Chapter 2
Basics

Abstract This chapter presents the basic components of the MPSoC hardware and software architecture. The MPSoC hardware architecture is made of several interconnected hardware and software subsystems. Each software subsystem executes a specific software stack. The software stack has a layered organization composed of application tasks, operating system, communication, and hardware abstraction layer. This chapter gives the definition of these different hardware and software components of MPSoC.

2.1 The MPSoC Architecture

System–on–Chip (SoC) represents the integration of different computing elements and/or other electronic subsystems into a single integrated circuit (chip). It may contain digital, analog, mixed-signal, and often radio-frequency functions – all on one chip.

Multi-processor System–on–Chip (MPSoC) are SoC that may contain one or more types of computing subsystems, memories, input/output devices (I/O), and other peripherals. These systems range from portable devices such as MP3 players, videogame consoles, digital cameras, or mobile phones to large stationary installations like traffic lights, factory controllers, engine controllers for automobiles, or digital set-top boxes.

The MPSoC architecture is made of three types of components: software subsystems, hardware subsystems, and inter-subsystem communication, as illustrated in Fig. 2.1.

The hardware subsystems (HW-SS) represent custom hardware subsystems that implement specific functionality of an application or global memory subsystems. The HW-SS contain two types of components: intra-subsystem communication and specific hardware components. The hardware components implement specific functions of the target application or represent global memories accessible by the computing subsystems. The intra-subsystem communication represents the communication inside the HW-SS between the different hardware components. This can
be in form of a small bus (collection of parallel wires for transmitting address, data, and control signals) or point-to-point communication links.

The *software subsystems* (SW-SS) represent programmable subsystems, also called processor nodes of the architecture. The SW-SS include computing resources, intra-subsystem communication, and other hardware components, such as local memories, I/O components, or hardware accelerators. The computing resources represent the processing units or CPUs. The *CPU* (central processing unit) also known as processor core, processing element, or shortly processor executes programs stored in the memory by fetching their instructions, examining them, and then executing them one after another [150]. There are two types of SW-SS: single core and multi-core. The single-core SW-SS includes a single processor, while the multi-core SW-SS can integrate several processor cores in the same subsystem, usually of same type. The intra-subsystem communication represents the communication inside the SW-SS, e.g., local bus, hardware FIFO, point-to-point communication links, or other local interconnection network used to interconnect the different hardware components inside the SW-SS.

The *inter-subsystem communication* represents the communication architecture between the different software and hardware subsystems. This can be a hardware FIFO connecting multiple subsystems or a scalable global interconnection network, such as bus or network on chip (NoC). Despite most of the buses, the NoC allows simultaneous data transfers, being composed of several links and switches that provide means to route the information from the source node to the destination node [42].

*Homogeneous MPSoC* architectures are made of identical software subsystems incorporating the same type of processors. In the *heterogeneous MPSoC* architectures, different types of processors are integrated on the same chip, resulting in different types of software subsystems. These can be GPP (general-purpose processor) subsystems for control operations of the application; DSP (digital signal processor) subsystems specially tailored for data-intensive applications such as signal processing applications; or ASIP (application-specific instruction set processor) subsystems with a configurable instruction set to fit specific functions of the application.

The different subsystems working in parallel on different parts of the same application must communicate each other to exchange information. There are two distinct MPSoC designs that have been proposed and implemented for the communication models between the subsystems: shared memory and message passing [42].
The *shared memory* communication model characterizes the homogeneous MPSoC architecture. The key property of this class is that communication occurs implicitly. The communication between the different CPUs is made through a global shared memory. Any CPU can read or write a word of memory by just executing LOAD and STORE instructions. Besides the common memory, each processor code may have some local memory which can be used for program code and those items that need not be shared. In this case, the MPSoC architecture executes a multithreaded application organized as a single software stack.

The *message-passing* organization assumes multiple software stacks running on identical or non-identical software subsystems. The communication between different subsystems is generally made through message passing. The key property of this class is that the communication between the different processors is explicit through I/O operations. The CPUs communicate by sending each other message by using primitives such as *send* and *receive*. There are three types of message passing: synchronous (if the sender executes a *send* operation and the receiver has not yet executed a *receive*, the sender is blocked until the receiver executes the *receive*), buffered or asynchronous blocking (when a message is sent before the receiver is ready, the message is buffered somewhere, for example, in a mailbox, until the receiver takes it out; thus the sender can continue after a *send* operation, if the receiver is busy with something else), and asynchronous non-blocking (the sender may continue immediately after making the communication call) [150].

Heterogeneous MPSoC generally combines both models to integrate a massive number of processors on a single chip [122]. Future heterogeneous MPSoC will be made of few heterogeneous subsystems where each may include a massive number of the same processor to run a specific software stack [87].

This book considers heterogeneous MPSoC architectures organized as it was illustrated previously in Fig. 2.1 with the support of message-passing communication model.

Besides the hardware architecture previously presented, the MPSoC means also software running on hardware. The major challenge for technical success of MPSoC is to make sure that the software executes efficiently on the hardware [18].

### 2.2 Programming Models for MPSoC

Several tools exist for automatic mapping of sequential programs on homogeneous multiprocessor architectures. Unfortunately, these are not efficient for heterogeneous MPSoC architectures. In order to allow the design of distributed applications, programming models have been introduced and extensively studied by the software communities to allow high-level programming of heterogeneous multiprocessor architectures.

The programming model specifies how parts of the application running in parallel communicate information to one another and what synchronization operations are
available to coordinate their activities. Applications are written in a programming model. The programming model specifies what data can be named by the different parallel processes, what type of operations can be executed on the named data, and what ordering exists between the different operations [42].

Examples of parallel programming models are as follows:

- Shared address space, when the communication is performed by posting data into shared memory locations, accessible by all the communicating processing elements. This programming model also involves special atomic operations for the synchronization and data protection.

- Data-parallel programming, when several processing units perform the same operations simultaneously, but on separate parts of the same data set. The data set has a regular structure, i.e., array or matrix. At the end of the operations, the processes exchange synchronization information globally, before continuing the operations with a new data set.

- Message passing, when the communication is performed between a specific sender and a specific receiver. This involves a well-defined event when the data is sent or received, and these events are the basis for orchestrating the individual activities. Anyhow, there are no shared locations accessible to all processing elements. The most common communication primitives used in message-passing programming model are variants of send and receive. In its simplest form, send specifies a local data buffer that is to be transmitted and a receiving process, typically a remote processor. The receive operation specifies a sending process and a local data buffer into which the transmitted data will be placed. The message passing can be further divided into two communication-centric programming models: client–server and streaming.

In the basic client–server model, the communicating processes are divided into two (possibly overlapping) groups. A server is a process implementing a specific service, for example, a file system service. A client is a process that requests a service from a server by sending it a request and subsequently waiting for the server’s reply. This client–server interaction, also known as request–reply behavior is shown in Fig. 2.2. When a client requests a service, it simply packages a message for the server, identifying the service it wants, along with the necessary input data. The

![Fig. 2.2 Client–server communication model](image-url)
message is then sent to the server. The latter, in turn, will always wait for an incoming request, subsequently process it, and package the results in a reply message that is then sent to the client.

Common object request broker architecture (CORBA) is a well-known specification for distributed systems which adopts an object-based approach for the communication, based on client–server model [116]. All communication takes place by invoking an object. An object provides services, and it combines functional interface as well as data. An object request broker (ORB) connects a client to an object that provides a service. Each object instance has a unique object reference. The client and the object do not need to reside on the same processor; a request to a remote processor can invoke multiple ORBs. The object logically appears to be a single entity, but the server may keep a thread pool running to implement object calls for a variety of clients. Because the client and the object use the same protocol, the object can provide a consistent service independent of the processing element on which it is implemented.

Streaming is a form of communication in which timing plays a crucial role [10]. The support for the exchange of time-dependent information is often formulated as a support for continuous media or stream, e.g., support for reproducing a sound wave by playing out an audio stream at a well specified rate, or displaying a certain number of images per second for a movie. Streams can be simple or complex. A simple stream consists of only a single sequence of data, whereas a complex stream consists of several related simple streams, called substreams. For example, stereo audio can be transmitted by means of a complex stream consisting of two substreams, each used for a single audio channel. It is important, however, that those two substreams are continuously synchronized. Another example of a complex stream is one for transmitting a movie. Such a stream could consist of a single video stream, along with two streams for transmitting the sound of the movie in stereo. The transmission of these data streams can be effectuated in the following:

- Asynchronous mode, when the data items in a stream are transmitted one after the other, but there are no further timing constraints when transmission of these items should take place.
- Synchronous mode, when there is a maximum end-to-end delay defined for each data unit of the stream. In this case, the time required for the data transmission has to be guaranteed to be lower than the maximum permitted delay.
- Isochronous mode, when there is a maximum and minimum end-to-end delay for each data unit of the stream. The end-to-end delays are usually expressed as quality of service (QoS) requirements.

The QoS ensures that the temporal relationship between the streams is preserved. There are several ways to enforce QoS for streaming applications, e.g., by data dropping if the communication network gets congested, or by applying error correction techniques, e.g., encoding the outgoing data units in such a way that any \( k \) out of \( n \) received data units is enough to reconstruct \( k \) correct data units.
StreamIt is an example of programming model for streaming systems [153]. The StreamIt language has mainly two goals: to provide high-level stream abstractions that improve programmer productivity and program robustness within the streaming domain and, second, to serve as a common machine language for grid-based processors. At the same time, the StreamIt compiler aims to perform stream-specific optimizations to achieve high performance.

2.2.1 Programming Models Used in Software

The programming model is usually embodied in a parallel language or a programming environment [42].

As long as only the software is concerned, Skillicorn [145] identifies five key concepts that may be hidden by the programming model, namely, concurrency or parallelism of the software, decomposition of the software into parallel threads, mapping of threads to processors, communication among threads, and synchronization among threads. These concepts define six different abstraction levels for the programming models.

Table 2.1 summarizes the different levels with typical corresponding programming languages for each of them. All these programming models take into account only the software side. They assume the existence of lower levels of software and a hardware platform able to execute the corresponding model.

The programming models are presented in decreasing order of abstraction in the following six categories:

– Programming models that abstract the parallelism completely. Such programming models describe only the purpose of an application and not how it is to achieve this purpose. Software designers do not need to know even if the application will execute in parallel. Such programming levels are abstract and relatively simple, since the applications need to be no more complex than sequential ones.

