CHAPTER 2

Global State-of-the-Art Overview

Abstract

The work presented in this book targets nomadic battery operated embedded systems. In this context, a large amount of related work exists. The goal of this chapter is to present a structured overview of the relevant related work in the design of embedded systems, which forms the broad context. The presented ordering will cover both the architectural as well as the related mapping aspects. An overview will be presented of the state of the art for the different components that form an embedded system. Specific related work and comparisons the individual contributions of this book will be presented in the respective chapters.

The rest of this chapter is structured as follows: Section 2.1 present an overview of the architectural components, namely the processor core, Data Memory Hierarchy (DMH), Instruction/Configuration Memory Organization (ICMO) and the inter-core communication architecture. Section 2.2 introduces the related work on the architecture exploration over this space which forms a key aspect of the embedded systems design, together with the evaluation methods and relevant criteria or cost metrics. Finally Section 2.3 concludes this chapter.

2.1 Architectural components and mapping

Figure 2.1 shows the main components of an embedded system. At the heart of the system, the *processor core* performs the computations of the application. It operates on the application data, which are stored in the *Data Memory Hierarchy*. The DMH can consist of multiple individual memories of different sizes. The type of operations that need to be executed by the processor and their required order is stored in the *Instruction/Configuration Memory Organization*. As with the DMH, the ICMO can consist of multiple memories. Modern embedded systems often contain multiple processor cores and use an inter-core *communication architecture* to connect all components and enable the data transfer between different processors and memories.

The rest of this section will discuss the relevant related work for each of the components of Figure 2.1, including the techniques that are used to map applications efficiently to these architectural components.

2.1.1 Processor core

The processor core consists of the hardware that executes the operations (the datapath), the foreground memory from which the operands are loaded and to which the results are stored back and the local interconnection between these components. In addition to these components, the processor core also consists of other components like processor pipelining and the issue type. In this book, the foreground memory is defined as the memory to and from which reads and writes happen in the same cycle as the processing, e.g. register files, pipeline registers. Related work on these individual parts can be structured according to these components as shown in Figure 2.2. Based on the design decisions made for each component, processor styles can be defined. The state-of-the-art processors representative for those styles, will be categorized and described and the end of this subsection.

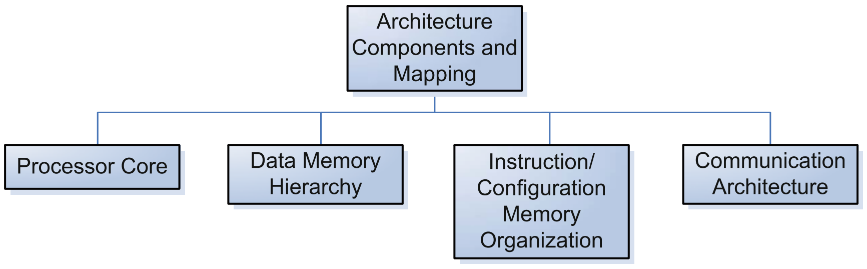
Architecture Components and Mapping
Processor Core
Data Memory Hierarchy
Instruction/ Configuration Memory Organization
Communication Architecture

Figure 2.1: Processor architecture space

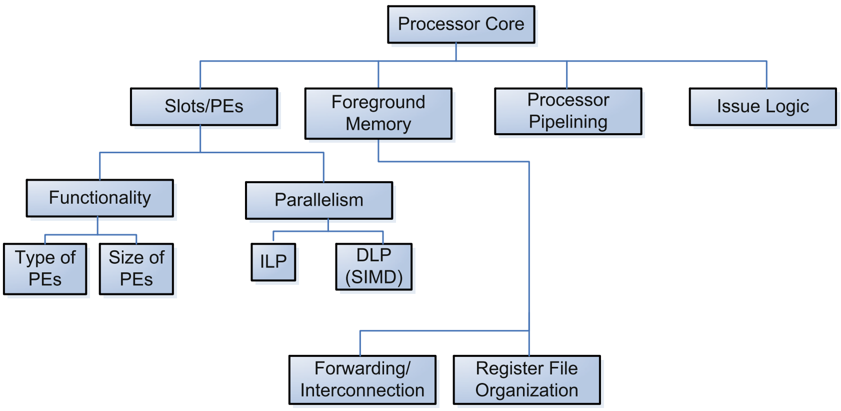

Figure 2.2: Processor core architecture space

2.1.1.1 The FUs, slots and PEs of the datapath

Depending on the community, the terms FU, slot and PE can be used to represent the same or different architectural components. Therefore a clear definition is presented below, that is followed throughout the rest of this book.

