LECTURE NOTES

Introduction to Embedded Systems

Winter Semester 24/25

Emma Bach

Lecture by
Prof. Dr. Oliver AMFT

February 16, 2025

Table of contents

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.
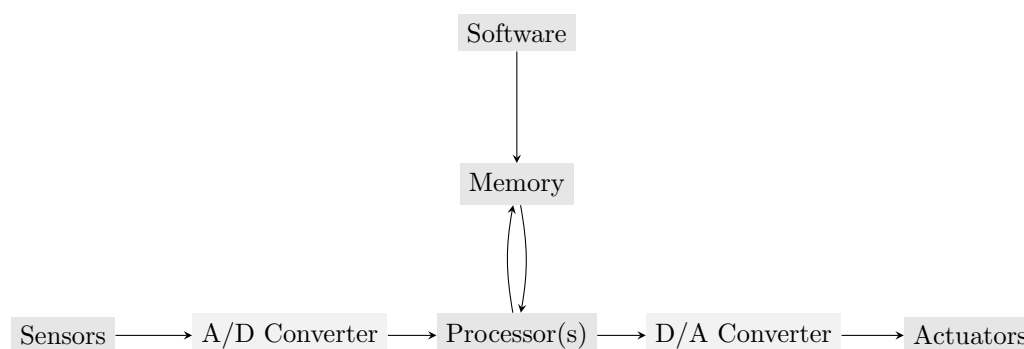

Figure 1.1: The structure of a typical embedded system.

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. 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.

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.

2.1   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.

2.1.1   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.

2.1.2   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 archi-
tecture 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;
```

2.1.3   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, $\ldots$, 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);
```

2.1.4   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.

2.1.5   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;
```

2.1.6 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.

2.1.7 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.

2.1.8   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                              case a is
...                                        when "01" =>
elsif a > b or a > c then                  ...
...                                        when "10" =>
else                                       ...
...                                        when others =>
end if;                                    null
                                           end case;
```

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

VHDL also supports `loops`. 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;     begin
constant PERIOD : time := 10 ns;            while NOW < MAX_SIM_TIME loop --!
...                                          clk <= not clk ;
clk_gen: process (clk)                       wait for PERIOD/2;
```

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

2.1.9   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.

2.1.10   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.

2.1.11   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").

2.1.12   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**.

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.

Chapter 4

Hardware

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.