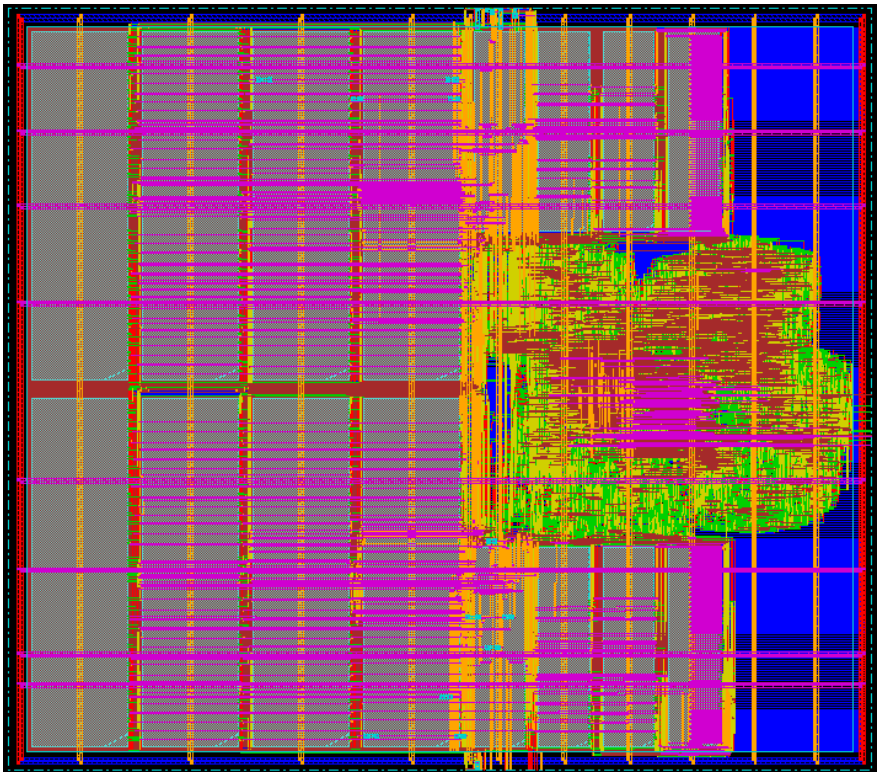

Development of an implementation-centric energy-evaluation framework for MIPS-I pipelines

Master's thesis in Embedded Electronic System Design

Daniel Moreau

Department of Computer Science & Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

Supervisor: Per Larsson-Edefors, Department of Computer Science & Engineering
Examiner: Sven Knutsson, Department of Computer Science & Engineering

Department of Computer Science & Engineering
Division of Computer Engineering
VLSI Research Group
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Placed and routed five stage pipeline built at Chalmers University of Technology.

Typeset in LaTeX
Gothenburg, Sweden 2016

Daniel Moreau
Master's thesis in Embedded Electronic System Design
Chalmers University of Technology

Abstract

An RTL-based energy evaluation framework dubbed CREEP (Chalmers Energy Evaluation framework for Pipelines) is implemented and evaluated. The framework consists of pipeline RTL and the architectural simulator SimpleScalar. Power estimates are extracted from the RTL and combined with performance counters generated by SimpleScalar. The combination lends SimpleScalar accurate energy estimates otherwise reserved to low level circuit analysis. The framework has been used to characterize several different embedded processor configurations. Additionally, a case study of the framework was used to implement and evaluate a speculative way-halting technique called SHA which pointed to a 25.6% energy reduction in a conventional four-way data cache.

Acknowledgements

I want to thank my supervisor Per Larsson-Edefors for his guidance and support throughout my work. I also want to thank Alen Bardizbanyan, without his help and technical expertise this work would would not have been possible. Lastly, I want to thank my family and girlfriend for their continued support and encouragement during hard times.

Daniel Moreau, Gothenburg, January 2016

Contents

List of Figures

List of Tables

Acronyms

5SP 5-stage pipeline. 2, 3, 4, 21, 24, 26, 33, 38, 67

AGU address generation unit. 41, 61
ALU arithmetic logic unit. 4, 26, 36, 37, 41, 58

CAM content addressable memory. 4
CMOS complementary metal–oxide–semiconductor. 1, 7, 8, 9, 10, 12
CPI cycles per instruction. 16, 18, 19
CREEP Chalmers RTL-based energy evaluation framework for pipelines. 2, 3, 4, 5, 6, 7, 33, 42, 43, 45, 54, 55, 57, 67

DC Synopsys Design Compiler. 35, 42
DTLB data translation lookaside buffer. 54, 55

EDA electronic design automation. 10, 11, 12, 34, 35, 36
EX execute. 16, 18, 19, 21, 26, 41

GP general purpose. 11

HDL hardware description language. 10, 11, 34, 45

IC integrated circuit. 1, 7, 10, 11, 12, 33, 35
ID instruction decode. 16, 18, 19, 20, 21, 26, 41
IF instruction fetch. 16, 18, 19, 26
ILP instruction-level parallelism. 15, 21
IPC instruction per cycle. 16, 48
ISA instruction set architecture. 13, 15, 16, 22, 33, 34, 67
ITRS International Technology Roadmap for Semiconductors. 4

L1 level-one. 22, 23, 47, 48, 50, 67
L1DC level-one data cache. 2, 22, 26, 33, 35, 36, 39, 47, 50, 51, 52, 53, 54, 55, 56, 62, 67
L1IC level-one instruction cache. 2, 22, 26, 33, 39, 42, 47, 50, 51, 52, 53
L2 level-two. 3, 22, 23, 24, 26, 39, 47, 48, 50, 51, 57
LP low power. 11, 35
LRU least recently used. 23, 26, 54
LSU load store unit. 41

MEM memory access. 16, 18, 26, 41

nMOS n-type metal–oxide–semiconductor. 7, 8, 10
NOP no-operation. 18, 26
NP nondeterministic polynomial time. 12

OOM orders of magnitude. 22
OP operation. 13

PC program counter. 15, 19, 26
pMOS p-type metal–oxide–semiconductor. 7, 8, 10
PnR place and route. 11, 12, 27, 29, 35, 37, 41, 58, 67
PT Synopsys PrimeTime. 36, 37, 40, 42

RAM random access memory. 4
RAW read after write. 18, 21
RISC reduced instruction set computing. 13
RT register transfer. 11, 67
RTL register-transfer level. 1, 2, 3, 4, 5, 10, 11, 12, 24, 25, 26, 27, 28, 29, 30, 31, 33, 34, 35, 36, 38, 39, 41, 42, 43, 45, 46, 54, 57, 58, 61, 67

SAIF switching activity interchange format. 35, 36, 42, 45
SPEF standard parasitic exchange format. 57
SRAM static random access memory. 27, 33, 34, 35, 37, 46, 50, 51, 52, 57, 61

VCD value change dump. 35, 57
VLSI very-large-scale integration. 1, 4

WB write-back. 16, 18, 26, 41

1

Introduction

In the early days of integrated circuit (IC) design, computer architectures were developed with a focus on achieving high performance. Other design factors such as cost, area and power were also considered but only as limiting factors. However, in the late 1990's it became apparent that this design philosophy was unsustainable. Complementary metal–oxide–semiconductor (CMOS) technology scaling allowed for higher densities and increasing clock frequencies, but performance-centered designs that tried to leverage these advances became hard or impossible to cool cost-effectively [5].

Currently energy efficiency is next to performance the major focal points in very-large-scale integration (VLSI) design. The driving forces behind this are increased portability, the power wall and environmental concerns. For portable battery-powered devices lower energy consumption directly translates into a more well-received product. The power wall, a direct consequence of discontinued Dennard scaling, means that technology scaling no longer is the obvious answer to increased performance and lower power [6][7]. Lastly, it is becoming painfully obvious that the rate at which the global energy consumption increases is not sustainable. ICs contribute to a considerable chunk of this increase [8].

To facilitate energy efficient design evaluation frameworks at the software and register-transfer level (RTL) are required to make vital early estimations. Early estimations are perhaps the most important estimations as changes at the architectural level have a larger impact on the final energy and performance numbers than changes at the circuit level. As such, these frameworks allow for accurate estimation and thereby more predictable prototyping results. Furthermore, these tools are significantly faster than those available at the circuit level which is essential when exploring a complex design space [9][10]. These framework have traded speed for accuracy, often adopting parameterizable models obtained through analytical or empirical studies of the underlying hardware. However, by adopting such models many of the existing frameworks neglects the impact of design integration, i.e., the synergy between the integrated parts of a design. We propose an open source energy evaluation framework for pipelines that facilitates software and hardware co-design.

The framework, named Chalmers RTL-based energy evaluation framework for pipelines (CREEP), extends down to the RTL which yields high accuracy and allows for detailed pipeline studies at the system level.

1.1 Goals and challenges

The aim of this thesis is to develop and demonstrate CREEP, a framework that estimates energy usage of integrated processor pipelines. The framework is based on an existing methodology that has been used in several published papers at Chalmers. The two major components that will be used to create the framework are; 1) Pipeline and cache RTL [11] and 2) A version of the SimpleScalar simulator [12]. The RTL and the simulator exist prior to the framework development but they are two separate and incoherent components combined in an ad-hoc manner. Hence, there are many challenges to address throughout the work, some of which are listed below:

- Create a scalable energy estimation framework from an ad-hoc methodology. Trade-offs between energy estimation accuracy and scalability are required throughout the development.
- The existing pipeline RTL code represents an in-order 5-stage pipeline (5SP) augmented with level-one instruction cache (L1IC) and level-one data cache (L1DC). The RTL code has been used in several projects that necessitated changes and quick patches in the code. As such, the code needs to be cleaned to make the RTL presentable and to reintroduce some features such as configurable cache sizes that is necessary for the framework. Additionally, a scalable power estimation methodology that includes the impact on power due to integration aspects needs to be implemented.
- The SimpleScalar simulator represents a more complex pipeline than the RTL. Thus, the simulator shall be modified to match the RTL code as closely as possible. This requires changes to the source code which must be verified to work as intended. Furthermore, the simulator will be modified to track additional resource usage information.
- The pipeline RTL and SimpleScalar components will then be combined by mapping resource usage obtained from the simulator to the RTL power estimations. The challenge here is to do the mapping in such a way that the framework estimates the processor energy adequately.
- Lastly, the framework will be automated to make it more approachable. The automation also serves the purpose of keeping the components coherent, which will make the results generated by the framework reproducible.

1.1.1 Goals

Several concrete goals related to CREEP have been identified and these are summarized below:

- Present a coherent and scalable framework with accuracy close to a placed and routed pipeline design.
- The framework should be automated through scripts which will make CREEP more user-friendly.
- The framework should support limited configuration, e.g., different cache configurations and processor speeds.
- Evaluate the applicability of the framework in a case study.
- Present the framework in a suitable forum to introduce it to the community.

1.2 Limitations

The framework development is complex and limitations need to be imposed on the development.

- The framework is only guaranteed to work as is. As such, any modifications made by the user to any of the framework components are not covered by the standard framework workflow.
- CREEP will be limited to the provided 5SP. Any changes to the RTL code are not guaranteed to work and the user needs to verify the changes in the context of the framework.
- CREEP does not include a level-two (L2) cache and does not attempt to approximate the impact of lower levels in the memory hierarchy on power dissipation and performance.
- CREEP is limited to the SimpleScalar source and configuration provided with the framework. Any changes to these components need to be verified and integrated into the framework by the user.
- This thesis will use the RTL for the pipeline and will only modify it to suit the needs of the framework. No performance enhancements will be done and the CREEP configurations are limited to run at 400MHz.
- The RTL will not be fabricated and no silicon of said design will be produced.

1.3 Related works

Energy evaluation frameworks have over time evolved from small frameworks limited to specific structures within a processor, to large and complex system-level frameworks. Depending on how the frameworks obtain circuit-level energy estimations they can be divided into analytical or empirical frameworks [5]. Analytical tools generally have the advantage of being more generally applicable to different architectures whilst empirical methods are best suited for the type of architectures from which they were derived. In this section previous energy evaluation frameworks and methods are presented.

The first high-level energy framework, CACTI, was released 1996 specifically targeting cache structures [13]. CACTI uses analytical models to estimate both power and delay within the cache structure. It has since its release been updated regularly to include leakage power, other types of memory cells, device scaling effects based on International Technology Roadmap for Semiconductors (ITRS) predictions and wire effects on delay and power [14]. The reason it targeted caches was that a significant amount of total chip power, up to 40%, was dissipated by the caches in embedded processors [9]. Furthermore, caches are highly regular structures thus less complex analytical models are needed to accurately estimate energy consumption and delay. It has allowed computer architects to explore trade-offs in the memory hierarchy design [13]. In contrast to CACTI, CREEP also models the datapath with which the caches are integrated. Hence CREEP provides an estimate for a complete integrated pipeline.

WATTCH and SimplePower, both released in 2000, analytically modeled power for a whole processor. WATTCH was one of the first tools to link a traditional architectural performance simulator, SimpleScalar [12], to analytical power models [15]. It bases its power estimations on a collection of parametrized power models for different hardware structures (for example random access memory (RAM), content addressable memory (CAM), other array structures, latches, buses, caches arithmetic logic unit (ALU)s) and per-cycle resource usage counts generated through cycle-level simulations using the SimpleScalar architectural simulator [9] [15]. SimplePower is an execution-driven, cycle-accurate RTL energy estimation tool that uses a combination of analytical and transition sensitive energy models [16] [17]. The SimplePower framework is built around a five stage datapath with instruction fetch, decode, execution, memory and write-back stages [16]. Transition-sensitive models are defined for each functional unit in the datapath and the models contain switch capacitance on a per-input basis obtained from VLSI layouts and extensive circuit simulation [17]. Models are provided for several technology nodes. SimplePower uses a combination of analytical and transition-sensitive energy models for the memory system. The analytical models are reserved for the memory arrays whereas the transition-sensitive models are used for the connecting buses [17]. In contrast to the functional units, the switching capacitance of these buses is based on pessimistic assumptions rather than HSPICE simulations [17]. The control path of the 5SP has been neglected because developing transition-sensitive models for this was consid-

ered extremely difficult. SimplePower leverages on the SimpleScalar simulator by using the same ISA and compiler. The framework simulates the generated executables providing cycle-by-cycle energy values based on the aforementioned models [16]. Both of these were fast and usefully accurate to quantify potential power savings in architecture design. However, compared to CREEP, WATTCH and SimplePower, while being more flexible, again fail to capture the integration aspect that CREEP addresses.

McPAT, another analytical tool, is an abbreviation of multicore power area and timing. The framework was released 2009 and it estimates power, area and timing which enables architects to use metrics that relate performance to both area and power [10]. In contrast to SimplePower and WATTCH, McPAT is compatible with any performance simulator through an XML interface. Furthermore, McPAT is built on more accurate analytical models compared to WATTCH and these models also include static and short-circuit power. Just as the name implies it also handles the complexities of multicore architectures. Similarly to McPAT, CREEP provides a system perspective but does so more accurately as power estimates are obtained from an RTL implementation and not analytical models. However, CREEP supports less complex systems as it targets simple embedded processors.

One empirical framework of interest is IBM's PowerTimer [15]. The major difference from the previous approaches that are based on analytical models is primarily the formation of the energy models. PowerTimer's models are based on empirical data collected from existing microprocessors. These models are then scaled to capture device scaling. PowerTimer takes a bottom-up approach and the energy models are derived from circuit-level power simulation data. Low-level circuit macros are analyzed and used to generate higher-level energy models for microarchitectural units [15]. These models are then controlled by two sets of parameters; 1) technology and circuit parameters, 2) microarchitectual parameters such as buffer sizes, pipeline latencies and bandwidth values. The microarchitectual parameters are also used in a stand-alone performance simulator. By connecting the performance simulator with the energy models a total or cycle-by-cycle energy evaluation can be performed. IBM's PowerPC architecture was used to create the energy models and as such was best suited for design exploration within that microarchitecture. CREEP is likewise limited to the specific architecture implemented in RTL. Both frameworks work at the system level but PowerTimer chooses to distance itself from the physical implementation through parameterized models which lends it greater flexibility at the expense of accuracy.

Yet another example of an empirical framework was proposed by Aziz et al. [18]. This framework is used for marginal-cost analysis. Their approach was to first create architectural models using design space sampling and statistical inference to capture the multi-dimensional space of microarchitectural parameters. The energy-delay trade-offs of the composing circuit blocks that formed the architectures were then stored in a circuit library. The created joint architecture-circuit design space was then combined with exploration engine which is given an optimization objective

and resource budgets [18]. The exploration engine then searches the design space to find the most efficient configuration under the given constraints. As it is a high-level framework, it is more flexible than CREEP but trades aspects such as accuracy and integration to achieve this flexibility.

Rance Rodriges et al. conducted a study in [19] on the usage of performance counters and how they can be used to estimate power in microprocessors. While performance counters have been widely used to estimate power online in situ the counters used vary widely between processor architectures. Rance Rodrices et al. attempts to identify a set architecture-agnostic counters that estimate processor power with low error. Two architectures, Intel Atom and Nehalem, at opposite ends of the design spectrum were used to select performance counters which in both architectures showed a strong correlation to power. Using SESC architectural performance simulator and WATTCH as reference they concluded that *#Fetched instructions, #L1 hit* and *#Dispatch stall* counters was sufficient to approximate processor power with an average error of 5%. Furthermore, the chosen set of counters variation between processor architectures only had a small impact, 3%, on the estimation accuracy. While the objective of this work is different from CREEP, it indicates what performance counters are relevant for a selection of architectures, albeit at higher performance design point, and can serve as an inspiration for CREEP.

A high-level estimation methodology and the associated tool, SoftExplorer, was presented in [20]. The methodology models a processor through functional analysis and a parametric software model is used to capture the software's impact on power. The processor model can be as coarse grained as a functional block diagram. The parametric software model accepts relevant algorithmic parameters such as cache miss rate. The first step in the methodology is to cluster the processor model into functional blocks that are concurrently activated when code is running. The relevant consumption parameters are chosen as the links between the functional blocks. The second step is to characterize the processor model's power consumption as the architectural and algorithmic parameters are varied. Lastly, a curve fitting of the graphical representation of the characterized power is performed through regression analysis. SoftExplorer was compared to SimplePower where the tool was found to be significantly faster and within 2.4% of the estimates. Compared to CREEP, SoftExplorer sacrifices accuracy for flexibility and speed and neglects the integration aspect covered by CREEP.

2

Background

This chapter will provide the reader with the basic knowledge to understand the concepts used to develop Chalmers RTL-based energy evaluation framework for pipelines (CREEP). First, the basics of CMOS logic with a focus on implementation, i.e., power and speed will be discussed in Sec. 2.1. Since complementary metal–oxide–semiconductor (CMOS) is the primary fabrication technology used to implement integrated circuits (ICs), CMOS speed and power are central to this work. Secondly, the basics of IC design with a focus on cell-based CMOS designs are presented in Sec. 2.2. The foundations of computer architecture with a focus on pipeline design and physical implementation are presented in Sec. 2.3. Lastly, the existing ad-hoc methodology which this work is based on is presented in Sec. 2.4.

2.1 CMOS

The abbreviation CMOS stems from the structure of the device as it was composed of at least one n-type metal–oxide–semiconductor (nMOS) and one p-type metal–oxide–semiconductor (pMOS) transistor [21]. The simplest CMOS circuit, the CMOS inverter, is shown in Fig. 2.1. The arrangement of the inverter is such that the input of the CMOS inverter is connected to the gate terminal of both transistors. Whilst the transistors' behavior in reality is more complex, they can ideally be viewed as switches that close and open when a voltage transition is detected on the gate. The behaviors of the nMOS and pMOS are opposite that of each other, i.e., when a potential V_{DD} (supply voltage, logic 1) is asserted on gate terminal the nMOS closes and the pMOS opens. Conversely, when no potential or GND (ground, logic 0) is present on the gate the nMOS opens and the pMOS closes.

2.1.1 Power dissipation

The power dissipation of a CMOS circuit is generally considered to be composed of three components; 1) Dynamic power, 2) Short-circuit power and 3) Static power [22]. The total gate power dissipation is given as the sum of these components as shown in Eq. 2.1.

