Consorzio per la Formazione e la Ricerca
in Ingegneria dell'Informazione

Politecnico di Milano

XIV Master in Tecnologia dell'Informazione

Rapporto finale

A model for Assembly Instruction Timing and Power Estimation on Superscalar Architectures

Autore: *Giovanni Beltrame*
Tutor: *Carlo Brandolese*
Area: *ESD*
Sponsor: *POET Project*

1 luglio 2002
Versione: *1.4*
Stato: *release*

Diffusione del documento

Revisioni

Data	Versione	Stato	Commento
3 marzo 2002	0.1	draft	Prima stesura
6 marzo 2002	0.2	draft	Corretti e aggiornati i capitoli 1, 2, 3, 5
8 marzo 2002	0.3	draft	Aggiunta del capitolo 4
11 marzo 2002	0.4	RC1	Correzioni e bugfix, aggiunta appendice
12 giugno 2002	1.1	draft	Prima versione del rapporto finale, aggiunta del capitolo 7
22 giugno 2002	1.3	RC2	Aggiornamento risultati e correzioni
1 luglio 2002	1.4	release	Correzioni stilistiche minori

Table of Contents

1 Introduction

Today we are facing an increasing market for embedded applications: therefore there is a stronger need to obtain design solutions that would concurrently meet time-to-market, low cost and flexibility. In order to obtain such solutions, the increasing relevance of power consumption in modern embedded applications demands tools to predict with reasonable confidence the power consumption both for hardware and software. While in literature the power estimation for hardware is well established, methodologies for software still require some further insight.

The aim of this work is to provide a simple and accurate static time and energy consumption model for micro-processor instruction sets. In particular, the novelty of this works stand in the introduction of the support of execution parallelism in the temporal and energy estimation process.

Most of today's embedded devices are heterogeneous systems composed of dedicated ICs, memories, A/D and D/A converters, sensors and microprocessor cores. In the past decade, a considerable research effort has been made to study the power behavior of digital systems and of specific components such as memories, leading to a firm ground on which EDA vendors have developed efficient and reliable tools. With the increasing request of complex functionalities, the amount of software composing embedded systems has rapidly grown, making the microprocessor and the software itself highly critical portions of the design.

Optimal embedded power savings result from an integrated hardware-software design methodology that focuses on the power dissipation problem, starting from the very early phases of the design process. One of the primary power-saving techniques to investigate is a lower supply voltage. Halving the supply voltage reduces static power dissipation to one fourth. However, a reduced supply voltage, along with today's fast clocks, produce noise-immunity problems. Another possible problem with a lower supply voltage is the limited availability of logic functions, that are usually essential to determine the success of a product.

Processor selection is the first step for low power design. A possible approach might be to select fast and powerful processors, relying on plenty of computing power in order to run complex programs. A wiser solution would be to select a lower speed processor with just enough capability for the application. The practical answer lies somewhere between these two approaches and depends on the power-management flexibility of the chosen processor.

System power consumption also directly relates to the processor clock speed. Many microprocessors support a variable-speed clock, allowing designers to adjust the frequency for optimal power savings and to retain the capability for extra speed when an application requires it. The program can increase the CPU clock speed when processing demands are high and then go back to a lower speed for non-critical tasks. This type of dynamic control is quite effective in battery-operated systems.

Intel, Microsoft, and Toshiba America Information Systems introduced an Advanced Configuration and Power Interface (ACPI) for desktop and notebook PCs in 1997. ACPI transfers the responsibility for power management from the firmware to the operating system. ACPI defines a series of reduced-power states for the system, processor, and peripheral devices. When the system has been idle for a specified amount of time, the software enters system power-management states, also called sleep states. The CPU does no work in any of the sleep states. The ACPI specification defines four levels of sleep states, each with increased power savings but requiring more time to resume.

Although ACPI targets desktops and notebooks, it might be an interesting model to follow when developing a power-management software for embedded systems. Not all applications, though, can afford the overhead of such a software. Often, the only way to optimize software power dissipation is to measure the energy consumption in real time as the application program executes in the embedded hardware platform. Then, a designer can either rewrite or optimize the most power-demanding routines or implement them in dedicated hardware. This approach falls in the class of the co-design methodologies, i.e. the concurrent design of both the hardware components and the software functions. In such a framework it is essential to delay as much as possible all the decisions concerning the target technology in order to evaluate many different alternatives.

The work presented here analyses the power related issues that arise when designing heterogeneous embedded systems concurrently. In particular it focuses on questions related to software power consumption evaluation, at the highest granularity level. The system to be modeled is usually described by means of a high-level language that captures the functional behavior while neglecting all the implementations and technology-dependent characteristics. A modification to the system when the different components have already been committed to the target technology may have a significant impact on the design times and costs. Furthermore, the sooner optimizations are performed, the more their effect is valuable. A simplistic explanation of this well-known fact is that at the beginning of the design cycle, all degrees of freedom can be exploited, while at later phases, when many decisions have already been taken, only a few alternatives are available to designers, possibly leading to modest enhancements.

Two phases are crucial, and relatively unexplored, in the co-design flow: system partitioning and model verification. Partitioning is the task of assigning each of the functionalities or modules to a specific partition, i.e. hardware or software. The assignment of functions to partitions is a NP-complete problem and is thus often solved heuristically. The three ingredients for a good solution are an efficient algorithm, a well-structured cost function and accurate models to estimate the characteristics of the different modules.

While a number of extremely efficient algorithms have been proposed in literature and a solid background exists about cost-functions, estimation models are still a big issue. This is basically due to the fact that an estimation model, to be considered acceptable and to be adopted, must meet some criteria such as accuracy, stability and computational efficiency. Most of all, a viable estimation technique must operate on a high-level description of the system so that no implementation decision needs to be committed prior to partitioning.

The second critical point in any hardware/software co-design flow is model verification. This can be done with two main purposes: functional verification and timing, or power, verification. In the former case a simulator for the high-level language is sufficient. In the latter case, however, simulation alone is not enough since execution times, or power consumptions, should also be checked. When simulating heterogeneous systems the main issue to be dealt with is synchronization. This implies the notion of a global, real time in which the system evolves according to the execution times of the composing processes or instructions. To determine the time duration of each process, again, an accurate estimation technique is necessary. Static figures, computed off-line, are then combined, during simulation, with the dynamical aspects of execution. Neglecting the dynamic effects, the basis for validation also is constituted by a static model of the power consumption.

This work considers the highest level of granularity of the co-design flow, and focuses in particular on power consumption and timing estimations. In this work previous models are extended in order to consider parallel execution of instruction, which is a typical characteristic of current superscalar microprocessors. The proposed model is founded on a theoretical analysis of the estimation problem and on the experimental results that have been obtained by using a set of tools, developed on purpose.

The report is organized as follows: chapter 2 summarizes some of the most interesting results obtained in the past few years on hardware and software power estimation. In chapter 3 a mathematical model for inter-instruction effects and parallel execution issues is introduced: this model is used for the statistical analysis and prediction of execution overheads and of parallel execution factors that influence the timing estimation. Chapter 4 introduces and details the application methodology associated with the model and chapter 5 the software tools developed on purpose to apply such methodology. Finally, chapter 8 describes the experimental setup, and the results of the validation process are presented.

2 Background

The increased use of portable applications has placed severe limitations on the power consumed by processors and systems. Energy efficient designs are now just as important as fast and high performance ones. Designing systems to operate longer on a single battery charge is an important consideration in the design of today's portable systems. However, the electronic integration technology cannot help much in power saving issues, while code and architectural optimization can significantly reduce power consumption. Therefore, many researchers have considered power minimization through the modification of the architecture, the high level software and the algorithms. This however will be more effective if a realistic power model for a microprocessor core, its instruction set and the various types of memory accesses were developed. In the following, Section 2.1 briefly describes a typical system design flow; Sections 2.2 describes the main classes of power estimation approaches, working at hardware and software level. Finally, Section 2.3 will detail a different approach: the instruction-level power modeling, which takes into account also architectural issues.

2.1 System Design

A typical hardware design flow is structured in a number of steps, each giving a different view of the system. The views differ with respect to two main aspects: the description language or formalism and the level of detail. The following scheme summarizes the foremost characteristics of the different views and gives an outline of a typical industrial design flow.

Architectural or System level. At this level the system is represented as an abstract network of interconnected functionalities. The functionalities are typically modeled as black-boxes whose interface only is known. This representation captures in synthetic and compact way the behavior of the system but does not give any detail on the internal implementation. A rather wide spectrum of formalisms is used to capture such a description of a system: graphical models [9, 23] (often invented ad-hoc by CAD or EDA vendors), different flavors of Petri Nets [15], CSP (Communicating Sequential Processes), State Charts [32, 31], or extension to the C/C++ languages such as Hardware C [14] and SystemC [1].

Behavioral level. The architectural level description is converted into a functionally equivalent behavioral description. This process is currently performed manually by the design team, in some cases with the support of commercial co-design tools. The behavioral view adds an algorithmic description of the functionalities of the system. The language used to describe a system at this level is typically VHDL, but recently SystemC is gaining more and more popularity. Although such a description gives much more details on the internal structure of the functional blocks of the system, the notions of time, i.e. clock, and availability of hardware resources are not present yet.

Register-transfer level. From the behavioral description of system and a set of constraints, a register-transfer model can be derived. The constraints typically specify the timing requirements and the resources availability. Both are used to drive and control automatic tools (behavioral synthesis tools such as Behavioral Compiler by Synopsys). These tools are cutting-edge technologies and often still require the human intervention. The result of behavioral compilation is a description of the system in terms of purely combinational logic and registers. The languages used are either a subset of the VHDL, usually referred to as RTL-VHDL, or Verilog. At this level of abstraction, data computation is expressed by means of high-level operators such as adders, multipliers, multiplexers and many others operating on compound data types (buses, records, etc.). Registers are described using specific language templates. The clock signal is introduced explicitly and hardware resources are allocated and bound to the symbolic operators according to the specified constraints. The recently introduced SystemC version 1.1 can describe systems at RT level. The register-transfer view, though sufficiently detailed, still ignores the internal structure of the operators.

Gate level. The register-transfer description is then translated into a gate level model by means of automatic logic synthesis tools (such as Design Compiler and FPGA Express by Synopsys or Galileo and Leonardo by Exemplar Logic). The result is a netlist where operators are expanded into logic gates and compound signals are substituted by sets of single nets. Subsets of the VHDL and Verilog languages, referred to as structural VHDL or Verilog, are often used to describe netlists. Other standard formats such as EDIF (Electronic Design Interchange Format), XNF (Xilinx Netlist Format), blif (Berkeley Logic Interchange Format) are also used. The gates used in the netlist may either be taken from a fictitious technology library or from a commercial library. In the latter case, each component is characterized in terms of delay and capacitance and this allows rather accurate estimates to be performed. Nevertheless, these models are lumped and the simulation is in the discrete time and considers logical values for the signals rather than actual voltage levels. Furthermore, interconnections are characterized using statistical wire-load models since no information is available on the relative geometrical positioning of the gates.

Transistor level. The gate-level netlist is usually given to a silicon foundry where the last steps of the design flow are performed. In particular, each gate or basic component is substituted with its corresponding transistor-level circuit, according to the specific library and technology used. The resulting representation is extremely accurate and details both the actual layout of the cells and the position and length of the interconnections. Combining geometrical information with physical details of the cells and the nets an accurate model can be derived and simulated to obtain a precise characterization, in terms of timing and power consumption.

These steps are typical of most design flows and refer to the realization of application-specific integrated circuits (ASICs) as well as microprocessor and microcontroller cores.

2.1.1 Co-Design

Concurrent development of hardware and software is progressively displacing the traditional sequential design. It is becoming common practice to begin the hardware and software design before the system architecture is finalized.

In the design practice, system architects define an architecture consisting of cooperating hardware and software *functions* that form the basis for the actual components design. One major problem of this approach is the definition, design and synthesis of the interfaces, that usually requires a tight cooperation of distinct design groups. Another drawback of such a design paradigm is that a change in the requirements implies a modification in the overall architecture that is often driven by a cost prediction (in terms of area, timing or power) done on the basis of the expertise of system architects.

Another big issues that must be addressed when considering separate design flows for hardware and software is the problem of maintaining permanent control over consistency and correctness. This problem becomes more complex with increasing levels of detail.

This brief outline of the design scenario of typical mixed hardware and software embedded systems highlights the need of a unified approach to the problem.

The co-design flow can be thought of as a sequence of the following steps:

Specification. The requirements are translated from an informal language into a formal description of the functionalities. This step should be independent of the target architecture that will be chosen.

Simulation. The formal specification is simulated to verify its functional correctness. It is often required to have a very detailed view of the system and thus accurate models are necessary.

Partitioning. The functionalities composing the system are partitioned, i.e. assigned either to the software or hardware options. In the current practice this step is performed manually by expert designers or system architects.

Synthesis. The hardware and software specifications are translated into their final form, typically a technology netlist for the hardware and an assembly code for the software. These tasks are usually performed using third party commercial tools.

Verification. The low-level models are simulated with a higher level of detail. At this stage, area, time and power figures are known and can be used to derive the exact characteristics of the complete system.

For a number of reasons, such as changes in the requirements or wrong choices performed during the partitioning phase, it is often necessary to evaluate different partitioning alternatives. To limit the design turn-around time it is thus essential to provide a framework that facilitates moving portions of the design from a partition to another. This, in turns, calls for accurate estimation metrics that allow designers to repeat the actual synthesis and verification steps.

2.2 Power Estimation Techniques

The assignment of functions to partitions is a NP-complete problem and is thus often solved heuristically. The three ingredients for a good solution are an efficient algorithm, a well-structured cost function and accurate models to estimate the characteristics of the different modules.

While a number of extremely efficient algorithms have been proposed in literature and a solid background exists about cost-functions, estimation models are still a big issue. Statistical power models have been proposed: they are simulation-based and the activity factors are computed over typical input streams.

Different levels of abstraction can be adopted in estimating the power consumption of a given design; these are described in the following subsections.

2.2.1 Transistor-Level Estimation

It is based on the representation of a microprocessor in terms of transistors and nets: this representation is extremely complex and rarely feasible. Furthermore, a transistor-level view of the system uses components models based on linearized differential equations and works in the continuous-time domain[1]. This implies that a simulation of more than one million transistors, even for few clock cycles, requires times that are usually not affordable and anyway not practical for the high-level power characterization. Nevertheless these techniques are extremely valuable as a replacement for physical measurement. Measuring the power consumption of a microprocessor, in fact, requires sophisticated and costly instruments and an electrical modification of the board hosting the microprocessor core in order to have access to the power supply pins. Both these problems often prevent any measurement to be actually performed.

2.2.2 Gate-Level Estimation

Methods to estimate the power consumption based on gate-level descriptions of microprocessors or micro-controller cores have been proposed in literature. The main advantage of such methods

[1]The time is actually discrete due to the finite precision of the computer representation of floating point numbers.

with respect to transistor-level simulation approaches is that the simulation is event-driven and takes place in a discrete-time domain, leading to a considerable reduction of the computational complexity, without a significant loss of accuracy. The main shortcoming of such an approach lays in the computational requirements of gate-level simulation. To overcome effectiveness limitations, solutions based on a statistical analysis of the design properties have been proposed in literature. These approaches can be classified in two main groups: static methods and dynamic methods. To the former class belong the techniques presented in [22], [21] and [11]. These methods rely on statistical information (such as the mean activity of the input signals and their correlations) about the input stream to estimate the switching activity of the internal nodes of the circuit. The methods classified in the latter group are aimed at achieving a high level of accuracy at the cost of longer run-times, and are thus unpractical for the microprocessor power-characterization problem.

2.2.3 RT-level estimation

A design described at *Register Transfer* (RT) level can be seen as a collection of blocks and a network of interconnections. The blocks, sometimes referred to as macros, are adders, registers, multiplexers etc., while the interconnections are simply nets or group of nets. An assumption underlying the great majority of the approaches presented in literature is that the power properties of a block can be derived from an analysis of the block *isolated* from a design, under controlled operating conditions. The main factor influencing the power consumption model of a macro is the input statistic: if the probabilistic distribution of the inputs is a good approximation of the typical operating conditions of the block, then the power consumption is considered almost independent of the boundary (electrical) conditions. Under this assumption, common macros can be characterized.

To combine the information available for each block into a complete power model for a given design, two further issues must be addressed, two problems arise: the characterization of the interconnections and the higher-level probabilistic description of the system. The former problem derives from the gap between the degree of detail available at layout-level (exact position, shape and length of the wires) and the lack of predictability on the final geometry at RT level. Statistical models have been derived to fill this gap, the most popular being the so-called *wire-load models*, extensively used in commercial synthesis tools. The latter issue has been addressed from different sides and a number of approaches have been proposed in the scientific literature.

2.2.4 Behavioral-Level Estimation

A behavioral model is usually provided by means of a high-level hardware description language such as VHDL, Verilog or some flavor of state charts and then translated into an intermediate internal representation typically based on *Control Data Flow Graphs* (CDFGs), where operators are represented by the nodes of the graph while functional dependencies are rendered with arcs. The purpose of behavioral synthesis is to translate such a CDFG into a more detailed, lower-level model. This is done by mapping operators to the available hardware resources (allocation and binding) and by deciding the order of execution of the different operations (scheduling). Binding, allocation and scheduling algorithms are designed to minimize some sort of cost metric while respecting constraints imposed on other metrics. Behavioral power estimation, in this context, is used either as an additional metric to be possibly minimized or as a synthesis constraint.

When analyzing the power consumption of a complex system, the problem can be split in three sub-problems: first of all, for each unit time interval, determine which units of the system are active, on the basis of the assembly instruction being executed; then, determine a model for the power consumption of the finite state machines implementing the control unit; finally, determine a model for the power consumption of each unit of the system with a sufficiently fine grain and an acceptable accuracy. In this context, a functional unit is a sub-circuit performing a complete operation at a given level of abstraction. The level at which behavioral power estimation operates is usually that of *library macros* (i.e. units performing atomic operations), leaving to a higher level analysis methodology the task of identifying the interactions and activation intervals of the macros.

2.2.5 Architectural-level estimation

The most abstract representation of a complex system, such as a microprocessor, is that at the architectural level. Such a model gives a view composed of a number of interconnected functional blocks, each one devoted to a specific and, in a sense, atomic functionality. In particular, the simplest model of a microprocessor is structured in two interacting units: a control unit and a data-path. The data-path is constituted by a number of arithmetic operators having a rather regular structure and a variable sized register file. This fact allows the adoption of constructive approaches for the power prediction, based on the knowledge of the elementary components constituting the different units. It is also possible to abstract from the internal structure of the functional blocks and perform a black-box analysis based on the input and output switching activities. These activities can be either calculated or estimated using ad-hoc software simulators. The analysis of control units, i.e. complex finite state machines, is much more complex for a twofold reason: their logical structure, in fact, is not regular and hardly predictable, furthermore, their implementation can vary in a spectrum of solutions ranging from PLA solutions to completely random logic implementations.

2.3 Instruction-Level Power Estimation

The power estimation methods described in the preceding sections exhibit a number of problems related either to the lack of details (gate-level models) of the microprocessor or to the unpractical time requirements or to both. To overcome these problems, instruction-level measurement-based models have been proposed [29][28][26]. The key point lays in measuring the current drawn by the processor as it executes a long sequence of the same instruction and considering the average current absorbed as representative of such an instruction. This procedure has to be repeated for all instructions to completely characterize the microprocessor model. In this way, a table of the currents drawn by each instruction in the Instruction Set of a given processor is obtained, knowing *a-priori* how many cycles each instruction will take, in a sort of *stall-free* analysis. To these measured *base costs*, Malik et al. propose to add a measured stall cost and cache miss cost to each basic block of code. This *overhead* cost is experimentally measured for each type of stall, and the same activity has to be performed for cache misses. This methodology, although generally applicable to any processor, is not viable: the measures have to be taken for every processor, and the information obtained for one processor cannot be used for estimating values for other processors. In fact, to model an alternative CPU core, a new costly analysis of the entire instruction set has to be carried out. Furthermore, the confidence of the estimations is also seldom considered under a formal viewpoint: the statistical significance of the model of consumption is usually neither considered nor justified.

In 1998, Ramalingam and Schindler proposed an instruction level power model that considered dynamic effects [24]. Their model is based on [29][30] but obtains a more precise estimate for base costs. What the authors actually did is to separate instructions with the same opcode but different addressing modes and to add a statistical analysis of cache and pipeline interlock overheads. But this is anyway not general, in the sense that this methodology needs measures for every processor it has to be applied to.

Another approach has been introduced to overcome the above mentioned limitations, proposing a general methodology, independent of the specific processor, allowing to accurately estimate the energy of an instruction set. The methodology abstracts from the architectural level and focuses on the *functionalities* involved in instruction execution [3]. The resulting functional model exhibits generalization capabilities and allows covering a broad range of 32-bits microprocessors architectures. The energy consumption of each instruction is obtained as linear combination of independent contributions corresponding to a set of disjoint functionalities. The methodology allows an early *virtual* prototyping of the software-bound section of embedded applications on different target processors. In section 2.3.1 the first methodology model is described: it has to be considered a basis from which to start building a more general framework. The analysis of inter-instruction effects is here missing and have been introduced in [2]: the extended model refers in particular to pipelined architectures and does not takes into account the effects related to memory access. The present work represent a step towards the extension of this approach to a general framework that includes parallel instruction execution and memory effects analysis (see chapter 3).

2.3.1 The Interlock-Free Model

As mentioned above, the approach proposed in [3] abstracts from the architectural level by determining a set of *functionalities* and by decomposing the computational activity of each instruction in terms of these functionalities. The model provides a *static* estimation of the energy consumption of single instructions. According to [3], the energy dissipation e_s of an instruction s is evaluated as:

$$e_s = \sum_{j=0}^{5} e_{s,j} = \left[\sum_{j=0}^{5} \mathit{if}_j \cdot a_{s,j} \right] \cdot V_{dd} \cdot \tau \tag{2.1}$$

where if_j is the average current associated with the j-th functionality, V_{dd} is the power supply voltage, τ is the clock period and $a_{s,j}$ is a coefficient expressing the execution time spent by instruction s in the j-th functionality. The coefficients $a_{s,j}$ satisfy the following relation:

$$\sum_{j=0}^{5} a_{s,j} = \mathrm{CPI}_{s,\mathrm{nominal}} \tag{2.2}$$

stating that the time —expressed in clock cycles— spent by instruction s in all the functionalities corresponds to its average CPI (*Clock-cycles Per Instruction*) [12]. According to this model the energy absorbed by each instruction is computed as the weighted sum of the contributions of the functionalities. A tuning phase, based on a limited set of experimental data, allows associating to each functionality an average current absorption per clock cycle. It is worth noting that the overall energy consumption is strongly dependent on the number of cycles taken for the execution of assembly instructions. In [3] the timing is assumed to coincide with the nominal value reported in the processor data-sheets. This timing data, being purely static, is a sound starting point for a general energy model but disregards the delays introduced by the interlocks arising from a pipelined execution of the code.

2.3.2 The Interlock-Aware Model

The interlock-aware model, presented in [2], is capable of describing timing overheads due to inter-instruction effects in a formal and general way, thus addressing the limitation of the previous approach. The advantages of a *static* model with respect to a dynamic, simulation-based, approach are evident: model application is extremely fast, less complex and less memory intensive. This approach is based on a dynamic characterization —to be performed once and for all— of a given instruction set aimed at producing statically usable figures.

This model focuses on inter-instruction effects related to pipelined execution. In pipelined processors instructions are executed with partial time overlap in order to minimize the average CPI. However, this execution scheme leads to some *hazard* conditions that have to be suitably managed in order to maintain the semantics of the original program. In some cases, it is necessary to stall the pipeline, consequently increasing the nominal CPI. The introduced overhead brings to an increase of the energy consumption that cannot be ignored [24]. According to these observations, equation (2.1) can be extended by explicitly adding the overhead $oh_{s,j}$, yielding:

$$e_s = \left[\sum_{j=0}^{5} if_j \cdot (a_{s,j} + oh_{s,j}) \right] \cdot V_{dd} \cdot \tau \tag{2.3}$$

where $oh_{s,j}$ is a statistical coefficient expressing the execution time spent by instruction s in the j-th functionality in a stall situation. According to equation (2.3), the actual execution time of instruction s is:

$$\text{CPI}_{s,\text{est}} = \sum_{j=0}^{5} (a_{s,j} + oh_{s,j}) \tag{2.4}$$

Since inter-instruction effects such as pipeline interlocks and cache misses are intrinsically *dynamic events*, a purely static analysis would lead to an oversimplification of the problem. Nevertheless, a static analysis is still viable if a characterization of the dynamic effects is available. Such information can be extracted once for each microprocessor considered and stored in a library. This approach is thus based on a dynamic analysis of the whole instruction set aimed at a statistical characterization whose results, i.e. the $\text{CPI}_{s,\text{est}}$, can then be statically used for the estimation process.

The limitation of this approach becomes evident when parallel execution of instruction and memory related stalls have to be introduced. Most of the modern microprocessors implement a superscalar architecture, thus a model based on a single pipeline is not sufficient to capture the dynamics of such instruction execution. Starting from this observation, the present work extends the previous model in order to overcome this problem, as Chapter 3 details.

