Lecture Notes

Introduction to Embedded Systems

Winter Semester 24/25

Emma Bach

Lecture by
Prof. Dr. Oliver Amft

February 18, 2025

Table of contents

1 Introduction **3**
1.1 Definition 3
1.2 Design 4

2 Specification **5**
2.1 Automaton Models 5
2.1.1 State Machines 5
2.1.2 State Charts 6
2.1.3 Petri Nets 8
2.2 VHDL 9
2.2.1 Testbenches 10
2.2.2 A Full Adder in VHDL 11
2.2.3 Data Types in VHDL 12
2.2.4 Operators 13
2.2.5 Constants and Signals 13
2.2.6 Variables 14
2.2.7 Processes 14
2.2.8 Statements 15
2.2.9 Functions and procedures 16
2.2.10 Synthesisable vs Non-Synthesisable Code 17
2.2.11 Simulation 18
2.2.12 Delay Modeling 18

3 Design Space Evaluation **20**
3.1 Power Consumption 20
3.2 Quality Testing 21
3.3 Pareto Frontier 22

4 Hardware **23**
4.1 Microprocessors vs Microcontrollers 23
4.2 Memory 23
4.2.1 Cache Design 24
4.2.2 Scratch Pad Memory 24
4.2.3 I/O Access 25
4.2.4 Interrupts 25
4.3 Communication 25
4.3.1 Pulse Width Modulation 26
4.3.2 Bus Standards 26
4.4 Debugging 28
4.4.1 JTAG 28
4.5 Signal Processing 28
4.5.1 Direct Conversion 28
4.5.2 Sequential Conversion 29
4.5.3 Signal Theory 31

5 Software **32**

A Sources **33**

Chapter 1

Introduction

1.1 Definition

There is no fully rigorous 'mathematical' definition that cleanly seperates everything that is an embedded system from everything that isn't. Instead, embedded systems exist on a spectrum. We can say that a system is *more embedded* or *less embedded* depending on how many of the typical properties of an embedded system apply to it. **Embedded Systems** are **computer systems** that tend to:

- be **integrated (embedded) into a larger system**, which they may control and/or provide information processing for.
- be **specialized** to provide exactly the functions they need to.
- be forced to work with **constraints** in time, memory, energy consumption, space, etc.

The term **Cyber-Physical System** generally refers to a larger system that combines computational elements with physical elements, with embedded systems generally being smaller components of such a system. Examples of Cyber-Physical Systems include **IoT** (Internet of Things) devices, **Ubiquitous Computing** devices, and **SCADA** (Supervisory Control and Data Acquisition) systems.

An embedded system generally consists of physical components such as sensors and actuators, computational components including memory and processors, and software. Since the computational components tend to work with digital representation of numbers, while the physical components work with analog voltages, additional conversion through A/D and D/A converters is needed.

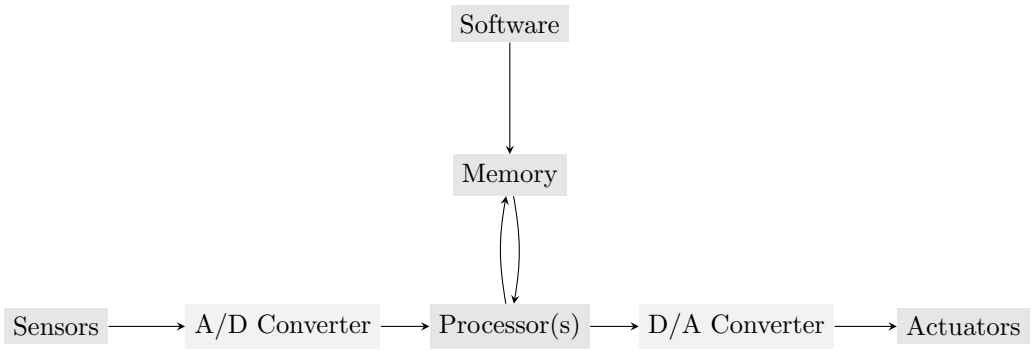

Figure 1.1: The structure of a typical embedded system.

Embedded systems have extremely widespread applications, especially in fields such as automotive and aerospace engineering and in medical technology.

1.2 Design

Embedded systems must be **dependable**:

- The **Availability** is the probability of the system working at time t.
- The **Reliability** of an embedded system is the probability of the system working correctly provided that it was working at time t=0.
- The **Maintainability** is the probability of the system working correctly at time t+d after encountering an error at time t.
- Additional factors are **Safety** (How much harm could the system potentially cause?) and **Security** (How resistant to outside interference is the system?)

Since embedded systems are often employed in safety-critical roles, such as in the aforementioned aerospace industry, dependability is extremely important. For safety-critical systems, **redundancy** is generally a desired trait, so that if one component fails there are still other components to cover the same function.

Embedded systems also generally need to be efficient enough to meet constraints in:

- Energy consumption
- Physical Size
- Code Size
- Required Memory
- Runtime
- Weight
- Cost

Lastly, Embedded systems are typically **reactive systems**, meaning that they work through interacting with their environment at a pace dictated by that environment. This often also makes them **real-time systems**, meaning they need to meet real-time constraints - If a right answer arrives too late, it is just as bad as a wrong answer. If failure to meet a deadline results in catastrophe, that constrained is called a **hard constraint**. This means that worse average runtimes are acceptable, or even necessary, if it leads to a better worst case runtime.

Chapter 2

Specification

Design by Contract (**DbC**), also known as contract programming or simply as internal testing, is the idea that software designers should define precise, formal, verifiable specifications for the desired behaviour of their systems. These often extend the ordinary usage of abstract data types with the description of desired preconditions, postconditions and invariants. The first steps of designing a system thus come down to translating the requirements of the project from natural language into a formal specification that verifiably meets the desired goals.

Such specifications generally involve the **abstraction** of a given system in order to simplify its description, and hierarchical separation of the description, in order to make a description more easily digestible. We distinguish between two kinds of hierarchy:

- Behavioral hierarchy, which describes a systems behavior in terms of states, events, and output signals. Examples of concepts of "high level" behavioral hierarchy are interrupts and exceptions.
- Structural hierarchy, which describes how a system can be thought of as a collection of seperate components: processors, actuators, sensors, etc.

Specifications generally need to describe a systems **Timing Behavior**, especially in the case of real-time systems. This involves specifying the elapsed time during execution of a given task, the delay between processes, Timeouts (maximum waiting times for a given event), and Deadlines.

It is helpful to model a system as a flow of states (**State-Oriented Behavior**). However, classical automata are often insufficient, since they don't model timing and don't support hierarchical description. Common traits that a specification has to be able to model are:

- Event-handling for both internal and external events (preferably supporting exception-oriented behavior)
- Modeling of concurrency
- Modeling of communication between different systems. Different models include:
  - Synchronous communication (A sends information directly to B)
  - Asynchronous communication (A writes to a buffer, the information is later sent to B)
  - A and B write to and read from shared memory

In order to qualify as a formal specification in the first place, naturally the specification needs to also have unambiguous semantics that support formal verification.

2.1 Automaton Models

2.1.1 State Machines

As a reminder: a **finite state machine (FSM)**, also known as a **deterministic finite automaton (DFA)**, is a tuple $M = (I, S, s_0, \delta, F)$, such that:

- I is a finite, non-empty set, called the set of input **symbols**
- S is a finite, non-empty set, called the set of **states**
- $s_0 \in S$ is called the **initial state**
- δ is a function $S \times I \to S$, called the **t r a n s ition function**
- $F \subseteq S$ is called the set of **final states** or **accepting states**

As a computer science student, I myself was already very comfortable working with DFAs before this lecture, and I will thus skip the basic explanations of the topic.

