then
      b <= '1';
      c := '1';
    else
      b <= '0';
      c := '0';
    end if;
    if b = '1' then
      f <= '1';
    else
      f <= '0';
    end if;
    if c = '1' then
      g <= '1';
    else
      g <= '0';
    end if;
  end if;
end process;
```

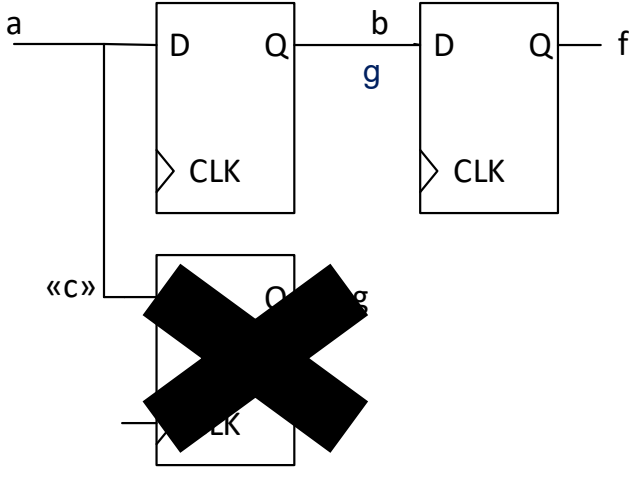
NOTE: c *could* be assigned multiple places in the process.

How would that affect the diagram..?

Variables update «*immediately*»
Signals are assigned «*where*» the process ends
when the process statement updates as a whole

Digression:

- Simplified...




```
signal_var_update :
process(clk)
  variable c : std_logic;
begin
   if rising_edge(clk) then
          b <= a;
          c := a;
          f <= b;
          g <= c; -- g <= a
   end if;
end process;
```

```
signal_var_update :
process(clk)
  variable c : std_logic;
begin
   if rising_edge(clk) then
      if a = '1' then
         b <= '1';
         c := '1';
      else
         b <= '0';
         c := '0';
      end if;
      if b = '1' then
         f <= '1';
      else
         f <= '0';
      end if;
      if c = '1' then
         g <= '1';
      else
         g <= '0';
      end if;
   end if;
end process;
```

Default values and positional priority in processes

```
architecture Sequential2 of priority is
begin
  process (a) is
  begin
    valid <= '1';
    if a(3)='1' then
      y <= "11";
    elsif a(2)='1' then
      y <= "10";
    elsif a(1)='1' then
      y <= "01";
    elsif a(0)='1' then
      y <= "00";
    else
      valid <= '0';
      y <= "00";
    end if;
  end process;
end architecture Sequential2;
```

- Default values
  - Ensures we always "have an output value" (*avoiding latches*).
  - The last assignment of a signal takes precedence
  - This works because processes are compiled sequentially…
- Only use default values on top in processes
  - Don’t bury them in nested if's...
    - Readability and maintainability suffer if you do..
- Default values are commonly used for state machine outputs
- Using positional priority except for default values
  - is bad practice

How processes work with signals and variables...

- Signals
  - Represent physical wires and drivers in the architecture
    - A wire can only have a single voltage at any given time.
  - Are updated once in a process invocation
    - This happens only at process exit.
      - No intermediate values are held.
      - A value that has been changed cannot be read as changed within the process.
    - When assigned multiple times within a process, the latest will be given priority
      - Allows for default values
    - Makes inferring storage elements (FFs+latches) deterministic and comprehensible.

How processes work with signals and variables...

- Variables
  - Variables *are local* to the process.
    - They must be both assigned and read within a process
  - Their value is *intended for intermediate purposes*
    - Making code more readable by turning complex statements into several simpler statements
    - By taking value(s) that can be used within the process
  - They *can* be given values multiple times within a single process invocation
    - *Doing so- is generally not a good idea*
  - Placement determines whether they will infer storage elements (latches and flipflops)!
    - *Reading before writing to* = storage
    - **Using variables for storage is considered bad practice** in most circumstances

Processes and the event queue (in simulation)

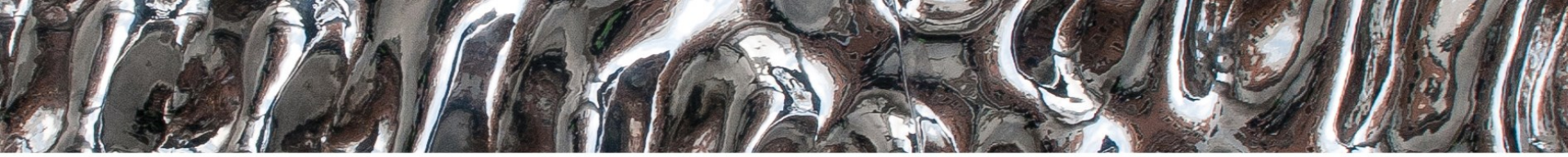
- Simulation uses an event queue to keep track of what happens.

- A process is invoked as a result of a change in one of the signals in the sensitivity list.
  - The whole process is "run" through «within that delta delay».
  - Each `signal` assignment is added to the queue of delta delays
    - **Only changes in signals that are in the sensitivity list will trigger the process again**.
  - `variable` updates *does not create* new events.
    - (They are updated immidiately…)

Cocotb and the event queue

- Cocotb code only runs within a delta delay
  - From triggered to next wait statement (ie `await <trigger>`)
  - Similar to a vhdl process but
    - `await` statements may be put almost anywhere
    - VHDL processes are always run through completely

- Each cocotb output (`dut.value = <new value>`) creates a new event in the simulation.



UiO : Department of Informatics
University of Oslo

IN 3160, IN4160

VHDL
conditional statements and structural design

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 conditional statements in VHDL
  - .
  - how to implement these structures using VHDL
    - *If, case, when-else, select*
    - *Loops*
    - *Type casting*
    - *Shift operators*
    - *Dataflow vs RTL descriptions*
- Know how to generate complex structures in VHDL
  - generate

Section overview

- VHDL:
  - Sensitivity list
  - Signals and variables example
  - If, case, when-else, select
  - Loops
  - Structural coding
    - Generate
    - Generics

Next lesson: Building blocks

Decoders vs encoders
Decoder
Multiplexer
Encoders
Arbiters
Shifters
Comparators
ROM
RAM

If and case in **VHDL Processes**

- ***if*** and ***case*** are used much like in other programming languages like C, Java etc.
  - *Their similarity in syntax may lead to errors if we do not understand how they work in digital circuits...*
  - If-tests can test on multiple signals/variables
    - built in priority
  - Case-tests uses single signal/variable (vector=OK)
    - No built in priority because the same signal are being used everywhere in the test

If

- Must be in process
- **Multiple conditions**
- **Multiple targets**
- prioritizes

- First option has priority
  - (think of two-input multiplexers)
- Can be used to infer latches and Flipflops
  - FF when edge triggered (`if rising_edge(clk) then...`)
  - **Latch when not sufficiently specified!**
    - This is a trap, avoid this!
- Can be nested using «`elsif`»
  - Can replace any other conditional statement
    - *Not recommended!*
  - Avoid deep nesting
  - ~4 degrees should be maximum…

If example (*all input specified*):

inp1	inp2	a	b
1	1	1	1
1	0	1	0
0	1	0	Latched
0	0	0	Latched

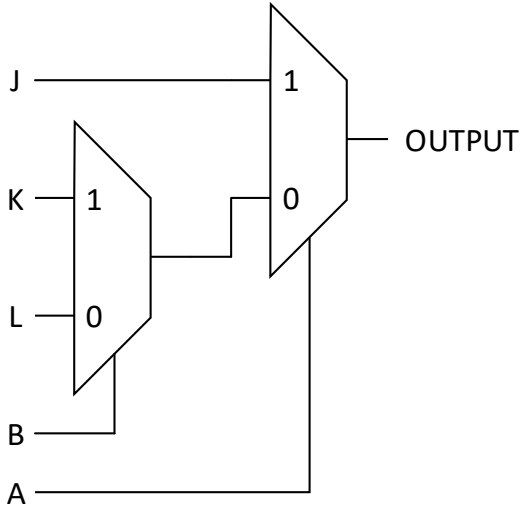
```
process(all) is
 begin
   if inp1 then
     if inp2 then
       a <= '1';
       b <= '1';
     else
       a <= '1';
       b <= '0';
     end if;
   else
     a <= '0';
   end if;
 end process;
```

```
process(all) is
 begin
   if inp1 then
     a <= '1';
     b <= inp2;
   else
     a <= '0';
     -- b ass. missing
   end if;
 end process;
```

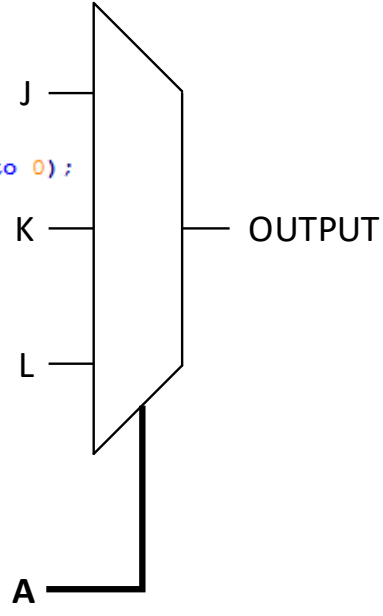
Always specify all outputs for all conditions of inputs!

if../case..

```
entity My_thing is
    port(A,B,J,K,L: in STD_LOGIC;
        OUTPUT: out STD_LOGIC);
end entity My_thing;

architecture prioritized of My_thing is
begin
    process(all)
    begin
        if A = '1' then
            OUTPUT <= J;
        elsif B = '1' then
            OUTPUT <= K;
        else
            OUTPUT <= L;
        end if;
    end process;
end architecture prioritized;
```




```
entity My_thing is
    port(A : in STD_LOGIC_VECTOR(1 downto 0);
        J,K,L: in STD_LOGIC;
        OUTPUT: out STD_LOGIC);
end entity My_thing;

architecture nonpri of My_thing is
begin
    case A is
        when "01" =>
            OUTPUT <= J;
        when "10" =>
            OUTPUT <= K;
        when others =>
            OUTPUT <= L;
    end case;
end architecture nonpri;
```



If nesting vs. chaining (*using elsif*)

```
process(all) is
begin
  if (input = 4d"1") then
    isprime <= '1';
  else
    if  (input = 4d"2") then
      isprime <= '1';
    else
      if (input = 4d"3") then
        isprime <= '1';
      ...
      end if;
    end if;
  else isprime <= '0';
  end if;
end process;
```

```
process(all) is
begin
  if (input = 4d"1") then isprime <= '1';
  elsif  (input = 4d"2") then isprime <= '1';
  elsif  (input = 4d"3") then isprime <= '1';
  ...
  else isprime <= '0';
  end if;
end process;
```

If nesting for priority – danger zone

Sometimes it can make sense to use nesting
- clocked processes and state machines

- It is easy infer latches
  - When not all input options are covered
  - When some output is not covered for all options

Consider other options when creating CL
- *improve readability*
- *Reduce risk for latches*

- *It is OK to nest other statements within if…*
  - *select …*
  - *when … else*
  - *case …*

```
process(all) is
begin
  if (inp1 = a) then
    if (inp2 = b) then
      if (inp3 = c) then
        <statement 1>
        <statement 2>
      else
        <statement 3>
      end if;
    end if;
    else
      <statement 4>
  end if;
end process;
```

Example

```
library ieee;
  use ieee.std_logic_1164.all;

entity latches is
  port(
    invec : in std_logic_vector(1 downto 0);
    outvec  : out std_logic_vector(3 downto 0);

    input : in std_logic;
    out1, out2 : out std_logic
  );
end entity latches;
```

- **Nesting if-statements will conceal these errors easily**, thus providing an endless source of errors

```
architecture poor of latches is
begin
  -- if invec = "11" => outvec is latched
  missing_input: process(all) is
  begin
    if invec = "00" then
      outvec <= "0000";
    elsif invec = "01" then
      outvec <= "1110";
    elsif invec = "10" then
      outvec <= "0110";
    end if;
  end process;

  -- if input='1' then out2 is latched.
  -- if input='0' then out1 is latched.
  missing_output: process(all) is
  begin
    if input then
      out1 <= '1';
    else
      out2 <= '0';
    end if;
  end process;

end architecture poor;
```

Case

- Must be in process
- **single input vector**
- **Multiple targets**
- Every alternative has same priority

- Every option for *input* must be declared
  - **'`when others`'** *can* be used
    - be wary of changes in input *type*…
  - Can infer latches too…
    - When not defining all outputs for all inputs
- Matching case- «case?»
  - Allowes for don't care's

```
process(input) is
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;
```

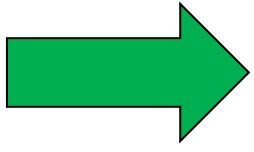
The typical use-case for case is state machines.

Case is excellent when you want to set several output vectors depending on one state vector.

Case creating latches:

Default values can be a good solution when using case statements.

'null' statement should only be used in CL when using default values for all outputs.



