Microarchitectural Low-Power Design Techniques for Embedded Microprocessors

THÈSE N^{O} 7168 (2016)

PRÉSENTÉE LE 11 NOVEMBRE 2016

À LA FACULTÉ DES SCIENCES ET TECHNIQUES DE L'INGÉNIEUR
LABORATOIRE DE CIRCUITS POUR TÉLÉCOMMUNICATIONS
PROGRAMME DOCTORAL EN GÉNIE ÉLECTRIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Jeremy Hugues-Felix CONSTANTIN

acceptée sur proposition du jury:

Prof. Y. Leblebici, président du jury
Prof. A. P. Burg, Prof. D. Atienza Alonso, directeurs de thèse
Prof. A. Chattopadhyay, rapporteur
Prof. G. Karakonstantis, rapporteur
Prof. L. Benini, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2016

Acknowledgements

I would like to express my gratitude to my thesis advisor Prof. Andreas Burg for giving me the opportunity to work together on such interesting topics throughout my PhD time, and for creating a supportive work environment in his laboratory. I am very grateful for our many technical discussions from which I learned a lot over the years regarding all aspects of engineering, first at ETH during my master studies and then later at EPFL. Moreover, I thank him for always having an open ear and that he was always available, whatever the problem or question, especially during important times.

I acknowledge my co-advisor Prof. David Atienza for enabling the good cooperation between his laboratory (ESL) and the Telecommunications Circuits Laboratory (TCL), when I first arrived at EPFL and during the early time of my PhD. Prof. Luca Benini, Prof. Anupam Chattopadhyay, and Prof. Georgios Karakonstantis I would like to thank for being the examiners on my thesis defense jury, and Prof. Yusuf Leblebici for taking the role as the jury president.

I want to thank Ahmed Dogan from ESL for the fruitful cooperation during the first years of my PhD, building and researching architectures for biomedical signal processing. My thanks also go to Oskar Andersson for his support with the sub-threshold characterization of TamaRISC-CS, and Pascal Meinerzhagen who generously provided the sub-threshold capable memories for the design. I am grateful to Frank Gürkaynak for our discussions and his help on the cores with SHA-3 ISEs. For their contributions to the LISA model of the OpenRISC and its extensions for fault injection, I would like to extend my thanks to Lai Wang and Zheng Wang, respectively.

I am particularly grateful to team DynOR (Andrea Bonetti, Adam Teman, Christoph Müller, and Lorenz Schmid) for all their efforts and extraordinary contributions to the successful design and measurement of our first test chip in 28 nm FD-SOI. This project would not have been possible without them.

During my time at EPFL I had the pleasure to work in a wonderful environment. I was lucky to meet many terrific people over the years and had a number of great colleagues, who especially were always up for fun activities inside as well as outside the lab, which made my stay at TCL that much more enjoyable. I will take many fond memories of this time with me, from the many culinary adventures of various kinds to great nature excursions, or just the funniest discussions that we had in the office. I am very happy to be able to call them my friends: Adi Teman, Alexios Balatsoukas Stimming, Andrea Bonetti, Andrew Austin, Christian Senning, Christoph Müller, Filippo Borlenghi, Georgios Karakonstantis, Kynthia Chamilothori, Lorenz Schmid, Maitane Barrenetxea, Nicholas Preyss, Orion Afisiadis, Pablo Garcia, Pascal Giard, Pascal Meinerzhagen, Pavle Belanovic, Reza Ghanaatian, Rubén Braojos, Shrikanth Ganapathy.

Acknowledgements

Finally, I want to thank my family for all their support over the years and for always believing in my goals, which helped me immensely during all my education. I also want to express my deepest gratitude to my girlfriend Sandra for all her love and her continuous support, which always kept me going through the last years of this journey.

Lausanne, 2 November 2016 Jeremy Constantin

Abstract

Over the last two decades, embedded processing has become omnipresent in all forms of electronic devices in order to provide increasingly complex features and richer user experiences. There is moreover a strong trend towards wireless, battery-powered, portable embedded systems which have to operate under stringent energy constraints. Consequently, low power consumption and high energy efficiency have emerged as the two key criteria for embedded microprocessor design. While technology scaling continues to provide improvements in both performance and power, architectural design has become equally if not even more important to meet desired performance requirements with severely limited power budgets. In this thesis we present a range of microarchitectural low-power design techniques which enable the increase of performance for embedded microprocessors and/or the reduction of energy consumption, e.g., through voltage scaling.

A popular technique for the improvement of processor performance in application-specific systems/scenarios are instruction set extensions (ISEs). In the context of cryptographic applications, we explore the effectiveness of ISEs for a range of different cryptographic hash functions on a microcontroller architecture. Specifically, we demonstrate the effectiveness of light-weight ISEs based on lookup table integration and microcoded instructions using finite state machines for operand and address generation. The proposed ISE concepts are evaluated on the PIC24 architecture (a 16-bit microcontroller) with the final round SHA-3 candidate hash algorithms.

On-node processing in autonomous wireless sensor node devices requires deeply embedded cores with extremely low power consumption. To address this need, we present TamaRISC, a custom-designed ISA with a corresponding ultra-low-power microarchitecture implementation. The TamaRISC architecture is employed in conjunction with an ISE and standard cell memories to design a sub-threshold capable processor system targeted at compressed sensing applications. For a case study of electrocardiogram (ECG) signal compression we show that significant power savings of more than 11x over the state of the art for programmable architectures are possible. We furthermore employ TamaRISC in a hybrid SIMD/MIMD multi-core architecture targeted at biomedical signal processing applications with moderate to high processing requirements (>1 MOPS). A range of different microarchitectural techniques for efficient memory organization are presented, which can provide significant energy savings of more than 50%. Specifically, we introduce a configurable data memory mapping technique for private and shared access, as well as instruction broadcast together with synchronized code execution based on checkpointing.

We then study an inherent suboptimality due to the worst-case design principle in synchronous circuits, and introduce the concept of dynamic timing margins. We show that dynamic timing margins exist in microprocessor circuits, and that these margins are to a large extent state-dependent and that they are correlated to the sequences of instruction types which are executed within the processor pipeline. To perform this analysis we propose a circuit/processor characterization flow and tool called dynamic timing analysis. Moreover, this flow is employed in order to devise a high-level instruction set simulation environment for impact-evaluation of timing errors on application performance. The presented approach improves the state of the art significantly in terms of simulation accuracy through the use of statistical fault injection.
The dynamic timing margins in microprocessors are then systematically exploited for throughput improvements or energy reductions via our proposed instruction-based dynamic clock adjustment (DCA) technique. To this end, we introduce a 32-bit microprocessor with cycle-by-cycle dynamic clock adjustment. The microarchitecture of the employed OpenRISC core comprises a 6-stage pipeline, and its implementation is specifically tuned for DCA. Besides a comprehensive design flow and simulation environment for evaluation of the DCA approach, we additionally present a silicon prototype of our DCA-enabled microarchitecture fabricated in 28 nm FD-SOI CMOS. The fully operational test chip includes a suitable clock generation unit which allows for cycle-by-cycle clock adjustment over a wide range with fine granularity at frequencies exceeding 1 GHz. Measurement results of speedups and power reductions through voltage scaling are provided.

Keywords: low-power design, microarchitecture, instruction set architecture, instruction set extensions, ultra-low-power embedded processor, microcontroller, ASIC, VLSI design, sub-threshold operation, multi-core architecture, sensor nodes, biomedical signal processing, cryptographic hash functions, SHA-3, compressed sensing, dynamic timing margins, dynamic timing analysis, dynamic clock adjustment, timing errors, statistical fault injection, approximate computing, OpenRISC

Zusammenfassung

In den letzten zwei Jahrzehnten ist eingebettet Datenverarbeitung in jeglichen Formen von Elektronik allgegenwärtig geworden, um immer komplexere Funktionen und reichere User Experiences zur Verfügung zu stellen. Es gibt zudem einen starken Trend zu drahtlosen, batteriebetriebenen, tragbaren eingebetteten Systemen, die unter strengen Energieeinschränkungen arbeiten müssen. Folglich haben sich niedriger Energieverbrauch und hohe Energieeffizienz als die beiden Schlüsselkriterien für eingebetteten Prozessorentwurf entwickelt. Obwohl CMOS-Technologiefortschritte weiterhin Verbesserungen in puncto Rechenleistung und Energieverbrauch ermöglichen, ist Architekturentwurf mittlerweile gleichbedeutend oder sogar noch bedeutender geworden um gewünschte Rechenleistungsanforderungen mit stark beschränkten Energiebudgets zu erfüllen. In dieser Dissertation präsentieren wir eine Reihe von mikroarchitekturellen energiesparenden Entwurfstechniken, die erhöhte Leistung für eingebettete Mikroprozessoren und/oder die Verringerung des Energieverbrauchs, zum Beispiel durch Spannungsskalierung, ermöglichen.

Eine beliebte Technik zur Verbesserung der Prozessorleistung in anwendungsspezifischen Systemen/Szenarien sind Befehlssatzerweiterungen (BSE). Im Rahmen von kryptographischen Anwendungen untersuchen wir die Effektivität von BSE für eine Reihe von verschiedenen kryptographischen Hash-Funktionen auf einer Mikrocontroller-Architektur. Insbesondere zeigen wir die Effektivität von leichten/einfachen BSE basierend auf Integration von Umsetzungstabellen und mikrocodierten Instruktionen unter Verwendung von endlichen Zustandsautomaten für die Generierung von Operanden und Adressen. Die vorgeschlagenen BSE-Konzepte werden auf der PIC24 Architektur (ein 16-bit Mikrocontroller) zusammen mit den SHA-3 Hash-Algorithmus-Kandidaten der letzten Runde ausgewertet.

Datenverarbeitung in autonomen Funksensorknoten erfordert tief eingebettete Rechenkerne mit extrem niedrigem Stromverbrauch. Um diesem Bedarf zu begegnen, stellen wir TamaRISC vor, eine speziell entworfene Befehlssatzarchitektur mit einer dazugehörigen ultra-energiesparenden Mikroarchitektur-Umsetzung. Die TamaRISC Architektur wird in Verbindung mit einer BSE und Standardzellen-Speicher verwendet, um ein Prozessorsystem für Anwendungen basierend auf komprimierter Abtastung zu entwerfen, welches unterhalb der Schwellenspannung betrieben werden kann. Für eine Fallstudie von Elektrokardiogramm (EKG) Signalkompression zeigen wir, dass erhebliche Energieeinsparungen von mehr als 11x über den Stand der Technik für programmierbare Architekturen möglich sind. Wir verwenden außerdem TamaRISC in einer hybriden SIMD/MIMD Mehrkernarchitektur welche für biomedizinischen Signalverarbeitungsanwendungen mit mittleren bis hohen Anforderungen an

die Verarbeitung (>1 MOPS) entworfen ist. Eine Reihe von unterschiedlichen Mikroarchitekturtechniken für eine effiziente Speicherorganisation werden vorgestellt, welche erhebliche Energieeinsparungen von mehr als 50% ermöglichen. Insbesondere stellen wir eine Technik für konfigurierbare Speicher-Address-Übersetzung für privaten und gemeinsamen Zugriff, sowie Befehlsübertragungen zusammen mit synchronisierter Codeausführung auf Basis von Programmhaltepunkten.
Wir untersuchen dann eine inhärente Suboptimalität aufgrund des Schlimmstfall-Entwurfsprinzips von synchronen Schaltungen, und führen das Konzept der dynamischen Zeit-Margen ein. Wir zeigen, dass dynamische Zeit-Margen in Mikroprozessorschaltungen vorhanden sind, und dass diese Margen zu einem großen Teil zustandsabhängig sind, und dass sie zu den Sequenzen von Befehlstypen korreliert sind, die innerhalb der Prozessor-Pipeline ausgeführt werden. Zur Durchführung dieser Analyse schlagen wir eine Schaltung/Prozessor-Charakterisierungs-Methodik und Werkzeug namens dynamische Zeit-Analyse vor. Darüber hinaus verwenden wir diese Methodik, um eine High-Level-Befehlssatz-Simulationsumgebung zu entwickeln für die Evaluation der Auswirkung von zeitlichen Fehlern auf die Anwendungsleistung. Der vorgestellte Ansatz verbessert den Stand der Technik im Hinblick auf die Simulationsgenauigkeit durch den Einsatz von statistischer Fehlerinjektion.
Die dynamischen Zeit-Margen in Mikroprozessoren werden dann für die Durchsatzsteigerung oder für Energieeinsparungen durch unsere vorgeschlagene befehlsbasierte dynamische Taktanpassung (DT) systematisch genutzt. Zu diesem Zweck stellen wir einen 32-bit Mikroprozessor mit Zyklus-zu-Zyklus DT vor. Die Mikroarchitektur des eingesetzten OpenRISC Kerns besteht aus einer 6-stufigen Pipeline und ihre Implementierung ist für DT speziell abgestimmt. Neben einer umfassenden Entwurfsmethodik und Simulationsumgebung für die Evaluierung des DT Ansatzes, stellen wir zusätzlich einen Silizium Prototyp unserer DT-fähigen Mikroarchitektur vor, welcher in 28 nm FD-SOI CMOS hergestellt wurde. Der voll funktionsfähige Testchip beinhaltet eine geeignete Takterzeugungseinheit, die eine Zyklus-zu-Zyklus Taktanpassung über einen weiten Bereich mit feiner Granularität ermöglicht, bei Frequenzen von mehr als 1 GHz. Die Messergebnisse der Beschleunigungen und der Energiereduzierungen durch Spannungsskalierung werden zusätzlich vorgestellt.

Stichwörter: energiesparender Entwurf, Mikroarchitektur, Befehlssatzarchitektur, Befehlssatzerweiterungen, ultra-energiesparender eingebetteter Prozessor, Mikrocontroller, ASIC, VLSI Entwurf, unter-Schwellenspannung Betrieb, Mehrkernarchitektur, Sensorknoten, biomedizinische Signalverarbeitung, kryptographische Hashfunktionen, SHA-3, komprimierte Abtastung, dynamische Zeit-Margen, dynamische Zeitanalyse, dynamische Taktsignalanpassung, zeitliche Fehler, statistische Fehlerinjektion, approximierende Datenverarbeitung, OpenRISC

Contents

1 Introduction

Embedded processing has arguably evolved as the predominant form of computing over the past two decades. Any form of electronics produced today typically integrates some form of embedded processor, covering every kind of application imaginable, from consumer to industrial, over medical to automotive. While the field of embedded systems has been dominated in the past by small microcontrollers focused on simple control tasks with very little processing power, the evolution of embedded computing has today given rise to complex systems-on-chip (SoCs) which provide very high processing capabilities under the most stringent energy and cost constraints.

With the advances in ubiquitous and pervasive computing [Wei99, Sat01, HB01, ECPS02], computing and specifically data processing now occurs already in many objects around us, anywhere at anytime. In recent years this development has accelerated significantly with the conception of the internet of things (IoT) and similar concepts, which aim to not only locally connect all these objects and devices, but to interconnect them on a global scale to enable new applications and ways of computing.

A model for the internet of things is illustrated in Figure 1.1 [IBM14]. The “things” or end devices provide often sensors and/or actuators and always comprise some form of processing, performed on one or multiple embedded microprocessor(s). The devices connect typically wirelessly to a local network, which can consist of multiple layers and intermediary larger nodes, again requiring embedded processing capabilities. These local networks integrate with the internet, which allows for the end devices to be controllable via data center driven applications. The end user typically interacts with the end devices or rather with the whole application or service, which encompasses the end devices, via a personal controlling device such as a smartphone or tablet.

Wireless sensor nodes are one of the most prominent types of end devices in the IoT context. Such nodes combine sensing and processing capabilities in a constantly shrinking form factor, and are comprised of an embedded low-power processor subsystem, a sensor with an analog to digital converter, a radio, as well as a battery. A prime example for

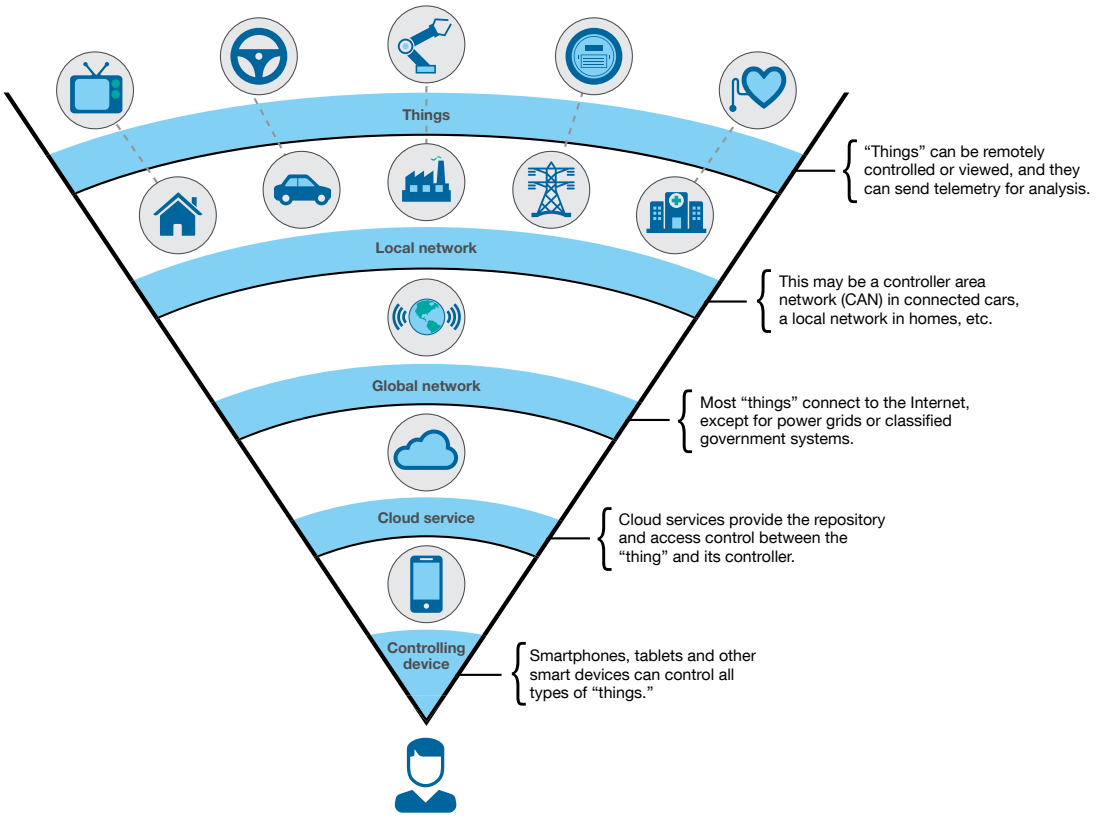


Figure 1.1 – Model for the internet of things [IBM14], with embedded microprocessors integrated as part of the end devices/"things", the intermediary nodes in the local network, and the controlling devices of the end users.

an application of wireless sensor nodes are wireless body sensor networks (WBSNs). WBSNs are very successfully employed in the domain of healthcare and fitness, where they provide a low-cost solution for continuous monitoring and logging of a person's vital parameters [RMC05, CLC+09, YY10, KMKA11]. An example for a WBSN-based system as it is used in a typical healthcare monitoring application is given in Figure 1.2 [CCHL12]. Here, the WBSN provides personal monitoring capabilities for the medical conditions of patients, which allows for example the implementation of automated warning systems, and can trigger emergency interventions in urgent cases. To allow for such a warning and detection system to operate efficiently, the microcontroller unit (MCU) in the end node has to provide appropriate capabilities for processing of the sensor data, since transmission of raw sensor data is normally too costly in terms of energy consumption [MKAV11, Dog13]. As illustrated in Figure 1.2 such a wireless sensor node can comprise a wide variety of different sensors, even with multiple channels. In this example, the temperature, blood pressure, and heartbeat of the patient are monitored via the wireless sensor node.

One of the most important aspects of such sensor nodes is that their operation is largely autonomous and battery-driven. Although there is extensive research into energy harvesting

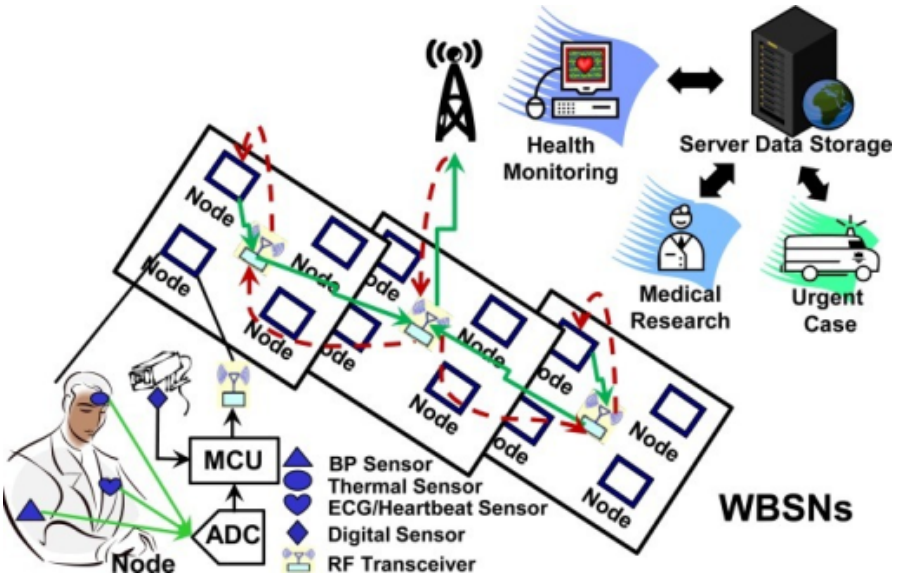


Figure 1.2 – Wireless body sensor network (WBSN) system for hospital and healthcare monitoring applications [CCHL12].

techniques for fully autonomous operation of sensing devices in general, the amount of energy and especially instantaneously available power that can be provided is overall still quite limited [PS05, MYR+08, GWZ13]. Wireless sensor nodes hence require highest energy efficiency on a complete system/device level in order to provide adequate operation time considering their limited battery capacity.

The need for deeply embedded processing is growing due to more and more complex tasks required by the applications that are performed on such devices. Moreover, especially because of the mentioned limitation in terms of data volume that can be transmitted over an integrated radio on a tight power budget, on-node (signal) processing is becoming increasingly necessary. However, significant gains in terms of energy efficiency and in turn battery lifetime can only be achieved through low-power microprocessor/microcontroller subsystems, which are executing the required signal processing algorithms.

The system diagram of a state-of-the-art embedded subsystem for IoT applications, integrating an ARM Cortex-M microprocessor is depicted in Figure 1.3 [ARM16]. The system includes, besides the processing core, a light-weight bus interconnect that provides access to memory controllers as well as to a radio and other peripherals. As an example for a microarchitectural measure to bring down energy consumption, the system uses a flash cache since any accesses that are made to the embedded flash memory are very energy-intensive. Furthermore, such systems comprise a power management unit, which allows the circuit to better operate according to the current requirements of the application and to save power by employing techniques such as clock gating or power gating for parts of the circuit which are temporarily not utilized. Another popular option for power optimization provided by a power management unit, is the adjustment of the different clock frequencies of the various components on the to their

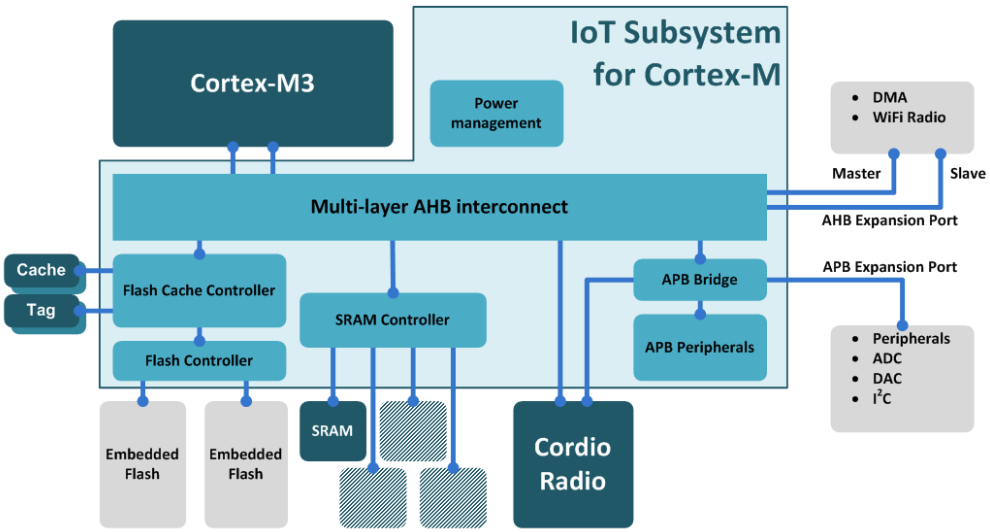


Figure 1.3 – System diagram of an exemplary state-of-the-art embedded subsystem for IoT applications, integrating a Cortex-M microprocessor [ARM16].

current throughput needs. Additionally, especially in more complex SoC designs of recent years, supply voltages can often be adjusted through the use of on-chip regulators to achieve very low power consumption.

