Specification and Verification of Hardware

VHDL 1

Original by Magnus Björk

Computing Science

Chalmers University of Technology

The course

- Course team:
  - Mary Sheeran (kursansvarig)
  - Dennis Walter
  - Emil Axelsson
- Guest lecturers:
  - Jiri Gaisler (Gaisler Research)
  - Emily Shriver (Intel)

About the course

- Check in often at: http://www.cs.chalmers.se/Cs/Grundutb/Kurser/svh/
- Two halves: VHDL part and Lava part.
- Each half includes one lab and one take home exam.
- Written exam in then end
- Sign up for one lab account each.

You may solve the labs in pairs, but are not allowed to cooperate between the pairs. No cooperation at all on the take home exams (do them individually).

Course Book: Two Options

- Peter Ashenden: **The Designer's Guide to VHDL**
  - Approaches VHDL as any programming language
  - Focuses on simulation.
  - *The* VHDL book used in industry (Ashenden is part of the committee behind the VHDL standard, this book is sometimes seen as a more readable version of the standard.)

- Stefan Sjöholm, Lennart Lindh: **VHDL för konstruktion**
  - Approaches VHDL as a tool for constructing hardware
  - Focuses on synthesis
  - Cheaper and more concise

Overview of the lecture

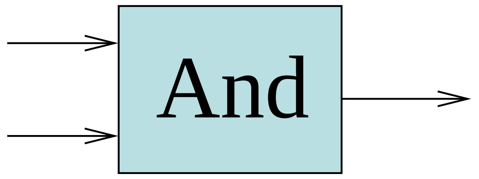
- A 15 minutes crash course in hardware design
- VHDL

Please interrupt me if you have questions

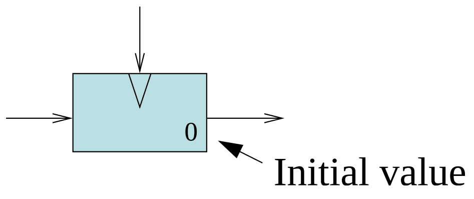
Fundamental hardware concepts

Gate level

- Can be compared to assembler for software
- We may use:
  - Gates: Not, And, Or, Implies, Multiplexers