<table>
<thead>
<tr>
<th>Table 2.1 The six programming levels defined by Skillicorn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction level</td>
</tr>
<tr>
<td>Implicit concurrency</td>
</tr>
<tr>
<td>Parallel level</td>
</tr>
<tr>
<td>Thread level</td>
</tr>
<tr>
<td>Agent models</td>
</tr>
<tr>
<td>Process network</td>
</tr>
<tr>
<td>Message passing</td>
</tr>
</tbody>
</table>
2.2 Programming Models for MPSoC

– Programming models in which parallelism is made explicit. But the decomposition of the application into threads is still implicit, hence so is the mapping, communication, and synchronization concepts. In such programming models, the software designers are aware that parallelism will be used and must have expressed the potential for it in the application. But they do not know how much parallelism will actually be applied at runtime. Such programming models often require the applications to express the maximal parallelism provided by the algorithm, and then reduce that degree of parallelism to fit the target architecture, while at the same time working out the implications for mapping, communication, and synchronization.

– Programming models in which parallelism and decomposition must both be made explicit, but mapping, communication, and synchronization are implicit. Such programming models require decisions about the breaking up of the application into parallel executed threads, but they relieve the software designer of the implications of such decisions.

– Programming models in which parallelism, decomposition, and mapping are explicit, but communication and synchronization are implicit. In this case, the software developer must not only decompose the application into parallel threads but also consider how best to map the parallel threads on the target processor. Since mapping will often have a marked effect on the communication performance, this almost inevitably requires an awareness of the target processor’s interconnection network. It becomes very hard to make such software portable across different architectures.

– Programming models in which parallelism, decomposition, mapping, and communication are explicit, but synchronization is implicit. In this case, the software designer is making almost all of the implementation decisions, except that fine-scale timing decisions are avoided by having the system deal with synchronization.

– Programming models in which all the five concepts are explicit. In this case, the software designers must specify the whole implementation. Thereby, it is extremely difficult to build software using such programming models, because both correctness and performance can only be achieved by attention to vast numbers of details.

2.2.2 Programming Models for SoC Design

In order to allow concurrent hardware/software design, we need to abstract the hardware/software interfaces, including both software and hardware components. Similar to the programming models for software, the hardware/software interfaces may be described at different abstraction levels. The four key concepts that we consider are explicit hardware resources, management and control strategies for the hardware resources, the CPU architecture, and the CPU implementation. These concepts define four abstraction levels described in the previous chapter, namely system architecture level, virtual architecture level, transaction-accurate architecture level,
Table 2.2 Additional models for SoC design

<table>
<thead>
<tr>
<th>Abstraction level</th>
<th>Typical programming languages</th>
<th>Explicit concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual architecture</td>
<td>Untimed SystemC [16]</td>
<td>+Abstract communication resources</td>
</tr>
<tr>
<td>Transaction-accurate</td>
<td>TLM SystemC [16]</td>
<td>+Resources sharing and control strategies</td>
</tr>
<tr>
<td>architecture</td>
<td></td>
<td>+ ISA and detailed I/O interrupts</td>
</tr>
<tr>
<td>Virtual prototype</td>
<td>Co-simulation with ISS</td>
<td></td>
</tr>
</tbody>
</table>

and virtual prototype level, as summarized in Table 2.2. The different abstraction levels may be expressed by a single and unique programming model that uses the same or different primitives for each level.

At the system architecture level, all the hardware is implicit similar to the message-passing model used for software. The hardware/software partitioning and the resources allocation are made explicit. This level fixes also the allocation of the tasks to the various subsystems. Thus, the model combines both the specification of the application and the architecture and it is also called combined architecture algorithm model (CAAM). At the virtual architecture level, the communication resources, such as global interconnection components and buffer storage components, become explicit. The transaction-accurate architecture level implements the resource management and control strategies. This level fixes the RTOS on the software side. On the hardware side, a functional model of the bus is defined. The software interface is specified to the HAL level while the hardware communication is defined at the bus transaction level. Finally, the virtual prototype level corresponds to the classical co-simulation with instruction set simulators (ISS). At this level the architecture of the CPU is fixed, but not yet its implementation that remains hidden by an ISS.

Several languages can cover multiple abstraction levels for SoC design, such as C, C++. In fact, most real embedded software at both higher abstraction levels (system architecture) and lower levels uses C/C++ as a stretch. While SystemC, a C++ class, is useful to model behavior and architecture blocks, the behavior is likely to be written in C especially at low level.

### 2.2.3 Defining a Programming Model for SoC

The use of programming models for the software design of heterogeneous MPSoC requires the definition of new design automation methods to enable concurrent design of hardware and software. This also requires new models to deal with non-standard application-specific hardware/software interfaces at several abstraction levels. The software design makes use of a programming model.
The programming model abstracts the hardware for the software design. It is made of a set of functions (implicit and/or explicit primitives) that can be used by the software to interact with the hardware. Additionally, the programming model needs to cover the four abstraction levels required for the software refinement previously presented (system architecture, virtual architecture, transaction-accurate architecture, and virtual prototype). In order to cover different abstraction levels of both software and hardware, the programming model needs to include three kinds of primitives:

- Communication primitives: these are aimed to exchange data between the hardware and the software.
- Task and resource control primitives: these are aimed to handle task creation, management, and sequencing. At the system architecture level, these primitives are generally implicit and built in the constructions of the language. The typical scheme is the module hierarchy in block structure languages, where each module declares implicit execution threads.
- Hardware access primitives: these are required when the architecture includes specific hardware. The primitives include specific primitives to implement specific protocol or I/O schemes, for example, a specific memory controller allowing multiple accesses. These will always be considered at lower abstraction layers and cannot be abstracted using the standard communication primitives.

The programming models at the different abstraction levels previously described are summarized in Table 2.3. The different abstraction levels may be expressed by a single and unique programming model that uses the same primitives applicable at different abstraction levels or it uses different primitives for each level.

<table>
<thead>
<tr>
<th>Abstraction level</th>
<th>Communication primitives</th>
<th>Task and resource control</th>
<th>HW access primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>System architecture</td>
<td>Implicit, e.g., Simulink links</td>
<td>Implicit, e.g., Simulink blocks</td>
<td>Implicit, e.g., Simulink links</td>
</tr>
<tr>
<td>Virtual architecture</td>
<td>Data exchange, e.g., send/receive (data)</td>
<td>Implicit tasks control, e.g., threads in SystemC</td>
<td>Specific I/O protocols related to architecture</td>
</tr>
<tr>
<td>Transaction-accurate architecture</td>
<td>Data access with specific addresses, e.g., read/write (data, adr)</td>
<td>Explicit tasks control, e.g., create/resume task HW management of resources, e.g., test/set</td>
<td>Physical access to HW resources</td>
</tr>
<tr>
<td>Virtual prototype</td>
<td>Load/store</td>
<td>HW arbitration and address translation, e.g., memory map</td>
<td>Physical I/Os</td>
</tr>
</tbody>
</table>
2.2.4 Existing Programming Models

A number of MPSoC-specific programming models, based on shared memory or message passing, have been defined recently. Examples of programming models can be considered: OpenMP [33] for shared memory architectures and MPI [108], TTL [160], or YAPI [78] for message passing architectures. This section will detail some of them.

2.2.4.1 Message-Passing Interface (MPI)

The message-passing interface (MPI) is a message-passing library interface specification. The last version 2.2 was recently adopted as standard [108]. It includes the specification of a set of primitives for point-to-point communication with message passing, collective communications, process creation and management, one-sided communications, and external interfaces.

The following APIs represent examples of blocking communication primitives within MPI:

- \texttt{MPI\_Send} (void *buf, int count, MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm)
- \texttt{MPI\_Recv} (void *buf, int count, MPI\_Datatype datatype, int src, int tag, MPI\_Comm comm, MPI\_Status *status)

where

- \texttt{buf} is the source/destination buffer to be sent/received
- \texttt{count} is the number of elements to be sent/received
- \texttt{datatype} is the data type of the data to be sent/received (e.g., MPI\_CHAR, MPI\_INT, MPI\_DOUBLE)
- \texttt{dest/src} represents the rank or identifier of the destination, respectively, source
- \texttt{tag} represents the message tag used to distinguish among the messages in case that two communicating partners exchange more than one message
- \texttt{comm} identifies the group of the communicator
- \texttt{status} indicates the result of the receive operation, whether or not an error occurred during the data transmission

The send call previously presented blocks until the message is copied to the destination buffer. But message buffering can decouple the send and receive operations. This means that the send operation can complete as soon as the message was buffered, even if no matching receive operation was executed by the receiver. Essentially, there are three types of communication modes:

- Buffered mode, when the send operation can start even if no matching receive operation was initiated and it may complete before the corresponding receive
starts. If there is no space in the buffer for the outgoing message, then an error will occur.

- Synchronous mode, when the send operation can start even if no matching receive operation was initiated, but the send will complete successfully only if the matching receive started to receive the message sent by the synchronous send.
- Ready mode, when the send operation can start only if the corresponding receive is already posted.

The primitives associated with these three modes are MPI_BSend, MPI_SSend, and MPI_RSend/MPI_RRecv.

MPI defines also non-blocking communication primitives: MPI_ISSend and MPI_IRRecv (immediate send and immediate receive). The non-blocking APIs support also the three communication modes: buffered, synchronous, and ready mode and they use the same naming conventions as the blocking type: B for buffered mode, S for synchronous, and R for ready, i.e., MPI_IBSend, MPI_ISSend, MPI_IRSend/MPI_IRRecv.

Besides communication, MPI defines also standard APIs for the process management. These are MPI_Comm_Spawn and MPI_Comm_Multiple, used to start new processes by a running MPI application and establish communication with them.

All these standard primitives are implemented in various libraries. An example of implementation is the MPICH library [109]. This supports three types of devices for the communication: sockets for communication between processing units; mixed socket for communication between processors and shared memory within a multicore processor; and shared memory within an SMP architecture.

2.2.4.2 Multi-core Communications API (MCAPI)

Other research works focus on the standardization of the communication APIs, such as the Multi-core Association working group, which developed the MCAPI (multi-core communications APIs) [98]. The MCAPI defines a set of communication APIs for multi-core communications, to support lightweight, high-performance implementations typically required in embedded applications. MCAPI captures the basic elements of inter-core communications that are required for embedded “closely distributed” systems and scales to support hundreds of processor cores. The potential applications for such an API are extremely varied, but its principal use is the embedded multi-core systems with tight memory constraints and task execution times, requiring reliable on-chip interconnect and high system throughput. Besides the details of the API, the MCAPI specification includes example usage models for multimedia, networking, and automotive applications. MCAPI provides three communication modes: connectionless messages, connected channels for packets, and connected channels for scalars. It also provides functions for endpoint and non-blocking operations management.
2.2.4.3 Y-Chart Application Programmer’s Interface (YAPI)

The Y-chart application programmer’s interface (YAPI) is an application programmer’s interface to write signal and stream processing applications as process networks, developed by Philips Research [78]. The communication between processes is based on Kahn process networks with blocking reads on theoretically unbounded FIFOs.

