IN3160 IN4160

Design Quality and System design

Yngve Hafting 2020

Kursinfo

- 28.4 (Neste uke): Siste ordinære forelesning: *Prinsipper for testbenker*
- 5. mai: Q&A / Oppsummering
  - Siste sjanse til å stille spørsmål til faglærer før eksamen
    - Kan ikke love svar på forespørsler som kommer rett før eksamen.
- 12. og 19. mai: Eksamensforberedelse med Mojtaba
- Siste uke med labveileder er 8.-12. mai (O10 frist er 11.5)
  - Evt oblig-forsøk etter det må enten leveres med video eller avtales individuelt med retter. (Hver og en må ta ansvar selv her)
- Eksamen er 2. juni

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:

- Be able to describe quality parameters for digital designs
- Be able to
  - define digital systems at an architectural level
  - define digital system specifications at lower levels
- Know
  - principles for dividing systems into modules

Note: This is not covered in lab exercises.

Quality in designs = Things to keep in mind

- Simplicity
- Understandability
- Modifiability
- Testability / Verifiability
- Extensibility / Scaleability
- Reusability
- Portability
- Maintainability
- Performance
- Efficiency

Source for Quality in designs: Ricardo Jasinski: «Effective coding with VHDL»

Simplicity

- Simple code
  - every sequence of statements follow a logical order
  - easy to read and maintain
- Complex code
  - full of exceptions and special cases
  - ...
- Simplicity requires effort:
  - resist quick fixes to «get the job done»
    - rather go: «what is it we are trying to achieve..»
    - try make every piece fit
- «managing complexity should pay a major role in every technical decision» in a project

Simplicity metrics

- Number of lines *and*
- Number of statements
  - both can be a bit *stretched*
  - correspond well to number of bugs and how much effort is needed to understand the code.
- McCabe complexity:
  - complexity in nesting "*nøstedybde*"
  - Preferably < 6
  - 6-10: consider refactoring (rewrite but maintain function)
  - >10 => rewrite
- Modularity
  - Between extremes:
    - 1 big containing all vs
    - large number of trivial modules

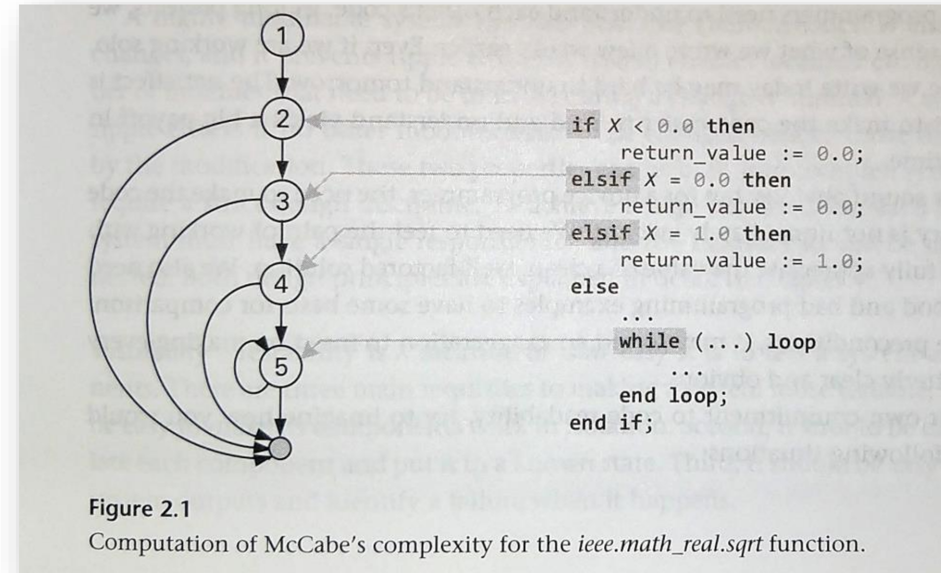


Figure 2.1
Computation of McCabe's complexity for the *ieee.math_real.sqrt* function.

The goal is not to have an exact number for every criteria, but to keep complexity at a manageable level.

