Embedded System Design

Lecture Notes for IL2206 Embedded Systems and IL2212 Embedded Software (Study Year 2021/22)

Ingo Sander

January 18, 2022

Contents

1. Introduction

1.1. Embedded Systems

Embedded systems are everywhere and control vital functions in our daily life. An embedded system can be defined as a system that uses a computer to perform a specific function, where the system is neither used nor perceived as a computer (Edwards et al., 1997). Embedded systems are key components in many safety-critical application domains, such as railway, automotive, avionics, or medical systems, but are also part of less critical systems, such as smartphones or consumer electronics. Embedded system designers have often a huge responsibility, since faults in safety-critical embedded systems can lead to disastrous consequences. Since embedded systems include more and more functionality the design process becomes increasingly complex. A disciplined design methodology is needed to design future safety-critical embedded systems.

Software Disasters

- Between 1985-87 at least six known accidents have caused deaths and serious injuries of cancer patients due to overdoses of radiation resulting from a race condition between concurrent tasks in the Therac-25 software (1985-87). For further information see (Leveson and Turner, 1993).

- The first test flight of the European Ariane 5 rocket in June 1996 ended with automatic self-destruction 37 seconds after launch, because of a malfunction caused by data conversion in the control software. A 64-bit floating-point number was converted to a signed 16-bit integer number. However, the 64-bit floating-point number was too large to be represented by a signed 16-bit integer number, and resulted in a processor trap. An interesting aspect was that the software subsystem that caused the Ariane crash was originally written for the Ariane 4 and reused in Ariane 5. It turned out that this part of the software was not needed in Ariane 5 because Ariane 5 had a different preparation sequence than Ariane 4. For further information see (Lions, 1996).

Embedded systems are part of different applications, which also have different characteristics. Nevertheless, there are a number of typical characteristics that are shared by many embedded systems. An embedded system

- is usually designed for one single task – its *functionality will never change*;

- is often a mass product – *design cost* is critical;

- interacts with the environment at the speed of the environment – many embedded systems are safety-critical systems and have to fulfil hard *real-time* and *safety* requirements;
- is often a hand-held device – *power-efficiency* is critical;
- is often a consumer products – *time-to-market* is critical.

Due to these special characteristics, the design process for embedded systems is also very different from general purpose programming.

- Embedded systems can be highly optimised.
- All unneeded features are a disadvantage (cost, power).
- The design process must
  - be cost-efficient;
  - ensure the correct functionality and timing of the implementation;
  - be fast to ensure a short time-to-market.

Embedded systems interact with the physical world. In order to cope with this situation and to be able to produce efficient embedded systems, the platforms for embedded systems very often consist of heterogeneous components:

- processing units of different kind (general-purpose Central Processing Unit (CPU), digital signal processors, hardware accelerators);
- memory components of different kind (flash-memory, Static Random Access Memory (SRAM), Dynamic Random Access Memory (DRAM));
- different Input/Output (I/O) units to communicate to the environment;
- Real-Time Operating System (RTOS), hypervisors, or schedulers implemented in software and running on processors.

Due to the advances in semiconductor technology, more and more functionality and components is integrated into a single chip. As a consequence embedded system platforms include an increasing number of processors and components, leading to complex and powerful multiprocessor platforms. There are two main communication paradigms that are used for the communication between processing elements, which are discussed in more detail in Section 2.4:

- shared memory communication, as used in shared memory multiprocessors (Figure 1.1); and
- message-passing communication, as used in network-on-chips (Figure 1.2).

Also, embedded systems can be composed of several physically distributed platforms (nodes), which are connected by means of a network. The communication over the network will then in general use a form of message passing between the different nodes of the network.

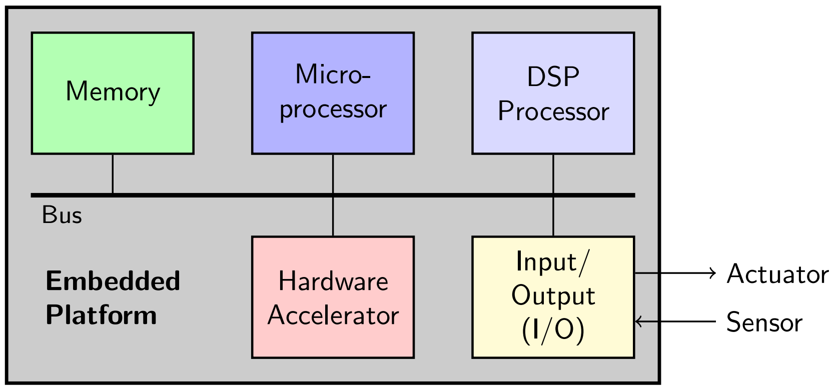


Figure 1.1.: In a shared memory multiprocessor, communication between processors is done via a shared memory that can be accessed by all processors.

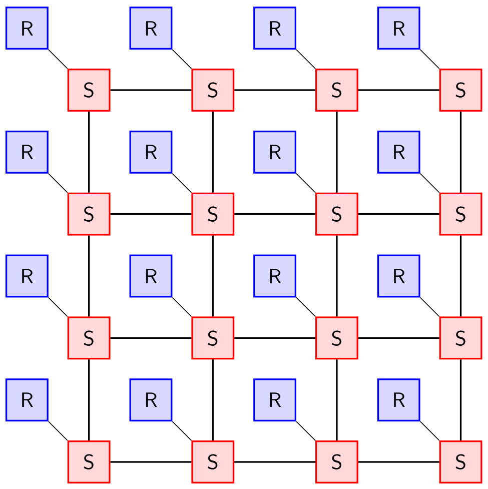


Figure 1.2.: In a network-on-chip (NoC), processor nodes, also called resources (R) or tiles, communicate via a packet-switched network. A processor node consists of a processing units, memories, and a network interface.

1.2. Cyber-Physical Systems

The term Cyber-Physical Systems (CPS) has recently been used to describe a system that integrates computation with physical processes. Many embedded systems are parts of cyber-physical systems, because they are often used to control physical systems. Often, the terms embedded system and CPS are used interchangeably. However, the term cyber-physical systems in general focuses more on the connection between the physical and the computational part, where for instance the network link plays an important role. The term embedded system is mainly used, when the focus is on the computation part.

A successful design of these systems requires to understand the interaction between the physical components and the computational components. This is a big challenge, because different disciplines are required to develop heterogeneous embedded systems or

CPSs, like analogue and digital hardware design, software design, design of the physical system, control theory and signal processing. The different competences are illustrated in Table 1.1.

Table 1.1.: The design of heterogeneous embedded systems or CPSs is very challenging, because many different competences are required.

	Analogue Hardware	**Digital Hardware**	**Software Design**	**Physical Design**
Background	Analogue electronics	Digital electronics	Computer science	Mechanical engineering
Components	Transistors, resistors, capacitances	logical gates, registers, ALU, FPGA, ASIC	Tasks, communication mechanisms, operating systems	Mechanical systems
Models	Continuous time models	Discrete time	Typically untimed	Continuous time
Mathematics	Differential equations	Discrete mathematics	Discrete mathematics	Differential equations
Description	SPICE models	Hardware description languages	Programming languages	Modelling languages like Matlab/Simulink

As can be seen from Table 1.1, the design of heterogeneous embedded or CPSs is especially difficult. Engineers are in general specialised in one discipline, because education is divided into different subjects. In particular interfaces are critical, and many projects fail in the integration phase, because there is no common understanding between the different domains.

1.3. Embedded System Design Process

The task of the embedded system design process it to convert a specification on an abstract level with detailed system requirements into an efficient implementation that not only fulfils the requirements, and also can be produced at a competitive cost. This is a very challenging task, because in the beginning of the design process the requirements imposed on the system have to be fully understood. Then the system's functionality has to be specified at a high level of abstraction, where the main objective is to capture the overall functionality of the system. At this stage low-level details are not only not important yet, and would even be counter-productive. However, in order to have an efficient implementation of the system, the understanding of low-level details becomes very important later in the design process to be able to use advanced features of the target architecture in the most efficient way.

Figure 1.3 illustrates the design process. The design specification is written at a high level of abstraction. Many possible implementations comply with the initial design specification. Together they comprise the *design space*. During the design process the initial specification model is refined. The designer takes design decisions and the design space is reduced. Finally the design process yields one single implementation. The implementation does not need to be optimal, but needs to fulfil the design requirements (functionality, power, speed, area, reliability, costs,. . .).

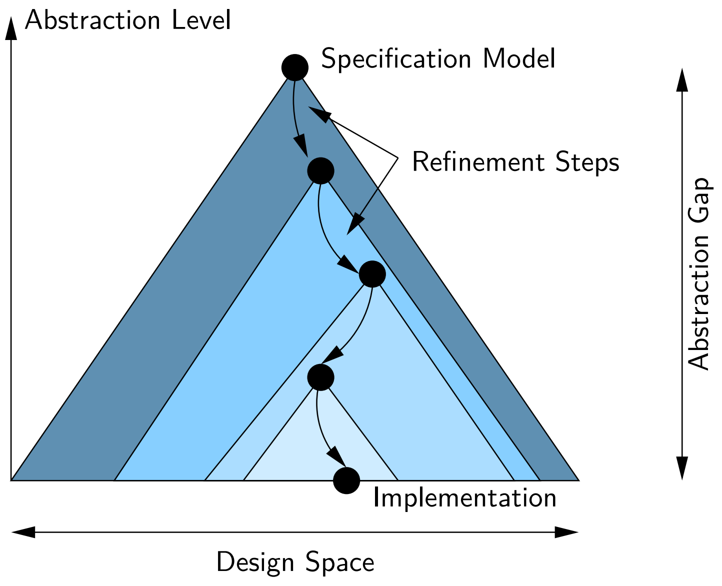


Figure 1.3.: The design process requires to express the system at different levels of abstraction. For each level a new model of a system is required.

Current practice in industry to validate the correctness of an implementation is to heavily rely on simulation and test, which results in very high and increasing validation costs. For safety-critical systems, the ability to demonstrate correctness is of particular importance also due to certification requirements. The increasing complexity of future designs imposes a huge challenge for future design methodologies, since it will be more and more difficult to demonstrate the correctness of a system only by simulation or test methods.

In order to be able to cope with the increasing level of complexity, future design methodologies have to start at a high level of abstraction and need to provide a clear path to the implementation, where tools support design decisions at different levels of abstraction and also enable the automation of refinement steps[1]. This requires to base the whole design process on a formal base, so that formal models with a well-defined semantics are used at the different levels of the design flow. This includes not only the high-level specification model, but all intermediate models and also the model of the platform architecture, which also has to provide well-defined quality-of-service guarantees to be able to generate implementations where worst case performance numbers can be guaranteed.

Obviously this is not an easy task, but already more than twenty years ago Edwards et al. (1997) suggested to manage the complexity and heterogeneity of the design process for system-on-chip applications in such a way, that the design approach should be based on one or more formal models to describe the behaviour of the system at a high

[1] A good example for such a tool are compilers or hardware synthesis tools, which convert a model in form of a program or hardware description into an implementation in form of software binaries or net-list of low-level hardware components.

level of abstraction, before a decision on its composition in hardware and software is taken. The final implementation of the system should then be made by using automatic synthesis from this high level of abstraction to ensure implementations that are *correct-by-construction*. Validation through simulation or verification should be done at the higher level of abstractions.

1.4. The Dream: Correct-by-Construction Design

The course IL2206 Embedded Systems and its follow-up course IL2212 Embedded Software will have as main theme the current problems of industrial design flows, and investigate and discuss prerequisites and possible methods to achieve a *correct-by-construction design flow* for safety-critical embedded real-time real-time systems, which would drastically reduce validation costs. To achieve this goal, existing design techniques and platform architectures have to be well-understood. From this understanding, conclusions can then be drawn for new approaches and techniques.

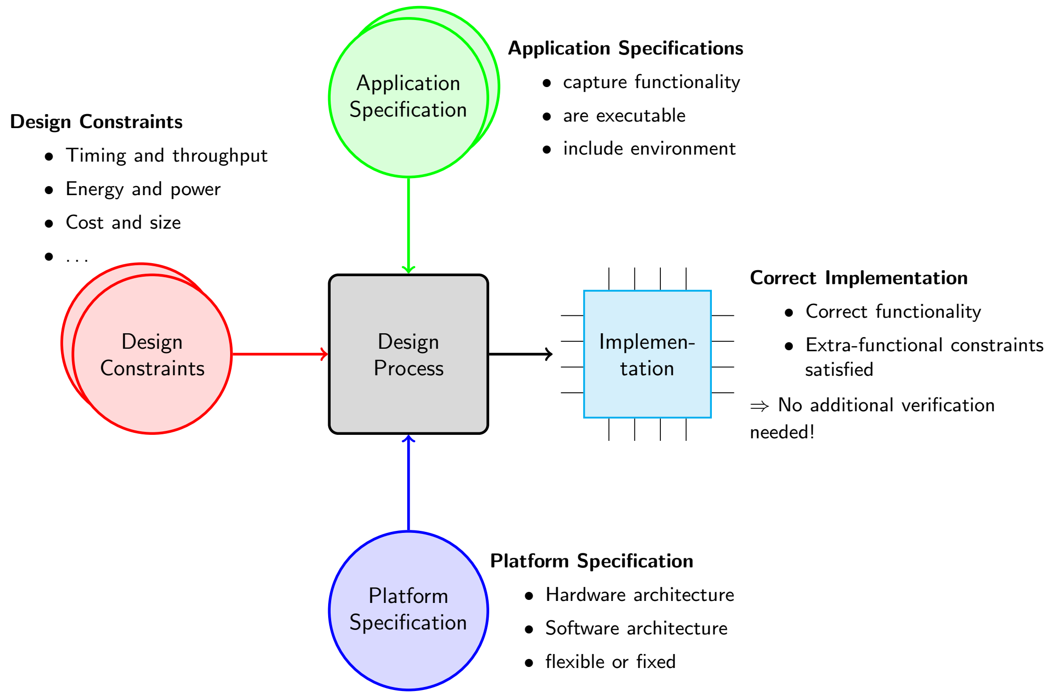


Figure 1.4.: The objective of a correct-by-construction design flow. Is it possible to create an automated design flow that generates an implementation that is correct with respect to its requirements?

Figure 1.4 illustrates the challenging task of correct-by-construction design. A set of applications shall be implemented on a shared platform. Each application has its own set of design constraints, which all have to be satisfied in the final implementation. Ideally, the applications are specified by an executable functional model, which helps the designer to validate the functionality also by the simulation of the high-level model.

A successful correct-by-construction design process would not only yield an implementation, where the functionality of each application is implemented correctly, and where all individual design constraints are satisfied, but also enables to prove the correctness of the final implementation without additional verification and validation. A complete correct-by-construction design flow will be very difficult to achieve, but already a partial implementation of such a flow would improve the current situation, since it significantly decreases the need for system verification.

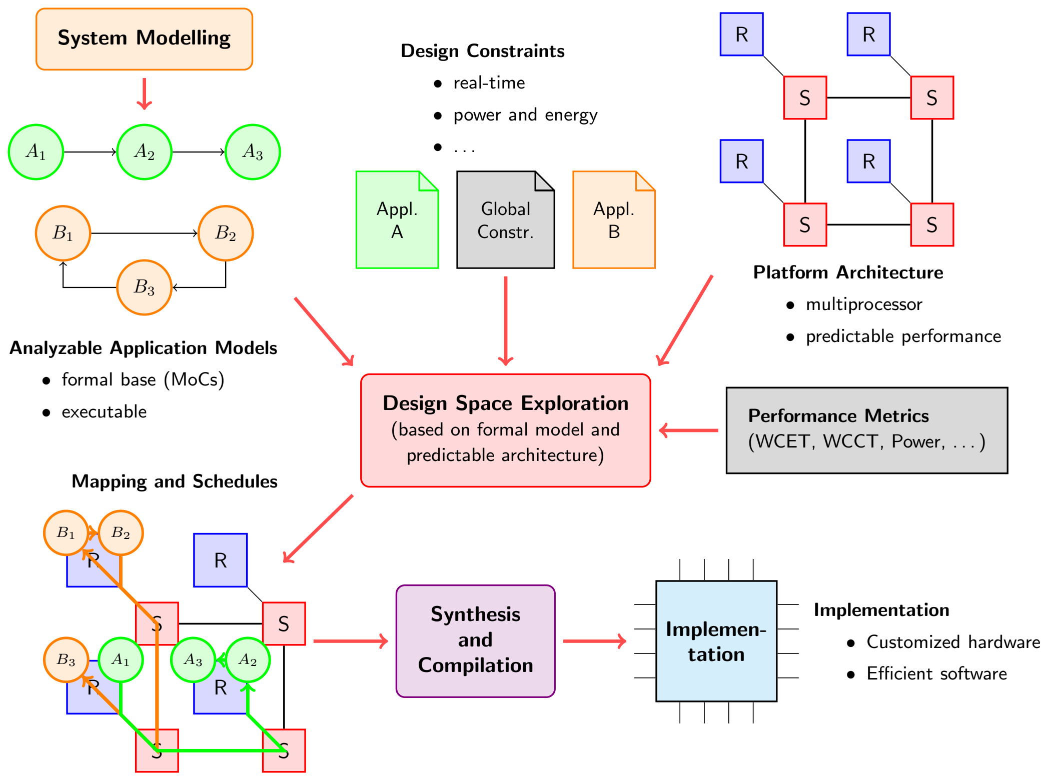


Figure 1.5.: Sketch of a design flow that aims for correct-by-construction. The different parts of the design flow will be further discussed and investigated in the course.

The principal design flow of Figure 1.5 will be used to introduce and discuss the different phases and challenges of a possible correct-by-construction design flow. The design process starts with the *system modelling* phase, where the functionality of each application is modelled. The next phase is the *design space exploration* (Pimentel, 2017), which aims at finding an efficient implementation for the set of applications on the shared target platform, so that each application satisfies its *individual design constraints* and also the *global design constraints* are satisfied. A requirement for an accurate design space exploration is that the properties and provided quality of service guarantees of the *platform architecture* are well-specified. In order to be able to give guarantees on the performance of an application, the target platform needs to be predictable, which means that a guaranteed worst case performance can be given. Examples for predictable platforms developed in academia are the CompSOC- (Hansson et al., 2009) and PRET-architectures (Liu et al., 2012). In particular in the area of safety-critical systems there exist several industrial approaches, like the integrated modular avionics (IMA) architecture (Watkins, 2006), which supports the design of real-time computer network airborne systems, where applications of different criticality share the same distributed platform. The result of the design space exploration is a mapping of application processes and communication links to platform resources together with schedules for computation and communication. This forms the entry point to the final part of the design flow, the *synthesis and compilation* phase, where the final *implementation* is generated.

The phases in the design flow illustrated in Figure 1.5 also appear in existing industrial design flows. The challenge is to be able to automate the design flow in such a way that guarantees for the final implementations can be given. This requires a careful selection of the underlying formal base, which in this course is established by basing the modelling technique on the theory of models of computation and choosing predictable architectures as platform targets, so that performance guarantees can be given.

1.5. Course Overview

1.5.1. IL2206 Embedded Systems and IL2212 Embedded Software

The main topics of the course IL2206 Embedded Systems are the embedded computing platform (Chapter 2 and Appendix C), the introduction to real-time systems and its software support (Chapter 3 until Section 3.5 and Appendix D), and hardware/software co-design (Chapter 4). Also, IL2206 Embedded Systems focuses on single processor systems.

The course IL2212 Embedded Software targets the design of multiprocessor real-time systems. The main topics are the high-level modelling of embedded systems (Chapter 5), more advanced techniques for real-time system design (Chapter 3, from Section 3.6 until the end of the chapter), design exploration and code generation from high-level models (Chapter 6).

The lecture notes will very likely be updated during the courses IL2206 Embedded Systems and IL2212 Embedded Software. The course web page in Canvas will clearly state, which topics are part of each individual course.

1.5.2. Important Questions for Embedded System Design

The lecture notes will cover different phases of the design flow and aim to give a good understanding how the embedded system design process can be automated. The lecture notes will centre around the following topics and questions. Students should have these questions in mind, even when they only take the IL2206 Embedded Systems, but not the IL2212 Embedded Software course.

System Modelling How can we model systems at a high level of abstraction and still keep a clear path to the implementation? Which models are more suitable? What are characteristics for good models? How can the application models be simulated? Can the modelling technique support the design of heterogeneous systems?

Platform Architecture What is a suitable target platform for a correct-by-construction design flow? How can such a platform support the design process? Can we establish a link between the application models and the target platform? The target platform will consist of hardware and software, e.g. operating systems, how can they interact?

Design Space Exploration and Performance Analysis How can we explore the enormous design space? Which performance analysis techniques should be used to conduct an accurate and efficient design space exploration? Can analytical and simulation-based techniques be combined to improve the design space exploration? Do we have access to all data to conduct an accurate design space exploration?

System Synthesis How can we generate the hardware and software for a correct implementation? Can existing software compilers and hardware synthesis tools be integrated into the design flow? How efficient is the final implementation?

Correct-by-Construction Design To what extent is a correct-by-construction design feasible? Which correctness guarantees can be given?

The objective is to give the students a good foundation to answer these challenging questions. Thus, the courses aim to give the students the prerequisites in form of the necessary knowledge and theory to get a good understanding on the design of electronic real-time systems. In order to have a clear scope, the courses focuses on real-time constraints as the main design constraint, although other extra-functional properties like for instance power consumption are also of large importance.

1.5.3. Course Material

The lecture notes are considered to be the main reading material for the course. Also, the students will get access to the lecture slides. Links to other material in form of research articles or web resources will be provided.

All material will be accessible via the Canvas page for the course.

1.5.4. Student Tasks

The course also defines tasks for the students. These activities shall deepen the understanding of the theoretical knowledge, and will mainly consist of tasks belonging to the following categories:

Modelling and Programming Tasks (Laboratories) These tasks will help the students to both apply the theoretical knowledge on a practical use case, and to gain a deeper understanding of the challenges with respect to embedded system design.

Problem Exercises These problems aim to deepen the theoretical understanding by applying the theoretical knowledge in form of exercise tasks that need to be solved.

Reflection Exercises (Seminars) These are open problems, which aim at getting a deeper understanding of more abstract concepts, for which a single solution cannot be formulated. Designing a system implies to make decisions, where most decisions imply a trade-off. Thus, it is often not clear, which one of two possible decisions is the better one. This is also, why the system designers are also called system architects. In the reflection exercises, the student gets the possibility to reflect about a problem and discuss it in written form.

2. The Embedded Computing Platform

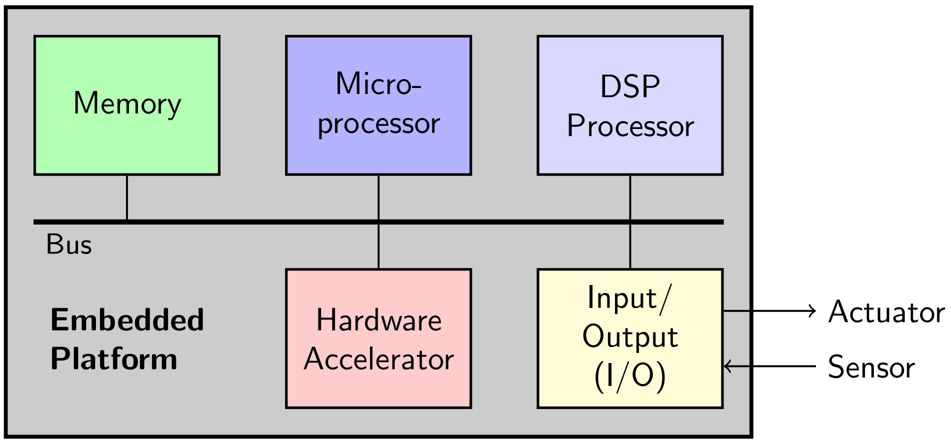


Figure 2.1.: Bus-based embedded system

Figure 2.1 shows a typical embedded system platform. The main components will be discussed in this chapter.

2.1. Microprocessor

The microprocessor is due to its flexibility the key component in most embedded systems. Modern cars include many embedded processors used for simple functions, like window control, up to advanced and critical functions, like engine control or the brake system. The trend to more autonomous systems and autonomous driving will require even more powerful processing capabilities.

Microprocessors are produced in very high volumes, which decreases the cost and enables high optimisation. Microprocessors appear in different variations, from small and very cheap 8-bit micro-controllers to powerful 32-bit microprocessors[1], from general purpose microcontrollers to specialised Digital Signal Processors (DSPs). An advantage of a microprocessor compared to custom hardware is that the microprocessor can be programmed in software, which provides a lot of flexibility, but at the same time it includes an overhead in form of required instruction decoding and memory access to load instructions and to load and store instructions and data variables.

Embedded processors come both as hard cores or integrated circuits, but also as soft cores, which can be instantiated and implemented on a Field Programmable Gate Array

[1]So,there 64-bit processors are in general not used for embedded systems.

(FPGA) or Application Specific Integrated Circuit (ASIC). Custom and programmable hardware, in form of FPGAs, ASICs or full-custom integrated circuits, can usually provide a better performance than microprocessors, but this comes at a higher design cost. Since embedded systems, usually have both a performance and cost requirement, the designer has to find the right trade-off between the use of software running on a microprocessor and hardware solutions. Normally, only the performance-critical functions are implemented in hardware, while all other functions are implemented in software, aiming at a low cost, high flexibility and high performance.

Although many embedded microprocessors for a vast amount of different purposes exist, the architectures and their associated programming model is based on similar general principles. To date, embedded processors are generally programmed in C, and only very performance-critical parts are programmed in assembly language. Also very specialised processors might be programmed in assembly, because for these architectures, it might be difficult or simply too costly to develop an efficient C-compiler.

2.1.1. Programming in Assembly

Each microprocessor has an instruction set, which enables to program the microprocessor in the assembly language. In contrast to standard programming languages, like C, Ada, or Fortran, which are higher-level programming languages, an assembly language can only be used for a certain processor or processor family. This means that assembly programs are not portable between different processor families.

The following example shows a short assembly code fragment for the Nios II processor, which is a soft processor core developed by Altera[2] for their FPGA. However, the choice of the assembly language is not important here, because the following example shall only illustrate basic principles of programming in assembly and the corresponding operations involving the processor hardware, the memory, the address and data bus, and the memory.

Listing 2.1. Tutorial Assembly program fragment implementing loop

```
1         ...
2         movi r1, 1         # Move 1 into register R1
3   loop: addi r1, r1, 1     # Add 1 to R1, store result in R1
4         bge  r2, r1, loop  # Branch to loop if R2 >= R1
5   cont: ...
```

Listing 2.1 gives a program fragment for a loop in assembly language. On the left hand side there are line numbers, which do not belong to the code. There is only one instruction per program line. It is possible to have a label, like `loop` or `cont`, which can be used to provide an address for branch instructions, like `bge`. Comments can be

[2] now Intel

put at the end of the program line using the character `#`. Program execution follows the order of the program lines, if it is not changed due to a successful branch instruction.

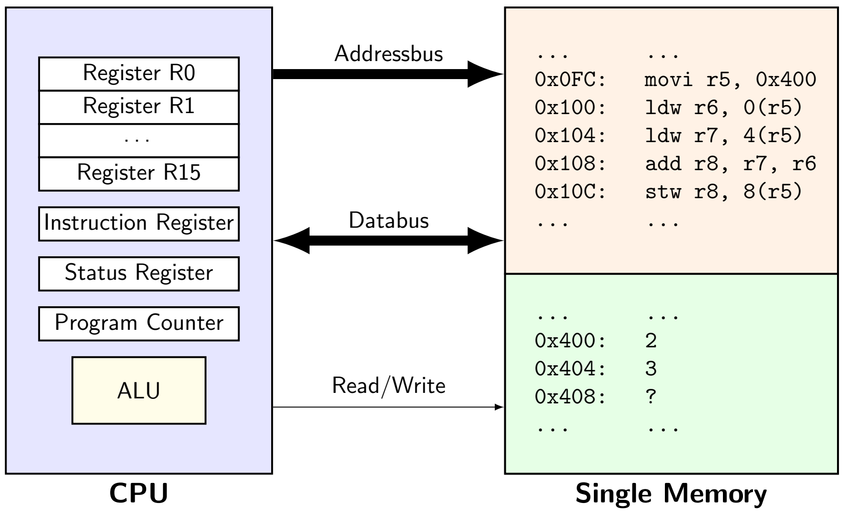
The program behaves as follows. In the first statement (line 2), a value 1 is moved into register `R1` using the `movi` (Move Immediate) instruction. The second statement (line 3) has the label `loop`, which provides an entry point for a branch instruction. The label itself is not an instruction. The instruction `addi` (Add Immediate) adds the value 1 to the value of register `R1` (second operand) and stores it in register `R1` (first operand). The third instruction `bge` (Branch If Greater or Equal) (line 4) compares the registers `R2` (first operand) and `R1` (second operand) and if the value in `R2` is larger or equal than the one in `R1`, the next statement to be executed will be the one at the label `loop` (line 3), otherwise the program will continue with the next program line (line 5).

Listing 2.1 only used registers to process data, but in general data is stored in a memory and the processor has to load data from and store data in the memory. Listing 2.2

Listing 2.2. Assembly program fragment

```
start:
    movi r5, 0x400
    ldw r6, 0(r5)
    ldw r7, 4(r5)
    add r8, r7, r6
    stw r8, 8(r5)
...
```

shows a program that accesses data from addresses and stores data at addresses in the address space of the processor. Figure 2.2 illustrates a simplified processor with a *von*



```
...       ...
0x0FC:    movi r5, 0x400
0x100:    ldw r6, 0(r5)
0x104:    ldw r7, 4(r5)
0x108:    add r8, r7, r6
0x10C:    stw r8, 8(r5)
...       ...

...       ...
0x400:    2
0x404:    3
0x408:    ?
...       ...
```

Figure 2.2.: von Neumann architecture: Processor, address and data bus, control signal and single memory for instruction and data executing an assembly program.

Neumann architecture, i.e. is it contains a single memory for instruction and data. The processor contains a number of registers, where only the most important registers are shown: general data registers `R0` to `R7`, an *instruction register*, a *status register*, and a *program counter*. In addition, the processor also contains an Arithmetic Logic Unit (ALU).

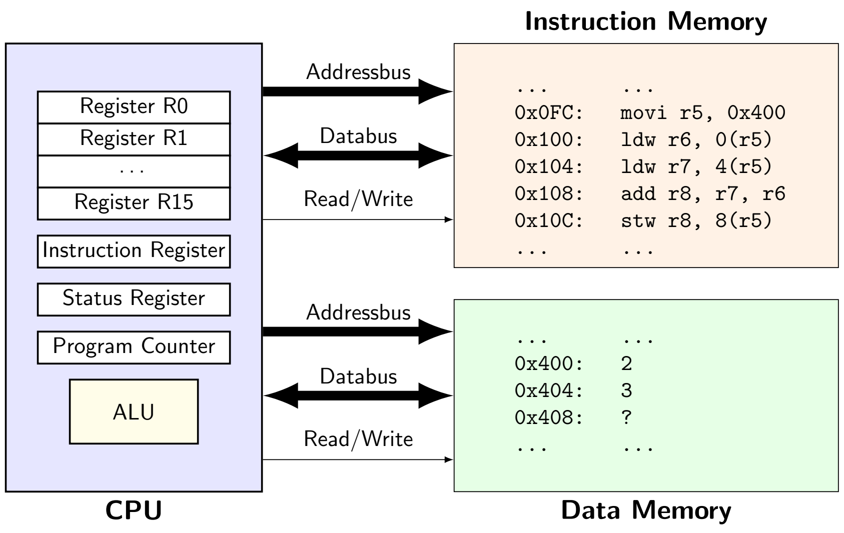
Both, instructions and data are located at different sections in the same memory. The program counter points always to the next instruction that needs to be executed. In this example, it is assumed that each instruction has a size of 4 bytes, and that the program counter has the current value `0x0FC`. This value (`0x0FC`) will then be asserted on the address lines of the address bus, and the current instruction `movi r5, 0x400` is then fetched from memory as bit pattern via the data bus lines into the instruction register. The instruction is then decoded and executed. This means that the value `0x400` is moved into register R5. Then the program counter is forwarded four bytes to the next instruction at address `0x100`. This instruction `ldw r6, 0(r5)` (Load Word) uses indirect addressing, where the address is calculated by the sum of the value located in the register `R5` and an immediate value (`0`), which means that the address is calculated as `0x400`. The value located at this address (`2`) is then loaded as a word into register `R6`. Then the next instruction `ldw r7, 4(r5)` is loaded, which means that the value `3` is loaded into register `R7`. The next instruction `add` (Add) adds the values stored in the registers `R6` and `R7`, and stores the result in register `R8`. Finally, in the last instruction `stw r8, 8(r5)`, the value of register `R8`, is stored in the memory at address `0x408`.

The Nios II processor is a Reduced Instruction Set Computer (RISC) processor and has a *load and store architecture*. The idea of a RISC processor is that all instructions have the same size and that their execution takes the same time, so that an efficient processor pipeline can be established. A load and store architecture means that data has to first be moved into the data registers, before arithmetic or logic instructions can be executed. These instructions require that the data for the operands is already available in the registers. In contrast Complex Instruction Set Computer (CISC) processors have also arithmetic and logic instructions, which can operate on data that is residing in the memory.

Figure 2.3 shows the same program, but executed on a *harvard architecture*, which contains dedicated memories for instruction (instruction memory) and data (data memory). Harvard architectures avoid the memory bottleneck of the von Neumann architecture, because instruction fetches and data fetches do not contend for the same data and address bus and can thus be executed in parallel. A *modified harvard architecture* also contains two separate memories, but even allows to put instructions and data into the same memory. This enables two simultaneous data fetches (without a simultaneous instruction fetch), if the data is stored in the two different memories.

Many modern embedded processors contain implement a harvard architecture with dedicated instruction cache and data cache memories internally on the processor core, but have only a single connection to an external memory off-chip. Figure 2.4 illustrates the architecture, where the access to the external memory needs to be controlled by an arbiter.

Data can be arranged in the memory in different ways. If a processor has a data word



```
...      ...
0x0FC:   movi r5, 0x400
0x100:   ldw r6, 0(r5)
0x104:   ldw r7, 4(r5)
0x108:   add r8, r7, r6
0x10C:   stw r8, 8(r5)
...      ...
```

```
...      ...
0x400:   2
0x404:   3
0x408:   ?
...      ...
```

Figure 2.3.: Harvard architecture: Processor, address and data buses, control signal and distinct memories for instruction and data executing an assembly program.

size of four bytes, as its size of the data registers, four consecutive byte addresses in the main memory needs to be used to store this data word. Different processor families follow different standards for the arrangement of the data, which is referred as *endianness*. In a processor or peripheral circuit that uses

- *big endianness*, the most significant byte is placed first (on the lowest address) and the least significant byte last (on the highest address);
- *little endianness*, the least significant byte is placed first (on the lowest address) and the most significant byte last (on the highest address).

Big and little endianness are illustrated in Figure 2.5.

It is important that devices in a processor or multiprocessor system follow the same endianness standard, because otherwise data can be read in the wrong order, which can lead to unintended bugs or even disasters. The following example shall illustrate this for a multiprocessor like the one in Figure 2.6. A processor using big endianness stores the word `0x11223344` at memory location `0x1000`. Then the word is stored as follows in the memory, `0x11` is stored in address `0x1000`, `0x22` is stored in address `0x1001`, `0x33` is stored in address `0x1001`, and `0x44` is stored in address `0x1003`. If another device, which uses little endian, wants to read this word from the same address `0x1000`, it will assume that the most significant byte is stored in the last address, i.e. in address `0x1003`. Thus, the word would be read `0x44332211`, which means that the value and thus the meaning of the word has changed and that the data is corrupted.

Modern processors provide many clever techniques aiming to improve the performance of the processor system. One idea, which has been introduced together with the RISC architectures is *pipelining*, which is now standard in more advanced embedded processors. To execute an instruction, a processor first fetches the instruction as bit pattern from

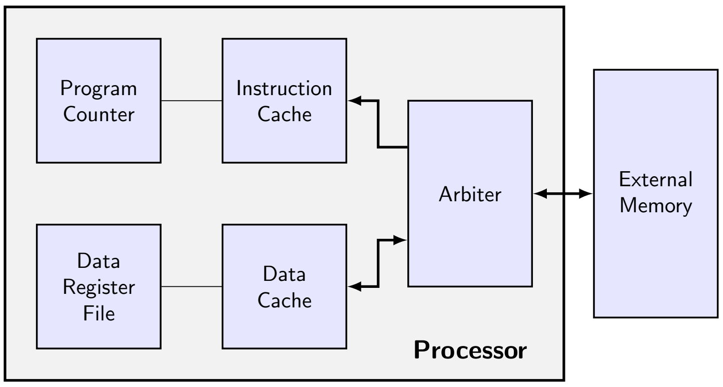


Figure 2.4.: Modern embedded processors often have an internal harvard architecture with dedicated instruction and data cache memories, but have only a normal bus connection to a single external memory.

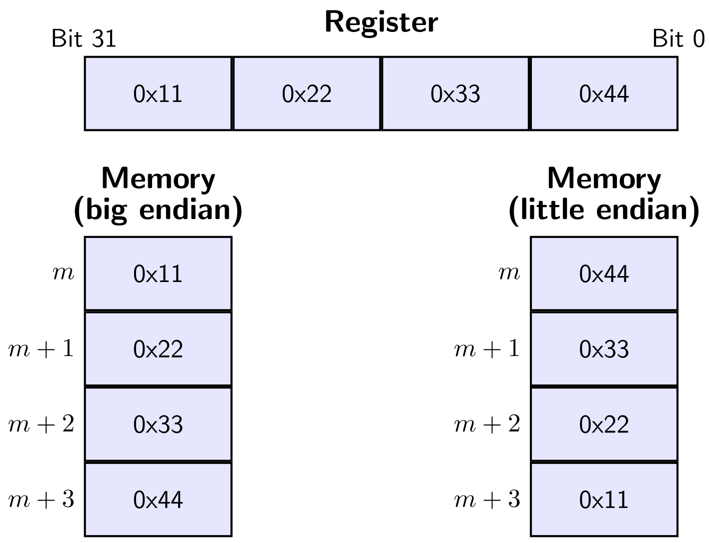


Figure 2.5.: Big and little endianness

memory, then decodes this bit pattern to yield the instruction, and then executes the instruction. The same procedure is repeated for all instructions. A processor with a pipeline will overlap the stages *fetch*(F), *decode* (D), and *execute* (E), to be able to increase the number of instruction per time unit. Figure 2.7 shows how an ideal pipeline can increase the performance of a processor assuming that the phases fetch (F), decode (D), and execute (E) take the same time, one time unit. It is important to point out that the first instruction still takes three time units to execute. This time is also called *latency*, i.e. the time required to go through all stages of the pipeline. But the pipeline improves the *throughput*. Once the pipeline is established, one instruction is executed per time unit.

Unfortunately, the ideal pipeline cannot always be maintained. Figure 2.8 illustrates this situation, where the pipeline *stalls* due to a successful branch. The example assumes

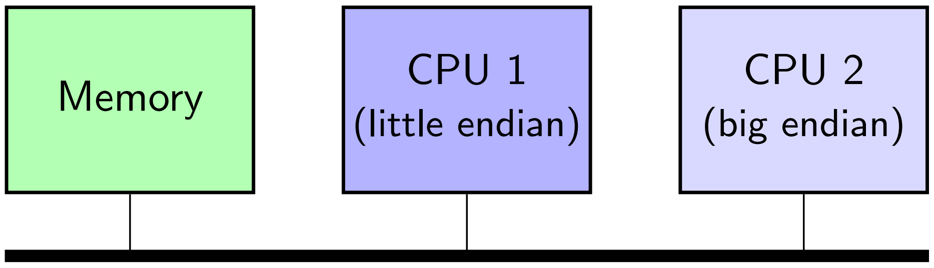


Figure 2.6.: Shared Memory Multiprocessor

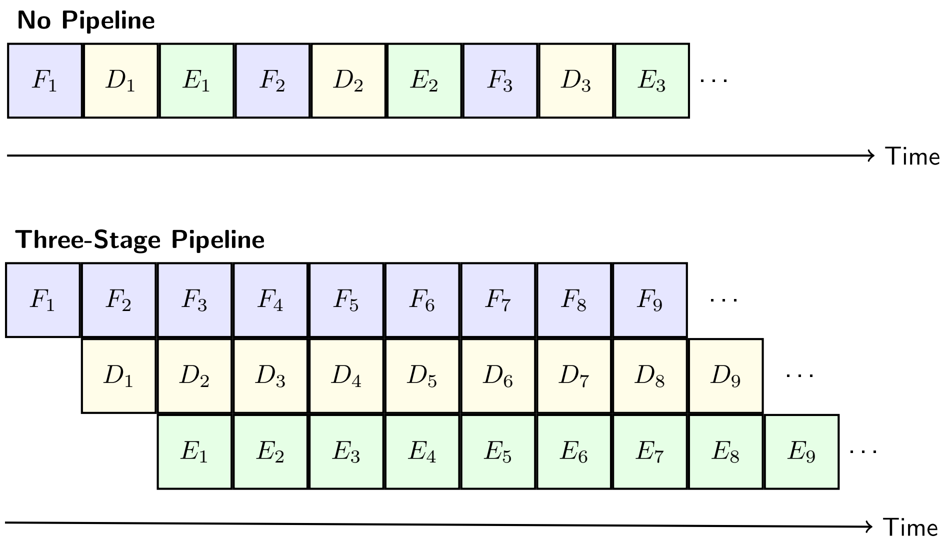

No Pipeline
F_1 D_1 E_1 F_2 D_2 E_2 F_3 D_3 E_3 ⋯
Time
Three-Stage Pipeline
F_1 F_2 F_3 F_4 F_5 F_6 F_7 F_8 F_9 ⋯
D_1 D_2 D_3 D_4 D_5 D_6 D_7 D_8 D_9 ⋯
E_1 E_2 E_3 E_4 E_5 E_6 E_7 E_8 E_9 ⋯
Time

Figure 2.7.: By overlapping different phases, a pipeline enables to increase the number of instructions that can be executed per time unit.

that the pipeline is filled according to the order of the instructions in the assembly program[3]. But when the instruction `bne r2, r0, loop` is executed, the branch is not taken, and the pipeline contains the wrong instructions and must be emptied of *flushed*.

Modern processors have pipelines with more then three stages. As an example, in the Nios II processor family, the processor Nios II/e (economy) processor has no pipeline, the Nios II/s (standard) has a five stage pipeline, and the Nios II/f (fast) has a six stage pipeline.

2.2. Memory System

The memory system is a key part of the embedded system platform. Both instructions and data have to be stored in memories. The embedded system designer can choose from different classes of memories depending on the requirements of the application that shall

[3] Modern processors often have branch prediction, but also in this case they cannot always predict the branches beforehand.

```
loop: movi r2, 5
      subi r2, r2, 1
      bne loop
      add r5, r3, r4
      muli r6, r5, 10
      addi r6, r6, 2
```

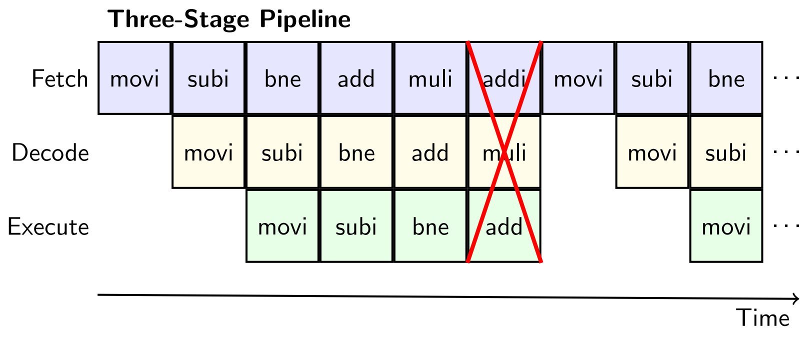


Figure 2.8.: Pipeline stall due to a branch instruction

be implemented. A bad choice may lead to low performance or to unnecessarily high costs. Thus, the embedded system designer should be aware of the available different memory types and should then compose a memory system, possibly consisting of several memories, that matches the needs of the application.

2.2.1. Types of memories

Memories can be classified into two categories, *volatile* and *non-volatile* memories. Volatile memories lose their information, when they are disconnected from power. Non-volatile memories keep their information when connected from power. Embedded systems require both volatile and non-volatile memories.

Non-volatile memories

Flash memory A flash memory is a non-volatile memory. It can erased and updated, but updates are quite slow compared to volatile memories. Flash memories are the bases for memory sticks and memory cards for smartphones and cameras. Thanks to low cost and low power consumption, flash memories are now replacing hard disks in laptop and desktop computers in form of solid state drives.

In general flash memories are used to hold information that is needed in case of power failure or power shutdown. A good example is the boot code for a microprocessor that needs to be executed at the start up of the microprocessor.

PROM, EPROM and EEPROM Older non-volatile memories are Programmable Read Only Memorys (PROMs), Erasable Programmable Read Only Memorys (EPROMs), and Electrically Erasable Programmable Read Only Memorys (EEPROMs), but these memories are much less flexible than flash memories. Especially the reprogramming takes much more effort.

Volatile memories

SRAM A Static Random Access Memory (SRAM) is a volatile memory. A SRAM is usually build with six transistors for the basic memory cell. It's advantage is it's simple memory interface and its small memory access time, because in contrast to the DRAM (see below) there is no need for refreshment. The disadvantage of the SRAM memory is its comparably high cost and large size of a memory cell.

The main application areas are buffers, look-up tables and memories that are often accessed, where access time needs to be small. SRAM memories are also suitable as on-chip or scratchpad memories instead of using a CPU cache.

DRAM A Dynamic Random Access Memory (DRAM) is a volatile memory, where each memory cell is composed by a capacitance and a transistor. The advantage is that the memory is very small and can be produced at a low cost. The disadvantages of the DRAM memory arise due to leakage currents, which can lead to a loss of information if the memory cell is not periodically refreshed. A consequence of this is that the DRAM memory interface is much more complex than the corresponding SRAM interface, and also the memory access time is larger than for the SRAM-technology.

DRAMs are usually used as main memories in an embedded systems, since they can store large blocks of data and available at a low cost.

2.2.2. The memory bottleneck

In an embedded system the access to memory is a major bottleneck, since access to the main memory is very slow compared to the execution of an instruction. In a computer system as shown in Figure 2.9 the major part of the execution time is usually spent for memory operations, while only a small percentage is spent on the actual execution of the program.

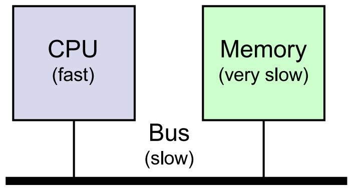


Figure 2.9.: A computer system without a cache

In order to improve the situation the main memory has to move closer to the CPU and needs to be faster. Unfortunately fast memory technologies like SRAM are very expensive as given in Table 2.1. Thus it is economically not justified to build large fast memories.

2.2.3. Cache Memory

By analysing computer programs in detail it can be observed that there is a large factor of *locality* inside computer programs, which exists in form of *temporal* and *spatial* locality. The presence of this locality can be exploited by means of *cache memories*.

Memory Technology	Typical Access Time	Cost per GB in 2004
SRAM	0.5 ns - 5 ns	\$4000 - \$10000
DRAM	50 ns - 70 ns	\$100 - \$200
Magnetic Disk	5 ms - 20 ms	\$0.5 -\$2

Table 2.1.: Memory Technologies (Patterson and Hennessy, 2005)

Temporal locality Temporal locality is based on the observation that a program address that has been used will often be used once again. This is for instance the case in loops, where a certain block of a program is executed several times.

Spatial locality Spatial locality arises from the fact that the location of instructions in memory follows to a large part the logical sequence of instructions in the program code. However, it is not a one-to-one mapping, since the nature of branch instructions destroys the nice order of instructions.

What is a cache memory?

The idea behind cache memories is to place a small, but fast, memory close to the processor that contains a copy of the memory locations that are frequently accessed by the processor. On most modern processors such a small *cache memory* is implemented on the same chip as the CPU as shown in Figure 2.10. This drastically reduces the

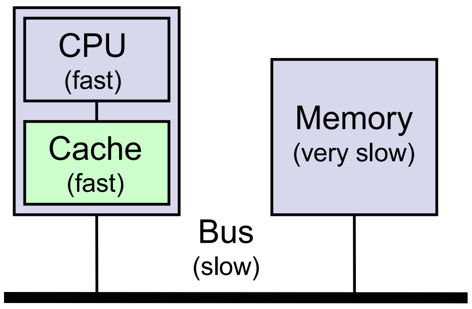


Figure 2.10.: A computer system with a cache

memory access time for the memory locations that are present in the cache.

While cache memories have very short access times, the registers in the processor can be considered as the fastest memories, because the can store data and also arithmetic and logic operations can directly operate on them. Caches can come at different levels, where most embedded processors do not have level-2 caches. Figure 2.11 illustrates the memory hierarchy, where the trade-off between short access time and large memory size is clearly visible.

The access to the cache and the main memory is controlled by a cache controller as illustrated in Figure 2.12. When a processor wants access a memory location, the cache controller checks, if the corresponding cache block is already in the cache. If the data

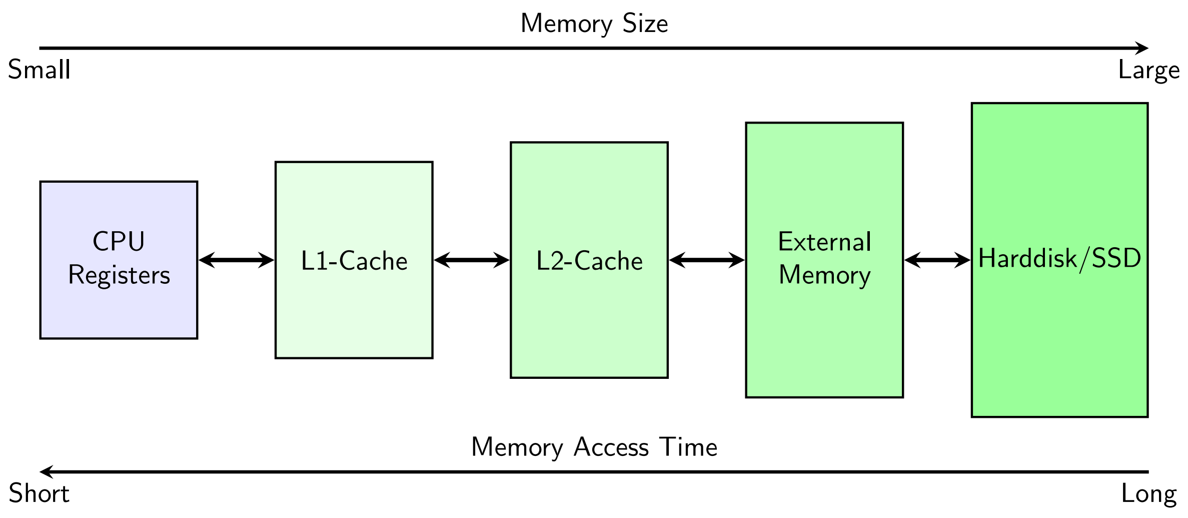


Figure 2.11.: Memory Hierarchy

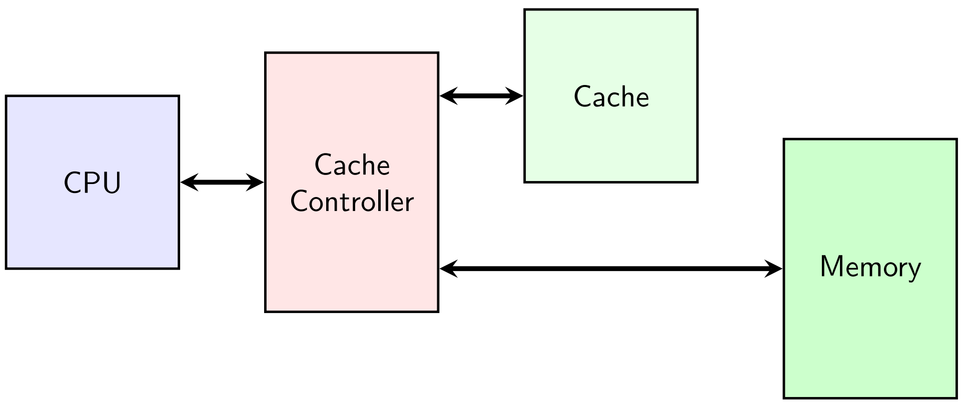


Figure 2.12.: Cache-Memory Access

is not in the cache, the cache controller loads the corresponding memory block into the cache.

If a memory location that a processor wants to access is located in the cache the corresponding data can be retrieved by the processor. This is called a *hit*. If the data is not located in the cache it is called a *miss*. The term *hit rate* is defined as

$$\text{hit rate} = \frac{\text{number of hits}}{\text{number of requests}} \tag{2.1}$$

and the *miss rate* as

$$\text{miss rate} = 1 - \text{hit rate} \tag{2.2}$$

In case of a cache hit the data can be much faster accessed by the processor as in a system without cache. This time is called *hit time*. However, when a cache miss not only the single memory location, but the whole memory block will be loaded into the cache as indicated in Figure 2.13.

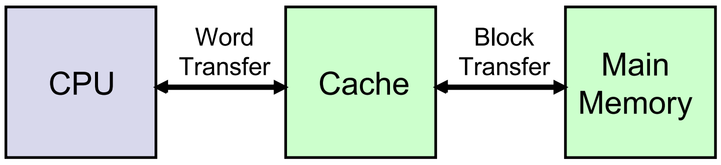


Figure 2.13.: Words are transferred between CPU and cache, while blocks are transferred between cache and main memory

The term *miss penalty* is defined as the time needed to replace a block in the cache with a block from memory and to deliver data from this block to the processor. Since the miss penalty is much larger than the hit time, a fast application requires a high hit ratio.

The following list gives important terms that are used to discuss cache performance.

- *Cache hit*: required location is in cache
- *Cache miss*: required location in not in cache
- *Working set*: set of locations used by a program during an interval of time
- *Compulsory (cold) miss*: location has never been accessed before
- *Capacity miss*: working set is too large
- *Conflict Miss*: two or more locations in working set map to the same cache entry

Cache Architectures

In this section, different cache architectures and their advantages and disadvantages will be discussed, the direct-mapped cache, the 2-way set-associative cache and the fully-associative cache.

Direct-Mapped Cache Figure 2.14 illustrates a direct-mapped cache. A direct-mapped cache has 2^m *cache lines*, where each cache line contains a *status bit*, a *tag* and a *block*, which contains n *data words*. Each cache line also contains a *status bit* that indicates, if the data in the cache block is valid (V), invalid (I) or modified ("dirty") (M). The example cache of Figure 2.14 can store $2^m \cdot n = 32$ data words. The main memory can store $2^{12} = 4096$ bytes or 1024 words, where each word is assumed to have four bytes. Like in this example, caches are normally considerably smaller that the external memory and can thus only store a limited part of the memory locations of the external memory.

As earlier illustrated in Figure 2.13, a full block is transfer-ed between the memory and the cache within a read or write transaction. A direct-mapped cache does not give any flexibility when a memory block shall be stored in the cache. For each memory block, there is a *direct mapping* to a cache line in the direct-mapped cache. The normal mapping for direct mapped caches follows the following operation:

$$\text{Cache Line} = (\text{Memory block number}) \text{ modulo } (\text{Number of cache lines}) \qquad (2.3)$$

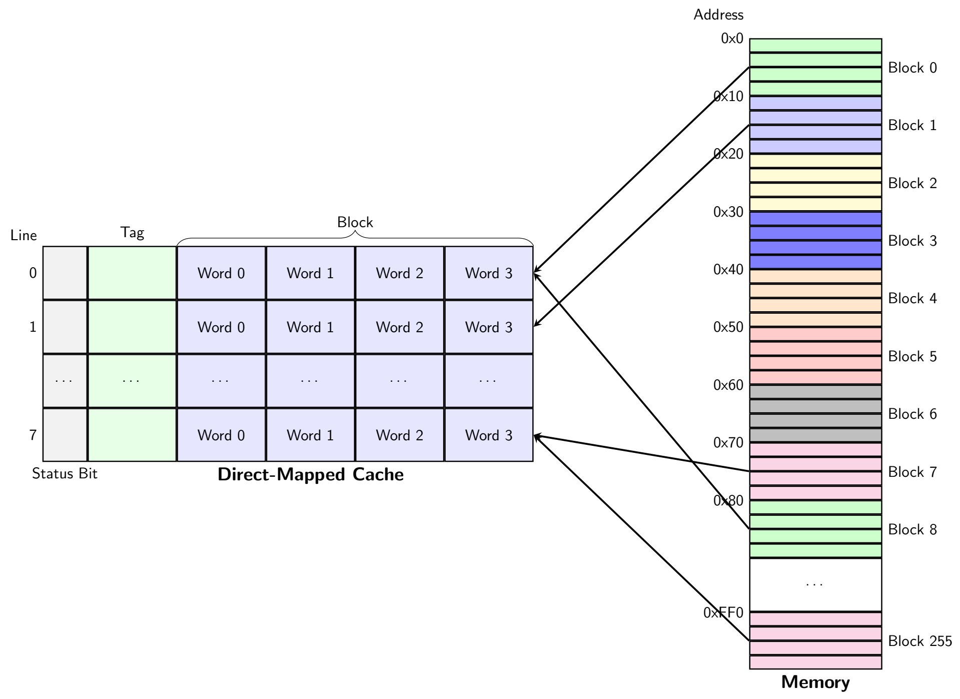


Figure 2.14.: Direct-Mapped Cache

Using the direct-mapping relation, the memory blocks 0, 8, 16, . . . are mapped to cache line 0, the memory blocks 1, 9, 17, . . . are mapped on cache line 1, and so on. Thus, the addresses `0x000` to `0x00F` contain the first four words in memory. In case of a read or write miss, will be transfer-ed to the corresponding block in cache line 0 of the direct-mapped cache. In order to be able to distinguish, which memory block is inside a cache line, further information is needed. This information is represented by the *tag*. Using the example of Figure 2.14, there 12 bits are required to decode the full address space of the byte addressable memory. Of these 12 bits the two least significant bits (b_1b_0) determine the byte inside a word, the two bits b_3 and b_2 determine the word inside a memory block, and the next three bits ($b_6b_5b_4$) determine the cache line following Equation (2.3). Since a cache line stores a whole memory block consisting of four words, the remaining most significant bits ($b_{11}b_{10}b_9b_8b_7$) are needed as the tag to determine, if a certain memory block is located in the cache memory.

Assume that the processor accesses a word in the memory location `0x3F4`. Decoding the address `0x3F4`

b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	1	1	1	1	1	1	0	1	0	0

0x07 Tag	7 Line	1 Word	0 Byte

yields its tag `0x3F`, its cache line `7`, the word location `1`, and the byte location `0`. Thus, if the memory location is in the cache, it is stored in cache line 7, and that tag will be `0x3F`. Furthermore byte 0 in word 1 will be accessed. When the address `0x3F4` is accessed and not already in the cache, then the complete block of four words from address `0x3F0` to address `0x3FF` will be transferred to the cache.

The advantage of the direct-mapped cache, is that its simple mechanism can be efficiently implemented in hardware. However, there is a large risk for cache conflicts. If two memory locations map to the same cache and are accessed after each other, for instance in a loop, there will be a conflict miss.

2-Way Set-Associative Cache A more flexible cache architecture is the 2-way set-associative cache. The idea is to "double" the memory space in each cache line (called *set* in set-associative caches), so that two conflicting memory locations can be mapped simultaneously to the same cache set. Figure 2.15 shows a 2-way set-associative cache

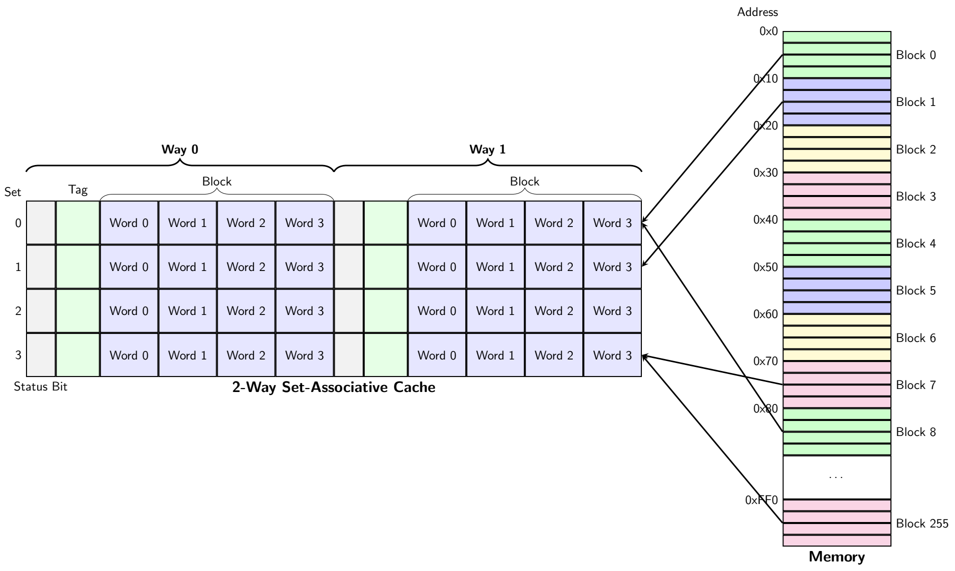


Figure 2.15.: 2-Way Set-Associative Cache

of the same size as the direct-mapped cache in Figure 2.14. Each set contains now space for two memory blocks, which means that an accessed memory block has two *ways* to be stored in the cache memory. Thus a 2-way set-associative cache of a size of 32 words, where each memory block contains four words, has four *sets*, where in each set the data can be stored in one of two "ways".

This means that the previous conflict, which is unsolvable in the direct-mapped cache, can be avoided, because if two memory locations are mapped to the same set, the first memory location can be stored in "Way 0" (or "Way 1") and the second memory location can be stored in "Way 1" (or "Way 0").

The smaller number of sets compared to the number of lines in a direct-mapped cache, in a cache of same size, means that the stored tag requires an additional bit as shown in the table below for the example `0x3F4`.

b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	1	1	1	1	1	1	0	1	0	0
0x0F						3		1		0	
Tag						Set		Word		Byte	

The advantage of the 2-way set-associative cache is its flexibility, which decreases the risk for cache conflicts. But this comes at a price. To determine, if data is in the cache, both the tags in both ways have to be compared with the accessed memory location. To avoid a performance penalty, this comparison needs to be done in parallel, which results in higher hardware costs.

The concept of the 2-way set-associative cache can be generalised to an N-way set-associative cache, where the most flexible cache is the fully-associative cache.

Fully associative cache The fully associative cache architecture is the most flexible architecture, which is an N-way set associative cache with is only one set, which is

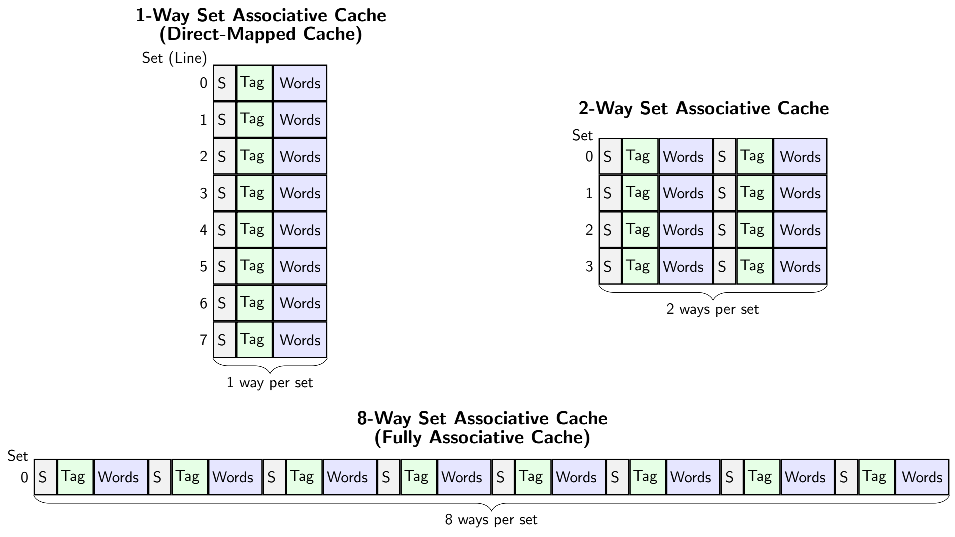

1-Way Set Associative Cache
(Direct-Mapped Cache)
Set (Line)
0
1
2
3
4
5
6
7
S
Tag
Words
1 way per set
2-Way Set Associative Cache
Set
2 ways per set
8-Way Set Associative Cache
(Fully Associative Cache)
8 ways per set

Figure 2.16.: Set Associative Caches

illustrated in Figure 2.16, which also shows that the direct-mapped cache is also a special case of an N-way set associative cache, where $N = 1$.

The tag in a fully-associative cache requires additional bits, which is again illustrated for the address `0x3F4` and a cache size of 32 words.

b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	1	1	1	1	1	1	0	1	0	0
3F								1		0	
Tag								Word		Byte	

The flexibility of the fully-associative cache allows very high hit rates, but it also makes the task of searching the cache for a specific memory location difficult. Since there are no restrictions on the placement of blocks, a search for a memory address needs to visit all cache lines. Thus an efficient implementation requires a lot of hardware, so that a fully associated cache is only justified for small cache memories.

Thus, there is no optimal cache that fits all situations. The direct-mapped cache has the lowest hardware costs, while the fully-associative cache due to its flexible mapping of memory locations to cache memory locations can in general achieve a higher hit rate. The embedded system designer has to choose the right cache based on the design requirements.

Cache block replacement policies When a new block is loaded into a set-associative cache, where all ways are already occupied, one old block has to be replaced by the new cache block. In order to determine, which old cache block should be removed from the cache various different strategies can be applied. Two of the most used *replacement policies* are:

- *random*: the cache block to be replaced is randomly chosen
- *least recently used*: the cache block that has been least recently used will be replaced

Write-through and write-back caches Caches treat read operations in the same way. In case of a cache hit, the data is read from the cache. In case of a cache miss, the cache block is loaded into the cache. But for write operations there are two different strategies.

During a write operation in a *write-through cache*, the cache block is not only updated in the cache, but also in the main memory. This means that in case of a single processor system, the memory has always a valid copy of the cache. A write-through cache requires the following values for the status bits: “valid” (V) and “invalid” (I). At system start all blocks are in the invalid state.

The idea of the *write-back cache* is that it not necessary to write data to the main memory as long as the cache block in not removed from the main memory. Thus a *write-back cache* does in case of a write operation only update the cache block in the cache, but does not update the main memory. Instead the updated cache block is marked with the status bit “modified” (M), also the term “dirty” is often used for a modified cache block. Only, when the cache block is replaced, the whole cache block is written back to the memory.

2.2.4. Scratchpad memories

Caches have been designed to significantly improve the average case performance of the system, but it is very difficult to provide worst case performance guarantees, because of the difficulty to predict, which memory locations will be in the cache at a certain time instant.

In order to improve worst case performance of caches, there are cache architectures, where the cache content can be controlled or locked, so that the designer can decide, which memory locations are in the cache. This techniques to lock part of the contents in the cache has been used in DSPs.

An alternative approach, which makes a lot of sense in the area of embedded systems, is to use a fast and small memory close to the processor, which has an own dedicated address space. These memories are also called *scratchpad memories* and are illustrated in Figure 2.17. The advantage of the scratchpad memory compared to the is that the

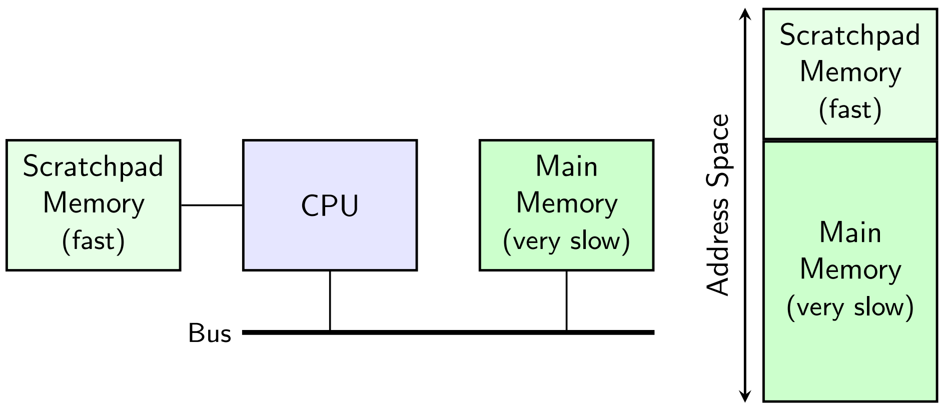


Figure 2.17.: Scratchpad memory

designer can control, which memory locations will be part of the scratchpad memory and thus always be accessible within a very short time. Since embedded systems usually run only a single dedicated applications, the designer can customise the system for this application, and often even dimension the structure of the memory system. Thus, the designer would then be able to reserve sufficient space for the most critical loops and can then calculate a Worst Case Execution Time (WCET), which is very close to the worst case that can occur in practice.

The following example, which is illustrated in Figure 2.18, shows the difference in average and worst case performance for a cache memory and a scratchpad memory based embedded system architecture. Given the example assembly program

```
ldw r2, 0x200
ldw r1, 0x1000
add r3, r1, r2   ; 1 cycle
stw r3, 0x2000
```

where the `add` instruction can be executed in one cycle, the time for a memory access to the cache or scratchpad memory as 1 cycle, and the access time to the main memory is 10 cycles.

In the cache architecture, the best case execution time is 4 cycles, if all data is in the cache, and 6.7 cycles, if an average hit rate of 90% is assumed. However, in the case of real-time system, the WCET is critical, and here the WCET will be 31 cycles, considering the worst case that no memory location is in the cache.

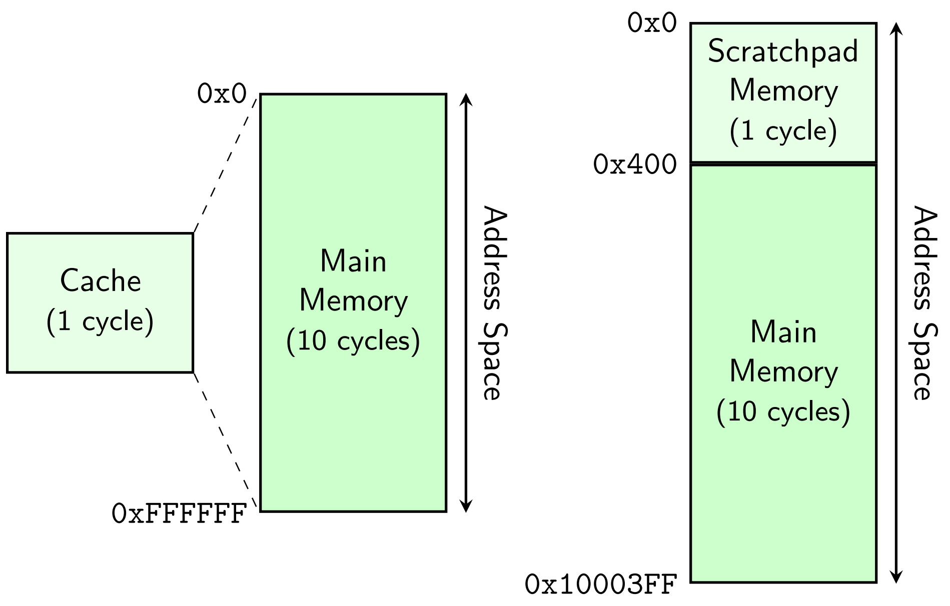


Figure 2.18.: Cache vs Scratchpad

In the scratchpad architecture, the execution time will always be 22 cycles, because there is a fixed mapping of the memory locations to the two memories. Furthermore, the designer can easily calculate the WCET. Thus, although the scratchpad memory architecture has a clearly worse average case performance (assuming a cache hit rate of 90%), it has a clearly better worst case performance, which is of key importance for hard real-time system. In addition, in this example only one memory location has been located in the scratchpad memory, so the performance could be improved, if the memory locations mostly used, can be assigned to the scratchpad memory.

2.3. Input/Output

2.3.1. Memory-Mapped Input/Output

Embedded systems communicate with the environment. This is done via input/output peripheral circuits, which need to communicate with the environment. These devices are controlled by reading and writing to registers. The general method is to connect I/O devices and other peripheral circuits via *memory-mapped I/O*, where the registers can be accessed by reading and writing to addresses, which are mapped to the registers of the peripheral circuit. Figure 2.19 shows the principle for memory-mapped I/O. A peripheral has

- control and data registers, which can be selected via register select inputs;
- data inputs, which allow to read from and write to the registers;

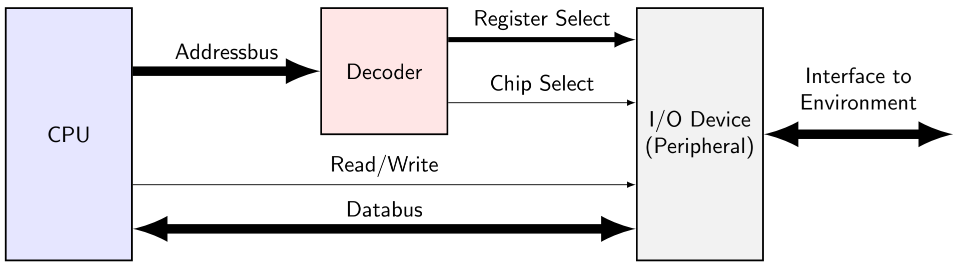


Figure 2.19.: Memory-Mapped I/O

- a read/write control input, which determines, if a data shall be read or written from the peripheral circuit;
- a chip enable input, which enables to read and write data to the register; and
- a separate interface to the environment, if it is an I/O device.

To map a peripheral circuit to the address space, it needs to connected in such a way that the chip enable input is active, if the corresponding address is assigned on the address bus. In addition, a part of the address pins needs also to select one of the registers. Figure 2.20 illustrates how an I/O-peripheral with eight registers and a data

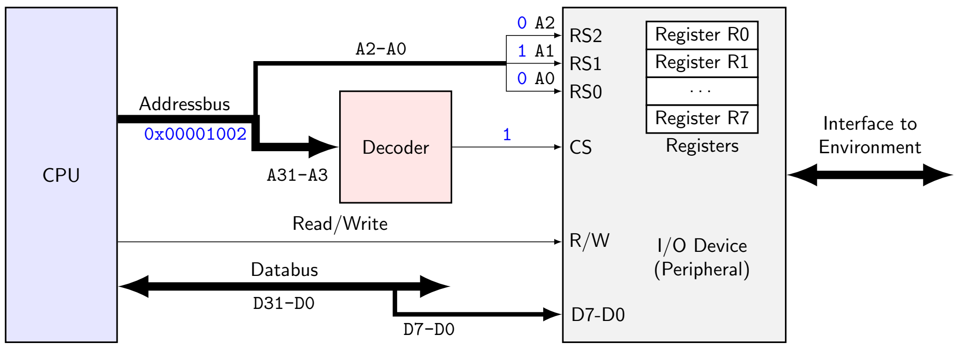


Figure 2.20.: Memory-Mapped I/O

bus of eight bits is connected to a 32-bit processor, so that the registers of the peripheral are mapped to the addresses `0x00001000` (Register 0) to `0x00001007` (Register 7). The registers are selected with the lowest address bits `A2` to `A0`. The other 29 address bits are input to the decoder, only when these bits address the address range from `0x00001000` to `0x10001007`, then the decoder output should be active and have the value `1`. In all other cases, the decoder shall have the output value `0`. The Figure also shows, the case where the address on the address bus is `0x00001002`. In this case, register 2 is selected. The following code sets bit 3 and clears all other bits in register 1.

```
movia r1, 0x1002
movi  r3, 0x08
stb   r3, (r1)
```

2.3.2. Busy Wait Input/Output and Interrupts

An embedded has often to monitor the state of its inputs. There are two main techniques to do this: *busy-wait I/O* (or polling) and *interrupt*.