The Kahn process network is a computational model which consists of a set of concurrent processes [72]. Each of the processes performs sequential computation on its private state space. The processes communicate with each other via uni-directional FIFO channels. A FIFO channel has one input end and one output end, i.e., there is exactly one process that writes to the channel and there is also exactly one process that reads values from the FIFO. The process has input ports which transfer data from the FIFO to the process by reading values and output ports which copy data from the process to the FIFO by writing values.

Y API is a C++ library with a set of rules which can be used to model and execute an application as a Kahn process network. The syntax for reading values is \texttt{read(p, x)}. This statement reads a value from input port \( p \) and stores this value in variable \( x \). The syntax for writing values is \texttt{write(q, y)}. This statement writes the value of variable \( y \) to output port \( q \). YAPI supports also to read and write vectors, not only scalars. The corresponding APIs are \texttt{read(p, x, m)}, which reads \( m \) values from port \( p \) into array \( x \), respectively, \texttt{write(q, y, n)}, which writes \( n \) values of array \( y \) to the port \( q \).

2.2.4.4 Task Transaction Level (TTL)

The task transaction level interface (TTL) proposed in [160] is derived from YAPI and focuses on stream processing applications in which concurrency and communication are explicit. The interaction between tasks is performed through communication primitives with different semantics, allowing blocking or non-blocking calls, in order or out of order data access, and direct access to channel data. The TTL APIs define three abstraction levels. The \texttt{vector\_read} and \texttt{vector\_write} functions are typical system level functions, which combines synchronization with data transfers. The \texttt{reAcquireRoom} and \texttt{releaseData} functions (re stands for relative) grant/release atomic accesses to vectors of data that can be loaded or stored out of order, but relative to the last access, i.e., with no explicit address. This corresponds to virtual architecture level APIs. Finally, the \texttt{AcquireRoom} and \texttt{releaseData} lock and unlock access to scalars, which requires the definition of explicit addressing schemes. This corresponds to the transaction-accurate architecture level APIs.

2.2.4.5 Distributed System Object Component (DSOC)

The Multiflex approach proposed in [122] targets multimedia and networking applications, with the objective of having good performance even for small granularity tasks. Multiflex supports both symmetric multi-processing (SMP) approach used
on shared memory multiprocessors and remote procedure call based programming approach, called DSOC (distributed system object component). The SMP functionality is close to the one provided by POSIX, i.e., thread creation, mutexes, condition variables, etc. [28]. The DSOC uses a broker to spawn the remote methods. These abstractions make no separation between virtual architecture and transaction-accurate architecture levels, since they rely on fixed synchronization mechanisms. The hardware support to locks and the run queues management is provided by a concurrency engine. The processors have several hardware contexts to allow context switches in one cycle. DSOC uses a CORBA-like approach but implements hardware accelerators to optimize the performances.

2.2.4.6 Compute Unified Device Architecture (CUDA)

Another example of programming model used in industry is the CUDA architecture provided by Nvidia [115]. CUDA (compute unified device architecture) is a software platform for massively parallel high-performance computing on powerful Nvidia GPUs (graphics processing units). CUDA requires programmers to write special code for parallel processing, but it does not require them to explicitly manage threads in the conventional sense, which greatly simplifies the programming model. CUDA development tools work alongside a conventional C/C++ compiler, so programmers can mix GPU code with general-purpose code for the host CPU. The architecture of GPUs is hidden beneath APIs. This hardware abstraction has two benefits: first, it simplifies the high-level programming model, insulating programmers from the complex details of the GPU hardware. Second, the hardware abstraction allows flexibility in the GPU architecture. Currently, CUDA aims at data-intensive applications that need single-precision floating-point math, but future perspective envisions a new double precision floating-point GPU.

CUDA’s programming model differs significantly from single-threaded CPU code and even the parallel code that some programmers began writing for GPUs before CUDA. In a single-threaded model, the CPU fetches a single instruction stream that operates serially on the data. A superscalar CPU may route the instruction stream through multiple pipelines, but there is still only one instruction stream, and the degree of instruction parallelism is severely limited by data and resource dependencies. Even the best four-way, five-way, or six-way superscalar CPUs struggle to average 1.5 instructions per cycle, which is why superscalar designs rarely venture beyond four-way pipelining. Single-instruction multiple-data (SIMD) extensions permit many CPUs to extract some data parallelism from the code, but the practical limit is usually three or four operations per cycle [115].

Another programming model is general-purpose GPU (GPGPU) processing. This model is relatively new and has gained much attention in recent years. Essentially, developers hungry for high performance began using GPUs as general-purpose processors, although “general purpose” in this context usually means data-intensive applications in scientific and engineering fields. Programmers use the GPU’s pixel shaders as general-purpose single-precision FPUs. GPGPU processing is highly parallel, but it relies heavily on off-chip “video” memory to operate
on large data sets. (Video memory, normally used for texture maps and so forth in graphics applications, may store any kind of data in GPGPU applications.) Different threads must interact with each other through off-chip memory. These frequent memory accesses tend to limit performance [115].

CUDA takes a third approach. Like the GPGPU model, it is highly parallel. But it divides the data set into smaller chunks stored in on-chip memory, and then allows multiple thread processors to share each chunk. Storing the data locally reduces the need to access off-chip memory, thereby improving performance. Occasionally, of course, a thread does need to access off-chip memory, such as when loading the off-chip data it needs into local memory. In the CUDA model, off-chip memory accesses usually do not stall a thread processor. Instead, the stalled thread enters an inactive queue and is replaced by another thread that is ready to execute. When the stalled thread’s data becomes available, the thread enters another queue that signals it is ready to go. Groups of threads take turn executing in round-robin fashion, ensuring that each thread gets execution time without delaying other threads [115].

### 2.2.4.7 Open Computing Language (OpenCL)

Open computing language (OpenCL) is an open standard for writing applications that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors, introduced by Khronos working group [81]. OpenCL provides parallel computing using task-based and data-based parallelism.

OpenCL includes a C-based language for writing kernels (functions that execute on OpenCL devices), called also run-time APIs, plus APIs that are used to define and then control the platforms, also known as platform layer APIs. The run-time APIs serve to execute computational or compute kernels and manage scheduling of computational and memory resources. The platform layer APIs represent a hardware abstraction layer over diverse computational resources and are used to query, select, and initialize compute devices in the systems and to create compute contexts and work queues. A compute device is a collection of one or more computational units or cores, which can be a CPU or GPU.

The execution model in OpenCL resides on two concepts: compute kernel and compute program. The compute kernel is the basic unit of executable code, similar to a C function, and it can be data parallel or task parallel. The compute program is a collection of compute kernels and internal functions, similar to a dynamic library. The applications queue the compute kernel execution instances in order. Then, the compute kernel execution instances can be executed in order or out of order. The data-parallel execution is achieved by defining work items that execute in parallel. The work items can be grouped together to form a work group. The work items within a group can communicate with each other and can synchronize their execution to coordinate the memory access. Also, multiple work groups can be executed in parallel. Some compute devices such as CPUs can also execute task-parallel compute kernels as a single work item. The following examples illustrate the usage of OpenCL APIs to create compute programs for the FFT application:
2.2 Programming Models for MPSoC

// create compute context with GPU device
context = clCreateContextFromType(CL_DEVICE_TYPE_GPU);
// create a work-queue
queue = clCreateWorkQueue(context, NULL, NULL, 0);
// allocate memory buffer objects
memobjs[0] = clCreateBuffer (context, CL_MEM_READ_WRITE,
    sizeof(float)*2, NULL);
// create the compute program for FFT application
program = clCreateProgramFromSource(context, 1,
    &fft1D_kernel_src, NULL);
// build the compute program executable
clBuildProgramExecutable (program, false, NULL, NULL);
// create the compute kernel
kernel = clCreateKernel (program, “fft1D”);

The memory model used in OpenCL is shared memory model but with multiple distinct address spaces that can be collapsed.

2.2.4.8 Open Multi-processing (OpenMP)

The open multi-processing (OpenMP) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran on many architectures [117].

The OpenMP assumes a shared memory model, with all the threads having access to the same, globally shared memory. The data needs to be labeled as shared or private. The shared data is accessible by all threads and there is a single instance of the data. Private data can be accessed only by the thread which owns it. The data transfer is transparent to the programmer and the synchronization is mostly implicit.

OpenMP consists of a set of compiler directives to define a parallel region, work sharing, data sharing attributes like shared or private, tasking, etc.; library routines and environment variables (e.g., number of threads, stack size, scheduling type, dynamic thread adjustment, nested parallelism, active levels, thread limit) that influence the run-time behavior. An example of parallel region which declares two parallel loops is illustrated below:

```c
#pragma omp parallel if(n>limit) default(none)
    shared(n,a,b,c,x,y,z) private(f,i,scale) {
    f = 1.0;
    // parallel loop, work is distributed
    #pragma omp for nowait
    for(i=0; i<n; i++)
        z[i]=x[i]+y[i];
    // parallel loop, work is distributed
```
2.2.4.9 Transaction-Level Modeling (TLM)

Another standardization work is related to SystemC transaction-level modeling (TLM) [118]. The open SystemC initiative (OSCI) proposes TLM-2 draft 1 and TLM-2 draft 2, which offers a set of standard APIs and a library that implements a foundation layer upon which interoperable transaction-level models can be built. The standard proposal is designed to facilitate intellectual property (IP) sharing and re-use, faster EDA tool development, and make it a lot easier for electronic OEMs to use TLM. The TLM-2 draft 1 version defines two transaction modeling styles – the untimed programmer’s view (PV) model and the PV+T (programmer’s view with timing information) model. The new draft adopted recently, TLM-2 draft 2, retires one previously proposed modeling style, the programmer’s view plus annotated timing (PV+T), and introduces two new transaction-level modeling styles – loosely timed (LT) and approximately timed (AT). The newly introduced loosely timed (LT) modeling style is suitable for software application development, software performance analysis, and hardware architectural analysis. It employs the flexible non-blocking transport in a lightweight manner and close to untimed performance. The newly proposed approximately timed (AT) modeling style is closer to timed behavior, i.e., in terms of modeling contention and arbitration. Thus, it is suitable for hardware architectural analysis and performance verification. The AT shares the same non-blocking transport used by LT but defines finer-grained timing control. Using a generic payload, it monitors four or more communication events – depending on the protocol – for example, beginning and end of request and beginning and end of response.

2.2.4.10 Other Examples of Programming Models

The authors in [24] introduce the concept of service dependency graph to represent HW/SW interface at different abstraction levels and to handle application-specific API. This model represents the hardware/software interface as a set of interdependent components providing and requiring services.

