Chapter 3

Software-Level Power Estimation

The main goal of this chapter is to define a methodology for the static and dynamic estimation of the power consumption at the software-level to be integrated in a more general HW/SW co-design environment for embedded systems. More in detail, this chapter introduces power evaluation metrics related to the SW part to be applied during or after the HW/SW partitioning phase. The power analysis at the SW-level is based on the characteristics of the system architecture in terms of processor and memory sub-system. The main contribution of the proposed approach is to be independent, at the high-level, of the system specification language and, at the low-level, of the target processor. This goal has been achieved by introducing three description levels: the OCCAM2 language at the high-level, the VIS (Virtual Instruction Set) pseudo-assembler language at the intermediate-level, and the assembler of the target processor at the low-level. The chapter aims at describing the overall power assessment methodology, that is composed of four main phases: the software compilation, the software back-annotation, the power estimation (static analysis), and the software profiling (dynamic analysis).

3.1. Introduction

Embedded systems have become widely used in the most recent past for a large number of computing and telecommunication applications. As opposed to general-purpose computer systems, embedded systems are computing and control systems conceived for dedicated applications, to respond to specific requirements [47]. In general, the embedded system architecture is composed of both a HW and a SW part: one or more dedicated devices, such as ASICs or FPGAs, implement the hardware part, while a set of routines running on a dedicated processor or ASIP implement the software part. Using the system-on-a-chip (SOC) approach,
the entire embedded system can be implemented into a single chip, usually including the embedded core processor, the on-chip memory, the I/O interface and the custom HW part.

Innovative HW/SW co-design techniques have recently emerged as a new CAD discipline to support embedded systems design. Co-design techniques aim at meeting the system-level requirements by adopting a concurrent design and validation methodology, thus exploiting the synergism of the HW and SW parts. As described in Section 2.2, a co-design flow includes several design activities, such as the system-level modeling and co-specification, analysis and validation of the co-specification, system-level partitioning, architectural design space exploration, co-synthesis and co-simulation. The availability of a co-design methodology, covering all the design phases, is a primary issue during embedded systems design to meet the system-level requirements.

More specifically, the cost/performance figure of merit at the system-level is greatly impacted by the effects of the partitioning phase, aiming at the assignment of the co-specification operations to the HW or SW parts. To guide the partitioning algorithm, estimation metrics should be provided to compare the results of alternative partitionings and to evaluate their conformance to system-level design goals. These goals typically include system-throughput, operating frequency, area, power dissipation, global cost, ... . Since an optimal partitioning solution satisfying all those constraints is difficult to obtain in a reasonable time, a sub-optimal solution is usually accepted, derived from a partial exploration of the architectural design space.

The importance of power constraints during the design of embedded systems has continuously increased in the past years, due to technological trends towards high-level integration and increasing operating frequencies, combined with the growing demand of portable systems. Despite of the increasing importance of power consumption in most of the embedded applications, only a few co-design approaches appeared so far have taken into consideration such a goal at the highest levels of abstraction [52]. Although high-level power assessment for both HW and SW components is the key factor in the early determination of the power budget for embedded systems, high-level power models have not yet achieved the maturity necessary to enable their use within current industrial CAD environments. One of our goals is to fill one aspect of such a gap, by providing a SW-level power model suitable for the typical architecture of embedded systems.
Our approach can be considered as an attempt to cover power estimation issues from a HW/SW comprehensive perspective, mainly focusing on the SW part and considering a general architecture adopted by most industrial synthesis systems. The goal is to widely explore the architectural design space during the system-level partitioning and to early re-target architectural design choices. Accuracy and efficiency should be the driving forces to meet the power requirements, avoiding re-design processes. As already mentioned, the relative accuracy in high-level power estimation is much more important than the absolute accuracy, the main objective being the comparison of different design alternatives.

Furthermore, a framework for HW/SW co-design of embedded systems, called TOSCA (TOols for System Co-design Automation) [6] has been used as basic environment with the twofold purpose of integrating and validating the proposed power estimation methodology. Figure 1 shows the co-design flow and tools of the TOSCA environment, while Table 1 reports the main activities composing such a flow.

The most relevant aspects of the system co-design strategy defined in the TOSCA environment can be summarized as follows:

- the target applications are control-dominated systems, typical, for instance, in the telecom switching systems domain;
- the set of tools defined and implemented aims at offering a single development environment, thus providing the user with a common user interface, although all tools can be separately managed and tuned;
- the target architecture is constituted at this time by a single embedded processor connected to a set of hardware units, called co-processors, that perform the operations bound to hardware implementation;
- a single internal model has been specified which maintains all the information required by the different tools to perform simulation, validation, partitioning, scheduling, and resources allocation.

Within the TOSCA co-design flow, the primary objective of our research is the definition of power evaluation metrics for the SW source code to be applied during or after the HW/SW partitioning phase (see Figure 1).
The most relevant contribution of the proposed methodology is fundamentally related to their independence, at the high-level, of the system specification language and, at the low-level, of the target processor. Thus, our approach is easily re-targetable towards several specification languages or graphical formalisms and several families of commercial processors. This goal has been achieved by introducing an intermediate pseudo-assembler level. The intermediate virtual assembler language preserves a good level of generality and provides an instruction set supporting the characteristics of a broad range of possible target processors.
The power assessment process is tightly correlated to a specific compilation strategy of the SW and to the characteristics of the system architecture in terms of the target processor and the memory sub-system. The dependence of the processor and memory is taken into account through the introduction of two technology files containing timing and power characterization data. The approach is thus re-targetable towards several system architectures, since it can easily accept characterization data coming from several commercial processors and memory architectures.