3 Model Definition

This chapter introduces the main issues related to the introduction of parallel execution in the methodology described in Section 2.3. Section 3.1 describes the problems related to the mathematical model adopted in the previous works and section 3.2 will extend the model in order to overcome these problems.

3.1 Problem Definition

The basic assumption made in [3] and maintained in [2] concerns the *a-priori* knowledge of instruction CPIs. Thus, the interlock-free timing of a code portion can be easily obtained by simply summing the CPIs of all the executed instructions. The interlock-aware timing can be obtained in a similar way: a statistical term representing the stall overhead associated to a single instruction is estimated (see Section 2.3.2) and the resulting CPIs can be finally summed up. Given an instruction s, its *static* stall overhead has to be obtained carefully averaging the contribution deriving from the *dynamic* interaction of s with all possible tuples of instructions, in order to obtain a reliable value. If parallel execution is considered, the actual CPI of an instruction is deeply influenced: if an instruction can be executed in parallel with other instructions, the global CPI decreases and consequently also the instruction actual CPI. In superscalar architectures, the parallel execution of assembly instructions strongly influences both the actual CPI of an instruction and the number and type of possible interlocks. For example, when the three instructions s_1, s_2 and s_3 are executed in an ideal pipeline the resulting CPI is 1.0 for all of them. In a superscalar architecture with three ideal pipelines in parallel, the resulting CPI would be $1/3$. However, real processors significantly differ from ideal architectures and only a portion of the theoretical parallelism can be exploited. To account for such deviation, a *parallelism coefficient* has been introduced and defined according to a statistical analysis of the execution of real-world programs on a given architecture. Indicating with $n(s)$ and $oh(s)$ the number of clock cycles for nominal execution and the number of stall cycles of instruction s and with $p(s)$ the parallelism coefficient, the estimated CPI is expressed as:

$$CPI_{est}(s) = p(s) \cdot [n(s) + oh(s)] \tag{3.1}$$

Considering the functionality decomposition, this equation becomes:

$$CPI_{est}(s) = \sum_{j=0}^{4} p(s) \cdot [n(s,j) + oh(s,j)] \tag{3.2}$$

where $p(s)$ is a statistical factor representing the parallelism that can be exploited executing instruction s with respect to the whole instruction set, while $n(s,j)$ and $oh(s,j)$ are latency values associated to instruction s and the j-th functionality, corresponding to the coefficients $a_{s,j}$ and $a'_{s,j}$ of equation (2.4). On the contrary, the energy associated to each instruction is still described by equation (2.3), because energy is an additive quantity, so it is not reduced by the parallel execution. However, it has to be noticed that the reduction of the actual CPI given by equation (3.1) implies that the average power $w(s)$ absorbed by each instruction is increased with the the parallel execution, the mean power $w(s)$ being given by the following equation:

$$w(s) = \frac{e(s)}{CPI_{est}(s) \cdot \tau} \tag{3.3}$$

where τ is the clock period.

The newly introduced parameter p_s is associated to a single instruction, so it can be used statically in a time or power estimation process. However, it is obtained by folding the *dynamic* information about the parallel instruction execution on the only instruction s in a way similar to what has been done for the coefficient $oh_{s,j}$. The methodology that lead to the estimation of these parameters is described in the following section.

3.2 Mathematical Model

For the purpose of producing a static estimation of the actual CPI associated to a given instruction, a taxonomy of instruction sets has been proposed. This taxonomy is essential to reduce model complexity and to allow for a feasible statistical analysis. Starting from such a taxonomy, a mathematical model is introduced with the purpose of estimating the overhead caused by inter-instruction effects and the coefficient related to parallel execution.

3.2.1 Instruction Set Taxonomy

In order to maintain the approach as general as possible, no specific architecture or set of architectures should be considered. This is due to the fact that each architecture is characterized by strongly different execution capabilities; choosing one of them thus would lead to a closed, non-extensible model.

A simple solution to this issue is to provide some general classes to be associated with architecture-specific instructions. Recalling what has been done in [2], instruction classes were associated to the type of hazard their members could cause, in order to classify instruction with respect to their dynamic behavior. Having introduced parallelism, the classification must take care of the dynamic interaction between instruction with respect to both inter-instruction effects and parallel execution, as formalized by the following definition.

Definition 1 *Given an instruction set $\mathcal{I}$, the equivalence relation $\mathcal{R} \subseteq \mathcal{I} \times \mathcal{I}$:*

$$s_i \; \mathcal{R} \; s_j \iff s_i \text{ and } s_j \text{ have similar dynamic behavior};$$

defines a taxonomy $\mathcal{C} \in 2^{\mathcal{I}}$ on the instruction set $\mathcal{I}$ as the partition induced by $\mathcal{R}$ on the instruction set $\mathcal{I}$. The cardinality $|\mathcal{C}|$ of the taxonomy depends on the relation $\mathcal{R}$. The taxonomy $\mathcal{C}$ is thus formed by the classes c_i with $i \in [1; |\mathcal{C}|]$.

Definition 1 gives a way to obtain the taxonomy based on the equivalence relation $\mathcal{R}$. Nevertheless, $\mathcal{R}$ is still to be properly defined for each instruction set and architecture. Definition 1 gives a way to obtain the taxonomy based

on the equivalence relation $\mathcal{R}$. Three approaches are possible:

Architectural The relation $\mathcal{R}$ is defined *a priori* and is based on the knowledge of both the instruction set and the architectural details.

Numerical The relation $\mathcal{R}$ is defined *a posteriori* based on the data extracted from simulation of the dynamic behavior of instructions.

Full The relation $\mathcal{R}$ is always false. In this case each instruction belongs to a different case, i.e. no classification is performed.

Chapter 8 shows and discusses the results obtained using these classifications.

3.2.2 Model Definition

A statistical characterization of the instruction timing can be obtained from equation (3.2). It is thus necessary to statistically characterize the overhead parameter $oh(s, j)$ and the parallelism coefficient $p(s)$. However, to estimate the instruction timing, it is sufficient to know the global overhead associated to each instruction, that is the parameter $oh(s) = \sum_{j=0}^{4} oh(s,j)$. The overhead per functionality can be obtained from this global value as described in [3].

3.2.2.1 Interlock model

As an extension of the approach described in [2], this work uses some definitions that are briefly recalled here. An *execution trace* Γ can be seen as an ordered set of instructions executed during a program run. Let a trace Γ be:

$$\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_N\}, \quad \gamma_k \in I, \quad N > 0$$

where N indicates the execution trace size. Instructions γ_k are then classified by means of the relation $\mathcal{R}$ and the *membership function* is accordingly:

$$\langle k, i\rangle = \begin{cases} 1 & \text{if } \gamma_k \in c_i \\ 0 & \text{otherwise} \end{cases}$$

where the membership function shows the following property:

$$\sum_{i=0}^{\mathcal{C}} \langle k, i\rangle = 1 \tag{3.4}$$

since an instruction belongs to one and only one class of a partition.

The overhead introduced by dynamic effects during the execution can be associated to the instruction that has been stalled in order to resolve an hazard situation. This is described by the following definition:

Definition 2 *The* delay *introduced with respect to instruction γ_k is given by the function $t(\gamma_k)$.*

The function $t(\gamma_k)$ represents the overhead associated to instruction γ_k; such overheads have to be collected and associated to instruction classes. Given the probability of finding a class in the execution trace, it is possible to define a stochastic variable associated to it:

Definition 3 *The* class delay *is the delay associated with the execution of an instruction belonging to class c_i when interacting with some other instruction; it is modeled by the stochastic variable D_i, which is characterized by its density function:*

$$f_{D_i}(d) = \frac{\sum_{k=1}^{N} \delta_{t(\gamma_k)=d} \langle k, i\rangle}{\sum_{k=1}^{N} \langle k, i\rangle} \tag{3.5}$$

where N is suitably large[1]*.*

The overhead parameter $oh(s)$ can be obtained directly from the class delay: it is sufficient to know the instruction class c associated to instruction s, and then average the corresponding stochastic variable.

3.2.2.2 Parallel Execution Model

The parallelism coefficient can be estimated experimentally starting from the execution trace Γ and observing the instructions that are executed in parallel. Similarly to the computation of overheads, the parallelism coefficients are referred to instruction classes. According to this approach, the more instructions $s \in c_i$ belonging to a given class are executed in parallel, the lower the corresponding parallelism coefficient $p(s)$ and $CPI_{est}(s)$ are. To determine $p(s)$ it is necessary to know when an instruction γ_k starts and ends executing. The notion of time is here intended as the number of clock cycles since the beginning of the execution. This is clarified by the following definition.

[1]For a good approximation, $N \geq 10^6$.

Definition 4 *Let $t_{in}(\gamma_k)$ the starting time of a generic instruction $\gamma_k \in \Gamma$ and $t_{out}(\gamma_k)$ its ending time. The **time range membership function** of instruction γ_k with respect to class $c_i \in \mathcal{C}$ at time t is defined as:*

$$\lceil t,k,i \rfloor = \begin{cases} \langle k,i \rangle & \textit{if } t_{in}(\gamma_k) \le t \le t_{out}(\gamma_k) \\ 0 & \textit{otherwise} \end{cases} \tag{3.6}$$

where the values $t_{in}(\gamma_k)$ and $t_{out}(\gamma_k)$ are properties of the instruction γ_k with respect to a given execution trace Γ.

It is worth noting that the time range between $t_{in}(\gamma_k)$ and $t_{out}(\gamma_k)$ not only depends on the instruction latency but also includes the inter-instruction overhead resulting from stalls. When an instruction is stalled, in fact, it still occupies some resources. The time range membership function allows to know, at each clock cycle, which instructions are being executed.

Starting from the time range membership function it is possible to aggregate values in a per-class vision.

Definition 5 *The **class load function** represents the number of instructions belonging to class c_i being executed at time t. It is defined as:*

$$\lceil t,i \rfloor = \sum_{k=1}^{N} \lceil t,k,i \rfloor \tag{3.7}$$

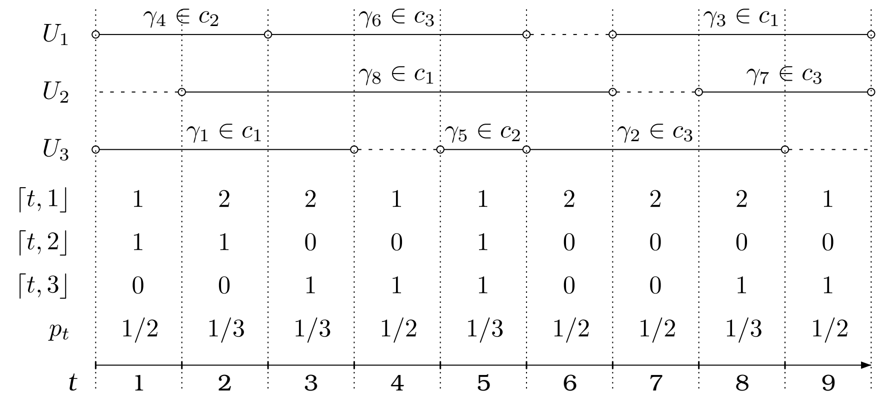
The class load function can be used to compute an instantaneous parallelism coefficient, defined as follows.

Definition 6 *The **instantaneous parallelism coefficient** is defined as:*

$$p_t = \begin{cases} 1/\sum_{i=1}^{|\mathcal{C}|} \lceil t,i \rfloor & \textit{if } \sum_{i=1}^{|\mathcal{C}|} \lceil t,i \rfloor \neq 0 \\ 0 & \textit{otherwise} \end{cases} \tag{3.8}$$

where the summation extends to all classes in the taxonomy.

Figure 3.1 clarifies these concepts with an example in which three functional units U_1, U_2 and U_3 execute eight instructions $\gamma_1, \ldots, \gamma_8$ belonging to the classes c_1, c_2 and c_3. The figure is composed of two parts: the upper portion shows the scheduling of instructions on each unit while the lower portion reports the values of $\lceil t,i \rfloor$ and p_t for the considered scheduling.



$\lceil t,1 \rfloor$	1	2	2	1	1	2	2	2	1
$\lceil t,2 \rfloor$	1	1	0	0	1	0	0	0	0
$\lceil t,3 \rfloor$	0	0	1	1	1	0	0	1	1
p_t	1/2	1/3	1/3	1/2	1/3	1/2	1/2	1/3	1/2
t	1	2	3	4	5	6	7	8	9

Figure 3.1: Example of parallelism computation

Consider, for instance, the clock cycle at $t = 3$ and with instructions γ_6, γ_8 and γ_1 being executed. The class load function $\lceil 3,1 \rfloor$ is equal to 2 since $\gamma_8, \gamma_1 \in c_1$. Similarly, $\lceil 3,3 \rfloor$ is equal to 1 since $\gamma_6 \in c_3$ and $\lceil 3,2 \rfloor$ is 0 since no instructions of class c_2 are being executed. According to equation

(3.8), the instantaneous parallelism coefficient p_3 is equal to $1/(2+0+1) = 1/3$. It can be proved that $p_t \in [1/M; 1] \cup \{0\}$ with M being the maximum number of instruction that the specific architecture is capable of handling in the same clock cycle. As an example consider a simple DLX-like 5-stage pipeline architecture [12]: in this case $M = 5$ since, when the pipeline is full, all its stages are executing an instruction at every clock cycle. In more complex architectures, where more pipelines are present and possibly share some of the stages, the computation of M becomes more sophisticated since the observation of the status of the single units of all pipelines is necessary. The instantaneous parallelism coefficient p_t must then be aggregated according to the selected taxonomy in order to obtain a per-class vision of the amount of parallelism that the architecture under analysis can actually exploit. The following definition formalizes this concept.

Definition 7 *The **class parallelism coefficient** is a scale factor influencing the execution time of an instruction belonging to class c_i when executed in parallel with other instructions. It is modeled by the stochastic variable P_i, which is characterized by the density function:*

$$f_{P_i}(x) = \frac{\sum_{t=0}^{\infty} \delta_{p_t=x} \lceil t, i \rfloor}{\sum_{t=0}^{\infty} \lceil t, i \rfloor} \tag{3.9}$$

where the summations actually extend only over all clock cycles needed for the execution of the trace Γ.

Referring again to the execution trace of figure 3.1, consider the density function $f_{P_3}(x)$. Since $p_t \in \{1/3,\ 1/2\}$ and thus $\delta_{p_t=x} = 1$ only when $x = 1/3$ or $x = 1/2$, then $f_{P_3}(x)$ is to be computed only for such values. In particular for $x = 1/3$:

$$\begin{aligned} f_{P_3}(1/3) &= \frac{\sum_{t=1}^{9} \delta_{p_t=1/3} \lceil t, 3 \rfloor}{\sum_{t=1}^{9} \lceil t, 3 \rfloor} \\ &= \frac{0+1+1+1}{0+0+1+1+1+0+0+1+1} = \frac{3}{5} \end{aligned} \tag{3.10}$$

The same procedure leads to the result $f_{P_3}(1/2) = 2/5$. The parallelism coefficient $p(s)$, similarly to the instruction overhead, can conveniently approximated with the expectation value of the stochastic variable P_i, that is:

$$p(s) = E[P_i] = \int_0^1 x \cdot f_{P_i}(x) dx \quad \text{with} \quad s \in c_i,\ x \in \mathbb{Q} \tag{3.11}$$

It must be noted that $x \in \mathbb{Q}$ since it is computed as the ratio of two integer numbers and that $0 \le x \le 1$ by definition, thus the integral is computed according to the Lebesgue's notion of measure. Concluding the example, $p(s)$ for instructions in class c_3 is:

$$p(s) = \frac{1}{2} \cdot f_{P_3}(1/2) + \frac{1}{3} \cdot f_{P_3}(1/3) = \frac{2}{5} \tag{3.12}$$

4 Methodology

This chapter presents the proposed methodology for the practical application of the model described in Chapter 3. After giving an overview of the flow in section 4.1, each section describes a methodology step, explaining also the reasons and previous literature that are behind the methodology design issues: section 4.2 presents the advantages and the motivation of a micro-compiled approach; section 4.3 details the proposed simulation approach; section 4.4 shows the tuning activity to be performed to obtain statistical figures for class and instruction timing. Finally, section 4.5 explains the algorithm used to apply class figures to estimate the execution time of an arbitrary program.

4.1 Methodology Flow

The proposed methodology consists in a series of activities that the model developer has to endure. These activities can be summarized as (see also figure 4.1):

1. Select a target processor architecture
2. Use the provided library (see Chapter 5) to build a behavioral simulator of the chosen architecture
3. Build a micro-compiler of the chosen instruction set
4. Possibly validate the simulator against a set of micro-benchmarks [13]
5. Develop the preferred classification, optionally using the developed simulator
6. Apply the tuning process to a series of benchmarks
7. Apply the model to the code

Each of these steps has an importance of its own, and it is detailed in the following.

First of all, the model developer has to focus on a target architecture: while the methodology is general, its application has to be restricted do a single architecture or a very narrow set of architectures. This is due to the fact that commercial architectures are extremely different between each other, and the design space for modern processors is also fairly large.

After choosing the target processor, as stated from definition 4, to apply the model the starting and ending instants of each instruction have to be known. The solution proposed here is to develop an architectural simulator that takes an assembly trace and annotates it by adding the starting and ending instants of each instruction. This simulator has a reference implementation that is discussed in Chapter 5. This simulator can be validated against a physical processor or against a validated simulator of the same architecture [8].

The provided library needs a micro-compiler to be built: this lets the simulator be as general as possible, without having to care about low-level assembly details. In fact, the micro-compiler translates the architecture-specific assembly into an expanded, loosely encoded micro-instruction set format called the *micro-code*. A library is provided to ease this task.

The instruction set has to be classified: this is done in order to reduce computational complexity (with a noticeable performance increase [2]), and to group instructions whose behavior is not known together with similar, but known, ones. The classification can be performed using the simulator to obtain figures for each instruction: this leads to a scatter diagram, where similar instructions tend to distribute in clusters. There are several statistical methods to obtain a supervised or non-supervised classification of data sets, among which we can cite clustering

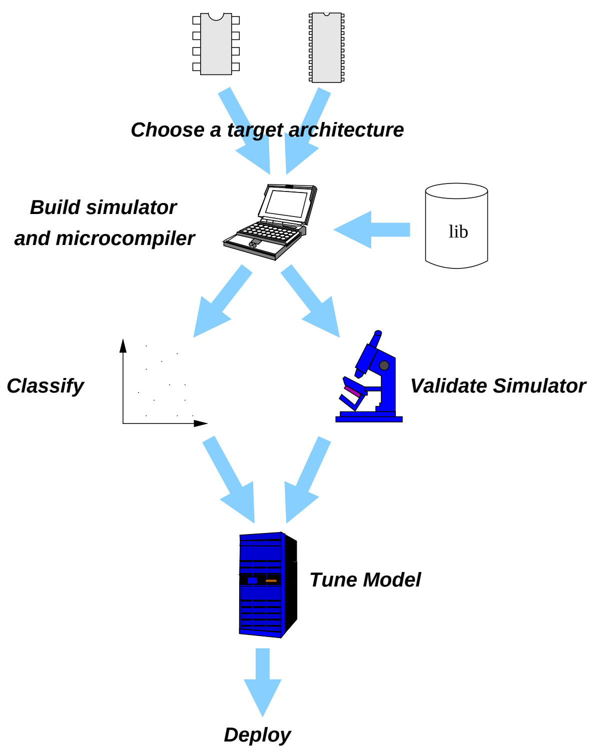

Choose a target architecture
Build simulator and microcompiler
lib
Classify
Validate Simulator
Tune Model
Deploy

Figure 4.1: Methodology flow for model application

algorithms, minimum deviation, maximum likelihood, K-means, etc [19]. The developer can choose the method he likes best or go through a manual classification. In either way, the reference implementation tools simplify remarkably this task.

Once the micro-compiler and classification have been defined, the model can be tuned by tracing a set of benchmarks: these are used to build the densities for every stochastic variable, i.e. overheads and coefficient parallelism. The data obtained can be used directly on newly produced code to obtain an accurate estimation of its execution time.

4.2 Micro-Compilation

This section details the reasons that brought to the choice of micro-compiling the assembly source before accepting it into the simulator. This has been done for some reasons:

Generality By using a loosely encoded, internal micro-code, the simulator can be completely general and it can abstract from the target assembly.

Code reuse The code developed for a specific architecture simulator may be used for others with similar characteristics. Also, the use of a unified assembly language for every simulator model pushes for the development of configurable and adaptable modules for different architectures.

Performance The proposed general micro-code has a very small set of instructions with extremely simple behavior: it can be faster than real decode and execution.

For each instruction set architecture, a new micro-compiler has to be built. Anyway, this should be less expensive than renewing and developing the simulator from scratch for every target architecture. In addition, the tool chain provides libraries and macros provided in order to make the writing of a new compiler as simple as possible.

The development of the micro-compiler requires some knowledge of the target architecture and instruction set: it is mandatory to know for each instruction which elements of the datapath are used and the corresponding latencies. These are not always available, but usually can be retrieved from assembly manuals and architecture datasheets. For micro-code details see Chapter 5, noticing that while its syntax is general, compilation is strictly bound to the structure of the simulated processor. This means that the micro-compiler has to be build *after* the simulator structure has been defined.

4.3 Behavioral Simulation

To obtain the data needed by the model, precise instruction performance figures has to be known. For the methodology to be effective, this data has to be retrieved in a fast and and simple way.

The simplest way to obtain the starting and ending cycle of each instruction, considering all the architectural details involved in the execution, is to use a simulator; RTL simulators represent an optimal choice as far as accuracy is concerned, but they significantly lack in speed. *Instruction Set Simulators* (ISSs) are another valid choice, since they are usually very accurate while running several orders of magnitude faster than RTL simulators. Various approaches have been proposed for ISS development; even these simulators are not particularly efficient, and are scarcely available for commercial systems: ISS are thus not a viable solution. Another possibility is to use a *performance simulator*, a simpler kind of simulator that considers the performance factors only [20]; these are often more available than ISSs, but anyway not easy to find for a given architecture and frequently lack of documentation. In addition, performance simulators are frequently less than cycle-accurate.

To solve such problems, an architectural simulator has been introduced: this is the meeting point between ISS and performance simulator in the sense that it provides nearly cycle accurate figures even if it avoids functional simulation. In practice, the proposed simulator models the architecture in its details under a performance-only point of view. instructions are processed into the various modules of the system only in the sense they are stored for the appropriate number of cycles. This performance simulation environment has to be highly customizable to fit any processor available and, in addition, it should to provide common modules to maximize code reuse and minimize development time.

4.3.1 Design Space Exploration

The simulation architecture should be able to consider every aspect of modern processors design space. In this way, even newly produced processors can be simulated without having to rewrite the core simulation code. Such design space is extremely large, but for performance analysis it restricts to a few fields of interest:

- Pipelining
- Branch Prediction Techniques
- Instruction Level Parallelism
- Memory access

To cover the design space in each of these fields, two possible solutions are viable: simulate the system at a lower level with respect to these (e.g. functional units that handle bitstreams instead of instructions) or to develop a modular system whose modules hold enough degrees of freedom. The former solution may result too complex, hence the latter was chosen.

To gain degrees of freedom, it has been chosen that the simulator would work on an *abstraction* of processor architecture: each system is seen as a set of *functional units* that communicate

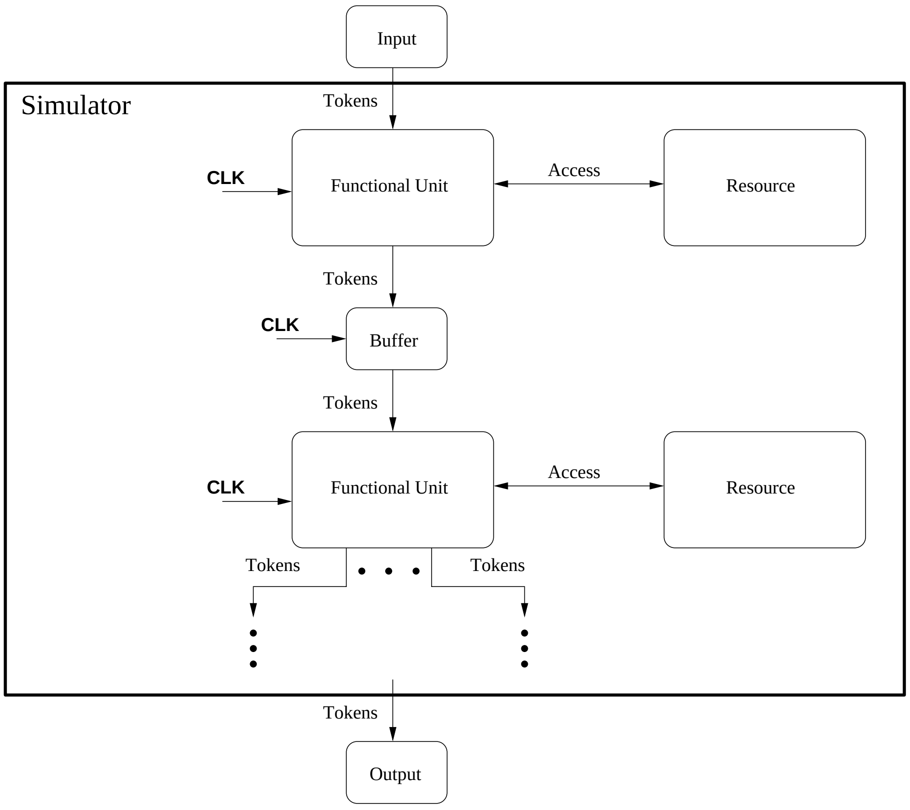


Figure 4.2: Proposed simulator architecture

with each other by exchanging *tokens*. Each functional unit may also have to access *resources* in order to process a token, and resources can grant or refuse access by a given unit. All of these elements work concurrently in a discrete time environment, where the unit time interval is represented by the clock cycle.