Cheong et al. [35] propose a programming model called TinyGALS, which combines the locally synchronous with the globally asynchronous approach for programming event-driven embedded systems.
In [176] the authors describe PTIDES (programming temporally integrated distributed embedded systems) programming model. This defines an execution strategy based on discrete-event semantics and then specializes this strategy to give distributed policies for execution scheduling and control.

LLVM stands for low-level virtual machine and it represents a compilation strategy designed to enable software code optimization at compile time, link time, and runtime [92]. It has a virtual instruction set to represent the low-level object code.

In the previous section (Table 2.3), we showed that a suitable programming model for MPSoC needs to be defined at several abstraction levels corresponding to different design steps. This hierarchical view of the programming model ensures a seamless implementation of high-level APIs onto the low-level ones. In order to ensure a better match between the programming model and the underlying hardware architecture, the APIs also have to be extensible at each abstraction level, to cope with the broad range of possible hardware components. The existing MPSoC programming models seem to focus either on one aspect or on the other. We think that it is important to consider both aspects, i.e., hierarchy and extensibility when designing an MPSoC-oriented programming model.

2.3 Software Stack for MPSoC

The software running on the MPSoC architecture is called embedded software. The software costs are often a large part of the total cost of an embedded system and are characterized by different performance requirements [69].

Often, the performance requirement in an embedded application is a real-time requirement. A real-time performance requirement is one where a segment of the application has an absolute maximum execution time that is allowed. For example, in a digital set-top box the time to process each video frame is limited, since the processor must accept and process the next frame shortly. In some applications, a more sophisticated requirement exists: the average time for a particular task is constrained as well as the numbers of instances when some maximum time is exceeded. Such approaches (sometimes called soft real time) arise when it is possible to occasionally miss the time constraints on an event, as long as not too many are missed. Real-time performances tend to be highly application dependent.

Two other key characteristics exist in many embedded applications: the need to minimize the memory and the need to minimize the power. Sometimes the application is expected to fit completely in the memory of the processor on chip; other times the application needs to fit totally in a small off-chip memory. In any event, the importance of memory size translates to an emphasis on code size, since data size is dictated by the application’s algorithm. Large memories also mean more power [69].

2.3.1 Definition of the Software Stack

In this book, the software running on the software subsystems is called software stack. In heterogeneous MPSoC architectures, each software subsystem executes
a software stack. The software stack is made of two components: the application tasks code and the hardware-dependent software (HdS). The HdS layer is made of three components: the operating system (OS), specific I/O communication software, and the hardware abstraction layer (HAL). The HdS is responsible to provide application- and architecture-specific services, i.e., scheduling the application tasks, communication between the different tasks, external communication with other sub-systems, hardware resource management and control. The following paragraphs detail the software stack organization, including all these different components.

### 2.3.2 Software Stack Organization

The software stack is structured in different software layers that provide specific services. Figure 2.3 illustrates the software stack organization in two layers: application layer and HdS (hardware-dependent software) layer. In the first section, the application layer will be presented and then the HdS will be defined.

![Software stack organization](image)

#### 2.3.2.1 Application Layer

The application layer may be a multi-tasking description or a single task function of the application targeted to be executed on the software (processor) subsystem. A *task* or *thread* is a lightweight process that runs sequentially and has its own program counter, register set, and stack to keep track of where it is. In this book, the terms task and thread are used as interchangeable terms. Multiple tasks can be executed in parallel by a single CPU (single-core) or by multiple CPUs of the same type grouped in the software subsystem (multi-core). The tasks may share the same resources of the architecture, such as processors, I/O components, and memories. On a single processor core node, the multithreading generally occurs by time slicing, wherein a single processor switches between different threads. In this case, the processing is not literally simultaneous, as the single processor is doing only one thing at a time. On a multi-core processor subsystem, threading can be achieved via multiprocessing, wherein different threads can run literally simultaneously on different processors inside the software node [148].
The application layer consists of a set of tasks that makes use of programming model or application programming interface (API) to abstract the underlying HdS software layer. These APIs corresponds to the HdS APIs.

2.3.2.2 HdS Layer

The HdS layer represents the software layer which is directly in contact with, or significantly affected by, the hardware that it executes on, or can directly influence the behavior of that hardware [127]. The HdS integrates all the software that is directly depending on the underlying hardware, such as hardware drivers or boot strategy. It also provides services for resource management and sharing, such as scheduling the application tasks on top of the available processing elements, inter-task communication, external communication, and all other kinds of resource management and control. The federative HdS term underlines the fact that, in an embedded context, we are concerned with application-specific implementations of these functionalities that strongly depend on the target hardware architecture [87].

Current research studies proved that the HdS debug represents 78% of the global system total debugging time of an MPSoC software design cycle [175]. This may due to incorrect configuration or access to the hardware architecture, e.g., a wrong configuration of the memory mapping for the interrupt control registers. In order to reduce its complexity, the HdS is structured into three software components: operating system (OS), communication management (Comm), and hardware abstraction layer (HAL).

Operating System

The operating system (OS) is the software component that manages the sharing of the resources of the architecture. It is responsible for the initialization and management of the application tasks and communication between them. It provides services such as tasks scheduling, context switch, synchronization, and interrupt management. In the following, more details about these OS services will be given.

The tasks scheduling service of the OS usually follows a specific scheduling algorithm. Finding the optimal algorithm for the tasks scheduling represents an NP-complete problem [162]. There are different categories of scheduling algorithms. The classic criteria are hard real time versus soft real time or non-real time; preemptive versus cooperative; dynamic versus static; and centralized versus distributed [148].

Contrary to non-real time, the real-time scheduler must guarantee the execution of a task in a certain period of time. Hard real time must guarantee that all deadlines are met.

Preemptive scheduling allows a task to be suspended temporarily by the OS, for example, when a higher priority task arrives, resuming later when no higher priority tasks are available to run. This is associated with time sharing between the tasks. Examples of preemptive scheduling algorithms are round-robin, shortest remaining time, or rate monotonic schedulers. The cooperative or non-preemptive scheduling
algorithm runs each task to its completion. In this case, the OS waits for a task to surrender control. This is usually associated with event-driven operating systems. Examples of non-preemptive algorithm are the shortest job next or highest response ratio next.

With static algorithms, the scheduling decisions (preemptive or non-preemptive) are made before run-time. Contrary to static algorithms, the dynamic schedulers make their scheduling decisions during the execution.

The implementation of the scheduler may be centralized or distributed. In case of a centralized scheduler implementation, the scheduler controls all the task execution ordering and communication transactions. In case of a distributed scheduler implementation, the scheduler distributes the control decision to the local task schedulers corresponding to each processor [38].

When a task is ready for execution and it is selected by the scheduler of OS according to the scheduler algorithm, the OS is also responsible to perform the context switch between the currently running task and the new task. The context switch represents the process of storing and loading the state of the CPU in order to share the available hardware resources between different tasks. The state of the current task, including registers, is saved, so that in case the scheduler gets back for execution of the first task, it can restore its state and continue normally.

In order to ensure a correct runtime and communication order between the different tasks running on parallel, synchronization is required. The tasks can synchronize by using semaphores to control access to shared resource or by sending/receiving synchronization signals (events) to each other. The mutex is a binary semaphore which ensures mutual exclusion on a shared resource, such as a buffer shared by two threads, by locking and unlocking it whenever the resource is accessed by a task [149, 150].

The interrupt handler is another OS service used for interrupts management. There are two types of processor interrupts: hardware and software. A hardware interrupt causes the processor to save its state of execution via a context switch and begins the execution of an interrupt handler. Software interrupts are usually implemented as instructions in the instruction set of the processor, which cause a context switch to an interrupt handler similar to a hardware interrupt. The interrupts represent a way to avoid wasting the processor’s execution time in polling loops waiting for external events. Polling means when the processor waits and monitors a device until the device is ready for an I/O operation.

Examples of commercial OS are the eCos [46], FreeRTOS [51], LynxOS [93], VxWorks [170], WindowsCE [104], or µITRON [158].

Communication Software Component

The second software component of the HDS layer constitutes the communication component, which is responsible to manage the I/O operations and more generally the interaction with the hardware components and the other subsystems. The communication component implements the different communication primitives used inside a task to exchange data between the tasks running on the same processor or between the tasks running on different processors. It may include different
communication protocols, such as FIFO (first in–first out) implemented in software, or communication using dedicated hardware components. If the communication requires access to the hardware resources, the communication component invokes primitives that implement this kind of low-level access. These function calls are done in form of HAL APIs.

The HAL APIs allow for the OS and communication components to access the third component of the software stack, that is, the HAL layer.

Hardware Abstraction Layer

Low-level details about how to access the resources are specified in the hardware abstraction layer (HAL) [174]. The HAL is a thin software layer which not only totally depends on the type of processor that will execute the software stack but also depends on the hardware resources interacting with the processor. The HAL includes the device drivers to implement the interface for the communication with the device. This includes the implementation of drivers for the I/O operations or for other peripherals. The HAL is responsible also for processor-specific implementations, such as loading the main function executed by an OS, more precisely the boot code, or the implementation of the load and restore of CPU registers during a context switch between two tasks, but also code for the configuration and access to the hardware resources, e.g., MMU (memory management unit), timer, interrupt enabling/disabling, etc.

The structured representation of the software stack in several layers (application tasks, OS, communication, and HAL), as previously described, has two main advantages: flexibility in terms of software components re-use by changing the OS or the communication software components, and portability to other processor subsystems by changing the HAL software layer.

2.4 Hardware Components

2.4.1 Computing Unit

The microprocessor, also known as a CPU, central processing unit, computing unit, or just processor, is a complete computation engine that is fabricated on a single chip. A chip is also called an integrated circuit. Generally it is a small, thin piece of silicon onto which the transistors making up the microprocessor have been etched. A chip might be as large as an inch on a side and can contain tens of millions of transistors. Simpler processors might consist of a few thousand transistors etched onto a chip just a few millimeters square.

Generally, the microprocessors provide the following characteristics:

- The date is the year that the processor was first introduced. Many processors are re-introduced at higher clock speeds for many years after the original release date.
- Transistors is the number of transistors on the chip. The number of transistors on a single chip has risen steadily over the years.
- **Microns** is the width, in microns, of the smallest wire on the chip. As the feature size on the chip goes down, the number of transistors rises.

- **Clock speed** is the maximum rate that the chip can be clocked at.

- **Data width** is the width of the ALU (arithmetic and logic unit), which is the main component of the processor. An 8-bit ALU can add/subtract/multiply/etc., two 8-bit numbers, while a 32-bit ALU can manipulate 32-bit numbers. An 8-bit ALU would have to execute four instructions to add two 32-bit numbers, while a 32-bit ALU can do it in one instruction. In many cases, the external data bus is the same width as the ALU, but not always. For instance, the 8088 Intel processor had a 16-bit ALU and an 8-bit bus, while the modern Pentium processors fetch data 64 bits at a time for their 32-bit ALUs [85].