<table>
<thead>
<tr>
<th><strong>TOSCA Activities</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Co-specification</td>
<td>Acquisition of system functionality, design constraints and application goals</td>
</tr>
<tr>
<td>Analysis &amp; Validation</td>
<td>High-level validation of the system behavior, execution profiling and bottle-neck identification</td>
</tr>
<tr>
<td>Partitioning</td>
<td>Design-space exploration to define a proper system modularization and hardware vs software binding</td>
</tr>
<tr>
<td>Co-synthesis</td>
<td>Synthesis of the hardware parts (VHDL), software modules (target assembly compilation), hardware-software interfaces (VHDL, OS communication routines) and linking with a real-time Operating System</td>
</tr>
<tr>
<td>Co-simulation</td>
<td>Simulation of the complete hardware-software system though a suitable instruction-level software executor written in VHDL</td>
</tr>
</tbody>
</table>

Table 1: The main activities composing the TOSCA co-design flow.

The present chapter is organized as follows. A background analysis related to the software-level power assessment is provided in Section 3.2, while the target system architecture is presented in Section 3.3. The overall software evaluation methodology is detailed in Section 3.4. The method operates at three different description levels: the high-level OCCAM2 language, the intermediate quasi-assembly VIS (Virtual Instruction Set) level, and the assembler level of the target processor. The motivations to work at the OCCAM2 and VIS levels are provided in Paragraph 3.4.1 and 3.4.2, respectively. The basic characteristics of the OCCAM2 and VIS languages have been reported as well. The overall methodology for the power analysis at the system-level is composed of four different processes:

- the software compilation;
- the back-annotation;
- the power estimation (static analysis);
- the software profiling (dynamic analysis).
First of all, the compilation process is briefly described. This process is in turn composed of two different phases: i) the OCCAM2-to-VIS compilation, and ii) the VIS to the target assembler mapping. The basic features of the OCCAM2-to-VIS compilation phase, described in Paragraph 3.4.3, are the extensibility, to provide a compiler structure as flexible as possible with respect to future extensions and improvements, and the modularity to easily maintain under control the single compilation passes. To meet these requirements, an orthogonal strategy to afford the different compilation passes independently of each other has been adopted. The second compilation step consists of the VIS to the target assembler mapping, shown in Paragraph 3.4.4. The main characteristics of the mapping phase lies in its efficiency and generality with respect to the set of selected target processors.

Paragraph 3.4.5 introduces the back-annotation process, aiming at reporting the relevant information from the lowest to the highest abstraction levels. The back-annotation process, as well as the compilation, takes place in two different phases: i) VIS-to-OCCAM2, and ii) target assembler-to-VIS. The purpose is to annotate, during the backward flow, the most relevant parameters related to the overall power budget.

Then, Paragraph 3.4.6 affords the SW-level power estimation process at the different levels of abstraction (namely OCCAM2, VIS, and target assembler). The basic assumption is that the timing and power characterization of the instruction set of the target processors and of the memory sub-systems are given. The analysis carried out so far is basically a static analysis, whereas the software profiling process, described in 3.4.7 and based on OCCAM2 and VIS code simulations, provides the dynamic analysis. The dynamic analysis has been introduced with the purpose to increase the accuracy level of the power figures, mainly to take into consideration the effects of data dependencies. Finally, Section 3.5 summarizes the original contributions introduced in this chapter as well as some considerations on the future directions of the research.

3.2. Previous Work on Software-Level Power Estimation

The relevance of the power issues among the other design constraints is steadily increasing for designing embedded applications ([34], [165], [49]). Power evaluation and optimization techniques should be included in the co-design flow at several abstraction levels to cope with system-level power requirements in an acceptable design time. However, the availability of a
high-level power estimation tool is mandatory to obtain early estimation results, while maintaining an acceptable accuracy level and a competitive design time [102] [137]. Up to now, accurate power estimation tools are available for the lower levels of the design flow, such as the circuit-level, gate-level and to a limited extent at the architectural-level [118]. Nevertheless, those techniques are inefficient and inaccurate to evaluate the power figures associated with the application software running on an embedded core processor.

The power dissipation associated with a source code is mainly related to the system characteristics in terms of target processor, memory sub-system and system-level buses. In the last few years, different methods to estimate the power at the SW-level have been proposed. According to the survey appeared in [157], they can be classified as: i) simulation methodologies based on the gate-level descriptions of the processor, ii) models based on the architectural-level processor description, iii) bus activity-based models, and iv) instruction-level models.

The lower level model is based on the usage of gate-level simulation and power estimation tools on a gate-level description of the processor [137], [41]. This is the most accurate method, assuming that a detailed gate-level description of the processor is available. However, in most cases, this method cannot be applied due to the lack of information related to the processor description, especially if the processor is not an in-house design. In a limited set of cases, where the detailed gate level description of the processor is available, those techniques are impractical to be applied due to a large amount of simulation time required.

Architectural-level power estimation is less precise but much faster than gate-level estimation. Some attempts to apply architectural-level power estimation techniques to processors have been recently investigated [159] [145] [160]. This approach requires a model of the processor at the major components level (ALU, register file, etc.) along with a model of which component will be active as a functions of the instructions being executed. These approaches are based on the use of software instructions as stimulus for the simulation, then they add the power contribution of each active processor module in each clock cycle. The current associated with each active module represents its power cost ant it is usually supposed to be constant and independent of the operand value, circuit state, and correlation with other active modules.
The switching activity of system-level buses is another indicator of the SW-related power. In this simplified model, the bus activity is assumed to be representative of the overall switching activity in a processor. Modeling the bus switching activity essentially requires the knowledge of the bus architecture, the op-codes generated by the processor, a representative set of input data to the program, the memory management scheme, and the architecture of the memory sub-system. By simulating the execution of a program on this model of the system, one can evaluate the transition activity of the system-level buses. A further attempt in this direction is the model proposed in Chapter 6 of this dissertation. Other efforts have been reported in [172], where the authors implement bus switching activity in their cold scheduling algorithm proposed for instruction scheduling. In this approach, only instruction and address related switchings are considered due to the unpredictability of data values.