Functional Unit (FU) The hardware component that executes an operation of a specific type, e.g. and adder, multiplier or shifter. A single-issue processor can still contain multiple FUs, but then only one FU can start execution in a certain cycle.

Slot A group of FUs that are used mutually exclusively. Multiple (issue) slots can execute in parallel. The terms *slot* is commonly used in Very Large Instruction Word (VLIW) processor literature.

Processing Element (PE) Similar to a slot, but can also contain a local data or instruction storage. The term PE is commonly used in reconfigurable hardware literature.

Operation The functionality that is executed by an FU, e.g. ADD, MUL, SHIFT.

Instruction The binary representation that determines what operation will be executed by a single slot or PE. In VLIW literature, the VLIW-instruction is a concatenation of the instructions for all parallel slots.

In conclusion, the slots or PEs of a processor typically consists of different types of Functional Units and each FU supports a different functionality. The *instructions* decide what *operation* is executed on a certain slot and on the corresponding FU within that slot. Typically the combination of FUs into slots or PEs is called the processor *datapath*.

Processors can differ in the type and mix of FUs and in the number of slots. In order to be able to efficiently execute their target application domains the designer has to match these parameters to the requirements of the application. The set of operations that is supported by the FUs of a processor is called the *instruction set*. Various techniques exist that identify customized instruction set extensions and implement them [VPr94, Che05, Lee03, Yu04] . These techniques identify appropriate sub-graphs from the control-data flow graph that can be executed by a single customized FU and in a way that efficiently accelerates the application.

The number of slots of the processor determines the amount of instructions that can be issued in parallel (Instruction Level Parallelism or ILP). Various processors offer different levels of instruction level parallelism [TI00, Phi, VdW05]. A range of techniques exist to compile to architectures with multiple issue slots. For example modulo scheduling or hyper-blocking [Ron07, Rau94, Mah92] increase the number of instructions that can be scheduled together.

Irrespective of the number of parallel slots (the ILP), each of the different FUs of the processor can potentially support the execution of the same operation on multiple data words in parallel. This is called SIMD (Single Instruction Multiple Data) and is a form of Data Level Parallelism or DLP. Various processors offer different possible SIMD possibilities [Fre, Int, TI00, VdW05]. The individual data elements on which the operation is performed in parallel are called *sub-words* and together they form a *word*. In order to exploit the DLP that is present in the application, often the data layout (the order in which the data is stored in the memory) has to be modified and the data sub-words that will be operated in parallel have to be (re-)packed together into words. This process, together with the re-structuring of the application code to perform the data parallel execution is called vectorization or SIMDization. Various SIMDization and transformations techniques have been proposed in literature [Bou98, Xue07, OdB03, Lar05].

2.1.1.2 Foreground memory (or register files)

The second component of the processor core is the foreground memory, from which the data is loaded and to which the data is stored back after the execution. The *foreground memory organization* consists of the Register File (RF) and the connectivity between the RF and the FUs or between different FUs (forwarding/interconnection) (see Figure 2.2).

The biggest challenge when designing a register file is to balance the need to deliver data in parallel to all slots with the requirement to keep the number of ports low for a feasible implementation and higher energy efficiency [Rix00a, Lap02]. Clustering (splitting into parts) of the register file is a typical technique to allow delivering data in parallel to many slots in a scalable way without increasing the number of ports per register file. However the communication between the different clusters can have a negative effect on performance. An extensive study of various inter-cluster communication architecture is presented in [Gan07].