One major limitation of classical DFAs is that they don't model outputs of a system in any way. The two basic ways of adding outputs to DFAs are **Moore machines**, which specify an output symbol for each state (i.e. in addition to the set O of output symbols, we add a map $S \to O$ to our specification), and mealy machines, which specify an output symbol for each **transition** (i.e. we change δ to be a map $S \times I \to S \times O$). The process by which Moore and Mealy Machines turn an input sequence into an output sequence is also known as **t r a n s ducing**.

Moore Machines are a subset of Mealy Machines, in the sense that a Moore Machine can be seen as a Mealy Machine where every transition to the same state $s \in S$ produces the same output symbol. Nevertheless, the majority of Mealy Machines have an equivalent Moore Machine, and the ones that don't still have a semi-equivalent Moore Machine, meaning that if a Mealy Machine M produces any output $o_1, \ldots, o_n$ from a input $i_1, \ldots, i_n$, the semi-equivalent Mealy Machine will output $x, o_1, \ldots, o_n$ (i.e. the output contains a new starting symbol, and everything afterwards is correct but comes one "iteration" later than expected).

When implemented as circuits, Moore Machines are safer to use than Mealy Machines, since the output always change exactly during the next clock cycle after the input has changed. Mealy Machines don't have to wait for a clock, making them faster to react to an input, but this may lead to undesired asynchronous feedback.

2.1.2 State Charts

Classical Automata are generally not suited for the description of complex systems, because they very quickly become unmanageable when trying to model concurrency or complex processes not suited to non-hierarchical descriptions. This is the primary use case of **state charts**.

Statecharts were first described by David Harel in 1987. They have since become industry standard in many applications, describing hardware, software (UML!) , and communication protocols.

Transitions within a statechart work the same way they do within a mealy machine, although the input and output symbols tend to be referred to as "events". Instead of "on recieving input a, the machine produces output b", one generally says "if event f is present, produce event g".

Since multiple events can happen at any given time, state charts may be non-deterministic. In 2.1, it is undefined what should happen if f and h happen at the same time.

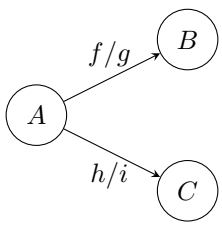

Figure 2.1: A statechart that may display unwanted non-deterministic behavior

This problem can be avoided by explicitly stating which transition should be prioritized. This is generally done by having all transition events be disjunct. 2.1.2 shows how this may be applied to the state chart from 2.1. If both f and h are encountered at the same time, the chart now knows to always prioritize the transition to B, while transition to C only happens if h is present while f is not.

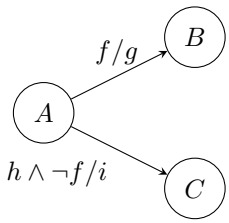

Figure 2.2: A fix to the chart's nondeterminism

The big advantage of state charts compared to regular state machines is that they allow **superstates**, where an entire state machine may be contained within a single state of a "higher level" machine.

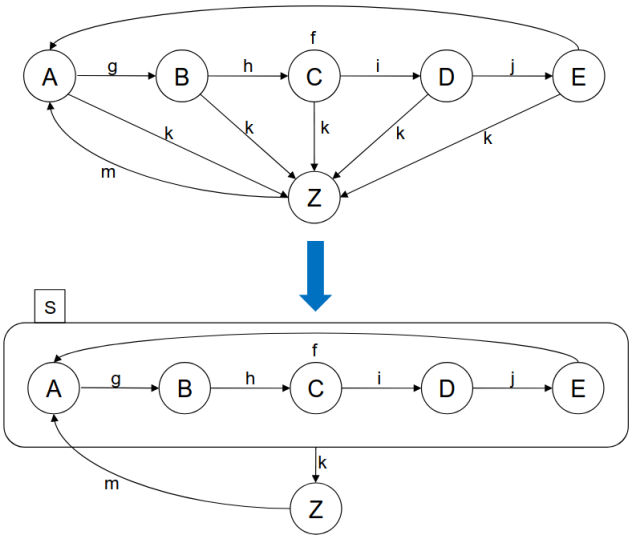

Figure 2.3: Using superstates, this automaton can be drawn in a much tidier way.

As shown in 2.3, this can make state charts significantly more readable than equivalent state machines. The basic type of superstate is the **OR-Superstate**, where an automaton may only be in one substate of a given superstate at any time. Later on, we will also introduce **AND-Superstates**, where an automaton is in <u>all</u> of the substates of a given state at the same time.

States within a state chart that don't contain substates are known as **basic states**. The set of superstates containing a given state are called the **ancestor states** of that state. When a basic state is active, all of its ancestor states are also considered active. If an active basic state and its ancestor state would both undergo a transition at the same time, the "higher level" transition of the ancestor state takes priority. See 2.4 for an example.

By default, each automaton within a superstate contains a default starting state, and whenever the superstate is left, the automaton "forgets" what state it was in at that point and simply enters the starting state again if the ancestor state becomes active. This can be changed using **History States**.

In 2.5, the state H is the history state (the notation for history states seems frustratingly opaque and inconsistent - it seems like in most state charts, history states really aren't drawn in a special way, instead the historyness is only shown through them being called H.) This means that, whenever H is entered,

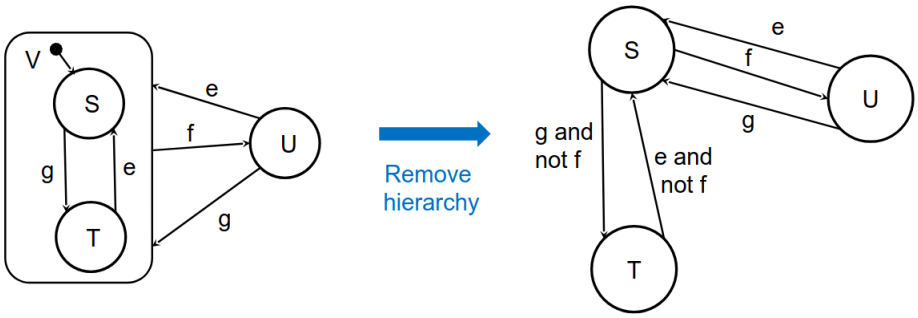

Figure 2.4: A state chart using OR-Superstates and an equivalent state chart without hierarchy

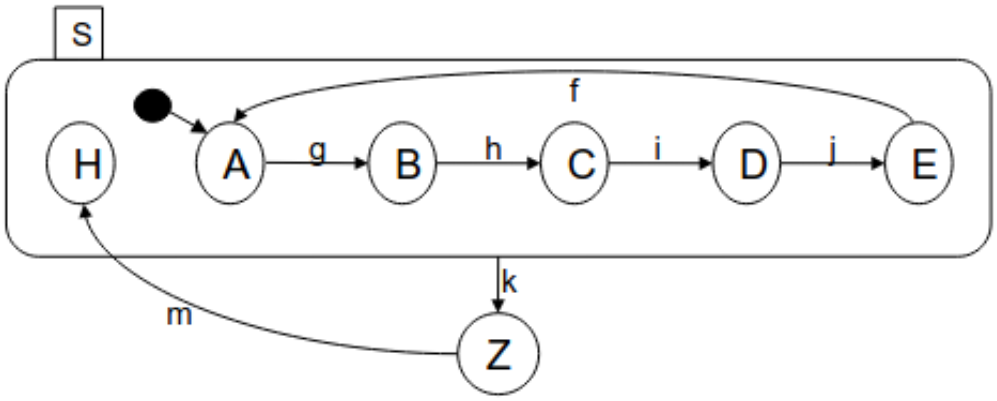

Figure 2.5: A state chart that uses history states

the subautomaton that H is a part of instead enters the state it was in when it was last active. If the state has never been active, it enters the starting state A, signified here through an arrow and a black circle. If the state chart uses a **Deep History Mechanism**, that includes remembering what substates were last active, if the state the history state is pointing to is not a basic state.