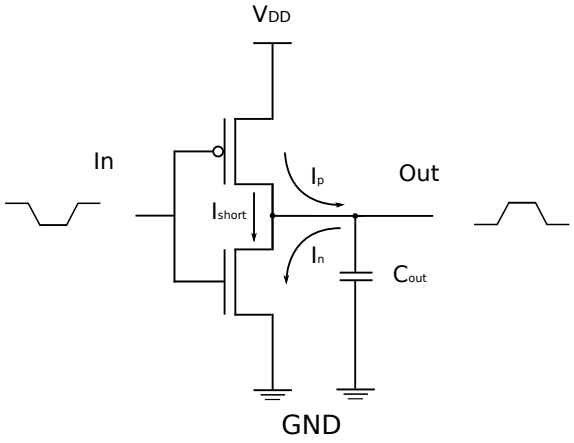

Figure 2.1: Schematic view of CMOS inverter

$$P_{total} = P_{dynamic} + P_{short} + P_{static} \tag{2.1}$$

The dynamic power dissipation is by far the most dominant source of power consumption in a CMOS circuit [22]. Dynamic power is also called switching power because power is consumed when the gate is switching, i.e., charging or discharging the gate output capacitance C_{out} to V_{DD} or GND [21]. The output capacitance consists of several components; C_{int}, C_{wire} and C_{load} as shown in Eq. 2.2 [22].

$$C_{out} = C_{int} + C_{wire} + C_{load} \tag{2.2}$$

The internal capacitance C_{int} is related to the structure of the gate and include parasitic capacitances. C_{wire} is the capacitance of the wire that connects the output of the device to the input of another CMOS gate which in turn constitutes the C_{load} capacitance. Consider Fig 2.1 where a voltage transition from V_{DD} to GND is asserted on the input. The nMOS transistor opens and the pMOS transistor closes. A current I_p flows from the voltage supply to the output capacitance which charges the capacitance. The amount of charge pulled from the supply is given by $C_{out}V_{DD}$ and the energy drawn from it by $C_{out}V_{DD}^2$. However, half of the energy drawn from the supply is dissipated as heat in the resistance posed by the pMOS transistor so the energy in the output capacitance is given by $E_c = 1/2C_{out}V_{DD}^2$. When the input voltage later is increased to V_{DD} the pMOS opens, the nMOS closes, and the output capacitance is discharged as a current I_n flows to ground. The stored energy in the capacitance E_c is dissipated in the resistance posed by the nMOS transistor. If this circuit is operated at a clock frequency f and and the output switches with a probability of α the total dynamic power drawn from the supply is given by Eq. 2.3 [21].

$$P_{dynamic} = C_{out}V_{DD}^2\alpha f \tag{2.3}$$

The short circuit power dissipation P_{short} is nowadays considered a small component of the total power [22]. The power is dissipated when the output of the gate switches. The nMOS and pMOS devices are in reality not behaving as ideal switches and require a finite time to open and close. This time is determined by how long the input voltage remains between the transistors threshold voltage V_{tn}, and $V_{DD} - V_{tp}$, where V_{tn} and V_{tp} are the threshold voltages of the nMOS and pMOS transistors respectively. Threshold voltage is the minimum gate to source potential that is needed to create a conducting path in the transistor, i.e., close the switch. Consequently there is a small period of time when both transistors are on and a current I_{short} shown in Fig. 2.1 is allowed to pass from the supply to ground, which consumes a small amount of power.

The last component is the static power dissipation that is intermediate in size compared to the previous components [22]. It is smaller than the dynamic power and has historically been negligible. It is called static because it is omnipresencent in all CMOS circuits that are powered. The static power stems from a collection of different currents passing between the various terminals of the devices most notably source to drain. The leakage power is closely connected to the threshold voltage and the temperature of the device [21]. As the feature size of the transistor is shrinking below 65nm, leakage power is increasing and in more recent technology nodes it has become a considerable contributor to the total power dissipation.

2.1.2 Speed

As discussed in the previous section, a transition on the output of a CMOS gate does not happen instantaneously as the current would have to be infinite in magnitude. Naturally, this is not the case in a real CMOS circuit. Instead, the current is determined by the transistor's ability to drive it, which due to nonlinear I-V and C-V characteristics is no simple thing [21]. However, the transistors can be approximated by an RC-delay model that allows the transistors to be viewed as simple RC circuits, which most electrical engineers are familiar with. R is the transistor's effective resistance that is the product of the V_{ds} and I_{ds}, i.e., the potential between the drain and source terminal and the current passing through the drain source junction [21]. The capacitance is the output capacitance of the CMOS circuit (see Sec. 2.1.1). The transfer function of the equivalent RC circuit is given in Eq. 2.4 and the step response in Eq. 2.5.

$$H(s) = \frac{1}{1 + sRC} \tag{2.4}$$

$$V_{out}(t) = V_{DD}e^{-t/\tau} \tag{2.5}$$

Solving the step response for $V_{out}(t) = 1/2V_{DD}$ gives the propagation delay through the CMOS circuit shown in Eq. 2.6.

$$t_{pd} = RC \ln 2 \tag{2.6}$$

The propagation delay is an approximation of how fast the output of the CMOS circuit transitions from V_{DD} to $1/2V_{DD}$ when an input step is asserted on the input of the circuit. A non-trivial CMOS circuit is composed of many CMOS circuits which are connected as shown in Fig. 2.2 and the propagation delay from input *In1* to the final output *Out n* can become significant.

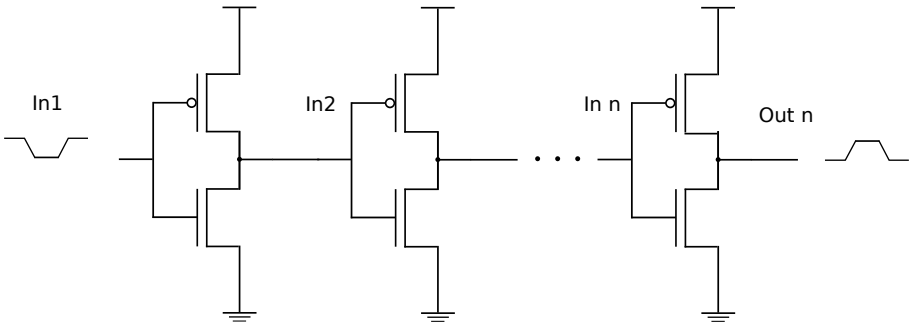

Figure 2.2: Example of a CMOS circuit consisting of multiple inverters

To manage the delay, the current drive capability of the transistors in the CMOS gate can be increased. This is done through transistor sizing whereby the widths of the pMOS and nMOS transistors are increased [21]. Essentially this reduces the effective resistance experienced by the current and a larger current is allowed through the circuit. However, increasing the transistor size also causes an increase of the gate capacitance, i.e., the output capacitance experienced by the driving gate in the circuit resulting in higher power dissipation. Moreover, gates that are unsized and connected to the resized gates will have to charge a larger load capacitance, which slows down unsized parts of the design.

2.2 IC design

Modern ICs are immensely complicated circuits often composed of several millions, if not billions, of transistors. Designing such complex beasts is without computer aid simply beyond the capabilities of a human designer. To facilitate IC development software assistance is key throughout the design process. The software tools providing this assistance are collectively refered to as electronic design automation (EDA). The term EDA spans a wide range of functionality required throughout the design of an IC, which will be the focus of this section.

Designing ICs is complex and it was discovered early on that doing so at the gate level, even with the aid of EDAs specializing in the practice, was too cumbersome. As a response, tools were developed to create gate-level representations, called netlists, from a specification at a higher level of abstraction through a process called logic synthesis. These abstractions are usually expressed in a hardware description language (HDL) such as verilog or VHDL. These design languages allow the designers

to express the behavior of the logic circuits at the register-transfer level (RTL) in the sense that an assignment to a register expresses functionality.

The process of designing an IC is composed of several stages and for digital circuits these are design, functional verification, logic synthesis and place and route (PnR). The initial design stage is followed by functional verification, which is first done at the register transfer (RT) level and infers testing that design described in HDL matches the expected functional behavior. This is normally done at the cycle level by applying stimuli to the design whereby the logic transitions of the output can be observed and compared with the desired behavior. The stimuli is commonly supplied by a testbench that provides input from a set of test vectors [23]. The test vectors can be selected with the intent of testing specific functionality (directed testing) or randomly to test corner-cases [23]. A key concern when selecting test vectors is coverage that can be defined as how large part of the design that has been tested (in percent). The RTL verification is facilitated by an HDL simulator tool. There are many different HDL simulators available such as ModelSim from Mentor Graphics, IES from Cadence and VCS from Synopsys [24][25][26].

After the RTL has been verified the design is brought through a cell-based logic synthesis with the aid of a synthesis tool. The designer supplies the RTL design together with design constraints with regards to timing which will guide the synthesis tool through the multiple stage process that is cell-based logic synthesis [23]. The cell library, which contains the standard gates (cells) used for synthesis, is provided by silicon foundries such as ST Microelectronic or TSMC. The cell libraries are unique to each manufacturer as they are tightly knit to their manufacturing processes. For each cell in the library, parameters such as size, internal power dissipation, leakage power and input pin capacitance are defined [23]. Normally several libraries are necessary to fully evaluate a process technology. The libraries are optimized for different design points, e.g., low power (LP) and general purpose (GP). The GP cell library is optimized for performance and the LP cell library for low power designs. Furthermore, the GP and LP libraries are further divided into sub-libraries with different threshold voltages, which allows for fine grained control of performance and power dissipation. Higher performance can be achieved by using a low-threshold voltage version but at the price of higher leakage power dissipation. Conversely, for design where power dissipation is a cause for concern, a high-threshold voltage version is a good choice as these are slower but have lower leakage-power dissipation. It is up to the designer to choose a library that suits the application at hand. Small variations in the manufactured design can have a large impact on cells' behavior. To capture these variations, design corners are used. The worst-case corner contains cells that have the worst possible (and still producing working devices) variations that affect speed negatively. Conversely, the best-case corner cell library has the best variations. Naturally, there is a nominal cell library that falls in between the two. Moreover, the cell libraries have been characterized for different temperatures and voltages. Temperature and voltage depend on in situ conditions and also affect the behavior of the final circuit. As such, every cell library exist in several models with different temperatures and voltages.

Synthesis is a complex process and EDA tools that specialize in the practice are available from different suppliers such as Encounter RTL Compiler from Cadence, Design compiler from Synopsys and HDL Designer from Mentor Graphics [27][28][29]. The different tools provide similar functionality but differ in the algorithms and heuristics used during the synthesis process. The synthesis results in a gate-level netlist, a sequence of standard cell logic gates realizing the functionality of the RTL code. In contrast to the HDL description of the design that solely captures the intended functionality, the netlist also includes parameters such as area, timing and power. The synthesis tool strives to meet the imposed timing constraint using the cells from the specified libraries. It accomplishes this through static timing analysis which allows it to find and balance the critical paths in the design [23]. This balancing act entails selecting gates with sufficient current drive capabilities for the entire circuit to switch within the timing constraint. As such, the same design will produce different gate-level netlists with different area and power. At the synthesis level, the functional verification amounts to ensuring that the netlist behaves the same as the RTL design. This is achieved through equivalence checking or simulation-based methods as described for RTL verification.

Lastly, the design netlist is brought through PnR which is a physical design phase composed of three steps; 1) Floorplaning where the design's blocks are organized, 2) Placement of standard cells and iterative optimization of placement and 3) Routing of standard cell interconnects, power lines and clock tree [23]. The process is strictly guided by design rules imposed to ensure that the placed and routed design is manufacturable. The most significant changes to the netlist are the addition of wires and clock tree. Wires constitute a part of the nodal capacitance described in Sec. 2.1.1 which in some cases necessitates larger, more powerful gates to be used. The addition of wire capacitance and larger gates with higher internal capacitances increases the power of the design. Furthermore, the clock tree is a significant contributor to the design power and is only included after the design has been placed and routed. This stage relies on EDA tools, such as Encounter from Cadence, that specialize on PnR as the burden of placing thousands of gates is simply beyond the capability of a human designer [30].

Implementation power closure is important for power constrained circuits, e.g., portable embedded processors. As such, methods for obtaining power closures are also included in many IC design flows. The power dissipation of CMOS-based circuits comes from active device switching and leakage where the former are the main contributor as discussed in Sec. 2.1.1. The switching powers are then summed over all capacitive nodes in the design. While the power estimates could be done prior to the PnR, the power would be underestimated as the nodal capacitance is greatly increased by the wire capacitances. The power also depends on the switching activity (see Sec. 2.1.1) of these nodes and there are different techniques used to approximate it. One such technique is probabilistic testing where the input statistics, asserted by a designer, are propagated to each node in the circuit [22]. However, this creates an nondeterministic polynomial time (NP) complete problem so the scope at which this is done must be limited, e.g., parts of the design are analyzed instead of the

whole design.
Another is use-case based switching activity which is facilitated through simulation-based methods as described for RTL verification [22].

2.3 Pipeline design

The most fundamental parts of computer architecture are the instructions that define what a computer is capable of and the microarchitecture that decides how it executes the instructions. To that end the structure of a complete set of instructions called instruction set architecture (ISA) and more specifically the MIPS I ISA, is presented in Sec. 2.3.1. Microarchitectural concepts relevant to this thesis such as pipelining and caching are then discussed in Sec. 2.3.2.

2.3.1 MIPS I instruction set architecture

All computer programs are made up of instructions which are the basic operations carried out by a processor [2]. Instructions are usually very frugal and each of them normally does one basic operation, e.g., memory access, arithmetic or flow control. To make up a complex computer program many different instructions are needed. The instructions are grouped together to form a set of instructions possibly unique to their implementing architecture thus forming an ISA.

The MIPS I ISA, first released in 1982, was developed by John Hennessy and his colleagues at Stanford [31][2]. MIPS I was one of the first successful reduced instruction set computing (RISC) ISAs built on four main principles; simplicity favors regularity, make the common case fast, smaller is faster and that good design demands good compromises. Derivatives of the MIPS ISA are still used today by CISCO (routers), Nintendo and Sony (hand held gaming consoles) and Silicon Graphics among others.

MIPS I instructions can have three different encoding formats referred to as R, I and J-type instructions in the literature [2]. By only allowing a limited number of instruction formats the ISA gains regularity which simplifies the instruction decoding [2]. Each instruction has its own operation (OP) code which is encoded in the op-field shown in Figs 2.3-2.5. Besides the OP-code, the main difference between the instruction formats is the number of operands that are encoded in the instruction. R-type instructions however, need two extra fields (shamt and funct) to characterize each operation, which includes mathematical or logical operations such as addition, subtraction and shift operations. R-type instructions require two operands encoded in the rt and rd fields shown in Fig. 2.3. In contrast, I-type instructions require just one operand encoded in the rt field (see Fig 2.4) and lastly J-type require no operands.

The operands are fetched from a small register-file whose modest size lends it speed. The size of the register-file and how the memory is addressed are the parameters, besides instruction width, that the MIPS I ISA enforces on the underlying microar-

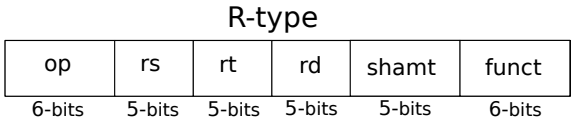

Figure 2.3: MIPS I R-type instruction format.

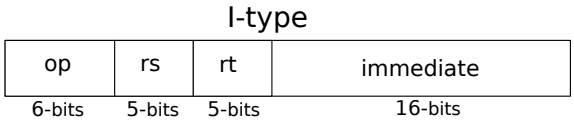

Figure 2.4: MIPS I I-type instruction format.

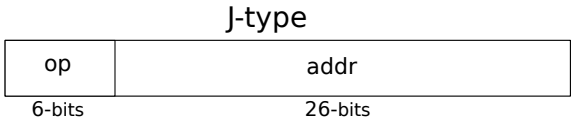

Figure 2.5: MIPS I J-type instruction format.

chitecture. MIPS is a load-store ISA because all operands are fetched from the register-file [1]. I-type instructions substitute one operand for a value, called immediate value, which is encoded directly in the instruction itself. Examples of I-type instructions are load and store operations. Stores read data from the register-file and store the data in data memory. Loads on the other hand read data from data memory and store the data in the register-file. Similarly, all R-type instructions also store the result of the operation back into the register-file to a location indicated by the rs field. However, other I-type instructions called control flow instructions, e.g., branch instructions, do not access the register-file. Instead, the control flow instruction decides the order in which the instructions in the program are executed. Lastly, J-type instructions, which mainly include another type of flow control instructions called jump instructions, trade both operands for a larger immediate value. MIPS I also defines the format of the operands. Operands of 8-bit (ASCII characters), 16-bit (Unicode characters, half word), 32-bit (integers, word), 64-bit (long integer, double word) and IEEE 754 floating point in 32-bit (single precision) and 64-bit (double precision) are allowed [1].

If the aforementioned register-file were the only storage available to a processor computer programs would be very limited in size. However, as implied above, another type of memory that is larger in size is usually available. The MIPS I ISA defines how the processor interfaces with memory by specifying how the memory is addressable. MIPS I specifies two ways of addressing memory; 1) byte-addressable or 2) word-addressable. This means that the smallest addressable data unit is a byte (8 bits) while the largest is a word (4 bytes) as illustrated in Fig. 2.6. All memory accesses must be aligned to either a byte or word access, otherwise the access is unaligned and erroneous [1]. In the same figure to the left, the address of the corresponding data is shown. The address used to access the memory is generated by the memory operation (I-type instruction) by adding the immediate value with the register indicated by the rt field.

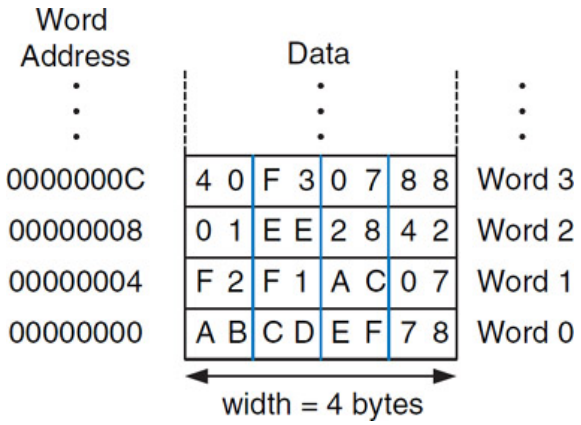

Figure 2.6: MIPS I memory is byte addressable [1].

Furthermore, MIPS I supports five ways of generating memory addresses through so called addressing modes; Register-only addressing, Base addressing, Immediate addressing, PC-relative addressing and pseudo-direct addressing [2]. Register-only addressing has already been described, all R-type instruction uses this addressing mode. Base addressing is used by some I-type instructions, such as stores and loads, and has likewise been described. Immediate addressing is similar to base addressing but it does not use the register pointed to by the rt field (I-type). Program counter (PC) relative addressing is used by conditional branch instructions (I-type) where if a condition holds true, the PC is added to the immediate field to produce the final address. Lastly, pseudo-direct addressing is used by J-type instructions where the larger address field (see Fig. 2.5) is first concatenated with the four most significant bits of the PC.

2.3.2 A MIPS I pipeline

An ISA does not define the implemented hardware besides register-file and addressing modes. A distinction is made between an architecture and a microarchitecture where the latter includes implementation details. This means that two different processor architectures can support the same ISA while being fundamentally different at the microarchitecture level. In this section a MIPS I compliant microarchitecture will be presented. The microarchitecture utilizes pipelining, which is a concept that is used in most modern processors that offers higher performance at the expense of design complexity.

The speed of a processor, and systems in general, depends on latency and throughput of the data passing through it [2]. Low latency is preferred for systems that are required to be responsive and deliver results in a timely manner. In contrast, throughput is beneficial for systems that prioritize computational performance over timeliness. Latency and throughput are often contradictory in the sense that measures that improve one degrade the other [2]. In general-purpose computing, throughput has historically been more important than latency. Throughput can mainly be improved by exploiting instruction-level parallelism (ILP), i.e., by executing mul-

tiple instruction at the same time. Parallelism can be divided into spatial and temporal parallelism [2]. Spatial parallelism entails executing more instructions simultaneously by utilizing increased computational resources. In contrast, temporal parallelism implies dividing the existing computational resources into discrete steps where each step is utilized by different instructions. Spatial parallelism has the benefit of increasing throughput with little or no impact on latency [2]. However, spatial parallelism requires additional hardware resources and results in larger and more complex designs. Conversely, temporal parallelism sacrifices latency to increase throughput while only requiring limited hardware additions to the design in the form of a few registers and control logic.

In the context of microarchitecture temporal parallelism is more commonly referred to as pipelining and the concept has been used in most processors for the last three decades [32]. Pipelining is implemented by dividing a processor's data path, i.e., computational resources, into stages separated by pipeline registers which limit the logic paths of the design to that between two consecutive pipeline registers. In effect, the design can meet stricter timing constraints and thus run at a significantly reduced cycle time. The reduced cycle time allows the design to be clocked at a higher rate, which causes a reduction of the execution time since the pipelined processor executes (ideally) one instruction each cycle. The discrete stages allow the pipelined processor to achieve temporal parallelism with several in-flight instructions.

The goal of a pipeline design is to evenly distribute the datapath's logic between the different pipeline stages. In a perfectly balanced n-stage pipeline the cycle time of the design is roughly $1/n$ of the cycle time of a corresponding un-pipelined designs [1]. However, in practice the stages in the pipeline are seldom balanced perfectly resulting in some stages requiring more time to finish their execution. The critical path, which imposes the lower bound on the design cycle time, is thus found in these stages. Furthermore, pipelining also introduces some performance overhead. A small part of this overhead is the delay introduced by the inserted pipeline registers but the by far more substantial overhead is caused by dependencies between instructions moving down the pipeline [2]. These dependencies are called hazards and will be discussed in greater detail later in this section. In short, hazards increase the cycles per instruction (CPI) or instruction per cycle (IPC) which has a detrimental effect on performance. The overhead caused by the pipeline registers and hazards increases the latency of each individual instruction in a pipelined processor [1].