Other types of foreground memory organizations have also been proposed, ranging from hierarchical register files [Zal00a] to FIFO based register files [Tys01, Fer97]. Vector register files [Kap03, Asa98, Koz03] target data paths that provide data level parallelism (SIMD) and form another important class of foreground memory.

For each of these architectures a range of register allocation techniques exist [Zha02, Smi04, Cha82]. Some techniques like [Das06] target specific streaming models of register file as well.

In addition to the register file and the datapath, most state-of-the-art processors have special forwarding paths between the functional units. This allows more flexibility as data can be sent from one FU to another without passing through the register file [Gan07, Sch07]. As the availability of forwarding paths has an impact on the register allocation, the compiler which maps to these architectures must also support such forwarding paths.

2.1.1.3 Processor pipelining

Processor pipelining is a design method that inserts an extra register between different phases of the processor execution. Typically different stages are used for Fetch, Decode, Execute and Writeback [Hen96]. As a result the critical path is shortened, which allows increasing the clock frequency. In order to reach the required clock frequency for current designs, state-of-the-art processors have a pipelining depth ranging from 3 to 12. However, increasing the frequency arbitrarily will lead to the additional stages and an increase in the dynamic energy/power consumption due to the extra registers. In extremely scaled technologies, timing differences due to process variation lead to even more design problems for very high clock frequency processors. Therefore the processor design needs to carefully balance the exploited parallelism and target frequency (linked to the pipelining depth) in order to reach the required performance.

2.1.1.4 Issue logic

Different processor types can be categorized based on the order in which they issue and complete their instructions. Some processors support the out-of-order issuing of instructions. Out-of-order execution requires the hardware to keep track of what hardware is currently in use, which instructions have been executed, which operands have been produced etc. Alternatively, the responsibility of keeping the hardware busy can be moved to the compiler, in which case, in-order issuing can be used and no expensive hardware is required. With respect to the completion of the instructions, two types exist. With in-order completion, the results need to be written back in the same order as the instructions have been issued. With out-of-order completion, this restriction is removed, which leads to more flexibility, but a higher complexity.

Traditionally techniques like speculative execution, out-of-order issue and completion are extensively used in high performance super-scalar processors e.g. Intel's series of x86 processors, Power PC. In battery operated embedded systems, due to the energy efficiency requirement, a preference is given to simple hardware and the issue (and completion) order of instructions is fixed by the compiler at compile time. Using techniques like software pipelining, the execution can follow a different order than is given by the DFG, thereby creating freedom to optimize the performance. This allows much lower overhead at run-time and therefore is more energy efficient.

2.1.1.5 Overview of state-of-the-art processor classes

The embedded processor space contains a wide range of options. On one side, very low power processors target battery operated or even self-sustaining systems, but provide only a limited performance. On the other side, very parallel types can provide an extremely high performance in case a larger battery or a connection to the power grid is acceptable. The state-of-the-art embedded processors can be split up into different processor types, each offering a different trade-off point in the performance vs. energy efficiency vs. mapping effort space. At the lowest performance and lowest flexibility side of the processor spectrum, ultra-low power micro-controllers, like TI's MSP430 [TI09a] can be used to do very basic types of processing. They are good candidates when the total energy budget is extremely constrained and when the workload and performance requirements are correspondingly low. They are namely optimized mostly for control tasks.

Slightly more flexibility can be provided by small sequential RISC processors, e.g. [ARM09a]. They are typically small in-order single slot machines that have a shallow pipeline and that can exploit only a limited amount of parallelism.

Another class of ultra low power processors are targeting sensor nodes (as shown in Figure 1.1). These processors have a power budget in the range of tens to hundreds of microwatt. Examples are [Eka04, Kar04, Cal05, Naz05]. But also in that case, the maximal performance is heavily limited. Especially at U.C. Berkeley [Rab0-] major efforts have been invested to motivate the need to improve the energy efficiency to reach the levels of these scavenging limits, and to contribute in these improvements for sensor node networks.

