Title	Average-case analysis of power consumption in embedded systems
Authors	Zeinolabedini, Nasim
Publication date	2015
Original Citation	Zeinolabedini, N. 2015. Average-case analysis of power consumption in embedded systems. PhD Thesis, University College Cork.
Type of publication	Doctoral thesis
Rights	© 2015. Nasim Zeinolabedini. - http://creativecommons.org/licenses/by-nc-nd/3.0/
Download date	2025-07-04 06:51:57
Item downloaded from	https://hdl.handle.net/10468/3375

Average-Case Analysis of Power Consumption in Embedded Systems

Nasim Zeinolabedini

**Thesis submitted for the degree of
Doctor of Philosophy**

NATIONAL UNIVERSITY OF IRELAND, CORK

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

August 2015

Head of Department:	Prof Nabeel Riza
Supervisors:	Dr Emanuel Popovici

Contents

List of Figures

List of Tables

I, Nasim Zeinolabedini, certify that this thesis is my own work and has not been submitted for another degree at University College Cork or elsewhere.

_____

Nasim Zeinolabedini

To my dear sister, Shamim

Acknowledgements

I would like to thank my supervisor, Dr. Emanuel Popovici, and all of my colleagues for having me as a part of the Embedded System Group through these years.

I am very thankful to my family for giving me the chance to continue my education away from them and away from my country. Specially I would like to thank my sister, Shamim, for being always encouraging and supportive to me. I am sorry for not being by her side during the difficulties she experienced through these years.

I would like also to thank all of my friends for their company, their advises and their supports that helped me to handle the hard situations.

Finally, I give my most sincere and heartfelt thanks to Prof. Nabeel Riza that completion of this work was never possible without his kind support.

Abstract

Power efficiency is one of the most important constraints in the design of embedded systems since such systems are generally driven by batteries with limited energy budget or restricted power supply. In every embedded system, there are one or more processor cores to run the software and interact with the other hardware components of the system. The power consumption of the processor core(s) has an important impact on the total power dissipated in the system. Hence, the processor power optimization is crucial in satisfying the power consumption constraints, and developing low-power embedded systems.

A key aspect of research in processor power optimization and management is "power estimation". Having a fast and accurate method for processor power estimation at design time helps the designer to explore a large space of design possibilities, to make the optimal choices for developing a power efficient processor. Likewise, understanding the processor power dissipation behaviour of a specific software/application is the key for choosing appropriate algorithms in order to write power efficient software.

Simulation-based methods for measuring the processor power achieve very high accuracy, but are available only late in the design process, and are often quite slow. Therefore, the need has arisen for faster, higher-level power prediction methods that allow the system designer to explore many alternatives for developing power-efficient hardware and software.

The aim of this thesis is to present fast and high-level power models for the prediction of processor power consumption. Power predictability in this work is achieved in two ways: first, using a design method to develop power predictable circuits; second, analysing the power of the functions in the code which repeat during execution, then building the power model based on average number of

repetitions.

In the first case, a design method called Asynchronous Charge Sharing Logic (ACSL) is used to implement the Arithmetic Logic Unit (ALU) for the 8051 microcontroller. The ACSL circuits are power predictable due to the independency of their power consumption to the input data. Based on this property, a fast prediction method is presented to estimate the power of ALU by analysing the software program, and extracting the number of ALU-related instructions. This method achieves less than 1% error in power estimation and more than 100 times speedup in comparison to conventional simulation-based methods.

In the second case, an average-case processor energy model is developed for the Insertion sort algorithm based on the number of comparisons that take place in the execution of the algorithm. The average number of comparisons is calculated using a high level methodology called MOdular Quantitative Analysis (MOQA). The parameters of the energy model are measured for the LEON3 processor core, but the model is general and can be used for any processor. The model has been validated through the power measurement experiments, and offers high accuracy and orders of magnitude speedup over the simulation-based method.

Chapter 1

Introduction

1.1 Motivation

In the past decade, the use of embedded systems has grown in almost every aspect of our daily lives, including simple household appliances, transportation systems, and many communication, recreation and entertainment products. As a result, the design and implementation of efficient embedded software and hardware systems have gained utmost importance.

In the design of embedded systems, in addition to the need that the system has to produce the desired outputs for the given inputs, there are a number of other requirements which must be satisfied. These requirements could be imposed by user expectations or resource constraints. Some examples of these types of requirements are limits on the response time, memory space, battery capacity or channel bandwidth. These requirements are integral to the correct operation of the system. For instance, the response time of the electronic braking system in automobiles, or the power consumption of remote sensor nodes that scavenge their energy from the environment are critical for the correct functionality of these systems [1].

Power efficiency is one of the most important requirements in the design of embedded systems since such systems are generally driven by batteries with limited energy budget or have a restricted power supply. The power consumption becomes a more critical element in the design of highly integrated systems with a constant increase in the number of transistors per die, smaller chip area, and a higher operating frequency from older to newer technology nodes.

In every embedded system, there are one or more processor cores to run the software and interact with the other hardware components of the system. The power consumption of the processor core(s) has an important impact on the total power dissipated in the system. Hence, the processor power optimization is crucial in satisfying the power consumption constraints, and developing low-power embedded systems.

A key aspect of research in processor power optimization and management is "power estimation". Power estimation is important for several technical and commercial reasons. Having a fast and accurate method for processor power estimation at design time helps the designer to explore a large space of design possibilities to make the optimal choices for developing a power efficient processor. This can be done well before the actual processor is designed, fabricated and tested. Likewise, understanding the processor power dissipation behaviour of a specific software/application is key for choosing appropriate algorithms in order to write power efficient software [2]. From a commercial point of view, accurate power estimation at the design stage avoids costly re-design cycles, and leads to a product with better power consumption characteristics, and thus ensures higher profitability.

The simulation-based methods for measuring the processor power achieve very high accuracy, but they are available only late in the design process, and are often quite slow. Thus, it is difficult to exploit these methods in order to measure the

power consumption of the processor for a large number of hardware or software design alternatives. For this reason, the need has arisen for the faster, higher-level power prediction methods that allow the system designer to explore different alternatives for development of power-efficient hardware and software.

The motivation of this work is taking a step toward building a framework to estimate the processor power consumption with high speed and accuracy early in the design flow of embedded systems. The work presented in this thesis is mostly focused on the estimation of the processor power for a given program code which can help software developers in writing power optimized embedded software code. In the rest of this chapter, the previous work in this area is outlined, and the contribution of the thesis is explained.

1.2 Prior Work

The proposed methods in the area of processor power estimation can be classified into five categories: methods based on architectural simulation, system-level models, hardware performance counters, on-chip temperature profile and program execution profile. The first two categories (architectural-level models and system-level models) are the only methods available at the design stage which are useful to avoid re-design cycles and reduce the time to market in the processor design. Three other categories are available at runtime to estimate the power consumption of the software application. In the following, each of these categories are concisely described. In Chapter 2, more detail of the methods proposed in the literature in each category is presented.

Architectural-level models for power estimation are based on calculation of the load capacitance of each functional unit inside the processor using circuit simulation, analytic equations or empirical data. The activity factor of each functional

unit during the execution of the test programs is generated through simulation, and is applied to the processor model to compute the power/energy consumption. The most famous work in this category is a power estimation framework called Wattch [3]. In this framework, the instruction cache, branch predictor, wakeup logic, register file, instruction window and the global clock are modeled at architectural-level, and the access counts for functional units are calculated using SimpleScalar [4] simulator. Other examples of architectural-level power estimation are the works presented in [5, 6, 7].

System-level models are communication-oriented models which describe a system of processing elements and the interactions between them. An important subset of such kind of models are Transaction Level Models (TLMs) [8]. TLMs model each message or event between processing blocks as a basic transaction. Most of the works with system-level models are presented to estimate power consumption for SystemC-based designs. In these works, the events relevant to power consumption are captured by modifying the SystemC kernel or using additional custom Application Program Interface (API). Some examples of this type of approach are presented in [9, 10, 11, 12, 13, 14].

Hardware performance counters are a set of special-purpose registers built into modern microprocessors to store statistics about the activity of different subsystems in the processor. These registers are typically readable by kernel-level or user-level software entities. For the processor power estimation, the performance counters that have a good correlation with the measured power are selected, and the power model is built as a function (F) of their sampled values. Different techniques are used in the literature for determining function F. Some of the approaches [15, 16] are purely mathematical and use regression based methods to solve the problem. Some approaches [17, 18, 19] use micro-benchmarks to generate events in a specific performance counter to determine the impact of

each counter on the total power. There are some other approaches [20, 21, 22] which combine mathematical methods with micro-benchmark-based methods.

In on-chip temperature profile techniques, the power estimation is based on the link between the power consumption and the temperature of a die. The problem of finding the power consumption map of a die, given the temperature map is known as the Inverse Heat Conduction Probem (IHCP) [23]. To collect the temperature data either the InfraRed (IR) photograph of a die or embedded performance counter based thermal sensors can be used. Some approaches [24, 25, 26] in this category solve the IHCP problem assuming that temperature values are exact, and some other approaches [27, 28] consider some thermal noise.

The last category of approaches for processor power estimation is based on program execution profile. The work in this thesis also fits in this category. The approaches in this category can be divided into three groups: instruction-level methods, function-level methods and functional unit based methods.

In instruction-level methods, first the processor energy consumption of each instruction is characterized, and then the program code is analysed to get the instruction counts. The total energy consumption is obtained by multiplying the number of executed instructions of each type by their corresponding energy values. Some of the works [29, 30] based on instruction-level power characterization are data independent. It means that the power models are built without taking the impact of the instruction operands into account. Some other works [31, 32] in this area are semi-data dependent. In these works, some parameters like the inter-instruction effects, circuit state, pipeline stalls and cache misses are also considered in building the power model. The last set of works [33, 34] are data dependent which take the effect of program input data on the power consumption into account.

Function-level power estimation approaches are based on the processor power

characterization at the level of functions and library calls. In these approaches, the number of executions for the frequently invoked functions are counted, and total power is estimated by multiplying this number by the corresponding energy values. The examples of Function-level power estimation are presented in [35, 36, 37].

In functional unit based methods, the activity of the relevant functional units in the processor during the execution of the code is extracted as task parameters by analysing the program code. The total energy is computed by applying these task parameters to arithmetic models developed for the functional units. This method is also known as Functional Level Power Analysis (FLPA). In [38, 39, 40], this method is used for processor power estimation.

1.3 Contributions of the Thesis

The work presented in this thesis is comprised of two main parts. The first part of the thesis can be classified as an instruction-level approach in the category of program execution profile based methods for processor power estimation. The second part represents a function-level approach in the same category.

In the first part of the thesis, a power estimation method is proposed which is based on a power predictable design methodology. This methodology that is called Asynchronous Charge Sharing Logic (ACSL) is a dynamic design style which offers two main properties. The first property is that it has lower power when compared with the other dynamic circuits, and the second property is that its power usage is almost constant and independent from the input patterns. This second property makes the ACSL circuits power predictable.

In this work, the Arithmetic Logic Unit (ALU) of 8051 microcontroller is implemented in ACSL. The power consumption of the arithmetic and logic operations

of this ALU has a very small variability, and is independent from the input data. This makes it possible to estimate the power usage for the ALU only by knowing the number and type of the operations it performs. The 8051 microcontroller is chosen in this work since it is a common microcontroller in embedded systems applications due to being widely available, and having many variants with different peripherals.

The power prediction method for the ACSL ALU is based on using an 8051 Instruction Set Simulator (ISS) to run the programs, and analysing their instruction trace to extract the number of ALU related instructions. This also provides the information on the number of times each ALU operation is used by the instructions during the execution of the program on the 8051. The average power of the ALU is then calculated by multiplying this number by the power consumption associated with each operation. This method can estimate the power with less than 1% error, and over 100 times faster than the gate-level simulation. Considering that the ALU is quite a small component in the processor core, when this method is applied to the entire core the speedup will be much higher.

The novelty of this work lies in the fact that in other methods the processor hardware is designed and implemented without considering the power predictability of the final circuit. The accuracy of these methods is affected by the fact that there is a high dependency of the power consumption on the input data profile which is often unknown at design time. As a result, it is hard to capture the behaviour of these circuits in terms of power to develop accurate power models.

In the second part of the thesis, an average-case processor energy model for the Insertion sort algorithm is proposed. This model is based on the average number of comparisons in the sorting algorithm that is calculated using MOdular Quantitative Analysis (MOQA).

MOQA is a high level methodology for static average-case analysis of the program

codes. This methodology enables the prediction of the average number of basic steps during the execution of a program which facilitates the estimation of the complexity measures such as average time or average power consumption.

The average-case analysis of the design metrics in embedded systems is important because it provides useful insight about the typical behaviour of the system, and complements the worst-case information to help the designer in taking better strategies in implementing an efficient system.

The energy model is built based on the average number of times that each part of the program code is repeated during the execution on the proceesor core, and the energy consumption of each part. In this work, the parameters of the energy model are determined for the LEON3 processor core, but the model is general and can be used for any processor.

This energy model enables the static estimation of the average-case processor energy consumption for the Insertion sort program for any given size of the input list. The accuracy and speedup of the model has been evaluated for the LEON3 processor through the power meaurement experiments. The model achieves high accuracy, and estimates the average energy in a fraction of a second in compare with gate-level simulation method which can take days or weeks to be run for a reasonable number of input samples.

1.4   Structure of the Thesis

The structure of the chapters in this thesis is as follows: In Chapter 2, the previous work in processor power estimation area presented in the literature is reviewed. In Chapter 3, the design concept of the Asynchronous Charge Sharing Logic (ACSL) is introduced, and the structure and general operation of the ACSL circuits are described. In Chapter 4, the implementation of the 8051 Arithmetic

Logic Unit in ACSL is explained. In Chapter 5, the power prediction method for the 8051 ACSL ALU is presented. Finally in Chapter 6, the average-case processor energy model for the Insertion sort algorithm is described.

Chapter 2

Processor Power Estimation Techniques

In this chapter, different methods proposed in the literature for processor power estimation are explored. In general, every processor power estimation method is composed of two parts: model and input. The model is independent from the program code, and can be built in different levels of abstraction. The input is derived from the execution of the program code. The energy or power is estimated by applying the input to the model.

The power estimation methods described in this chapter are classified in five categories: methods based on program execution profile, architectural simulation, system-level models, hardware performance counters and on-chip temperature profile. Some of these methods work at design time, and are suitable for early stage architectural exploration. Some other methods work at runtime, and are useful for developing power efficient application software. The most prominent works presented in the literature in each category are explained.

The structure of this chapter is as follows: In Section 2.1, the methods based on program execution profile are described. In Section 2.2, the methods based

on architectural parameters are presented. In Section 2.3, the methods using system-level models are investigated. The performance counter based methods and thermal profile based methods are introduced in Section 2.4 and Section 2.5 respectively.

2.1 Methods Based on Program Execution Profile

Processor power estimation methods based on the program execution profile use a group of instructions as the basic atomic unit. The generic approach is to characterize the energy consumption of each instruction, and then analyse the code to get the instruction counts. It is also possible to do the characterization and profiling at a higher level of granularity e.g. functions or traces. Some approaches predict the functional unit access counts by using program analysis or from instruction access counts. Subsequently, they compute the total energy by multiplying the access counts with pre-characterized energy values. In the following, each of these approaches are explained in more detail.

2.1.1 Instruction-Level Methods

All instruction-level approaches have a similar structure. In the first phase of these approaches, a profiling run performs which executes different pieces of code repeatedly, and measures their energy usage. This allows the estimation of the energy associated with the set of instructions. In the second phase, some counters are embedded in the software that gives the execution frequency of each basic block. In the final phase, the total energy consumption is obtained by multiplying the number of executed instructions of each type by their corresponding energy

values. The estimated energy is divided by the execution time to yield the average power.

2.1.1.1 Data Independent Approaches

In [29], a tool called Jouletrack is proposed to estimate the processor power at the basic block/instruction level. This tool also calculates the processor leakage power. In [41] a similar approach is taken but the leakage power is not modeled explicitly. Their model considers all sources of power that cannot be classified as dynamic power as a lumped constant.

In [42], the instruction-level profiling method is extended to VLIW processors. These processors execute a group of instructions as a bundle, therefore, the processor power is characterized at the level of each bundle of instructions. In this way, the power consumed by an instruction depends on three factors : opcode/operands of the instruction, the pipeline/circuit state, and the other instructions in the bundle. In [43], a simpler approach is taken for VLIW processors. In this approach, the instruction trace is passed to an architectural power simulator which is calibrated with RTL models.

In [30], a very low level method for modeling the timing and power of C programs is proposed. In the proposed method, every statement in the C language is broken to a set of micro-instructions which resembles a very primitive RISC ISA. The power consumption is characterized for each such micro-instruction. Subsequently, the software counters are embedded in a block of C statements for each high level construct such as a switch case or a loop statement. The total power is estimated based on the access counts.

2.1.1.2 Semi-Data Dependent Approaches

For the first time, a systematic approach for the estimation of the processor power during the execution of the instructions, with accounting for the inter-instruction effects, was proposed in [32, 44, 31]. In these works, three types of inter-instruction effects are considered: pipeline stalls, change in circuit state and cache misses. The average number of switching bits for every consecutive pair of instructions are measured through the extensive simulations. The other effects are modeled by adding a constant to the total instruction power/energy. Considering all the above parameters, an instruction-level power model is presented to estimate the total energy for the processor (E_P) during the execution of a given program (P).

$$E_P = \sum_i (B_i * N_i) + \sum_{i,j} (O_{i,j} * N_{i,j}) + \sum_k E_k \tag{2.1}$$

where for each instruction i, B_i is the base cost, and N_i is the number of times it is executed. For each pair of consecutive instructions (i, j), $O_{i,j}$ is the circuit state overhead, and $N_{i,j}$ is the number of times the pair is executed. E_k is the energy contribution of the other inter-instruction effects, k (stalls and cache misses), that occures during the execution of the program.

In [45], this energy model is slightly simplified by using a constant power dissipation per instruction. This includes the effect of some of the inter-instruction factors. In [46], the impact of instructions inter-dependency on energy is accounted for based on characterizing the energy for pairs of instructions, and building a fine-grain 2-instruction-based model.

2.1.1.3 Data Dependent Approaches

In [33], the work of [31] has been extended by introducing a new instruction model able to consider the influence of the operands distribution on the processor power consumption. This model tries to relate the instruction power usage to the internal switching activity induced by operands. The power model in this work (given in Equation 2.2) is developed for the execution unit (EX+MEM stage) which is the main source of the power consumption in the processor, but it is extensible to the entire microprocessor.

$$Power = K_1 n_1 + K_2 n_2 + ... + K_n n_n + K_0 + C_{ij} \qquad (2.2)$$

In this model, coefficients K_i and variables n_i are respectively the weights and the number of transitions of the activity indices. Activity indices are the elements inside the processor that have a strong impact on the power consumption. The activity indices used in this work are the data write bus, address bus, and ALU bus. K_i and n_i parameters represent the average effect of the operands on the power, and are determined using uniformly distributed operands. K_0 is the power cost for null switching activity on activity indices. It is the minimum cost for the particular instruction. C_{ij} is the changing-instruction cost between instructions i and j. That is a fixed cost due to the changing in the datapath configuration because of changing instruction. C_{ij} must be added to j instruction when it is preceded by i instruction.

In [34], an automated method is proposed for characterizing the energy usage of the instructions. In this work, the energy per cycle is decomposed into four parts: instruction-dependent energy dissipation, data-dependent energy dissipation, energy dissipation of the cache system and the dissipation of all external components including the bus system, memories, and peripherals. Since a complete character-

ization of the whole range of values of the operands is only theoretically possible, the data-dependent energy consumption is modeled in this work by means of linear regression.

In [47], a multi-granularity power model is proposed at functional, architectural and cycle-accurate micro-architectural stages of the design flow. These models offer a designer the flexibility to trade off estimation accuracy with estimation/simulation effort. A 3-D power contribution LUT is created that holds the power dissipation for each instruction, at each pipeline stage, for every functional unit in the processor. For improved accuracy, a set of three such 3-D LUTs is created, corresponding to average power, minimum power and maximum power depending on the operands values.

2.1.2   Function-Level Methods

Function-level power estimation approaches are based on the processor power characterization at the level of functions and library calls. In these approaches, number of executions for the frequently invoked functions are counted by putting software counters at their entry point.

In [35, 48], a "power data bank" is built which stores the power information of the built-in library functions and basic instructions. In this work, the machine code is decomposed into library functions and user-defined functions. Then program profiling/tracing tools are used to get the execution information of the target software. Next, the total energy consumption and execution time is evaluated based on the "power data bank", and their ratio is taken as the average power.

In [36], two kinds of macro-modeling techniques for high-level energy estimation for software functions are proposed: Complexity-based macro-modeling and profiling-based macro-modeling. Both of the techniques are based on linear re-

gression models. Complexity-based macro-modeling uses the algorithmic complexity of the functions to determine the macro-model template. For example, for an algorithm which has an average-case complexity of $O(n^2)$, the energy macro-model is:

$$E = c_1 + c_2 n + c_3 n^2 \tag{2.3}$$

where n is the input size. The regression analysis is used along with low-level software power measurements to obtain the unknown coefficients c_is.

In profiling-based macro-modeling, internal profiling statistics for the functions are used as parameters in the energy macro-models. Several variants of profiling-based macro-modeling is proposed, starting from simple basic-block profiling, to different lengths of basic-block correlation profiling and Ball-Larus path [49] correlation profiling.

[37] presents a systematic automated methodology for macro-model generation for frequently used functions/libraries. This work is based on the observation that large embedded software programs are rarely written from scratch, and a large fraction of the execution time is due to the reused software components (including embedded-operating systems, middleware, run-time libraries, domain-specific algorithm libraries, etc.). The energy consumption macro-models for these functions/libraries is generated by determining the right set of parameters, collecting data through simulation, and building the models using symbolic regression.

2.1.3   Functional Unit Based Methods

In this type of approach, the activity of the relevant functional units in the processor (e.g. fetch unit, processing unit, clock network, internal memory and others) during the execution of the code is extracted as task parameters by analysing

the program code. The total energy is computed by applying these task parameters to arithmetic models developed for the functional units. This method is also known as Functional Level Power Analysis (FLPA). Figure 2.1 depicts the principal of this approach.

In the work presented in [50, 38, 51, 52], a tool called SoftExplorer is developed to perform power and energy estimation of generic C programs for DSP applications at both assembly level and C level. This work is based on three models: a processor model, an algorithm model and a compiler model. To perform estimation from the assembly code, only the two former models are needed. The model for the processor represents the way the processor's power consumption varies with its activity. The model for the algorithm links the algorithm with the activity it induces in the processor. To perform estimation at the algorithmic level (C level), a model for the compiler is also needed to take the effect of compiler behavior on the assembly code into account.
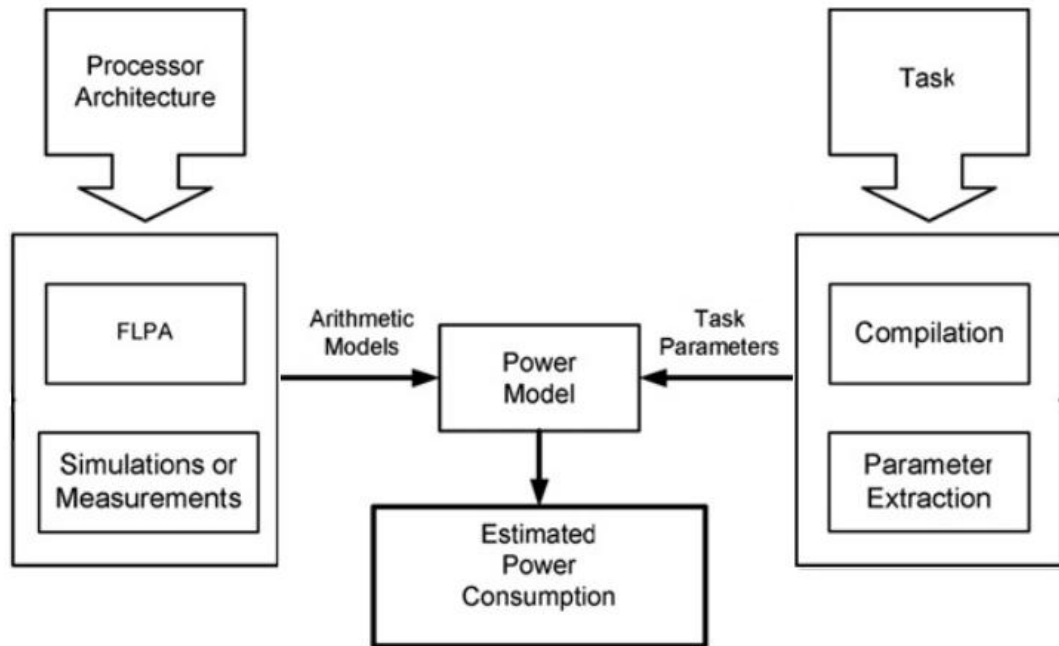
Figure 2.1: The basic FLPA principle

The processor model is built by identifying the functional units, and characterizing the energy consumption of each unit through the physical measurements. The algorithm model extracts the values of a few parameters from the code. These values are injected in the processor model to estimate the power consumption. The compiler model represents the behavior of the compiler, and how it will allow the algorithm to use the processor's resources.

Two sets of parameters are defined in this work: architectural and algorithmic parameters. The architectural parameter values depend on the processor configuration settled by the designer. This includes: clock frequency (F), memory mode (MM), data mapping (DM) and data width (W) during Direct Memory Access (DMA). The algorithmic parameters depend on the code execution, and represent the activity rate of the functional units and their interactions. Five algorithmic parameters are identified: fetch rate (α), execution rate (β), cache miss rate (γ), activity rate between the data memory controller and the DMA (ε) and Pipeline Stall Rate (PSR).

In [39, 53], an Energy-Aware Compilation (EAC) framework is presented that estimates and optimizes energy consumption of a given code, taking as input the energy/performance constraints, architectural and technological parameters and energy models. The energy consumption in this work has been modeled for data-path, clock network, buses, caches, and main memory. Some of the application-dependent parameters extracted from the code using the compiler are: data-path accesses, number of execution cycles, bus transactions, cache misses and memory transactions.