- **MIPS** stands for “millions of instructions per second” and is a rough measure of the performance of a CPU.

The microarchitecture of the CPU is comprised of five basic components: memory, registers, buses, the ALU, and the control unit. Each of these components is pictured in Fig. 2.4.

![CPU microarchitecture](image)

- **Memory**: this component is created from combining latches with a decoder. The latches create circuitry that can store information, while the decoder creates a way for individual memory locations to be selected.

- **Registers**: these components are special memory locations that can be accessed very fast. Three registers are shown in the figure: the instruction register (IR), the program counter (PC), and the accumulator.

- **Buses**: these components are the information highway for the CPU. Buses are bundles of tiny wires that carry data between components. The three most important buses are the address, the data, and the control buses.
– **ALU**: this component is the number cruncher of the CPU. The arithmetic/logic unit performs all the mathematical calculations of the CPU, including add, subtract, multiply, divide, and other operations on binary numbers.

– **Control Unit**: this component is responsible for directing the flow of instructions and data within the CPU. The control unit is actually built of many other selection circuits such as decoders and multiplexors. In the diagram, the decoder and the multiplexor compose the control unit.

A microprocessor executes a collection of machine instructions that tell the processor what to do. Based on the instructions, a microprocessor does three basic activities:

– Using its ALU (arithmetic/logic unit), a microprocessor can perform mathematical operations like addition, subtraction, multiplication, and division. Modern microprocessors contain complete floating-point processors that can perform extremely sophisticated operations on large floating-point numbers.

– A microprocessor can move data from one memory location to another.

– A microprocessor can make decisions and jump to a new set of instructions based on those decisions.

To support these basic activities, the processor architecture includes the following:

– An address bus (that may be 8, 16, or 32 bits wide) that sends an address to memory

– A data bus (that may be 8, 16, or 32 bits wide) that can send data to memory or receive data from memory

– An RD (read) and WR (write) line to tell the memory whether it wants to set or get the addressed location

– A clock line that lets a clock pulse sequence the processor

– A reset line that resets the program counter to zero (or whatever) and restarts execution

The processor can perform a large set of instructions. The collection of instructions is implemented as bit patterns, each one of which has a different meaning when loaded into the instruction register. A set of short words are defined to represent the different bit patterns. This collection of words is called the *assembly language* of the processor. An *assembler* can translate the words into their bit patterns very easily, and then the output of the assembler is placed in memory for the microprocessor to execute.

Examples of assembly language instructions for the Intel x86 processors are as follows [85]: ADC (add operation with carry), ADD (add operation), AND (logical AND operation), CLI (clear interrupt flag), CMP (compare operands), DEC (decrement by 1), DIV (unsigned divide operation), IN (input from data port), INC (increment by 1), INT (call interrupt), JMP (jump), LEA (load effective address
operation), MOV (move), MUL (unsigned multiplication operation), NOT (logical NOT operation), OR (logical OR operation), PUSH (push data into stack), RET (return from procedure), SHL (shift left operation), SUB (subtraction operation), or XOR (exclusive OR logical operation). The list of instructions that can be executed by a processor is called instruction set architecture, shortly ISA.

Based on the instruction set style, the processors can be classified in CISC (complex instruction set computer) and RISC (reduced instruction set computer). The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible. This is achieved by building processor hardware that is capable of understanding and executing a series of operations. One of the primary advantages of this system is that the compiler has to do very little work to translate a high-level language statement into assembly. Because the length of the code is relatively short, very small memory capacity is required to store instructions. The complex instructions are built directly into the hardware.

RISC processors only use simple instructions that can be executed within one clock cycle. Because there are more lines of code, more memory is needed to store the assembly-level instructions. The compiler must also perform more work to convert a high-level language statement into code of this form. These RISC “reduced instructions” require less transistors of hardware space than the complex instructions, leaving more room for general-purpose registers. Because all of the instructions execute in a uniform amount of time (i.e., one clock), pipelining is also possible.

Another type of classification of the processors takes into account the amount of data being processed and the number of instructions being executed. The Flynn’s taxonomy divides the processors into four categories [50]:

- Single instruction, single data (SISD), also known nowadays as a RISC processor. In this case, a single stream of instructions operates on a single set of data. Instructions are executed sequentially, but may be overlapped by pipelining. Most of the SISD systems are now pipelined.
- Single instruction, multiple data (SIMD). These machines include several interconnected processing elements, each with its own data, but under the supervision of a single control unit. All the processing elements perform the same operations on their data in lockstep. Thus, the execution of the instructions is synchronous. A single program counter can be used to describe the execution of all the processing elements.
- Multiple instruction, multiple data (MIMD). In this case, several processing elements have their own data and their own program counters. The tasks executed by different processors can start or finish at different times. The programs do not have to run in lockstep.
- Multiple instruction, single data (MISD). These machines have many processing elements, all of which execute independent stream of instructions, but on the same data stream.
2.4 Hardware Components

2.4.1 General-Purpose Processor

The general-purpose processors can be found in laptop, desktop, or server computers. These processors have very high performance and designed to work well in a variety of contexts. They support most of the popular Windows, Linux, and real-time operating systems. They run a wide range of application systems and are relatively inexpensive for high-end applications [59].

An example of well-known general-purpose processor with both desktop and embedded system markets is x86 from Intel [85].

2.4.1.2 Application-Specific Instruction Set Processor

An application-specific instruction set processor (ASIP) is a stored memory CPU whose architecture is tailored for a particular set of applications. This specialization of the processor core provides a trade-off between the flexibility of a general-purpose CPU and the performance of a DSP. The ASIP exploits special characteristics of the application to meet the desired performance, cost, and power requirements. The programmability of the ASIP allows changes to the implementation, use in several different chips, and high data path utilization. The application-specific architecture provides smaller silicon area and higher computation speed [171].

Compared to general-purpose processors, usually the ASIPs are enhanced with the following features:

– Special-purpose registers and buses to provide the required computations without unnecessary generality.
– Special-purpose function units to perform long operations in fewer clock cycles.
– Special-purpose control for instructions to execute common combinations in fewer clock cycles.

Some ASIPs have a configurable instruction set. Usually, these cores are divided into two parts: static logic, which defines a minimum ISA, and configurable logic which can be used to design new instructions. The configurable logic can be programmed either in the field in a similar fashion to an FPGA, dynamically reconfigured during execution, or during the chip synthesis.

Generally, the ASIP design relies on automatic tools. Usually these tools start from a set of characteristics of the target application domain and the required execution profiling (e.g., number of execution clock cycles for a specific application function). Then, the automatic tools generate both micro-architecture for the ASIP core and an optimized compiler targeted to the synthesized ASIP. Finally, the application is implemented using the generated ASIP core and ASIP compiler. Thus, the ASIP design consists of two main steps: processor synthesis and compiler design. The processor synthesis consists of choosing an instruction set, optimizing the data path, and extracting the instruction set from the register transfer design. The compiler design consists of driving the compilation from a parametric description of the
data path, binding values to registers, selecting instructions for the code matched to
the parameterized architecture, and scheduling the processor instructions.

Example of ASIP processors is the Xtensa processor from Tensilica [152]. The
Xtensa processors are synthesizable processors that are configurable and extensible.
The processor can be configured to fit the application by selecting and configuring
predefined elements of the architecture, such as the following:

- **Instruction set**: ALU extensions, co-processors, wide instructions, DSP style,
  function unit implementation
- **Memory**: instruction cache configuration, data cache configuration, memory pro-
  tection/translation, address space size, mapping of special purpose memories,
  DMA access
- **Interface**: bus width, bus access protocol, system registers access, JTAG, queue
  interfaces to other processors
- **Peripherals**: timers, interrupts, exceptions, and remote debug procedures

Additionally, the designers can optimize the processor by inventing completely
new instructions and hardware execution units that can deliver high performance.
The new instruction sets can be defined using the TIE language, which offers support
to new state declarations, new instruction encodings and formats, and new operation
descriptions. The TIE instructions can be manually written or automatically gener-
ated by using the XPRESS compiler, a tool which identifies which functions of a
C/C++ application need to be accelerated in hardware.

Once the designer determines the optimal configuration and extensions, the
Xtensa processor generator automatically generates a synthesizable hardware
description as well as a complete optimized software development environment
[152].

An example of commercial ASIP design tool is the Coware Processor Designer,
which represents an integrated design environment for unified application-specific
processor, programmable accelerator design, and software development tool gener-
ation [41].

The key to processor design’s automation is its Language for Instruction Set
Architectures, shortly LISA. In contrast to SystemC, which has been developed for
efficient specification of systems, LISA is a processor description language that
incorporates processor-specific components, such as register files, pipelines, pins,
memory and caches, and instructions. The LISA language enables the efficient cre-
ation of a single “golden” processor specification as the source for the automatic
generation of the instruction set simulator (ISS) and the complete suite of software
development tools, like assembler, linker, and C-compiler, and synthesizable RTL
code. An example of LISA code is illustrated in Fig. 2.5.

The development tools, together with the extensive profiling capabilities of the
debugger, enable rapid analysis and exploration of the application-specific processor’s instruction set architecture to determine the optimal instruction set for the
2.4 Hardware Components

OPERATION add
{
  DECLARE { GROUP dest, src1, src2 = { register }; }
  CODING { dest src2 src1 0b01000010000 }
  SYNTAX { "ADD" ".D" src1 "," src2 "," dest }
  BEHAVIOR { dest = src1 + src2; }
}

OPERATION register
{
  DECLARE { LABEL index; }
  CODING { index:0bx[4] }
  SYNTAX { "A" index:#U }
  EXPRESSION { A[index] }
}

Fig. 2.5 Sample LISA modeling code

target application domain. Processor designer enables also the designer to optimize instruction set design, processor micro-architecture and memory subsystems, including caches [41].

Another example of commercial ASIP design tool is provided by Target Compiler Technologies IP designer tool suite [151]. The design starts from descriptions of the processor architecture and the instruction set, using a high-level definition language, called nML. Then, based on the ASIP description and the targeted application, the Chess tool automatically maps the C application into optimized machine code of the target ASIP. The Checkers tool generates automatically the instruction set simulator and the graphical software debugger. The Go tool of the tool suite produces the synthesizable RTL architecture model of the ASIP from the nML processor description. Darts is used as assembler and disassembler of the ASIP that translates the machine code into binary format and vice versa.

2.4.1.3 Digital Signal Processor

The digital signal processor (DSP) is a specialized microprocessor designed specifically for digital signal processing [5].