An example of a pipeline implementing the MIPS ISA is shown in Fig 2.7. The pipelined processor performs operations in five discrete stages separated by pipeline registers as shown in the figure. The stages are instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM) and write-back (WB).

Fig 2.8 shows the same pipeline as Fig 2.7 but it also shows the control unit. The control unit generates control signals in the decode stage and the signals are propagated alongside the instruction in a control-path that in each stage reflect the instruction's individual needs.

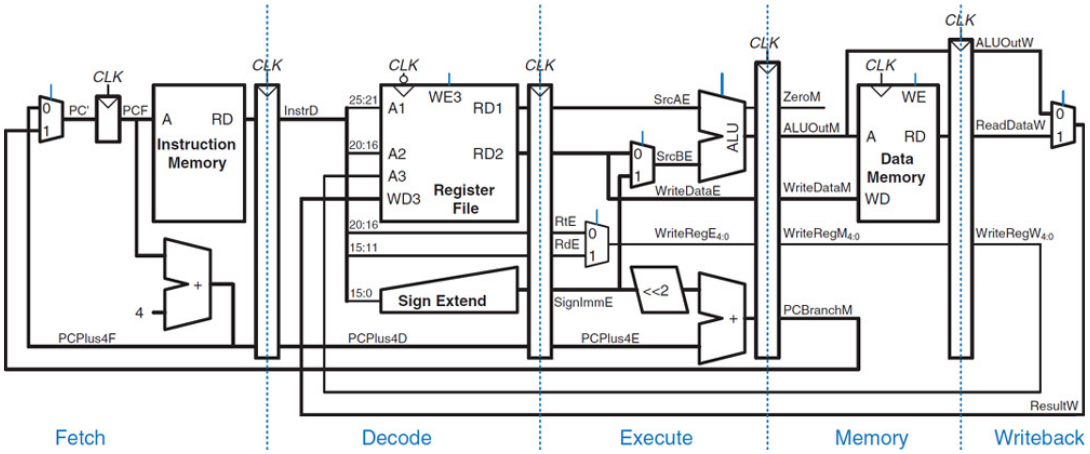

Figure 2.7: A MIPS I 5SP [2]

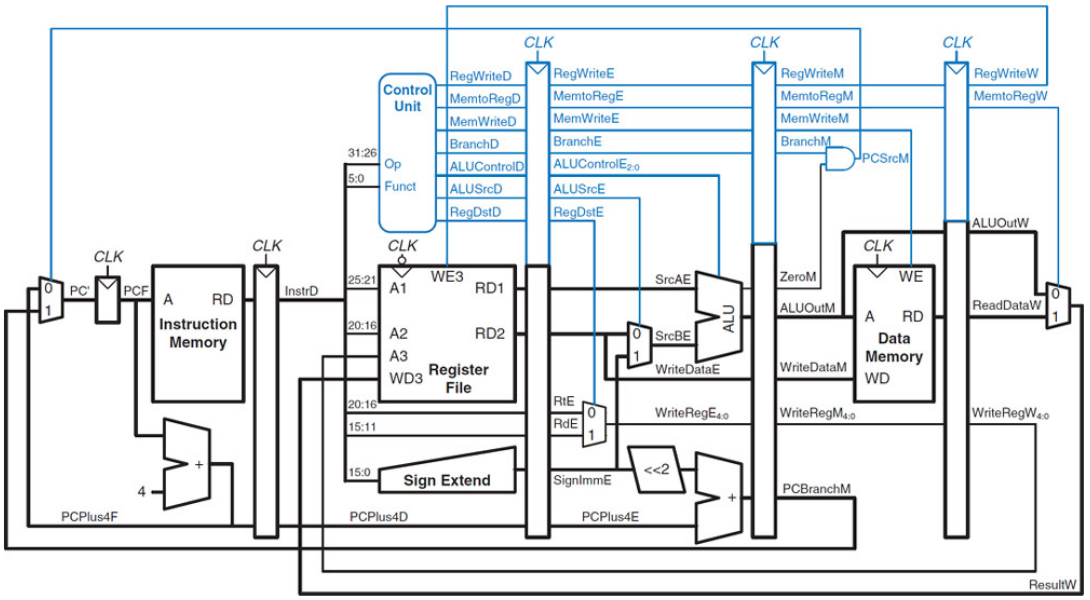

Figure 2.8: A MIPS I 5SP augmented with a control unit [2]

All instructions traverse the datapath one stage at a time and need five clock cycles to fully traverse the pipeline. Fig 2.9 illustrates an example of an instruction flow. The first instruction is fetched in the first cycle and stored in the subsequent pipeline register. In cycle two a new instruction is fetched while the preceding instruction is decoded, both of the instructions are stored in the respective pipeline register after the stage they passed through. In the third and fourth cycle yet another instruction is fetched while the later instructions proceed through the pipeline. In cycle five the pipeline is utilized fully with an instruction being executed in all stages. The first instruction has now cleared the pipeline and is written back (if R-type or load) to the register-file. After cycle 5 the pipeline should ideally remain fully utilized until the program is completed.

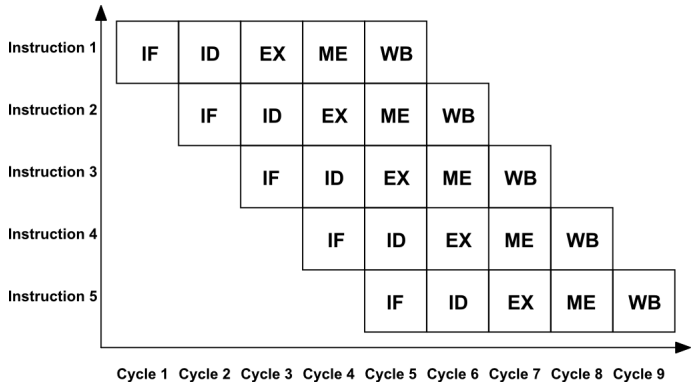

Figure 2.9: Instruction sequence in a 5SP.

However, in reality pipelined processors do not achieve full utilization in most cases because of the occurrence of pipeline hazards. Hazards are defined as dependencies between consecutive instructions in the pipeline. Hazards are divided into three categories; data hazards introduced by arithmetic and load/store instructions, control hazards introduced by flow-control instructions, e.g., branches, and lastly structural hazards where in-flight instructions compete for pipeline resources. Data and control hazards will be explained in greater detail but structural hazards, which are non-existent by design, will not be elaborated on.

Data hazards occur when a subsequent instruction needs data generated by a previous instruction. For instance an add instruction is followed by a subtraction that uses the value produced by the addition. The addition instruction is unable to reach the WB stage before the subtraction clears the ID stage and finishes the register-file access thus entering the EX stage with incorrect operands. This is called a read after write (RAW) hazard and if unaddressed would lead to program errors. A less elegant solution would be to stop the IF and wait for the instruction causing the dependency to write back its result to the register-file. While simple, stalling the pipeline increases the CPI and incurs performance losses. A more elegant solution is forwarding. It is possible for the addition to provide the correct value to the subtraction by passing it to the subtraction as it enters the EX stage. The addition forwards the data causing the dependency to the subtraction. The pipeline shown in Fig 2.10 has been augmented with a forwarding unit that controls the added forwarding paths between the EX, MEM and WB stages. The forwarding unit reads the Rs and Rt registers of the instruction entering th execute stage and compares it to the Rd of the instruction entering the MEM and WB stage if this instruction is a R-type instruction and forwards data as needed.

Forwarding does not solve RAW hazards where a dependency exists between a load and a subsequent instruction. Assume a load followed by an addition: The load needs to propagate to the WB stage before the data is brought from memory. At this point however, the addition has already passed the EX stage and is entering the MEM stage. The only solution to this problem is to stop the addition from propagating in the pipeline by stalling it. This allows the load to propagate to the

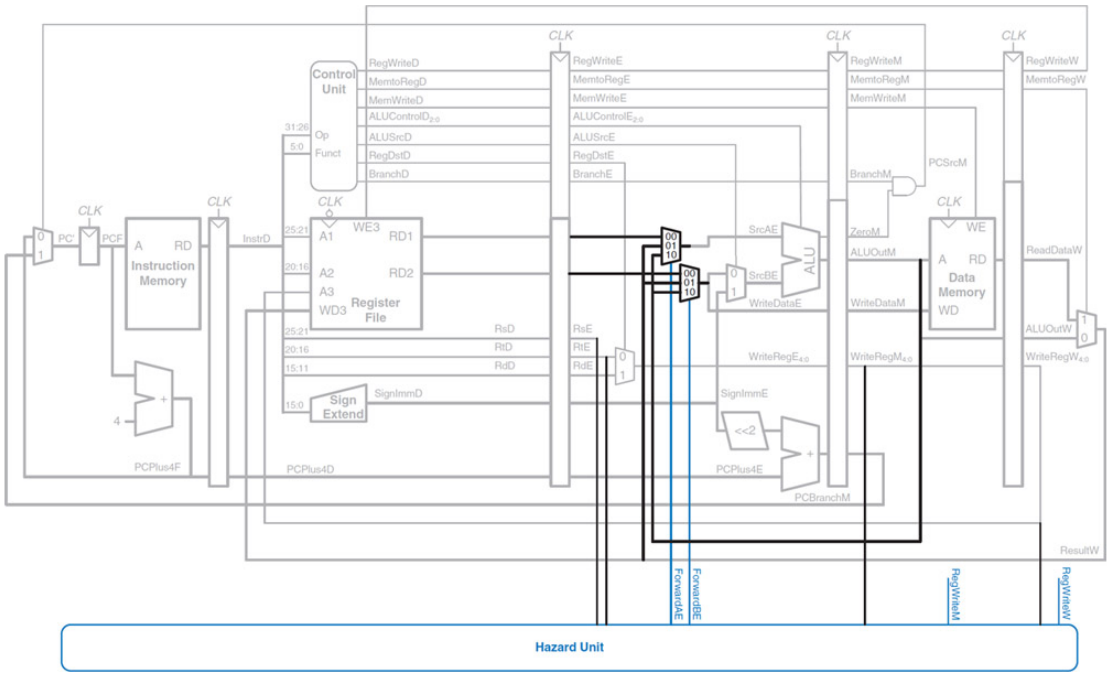

Figure 2.10: The 5SP augmented with a hazard detection unit [2]

WB stage where the load is able to forward data to the addition waiting in the ID stage. The necessary addition to the pipeline and hazard detection is shown in Fig 2.11. The pipeline register between IF and ID stages now has an enable signal that when asserted forces it to hold its contents. The pipeline register between the ID and EX stage has an additional clear signal that sets the register contents to zero which effectively stops random data from propagating down the pipeline after the load instruction. Instead, the pipeline stages after the load are idle or conceptually executing a no-operation (NOP) instruction. Additional inputs to the hazard detection unit are added to allow it to detect hazards that require stalls.

Control hazards are caused by branch and jump instructions because they update the PC. Assume that a branch instruction is fetched from instruction memory. The pipeline is oblivious to the fact that the branch could redirect the IF to a different portion in the program and erroneously continue to fetch instructions sequentially. When the branch is resolved to be taken in the EX stage (see Fig 2.8) two instructions from the wrong execution path have already been fetched. The pipeline would then need to be flushed (pipeline registers emptied). Alternatively the pipeline could be stalled, i.e., instruction fetch halted. Both solutions would degrade performance by increasing the CPI. A better solution is instead to use a delayed branch slot. The delayed branch slot scheme relies on the compiler to move an instruction originally placed before the branch to immediately behind it. The compiler must be able to ensure that no dependencies are created when moving the instruction [1]. This scheme works reasonably well in the pipeline in Fig 2.8, but would still require one stall cycle for the branch to be resolved in time for the instruction after the delayed branch slot. However, this stall cycle can be avoided by moving the branch resolution to the ID stage as depicted in Fig 2.12 below.

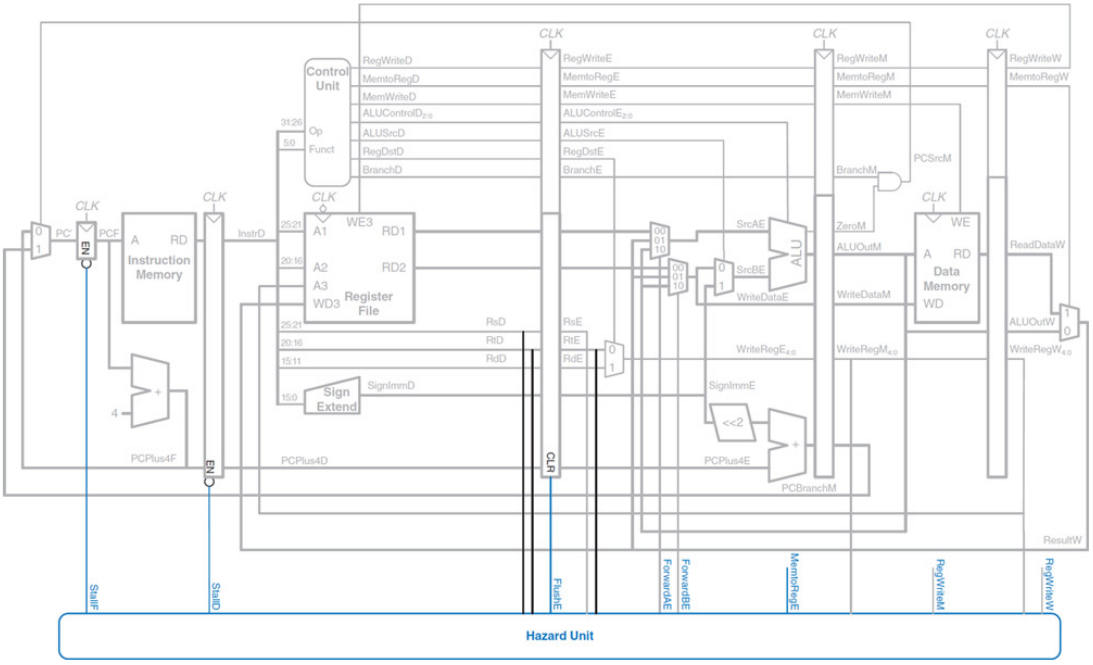

Figure 2.11: The 5SP with stall support [2]

A dedicated comparator has been added in the ID stage that operates immediately on the fetched register contents. Likewise, the sign extension and address generation have also been moved to the ID stage.

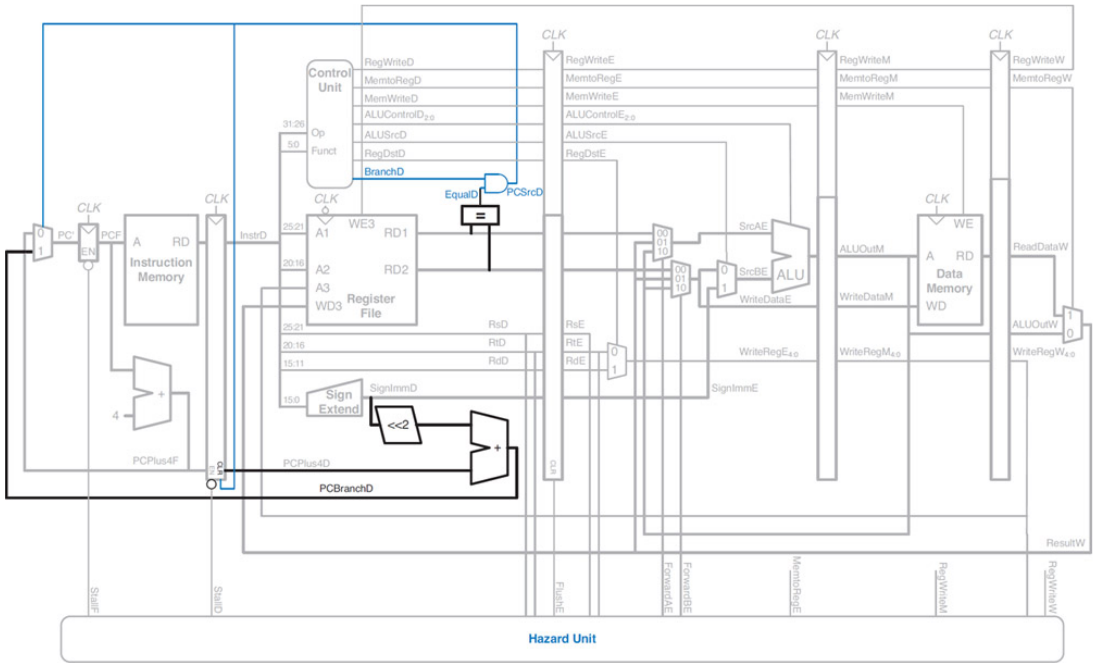

Figure 2.12: The 5SP with branch resolution in the ID stage [2]

While the stall cycle is eliminated in the pipeline in Fig 2.12 the early branch resolution introduces additional RAW hazards. The branch condition could possibly depend on a preceding instruction about to enter the EX stage and the lack of forwarding paths from the EX stage to the ID stage where the branch is about to be resolved could result in erroneous branching. However, forwarding paths can be added and the hazard-detection unit could be expanded to detect and handle this forwarding as shown in Fig 2.13.

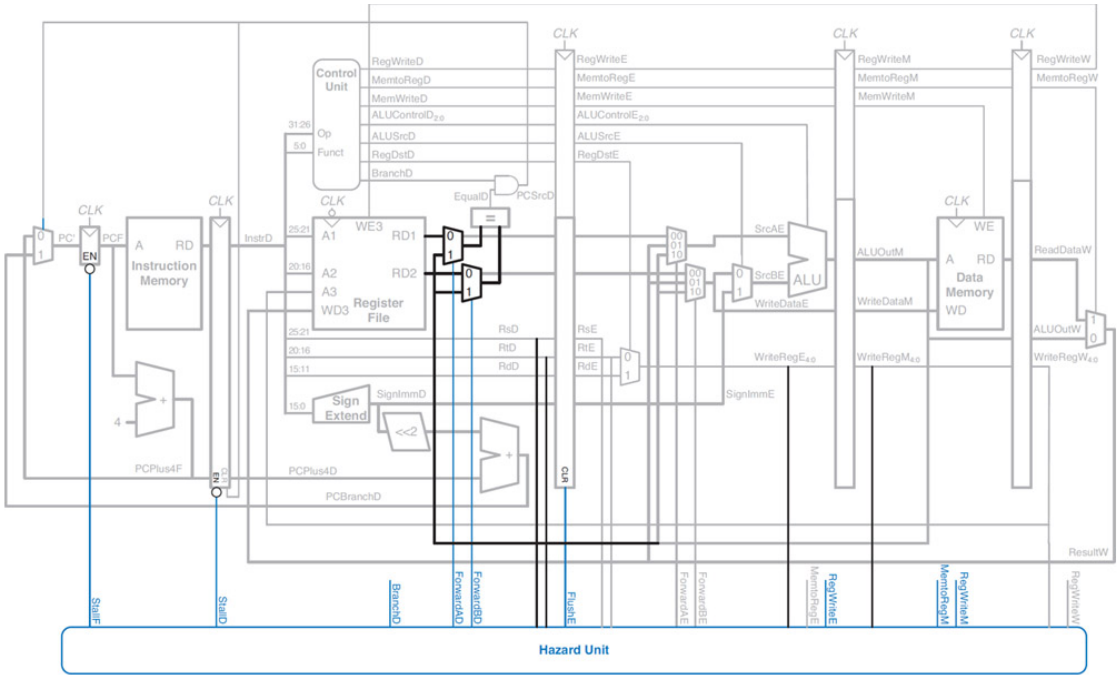

Figure 2.13: The 5SP with branch resolution in ID stage with added forwarding paths [2]

In this section a simple MIPS I 5-stage pipeline (5SP) was outlined. More advanced pipelines are in use today and these are usually deeper than five stages. However, deeper pipelining increases the occurrence of data hazards, which necessitates a more complex control path. Adding more stages further decreases the logic per stage, but increases the number of dependencies at the same time and ultimately deeper pipelines will be stalled more than their simpler counterparts. Furthermore, because of the minute logic in each stage, the setup time and input to output delay of the pipeline register become more prominent [2]. This causes diminishing returns and a minimum in execution time can be found at a specific number of pipeline stages. If energy is also considered, determining the number of stages becomes even more daunting because power increases linearly with frequency (see Sec. 2.1.1) which in turn grows higher with the number of pipeline stages. Optimum pipeline depth is dependent on the architecture and the specific program being executed, therefore there is no way to determine a general optimal number of pipeline stages [2]. Historically, processor pipelines grew deep to exploit the available ILP in the running instruction stream. However, ILP is limited and exceedingly deep pipelines, called super pipelines, only yielded marginally more performance while significantly increasing the power dissipation. Approaching the power wall, a practical upper

limit on power dissipation due to discontinued Dennard scaling, and the advent of portable computing made energy efficiency an important design goal [33]. The design space became more complex as performance and energy efficiency on most occasions warrant different design choices.