In [54], the power consumption of the processor functional blocks has been modeled in terms of parameterized arithmetic model functions. A parser which allows to analyze automatically the assembler codes has been implemented. This parser yields the input parameters of the arithmetic functions, e.g. the achieved degree

of parallelism or the kind and number of memory accesses.

In [40], a hybrid method for processor power estimation is presented which combines FLPA and instruction-level modeling approaches. In this work, an instruction dependent part is added to the FLPA in order to achieve high estimation accuracy.

2.2 Methods Based on Architectural Parameters

The generic approach for power estimation at architectural-level consists of the following steps: First, the load capacitance for each functional unit in the processor is calculated using either circuit simulation, analytic equations or empirical data. Next, the functional unit activity factor (α) is generated through simulation. Finally, the total power is computed using Equation 2.4.

$$P = \alpha C V^2 f \qquad (2.4)$$

In [3], a framework called Wattch is presented for architectural-level power analysis and optimization. In this framework different blocks are classified by their structure and functionality. A suit of parameterized power models for different hardware structures and on per-cycle resource usage counts is generated through cycle-level simulation. Some of these hardware structures include instruction cache, branch predictor, wakeup logic, register file, instruction window and the global clock. Wattch calculates the access counts for functional units using SimpleScalar simulator.

In [7], a new power, area, and timing modeling framework called McPAT (Multicore Power, Area, and Timing) is introduced which advances the state of the art

in several directions in compared to Wattch. First, McPAT enables architects to use new metrics combining performance with both power and area such as energy-delay-area^2 product (EDA^2P) and energy-delay-area product ($EDAP$), which are useful to quantify the cost of new architectural ideas. Second, McPAT models more than just dynamic power, which is critical in deep-submicron technologies since static power has become comparable to dynamic power. All three types of power dissipation (dynamic, static, and short-circuit power) are modeled to give a complete view of the power envelope of multicore processors. Third, McPAT provides a complete, integrated solution for multithreaded and multicore/manycore processor power. Fourth, McPAT handles technologies that can no longer be modeled by the linear scaling assumptions used by Wattch.

In [55, 6], an architecture-level power estimation framework called SimplePower is presented which also takes the impact of input values on power consumption into account in compare with previously mentioned architecture-level models that only consider the number of accesses to the functional units.

2.3   Methods Based on System-Level Models

System-level models are communication-oriented models which describe a system of processing elements and the interactions between them. An important subset of such kind of models are Transaction Level Models (TLMs) [8]. TLMs model each message or event between processing blocks as a basic transaction. SystemC [56] is one of the most common TLM-based languages for high-level modeling which contains a basic event-driven simulation engine (kernel), and provides an interface for modeling system-level designs.

Most of the works with system-level models are presented to estimate power consumption for SystemC-based designs. The first set of approaches modify the

SystemC kernel. These approaches are more generic and user-friendly, but they are not very flexible. The second set of approaches use additional custom Application Program Interfaces (APIs) to capture events relevant to power consumption. These approaches make no modification to the kernel.

2.3.1   Kernel Based Approaches

In [11, 10], a framework is presented for the estimation of area, power and delay characteristics of hardware systems modeled at the Register-Transfer Level (RTL) using the SystemC modeling language. The framework also allows for dynamic power profiling and analysis based on the state of the modeled circuit. In this work, SystemC kernel is modified to calculate the number of 0-to-1 transitions at input and output ports of each component. The power consumption is then calculated using the number of input and output transitions and the type of the component.

[12] introduces a modeling and simulation technique that extends TLM method and modifies the SystemC kernel to support multi-accuracy models and power estimation. This allows the designer to trade off between simulation accuracy and speed at runtime. Another work in this area is presented in [57] for designs that have voltage scaling. [58] propose a tool called PowerSim on the same line. PowerSC [59] is a commercial tool that supports power estimation for SystemC, and PowerKernel [60] is one of the prominent open source tools in this domain.

2.3.2   API Based Approaches

In [13], a high-level power estimation methodology based on SystemC and Aspect Oriented Programming (AOP) is proposed. AspectC++ [61] is used to define special power-aware *aspects*. These *aspects* can be viewed as configuration files to

link the power aware APIs and SystemC functionality model. This methodology supports multi-macro-models and multi-accuracy power estimation.

In [62, 14], a VHDL to SystemC translator is described which can insert power simulation routines in the SystemC code. In contrast to previous approaches, the API calls in this work do not need input values, and estimate the average number of transitions per operation using the stochastic methods. For this reason, they are significantly faster.

2.3.3   Other Approaches

In [63, 64, 65, 66], a system-level methodology for energy and performance estimation of System-on-Chip (SOC) architectures is proposed. This methodology operates at a very high abstraction level, namely the functional untimed level. For this reason it has been called Funtime. Funtime approach achieves more speed in compare to TLM methods since it needs no architectural-level simulation, and all information is inferred from functional level.

Funtime consists of three layers. The bottom layer relies on building a library of IP energy and performance models, where each IP's functionality is pre-characterized through gate-level simulation. At the intermediate layer, applications are run and profiled on a development host (a common PC). This allows to create a trace of the executed source code, which is then mapped to the assembly code of the target architecture. Once the target trace is inferred, energy and performance figures can be extracted by using the IP models from the bottom layer. The top layer is a refinement layer that accounts for the presence of caches and for the fact that multiple applications normally run concurrently, share the same resources and are controlled by an operating system. Statistical models are built to account for the impact of each of these components.

2.4   Performance Counter Based Methods

Hardware performance counters are a set of special-purpose registers built into modern microprocessors to store statistics about the activity of different subsystems in the processor. These registers are typically readable by kernel-level or user-level software entities.

Assuming that vector V represents the sampled values of performance counters, the total power can be computed using Equation 2.5.

$$P = F(V) = VW + P_{idle} \tag{2.5}$$

In this equation, the power (P) is a function (F) of the sampled performance counter values. W is a vector of weights, where W_i represents the weight associated with i^{th} performance counter. P_{idle} represents the static power.

Three different approaches are proposed in the literature for determining function F. The first set of approaches are purely mathematical. They view the problem as an optimization problem, and try to find a least squares based estimate. The works presented in [15, 67, 68] are in this category. In the second set of approaches, the W_i coefficients are determined by measuring the total power dissipated by a micro-benchmark that exclusively generates events for the ith performance counter. The examples of using this approach is presented in [17, 18, 19]. The third approach combines purely mathematical approaches with with micro-benchmark based approaches. In this type of approach, additional constraints are enforced in the optimization process by taking inputs from micro-benchmark based methods or data from architectural simulators. [20, 21, 22] are in this group.

2.5   Thermal Profile Based Methods

In this type of method, the power estimation is based on the link between the power consumption and the temperature of a die. The problem of finding the power consumption map of a die, given the temperature map is known as the Inverse Heat Conduction Probem (IHCP) [23]. This problem can be formulated according to Equation 2.6.

$$P = AT + C\frac{dT}{dt} \tag{2.6}$$

Where P and T are column vectors representing the power and temperature of each core, and C is a diagonal matrix which contains the thermal capacitance of each node. The challenge is to estimate the matrix A which is called the conductance matrix.

The first step to solve the IHCP problem is to collect the temperature data. This can be done either through an IR (InfraRed) photograph of a die [24, 25, 27], or using embedded performance counter based thermal sensors [26]. In either case, the same mathematical techniques need to be used. The first set of approaches try to solve the Equation 2.6 by assuming that the temperature values are exact. The work presented in [24, 25] is based on this approach. The second set of approaches consider some thermal noise which can arise due to the limits of heat transfer or measurement error. [27, 28] are in this category.

2.6   Summary

In this chapter, the most important processor power estimation methods proposed in the literature are investigated. Some of these methods work at design time, and are useful for early stage architectural exploration. Some other methods work

at runtime, and are suitable for developing power efficient application software. The methods are classified in five categories, and the most prominent works in each category are described. The methods in these categories are based on program execution profile, architectural simulation, system-level models, hardware performance counters and on-chip temperature profile.

Chapter 3

Asynchronous Charge Sharing Logic (ACSL)

In this chapter, the design concept of the Asynchronous Charge Sharing Logic (ACSL) is introduced. ACSL is a dynamic design style which is proposed in [69] as an ultra-low power methodology. ACSL circuits have two main properties. The first property is that they consume lower power in comparison with the other dynamic circuits, and the second property is that their power usage is almost constant and independent from the input patterns. This second property makes the ACSL circuits desirable in terms of power prediction. This property is the base of the power prediction method that is proposed in Chapter 5.

The structure of this chapter is as follows: in Section 3.1, the background information that is needed for understanding the ACSL design concept is presented. This information includes the description of the asynchronous logic, dynamic logic, and adiabatic differential logic family. In Section 3.2, the structure, general operation and circuit design of the ACSL is described. ACSL has been developed by combining an adiabatic differential logic with charge sharing technology. A modified version of ACSL called Latch-less ACSL (LACSL) is also introduced that

provides extremly low variation in the power consumption for the applications that data independency of power is crucial for their operation.

3.1 Background

3.1.1 Asynchronous vs. Synchronous Logic

It is widely accepted that a single clock (global clock) scheme would not adjust to the nano-scaled Very Large Scale Integration (VLSI) circuits and, thus asynchronous architectures (or hybrid) emerge as potential alternatives [70]. In largely conventional systems such as modern system-on-chip designs, global clock distribution has become such a challenge that some systems have separate clocks for each processor and exchange data asynchronously between them, referred to as Globally Asynchronous Locally Synchronous (GALS) [71].

As shown in Figure 3.1 (a), the synchronous design consists of the several stages of the combinational logic which are separated by the memory blocks (registers) that transfer signals from one stage to the next. Each memory block is controlled by the clock signal (CLK) which is distributed through the clock tree. The whole system is under the control of this global clock. However, the essential clock tree results in large overhead in the area and the power consumption [72]. Other than this, the speed of the system is constrained by the the worst-case delay of the critical path.

Rather than using the global clock signal, the asynchronous circuits use a protocol called handshaking [73]. The basic structure of an asynchronous circuit is exhibited in Figure 3.1 (b). In this circuit, the flow of the data is controlled by the pipeline controllers (CTL) through the handshaking signals represented by *ack* and *req*. The controllers detect the completion of each stage, and produce
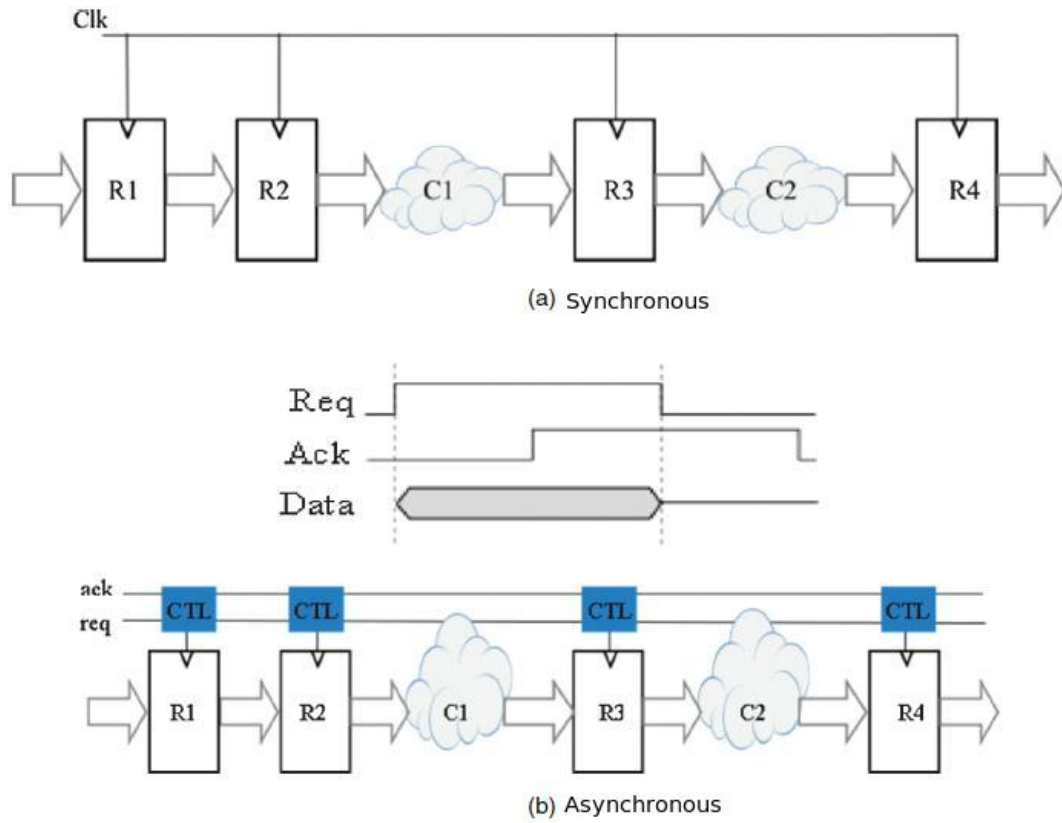
Figure 3.1: Synchronous Architecture vs. Asynchronous Architecture [69]

the *ack* signal which is used to trigger the operation of the next stage.

Unlike the conventional synchronous logic whose operation speed is determined by global worst-case latency, in asynchronous designs the speed depends on the actual local latencies. In other words, an asynchronous circuit has the potential to run at the highest possible speed. Moreover, not having to distribute a global clock leads to power savings, since the effective distribution of such a clock can cost 40% to 50% of the power in a modern digital system [74].

3.1.2 Dynamic vs. Static Logic

Most digital logic circuits are implemented using static CMOS logic gates for the combinational functions. Static gates always provide a definite output based on the current input, and update that output as soon as the input changes. As

shown in Figure 3.2 (a), the static circuits are made of NMOS and PMOS logic blocks. These blocks are dual, and for any given input only one of them creates a path between output to either the power source or the ground [75].
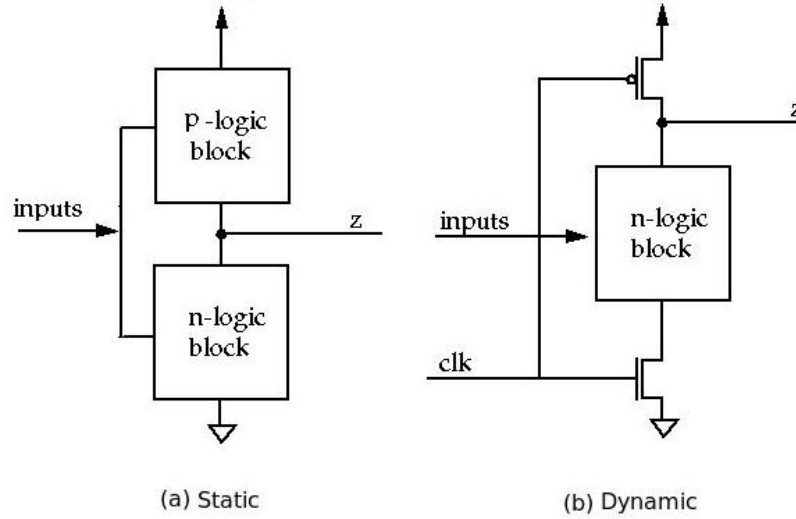
Figure 3.2: Static Logic vs. Dynamic Logic

Dynamic logic refers to the logic gates where the gate does not drive the output constantly. Instead, the output value is stored temporarily in stray and gate capacitances. As shown in Figure 3.2 (b), a simple dynamic gate has just the NMOS pull-down network in series with an evaluate NMOS transistor to ground. There is no pull-up logic function, and just a single PMOS pre-charge transistor connects the output to the power supply. The execution of these gates is governed by a clock. When the clock goes low, the circuit goes to pre-charging phase. In this phase, the pre-charge PMOS transistor pulls the output of the gate high. When the clock goes high, the gate evaluates. In the evaluation phase, if the NMOS pull-down network is satisfied, in series with the pull-down NMOS transistor pulls the output low. If it is not satisfied, the output remains high due to the gate capacitances [76].

Dynamic logic circuits are usually faster than static counterparts, and require less surface area, but are more difficult to design. Static logic is slower because it has twice the capacitive loading, higher thresholds, and uses slow PMOS transistors

for logic. Dynamic logic can be harder to work with, but it may be the only choice when increased processing speed is needed [77]. Dynamic logic has a higher toggle rate than static logic [78] but the capacitative loads being toggled are smaller [79], so the overall power consumption of dynamic logic may be higher or lower depending on various tradeoffs.

3.1.3 Adiabatic Dynamic Differential Logic

Dynamic logic gates cannot directly be cascaded. The reason is that for the correct operation of the dynamic gates the inputs need to be monotonically rising during the evaluation phase [80]. Considering the case where two gates are connected directly, and the final output should take value '1', the pre-charged '1' on the intermediate node can partially discharge the output of the second gate before the intermediate node takes its correct value.

Various designs are available to address this problem. Domino logic is one design where each dynamic gate is followed by a static inverter. However this means that all the intermediate nodes must be non-inverting, limiting the range of functions that can be implemented. Thus dynamic logic families are often differential (dual-rail), that is each signal is computed in both true and complemented form. There are several dynamic differential CMOS logic types such as Dual-rail Domino logic [81], DDCVSL (Dynamic Differential Cascode Voltage Switch Logic) [82], SABL (Sense Amplifier Based Logic) [83], etc.

Dynamic differential logic is well-known for its high speed property. However, its drawaback is inevitably high energy dissipation. Adiabatic logic [84] is a new type of low power differential logic which has drawn a lot of attention in recent years. The term, adiabatic (meaning no heat transfer), comes from the fact that an adiabatic process is one in which the total heat or energy in the system remains constant.

Adiabatic circuits recycle the energy after the evaluation through the Power Clock Generator (PCG) which usually is a LC resonant circuit [85] or a switch capacitor tank [86]. Power Clock Generator (PCG) is a replacement for DC supply (VDD) that is used in the standard CMOS circuits. The general structure of the adiabatic logic circuits is shown in Figure 3.3. The special design of the gate, and the use of the Power Clock Generator (PCG) satisfies two fundamental rules that leads to energy saving in the adiabatic logic. The first rule is never to turn on a transistor when there is a voltage difference between the drain and source. The second rule is never to turn off a transistor that has current flowing through it [84].
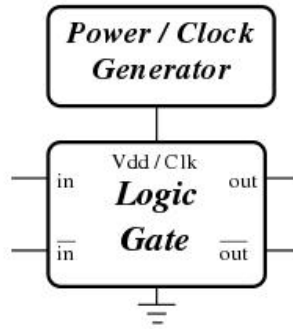
Figure 3.3: Basic Blocks of Adiabatic Logic System [85]

Figure 3.4 depicts three main styles in adiabatic logic family, which are Positive Feedback Adiabatic Logic (PFAL) [87], 2N-2N2P [88] and Efficient Charge Recovery Logic (ECRL) [89]. All three structures are charged and discharged through the Power Clock Generator (PCG).

Each PCG cycle consists of four intervals: evaluate, hold, recovery and wait. In the evaluate interval, PCG is charged up to a certain value, usually VDD, and the differential outputs are set as '1' or '0' depending on the function of the n-tree. During the hold interval, outputs are kept stable for supplying the subsequent gate with a stable input signal. In the recovery interval, PCG recycles the energy stored in the circuit by discharging itself to zero. For symmetry resons, a wait interval is also inserted [90].
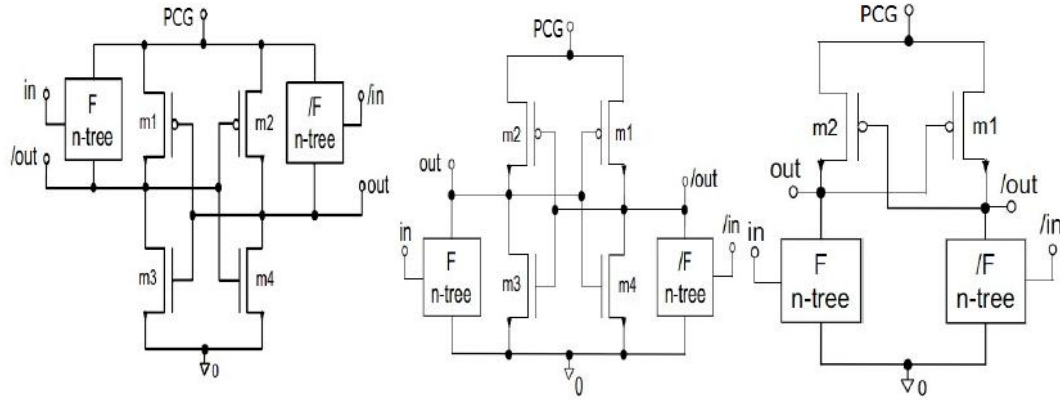
Figure 3.4: (a)General Schematic for PFAL, (b)General Schematic for 2N-2N2P, (c)General Schematic for ECRL

PFAL has the lowest power dissipation and the best consistency of voltage scaling in contrast to 2N-2N2P and ECRL [91]. However, the efficiency and performance of adiabatic circuits is restricted by PCG. Sometimes it even minimizes the savings achieved by adiabatic circuits itself. Also the area overhead due to resonant LC circuits for PCG is high.

To avoid the effort of designing power clock generator for adiabatic circuits, a new logic called Asynchronous Charge Sharing Logic (ACSL) is proposed in [69]. This new logic achieves reduced power consumption as well as low power variability. The ACSL design structure and general operation is explained in the next section.

3.2   Asynchronous Charge Sharing Logic

The main methodology of Asynchronous Charge Sharing Logic (ACSL) is to combine PFAL adiabatic logic with charge sharing technology. The main body of the PFAL is inherited by the ACSL, but the Power CLock Generator (PCG) is replaced by the charge sharing mechanism.

3.2.1   General Operation of ACSL
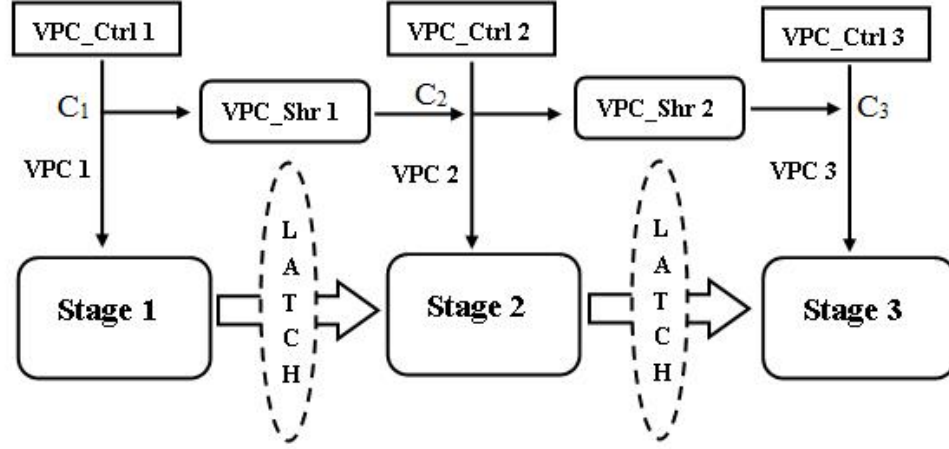
Figure 3.5: General Structure of ACSL [92]

The general architecture of the ACSL circuit is shown in Figure 3.5. The ACSL circuit consists of some stages of logic which are controlled by an asynchronous handshake. The internal structure of each stage is the same as the PFAL circuit shown in Figure 3.4 (a). The charging, discharging and sharing are performed by a power control block called *VPC_Ctrl* and a power sharing block called *VPC_Shr*. The *VPC_Ctrl* enables the evaluation and discharging of the ACSL circuit while the *VPC_Shr* is used to share the energy between two neighboring stages [69].

In Figure 3.5, C_1 is the capacitance load at $VPC1$, and C_2 is the capacitance load at $VPC2$. Before the charge sharing happens, the voltage of $VPC1$ is at VDD, and the voltage of $VPC2$ is at zero. At the end of the charge sharing process between $VPC1$ and $VPC2$, the voltage of both nodes equals to $VDD/2$ assuming that C_1 is the same as C_2. After the charge sharing, $VPC1$ is discharged to zero, and $VPC2$ is charged to full VDD. The waveforms for VPC signals are shown in Figure 3.6 [69].

The stages of the ACSL circuit shown in Figure 3.5 make an asynchronous system, and the *VPC_Ctrl* units control the flow of data between the stages. Firstly,
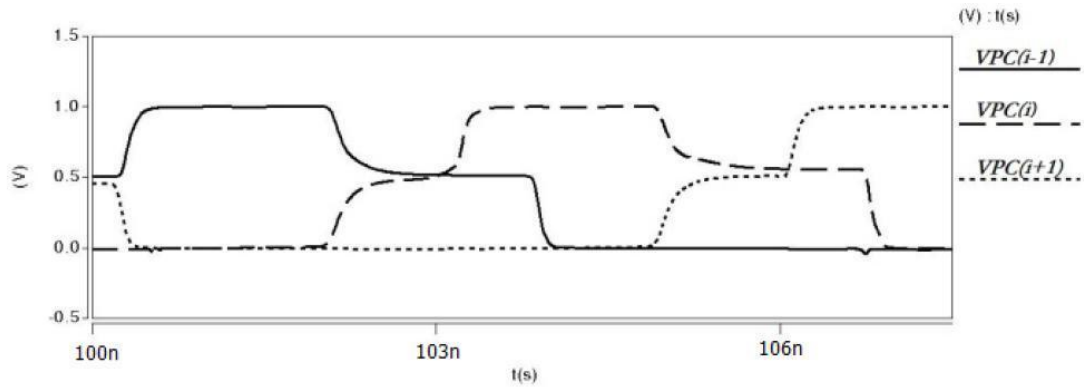
Figure 3.6: VPC Signals Waveform [92]

*VPC_Ctrl*1 evaluates the logic block of stage 1 by charging the *VPC*1 to *VDD*. The output of the stage 1 is latched to be available as the input to the stage 2. Then the *VPC_Shr*1 placed between stage 1 and stage 2 is switched on to start the charge sharing process. Once *VPC1* and *VPC2* reach almost the same level, nearly *VDD*/2, a Sharing Detector (*SD*) unit turns off the *VPC_Shr*1. *VPC_Ctrl*2 is then activated to charge *VPC*2 from *VDD*/2 to full *VDD*. Meanwhile, *VPC*1 is discharged to zero by *VPC_Ctrl*1. This process repeats for the following stages [69].