2.1.3 Petri Nets

This section has been significantly shortened because of time pressure and general sleepyness, and also because the lecture spends ages on the trivial parts while completely rushing through the harder parts =w=

A **place/transition net**, or petri net, is a tuple $N = (P, T, F, M_0$, such that:

- P is a set of *places* (equivalent to an automatons states)
- T is a set of *events* (transition conditions)
- F is a set called the *flow relation*, connecting each place to an arbitrary amount of events and each event to an arbitrary amount of places using directional edges
- $K : P \to (\mathbb{N}_1 \cup \omega)$ assigns a capacity to each place (assumed to be ω if not explicitly stated)
- $W : F \to (\mathbb{N}_1)$ assigns a weight (= capacity) to each edge (assumed to be 1 if not explicitly stated)
- $M_0 : P \to \mathbb{N}_0$ is the initial marking, which specifies the amount of tokens each place starts with.

A **condition/event net** is a petri net such that:

- the net is **simple**, meaning that there can't be two seperate events directly connecting two places while going in the same direction (cycles are allowed)
- the net has no isolated elements
- the set of markings is closed with regards to firing and inverse firing (i.e. any marking can be turned into any other marking by the right sequence of firing and inverse firing)
- the net is **deadlock-free**, meaning that each reachable marking M has at least one event e that can be fired from M. This condition is also known as **weak liveness**.
- every place has a capacity of 1 and intuitively represents a boolean condition.

A C/E net is called **cyclic** if any two markings are reachable from each other. C/E fulfills **liveness** if for each Marking M and event e there exists a Marking M' reachable from M that can fire e. A live petri net is always deadlock-free.

A petri net can be specified by a matrix M where the rows represent places and the columns represent events, with the value of each entry corresponding to the amount of tokes moved to (positive value) or from (negative value) the corresponding place by the corresponding transition. A petri net is called **contact-free** if capacities of places never prevent firing of the net. A petri net is called **pure** if it doesn't contain any self-loops.

2.2 VHDL

VHDL is a **Hardware Description Language**, meaning that it describes digital circuits (instead of abstract algorithms).

VHDL code is split into **entities** and **architectures**. Entities describe ports, such as inputs (*in*), outputs (*out*), bi-directional ports (*inout*), and *buffers* (Output that the entity itself can read). An architecture defines the actual implementation of an entity - internal wiring, connection of signals, and assignment of values. For example, an OR gate could be implemented as:

```
entity orGate is
    port(a,b: in bit;
    c: out bit);
end orGate;
architecture arch1 of orGate is
begin
    c <= a or b;
end arch1;
```

or:

```
entity orGate is
    port(a,b: in bit;
    c: out bit);
end orGate;
architecture arch2 of orGate is
begin
    c <= 1 when (a = '1' or b = '1') else 0;
end arch2;
```

There may be several architectures for a single entity. By default, the most recently analyzed architecture is the one that ends up being used.

2.2.1 Testbenches

A testbench is a VHDL Design without inputs or outputs, designed to test another VHDL Design, generally through Port mapping and verifying that the entity produces the correct outputs for given inputs. For example, a test bench for our OR gate could be realized as:

```
entity testbench is
 --empty
end testbench;

architecture test of testbench is
 signal d,e,f: bit := '0'

begin
 or1: entity orGate port map (a=>d,b=>e,c=>f);
 d <= not d after 10 ns;
end;
```

The **port map** maps the signals d, e, f in the software to the ports a, b, c in the hardware. It is also possible to use positional association instead of explicit association, which means simply writing

```
architecture test of testbench is
 signal d,e,f: bit := '0'

begin
 or1: entity orGate port map (d,e,f);
 d <= not d after 10 ns;
end;
```

after which the compiler will assign the ports based on the order of the signals in the port map. It is however good practice to always use explicit association.

There are three concepts used in testing:

- `report`, for print-like outputs
- `assert`, for specifying conditions
- `severity`, for specifying how the statement should affect the run of a simulation

If needed, combinations of the three can be used within a single line:

```
report <message_string>
report <message_string> severity <severity_level>;

assert <condition>;
assert <condition> severity <severity_level>;
assert <condition> report <message_string>;
assert <condition> report <message_string> severity <severity_level>;
```

If all three are used, the following happens: If the assertion is violated, the program sends a message `"message"`, and the whole thing is treated as an 'incident' with a predefined `severity_level`. If an `assert` doesn't have a `severity_level`, then the severity level will be `error`. If no `message_string` is specified, the message will be `"Assertion Violation"`. If a `report` without an assertion does not have a `severity_level`, the severity level is implicitly defined to be `note`.

`report` statements are inherently sequential, meaning they can occur inside processes, but not (by themselves) in architectures. However, `assert`ions can be either sequential or concurrent.

2.2.2 A Full Adder in VHDL

```
entity fullAdder is

 port(a,b,cin: in bit;
      sum,cout: out bit);

end fullAdder;
```

Dataflow description of the architecture:

```
architecture dataflow of fullAdder is

begin
 sum <= (a xor b) xor cin;
 cout <= (a and b) or (a and cin) or (b and cin);
end dataflow;
```

Components are entities used within a **structural definition** of an architecture, where a new architecture is defined as an interconnected circuit of already known smaller components. They are defined either via component and signal binding or via entity instantiation. For example, a fully structural definition of a full adder would be something like:

```
entity FULLADDER is
  port (A,B, CARRY_IN : in bit;
        SUM, CARRY    : out bit);
end FULLADDER;

architecture STRUCT of FULLADDER is
  component HALFADDER
    port (A, B       : in  bit;
          SUM, CARRY : out bit);
  end component;

  component ORGATE
    port (A, B : in  bit;
          RES  : out bit);
  end component;
  signal W_SUM, W_CARRY1, W_CARRY2 : bit;

begin

  MODULE1 : HALFADDER
    port map(A, B, W_SUM, W_CARRY1);

  MODULE2 : HALFADDER
    port map (W_SUM, CARRY_IN,
              SUM, W_CARRY2);

  MODULE3 : ORGATE
```

```
    port map (W_CARRY2, W_CARRY1, CARRY);

end STRUCT;
```

2.2.3 Data Types in VHDL

Standard data types

The standard data types provided by VHDL are:

bit	0,1
boolean	true,false
character	most ASCII characters
integer	$-2^{31}-1, \ldots, 2^{31}-1$
real	$-1.7e38, \ldots, 1.7e48$
time	1fs, ..., 1hr

Users can also define their own datatypes, either as integer types:

```
 --64 bits
type small is range 0 to 63;

 --32 bits
type result32 is range 31 downto 0;

 --16 bits
subtype result16 is result32 range 15 downto 0;
```

or as enumeration types:

```
type state is (idle,start,stop);
type hexDigits is ('0', {...} , '9', 'A', 'B', 'C', 'D', 'E', 'F')
```

It is important not to get confused between statements like the following:

- `signal S : integer range 0 to 3;`, meaning a number between 0 and 3
- `signal S : unsigned(3 downto 0);`, meaning an unsigned 3-bit integer (i.e. a number between 0 and 7)

Also note that for most datatypes, `downto` corresponds to little endian (i.e. most significant bit first), while `to` corresponds to big endian.

std_logic

In realistic circuits, voltages may come in many forms not accurately described as simple boolean variables / bits. To model these, the datatype `std_logic` is used, which contains signal types such as:

0,1	"Ground" and "High" Voltages
U	uninitialized
X	unknown, impossible to determine (generally a short circuit)
Z	high impedance (circuit connected to neither ground nor voltage)
H	weak drive, logic one (i.e. voltage behind resistor)
L	weak drive, logic zero
W	weak drive, undefined logic value
-	don't care

These values take priority over each other in the following order: $X > (0 \sim 1) > W > (L \sim H) > Z$.