Some approaches for the instruction-level power assessment are based on stochastic data modeling suitable for the class of DSP applications ([98], [101]). An instruction-level power analysis model has been proposed in ([179], [181]) based on physical measurements of the current drawn by the processor during the execution of embedded software routines. More in detail, this approach discusses the application of the proposed technique for the power assessment of two commercial processors: the Intel 486DX2 and the Fujitsu SPARClite 934. The basic idea is to measure the current drawn by the processor as it repeatedly executes an instruction sequence composed of a loop with several instances of the same instruction. The goal is to assign a given power cost (called base energy cost) to each single instruction of the instruction set. The assumption is that each single instruction involves a specific processing in the various modules of the processor. The resulting circuit activity has been considered as representative of each instruction and can vary with instructions. The variation in the basic cost of a given instruction due to different operands and address values has also been quantified.

However, during the execution of a software routine, certain inter-instruction effects occur whose power cost is not taken into account if only the base energy cost is considered. The first type of inter-instruction effects is due to the previous state of the circuit. The cost of a pair of instructions is always greater than the base cost of each instruction in the pair and the difference cost is called the circuit state overhead. Other inter-instruction effects are related to resource constraints that can lead to stalls (such as pipeline stalls and write buffer stalls) and
cache misses which imply a penalty in terms of power. The results of the analysis motivated several software-level power optimization techniques, such as instruction re-ordering to exploit the power characteristics of each instruction.

The instruction-level power model has also been applied for the power analysis of the commercial 32-bit RISC-based microcontroller Fujitsu SPARClite MB86934 suitable for embedded applications [181]. Experimental results have been provided to figure out the cost of the basic instructions. The effects on the power budget of external memory accesses, as well as the effects of enabling and disabling caches have been explored. Results have been reported to consider the impact of SW-controlled power management on the power cost of the basic instructions. Other benefits of the proposed analysis are related to the definition of micro-architectural design optimization.

The application of instruction-level techniques for the power evaluation of an embedded DSP processor has been discussed in ([107], [182], [108]). The target processor is a Fujitsu proprietary 3.3V 40 MHz processor including special architectural features to support embedded DSP applications. Some of these special features include dual-memory accesses and packing of instructions into pairs. Significant differences have been found between the power models of the previous examined general-purpose processors and the target DSP processors. In particular, the effects of the circuit state changes have been found to be more effective for the power for the target DSP rather than for the previous architectures. The significant impact of circuit state change in the case of the DSP processor suggested power minimization techniques based on an appropriate scheduling of instructions that can modify the global cost of a given software routine. In addition, concerning certain instructions involving the data-path, even non-adjacent instructions can cause some changes in the circuit state that can contribute by approximately the 13% of the power cost.

The problem of scheduling aiming at energy reduction has been addressed in ([178], [169], [106]). Based on the results of previous works on instruction-level power model in processors, Tiwari et al. afforded in [178] the issue of power-oriented compilation techniques exploiting the capabilities of a given architecture. Basically, these techniques include instructions re-ordering, code generation through pattern matching and reduction of memory operands. In [169], the authors targeted the power minimization due to instructions scheduling by considering the power dissipated by the controller of a processor. The power figures
related to different instruction schedules have been derived by measuring the switching activity on an RTL model of the control-path of the processor.

In [106], the authors address the problem of allocating the memory to variables in embedded DSP software to maximize simultaneous data transfers from different memory banks to registers. The memory bank assignment for low energy, the instruction packing and other low power scheduling techniques suitable for DSPs such as the operand swapping for the Booth multiplier have been also proposed in [108].

In summary, although different promising approaches have been proposed in the last few years to deal with SW-level power estimation, we can consider this field a young area of investigation. A further interest in the SW-level power estimation should be expected in the next future.

In the present chapter, we define a power assessment framework at the SW-level. The most relevant features of the proposed approach can be summarized as follows:

- it is integrated into a more general HW/SW co-design environment for control-dominated embedded systems;
- it copes with the definition of power evaluation metrics for the SW to be applied during or after the HW/SW partitioning phase;
- it is tightly related to the characteristics of the system architecture in terms of the target processor and the memory sub-system; thus it is re-targetable towards several system architectures, since it can easily accept characterization data coming from several commercial processors and memory architectures;
- it is independent, at the high-level, of the system specification language and, at the low-level, of the target processor; thus it is easily re-targetable towards several specification languages or graphical formalisms and towards several families of commercial processors;
- it addresses both the static and dynamic estimation of power consumption.

### 3.3. The Target System Architecture

The target system architecture of the embedded system is based on a *System-On-a-Chip* approach. So it is implemented into a single ASIC, including both the software and the
hardware bound parts, the latter described at the behavioral/RT levels. The target system architecture is depicted in Figure 2.