By providing multiple slots, VLIW processors can increase the performance, while minimizing the required hardware overhead (compared to superscalar processors). Low power embedded VLIW processors can typically execute between 2 [TI09b] and 8 [TI06] operations in parallel. Most of them combine this however with a limited form of data parallelisation on several of the slots (e.g. the C6x series of TI and the TriMedia of NXP [VdW05]). So every instruction then executes several operations in parallel (e.g. 2 tot 4). This can potentially increase the maximal performance significantly. Many VLIWs provide a rather general instruction set and therefore are still quite flexible. But on the other hand, most of them are quite optimized for executing digital signal processing tasks in e.g. wireless or media processing.

Another class of VLIW style processors is organized as wide or hierarchical VLIW processors, which provides more flexibility than pure vector processors, as different operations can be executed in parallel [SilH, Mon05]. They form very heterogeneous VLIWs.

Some VLIW processors however support only quite specific operations that improve the performance for a selected target application or application domain. Various processor extensions (SIMD, loop buffering, clustering, etc.) can be used to improve energy efficiency, performance or both. When the processors are more customized in this way, they become an Application Specific Instruction set Processor or VLIW-ASIP. In this case a distinction can be made between ASIPs and accelerators, e.g. [Lu99]. An accelerator is customized to accelerate only part of the application, while the rest is executed by a so-called host processor. An ASIP often combines both into one processor.

Worth mentioning in the wireless domain are the following ASIPs. NXP's embedded vector processor (EVP) is a software-programmable platform for basebands with a relatively wide data-path [Kum08]. Infineon's MuSIc (Multiple SIMD Cores), is a DSP-centered and accelerator-assisted architecture and aims at battery-powered mass-market handheld terminals ([Ram07]) The University of Michigan has developed a very low-power processor meant especially for relatively low performance sensor applications – called Phoenix [Woh08]. It consumes power as low as 30 pW. Sandbridge [Glo04, Sandbridge] has launched a multi-core, multi-threaded, dynamically reprogrammable processor that supports SIMD vector processing. At the Univ. of Dresden, the Tomahawk platform and its predecessors have been introduced [Hos04]. And also at the University of Aachen, ASIP related work has been ongoing for several years [Schl07].

In addition, several commercial activities on such ASIP IP cores have been launched. Some of these are also template-based. In particular, Tensilica's Xtensa and Lx processors belong to this family [Xtensa]. And Silicon Hive has proposed several VLIW-ASIP cores oriented to multi-media and wireless processing too [Phi]. Also CSEM's Macgic DSP template belongs to this category [Ram04].

Heterogeneous cores, like [Bar05b], combine multiple styles into a single processor, e.g. a RISC core with a more parallel datapath part, to accelerate the execution of specific application parts only when needed.

To meet the real-time constraints of some highly regular application domains (limited control flow) with high throughput requirements, extremely parallel processors have been developed. These processors are called vector processors. Some make extreme use of SIMD like [Abb07].

Others are explicitly targeting graphics applications, can handle many parallel threads with low overhead and are called GPUs or Graphics Processing Units for e.g. [Nvi09, ATI09]. These GPUs also offer very high SIMD and accelerators for various graphics operations. They also contain extra hardware to support the dynamism of fast thread creation and management used in, e.g. objects in 3D gaming.

Finally some embedded processors extensively make use of data communication between their PEs in order to map larger parts of the data flow graph onto the processor. These processors are often organized as a 2D array, and are usually called Coarse Grained Reconfigurable Architectures or CGRAs, e.g. [Mei03a, PAC03]. CGRAs offer a wide range of design parameters, like the type and topology of their interconnect, the number of slots, etc.

Figure 1.4 provides a comparison of several of the main processor classes and a few representative instances in the energy-performance trade-off range.

In order to achieve an efficient mapping of software onto this wide range of processors, various mapping and compilation techniques have been developed. This ranges from the efficient extraction of data parallelism, thread level parallelism, exploiting ILP and various other techniques. As architecture and compiler are closely linked, most of the above mentioned references include a description on relevant compilation strategies for the respective processors.