Arrays and Vectors

```
type intArray is array (15 downto 0) of integer;
type bitArray is array (0 to 7) of bit;
type myMatrix is array (1 to 3, 1 to 3) of std_logic;
subtype myVector4 is std_logic_vector(3 downto 0);
```

2.2.4 Operators

No.	Type	Examples
7	Other Operators	`abs, not` (Negation of bits), `**` (exponentiation)
6	Multipliying Operators	`*, /, mod, rem` (remainder)
5	Unary Operators	+ (identity), - (negation of a numeric type)
4	Addition Operators	`+, -, &` (vector concatenation)
3	Shift Operators	`sll, srl, sla, sra, rol, ror`[1]
2	Relational Operators	`=, /=` (not equal), `<, <=, >, >=`
1	Logical Operators	`and, or, nand, nor, xor, xnor`

[1] - Shift operators ending in "l" are "logical", meaning vacated bits are filled with 0. Shift operators ending in "a" are "arithmetical", meaning vacated bits are filled with the value of the rightmost/leftmost bit. The operators "rol" and "ror" rotate the bits instead of shifting them.

Operators with higher numbers in this table take priority over operators with lower numbers.

2.2.5 Constants and Signals

Constants work as expected in a programming language:

```
constant PI: real := 3.1415;
contant PERIOD: time := 100ns;
type vecType is array (0 to 3) of integer;
contant VEC: vecType := (2,4,-1,7)
```

Signals represent a wire or register. They can be of any data type, can be declared in architectures only.

```
signal sum: std_logic;
signal clk: bit;
signal data: std_logic_vector(0 to 7) := "00X0X011";
signal value: integer range 16 to 31 := 17;
```

Signals assignments are performed **concurrently**, meaning that they are sequentially collected until the process is stopped, and then collectively performed in parallel after al processes are stopped.

Signals can be assigned with either an explicit user-defined time delay ("after 10ns", etc.), or with an implicit small delta delay:

```
sum <= (a xor b) after 2 ns; -- explicit delay
data(1) <= 'x'; -- implicit delay
```

Signal assignments can also include conditionals. This can be done using the when-else condition:

```
clk <= '0' after 5ns when clk = '1' else '1' after 7ns when clk = '0';
a <= "1000" when b = "00"
else "0100" when b = "01"
else "0010" when b= "10"
else "0001" when b = "11";
```

Or using the with-select condition:

```
with b select a <=
"1000" when "00",
"0100" when "01",
"0010" when "10",
"0001" when "11";
```

Neither of the two conditionals may be used inside a process. Within the finished hardware, conditions like this are realized using a multiplexer. Custom multiplexer code would look something like this:

```
entity mux is
port (i3, i2, i1, i0: in bit;
    sel: in bit_vector(1 downto 0);
    otp: out bit);
end;

architecture wSelect of mux is
begin
    with sel select
    otp <= i0 when "00",
    i1 when "01",
    i2 when "10",
    i3 when others;
end;
```

2.2.6 Variables

Variables work like variables in other programming languages. They store temporary values and are only usable in processes, procedures and functions. **Usage of them is not recommended in VHDL for synthesis.** Unlike signal assignments, variables assignments are performed sequentially as they are encountered in the code.

2.2.7 Processes

We've already seen two styles of modelling using VHDL: A Dataflow architecture uses concurrent signal assignment statements, while a structural architecture uses only component instantiation statements. We will now learn a third style: **Behavioural architecture**, which uses **process statements**. A process is simply a set of statements that are executed sequentially-ish:

```
signal clk : std_logic := '0';
clk_gen: process ( )
begin
 clk <= '0';
 wait for 10 ns;
 clk <= '1';
 wait for 10 ns;
end process;
```

VHDL supports four different types of wait statements:

- `wait on` waits until one of the given signals changes (e.g. `wait on a,b,c;`).
- `wait until` waits until the given condition is met (e.g. `wait until (clkEvent and clk = '1')`).
- `wait for` waits for a specified amount of time (e.g. `wait for 25 ns;`).
- `wait` waits indefinitely.

Only simple signal assignments are allowed inside a process. When a simulation starts, each process will be executed at least once. Afterwards, they will loop infinitely. If the process has a *sensitivity list*, a new iteration will occur whenever a signal from the sensitivity list changes:

```
entity DFF is
port (D, clk: in std_logic;
Q: out std_logic);
end DFF;
architecture rtl of DFF is
begin
 p : process(clk) -- sensitivity list
 begin
 if (clk‘event) and (clk=‘1‘) then
  Q <= D;
 end if;
 end process p;
end rtl;
```

Processes with sensitivity lists are equivalent to processes without a sensitivity loop that have `wait on` statements instead:

```
entity DFF is
port (D, clk: in std_logic;
Q: out std_logic);
end DFF;
architecture rtl of DFF is
begin
 p : process
 begin
  if (clk‘event) and (clk=‘1‘) then
   Q <= D;
  end if;
  wait on clk; -- equivalent wait statement
 end process p;
end rtl;
```

Processes are not allowed to have subprocesses. They always loop, and are often used to specify sequential hardware. Everything in VHDL is implicitly part of a "main" process.

2.2.8 Statements

`if`-Statements and `case`-Statements are comparable to `if`-Statements and `switch`-statements in other languages. Both of them can be nested. Conditions in `if`-Statements can be any boolean expression.

```
if a = b then
...
elsif a > b or a > c then
...
else
...
end if;
```

```
case a is
when "01" =>
...
when "10" =>
...
when others =>
null
end case;
```

As seen here, a case where nothing happens can be specified using the `null` Statement.

VHDL also supports `loop`s. Here are two variants of a clock that counts up to 10, incrementing once every 5ns, using a `while` loop and a `for` loop:

```
constant MAX_SIM_TIME : time := 50 ns;
constant PERIOD : time := 10 ns;
...
clk_gen: process (clk)
```

```
begin
 while NOW < MAX_SIM_TIME loop --!
  clk <= not clk ;
  wait for PERIOD/2;
```

```
 end loop;
 wait;
end process clk_gen;

constant MAX_CYCLES : integer := 10;
constant PERIOD : time := 10 ns;
...
clk_gen: process (clk)
variable cnt: integer := 0
begin
 for cnt in 1 to MAX_CYCLES loop --!
  clk <= not clk ;
  wait for PERIOD/2;
 end loop;
 wait;
end process clk_gen;
```

And a third variant using the `exit when`-Statement:

```
constant MAX_CYCLES : integer := 10;
...
clk_gen: process (clk)
variable cnt: integer := 0;
begin
 L1: loop
  clk <= not clk;
  cnt := cnt + 1;
  wait for 5ns;
 exit when cnt > 2*MAX_CYCLES; --!
end loop;
wait;
end process clk_gen;
```

During synthesis, all loops have to be unrolled, meaning that loops with a non-static range are non-synthesisable. It is generally good practice to only use loops in testbenches.

2.2.9 Functions and procedures

Apart from entities and architectures, VHDL also supports functions and procedures, similar to traditional programming languages. A **Function** has a return value and can be used in statements:

```
architecture rtl of example is
 signal test : integer := 0;
 ...

begin
 function b2i(b : bit) return integer is
 begin
  if b = '1' then
   return 1;
  else
   return 0;
  end if;
 end b2i;

 test <= b2i('0');
end;
```

Functions can be overloaded, meaning that there can be Functions with the same name but different type signatures. By default, functions have to be **pure**, meaning they are free of side effects - formally $f(a)$ always returns the same value if a is the same. An impure function can be declared by prepending the function with `impure`. This will let the function gain access to all variables and signals outside of its scope.