The system architecture consists of the following main components:

- **the data-path**, composed of storage units, functional units and multiplexers. The storage units consist of registers and register files, while functional units can include a wide set of units such as adders, multipliers, and so on. A two-level multiplexer structure is considered for the interconnection among storage and functional units. The typical operation along the data-path implies a register-to-register transfer, consisting of the operands read from the input registers, an operation performed on the operands and the results stored in the output registers;

- **the main memory**, based on a memory hierarchy, that can be constituted by single or multi-port memories, cache memories, TLBs, FIFOs, LIFOs, etc.. We assume that all read/write accesses to the memory will be performed through input/output registers;
• **the control unit**, implemented as a set of FSMs and generating the control lines for the data-path components and the memory;

• **the embedded core processor**, such as a standard processor, a microcontroller, a DSP, etc., with its memory (even if part of the memory can be external) implementing the SW bound part;

• **the clock distribution logic**, including the buffers of the distribution network, organized for example as a balanced clock tree;

• **the crossbar network**, to interface the architectural units by using a communication protocol at the system-level. The interconnection power of the crossbar network is included in the power dissipated by the outputs of data-path, memory and control logic;

• **the primary I/O pads**.

### 3.4. The Overall Software Evaluation Methodology

As mentioned before, the overall power analysis operates at three different description levels (as depicted in Figure 3): the high-level OCCAM2 language, the intermediate quasi-assembly VIS (Virtual Instruction Set) level, and the assembler level of the target processor. Starting from the internal specification model based on concurrent processes, described in the high-level language (a modified version of OCCAM2) the software partition is then translated into the intermediate virtual assembly language, which is in turn mapped into the target assembly language. The first two paragraphs of this section present the main characteristics of the OCCAM2 and VIS languages, respectively. The main motivations to work at the OCCAM2 and VIS levels are also provided.

The basic assumption for the analysis is that the timing and power characterizations of the instruction set of the target processors and of the memory sub-systems are given in two technology files:

• **Processor Technology File**: The file contains the power consumption figures for each instruction or class of instructions and for each addressing mode provided by the processor instruction set.

• **Memory Technology File**: The file contains power consumption data of the read/write operations for each level of the memory hierarchy (on-processor and off-processor).
These two files completely characterize the target system architecture from the power consumption standpoint.

![Diagram of the overall power estimation design flow.](image)

Figure 3: The overall power estimation design flow.

The remaining of this section aims at illustrating the four different phases composing the power estimation design flow at the software-level (see Figure 3): the software compilation, the software back-annotation, the power estimation (static analysis), and finally the software profiling (dynamic analysis).

The compilation process moves in the forward direction from the high-level description (OCCAM2) to the low-level one (target-assembler), whilst the back-annotation process moves in the opposite backward direction. The compilation process is in turn composed of two different phases: i) the OCCAM2-to-VIS compilation and ii) the VIS-to-assembler mapping, detailed in paragraphs 3.4.3 and 3.4.4 respectively.

Then, Paragraph 3.4.5 describes the back-annotation process, aiming at reporting the relevant power and timing information from the lowest to the highest abstraction levels. The back-annotation process, as well as the compilation, takes place in two different phases: i) VIS-to-OCCAM2 and ii) assembler-to-VIS. The purpose is to annotate, during the backward flow, the most relevant parameters related to the overall power budget.
Finally, Paragraph 3.4.6 affords the SW-level power estimation process at the different levels of abstraction (OCCAM2, VIS and target-assembler). The analysis carried out so far is basically a static analysis, whereas the software profiling process, described in Paragraph 3.4.7, provides the dynamic analysis.

3.4.1. The OCCAM2 Language

In the TOSCA co-design environment, a modified version of the OCCAM2 language [81] has been used to model the high-level functionality of the system specification, without the necessity to introduce a specific characterization in terms of hardware or software. The main goal is to provide the designer with an environment in which the problems related to the actual implementation and architecture definition can be initially neglected, thus concentrating on the system functions. The language used for modeling must therefore be able to model in a non-ambiguous form both software processes, that will be implemented in the assembler language, and hardware modules, that will then be translated in the RT-level VHDL. This implies that not all the OCCAM2 constructs are allowed, in order to allow a correct mapping. Fundamentally, OCCAM2 language enables our methodology to be independent of the high-level description language (C, C++, ...).

The OCCAM2 syntax allows an easy definition of a formal system representation based on process algebra (Communicating Sequential Processes [80]). OCCAM2 supports the representation of both parallel and sequential execution of processes at any abstraction level. The language is composed of a restricted set of basic constructs.

Communication between concurrent processes is obtained through channels [80]. The underlying semantic indicates that the first process becoming ready will be executed as a consequence of the successful termination of a preceding input process. An important advantage of using channels to represent communication is the possibility of applying the same compact model of data exchange between heterogeneous processes (and consequently interfaces). The channels behave as variables with an attached unidirectional pipe management, leaving unmodified the original value (the channel receives only a copy). Since channels provide a point-to-point connection only, broadcast communication can be modeled via multiple channels. To simplify the design activity, such a case has been incorporated in
Chapter 3. Software-Level Power Estimation

the built-in graphical editor as a pre-defined broadcast module. Synchronization among processes is realized through channels, since they implement a rendez-vous mechanism.

The basic constructs supported in TOSCA are shown in Table 2. The data types accepted are integer, byte, bit. Arrays are also supported.