```
process(input) is
begin
 isprime <= '0';
 isfour  <= '0';
  case input is
    when x"1" | x"2" | x"3" | x"5" | x"7" | x"b" | x"d" =>
      isprime <= '1';
      isfour <= '0';
    when x"4" =>
      isprime <= '0';
      isfour <= '1';
    when others =>
      null;
  end case;
end process;
```

= latch inferred

When … else

- Can be used concurrently (outside processes).
- **Multiple conditions**
- **Single target**
- prioritizes

```
isprime <=
    '1' when input = x"1" else
    '1' when input = x"2" else
    '1' when input = x"3" else
    '1' when input = x"5" else
    '1' when input = x"7" else
    '1' when input = x"b" else
    '1' when input = x"d" else
    '0';
```

- Can replace if statements for *single target*
- Can infer FF's/latches

```
q <= '0' when reset else 'd' when rising_edge(clk);

a <= b when en;
```

- Compact
  - Suitable when complexity is low

^^ always keep '**else**' in mind…

`with … select` "Selected statement"

- Can be used concurrently
- **single input vector**
- **Single target**
  - Must have all input cases defined

```
with input select isprime <=
  '1' when x"1" | x"2" | x"3" | x"5" | x"7" | x"b" | x"d",
  '0' when others;
```

- Can also infer latches
  - *Least likely*
    - Feedback obvious ☺

```
with a select g <=
   16d"1" when 16d"1",
   16d"4" when 16d"2",
   16d"8" when 16d"3",
   g when others;
```

- Compact and readable

If, case, when … else, with select - summary

- When in doubt…
  - Try '`with…select`'
    - This will force you to make visible choices.

Statement	Targets	Conditions	Process
`if`	*Multiple*	*Multiple*	Required
`case`	*Multiple*	**Single**	Required
`when … else`	**Single**	*Multiple*	Optional
`with … select`	**Single**	**Single**	Optional

- *Only* use '`if`'…
  - When you need to prioritize conditions…
  - and have multiple targets
    - Typically used for clocked processes.

- It is fine to `use select…` or `when/else` inside `if` and `case`
  - *Do you need if inside if?..*
  - *Case inside case? ..*

  - Readability suffers when nesting several levels of if or case

Whatever you choose, keep the following in mind:

define
- all outputs *for*
- all conditions

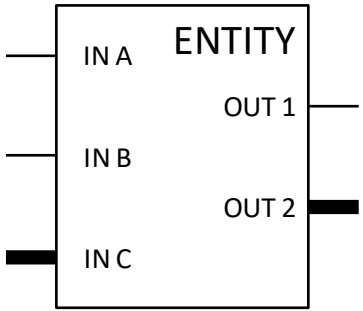
Structural Design

Entity/architecture

REPETITION

- Entity and architecture are the two most fundamental building blocks in VHDL
- Entity
  - Connection to the surroundings
  - Port description
    - Input/output/bi-directional signals
- Architecture
  - Describes behavior
  - An entity can have many architectures
  - Can be used to describe the circuit on several levels of abstraction:
    - Behavioral (for simulation)
    - RTL (Register Transfer Level)
    - Dataflow
    - Structural
      - Post synthesis (netlist)
      - Post Place & Route (netlist + timing)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity My_thing is
    generic(width: integer := 8);
    port(INA, INB : in STD_LOGIC;
         INC : in STD_LOGIC_VECTOR(width-1 downto 0);
         OUT1: out STD_LOGIC;
         OUT2: out STD_LOGIC_VECTOR( width/2 - 1 downto 0)
    );
end entity My_thing;
```



Generics

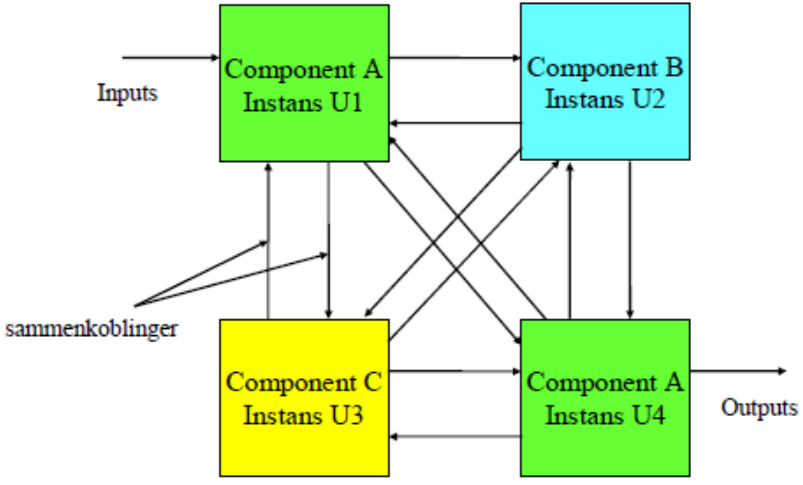
- In addition to the port description an entity can have a generic description
- ***Generics*** can be used to make parameterized components (generic)
  - can be used for structural information
    - both synthesis and simulation
  - can be used for timing information
    - for simulation only
  - Example 1:
    - Time delay can vary between circuits, but the behavior is the same
  - Example 2:
    - The number of bits can vary between circuits, but the behavior is the same

```
24: entity And2 is
25:   generic (delay : DELAY_LENGTH := 10 ns);
26:   port (x, y : in BIT; z: out BIT);
27: end entity And2;
28:
29: architecture ex2 of And2 is
30: begin
31:   z <= x and y after delay;
32: end architecture ex2;
```

```
architecture Structural of My_tb is
    component And2
        port( x,y : in BIT; z : out BIT);
    end component;
    signal a,b,c : BIT;
begin
    MY_COMP1: And2
        generic map(delay => 1 us);
        port map(x=>a, y=>b, z=>c);
end architecture Structural;
```

DELAY_LENGTH is a subtype of the type time from the predefined (always in use) package “std”

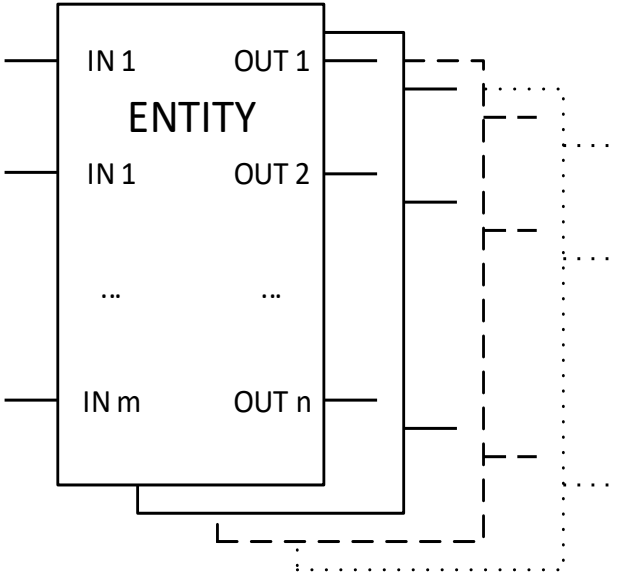
Structural design



- Every Component has an underlying Entity/architecture pair
- Components can be used multiple times
- We *can* make a hierarchic design with as many levels we want
  - Try keep design hierarchy manageable...
  - Structural top layer is the norm (for multiple modules)

Structural design

- Reuse of modules (entities and architectures)
- Generic modules (generics)
  - For example scalable bus widths
  - Configurable functionality
- Breaking up big designs to smaller and more manageable building blocks
  - Think functional blocks
  - Connection of functional blocks (entities/components/modules)
- Easier to collaborate within a design team
  - Well defined interface between modules
- Any entity-/architecture pair can be used as a building block in a structural description
  - Pairing of components



Structural design (netlist)

```
25: architecture netlist2 of comb_function is
26:
27:   component And2 is
28:     port (x, y : in BIT; z: out BIT);
29:   end component And2;
30:
31:   component Or2 is
32:     port (x, y : in BIT; z: out BIT);
33:   end component Or2;
34:
35:   component Not1 is
36:     port (x : in BIT; z: out BIT);
37:   end component Not1;
38:
39:   signal p, q, r : BIT;
40:
41: begin
42:   g1: Not1 port map (a, p);
43:   g2: And2 port map (p, b, q);
44:   g3: And2 port map (a, c, r);
45:   g4: Or2 port map (q, r, z);
46: end architecture netlist2;
```

- A netlist is a description of components used, and their connections
  - Synthesizing *is* creating a netlist using the available primitives for a (PL/ASIC) device.
  - The top level in larger designs is normally purely structural
- Component declaration pick up entities from «work» library
- The last compiled architecture are being used unless specified different
- Port mapping:
  - «Association» *can* be done by position
    - **Will lead to disasters when making changes.**
  - named association is less error prone. ex:
    `g1: Not1 port map (x=>a, z=>p);`

Component instantiation

- Instantiation of a declared component

```
Label: component <name> generic map(...) port map(...);
```

  - The entity of the component is not required for compilation (**for sim: yes**)
  - Preferred method in medium to large projects

- Direct instantiation

```
Label: entity <library>.<name>(<arch>) generic map(...) port map(...);
```

  - The instantiated entity must be compiled before use.

Example in next slide

Source: Ricardo Jasinski: Effective coding with VHDL Chapter 9.4

Instantiation example 1/2
Direct instantiation

```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;

entity adder is
  generic(N: positive := 5);
  port(
    a, b : in std_logic_vector(N-1 downto 0);
    sum  : out std_logic_vector(N downto 0)
  );
end entity adder;

architecture RTL of adder is
begin
  sum <= std_logic_vector(
    signed(a(a'left) & a) +
    signed(b(b'left) & b)
  );
end architecture RTL;
```

```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;

entity instantiate is
  port(
    addend, augend : in integer;
    sum  : out integer
  );
end entity instantiate;


architecture direct of instantiate is
  constant size : positive := 8;
  signal std_sum : std_logic_vector(size downto 0);
begin
  direct_inst: entity work.adder(rtl)
    generic map(N => size)
    port map(
      a => std_logic_vector(to_signed(addend,size)),
      b => std_logic_vector(to_signed(augend,size)),
      sum => std_sum
    );
  sum <= to_integer(signed(std_sum));
end architecture direct;
```

2/2 Using component

```vhdl
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;

entity instantiate is
  port(
    addend, augend : in integer;
    sum  : out integer
  );
end entity instantiate;
```

```vhdl
architecture structural of instantiate is
  constant size : positive := 8;
  component adder is
    generic(
      N: positive := 8
    );
    port(
      a, b : in std_logic_vector(N-1 downto 0);
      sum  : out std_logic_vector(N downto 0)
    );
  end component;

  signal std_sum : std_logic_vector(size downto 0);
begin
  --          "component" is optional
  direct_inst: component adder
    generic map(N => size)
    port map(
      a => std_logic_vector(to_signed(addend,size)),
      b => std_logic_vector(to_signed(augend,size)),
      sum => std_sum
    );
  sum <= to_integer(signed(std_sum));
end architecture structural;
```

Loops in VHDL

- Both simulation and synthesizable code

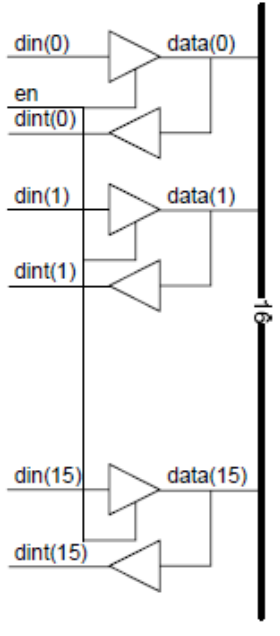
- Three types
  - Simple loop- until exit
  - While- loop condition is true
  - For loop
    - Counted
      - Numbers or elements/ ‘range
    - Loop parameter static
      - Can be increased using ‘**next**’
      - ‘**next when** <condition>’
- ‘**exit**’+(optional loop_label)
  - Can be used in all loops
  - Innermost loop is default
  - Nested loops: use label

```
--SIMPLE LOOP--
variable i: integer := 0;
...
loop
  statements;
  i := i + 1;
  exit when i = 10;
end loop;

--WHILE LOOP--
variable i: integer := 0;
...
while i < 10 loop
  statements;
  i := i + 1;
end loop;

--FOR LOOP--
for i in 1 to 10 loop
  statements;
end loop;

--FOR LOOP2–
type frukt_type is (eple, pære, banan);
...
frukt_loop: for f in frukt_type loop
  statements;
  when <condition1> next frukt_loop;
  when <condition2> exit frukt_loop;
end loop;
```

Structural design with `generate` statement

- `generate` - loop
  - can build multiple components .
  - requires indexable parameters in some connected signals
  - non-indexable signals will be connected to all instances
- Example: Bidirectional bus
- `generate`
  - can be used to conditionally build structures
  - `if` and `case` + `generate`
    - **Conditions must be resolved at compile-time**
      - only constants/generics, *no signals involved*
      - *This is not runtime-reconfiguration...*

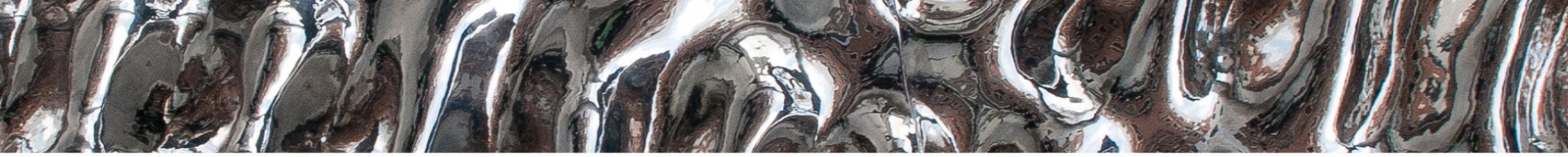
```
bidir_bus_inst: for i in 0 to 15 generate
  buft_inst: buft port map (data(i),en,din(i));
  ibuf_inst: ibuf port map (dint(i),data(i));
end generate;
```



Suggested reading

- D&H 7.1- 7.3 p129-153
  - (The testbench code in 7.2 is not curriculum).