Understandability

- IRL, Code is usually read more often than written
  - => It pays off making it readable for humans

- Reduce complexity
- Use comprehensive layout scheme

- example 2 next slides:

Understandability example 1/2...

```
architecture _1 of Count is
  signal Q: Unsigned(7 downto 0);
begin
  process (Clock, Reset)
    constant decade_max  : Unsigned(3 downto 0) := "1001";
    constant zero_nibble : Unsigned(3 downto 0) := "0000";
    constant zero_byte   : Unsigned(7 downto 0) := "00000000";
    variable next_Q      : Unsigned(Q'range);
  begin
    next_Q := Q + 1;
    if (Mode = '1') then
      if (Q(3 downto 0)) = decade_max then
        next_Q(3 downto 0) := zero_nibble;
        next_Q(7 downto 4) := Q(7 downto 4) + 1;
        if Q(7 downto 4) = decade_max then
          next_Q(7 downto 4) := zero_nibble;
        end if ;
      end if;
    end if;
    next_Q := Unsigned(Data) when not Load;
    next_Q := Q when Enable;
    Q <= zero_byte when not reset else next_Q when rising_edge(Clock);
  end process;
  X <= Std_logic_vector(Q);
end;
```

Specification of IO and function

```
entity Count 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;
```

Enable	Load	Mode	Next X
0	0	-	Data
0	1	0	X+1 (binary counted)
0	1	1	X+1 (decimal counted)
1	-	-	X

What are strong and weak points in this design?

Understandability example 2/2...

```
architecture _3 of Count is
  constant zero_byte: std_logic_vector(7 downto 0) := "00000000";

  function dec_count(input: Unsigned) return Unsigned is
    constant decade_max : Unsigned(3 downto 0) := "1001";
    constant zero_nibble: Unsigned(3 downto 0) := "0000";
    variable output     : unsigned(input'range);
  begin
    output :=
      input + 1                              when input(3 downto 0) /= decade_max else
      (input(7 downto 4) + 1) & zero_nibble when input(7 downto 4) /= decade_max else
      unsigned(zero_byte);
    return output;
  end function dec_count;

 signal Q          : Unsigned(7 downto 0);

begin
  Q <=
    unsigned(X)     when Enable   else
    unsigned(Data)  when not Load else
    unsigned(X) + 1 when not Mode else
    dec_count(unsigned(X));

  X <=
    zero_byte when not reset else
    std_logic_vector(Q) when rising_edge(Clock);
end;
```

Specification of IO and function

```
entity Count 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;
```

Enable	Load	Mode	Next X
0	0	-	Data
0	1	0	X+1 (binary counted)
0	1	1	X+1 (decimal counted)
1	-	-	X

How is this better or worse than the previous example?
Consider: *Being specific vs letting structure decide*

Modifiability

- A highly modifiable system
  - enables localized changes
  - prevents ripple effects – *avoid duplicate information*
    - every feature is written in one location only
      - (ie when changing bus width in the top module, all other modules follow without the need for changing every module separately)
  - Each module should have a single responsibility
  - Minimize connection between modules

Testability (verifiability)

- Components should be testable on their own
  - Make testbenches for
    - Modules
    - Subsystems
    - Entire system
  - Design modules in ways that are testable
  - Design test bench in parallell with module.
    - Test Driven Design:
      1. Write a failing test
      2. Make all tests pass
      3. Clean up and remove duplication
      4. Repeat until module is finished

Extensibility / Scaleability

- The ability to accomodate new functionality or being rescaled
- Use generic and parameterizable modules
  - Use packages...
    - Define (sub)types for data-transfer between modules in a package
    - Use functions for reusable CL
    - *Example...*

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

package inverter_package is
  constant WORD_SIZE : natural := 32;
  -- Design wide types
  subtype direction_type is unsigned(WORD_SIZE/2 -1 downto 0);

  -- Degrees in term of 2^(SIZE) = 360 degrees
  constant DEG0   : natural := 0;
  constant DEG60  : natural := (2**(WORD_SIZE/2))/6;
  constant DEG120 : natural := (2**(WORD_SIZE/2))/3;
  constant DEG180 : natural := (2**(WORD_SIZE/2-1));
  constant DEG240 : natural := DEG180 + DEG60;
  constant DEG300 : natural := DEG180 + DEG120;

    -- field_states: one state for each 60 degree sector in a full circle
  type field_state_type is
    (inactive, S1, S2, S3, S4, S5, S6, illegal);

  function determine_field_state(field_direction: direction_type) return field_state_type;
end package;

package body inverter_package is

  function determine_field_state(field_direction: direction_type) return field_state_type is
    variable field_state : field_state_type;
  begin
    field_state :=
      S1 when field_direction >=   DEG0 and field_direction <  DEG60 else
      S2 when field_direction >=  DEG60 and field_direction < DEG120 else
      S3 when field_direction >= DEG120 and field_direction < DEG180 else
      S4 when field_direction >= DEG180 and field_direction < DEG240 else
      S5 when field_direction >= DEG240 and field_direction < DEG300 else
      S6 when field_direction >= DEG300 else illegal;
    return field_state;
  end function;