A **Procedure** can be seen as a function without a return value. Instead, it has `in, out` or `inout`-signals, similar to an entity:

```
architecture behave of ex_procedure_simple is
 signal r_TEST : std_logic_vector(7 downto 0) := X"42";

-- Purpose: Increments a std_logic_vector by 1
 procedure INCREMENT_SLV (
  signal r_IN : in std_logic_vector(7 downto 0);
  signal r_OUT : out std_logic_vector(7 downto 0)
  ) is
  begin
   r_OUT <= std_logic_vector(unsigned(r_IN) + 1);
  end INCREMENT_SLV;
...
 signal test : std_logic_vector(7 downto 0) := (others => '0');
...
 test_p: process
 begin
  INCREMENT_SLV(test, test);
 end test_p;
```

Because Procedures do not return anything, they can't be used in statements. They can still be used inside of processes, or, if `out` and `inout` parameters are signals, as their own processes.

2.2.10 Synthesisable vs Non-Synthesisable Code

Only a subset of VHDL statements is **synthesisable** (i.e. compilable with the output being hardware). Non-synthesisable statements include time statements, asserts, and dynamic loops.

You can work around many of these restrictions. For example, the following code is non-synthesisable, because it uses a `wait`-Statement:

```
architecture behavior of testbench is
begin
 enable <= '0';
 wait for 100 ns; -- !
 enable <= '1'
end behavior;
```

However, the following code is synthesisable:

```
architecture behavior of realCircuit is
signal cnt : unsigned(3 downto 0) := (others => '0');
begin
 process ( clk )
 begin
  if rising_edge( clk ) then
   if cnt < 10 then -- assume clk  period is 10 ns
    cnt <= cnt + 1;
    enable <= '0';
   else
    cnt <= (others => '0');
    enable <= '1';
   end if;
  end if;
 end process;
end behavior;
```

Many statements in VHDL are technically synthesisable, but are best avoided, generally because they quickly lead to significant, often unexpected, increases in hardware complexity. These include:

- division or multiplication with numbers that aren't powers of 2

- `if rising_edge (clk)` with `else`
- latches

a **latch** is a memory element that is triggered by a changes immediately whenever the input changes:

```
architecture rtl of latch is
...
begin
 process (E, D) begin
  if E = '1' then
   Q <= D;
  end if;
 end process;
end;
```

The big problem with latches is that they leaf to uncertainty in a circuit's timing behavior. You can avoid latches by including a clock and triggering things only on rising edges, and by stating all possibilities in if statements.

2.2.11 Simulation

A simulation works in the following way:

- Initialization:
  - Initialize all signals
  - Set simulation time to 0
  - Execute all processes once
  - Start simulaton cycles
- Simulation cycles:
  - update signals
  - execute processes
  - repeat
- Simulation ends when:
  - No more signal changes are possible
  - A maximum simlulation time has been reached
  - An explicit `wait` is encountered

Values are assigned to signals using a **transaction list**. The list contains entries of the form (s, v, t), meaning "signal s is set to value v at time t". Processes are similarly reactivated using a **process activation list**, with entries of the form (p, t) ("process p resumes at time t").

2.2.12 Delay Modeling

Real components always work on a delay. **Delay of components** can be modeled in VHDL using the `inertial` Keyword:

```
output <= not input after 10 ns;
-- with inertial delay:
output <= reject 5 ns inertial not input after 10 ns;
```

If a signal assignment happens for an amount of time shorter than the signals inertial delay, then the signal doesn't change at all.

There is also the Keyword `transport` to model the **delay of wires**:

```
output <= not input after 10 ns;
-- with transport delay:
output <= transport not input after 10 ns;
```

A signal that uses `transport` delay always gets changed after the specified time.

Chapter 3

Design Space Evaluation

Design Space Evaluation is the process of considering different possible ways to realize a given plan and comparing them based on criteria such as:

- Cost
- Performance
- Power consumption
- Quality

'Cost' can be further split into factors such as:

- Manufacturing cost
- Design cost
- Field support
- Administration
- Design time

While 'Performance' comes down to factors like:

- Clock Frequency / Operations per Second
- Bandwidth
- Quality of service

Note that especially in safety-critical systems, it is preferable to accept a worse average runtime if it leads to a better worst case runtime and more predictability. For example, caching is usually avoided because of its inherent unpredictability.

Within the context of Embedded Systems, common decision points include choosing between:

- ASICs (Application specific integrated circuits)
- Field Programmable Gate Arrays (FPGAs)
- Microprocessors
- Microcontrollers
- Different Memory Architectures
- Different Interfaces (I^2C, SPI, CAN, ...)
- Different possible Sensors & Actuators
- Different possible AD and DA converters
- etc.

Formally, this comes down to a **multiobjective optimisation problem**.

3.1 Power Consumption

Generally, **power** is the most important constraint in Embedded Systems, and thus minimizing power consumption is one of the primary concerns during the design process. Modern processors have a power density of up to $100\frac{W}{cm^2}$!

Minimizing the power consumption leads to less pressure for the power supply and for voltage regulators and a much lower risk of overheating (also meaning less effort needed to introduce cooling). Naturally it also means lower costs.

Low power design techniques include the usage of different components such as **low-power transistors**, which tend to come with drawbacks in speed. It may also involve dynamic power management, i.e. sleep modes (temporarily switching off components that aren't needed). Switching off the clock of a flip-flop specifically is known as **clock gating**. Dynamic power management naturally brings with it the cost of requiring additional logic. One can also use **dynamic voltage and frequency scaling**, where the power supply voltage is lowered when needed and the clock frequency is lowered accordingly.

The power consumption P of a CMOS circuit is

$$P = \alpha C_L V_{DD}^2 f$$

Where:

- α is the *switching activity* or *activity factor*, defined as the probablity that a circuit node changes from logic 0 to logic 1 in any given clock cycle,
- C_L is the *load capacitance*, i.e. the capacitance between the output of a circuit and ground,
- V_{DD} is the supply voltage,
- and f is the clock frequency.

For a more detailed breakdown of this equation, I found *Power Consumption in CMOS Circuits* by Len Luet Ng et al. to be helpful.

The delay of a CMOS circuit is

$$\tau = k \cdot C_L \frac{V_{DD}}{(V_{DD} - V_t)^2}$$

Where k is a contant that depends on the circuit and V_t is the threshold voltage, i.e. the voltage defined as the cutoff between a logical "1" and a logical "0".

The important takeaway from these equations is that

$$P \propto V_{DD}^2, \text{ while } \tau \propto \frac{1}{V_{DD} + \frac{1}{V_{DD}}}.$$

This means that, by decreasing the supply voltage of a circuit, the circuit's power consumption can be decreased quadratically, while the circuit's delay increases roughly linearly (by a pretty generous definition of "roughly").

3.2 Quality Testing

All manufacturing processes are inherently prone to defects. Naturally, defective parts should not be delivered to customers. Therefore, a need arises for **test processes** to distinguish good components from faulty ones. Testing can incur significant costs (the slides state an unsourced figure of "up to 60%"). Naturally, these still cannot identify 100% of defective parts. There are many different approaches of testing for different types of components and the systems they make up. Parts are also susceptible to aging, which leads to parts which were previously defect-free to become defective over time. Lastly, parts may be susceptible to external effects like changing temperatures, noise or radiation.

The fraction of defective delivered parts over total delivered parts is known as the **Defect Level (DL)** or as the number of **test escapes**. Naturally, a high defect level can lead to a loss of reputation and eventually to legal penalties. The fraction of defective-free part over all parts is known as the **Yield**. A higher lead naturally leads to lower testing costs and fewer test escapes. Finally, as already defined in the introduction, **Reliability** measures the probability that a part will work correctly over a given

time, and is incredibly important especially for safety-critical systems. Ideally, a system should be able to undergo **graceful degradation**, meaning that as the system degrades, performance merely decreases gradually instead of suddenly catastrophically failing.