- The instantiation code can be found on https://github.uio.no/in3160/lectures/tree/main/week03/instantiation



UiO : Department of Informatics
University of Oslo

IN 3160, IN4160

Combinational building blocks (and their VHDL)

Yngve Hafting

Messages

- Draw.io / Diagrams.net
  - Online tool https://app.diagrams.net
  - Download https://www.drawio.com
  - editor for diagrams
    - Will be available on this years exam!
    - ***replaces*** *hand drawing*

- Access to Lisp...
  - With card: 0500-2400
    - https://www.mn.uio.no/ifi/om/finn-fram/apningstider/
  - Do contact studieinfo@ifi.uio.no if this does not work

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 the basic structure and function of widely used combinational structures.
  - Multiplexers
  - Encoders
  - Decoders
  - Arbiter
  - Comparator
  - Shifters
  - ROM

Today: Buiding blocks

- About dataflow representations
- **Encoders vs Decoders**
- Decoder
- **Multiplexer**
- Encoders
- Arbiters
- **Shifters**
- Comparators
  - VHDL: dataflow vs RTL examples
- **ROM**
- **RAM**

Next lecture:

- Subroutines
- Packages & Libraries
- Sequential (clocked) statements.

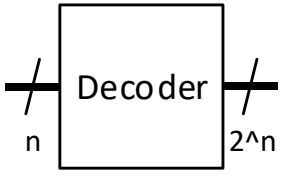
Goal:
Recognising & describing Building blocks

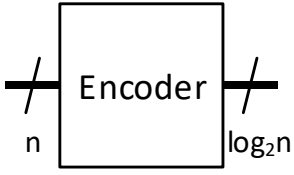
Getting to know more VHDL techniques...

Data flow representations

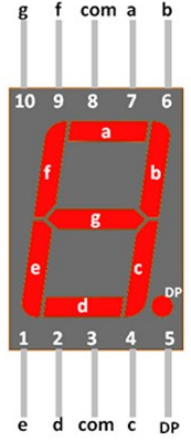
- Dataflow
  - Matches port/gate schematics
  - Use *when this is the only way* to achieve desired fuction
    - Tweaking (speed / area / power).
- *To show how buidling blocks are made*,
  this presentation uses low level representations
  - *Normally we want our code to be at a higher level*
    - *easier to read,*
    - *easier to maintain*
    - *synthesis tool decide primitives*

Encoders and decoders



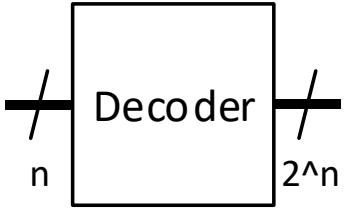


- ... convert signals from one type to another
- Encoder = inverse decoder
- Several types:
  - One hot decoder "$n \rightarrow 2^n$"
    - Typical use: in multiplexers, memory arrays (RAM, ROM)
  - Binary encoder "$n \rightarrow \log_2(n)$"
  - Priority encoder
    - Arbiter
- Conditional statements generally creates decoders as needed…
- De-/encoders not considered "building blocks"
  - Seven segment decoder
    - 4-5 bit Binary / BCD (binary coded decimal) to 8 bit... (Assignment 6)
  - Quadrature encoder
    - Converts rotational position/ speed to a two-bit pulse train (Assignment 8)



N to 2^N Decoder

- Ex: generic N to 2^N decoder
  - (binary to *one hot* converter)
- Demonstrates strict type check in VHDL
  - numeric_std is required for '**unsigned**' and '**integer**' conversions



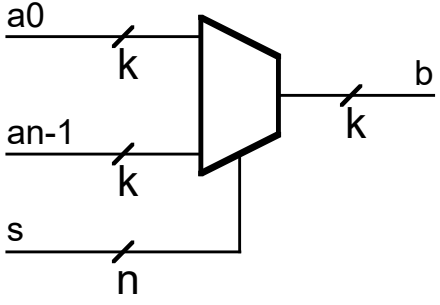

```
library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.numeric_std.all;

entity decoder is
  generic(n : positive := 4);
  port(
    a: in  std_logic_vector(n-1 downto 0);
    z: out std_logic_vector(2**n-1 downto 0)
  );
end entity decoder;

architecture rotate of decoder is
  constant one_vector : unsigned(z'range):= to_unsigned(1, z'high+1);
begin
  z <= std_logic_vector(one_vector sll to_integer(unsigned(a)));
end architecture rotate;

-- signal shift: integer;
-- shift <= to_integer(unsigned(a));
-- z <= std_logic_vector(one sll shift);
```

Total amount of bits

integer 1

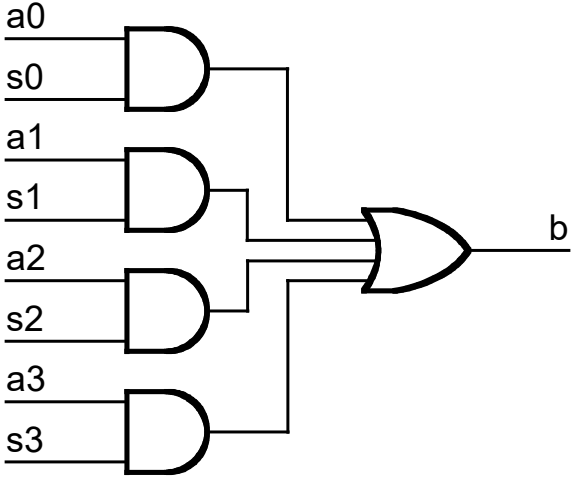
Multiplexer

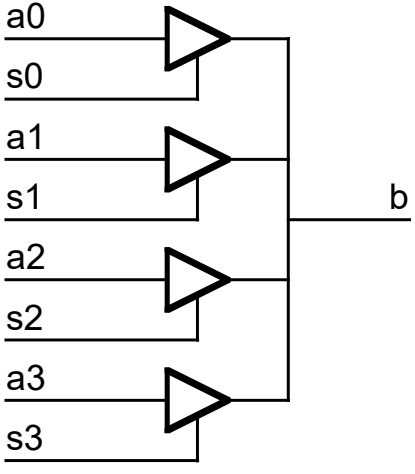
- k-bit Binary-Select Multiplexer:
  - n k-bit inputs
  - n-bit one-hot select signal s
  - Multiplexers are commonly used as *data selectors*



Selects one of n k-bit inputs
s must be one-hot
b = a[i] if s[i]=1

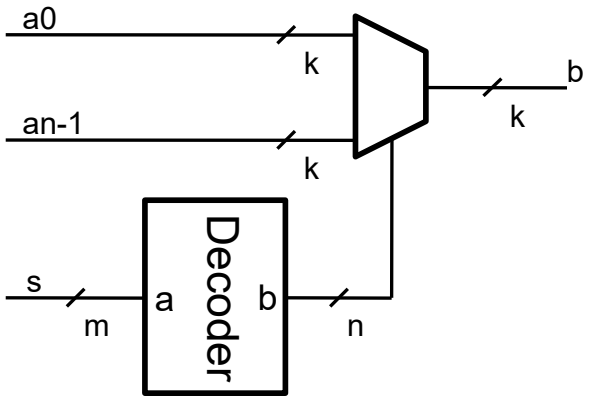
Multiplexer Implementation

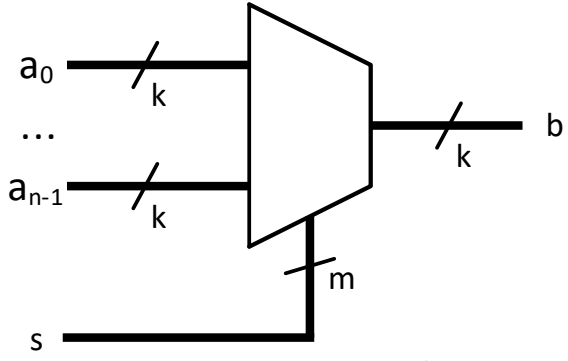




Kilde: W. J. Dally

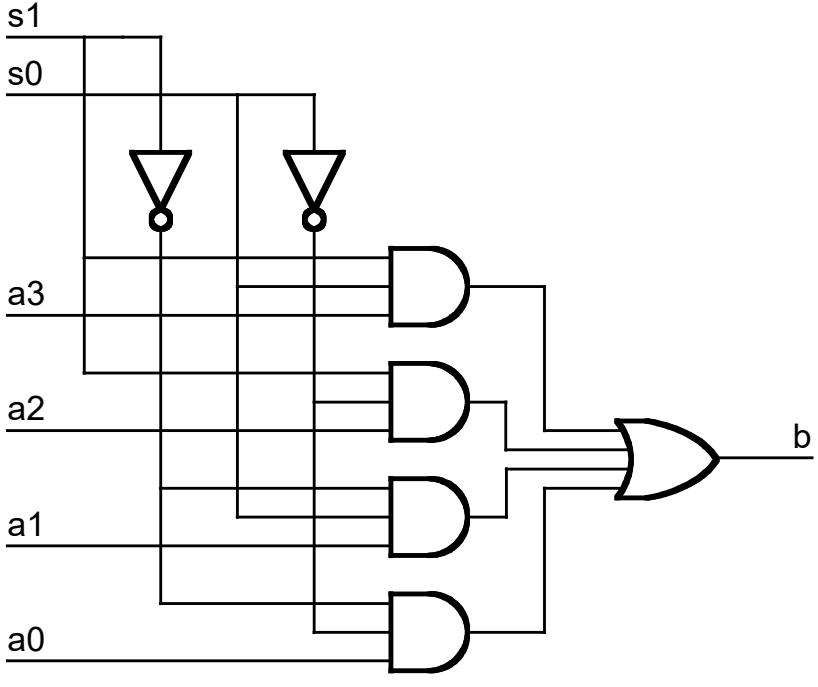
k-bit Binary-Select Multiplexer (Cont)





Normally, the decoder part is taken for granted

Ex: select 1 out of 4 bit (4 to 1 bit mux)
k=1, m=2, n=4



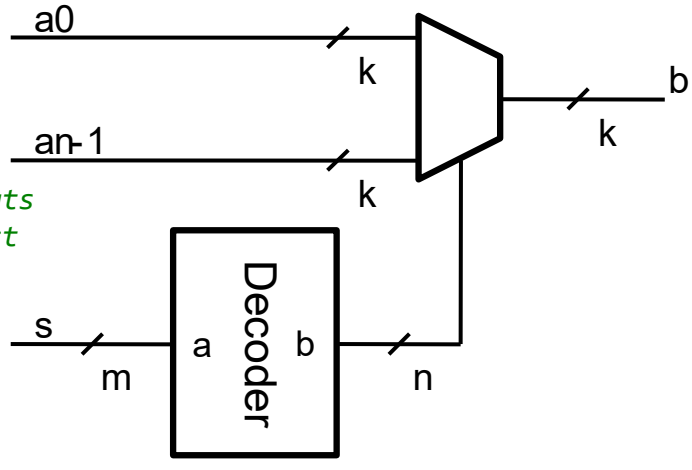
```
-- three input mux with one-hot select (arbitrary width)
library ieee;
use ieee.std_logic_1164.all;

entity Mux3a is
  generic( k : integer := 1 );
  port( a2, a1, a0 : in std_logic_vector( k-1 downto 0 ); -- inputs
        s : in std_logic_vector( 2 downto 0 ); -- one-hot select
        b : out std_logic_vector( k-1 downto 0 ) );
end Mux3a;

architecture case_impl of Mux3a is
begin
  process(all) begin
    case s is
      when "001" => b <= a0;
      when "010" => b <= a1;
      when "100" => b <= a2;
      when others => b <= (others => '-');
    end case;
  end process;
end case_impl;
```

```
architecture select_impl of Mux3a is
begin
  with s select b <=
    a0 when "001",
    a1 when "010",
    a2 when "100",
    (others => '-')when others;
end select_impl;
```