2.1.2 Data memory hierarchy

Next to the processor core, the data memory hierarchy is a second component that has a high impact on the performance and power consumption of the platform. In general purpose computing the movement of data between different parts of the data memory hierarchy is often handled by hardware support. This corresponds to the use of so-called hardware controlled *caches*. In most embedded applications the data access pattern is more regular and can be analyzed. Therefore the movement of data can be more efficiently handled by the compiler or programmer and scratchpads are used (data memory without hardware support to move data around). Hybrids also exist, in which part of the cache can be used as a scratchpad, e.g. in [VdW05]. A scratchpad based solution is power and performance efficient [Kan04a, Ban02], but it requires a DMA or Direct Memory Access, which is a separate datapath that needs be programmed to handle the required transfer. [Tal08, Mat03, Het02] present various architectural possibilities and their trade-offs for DMAs and Address Generation Units (AGUs) [Tal08].

Besides the aspect of the type of memory that is used (hardware controlled cache or scratch pad), it is also possible to exploit the reuse of data over time and the access order pattern (locality). Therefore a right choice of the memory sizes, multiple levels and their inter-connectivity (memory hierarchy) is needed for low power and high performance. The locality can only be efficiently exploited if the application is transformed to make the most efficient use and transfer of the data possible. Various transformation techniques that expose the data locality (both spatial and temporal) have been proposed in the literature [Abs08, Bro00a, Mar03, Pan98].

2.1.3 Instruction/configuration memory organization

The instruction memory contains the bits which describe the functionality of the different slots and other processor components. In the CGRA/FPGA community this is also called the configuration memory. They are conceptually similar and are treated together in this book. Instructions/configurations are distributed and stored in the Instruction/Configuration Memory Hierarchy or ICMO.

A good ICMO aims at distributing the instructions over the architecture at the appropriate time and at a low cost. The ICMO can be organized similar to the data memory hierarchy, namely as multiple hierarchical layers. However, instructions differ from data in some important aspects (e.g. they are only read, not written) and therefore some extra optimizations can be used. With respect to the locality, instructions offer extra opportunities for (mostly temporal) reuse as most embedded applications consist of computations that are organized in the form of loops. Various techniques like filter caches [Kin00], trace caches [Rot96], victim caches [Jou90, Jou94] have been proposed to optimize the behavior of caches for loop code.

Further optimizations, like loop buffers [Kav05, Uh99] go one step further, by adding a special scratchpad (software controlled) to store the instructions of loops only. Loop buffers can also be clustered [Vda04b, Jay05b] and their size can be customized to reduce the energy/power consumption.

Independent of the fact if hardware controlled caches or scratchpads are used, the instructions stored in memory can be compressed. This allows a smaller memory foot print and is especially important for very parallel architectures, e.g. VLIW processors. These techniques also reduce the traffic between the instruction memory and the processor cores, but require an additional decompression before the actual execution step. Various techniques of instruction compression and NOP compression exist in literature [Deb00, Tan02, Gor02b, Adi00]. A detailed survey of instruction compression is given in [Bes03].

A more detailed overview of the design space of instruction/configuration memory organization is presented in Chapter 6.

2.1.4 Inter-core communication architecture

In order to use all described components in an embedded system, they have to be connected. This can be done using different architectures, ranging from custom point to point connections, over a standard bus (e.g. ARM's AMBA), to a network-on-chip. An overview of state-of-the-art communication architectures can be found in [Kog06, DeM06, Ler08] for network-on-chip solutions or [Pap06, Hey09] for bus based work. The communication architecture can be chosen based the traffic requirements, number of components, design time, etc. A description of the trade-off space for communication architectures is outside the scope of this book and is not discussed further.