As figure 4.2 shows, functional units are exchange tokens via buffers that are used to hold tokens until the next clock cycle: this means that the simulator is strictly synchronous. This last choice makes the modeling of asynchronous circuits impossible, but this can be considered a good trade-off since asynchronous systems have very scarce diffusion.

In the following, some examples on how this structure can fit on the design space of ILP, Pipelining and Memory Access are presented.

4.3.1.1 Pipelining

Pipelining has been introduced in micro-processors starting from the end of the 1960s as an effective floating point number crunching technique [25]. With pipelining, a number of functional units are employed in sequence to perform a single computation. These units form an assembly line or pipeline: each unit performs a certain stage of the computation and each computation goes through the pipeline. Pipelining fits perfectly in the model, as can be seen directly from figure 4.2. From this starting point, many other techniques are used to increment processor performance, and thus extend the design space associate to modern architectures.

To exploit potential instruction parallelism (due to independent instructions) at its maximum, superscalar processors have been introduced. Such processors can process more than one instruction per clock cycle. These are the main focus of this work, since superscalarity and aggressive ILP are the key issues of today's processor design.

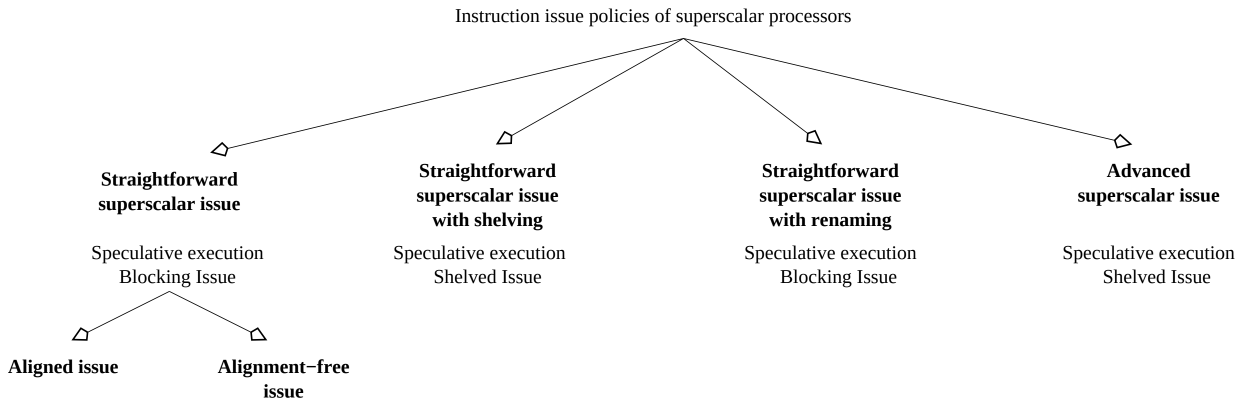

Instruction issue policies of superscalar processors
Straightforward superscalar issue
Speculative execution Blocking Issue
Straightforward superscalar issue with shelving
Speculative execution Shelved Issue
Straightforward superscalar issue with renaming
Speculative execution Blocking Issue
Advanced superscalar issue
Speculative execution Shelved Issue
Aligned issue
Alignment−free issue

Figure 4.3: Instruction issue design space for superscalar processors

4.3.1.2 Superscalarity

While considering superscalarity, it is important to assure that the proposed simulation model can deal with:

- Superscalar Issue
  - Alignment
  - Shelving/Dispatching
  - Register Renaming/Forwarding
- Parallel Execution
- Preservation of sequential integrity

Superscalar issue is a complex field, and involves techniques such as the alignment of instructions, forwarding and register renaming.

There are two types of instruction alignment: aligned issue, instructions are issued in groups only when there is a sufficient number of instructions to fill the group, and unaligned issue, instructions are issued when available. In both cases, the functional unit representing the issue stage of the pipeline models this behavior: for aligned issue it waits until it is full of tokens, for unaligned issue it fires tokens as they arrive (see figure 4.4). This means that each unit has an *internal queue* with variable width: this queue is used to hold tokens inside the functional unit until their processing is finished. The size of such queue can be defined as a parameter of the unit.

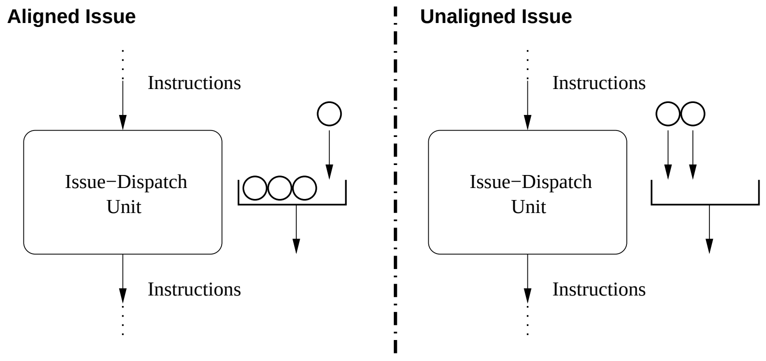


Figure 4.4: Modeling of issue alignment with the proposed simulation architecture

Issue alignment has to be combined with dispatching techniques: there is usually one or more functional unit whose job is to distribute instructions to different execution units. This is modeled using multiple functional units that are connected to a single source. The most popular way of detecting instruction dependencies in superscalar execution is the use of *shelving buffers*, i.e. buffers that hold instructions before they are passed to the execution stage until all dependencies are satisfied. Dependencies are checked by communication with resources and by special flag bits associated to the buffer. Shelving buffers can be *distributed* (exactly one buffer for execution unit), *partially distributed* or *centralized* if they also behave like dispatching units (by being bound to more than one execution unit). Such considerations lead to the consequence that each functional unit can be bound to many other functional units; a functional unit has to know which one it is sending instructions to, so it is mandatory to associate a *unique identifier* to functional units. In addition, dependency check is done by resource access, as figure 4.5 shows.

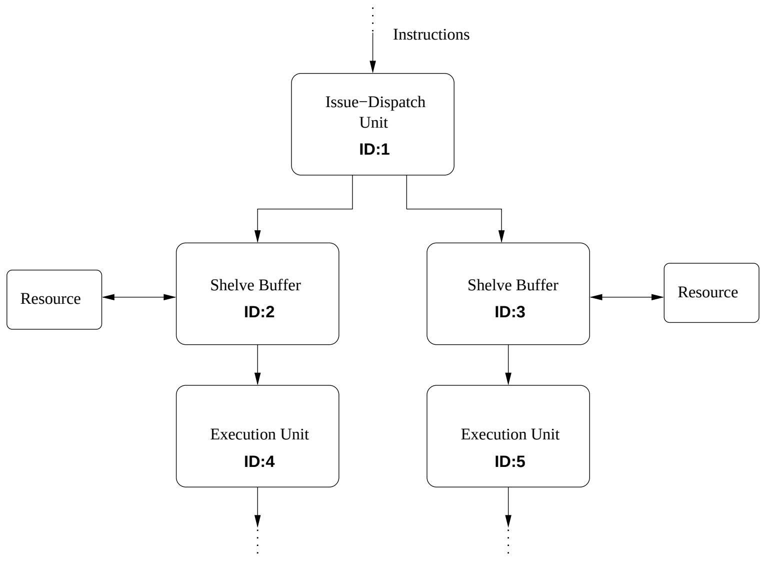


Figure 4.5: Shelving buffer model, considering distributed shelving

Register renaming and forwarding are two common (the latter a bit obsolete) means of avoiding data dependencies between instructions. The former uses a number of physical registers, on which the logical (the accessible ones) are mapped on, the latter forwards operation results from a set of pipeline stages (i.e. memory access and execute) to the preceding one on certain conditions. These approaches can be modeled with resources: a resource behaving as a register file can include all the logic pertaining register renaming while a resource flag can be raised if forwarding is not possible, and thus requiring a stall.

Finally, parallel execution and the preservation of sequential integrity have to be considered. Parallel execution is the simplest: all functional units work concurrently as stated before, this means that the core simulator architecture is sufficient to model a set of concurrent operations. The presence of concurrent operations implies, as a logical consequence, an enhancement of instruction throughput, *out-of-order execution*: instructions are executed whenever resources are available, possibly in an order that differs from the one in which they were fetched. While this is no modeling problem, it means that sequential integrity preservation techniques have to be introduced.

The preservation of sequential integrity is usually obtained by the use of *reorder buffers*, special buffers that act as circular registers, returning only the oldest instruction when requested. In this way, instructions are retired (i.e. exit) from the processor only in sequential order. Reorder Buffers are modeled with resources, their size defined as a parameter.

As shown, all of these approaches can be modeled with the proposed simulation architecture: all the logic pertaining to the various ILP characteristics of a target processor is hidden inside functional unit and resource code. In conclusion, the main simulator characteristics are:

- The simulator is made by *Functional Units*, *Resources* and *Buffers*.
- Each functional unit has only one input, but it may have many outputs
- Each functional unit has a variable-sized internal instruction queue
- A functional unit is identified by a unique ID, known also by other linked functional units
- All functional units act concurrently

These characteristics appear in the reference implementation, as described in detail in Chapter 5.

4.3.1.3 CISC over RISC

In today's processors, it is not infrequent to find CISC processor implementations built over RISC cores. These systems have specific units destined to the decoding of CISC instructions into RISC ones. This is more difficult to model with the proposed structure, but it is still possible: the decode units may output more tokens than they receive as input. In this way there is the added ability of *token generation*: in practice, the simulator decode units act exactly as their physical counterparts, by issuing a set (i. e. one or more) of micro-instructions for every one processed. However, since all the interest is put on performance simulation, this behavior is not mandatory. In some architectures, decoding and execution over a RISC core may be modeled considering CISC instructions only, looking at the RISC core as a black box, maintaining accuracy. As an example, an architecture where RISC instructions have all the same latency could be modeled with a CISC-only solution, with instructions having variable latency.

4.3.2 Simulation algorithm

Now that all the modules of the simulator are defined, the simulation algorithm can be specified. Such algorithm can be described using pseudo-code:

Algorithm 1 Simulation of a clock cycle

```
 1: sort_functional_units();
 2: for all functional units do
 3:   execute_cycle();
 4: end for
 5: for all buffers do
 6:   update();
 7: end for
 8: for all resources do
 9:   update();
10: end for
```

As the code clearly shows, the algorithm is made by a set of sequential steps:

1. To avoid access conflicts on resources, concurrent access is granted only to the oldest instruction: functional units are sorted according to the *age* of the instructions they contain, from oldest to youngest (line 1)
2. Each functional unit executes a cycle in the given order (lines 2 to 4)

3. Each instruction buffer is updated: instructions are moved from master to slave entry, so they can be accessed from the next functional units in the following cycle (lines 5–7)

4. Eventually, resources are updated. It is worth noting that this step is optional, since resources may be clock-independent (lines 8–10)

The simulation iteration is repeated for the number of clock cycles needed to execute the chosen benchmark.

4.4 Tuning

The tuning process is a straight application of the model presented in Chapter 3. The available timing information, due to simulation, is aggregated in density functions: delays and parallelism coefficients are computed (by application of the corresponding formulae) for each class, their values summed and normalized. The result is a set of matrices that represent delay and parallelism coefficients density functions. The expectation value of these functions is then computed and stored, completing the model tuning. The algorithm describing the tuning process follows:

Algorithm 2 Tuning of the model given a sample trace

```
 1: clock := 0;
 2: $V$ := empty vector of instructions $s$;
 3: for all instructions in the trace do
 4:   get next instruction $I$;
 5:   while clock $< t_{in}(I)$ do
 6:     remove instructions $s$ from $V$ such that $t_{out}(s) > clock$;
 7:     $p_s(t) := 1/\texttt{sizeof}(V)$;
 8:     for all instructions $s \in V$ do
 9:       update $P_j$ density function, with $j$ such that $s \in c_j$;
10:     end for
11:     clock++;
12:   end while
13:   update $D_k$ density function, with $I \in c_k$;
14:   put $I$ into a vector $V$;
15: end for
```

As the algorithm shows, all tuning takes place during a single scan of the trace file. The process starts at the first clock cycle (line 1) and reads each instruction (lines 3–4); after each instruction is read, the clock is incremented until its starting cycle is reached (lines 4–11). Concurrently, the *instantaneous parallelism coefficient* is computed (line 7) and it is used to update the density function of the *class parallelism coefficient* (P_j) for every active instruction, as it is clearly shown on lines 8–10. Active instructions are held in a vector when the clock reaches their starting time and removed when it reaches their ending time, namely t_{in} and t_{out} (lines 5 and 13).

It is worth noting that the parallelism coefficient is computed using a vector of the instructions that are being executed (lines 3–9) at each clock cycle, while the delay does not need such information (line 11).

4.5 Annotation

The obtained values are an estimate of the time needed for each instruction to be executed, given its class. By summing all the estimated times of each instruction, an estimation of the total time needed to execute a program is obtained. The algorithm that estimates the total time of a given code portion follows.

Algorithm 3 Estimation of the execution time of program P

```
1: total := $0$;
2: for all instructions in the trace do
3:     get next instruction $s$;
4:     identify class $c_j$ such that $s \in c_j$;
5:     $n(s)$ := nominal time of instruction $s$
6:     $p_j$ := expected value of parallelism for class $j$
7:     $oh_j$ := expected value of overhead for class $j$
8:     total = total $+ p_j \cdot (n(s) + oh_j)$
9: end for
```

This activity requires a single scan of the trace file: the instructions are read (line 3), their time characteristics and class are identified (lines 4–7) and finally the average execution time of the read instruction is computed (line 8).

5 Software Tools

ATT
atomic tribes tune
Assembly Trace-based Tuner & Classifier
CEFRIEL
A Centre for Information Technology Research and Education
POET

The methodology described in Chapter 4, has been implemented in a software tool-set, namely the ATT tool-set, to be integrated in a co-design environment for the evaluation and early-prototyping of embedded systems: the **POET** (*Power Optimization of Embedded Systems*) project. The tool implementation had to consider different target architectures on which the embedded system can be developed. Besides, the introduced model requires a deeper knowledge of the instruction flow into the given architecture in order to capture the information about both inter-instruction overhead and execution parallelism. In particular, Definition 4 introduces the time aspect that is necessary to compute the parallelism coefficient: it is mandatory to know the exact starting and ending instants of each instruction in the execution trace. To this purpose, the software implementation needs to simulate the instruction flow into the architecture with clock cycle precision. However, a functional or an instruction set simulator is not required; what is needed is a *behavioral simulator*, i.e. a tool capable of extracting information on the dynamic inter-instruction behavior from the execution trace. To this purpose, TrIBeS (*Trace-based Instruction Behavioral Simulator*) has been developed: it is meant to provide a general framework to develop processor architecture simulators (see Chapter 4 and Section 5.1). TrIBeS works in conjunction with another tool, called ATOMIC (*Architecture-specific Trace Oriented Micro Instruction Compiler*), aimed at performing a micro-compilation of assembly instructions so that they can be fed to TrIBeS (see Section 5.2).

TrIBeS produces a modified instruction trace providing all the data necessary to the tuning of the model, which is performed by the TUNE tool (see Section 5.3). Figure 5.1 shows the control flow for the software tools.

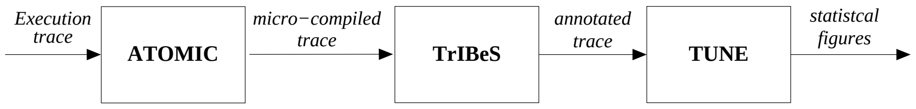


Figure 5.1: Software tool control flow

Once tuned, the model may be applied directly to the code that has to be analyzed; This process is carried out be the ANNOTATE tool (see Section 5.4). Such framework is flexible enough to model a very large set of architectures and, at the same time, sufficiently standard to provide a wide range of capabilities that each specific simulator can exploit without modification. A similar proprietary simulator (Asim) has been developed at Compaq [10]. However, neither sufficient details nor an implementation for a commercial architecture are currently available to allow a comparison with the framework presented here.

5.1 Behavioral Simulator

In a top-down design approach, TrIBeS can be seen as a set of independent objects, called *functional units* that receive one or more instructions as input, work on them for some time, and output these instructions when they have finished. To execute a job, each functional unit accesses a set of *resources*, that can grant or refuse access in a given time instant. The simulator itself only takes care of synchronizing the whole structure. Figure 5.2 explains this.

The *instructions* shown in figure 5.2 are, in particular, the *assembly instructions* of the simulated architecture. This means that TrIBeS acts as an *instruction scheduler*: instructions flow into

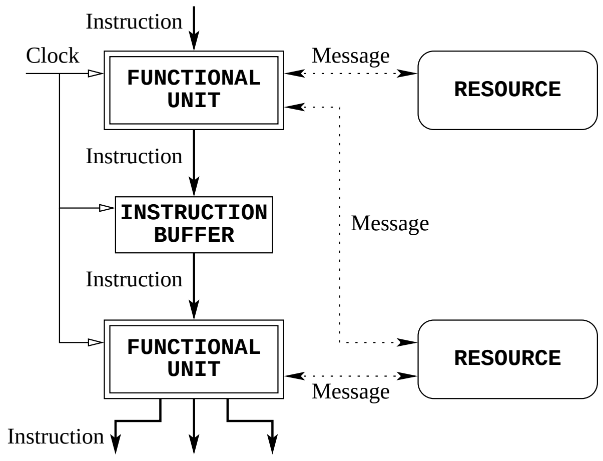


Figure 5.2: TrIBeS architecture

the simulator and are scheduled for execution according to resource constraints. The shown buffers represent the link between functional units, while *CLK* is a synchronization signal used to simulate the concurrent activity of all functional units; the simulation algorithm is described in section 4.3.2.

As already stated, TrIBeS is not a functional simulator; it simulates a processor architecture, that is its internal token flow, starting from a set of already executed instructions (i.e. an *instruction trace*).

Looking more in detail (and in an Object-Oriented way) at TrIBeS leads to a class view of the system. The base classes are:

Simulator :
The main class, it is completely static (actually it works like a namespace) and offers a `run()` method that starts the simulation.

Instruction :
This class represents the main token of the simulation, that flow in the different functional units.

Microcode :
This class has a direct association with *Instruction* and represents the corresponding compiled microcode. It acts as a simple structure with a few fields (microcode plus some optional parameter).

InstructionQueue :
This can be seen as a master-slave buffer that acts as a connection between functional units. Instructions enter the *InstructionQueue* and are put out in the next cycle.

Resource :
Functional units try to access resources and, depending on the state of the simulated processor, they are granted access or forced to wait. This behavior of *Resources* is used to model actual inter-instruction dependencies.

Figure 5.3 shows how these classes are related to each other.

In commercial processors, several heterogeneous components can be modeled with resources: this forces the specification of some resource subclasses that represent five main resource types that have been identified:

BranchPredictionResource :
Tells a functional unit whether the branch target has been correctly predicted or not.

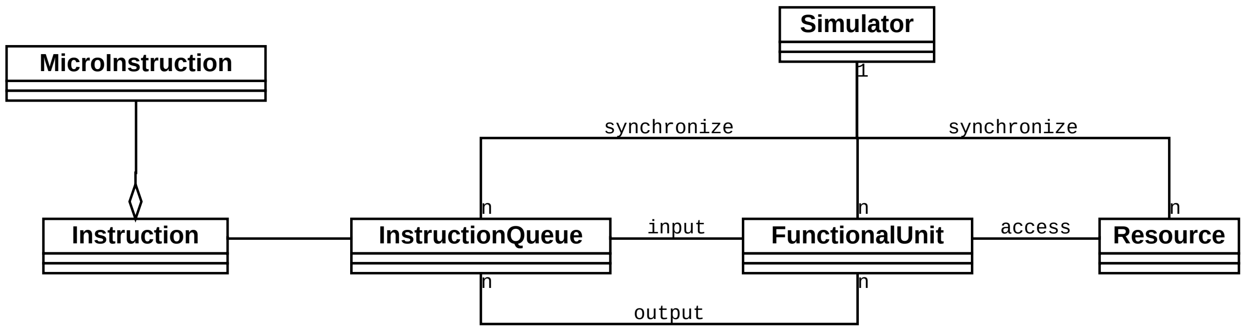


Figure 5.3: An undetailed class view of TrIBeS

RegisterFileResource :
One of the processor's register files, it allows functional units to read, write, lock and free registers.

BufferResource :
Models a generic buffer (queue) with insertion and extraction.

MemoryResource :
Models memory access with reads and writes. This class can be suitably extended to account for arbitrarily complex memory hierarchies.

FlagResource :
Models a generic flag or signal with increment and decrement capabilities.

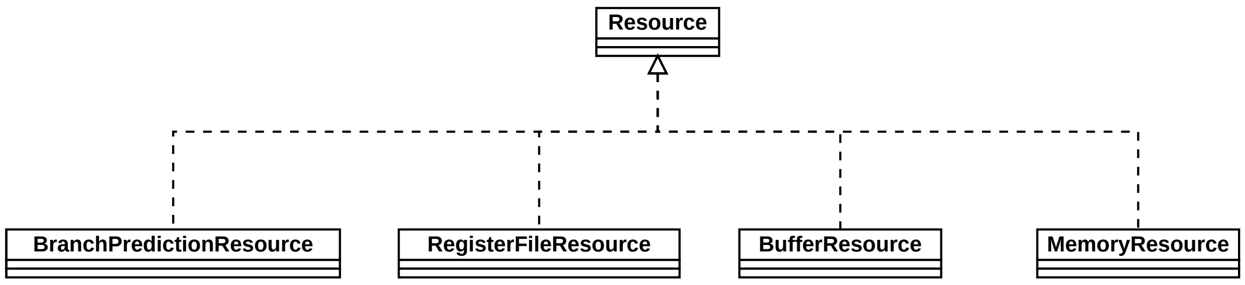


Figure 5.4: Some extensions for the *Resource* class

The inter-instruction effects such as interlocks or cache misses can be modeled by means of request to resources: for example, a typical Read-After-Write (RAW) hazard can be modeled as a successful lock of a register in a *RegisterFileResource* requested by an instruction, followed by an unsuccessful read access request to the same register by a subsequent instruction. Once a resource refuses a request, the functional unit issuing the request stalls the corresponding instruction for one clock cycle, in order to send the same request in the subsequent cycle. The stall is then propagated to the preceding functional unit via the *InstructionQueues*, as they will get full and will not accept the insertion of new instructions. This mechanism is powerful enough to model any hazard type in a simple and distributed manner. Besides, an important improvement with respect to the previous work is the ability to model the processor memory, in order to evaluate the impact of memory effects on the time and power consumption of a code portion. This feature is missing at the actual state of development of TrIBeS (an ideal, zero-delay memory is considered), but its introduction is straightforward.

Back to the architectural description of TrIBeS, it is necessary to specify two special *InstructionQueue* classes: one that reads file input and one that actually writes the output file. Such classes are produced by extending the *InstuctionQueue* class into *InputQueue* and *OutputQueue* as can be seen from figure 5.6.

OutputQueue has another important function: it tells the simulator when the last instruction has been executed. This is done by tracking the number of instructions that exit the simulator and by decrementing a counter; when such a counter reaches zero, the simulation ends.

TrIBeS also produces the data necessary for the computation of the parallelism coefficient, namely the times $t_{in}(\gamma_k)$ and $t_{out}(\gamma_k)$, for each instruction γ_k in the execution trace Γ. In fact, the first functional unit who receives an instruction γ_k (generally, a fetch unit) sets the $t_{in}(\gamma_k)$ value, while each functional unit that γ_k traverses sets the $t_{out}(\gamma_k)$ value. In this way, at the end of the instruction flow, γ_k has the correct values whatever path it has taken through the architecture. The range specified by the so computed $t_{in}(\gamma_k)$ and $t_{out}(\gamma_k)$ includes any possible interlock overhead, as required by the model definition (see Section 3.2.2). A detailed view of the simulator architecture in UML can be seen in figure 5.5.