- Can this be implemented using '**select**' statement?
  - Single input vector
  - Single output vector…
  - QED…

Structural Implemention of k-bit Binary-Select Multiplexer

```
-- 3:1 multiplexer with binary select (arbitrary width)
library ieee;
use ieee.std_logic_1164.all;
use work.ch8.all; -- D&H library with generic decoder

entity Muxb3 is
  generic(k : integer:= 1);
  port(
    a2, a1, a0 : in std_logic_vector(k-1 downto 0); -- 3 k-bit inputs
    sb         : in std_logic_vector(1 downto 0);   -- binary select
    b          : out std_logic_vector(k-1 downto 0)
  );
end Muxb3;

architecture struct_impl of Muxb3 is
  signal s: std_logic_vector(2 downto 0);
begin
  -- decoder converts binary to one-hot
  d: Dec generic map(2,3) port map(sb,s);
  -- multiplexer selects input
  mx: Mux3 generic map(k) port map(a2,a1,a0,s,b);
end struct_impl;
```



Binary Encoder *(output is the number of most significant bit set)*

Don't cares vs ordered priority:

```
architecture dont_care of priority is
begin
  with a select y <=
    "00" when "0001",
    "01" when "001-",
    "10" when "01--",
    "11" when "1---",
    "00" when others;

  with a select valid <=
    '1' when "1---" | "01--" | "001-" | "0001",
    '0' when others;
end architecture dont_care;
```

```
architecture ordered of priority is
begin
  y <=
    "11" when a(3) else
    "10" when a(2) else
    "01" when a(1) else
    "00" when a(0) else
    "00";

  valid <= or a;

  --valid <='0' when a="0000" else '1';
end architecture ordered;
```

A3	A2	A1	A0	Y1	Y0	Valid
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	-	0	1	1
0	1	-	-	1	0	1
1	-	-	-	1	1	1

Selected and when-else statement

- *Two checks on the same signal*
  - *= Small maintainability issue compared to* ***case***

- Don't cares in dataflow representations-
  - *may* help the synthesizer pick glitch free implementations (ref. karnaugh diagrams)

Don't cares vs sequential ordered priority

```
architecture mcase of priority is
begin
  process(a) is
  begin
    --default values
    y <= "00";
    valid <= '1';
    case? a is
      when "0001" => y <= "00";
      when "001-" => y <= "01";
      when "01--" => y <= "10";
      when "1---" => y <= "11";
      when others => valid <= '0';
    end case?;
  end process;
end architecture mcase;
```

Case/ matching case:

- Test on input (a) is written once

A3	A2	A1	A0	Y1	Y0	Valid
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	-	0	1	1
0	1	-	-	1	0	1
1	-	-	-	1	1	1

```
architecture default_if of priority is
begin
  process(a) is
  begin
    --default values
    y <= "00";
    valid <= '1';
    if    a(3) then y <= "11";
    elsif a(2) then y <= "10";
    elsif a(1) then y <= "01";
    elsif a(0) then y <= "00";
    else  valid <= '0';
    end if;
  end process;
end architecture default_if;
```

Sequential priority (if):

- Easy to forget specifying all outputs for all input options
- Readability suffers as complexity grows

```
architecture pos_pri of priority is
begin
  process(a) is
  begin
    --readability suffers- don't do
    y <= "00";
    if a(1) then y <= "01"; end if;
    if a(2) then y <= "10"; end if;
    if a(3) then y <= "11"; end if;
  end process;
  valid <= or a;
end architecture pos_pri;
```

Bad idea, but

Case- or selected- statement is preferred

```
architecture non_default of priority is
begin
  process(a) is
  begin
    --no default values
    if    a(3) then
      y <= "11";
      valid <= '1';
    elsif a(2) then
      y <= "10";
      valid <= '1';
    elsif a(1) then
      y <= "01";
      valid <= '1';
    elsif a(0) then
      y <= "00";
      valid <= '1';
    else
      y <= "00";
      valid <= '0';
    end if;
  end process;
end architecture non_default;
```

Positional priority

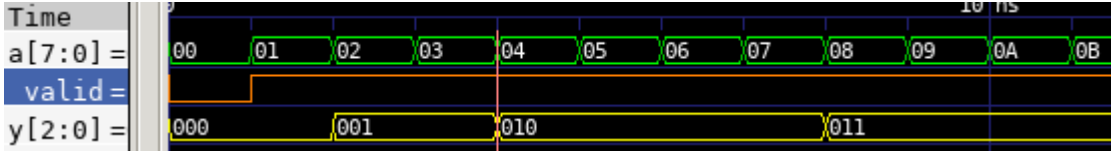
- Not normally used in processes
  - Difficult to read
  - Mixing with sequential priority makes it worse
- *Can make sense when using loops..*
  - *=> Next page*

Generic priority encoder

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

entity priority is
  generic ( n : positive := 2);
  port(
    a : in  std_logic_vector(2**n-1 downto 0);
    y : out std_logic_vector(n-1 downto 0);
    valid : out std_logic
  );
end entity priority;

architecture range_iterative of priority is
begin
  process(a) is
  begin
    -- default values
    valid <= '0';
    y <= (others => '0');

    -- iterate i from 2**n - 1 down to 0
    -- set y to i when the highest bit is true
    for i in a'range loop
      if a(i) = '1' then
        y <= std_logic_vector(to_unsigned(i,n));
        valid <= '1';
        exit;     -- without exit (loop exit),
      end if;     -- lower bits would be prioritized
    end loop;
  end process;
end architecture range_iterative;
```

These architectures are equivalent = Does the same

- Note how the priority changes due to the **exit** statement in the for-loop.
- Changing the direction of a'range would also change priority…
  - (ie `(0 to 2**n-1)`)




```
architecture reverse_range_iterative of priority is
begin
  process(a) is
  begin
    -- default values
    valid <= '0';
    y <= (others => '0');

    -- iterate from 0 to  2**n-1
    for i in a'reverse_range loop
      if a(i) = '1' then
        y <= std_logic_vector(to_unsigned(i,n));
        valid <= '1';
      end if;     -- no exit, means the last iteration (i=2**n-1)
    end loop;     -- will be prioritized for setting y
  end process;
end architecture reverse_range_iterative;
```

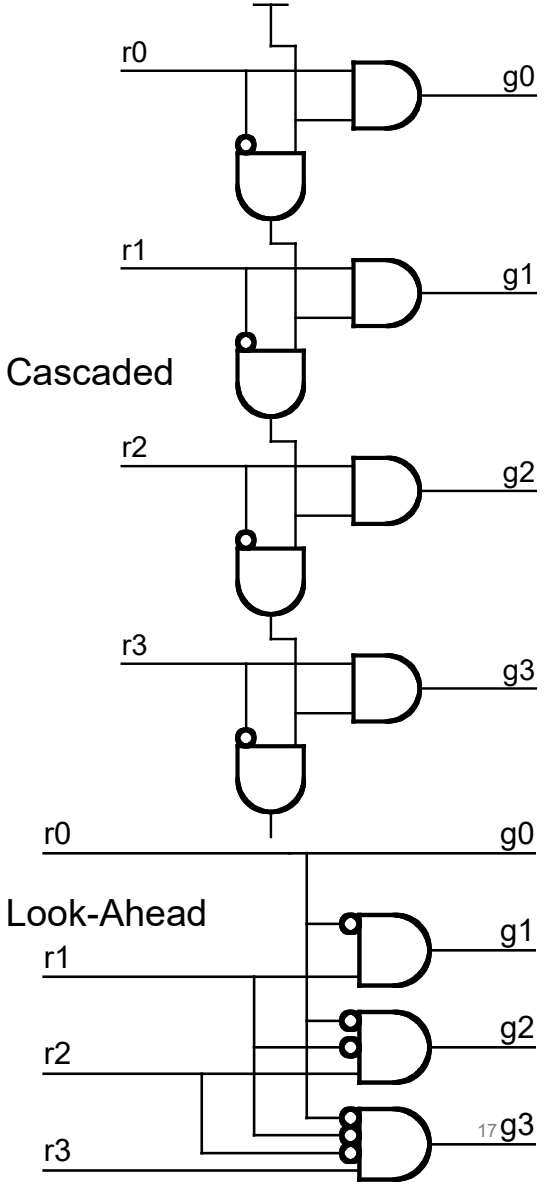
NOTE: All iterations create HW

how we iterate determine priority

There is no "conditional execution"

Arbiter

- Arbiters is used to sort requests for resources
  - interrupt handling in a cpu or microprocessor
  - Finds the least (or most significant) one-bit
  - cascaded vs look-ahead principle
  - VHDL = priority encoder (previous page).
  - Normally we let synthesis tool decide
    - FPGA => mostly LUT based
    - Structural code may bind a solution
      - Is it a critical feature?
      - Does not synthesis provide desired result?



Priority encoder test bench

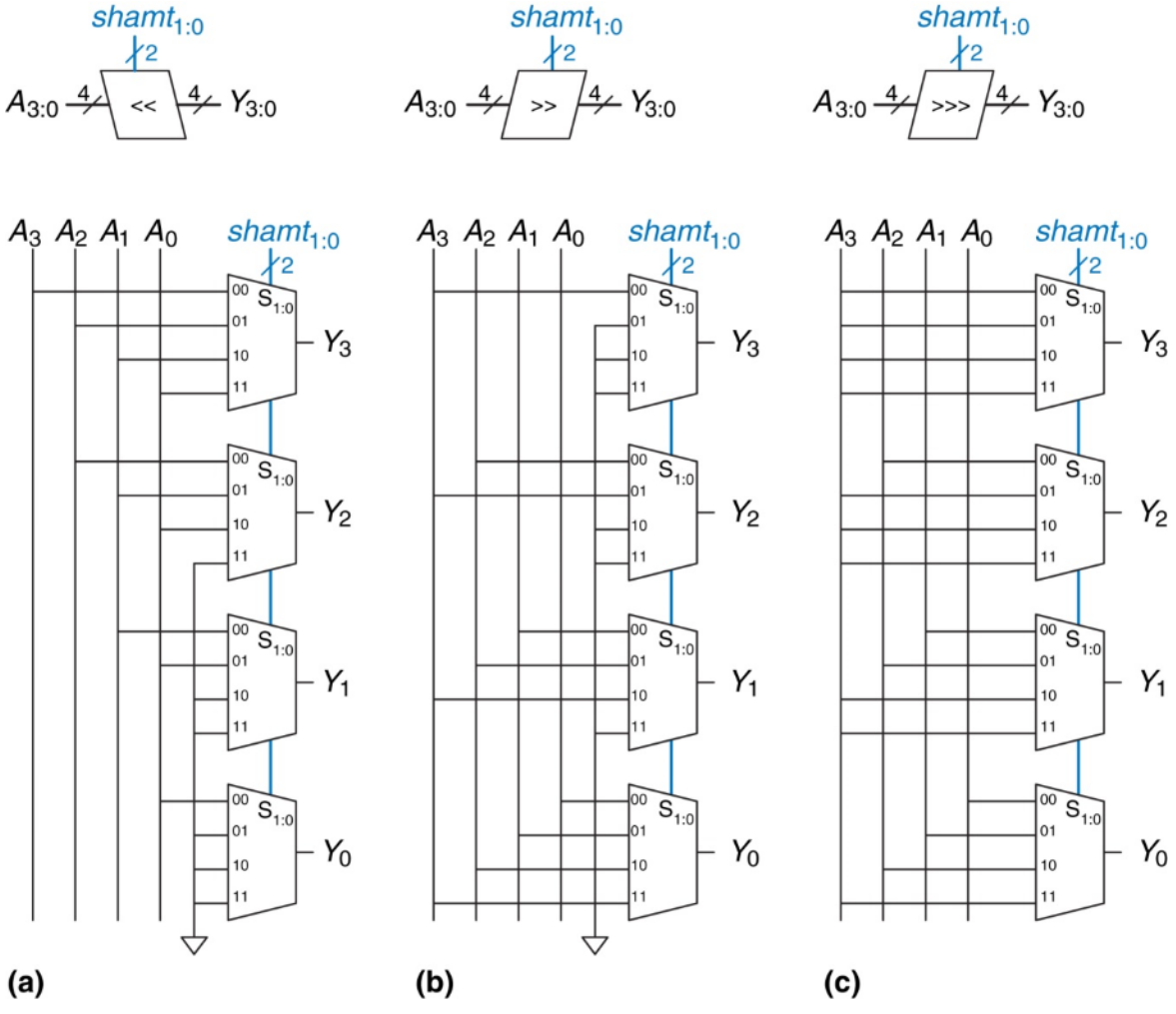
- The example makes stimuli to a combinational function independent of number of bits
- The attribute x'high gives the highest bit number to the vector x and x'low the lowest bit number
- Python equivalent tb code:

```
STIMULI :
process
  variable ain : integer := 0;
begin
  loop
    for ain in 0 to 2**(a'high-a'low+1)-1 loop
      a <= std_logic_vector
           (TO_UNSIGNED(ain, a'high-a'low+1));
      wait for 50 ns;
    end loop;
  end loop;
end process;
```

```
async def stimuli_generator(dut):
    ''' Generates all data for this tesbench'''
    for i in range( 2**len(dut.a)):
        dut.a.value = i
        await Timer(1, units= 'ns')
```

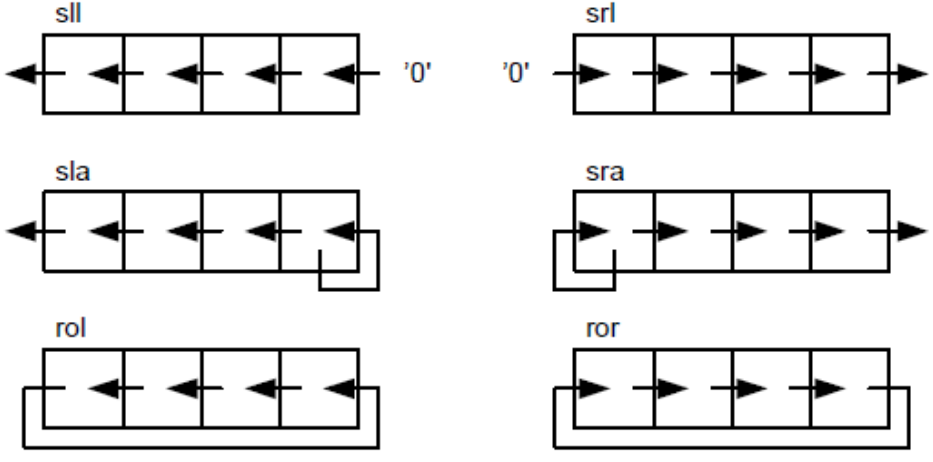
Shifters

- Ex, 4 bit :
  - a) SLL
  - b) SRL
  - c) SRA



Shift operators in VHDL

- The shift operators are defined for `bit_vector` (originally)
  - and `unsigned` and `signed` in `numeric_std`
- If you are defining shift operators for other types, you have to make so called "overload"-operators
- By overload we mean that there are an already existing operator with the same name, but is written for another data type




```
-- simple shift left operation in VHDL
variable n : positive := 5;
...
a(31 downto n) <= a(31-n downto 0); -- a'high is 31,
a(n-1 downto 0) <= (others => '0'); -- a'low is 0.
```

```
-- using sll (requires numeric_std library)
a <= std_logic_vector( to_unsigned(a) sll(n) );
```

Shift operators

- The standard libraries does not define shift operators for **std_logic_vector**
- The standard synthesis library **numeric_std** defines two data types which are sub types of **std_logic**:
  - **unsigned**
  - **signed**
  - For these two it exists shift operators (overload)

- Use type casting to go between **std_logic_vector** and **signed** / **unsigned**

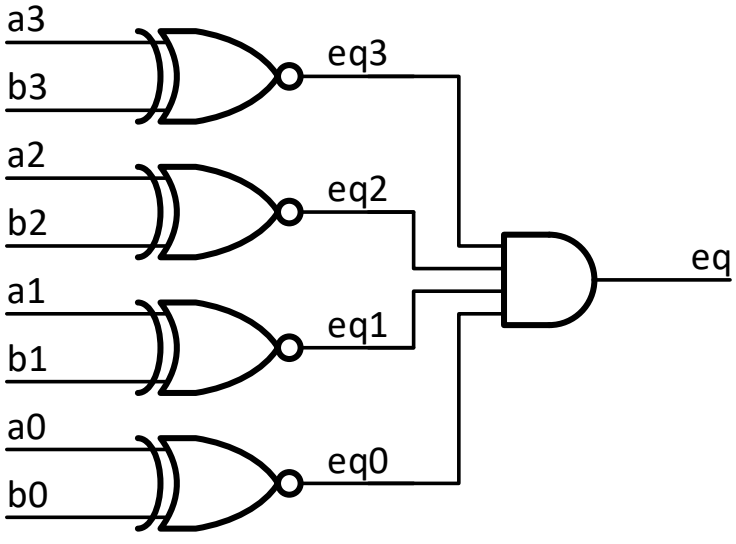
```
-- a is std_logic_vector.
a <= std_logic_vector( to_unsigned(a) sll(n) );
```

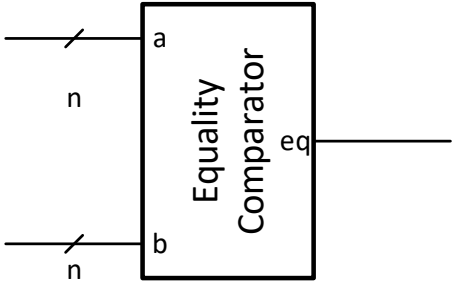
Comparators

- Equality ‘=‘
- Magnitude ‘<‘, ‘>’

Equality Comparator

Is true when both input vectors are equal



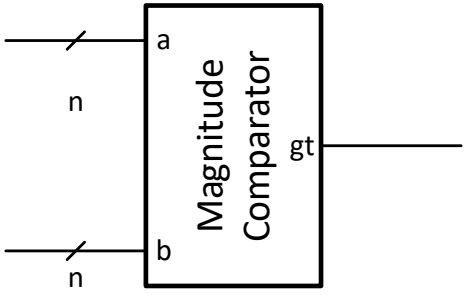



```
eq <= '1' when (a = b) else '0';

-- Dataflow: eq <= and (a xnor b);
```

```
-- high level comparator usage, IF (inside process)
if (a = b) then
  p <= q;
else
  p <= (others => '0');
end if;

-- high level comparator usage, WHEN … ELSE
p <= q when (a = b) else (others => '0');
```

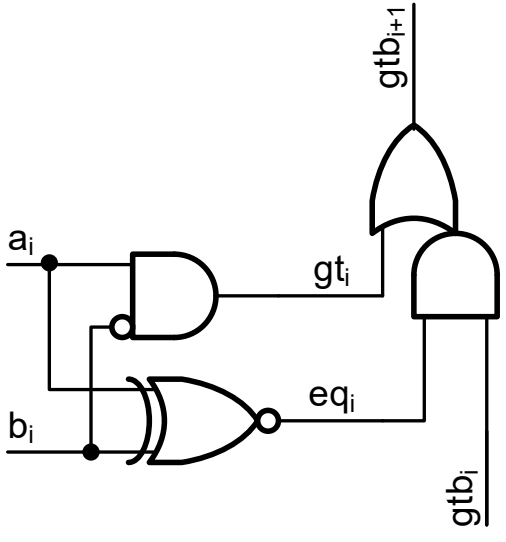
Magnitude Comparator

- 'if (a>b) then …'
  will infer what you need most of the time

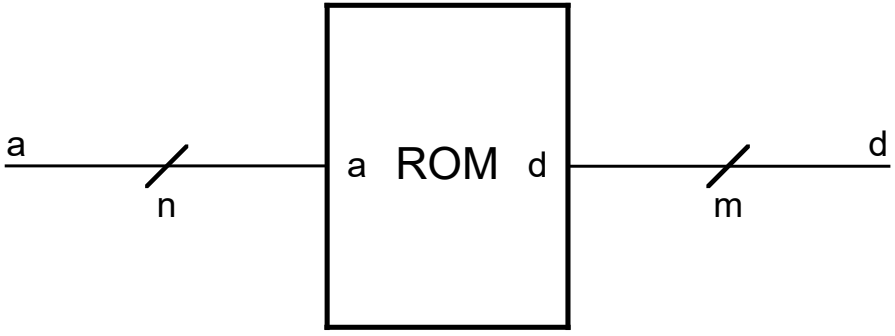


- Dataflow example.

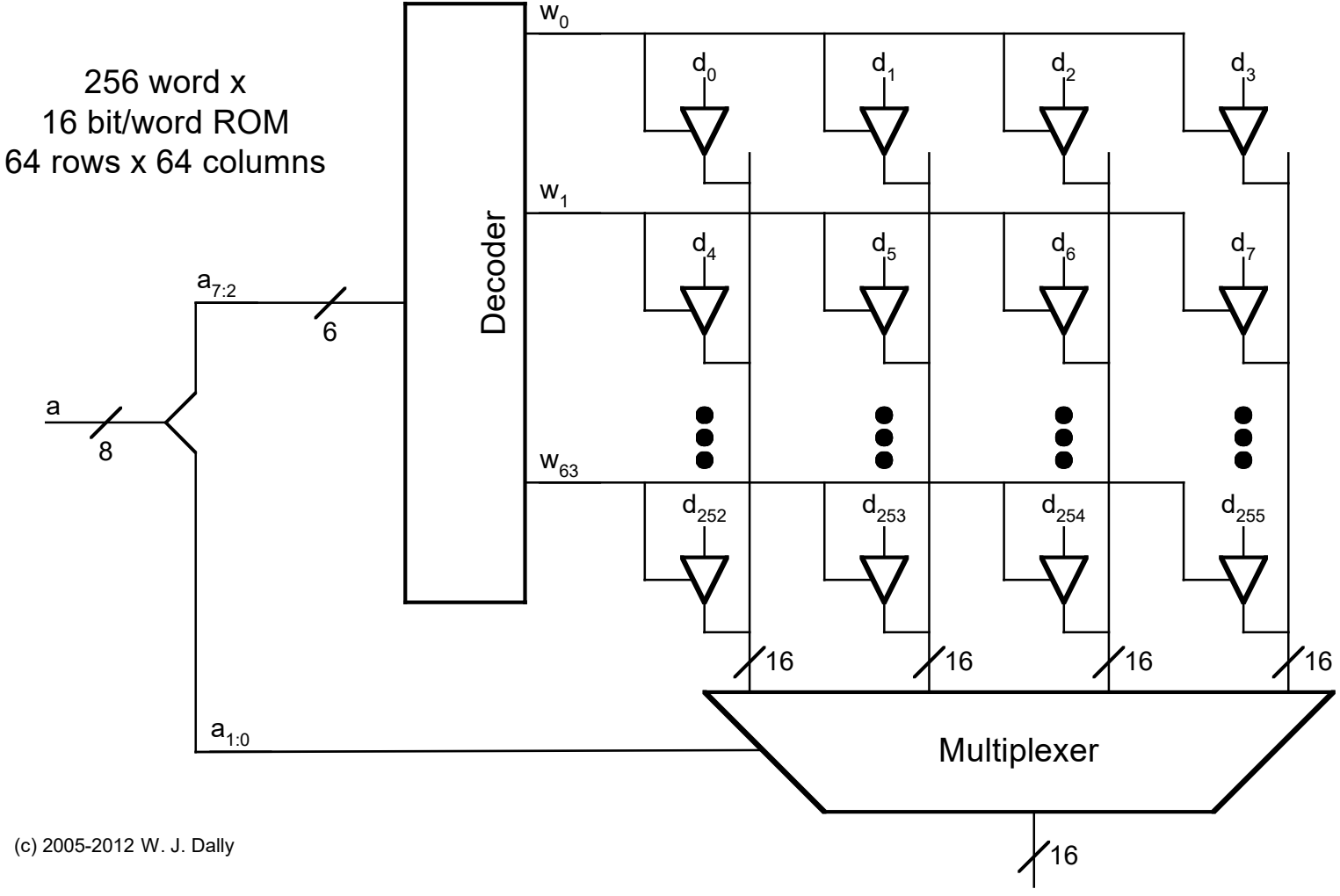
```
library ieee;
use ieee.std_logic_1164.all;

entity MagComp is
  generic( k: integer := 8 );
  port( a, b: in std_logic_vector(k-1 downto 0);
        gt: out std_logic );
end MagComp;