Busy-Wait I/O

Busy-wait I/O or polling does not require any additional control signals. The processor waits for the I/O-device by constantly monitoring a possible change I/O-device. The following pseudo-code illustrates busy-wait I/O, where the program shall wait until a button has been pressed.

```
while (button not pressed) {
  do nothing - just wait;
}
Continue after button has been pressed!
```

The disadvantage of busy-wait I/O is that the processor cannot do any other useful work while waiting for an I/O-device. Furthermore, it is very difficult to create a program that can deal with simultaneous input from multiple devices.

Interrupt

The idea of the interrupt is that the processor gets notified, when an input event is happening. Thus, there is no need for the processor to actively wait for the input event, but instead the processor can do useful work. Once an I/O-event happens, the processor is notified by an *interrupt request*. If no other interrupt is handled, the processor *acknowledges* the interrupt, and executes an *interrupt service routine*, which corresponds to the identity or level of the interrupt. Figure 2.21 shows the interface that is required

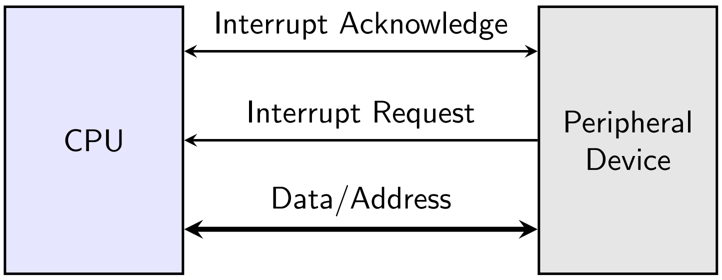


Figure 2.21.: Interrupt Interface

to be able to handle interrupts, in form of control signals for interrupt request and acknowledge. Also additional information like an interrupt vector might be send using the address and data bus. Figure 2.22 illustrates the interrupt mechanism. In the normal state a foreground program, the main program, is running. In case of an interrupt, the program receives an interrupt vector and jumps to an interrupt service routine. In the

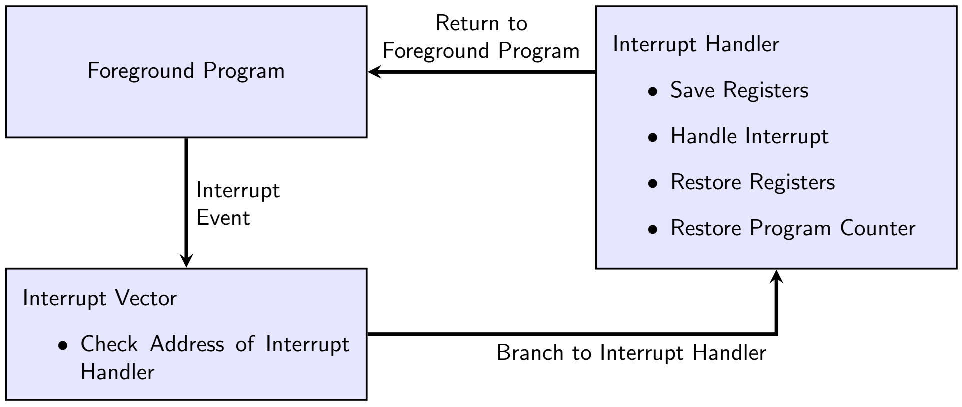


Figure 2.22.: Interrupt Mechanism

interrupt service routine, first registers need to be saved before the interrupt can be handled, then the registers are restored and the program counter is loaded with the address before the interrupt, so that the foreground program can continue its work exactly in the same state as before the interrupt.

Interrupts allow to create a concurrent execution of several activities. Although the processor still runs only a single thread, several activities can run concurrently and give the impression of a parallel execution. Assume the following program, where a program waits for inputs from a receiver, operates on the input, and then sends the modified input. If this program would be implemented with busy-wait I/O, the program would be run the following loop shown as pseudo-code.

```
loop
  Wait for input from receiver;
  Operate on input;
  Send modified input when sender is ready;
end loop;
```

The program cannot do anything else than to wait, if the input has not arrived at the receiver, or when the sender is not ready. In systems, where data does not arrive at regular or periodic time instances, this means that the system resources are used very inefficiently and the program spends a lot of time waiting for input or that the sender becomes ready.

Interrupt allows the parallelisation of duties, so that the system can do several things at the same time. For this example, interrupt combined with buffers to store data from the receiver for buffering data that shall be send enable a much more efficient solution. The following three different tasks can be parallelised:

- Wait for new data from receiver and if received store it in receive buffer (interrupt)
- Consume data from receive buffer, operate on it and store it in send buffer (foreground program)

- Wait for the sender to be ready and the availability of data in the send buffer, and the send data (interrupt)

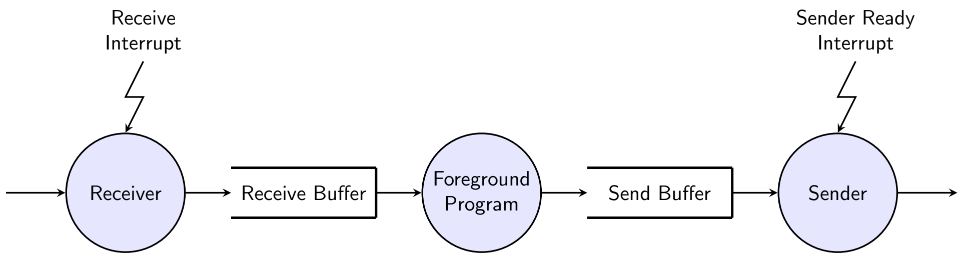


Figure 2.23.: Parallelisation of duties using the interrupt mechanisms and buffers

Figure 2.23 illustrates this solution. The buffers are needed to decouple the two interrupt service routines and the foreground program, which now run concurrently and can handle to process a non-periodic arrival of input data by being able to buffer input data. However, the solution only works, if the buffers are dimensioned correctly. If the buffers are too small, the buffers will overflow. If the buffers are too large, they design is unnecessarily costly. The dimensioning of buffers is a critical activity in the embedded system design process.

Interrupts enable concurrent behaviour, and are essential to implement RTOS, where timer interrupts are used to periodically schedule tasks.

Although interrupts enable concurrent execution and more efficient systems, they also pose a new challenge. It is very difficult to debug programs, which use interrupts. The foreground program can be interrupted at any time, and it is very difficult or even impossible to reproduce the exact situation, where the error occurred. This makes it hard to repeat errors and to debug an interrupt routine. Thus, a very careful design is required.

2.4. Interconnection Network

An embedded system contains many hardware components, which need to be connected with each other by an interconnection network. An interconnection network connects these components as indicated in Figure 2.24 and would ideally provide (a) full connectivity between components (b) negligible delay; (c) non-blocking connections; (d) unlimited bandwidth; and (e) scalability. But in practice, it will not be possible to achieve such a network at a reasonable cost.

The dominating interconnection network in embedded systems has traditionally been the bus structure, which connects the different components by a single logical communication line as illustrated in Figure 2.25. The bus structure has a very low hardware cost, but cannot always provide the scalability, which is required in modern embedded system architectures, which consists of a very large number of components. In order to address this situation, Network on Chip (NoC) architectures have been introduced, which provide a much better scalability. Figure 2.26 illustrates a typical Network on

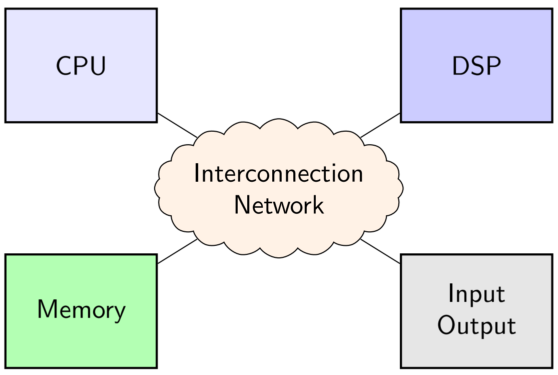


Figure 2.24.: Interconnection Network

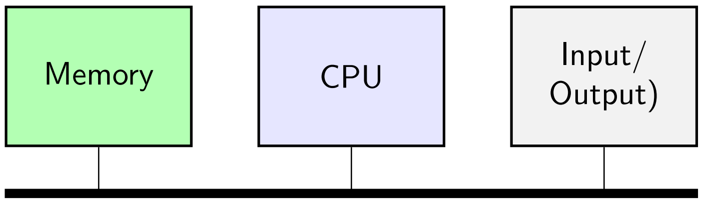


Figure 2.25.: Bus Architecture

Chip (NoC). The resources (R) contain the computing resources. A typical resource in the NoC would be the bus-based processor memory network of Figure 2.26, where the I/O-device is replaced by a *network interface*, which is the communication interface to the packet-switched network, where packets are sent from the resources using the network interface to other resources via communication links and switches (S). Obviously, the high scalability comes with additional hardware costs, which cannot always be motivated. So, in practice, often hybrid solutions are used for modern interconnection architectures, using elements of both, bus-based and NoC, architectures. The following text will focus on the bus architectures, for more information on NoC architectures see the related literature in the field, for instance the survey by Bjerregaard and Mahadevan (2006). There exist many different standardised bus architectures for different application domains. The chapter only focuses on important principles, but does not aim at giving a survey on bus architectures.

2.4.1. Bus-based networks

The bus provides a communication link, which all connected components can access. Since this shared communication link is only a single logical connection, it can only be used for one type of data transfer at each time instance. This means that communication has to be *serialised*, because otherwise communication messages will interfere and corrupt each other. This serialisation means that the messages or data transfers are *totally ordered*. Although this serialisation implies that only one component can write or send to the bus at a single time instance, all other components on the bus can read or receive

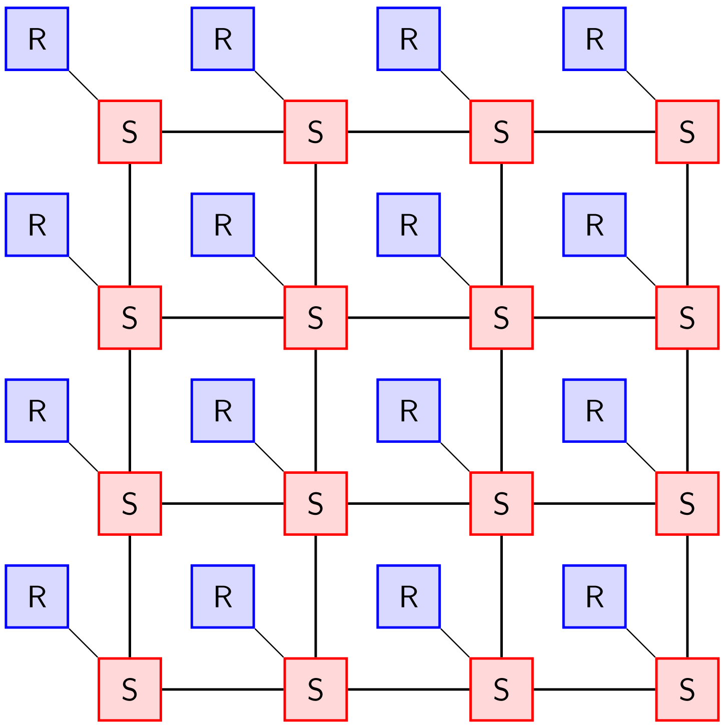


Figure 2.26.: Network-on-Chip Architecture

a message, because they are all connected to the same communication link and reading does not harm the message on the bus. This *broadcast* property is a special property of the bus, the NoC does not have this property.

A *bus transaction* consists of a sequence of messages which together form a transaction. An example for such a transaction is a memory read transaction, which requires a memory read message and a reply with the requested data. A *bus message* is formed by its logical unit of information, e.g. a read message contains an address and control signals for read access. A message requires a number of *cycles* to be transferred from sender to receiver over the bus.

Buses can be divided into synchronous and asynchronous buses. A *synchronous bus* requires a clock, allows to divide the time line into clock cycles and enables a protocol that expects that data is available after certain clock cycles. The advantage of such the synchronous bus, that it does require very little extra logic and can run very fast. The disadvantage is that devices are required to share the same clock. Furthermore, for larger systems there is the potential problem of *clock skew*, which means that due to different physical distances from the clock source, the clock edge will not arrive at the same time at all components. Figure 2.27 illustrates the functionality of a synchronous bus.

An *asynchronous bus* does not require a clock, which means that many different devices can be connected. The absence of a clock also removes the potential problem of clock skew and enables to connect components that have a data-dependent delay. The asynchronous bus requires a *handshake protocol* to order the messages in a bus trans-

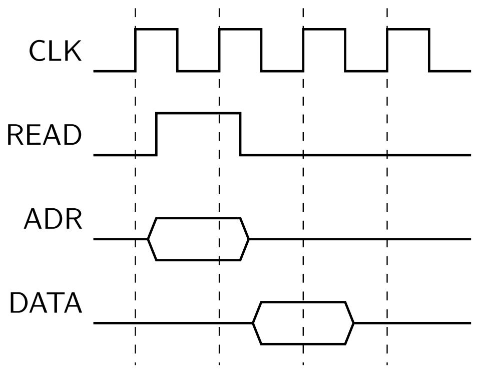


Figure 2.27.: Synchronous bus

action. The functionality of an asynchronous bus is illustrated in Figure 2.28 using a

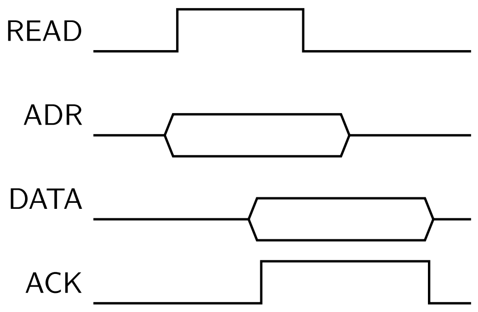


Figure 2.28.: Asynchronous bus

read transaction. The handshake uses the following steps: (a) master assigns address on bus and activates READ when address is stable, (b) slave activates data signals with requested data and activated ACK if the data is stable, (c) master deactivates READ when data is read; (d) slave deactivates ACK.

On a more abstract level, a bus can be seen as a single logical bidirectional communication link. But physically, as bus is composed of a unidirectional address bus, a bidirectional data bus and several control signals, which is illustrated in Figure 2.29.

2.4.2. Bus Arbitration

In order to ensure that not more than one message is send on the bus at each time instance, a clear protocol is required, which avoids simultaneous messages. Different devices connected to the bus have different roles and not all devices are allowed to take control of the bus. A *bus master* is a device that is allowed to initiate operations on the bus. A CPU is a typical bus master, but there are also other devices that are allowed to act as bus master, and in this case a protocol is needed that only one device can take over the control of the bus at each time instance. Such a protocol requires additional control signals and hardware.

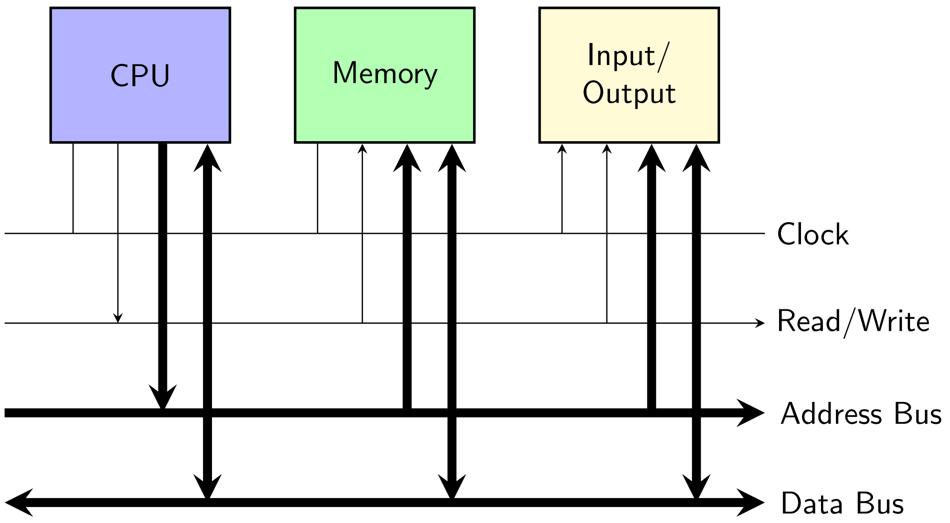


Figure 2.29.: Physical view of an embedded system bus

The devices that are not allowed to take control of the bus on their own are called *bus slaves*. A bus slave is only allowed to response to a request of a bus master. Thus, a bus transaction includes two parts as illustrated in Figure 2.30. A bus master issues

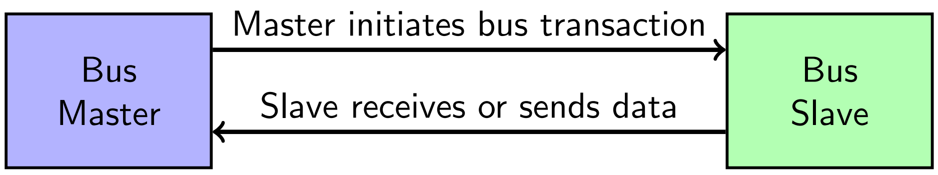

Bus Master
Master initiates bus transaction
Slave receives or sends data
Bus Slave

Figure 2.30.: Bus Master and Bus Slave

a request for a bus transaction by sending a message, i.e. assigning an address and the corresponding control signals, to a bus slave, and then the slave reacts by either sending data to or receiving data from the bus master.

One of the most important issues in the design of embedded systems is to decide about a policy and a protocol, which decides, which of the bus masters that want to access the bus is granted the right to take the bus. In order to implement an arbitration scheme another device, the *arbiter*, and additional control signals are introduced as shown in Figure 2.31. Then a bus arbitration scheme is used to decide, which bus master should get access to the bus. A bus master that wants to use the bus issues a request to the arbiter by activating its Request signal. The arbiter selects one bus master by activating the corresponding Grant signal. Only a bus master that has received a grant from the bus master is allowed to take control of the bus. Once the bus transaction is finished, the bus master deactivates its Request signal and the arbiter selects the next bus master that is allowed to take control of the bus.

The arbitration scheme that is used by the arbiter tries to balance two important factors.

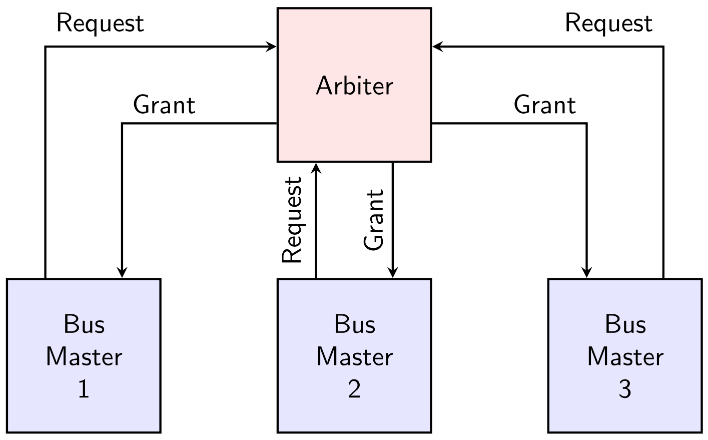


Figure 2.31.: Bus Arbitration

- *Priority:* There might be bus masters, which control functions that are more important. In particular, in safety-critical systems, it is of extreme importance that these devices can execute their functions in time.
- *Fairness:* All devices should get access to the bus. The situation that a device is totally locked out from a bus shall be avoided.

Fairness is a key property of an arbiter, and the designer has to choose, which degree of fairness is suitable for the system. Different degrees of fairness can be defined:

- *No fairness:* There are devices in the system, which requests might never be served
- *Weak fairness:* A request for a device in a system will be eventually served
- *Strong fairness:* Requests from different devices will be served equally often
- *Weighted strong fairness:* Each device has a *weight*, and the number of times a device is served is corresponding to its weight
- *First In First Out (FIFO) fairness:* Requests of different devices are served in the timely order the requests have been made

There is no general rule, which fairness should be applied. This depends totally on the application and its design constraints.

The original bus structure is a very versatile interconnection network that enables to easily add new devices as long as they follow the same bus standard. It has important inherent properties like *serialisation* and *broadcast*. The original bus can be implemented at a low cost using a shared communication links. Unfortunately, the original bus only provides limited scalability and can easily create a communication bottleneck. Bus bridges have been used to connect different buses, which can then work independently of each other but can share information via the bridge.

2.4.3. Slave-Side Arbitration

Slave-side arbitration extends the bus concept by not having a single arbiter for a whole bus, but by using a dedicated arbiter for each slave device. This enables that several bus master can access different slaves simultaneously, which can increase the performance significantly. The principal for slave-side arbitration is illustrated in Figure 2.32. Three

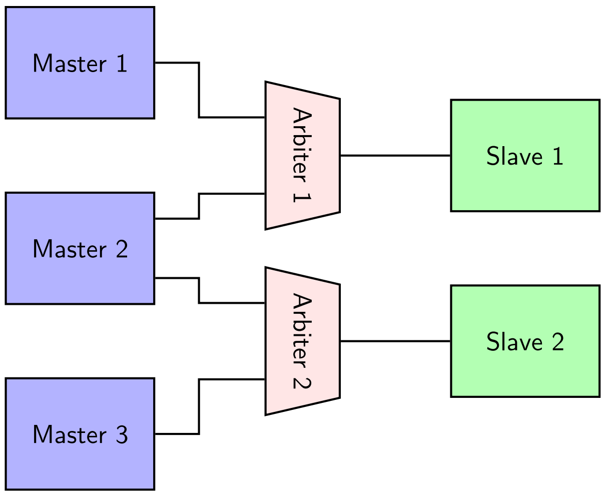


Figure 2.32.: Slave-side arbitration

bus masters communicate with two slaves. Both bus master 1 and bus master 2 are connected to bus slave 1, which requires an arbiter for slave 1. Also for slave 2 an arbiter is needed, because both bus master 2 and bus master 3 access bus slave 2. Bus master 2 has two master ports, one for bus slave 1 and another one for bus slave 2. It is then possible that for instance bus master 1 communicates with bus slave 1, while bus master 2 communicates simultaneously with bus slave 2. The arbiters can use different arbitration policies as discussed in Section 2.4.2.

An example for an architecture that uses slave-side arbitration is the Altera Switch Fabric, which is used for their Nios II processor systems. The arbitration is done using weighted round-robin. Figure 2.33 illustrates weighted round-robin as part of slave side-arbitration. Bus master 1 has a weight of 2 for the access to slave 1, bus master 2 has a weight of 1 for the access to slave 1, and a weight of 2 for the access to slave 2. Bus master 3 has a weight of 3 for accessing slave 3. Thus for slave 1, each round has a length of $2 + 1 = 3$ access cycles, and a round will have two accesses of bus master 1 and one access of bus master 2. For slave 2, the length of the round is $2 + 3 = 5$ cycles. Thus, master 2 will have 2 accesses in a round, while bus master 3 has three accesses in a round. If a bus master does not want to use its share, it depends on the particular slave-side arbitration protocol, if another bus master can use this share.

Slave-side arbitration can significantly increase the system performance compared to the original bus, but the interconnection network can grow very quickly and may result in very high hardware costs for larger systems. Thus, often slave-side arbitration is a very promising solution for high-performance systems, while the original bus is suitable

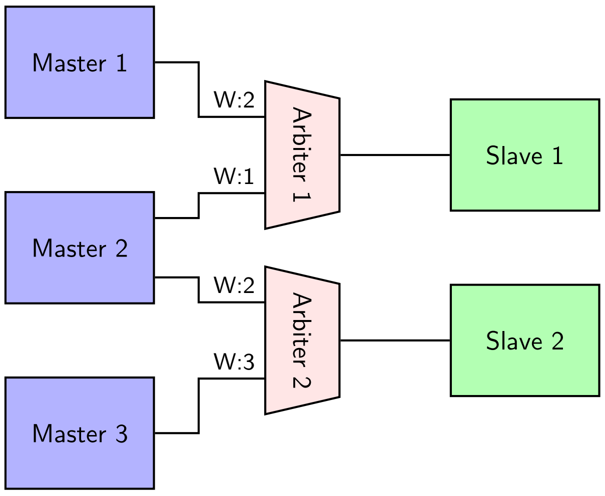


Figure 2.33.: Slave-side arbitration with weighted round-robin

for systems with lower performance. A bus bridge can be used to connect the buses into an integrated system.

2.4.4. Time-Division Multiplex Bus

In a Time-Division Multiplex (TDM) bus, the bus access is divided into time slots. These time slots are statically assigned to different bus masters, which can only send or receive during these time slots. Thus, a Time-Division Multiplex (TDM) scheme gives each bus master a guaranteed dedicated share of the communication resources at well-defined time instances. There is no need for an arbiter, since the time slots are pre-assigned. Figure 2.34 shows an example of a TDM bus. Three bus masters, M1, M2, and M3,

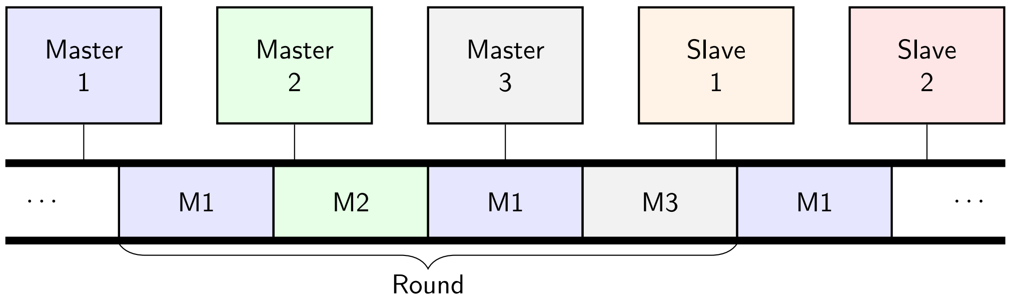


Figure 2.34.: Time-Division Multiplex (TDM) bus

share a TDM bus with two slaves, S1 and S2. The periodic TDM schedule is defined as M1-M2-M1-M3, which means that bus master M1 can use the bus in the first and third cycle of each period, M2 can use the the bus in the second cycle of each period, and bus master M3 can use the bus in the third period. During a period, when the bus master owns the bus, the bus master can conduct a bus transaction with a selected bus slave.

The TDM scheme us fully predictable and enables to give a guaranteed Quality of Service (QoS) level for each band master. Thus, using a TDM architecture, it is much easier to give real-time guarantees for an application, since the QoS level is well-defined. The disadvantage of the TDM architecture is that it is does not provide any flexibility, for instance can an empty slot not be used by another bus master. Thus TDM should be used for safety-critical applications with hard real-time constraints, but they are less suited for best-effort or non-real-time applications, which mainly aim at a high average case performance.

2.5. Peripheral Components

I/O components are a special case of peripheral components. These components are used to support or offload the CPU to conduct certain tasks. There exist a vast amount of different peripherals, so in the following text only a few of them can be described.

2.5.1. Parallel Input/Output Peripheral

The parallel I/O peripheral is used to connect external I/O units like buttons, switches or Light Emitting Diodes (LEDs) to the embedded computer system. As shown in

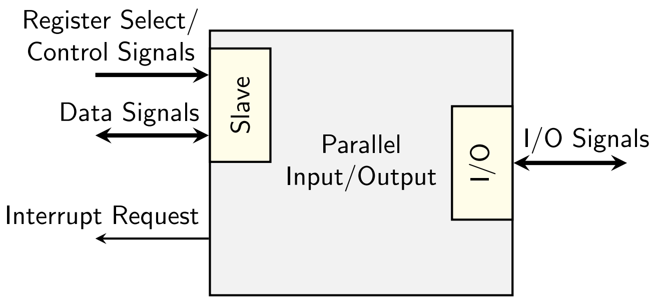


Figure 2.35.: Parallel Input/Output Peripheral

Figure 2.35, the peripheral is connected as bus slave via address bus, data bus and control signals to the bus and often has an interrupt request output to issue an interrupt when there is an I/O event. The parallel I/O peripheral has several parallel configurable I/O connections, which can by programming the peripheral's internal registers be configured as input, output or bidirectional I/O port.

2.5.2. Direct Memory Access Peripheral

The task of the Direct Memory Access (DMA) peripheral is to offload the CPU by handling the transfer of input data from an I/O peripheral to the memory or the transfer of output data from the memory to the I/O peripheral. While the DMA handles the I/O transfer, the CPU can do other useful work. Figure 2.36 shows the interface of a DMA peripheral. The DMA peripheral has both a slave and a master port. The slave port is

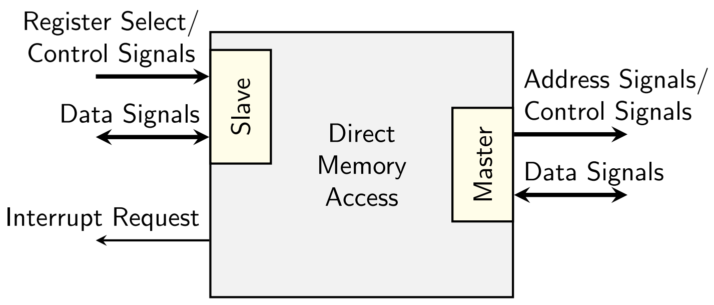


Figure 2.36.: Direct Memory Access Peripheral

used by the CPU to configure the DMA for the next transaction, which is defined by a source address, a destination address, the block length corresponding to the data that shall be transferred, and a transfer mode. Then the CPU gives the DMA the right to act as bus master. Using the bus master port, the DMA starts the bus transaction. Once the data is transferred, the CPU is informed by the DMA using an Interrupt Request that the data transfer is finished.

2.5.3. Timer Peripheral

A timer is a key components for any embedded system that has to react in a timely manner. A timer is implemented as a counter, which can be loaded with an initial value and the counts normally downwards. The current state of the counter can be accessed by reading from the corresponding timer register. In this way, the timer can be used to measure the time between two time instances. Another and very important usage of the timer is to use the timer as a periodic timer, which means that the timer is loaded with an initial value corresponding to the desired time period and then counts downwards. Once the timer reaches zero, an interrupt request is issued, and then the timer is reloaded with the same initial value. Thus, the timer will deliver an interrupt request periodically. This usage of a timer is a prerequisite for RTOS, where tasks are released periodically. Figure 2.37 shows the physical interface of the timer device. The

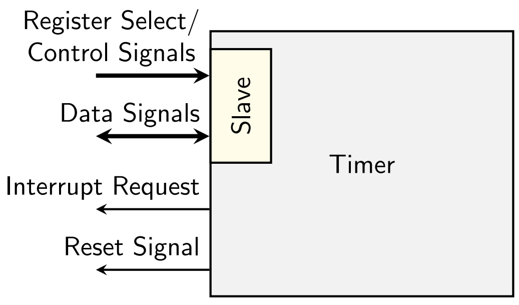


Figure 2.37.: Timer Peripheral

timer is a slave device, which is configured by assigning values to registers. The timer often not only has an interrupt request output port, but also a reset request output port to implement a watchdog timer.

A watchdog timer is used to check periodically, if the system is still working in a correct way. The idea is that, if the system executes correctly, it will always with in a certain amount of time reach the same statement of a program. If the system does not reach the statement within this time, the system seems is considered as faulty and action has to be taken. The watchdog timer is a timer that works as a periodic timer counting downwards. The watchdog timer is loaded with a value corresponding to a time interval. Normally, the timer should never reach zero, because the code contains a statement that reloads the timer, which should always be reached in the selected time interval during fault-free operation. If there is a fault, like a deadlock, the system will not reach the statement that reloads the timer, and then the watchdog timer activates the reset signal. This reset signal can be used either to reset the CPU or to take another appropriate action. Figure 2.38 illustrates how the watchdog timer interacts with the CPU.

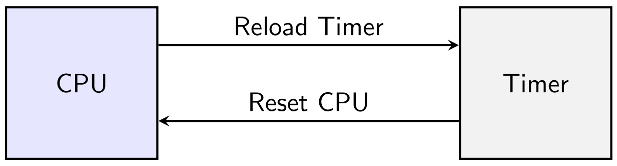


Figure 2.38.: Watchdog Timer

2.6. The embedded software platform

The chapter discussed mainly the embedded *hardware platform*. However, the platform also consists of low-level software, including Operating System (OS) support. In general, this low-level software is programmed in C. Appendix C discusses important properties of the C-programming language in relation to embedded system programming. Furthermore, Section 3.2 gives an overview of RTOSs. A programming language that has been developed for rbust safety-critical systems is Ada, which is discussed in Appendix D.

3. Real-Time Systems

3.1. Introduction to Real-Time-Systems

Many embedded systems are *real-time systems*. A system is classified as a real-time system, if it is required to work on a timely basis. In a strict meaning, this does not mean that the system has to meet any timing deadlines! Thus a system that reads an input every second can be considered a real-time system.

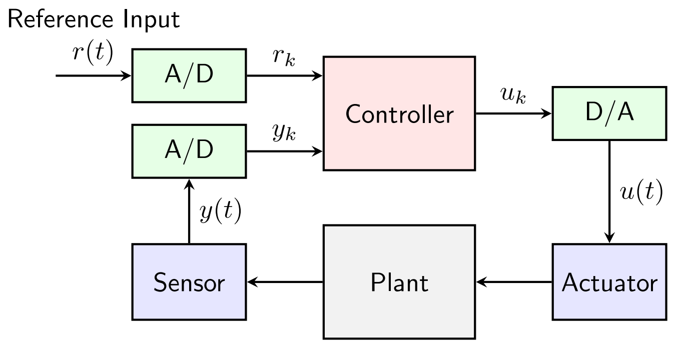


Figure 3.1.: A real-time control system regulates a plant by on a timely basis sensing the plant's state and after computing an appropriate reaction stimulating the plant through actuators.

Figure 3.1 shows a typical real-time system in form of a controller, which is designed to control a plant. A plant can be any controlled system, e.g. an aircraft, an autonomous car, or a nuclear power plant. The aim is that the plant follows a reference input $r(t)$. The state of the plant is monitored by sensor inputs, and the plant is stimulated by the controller via actuators. Assuming that the controller is implemented as embedded system, analogue/digital conversion is required to convert the analogue sensor output signals$y(t)$ into digital signals y_k, and digital/analogue conversion is required to convert the digital actuator output signals u_k to analogue signals $u(t)$.

The controller receives periodically the sensor and reference input signals and has to compute a reaction based on a control law within a given amount of time, so that the plant can be stimulated in time. If the reaction of the controller cannot be computed in time, the plant cannot be stimulated in time, which can have disastrous consequences.

Within the real-time theory, the controller can be modelled as a periodic task τ. A task is a sequence of identical activities, which are continuously executed until completion. These activities are called *task instances* or *jobs*. Although different jobs run identical instructions, they consume different input data.

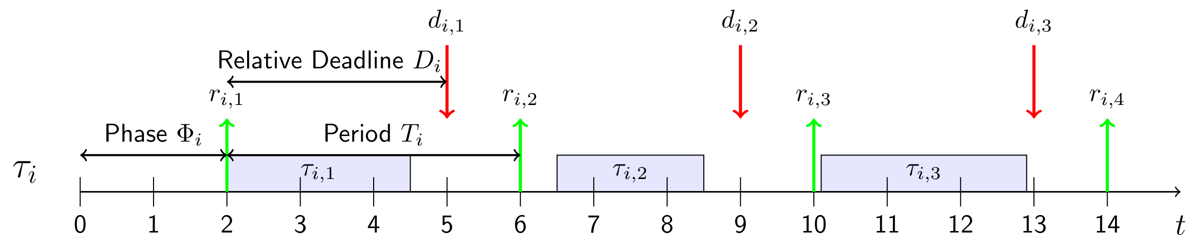


Figure 3.2.: A periodic task is a set of identical activities, called task instances or jobs.

Figure 3.2 illustrates the modelling of the controller as a periodic task τ_i. After an initial startup time, or *task phase* ϕ_i, the first job $\tau_{i,1}$ of the task is *released* at the *release time* $r_{i,1} = \phi_i$, when the input values are available. Each controller job has to compute the result of the control law within a given time after its release time. This time is called the *relative deadline* D_i. Given the release time $r_{i,k}$ of a job k within a task τ_i and the relative deadline D_i, the absolute deadline $d_{i,k}$ of a job can be calculated as $d_{i,k} = r_{i,k} + D_i$. Jobs are released periodically with a period T, which enables to calculate the release time $r_{i,k}$ of the k-th job of a task τ_i as $r_{i,k} = \phi_i + (k-1)T$. In Figure 3.2 the following values are used: $\phi_i = 2$, $T_i = 4$, and $D_i = 3$. More detailed information about the real-time model and also algorithms for the scheduling of real-time systems will be given in Section 3.3

Real-time systems are classified dependent on the timing constraints that are imposed on them. Unfortunately, there is not a single well-defined definition in the literature, although there quite a strong informal agreement on what a *soft* and *hard* timing constraint is.

Buttazzo (2011) introduces the following definitions for real-time tasks, where also the term *firm* task is introduced.

A real-time task is said to be

- *hard*, if producing the results after its deadline may cause catastrophic consequences on the system under control.
- *firm*, if producing the results after its deadline is useless for the system, but does not cause any damage.
- *soft*, if producing the results after its deadline has still some utility for the system, although causing a performance degradation.

Real-time systems are mainly grouped into hard and soft real-time systems. A hard real-time system is a system that has to satisfy hard real-time constraints, and a system that only has to meet soft real-time constraints is classified as soft real-time system. Examples for hard real-time systems are the break system in a car or an system that automatic control systems for trains. A typical soft real-time system is a video processing system, where a non-satisfied timing constraint will lead to pixel or image errors, which imply a loss of service quality, but can be occasionally tolerated.

For safety-critical real-time systems it is of utmost importance to determine the correctness of the timely behaviour of the system. There is a need for validation methods already at early design stages, because a conceptual error leading to a missed deadline can be very costly to correct afterwards or even worse may first be detected during the usage of the system with the possibility of severe consequences.

Many embedded real-time systems do execute several activities concurrently. Fig-

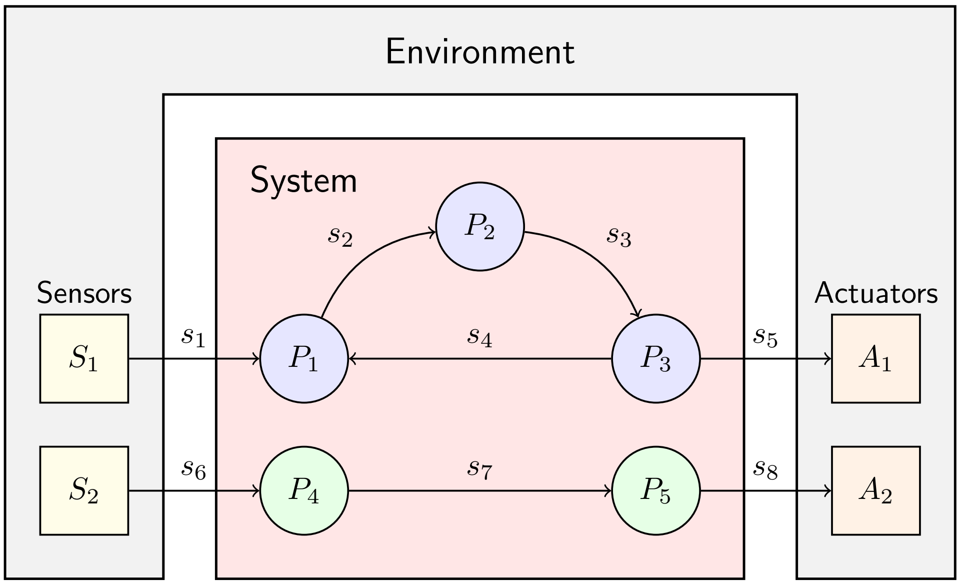


Figure 3.3.: Concurrent Processes

ure 3.3 illustrates such a system, which can be divided into two subsystems. The first subsystem consists of the communicating processes P_1, P_2 and P_3, and receives inputs from sensor S_1, and sends it results to actuator A_1. The second subsystem consists of the tasks P_4 and P_5 and sends its results to actuator A_2. Figure 3.3 does not prescribe any implementation, so the system could be developed using different modelling techniques[1]. This chapter focuses on the classic real-time theory originated by the seminal paper of Liu and Layland (1973), where the system is modelled as a set of tasks, which are scheduled and executed on a single processor. This basic idea has successively extended to more complex applications and architectures. One strong point of this approach is that the theoretical approach is supported by Real-Time Operating System (RTOS), which implement the priority-driven scheduling assumption used in the real-time theory.

The chapter continues by giving a short introduction to RTOSs in Section 3.2 to better understand the underlying mechanisms of the underlying implementation, which obviously affects the more abstract theoretical model. A RTOS is also an important part of the embedded computing platform, which includes both the hardware platform, discussed in Chapter 2, but also the software platform. Section 3.3 presents the real-time system model.

[1]System modelling will be part of the IL2212 Embedded Software course

3.2. Real-Time Operating Systems

A previously illustrated in Figure 3.3 embedded system have to deal simultaneously with different activities. Many embedded systems are implemented on a single processor, which can only run a single thread of execution. This means that it in general is very difficult to write a program, which can react to simultaneous input events, because the program executes in a purely sequential order. A single-thread processor cannot provide true parallelism, but an operating system can be used to give the impression of parallelism, although there is still only a single thread of execution. In this context the terms concurrency and parallelism are important.

⚠ Concurrency and Parallelism

Concurrency Several programs can *potentially* start, run and complete in overlapping time. It does not mean that the programs will ever run in overlapping time. Concurrency is a property of the more abstract programming model level. If a model provides concurrency, it can provide true parallelism on a parallel execution platform.

Parallelism Two or more programs execute in overlapping time, which is normally achieved by running the programs on different processors.

Thus, an Operating System (OS) can provide concurrency in the form of concurrent tasks or processes even if it runs on a sequential processor. The idea of an OS is to encapsulate sequential programs as concurrent processes and to schedule them at run-time, so that an impression of parallelism appears. In contrast to normal OSs, a RTOS also provides a predictable scheduler, which is a prerequisite to determine, if each task will always execute before its deadline.

In a desktop OS the term *process* is used for a concurrent sequential program that has its own protected address space, which does not interfere with the address space of other processes in the system. The protection of address space requires extensive hardware and software resources. Most embedded system RTOSs use instead the *threads* for the concurrent sequential programs, which are called tasks in RTOSs. All threads share the same address space. This means that RTOS tasks can also overwrite memory locations that another thread is using, which can lead to inconsistent data. The advantage of threads is that the implementation requires very few resources, and also that communication between threads is directly supported using shared variables in the common address space.

Tasks can be created and deleted dynamically during run-time. However, it is far more difficult to analyse a dynamic system than a static system, where all tasks are present from the system start and neither new tasks are created nor existing tasks are deleted during run-time. Thus, safety-critical real-time systems uses in general only static tasks for which well-established analysis methods exist, see Chapter 3.3.

There are two ways to support concurrent tasks and real-time behaviour from a programming language perspective. The first option is to integrate the support directly within the programming language. The programming language Ada uses this approach and supports concurrent tasks as an integrated part of the language and also provides a standardised real-time annex for the design of real-time systems. An overview of the Ada language is given in Appendix D. The second option is to provide support for concurrent tasks and real-time on top of an existing programming language. This is the approach taken by RTOSs, which in general are developed for the programming language C. There exist many different RTOSs. Although they share the same principles, it is difficult to port a program developed for one RTOS to another one, because each RTOS has its own syntax and also supports a varying set of communication objects and mechanisms. Still, the principle mechanisms to support concurrency and real-time are very similar in different RTOSs and also in the language Ada. The following discussion will focus on the main principles, which enable to support real-time systems. The term RTOS will be used as a general term and also includes the programming language Ada.

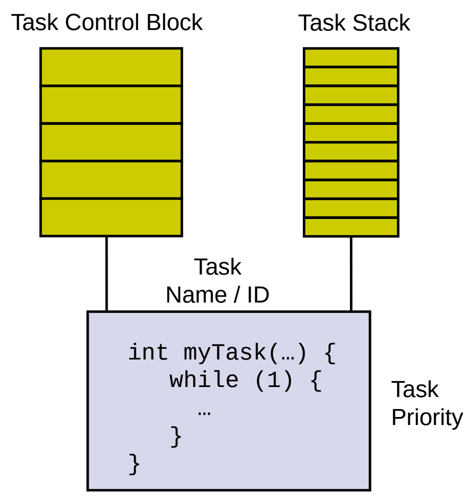
A RTOS schedules the tasks in a predictable manner. This enables to apply real-time models and analysis methods to determine, if tasks will their deadlines. A RTOS provides the following components and services.

- The *scheduler* determines which task shall run at each timing instant.
- *Communication objects and mechanisms* like semaphores or message queues are used for a controlled communication between tasks.
- Additional *services* for time management, memory management, interrupt handling or device management, might be provided.

The level of features and services provided by a RTOS varies between different RTOSs. RTOSs for small embedded CPUs provide only the most essential features, while more complex RTOSs can provide communication services like network protocols.

Tasks are used in a RTOS to create an independent thread of execution. Several tasks build a concurrent program, in which the scheduler schedules the tasks at run-time. Most real-time scheduling algorithms use priorities associated to tasks as a base for their scheduling decisions. The implementation of tasks requires data structures in form of a Task Control Block (TCB) and a task stack to save the state of the task, when it is not scheduled. Figure 3.4 illustrates this data structure.

The scheduler applies an algorithm that decides, which task shall execute at a certain time instant. This scheduling creates the impression of "parallelism" on a single processor, although there still is only a single thread of execution, i.e. only one task can run at any time instance. There are different strategies, when the scheduler shall be activated. This can happen at when a task is released or completed, at certain time instances controlled by a timer, or at the occurrence of a specific system call. When the scheduler runs, no other tasks can run. When the scheduler is activated, it decides, which tasks should be scheduled next to run on the processor. If a new task τ_{new} shall be scheduled, the state, called *context*, of the previously running task τ_{old} needs to be saved in its current state in memory, and the current context of task τ_{new} needs to be loaded from the memory. This is called a *context switch*, after which the new task is



```
int myTask(…) {
   while (1) {
      …
   }
}
```

Figure 3.4.: A task is created with an associated TCB and a task stack

activated and dispatched, so that it can run on the processor. A context switch implies a certain overhead, during which no useful work can be executed. The context switch is

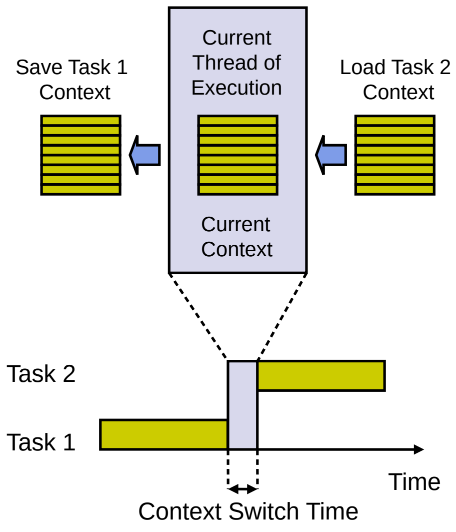


Figure 3.5.: During a context switch, the context of the previously running task is replaced with the context of the new running task. The context of the previously running tasks has to be saved.

illustrated in Figure 3.5.

There are different principal approaches to decide when the scheduler should be invoked. In *cooperative multitasking* or non-preemptive multitasking, scheduling is based on cooperation, where the running task needs to explicitly give up the control of the CPU before the scheduler is allowed to schedule another task. For real-time systems cooperative multitasking is very difficult to use, since the responsiveness of the system can be very low. An important task might have to wait for a long time until a low priority task gives up the control of the CPU. Thus, to guarantee that an important task still always can meet its deadline, the designer has to make sure that the execution time of a low-priority task is not too long.

Preemptive multitasking is the normal choice for a RTOS, since it simplifies to guarantee and calculate the real-time behaviour of the tasks. In preemptive multitasking, tasks can be *preempted* ("interrupted") at any time instance. The idea is that always the highest priority task, which is ready for execution, shall be scheduled to run on the processor. Thus, a running task is preempted, if a higher priority task becomes ready. In this moment the scheduler is invoked, a context switch is executed, and the higher priority tasks is scheduled and dispatched. Preemptive multitasking increases not only the responsiveness of the system, but also simplifies to calculate the response time of high priority tasks. On the negative side, the designer has now to be more careful, when creating programs, since a program can be preempted at any time and a shared variable might get corrupted as can be shown in the program example of Listing 3.1. In this

Listing 3.1. Non-reentrant function `Swap()`

```c
int Temp; /* Global Variable */

void Swap(int *x, int *y)
{
  Temp = *x;
  *x = *y;
  *y = Temp;
}
```

example, the function `Swap()` uses a global variable `Temp`. If the running thread is preempted after `Temp = *x` in the function `Swap()`, and another thread calls the function `Swap()` and executes it, `Temp` will be changed and the original task will get a wrong value for `y`, when rescheduled. Such a function is called a *non-reentrant* function. A *reentrant* function is a function, where multiple invocations can safely run concurrently, and where the function can be preempted or interrupted during execution and can then safely be called again (re-entered) before the previous invocation has finished its execution.

Listing 3.2 shows a reentrant version of the function `Swap()`, where the variable `Temp` is declared locally, and thus saved on the stack instead of in a global memory location. Thus, the variable `Temp` will not be shared between different tasks in an RTOS.

Special care has to be taken, when shared global variables are used in a concurrent program. A *thread-safe* program ensures that by accessing shared global variables in disciplined manner, threads do not interfere with each other in an unintended way. Besides the use of reentrant functions, thread safety can be achieved by the proper use of communication objects and mechanisms provided by the RTOS.

A task in a RTOS can be in different states, which are visualised in Figure 3.6, which shows five possible states of a task.

Listing 3.2. Reentrant function `Swap()`

```c
void Swap(int *x, int *y)
{
   int Temp; /* Local Variable */

   Temp = *x;
   *x = *y;
   *y = Temp;
}
```

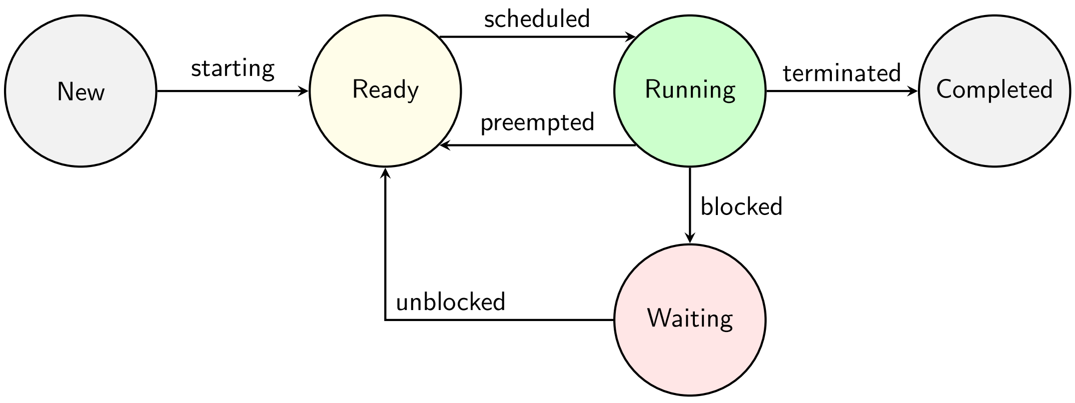


Figure 3.6.: A task in a real-time operating system can be in different states.

- *New:* Task has been created
- *Ready:* Task is ready for execution, but has not been scheduled
- *Running:* Task is scheduled
- *Waiting:* Task waits for resources and is not ready for execution
- *Completed:* Task has executed its final statement

During the life-time of a task, a task is always in one of the five states. After the creation of a task, the task will be moved to the state Ready. In a priority-driven operating system, the highest priority task, which is ready (or running) will be the task that is scheduled on the processor and put into the state Running. On a single-thread processor, at each time instance there is only one task, which can be in running mode. If the running tasks lacks resources and cannot proceed, e.g. the task tries to access an item from an empty message queue, the running task is blocked and moved to the state Waiting. Once the blocking condition is removed, the task is moved to the state Ready. A running task is preempted and moved back to the state Ready, if task with a higher priority becomes ready and moved to the state Running. If a task runs to its last statement, the task is terminated and moves to the state Completed.

As mentioned earlier, simultaneous access to a shared resource can lead to unintended behaviour, which could result in disastrous consequences. Figure 3.7 illustrates this

```c
void task_1()
{
 printf("Hello from Task 1!");
}
```

```c
void task_2()
{
 printf("Hello from Task 2!");
}
```

Possible Output: `Hello from TaHello from Task 1!sk 2!\verb`

Figure 3.7.: Simultaneous access to a shared resource can lead to non-intended behaviour

situation by two tasks that want to use the I/O-device. If task 1 preempts task 2 while task 1 has not completed the printing of the full message, the resulting output makes very little sense. Here, the I/O-device can be seen as a *serially reusable shared resource*, where a task that uses this resource shall not be interrupted. The code that accesses a serially reusable resource is called a *critical section* or *critical region*. In order to be able to ensure a thread-safe access to serially shared resources, different communication objects and mechanisms based on abstract data types have been developed to control the access to a shared resource. In the following a few of these communication objects are introduced. The text book of Ben-Ari (2006) gives a good introduction over concurrent and distributed programming, which includes the principles of different communication mechanisms.

3.2.1. Semaphore

A semaphore S can be viewed as a record with two fields (Ben-Ari, 2006):

- a non-negative integer $S.V$, and
- a set of processes (tasks) $S.L$

A semaphore is initialised with a value $k \geq 0$ for $S.V$ and with the empty set $\emptyset$ for $S.L$. A process p can use two statements that are defined on a semaphore: `wait(S)` and `signal(S)`[2]. A semaphore, where $S.V$ can take arbitrary non-negative values is called *general semaphore*, a semaphore, where $S.V$ can only take the values 0 and 1 is called *binary semaphore*.

The operation `wait(S)` is defined in Algorithm 3.1. If the value of $S.V$ is zero, then the process `p` is blocked, moved to the state Waiting, and is added to the set of processes waiting for S. This is called, that the process `p` is blocked on the semaphore S. If the value of $S.V$ is non-zero, then $S.V$ is decremented and the process `p` continues execution.

The operation `signal(S)` is defined in Algorithm 3.2. If $S.L$ is empty, then the value

[2] Originally Dijkstra used `P(S)` for `wait(S)` and `V(S)` for `signal(S)`

Algorithm 3.1.: Definition of `wait(S)`

```
**if** $S.V > 0$ **then**
    $S.V \leftarrow S.V - 1$
**else**
    $S.L \leftarrow S.L \cup$ p
    p.state $\leftarrow$ blocked
**end if**
```

Algorithm 3.2.: Definition of `signal(S)`

```
**if** $S.L = \emptyset$ **then**
    $S.V \leftarrow S.V + 1$
**else**
    ($q$: arbitrary process in $S.L$)
    $S.L \leftarrow S.L - \{$q$\}$
    q.state $\leftarrow$ ready
**end if**
```

of $S.V$ is incremented. If $S.L$ is not empty, an arbitrary process q in the waiting list $S.L$ is unblocked and is put into the state Ready[3]. The states of a binary semaphore and general semaphore are illustrated in Figure 3.8, where the state transition from the

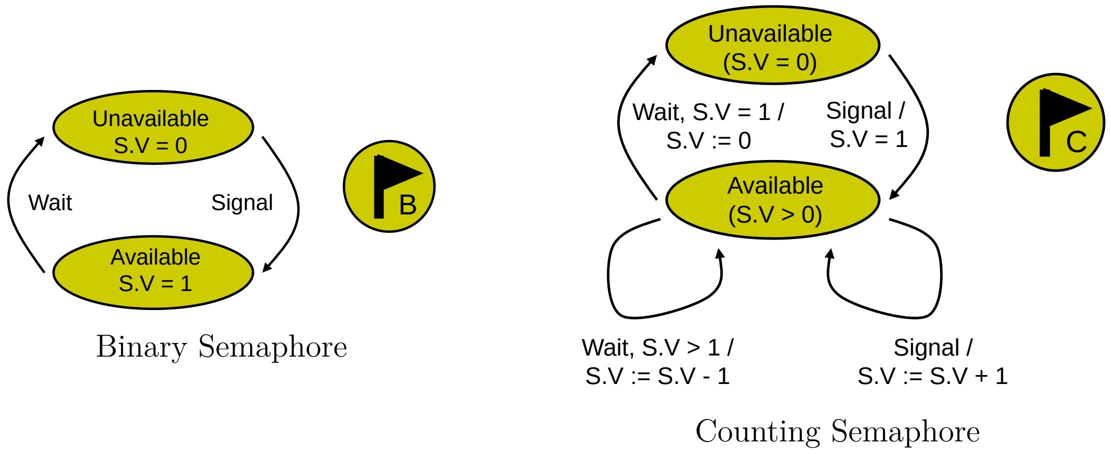


Figure 3.8.: States of a Binary and Counting Semaphore

state signal is only conducted, if there is shown for the case that there is no process in the waiting list, when the statement `signal(S)` is executed.

Semaphores can be used to solve the *critical section problem*, which aims at solving the access to critical sections. The problem is defined as follows. Each of N processes is executing in an infinite loop a sequence of statements that can be divided into two

[3]In a RTOS the process in the waiting list with the highest priority is unblocked and moved to the state Ready.

subsequences, the critical section (CS) and the non-critical section (NCS). The following properties need to be fulfilled by the implementation.

- *Mutual exclusion.* Statements from the critical section of two or more processes must not be interleaved
- *Freedom from deadlock.* If one or more processes are trying to enter their critical section, then *one* of them must eventually succeed
- *Freedom from (individual) starvation.* If *any* process tries to enter the critical section, then *that* process must eventually succeed

The critical section problem is difficult to solve on a bare machine without support for interprocess communication mechanisms, but it can be solved for two processes with a binary semaphore S that is initialised to $S = \{1, \emptyset\}$ as shown in Table 3.1. For more

Table 3.1.: Implementation of critical section problem using semaphores

P_1	P_2
`loop forever`	`loop forever`
`  non-critical section (NCS)`	`  non-critical section (NCS)`
`  wait(S)`	`  wait(S)`
`  critical section (CS)`	`  critical section (CS)`
`  signal(S)`	`  signal(S)`

than two processes, the requirement *freedom from (individual) starvation* cannot be guaranteed, because the there is no guarantee that a blocked process will ever proceed using the definition of a semaphore according to Algorithms 3.1 and 3.2, which is called a *weak semaphore.*

The requirement *freedom from (individual) starvation* for two or more processes can be guaranteed, if instead of a weak semaphore, a *strong semaphore* is used as defined in Algorithms 3.3 and 3.4. The strong semaphore implements a FIFO-fairness regarding the access to the semaphore, which ensures that a process that waits for a semaphore will eventually access it. The FIFO-fairness is implemented by instead of using an unsorted set for processes waiting for a semaphore, a sorted queue is used.

Algorithm 3.3.: Definition of `wait(S)` for a strong semaphore

```
if $S.V > 0$ then
  $S.V \leftarrow S.V - 1$
else
  $S.L \leftarrow$ append($S.L$, p)
  p.state $\leftarrow$ blocked
end if
```

Algorithm 3.4.: Definition of `signal(S)` for a strong semaphore

```
if $S.L = \emptyset$ then
    $S.V \leftarrow S.V + 1$
else
    $q \leftarrow$ head($S.L$)
    $S.L \leftarrow$ tail($S.L$)
    q.state $\leftarrow$ ready
end if
```

It is important to note that a semaphore does not support the concept of *ownership*. This means that any process can execute the command `signal(S)`. It is not required that the process had previously successfully executed the command `wait(S)`.

The implementation of the semaphore requires very few resources, a semaphore control block, a unique identity, an initial value and a list for the waiting tasks. This is illustrated in Figure 3.9.

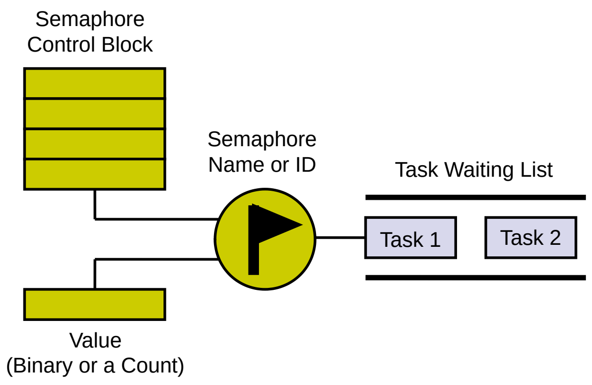


Figure 3.9.: A semaphore requires a semaphore control block, a value, and a list for the waiting tasks

3.2.2. Mutex

A *mutex* can be viewed as a special kind of semaphore that enables mutual exclusive access to a shared resource. In contrast to a semaphore a mutex includes the concept of ownership. Only the process that has previously locked the mutex `M` by successful execution of the command `lock(M)` (corresponding to `wait(S)`) can also unlock the mutex with the command (`unlock(M)` (corresponding to `signal(S)`). A mutex is created in the unlocked state, and the command `lock(M)` only succeeds, if the mutex is in the unlocked state. Otherwise the process is blocked on the mutex `M`. The states of a mutex are illustrated in Figure 3.10.

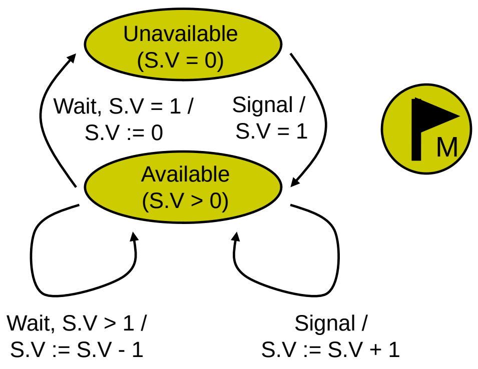


Figure 3.10.: States of a mutex

3.2.3. Message Queues

A *message queue* is buffer that can be used by the tasks to communicate by sending and receiving messages. The sent messages are temporarily stored in the message queue until another task receives the message. The messages are normally sent and received in FIFO-order. The message queue buffer has a certain buffer size and can receive messages of a certain data type or size.

The use of a buffer in the message queue decouples the sender of a message from the receiver. Send and receive commands can be *blocking* or *non-blocking*. The states of a message queue are illustrated in Figure 3.11.

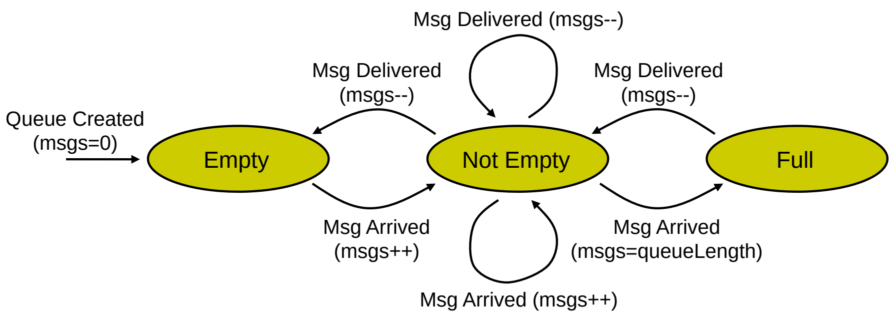


Figure 3.11.: State diagram for message queue

A *blocking send* command succeeds, if the message queue is not full, and in that case the message is added to the message queue. If the message queue is full, then the sender will be blocked on the message queue, and will be moved to the state Waiting. A *blocking receive* command succeeds, if the message queue is not empty. In that case, the task issuing the receive command receive the message and can continues its execution. If the message queue is empty the receiving task is blocked and will be moved to the state Waiting.

If a tasks uses a *non-blocking send* command, the task can continue its execution even if the message queue is full. In this case, the task will not send its message and is normally informed by an error message, that the send command did not succeed. In the same way, a task issuing a *non-blocking receive* command can continue its execution, if the message queue is empty.

The implementation of the message queue requires a queue control block, a queue name, an ID, memory buffers, a queue length, a maximum message length, and one or more task-waiting lists as illustrated in Figure 3.12.

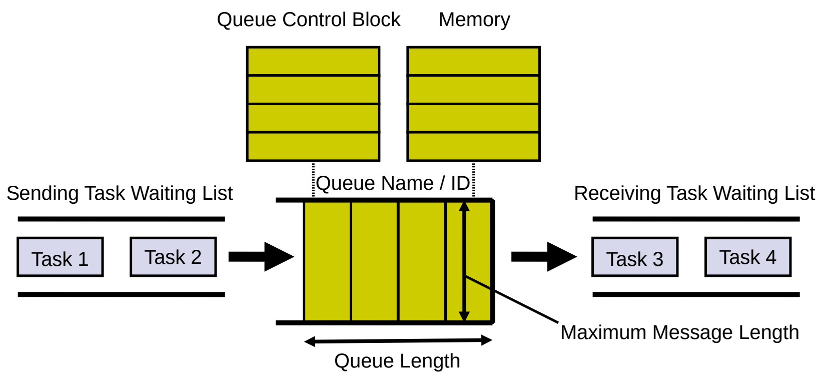


Figure 3.12.: Data structure for the implementation of a message queue

3.2.4. Soft Timer

Timers are an integral part of an embedded system. *Hardware timers* are available as peripheral components to an embedded processor core, but many real-time operating systems provide *soft timers* as services to the user. Usually multiple soft timers can run at the same time.

Soft timer needs to be triggered by an external signal, a *timer tick*, which is generated from a hardware timer. The resolution of the soft timer depends on the frequency of the external timer tick. A soft timer can usually be operated in two modes.

- Periodic: Timer runs for some period and restarts
- One-Shot: Timer runs once for a defined period

An action can be defined to be executed, when period is expired. The action is specified by a callback function.

Soft timers can be used to implement a periodic task τ_i. A periodic soft timer signals a semaphore S every period T_i as illustrated in Figure 3.13 for the tasks $\tau_1(8,4)$ and $\tau_2(3,1)$. The periodic task contains the computation function and a `wait(S)` statement for the semaphore S. Thus, after executing its computation function, the task is blocked on the semaphore S and will be first released after the soft timer executes the `signal(S)` statement. The code for the task is illustrated in the following code fragment.

```
void task() {
  computation_function();
  wait(S)
```

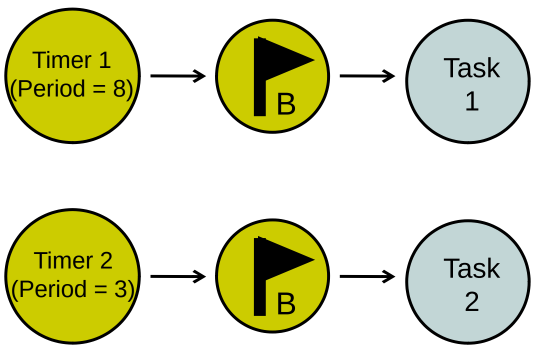


Figure 3.13.: Soft timers can be used to implement periodic tasks

Please observe, that the worst case execution time C_i cannot be controlled by the RTOS, but must be ensured by the designer.

3.3. Real-Time System Model

The classic real-time theory originated from the seminal paper of Liu and Layland (1973) uses a) a *workload model* describing the applications in the system; b) a *resource model* describing the resources available to the system; and c) scheduling *algorithms* defining how the applications uses the resources at all times.

Real-time applications can be described as a set of *tasks*, where each task contains a set of related *jobs* or *task instance*, which jointly provide a system function. A job is the unit of work that is scheduled and executed by the system.

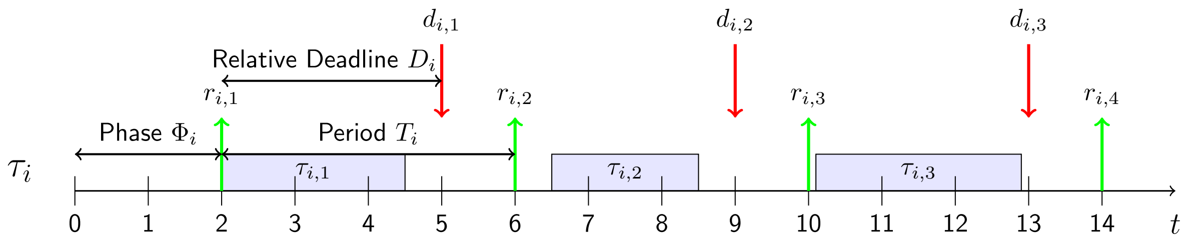


Figure 3.14.: Simulation of the plant of Figure 3.1, where all jobs meet their deadline

Figure 3.14 shows a simulation of the plant of Figure 3.1, where all jobs meet their deadline.

The real-time model defines many terms, variables and symbols. Unfortunately, this terms and notations are not fully standardised, and different terms are used in scientific publications and text books. The course will mainly use the notation used in (Buttazzo, 2011), but will also introduce the notation used in (Liu, 2000).

Real-Time Systems: Terminology, Notations, Seminal Papers

The Technical Committee on Real-Time Systems provides a web page with definitions for important terms and notations within the area of real-time systems (TCRTS-Definitions). They also provide a web page with seminal papers within this area (TCRTS-Seminal).

The following list presents important terms and notations used by the real-time community.

- A *task* i is denoted τ_i in (Buttazzo, 2011) and T_i in (Liu, 2000).
- A *job* k of a task i is denoted $\tau_{i,k}$ in (Buttazzo, 2011) and $T_{i,k}$ in (Liu, 2000).
- The *period* of a task i is denoted T_i in (Buttazzo, 2011) and P_i in (Liu, 2000).
- The *computation time* or *execution time* of a job within task i is denoted C_i (Buttazzo, 2011) and e_i in (Liu, 2000). It is the amount of time required to complete the job, when it executes alone and has all resources it needs. Especially for hard-real time systems C_i typically denotes the maximum execution time, the so called WCET will be used to be able to give save bounds.
- The *phase* of a task i is denoted ϕ_i.
- The *release time* r_i or *arrival time* a_i is the time when the task i becomes ready for execution.
- The *absolute deadline* is the time, when the k-th job $\tau_{i,k}$ in task i needs to be completed.
- The *relative deadline* D_i is the difference between the absolute deadline $d_{i,k}$ and the release time $r_{i,k}$ for a job $\tau_{i,k}$ within a task τ_i. The relative line is the same for all jobs in a task τ_i.
- The *start time* $s_{i,k}$ of a job $\tau_{i,k}$ in task i is the time at which job $\tau_{i,k}$ starts to execute.
- The *finishing time* or *completion time* $f_{i,k}$ of a job $\tau_{i,k}$ in task i is the time at which job $\tau_{i,k}$ completes it execution.
- The *response time* of a job $\tau_{i,k}$ is defined as $f_{i,k} - s_{i,k}$ and denoted $R_{i,k}$ in (Buttazzo, 2011) and $w_{i,k}$ in (Liu, 2000).

The *tardiness* or exceeding time $E_{i,k}$ and the lateness $L_{i,k}$ measures how late a job $\tau_{i,k}$ completes respective to its deadline. These terms are useful for soft real-time systems.

- The lateness $L_{i,k} = f_{i,k} - d_{i,k}$ can be positive or negative.
- The tardiness $E_{i,k} = \max(0, L_{i,k})$ can only be positive.

Tasks can be *periodic*, *aperiodic* or *sporadic* according to the following definitions.

- A *periodic task* executes periodically, hard deadline is assumed
- An *aperiodic task* executes jobs on demand, jobs have a soft deadline
- A *sporadic task* executes jobs on demand, jobs have a hard deadline

The chapter will focus on periodic tasks. A periodic task τ_i is defined as a tuple $\tau_i(\phi_i, T_i, C_i, D_i)$. The notation $\tau_i(\phi_i, T_i, C_i)$ implies that $D_i = T_i$, and the notation $\tau_i(T_i, C_i)$ implies that $\phi_i = 0$ and $D_i = T_i$.

3.4. Scheduling Algorithms

One the tasks have been modelled and the architecture has been defined, tasks have to be assigned to and scheduled on computing resources. Thus in general, the task assignment and scheduling problem can be defined as to identify a task assignment accompanied by a *feasible schedule*, where all design constraints are met.

RTOSs use *dynamic* scheduling algorithms, which perform scheduling at run-time. It is also possible to use *static* scheduling algorithms, where the schedule is calculated before system start and stored in tables in memory. This is also called *timeline scheduling*.

The text will focus on the most popular dynamic scheduling algorithms, which are used for RTOSs. These algorithms use priorities in order to decide, which task is scheduled onto which processor at what time instant. An *optimal* algorithm produces a feasible schedule, if it exists. The discussion only focuses on algorithms for single processors with single thread of execution, which means that task assignment is trivial.

The basic real-time model as defined by Liu and Layland (1973) uses a set of assumptions. Tasks are periodic and are characterised by $\tau_i(T_i, C_i)$ where a) each job in task $\tau_i(T_i, C_i)$ has the computation time C_i; b) jobs in task $\tau_i(T_i, C_i)$ are released periodically with a period T_i at the beginning of period; c) the period and the relative deadline are equal, $T_i = D_i$; d) there is no initial phase, $\phi_i = 0$. All jobs in a task have to complete before their deadline. Tasks can be preempted. There is no dependency between tasks. The scheduler can run at any time instance. The context-switch time is zero. Tasks are scheduled on a single-thread processor. These assumptions idealise many practical factors, but enable an elegant model to solve the scheduling problem.

3.4.1. Rate-Monotonic Scheduling

The Rate-Monotonic (RM) algorithm assigns fixed task priorities in reverse-order of the period length, i.e. the shorter the period, the higher the priority. This is based on the idea, that tasks that require more frequent action should receive higher priority. The RM algorithm uses preemption, and fixed (static) priorities, which means that all jobs in a task have the same priority. The scheduler will always schedule a job of the task that is ready and has the highest priority.

In the first example, the RM algorithm shall be used to scheduling the task set $\tau_1(4,3)$, $\tau_2(8,2)$. Task τ_1 has the higher priority due its shorter period, i.e. $P_1 > P_2$, where P_i denotes the priority of task τ_i. Figure 3.15 shows the resulting schedule for this set

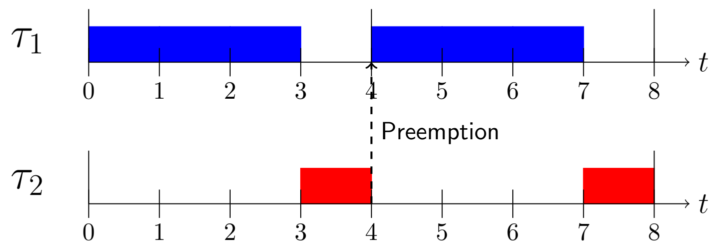


Figure 3.15.: Rate-monotonic schedule for the task set: $\tau_1(4,3)$, $\tau_2(8,2)$

of tasks. Both tasks τ_1 and τ_2 are released at time $t = 0$. Thus they both have the state Ready. The scheduler selects the highest priority task τ_1, and schedules its first job $\tau_{1,1}$ for execution, i.e. into the state Running. The job $\tau_{1,1}$ has a computation time $C_1 = 3$, which means that it finishes its execution at time $t = 3$ and is moved to the state *Waiting*. Then, at time $t = 3$, task τ_2 is the ready task with the highest priority and is scheduled. Thus, the job $\tau_{2,1}$ is scheduled. It has a computation time $C_2 = 2$, but can only run for 1 time unit. At time $t = 4$, the second job $\tau_{1,2}$ of task τ_1 is released and moved to the state Ready. Thus task τ_1 is now the ready task with the highest priority, so at time 4 job $\tau_{2,1}$ is preempted and moved back to state Ready, and instead job $\tau_{1,2}$ is moved to the state Running. Job $\tau_{1,2}$ runs for $C_1 = 3$ time units and completes at time $t = 7$. At that time job $\tau_{2,1}$ is the only ready task and will complete its execution at time $t = 8$. At time $t = 8$ both tasks are released again and both tasks are exactly in the same state as at time $t = 0$. Thus, the schedule will now repeat every 8 time units. The term *hyperperiod* H is defined as the time interval until a schedule repeats itself, and its length can be calculated as the least common multiplier of the periods of all tasks in the task set. Thus, here the hyperperiod can be calculated as $H = lcm(T_1, T_2) = lcm(4, 8) = 8$. Thus, in order to detect, if all tasks meet their deadlines, the set of task only needs to be simulated for one hyperperiod. Thus, the schedule of Figure 3.15 shows that both tasks τ_1 and τ_2 meet their deadlines, since the schedule covers one full hyperperiod. The system has a feasible schedule.