<table>
<thead>
<tr>
<th>OCCAM2 constructs</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKIP</td>
<td>No operation</td>
</tr>
<tr>
<td>STOP</td>
<td>Stop execution</td>
</tr>
<tr>
<td>Variable := expression</td>
<td>Assignment</td>
</tr>
<tr>
<td>Channel ? variable</td>
<td>Channel input</td>
</tr>
<tr>
<td>Channel ! expression</td>
<td>Channel output</td>
</tr>
<tr>
<td>SEQ</td>
<td>Sequential execution of the operations following</td>
</tr>
<tr>
<td>PAR</td>
<td>Parallel execution of the operations following</td>
</tr>
<tr>
<td>IF (condition) (process)</td>
<td>Execution of the process associated with the true boolean condition</td>
</tr>
<tr>
<td>ALT (condition1) &amp; channel?variable1 (process)</td>
<td>Alternative execution conditioned by channel events. The first process associated with a branch which verifies both the condition and the channel event is executed</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>WHILE (expression) (process)</td>
<td>Cyclic execution of the process until the boolean expression is true</td>
</tr>
</tbody>
</table>

Table 2: The set of OCCAM2 constructs considered in TOSCA

Time related issues are supported through the use of timer objects, acting as read-only channels providing an integer value corresponding to the current time. Delays can be introduced by using deferred inputs (through the AFTER clause). Typical real-time schemes can be modeled by combining the statements for managing time with the construct of alternative composition of processes in OCCAM2 [81].

In general, the description of the overall digital system can be viewed as a collection of OCCAM2 processes corresponding to different modules of the description provided by the user, possibly through different representation formalisms and design environments. A module can be constituted by a single OCCAM2 procedure, which represents the lowest level of granularity also for all operations performed in the co-design flow.

The choice of using the process algebra/OCCAM2 paradigm with respect to other formalisms, is based on the following advantages we found relevant for our co-design purposes:

- a formal and well assessed theory of transformations is available, so that it is easier to automatically produce semantically equivalent alternative designs;
the concurrency model is powerful enough to manage system-level specifications. In fact, it is possible to specify: channels, user-defined communication protocols, timers and timeouts, process priorities, regular parallelism (array of processes and channels);

- the modeling of arbitrary complex behaviors can be performed by using a computational model employing a compact set of primitives (assignment, input and output) and composition operators (sequential, parallel, alternative);

- multiple paradigms are supported (event-driven, dataflow, rendez-vous, FSM,...) through the same syntax and semantics;

- the OCCAM2 syntax is simple and regular, so that parsing as well as the generation of graphical representations of the system specifications is simplified.

A system specification, in particular in real-time systems design, requires the specification of timing and performance constraints. For instance, the user can specify maximum, minimum and duration constraints among events, which can be stimuli both internal to the system and external. These constraints cannot be expressed considering the standard OCCAM2 constructs. To include this important feature in our system-specification editor, we have extended the OCCAM2 syntax to describe the following combinations of timing constraints.

- stimulus-stimulus: maximum admissible delay between two stimuli;
- stimulus-reaction: maximum delay between the stimulus and the corresponding system reaction;
- reaction-stimulus: maximum insensitivity time among a system reaction and the following stimulus;
- reaction-reaction: maximum time interval between two system reactions.

The OCCAM2 representation stored within the design database has been customized to link the functional specification to the timing constraints (mindelay, maxdelay, minrate, maxrate, minburst and maxburst).

### 3.4.2. The VIS Language

Essentially, the choice to introduce the intermediate VIS level is to enable the proposed methodology to be independent of the target processor.

The compilation of SW-bound parts of the specification is a two-steps process. The first is based on the use of a low-level language, called VIS (Virtual Instruction Set), that is very
close to the assembly but still preserves a good generality [8]. The goal is to provide an instruction set aiming at analyzing the characteristics of a broad range of possible target microprocessors (tailored for embedded systems) and taking into account low level optimizations that usually cannot be easily considered by operating only at the source code abstraction level. Moreover, the second step, consisting of mapping the VIS code into the target assembly, is simplified due to the similarity between the VIS code and the commercial instruction sets.

The VIS has intermediate features between the RISC and CISC philosophies, mixing instructions belonging to both architectural classes. In particular, CISC instructions improve code compactness and simplify the final mapping step, while the presence of RISC instructions pushes the analysis closer to the target RISC microprocessor. In summary, the ultimate reasons to consider the analysis at the VIS level are to obtain a code with the following features:

- **Portability**: minimum effort to map the VIS into different target microprocessors;
- **Resources Independence**: code generation should abstract as much as possible from the available resources;
- **Simulatability**: VIS code can be efficiently simulated while considering the characteristics of the target microprocessor [7].

The architecture of the virtual CPU is composed of a single execution unit, without pipelining, a 16-bit General Purpose Registers (GPRs) bank and some Special Purpose Registers (SPRs). The external interface is based on a 16-bit bus (which can be scaled up) and a 20-bit address bus (which can be scaled up). The simpler memory model is without cache and there is a unified addressing space for both data and code. Code segment will be located in ROM while I/O registers are memory mapped.

The VIS architecture is orthogonal in the sense that all the arithmetic/logic operations are allowed among all GPRs. Furthermore, each GPR can behave as accumulator. The cardinality of GPRs is a user-defined parameter, typical values are in the range from 4 to 32. Special purpose registers are the Program Counter (PC), the Stack Pointer (SP), the Base Pointer (BP) and the program status word (PSW). Two specialized instruction pairs have been included to manage the interrupt and carry flags.
The VIS instructions refer to 16-bit registers, given the structure of the OCCAM2 data types. The possible datum is BIT, BYTE, and 16-bit integers, called INT. The addressing space, defined over 32-bit even if the external bus can be smaller, supports the most common addressing modes, as reported in Table 3. The VIS architecture does not allow the direct access to the memory for operations (as for typical load/store RISC architectures) and, similarly to the RISC-based machines, arithmetic/logic operations can be carried out only on GPRs.