architecture impl of MagComp is
  signal eqi, gti : std_logic_vector(k-1 downto 0);
  signal gtb: std_logic_vector(k downto 0);
begin
  eqi <= a xnor b;
  gti <= a and not b;
  gtb <= (gti or (eqi and gtb(k-1 downto 0))) & '0';
  gt <= gtb(k);
end impl;
```



Read-only memory (ROM)



2-D array implementation



ROM using VHDL

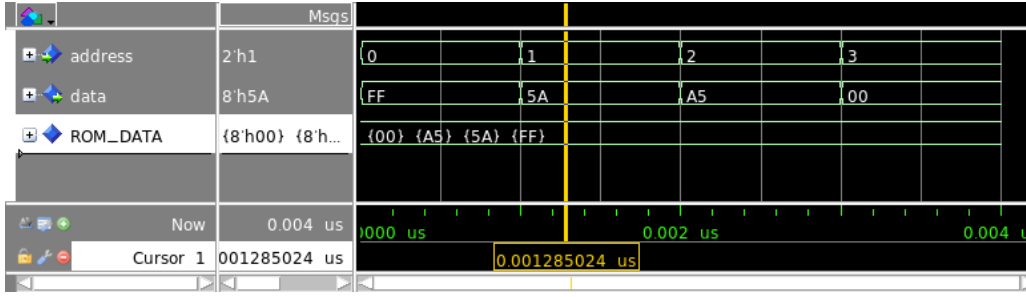
- ROM can be implemented using
  - `select`'ed statement
  - `case`
    - D&H demonstrates this.
  - `constant`'s
    - Example next slide

- File IO *can* be used to store ROM values.
  - Tools may be picky about implementations.
  - *We will look into that later.*

Example: ROM

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

entity ROM is
  generic(
    data_width: natural := 8;
    addr_width: natural := 2);
  port(
    address:  in std_logic_vector(addr_width-1 downto 0);
    data: out std_logic_vector(data_width-1 downto 0));
end entity;

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

  constant ROM_DATA: memory_array := (
    8x"00", -- address 3 (from 'left to 'right)
    8x"A5", -- address 2
    8x"5A", -- address 1
    8x"FF"  -- address 0
    );

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

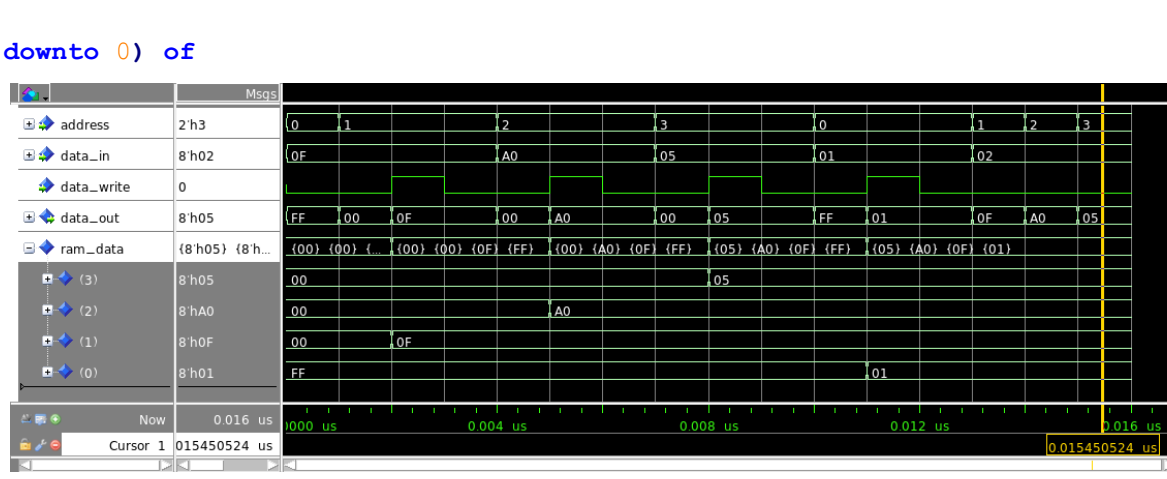
- 4 byte ROM example
  - 8 bit data
  - 2 bit address
- We can define array types in VHDL
- Constants are set using :=
- Array data is listed in the sequence given by the type (array) definition
  - Here: (2**addr_width-1 downto 0) => 3, 2, 1, 0
- Indexing requires conversion to integer



RAM using VHDL

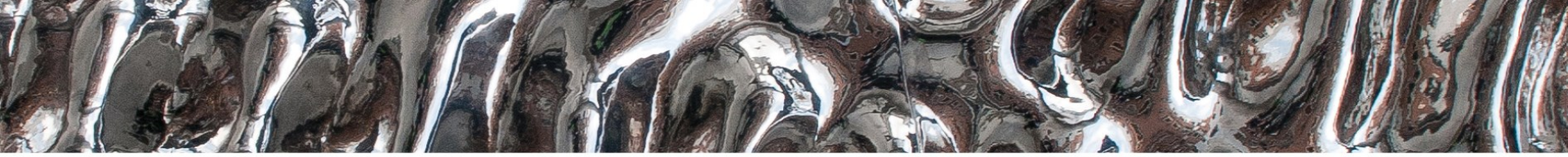
```
library IEEE;
  use IEEE.STD_LOGIC_1164.all;
  use IEEE.numeric_std.all;

entity RAM is
  generic(
    data_width: natural := 8;
    addr_width: natural := 2
    );
  port(
    address:  in std_logic_vector(addr_width-1 downto 0);
    data_in:  in std_logic_vector(data_width-1 downto 0);
    data_write: in std_logic;
    data_out: out std_logic_vector(data_width-1 downto 0)
    );
end entity RAM;

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

  signal ram_data: memory_array :=
    (8x"00", 8x"00", 8x"00", 8x"FF");
begin
  data_out <=
    ram_data(to_integer(unsigned(address)));

  ram_data(to_integer(unsigned(address))) <=
    data_in when data_write; -- else latched
end architecture synth;
```

- 4 Byte RAM example
- Mostly like the ROM example
  - Added write and data_in
  - Data is a **`signal`**, not **`constant`**
- Signals *may* have default values in synthesis (RAM based FPGAs)
- Writing is latched
  - *not combinational*



Suggested reading

- D&H 8.1- 8.9 p157-192
- *(8.10 PLA -> Architecture)*



UiO : Department of Informatics
University of Oslo

IN3160, IN4160

1: Subroutines, packages and libraries

2: Clocked statements

Messages

- Remember to **log out** after using the lab machines
  - Reduces the need for reboot (USB port locked to user)

- Peer review oblig 3
  - In time = automatically assigned
  - Late = manual assignment Monday
  - On sick leave = by appointment

  - Be polite!
    - Reviewing others work is frequently used in industry.

Goal

- Learn how to create subroutines using VHDL
- Learn good practice for writing subroutines
- Learn which packages are most used in VHDL
- Learn how to use and create libraries and packages in VHDL

Overview

- Subroutine types
  - Functions
  - Procedures
- Functions and operators
- Procedures
- Overloading in VHDL
- Libraries
  - Package/package body
- Standard libraries
- **Clocked statements**

Next: Verification & file IO

Why Subroutines

- *avoid duplicating code*
  - Make the code more readable
  - Reduce code complexity
  - Make the code easier to maintain
  - Make code easier to test or verify

VHDL Subroutine types and practice:

- Two types:
  - Functions – *returns* one *value*
    - = *CL*
  - Procedures – *a group of statements*

- General good practice:
  - **use functions!**
  - Limit the use of procedures to structural code or testbenches
    - Consider functions, entites or processes before using procedures

Functions-

- take one or more parameters
  - Parameters in functions
    - Can not be changed/manipulated
      - always mode “in”
    - (only) *constant*, *signal* or *file*
      - *constant is default*
  - Parameters are separated by ‘;’
    (a: bit; b: my_type;…)

- return only a single *value*
  - The value can be of any *type*
    - *Including vectors and custom types*

parameter(s)

single return value

```
function sum_function(vect: integer_vector) return integer is
  variable sum: integer := 0;
begin
  for i in vect'range loop
    sum := sum + vect(i);
  end loop;
  return sum;
end;
```

```
b <= std_logic_vector(to_signed(
  sum_function((
    to_integer(signed(a2)),
    to_integer(signed(a1)),
    to_integer(signed(a0)) )),
  b'high - b'low + 1) );
```

The «extra» paranthesis is needed to make one vector out of the three integers. Without, they will be interpreted as three separate parameters of wrong type

- *pure* functions use only their input parameters => CL
- *Impure* functions make use of data visible where they are declared (as parameters)
  - *Ex: File IO (next lecture)*
- Cannot have wait- statements. (single event / completes within a single delta delay)
- Cannot have internal signals (no storage)

Function usage example:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity subprogram is
  generic( k: positive := 4 );
  port(
    a2, a1, a0: in std_logic_vector(k-1 downto 0);
    b : out std_logic_vector(k-1 downto 0) );
end subprogram;

architecture example of subprogram is
  function sum_function(vect: integer_vector)
    return integer is
    variable sum: integer := 0;
  begin
    for i in vect'range loop
      sum := sum + vect(i);
    end loop;
    return sum;
  end;
```

```
begin
  local: process(all) is
    variable v: integer_vector(2 downto 0);
    variable sum: integer;
    variable s: signed(b'high-b'low downto b'low);
  begin
    v:=(
      to_integer(signed(a2)),
      to_integer(signed(a1)),
      to_integer(signed(a0)) );

    sum := sum_function(v);

    s := to_signed(sum, b'high - b'low + 1);
    b <= std_logic_vector(s);
  end process local;

end architecture example;
```

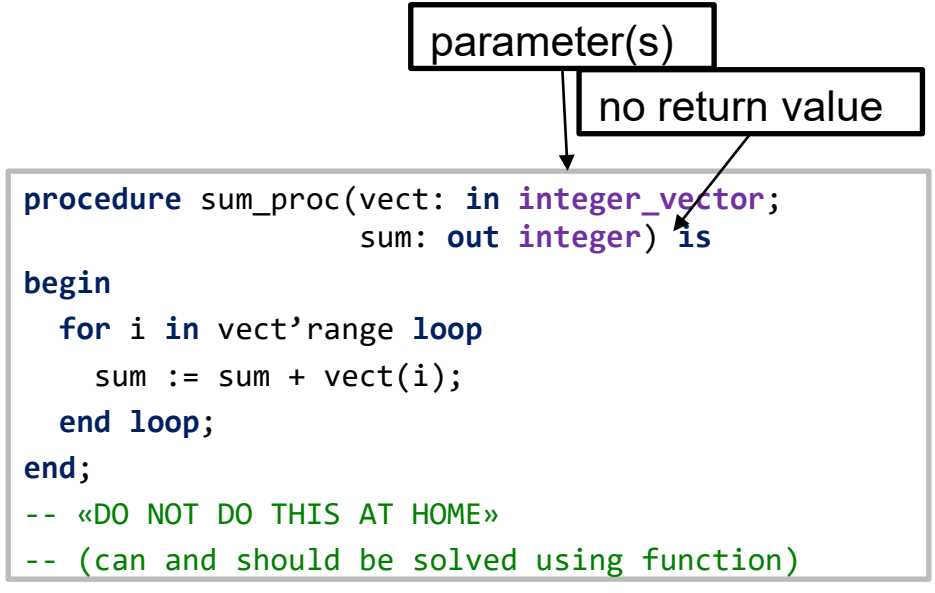
More Functions…

- Can be used for both synthesis and simulation
- Can (also) be *overloaded* (two or more functions having same name)
  - different parameters and or return type
- Are declared in the declarative region of
  - architectures
  - processes
  - packages (declaration and body – example later)
- Are frequently used for
  - Computation
  - Type converting
- Packages in libraries we use typically define functions
  - IEEE (library)
    - std_logic_1164 (package)
    - numeric_std
    - …
  - *We use these all the time*…

```
architecture func_arch of functest is

-- declarations
function bool2bit(a: boolean) return bit is
begin
  if a then
    return '1';
  else
    return '0';
  end if;
end bool2bit;

-- statements
begin
  …
end func_arch;
```

Procedures…

- do not have a return value
- can have
  - **in** and **out** *parameters*
    - in is default
    - out parameters (must be set)
  - **wait** statements
  - signals
  - file access
- cannot be used in a statement
  - *Only standalone «calls»*
- Are typically used in test benches
  - Reading test vectors from file
  - Applying test vectors
  - Writing test results to file



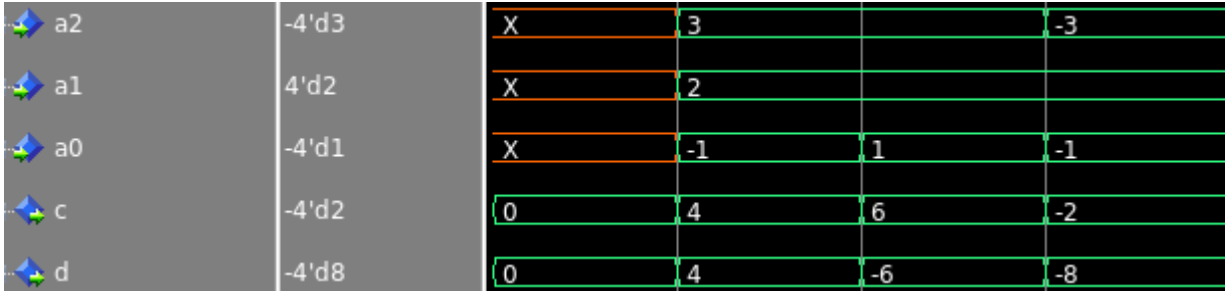
- *Can manipulate both **out**-parameters and other **signal**s* declared in the same (underlying) region...