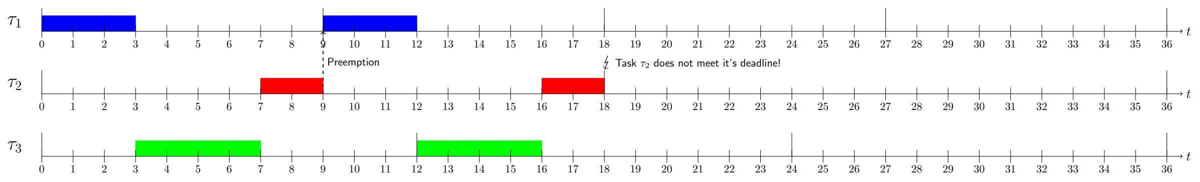


Figure 3.16.: Rate-monotonic schedule for the task set: $\tau_1(9,3)$, $\tau_2(18,5)$, $\tau_2(12,4)$

Figure 3.16 shows a second example with the set of tasks $\tau_1(9,3)$, $\tau_2(18,5)$, $\tau_2(12,4)$. Here task τ_2 does not meet its deadline at time $t = 18$, because it has only executed for 4 time units instead of $C_2 = 5$ time units. Thus, the RM algorithm cannot find a feasible schedule for this set of task.

3.4.2. Earliest Deadline First Algorithm

The RM algorithm is restricted because the priorities of all task is fixed at run-time and all jobs in a task inherit this same priority. The Earliest Deadline First (EDF) algorithm is a more dynamic algorithm, which assigns jobs priorities based on their deadlines. The job with the shortest deadline has the highest priority. Thus, in the Earliest Deadline First (EDF) algorithm jobs in a task can have different priorities, which requires that the scheduler has information about the deadlines of all jobs.

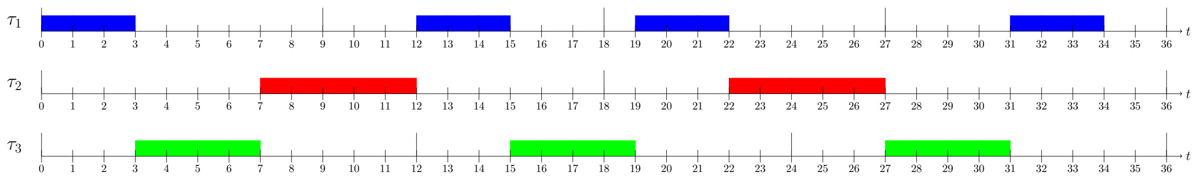


Figure 3.17.: Earliest deadline first schedule for the task set: $\tau_1(9,3)$, $\tau_2(18,5)$, $\tau_2(12,4)$

Figure 3.17 uses the same task set, $\tau_1(9,3)$, $\tau_2(18,5)$, $\tau_2(12,4)$, as the RM example from Figure 3.16. But in contrast to the RM algorithm, the EDF algorithm is able to generate a feasible schedule. The reason is that more urgent tasks, i.e. the tasks close to their deadline, get a higher priority.

At time $t = 0$ all three tasks are released, and because the job of task τ_1 has the closest deadline, it has the highest priority and is scheduled. In contrast to the RM schedule of Figure 3.16, task τ_2 does not miss its deadline, because the closer the deadline, the higher its priority. As the schedule shows, all tasks meet their deadlines. Please note, that if two tasks have the same deadline, the scheduler can schedule any of these tasks. For instance at time $t = 9$, both τ_1 and τ_2 can be scheduled, because both tasks have the same deadline at $t = 18$.

Optimality of EDF (Theorem 4.1 in Liu (2000))

The EDF algorithm is optimal under the following condition.

> When preemption is allowed and jobs do not content for resources, the EDF algorithm can produce a feasible schedule of a set **J** of jobs with arbitrary release times and deadlines on a processor if and only if **J** has a feasible schedule.

The EDF algorithm is not optimal, if jobs are non-preemptible or more than one processor is used.

3.4.3. Comparison: Fixed- and Dynamic Priority Algorithms

Fixed- and dynamic-priority scheduling algorithms have different properties. The EDF algorithm outperforms the RM algorithm with respect to schedulable utilisation (see Section 3.5). Nevertheless, the behaviour of dynamic algorithms is much more difficult to predict, since tasks change their priority dynamically. In contrast, in a fixed-priority algorithm there is one task, which will always have the highest priority. This means that jobs in this task will always run, if it is ready and has all required resources. Due to this clear assignment of priorities, fixed-priority algorithms are conceptually easier to analyse.

In addition, fixed-priority algorithms are also much easier to implement. When the scheduler is executed, it only has to check, which tasks are ready for execution, and then to schedule the one with the highest priority. Due to its efficient and lightweight implementation the RM algorithm is the natural choice for most RTOSs. Instead, an EDF implementation requires to monitor the current time, and to compute the next absolute deadline for each job that is ready. Then, the ready job with the closest deadline is selected and scheduled for execution. EDF requires a clearly larger implementation effort and is thus too expensive to be used for many embedded RTOS.

3.5. Schedulability Tests

Simulation of the schedule for one hyperperiod is one possible technique to determine, if a feasible schedule exists for a given task set. However, this technique is difficult to apply for a large set of tasks or task sets, for which the hyperperiod is a large number. More efficient methods are needed.

A *schedulability test* is a mechanism that proofs that all deadlines are met, when scheduling with a particular algorithm. It can also be used in a dynamic system to decide, if new jobs can be accepted.

3.5.1. Schedulable Utilisation

One important method is to use the *utilisation* of the task set to determine, if a set of tasks is schedulable. The utilisation of a periodic task τ_i is defined as $u_i = \frac{C_i}{T_i}$, and the total utilisation U of a set of tasks $\Gamma = \{\tau_1, \dots, \tau_n\}$ is defined as $U = \sum\limits_{i=1}^{n} u_i$.

Liu (2000) defines the term *schedulable utilisation* U_{ALG} of an algorithm ALG.

> A scheduling algorithm can feasibly schedule any set of periodic tasks on a processor if the total utilisation of the tasks is equal to or less than the *schedulable utilisation* U_{ALG} of that algorithm!

Schedulability tests can be developed based on U_{ALG}. An algorithm is optimal, if $U_{ALG} = 1$.

Obviously the first condition that has to be fulfilled for single-processor real-time systems is that a feasible schedule can only exist, if $U \leq 1$.

EDF is an optimal algorithms. Thus $U_{\mathrm{EDF}} = 1$ for the systems, where $T_i = D_i$ for tasks. Also $U_{\mathrm{EDF}} = 1$, if $D_i \geq T_i$. This is independent of the phase ϕ_i.

If there is at least one task, where $D_i < T_i$, in a system with $U \leq 1$ there might not be a feasible schedule any longer. For this situation, the text can be extended by using the *density* of a task. The density of a task τ_i is defined as $\delta_i = \frac{C_i}{\mathrm{min(D_i,T_i)}}$, and the density of a set of tasks as $\Delta = \sum \delta_i$. Theorem 6.2 in (Liu, 2000) states that "a system Γ of independent, preemptable tasks can feasibly be scheduled, if its density is equal or less to 1". This theorem is a sufficient, but not necessary condition. The system may nevertheless be feasible when its density is greater than 1.

For the rate-monotonic-algorithm Liu (2000) states in Theorem 6.11 one of the fundamental results of Liu and Layland (1973).

> A system of n independent, preemptable periodic tasks with relative deadlines equal to their respective periods can be feasibly scheduled on a processor according to the rate-monotonic algorithm if its total utilisation U is less than or equal to $U_{RM}(n) = n(2^{\frac{1}{n}} - 1)$.

This means that the schedulable utilisation of the RM algorithm U_{RM} depends on the number of tasks, but is at least $ln(2)$. Table 3.2 shows the schedulable utilisation for

Table 3.2.: Schedulable utilisation U_{RM} for different ns

n	$U_{\mathrm{RM}}(n)$
1	1.000
2	0.828
3	0.779
4	0.756
5	0.743
6	0.734
∞	$0.693 = ln2$

the RM algorithm for different ns.

Although the RM algorithm cannot be optimal due to the usage of fixed task priorities, it is optimal for a task sets, which are belong to *simply periodic systems.* A A system of periodic tasks is *simply periodic*, if for every pair of tasks T_i and T_k in the system and $T_i < T_k$, T_k is an integer multiple of T_i. The optimality of the RM algorithm is stated in Theorem 6.11 in (Liu, 2000).

> A system of simply periodic, independent, preemptable tasks whose deadlines are equal or larger than their periods is schedulable on one processor according to the RM algorithm, if and only if its total utilisation is equal or less than 1.

Thus, the set of tasks $\Gamma = \{\tau_1(2,1), \tau_2(4,1), \tau_3(8,2)\}$ is schedulable using the RM algorithm.

To summarise, given a set of n periodic tasks Γ that shall be scheduled with the RM algorithm on a single processor,

- a feasible schedule exists, if
  - $U(\Gamma) \leq U_{\mathrm{RM}}(n)$, or
  - $U_{\mathrm{RM}}(n) \leq U(\Gamma) \leq 1$ when the task set Γ forms a simply periodic system;
- a feasible schedule does not exist if $U(\Gamma) > 1$;
- a feasible *might* exist, if $U_{\mathrm{RM}}(n) \leq U(\Gamma) \leq 1$ when the task set Γ does not form a simply periodic system.

For the last case, where $U_{\mathrm{RM}}(n) \leq U(\Gamma) \leq 1$ and where the task set Γ does not form a simply periodic system, either task simulation can be used to decide, if there is a feasible schedule, or the schedulability test must be extended.

3.5.2. Time Demand Analysis

The time demand analysis or workload analysis has been designed to improve the test for fixed-priority-driven algorithms, like Rate-Monotonic Algorithm (RMA), to analyse the cases where $U_{RM} < U(\Gamma) \leq 1$. The main idea is to identify the possible worst case for each task analyse that case.

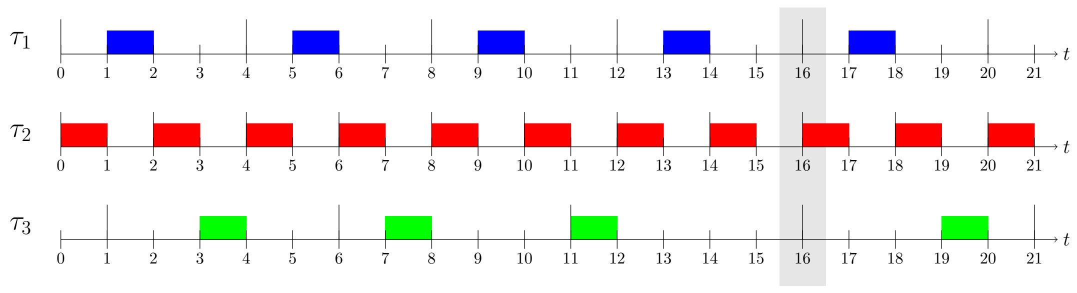


Figure 3.18.: Critical instance for task τ_3 is at $t = 16$

Figure 3.18 shows the schedule for the set of tasks $\tau_1 = (4,1), \tau_2 = (2,1), \tau_3 = (1,5,1,5)$. The longest response time for task τ_3 happens at time $t = 16$. This is the time instance, when all tasks with a higher priority are released at the same time as the task τ_i. Such a time instant is called the *critical instant* for the task τ_i, and is the time instant, when a task experiences its worst case response time.

Definition: Critical Instance (Theorem 6.5 (Liu, 2000))
In a fixed-priority system where every job completes before the next job in the same task is released, a *critical instant* of any task τ_i occurs when one of its job $\tau_{i,c}$ is released at the same time with a job in *every* higher-priority task, that is $a_{i,c} = a_{k,l_k}$ for some l_k for every $k = 1, 2, \ldots, i-1$.

The *time demand analysis* (or workload analysis) analyses the *time demand* (workload) for all tasks at their critical instance. The time demand $w_i(t)$ for a job of the task τ_i released at the critical instance at time t is calculated as

$$w_i(t) = C_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{T_k} \right\rceil C_k, \text{ for } 0 < t \leq T_i \tag{3.1}$$

For a task τ_i, the time demand needs to be compared with the available time for the period until the relative deadline D_i.

- If $w_i(t) \leq t$ for some $t \leq D_i$ where $D_i \leq T_i$, all jobs in τ_i can complete before its deadline.
- If $w_i(t) > t$ for all $0 \leq t \leq Di$ where $D_i \leq T_i$, this job in τ_i cannot complete by its deadline.

If all tasks in a task set Γ meet their time demand at their critical instant, then the task set Γ is schedulable.

NOTE: The time demand analysis uses the assumption that the critical instant occurs, but this must not be the case. Thus, the time demand analysis is pessimistic.

The time demand analysis can be illustrated graphically using the set of tasks $\tau_1 = (2,1), \tau_2 = (5,1), \tau_3 = (6,1)$ in Figure 3.19. The time demand for each task τ_i is given by the time demand function $w_i(t)$ of Equation (3.1). The intersection between the time demand function $w_i(t)$ and the linear function $y(t) = t$ gives the worst case response time W_i for task τ_i.

It is also possible to compute the worst case response time for a task τ_i iteratively based on Equation (3.1) starting with an initial guess for t, i.e. $t^{(0)}$. This iterative method can be automated by a computer algorithm.

$$t^{(l+1)} = C_i + \sum_{k=1}^{i-1} \left\lceil \frac{t^{(l)}}{T_k} \right\rceil C_k$$

If the left hand side and the right hand side of the equation give the same value, this value is the worst case response time. Otherwise, the iteration needs to continue with the right hand side value as the new value for t. The iteration can stop, if either the worst case response time is found through stabilisation of the equation ($t^{(i)} = t^{(i+1)}$, or when $t^{(i)} > D_i$, in which case the task τ_i will not meet its deadline!

The iterative approach is shown for the same set of tasks as in Figure 3.19, i.e. $\tau_1 = (2,1), \tau_2 = (5,1), \tau_3 = (6,1)$.

- Task τ_1: $w_1(t) = C_1 = 1$

  The worst case response time is $W_1 = 1$.

- Task τ_2: $w_2(t) = C_2 + \left\lceil \frac{t}{T_1} \right\rceil C_1$

  Here, iteration is needed. An initial guess would be $t^{(0)} = C_2 + C_1 = 2$, since both tasks need to execute at least once.

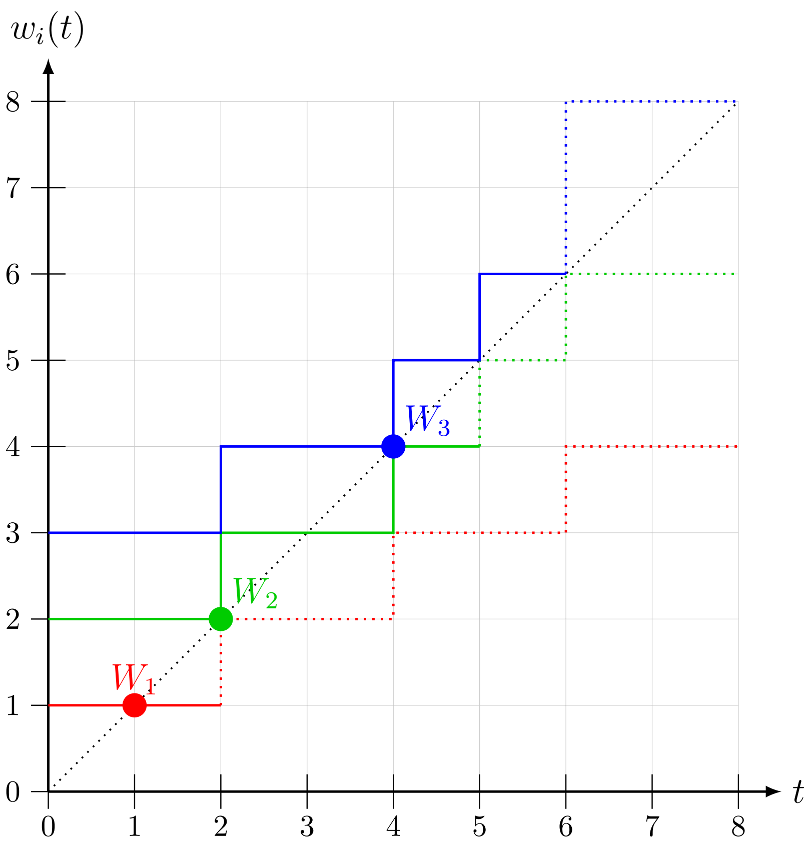


Figure 3.19.: Time Demand Analysis

$$\begin{aligned} t^{(1)} &= C_2 + \left\lceil \frac{t^{(0)}}{T_1} \right\rceil C_1 \\ &= 1 + \left\lceil \frac{2}{2} \right\rceil 1 \\ &= 2 \end{aligned}$$

Since $t^{(1)} = t^{(0)}$ a stable solution has been found and the worst case response time is $W_2 = 2$.

- Task τ_3: $w_3(t) = C_3 + \left\lceil \frac{t}{T_1} \right\rceil C_1 + \left\lceil \frac{t}{T_2} \right\rceil C_2$

  Again, iteration is needed. An initial guess would be $t^{(0)} = C_3 + C_2 + C_1 = 3$, since all tasks need to execute at least once.

$$\begin{aligned} t^{(1)} &= C_3 + \left\lceil \frac{t^{(0)}}{T_1} \right\rceil C_1 + \left\lceil \frac{t^{(0)}}{T_2} \right\rceil C_2 \\ &= 1 + \left\lceil \frac{3}{2} \right\rceil 1 + \left\lceil \frac{3}{4} \right\rceil 1 \\ &= 4 \Rightarrow \text{new iteration with } t^{(1)} = 4 \text{ needed!} \\ t^{(2)} &= 1 + \left\lceil \frac{4}{2} \right\rceil 1 + \left\lceil \frac{4}{4} \right\rceil 1 \\ &= 4 \end{aligned}$$

Since $t^{(2)} = t^{(1)}$ a stable solution has been found and the worst case response time is $W_3 = 4$.

3.5.3. Response Time Analysis

Another method, which also is based on the critical instant is the response time analysis. The longest response time R_i of a task τ_i is calculated as the sum of its computation time and the interference I_i, which is caused by the tasks with a higher priority than τ_i.

$$\begin{aligned} R_i &= C_i + I_i \\ I_i &= \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \\ \Rightarrow R_i &= C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \end{aligned}$$

As can be seen, the formula is very similar to the time demand analysis (Equation (3.1)), and the iterative solution works in a very similar way, i.e. starting with a guess on the response time R_i. Here, the principle is shown for the same task set $\tau_1 = (2,1), \tau_2 = (5,1), \tau_3 = (6,1)$.

- Response Time τ_1:
  - $R_1^{(0)} = C_1 = 1$
- Response Time τ_2:
  - $R_2^{(0)} = C_2 = 1$
  - $R_2^{(1)} = C_2 + \left\lceil \frac{R_2^{(0)}}{T_1} \right\rceil \cdot C_1 = 1 + \left\lceil \frac{1}{2} \right\rceil \cdot 1 = 2$
  - $R_2^{(2)} = C_2 + \left\lceil \frac{R_2^{(1)}}{T_1} \right\rceil \cdot C_1 = 1 + \left\lceil \frac{2}{2} \right\rceil \cdot 1 = 2$
- Response Time τ_3:
  - ...

3.5.4. Practical Factors

Any practical factors have so far not been taken into account. So far the following assumptions have been used, which are idealisations.

- High priority-job is never blocked by a low-priority job
- Every job is preemptable at any time
- Released jobs never suspend themselves
- Scheduling and context switch overhead is negligible
- The scheduler is event-driven and acts immediately upon event occurrences

- Tasks have distinct priorities
- Priorities in a fixed-priority system never change

Many of these practical factors can be taken into account by extending the time demand or response time analysis.

An additional term for the *blocking time* B_i can be used in the time demand or response time analysis function. The term B_i reflects the blocking time, which a high priority job can experience when a low priority job executes a non-preemptable section, for instance a critical section protected by a semaphore.

- Time Demand Analysis: $w_i(t) = B_i + C_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{T_k} \right\rceil C_k, for 0 < t \leq T_i$
- Response Time Analysis: $R_i = B_i + C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j$

3.6. Resource Access Protocols

3.6.1. Resource Model

So far, the discussion assumed independent tasks, but in practice tasks communicate via communication objects like semaphores, message queues or protected objects with each other. Thus tasks are often not independent. The discussion in the section is mainly based on the books of Liu (2000) and Buttazzo (2011). Most of the examples have been taken from the book of Liu (2000), while the terminology mainly follows the book of Buttazzo (2011).

Buttazzo (2011) defines a resource S[4] as “any software structure that can be used by a task to advance its execution”. A resource

- that is dedicated to a particular task is called a *private resource*;
- that can be used by more tasks is called a *shared resource*;
- that is protected against concurrent access is called a *exclusive resource*.

Resources can have several units as in the case of a counting semaphore. The following discussion only deals with exclusive resources with one unit. These exclusive resources are allocated to tasks on a *non-preemptive basis* and used in a *mutually exclusive manner*.

It is assumed that a lock-based concurrency control mechanism assumed to be used to enforce mutual exclusive access to resources.

- When a task wants to use a resource S_i, it executes a command `lock` $L(S_i)$ to request the resource.

[4]The symbol S is used, because the symbol R is already occupied to denote the response time. Also Buttazzo (2011) uses the semaphore as the main example for a resource.

- When a task no longer needs the resource S_i, it releases the resource by executing a command *unlock* signal $U(S_i)$.

Correspondence to critical sections protected by a semaphore

A critical section protected by a semaphore `S` is modelled as an exclusive resource S

- The command `wait(S)` is modelled by a `lock` command $L(S)$
- The command `signal(S)` is modelled by an `unlock` command $U(S)$

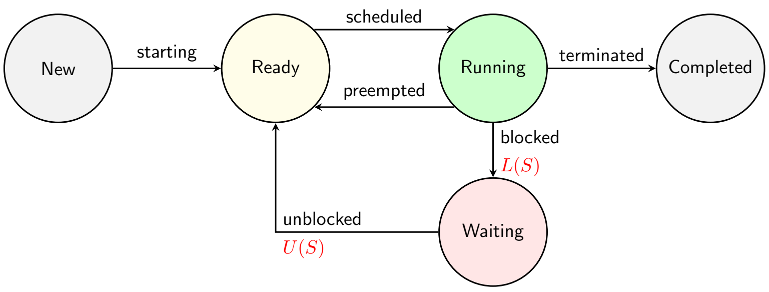


Figure 3.20.: A lock request $L(S)$ or an unlock command $U(S)$ can trigger tasks state changes.

Figure 3.20 illustrates how a lock request $L(S)$ or an unlock command $U(S)$ can trigger tasks state changes. When a lock request $L(S)$ fails, the requesting job is blocked and loses the processor. An unlock command $U(S)$ frees the resource and might cause that another task is unblocked. A task stays blocked until the scheduler grants the resources the task is waiting for.

A segment of a task τ_i that begins with a lock $L(S_k)$ and ends with a matching unlock $U(S_k)$ is called a *critical section*, and is denoted by $z_{i,k}$. The longest critical section of τ_i to a resource S_k is denoted by $Z_{i,k}$. The duration of $Z_{i,k}$ is denoted by $\delta_{i,k}$. The following pseudo code illustrates how a critical section is supported by a lock.

```
-- Critical Section
lock(S_k)
-- Access to exclusive resource S_k
unlock(S_k)
```

It is assumed that in *nested critical sections*, resources are always released in last-in-first-out order. Such nested critical sections are called *properly nested critical sections.*

- $z_{i,h} \subset z_{i,k}$ indicates that the critical section $z_{i,h}$ is entirely contained in $z_{i,k}$.
- A critical section that is not included in other critical sections is called an outermost critical section

Properly Nested Critical Section	Not Properly Nested Critical Section
`lock(S_1)`	`lock(S_2)`
`...`	`...`
`lock(S_2)`	`lock(S_1)`
`..`	`..`
`unlock(S_2)`	`unlock(S_2)`
`...`	`...`
`unlock(S_1)`	`unlock(S_1)`

3.6.2. Priority Inversion

Priority inversion occurs, when a low-priority job executes while a ready higher-priority job waits. The following example illustrates priority inversion.

There are three jobs J_1, J_2, J_3 with their feasible intervals (6,14], (2,17], (0,18]. The jobs are scheduled using EDF algorithm, and access critical sections with the exclusive resource S_k for the following durations: $\delta_{1,k} = 2$, $\delta_{2,k} = 4$, $\delta_{3,k} = 4$. Figure 3.21 shows

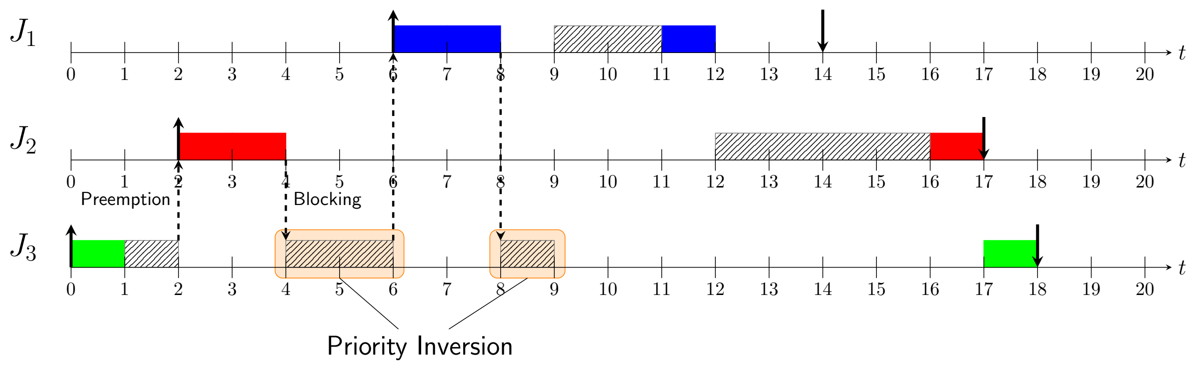


Figure 3.21.: Priority inversion occurs when a low-priority job executes while a ready higher-priority job waits.

that priority inversion happens twice. At time $t = 4$ the higher priority job J_2 is blocked, because it executes an lock request $L(S_k)$ that does not succeed because the low priority job J_3 holds the resource S_k. At time $t = 6$, job J_1 is also blocked due to an unsuccessful lock request $L(S_k)$. At time $t = 8$ job J_3 executes the unlock command $U(S_k)$ and gives up the resource S_k, which causes a preemption and finally the higher priority job J_1 can run. Although priority inversion occurs, all jobs meet their deadline.

Timing anomalies can occur due to priority inversion. An example is shown in Figure 3.22, where the execution time of job J_3 is *reduced* to $\delta_{3,k} = 2.5$ instead of $\delta_{3,k} = 4$ in Figure 3.21. Although the total utilisation of the system is reduced, the highest priority job J_1 misses its deadline. This behaviour can be seen as a timing anomaly, because it is counter-intuitive.

Another large problem that can occur without suitable protocols is a phenomenon called *uncontrolled priority inversion*, which is illustrated in Figure 3.23. A job J_1 who wants to execute its critical section $z_{1,k}$ can be blocked by a job J_n that holds the resource

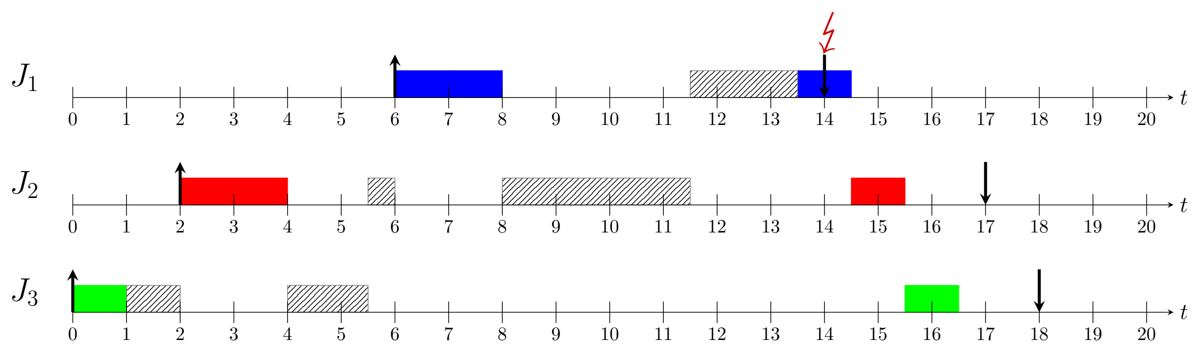


Figure 3.22.: Priority inversion can lead to timing anomalies. Reducing the execution time of job J_3 causes that job J_1 misses its deadline.

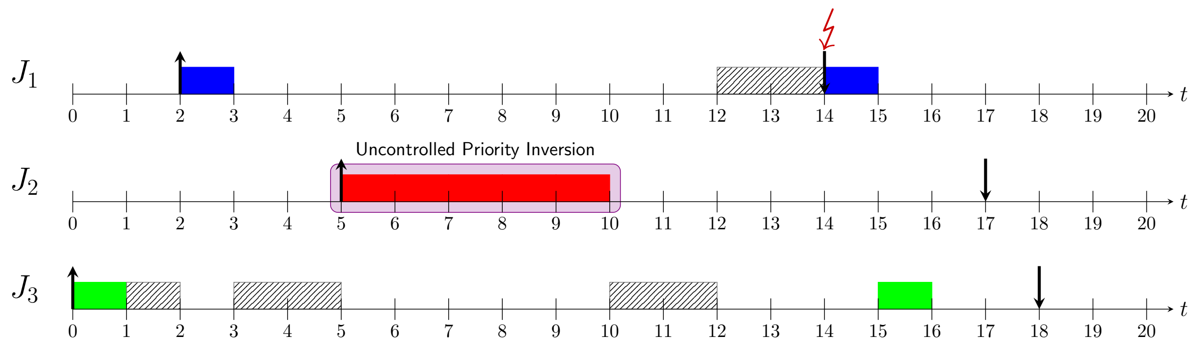


Figure 3.23.: Uncontrolled priority inversion prolongs the blocking duration of a high priority job through a low priority job.

S_k. The duration of of $\delta_{i,k}$ can be prolonged by execution of several jobs J_j with a lower priority than the blocked job and a higher priority than the blocking job. The duration of priority inversion can become unbounded. In Figure 3.23 job J_1 misses its deadline due to uncontrolled priority inversion, because job J_2 prolongs the blocking time of job J_1, although job J_2 does not hold or contend for the exclusive resource S_k.

Resource access protocols are required to deal with priority inversion in a controlled way. Some protocols also help to avoid *deadlock*, which can occur when dealing with exclusive resources. An example id illustrated in Figure 3.24. A disciplined approach is required to avoid deadlocks. If a deadlock occurs the program cannot proceed. It is very difficult debug and test for deadlock, because the deadlock situation is very difficult to create.

Resource requirements can be defined using a resource requirements graph as illustrated in Figure 3.25. This graph expresses that

- τ_1 needs S_1 for at most 2 time units $\Rightarrow \delta_{1,1} = 2$
- τ_2 needs S_1 for at most 4 time units $\Rightarrow \delta_{2,1} = 4$
- τ_3 needs S_1 for at most 4 time units $\Rightarrow \delta_{3,1} = 4$

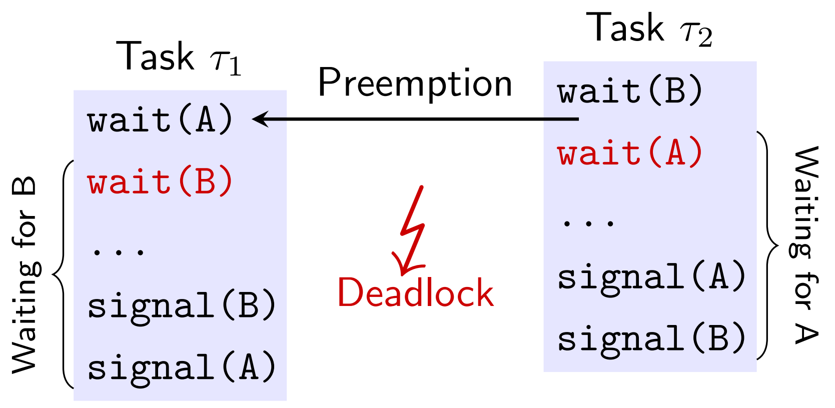


Figure 3.24.: Deadlock can occur, if a cyclic dependence is created. In this situation neither task can continue.

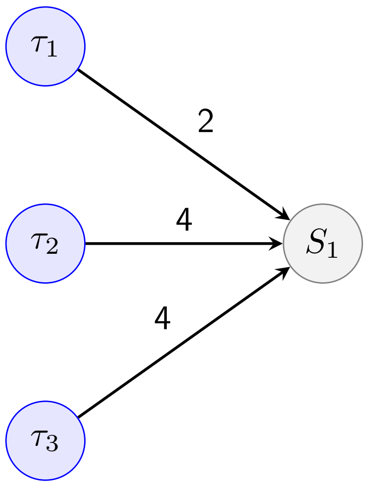


Figure 3.25.: Resource requirements graph

The list defines additional terms and symbols, which will help to study resource access protocols.

- The maximum *blocking time* that task τ_i can experience is denoted by B_i.
- The set of the resources a task τ_i uses is denoted by σ_i.
- The set of resources used by the lower priority task j that can block the task τ_i is denoted by $\sigma_{i,j}$.
- $\gamma_{i,j}$ denotes the set of the longest critical sections of task τ_j that can block task τ_i by accessing a resource S_k.

$$\gamma_{i,j} = \{Z_{j,k} | P_j < P_i \wedge S_k \in \sigma_{i,j}\}$$

- γ_i denotes the set of all longest critical sections that can block task τ_i

$$\gamma_i = \bigcup_{j:P_j<P_i} \gamma i, j$$

3.6.3. Non-Preemptive Protocol

The Non-Preemptive Protocol (NPP)[5] is the simplest resource access protocol. All critical sections are scheduled *non-preemptively*, i.e. the *current priority* $p_i(S_k)$ of a task τ_i that starts executing a critical section $z_{i,k}$ is raised to the highest priority level of all tasks, $p_i(S_k) = \max_h\{P_h\}$. The current priority of the task τ_i is reset to its assigned priority, when the tasks leaves its critical section. Figure 3.26 illustrates the protocol by

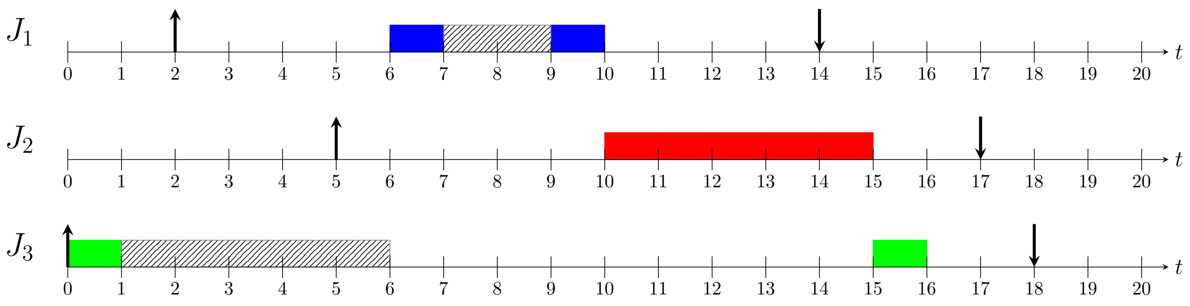


Figure 3.26.: The Non-Preemptive Protocol is the simplest resource access protocol.

means of the example for uncontrolled priority inversion in Figure 3.23. When job J_3 runs its critical section, it runs at the highest priority of all jobs. Uncontrolled priority inversion cannot occur. Job J_1 meets its deadline.

There are many advantages with the Non-Preemptive Protocol (NPP) protocol: a) it is simple to implement, b) uncontrolled priority inversion cannot occur, c) a high-priority job can only be blocked once because of a low-priority job, and d) the blocking time B_i can be calculated by

$$B_i = \max_{j,k}\{\delta_{j,k} | Z_{j,k} \in \gamma_i\}$$

where

$$\gamma_{i,j} = \{Z_{j,k} | P_j < P_i, k = 1, \ldots, m\}$$

NPP is a very good protocol when the critical sections are short.

However, there is a large disadvantage of the NPP protocol, because every job can be blocked by any lower-priority job even if there is no resource conflict between them. This might lead to unacceptably long response times for the tasks with highest priority.

3.6.4. Priority Inheritance Protocol

The idea of the *priority inheritance protocol* is that the blocking jobs J_l *inherits* the priority of the blocked job J_h, when it blocks job J_h. Thus, it avoids that a high priority

[5]Non-preemptive critical section (NPCS) protocol in (Liu, 2000).

job J_h is unnecessarily blocked by a low priority job J_l that does use a different resource than the low priority job J_l.

Liu (2000) gives the following definitions and rules that describe the priority inheritance protocol.

- Definitions
  - The priority of a job J_i according to the scheduling algorithm is its assigned priority.
  - At any time t, each ready job J_i is scheduled and executes at its current priority $p_i(t)$, which may differ from its assigned priority and vary with time.
- Rules
  1. Scheduling Rule
     a) Ready jobs are scheduled on the processor preemptively in a priority-driven manner according to their current priorities.
     b) At its release time t, the current priority $p_i(t)$ of every job J_i is equal to its assigned priority.
     c) The job remains at this priority except under the condition stated in the priority-inheritance rule.
  2. Allocation Rule
     a) When a job J_i requests a resource R at time t,
        - if R is free, R is allocated to J_i until J releases the resource;
        - if R is not free, the request is denied and J_i is blocked.
  3. Priority Inheritance Rule
     a) The current priority $p_l(t)$ of a job J_l may be raised to the higher priority $p_h(t)$ of a job J_h.
     b) Then, the lower-priority job J_l inherits the priority of the higher-priority job J_h, and then J_l executes at its inherited priority $p_h(t)$.

Figure 3.27 illustrates the priority inheritance protocol by means of the example of Figure 3.23. At time $t = 3$ job J_1 wants to access the shared resource and is blocked by job J_3, which inherits the priority of task job J_1 as its current priority $p_3(3) = 1$. When job J_3 finishes its critical section at time $t = 7$, its current priority drops back to $p_3(7) = 3$.

A more complex example uses five jobs and two shared resources. The jobs and their use of the resources are summarised in the following Table 3.3. The resource usage in the table should be read as follows. Job J_4 will execute for 1 time unit without using a shared resource, then it will enter a nested critical section, which first uses resource S_1 for 2 time units, then S_2 for 1.5 time units, and then S_1 for 0.5 time units. Finally, J_4 will execute for 1 time unit without using a shared resource. Figure 3.28 gives the schedule

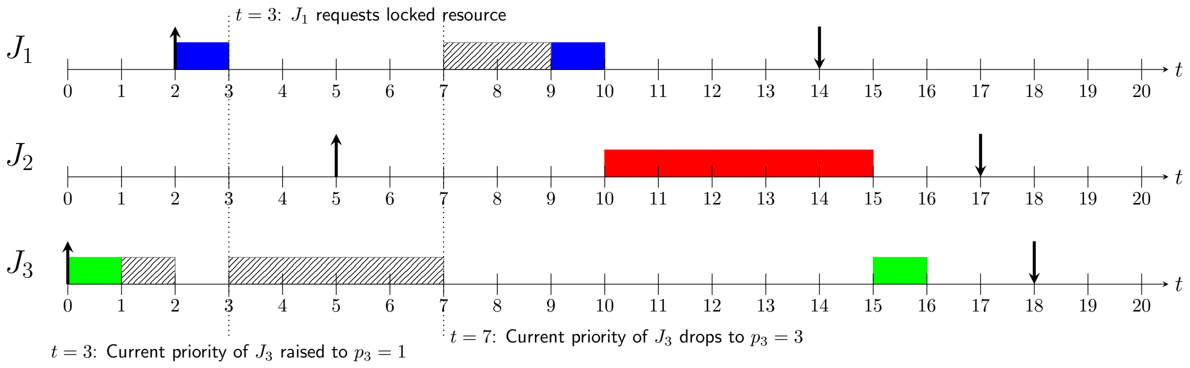


Figure 3.27.: In the priority inheritance protocol Non-Preemptive Protocol the blocking task inherits the priority from the blocked task.

Table 3.3.: Parameters of jobs and their use of shared resources

Job	r_i	C_i	P_i	Critical Sections
J_1	7	3	1	[(-,1);(S_1,1),(-,1)]
J_2	5	3	2	[(-,1);(S_2,1),(-,1)]
J_3	4	2	3	[(-,2)]
J_4	2	6	4	[(-,1);((S_1,2),(S_2,1.5),(S_1,0.5));(-,1)]
J_5	0	6	5	[(-,1);(S_2,4),(-,1)]

for the priority inheritance protocol. Whenever a higher priority job is blocked by a lower priority job, the lower priority job inherits the priority from the higher priority job.

The priority inheritance protocol is still a simple protocol that guarantees that uncontrolled priority inversion cannot occur. However, the protocol does not minimise the blocking time. A high priority job J_h can be blocked by several lower priority jobs, if it runs a nested critical section. This also makes it difficult to calculate blocking times. Furthermore, the protocol does not prevent deadlock. External mechanisms are needed to prevent deadlock.

3.6.5. Priority Ceiling Protocol

The priority ceiling protocol extends the priority inheritance protocol to prevent deadlocks and aims at further reducing the blocking time. The protocol requires that

- the assigned priorities of all jobs are fixed
- the resources required by all jobs are known a priori before the execution of any job begins

Liu (2000) gives the following definitions and rules that describe the priority ceiling protocol.

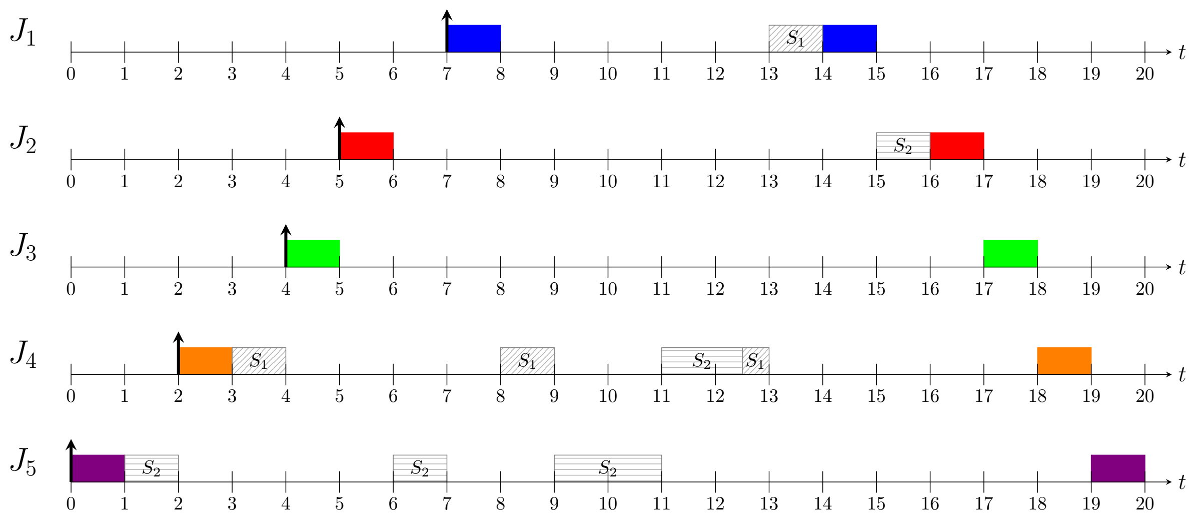


Figure 3.28.: Priority inheritance protocol using the job parameters of Table 3.3

- Definitions
  - The *priority ceiling* of any resource S_k is the highest priority of all jobs that require S_k and is denoted $C(S_k)$.
  - At any time t, the *current priority ceiling* $C(S^*)(t)$ of the system is
    - equal to the highest priority ceiling of the resources that are in use at the time, if some resources are in use
    - otherwise it is Ω, a non-existing priority that is lower than the priority of any job

- Rules
  1. Scheduling Rule
     a) At its release time t, the current priority $p_i(t)$ of every job J_i is equal to its assigned priority.
     b) The job remains at this priority except under the condition stated in the priority-inheritance rule.
     c) Every job J_i is scheduled preemptively and in a priority-driven manner at its current priority $p_i(t)$.
  2. Allocation Rule
     a) When a job J_i requests a resource S_k at time t,
        i. if S_k is not free, the request is denied and J_i is blocked.
        ii. if S_k is free,
           - if J_i's current priority $p_i(t)$ is higher than the current priority ceiling of the system $C(S^*)(t)$, S_k is allocated to J.

– if J_i's current priority $p_i(t)$ is not higher than the current ceiling of the system $C(S^*)(t)$, S_k is allocated to J_i only if J_i is the job holding the resource(s) whose priority ceiling is equal to $C(S^*)(t)$; otherwise J_i's request is denied and J_i becomes blocked.

3. Priority Inheritance Rule

   a) When J_h becomes blocked, the job J_l which blocks J_h inherits the current priority $p(_h(t)$ of J_h.

   b) J_l executes at its inherited priority $p_l(t)$ until the time t' where it releases every resource whose priority ceiling is equal to or higher than $p_l(t)$.

   c) At that time t', the priority of J_l returns to its priority $p_l(t')$ at the time t' when it was granted the resource(s).

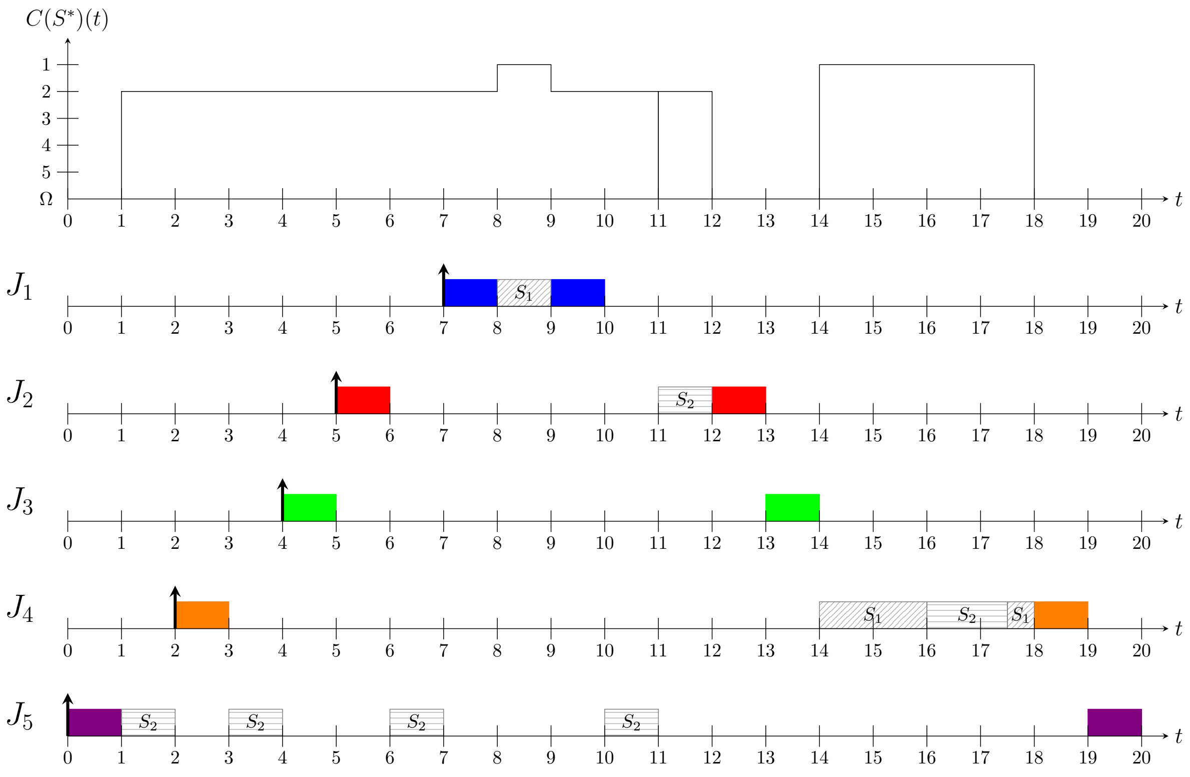


Figure 3.29.: Priority ceiling protocol using the job parameters of Table 3.3

Figure 3.29 illustrates the priority ceiling protocol using the job parameters of Table 3.3. It also shows the current priority ceiling $C(S^*)(t)$ at each time instance. When resource S_1 is in use, $C(S^*)(t) = 1$, because the highest job that can request S_1 is job J_1 with the priority $P_1 = 1$. When resource S_2 is in use, but not resource S_1, then $C(S^*)(t) = 2$, because the highest job that can request S_2 is job J_2 with the priority $P_2 = 2$.

- At time $t = 0$ no resource is in use, thus the current priority ceiling $C(S^*)(0)$ is Ω.

- At time $t = 1$ job J_5 locks successfully the resource S_2, and thus current priority ceiling $C(S^*)(1)$ is raised to 2. Job J_5 still executes at its assigned priority, $p_5(2) = 5$.

- At time $t = 2$ job J_4 starts execution and preempts job J_5.

- At time $t = 3$ the special priority ceiling rule comes in use. Job J_4 wants to access the free resource S_1. However, since the current priority ceiling $C(S^*)(3) = 2$ is higher than the current priority $p_4 = 4$ of job J_4, J_4 is blocked. Since the blocking has been indirectly be caused by job J_5, job J_5 inherits the current priority of J_4 and executes now at the new current priority $p_5(3) = 4$.

The rest of the schedule is left as an exercise for self-studies.

A comparison of Figure 3.28 and Figure 3.29 shows that the higher priority job J_1 finishes earlier in the priority ceiling protocol. This is a consequence of an important property of the priority ceiling protocol, which ensures that a job can only by blocked for the duration of one critical section as stated as Theorem 8.2 in the book of Liu (2000).

Blocking Duration (Theorem 8.2 ((Liu, 2000)))

When resource accesses of preemptive, priority-driven jobs on one processor are controlled by the priority-ceiling protocol, a job can be blocked for at most the duration of one critical section.

Furthermore, the priority ceiling protocol is free from deadlock as stated as Theorem 8.1 in the book of Liu (2000).

Freedom from Deadlock (Theorem 8.1 ((Liu, 2000)))

When resource accesses of a system of preemptive, priority-driven jobs on one processor are controlled by the priority-ceiling protocol, deadlock can never occur.

The priority inheritance protocol is a *greedy* protocol, where the requesting job will always get access to a resource, if this resource is free. In contrast, the priority ceiling protocol is a*non-greedy* protocol. A requesting job may not get access to a resource, though the resource is free (see the additional allocation rule).

There are three ways in which a job J_h can be blocked by a low-priority job J_l

- *Direct Blocking*: A job can be directly blocked by a lower-priority job that owns the requested resource

- *Priority Inheritance Blocking*: A job J_l can be blocked by a lower-priority job J_l that has inherited the priority of a higher-priority job J_h

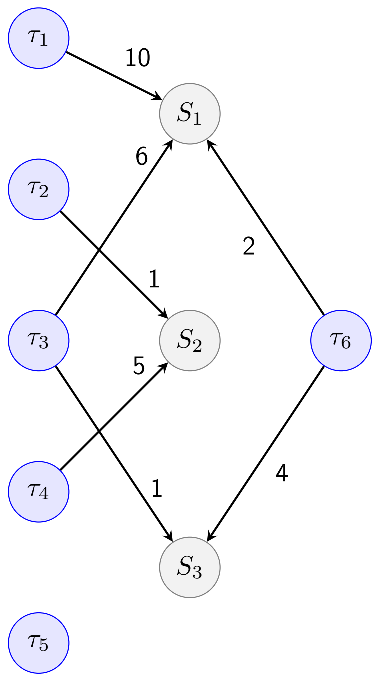


Figure 3.30.: Resource requirements

- *Avoidance (Priority Ceiling) Blocking*: A job J is blocked by a lower-priority job J_l when J requests a resource R_x that is free at that time, and where J_l currently holds another resource R_y whose priority ceiling is equal to or higher than J's current priority $p(t)$.

Figure 3.30 gives the resource requirements for a system of five tasks and three resources, which shall use the priority ceiling protocol. In order to calculate the maximum blocking times for all tasks, it is important that all three ways of blocking are taken into account as shown in Table 3.4. It is important to point out that "when the priorities of

Table 3.4.: Different blockings in the priority ceiling protocol for the resource requirements of Figure 3.30.

	Direct Blocking by					Priority Inheritance Blocking by					Priority Ceiling Blocking by				
	J_2	J_3	J_4	J_5	J_6	J_2	J_3	J_4	J_5	J_6	J_2	J_3	J_4	J_5	J_6
J_1		6			2										
J_2	-		5			-	6			2	-	6			2
J_3		-			4		-	5		2		-	5		2
J_4			-					-		4			-		4
J_5				-					-	4				-	

all the jobs are distinct, the entries in the priority ceiling blocking table are equal to the corresponding entries in the priority inheritance blocking table, except for jobs that do not require any resources (Liu, 2000)".

From the table, the following blocking times can be derived: $B_1 = 6$, $B_2 = 6$, $B_3 = 5$, $B_4 = 4$, and $B_5 = 4$.

These blocking times can then be used in a time demand analysis to analysis a given set of tasks that access a set of resources under the priority ceiling protocol.

4. Hardware/Software Co-Design

4.1. Introduction to Hardware/Software Co-Design

So far, the course has focused only on systems that are implemented in software running on a single processor platform. However, for many systems it is difficult to achieve the necessary performance with this approach. Software implies an overhead, because to execute an operation, first an instruction has to be fetched, decoded and executed from memory, and then also the data has to be transferred between memory and processor. Furthermore, software is dependent on the underlying processor hardware. If an algorithms specifies several independent multiplications, a processor with a single multiplier cannot run these multiplications in parallel, but has to run the multiplications sequentially. A pure hardware solution could instead generate the necessary parallel multipliers and could run then these multiplications in parallel, leading to a far better performance compared to the single multiplier processor.

Thus, it is evident that implementations would profit from the use of hardware, in particular for subsystems that are performance-critical. But how should these systems be designed? Traditionally, the software and hardware design processes are different. Embedded software is programmed in languages like C (Appendix C) or Ada (Appendix D), while hardware is described in a hardware description language like VHDL (Appendix E). Research on on *hardware/software co-design* started in the 1990s (Wolf, 1994) (Chiodo et al., 1994) (Teich, 2012) and aimed at establishing an automated design process for systems consisting of hardware and software, where the following problems were addressed:

Co-specification How to create specification that are suitable to describe both the hardware and the software of a system?

Co-synthesis How to automatically generate the software and hardware required to meet the system specification?

Co-simulation How to simulate a system consisting of software and hardware at different levels of abstraction?

Co-verification How to verify the correctness of a system consisting of software and hardware?

Although the topic of hardware/software co-design has now a history of more than 25 years, the problem is are far from solved. Most larger industrial systems fall into the

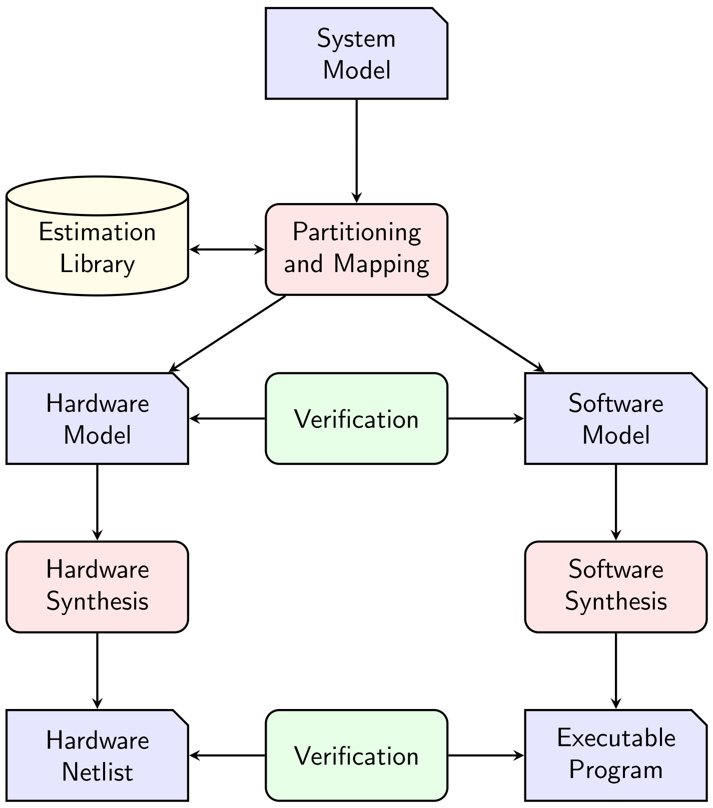


Figure 4.1.: A typical hardware-software co-design flow

area of hardware/software co-design, but the design process for these systems is far from automated.

Figure 4.1 illustrates a typical hardware/software co-design flow. The application is described by a system model, which describes the functionality of the system without specifying which parts shall be implemented in hardware or software (*co-specification*). The system model is often based on a concurrent process model, where processes communicate with each other via signals. During the *co-synthesis* activity, the system is partitioned and a decision is taken, which processes are implemented on the different hardware and software components. This step involves several tasks, which depend on each other.

- *Partitioning:* The functionality of the system is divided into smaller, interacting computation units;

- *Allocation:* The decision, which computational resources are used to implement the functionality of the system;

- *Scheduling:* If several system functions have to share the same resource, the usage of the resource must be scheduled in time; and

- *Mapping:* The selection of a particular allocated computational unit for each computation unit.

There are many possible solutions for the co-synthesis problem due to the large *design space*. The activity to explore the design space is called Design Space Exploration (DSE) and is ideally supported by tools, which can give good performance estimates for a specific candidate solution. The output of the co-synthesis phase are hardware models, e.g. in VHDL (Section E), or software programs, in a programming languages like C (Section C) or Ada (Section D). These models will then be synthesised or compiled to hardware and software using hardware synthesis tools and software compilers. The task of *co-verification* is to validate the design at different levels of abstraction. *Co-simulation*, the simultaneous simulation of the interaction of the hardware and software models is a very challenging task, because hardware and software designers use simulation tools, which do not share the same time model. Simulation in software is usually done with Instruction Set Simulators (ISSs), which only define an order of instructions, but no clear time relation, while hardware simulators use clock cycles as their time base.

In summary, the automation of hardware/software co-design is not only a very challenging task, but also a very important topic. All industrial designs of a certain complexity consist of both hardware and software, and thus are hardware/software co-designs, but a major part of this activity is currently not supported by tools and conducted manually.

Two early important works in hardware/software co-design shall be mentioned here. In these approaches the platform is an microprocessor, an FPGA and a shared memory. The assumption is that hardware gives better performance, but is also more costly.

The COSYMA tool (Ernst et al., 1993) started their co-synthesis approach with the idea to locate all functions in software. In each iteration, the function is moved to hardware that gave the best improvement. This iteration continues until the system satisfies its performance constraints, or until all functions have been moved to hardware.

The VULCAN tool (Gupta and Micheli, 1993) followed the opposite approach and started with a configuration, where all functions where implemented in hardware, and as long as the performance constraints were fulfilled moved the function that gave the largest cost increase to software.

4.2. Hardware Accelerator

Software solutions are very flexible, so they are often the first choice for embedded systems. Nevertheless, if the performance cannot be met by a pure software solution, then the performance of the system can be improved by moving critical parts of the system to hardware, in form of a *hardware accelerator*. As illustrated in Figure 4.2, a hardware accelerator is a device that is connected to the processor system via the interconnection network. Here, the discussion focuses on bus-based systems. In this case, the communication between processor and hardware accelerator is often done via a shared memory.

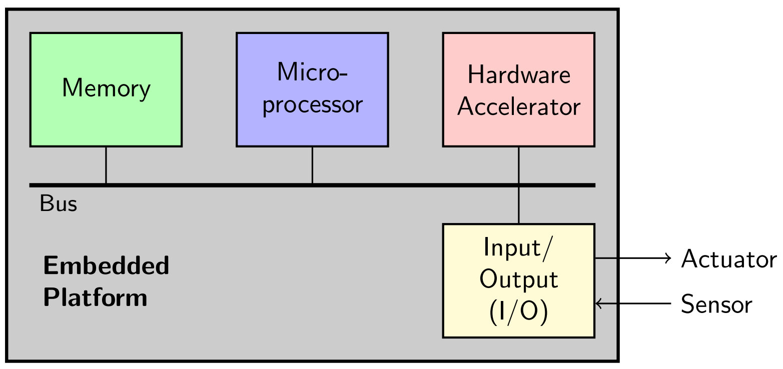


Figure 4.2.: The hardware accelerator is connected via the interconnection system to the processor system. Communication between processor and hardware accelerator is often done via a shared memory.

One alternative to a hardware accelerator is a more powerful processor, but often this is not feasible. First, the cost for a more powerful processor increases in general not linear, but rather exponentially, with the performance. And second, a more powerful processor will also consume more power. Another alternative is to parallelise the system by adding additional processors, but a hardware accelerator has the advantage that it can be selected or customised for a specific functionality.

Amdahl's law can be used to determine, which performance can be achieved, when a part of a system can be accelerated. The speedup S is defined as

$$S = \frac{C_{old}}{C_{new}}\,, \tag{4.1}$$

where C is the execution or computation time. Amdahl's law gives the *theoretical speedup* S of the system as

$$S = \frac{1}{(1-f)+\frac{f}{s}}\,. \tag{4.2}$$

Here, f is the fraction of the part of the system that benefits from an improved execution time, and s is the speedup of the part of the system that has an improved performance. As can be seen from Equation 4.2, there is a limit on the speedup that can be achieved. Assuming an infinite speedup $s = \infty$ of the accelerated part of the system, the maximum theoretical speedup that can be achieved is

$$S_{max} = \frac{1}{1-f}\,. \tag{4.3}$$

From Equation (4.3), it can be seen that a significant speedup can only be achieved, if the share of the part that is improved is large in comparison to the other parts of the system. This is also illustrated in Table 4.1.

Amdahl's law can also be used for parallel architectures, where a sequential program is parallelised and runs on identical parallel units. Here, the theoretical speedup depends both on the fraction f of the program that can be parallelised and the number of identical

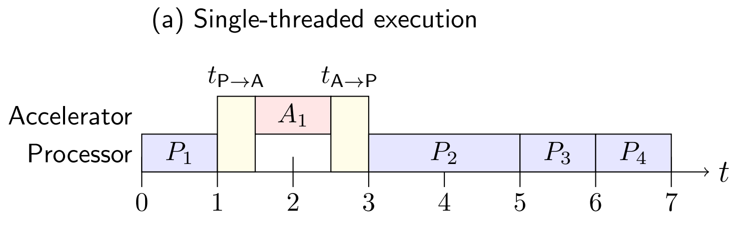
Table 4.1.: The theoretical speedup S is limited by the fraction f that the improved part is used in the original system. The table shows the theoretical speedup S, assuming an infinite speed-up $s = \infty$ of the improved part, as function of the fraction f.

Fraction f	0.1	0.3	0.5	0.9
Theoretical Speedup S	1.11	1.43	2	10

parallel units p that can be used to run the parallelised code. The theoretical speedup S for parallelisation of a sequential program is shown in Equation (4.4)

$$S = \frac{1}{(1-f) + \frac{f}{p}} \,. \tag{4.4}$$

It is important to point out, that Amdahl's law is an abstract law and does not take into account low-level details, like a possible communication bottleneck or overhead. Thus, Amdahl's law should be mainly used to get a good first estimate about the performance of the system that can be achieved using a hardware accelerator or parallelisation techniques, but for a more accurate estimate additional details are required.



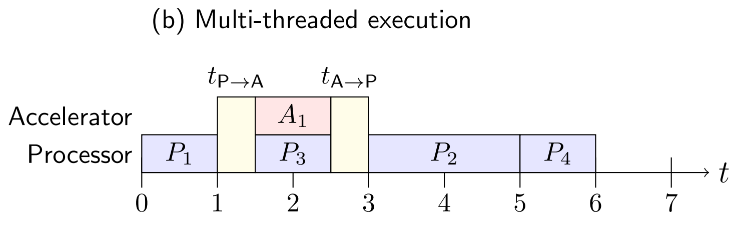


Figure 4.3.: Exploiting potential parallelism is key for system performance

Figure 4.3 shows how an accelerated system can benefit from the exploitation of concurrence, i.e. potential parallelism. In a *single-threaded execution*, the processor will wait for the accelerator, while in a *multi-threaded execution*, the processor can execute in parallel with the accelerator, which reduces the execution time because of the overlapping of activities. The following list shows several possibilities to exploit parallelism.

- Functions can be executed in parallel, if there is no data dependency between these functions.
- I/O-handling can be overlapped with computation, which is the main idea of the DMA in Section 2.5.2.

- Using buffers, the idea of pipelining can be used for parallel execution. The basic prinicple has been introduced in Figure 2.23, as an example how interrupts in combination with buffers can be used to enable a concurrent execution. If the platform supports parallelism then a true parallel execution can be achieved.

The following tutorial example illustrates the importance of the exploitation of parallelism to achieve a good system performance.

The system shall implement the function $system(x, y) = h(f(x), g(y))$. The system shall be implemented on a bus-based shared memory architecture consisting of a processor P, an accelerator A and a memory M. In the beginning of the execution the variables x and y are stored in the memory M. Also, the result shall be stored in the memory at the end of the execution. The computation time of the functions f, g and h depends on the Processing Element (PE) they are executed on. The computation times are given in Table 4.2. The functions f and g can be executed on both processor P and accelerator A, but function h can only be executed on the processor P, but not on the accelerator A.

Table 4.2.: Computation times for functions executed on different processing elements

Function	Processor P	Accelerator A
f	5	2
g	5	2
h	5	–

It is further assumed, that both processor and accelerator have a sufficient number of registers and that processor and accelerator cannot access the bus simultaneously. An access to the memory takes $C_{\mathsf{M}} = 1$ time unit for both processor and accelerator. The access time to a register on processor or accelerator is negligibly small. The task is to find the implementation that gives the shortest computation time, which includes the time for reading the values x and y from memory M and writing the result to memory M.

Three alternatives have to be analysed.

1. The implementation used only the processor, the accelerator is not used.
2. Only function h is implemented on the processor, the functions f and g are implemented on the accelerator.
3. Only one function of the functions f and g is implemented on the accelerator, and function h is implemented on the processor. Since the functions f and g have the same computation times for both processor and accelerator ($C_{\mathsf{P},f} = C_{\mathsf{P},g}$ and $C_{\mathsf{A},f} = C_{\mathsf{A},g}$), it does not matter, which of the functions f and g is put on the accelerator.

In the following the different alternatives are analysed.

Table 4.3.: Schedule and computation time calculation for alternative 1

Instruction	Computation Time Processor P
Load x	1
Load y	1
$f(x)$	5
$g(y)$	5
$h(\dots)$	5
Store $h(\dots))$	1
Sum	**18**

- Alternative 1: The results of f and g are stored in the registers of the processor. Thus, there is no internal extra communication time, when the function h accesses these values. The total computation time C_1 is 18 time units.

- Alternative 2: This solution includes a lot of communication overhead, since results

Table 4.4.: Schedule and computation time calculation for alternative 2

Instruction	Computation Time Processor P	Instruction	Computation Time Accelerator A	Computation Time System
		Load x	1	1
		Load y	1	1
		$f(x)$	2	2
		Store $f(x)$	1	1
Load $f(x)$	1	$g(y)$	2	2
		Store $g(x)$	1	1
Load $g(x)$	1			1
$h(\dots)$	5			5
Store $h(\dots)$	1			1
Sum				**15**

of f and g have to be transferred to the processor, which is done via the shared memory. Thus, the speedup compared to alternative 1 is very limited and suffers from the fact that the execution is still almost *single-threaded*.

- Alternative 3: Alternative 3 gives the clearly the fastest implementation. The reason is that this implementation exploits the potential parallelism of the architecture by overlapping the execution of function g with the loading of x, the computation of $f(x)$ and the storing of the result in the memory.

Table 4.5.: Schedule and computation time calculation for alternative 3

Instruction	Computation Time Processor P	Instruction	Computation Time Accelerator A	Computation Time System
Load y	1			1
$g(y)$	5	Load x $f(x)$ Store $f(x)$	1 2 1	5
Load $f(x)$	1			1
$h(\dots)$	5			5
Store $h(\dots)$	1			1
Sum				**13**

4.3. Implementation Alternatives

Although the microprocessor is often the first choice to implement an embedded systems, the embedded system designer should be aware of the different technologies that can be used for an embedded system. Figure 4.4 illustrates the implementation alternatives, which imply a trade-off between design time and design costs against performance. Software solutions require in general a shorter design time and lower costs, but cannot deliver the performance of pure hardware solutions. ASIC and full-custom designs need to be fabricated in a semiconductor fabrication plant. Compromises are specialised processors, like DSPs, or Complex Programmable Logic Devices (CPLDs) and FPGAs, which can be programmed in-house without a semiconductor fabrication plant. ASICs and full-custom designs require high-volume production because of the very high initial engineering costs, but for high or very high volumes the cost per fabricated chip will become lower than for an FPGA. Obviously, the different technologies can be combined in an embedded system design as illustrated in Section 4.1 and Section 4.2.

In general, the performance of hardware designs can be considerably better than software solutions due to the following aspects:

- Execution in software requires a significant overhead in form of the need to
  - load instructions from memory and to decode them before they can be executed, and
  - load and store operands and results of an operation in memory;
- Hardware can be optimised for the application in different ways:
  - the size of data types can be fully customised, while software in general only supports 8-, 16-, 32-, and 64-bit data types, and
  - operations can be optimised for speed or area in hardware, as can be exemplified by different adders that can be chosen in a hardware design, e.g. ripple-carry adder (low area, low speed), carry-lookahead adder (large area, high speed), carry-select adder (compromise between these extremes);

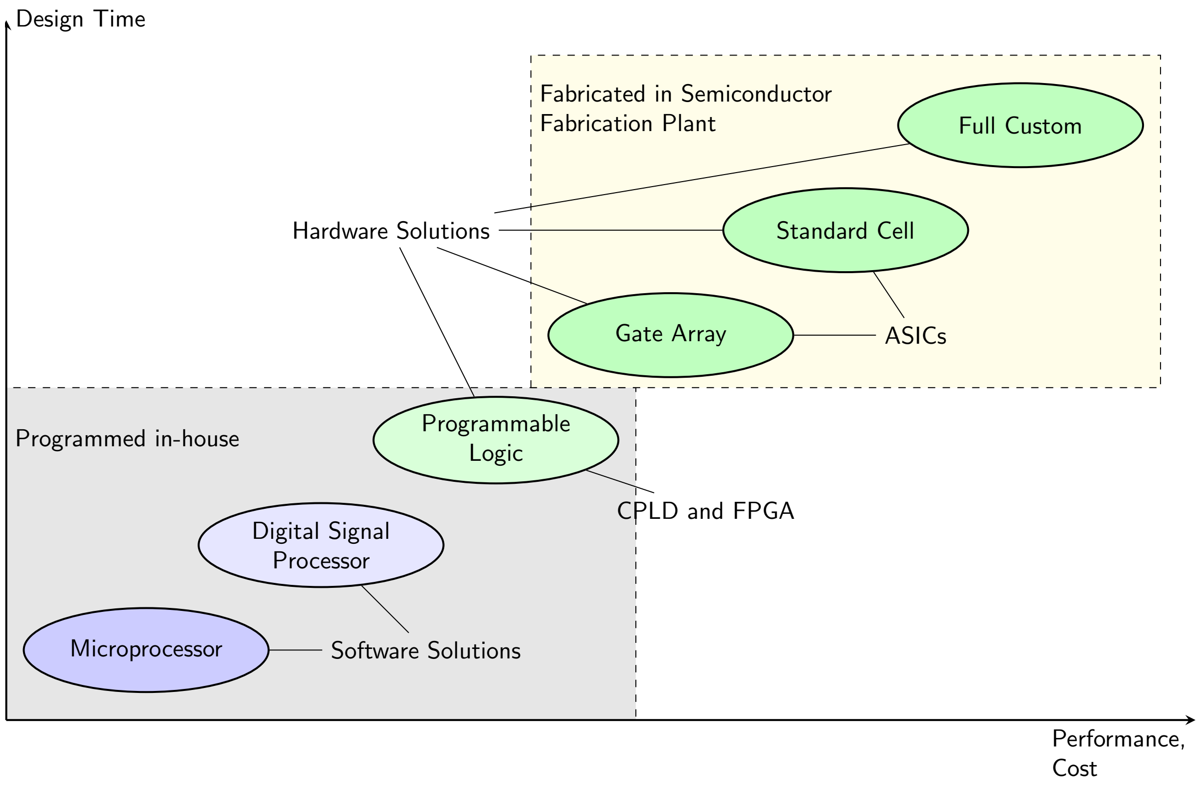


Figure 4.4.: Implementation alternatives for embedded systems

- Potential parallelism can be fully exploited in hardware:
  - if several functional units, like adders, are needed, any number of them can be implemented in hardware, as long as they fit in the hardware device, while
  - microprocessors come with a limited number of hardware resources, which cannot be changed.

Since the course concentrates mainly on embedded software solutions, with possible hardware acceleration as discussed in Section 4.2, the next section will discuss a few principle aspects of DSPs, since these devices are very suitable for many digital processing applications for images, video or audio. For students interested in hardware design, there is an introduction on VHDL (Section E, but for hardware devices or advanced hardware design other textbooks or courses specialised on these topics should be consulted.

4.4. Digital Signal Processor (DSP)

DSPs are specialised processors that have been developed for digital signal processing. These applications are computation intensive applications, which process regular streams of data, and make intensive use of multiply and accumulate operations. The DSP architecture aims at providing a good trade-off for the area of digital signal processing between a general purpose microprocessor and a specialised hardware solution. Thus, it aims at providing the flexibility and short design time for software while still providing a high performance due to a customised architecture suitable for digital signal processing. Note, that a DSP is a specialised processor, which shall not be used for general purpose computing.

There is a huge variety of DSP architectures. One of the most important decisions is to select, if a fixed- or floating-point DSP is more suitable for the application. These number formats will be discussed in Section 4.4.1. An overview of the digital signal processor architecture will be given in Section 4.4.2.

4.4.1. Fixed-Point and Floating-Point Number Representations

One of the most important design decisions is to select the type of DSP, either fixed-point or floating-point. In order to be able to make a good decision the underlying number representation and its implications have to be understood.

Fixed-point representation

The fixed-point number representation is based on the integer representation with binary numbers.

A signed n-bit integer number $B = b_{n-1}b_{n-2}\dots b_1b_0$ using two's complement representation is expressed as

	Sign Bit				
	b_{n-1}	b_{n-2}	$\dots$	b_1	b_0
Weight	$-(2^{n-1})$	2^{n-2}	$\dots$	2^1	2^0

.

The decimal value can be calculated as

$$D_{10} = -b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \cdots + b_1 2^1 + b_0 2^0\,, \tag{4.5}$$

which enables to represent the decimal integer numbers from $-(2^{n-1})$ to $2^{n-1}-1$. Thus a signed 8-bit integer variable can represent the integer numbers from -128 to +127.

There are important properties that follow the two's complement representation:

- *Negation*: The negated number can be derived by taking the Boolean complement of each bit and then adding 1.
- *Extension of bit length*: If a binary number with n bits is extended to $m+n$ bits, then m additional bits with the value of the sign bit are added to the left.
- *Overflow:* An overflow occurs, if two number with the same sign bit are added and the result has the opposite sign bit.
- *Subtraction:* To subtract B from A, the two's complement is taken from B and added to A.

The fixed-point format is based on the representation of integer numbers using two's-complement. In the following, the Q_f format will be discussed, where it is assumed that the a fixed-point number is represented with a sign bit and f bits for the fraction[1].

A signed n-bit fixed-point number $F_{fixed} = B = b_{n-1}b_{n-2}\ldots b_1 b_0$ in the Q_{n-1} is expressed as

	Sign Bit				
	b_{n-1}	b_{n-2}	$\ldots$	b_1	b_0
Weight	$-(2^0)$	2^{-1}	$\ldots$	$2^{-(n-2)}$	$2^{-(n-1)}$

.