Highest energy efficiency is however not only required for deeply embedded IoT end node type devices with their severely limited battery capacities. Also devices incorporating larger SoCs for embedded and mobile applications operate under stringent energy constraints, often with total system power budgets of a few hundred mW to one W, even for computing scenarios with high performance requirements. This is not only due to battery limitations, but also increasingly due to thermal limits that have to be met for high performance hand-held devices with very small form factors, imposing strong constraints on the employed heat management and cooling solutions [SA14, SVC15].

As a consequence, there is an increasing need for very high energy efficiency in embedded microprocessors, over the full processing performance and application spectrum. Energy efficiency has become the most important and driving design constraint in the further evolution of such systems, even over other important aspects such as silicon area or maximum achievable clock frequency. There are many layers in the computing stack of a programmable architecture, which can all contribute significantly to the power consumption of the complete system. While choosing the appropriate algorithm for the given task at hand is of course paramount, lower layers of the stack can still provide significant improvements after optimizations on the higher, more abstract layers have been exploited to their full potential.

In this thesis the focus lies on microarchitectural design techniques and optimizations, which are on the boundary between the instruction set architecture of a microprocessor — i.e., the fixed contract between the programmer/software and the hardware — and the physical

hardware implementation of the circuit. The microarchitectural improvements especially enable energy efficient and/or low-power operation, when they are combined with supply voltage scaling, a key technique for lowering the power consumption of digital circuits.

The following two sections provide an overview of the contributing factors for the power consumption in a digital complementary metal–oxide–semiconductor (CMOS) circuit, and then introduce the general benefits and challenges stemming from voltage scaling for microprocessor circuits.

1.1 CMOS Power and Energy Consumption

The power consumption in watts [W] of a circuit denotes the amount of energy in joules [J] that is dissipated per second.[1] While absolute power numbers for a given state of a system (active, idle, sleep, etc.) are useful on a full system level, note that this measure usually does not relate to the performance of the circuit since it does not reflect the amount of computational work or processing that is performed for the given power consumption. Especially from an architectural design point of view it is hence often more useful to consider the amount of energy used per computational operation or per processed data item [Kae08], which is often also referred to as the energy efficiency. Thus, in the following we focus on the different energy components that get dissipated in a CMOS circuit and relate these quantities to the power consumption P, with the following simple equation:

$$P = f_c E_c. \tag{1.1}$$

Here, f_c denotes the computation rate, which for a single-edge-triggered digital circuit is normally equivalent to the clock frequency f_{clk} at which the circuit operates, while E_c stands for the total energy dissipated per computation cycle (or rather its average over many cycles).

E_c has two main components, the active energy E_{active}, which is dissipated due to switching of the circuit, and the static energy E_{static}, which is constantly dissipated, as long as the circuit is powered, i.e., connected to a supply voltage. This means that E_{active} is only dissipated when the circuit receives a clock[2], while E_{static} is also consumed during phases where the clock is not active.

$$E_c = E_{active} + E_{static} \tag{1.2}$$

The active energy E_{active} can be split up in two components, the energy E_{ch} spent on charging

[1] The introduction to power consumption and energy dissipation in CMOS circuits of this section is based on the excellent explanations and observations in Chapter 9.1 of the textbook “Digital Integrated Circuit Design” [Kae08]. This includes most formulas.

[2] Technically, it is also possible for active energy to be dissipated when primary inputs of a circuit or module switch, even with a gated/disabled clock.

and discharging of capacitive loads, and the energy E_{cr} dissipated due to crossover currents.

$$E_{active} = E_{ch} + E_{cr} \tag{1.3}$$

The static energy E_{static} also has two parts, the leakage energy E_{leak}, and the energy E_{rr} consumed by driving of resistive loads.

$$E_{static} = E_{leak} + E_{rr} \tag{1.4}$$

Charging and discharging of capacitive loads (E_{ch}) The energy that is dissipated in the p-MOS/n-MOS transistors as thermal energy in order to either charge or discharge the capacitive load C_k of a node k is $\frac{1}{2}C_k V_{dd}^2$, with the supply voltage at V_{dd}. Together with the node activity α_k, which indicates how many times per cycle a node k on average switches from one logic state to the other, we derive E_{ch} for the complete circuit with K different nodes as:

$$E_{ch} = V_{dd}^2 \sum_{k=1}^{K} \frac{\alpha_k}{2} C_k. \tag{1.5}$$

We can observe from (1.5) that E_{ch} can be either reduced by decreasing the total amount of capacitance over all nodes K (e.g., via proper gate sizing), or by lowering the activities α_k, for example through clock gating or by avoiding as many glitches as possible during the settling phases of nodes. More important however, E_{ch} can be significantly reduced by lowering V_{dd}, due to the quadratic dependence of E_{ch} on V_{dd}. Note that this is the key observation which makes supply voltage down-scaling such an effective technique for power reduction [WCC06].

Crossover currents (E_{cr}) During the switching phase of the p- and n-network of a CMOS gate, energy is additionally dissipated due to current that flows directly from supply to ground while for a short time both the p- and n-paths are partially conducting. The exact energy that is dissipated due to crossover/short-circuit currents depends on numerous factors. However, if transition times of gate inputs can be kept as short as possible, the contribution of E_{cr} towards the total active energy is small. Furthermore, E_{cr} is also decreased by a supply voltage reduction, with a cubic dependency on V_{dd}.

Leakage currents (E_{leak}) The major part of static power consumption typically stems from leakage currents that occur due to the imperfections of the p- and n-MOS devices in a CMOS process. The contributors to E_{leak} are sub-threshold conduction of channels that are turned off, gate leakage, and leakage currents through multiple combinations of reverse-biased junctions (drain/source with bulk, and well with well/substrate). Since leakage energy is dissipated for all devices in the circuit, the total leakage energy is linearly dependent on the number of transistors (or more specifically the sum of their widths) or roughly on the overall utilized silicon area. Leakage has become an increasingly larger part of the total energy/power

consumption of circuits in more advanced bulk CMOS technology nodes (sub-65 nm), mainly due to reduced threshold voltages, which create more issues for the channel control in its off state. However, to address these problems, an increasing number of different process options for a single technology node are offered by foundries, allowing to better optimize the device performance towards the circuit requirements (low power, low leakage, high speed, etc.). In relation to voltage scaling, leakage energy becomes increasingly relevant for circuit operation near and especially below the threshold voltage (i.e., for $V_{dd} \lesssim V_T$). This is due to the significantly reduced active energy dissipation at these operating points, which requires the careful management of leakage energy to achieve further energy reductions. Leakage energy exponentially depends both on the supply voltage (lower V_{dd} reduces leakage power) and on the threshold voltage (higher V_T reduces leakage).

Driving of resistive loads (E_{rr}) In theory, there should be no static currents flowing in a CMOS circuit that is not switching (besides the unavoidable leakage currents), since by design there should be no direct static paths between supply and ground. However, many exceptions to this exist, involving: amplifiers (e.g., contained in memories), clock generation and conditioning subcircuits, voltage converters and regulators, on-chip pull-ups/downs, and electrostatic discharge (ESD) protection structures. For a more extensive list of examples, the reader is referred to [Kae08]. Much like E_{ch}, which is dissipated by the charging of capacitive loads, also E_{rr} has a quadratic dependence on V_{dd}.

1.2 Supply Voltage Scaling

Reduction of the supply voltage of a CMOS circuit has the potential to provide significant energy and power savings, as indicated by (1.5). The evolution of submicron technology nodes has hence initially delivered substantial power savings, due to technology-driven reduction of the nominal supply voltage (mainly to avoid dielectric breakdown of thinner gate oxides [Kae08]), besides other aspects. However, about a decade ago, in deep submicron technologies below 90 nm, this trend of nominal V_{dd} reduction began to level off to around 1 V (at 65/40 nm).[3] Since then we have seen an increasing usage of many different forms of voltage scaling techniques, which aim at operating circuits below *nominal* V_{dd} levels, down to the near- or even sub-threshold level to further reduce energy consumption. Especially the concept of dynamic voltage frequency scaling (DVFS) has been widely adopted by industry, which operates the circuit at fixed pairs of voltage and frequency tuned for specific usage scenarios. In more forward looking proposals DVFS is used in a closed-loop fashion, dynamically adjusting voltage and/or frequency according to detected timing violations [EKD+03, TBW+09].

However, dynamic voltage scaling has its limitations [ZBSF04] and energy gains from supply voltage scaling do not come for free since reduced V_{dd} increases the gate delays of the circuit,

[3] The latest non-bulk CMOS nodes, using FinFET devices, seem to further push these limits again, with standard operation at 0.7 V in 14 nm [NAA+14].

therefore reducing its maximum clock frequency. In the operating region above the threshold voltage V_T, the proportionality of a CMOS gate delay t_{pd} (propagation delay) [Kae08] with respect to the supply voltage V_{dd} is given by:

$$t_{pd} \propto \frac{C_k V_{dd}}{(V_{dd} - V_T)^{\alpha}}. \tag{1.6}$$

Here, C_k is again the node capacitance which the output of the gate has to drive, while α is an empirical constant greater than 1, which depends on the specifics of the technology that is employed. For deep submicron technologies α is typically close to 1, due to velocity saturation. With C_k and V_T fixed for a given circuit implementation in a specific technology, and $\alpha \approx 1$, we can consider the proportionality of the clock frequency $f_{clk} = \frac{1}{t_{pd}}$ as a function of the supply voltage V_{dd}:[4]

$$f_{clk} \propto \frac{1}{C_k} - \frac{V_T}{C_k} \cdot \frac{1}{V_{dd}}. \tag{1.7}$$

We can observe that the clock frequency is linearly dependent on the supply voltage, since the right-hand side of (1.7) grows linearly with increasing V_{dd} (with the subtrahend of the difference decreasing linearly, while the minuend remains constant). Furthermore, when scaling of the supply voltage continues below the threshold voltage, sub-threshold currents drive transistor operation, causing (1.6) and (1.7) to be not applicable anymore for this operation region. For the sub-threshold regime the delay starts to become exponentially dependent on the supply voltage, which creates a severe performance degradation for any additional savings that should be achieved in terms of power consumption.

A qualitative overview and comparison of circuit power, frequency, and energy per operation as a function of supply voltage is given in Figure 1.4 [KKT13].[5] The figure illustrates how performance in terms of maximum operating frequency is degraded by 5-10x when scaling the supply voltage from the nominal operating point ($V_{DD_{STV}}$) to an operating point that is near the threshold voltage ($V_{DD_{NTV}}$). At the same time, power consumption is reduced by 10-50x due to two factors: the circuit is operated at a lower frequency, so the amount of switching per second is reduced; additionally, the circuit operates at a lower supply voltage, which especially reduces active power even further according to (1.5). As a result the energy efficiency of the circuit improves by 2-5x when it is operated at near-threshold, compared to the nominal operating point in the above-threshold region. As can be seen in the frequency plot, the performance degradation starts to become exponential near the threshold voltage, actually causing an increase in the energy per operation after some point in the sub-threshold region. This is due to the leakage energy which starts to dominate over the dissipated active energy, when computation cycles become exponentially longer. The operating point with the lowest possible energy per operation, i.e., the best energy efficiency, is called the energy minimum

[4] For simplicity, the delay t_{pd} is here assumed to be the total delay of all gates on the critical path, while C_k denotes the overall capacitance.

[5] The figure is based on data from [DWB+10] and [JKY+12].

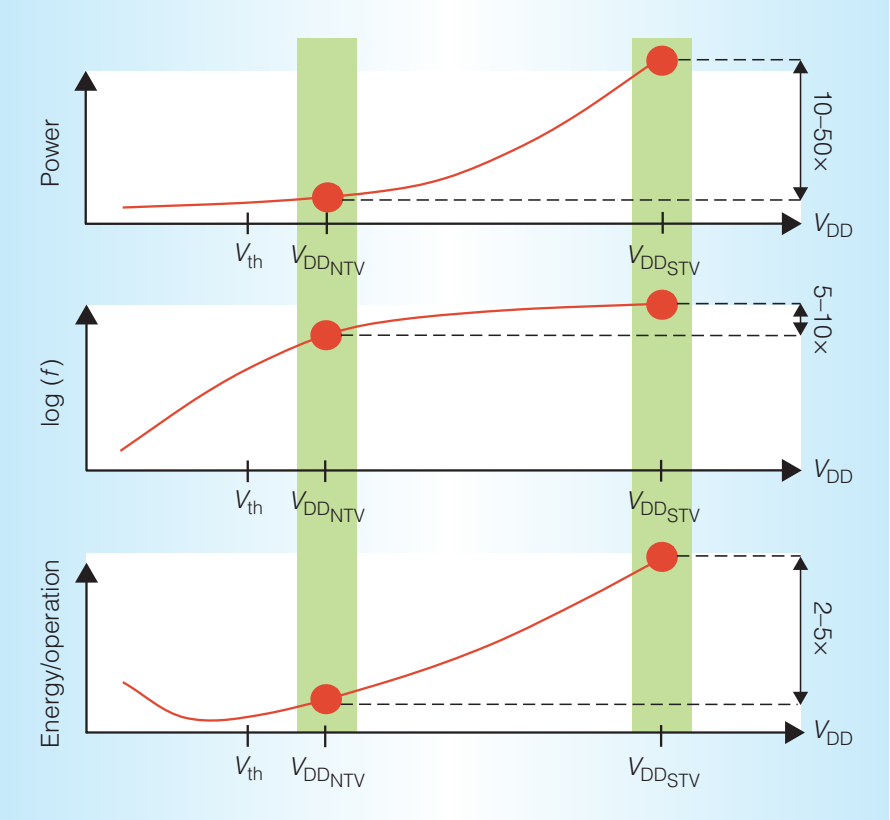


Figure 1.4 – Qualitative overview for power, frequency (f), and energy per operation as a function of supply voltage (V_{DD}) [KKT13, DWB+10, JKY+12], highlighting the gains from supply voltage scaling, and the induced performance degradation. V_{th} denotes the threshold voltage, $V_{DD_{NTV}}$ denotes a near-threshold operating voltage, while $V_{DD_{STV}}$ denotes the classical super/above-threshold operating voltage.

voltage (EMV) point. The position of the EMV relative to the threshold voltage depends on many factors, such as the employed CMOS technology and process option, but of course is also highly dependent on the architecture and circuit structure of a design, and can range from deep sub-threshold to near-threshold.

In addition to the strong performance degradation, low-voltage operation in the near-threshold regime and below gives rise to a set of problems regarding increased parametric and performance variation [DWB+10, KKT13]. For example, at near-threshold delay uncertainty due to global process variation can increase from 30% at nominal voltage (1 V) to 400% at 400 mV supply. Together with the more severe sensitivity to temperature and supply voltage noise effects, the total performance uncertainty rises to 20x, compared to only ≈1.5x at nominal voltage [DWB+10]. Moreover, near- and sub-threshold computing poses significant challenges regarding increased functional failures. While logic circuits are typically able to reach EMV operation without failures, aggressively scaled embedded memory structures, especially static random-access memory (SRAM), impose stronger limitations on voltage scaling [CC06], giving

rise to a host of new circuit design approaches for sub-threshold SRAMs [VC08, ZHBS08] and other types of embedded memories [Mei14].

Besides these circuit level challenges, which need to be addressed for effective low-power design based on aggressive supply voltage scaling, the compensation of performance losses (delay increase) remains essential. The promised gains in energy efficiency can only be achieved for real-world embedded systems with fixed computational throughput constraints, if the induced performance degradation can be recovered through architectural techniques. The key motivation for the microarchitectural design techniques proposed in this thesis is consequently to enable voltage scaling for low-power operation, while retaining the computational performance of the microprocessor at acceptable levels.

To conclude this introduction to supply voltage scaling, Figure 1.5 and Figure 1.6 provide measured, absolute numbers on frequency, power, and energy per cycle, for two published state-of-the-art research designs [MSG+16, JKY+12], to complement and underline the presented theoretical explanations and observations of this chapter. Both designs are microprocessor-based SoCs, which are capable of operation over a wide voltage range, from above- to sub-threshold, but aimed at different applications with different processing performance requirements.

Figure 1.5 [MSG+16] showcases one of the most energy efficient complete microprocessor/-SoC designs in the area of wireless sensor nodes (WSNs). The processing subsystem is implemented in low-leakage 65 nm CMOS, and includes integrated voltage regulators for direct battery operation, as well as 10T SRAM for reliable operation in the sub-threshold regime. As can be seen in Figure 1.5, the operating range of the 32-bit ARM Cortex-M0+ processor subsystem is extremely wide, ranging from nominal voltage operation at 1.2 V with 66 MHz and 5.9 mW, to deep sub-threshold operation at only 250 mV with 27 kHz and 850 nW total power consumption. The EMV is around 0.39 V with a minimum energy consumption of 11.7 pJ/cycle at 688 kHz. Since WSNs often operate with short duty cycles, standby power reduction is critical. Consequently, the authors of [MSG+16] have applied many dedicated circuit level optimizations to reduce retention power to 80 nW (for the CPU + 4 kB SRAM). Another interesting aspect that can be observed in Figure 1.5 is the change in the EMV, depending on dynamic factors such as the core temperature or even the executed application. Moreover, we can see that, as was indicated earlier in Figure 1.4, a reduction in energy per cycle of around 4-5x is achieved when scaling the voltage from nominal above-threshold operation to near-threshold, with up to 8x gains when scaling further until the EMV is reached.

In contrast to the just presented sensor node processing subsystem, Figure 1.6 [JKY+12] provides details on a general purpose embedded processor aimed at near-threshold computing, which can operate at a more than one order of magnitude higher clock frequency, while additionally providing significantly more compute performance per clock cycle. The design is a wide-operating-range IA-32 processor (x86, Intel Pentium class) fabricated in 32 nm CMOS, including 10T SRAM and other optimized circuits for robust and reliable ultra-low voltage operation. This includes for example a clock distribution network incorporating programmable

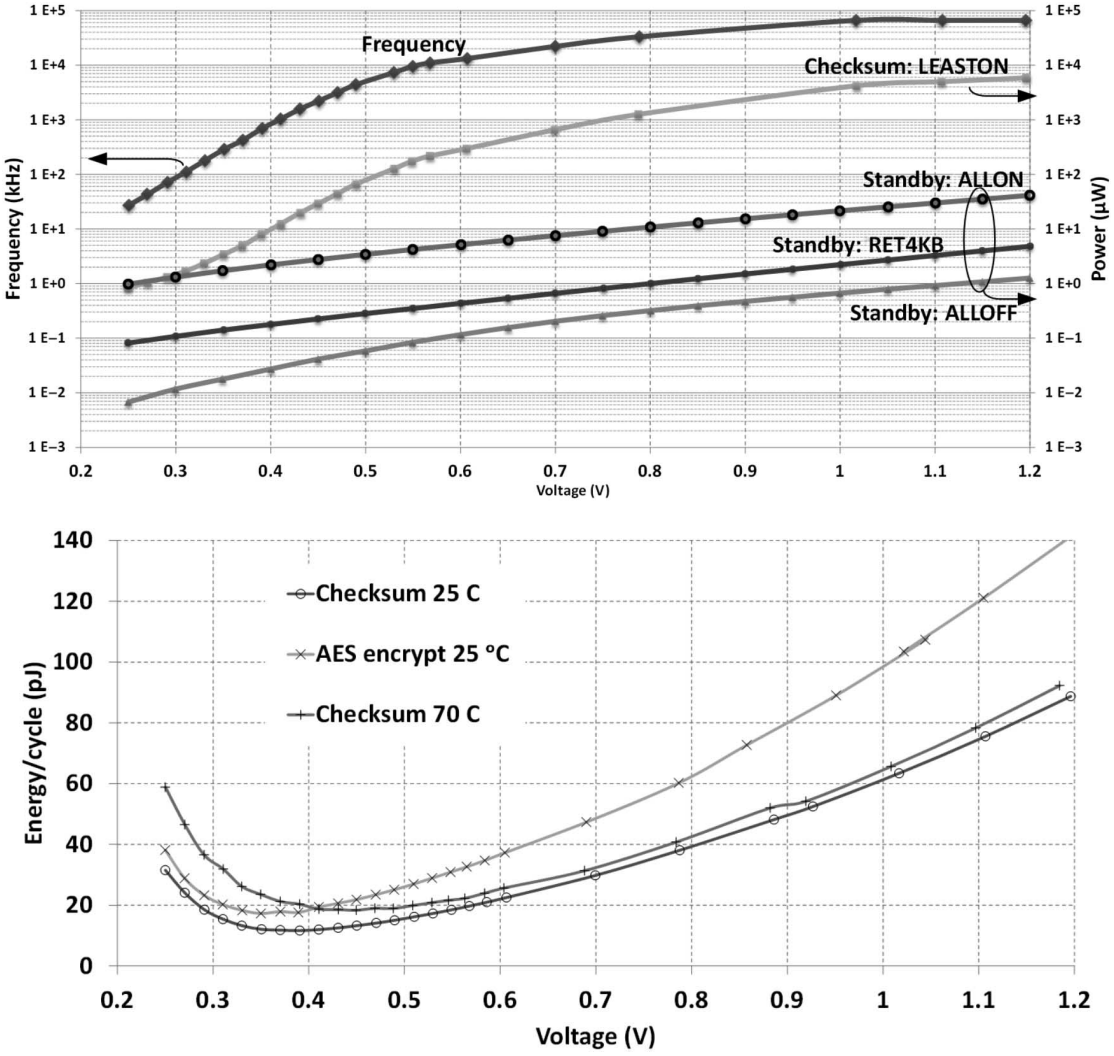


Figure 1.5 – Measured frequency, power, and energy per cycle versus supply voltage, of a sub-threshold ARM Cortex-M0+ subsystem in 65 nm CMOS for WSN applications; reproduced from [MSG+16]. Active power is indicated by the power curve labeled *Checksum: LEASTON*, while the other three curves show standby power for different modes.