Digital signal processing algorithms typically require a large number of mathematical operations to be performed quickly on a set of data. Signals are converted from analog to digital, manipulated digitally, and then converted again to analog form. Many DSP applications have constraints on latency; that is, for the system to work, the DSP operation must be completed within some time constraint. Most general-purpose microprocessors and operating systems can execute DSP algorithms successfully. But these microprocessors are not suitable for application of mobile telephone and pocket PDA systems, etc., because of power supply and space limit. A specialized digital signal processor, however, will tend to provide a lower cost solution, with better performance and lower latency.
The architecture of a digital signal processor is optimized specifically for digital signal processing applications. They usually provide parallel multiply and add operations, multiple memory accesses to fetch two operands and store the result, lots of registers to hold data temporarily, efficient address generation for array handling, and special features such as delays or circular addressing.

Examples of commercial DSPs are provided by Freescale [52], Texas Instruments [156], or Atmel [9].

A key player at the high-end signal processor manufacturer is Freescale. The company provides the fully programmable MSC8144 multi-core DSP architecture, which is based on next-generation StarCore™ technology (Fig. 2.6) [52]. Freescale’s MSC8144 is helping to make IP-based connections faster, easier, and more reliable by taking advantage of one programmable DSP platform that supports VoIP/data, video, and wireless standards using multiple software implementations. The MSC8144 DSP combines four programmable SC3400 StarCore™ DSP cores. Each SC3400 StarCore™ DSP core runs at 1 GHz. The quad-core DSP delivers the equivalent performance of a 4 GHz single-core DSP [52].

![Fig. 2.6 The Freescale MSC8144 SoC architecture with quad-core DSP](image)

Other examples of DSP are the C6000 DSP series from Texas Instruments. These series are fix point DSPs, which may operate at the clock frequency up to 1.2 GHz. They implement separate instruction and data caches with 2 MB second-level cache (L2). The access to the I/O is fast thanks to its 64 DMA (direct access memory) channels. The top models are capable of even 8,000 MIPS (million instructions per second), use VLIW (very long instruction word) encoding, perform eight operations per clock cycle, and are compatible with a broad range of external peripherals and various buses (PCI/serial/etc.) [156].

### 2.4.1.4 Microcontroller

The microcontroller (MCU) is a special processor used for control functions. Microcontrollers are embedded inside consumer products, so that they can control...
the features or actions of the product. Another name for a microcontroller, therefore, is *embedded controller*. The microcontrollers are dedicated for the execution of one application task. Usually they are small and low power devices.

A microcontroller may have many low power modes, depending on the application. There are numerous methods that the microcontroller can use to lower the static power consumed by the devices in standby or low power mode (also called standby or leakage power). These include low-leakage transistors and turning off the power to various parts of the MCU. Usually, the deeper asleep the device is, the longer it takes to wake up. Wakeup time becomes an important consideration when determining how low power modes are implemented.

Some microcontrollers provide digital signal processing functionality. Processor cores for control are different from those that perform very complex mathematical functions. Cores that perform both functions are blurring that line.

In some cases, DSP-like mathematical functions are being added to a regular core’s instruction set, with the hardware to support it. And the opposite is occurring too, as DSP cores add control-like instructions.

An alternative is to embed both a controller and a DSP core in the same device, creating a hybrid. Whether these devices are considered microcontrollers with DSP functionality or DSPs with microcontroller functionality is up to the vendor to decide.

Examples of microcontrollers are the 16-bit MSP430 from Texas Instruments [156], AVR from Atmel [9], the PIC microcontroller families from Microchip [103], or the 32-bit ARM Cortex-M3 or the 8-bit 8051 [6] or those provided by Freescale for automotive applications [52].

### 2.4.2 Memory

The memory is a hardware component used to store data. The basic unit of storage is called memory cell [171]. Cells are arranged in a 2D array to form the memory circuit. Within the memory core, the cells are connected to row and bit (column) lines that provide a 2D addressing structure. The row line selects a one dimensional row of cells, which then can be accessed (written or read), via their bit lines. The memory may have multiple ports to accept multiple addresses and data for simultaneous read and write operations.

The memories can be classified into two main types: RAM (random access memory) and ROM (read-only memory). Traditional RAM memories store data that can be read and written in a random order. They are usually volatile memories, meaning that their content is lost after turning off the power. The ROM memories store data that cannot be modified (at least not quickly or easily). They are mainly used to store firmware, software very closely tied to the hardware.

The RAM family includes two important memory devices: static RAM (SRAM) and dynamic RAM (DRAM). The primary difference between them is the lifetime of the data they store. SRAM retains its contents as long as electrical power is applied
to the chip. If the power is turned off or lost temporarily, its contents will be lost forever. DRAM, on the other hand, has an extremely short data lifetime – typically about few milliseconds, even when power is applied constantly [14]. The DRAM can behave like SRAM if a piece of hardware called a DRAM controller is used. The job of the DRAM controller is to periodically refresh the data stored in the DRAM. By refreshing the data before it expires, the contents of memory can be kept alive for as long as they are needed.

SRAM devices offer extremely fast access times (approximately four times faster than DRAM) but are much more expensive to produce. Generally, SRAM is used only where access speed is extremely important. A lower cost per byte makes DRAM attractive whenever large amounts of RAM are required. Many embedded systems include both types: a small block of SRAM (a few kilobytes) along a critical data path and a much larger block of DRAM (perhaps even megabytes) for other types of data.

The NVRAM (non-volatile RAM) is an SRAM memory with a battery backup. When the power is turned on, the NVRAM operates like an SRAM. When the power is turned off, the NVRAM uses the battery to retain its data. NVRAM is common in embedded systems. However, it is more expensive than SRAM, because of the battery. So the applications are typically limited to the storage of a few hundred bytes of system-critical information that cannot be stored in any other type of memory.

Memories in the ROM family are distinguished by the methods used to write new data to them (usually called programming) and the number of times they can be rewritten. This classification reflects the evolution of ROM devices from hardwired to programmable to erasable and programmable. A common feature of all these devices is their ability to retain data and programs forever, even during a power failure.

The very first ROMs were hardwired devices that contained a preprogrammed set of data or instructions. The contents of the ROM had to be specified before chip production, so the actual data could be used to arrange the transistors inside the chip. Hardwired memories are still used, though they are now called “masked ROMs” to distinguish them from other types of ROM. The primary advantage of a masked ROM is its low production cost. Unfortunately, the cost is low only when large quantities of the same ROM are required.

Another type of ROM is the PROM (programmable ROM), which is purchased in an unprogrammed state (the data are made up entirely of bits with value equal to 1). The process of writing your data to the PROM involves a special piece of equipment, called device programmer. The device programmer writes data to the device one word at a time by applying an electrical charge to the input pins of the chip. Once a PROM has been programmed in this way, its contents can never be changed. If the code or data stored in the PROM must be changed, the current device must be discarded. As a result, PROMs are also known as one-time programmable (OTP) devices.

An EPROM (erasable-and-programmable ROM) is programmed in exactly the same manner as a PROM. However, EPROMs can be erased and reprogrammed repeatedly. To erase an EPROM, the device is exposed to a strong source of
ultraviolet light. Thus, the entire chip is reset to its initial and unprogrammed state. The EPROMs are more expensive than PROMs, but they are essential for the software development and testing process.

EEPROM (electrically erasable-and-programmable ROM) is similar to EPROM, but the erase operation is accomplished electrically, rather than by exposure to ultraviolet light. Any byte within an EEPROM may be erased and rewritten. Once written, the new data will remain in the device forever or until it is electrically erased. The primary trade-off for this improved functionality is higher cost, though write cycles are also significantly longer than writes to a RAM.

Flash memory combines the best features of the memory devices described thus far. Flash memory devices are high density, low cost, non-volatile, fast (to read, but not to write), and electrically reprogrammable. The use of flash memory has increased dramatically in embedded systems. From a software point of view, flash and EEPROM technologies are very similar. The major difference is that flash devices can only be erased one sector at a time, not byte by byte. Typical sector sizes are in the range of 256 bytes to 16 kB. Despite this disadvantage, flash is much more popular than EEPROM.

Another frequently used type of memory is the cache. The cache plays a key role in reducing the average memory access time of a processor. It also decreases the bandwidth requirement each processor places on the shared interconnect and memory hierarchy. The cache is located near the processor, thus it offers a very fast access time (Fig. 2.7). It replicates parts of the data stored in the main memory, i.e., the most often used or the latest used memory blocks, depending on the cache policy. In cache-based SoC, every time a processor requires data from the main memory, first it checks whether the data are already in the cache. In this case, called cache hit, the data are directly retrieved from the cache, thus avoiding the transfer through the global interconnect component. If the data are not stored in the cache, situation called cache miss, the data are retrieved from the main memory and eventually stored in the local cache for a latter access.

![Fig. 2.7 Cache and scratch pad memory](image-url)
But the use of cache memories raises another important issue: the cache coherence. The cache coherence problem appears in architectures made of multiple processors with their own cache memories. The problem arises when a memory block is present in the caches of one processor or more processors, and another processor modifies that memory block. Unless special action is taken, the other processors continue to access the old copy of the block that it is in their caches [42]. The algorithms used to maintain the cache coherence are implemented in hardware. The hardware decides when values are added or removed from the cache [171].

As an alternative to the hardwired algorithms to manage close-in memory, software-oriented approaches have been proposed. The scratch pad memories represent another type of memory, similar to cache memories, but which require software algorithms for their management (Fig. 2.7). The scratch pad memory is a high-speed memory, located near the processor. It does not include hardware to manage its contents. The CPU can access the scratch pad to read and write directly, because the scratch pad is part of the address space of the main memory. Therefore, its access time is predictable, unlike cache accesses. The software is responsible to manage which data are in the scratch pad or which data need to be removed. Usually the software manages the scratch pad by combining compile-time information with run-time decision making.

Memory is a key bottleneck in embedded systems. Many embedded computing applications spend a lot of their time accessing memory. The memory hierarchy is a prime determinant of not only performance but also energy consumption. So optimizing the memory system becomes crucial. There are several techniques of memory optimization which target either data or instructions, e.g., loop transformations to express data parallelism, dataflow transformations to improve memory utilization, minimal buffer size usage, or optimal scratch pad memory allocation algorithms. The access time of the main memory can be reduced by using burst modes to access a sequence of memory locations, or by integrating paged memories to take advantage of the properties of memory components to reduce the access times, or banked memories, which are systems of memory components that allow parallel accesses.

2.4.3 Interconnect

The interconnect component is a shared resource between various hardware components. Its role is to transmit data from a source hardware resource to a destination hardware resource, thus implementing the communication network.