The decimal value can be calculated as

$$D_{10} = -b_{n-1} + b_{n-2}2^{-1} + \cdots + b_1 2^{-(n-2)} + b_0 2^{-(n-1)}\,, \tag{4.6}$$

which enables to represent the decimal integer numbers from -1 to $1 - 2^{-(n-1)}$. Thus a signed 8-bit Q_7 fixed-point variable can represent the decimal numbers from -1 to 0.9921875. There is an equal spacing between adjacent numbers, which is 2^{-f} for any Q_f-format. In the Q_7 format, it is $2^{-7} = 0.0078125$.

Addition is performed in the same way as for integer numbers, but multiplication does not cause overflow in the Q_f-format, because all numbers have an absolute value, which is less than or equal to 1. Instead extended sign bits can be produced as illustrated in the following example.

The multiplication of the Q_2-numbers $A_{Q2} = 011$ and $A_{Q2} = 110$ has to be conducted with care and shows some interesting aspects of the fixed-point number system. A_{Q2} represents the decimal number 0.75 and B_{Q2} represents the decimal number -0.5.

Since two Q_2 numbers are multiplied the result is a Q_4 number. Special care has to be taken to correctly sign extend the numbers, which is indicated by the bits originating from sign extension marked in blue colour.

$$\begin{array}{ccccccccc}
 & & 1 & . & 1 & 0 & \times & 0 & . \; 1 \; 1 \\
\hline
 & 1 & . & 1 & 1 & 1 & 0 & & \\
 & 1 & . & 1 & 1 & 0 & & & \\
\hline
1 & 1 & . & 1 & 0 & 1 & 0 & &
\end{array}$$

However, the calculation yields a result, which uses six bits instead of five bits used in the Q_4-format. The bit to the right, marked in red colour, is a so called *extended sign bit*, which is discarded. Thus, the result of the calculation is $F_{Q4} = A_{Q2} \cdot B_{Q2} = 11010$, which corresponds to the decimal number -0.3671875. This number cannot be represented in the Q_2-format. Thus, multiplications in fixed-point can lead to a loss of precision, since the result requires often larger registers than available in the DSP.

Floating-point representation

A floating-point number is represented with one sign bit, n_e exponent bits and n_m mantissa (or fraction) bits.

Sign Bit S	Exponent (e)	Mantissa (m)
b_{n-1}	b_{n-2} $\ldots$ b_{n-1-n_e}	b_{n_m-1} $\ldots$ b_0

[1]There is also the more general $Qm.f$-format, where m denotes the number of integer bits.

The term $1.m$ is called the *significand*, with a mantissa $m = m_{n_m-1}m_{n_m-2}\dots m_0$ of m_{n_m} bits. Each mantissa bit m_i corresponds to the value 2^{i-n_m}. The value of the exponent corresponds to its unsigned integer value. The general formula to calculate the decimal value is

$$D_{10} = (-1)^S \cdot (1.m) \cdot 2^{e-bias}, \tag{4.7}$$

where the *bias* is an integer number.

Assuming a floating-point representation with $m = 1$, $e = 2$, and a bias $bias = 1$, the floating-point number $A_{float} = 1101$ represents the decimal number $D_{10} = -1 \cdot 1.5 \cdot 2^{2-1} = -1.5$. Observe, that this representation cannot represent the decimal number 0!

The IEEE-754 floating-point standard defines a 32-bit floating point number with an exponent of eight bits, a mantissa of twenty-three bits, and a bias of 127. A special bit pattern is used to represent a negative and positive zero, but also special bit patterns among others are used for numbers between 0 and 2^{-126} and positive and negative infinity. The IEEE-754 standard covers half- (16 bits), single- (32 bits), double- (64 bits), x86 extended- (80 bits) and quad- (128 bits) precision.

Arithmetic operations are more complex than the corresponding fixed-point operations, since for instance an addition requires to first align the numbers to the same exponents, before the significands can be added.

An *addition* algorithm covers the following four phases to yield the result of an addition.

1. *Zero Check*: The result can be directly reported, if one operand is zero;
2. *Significand Alignment*: The operands have to be aligned so that the exponents are equal;
3. *Addition*: The significands are added taking the sign bit into account;
4. *Normalise*: The resulting significand need to be shifted so that the leftmost bit is one, then the exponents need to be updated. If the result does not fit into the register, the significand needs to be rounded.

A *multiplication* algorithm covers the following four phases to yield the result of a multiplication.

1. *Zero Check*: Result is zero, if one operand is zero;
2. *Add exponents*: The exponents need to be added according to $e_{\text{Result}} = e_A + e_B - bias$;
3. *Multiplication*: Significands are multiplied taking the sign bits into account;
4. *Normalise and round*: The resulting significand needs to be shifted, so that the leftmost bit is one, then the exponents need to be updated. If the result does not fit into the register, the significand needs to be rounded.

Comparison: Fixed-Point and Floating-Point

To compare number systems, different properties can be used. The *maximum quantisation error* expresses the error that can exist in a number system due to the fact that not all numbers can be represented in a number system. The *dynamic range* expresses the ratio between the largest and smallest number, which can be expressed in a number system.

The following example shall illustrate the difference between fixed-point numbers and floating-point numbers. The example compares four-bit fixed-point numbers of Q_3-format with four-bit floating-point numbers, where $e = 2$, $m = 1$ and $bias = 1$.

Table 4.6.: Comparison of fixed-point Q_3 and floating-point $(e = 2, m = 1, bias = 1)$

Bit Pattern	Decimal Value Fixed-Point	Decimal Value Floating-Point
0000	0	0.5
0001	0.125	0.75
0010	0.25	1
0011	0.375	1.5
0100	0.5	2
0101	0.625	3
0110	0.75	4
0111	0.875	6
1000	-1	-0.5
1001	-0.875	-0.75
1010	-0.75	-1
1011	-0.625	-1.5
1100	-0.5	-2
1101	-0.375	-3
1110	-0.25	-4
1111	-0.125	-6

As can be seen from Table 4.6, each of the format represents 16 different numbers. The fixed-point format can represent the numbers from -1 to +0.875. The quantisation error that can occur between two adjacent representable numbers is the same over the whole range of representable numbers, in this case 0.0625. By choosing a large exponent, the floating point-format can have a much larger dynamic range and can represent numbers from -6 to +6. The number zero cannot be presented and the precision close to 0 is not very good in the floating-point example. The quantisation error increases with larger numbers. The maximum quantisation error is 1, and can be caused if the number 5 or -5 shall be represented in this number format.

Fixed-point operations work in the same way as integer operations and require a much simpler algorithm compared to floating-point operations. Thus, a fixed-point requires also considerably less hardware resources and can be faster executed than the corresponding floating-point architecture. Due to the low dynamic range, scaling is often required for fixed-point, which also can lead to a loss of precision.

4.4.2. Digital Signal Processor Architecture

The overview of DSP architectures in this section follows to a large extent the book on DSPs of (Lapsley et al., 1997), which although not very recent, gives an excellent overview of the main ideas of DSP architectures. The DSP architecture is optimised for extremely high performance for a specific kind of arithmetic-intensive algorithms, which are used in digital signal processing.

- The *data path* is optimised, so that operations like multiply-accumulate shall only take one clock-cycle, and
- the *memory architecture* is optimised, because large amounts of data have to be moved from and to the memory.

The DSP architecture has not been designed for general purpose computing, and shall only be used for digital signal processing.

A Finite Impulse Response (FIR) filter, illustrated in Figure 4.5, is a typical digital

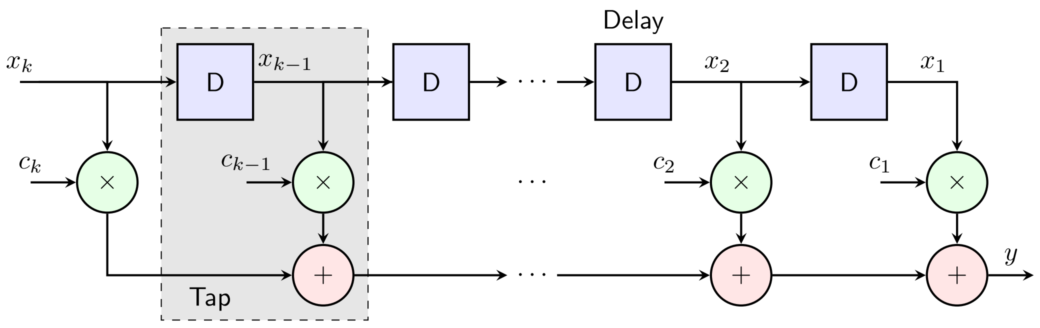


Figure 4.5.: A FIR filter is a typical digital signal processing application

signal processing application and shall be used to explain the architecture of the DSP. The FIR filter executes the function

$$y = \sum_{i=1}^{k} c_k x_k \ . \tag{4.8}$$

In each stage i, also called “tap”, of the FIR filter, a multiply-accumulate operation is executed, using the result from the previous stage, the delayed data value x_i and the coefficient c_i.

An *optimised data path* for an FIR filter would require to

- execute the multiply-accumulate instruction in a one cycle,
- and to store the intermediate result of this operation with high precision in a register, so that an additional access to the memory is not required.

Furthermore, there are several memory accesses, which are required at each stage i:

1. fetch the multiply-accumulate instruction that executes $y_i = y_{i+1} + c_i x_i$,
2. read the delayed data value x_i,
3. read the coefficient value c_i, and
4. store the data value x_{i+1} into the next location in the delay line x_i to shift data through the delay line.

Fixed-point DSP architecture

Figure 4.6 shows the data path of a typical fixed-point DSP architecture. The main parts of this architecture are operand registers, which can hold two operands, a multiplier, an Arithmetic Logic Unit (ALU), accumulator registers, a shifter and a combined shifter/limiter. The data path is not used for address calculation, instead there is a special address calculation unit (not discussed here).

Ideally, the multiplier and accumulator are integrated into a single unit as in Figure 4.6, which enables to conduct a multiply-accumulate operation in a single cycle using two native-sized operands for the multiplication, where the result is fit into the ALU to be added to an intermediate result stored in one of the accumulator registers. The result of the multiplication requires at least double the size of the operand registers, but additional guard bits are used in the accumulators to avoid unnecessary overflows, which have to be prevented by scaling, and a higher precision. In principal only one accumulator register is required, but the use of two accumulator registers allows to for instance use one accumulator as source and the other one as destination register. The ALU implements the basic arithmetic and logic operations. The final result needs to be stored in native size, and thus the programmer has to decide, which part of the result should be kept. Since a fixed-point DSP is also capable of execute operations on integers, it depends on the type of operation, which part of the result is kept.

- In an *integer operation*, the sign bit and the lowest bits are kept as result and the most significant bits are discarded. The designer has to take care that the result fits into the size given by the native operand size to be able to store the correct result.
- In a *fixed-point operation*, the extended sign-bit and the least significant values are discarded, while the sign-bit and the most significant bits are kept. If the result requires more bits than given by the native operand size, then result will be stored with a loss of precision leading to a quantisation error.

The role of the *shifter* is to scale the results by a factor of 2^m using bit shifting. It is required when the result due to multiple and addition operations either need to be either up-scaled or down-scaled to avoid overflow or loss of precision. Another way to deal with overflow is to use *saturation arithmetic*, which means that in case of overflow

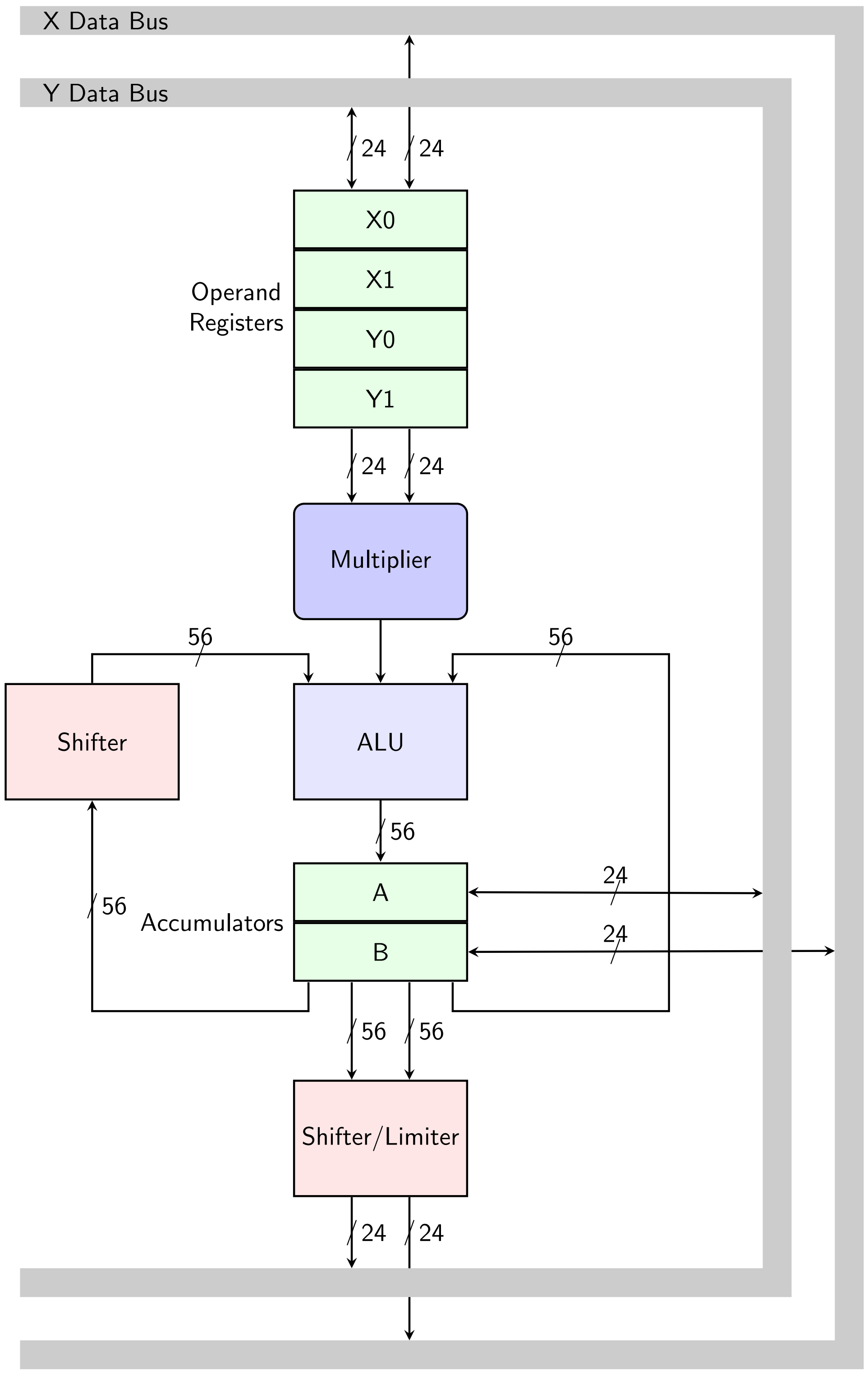


Figure 4.6.: Representative fixed-point DSP data path (Motorola DSP5600x, 24-bit fixed-point processor) - Figure redrawn from (Lapsley et al., 1997)

the result will be converted to the largest representable positive or negative number will be. The component implementing this functionality is the *limiter*, which for instance is useful, when an intermediate result needs to be converted to a native size result as at the output of the data path in Figure 4.6.

Floating point DSP architecture

Floating-point data paths are similar to fixed-point data path, but differ in some respects. Processors with one floating-point unit are usually capable of both floating-point and fixed-point operations, but can only execute one of these operation per instruction cycle. Floating-point multipliers have usually two 32-bit floating-point operands. The output format of the multiplication does in general not provide enough bits to avoid loss of precision, but provides only a limited number of additional bits. ALUs perform floating-point arithmetic instructions. Overflow is much less of a concern in floating-point data paths, due to the large dynamic range. However, loss of precision is much more likely to occur in floating-point operations.

Compared to fixed-point processors, floating-point processors provide a larger dynamic range, which minimises the need for scaling. This means also that it is much easier to use standard C-compilers for floating-point DSPs. However, this comes as a price. For a fixed-point DSP much simpler hardware is required, which means that these processors can be produced at a lower cost, are faster and consume less power. Furthermore, due to the underlying fixed-point representation, they provide the same precision over the full range of representable numbers and can provide a more accurate precision as long as the designer uses proper scaling.

DSP Memory Architecture

In the context of the FIR filter illustrated in Figure 4.6 it was stated in Section 4.4 that four memory access are needed to compute the result of one filter stage. Different architectures have been discussed in Chapter 2.1. The von Neumann architecture (Figure 2.2), which uses a single memory for instructions and data, requires four memory accesses for the FIR filter, because only one instruction or data item can be transferred in each access cycle. In contrast, the Harvard (Figure 2.3) and modified Harvard architecture have two address and data busses and enable a simultaneous instruction and data access (Harvard) or even two simultaneous data accesses. Thus, the memory access for the FIR filter can finish the required memory accesses within three memory access cycles (Harvard) or only two memory accesses (modified Harvard), respectively. Thus, DSP architectures aim at minimising the number of memory accesses by using suitable architectures like the modified Harvard architecture.

4.5. Program Analysis

In order to enable an efficient implementation and to provide good estimates for hardware/software co-design, *program analysis* techniques are of key importance. This section aims to give an overview on program analysis and the role of the compiler in the embedded design process. The major part this section is based on Chapter 5 in (Wolf, 2001).

Program analysis techniques can be used to determine a good estimate for different extra-functional properties of a program designed for an embedded system, such as (a) the average and worst case execution time, (b) the required number of registers and amount of memory, and (c) energy and power that will be consumed during execution. The results of the analysis can be used at different stages in the design process, for instance during co-synthesis to determine a good mapping of functions to platform components. The accuracy and complexity of program analysis techniques depends to a large extent on the complexity and predictability of the target processor platform.

4.5.1. Data flow and control flow graph

Program analysis requires a format that supports analysis. Program source code is difficult to analyse directly. It is very difficult to identify data dependencies or potential parallelism from source code in a sequential programming language, such as in the source code of Listing 4.1.

Listing 4.1. It is very difficult to detect data dependencies or potential parallelism from the source code

```
r = a + b;
u = e + f;
v = 2 * d;
s = a * c;
t = g + s;
w = r + s;
```

If the source code is instead converted into a Data Flow Graph (DFG), then both data dependencies and potential parallelism can be easily detected, as can be seen in Figure 4.7.

Compilers and synthesis tools do not operate directly on the source code, because source code is difficult to analyse, and the same intermediate language representation can be used for languages of the same paradigm. The imperative languages C, Fortran, Pascal, Ada, Java share the same underlying mechanisms and can be supported by the same intermediate representation. An intermediate representation with a small set of set of constructs and a well-defined meaning is a prerequisite for formal analysis and implies a high potential for optimisation.

The source code can be described by an intermediate format consisting of *data flow nodes* (Figure 4.8) to express computation and *decision* or *control nodes* (Figure 4.9) to express express control statements (`if, case, while, for`). The result is a Control and Data Flow Graph (CDFG).

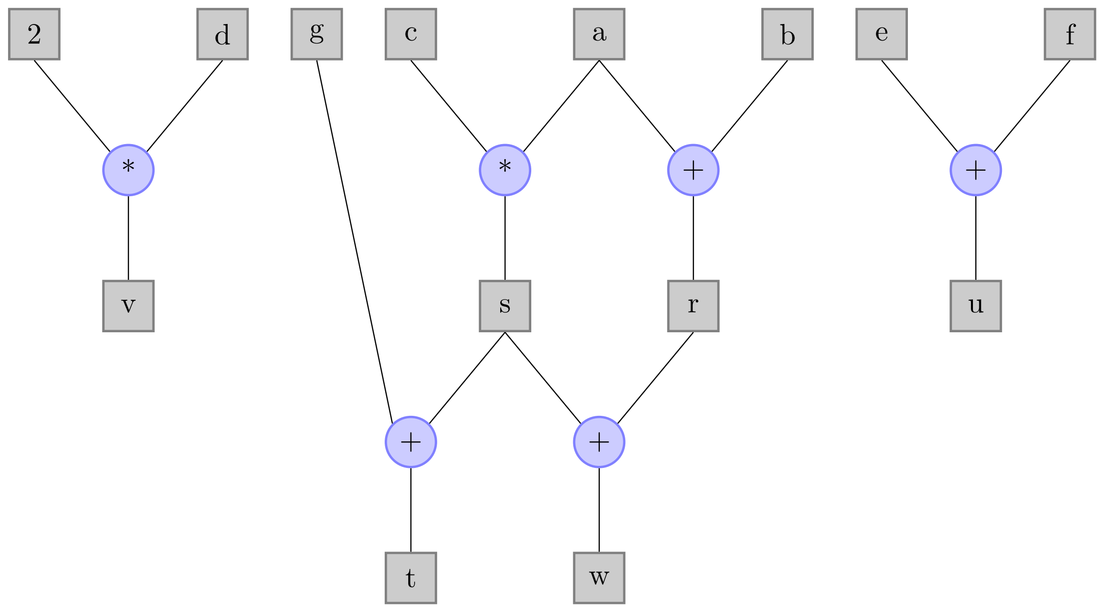
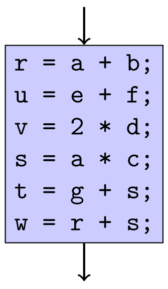


Figure 4.7.: Data dependencies and potential parallelism can be easily detected in the data flow graph corresponding to the source code of Listing 4.1.



```
r = a + b;
u = e + f;
v = 2 * d;
s = a * c;
t = g + s;
w = r + s;
```

Figure 4.8.: A data flow node is a code block without any control structures.

The following **for**-loop

```
for(i=0;i<=3;i++)
   function();
```

can be converted into a **while**-loop

```
i = 0;
while (i<=3) {
   function();
   i++;
}
```

which can then be expressed as a CDFG as illustrated in Figure 4.10.

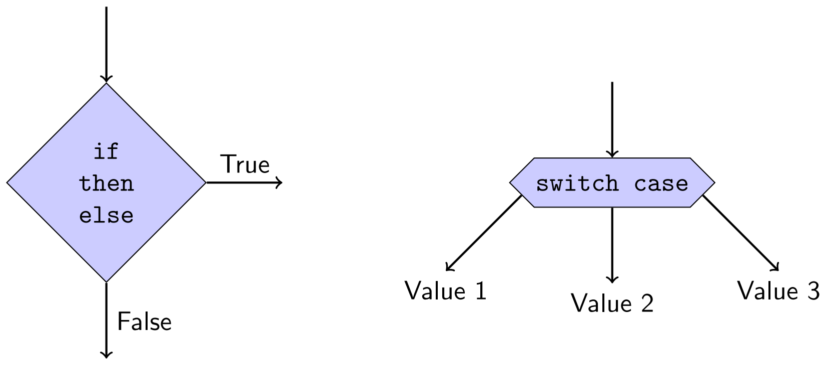


Figure 4.9.: A decision or control node expresses control statements.

4.5.2. Compiler optimisations

The control and data flow graph can be analysed in order to optimise the program towards different parameters. Compilers in general support optimisation with respect to execution time and program size. However, for battery-driven embedded systems, also optimisation for power efficiency would be important. Different parameters conflict with each other, i.e. some optimisations will improve one parameter, but will worsen the other parameter. Loops are of particular importance for optimisations, because the loop body will be executed many times. In the following a few optimisation techniques are presented.

Inlining

Inlining or inline expansion eliminates overhead of procedure and function calls. The function call

```
int function(int x,int y,int z) {
   return x + y + z;
}

z = function(a,b,c);
```

can be expanded by replacing the function call with the corresponding function body.

```
 z = a + b + c;
```

Inlining reduces execution time because the loop overhead is eliminated, but increases code size, if a function is used several times in a program.

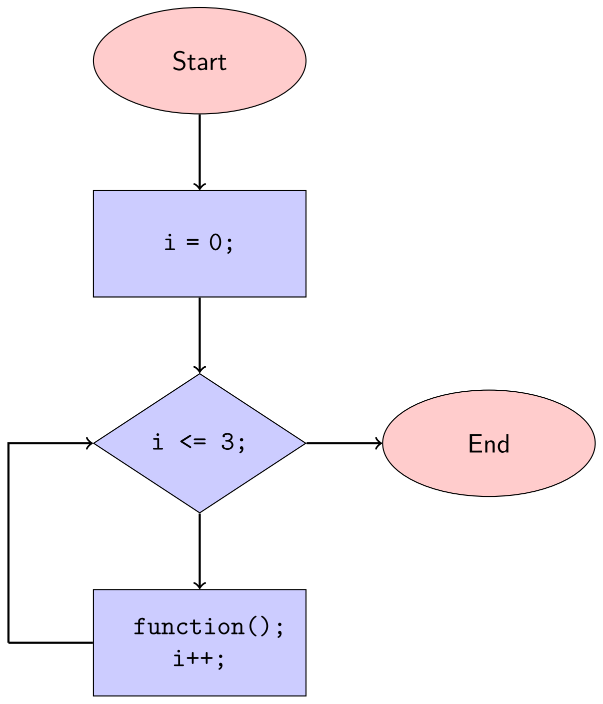


Figure 4.10.: A `for`-loop represented as a CDFG.

Loop unrolling

Loop unrolling eliminates the loop overhead by reducing the loop overhead through expanding the loop body. The `for`-loop

```
for(i=0; i<5; i++) {
  x[i] = a[i] + 3;
}
```

can be unrolled to eliminate the loop body.

```
x[0] = a[0] + 3;
x[1] = a[1] + 3;
x[2] = a[2] + 3;
x[3] = a[3] + 3;
x[4] = a[4] + 3;
```

Loop unrolling improves the execution time, but leads to larger programs.

Loop Fusion

Loop fusion combines two loops with the same loop into a single loop. The two `for`-loops

```c
for(i=0; i<5; i++) {
  x[i] = a[i] + 3;
}
for(j=0; j<5;j++) {
  b[j] = c[j] + d[j];
}
```

can be integrated into a single loop, since both loops have the loop indices.

In general, loop fusion reduces both code size and execution time, because the loop overhead is reduced. However, there are exceptions because loop fusion might lead to worse performance for architectures with a cache, if both loops do not fit in the working set or other cache conflicts.

Technology dependent optimisations

Some optimisations depend on the underlying hardware architecture. The code

```c
int x;
x = a * 8;
```

can be converted to

```c
x = a << 3;
```

However, the resulting code will only be faster, if a shift-operation is faster than a multiplication. Another example is a multiply-accumulate operation, which is supported by DSP processors and can be executed on such architectures in a single cycle.

Register allocation

Registers are a very important resource in the processor, which only has a limited number of registers. If all registers are in use, variables have to be stored in a memory, which requires load and store operations to access it. This is called *spilling* and leads to a decrease in performance. An efficient program uses registers efficiently and avoids spilling!

Static analysis can improve the use of registers. The idea is to analyse the lifetime of each variable and data dependencies between the variables. If there is no data dependency, then instructions can be reordered to use registers more efficiently.

The principle technique for an efficient register allocation shall be illustrated with the following example code fragment.

```
u = c + d;
v = a - b;
w = a - u;
x = v + e;
```

It is assumed that processor only provides the following type of operations *op* R_1, R_2, R_3, where $R_1 \neq R_2$ and $R_1 \neq R_3$ and $R_2 \neq R_3$.

The *register life-time graph* visualises, which registers are needed at a certain instruction cycle, and can be used to determine the number of registers, which are used for a certain part of the code. Assuming that each variable shall be stored in a register, the register life-time graph for the code block above is

Instruction		(1)	(2)	(3)	(4)
	a		⊢	⊣	
	b		⊢⊣		
	c	⊢⊣			
	d	⊢⊣			
Variables	e				⊢⊣
	u	⊢	−−	⊣	
	v		⊢	−−	⊣
	w			⊢⊣	
	x				⊢⊣
Number of Registers		3	**4**	**4**	3

and from the graph it can be concluded that four registers will be needed.

To analyse, if the required number of registers can be reduced, the corresponding DFG of the code fragment, which is illustrated in Figure 4.11, is analysed. The data dependencies in this graph allow a reordering, so that the instruction `w = a - u;` is executed before the instruction `v = a - b;`.

```
u = c + d; /*1*/
w = a - u; /*3*/
v = a - b; /*2*/
x = v + e; /*4*/
```

Then the corresponding register life-time graph is analysed.

```
u = c + d; /*1*/
v = a - b; /*2*/
w = a - u; /*3*/
x = v + e; /*4*/
```

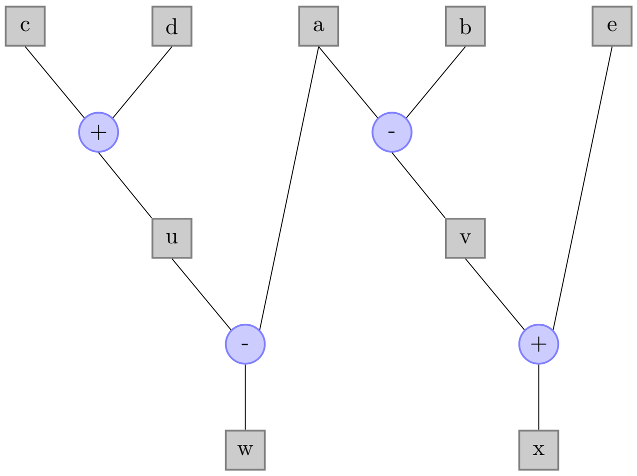


Figure 4.11.: DFG for register allocation

Instruction		(1)	(2)	(3)	(4)
	a		⊢	⊣	
	b			⊢⊣	
	c	⊢⊣			
	d	⊢⊣			
Variables	e				⊢⊣
	u	⊢	⊣		
	v			⊢	⊣
	w		⊢⊣		
	x				⊢⊣
Number of Registers		**3**	**3**	**3**	**3**

This result of this analysis that only three registers are needed at each instruction cycle. Thus, the data dependency analysis and the following reordering of instructions save one register.

Cache analysis

Loops need to be carefully analysed in particular for efficient cache memory access. The execution time of the program

```
for(i=0; i<1000; i++) {
  x[i] = a[i] + b[i];
}
```

depends to a large extent how memory locations are mapped to locations in the cache. If `x[i]`, `b[i]`, `c[i]` are mapped to the same cache line in a direct-mapped cache, there will be conflict misses and low performance.

The GNU compiler GCC

It is important for an embedded system designer to understand the optimisations that can be conducted by the compiler. The GNU-compiler GCC will be used as example, because many tool chains for embedded systems are based on GCC, which provides extensive documentation (GCC-GNU-Compiler).

The GCC compiler has many different optimisations, which can be individually selected with optimisation flags. The flags `-O`, `-O1`, `-O2`, `-O3`, and `-Os` can be viewed as optimisation levels and integrate many individual optimisations. These optimisation levels are summarised here. The explanations follow the GCC-documentation (GCC-GNU-Compiler).

- `-O` or `-O1` (Optimise). Optimising compilation takes somewhat more time, and a lot more memory for a large function. With `-O`, the compiler tries to reduce code size and execution time, without performing any optimisations that take a great deal of compilation time.
- `-O2` (Optimise even more). GCC performs nearly all supported optimisations that do not involve a space-speed trade-off. As compared to `-O`, this option increases both compilation time and the performance of the generated code. `-O2` turns on all optimisation flags specified by `-O`. This includes even inlining, but only for simple functions.
  - `-finline-functions`. Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way.
- `-O3` (Optimise yet more). `-O3` turns on all optimisations specified by `-O2` and also turns on additional optimisations.
- `-O0`. Reduce compilation time and make debugging produce the expected results. This is the default.
- `-Os` (Optimise for size). `-Os` enables all `-O2` optimisations except those that often increase code size. It also enables `-finline-functions`, causes the compiler to tune for code size rather than execution speed, and performs further optimisations designed to reduce code size.
- `-Og` (Optimise debugging experience). `-Og` should be the optimisation level of choice for the standard edit-compile-debug cycle, offering a reasonable level of optimisation while maintaining fast compilation and a good debugging experience. It is a better choice than `-O0` for producing debuggable code because some compiler passes that collect debug information are disabled at `-O0`.

4.5.3. Program Analysis Techniques

In addition to the requirement that the function of the embedded system is correctly implemented, embedded systems also have to fulfil extra-functional design constraints. To fulfil these design constraints good analysis methods are required to determine among others, the worst cases execution time, the power that is consumed by the embedded system, and the required size of the memory components. An accurate estimate about the performance and cost of the system can be used at several stages of the design process and enables a good start for design optimisation, an accurate implementation of the design, and to keep safety margins minimal in the final implementation.

For hard real-time embedded systems worst case execution time is of critical importance. The term WCET defines the longest execution time that can occur during execution of the program. Unfortunately, the WCET is difficult to determine, because of cache and pipeline effects, or compiler optimisations Furthermore, the program can take many different path during execution due to 1. varying combinations of input data and 2. different states of the program. In addition, the execution time of an operation maybe data-dependent as in the case of a floating-point operation, where a multiplication with zero takes much less time than multiplications with other floating-point-numbers.

Due to these effects, it is very difficult to determine the WCET by measurement. A measurement-based approach will require 1. representative input data, and a 2. hardware timer and extra instrumentation of program, which in turn might affect the execution time of the program.

A tool that can be used to determine the execution time of a program or parts of the program is a *profiler*. The profiler of GNU project `gprof` records the number of calls to each function and the amount of time spent there, on a per-function basis. In this way, functions which consume a large fraction of the run-time can easily be identified easily from the output of `gprof`. Thus, a profiler can be used at an early design stage to identify the parts of the program that will likely consume most time, even if executed on another platform, and should be investigated in more detail.

Given the following two functions

```c
#include <stdio.h>

int series(int n)
{
  int y = 1;
  int i;
  for(i = 1; i <= n; i++)
    y += i;
  return y;
}

int fib(int n)
{
  int y;
  if (n == 1)
    y = 1;
  else
    if(n == 2)
      y = 2;
    else
      y = fib(n-1) + fib(n-2);

  return y;
}
```

and the corresponding main program

```
int main()
{
  int i;
  int n = 35;
  for(i = 1; i < n; i ++)
    {
      printf("series(%d) = %d\n", i, series(i));
      printf("fib(%d) = %d\n", i, fib(i));
    }
  return 0;
}
```

the profiler's output shows, how often each function is called and how much time is spent in each function.

```
granularity: each sample hit covers 4 byte(s) for 1.59% of 0.63 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.04    0.59                 main [1]
                0.57    0.00      34/34          fib [2]
                0.01    0.00      34/34          series [3]
-----------------------------------------------
                             29860634             fib [2]
                0.57    0.00      34/34          main [1]
[2]     91.3    0.57    0.00      34+29860634 fib [2]
                             29860634             fib [2]
-----------------------------------------------
                0.01    0.00      34/34          main [1]
[3]      2.4    0.01    0.00      34         series [3]
-----------------------------------------------
```

Another approach is the simulation of the program, but this requires an accurate model of the CPU its and peripheral circuits, where all necessary details might not be published by the company that fabricates the hardware components.

A third approach is to derive the WCET analytically, by using the intermediate representation of the source code and a performance model of the processor system. There is a lot of active research on WCET analysis, but the problem is very challenging. Modern processor architectures are very difficult to analyse due to advanced features to improve average case performance, such as caches, pipelines, or branch prediction techniques. Also, due to advanced compiler optimisation techniques, it is in practice not feasible to analyse the source code of the high-level language, like C or Ada, but for an accurate WCET analysis, the binaries of the compiled code need to be analysed. A good and detailed overview of WCET analysis is given in (Wilhelm et al., 2008). The following example will introduce the basic idea to analyse the WCET.

The main idea to determine the WCET is to analyse the CDFG and to identify the longest execution time. Here, all possible paths of the program have to be analysed to determine the WCET for each path.

The principle of WCET analysis is illustrated by means of the `for` of Figure 4.12.

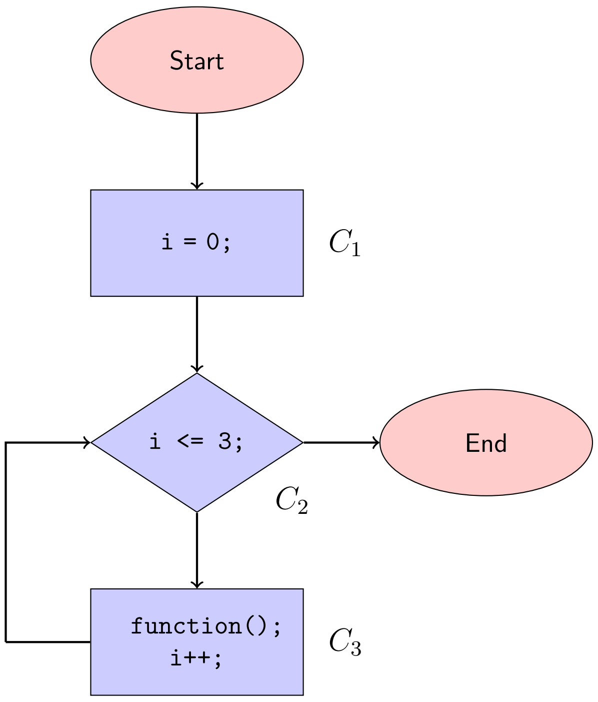


Figure 4.12.: Determining the WCET for a simple `for`-loop

Here, the loop initialisation, with the WCET (C_1) is executed once, the decision node (C_2) is executed five times, and the loop body (C_3) is executed four times. Thus the WCET for the whole program can be determined as

$$C_1 + 5C_2 + 4C_3 \tag{4.9}$$

if all WCETs C_1, C_2, and C_3 are known.

Many embedded systems need also to be optimised for a low power or energy consumption. Here, it is important to distinguish between energy and power. Energy consumption is important for battery-driven devices. Power is the consumed energy per time unit and determines the temperature of the device. Lowering only the frequency reduces power, but does not reduce energy consumption, because the program will double its execution time. Simultaneous frequency and voltage scaling reduces both energy and power.

In general, to optimise a program for energy and power consumption follows the same guidelines as the design for efficient embedded systems. External memory transfers should be minimised, cache sizes should be adapted to the size of the working set, and registers and caches should be used efficiently. Processors do not have specific low-power instructions, but there are processors, which are optimised for a low power consumption.

4.5.4. Optimising for code size

For embedded systems it is often of key importance to minimise the memory footprint, the amount of memory that a program requires during execution. In the following, different measures to minimise the memory footprint for programs running on a Nios II soft processor on an Intel FPGA will be discussed. The Nios II processor only serves as an example, for other processor families similar optimisation techniques should be available.

To determine the amount of memory on a Unix system, the command `size` is available to determine the memory footprint of an executable. For the Nios II processor system, the corresponding command is instead `nios2-elf-size` and takes an binary ELF file as argument.

```
$ nios2-elf-size Lab2_Functions.elf
   text    data     bss     dec     hex
  71320    7652     548   79520   136a0
```

The sections in the output have the following meaning.

- `text` gives the size of the code of the executable instructions,
- `data` gives the size of the initialised data section,
- `bss` gives the size of the uninitialised data section,
- `dec` is the overall program size as a decimal number, and
- `hex` is the overall program size as a hexadecimal number.

There is a large potential to reduce code size for Nios II applications. The Nios II system library includes the drivers that are needed for the underlying hardware, but by default fast drivers are installed, and the full newlib ANSI C library is installed.

The first step is to use the flag `-Os` for compiler optimisation. This flag shall be used to optimise both the *application library* (user code) and the *system library* (board support package).

For many hardware devices, there exist two different device drivers. A fast driver, supporting all features of the device, and a small driver, optimised for the memory footprint. Although, the small device drivers do have not full functionality, for instance the Nios II UART and JTAG-UART support only polled operation instead of IRQ-driven operation, they might be sufficient.

Also, there is a smaller version of the newlib C library, which at the expense of supporting only simple I/O functions, has a lower memory footprint. There is also a minimal character-mode API, which limits the use of character-mode I/O to very simple features and includes the functions `alt_printf()`, `alt_putchar()`, `alt_putstr()`, and `alt_getchar()`. As an example, `alt_printf()` supports only `%c`, `%s`, `%x` and `%%` substitution strings.

By default for each hardware device in a Nios system a driver is created. Drivers that are not needed by your software application can be excluded.

The Nios II Hardware Abstraction Layer (HAL) library executes the `exit()`-function for a clean exit from the program, but many embedded systems will never exit. In this case, the generation of the exit-code can be disabled. Also by default, C++ constructors and destructors are supported. This option can also be turned off.

5. System Modelling

5.1. Introduction to System Modelling

5.1.1. Requirement Specification

System design starts with the specification of the requirements that the final implementation of the system has to fulfil. Requirement specification is a key activity in system design, where it is very important that the *requirement specification*

- is *complete*, i.e. no relevant requirement is missing in the requirement specification,
- is *free from inconsistent requirements*, i.e. two requirements that contradict each other, and
- is *free from ambiguous requirements*, i.e. requirements that can be interpreted in different ways.

Requirements can be divided into *functional requirements*, i.e. requirements that specify the functionality of the system, or *non-functional* or *extra-functional requirements*, i.e. requirements that define properties of the system that are not directly related with the functionality of the system. Such non-functional requirements could be requirements on power consumption, timing, safety, reliability or design costs. Another term for non-functional requirement is *design constraint*.

The course will not discuss requirement engineering, the activity to derive a requirement specification, but assumes that there exists already a requirement specification, which has to be fulfilled by the final implementation.

5.1.2. Importance of System Modelling

The course will follow the design flow of Figure 1.5, which sketches a design flow starting from the initial system model down to the final implementation, which shall fulfil both the functional requirements and non-functional requirements (design constraints) specified in the requirement specification. Figure 1.5 envisions a correct-by-construction design flow, but the same activities are also part of a traditional industrial design flow, where many steps are not formalised or automated.

System modelling is a key activity in the design process. During system modelling, the system is modelled at a high level of abstraction. Embedded systems become increasingly complex, which also makes the design process increasingly challenging. Two important techniques are used to cope with this challenge at different levels of the design flow:

1. *abstraction*, i.e. the initial system model needs to be modelled at a high level of abstraction, and
2. *partitioning*, i.e. the system is composed into smaller subsystems, which can be analysed and designed separately.

The system model can describe both functionality, the embedded platform and other aspects. This chapter will only discuss the functional aspects of the system model. The embedded platform is described in Chapter 2, but needs also to be abstracted to be able to use it efficiently in phases like design space exploration (Section 6.1). Figure 1.3 shows how the *abstraction gap* is bridged between an abstract high-level specification and a detailed low-level implementation. During the design process, the initial specification model is converted into more detailed models using refinement steps, which yield a model at a lower level of abstraction. These refinement steps are *design decisions*, which are taken during the design process. After the application of a number of refinement steps, the final implementation is generated. The implementation needs to fulfil the *design space* that has been defined by the functional and non-functional requirements in the requirement specification.

The system model is very important for the design process, since it is the starting point for the further refinement of the system. Thus, the system model should not only capture the functionality of the system, but also support the succeeding activities of the design process. A good system model

- defines the functionality the system has to perform,
- enables to simulate or formally verify the functionality of the system, so that designers can confirm that the intended functionality is modelled correctly,
- uses a formal language with a standardised semantics, so that the model has one clear meaning,
- abstracts from low-level details, but provides at the same time a path to the final implementation,
- abstracts from implementation techniques, so that the final implementation can use different implementation techniques, like hardware, bare-metal software or real-time operating systems, and
- allows the designer to focus on the most important aspects.

A good system model also gives a good base for a discussion between designers. System models can be both graphical or textual. Errors at an early stage of the design process can be avoided, if the system model can be simulated or formally verified, which can prevent high unnecessary design costs. It is also claimed that good models shorten the design process, since they both prevent errors at an early stage, and also might enable automatic code generation to yield the final implementation.

Another important aspect is that the modelling language should support the development of tools, which are used during the design process, such as tools for

- simulation
- performance analysis and design space exploration,
- compilation and synthesis, or
- formal verification.

The simpler the modelling language is and the cleaner the formal semantics are defined, the easier it is to develop powerful design tools. Thus, an expressive modelling language, with many language components and a complex formal semantics, will allow to express many different models, but makes it more difficult to develop the necessary tools for the design process. Here, it is important to find the right trade-off between the expressiveness and the simplicity of the modelling language.

5.1.3. Unified Modelling Language: A Short Overview

The Unified Modelling Language (UML) is one of the two dominating languages for the modelling of software systems. The other large modelling language, MATLAB/Simulink, is not discussed here. UML is a graphical, object-oriented language, standardised by the Object Management Group (OMG). The discussion will give a short overview of the UML 2 standard, for further information see (Haugen et al., 2005). The focus will be on general modelling concepts, which are part of UML, but can and are used outside UML. UML supports several additional modelling concepts and diagrams, but practical industrial designs use in general only a restricted set of UML diagrams.

Finite State Machines The Finite State Machine (FSM) is a key element in UML to express control-dominated behaviour. The UML state machines are based on Harel's Statecharts (Harel, 1987; Harel and Pnueli, 1985), and offer advanced concepts to extend the Finite State Machine (FSM) concept by for instance hierarchically nested states, and composite states (AND-state, OR-state), which reduce the size of the graph. FSMs are often associated to hardware design, an FSM is a general concept and heavily used for the design of software systems. Thus, it is a very important modelling concept in UML.

Petri Nets A Petri net (Murata, 1989) consists of a set of *tokens*, a set of *places* and a set of *transitions*. Tokens reside in places and circulate through the Petri net by being consumed and produced whenever a transition *fires* (executes). A transition in a Petri net can only fire, if it is enabled, i.e. each input place has at least one token. After firing, the tokens in the input places are moved to the output places. The principal execution of a Petri net is illustrated in Figure 5.1.

The execution of a Petri net is *nondeterministic*. This means, multiple transitions can be enabled at the same time, where any one of these can fire. Since firing is nondeterministic, Petri nets are well-suited for modelling the *concurrent* behaviour of distributed systems, as illustrated in Figure 5.2.

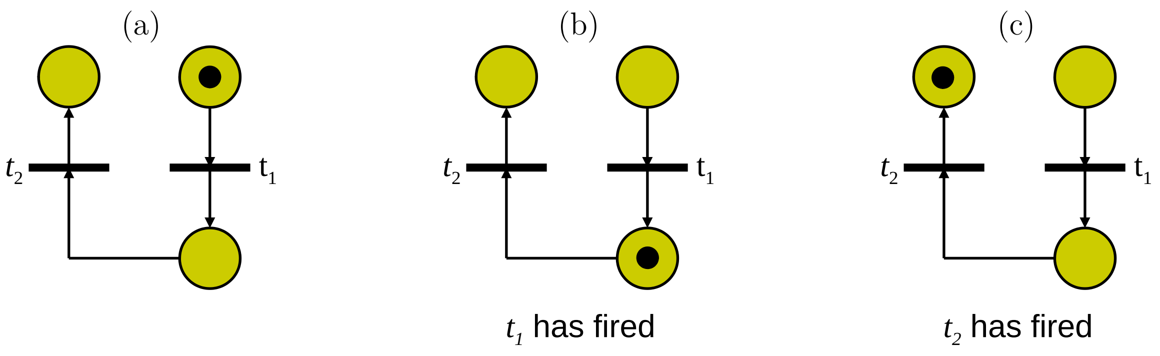


Figure 5.1.: In the Petri net (a) the transition t_1 is enabled due to the token in the place preceding t_1. After the firing of t_1, the token is transferred to the place succeeding the transition t_1 (b). After the firing of t_2, there is no more enabled transition, which can fire, and the Petri net is in a final state.

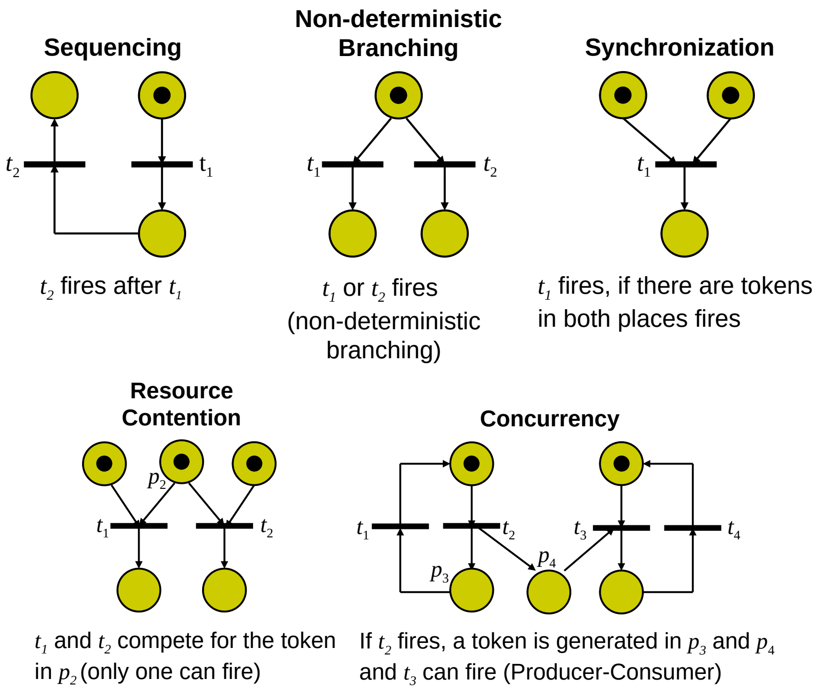

Sequencing
t_2 fires after t_1
Non-deterministic Branching
t_1 or t_2 fires (non-deterministic branching)
Synchronization
t_1 fires, if there are tokens in both places fires
Resource Contention
p_2
t_1 and t_2 compete for the token in p_2 (only one can fire)
Concurrency
t_1 t_2 t_3 t_4 p_3 p_4
If t_2 fires, a token is generated in p_3 and p_4 and t_3 can fire (Producer-Consumer)

Figure 5.2.: Petri nets can be used to model different communication mechanisms in concurrent networks.

A distribution of tokens over the places of the net is called a *marking* M. The initial marking of the Petri net in Figure 5.3 is $M = \{1, 1, 0, 0, 0\}$. After the firing of t_2, the new marking is $M = \{0, 1, 1, 1, 0\}$. A marking gives the state of the Petri net.

Petri Nets can be used to check important properties formally, e.g.

- *Reachability*: Is it possible that a state can be reached from an initial marking? This is important to analyse, if a wrong state, like an elevator moving with an

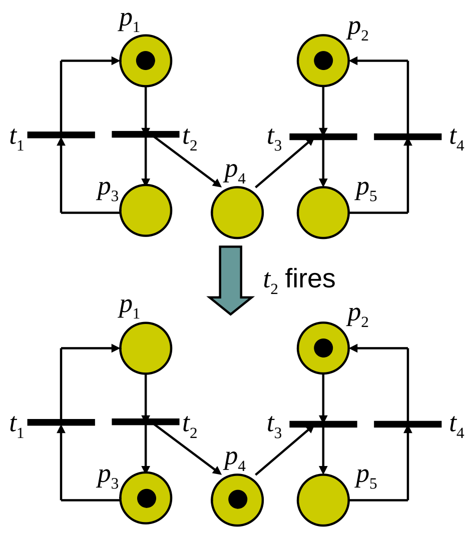


Figure 5.3.: The state of a Petri net can be described with a marking, that gives the number of tokens in all places.

open door, can occur.

- *Liveness*: Is there always at least one transition that can be fired? If not, the system might end in a state, from which the system can never proceed.

Petri nets provide an elegant and simple modelling paradigm that is very suitable for the modelling and analysis of concurrent systems. However, Petri nets can quickly grow and become incomprehensible, when the system complexity increases. There are many different variations of Petri nets. *Activity diagrams* in UML 2 are based on Petri nets.

Sequence Diagrams Sequence diagrams describe interactions between two parts of a system. The horizontal lines represent messages, while the vertical lines represent time. Sequence diagrams can be hierarchical and have powerful constructs like loops, like alternative and optional interactions. An example of two sequence diagrams to enter the pin code for an access control is shown in Figure 5.4.

Component Diagrams The previous diagrams described the behaviour of the system. Component diagrams are needed to describe the connection between components in the system as illustrated in Figure 5.5. Ideally the behaviour of a component can be *fully separated* from the communication with other components by means of the interface. This leads to a separation of concerns, so that the computation in form of the component's function can be separated from the communication between system components.

Another diagram that gives structural information is the class diagram, which shows the system's classes, their attributes and methods, and the relation between different classes.

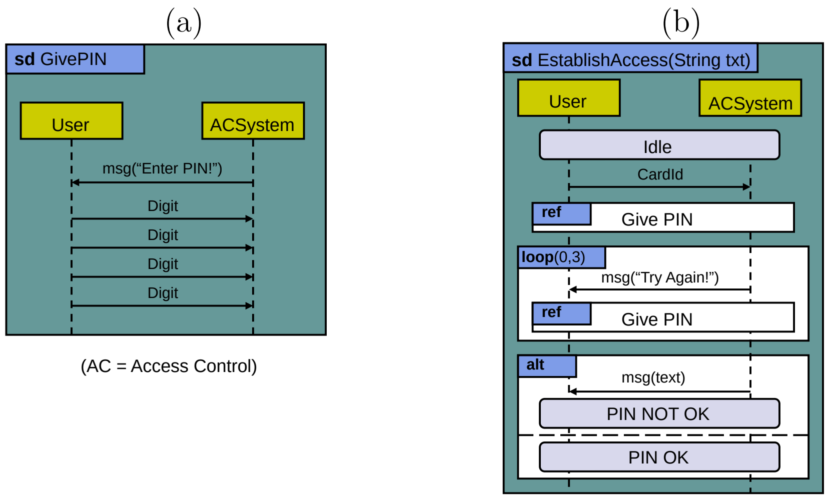


Figure 5.4.: Sequence diagrams enable to express the expected behaviour of specific use cases, but can also be used as an output of a simulation.

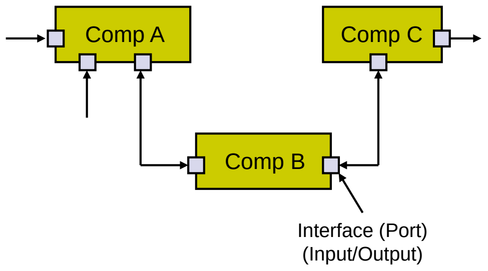


Figure 5.5.: The component diagram shows how different components are connected to form a larger software system. The components communicate with each other via interfaces or ports.

Discussion UML offers many diagram types for different modelling purposes. Thus, it is a very expressive language, since a UML model can consist of many different diagrams and use many modelling concepts. However, this is not always an advantage. Although the semantics of each individual modelling concept (diagram) is often well-defined, it is very difficult to find a simple and clean semantics for the composition of different modelling concepts.

5.1.4. Models of Computation

To discuss different modelling concepts, the course focuses on the theory of Models of Computation (MoCs) (Lee and Sangiovanni-Vincentelli, 1998). A Model of Computation (MoC) is a mathematical description that is used to specify the semantics of computation and concurrency, where processes or actors communicate with each other only via signals as illustrated in Figure 5.6. According to Lee and Seshia (2017) a MoC is specified by rules which specify (a) what constitutes a component, (b) the concurrency mechanism, and (c) the communication mechanism. A MoC abstracts from a programming language.

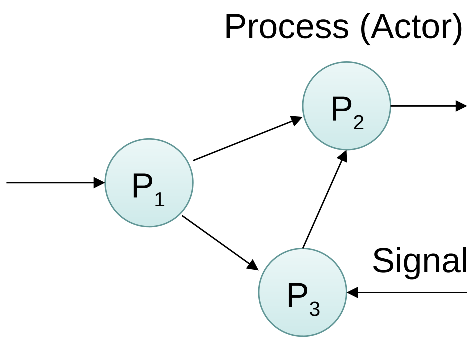


Figure 5.6.: In a model of computation, processes or actors communicate with each other only via signals.

Since MoCs are formally defined they can be mathematically analysed, which enables the development of tools for simulation, performance analysis and design space exploration, formal verification, and synthesis and code generation. Several MoCs are relevant for embedded systems. The *synchronous MoC* forms the base for the family of synchronous languages and also synchronous hardware, the *data flow MoC* supports the development of streaming media applications, the *discrete-event MoC* is used for simulation tools, for instance VHDL and Verilog simulators, and the *continuous time MoC* is used for the simulation of analogue circuits or the physical environment. The course will focus on data flow MoCs and the synchronous MoC, since these two MoCs are most relevant for the design and analysis of embedded systems. Before general aspects on the theory of MoCs are formally defined in Section 5.4, the data flow MoC is introduced in Section 5.2, and the synchronous MoC in Section 5.3.

5.2. Data Flow Model of Computation

5.2.1. Data Flow Graphs

Many streaming media application can be naturally described by the flow of data, which can be captured in a data flow graph. Examples are digital processing systems using digital filters, fast Fourier transform or other signal processing algorithms. These systems consume infinite streams of input data tokens and produce infinite streams of output data tokens.

Figure 5.7 illustrates a data flow graph. In general, a data flow graph consists of a set of *nodes* (vertices), which communicate via *arcs* (edges). A node can *fire* (execute) whenever sufficient data tokens are available on each input arc. A node represents a computation function or program that will be executed, when the node is fired. The arcs represent infinite FIFO queues, which are used to carry infinite streams of data tokens from the output of one node to the input of another node. The execution of a data flow graph is data-driven and purely based on data dependencies. There is no other way of communication than sending tokens from one node to another node via an arc.

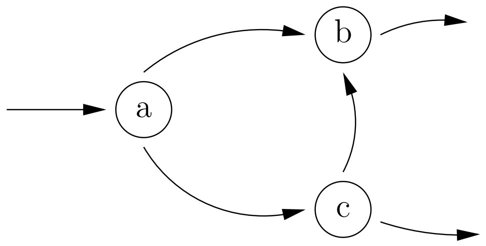


Figure 5.7.: A data flow graph consists of nodes (vertices), which are connected via arcs (edges)

There exist different terminologies for data flow graphs. Currently, the term actor is the standard term for a node, but also the term process is used. For the arcs, the terms signal or channel are often used. The course will mainly use the terms *actor* and *signal*.

For general data flow, the number of tokens consumed and produced when firing a node can vary from firing to firing and depends on the state of the actor. The functionality of the actor can be modelled or implemented as a sequential program. The data flow graph can be viewed as a concurrent program, which can be executed on a parallel computer. General data flow is expressive, but due to the dynamics caused by the varying consumption and production rates in different firings, the behaviour is difficult to predict and properties of a general data flow graph are difficult to analyse. Thus, in general data flow, there is no general method to derive a feasible static schedule for the execution of the nodes or to calculate the required buffer sizes for the FIFOs on each signal. This is in particular a problem for the design of safety-critical systems, where the correctness and worst case performance of the final implementation needs to be guaranteed.

There is a trade-off between *expressiveness* and *analysability* when comparing different modelling languages and paradigms. A high expressiveness means that the modelling paradigm enables to express many different models or programs, while a high analysability means that the modelling paradigm enables to analyse the model and to prove or derive certain properties of the model. Current research tries to raise the level of expressiveness without sacrificing too much analysability.

5.2.2. Synchronous Data Flow

Overview and Definition

Synchronous Data Flow (SDF) (Lee and Messerschmitt, 1987b,a) is a special case of data flow that aims at providing high analysability for a relevant subset of data flow applications. In order to increase analysability, SDF puts the following constraint on each actor: the number of consumed tokens on each input channel and the number of tokens produced on each output channel is static and will never change during the execution of the SDF graph. Lee and Messerschmitt (1987a) used the term *synchronous* to define a synchronous node, where the numbers of input tokens that are consumed on each input and the number of output tokens that are produced on each output can be specified a priori and will never change. This has led to some confusion, since the term 'synchronous' is also used for the family of synchronous programming languages (Benveniste and Berry, 1991), where in particular languages like Lustre (Halbwachs et al., 1991) and Signal (Le Guernic et al., 1991) are *synchronous data flow languages*, where the term 'synchronous' is used in the sense of the *synchronous MoC* (Section 5.3). It seems that the term *static data flow* had been a better term then synchronous data flow (SDF), but now the term 'synchronous data flow' is well established in the community and difficult to change. The following presentation of synchronous data flow is to a large extent based on the seminal papers of Lee and Messerschmitt (1987b,a).

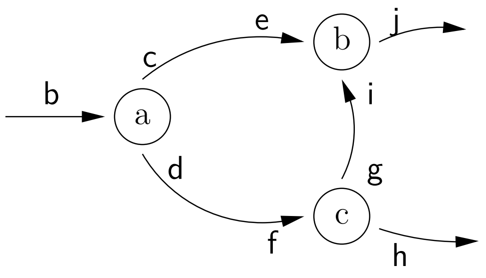


Figure 5.8.: Synchronous Data Flow Graph

Figure 5.8 shows an example of a Synchronous Data Flow Graph (SDFG). The number of tokens consumed on each arc during a firing of an SDF-actor is annotated as a label on the sink of the arc, while the number of tokens produced by an SDF-actor during each firing is annotated as a label on the source of the arc. Thus, the SDF-actor a will during each firing consume b tokens on its input arc, and produce c and d tokens on its outgoing arcs. The number of tokens produced and consumed is static and will never change. During each firing, each SDF-actor will execute a certain computation function. This function might have a local state, but this state will not affect the number of tokens consumed or produced. If all actors in a data flow graph are synchronous data flow actors, then the data flow graph is a synchronous data flow graph.

Towards an implementation of an SDF Graph

In order to implement a data-flow graph, the semantics of the data flow MoC needs to be fully satisfied in the final implementation. This requires to

- implement the communication between the actors compliant to the semantics of the used data flow MoC, and to
- implement the functions that the actors execute in the target implementation (not covered in this lecture).

A faithful implementation of the communication semantics requires that an actor only fires, when sufficient data is available on all input arcs and that there is sufficient buffer space available on the output arcs. The assumption of infinite FIFO-buffers in the data flow MoC cannot be implemented in practice, and would not be efficient either. Thus, a correct implementation of the communication semantics of a data flow graph requires to

- determine the required size of the FIFO-buffers so that there is sufficient buffer space for each arc, and to
- derive a schedule so that the actors are fired when data is available.

There are different ways to implement the scheduling. A *dynamic scheduler*, like a Real-Time Operating System (RTOS), could be used that schedules the actors at run-time. If a *static schedule* can be derived, the overhead of the dynamic scheduler can be avoided, and instead the system can be implemented on a bare-metal processor, i.e. a processor that does not run any OS.

The big advantage of Synchronous Data Flow (SDF) is that it enables to derive a static schedule, if it exists, at compile time. This method is carefully described by Lee and Messerschmitt (1987a,b), where the resulting schedule is called Periodic Admissable Sequential Schedule (PASS), if it is a schedule for a single processor computer, or Periodic Admissable Parallel Schedule (PAPS), if it is a schedule for a parallel computer. Here, an admissible schedule is a correct schedule, where only a finite buffer memory is required. Furthermore, if such a schedule exists, the required FIFO-buffer sizes can also be determined. The method assumes that the SDF graph is non-terminating and does not deadlock, and that the SDF graph is connected.

Periodic Admissible Sequential Schedule (PASS)

In the following, the original examples from Lee and Messerschmitt (1987a,b) are used. The first step is to find a formal representation, which captures the topology of the SDF graph and can be used to determine a possible PASS or PAPS.

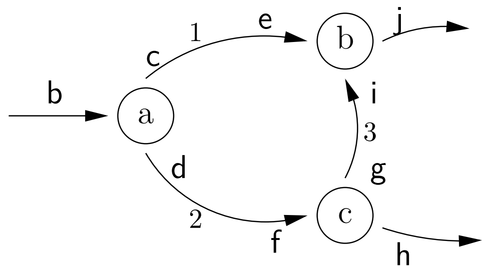


Figure 5.9.: SDF graph

A *topology matrix* can be used to identify, if a PASS or PAPS exists. For the calculation of the schedule, the incoming arcs can be discarded. Thus the following topology matrix Γ describes the Synchronous Data Flow Graph (SDFG) of Figure 5.9, but does not express the incoming and outgoing arcs.

$$\Gamma = \begin{bmatrix} c & -e & 0 \\ d & 0 & -f \\ 0 & -i & g \end{bmatrix}$$

The entry of row i and column j is the number of tokens produced (positive number) or consumed (negative number) by node j on arc i. Thus the first column describes the node `a`, where the value c in the first row denotes the number of tokens produced on arc `1`, the value d in the second row denotes the number of produced on arc `2`, and the value `0` in the last row of the first column expresses that node `a` is not connected to arc `3`. A positive number expresses that tokens are produced on an arc, while a negative number expresses that tokens consumed on an arc. Thus, the value `-e` in the first row of the second column denotes that node `b` consumes e tokens from arc `1`.

Lee and Messerschmitt (1987a) have proven the important Theorem 5.1.

Theorem 5.1. For a connected SDF graph with s nodes and topology matrix Γ, $\text{rank}(\Gamma) = s - 1$ is a necessary condition for a PASS to exist.

For an SDF-graph that fulfils Theorem 5.1, it is possible to give a *repetitions vector* q, which is defined by Sriram and Bhattacharyya (2009) as in Definition 5.1.

Definition 5.1. The *repetitions vector* q for an SDF graph with s actors numbered 1 to s is a column vector of length s, with the property that each actor i is invoked a number of times equal to the i-th entry of q, then the number of tokens on each edge of the SDF graph remains unchanged. Furthermore, q is the smallest integer vector for which the property holds.

Theorem 5.2 is given in (Sriram and Bhattacharyya, 2009) and follows from (Lee and Messerschmitt, 1987a), which also includes the proof, and gives the relation of the repetitions vector q and the topology matrix Γ.

Theorem 5.2. The repetitions vector for an SDF-graph with consistent sample rates is the smallest integer vector in the null-space of its topology matrix Γ. That is, q is the smallest integer vector such that $\Gamma q = 0$

The following example illustrates the process to determine a PASS for the SDFG of

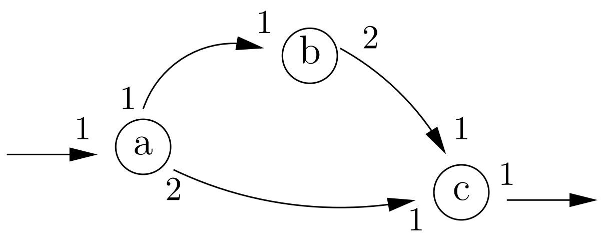


Figure 5.10.: SDFG for the running example

Figure 5.10.

In order to calculate a possible PASS and the required buffer sizes, the first step is to derive the topology matrix Γ for the SDF-graph.

The incoming and outgoing arcs are not represented in the topology matrix.

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 2 & -1 \\ 2 & 0 & -1 \end{bmatrix} \tag{5.1}$$

In order to know, if a Periodic Sequential Schedule (PSS) exists, the rank s of the matrix needs to be determined. The rank is given by the number of independent rows in the matrix. Gaussian elimination is used to derive the rank of the matrix.

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 2 & -1 \\ 2 & 0 & -1 \end{bmatrix} \overset{(1)}{=} \begin{bmatrix} 2 & -2 & 0 \\ 0 & 2 & -1 \\ 2 & 0 & -1 \end{bmatrix} \overset{(2)}{=} \begin{bmatrix} 2 & 0 & -1 \\ 0 & 2 & -1 \\ 2 & 0 & -1 \end{bmatrix} \overset{(3)}{=} \begin{bmatrix} 2 & 0 & -1 \\ 0 & 2 & -1 \\ 0 & 0 & 0 \end{bmatrix} \tag{5.2}$$

During Gaussian elimination the following steps have been used: (1) Multiply row 1 with 2; (2) Add row 2 to row 1; (3) Subtract row 1 from row 3. Since the matrix has two independent rows, $\text{rank}(\Gamma) = s - 1 = 2$, so a PASS can exist.

The next step is to derive the repetitions vector q. To determine the vector a set of independent linear equations, also called *balance equations* has to be solved, such that $\Gamma q = 0$. Here, the equations are taken from row 1 and 2 of the original topology matrix.

$$\begin{array}{ccccccc} q_1 & - & q_2 & & & = & 0 \\ & & 2q_2 & - & q_3 & = & 0 \end{array} \tag{5.3}$$

It follows that the minimal vector q that solves the balance equations, is the repetition vector $q^T = \begin{bmatrix} 1 & 1 & 2 \end{bmatrix}$ (transposed form).

This means that actor a and b have to be fired once in each iteration, while the actor c has to be fired twice in each iteration. There are several possible permutations, but not all of them will yield a schedule that constitutes a PASS. The schedule $\phi = \{a, b, c, c\}$ constitutes a PASS, but not the schedule $\phi = \{b, a, c, c\}$, because b does not have sufficient tokens to be fired at the beginning of the schedule.

The required buffer sizes on each arc can be calculated by simulating the schedule for a full iteration. Here, the required buffer sizes are $b^T = \begin{bmatrix} 1 & 2 & 2 \end{bmatrix}$ (transposed form).

Thus, the a buffer size of 1 is required for the arc from a to b, and a buffer size of 2 for the arc from b to c and the arc from a to c. This concludes the example.

The SDF graph in Figure 5.11 does not have a PASS. The FIFO-buffer on the arc

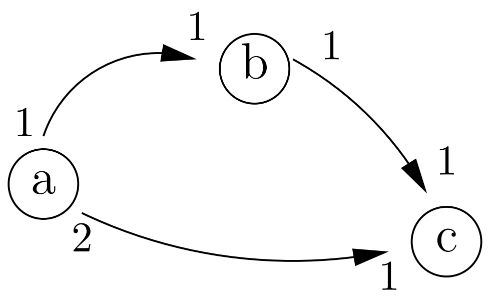


Figure 5.11.: No PASS exists for this SDF graph.

between the nodes a and c will require an unbounded buffer size. This can also be shown by the calculation of the rank of the matrix, which in this case equal to the number of nodes and thus violates Theorem 5.1.

The graphs in Figure 5.12 have a PSS, but a PASS does not exist, because the

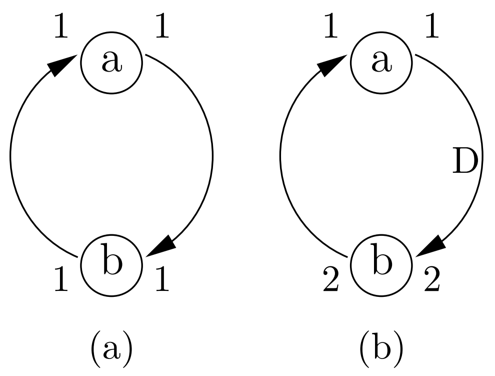


Figure 5.12.: Two SDF graphs that have a PSS, but no PASS exists

- SDF graph (a) is not computable (no delay element in the feedback loop)
- SDF graph (b) has not enough delay elements

Both graphs require additional delay elements (D), so that a PASS can be constructed. A delay element is an initial token that already exits on the arc before the first firing. These initial tokens are often also shown as filled circles on the arcs.

Periodic Admissible Parallel Schedule (PAPS)

In the following, a short overview is also given about the derivation of a PAPS, which can be executed on a parallel computer. The SDF graph of Figure 5.13 will be used as example. The example assumes that each actors has a given computation time C. Here,

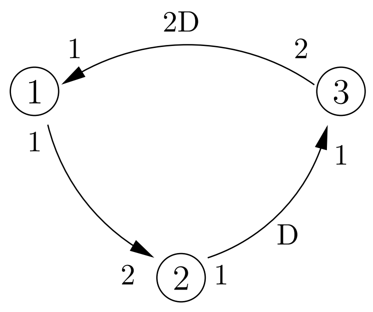


Figure 5.13.: SDF graph

the computation time of actor 1 is $C_1 = 1$, the computation time of actor 2 is $C_2 = 2$, and the computation time of actor 3 is $C_3 = 3$. It is assumed that communication takes no time.

If a schedule for a single processor (PASS) exists, then there also exists a schedule for a parallel computer (PAPS), since all computations can be scheduled on one single processor of the parallel computer. Figure 5.14 shows this trivial case.

Figure 5.15 shows how the performance can be improved by exploiting the potential parallelism in the SDF graph. Here the schedule covers one single period. The initial tokens enable the execution of actor 3 in parallel to actor 1 and 2.

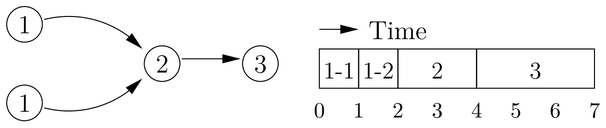


Figure 5.14.: Trivial case: A PAPS exists - all actors are executed on the same processor

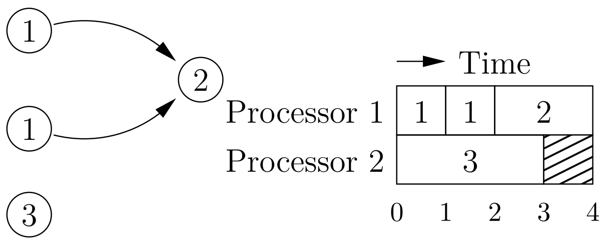


Figure 5.15.: Parallel execution is enabled due to initial tokens on the arcs

An even better schedule can be achieved, if the schedule is constructed over two periods as shown in Figure 5.16.

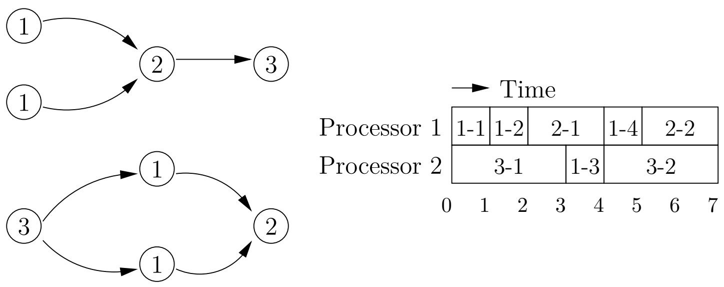


Figure 5.16.: Parallel execution is improved by considering two periods

There has been a lot of research since the initial work of Lee and Messerschmitt (1987a,b) on the scheduling and implementation of SDF graphs on multiprocessors. The textbook by Sriram and Bhattacharyya (2009) provides an very good overview about different techniques.

5.2.3. Homogeneous Synchronous Data Flow (HSDF)

A Homogeneous Synchronous Data Flow Graph (HSDFG) is a special case of a SDFG in which every actor consumes exactly one token from each of its input arcs and produces exactly one token on each of the output arcs during each firing. It is possible to convert an SDFG into a Homogeneous Synchronous Data Flow Graph (HSDFG) using the repetitions vector (Definition 5.1). The repetitions vector q specifies the number of invocation of each actor during an iteration period.

Assuming the SDFG of Figure 5.17, which has the repetition vector $q^T = \begin{bmatrix} q_A & q_B & q_C \end{bmatrix} = \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$. Since an Homogeneous Synchronous Data Flow (HSDF) actor only consumes and produces a single token during each firing, the resulting HSDFG will contain

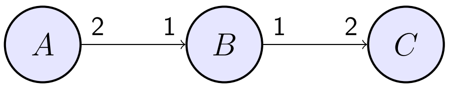


Figure 5.17.: Synchronous Data Flow Graph

1. four actors A', B'_1, B'_2, and C', and
2. four arcs $A' \to B'_1$, $A' \to B'_2$, $B'_1 \to C'$, and $B'_2 \to C'$,

and is illustrated in Figure 5.18.

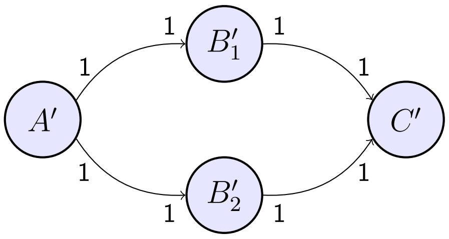


Figure 5.18.: Corresponding HSDFG for the SDFG of Figure 5.17

In order to preserve the semantics of the initial SDFG, where the tokens on the arcs $A \to B$ and $B \to C$ are processed in FIFO-order, it is required that this order is also preserved in the resulting HSDFG. This can be achieved by ordering the different input arcs, so that the tokens are produced and consumed in FIFO-order. In this example, this would mean that in each firing of

- node A' the first token is emitted on the arc $A' \to B'_1$, and the second token on the arc $A' \to B'_2$,
- node C' the token on the arc $B'_1 \to C'$ needs to be processed before the token on the arc $B'_2 \to C'$.

The conversion from an SDFG to a HSDFG is often done to increase the possible parallelism in a data flow graph. However, as written before, the FIFO-order relation given by the arcs in the SDFG needs to be preserved in the final implementation.

Algorithms for the conversion of SDFGs to HSDFGs exist, see for instance Sriram and Bhattacharyya (2009) for such an algorithm and a further discussion about HSDFGs and their implementation.