5.2 Micro-Compiler

To be as general as possible, TrIBeS uses a proprietary microcode, which describes the operation to be carried out by the functional units, the resources to be requested and the flow into the data-path. This way, the tool can be independent from the instruction set of a specific architecture. The compilation process is carried out by ATOMIC, which is specific to each architecture. ATOMIC is developed in C using a parser generator (*bison*) in conjunction with a lexical analyzer generator (*flex*). Each assembly instruction is compiled in a microcode, i.e. in a sequence of micro-instructions that are directly interpreted by TrIBeS. The ATOMIC output can also be binary, in order to speed up the computation. A micro-instruction is composed of an operation code, an integer value and a list of optional parameters. When optional parameters are needed and their number is a-priori unknown, the integer value is used to specify such number. An optional integer field specifies the functional unit to which the micro-instruction is destined. Possible micro-instructions are:

`require <clock-cycles> <fu-name> ;`
: Defines the number of clock cycles that the instruction must spend in the given functional unit; it is also used to decide the functional unit the instruction must be sent to, when multiple choices are possible.

`read <register-number> <register-file> ;`
: Access request to the specified register file for reading a register.

`write <register-number> <register-file> ;`
: Access request to the specified register file for writing a register.

`load <address> ;`
: Access request to the memory resource for loading a data from the specified address.

`store <address> ;`
: Access request to the memory resource for storing a data to the specified address.

`branch <target> <direction> <type> ;`
: Access to the branch prediction resource to verify the correctness of the branch prediction with respect to the type of the branch (taken or not taken).

`use <resource-type> <resource> ;`
: Access to the specified resource, used when the action can be decided directly by the functional unit.

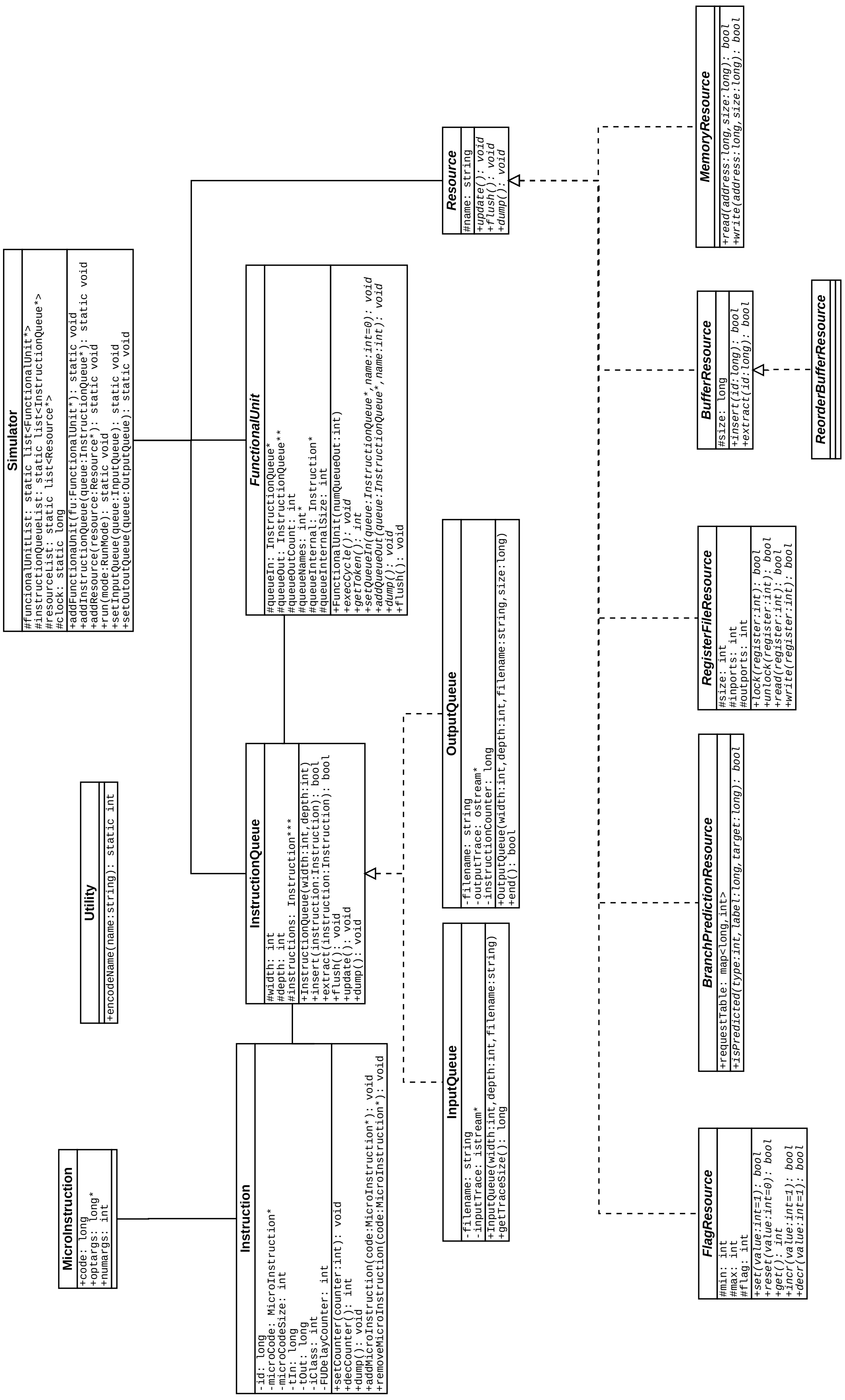

Simulator
#funcionalUnitList: static list<FunctionalUnit*>
#instructionQueueList: static list<InstructionQueue*>
#resourceList: static list<Resource*>
#clock: static long
+addFunctionalUnit(fu:FunctionalUnit*): static void
+addInstructionQueue(queue:InstructionQueue*): static void
+addResource(resource:Resource*): static void
+run(mode:RunMode): static void
+setInputQueue(queue:InputQueue): static void
+setOutputQueue(queue:OutputQueue): static void
Utility
+encodeName(name:string): static int
MicroInstruction
+code: long
+optargs: long*
+numargs: int
Instruction
-id: long
-microCode: MicroInstruction*
-microCodeSize: int
-tIn: long
-tOut: long
-iClass: int
-FUDelayCounter: int
+setCounter(counter:int): void
+decCounter(): int
+dump(): void
+addMicroInstruction(code:MicroInstruction*): void
+removeMicroInstruction(code:MicroInstruction*): void
InstructionQueue
#width: int
#depth: int
#instructions: Instruction***
+InstructionQueue(width:int,depth:int)
+insert(instruction:Instruction): bool
+extract(instruction:Instruction): bool
+flush(): void
+update(): void
+dump(): void
FunctionalUnit
#queueIn: InstructionQueue*
#queueOut: InstructionQueue**
#queueOutCount: int
#queueNames: int*
#queueInternal: Instruction*
#queueInternalSize: int
+FunctionalUnit(numQueueOut:int)
+execCycle(): void
+getToken(): int
+setQueueIn(queue:InstructionQueue*,name:int=0): void
+addQueueOut(queue:InstructionQueue*,name:int): void
+dump(): void
+flush(): void
InputQueue
-filename: string
-inputTrace: istream*
+InputQueue(width:int,depth:int,filename:string)
+getTraceSize(): long
OutputQueue
-filename: string
-outputTrace: ostream*
-instructionCounter: long
+OutputQueue(width:int,depth:int,filename:string,size:long)
+end(): bool
Resource
#name: string
+update(): void
+flush(): void
+dump(): void
FlagResource
#min: int
#max: int
#flag: int
+set(value:int=1): bool
+reset(value:int=0): bool
+get(): int
+incr(value:int=1): bool
+decr(value:int=1): bool
BranchPredictionResource
+requestTable: map<long,int>
+isPredicted(type:int,label:long,target:long): bool
RegisterFileResource
#size: int
#inports: int
#outports: int
+lock(register:int): bool
+unlock(register:int): bool
+read(register:int): bool
+write(register:int): bool
BufferResource
#size: long
+insert(id:long): bool
+extract(id:long): bool
ReorderBufferResource
MemoryResource
+read(address:long,size:long): bool
+write(address:long,size:long): bool

Figure 5.5: Detailed view of TrIBeS architecture

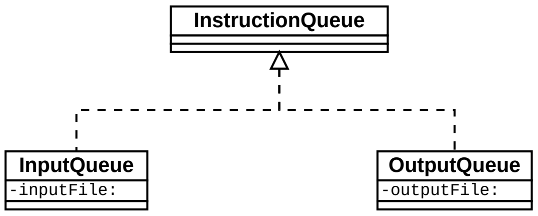


Figure 5.6: Extensions for the *InstructionQueue* class

SPARCv8 assembly	TrIBeS microcode

`mul %r0, %r1, %r2`	`read 0 regfile-int`
	`read 1 regfile-int`
	`require 16 alu-int`
	`write 2 regfile-int`

`ld [%r0 + %r1], %r2`	`read 0 regfile-int`
	`read 1 regfile-int`
	`load 1 address`
	`write 2 regfile-int`

`fadd %f0,%f1, %f2`	`read 0 regfile-fp`
	`read 1 regfile-fp`
	`require 5 alu-fp`
	`write 2 regfile-fp`

Table 5.1: Microcode examples

Table 5.1 shows some example of microcode related to some assembly instruction of the SPARCv8 Instruction Set. It can observed that the multiply is translated in four micro-instructions, three of which are used to access the integer register file, two in read mode and one in write mode. The remaining instruction states that the execution of the multiplication takes 16 clock cycles in the integer ALU. The second example refers to a load instruction: it is worth noting that the `load` micro-instruction has one optional parameter, as specified by the integer value: in fact, the memory destination may require more than one optional parameter, depending on the address size[1]. Finally, the third example shows the microcode associated to a floating-point operation: it can be remarked the use of the floating-point register file and ALU.

5.3 Model Tuning Tool

One important step is the model tuning, (also called parameter estimation), which is carried out by the TUNE tool. In this phase, the software tool analyzes an execution trace produced by TrIBeS, which contains all data necessary to create the statistical figures defined in the model.

The TUNE tool reads instructions from the trace, gets their class, the timing values and the overhead associated to each instruction. Finally, it accumulates this values and computes the delay functions introduced in Section 3.2.2. In particular, if the instruction has a not null delay, it updates the data structures that represent the probabilities and delay random variables for the corresponding class with the associated penalty (see section 4.4. Concurrently, it updates the data structures related to the parallelism coefficient of the corresponding class. These activities take place in a single scan of the trace file. The final result is the computation of class frequencies (by the means of the taxonomy), class delay variables (by the means of the penalty per instruction) and class parallelism variables (by the means of the instant parallelism factor). In practice, we obtain:

[1]The address specified in the load instruction is not resolved here.

- The probability of finding each taxonomy class in the execution trace (its frequency).
- A delay random variable for each taxonomy class, specified by means of its density function.
- A parallelism random variable for each taxonomy class, specified by means of its density function.
- The mean value and variance of all these variables.

Model validation takes place on the mean values, as it is described in the following section.

5.4 Model Application Tool

The model application tool is a straightforward implementation of the algorithm proposed in section 4.5: ANNOTATE reads an assembly trace of a target program, identifies the class of each instruction and accumulates the corresponding estimated time. The result of this activity is an overall estimate of the processor-time needed to execute the target program.

5.5 Tool Performance

The experiments have been performed on a dual Pentium III 966MHz with 512MB of main memory running Linux RedHat 7.2. The performance of all the processing phases have been measured leading to the results summarized in table 5.2

Phase	**Tool**	**Performance**
Instruction tracing	`bintrace`	1.9 Minst/sec
Micro-compilation	`atomic`	70 Kinst/sec
Behavioral simulation	`tribes`	4 Kinst/sec
Model tuning	`tune`	90 Kinst/sec
Estimation	`annotate`	140 Kinst/sec

Table 5.2: Toolset performance

Since the simulation flow involves the first three phases and since all tools can be concatenated in a single pipeline, the resulting simulation throughput is around 4 Kinst/sec. On the other hand, the estimation only requires instruction tracing which is much faster than estimation itself and thus does not impact on the estimation performance. The estimation flow is thus roughly 35 times faster than simulation. This justifies the construction of a model and the partitioning of instruction sets into classes.

6 Target Architectures

Aim of this chapter is the analysis of the inter-instruction dependencies that may affect instruction execution on a given processor. As shown in previous chapters, in general, the energy consumption of a processor depends on many dynamic (data dependent) effects, like cache misses, stalls and processor state. As this work focuses on the pipelined and superscalar execution of code, section 6.1 presents a taxonomy of inter-instruction dependencies; section 6.2 briefly summarizes the characteristics of two different processors, the Intel486™ and the microSPARC™-II, chosen as target architectures on which to build the experimental analysis.

6.1 Taxonomy

Pipeline hazards are caused by inter-instruction dependencies. Hazards prevent the next instruction in the instruction stream from being executed during its designated clock cycle: hazards in pipelines may enforce pipeline stalls. To detect a stall, it is necessary to determine the type of dependency involved.

There are three classes of hazards:

- *Structural Hazards*: they arise from resource conflicts when the hardware does not support every possible combination of instructions in simultaneous overlapped execution.
- *Data Hazards*: they arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
- *Control Hazards*: they arise from the pipelining of branches and other instructions that change the program counter.

Avoiding a hazard often requires some instructions in the pipeline to be allowed to proceed while others are delayed. When an instruction is stalled, all the instructions issued later than the stalled one are also stalled while instructions issued earlier than the stalled instruction must continue, otherwise the hazard will never clear.

6.1.1 Structural Hazards

When a machine is pipelined, the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow every possible combination of instructions in the pipeline. If some of these combinations of instructions cannot be accepted due to resource conflicts, the machine is said to have a *structural hazard*.

Common instances of structural hazards arise when some functional unit is not fully pipelined, so that a sequence of instructions using that unpipelined unit cannot proceed at the rate of one per clock cycle, or when some resource has not been sufficiently replicated. For example, a machine may have only one register-file write port, but in some cases the pipeline might want to perform two writes in a single clock cycle. Structural hazards are heavily architecture and instruction ordering dependent.

6.1.2 Data Hazards

Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on the unpipelined machine. In other words, data hazards occur when an instruction access a register (reading or writing it) after a succeeding instruction tries to access it. This is due to execution overlapping of subsequent instructions in a pipelined architecture.

By convention, the hazards are named by the ordering in the program that must be preserved by the pipeline. Consider two instructions i and j, with i occurring before j. The possible data hazards are:

- *RAW* (read after write): j tries to read a source before i writes it, so j incorrectly fetches the old value. This is the most common type of hazard.
- *WAW* (write after write): j tries to write an operand before it is written by i. The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by j in the destination. This hazard is present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled.
- *WAR* (write after read): j tries to write a destination before it is read by i, so i incorrectly fetches the new value. This hazard occurs when there are instructions that write results early in the instruction pipeline, and others that read a source late in the pipeline. Because of the standard structure of a pipeline, which typically reads values before it writes results, such hazards seldom occur. Pipelines for complex instruction sets that support auto-increment addressing and require operands to be read late in the pipeline could create a WAR hazard.

A simple hardware technique called *forwarding* can solve the problem of data hazards: when an instruction produces a new value that has to be employed by another instruction, the control logic pass the results across the pipeline stages to the functional unit that requires it. Unfortunately, not all potential hazards can be handled by forwarding: some architecture are limited in using this technique due to their data path design.

6.1.3 Control Hazards

When a branch is executed, it may or may not change the PC (program counter) to something different from the next sequential instruction address. If a branch changes the PC to its target address, it is a taken branch, if it does not, it is not taken. If instruction i is a taken branch, no instruction must be issued until the completion of the address calculation. The simplest method of dealing with branches is to stall the pipeline as soon as the branch is detected and until the new PC is determined.

In some machines, control hazards are more expensive in terms of clock cycles. For example, a machine with separate decode and register fetch stages will probably have a branch delay –the length of the control hazard– that is at least one clock cycle longer. The branch delay, unless it is suitably dealt with, turns into a branch penalty. Many older machines that implement more complex instruction sets have branch delays of four clock cycles or more. In general, the deeper the pipeline, the higher the branch penalty in clock cycles.

There are many methods for dealing with pipeline stalls caused by branch delays. A common technique refers to *branch prediction schemes*: the processor tries to predict the target of a branch, which allows fetching instructions from this location; if the prediction is correct, no stall is required, otherwise the incorrectly fetched instruction must be invalidated. Many prediction schemes are known and tested: the simplest predicts the branch as not taken, simply allowing the hardware to continue as if the branch were not executed. Care must be taken not to change the machine state until the branch outcome is definitely known.

A different technique is the *delayed branch*: every branch is followed by n sequential successors, where n corresponds to the branch penalty. Sequential successors are in the *branch delay slots*. These instructions are executed whether or not the branch is taken. The job of the compiler is to make the successor instructions valid and useful. The limitations on delayed branch scheduling arise from the restrictions on the instructions that are scheduled into the delay slots and the ability to predict at compile time whether a branch is likely to be taken or not.

To improve the ability of the compiler to fill branch delay slots, most machines with conditional branches have introduced a *canceling branch*. In a canceling branch the instruction includes the direction that the branch was predicted, so that if the branch behaves as predicted, the instruction in the branch delay slot is fully executed; if the branch is incorrectly predicted, the instruction in the delay slot is turned into a *no-op* (idle).

6.1.4 Dynamic Scheduling

Pipelining can overlap the execution of instructions when they are independent of one another. This potential overlap among instructions is called *instruction-level parallelism* (ILP), since the instructions can be evaluated in parallel. Earlier processors used a technique called *dynamic scheduling*, where the hardware exploit the ILP present in the code by rearranging the instruction execution to reduce the stalls. Dynamic scheduling offers several advantages:

- it enables handling some cases when dependencies are unknown at compile time (e.g., because they may involve memory references);
- it simplifies the compiler;
- it allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline.

These advantages are gained at a cost of a significant increase in hardware complexity. A major limitation of the pipelining techniques is that they use in-order instruction issue: if an instruction is stalled in the pipeline, no later instructions can proceed. The dynamic scheduling approach allows the instructions to begin execution as soon as their data operands are available. Thus, the pipeline will do *out-of-order execution*, which usually implies *out-of-order completion*.

Scoreboarding is a technique allowing instructions to execute out of order when there are sufficient resources and no data dependencies; it is named after the CDC 6600 scoreboard, which included this capability. The goal of a scoreboard is to maintain an execution rate of one instruction per clock cycle (when there are no structural hazards) by executing an instruction as early as possible. Thus, when the next instruction to execute is stalled, other instructions can be issued and executed if they do not depend on any active or stalled instruction. The scoreboard takes full responsibility for instruction issue and execution, including all hazards detection. Every instruction goes through the scoreboard, where a record of the data dependencies is constructed; this step corresponds to instruction issue. The scoreboard then determines when the instruction can read its operands and begin execution.

Tomasulo Approach is another scheme to allow execution to proceed in the presence of hazards developed by the IBM 360/91 floating-point unit. This scheme combines key elements of the scoreboarding scheme with the introduction of *register renaming*. This functionality is provided by the *reservation stations*, which buffer the operands of instructions waiting to be issued, and by instruction issue logic. The basic idea is that a reservation station fetches and buffers an operand as soon as it is available, eliminating the need to get the operand from a register. In addition, pending instructions designate the reservation station that will provide their input. Finally, when successive writes to a register appear, only the last one is actually used to update the register.

As instructions are issued, the register specifiers for pending operands are renamed to the names of the reservation station in a process called register renaming. This combination of issue logic and reservation stations provides renaming and eliminates WAW and WAR hazards. This additional capability is the major conceptual difference between scoreboarding and Tomasulo's algorithm. Since there can be more reservation stations than real registers, the technique can eliminate hazards that cannot be eliminated by a compiler.

6.2 Target Architectures

In this section we will examine in detail two commercial processors chosen for supporting the experimental phase; this analysis is essential to know how the chosen architecture handles hazards and stalls.

The chosen processors are the Intel486™ and the microSPARC™-II; they have been chosen because of their different characteristics (e.g., they are based on a CISC and RISC architecture respectively) and their widespread use in the embedded market. For both architectures, particular attention will be paid to the inter-instruction dependencies that possibly result in a pipeline stall.

6.2.1 Intel486™

The Intel[1] embedded processor family is an implementation of the Intel486™ architecture to be used for embedded applications. It consists of a number of processors with different features, although most of these differences are related to clock speed and power management features. The specific implementation chosen in this document is the Intel™ i80486DX, a full-featured 32-bit CISC processor with integrated floating-point unit.

The Intel™ i80486DX has a 32-bit RISC integer core that perform a complete set of arithmetic and logical operations on 8-, 16-, and 32-bit data types using a full-width ALU and eight general purpose registers.

The Intel486™ processor has four modes of operation: Real Address Mode (Real Mode), Protected Mode, Virtual Mode (within Protected Mode), and System Management Mode (SMM). In Real Mode the Intel486™ processor operates as a very fast 8086. The mode is primarily intended to set up the processor for Protected Mode operation. The other modes are used to simulate an 8086 processor for each task (Virtual Mode) and for OS operation (SMM). We will focus on Real Mode operation.

The Intel486™ uses a microcoded architecture: CISC instructions are decoded and split into one or more microinstructions (RISC type), read from an internal ROM. Each instruction has thus an associated microcode section in ROM, that is executed by the ALU or FP unit.

The processor has a 5-stage pipeline: fetch, a two-stage decode, execution and register write-back; this data-path is embedded in a relatively simple architecture: it has no branch prediction (i.e. branches are considered as always not taken), no forwarding or out-of-order execution. The pipeline stages perform the following actions:

- **F** (Instruction Fetch), fetches instruction either from the 2-deep (32 bytes) prefetching queue or directly from the cache.
- **D1** (Instruction Decode 1), decodes the instructions and sends the appropriate signals to the *control and protection unit.*
- **D2** (Instruction Decode 2), receives the signals from the decode unit and sends the microcode instructions to the execution unit. Addresses for memory operations or jumps are computed in this stage.
- **EX** (Execute), performs ALU, logical and shift operations as well as loading of memory operands.
- **WB** (Write Back), stores computed values in the register file or cache memory.

Because the Intel486™ processor's integer and floating-point units are separate, floating-point instructions can execute in parallel to integer instructions. This simultaneous execution of

[1]All information about Intel processors comes from [5][6]. Hazard analysis is based on [7]

different instructions is called Concurrency. The Floating Point unit is not pipelined, and so it can execute a single instruction at a time.

The processor provides 16 registers for use in general system and application programming.

- *General-purpose data registers.* These eight registers are available for storing operands.
- *Segment registers.* These registers hold up to six segment selectors.
- *Status and control registers.* These registers report and allow modification of the state of the processor and of the program being executed.

The 32-bit general-purpose data registers EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP are provided for holding the following items:

- Operands for logical and arithmetic operations
- Operands for address calculations
- Memory pointers.

General purpose registers can be used for some special purposes: this can be important when taking into account inter-instruction dependencies. The following is a summary of these special uses:

- *EAX* Accumulator for operands and results data.
- *EBX* Pointer to data in the DS segment.
- *ECX* Counter for string and loop operations.
- *EDX* I/O pointer.
- *ESI* Pointer to data in the segment pointed to by the DS register; source pointer for string operations.
- *EDI* Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations.
- *ESP* Stack pointer (in the SS segment).
- *EBP* Pointer to data on the stack (in the SS segment).

The lower 16 bits of the general-purpose registers map directly to the register set found in the 8086 and Intel 286 processors and can be referenced with the names AX, BX, CX, DX, BP, SP, SI, and DI. Each of the lower two bytes of the EAX, EBX, ECX, and EDX registers can be referenced by the names AH, BH, CH, and DH (high bytes) and AL, BL, CL, and DL (low bytes).

The segment registers (CS, DS, SS, ES, FS, and GS) hold 16-bit segment selectors. A segment selector is a special pointer that identifies a segment in memory. To access a particular segment in memory, the segment selector for that segment must be present in the appropriate segment register.

The 32-bit EFLAGS register contains a group of status flags, a control flag, and a group of system flags.

The instruction pointer (EIP) register contains the offset in the current code segment for the next instruction to be executed. All Intel Architecture processors prefetch instructions. Because of instruction prefetching, an instruction address read from the bus during an instruction load does not match the value in the EIP register.

The fundamental data types of the Intel Architecture are bytes, words, doublewords, and quadwords. A byte is eight bits, a word is 2 bytes (16 bits), a doubleword is 4 bytes (32 bits), and a quadword is 8 bytes (64 bits). Words, doublewords, and quadwords do not need to be aligned in memory on natural boundaries. (The natural boundaries for words, double words, and quadwords are even-numbered addresses, addresses evenly divisible by four, and addresses evenly divisible by eight, respectively.) However, to improve the performance of programs, data structures (especially stacks) should be aligned on natural boundaries whenever possible. The reason for this is that the processor requires two memory accesses to make an unaligned memory access; whereas, aligned accesses require only one memory access.

An Intel Architecture machine-instruction acts on zero or more operands. Some operands are specified explicitly in an instruction and others are implicit to an instruction. An operand can be located in any of the following places:

- The instruction itself (an immediate operand).
- A register.
- A memory location.
- An I/O port.

6.2.1.0.1 Immediate Operands Some instructions use data encoded in the instruction itself as a source operand. These operands are called immediate operands (or simply immediates). For example, instruction *ADD EAX, 14* adds an immediate value of 14 to the contents of the EAX register. All the arithmetic instructions (except the DIV and IDIV instructions) allow the source operand to be an immediate value. The maximum value allowed for an immediate operand varies among instructions, but can never be greater than the maximum value of an unsigned doubleword integer (2^{32}).