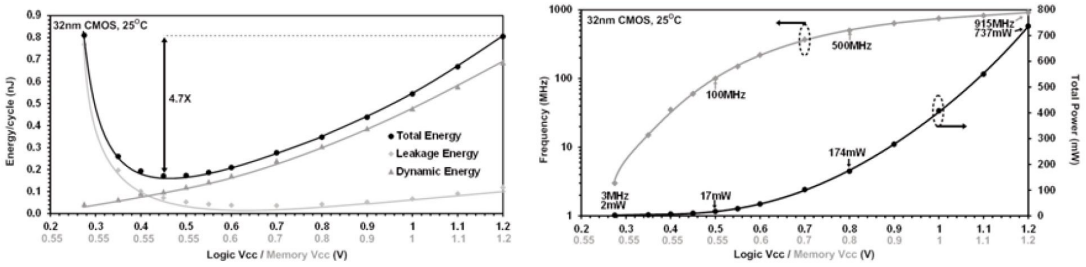


Figure 1.6 – Measured frequency, power, and energy per cycle versus supply voltage, of a wide-operating-range IA-32 processor (x86, Intel Pentium class) in 32nm CMOS for near-threshold computing; reproduced from [JKY+12].

delay buffers for mitigation of skew variations over the full voltage range. The 2x-superscalar 32-bit IA-32 CPU with FPU and branch prediction scales from a maximum supply voltage of 1.2 V[6] with 915 MHz and 737 mW, to 280 mV with 3 MHz and 2 mW total power consumption (with memories at 550 mV, due to scaling limitations). Here, the EMV is reached in the near-threshold region at 0.45 V, with an energy consumption of 170 pJ/cycle. As Figure 1.6 shows, this constitutes a 4.7x improvement in energy efficiency, over operation at 1.2 V. The plot on the left-hand side also illustrates well how the leakage energy quickly becomes dominant beyond the EMV (at which it already consumes 42% of the total energy), exploding exponentially in magnitude as the supply voltage decreases.

[6]The nominal voltage for this 32 nm node is 1.0 V for all three process options (HP/SP/LP) [JAB+09]. The employed low-power (LP) process option additionally provides a maximum of 1.2 V.

1.3 Contributions

The contributions of this thesis towards microarchitectural low-power design techniques for embedded microprocessors are focused in two main areas: On the one hand, new types of light-weight instruction set extensions are introduced and combined with ultra-low-power core design and multi-core architectures, covering the spectrum from deeply embedded cores with 10s of kHz to 10s of MHz clock frequency to general purpose microcontroller architectures with 10s of MHz to 100s of MHz clock frequency. On the other hand, we define the concept of dynamic timing margins in embedded microprocessors and we characterize and exploit these margins via instruction-based dynamic clock adjustment to obtain performance and energy gains. As a proof of concept, we demonstrate a silicon prototype (supporting dynamic clock adjustment) focused on general purpose embedded processing with 100s of MHz to 1 GHz clock frequency. Figure 1.7 provides an overview of the specific contributions of this thesis, categorized by microprocessor type and performance range. Further, Figure 1.7 illustrates the compute/design stack (from the software level down to the physical design level), and indicates which topics span which parts of the stack.

Specifically, the main contributions of this PhD dissertation are summarized as follows:

Light-Weight Instruction Set Extensions & Ultra-Low-Power Core Design

- The performance, resource, and energy requirements of cryptographic applications often pose challenges for embedded microcontroller architectures, hindering the seamless adoption and integration of cryptographic algorithms on such platforms. This thesis proposes three different general categories of light-weight instruction set extensions, targeted at cryptographic hash functions. Especially the category of instruction set extensions which is based on finite state machines for address generation provides a promising new way of addressing this problem, with very low hardware overhead.
- As part of the evaluation of the proposed instruction set extension types, we design and implement sets of suitable light-weight extensions for all five SHA-3 final round candidate algorithms. These extensions are designed and evaluated for a popular 16-bit microcontroller architecture (PIC24), providing insight into the real-world applicability of the approach.
- A custom-designed 16-bit core for ultra-low-power applications, named TamaRISC, is presented. Its microarchitecture as well as instruction set architecture are designed from scratch, with the aim of instruction set simplicity, while at the same time providing useful features for deeply embedded signal processing applications, especially in the biomedical domain. Besides a full hardware implementation, a software tool-chain including a C compiler and support for a real-time operating system are provided.
- TamaRISC is combined with the concept of light-weight instruction set extensions to create a processor system for compressed sensing, which operates in the sub-threshold

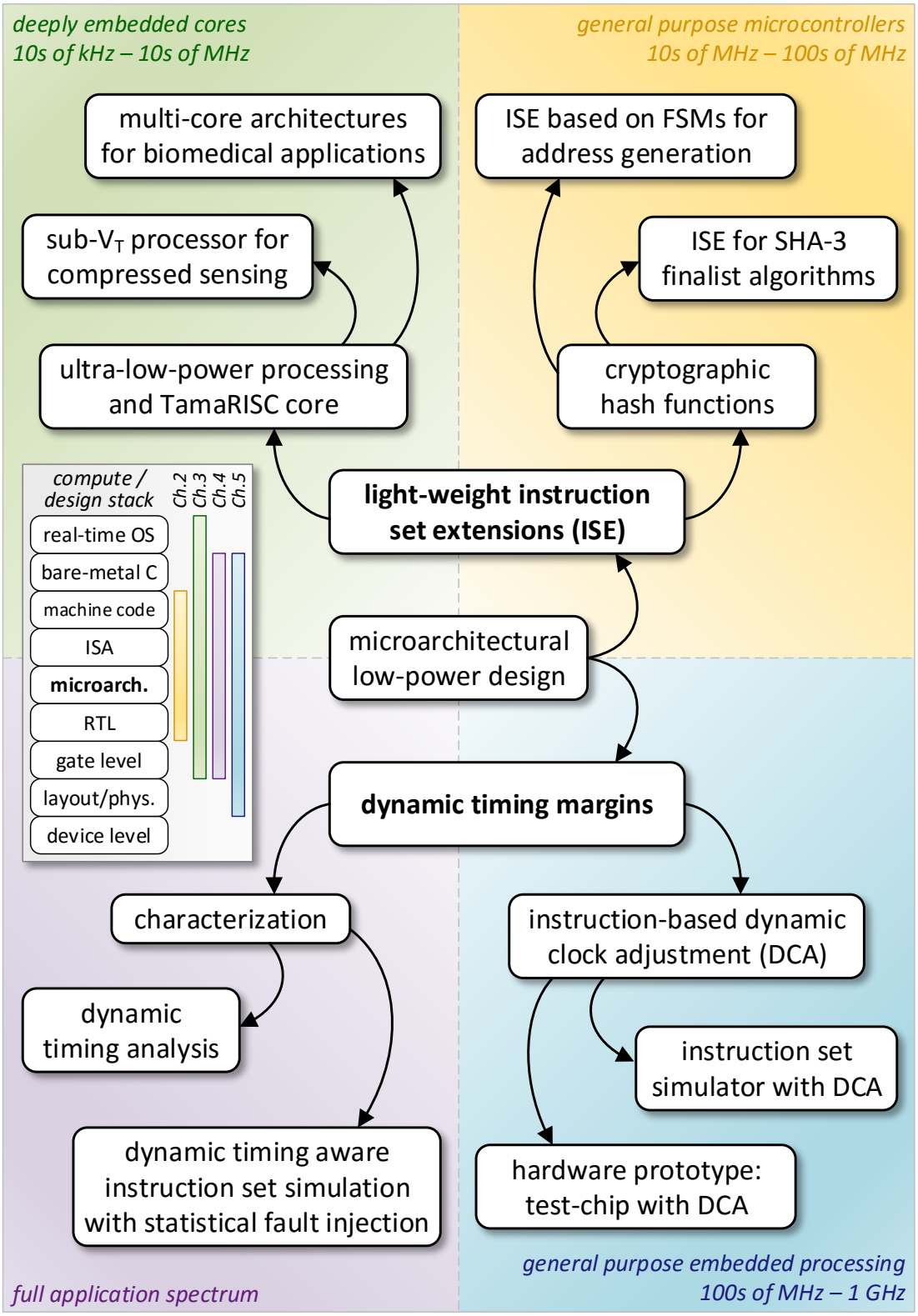


Figure 1.7 – Overview, structure, and classification of the covered topics and contributions of this thesis in microarchitectural low-power design for embedded microprocessors.

regime. We demonstrate with this optimized processor core how instruction set extensions are key to enable further voltage scaling while maintaining the required processing performance. The processor system embeds existing components, such as special sub-threshold capable memories, to enable electrocardiogram signal compression with nW power consumption.

- In the context of multi-core processing systems for biomedical and other ultra-low-power embedded applications, TamaRISC is employed to build SIMD-like architectures. Such multi-core systems strive to provide highest possible energy efficiency, outperforming single-core architectures for scenarios with medium to high work loads. This thesis provides with TamaRISC the low-power processing element for these architectures, and additionally introduces microarchitectural design optimizations for the memory organization in such multi-core systems. Specifically, we introduce core enhancements that enable configurable data memory mapping for private and shared access, instruction broadcast, and synchronized code execution with check points.

Dynamic Timing Margins

- Dynamic timing margins in digital circuits are a promising opportunity for regaining performance and energy efficiency, especially in the context of the increasing trend of over-design and over-margining of digital systems. In this thesis, we systematically define the concept of dynamic timing margins. Furthermore, a dynamic timing analysis methodology and tool are developed, which allows the characterization of dynamic timing margins in digital circuits on gate level, with extensions to specifically enable the detailed characterization of processor based systems and their pipelines.

- The results of the dynamic timing analysis flow are used to build an instruction set simulator with statistical fault injection. The statistical nature of the collected timing information allows to significantly increase the simulation accuracy over the state of the art regarding the high-level simulation of the behavior of a processor during frequency and/or voltage over-scaling, which has recently been proposed in the context of approximate computing to push energy efficiency to the maximum possible level. A dynamic timing aware simulator for the evaluation of application behavior under timing errors is developed in this thesis by combining an existing fault-injection framework with our dynamic timing analysis flow. This approach to accurate simulation of a processor allows the efficient analysis of the circuit in the regime between fully error-free operation and total circuit failure, facilitating new ways of circuit design under the approximate computing paradigm.

- One of the key contributions of this dissertation is the concept of instruction-based dynamic clock adjustment. The concept is based on our findings from dynamic timing analysis of a microprocessor and leverages the available dynamic timing margins to adjust the clock frequency of a processor core on a cycle-by-cycle basis. To enable

proper analysis of potential gains from this technique, the results of dynamic timing analysis are again combined with an instruction set simulator, this time under a worst-case dynamic timing scenario which should guarantee fully error-free operation. The simulator allows to predict the application dependent speedups, achievable by a 32-bit OpenRISC processor design tailored for dynamic clock adjustment.

- The feasibility of the proposed instruction-based dynamic clock adjustment concept is shown by the design and fabrication of a full-featured hardware prototype in an advanced 28 nm FD-SOI CMOS technology. This test-chip is — to the best of the author's knowledge — the first silicon implementation of a microprocessor operated by a dynamic clock, which adapts its frequency with a very fine granularity on a cycle-by-cycle level, based on the instruction types currently in flight in the pipeline. Extensive measurement results for this dynamically clocked 32-bit 6-stage OpenRISC processor are presented, demonstrating the achievable speedups and power savings.

1.4 Thesis Outline

As indicated in Section 1.3, this thesis covers two different areas in microarchitectural low-power design. Ultra-low-power design together with light-weight instruction set extensions for small microcontroller-type cores are covered in Chapter 2 and Chapter 3, while the topic of dynamic timing margins is discussed in Chapter 4 and Chapter 5. In more detail, the remainder of the thesis is structured as follows.

Chapter 2 covers microcontroller architecture enhancements for cryptographic applications, in the form of light-weight instruction set extensions. First, the target algorithm class of cryptographic hash functions is introduced, followed by a description of the considered set of specific algorithms. The PIC24 microcontroller architecture is then presented as the chosen reference core, and the baseline performance of the SHA-3 algorithms on this architecture is established. The identified bottlenecks in the baseline implementations are then used as motivation for the introduction of three general classes of instruction set extensions for cryptographic hash functions. Finally, these classes are applied to the five SHA-3 algorithms, resulting in the design of algorithm-specific extensions, as well as the implementation of the modified cores with extended microarchitectures. The chapter is concluded by a discussion of the performance results, regarding software as well as hardware aspects.

The focus of Chapter 3 lies on ultra-low-power processing and different aspects of microarchitectural design that enable it. After presenting the context of ultra-low-power applications, our custom-designed 16-bit core for such applications, named TamaRISC, is introduced. This is followed by a section on an ultra-low-power application-specific processor for compressed sensing, which is based on TamaRISC, and can be operated in the sub-threshold regime. The light-weight instruction set extension which enables the extremely low power consumption of the system is presented, together with an extensive power and performance characterization of the circuit. The chapter is concluded by an overview of single- vs multi-core processing for

ultra-low-power sensing platforms, followed by a description of the developed microarchitectural techniques for efficient memory organization in such multi-core architectures.

The concept of dynamic timing margins is introduced and explored in detail in Chapter 4. We begin with an overview of general timing margins in digital circuits, including those induced due to guardbanding employed to counter static and dynamic variations. This is followed by the introduction of the orthogonal concept of dynamic timing margins. Our dynamic timing analysis approach is then presented, and we show how it can be applied to microprocessor architectures. Moreover, dynamic timing analysis results for a case study on a general purpose 32-bit OpenRISC processor are employed to construct an instruction set simulator with statistical fault injection. In the context of this simulator we introduce a new physically motivated timing error model for high-level instruction set simulation based on stochastic timing information. Finally, the application behavior under frequency- and/or voltage-over-scaling for a set of common algorithms is analyzed using this new simulation approach.

In Chapter 5 we introduce an instruction-dependent dynamic clock adjustment technique for microprocessors. Together with the dynamic timing analysis we present a full design flow, which allows the efficient evaluation of the introduced technique on an instruction set simulator. The main part of the chapter then focuses on the silicon prototype which applies dynamic clock adjustment to a modified OpenRISC core. The hardware architecture of the test-chip is detailed, together with the full microarchitecture of the core and the dynamic clocking mechanism which enables this approach on a real design. The chapter finally provides measurement results for the test-chip fabricated in a 28 nm FD-SOI CMOS technology. The measured performance and energy efficiency gains, enabled by dynamic clock adjustment, are presented and discussed for multiple different dies.

Chapter 6 provides concluding remarks for the presented concepts and results, as well as an outlook regarding possible future work.

1.5 Selected Publications

This thesis is largely based on the following publications (in case of some, only in part). A complete list of the author's book chapters, conference papers, journal articles, and other publications can be found in the List of Publications as part of the back matter.

Chapter 2
Microcontroller Architecture Enhancements for Cryptographic Applications

Jeremy Constantin, Andreas Burg and Frank Gürkaynak. *Instruction Set Extensions for Cryptographic Hash Functions on a Microcontroller Architecture.* 23rd IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP), Delft, The Netherlands, July 9-11, 2012.

Jeremy Constantin, Andreas Burg and Frank Gürkaynak. *Investigating the Potential of Custom Instruction Set Extensions for SHA-3 Candidates on a 16-bit Microcontroller Architecture.* Cryptology ePrint Archive, February, 2012.

Chapter 3
Microprocessor Design for Deeply Embedded Ultra-Low-Power Processing

Jeremy Constantin, Ahmed Dogan, Oskar Andersson, Pascal Meinerzhagen, Joachim Rodrigues, David Atienza Alonso and Andreas Burg. *An Ultra-Low-Power Application-Specific Processor with Sub-V_T Memories for Compressed Sensing.* VLSI-SoC: From Algorithms to Circuits and System-on-Chip Design, pp. 88-106, IFIP Advances in Information and Communication Technology 418, Springer, 2013.

Jeremy Constantin, Ahmed Dogan, Oskar Andersson, Pascal Meinerzhagen, Joachim Rodrigues, David Atienza Alonso and Andreas Burg. *TamaRISC-CS: An Ultra-Low-Power Application-Specific Processor for Compressed Sensing* [best paper nominee]. 20th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), Santa Cruz, California, USA, October 7-10, 2012.

Ahmed Dogan, Jeremy Constantin, Martino Ruggiero, Andreas Burg and David Atienza Alonso. *Multi-Core Architecture Design for Ultra-Low-Power Wearable Health Monitoring Systems.* 2012 IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, Germany, March 12-16, 2012.

Ahmed Dogan, Jeremy Constantin, David Atienza Alonso, Andreas Burg and Luca Benini. *Low-power Processor Architecture Exploration for Online Biomedical Signal Analysis.* IET Circuits, Devices & Systems, vol. 6, num. 5, pp. 279-286, September, 2012.

Ahmed Dogan, Ruben Braojos Lopez, Jeremy Constantin, Giovanni Ansaloni, Andreas Burg and David Atienza Alonso. *Synchronizing Code Execution on Ultra-Low-Power Embedded Multi-Channel Signal Analysis Platforms.* 2013 IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, March 18-22, 2013.

Chapter 4
Dynamic Timing Margins in Embedded Microprocessors

Jeremy Constantin, Lai Wang, Georgios Karakonstantis, Anupam Chattopadhyay and Andreas Burg. *Exploiting Dynamic Timing Margins in Microprocessors for Frequency-Over-Scaling with Instruction-Based Clock Adjustment.* 2015 IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, March 9-13, 2015.

Jeremy Constantin, Zheng Wang, Georgios Karakonstantis, Anupam Chattopadhyay and Andreas Burg. *Statistical Fault Injection for Impact-Evaluation of Timing Errors on Application Performance.* 53rd ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, Texas, USA, June 5-9, 2016.

Chapter 5
A Microprocessor with Cycle-By-Cycle Dynamic Clock Adjustment

Jeremy Constantin, Lai Wang, Georgios Karakonstantis, Anupam Chattopadhyay and Andreas Burg. *Exploiting Dynamic Timing Margins in Microprocessors for Frequency-Over-Scaling with Instruction-Based Clock Adjustment.* 2015 IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, March 9-13, 2015.

Jeremy Constantin, Andrea Bonetti, Adam Teman, Christoph Müller, Lorenz Schmid and Andreas Burg. *DynOR: A 32-bit Microprocessor in 28 nm FD-SOI with Cycle-By-Cycle Dynamic Clock Adjustment.* 42nd IEEE European Solid-State Circuits Conference (ESSCIRC), Lausanne, Switzerland, September 12-15, 2016.

1.6 Third-Party Contributions

In this section, I list all direct third-party contributions to the work presented in this thesis.

I worked in close collaboration with my colleague Ahmed Dogan for the work on TamaRISC-CS (Section 3.2) and the use of TamaRISC in multi-core processing architectures (Section 3.4). Ahmed did the backend design of TamaRISC-CS and performed the characterization using the sub-threshold model of [ARLO12] with the support of Oskar Andersson. The RTL code for the sub-threshold capable SCMs for the TamaRISC-CS design were provided by Pascal Meinerzhagen. Ahmed moreover implemented all core-external RTL modifications for the multi-core architectures incorporating the TamaRISC core as processing elements and did the front- & backend design and evaluation of the complete multi-core architectures with efficient memory organization (as part of his PhD thesis [Dog13]).[7]

Lai Wang developed the OpenRISC LISA model (based on an initial version by Anupam Chattopadhyay) and ISS used for the evaluation of the dynamic clock adjustment concept (Section 5.2). The model was additionally used together with a fault injection extension for the work on statistical fault injection (Section 4.3). The baseline fault injection extension source code was provided by Zheng Wang who also contributed initial parts of the necessary adaptations for the OpenRISC model, including some of the fault injection framework feature extensions.

The implementation and evaluation of the DynOR test-chip (Section 5.4) in 28 nm FD-SOI was a multi-person effort for many months. Andrea Bonetti contributed the clock generation unit (front- & backend integration), as well as support for various backend tasks and final signoff of the chip. Moreover, he later provided measurements for the clock generation unit. Adam Teman lead and oversaw the backend efforts for DynOR, providing many valuable automated

[7] I later implemented an enhanced version of the MMUs enabling configurable data memory mapping (Section 3.4.2.1) as a small part of the PULPv2 architecture [RPL+16]. The design of the overall PULP architecture and the corresponding ASIC(s) were done at University of Bologna and ETH Zürich by Davide Rossi et al. [RPL+16].

solutions for the full backend flow in the employed 28 nm FD-SOI technology. Christoph Müller additionally contributed significantly to the final backend design, with floor-planing, clock tree design, and layout fixes, besides many other backend tasks. He also provided support for final chip signoff. Lorenz Schmid designed two revisions of a custom validation PCB for DynOR, which enabled me the setup of a highly automated test flow and environment. He furthermore assisted with systematic measurements of many of the DynOR samples.

2 Microcontroller Architecture Enhancements for Cryptographic Applications

In the last decade it has become clear that an increasing number of connected embedded devices have already been deployed and will continue to be deployed in our direct environment. This growth will only accelerate with the rise of the internet of things (IoT), enabled by better energy efficiency, higher circuit integration densities, and shrinking costs for these embedded devices.

One of the main purposes of these connected devices is the collection and processing of sensor data. This sensor data comes from a multitude of sources — public and private — such as from environmental monitoring, manufacturing environments, public infrastructure, home automation systems, or health care systems often in the form of wireless body sensor networks (WBSNs). It is therefore imperative that these interconnected devices and IoT infrastructures are engineered from the ground up with security as well as privacy concerns in mind [ACH15, MYAZ15]. Security is essential in many scenarios to guarantee integrity and authenticity of the communication and processed data to prevent manipulation of the surrounding infrastructure and devices, which in case of deliberate tampering can for example in a health care environment even result in serious harm to patients. With more and more data being aggregated, privacy is another key aspect that must be addressed. The guarantee for privacy provides the necessary confidentiality for information collected about us, our daily activities, and our immediate environment in which we live. Only with sufficient trust in security and privacy will a real widespread integration of such devices in our society be possible. Unfortunately, research has shown that relatively simple embedded devices are especially prone to attacks on security and privacy [ZG13, Gra15, KSV+16, Hew15].

There is hence a strong need for cryptographic applications in deeply embedded systems. The challenge to fulfill this need, is to provide such cryptographic applications that are often computationally intensive, on embedded systems which are heavily energy constrained. A key aspect to achieve this goal is the efficient implementation of cryptographic algorithms on typical embedded processing architectures, which provide the flexibility (regarding software programmability) required by the higher level applications utilizing the cryptographic functionality to provide adequate security and privacy.

In the case of embedded systems with constant high throughput requirements regarding the processed cryptographic data (multiple megabit per second (Mbps)), dedicated accelerator circuits can be considered for the specific cryptographic task to provide the best possible energy efficiency, given the high data rate. However, for moderate throughput requirements, as it is often the case in deeply embedded IoT applications, acceleration of cryptographic software running on an embedded microprocessor or microcontroller can be a more suitable approach. As we show in this chapter, acceleration of cryptographic algorithms through instruction set extensions for an existing microcontroller architecture can provide a viable solution to help meeting stringent energy efficiency requirements.

This chapter starts with an introduction on cryptographic hash functions, followed by an implementation-centric description of selected algorithms for the computation of cryptographic hash functions in Section 2.1. The microcontroller architecture used in this study is then presented in Section 2.2, together with a performance evaluation of the selected algorithms on this architecture in Section 2.4, based on the performance metrics given in Section 2.3. Possible architecture enhancements through instruction set extensions are then discussed in Section 2.5, providing insight on three general types of extensions that are specifically applicable to cryptographic hash functions. This analysis is followed in the same section by a detailed description of the designed instruction set extensions for the different algorithms. Finally, the improved performance results of all algorithms are analyzed in Section 2.6.

2.1 Applications: SHA-3 Selection Competition Algorithms

In the following, the cryptographic algorithms considered for this study are introduced in more detail. The study focuses on 5 different cryptographic hash functions from the Secure Hash Algorithm-3 (SHA-3) standard selection process. Cryptographic hash functions play an essential part in providing integrity for transmitted data, and are used as the building blocks of message authentication algorithms.

In 2007, the U.S. National Institute of Standards and Technology (NIST) started a public competition aiming at the selection of a new standard for cryptographic hashing [NIS07]. The cryptographic community was asked to propose new hash functions and to evaluate the security level of other candidates. In 2008, 51 functions were accepted to the first round, the second round (2009) reduced this number to 14, and for the final, third round (2010) the field was further reduced to five candidates: BLAKE, Grøstl, JH, Keccak, and Skein. For the competition, NIST stated that the selection for the final round has been done with the factor of diversity in mind, which makes this particular algorithm subset interesting, as it covers various different approaches for the task of cryptographic message digest calculation (hashing). Following the third SHA-3 candidate conference, the winner algorithm Keccak was announced by NIST in 2012. Since 2015 SHA-3 is an official NIST hashing standard, as defined by FIPS 202 [NIS15b].