5.2.4. Boolean Data Flow

Boolean Data Flow (BDF), originally called token flow model (Lee, 1991; Buck and Lee, 1993), extends SDF with additional actors, that have to fulfil the condition “that the number of tokens produced or consumed on each arc must either be constant, or a function of a Boolean-valued token produced or consumed on another arc, which is called a control arc” (Buck and Lee, 1993).

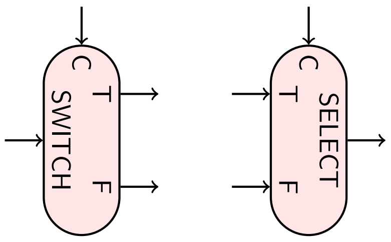


Figure 5.19.: The Boolean Data Flow (BDF) actors SWITCH and SELECT

Figure 5.19 shows the two basic Boolean actors SWITCH and SELECT. The SWITCH-actor consumes a normal data token on the SWITCH-input and a Boolean-valued control token on the control input (C). The input data token is emitted on the output T, if the control token has the value *true*, or, if the control token has the value *false*, on the output F. In both cases no token is produced on the other arc. The actor SELECT consumes a token from the input T and emits it at the output, if the consumed Boolean-valued control input token has the value *true*, or, if the control input token has the value *false*, consumes a token from input F and emits it as output token. In both cases no token is consumed from the other input arc. Thus, the production rate on the outgoing arcs can vary for different firings of the SWITCH-node, which is in contrast to SDF, where the requirement is that consumption and production rates in an SDF-actor never change. Also the SELECT-node does not fulfil this requirement due to the varying consumption rates.

The main idea of BDF enables to model dynamic data flow applications. In particular, the actors SWITCH and SELECT enable to express if-then-else control structures, as illustrated in Figure 5.20, where the actors A - E are SDF actors.

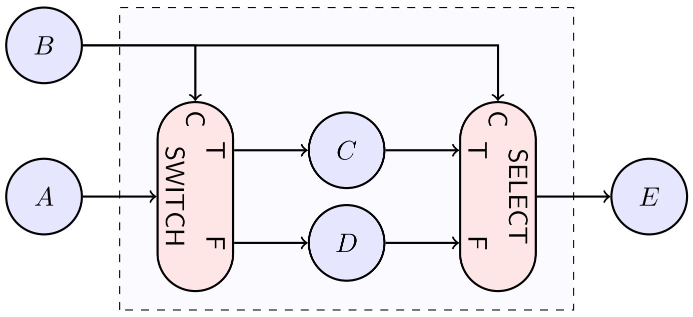


Figure 5.20.: An if-then-else BDF graph, which behaves as SDFG

The actor B produces Boolean-valued tokens, which are consumed at the control inputs C of the actors SWITCH and SELECT. Although the complete BDF graph includes dynamic BDF actors, the composed BDF graph in the dashed area behaves like a SDF graph, because the consumption and production rates for this composed graph are constant.

By enabling dynamic actors like SWITCH and SELECT, BDF provides a huge *expressiveness*, which corresponds to turing-completeness (Buck and Lee, 1993). Unfortunately, this also means that general BDF graphs provide a low analysability.

5.2.5. Cyclo-Static Data Flow

The aim of Cyclo-Static Data Flow (CSDF) (Bilsen et al., 1996) is to increase the expressiveness of SDF without sacrificing the analysability. The consumption and production rates of a CSDF actor can vary, but have to be expressed as a periodic sequence, so that CSDF graphs are still fully deterministic, and keep the important SDF properties, enabling to calculate a periodic schedule, if it exists, and to determine the required sizes for the FIFO-buffers.

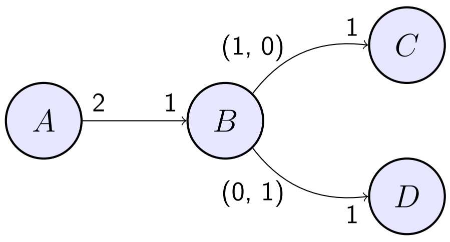


Figure 5.21.: Cyclo-Static Data Flow graph

Figure 5.21 gives an example for a CSDF graph. The actors A, C, and D are normal SDF actors, which are special cases of CSDF actors. The CSDF actor B consumes always one token on its input arc, but the production of the output tokens follows a periodic schedule. In the first firing of B, a token is produced only on the arc $B \rightarrow C$, in the next firing of B a token is instead produced only on the arc $B \rightarrow D$. Then this periodic schedule repeats.

For this example a possible schedule would be $\{A, B, C, B, D\}$, and the required buffer sizes would be $size(A \rightarrow B) = 2$, $size(B \rightarrow C) = 1$, and $size(B \rightarrow D) = 1$. Bilsen et al. (1996) also give a method how to derive a static schedule for CSDFs in their paper.

5.2.6. Scenario-Aware Data Flow

Scenario-Aware Data Flow (SADF) (Theelen et al., 2006) is another approach to further increase the expressiveness of SDF, while still providing sufficient analysability. The idea is to use *controlled dynamism* by enabling to model different modes of operation and to have a well-defined mechanism to switch between these modes. SADF provides two types of actors (called processes in Theelen et al. (2006)): *kernel* and *detector*. Figure 5.22 illustrates the kernel actor.

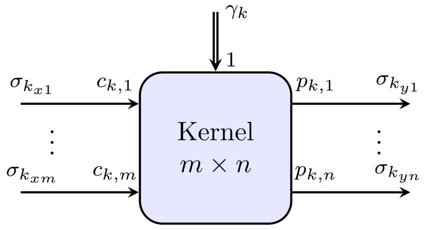


Figure 5.22.: A SADF *kernel* actor

A kernel is responsible for the processing of data and has m data input signals and n data output signals. In addition, a kernel can have one or more control inputs that define in which *scenario* or mode the kernel operates. The following discussion assumes that there is either zero or one control input signal. A kernel without control port will always run in the same mode and behaves like an SDF actor with fixed production and consumption rates. When a kernel executes, it starts by consuming the control token, which sets a scenario in form of functions that will be executed by SADF detectors, and the consumption and production rates of the input and output signals. A consumption or production rate can have the value 0. Once sufficient tokens are available, the output tokens are produced.

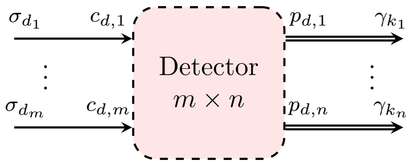


Figure 5.23.: A SADF *detector* actor

Figure 5.23 illustrates the detector actor. A detector is responsible to select the next scenario given its state and the input signals. The detector has m data input signals and n control output signals. During each firing the detector consumes a fixed number of tokens on each data input signal, and produces a possibly varying number of control output tokens on each arc, which are used to specify the scenarios for the kernel actors. If the next scenario is determined by a FSM, then the whole SADF graph is analysable, because it combines the analysability of a FSM, for the determination of the scenarios, with the analysability of SDF for the computation data flow graph.

Figure 5.24 gives an example for a SADF graph taken from (Theelen et al., 2006). The SADF graph contains four kernel actors, A, B, C, and D, and two detectors E and

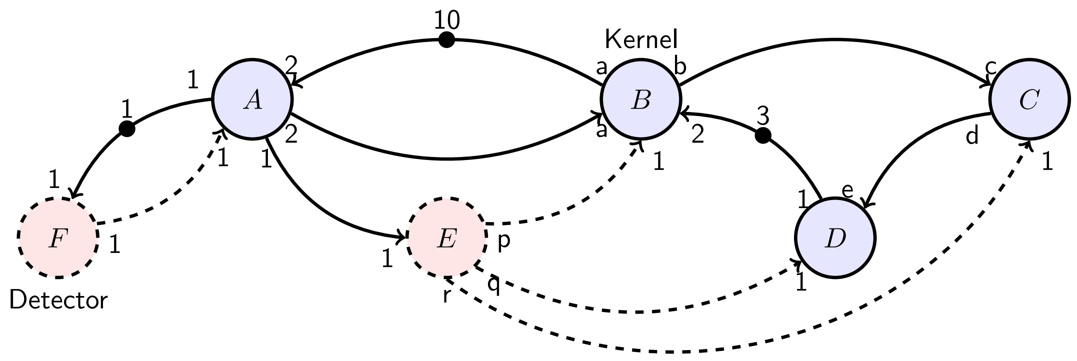


Figure 5.24.: A SADF graph has two types of nodes, *kernels* (blue, solid) responsible for the computation, and *detectors* which configure the kernel during each firing. The example is taken from Theelen et al. (2006).

F. Detector E controls the scenarios for the kernels B, C, and D and bases the next scenario on its own state and the data input token from A. During a scenario, detector E sets the consumption and production rates a, b, c, d, and e, and determines the scenario function via its outputs arc. During a firing of detector E, a variable number of tokens can be produced, expressed by the production rates p, q, and r. The detector F controls only kernel A, and only sets which scenario function shall be executed, since kernel A always consumes and produces the same number of data tokens.

5.2.7. Modelling Support

The Formal System Design (ForSyDe) methodology (ForSyDe; Sander et al., 2017) offers libraries for several MoCs, including support for the SDF, CSDF, and SADF MoCs. More information is given in Appendix G. The Ptolemy project (Ptolemy; Eker et al., 2003), which studies modelling, simulation, and design of concurrent, real-time, embedded systems, provides support for many MoC, including SDF, as part of their modelling framework Ptolemy II.

5.3. Synchronous Model of Computation

5.3.1. Synchronous Languages

The family of synchronous programming languages (Benveniste and Berry, 1991; Benveniste et al., 2003) addresses the design of *reactive systems*. A reactive system is a system that maintains permanent interaction with the environment (Benveniste and Berry, 1991). Many reactive systems are also *real-time* systems, which have to interact with the environment at the speed of the environment and have to satisfy externally defined timing constraints. For safety-critical reactive systems, properties like *safety*, *logical correctness* and *temporal correctness* are of key importance. Halbwachs et al. (1991) formulated three important features that are shared generally by reactive systems, which are summarised in the following.

- *Parallelism:* The design of these systems must take the parallel interaction between the environment and the system into account. Furthermore the implementation comprises often several parallel modules, which cooperate to achieve a given behaviour.

- *Time Constraints:* The system has to satisfy constraints imposed by the environment, for instance input frequencies and input-output response times. These timing constraints need to be specified, and have to be taken into account by the design process, so that they can be verified as an important part of system correctness.

- *Dependability:* Many reactive systems are highly critical systems. A design error can cause disasters in systems like a nuclear power plant or the flight control system of a commercial plane. Design methods and tools that support formal methods should be adopted for these systems.

The basic concepts for the synchronous languages have been developed in the 1980s, and there are many similarities with the design of synchronous hardware as pointed out by Benveniste and Berry (1991), where the communication between sub-components behaves fully synchronous, provided the clock frequency is not too high. Hardware design offers a very high predictability, so that the hardware synthesis tools can predict if the generated circuit will work at a certain clock frequency. The synchronous languages also have a similar goal and aim at increasing the predictability and correctness for software design in particular of safety-critical systems.

The synchronous languages have been created to address the problems of existing approaches for the design of reactive systems. Benveniste and Berry (1991) reviewed several techniques, which still are in use today and commented on their weaknesses. The most common approach, *connecting classical programs by making them communicate using OS primitives*, is heavily used in industry and is the base for RTOSs. Benveniste and Berry (1991) argue that although there exists a lot of experience using this techniques, it is very difficult to understand, debug and maintain the applications, since there is no single object to study, but rather a set of more or less loosely connected programs. This in turn leaves little room for clean automatic program behaviour analysis, and therefore no way of formally guaranteeing safety properties. Benveniste and Berry (1991) highlight numerous advantages of FSMs, also called *finite automata*. They are deterministic, efficient and can be automatically analysed by numerous available verification systems. But they also point out a serious problem, FSMs do not directly support hierarchical design and concurrency. This can easily become a problem when designing larger systems.

The synchronous languages take a different approach than the OS-based approaches. Halbwachs et al. (1991) discuss the main ideas of the synchronous languages and state that “such languages provide ‘idealised’ primitives allowing programmers to think of their programs as reacting *instantaneously* to external events”. There is a whole family of synchronous languages that are all based on this *perfect synchrony hypothesis*, which has also be formulated (Boussinot and De Simone, 1991) as “reactions are instantaneous so that activations and productions of output are synchronous, as if programs where executed on an infinitely fast machine”. The perfect synchrony hypothesis is visualised in Figure 5.25. This means that system reactions are *atomic* and do not interfere with each other. It is important to note, that the perfect synchrony hypothesis only states that reactions are instantaneous, it does not say when reactions happen, or how these reactions are spread in physical time. Thus, one can view the sequence of input events as a sequence of (abstract) clock events that triggers the execution of the system. But only the *total order* of events is defined, there is no mapping to physical time as illustrated in Figure 5.26.

Halbwachs et al. (1991) argue that “this feature [(the perfect synchrony hypothesis)], together with the limitation to deterministic constructs, results in deterministic programs from both functional and temporal point of view.”.

The family of synchronous languages consists of several different languages, among others Esterel (Boussinot and De Simone, 1991), Lustre (Halbwachs et al., 1991), and Signal (Le Guernic et al., 1991). This course will concentrate on Lustre to enable a deeper understanding of the properties and potential of these languages. The main

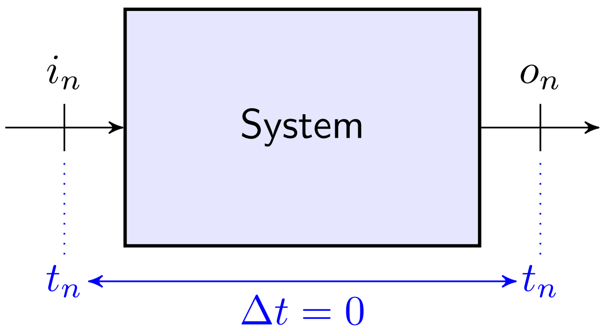


Figure 5.25.: Perfect synchrony hypothesis: System reactions are instantaneous, an input i_n and the corresponding output o_n happen at the same (abstract) time instance.

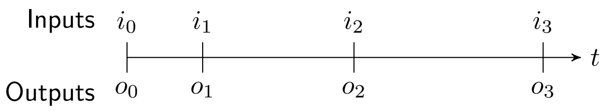


Figure 5.26.: Perfect synchrony hypothesis: Only the total order of events is defined, but there is no mapping to physical time

reference that is used for the following discussion is (Halbwachs et al., 1991), and several examples have been taken from that article.

5.3.2. The Synchronous Language Lustre

Overview of the Lustre Language

Lustre is a *synchronous* data flow language. Here the term synchronous is used in the sense of the perfect synchrony hypothesis (Section 5.3.1) and not in the sense of Synchronous Data Flow (SDF) (Section 5.2.2). Halbwachs et al. (1991) discuss also advantages of a data flow paradigm for a high level programming language. Data flow is "(a) a *functional model* with its subsequent mathematical cleanness, and particularly with no complex side effects. This makes it well adapted to formal verification and safe program transformation, since functional relations over data flows may be seen as time invariant properties; (b) a *parallel model*, where any sequencing and synchronisation constraints arise form data dependencies." In Lustre each variable and expression denotes a *flow*, which is a pair consisting of

- a possibly infinite sequence of *values* of a given type
- a *clock*, representing a sequence of times

A flow takes the n-th value of its sequence of values at the n-th time instance of its clock. There is a *basic clock*, which is the fastest clock, and all other clocks can be defined using the basic clock. A slower clock C_{slow} is defined by a faster clock C_{fast} using a Boolean-value flow and is the sequence of time instances, where the faster clock C_{fast} has the value `true`. Table 5.1 shows an example shows a flow C, whose clock is defined by the basic clock, and a slower flow C_1, whose clock is defined by C.

Table 5.1.: A slower clock can be defined by Boolean-valued flow

Basic cycles	1	2	3	4	5	6	7	8
Values of flow C	true	false	true	true	false	true	false	true
Cycles on C	1		2	3		4		5
Values of flow C_1	false		true	false		true		true
Cycles on C_1			1			2		3

The following source code shows an introductory Lustre program.

```
node combinational (x,y : int) returns (s:int);
let
  s = 2 * (x + y);
tel.
```

It defines a *node* `combinational`, which takes two input flows of integer values `x` and `y` and returns an integer flow `s`. Each output value of `s` is calculated using the statement `s = 2 * (x + y)`, which means that for each clock instance i of the flow, the value $s_i = 2(x_i + y_i)$ is produced.

The code can be compiled and executed using the Lustre V6 compiler (Jahier et al.), which is available at (Lustre V6 Tools). As a first step, the file needs to be compiled.

```
console> lv6 combinational.lus -node combinational -2c -cc
...
combinational_combinational_loop.c has been generated.
combinational_combinational.h has been generated.
combinational_combinational.c has been generated.
```

Then the code can be executed and the behaviour of the program can be simulated.

```
console> ./combinational.exec
#inputs "x":int "y":int
#outputs "s":int
#step 1
1 1
4
#step 2
1 0
```

```
2
#step 3
...
```

The Lustre program `combinational` did not contain any state, but only executes a function. All operators used in the program are *data operators*, which are arithmetic (e.g. +), Boolean (e.g. and), relational (e.g. <=), and conditional operators (e.g. if then else). These data flow operators only operate on operands sharing the same clock. An expression `z = x + y` means that for all n, $z_n = x_n + y_n$. Here, n denotes the cycle of the clock of the flows x, y, z, which all share the same clock.

Lustre also defines four *temporal operators*, which operate on

- `pre` ("previous") acts as memory
- `->` ("followed-by") is used to define initial values
- `when` "down-samples" an expression to a slower clock
- `current` "interpolates" an expression on a faster clock

The following program `sequence` defines a counter, which only counts upwards on each clock event. It uses the temporal operators pre and ->.

```
node sequence() returns (n:int);
let
  n = 0 -> pre(n) + 1;
tel;
```

In order to understand the meaning of the program, it is important to understand the following central statement, which uses the two temporal operators `pre` and `->`.

```
n = 0 -> pre(n) + 1;
```

Table 5.2 shows the general operation of both operators, where *nil* is an undefined value.

Table 5.2.: The temporal operators pre and ->

Basic cycles	1	2	3	4	5	6
`e`	e_1	e_2	e_3	e_4	e_5	e_6
`f`	f_1	f_2	f_3	f_4	f_5	f_6
`pre(e)`	*nil*	e_1	e_2	e_3	e_4	e_5
`e -> f`	e_1	f_2	f_3	f_4	f_5	f_6

Table 5.3 illustrates the usage of `pre` and `->` in the program `sequence`. The operator `pre` ("previous") acts as a memory, and "delays" each event of the incoming flow one

clock cycle. The operator `->` ("followed-by") operates on two flows, where the operation `e->f` replaces the first event of the flow `f` with the first event of the flow `e`, while it uses the event of the flow `f` in all other clock instances.

Table 5.3.: The temporal operators pre and fby

Basic cycles	1	2	3	4	5	6
0	0	0	0	0	0	0
1	1	1	1	1	1	1
`pre(n)`	nil	0	1	2	3	4
`pre(n) + 1`	nil	1	2	3	4	5
`0 -> pre(n) + 1`	0	1	2	3	4	5

Figure 5.27 illustrates the structure of the expression as a data flow graph. Here it is important to understand that `0` and `1` are constant flows, where each event in the corresponding flow has the value 0 or 1, respectively.

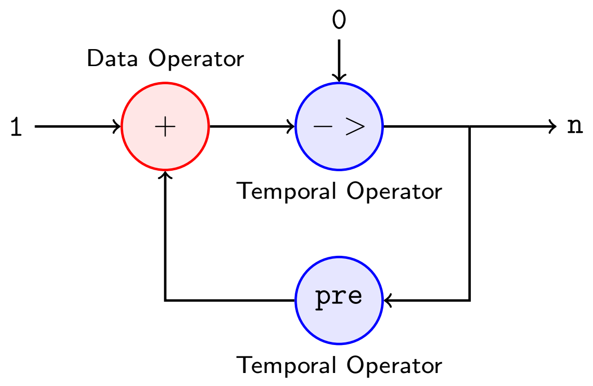


Figure 5.27.: Illustration of program sequence as data flow graph

The other two temporal operators are `when` and `current`. The operator `when` "down-samples" an expression according to a slower clock. The expression `y = e when b` creates a flow `y`, which is defined by the sequence of events of the flow `e`, when the clock `b` has the value *true*. The operator `current` "interpolates" or "up-samples" an expression according to a faster clock. Assuming a flow `e` is based on a Boolean-valued flow `b`, then the expression current `current y` defines a flow, where its own clock `c` is the same clock as `b`, and where the value of `y` at any time instance of the clock `c` is the value of `e` at the last time, when `b` was true. Otherwise, it is undefined and has the value `nil`. The operation of `when` and `current` is illustrated in Table 5.4.

The following program illustrates the use `when` and `current`.

Table 5.4.: The temporal operators when and current

Basic cycles	1	2	3	4	5	6	7
`b`	false	true	false	true	false	false	true
`e`	e_1	e_2	e_3	e_4	e_5	e_6	e_7
`y = e when b`		e_2		e_4			e_7
`z = current y`	*nil*	e_2	e_2	e_4	e_4	e_4	e_7

```
node current_when(clk: bool; set: int) returns (level: int);
let
   level = current(set when clk);
tel;
```

```
> ./current_when.exec
#inputs "clk":bool "set":int
#outputs "level":int
#step 1
0 5
0
#step 2
1 3
3
#step 3
0 0
3
#step 4
0 2
3
#step 5
1 2
2
```

Lustre also provides *assertions*. An assertion is a statement that should always be true, so the assertion

```
assert not (x and y);
```

represents that the Boolean variables `x` and `y` should never have the value *true* at the same time. The assertion

```
assert (true -> not (x and pre(x));
```

says that event x never appears twice in a row. Note, that this assertion requires an initial value *true*, because in assertions, clocks, and output sequences, the value `nil` is forbidden.

Assertions enable the compiler to do further optimisations and also play a very important role in program verification, with the objective to validate or even prove that the assertion is always satisfied.

The following Lustre program implements a flexible counter.

```
node counter(val_init, val_incr: int; reset: bool)
            returns (n:int);
let
  n = val_init
      -> if reset then
            val_init
         else
            pre(n) + val_incr;
tel;
```

A Lustre `node` can both be compiled and executed as a standalone program or be functionally instantiated in any expression. Thus the following program uses the node `counter` and creates a new program, which defines the sequence of modulo 5 numbers over the basic clock.

```
node modulo5() returns (y:int);
let
   y = counter(0,1,pre(y)=4);
tel;
```

Lustre Compiler

Static Verification The Lustre compiler conducts several static analysis techniques, which are enabled due to the clean and well-defined semantics, to check the correctness of the program at compile-time. This avoids overhead of dynamic checks at run-time.

Clock Calculus The Lustre *clock calculus* associates a clock with each expression of the program and checks that any operator applies to correctly clocked operands.

The following program illustrates the idea of the clock calculus.

```
b = true -> not pre(b);
y = x + (x when b);
```

The flow `b` is a Boolean-valued flow, with alternating values *true* and *false*. The data operator `+` in the flow `y` takes two flows `x` and `x when b`. Thus, the flow `y` needs both the n-th (flow `x`) and the $2n$-th value of `x` (flow `x when b`). This would cause an unbounded memory assuming that a reactive program runs infinite time. In addition, it is also is physically inconsistent, since it would mean that a value is computed out of an event at time n with another event at time $2n$.

In order to detect this and similar problems, the Lustre clock calculus consists of associating a clock with each expression of the program, and of checking that any operator applies to appropriately clock operands (Halbwachs et al., 1991):

- any primitive operator with more than one argument applies to operands sharing the same clock
- the clock of any operand of a `current` operator is not the basic clock of the node it belongs to
- the clocks of a node operand should obey the clock requirements stated in the node definition header

The compiler uses a more restricted notion of equality: two Boolean expressions define the same clock if and only if these can be unified by means of syntactical substitutions (Halbwachs et al., 1991). In the following example

```
x = a when (y > z);
y = b + c;

u = d when (b+c > z);
v = e when (z < y);
```

the compiler can detect that `x` and `u` share the same clocks (`y = b+c`), while `v` is considered as a different clock than `x` and `u`.

Absence of Cyclic Definitions The Lustre compiler rejects combinational cyclic definitions or zero-delay feedback loops, which means that there has to be `pre`-operator in each cyclic definition. Thus, the following program is rejected by the Lustre V6 compiler.

```
node srlatch(s:bool; r:bool) returns (q:bool);
var qbar : bool;
let
  q = s and qbar;
  qbar = r and q;
tel;
```

The compiler detects a combinational cycle and reports an error.

```
console> lv6 sr-latch.lus -node srlatch -2c -cc
Error: A combinational cycle been detected in file ...,
line 3, col 5 to 11, token 'srlatch': a0>a1>a0 where
   '>' means 'should be done after'
   a0: q = Lustre::and(s, qbar)
   a1: qbar = Lustre::and(r, q)
```

The problem of zero-delay feedback loop can be solved in different ways. Figure 5.28

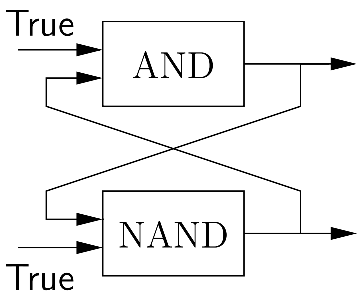


Figure 5.28.: A system with a zero-delay feedback loop

shows a system with zero-delay feedback loop that does not have a stable solution. If the output of the Boolean AND function is *true* then the output of the NAND function is *false*. But this means that the output of the AND function has to be *false*, which is in contradiction to the starting point of the analysis. Starting with the value *false* on the output of AND does not lead to a stable solution either, since this implies that the output of the NAND function is *true* and thus the output of the AND function must be *true*. Clearly there is no solution to this problem.

It is crucial for the design of safety-critical systems that feedback loops with no solution as in Figure 5.28 are detected and eliminated, since they result in an oscillator. Also feedback loops with multiple solutions imply a risk for safety-critical systems, since they lead to non-determinism. Non-determinism may be acceptable, if it is detected and the designer is aware of its implications, but may have serious consequences, if it stays undetected.

Since feedback loops in synchronous models are of such importance there are several approaches which address this problem (Benveniste et al., 2003).

Microstep In order to introduce an order between events that are produced and consumed in an event cycle, the concept of microsteps has been introduced into languages like VHDL. VHDL does not belong to the synchronous languages, but has a similar model of time. In order to solve the zero-delay feedback problem, VHDL distinguishes between two dimensions of time. The first one is given by a time unit, e.g. a picosecond, while the second is given by a number of delta-delays. A delta-delay is an infinitesimal small amount of time. Each operation takes zero time units, but one delta-delay. Delta-delays are used to order operations within the same time unit. While this approach partly solves the zero-delay feedback problem, it introduces another problem, since delta delays will never cause the advance of time measured in time units.

Thus, during an event cycle there may be an infinite amount of delta-delays. This would be the result, if Figure 5.28 would be implemented in VHDL, since each operation causes time to advance with one delta-delay. An advantage of the delta-delay is that simulation will reveal that the composite function oscillates. However, a VHDL simulation would not detect, if multiple solutions for a feedback loop exist, since the simulation semantics of VHDL would assign an initial value for their uninitialised values[1], and thus would only give one stable solution, concealing the non-determinism from the designer. Another serious drawback of the microstep concept is that it leads to a more complicated semantics, which aggravates the task of formal reasoning.

Forbid zero-delays The easiest way to cope with the zero-delay feedback problem is to forbid them. In case of Figure 5.28 this would mean the insertion of an extra delay function, e.g. after the upper AND function. Since a delay function has an initial value the systems will stabilise. Assuming an initial value of *true* as output of the delay function, Figure 5.28 will stabilise in the current event cycle with the values *false* for the output of the NAND function and *false* for the value of the AND function. A problem with this approach is that even stable systems are rejected, when they contain a zero delay feedback loop. This approach is adopted in the synchronous language Lustre (Halbwachs et al., 1991).

Unique fixed-point The idea of this approach is that a system is seen as a set of equations for which one solution in form of a fixed-point exists. There is a special value $\perp$ ("bottom") that allows it to give systems with no solution or many solutions a fixed-point solution. The advantage of this method is that the system can be regarded as a functional program, where formal analysis will show, if the system has a unique solution. Also systems that have a stable feedback loop are accepted, while the systems with no solutions or multiple solutions are rejected (the result will be the value $\perp$ as solution for the feedback loops). Naturally, the fixed-point approach demands a more sophisticated semantics, but the theory is well understood (Winskel, 1993). Esterel has adopted this approach and the constructive semantics of Esterel is described in (Berry, 1999).

Relation based This approach allows the specification of systems as relations. Thus a system specification may have zero solutions (the program is blocked and has no reaction), one solution or multiple solutions. Though an implementation of a system usually demands a unique solution, other solutions may be interesting for

[1] At program start, VHDL variables and signals take the leftmost value of their data type definitions. In case of the boolean data type this is the value `false`, since VHDL defines the data type `boolean` by means of `type boolean is (false, true)`.

high-level specifications. The relation-based approach has been employed in the synchronous language Signal (Le Guernic et al., 1991).

Code Generation

Halbwachs et al. (1991) describe a method to derive code for sequential programs in form of a Finite State Machine or finite automaton. The main idea is to select suitable state variables and then starting with the initial state to *simulate* the behaviour of the state variables and to generate the control structure in form of an FSM.

The method will be introduced with a running example using the following program `prog`.

```
node prog(x: bool) returns (y: bool);
var t: bool;
let
  t = false -> if not pre(y) and not pre(t) and x then
                 true
               else
                 if pre(y) and not pre(t) and x then
                   true
                 else
                   false;

  y = false -> if not pre(y) and pre(t) and not x then
                 true
               else
                 false;
tel.
```

State Variables The state variables are chosen from ideally Boolean expressions using `pre` or `current` operators, since these carry a state. Also, it is important to define an initial state by introducing an auxiliary variable `_init = true -> false`, which is only `true` in the initial state. Thus, in this example the following state variables are selected:

- the state variable `_init = true -> false` for the initial state;
- the state variables `pre(y)` and `pre(t)` for the states of the control FSM.

Control Synthesis The control synthesis starts at the initial state. For this and all other states, the state variables are assigned with the values for this state, and then the resulting code determines the possible values for the next state. Then each new state is treated in the same way, and finally the complete control FSM is generated.

- **State `S0` (Initial state).** The control structure is simulated by starting from the initial state `S0`. In this state the state variables have the following values:

- `_init  = true`
- `pre(y) = nil`
- `pre(t) = nil`

Then the values of the state variables are used for the simulation of the source code. The task is to determine the possible transitions of the state variables. Since the system is in the initial state, the values for `t` and `y` will be the first value in the sequence of the flows `y` and `t`, which in both cases is `false`. So, the system will move to a next state `S1`, in which `pre(y) = false` and `pre(t) = false`. Furthermore, in this new state `S1` the state variable `_init` has the `false`. Thus, these is only one transition from `S0`, which is to go to the next state `S1`. This is expressed as

```
S0 : goto S1; -- t = false; y = false;
```

- **State `S1`**. In the state `S1`, the state variables have the following values:
  - `_init  = false`
  - `pre(y) = false`
  - `pre(t) = false`

  Again, the state variables in the source code need to be replaced with their values in state `S1`. For this, the code after the followed by operator `->` has to be evaluated. For `t` this means the statement

```
t = if not pre(y) and not pre(t) and x then
                 true
               else
                 if pre(y) and not pre(t) and x then
                   true
                 else
                   false;
```

evaluates to

```
t = if true and true and x then
                true
              else
                if false and true and x then
                  true
                else
                  false;
```

and this finally is reduced to

```
t = if x then
       true
    else
       false
```

Using the same method yields that `y` always evaluates to `false`. Thus, there are two state transitions are possible, one to a new state `S2` and the other one to the same state `S1`. These transitions can be expressed as below.

```
S1: if x = true then
       goto S2; -- t= true; y = false;
    else
       goto S1; -- t= false; y = false;
```

- **State `S2`**. In the state `S1`, the state variables have the following values:
  - `_init  = false`
  - `pre(y) = false`
  - `pre(t) = true`

  Following the same method, which has been described for state `S1`, the following state transitions can be determined, where one transitions goes to an existing state `S1`, and the other one to a new state `S3`.

```
S2: if x = true then
       goto S1; -- t= false; y = false;
    else
       goto S3; -- t= false; y = true;
```

- **State `S3`**. In the state `S3`, the state variables have the following values:
  - `_init  = false`
  - `pre(y) = true`
  - `pre(t) = false`

  In this state, the following state transitions can be derived, where both transitions go to existing states, which means that the simulation is finished and the complete FSM is derived.

```
S3: if x = true then
       goto S2; -- t= true; y = false;
    else
       goto S1; -- t= false; y = false;
```

The resulting FSM is illustrated in Figure 5.29. The presented example uses only two

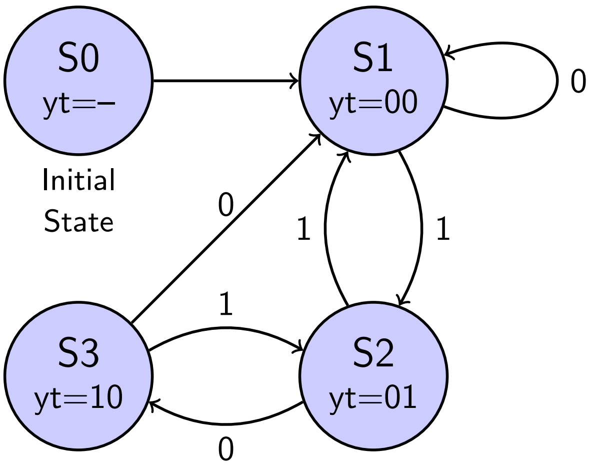


Figure 5.29.: Control structures in form of an FSM

state variables, y and t, and one input variable x and has been chosen for tutorial purposes. The article of Halbwachs et al. (1991) contains a more complex example with more variables.

Generation of Code for Target Language Once the state machine control structure been created, the Lustre program can be easily be converted into a target language. The following code is an example for the programming language C, including initial statements for reading and writing to the terminal input.

```
#include <stdio.h>

int main()
{
  unsigned char x; /* Integers are used as booleans here */
  unsigned char y;
  unsigned char state; /* State variable for S0-S3 */

  /* Initial State S0 */
  state = 0;
  while(1) {
    printf("Read input: ");
    scanf("%hhu", &x);
    switch(state) {
    case 0:
      y = 0;
      state = 1;
      break;
    case 1:
      y = 0;
      if (x) {state = 2;}
      else   {state = 1;}
      break;
    case 2:
      if (x) {state = 1; y = 0;}
      else   {state = 3; y = 1;}
      break;
    case 3:
      y = 0;
      if (x) {state = 2;}
      else   {state = 1;}
      break;
    default:
      printf("Error: State %hhu should not occur!", state);
    }
    printf("%hhu\n", y);
  }
  return 0;
}
```

5.4. Models of Computation

5.4.1. Tagged Signal Model

Lee and Sangiovanni-Vincentelli (1998) created the formalism of the *tagged signal model* to enable the "description, abstraction, and differentiation of Models of Computation (MoCs)." This framework addresses MoCs for concurrent systems, which can be modelled as concurrent process networks, where processes communicate via signals. The lecture is heavily based on the article of Lee and Sangiovanni-Vincentelli (1998), but that article contains much more information.

Figure 5.30 illustrates a concurrent process network, where processes communicate via signals. In general, a process in the framework is a set of possible *behaviours* between the signals connected to the process. These behaviours can be expressed by a *relation* of the signals connected to the process. However, in the context of this chapter, only *functional* processes, which distinguish between system inputs and system outputs, and

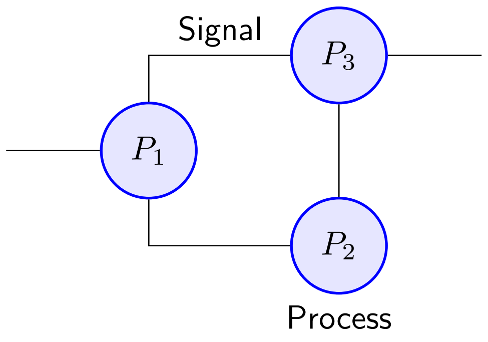


Figure 5.30.: A concurrent processes network where processes communicate via signals

where the process describes a single-valued mapping between the system inputs and system outputs, are discussed. A system is modelled as a set of processes, and is in itself a process.

Signals

A *signal* s is defined as a set of *events* $s = \{e_0, e_1, e_2, \dots\}$, where each event e_i has a tag $t_i \in \mathbf{T}$ and a value $v_i \in \mathbf{V}$, thus $e_i = (t_i, v_i)$. A *functional signal* s is a function from $\mathbf{T}$ to $\mathbf{V}$, where the function $s = \mathbf{T} \rightarrow \mathbf{V}$ can also only be partially defined, i.e. the function is only defined for a subset of $\mathbf{T}$. In order to be able to model the *absence* of a value, a special additional value $\perp$ (“bottom”) is used in a few MoCs to define the following set of values $\mathbf{V}_\perp = \mathbf{V} \cup \{\perp\}$. Depending on the set of tags $\mathbf{T}$, tags can be used to express (a) physical time, where $\mathbf{T} = \mathbb{R}$; (b) discrete time, where $\mathbf{T}$ is a totally ordered discrete set; or (c) precedences, where $\mathbf{T}$ is a partially ordered discrete set.

Processes

A process P is a set of possible behaviours. In general, this can be expressed by a relation between the signals connected to a process. In this lecture only functional processes with dedicated input and output signals will be discussed. Using this restriction, a process P is a function of input signals to output signals, i.e. $P = (s_{i,1}, \dots, s_{i,m}) \rightarrow (s_{o,1}, \dots, s_{o,n})$, as illustrated in Figure 5.31. A specific *behaviour* of a process is one possible valuation of system inputs and system outputs that satisfies the process.

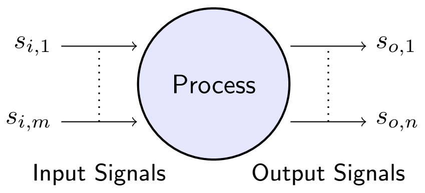


Figure 5.31.: A functional process is a set of behaviours, which can be specified by a function of input signals to output signals

Processes can be composed out of other processes. The process network of Figure 5.32 is composed be means of the processes P_1, P_2, and P_3, and is itself a process P.

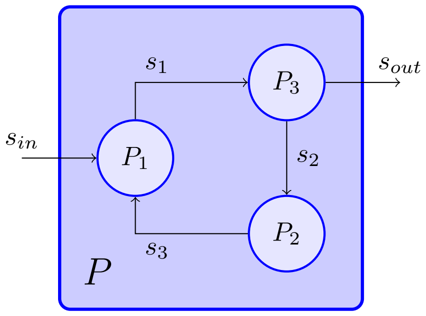


Figure 5.32.: Processes can be composed of other processes

This process composition can be formulated as a set of equations as in Equation (5.4).

$$
\begin{aligned}
&P(s_{in}) = s_{out} \\
&\text{where} \\
&\qquad \begin{aligned} s_1 &= P_1(s_{in}, s_3) \\ s_3 &= P_2(s_2) \\ (s_{out}, s_2) &= P_3(s_1) \end{aligned}
\end{aligned}
\tag{5.4}
$$

Defining Models of Computations

Tags and Order of Events Tags are used to reason about the order of events. The main purpose of the tags is to be able to express a possible order relation between the tags, and by this an order relation between two events. Lee and Sangiovanni-Vincentelli (1998) define that "an *ordering relation* on the set $\mathbf{T}$ is a reflexive, transitive and antisymmetric relation on members of the set". The ordering relation $\leq$ on the set of tags $\mathbf{T}$ is

- *reflexive*, that is $t \leq t$
- *transitive*, that if $t \leq t'$ and $t' \leq t''$, it implies that $t \leq t''$
- *antisymmetric*, that if $t \leq t'$ and $t' \leq t$, then $t = t'$

An *irreflexive* relation $<$ can then be defined as follows, if $t < t'$, then $t \leq t'$ and $t \neq t'$. An ordering relation defines then also the order of events, if $e_1 = (t_1, v_1)$, $e_2 = (t_2, v_2)$ and $t_1 < t_2$, only then $e_1 < e_2$. A set $\mathbf{T}$ with an ordering relationship is called an *ordered set*. If the ordering relationship is only partial (there exist two distinct tags $t, t' \in \mathbf{T}$, such that neither $t < t'$ or $t' < t$), then $\mathbf{T}$ is called a *partial ordered set*.

Timed Models of Computation A timed MoC has a tag system $\mathbf{T}$, where $\mathbf{T}$ is a *totally ordered set*, i.e. for any distinct two tags t and t' in $\mathbf{T}$, either $t < t'$ or $t' < t$. In timed systems, a tag is also called a *time stamp*. Two examples for timed MoCs are the synchronous MoC and the continuous time MoC.

Synchronous Model of Computation The term *synchronous* is defined by Lee and Sangiovanni-Vincentelli as follows in (Lee and Sangiovanni-Vincentelli, 1998):

1. two events are synchronous, if they have the same tag
2. two signals are synchronous, if all events in one signal are synchronous with an event in the other signal and vice versa
3. a process is synchronous, if every signal in any behaviour of the process is synchronous with every other signal in the behaviour

This means that Synchronous Data Flow (SDF) (Section 5.2.2) is *not synchronous*. The synchronous languages (Section 5.3.1) are synchronous, if the special value $\perp$ is used to denote the absence of an event. Then, all signals have "present" or "absent events" at each instance of the basic clock of the system.

Continuous Time Model of Computation Continuous time MoCs are used to model physical systems. It uses metric time, where the tag system often consists of the real numbers ($\mathbb{R}$). In the embedded system or cyber-physical system domain, the continuous MoC is mainly used to model the environment or the "physical" part of a cyber-physical system

Untimed Models of Computation In an *untimed* model of computation, the tags in the tag system are only *partially ordered*. There is a relation between certain tags, but not between all tags in the tag system. Among others data flow MoCs and Petri nets Peterson (1977) belong to the untimed MoCs.

Data Flow Models of Computation Data flow MoCs (Section 5.2) belong to the class of untimed MoCs. The reason for this is illustrated in Figure 5.33. The tags in each

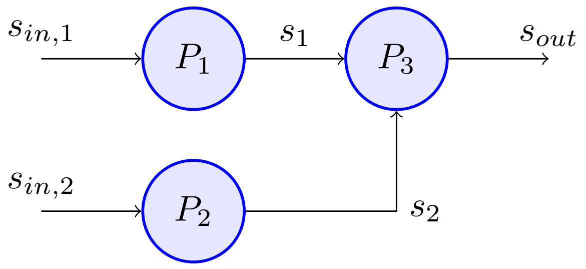


Figure 5.33.: An order between specific events in the signals $s_{in,1}$ and $s_{in,2}$ can in general not be determined in a data flow MoC.

signal are totally ordered, since the signals in a data flow MoC correspond to unbounded FIFOs. However, for the events in different signals a total order cannot always be given, for instance an event in $s_{in,1}$ and an event in $s_{in,2}$ can in general not be ordered. Thus, there is only a partial order in data flow models of computation.

Comparing Timed and Untimed Models of Computation One can compare timed and untimed MoCs with respect to their level of abstraction, or in other words the level of freedom for the design flow that they provide with respect to an implementation in the respective MoC.

A faithful implementation of a system belonging to

- a *timed* MoC, requires that the total order of events is preserved in the implementation;
- an *untimed* MoC only needs to preserve the partial order in the implementation.

Thus, more details with respect to the order of events are specified in a timed MoC than in an untimed MoC. This means in particular that the partial order within the untimed MoC gives in general more freedom for the implementation, in particular it allows to choose different schedules as long as the partial order is respected.

Benefits of the Tagged Signal Model

The tagged signal model provides a formal framework for MoCs, which allows it to reason and compare on an abstract level over communication and computation of parallel systems. It abstracts from specific programming languages, so that techniques applicable to a particular MoC can be used for different languages and modelling techniques compliant to that MoC. This means that for instance results for the synchronous MoC can not only be used for the family of synchronous languages (Section 5.3.1), but also for the implementation of the synchronous MoC in ForSyDe. Furthermore, new analysis techniques are constantly developed by the research community for different MoCs, which means that a sound basis in form of a suitable MoC enables the possible adaption of new future analysis techniques.

5.4.2. The ForSyDe Modelling Framework

The Formal System Design (ForSyDe) modelling framework is based on the tagged signal model (Section 5.4.1) and the theory of models of computation. Also, ForSyDe uses the *functional programming paradigm* for the modelling of processes and process networks, where each process is a pure mathematical function from input signals to output signals. This means that the implementation of a ForSyDe model in Haskell has a direct mapping to a mathematical functional description of the system. Appendix G provides additional information on ForSyDe, in particular by giving short tutorials for selected MoCs.

This section will explain how ForSyDe uses the tagged signal model as its underlying base, and how signals, processes, and systems are modelled in ForSyDe. The following discussion uses `ForSyDe-Shallow`, the main implementation of ForSyDe in Haskell (Appendix F). Other implementations of ForSyDe exist, in SystemC (Attarzadeh Niaki et al., 2012; Attarzadeh-Niaki and Sander, 2016) and also a newer implementation in Haskell, `ForSyDe-Atom` (Ungureanu and Sander, 2017; Ungureanu et al., 2019), which aims at further pushing orthogonalisation of concerns (Keutzer et al., 2000). The discussion does not aim to provide a tutorial on how to model systems in ForSyDe or to give a more detailed overview on the language. For these purposes, the book chapter on ForSyDe (Sander et al., 2017) and the ForSyDe web page (ForSyDe) should be consulted.

Systems are modelled in ForSyDe as heterogeneous hierarchical concurrent process networks, where *processes* communicate with each other via *signals* as illustrated in Figure ??. Processes belonging to different MoCs cannot directly communicate with

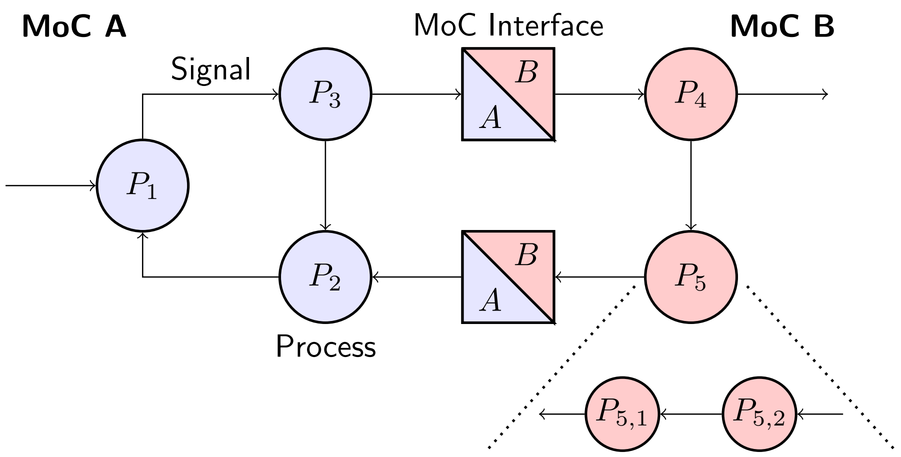


Figure 5.34.: The ForSyDe system model is a hierarchical concurrent process network, where processes belonging to different MoCs can communicate via MoC interfaces.

each other, instead special processes called, *MoC interfaces* are used to give a well-defined semantics for the conversion of signals between two MoCs.

The following discussion will use the synchronous MoC (Section 5.4.1) to illustrate the ForSyDe modelling framework and its relation to the tagged signal model.

Signals

ForSyDe defines signals as a *sequence* of *events*, modelled as a *list* data type in Haskell. The *tag* is given implicitly by the position in this list, and the value of the event is directly given by the data value at that position. **ForSyDe-Shallow** defines an own data type for **Signal** for a signal in order to distinguish a signal from a list.

A signal s can be created by converting a list to a **Signal**

```haskell
s = signal [1,2,3]
```

which defines a signal $s = \{e_0, e_1, e_2\}$, with the implicit tags t_0, t_1, t_2, and the values $v_0 = 1$, $v_1 = 2$, $v_3 = 3$. The tags are ordered within each signal, but the interpretation of the tags is defined by the specific MoC, which means that it is possible that there is no defined order relation between two different signals.

ForSyDe also allows to create data types that enable to model signals carrying *absent values*, as required in the synchronous MoC. A signal s_{sy} that contains absent values can be modelled by means of the data type constructors `Prst x` and `Abst`.

```
s_sy = signal [Prst 1, Abst, Prst 3]
```

This signal has the (present) value 1 at tag t_0, the absent value $\perp$ at tag t_1, and the (present) value 3 at tag t_2, thus $s_{sy} = \{1, \perp, 3\}$.

Processes

A process is a function of input signals to output signals. Processes in ForSyDe are created by *process constructors*. A process constructor is a higher-order function that takes arguments in form of functions and values and creates a process. A similar set of process constructors exist in each MoC, where the process constructors can be classified as

- *combinational process constructors*, which create processes that have no internal state;
- *sequential process constructors*, which create processes that have an internal state.

The basic combinational process constructor for the synchronous MoC is `mapSY`, which takes a function `f` as argument and applies it to all values in a signal. Thus, using the previously defined signal `s`,

```
console> mapSY (+1) s
{2,3,4}
```

`mapSY (+1)` creates a process, which increments all values in the signal s. In the same way, combinational processes with more than one input signal can be created. The process constructor `zipWithSY` is used to create processes with two input signals and can be used to define an adder:

```
adder = zipWithSY (+)
```

The names for the combinational process constructors `mapSY` and `zipWithSY` originate from functional programming, `ForSyDe-Shallow` also includes aliases in form of `combSY`, `comb2SY`, `comb3SY`, etc. in order to simplify for industrial designers with less background in functional programming.

The basic sequential process constructor in the synchronous MoC is `delaySY`, which delays an input value one event cycle by returning an initial value at tag t_0.

```
console> delaySY 0 s
{0,1,2,3}
```

Process Composition

Processes can be composed by formulating a process netlist as a set of equations. Figure 5.35 shows the process network of a counter that shall be modelled in ForSyDe-Shallow using the synchronous MoC. The counter has an input signal, which specifies if the counter should count upwards UP or should keep its present value DOWN. Whenever the counter reaches the value 4, the counter shall go to the value 0 at the next occurrence of UP. The counter has also an output, which should return the value TICK, when it is in state 0, otherwise it should return the absent value ⊥.

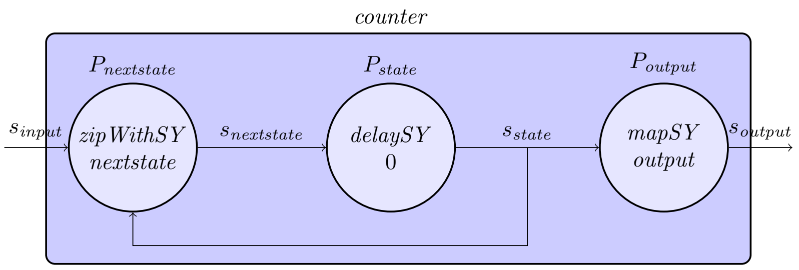


Figure 5.35.: Counter modelled in the synchronous MoC in ForSyDe

Listing 5.1 gives the complete ForSyDe-Shallow code of the counter. It also illustrates the different steps as part of a top-down process, which the designer has to execute in order to create the model.

1. The designer sketches the process network including the selection of the model of computation and the communication between the processes.

2. The designer selects suitable process constructors for all processes in the process network, alternatively expresses a high-level process by a composition of other processes. In Figure 5.35 the process constructors *zipWithSY*, *mapSY* are used to form combinational processes, while the process constructor *delaySY* is used to model a process with internal state.

3. The designer formulates the arguments to the process constructors, i.e. the leaf functions (*nextstate*, *output*) and other parameters (initial state for *delaySY* is 0), to form ForSyDe processes.

In particular, the code below `Step 1: Specification of process network` shows how a netlist is created by means of a set of equations.

Haskell is a strong typed language and although no data types are given, Haskell can infer the data type of the process `counter` to

Listing 5.1. ForSyDe model of the counter of Figure 5.35

```haskell
module Counter where

import ForSyDe.Shallow

data Direction = UP
               | HOLD deriving (Show)

data Clock = TICK deriving (Show)

-- Step 1: Specification of process network
counter s_input = s_output
   where s_output = p_output s_state
         s_state = p_state s_nextstate
         s_nextstate = p_nextstate s_state s_input

-- Step 2: Selection of process constructors
p_nextstate = zipWithSY nextstate
p_state = delaySY 0
p_output = mapSY output

-- Step 3: Specification of leaf functions
nextstate state HOLD = state
nextstate 4     UP   = 0
nextstate state UP   = state + 1

output 0 = Prst TICK
output _ = Abst
```

```haskell
counter :: Signal Direction -> Signal (AbstExt Clock)
```

This means that counter is a function that takes an input signal with data of type `Direction` and produces a signal with possibly absent events of type `Clock`. A simulation of the counter in ForSyDe shows the absent events as `'_'`-characters.

```
*Counter> counter (signal ([HOLD,UP,HOLD,UP,UP,UP,UP,HOLD,UP]))
{TICK,TICK,_,_,_,_,_,TICK,TICK,_}
```

6. Design Space Exploration and Code Generation

6.1. Design Space Exploration

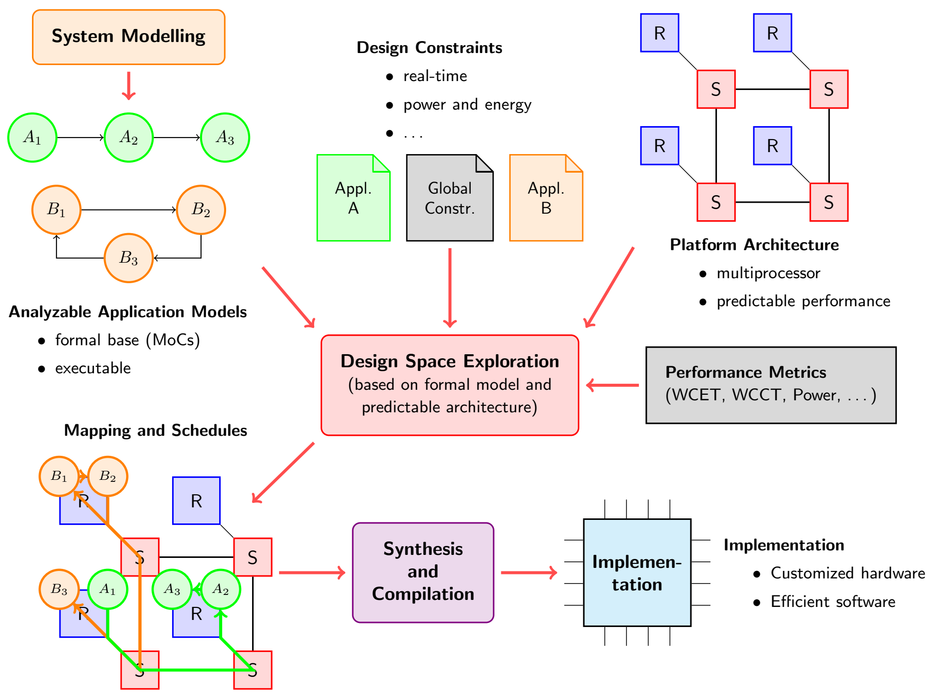


Figure 6.1.: Design space exploration is a central activity in the design flow

Design Space Exploration (DSE) is a very important and central activity in the system design process as illustrated in Figure 6.1. DSE aims to identify efficient and suitable implementations in an early stage of the design process, and the outcome of the DSE is used for the following synthesis and implementation phases. The DSE activity uses workload models for the individual application, a model of the platform architecture, application-specific and system-wide design constraints, and performance parameters as input and produces a mapping of application models to platform components together with a schedule and performance parameters as inputs.

6.1.1. Introduction

The recent article of Pimentel (2017) gives a very good overview of design space exploration and should be considered for further information on the topic.

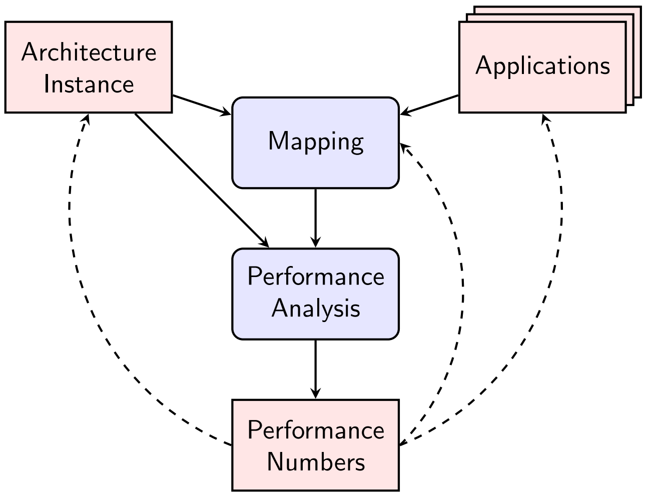


Figure 6.2.: Y-Chart(Kienhuis et al., 2002)

Figure 6.2 shows the Y-chart of Kienhuis et al. (2002)[1], which illustrates the DSE activity and its relation to *performance analysis*. This Y-chart has originally been developed to identify a suitable architecture for a class of applications, but can be easily adapted to other purposes. During DSE different *mappings* are investigated with the objective to find solutions that satisfy the design constraints. As drawn in Figure 6.2, DSE is an iterative process, where in each step a single mapping is analysed using a suitable *performance analysis* method. The result of the performance analysis method is a set of *performance numbers*, which enable to identify, if the design constraints have been met, and to compare different solutions. A typical DSE requires many iterations, since the design space is normally huge and cannot be fully explored. In each iteration, several parameters, such as mapping, schedule, or architecture, can be changed as indicated in the figure.

Thompson (2012) illustrates the relation between *performance analysis* and *DSE*. Performance analysis *evaluates a single design point*, where the criteria for good performance analysis methods are *accuracy*, *speed* and *effort* as illustrated in Figure 6.3.

DSE *searches the design space* to find suitable implementations, where criteria for good DSE methods are *confidence*, *convergence* and *effort* as indicated in Figure 6.4.

In general the design DSE for complex embedded systems is very large. Also, there are many different *design objectives* or *decision variables*, such as execution time, power consumption, design cost, reliability, or safety.

Assume two decision variables f_1 and f_2 as illustrated in Figure 6.5. In both cases a smaller value of f_1 and f_2 is better. Which solutions should be further analysed as implementation candidates? By closer looking at solution H it can be seen that solution

[1]There is also Gajski's Y-Chart, which describes a different aspect of system design.

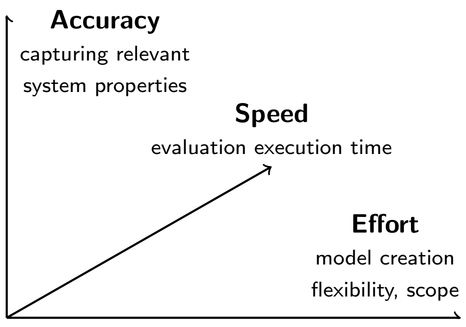

Accuracy
capturing relevant system properties
Speed
evaluation execution time
Effort
model creation flexibility, scope

Figure 6.3.: Criteria for performance analysis(Thompson, 2012)

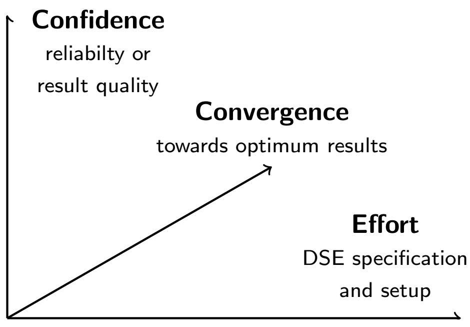


Figure 6.4.: Criteria for design space exploration(Thompson, 2012)

C is better that solution H in all design objectives. If a solution X is better than a solution Y in all design objectives, then solution X *dominates* Y. Figure 6.6 illustrates *domination* by comparing candidate solution H with other candidate solutions.

- The candidates in area I *dominate* H ($f_{1,I} < f_{1,H}$ and $f_{2,I} < f_{2,H}$)
- H *dominates* the candidate solutions in II ($f_{1,H} < f_{1,II}$ and $f_{2,H} < f_{2,II}$)
- H and candidate solutions in III are *incomparable* ($f_{1,III} < f_{1,I}$ and $f_{2,H} < f_{2,III}$)
- H and candidate solutions in IV are *incomparable* ($f_{1,H} < f_{1,IV}$ and $f_{2,IV} < f_{2,H}$)

A *Pareto-point* is a solution, which is not dominated by any other candidate solution. Thus, candidate solution H is not a *Pareto-point*, because it is dominated by candidate solution C. A *Pareto-curve* is built by the solutions that are *incomparable* with each other and are not dominated by any other candidate solution. Figure 6.7 shows the Pareto-curve given by the candidate solutions A to F. All other candidates (G to M) are *dominated* by at least one candidate on the Pareto-curve. An objective of the DSE is to find the Pareto-points that satisfy the design requirements, and to select a suitable solutions from these points.

The DSE can use different methods to conduct the performance analysis and can divided into simulation-based methods, analytical methods and hybrid methods. In simulation-based methods, each candidate solution is simulated and the performance is

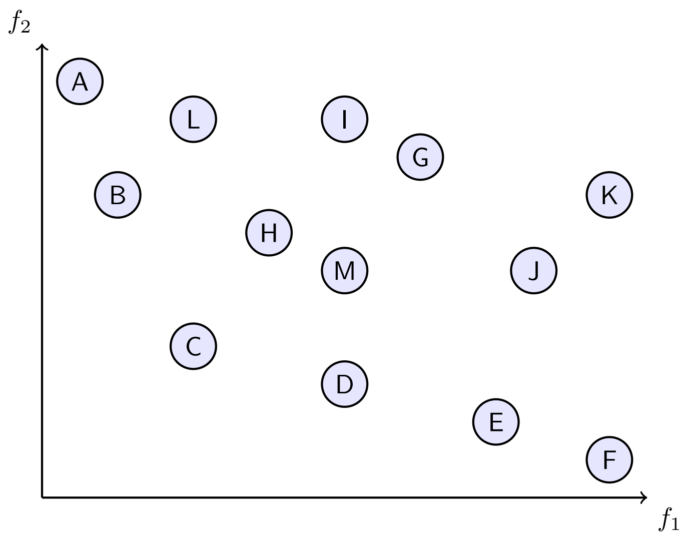


Figure 6.5.: Candidate solutions in a design problem with two design objectives f_1 and f_2

measured, while in analytical methods, an abstract model is analysed and performance figures are mathematically derived. Hybrid methods combine analytical and simulation-based methods. The following list gives advantages and disadvantages of these methods.

- *Simulation-based Methods*
  - 👍 Detailed models can be used
  - 👍 Specialised simulators can be used for different objectives
  - 👎 Simulation is very time consuming
  - 👎 Impossible to simulate all possible input and internal state combinations
- *Analytical Methods*
  - 👍 Abstract models enable to explore a large search space
  - 👍 Analytical models are formal, enabling powerful analysis methods
  - 👍 Possible to show optimality, if design space is completely analysed
  - 👎 Abstract models and often very pessimistic (e.g. assuming WCET)
  - 👎 Difficult to find good analytical models (e.g. power)
- *Hybrid Methods*
  - Larger search space can be explored than in simulation-based methods
  - Unclear, how exactly they can be combined?

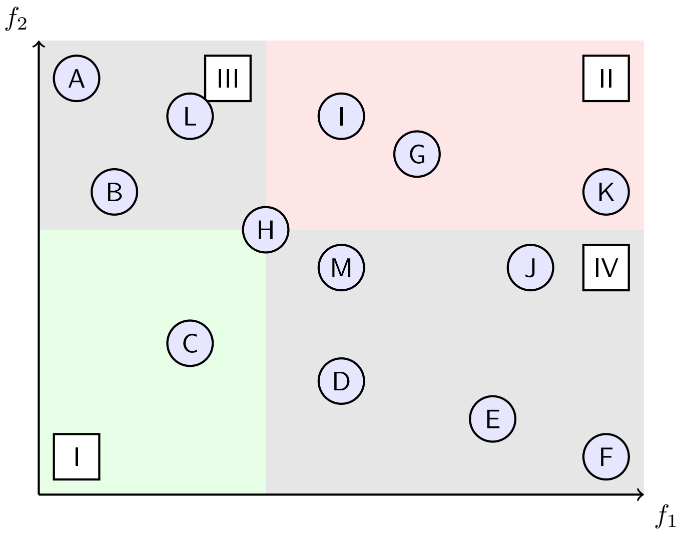


Figure 6.6.: Dominance with respect to candidate solutions H

6.1.2. Design Space Exploration in ForSyDe

ForSyDe has its own DSE tool Design Space Exploration for System Design (DeSyDe)[2], which is available on the ForSyDe web pageForSyDe. Section 6.1.2 will give an overview of the main concepts of the DeSyDe tool, while Section 6.1.2 will discuss an approach for joint-analytical DSE, which includes DeSyDe.

The design space exploration tool DeSyDe

DSE in ForSyDe exploits the formal character of ForSyDe in form of MoCs (Section 5.4. Figure 6.8 gives an overview of DeSyDe based on the first publication on the tool (Rosvall and Sander, 2014), where the main ideas have been described. The tool is still under development and the current state of DeSyDe is best described in (Rosvall and Sander, 2018) and (Rosvall et al., 2018). A new extended version of DeSyDe is under development.

DeSyDe is an analytical DSE tool and uses a *declarative approach* based on *constraint programming*. The benefits of the approach are grounded in constraint programming, which enables a) a separation of concerns, where the modelling of the problem is separated from the solving of the problem; b) a flexible, modular and extendable model; c) to potentially derive an optimal and complete solution; d) to conduct mapping, scheduling and performance analysis simultaneously; and e) to use the same model for a complete or heuristic search.

The applications in DeSyDe are modelled as SDF graphs. Supported platforms are a TDM shared memory multiprocessor (Rosvall and Sander, 2014) and a NoC (Rosvall et al., 2018). Performance parameters like WCET and Worst Case Communication Time (WCCT) or power consumption are given for different platforms. For each appli-

[2]DeSyDe: Design Space Exploration for System Design

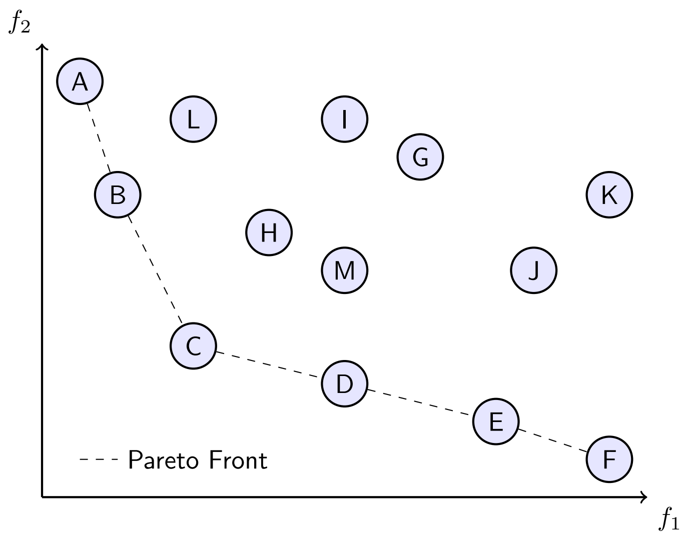


Figure 6.7.: Pareto-curve

cation individual design constraints are given together with design constraints for the whole system, e.g. power consumption. DeSyDe explores the design space and returns the following results: a) an *allocation* of platform components required for the design b) a *mapping* of application computation and communication components to platform resources c) a set of *schedules* for computation and communication d) *performance metrics* to show the efficiency of the solution for design objectives like throughput, latency, memory consumption, utilisation and minimal buffer sizes.

Internally, DeSyDe constructs a *mapping and scheduling aware graph* modelled as SDFG, where constraints in the constraint program formulation are used to model various aspects, like the mapping of actors to processors, schedules for ordering actor firings on each processor, the allocation of TDM-slots, the scheduling for sending and receiving of messages, or the WCCT. Figure 6.9 illustrates how the mapping and scheduling aware graph expresses communication as part of a SDFG.

More details on DeSyDe can be found in the articles (Rosvall and Sander, 2014, 2018; Rosvall et al., 2018), which also include experiments.

Joint analytical and simulation-based design space exploration

Figure 6.10 shows a hybrid DSE flow, where analytical DeSyDe DSE tool is used in the first phase of the design flow to analyse the design space for solutions, which satisfy the hard real-time design constraints. After this stage many candidate solutions are eliminated and the design space is significantly reduced. As a second step a simulation-based DSE is performed, which analyses typical use case scenarios. This approach builds on the idea that the analytical DSE is much faster and can filter out many non-compliant solutions, while simulation-based DSE takes significantly more time, but can analyse more detailed scenarios, and also enables to analyse different design objectives by using a suitable simulator. The main idea of this joint analytical and simulation-based

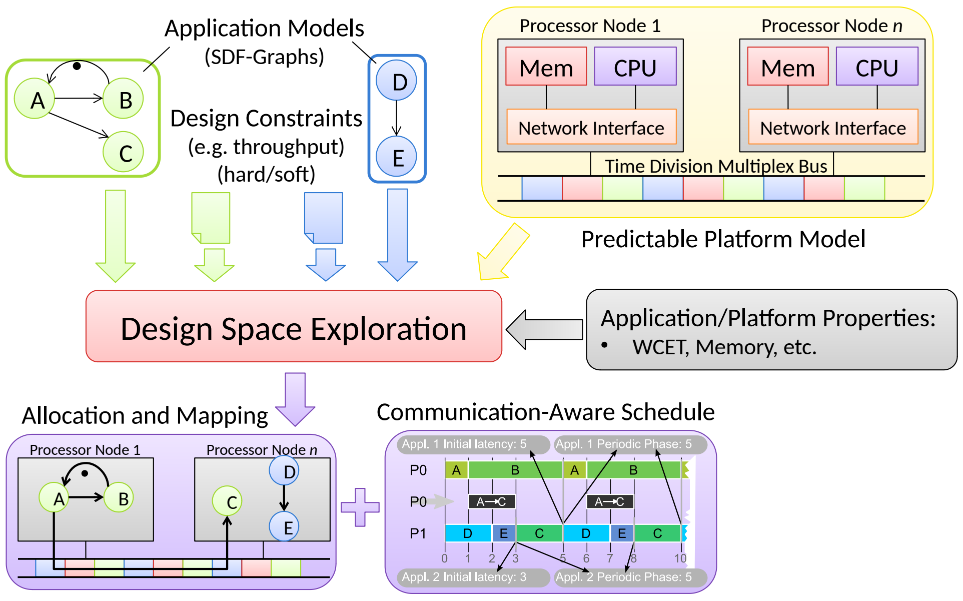

Application Models
(SDF-Graphs)
Design Constraints
(e.g. throughput)
(hard/soft)
Processor Node 1
Processor Node n
Mem
CPU
Network Interface
Time Division Multiplex Bus
Predictable Platform Model
Design Space Exploration
Application/Platform Properties:
• WCET, Memory, etc.
Allocation and Mapping
Communication-Aware Schedule
Appl. 1 Initial latency: 5
Appl. 1 Periodic Phase: 5
Appl. 2 Initial latency: 3
Appl. 2 Periodic Phase: 5

Figure 6.8.: Overview of DeSyDe (Rosvall and Sander, 2014)

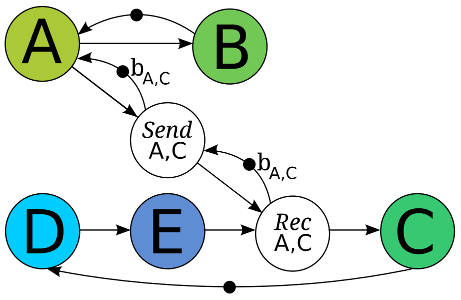


Figure 6.9.: Mapping and scheduling aware graph to express different candidate solutions during DSE in DeSyDe

design approach have initially formulated in (Herrera and Sander, 2013) and a concrete design flow using the tools DeSyDe for analytical DSE, the tool KiSTa for simulation-based performance analysis, and the tool MOST to guide the simulation-based DSE is described in (Herrera et al., 2015).

6.1.3. Further Reading

There exist several different approaches to DSE for embedded systems. The tutorial of Pimentel (2017) provides a very good starting point for further studies on this very importing topic.

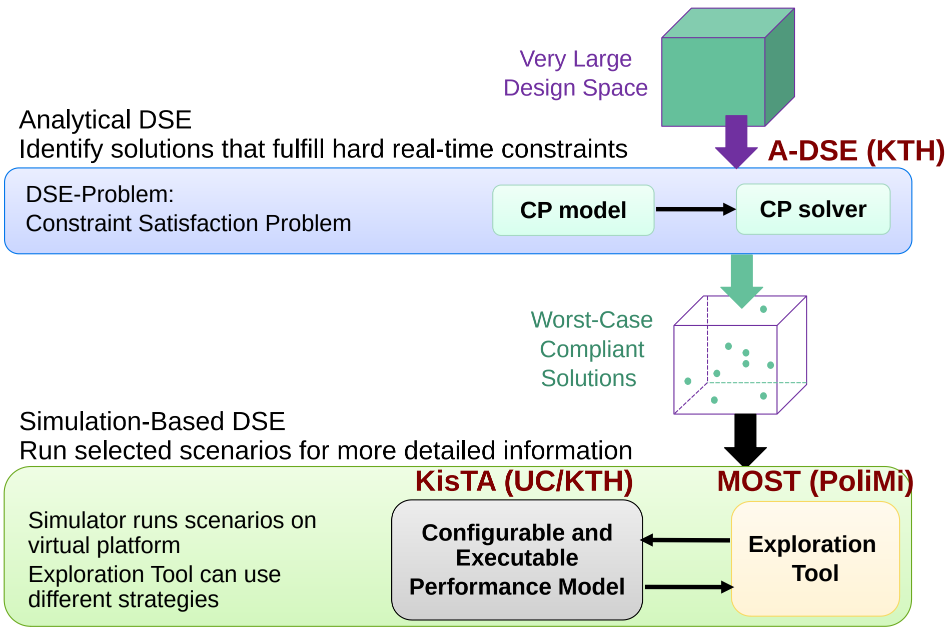

Very Large
Design Space
Analytical DSE
Identify solutions that fulfill hard real-time constraints
A-DSE (KTH)
DSE-Problem:
Constraint Satisfaction Problem
CP model
CP solver
Worst-Case
Compliant
Solutions
Simulation-Based DSE
Run selected scenarios for more detailed information
KisTA (UC/KTH)
MOST (PoliMi)
Simulator runs scenarios on virtual platform
Exploration Tool can use different strategies
Configurable and Executable Performance Model
Exploration Tool

Figure 6.10.: Joint analytical and simulation-based design space exploration (Herrera et al., 2015)

6.2. Synthesis from ForSyDe Models

This chapter describes how ForSyDe system models can be synthesised to implementations in hardware and software, so that the semantics of the original ForSyDe model is preserved in the hardware or software implementation. The chapter also gives the general principles and rules for the synthesis of a ForSyDe model that need to be followed for any target implementation. It is important to point out that the synthesis method described in this chapter is only possible due to the sound underlying theoretical foundation in form of well-defined MoCs, and the well-defined structure and semantics of ForSyDe models.

6.2.1. Semantics of ForSyDe Models

Section 5.4.2 has introduced the ForSyDe modelling framework, so only a short summary is given in this section. ForSyDe is based on a sound formal foundation in form of the theory of MoCs and the functional paradigm. This enables to express ForSyDe systems as a mathematical function, which takes input signals as arguments and returns output signals. A *signal* has been defined in Section 5.4.2 as a *sequence of events*, where each event has a *tag* and a *value*. Processes are introduced in Section 5.4.2functions of input signals to output signals, and are created by *process constructors*. These process constructors are higher-order functions that takes arguments in form of side-effect free functions and values and creates a process. Process constructors can be divided into combinational and sequential process constructors and a similar set of process constructors exists for each supported MoC. Process networks are created by process composition, where the netlist is formulated as a set of equations.

Thus all parts of the ForSyDe model are well-defined and have a clear mathematical semantics, which follows the functional paradigm.