6.2.1.0.2 Register Operands Source and destination operands can be located in any of the following registers, depending on the instruction being executed:

- The 32-bit general-purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP).
- The 16-bit general-purpose registers (AX, BX, CX, DX, SI, DI, SP, or BP). The 8-bit general-purpose registers (AH, BH, CH, DH, AL, BL, CL, or DL).
- The segment registers (CS, DS, SS, ES, FS, and GS).
- The EFLAGS register.
- System registers, such as the global descriptor table (GDTR) or the interrupt descriptor table register (IDTR).

Some instructions (such as the DIV and MUL instructions) use quadword operands contained in a pair of 32-bit registers. Register pairs are represented with a colon separating them. For example, in the register pair EDX:EAX, EDX contains the high order bits and EAX contains the low order bits of a quadword operand. Several instructions (such as the PUSHFD and POPFD instructions) are provided to load and store the contents of the EFLAGS register or to set or clear individual flags in this register. Other instructions (such as the JCC instructions) use the state of the status flags in the EFLAGS register as condition codes for branching or other decision making operations. The processor contains a selection of system registers that are used to control memory management, interrupt and exception handling, task management, processor management, and debugging activities. Some of these system registers are accessible by an application program or the operating system through a set of system instructions. When accessing a system register with a system instruction, the register is generally an implicitly specified operand of the instruction.

6.2.1.0.3 Memory Operands Source and destination operands in memory are referenced by means of a segment selector and an offset. The segment selector specifies the segment containing the operand and the offset (the number of bytes from the beginning of the segment to the first byte of the operand) specifies the linear or effective address of the operand.

The offset part of a memory address can be specified either directly as a static value (called a displacement) or through an address computation made up of one or more of the following components:

- *Displacement* An 8-, 16-, or 32-bit value.
- *Base* The value in a general-purpose register.
- *Index* The value in a general-purpose register.
- *Scale factor* A value of 2, 4, or 8 that is multiplied by the index value.

The offset which results from adding these components is called an effective address. Each of these components can have either a positive or negative (2's complement) value, with the exception of the scaling factor.

6.2.1.0.3.1 Displacement A displacement alone represents a direct (uncomputed) offset to the operand. Because the displacement is encoded in the instruction, this form of an address is sometimes called an absolute or static address. It is commonly used to access a statically allocated scalar operand.

6.2.1.0.3.2 Base A base alone represents an indirect offset to the operand. Since the value in the base register can change, it can be used for dynamic storage of variables and data structures.

6.2.1.0.3.3 Base + Displacement A base register and a displacement can be used together for two distinct purposes:

- As an index into an array when the element size is not 2, 4, or 8 bytes. The displacement component encodes the static offset to the beginning of the array. The base register holds the results of a calculation to determine the offset to a specific element within the array.
- To access a field of a record. The base register holds the address of the beginning of the record, while the displacement is a static offset to the field. An important special case of this combination is access to parameters in a procedure activation record. A procedure activation record is the stack frame created when a procedure is entered. Here, the EBP register is the best choice for the base register, because it automatically selects the stack segment. This is a compact encoding for this common function.

6.2.1.0.3.4 (Index * Scale) + Displacement This address mode offers an efficient way to index into a static array when the element size is 2, 4, or 8 bytes. The displacement locates the beginning of the array, the index register holds the subscript of the desired array element, and the processor automatically converts the subscript into an index by applying the scaling factor.

6.2.1.0.3.5 Base + Index + Displacement Using two registers together supports either a two-dimensional array (the displacement holds the address of the beginning of the array) or one of several instances of an array of records (the displacement is an offset to a field within the record).

6.2.1.0.3.6 Base + (Index * Scale) + Displacement Using all the addressing components together allows efficient indexing of a two-dimensional array when the elements of the array are 2, 4, or 8 bytes in size.

6.2.1.1 Structural Hazards

- *Floating-point Flow Dependency:* On Intel486™ processors, only one floating-point instruction can execute at a time. Subsequent instructions are stalled until the previous instruction executes its latency clock cycles[2].
- *Implicit AGI Conflict:* The instruction for which this interlock is issued has an implicit Address Generation Interlock (AGI) conflict with a previous instruction. This results in a one-clock stall on Intel486™ processors. Even if the instruction incurs more than one penalty, there is still only a one-cycle stall. An Address Generate Interlock (AGI) conflict occurs when a register that is used as the base or index component of an effective address calculation was the destination register of an instruction executed in the immediately preceding cycle. An implicit AGI conflict occurs when the previous instruction implicitly wrote to the destination register. For example, *POP EBP* implicitly writes to the ESP register.
- *Immediate and Displacement:* The instruction for which this hazard is issued has both an immediate operand, and an operand with an address displacement (in the address calculation). This causes a stall on Intel486™ processors. Even if the instruction includes more than one penalty condition, there is still only a one-cycle stall.
- *Index Register:* One of the operands of the instruction for which Index Register Interlock is issued uses an index register in the address calculation. This results in a one-cycle stall on Intel486™ processors. Even if the instruction includes more than one penalty condition, there is still only a one-cycle stall.
- *Prefixed Instruction:* On Intel486™ processors, all prefix opcodes require an additional clock to decode.

6.2.1.2 Data Hazards

- *Data dependency:* (memory, register): Since the Intel486™ does not support forwarding, every data dependency results in an interlock equal to the latency of the preceding instruction
- *Explicit AGI Conflict:* It is caused by an explicit Address Generation Interlock (AGI) conflict with a previous instruction. This results in a one-clock stall on Intel486™ processors. Even if the instruction incurs more than one penalty, there is still only a one-cycle stall. An Address Generate Interlock (AGI) conflict occurs when a register that is used as the base or index component of an effective address calculation was the destination register of an instruction executed in the immediately preceding cycle. An explicit AGI conflict occurs when the previous instruction explicitly wrote to the destination register. For example, add EAX, 3 explicitly writes to the EAX register. On the Intel486™ processor, AGI conflicts occur only when the register that is used as the base (not the index) component of an effective address calculation was the destination register of an instruction executed in the immediately preceding cycle.
- *Partial Register Reference:* This happens when an instruction reads from a large register (EAX) after the previous instruction wrote to a partial register (AL, AH, AX) that is contained in the large register. This causes a one-cycle stall on Intel486™ processors. This applies to all register pairs involving either a larger register with any of its partial registers, or two partial registers in the same set. Examples of larger registers with one of its partial

[2]Execution latency is the number of clock cycles an instruction takes to compute its result.

registers are:, AX with EAX, BL with BX, and SI with ESI. Examples of two partial registers in the same set are: AL with AH, and CL with CH. Even if the instruction includes more than one penalty condition, there is still only a one-cycle stall.

6.2.1.3 Control Hazards

- *Jump not Taken:* On the Intel486™ branches are predicted as always not taken: any branch that is taken forces a pipeline flush, and causes a two-cycle stall.

6.2.2 microSPARC™-II

SPARC is a CPU *Instruction Set Architecture* (VISA), derived from a reduced instruction set computer (RISC) lineage; it was designed as a target for optimizing compilers and easily pipelined hardware implementations [17].

The microSPARC™-II CPU is a highly integrated, low-cost implementation of the SPARC version 8 RISC architecture with a PCI interface [18]. High performance is achieved by the high level of integration, including on chip instruction and data cache, built-in DRAM controller and PCI local bus controller.

A SPARC processor logically comprises an integer unit (**IU**), a floating-point unit (**FPU**), and an optional co-processor (**CP**), each with its own registers. This organization allows for implementations with maximum concurrency between integer, floating-point and co-processor instruction execution. Generally, all of the registers are 32-bit wide; instruction operands are generally single register, register pairs or quadruples.

The IU contains the general-purpose registers and controls the overall operation of the processor. An implementation of the IU may contain from 40 to 520 general-purpose *r* registers. This corresponds to a grouping of the registers into 8 *global* registers, plus a circular stack of 2 to 32 sets of 16 registers each, known as *register windows*. At a given time, an instruction can access the 8 globals and a register window into the *r* registers. A 24-register window comprises a 16-register set –divided into 8 *in* and 8 *local* registers– together with the 8 *in* registers of an adjacent register set, addressable from the current window as its *out* registers.

The FPU has 32 32-bit floating-point *f* registers; double precision values occupy an even-odd pair of registers, and quad precision values occupy a quad-aligned group of 4 registers. Floating-point load/store instructions are used to move data between the FPU and memory, the memory address being computed by the IU.

SPARC is a load/store architecture, since the only instructions that access memory are load and store. Integer load and store support byte, halfword (16-bit), word (32-bit) and doubleword (64-bit); there are version of load and store that perform sign extension when loading a byte or an halfword into a 32-bit register. Floating point load and store can access memory only in word and doubleword modes.

The microSPARC™-II implements a SPARC version 8 architecture; the IU presents a 5-stage pipeline, a 136-register register file supporting 8 register windows, hardware implementation of IMUL and IDIV, 4-deep instruction queue supporting instruction prefetching, delayed branch execution (*branch folding*). The pipeline stages perform the following actions:

- **F** (Instruction Fetch), fetches instructions either from the 4-deep prefetching queue or directly from the cache.

- **D** (Instruction Decode), decodes the instructions and reads the necessary operands, which may come from the register file or from internal data bypasses (forwarding). Addresses are computed for call and branch in this stage.

- **E** (Execute), performs ALU, logical and shift operations. Addresses for memory operation or jumps are computed in this stage, while a store accesses the register file through a special port to execute the store operand read.
- **W** (Write), accesses the data cache: a loaded data is available at the end of this stage.
- **R** (Result), loads the result of any ALU, logical shift or cache read operation into the register file.

To optimize branch execution, the IU pipeline supports a double instruction issue (1 branch, 1 other); branches are handled in two ways: a branch may be folded with its delay slot instruction or it may flow down the integer pipeline. An always-taken prediction scheme is supported, thus allowing the fetch of the target instruction in the D-stage of the branch/delay slot pair. If the branch is not taken, the instruction fetched from the target is ignored, and a new instruction (called delay slot+1) has to be fetched. Table 6.1 summarizes the cycles taken for a branch.

Branch	**Taken**	**Not Taken**
Folded	0	1
Not Folded	1	1 or 2

Table 6.1: Cycles for a branch

The microSPARC™-II floating-point unit (FPU) serves multiple purposes: it executes floating point instructions, detects data dependencies among those instructions and handles floating-point related exceptions. The FPU consists of a fast multiply unit, a separate core for all other operations and state machines to control the two data-paths. Operations may be executed in parallel in the two data-paths.

A 3-deep floating-point instruction queue serves as interface between the IU and the FPU: this queue allows out-of-order issue but in-order completion of floating-point instructions; the IU is responsible of fetching any floating-point operation and passing it to the FPU, which can start execution whenever the needed unit is available. Loads and stores are executed in cooperation with the IU, because the FPU has no direct access to the data cache. The following sections describe the possible hazards that may arise during a program execution.

6.2.2.1 Structural Hazards

As we discussed in section 6.1.1, structural hazards arise from resource conflicts; in the microSPARC™-II there are essentially two different cases: an instruction having an high latency in the pipeline prevents the immediate execution of any other instruction; otherwise, a conflict on a limited hardware resource arises. Thus, a structural hazard arises when:

- an instruction traversing the integer pipeline has a latency higher than 5 clock cycles: this means that one or more pipeline stage are occupied by the same instruction for more than one cycle;
- the floating-point queue is full, preventing the execution of other incoming floating-point instructions;
- the register file port is used by an instruction, preventing the IMUL or the IDIV from executing, until all the register file ports are unused.

6.2.2.2 Data Hazards

Data hazards are generally bypassed with forwarding, but in some cases the data path is unable to prevent a stall; this cases correspond to the following situations:

- a load is immediately followed by an instruction that uses the loaded data;
- a call is immediately followed by an instruction that uses the register *r[15]*;
- a read state register is followed by a dependent operation;
- a floating-point store is waiting for data being produced by an operation in queue;
- a floating-point operation in queue uses the same register (source or destination) as a floating-point load (WAR or WAW hazard);
- a floating-point load is followed by a floating-point store on the same register;
- Floating-point operations in queue present some data-dependencies on their operands.

It is worth noting that the forwarding hardware is implemented only for integer operations, thus floating-point operations in queue, even if issued out of order, have to wait for their operands being ready. Other hazards may be avoided by specific optimizations at compile time.

6.2.2.3 Control Hazards

The SPARC architecture uses a delayed canceling branch technique (see section 6.1.3), allowing the parallel execution of the branch/delay slot pair (branch folding); however, branch folding is possible only under certain conditions verifiable at run-time; besides, an always-taken prediction scheme is implemented. Control hazards arise when a branch is not taken, resulting in a misprediction of the instruction to be executed. The following situations can be observed:

- if the branch has been folded, there is no need to stall the pipeline, and the fetched target instruction is ignored;
- if the branch has not been folded but the delay slot+1 instruction is available in the prefetching queue or in streaming from the cache, there is no need to stall the pipeline, and the fetched target instruction is ignored;
- if the branch has not been folded and the delay slot+1 instruction has to be fetched, the fetched target instruction is ignored and there is one cycle stall.

The canceling branch technique does not modify the occurrence of control hazards: it simply states whether the delay slot instruction have to be annulled in case of misprediction or not. However, it allows the compiler to have more freedom in filling the delay slot with a valid instruction.

7 Developed Simulators

This chapter describes in detail the simulators that have been produced using the proposed methodology and the provided tools and libraries. In particular, two architectures were considered: the Intel486™ and microSPARC™-II.

7.1 microSPARC™-II

The microSPARC™-II is a RISC processor featuring a relatively simple instruction set on a complex data path (see Chapter 6). Much of the simulator complexity resides then in the TrIBeS simulated architecture, while the atomic micro-compiler is almost straightforward. The next sections will describe the inner structure of both.

7.1.1 The microSPARC™-II TrIBeS library

The simulated architecture is modeled over the physical one, splitting the processor into nine *functional units* (FUs). Each of these functional units represent a particular pipeline stage of the microSPARC™-II datapath. The connection between functional units is sufficient to model the processor pipeline, all the other features are either embedded into specific functional units or modeled by resource access. Between each couple of functional unit there is an instruction buffer (see Chapter 5) that works as a master-slave register between pipeline stages. Figure 7.1 shows the simulated architecture: functional units are displayed as double-line squares, resources as white squares and instruction queues as white rectangles. Dashed lines connecting the functional units with resources indicate that the functional unit has access to the resource, other connections indicate the possible flows of instructions

7.1.1.1 General Structure

As shown by the block diagram of figure 7.1, instructions flowing into the simulator can follow two paths: the integer datapath and the floating point datapath. The DecodeUnit takes care of issuing instructions in the correct datapath, where, once arrived, instructions are executed. Another interesting aspect shown in the diagram is that the microSPARC™-II can fetch up to two instructions per clock cycle: this may happen if and only if one of the instructions is a branch and the other is an integer ALU operation. For this reason, the integer datapath has a width of two instructions for all its length, from the input to the output queue. Moreover, the floating point datapath can hold up to two instructions per clock cycle since it integrates a 2-stage pipelined multiplier and a generic ALU that work concurrently.

Each functional unit has access to a set of resources: between these there are some flags (like the FoldingFlagResource) that are used to model peculiar architectural characteristics and subtle hazards of the microSPARC™-II (see Chapter 6). Dotted lines in the diagram indicate the connection between a resource and a functional unit.

It is worth noting that the InputQueue reads instructions from an input file as it receives requests, ignoring the system clock. In this way an arbitrary number of instructions can be read in a clock cycle. This is used to simulate the data stream between processor cache and the prefetch buffer of the microSPARC™-II. The output queue works exactly in the same way: every received instruction is simply printed out as it is received.

The functional units and resources are described in detail in the following paragraphs.

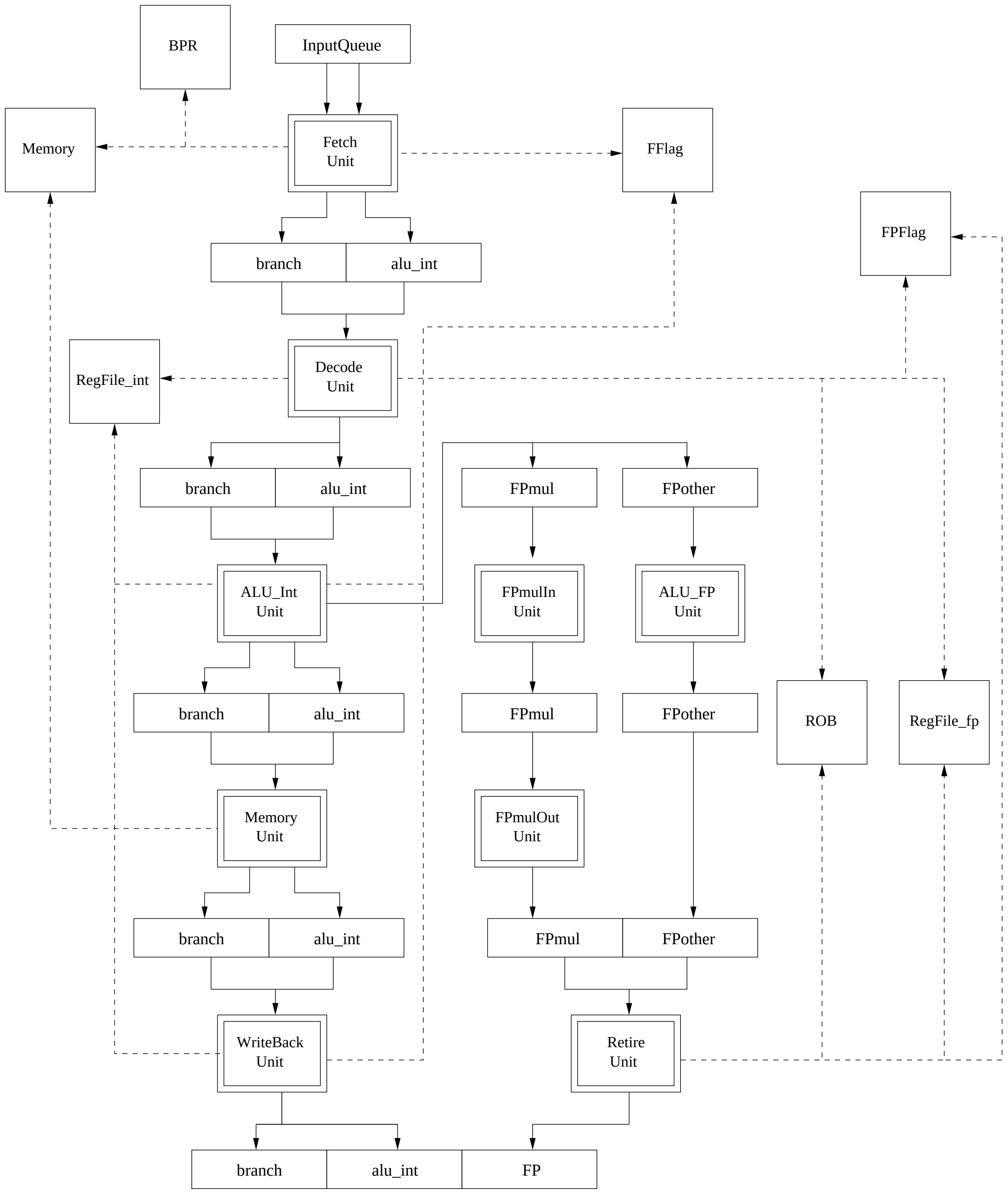


Figure 7.1: TrIBeS simulated architecture for the microSPARC™-II pipeline

7.1.1.2 FetchUnit

This unit takes care of loading instructions from memory and sending them to the decode unit. Before being actually fetched, instructions are loaded in a 3-slot prefetch buffer. During simulation, instructions are extracted from the InputQueue, and possible delays caused by memory access are emulated by accessing the MemoryResource, which returns the number of clock cycles that the unit has to stall before continuing.

Algorithm 4 Execution Cycle of the FetchUnit

```
while prefetch buffer not full do
  Ask the InputQueue for an instruction;
  Access MemoryResource;
  if MemoryResource returns $d > 0$ then
    Stall for $d$ clock cycles;
  end if
end while
Fill the internal queue (2 instructions);
for all Instructions $s$ in the internal queue do
  for all micro-instructions $m$ in $s$ do
    if $m$ is a branch then
      if FFlag $> 0$ then
        send out the $s$;
        put the instruction back in the prefetch buffer;
        return;
      else
        delayslot := true;
        FFlag++;
      end if
    else if $m$ is a use instruction then
      FFlag$++$;
    else if delayslot = true then
      FFlag$--$;
      delayslot := false;
      if annulled then
        remove all remaining micro-instructions;
      end if
      send out the instruction and return;
    else
      send out the instruction and return;
    end if
  end for
end for
return;
```

In the TrIBeS model, the FetchUnit handles branches also: for every branch read, the FetchUnit asks the BranchPredictionResource if the branch was correctly predicted or not, and stalls the processor in case of misprediction. An important consideration concerns the use of instruction traces: the instruction following a branch is always the right one, whether the branch was taken or not. Hence the pipeline does not need to be flushed, bubbles (i.e. null instructions) are generated to emulate the stall instead.

The FetchUnit takes care of handling the branch folding technique also, this is a rather complex method to have branches executed in parallel with other instructions into the integer datapath. The fetch unit checks if the branch folding conditions are met, using the FoldingFlagResource (FFlag in the diagram). If the flag is up the folding can not take place, and the instruction following the considered branch is not fetched in the same clock cycle. Some instruction raise the FFlag (and lower it after execution is completed) to emulate microSPARC™-II architectural limitations concerning the branch folding, for example branches, meaning that two consecutive control transfer instructions are not foldable.

Another condition branches might encounter is the annulling state; when a branch is annulling, its delay slot must traverse the pipeline without any effect. To this purpose, when a branch is annulling (the annulling bit is set), the instruction in its delay slot is stripped off of all its micro-instructions, and traverses the datapath as a null instruction.

The algorithm executed by the functional unit is Algorithm 4

7.1.1.3 DecodeUnit

The DecodeUnit takes care of loading operands by checking and setting locks on registers. The DecodeUnit receives up to two instructions per clock cycle, reads their micro-instructions checking the lock of the source registers and locking the destination ones. When it encounters a `require` micro-instruction, it checks destination: if it is the DecodeUnit itself, the instruction is stalled for the required number of cycles, otherwise it is sent in the next destination queue, namely the AluIntUnit one.

The DecodeUnit has access to three flags: FFlag, FPFlag and FPComp. While the first is used to unlock branch folding (locked by access to special registers, special instructions, etc.), the others are used to detect a couple of subtle hazards of the microSPARC™-II architecture, specifically the FCMP signal and the IMUL/IDIV hazards (see section 6.2.2.1). If the flags are up, the unit stalls until they are down again.

The algorithm conducting the DecodeUnit execution is shown by Algorithm 5

Algorithm 5 Execution Cycle of the DecodeUnit

```
Fill the internal queue (2 instructions);
for all Instructions $s$ in the internal queue do
  for all micro-instructions $m$ in $s$ do
    if $m$ is a read on register $r$ then
      if $r$ is locked then
        stall until it is free;
      end if
    else if $m$ is a lock then
      if $r$ is locked then
        stall until it is free;
      end if
      lock $r$;
    else if $m$ is a require unit $u$ for $n$ cycles then
      if $u ==$ DecodeUnit then
        stall for $n$ cycles and break;
      end if
      send out $s$;
    end if
  end for
end for
```

7.1.1.4 AluIntUnit

The microSPARC™-II AluIntUnit takes care of dispatching instructions to the subsequent units. To do so, it relies on the `require` micro-instruction, that tells the unit in which queue the instruction has to be put. There are three possible output queues: MemoryUnit, FpMulUnit and FpAluUnit. MemoryUnit acts as the default queue, where instruction with no specified destinations are sent. The algorithm is very similar to Algorithm 5, but the `require` destination is used to choose the appropriate output queue.

The AluIntUnit is also connected to the FFlag and the integer register file. In fact, every instruction that uses the forwarding unlocks its registers in the AluIntUnit (this means zero wait states), and some instructions lower the folding flag when they exit the AluIntUnit. This means that this unit recognizes `require`, `write` and `use` micro-instructions: the first is used for destination definition, the second for register unlocking and the last to lower the FFlag.

When the AluIntUnit sends instructions to the floating point datapath, it also inserts instructions into the ReorderBufferResource. This resource is used to emulate the 4-slot floating point instruction queue of the microSPARC™-II and its in-order-completion of FP instructions. When an instruction requires to be executed in the floating point datapath, it is registered in the 4-slot re-order buffer and then sent to the corresponding unit. If the re-order buffer is full, the instruction (and consequently the entire integer pipeline) is stalled until there is a free slot.

7.1.1.5 MemoryUnit and WritebackUnit

These are very simple units that do little more than passing instructions from input to output. MemoryUnit takes care of memory access (note that the SparcV8 is a load-store architecture), and accesses the MemoryResource whenever a `load` or `store` micro-instruction is encountered. The unit stalls in case memory access should fail. This unit also unlocks registers in case the forwarding has been disabled for some reason, in order to obtain a single-cycle stall in case of data dependency between instructions.