Applications of the SHA-3 standard range from multi-gigabit data transmission protocols with

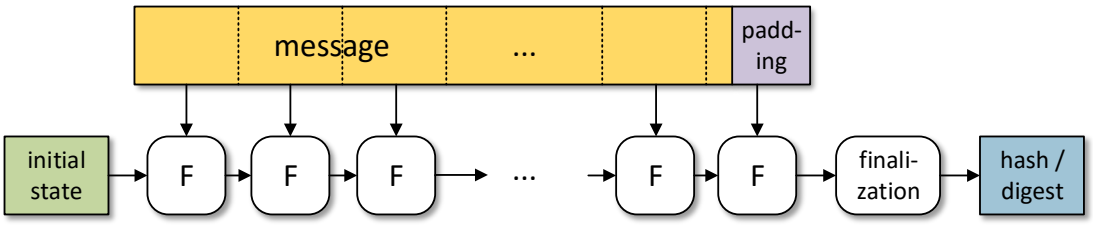


Figure 2.1 – Merkle-Damgård construction for cryptographic hash functions.

high performance requirements to radio-frequency identification (RFID) tags [PH13], which typically have to operate with severely constrained power, energy, storage, and processing resources. As a result, the organizers were not only interested in the cryptographic strength of the candidates, but also in the evaluation of the performance of the algorithms implemented on different platforms. Following the success of the public Advanced Encryption Standard (AES) selection process, the SHA-3 evaluation attracted many contributions comparing the performance of candidate algorithms on different platforms. Extensive results have been published for software [Be11, Por11] and hardware [GGM$^+$12, ECR11, HGG$^+$10] implementations of the SHA-3 candidates.

Before discussing the implementation details and acceleration potential of the five different algorithms on a specific microcontroller architecture, the following section introduces common properties of cryptographic hash functions that focus on the aspects that are relevant for their implementation (and less on the cryptographic properties). Afterwards, general descriptions of the five final-round SHA-3 candidates are given.

2.1.1 Principles of Cryptographic Hash Functions

We define the input of a hash function as the *message data*, or simply the *message*. The output of the hash function is defined as the *hash value*, *hash*, or *message digest*. Moreover, hash functions typically process their input in the form of *blocks*, one block at a time, while a message can be composed of multiple blocks. Hence, a part of the hash function is repeatedly applied on the different blocks of the message. Consequently, the hash function itself has a state, which depends on the message blocks that have already been processed. We call this data which describes the current state of the hash function, the *internal state data*, or *internal state*.

All algorithms are based on the common principle of combining message data (in fixed block sizes) with internal state data to produce a hash value of fixed size (224, 256, 384 or 512 bit). Each message block is processed separately using the same core algorithm, transforming the previous internal state into a new state. The internal state data is commonly partitioned into a set of state words of fixed size (8-64 bit), and the state is modified through multiple passes through a set of core functions for each message block.

A common way of building a (collision resistant) hash function according to this general scheme is the Merkle-Damgård construction [Mer90, Dam90], as depicted in Figure 2.1. An initial internal state of the hash function is transformed into the output hash value by repeatedly applying a (collision resistant) compression function F, which in each step combines the current internal state with the next block of the message, to produce the next internal state. The message is padded such that the total length is a multiple of the block size. A single step of the hashing process (one application of F) consists of many sub-steps and transformations, which are themselves most commonly applied for multiple rounds, to enhance the security of the hash function. Typically, a type of finalization step is performed in the end, to produce the final hash value. This finalization step mostly includes further transformation of the last internal state, by another function (which can be similar to F). This final transformation ensures the avalanche effect of the hash function, i.e. that for any two messages, which might only differ by a very small amount in their content, the difference in their hashes is significant. Moreover, the finalization often includes a truncation of the internal state to produce a hash value with a length that is shorter than the length of the internal state.

Please note that the Merkle-Damgård construction is simply presented here to illustrate the basic process of message digest calculation. For the specific SHA-3 final-round candidate algorithms, different variants of this general construction (that improve upon it for cryptographic reasons) are often used, e.g., the HAsh Iterative FrAmework (HAIFA) [BD06], or different approaches such as the sponge construction [BDPA11a]. However, the presented general scheme of combining message blocks with internal state to transform this state, applies to all hashing algorithms considered in this study, and gives a sufficient overview regarding the computational structure of the hash functions, relevant for their implementations.

All of the five SHA-3 candidate hash functions build their compression functions F (and the finalization step) out of three main types of operations: permutations of the set of state words, state word transformations, and lookups in tables of constants. The different types of operations are illustrated in Figure 2.2. Algorithmic constants are typically used in the form of lookup tables (LUTs), and are most commonly part of state word transformations. However, the applied permutation patterns and schedules can also be regarded as algorithm intrinsic constants.

Regarding the involved computational operations, state word permutations generally relate to memory moves, while state word transformations are realized using basic arithmetic and logical operations, such as addition, XOR, AND, NOT and state word rotations. The values of the incorporated constants are algorithm specific and depend on the current round number or internal state data (e.g., in the case of a substitution lookup table (S-box)).

The five SHA-3 algorithm proposals each offer different versions or instances of their hash functions, to provide different options regarding the output length, and therefore also security level of the cryptographic functions. We focus in this study only on the versions which are proposed as the replacements for SHA-256 [NIS15a], since it is the most commonly used

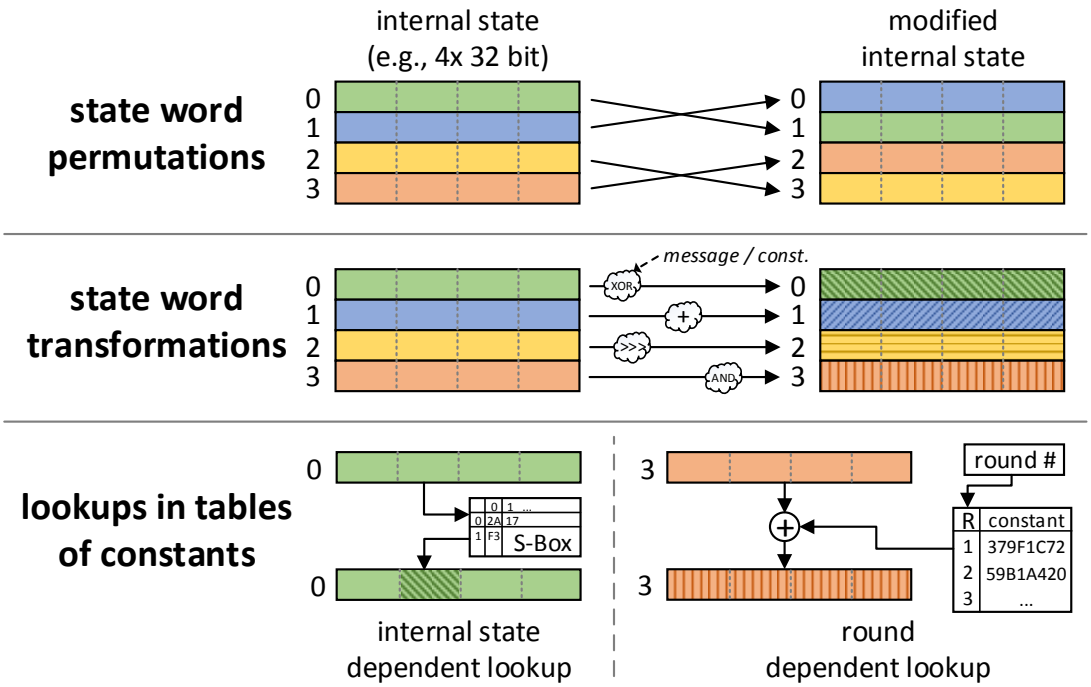


Figure 2.2 – Illustration of different operation types used in cryptographic hash functions.

digest size, and because it is suitable in the context of embedded systems. However, even when restricting the specific flavor of the SHA-3 algorithms to a common hash value length of 256 bit, the algorithms differ in multiple core aspects, which highlights the diversity of the considered set of algorithms. The internal state size varies between 512 and 1600 bit, while there are multiple ways for the logical organization of the state, e.g., as an 8×8 matrix of bytes, or as 8 state words each of length 64-bit. The message block size also depends on the algorithm, and is either 512 or 1088 bit. The algorithms furthermore differ in how they utilize their lookup tables, e.g., for initial constants only, or state data dependent lookups. Some algorithms like Keccak apply the mentioned full range of arithmetic and logical operations, while algorithms like Grøstl can be described efficiently only using XOR and memory moves, with the help of suitable lookup tables and architecture support for fine-grained memory accesses on byte-level.

2.1.2 SHA-3 Candidate Algorithms

In the following the 256 bit digest versions of the five SHA-3 finalist algorithms are summarized regarding their basic structures, employed operations, and organization of the internal state. Specific algorithm details such as initial state preparation or padding schemes are not discussed further, since they have typically not much impact on the algorithm performance compared to the compression functions, and would be the last parts to be optimized in a system, especially when considering performance for messages that consist of multiple blocks.

For more detailed descriptions of all five algorithms, especially regarding their various modes and features, as well as cryptanalytic results, please refer to the respective technical algorithm proposal documents [AHMP10, GKM+11, Wu11, BDPA11b, FLS+10].

BLAKE BLAKE-256 [AHMP10] uses an internal state of 512 bit, organized as a 4×4 matrix of 32-bit words. The compression function is applied for 14 rounds per message block of size 512 bit, and one round consists of 8 applications of the round function G, each using a different permutation of state words as input. The G function is the computational core of BLAKE. It transforms 4 state words of the internal state matrix (either one of the columns, or one of the diagonals), by intermixing them with each other and with parts of the message block, as well as with different entries from the table of 16 32-bit constants, depending on the round. A schedule known as the *Sigma Permutation* defines which parts of the message, and which constants are used for a specific application of G. The computational operations involved in the G function are 32-bit addition, XOR, and 4 different 32-bit right rotate operations by 7,8, 12, and 16 bit. At the end of each 14 rounds of calculations, a short finalization step generates the next hash chain value by combining the current hash value with the previous chain value and a so-called *salt value*.

Grøstl The internal state of Grøstl-256 [GKM+11] is 512 bit in size and organized as an 8×8 matrix of 8 bit words. The compression function f is based on two main operations called P and Q, which have a very similar construction but use different parameters for two of their four suboperations. P and Q are each applied for 10 rounds per application of f, combining the current hash value with a message block of size 512 bit. Both P and Q operations consist of four suboperations (*AddRoundConstant*, *SubBytes*, *ShiftBytes*, and *MixBytes*) that are similar to the ones used in AES [NIS01]. *AddRoundConstant* XORs the current state with a round based constant, which is different for P and Q. Each word (byte) of the internal state is substituted one-for-one in the *SubBytes* operation, by using the same 8-bit substitution table (S-box) that is also employed by AES. *ShiftBytes* performs a row based permutation of the internal state matrix, by left rotating each row of state words by a different amount. Moreover, the order of the shift amounts differs for P and Q. In the last step, the *MixBytes* operation, each column of the state is transformed by multiplying it in the finite field $\mathbb{F}_{256}$ with a circular constant matrix. The entries in this constant matrix can only take a few different values: 2, 3, 5, or 7. The final *Output Transformation* of Grøstl-256 (after all messages blocks have been processed) consists of 10 rounds of the P operation.

JH JH-256 [Wu11] uses a 1024 bit internal state, which has no specific organization into separate state words. In general the algorithm works on the granularity of bits, sometimes forming tuple arrangements of 4 bits, which are however not necessarily consecutive. The compression function is applied for message blocks of size 512 bit. It combines the message block with the first half of the current internal state by XOR, then applies the core function

E_8 of the algorithm with 42 rounds, and finally combines again the message with the second half of the state. E_8 consists of three main steps, and uses a different constant for each round. Depending on the bits of this round constant, one of two different 4-bit substitution tables (S-boxes) is applied on 4-bit tuples of the internal state. Note that the four state bits of these tuples are spread out over the entire state block, and are not necessarily grouped together. As a second step, a linear transformation L over two tuples of 4 bits is applied, which is defined as a multiplication in the finite field $\mathbb{F}_{16}$, but can be calculated simply by the use of XOR. The third step concludes one round of E_8, by a fixed permutation that shuffles the state.

Keccak The 256 bit digest variant of Keccak [BDPA11b] is defined as *Keccak[512]* (with a so called capacity of 512 bit), which is based on the sponge construction [BDPA11a] and employs an internal state of size 1600 bit, with a message block size of 1088 bit. Both, state and message block size are therefore uncommon compared to the other algorithms. The Keccak sponge construction absorbs (XORs) the message block into the state, and uses as the compression function the *Keccak-f[1600]* permutation, which works on the internal state that is organized as a 5×5 matrix of state words (lanes) of length 64 bit. One message block is processed in 24 rounds, and each round consists of the applications of 5 different functions θ, ρ, π, χ, and ι (*Theta*, *Rho*, *Pi*, *Chi* and *Iota*). *Theta*, *Rho*, and *Pi* are all linear transformations used for diffusion of the state, while *Chi* provides the required non-linearity for the hash function by intermixing the bits of different state words (rows in the 5×5 matrix for single bit positions in the state words) using AND, NOT, and XOR operations. *Theta* includes XORs and fixed rotation of single state words by 1 position (for operand calculations). *Rho* rotates all the state words by constant amounts, with different translation offsets per state word, and the *Pi* operation permutes the different state words within the 5×5 matrix. Finally, *Iota* XORs a round-dependent constant to the internal state to break the symmetry of the round operations.

Skein The Skein hash function [FLS+10] is constructed out of a tweakable block cipher called *ThreeFish*. Skein-512-256 uses a 512 bit state, consumes message blocks of size 512 bit, and produces an output hash of 256 bit. The internal state is organized as an array of 8 64-bit words. The main idea behind Skein is to keep the round structure simple, while using a large number of rounds to obtain security. In total there are 72 *ThreeFish* rounds, each of which is made up of four parallel applications of the *Mix* operation defined over 2 state words, followed by a fixed permutation of the array of state words. The *Mix* function has a simple construction, and consists of a 64-bit addition, an XOR and one rotation by a constant which depends on the current round and on which of the 8 state words the *Mix* operation is applied. Additionally, for every four rounds of *ThreeFish* a constant called a *subkey* is XORed onto the state. These *subkeys* are generated from a tweak value and the round counter, using a key schedule.

2.2 Embedded Microcontroller Architecture (PIC24)

Embedded microcontroller units (MCUs) come in a variety of different configurations, not only concerning the embedded peripherals, but also regarding the employed processing core architectures and on-chip memory types and their organization. MCU architectures are generally categorized by their data path width, which ranges from 8-bit, over 16-bit, up to 32-bit, with varying instruction word lengths. The series of PIC microcontrollers is frequently employed in numerous embedded systems with over one billion MCU devices sold per year (as of 2013) [Mic13]. In the following we focus on the PIC24 device family [Mic09b], which is the general purpose 16-bit MCU architecture variant of the PIC family. The PIC24 comprises common features of 16-bit MCU architectures used for embedded systems with light data processing requirements, and was hence chosen as the reference architecture for this study. Due to its more extensive instruction set and more complex addressing features — compared to typical 8-bit MCUs — the PIC24 architecture is suitable for applications that go beyond simple control tasks, and also require simple processing of data, under low-cost and low-power constraints.

The PIC24 [Mic09b] derives its name from its 24-bit wide instruction words, and is based on a Harvard architecture with a 16-bit data path. Both the data memory bus and the working registers are 16-bit wide. The block diagram of the CPU core of the PIC24 MCU is shown in Figure 2.3. The core features a 16-entry register file, a 16-bit arithmetic logic unit (ALU), a dedicated multiplier, and a unit providing hardware support for multi-cycle division. The commercial implementation from Microchip furthermore includes a direct memory access (DMA) controller, an interrupt controller, as well as a module which allows access to the program memory to use its content as data within the CPU. Moreover, the data bus is connected to an extensive set of peripherals, which are co-integrated on the system-on-chip (SoC) and vary depending on the specific model of the device.

Section 2.2.1 gives a summary of the instruction set architecture (ISA), which is followed by the description of our custom microarchitecture implementation of the PIC24 in Section 2.2.2.

2.2.1 Instruction Set Architecture

The PIC24 ISA comprises 87 base instructions, divided into 9 functional groups, covering standard program flow and control instructions, as well as integer arithmetic, logic, and bit manipulation operations. The majority of the instructions allow for a variety of addressing schemes and operand modes, resulting in 190 different effective instructions. The addressing modes supported by the ISA are: register direct, register indirect, file register[1], and immediate. Register direct addressing accesses the 16 working registers directly, while register indirect addressing accesses the data memory location referenced by the content of the given work

[1] *File register* is a term that Microchip uses in its PIC architectures to refer to the data memory. The term originates from 8-bit MCU architectures which only operate with a single working register and address their small on-chip memory in the form of a file register.

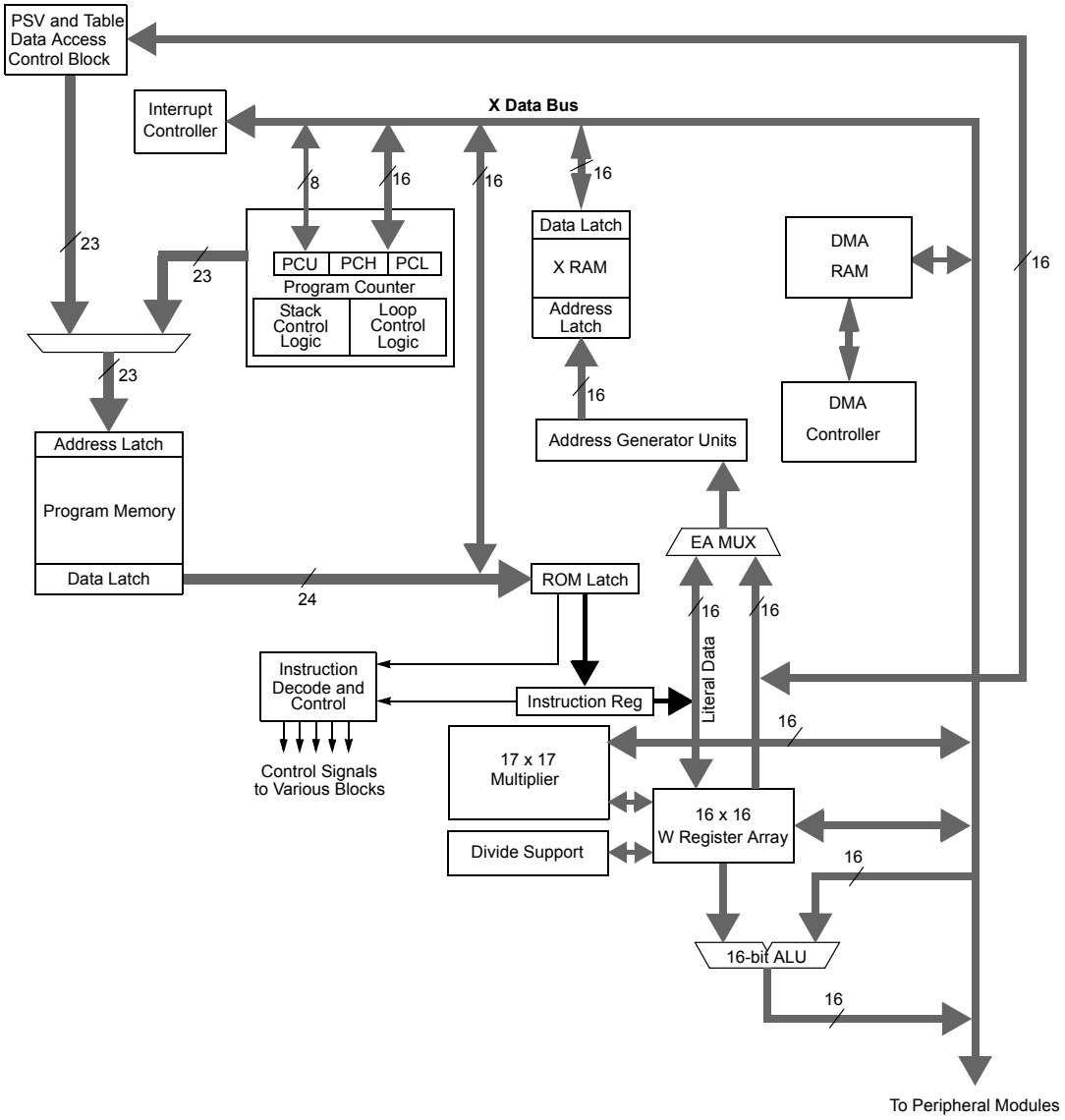


Figure 2.3 – Microchip PIC24 CPU core block diagram [Mic09b], illustrating the internal MCU core structure of a commercial PIC24HJXXXGPXXX device.

register. File register addressing uses a predetermined address encoded in the instruction word, to access the data memory directly. The immediate addressing mode refers to the possibility of encoding immediate constants in the instruction word for the use as operands.

Register indirect addressing additionally provides six different modes to improve the efficiency of the code, when processing sequential data in memory. The modes allow to modify the register that holds the memory address, or perform additional address arithmetic, within the same instruction. The six modes are: pre-increment, pre-decrement, post-increment, post-decrement, register offset, and no modification.

Furthermore, all operands and destinations can be either addressed in byte or word mode, for most instructions. Word mode represents native 16-bit data addressing, while byte mode only operates on the least significant 8 bit of the corresponding data word. This feature is not necessarily supported by all 16-bit MCU architectures, and can be an important factor for the overall performance of an algorithm implementation (e.g., for the Grøstl algorithm; see Section 2.1.2 and Section 2.4).

The CPU uses a 16-entry general purpose register array named *W0-W15*, with a stack pointer assigned to register *W15*. In addition, there are several status and control registers, and a 16-bit repeat loop counter used in conjunction with the *REPEAT* instruction, which loads this counter. This instruction repeats the immediately following instruction as often as specified, allowing for very simple hardware loops, thus reducing the program size.

2.2.2 Microarchitecture

The microarchitecture of our PIC24 implementation, as detailed in Figure 2.4, is based on a 3-stage pipeline, and hence provides similar timing behavior, concerning instructions per cycle (IPC) and stalls compared with the original, commercial PIC24 device. The pipeline consists of a fetch, a decode, and an execute stage, and has an IPC throughput of almost 1. All standard instructions (which are 24-bit long) execute in a single cycle. Rare double-word (48-bit) instructions, such as *CALLs* and *GOTOs*, with long immediates execute in 2 cycles. The only exception to this are the division instructions, which take multiple cycles (equal to the number of divisor bits) to complete, due to their iterative hardware implementation.

As in the commercial device, the instruction memory of our implementation is a single-port 24-bit wide memory with 23-bit addresses, and single cycle access latency. Depending on the specific implementation, only a certain range of this address space is used. Here we implement the instruction memory with a size of 2048 instruction words, i.e., 6 kbyte of SRAM, since it is the minimum size to hold the largest of the 5 SHA-3 algorithm implementations. For comparison, the commercial PIC24 devices come in a variety of sizes, providing from 4 up to 1024 kbyte of flash instruction memory. The data memory is realized as a dual-port memory with one read port and one write port, both providing single cycle access latency. The memory is 16-bit wide and uses 16-bit addresses, allowing concurrent read and write access within the same clock cycle. The size of the data memory implementation is 1024 words, which corresponds to 2 kbyte of SRAM.