end inverter_package;
```

Reusability

- Reusable code can easily be used in other places than it was designed for.
  - Using what is tested and proven is often preferred.

- Different levels of reuse:
  - Causual/ opportunistic reuse
    - using previous design code as template for modification
  - Module reuse
    - Can work both ways-
      - generic modules are harder to design
        - » *use only when you know it will be benificial.*
  - Formal / Planned reuse
    - Using libraries designed for reuse
      - *requires detailed documentation and testing*
  - Using Macros or IPs
    - *Developing a macro generally cost 10x a single use model.*

Portability

- The ability to be shifted from one environment to another
- *The opposite is tool-dependent or technology-dependent.*
- To ensure portability
  - avoid use of pragmas or metacomments
    - (ie compiler directives that are non-VHDL, often referring to vendor specific components).
  - avoid instantiating specific components in high level code
    - *isolate code that needs specific instantiation in specific modules.*
      - (typically used for clock networks etc.)

Maintainability

- The combination of
  modifiability, understandability, extensibility and portability

- Maintenance represent on average 60% of cost in SW system
  - => Making code easy to understand, then fix or modify is cruicial.

Performance

- Better addressed at higher levels rather than tweaking code
  - *Address issues in the architecture level rather than in code.*

- Keeping the focus at modularity and modifiability
  - ensures performance better than focusing on tuning performance.
- Tweaking code can lead to sacrificing portability, maintainability etc.-
  - Performance should be adressed *when it is known* that a module will not be able to meet performance requirements.
    - do not «optimize as you go»
    - Normally the compiler will select the best available option
    - Measure the performance before resorting to tuning. (synthesis reports)
    - Changing technology will change the assumptions you made when tuning.
- Effort put into tweaking parts that are not a part of the critical path is wasted...

Example needless performance tweaking:

- multiplying by 7 tweaked:

```
y <=
   ("0" & a & "00") +
   ("00" & a & "0") +
   ("000" & a );
```

performs the same or worse than in typical compilers...