Generally, a specification will include required bounds on quality parameters. For example, the IEC 61508 standard defines four safety integrity levels (SIL 1 through SIL 4). By this definition, an SIL 1 system must have a failure probability per hour lower than $\frac{1}{10^5}$, while an SIL 4 system needs to meet a much stricter requirement of at most $\frac{1}{10^8}$ failures per hour.

3.3 Pareto Frontier

A solution of the multiobjective optimisation problem posed by design space exploration is given by a vector $V = (v_1, \ldots, v_n)$ of parameters describing the performance, costs etc. associated with a possible realisation of a specification. We define these parameters in such a way that higher values are always better, which means that, for parameters describing a cost, we need to either take the negative or the inverse of the cost value. Naturally, these parameters are generally only estimates, since determining the exact values would have to involve production of every single possible solution at a non-trivial scale.

A natural way of discarding inferior solutions is given by **pareto superiority**. We say that a solution A is pareto-superior to a solution B iff we have both $\forall j(a_j \geq b_j)$ (A is at least as good as B in every aspect) and $\exists i(a_i > a_j)$ (There is at least one aspect of A that is better than B). It should be intuitively obvious that if A is superior to B in every single criterium considered, then A is preferable over B. The set of all solutions not dominated by any other solution is known as the **Pareto Frontier**.

Chapter 4

Hardware

4.1 Microprocessors vs Microcontrollers

A Microprocessor consists of **only a computing unit**, which then needs to be complemented by memory, I/O interfaces etc. Microprocessors are fast, but often expensive.

Meanwhile, a Microcontroller is a **System-on-chip (SOC)**. It already contains memory, timers, voltage converters, and at least one I/O interface (generally even multiple). They are slower than microprocessors, but are cheap and energy efficient.

4.2 Memory

Technological improvements to computer memory size and access times are currently made at a much slower pace than improvements to processor cycle times. This **Memory Wall** is a key challenge in the development of new AI models and, more generally, in most high performance computing applications.

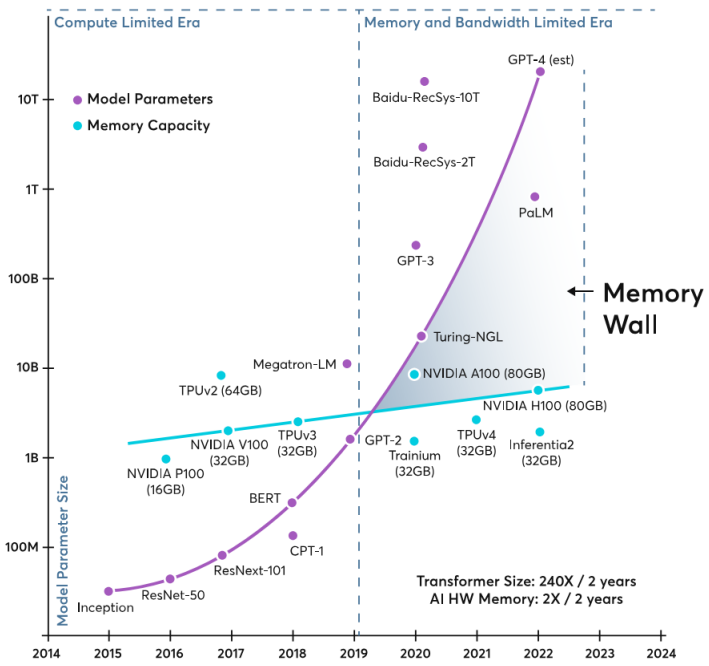

Figure 4.1: The memory wall in action (Source: ayarlabs)

The **Principle of Locality** is the Observation that programs tend to use data with addresses near those they have already used recently. It can be further split into **Temporal Locality**, which means that

recently referenced items are likely to be referenced again soon, and **Spatial Locality**, meaning that items with nearby addresses are often referenced around the same time. Modern memory architectures exploit these properties through **caching**, where several types of memory exist within a system, and items are placed in faster memory or slower memory depending on how likely it is that they will be needed in the near future.

The most basic types of memory, ordered by speed, are:

1. Registers
2. Cache, Scratchpad Memory (SPM)
3. Main Memory (RAM)
4. Secondary Storage (SSD, HDD)

On its own, a processor tends to only include registers and cache or SPM. The registers of a processor could technically be considered a particularly fast type of cache. Fast caches have to be placed near the processing unit - the increases in signal travel time and especially in ambient noise that come with longer wires are actually relevant factors in determining memory speed.

4.2.1 Cache Design

Some key questions of cache design are:

- Which memory blocks do we place into which cache blocks?
- How do we detect if and where a block is in the cache?
- Which cache block is replaced after a **cache miss**, i.e. when an item is needed that isn't in the cache yet?
- What happens if we write new things into memory?
- How do we deal with multi-core architectures?

A **direct mapping cache** is a cache where the adress of any item in cache is simply the item's adress in main memory modulo the number of cache blocks. A cache where any item can occupy any block is known as an **associative cache**. To use an associative cache, one needs to specify what happens when the cache is full - a predefined **cache replacement strategy** is needed. Common cache replacement strategies include:

- FIFO (first in, first out)
- LRU (least recently used)
- random selection

Interestingly, random selection is provably optimal in cases where the order that items are needed in is provided by an *oblivious adversary* (See lecture "Algorithm Theory" for more details). However, for real-time embedded systems, predictability is key, which means that non-deterministic cache algorithms are used. For a worst case execution time analysis, one has to assume that almost every access to the cache leads to a cache miss.

4.2.2 Scratch Pad Memory

A scratch pad memory architecture maps small physical memories directly into the CPU's address space. Frequently used variables and instructions can thus be allocated to the scratch pad memory **at compile time**. This leads to **fast and predictable behavior and lower energy consumption**, especially compared to associative cache architectures. Some architectures contain both caches and SPM, leading to a hierachical memory structure:

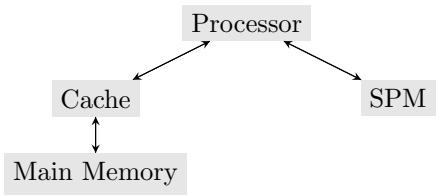

Figure 4.2: A memory architecture that combines caching with SPM

4.2.3 I/O Access

The two most basic ways of handling I/O are **memory-mapped I/O**, where components get mapped directly into controller address space, and **port-mapped I/O**, also known as **programmed I/O**, where the CPU transfers bytes using a special instruction. However, both of those have the major disadvantage of keeping the CPU busy, with updates requiring polling.

Most microprocessors and -controllers include a component that allows **DMA (Direct Memory Access)**. These are dedicated components that allow the CPU to do other work or sleep during a data transfer. However, the CPU still needs to initiate the transfer on its own. Typically, the CPU then receives an interrupt from the DMA once the transfer is done.

Usage of DMAs brings with it the problem of **cache coherency**. Imagine the following scenario:

- The CPU accesses location X in memory and stores the current value of X in cache
- Subsequent operations on X update the cached copy of X, not the external memory version of X (since updating the external version would mean dealing with slow memory speeds, defeating the whole purpose of a cache.)
- Whenever an outside device now tries to access X, that device will **recieve the old value of X.** Similary, if a device updates X in main memory, then **the CPU will be working with an outdated version of X.**

A simple cache consistency protocol consists of having both the CPU and outside devices broadcast an "invalidate" flag on every write to memory, signaling to each other that a value has become outdated. Alternatives include having a **write-through** or **write-around** cache instead of a write-back cache, meaning that everything that is written to cache is copied to main memory immediately. This sacrifices speed on writes (but still leaves the benefits caches have for reads).