2.2 Platform architecture exploration

The range of embedded processors is large and both architectural as well as mapping extensions can be used to modify a processor style. Therefore an exploration phase should be used to find the best match between architecture and application. Conceptually, the architecture exploration can be spit into three different parts, as is shown in Figure 2.3. Firstly, a *search strategy* has to be chosen to go over the different candidates. Secondly, a number of

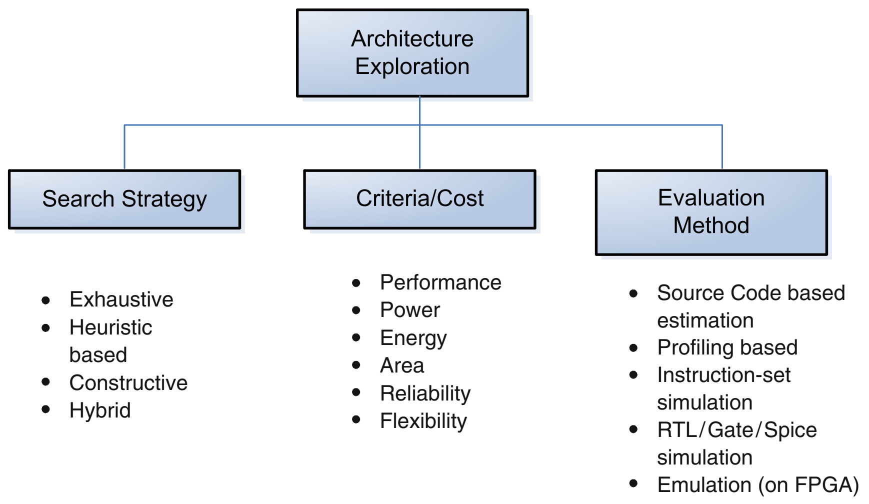

Figure 2.3: Architecture exploration tools/models

criteria or cost estimates, on which each candidate architecture has to be evaluated on. These are often cost models which may be analytical or empirical. Finally, each candidate solution has to be evaluated using a certain *evaluation method*, with respect to the design criteria. These three parts are discussed in more detail below.

2.2.1 Exploration strategy

In order to design an embedded processor that meets all requirements with respect to performance, energy efficiency, area, etc. architecture exploration is used to compare a range of architecture candidates and gradually improve the design. An exploration strategy or technique is needed to steer this process, as the exploration space is extremely large. In most practical cases, the exploration is restricted to a part of the space, usually called the supported architecture template [Gon02, Mei03a, Tar08]. This template can be the result of the experience of the design engineer, often including a combination of application knowledge and intuition, or the result of restrictions in the used mapping tools (e.g. the compiler). The exploration within the template can be further steered by a set of heuristics to speed up the exploration further, as in most cases an exhaustive search would still be infeasible. Such heuristic based exploration is performed in commercial tools, like Tensilica's Xtensa [Gon02] and in academic work, like [Jac00, Lap02]. Alternatively, constructive techniques can be used to generate an architecture based on the analysis of a set of applications, usually starting from a minimal (or extremely rich) base architecture and demand driven adding (or removing) resources. This approach is popular in high level synthesis and in cases where only a fixed set of applications have to be run. An example of such constructive techniques can be found in Synfora's PICO [Syn08, Phi04].

2.2.2 Criteria/cost metric

The target of the architecture exploration is to find an architecture that meets the requirements. These requirements can be defined with respect to a number of criteria or cost metrics, of which performance, energy efficiency, area, design effort, flexibility are the most used examples. Requirements can be a combination of relative (trade-offs) or absolute (constraints) criteria, e.g. minimize the energy consumption within the available design time (trade-off) for a given minimal performance target (constraint). The effect of a modification at the architecture level, be it a change in processor or memory style or an extension to a component of a processor, propagates to the various design criteria in a non-trivial way. This is because all criteria are linked and strong trade-offs exist, either in the design of a single component or elsewhere in the system.

2.2.2.1 Performance

As most embedded systems interact with the physical world, their real-time performance is often essential. Therefore performance is one of the most important requirements. *Simulation* is a commonly used technique to estimate the performance during embedded processor design, e.g. [Aus02]. The processor can be modeled at different abstraction levels and the corresponding simulation can be instruction accurate, cycle accurate or bit accurate. Cycle accurate simulations are commonly used to estimate the performance of the processor. However, to validate the real-time performance (in seconds), a target clock frequency is needed. To obtain an accurate estimate for this, the complete design needs to be synthesized/placed and routed. The more accurate the simulation needs to be, the slower the process of estimating the performance. In most cases, for relative comparisons a crude estimate suffices however.