```
y <= a*7;
```

- Which code is easiest to maintain?
- Which code is most readable
  - will most likely conceal bugs?...

Efficiency

- The ratio of work done to the resources used.
- A highly efficient system use less energy or area than a lesser one.
- The same rules as for performance applies:
  - Tweaking for efficiency is something you do when
    - you have proof that this is what you should do.
      - Reports from the synthesis tool will aid you to make such decisions.

Architectural design

- What do we mean by architectural design?
(≠ VHDL architecture)
- A. decisions
- A. specification

Source for Architectural design: Ricardo Jasinski: «Effective coding with VHDL»

Architectural vs non-architectural design

Feature	Architectural design	Non-architectural design
Quality attributes	defines system-level constraints and goals for each module	Does it meet the architectural goals and constraints?
Scope	system–wide decisions	descisions are local to component
Behavior	defines behavior of a system	defines behavior of a component
Structure	defines the major blocks of a system and how they communicate	Must adhere to the given architectural structure
Data	Abstract and conceptual models of data	Concrete data models and structures (specified down to bit level)

Source: Ricardo Jasinski: «Effective coding with VHDL»

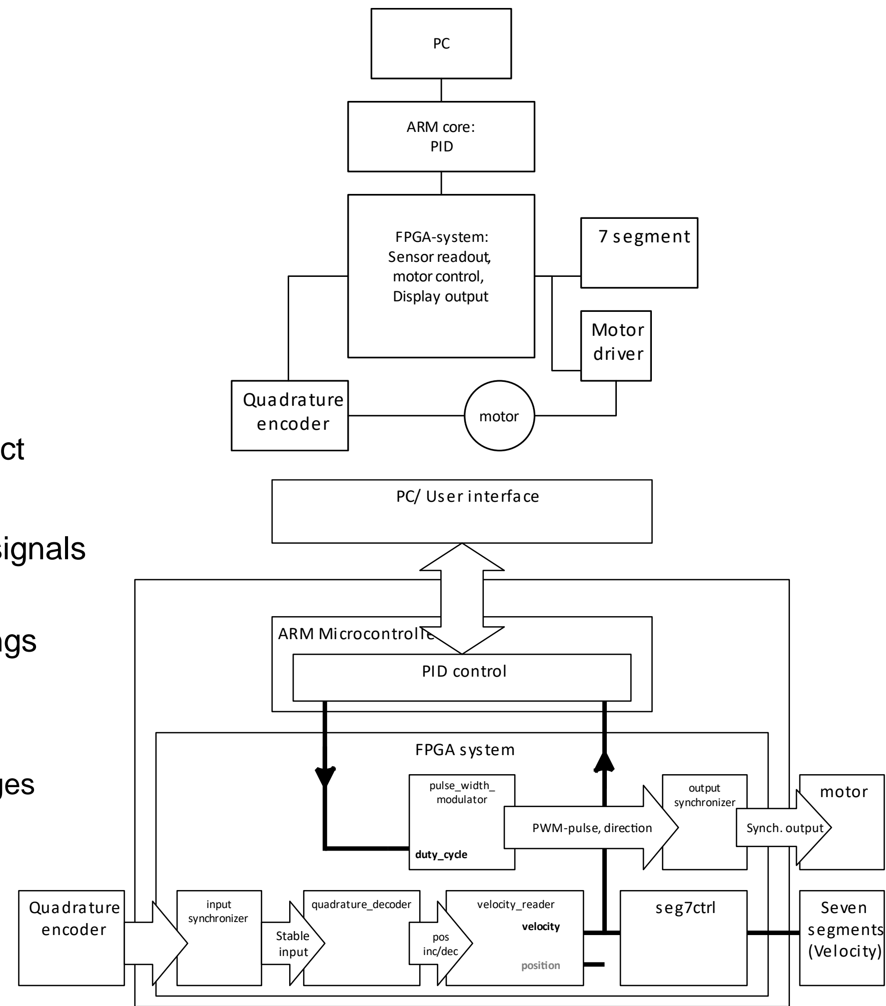
Architectural decisions (examples)

- Clock and reset schemes
  - Should all the system inputs and outputs be registered?
  - What type of storage elements are allowed?
    - FFs, latches, RAM,
  - Should reset be synchronous or asynchronous?
  - How many clock domains do we need?
    - how do we communicate across clock domains?
- Computation
  - Should computation be done serially, or in parallell?
  - Do we need pipelining?
- Modules
  - how should the different modules communicate
  - Do we use ad-hoc connections or an on-chip bus?
- etc.

Architecture specification

should consist of:

- An overview of the system
  - Major components and their interactions
  - understandable to all participants (new and old) in a project
- High level diagrams
  - block diagram: Major blocks, data paths, memories, key signals
  - information exchange - not details
  - context diagram: how the system interacts with surrondings
- Design decisions
  - Decisions backed by rationale behind
    - to enable all factors to be considered when making changes
- Mapping between requirements and components
- Constraints
  - Resources that are allowed/ disallowed
    - design libraries,
    - chip families,
- Design principles that all designers should adhere to.
  - such as organization in layers



There exists standards for architectural descriptions, such as ISO/IEC/IEEE 42010 «Systems and software engineering- Architecture description»

System design

- Overview
- Process
- Specification
- Divide and Conquer

Source for System design: Dally, Harting, Aamodt: Digital Design using VHDL

source: D&H 21. Book uses examples and should be read.

System design process (in practice)

- Specification
- Partitioning
- Subsystem interfaces
- Timing
- Module design
- Tuning

System Design – a process

- Specification
  - *Understand what you need to build*
- Divide and conquer (Partitioning)
  - *Break it down into manageable pieces*
- Define interfaces
  - *Clearly specify every signal between pieces*
  - *Hide implementation*
  - *Choose representations*
- Timing and sequencing
  - *Work flow / Data flow between modules*
  - *Overall timing – use a table*
  - *Timing of each interface*
    - *– use a simple convention (e.g., valid – ready)*
  - *Add parallelism or pipelines as needed*
- Design each module
- Code
- Verify

Iterate back to the top at any step as needed.

	Output / deliverables
	Functional description
	Simple Block diagram
	Block diagram with interface description
	Wave diagrams (for data transfers) Data path diagrams
	FSM diagrams and tables • ASM(D)/bubble and state/input-..
	HDL files, Schematics, source code
	Testbench reports Static timing reports

(c) 2005-2012 W. J. Dally

Specification

- Write the user's manual first
- Putting it on paper means that there are no misunderstandings about operation
  - In practice, this also serves to validate the specification with users/customers
- Spec includes
  - Inputs and outputs
  - Operating modes
  - Visible state
  - Discussion of "edge cases"
- Most of design is done writing English-language documents – with associated drawings. Coding comes later.
  - Don't start coding until your design is complete.

(c) 2005-2012 W. J. Dally

Divide and Conquer –common themes

- Task
  - Divide system into a network of tasks
  - One module per task
- State
  - Divide system by state
  - Separate module for each set of state variables (each state machine on its own)
- Interface
  - Module for each external interface

(c) 2005-2012 W. J. Dally

Some comments on Coding

- Don’t start coding until your design is done.
- *"Don’t even think about coding until your design is done"…*

- Code a separate module for every block in your basic block diagram
- Verify each module before moving on to the next

- Follow good VHDL coding practice…
  - Don’t forget it is hardware
    - *synthesizable = a circuit description*
    - *No falling edges, please.. (!)*

- Debug the whole system in simulator
  - *before implementing on hardware*

Suggested reading

- DHA
  - 21 p 467-477
  - Appendix A p 611-621

- Exercise suggestion:
  Try make an architectural specification of your oblig 8 system.
  - Does this in any way change your perception of this task?