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>MOVE const</td>
<td>-R0 [R0] ← const</td>
</tr>
<tr>
<td>Direct</td>
<td>MOVE R1</td>
<td>-R0 [R0] ← [R1]</td>
</tr>
<tr>
<td>Indirect</td>
<td>MOVE (R1)</td>
<td>-R0 [R0] ← [[R1]]</td>
</tr>
<tr>
<td>Base and Offset</td>
<td>MOVE #offset(BP)</td>
<td>-R0 [R0] ← [[BP] + offset]</td>
</tr>
<tr>
<td>Relative</td>
<td>BRA #offset</td>
<td>-- [PC] ← [PC] + offset</td>
</tr>
<tr>
<td>Relative to register</td>
<td>JUMP (R31)</td>
<td>-- [PC] ← [PC] + [R31]</td>
</tr>
<tr>
<td>Base, index and offset</td>
<td>MOVE #offset(BP+R1)</td>
<td>-R0 [R0] ← [[BP]+[R1]+offset]</td>
</tr>
<tr>
<td>Absolute</td>
<td>MOVE (#addr)</td>
<td>-R1 [R1] ← [[addr]]</td>
</tr>
</tbody>
</table>

Table 3: The VIS addressing modes.

The VIS instructions can be classified according to the argument count, ranging from zero to three. Zero-argument instructions typically are control instructions, while three-argument operations are arithmetic/logic operations. Concerning the addressing modes and operating codes, the VIS is not completely orthogonal, since not all the addressing modes can be applied to each operating code.

Two different types of communication are possible, affecting the VIS structure, depending on the partitions involved: HW/SW and SW/SW. To implement the rendez-vous type of synchronization it is necessary to define:

- **Channels State Table**: maintaining the information concerning the state of the communication on the channel and the processes involved.
- **Communication Protocol**: specified at the level of VIS code.
- **Operating System**: in terms of basic system calls to be inserted at compile time.
- **Interrupt**: some interrupt functions are present to manage the events generated by the coprocessors.
Under the architectural point of view, no significant differences exist between HW/SW and SW/SW communication, since each communication is assigned to a memory area with a different entry.

### 3.4.3. The OCCAM2 to VIS Compilation

The basic features of the OCCAM2 to VIS compiler are: i) *extensibility* to provide a compiler structure as flexible as possible with respect to future extensions and improvements and ii) *modularity* to easily maintain under control the single compilation passes. To meet these requirements, a strategy to afford the different compilation passes independently of each other has been adopted.

**Memory Management in VIS**

The VIS operates on a separated memory space for instructions and data. Concerning the data management of single procedures, the local data are kept unchanged until the procedure is completely executed. The data area must remain allocated until the procedure is completed. To allow this behavior, two different approaches can be adopted: the *static allocation* or the *semi-dynamic allocation*.

In the first approach, all data segments are allocated *statically*. A table is created to associate each procedure with the corresponding base pointer \((BP)\). Before a procedure call, the value of the \(BP\) of the called procedure is set to make possible the access of the procedure to its local data. If it is impossible to allocate statically all the data areas, as it occurs in the cases where the data segment is not large enough to contain all the data, a semi-dynamic approach has been adopted. In this case, the value of the \(BP\) is computed dynamically by the operating system the first time the procedure is called and maintained in the same table used by the static allocation. The next time the procedure is allocated, the correct value of the \(BP\) is set and the corresponding data area is activated.

When dealing with a concurrent language, memory fragmentation can become a major concern, due to the fact that data segments are not de-allocated in exactly the reverse order in which they are allocated. This problem has been addressed and a segmented memory approach has been preferred with respect to a more complex method based on garbage collection.
Another major concern is related to the method to pass parameters to procedures, that in VIS can be passed by copy or by reference. In general, the best techniques consist of passing to the called procedure the memory address where the data are stored. Pass-by-value parameters are an exception to the previous rule: in this case the values are directly copied rather than their addresses.

The limitations related to the addressing modes of VIS prevent the passing of the data address, since it is necessary a double indirection level to access the data. The address passing is also incompatible with the parallelism of the OCCAM2 language, because a local procedure, until not terminated, cannot modify in the external environment the value of the current parameters. This fact implies to pass the parameters by value-result, where the current parameters are copied in the data area of the calling procedure and used as local parameters. At the exit time, the called procedure executes a push on the stack of the parameters passed by reference, that are successively copied by the caller in the correct memory locations. In this scheme, it is possible to modify locally the values passed by copy: at the exit they do not modify the external environment since they are put on the stack and thus their local values will be lost.

**Directives for the Back-End**

During the VIS generation process, great emphasis has been posed on making compilation as independent as possible of the single procedures. For this purpose, some specific constructs have been added in the VIS code to allow local compilation and to detail to the compiler back-end and to the linker the information needed to the compilation process. A set of directives has been added to the VIS language to propagate the information required in the further steps of the compilation flow.

**Compilation Passes**

The compilation phase consists of several compilation passes to provide the possibility for the user to control the execution of the operations and to easily extend the current tool features. Each compilation pass performs an elementary operation. Basically, before getting to the final VIS, the code passes through a symbolic intermediate representation containing high level operators, to allow the simple execution of the register allocation and the resolution of the addressing modes.

- **Pass 1: Computation of requested temporaries.** Aim of the first pass is the computation of the number of temporaries necessary to each procedure. This process includes the
computation of the temporary variables for the expressions, the variables containing the results of boolean comparisons and the results of operating system function calls. During the computation of temporaries related to expressions, a simple algorithm has been applied to re-use the temporary already allocated.

- **Pass 1.1: Allocation of the static frame.** This pass creates two data structures allocated in memory to store the information related to the local variables and the formal parameters. During this pass, the memory necessary to store the data area is also computed. The first table contains also information related to register status to be used in the pass to solve the addressing modes. The second table contains an entry for each register to point to the associated variables.