2.3.2.1 Caching

As mentioned before, if the register-file were the only memory available to the processor, program complexity would be limited. However, as implied above, more memory is available and the ISA defines how the memory is interfaced with the processor. More available memory allows for more complex and useful programs to be created. However, this memory would need to remain fast even though its size is increased to not slow down the processor. Such memory, if it existed, would be very expensive. A better solution can be found by taking into account how a program is executed. Only a limited part of the program is executed at any given time and due to code constructs such as loops the same parts of the program are likely to be executed in the near future. The insights of spatial and temporal locality can be used to construct a memory hierarchy that delivers on both speed and capacity at less expense [1].

A conceptual view of a memory hierarchy is shown in Fig. 2.14. In the figure, access times of the structures and their size are shown. As can be seen, smaller memory is generally faster and kept closer to the processor. The register-file, as discussed, is a very small and fast structure integrated into the datapath. The cache is likewise integrated into the pipeline and is larger than the register-file and consequently slower, but it is still fast enough to be accessed without imposing intolerable performance losses [1]. Fig. 2.14 depicts the cache as several layers where the level-one (L1) cache is small and fast followed by a larger, but slower level-two (L2) cache. Modern designs stretches further with larger lower-level caches located off chip or possibly integrated onto the chip [1]. Succeeding the caches is a larger and slower (two orders of magnitude (OOM)) main memory. Lastly, the largest and slowest part of the memory hierarchy is disk memory. The memory hierarchy is a very complex system and describing it in its entirety is beyond the scope of this report. Instead, the report is limited to describing the highest part of the hierarchy, i.e., the caches.

In load-store architectures, such as MIPS I described in Sec. 2.3.1, the processor is only allowed to interact with the memory through dedicated load and store instructions [2]. Consequently, if the processor needs data that is not present in the register-file, a load instruction in the program instruction flow must bring it into the register-file. This load instruction is directed to the cache. If the data is found in the cache, a hit is generated and the data is sent to the processor. However, because caches are small, it is likely that the data is not present and a miss is generated. The memory access then continues searching in lower levels in the hierarchy until the data is found. To support overlapping instruction fetch and data access, the L1 cache is usually separated into separate caches, i.e., an level-one instruction cache

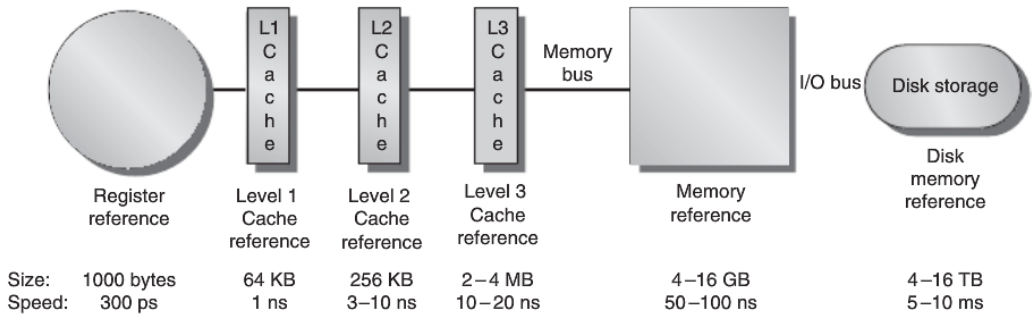

Figure 2.14: A conceptual overview of a memory-hierarchy [1]

(L1IC) and an level-one data cache (L1DC), and this approach is referred to as the Harvard cache model. Lower levels in the memory hierarchy are usually unified, containing both data and instructions, according to the Princeton cache model [34].

Caches are arrayed structures where each row, usually referred to as a cache line, is addressable using the address generated by a store or load instruction. [1] The simplest way to structure a cache is called direct-mapped, where each memory address maps to one specific line within the cache. As caches are small, memory addresses will overlap and map to the same location in the cache. Thus, the processor needs to be able to distinguish between the addresses. This is achieved by storing the higher-order address bits, called a tag, alongside the data and comparing these with the address used to access the cache [1]. The portion of the cache that stores the tags is referred to as tag-array and the part that stores the data the data-array. If the tag matches the address used to access the cache a hit is generated. Conversely, if the tag does not match the address a miss is generated and the cache line is replaced by the requested data brought in from lower levels in the memory hierarchy.

A cache that supports a more flexible address mapping is called n-set-associative where the n denotes the flexibility of the mapping [1]. Assuming a four way associative cache the cache is effectively split into four sets, each able to store data mapped to one cache address. At the extreme end of associativity is a full-associative cache where every address maps freely into the cache. Associative caches require more hardware because each set needs to be searched for the proper cache line and compare the tag values with the requested address [1]. Furthermore, when an address maps to a full set and generates a miss, a victim selection mechanism needs to be in place. There are several techniques available but the least recently used (LRU) or random selection schemes or variations thereof are usually enforced. Again, this adds to the hardware overhead of using a set-associative cache.

Direct-mapped and associative caches both have their advantages and disadvantages. Direct-mapped caches are simple in terms of hardware but generally suffer from lower hit rate than their associative counterparts [1]. In contrast, the higher hit rate of associative caches comes at an expense of more hardware and thus power dissipation overhead. Which scheme to use depends on the application [1].

Irrespective to what cache scheme that is used, store instructions pose problems when it comes to memory coherency [1]. This problem is present in all layers in the memory hierarchy, but is more acute in caches which are latency sensitive. A store operation will update the cache content in the L1 cache and unless this is reflected in lower levels, e.g., the L2 cache, the memory is said to be incoherent. Should the L1 cache line be evicted, the line cannot be restored and data is irreversibly lost. Thus memory coherency is required to ensure program correctness. The simplest solution is to let stores propagate down the memory hierarchy [1]. While simple, this solution increases the cache latency and ergo the execution time of the running application. Another less penalizing scheme is the write-back scheme, which only writes back the cache line when evicted to lower levels in the hierarchy. The write-back approach requires extra bookkeeping hardware, called a dirty bit, to indicate whether a write-back operation is necessary. Naturally variations and enhancements of these approaches exist but they will not be discussed.

Because misses are expensive, the performance of a memory hierarchy is to a large extent determined by the miss rate as shown in Eq. 2.7 [1].

$$\text{Cache access} = H_t + M_r * M_p \tag{2.7}$$

The hit time (H_t) is the time paid for a successful cache access, the miss rate (M_r) the fraction of misses to the total access count and lastly miss penalty (M_p) the cost to access lower levels in the hierarchy. This formula can easily be extended to include more layers in the hierarchy, as shown in Eq. 2.8 where a L2 cache is included.

$$\text{Cache access} = H_{tL1} + M_{rL1} * (H_{tL2} + M_{rL2} * M_{pL2}...) \tag{2.8}$$

Depending on the workload, an increased average access time can be very deteriorative to performance [1]. As such, great care must be taken when designing the memory hierarchy.

2.4 Existing pipeline evaluation method

As stated in Sec. 1.1 this work aims at implementing a methodology that has been used with success at Chalmers University of Technology. The methodology builds on two components, an architectural simulator and pipeline and cache RTL. This section will elaborate on these components, starting with the simulator in Sec. 2.4.1 followed by the RTL in Sec. 2.4.2 and lastly how they have previously been combined in Sec. 2.4.3.

2.4.1 Architectural simulator

SimpleScalar is an execution-driven functional simulator capturing both the behavior and performance of the simulated architecture [12]. The fact that it is execution-driven is essential as this captures the dynamic behavior caused by branches and cache misses of the underlying architecture, which can have a dramatic impact on performance and energy. Furthermore, because SimpleScalar captures both the functionality and performance of the architecture, correct program behavior is ensured and accurate resource usage and time measurements (execution time in clock cycles) are possible [12]. It can be argued that the simulator is too old and limited to single core designs in an age of multi-core processors. Other more modern tools, with equal or a super-set of SimpleScalar's features, such as McPAT or Gem5 are available but were turned down in favor for SimpleScalar because SimpleScalar was sufficient for the relatively simple 5SP design that it was used to model.

SimpleScalar provides several different simulators of varying detail and speed [12]. The simplest and fastest simulator, called sim-fast, is a purely functional simulator that does not account for time (cycles). In contrast, the most complex simulator, the sim-outorder, supports out-of-order issue, speculative execution, multiple issue while also accounting for time.

The structure of SimpleScalar is shown in Fig. 2.15. The bpred block defines the branch predictor behavior, the cache block defines cache behavior (cache size, associativity and replacement technique), the regs block defines register related behavior and the memory block the memory related behavior. The simulator core defines the datapath's architecture and it is by far the most substantial block.

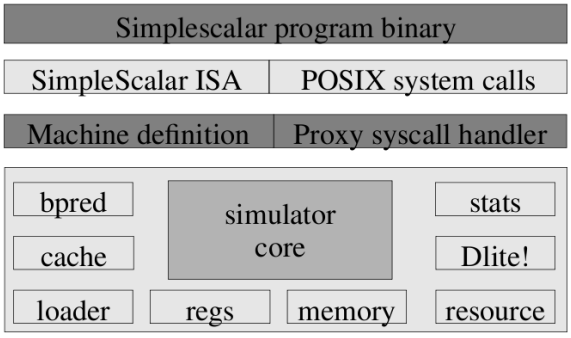

Figure 2.15: Modular structure of SimpleScalar

The simulators support configuration through configuration files that are provided to the simulator of choice when calling it from the command line [12]. The configuration files allow features such as branch resolution, cache parameters, speculative execution, decode width, issue width and number of functional units to be tweaked without the need to rebuild the simulator.

The simulator used in the methodology is based on a modified version of the sim-outorder simulator. The modifications were implemented to reduce the out-of-order

pipeline modeled by the simulator to an in-order pipeline similar to the RTL pipeline described in Sec. 2.4.2 below. The base simulator was then augmented with specialized performance counters that tracked usage of, for the project, relevant pipeline resources. Because of frequent use in various projects the base simulator had been modified, sometimes extensively, to fit the needs of each new project.

2.4.2 RTL design and verification

The RTL used in the methodology captures 5SP MIPS I pipeline design developed at Chalmers. The 5SP has been enhanced with one cycle access latency L1IC and L1DC caches also developed at Chalmers. This setup has since been used in several well-received publications [4][35].

The implemented microarchitecture features some 50 instructions including different branches, logic and memory instructions and a register-file with re general-purpose 32-bit registers. This microarchitecture does not include a floating-point unit to provide floating-point support, which was motivated by the targeted embedded market where floating-point operations usually are replaced by fixed-point calculations.

An overview of the microarchitecture is shown in Fig. 2.16 and it is similar to the pipelines discussed in Sec. 2.3.2. The instructions are processed in five stages; IF, ID, EX, MEM, and WB. In the IF stage, instructions are read from the instruction cache from an address pointed to by the PC register, which is updated to point to consecutive instructions or to branch target addresses. During the ID stage the register-file is accessed and control signals for later stages are set based on the instruction type. Branch and jump instructions are solved in the ID stage, but by the time they are resolved the next instruction has already been fetched. A delayed branch slot is utilized to solve this problem and is accounted for by the compiler (see Sec. 2.3.2). In the EX stage arithmetic and logic operations are executed in an arithmetic logic unit (ALU). A dedicated two-stage multiplication unit is also available, spanning the EX and MEM stages. In the memory access stage, loads and stores access the L1DC. Finally, in the WB stage, results are written back to the register-file. In the RTL code the MEM and WB stages were combined to simplify the implementation. However, the combined stage logically functions as two separate stages.

A hazard-detection unit, which physically resides in the ID stage but is shown separately in Fig. 2.16, detects any potential hazards and stops the pipeline by stalling the IF stage. In this manner, NOPs are inserted into the pipeline. The cache also produces a stall signal, which is asserted upon a cache miss. In contrast to the hazard stalls, cache misses stall the entire pipeline in Fig. 2.16 where the arrows pointing to the pipeline registers denote the stall signals. The microarchitecture does not support exceptions, but these are by design rare events. Exceptions are necessary to support I/O and recover from errors (Invalid Opcode etc.) and system calls.

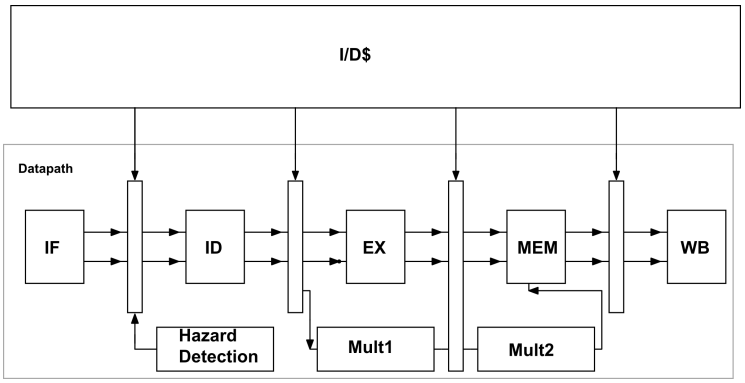

Figure 2.16: Microarchitectural overview of the 5SP.

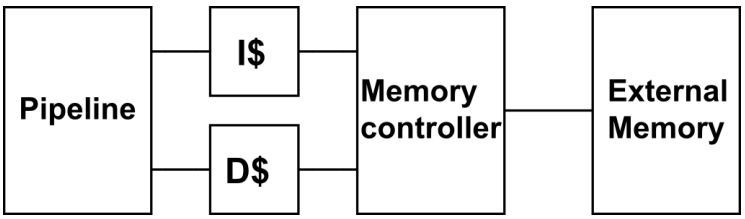

Figure 2.17: Memory hierarchy of the 5SP.

The design includes L1IC and L1DC caches. These caches are separate from each other according to the Harvard architecture to avoid structural hazards as explained in Sec. 2.3.2.1. No L2 cache is included in the design, instead an ideal memory module serves as a replacement for the lower levels of the memory hierarchy as shown in Fig. 2.17. The data cache is available for read and write accesses, while the instruction cache only serves reads. However, the instruction cache still needs to access external memory on cache fills and in the case of a cache miss. The two caches share one memory bus to the external memory and a memory controller (arbiter) orchestrates which one of the caches that is allowed to access the external memory. Both caches were designed to be flexible and allow for any size the user desires. However, because the RTL has been used in several projects, which sometimes required extensive modifications to the RTL code much of this flexibility was lost. Previously the associativity could be set to zero (effectively direct-mapped cache), two-way or four-way with replacement algorithms LRU or pseudo random. The cache also supported banking whereby cache lines are split across separate memory macros.

2.4.2.1 Design and verification flow

The existing evaluation method loosely defines a RTL design and verification flow which has been used to verify and extract power from the pipeline RTL. The RTL design was first brought through a functional verification as described in Sec. 2.2 followed by a cell-based synthesis and then PnR. From the place and routed netlist, power estimates of the design were obtained. As the power estimates are obtained from a complete pipeline design that has been synthesized and placed and routed to meet a set timing constraint, they capture the synergy between the different com-

ponents in the pipeline. The synergy is due to the fact that logic paths stretches over several components, i.e., the speed of one component imposes speed requirements on subsequent components. These logic paths are then adapted to meet the imposed timing constraint, which is achieved through transistor sizing. However, the evaluation method focused solely on the caches of the RTL design which allowed for probabilistic testing. The probabilistic approach was used to obtain power estimates of peripheral units of the cache such as DTLB, arbiter and replacement logic. The power of the actual static random access memory (SRAM) memory cuts was obtained in the library files used during synthesis. The existing evaluation methodology strikes a balance between quick prototyping and estimation accuracy but neglects scalability. Changes to the cache would require a complete reiteration of the design flow, with a lot of effort spent on PnR and power estimates.

2.4.3 Ad-hoc combination of RTL and simulator

The RTL and SimpleScalar components of the methodology are then combined in a way specific to each project. Performance counters were introduced for each project and power estimates were extracted from RTL structures that were represented by these performance counters. An example of a prior application is the STA (Speculative Tag Access) project where power estimates were extracted from the caches through probabilistic techniques and the simulator was augmented with performance counters that monitored cache access patterns [4]. Similar approaches were used in several other publications with the main exception being the introduction of additional RTL structures and different performance counters [35][36] [37]. However, the new RTL was not integrated into the pipeline but instead analyzed separately.

3

A unified evaluation framework

The goal of this work is to develop an energy estimation framework for pipelines that captures the ad-hoc methodology outlined in Sec. 2.4. As discussed the pipeline register-transfer level (RTL) code and SimpleScalar simulator constitute the two major components of the framework. The methodology was established previous to this work, albeit in an ad-hoc manner, so this section will instead elaborate on the methods that allow the two components to be integrated into one coherent framework.

3.1 Framework workflow

The conceptual workflow of the framework is shown in Fig 3.1 where the RTL and architecture simulator constitute the two branches of the flow. The RTL branch consists of an RTL verification flow similar to the one described in Sec. 2.4.2.1, which will be implemented to verify the RTL and later netlists. Slight alterations to this approach are necessary for it to be scalable and applicable to a range of designs. For instance the power estimation previously based on probabilistic methods is substituted for a use-case based method, which allows the whole design to be studied on a pipeline unit basis. Whether the power estimates are averaged or time-based is similarly a matter of scalability. Time-based analysis produces considerably more data than averaged but allows for detailed analysis of the power dissipation. In contrast averaged analysis is easier to integrate into a scalable framework since less data are produced. However, as the power is averaged over a unit of time, the power dissipation for units with low utilization is amortized over the estimation interval. The issue can be addressed by introducing resource counters in the testbenches used in the verification flow and scaling the final average power according to the counters as shown in Eq. 3.1.

$$P_{scaled} = \frac{P_{unscaled}}{Utilization} \tag{3.1}$$

where *Utilization* is the total percentage of the total execution time, in either seconds or cycles, when the unit is used. Furthermore, doing the power estimates after

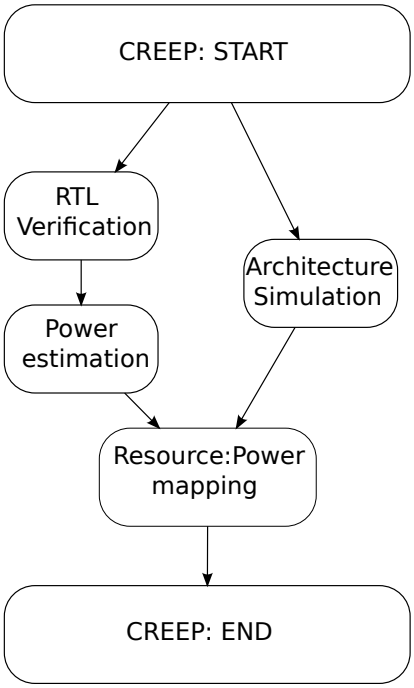

Figure 3.1: The methodology embodied in the CREEP framework.

the place and route (PnR) creates an issue of scalability as all designs are different and would need to be placed and routed manually. Instead, one design will be placed and routed and serve as an indication for how much the power increases post PnR.

The simulator branch consists of SimpleScalar as described in Sec. 2.4.1. The simulator is used to acquire accurate per cycle resource usage using workloads that are impractical (impossible) to use during RTL power estimation due to their complexity and size. These resource usage statistics are then combined with the power estimates obtained from the RTL.

There are two issues that arise in the combination of the power estimates and simulator statistics. Firstly, the mismatch between resource counter and RTL power estimates must be bridged and secondly, the pipeline power must be mapped to resource counters in a manner that does not systematically over or underestimates the energy. The first issue stems from the power reports, e.g., unit conversions, grouping into different sources of power dissipation, i.e., leakage and switching power and whether the power reports are time-based or averaged. Unit conversions are trivial and will be done to obtain energy per cycle in order to combine the power estimates with resource counters. The power grouping should likewise pose no issues but will require some consideration of scalability as dealing with different types of power dissipation sources will complicate the framework workflow. The second issue is brought about by the different structure and granularity of the architectural simulator and the RTL code. The resource mapping will be done by carefully grouping the RTL pipeline units and introducing selective performance counters in the architectural simulator where it is necessary. The granularity of the resource mapping will be done incrementally from coarse to finer.

The combination of said energy estimates and performance counters will be automated in order for the framework to be able to reproduce results consistently, which is stated as a goal in Sec. 1.1. Furthermore, to meet the configurability goal the user shall be able to configure the RTL and simulator components. Settings related to the cache and possibly to the pipeline will be exposed to the user, but the user will not make any changes to the components themselves. Instead, this process will also be automated thus ensuring that the components are combined consistently. Inconsistencies between the two components would needlessly affect the energy estimates negatively.

3.2 Verification

Verification of the framework will be done at the component level. For the RTL the functional verification is inherent to the established RTL verification and power estimation flow. However, the power estimates themselves need to be verified. As the RTL has been used in previous projects, estimates for parts of the design, more specifically, the caches are available. These estimates will be used to do a rudimentary evaluation of the power estimates.

For the simulator the modifications as well as the added resource counters need to be verified. The modifications can be verified by comparing the default performance counters to the counters produced by the modified simulator. Several counters should indicate in-order, non-speculative and single-issue behavior. The added resource counters can similarly be validated by comparing them to default counters produced by SimpleScalar.