The work register file has three read ports to allow up to two operands and/or memory addresses (read source and/or write destination) to be fetched at the same time. Two write ports are dedicated to the operand fetch module, to support the register indirect addressing modes with increment or decrement on both one operand and the destination of the instruction, within one cycle. The register file has an additional two write ports, which are used for write-back of ALU results (and memory loads). Double register writeback is used for the division instructions, which write/update two values per cycle (quotient and remainder).

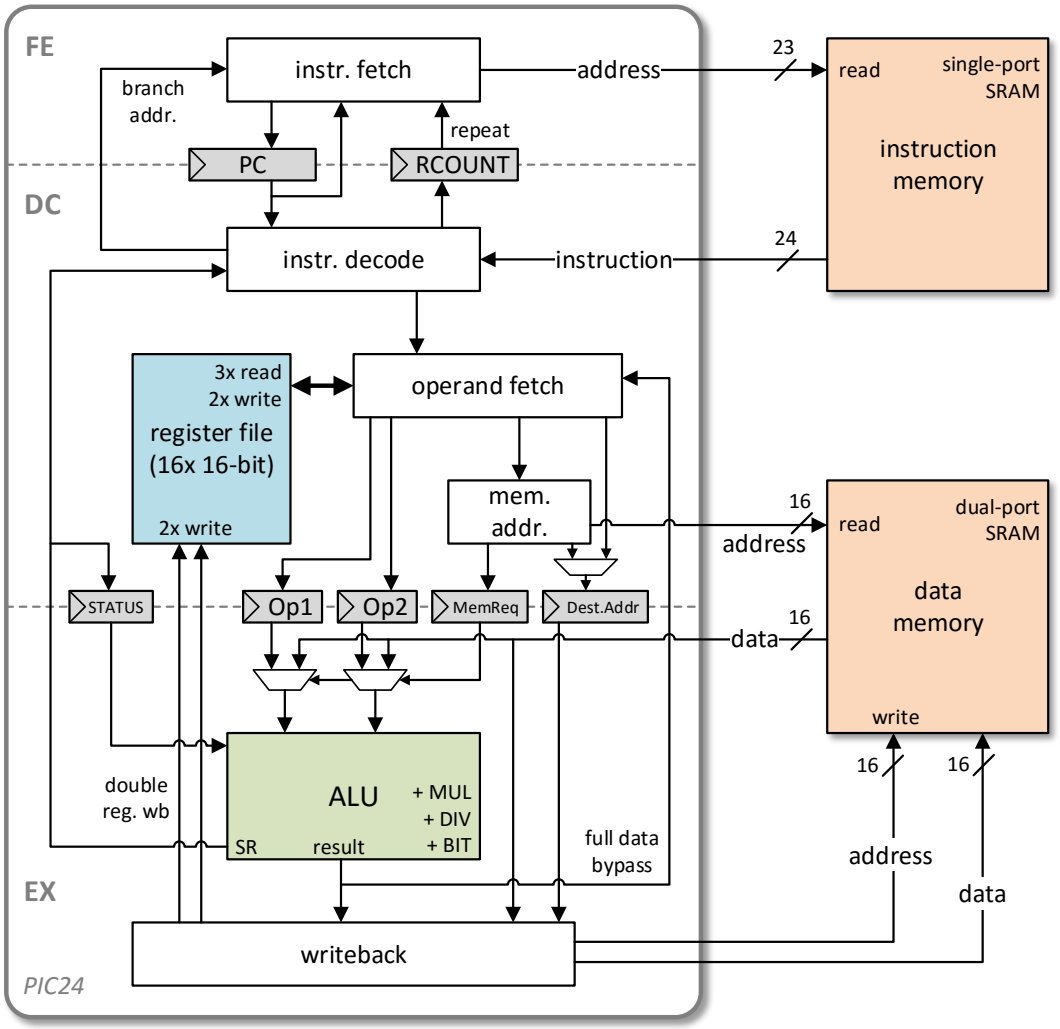


Figure 2.4 – Microarchitecture of custom PIC24 implementation with a 3-stage pipeline, comprising a fetch (FE), decode (DC), and execute (EX) stage. The instruction memory and data memory are tightly coupled to the core.

Our custom microarchitecture features some optimizations and design choices, which increase IPC performance of the core, compared to the commercial implementation. Branch instructions are always executed immediately in one cycle, also for the case when the branch is taken, which eliminates the otherwise required stall cycles. Moreover, in our microarchitecture all forms of read after write (RAW) hazards are resolved without stalls through bypassing, which also includes data memory locations, and is not limited to working registers only. Additionally, stalls are no longer issued, if a data dependency stems from a register, which is used with a register indirect addressing mode in the subsequent instruction, i.e., when the register content is used as a data address.

2.3 Performance Metrics

In order to assess the performance of the software implementations together with the respective hardware system, we report four main metrics: cycle count, data memory consumption, program memory consumption, and required hardware area. This section briefly discusses these metrics, and explains how they are calculated.

Cycle Count The cycle count is a measure for the complexity and also for the energy consumption of an implementation on a specific processor. The cycle count (per unit of data) is also inversely proportional to the throughput, which describes how fast the hash algorithm works for a given clock frequency. As introduced in Section 2.1.1, each hash algorithm generally operates on message blocks of fixed length. For all SHA-3 candidates this block length is 512 bits, except for Keccak which uses 1088-bit message blocks. The average number of cycles needed to process one block is recorded, and then divided by the number of bytes per block, while excluding any message initialization efforts (e.g., for padding), or finalization steps that occur at the end of a message. Hence, we report the long message performance in the commonly used cycles/byte format.

Data Memory As described in Section 2.2.2, the microcontroller employed in this work uses 16-bit wide data memory, realized as SRAM. In a microcontroller, the total amount of SRAM available for all applications is a scarce resource due to the fact that SRAMs occupy significant circuit area for practical memory sizes. The data memory utilization is always expressed in number of bytes throughout the reported results.

Program Memory Since the PIC24 uses the Harvard architecture, which is common in microcontrollers, the data and program memories are fully separated. In practice (i.e., a full embedded system implementation), the program memory could hence be implemented depending on the application, as a read-only memory, a one-time programmable memory, or most commonly flash memory, all of which have less hardware overhead than an SRAM, as used for the data memory. While still a significant burden, program memory is therefore often not considered to be as expensive as data memory.

All PIC24 instructions are either one or two instruction words long, and can therefore be stored as three or six bytes, respectively. The overall comparison tables always list the total number of bytes. Some tables use *instructions* when reporting the complexity of individual functions for clarity. We use the word *text* to describe the complete content of program memory. Static initialization data (e.g., a set of initial values for the hash chain) is normally used once per execution of an algorithm. It can therefore in practice be stored in sections linked to the less expensive program memory, and is hence in addition to program code also accounted for as text.

Hardware Area A hardware implementation always involves a compromise between operation speed, power & energy used, and the circuit area. In this study, all microcontroller descriptions are synthesized using a standard-cell-based design flow for a 90 nm CMOS technology. Furthermore, all MCU instances reported in this study are analyzed regarding their area requirements for the full range of different core clock constraints covering all target constraint corners (see Section 2.6.3). Area overhead of the proposed instruction set extensions (ISEs) with respect to the original implementation or any other absolute area figures given in comparison tables and the continuous text are always for a chosen target core-clock frequency of 200 MHz.

Area is expressed in terms of kilo gate equivalents (kGEs), where one GE is taken as the area of a 2-input NAND gate with a standard driving strength. The conversion factor is 3.136 μm^2 per GE. The results are all synthesis results, and do not include post-layout parasitic effects.

The total area of the microcontroller subsystem comprises the data memory, program/instruction memory, and the actual MCU core. In our implementation, the two memories are implemented as SRAM macros. Even when very modest/minimal sizes were to be selected for these memories, the SRAMs would occupy at least two thirds of the total area. Since the memory overhead is large and determining a fair size for the memory (which covers different feasible application scenarios) is not straightforward, we report only the change in the MCU core area when making comparisons between different designs.

2.4 Baseline Performance on PIC24

This section describes the characteristics of our baseline implementations for the PIC24 architecture for all five SHA-3 finalist algorithms, followed by a performance comparison with other state-of-the-art software implementations on MCU systems.

2.4.1 Baseline Implementations

The baseline implementations for all algorithms are developed from scratch[2] in PIC24 assembly language, in order to start from a baseline which is already highly optimized regarding hashing performance, as well as memory requirements. Compiled programs from the provided reference implementations in C were not further considered as a baseline for this study for two reasons: 1) the resulting assembly code structure would have made systematic, architecture specific bottleneck analysis more challenging, complicating the development of efficient ISEs; and 2) the performance, both in terms of cycle count and code density, is significantly lower with the compiled programs, even when using the highest optimization settings of the commercial compiler tool chains. Consequently, the hand-optimized assembly implementations allow to identify clearly the core performance bottlenecks of each algorithm implementation.

[2]The implementations are based on the official algorithm proposal documents, and their accompanying material (documentation and sample code) discussing relevant code optimization techniques.

Note that every implementation is fully functional as a stand-alone application, performing two invocations of the algorithm code (one for single block hashing, and one for multi block hashing), followed by an automatic check of the message digest, to verify the correctness of the implementation. All implementations include code written for initialization and preparation of message blocks, however these have not been included for performance evaluation, i.e., only long message performance values are considered in this study. This choice — to focus only on the algorithm kernels — was made, since the code for message block preparation (e.g., padding) is similar for all candidates, and only executed once per block, making it the least relevant code regarding any possible optimizations.

The following baseline reference implementation descriptions refer to algorithm specifics and details as introduced in Section 2.1.2.

BLAKE BLAKE is shown to be one of the fastest implementations with the smallest memory footprint in various studies. The throughput performance of our baseline implementation confirms this trend, with a hashing speed of 155 cycles/byte. Table 2.1 shows the breakdown of the memory resources in our PIC24 assembly implementation (together with the improvements due to the introduction of ISE, which are discussed in Section 2.5.3).

The core of the implementation is the *G* function which operates on four 32-bit words. To this end, the corresponding words of the internal state are first loaded into four register pairs using the appropriate permutation pattern. After the transformation through the *G* function call,

Table 2.1 – Memory resource breakdown of the baseline implementation of the BLAKE algorithm on the PIC24 architecture, and the changes due to instruction set extensions.

	Description	Baseline	Δ ISE	PIC24	PIC24+ISE
Data				**488 byte**	**200 byte**
	Hash State	32 byte			
	Salt	16 byte			
	Counter	8 byte			
	Message Block	64 byte			
	Internal State	64 byte			
	Misc. Variables	4 byte			
	Stack	12 byte			
	Constants (Pi)	64 byte	-64 byte		
	Sigma Permutations	224 byte	-224 byte		
Text				**1028 byte**	**818 byte**
	CompressBlock()	274 instr	-39 instr		
	RoundFuncG()	58 instr	-31 instr		
	Initial State	32 byte			

these state words are stored back to their original positions in memory.

The *G* function performs a lookup of the required indices for the current round in the *Sigma Permutation* table (stored in data memory) and uses those indices to address the correct words in the message data block and the round constants table. The employed additions are 32-bit wide and use the add-with-carry instruction of the PIC24 ISA. The 32-bit rotations by 7, 8 and 12 bits are emulated with 6 instructions each, utilizing the OR, left shift, and right shift instructions of the reference ISA. The rotations by 16-bit can be implemented without any computations, simply through reordering of the used working registers.

Grøstl Grøstl implementations for small microcontrollers have in general been considered to be slow and large. In terms of throughput our baseline implementation requires 462 cycles/byte, making it one of the slowest algorithms. Table 2.2 provides a breakdown of the memory footprint of our implementation.

It is important to note that due to the organization of the state matrix into byte elements that are used in row as well as column order, byte mode support of the MCU architecture is imperative to avoid even costlier implementations in terms of cycles and code size. The byte mode feature is particularly important for efficiently addressing single elements in byte

Table 2.2 – Memory resource breakdown of the baseline implementation of the Grøstl algorithm on the PIC24 architecture, and the changes due to instruction set extensions.

	Description	Baseline	Δ ISE	PIC24	PIC24+ISE
Data				**982 byte**	**214 byte**
	Hash State	64 byte			
	Message Block	64 byte			
	Internal State	64 byte			
	Misc. Variables	6 byte			
	Stack	16 byte			
	S-Box	256 byte	-256 byte		
	Mult 2 LUT	256 byte	-256 byte		
	Mult 4 LUT	256 byte	-256 byte		
Text				**2619 byte**	**819 byte**
	CompressBlock()	173 instr			
	PermutationPQ()	26 instr			
	AddRoundConstPQ()	70 instr	-70 instr		
	SubBytes()	134 instr	-134 instr		
	ShiftBytesPQ()	246 instr	-246 instr		
	MixBytes()	150 instr	-150 instr		
	OutputTransform()	74 instr			

increments in the data memory, which is commonly organized in more coarse-grained word sizes.

The instructions underlying the four suboperations of P and Q can be reduced to simple XORs and memory moves, combined with the use of three LUTs. The first LUT is the AES S-box, which maps every 8-bit value of the state to another 8-bit value. The remaining two LUTs hold the precomputed results for the multiplications of any 8-bit value by either 2 or 4 in the finite field of $\mathbb{F}_{256}$ as used in AES. The results of multiplications with the constants 3, 5, and 7 (as they are required in the *MixBytes* step) are constructed by XOR'ing together the corresponding combinations of the multiples of 2 and 4, and the value itself (multiple of 1).

JH While the reference algorithm description (as summarized in Section 2.1.2) is suitable for hardware implementations, a so called *bit-sliced* implementation [Wu11] can be used in software to improve the efficiency on a processor based system significantly. In bit-sliced implementations, instead of performing operations on small bit-width words (i.e., for the 4-bit S-box functions), large words are formed by combining the same binary digit of multiple input words together into one working register. Since the PIC24 is a 16-bit architecture, we have implemented a bit-sliced implementation tailored to a data word size of 16 bit. Operations in JH can be transformed so that four 16-bit words can be formed by combining the bits of 16 separate 4-bit tuples. To store the precomputed round constants for this bit-sliced implementation, 42x16x16 bits = 1344 byte are required, which accounts for the majority of the *data* segment, as can be observed in Table 2.3. This contribution also renders JH the algorithm with the largest memory footprint in terms of *data* size for the baseline implementation. The

Table 2.3 – Memory resource breakdown of the baseline implementation of the JH algorithm on the PIC24 architecture, and the changes due to instruction set extensions.

	Description	Baseline	Δ ISE	PIC24	PIC24+ISE
Data				**1550 byte**	**206 byte**
	Hash State	128 byte			
	Message Block	64 byte			
	Misc. Variables	4 byte			
	Stack	10 byte			
	Round Constants	1344 byte	-1344 byte		
Text				**4649 byte**	**2183 byte**
	CompressBlock()	144 instr			
	SBox()	432 instr	-37 instr		
	L/MDS()	144 instr			
	Swap()	787 instr	-785 instr		
	Initial State	128 byte			

hashing speed of the baseline implementation is the slowest of all five SHA-3 candidates with 464 cycles/byte.

The bit-sliced implementation calculates its core function (which is the 4-bit S-box function) via an instruction sequence using only AND, NOT, and XOR (as described in [Wu11]), which operates on four working registers holding the 16 bit-sliced input tuples. This calculation is more efficient than performing actual table lookups on separate tuples, due to the ability to process 16 separate bits in parallel, and the reduced overhead in how tuples would have to be assembled. This is followed by the L transformation, which only requires a set of XORs applied on 8 bit-sliced data words. The final *Swap* operation is performed on a group of 8 words each, by either using bit-masks combined with left and right shifts and logic ORs for small shift/interleave values (1, 2 and 4), or simple MOV instructions for larger values (8, 16, 32 and 64).

Keccak The Keccak algorithm requires in its baseline implementation 188 cycles/byte. As shown in Table 2.4, the data memory consumption is moderate, while the text section is larger than for most of the other algorithms, due to extensive unrolling of all core operations, which incorporate many positional constants (state word coordinates), as well as permutation schedules including rotation constants. It is possible here to trade hashing performance for more compact code size, however priority has been given to improved hashing speed.

All five core operations work on 64-bit state words, which are typically held in a set of four working registers. The *Theta* operation performs a series of XOR combinations on all state elements, including single bit rotations implemented with the rotate through carry instruc-

Table 2.4 – Memory resource breakdown of the baseline implementation of the Keccak algorithm on the PIC24 architecture, and the changes due to instruction set extensions.

	Description	Baseline	Δ ISE	PIC24	PIC24+ISE
Data				**448 byte**	**352 byte**
	Hash State	200 byte			
	Message Block	136 byte			
	Misc. Variables	4 byte			
	Stack	12 byte			
	Round Constants	96 byte	-96 byte		
Text				**3480 byte**	**2415 byte**
	CompressBlock()	147 instr			
	Theta()	307 instr	-20 instr		
	RhoPi()	349 instr	-324 instr		
	Chi()	345 instr			
	Iota()	12 instr	-11 instr		

tions of the PIC24 ISA. The *Rho* step performs a 64-bit rotation on each state element, using a different rotation constant for each element of the 5 × 5 state matrix, depending on its row and column index. The rotations are implemented either by shifts together with OR instructions, or using MOV instructions only, where possible. The *Pi* step permutes the elements of the state matrix and is combined with the *Rho* step in our baseline implementation. A state element is loaded from memory, rotated and then directly written to its new destination, according to the permutation table defined by *Pi*. This permutation can be chained, so that only one element has to be buffered for the complete state to be rotated and permuted in place. The next transformation is *Chi*, it applies a combination of logical operations of AND, NOT, and XOR to rows of the state word matrix. The final step *Iota* XORs the constant depending on the current round onto the state element (0,0).

Skein In its baseline implementation, the hashing performance of Skein with 158 cycles/byte is close to the fastest algorithm (BLAKE). Skein has the largest memory consumption in terms of text (as detailed in Table 2.5), which is dominated by the fully unrolled *ThreeFish* rounds with varying input pairs of state words and rotation constants (totaling 32 different invocations of the *Mix* function). Another major contributing factor to the large program size is the key injection, which is implemented in 9 slightly different variations to cover all 9 injection cases separately. Also here, as in the case of Keccak, a decrease in hashing speed could be traded for reduced program memory consumption.

The computational core of Skein is the *Mix* function of *ThreeFish*. This simple *Mix* function

Table 2.5 – Memory resource breakdown of the baseline implementation of the Skein algorithm on the PIC24 architecture, and the changes due to instruction set extensions.

	Description	Baseline	Δ ISE	PIC24	PIC24+ISE
Data				**242 byte**	**242 byte**
	Hash State	64 byte			
	Key Schedule	72 byte			
	Tweak	24 byte			
	Message Block	64 byte			
	Misc. Variables	2 byte			
	Stack	16 byte			
Text				**5734 byte**	**4678 byte**
	CompressBlock()	75 instr			
	KeyscheduleGen()	83 instr			
	ThreeFishRounds()	876 instr	-352 instr		
	InjectKey()	844 instr			
	OutputTransform()	12 instr			
	Initial State	64 byte			

consists of arithmetic and logic operations on 64-bit words, some of which are well supported by the standard ISA, through the use of addition with carry and XOR. The only problematic operation is the left rotation over 64-bit words, which has to be emulated (as in the case of Keccak), either in the general case by shifts together with OR instructions, or in the case of rotations by a multiple of 16, by using MOV instructions only. The key schedule is generated using XOR operations, while the subkeys are injected via 64-bit additions.

2.4.2 Performance Results and Comparison

Embedded systems and particularly performance and/or resource constrained small microcontrollers have been identified as important targets for future SHA-3 implementations. Hence, authors of nearly all algorithms have also published preliminary performance results and/or estimations for specific small microcontrollers on their webpages. The most important study regarding benchmarking of SHA-3 candidates on deeply embedded processors and microcontrollers has been done by Christian Wenzel-Benner and Jens Gräf [WBG10]. They maintain a webpage [WBG11], where their results are published. These results have also been included on the eBASH website [Be11]. Thomas Pornin [Por11] has developed a library (sphlib) which uses standard C, and therefore can easily be ported to a variety of platforms, even with restricted resources. The library includes comparisons on several platforms, but does not necessarily reflect the results that are achievable with hand-crafted assembly kernels.

While large general purpose microprocessors often share a very similar ISA and structure[3] (i.e., x86/x86-64), small microcontrollers, however, typically come in many different flavors. They differ in their structure (Harvard, von Neumann), width of their datapath (4-32 bits), and provide differing resources (number of registers, availability of hardware multipliers, etc.), on top of the many different ISAs. This makes a general and fair comparison between microcontrollers very difficult.

Nevertheless, we use the PIC24 microcontroller as a representative for the variety of available 16-bit microcontrollers, which complements other published results which so far have focused on 8-bit and 32-bit MCU architectures only. In Table 2.6 we list our implementation results regarding hashing performance, together with the results of other state-of-the-art implementations. The relative performance of the algorithms is given in relation to BLAKE to allow for easy comparison of the individual performance differences between the various MCU platforms.

From Table 2.6 it can be observed that in general 32-bit MCU architectures clearly benefit in terms of cycle count over a 16-bit architecture, since most algorithms internally use state word sizes that are larger (64-bit) than the native word sizes of the MCUs. Moreover, it can be seen, that BLAKE is consistently fast throughout all published works. The ranking is not so clear for other candidates, however Grøstl and JH are mostly ranked among the slower

[3] Although the specific implementations on microarchitecture level can differ significantly, devices aimed at the same market typically provide comparable computational resources and features.

Table 2.6 – Comparison of throughput numbers [cycles/byte] of published microcontroller implementations. Numbers in parentheses show performance normalized to BLAKE performance within the respective architecture.

	This work	[Por11]	[Por11]	[WBG11]	[WBG11]	[Gou10]
Architecture	PIC24	ARM-M3	ARM920T	ARMv5TE	ATmega128	8051
Datapath	16-bit	32-bit	32-bit	32-bit	8-bit	8-bit
Specification	third	third	third	third	third	second
BLAKE	155 (1.00)	89 (1.00)	54 (1.00)	87 (1.00)	1241 (1.00)	643 (1.00)
Grøstl	462 (2.98)	455 (5.11)	313 (5.79)	216 (2.48)	11198 (9.02)	1327 (2.06)
JH	464 (2.99)	370 (4.16)	395 (7.31)	361 (4.15)	3829 (3.06)	-
Keccak	188 (1.21)	192 (2.16)	197 (3.65)	-	1115 (0.89)	1745 (2.71)
Skein	158 (1.02)	128 (1.44)	129 (2.39)	184 (2.11)	1444 (1.16)	1944 (3.02)

implementations. It can be observed that our PIC24 baseline implementations rank quite favorably when compared to other implementations, considering the 16-bit data path of the architecture. In particular, our Keccak realization outperforms all published results for microcontrollers, even those for 32-bit architectures. This is partially attributed to the fact that we use hand-crafted assembly code, rather than compiled C-code.

The result comparison furthermore highlights that all our baseline implementations are in general in line with the current state of the art for MCU architectures, and hence provide a useful basis for the development of ISEs, as well as a valid reference point for the benchmarking of the resulting relative performance gains from those ISEs.

2.5 Architecture Enhancements Through Instruction Set Extensions

An important point about embedded MCUs is that, increasingly for many applications, the MCU is often integrated in a custom SoC, where it can potentially be customized with instruction set extensions to the application. The basic idea behind such extensions is to enhance the architecture (specifically, often the datapath) of an MCU so that an application can be executed with fewer number of instructions, increasing throughput, reducing energy consumption, and saving resources (RAM and ROM). Such new additions can add some overhead (increased circuit size, and/or reduced maximum operating speed due to additions to the datapath), but in many cases are still able to offer significant advantages. Most intellectual property (IP) vendors already offer a wide variety of ISE options for their embedded MCUs.