6.2.2. General Principles and Rules for Synthesis

The ForSyDe model can be viewed as a component model, where the ForSyDe processes are the components, which communicate via signals. The *ports*, which exist in normal components, are implicitly given by the input and output signals as illustrated in Figure 6.11.

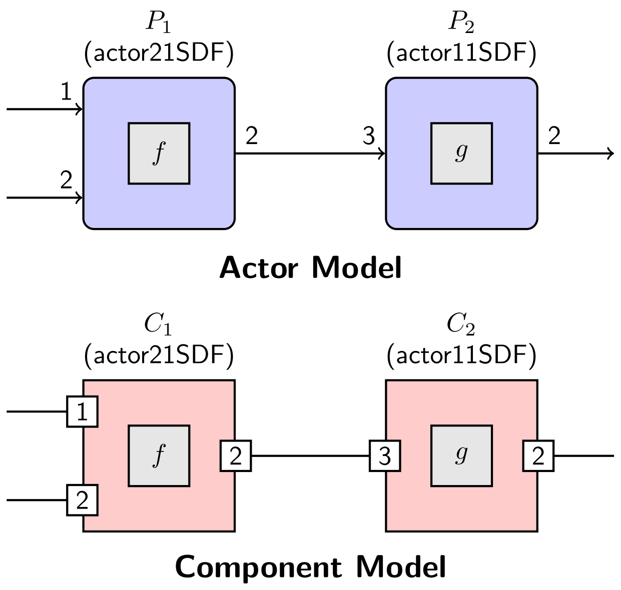


Figure 6.11.: A ForSyDe model is a component model, where the ports are implicitly given by the input and output signals.

An implementation of a ForSyDe model requires to realise the components and their interconnection in the target language, where both the functional and also the timely behaviour with respect to the underlying MoC, needs to be satisfy the semantics of the ForSyDe model.

Thus a correct implementation of a ForSyDe model in a particular target technology, which can be a target language or platform consisting of hardware and software, requires a

1. a correct implementation of the process network,
2. a correct implementation of all process constructors in the target technology, and
3. a correct translation of all function arguments and parameters

in the target technology.

The chapter will illustrate the general rules and principles by a) the synthesis of ForSyDe models belonging to the synchronous MoC to hardware in Section 6.2.3, and b) the synthesis of models belonging to the SDF MoC to software in Section 6.2.4.

6.2.3. Hardware Synthesis

In order to generate a hardware implementation, ForSyDe models have to be converted into synthesisable hardware models, expressed in hardware description languages like VHDL or Verilog. The ForSyDe tool `ForSyDe.Deep` synthesises ForSyDe models belonging to the synchronous MoC to synthesisable hardware in VHDL. Appendix E gives a short introduction to VHDL and includes an example of a synthesisable VHDL design of a counter with two seven segment encoders. The corresponding design is also used to give a short introduction to `ForSyDe.Deep` in Appendix G.5. The following discussion will focus on the more abstract concepts, which are implemented in the tool `ForSyDe.Deep`.

Process Synthesis

In order to synthesise a ForSyDe process to hardware, the process constructor and its arguments in form of function and parameters have to translated to the corresponding hardware description. A process constructor is a higher order function with a well-defined semantics, which takes functions and parameters as arguments and returns a process.

A process constructor can be realised in VHDL by a code template that takes contains place holders for its arguments in form of functions and arguments.

The idea is illustrated in the following by means of the process constructor *mapSY* and *delaySY*. Since *mapSY (f)* is a combinational process with one input and one output, it needs to be implemented as combinational process with one input and one output that executes the function *f*. This translation is illustrated in Figure 6.12 where $\mathsf{f_{HW}}$ denotes

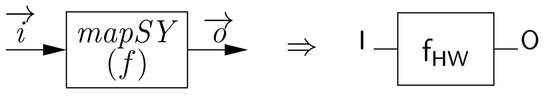


Figure 6.12.: Hardware implementation of *mapSY(f)*

the hardware implementation of *f*.

The combinational process constructor *mapSY* is formally defined as

$$\begin{array}{lcll}
\multicolumn{4}{l}{mapSY(f) = P_{SY} \in \mathbf{P_{SY}}} \\
\multicolumn{4}{l}{\text{where}} \\
\vec{o} & = & P_{SY}(\vec{i}) & \\
T(\vec{o}, j) & = & T(\vec{i}, j) & \forall j \in \mathbb{N}_0 \\
V(\vec{o}, j) & = & f(V(\vec{i}, j)) & \forall j \in \mathbb{N}_0
\end{array} \tag{6.1}$$

where P_{SY} is a specific process in the set of processes $\mathbf{P_{SY}}$ belonging to the synchronous MoC. $T(\vec{o}, j)$ denotes the tag of the signal $\vec{o}$ at position j. $V(\vec{o}, j)$ denotes the value of the signal $\vec{o}$ at position j.

The VHDL-template for $mapSY$ is given as

```
ENTITY mapSY_f IS
   PORT(
      i : IN type_mapSY_f_i;
      o : OUT type_mapSY_f_o
      );
END;

ARCHITECTURE Comb OF mapSY_f IS
BEGIN
   o <= f(i);
END;
```

where the function `f(i)` needs to be replaced by the $\mathsf{f_{HW}}$, the semantically equivalent implementation of *f*. Also the data types `type_mapSY_f_i` and `type_mapSY_f_o` need to be replaced by the data types of the input values and output values.

The basic process constructor for a sequential process is the process constructor *delaySY*$_k$, which delays a signal k event cycles. It takes an initial value s_0 as argument, which will be the value of the first k events of the output signal. The formal definition of *delaySY*$_k$ is given in Equation (6.2).

$$\begin{aligned}
&delaySY_k(s_0) = P_S \in \mathbf{P_S}\\
&\text{where}\\
&\vec{o} = P_S(\vec{i})\\
&T(\vec{o}, j) = T(\vec{i}, j) && \forall j \in \mathbb{N}_0\\
&V(\vec{o}, j) = \begin{cases} s_0 & \text{if } j < k \\ V(\vec{i}, j-k) & \text{otherwise} \end{cases} && \forall j \in \mathbb{N}_0
\end{aligned} \tag{6.2}$$

The process constructor *delaySY*$_k$ is realised with the hardware template illustrated in Figure 6.13, which uses k stages of registers. The VHDL-template for this structure is

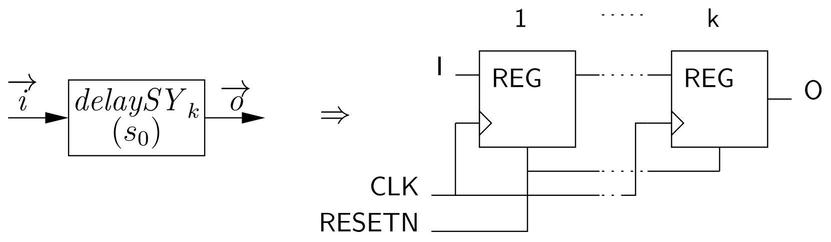


Figure 6.13.: Hardware implementation of *delaySY(f)*

shown in the following listing.

```
ENTITY delaySY_1 IS
   PORT
      (
         i : IN type_delaySY_1_i;
         o : OUT type_delaySY_1_o;
         resetn : IN std_logic;
         clk : IN std_logic
         );
END;

ARCHITECTURE Seq OF delaySY_1 IS
  SIGNAL s : type_delaySY_1_o := s0;
BEGIN
   PROCESS(clk, resetn)
   BEGIN
      IF resetn = '0' THEN
         s <= s0;
      ELSIF rising_edge(clk) THEN
         s <= i;
      END IF ;
   END process;

   o <= s;
END;
```

The synchronous MoC of ForSyDe provides many process constructors. However, the synthesis of the process constructor *mapSY* as the basic combinational process constructor and *delaySY* as the basic sequential process constructor illustrates the basic principles of the translation of ForSyDe processes to VHDL. More complex process constructors are created by means of the basic process constructors together with an accompanying VHDL-template. Figure 6.14 illustrates how processes based on the basic

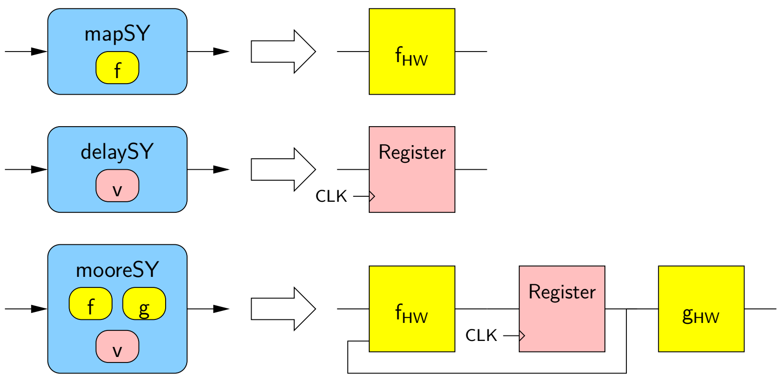


Figure 6.14.: Templates for hardware synthesis from ForSyDe processes

process constructors are implemented in hardware. It also includes the more complex process constructor *mooreSY* that is used to model Moore finite state machines, for which an efficient hardware implementation exists.

Process Network Synthesis

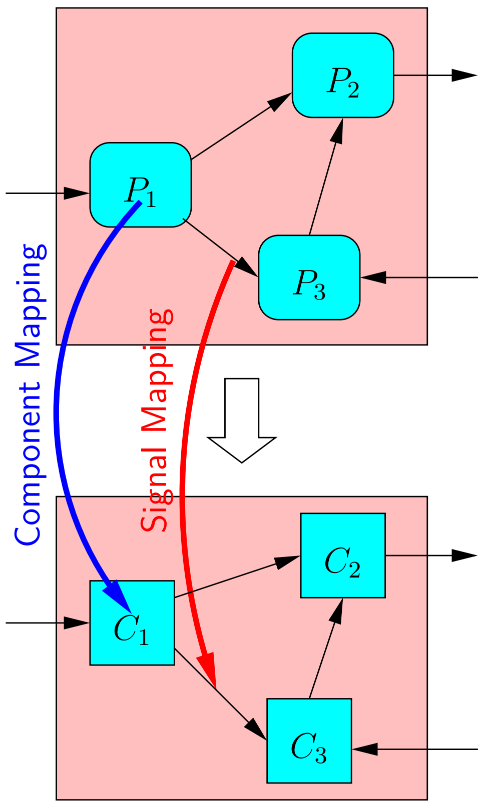


Figure 6.15.: ForSyDe process networks are synthesised to hardware by instantiating synthesised ForSyDe processes to VHDL components and ForSyDe signals to VHDL signals

Figure 6.15 illustrates the synthesis of ForSyDe process networks to hardware. ForSyDe processes are converted into VHDL using the methods described in Section 6.2.3 and instantiated as VHDL components. ForSyDe signals are converted to VHDL signals, and then the structure of the ForSyDe process network, expressed as a netlist in form of a set of equations, is converted into the corresponding VHDL netlist.

Hardware design with `ForSyDe.Deep`

In order to support hardware design, a special version of ForSyDe, `ForSyDe.Deep`, has been developed. Please consult Appendix G.5 for an introduction to `ForSyDe.Deep`, which also give a concrete example of a synthesisable ForSyDe model and its synthesis to hardware.

6.2.4. Software Synthesis

Also, the synthesis to software follows the general principles and rules for the synthesis of ForSyDe models described in Section 6.2.2. This section will discuss the synthesis from ForSyDe SDF models to software, in form of C-code. SDF has been discussed in Section 5.2.2 and the modelling of SDF in ForSyDe in Appendix G.2. The principles of the method are quite general and can be used outside ForSyDe as long as a) components communicate according to a well-defined semantics based on models of computation theory (here SDF), and b) communication and computation are separated as in ForSyDe.

The discussion will be conducted in a tutorial style, which uses the ForSyDe SDF model of Listing 6.1. The functions in the model are simplified to be able to focus on the principles of the synthesis process, but the model includes all important ingredients of an SDFG, such as different production and consumption rates and a feedback loop with initial tokens. A simulation of the model in Haskell gives the following output.

Listing 6.1. ForSyDe SDF model that serves as a running example for software synthesis

```haskell
module SDF_System_Model where

import ForSyDe.Shallow

-- System Netlist
system s_in = s_out where
  s_1 = p_1 s_in s_6_delayed
  (s_2, s_3) = p_2 s_1
  s_6 = p_3 s_3 s_5
  (s_out, s_4) = p_4 s_2
  s_5 = p_5 s_4
  s_6_delayed = delaySDF [0,0] s_6

-- Process Specification
p_1 = actor21SDF (2,1) 1 f_1
  where f_1 [x1,x2] [y] = [x1+x2+y]
p_2 = actor12SDF 1 (1,1) f_2
  where f_2 [x] = ([x],[x+1])
p_3 = actor21SDF (2,2) 2 f_3
  where f_3 [x1,x2] [y1,y2]
     = [x1+x2,y1+y2]
p_4 = actor12SDF 1 (3,1) f_4
  where f_4 [x] = ([x,x+1,x+2],[x])
p_5 = actor11SDF 1 1 f_5
  where f_5 [x] = [x+1]
```

```
*SDF_System_Model> let s = signal [1..10]
*SDF_System_Model> system s
{3,4,5,7,8,9,23,24,25,27,28,29,71,72,73}
```

The following different implementation alternatives will be discussed.

1. Synthesis to a bare-metal implementation on a single-processor
2. Synthesis to a RTOS implementation on a single-processor
3. Synthesis to a bare-metal implementation on a multi-processor

The ForSyDe model of Listing 6.1 can be represented as the SDFG in Figure 6.16, which

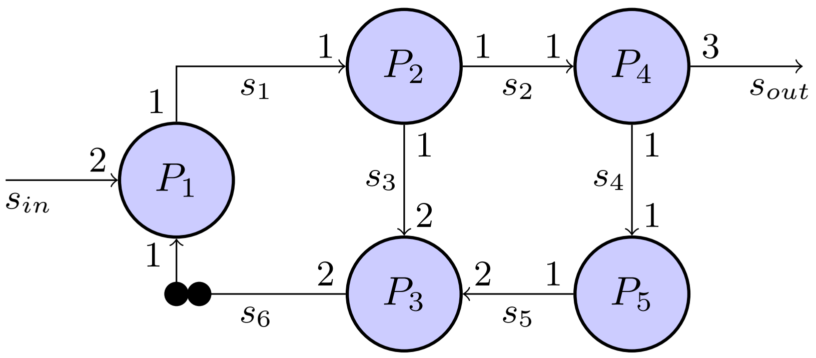


Figure 6.16.: SDFG-representation of ForSyDe model in Listing 6.1

can be analysed using the methods for SDF described in Section 5.2.2.

Following the principles and rules for synthesis from ForSyDe models, a synthesis of a SDF model to C-code requires that

1. the process network is translated into a static schedule with sufficient buffer space using the methods for SDF discussed in Section 5.2.2,
2. each process constructor is converted into a corresponding C-code template, and
3. each function is translated into the corresponding C-code.

Bare-metal implementation on single processor

Using the methods of Section 5.2.2, both a feasible schedule and the minimum buffer sizes can be derived.

- Feasible Schedule: $P_1P_2P_4P_5P_1P_2P_4P_5P_3$
- Buffer Sizes: $s_1 = 1, s_2 = 1, s_3 = 2, s_4 = 1, s_5 = 2, s_6 = 2$

Actors in SDF communicate via FIFO-buffers. This requires to implement a FIFO and the read and write access functions in the target language, such as the functions below.

- Function `void readToken(channel ch, token* x)` that reads a token `x` of type `token` from a channel `ch`
- Function `void writeToken(channel ch, token x)` that writes a token `x` of type `token` to channel `ch`
- Function to create data structure for internal FIFO-buffers:

  `channel createFIFO(token* buffer, size_t size)`

An efficient implementation of a FIFO is a circular buffer. In the following discussion it is assumed that an efficient structure for the FIFO with the corresponding access functions exists.

Then for each SDF arc, the corresponding data structure for the FIFO-buffers is created.

```
/* Create FIFO-Buffers for signals */
token* buffer_s_in  = malloc(2 * sizeof(token));
channel s_in = createFIFO(buffer_s_in, 2);
token* buffer_s_out  = malloc(3 * sizeof(token));
channel s_out = createFIFO(buffer_s_out, 3);
token* buffer_s_1  = malloc(1 * sizeof(token));
channel s_1 = createFIFO(buffer_s_1, 1);
token* buffer_s_2  = malloc(1 * sizeof(token));
...
token* buffer_s_6  = malloc(2 * sizeof(token));
channel s_6 = createFIFO(buffer_s_6, 2);
```

In addition, the initial tokens need to be created using the function `writeToken`

```
/* Put initial tokens in channel s_6 */
writeToken(s_6, 0);
writeToken(s_6, 0);
```

As a next step, the netlist needs to be created by using the schedule derived earlier. The schedule is implemented as an infinite while loop, where each actor is implemented using a powerful function template `actorXYSDF(..)`, where `X` corresponds to the number of inputs of the actor and `Y` to the number of outputs.

```
while(1) {
   for(i = 0; i < 2; i++) {
     /* Read input tokens */
```

```
      ...
      /* P_1 */
      actor21SDF(2, 1, 1, &s_in, &s_6, &s_1, f_1);
      /* P_2 */
      actor12SDF(1, 1, 1, &s_1, &s_2, &s_3, f_2);
      /* P_4 */
      actor12SDF(1, 3, 1, &s_2, &s_out, &s_4, f_4);
      /* Write output tokens */
      ...
      /* P_5 */
      actor11SDF(1, 1, &s_4, &s_5, f_5);
    }
    /* P_3 */
    actor21SDF(2, 2, 2, &s_3, &s_5, &s_6, f_3);
  }
```

For each process constructor *actorXYSDF*, a corresponding function template is available. An actor process constructor in ForSyDe is a higher-order function that takes several arguments. One of these arguments is the function that is executed during an actor firing. The `actorXYSDF` template is a implemented as higher-order function using a function pointer, which enables a direct conversion of a the ForSyDe actor process into an implementation in C.

The actor process constructor `actor12SDF` is used as example to explain the how these process constructors are implemented in C.

```
void actor12SDF(int consum, int prod1, int prod2,
                channel* ch_in, channel* ch_out1, channel* ch_out2,
                void (*f) (token*, token*, token*))
{
  token input[consum], output1[prod1], output2[prod2];
  int i;

  for(i = 0; i < consum; i++) {
         readToken(*ch_in, &input[i]);
  }
  f(input, output1, output2);
  for(i = 0; i< prod1; i++) {
         writeToken(*ch_out1, output1[i]);
  }
  for(i = 0; i< prod2; i++) {
         writeToken(*ch_out2, output2[i]);
  }
}
```

All actor process constructor functions have the following function arguments:

- number of consumed tokens on each input arc
- number of produced tokens on each input arc
- input channels
- output channels
- function that is executed on each input firing

The actor functions first consume the input tokens from each input channel, then apply their function on the consumed inputs, and finally produce the tokens on each output channel.

Finally, each function argument needs to be translated into the corresponding C-code. The requires that the number of produced input and output tokens of the actor is matched between by the function. To illustrate this, the original ForSyDe code for actor P_4 and the C-code for the implementation is given.

```
// Original ForSyDe model:
// p_4 = actor12SDF 1 (3 ,1) f_4
//   where f_4 [ x ] = ([ x , x +1 , x +2] ,[ x ])
void f_4(token* in, token* out1, token* out2) {
  out1[0] = in[0];
  out1[1] = in[0] + 1;
  out1[2] = in[0] + 2;
  out2[0] = in[0];
```

The complete code is available on the course web page. An execution of the code gives the following result, which match the simulation of the original ForSyDe model.

```
c-bare-metal> gcc c-bare-metal.c
c-bare-metal> ./a.out
Read two input tokens: 1 2
Output: 3 4 5
Read two input tokens: 3 4
Output: 7 8 9
Read two input tokens: 5 6
Output: 23 24 25
Read two input tokens: 7 8
Output: 27 28 29
Read two input tokens: 9 10
Output: 71 72 73
```

RTOS-implementation on single processor

A RTOS provides support for the implementation of communicating concurrent process networks in form of concurrent tasks, suitable communication mechanisms in form of message queues, and a dynamic scheduler. It is important that the message queues support blocking read and blocking write, so that the semantics of the SDF MoC can be correctly implemented by dynamic scheduling.

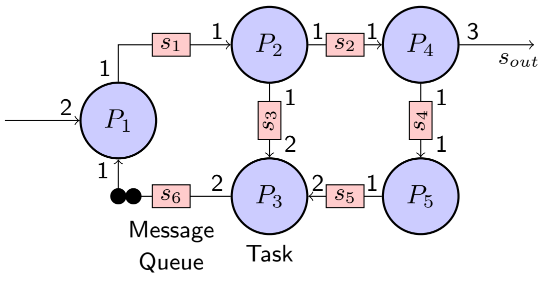


Figure 6.17.: Principal RTOS implementation

Figure 6.17 shows the principal implementation using a RTOS. Here,

- actors are implemented as tasks
- FIFO buffers are implemented as message queues with blocking read and write
- the system is dynamically scheduled by the scheduler, which blocks a task when it
  - try to write to a full message queue
  - try to read from an empty message queue

Bare-metal implementation on a multi-processor

Given a partitioning of the initial SDF model, so that different parts of the model are mapped to different processors, the steps to derive a multiprocessor implementation are described. Figure 6.18 gives a partitioning that could be the result of a previous

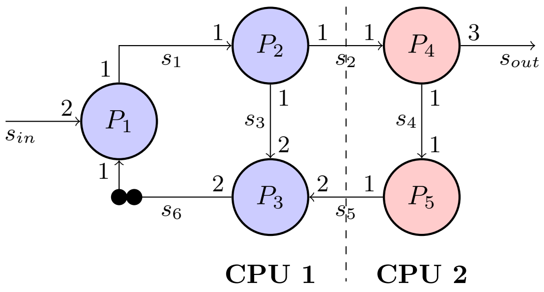


Figure 6.18.: Partitioned SDF model

Design Space Exploration (DSE), where it assumed that the two CPUs communicate via a shared memory. A possible schedule for actors on CPU 1 is P_1, P_2, P_1, P_2, P_3, and a possible schedule for actors on CPU 2 is P_4, P_5. Also, it can be seen that the communication between P_2 and P_4, and P_5 and P_3 has to use the shared memory.

The results for the bare-metal implementation on a single processor from Section 6.2.4 can even be used for the bare-metal implementation on a multi-processor, if the functions `readToken(channel ch, token* x)` and `writeToken(channel ch, token x)` are also implemented for the external channels s_2 and s_5. This a) requires support from the platform architecture and its low-level software, and b) external FIFO communication objects with blocking read and write need to be created and the corresponding access functions need to be defined In general, multi-processor platforms do not directly provide this support, so the platform must be chosen with care. If this prerequisite is fulfilled, the schedules for the partitioned process networks need to be implemented for each CPU as illustrated for the partitioned running example in Figure.

Schedule for Processor 1

```c
 while(1) {
   for(i = 0; i < 2; i++) {
     /* Read input tokens */
     ...
     /* P_1 */
     actor21SDF(2, 1, 1, &s_in,
                  &s_6,&s_1,f_1);
     /* P_2 */
     actor12SDF(1, 1, 1, &s_1,
                  &s_2, &s_3, f_2);
   }
   /* P_3 */
   actor21SDF(2, 2, 2, &s_3,
                &s_5, &s_6, f_3);
}
```

Schedule for Processor 2

```c
while(1) {
  /* P_4 */
  actor12SDF(1, 3, 1, &s_2,
               &s_out, &s_4, f_4);
  /* Write output tokens */
  ...
  /* P_5 */
  actor11SDF(1, 1, &s_4,
               &s_5, f_5);
}
```

Figure 6.19.: Schedule for each processor

A. Exercises

A.1. Embedded Computing Platform

A.1.1 (ID: 0002) In order to fulfill the design requirements, a system needs an average memory access time of 8 ns, the cache access time is 5 ns and a main memory access time is 70 ns.

a) Which cache hit rate is needed to achieve this performance?

b) Which cache hit rate would be needed for an average access time of 15 ns?

c) Which cache hit rate would be needed for an average access time of 6 ns?

A.1.2 (ID: 0003) A small *byte-addressable* embedded computer system, with a word length of 32 bits, has a main memory consisting of 4 KBytes. It also has a small data cache capable of holding eight 32-bit words, where each cache line contains only two words. When a given program is executed the processor reads data from the following sequence of hex addresses:

0x010, 0x1FC, 0x168, 0x008, 0x014, 0x1F8, 0x00C.

This pattern is repeated four times.

a) Consider a direct-mapped cache.

   i. How many bits are used for the tag, block, and offset fields for the representation of a memory address?

   ii. Show the contents of the cache at the end of the execution. Assume the cache is empty at the start of the program.

   iii. Compute the hit-rate for this example. Assume that the cache is initially empty.

b) Consider a 2-way set associative cache with the same cache size (eight words of 32 bits) and block size (2 words in a block).

   i. How many bits are used for the tag, set, and offset fields for the representation of a memory address?

   ii. Give an example for a sequence of read accesses, where the 2-way set-associative cache memory performs much better than the direct-mapped cache.

A.1.3 (ID: 0001) A peripheral circuit shall be connected to a CPU with an eight-bit address and data bus by memory-mapped I/O. The peripheral circuit has eight registers, which are controlled by the inputs $\mathtt{RS}_2$ to $\mathtt{RS}_0$. The circuit has also a chip select input `CS`, a read/write input $\mathtt{R}/\overline{\mathtt{W}}$ and eight data bits $\mathtt{D}_7$ to $\mathtt{D}_0$.

Show with a diagram how the peripheral circuit can be connected with the CPU, so that the eight registers are accessed only via the adresses 0x10 to 0x17.

A.1.4 (ID: 0004) An embedded processor system has three masters and four slaves. The processor system shall run the following abstract parallel program that only shows the order of slave accesses.

M_1	M_2	M_3
ReadS_1	ReadS_2	ReadS_4
ReadS_1	ReadS_3	ReadS_3
	ReadS_2	

To simplify we assume that a read transaction takes one cycle.

a) Assume that master and slaves are connected by a classical bus, i.e. only one master can access the bus during a bus cycle. How many cylces are needed to execute the abstract program?

b) Assume that the processor system uses weighted round-robin slave-side arbitration as implemented in the Avalon Switch Fabric. The weights are given below. A '-' means that the master and slave are not connected.

	S_1	S_2	S_3	S_4
M_1	1	-	-	-
M_2	-	1	2	-
M_3	-	-	1	1

i. Draw a schematic of the switch fabric described by the table.

ii. For all slaves, give the percentage of accesses that each master is guaranteed during a longer time period.

iii. Give the best case execution time for the abstract program.

iv. Give the worst case ecxecution time for the abstract program.

A.1.5 (ID: 0005) Given is the following program:

```c
#include <stdio.h>

int main(void) {
  short *sp;
  char* cp;
  int x[4], i, y;

  y = 0x01020304;
  for (i = 0; i < 4; i++) {
      x[i] = y;
      y = y + 0x10101010; }
  sp = (short*) x;
  printf("1) *sp   = %x\n", *sp);
  sp = sp + 3;
  cp = (char*) sp;
  printf("2) *sp   = %x\n", *sp);
  printf("3) cp[3] = %x\n", cp[3]);
  return 0;
}
```

The execution of the program gives

```
1) *sp   = 304
```

as first output. What will be the remaining output?

Assume the sizes for the following data types: `int`: 4 bytes, `short`: 2 bytes, `char`: 1 byte.

A.1.6 (ID: 0006) What will be the output of the following C-program?

```
#include <stdio.h>

char number[3] = {1,2,3};
char c = 'A';
char* text = "Hello";

int main() {
   printf("Print text: ");
   while(*text) {
      printf("%c", *text++);
   };
   printf("\n");
   int i = 0;
   printf("Numbers: ");
   for(i = 0; i <=3; i++) {
      printf("%d\n ", number[i]);
   }
   return 0;
}
```

A.1.7 (ID: 0007)

Given are the following functions A and B.

```c
/* Function A */
int i;
int w[512];
...
for(i = 1; i < 25; i++) {
    w[159]++;
    w[160]++;
}
y = w[159] + w[160];
...
```

```c
/* Function B */
int i;
int w[512];
...
for(i = 1; i < 25; i++) {
    w[145]++;
    w[274]++;
}
y = w[145] + w[274];
...
```

Will one of the two functions A and B execute faster than the other function (assuming no optimization),

a) if a direct-mapped cache with a size of 128 Bytes and a block size of 32 Bytes is used.

b) if a direct-mapped cache with a size of 128 Bytes and a block size of 4 Bytes is used.

Assume that the variable `i` is stored in a register and that `w[0]` is stored in the memory location `0xA80`, followed by the other array elements of `w[i]` in order of their index. The size of an integer is 4 bytes. Motivate!

A.1.8 (ID: 0009) A critical loop in an embedded system program reads data from the following sequence of memory locations (addresses in hexadecimal format):

0xCB3, 0xCE0, 0xA24, 0xCBE, 0x0FF.

This pattern is repeated 100 times.

The *byte-addressable* embedded computer system uses a word length of 32 bits and has a main memory consisting of 4 KByte. Also a data cache capable of holding in total sixteen 32-bit words, where each block contains four words, can be added to the computer system. The designer has the task to select the appropriate cache architecture with maximum hit rate at reasonable hardware cost for this embedded program and can choose from

- a direct-mapped cache,
- a 2-way set-associative cache,
- a 4-way set-associative cache.

For the following questions assume that the cache is empty at the start of the program.

a) Which cache is appropriate?

b) What hit rate can be achieved?

c) Show the contents of the cache at the end of the execution. Assume the cache is empty at the start of the program.

A.1.9 (ID: 0010) The following C-fragment shall be used to communicate with a memory-mapped I/O device with four registers. Each register has a size of eight bits. A `char`-variable has the size of eight bits.

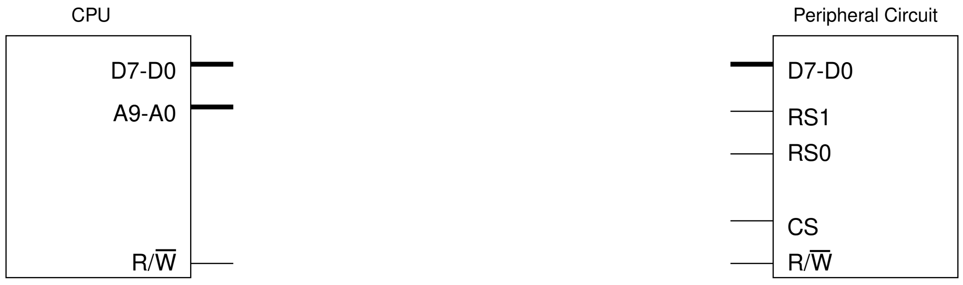
```
#define REGISTER_0 0x0B0
#define REGISTER_1 0x0B1
#define REGISTER_2 0x0B2
#define REGISTER_3 0x0B3

char readByte(char *location) { return *location; }

void writeByte(char *location, char newval) { *location = newval; }

int main() {
   ...
   /* Your code */
   ...
   return 0;
}
```

a) Complete the following diagram, so that the I/O-device can be accessed by memory-mapped I/O from a CPU with a ten-bit address and an eight-bit data bus, but only when the address is in the range from `0x0B0` to `0x0B3`. The four registers of the I/O-device are controlled by the inputs $\mathtt{RS}_1$ to $\mathtt{RS}_0$. The circuit has also a chip select input `CS`, a read/write input $\mathtt{R}/\overline{\mathtt{W}}$ and eight data bits $\mathtt{D}_7$ to $\mathtt{D}_0$.



b) Write C-code for the function `main` so that the program waits until bit 3 or bit 0 are set in register 2, and continues afterwards.

A.1.10 (ID: 0011) Given is the shared memory platform in Figure A.1, where the CPU 1 uses little endianness and CPU 2 uses big endianness.

Assume that the memory contains the following bytes starting from the address `0x1000` in ascending order: `0x24`, `0xA3`, `0x53`, `0x0F`, `0x01`, `0x34`, `0xCC`, `0x21`.

Give the content of the memory, after the following transactions.

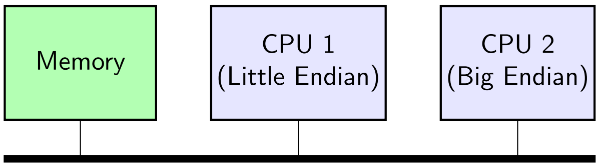


Figure A.1.: Shared Memory Architecture

- CPU 1 writes the four-byte word `0xFEDCBA98` into address `0x1004`
- CPU 2 reads a two-byte variable from address `0x1002`, adds then 256 to this variable, and writes the value as a four-byte word to address `0x1000`.

A.1.11 (ID: 0013) Given is the following C-program fragment.

```
#include <stdio.h>

int main()
{
  int x = 0x01234567;
  char* y;

  y = (char*) &x;

  /* Complete the Code */

  return 0;
}
```

a) Complete the code, so that the program prints `Little Endian`, if you have a little endian architecture, and `Big Endian`, if you have a big endian architecture.

b) Explain how this program detects the endianness.

A.1.12 (ID: 0014) Given is the shared memory processor platform shown in Figure A.2,

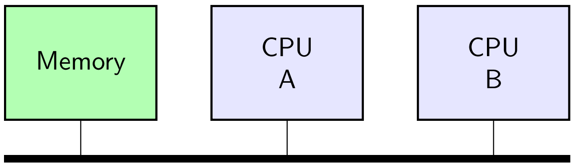


Figure A.2.: Shared Memory Multiprocessor

which uses a normal bus architecture. CPU A and CPU B have a shared address space and access the same variables, which are located in the shared memory.

Given are the following program fragments, which shall be executed on CPU A and CPU B, respectively.

CPU A

```
y = x;        \\ (A1)
```

CPU B

```
x = 7;        \\ (B1)
z = y + 1;    \\ (B2)
```

Assume that

- both processors are currently ready to execute the first line of their program;
- the shared variable `x` has the value 5
- the shared variable `y` has the value 10
- if a cache is used,
  - both caches are empty;
  - the variables `x`, `y`, `z` are mapped to different cache lines.

The parallel program can run in three different ways (A1-B1-B2, B1-A1-B2, B1-B2-A1).

a) For each of these execution schemes, give the values of the variables `x`, `y`, and `z` in the memory and the caches for the following memory system configurations, where
   i. both processors do not have a cache (fill in Table A.1)
   ii. both processors have a write-through direct-mapped cache (fill in Table A.2)
   iii. both processors have a write-back direct-mapped cache (fill in Table A.3).

b) Mark all the inconsistencies in Table A.2 and Table A.3, i.e. where the processors would read a different value of one of the shared variables `x`, `y`, and `z`.

c) Explain, why the three different memory configurations give different results. What exactly causes this "cache coherence" problem for both cache architectures?

Table A.1.: Values after execution of parallel program (no cache)

	A1-B1-B2	B1-A1-B2	B1-B2-A1
`x` (Memory)			
`y` (Memory)			
`z` (Memory)			

A.1.13 (ID: 0015) Given is the following program:

Table A.2.: Values after execution of parallel program (write-through cache)

	A1-B1-B2	B1-A1-B2	B1-B2-A1
x (Memory)			
y (Memory)			
z (Memory)			
x (CPU A)			
y (CPU A)			
x (CPU B)			
y (CPU B)			
z (CPU B)			

Table A.3.: Values after execution of parallel program (write-back cache)

	A1-B1-B2	B1-A1-B2	B1-B2-A1
x (Memory)			
y (Memory)			
z (Memory)			
x (CPU A)			
y (CPU A)			
x (CPU B)			
y (CPU B)			
z (CPU B)			

```
#include <stdio.h>

int main(void)
{
  char *cp;
  short *sp;
  int *ip;
  short x[6];

  int i, y;

  y = 0x0102;
  for (i = 0; i < 6; i++) {
      x[i] = y;
      y = y + 0x1010;
  }
  cp = (char*) x;
  printf("1) *cp   = %x\n", *cp);
  sp = x;
  printf("2) *sp   = %x\n", *sp);
  printf("3) cp[3] = %x\n", cp[3]);
  ip = (int*) x;
  ip = ip + 1;
  printf("A) *ip   = %x\n", *ip);
  printf("B) cp[6] = %x\n", cp[6]);
  sp = sp + 5;
  printf("C) *sp   = %x\n", *sp);
  *x = *cp + 2;
  printf("D) cp[1] = %x\n", cp[1]);
  return 0;
}
```

Execution of the code gives the following output:

```
1) *cp   = 2
2) *sp   = 102
3) cp[3] = 11
A) *ip   = ??
B) cp[6] = ??
C) *sp   = ??
D) cp[1] = ??
```

Fill in the output for A), B), C), D). Assume the following sizes for the variables, `int`: 4 bytes, `short`: 2 bytes, `char`: 1 byte.

A.1.14 (ID: 0029) (*Wolf: Computers as Components, Q5-16*) The loop appearing below is executed on a machine that has a 1K Word direct-mapped data cache with four words per cache line.

```
for (i = 0; i < 50; i++)
  for(j = 0; j < 4; j++)
    x[i][j] = a[i][j] * c[i];
```

Assume that all variables have the size of 4 Bytes.

a) How must `x` and `a` be placed relative to each other in memory to produce a conflict miss every time the inner loop's body is executed?

b) How must `x` and `a` be placed relative to each other in memory to produce a conflict miss one out of every four times the inner loop's body is executed?

c) How must `x` and `a` be placed relative to each other in memory to produce no conflict misses?

A.1.15 (ID: 0030) Given is the following function. Assume that

- after line 4, the arrays `u` and `w` are located in sequence in the memory, first all elements of `u` and then all elements of `w`
- `u[0]` is mapped to the first word of the first cache line
- one int value consists of 32 bits
- in every execution of the loop, `u` is accessed before `w`

```
int i;
int y = 0;
int u[60];
int w[60];
...
for(i = 0; i < 60; i++) {
    y += u[i] * w[i];
}
...
```

Assuming that the variables `i` and `y` are stored in registers and no further optimization,

a) Calculate the hit rate in the cache, if a direct-mapped cache with a size of 128 Bytes and a block size of 32 Bytes is used. Motivate!

b) What is needed to maximize the cache hit rate and how can this be implemented? Which cache rate can be achieved? Is it possible to reach 100%? Motivate!

A.2. Real-Time Systems

A.2.1 (ID: 0016) Given are the following sets of periodic tasks.

a) $\tau_1(6,2)$, $\tau_2(8,1)$, $\tau_3(4,1)$, $\tau_4(12,2)$.

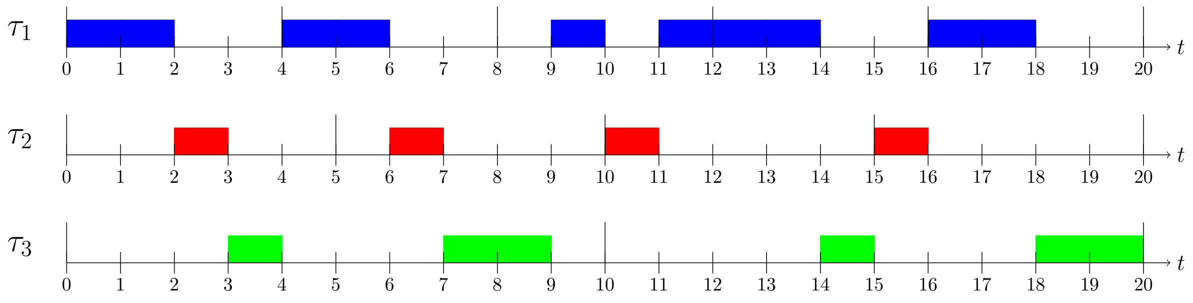
b) $\tau_1(4,1)$, $\tau_2(5,1)$, $\tau_3(10,2)$, $\tau_4(20,2)$.

c) $\tau_1(3,1)$, $\tau_2(12,1)$, $\tau_3(6,2)$, $\tau_4(24,3)$.

Determine for each set of tasks, whether a feasible schedule exists, if the tasks are scheduled according to the rate-monotonic algorithm. Motivate your answer!

The notation $\tau(T, C)$ defines a task τ, where T is the period and C is the execution time. For all tasks we assume that the relative deadline D is equal to the period T.

A.2.2 (ID: 0017) Can the following schedule of the task set $\tau_1(4,2)$, $\tau_2(5,1)$, $\tau_3(10,3)$ be produced by an earliest-deadline-first algorithm? Motivate your answer!



A.2.3 (ID: 0018) Given is the pseudo-code for the following three tasks τ_1, τ_2 and τ_3. Task τ_1 has the highest priority and task τ_3 has the lowest priority ($\pi(\tau_1) > \pi(\tau_2) > \pi(\tau_3)$). The tasks shall be executed using a preemptive real-time operating system, which uses preemptive multi-tasking.

```
void t1(...)
{
  while(1 < 2) {
    Send(queue,1);
    printf("1");
  }
}
```

```
void t2(...)
{
  while(1 < 2) {
    a=Receive(queue);
    printf("2");
    Wait(sem);
  }
}
```

```
void t3(...)
{
  while(1 < 2) {
    Signal(sem);
    printf("3");
}
```

Give the sequence of output values of the program. You can assume that all tasks are ready at program start and that

- the semaphore `sem` is initialized with the value 0.
- the message queue `queue` has space for two items and is initially empty. The operations `Receive` and `Send` implement blocking read and blocking write, respectively.

Motivate your answer!

A.2.4 (ID: 0019) Two tasks in a preemptive real-time operating system have both access to the shared variables `a` and `b`. Task 1 writes variable `a` and reads variable `b`. Task 2 writes variable `b` and reads variable `a`. Task 1 never writes variable b and task 2 never writes variable a. Given is the following code fragment.

Task 1	Task 2
...	...
a = f1(...);	b = g1(...);
// Synchronization Point	// Synchronization Point
f2(b);	g2(a);
...	...

The tasks shall now be synchronized so that

- task 1 does not execute `f2(b)` before the statement `b = g1(...)` has been executed in task 2
- task 2 does not execute `g2(a)` before the statement `a = f1(...)` has been executed in task 1

Assume the operating system supports semaphores and provides the functions `Wait(Sem)` and `Signal(Sem)` to access or release the semaphore `Sem`. Modify the code so that the tasks are synchronized as specified above. Draw also a diagram that illustrates how processes and semaphore(s) interact.

A.2.5 (ID: 0020) The following set of tasks shall be scheduled by the earliest deadline first algorithm: $\mathbf{\Gamma} = \{\tau_1(0, 5, 2, 7), \tau_2(2, 12, 4, 10), \tau_3(5, 20, C_x, 22)\}$. Give the largest integer number for C_x, so that all tasks still meet their deadline.

In this notation for a task $\tau_i(\Phi_i, T_i, C_i, D_i)$, Φ_i denotes the phase, T_i the period, C_i the computation time, and D_i the relative deadline.

A.2.6 (ID: 0021) The following algorithm using a *weak* binary semaphore S that was initialized with $S = (1, \emptyset)$ solves the critical section problem for two processes.

P_1	P_2
`loop forever`	`loop forever`
`  non-critical section (NCS)`	`  non-critical section (NCS)`
`  wait(S)`	`  wait(S)`
`  critical section (CS)`	`  critical section (CS)`
`  signal(S)`	`  signal(S)`

a) Show why it solves the problem for two processes on a single processor. Create a state diagram, where each states contains the following information:

   - State of process P_1 (`NCS` or `CS`)
   - State of process P_2 (`NCS` or `CS`)
   - State of semaphore $S = (S.V, S.L)$

   The state transistions are executed when a processor executes `wait(S)` or `signal(S)`.

b) Does it also solve the critical section problem for three or more processes? Give a clear and convincing explanation!

A.2.7 (ID: 0022) Create a concurrent program for a single processor with two processes P_1 and P_2 and two binary semaphores S_1 and S_2 that can potentially result in a deadlock. Give the program in pseudo-code. Explain in what situation a deadlock occurs.

A.2.8 (ID: 0023) Given is the following set of independent periodic tasks, which shall be scheduled rate-monotonically.

$$\tau_1 = (7{,}2);\ \tau_2 = (1{,}10{,}4{,}10);\ \tau_3 = (\phi{,}12{,}2{,}12);$$ [1]

a) Show with the graphical form of the time-demand analysis, if the set of independent periodic tasks is schedulable using the rate-monotonic algorithm. Give the maximum possible response times W_i for each task!

b) Determine the lowest possible value for ϕ, so that the critical instant for task τ_3 arises at the earliest possible time. Give the time for the first critical instance.

c) An addititional task $\tau_x = (50, 1)$ is added to the system. Use the *iterative form* of the time demand analysis to determine, if all tasks including τ_x still meet their deadlines? Give the response times for all tasks that meet their deadline.

A.2.9 (ID: 0034) The following code shall be executed in Ada 2005. Give all possible results for the variable `N`.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Concurrency is
   N : Integer := 1;
   task Task1; task Task2; task Task3;
   task body Task1 is
   begin
     N := N * 2;
   end Task1;
   task body Task2 is
   begin
      N := N + 3;
   end Task2;
   task body Task3 is
   begin
      N := 1;
   end Task3;
```

[1]The 4-tuple (ϕ_i, T_i, C_i, D_i) gives the phase ϕ_i, period T_i, execution time C_i and relative deadline D_i. In case of a 2-tuple (T_i, C_i) ϕ_i is 0 and D_i is T_i.

```
begin
   delay 1.0; -- Wait 1 second
   Put_Line("N = " & Integer'Image ( N ));
end Concurrency;
```

A.2.10 (ID: 0035) A client-server application shall be implemented in Ada 2005. The server shall monitor the number of requests from the clients. For this it uses an internal variable `N`. For each request `N` is incremented and the current number of requests is returned to the client.

a) Which Ada interprocess communication mechanism would you use?

b) Give the Ada code for the client.

c) Give the Ada code for the server.

A.3. Hardware/Software Co-Design

A.3.1 (ID: 0024) A DSP application implemented on a uniprocessor uses a multiply accumulate subroutine 15% of the total execution time. What is the speed-up, if a new version of the processor would include a multiply accumulate instruction that only needs 10% of the execution time of the subroutine?

A.3.2 (ID: 0025) A system consists of a processor and an accelerator. They are connected by a shared bus with a memory. Each of them has many registers to store values. Communication between them is done via the shared memory.

Each `Load` and `Store` takes 2 time units. Register accesses inside the processor or accelerator do not consume time. An add operation can only be performed on the processor and takes 1 time unit. A multiplication takes 8 time units on the processor, but only 2 time units on the accelerator.

The following algorithm shall be implemented:

```
X = A * C + B * C + D
Y = (A + B + C) * D
```

At the start of the program the variables `A`, `B`, `C` and `D` are stored in memory and the results `X` and `Y` shall bestored in memory at the end of the algorithm.

Both processor and accelerator have `LOAD` and `STORE` instructions and transfer a memory value to a register and vice versa. The arithmetic instructions `ADD` and `MUL` take two register operands and store the result in a register. The instruction `ADD` is not available on the accelerator.

a) How much time will the algorithm take, if is executed on the processor?

b) How much time will the same algorithm take, if it is executed on processor and accelerator? How large is the speed-up?

In both cases show the schedule of the program.

A.3.3 (ID: 0026) A system consists of a processor and an accelerator. They are connected by a shared bus with a memory. Each of them has many registers to store values. Communication between them is done via the shared memory.

Each `Load` and `Store` takes 2 time units. Register accesses inside the processor or accelerator do not consume time. An add operation can only be performed on the processor and takes 1 time unit. A multiplication takes 8 time units on the processor, but only 1 time unit on the accelerator.

The following algorithm shall be implemented:

```
X = A * B + C * D
```

At the start of the program the variables `A`, `B`, `C` and `D` are stored in memory and the result `X` shall be stored in memory at the end of the algorithm.

Both processor and accelerator have `LOAD` and `STORE` instructions to transfer a memory value to a register and vice versa. The arithmetic instructions `ADD` and `MUL` take two register operands and store the result in a register. The instruction `ADD` is not available on the accelerator.

a) How much time will the algorithm take, if is executed on the processor?

b) How fast can the same algorithm be executed, if it is executed on processor and accelerator? How large is the speed-up?

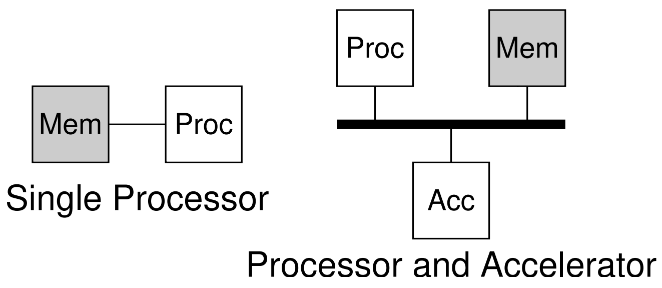
In both cases show the schedule of the program.

A.3.4 (ID: 0036) The function

$$y(x) = f_4(f_3(f_1(x)), f_2(f_1(x)))$$

which is composed by several sub-functions shall be implemented in such a way that the total execution time is not larger than 800 time units. Before execution, x is stored in memory and also the result y has to be stored in memory after execution. Each memory access (`Load` and `Store`) takes 50 time units.

The function can either be implemented on a single processor or on a combination of one processor and one hardware accelerator. Processor and accelerator are connected by a shared bus with a memory. Communication between them is done via the shared memory.



The following table shows the execution time for different processors depending on the processor speed S, where the speed S can have one of the following values: 2, 3, 4.

Function	Processor	Accelerator
f_1	$600/S$	not available
f_2	$600/S$	75
f_3	$600/S$	75
f_4	$600/S$	not available

The cost of a processor is $50 * S^2$ and the cost of the accelerator is 200.

Each processor and accelerator has a sufficient number of registers. A register access is assumed to take zero time.

Using the available architectures, *give the solution that fulfils the design constraints at the lowest possible cost.* You do not have to calculate all possible alternatives, but it should be very clear from your answer, why you consider other alternatives worse. To illustrate your calculations *provide the corresponding schedules.*

A.3.5 (ID: 0028) Given are the following $Q3$-Numbers: $A = (0.110)_2$ and $B = (1.100)_2$.

a) Add the numbers A and B

b) Multiply the numbers A and B

In both cases the result shall be stored in $Q3$-format.

A.3.6 (ID: 0027) Given is the following floating-point format $b_5b_4b_3b_2b_1b_0$, where the sign-bit is given by b_5, the mantissa by $1.b_2b_1b_0$, and the exponent by b_4b_3. The value is calculated as $-1^{b_5} * 1.b_2b_1b_0 * 2^{b_4b_3-1}$.

a) Is it possible to represent the numbers 0.25, 0.8125, -1.375, 4.25 and 7.5 in this format? Motivate!

b) Add the following numbers: $b_5b_4b_3b_2b_1b_0 = 001111$ and $b_5b_4b_3b_2b_1b_0 = 010010$. What would be needed to avoid loss of information?

c) Multiply the numbers of the previous subtask.

A.3.7 (ID: 0033) Given is the following code fragment.

```
r = a + b;   /* 1 */
u = e + f;   /* 2 */
v = 2 * d;   /* 3 */
s = a * c;   /* 4 */
t = g + s;   /* 5 */
w = r + s;   /* 6 */
```

Assume a clever compiler that can optimze the usage of registers on a single-thread processor with load-store architecture. How can the code be restructured so that a minimal number of registers is used?

Assume that the processor provides only the following type of operations *op* R_1, R_2, R_3, where $R_1 \neq R_2$ and $R_1 \neq R_3$ and $R_2 \neq R_3$. Assume further that after the result has been written to a variable that the result is written to memory and can be removed from the register.

a) Draw the data-flow-graph for the code fragment.

b) Give the restructured code and the register-life-time-graph to illustrate your solution.

c) How many registers are needed?

A.3.8 (ID: 0031) Given is the following code fragment.

```
#define MAX 10

int a[MAX], b[MAX], c[MAX], x[MAX], y[MAX];
int i, j, r, s;
...
int f(int a, int b) {
  int z;
  z = 2 * a - b;
  return z;
}
int g(int a, int b, int c) {
  int z;
  z = a * c - c* b;
  return z;
}
...
for(i = 0; i <= MAX - 1; i++){
  x[i] = f(a[i], b[i]);
}
s = 2 * r;
for(j = 0; j <= MAX - 1; j++){
  y[j] = s * g(a[j], b[j], c[j]);
}
...
```

Try to find optimizations that result in a shorter execution time, while not increasing code size. Explain each optimization and give the optimized code as C-code.

A.3.9 (ID: 0032) Given is the following code fragment.

```
y = 0;
if (mode == 1) {
   for (i = 0; i < 5; i++) {
      y += a[i] * b[i];
   }
}
```

a) Draw the control and data flow graph (CDFG) for this code fragment.

b) Determine the best case and worst case execution time, if the multiplication instruction takes three time unit, an addition takes one time unit and all other statements and branches take one time unit. The processor can only execute one instruction at each time instance. The processor does not have a cache.

c) How much is the execution time reduced, if the processor has a special instruction *Multiply-Accumulate* that calculates `R1 = R1 + R2 * R3` in a single cycle (1 time unit)?

B. Solutions to Selected Exercises

B.1. Embedded Computing Platform

Solution to Exercise A.1-1 (ID: 0002)

- The formula for the average memory access time is:
  $t_{AVG} = h * t_C + (1 - h) * t_M$
- Solve for h:
  $t_{AVG} = h * (t_C - t_M) + t_M$
  $h = \frac{t_M - t_{AVG}}{t_M - t_C}$
- Insert values given for t_M, t_C and t_{AVG}.
  (a) $h \approx 0.954$
  (b) $h \approx 0.846$
  (c) $h \approx 0.985$

Solution to Exercise A.1-2 (ID: 0003)

(a) Direct-mapped cache:

i. A word has a size of 32 bits or 4B (bytes). The size of the memory is 4 kB, which means that 4096 bytes or 1024 words are stored in the memory. The cache has a size of 8 words, organised in 4 cache lines, where each cache line contains 2 words. Given a block size of 2 words, the memory contains 512 different blocks.

Each memory block has a direct mapping to a cache line, which is given by its memory address. Analysing the 12-bit memory address from the least significand bit, the last 2 bits (b_1 and b_0) point out the position of a byte in a word. Since each cache line holds two words 1 bit (b_2) points out the position of the word in a cache. The cache has four lines. Thus, 2 bits (b_4 and b_3) are used to determine the cache line which the memory block is mapped to. All other bits (b_{11} to b_5) are used for the tag, which is used to determine, which block resides in a cache line.

7	2	1	2
tag	line	word	byte

The cache is organized as follows:

Cache Line			
0 (00_2)	Tag	Word 0 (B0 B1 B2 B3)	Word 1 (B0 B1 B2 B3)
1 (01_2)	Tag	Word 0 (B0 B1 B2 B3)	Word 1 (B0 B1 B2 B3)
2 (10_2)	Tag	Word 0 (B0 B1 B2 B3)	Word 1 (B0 B1 B2 B3)
3 (11_2)	Tag	Word 0 (B0 B1 B2 B3)	Word 1 (B0 B1 B2 B3)

ii. All memory addresses are given with tag, cache line, and word and byte offsets in the following table:

addr.	tag.block.word.byte	tag (hex)	block
0x010	0000000.10.0.00	0x00	[0x010–0x017]
0x1FC	0001111.11.1.00	0x0F	[0x1F8–0x1FF]
0x168	0001011.01.0.00	0x0B	[0x168–0x16F]
0x008	0000000.01.0.00	0x00	[0x008–0x00F]
0x014	0000000.10.1.00	0x00	[0x010–0x017]
0x1F8	0001111.11.0.00	0x0F	[0x1F8–0x1FF]
0x00C	0000000.01.1.00	0x00	[0x008–0x00F]

Each address references a byte-addressable memory location, but in case of a miss, an entire block (2 words or 8 bytes) will be loaded into the respective cache line. The column "block" shows the memory addresses for the entire block that is loaded into the cache.

The content of the cache after reading the sequence of memory addresses four times, where [ADR] denotes the contents of the memory location at the address range ADR, is given in the following table:

Cache Line	Tag	Word 0	Word 1
0 (00_2)	empty	empty 0	empty
1 (01_2)	0x00	[0x008–0x00B]	[0x00C–0x00F]
2 (10_2)	0x00	[0x010–0x013]	[0x014–0x017]
3 (11_2)	0x0F	[0x1F8–0x1FB]	[0x1FC–0x1FF]

The cache controller performs the following steps when accessing the cache.

A. Identify the line in the cache which corresponds to the block part of the address (i.e. for 0x010 this is line 2, because bits b_4 and b_3 are 10_2).

B. Compare the tag of the requested memory address to the tag stored in the cache line.

C. If the tags are equal than there is a cache hit and the requested address is located in the cache.

D. If the tags are different, then there is a cache miss. The new block is loaded into the cache line to replace the old block and the tag is updated. Note that not only the requested byte is loaded into the cache, but all the whole memory block consisting of two words.

iii. To calculate the hitrate, we have to count the number of hits during the read accesses. In the first run, there a three misses due to an empty cache (also called “cold miss”) and one “conflict miss” (different tag). In the second, third and fourth run, there are only two (conflict) misses.

addr.	1st run hit/miss	2nd-4th run hit/miss
0x010	(cold) miss	hit
0x1FC	(cold) miss	hit
0x168	(cold) miss	miss
0x008	(cold) miss	miss
0x014	hit	hit
0x1F8	hit	hit
0x00C	hit	hit

Then the hit rate can be calulated as $h = \frac{num_{hits}}{num_{total}} = \frac{1}{4} \cdot \frac{3}{7} + \frac{3}{4} \cdot \frac{5}{7} \approx 0.643$

(b) 2-way set-associative cache

i. The 2-way set-associative cache has the same size of 8 words, but it instead of 4 cache lines, it contains 2 sets, where in each set contains 2 “ways” for the data to go. Each way contains a tag and a block of two words.

Each memory block has a direct mapping to a set, which is given by its memory address. Analysing the 12-bit memory address from the least significand bit, the last 2 bits (b_1 and b_0) point out the position of a byte in a word. Since each cache line holds two words 1 bit (b_2) points out the position of the word in a cache. The cache has two sets. Thus, only 1 bits (b_3) is used to determine the cache set which the memory block is mapped to. All other bits (b_{11} to b_4) are used for the tag, which is used to determine, which block resides in a cache line.

8	1	1	2
tag	set	word	byte

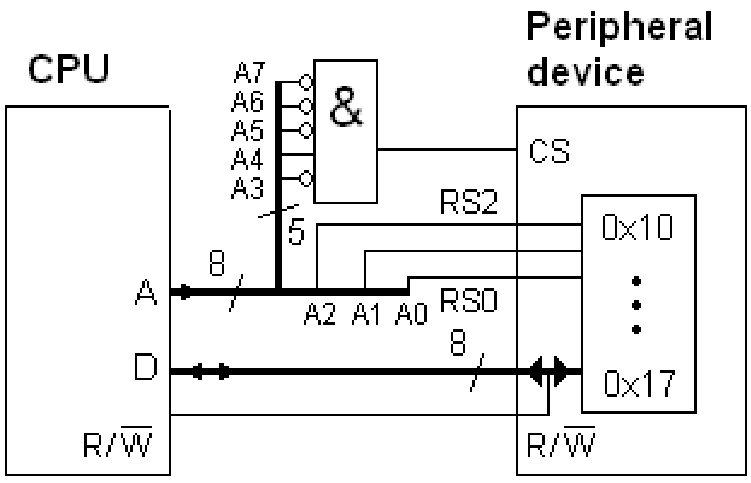
ii. Consider the following pattern of memory addresses: **0x168**, **0x00A**. These addresses should be accessed in an alternating fashion (i.e. the pattern shall be repeated n times). In both memory addresses, bit b_3 is 1, both memory accesses will go to the same set, but because there are two ways in each set, both addresses can be stored in the same set. Therefore, the combination of these memory access will be: CM, CM, H, H, H, H, H, H, . . . , which leads to a general hitrate of $h = \frac{2n-2}{2n}$, which is close to 100% for a large n.

In a direct-mapped cache, the given pattern will always lead to a hitrate of 0%, because the block part of the addresses is the same (bits $b_4 b_3$ are 01_2), which means that they map to the same cache line 1.

Therefore, the 2-way set associative cache performs far better than the direct-mapped cache, $n > 1$, for this pattern.

Solution to Exercise A.1-3 (ID: 0001)

The peripheral device is connected as follows:

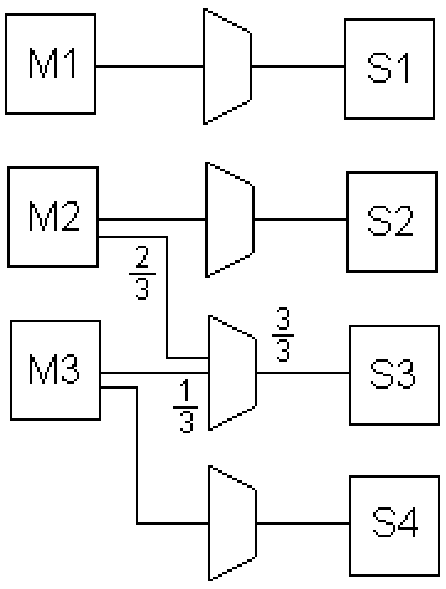


Solution to Exercise A.1-4 (ID: 0004)

(a) A classic bus always means that accesses are handled in sequential fashion. Therefore, the given abstract program will need 7 cycles to be executed completely.

(b)

i.



ii.

see (i.)

iii. Best case: 3 cycles

M1 - Rd. S1	M2 - Rd. S2	M3 - Rd. S4
M1 - Rd. S1	M2 - Rd. S3	
	M2 - Rd. S2	M3 - Rd. S3

iv. Worst case: 4 cycles

M1 - Rd. S1	M2 - Rd. S2	M3 - Rd. S4
M1 - Rd. S1		M3 - Rd. S3
	M2 - Rd. S3	
	M2 - Rd. S2	

Solution to Exercise A.1-5 (ID: 0005)

To solve the problem, it is important to understand how the data is located in memory. The output

```
1) *sp   = 304
```

indicates that the processor uses *little endian* (the least significant bit is located in the lowest memory address). With this knowledge, the current memory content is illustrated in the following figure.

Address	Content	
[x]	04	
[x]+1	03	*sp (Output 1)
[x]+2	02	
[x]+3	01	
[x]+4	14	
[x]+5	13	
[x]+6	12	
[x]+7	11	*sp (Output 2)
[x]+8	24	
[x]+9	23	cp[3] (Output 3)
[x]+10	22	
[x]+11	21	
[x]+13	34	
[x]+14	33	
[x]+15	32	
[x]+16	31	

Thus the remaining output will be:

```
2) *sp   = 1112
3) cp[3] = 23
```

Solution to Exercise A.1-6 (ID: 0006)

The program reads from a location that has not be declared `a[3]`. Many different outputs are possible, because an optimizing compiler may freely decide about the location of the variable `c`!

Examples for correct outputs are:

```
Print text: Hello
Numbers: 1 2 3 0
```

```
Print text: Hello
Numbers: 1 2 3 65
```

```
Print text: Hello
Numbers: 1 2 3 65
```

Solution to Exercise A.1-7 (ID: 0007)

(a) Cache with block size of 32 Bytes:

Line								
0	w[0] w[160]	w[1]	w[2]	w[3]	w[4]	w[5]	w[6]	w[7]
1	w[8]							w[15]
2	w[16]	w[145]	w[274]					w[23]
3	w[24]							w[31] w[159]

Function B will run slower than function A because of a cache conflict.

(b) Cache with block size of 4 Bytes:

Line	
0	w[0] w[160]
...	...
17	w[17] w[145]
18	w[18] w[274]
...	...
31	w[31] w[159]

Function A and function B will not experience cache conflicts and should execute at similar speed.

Solution to Exercise A.1-8 (ID: 0009)

(a) 4 KByte = 2^{12} Byte. Address Mapping for the different caches:

- Direct-Mapped-Cache:

6	2	2	2
Tag	Line	Word	Byte

- 2-way set-associative cache:

7	1	2	2
Tag	Set	Word	Byte

- 4-way-set-associative cache (fully-associative):

8	2	2
Tag	Word	Byte

Thus the memory addresses are mapped to the following cache lines (sets):

Address	Direct-Mapped				2-Way Set-Associative				4-Way Set-Associative		
	Tag	**Line**	Word	Byte	Tag	**Set**	Word	Byte	Tag	Word	Byte
0xCB3	110010	**11**	00	11	1100101	**1**	00	11	11001011	00	11
0xCE0	110011	**10**	00	00	1100111	**0**	00	00	11001110	00	00
0xA24	101000	**10**	01	00	1010001	**0**	01	00	10100010	01	00
0xCBE	110010	**11**	11	10	1100101	**1**	11	10	11001011	11	10
0x0FF	000011	**11**	11	11	0000111	**1**	11	11	00001111	11	11

- Direct-mapped Cache: The addresses 0xCE0 and 0xA24 map to the same cache line. Thus there will be two conflict misses for every run. The addresses 0x0FF, 0xCB3 and 0xCBE also map to the same cache line. Here, 0xCB3 and 0xCBE are part of the same memory block. Thus we get two further conflict misses each round!

- 2-way set-associative cache: The addresses 0xCE0 and 0xA24 map to the same cache set, but since there are two ways to go, there is no conflict miss. Also addresses 0xCB3, 0xCBE (same memory block) and 0x0FF map to the same set, but since there are two ways to go there is no conflict! Thus using a 2-way set-associative cache will only result in cold misses.

- The fully associative cache has 4 ways to go, which is sufficient for not causing any conflict.

Thus the 2-way set associative cache achieves the optimal hit rate at a lower hardware cost than the 4-way set-associative cache.

a) The program will experience four cold misses at the first iteration of the loop. Thus $HitRate = \frac{500-4}{500} = 0.992$.

b) The content of the 2-way set associative cache looks at the end of the program looks as follows:

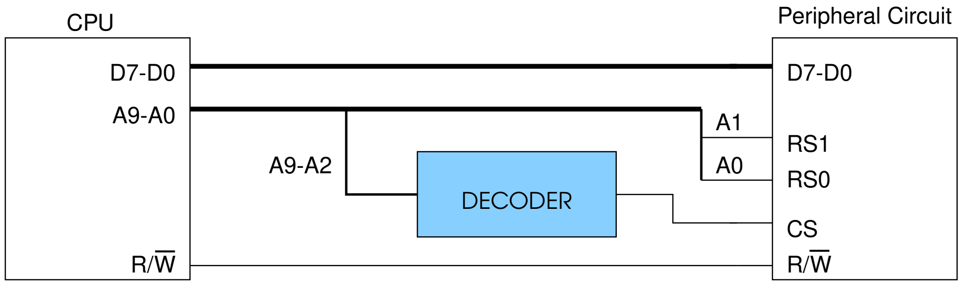
	Way 0					Way 1				
Set	Tag	Word 0	Word 1	Word 2	Word 3	Tag	Word 0	Word 1	Word 2	Word 3
0	0x67	[0xCE0]	[0xCE4]	[0xCE8]	[0xCEC]	0x51	[0xA20]	[0xA24]	[0xA28]	[0xA2C]
1	0x65	[0xCB0]	[0xCB4]	[0xCB8]	[0xCBC]	0x07	[0x0F0]	[0x0F4]	[0x0F8]	[0x0FC]

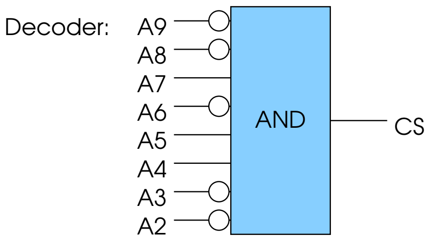
Solution to Exercise A.1-9 (ID: 0010)

a) Translate memory addresses:

Adr	CS	RS_1	RS_0
0x0B0	00101100	0	0
0x0B1	00101100	0	1
0x0B2	00101100	1	0
0x0B3	00101100	1	1

$\mathtt{CS} = \overline{\mathtt{A}_9} \cdot \overline{\mathtt{A}_8} \cdot \mathtt{A}_7 \cdot \overline{\mathtt{A}_6} \cdot \mathtt{A}_5 \cdot \mathtt{A}_4 \cdot \overline{\mathtt{A}_3} \cdot \overline{\mathtt{A}_2}$





b)
```
char pattern = 0x09; // 00001001: bit 3 and 0 set
char register2 = readByte(REGISTER_2);
while(!(register2 & pattern)) { //bitwise and:
                     // result is 0 only if both bit 3 and 0 are 0
    register2 = readByte(REGISTER_2);
/* continue */
}
```

Solution to Exercise A.1-10 (ID: 0011)

- *Big endianness*: the most significant byte is placed first (on the lowest address) and the least significant byte last (on the highest address)
- *Little endianness*: the least significant byte is placed first (on the lowest address) and the most significant byte last (on the highest address).
- Original memory:

Address	0x1000	0x1001	0x1002	0x1003	0x1004	0x1005	0x1006	0x1007
Content	0x24	0xA3	0x53	0x0F	0x01	0x34	0xCC	0x21

- CPU 1 (little endian) writes the four-byte word 0xFEDCBA98 into address 0x1004:

Address	0x1000	0x1001	0x1002	0x1003	0x1004	0x1005	0x1006	0x1007
Content	0x24	0xA3	0x53	0x0F	0x98	0xBA	0xDC	0xFE

- CPU 2 (big endian) reads a two-byte variable from address 0x1002, adds then 256 to this variable, and writes the value as a four-byte word to address 0x1000:
  - CPU 2 reads 0x530F
  - CPU 2 adds 256 (0x100) to 0x530F: 0x540F
  - CPU 2 writes the value as four-byte word 0x0000540F to address 0x1000 (see table)

Address	0x1000	0x1001	0x1002	0x1003	0x1004	0x1005	0x1006	0x1007
Content	0x00	0x00	0x54	0x0F	0x98	0xBA	0xDC	0xFE

Solution to Exercise A.1-11 (ID: 0013)

No answer provided for this exercise.

Solution to Exercise A.1-12 (ID: 0014)

a) Program Executions

i. see Table B.1

Table B.1.: Values after execution of parallel program (no cache)

	A1-B1-B2	B1-A1-B2	B1-B2-A1
x (Memory)	7	7	7
y (Memory)	5	7	7
z (Memory)	6	8	11

ii. see Table B.2

Table B.2.: Values after execution of parallel program (write-through cache)

	A1-B1-B2	B1-A1-B2	B1-B2-A1
x (Memory)	7	7	7
y (Memory)	5	7	7
z (Memory)	6	8	11
x (CPU A)	**5**	7	7
y (CPU A)	5	7	**7**
x (CPU B)	**7**	7	7
y (CPU B)	5	7	**10**
z (CPU B)	6	8	11

Table B.3.: Values after execution of parallel program (write-back-cache)

	A1-B1-B2	B1-A1-B2	B1-B2-A1
`x` (Memory)	5	5	5
`y` (Memory)	10	10	10
`z` (Memory)	?	?	?
`x` (CPU A)	**5**	**5**	**5**
`y` (CPU A)	**5**	**5**	**5**
`x` (CPU B)	**7**	**7**	**7**
`y` (CPU B)	**10**	**10**	**10**
`z` (CPU B)	11	11	11

iii. see Table B.3

b) The inconsistencies are marked in the tables of the previous task.

c) The problem is that processors are not notfied, when data is changed in another processors' cache memory. A possible solution, which also is implemented for cache coherence protocols, is to use the bus for sending additional messages, when data is written to the cache. This is called bus-snooping, and several cache coherence protocols are based on this idea.

Solution to Exercise A.1-13 (ID: 0015)

Contents of the memory where `x[]` is located, after the for-loop:

02	01	12	11	22	21	32	31	42	41	52	51

Each field with two (hexadecimal) digits is one Byte, the double lines mark the individual entries of the array `x[]` of `short`s, i.e. two Bytes.

Note: From outputs 1)-3), we know that the byte order in the memory is little endian, i.e. least significant byte first. Each pointers' type determines how many bytes are read/written (`char`: 1, `short`: 2, `int`: 4). With this knowledge, we can deduce outputs A)-D):