4.2.4 Interrupts

Interrupts allow the CPU to be notified about important events. An interrupt generally gets processed as follows:

1. an interrupt request is triggered
2. the CPU stops its current exectution flow and saves its current state to memory
3. the CPU executes the interrupt service routine associated with the specific interrupt request
4. the CPU reloads its state and continues normal operation

Modern CPUs feature at least one programmable interrupt controller (PIC) that manages interrups along several lines (to allow several devices to send interrupts) by assigning priorities and delaying or outright ignoring low-important interrupts.

4.3 Communication

In order to communicate with each other, digital devices need to send **modulated signals**. Signal modulation can be split into analog (i.e. continuous) methods and digital (i.e. discrete) methods.

4.3.1 Pulse Width Modulation

Pulse Width Modulation **(PWM)** is a digital modulation method that is very common in sensor and actuator interfaces. It represents a signal as a rectangular wave, where each rate corresponds to a certain linear speed at which the resulting analog symbol should change.

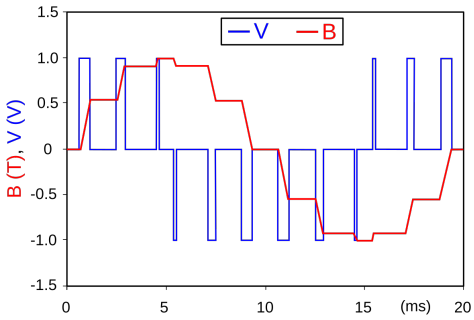

Figure 4.3: A digital signal representing a sine-like curve using 3-Level PWM. (Source: Wikipedia)

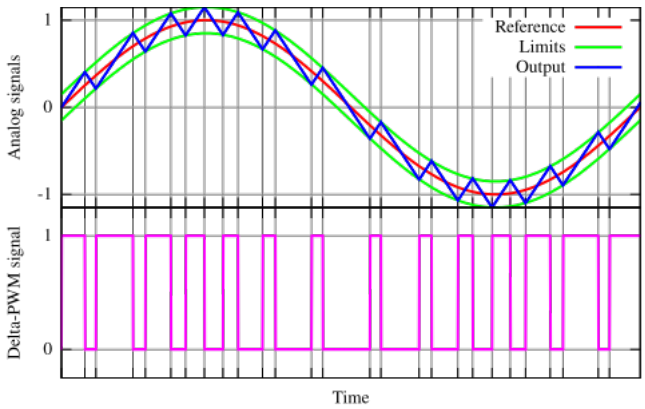

Figure 4.4: A similar curve represented using 2-Level Delta PWM. (Source: Wikipedia)

The **Duty Cycle D** of a 2-Level PWN Signal describes the proportion of "on" time during a given interval.

The most common way of realizing 2-Level PWM is **intersective PWM**, where the PWM output is switched every time a sawtooth wave with the desired growth rate intersects with the input waveform. Another method is **Delta PWM**, where '0' level represents a negative slope and the output is switched every time the output signal exceeds a given upper or lower bound around the input signal (see 4.8).

4.3.2 Bus Standards

A large number of different bus standards exist, all with their own capabilities, benefits and drawbacks. Protocols where data is transferred based on a common clock are called **synchronous**, while ones without a clock are called **asynchronous**. An example of an asynchronous protocol would be UART (universal asynchronous receiver-transmitter), which wasn't covered in detail by the lecture. The protocols covered by the lecture are:

I^2C:

- stands for "inter-integrated circuit", sometimes alternatively abbreviated to I2C or IIC

- main-sub architecture[1] - a controller ('main') generates a clock signal an sends out requests for data, which are then replied to by the targets ('sub').
- one single data line (`SDA`) and clock line (`SCL`) shared by all connected devices
- slow and high-power but cheap

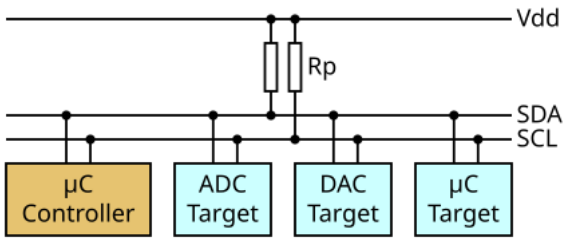

Figure 4.5: The I2C bus architecture

SPI:

- main-sub architecture. unlike I2C, SPI only supports a single controller.
- two seperate data buses: `MOSI` (main out sub in) and `MISO` (main in sub out). This has the major advantage of allowing data to be transmitted and recieved simulataneously (making SPI a **full duplex protocol**).
- additional lines $\overline{\text{SS}}$ (sub select)
- fast and low-power at the cost of more required space and more complex logic

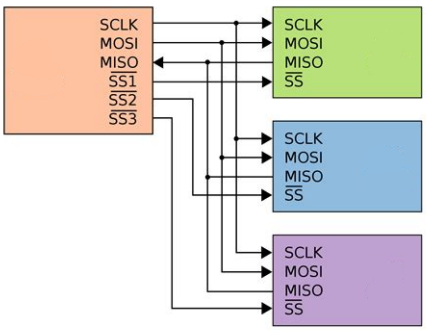

Figure 4.6: The SPI bus architecture (controller on the left, targets on the right)

Name	synchronicity	main-sub?	full duplex?	speed	power	cost
I2C	synchronous	yes	no	low	high	low
SPI	synchronous	yes	yes	high	medium	high
UART	asynchronous	no	yes	lowest	lowest	lowest

Figure 4.7: Comparison of common bus architectures

UART is often considered somewhat outdated and is being gradually replaced by more modern architectures like I2C, SPI, or USB.

[1] Many places, including the original slides presented by Prof. Amft, still use the terminology "master-slave". Since many people, myself included, find this terminology inappropriate, I will be using "main-sub", which has established itself as a common alternative that still works with any relevant acronyms.

4.4 Debugging

The general methods of debugging an embedded system, ordered by increasing realism and increasing effort, are:

1. Emulation (Imitate behavior of the target without necessarily imitating the internal state of the target)
2. Simulation (Evaluate with an accurate model of the target system)
3. On-Targets (Evaluate on actual target hardware within a lab or simulation environment)
4. On-Field (Evaluate on actual target hardware within in the targeted application environment)

Testing can prove the presence of errors, but in order to prove the absence of errors, more complicated methods of Program and Hardware Verification are needed, which are not covered in this course.

4.4.1 JTAG

JTAG is a standard for testing and debugging integrated circuits, introduced in 1985. It introduces a **Test Access Port (TAP)** to integrated circuits, which allow easy access to internal logic of ICs and PCBs via connection of TAPs to an external testing device.

JTAG specifies a serial protocol with four to five lines: TDI (Test Data In), TDO (Test Data Out), TCK (Test Clock), TMS (Test Mode Select), TRST (Test Reset, Optional). It has no maximum speed, but is typically used with a speed of 10-100 MHz. Using JTAG, one can recover "dead" boards where

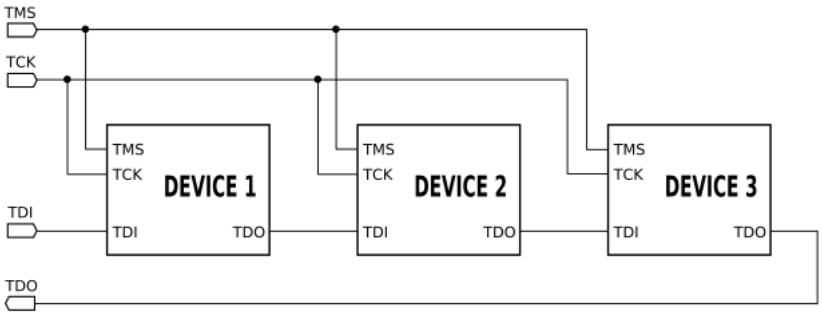

Figure 4.8: The structure of a daisychained JTAG interface. (Source: Wikipedia)

a microcontroller doesn't boot. No specific software needs to be installed on the target. The TAP controller itself implements a stateful protocol to access the test registers. JTAG is available on most modern embedded devices. It is also a common way of hacking devices such as video game consoles, if no sufficient encryption is present.