Moreover, once an algorithm has been selected as an official standard (e.g., a NIST standard), processors and MCUs with specifically tailored ISEs are expected to appear as IP blocks or stand-alone components, similar to the case of AES in modern CPUs. For the SHA-3 standard, which has just recently been standardized [NIS15b], first IP cores in the form of look-aside cores/accelerators are already available [Syn16]. Such look-aside cores implement the full

hashing functionality in the form of an accelerator, which operates asynchronously, attached to a host processor. This asynchronous approach is typically interesting for applications with high throughput requirements and/or streaming characteristics, since the host core remains available for other tasks, while the current message is processed in parallel. In contrast to such look-aside cores, which in the context of MCUs consume considerable hardware area by themselves, ISEs operate synchronous within the host core, to provide in-core acceleration. The ISEs are typically based on the concept to reuse as much as possible of the resources which are already provided by the MCU microarchitecture.

Many companies offer solutions [Syn12, Ten12, Tar12] that facilitate the design of application-specific processors and the customization of commonly used MCUs. Starting from a high-level description, such solutions allow changes to be made to an ISA, and not only produce the corresponding hardware description for integration into a SoC, but also generate the necessary tool chain (compiler, assembler, linker) that will support this enhanced processor, allowing it to be used with relative ease.

In the following, we consider the optimization of the 16-bit PIC24 MCU with ISEs, for improving the execution speed and the memory footprint of cryptographic hash functions. We show how one can achieve considerable performance gains, with types of ISEs that are different from the usual approach of common datapath extensions. Highlighting the algorithm-specific performance bottlenecks, we propose ISEs that facilitate the generation of memory access patterns, lookup table integration, and extension of computational units through microcoded instructions. We report the execution speedup, the memory reduction, and the overhead associated with these extensions.

2.5.1 ISE Design Flow

The design flow used for the implementation and evaluation process is illustrated in Figure 2.5. The flow comprises a hardware implementation path and a software development, optimization and verification path.

On the processor architecture side, the ISA and the microarchitecture of the PIC24 processor are first described in LISA (Language for Instruction Set Architectures), a language tailored to the design of application-specific processors [HKN+01, SLM07]. This description is then automatically translated into a register transfer level (RTL) VHDL description that can be synthesized into gates using RTL synthesis tools (in our case Synopsys Design Compiler Ultra) to evaluate the silicon area and performance. The design is mapped to a standard-performance regular threshold-voltage 90 nm CMOS technology, with nominal supply voltage of 1.0 V. In addition to the hardware description, Processor Designer [Syn12] generates the basic software tool chain (assembler and linker) and a fully cycle-accurate instruction set simulator (ISS).

On the software side, the algorithm specifications for the five SHA-3 candidates provide the starting point for an initial implementation in the native assembly language of the PIC24

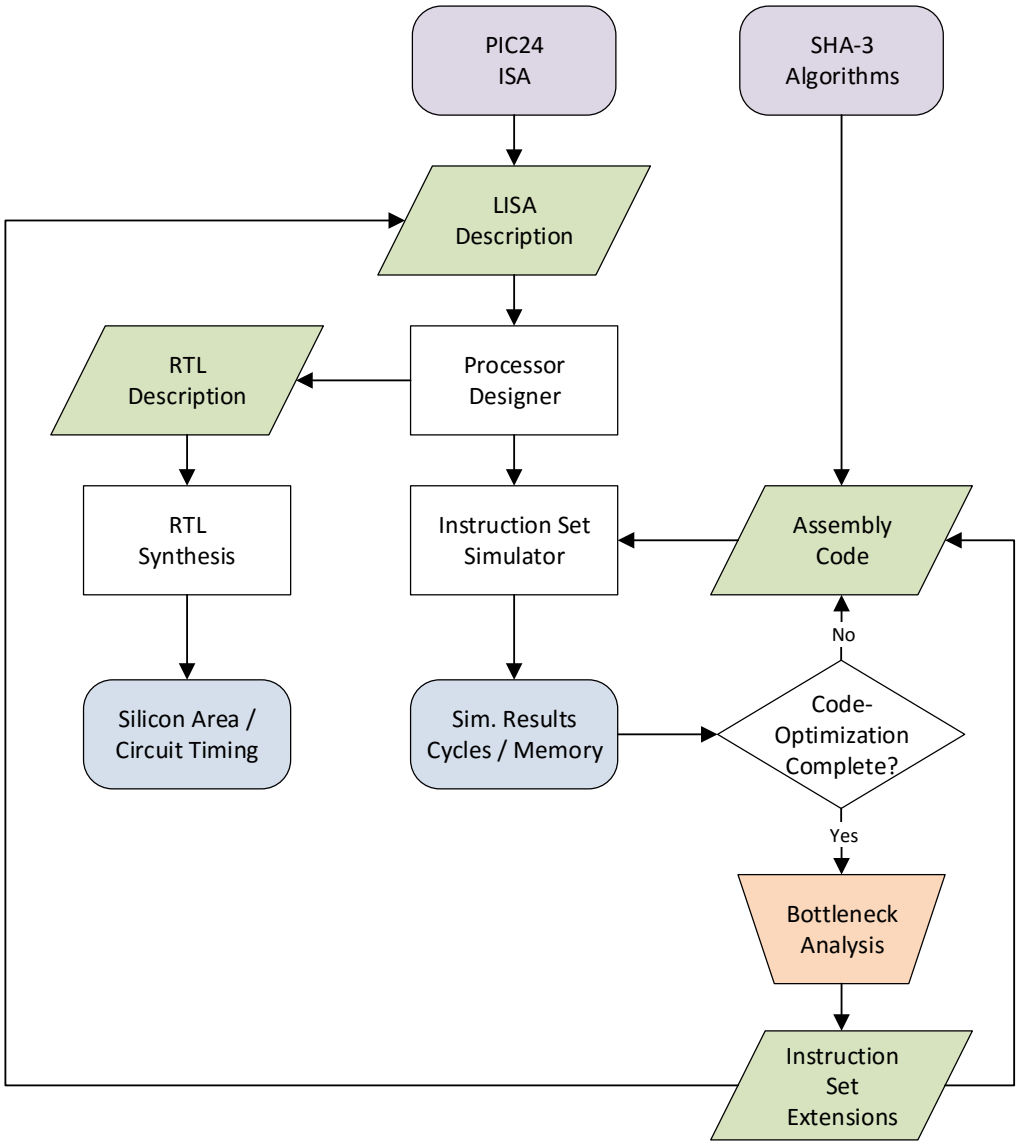


Figure 2.5 – Illustration of the design flow and tools employed for implementation, benchmarking, and development of instruction set extensions.

ISA. Multiple iterations of profiling and software-optimization are carried out, with the help of profiling tools provided by the ISS, to reduce code and memory size and to reduce cycle counts. During the optimization phase, improvements are mainly achieved by efficient use of all available operand addressing modes provided by the architecture, and applying coding techniques such as unrolling and precomputation, where feasible (considering execution time versus program and data memory trade-offs with a slight preference for execution time over memory requirements). All implementations are always verified against the provided test patterns to guarantee full compliance with the original algorithm specifications.

At the point, where no further gains are achieved using the standard ISA of the PIC24 micro-

controller, the different algorithms are analyzed manually, to identify specific performance and memory bottlenecks. For each candidate, we identify custom instructions that promise an improvement in terms of memory footprint and/or cycle count, while remaining compatible with the general microarchitecture of the core with only minor modifications. To be more precise, the extended implementations of the PIC24 core remain fully backward compatible and no changes are made to key components such as the register file, the memory interfaces, the pipeline stages, or the instruction formats. The additional instructions are incorporated into the LISA model and into the assembly descriptions of the SHA-3 kernels. ISEs are fine tuned through multiple iterations of hardware/architecture and code adjustments followed by benchmarking of the gains in terms of cycle count and memory utilization.

2.5.2 ISE Types for Cryptographic Hash Functions

Although the reference PIC24 ISA comprises a large number of instructions, our implementations without ISEs mainly focus on a core set of instructions, directly matching the base operations intrinsic to the class of cryptographic hash functions, as described in Section 2.4.1.

We investigate three different forms of performance bottlenecks that cannot be removed using the reference ISA. First, we analyze conventional basic arithmetic and logical operations, which however have to be carried out on data word lengths that are wider than the ones supported by the underlying native architecture (e.g. 64-bit data words on a 16-bit MCU). Depending on the operation, these operations often have to be emulated by a number of assembly instructions that is even much larger than the ratio between the word size and the width of the data path. A very relevant example in the context of cryptographic hash functions is the rotation of state data words. Furthermore, we examine complex memory access patterns, which produce overhead in form of advanced address arithmetic and potential multiple accesses of the same data in data memory. Last, we investigate static data memory requirements in the form of lookup tables, occupying valuable data memory space, and further increasing the number of required memory accesses. Derived from these three types of performance bottlenecks, the corresponding three classes of proposed ISEs for cryptographic hash functions are described in the following. To illustrate the three ISE types, algorithm-specific examples of their application to the introduced group of five SHA-3 hash algorithms on the PIC24 architecture are given. Further more specific implementation details regarding all the mentioned examples are provided in Section 2.5.3.

The general approach followed by all three ISE types is to keep the introduced hardware overhead low, especially regarding any new additions to the data path of the processor core. This is achieved by utilizing already existing microarchitectural resources as much as possible, and designing the ISEs such that they only address the manually identified inefficiencies within the algorithms kernels caused by microarchitectural bottlenecks and/or the standard ISA, without necessarily compacting entire sequences of code into new instructions requiring completely new functional units. As such, the proposed ISEs provide a slightly different

approach to in-core accelaration and the design of ISEs, especially compared to automated methods for ISE identification/extraction and generation for embedded processors [PPIM03, AOD+08].

Extension of Computational Units Through Microcoded Multi-Cycle Instructions

The most common way of extending an ISA is by introducing additional, often more complex, computational units to the datapath of the processor. The new instruction hence performs typically a new computational task, e.g., the calculation of a multiply-accumulate operation. In order to reduce the hardware overhead introduced by the ISE, our aim is to only extend the already available computational units in the data path (e.g., parts of the ALU), and mainly focus the functional contribution of the new instruction on its microcoded efficient use/addressing of the available resources (such as the register file and ALU) over multiple cycles, in a way that cannot be achieved with the standard ISA. To this end, a new instruction derives for example from a single immediate parameter (encoded in the instruction) and depending on the state (cycle) it is in, which registers to use as operands, and how these should be employed by the extended computational unit.

In the case of cryptographic hash functions the most common computational bottleneck on an MCU lies in the support for large data word rotations. These rotations in general need to be performed on state words which are wider than the native data words, for all algorithms which use this operation, namely BLAKE, Keccak and Skein. As an example, the problem of a 64-bit data word rotation on a 16-bit architecture can be solved in the form of a microcoded multi-cycle instruction, with minimal additions to the processor datapath. Since the the PIC24 16-bit architecture already supports a double register writeback, the 64-bit word rotation instruction can be performed in only two cycles. In each cycle, only a set of three specific 16-bit values need to be shifted to create two resulting MCU native data words, which form the upper or lower half of the 64-bit result. Thus, the required hardware only amounts to two special 16-bit barrel shifters (with 32-bit inputs), and not a full 64-bit barrel shifter. The block diagram detailing this approach is depicted in Figure 2.6. In each cycle of the new multi-cycle instruction, a specific set of 3 operands is loaded, which inherently already performs a shift by either 0, 16, 32, or 48, due to the chosen addressing pattern, depending on the rotation offset. The PIC24 ALU already contains a full 16-bit barrel shifter, which in this case can be extended to allow for the wider 32-bit input, taking two data words as input.[4] The extended shifter is then replicated once, to enable the core to support the 64-bit operation in only two cycles. The emulation of a generic 64-bit rotation by an arbitrary amount of bits utilizing the standard PIC24 ISA requires 12 cycles, which results in a speedup of all (register-based) rotation operations by a factor of 6x with the help of the described ISE.

In general, rotations have to be performed by an arbitrary number of bits. However, if an algorithm such as BLAKE only requires rotations by a small set of constants (i.e., the number

[4] Standard functionality can be preserved by simply using the same operand for both inputs during standard 16-bit rotate operations.

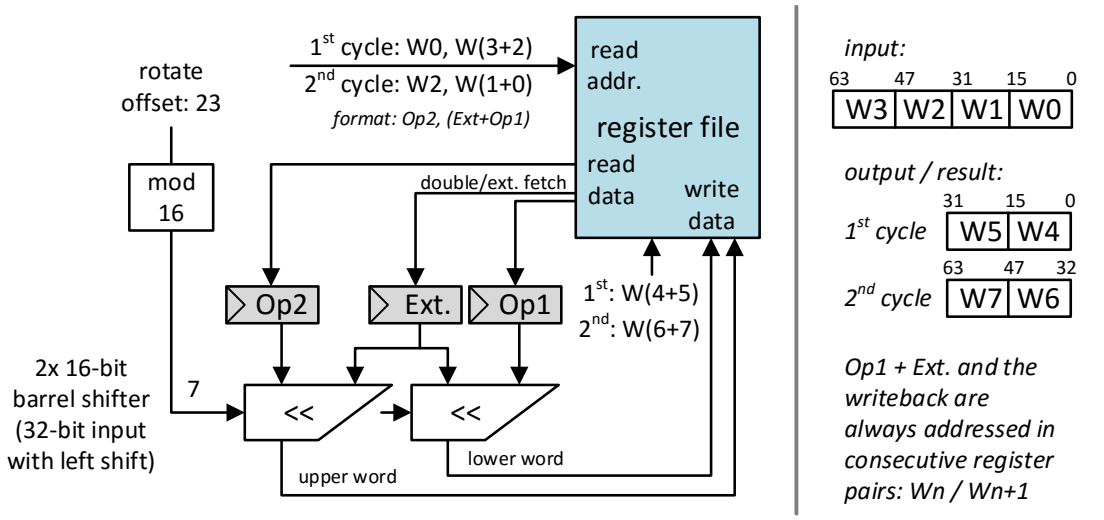


Figure 2.6 – Block diagram of extended 16-bit shifter units within the PIC24 microarchitecture, providing support for a two-cycle 64-bit rotation instruction. The annotated resource/register addressing schedule is an example for a 64-bit left rotation by 23 where W0-W3 are chosen as source, and W4-W7 as destination.

of different rotation offsets is small), a trade-off in favor of shifters which operate only with hard-wired fixed offsets can be made, to reduce the induced hardware overhead of the ISE.

Furthermore, even more complex arithmetic operations such as a matrix multiplication in the field of F_{256} on a 8×8 matrix of 8-bit state words (as required by Grøstl), can be supported by the help of small extensions to already existing computational units. Adding shift units and XOR units on small 8-bit input values can provide a considerable speedup factor of 14.4x for this operation, when combined with the right microcoded multi-cycle operand and memory addressing support.

Another example for an extended computational units is the advanced swap instruction of JH, performing shifts combined with the interleaving of data words (utilizing existing and replicated AND, OR, and left + right shifter units of the ALU). Depending on the mode of the operation, this instruction can also be regarded as an advanced memory addressing instruction, when operating on bit-tuple sizes greater than 8, where the state words can be interleaved on 16-bit word boundaries.

Finite State Machines for Data Address Generation Cryptographic hash functions are heavily based on permutations, as has been discussed in Section 2.1.1. These permutations of state words generally follow some fixed permutation schedules or tables, which are intrinsic to the algorithm definition. Effectively, this means that state data needs to be loaded and stored in predefined regular patterns, but often not in the sequential fashion in which the state words are mapped in memory. The existence of multiple of these access patterns applied

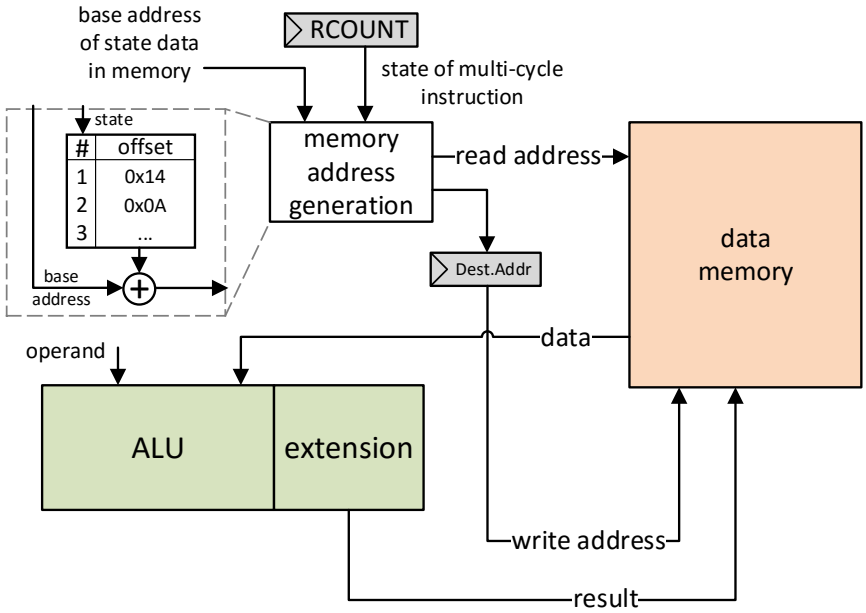


Figure 2.7 – Block diagram of a data address generation module within the PIC24 microarchitecture, providing support for the efficient generation of a state permutation schedule within a multi-cycle instruction. The generation utilizes the available repetition counter (RCOUNT) as its FSM state.

to the same state, often renders general relabeling or physical reordering of state words in the data memory impossible. The assembly implementations therefore have to generate these memory address patterns by the use of additional instructions before the actual memory accesses. Moreover, there is often a trade-off between which data can be loaded efficiently without much overhead at the current position in the algorithm, and which data is actually needed for immediate calculations and can be retained temporarily in the general purpose register file of limited size.

To address this bottleneck, we propose to utilize ISEs which efficiently generate the required data memory addresses, ideally combining this together with a computational operation, which typically precedes or follows the permutation step. The fixed permutation schedule of an algorithm can be seen as a series of memory accesses, which can be represented by a sequence of address offsets relative to a provided memory base address. Similarly to the already available register indirect addressing modes, which allow a given memory address to be incremented, decremented, or have a fixed offset, the new instruction enables a more complex form of memory address modification, based on a fixed sequence of offsets. The memory accesses are typically performed in a tight loop, which can cover the whole internal state or only selected parts of it, depending on the operation. An implementation with improved efficiency is achieved by essentially mapping part of the loop-based program flow in the algorithm core onto a *microcoded multi-cycle instruction*, which accesses the state data in a hard-wired pattern. Based on a small finite state machine (FSM) embedded in the core, the instruction

generates for each state (repetition count) the required data memory address offsets, register indices, and operand data. As depicted in the exemplary block diagram in Figure 2.7, the state information required by the instruction can be derived from an internal loop counter, which can be efficiently realized by reusing existing resouces in the microarchitecture, such as the PIC24's hardware loop counter of its repetition instruction (`REPEAT`). We note that this FSM based method is particularly applicable, if the computational part of the repeated instruction is simple, and performs a transformation of the state data without too extensive computational effort.

ISEs that incorporate the efficient generation of memory accesses can significantly reduce the described type of performance bottleneck, especially on resource constrained systems, which commonly do not have a memory system with advanced addressing features that are geared towards high performance (as sometimes available in DSPs for example for FFT processing). The proposed replacement of program code by an ISE with an FSM-controlled multi-cycle instruction has two effects in general: execution speedup through memory access optimization and considerable code size reduction depending on the repetition count of the new instruction and its comprising computational complexity.

The Grøstl algorithm with very modular structure of its operations, and their rather low individual complexity, can especially benefit from this approach. All four of the core operations of the hash function can be replaced by one single multi-cycle instruction each, utilizing the described ISE type. The BLAKE, JH, and Keccak ISEs also apply this approach for some of their custom instructions, although to a lesser extent. For BLAKE, the approach is used to embed the *Sigma Permutation* tables directly within the address generation, which determine the constant and message data addressing schedule. JH benefits for its *Swap* operation (which shuffles the full state in memory) from this type of advanced address generation, which in this case additionally adapts its generation pattern depending on the round number of the algorithm. While Keccak has a permutation operation *Pi*, this operation can already be efficiently implemented with standard literal offset addressing (as part of a new rotation instruction), and therefore does not require FSM-driven address generation. The *Iota* operation of Keccak however can benefit from dynamic address generation as part of a multi-cycle instruction (with varying cycle count), which optimizes the way how the 64-bit constants (containing mostly 0) are applied to the state in memory.

Lookup Table Integration Due to the large area occupied by on-chip memory, reduction of data memory consumption is often as important as the speedup of algorithm execution. A prime candidate for significant reduction of data memory is the removal of constant lookup tables which are typically stored in data memory. Almost all of the evaluated hash algorithms use some form of LUT or constant table of various sizes, with the exception of Skein, which only requires initial state data (stored as text in the program memory).

Removal of constant table data from memory is possible through integration of these LUTs

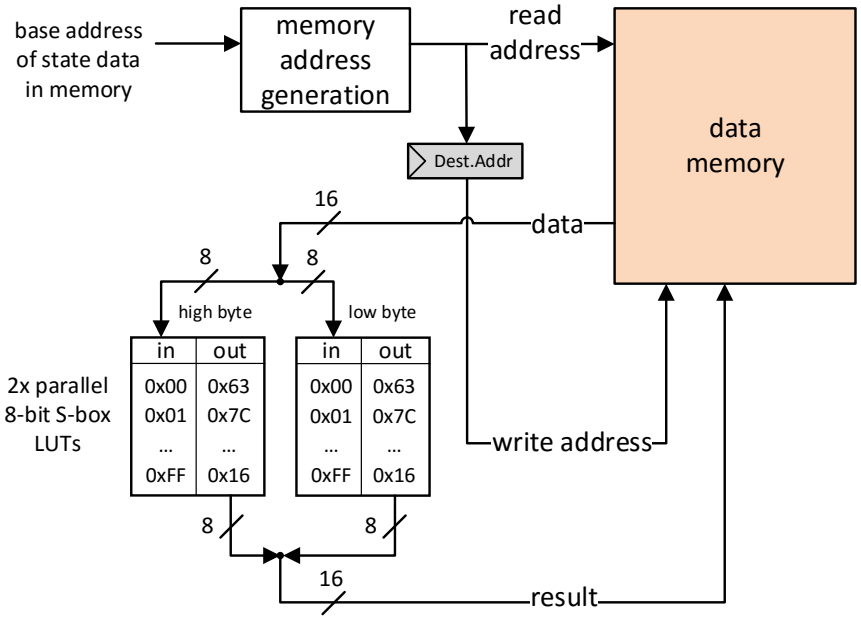


Figure 2.8 – Block diagram of dual S-box LUTs integrated in the execution stage within the PIC24 microarchitecture, providing support for the efficient substitution of the algorithm state data residing in data memory. In this example, two identical S-boxes are implemented in parallel to maximize the throughput of the ISE.

into the processor core. There are typically two different types of LUTs that can be found in hash algorithms. The first type represents tables holding round- or data-dependent constants, i.e., LUTs providing values which are part of the definition of the algorithm. The second type are LUTs which are introduced to speed up execution of the algorithm (often considerably), holding precomputed values of some form. Depending on the size of the LUT and targeted hardware overhead in terms of area, different trade-offs can be explored, especially concerning the second type of LUTs. The indices for these LUTs (i.e., their inputs addressing the entries in the table) are derived from different sources, such as the current round number or state data. An example of a LUT integrating state data-dependent constants within the core is given in Figure 2.8. Here, two S-boxes are added to the data path in the execution stage of the PIC24 microarchitecture to accelerate the state substitution operation of Grøstl.

This class of ISEs has been utilized for all four applicable algorithms, moving all LUTs from data memory into the MCU core. The integration of LUTs in general was possible in an area-efficient manner using only standard synthesis techniques. As an example, the integration of the by far largest LUT of size 1344 byte, which is part of the JH algorithm, only resulted in a negligible core area overhead of around 2 kGE, compared to an SRAM implementation of 14 kGE equivalent area for an equal data size. In this case, the relatively large LUT is used to provide precomputed round constants enabling an efficient bit-sliced implementation of the algorithm, as discussed in Section 2.4.1. In the case of Grøstl, which employs an S-box LUT as

it is used in AES, the integration of two parallel 8-bit LUTs in hardware (as shown in Figure 2.8) not only saves 64 byte of data memory, but additionally facilitates a speedup of the full state substitution operation by a factor of 4.2x.