2.2.2.2 Energy consumption

In the context of battery-operated embedded systems, trade-offs with respect to the energy efficiency of the architecture can be made visible throughout the design if the energy consumption can be estimated. Traditionally, energy estimation is done rather late in the design flow, e.g. at gate level. However, this means that many decisions have been taken and a design change at this stage of the design process would require a large effort. Therefore it is difficult to perform a broad exploration at this level or to make significant modifications based on the feedback from the gate level energy estimation. It is however possible to create energy models at various levels of abstraction, which can then be used during exploration at higher abstraction levels. In this way both benefits, namely speed and accuracy can be combined to some extent, which suffices for relative comparisons during early exploration.

Models calibrated after measurement For widely used processors like an ARM9 [ARM09b], TI's TMS320C6x [TI00] series, Intel's processors measurements can be performed to create an accurate energy model. This energy model is often created by a test sequence of instructions or program and measuring the current. Various related work, like [Sin01, Ben02, Jul03, Cha00, Isc03, Tiw96] have created an energy/power model after measurement. These models are often at the granularity of energy consumption per instruction for the complete processor. This complicates the use of the model to make architectural modifications, as they don't produce a breakdown for different processor components.

Analytical models It is also possible to create models that are analytic. Related work like Wattch [Bro00b], CACTI [Shi01] and others like [Mam04, Ye00] create an energy model for individual components. They make assumptions on the micro-architecture for each of the components and on the design style (e.g. standard cell vs. custom design). Based on the technology choice and the micro-architecture assumption, the analytical model gives the energy consumption per activation of the component. Most work in analytical modeling is targeted towards memories as their structure is quite regular and can be captured in a model with reasonable accuracy. However, some work on analytical datapath modeling exists, e.g. [Ask99, Hel02]. The energy models from analytical modeling are often energy consumption per activation of a component. They can also be sensitive to the actual bit switching.

Models at/after RTL Models can also be created at Register Transfer Level (RTL), either for components or for the complete processor. The RTL can be synthesized and the energy consumption can be estimated based on statistical activity. Instead of statistical activity real application data can be used, but this can be very time consuming, especially for a complete processor. The estimation can be made more accurate if done at gate level, and even more accurate after place-and-route and parasitic extraction. A detailed study of the error incurred and the time consumption between the different stages is done by [Bal04]. Based on the granularity of the modeling, the models will represent the energy consumption per activation of a component or the complete processor respectively. These models can also be extended to take into account the switching activity of the input.

Peak power consumption In the design of embedded systems for mobile devices, energy consumption is an important metric, as it is directly linked to what can be drawn from the battery. However, the peak power consumption has a large impact indirectly, as is influences the worst case heat dissipation and thereby the packaging cost. Estimating the peak power consumption requires, in addition to e.g. the simulation set-up, the availability of worst case test patterns [Hsi97].

2.2.2.3 Area

In the competitive market of consumer electronics, the cost of the design is extremely important and for some designs the cost is directly related to the area of the chip. A first synthesis of individual components gives a first area estimate at the component level. Further placement and routing of components can improve the accuracy of this estimate. However, both methods ignore the overhead/optimization potential of global place and route. Early in the design, a crude estimation suffices to be able to do relative comparisons. For memories this can be often be estimated by analytical models like Wattch [Bro00b] or by using commercial memory generators like Artisan [ARM] or Virage [Vir].

2.2.2.4 Design effort

Aside from the impact different processor styles have on the performance and energy efficiency, there usually is also an effect on the design effort that is to be expected when choosing for a certain style. For example a simple RISC processor, using a push-button compilation approach, will take a much less design effort than the implementation of an ASIP, which requires the manual modification of the application code to make use of the application specific processor extensions. It can be expected that the energy efficiency and performance of the ASIP will be superior to that of the RISC. But when comparing an ASIP solution with a CGRA implementation, the trade-off becomes less obvious. The design effort is part of the overall design trade-off and therefore should be taken into account, but is rather hard to quantify accurately. Therefore in most cases qualitative comparisons are often used and the weight of this criterion with respect to the other cost metrics is up to the designer.