3.2.2   ACSL Circuit Design

Figure 3.7 shows the detailed architecture of a two stage ACSL circuit. The evaluation and charge sharing in this circuit is controlled by an asynchronous handshake. This handshake is based on the stage power clock, *VPC(i)* and three other signals, *Ctrl(i)*, *Req(i)* and *SD(i)*. *Ctrl(i)* puts the gate logic in the evaluation mode, and *Req(i)* indicates the completion of the stage. *SD(i)* triggers the charge sharing between the stages.

A dynamic AND is used to generate *Ctrl(i)*. Using the dynamic AND leads to *Ctrl(i)* switching to low voltage immediately once *Ctrl(i+1)_n* becomes low. In this way, activation of the next stage leads to the deactivation of the current stage.
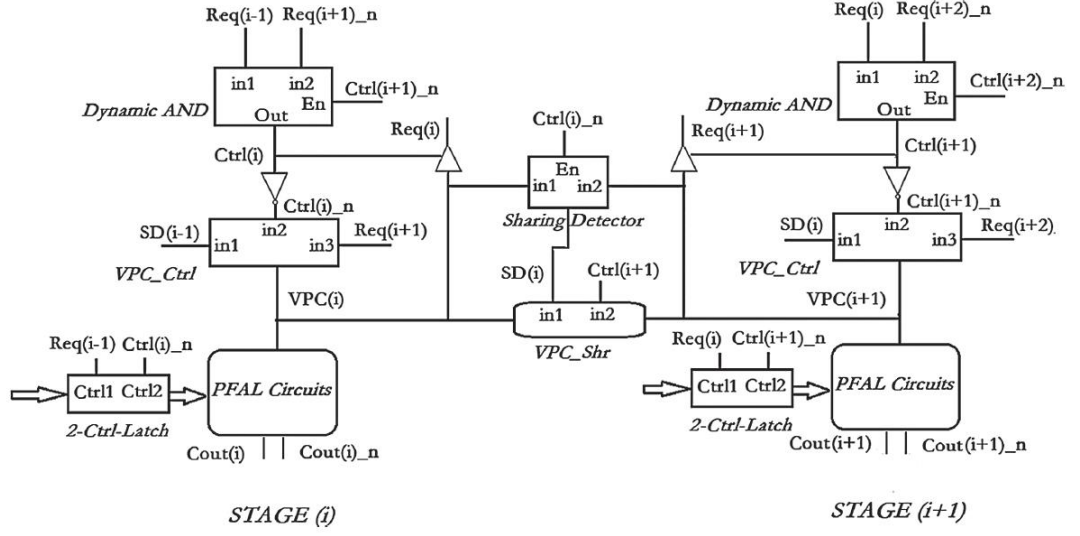
Figure 3.7: 2-Stage Architecture of The ACSL circuit [69]

A dynamic buffer which is controlled by the *Ctrl(i)* signal is used to accomplish the completion detection. This buffer senses the *VPC* signal, and generates the *Req(i)* signal. The completion of each stage triggers the activation of the next stage. The signal transition diagram of the ACSL handshaking protocol is exhibited in Figure 3.8. As seen in the figure, the control signal of an indivisual stage is only valid when the control signal of two adjacent stages are low.

When the *Ctrl(i)* becomes low, the Sharing Detector (SD) gets active, and starts the charge sharing between the stages. When both *VPC*s are higher than the threshold voltage of the NMOS transistor, the signal *SD* becomes low. It indicates that sharing operation can be stopped by switching off *VPC_Shr*.

The two-controlled latch placed between the stages is crucial for the correct op-
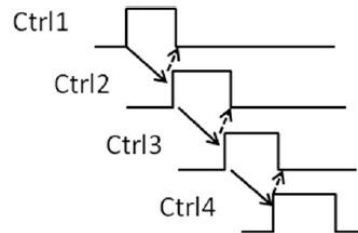
Figure 3.8: ACSL Handshaking Protocol [69]
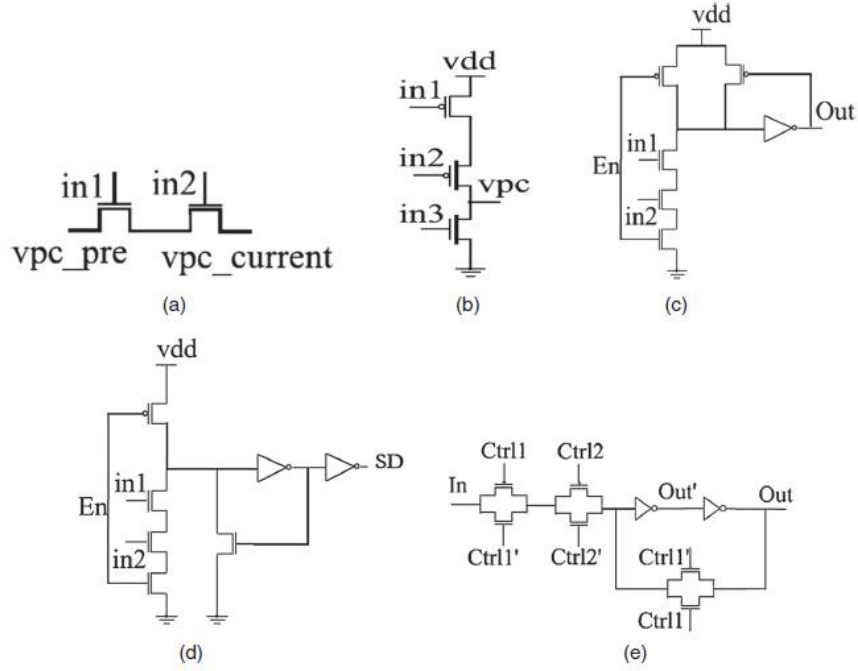
Figure 3.9: Schematic for (a) VPC_Shr, (b) VPC_Ctrl, (c) Dynamic AND, (d) Sharing Detector, and (e) Two-controlled Latch [69]

eration of the ACSL. It assures that all the data from previous stage is loaded before the sharing happens. The latch is accessed only when the signal *Req(i)* is high and the signal *Ctrl(i+1)* is low, and it enters into hold mode as soon as *Ctrl(i+1)* becomes high. In this way, the output is stored when the evaluation of the current stage is complete, and stays unchanged when the next stage is in the evaluation mode. The circuits of *VPC_Shr*, *VPC_Ctrl*, Dynamic AND, Sharing Detector and two-controlled latch are all shown in Figure 3.9.

3.2.3   Latch-Less ACSL

Asynchronous Charge Sharing Logic (ACSL) has good power constancy, due to the symmetry of the gates and the fact that the gates are completely discharged in between executions. However, the latches which are necessary for the complete discharge of the gates consume different amounts of power depending on whether they are rewritten with the same or opposite value. In some applications like
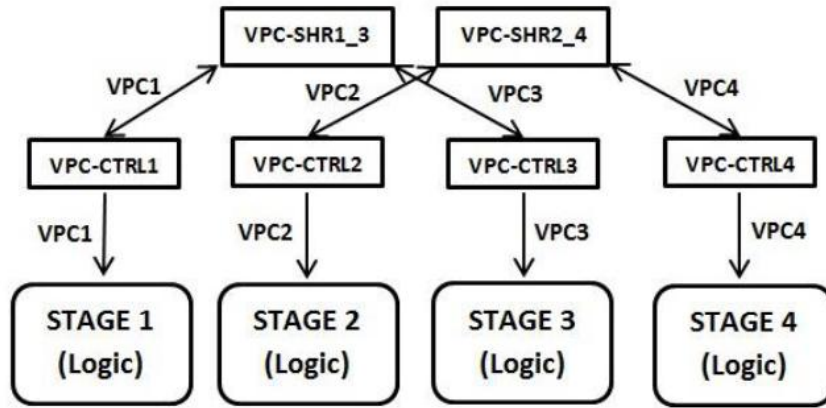
Figure 3.10: Block Diagram of Latch-Less ACSL [93]

cryptography, it is very important for the system to have extremely low power variation to avoid power attacks [93]. In such cases, it is desirable to design the circuit without the latches, since they are the only ACSL component with intrinsic data dependence.

It is not practical to exclude the storage elements directly from ACSL, doing so reduces the stability of the gates. without data retention devices, once the charge sharing finishes, the input data for the next stage might not be valid (only at half *VDD*), the consequent unequal voltage distribution not only slowing down the evaluation speed but also leading to possible error occurrences like the write error in SRAM cells [93].

To tackle this problem, interleaved charge sharing is considered, because it could intuitively solve the voltage unbalance situation discussed above. The block diagram of the Latch-less ACSL (LACSL) is shown in Figure 3.10. It can be seen that charge sharing occures between the Voltage Power Clocks (*VPCi*, i=1..4) which are one stage apart instead of the adjacent stages. By doing this, input data for each stage is always fully charged during the computation. Moreover, it is found to be generally efficient in terms of performance, power consumption and area. The main adjustment needed in LACSL is to re-design the *VPC_Ctrl* circuit to maintain the power-up situation across the stages. LACSL offers extremely

low power variation, and thus very high predictability. In [93], a Montgomery modular multiplier is developed using LACSL for cryptography applications.

3.2.4   Summary

The Asynchronous Charge Sharing Logic (ACSL) is a dynamic design style that combines PFAL adiabatic logic family with charge sharing technology. The ACSL has low power usage, and exhibits strong power predictability that is a useful feature for high level power analysis.

In this chapter, the background information for understanding the ACSL design concept is presented, and the structure and operation of the ACSL circuits are explained. A variant of ACSL is also introducd that offers even more constant power usage by excluding the latches from the ACSL structure.

In next chapter, the implementation of the Arithmetic Logic Unit (ALU) of 8051 microcontroller in ACSL is described. A power prediction method for this ALU is proposed in Chapter 5. The prediction method is based on the data independency of power consumption which is one of the main properties of ACSL circuits.

Chapter 4

Implementation of 8051 Arithmetic Logic Unit (ALU) in ACSL

In this chapter, the implementation of the 8051 Arithmetic Logic Unit (ALU) in Asynchronous Charge Sharing Logic (ACSL) is described. The aim of this implementation is using the property of ACSL in providing power predictibility for the circuits. The ALU operations in ACSL show almost constant power usage independent from the inputs. This property makes it possible to estimate the power usage for the ALU only by knowing the number and type of the operations it performs.

The 8051 [94] is an 8-bit Complex Instruction Set Computer (CISC) design, with an instruction set optimised for manually developed assembly code. Unlike many 8-bit microcontroller architectures, the 8051 has a generic architecture available commercially from many manufacturers [95, 96] and as an open-source soft core [97]. It is a common microcontroller in embedded systems applications due to being widely available, and having many variants with different peripherals. For

39

this reason, the ALU of 8051 has been chosen in this work for the purpose of power prediction.

The 8051 ALU operations are implemented individually through ACSL design flow, and then integrated in the final structure of the ALU. The functionality of the ACSL ALU is verified through simulation, and the analysis of the power, delay and area of the circuit is performed.

The structure of this chapter is as follows: in Section 4.1, the design flow used for implementing the ALU in ACSL is described. In Section 4.2, the 8051 ALU operations are introduced, and in Section 4.3 the implementation of each of these operations in ACSL is explained. In Section 4.4, the structure of the ALU and the integration of the indivisual operations in the final design is described. In Section 4.5, the approach for verifying the functionality of the ALU design is discussed. Finally in Section 4.6 the ACSL ALU design and performance characteristics has been analysed, and the simulation results for power, delay and area of the design are presented.

4.1   ACSL Design Flow

In order to implement a specific digital unit in ACSL, a systematic approach needs to be followed. In the absence of such approach, the ACSL design has to be done by either analog IC schematic capture tools or directly using SPICE netlists. In both of these approaches, hand editing of the netlists is necessory. Thus, it is difficult and time consuming to use these methods for the large circuits.

The approach followed in this work is based on using a structural style of Verilog hardware description language in which the circuit is described entirely using module instances of sub-modules. A tool reads this Verilog description, and translates Verilog modules to SPICE sub-circuits, and Verilog module instances therein

Table 4.1: ACSL Gates and Primitive Modules

Gates	acsl_addsub_cell	acsl_buffer	acsl_inv
	acsl_xor2	acsl_and2	acsl_half_adder
	acsl_full_adder	acsl_mux2	acsl_or2
	acsl_or3	acsl_mux_2to1	acsl_muxoh16
Pirimitive Modules	sram_latch	dynamic_and	vpc_ctrl
	completion_detector	vpc_shr	sharing_detector
	inv	c_element	nand2
	and2	or2	

to SPICE sub-circuit instances. SPICE implementations of the basic modules are provided to the tool as a library to replace the behavioural descriptions.

Verilog is the standard language in many modern VLSI design flows. The main advantage of using Verilog is that it can be simulated at high level very quickly, allowing for easy logical verification of the circuit. Verilog is a high level language, with a much more readable syntax than SPICE. Verilog has detailed warnings, making mistakes in schematic capture, such as disconnected wires much more obvious and quicker to find than waiting for SPICE convergence to fail, and having to manually diagnose the problem.

The Verilog description of the ACSL circuit uses the basic ACSL gates and primitive modules as building blocks to describe any particular circuit. These gates and modules are listed in Table 4.1. The basic blocks in the structure of the ACSL circuits are also shown in Figures 3.7 and 3.9 in the previous chapter. The ACSL gates are used in each stage of the circuit to generate the outputs for the next stage. The primitive modules are used to implement the handshaking and charge sharing logic between the stages.

Handshaking and charge sharing for asynchronous stages are implemented in a modular way. That is a Verilog module is constructed corresponding to one stage of the handshake and an array of these is implemented to support all the stages.

The Synopsys VCS [1] simulator is used to verify the operation of the circuit that

---

[1]Version C-2009.06

is described in Verilog. Behavioural models of ACSL gates and primitive modules are coded to be used in the simulation at this stage.

Figure 4.1 shows the ACSL design flow. The entry of the flow is the Verilog description of the ACSL circuit. Synopsys V2S [2] translates this Verilog structural description to SPICE. V2S is a generic Verilog-to-SPICE translator. It is designed to be used on structural Verilog, translating Verilog modules to SPICE subcircuits and Verilog module instances therein to SPICE subcircuit instances. A custom translator generates the SPICE netlists for the ACSL gates which is given to V2S to replace the behavioral descriptions. SPICE implementations of primitive modules are also available to V2S as a library.
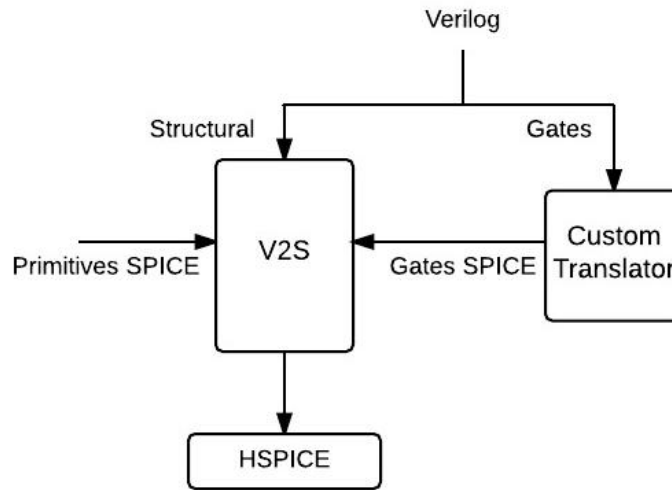
Figure 4.1: ACSL Design Flow

The custome translator is a simple compiler written in Python. It converts the Verilog gate definitions to SPICE netlists, with a generic gate core (cross-coupled inverters) and NMOS pull-up networks corresponding to the given logic functions.

The SPICE implementation resulted from this design flow can be simulated at transistor level using Synopsys HSPICE tool to analyse the timing and power of the design.

---

[2]Version 2009-7.0

4.2   8051 ALU Operations

The Intel MCS-51 (commonly referred to as 8051) is a Harvard architecture, CISC instruction set, single chip microcontroller (μC) series which was developed by Intel in 1980 for use in embedded systems [98]. The 8051 is classified as an 8-bit processor, because the internal registers and the internal data bus are 8-bit wide. However, in addition to the operations on 8-bit (byte) data, the ALU of 8051 can perform operations on bit, nybble (4-bit), and double-byte (16-bit) in a limited way [99].

The ALU of 8051 microcontroller can perform a set of arithmetic, logical and shift operations. These operations are shown in Table 4.2.

Arithmetic operations are add, subtract, multiply and divide. Since the numbers are in 2's complement, the add and subtract operations are suitable for both signed and unsigned operands. As well as 8-bit additions, 16-bit plus 8-bit additions are also supported for address calculations. The multiply and divide operations are unsigned. Having multiply and especially divide instructions contrasts to many other microcontrollers which require the user to implement both or just divide in software, or provide separate hardware outside the ALU or processor core for these operations.

Three flags are set based on arithmetic function: carry (CY), auxiliary carry (AC) and overflow (OV). The auxiliary carry is used as a half carry, being the carry out associated with the 4th bit of the result. This is used in Binary-Coded

Table 4.2: 8051 ALU Operations

Arithmetic Operations	Logical Operations	Shift Operations	Other Operations
Add	NOT	RL	XCH(Exchange)
Subtract	AND	RLC	DA(Decimal Adjust)
Multiply	OR	RR	NOP(No Operation)
Divide	XOR	RRC	

Decimal (BCD) arithmetic operations. The overflow flag is set when the result has overflowed in a signed manner (unsigned overflow is the same as carry).

Logic and shift operations are also provided. Logical operations are AND, OR, XOR and NOT. Shift operations are in the form of rotations (cyclical), allowing rotating left or right (RL or RR) of the 8-bit input operand by itself or through both the operand and the carry flag (RLC or RRC). An exchange operation is also available to swap the low order nybbles of two 8-bit inputs.

Additionally a Decimal Adjust operation is provided to allow for BCD additions. This assumes a packed BCD format. In this format, each nybble of a byte represents one BCD digit. Decimal Adjust is designed to be used after a binary add instruction, and corrects each nybble for BCD operation based on half carry and carry flag.

NOP (No Operation) is provided to be used when no other operation is needed to perform. This means that when NOP is selected, ALU passes the input operands and flags to the outputs without any change.

4.3 Implementation of the ALU Operations in ACSL

The ACSL design flow described in Section 4.1 has been used to implement the 8051 ALU operations. Unlike many 8-bit microcontroller architectures, the 8051 architecture is a generic architecture available commercially from many manufacturers. The ALU operations in this work are in reference to programming manuals, both the original from Intel [94] and the manuals from clone manufacturers [95, 96]. The interfaces of the ALU for this work resemble the ALU of the *opencores.org* 8051 [97], an open-source soft-core 8051. In this ALU, there are 3

input bytes and 2 output bytes, as well as flags.

4.3.1 Addition and Subtraction Operations

The 8051 ADD, ADDC and SUBB instructions are used for addition and subtraction operations. The ADD instruction adds a byte value to the accumulator and stores the results back in the accumulator. The ADDC instruction adds a byte value and the value of the carry flag to the accumulator. The SUBB instruction subtracts the specified byte variable and the carry flag from the accumulator. The SUBB instruction sets the carry flag if a borrow is required for bit 7 of the result. If no borrow is required, the carry flag is cleared [96].

In addition to these 3 instructions, 8051 also has some other instructions which need to use ALU for addition or subtraction as a part of their execution. For example, an addition is required for calculating the destination address for the jump (JMP) instruction. To satisfy the requirements of all of these instructions, the 8051 ALU needs to support 16-bit plus 8-bit addition and subtraction.

A ripple carry adder circuit is used to implement addition and subtraction operations as shown in Figure 4.2. This circuit takes a 16-bit number (A), an 8-bit number (B) and carry-in (Ci) signal as the inputs, and generates a 16-bit number (F), carry-out (Co), auxiliary carry (AC), and overflow (Ov) signals as the outputs. The input signal (Sub) determines wheter addition or subtraction needs to take place. Depending on which operation is selected, either (A+B+Ci) or (A-B-Ci) are generated as the result on the output (F), and the flags are set accordingly.

The main 8-bit result is computed using 8 XOR gates and full adders as usual for 2's complement. The 8051 carry flag is defined as always active high. The half carry signal is generated from the middle carry of the array. The overflow
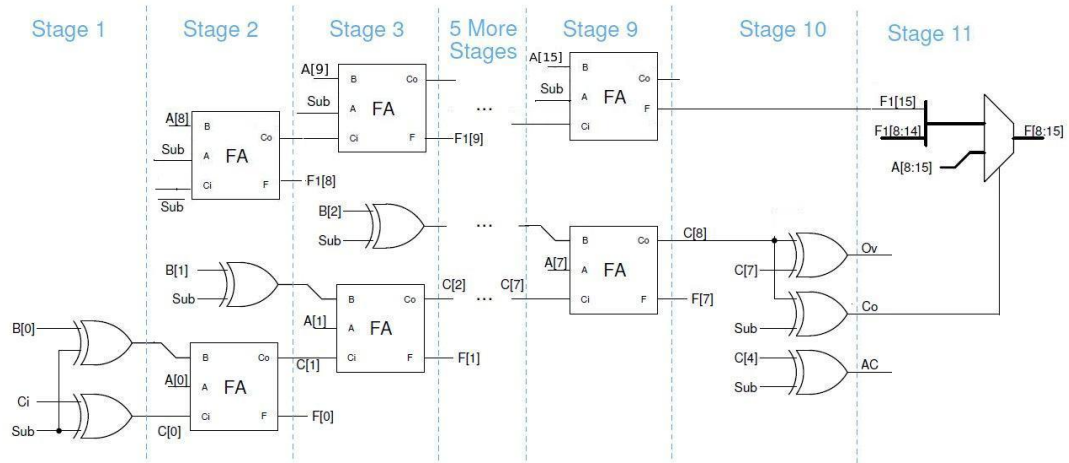
Figure 4.2: ACSL 8051 Add/Subtract Circuit

flag, being defined as overflow of a 2's complement signed number, is calculated as the XOR of the 7th and final carry signals.

For the 16-bit plus 8-bit add and subtract, 8 full-adders are used in parallel with the initial array to do the required operation on the high order 8 bits. In case of addition these full adders compute the value of the A[15:8]+1, and in case of subtraction, they compute the value of the A[15:8]-1. The final result is selected using a multiplexer based on (Co) flag which is the carry or borrow of the add or subtract operation on the lower 8 bits. The add-subtract circuit thus totals 11 stages of asynchronous logic: 1 set-up, 8 computation, 1 for flags generation, and 1 for high multiplexer.

4.3.2 Multiplication Operation

The 8051 MUL instruction multiplies the unsigned 8-bit integer in the accumulator and the unsigned 8-bit integer in the B register producing a 16-bit product. The low-order byte of the product is returned in the accumulator. The high-order byte of the product is returned in the B register. The overflow (OV) flag is set if the product is greater than 255 (0FFh), otherwise it is cleared [96].
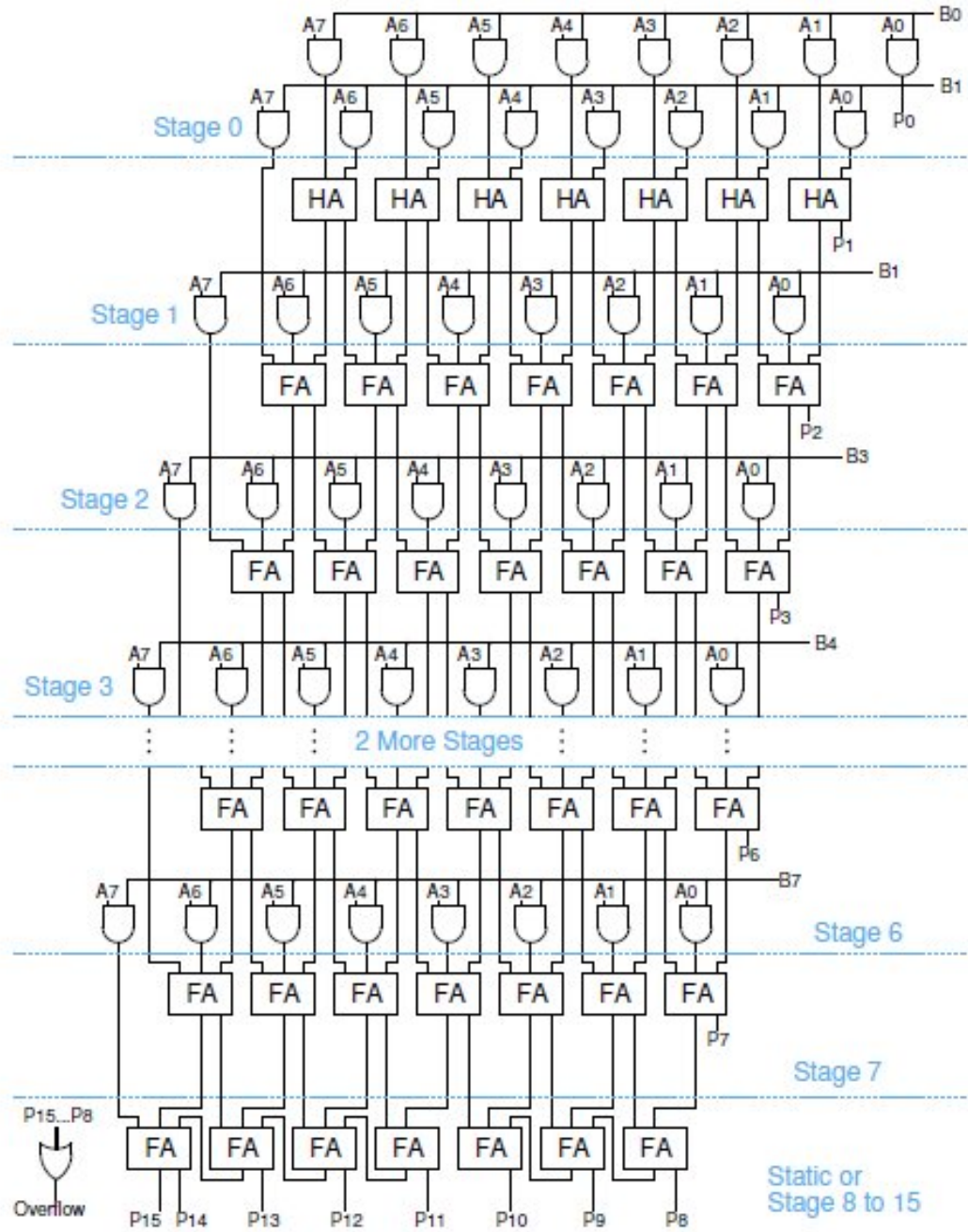
Figure 4.3: ACSL 8051 Multiplier Circuit

Figure 4.3 shows the implementation of multiplication operation in ACSL. The

circuit takes two 8-bit numbers (A,B) as the input, and generates a 16-bit output

(P), and the overflow flag.