2.5.3 Algorithm-Specific ISE of SHA-3 Finalists

In this section the developed ISEs for cryptographic hash functions are presented in more detail, including discussions on the resulting individual performance gains[5] and trade-offs. Each of the five hash algorithms has a separate set of custom ISEs, which address the main bottlenecks of the algorithm, inferred after hand-analysis of the optimized assembly code implementations discussed in Section 2.4.1. An overview of all ISEs, the functions they implement, as well as the individual gains they provide is given in Table 2.7, together with a categorization according to the three different ISE types that have been introduced in Section 2.5.2. Note that during development of the ISEs, every opportunity (which promised relevant algorithm speedup or memory savings) has been taken to apply the three presented ISE concepts to each of the algorithms equally. Nevertheless, significant differences in applicability of the concepts to the five algorithm implementations can be observed.

The syntax of the presented new instructions uses the following nomenclature:

- `Ws`: source register; one of the working registers holding the state data to be operated on; can also indicate the first of a set of registers (i.e., when handling 32-bit or 64-bit state words at once with a multi-cycle instruction).
- `Wd`: destination register; analogous to the source register, indicates the working register(s), where the result of the instruction (typically the transformed state data) is stored.
- `Wm`: register holding a memory (base) address; this is in general used to provide the instruction with the base address of the state data in memory.
- `#lit`: literal (number constant); the literals encoded in the instruction word are used for various purposes, such as to indicate rotation offsets, or to address specific state words within a state array/matrix.

As a general remark regarding the implementations of the ISEs for all candidates: If an instruction relies on the information during which round it is executed (e.g., to adjust which constant to choose from a LUT), this instruction-state information is generally read from a dedicated fixed working register (W14 has been chosen for this purpose), which always contains (by convention) the current round number during execution of the whole algorithm kernel. The number of the register providing the round information is therefore not explicitly encoded

[5] Any gains in terms of speedup reported in this sub-section are given relative to the specific function they implement using the standard PIC24 ISA.

Table 2.7 – Overview of proposed instructions and the individual performance gains or data memory savings that they enable, grouped by ISE type. Gains of each ISE are given relative to the specific function they implement using the standard PIC24 ISA. The additional instruction memory savings due to increased code density are not reported here.

	Extension of Computational Units	
	Function realized by ISE	Speedup
BLAKE	32-bit rotation (fixed offsets)	6.0x
Grøstl	*MixBytes* step	14.4x
JH	AND-then-XOR for bit-sliced *S-box*	2.0x
Keccak	64-bit rotation (single bit)	2.5x
	64-bit rotation (w/ permutation in mem.)	4.1x
Skein	64-bit rotation (generic)	6.0x
	FSMs for Data Address Generation	
	Function realized by ISE	Speedup
BLAKE	constant XOR message	4.0x
Grøstl	*AddRoundConstant* step (P/Q)	3.2x/1.5x
	ShiftBytes step	3.4x
	MixBytes step	14.4x
JH	*Swap* permutation	5.2x
	Lookup Table Integration	
	Function realized by ISE	Mem. savings [byte]
BLAKE	initialization constants	64
	Sigma permutations	224
Grøstl	*S-box*	256
	F_{256} Multiplication LUTs	512
JH	bit-sliced round constants	1344
Keccak	round constants	96

into the instruction word, or syntax of the assembler instruction, for that matter. Since the register holding the round information is fixed, no full read port for the register file is required, i.e., the introduced hardware overhead by this optimization choice is negligible.

2.5.3.1 BLAKE ISE

Three dedicated instructions are developed for BLAKE, to help accelerate the handling of initialization constants, 32-bit state word rotations, and the addressing of constants and message data via the *Sigma Permutation* within the *G* function.

- `S3_XCST Wm`
  During the initialization phase of each block compression (i.e., when starting to process

the next message block), eight 32-bit constants have to be XOR'ed onto the internal state (`Wm`). This instruction performs this operation efficiently, while iterating over 16 cycles in conjunction with a preceding `REPEAT #15` statement. In each repetition the MCU treats 16-bit of the state and derives the necessary constant and the memory address offset for reading and writing back the modified state. While the instruction provides a speedup of 2x for this operation, it is not part of the inner algorithm kernel. Hence, the main benefit of the ISE comes through the efficient integration of the constant table into the core, freeing up data memory.

- `S3_RROT_7 Ws, Wd`
  `S3_RROT_8 Ws, Wd`
  `S3_RROT_12 Ws, Wd`
  Rotations are among the most time consuming operations of the reference implementation, since they are a core part of the *G* function. This family of three instructions performs a 32-bit right rotate by a specific amount of either 7, 8, or 12 bits, which are three of the four rotation offsets required by the *G* function. Choosing the three rotation offsets as fixed avoids the need for a full 32-bit barrel-shifter in the MCU, which reduces the incurred hardware overhead significantly. The input data is provided in a register pair (`Ws`) and the result is written to another register pair (`Wd`), using the double write-back feature of the MCU to perform the complete operation in only a single cycle. The fourth type of rotation employed by BLAKE is by 16 bits and can directly be performed efficiently using existing `MOV` instructions.

- `S3_CXM Wm, Wd`
  This instruction calculates the term that combines constant data and message block data (`Wm`) by an XOR, used twice per *G* function call. The instruction internally generates the lookup data from the *Sigma Permutation* table as a memory address offset, as well as the corresponding constant. This measure avoids costly memory accesses with very little area overhead for implementing the constant tables inside the processor. The order of the indices for addressing the message and constant data is derived from a round & index counter. The instruction requires two cycles to complete in order to fetch the 32-bit message block data word from memory.

The instruction set extensions developed for BLAKE reduce the program memory by 20% (see Table 2.1), the data memory by 59%, and the cycle count by 34%, making an already fast and small implementation even faster and smaller. Memory for *text* is saved by a compact way to describe the addition of constants without the need for loop-unrolling, the automated *Sigma Permutation* removing instructions dedicated to address generation, and the compaction of the rotation operations. *Data* is saved by placing all lookup tables into the processor core. Throughput is mainly gained by a speedup of the rotations by a factor of 6x, and efficient *Sigma Permutation* access, which eliminate additional memory accesses for the purpose of fetching indicies and constants data. The additional instructions have no noticeable impact on the

core area compared to the reference implementation as they do not add new computational units, but modify only register and memory accesses for existing operations.

2.5.3.2 Grøstl ISE

The two core operations of Grøstl are *P* and *Q* (which are very similar), each consisting of four suboperations, which are constructed in a very modular fashion, each transforming the complete 64 byte state in memory. These four suboperations can be implemented without performing too complex arithmetic per state element. Consequently, it was possible to develop four instructions for Grøstl, each replacing exactly one of the suboperations (with different flavors for *P* and *Q*, where necessary). All of the new instructions are memory-to-memory multi-cycle instructions, i.e., they modify the full state matrix in place, sometimes utilizing a set of work registers during their operation.

- `S3_CST_P Wm`
  `S3_CST_Q Wm`
  This instruction replaces the *AddRoundConstant* suboperation, which XORs the current round number with the entries of a fixed constant matrix, and then XORs this modified matrix onto the entries of the state matrix (`Wm`). The described steps are performed in a hardware-loop, using the `REPEAT` instruction. The P version of the suboperation has mostly zero-entries in the constant matrix, which allows the use of `REPEAT #7`, to perform the complete transformation in 8 cycles, enabled by state dependent address generation. The Q version on the other hand has to transform every element of the state, requiring the use of `REPEAT #31`, resulting in an execution time of 32 cycles.

- `S3_SBOX Wm`
  This instruction adds the logic for the AES S-box lookup table into the processor core and executes effectively two parallel 8-bit AES *SubBytes* table lookups per cycle including the substitutions. Therefore the complete state matrix (`Wm`) can by substituted in 32 cycles, using a `REPEAT #31` instruction, followed by a single `S3_SBOX` instruction, providing a speedup of 4.2x for this suboperation.

- `S3_SHIFT_P Wm`
  `S3_SHIFT_Q Wm`
  The *ShiftBytes* suboperation is implemented by this instruction, which exists in both *P* and *Q* variants. The instruction is used in conjunction with `REPEAT #35`, performing the complete shift of all state rows (`Wm`) in only 36 cycles. *ShiftBytes* performs shifts/rotations of full rows of the state matrix, aligned on the 8-bit state word boundaries. This means the suboperation can theoretically be implemented solely with byte moves. Our ISE operates on pairs of state elements (2x 8-bit) from two adjacent rows at the same time, since the state matrix is addressed column-wise in data memory. Two different pairs of elements are combined in each repetition (cycle), normally one pair from memory and one pair from the register file, to form a new result pair. This result pair is

constructed by concatenating two elements coming from the two different rows, such that they are rotated to the required position when the result is written back to the state matrix. This concatenation of two separate state words (bytes) is the only required computational extension of this instruction. The actual complexity of the instruction lies in providing the correct memory addressing schedule derived from the repeat counter state, i.e., when to load which pair of elements to perform the entire transformation in the minimum amount of repetitions (cycles). For optimization purposes, the instruction uses registers W0-W7 as temporary work registers to buffer values that have already been overwritten in memory.

- `S3_MIX Wm`
  Most of the performance gain can be attributed to this instruction that performs the *MixBytes* operation. The instruction is used together with `REPEAT #79` and is able to multiply the entire 8×8 state matrix (`Wm`) in $\mathbb{F}_{256}$ with a constant matrix in only 80 cycles. This is achieved by processing the state elements in pairs (subsequent entries of the same column of the state matrix). Each pair is multiplied with all corresponding elements of the constant matrix in only two cycles, internally using new computational units for two 8-bit elements in parallel. This produces after two cycles 8 partial result elements (forming the current column that is transformed) held in 4 working registers. In the subsequent cycles the missing partial results are accumulated before the final result is written to memory. Hence, the theoretical throughput of the operation is 1 element per cycle. The 16 required extra cycles of the instruction stem from phases at the end of the calculations for each column, where only results can be written back to the memory, but no computation can be carried out. With an optimized ordering of the memory addressing pattern, this idle time can be limited to two cycles per column, but cannot be further reduced due to the given memory bandwidth limitations of the writeback. The multiplication is performed by a new functional unit that calculates for each input byte the multiplication results in $\mathbb{F}_{256}$ for all possible factors (2, 3, 4, 5 and 7) by using shifts and XORs on 8-bit values only. These efficiently calculated products are then used to generate the partial sums, according to the currently active submatrix/field of the constant matrix. Since the given constant matrix is circulant, only four different fields have to be distinguished, which reduces the complexity of the hardware implementation. The instruction uses W0-W3 as accumulator registers for the temporary results of the new column values.

The `S3_MIX` instruction has been instrumental in reducing the cycle count for Grøstl, by providing a 14.4x speedup for the corresponding *MixBytes* suboperation. However, this new instruction does not add complex new components to the datapath, since the matrix multiplication can be performed by a series of small XOR operations. Furthermore, even though the ISE is a very long multi-cycle instruction with a total of 80 states, the memory and resource addressing schedule can be implemented extremely efficiently within the MCU core. In effect, the operational flow of the full *MixBytes* step has been moved from regular instructions

stored in the instruction memory to the instruction decoder and implemented as an FSM. Additionally, due to the new computational units, there is no need anymore for the loading of precomputed products from data memory, which allows to maximize the achievable memory bandwith utilization for the operation, further accelerating it.

Overall, it was possible to reduce the cycle-count of Grøstl by more than 87% with these additions, making the ISE modified implementation of Grøstl by far the fastest of the implementations of the SHA-3 candidates. Moreover, data memory is reduced by 75% (see Table 2.2), due to cutting of the LUTs originally required for the *MixBytes* multiplications, and the move of the S-box LUT to the core. Instruction memory is also drastically reduced by 69%, since all four suboperations are replaced by single instructions. The area overhead for these ISEs is only 10% (2 kGE), relative to the baseline MCU core, i.e., excluding memories.

2.5.3.3 JH ISE

The following two instructions are added to the instruction set to improve the performance of interleaving state data during the *Swap* permutation operation, as well as to remove the large round constant LUT from data memory. This LUT is required for a bit-sliced implementation, and utilized during the *S-box* operation of JH.

- `S3_CST_ACXM #index, Ws, [Wm], Wd`
  `S3_CST_AMXC #index, Ws, [Wm], Wd`
  This instruction combines two operations used during the calculations for the JH-specific 4-bit *S-box* (bit-sliced, algorithmic version).[6] It first performs a lookup using a LUT built into the processors datapath, holding all round constants, depending on the current round number and a position index (0..15) passed as a literal (`#index`). This constant is then used in a combined AND-then-XOR operation, using additionally one source register (`Ws`) and one data word of the state read from memory (`Wm`) as the operands. By extending the ALU with a computational unit that provides an optional pre-AND to the existing XOR, the instruction is able to be executed in a single cycle. The result is stored in a register (`Wd`), since it is required for further processing in the *S-box* algorithm. The instruction has two different variants, both used once per *S-box* call, effectively exchanging the order of the operands (the constant and the data word from memory). `S3_CST_ACXM` calculates the term $\texttt{Wd} = (\texttt{Ws} \wedge \text{const}(\texttt{\#index}, \text{round})) \oplus \texttt{[Wm]}$, while `S3_CST_AMXC` calculates the term $\texttt{Wd} = (\texttt{Ws} \wedge \texttt{[Wm]}) \oplus \text{const}(\texttt{\#index}, \text{round})$.

- `S3_SWAP Wm`
  This instruction is used in conjunction with `REPEAT #31` to carry out the *Swap* operation on the full state (`Wm`) in 32 repetitions. The parameter of the swap (the size of the bit-tuples to be swapped, 1, 2, 4, 8, 16, 32, or 64) is thereby derived from the round counter

[6] The *S-box* output is calculated here, instead of being translated by a table, as this is more efficient in the case of JH, specifically when processing multiple inputs (16) in parallel, because of the chosen bit-slicing approach.

value which is taken from a fixed register (W14). For swaps with tuple sizes smaller or equal to 8, the instruction utilizes a computational extension of the ALU, providing bit selection with fixed shifting, which allows to perform a swap on each 16-bit data word via reshuffling and interleaving of the bits within the word. In rounds where the tuple sizes are greater or equal to 16, the instruction bypasses the swap unit in the ALU and instead provides the appropriate memory read and write addresses via a micro-coded schedule, to efficiently execute the permutation of the state array.

Our implementation results with ISEs show a reduction of cycles/byte of 17%, mostly due to the improved *Swap* operation. However, the main benefit comes from the reduction of costly data memory by 87% (see Table 2.3), since the complete LUT containing the round constants (1344 byte) has been moved into a hardwired LUT in the core, where it only generates an area overhead of about 10% (2 kGE). *Text* is reduced by 53% due to the drastic simplification of the different flavors of the *Swap* operation into a single instruction.

2.5.3.4 Keccak ISE

The main bottleneck of the algorithm on a 16-bit architecture are the 64-bit rotations, which need 12 cycles per rotation for the generic case (i.e., for an arbitrary rotation offset) in the reference implementation. Hence, this operation is the main focus of the ISEs. Furthermore, the round constants table is moved into the core in a compressed form to reduce *data* consumption.

- `S3_LROT1 Ws, Wd`
  This instruction performs a 64-bit left rotate by a fixed amount of 1 bit, as required by the *Theta* step. The ISE executes in two cycles and uses a set of four consecutive registers as source (`Ws`), and a set of four other consecutive registers as destination (`Wd`). In each cycle three of the four registers are used as operands to produce either the higher or the lower half of the 64-bit result. The instruction utilizes the double writeback capabilities of the architecture, and causes almost no hardware overhead for its integration, due to the fixed shift offset.

- `S3_LROTM_A #rot, #coord, Wm`
  `S3_LROTM_B #rot, #coord, Wm`
  This instruction implements a generic 64-bit left rotate by an arbitrary number of bits (`#rot`). The operation works on a set of four source registers and writes the result to a memory location (`Wm`) defined by coordinates in the 5x5 state matrix, given as a literal (`#coord`). The execution takes 4 cycles and the instruction performs the additional function of saving the data which is located at the destination memory address to another set of registers, before writing the rotation result to memory. Since the instruction writes to data memory, it is limited by a throughput of 16 bit per cycle. As a result the required computational extension for the instruction only amounts to the extension of the

available 16-bit barrel shifter to support a 32-bit input (see Section 2.5.2 and Figure 2.6). Replication of the shifter is hence not required to achieve the maximum throughput. The instruction exists in two variants, differing only in the set of registers used for rotation (input state data) and for buffering/backup storage. `S3_LROTM_A` rotates the content of W0-W3, and loads the memory content at the same time into W4-W7, while `S3_LROTM_B` rotates W4-W7 and loads into W0-W3. Using these two variants in alternating order allows to effectively store the result of the currently rotated state element, while at the same time already pre-loading the next element of the permutation chain, making it possible to very efficiently combine the *Rho* and *Pi* operations (rotations and permutations) over all 25 elements of the state matrix.

- `S3_XCST Wm`
  This instruction applies the current round constant via XOR onto the internal state element at (0,0) (`Wm`), performing the *Iota* step. The round constants are 64-bit values containing a significant amount of bytes with value zero. The table is hence stored in a compressed format as synthesized logic within the core, to reduce the hardware overhead. Execution time varies between one and three cycles, depending on the constant value, since possible zero bytes of the value are not applied.

The above described ISEs for Keccak result in a reduction of the number of cycles by 30%, which can mainly be attributed to the improved efficiency of the 64-bit rotation, as well to the fact that rotations are now combined directly into the element chaining of the *Pi* state permutation. The *data* size is reduced by 21% (see Table 2.4), by placing the round constant table into the core. The *text* size is lowered by 31%, through compact description of the combined *Rho* + *Pi* operation in a single instruction. All three extensions cause virtually no core area overhead, since they contain no large LUTs and do not add considerably to the datapath. The round constant lookup table can be implemented in the core in an even more compressed way, compared to the standard implementation, only utilizing 24x11 bits = 33 byte of information.

2.5.3.5 Skein ISE

Profiling of the reference implementation showed that large gains can be achieved by removing every unnecessary cycle from the *Mix* function core of the *ThreeFish* block cipher. This lead to the addition of one instruction, performing the rotation in an efficient way, providing a general speedup of 6x for the rotation function. Due to the general simplicity of the algorithm core (which was an explicit design goal [FLS+10] of the algorithm), all other operations were identified as sufficiently well supported by the reference PIC24 ISA.

A critical point with Skein in terms of *text* is the unrolling of all 9 different *subkey* injections. Even though the required memory for this operation is significant (compare Table 2.5), it was not possible to develop an ISE that would for example support a single block of code that allows the *subkey* injection to be executed efficiently in the required 9 different modes, without

reduction of the hashing performance. This is due to the fact that any additional checks or looping during the injection (with the aim of code reuse) would introduce significant overhead in terms of cycles for the operation.

- `S3_LROT #rot, Ws, Wd`
  This instruction performs a generic 64-bit left rotate by an arbitrary amount of bits (given as a literal `#rot`) in two cycles, using a set of four registers as source (`Ws`) and a set of four other registers as destination (`Wd`). The double writeback feature of the microarchitecture is used for the instruction. Since the instruction uses two cycles to produce the result, in each cycle using three specific input values, the added logic for the computational extension corresponds to an additional 16-bit barrel shifter (as illustrated in Figure 2.6), instead of a full 64-bit barrel shifter, and is therefore very small.

Besides the simplicity of the *Mix* function, which did not justify any additional ISEs, the algorithm also does not utilize any sort of LUT that could be moved into the core. Furthermore, the address arithmetic needed for performing the permutations and selection of state words is quite straightforward and does not warrant any special supporting instructions, since possible performance gains would be minimal.

Nevertheless, with a single carefully chosen instruction set extension the results show an improvement in throughput of 29%, with virtually no core area overhead. The *text* size is reduced by 18%, but still the largest implementation of all considered algorithms, because of unrolling of the complete key injection schedule for all 9 cases (mainly caused by the round cycle length of 9, which is not a power of 2). The *data* size did not change, but is already small due to the lack of LUTs, as can be seen in Table 2.5.

2.6 Performance on PIC24 with ISEs

In the following, the performance results of the different PIC24 implementations with their respective ISE enhancements are compared and further discussed. The results cover all software performance aspects, i.e., hashing speed and memory consumption improvements, as well as a detailed assessment of the induced hardware overhead due to the proposed ISEs. The section is then concluded by a discussion of the effects on energy consumption from a system perspective.

2.6.1 Core Performance and Execution Speed

By developing the presented ISEs for the PIC24 ISA we are able to provide an average speedup of 1.4x for the slowest 4 out of 5 SHA-3 hash algorithm candidates, with the exception of Grøstl for which we achieve a speedup of 8x. Table 2.8 shows these results in detail and compares the overall low area overhead that the ISEs incur to the reference microcontroller

Table 2.8 – Improvement of hashing speed (long message performance) and respective core area due to ISEs, for all SHA-3 candidates.

Algorithm	**Cycles per Block**		**Cycles/byte**			**Core Area [kGE]**	
	PIC24	w/ ISE	PIC24	w/ ISE	Speedup	w/ ISE	Overhead[a]
BLAKE	9,933	6,583	155.2	102.9	**1.51x**	22.77	**-0.5**%
Grøstl	29,585	3,685	462.3	57.6	**8.03x**	24.97	**+9.1**%
JH	29,683	24,541	463.8	383.5	**1.21x**	25.20	**+10.1**%
Keccak	25,613	17,908	188.3	131.7	**1.43x**	22.22	**-2.9**%
Skein	10,084	7,204	157.6	112.6	**1.40x**	23.00	**+0.5**%

[a]Reference PIC24 core area: 22.88 kGE at 200 MHz.

implementation without ISEs. For an in-depth evaluation of the area and timing requirements of the implemented cores refer to Section 2.6.3.

It can be observed that for 3 out of 5 candidates, the ISEs do not result in any noticeable core area overhead, while still providing significant improvements, in both cycle-count and memory requirements (see Table 2.10). This is mainly due to the fact that some ISEs do not actually add datapath components, but provide small FSMs that are able to resolve complex but regular addressing schemes to fetch operands for already present datapath components (XOR, AND, ADD). The smallest speed up that is obtained through the ISEs is for JH (1.21x), whereas the largest improvement with a speedup factor of 8.03x is obtained for Grøstl. Interestingly, these two implementations both result in about 10% area overhead for the MCU core, which is still negligible (especially when considering that overall area is often dominated by the on-chip memories).

One important parameter to investigate, when determining how much faster an algorithm can be implemented, is the number of memory accesses (read/write) required for the algorithm. Since the memory bandwidth for the given microarchitecture will not change with additional instructions, the program will always be limited by these numbers. Part of the speedup is achieved by eliminating memory accesses, mostly by embedding operational constants into the ISE, and optimization of the load- and store-patterns for state words. In Table 2.9 we list the number of read and write memory accesses for all candidate algorithms. When comparing the memory accesses to the total number of cycles listed in Table 2.8, note that the PIC24 architecture allows concurrent read and write operations to the data memory within the same cycle. It can be seen that only for Grøstl a very significant improvement could be made. This also explains the relatively high performance gain for this algorithm. Not only the read access, but also the write accesses of Grøstl could be significantly reduced, mainly due to the possibility to implement a highly optimized addressing schedule within the core for the *MixBytes* suboperation, which greatly reduced the number of required store operations for temporary partial results during the matrix multiplication. For Keccak and Skein, on the other hand, we could not find any instructions that would be able to reduce the number of memory accesses.