4.5 Signal Processing

Signals are differentiated by whether they are **time-continuous or time discrete** (i.e. whether a signal is always present or whether a set of seperate short pulses is sent with some delay inbetween) and by whether the possible amplitues of a signal are a limited set of discrete (digital) values or a continuous (analog) range ofvalues. ADC (Analog-Digital-Conversion) always includes discretization in both time and amplitude.

4.5.1 Direct Conversion

4.9 shows a realization of a basic ADC that encodes an n-bit output using $2^n - 1$ comparators and 2^n resistors. A simple realization like this inherently allows for very high conversion frequencies ($>$ 1GHz).

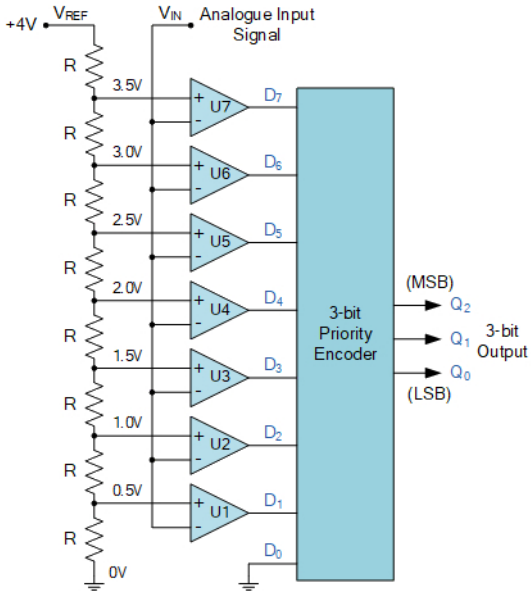

Figure 4.9: A basic ADC (Source: electronics-tutorials.ws)

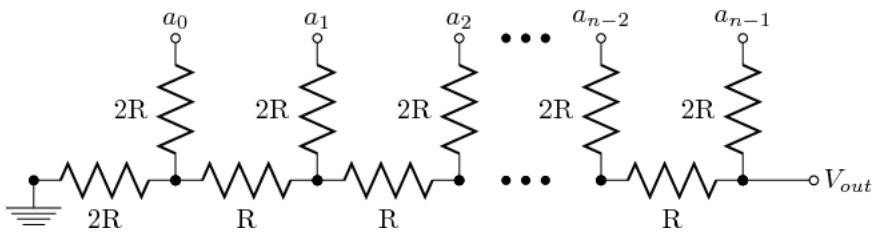

Figure 4.10: A basic DAC (Source: Wikipedia

4.10 shows that a DAC can be realized in an even simpler way. The given resistor ladder encodes an n-bit input using $2n$ resistors. a_0 represents the least significant bit while a_{n-1} represents the most significant bit.

Basic ADC or DAC designs like the ones just presented have the disadvantage of requiring the resistances of all of the present resistors to be sufficiently accurate.

4.5.2 Sequential Conversion

An n-bit succesive approximation converter is an ADC that works as follows:

1. Start by assuming the digital voltage is 0.
2. Flip the most significant bit, and generate the associated voltage using an n-bit DAC. If this voltage is higher than the input voltage, flip the bit back. If not, keep it.
3. Repeat by proceeding the same way with the nextmost significant bit.

Intuitively, using such a circuit is equivalent to using binary search to find the closest possible digital voltage.

A common way of stabilizing the in- and outputs of many of these converters is to use a **Sample & Hold Circuit**, which uses a capacitor to charge up to an input voltage as fast as possible, and then maintain that voltage for a specified amount of time. A simplified schematic of such a circuit can be seen in 4.11.

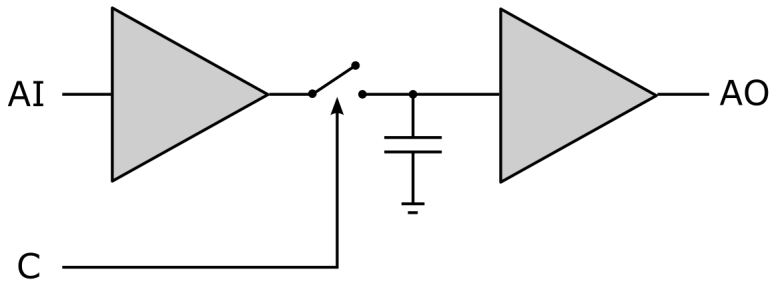

Figure 4.11: A simplified schematic of an S&H circuit (Source: Wikipedia)

4.5.3 Signal Theory

Aliasing is a process where high-frequency information in an image is mistakenly interpreted as a lower frequency. For a given sampling rate f_s, two frequencies f and f' are aliases of each other, meaning that f' will be mistaken for f, if $f' = f + k \cdot f_s$ for some integer k.

The **Nyquist–Shannon sampling theorem** [2] states that aliasing occurs when the sampling frequency f_s is lower than the maximum input frequency f_b multiplied by 2, and therefore a signal must be sampled at $f_s \geq 2f_b$ to get an accurate result[3].

Baseband sampling occurs when all signals of interest lie within the **Nyquist Bandwidth**, i.e. they are all smaller than $\frac{1}{2}f_s$. In this case, we need to filter out aliases at higher frequencies.

Undersampling occurs when images between $\frac{1}{2}f_s$ and f_s are sampled. In the case of undersampling, the first Nyquist zone (the interval $[0, \frac{1}{2}f_s]$) contains aliases of the same signals in frequency-reversed order. This means that, if there is also a known lower bound f_a on the input frequencies, then we can clearly identify the high-frequency signals through their aliases in the first Nyquist zone, meaning that **if we want to measure a signal of bandwidth Δf, then it is enough to sample at $f_s \geq 2\Delta f$.** An equivalent statement is that **if we want to sample a signal whose frequencies are known to lie between f_a and f_b, then it is enough to sample at $f_s \geq 2(f_b - f_a)$.**

[2] Actually first discovered by by E. T. Whittaker in 1915, and thus also known as the *Whittaker–Shannon sampling theorem, Whittaker–Shannon theorem*, or *Whittaker–Nyquist–Shannon theorem.* It is sometimes also simply called the *cardinal theorem of interpolation.*

[3] Shannon's original version only stated *"If a function $x(t)$ contains no frequencies higher than B hertz, then it can be completely determined from its ordinates at a sequence of points spaced less than $\frac{1}{2B}$ seconds apart."*

Chapter 5

Software

A big software chapter was teased all throughout the lecture but besides a non-exam-relevant chapter on the very very basic basics of embedded AI it was basically skipped :^)

Appendix A

Sources

The content of these notes primarily comes from the slides provided by Prof. Amft and Lars Häusler.

Additional sources include Wikipedia for theoretical topics and vhdlwhiz.com, vhdl-online.de and sigasi.com for VHDL.

A tool of dubious quality that was nevertheless used frequently throughout the lecture for playing around with VHDL was edaplayground.com.

Most of the explanations of petri nets (and state charts) given by the lecture was pretty awful, I found *"Free Choice Petri Nets"* by Jörg Desel and Javier Esparza to be a helpful resource in properly explaining properties like liveness or deadlock-freeness. Most of the other content of the lecture was very badly presented too, but at least for those it was either easy to deduce correct definitions from the unclear/informal/wrong definitions given on the slides. The signal processing theory chapter was also really really bad though, but thankfully, unlike for petri nets, information on signal processing is readily available on wikipedia. =w=