The WriteBackUnit in the physical architecture is the unit that accesses the integer register file to write instruction results while in our simulator it is used to detect hazards related to register file port usage. In practice, an access to the register file for writing (`write` micro-instruction) uses a write port, an access for reading (`read` micro-instruction) a read port. If all ports are already used in the current clock cycle, the unit stalls for a cycle[1].

7.1.1.6 FPMulUnit and FPAluUnit

These units recognize only the `require` micro-instruction, passing instructions from input to output with a given latency in clock cycles. The FPMulUnit is pipelined, i.e. it is formed by two consecutive functional units, and its two stages have a 3-cycle and a 2-cycle latency respectively.

7.1.1.7 RetireUnit and ReorderBufferResource

The retire unit takes instructions from the floating point datapath and outputs them to the output queue for printout. It unlocks floating point registers an lowers FPFlag and FPComp

Algorithm 6 Execution Cycle of the RetireUnit

```
1: fill the internal queue (3 instructions);
2: for all instructions $s$ in the internal queue do
3:   if $s$ is the first entry in the reorder buffer then
4:     unlock all registers locked by $s$;
5:     lower FPFlag or FPComp if needed;
6:     output $s$ and return;
7:   end if
8: end for
```

[1]The microSPARC™-II features an integer register file with three read ports and a single write port.

also. Its main characteristic is the 3-deep internal queue, necessary for instruction reordering before output; at every cycle the unit asks the reorder buffer which instruction in its internal queue is the oldest and consequently, passes it to the output queue. The reorder buffer is queried based on the instruction progressive identification number: the buffer holds an ordered list of the instructions that entered the floating point datapath and returns true only if the given identification number is on top of the list; in this case, the matched entry is removed. Algorithm 6 details the inner workings of this unit.

7.1.2 The SparcV8 assembly compiler

The trace micro-compiler receives as input an assembly trace of a program execution and compiles it into the TrIBeS format. Tracing is obtained via the Shade toolset [4], a performance analysis package distributed by Sun. The trace format is simply a text file containing the disassembled instructions of the executed code, in the order in which they were actually executed. The file is parsed via the `atomic sparc library`: this library is a scanner linked with a parsing table, in which instructions (as they are read from the trace) are associated to the corresponding TrIBeS micro-instructions. Figure 7.2 shows an excerpt of such a table. As shown, each

```
| T_PREFIX T_NOP                    {START(28, 5); [...] END;}
| T_PREFIX T_SETHI  imm ',' reg     {START(29, 5); [...] END;}
| T_PREFIX T_SET    imm ',' reg     {START(30, 5); [...] END;}
```

Figure 7.2: Translation table excerpt

instruction has a starting point and is tagged by an identification number (the first number in the `START` clause) and a nominal latency in clock cycles (the second number). What follows is a list of microinstructions that describe the instruction activities in term of resource use and latency in the various microSPARC™-II functional units. This means that the library must be aware of the internal structure of the corresponding TrIBeS library. The instruction trace format of the SparcV8 assembly is explained by figure 7.3. As shown, the trace is made of five fields:

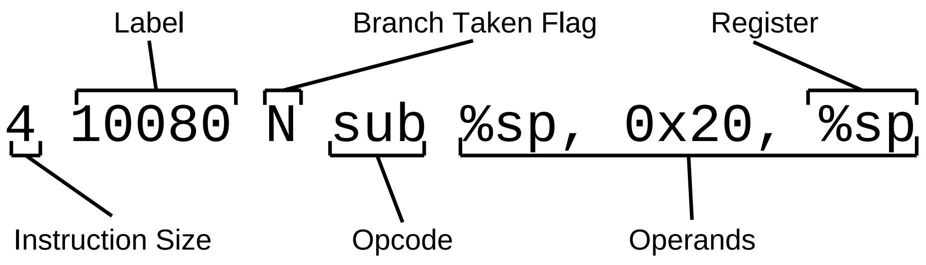


Figure 7.3: SparcV8 instruction trace format

Instruction Size It is the number of bytes the instruction occupies in memory

Label It is the memory address where the instruction is located

Branch Taken Flag This flag is used only if the considered instruction is a control transfer instruction (CTI). A `T` value means a branch was taken, a `N` means that it was not.

Opcode It is instruction opcode in plain text format

Operands Each instruction may have different numbers and types of operands, involving immediate numbers, registers and memory addresses. Table 7.1 lists all possible SparcV8 operands, and their syntax. Instructions may have zero to three operands. For instructions with more than one operand, the last is the destination of the operation.

Operand Type	Meaning
`%r0`	Direct access to a register
`%hi(%r0)`	Direct access to a register (high bits)
`%lo(%r0)`	Direct access to a register (low bits)
`0x20`	Immediate value
`800142`	Direct memory address
`%hi(800142)`	Direct memory address (high bits)
`%hi(800142)`	Direct memory address (low bits)
`[ %o0 + 0x120 ]`	Indirect access (register +/- immediate)
`[ %r0 + %l7 ]`	Indirect access (register +/- register)

Table 7.1: SparcV8 operand types

The operands are recognized by the scanner and their value is passed to the parser with the corresponding token. Each instruction is composed using a set of tokens and parser rules: when an instruction is recognized, the associate rule is called. Rules are formed by the micro-instruction output (plus some conditional statements), obtained using a set of macros. Such macros include register reading and writing, latency exploiting and so on.

The SparcV8 assembly can be classified into five types of instructions, according to how instructions are micro-compiled. These groups are Integer ALU Instructions, Control Transfer Instructions, Integer Load/Store Instructions and Floating Point Instructions. Each of these classes has a set of micro-code templates that describe the their activity in TrIBeS. As an example, figure 7.4, shows a template for an Integer ALU Instruction with 2 operands, see Appendix B for the entire template list.

Micro-code	Unit	Description
`INSTRUCTION_START <class> <latency>`	Input	instruction initialization
`: read <sreg1> REG:IntRegFile`	Decode	check <sreg1> lock state
`: write <dreg> REG:IntRegFile`	Decode	lock <dreg>
`: require <latency> AI:AluInt`	Decode	send to AluInt
	AluInt	set Instruction latency
`AI:AluInt write <reg> REG:IntRegFile`	AluInt	unlock <reg> (forwarding)
`WB:WriteBack write <reg> REG:IntRegFile`	WriteBack	write <reg> (use port)
`INSTRUCTION_END`	Output	output values

Figure 7.4: Template for a 2-operand instruction

The instruction is compiled by TrIBeS into a sequence of micro-instructions: these are executed in the exact order in which they are written. Each functional unit reads the first micro- instruction: if unit recognizes the micro-opcode, it executes the micro-instruction, then removes it and starts reading the next one; otherwise, the entire instruction is passed to the output buffer. In the example, the instruction is initialized, then is passed to the Fetch unit. Fetch reads the first micro-instruction but can not execute it, so the instruction is passed forward to the Decode Unit. Decode can execute both `read` and `write`: waits for the source register to unlock and locks its destination register. Finally, Decode reads the `require` micro-instruction (but does not delete it) and sends the instruction to the AluIntUnit. Here the instruction for as many clock cycles as specified by the `require` micro-instruction, and then the AluIntUnit unlocks the locked register. This way the forwarding technique is properly simulated. Finally, the instruction enters the WriteBack unit, which commits register writes into the register file occupying one of its input ports.

The behavior of the instructions may change depending on the data the instruction uses. To include this behavior into the microcode, conditional statements are inserted in the rules triggered by instructions. As an example, the move instruction disables forwarding if it uses one of the special registers:

```
T_PREFIX T_MOV   reg ',' reg  {if($3 > R31 ) { [...]
                                     WRITE_INT(MEMORY_NAME, $5);  [...]
                               } else { [...]
                                     WRITE_INT(ALUINT_NAME, $5);} END;$}
```

This means that if the register is higher than R31, i.e. a special register, the instruction unlocks the register in the WriteBackUnit, otherwise in the AluIntUnit, emulating the disabling of forwarding (this means a one-cycle stall in case of a data dependency).

7.1.3 Simulator validation

To verify the correctness of the simulator engine, 12 benchmarks, taken from different application domains, have been simulated and the total estimated execution time has been compared with the actual execution time [8]. The actual execution time has been measured by running the

	Error (%)	
Code	**w/o Memory**	**w/ Memory**
adpcm	-10.47	-9.23
gsm	-11.87	-7.48
lagrange	-4.79	-3.01
qsort	-8.74	+1.39
g723	-4.20	-3.00
fdct	-10.68	-9.48
crc16	-0.83	+2.37
md5	-11.55	+5.15
rle	-2.79	-1.59
bsort	-2.64	-0.11
matrix	-34.27	-3.38
Overall	**9.33**	**4.19**

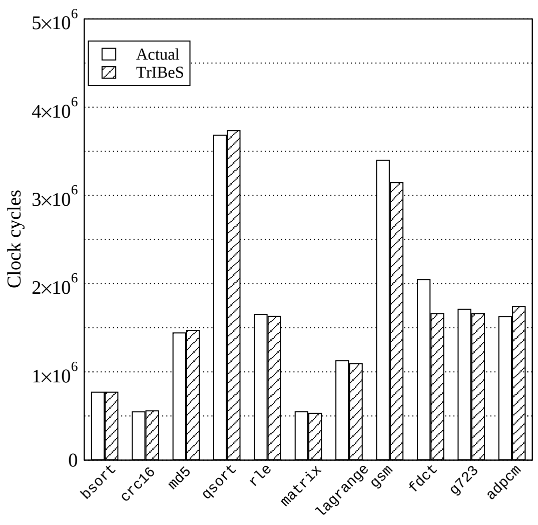


Figure 7.5: Simulator accuracy

benchmarks on the target platform configured to enable microstate accounting and to use high-resolution timers [27]. Each benchmark has been run with different sets of input data, leading to the overall results reported in figure 7.5 both considering and ignoring memory effects. The delays caused by cache misses, write-buffer overflows and memory refresh have been currently determined separately using the SUN Microsystems proprietary simulator uni_per [16]. The overall average error obtained considering memory effects also is 4.19% with a standard deviation of 4.72%. This proves that the simulator has a satisfactory accuracy.

7.2 Intel486™

The Intel486™ is a CISC processor with a simple data path, but with a very large and complex instruction set (see Chapter 6). Much of the simulator complexity resides in the atomic microcompiler, while the TrIBeS simulated architecture is almost trivial. More details are given in the following sections.

7.2.1 The Intel486™ TrIBeS library

The simulator is made of a set of five functional units, whose connections model the internal pipeline of the processor. Figure 7.6 shows the simulator structure, that is remarkably simpler than the one developed for the Sparc.

7.2.1.1 General Structure

The Intel486™ is a straight 5-stage pipeline, with the addition of a floating point unit that works in parallel with the integer one. The unit that acts as a dispatcher between the floating point and the integer ALU is the second decode stage. This stage takes care of locking registers also, while the WriteBackUnit unlocks them when instruction execution terminates, meaning that there is a one-cycle stall for every data dependency (the Intel486™ does not have forwarding).

It is also worth noting that, given that the processor has micro-coded instructions executed on a RISC core, there is no mention of instruction coding except for the ROMFlag (which represents the locking of the ROM microcode operated by floating point instructions). This is due the fact that it is not really necessary to simulate a target architectures in all its details, since what really matters is only the instruction timing. Hence, the RISC core can be seen as a black box represented by the AluIntUnit.

7.2.1.2 FetchUnit

Just like the Sparc, the Fetch unit fills a prefetch buffer (in this case with 4 slots), then fetches one instruction at a time. If the instruction is a branch, the unit asks the BranchPredictionResource (BPR) if the branch was predicted or not, with the always-not-taken prediction scheme. If the branch was not predicted, a 2-cycle stall is introduced, with two bubbles (null instructions) flowing in the pipeline. No other stalls are considered in this stage.

7.2.1.3 DecodeUnit and Decode2Unit

The two decode units recognize the `require` micro-instruction, in the DecodeUnit for exploiting instruction latency and in the Decode2Unit for dispatching instruction to both AluIntUnit and AluFpUnit. Decode2 checks the register locks before dispatching the instructions and, in case of a floating point instruction, locks itself for three cycles, in order to emulate the locking of the micro-code ROM. Since this is not a load-store architecture, the Decode2Unit also performs memory access to load operands, stalling whenever the memory is not ready.

All the other functional units are very similar to the corresponding microSPARC™-II units (see section 7.1 for details).

7.2.2 The 80x86 assembly compiler

Much of the complexity of the simulator resides in the micro-compiler. In this case, there are many stalls that can be detected without dynamic analysis, just by looking at the instruction structure and operands. In addition, many instructions have different latencies, depending on the type of data and address the use. These stalls and delays are detected directly by `atomic`, via many conditional statements in the translation table.

The instruction format of the 80x86 assembly (shown in figure 7.7), is similar to the SparcV8 with the addition of optional instruction prefixes (to add special functionality to some instructions) and suffixes (to specify operand size). Tracing is obtained via the standard ptrace command. All other fields have the same meaning as in the SparcV8 assembly. Instruction prefixes are used to have one instruction repeated for a given number of times (REP and similar prefixes):

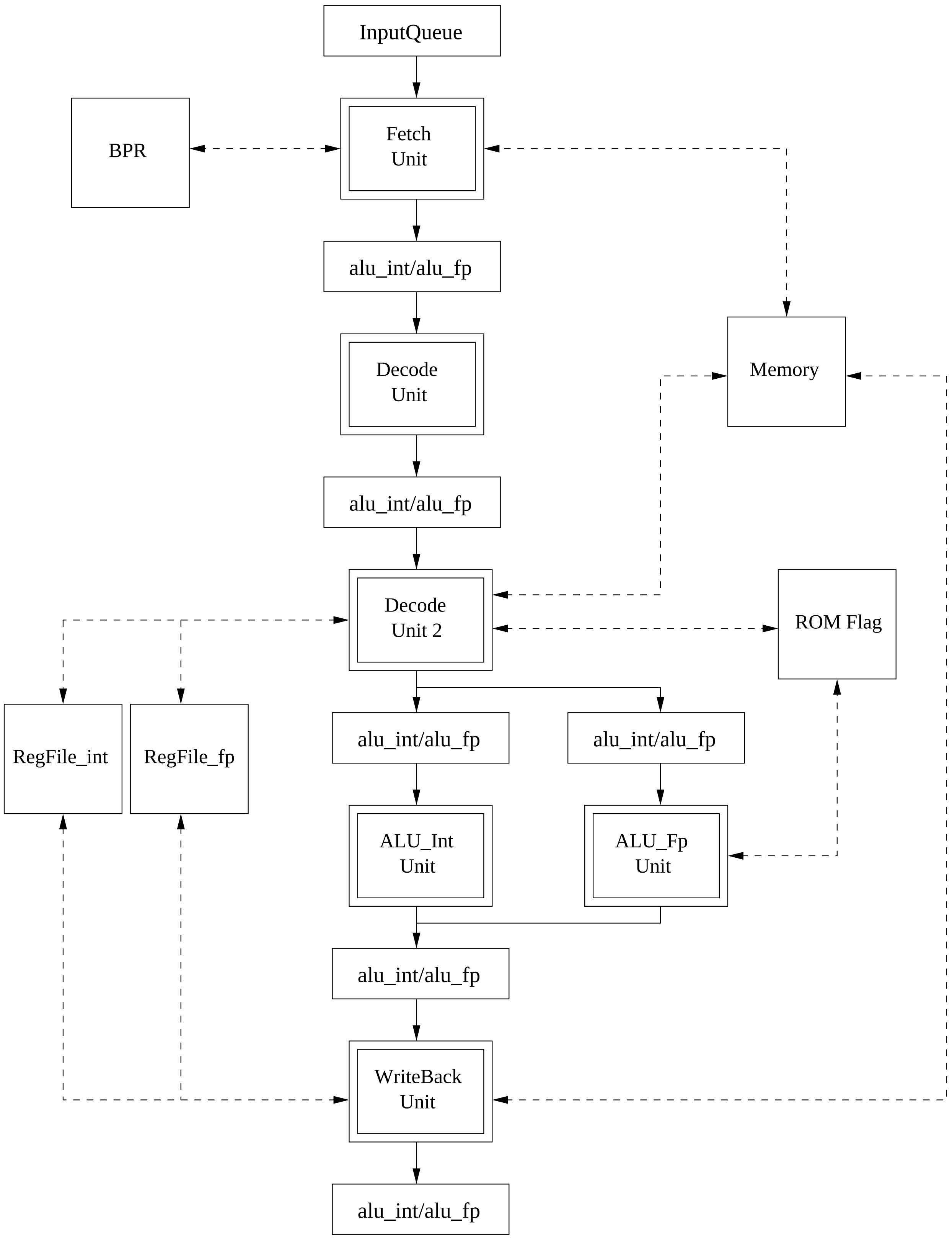


Figure 7.6: TrIBeS simulated architecture for the Intel486™ pipeline

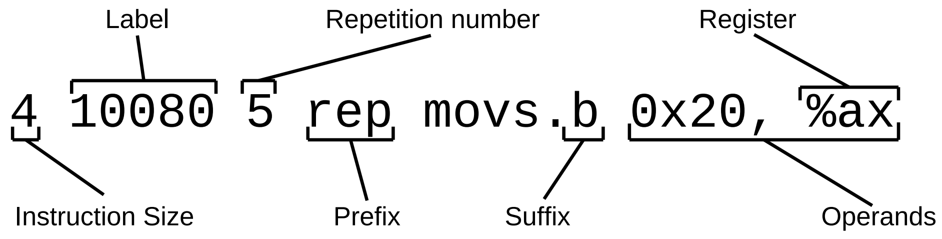


Figure 7.7: 80x86 instruction trace format

`atomic` has to recognize the number of repetitions and set instruction latency accordingly (multiplying the latency in the execution stage for the number of repetitions). The branch taken flag field is used to hold the number of repetitions of prefixed instructions if necessary. This is possible since branch instructions never have prefixes.

The stalls recognized directly by `atomic` are:

Prefixed Instruction A prefixed instruction requires an additional clock to decode.

Immediate and Displacement Whenever there is an immediate operand followed by a displacement one, there is a one cycle stall.

Index Register If an index register is used, there is a one-cycle stall.

All these stalls are mutually exclusive, this means that if one of them occurs, the others can not. The implementation consists in conditional statements inserted in the rules triggered by instruction sensitive to these hazards. These rules check the condition and then add, `require` instructions that increase the global instruction latency in some functional units, simulating stalls. As an example, for a prefixed instruction would be added:

```
REQUIRE( DECODE1_NAME , 2 , DECODE1_NAME );
```

This simulates the extra cycle needed to decode prefixed instructions, delay spent into the first decode unit. For instructions that use immediates and displacements or index registers there would be instead:

```
REQUIRE( DECODE2_NAME , 2 , DECODE2_NAME );
```

This is a one-cycle stall, in the Decode2Unit.

The operands used by the 80x86 assembly are much more than those used by the SparcV8. This is due to the addition of memory access to every instruction (instead of having specific load and store instructions). The operand types are reported in table 7.2.

The complex system of accessing memory with Base, Displacement, Index and Scale is parsed using a data structure that is filled with the registers and address accessed by each instruction; this structure is then used by the rules to build the appropriate sequence of micro-instructions (usually a combination of `load`, `store`, `write` and `read` micro-instructions). As a sample rule for micro-code translation, figure 7.8 reports a prefixed instruction with two operands.

7.2.3 Simulator validation

To verify the correctness of the simulator engine, 9 benchmarks, taken from different application domains, have been simulated and the total estimated execution time has been compared with the actual execution time.

The actual execution time has been measured by running the benchmarks on the target platform using the standard operating system timers. Each benchmark has been run with different sets

Operand Type	Meaning
`%eax`	Direct access to a register
`0x123416`	Displacement access to memory
`0x123416(%eax)`	Base + Displacement
`0x123416(%ebx,%eax)`	Base + Index + Displacement
`0x123416(%ebx,%eax,4)`	Base + (Index * scale) + Displacement
`0x123416(,%eax,4)`	(Index * scale) + Displacement
`0x123416(,%eax)`	Index + Displacement
`$0x20`	Immediate value
`gs:0x2123`	Segment modifier + displacement

Table 7.2: 80x86 operand types

```
T_MOVS suffix  imm ',' mem  {START(140,4+REPTIME(7));
/* Check prefix stall   */   ISPREFIXED( DECODE1_NAME );
/* Check displacement   */   ISDISP( DECODE1_NAME );
/* Check index register */   ISINDEX( DECODE1_NAME );
[...]
/* Latency=7*rep        */   REQUIRE_INT(ALUINT_NAME,REPTIME(7));
[...]
/* Store result         */   STORE(WRITEBACK_NAME,0); END;}
```

Figure 7.8: Rule for a prefixed instruction

Code	Error (%)
adpcm	-14.88
lagrange	-6.41
qsort	-0.60
fdct	-13.72
crc16	+2.74
md5	-5.90
rle	-13.31
bsort	-15.72
matrix	-27.88
Overall	**12.22**

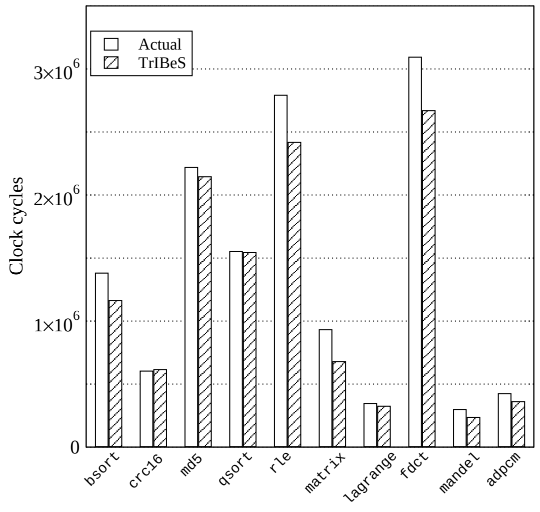


Figure 7.9: Simulator accuracy

of input data, leading to the overall results reported in figure 7.9. Current results memory effects, since no free tool to estimate such effects is known to be available. The overall average error obtained without considering memory effects is 12.22%, meaning that the simulator has a satisfactory accuracy, and that this error can be certainly reduced by introducing memory effects also.

8 Experimental Results

This chapter describes the setup on which the methodology proposed in Chapter 4 has been tested and the results of such experimental activities. Section 8.1 describes the taxonomy classes (see Section 3.2.1) used in the previous and in the present work Section 8.2 describes the methodology and the results obtained with the previous inter-instruction effects model [2]. Section 8.3 describes the results of the tuning process obtained implementing the new model with the tools described in Chapter 5, and finally Section 8.4 describes the results of the validation process and compares them with the previous ones. The experimental activity has been carried out on a microSPARC™-II and an Intel 486™ machine.

8.1 Taxonomy Class Definition

In order to have comparable results with those obtained with the model presented in [2], it is necessary to maintain the same instruction taxonomy, so that possible differences in the results are not biased by different choices in the classification process. As mentioned in definition 1, the dynamic behavior of single instructions drives the classification process by means of an equivalence relation $\mathcal{R}$. In the original model, parallelism was not taken into account, thus classification was made only with respect to inter-instruction effects like execution hazards. There are three classes of hazards [12]:

Structural Hazards they arise from resource conflicts when the hardware cannot support all possible combinations of instructions in parallel execution.

Data Hazards they arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the execution.

Control Hazards they arise from the parallel execution of branches and other instructions that change the program counter.

Hazard types are here conventionally named *S–type*, *D-type* and *C-type* hazards respectively. The following definition holds:

Definition 8 *Given an instruction set I,a* hazard-based partition

$$\mathcal{I}_H = \{I_{H,j}, j \in \{0,1\}\} \subset 2^I$$

distinguishes instructions that may cause H-type hazards and those that may not, having $H \in L = \{S, D, C\}$. The classes $I_{H,0}$ and $I_{H,1}$ are defined as:

$$\begin{aligned} I_{H,0} &= \{s \in I | s \text{ cannot cause an H-type hazard}\} \\ I_{H,1} &= \{s \in I | s \text{ may cause an H-type hazard}\} \end{aligned}$$

and constitute a partition of I by design.

This definition explains the concept that an hazard depends on an ordered pair of instructions: we consider the first instruction in the pair as the *cause* of the hazard. Definition 8 can be used to define the equivalence relation needed by the taxonomy:

Definition 9 *Given two instructions s_i and s_j, the relation $\mathcal{R}$ is defined as:*

$$s_i \mathcal{R} s_j \iff (\forall l \in L)\ (s_i, s_j \in H_{l,0} \vee s_i, s_j \in H_{l,1})$$

This definition explains that two instructions are related if and only if they belong to the same set of all H-type partition. This relation is evidently an equivalence relation, and the demonstration is left out for the sake of brevity. R relation $\mathcal{R}$ allows to generate the taxonomy needed for the experimental phase; in particular, the taxonomy results in eight different classes, conventionally numbered from 0 to 7.