- **Pass 2: Symbolic code generation.** After the definition of the variable table, a first translation occurs from the syntax-tree to an intermediate language. This intermediate representation is used to de-couple the operating code selection phase from register binding, thus rendering the compiler orthogonal and flexible.

- **Pass 2.1: Choice of addressing modes and register allocation.** After the symbolic code generation, this pass solves the addressing modes of abstract operands. As a matter of fact, the symbolic code still contains the operands related to the OCCAM2 code and the temporary variables allocated in the data area. Based on the possible addressing modes associated with each operator and the status of the datum (in memory, in a register, etc.), an addressing mode is chosen. If direct access is impossible, some additional code is generated to transfer the operand in a location (register or stack) suitable for the specific operating code.

- **Pass 3: VIS Generation.** After the previous pass, the code contains symbolic operators and the addressing modes related to each operator have been solved. This pass translates the intermediate code into the VIS code. Most of the operators remain unchanged after this pass, while some of them, such as those related to jump and comparison operators, are expanded.

- **Pass 4: Scheduling.** During this pass the generated VIS is completed by adding flag lists initialization and management, and suitable OS routine calls. The added code for a basic process strongly differs from that of a container process. The structure of the code for these two cases is:
• **Basic Process:** The added code for scheduling only have to keep the status of the process’ flag up to date. This often reduces to a MOVE operation, in some cases preceded by a CMP/BRANCH pair.

• **Container Process:** The code added for a container process is responsible for a number of actions. First the address and the length of the local flag list is pushed onto the stack and the appropriate scheduling routine is called. Then, depending on the routine’s return value, the control is passed to the selected sub-process. Finally, some actions must be taken to force the program control flow to return to the container process’ father.

During the scheduling pass, some back-end directives are added to allow the linker and the mapper to properly interpret and manage the generated code.

• **Pass 5: Linking.** The VIS compiler works on single OCCAM2 PROCs and generates separate VIS modules. To obtain a complete, working and simulatable VIS code, the different modules must be linked. The linking phase is accomplished by a specific stand-alone tool. During linking some operations are performed and the most important of them are listed below:
  • Generation of code for memory allocation table and channel table. This operation is the result of the interpretation of a set of directives present in the VIS code.
  • Generation of code for initialization of the local data segment of each procedure.
  • Procedure call resolution based on procedure prototype and signatures matching. Signatures of the procedure declarations and the procedure calls are present in the generated VIS code in the form of compiler directives.
  • External procedure linking. This step allows the user to link modules libraries into the design. To this purpose the OCCAM2 language has been enriched with the external procedure prototype declaration statement **EXTERN PROC.**
  • Operating system linking.

The linker is structured in such a way the linked VIS is generated in three passes. During the first pass the VIS code of each module is read in and a number of auxiliary data structure are built. The second pass performs the procedure call checking and generates the additional VIS code mentioned before. The third and last pass builds the final code and, at the same time, drops all the directives not useful any longer.
**3.4.4. The VIS to Target Assembler Mapping.**

The mapping phase translates the VIS code obtained after both compilation and linking into an assembler for a specific target processor. As reported in the previous section, the VIS Compiler generates a VIS code that, in order to be correctly mapped on a specific target architecture, has to be compiled for a number of registers corresponding to those available on the selected class of microprocessors. A similar consideration applies for the stack growth direction according to the actual architectural characteristics. These and other architecture peculiarities are supposed to be already present in the VIS code in the form of directives.

The main characteristic of the mapping phase lies in its efficiency and generality; in particular, this allows the designer an effective exploration of the solution space. The tasks performed in this step are: VIS instruction expansion and VIS directives expansion and/or dropping. The expansion of a VIS instruction into a sequence of specific assembler instructions is performed in two steps: *op-code selection* and *addressing mode resolution*. The *op-code selection* step selects a particular instruction code among those having the same functionality. In particular, the choice may be driven by a user-defined cost function aimed at locally optimizing a number of figures, like execution time, code area, power consumption or a combination of these.

The second step, the *addressing mode resolution*, is required only for those target instructions that cannot be mapped into three-operand instructions. As a matter of fact, while the number of registers used by the VIS code remains unchanged regardless of the target instruction set, the number of instructions required to properly translate the source VIS code may be greater than one.

The generated assembler code can still be optimized due to the fact that some inter-instruction effects at VIS level can be neglected during mapping.

**3.4.5. The Software Back-Annotation**

The back-annotation process occurs at two different abstraction levels: VIS-to-OCCAM2 and Assembler-to-VIS. The basic idea behind both levels is to keep trace, during the forward flow from the OCCAM2 specification down to the target assembler code, of the links connecting instructions between the different abstraction levels. In general, a single instruction of a higher level language translates into many instructions of a lower level one. In particular, in
the forward compilation flow each instruction of the generated code is associated with the originating statement.

The back-annotation process can start either at assembler or at VIS level. For the sake of completeness, the most general flow, starting from the target assembler to the OCCAM2, is detailed. The only assumption made here is that each line of the assembler code has already been annotated with the parameters interesting for the designer. The back-annotation Assembler-to-VIS is performed by scanning sequentially the assembler, adding up annotated values coming from the same line of the VIS code and associating the collected information to the corresponding VIS line. The back-annotation from VIS to OCCAM2 is performed in a likewise manner.

3.4.6. The Software Power Estimation

The proposed methodology for the estimation of power consumption operates at three abstraction levels: OCCAM2, VIS, and target assembler. The analysis of the OCCAM2 source code, when such a code is assigned to the SW partition, is based on the following assumptions:

- The timing and power characterization of the instruction set of target processors is given in the Processor Technology File;
- The timing and power data related to the memory sub-system are given in the Memory Technology File;
- The compilation of the OCCAM2 source code into the VIS pseudo-assembler with the suitable number of registers has been performed.