2.2.2.5 Flexibility

A final important aspect of the embedded platform design trade-off is the flexibility of the designed solution. This aspect covers the ability to map other applications to the platform after the design. One extreme case is an Application Specific Integrated Circuit or ASIC, which is completely fixed to a single task, but performs this task very efficiently. On the other side of the spectrum, a general purpose processor can run any application, but at a severe energy and performance cost. The embedded processor space contains multiple in-between solutions, that provide varying levels of flexibility, at various performance and energy efficiency points. However flexibility is a qualitative criteria as often it is a trade-off with efficiency of mapping.

2.2.3 Evaluation method

A key element of a successful exploration is the ability to evaluate different options with respect to the criteria and cost metrics. Depending on the cost metric, different evaluation methods can be used and the evaluation can be (or needs to be) performed at different abstraction levels or with different accuracy. At each of the different levels the cost models can be used to evaluate the quality of the architecture.

Often the initial algorithm development happens in a high level language like Matlab, at which stage the functionality is evaluated. Based on the computational complexity of the application a this level, a first rough estimate can be made of the required peak performance of the processor. It is then refined down to C,[1] which is functionally equivalent to the Matlab reference and now is ready for mapping on an architecture.

The initial architecture exploration is often performed on Instruction Set Simulators. Various simulators exist, like Trimaran [Tri08], Simplescalar [Aus02] etc. Instruction Set Simulators are often accompanied with an associated (retargetable in some cases) compiler front end which compiles the code on the architecture. Commercial instruction set simulators, like Target Tools [Tar08], Processor Designer [CoW08a], Synfora's PICO [Syn08], ASIP Meister [EDA05], Tensilica [Gon02] are available at this level.

Architectures can also be modeled at transaction level. The architecture in this case is modeled in a transaction accurate model often in SystemC or similar languages. Frameworks like Liberty [Lib02], Unisim [Uni05] can be used for this purpose. In practice these simulators model on an abstraction level below ISS level and therefore are more accurate, but slower

[1]The C-language is used most for the embedded systems domain, which is the target of this book, but other languages can be used.

than instruction set simulators. Some commercial frameworks like CoWare's Virtual Platform [CoW08b] also exist to support TLM (Transaction Level Modeling) of such processor architectures together at the platform.

On further refinement, architectures can also be evaluated/explored at Register Transfer Level (RTL). This is very time consuming and often only minor modifications are done during exploration at RTL level at the complete processor level. This can then be refined to gate level and simulated as well. Both RTL and gate level simulation for multi-million gate designs are prohibitively slow and therefore are not usable.

However, instead of simulating the gate level model, it can also be mapped on an FPGA. This is called *emulation* and it is commonly used practice to verify the final design, before producing the chip. Emulation of an architecture is faster than RTL or gate level simulation and therefore some amount of exploration can also be done at this level. Various academic FPGA emulation projects like [UCB07, Ati07] and commercial emulation platforms like Mentor Graphics' VStation [Men07] exist.

2.3 Conclusion and key messages of this chapter

This chapter has presented a structured overview of the related work in the design of battery operated embedded systems, with an emphasis on the design, exploration and evaluation of embedded processors. Specific related work that is directly related to the specific contributions of this book will be discussed at the end of each respective technical chapter.

The design space for embedded processors is extremely large. In addition, the complete embedded platform consists of multiple components (each with a range of options) that can not be studied in total isolation, as modifications to one component can influence the other components. Architecture exploration can be used to find a good match between the application requirements and the architecture design. To be able to perform an efficient exploration and compare different processor styles, this exploration should be performed early on in the design. Therefore a trade-off has to be made between the implementation effort that is required to model an architecture, the evaluation speed and the accuracy. In the next chapters this broad exploration will be performed based on a realistic case study resulting in a set of high-level requirements and a proposal for an ultra-low energy platform template. In addition, a framework will be proposed to effective support this broad exploration.