An array multiplier [100] is used to implement multiplication. The array mul-

tiplier was chosen since its rectangular array shape allows effective balanced scheduling of ACSL stages, as required for smooth and effective sharing.

A 7-bit by 8 layer rectangular array is used. That is, there are 8 sets of 8 AND gates generating partial products, and 8 sets of 7 full adders combining these partial results (each adder being a generalised 3 to 2 reduction, as opposed to a ripple carry arrangement). These are implemented as 1 stages of initial partial product generation, followed by 8 stages of adder array with further partial product generation in each stage.

For the first version of multiplier the final adder is implemented using a static ripple carry adder. The intention was to avoid having a large number of single-gate stages. However when tested this is found to result in very variable performance (33.5% variablity in delay and 9.8% variability in power). Thus a second version of the multiplier is implemented, using an ACSL ripple carry adder. The first multiplier uses 9 stages of asynchronous logic plus the static final adder. There is 1 stage of initial partial product generation and 8 stages of array. The second version adds 7 stages of final adder to make 16 total stages.

4.3.3 Division Operation

The 8051 DIV instruction divides the unsigned 8-bit integer in the accumulator by the unsigned 8-bit integer in register B. After the division, the quotient is stored in the accumulator and the remainder is stored in the B register. If the B register begins with a value of 00h the division operation is undefined, the values of the accumulator and B register are undefined after the division, and the overflow (OV) flag will be set indicating a division-by-zero error [96].

Figure 4.4 shows the Implementation of division operation in ACSL. The circuit takes two 8-bit inputs (y, div), and generates two 8-bit outputs (q, rem), and the
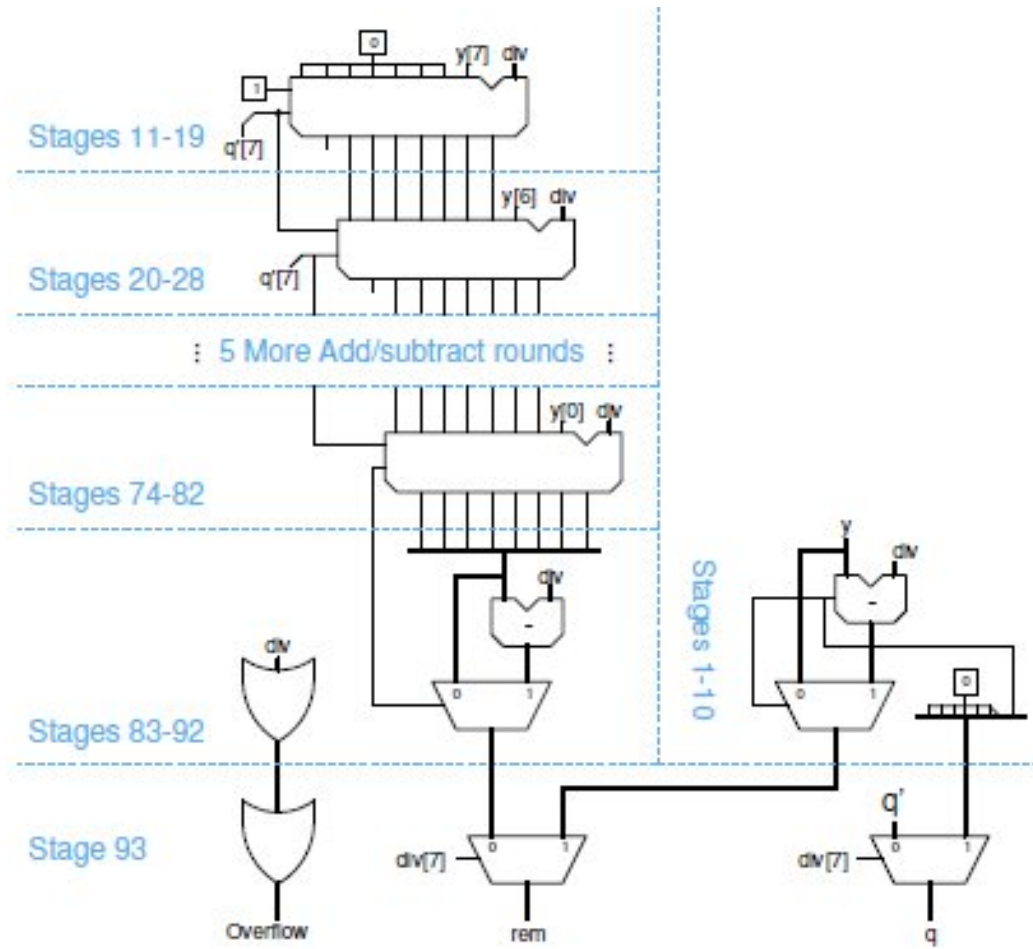
Figure 4.4: ACSL 8051 Division Circuit

overflow flag.

Division is achieved using a non-restoring array divider [101]. This divider is similar to the standard repeated-subtraction algorithm, but avoids using multiplexers at each stage for restoration by selecting addition for the following stage instead of subtraction when overflow occurs. The main divider array consists of 9 by 8-bit add/subtract units, 8 producing 1 bit each of the 8 quotient bits and 1 to correct the remainder.

The add/subtract units are ripple carry adders the same as the core of the circuit in Figure 4.2. Each add/subtract unit has 1 stage of set-up and 8 stages of XOR gates and full adders. A stage of ACSL multiplexers is then used to select between

the corrected and uncorrected remainder.

The basic non-restoring divider array performs signed division. To compute the unsigned division required for an 8051, an additional side logic path is provided to handle divisors with the most significant bit set. A single subtraction and restore is performed. A final multiplexer is used to select between the main result and the big divisor result.

The whole circuit consists of 94 stages of asynchronous logic. The logic for handling large stages consists of 11 stages: 1 stage of buffer, 1 stage of multiplexer and 9 stages for a subtracter. The main division array consists of 72 stages, and the correction adder and multiplexer totals 10 stages. The final multiplexer for selecting between the big divisor and regular results is the last stage.

4.3.4   Logic Operations

The 8051 ANL, ORL and XRL instructions perform bitwise logical AND, OR and XOR operations on the two byte operands, leaving the result in the first. The CPL instruction performs logical NOT operation on the specified operand [96].

In addition to the byte-level operations, the ANL, ORL and CPL instructions can also be used for bit-level operations. Bit-level manipulations are very convenient when it is necessary to set or reset a particular bit in internal RAM or Special Function Registers (SFRs). A part of internal RAM and some SFRs are bit addressable. When ANL or ORL are used in bit-level, one of their operands is carry flag (CY) and the other operand is an addressable bit. The result of the AND or OR operations on carry flag (CY) and the source bit is written back in the carry flag (CY). The 8051 also provides the option that the inverted value of the source bit is used in ANL or ORL instructions. The CPL when used in bit
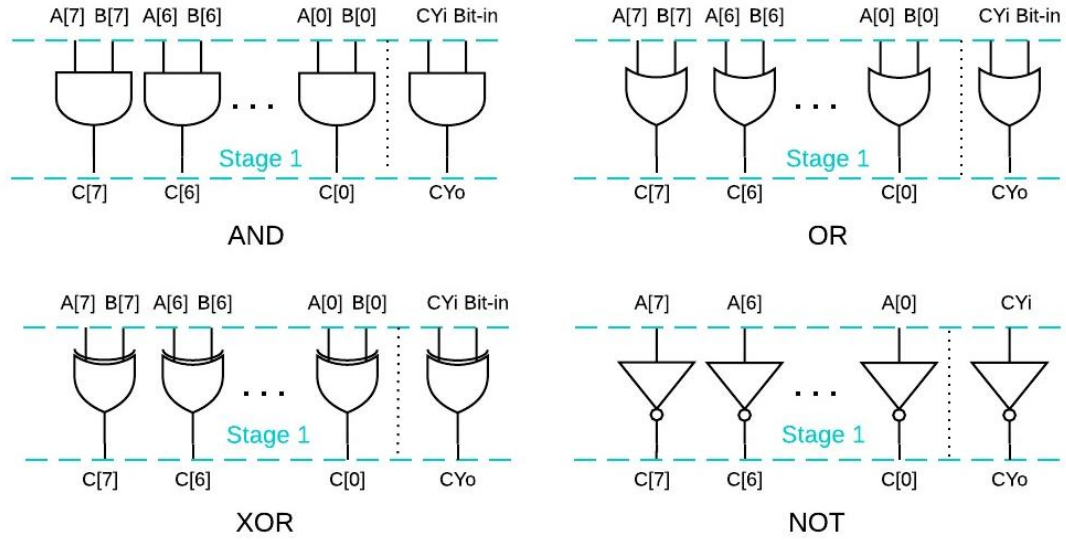
Figure 4.5: ACSL 8051 Logic Operations Circuits

level can complement the carry flag (CY) or any directly addresable bit [102].

As shown in the Figure 4.5, logic operations are implemented using a single stage of the relevant ACSL gates in parallel. To provide support for bit-level operations, an extra gate is used to operate on carry flag (CY) and an input bit. For the cases that inverted value of the input bit needs to be used in AND or OR operations, the required circuit is provided as a part of the implementation for the shift operations. This will be discussed in the next section. The reason for this approach is to maintain the compatibilty with the *opencores* 8051 design [97].

4.3.5 Shift Operations

The 8051 RR, RL, RRC and RLC instructions perform right or left rotation (cyclical shift) on the accumulator register by itself or through both the accumulator and the carry flag [96].

As shown in the Figure 4.6, shift operations are implemented using a single stage
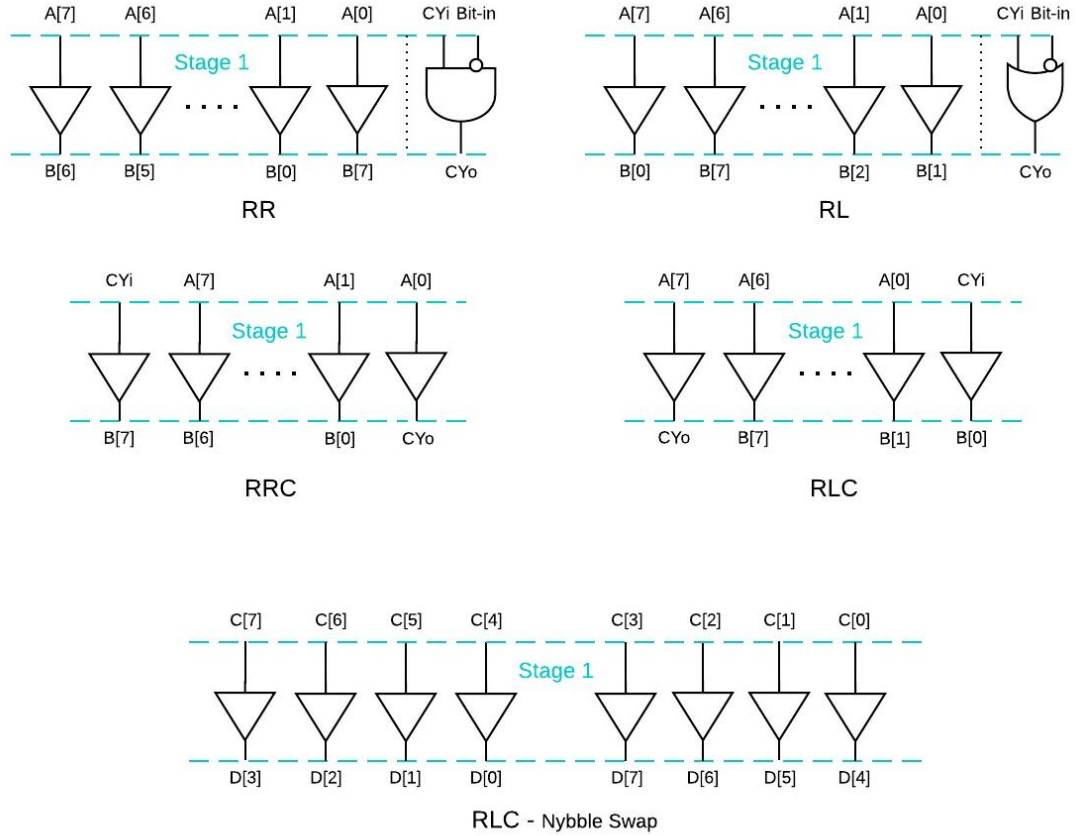
Figure 4.6: ACSL 8051 Shift Operations Circuits

of ACSL buffers, wired appropriately. The rotation is applied to an 8-bit input (A), and the result is placed on an 8-bit output (B).

For rotate right (RR), the least significant bit of A transfers to the most significant bit of B. Similarly for rotate left (RL) the most significant bit of A transfers to the least significant bit of B. For rotate right with carry (RRC), the least significant bit of A transfers to carry flag, and the previous value of carry flag transfers to the most significant bit of B. This also happens to rotate left with carry (RLC) but in the other direction.

As mentioned in the previous section, the required logic for AND and OR operations between carry flag and inverted value of an input bit is provided along with the RR and RL circuits as shown in the Figure 4.6.

In the *opencores* 8051 design [97], the RLC operation also performs a nybble swap on the second 8-bit input of the ALU and transfers the result to the second 8-bit output. To keep the consistency with this design, ACSL circuit for the nybble swap is also provided for the RLC operation. This operation is used by 8051 SWAP instruction which exchanges the low-order and high-order nybbles within the accumulator [96].

4.3.6 Exchange Operation

The 8051 XCH instruction loads the accumulator with the byte value of the specified operand while simultaneously storing the previous contents of the accumulator in the specified operand. The XCHD instruction exchanges the low-order nybble of the accumulator with the low-order nybble of the specified internal RAM location [96].

The exchange operation of the ALU provides support for the execution of XCH and XCHD instructions. Figure 4.7 shows the circuit for implementation of the exchange operation. This operation is implemented using a single stage of two ACSL multiplexers in parallel.
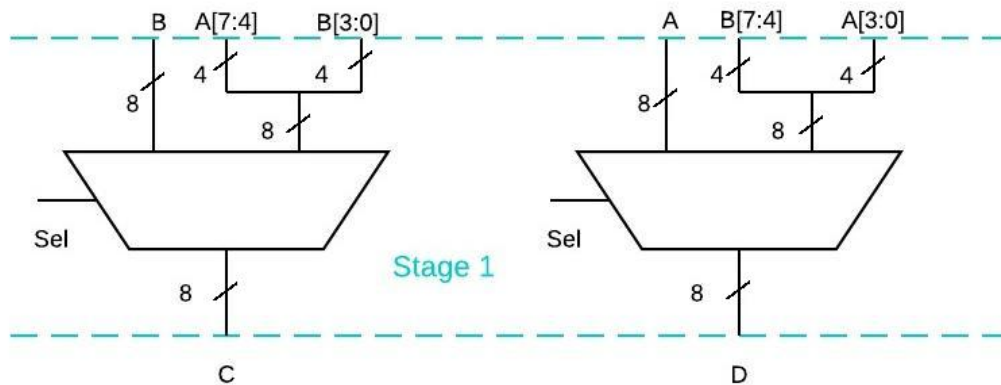
Figure 4.7: ACSL 8051 Exchange Operation Circuit

The circuit has two 8-bit inputs (A, B), and two 8-bit outputs (C, D). Based on the value of a select input (Sel), either the 8 bits or only the low order 4 bits of the inputs are swapped, and transfered to the outputs.

4.3.7 Decimal Adjust Operation

The 8051 DA instruction adjusts the eight-bit value in the accumulator resulting from the earlier addition of two variables (each in packed-BCD format), producing two four-bit digits. Any ADD or ADDC instruction may have been used to perform the addition [96].

If accumulator bits 3-0 are greater than nine, or if the auxiliary carry flag (AC) is one, six is added to the accumulator, producing the proper BCD digit in the low-order nybble. This internal addition would set the carry flag if a carry-out of the low-order four-bit field propagated through all high-order bits, but it would not clear the carry flag otherwise [96].

If the carry flag is now set, or if the four high-order bits now exceed nine, these high-order bits are incremented by six, producing the proper BCD digit in the high-order nybble. Again, this would set the carry flag if there was a carry-out of the high-order bits, but would not clear the carry. The carry flag thus indicates if the sum of the original two BCD variables is greater than 100, allowing multiple precision decimal addition [96].

Essentially, this instruction performs the decimal conversion by adding 00H, 06H, 60H, or 66H to the accumulator, depending on initial accumulator and the flags values [96].

Figure 4.8 shows the circuit which implements decimal adjust operation in ACSL. The input of this circuit is an 8-bit number (A), carry flag (CYi) and auxiliary flag (AC), and the output is an 8-bit number (DA), and the new value generated
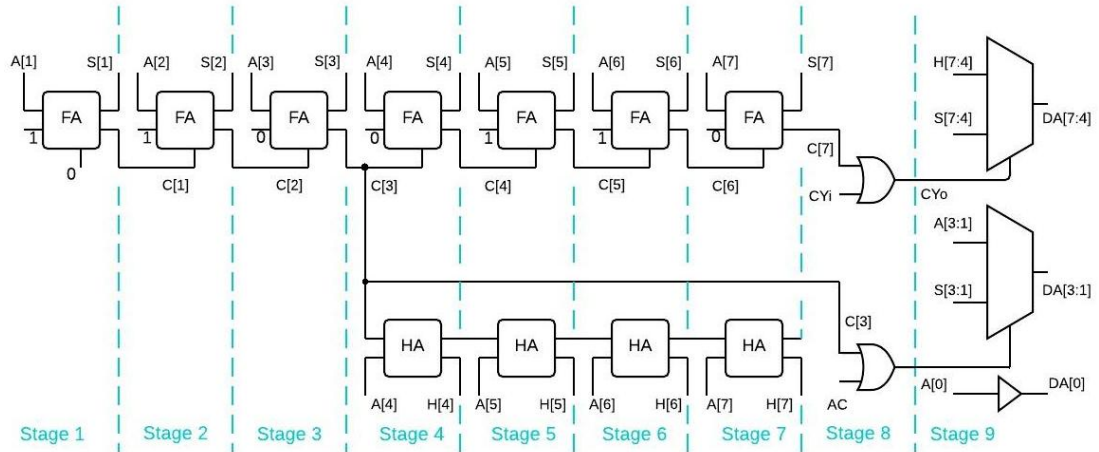
Figure 4.8: ACSL 8051 Decima Adjust Operation Circuit

for the carry flag (CYo). Decimal adjustment is applied on (A) based on the flags values.

For correcting the bottom nybble of the input, a 3-bit adder is used to add '011' to A[3:1]. Full adders in stages 1 to 3 implement this 3-bit adder. For the top nybble of (A) two 4-bit adders are used to generate the possible corrections. The first one consists of 4 full adders that add '0110' and carry of the bottom nybble to the top nybble (A[7:4]), and the second one consists of 4 half adders that only propagate carry of the bottom nybble to the top nyyble (A[7:4]). These two adders work in parallel in stages 4 to 7 of the ACSL circuit. In the last stage multiplexers are used to choose the correct output for each nybble based on the flags values and carry-out signals of the adders. The least significant bit of the input transfers to the output without change. Decimal adjust operation totals 9 stages of asynchronous logic.

4.3.8   No Operation

For the 8051 instructions that do not need to use ALU operations during their execution, No Operation (NOP) is selected by the ALU operation selecting signal.

As the result, the ALU inputs are transferred to the outputs without any change.

To implement No Operation (NOP) in ACSL, a single stage of buffers are used that transfer two 8-bit inputs and 3 input flags to the outputs.

4.4   The ALU Structure

The operations introduced in the previous section are integrated into the ALU structure as shown in Figure 4.9. The ALU has a central part consisting of the operations implemented in ACSL. The circuit for each operation has a request input (req) and an acknowledgement output (ack). Each operation starts by a short pulse on the req signal, and activates the ack signal when it is finished.

Based on the value of the Opcode input, one of the operations is selected for the execution, and the others stay inactive. The inputs of the inactive operations are set to zero by the input multiplexers to avoid unwanted switching activity in the non-operative mode. A 4-to-16 decoder is used to generate the request signals for the operations based on the Opcode value. A multiplexer is used to transfer the output of the selected operation to the output of the ALU. An OR gate generates the Ack output of the ALU based on the ack outputs of all the operations.

The Add/Sub module provides support for 3 of the 8051 ALU operations. These 3 operations are ADD, SUBB and INC. An OR gate is used to generate the request for Add/Sub module when any of these operations are selected by the Opcode. Based on the *opencores* [97] implementation of 8051 ALU, the INC operation is used for 16-bit increment or decrement depending on the value of the ALU Carry-in flag (CYi). If this flag is high, this operation performs decrement, and if it is low, increment takes place.

Since this ALU is implemented to be used inside a clock-based design of the 8051 microcontroller, it needs to start operating on each rising edge of the clock signal.
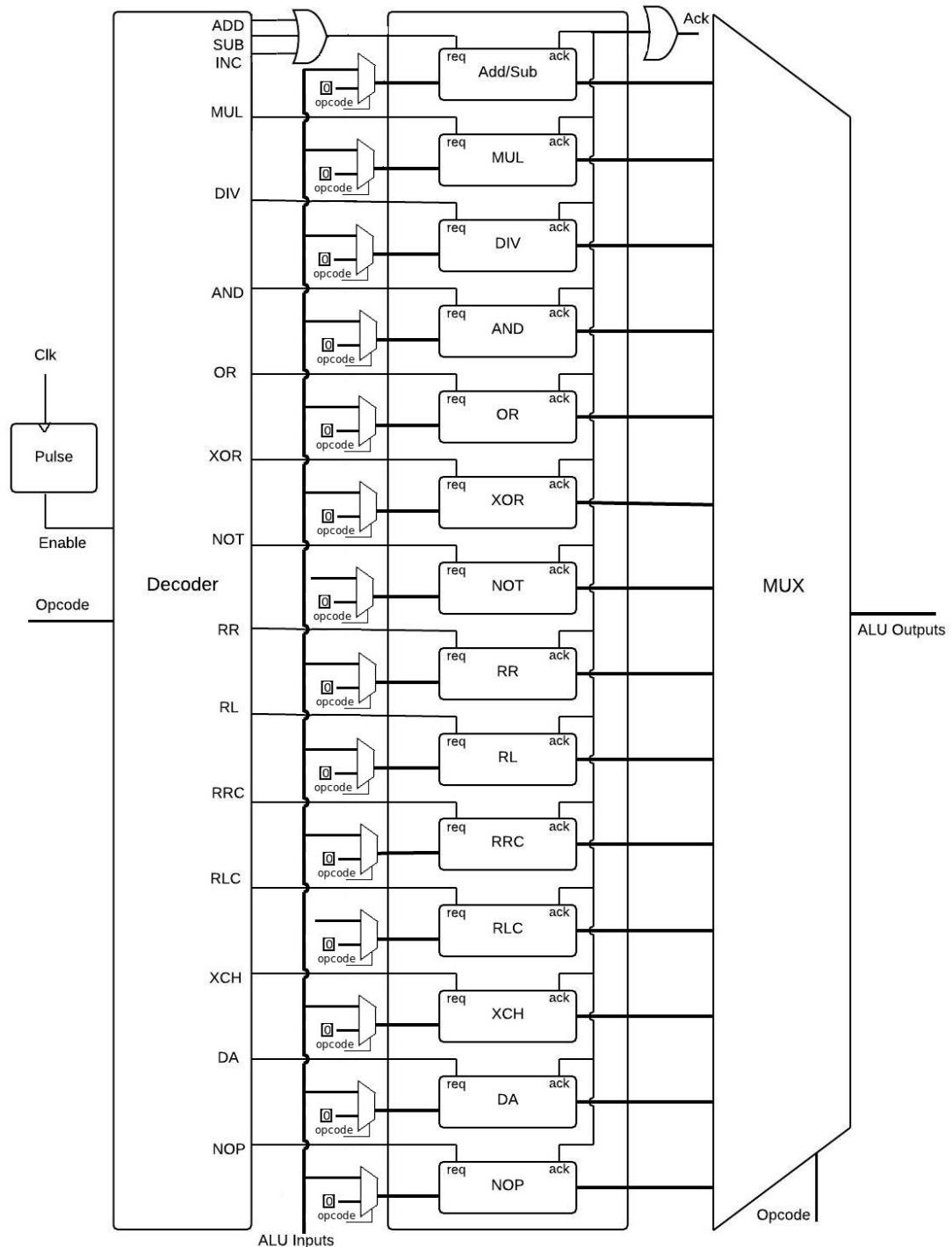
Figure 4.9: The ALU Structure

For this reason, a pulse module is used to enable the Decoder only for a short time after the clock edge. In this way, the Decoder generates a short pulse on the request signal of the operation that is selected by the Opcode. The selected operation needs to finish, and produce the outputs before the next rising edge of the clock. This can take up to 4 clock cycles for multiplication and division operations in order to be compatible with *opencores* 8051 design [97].

A beneficial point of this design is that at any moment only one of the operations is active, and this prevents the other operations to consume power. This is an advantage over the conventional design of the ALU in which all the operations start working in parallel when the input signals change.

4.5   Functional Verification of the ACSL ALU

The verification method used to verify the functionality of the ALU implemented in ACSL is shown in Figure 4.10. The reference model for the correct functionality is the *opencores* 8051 ALU [97] which is designed in Verilog hardware description language in RTL level.

As the figure shows, the test vectors are applied to the inputs of both ACSL ALU and the reference ALU at the same time, and their outputs are compared. If the results are not the same, the simulation stops, and the bug details are reported. The required modifications is then applied to the ACSL design in order to fix the bug. This process repeats until the design passes all the tests successfully.