```
A) *ip   = 31322122
B) cp[6] = 32
C) *sp   = 5152
D) cp[1] = 0
```

Solution to Exercise A.1-14 (ID: 0029)

(a)

- The following constellation inside the cache would generate a conflict miss every time:

a[0][0] x[0][0]	a[0][1] x[0][1]	a[0][2] x[0][2]	a[0][3] x[0][3]

- in this constellation, every element of a is mapped to the same cache word as the corresponding element of x, leading to conflict misses every time the inner loop of the given program is executed
- what condition (regarding a's and x's location in the main memory) has to be fulfilled so that this cache constellation will result?
- since we are dealing with a direct mapped cache, the cache mapping depends only on the main memory location
- answer: if the distance between the start location of x and the start location of a in main memory is a multiple of the cache size (i.e. 1024 Words), the given constellation will result

(b)

- The following mapping constellation inside the cache is an example which would generate a conflict miss every fourth time:

	a[0][0]	a[0][1]	a[0][2]
x[0][0] a[0][3]	x[0][1]	x[0][2]	x[0][3]

- To generate a conflict miss every fourth time either $a[i][3]$ must be mapped to the same cache location as $x[i][0]$ or $a[i][0]$ to the same as $x[i][3]$.
- analog to part (a), the modified condition for this constellation is: if the distance between the start location of x and the start location of a in main memory is a multiple of the cache size (i.e. 1024 Words) **+/- 3 Words**, the given constellation will result

(c)

- in order to get no conflict misses at all, the mapping of $a[i][j]$ and $x[i][j]$ may not overlap in the cache:

a[0][0]	a[0][1]	a[0][2]	a[0][3]
x[0][0]	x[0][1]	x[0][2]	x[0][3]

- analog to part (a), the modified condition for this constellation is: if the distance between the start location of x and the start location of a in main memory is **not** a multiple of the cache size (i.e. 1024 Words) +/- 0, 1, 2 or 3 Words, the given constellation will result

Solution to Exercise A.1-15 (ID: 0030)

- $size_{cache} = 128\ Bytes$
- $size_{block} = 32\ Bytes$
- $num_{lines} = size_{cache}/size_{block} = (128\ Bytes/32\ Bytes) = 4$
- number of int-values a cache line can hold:
  $32\ \frac{Bytes}{line}/(4\ \frac{Bytes}{int}) = 8\frac{int}{line}$

(a)

- draw the cache contents to visualize mapping!
- take into consideration the fact that there are 60 int values (u) followed by 60 more int values (w)
- all that is being mapped onto a cache that can hold 32 (4x8) int values at once:

line								
00	u[0] & u[32] w[4] & w[36]	... ...	... ...	... ...	... ...	... ...	... ...	u[7] & u[39] w[11] & w[43]
01	u[8] & u[40] w[12] & w[44]	... ...	... ...	... ...	... ...	... ...	... ...	u[15] & u[47] w[19] & w[51]
10	u[16] & u[48] w[20] & w[52]	... ...	... ...	... ...	... ...	... ...	... ...	u[23] & u[55] w[27] & w[59]
11	u[24] & u[56] w[28]	... ...	... ...	... & u[59] ...	... w[0]	... w[1]	... w[2]	u[31] w[3] & w[35]

- simulate the loop executions and annotate whether the cache will generate a cold miss (CM), conflict miss (M) or hit (H) for each access to u and w.

i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...
u	CM	H	H	H	H	M	M	M	CM	H	H	H	H	M	M	M	...
w	CM	H	H	H	M	M	M	M	H	H	H	H	M	M	M	M	...

- The patterns "M 4xH 3xM" for u and "4xH 4xM" for w will continue to repeat as in the table above. Hence, we get a total hit count of:

i	hit_u	hit_w	hit_{total}
0-7	4	3	7
8-55	6x4	6x4	46
56-59	3	4	7
$\sum$			62

- For all 60x2 cache accesses, we get a hitrate of $h = \frac{62}{120} = 0.52$

(b)

- in order to improve the cache mapping (and therefore reduce conflict misses), we declare a new element that will lie **between** u and w in the

main memory:
int dummy[12];

- this will lead to a different cache mapping constellation:

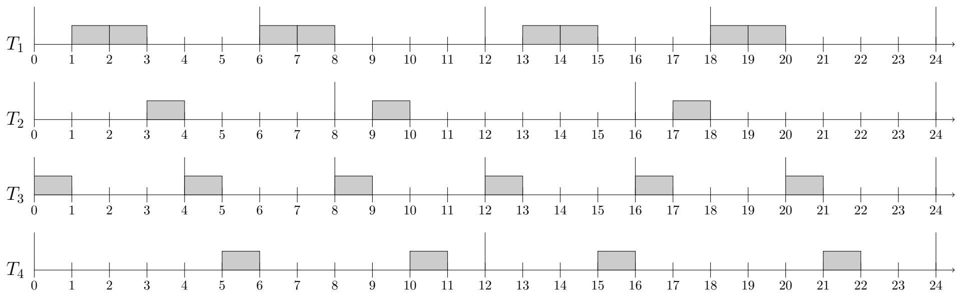
u[0]	u[1]	u[2]	u[3]	u[4]	u[5]	u[6]	u[7]
w[0]	w[1]	w[2]	w[3]	w[4]	w[5]	w[6]	w[7]

- apart from (cold) misses at the beginning of every cache line, there will be only cache hits now
- therefore, our estimation for the hitrate has improved to $h \approx 7/8$
- it is not possible to achieve 100%. Apart from the initial cold misses, there will be a miss for every $i\%8 = 0$ (i.e. when the accessed int is located in a new cache line).

B.2. Real-Time Systems

Solution to Exercise A.2-1 (ID: 0016)

a) $\frac{2}{6} + \frac{1}{8} + \frac{1}{4} + \frac{2}{12} = 0.875$
$1 > 0.875 > 4 \cdot (2^{\frac{1}{4}} - 1) = 0.757$
$\Rightarrow$ a feasible schedule *can* exist: apply RMS algorithm



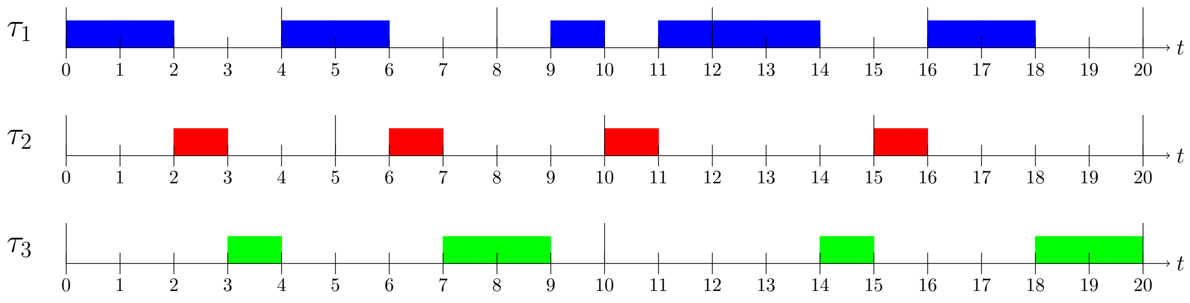
Result: Yes, a feasible schedule exists.

b) $\frac{1}{4} + \frac{1}{5} + \frac{2}{10} + \frac{2}{20} = 0.75$
$0.75 < 4 \cdot (2^{\frac{1}{4}} - 1) = 0.757$
$\Rightarrow$ Result: Yes, a feasible schedule exists.

c) $U = \frac{1}{3} + \frac{1}{12} + \frac{2}{6} + \frac{3}{24} = \frac{8+2+8+3}{24} = \frac{21}{24} = 0.875$
$0.875 > 4 \cdot (2^{\frac{1}{4}} - 1) = 0.757$. However, the set of tasks form a simply periodic system, and since $U < 1$ there exists a feasible schedule.
$\Rightarrow$ Result: Yes, a feasible schedule exists.

Solution to Exercise A.2-2 (ID: 0017)

No, at time 10 τ_1 should be scheduled and not τ_2, since it has the earlier deadline!



Solution to Exercise A.2-3 (ID: 0018)

a) All tasks are ready to execute. Task `t1` is scheduled to execute, because it has the highest priority.

b) Task `t1` will succeed with the `Send` statement twice, since the message queue `queue` is initially empty and has space for two items. During this time it will execute the output statement twice, and will cause the output "`11`".

c) Task `t1` will then be blocked on the `Send` statement.

d) Task `t2` is now the ready task with the highest priority, and will execute the `Receive` statement, which removes one item from the message queue. The message queue is not full anymore. Thus, task `t1` is unblocked and moved via the ready state to the running state, preempting task `2`, which goes back to the ready state.

e) Task `t1` succeeds with the `Send` statement once, executes the print statement, before it is again blocked on the `Send` statement. The output during this phase is "`1`".

f) Task `t2` is moved to the ready state again, executes the print statement and is then blocked on the semaphore. The output during this phase is "`2`".

g) Task `t3` is moved to the ready state and executes the `Signal` statement. This causes task `t2` to be unblocked, so that task `t2` preempts task `t3`.

h) Task `t2` is now the ready task with the highest priority, and will execute the `Receive` statement, which removes one item from the message queue. The message queue is not full anymore. Thus, task `t1` is unblocked and moved via the ready state to the running state, preempting task `2`, which goes back to the ready state.

i) Task `t1` succeeds with the `Send` statement once, executes the print statement, before it is again blocked on the `Send` statement. The output during this phase is "`1`".

j) Task `t2` is moved to the ready state again, executes the print statement and is then blocked on the semaphore. The output during this phase is "**2**".

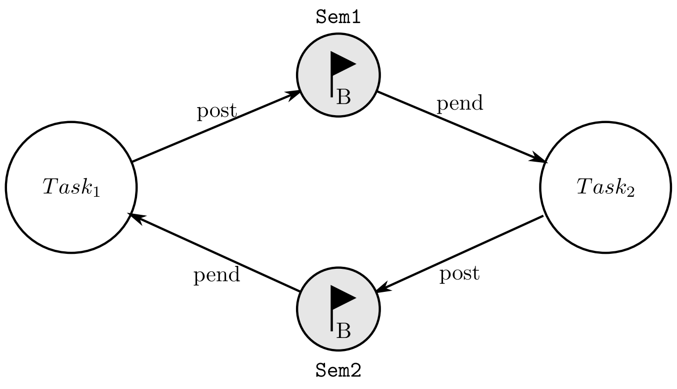
k) Task `t3` is moved to the ready state and executes first the print statement and then the `Signal` statement. This causes task `t2` to be unblocked, so that task `t2` preempts task `t3`. The output during this phase is "**3**".

l) From that moment the pattern from step 0h to step 0k will repeat.

Thus, the ouput is "**1112123123...** "

Solution to Exercise A.2-4 (ID: 0019)

Synchronization using semaphores:



Code:

Task 1:

```
a = f1(...);
releaseSem(Sem1);
// Synchronization point
accessSem(Sem2);
f2(b);
```

Task 2:

```
b = g1(...);
releaseSem(Sem2);
// Synchronization point
accessSem(Sem1);
g2(a);
```

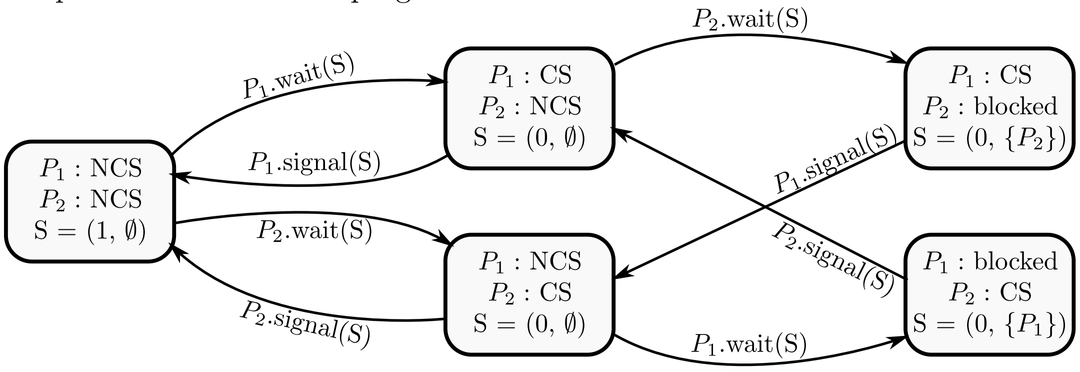
Note that both semaphores shall be initialized with the value zero.

Solution to Exercise A.2-5 (ID: 0020)

We have to look at the density $\Delta_i = \frac{C_i}{min(T_i,D_i)}$, because the relative deadline of τ_2 is shorter than the period of τ_2. The set of tasks is schedulable, if the density $\Delta \leq 1$. Here $\Delta = \frac{2}{5} + \frac{4}{10} + \frac{C_x}{20} \leq 1$. Thus $\frac{8+8+C_x}{20} \leq 1$ means that the largest possible value of C_x is 4.

Solution to Exercise A.2-6 (ID: 0021)

(a) All possible states of the program:



Since there is no state with no outgoing edges, the algorithm is deadlock-free for two tasks.
Since there is *no path* on which a task has issued a `wait` on the semaphore S, but does never acquire it, the algorithm does not starve any of the two tasks. Or with other words: *On all paths*, where the semaphore S is requested by a task, the access is eventually granted to that task.

(b)

	Process P_1	Process P_2	Process P_3	S
(0)	NCS	NCS	NCS	$\{1, \emptyset\}$
	wait(S)			
(1)	CS			$\{0, \emptyset\}$
		wait(S)		
(2)		blocked		$\{0, \{P_2\}\}$
			wait(S)	
(3)			blocked	$\{0, \{P_2, P_3\}\}$
	signal(S)			
(4)	NCS		CS	$\{0, \{P_2\}\}$
	wait(S)			
(5)	blocked			$\{0, \{P_1, P_2\}\}$
			signal(S)	
(6)	CS		NCS	$\{0, \{P_2\}\}$
			wait(S)	
(7)			blocked	$\{0, \{P_2, P_3\}\}$

State (3) and (7) are identical and the schedule can repeat infinitely. Thus an implementation with a weak semaphore can lead to starvation in the sense that a process (like P_2) that has issued a request, may never be served.

Solution to Exercise A.2-7 (ID: 0022)

- Process P_1

```
loop
   wait(S1);
   wait(S2);
```

```
      ...
      signal(S2);
      signal(S1);
   end loop;
```

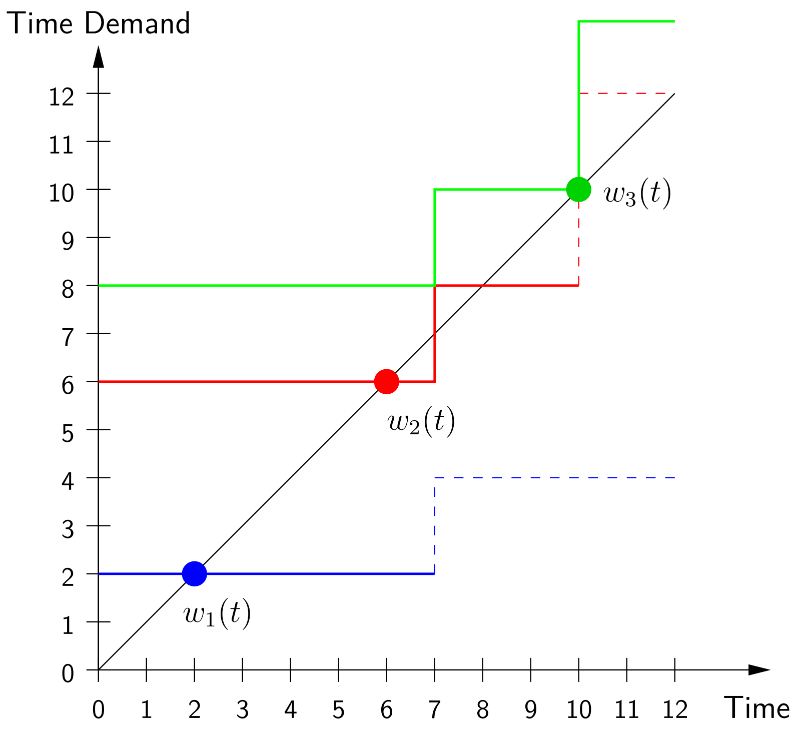
- Process P_1

```
   loop
      wait(S2);
      wait(S1);
      ...
      signal(S1);
      signal(S2);
   end loop;
```

If process P_1 has executed statement `wait(S1)` and process P_2 has executed statement $wait(S2)$ both process cannot proceed.

Solution to Exercise A.2-8 (ID: 0023)

a) The set of tasks is schedulable:



All tasks meet their deadlines: $W_1 = 2$, $W_2 = 6$, $W_3 = 10$

b) The earliest critical instant can be at 21, if $\phi = 9$.

c) The task τ_x has a lower priority than task τ_3. Thus, τ_1, τ_2 and τ_3 will have the same response times as in the previous tasks.

The time demand function for τ_x is:

$w_x(t) = C_x + \lceil\frac{t}{T_1}\rceil C_1 + \lceil\frac{t}{T_2}\rceil C_2 + \lceil\frac{t}{T_3}\rceil C_3 = 1 + \lceil\frac{t}{7}\rceil 2 + \lceil\frac{t}{10}\rceil 4 + \lceil\frac{t}{12}\rceil 2$

We start the iteration at $t^{(0)} = C_1 + C_2 + C_3 + C_x = 2 + 4 + 2 + 1 = 9$.

$$\begin{aligned}
w_x(9) &= 1 + \lceil\frac{9}{7}\rceil 2 + \lceil\frac{9}{10}\rceil 4 + \lceil\frac{9}{12}\rceil 2 = 11\\
w_x(11) &= 1 + \lceil\frac{11}{7}\rceil 2 + \lceil\frac{11}{10}\rceil 4 + \lceil\frac{11}{12}\rceil 2 = 15\\
w_x(15) &= 1 + \lceil\frac{15}{7}\rceil 2 + \lceil\frac{15}{10}\rceil 4 + \lceil\frac{15}{12}\rceil 2 = 19\\
w_x(19) &= 1 + \lceil\frac{19}{7}\rceil 2 + \lceil\frac{19}{10}\rceil 4 + \lceil\frac{19}{12}\rceil 2 = 19
\end{aligned}$$

The task τ_x will complete its execution at $W_x = 19$. All tasks meet their deadline!

Solution to Exercise A.2-9 (ID: 0034)

The following execution orders are possible:

- T_1 - T_2 - T_3: Result: 1
- T_1 - T_3 - T_2: Result: 4
- T_2 - T_1 - T_3: Result: 1
- T_2 - T_3 - T_1: Result: 2
- T_3 - T_1 - T_2: Result: 5
- T_3 - T_2 - T_1: Result: 8

The following results are possible: 1, 2, 4, 5, 8

Solution to Exercise A.2-10 (ID: 0035)

(a) The Rendezvous communication mechanism.

(b)
```
...
task body Client is
    X: Integer;
begin
  loop
    ...
    ServerTask.Request(X);
    ...
  end loop;
end Client;
```

(c)
```
...
task body Server is
```

```
        N: Integer := 0;
    begin
      loop
        ...
        accept Request (X: out Integer) do
          N := N + 1;
          X := N;
        end Request;
        ...
      end loop;
    end Server;
```

B.3. Soutions to Hardware/Software Co-Design

Solution to Exercise A.3-1 (ID: 0024)

$t_{MAC\_instruction} = 0.1 * t_{MAC\_function}$

What is the speed-up, if a new version of the processor would include a multiply accumulate instruction?

Without MAC:
$0.15 + 0.85 = 1$

With MAC:
$(0.1 \cdot 0.15) + 0.85 = 0.865$

Result: speed-up $s = \frac{1}{0.865} \approx 1.16$

Solution to Exercise A.3-2 (ID: 0025)

The expression

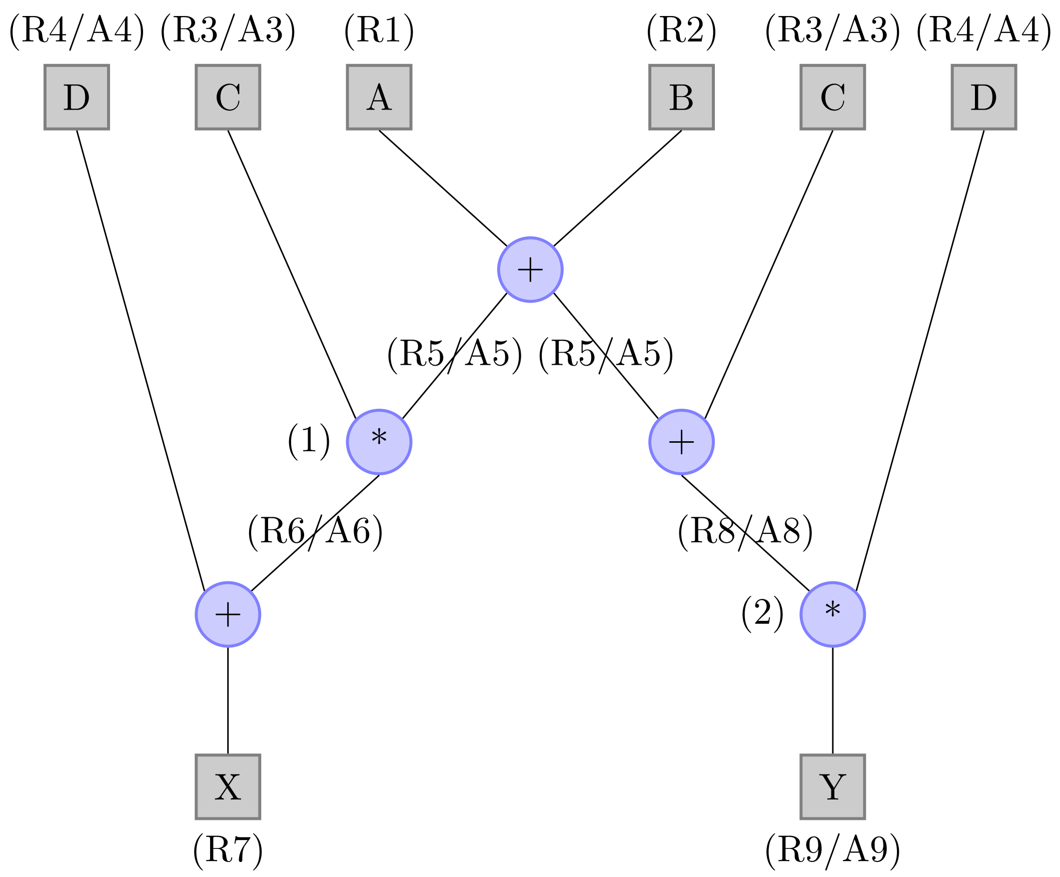
```
X = A * C + B * C + D
Y = (A + B + C) * D
```

can be simplified to

```
X =  (A + B) * C + D
Y = ((A + B) + C) * D
```

so that one multiplier is saved and the term `A + B` can be shared between the expression for `X` and `Y`.

This results in the following data-flow graph:



(a) Instructions and execution times on a single processor:

Processor	**Time Units**
Load R1,A	2
Load R2,B	2
Load R3,C	2
Load R4,D	2
Add R5,R1,R2	1
Mul R6,R5,R3	8
Add R8,R5,R3	1
Add R7,R4,R6	1
Mul R9,R4,R8	8
Store R7,X	2
Store R9,Y	2
SUM:	31

(b)

We have to analyse the following three possibilities:

(i) Both multiplications are implemented on the accelerator

(ii) Multiplication (1) is implemented on the accelerator

(iii) Multiplication (2) is implemented on the accelerator

(i) Both multiplications are implemented on the accelerator

Processor	**Accelerator**	**Time Units (P)**	**Time Units (A)**	**Maxi**
Load R1,A		2		2
Load R2,B		2		2
Load R3,C		2		2
Add R5,R1,R2 Add R8,R5,R3	Load A3,C	1 1	2	2
Store R5,M5		2		2
	Load A5,M5		2	2
Load R4,D	Mul A6,A3,A5	2	2	2
	Store A6,M6		2	2
Load R6,M6		2		2
Store R8,M8		2		2
Add R7,R4,R6	Load A8,M8	1	2	2
	Load A4,D		2	2
Store R7,X	Mul A9,A8,A4		2	2
	Store A9,Y		2	2
SUM				2

Speed-up: $s = time_{orig}/time_{enhanced} = 31/28 \approx 1.11$

(ii) Multiplication (1) is implemented on the accelerator

Processor	**Accelerator**	**Time Units (P)**	**Time Units (A)**	**Maxi**
Load R1,A		2		2
Load R2,B		2		2
Load R3,C		2		2
Add R5,R1,R2 Add R8,R5,R3	Load A3,C	1 1	2	2
Store R5,M5		2		2
Load R4,D		2		2
Mul R9,R4,R8	Load A5,M5 Mul A6,A3,A5 Store A6,M6	8	2 2 2	8
Load R6,M6		2		2
Add R7,R4,R6		1		1
Store R7,X		2		2
Store R9,Y		2		2
SUM				2

Speed-up: $s = time_{orig}/time_{enhanced} = 31/27 \approx 1.15$

(iii) Multiplication (2) is implemented on the accelerator

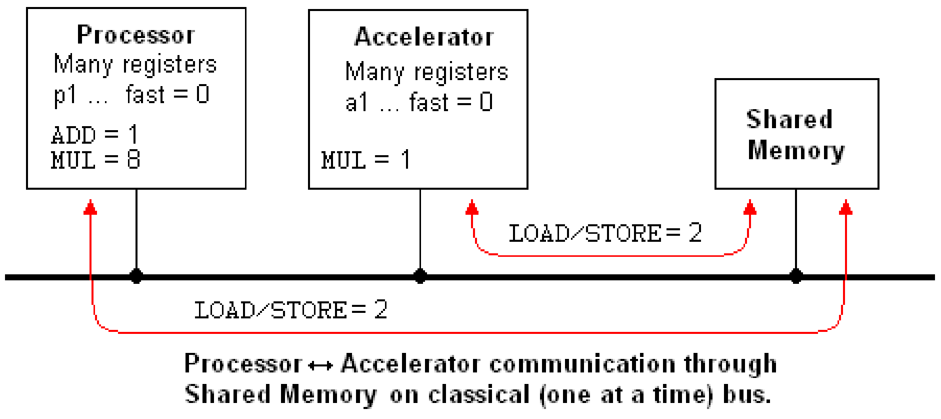
Processor	**Accelerator**	**Time Units (P)**	**Time Units (A)**	**Maximum**
Load R1,A		2		2
Load R2,B		2		2
Load R3,C		2		2
Add R5,R1,R2 Add R8,R5,R3	Load A4,D	1 1	2	2
Store R8,M8		2		2
Mul R6,R5,R3	Load A8,M8 Mul A9,A8,A4 Store A9,Y	8	2 2 2	8
Load R4,D		2		2
Add R7,R4,R6		1		1
Store R7,X		2		2
SUM				23

Speed-up: $s = time_{orig}/time_{enhanced} = 31/23 \approx 1.35$

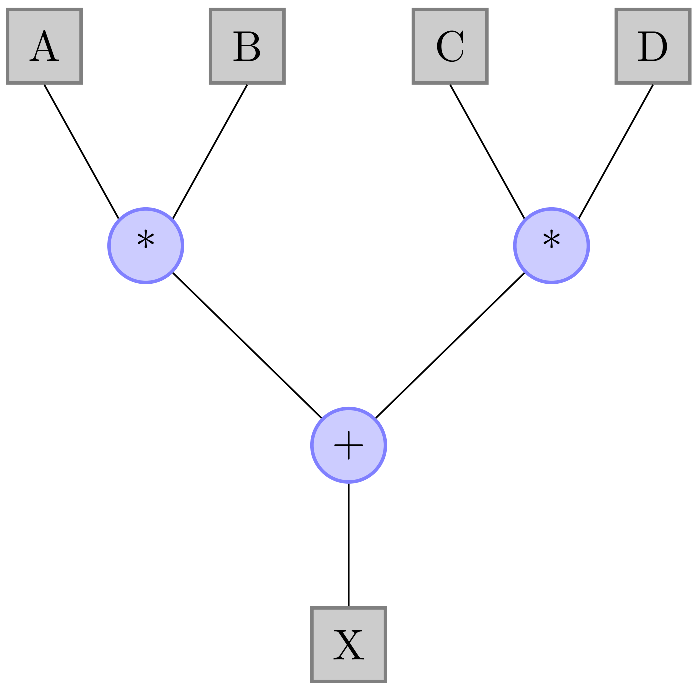
The highest speed-up can be achieved by placing multiplication operation (2) on the accelerator.

Solution to Exercise A.3-3 (ID: 0026)

We are dealing with a classical bus here:



The data-flow graph exposes possible parallelism and data dependencies.



(a) Instructions and execution times:

on processor	time units
load p1,A	2
load p2,B	2
load p3,C	2
load p4,D	2
mul p5,p1,p2	8
mul p6,p3,p4	8
add p7,p5,p6	1
store p7,X	2
SUM:	27

(b)

There are two possibilities: We can either do both multiplications on the accelerator (in sequential fashion), or do one multiplication on the accelerator and the other one on the processor, which allows for parallelism. Instructions and execution times for sequential approach:

on processor	**on accelerator**	**time units**
	load a1,A	2
	load a2,B	2
	load a3,C	2
	load a4,D	2
	mul a5,a1,a2	1
	mul a6,a3,a4	1
	store a5,S	2
	store a6,T	2
load p1,S		2
load p2,T		2
add p3,p2,p1		1
store p3,X		2
SUM		21

Speed-up: $s = time_{orig}/time_{enhanced} = 27/21 \approx 1.29$

Instructions and execution times for parallel approach:

on processor	**on accelerator**	**time units (P)**	**time units (A)**
load p1,A		2	
load p2,B		2	
mul p3,p1,p2	load a1,C	8	2
	load a2,D		2
	mul a3,a1,a2		1
	store a3,T		2
load p4,T		2	
add p5,p4,p3		1	
store p5,X		2	
SUM		17	

Speed-up: $s = time_{orig}/time_{enhanced} = 27/17 \approx 1.59$

Solution to Exercise A.3-4 (ID: 0036)

There are 4 alternatives:

1: f_1, f_2, f_3, f_4 on proc

2: proc and acc

(a) f_1, f_3, f_4 on proc and f_2 on acc

(b) f_1, f_2, f_4 on proc and f_3 on acc

3: f_1, f_4 on proc and f_2, f_3 on acc,

with three processor speeds to choose from in each scenario.

Scenario 1 fulfills the completion time constraint of 800 time units only with processor speed $S = 4$. $cost = 50 \cdot 4^2 = 800$

$t = 4 \cdot 150 + 2 \cdot 50 = 700$

Scenario 2a and 2b, try with $S = 3$:

t	Proc	t	Acc
0	load x		
50	$a = f_1(x)$		
250	store a		
300	$b = f_3(a)$	300	load a
		350	$c = f_2(a)$
		425	store c
500	load c		
550	$y = f_4(b, c)$		
750	store y		
800			

$\Rightarrow t = 800$

$cost = 50 \cdot 3^2 + 200 = 650$

Scenario 2 fulfills the completion time constraint of 800 at a lower cost. The higher processor speed does not need to be considered for this scenario, because it would increase the cost.

Scenario 3 cannot exploit the potential parallelism, while including more communication overhead and performs therefore worse than Scenario 2. For $S = 3$ (because $S = 4$ increases cost compared to the so far optimal solution):

t	Proc	t	Acc
0	load x		
50	$a = f_1(x)$		
250	store a		
		300	load a
		350	$b = f_2(a)$
		425	store b
475	load b	475	$c = f_3(a)$
		550	store c
600	load c		
650	$y = f_4(b, c)$		
850	store y		
900			

Conclusion: Scenario 2a and 2b with $S = 3$ fulfill the design constraint at lowest possible cost.

Solution to Exercise A.3-5 (ID: 0028)

(a)

```
   0.110    (0.75)
+  1.100   (-0.5)
  11
 -------
  10.010
```

Extended sign-bit is discarded, i.e result is $(0.010)_{Q3} = 0.25_{10}$).

(b)

```
0.110 * 1.100
-------------
1.001      (1-complement)
    1      (+1 = 2-complement)
  0110
  11
-------
1.1010     (-0.375)
```

Solution to Exercise A.3-6 (ID: 0027)

(a) 0.8125, -1.375 and 7.5: possible.

0.25: not possible (required exponent can't be represented)
4.25: not possible (significand can't be stored without information loss)

(b)

- align significands (shift binary point of one number such that the exponents of both numbers are equal):
  $1.111exp01 = 0.1111exp10$
- add significands: $0.1111 + 1.010 = 10.0011$
- normalize (alter exponent so that point gets shifted to position after the left-most "1"): $10.0011exp10 = 1.00011exp11$
- round significand (so that it fits into given format): $1.00011 \approx 1.001$
  In order to avoid loss of information, we would need two additional bits to store the significand.

Result: 0 11 001

(c)

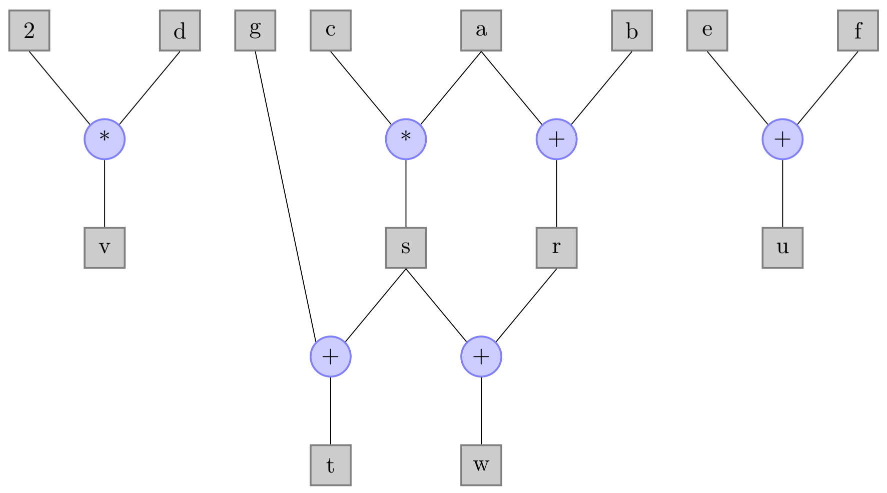
- $exp_{new} = (exp_a + exp_b) - bias = (01 + 10) - 01 = 10$
- multiply significands: $1.111 * 1.010 = 10.010110$
- normalize (alter exponent so that point gets shifted to position after the first "1"): $10.010110exp10 = 1.0010110exp11$

- round significand (so that it fits into given format): $1.0010110 \approx 1.001$

Result: 0 11 001

Solution to Exercise A.3-7 (ID: 0033)

a) Data-flow-graph



b) The instructions can be reordered to save registers:

```
v = 2 * d;   /* 3 */
u = e + f;   /* 2 */
r = a + b;   /* 1 */
s = a * c;   /* 4 */
w = r + s;   /* 6 */
t = g + s;   /* 5 */
```

Then the following register-life-time-graph is received:

Instruction		(3)	(2)	(1)	(4)	(6)	(5)
	a			⊢	⊣		
	b			⊢⊣			
	c				⊢⊣		
	d	⊢⊣					
Variables	e		⊢⊣				
	f		⊢⊣				
	g						⊢⊣
	r			⊢	–	⊣	
	s				⊢	–	⊣
	t						⊢⊣
	u		⊢⊣				
	v	⊢⊣					
	w					⊢⊣	
Number of Registers		2	3	3	**4**	3	3

c) From the life-time graph it follows that *four* registers are needed

Solution to Exercise A.3-8 (ID: 0031)

Optimizations for the given code fragment that result in a shorter execution time, but do not increase code size:

Loop integration: The two for loops have the same range (0, MAX-1) and no data dependency, therefore they can be integrated to save loop overhead:

```
s = 2 * r;
for(i = 0, i<= MAX - 1; i++)
{
  x[i] = f(a[i], b[i]);
  y[i] = s * g(a[i], b[i], c[i]);
}
```

Precalculation at compile time: The defined constant MAX is used as MAX-1 in the loop. MAX-1 could be precalculated as 10-1=9 at compile time.

```
...
for(i = 0; i <= 9; i++)
...
```

alternatively:

```
...
for(i = 0; i < MAX; i++)
...
```

Algebraic simplification: Rewriting function g can save one multiplication operation: $z = a \cdot c - c \cdot b = c \cdot (a - b)$

```
int g(int a, int b, int c)
{
  int z;
  z = c * (a - b);
  return z;
}
```

Variable declaration: The variable z in functions f and g is not needed.

```
int f(int a, int b) {
return 2 * a - b;
}
int g(int a, int b, int c)
{
  return c * (a - b);
}
```

Shift instead of multiplication: A multiplication by 2 can be implemented as a left-shift instead.

```
s = r << 1;
...
 int f(int a, int b) {
 return (a - b) << 1;
 }
```

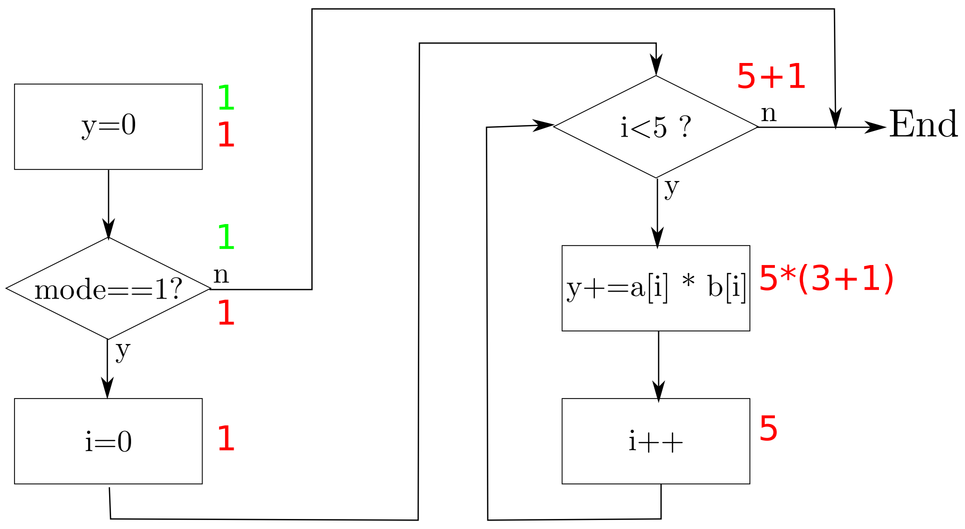
Inlining of functions: Both functions f and g are short and simple. Their code could be inserted directly into the loop.

```
#define MAX 10
int a[MAX], b[MAX], c[MAX], x[MAX], y[MAX];
int i, r, s;
s = 2 * r;
for(i = 0; i <= 9; i++)
{
  x[i] = 2 * a[i] - b[i];
  y[i] = s * ((a[i] - b[i]) * c[i]);
}
```

Loop unrolling: Would give shorter execution time, but it would also increase the code size (!), so it cannot be used in this case.

Solution to Exercise A.3-9 (ID: 0032)

(a)



(b) Best case execution time when mode = 0 (path with green execution times in CDFG above):
$T_{Best} = 1 + 1 = 2$
Worst case execution time when mode = 1 (path with red execution times in CDFG above):
$T_{Worst} = 1 + 1 + 1 + (5 + 1) + (5 * (3 + 1)) + 5 = 34$

(c) Worst case execution time with multiply-accumulate instruction:
$T_{Worst} = 1 + 1 + 1 + (5 + 1) + (\mathbf{5} * \mathbf{1})) + 5 = 19$

C. Programming Embedded Systems in C

The chapter can only give a very short introduction on the programming language C, and its support for embedded systems. A basic understanding of C is assumed in the following presentation. For more information, please consult one of the many text books on C.

C.1. The Programming Language C

Th programming language C has been developed as a systems programming language around 1970. C is a sequential imperative language, and is often viewed as a "macro-assembler", since most C-statements can efficiently be mapped on the instruction set of a von Neumann processor. Java has inherited large parts of the C-syntax, but is a more abstract language.

C has been designed with respect to an efficient implementation. Thus, *unnecessary* features have been left out. For instance there is no Boolean variable. Instead, integers are used for the Boolean values `True` and `False`, where 0 represents `False`, and all other integer values correspond to `True`. Also, C does not have garbage collection, so the programmer has to take the responsibility to allocate and deallocate memory, so that no memory leaks appear. Also, it is possible to access elements in an array that have never been declared.

Thus, C cannot be regarded as a very safe language. The embedded system programmer has a very huge when programming in C. Although the code can be very efficient and fast, it is easy to produce bugs, which are not caught by the compiler. So, programmers have to be very careful when using C!

C.2. Data Types

There is no defined size for the common C data types in the C standard from 1989 (C89), often referred as "ANSI C". The C standard only specifies a *minimum size* for the standard data types, where the integer data types have the following minimum size: `char` (8 bits), `short` (16 bits), `int` (16 bits), `long` (32 bits) and `long long` (64 bitss). Also a relation is defined between these types. The data type `char` is the smallest data type, `short` cannot be larger than `int`, which cannot be larger than `long`. The data type `long long` is the largest integer data type. Also the floating-point data types `float`, `double`, and `long double` are not fully defined and vary by implementations.

The sizeof operator can be used to determine the size of a data type. Thus, the statement `sizeof(int)` returns the size of an integer in bytes.

The C99 standard has defined new fixed size data types in `stdint.h`, like `int8_t`, which defines a signed integer data type with the size of 8 bits, and `uint64_t`, which defines an unsigned integer of 8 bits. It is suggested that you use these data types. However, despite these new data types, the standard integer data types can still be used, so the designer has to be careful if using the original data types.

⚠ Lecture Notes still use the ANSI C (C89) standard

The lecture notes still use the ANSI C data types from the C89 standard. Please use the C99-data types for your own programs or be at least very sure about the size of the data types on your target platform.

C.3. Arrays and Pointers

In order to design efficient programs, it is important to know how data structures are located in the memory. A good understanding can for instance avoid unnecessary cache misses. Of particular importance are arrays, because arrays are often used in loops and thus an efficient handling of arrays can result in an efficient and fast program.

Given the following C-code, where we assume that an integet has the size of 4 bytes,

```c
int x = 5;
int y = 6;
int a[5];
int* intpointer;
```

and assuming that the variable `x` is located at address `0x80`, the variable `y` at address `0x84`, the array `a` has the start address `0x88`, and the pointer variable `intpointer` at address `0x98`, Figure C.1 shows the content of the memory. When an array `a` is declared, its first element `a[0]` is located at the start address `0x80` of the array. Then the next array element `a[1]` will be located directly after `a[0]`. In the example, the array is of integer (`int`) type. Thus `a[1]` is located at address `0x84`. So far neither the array `a` nor the integer pointer `intpointer` have been assigned a value, thus these variables will have some random and unknown value. A pointer variable “points” to a memory location, and thus will have the size given by the address space of the system. In this example, the address space is assumed to be 32 bits, and thus an integer pointer (`int*`) has the size of four bytes.

After executing the following program code,

```c
/* Point to x */
intpointer = &x;
```

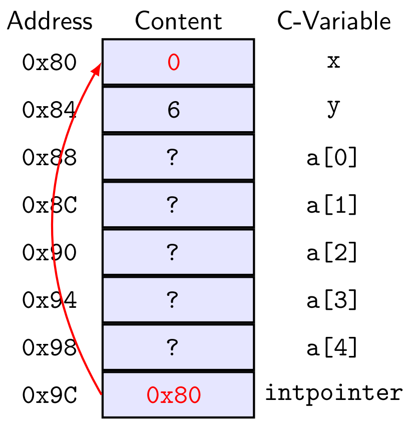


Figure C.1.: Initial configuration of the memory

```
/* Set x to 0 */
*intpointer = 0;
```

the memory content changes as illustrated into Figure C.2. The statement `intpointer = &x`, which uses the *reference operator* `&`, assigns the integer pointer `intpointer` the address of the variable `x`. The following statement `*intpointer = 0` uses the *dereference operator* `*`, and assigns the memory location the pointer `intpointer` points to (`0x80`) the value `0`.

There is a close relation between arrays and pointers. The statement `intpointer = a` is equivalent to `intpointer = &a[0]`, and the statement `intpointer = a + 1` is equivalent to the statement `intpointer = &a[1]`. Please note that the statement `intpointer = a + 1` calculates the new address relative to the size of variable `a`. In this case, the size of the integer type **`int`** is assumed to be four bytes. Thus, `intpointer` will get the address `0x8C`, which is the address of `a` (`0x88`) plus an offset of `4`.

Multidimensional arrays are stored as a sequence of elements where the rightmost subject varies more often. Given the code

```
#define ROW 2
#define COL 3

int a[ROW][COL];
intpointer = a + 4;
*intpointer = 27;
```

and a base address for `a` at `0x80`, an integer area, where **`int`** has a size of four bytes is visualised in Figure C.3, where the column varies more often. As can be seen from the example pointers can be used to access multi-dimensional arrays. Here the statement `a + 4` is equivalent to the statement `a[1][1]`.

Exercise A.1-5 illustrates the correlation between arrays and pointers in C.

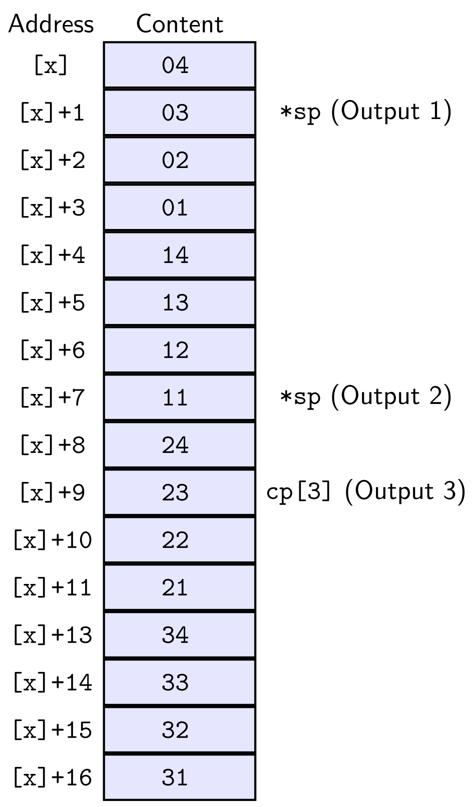


Figure C.2.: Memory after pointer assignment

C.4. Relational, Logical and Bitwise Operators

Since C has been designed for system programming, it does not only define relational and logical operators, but also bit-wise operators.

- *Relation operators:* `<` (less than), `>` (greater than), `<=` (less than or equal), `>=` (greater than or equal), `==` (equal to), `!=` (not equal to)
- *Logical operators:* `!` (negation), `&&` (logical and), `||` (logical or)
- *Bitwise operators:* `~` (bit-wise complement), `&` (bit-wise and), `^` (bit-wise exclusive or), `|` (bit-wise inclusive or), `<<` (left shift), `>>` (right shift)

It is important to note, that the operator `=` is the assignment operator, while `==` is used for equality. Also, when the left shift operator is used, `0`'s are shifted in, but when the right shift operator `>>` is used, it depends on the data type if a `1` or a `0` is shifted in. In case of an unsigned data type, e.g. **`unsigned char`**, `0`'s are shifted in, but in case of a signed data type, e.g. **`char`**, the sign-bit is shifted in!

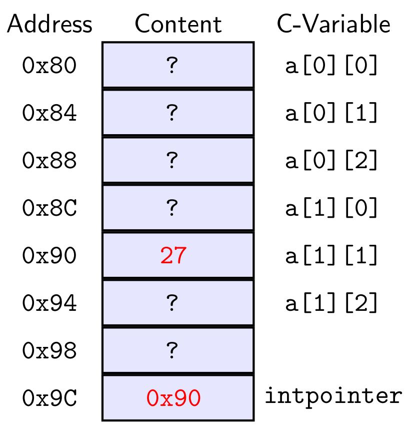


Figure C.3.: Memory location of multidimensional array

C.5. Storage Classes

A variable in C has two attributes:

- *type* (**int**, **char**, **float**, **double**, ...)
- *storage class* (**auto**, **extern**, **register**, **static**)

C.5.1. The storage class auto

The storage class **auto** (automatic) is the default storage class for variables declared in a function. An automatic variable is only visible in the block it is declared in (local variable). The system allocates memory when entering the block, and the system releases the memory when leaving the block, which means that the value of the variable is lost. The following code illustrates the storage class **auto**.

```c
int x; /* storage class extern */
...

int f(int x) {
   /* both a and b are */
   /* of storage class auto */
   int a;
   auto int b;
   ...
   /* a, b are visible here */
   /* x is visible here */
   ...
}
...
```

```c
/* a, b are not visible here */
/* x is visible here */
```

C.5.2. The storage class `extern`

The storage class `extern` is the default storage class for variables declared outside a function. A variable with this storage class is visible in the whole file after its declaration (global variable). The system allocates permanent storage for the variable. The following code illustrates the storage class `extern`.

```c
int x; /* storage class extern */
...

int f(int y) {
   /* both a and b are */
   /* of storage class auto */
   int a;
   auto int b;
   ...
   /* a, b are visible here */
   /* x is visible here */
   ...
}
...
/* a, b are not visible here */
/* x is visible here */
```

C.5.3. The storage class `register`

The storage class `register` can only be declared inside functions. It behaves in the same way as `auto`, but gives an additional recommendation to the compiler that the variable should ideally be placed in a register. The compiler *does not have to follow this advice*, because of the limited number of registers is in a CPU. The following code illustrates the storage class `extern`.

```c
int f() {
   register int i;
   ...
   for(i=0; i < 1000; i++)
      a[i] = i;
   ...
}
```

C.5.4. The storage class `static`

In contrast to variables belonging to the storage class **`auto`**, variables of the storage class **`static`** retain their values when exiting the block. Permanent storage is allocated for these variables. The following code illustrates the storage class **`static`**.

```c
int counter(void) {
   static int i = 0;
   i++;
   return i;
}
```

C.5.5. The type qualifier `volatile`

The type qualifier **`volatile`** indicates that a variable can be changed by other parts of the hardware in the system. Thus, this variable should not be put in register or cache. The use of **`volatile`** is discouraged by many C user groups, because operations are neither atomic nor are **`volatile`** variables thread safe in many current C implementations.

C.6. Masks, bit-fields and unions

The smallest standard data type in C has the size of one byte **`char`**, i.e. 8 bits. If variables are used, which require less bits, it is possible to use specific techniques to minimise the memory requirements for these variables.

C.6.1. Masks

Several variables of small size can be put in a single standard variable, e.g. a **`char`** variable. A typical example are the status flags for a control register in a peripheral device. The following code defines a set of *masks*, which are used to access certain flags (bits) in a variable, and shows the code fragment, which implements a busy wait loop that waits as long as both bit 2 and bit 0 are 0.

```c
/* Set of masks to access specific bits */
#define FLAG0 1
#define FLAG1 2
#define FLAG2 4
...
char flags; /* contains all flags */
...
while ((flags & (FLAG0 | FLAG2)) == 0)
   ;  /* Continue if both flags are 0 */
```

C.6.2. Bit fields

A *bit field* is a packed representation, where several small variables are integrated into one variable. The implementation of the bit field is machine dependent! The following example illustrates the declaration and use of a bit field.

```
struct field {
   unsigned var1 : 4; /* values from 0 to 15 */
   int      var2 : 3; /* values from -4 to 3 */
} x;
...
x.var1 = 6;
```

C.6.3. Unions

A *union* defines a set of alternative values that can be stored in the same portion of a memory location. The programmer has the responsibility to keep track of the data type stored in a union. The size of the union corresponds to the largest data type used in the union. The following example illustrates the declaration and use of a union, where the union variable `u` can either be used as an **`int`** or a **`float`**.

```
union int_or_float {
   int i;
   float f;
} u;

int main(void) {
   u.i = 0;
   printf("i: %10d   f: %16.10e\n", u.i, u.f);
   u.i = 11;
   printf("i: %10d   f: %16.10e\n", u.i, u.f);
   u.f = 11;
   printf("i: %10d   f: %16.10e\n", u.i, u.f);
   return 0;
}
```

The output below shows also clearly that the internal representations for an integer number (**`int`**) or a floating-point **`float`** are completely different. Only the number 0 has the same internal representation.

```
i:          0   f: 0.0000000000e+00
i:         11   f: 1.5414283108e-44
i: 1093664768   f: 1.1000000000e+01
```

C.7. Discussion

C is a very powerful, but also difficult language to master. It has been designed for efficiency, not for robustness. Thus, it is easy to go wrong. Exercise A.1-6 aims at high-lightening the potential traps, which exist in the C programming language. Use it with care! In particular, if it is used in the context of safety-critical systems.

D. Ada

D.1. Introduction

The programming languages Ada[1] has been created as result of an initiative of the United States Department of Defense in order to develop a language for embedded and real-time systems. The reason for this initiative was that very many programming languages had been in use for their embedded computer systems projects, and none of these languages fulfilled their requirements, in particular with respect to safe modular programming. As a result of this effort, the number of programming languages in use in projects for the Department of Defense dropped from about 450 in 1983 to less then 40 in 1996.

There have been several iterations of the Ada standard, the current standard is an ISO standard from 2012, also called Ada 2012. Previous standards were from 1983 (Ada 83), 1995 (Ada 95), and 2005 (Ada 2005).

The Ada language has been designed for the area of safety-critical systems, and is mainly not only used in military applications, but also in many other safety-critical areas. Prominent examples are the fly-by-wire system of the Boeing 777, the Canadian automated air traffic system, and the software for the Ariane 4 and Ariane 5 rockets.

Due to its focus on safety-critical embedded and real-time systems, it has been developed as a robust language, and has several important features: strong typing, modularity mechanisms (packages), run-time checking, exception handling, generics, object-oriented, concurrent processing (tasks), real-time support.

The course will focus on two of these features, which are not provided in a standardised way in other programming languages: the support of task-based concurrency and the support for the development of real-time systems. The course will only give an overview, which shall serve as a first introduction to the capabilities and potential of the language. The presentation is centred around tutorial examples, which aim to introduce the central aspects of the concurrency and real-time features of the Ada language.

The Ada language is very well documented. The Ada standard, in form of the Ada 2012 Language Reference Manual (LRM) (Ada-2012-LRM), the Annotated Ada 2012 Language Reference Manual (AARM) (Ada-2012-AARM), and the Ada 2012 Rationale (Ada-2012-Rationale), is freely available and does not only provide the definition of the language, but also the rationale behind it. Furthermore, there very good and informative textbooks on Ada. The textbook of Barnes (2014) provides detailed information on the

[1]The name Ada is attributed to Augusta Ada King, Countess of Lovelace (1815–1852), who was an English mathematician and writer, and is also considered to be one of the first programmers by developing programs for the Analytical Engine, the proposed mechanical general-purpose computer of Charles Babbage.

different aspects of the Ada 2012 language. The textbook of Burns and Wellings (2007) focuses on the concurrency and real-time aspects, and the recent textbook of Burns and Wellings (2016) discusses both the analysis of real-time systems and the programming of these systems in Ada.

D.2. Structure of an Ada Program

An Ada program consists of several parts. Ada uses the concept of *packages* as modularity mechanism. A package consists of a *package specification*, which defines the *interface* of the package, and a *package body*, which defines the implementation of the package.

The following code shows the package specification of the package `Greetings`, which defines the interface of the procedures `Hello` and `Goodbye`. The file should be saved as `greetings.ads`, because file names should be non-capitalised and should have the suffix `ads` shall be used for specifications.

```ada
package Greetings is
   procedure Hello;
   procedure Goodbye;
end Greetings;
```

The *package body* defines the details of the implementation of the package. Package body files shall be saved with the suffix `adb`.

```ada
with Ada.Text_IO; use Ada.Text_IO;

package body Greetings is
   procedure Hello is
   begin
      Put_Line ("Hello WORLD!");
   end Hello;

   procedure Goodbye is
   begin
      Put_Line ("Goodbye WORLD!");
   end Goodbye;
end Greetings;
```

The main program `Gmain` uses the package Greetings. Also the main program shall be saved in a file with the suffix `adb`. The main program is defined as a procedure and uses the procedures `Hello` and `Goodbye` defined in the package `Greetings`.

```ada
with Greetings;  -- "include Package 'Greetings'"

```

```
procedure Gmain is
begin
   Greetings.Hello;
   Greetings.Goodbye;
end Gmain;
```

D.3. GNAT - The GNU Compiler for Ada

The course will use GNAT[2], the GNU compiler of Ada, which is freely available under the GNU license. Here, only a short overview about GNAT can be given. For more detailed information, see the GNAT User's Guide for Native Platforms (GNAT-User-Guide).

The Ada program `Gmain` can be compiled using by compiling both the main program file `gmain.adb` and the file `greetings.adb` for the package `Greetings` that it uses.

```
$ gcc -c gmain.adb
$ gcc -c greetings.adb
```

This command generates the corresponding object file `*.o.` and an Ada library information file `*.ali`, which contains extra information about the consistency of the Ada program.

To build an executable file `gnatbind` is used to bind the program and `gnatlink` is used to link it. Both commands take the `ali`-file as argument, but the extension can be omitted.

```
$ gnatbind hello
$ gnatlink hello
```

This produces an executable program `gmain`, which results in the following output:

```
Hello WORLD!
Goodbye WORLD!
```

Instead of using the commands `gcc -c`, `gnatbind` and `gnatlink`, is to use the command `gnatmake`, which executes all of these programs in the correct order.

```
$ gnatmake gmain.adb
```

[2]All examples have been compiled with GNAT version 7.5.0 on Ubuntu 18.04.4 LTS.

D.4. Tasks

Tasks are supported in Ada directly in the language. Concurrent activities are defined by means of tasks, which have a *task specification* and a *task body* according to the following structure.

```
task type T is    -- Specification
   ...
end T;

task body of T    -- Body
   ...
end T;
```

The exact scheduling policy can be defined using the real-time annex.

The following concurrent program consists of two identical tasks `T1` and `T2`, which both access the global variable `N`. At program start both tasks and also the main program is started. The main program also executes as a concurrent task. The statement `delay 2.0` suspends the main program for 2 seconds, so that it is ensured that the tasks `T1` and `T2` have finished.

```
procedure SimpleTasks is
   N : Integer := 0;
   task type Simple;
   T1, T2 : Simple; -- two tasks defined, both
                    -- have the same body
   task body Simple is
   begin
      for I in 1..20 loop
         N := N + 1;
      end loop;
   end Simple;
begin
   delay 2.0; -- Wait 2 seconds, so that
              -- both tasks have completed
   Put("N = " & Integer'Image(N));
end SimpleTasks;
```

Because tasks run concurrently, the order of execution plays an important role for the final result. The following concurrent program can result in different final values.

```
with Ada.Text_IO;
use Ada.Text_IO;

```

```
procedure Concurrency1 is
   N : Integer := 0;
   task Task1;
   task body Task1 is
   begin
      for I in 1..2 loop
         N := N + 1;
      end loop ;
   end Task1;

   task Task2;
   task body Task2 is
   begin
      for I in 1..2 loop
         N := N * 2;
      end loop ;
   end Task2;
begin
   delay 1.0; -- Wait 1 sec
   Put_Line("N = " & Integer'Image ( N ));
end Concurrency1;
```

D.5. Communication Mechanisms

Ada defines two powerful communication mechanisms, protected objects (see Section D.5.1) and Rendezvous (see Section D.5.2).

D.5.1. Protected Object

A *protected object* is a powerful and flexible communication mechanism that is based on the concept of the monitor. A protected object is specified by a *specification*, which provides the access protocols to the protected object and the *body*, which gives the details of the specification.

```
protected Object is -- Specification
   -- Access Protocol
end Object;

protected body Object is -- Body
   -- Implementation details
end Object;
```

To illustrate the protected object, the following system shall be modelled by means of a protected object. There is a box, which is used to store coins. Different people can put money into the box, and also out of the box. The amount of money stored in the box can be viewed as a shared variable, which is accessed by several tasks (people) by well-defined access functions. To keep the system general, the following access methods shall be defined on the shared variable: `Add`, `Subtract`, `Read`, and `Write`. It is important that only one of the access methods `Add`, `Subtract`, and `Write` is executed and finished before another of the access methods is started. But, several tasks can issue and conduct a `Read` operation simultaneously.

A protected object can use *protected procedures* and *protected functions*, which have well-defined behaviours with respect to the concurrent access to a protected object. A *protected procedure* provides mutually exclusive read/write access to the data encapsulated. A *protected function* provides concurrent read-only access to the encapsulated data.

Thus, the protected object specification uses protected procedures for the access methods `Add`, `Subtract` and `Write`, because mutually exclusive read/write access needs to be ensured, and a protected function for the access method `Read`, because it allows concurrent read access. The shared variable `X` is defined as *private* with an initial value, and can only be accessed by the access functions.

```ada
protected type Shared_Variable(Initial_Value : Integer)
is
   procedure Add(Number : Integer);
   procedure Subtract(Number : Integer);
   procedure Write(Number : Integer);
   function Read return Integer;
private
   X : Integer := Initial_Value;
end Shared_Variable;
```

The package body gives the implementation of the four access methods.

```ada
package body Shared is
   protected body Shared_Variable is
      procedure Write(Number : Integer) is
      begin
                          X := Number;
      end Write;

      procedure Add(Number : Integer) is
      begin
                          X := X + Number;
      end Add;

      procedure Subtract(Number : Integer) is
      begin
                        X := X - Number;
      end Subtract;

      function Read return Integer is
      begin
                        return X;
      end Read;
   end Shared_Variable;
end Shared;
```

The following concurrent program `Box_App` creates five tasks, which access the protected object `Box` of type `Shared_Variable`, which is initialised with the value 2. The syntax for the access to the protected object uses the “dot”-notation of the form `protected_object.access_method`. In the program six tasks are created, which either put a coin into the box, remove a coin from the box or check the content of the box. Please also note, that the main program, which constitutes an own task, only contains the `null` statement. Thus the “main task” will directly terminate.

```
with Ada.Text_IO; use Ada.Text_IO;
with Shared; use Shared;

procedure Box_App is

   Box : Shared_Variable(2);

   task type Put_Money_Task;
   task type Get_Money_Task;
   task type Check_Money_Task;

   task body Put_Money_Task is
   begin
      for I in 1..3 loop
         Box.Add(1);
         Put_Line("Added 1 coin - Coins:" & Integer'Image(Box.Read));
      end loop;
   end Put_Money_Task;

   task body Get_Money_Task is
   begin
      for I in 1..3 loop
         Box.Subtract(1);
         Put_Line("Taken 1 coin - Coins:" & Integer'Image(Box.Read));
      end loop;
   end Get_Money_Task;

   task body Check_Money_Task is
   begin
      for I in 1..3 loop
         Put("Coins:");
         Put_Line(Integer'Image(Box.Read));
      end loop;
   end Check_Money_Task;

   P1, P2 : Put_Money_Task;
   T1, T2 : Get_Money_Task;
   C1, C2 : Check_Money_Task;

begin
   null;
end Box_App;
```

Because the program is a concurrent program, different execution will lead to different results. Repeating simulations also show that there are some flaws with the program. The current model of the shared variable does not ensure that a) coins cannot be taken, if the box is empty; and that b) coins cannot be put into the box, if the box is full. A simple solution would be to include **if**-statements in the procedure body, but this would not block the task from execution. The protected object provides a better solution, which allows to block a task, if a condition is not met. Instead of a protected procedure a *protected entry*, which is guarded by a Boolean expression can be used. The following code shows how protected object is specified with the protected entries `Add` and `Subtract`. In addition the protected objected contains now also the additional private variables `Min_Value` and `Max_Value`.

```
protected type Shared_Variable(Initial_Value : Integer;
                               Min_Value : Integer;
                               Max_Value : Integer) is
   entry Add(Number : Integer);
   entry Subtract(Number : Integer);
   procedure Write(Number : Integer);
   function Read return Integer;
private
   X : Integer := Initial_Value;
   Max : Integer := Max_Value;
   Min : Integer := Min_Value;
end Shared_Variable;
```

The body of the protected objected shows how to define a condition as part of the protected entry, which has to be fulfilled in order to enter the access method defined by the protected entry. This condition is formulated using a Boolean expression, called a barrier, which uses the **when** statement.

```ada
package body Shared_Guard is
   protected body Shared_Variable is
      entry Add(Number : Integer) when X < Max is
      begin
         X := X + Number;
      end Add;

      entry Subtract(Number : Integer) when X > Min is
      begin
         X := X - Number;
      end;

      procedure Write(Number : Integer) is
      begin
         X := Number;
      end Write;

      function Read return Integer is
      begin
         return X;
      end Read;
   end Shared_Variable;
end Shared_Guard;
```

The *protected entry* is similar to a protected procedure in that it is guaranteed to execute in mutual exclusion and has read/write access to the encapsulated data. A protected entry is guarded by a Boolean expression, called a barrier, inside the body of the protected object. If this barrier evaluates to false when the entry call is made, the calling task is suspended until the barrier evaluated to true and no other tasks are currently active in the protected object. Figure D.1 illustrates the protected entries in the protected object. Several tasks can be blocked by the barrier of a protected objects, and only one can execute a protected entry due to the mutual exclusive access that guaranteed by the protected object for protected procedures and protected entries.

The protected project is a very powerful communication mechanism, which is implemented by a *passive* communication object, which contains private data that can be accessed via access methods. The protected object ensures mutually-exclusive write access. Task communication using the protected object is based on accessing the shared data within the protected object.

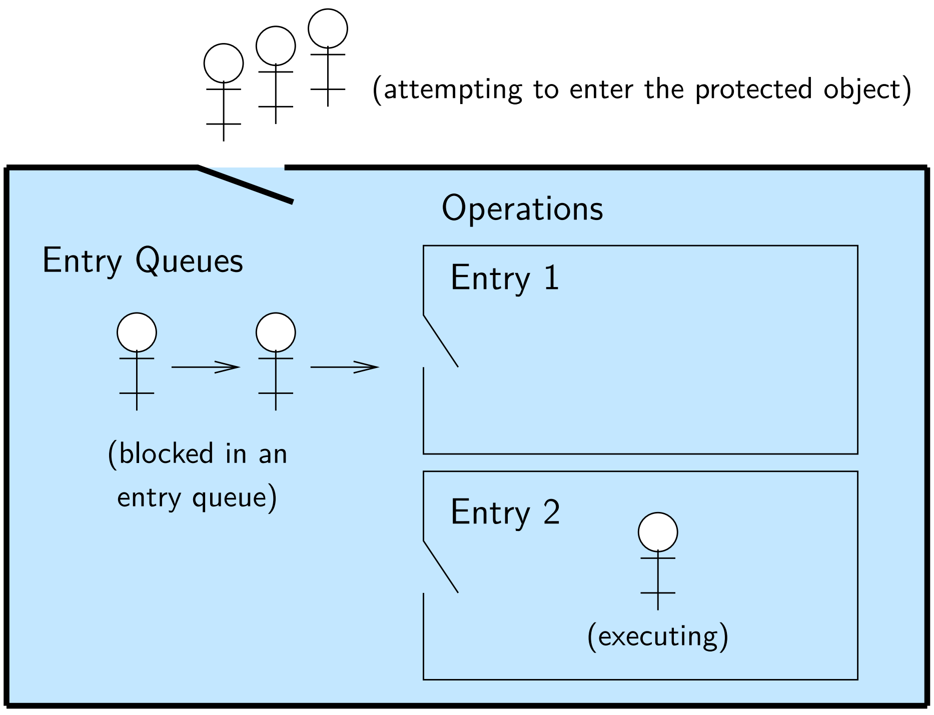

(attempting to enter the protected object)
Operations
Entry Queues
Entry 1
(blocked in an entry queue)
Entry 2
(executing)

Figure D.1.: Protected entries in a protected object

D.5.2. Rendezvous

In contrast to the protected object, *rendezvous*, the second communication mechanism supported by Ada, is based on the communication via an active task, which takes the role of a *server*, while the other tasks are *clients* requesting services from the server task. During a rendezvous, the client task and the server task meet and synchronise for a short amount of time. The tasks take the following roles during a rendezvous:

- **Calling process (client).** The calling process calls an entry of the accepting process. The calling process must know the identity of the accepting process (server) and the identity of the rendezvous (an *entry*).
- **Accepting process (server).** The server process offers services in form `accept`-statements. The location of the rendezvous belongs to the accepting process. The accepting process does not need to know the identity of the calling process.

Figure D.2 illustrates the rendezvous communication. A server task `Server` provides a service `Request`, which can be accessed by the client tasks. In this example the server executes an infinite loop and waits for clients that request the service `Request` by executing the statement `Server.Request`. The client waits until the accepting process is ready to accept the call. During the rendezvous, the accepting process produces a result using parameters that have been provided by the caller. At the completion of the rendezvous, the calling process receives the result and is unblocked and can proceed. The server can now accept a new service request.

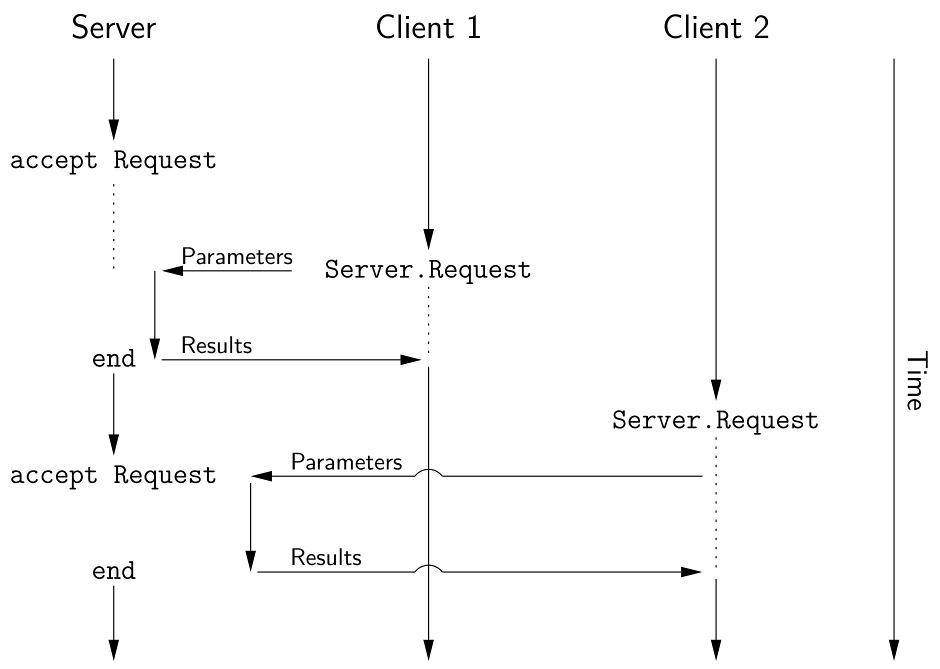


Figure D.2.: Rendezvous: Several client tasks can request services from a server task.

Ada implements the rendezvous mechanism. Tasks can communicate with each other directly by sending messages to each other. The following program implements the previous example of the box with money by means of rendezvous.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Box_App_Rendezvous_Select is

   task type Box(Initial_Value : Integer;
                 Min_Value : Integer;
                 Max_Value : Integer) is
      entry Add(Number : Integer);
      entry Subtract(Number : Integer);
      entry Read;
   end Box;

   task body Box is
      X : Integer := Initial_Value;
   begin
      loop
         select
            when X < Max_Value =>
               accept Add(Number : Integer) do
                  X := X + Number;
               end Add;
         or
            when X > 0 =>
               accept Subtract(Number : Integer) do
                  X := X - Number;
               end Subtract;
         or
            accept Read do
               Put("Coins:");
               Put_Line(Integer'Image(X));
            end Read;
--          or
--             delay 5.0;
--             Put_Line("No request received for 5 seconds");
         or
            terminate;
         end select;
      end loop;
   end Box;

   B : Box(2, 0, 4); -- Box includes initially 2 coins
                     -- Size : 4 Coins

   task type Put_Money_Task;
   task type Get_Money_Task;
   task type Check_Money_Task;

   task body Put_Money_Task is
   begin
      for I in 1..5 loop
         B.Add(1);
      end loop;
   end Put_Money_Task;

   task body Get_Money_Task is
   begin
      for I in 1..5 loop
         B.Subtract(1);
      end loop;
   end Get_Money_Task;

   task body Check_Money_Task is
   begin
      for I in 1..5 loop
         B.Read;
      end loop;
   end Check_Money_Task;

   P1, P2 : Put_Money_Task;
   T1, T2 : Get_Money_Task;
   C1, C2 : Check_Money_Task;

begin
   null;
end Box_App_Rendezvous_Select;
```

The program contains new features, which are explained in the following. A calling task calls an *entry*, the method that implements the service, in the called task. The entry is specified as follows.

```
task CalledTask is
   entry E(...);
end CalledTask;
```

Each entry has an associated queue of tasks waiting to call the entry. Usually this queue is processed in a first-in-first-out manner. The calling task calls the entry in the called task using the "dot"-notation, in the form of `Server.Service`, here `CalledTask.E(...)`. Each entry is accompanied by an **`accept`**-statement in the body of the called task, which specifies the code that shall be executed during the rendezvous.

```
accept E(...) do
  ...  -- sequence of statements
end E;
```

Guards using **`when`**- and **`select`**-statements can be used to express more complex conditions for a rendezvous. The box is modelled as a *server task*, which contains a set of services, which are offered to the other *client tasks*.

The **`accept`** statement specifies the actions to be performed when an entry is called. Only, when these conditions are fulfilled, the rendezvous can take place. For each and every entry defined in a task there must be at least one **`accept`** statement in the corresponding task body. The entry `Add` has a guard in for of the **`when`** `X < Max_Value`.

```
when X < Max_Value =>
   accept Add(Number : Integer) do
      X := X + Number;
   end Add;
```

The **`select`**-statement allows a server to

- wait for more than a single rendezvous at any time (otherwise services will be served in the order of the **`accept`**`-statements`);

- time out and execute other statements, if no client request is issued with in a specified period specified by a **delay**-statement;
- terminate if no client can possibly call its entries (**terminate**);
- withdraw its offer to communicate if no rendezvous is immediately available (not discussed here).

A selective accept can be combined with guards (`when`-statements) as seen in the source code above.

D.6. Real-Time Annex

The real-time annex defines additional semantics and facilities for real-time applications. It has been significantly extended in the Ada 2005 standard. The real time annex covers among others scheduling policies, support for priority ceiling protocol, real-time clock, timers.

D.6.1. Time in Ada

Ada defines two different packages that deal with time

- `Ada.Calendar` provides an abstraction for wall clock time
- `Ada.Real_Time` specifies a high-resolution, monotonic (non-decreasing) clock package. It is defined in the real-time annex.

The current time is returned by the function `Clock` which has no parameters. Ada defines many functions on time in the package `Ada.Real_Time`. Many functions, like the function + are overloaded to work with different types. Since Ada is a strongly-typed language, it is important to keep track of the types, e.g. `Time`, `Time_Span` and `Duration` are different types! The following code shows a small part of the package specification of `Ada.Real_Time`.

```ada
package Ada.Real_Time is
  type Time is private;
  ...
  type Time_Span is private;
  ...
  function Clock return Time;
  ...
  function "+" (Left : Time; Right : Time_Span)
     return Time;
  function "+" (Left : Time_Span; Right : Time)
     return Time;
  function "+" (Left, Right : Time_Span)
     return Time_Span;
  ...
  function To_Duration (TS : Time_Span)
     return Duration;
  function To_Time_Span (D : Duration)
     return Time_Span;
  ...
  function Microseconds (US : Integer)
     return Time_Span;
  function Milliseconds (MS : Integer)
     return Time_Span;
  function Seconds      (S  : Integer)
     return Time_Span;
  ...
end Ada.Real_Time;
```

There are usages of the statement `delay`.

- `delay 5.0` delays the task for five seconds.
- `delay until` will delay a task until a certain absolute time.

D.6.2. Periodic Tasks

To implement a periodic task in Ada, the function `Clock` and the `delay until` statement has to be used as illustrated in the program below.

```
task type T(Id: Integer; Period_Ms : Integer) is
   pragma Priority(System.Default_Priority + Id);
end;

task body T is
   Interval : Time_Span := Milliseconds(Period_Ms);
   Next : Time;
begin
   Next := Clock;
   loop
      -- Do something
      Next := Next + Interval;
      delay until Next; -- Wait until start of period
   end loop;
end T;
```

The first time the task is released, the current time is saved before an infinite loop is entered. For each task instance, after execution of the computation function, the next absolute deadline is calculated and stored in the variable `Next`. Then the task is suspended with the statement `delay until` until the beginning of the beginning of the next period. The task can also have a priority, which is specified with the pragma `Priority`. The higher the number, the higher the task's priority.

D.6.3. Scheduling Policies

Ada allows to define different scheduling policies by using the pragma `Task_Dispatching_Policy(`
where policy can be one out of four scheduling policies.

FIFO Within Priorities Within each priority level tasks are scheduled on a first-in-first-out basis. A task may preempt another task with lower priority.

Non-Preemptive FIFO Within Priorities A task runs to completion and cannot be pre-empted by a task with higher priority.

Round-Robin Within Priorities Within each priority level tasks are time-sliced with an interval that can be specified.

EDF Across Priorities Each task has relative deadline (given by `Relative_Deadline`). Tasks within a range of priorities are scheduled according to the earliest-deadline-first algorithm.

Scheduling policies can be combined in order to divide a set of tasks into hard and soft real-time tasks.

```
pragma Priority_Specific_Dispatching
          (Round_Robin_Within_Priorities, 1, 4);
pragma Priority_Specific_Dispatching
          (FIFO_Within_Priorities, 5, 20);
```

Normally hard real-time tasks should be given *distinct priorities*, if they are used with a priority-driven scheduling algorithm, like the Rate-Monotonic (RM) algorithm. In this example, priority levels 5 to 20 use the policy `FIFO_Within_Priorities` in order to implement hard real-time tasks, while levels 1 to 4 are used for best-effort or soft-real time tasks. Background tasks with no timing requirements can be scheduled with round-robin scheduling, in which case a time-sharing policy will be used. Please note, that in this example, there are four different priority levels for round-robin tasks. Thus, a round-robin task with priority level 2 will have priority over a task of priority level 1. The following code is used to exemplify this behaviour.

```
pragma Priority_Specific_Dispatching(Round_Robin_Within_Priorities, 1, 2);

with Ada.Text_IO; use Ada.Text_IO;
with Ada.Real_Time; use Ada.Real_Time;
```

```ada

procedure Round_Robin_Priorities is
   package Int_IO is new Ada.Text_IO.Integer_IO(Integer);

   Dummy: Integer;

   task type Round_Robin(Id: Integer; Prio: Integer) is
      pragma Priority(Prio);
   end Round_Robin;

   task body Round_Robin is
   begin
      loop
         Put("Round Robin Task ");
         Int_IO.Put(Id, 1);
         Put_Line("");
         for I in 1..1000 loop -- Waiting loop
            for J in 1..50000 loop
               Dummy := I * 3 - 2 * J; -- Dummy Function
            end loop;
         end loop;
      end loop;
   end Round_Robin;

   -- Round Robin Tasks
   RR_1 : Round_Robin(1, 1);
   RR_2 : Round_Robin(2, 2);
   RR_3 : Round_Robin(3, 2);
begin
   null;
end Round_Robin_Priorities;
```

Task RR_1 will never run, when executing this program in a single thread processor, because it has a lower priority than RR_2 and RR_3. The tasks RR_2 and RR_3 will run according to the round-robin policy.

```
ada> sudo taskset -c 0 ./round_robin_priorities
Round Robin Task 2
Round Robin Task 3
Round Robin Task 2
Round Robin Task 3
...
```

D.6.4. Resource Access Control

Ada supports the *priority ceiling protocol* by allowing to assign a priority to a protected object.

```
protected Object is
   pragma Priority(20);
   entry E;
   ...
```

If the pragma `Locking_Policy(Ceiling_Locking` is used than the priority ceiling protocol will be used to control the access of shared resources. A task calling the entry `E` will inherit the ceiling priority (20) while executing the protected operation.

D.6.5. The Ravenscar Profile

The *Ravenscar* profile is a restricted subset of the Ada tasking model in order to meet requirements for safety-critical real-time systems on

- determinism
- schedulability analysis
- memory boundness

and to allow for a small and efficient run-time system that supports task synchronisation and communication. The Ravenscar profile was defined in Ada 2005 and can be used by the pragma `Profile(Ravenscar)`. Only the protected object is supported in the Ravenscar profile. The rendezvous mechanism is *not supported* due to the difficulty to do a timing and schedulability analysis for systems that use rendezvous communication.