4

Implementation

This chapter will outline the implementation of Chalmers RTL-based energy evaluation framework for pipelines (CREEP), starting with the implementation of the register-transfer level (RTL) flow in Sec. 4.1.1 followed by the simulator in Sec. 4.1.2. How these two are integrated into the framework workflow is then discussed in Sec. 4.2. Lastly, the work related to automating the framework will be outlined in Sec. 4.3.

4.1 Implementation of framework components

The RTL of the framework describes a 5-stage pipeline (5SP) that supports the integer subset of the 32-bit MIPS I instruction set architecture (ISA) (see Sec. 2.4.2). However, in order for the RTL to fulfill the framework's needs several modifications of it was necessary. Furthermore, an integrated circuit (IC) design and verification flow with emphasis on scalability was implemented in order to obtain power estimates from a wide range of designs. The SimpleScalar simulator was similarly modified to fit into the framework. For clarity this section is split up into three parts: The first part, Sec. 4.1.1, discusses the RTL modifications and establishment of the design and verification flow. The second part, Sec. 4.1.2, elaborates on the simulator modifications. The last part, Sec 4.1.3, deals with the framework's configurability.

4.1.1 RTL modifications

As previously mentioned in Sec. 2.4.2 the caches were originally designed to have adjustable dimensions, associativity and replacement techniques. However, the pipeline design was used in other projects prior to the framework development during which the level-one instruction cache (L1IC) and level-one data cache (L1DC) were fixed to a configuration with 16kB 4-way associativity and LRU as replacement algorithm. To meet the goal of configurability stated in Sec. 1.1, it was decided that the caches should be restored to their original flexible condition.

The main limitation of the cache components in their original initial condition was set by the use of 1024x32-bit static random access memory (SRAM) memories for the

data-arrays and 128x32-bit SRAM memories for the tag-arrays. This corresponded to 128 sets with a line size of 8 instructions for the L1IC or 8 words for the L1DC. Furthermore, the setup was locked to a 4-way configuration producing a cache of 16kB in total ($4 \times 1024 \times 32$). The tag arrays were optimized to fit four 21-bit tags (tag + dirty bit) into three SRAM memories rather than the conventional four memories (one per way). The rationale behind this was that three tag arrays were sufficient to hold four tags ($21 \times 4 = 84 < 96$). The code was modified to instead map the tags into four tag-arrays. This change allowed the associativity to be set within the original bounds of direct-mapped to 4-way associative. However, this caused an increase in the bit overhead in the tag-arrays as only 21 of 32 bits were used. This was deemed unavoidable since SRAM memory matching the tag width of 21-bits was unavoidable.

Additional SRAM memories of different sizes were introduced to allow for additional cache sizes of 8kB and 32kB. Furthermore, banking was reintroduced which can be used to divide the cache lines between several SRAM memories. In addition to the LRU replacement algorithm, the pseudo-random replacement algorithm was also reintroduced.

Because of licensing issues no SRAM memories can be shipped with the framework. To make the RTL available without SRAMs the caches were augmented with the ability to use logic-based memory (flip-flops). Compared to SRAM logic-based memory produces larger (area) and more power dissipating caches.

Other ways of allowing more flexibility in the pipeline design were considered, but the remaining options were related to the datapath. Changes to the datapath, such as allowing wider-issue width, speculative execution, different branch resolution techniques would all essentially warrant a complete redesign of all or some pipeline stages. Furthermore, allowing such flexibility was outside the scope of the framework, which targets simpler embedded processors.

4.1.1.1 Design verification flow

A verification flow built on the methods described in Sec. 2.4.2.1 was established. Cadence IES was the electronic design automation (EDA) tool of choice for hardware description language (HDL) simulations. More specifically NCVHDL was used to compile the RTL code, NCELAB was used to elaborate the design and NCSIM was used for simulations. To verify the design, a testbench was constructed around the pipeline design. As the design is complex test vectors were chosen as stimuli for the design. More specifically, the vectors were directed to test the design's implementation of the MIPS I ISA through instruction set simulations using executables compiled for the MIPS I ISA. However, RTL simulations of large designs are time consuming but can be facilitated through the use of small and effective workloads with ample test coverage. A good match was found in the EEMBC benchmark suite, which is a benchmark suite that targets embedded processors. The benchmarks in the suite are light weight and utilizes fixed point arithmetics. The benchmarks used

in the framework are listed below:

- Autocorrelation
- Convolutional Encoder
- FFT/IFFT
- Viterbi Decoder
- RGBCMY01 (Consumer RGB to CMYK)

The next stage in the verification flow is synthesis. The EDA tool of choice for synthesis was Synopsys Design Compiler (DC), which was chosen because it is the de-facto standard tool used by the community. The synthesis was done using the compile-ultra command and cells from the 65nm low power (LP) low threshold worst-case corner library (1.1V 125°) provided by ST Microelectronics. A corresponding library was used for the SRAM memories in the caches. These libraries represent the worst-case process corner and use scenario (extreme temperature and low voltage) and were used because of the strict performance requirements placed on ICs. Additionally, automatic clock-gating was enabled in an effort to reduce the design's dynamic power dissipation [38]. Clock-gating is widely used in the industry because of its potentially large energy savings at little extra design effort. Most EDAs specializing in synthesis support it, DC included. The synthesis was carried out for increasingly strict timing constraints to find the maximum achievable clock frequency of the design and the design was established to meet a timing constraint of 2.5ns, producing a netlist running at 400 Mhz. The netlist verification was done using the same testbench developed for RTL verification.

The final stage is place and route (PnR) for which Cadence Encounter was used. As described in Sec. 2.4.2.1 PnR produces yet another netlist, but this netlist has now been subjected to a number of structural changes to facilitate physical implementation. PnR was necessary to include in the verification flow because of two reasons. Firstly, the utilized SRAM memories are already placed and routed and thus dissipates significantly more power than the rest of the design unless this is also placed and routed. Secondly, a placed and routed design allows for more accurate power estimations than a post-synthesis design (see Sec. 2.2). However, the PnR stage is unique to each design, which conflicted with the desired scalability of the verification flow. A more scalable approach was implemented that estimated the PnR impact on power dissipation by comparing the power of one placed and routed netlist to a post-synthesis netlist and from this comparison a scaling factor could be deduced. The rationale behind this approach was that the pipeline was not subjected to any modifications (see Sec. 4.1.1) and remains relatively unaffected by the configuration of the caches.

4.1.1.2 Design power estimations

It was not possible to use the power-estimation method used in the ad-hoc methodology in order to obtain power estimates of the design (see Sec. 2.4.2.1). The main reason was the larger scope of the power estimates, which previously were limited to the L1DC, but now included the entire pipeline. As such, using a probabilistic approach was unfeasible. Instead, the power dissipation of the design was estimated by using use-case statistical simulations whereby switching activities for the nodes in the design were obtained. Two different statistical methods were considered. The first considered method was switching activity interchange format (SAIF) generation. During SAIF generation the average switching activities of the nodes in the design are recorded throughout simulation, which allows an average power estimate of the design to be produced. The second was value change dump (VCD) generation which tracks the nodes' switching on a per-cycle basis which allows for time-based power analysis. The VCD based method allows for detailed analysis of the power dissipation, e.g., maximum power dissipation analysis, but the usage of VCD is computationally complex and hence less scalable than the SAIF-based method. Thus, VCD generation was dropped in favor for SAIF generation. Cadence NCSIM was used to simulate the netlist using the aforementioned RTL testbench and the previously listed EEMBC benchmarks as stimuli. A total of five of five SAIF-generations were done (one per EEMBC benchmark).

Synopsys PrimeTime (PT) was the EDA tool used to generate the final power estimates [39]. PT was first used to remap the gate netlist to a different cell library from the one used during synthesis. In contrast to the synthesis, which was done with the worst-case high-temperature corner library, the nominal-nominal variation (1.2V NOM 25°) was used to generate the power reports. The reason for using the nominal corner and nominal voltage and temperature library was to provide nominal power estimates for the pipeline design and thus allow different pipeline configurations to be compared under normal circumstances. The power estimation was done by reading the netlist and each of the aforementioned SAIF files. Thus, a total of five power reports were produced and averaged to create the final design power estimate. Hierarchal reports were produced for the design and the granularity of these reports was tweaked to reveal major pipeline units within each pipeline stage. However, the granularity of these reports was later tweaked to better suit the performance counters generated by the simulator component (Sec. 4.1.2). PT reported the power divided into three different categories: 1) switching power, 2) internal power and 3) leakage power. To simplify the workflow the sum of all these powers was used for each reported pipeline unit.

As discussed in Sec. 3.1, average power reports amortize the power of certain units over the power estimation interval. Units such as the arithmetic logic unit (ALU), multiplier and L1DC are associated with enable signals that prompts them to activate, i.e., start switching and dissipating power. Unless the power of these units are scaled according to their usage, the framework would greatly underestimate their contribution to the final energy results. The solution to this problem, which was dis-

cussed in Sec. 3.1, required information of how many cycles the affected units were used during the power estimation interval. The usage information was obtained by augmenting the RTL testbench used during verification and power estimation with counters that were incremented when the enable signal for these structures was asserted. These counters were then divided by the total number of cycles also tracked by the testbench. The scaling factor was then computed as shown in Eq. 4.1.

$$Utilization = \frac{active\_cycles}{total\_cycles} \tag{4.1}$$

The power dissipation reported by PT was then divided by this utilization factor, as shown in Eq. 4.2 to obtain the final power values used in the framework.

$$P_{scaled} = \frac{P_{unscaled}}{Utilization} \tag{4.2}$$

As discussed briefly in Sec. 3.1 and Sec. 4.1.1.1 a scalable approach to power estimation of a placed and routed design was necessary for the scalability goal as stated in Sec. 1.1. An attempt was made at running the post-PnR netlist through the implemented verification flow but this was met with technical issues that proved hard to solve. Instead a probabilistic approach was adopted, which limited the scope at which the design could be analyzed. Since the SRAM memories comes placed and routed, the PnR scaling should only be applied to combinatorial pipeline units. Hence, the ALU was chosen as a representative combinatorial unit and switching activities were assigned to the ALU input. The power dissipation was then extracted from a synthesized and a post-PnR netlist. Then the PnR factor was derived from the fraction of the PnR power to the power based on the synthesized design as shown in Eq. 4.3. The PnR-scaling factor was then applied to all combinatorial units in the pipeline.

$$PnR_{scaling} = \frac{ALU_{synth}}{ALU_{PnR}} \tag{4.3}$$

A similar estimation was done for the clock-tree power, which is small in a synthesised netlist. The limited clock power dissipation accounted for in a post-synthesis design is related to the clock pins on registers in the design. Hence, the majority of the difference in clock power dissipation between a synthesized netlist and a post-PnR netlist is due to the added clock-tree.

4.1.2 SimpleScalar

There was no reason to replace SimpleScalar as the architectural simulator used in CREEP. As discussed in Sec 2.4.1 SimpleScalar consists of many different simulators, each with a different level of abstraction, but the only simulator that allowed for accurate resource tracking was the sim-outorder simulator. A modified version of this simulator was used previously in the ad-hoc methodology. However, this version of the simulator had repeatedly been modified to include project-specific functionality, which inhibited its direct use in the framework. It was decided that it was a better to start with an unmodified version of the simulator rather than to spend time on restoring the ad-hoc simulator. Consequently, a default SimpleScalar version was obtained and modified to allow it to be used used in the framework.

The simulator modifications were aimed at reducing the modeled pipeline to a state that closely models the provided RTL pipeline. Hence, the out-of-order execution capabilities were removed reducing the simulated pipeline to a strictly in-order 5SP. This was achieved through source code modifications. These modifications included moving store operations from the commit stage to the issue stage (for example loads) whereby the stores were locked to non-speculative in-order execution. Other features such as multi-issue, speculative execution (for instructions besides stores) and branch prediction were configured through configuration files. Because the RTL code does not support these microarchitectural features they fall outside the features supported by the framework. Thus, the configuration file was changed to limit the issue width to one and disable speculative execution. However, an exception was made for the issue bandwidth which was set to two. This was necessary as load and store instructions are split up into a separate address calculation and read/write instruction. This will not cause any issues for the other instructions as only one instruction at a time leaves the preceding dispatch stage. The branch prediction was configured to perfect (always correct), which resembles the delayed branch slot technique used in the RTL pipeline. A common ground between the simulator and RTL was found in the caches. The SimpleScalar caches are configured through the aforementioned configuration file and support a superset of the settings available in the RTL code. The caches were configured to mirror the caches in the RTL code. All relevant settings for the framework are summarized in Table 4.1.

One major difference between the provided 5SP and the corresponding SimpleScalar implementation, that cannot be solved through configuration files nor reasonable source code modifications, was that the latter supported a larger set of instructions, e.g., floating point instructions and system-calls. The impact of this issue was limited by choosing simulator benchmarks that included few of the unsupported instructions and features. Furthermore, the unsupported instructions could be ignored when introducing the resource counters and thus not allow them to affect the framework result.

Table 4.1: Summary of relevant SimpleScalar configurable settings

Setting	Description	Value
-fetch:ifqsize	Instruction fetch que size, set to 1 to model a single-issue pipeline.	1
-decode:width	Decode width, set to 1 to model single-issue pipeline	1
-issue:width	Issue width, set to 2 to model single-issue and allow loads/stores to be issued in one cycle.	2
-commit:width	Commit width, set to 1 to modle a single-issue processor.	1
-issue:inorder	Pipeline is set to issue in-order	true
-issue:wrongpath	Pipeline is set to issue non-speculatively	false
-bpred	Branch predictor component set to perfect to emulate delayed branch slot	perfect
-cache:il1 / -cache:dl1	L1 Cache settings, to be coordinated with RTL	-RTL
-cache:il1lat / -cache:dl1lat	L1 Cache access latency	1
-cache:il2lat /-cache:dl2lat	L2 Cache access latency	12

Several workloads were considered for the framework. One candidate was provided by SPEC, a non-profit organization with the philosophy to ensure that the marketplace has a fair and useful set of metrics to differentiate systems. SPEC provides several benchmark suites for different applications [40]. One of the suites, SPEC CPU2006, is specifically designed for microprocessors. SPEC CPU2006 is in turn composed of two different benchmark suites; CFP2006 and CINT2006. Sadly, SPEC is not open source which effectively inhibits it from being included in the framework. An other candidate was MiBench, which in contrast to CPU2006, is open source and fills roughly the same niche as SPEC but is leaning more towards embedded processors. EEMBC consists of a set of 35 embedded applications which are divided into six suites targeting a specific area in the embedded market [41]. MiBench was chosen as the simulator workload for the framework mainly because it was open source that would allow it to be shipped with the framework. However, not all 35 benchmarks in the suite are used because of incompatibility with the simulator. Instead, a subset of 20 benchmarks were selected from the automotive, consumer, network, office and security categories. The categories and pertaining benchmarks are shown in Table 4.2.

Table 4.2: MiBench benchmarks

Category	**Applications**
Automotive	Basicmath, Bitcount, Qsort, Susan
Consumer	JPEG, Lame, TIFF
Network	Dijkstra, Patricia
Office	Ispell, Rsynth, Stringsearch
Security	Blowfish, Rijndael, SHA, PGP
Telecomm	ADPCM, CRC32, FFT, GSM

4.1.3 Configurability

One of the goals with the framework was to allow for limited configurability (see Sec. 1.1). However, this configurability must be found in both the RTL and simulator as the simulator captures the behavior of the RTL and vice versa. During the implementation it became clear that the RTL placed the most limitations on configurability. Essentially only the caches in the RTL supported configurability,

the pipeline was not to be configured as discussed in Sec. 4.1.1. Conversely, the simulator supported extensive configuration of both datapath and caches through the aforementioned configuration files. Thus, the configuration of the framework was limited to the L1IC and L1DC. The RTL pipeline does not include a level-two (L2) cache, hence the framework cannot estimate the power dissipation of this cache. However, SimpleScalar could still be used to simulate the L2 cache's impact on performance and indirect impact on the energy estimates. Thus, settings related to the L2 cache were also included amongst the configurable settings. The supported settings are presented below:

- L1 Data cache settings:
  - Associativity: 1-4
  - Replacement policy: LRU/Pseudo random
  - Number of sets: 64, 128, 256, 512
  - Line size (words): 8
  - Bank size (words): 4, 8
- L2 Data cache latency: 12
- L1 Instruction cache settings:
  - Associativity: 1-4
  - Replacement policy: LRU/Pseudo random
  - Number of sets: 64, 128, 256, 512
  - Line size (instructions): 8
  - Bank size (instructions): 4, 8
- L2 Instruction cache latency: 12

These settings were deemed sufficient for the framework as it targets embedded processors which usually have small capacity caches.

4.2 Combining RTL and SimpleScalar

With the core components of the framework implemented they needed to be combined to form one complete workflow. As discussed in Ch. 3 this creates unit mismatches and introduces the issue of resource mapping. These problems are the topic of this section.

It was noted that there was a mismatch between the power reports produced by PT and the usage scenarios captured by SimpleScalar. PT reported the power in

the SI unit for power, i.e., J/s rather than Joules per cycle (J/c), which is required for the power estimates to be combined with the resource counters reported by SimpleScalar. A conversion from J/s to J/c was necessary, which was trivial as the clock frequency was set during synthesis. The conversion is shown in Eq. 4.4 where J is Joules, c refers to clock-cycle, s seconds and f the clock frequency.

$$J/c = \frac{J/s}{f} \tag{4.4}$$

The most pressing issue when combining the RTL pipeline design with SimpleScalar was combining the power obtained from the pipeline with the simulator. In both components, matching resources were to be identified. Moreover, the scale at which this would be done would have a great effect on the final energy estimates outputted by the framework. The goal was set optimistically with the aim of achieving estimates close to the PnR level as stated in Sec. 1.1.

The main limitation on how fine-grained the resource mapping could be done was that the sim-outorder simulator had a high level of abstraction capturing the discrete pipeline stages and major events coupled to each of these such as ALU, multiplier and cache usage. In contrast, the RTL power reports can be generated down to the transistor level. It was decided that a simple approach was preferred and if necessary the matching was to be refined. The initial resource mapping was done on a pipeline stage basis. However, exceptions were made for the ALU and multiplier that were mapped directly to their counterpart in the RTL code. The clock network power was combined with a counter that tracked the total number of simulated processor cycles.

It was quickly discovered that this initial resource mapping was too coarse grained as some pipeline stages, such as MEM/WB (the combined memory access (MEM) and write-back (WB) stage) and execute (EX) were unrealistically penalized. In the MEM/WB case the resource counter was increased every cycle while the load store unit (LSU) contained in the stage was only used for load and store instructions. For the EX stage the ALU was used for address calculation related to loads and stores whereas in the RTL the address generation unit (AGU), a smaller and less power dissipating unit, was used for this purpose. These issues were rectified by excluding these units power from the pipeline stage's power and introducing specialized counters which were selectively increased whenever a load/store instruction were encountered. To this end, two new counters, one for the LSU in the MEM/WB stage AGU in the EX stage were added. Another issue with the initial resource mapping was that important information used during design space exploration was drowned by other resources. Examples of this were the hazard detection logic, branch logic and decode logic which were overshadowed in terms of power by the register-file also located the instruction decode (ID) stage. The solution was to allocate separate resource counters for these units and combine them with corresponding power estimates from the RTL code.

The cache resource counters were also a cause for concern. The caches are by far the largest and most power consuming structures in the design. As such the energy related to cache accesses was substantially larger than for the other pipeline units. This was especially the case for the L1IC which dominated the initial energy results of the framework. Instruction caches have historically been very energy consuming and were the focus of many energy saving techniques. Most modern instruction caches incorporate such techniques. As such, it was decided to include one energy optimizing technique called way-prediction [42] in order to bring the L1IC energy in line with more contemporary designs. Way-prediction was emulated in SimpleScalar, which was achieved by introducing code which detected whenever a successful way-prediction was possible and a counter was incremented whenever this occurred and energy could be saved.

4.3 Framework automation

One of the goals with the framework was for it to be approachable. This was achieved by automating the framework and present the configurable parameters, as well as a few options on how to run the framework workflow, to the user.

The framework RTL and simulator components create an intimidating amount of output data that need to be combined as discussed in Sec. 4.2. Leaving this to the user would only reduce the usability of the framework and deter users from using it. The issue was addressed by creating a script that when invoked runs the entire workflow. The scripting language chosen to implement the script was Perl because it excels at text handling and file I/O which was the major requirements for the script.

To ensure that the architectural parameters are the same for the RTL and simulator all parameters, save the configurable settings, were hidden from the user. In additional to the configurations listed in Sec. 4.1.3, a setting used to enable or disable the emulated way-prediction in the L1IC was added (see Sec. 4.2). Configuration files, called CREEP-configurations, were created which lists the configurable options. As users would want to create several configuration files the main script was designed to use one such configuration as argument when calling the script as shown below:

```
# ./CREEP.pl -[CREEP_configuration]
```

The parameters specified in the configuration file would be parsed by the main Perl script, applied to the RTL code, the simulator configuration and all auxiliary scripts surrounding these components such as RTL testbenches, DC synthesis script, SAIF-generation scripts and PT-scripts. The main script would then start both components and use the outputs, power estimates from the RTL and resource usage from the simulator.