8.2 Original Model Results

This section describes the methodology for tuning the model proposed in [2]: this methodology is intended to provide the estimates for the overhead introduced by inter-instruction effects. The

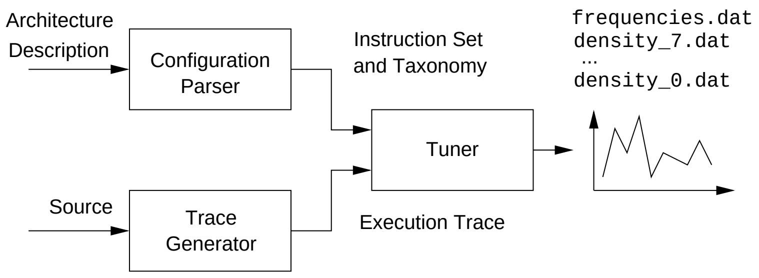


Figure 8.1: Experimental flow

tuning process of this methodology is summarized in figure 8.1; it can be split into four main activities:

Trace Generation :
all possible dynamic information have to be extracted from the examined programs; this is achieved by actually executing such programs and by saving the resulting assembly trace.

Configuration Loading :
in order to abstract from specific architectures, a configuration file corresponding to the examined one has to be loaded, in order to translate architecture-specific information into architecture-independent ones.

Model Tuning :
the trace is analyzed by the means of the loaded configuration file to obtain the estimation of class probabilities and delay variables (see Section 2.3.2).

Model validation :
the obtained results are compared with the actual timings of some benchmark programs.

In this context, results of the tuning process are most interesting, because they can be compared with the ones obtained with the model introduced in this work.

8.2.1 microSPARC™-II

This architecture introduces many difficulties in the tuning process, especially concerning branch execution. Above all, particular care has to be given to the branch folding technique, which allows a branch to be executed in parallel with the subsequent instruction, namely its *delay slot*: this reduces the average CPI of the branch-delay slot pair. Taking into account this timing reduction is not trivial: it has been decided to assign a nominal CPI of 0 to branch instructions, thus considering the best possible situation, where all branches are folded. A more realistic estimate can be obtained recognizing when a branch cannot be folded; this situation happens on dynamic conditions that can be assimilated to structural hazards: in fact, the branch folding technique cannot be applied only either if the processor is in some particular state driven by the instructions preceding the branch or if there is some cache miss. This kind of structural hazards are treated similarly to all other hazards, resulting in a more precise estimate of the branch dynamic timing.

As mentioned above, a RISC architecture has a simple instruction set: this fact leads to a narrow taxonomy, where only four class are present; in particular, class c_2, c_3, c_5 and c_7 contain no instructions. In Table 8.2.1 the tuning results are summarized.

The average overhead is very low, due the the high complexity of the microSPARC™-IIep datapath and to the limitation of the previous approach in handling the parallelism introduced by the pipelined execution and the branch folding technique.

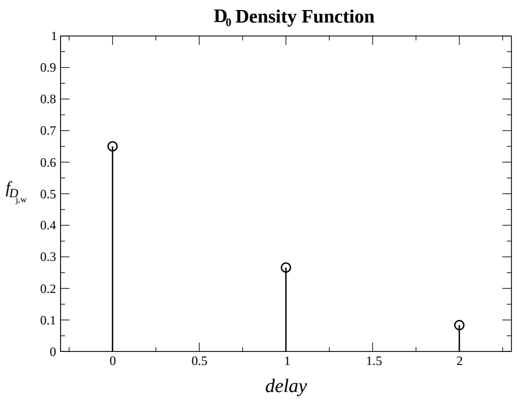
	Frequency	Overhead
c_0	0.077500	0.082264
c_1	0.112200	0.143028
c_2	0.000000	0.000000
c_3	0.000000	0.000000
c_4	0.601500	0.160880
c_5	0.000000	0.000000
c_6	0.208800	0.092958
c_7	0.000000	0.000000

Figure 8.2: Class Frequency and overhead on the microSPARC™-II

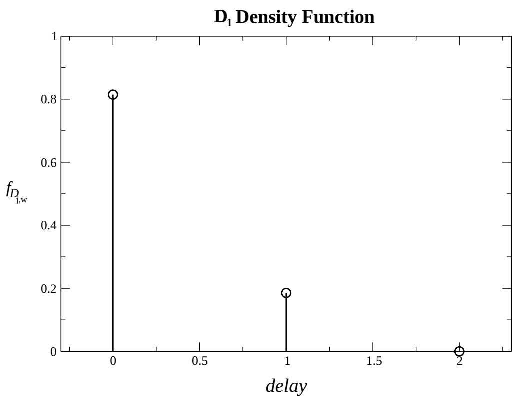
8.2.2 Intel486™

The Intel486™ has a simpler pipeline but features many more instruction types. This way, it is much more suitable for this kind of taxonomy: only class c_3 contains no instructions, and the delay is distributed on every other class.

The overhead introduced by each class is shown in figure 8.2.2. It can be noted that the average overhead introduced by each class is generally high: however, also the standard deviation is high, showing great variability between instructions belonging to the same class. These values are obtained as the average of the corresponding class-associated delay. The density functions show the predicted decreasing behavior: they span from no delay (the most probable) to delays of two clock cycles (actually the maximum for the considered set of benchmarks) as shown by figure 8.3.



(a)



(b)

Figure 8.3: Density functions of the delay distribution of class c_0 (a) and c_1 (b)

It can observed that classes c_3 and c_7 constitute two exceptions: the former is not applicable to the Intel assembly and the latter appears very seldom and with a very low overhead value, having a very low impact on the overall dynamic behavior.

	Frequency	Overhead
c_0	0.117686	0.433673
c_1	0.148612	0.185519
c_2	0.565506	0.484779
c_3	0.000000	0.000000
c_4	0.076104	0.515101
c_5	0.019178	0.108412
c_6	0.053744	0.499436
c_7	0.019174	0.042164

Figure 8.4: Class Frequency and overhead on the Intel486™

8.3 Tuning Results

As in the original work, the model proposed here has to be tuned in order to produce statistical figures to be used statically during the validation process. The tuning process flow has been introduced in Chapter 4 and in Chapter 5 while describing the software tools; a pictorial view has been given in figure 8.1.

8.3.1 microSPARC™-II

Given the microSPARC™-II data-path description, TrIBeS has been configured in order to simulate all the different functional units and resources presents in the microprocessor, so that the instruction flow can be correctly simulated, as described in chapter 7.

With respect to the the original model, its extension is able to overcome the problems related to the branch folding technique: the behavioral simulation of branch and delay slot can take into account their completely parallel execution when it is possible, while the situations in which folding is not possible are treated as a normal pipelined execution.

Fed with the same trace used for the non-superscalar model, the tuning process has given the results showed in table 8.1.

	Frequency	Overhead	Parallelism
c_0	0.077500	0.427135	0.201025
c_1	0.112200	0.270775	0.175171
c_2	0.000000	0.000000	0.000000
c_3	0.000000	0.000000	0.000000
c_4	0.601500	0.416031	0.201293
c_5	0.000000	0.000000	0.000000
c_6	0.208800	0.234593	0.214188
c_7	0.000000	0.000000	0.000000

Table 8.1: Class frequency, overhead and parallelism

It can be observed that the overhead values are very different with respect to the previous model; it is worth nothing that class c_0 and c_4 have a high overhead, principally due to the fact that they are composed of store and ALU instructions, which are often stalled after the loading of the data they use for register spilling activities and for arithmetic/logic computations. It is worth noting that the overhead so obtained take into account also the structural stalls related to the execution of multi-cycle instructions, which were explicitly neglected in the previous approach because their impact was accounted in the multi-cycle instruction CPIs. In the end, the previous approach results in a average overhead value of 0.14 clock cycles per instruction, while the new model reaches 0.36 clock cycles per instruction, which accounts for the relevant impact of inter-instruction effects in the global timing.

For what concerns parallelism parameters, it can be noticed that all values are around 0.2, which is the ideal parallelism factor for a pipelined architecture having CPI $= 1$ and five stages. However, the branch folding technique allows some instructions to be executed in a fully parallel way, thus generally reducing the parallelism coefficients. In particular, $p_{c_1} = 0.175$ is the lowest value, as class c_1 refers exclusively to branch instructions, which are the most likely to be folded. On the contrary, $p_{c_6} = 0.214$ has the highest value, being populated by load and control transfer instructions like `call` and `jmpl`, which are seldom folded due to architectural limitation. The average parallelism value is 0.198, which confirms that the microSPARC™-II has parallel execution capabilities that goes beyond a simple pipelined architecture. However, this does not mean that the architecture reaches a CPI lower than 1, since it is necessary to take into account the overhead value also. The average CPI value obtained from the tuning process for the microSPARC™-IIep is 1.08. Even though this value seems quite low, it has to be considered that memory effects are not taken into account, not only for the overhead introduced by cache misses, but also for the decrease of the parallelism, as memory-related effects deeply influences the branch folding technique.

8.3.2 Intel486™

Given the Intel486™ data-path description, TrIBeS has been configured in order to simulate all the different functional units and resources presents in the microprocessor, so that the instruction flow can be correctly simulated, as described in chapter 7.

Fed with a trace similar to the one used for the original model, the tuning process led to the results shown in table 8.2. Obviously, instruction frequencies are almost identical, the only

	Frequency	Overhead	Parallelism
c_0	0.0974435	2.013180	0.240998
c_1	0.1188520	2.431340	0.246369
c_2	0.6650900	2.347600	0.233041
c_3	0.0000000	0.000000	0.000000
c_4	0.0623378	2.268240	0.220810
c_5	0.0144432	2.141210	0.226939
c_6	0.0273912	4.436530	0.230612
c_7	0.0144423	6.868650	0.234246

Table 8.2: Class Frequency, overhead and parallelism coefficient for the Intel486™

differences due to the different traces used. The overhead per class is much greater instead. This is explained by the introduction of the parallelism coefficient. Since the overhead is computed on latencies (and not on CPIs), it has to be scaled down by the parallelism value in order to be comparable with previous results. This operation gives overhead values very similar (almost the same) to the ones obtained with the previous model, with two exceptions: classes c_6 and c_7. In these two cases the computed overhead is much closer to the actual value, since in the previous model floating point hazard detection was flawed and tuning was done on integer benchmarks only. Since most of floating point instructions belong to class c_6 and c_7, these differences are explained.

It is also worth noting that the parallelism coefficient is almost the same for every class and close to the ideal value of $1/5$. This means that the Intel486™, as clearly shown by its architecture in figure 7.6, does not include a high level of parallelism, implementing a simple DLX-like pipeline. In this case the parallelism coefficient coefficient does not show class biasing, since the parallelism exploited is only related to the partial instruction overlapping in the pipeline, which is common to all instructions.

8.4 Validation Results

The validation methodology consists in applying the measured timings to a number of assembly programs, obtaining their overall computation time, which is related to inter-instruction effects and parallel execution. The considered programs have been chosen so that all their data is cache-resident, in order to avoid errors or wrong validation due to cache memory stalls. These activities can be summarized as a series of steps:

1. The benchmark programs are traced
2. The dynamic timing of the program is computed: this means calculating the number of clock cycles needed to execute the program considering mean overhead parameters and parallelism factors.
3. The obtained timing is compared with the execution time of the program when run on the target architecture.

The toolset has then been used to determine the density functions f_{D_i} and f_{P_i} for each instruction using a subset of the benchmarks reported in figure 7.5. For each instruction the expectation value of the variables D_i and P_i have been calculated both with *hazard* and *full* classifications.

The model validation has been applied on a set of 11 benchmarks, executed on different data set:

bsort sorts a vector of 100 integers, using the well known bubblesort algorithm.

qsort sorts a vector of 100 integers, represented as strings, using the well known Quicksort algorithm.

rle computes the Redundancy Length Encoding on 128-characters randomly generated strings.

crc16 computes the 16–bit cyclic redundancy check on strings.

md5 computes the message digest of 500-characters randomly generated strings, using the MD5 algorithm.

adpcm encodes with adaptive differential pulse code modulation a set of samples

gsm encodes a set of audio samples in gsm format

g723 encodes in voice over IP standard g.723 a stream of data

lagrange computes the lagrange coefficient of a 100x100 matrix

fdct computes the discrete cosine transform on a set of samples, using floating point arithmetic

matrix computes a set of matrix operations among which determinant, summation and matrix product

8.4.1 microSPARC™-II

Actual timings have been obtained on a SPARCstation5 running Solaris5 as operating system. The processor is a microSPARC™-II with a clock speed of 85 MHz, a data cache of 8 kB and an instruction cache of 16 kB.

The traces used for tuning have been generated from the benchmarks `cpp2html`, `bc` (integer calculus), `gzip`, `mandel` (fractal generation), and `rasta` (image manipulation), for an overall trace length of approximately 1.5×10^8 instructions. This phase has led to two differently tuned models that have been been validated by estimating the execution time on all the benchmarks not used for the tuning phase. Figure 8.5 shows the results obtained by annotating the execution traces using the parallelism coefficients and the timing overheads resulting from the two considered classification schemes. In both cases the accuracy is more than satisfactory. The only exception

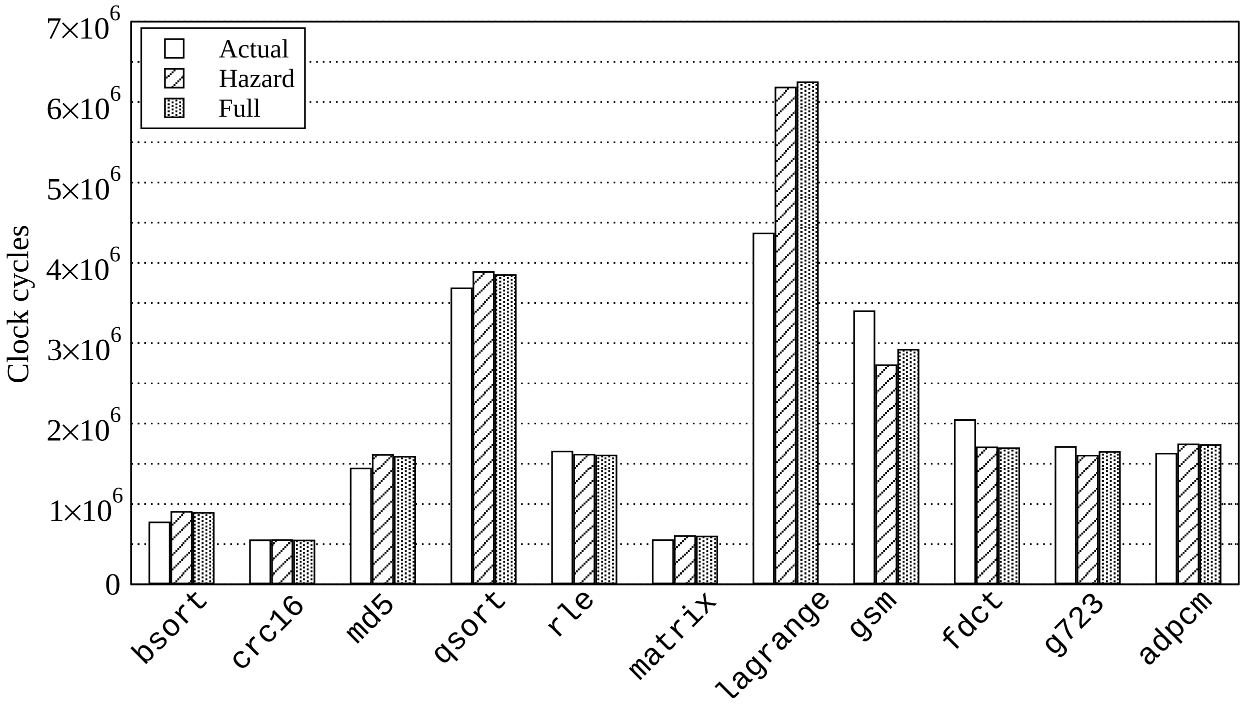


Figure 8.5: Accuracy of classification schemes for the microSPARC™-II

is the `lagrange` benchmark which shows a much higher error. This is due to the fact that it executes thousands of times a very tight loop where branch folding occurs with a probability that is much higher than the average case. Nevertheless, the average relative error is as low as 13% without classification and 11% with the hazard-based one[1]. It is worth noting that using classification leads to an improved accuracy for two main reasons:

- the resulting model is less sensitive to the instruction trace used for tuning;
- the peculiar timing behavior of certain (possibly frequent) instructions is averaged with that of more regular ones, smoothing the effect of borderline cases.

The validation process has been carried out similarly for both (previous and current model) cases. In particular, the two process are slightly different because the previous model takes into account the average CPI of instructions, while the present model is based on the typical instruction latency, as the actual CPI results from the product with the parallel coefficient. However, these differences in the validation process does not influence the estimation error resulting from the two approaches.

[1]Excluding the critical case of `lagrange`, the error drops to less than 8%.

The obtained results confirm that the new model can achieve a better accuracy than before, considering also floating point instructions and memory access. It must be noticed that the hazard classification is very rough for the new model: in fact, four classes are not able to describe the dynamic properties of instructions with respect to both inter-instruction effects and parallel execution. This fact suggests that a finer classification could lead to better accuracy.

8.4.2 Intel486™

The model validation has been applied on a set of 9 out of the 11 benchmarks used for the microSPARC™-II: `bsort`, `crc16`, `md5`, `qsort`, `rle`, `matrix`, `lagrange`, `fdct`, `mandel`, `adpcm`. These are the same benchmarks used for the microSPARC™ validation. The traces used for tuning have been generated from the benchmarks `ccrypt` (symmetric encryption), `cpp2latex` (text manipulation), `mpeg2encode` and `wget` (network utility), for an overall trace length of approximately 3.5×10^7 instructions. Figure 8.6 shows the results obtained by annotating the execution traces using the parallelism coefficients and the timing overheads resulting from the two considered classification schemes. Actual timings have been obtained on a Intel 486DX2-66 running

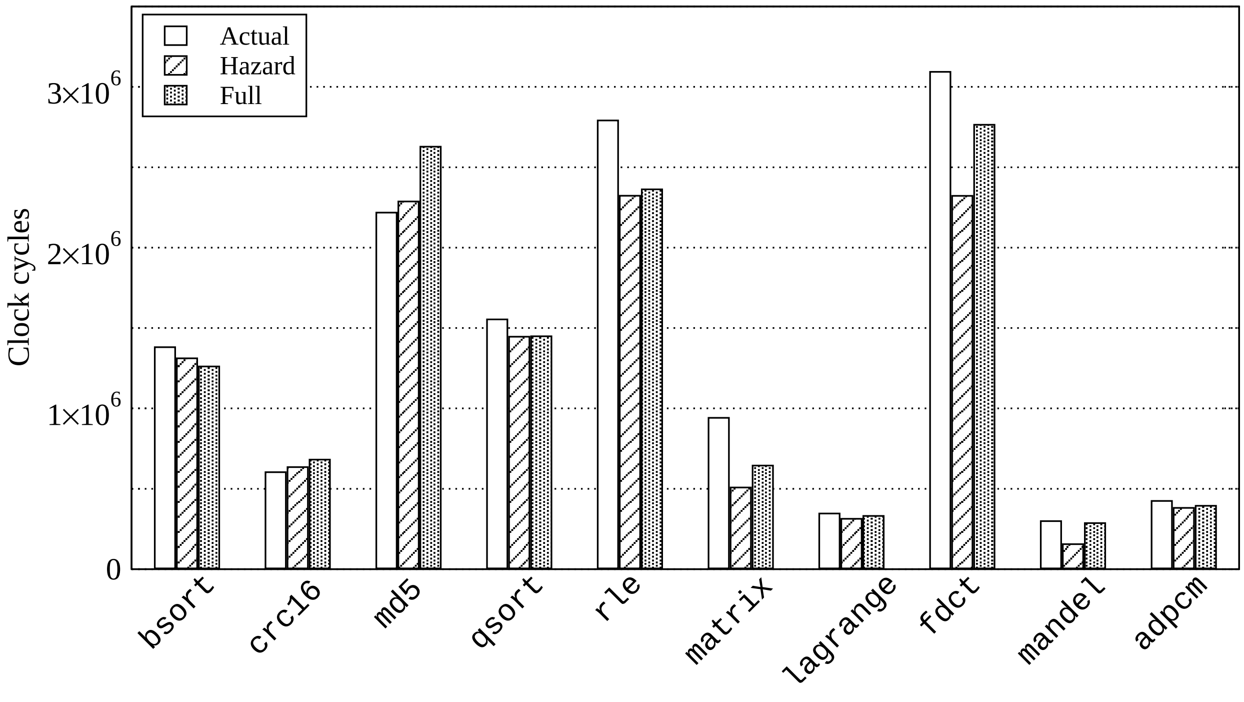


Figure 8.6: Accuracy of classification schemes for the Intel486™

Debian Linux 2.0 as operating system. The processor has a 66 MHz clock and a unified cache of 8 kB.

Since no memory model was used to patch TrIBeS results for memory effects, the error is slightly greater with respect to the Sparc model, reaching 17% for the hazard based classification and 11% for the full one. It is worth noting that for this processor there is an inversion in the trend, with the full classification leading to smaller errors. This is due to the fact that the Intel instruction set present a more uniform behavior, with very few borderline cases, mostly thanks to its simple pipeline. This uniform behavior of single instruction and a very heterogeneous instruction set leads to small errors for the full classification: each instruction is a case of its own, with its standard behavior. The hazard classification instead considers very different instructions belonging to the same class, giving slightly higher errors. Even in this case a fine classification, possibly based on numerical results, would probably give better results, taking the best of both approaches.

8.5 Future Work

Since validation is the key issue to confirm the efficacy of the methodology, a reliable validation method for obtained results should be found. While Solaris *high resolution timer* and Linux *system timer* can be considered sufficiently accurate for a preliminary analysis, significantly better results could be obtained with a cycle-accurate validation. This would be possible if RTL simulators or cycle counters were available; in fact, a Verilog RTL model of the microSPARC™-IIep is available, but it is not easy to simulate without complex and expensive tools, and Sun provides no support. Concerning the Intel486™, a number of simulators is available, but they are extremely expensive commercial products. Future work will include experiments on this and on other validation sources. It is also worth noting that a better classification, as already stated, would bring to significantly better results. Hence, the development of classification helping tool is on the way.

Another interesting field at which the model could be applied is to add memory support: currently memory is supported by the simulator, but it is not considering stalls of any kind. Adding a complete memory module to TrIBeS could increase the accuracy of the model both in time and energy estimation.

Finally, to obtain a validation of the model on a larger scope, it can be applied to a set of target architectures: as an example the ARM and PowerPC processors. In this way, methodology efficacy and costs could be determined.

8.6 Conclusions

This work has presented a new approach in time and power estimation of software execution on a given architecture. In particular, the proposed methodology extends previous approaches with the introduction of parallel execution of instructions, resulting in a rigorous mathematical model, which exhibits good statistical properties. The preliminary results shown here are very promising: a finer model tuning and a better validation methodology could lead to improved accuracy. Besides, the instruction set taxonomy proved to be a crucial point. Future works could lead to the definition of the taxonomy by means of an *a-posteriori* analysis of data resulting from the behavioral simulation. Finally, it has to be considered the introduction in the proposed methodology of a model for memory hierarchies, whose integration in the behavioral simulator is straightforward.