Example- when not to use procedure:

```
process(all) is
     variable v: integer_vector(2 downto 0);
     variable sum: integer;
     variable s: signed(b'high-b'low downto b'low);
  begin
    v:=(
      to_integer(signed(a2)),
      to_integer(signed(a1)),
      to_integer(signed(a0)) );

    sum := sum_function(v);

    s := to_signed(sum, b'high - b'low + 1);
    c <= std_logic_vector(s);
  end process local;
```

Only difference apart from declarations (previous slides)

```
process(all) is
    variable v: integer_vector(2 downto 0);
    variable sum: integer;
    variable s: signed(b'high-b'low downto b'low);
  begin
     v:=(
      to_integer(signed(a2)),
      to_integer(signed(a1)),
      to_integer(signed(a0)) );

    sum_proc(v,sum);

    s := to_signed(sum, b'high - b'low + 1);
    d <= std_logic_vector(s);
  end process;
```



- Why aren't c and d equal?

- **the procedure is not CL.**
  - *sum accumulates*
  - *process variable -> single instantiation*
- In HW, d would be unstable due to this feedback loop, since the process is not clocked.
- *-6 and -8 is the result of the 4 digit two-complement representation.*

Functions vs Procedures

Functions	Procedures
Returns a value (*can be vector*) of any type	A collection of statements (Sets signals)
Can only use variables, no signals.	Can contain both signals and variables that will be hidden from the outside. May use signals from the underlying structure.
Cannot replace procedures fully	*Can* replace functions (DON'T DO THAT!)
Much used in conversions (from bit to STD_LOGIC, from some_type to my_type, etc).	Much used for repetitive tasks- particularly in test benches.
Typically found in libraries and packages	*Mostly used for simulation/ test benches.*
Always "instant" (CL), never time based	Can use "wait" and timing information.
Can be used in statements… `a <= my_func(…);`	Can only be used standalone… `my_procedure(..);`

Neither can store internal values between calls.

Parameters for subprograms

Parameters or «interface objects» have up to five parts

1. Class : **`constant`** (default), **`variable`**, **`signal`**, **`file`**
2. Identifier: the name you decide must be defined
3. Mode: **`in`** (default) or **`out`**
4. Type: **`std_logic`**, **`integer`**, **`bit`**, **`text`**, … must be defined
5. Default value := optional

Ex:

```
procedure apply_vectors(
  file vector_input : text;
  addend : integer := 42;
  signal valid : out boolean;
  file vector_output : text);
```

Good practice

- When considering to create a subprogram:
  - Is it possible to do this using a function?
    - Yes: **use function**
    - No:.. Is it for creating HW?
      - If yes: consider a process, or a separate entity + architecture
  - Is it for simulation only, and a function will not do:
    - Use a procedure

- Subprograms generally should have a single purpose.
  - Try see if the purpose can be said in one sentence without use of “and” or “or”…

Good practice

- Use functions when you can
  - Limitations in functions makes it easier to achieve well structured code
    - readable
    - maintainable
    - short
- Limit procedures to structural code (or testbenches)
  - It is easy to create messy code using procedures since they allow
    - multiple in and out parameters
    - to use signals and create storage elements

Packages

- In a package declarative region you can add:
  - Component declarations
  - Data type definitions
  - Constants
  - Subprogram declarations
    - Functions
    - Procedures
- The declarative region is publicly visible
  - similar to header files in C
- Package body-
  - declarations is not publicly visible
  - typically contains content of-
    - subprograms
    - components

```
package my_pkg is
  -- publicly visible declarations
  type imb_vec is record
    re: bit_vector;
    im: bit_vector;
  end record;

  constant IMB_VEC1: imb_vec := (re => "010", im => "001");
  function bool2bit(a: boolean) return bit;
  …
end;

Package body my_pkg is
  -- non visible, internal declarations
  function bool2bit(a: boolean) return bit is
  begin
    …
  end bool2bit;
  …
end my_pkg;
```

Packages

Save and compile your package in the work folder

To use package contents, include these two lines:

```
1 library work;
2 use work.my_package.all;
```

```
1  -- Package Declaration Section
2  package my_package is
3
4    constant c_PIXELS : integer := 65536;
5
6    type t_my_rec is record
7      full: std_logic;
8      empty: std_logic;
9    end record t_my_rec;
10
11   component my_component is
12     port (i_data  : in  std_logic; o_res : out std_logic);
13   end component my_component;
14
15   function Bit_OR (i_vec : in std_logic_vector(3 downto 0))
16      return std_logic;
17
18 end package my_package;
19
20 -- Package Body Section
21 package body my_package is
22
23   function Bit_OR (i_vec : in std_logic_vector(3 downto 0))
24     return std_logic is
25   begin
26     return (i_vec (0) or i_vec (1) or i_vec (2) or i_vec (3));
27   end;
28
29 end package body my_package;
```

Typical use of packages

- Typically packages is organized such that it contains
  - custom or abstract data types
    - Ex: I have a project that will incorporate calendar data
      - => lets make a package for all types used
  - functions that work on these abstract data types.

- Create packages when you have
  - function(s) that may be used by more than one design unit.
  - Types that may be used in several design units
  - Components that may be used in several designs
  - Simulation -models, -procedures and -functions that can be re-used

Use of functions library

- A “package/body” pair can be compiled to work or to another library.
  - This needs to have a logic name. Here: mylib
- The logic name is given in the current tool and is reflected in the directory structure of the tool

```
use work.my_func.all;
-- eller
-- use.work.my_func.bl2bit;
-- dersom bare bl2bit skal gjøres synlig

entity ......
architecture ...
signal a: BOOLEAN;
signal b: bit;
begin
  b <= bl2bit(a);
end architecture ..;
```

```
library mylib;
use my_lib.my_func.all;
-- eller
-- use.my_lib.my_func.bl2bit;
-- dersom bare bl2bit skal gjøres synlig
```

Operators

- Operators are defined in the same way as functions, but by “<operator name>”
- Operators are being used differently from functions
- You can create overloaded operators (ie ‘+’ for my_type),
  - *but not create new*

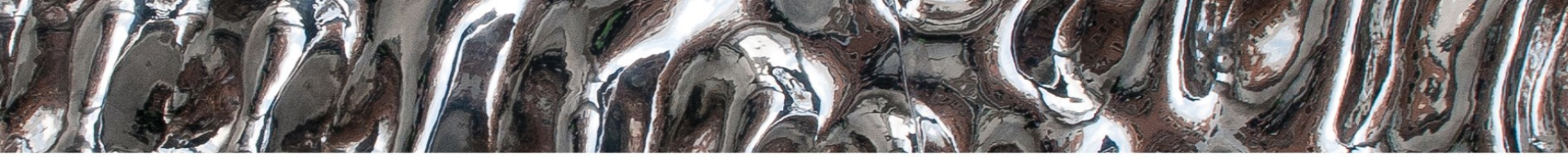
```
-- package declaration (overload)
 function "+" (a,b :std_logic_vector) return std_logic_vector;

 ...
 -- usage
 sum <= a + b;


 -- package declaration (non overload)
 function add (a,b :std_logic_vector) return std_logic_vector;

 ...
 -- usage
 sum <= add(a, b);
```

Overloading

- **Overloading** means defining the same operator-, function- or procedure-name for different data types or a mix of data types.
- Overloaded subprograms (operators, functions and procedures) may have different number of parameters
- Synthesis tools separates the usage of overloaded subprograms by comparing actual parameters (those in use) with formal parameters (in the subprogram declaration)

Overloading

- There are a lot of standard libraries with overload operators, functions and procedures in IEEE 1164 and IEEE 1076.3
  - IEEE *1164*
    - ***Package std_logic_1164***
    - Synopsis libraries (compiled to IEEE, but not standard)
      - Package std_logic_unsigned
      - Package std_logic_signed
      - Package std_logic_arith
      - Package std_logic_textio
      - *Don't use these in this course.*
        - » *Std libraries covers the usage and there are some differences.*
  - IEEE *1076.3*
    - ***Package numeric_std***
    - Use the package `IEEE.numeric_std` for integer arithmetics with the use of the data types **signed** and **unsigned**



UiO : Department of Informatics
University of Oslo

IN 3160, IN4160

Clocked processes and statements

Yngve Hafting

UiO : Department of Informatics
University of Oslo

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.

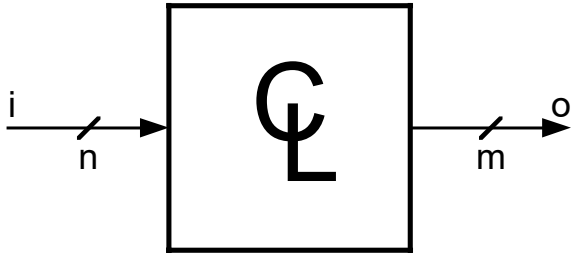
After completion of the course you will:

- 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.

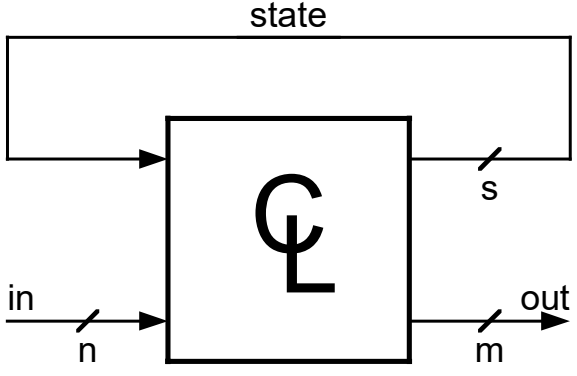
Goals for this lesson:

- Know different approaches to achieve clocked logic in VHDL
  - Why they exists
  - Benefits and pitfalls
  - ...

Sequential logic has state



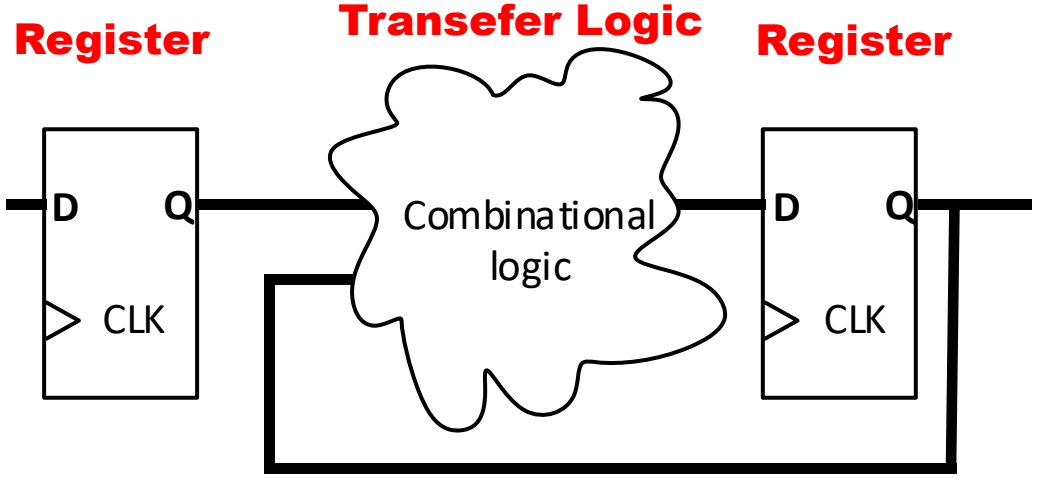
Combinational Logic



Sequential Logic

Do not use feedback into CL without using flipflops!

Sequential design = CL + FFs



=> RTL design

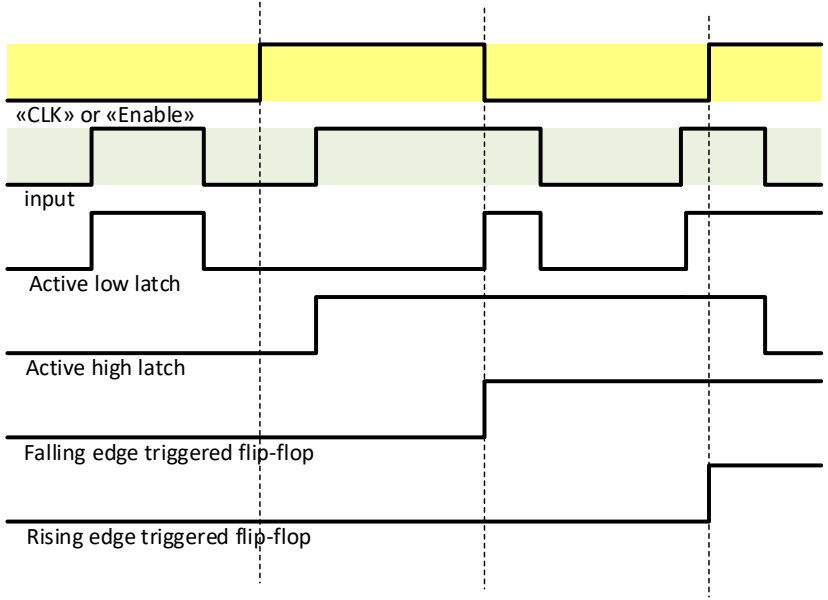
- Good practice:
  - Keep CL and register storage separate
  - *Know* when you infer storage elements!