The RTL output would then be scaled by the main script according to the scaling discussed in Sec. 4.1.1.2 and then combine the partial results according to the resource mapping discussed in Sec. 4.2 to produce the final energy estimates.

The script was designed to allow the user three options; 1) run the entire framework, i.e., RTL and simulator components, 2) run the RTL component alone and 3) run the simulator separately. The options were designed to be specified when invoking the script from a terminal as such:

```
# ./CREEP.pl -[flag] -[CREEP_configuration]
```

Allowing separate components to be used would allow users that, because of licensing issues, are unable to use the RTL component to still use the framework. An issue here is that simulator statistics alone are not enough to produce the final energy results. This issue was addressed by saving the partial results generated from the RTL and simulation components for later use. The main script would then be able to load these partial results when running one of the components. It was decided to name the partial results after the provided CREEP-configuration file to allow the aforementioned command invocation to remain unaltered and the interaction with the main script simple.

5

Results and discussion

In this chapter, the finalized Chalmers RTL-based energy evaluation framework for pipelines (CREEP) is presented and discussed. A user-centric overview of the framework is presented in Sec. 5.1, which shows the final framework as most users will come into contact with it. The framework is then demonstrated in Sec. 5.2 by showing the framework results of a selection of configurations supported by the framework. The verification of the framework is then presented and discussed in Sec. 5.3 and lastly future improvements to the framework are discussed in Sec. 5.4.

5.1 User-centric overview

Fig. 5.1 provides an overview of the final framework package. The benchmark folder contains the EEMBC benchmarks used for register-transfer level (RTL) verification and switching activity interchange format (SAIF)-generation. The RTL folder contains all hardware description language (HDL) files for the pipeline, the caches and multiplier. The documentation folder contains the documentation of the framework. The scripts folder contains all script files used in the framework's workflow, e.g., synthesis and power estimation scripts. More importantly this folder contains the main perl script, CREEP.pl, which orchestrates the entire framework. The sim folder is used to store simulation-related files generated during RTL verification and the SAIF-files created during SAIF-generation. The SimpleScalar folder contains the SimpleScalar framework and the modified simulator source code used in the framework. The MiBench folder contains the provided open-source benchmark suite MiBench which is used as stimuli to generate the final performance and energy values. The synthesis folder contains the files produced during synthesis, most notably the design netlist. The configurations folder contains CREEP-configuration files, either shipped with the framework or user created. The configurations_sim and energy_configurations folders contain the partial results generated by the simulator and RTL component respectively. Lastly, the final results outputted by the framework are found in the results folder.

The finalized workflow is shown in Fig. 5.2. The workflow starts with the user supplying an option flag and a CREEP-configuration file as shown in Sec. 4.3. The main script, named CREEP.pl, applies the configuration to all relevant files in the

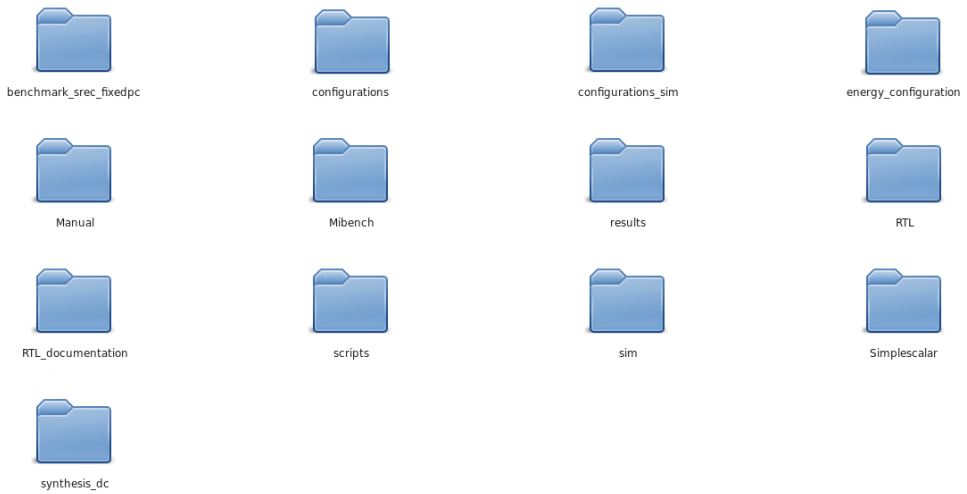

Figure 5.1: CREEP package overview

framework and starts the components designated by the flag. The RTL and simulator components generate data, i.e., power scaling information, pipeline power dissipation estimates and resource counters, which are combined by the CREEP.pl script according to the power scaling and resource mapping discussed in Sec. 4.2.

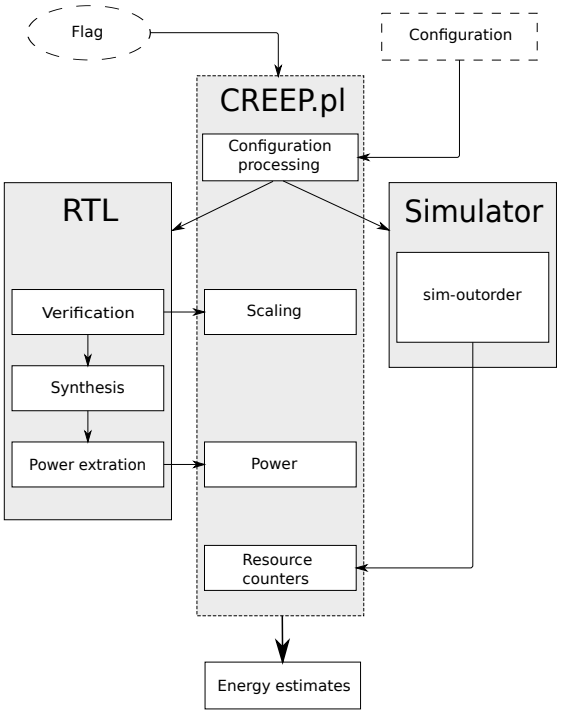

Figure 5.2: The workflow of the framework showing the RTL and simulator components and the central CREEP.pl script.

Due to licensing issues, the RTL component is shipped without EDA tool binaries and cell libraries, i.e., no logic nor static random access memory (SRAM) libraries are included in the framework package. However, as discussed in Sec. 4.3, a selection of configurations and pertaining power estimates are supplied with the framework to allow the simulator component to function separately and produce energy estimates of the provided configurations.

5.2 Demonstration

A selection of CREEP configurations were used to demonstrate the framework. The number of configurations had to be limited due to the large design space supported by the framework (see Sec. 4.1.3). The configurations are shown in Table 5.1 and they were selected on the basis that they span the design space supported by the framework, i.e., cache sizes and associativity. Unless specified, all configurations use a 12 cycle access latency to the level-two (L2) cache.

The 8kB direct-mapped configuration (8kB 1-1) is representative of a light-weight embedded processor. The two and four-way associative configurations, 8kB 2-2 and 8kB 4-4, were chosen to evaluate the impact of increased associativity on performance and energy. Furthermore, it is common to have lower associativity in the level-one instruction cache (L1IC) than in the level-one data cache (L1DC) due to simpler access patterns in the former. Configuration with low associative L1IC would ideally reduce power dissipation, while sacrificing a small degree of performance compared to a configuration with high associativity in both level-one (L1) caches. To evaluate lower associativity in the L1IC the 8kB 1-4 configuration was chosen. Two variants of the aforementioned 8kB 1-1, 8kB 2-2 and 8kB 4-4 configurations with with variations in L2 access latency were derived in order to evaluate the impact of the L2 cache on execution time (and thus energy). These configurations are named 8kB 1-1 10, 8kB 1-1 14, 8kB 2-2 10, 8kB 2-2 14, 8kB 4-4 10 and 8kB 4-4 14. The framework does not include any energy estimates of the L2 cache and lower levels of the memory hierarchy. Thus the primary impact on power of the lower levels, such as the L2 cache, is neglected (see Sec. 1.2). However, secondary effects on energy are expected due to the changes in the execution time.

The 16kB 4-4 configuration represents a performance-oriented embedded processor and it was the mainstay configuration used in the ad-hoc methodology prior to the framework development as described in Sec. 2.4.2. From the 16kB 4-4 configuration two configurations with lower associativity are derived, 16kB 2-4 and 16kB 2-2, to further evaluate the impact of associativity as cache capacity increases.

The 32kB configuration represents a no compromise performance-oriented embedded processor and uses the upper bound on the supported cache size in the framework. All configurations were synthesized to meet a 2.5ns cycle time.

Table 5.1: Cache parameters for the selected configurations

Name	Size	Associativity IC	Associativity DC	Replacement policy	Sets IC	Sets DC	Line size	Bank size
8kB 1-1 10	8kB	1	1	-	256	256	8	8
8kB 1-1	8kB	1	1	-	256	256	8	8
8kB 1-1 14	8kB	1	1	-	256	256	8	8
8kB 2-2 10	8kB	2	2	LRU	128	128	8	8
8kB 2-2	8kB	2	2	LRU	128	128	8	8
8kB 2-2 14	8kB	2	2	LRU	128	128	8	8
8kB 4-4 10	8kB	4	4	LRU	64	64	8	8
8kB 4-4	8kB	4	4	LRU	64	64	8	8
8kB 4-4 14	8kB	4	4	LRU	64	64	8	8
8kB 1-4	8kB	1	4	LRU	256	64	8	8
16kB 4-4	16kB	4	4	LRU	128	128	8	8
16kB 2-4	16kB	2	4	LRU	256	128	8	8
16kB 2-2	16kB	2	2	LRU	256	256	8	8
32kB	32kB	4	4	LRU	256	256	8	8

The aforementioned configurations will be analyzed in terms of performance and power in Sec. 5.2.2 and then the distribution of energy in the designs are presented in Sec. 5.2.3. Before this however, the execution time of the MiBench suite is discussed in Sec. 5.2.1.

5.2.1 MiBench execution time

The MiBench execution time of the 16kB 4-4 configuration is shown in Fig. 5.3. Note that this is not simulation time, but the execution time based on simulated processor cycles multiplied with the cycle time: $sim\_cycles \times T_{cycle}$. As can be seen there are a few deviating benchmarks, the most noticeable are basicmath, rsynth and stringsearch. As the name implies basicmath is mainly composed of arithmetic operations and is by far the largest benchmark with 6,360,380,890 simulated instructions. In contrast, rsynth and stringsearch are included in the office category (see Sec. 4.1.2) and are composed of text processing computations. Both rsynth and stringsearch are small compared to basicmath with 113,849,0 and 465,678,2 simulated instructions respectively. Moreover, as the simulator component captures dynamic events in the pipeline, such as cache misses and hazards stalls, that degrade the processor's instruction per cycle (IPC) (see Sec. 2.3.2) the number of simulated cycles are greater than the simulated instructions. For basicmath the 16kB 4-4 configuration manages an IPC of 0.7 which offers further insight into the execution time of the benchmark. Fig. 5.3 also shows the average execution time that lands around 2,887 seconds.

5.2.2 Power and performance

The average execution time versus the average power dissipation for all aforementioned configurations is shown in the scatter plot in Fig. 5.5. Fig. 5.4 shows the average miss rates of the L1 caches. The execution time, power dissipation and miss rates are based on the entire MiBench suite. The 8kB configurations have the lowest performance with average execution times ranging from 3.24 to 3.46 seconds. The small cache capacity causes high cache miss rates, which can be seen in Fig. 5.4 where the 8kB 1-1 configuration shows the highest miss rates. Each miss is associated with cycle penalties of accessing the L2 caches as described in Sec. 2.3.2.1. The impact

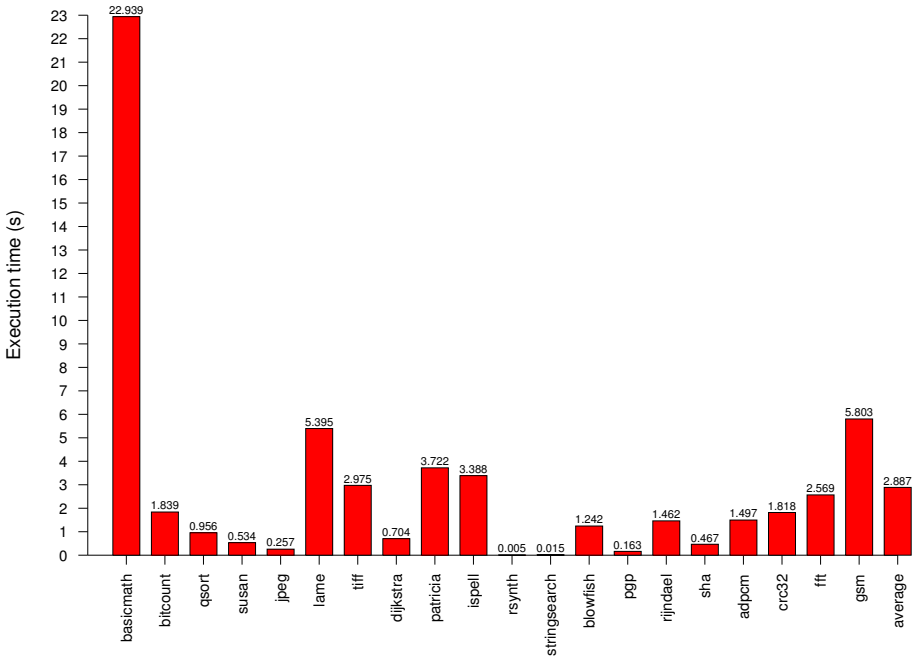

Figure 5.3: Per-benchmark execution time for standard 16kB 4-4 configuration.

of the L2 access latency is clearly visible between the 8kB 1-1 10, 8kB 1-1 and 8kB 1-1 14 configurations which are evenly spaced on the performance axis in Fig. 5.5. The 8kB 1-1 10 configuration is the fastest due to the lower L2 latency and the 8kB 1-1 14 is the slowest due to the higher L2 latency. The 8kB 1-1 configuration has the standard 12 cycle latency and falls between the aforementioned configurations. The minor power dissipation differences between the 8kB 1-1 variants are likewise explained by the difference in L2 latency. The faster 8kB 1-1 10 configuration has a higher power density than the slower 8kB 1-1 configuration and slower still 8kB 1-1 14 configuration.

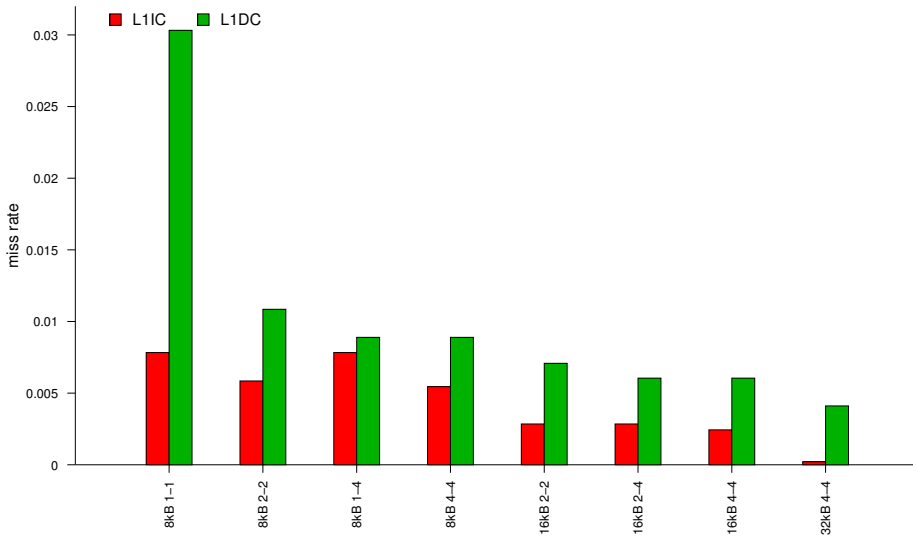

Figure 5.4: Miss rates of the different configurations.

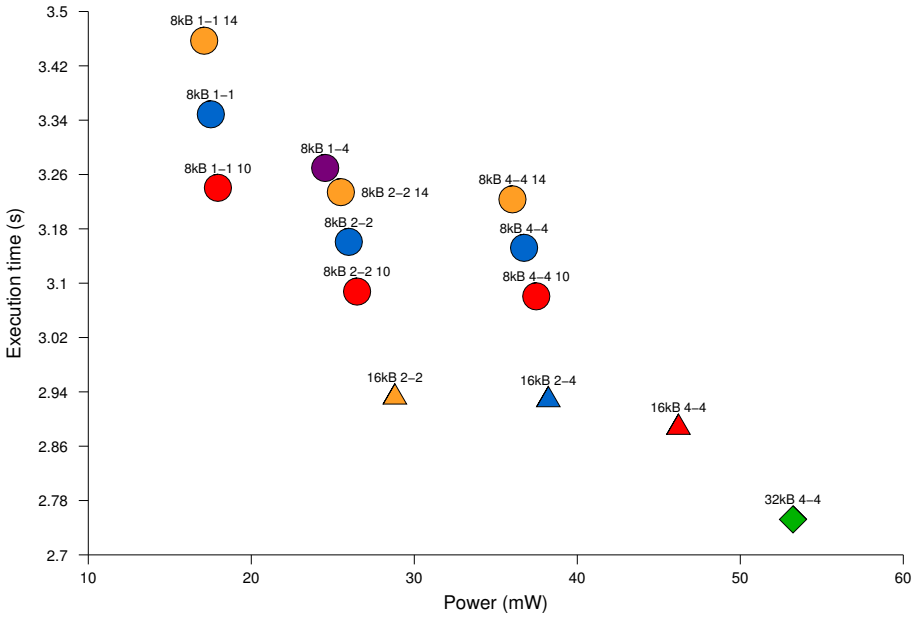

Figure 5.5: Execution time versus power of the different configurations.

Increasing the associativity in the L1DC to four-way yields the 8kB 1-4 configuration. This configuration manages to perform better than the 8kB 1-1 and 8kB 1-1 14 configurations but not the 8kB 1-1 10 configuration. The performance increase can be attributed to the increased associativity in the L1DC clearly visible in Fig. 5.4, which reduces the miss rates in the L1DC (see Sec.2.3.2.1). Increasing the associativity in the L1IC while reducing the associativity in the L1DC yields the 8kB 2-2 configurations. These configurations trade increased miss rates in the L1DC for decreased miss rates in the L1IC. Since the L1IC is accessed almost every cycle, lower L1IC miss rate has a larger impact on performance than lower L1DC miss rate, which explains why the 8kB 2-2 configuration performs better than the aforementioned 8kB configurations. Again, the impact of the L2 latency is observable in Fig. 5.5 with the 8kB 2-2 14 configuration being the slowest, followed by the 8kB 2-2 configuration and then the 8kB 2-2 10 configuration with the lowest latency. However, the observed speedups for these configurations are lower compared to the direct-mapped 8kB 1-1 configurations. The lower speedups are caused by the reduced cache miss rates due to associativity that subsequently decreases the number of L2 accesses susceptible to the access latency. Likewise, the differences in power dissipation is due to the different execution times causing the faster configurations to have a higher power density.

Increasing the associativity of both L1 caches results in the 8kB 4-4 configurations which are the best performing 8kB cache configuration. The performance increase is caused by the higher associativity, which leads to less cache misses in both L1 caches as shown in Fig. 5.4. However, the miss rate reduction is small compared to the 8kB 2-2 configuration which results the minor speedup observable in Fig. 5.5. The minor impact on miss rate for increasing associativity indicates diminishing effects of associativity. Similar effects of increased L2 latency as observed for the afore-

mentioned configurations are visible for the 8kB 4-4 configurations in Fig. 5.5, but they are less pronounced. The reason why the latency affects these configurations less is that the cache miss rates have been further reduced due to higher associativity. For all associative 8kB configurations, the performance increase comes at the price of higher power dissipation. The rise in power dissipation is caused by additional SRAM memories that are added to the associative designs. In the 8kB 1-4 case six memories, in the 8kB 2-2 configuration four memories and in the 8kB 4-4 configuration 12 memories are added when compared to the direct-mapped 8kB 1-1 configurations.

When stepping up the cache capacity to 16kB the performance increases substantially. Larger capacity caches can store larger parts of the program and more data without the need to evict items to the L2 caches. In effect, the number of misses and subsequent L2 accesses are reduced, which can be seen in Fig. 5.4 where the 16kB configurations have lower miss rates in both caches compared to the 8kB configurations. The lowest performing 16kB configuration is the 16kB 2-2 configuration with the lowest degree of associativity. However, as seen in Fig. 5.5 the configuration offers good performance with comparatively low power dissipation that is substantially better than the other 16kB configurations. Moreover, the 16kB 2-2 configuration is both faster and has a lower power dissipation than the smaller 8kB 4-4 configurations. Increasing the associativity in the L1DC to four-way yields the 16kB 2-4 configuration. The increase in associative does increase performance but only slightly which is explained by the minor reduction in the L1DC miss rate shown in Fig. 5.4. In contrast, the power dissipation increases substantially because additional SRAM memories are added to facilitate the associativity, which shifts the configuration to the right in Fig. 5.5. Increasing the associativity in the L1IC results in the 16kB 4-4 configurations which manages to achieve the highest performance of all the 16kB configurations but also dissipates the most power. As the capacity of the cache is increased to 32kB the performance rises considerably but less than the step from 8kB to 16kB. The increase in performance stems from the lower miss rates in the L1DC and especially the L1IC shown in Fig. 5.4. The 32kB cache clocks in at 2.75 seconds which is by a sizable margin the fastest configuration. However with eight large SRAM macros and low execution time the configuration dissipates the most power at around 53 mW.