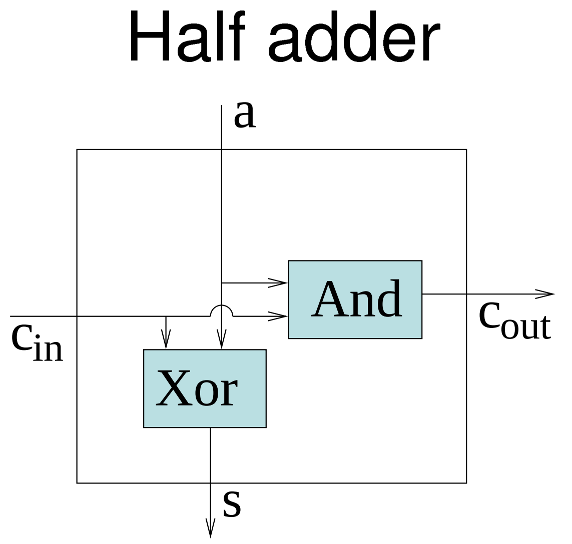


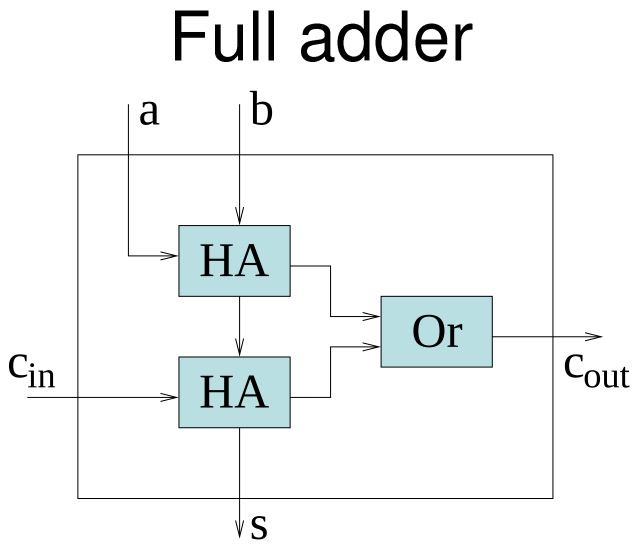
  - Memory cells: d-flipflop

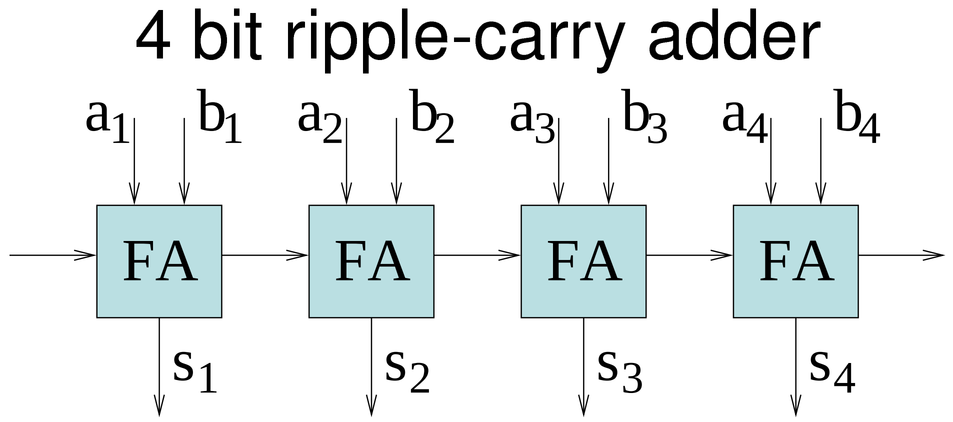


  - Components: boxes containing gate-level descriptions

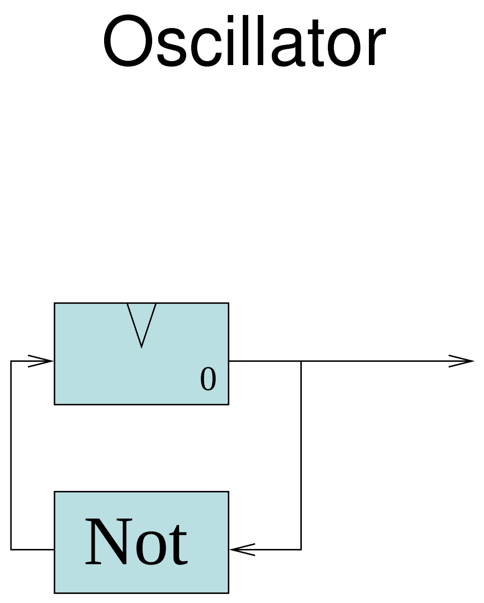
Combinational gate level examples

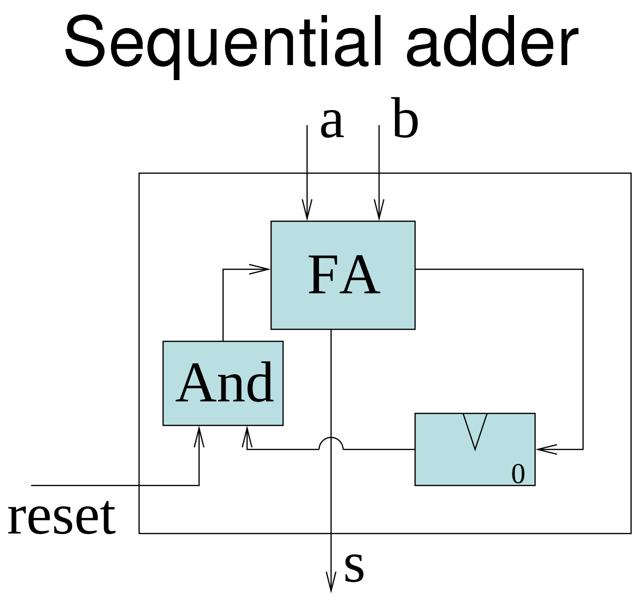






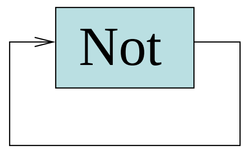
Sequential gate level examples



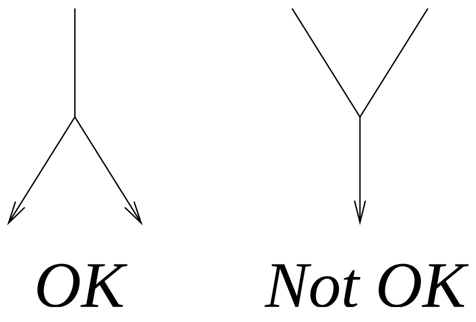


Be careful with

- Feedback **only** through flipflops



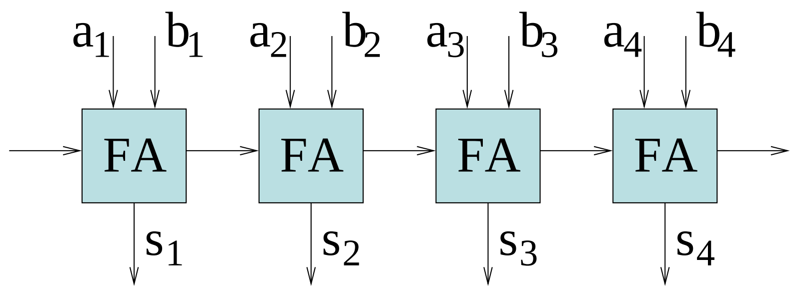
- You may split wires, but never join them:



In this course, any wire may only have **one** driver.