The network component can be profiled using two types of measurements: latency and bandwidth. The latency represents the total time required to transfer \( n \)-bytes of information from the source to the destination component. The bandwidth represents the amount of data (number of bytes) that can be delivered by the communication network per second. It is desirable for the interconnection network to provide high bandwidth, because higher bandwidth decreases the occupancy of the shared interconnect component. Thus, it can reduce the likelihood of network
contention. High bandwidth also allows for the software to exchange large volume of data without waiting for individual data units to be transmitted along the way.

The performance constraints for MPSoC architectures place a requirement on the data bandwidth the network must deliver, if the processors are to sustain a given computational rate. However, this load varies considerably between the applications. The flow of information may be physically localized or dispersed. The data may be transmitted in burst mode or fairly uniformly in time. In addition, the waiting time of the processor is strongly affected by the latency of the network. The time spent waiting affects the bandwidth requirement.

The design of the interconnect component is a complex process for system-on-chip. Any communication failure, whether due to noise or an error in timing or protocol, is likely to require a design iteration that will be expensive in both mask charges and time to market [53].

Early SoCs used an interconnect paradigm inspired by the rack-based microprocessor systems of earlier days. In those rack systems, a backplane of parallel connections formed a “bus” into which all manner of cards could be plugged. A system designer could select cards from a catalogue and simply plug them into the rack to yield a customized system with the processor, memory, and interfaces.

In a similar way, a designer of an early SoC could select hardware IP blocks, place them onto the silicon, and connect them together with a standard on-chip bus (OCB) (Fig. 2.8). The backplane might not be apparent as a set of parallel wires on the chip, but logically the solution is the same. The on-chip bus connects a central processor and standard components like memory, peripherals, interrupt units plus some application-specific components. Among the advantages of this approach we have power savings, higher integration density, lower systems costs, easier procurement, etc.

**Fig. 2.8** SoC architecture based on system bus

Among the different approaches to the interconnect concept, Virtual Socket Initiative Alliance (VSIA) [168] merits special remark for the effect in standardizing criteria for concepts, methods, and allowed interoperability.

The most popular SoC approach has been the ARM processor strategy [6]. ARM disposes of a complete family of RICS processors, with the AMBA OCB. AMBA (Advanced Microcontroller Bus Architecture) is currently one of the most widely
used systems bus architectures for SoC applications (even for processors other than ARM).

However, buses do not scale well. With the rapid increase in the number of hardware components to be connected and the increase in performance demands, today’s SoCs cannot be built around a single bus. Instead, complex hierarchies of buses are used (as illustrated in Fig. 2.9), with sophisticated protocols and multiple bridges between them. The communication between remote hardware IPs can go via several buses. Thus, a SoC can implement more than one bus.

![Fig. 2.9 SoC architecture based on hierarchical bus](image)

A typical SoC architecture is comprised of one high-performance bus for memory access and high-speed peripherals and one low-speed bus for low-speed peripherals like UARTs. Examples of buses that can be linked through bridges are those defined within the AMBA specification: advanced high-performance bus (AHB), advanced system bus (ASB), and advanced peripheral bus (APB) [6].

AHB bus is devoted to high-performance communication, for system modules requiring high clock frequencies. This bus acts as the backbone bus. It is intended for connection of processors and coprocessors, DSP units, on-chip memories, and off-chip external memory interfaces. The ASB is mainly deprecated and it has been substituted by AHB. The APB bus is devoted to low-speed peripherals and it is optimized to minimize power consumption and to reduce interface complexity. APB is designed to be used in conjunction with a system bus (AHB/ASB).
The hierarchical bus-based systems have no limitations about the number of buses and its hierarchy. These systems are more flexible and powerful than single bus systems, as they allow any number of CPUs.

Where bus-based solutions reach their limit, packet-switched networks are poised to take over (Fig. 2.10) [53]. A packet-switched network offers flexibility in topology and trade-offs in the allocation of resources to clients. A network on chip (NoC) is constructed from multiple point-to-point data links interconnected by switches (also called routers). The data messages can be relayed from any source module to any destination module over several links, by making routing decisions at the switches [17]. A NoC is similar to a modern telecommunications network, using digital bit-packet switching over multiplexed links. Although packet-switching is sometimes claimed as necessity for a NoC, there are several NoC proposals utilizing circuit-switching techniques.

**Fig. 2.10** SoC architecture based on packet-switched network on chip

In NoC-based architectures, the IP blocks communicate over the NoC using a five-layered communication scheme, similar to the OSI transmission protocol, as illustrated in Fig. 2.11: application, transport, network, data link, and physical layers.

The application layer corresponds to the communication primitives used by the application for the data exchange. The units of communication consist of data messages.

The transport layer establishes and maintains end-to-end connections. It performs packet segmentation and reassembling and ensures message ordering. The units of communication are packets. The transport layer defines rules that apply as packets are routed through the switch fabric. The packets can include byte enables, parity information, or user information depending on the actual application requirements.

The network layer defines how data are transmitted over the network from a sender to a receiver, such as routing algorithm. The units of communication are flits.
The data link layer defines a protocol to transmit the information between the entities. It may include flow control and error correction. The units of communication in this layer are expressed in bits or words.

The physical layer defines how packets are physically transmitted over an interface. It also determines the number and length of wires connecting the IP blocks and switches. The units of communication at this level are electronic signals.

Examples of NoC interconnect components are Spidergon [40] or the Hermes NoC [107].

### 2.5 Software Layers

Programmable hardware components are important in a re-usable architectural platform, since it is very cost-effective to tailor a platform to different applications by simply adapting the low-level software and maybe only configuring certain hardware parameters, such as memory sizes and peripherals.

As illustrated in Fig. 2.3, the software view of an embedded system shows three different layers:

- The bottom layer, named hardware abstraction layer or shortly HAL, is comprised of services directly provided by hardware components (processor and peripherals) such as instruction sets, memory and peripheral accesses, and timers. It also includes instance of device drivers, boot code, parts of a real-time operating system (RTOS), such as context-switching code and configuration code to access the MMU (memory management unit), and even some domain-oriented algorithms that directly interact with the hardware.
- The top layer is the multitasking application software, which should remain completely independent from the underlying hardware platform.
- The middle layer is comprised of three different components:
(a) Hardware-independent software, typically high-level RTOS services, such as task scheduling or interrupt service routines;
(b) Communication layer, which implements the high-level communication primitives and offers support for specific I/Os;
(c) The API (application programming interface), which defines a system platform that either isolates the application software from the hardware-dependent software (HdS) (HdS API) or separate the middle layer from all basic low software layer (HAL APIs), enabling their concurrent design.

The standardization of these APIs, which can be seen as a collection of services usually offered by an operating system, is essential for software re-use above and below it. At the application software level, libraries of re-usable software IP components can implement a large number of functions that are necessary for developing systems for given application domains. If, however, one tries to develop a system by integrating application software components that do not directly match a given API, software retargeting to the new platform will be necessary [166]. This can be a very tedious and error-prone manual process, which is a candidate for an automatic software synthesis technique.

Nevertheless, re-use can also be obtained below the API. Software components implementing the hardware-independent parts of the RTOS can be more easily re-used, especially if the interface between this layer and the HAL layer is standardized. Although the development of re-usable HAL may be harder to accomplish, because of the diversity of hardware platforms, it can be at least obtained for platforms aimed at specific application domains.

There are many academic and industrial alternatives providing RTOS services. The problem with most approaches, however, is that they do not consider specific requirements for SoC, such as minimizing memory usage and power consumption. Recent research efforts propose the development of application-specific RTOS containing only the minimal set of functions needed for a given application [55] or including dynamic power management techniques. Embedded software design methodologies should thus consider the generation of application-specific RTOS that are compliant to a standard API and optimized for given system requirements.

The hardware-dependent software part of the software stack is usually comprised of three components called (from lower layer to upper layer) hardware abstraction layer, middleware (or communication layer), and operating system (Fig. 2.3). These three components manage the execution of tasks running on processors and the use of shared resources, including hardware resources. The lowest layer, the hardware abstraction component, is separated from the operating system to facilitate the porting of the OS from one processor to another, which facilitates the porting of an application from one processor to another. Usually, few parts of the HAL are written in assembly, as they are closely linked with the underlying processor. The middleware component includes communication primitives used by the application tasks through the OS, and based on device drivers provided by the HAL component.
This structure by layers (or components) maintains the separation of skills for the development of the software stack. The HAL is usually provided with the hardware or developed by someone who knows deeply the hardware. The OS is developed depending on the main characteristics awaited (scheduling algorithm, real time, symmetric multi-processor, etc.). The communication component makes the link between OS and HAL. The application is developed by the system engineer.

### 2.5.1 Hardware Abstraction Layer

In the following section, several examples of existing commercial HAL are given. These examples of HAL are used in both academic and semiconductor industry areas.

Even if the HAL represents an abstraction of the hardware architecture, since it has been mostly used by OS vendors and each OS vendor defines its own HAL, most of the existing HAL is OS dependent. In case of an OS-dependent HAL, the HAL is often called board support package (BSP). In fact, the BSP implements specific support code for a given hardware platform or board, corresponding to a given OS. The BSP includes also a boot loader, which contains a minimal device support to load the OS and device drivers for all the devices on the hardware board.

The embedded version of the Windows OS, namely, Windows CE, provides BSP for many standard development platforms that support several microprocessors family (ARM, x86, MIPS) [104]. The BSP contains an OEM (original equipment manufacturer) adaptation layer (OAL), which includes a boot loader for initializing and customizing the hardware platform, device drivers, and a corresponding set of configuration files.

The VxWorks OS offers BSP for a wide range of MPSoC architectures, which may incorporate ARM, DSP, MIPS, PowerPC, SPARC, Xscale, and other processors family [170].

In eCos, a set of well-defined HAL APIs are presented [46]. However, there is no clear difference between HAL and device driver. Examples of HAL APIs used by eCos are as follows:

- Thread context initialization: \texttt{HAL\_THREAD\_INIT\_CONTEXT()}
- Thread context switching: \texttt{HAL\_THREAD\_SWITCH\_CONTEXT()}
- Breakpoint support: \texttt{HAL\_BREAKPOINT()}
- GDB support: \texttt{HAL\_SET\_GDB\_REGISTERS()}, \texttt{HAL\_GET\_GDB\_REGISTERS()}
- Interrupt state control: \texttt{HAL\_RESTORE\_INTERRUPTS()}, \texttt{HAL\_ENABLE\_INTERRUPTS()}, \texttt{HAL\_DISABLE\_INTERRUPTS()}
- Interrupt controller management: \texttt{HAL\_INTERRUPT\_MASK()}
- Clock control: \texttt{HAL\_CLOCK\_INITIALIZE()}, \texttt{HAL\_CLOCK\_RESET()}, \texttt{HAL\_CLOCK\_READ()}
– Register read/write: HAL_READ_XXX(), HAL_READ_VECTOR_XXX(), 
   HAL_WRITE_XXX(), and HAL_WRITE_VECTOR_XXX()
– Control the dimensions of the instruction and data caches: HAL_ 
   XCACHE_SIZE(), HAL_XCACHE_LINE_SIZE()

In the software development environment for the Nios II processor provided 
by Altera [3], the HAL serves as a device driver package, providing a consis-
tent interface to the system peripherals, such as timers, Ethernet MAC, and I/O 
peripherals.