The general trend that can be observed in the results indicates that higher capacity rather than increased associativity produces the most power-efficient configurations that also boasts good performance increases. This can be explained by the non-linear power increase of SRAM blocks, which are designed to be dense and power-efficient. There are fixed costs associated with the input pins on the blocks and the internal power dissipation scales well with size. Thus using fewer larger SRAM blocks, i.e., lower associativity, is more power-efficient than several smaller SRAM blocks. Another trend is the diminishing returns in performance for increasing cache size and especially associativity, which could indicate that MiBench benchmarks are too simple to capture the performance increases normally associated with high capacity and associative caches.

5.2.3 Energy distribution

The total energy used by the configurations for the entire MiBench suite is shown in Fig. 5.6. The energy is further divided into four categories; 1) clock network, 2) pipeline, 3) L1IC and 4) L1DC. Energy relates execution time to power dissipation, which means that power-efficient designs are not necessarily the most energy-efficient if their performance is too low. However, in this case the 8kB direct-mapped configurations offer adequate performance, in conjunction with their low power dissipation, to also be the most energy-efficient configurations. A minor energy difference can be observed between the 8kB 1-1 configurations which is caused by the difference in L2 latency. Higher L2 latency causes longer simulation times during which the clock network dissipates power. Consequently, the 8kB 1-1 14 configuration uses more energy in the clock network than the 8kB 1-1 configuration and more still when compared to the 8kB 1-1 10 configuration. Similar reasoning on the L2 applies to the 8kB 2-2 configurations. The 8kB 2-2 configurations are less energy-efficient than their direct-mapped counterparts because the speedup offered by the associativity is insufficient to compensate for their higher power dissipation. Interestingly, the 8kB 1-4 configuration is less energy-efficient than the direct-mapped configurations, but more energy-efficient than the 8kB 2-2 configurations. The energy difference between the 8kB 1-4 and 8kB 2-2 configurations can be attributed to higher L1IC utilization compared to L1DC utilization, which causes the power markup in the L1IC to affect the energy efficiency to a larger degree. The 8kB 4-4 configurations are even less energy-efficient as the performance increase is greatly outweighed by the rise in power dissipation in the L1IC and L1DC.

The 16kB 2-2 configuration manages to be the most energy-efficient 16kB configuration while also beating the 8kB 4-4 configurations and breaking even with the 8kB 2-2 configurations. It performs good enough compared to the 8kB 2-2 to compensate for its higher power dissipation and it beats the 8kB 4-4 configurations in both performance and power. The 16kB 2-4 configuration dissipated more power in the L1DC than the 16kB 2-2 configuration while only slightly increasing the performance and is thus less energy-efficient. The least energy-efficient caches are the large capacity and four-way associative 16kB 4-4 and 32kB configurations. The lower execution time offered by both configurations is insufficient to offset the higher power dissipation in the caches.

The general trends are that the caches dominate the energy consumption and the number of SRAM macros, i.e., associativity, is the main source to this. Size, as mentioned while discussing power, does contribute but to less extent. The L1IC tend to consume the most energy when the caches are balanced (same associativity in the caches), which is expected as an instruction is ideally fetched each cycle. In contrast, the L1DC is accessed roughly every fourth instruction. As mentioned, the clock energy depends on the execution time and decreases with increasing performance. However, the clock energy differences are small and is really only observable between the 8kB 1-1 and the 32kB configurations. The energy of the pipelines remains relatively constant across the configurations with a difference of roughly 0.2 mJ

between the 8kB 1-1 and 32kB configurations which most likely is due to synthesis heuristics.

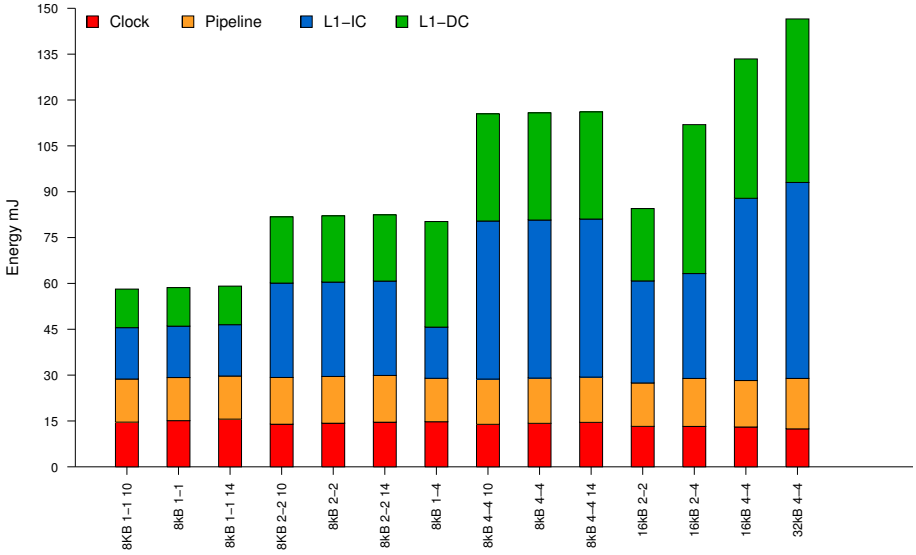

Figure 5.6: Absolute energy of the different configurations.

All results presented thus far have utilized the power scaling discussed in Sec. 4.1.1.2 and the way-halting technique discussed in Sec. 4.2. The impact of these interventions are shown in Fig. 5.7 where the average relative energy distribution for the standard 16kB 4-4 configuration is shown. In a) no power scaling nor way-prediction are used and the L1IC dominates the energy, drowning out the other components' energy. The distribution changes in b) where the power scaling is added. The pipeline energy consumption decreases marginally due to large energy increase in the L1DC. The L1DC is large and dissipates more power compared to the pipeline and is used around 20% of the EEMBC execution time which results in a considerable scaling factor (see Sec. 4.1.1.2). In contrast, the pipeline dissipates comparatively small amounts of power and only parts of the pipeline are scaled as discussed in Sec. 4.1.1.2. The L1IC and clock tree are unaffected by the power scaling and their respective portion shrinks. Lastly, in c) the way-prediction emulation is taken into account which results in a considerable decrease in the L1IC energy.

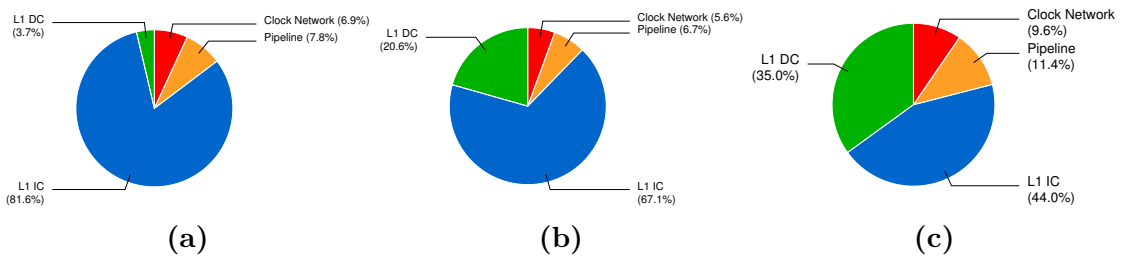

Figure 5.7: Energy distribution for **a)** Unscaled and without way-prediction **b)** Scaled and without way-prediction **c)** scaled with way-prediction

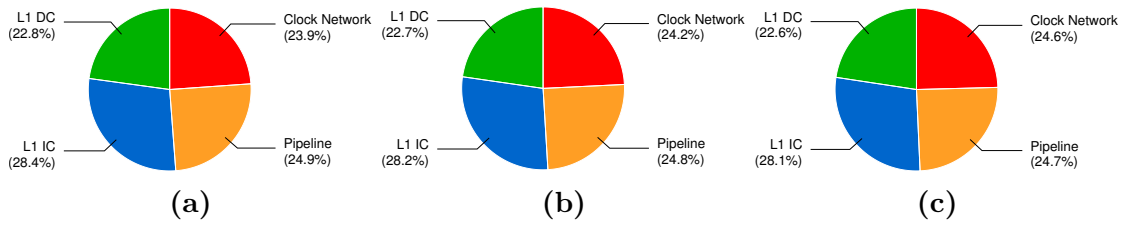

Figure 5.8: Energy distribution of three 8kB 1-1 configurations with different L2 latencies, **a)** 8kB 1-1 10 cycles **b)** 8kB 1-1 12 cycles **c)** 8kB 14 cycles

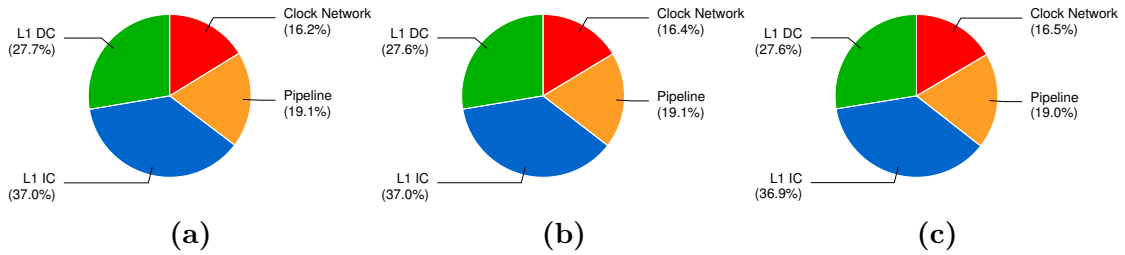

Figure 5.9: Energy distribution of three 8kB 2-2 configurations with different L2 latencies, **a)** 8kB 2-2 10 cycles **b)** 8kB 2-2 12 cycles **c)** 8kB 2-2 14 cycles

The average relative energy distribution for all aforementioned configurations are shown in Fig. 5.11-5.13. What can be seen is that increasing cache sizes, with the exception of lower associative caches, shifts the energy distribution towards the caches. For the 8kB 1-1, 8kB 2-2, 16kB 2-4 and 16kB 2-2 configurations the energy instead shifts towards the clock network and the pipeline, most notably so for the 8kB 1-1 configuration as the cache energy has been reduced significantly when compared to the other configurations.

5.3 Evaluation

This section is split into two parts. Firstly, the framework verification methods presented in Sec. 3.2 will be used to evaluated the framework. Secondly, the work will be held to the goals stated in Sec. 1.1.

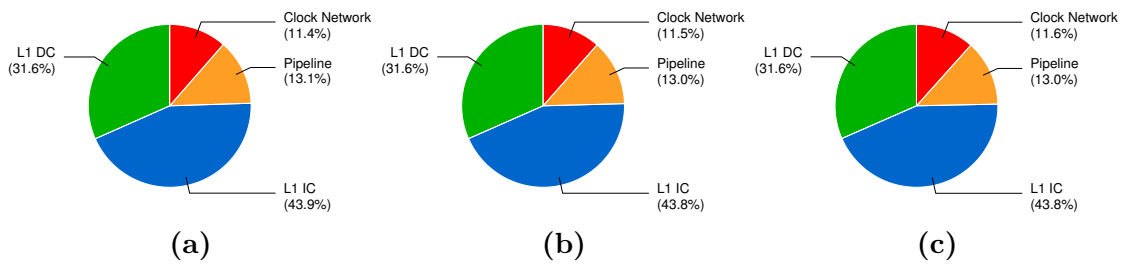

Figure 5.10: Energy distribution of three 8kB 4-4 configurations with different L2 latencies, **a)** 8kB 4-4 10 cycles **b)** 8kB 4-4 12 cycles **c)** 8kB 4-4 14 cycles

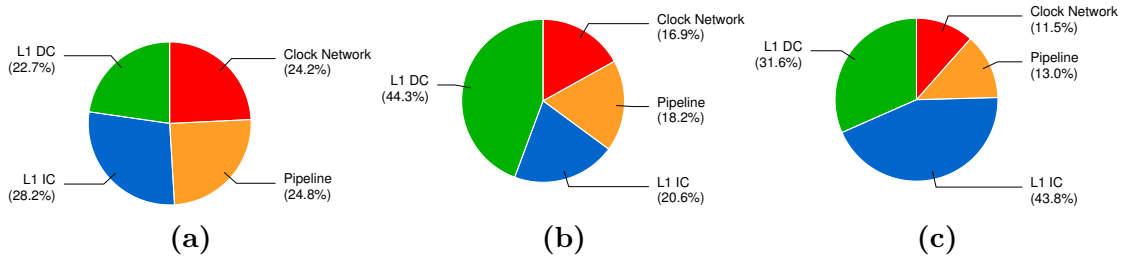

Figure 5.11: Energy distribution of three 8kB configurations, **a)** 8kB 1-1 **b)** 8kB 1-4 **c)** 8kB 4-4

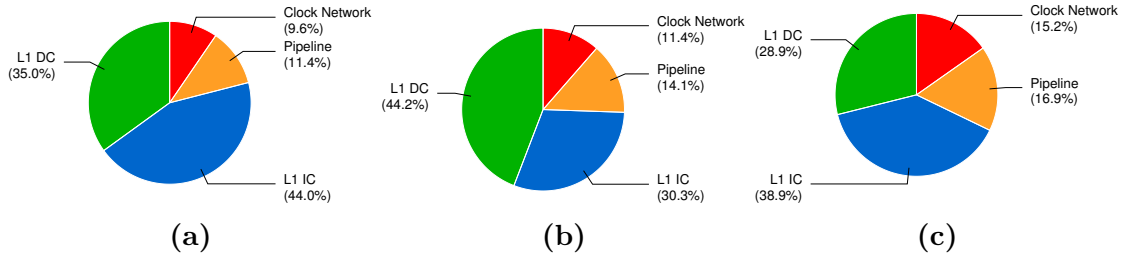

Figure 5.12: Energy distribution the different 16kB configurations: **a)** 16kB 4-4 **b)** 16kB 2-4 **c)** 16kB 2-2

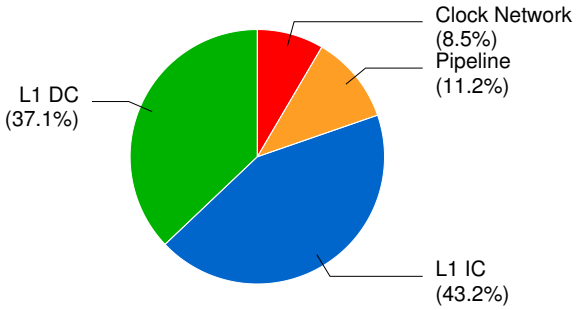

Figure 5.13: Energy distribution 32kB configuration.

5.3.1 Evaluation of verification methods

The functional verification of the RTL code was inherent to the RTL verification and power estimation flow as discussed in Sec. 3.2. However, to make a rudimentary verification of the power estimates they can be compared to access energy estimates used in previous projects. However, in the previous projects only the L1DC power was estimated from the RTL design. Furthermore, these power estimates were based on a probabilistic approach where components of the L1DC were analyzed in isolation. Due to the nature of the earlier projects load and store power are estimated separately. The ad-hoc methodology utilized a 16kB four-way associative cache with least recently used (LRU) as replacement technique. Additionally, the access energy includes data translation lookaside buffer (DTLB) power estimates, which is neglected in CREEP. The access energy of the ad-hoc methodology are shown in Table 5.2 below.

Table 5.2: Power estimates obtained from the RTL during previous projects [4]

Access	Energy (pJ/cycle)
Load	201.2
Store	122.4
Miss w/o writeback	251.2
Miss w writeback	479.1

Power estimates of the L1DC obtained from CREEP using the same configuration, results in an aggregate access energy of 225 pJ. This is close to the previous estimated L1DC load energy. However, as the previous estimation include a DTLB and the aggregated CREEP estimates also include stores, the CREEP estimates could be expected to be lower. However, the slightly higher access energy estimated in CREEP could be explained by the fact that other events such as misses, which have a considerably larger power impact, are also included.

The modified simulator component was verified by inspecting a number of performance counters generated by the simulator. More specifically the number of executed instructions was compared to the number of committed instructions. A mismatch between the two would indicate that the simulator is issuing instructions speculatively. Moreover, as stores were moved in the simulator the number of L1DC accesses was compared to the total number of memory references. Lastly, as a final check the total number of instructions was compared to the total amount of committed instructions. No mismatches were detected in any of the counters, which points to the modified simulator working correctly.

5.3.2 Achievement of goals

The goals of this thesis were stated in Sec. 1.1 and the framework will be evaluated according to these goals. The accuracy of the energy estimates produced by the framework was evaluated previously in Sec. 5.3.1 and based on this rudimentary verification, it seems as the framework is producing reasonable energy estimates. The framework was successfully automated as discussed in Sec. 5.1 and 4.3. Furthermore, the framework can be configured through CREEP configuration files as described in 4.3 and this procedure was facilitated as part of the automation. This feature was demonstrated in Sec. 5.2 where results of several configurations are shown. A case study of a practical way-halting technique called SHA was produced as part of the work, refer to Ch. 6 for more information about SHA. Currently, several venues where the framework can be introduced are being investigated, but this remains a work in progress.

5.4 Discussion

The implementation of CREEP required compromises between scalability and power estimation accuracy as has been discussed in Sec. 3 and Sec. 4.1.1.1. However, there are issues that in hindsight could have been implemented differently resulting in

possible improvements of the energy estimates produced by the framework.

The biggest bottleneck in the framework is the MiBench suite that proved to be too simple to fully characterize the behavior of large capacity and associative caches. It is likely that a more comprehensive suite, like SPEC, would address this issue. SPEC would fully load the configurations with larger data sets and under longer executions and would better elucidate the benefits of cache capacity and associativity.

The energy approximations of the caches, and the rest of the pipeline, are based on average power dissipation and make no distinction between accesses, i.e., load and stores and whether these are hits or misses. These accesses represent events with vastly different power costs, with loads being roughly twice as costly as store and misses dissipating even more power. However, more accurate energy estimations would require the use of either probabilistic methods or value change dump (VCD). The probabilistic approach could have been used to characterize each kind of access. Alternatively a characteristic power consumption could have been extracted from a VCD file for each access. Both would reduce the scalability of the framework as each approach would need to be adapted every new configuration and cache size. It is possible that a parametric model could have been deduced, like many of the related works discussed in Sec. 1.3, but that would compromise the integration aspect of CREEP.

Another issue with the current cache power estimates is the implemented way-prediction technique. The power of a successful prediction is calculated as one fourth of the cache access power, i.e., three fourth of the power is saved. This calculation does not take into account that the power savings are only related to the SRAM memories. However, addressing this issue would require reverting to the power estimation techniques used in the ad-hoc methodology but this would greatly reduce the scalability of the framework. Furthermore, the power penalties for a failed prediction is not taken into account.

The framework does not include any access energy estimates for lower levels in the memory hierarchy, such as L2 or main memory. Lower levels in the memory hierarchy are not only larger and slower, but they also requirer large amounts of energy on an access. It is likely that if this energy would have been included the evaluation done in Sec. 5.2 would favor larger and possibly associative caches as these were demonstrated to have smaller miss rates in Fig. 5.4. Fewer misses would in the long run lead to less accesses to the L2 cache, possibly resulting in less energy usage. In contrast, the smaller caches would require more accesses to the L2 cache, which would increase the power dissipation. All in all, the energy efficiency design point is likely to shift towards larger caches.

During power estimation a zero delay model was used which could affect the accuracy of the power estimates. If signal propagation was taken into account power dissipation caused by glitches would have been accounted for [23]. However, whether the glitching power would affect the overall results of the framework to any meaning-

ful degree is questionable, especially since the glitching power only affects the logic elements of the design. Using a unit delay model with a fixed delay could be investigated with minimal impact on scalability but using delay models based around standard parasitic exchange format (SPEF) files would increase the complexity of the RTL verification flow.

The leakage power could have been handled differently. As discussed in Sec. 4.1.1.2 the total power is extracted from the design, which includes the leakage power. The total power is then subsequently scaled, which causes the leakage power to be scaled erroneously. The leakage power does however constitute a fraction of the total power and when coupled with the performance counters this should have limited impact on the final results. While the leakage power is indirectly included, its accumulative effect could currently be underestimated. To properly estimate the the leakage energy the leakage power of the design should instead be combined with the simulated cycles similar to the clock network power. However, separating the different causes of power dissipation would have increased the complexity of the power calculations and hence reduce the scalability.