- Sequential designs *are* state machines
  - *Sometimes* they have other names
    - *Counters*
    - *Shift Registers*
    - *LFSR – Linear Feedback shift Registers*
    - ...

Latch vs Flip-flop

- The functions **rising_edge** and **falling_edge** gives a true (0->1 or 1->0) edge detection
  - `if CLK'event and CLK = '1' then` reacts on all transitions to '1', for example U->1 *(simulation)*

- NB! *An incomplete* conditional statement will be synthesized to a latch* (implied memory)
  - *complete defines all outputs for all conditions of the input variable(s).

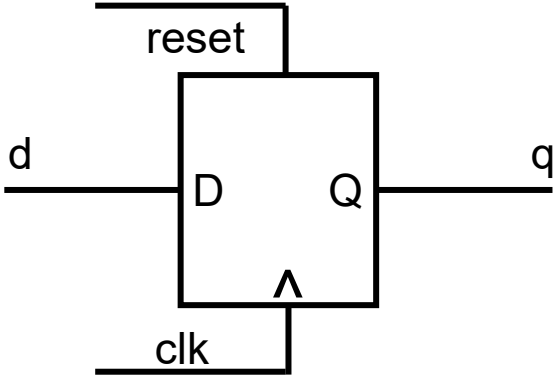



```
architecture RTL_DFF of DFF is
begin
process(CLK)
begin
  if rising_edge(CLK) then
    Q <= D;
  end if;
end process;
end architecture RTL_DFF;
```

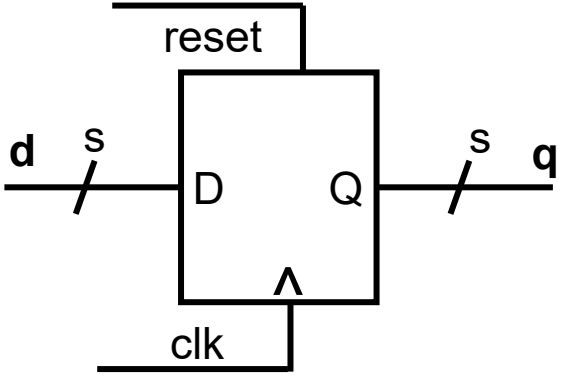
```
architecture RTL_DL of DL is

begin
  process(ENABLE,D)
  begin
    if ENABLE = '1' then
      Q <= D;
    end if;
  end process;
end architecture RTL_DL;
```

D Flip-Flop

- Input: D
- Output: Q
- Clock

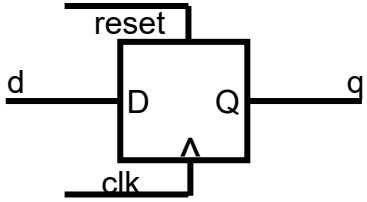


- Q outputs a steady value
- Edge on Clock changes Q to be D
- Flip-flop stores state
- Allows sequential circuits to iterate



D-flip-flop with asynchronous reset

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity DFF is
  port(
    clk, reset, d : in std_logic;
    q : out std_logic);
end entity DFF;
```



one-liner... (VHDL 2008 style)
- Use for single async. register assignment.

```vhdl
architecture oneliner of DFF is
begin
  q <= '0' when reset else d when rising_edge(clk);
end architecture;
```

- reset has priority
- Both clk and reset in sensitivity lists
  - **NEVER use 'all' for clocked sensitivity lists**

```vhdl
architecture signal_and_process of DFF is
begin
  process(clk, reset) is
  begin
    if reset then
      q <= '0';
    elsif rising_edge(clk) then
      q <= d;
    end if;
  end process;
end architecture;
```

Use when assigning multiple registers in a process

Variables *can* be used for FF instantiation, but...

No FF is created unless a signal is assigned to the state variable

```vhdl
architecture variabled of DFF is
begin
  process(clk, reset) is
    variable state;
  begin
    if reset then
      state := '0';
    elsif rising_edge(clk) then
      state := d;
    end if;
    Q <= state;
  end process;
end architecture;
```

Synchronous D-flip-flop

- Clock has priority
- Only clk in sensitivity list
  - (not reset, and *definetely not 'all'*)

```
architecture synchronous_reset of DFF is
begin
  process(clk) is
  begin
    if rising_edge(clk) then
      if reset then
        q <= '0';
      else
        q <= d;
      end if;
    end if;
  end process;
end architecture;
```

```
architecture sync_compact of DFF is
begin
  process(clk) is
  begin
    if rising_edge(clk) then
      q <= '0' when reset else d;
    end if;
  end process;
end architecture;
```

```
architecture separate_CL of DFF is
  signal next_state : std_logic;
begin
  next_state <= '0' when reset else q;
  q <= next_state when rising_edge(clk);
end architecture;
```

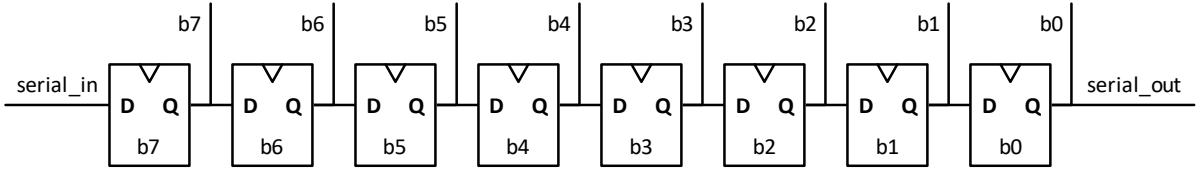
- Use this style when assigning multiple registers

Generally when writing RTL code:

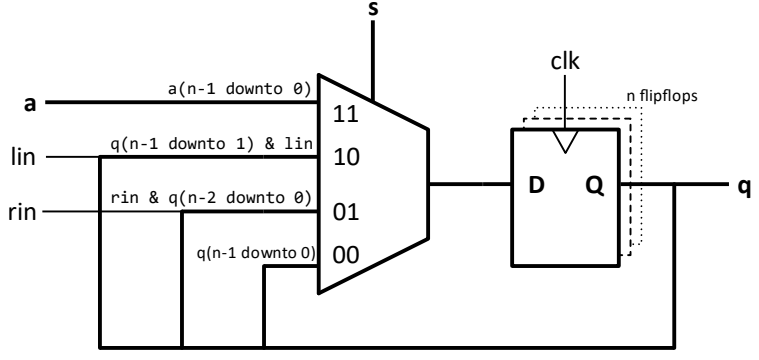
- Separate use of CL and clocked processes
  - Keep (all) register assignment in one process
  - Use one process for each purpose : ex
    - process for state assignment
    - process for state output
    - process for registers
- Avoid unintentional states by not combining CL and registers(!)
- Use synchronous reset unless specific reasons for async reset.
  - (reset circuits will be covered later)
- Simple statements can be written concurrently
  - Use only for 1 or 2 registers

```
architecture RTL of my_entity is
  signal r_x, next_x : std_logic_vector(7 downto 0);
  signal r_y, next_y : <your type>;
begin
  CL: process(all)is
  begin
     -- Create next based on input and registers>
     next_x <= ...
     next_y <= ...
  end process;

  REG_ASSIGNMENT: process(clk) is
  begin
     if rising_edge(clk) then
       if reset then
         r_x <= (others => '0');
         r_y <= ...; -- depending on your type
       else
         r_x <= next_x;
         r_y <= next_y;
     end if;
  end process;

end architecture;
```

Shift registers (not shifters)



- Shifter = CL
- Shift register = Flipflops connected in series
  - Used to parallelize serial input
    - Serial data transfer is used for high speed IO over distance
      - Ie largest parallel high speed io is memory buses on a PC main board
  - Can sometimes be used both ways
    - Ie both serial and parallel in/out
  - Assignment 4 lets you attempt this using structural code.



Parity calculation in VHDL 2008

- VHDL-2008 adds Unary Reduction Operators of the form:

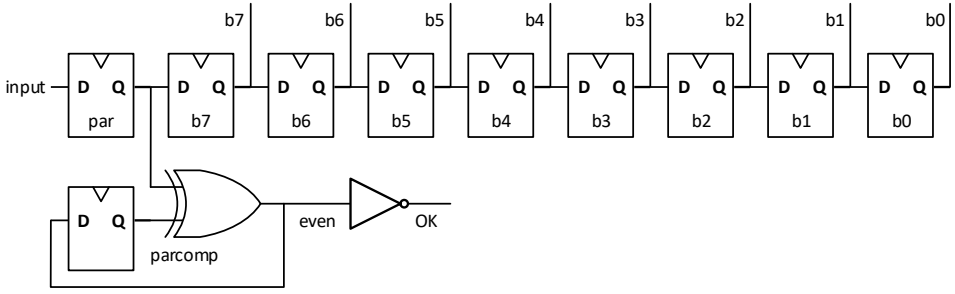
```
function "xor" ( anonymous: BIT_VECTOR) return BIT;
```

- Defined for arrays of bit and std_ulogic
- Defined for all binary logic operators:
  - AND, OR, XOR, NAND, NOR, XNOR

- Simplifies parity calculation

```
signal data : std_logic_vector(7 downto 0) ;
signal parity : std_logic;
. . .
parity <= xor data; -- even parity
```

Serial parity check

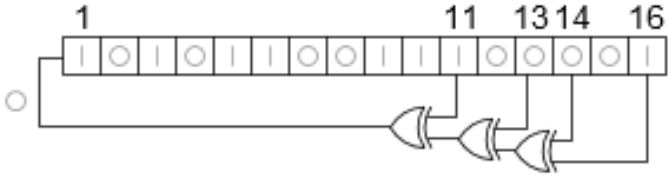


- Even parity
  - parity bit is even (‘0’) when there is an even number of bits that are ‘1’
  - Using even parity bit, each byte transmission (*including parity bit*) should always have even parity.
    - OK signal is high when even is ‘0’.

- Odd parity is «**not** even»

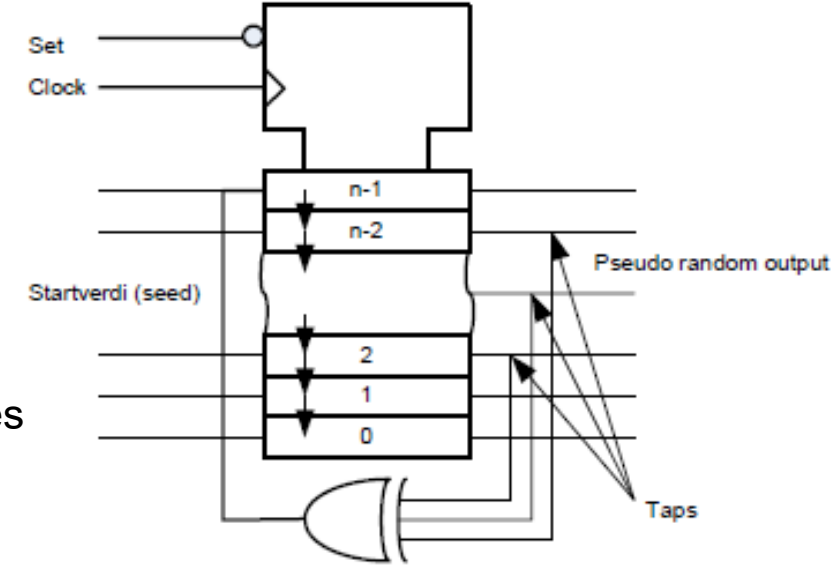
Linear Feedback Shift Register(LFSR)

- Made by xor-ing one and one bit that are connected back to MSB



- Apparently a random counting sequence
  - Nicknamed “Pseudo-random generator” since the counting sequence looks random
- It can be shown that it’s not needed more than three xor gates to make a random sequence
- *Some combinations are better*

https://web.archive.org/web/20161007061934/http://courses.cse.tamu.edu/csce680/walker/lfsr_table.pdf



- Used in testing of communication lines and buses
- Used in encryption

Oblig 3, Reccomended reading

- Oblig 3:
  - Peer review is required for passing
  - 2 peer review will be assigned to each
  - When in trouble, call the lab-assistant.
  - Be polite!

Challenge next page..

- Subprograms: This lecture
- Clocked processes and statements:
  - D&H:
    - 14.1-2 p 305-309,
    - 16.1-2 p 344-356

```
entity XXX is
  port (Clock : in  Std_logic;
  Reset : in  Std_logic;
  Enable: in  Std_logic;
  Load  : in  Std_logic;
  Mode  : in  Std_logic;
  Data  : in  Std_logic_vector(7 downto 0);
  X     : out Std_logic_vector(7 downto 0));
end;
```

(If-15 min...) challenge:

- 5 different architectures...
- Fill in what X is based on the input signals (in the table)
- How many FF's are created here?
- What type of circuit is this / What does it do?
- Raise you hand when finished...
- We will discuss and elaborate after

Enable	Load	Mode	X
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

https://www.uio.no/studier/emner/matnat/ifi/IN3160/v21/10-min-challenge/

To be revealed in the lecture