The proposed estimation methodology aims at building a parametric model of each VIS instruction or instruction class (e.g. data movement instructions, branch instructions, add instructions, ...) characterized by a suitable set of parameters specializing the model for the different target instruction sets. The power consumption of an OCCAM2 source code is estimated by means of a methodology structured in three consecutive steps.

In the first stage, the OCCAM2 code is compiled into VIS code limiting the family of the envisaged target microprocessor to be used. In this context the term family means a set of microprocessor architectures with the same number of general purpose registers. The second step computes the power consumption of each VIS instruction through both its model and the
set of specific parameters for the target processor. The specialization into a different processor of the same family simply implies the choice of the appropriate set of parameters. The last step of the methodology consists of the back-annotation of the computed values to OCCAM2 code.

The overall power estimation design flow has been depicted in Figure 3. The three estimation phases (namely LEVEL 2, LEVEL 1, and LEVEL 0) are detailed in the following.

**LEVEL 2 Power Annotation**

The lowest level of abstraction for the software implementation of an OCCAM2 model description is its translation into the target microprocessor assembler. The power consumption calculation performed at the lowest level of abstraction is referred to as **LEVEL 2 power annotation**. To perform the **LEVEL 2** annotation step, a detailed knowledge of the power and timing characteristics of specific processors and memory hierarchy is needed. In particular, for each instruction (or class of instructions) and each combination of addressing modes, the power consumption and the related duration (in terms of clock cycles) are supposed to be given. All these power figures are collected, in a suitable format, into the technology files. The **LEVEL 2** annotation mainly consists of matching each instruction of the compiled and mapped assembler code against its power dissipation value in the specific technology file.

**LEVEL 1 Power Estimation**

One level of abstraction above the target assembler code is the VIS code, generated for the suitable number of registers of the family of microprocessors envisaged for the application. The power evaluation performed at this level is referred to as **LEVEL 1 power estimation**. The power evaluation at this level is based on an analytical parametric model of each class of VIS instructions (or, of each VIS instruction) and a set of specific parameters associated with the target microprocessor.

The definition of an analytical parametric model of each class of VIS instructions requires a detailed knowledge of the compilation and mapping phases. The knowledge of the compilation process and a high level of control over it are the reasons why a specific compiler has been developed. Furthermore, a set of compiled, mapped and back-annotated (from **LEVEL 2** to **VIS**) benchmarks is needed for microprocessor-specific parameters extraction. By means of both the model and the parameters available, the **LEVEL 1** estimation can be performed in a similar way to that outlined for **LEVEL 2** annotation; the main difference is
that the basic power consumption figures are computed rather than collected from the technology files.

**LEVEL 0 Power Estimation**

At the OCCAM2 level, the power assessment is achieved through a back-annotation of the power consumption figures calculated at the VIS level. Basically, the VIS-to-OCCAM2 back-annotation process slightly differs from those adopted for the assembler-to-VIS back-annotation for the following reasons. First, at the OCCAM2 level, the overhead due to scheduling (implemented as OS function calls) is not explicit: its contribution is hidden in the semantics of the container processes. To obtain a realistic power estimation, the scheduling cost has to be adequately distributed among the container process itself and its sub-processes. Second, the translation of OCCAM2 into VIS imposes an additional amount of code for memory initialization and ancillary data structures initialization and maintenance, not logically related to specific OCCAM2 statements. The cost of these operations has been taken into account.

The result of the back-annotation is a source OCCAM2 description in which every single line is annotated with a static estimation of the power consumption. To obtain a dynamic power characterization code, a profiling is mandatory.

**3.4.7. The Software Power Profiling**

The power analysis carried out so far is fundamentally a static analysis, while on the contrary a dynamic analysis is mandatory to provide more accuracy to the final power estimates, mainly to take in consideration the effects on power of data dependencies. The software profiling consists of two main phases: i) the LEVEL 0 functional profiling of the OCCAM2 code, and ii) the LEVEL 1 structural profiling of the VIS code. During the LEVEL 0 profiling, the functional instructions of the OCCAM2 source code are simulated. The OS calls and scheduling overhead are neglected at this level. The profiling carried out at this level results in a limited accuracy, but it requires short execution times. During the LEVEL 1 profiling, the instructions of the VIS code, after compilation and scheduling, are simulated. This level profiling results in a higher accuracy with respect to the LEVEL 0, but at the cost of longer execution times.
3.5. Summary

This chapter focuses on the problem of the power assessment of the SW-bound part of embedded systems. The proposed estimation methodology is integrated into a more general HW/SW co-design methodology for control-dominated embedded systems. The power metrics that have been defined for the SW code can be applied during or after the partitioning phase, to evaluate the conformance of the different partitioning solutions to the system-level requirements and to early re-target architectural design choices. The proposed power model is suitable for a target system architecture, which is quite general and common for the most part of embedded systems applications.

The power analysis carried out at the SW-level is tightly related to the characteristics of the system architecture in terms of processor and memory sub-system. This dependence of the processor and memory is taken into account through two technology files containing the timing and power characterization data. The approach is thus re-targetable towards several system architectures, since it can easily accept characterization data coming from several commercial microprocessors and memory architectures.

Furthermore, the main contribution of our model is related to its independence of the high-level specification language and of the assembler of the target processor. The independence has been achieved by introducing an intermediate virtual assembler level and a specific SW compilation strategy. The power estimation is essentially a static analysis. However, a dynamic analysis has been defined with the purpose to increase the accuracy level of the power figures, mainly to take into consideration the effects of data dependencies.