The place and route (PnR) scaling that is implemented in the framework (see Sec. 4.1.1.2) could have been done at a finer granularity. Because of technical issues the placed and routed netlist could not be verified in the implemented RTL verification flow. Consequently it was not possible to obtain use-case based switching activity for the post-PnR netlist. Instead, a probabilistic method was used which meant that the scope at which the scaling factor could be derived was limited. The scaling factor was derived from the arithmetic logic unit (ALU) and was applied to units which also included registers that scales differently. Thus, it is likely that a mismatch in PnR power growth was allowed to affect the final framework energy results.

The clock network power was obtained as discussed in Sec. 4.1.1.2. This approach causes clock energy to be underestimated as the clock pin power is neglected on all the cells in the clock tree. Instead the clock pin-power is distributed to the pipeline units and caches. Consequently, the clock power is accounted for but not attributed to the clock network. Lastly, the clock tree power was derived from the standard 16kB configuration and it is likely that power varies between the different designs. However, to extract accurate clock tree powers all the designs would have to be placed and routed separately which was unfeasible to include in the framework.

Lastly, the framework needs to be verified to a greater extent before it is adapted by the research community. The current verification, described in Sec. 5.3.1, is rudimentary at best and only verifies the individual components. For the framework to be satisfactorily verified the resource mapping needs to be scrutinized. Initially, this was planned as a part of this work and the method of choice was based on EEMBC simulations. The EEMBC benchmarks were to be simulated by SimpleScalar and the energy results were to be compared to the energy estimates based on the average power dissipation of the design when combined with number of simulated cycles

produced by SimpleScalar. However, this approach suffered technical difficulties as the EEMBC benchmarks did not compile for the SimpleScalar simulator.

6

Case study

This chapter will showcase the framework methodology in a case study that shows how the framework can be customized to suit a specific project. The automated workflow cannot accommodate new register-transfer level (RTL) or performance counters in the simulator without changes to the automating scripts. Thus, framework customization is reserved for advanced users that do not need to rely on the automated standard workflow.

6.1 SHA - practical way-halting

The idea of this project was to address issues with earlier way-halting techniques that were impractical to implement, either posing limitations on static random access memory (SRAM) macros or causing long critical paths that would impact the speed of the data cache [43][44]. Both of these issues could be addressed by speculatively accessing the cache during the address generation stage as proposed by Bardizbanyan et al. [4] using a technique called STA (Speculative Tag Access). The combination of STA and way-halting was dubbed Speculative Halt-tag Access (SHA) and resulted in a publication at Design Automation and Test in Europe 2016 [3].

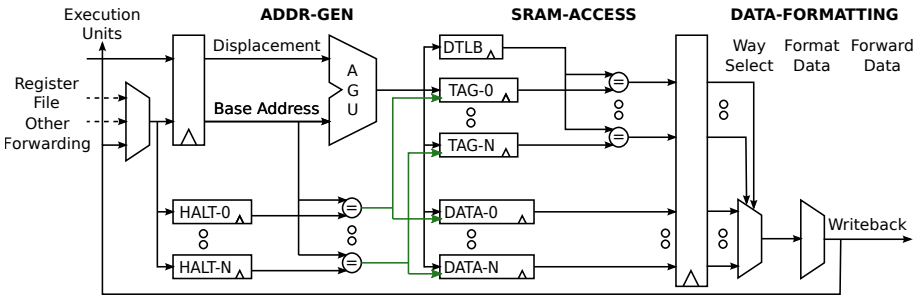

Figure 6.1: Overview of SHA. [3]

SHA works by accessing halt-tags, which contains the lower portion of the tag, with the base-address index supplied by the resister-file or by forwarding paths as shown in Fig. 6.1. The halt-tags are then compared to the lower-order tag bits of the base address. If a hit is detected in the halt-tag array, corresponding tag and data-array is enabled. Conversely, if no hit is detected in the halt-ways the enable

signal is not asserted and the consequent access to tag and data-arrays is avoided. Ideally only one halt-way would signal a hit and only one tag and data-way would be accessed, thus saving roughly 75% of a conventional data cache's read energy. For stores, the energy saving is less as stores are accessed sequentially, i.e., only the tag-array access energy can be reduced by halting. If more hits are detected in the halt-tags for both reads and writes, corresponding tag and data-arrays are enabled causing the energy savings to decrease gracefully. The address generation unit (AGU) address computation is shown in Fig. 6.2 and speculative accesses are successful when the base address does not cause an overflow from the line offset into the line index. This information can easily be extracted from the adder and if a speculation failure is detected the cache is accessed conventionally the next cycle. A speculation failure comes at an energy overhead of a halt-tag access and has no negative effect on performance.

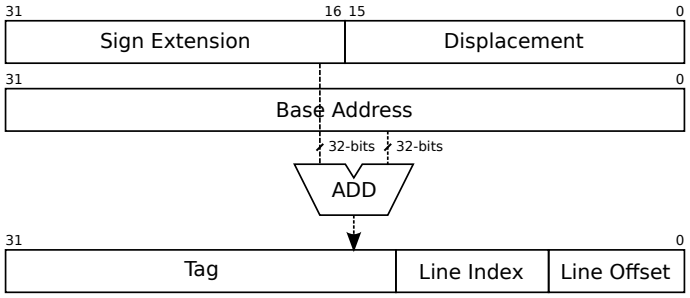

Figure 6.2: AGU address calculation showing the address fields of interest. [3]

The evaluation method of the technique is based on the methodology embodied by CREEP. However, because the project required modifications to be done to the components of the framework, the automated workflow was not used.

The RTL component of the framework remained largely intact, with the design being brought through a similar design flow to the standard workflow used in CREEP. The exception was the power estimation methodology which was changed to a probabilistic method because the scope of the power estimation was limited to the level-one data cache (L1DC). This allowed the L1DC access power to be characterized in greater detail, i.e., for read, writes and misses. The access energy estimates for the different cache components are shown in Table 6.1. These estimates were then combined to form the complete cache operations shown in Table 6.2 and 6.3, which list the energy of different SHA cache events and cache miss energy respectively.

The simulator component was changed more substantially as several new architectural events needed to be tracked and the cache component of the simulator needed to be modified. The new events were related to the address generation and how often the speculative accesses were successful, i.e., whether the address offset would cause an overflow from the line offset into the index. 16 counters were added, one for each bit in the address offset and these were increased on the condition that no overflow was caused (speculative success) by that bit. The cache modifications were aimed at emulating the halt-tag array. This was done by forcing the cache to do two cache accesses, the first to do a linear search through the tag-array to

Table 6.1: L1 DC Component Energy. [3]

Component	Energy (pJ)
Read Halt	19.1
Write Halt	17.7
Read Tag	19.1
Write Tag	17.6
Read Data	26.5
Write Data	27.2
DTLB	17.5
Peripheral	18.8
Arbiter	2.0

Table 6.2: Components Accessed for Each Case. [3]

Case	Read Halt	Read Tag	Read Data	Write Data	DTLB	Peripheral	Energy (pJ)
BL	-	3	4	0	1	1	182.1
BS	-	3	0	1	1	1	103.3
STA0	-	3	1	0	1	1	102.6
STA1	-	6	4	0	1	1	239.4
SHA0	0	4	4	0	1	1	201.2
SHA1	0	4	0	1	1	1	122.4
SHA2:0	1	0	0	0	1	1	37.9
SHA2:1	1	1	1	0	1	1	83.5
SHA2:2	1	2	2	0	1	1	129.1
SHA2:3	1	3	3	0	1	1	174.7
SHA2:4	1	4	4	0	1	1	220.3
SHA3	1	4	4	0	1	1	220.3
SHA4:0	1	0	0	0	1	1	37.9
SHA4:1	1	1	0	1	1	1	84.2
SHA4:2	1	2	0	1	1	1	103.3
SHA4:3	1	3	0	1	1	1	122.4
SHA4:4	1	4	0	1	1	1	141.5
SHA5	1	4	0	1	1	1	141.5

find a potential halt-tag hit and a second time to produce a proper tag hit. The halt-tag search was based on a partial tag comparison with flexible width based on the desired halt-tag width. Performance counters were added for the five possible cases: 1) no hit was detected in any of the halt-ways, 2) one hit detected, 3) two hits detected, 4) three hits detected and 5) four hits detected. Several modifications to the cache component were necessary to achieve this, e.g., the tag structure was changed from a hashmap to a linked list and fast lookups were disabled (features that improved the speed of the simulator). Additionally, the cache component was augmented to accept an additional argument that disables the the halt-tag search. The halt-tag search was then enabled based on the selected address offset width so that it was only enabled within the selected offset and for speculative successes. Lastly, the cache misses were also monitored as these are not accounted for in the other counters and represent energy costly events.

The performance counters were then combined with the access energy estimates in Table 6.2 for the use-cases in a similar manner as in the standard CREEP workflow. Depending on the selected address offset width, different amounts of speculative successes and failures were recorded. The speculative successes were then further

Table 6.3: Components Accessed on Miss Events. [3]

Case	Write Halt	Write Tag	Read Data	Write Data	Peripheral	Arbiter	Energy (pJ)
B-NoWB	-	1	0	8	8	8	251.2
B-WB	-	1	8	8	16	16	479.1
H-NoWB	1	1	0	8	8	8	268.9
H-WB	1	1	8	8	16	16	496.8

categorized as an **SHA2:X** or an **SHA4:X** (load or store) access where the X denotes how many halt-ways produced a hit. The speculative failures were combined with **SHA3** and **SHA5** counters. Accesses for which the address offset exceeded the chosen width are combined with **SHA0** and **SHA1** (loads and stores). The counted cache misses were then combined with the events in Table 6.3.

The final results of the customized workflow is shown in Fig. 6.4 where the SHA technique on average saves 25.6% energy compared to a conventional data cache. Compared to the STA technique, it saves an additional 7.4% as can be seen in Fig. 6.4. In addition to saving more energy on loads than STA, SHA also saves energy on stores.

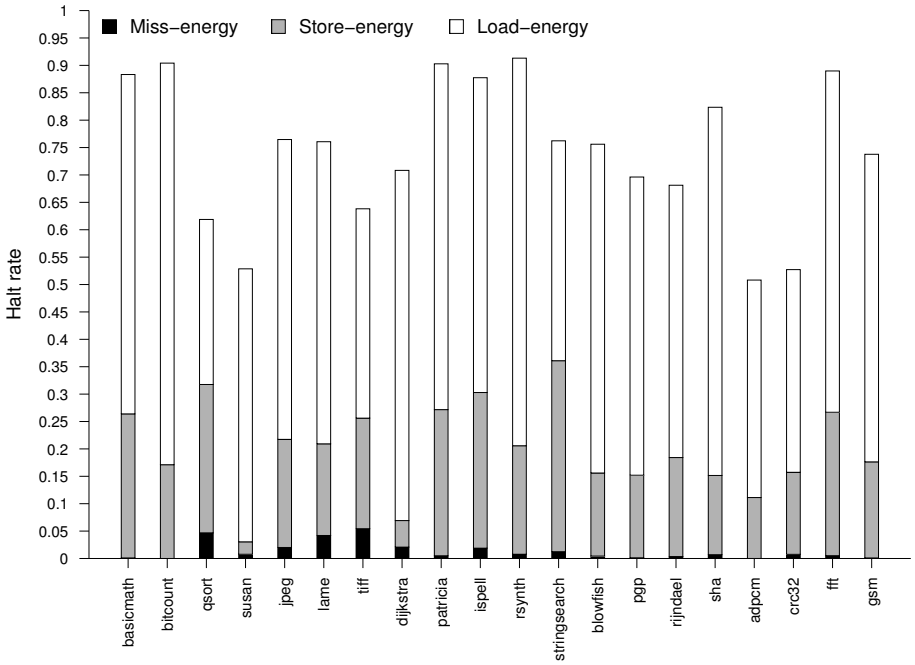

Figure 6.3: SHA energy for the MiBench suite used in the CREEP framework. [3]

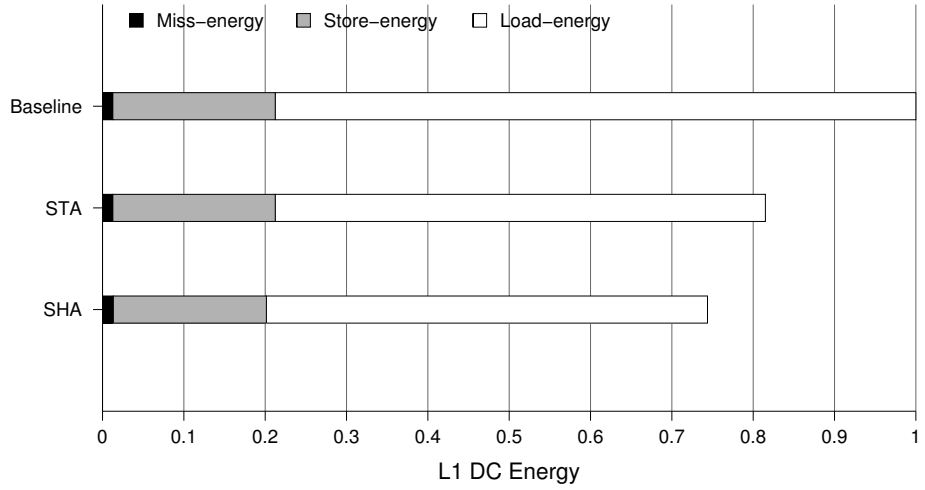

Figure 6.4: SHA energy compared to STA and a conventional baseline cache. [3]

7

Conclusion

Chalmers RTL-based energy evaluation framework for pipelines (CREEP), a framework that combines register transfer (RT)-level power estimates with SimpleScalar, an architectural-level simulator, to allow software and hardware co-evaluation was implemented and demonstrated. The register-transfer level (RTL) component of the framework consists of a MIPS I compliant 5-stage pipeline (5SP) with integrated level-one (L1) caches, which are synthesized for the 65nm process node. The power estimates of the design are obtained at a pipeline-unit basis through instruction set architecture (ISA) simulation based on the EEMBC benchmark suite. The power estimates are scaled in order to approximate the impact of place and route (PnR), which allows the framework to estimate the power of a range of designs. The simulator component is used to model the resource usage of the RTL 5SP to produce resource usage statistics of pipeline units based on the MiBench benchmark suite. The final energy estimates of the framework are obtained by combining the RTL power estimates with the simulated resource statistics. The framework is fully automated and supports configuration of the integrated RTL caches through configuration files. The framework was demonstrated by selecting a range of configurations, spanning the design space supported by the framework. Lastly, the extendability of the framework was showcased in a case-study of SHA, a practical way-halting technique, which was shown to save 25.6% of the energy used in a level-one data cache (L1DC).

Bibliography

[1] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2012.

[2] D. M. Harris and S. L. Harris, *Digital design and computer architecture*. Elsevier, 2013.

[3] D. Moreau, A. Bardizbanyan, M. Själander, D. Whalley, and P. Larsson-Edefors, "Practical way halting by speculatively accessing halt tags," *to be presented at the 2016 Conference on Design, Automation & Test in Europe (DATE)*.

[4] A. Bardizbanyan, M. Själander, D. Whalley, and P. Larsson-Edefors, "Speculative tag access for reduced energy dissipation in set-associative L1 data caches," in *Proc. of IEEE 31st International Conference on Computer Design (ICCD)*. IEEE, 2013, pp. 302–308.

[5] S. Kaxiras and M. Martonosi, "Computer architecture techniques for power-efficiency," *Synthesis Lectures on Computer Architecture*, vol. 3, no. 1, pp. 1–207, 2008.

[6] T. Kuroda, "CMOS design challenges to power wall," in *International Microprocesses and Nanotechnology Conference*, Oct. 2001, pp. 6–7.

[7] B. Davari, R. Dennard, and G. Shahidi, "CMOS scaling for high performance and low power-the next ten years," *Proc. of the IEEE*, vol. 83, no. 4, pp. 595–606, Apr. 1995.

[8] P. Somavat and V. Namboodiri, "Energy consumption of personal computing including portable communication devices," *Journal of Green Engineering*, vol. 1, no. 4, pp. 447–475, 2011.

[9] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 83–94, May 2000.

[10] S. Li, J. H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. of 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2009, pp. 469–480.

[11] V. Saljooghi, A. Bardizbanyan, M. Sjalander, and P. Larsson-Edefors, "Configurable RTL model for level-1 caches," in *Proc. of NORCHIP*, Nov. 2012.

[12] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[13] S. Wilton and N. Jouppi, "CACTI: an enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, May 1996.

[14] ITRS. [Online]. Available: http://www.itrs.net/reports.html

[15] D. Brooks, P. Bose, and M. Martonosi, "Power-performance simulation: design and validation strategies," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 4, pp. 13–18, 2004.

[16] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "The design and use of SimplePower: a cycle-accurate energy estimation tool," in *Proc. of the 37th Annual Design Automation Conference (DAC)*, 2000, pp. 340–345.

[17] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye, "Energy-driven integrated hardware-software optimizations using SimplePower," in *Proc. of the 27th Annual International Symposium on Computer Architecture*, 2000, pp. 95–106.

[18] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz, "Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 26–36.

[19] R. Rodrigues, A. Annamalai, I. Koren, and S. Kundu, "A study on the use of performance counters to estimate power in microprocessors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 60, no. 12, pp. 882–886, 2013.

[20] J. Laurent, N. Julien, E. Senn, and E. Martin, "Functional level power analysis: An efficient approach for modeling the power consumption of complex processors," in *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*. IEEE Computer Society, 2004, p. 10666.

[21] N. H. Weste and D. M. Harris, *Integrated circuit design*. Pearson, 2011.

[22] V. Tiwari, J. Monteiro, and R. Patel, ser. Industrial Information Technology. CRC Press, Mar. 2006, ch. Power Analysis and Optimization from Circuit to Register-Transfer Levels, pp. 3–1–3–16, 2. [Online]. Available: http://dx.doi.org/10.1201/9781420007954.ch3

[23] R. Damiano and R. Camposano, ser. Industrial Information Technology. CRC Press, Mar. 2006, ch. The Integrated Circuit Design Process and Electronic Design Automation, pp. 2–1–2–14, 2. [Online]. Available: http://dx.doi.org/10.1201/9781420007947.ch2

[24] *ModelSim®*, Mentor Graphics, Inc., Feb. 2016.

[25] *Incisive Enterprise Simulator (IES)*, Cadence Design Systems, Inc., Jul. 2011.

[26] *VCS®*, Synopsys, Inc., Feb. 2016.

[27] *Encounter RTL Compiler®*, Cadence Design Systems, Inc., Feb. 2016.

[28] *Design Compiler®*, Synopsys, Inc., Mar. 2010.

[29] *HDL Designer®*, Mentor Graphics, Inc., Feb. 2016.

[30] *Encounter® Digital Implementation (EDI)*, Cadence Design Systems, Inc., Jul. 2011.

[31] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross, F. Baskett, and J. Gill, "MIPS: A microprocessor architecture," in *ACM SIGMICRO Newsletter*, vol. 13, no. 4. IEEE Press, 1982, pp. 17–22.

[32] B. R. Rau and J. A. Fisher, "Instruction-Level Parallel Processing: History, Overview, and Perspective," *The Journal of Supercomputing*, vol. 7, no. 1-2, pp. 9–50, 1993.

[33] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus IPC: The end of the road for conventional microarchitectures," in *Proc. of the 27th Annual International Symposium on Computer Architecture*. ACM, 2000, pp. 248–259.

[34] M. Dubois, M. Annavaram, and P. Stenström, *Parallel computer organization and design*. Cambridge University Press, 2012.

[35] A. Bardizbanyan, M. Själander, D. Whalley, and P. Larsson-Edefors, "Reducing set-associative L1 data cache energy by early load data dependence detection (ELD^3)," in *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, ser. DATE '14, 2014, pp. 82:1–82:4.

[36] ——, "Improving data access efficiency by using context-aware loads and

stores," in *Proc. of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, 2015, pp. 3:1–3:10.

[37] ——, "Designing a practical data filter cache to improve both energy efficiency and performance," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 4, pp. 54:1–54:25, Dec. 2013.

[38] D. Hathaway, L. Stok, D. Chinnery, and K. Keutzer, ser. Industrial Information Technology. CRC Press, Mar. 2006, ch. Design Flows, pp. 1–2–1–15, 2. [Online]. Available: http://dx.doi.org/10.1201/9781420007954.sec1

[39] *PrimeTime® PX*, Synopsys, Inc., Jun. 2011.

[40] Spec. [Online]. Available: https://www.spec.org/benchmarks.html

[41] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *IEEE International Workshop on Workload Characterization.* IEEE, 2001, pp. 3–14.

[42] M. D. Powell, A. Agarwal, T. Vijaykumar, B. Falsafi, and K. Roy, "Reducing set-associative cache energy via way-prediction and selective direct-mapping," in *Proc. of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, 2001, pp. 54–65.

[43] C. Zhang, F. Vahid, J. Yang, and W. Najjar, "A way-halting cache for low-energy high-performance systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 2, no. 1, pp. 34–54, Mar. 2005.

[44] ——, "A way-halting cache for low-energy high-performance systems," in *International Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2004, pp. 126–131.