In Real-Time Linux a HAL, called real-time HAL (RTHAL), is defined to give 
an abstraction of the interrupt mechanism to the Linux kernel [135]. It consists of 
three APIs for disabling and enabling interrupts and return from the interrupt.

An example of HAL that does not depend on the targeted OS is the a386 library 
[1]. The a386 represents a C library which offers an abstraction of the Intel 386 
processor architecture. The functions of the library correspond to privileged pro-
cessor instructions and access to the hardware. The library serves as a minimal 
hardware abstraction layer for the OS. Later, the library is ported on ARM and 
SPARC processors.

2.5.2 Operating System

The complexity of the application and the increasing capabilities of hardware 
arichedrues have moved embedded software from simple sequential program 
to concurrent complex system with specific software architecture. This is why 
embedded systems require some kind of OS to manage the processor (or several 
processors) and hardware resources efficiently. The OS can be seen as an abstraction 
of the hardware (processor and resources) used by the software tasks. This abstrac-
tion consists in sharing these hardware resources available between tasks and goes 
to complete virtualization of the hardware architecture for which the number of 
resources is infinite.

A multiprocessor architecture can run one OS (usually called centralized OS) 
in only one processor (the others are seen as co-processor) in heterogeneous or 
asymmetric architecture or in any of the free processors in a symmetric architecture. 
If one OS is running on each processor (distributed OS), it could be the same copy 
(homogeneous architecture) or a different copy (heterogeneous architecture). This 
leads to different performances and different capabilities the software designer has 
to deal with.

A lot of commercial OSs (see section 2.3.2.2) exist, but an OS can be developed 
for a particular class of applications or with specific features. In both cases, the 
software designer has to configure the OS in case of embedded systems to reduce 
the memory footprint including communication and synchronization services (such 
as messages, semaphores, clocks) and tasks scheduling policy.
The ASOG (application-specific operating system generator) tool of the ROSES flow [32] allows generation of an application-specific OS, also known as software interfaces or software wrappers, for each processor. The following paragraphs give more details about this OS generator tool.

As shown in Fig. 2.12, the software wrappers provide the implementation of the high-level communication primitives (available through the APIs) used in the system specification and the drivers to control the hardware. If required, the wrapper will also provide sophisticated OS services, such as tasks scheduling and interrupt management, minimally tailored for a particular application.

**Fig. 2.12** Software wrapper

The synthesis of wrappers is based on libraries of basic modules from which dedicated OSs are assembled. These libraries may be easily extended with modules that are needed to build software wrappers for any type of processors, memories, and other components that follow various bus and core standards.

The software wrapper generator [55] produces a custom OS for each processor on the target platform. The software wrapper generator produces operating systems streamlined and pre-configured for the software module(s) that run(s) on each target processor. It uses a library organized in three parts: APIs, communication/system services, and device drivers. Each part contains elements that will be used in a given software layer in the generated OS. The generated OS provides services: communication services (e.g., FIFO communication), I/O services (e.g., AMBA bus drivers), memory services (e.g., cache or virtual memory usage), etc. Services have dependency between them, for instance, communication services are dependent on I/O services. Elements of the OS library also have dependency information. This mechanism is used to keep the size of the generated OS at a minimum; the elements that provide unnecessary services are not included.

There are two types of service codes: re-usable (or existing) code and expandable code. As an example of existing code, AMBA bus-master service code can exist in the OS library in the form of C language. As an example of expandable code, OS kernel functions can exist in the OS library in form of macrocode (m4 like). There are several preemptive schedulers available in the OS library such as round-robin scheduler, or priority-based scheduler. In the case of round-robin scheduler, time slicing (i.e., assigning different CPU load to tasks) is supported. To
make the OS kernel very small and flexible (1) the task scheduler can be selected from the requirements of the application code and (2) a minimal amount (less than 10% of kernel code size) of processor-specific assembly code is used (for context switching and interrupt service routines).

The software component interfaces must be composed using basic elements of the software wrapper generators library. Table 2.4 lists some API functions available for different kinds of software task interfaces. The application tasks must communicate through API functions provided by the software wrapper generator library. For instance, the shared memory API (SHM) provides read/write functions for inter-task communication. The guarded shared memory API (GSHM) adds semaphore services to the SHM API by providing lock/unlock functions.

<table>
<thead>
<tr>
<th>Basic component interfaces</th>
<th>API functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Put/get</td>
</tr>
<tr>
<td>Signal</td>
<td>Sleep/wakeup</td>
</tr>
<tr>
<td>FIFO</td>
<td>Put/get</td>
</tr>
<tr>
<td>SHM</td>
<td>Read/write</td>
</tr>
<tr>
<td>GSHM</td>
<td>Lock/unlock/read/write</td>
</tr>
</tbody>
</table>

A recurrent problem in library-based approaches is library size explosion. In the ROSES flow, this problem is minimized by the use of layered library structures where a service is factorized so that its implementation uses elements of different layers. This scheme increases re-use of library elements since the elements of the upper layers must use the services provided by the elements in the immediate lower layer [32]. The designers are able to extend the ROSES libraries since they are implemented in an open format. This is an important feature since it enables the support of different standards while re-using most of the basic elements in the libraries.

The OS generation consists of assembling the required OS services. The services reside in a library made of a set of macrocodes files corresponding to each OS service. The macrocode files are written using a macrolanguage and serve to generate the customized files for the application-specific OS.

The services of the OS are assembled based on a service dependency graph, used to determine, select, and configure the services for the OS generation. The service dependency graph is described using a structural description language, called LiDeL (library description language) [55]. LiDeL is composed of a set of data structures manipulated by several APIs. The structural description language contains three types of items:

- **Elements**, which represent an OS part. The elements are basic components of an OS. They represent a non-specialized component, which is not yet dedicated for a particular architecture case.
-- Services, which represent system functionality. It is an abstract term, which allows dividing and structuring the behavior of an OS. The services are provided by elements, but an element may also require a service from another element.

-- Implementations, which represent a particular behavior description. An element can have multiple implementations. Each implementation corresponds to a part of the generic code of an OS.

The ASOG tool uses as input the system description, more precisely the partitioning and mapping information, the tasks code, the LiDeL library, and the services library written as macrocode. The Colif description contains the services needed by the application, along with the parameters needed for these services (Fig. 2.13) [32].

Fig. 2.13 Representation of the flow used by ASOG tool for OS generation

When application tasks require a service (i.e., services for the data exchange in the form of MPI communication), the ASOG tool starts crossing the service dependency graph from the required service down to the low-level services. Based on the crossed services, the ASOG will macrogenerate the files for the implementations.
of the elements associated with these services. The generated files are C or assembly code files. The ASOG also generates the required compilation Makefile scripts, along with some log files useful for debugging the OS generation process.

Table 2.5 shows some of the existing software components in the current ROSES IP library and gives the type of the communication they use in their interfaces.

<table>
<thead>
<tr>
<th>IP</th>
<th>Description</th>
<th>Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW</td>
<td>host-if</td>
<td>Register/signal</td>
</tr>
<tr>
<td>Rand</td>
<td>Random number generator</td>
<td>Signal/FIFO</td>
</tr>
<tr>
<td>mult-tx</td>
<td>Multipoint FIFO data transmission</td>
<td>FIFO</td>
</tr>
<tr>
<td>reg-config</td>
<td>Register configuration</td>
<td>Register/FIFO/SHM</td>
</tr>
<tr>
<td>shm-sync</td>
<td>Shared memory synchronization</td>
<td>SHM/signal</td>
</tr>
<tr>
<td>Stream</td>
<td>FIFO data streaming</td>
<td>GSHM/FIFO/signal</td>
</tr>
</tbody>
</table>

Table 2.5 Sample software IP library

Figure 2.14 shows the “stream” software IP and part of its code to demonstrate the utilization of the communication APIs. Its interface is comprised of four ports: two for the FIFO API (P3 and P4), one for the signal API (P2), and one for the GSHM API (P1). In line 7 of Fig. 2.14, the stream IP uses P1 to lock the access to the shared memory that contains the data that will be streamed. P2 is used to suspend

```c
1 void stream::stream_beh()
2 {
3   long int *P;
4   ...
5   for(;;)
6     {...
7     P=(long int*)P1.Lock();
8     P2.Sleep();
9     for (int i=0; i<8; i++)
10    {
11       long int val = P3.Get();
12       P4.Put(*(P+i+8));
13      ...;}...
14     P1.Unlock();
15  }
16 ...
17 }
```

Fig. 2.14 The stream software IP
the task that fills up the shared memory (line 8). Then, some header information is got from the input FIFO using P3 (line 11) and streamed to the output FIFO using P4 (line 12). When streaming is finished, P1 is used to unlock the access to the shared memory (line 14).

2.5.3 Communication and Middleware

The middleware points to the software component that links the application code and the data transfer done by the network. The use of this component or layer helps software designer in porting an application from one architecture to another. In embedded systems, middleware should provide API primitives for several communication schemes (send and receive, blocking or non-blocking, etc.). This API isolates the communication services requested by the application from the network.

The middleware as well as the OS should be configured and tuned specifically for the application/architecture.

2.5.4 Legacy Software and Programming Models

The legacy software and the legacy programming models at both high and low levels can be integrated in the software stack easily. The clear separation between the different software components through well-defined APIs allows using legacy codes. The integration consists of adding new interfaces between the supported components and relying the legacy software on the virtualization technology.

2.6 Conclusions

This chapter detailed the MPSoC hardware and software organization. The definitions of the main hardware components (CPU, memory, interconnect) and software components (applications tasks code, OS, HAL, communication) were given.

The layered organization of the software stack allows a gradual design performed in several steps which correspond to different abstraction levels (system architecture, virtual architecture, transaction accurate architecture, and virtual prototype). The software validation is performed by simulation using an abstract architecture model.

The next chapters will detail the software design and validation of each of these different abstraction levels.
Embedded Software Design and Programming of Multiprocessor System-on-Chip Simulink and System C Case Studies
Popovici, K.; Rousseau, F.; Jerraya, A.A.; Wolf, M.
2010, XV, 290 p. 134 illus., Hardcover
ISBN: 978-1-4419-5566-1