Abstraction domains (1)

- Structural domain:
  Circuit described as a composition of sub-circuits



- Behavioral domain:
  Describes **what** the circuit does, **not how** it does it

s := a + b;

Abstraction domains (2)

- Behavioral is the “desired” domain *(why make more complicated code?)*
- But, behavioral code
  - May not be synthesizable

    ```
    if c /= c'delayed(T) then ...
    x <= y / z;
    ```

    Time-shift and division can be simulated, but not synthesized
  - May lead to inefficient circuit

    ```
    a := b + c;
    d := e - f;
    ```

    Maybe only one adder is needed?

Abstraction domains (3)

- Structural decomposition is usually necessary for getting efficient and synthesizable code
- No design is completely structural
  - Components $\rightarrow$ Gates $\rightarrow$ Transistors $\rightarrow$ Silicon crystals $\rightarrow \ldots$
- Sooner or later we reach the leaves of struct. decomposition. The structural leaves are always described behaviorally.
- What is a suitable level for the leaves?
- Common answer: **Register Transfer Level (RTL)**

Register Transfer Level (1)

- RTL = Gate level, plus:
  - Wires carrying more complex data types (bit-vectors, integers, arrays of integers, ...)
  - Components operating on those types (adders, multipliers, sorters, ...)
  - Registers and memories (generalization of d-flipflop, storing any type)
  - Subcomponents for hierarchical structure
  - Finite state machines (FSMs)
- These components are described **behaviorally**!
  - An adder does not have to be decomposed into full adders
  - A counter (an FSM) does not have to be decomposed into a register and an adder

Register Transfer Level (2)

- Correct timing behavior internally: it should be possible to see from the RTL description what happens in each clock cycle.
- Designer controls overall structure, synthesis tool controls low-level details (RTL synthesis)

Behavioral Level (1)

- Describes **what** the circuit does, **not how** it does it
- Doesn’t say anything about the structure of the circuit
- Uses standard programming language concepts such as algorithms, loops, complex data types and processes.
- Shows correct timing behavior externally, but not internally
- Important aspects
  - Must be easy to understand
  - Must be unambiguous
  - Is used as a runnable specification; behavior is compared with RTL implementation during test phase

Behavioral Level (2)

- Used to specify what your circuit does
- Can be discussed with customer as initial step of design
- Can also be used to simulate other parts of the circuitry which are developed by somebody else
- Is probably not synthesizable

Functional Level

- Even higher level than behavioral
- Only describes the function of the circuit
- Not even correct timing behavior
- May use any programming language constructs

Abstraction Levels (1)

Say we want to describe a pipelined sorter for integers:

- **Functional level:** Your favorite sorting algorithm
- **Behavioral level:** The same algorithm, with result delayed a number of clock cycles to get correct timing
- **Register transfer level:** A structural description using comparators, registers, and wires carrying integers. Uses the same sorting method as the final circuit (probably not the same as in previous steps).
  Two versions:
  - All registers at the end of the datapath
  - Registers distributed in the correct places.

Abstraction Levels (2)

- **Gate level:** An incomprehensible structural description using registers, gates and wires carrying booleans
- **Transistor level:** A structural description using only transistors and wires carrying different voltages

Design process (simplified)

1. Write a specification
2. Write an implementation
3. Verify that your implementation meets the specification
4. If not, are the errors in the specification or the implementation?
   Correct the erroneous one.
5. Repeat step 3-4 until no more errors are found.

Iterative design process

1. Write a (runnable) specification.
2. Do some verification of the specification, to convince yourself that it is correct.
3. Write an implementation on a slightly less abstract level or with more optimizations
4. Verify that the new implementation works like a previous one (perhaps the last implementation, or the specification)
5. Repeat step 3-4 until you have an optimized low level description of your circuit.

If Spec. and Implementation Differs?

If your final design doesn't behave as the behavioral description, you have three options:

1. Change your final design to match the behavioral model
2. Change your behavioral model to match the final design
3. Find out exactly how different they are (this isn't as easy as you think). Determine if they are similar enough, and thoroughly document the differences

First option is preferable, since you don't need to update the contract with your customer. If you discover that the agreed upon behavior was hard to implement, second option is probably best. Third option is usually harder than second.

(Functional) Verification techniques

- Testing: write testbenches, simulate circuit (ModelSim)
- Exhaustive testing: Generally not feasible
- Automated testing: Safelogic Monitor, FoCs, QuickCheck
- Property checking (model checking): Jasper Gold, RuleBase, Solidify, Formality
- Equivalence checking: eCheck, conformal (if 1-1-match of dffs)
- Other methods: Symbolic simulation, Theorem proving, ...

Last three known as *formal verification*