D.6.6. GNAT and the Real-Time Annex

Although GNAT supports the full real-time annex, it depends on the underlying architecture and operating system, which of the GNAT features are implemented. On a operating system like Linux or Windows, EDF-scheduling is not supported, while fixed-priority scheduling, like Rate-Monotonic Schedule (RMS), is supported.

Real-Time Programs need to run in Supervisor Mode

Programs that use the real-time annex have to run in supervisor mode. Otherwise the programs will not be scheduled according to the real-time annex.

Priority-Driven Concurrent Tasks on a Multiprocessor

A concurrent priority-driven program behaves different, if scheduled on a single-processor or a multiprocessor architecture. If one processor shall be used, use a command to enforce an execution on only one processor. In Linux this can be done with the following command `sudo taskset -c 0 ./name_of_program`.

If you on the other hand want to use several cores, you can specify this also with the `taskset` command. If the cores 0 to 3 shall be used, then use the command `sudo taskset -c 0,1,2,3 ./name_of_program`. For more information on your cores on your machine, run `sudo less /proc/cpuinfo`.

E. Hardware Design with VHDL

The objective of the chapter is to give embedded software designers an overview about digital hardware design with a hardware description language (HDL). In this case we use VHDL[1]. However, another very popular language is Verilog. For a deeper information view one of the numerous books on these HDLs, such as Ashenden (2002) for VHDL and Ciletti (2003) for Verilog.

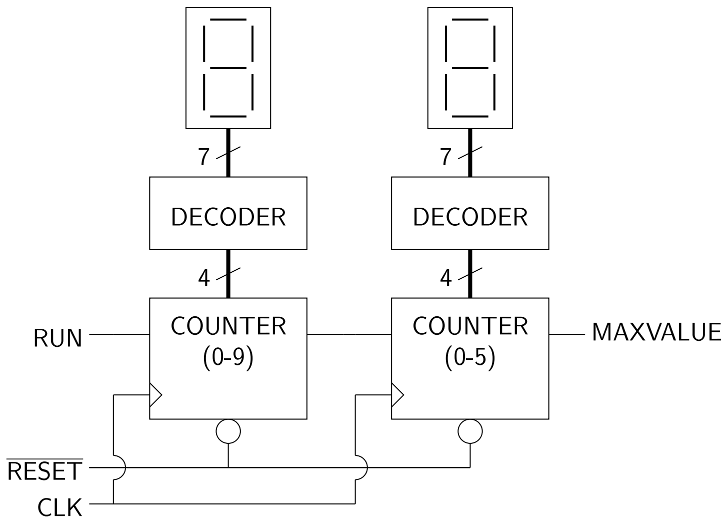


Figure E.1.: A counter that counts between 0 and 59 and displays the number on two seven-segment displays

Figure G.6 shows an example of a simple counter that will be used to illustrate how to design hardware using VHDL. The counter counts from zero to 59 and restarts then counting from 0. The design consists of two counters and two seven-segment decoders.

We start the discussion with the design of the seven-segment decoder, which is a combinational circuit. A design consists of an *entity*-description and an *architecure*-description. The entity defines the interface to the environment, while the architecture defines the functionality.

Listing E.1 shows the entity description of the seven-segment decoder. The decoder has an input signal `number`, which is of type `std_logic_vector` with a range from three downto 0. The data type `std_logic` can be viewed as special data type for bits[2]. The

[1]VHDL = VHSIC HDL, where VHSIC = Very High Speed Integrated Circuit

[2]The data type `std_logic` does not only have the values `'0'` and `'1'`, but also seven additional values for instance for `'U'` (uninitialized), `'X'` (unknown), `'Z'` (high impedance), or `'-'` (don't care). Some of these values are mainly used for simulation.

Listing E.1. The entity description of the seven-segment decoder. An entity defines the interface of a design.

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

entity SevenSegDecoder is

  port ( number : in std_logic_vector(3 downto 0);
         segments : out std_logic_vector(6 downto 0));

end SevenSegDecoder;
```

output signal is another `std_logic_vector` with seven bits.

Listing E.2 shows the architecture description of the seven-segment decoder. The architecture contains a single process with the signal `number` in its sensitivety list. Inside an architecture processes run concurrently and are executed, whenever a value in their sensitivety list is changed. The statements inside a process are executed sequentially. Here, the architecture uses a `case`-statement to implement the functionality of the seven-segment decoder.

Since both counters only differ by their counting range, the presentation is restricted to the design of the counter counting to nine. The other counter is designed accordingly.

Listing E.3 shows the entity description of the counter counting from 0 to 9. The counter is a sequential circuit and has a clock input `clk` and an active low reset input `reset_n`. Whenever the input signal `run` is active, the counter is counting, otherwise it is stopped. The output signal `number` is a `std_logic_vector` with a size of four bits. Whenever the counter reaches its maximum value, the output value `max_value` will be set to `'1'`.

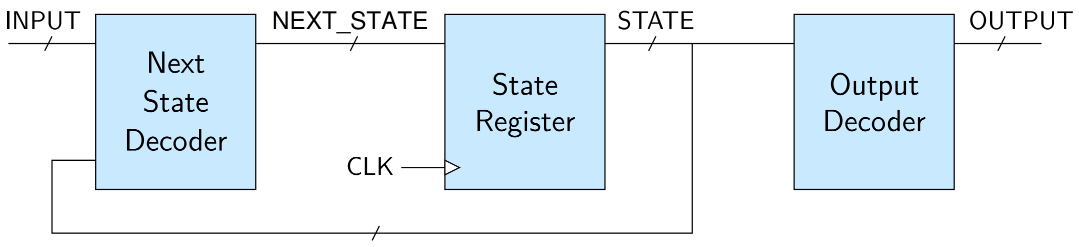


Figure E.2.: In a Moore FSM the current output value does only depend on the current state but not on the input value.

The architecture of the counter is given in Listing E.4. The is implemented as finite state machine (FSM) of Moore type as shown in Figure E.2, i.e. the output does only depend on the state of the FSM.

The design consists of three processes, which run in parallel. There is one process for

Listing E.2. The architecture description of the seven-segment decoder. An architecture description defines the functionality of a design.

```vhdl
architecture behavioral of SevenSegDecoder is

begin  -- behavioral

  process (number)
  begin  -- process
    case number is
      when "0000" => segments <= "1000000";  -- 0 turns LED on
      when "0001" => segments <= "1111001";
      when "0010" => segments <= "0100100";
      when "0011" => segments <= "0110000";
      when "0100" => segments <= "0011001";
      when "0101" => segments <= "0010010";
      when "0110" => segments <= "0000010";
      when "0111" => segments <= "1111000";
      when "1000" => segments <= "0000000";
      when "1001" => segments <= "0010000";
      when others => segments <= "1111111";
    end case;
  end process;

end behavioral;
```

the state registers, `clk_process`, one process for the next state decoder, `next_state_process`, and one process for the output decoder `output_process`. It uses two internal signals to denote the current state, `current_state`, and the next state `next_state`. The data type for these signals is a subtype of integers, which can take the values from 0 to 9.

The sequential process `clk_process` implements state registers that are triggered on the positive clock edge. Whenever there is a positive clock edge, the signal `current_state` gets the value of the signal `next_state`. The state registers can be reset using an active low asynchronous reset.

The combinational process `next_state_process` implements the next state decoder. As long as the current state is below `max_count`, the output `next_state` is the value of the current state plus one. In case the current state is the maximum value, the next state is zero.

The combinational process `output_process` calculates the output signals `max_value` and `number`. Whenever the current state reaches its maximum value, the output `max_value` is `'1'`, otherwise it is `'0'`. The output `number` is a four-bit-representation of the current state.

Finally, we generate the design of Figure G.6 using the structural capabilities of VHDL,

Listing E.3. The entity description of the counter counting from 0 to 9

```vhdl
entity counter_0_9 is

  port (
    reset_n : in std_logic;
    clk : in std_logic;
    run : in std_logic;
    number :out std_logic_vector(3 downto 0);
    max_value : out std_logic );

end entity counter_0_9;
```

by generating a netlist of components. The entity `Seconds` of this component is illustrated in Listing E.5.

The architecture description of the top-level entity is shown in Listing E.6 and E.7. This architecture description uses the structural capabilities of VHDL by describing the system as a netlist of components. The first part lists the components that are uses in the design. The second part describes the mapping from each input or output pin of an instantiated component to a input-, output- or internal signal.

This design can then be synthesized using a synthesis tool. Figure E.3 shows the summary of the synthesis results for a synthesis using the Altera Quartus tool.

```
+-------------------------------------------------------------------------------+
; Flow Summary                                                                  ;
+-------------------------------------+-----------------------------------------+
; Flow Status                         ; Successful - Wed Apr 15 15:01:40 2009   ;
; Quartus II Version                  ; 8.0 Build 231 07/10/2008 SP 1 SJ Full Version ;
; Revision Name                       ; top                                     ;
; Top-level Entity Name               ; Top                                     ;
; Family                              ; Cyclone II                              ;
; Device                              ; EP2C35F672C6                            ;
; Timing Models                       ; Final                                   ;
; Met timing requirements             ; Yes                                     ;
; Total logic elements                ; 23 / 33,216 ( < 1 % )                   ;
;     Total combinational functions   ; 23 / 33,216 ( < 1 % )                   ;
;     Dedicated logic registers       ; 7 / 33,216 ( < 1 % )                    ;
; Total registers                     ; 7                                       ;
; Total pins                          ; 25 / 475 ( 5 % )                        ;
; Total virtual pins                  ; 0                                       ;
; Total memory bits                   ; 0 / 483,840 ( 0 % )                     ;
; Embedded Multiplier 9-bit elements  ; 0 / 70 ( 0 % )                          ;
; Total PLLs                          ; 0 / 4 ( 0 % )                           ;
+-------------------------------------+-----------------------------------------+
```

Figure E.3.: Summary of synthesis results using the Altera Quartus tool.

Listing E.4. The architecture description of the counter counting from 0 to 9

```vhdl
architecture FSM of counter_0_9 is
  constant max_count : integer := 9;
  subtype state_type is integer range 0 to max_count;
  signal current_state, next_state : state_type := 0;
begin  -- FSM

  clk_process: process(clk, reset_n)
    begin
      if reset_n = '0' then
        current_state <= 0;
      else
        if clk'event and clk = '1' then
          current_state <= next_state;
        end if;
      end if;
    end process;

    next_state_process: process (run, current_state)
    begin  -- process
      if run = '1' then
        if current_state = max_count then
          next_state <= 0;
        else
          next_state <= current_state + 1;
        end if;
      else
        next_state <= current_state;
      end if;
    end process;

    output_process: process (current_state)
    begin  -- process output_process
      if current_state = max_count then
        max_value <= '1';
      else
        max_value <= '0';
      end if;

      case current_state is
        when 0 => number <= "0000";
        when 1 => number <= "0001";
        when 2 => number <= "0010";
        when 3 => number <= "0011";
        when 4 => number <= "0100";
        when 5 => number <= "0101";
        when 6 => number <= "0110";
        when 7 => number <= "0111";
        when 8 => number <= "1000";
        when 9 => number <= "1001";
        when others => null;
      end case;

    end process output_process;
end FSM;
```

Listing E.5. The entity description of the counter counting from 0 to 59.

```vhdl
entity Seconds is

  port (
    reset_n   : in  std_logic;
    clk       : in  std_logic;
    run       : in  std_logic;
    max_value_tens : out std_logic;
    segments_tens : out std_logic_vector(6 downto 0);
    segments_ones : out std_logic_vector(6 downto 0));

end Seconds;
```

Listing E.6. First part of the architecture description of the counter counting from 0 to 59. This architecture description uses the structural capabilities of VHDL by describing the system as a netlist of components.

```vhdl
architecture Structure of Seconds is

  -- Components
  component sevenSegDecoder
    port (
      number   : in  std_logic_vector(3 downto 0);
      segments : out std_logic_vector(6 downto 0));
  end component;

  component counter_0_5
    port (
      reset_n : in std_logic;
      clk : in std_logic;
      run : in std_logic;
      number :out std_logic_vector(3 downto 0);
      max_value : out std_logic );
  end component;

  component counter_0_9
    port (
      reset_n : std_logic;
      clk : in std_logic;
      run : in std_logic;
      number :out std_logic_vector(3 downto 0);
      max_value : out std_logic );
  end component;

  -- Internal signals
  signal max_value_ones : std_logic;
  signal dec_in_ones, dec_in_tens : std_logic_vector(3 downto 0);
```

Listing E.7. Second part of the architecture description of the counter counting from 0 to 59. This architecture description uses the structural capabilities of VHDL by describing the system as a netlist of components.

```vhdl
begin
  decoder_tens : sevenSegDecoder
    port map (
      number   => dec_in_tens,
      segments => segments_tens);

  decoder_ones : sevenSegDecoder
    port map (
      number   => dec_in_ones,
      segments => segments_ones);

  counter_tens : counter_0_5
    port map (
      clk       => clk,
      reset_n   => reset_n,
      run       => max_value_ones,
      number    => dec_in_tens,
      max_value => max_value_tens);

  counter_ones : counter_0_9
    port map (
      clk       => clk,
      reset_n   => reset_n,
      run       => run,
      number    => dec_in_ones,
      max_value => max_value_ones);

end Structure;
```

F. Introduction to Haskell

Since the functional language Haskell is used to express the system models in ForSyDe, this section gives a small introduction to functional languages and Haskell in particular. Many examples are taken from "A Gentle Introduction to Haskell 98" (Hudak et al., 1999). There exist several good textbooks, like (Bird, 2014) and online resources on Haskell (Lipovača, 2011).

A functional program is a function that receives the program's input as argument and delivers the program's output as result. Usually the main function is defined in terms of other functions, which can be composed of still other functions until at the bottom of the functional hierarchy the functions are language primitives. Each function is free from side-effects, i.e. they have no internal state. This means that the whole functional program is free from side-effects and thus behaves totally deterministic. Given the same inputs, the functional program will always produce identical outputs. Since all functions are free from side-effects, the order of evaluation is only given by data dependencies. But this means also that there may exist several possible orders for the execution of a functional program. Considering the function

$$f(x, y) \quad = \quad u(h(x), g(y))$$

the data dependencies imply that the functions $h(x)$ and $g(y)$ have to be evaluated before $u((h(x), g(y))$ can be evaluated. However, since there is no data dependency between the functions h and g, there are the following possible orders of execution:

- $h(x)$ is evaluated before $g(y)$;
- $g(y)$ is evaluated before $h(x)$;
- $h(x)$ and $g(y)$ are evaluated in parallel.

Thus functional programs contain implicit parallelism, which is very useful when dealing with embedded system applications, since they typically have a considerable amount of built-in parallelism. Of course it is also possible to parallelize imperative languages like C++, but it is much more difficult to extract parallelism from programs in such languages, since the flow of control is also expressed by the order of statements.

The foundations of functional languages have been discussed in depth in (Hudak, 1989), while (Hughes, 1989) discusses how functional languages can be used to improve modularity. There is also a study (Hudak and Jones, 1994) that supports the claimed advantages of functional languages - brevity, rapidity of development and ease of understanding - over conventional imperative languages as C++ and Ada.

The following part introduces the functional programming language Haskell and shows some important features.

In addition to common data types, such as `Bool`, `Int` and `Double`, the standard prelude also defines lists and tuples. An example for a list is `[1,2,3,4] :: [Integer]`, which is a list of integers. The notation "`::`" means "has type". An example for a tuple, which is a structure of different types is `('A', 3) :: (Char, Integer)` where the first element is a character and the second one is an integer.

Haskell has adopted the Hindley-Milner type system (Milner, 1978), which was developed for the functional language ML (Gordon et al., 1978). It has the following significant features (Hudak, 1989):

1. It is strongly and statically typed.
2. It uses type inference to determine the types of every expression, instead of relying on explicit type declarations.
3. It allows polymorphic functions and data structures; that is, functions may take arguments of arbitrary type, if in fact the function does not depend on that type (similarly for data structures).
4. It has user-defined constructs and abstract data types.

The type system is not only strongly typed, like the type system for VHDL or Ada, but it is also capable to infer (calculate) the maximal possible data type for an expression. Given the function

```
fst (x, y) = x
```

which returns the first value of a pair, Haskell's type system will determine the type as

```
fst :: (a,b) -> a
```

This type declaration means that `fst` is a function that has a tuple (a pair) as input parameter, where the first value is of some data type `a` and the second value is of some data type `b`. The output value is of data type `a`. The type declaration does not imply that `a` and `b` have to be of a different type, but since the type declaration gives the maximal type, it is allowed that they are of different types.

The function `fst` can now be used with all kinds of pairs, such as `fst(3, [1,2,3])`, `fst(1,2)` or `fst('A', 3)`.

Haskell is based on the lambda-calculus and allows to write functions in *curried* form, after the mathematician Haskell B. Curry, where the arguments are written by juxtaposition. The function `add` is written in curried form.

```haskell
add :: Num a => a -> a -> a
add x y = x + y
```

Since "`->`" associates from right to left, the "real" type of `add` is `add :: Num a => a -> (a -> a)`. This means that given the first argument, which is of a numeric type `a`, it returns a function from `a` to `a`. This can be used for *partial application* of a curried function. New functions can then be defined by applying the first argument, e.g.

```haskell
inc x = add 1
dec x = dec 1
```

These functions only have one argument and the following type

```haskell
inc :: Num a => a -> a
dec :: Num a => a -> a
```

It is not possible to use partial application, if the *uncurried* form of `add` is used,

```haskell
add :: Num a => (a, a) -> a
add (x, y) = x + y
```

since there is only one argument, which is a tuple of two values and must be supplied as a whole.

Another powerful concept in functional languages is the *higher-order function*. A higher-order function is a function that takes functions as argument and/or produces a function as output. An example of a higher-order function is `map`, which takes a function and a list as argument and applies ("maps") the function `f` on each value in the list. The function is defined as follows

```haskell
map f []     = []                -- Pattern 1 (empty list)
map f (x:xs) = f x : map f xs -- Pattern 2 (all other lists)
```

The higher-order function uses an additional feature of the language, which is called *pattern matching* and is illustrated by the evaluation of `map (+1) [1,2,3]`.

During an evaluation the patterns are tested from the top to the bottom. If a pattern, the left hand side, matches, the corresponding right hand side is evaluated. The expression `map (+1) [1,2,3]` does not match the first pattern since the list is not empty (`[]`). The second pattern matches, since (`x:xs`) matches a list that is constructed of a single value and a list. Since the second pattern matches, the right hand side of this pattern is evaluated. This procedure is repeated recursively until the first pattern matches, where the right hand side does not include a new function call. As this example shows, lists are constructed and processed from head to tail.

```
   map (+1) [1,2,3]
⇒  map (+1) (1:[2,3])        Pattern 2 matches
⇒  1+1 : map (+1) [2,3]      Evaluation of Pattern 2
⇒  2 : map (+1) (2:[3])      Pattern 2 matches
⇒  2 : 2+1 : map (+1) [3]    Evaluation of Pattern 2
⇒  2 : 3 : map (+1) (3:[])   Pattern 2 matches
⇒  2 : 3 : 4 : map (+1) []   Evaluation of Pattern 2
⇒  2 : 3 : 4 : map (+1) []   Pattern 1 matches
⇒  2 : 3 : 4 : []            Evaluation of Pattern 2
⇒  [2,3,4]
```

The higher-order function map can now be used with all functions and lists that fulfill the type declaration for map, which Haskell infers as

```haskell
map :: (a -> b) -> [a] -> [b]
```

The type declaration reads as follows. The first argument of map is a function that takes a value of some data type a and returns a value of another data type b. The second argument is a list of some data type b and the result is a list of some data type b.

Thus functions as fst and map are polymorphic and can be used with several types. However, Haskell has a static type system, that ensures that Haskell programs are *type safe*, i.e. all type errors are detected at compile time. The type system will allow function calls like map even [1,2,3], which will be evaluated to a list of Boolean values [False, True, False] :: [Bool], but reject the list [1,'A',3] since this list contains elements of different types.

A very powerful higher-order function is *function composition*, which is expressed by the composition operator " . ".

```haskell
(.)        :: (b -> c) -> (a -> b) -> (a -> c)
f . g      = \x -> f (g x)
```

This definition uses "lambda abstractions" and is read as follows. The higher-order function f . g produces a function that takes a value x as argument and produces the value $f(g(x))$. The expression f = (+3) . (*4) creates a function f that performs $f(x) = 4x + 3$. Function composition is extremely useful in ForSyDe since it is allows to merge processes in a structured way.

Haskell allows to define own data types using a data declaration. It allows for recursive and polymorph declarations. A data type for a list could be recursively defined as

```haskell
data AList a = Empty
             | Cons a (AList a)
```

The declaration has two *data constructors*. The data constructor `Empty` constructs the empty list and `Cons` constructs a list by adding a value of type `a` to a list. Thus `Cons 1 (Cons 2 (Cons 3 Empty))` constructs a list of numbers. The term *type constructor* denotes a constructor that yields a type. In this case `AList` is a type constructor.

As mentioned before, in Haskell the list data type is predefined. Here `[]` corresponds to `Empty` and `:` to `Cons`. `[a]` corresponds to `AList a`.

An important aspect of Haskell is that it uses *lazy evaluation*, which is in contrast to *eager evaluation*. Here, the value $\perp_T$ is used to denote the *value* for a non-terminating expression or an expression that result in some kind of run-time error, such as `1/0`. A function f is *strict*, if

$$f(\perp_T) = \perp_T$$

In most program languages *all* functions are strict and this means that all arguments to a function must terminate and must not cause a run-time error in order to terminate f. But this is not the case in a lazy language as Haskell. Consider the constant 1 function `const1`, defined by:

```haskell
const1 x = 1
```

The result of `const` $\perp_T$ `1` in Haskell is `1`, since Haskell never attempts to evaluate its argument as the result is always `1`. Functions like `const1` are *non-strict* functions, which are also called "lazy functions", as they evaluate their arguments "lazily" or "by need". Thus expressions like `const1 (1/0)` evaluate to `1`. In an eager language like Standard ML Milner et al. (1997), where all functions are strict, the result will be a run-time error since the program tries to evaluate `1/0`.

Lazy evaluation is of great use when dealing with possibly infinite data structures such as signals, since the program will only evaluate an expression when needed. Thus, it is possible to define an infinite list of the natural numbers `nat` as

```haskell
nat = numsFrom 1
where
  numsFrom n = n : numsFrom (n+1)
```

Of course it is still not possible to evaluate the whole (infinite structure) `nat`, but finite parts of it can be evaluated such as

```haskell
head nat                 =>    1
take 5 nat               =>    [1,2,3,4,5]
(take 5 . map (^2)) nat  =>    [1,4,9,16,25]
```

Abstract data types can be defined using the Haskell module system. A module is a collection of functions. Inside a module all functions and data types are visible for each other. Other modules may only access the functions that are explicitly exported by the module. The following code shows a part of the module `ForSyDe.Shallow.MoC.SDF`, which provides process constructors for the synchronous data flow model of computation (Section 5.2.2) and is part of ForSyDe-Shallow.

```haskell
module ForSyDe.Shallow.MoC.SDF (
  ...
  -- * Sequential Process Constructors
  -- | Sequential process constructors are used for processes that
  -- have a state. One of the input parameters is the initial state.
  delaySDF,
  ...
  -- * Actors
  -- | Based on the process constructors in the SDF-MoC, the
  -- SDF-library provides SDF-actors with single or multiple inputs
  actor11SDF, actor12SDF, actor13SDF, actor14SDF,
  actor21SDF, actor22SDF, actor23SDF, actor24SDF,
  actor31SDF, actor32SDF, actor33SDF, actor34SDF,
  actor41SDF, actor42SDF, actor43SDF, actor44SDF
  ) where

import ForSyDe.Shallow.Core
...
```

The module imports the library `ForSyDe.Shallow.Core`, which defines the core language of ForSyDe-Shallow. It exports the process constructors for initial tokens `delaySDF` and the process constructors for actors beginning with `actor11SDF` until `actor44SDF`. Functions that are defined in the module, but not listed in the parameter list are not visible to other modules.

G. ForSyDe

G.1. Overview

ForSyDe is a methodology with a formal basis for modelling and design of heterogeneous embedded systems that has been developed with the following objectives (Sander, 2003) in mind:

- System design must start at a high level of abstraction. The designer shall focus on functionality, and low-level implementation details shall not be an issue at the design stage.
- The design methodology must give a solid base for the incorporation of formal methods. This means that verification must be a first class citizen from the start.
- The abstraction gap between specification and implementation must be bridged by formal refinement techniques.

ForSyDe is based on the theory of models of computation as formulated by Lee and Sangiovanni-Vincentelli (1998). ForSyDe provides several libraries for different MoCs. For more information visit the ForSyDe web page.

The installation instructions for **`ForSyDe-Shallow`**, the main modelling environment of ForSyDe, are available on the (ForSyDe) home page. It is also recommended to run the tutorial on **`ForSyDe-Shallow`** to get a better understanding of the modelling concepts of ForSyDe. Appendix F provides a short introduction to Haskell, the language that **`ForSyDe-Shallow`** is implemented in.

G.2. Synchronous Data Flow Model of Computation

ForSyDe provides a modelling library for the SDF MoC. This library provides the necessary process constructors to create SDF actors and initial tokens, which can then be composed into an executable SDFG model.

The following tutorial gives an introduction how to model SDF applications in ForSyDe and will use the application of Figure G.1. The purpose of this application is only tutorial, but it introduces the main concepts to model SDFGs in ForSyDe.

The application of Figure G.1 consists of two actors and one initial token. The first actor A_1 performs an addition operation, while the second actor A_2 takes the average of three consecutive tokens. There is also an arc constituting a self loop connecting an output of the first actor with an input of the same actor. It is important to point out,

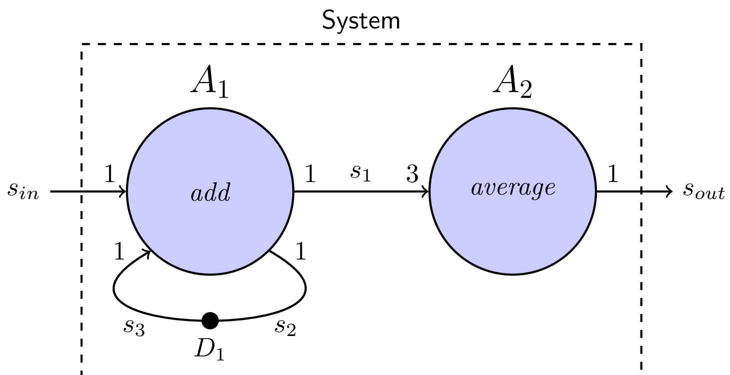


Figure G.1.: A tutorial example for SDF

that the output signals of actor A_1, s_1 and s_2, carry identical information, but SDF requires that they are modelled as separate arcs, because s_1 and s_2 have different destinations. The first step is to include the necessary modelling libraries and to create the process network for the application. **ForSyDe-Shallow** includes several MoC libraries, including the SDF library, and needs to be imported. The netlist in form of a set of concurrent ForSyDe processes can be directly created from the specification as illustrated in Figure G.2 and is then formulated as a set of equations.

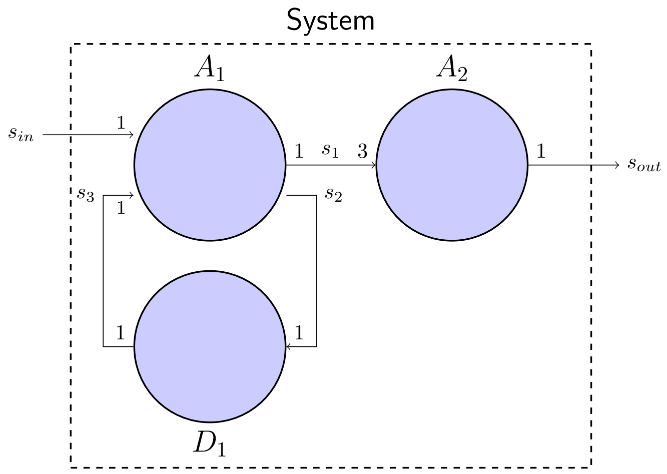


Figure G.2.: Representation of SDF application as concurrent process network

```haskell
module SDF_Intro where

import ForSyDe.Shallow

-- Netlist
system s_in = s_out where
    (s_1, s_2) = a_1 s_in s_3
    s_3        = d_1 s_2
    s_out      = a_2 s_1
```

As a second step, both actors A_1 and A_2, and the delay element D_1 have to be modelled. Actors and delay elements are *processes* in ForSyDe and need to be modelled with process constructors, which are part of the modelling library. The actor A_1 has two input signals and two output signals, thus the process constructor `actor22SDF` has to be used. The actor A_2 has one input signal and one output signal, so the process constructor `actor11SDF` is used. For the delay element D_1 there is the process constructor `delaySDF`. Each process constructor takes a set of arguments. The two process constructors for the SDF actors require the consumption and production rates as additional inputs, while the process constructor for delay element requires a list of values for the initial tokens as argument.

The online documentation for the **ForSyDe-Shallow** library (ForSyDe-Shallow), which is available on Hackage, the Haskell repository (Hackage), gives the following type signatures for the corresponding process constructors.

```
actor11SDF :: Int -> Int -> ([a] -> [b])
           -> Signal a -> Signal b
actor22SDF :: (Int, Int) -> (Int, Int)
           -> ([a] -> [b] -> ([c], [d]))
           -> Signal a -> Signal b -> (Signal c, Signal d)
delaySDF   :: [a] -> Signal a -> Signal a
```

Using this information, the specification for the actors and delay elements can be formulated. Here, to create the code for the actor A_1, the process constructor `actor22SDF` takes the consumption rates of the incoming signals s_{in} and s_3 as tuple `(1,1)`, the production rates of the outgoing signals s_1 and s_2 as tuple `(1,1)`, and the function argument `add` as its arguments. The code for the actor A_2 is created using the process constructor `actor11SDF`, which takes the consumption rate `3` of its incoming signal s_1, the production rate `1` of its outgoing signal s_{out}, and the function argument `average` as its arguments. The functions `add` and `average` have to be specified in the next step. For the process that models the delay element D_1, there is only a single initial token, which has the value 0. Thus, the delay element is created using the process constructor `delaySDF`, which takes the singleton list `[0]` as argument.

```
-- Process specification
a_1 s1 s2 = actor22SDF (1,1) (1,1) add s1 s2
d_1 s     = delaySDF [0] s
a_2 s     = actor11SDF 3 1 average s
```

This more detailed process network is illustrated in Figure G.3.

In the final step, the functions `add` and `average` have to be specified. Since each SDF actor can potentially take a list of tokens, the functions `add` and `average` use lists with

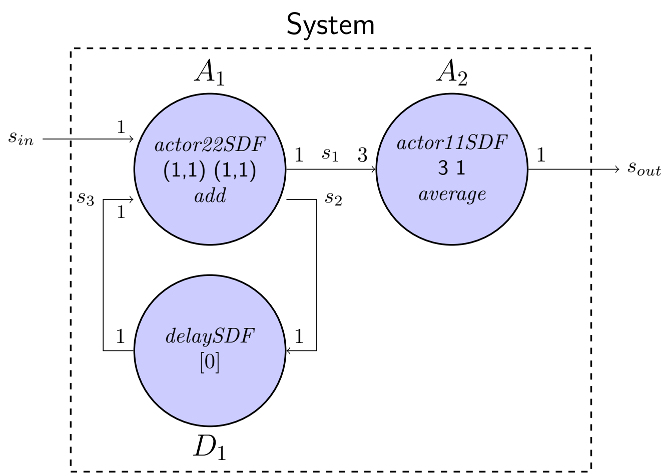


Figure G.3.: ForSyDe process network with process constructors

the required number of tokens for the input and output parameters. The code for the functions is given below.

```
-- Function definition
add [x] [y]         = ([x + y], [x + y])
average [x1,x2,x3] = [(x1 + x2 + x3) / 3.0]
```

Once `ForSyDe-Shallow` is installed, then the model can be simulated. Note, that signals in `ForSyDe-Shallow` are modelled as an own data type, which can be converted from the built-in list data type using the function `signal`.

```
console> ghci SDF_Intro.hs
GHCi, version 8.8.1: https://www.haskell.org/ghc/  :? for help
...
[1 of 1] Compiling SDF_Intro        ( SDF_Intro.hs, interpreted )
Ok, one module loaded.
*SDF_Intro> s_test = signal [1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0]
*SDF_Intro> system s_test
{3.3333333333333335,15.333333333333334,36.333333333333336}
```

The complete executable ForSyDe model of the application is shown in Listing G.1.

G.3. Cyclo-Static Data Flow

The CSDF (Section 5.2.5) library in `ForSyDe.Shallow` defines a set of CSDF process constructors, which together with the SDF process constructors enable to model CSDF applications.

Listing G.1. The tutorial SDF modelled as executable ForSyDe model

```
module SDF_Intro where

import ForSyDe.Shallow

-- Netlist
system s_in = s_out where
    (s_1, s_2) = a_1 s_in s_3
    s_3        = d_1 s_2
    s_out      = a_2 s_1

-- Process specification
a_1 s1 s2 = actor22SDF (1,1) (1,1) add s1 s2
d_1 s     = delaySDF [0] s
a_2 s     = actor11SDF 3 1 average s

-- Function definition
add [x] [y]        = ([x + y], [x + y])
average [x1,x2,x3] = [(x1 + x2 + x3) / 3.0]
```

Figure G.4 shows the CSDF graph for the tutorial application that shall be modelled. The actors A_2 and A_3 are normal SDF actors, which shall implement the *identity* function

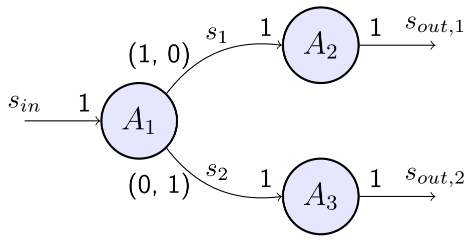


Figure G.4.: CSDF tutorial application

(A_2), and the *negation function* (A_3). The CSDF actor A_1 shall execute a periodic schedule. In the first cycle the input token is passed to actor A_2, while in the second cycle the input token is passed to actor $A3$.

The first step to model this application is to create the netlist as a set of equations.

```
system s_in = (s_out1, s_out2) where
  s_out1     = a_2 s_2
  s_out2     = a_3 s_3
  (s_2, s_3) = a_1 s_in
```

The actors A_2 and A_3 are normal SDF actors and are modelled as described in Appendix G.2.

```
a_2 = actor11SDF 1 1 f_2 where
   f_2 [x] = [x]
a_3 = actor11SDF 1 1 f_3 where
  f_3 [x] = [-x]
```

The CSDF actor A_1 has a different functionality in the different cycles of the period. Each of these *scenarios* is described by

- the number of tokens produced
- the number of tokens consumed
- the function that will be executed

The process constructor `actor12CSDF` takes a list of the scenarios `[sc_1, sc_2]` and returns the CSDF actor.

```
a_1 = actor12CSDF [sc_1, sc_2] where
  sc_1      = (1,(1,0), f_sc_1)
  f_sc_1 [x] = ([x], [])
  sc_2      = (1,(0,1), f_sc2)
  f_sc2 [x] = ([], [x])
```

Here the first scenario is modelled by the tuple `(1,(1,0), f_sc_1)`, where the first term is the consumption rate, the second term is the production rate for both arcs given as a tuple, and the final term is the function `f_sc_1` that will be executed in this scenario.

Simulation of the code gives the following output. In the first step an infinite signal `s_inf` is defined using the `infiniteS` function. Then the signal `s_test` is created using the function `takeS` and finally the application is simulated.

```
*CSDF_Tutorial_Example> s_inf = infiniteS (+1) 0
*CSDF_Tutorial_Example> s_test = takeS 10 s_inf
*CSDF_Tutorial_Example> s_test
{0,1,2,3,4,5,6,7,8,9}
*CSDF_Tutorial_Example> system s_test
({0,2,4,6,8},{-1,-3,-5,-7,-9})
```

The simulation could also have been written as follows using the function application operator (`$`), which enables to chain functions.

```
 *CSDF_Tutorial_Example> system $ takeS 10 $ infiniteS (+1) 0
({0,2,4,6,8},{-1,-3,-5,-7,-9})
```

The full source code is given in Listing G.2.

Listing G.2. The ForSyDe model for the CSDF application in Figure G.4

```haskell
module CSDF_Tutorial_Example where

import ForSyDe.Shallow

-- Netlist
system s_in = (s_out1, s_out2) where
  s_out1      = a_2 s_2
  s_out2      = a_3 s_3
  (s_2, s_3) = a_1 s_in

-- CSDF actor 'a_1' that in each
--    - odd cycle:  outputs value on signal s_1
--    - even cycle: outputs value on signal s_2
a_1 = actor12CSDF [c_1, c_2] where
  c_1      = (1,(1,0), f_c1)
  f_c1 [x] = ([x], [])
  c_2      = (1,(0,1), f_c2)
  f_c2 [x] = ([], [x])

-- SDF actor 'a_2' that implements identity function
a_2 = actor11SDF 1 1 f_2 where
  f_2 [x] = [x]

-- SDF actor 'a_3' that implements negation function
a_3 = actor11SDF 1 1 f_3 where
  f_3 [x] = [-x]
```

G.4. Scenario-Aware Data Flow

The SADF (Section 5.2.6) library of `ForSyDe.Shallow` defines a set of kernel processes and a set of detector process constructors, which together with the SDF process constructors enable to model SADF applications.

The tutorial uses the same specification as the system used in the CSDF tutorial (Appendix G.3, Figure G.4).

Figure G.5 shows the structure of the SADF model. The actors A_1 and A_2 are normal SDF actors, which shall implement the *identity* function (A_1), and the *negation function* (A_2). The SADF kernel K_1 shall execute a periodic schedule. In the first cycle the input token is passed to actor A_1, while in the second cycle the input token is passed to actor

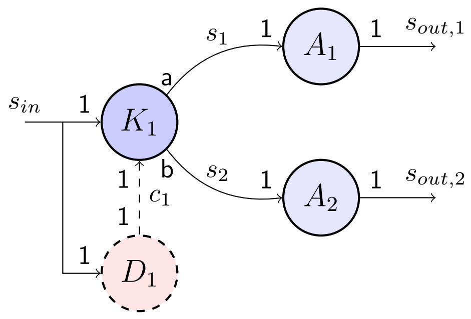


Figure G.5.: SADF tutorial application

A_2. The detector D_1 is responsible to select the different scenarios that the kernel shall run.

As a first step, the netlist of the SADF application needs to be implemented as a set of equations.

```
system s_in = (s_out_1, s_out_2) where
   s_out_1    = a_1 s_1
   s_out_2    = a_2 s_2
   (s_1, s_2) = k_1 c_1 s_in
   c_1        = d_1 s_in
```

The signal c_1 between detector D_1 and the kernel K_1 is a control signal that carries the scenarios that shall be run in the kernel. The actors A_1 and A_2 are normal `actors` and implement the identity function (A_1) and negation function (A_2).

```
a_1 = actor11SDF 1 1 (\[x] -> [x])
a_2 = actor11SDF 1 1 (\[x] -> [-x])
```

Here the functions are given as anonymous functions using lambda notation. The function in actor A_2 can be read as the function takes a list with a single item x as argument and returns a list of with the negated value of x.

The kernel is modelled by be means of `kernel12SADF`, since the kernel has one data input signals (solid arrow) and two data output signals. `kernel12SADF` does not take any arguments and is thus a process that takes one control input signal and one data input signal and produces two data output signals. Thus K_1 is modelled simply as

```
k_1 = kernel12SADF
```

The detector D_1 takes a data input signal s_{in} and returns a control signal c_1. The detector selects the next scenario based on its own state and the value of the current input tokens. The detector is modelled using the process constructor $detector11SADF$ and additional arguments. Depending on the value of the input, the detector returns a tuple that describes the next scenario with a set of arguments. The arguments are

- the consumption rate
- the function that calculates the next state
- the function that selects the next scenario
- the initial state of the detector

```
  d_1 = detector11SADF consume_rate next_state
                       select_scenario initial_state
```

In order to model the detector, it is important to understand how scenarios are modelled. And the modelling of scenarios follows the same way in the case of the CSDF MoC (Appendix G.3). Here the first scenario is modelled by the tuple

```
k_1_scenario_0 = (1,(1, 0), \[x] -> ([x], []))
```

where the first term is the consumption rate, the second term is the production rate for both arcs given as a tuple, and the final term is the anonymous function `\[x] -> ([x], [])` that will be executed in this scenario.

The function that selects the scenarios takes a state as input and returns a tuple, where the first argument is the number of tokens that are produced and the second argument a list of scenarios.

```
select_scenario 0 = (1, [k_1_scenario_0])
select_scenario 1 = (1, [k_1_scenario_1])
```

The detector executes a finite state machine, where transitions depend on the value of the input tokens and the state of the machine. Here, the kernel shall run alternating scenarios, which are only dependent on the state, but independent of the input token. This means that the next state function shall ignore the data input, which can be expressed by '_' and only use the current state to determine the next state.

```
next_state 0 _ = 1
next_state 1 _ = 0
```

The final parts to model the detector are to specify the consumption rate and the initial state.

```
consume_rate = 1
initial_state = 0
```

Simulation of the code gives the following output, which exactly the same as in the CSDF model of Section G.3.

```
 *SADF_Tutorial_Example_3> system $ takeS 10 $ infiniteS (+1) 0
({1,3,5,7,9},{0,-2,-4,-6,-8})
```

It is important to point out that this tutorial example does not show the full power of SADF, also because the next state function did not use the values of the input tokens to determine the next state, which enable to model more dynamic systems. Still, also the modelling of more dynamic systems follows the same principles discussed in this tutorial.

The full source code is given in Listing G.3.

G.5. Hardware Design with `ForSyDe.Deep`

In order to introduce `ForSyDe.Deep` the example of a simple counter that counts from 0 to 59 and displays its state on two seven segment decoders is used. It is the same example that is used in Appendix E to introduce the hardware description language VHDL. Thus, it is easy to compare the two different approaches to hardware design.

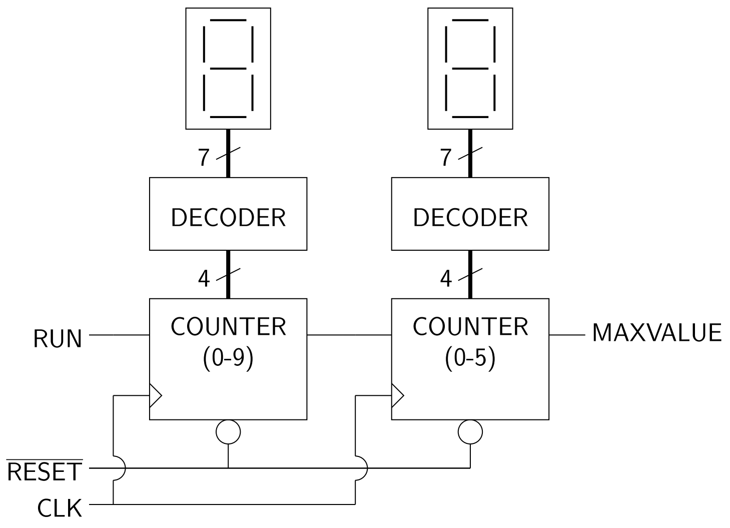


Figure G.6.: A counter that counts between 0 and 59 and displays the number on two seven-segment displays

Figure G.6 shows the schematics of the design. The counter counts from zero to 59 and restarts then counting from 0. The design consists of two counters and two seven-segment decoders.

G.5.1. Installation of `ForSyDe.Deep`

To install `ForSyDe.Deep` and the free version of the Quartus synthesis tool from Intel follow the following instructions.

Listing G.3. The ForSyDe model for the SADF application in Figure G.5

```haskell
module SADF_Tutorial_Example_3 where

import ForSyDe.Shallow

system s_in = (s_out_1, s_out_2) where
   s_out_1    = a_1 s_1
   s_out_2    = a_2 s_2
   (s_1, s_2) = k_1 c_1 s_in
   c_1        = d_1 s_in

-- SDF Actors 'a_1' and 'a_2'
a_1 = actor11SDF 1 1 (\[x] -> [x])
a_2 = actor11SDF 1 1 (\[x] -> [-x])

-- Kernel 'k_1'
k_1 = kernel12SADF

-- Detector 'd_1'
--    - starts in state 0
--    - alternates between state 0 and 1
--    - ignores input data value
--    - executes scenario 0 in state 0
--    - executes scenario 1 in state 1
d_1 = detector11SADF consume_rate next_state
                     select_scenario initial_state where
  consume_rate = 1
  -- Next State Function 'next_state' ignores input value
  next_state 0 _ = 1
  next_state 1 _ = 0
  -- Definition of scenarios
  --   - Scenario 0: Send token
  k_1_scenario_0 = (1,(1, 0), \[x] -> ([x], []))
   -- - Scenario 1: Ignore token
  k_1_scenario_1 = (1,(0, 1), \[x] -> ([], [x]))
  -- Function for Selection of scenarios
  select_scenario 0 = (1, [k_1_scenario_0])
  select_scenario 1 = (1, [k_1_scenario_1])
  -- Initial State
  initial_state = 0
```

1. Download and install the Haskell project and package manager tool `stack` from www.haskellstack.org. Follow the instructions on the `stack` web page.

2. ForSyDe.Deep is available at https://github.com. However, for the tutorial, install the forsyde-deep-examples, which will also install ForSyDe.Deep.

   Clone the repository forsyde-deep-examples on https://github.com/forsyde/forsyde-deep-examples with

   - (clone with ssh)

     ```
     git clone git@github.com:forsyde/forsyde-deep-examples.git
     ```

   - (clone with https)

     ```
     git clone https://github.com/forsyde/forsyde-deep-examples.git
     ```

3. Enter the repository and install it using stack.

   ```
   cd forsyde-deep-examples
   ```

   and

   ```
   stack install
   ```

   The installation will install both a suitable version ghc of the Glasgow Haskell Compiler, Forsyde.Deep and examples for ForSyDe.Deep. The installation takes quite some time.

4. The examples are located in the folder examples. To validate your installation, load the design of an 2-bit AND-gate with

   ```
   stack repl And2.hs
   ```

   and check, if it can be simulated by

   ```
   *AND2> simulate and2Sys [L,L,H,H] [L,H,L,H]
   [L,L,L,H]
   ```

   NOTE: Always run the command stack repl or stack ghci to load ForSyDe.Deep examples. Also put your own examples into the examples folder and run them from this folder.

5. Download and install Quartus Prime Lite, which is the free version of Quartus Prime from the Intel FPGA web site at https://fpgasoftware.intel.com/.

6. Set the necessary environment variables for Quartus Prime Lite, so that Quartus and Modelsim can be executed from the command line. This can be done from the installation directory of Quartus Prime with the command

   ```
   nios2eds/nios2_command_shell.sh
   ```

   NOTE: Quartus Prime Lite and Modelsim formally not supported by several Linux distributions. Thus, a few tweaks might be required, for which solutions are documented by various sources on the web. In case you encounter a serious problem, please post your problems in the discussion forum.

G.5.2. Combinational Processes

The modelling of a combinational process is illustrated by means of a seven-segment decoder. This example is availbale as example `Counter_59` from `forsyde-deep-examples`.

The first step is to create a module for the seven-segment decoder component and to integrate the necessary libraries, which are required for the design. The following code includes the Template Haskell extension, which enables to access the abstract syntax tree of the design. This means that the full structure of the design is known, and different commands can be used to operate on this structure. This example will demonstrate the following backends: a) simulation within the Haskell interpreter, b) simulation of the synthesised design with the Modelsim VHDL-simulator, and c) synthesis to an Altera FPGA using the Quartus synthesis tool, The `import` statements include the necessary data types for typical synthesisable designs, the `Bit` data type with the values `L` (low) and `H` (high), integer data types for 8-, 16-, 32- and 64-bit integers, and fixed-sized vectors, which enable to specify vectors of a fixed size.

```haskell
{-# LANGUAGE TemplateHaskell #-}

module SevenSegmentDecoder (sevenSegDecSys) where

import ForSyDe.Deep
import ForSyDe.Deep.Bit
import Data.Param.FSVec
import Data.Int
import Data.TypeLevel.Num.Reps
import Data.TypeLevel.Num.Aliases
```

As in `ForSyDe.Shallow`, also in deep-embedded ForSyDe processes are created by means of *process constructors* taking functions and values as arguments. However, `ForSyDe.Deep` defines an additional level, which is called *system definition* and creates a reusable synthesisable component.

In the deep-embedded world the functions that are arguments of the process constructor are called *process functions* and have a different data type `ProcFun`. Since access to the internal representation of the function is required, Template Haskell statements are used for the definition of the function.

The main difference to the shallow-embedded version is the special syntax. First, the function has the data type `ProcFun (Int8 -> FSVec D7 Bit)` instead of `Int8 -> FSVec D7 Bit` which would be used in `ForSyDe.Shallow`. Then the computation function `decode` is declared inside a code pattern that allows to have access to the internal structure of the model, the abstract syntax tree. This code pattern is `$(newProcFun [d| ... |])`. Inside this pattern the function is declared in the same way as before. In contrast to the shallow-embedded version of ForSyDe, the type for the function, here `decode :: Int8 -> FSVec` needs to be provided.

```
decodeFun :: ProcFun (Int8 -> FSVec D7 Bit)
decodeFun
   = $(newProcFun [d|decode :: Int8 -> FSVec D7 Bit
                       decode x
                         = case x of
                             0 -> H +> L +> L +> L +> L +> L +> L +> empty
                             1 -> H +> H +> H +> H +> L +> L +> H +> empty
                             2 -> L +> H +> L +> L +> H +> L +> L +> empty
                             3 -> L +> H +> H +> L +> L +> L +> L +> empty
                             4 -> L +> L +> H +> H +> L +> L +> H +> empty
                             5 -> L +> L +> H +> L +> L +> H +> L +> empty
                             6 -> L +> L +> L +> L +> L +> H +> L +> empty
                             7 -> H +> H +> H +> H +> L +> L +> L +> empty
                             8 -> L +> L +> L +> L +> L +> L +> L +> empty
                             9 -> L +> L +> H +> L +> L +> L +> L +> empty
                             _ -> H +> H +> H +> H +> H +> H +> H +> empty
                               |])
```

The function `decode` uses the data type `Bit` with the values `L` and `H`, which can be synthesised in the deep-embedded version of ForSyDe. Even more important is the use of the fixed-sized vector data type `FSVec` instead of a list or vector data type. For hardware synthesis it is of essential importance to give the exact size of a vector, which is possible with `FSVec`. Here `D7` gives the size of the fixed-sized vector, which in this case is a vector of seven bits. So far, `ForSyDe.Deep` does not support a more elegant way to define a fixed-sized vector, so the notation `H +> L +> L +> L +> L +> L +> L +> empty` is used to describe a bit vector with seven elements and the value `[H,L,L,L,L,L,L]`, which is the seven-segment pattern for the number 0. Here, `L` turns the segment on, and `H` turns it off.

Although the code might look quite verbose and not so intuitive at first sight, it fully follows the `ForSyDe.Shallow` philosophy and just requires to follow the template for `ForSyDe.Deep`.

The next level declares the process using process constructor and process function. In contrast to the shallow-embedded version an additional label needs to be provided to distinguish different processes.

```
sevenSegDecProc :: Signal Int8 -> Signal (FSVec D7 Bit)
sevenSegDecProc = mapSY "decode" decodeFun
```

Finally the system definition is created. The system is defined providing the process, a label, a list of input port names, and a list of output port names.

```
sevenSegDecSys :: SysDef (Signal Int8 -> Signal (FSVec D7 Bit))
sevenSegDecSys = newSysDef sevenSegDecProc "sevenSegDec" ["in"] ["out"]
```

The deep-embedded ForSyDe compiler makes use of the knowledge of the internal structure of a system by providing different backends that operate on the internal structure of the system. The first backend is the *simulation backend*.

After loading the design with the command

```
Counter_59> stack ghci SevenSegmentDecoder.hs
...
Ok, modules loaded: SevenSegmentDecoder.
Loaded GHCi configuration from /tmp/...
```

The seven segment decoder can be simulated using the following command:

```
*SevenSegmentDecoder> simulate sevenSegDecSys [4,0,2]
[<L,L,H,H,L,L,H>,<H,L,L,L,L,L,L>,<L,H,L,L,H,L,L>]
```

Observe that the deep-embedded compiler uses plain Haskell lists for signals.

Of special interest is the second backend that generates VHDL-code for a system.

```
*SevenSegmentDecoder> writeVHDL sevenSegDecSys
```

This generates the VHDL-code for the seven segment decoder. The VHDL-code is located in the directory `sevenSegDec/vhdl`. Further the Quartus tool with design options can be started. The function `writeVHDLOps` enables to give options, which not only start the Quartus tool, but also give constraints and pin assignments important for synthesis. In the is case the target platform is the Altera DE 10 Standard board, but naturally other Altera boards can be targeted as well.

```
generateHW_DE_10_Standard :: IO ()
generateHW_DE_10_Standard = writeVHDLOps vhdlOps sevenSegDecSys
 where vhdlOps = defaultVHDLOps{execQuartus=Just quartusOps}
       quartusOps
         = QuartusOps{action=FullCompilation,
                      fMax=Just 24, -- in MHz
                      fpgaFamiliyDevice=Just ("Cyclone V",
                                     Just "5CSXFC6D6F31C6"),
                      -- Possibility for Pin Assignments
                      pinAssigs=[("in[0]", "PIN_AB30"),  -- SW0
                                 ("in[1]", "PIN_Y27"),   -- SW1
                                 ("in[2]", "PIN_AB28"),  -- SW2
                                 ("in[3]", "PIN_AC30"),  -- SW3
                                 ("in[4]", "PIN_W25"),   -- SW4
                                 ("in[5]", "PIN_V25"),   -- SW5
                                 ("in[6]", "PIN_AC28"),  -- SW6
                                 ("in[7]", "PIN_AD30"),  -- SW7
                                 ("out[6]","PIN_W17"),   -- HEX0[0]
                                 ("out[5]","PIN_V18"),   -- HEX0[1]
                                 ("out[4]","PIN_AG17"),  -- HEX0[2]
                                 ("out[3]","PIN_AG16"),  -- HEX0[3]
                                 ("out[2]","PIN_AH17"),  -- HEX0[4]
                                 ("out[1]","PIN_AG18"),  -- HEX0[5]
                                 ("out[0]","PIN_AH18")   -- HEX0[6]
                                ]
                              }
```

If Quartus is installed on the computer and the environment variables for Quartus are set, the following command starts the synthesis within the Haskell interpreter.

```
*SevenSegmentDecoder> generateHW_DE_10_Standard
Running quartus_sh -t quartus.tcl
...
Info: Quartus Prime Shell was successful. 0 errors, 16 warnings
    Info: Peak virtual memory: 812 megabytes
    Info: Processing ended: Wed May 27 11:39:36 2020
    Info: Elapsed time: 00:00:51
    Info: Total CPU time (on all processors): 00:01:13
```

Finally, the generated netlist can be downloaded to the Altera DE 10 board by the following command after leaving the Haskell interpreter.

```
  Counter_59> quartus_pgm -c DE-SoC -m JTAG
               -o "p;./sevenSegDec/vhdl/sevenSegDec.sof@2"
Info: *******************************************************************
Info: Running Quartus Prime Programmer
...
Info: Quartus Prime Programmer was successful. 0 errors, 0 warnings
    Info: Peak virtual memory: 489 megabytes
    Info: Processing ended: Wed May 27 11:40:29 2020
    Info: Elapsed time: 00:00:04
    Info: Total CPU time (on all processors): 00:00:01
```

Now, the switches can be used to select the number that shall be shown on the seven segment display.

If the VHDL simulator Modelsim is installed on the system, the design can also be simulated with Modelsim within the Haskell interpreter. The command writeAndModelsimVHDL writes VHDL and starts the Modelsim simulator.

```
writeAndModelsimVHDL Nothing sevenSegDecSys [0,2,4]
*SevenSegmentDecoder> writeAndModelsimVHDL Nothing sevenSegDecSys [0,2,4]
Running: vmap forsyde /home/...
...
# End time: 11:52:10 on May 27,2020, Elapsed time: 0:00:00
# Errors: 0, Warnings: 1
[<H,L,L,L,L,L,L>,<L,H,L,L,H,L,L>,<L,L,H,H,L,L,H>]
```

Again, input and output signals are represented as lists.

G.5.3. Sequential Processes

The complete design has two counters. Both counters have an input run, which enables counting, upwards, if the input is H. Otherwise, the counter will keep its value. One of the counters will count to from 0 to 9 , the other one from 0 to 5. The counters have two outputs, the current number and an output that has the value H (high), when the maximum value has been reached. The following listing shows the code for the counter, which counts from 0 to 9.

```
{-# LANGUAGE TemplateHaskell #-}

module Counter_9 (counter_9_Sys) where

import ForSyDe.Deep
import Data.Int


-- Input: Run
-- 'H' : Count
-- 'L' : Hold
type Run = Bit

nextStateFun :: ProcFun (Int8 -> Run -> Int8)
nextStateFun = $(newProcFun
   [d| nextState state dir = if dir == H then
                                if state < 9 then
                                   state + 1
                                else
                                   0
                             else
                               state
     |])

```

```haskell
outputFun :: ProcFun (Int8 -> (Run, Int8))
outputFun = $(newProcFun
   [d| output state = if state == 9 then
                         (H, state)
                       else
                         (L, state)
     |])


counterProc :: Signal Run -> Signal (Run, Int8)
counterProc = mooreSY "counterProc" nextStateFun outputFun 0


counter_9_Sys :: SysDef (Signal Run -> Signal (Run, Int8))
counter_9_Sys = newSysDef counterProc "Counter_9"
                          ["direction"] ["run", "number"]
```

The structure of the counter follows the `ForSyDe.Deep` structure, which has been described in Section G.5.2. The counter is modelled by means of the sequential process constructor *mooreSY*, which takes one function to calculate the next state, another function to calculate the output, and the value 0 to specify the initial state. The system specification `counter_9_Sys` can be simulated in Haskell or Modelsim and translated to VHDL using the functions described in Section G.5.2.

G.5.4. Process Networks

Finally, instances of the seven segment decoder and the counters can be used to create a new synthesisable design that counts up- and downwards and shows the actual state on the seven-segment displays.

```haskell
systemProc :: Signal Run -> (Signal Bit, Signal (FSVec D7 Bit),
                             Signal (FSVec D7 Bit))
systemProc run = (run_5, sevenSeg_5, sevenSeg_9)
   where
     sevenSeg_9
       = (instantiate "sevenSegDec_9" sevenSegDecSys) counter_9_Out
     sevenSeg_5
       = (instantiate "sevenSegDec_5" sevenSegDecSys) counter_5_Out
     (run_5, counter_5_Out)
       = (unzipSY "unzip_5" . instantiate "counter_5" counter_5_Sys)
           run_9
     (run_9, counter_9_Out)
       = (unzipSY "unzip_9" . instantiate "counter_9" counter_9_Sys)
           run


system :: SysDef (Signal Run -> (Signal Bit, Signal (FSVec D7 Bit),
                                 Signal (FSVec D7 Bit)))
```

```haskell
system = newSysDef systemProc "system" ["run"]
                              ["run_5", "sevenseg_5", "sevenseg_9"]
```

Here we see that a composite process systemProc is created by describing a “netlist” as a set of equations. In this set of equations the systems definitions of the seven segment decoders `sevenSegDecSys` and the counters `counter_5_Sys` and `counter_9_Sys` are instantiated. Finally, the new system `counter_59_Sys` is created by a system definition using the process `systemProc`.

The fully synthesisable code for the system is given below including parameters for the Quartus tool targeting the DE 10 standard board.

```haskell
{-# LANGUAGE TemplateHaskell #-}

module Counter_59 where

import ForSyDe.Deep
import Counter_5
import Counter_9
import SevenSegmentDecoder
import ForSyDe.Deep.Bit
import Data.Param.FSVec
import Data.Int
import Data.TypeLevel.Num.Reps
import Data.TypeLevel.Num.Aliases

type Run = Bit

systemProc :: Signal Run -> (Signal Bit, Signal (FSVec D7 Bit),
                             Signal (FSVec D7 Bit))
systemProc run = (run_5, sevenSeg_5, sevenSeg_9)
   where
     sevenSeg_9
       = (instantiate "sevenSegDec_9" sevenSegDecSys) counter_9_Out
     sevenSeg_5
       = (instantiate "sevenSegDec_5" sevenSegDecSys) counter_5_Out
     (run_5, counter_5_Out)
       = (unzipSY "unzip_5" . instantiate "counter_5" counter_5_Sys)
         run_9
     (run_9, counter_9_Out)
       = (unzipSY "unzip_9" . instantiate "counter_9" counter_9_Sys)
         run


counter_59_Sys :: SysDef (Signal Run -> (Signal Bit, Signal (FSVec D7 Bit),
                                         Signal (FSVec D7 Bit)))
counter_59_Sys = newSysDef systemProc "counter_59" ["run"]
                              ["maxvalue", "sevenseg_5", "sevenseg_9"]

-- Hardware Generation
generateHW_DE_10_Standard :: IO ()
```

```
generateHW_DE_10_Standard = writeVHDLOps vhdlOps counter_59_Sys
 where vhdlOps = defaultVHDLOps{execQuartus=Just quartusOps}
       quartusOps
          = QuartusOps{action=FullCompilation,
                       fMax=Just 50, -- in MHz
                       fpgaFamiliyDevice=Just ("Cyclone V",
                                               Just "5CSXFC6D6F31C6"),
                       -- Possibility for Pin Assignments
                       pinAssigs
                         = [("run", "PIN_AB30"),  -- SW0
                            ("resetn", "PIN_Y27"),  -- SW1
                            ("clock","PIN_AJ4"),    -- KEY[0]
                            ("sevenseg_9[6]","PIN_W17"),   -- HEX0[0]
                            ("sevenseg_9[5]","PIN_V18"),   -- HEX0[1]
                            ("sevenseg_9[4]","PIN_AG17"),  -- HEX0[2]
                            ("sevenseg_9[3]","PIN_AG16"),  -- HEX0[3]
                            ("sevenseg_9[2]","PIN_AH17"),  -- HEX0[4]
                            ("sevenseg_9[1]","PIN_AG18"),  -- HEX0[5]
                            ("sevenseg_9[0]","PIN_AH18"),  -- HEX0[6]
                            ("sevenseg_5[6]","PIN_AF16"), -- HEX1[0]
                            ("sevenseg_5[5]","PIN_V16"), -- HEX1[1]
                            ("sevenseg_5[4]","PIN_AE16"), -- HEX1[2]
                            ("sevenseg_5[3]","PIN_AD17"), -- HEX1[3]
                            ("sevenseg_5[2]","PIN_AE18"), -- HEX1[4]
                            ("sevenseg_5[1]","PIN_AE17"),  -- HEX1[5]
                            ("sevenseg_5[0]","PIN_V17"),   -- HEX1[6]
                            ("maxvalue", "PIN_AA24") -- LEDR[0]
                           ]
                       }
```

Terms and Abbreviations

ALU Arithmetic Logic Unit. 20, 101, 103

ASIC Application Specific Integrated Circuit. 18, 94

BDF Boolean Data Flow. 131, 132

CDFG Control and Data Flow Graph. 104, 105, 107, 114

CISC Complex Instruction Set Computer. 20

CPLD Complex Programmable Logic Device. 94

CPS Cyber-Physical Systems. 9, 10

CPU Central Processing Unit. 8, 25, 26, 28, 41, 46–48, 53–55, 114, 177, 178, 234

CSDF Cyclo-Static Data Flow. 133, 135, 276–279, 281, 282

DeSyDe Design Space Exploration for System Design. 163–165

DFG Data Flow Graph. 104, 109, 110

DMA Direct Memory Access. 46, 47, 91

DRAM Dynamic Random Access Memory. 8, 25, 26

DSE Design Space Exploration. 89, 159–161, 163–165, 177

DSP Digital Signal Processor. 17, 33, 94–97, 100–103, 108

EDF Earliest Deadline First. 67, 68, 256

EEPROM Electrically Erasable Programmable Read Only Memory. 24

EPROM Erasable Programmable Read Only Memory. 24

FIFO First In First Out. 43, 174, 177, 178

FIR Finite Impulse Response. 100, 103

ForSyDe Formal System Design. 135, 154–158, 163, 166–168, 170–173, 175, 176, 273–277, 279, 283, 286

FPGA Field Programmable Gate Array. 17, 18, 89, 94

FSM Finite State Machine. 119, 134, 136, 146, 148, 149

HAL Hardware Abstraction Layer. 116

HSDF Homogeneous Synchronous Data Flow. 130

HSDFG Homogeneous Synchronous Data Flow Graph. 130, 131

I/O Input/Output. 8, 34–37, 39, 46, 57, 91

IRQ Interrupt Request. 47

ISS Instruction Set Simulator. 89

LED Light Emitting Diode. 46

MoC Model of Computation. 122, 123, 125, 135, 150–157, 163, 166–170, 177, 273, 274, 281

NoC Network on Chip. 38–40, 163

NPP Non-Preemptive Protocol. 79, 81

OMG Object Management Group. 119

OS Operating System. 48, 52, 126, 136

PAPS Periodic Admissable Parallel Schedule. 126, 129, 130

PASS Periodic Admissable Sequential Schedule. 126–129

PE Processing Element. 92

PROM Programmable Read Only Memory. 24

PSS Periodic Sequential Schedule. 128, 129

QoS Quality of Service. 46

RISC Reduced Instruction Set Computer. 20, 21

RM Rate-Monotonic. 65–69, 254

RMA Rate-Monotonic Algorithm. 70

RMS Rate-Monotonic Schedule. 256

RTOS Real-Time Operating System. 8, 38, 47, 48, 51–53, 55, 58, 63, 65, 68, 126, 136, 173, 177

SADF Scenario-Aware Data Flow. 133–135, 279, 280, 282, 283

SDF Synchronous Data Flow. 126–135, 137, 153, 163, 168, 172–174, 177, 273–277, 279

SDFG Synchronous Data Flow Graph. 125–127, 130–132, 164, 172, 173, 273

SRAM Static Random Access Memory. 8, 24–26

TCB Task Control Block. 53, 54

TDM Time-Division Multiplex. 45, 46, 163, 164

UML Unified Modelling Language. 119, 122

WCCT Worst Case Communication Time. 163, 164

WCET Worst Case Execution Time. 33, 34, 64, 112, 114, 115, 162, 163

Bibliography

Ada-2012-AARM. Annotated ada 2012 language reference manual (aarm). URL http://www.ada-auth.org/standards/12aarm/html/AA-TTL.html.

Ada-2012-LRM. Ada 2012 language reference manual (lrm). URL http://www.ada-auth.org/standards/12rm/html/RM-TTL.html.

Ada-2012-Rationale. Ada 2012 rationale. URL http://www.ada-auth.org/standards/rationale12.html.

Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers, 2002.

Seyed-Hosein Attarzadeh-Niaki and Ingo Sander. An extensible modeling methodology for embedded and cyber-physical system design. *SIMULATION: Transactions of The Society for Modeling and Simulation International*, 92(8):771–794, 2016. doi: 10.1177/0037549716659753.

S.H. Attarzadeh Niaki, M.K. Jakobsen, T. Sulonen, and I. Sander. Formal heterogeneous system modeling with SystemC. In *Forum on Specification and Design Languages (FDL 2012)*, pages 160–167, Vienna, Austria, 2012.

John Barnes. *Programming in Ada 2012*. Cambridge University Press, 2014.

Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming*. Addison-Wesley, 2006.

Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

Gérard Berry. The constructive semantics of pure Esterel. Draft Version 3, 1999.

Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cyclo-static data flow. *IEEE Transactions on Signal Processing*, 44(2):397–408, February 1996.

Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2014.

Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys*, 38(1):1–es, June 2006. ISSN 0360-0300. doi: 10.1145/1132952.1132953. URL https://doi-org.focus.lib.kth.se/10.1145/1132952.1132953.

Frédéric Boussinot and Robert De Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

Joseph. T. Buck and Edward A. Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages 429–432 vol.1, April 1993. doi: 10.1109/ICASSP.1993.319147.

Alan Burns and Andy Wellings. *Concurrent and Real-Time Programming in Ada.* Cambridge University Press, 2007.

Alan Burns and Andy Wellings. *Analysable Real-Time Systems: Programmed in Ada.* Create Space Independent Publishing Platform, 2016.

Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Springer, 3rd edition, 2011.

Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. Hardware-software codesign of embedded systems. *IEEE Micro*, pages 26–36, August 1994.

Michael D. Ciletti. *Advanced Digital Design with the Verilog HDL.* Prentice Hall, 2003.

Stephen Edwards, Luciano Lavagno, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, March 1997.

Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

Rolf Ernst, Jörg Henkel, and Thomas Brenner. Hardware-software cosynthesis from microcontrollers. *IEEE Design & Test of Computers*, 10(4):64–75, December 1993.

ForSyDe. ForSyDe: A methodology for Formal System Design. URL https://forsyde.github.io/.

ForSyDe-Shallow. forsyde-shallow: ForSyDe's Haskell-embedded domain specific language. http://hackage.haskell.org/package/forsyde-shallow/.

GCC-GNU-Compiler. GCC, the GNU compiler collection. URL https://gcc.gnu.org/.

GNAT-User-Guide. Gnat user's guide for native platforms. URL http://docs.adacore.com/live/wave/gnat_ugn/html/gnat_ugn/gnat_ugn.html.

M. Gordon, R. Milner, L. Morris, M. Newey, and C. P. Wadsworth. A metalanguage for interactive proof in LCF. In *Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages*, pages 119–130. ACM, 1978.

Rajesh Kumar Gupta and Giovanni De Micheli. Hardware-software cosynthesis for digital systems. *IEEE Design & Test of Computers*, 10(3):29–41, September 1993.

Hackage. Hackage: The Haskell package repository. http://hackage.haskell.org/.

Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems*, 2009. ISSN 1084-4309.

D. Harel and A. Pnueli. *Proc. NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems*, volume 13 of *NATO ASI Series F.*, chapter On the development of reactive systems: logic and models of concurrent systems, pages 477–498. Springer-Verlag, New York, 1985.

David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

Øystein Haugen, Birger Møller-Pedersen, and Thomas Weigert. *Embedded Systems Handbook*, chapter Introduction to UML and the Modeling of Embedded Systems. 2005.

Fernando Herrera and Ingo Sander. Combining analytical and simulation-based design space exploration for time-critical systems. In *Forum on Specification and Design Languages (FDL 2013)*, Paris, France, September 2013.

Fernando Herrera, Kathrin Rosvall, Ingo Sander, Edoardo Paone, and Gianluca Palermo. An efficient joint analytical and simulation-based design space exploration flow for predictable multi-core systems. In *Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO)*, Amsterdam, The Netherlands, January 2015. doi: 10.1145/2693433.2693435.

Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

Paul Hudak and Mark Jones. Haskell vs Ada vs C++ vs Awk vs . . . : An experiment in software prototyping productivity. Technical report, Dept. of Computer Science, Yale University, July 1994.

Paul Hudak, John Peterson, and Joseph H. Fasel. *A Gentle Introduction to Haskell 98*. October 1999. http://www.haskell.org/tutorial.

J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

Erwan Jahier, Pascal Raymond, and Nicolas Halbwachs. The Lustre V6 reference manual. Software Version: (03-03-20).

Kurt Keutzer, Sharad Malik, A. Richard Newton, Jan M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonolization of concerns and platform-based design. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, December 2000.

Bart Kienhuis, Ed F. Deprettere, Pieter van der Wolf, and Kees A. Vissers. A methodology to design programmable embedded systems - the y-chart approach. In *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS*, page 18–37, Berlin, Heidelberg, 2002. Springer-Verlag. ISBN 3540433228.

Phil Lapsley, Jeff Bier, Amit Shoham, and Edward A. Lee. *DSP Processor Fundamentals*. IEEE Press, New Yok, USA, 1997.

Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Marie. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1335, September 1991.

Edward A. Lee. Consistency in dataflow graphs. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):223–235, April 1991. ISSN 2161-9883. doi: 10.1109/71.89067.

Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, January 1987a.

Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987b.

Edward A. Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*. MIT Press, 2 edition, 2017.

Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

Jacques-Louis Lions. Ariane 5, flight 501 failure, report by the inquiry board. Technical report, European Space Agency (ESA), July 1996.

Miran Lipovača. *Learn You a Haskell for a Great Good!* 2011. available online: http://learnyouahaskell.com/.

C.L. Liu and James. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–71, January 1973.

Isaac Liu, Jan Reineke, David Broman, Mike Zimmer, and Edward A. Lee. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *International Conference on Computer Design*, 2012.

Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

Lustre V6 Tools. Development tools for critical reactive systems using the synchronous approach – the verimag reactive tool box. http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/reactive-toolbox/.

R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 1978.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. MIT Press, 1997.

Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989. doi: 10.1109/5.24143.

David A. Patterson and John L. Hennessy. *Computer Organization and Design - The Hardware/Software Interface*. Morgan Kaufman, third edition, 2005.

James L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.

A. D. Pimentel. Exploring exploration: A tutorial introduction to embedded systems design space exploration. *IEEE Design Test*, 34(1):77–90, February 2017. ISSN 2168-2356. doi: 10.1109/MDAT.2016.2626445.

Ptolemy. The Ptolemy project. https://ptolemy.berkeley.edu/.

Kathrin Rosvall and Ingo Sander. A constraint-based design space exploration framework for real-time applications on MPSoCs. In *Design Automation and Test in Europe (DATE '14)*, Dresden, Germany, March 2014. doi: 10.7873/DATE.2014.339.

Kathrin Rosvall and Ingo Sander. Flexible and trade-off-aware constraint-based design space exploration for streaming applications on heterogeneous platforms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 23(2), January 2018. doi: 10.1145/3133210.

Kathrin Rosvall, Tage Mohammadat, George Ungureanu, Johnny Öberg, and Ingo Sander. Exploring power and throughput for dataflow applications on predictable NoC multiprocessors. In *Euromicro Conference on Digital System Design (DSD 2018)*, Prague, Czech Republic, August 2018.

Ingo Sander. *System Modeling and Design Refinement in ForSyDe*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, April 2003. URL http://web.it.kth.se/~ingo/Papers/Thesis_Sander_2003.pdf.

Ingo Sander, Axel Jantsch, and Seyed-Hosein Attarzadeh-Niaki. ForSyDe: System design using a functional language and models of computation. In Soonhoi Ha and Jürgen Teich, editors, *Handbook of Hardware/Software Codesign*, pages 99–140. Springer Netherlands, 2017.

Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, 2nd edition, 2009.

TCRTS-Definitions. Real-time systems: Terminology and notations. URL http://sites.ieee.org/tcrts/education/terminology-and-notation/.

TCRTS-Seminal. Real-time systems: Seminal papers. URL https://site.ieee.org/tcrts/education/seminal-papers/.

Jürgen. Teich. Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1411–1430, May 2012. ISSN 0018-9219. doi: 10.1109/JPROC.2011.2182009.

B.D. Theelen, M.C.W. Geilen, T. Basten, J.P.M. Voeten, S.V. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Formal Methods and Models for Co-Design*, 2006. doi: 10.1109/MEMCOD.2006.1695924.

Mark Thompson. *Tools and techniques for efficient system-level design space exploration*. PhD thesis, University of Amsterdam, January 2012.

George Ungureanu and Ingo Sander. A layered formal framework for modeling of cyber-physical systems. In *Design Automation and Test in Europe (DATE 2017)*, Lausanne, Switzerland, March 2017.

George Ungureanu, Timmy Sundström, Anders Åhlander, Ingo Sander, and Ingemar Söderquist. Formal design, co-simulation and validation of a radar signal processing system. In *Forum on Specification and Design Languages (FDL 2019)*, Southamption, United Kingdom, September 2019.

Christopher B. Watkins. Integrated modular avionics: Managing the allocation of shared intersystem resources. In *Digital Avionics Systems Conference*, pages 1–12, October 2006.

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, 2008. ISSN 1539-9087. doi: http://doi.acm.org/10.1145/1347375.1347389.

Glynn Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.

Wayne Wolf. *Computers as Components: Principles of Embedded Computing System Design.* Morgan Kaufman Publishers, San Francisco, California, USA, 2001.

Wayne H. Wolf. Hardware-software co-design of embedded systems. *Proceedings of the IEEE*, 82(7):967–989, July 1994.