The *opencores* 8051 ALU [97] has a sequential implementation for the multiplication and division operations. Therefore, this ALU has a clock input which connects to these operations modules. A clock signal with 12 Mhz frequency has been used in the simulation testbench, and the test vectors are applied to both ALUs with reference to this clock. 12MHz clock frequency is one of the most
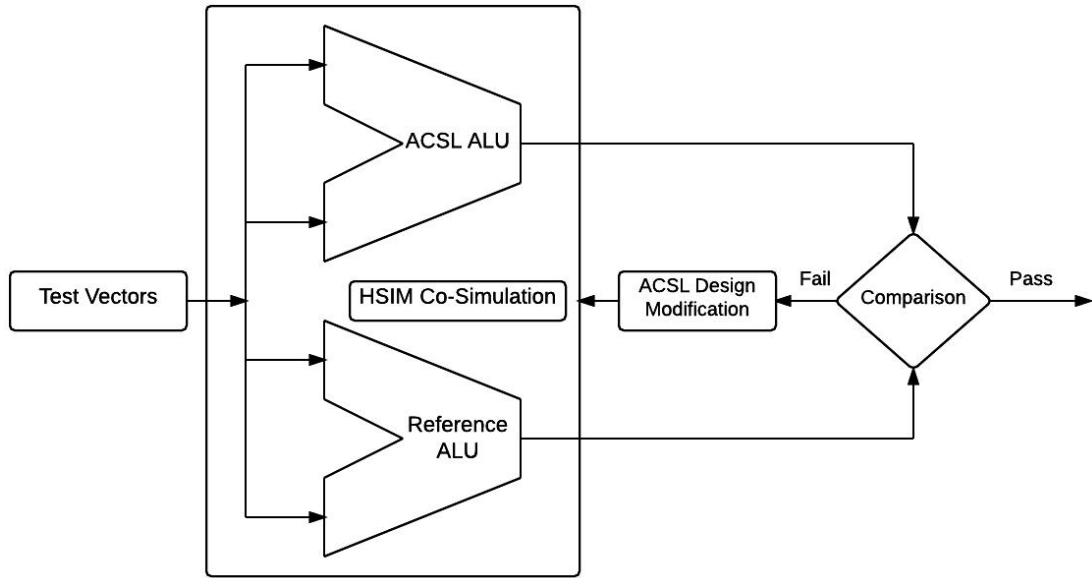
Figure 4.10: Verification Method for ACSL 8051 ALU

common crystal frequencies used for the 8051 [103].

At each rising edge of the clock, the next test vector is applied, and the results of the previous test vector are compared. The mutiplication and division in *opencores* [97] design take 4 clock cycles to complete their computation. For this reason, applying the next test vector and comparing the results are delayed for 4 cycles after these operations.

To simulate the Verilog and SPICE designs together, Synopsys HSIMplus [3] tool has been used. HSIM [104] is a SPICE simulator which provides better speed vs. accuracy trade-off in compare to Synopsys HSPICE. HSIMplus enables VCS/HSIM co-simulation that is useful for simulation of the designs that consist of a combination of SPICE transistor-level circuit netlists and Verilog gate or RTL-level digital modules. It provids an interface from HSIM simulator to Synopsys VCS Verilog simulator [105]. This mechanism is shown in Figure 4.11.

---

[3]Version 2009.07.5
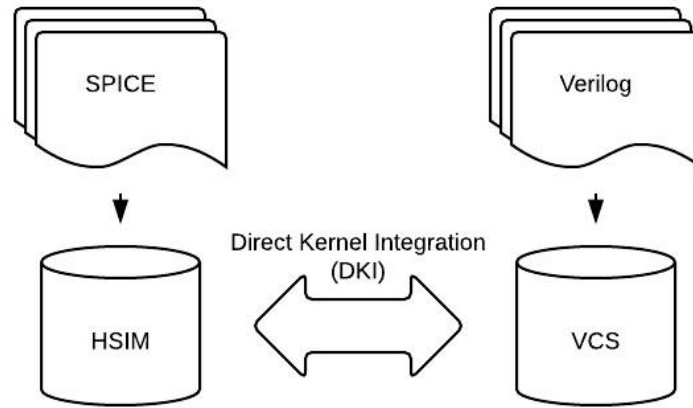
Figure 4.11: HSIMplus Co-Simulation Environment

4.6 Performance Analysis of the ACSL ALU

The ALU implemented in ACSL has been analysed through simulation to extract its main design and performance characteristics. Synopsys HSPICE [4] has been used for simulating the SPICE netlist of the ALU which results from the design flow described in Section 4.1. Figure 4.12 shows the inputs to the HSPICE tool to perform this simulation.
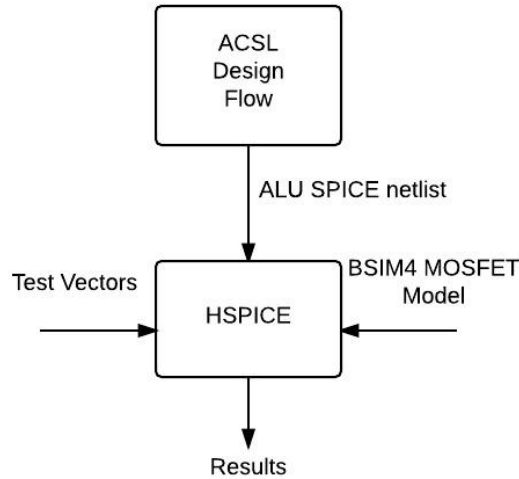
Figure 4.12: HSPICE Simulation of the ACSL ALU

BSIM4 MOSFET model [5] has been used for the transistor level simulation. BSIM (Berkeley Short-channel IGFET Model) [106] refers to a family of MOSFET

---

[4]Version G-2012.06

[5]Version 4.5

transistor models for integrated circuit design. It also refers to the BSIM group
located in the Department of Electrical Engineering and Computer Sciences at
the University of California, Berkeley, that develops these models. Accurate
transistor models are needed for electronic circuit simulation, which in turn is
needed for integrated circuit design [107].

To analyse the ALU circuit, 160 test vectors are applied to the ALU inputs. This
consists of 10 test vectors for each ALU operation. The numbers for ALU inputs
are generated randomly with the uniform distribution. These vectors are applied
in reference to a 12Mhz clock signal similar to the experience described in Section
4.5.

The circuit for the ACSL ALU has been implemented using 45nm technology.
The power supply voltage is set to 1.0V, and the circuit is simulated for 25°C
operating temperature. Table 4.3 summarizes these main parameters.

HSPICE simulation is performed for the transient analysis with 100ps computa-
tion interval time. The initial condition for the node values is set to zero. The
simulation took about 4 hours CPU [6] time to be run for all the test vectors.

The simulation results for power, delay and area of the ALU circuit are shown
in Table 4.4. The longest path delay in the table is associated with division
operation of the ALU. As mentioned in Section 4.3.3, the ACSL circuit for divi-
sion operation consists of 94 asynchronous stages which is more than any other
operation. This justifies having the largest delay for this operation. The ACSL
ALU longest path delay is still far less than the typical operating clock period

Table 4.3: Technology Parameters and Operating Conditions

Transistor Channel Length (L)	45nm
Power Supply Voltage (VDD)	1.0V
Operating Temperature	25°C

---

[6]Dual-Core AMD Opteron(tm) Processor 1222 - 3000.0Mhz

Table 4.4: Power, Delay and Area for ACSL 8051 ALU

Average Power (μw)	47.9209
Peak Power (mw)	1.5731
Longest Path Delay (ns)	12.94ns
Number of Transistors	18026

of 8051 microcontroller, so it makes the ALU sufficiently fast to work in typical 8051 frequency range.

The division and multiplication operations in *opencores* 8051 ALU [97] design need 4 clock cycles to complete their operation. The ACSL ALU makes it possible to run these operations only in one clock cycle, and this highly increases the speed of the microcontroller for the codes containing a large number of MUL and DIV instructions.

The average power and peak power resulted from HSPICE simulation are also reported in Table 4.4. The area is reported in terms of number of transistors in the design. A more detailed analysis of power, delay and area for each individual operation of the ACSL ALU would be presented in Chapter 5.

4.7   Summary

The ACSL circuits offer power predictibility by ensuring that the power required to complete an operation is independent of its inputs. The 8051 ALU is implemented in ACSL to exploit this property for having a power predictable ALU design. The indivisual ALU operations are implemented separately, and then integrated in the final ALU structure. The functionality of the ALU is verified through the simulation, and its design and performance charcteristics are analysed.

In next chapter, the power, delay and area are measured for each indivisual ALU operation, and a power prediction method is presented to estimate the ACSL

ALU power consumption for any given program code.

Chapter 5

Power Prediction Method for The 8051 ALU

In this chapter, a power prediction method for the 8051 ACSL ALU is presented. The ACSL ALU operations have been analysed through the simulation to determine their design and performance characteristics. The power consumption of these operations has a very small variability, and is almost independent from the input patterns. This property makes it possible to predict the power of the ALU only by knowing the number and type of the operations it performs. The 8051 is chosen in this work because of its popularity in embedded system applications, but the method can be applied to any processor.

The prediction method is based on using an 8051 Instruction Set Simulator (ISS) to run the programs, and analyse their instruction trace to extract the number of the ALU related instructions. This also provides the information on the number of times each ALU operation is used by the instructions during the execution of the program on the 8051. The average power of the ALU is then calculated using this information. This method can estimate the power with high accuracy, and over 100 times faster than the gate-level simulation and hundreds of thousends

times faster than the transistor-level simulation.

The structure of this chapter is as follows: In Section 5.1, the power, delay and area of the ACSL ALU operations are analysed using the transistor-level simulation. In Section 5.2, the power prediction method is presented. In Section 5.3 the presented method is used to predict the ALU power for a number of benchmark programs, and the results are compared to the simulation-based methods in terms of accuracy and speed.

5.1 Analysis of Power, Delay and Area for the 8051 ACSL ALU Operations

All the operations of the 8051 ACSL ALU have been analysed separately through the simulation to extract the power, delay and area of each of them. The Synopsys HSPICE [1] is used to simulate the ACSL circuit for each operation in transistor level. The general set-up for this experiment is the same as what described in 4.6. The BSIM4 MOSFET model for 45nm technology is used, and the circuit is simulated for 1.0V power supply at 25°C operating temperature. The initial condition for the node values is set to zero.

In order to evaluate each circuit, 30 random test vectors are selected. These test vectors are applied to the circuit inputs every 82ns (12Mhz). A short pulse (0.5ns) is generated on the request signal of the operation for each test vector applied. This request signal triggers the circuit to operate on the inputs. The experiment is also repeated without activating the request signal to evaluate the circuit in non-operating mode. 12Mhz frequency is used in this experiment, because it is one of the most common crystal frequencies for the 8051 [103]. This frequency is also used in the later simulations for running the software on the 8051 core.

---

[1]Version 4.5

Table 5.1: Power of the ALU operations in operative and non-operative modes

Operation	ADD/SUB	MUL	DIV	AND	OR	XOR	NOT
Power (μw) (Non-Operative)	1.44	3.04	33.65	0.32	0.31	0.57	0.25
Power (μw) (Operative)	6.31	13.69	61.87	1.07	1.08	1.26	0.96
Variation (%)	8.9	13.44	3.9	6.5	6.2	5.5	6.3
Operation	RR	RL	RRC	RLC	XCH	DA	NOP
Power (μw) (Non-Operative)	0.29	0.26	0.25	0.36	0.52	1.04	0.40
Power (μw) (Operative)	0.97	0.97	0.96	1.23	1.66	4.19	1.30
Variation (%)	8.1	7.2	6.9	13.2	9.6	8.7	9.1

HSPICE simulation is performed for the transient analysis with 10ps computation interval time. The CPU[2] time for the simulation was variable from 10 seconds to 3.3 hours depending on the complexity of the operations.

The average power and delay of the operations are measured for each test vector applied. The average power is measured over 82ns period, and the delay is measured as the time interval between the rising edge of the request signal to the rising edge of the acknowledgment signal. The results for the first test vector is discarded to ignore start-up effects, and the average of the results is calculated for the rest of the test vectors.

The results for the average power of the ACSL ALU operations are shown in Table 5.1. The average power is measured for both operative and non-operative modes. In non-operative mode the input values are all zero, and the request signal is inactive, so the power has a lower value and is constant. The power in non-operative mode is mostly consumed by the latches placed between the stages of the ACSL circuit. For this reason, the division circuit which has the maximum number of the ACSL stages consumes the most power in non-operative mode in comparison with the other operations.

---

[2]Dual-Core AMD Opteron(tm) Processor 1222 - 3000.0Mhz

Table 5.2: Power of Multiplication and Division Operations over 4 Clock Cycles

Operation	MUL	DIV
Power (μw)	6.10	40.66
Variation (%)	10.6	3.23

The power values in the operative mode slightly change for each test vector. The average value and the variation are given in the table. The variation metric is the percentage change that the furthest outlier is away from the average, thus represents worst case variation.

The power values in Table 5.1 are measured for all the operations over one cycle of a 12Mhz clock. In the *opencores* 8051 ALU [97] design, Multiplication and division operations take 4 clock cycles to operate. In order to predict the power of the ACSL ALU when it is used inside the *opencores* design, the average power for these operations is also measured over 4 clock cycles. The results of this measurement are given in Table 5.2. These power values would be used in the next section of this chapter to introduce the power prediction method for the ACSL ALU.

The delay and the number of transistors for the ACSL ALU operations are given in Table 5.3. As described in Chapter 4, DIV, MUL, ADD/SUB and DA operations have 94, 16, 11 and 9 ACSL stages respectively. Therefore, the power, delay and area of these operations are more than the others which are implemented in a single stage of ACSL.

Table 5.3: Delay and Area of the ACSL ALU operations

Operation	ADD/SUB	MUL	DIV	AND	OR	XOR	NOT
Delay (ns)	2.22	3.34	13.06	0.79	0.79	0.80	0.78
Variation (%)	1.0	3.0	0.6	0.8	1.0	0.9	1.1
Transistors	1546	4140	8968	222	222	258	186
Operation	RR	RL	RRC	RLC	XCH	DA	NOP
Delay (ns)	0.78	0.78	0.78	0.80	0.85	1.83	0.81
Variation (%)	0.9	0.8	0.7	0.9	0.6	0.8	0.9
Transistors	190	190	186	282	382	942	312

The low variation observed in the power usage of the operations is due to the dynamic nature of the gates. ACSL gates are always discharged and evaluated fully once for each evaluation of the circuit. Static gates only evaluate if the inputs change, but can evaluate many times per circuit evaluation due to glitching. The large timing variation in the static circuit occurs due to the combinatorial, non-synchronised nature of the circuit. ACSL is synchronised to the handshake, thus has less timing variability.

The low power variability of ACSL ALU makes it possible to predict its power just by knowing the types of the operations and the number of times they get activated during the execution of the software on the processor. In next section, a power prediction method for the ALU is presented based on this property.

5.2   Power Prediction Method

5.2.1   The 8051 ALU related Instructions

The *opencores* 8051 [97] that is used in this work includes a 2-stage pipeline. The first pipeline stage fetches and decodes the instruction and its operands. The second pipeline stage computes the result of the first stage and writes it to the memory. An execution cycle is associated with each pipeline stage. In the first execution cycle, the operation code is forwarded to the decoder module where all control signals are set. This includes ALU operand and operation selecting signals. In the second execution cycle, signals reach their destination, the ALU operands are chosen, the operation in the ALU is executed and the result is written to the selected address in the memory.

Not all the instructions need an ALU operation to be performed during their execution. For those that do not need an ALU operation, the operation selecting

Table 5.4: The 8051 Instructions Using ALU operations

Operation	Instruction	Operation	Instruction
ADD/SUB	ADD ADDC SUBB INC DEC CJNE DJNZ JMP CLR[a] MOVC	RR	RR ANL[b]
MUL	MUL	RL	RL ORL[c]
DIV	DIV	RRC	RRC
AND	ANL	RLC	RLC SWAP
OR	ORL	XCH	XCH XCHD
XOR	XRL	DA	DA
NOT	CPL	NOP	Others

a : CLR A   b : ANL C, /bit   c : ORL C, /bit

signal is set to NOP (No Operation). Table 5.4 shows all the instructions which need an ALU operation for their execution. As seen in the table, the ADD/SUB, RR, RL, RLC and XCH operations are used by more than one instruction, and the rest of the operations are related to only one single instruction.

Several instructions need to use the ADD/SUB operation of the ALU. For some of thses instructions, addition or subtraction is their main function, and for some others, it is only one of the steps during their execution. A brief description of each of these instructions follows.

The ADD, ADDC and SUBB instructions are used for addition or subtraction of two 8-bit operands. They store the result of the operation in accumulator, and also set the flags. The INC and DEC instructions increment or decrement the specified 8-bit operand by 1 [96].

The CJNE instruction compares two operands and jumps to the specified destination if their values are not equal. This needs a subtraction to be performed and a zero flag has to be checked to verify the equality of the operands. The DJNZ instruction takes two 8-bit operands. It decrements the byte indicated by the first operand and, if the resulting value is not zero, jumps to the address specified in the second operand [96].

The JMP instruction transfers execution to the address generated by adding the 8-bit value in the accumulator to the 16-bit value in the DPTR register [96]. This means that the JMP instruction needs to use an ALU addition operation to calculate the target address. For this reason the circuit for ADD/SUB operation is implemented so that it can also support 16-bit plus 8-bit addition as described in Section 4.3.1.

The CLR instruction sets the specified destination operand to a value of 0 [96]. In the case that the operand of this instruction is the accumulator (CLR A), a subtraction takes place that subtracts the accumulator from itself, and writes the result back in the accumulator. This is the way that *opencores* 8051 [97] design implements this instruction. If the CLR is used with other operands rather than accumulator, it does not need to use the ADD/SUB operation.

The MOVC instruction moves a byte from the program memory to the accumulator. The address of the desired byte in the code space is formed by adding the accumulator to either the DPTR register or the Program Counter (PC) [96]. Therefore, the MOVC instruction also needs an addition to be performed by the ALU.

The RR and RL operations are used by the RR and RL instructions which rotate the eight bits in the accumulator one bit position to the right or to the left [96]. As described in 4.3.5, the required logic for the AND and OR operations between carry flag and the inverted value of an input bit is provided in the circuit implementing the RR and RL operations. For this reason, ANL and ORL instructions also need to use RR and RL operations when their operands are carry flag and an inverted bit value (ORL C, /bit or ANL C, /bit). If the ORL and ANL instructions are used with other operands, they will use ORL and ANL operations for their execution.

The RRC and RLC operations are used by RRC and RLC instructions to rotate

the eight bits in the accumulator and the one bit in the carry flag one bit position to the right or to the left [96]. As mentioned in 4.3.5, the RLC operation circuit also implements an 8-bit nybble-swap. Because of this, the SWAP instruction also uses this operation to exchange the low-order and high-order nybbles within the accumulator.

The last operation which is used by more than one instruction is XCH. The XCH and XCHD instructions use this operation to exchange either the 8-bit or only the low-order nybble between the accumulator and another operand [96]. The rest of the operations are used only by one single instruction. These instructions were introduced in 4.2 for each operation.

5.2.2   Steps to Predict the ALU Power

After identifying the instructions which use the ALU, the program code can be analysed to extract the number of times these instructions are used during the execution of the software on the processor. For this purpose, an Instruction Set Simulator (ISS) for the 8051 microcontroller is used [108].

An Instruction Set Simulator reproduces the operation of an actual microprocessor by means of a high level microprocessor model. An instruction set simulator can also determine the states of the registers in an actual microprocessor when a specific program is executed [109]. The advantage of using the instruction set simulator is its speed which is much higher than the gate-level or Register Transfer Level (RTL) hardware simulators [110].

Figure 5.1 depicts a block diagram presenting the steps of the proposed power estimation method. In the first step and the second step, the program code is compiled and run on the 8051 Instruction Set Simulator, respectively. The instruction set simulator produces the instruction trace associated with the code.

In the third step, the number of each ALU instruction is extracted from the instruction trace. In the last step, the power calculation unit generates the power estimation for the ALU from the information about the ALU instructions and about the power consumption that is associated with each ALU operations. In the following, the four steps to predict the power consumption of a the ALU are described in more detail.
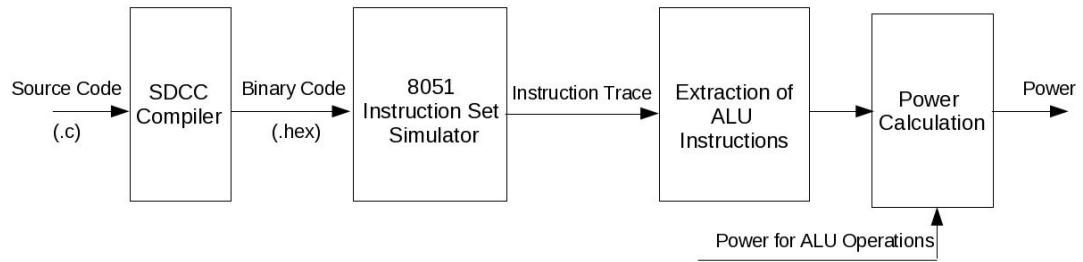
Figure 5.1: Power Prediction Flow

In the first step of the flow, the C program source code (.c) is compiled by the Small Device C Compiler (SDCC)[3]. The SDCC is a retargettable, optimizing ANSI - C compiler suite that targets the Intel MCS51 based microprocessors [111]. The SDCC compiles, assembles and links the source code, and generates the binary file in Intel hex format (.ihx). In addition to the binary file, the assembler source file (.asm) and some other output files are generated.

In the second step, the source code converted into the hex format is fed as the input to the 8051 Instruction Set Simulator (ISS). The 8051 instruction set simulator used in this work is implemented as a part of a project in University of California, Riverside [108]. This simulator[4] is written in C++, and provides statistics on the number of the instructions executed, the number of the clock cycles required for the 8051, the average instructions per second and the execution time for the 8051 working at 12Mhz frequency.

---

[3]SDCC is a free open source software, distributed under GNU General Public License (GPL) - Version 3.5.0

[4]Version 1.4

The simulator continues the execution of the code until the program completion condition is met. The program completion condition is set by the user, and it could be defined as having a specific value in a memory address or on the output ports. The simulator also has the option to print out the value of the output ports of the 8051 anytime that one of them changes during the execution.

The instructions associated with the code are executed one by one on the simulator, and an instruction trace file is generated as the output. The instruction trace is the sequence of all the instructions the 8051 executes while running the code. The Program Counter (PC) and the operands related to each instruction can also be included in the trace file.

In the third step of the power prediction flow, a program (written in C) searches the instruction trace to find the number of the ALU related instructions given in Table 5.4. The number and type of the ALU related instructions along with the power consumption of each ALU operation (given in Table 5.1 and 5.2) are used in the last step to calculate the power of the ALU based on the following equations:

$$P_{ALU} = P_{0\_ALU} + (1/N_{cycles}) \sum_{op} N_{op} C_{op} (P_{op} - P_{0\_op}) \qquad (5.1)$$

$$P_{0\_ALU} = \sum_{op} P_{0\_op} \qquad (5.2)$$

In the equation 5.1, P_{ALU} is the average power dissipated in the ALU during the execution of the code. P_{0_ALU} is the power consumed by the ALU in the non-operative mode. This is calculated as sum of the non-operative mode power for all the operations as given in the equation 5.2.

N_{cycles} is the number of clock cycles required for the 8051 to run the program. This number is provided by the instruction set simulator along with the other statistical information. C_{op} is the number of clock cycles that each ALU operation needs.

For MUL and DIV operations, C_{op} is equal to 4, and for the other operations it is equal to 1.

P_{0_op} is the power consumption of the ALU operation in the non-operative mode. P_{op} is the power consumption related to each ALU operation in the operative mode, and N_{op} is the number of instructions which use that operation. For NOP, this number is calculated as the number of clock cycles minus the total number of cycles related to the other operations. This is given in the equation 5.3.

$$N_{NOP} = N_{cycles} - \sum_{op} N_{op} C_{op} \tag{5.3}$$

Each ALU operation power consumption is P_{0_op} when it is not operating. If the operation gets active, $(P_{op} - P_{0\_op})$ is added to that amount. This explains the equation 5.1 in which this extra amount is multiplied by the number of times the operation is used, and the average of that over all of the cycles is added to P_{0_ALU}.

5.3   Results and Analysis

The method for the power prediction described in 5.2.2 has been applied to a number of benchmark programs in order to estimate the ALU power consumption during their execution on the 8051 microcontroller. These programs represent some popular functions and algorithms in the embedded applications. These include: greatest common divisor (gcd.c), Fibonacci (fib.c), checksum calculator (csumex.c), square root function (sqroot.c), sorting algorithm (sort.c), Proportional Integral Derivative controller or PID controller (pid.c) and discrete cosine transform (dct.c).

In Table 5.5, number of times that each ALU operation has been used by each

Table 5.5: Number of Times the ALU Operations Used by Benchmark Programs

Operation	gcd	fib	csumex	sqroot	sort	pid	dct
ADD/SUB	146	340	512	565	942	24366	722404
MUL	0	0	0	3	0	1243	165184
DIV	0	0	0	0	0	106	0
DA	0	0	0	0	0	0	0
NOT	0	0	0	2	0	103	0
AND	0	0	0	12	0	1047	66
XOR	10	19	0	2	10	262	9288
OR	0	0	69	38	2	2556	0
RL	0	0	0	0	3	0	0
RLC	0	45	0	154	20	8853	165056
RR	0	0	0	50	0	1275	0
RRC	0	0	0	470	0	14097	495616
XCH	0	9	0	937	90	24118	324577
NOP	228	805	1320	1798	3171	101996	2370093

benchmark program is displayed. This number for each operation is the total number of the instructions which use that operation during the execution of the program. It is generated by analysing the instruction trace of the program and counting the number of ALU related instructions given in Table 5.4.

The predicted power of the 8051 ACSL ALU for the benchmark programs is given in Table 5.6. The number of instructions and the number of execution cycles showed in the table are provided by the 8051 instruction set simulator. The power is estimated using the method described in 5.2.2.