A Used Notation

V_{dd}	Supply voltage for a given processor
BC_{stall}	Base cost of an interlock
SP	Stall Penalty
MP	Cache Miss Penalty
N	Code size in number of instructions
s	instruction of a microprocessor
i_s	Average current needed to execute instruction s
e_s	Average energy associated with instruction s
$n_{ck,s}$	Number of clock cycles needed to execute instruction s
τ	Clock period
F_i	ith processor functionality
$a_{s,j}$	activation coefficient of instruction s and the jth functionality
$a_{s,F\&D}$	activation coefficient of instruction s and the $F\&D$ functionality
if_j	average current absorbed by jth functionality in one clock cycle
I	Set of all the instructions of a given processor (Instruction Set)
I_L	Learning set of a given processor
I_G	Generalization set of a given processor
$\mathbf{IN}$	$m_L \times 1$ column vector whose elements are the terms $i_s \cdot n_{ck,s}$
$\mathbf{IF}$	$k \times 1$ column vector whose elements are if_j
$\mathbf{A}$	$m_L \times k$ matrix whose entries are the activation coefficients $a_{s,j}$
$\mathbf{R}$	residual vector
$\hat{\lambda}^2$	Estimator of error variance
$i_{rel,s}$	Relative average current to execute instruction s
$b_{s,j}$	Indicates the involvement of instruction s with the jth functionality
w_s	Execution weight of instruction s
$\mathcal{I}_{H,i}$	H-type partition of the instruction set I
S-type	Structural hazard type
D-type	Data hazard type
C-type	Control hazard type
$\mathcal{C}$	Taxonomy of the instruction set I
$C_{i,j,k}$	Class of the instruction set obtained intersecting $I_{S,i}$, $I_{D,i}$ and $I_{C,i}$
c_i	Short notation for $C_{i,j,k}$ where i is the decimal value of the binary ijk
Γ	Execution trace of a program
γ_k	Instruction in position k of an execution trace Γ
$w(\gamma_{k_1}, \gamma_{k_2})$	Distance between the two instructions γ_{k_1} and γ_{k_2}
$\gamma_{k_1} \overset{\hat{w}}{\dashv} \gamma_{k_2}$	Distance operator that indicates that two instructions have $w = \hat{w}$
$\langle k, i \rangle$	Membership function: indicates that instruction γ_k belongs to class c_i
$c_i \overset{\hat{w}}{\dashv} c_j$	Distance between two classes: indicates that exist two classes c_i and c_j at distance $\hat{w}$
$P(c_i)$	Probability of finding an instruction of class c_i into the execution trace Γ
$P(c_i \overset{\hat{w}}{\dashv} c_j)$	Probability of finding a pair of instructions, of class c_i and c_j respectively, at distance $\hat{w}$ into the execution trace Γ

$t(\gamma_k, \gamma_{k+\hat{w}}, \hat{w})$	Execution overhead in clock cycles due to inter-instruction effects between two instructions at distance $\hat{w}$
$D_{i,j,\hat{w}}$	Statistical value of the delay between two classes c_i and c_j at distance $\hat{w}$
$f_{D_{i,j,\hat{w}}}(d)$	Density function of stochastic variable $D_{i,j,\hat{w}}$
$\delta_{delay\hat{w}}$	Kronecker delta
$\mathbf{F}$	Matrix of the density functions $f_{i,j,d}$
$D_{i,\hat{w}}$	Statistical value of the delay between class c_i any other class at distance $\hat{w}$
$f_{D_{i,\hat{w}}}(d)$	Density function of stochastic variable $D_{i,\hat{w}}$
$\mu_{D_{i,j,\hat{w}}}$	Expected value of $D_{i,j,\hat{w}}$
$\sigma_{D_{i,j,\hat{w}}}$	Variance of $D_{i,j,\hat{w}}$
$\mu_{D_{i,\hat{w}}}$	Expected value of $D_{i,\hat{w}}$
$\sigma_{D_{i,\hat{w}}}$	Variance of $D_{i,\hat{w}}$
$n_{ck,s,Stall}$	Number of clock cycles wasted due to stalls by instruction s
$a'_{s,j}$	Stall coefficients: determine the functionalities used by pipeline interlocks
w'_s	Stall weight of instruction s
$\mathbf{A}'$	Matrix of the stall coefficients $a'_{s,j}$

B Micro-Instruction Templates

B.1 microSPARC™-II

B.1.1 Integer ALU Instructions

```
INSTRUCTION_START <class> <latency>         # Multicycle Instruction - 3 operands
  FE:Fetch  use 4 FLAG:FoldingFlag          # Fetch:     Set Folding Flag
  : read <sreg1> REG:IntRegFile             # Decode:    check <sreg1> lock state
  : read <sreg2> REG:IntRegFile             # Decode:    check <sreg2> lock state
  : write <dreg> REG:IntRegFile             # Decode:    lock <dreg>
  : require <latency> AI:AluInt             # Decode:    send to AluInt
                                            # AluInt:    set Instruction latency
  AI:AluInt use 4 FLAG:FoldingFlag          # AluInt:    Reset Folding Flag
  AI:AluInt write <dreg> REG:IntRegFile     # AluInt:    unlock <reg> (forwarding)
  WB:WriteBack write <dreg> REG:IntRegFile  # WriteBack: write <reg> (use output port)
INSTRUCTION_END

INSTRUCTION_START <class> <latency>         # Integer ALU Instruction - 3 operands
  : read <sreg1> REG:IntRegFile             # Decode:    check <sreg1> lock state
  : read <sreg2> REG:IntRegFile             # Decode:    check <sreg2> lock state
  : write <dreg> REG:IntRegFile             # Decode:    lock <dreg>
  AI:AluInt write <dreg> REG:IntRegFile     # AluInt:    unlock <reg> (forwarding)
  WB:WriteBack write <dreg> REG:IntRegFile  # WriteBack: write <reg> (use output port)
INSTRUCTION_END

INSTRUCTION_START <class> <latency>         # Integer ALU Instruction - 2 operands
  : read <sreg1> REG:IntRegFile             # Decode:    check <sreg1> lock state
  : write <dreg> REG:IntRegFile             # Decode:    lock <dreg>
  : require <latency> AI:AluInt             # Decode:    send to AluInt
                                            # AluInt:    set Instruction latency
  AI:AluInt write <reg> REG:IntRegFile      # AluInt:    unlock <reg> (forwarding)
  WB:WriteBack write <reg> REG:IntRegFile   # WriteBack: write <reg> (use output port)
INSTRUCTION_END
```

```
INSTRUCTION_START <class> <latency>        # Integer ALU Instruction - 1 operands
  : read <reg> REG:IntRegFile              # Decode:    check <reg> lock state
  : write <reg> REG:IntRegFile             # Decode:    lock <reg>
  : require <latency> AI:AluInt            # Decode:    send to AluInt
                                           # AluInt:    set Instruction latency
  AI:AluInt write <reg> REG:IntRegFile     # AluInt:    unlock <reg> (forwarding)
  WB:WriteBack write <reg> REG:IntRegFile  # WriteBack: write <reg> (use output port)
INSTRUCTION_END

INSTRUCTION_START <class> <latency>        # Integer Multiply/Divide
  FE:Fetch  use 4 FLAG:FoldingFlag         # Fetch:     Set Folding Flag
  : read <reg> REG:IntRegFile              # Decode:    check <reg> lock state
  : read <reg> REG:IntRegFile              # Decode:    check <reg> lock state
  : write <reg> REG:IntRegFile             # Decode:    lock <reg>
  : read Y REG:IntRegFile                  # Decode:    check Y lock state
  : require <latency> AI:AluInt            # Decode:    send to AluInt
                                           # AluInt:    set Instruction latency
  AI:AluInt use 4 FLAG:FoldingFlag         # AluInt:    Reset Folding Flag
  AI:AluInt read Y REG:IntRegFile          # AluInt:    check Y lock state
  AI:AluInt write <reg> REG:IntRegFile     # AluInt:    unlock <reg> (forwarding)
  WB:WriteBack write <reg> REG:IntRegFile  # WriteBack: write <reg> (use output port)
INSTRUCTION_END
```

B.1.2 Control Transfer Instructions

```
INSTRUCTION_START <class> <latency>       # BRANCH Instruction
  FE:Fetch branch <annull||taken>         # Fetch:     exec branch folding logic
  AI:AluInt use 4 FLAG:FoldingFlag        # AluInt:    decr FoldingFlag
INSTRUCTION_END

INSTRUCTION_START <class> <latency>       # CALL Instruction
  FE:Fetch  use 4 FLAG:FoldingFlag        # Fetch:     Set Folding Flag
  : write %r15 REG:IntRegFile             # Decode:    lock %r15
  AI:AluInt use 4 FLAG:FoldingFlag        # AluInt:    Reset Folding Flag
  MM:Memory write %r15 REG:IntRegFile     # Memory:    unlock %r15
  WB:WriteBack write %r15 REG:IntRegFile  # WriteBack: write <dreg> (JMPL only)
INSTRUCTION_END

INSTRUCTION_START <class> <latency>       # JMPL/RETT Instructions
  FE:Fetch  use 4 FLAG:FoldingFlag        # Fetch:     Set Folding Flag
  : read <sreg1> REG:IntRegFile           # Decode:    check <sreg1> lock state
  : read <sreg2> REG:IntRegFile           # Decode:    check <sreg2> lock state
  : write <dreg> REG:IntRegFile           # Decode:    lock <dreg> (JMPL only)
  : require <latency> AI:AluInt           # Decode:    send to AluInt
                                          # AluInt:    set Instruction latency
  AI:AluInt use 4 FLAG:FoldingFlag        # AluInt:    Reset Folding Flag
  AI:AluInt write <dreg> REG:IntRegFile   # AluInt:    unlock <dreg> (JMPL only)
  WB:WriteBack write <dreg> REG:IntRegFile # WriteBack: write <dreg> (JMPL only)
INSTRUCTION_END
```

```
INSTRUCTION_START <class> <latency>        # Trap Instruction
  FE:Fetch  use 4 FLAG:FoldingFlag         # Fetch:    Set Folding Flag
  : require <latency> AI:AluInt            # Decode:   send to AluInt
                                           # AluInt:   set Instruction latency
  AI:AluInt use 4 FLAG:FoldingFlag         # AluInt:   Reset Folding Flag
INSTRUCTION_END
```

B.1.3 Integer LOAD/STORE Instructions

```
INSTRUCTION_START <class> <latency>        # LOAD Single Instruction
  : read <sreg1> REG:IntRegFile            # Decode:    check <sreg1> lock state
  : read <sreg2> REG:IntRegFile            # Decode:    check <sreg2> lock state
  : write <dreg> REG:IntRegFile            # Decode:    lock <dreg>
  MM:Memory load <address>                 # Memory:    load data from <address>
  MM:Memory write <dreg> REG:IntRegFile    # Memory:    unlock <dreg>
  WB:WriteBack write <dreg> REG:IntRegFile # WriteBack: write <dreg> (JMPL only)
INSTRUCTION_END

INSTRUCTION_START <class> <latency>        # LOAD Double/LDA Instruction
  FE:Fetch  use 4 FLAG:FoldingFlag         # Fetch:     Set Folding Flag
  : read <sreg1> REG:IntRegFile            # Decode:    check <sreg1> lock state
  : read <sreg2> REG:IntRegFile            # Decode:    check <sreg2> lock state
  : write <dreg> REG:IntRegFile            # Decode:    lock <dreg>
  : require <latency> AI:AluInt            # Decode:    send to AluInt
                                           # AluInt:    set Instruction latency
  AI:AluInt use 4 FLAG:FoldingFlag         # AluInt:    Reset Folding Flag
  MM:Memory load <address>                 # Memory:    load data from <address>
  MM:Memory write <dreg> REG:IntRegFile    # Memory:    unlock <dreg>
  WB:WriteBack write <dreg> REG:IntRegFile # WriteBack: write <dreg> (JMPL only)
INSTRUCTION_END

INSTRUCTION_START <class> <latency>        # STORE Single Instruction
  : read <sreg1> REG:IntRegFile            # Decode:    check <sreg1> lock state
  : read <sreg2> REG:IntRegFile            # Decode:    check <sreg2> lock state
  : read <sreg3> REG:IntRegFile            # Decode:    check <sreg3> lock state
  MM:Memory store <address>                # Memory:    store data at <address>
INSTRUCTION_END

INSTRUCTION_START <class> <latency>        # STORE Double/SDA Instruction
  FE:Fetch  use 4 FLAG:FoldingFlag         # Fetch:     Set Folding Flag
  : read <sreg1> REG:IntRegFile            # Decode:    check <sreg1> lock state
  : read <sreg2> REG:IntRegFile            # Decode:    check <sreg2> lock state
  : read <sreg3> REG:IntRegFile            # Decode:    check <sreg3> lock state
  : require <latency> AI:AluInt            # Decode:    send to AluInt
                                           # AluInt:    set Instruction latency
  AI:AluInt use 4 FLAG:FoldingFlag         # AluInt:    Reset Folding Flag
  MM:Memory store <address>                # Memory:    load data at <address>
INSTRUCTION_END
```

```
INSTRUCTION_START <class> <latency>        # LDSTUB/SWAP Instructions
  FE:Fetch  use 4 FLAG:FoldingFlag         # Fetch:     Set Folding Flag
  : read <sreg1> REG:IntRegFile            # Decode:    check <sreg1> lock state
  : read <sreg2> REG:IntRegFile            # Decode:    check <sreg2> lock state
  : write <dreg> REG:IntRegFile            # Decode:    lock <dreg>
  : require <latency> AI:AluInt            # Decode:    send to AluInt
                                           # AluInt:    set Instruction latency
  AI:AluInt use 4 FLAG:FoldingFlag         # AluInt:    Reset Folding Flag
  MM:Memory load <address>                 # Memory:    load data from <address>
  MM:Memory load <address>                 # Memory:    load data from <address>
  MM:Memory write <dreg> REG:IntRegFile    # Memory:    unlock <dreg>
  WB:WriteBack write <dreg> REG:IntRegFile # WriteBack: write <dreg> (JMPL only)
INSTRUCTION_END
```

B.1.4 Read /Write Special Register

```
INSTRUCTION_START <class> <latency>        # Read Special Register
  : read <sreg1> REG:IntRegFile            # Decode:    check <sreg1> lock state
  : write <dreg> REG:IntRegFile            # Decode:    lock <dreg>
  MM:Memory write <dreg> REG:IntRegFile    # Memory:    unlock <dreg>
  WB:WriteBack write <dreg> REG:IntRegFile # WriteBack: write <dreg>
INSTRUCTION_END

INSTRUCTION_START <class> <latency>        # Write Special Register
  FE:Fetch  use 4 FLAG:FoldingFlag         # Fetch:      Set Folding Flag
  : read <sreg1> REG:IntRegFile            # Decode:     check <sreg1> lock state
  : read <sreg2> REG:IntRegFile            # Decode:     check <sreg2> lock state
  : write <dreg> REG:IntRegFile            # Decode:     lock <dreg>
  AI:AluInt write <dreg> REG:IntRegFile    # AluInt:     unlock <dreg>
  WB:WriteBack write <dreg> REG:IntRegFile # WriteBack:  write <dreg>
  WB:WriteBack use 4 FLAG:FoldingFlag      # Fetch:      Set Folding Flag
INSTRUCTION_END
```

B.1.5 Floating Point Instructions

```
INSTRUCTION_START <class> <latency>        # Floating Multiply/Divide
 : read <sreg1> REG:FpRegFile              # Decode:   check <sreg1> lock state
 : read <sreg2> REG:FpRegFile              # Decode:   check <sreg2> lock state
 : write <dreg> REG:FpRegFile              # Decode:   lock <dreg>
 : use 2 <re-order-buffer>                 # Decode:   insert instruction in ROB
 : require 3 FP:FpMulIn                    # Decode:   send to FpMulIn
                                           # FpMulIn:  set Instruction latency
 : require 2 FP:FpMulOut                   # FpMulOut: set Instruction latency
 FP:Retire write <dreg> REG:IntRegFile     # Retire:   write and unlock <dreg>
 : use 2 <re-order-buffer>                 # Decode:   insert instruction in ROB
INSTRUCTION_END
```

```
INSTRUCTION_START <class> <latency>        # FloatingPoint Operations
  : read <sreg1> REG:FpRegFile             # Decode:   check <sreg1> lock state
  : read <sreg2> REG:FpRegFile             # Decode:   check <sreg2> lock state
  : write <dreg> REG:FpRegFile             # Decode:   lock <dreg>
  : use 2 <re-order-buffer>                # Decode:   insert instruction in ROB
  : require <latency> FP:AluFp             # Decode:   send to AluFp
                                           # AluFp:    set Instruction latency
  FP:Retire write <dreg> REG:IntRegFile    # Retire:   write and unlock <dreg>
  : use 2 <re-order-buffer>                # Retire:   remove instruction form ROB
INSTRUCTION_END

INSTRUCTION_START <class> <latency>        # COMPARE FloatingPoint
  : read <sreg1> REG:FpRegFile             # Decode:   check <sreg1> lock state
  : read <sreg2> REG:FpRegFile             # Decode:   check <sreg2> lock state
  : write <dreg> REG:FpRegFile             # Decode:   lock <dreg>
  : use 2 <re-order-buffer>                # Decode:   insert instruction in ROB
  : use 4 <FLAG:floatingFlag>              # Decode:   increments floating flag
  : require <latency> FP:AluFp             # Decode:   send to AluFp
                                           # AluFp:    set Instruction latency
  FP:Retire write <dreg> REG:IntRegFile    # Retire:   write and unlock <dreg>
  : use 4 <FLAG:floatingFlag>              # Retire:   decrements floating flag
  : use 2 <re-order-buffer>                # Retire:   remove instruction form ROB
INSTRUCTION_END

INSTRUCTION_START <class> <latency>        # BRANCH FloatingPoint
  FE:Fetch  use 4 FLAG:FoldingFlag         # Fetch:     Set Folding Flag
  : read <sreg1> REG:IntRegFile            # Decode:    check <sreg1> lock state
  : read <sreg2> REG:IntRegFile            # Decode:    check <sreg2> lock state
  : write <dreg> REG:IntRegFile            # Decode:    lock <dreg> (JMPL only)
  : require <latency> AI:AluInt            # Decode:    send to AluInt
                                           # AluInt:    set Instruction latency
  AI:AluInt use 4 FLAG:FoldingFlag         # AluInt:    Reset Folding Flag
  AI:AluInt write <dreg> REG:IntRegFile    # AluInt:    unlock <dreg> (JMPL only)
  WB:WriteBack write <dreg> REG:IntRegFile # WriteBack: write <dreg> (JMPL only)
INSTRUCTION_END
```

B.2 Intel486™

B.2.1 Integer ALU Instructions

```
INSTRUCTION_START <class> <latency>        # Multicycle Instruction - 3 operands
  DE:Decode2 read <sreg1> REG:IntRegFile   # Decode:    check <sreg1> lock state
  DE:Decode2 read <sreg2> REG:IntRegFile   # Decode:    check <sreg2> lock state
  DE:Decode2 write <dreg> REG:IntRegFile   # Decode:    lock <dreg>
  Ex:Execute require <latency> Ex:Execute  # AluInt:    set Instruction latency
  WB:WriteBack write <dreg> REG:IntRegFile # WriteBack: unlock <reg>
INSTRUCTION_END
```

```
INSTRUCTION_START <class> <latency>        # Integer ALU Instruction - memory read
  DE:Decode2 read <sreg1> REG:IntRegFile   # Decode:    check <sreg1> lock state
  DE:Decode2 load <mem>   REG:IntRegFile   # Decode:    load <mem> operand
  DE:Decode2 write <dreg> REG:IntRegFile   # Decode:    lock <dreg>
  Ex:Execute require <latency> Ex:Execute  # AluInt:    set Instruction latency
  WB:WriteBack write <dreg> REG:IntRegFile # WriteBack: unlock <reg>
INSTRUCTION_END

INSTRUCTION_START <class> <latency>        # Integer ALU Instruction - memory write
  DE:Decode2 read <sreg1> REG:IntRegFile   # Decode:    check <sreg1> lock state
  DE:Decode2 read <sreg2> REG:IntRegFile   # Decode:    check <sreg2> lock state
  Ex:Execute require <latency> Ex:Execute  # AluInt:    set Instruction latency
  WB:WriteBack write <dreg> REG:IntRegFile # WriteBack: unlock <reg>
  WB:WriteBack store <mem>  REG:IntRegFile # WriteBack: store  <mem>
INSTRUCTION_END
```

B.2.2 Control Transfer Instructions

```
INSTRUCTION_START <class> <latency>        # BRANCH Instruction
  FE:Fetch branch <taken>                  # Fetch:     exec branch folding logic
INSTRUCTION_END
```

B.2.3 Floating Point Instructions

```
INSTRUCTION_START <class> <latency>        # Floating Operations
  DE:Decode2 read <sreg1> REG:IntRegFile   # Decode:   check <sreg1> lock state
  DE:Decode2 read <sreg2> REG:IntRegFile   # Decode:   check <sreg2> lock state
  DE:Decode2 write <dreg> REG:IntRegFile   # Decode:    lock <dreg>
  : require <latency> FP:AluFp             # Decode:   send to AluFp
                                           # AluFp:    set Instruction latency
  WB:WriteBack write <dreg> REG:IntRegFile # WriteBack: unlock <reg>
  WB:WriteBack write <dreg> REG:IntRegFile # WriteBack: unlock <reg>
INSTRUCTION_END

INSTRUCTION_START <class> <latency>        # FloatingPoint Operations
  DE:Decode2 read <sreg1> REG:IntRegFile   # Decode:    check <sreg1> lock state
  DE:Decode2 read <sreg2> REG:IntRegFile   # Decode:    check <sreg2> lock state
  : require <latency> FP:AluFp             # Decode:   send to AluFp
                                           # AluFp:    set Instruction latency
  WB:WriteBack write <dreg> REG:IntRegFile # WriteBack: unlock <reg>
  WB:WriteBack store <mem>  REG:IntRegFile # WriteBack: store  <mem>
INSTRUCTION_END

INSTRUCTION_START <class> <latency>        # COMPARE FloatingPoint
  DE:Decode2 read <sreg1> REG:IntRegFile   # Decode:   check <sreg1> lock state
  DE:Decode2 read <sreg2> REG:IntRegFile   # Decode:   check <sreg2> lock state
  : require <latency> FP:AluFp             # Decode:   send to AluFp
                                           # AluFp:    set Instruction latency
INSTRUCTION_END
```

List of Tables

List of Figures

Bibliography

[1] G. Arnout. Systemc standard. In *Proceedings of the Asia and South Pacific Design Automation Conference, ASP–DAC*, pages 573–577, 2000.

[2] G. Beltrame, C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto, and V. Trianno. An assembly-level execution-time model for pipelined architectures. In *Proceedings of International Conference on Computer Aided Design, ICCAD2001*, pages 195–200, San Jose, CA, November 2001.

[3] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. Power modeling of 32-bit microprocessors. Technical report, Politecnico di Milano, Piazza Leonardo da Vinci 32, 2000.

[4] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report SMLI 93-12, UWCSE 93-06-06, 1993.

[5] Intel Corp. *Intel Architecture Software Developer's Manual vol 1*. Intel Corp., 1997.

[6] Intel Corp. *Intel Architecture Software Developer's Manual vol 2*. Intel Corp., 1997.

[7] Intel Corp. *VTune Online Help*. Intel Corp., 1999.

[8] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *Prooceding of the 28th Annual International Symposium on Computer Architecture*, pages 49–58, 2001.

[9] D. Drusinsky and D. Harel. Using statecharts for hardware description and synthesis. In *IEEE Transactions for Hardware Description Synthesis*, volume 8, pages 798–807, 1989.

[10] J. Emer. Asim: A performance model framework. *Computer*, 35(2):68–76, February 2002.

[11] A.A. Ghosh, S. Devadas, K. Keutzer, and J. White. Estimation of Average Switching Activity in Combinational and Sequential Circuits. In *Proceedings of the 29th Design Automation Conference*, pages 253–259, 1992.

[12] J.L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, II edition, 1996.

[13] C. J. Hughes, S. P. Vijay, R. Parthasarathy, and A. V. Sarita. Performing valid studies with unvalidated simulators. *VLSI Design Journal*, 35(2), 2002.

[14] D Ku and G. De Micheli. Hardwarec - a language fo hardware design (version 2.0). Technical report, Stanford University, April 1990.

[15] P. Maciel and E. Barros. Capturing time constraints by using petri nets in the context of hardware/software codesign. In *Proceedings of the 7th IEEE International Workshop on Rapid System Prototyping*, page 36/41, 1996.

[16] Sun Microsystems. microsparc™-iiep source distribution. `http://www.sun.com`.

[17] Sun microsystems. *The SPARC Architecture Manual, version 8*. Sun microsystems, 1990.

[18] Sun microsystems. *microSPARC™-IIep User's Manual*. Sun microsystems, 1997.

[19] A. Mood, F. Graybill, and D. Boes. *Introduction to the theory of statistics*. McGraw–Hill, New York, NY, 1988.

[20] S. Shubhendu Mukherjee, V. Sarita Adve, Todd Austin, Joel Emer, and S. Peter Magnusson. Performance simulation tools. *VLSI Design Journal*, 35(2), 2002.

[21] F.N. Najim. Transition Density: A New Measure of Activity in Digital Circuits. *IEEE Transactions on Computer Aided Design*, February 1993.

[22] F.N. Najim, R. Burch, P. Yang, and I.N. Hajj. Probabilistic Simulation for Reliability analysis of CMOS circuits. *IEEE Transactions on Computer Aided Design*, April 1990.

[23] J. Philipps and P. Scholz. Synthesis of digital circuits from hierarchical state machines. In *Proceeding of the Fifth GI/ITG/GMM Workshop*, 1997.

[24] Sridhar Ramalingam and Kris Schindler. Instruction level power model and its application to general purpose processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 753–756, 1998.

[25] Dezso Sima, Terence Fountain, and Peter Klaksuk. *Advanced Computer Architectures – A Design Space Approach*. Addison-Wesley, 1998.

[26] StatSoft Inc. Electronic testbook. `http://www.statsoftinc.com/textbook/stathome.html`.

[27] Sun Microsystems. Prying into processes and workloads. FAQ published on Unix Insider 4/1/98.

[28] V. Tiwari and M.T.C. Lee. Power analysis of a 32-bit embedded microcontroller. *VLSI Design Journal*, 7(3), 1998.

[29] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *IEEE Transactions on VLSI Systems*, 2(4):437–445, December 1994.

[30] V. Tiwari, S. Malik, and A. Wolfe. Power Analysis of the Intel 486DX2. Computer Engineering Technical Report No. CE-M94-5, Princeton University, June 1994.

[31] F. Vahid, S. Narayan, and D. Gajski. System specification with the speccharts language. *IEEE Design & Test of Computers*, December 1995.

[32] F. Vahid, S. Narayan, and D. Gajski. A vhdl front end for embedded systems. In *IEEE Transactions for Hardware Description Synthesis*, volume 14, pages 798–807, 1995.