To verify the accuracy of the prediction method, the average power of the ALU is also measured through the HSPICE simulation. For this purpose, first the

Table 5.6: Predicted Power and Measured Power for The Benchmark Programs

	gcd	fib	csumex	sqroot	sort	pid	dct
#Instructions	384	1218	1901	4031	4238	180022	4252284
#Cycles	543	1669	3082	4819	6439	224474	5571360
Predicted Power (μw)	44.65	44.40	44.26	44.10	44.18	44.12	44.45
Measured Power (μw)	45.07	44.85	44.62	44.38	44.48	-	-
Error (%)	-0.95	-1.00	-0.81	-0.63	-0.66	-	-

ALU in the *opencores* 8051 [97] design is replaced with the ACSL ALU. Then the benchmark programs are run on the processor core using HSIMplus[5] co-simulation, and the signal values on the input pins of the ALU are stored in a file. In the next step, these values are applied as the input vectors to the ACSL ALU, and the circuit is simulated at transistor level using HSPICE [6]. The HSPICE simulation is very time consuming, but the power measurement result has very high accuracy.

The structure of the 8051 core and its peripherals is shown in Figure 5.2. This structure is depicted based on the *opencores* [97] implementation of the micro-controller. The main components of the 8051 core are the instruction decoder, ALU, Special Function Registers (SFR), memory interface and internal RAM and ROM memories. The core has four 8-bit I/O ports , and is connected to a UART unit and external ROM and RAM memories.
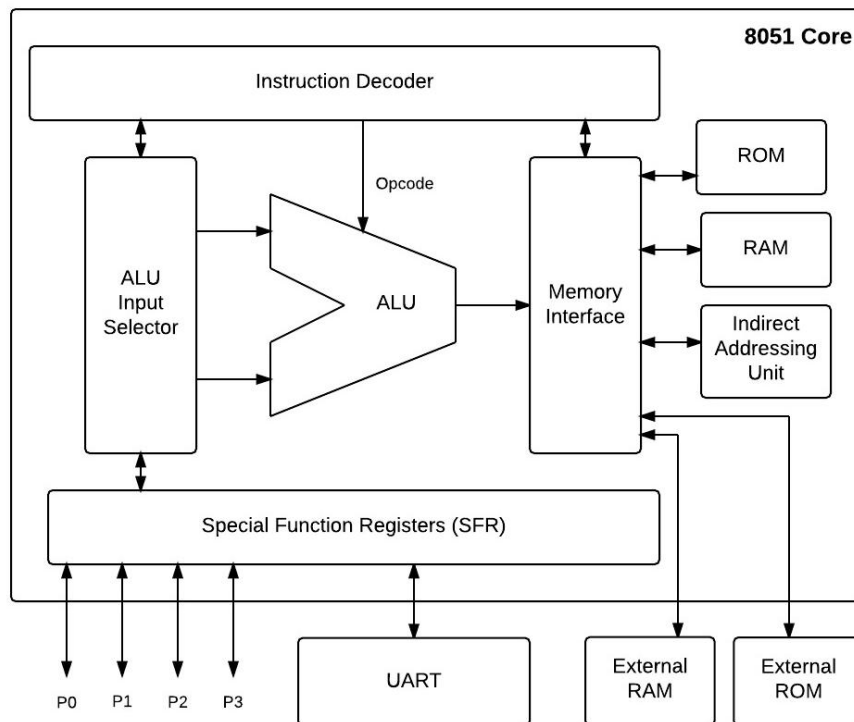
Figure 5.2: The 8051 Microcontroller Structure

---

[5]Version 2009.07.5
[6]Version G-2012.06 32-BIT

The 8051 has separated data and program memory (Harvard architecture). There is 64KB of program memory that 4KB of it is on-chip, and the remaining is external. The program memory is read-only. On-chip data memory is 256 bytes which includes Special Function Registers (SFR). 64KB of external data memory is also available [112] .

Each benchmark program is compiled into Intel hex format using the SDCC compiler. The hex file is then converted to a memory pattern file using a ROM maker program provided by *opencores* [97]. This memory pattern file is the binary representation of the program memory content, and is used to initialise the external ROM memory.

The benchmark programs send the result of their computations to the output ports of the 8051 (P0 to P3). The simulation is terminated when the expected outputs are seen on these ports. Having the expected results on the output ports is also the program completion condition for the instruction set simulator.

As seen in Table 5.6, the percentage error of the estimated power is very small, and even less than the power variability of the operations. The reason is that only one ALU operation is active in each clock cycle, and the rest of the operations are in non-operative mode with completely constant power.

In the simulation-based method for the power measurement, the design needs to be synthesized to generate the gate-level netlist for the target technology. Then the timing information that results from the timing analysis is back annotated to the netlist to define the delay of the gates. Next, a gate-level simulation is performed to generate the switching activity for all of the nodes inside the circuit. At the end, the switching activities are given to the power analysis tool to generate the power consumption for the design. These steps could be very time consuming specially if the circuit is large.

The power consumption of the ALU in *opencores* 8051 design is also measured

Table 5.7: Speedup of The Power Prediction Method

Benchmark	Prediction Method	Gate-Level Simulation	Speedup	Transistor-Level Simulation	Speedup
gcd	2s	1m 44s	52	11h 28m	20646
fib	2s	1m 43s	52	27h 10m	48904
csumex	2s	1m 45	53	35h 46m	64380
sqroot	2s	1m 49s	55	72h 30m	130503
sort	2s	1m 53s	57	103h 21m	186031
pid	4s	5m 50s	88	-	-
dct	6s	14m 28s	145	-	-

for the benchmark programs using a gate-level simulation. The purpose of this measurement is to assess the speed of the simulation-based method at gate-level, and to compare it to the prediction method. The *opencores* 8051 design is implemented in RTL-level using Verilog language. Only the ALU module in this design is synthesized to the gate-level and instantiated inside the RTL level implementation of the core. This approach is adopted because only the switching activity of the ALU nodes is needed for the power measurement, and the rest of the components could be kept in RTL level to increase the speed of the simulation.

In this experiment, Synopsys Design Compiler[7] and Synopsys PrimeTime[8] tools are used for the synthesis and timing/power analysis of the circuit respectively. For the synthesis of the circuit, a TSMC 65 nm process is used operating at 0.9 V. The Synopsys VCS[9] tool is used for the gate-level simulation.

Table 5.7 compares the speed of the power estimation for the prediction method with the simulation-based methods. As seen in the table, the prediction method calculates the power only in a few seconds, but it can take several minutes for the gate-level simulation or several days for the transistor-level simulation depending on the size of the program. This means that the prediction method can be over 100 times faster than the gate-level simulation for the program power estimation,

---

[7]Version B-2008.09-SP4
[8]Version B-2008.12-SP2
[9]Version C-2009.06

and hundreds of thousands times faster than the transistor-level simulation. For the last two benchmark programs (pid and dct), only the gate-level simulation has been performed since the transistor-level simulation could take hundreds of days to complete.

It is worth to mention that the ALU is a quiet small component of the processor core. If the simulation-based methods are used to measure the power of the entire core, it will be much more time consuming due to the growing number of the switching nodes in the circuit. On the other hand, the prediction method will have the same speed if it is implemented for the entire core since it only uses the Instruction Set Simulator (ISS) and the power formulas.

5.4   Summary

In this chapter a power prediction method is presented for the 8051 ACSL ALU. The power consumption of the operations of the ACSL ALU are constant and independent from the input patterns. This property makes it possible to estimate the ALU power only by knowing the number of times each ALU operation is activated during the execution of the program.

In the proposed prediction method, the program is run on an Instruction Set Simulator (ISS) which is a fast and high-level model of the processor core. The instruction set simulator generates the trace of the instructions for the code which is then analysed to extract the number and type of the ALU related instructions. Using the information about the ALU instructions and the energy consumption that is associated with each ALU operation, the ALU power for the software program is calculated.

The presented method achieves less than 1% error and more than 100 times speedup over the gate-level simulation method and hundreds of thousends times

speedup over the transistor-level simulation method.

The proposed methodology can be scaled to full core as well as to other larger processors and processing units. Hence, it can be a first step for design time software power performance estimation.

Chapter 6

Static Average-Case Power Analysis of a Sorting Algorithm

The average-case analysis of the design metrics in the embedded systems is important for efficient budgetting of the resources, and satisfying the design constraints. The average-case analysis provides useful insight about the typical behaviour of the system, and complements the worst-case information to help the designer in taking better strategies in implementing an efficient system.

MOdular Quantitative Analysis (MOQA) is a high level methodology recently proposed for static average-case analysis of the program codes. This methodology enables the prediction of the average number of basic steps during the execution of a program which facilitates the estimation of the complexity measures such as average time or average power consumption.

In this chapter, an average-case processor energy model is presented for the Insertion sort algorithm based on the average number of comparisons in the sorting algorithm resulted from the MOQA analysis. In this work, the parameters of the model are determined for the LEON3 processor core, but the model is general and can be used for any processor. This energy model enables the static estimation of

the average-case processor energy consumption for the Insertion sort program for any given size of the input list. The accuracy and speedup of the model has been evaluated for the LEON3 processor through the power meaurement experiments.

The structure of this chapter is as follows: In Section 6.1, the MOQA methodology and its underlying concepts are introduced. In Section 6.2, the MOQA analysis for the Insertion sort algorithm is presented. In Section 6.3, the LEON3 system design, and the structure of the LEON3 processor core is described. In Section 6.4, the experimental method for the processor power measurement during the execution of the program code is explained. In Section 6.5, the processor energy model is presented, and in Section 6.6 the model is validated using the experimental results.

6.1 MOdular Quantitative Analysis (MOQA)

The average-case timing analysis in the embedded systems is important for implementing the applications that satisfy the certain design constraints, and also for allocating the system resources in an efficient way. The worst-case timing analysis that usually takes place may be too pessimistic, and overshoot the actual time of a large portion of the executions. Average-case information can complement worst-case information to improve budgetting of resources, support soft real-time analysis, and support low-power design [113].

MOdular Quantitative Analysis (MOQA) [114] is a high level methodology for average-case timing analysis of the programs. Time in this context refers to a broad notion of cost, which can be used to estimate the actual running time, but also other quantitative information such as power consumption. In fact, MOQA methodology enables the prediction of the average number of basic steps performed in a computation which can be used to statically analyse the complexity

measures such as average time or average power.

The compositionality is the key property for the static timing analysis. This property allows to estimate the average time of a program code as the sum of the times related to each part of the code. To have a better view of this concept, consider the sequential execution of two program codes P_1 and P_2. If the compositionality is achieved, average-case time of the execution of $P_1;P_2$ could be calculated as the sum of the average-times for each of the programs as shown in Equation 6.1.

$$\overline{T}_{P_1;P_2}(I) = \overline{T}_{P_1}(I) + \overline{T}_{P_2}(O_{P_1}(I)) \tag{6.1}$$

The problem is that the computation of P_1 over its input data (I) will produce new input data ($O_{P_1}(I)$) for P_2, and the average-case time of P_2 depends on the distribution of this new input data. However, one typically cannot track the distribution throughout the computation, and the methods for distribution transformations [115] are purely mathematical, and cannot be used for effective computing of new distributions from prior ones [114].

The compositionality problem for average-case analysis has been overcome in the MOQA approach through the randomness preservation of data. MOQA introduces the notion of "random bag" to represent the data distribution, and uses carefully designed basic operations to ensure that the capacity for such distribution representation is preserved through the computation. This approach makes it possible to track the data distribution during the computation of MOQA programs [116].

To achieve random bag preservation and compositionality, MOQA presents a novel programming language [117] which consists of a suite of data structuring operations, together with conditional, for-loops and recursion. MOQA language

constructs have been designed, when needed, to replace the standard data structuring operations to achieve compositionality. In this way, MOQA enables the compositional determination of the average-case number of basic operations of the programs. MOQA has been specified and implemented in Java 5.0 at CEOL[1]. However, MOQA data structuring operations can be implemented in any standard programming language [116].

The MOQA average-case timing analysis has been focused on data restructuring algorithms which are comparison driven, i.e. for which each action (data-reorganization) is based on a prior comparison between data. The examples are the popular sorting and searching algorithms which are implemented and analysed using the MOQA approach as reported in [114]. The average-case time $\overline{T}_A(n)$ of an algorithm A is defined as the average number of comparisons carried out over inputs of size n. To fine-tune the static analysis further, other basic operations (such as swaps and assignments) can also be accounted for [116].

6.2 MOQA Average-Case Analysis for The Insertion Sort

Sorting algorithms are used to arrange elements of a list in a specific order. Efficient sorting is important to optimize the use of search and merge algorithms that require sorted list. Furthermore, sorting algorithms are widely used in parallel computation, image processing, data aggregation, scheduling, database management and other applications. Due to their widespread applicability, analysing the performance and energy consumption of these algorithms is an important issue.

There are numbers of popular sorting algorithms, like Bubble sort, Heap sort, Insertion sort, Quick sort, Merge sort and etc. Quick sort is the fastest sorting

---

[1]Centre of Efficiency-Oriented Languages, University College Cork, Ireland

algorithm which is used in many programming languages and libraries, but it is
not the most energy saving one. The experiments has shown that Insertion sort
provides the best rationale between performance and energy consumption [118].
For this reason, Insertion sort is chosen in this work for static energy analysis
based on MOQA approach.

In this section, the Insertion sort algorithm is described, and its average-case
analysis by MOQA is presented. This analysis generates the number of compar-
isons that take place in the computation of the algorithm. This number is used
later in this chapter for building the energy estimation model.

6.2.1 Insertion Sort Algorithm

As its name indicates, the Insertion sort algorithm is based on "inserting" a new
element into a sorted list, so that the list remains sorted after this insertion. The
pseudocode for the Insertion sort algorithm is shown in Figure 6.1. At each stage
of the algorithm, the input list (A) consists of two sub-lists: a sorted one and an
unsorted one. Each repetition of the algorithm moves one item from the unsorted
list into the right position in the sorted list, until there are no elements left in the

```
INSERTION-SORT (A)

  for j <- 2 to length[A]

      do key <- A[j]

        Insert A[j] into the sorted sequence A[1 . . j - 1].

      i <- j - 1

      while i > 0 and A[i] > key

          do A[i + 1] <- A[i]

            i <- i - 1

      A[i + 1] <- key
```

Figure 6.1: Insertion Sort Pseudocode

unsorted list. At the beginning, the sorted list contains only the first element of the list. Each time, one element (key) of the unsorted list is compared with the elements of the sorted list until it gets to the right place. The elements that are bigger than key are shifted one place to make space for inserting the key into the list. Figure 6.2 shows that how the element X is inserted into the sorted part of a list.
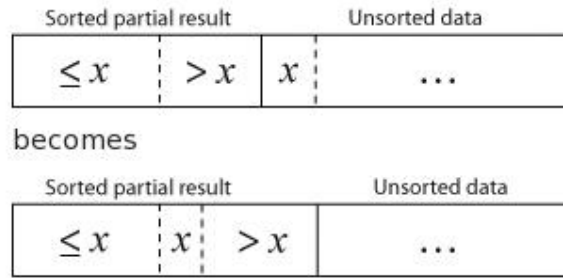
Figure 6.2: Insertion of an element into the sorted part of the list [119]

In the best case of an already sorted list, the insertion sort takes $O(n)$ time: in each iteration, the key element is only compared with the last element of the sorted list. It takes $O(n^2)$ time in the average and worst cases [120].

6.2.2   MOQA Analysis

The MOQA code for the Insertion sort captures the traditional insertion operation, of inserting a single element into a sorted list, via the MOQA product operation ($\otimes$) [114]. This code is shown in Figure 6.3.

Figure 6.3: MOQA Code for Insertion Sort [114]

The MOQA product is a randomness preserving operation which allows the estimation of the average-case time based on the compositionality property. The average-case time for the Insertion sort program ($\overline{T}_I(n)$) is defined as the average

number of comparisons carried out over input size n. The formula for $\overline{T}_I(n)$ resulted from MOQA analysis is given in Equations 6.2 and 6.3. As these equations show, the algorithm has $O(n^2)$ average-case time as expected.

$$\overline{T}_I(n) = \frac{n^2 + 7n - 8H_n}{4} \tag{6.2}$$

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \tag{6.3}$$

For simplicity, the mathematical detail of the MOQA analysis is not given here, but it is available in [114] for further reading.

6.3   SPARC LEON3 Processor

The LEON3 is a synthesisable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture [121]. This open-source processor is designed by Aeroflex Gaisler [122] and is provided as a part of GRLIB IP library [123] under GNU GPL license. The GRLIB IP library is an integrated set of reusable IP cores, designed for System-On-Chip (SOC) development.

Figure 6.4 shows the structure of the LEON3 system platform which consists of the pocessor core, memories and peripherals connecting together via a central AMBA AHB/APB on-chip bus. This structure supports multi-processor design, with up to 4 processor cores capable of delivering up to 1600 Dhrystone MIPS of performance [124].

The LEON3 processor has a 7-stage pipeline and separate instruction and data memories (Harvard architecture). Figure 6.5 shows the internal structure of the processor core. The Integer Unit (IU3) is the heart of the processor which enables the execution of the instructions through the pipeline. The register file contains a configurable number of register windows within the limit of the SPARC standard
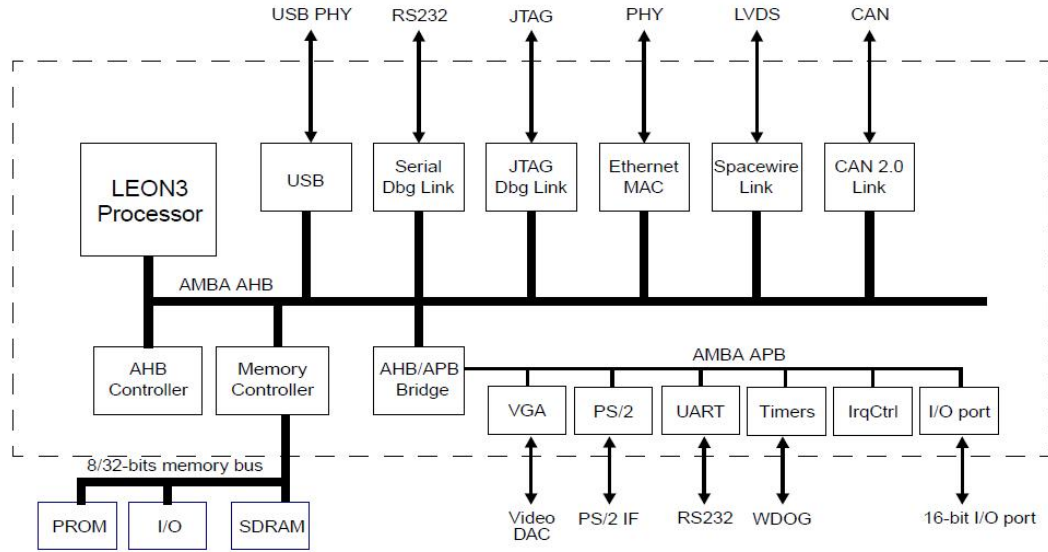
Figure 6.4: Leon3 System Design [125]

(2 - 32), with a default setting of 8. There is the flexibilty in the design to have
division and multiplication units inside the core, or to use compiler to implement
the multiply and divide instructions in software. Using the Floating Point Unit
(FPU) and cache memory is also optional. The trace buffer in the core is a
circular buffer for storing executed instructions and their results. The buffer can
be read out by any AHB master, and in particular by the debug communication
link.
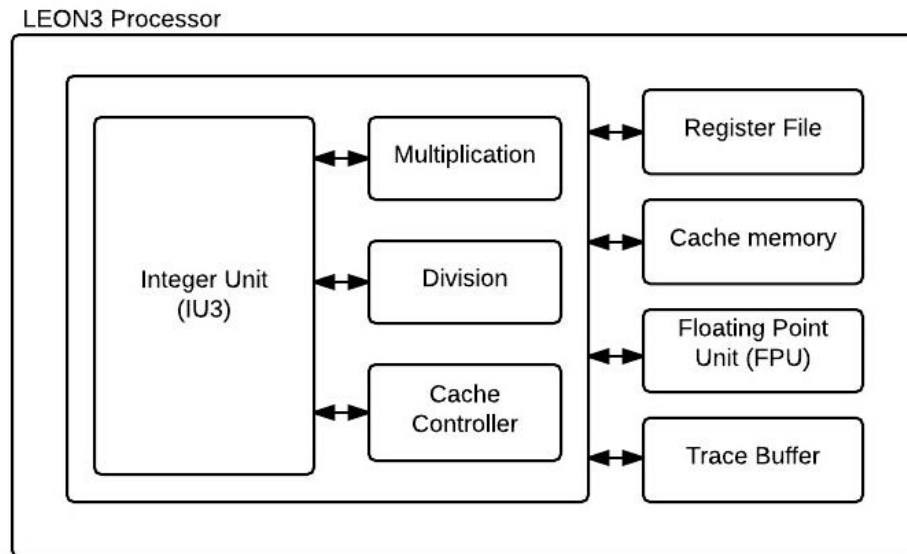
Figure 6.5: LEON3 Processor Core Structure
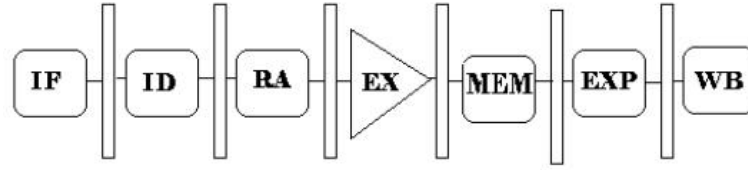
Figure 6.6: Pipelined Stages of LEON3 Integer Unit [126]

The pipeline stages of the LEON3 Integer Unit (IU) are shown in Figure 6.6. These stages include Instruction Fetch (IF), Instruction Decode (ID), Register Access (RA), Execute (EX), Memory (MEM), Exception (EXP) and Write-Back (WB).

The LEON3 uses off-chip external PROM and SRAM memories as the instruction and data memories respectively. These memories along with the memory mapped I/O are connected to an external memory bus which is controlled by a programmable memory controller. The memory controller also works as an interface between the memory bus and the AMBA AHB bus [127].

A full set of scripts is available for the simulation and synthesis of the LEON3 system for ASIC technologies, and a wide range of FPGA boards. The processor is able to work with up to 125 MHz clock frequency in FPGA and with 400 MHz frequency on 0.13 μm ASIC technologies [124].

The LEON3 model is fully parametrized through the use of VHDL generics, and this makes it highly configurable. A graphical configuration tool is available to configure the processor and other on-chip peripherals. Number of processors, number of register windows, size of the cache memories and many other parameters are configurable [124].

The LEON3 also provides a Debug Support Unit (DSU) which allows non-intrusive debugging on the target hardware by entering the processor in the dubug mode, and providing full access to all the registers and caches through a debug support interface. A SPARC Reference Memory Management Unit (MMU) is

also provided for advanced memory management and protection. Having the DSU and MMU in the design is optional [128].

Being SPARC V8 conformant, compilers and kernels for SPARC V8 can be used with LEON3. To simplify software development, Aeroflex Gaisler is providing BCC (Bare-C Cross Compilation) [129], a free C/C++ cross-compiler system based on GCC [130] and the Newlib [131] embedded C-library [124]. The required scripts are also provided to compile the C programs, and generate the ROM and RAM images for initializing the PROM and SRAM memories with instruction and data contents.

6.4 Experimental Method

The sorting algorithm takes a list of numbers as the input, and generates the sorted list as the output. For building the average-case energy model for the algorithm, the processor power consumption needs to be measured for a large set of random input numbers and permutations. In this section, the method used for generating the random numbers, the power measurement flow, and the way this flow is automated are described.

6.4.1 Random Number Generation

In this work, the random numbers are generated using the Pseudo-Random Number Generator (PRNG) libraries developed at Technical University of Denmark [132]. These libraries are written in C++ language, and provided with open source under the GNU general public license. They can be used to generate floating point or integer random numbers with uniform distributions, or non-uniform random numbers with several different distributions.

These C++ libraries implement the following PRNGs for uniform distribution: Mersenne Twister, SFMT and Mother-of-all [133]. In this work, the Mersenne Twister [134] PRNG is used to generate the input numbers with uniform distribution for the sorting algortithm. The Mersenne Twister is the first PRNG to provide fast generation of high-quality pseudo-random integers , and has become very popular in recent years because of its long cycle length [135].

The random numbers generated for the sorting algorithm can be placed in the input list in different permutations. The number of the permutations for a specific combination of numbers can be very large, and it is not always possible to run the experiment for all of them. In such cases, it is needed to select a set of the possible permutations randomly.

The number of permutations for a list consisting of n numbers is $n!$. These permutations can be numbered from 0 to $n! - 1$ where the first permutation is the ordered list, and the last permutation is the reverse-ordered list. The list gets more and more disorderd going from the first to the last permutation. For selecting a set of permutations for the experiment, the Mersenne Twister PRNG is used to generate random numbers between 0 to $n! - 1$ with uniform distribution. In this way, the level of disorder for the selected permutations has a uniform distribution, and this is useful for getting more precise results for the average-case.

6.4.2   Power Measurement Flow

Figure 6.7 shows the flow for measuring the power consumption of the LEON3 processor core. According to this flow, the processor core is first synthesised to generate the gate-level netlist. Then, the timing analysis is performed, and the delay of the gates is extracted, and back-annotated to the netlist. Next, the switching activities of the nodes are recorded through the gate-level simulation.

Additionally, the RTL-level simulation is performed to extract the timing interval for the execution of the code on the processor. In the final step, the information about the exection time interval and switching activities are used to analyse the processor power consumption.
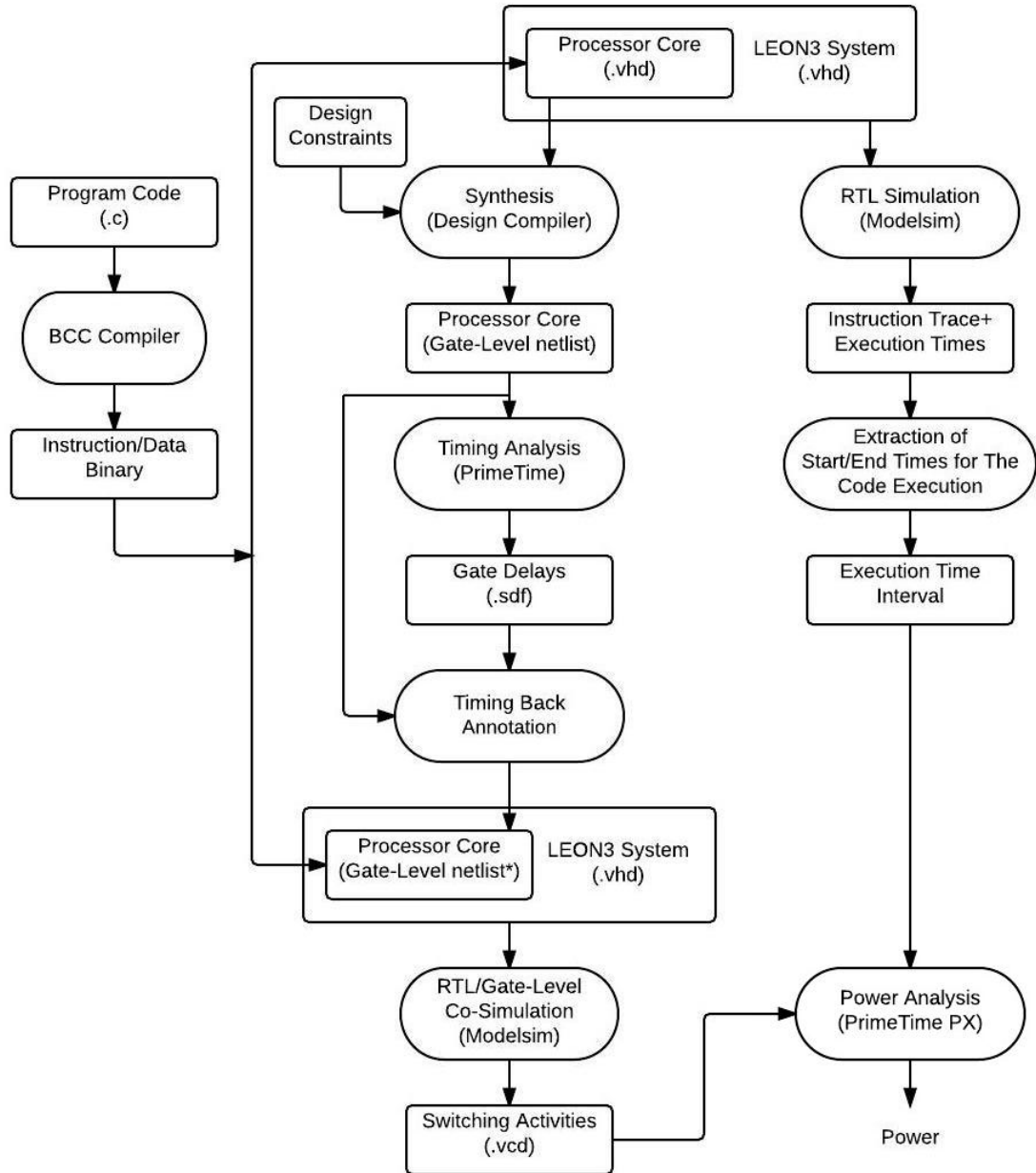
Figure 6.7: Power Measurement Flow

The power analysis is also possible with the switching activities extracted from the RTL-level simulation. However, the accuracy of this method is low, because all

of the design nodes are not covered in such analysis. Specifically for the LEON3 processor, most of the signal names are changed during the gate-level synthesis, and cannot be mapped to the names at the RTL-level. This leads to the poor net coverage in the power analysis. The extensive use of the *record* data structure in the VHDL code for the LEON3 processor is the reason for the change of the signal names at the gate-level.

The default configuration is used in this work for the LEON3 system. In this configuration, the LEON3 processor core has the multiply/divide units, and uses 4KB 1-way (direct-mapped) instruction and data caches. The FPU is not used by default, and the size of the PROM and SRAM external memories is 4MB.

In the following, each step of the power measurement flow is described in more details.

6.4.2.1   Compiling The Code

As mentioned in 6.3, the BCC (Bare-C Cross Compilation) is provided as a free C/C++ cross-compiler system for the LEON3 processor. It supports hard and soft floating-point operations, as well as SPARC V8 multiply and divide instructions. The BCC[2] compiler is used in this work to compile the C program code for generating the SRECORD [136] files to initialize the PROM and SRAM memories. The SRECORD files convey the instruction and data memory images in the form of binary information in ASCII hex.

Since the Floting Point Unit (FPU) is not used in the processor core, the required option is selected for the compiler to emulate the floating point operations in software. The optimization option is also set for the maximum performance and minimal code size.

---

[2]Version 3.4.4

6.4.2.2   RTL-Level Simulation

To determine the processor power consumption during the execution of a specific part of the program code, it is required to find the time interval in which the processor executes the instructions associated to that part of the code. The purpose of the RTL-level simulation is to find this execution time interval which would be used later in the flow for analysing the power.

The required scripts for a number of popular design simulators are provided by Aeroflex Gaisler to facilitate the simulation of the LEON3 system . In this work, Mentor Graphics ModelSim[3] simulator is used for the RTL-level simulation.

The entire LEON3 design including the external memories is instantiated in a VHDL testbench. In this testbench, the PROM and SRAM memories are loaded with the instruction and data contents generated by the compiler. The initial value for the input signals and the condition for terminating the simulation are also set.

A SPARC disassembler is provided in the LEON3 design to disassemble the executed instructions during the simulation, and print them in the simulator console [127]. This also prints the exection time for each instruction. The execution time is the time that the execution of the instruction is completed, and it leaves the processor pipeline.

To find the execution time interval, the dissassembly feature is enabled using the LEON3 configuration tool, and the instruction trace for the code is generated during the RTL-level simulation. The first and last assembly instructions are determined for the specific part of the code that its power consumption is desired. A program searches the instruction trace to find these instructions, and the execution time associated to them are recorded to be used later in the flow

---

[3]Version SE-64 6.5c

for measuring the power.

6.4.2.3   Synthesis

The LEON3 processor core is synthesised using the Synopsys Design Compiler[4] tool for the TSMC 65nm general-purpose (GP) CMOS technology. The nominal process with the worst-case operating condition (0.9V VDD voltage and 125°C temperature) is used in the synthesis, and the timing constraint is set for 300Mhz clock frequency. The required commands are used to fix the hold time violations reported by the tool.

The LEON3 design needs to be configured to adapt with the synthesis tool and the target library [127]. This configuration determines whether the technology dependant mega-cells (ram, rom, pads) get automatically inferred or directly instantiated by the synthesis tool. The LEON3 supports a number of target technologies for using the direct instantiation option. For any technology that is supported by synthesis tool, and is capable of automatic inference of mega-cells, the inference option can be used. The target technology and synthesis tool used in this work supports automatic inferring of RAMs and pads, so this option is used in the synthesis configuration.

The gate-level netlist for the LEON3 processor core is generated in both DDC and Verilog format. The hierarchical boundaries of the top-level submodules in the core are preserved by using the required synthesis constraints. This makes it possible to also have the report of the power consumtion of the submodules later in the flow. Integer unit, cache controller, multiply and divide units are considered as one submodule. The other submodules are the register file, cache memory and trace buffer. The area and timing information resulted from the synthesis of the design are reported in Table 6.1.

---

[4]Version B-2008.09-SP4

Table 6.1: Area and Delay for the LEON3 Processor Core

Combinational Area (μm^2)	334,509.84
Non-Combinational Area (μm^2)	813,892.65
Total Cell Area (μm^2)	1,148,402.49
Leaf Cell Count	183,373
Levels of Logic in Critical Path	24
Critical Path Delay (ns)	2.16
Critical Path Slack (ns)	1.06

6.4.2.4 Timing Analysis

The timing analysis for the gate-level netlist is performed using the Synopsys PrimeTime[5] tool. This analysis produces the SDF (Standard Delay Format) [137] file which contains timing information of all the cells in the design according to the target technology [138]. This timing information is needed for simulating the gate-level netlist.

6.4.2.5 Gate-Level Simulation

The gate-level simulation is required to find the switching activity of the signals in the processor core. For the gate-level simulation, the timing information in the SDF file is back-annotated to the cells in the netlist. The RTL model of the LEON3 processor core is replaced with the gate-level model resulted from the synthesis. Since the rest of the system is still in the RTL level, the RTL/gate-level co-simulation is performed using the Mentor Graphics ModelSim[6] simulator. For this simulation, the library of the standard cells for the target technology is compiled along with the processor netlist.

The name of most of the I/O ports for the processor module changes through the gate-level synthesis. The change in the names happens since the I/O ports with *record* type are converted to the *bit vector* type by the synthesis tool. For this

---

[5]Version B-2008.12-SP2
[6]Version SE-64 6.5c

reason, a Verilog interface is added to the netlist to adapt the gate-level names with the RTL-level names.

The switching activity of the signals are recorded in the VCD (Value Change Dump) [139] file format. In order to use Verilog language features for generating the VCD file, the LEON3 VHDL testbench is converted to the Verilog format for the gate-level simulation.

If the size of the VCD file gets very large, it can stop the simulation. To avoid this problem, when the the number of input samples for the simulation is large, the samples are divided into the smaller groups to perform the simulation separately for each group. In this way, the size of the VCD files is kept under 1.8GB.

6.4.2.6   Power Analysis

The power consumption of the processor during the execution of the code is determined in the final step of the flow. The Synopsys PrimeTime PX[7] tool is used for measuring the power. The switching activity of the processor signals (VCD file), the execution time interval for the code, and the gate-level netlist are given to the power analyser, and the processor power during the given time interval is measured. This measurement is very accurate since it achieves 100% net coverage. The power for the top-level submodules in the core is also reported.

6.4.3   Automation of The Flow

To run the power measurement flow for a large number of input samples, it is needed to automate the flow using the required programs and scripts. Figure 6.8 shows the programs and scripts used in this work to automate the different parts of the flow.

---

[7]Version B-2008.12-SP2
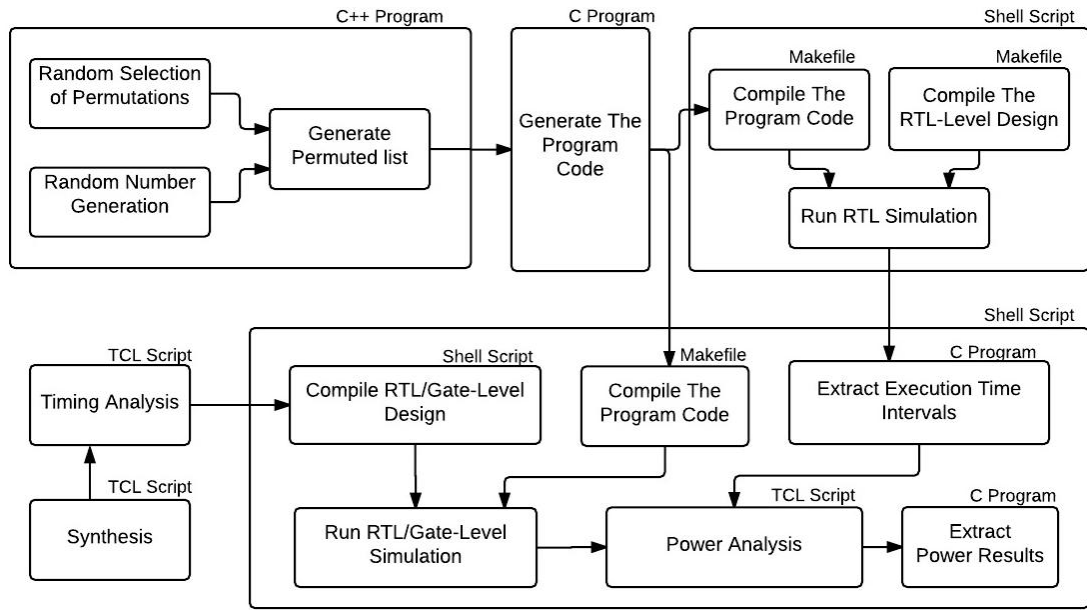
Figure 6.8: Programs and Scripts to Automate the Power Measurement Flow

For generating random numbers and permutations with uniform distribution, a C++ program is written using the PRNG libraries mentioned in 6.4.1. This program generates the random numbers, selects the random permutations, and permutes the list of numbers according to the selected permutations. The permuted lists for each set of random numbers are stored in a separate input data file.

In the next step, a C program reads the input data file, and generates the program code for the sorting algorithm operating on the input lists. The generated code is a C program in which the input lists are stored in the array data structure, and the sort function is called repeatedly to operate on the lists. The large input data files are broken in smaller parts, and the code is generated for each part separately. This prevents the problem of having large VCD files later during the gate-level simulation.

A shell script is written for automating the RTL-level simulation. The script calls the required Makefiles to compile the program code and the LEON3 system, and performs the RTL-level simulation to run the code on the processor. This is

repeated for all of the codes generated for different input data , and the instruction trace for each code is produced.

Another shell script automates the rest of the flow. A C program is called in this script to extract the execution time intervals from the instruction traces produced by the RTL-level simulation. This program extracts the time intervals, and generates a TCL (Tool Command Language) [140] script for the power analysis tool to measure the processor power during those intervals. The shell script also compiles the design, runs the RTL/gate-level simulation for all the program codes, and records the switching activities in the VCD files. It also launches the power analysis tool using the TCL script, and produces the reports for the processor power measurements. In the final step, a C program is called to extract the power results for the processor core and its submodules from the reports generated by the power analysis tool.

The synthesis of the processor core, and the timing analysis of the netlist are also performed using TCL scripts.

6.5 Processor Energy Model For the Insertion Sort Algorithm

The average-case processor energy model for the insertion sort algorithm is based on the average number of times each part of the program code is executed, and the processor energy consumption associated to those parts. In Table 6.2, the program code for the insertion sort is divided into the smaller parts, and the average number of the executions for each part is given. The SPARC assembly code for each part of the C program is also shown in the table.

The Insertion sort program consists of a *for* loop with a *while* loop nested inside

Table 6.2: Partitioning of the code for the Insertion Sort Program

SPARC Assembly Code	Program Code in C	Average of Execution Times	Energy Usage
mov 1, %o7	for(i=1; i<LIST_SIZE; i++){	$N_1 = 1$	E_1

L1: sll %o7, 2, %g1
add %o7, -1, %i4
ld [%i0 + %g1], %i2
clr %i1
	key = num_list[i];
	j = i - 1;
	done = 0;
		$N_2 = n - 1$	E_2

sll %i4, 2, %g1 ld [%i0 + %g1], %i5 L2: cmp %i5, %i2 ble L3 add %g1, %i0, %i3	do{ if (key < num_list[j]){	$N_3 = \overline{T}_I(n)$	E_3

addcc %i4, -1, %i4
st %i5, [%i3 + 4]
bneg L3
sll %i4, 2, %g1
	num_list[j+1] = num_list[j];
	   j--;
	   if (j < 0)
	   done = 1;
	}else
	   done = 1;
		$N_4 = \overline{T}_I(n) - (n-1)$	E_4

cmp %i1, 0 be,a L2 ld [%i0 + %g1], %i5	}while (!done);	$N_5 = \overline{T}_I(n)$	E_5
L3: add %g1, %i0, %g1 inc %o7 cmp %o7, 7 ble L1 st %i2, [%g1 + 4]	num_list[j+1] = key; }	$N_6 = n - 1$	E_6

it. The *for* loop is executed for all the elements in the input list except the first one. Therefore, the parts of the code inside the *for*, and outside the *while* loop are executed $n - 1$ times (N_2, N_6), assuming the size of the list is n.

According to the Insertion sort algorithm explained in 6.2.1, the *while* loop is used for comparing one element (*key*) of the list with the previous elements which are already sorted. This comparison continues until the *key* element reaches to the right place to be inserted in the sorted list. The average number of comparisons ($\overline{T}_I(n)$) is known from the MOQA analysis described in 6.2.2, and is given in the Equation 6.2. The third part of the code in the table is related to the comparison operation, so it is executed $\overline{T}_I(n)$ times in average (N_3).

One shift operation takes place after each comparison except for the last comparison when the *key* reaches to the right place. For this reason, the number of shift operations for each execution of the *for* loop is equal to the number of comparisons minus one. For the entire algorithm, the shift operations take place $n - 1$ times less than the comparisons (N_4).

By knowing the average number of the executions of each part of the code, the average-case energy for the Insertion sort program can be estimated using the energy model given in the Equation 6.4.

$$\overline{E}_I(n) = \sum_{i=1}^{6} N_i E_i \tag{6.4}$$

By substituting the N_i values from the Table 6.2, and $\overline{T}_I(n)$ from the Equation 6.2, the extended form of the energy model can be obtained as shown in the Equation 6.5.

$$\overline{E}_I(n) = E_1 + (n - 1)(E_2 - E_4 + E_6) + (\frac{n^2 + 7n - 8H_n}{4})(E_3 + E_4 + E_5) \tag{6.5}$$

The energy consumption of each part of the code (E_i) is determined for the LEON3 processor through the power measurement flow described in 6.4.2. However, the energy model is general, and can be used for any processor core. Only the E_i parameters need to be measured for the target processor to customize the model.

The proposed energy model enables the estimation of the average-case processor energy consumption for the Insertion sort algorithm statically for different sizes of the input list, and eliminates the need for the time-consuming simulation-based measurements.

6.6 Results and Analysis

The processor energy model presented for the Insertion sort program has been validated through the power measurement experiments. The LEON3 processor energy consumption is measured for the Insertion sort program for size 4, 8 and 16 of the input list. The experimental results are compared to the energy estimation results to evaluate the accuracy of the model.

6.6.1 Energy Model Parameters

One of the factors which impacts the processor energy consumption is the range of the input data for the program. The energy consumption depends on the number of switching bits in the input data. The bigger numbers with more toggling bits create more switching activity inside the processor which leads to the higher energy consumption.

The LEON3 is a 32-bit processor, so the input numbers for the programs are represented in 32-bit format. However, depending on the range of the numbers, the switching bits can be between 0 to 32 bits. Figure 6.9 shows the LEON3 processor power for the Insertion sort program with size 4 for the input list. The power is measured for four different data input ranges: 8-bit, 16-bit, 24-bit and 32-bit. For each range, 10 different sets of random numbers with uniform distribution over the range are selected, and the processor power is measured for all the possible permutations ($4! = 24$) of each set. As expected, the results show that power consumption goes high when the range of the numbers increases.

Considering the impact of the input data range on the processor power consumption, it is reasonable to take this factor into account as a parameter in the proposed energy model. However, it would make the model very complex to be built for all the possible ranges of data. A reasonable compromise is to provide
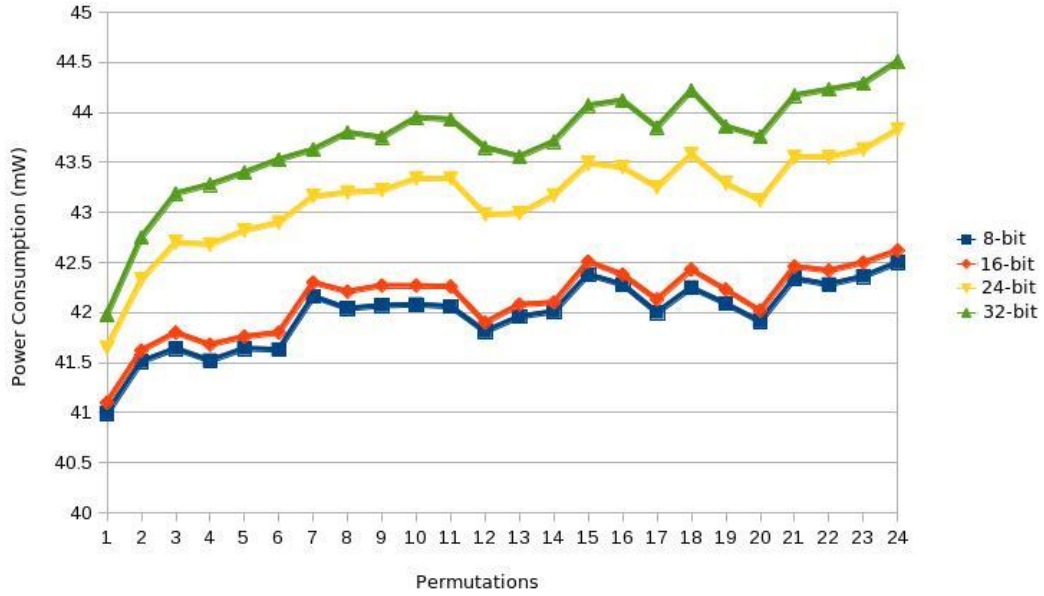
Figure 6.9: LEON3 Processor Power Consumption For The Insertion Sort with Input List Size $= 4$

the option to use the model for a few typical ranges of data. For this reason, the parameters of the energy model (E_i) are measured for four different data input range (8, 16, 24 and 32 bits). This creates the flexibilty to choose the right set of parameters depending on the range of the input data for the specific application to have a more precise estimation of the energy.

The LEON3 processor energy consumption for individual parts of the code is measured to determine the E_i parameters in the model. The value for theses parameters are given in Table 6.3 for 8-bit, 16-bit, 24-bit and 32-bit ranges of data.

Table 6.3:

Data Range	E_1 (nJ)	E_2 (nJ)	E_3 (nJ)	E_4 (nJ)	E_5 (nJ)	E_6 (nJ)
8-bit	0.825	3.289	4.958	4.113	2.484	4.918
16-bit	0.828	3.301	4.974	4.127	2.492	4.934
24-bit	0.848	3.379	5.094	4.226	2.552	5.053
32-bit	0.859	3.425	5.163	4.283	2.587	5.121

6.6.2   Validation of The Energy Model

To validate the proposed energy model, the average energy consumption of the LEON3 processor is measured for the Insertion sort program for size 4, 8 and 16 of the input list. In Table 6.4, the results of these measurements are compared with the energy estimated by the proposed model, and the percentage of the estimation error is calculated. This error is not merely related to the accuracy of the model, and it is partly due to the fact that measurements represent the average energy values for a set of random numbers and permutations not all of the possible input data.

The measurements for list size 4 has been performed for 10 different sets of input numbers and for all the possible permutations of each set. For list size 8 and 16, 1000 permutations are chosen randomly as described in 6.4.1, and are used with 10 sets of random data to perform the measurements. The range of the input data in these experiments is between 0 to 255 (8-bit). Accordingly the E_i parameters for 8-bit data range are used in the model to estimate the energy.

Table 6.5 shows the amount of time needed for running the power measurement experiments for each size of the input list. As the table shows, the simulation-

Table 6.4: Measured Energy and Estimated Energy for LEON3 Processor for the Insertion Sort Program

Input List Size	Measured Energy (μJ)	Estimated Energy (μJ)	Error (%)
4	0.087	0.092	8.31
8	0.277	0.313	10.56
16	0.939	1.047	11.45

Table 6.5: Input Data and Processor Power Measurement Time for The Insertion Sort Program

Input List Size	Input Data	Power Measurement Time
4	10 sets, 24 Permutations	6.5 hours
8	10 sets, 1000 Permutations	11 days
16	10 sets, 1000 Permutations	14 days

Table 6.6: Percentage of The Energy Consumption in Each Submodule of The LEON3 Processor

Processor Submodules	Percentage of Energy Usage (%)
Integer Unit, Multiply/Divide Units, Cache Controller	4.05
Register File	5.09
Cache Memory	83.11
Trace Buffer	7.74

based power measurements are very time-consuming, and can take several days to be performed for a reasonable number of input samples. The energy model eliminates the need for these measurements, and makes it possible to estimate the processor energy consumption in a fraction of a second.

The percentage of energy consumption of top-level submodules of the LEON3 processor are given in Table 6.6. These are the average values measured through the experiments to give a view of the the amount of energy consumed by each part of the processor core. As the results show, most of the energy in the processor core is consumed in the cache memory.

6.7   Summary

In this chapter, a static model is proposed to estimate the average-case energy consumption of the processor core during the execution of the Insertion sort program. This energy model is built based on the average number of comparisons carried out during the execution of the sorting algorithm. The number of comparisons is derived from the average-case timing analysis using MOQA (MOdular Quantitative Analysis) methodology. MOQA is a new high level methodology which enables the static prediction of the average number of basic steps performed in the computation of the programs.

The proposed energy model is general, and can be used for any processor, but the parameters of the model in this work are determinded for the LEON3 processor core. The model has been validated through the power measurement experiments. Using this model enables the designers to statically estimate the energy usage of the processor, and eliminates the need for time-consuming measurements.

Chapter 7

Conclusion

In this thesis, two fast and high-level methods for the prediction of processor power consumption have been presented. The first method is based on a design methodology called Asynchronous Charge Sharing Logic (ACSL) and uses the power predictability property of the circuits designed with this methodology. The second method is based on the average number of basic steps during the execution of a program, and the processor power consumption associated to each step. In this method, the number of basic steps is generated using a high-level methodology called MOdular Quantitative Analysis (MOQA). These methods enable the software developers to have a fast and accurate power estimation for their programs, and generate more power-efficient codes.

In the first part of the thesis, the Arithmetic Logic Unit (ALU) of 8051 micro-controller is implemented using ACSL design style. The power consumption of arithmetic and logical operations of this ALU is almost constant and independent from the input patterns. This property makes it possible to estimate the ALU power usage by knowing the number of times each operation is performed, and the amount of power it consumes. An 8051 Instruction Set Simulator (ISS) is used to generate the program instruction trace, and the ALU related instructions in the

107

trace are counted to find the number of times each ALU operation is activated during the execution of the program. The total power is estimated using this number and the power usage of each ALU operation which is measured through the simulations. This method can estimate the ALU power with less than 1% error, and over 100 times faster than the gate-level simulation and hundreds of thousends times faster than the transistor-level simulation.

In the second part of the thesis, an average-case energy model for the Insertion sort algorithm is developed. This model is based on the number of comparisons that take place during the execution of the sorting algorithm. This number is derived from analysing the Insertion sort algorithm using MOQA methodology. The number of times each part of the program code is executed on the processor is calculated, and the energy consumption associated to each part is measured. Using this information the processor energy model for the sorting algorithm is built. This model can predict the processor power usage for any given size of the input list for the Insertion sort algorithm. The parameters of the model are determind for LEON3 processor core, but the model is general and can be used for any processor. The model is validated through the power measrement experiments, and estimates the energy usage with high accuracy and orders of magnitude faster than simulation-based methods.

7.0.1   Future Work

In this work, the ACSL design methodology is used to implement a power predictable ALU circuit. ALU is only one of the functional units inside the processor core. The other functional units include the instruction decoder, fetch unit, memory interface, register file, etc. The next step toward a fully predictable processor core is implementing these units using ACSL design style. The relationship between the instruction trace and the activation of each unit during the execution

of the code needs to be determined, and the relevant power models get built.

The other potential future work is building the power models for the other algorithms which can be developed using MOQA methodology. In fact, the meaurement of power consumption for the basic operations used in MOQA language can extend this methodology to be used for power analysis of a wide range of programs.

References

[1] Thomas A Henzinger. Two challenges in embedded systems design: Predictability and robustness. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366(1881):3727–3736, 2008.

[2] Shinan Wang, Hui Chen, and Weisong Shi. Span: A software power analyzer for multicore computer systems. *Sustainable Computing: Informatics and Systems*, 1(1):23–34, 2011.

[3] David Brooks, Vivek Tiwari, and Margaret Martonosi. *Wattch: a framework for architectural-level power analysis and optimizations*, volume 28. ACM, 2000.

[4] Doug Burger and Todd M Austin. The simplescalar tool set, version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, 1997.

[5] G Cai and C Lim. Architectural level power/performance optimization and dynamic power optimization. *Proc. Cool Chips Tutorial at 32nd ISCA*, 1999.

[6] Rita Yu Chen, Mary Jane Irwin, and Raminder S Bajwa. Architecture-level power estimation and design experiments. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 6(1):50–66, 2001.

[7] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480. IEEE, 2009.

[8] Lukai Cai and Daniel Gajski. Transaction level modeling: an overview. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24. ACM, 2003.

[9] Nagu Dhanwada, Reinaldo A Bergamaschi, William W Dungan, Indira Nair, Paul Gramann, William E Dougherty, and Chao Lin. Transaction-level modeling for architectural and power analysis of powerpc and coreconnect-based systems. *Design Automation for Embedded Systems*, 10(2-3):105–125, 2005.

[10] Robertas Damaševičius and Vytautas Štuikys. Estimation of power consumption at behavioral modeling level using systemc. *EURASIP Journal on Embedded Systems*, 2007(1):068673, 2007.

[11] Robertas Damaševičius. Estimation of design characteristics at rtl modeling level using systemc. *Information Technology And Control*, 35(2), 2015.

[12] Giovanni Beltrame, Donatella Sciuto, and Cristina Silvano. Multi-accuracy power and performance transaction-level modeling. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(10):1830–1842, 2007.

[13] Feng Liu, Qingping Tan, Xiaoyu Song, and Naeem Abbasi. Aop-based high-level power estimation in systemc. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, pages 353–356. ACM, 2010.

[14] Matthias Kuehnle, Andre Wagner, Alisson V Brito, and Juergen Becker.

Modeling and implementation of a power estimation methodology for systemc. *International Journal of Reconfigurable Computing*, 2012:5, 2012.

[15] Gilberto Contreras and Margaret Martonosi. Power prediction for intel xscale® processors using performance monitoring unit events. In *Low Power Electronics and Design, 2005. ISLPED'05. Proceedings of the 2005 International Symposium on*, pages 221–226. IEEE, 2005.

[16] Michael D Powell, Arijit Biswas, Joel S Emer, Shubhendu S Mukherjee, Basit R Sheikh, and Shrirang Yardi. Camp: A technique to estimate per-structure power at run-time using a few simple parameters. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 289–300. IEEE, 2009.

[17] Frank Bellosa. The benefits of event: driven energy accounting in power-sensitive systems. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, pages 37–42. ACM, 2000.

[18] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 93. IEEE Computer Society, 2003.

[19] Ramon Bertran, Marc Gonzalez, Xavier Martorell, Nacho Navarro, and Eduard Ayguade. Decomposable and responsive power models for multicore processors using performance counters. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 147–158. ACM, 2010.

[20] Russ Joseph and Margaret Martonosi. Run-time power estimation in high performance microprocessors. In *Proceedings of the 2001 international symposium on Low power electronics and design*, pages 135–140. ACM, 2001.

[21] Wei Wu, Lingling Jin, Jun Yang, Pu Liu, and Sheldon X-D Tan. A systematic method for functional unit power estimation in microprocessors. In *Proceedings of the 43rd annual Design Automation Conference*, pages 554–557. ACM, 2006.

[22] Karan Singh, Major Bhadauria, and Sally A McKee. Real time power estimation and thread scheduling via performance counters. *ACM SIGARCH Computer Architecture News*, 37(2):46–55, 2009.

[23] Diego A Murio. Inverse heat conduction problem. *The Mollification Method and the Numerical Solution of Ill-Posed Problems*, pages 60–106, 1993.

[24] Hendrik F Hamann, James Lacey, Alan Weger, and Jamil Wakil. Spatially-resolved imaging of microprocessor power (simp): hotspots in microprocessors. In *Thermal and Thermomechanical Phenomena in Electronics Systems, 2006. ITHERM'06. The Tenth Intersociety Conference on*, pages 5–pp. IEEE, 2006.

[25] Francisco Javier Mesa-Martinez, Joseph Nayfach-Battilana, and Jose Renau. Power model validation through thermal measurements. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 302–311. ACM, 2007.

[26] Dongkeun Oh, Nam Sung Kim, Charlie Chung Ping Chen, Azadeh Davoodi, and Yu Hen Hu. Runtime temperature based power estimation for optimizing throughput of thermal-constrained multicore processors. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, pages 593–599. IEEE Press, 2010.

[27] Xi Wang, Sina Farsiu, Peyman Milanfar, and Ali Shakouri. Power trace: An efficient method for extracting the power dissipation profile in an ic chip from its temperature map. *Components and Packaging Technologies, IEEE*

Transactions on, 32(2):309–316, 2009.

[28] Zhenyu Qi, Brett H Meyer, Wei Huang, Robert J Ribando, Kevin Skadron, and Mircea R Stan. Temperature-to-power mapping. In *Computer Design (ICCD), 2010 IEEE International Conference on*, pages 384–389. IEEE, 2010.

[29] Amit Sinha and Anantha P Chandrakasan. Jouletrack: a web based tool for software energy profiling. In *Proceedings of the 38th annual Design Automation Conference*, pages 220–225. ACM, 2001.

[30] Carlo Brandolese, William Fornaciari, Fabio Salice, and Donatella Sciuto. Timing and energy estimation of c programs. *Special issue on Power Aware Embedded Computing*, 1, 2010.

[31] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Power analysis of embedded software: a first step towards software power minimization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):437–445, 1994.

[32] Hztzefa Mehta, Robert Michael Owens, and Mary Jane Irwin. Instruction level power profiling. In *Acoustics, Speech, and Signal Processing, 1996. ICASSP-96. Conference Proceedings., 1996 IEEE International Conference on*, volume 6, pages 3326–3329. IEEE, 1996.

[33] Davide Sarta, Dario Trifone, and Giuseppe Ascia. A data dependent approach to instruction level power estimation. In *Low-Power Design, 1999. Proceedings. IEEE Alessandro Volta Memorial Workshop on*, pages 182–190. IEEE, 1999.

[34] Manuel Wendt, Matthias Grumer, Christian Steger, Reinhold Weiss, Ulrich Neffe, and Andreas Muehlberger. Tool for automated instruction set

characterization for software power estimation. *Instrumentation and Measurement, IEEE Transactions on*, 59(1):84–91, 2010.

[35] Gang Qu, Naoyuki Kawabe, Kimiyoshi Usami, and Miodrag Potkonjak. Function-level power estimation methodology for microprocessors. In *Proceedings of the 37th Annual Design Automation Conference*, pages 810–813. ACM, 2000.

[36] Tat Kee Tan, Anand Raghunathan, Ganesh Lakshminarayana, and Niraj K Jha. High-level software energy macro-modeling. In *Design Automation Conference, 2001. Proceedings*, pages 605–610. IEEE, 2001.

[37] Anish Muttreja, Anand Raghunathan, Srivaths Ravi, and Niraj K Jha. Automated energy/performance macromodeling of embedded software. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(3):542–552, 2007.

[38] Eric Senn, Johann Laurent, Nathalie Julien, and Eric Martin. Softexplorer: Estimating and optimizing the power and energy consumption of a c program for dsp applications. *EURASIP Journal on Applied Signal Processing*, 2005:2641–2654, 2005.

[39] Ismail Kadayif, M Kandemir, Guilin Chen, Narayanan Vijaykrishnan, Mary Jane Irwin, and Anand Sivasubramaniam. Compiler-directed high-level energy estimation and optimization. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(4):819–850, 2005.

[40] Holger Blume, Daniel Becker, Lisa Rotenberg, Martin Botteck, Jörg Brakensiek, and Tobias G Noll. Hybrid functional-and instruction-level power modeling for embedded and heterogeneous processor architectures. *Journal of Systems Architecture*, 53(10):689–702, 2007.

[41] Nicolas Fournel, Antoine Fraboulet, and Paul Feautrier. Embedded soft-

ware energy characterization: Using non-intrusive measures for application source code annotation. *Journal of Embedded Computing*, 3(3):10, 2009.

[42] Mariagiovanna Sami, Donatella Sciuto, Cristina Silvano, and Vittorio Zaccaria. An instruction-level energy model for embedded VLIW architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 21(9):998–1010, 2002.

[43] Luca Benini, Davide Bruni, Mauro Chinosi, Christina Silvano, Vittorio Zaccaria, and Roberto Zafalon. A power modeling and estimation framework for VLIW-based embedded systems. In *Proc. Int. Workshop on Power And Timing Modeling, Optimization and Simulation PATMOS*, volume 1, pages 2–3. Citeseer, 2001.

[44] Vivek Tiwari, Sharad Malik, Andrew Wolfe, and Mike Tien-Chien Lee. Instruction level power analysis and optimization of software. In *Technologies for wireless computing*, pages 139–154. Springer, 1996.

[45] Jeffry T Russell and Margarida F Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. In *Computer Design: VLSI in Computers and Processors, 1998. ICCD'98. Proceedings. International Conference on*, pages 328–333. IEEE, 1998.

[46] Sandro Penolazzi, Luca Bolognino, and Ahmed Hemani. Energy and performance model of a sparc leon3 processor. In *Digital System Design, Architectures, Methods and Tools, 2009. DSD'09. 12th Euromicro Conference on*, pages 651–656. IEEE, 2009.

[47] Young-Hwan Park, Sudeep Pasricha, Fadi J Kurdahi, and Nikil Dutt. A multi-granularity power modeling methodology for embedded processors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19(4):668–681, 2011.

[48] Gang Qu, Naoyuki Kawabe, Kimiyoshi Usami, and Miodrag Potkonjak. Code coverage-based power estimation techniques for microprocessors. *Journal of Circuits, Systems, and Computers*, 11(05):557–574, 2002.

[49] Thomas Ball and James R Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57. IEEE Computer Society, 1996.

[50] Eric Senn, Johann Laurent, Nathalie Julien, and Eric Martin. Softexplorer: estimation, characterization, and optimization of the power and energy consumption at the algorithmic level. In *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, pages 342–351. Springer, 2004.

[51] Johann Laurent, Nathalie Julien, Eric Senn, and Eric Martin. Functional level power analysis: An efficient approach for modeling the power consumption of complex processors. In *Proceedings of the conference on Design, automation and test in Europe-Volume 1*, page 10666. IEEE Computer Society, 2004.

[52] Nathalie Julien, Johann Laurent, Eric Senn, and Eric Martin. Power consumption modeling and characterization of the ti c6201. *IEEE Micro*, (5):40–49, 2003.

[53] Ismail Kadayif, M Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and Anand Sivasubramaniam. EAC: A compiler framework for high-level energy estimation and optimization. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 436–442. IEEE, 2002.

[54] M Schneider, H Blume, and TG Noll. Power estimation on functional level for programmable processors. *Advances in Radio Science*, 2(8):215–219,

2005.

[55] Wu Ye, Narayanan Vijaykrishnan, Mahmut Kandemir, and Mary Jane Irwin. The design and use of simplepower: a cycle-accurate energy estimation tool. In *Proceedings of the 37th Annual Design Automation Conference*, pages 340–345. ACM, 2000.

[56] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Springer Science & Business Media, 2002.

[57] Hugo Lebreton and Pascal Vivet. Power modeling in systemc at transaction level, application to a dvfs architecture. In *Symposium on VLSI, 2008. ISVLSI'08. IEEE Computer Society Annual*, pages 463–466. IEEE, 2008.

[58] Marco Giammarini, Massimo Conti, and Simone Orcioni. System-level energy estimation with powersim. In *2011 18th IEEE International Conference on Electronics, Circuits, and Systems*, 2011.

[59] Felipe Klein, Rodolfo Azevedo, Luiz Santos, and Guido Araujo. Systemc-based power evaluation with powersc. In *Electronic System Level Design*, pages 129–144. Springer, 2011.

[60] L Pieralisi, M Caldari, GB Vece, M Conti, S Orcioni, and C Turchetti. Power analysis methodology and library in systemc. In *Microtechnologies for the New Millennium 2005*, pages 446–455. International Society for Optics and Photonics, 2005.

[61] Daniel Lohmann, Olaf Spinczyk, and A Gal. Aspect-oriented programming with c++ and aspectc++. In *Tutorial held during the AOSD 2004 conference (Lancaster, UK)*, 2004.

[62] Matthias Kuehnle, Andre Wagner, and Juergen Becker. A statistical power estimation methodology embedded in a systemc code translator. In *Pro-*

ceedings of the 24th symposium on Integrated circuits and systems design, pages 79–84. ACM, 2011.

[63] Sandro Penolazzi, Ahmed Hemani, and Luca Bolognino. A general approach to high-level energy and performance estimation in socs. In *2009 22nd International Conference on VLSI Design*, pages 200–205. IEEE, 2009.

[64] Sandro Penolazzi, Ingo Sander, and Ahmed Hemani. Predicting energy and performance overhead of real-time operating systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 15–20. IEEE, 2010.

[65] Sandro Penolazzi, Ahmed Hemani, and Luca Bolognino. A general approach to high-level energy and performance estimation in system-on-chip architectures. *Journal of Low Power Electronics*, 5(3):373–384, 2009.

[66] Sandro Penolazzi. A system-level framework for energy and performance estimation in system-on-chip architectures. 2011.

[67] W Lloyd Bircher, Madhavi Valluri, Jason Law, and Lizy K John. Runtime identification of microprocessor energy saving opportunities. In *Low Power Electronics and Design, 2005. ISLPED'05. Proceedings of the 2005 International Symposium on*, pages 275–280. IEEE, 2005.

[68] W Lloyd Bircher and Lizy K John. Complete system power estimation: A trickle-down approach based on performance events. In *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 158–168. IEEE, 2007.

[69] Jiaoyan Chen, Dilip Vasudevan, Michel Schellekens, and Emanuel Popovici. Ultra low power asynchronous charge sharing logic. *Journal of Low Power Electronics*, 8(4):526–534, 2012.

[70] Marc Renaudin. Asynchronous circuits and systems: a promising design alternative. *Microelectronic engineering*, 54(1):133–149, 2000.

[71] Miloš Krstić, Eckhard Grass, Frank K Gürkaynak, and Pascal Vivet. Globally asynchronous, locally synchronous circuits: Overview and outlook. *IEEE Design & Test of Computers*, (5):430–441, 2007.

[72] Ahmed Hemani, Thomas Meincke, Sudhakar Kumar, Adam Postula, et al. Lowering power consumption in clock by using globally asynchronous locally synchronous design style. In *Design Automation Conference, 1999. Proceedings. 36th*, pages 873–878. IEEE, 1999.

[73] Alain J Martin. The design of an asynchronous microprocessor. 1989.

[74] Vivek Tiwari, Deo Singh, Suresh Rajgopal, Gaurav Mehta, Rakesh Patel, and Franklin Baez. Reducing power in high-performance microprocessors. In *Proceedings of the 35th annual Design Automation Conference*, pages 732–737. ACM, 1998.

[75] Silvia Annaratone. *Digital CMOS circuit design*, volume 16. Springer Science & Business Media, 2012.

[76] John P Uyemura. Cmos dynamic logic families. *CMOS Logic Circuit Design*, pages 349–434, 2001.

[77] Avantika Singh, Ankur Jaiswal, and Prof Shahiruddin. Switching techniques: Concepts for low loss switching. *International Journal of Scientific Engineering and Technology*, pages 905–908, 2013.

[78] S Natarajan. Soi design: analog, memory and digital techniques, 2001.

[79] A Albert Raj and T Latha. *VLSI design*. PHI Learning Pvt. Ltd., 2008.

[80] Giby Samson. *Robust dynamic circuits with low power and high performance for nanometer CMOS technologies*. ProQuest, 2008.

[81] William J Dally and John W Poulton. *Digital systems engineering.* Cambridge university press, 1998.

[82] Pius Ng, Poras T Balsara, and Don Steiss. Performance of cmos differential circuits. *Solid-State Circuits, IEEE Journal of*, 31(6):841–846, 1996.

[83] Kris Tiri, Moonmoon Akmal, and Ingrid Verbauwhede. A dynamic and differential cmos logic with signal independent power consumption to withstand differential power analysis on smart cards. In *Solid-State Circuits Conference, 2002. ESSCIRC 2002. Proceedings of the 28th European*, pages 403–406. IEEE, 2002.

[84] Benjamin Gojman. Adiabatic logic. *CalTech University, CA*, 8, 2004.

[85] Muhammad Arsalan and Maitham Shams. Charge-recovery power clock generators for adiabatic logic circuits. In *null*, pages 171–174. IEEE, 2005.

[86] Shunji Nakata. Adiabatic charging reversible logic using a switched capacitor regenerator. *IEICE transactions on electronics*, 87(11):1837–1846, 2004.

[87] A Vetuli, Stefano Di Pascoli, and LM Reyneri. Positive feedback in adiabatic logic. *Electronics Letters*, 32(20):1867–1869, 1996.

[88] Alan Kramer, John S Denker, B Flower, and J Moroney. 2nd order adiabatic computation with 2n-2p and 2n-2n2p logic circuits. In *Proceedings of the 1995 international symposium on Low power design*, pages 191–196. ACM, 1995.

[89] Yong Moon and Deog-Kyoon Jeong. An efficient charge recovery logic circuit. *Solid-State Circuits, IEEE Journal of*, 31(4):514–522, 1996.

[90] Philip Teichmann. *Adiabatic logic: future trend and system level perspective*, volume 34. Springer Science & Business Media, 2011.

[91] Ettore Amirante, Agnese Bargagli-Stoffi, Jurgen Fischer, Giuseppe Iannaccone, and Doris Schmitt-Landsiedel. Variations of the power dissipation in adiabatic logic gates. In *Proceedings of the 11th International Workshop on Power And Timing Modeling, Optimization and Simulation, PATMOS*, volume 1, pages 9–1, 2001.

[92] Jiaoyan Chen. *Low power predictable memory and processing architectures.* PhD thesis, University College Cork, 2013.

[93] Jiaoyan Chen, Arnaud Tisserand, Emanuel Popovici, and Sorin Cotofana. Asynchronous charge sharing power consistent montgomery multiplier. In *Asynchronous Circuits and Systems (ASYNC), 2015 21st IEEE International Symposium on*, pages 132–138. IEEE, 2015.

[94] Intel Corporation. *MCS-51 8-bit Control-Oriented Microcomputers*, 1988.

[95] Atmel Corporation. *Atmel 8051 Microcontrollers Hardware Manual*, 2007.

[96] Arm Ltd. 8051 instruction set manual. `http://www.keil.com/support/man/docs/is51/`. Accessed on 9 May 2015.

[97] S. Teran J. Simsic. 8051 core. `http://opencores.org/project,8051`. Accessed on 9 May 2015.

[98] John Wharton. An introduction to the intel-mcs-51 single-chip microcomputer family. *Intel Corporation*, 1980.

[99] A jit Pal. *Microcontrollers: Principles and Applications.* Prentice-Hall of India Pvt.Ltd, Delhi, India, 2011.

[100] Behrooz Parhami. Computer arithmetic: algorithms and hardware designs. 1999.

[101] Joseph J. F. Cavanagh. *Digital Computer Arithmetic.* McGraw-Hill, Inc., New York, NY, USA, 1983.

[102] DA Godse and AP Godse. *Microprocessors, Microcontrollers and Applications.* Technical publications Pune, third revised edition, 2008.

[103] Alka Kalra and Sanjeev Kumar Kalra. *Architecture and Programming of 8051 Microcontroller.* University Science Press, first edition, 2010.

[104] Synopsys Inc. HSIM. `http://www.synopsys.com/Tools/Verification/AMSVerification/CircuitSimulation/HSIM/Pages/default.aspx`. Accessed on 8 July 2015.

[105] Synopsys Inc. HSIMplus HDL Co-Simulation. `http://www.synopsys.com/Tools/Verification/ AMSVerification/DesignAnalysis/Pages/HDLCo-Simulation.aspx`. Accessed on 8 July 2015.

[106] Bing J Sheu, Donald L Scharfetter, P-K Ko, and Min-Chie Jeng. Bsim: Berkeley short-channel igfet model for mos transistors. *Solid-State Circuits, IEEE Journal of,* 22(4):558–566, 1987.

[107] The free encyclopedia Wikipedia. Bsim. `http://en.wikipedia.org/wiki/BSIM`. Accessed on 25 May 2015.

[108] The UCR Dalton Project. Intel 8051 simulator. `http://www.cs.ucr.edu/~dalton/i8051/i8051sim/`. Accessed on 13 June 2015.

[109] Shih-Yi Yuan, Huai-En Chung, and Shry-Sann Liao. A microcontroller instruction set simulator for emi prediction. *Electromagnetic Compatibility, IEEE Transactions on,* 51(3):692–699, 2009.

[110] Gunnar Braun, Achim Nohl, Andreas Hoffmann, Oliver Schliebusch, Rainer Leupers, and Heinrich Meyr. A universal technique for fast and flexible instruction-set architecture simulation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on,* 23(12):1625–1639, 2004.

[111] SDCC - Small Device C Compiler. `http://sdcc.sourceforge.net/`. Accessed on 14 June 2015.

[112] Jaka Simsic and Simon Teran. oc8051 Design Document. `http://opencores.org/project,8051`, 2002. Accessed on 20 June 2015.

[113] M Boubekeur, D Hickey, and M Schellekens. Evaluation of MOQA average-case timing results on a real time platform. In *Proc., Conf. Information of MFCSIT*, 2006.

[114] Michel Schellekens. *A Modular Calculus for the Average Cost of Data Structuring.* Springer, 2008. ISBN 9780-387-73383-8.

[115] Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981.

[116] Michel P Schellekens. MOQA; unlocking the potential of compositional static average-case analysis. *The Journal of Logic and Algebraic Programming*, 79(1):61–83, 2010.

[117] M Schellekens, D Hickey, and G Bollella. MOQA, a linearly-compositional programming language for (semi-) automated average-case analysis. In *WIP Proceedings, 25th IEEE International Real-Time Systems Symposium, Lisbon, Portugal*, 2004.

[118] Christian Bunse, Hagen Hopfner, Essam Mansour, and Suman Roychoudhury. Exploring the energy consumption of data sorting algorithms in embedded and mobile environments. In *Mobile Data Management: Systems, Services and Middleware, 2009. MDM'09. Tenth International Conference on*, pages 600–607. IEEE, 2009.

[119] Wikipedia. Insertion sort. `https://en.wikipedia.org/wiki/Insertion_sort`. Accessed on 18 July 2015.

[120] Prabhakar Gupta and Manish Varshney. *Design and Analysis of Algorithms.* PHI Learning Pvt. Ltd., second edition, 2012.

[121] SPARC International Inc. *The SPARC Architecture Manual, Version 8*, 1992.

[122] Aeroflex Gaisler. `http://www.gaisler.com/`. Accessed on 15 July 2015.

[123] GRLIB IP Library. `http://www.gaisler.com/index.php/products/-ipcores/soclibrary`. Accessed on 15 July 2015.

[124] LEON3 Processor. `www.gaisler.com/index.php/products/processors/leon3`. Accessed on 16 July 2015.

[125] Aeroflex Gaisler. *GRLIB IP Library User's Manual, Version 1.1.0 B4108*, June 2001.

[126] Shakeel Sultan and Shahid Masud. Rapid software power estimation of embedded pipelined processor through instruction level power model. In *Performance Evaluation of Computer & Telecommunication Systems, 2009. SPECTS 2009. International Symposium on*, volume 41, pages 27–34. IEEE, 2009.

[127] Gaisler Reseach. *The LEON Processor User's Manual, Version 2.3.5*, July 2001.

[128] Gaisler Reseach. *LEON3 GR-XC3S-1500 Template Design, Based on GRLIB-1.0.7*, February 2006.

[129] Aeroflex Gaisler. *BCC - Bare-C Cross-Compiler User's Manual, Version 1.0.43*, June 2013.

[130] Free Software Foundation Inc. GCC, the GNU Compiler Collection. `https://gcc.gnu.org/`. Accessed on 22 July 2015.

[131] Corinna Vinschen and Jeff Johnston. Red Hat Newlib C Library. `https://sourceware.org/newlib/`. Accessed on 22 July 2015.

[132] Agner Fog. Pseudo random number generators, uniform and non-uniform distributions. `http://www.agner.org/random/`. Accessed on 25 July 2015.

[133] Agner Fog. Technical University of Denmark. *Instructions for the random number generator libraries on www.agner.org, Version 2.11*, June 2014.

[134] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[135] Archana Jagannatam. Mersenne twister–a pseudo random number generator and its variants. 2008.

[136] Wikipedia. SREC (file format). `https://en.wikipedia.org/wiki/SREC_file_format`. Accessed on 22 July 2015.

[137] Open Verilog International. *Standard Delay Format Specification, Version 3.0*, May 1995.

[138] Himanshu Bhatnagar. *Advanced ASIC Chip Synthesis: Using Synopsys® Design CompilerTM Physical CompilerTM and PrimeTime®*. Springer Science & Business Media, 2007.

[139] The Institute of Electrical and Electronics Engineers, Inc. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*, December 1995.

[140] John K Ousterhout and Ken Jones. *Tcl and the Tk toolkit*. Pearson Education, 2009.