Table 2.9 – Change in the number of memory accesses, both for read and write, during the processing of one message block due to the introduction of ISEs, for all SHA-3 candidates.

Algorithm	**Read**			**Write**		
	PIC24	w/ ISE	Change	PIC24	w/ ISE	Change
BLAKE	2,370	1,682	**-29%**	1,187	1,187	**0%**
Grøstl	16,566	3,126	**-81%**	13,271	2,391	**-82%**
JH	11,836	9,874	**-17%**	4,141	4,099	**-1%**
Keccak	14,660	14,779	**0%**	7,345	7,416	**+1%**
Skein	5,264	5,264	**0%**	3,289	3,289	**0%**

2.6.2 Memory Consumption

As discussed earlier in the performance metrics section (Section 2.3), in the best case, around two thirds of a comparable embedded MCU system consist of program and data memory. Hence, from a system designers point of view the amount of memory used by an algorithm can sometimes be even more important than its outright execution speed. We have listed the total data and program (text) memory used by all candidate algorithms separately for both the standard and ISE-enhanced implementations of the PIC24 in Table 2.10. The achieved memory reductions for each algorithm, regarding their data structures and functions, are reported in the memory overview tables of Section 2.4.1, specifically in Table 2.1 (BLAKE), Table 2.2 (Grøstl), Table 2.3 (JH), Table 2.4 (Keccak), and Table 2.5 (Skein).

It can be seen that the largest combined improvement (*data* + *text*) is achieved for JH (61.5% total reduction) and Grøstl (71.3% total reduction), whereas the smallest improvement is achieved for Skein (17.7% total reduction). *Text* is generally saved through the compaction of more complex algorithm parts into single instructions, especially by mapping address generation patterns onto FSM-based instructions. In the case of frequent use of rotations, their single instruction representation further reduces the program code size. The data memory

Table 2.10 – Reduction of data and instruction memory requirements by using instruction set extensions, for all SHA-3 candidates.

Algorithm	**Data [byte]**			**Text [byte]**		
	PIC24	w/ ISE	Reduction	PIC24	w/ ISE	Reduction
BLAKE	488	200	**-59%**	1,028	818	**-20%**
Grøstl	982	214	**-78%**	2,619	819	**-69%**
JH	1,550	206	**-87%**	4,649	2,183	**-53%**
Keccak	448	352	**-21%**	3,480	2,415	**-31%**
Skein	242	242	**0%**	5,734	4,678	**-18%**

footprint is reduced by integrating round constants and other lookup tables into the processor core. The amount of *data* required by the different algorithms is mostly evened out by the introduction of the ISEs to a level of about 200 byte, with the exception of Keccak, which has a higher requirement due to the both larger internal state and message block.

2.6.3 Hardware Overhead

A general overview of the hardware area requirements for all implemented cores is given in Table 2.8. The area of our reference PIC24 MCU core without ISEs is 22.88 kGE (with an achieved target clock speed of 200 MHz), while the maximum observed overhead due to ISEs is only 2.32 kGE (+10.1%). The digital design flow is known to produce results that can vary as much as ±5%, hence changes in area smaller than 5% can typically not reliably be reported. In fact, for 2 out of the 5 algorithms, the resulting core area for the processor after full synthesis actually decreases slightly when ISEs are added.

The generated SRAM macro blocks used for evaluation and complete synthesis of the designs in this work are of the sizes 2048 instructions (6 kbyte) and 1024 data words (2 kbyte), together occupying an area of 50.28 kGE. As such, the net-overhead due to the ISEs is even smaller. Factoring in the possible system memory size reductions due to improved code size and usage of working data, the total MCU system area would potentially even decrease.

Enhancing the processor datapath can have two consequences. First of all, in most cases, additional logic is added to the datapath, increasing the overall area of the processor. In some cases, the additions may also increase the critical path of the processor. Most synthesizers are able to exploit various techniques to trade off area versus speed, and can compensate for the increased delay by increasing the area further. However, in some cases the increase is simply too much to be compensated, resulting in a circuit that is not only larger than the original, but also slower.

For assessment of the design corners, emphasizing optimization for high speed or low area, the reference core as well as all five modified versions, each including their algorithm-specific ISEs, are implemented and synthesized for various timing constraints. As shown in Figure 2.9, we perform an area-delay trade-off evaluation for all six designs by synthesizing each architecture multiple times while sweeping the core clock constraint, and we report the achieved clock speed after static timing analysis.

Optimization for area shows a size of 19.0 kGE for the reference core as well as the core with Keccak ISEs. The cores with BLAKE ISEs and Skein ISEs each occupy a minimum area of 19.5 kGE, while Grøstl ISEs and JH ISEs each produce extended minimal core sizes of 21.4 kGE, due to integration of relatively large LUTs in the processor datapath. All designs optimized for area can still operate at a core clock frequency of about 70 MHz.

Tightly constrained synthesis, i.e. optimization for speed, shows a similar picture of the designs performing on a comparable level regarding maximum clock frequency and required core area.

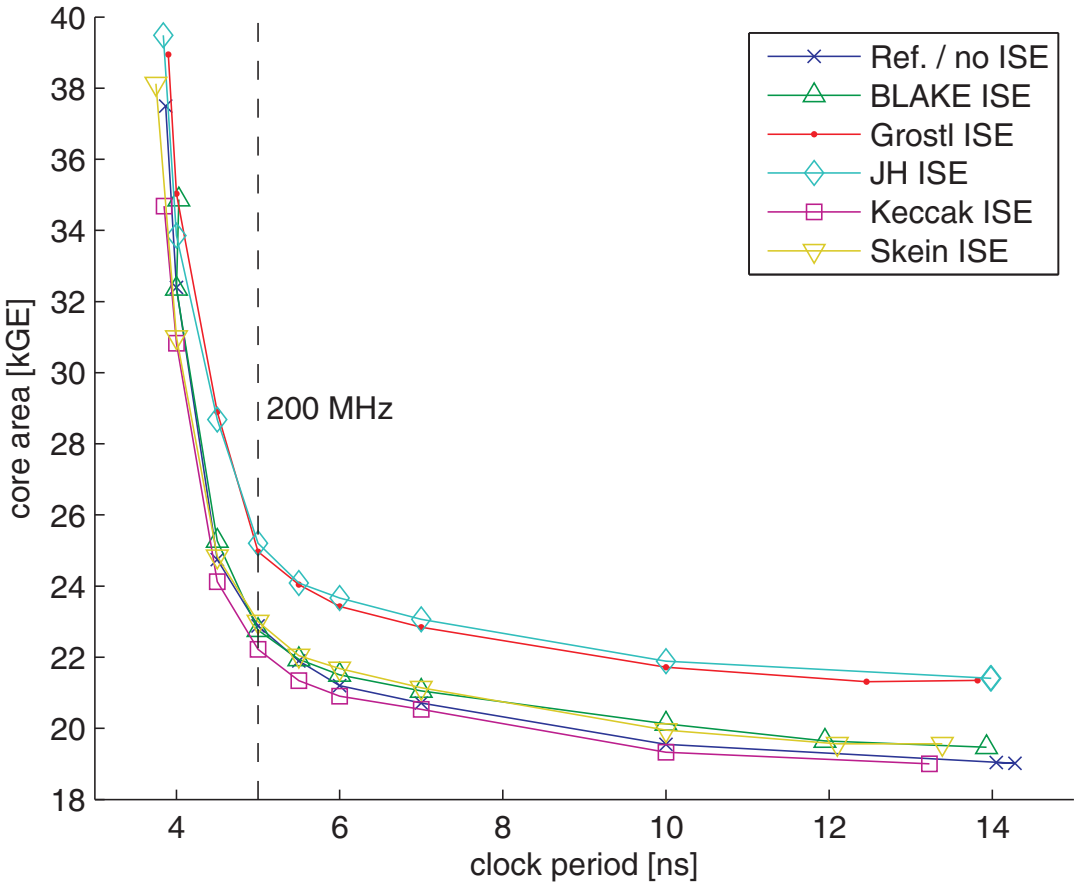


Figure 2.9 – Area of synthesized core versus maximum core clock period for the reference PIC24 design and five cores with individual SHA-3 candidate ISEs each.

Maximum frequencies are around 260 MHz, with the exceptions of the two cores with BLAKE ISEs (248 MHz) and Keccak ISEs (267 MHz). The occupied area for the high speed designs varies between 34.7 kGE (Keccak) and 39.5 kGE (JH), around the reference core without ISEs (37.5 kGE). These figures can mainly be explained by the mentioned optimization-variations during synthesis, which seem to be more pronounced due to aggressive over-constraining of the clock period. This phenomenon can especially be observed for the core with Keccak ISEs, which is identical to the reference core plus three additional instructions.

As a result, it can be seen that the area overhead produced by the introduced ISEs is in general negligible, independent of the clock constraint.

To give the synthesizer some room for exploiting various techniques to trade off area versus speed, the reference clock period for general overhead comparison is chosen at 5 ns (200 Mhz), about 1 ns above the achievable minimum, hence allowing for all six designs to meet the same timing constraint. The difference in core area can this way be attributed to the overhead due to the ISEs only.

2.6.4 Energy Considerations

In the context of the discussed MCU-type systems, the achievable energy efficiency is increasingly the key criterion for the design of such a system. Under nominal conditions, any savings in necessary processing cycles directly translate into saved energy for the completion of a given task, in this case the hash digest calculation. Additionally, when the required maximum throughput of the application utilizing the hash function is known, voltage scaling can be employed to adjust the supply voltage to an optimal point, such that the throughput is still satisfied and the energy consumption of the MCU system is minimized. The synthesis results presented in 2.6.3 indicate that the impact of the introduced ISEs is minor and especially does not negatively impact the critical path of the MCU cores with ISEs.[7] This is relevant for the evaluation of the overall energy benefits of the ISE-enabled cores, since consequently voltage scaling can be applied in an equal fashion to both the reference PIC24 core as well as the extended cores. Hence, the amount of voltage scaling that can be performed (while keeping an equal clock frequency for all cores) is likely to be very similar for all considered designs.[8] Note that if the clock frequency for the different cores/algorithms can be freely adjusted in order to meet a fixed deadline for the algorithm execution, differences in the amount of available voltage scaling are of course present, as will be discussed later in some more detail.

Another aspect is that embedded MCUs are geared towards very low power consumption during idle phases. Any commercially available device typically supports some type of sleep mode (often multiple, with differences in the wake up time/effort), which allows to conserve energy whenever no processing is currently required from the MCU. The MCU ideally only consumes leakage power when in sleep mode, which is orders of magnitudes lower than the dynamic power, at nominal operating conditions. If the hashing task can hence be executed in fewer cycles due to the introduced ISE, any savings in active energy can be regarded directly as savings in overall system energy, since the MCU consumes significantly less energy during its sleep phases.

Figure 2.10 compares the energy consumption of the five different SHA-3 hashing algorithms, and shows the impact of the introduction of our ISEs, individually for each modified PIC24 core. The energy values are based on an estimated average consumption for all cores of 30 pJ/cycle @ 1.2 V[9] with a clock speed of 200 Mhz. All cores are assumed to have the same average energy efficiency, since the overall microarchitectures of the different cores are almost identical, hence would only marginally differ, especially compared to the energy differences due to the reduced cycle counts. The energy consumption only refers to the MCU core, excluding the instruction and data memories, whose consumption can differ widely, depending on the chosen sizes.

[7] The core with BLAKE ISEs is a slight exception, seeing a small degradation of 5% in the maximum achievable clock frequency.

[8] It is possible that the gate compositions on the critical path of the different cores differ, while having a similar maximum clock frequency, depending on the specific design. Consequently, differences in the delay degradation with voltage scaling are possible in this case.

[9] Nominal supply voltage of the employed 90 nm CMOS technology.

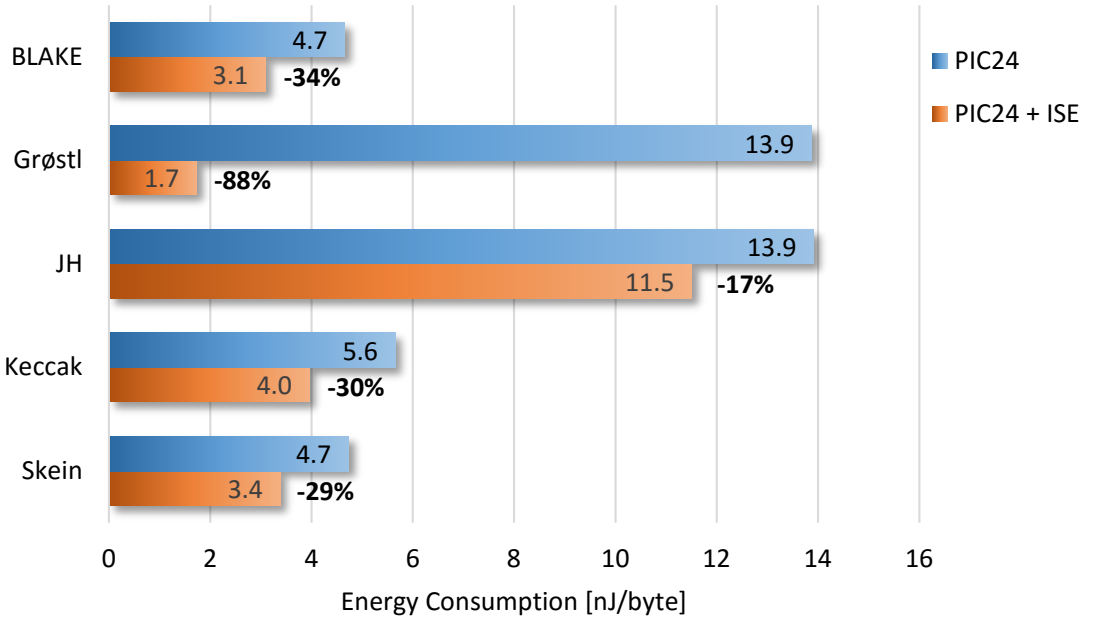


Figure 2.10 – Comparison of average energy consumption per processed message byte for the reference PIC24 design and the achievable savings due to ISEs, for all SHA-3 candidates.

Additional power savings over the ones indicated in Figure 2.10 are possible by potentially utilizing voltage scaling to reduce the supply voltage and with it the maximum operating frequency, as mentioned. Note that this approach either requires that the hashing function is the limiting task regarding throughput running on the MCU, or that DVFS is available and feasible to be applied for the duration of the hashing task. In the case where voltage scaling can be utilized, the hashing algorithms and respective cores can be compared using a fixed iso-throughput (bytes per second), while operating at algorithm-specific optimal frequencies and voltages. Especially the core with Grøstl ISEs is expected to disproportionately gain in terms of energy savings, since it can be operated with about half the clock frequency, while still providing the same hashing throughput as the other candidates. JH on the other hand would disproportionately lose, since it requires even in its ISE-enhanced version a 3-4 times higher clock frequency to achieve the same hashing throughput as the other candidates, and hence would only tolerate a much smaller voltage reduction.

Moreover, for the cases where the MCU is operated in near- or sub-threshold regions, the introduced ISEs can enable even further savings on system level. Since leakage power becomes significant, or even dominant for very low supply voltages, any reduction in area is crucial to minimizing power consumption. Especially the SRAM can be a large contributor to the overall leakage power consumed. As a result, the significant reduction in memory requirements due to our ISEs allows to either reduce the integrated amount of SRAM or provides the possibility to use power gating strategies, powering down unused banks, to minimize leakage during operation of the hashing kernels.

3 Microprocessor Design for Deeply Embedded Ultra-Low-Power Processing

As indicated in Chapter 1, there are an increasing number of applications targeted towards small sensing devices operating on battery and/or supplied by energy harvesting devices, which require a long life time. For example, in industrial settings, operation of many years (often >10 years) on a single battery charge is desired to reduce maintenance costs. Similarly, widespread adoption and feasibility of smart home appliances largely depend on their ability to provide advanced functionality under stringent power constraints. Finally, also other sectors such as personal healthcare & fitness strive towards significant lifetime extensions for their battery-powered devices to increase user comfort. With increasing demands towards heavier on-node processing to support richer features on such devices, the architectural choices regarding the employed microprocessors become more relevant and provide many opportunities for power reductions.

In this chapter, the TamaRISC architecture is presented, a custom 16-bit deeply embedded processor core for ultra-low-power (ULP) applications. The motivations for the design choices of the core are given in Section 3.1, together with a description of the ISA and the microarchitecture, followed by a brief overview of the available toolchain and features on the software side. In Section 3.2 a sub-threshold processing system incorporating TamaRISC is presented. Detailed performance and power numbers of a placed & routed TamaRISC core with sub-threshold memories are provided, together with a case study for a compressed sensing application. Furthermore, the baseline TamaRISC microarchitecture is enhanced with an ISE for this application to significantly decrease the system power consumption. Section 3.3 then provides an overview of the state of the art for embedded low-power microprocessors to put our TamaRISC core into context, while Section 3.4 discusses the use of TamaRISC in a multi-core context for WSN applications and presents microarchitectural memory management techniques to improve energy efficiency in this context.

3.1 TamaRISC: A 16-bit Core for ULP Applications

Dogan et al. [DCA+12, Dog13] have shown that a careful microprocessor architecture choice and implementation can result in significant power savings over commercial off-the-shelf devices for biomedical applications, specifically for embedded biosignal analysis. After working with optimized miroarchitecture implementations[1] of a common 16-bit MCU ISA (PIC24 [Mic09a]), the question regarding the impact of the ISA on the studied applications arises, especially concerning energy consumption.

To explore the impact of the ISA, and consequently the impact of the microarchitecture implementing the ISA, a new clean-slate ISA is designed. In the following, we shall refer to this design as TamaRISC. The design goal is the creation of a baseline 16-bit instruction set which allows the microarchitecture of the core to be as simple as possible. While the aim is towards simplicity in terms of supported ALU operations, and in general number of instructions, operand addressing modes which can effectively support signal processing applications are still included, similarly to the PIC24 ISA. This design choice is made to not only keep code size compact to reduce memory requirements, but also to lower cycle requirements (as fewer instructions need to be spent to explicitly access the data memory), since instruction fetch can be one of the major contributors to total active energy consumption [DCR+12]. Consequently, TamaRISC is not a RISC-like ISA in the classical load/store architecture sense, since it has memory addressing built-in for each instruction, for both source and destination operands. As such, it follows in terms of operand addressing more classical MCU-like ISAs/architectures. However, due to the severely reduced instruction set, and because of the general design idea of "simplicity is key", the connection to RISC is made, referring to the original meaning of reduced instruction set computing.

As a direct result of the reduced ISA, the TamaRISC core can improve upon the previous hand-optimized PIC24-compatible design with a 30% increase in speed, while at the same time providing a 40% reduction in core area.[2] When operated at very low supply voltage (down to the sub-threshold regime), the significantly reduced area saves considerable leakage power, while the higher clock frequency typically allows for more aggressive voltage scaling to even lower operating points. Both cores comprise a register file with identical size, and require a similar amount of cycles for the execution of the targeted signal processing applications, with sometimes a slight advantage for the TamaRISC core (up to 15% fewer cycles), even with its reduced ISA, utilizing only a basic C compiler for TamaRISC and an established/commercial compiler for the PIC24 reference design. As a result, energy consumption is reduced by 35-55% with the TamaRISC architecture, compared to the PIC24-based architecture [DCA+12], depending on the target application.

The following subsections describe the details of the TamaRISC ISA and microarchitecture,

[1] The two considered PIC24 microarchitectures have been optimized in terms of data bypassing (as mentioned in Section 2.2.2), and were written in VHDL and LISA, both targeted at generation of efficient hardware.

[2] Comparison is for synthesis results in a 180 nm CMOS technology, with identical memory interfaces/employed SRAM macros.

and give an overview of its software toolchain. More detailed performance and power results for a fully implemented TamaRISC core (down to the layout level) targeted at sub-threshold operation, as well as more in-depth comparison with other embedded microprocessors are further provided in the subsequent sections of this chapter.

3.1.1 Instruction Set Architecture

The ISA comprises a total of 17 base instructions, with 8 arithmetic logic unit (ALU) instructions, 2 general data move instructions, 2 program flow instructions, 4 control instructions (related to: sleep mode, interrupts, and the status flags), and an instruction to provide basic hardware loops. All instructions generally work on 16-bit data words, and some of the base

Table 3.1 – Instruction set of TamaRISC, consisting of 17 base instructions.

Type	Instruction	Variant/Mode	Description
ALU	ADD	-	addition
		ADDC	addition with carry
	SUB	-	subtraction
		SUBC	subtraction with borrow
	AND	-	logical AND
	OR	-	logical OR
	XOR	-	logical XOR
	RS	-	logical right-shift (high bits = 0)
		RSA	arithmetic right-shift (high bits = MSB)
	LS	-	left-shift
	MUL	-	unsigned multiplication (32-bit result)
		MULS	signed multiplication (32-bit result)
move	MOV <reg>	-	move between reg./mem. and reg./mem.
	MOV <imm>	-	move/load 16-bit immediate to reg.
program-flow	BRA	-	conditional branch (15 modes/cond.)
	GOTO	-	unconditional branch/jump
		GOTO IRET	return from interrupt
		CALL	jump with link register (R15 = PC)
control	SLP	-	enter sleep mode (clock-gate core)
	*I	EI	enable interrupts
		DI	disable interrupts
	MOVS	-	read/write status flags & IRET addr.
	NOP	-	no operation
loop	REP	-	repeat following instruction X times

Table 3.2 – Operand addressing modes of the TamaRISC ISA.

Syntax	Addressing Mode
`Rn`	register direct
`*Rn`	register indirect
`++*Rn`	register indirect with pre-increment
`--*Rn`	register indirect with pre-decrement
`*++Rn`	register indirect with post-increment
`*--Rn`	register indirect with post-decrement
`@litRn`	register indirect with literal offset
	(only with `MOV<reg>`; 4-bit signed offset)
`#lit`	4-bit immediate value (16-bit for `MOV<imm>`)

instructions provide different modes of execution. A full listing of the instruction set is given in Table 3.1. The ISA also defines a 16-entry register file named `R0...R15`, with `R15` set as the link register.

The ALU supports addition, subtraction (each with optional carry/borrow), logical AND, OR and XOR, right (arithmetic or logic) and left shift, as well as full 16-bit by 16-bit multiplication (32-bit result) on unsigned and signed data. All ALU instructions work on two source and one destination operands, using the exact same addressing mode options for each instruction, which helps to reduce complexity of the architecture, since the operand fetch logic and the arithmetic operation can be completely decoupled. The supported addressing modes are register direct, register indirect (with pre- or post-increment and decrement) as well as register indirect with offset. The second operand also supports the use of 4-bit literals. Table 3.2 provides an overview of all addressing modes, together with the respective assembly syntax. Moreover, register indirect modes can be used both for one source operand and at the same time for the destination, allowing efficient memory-to-memory operations.

Regarding program flow instructions, branching is possible in direct and register indirect mode, as well as by offset with 15 different condition modes (dependent on the processor status flags: carry, zero, negative and overflow). The ISA also includes instructions for interrupt and sleep mode support of the core. The sleep mode allows external clock-gating of the entire core, until a wakeup event/interrupt occurs.[3]

Furthermore, the instruction word encoding uses 24 bit, and is in general designed as regular and simple as possible (fixed bit positions for fields), with the aim to provide a good starting point for a hardware implementation with low-complexity decoding logic.

[3] An interrupt request is for example triggered by a new ADC sample in a sensor node application.

3.1.2 Microarchitecture

The microarchitecture of the TamaRISC core is depicted in Figure 3.1. TamaRISC is implemented as a Harvard architecture, with a 3-stage pipeline, comprised of a fetch, decode and execute stage. Instruction words are 24 bit wide, with every instruction using only a single 24-bit word. The core operates on a data word width of 16-bit, comprises 16 physical general-purpose working registers and 3 external memory ports, one for instruction fetch, one for

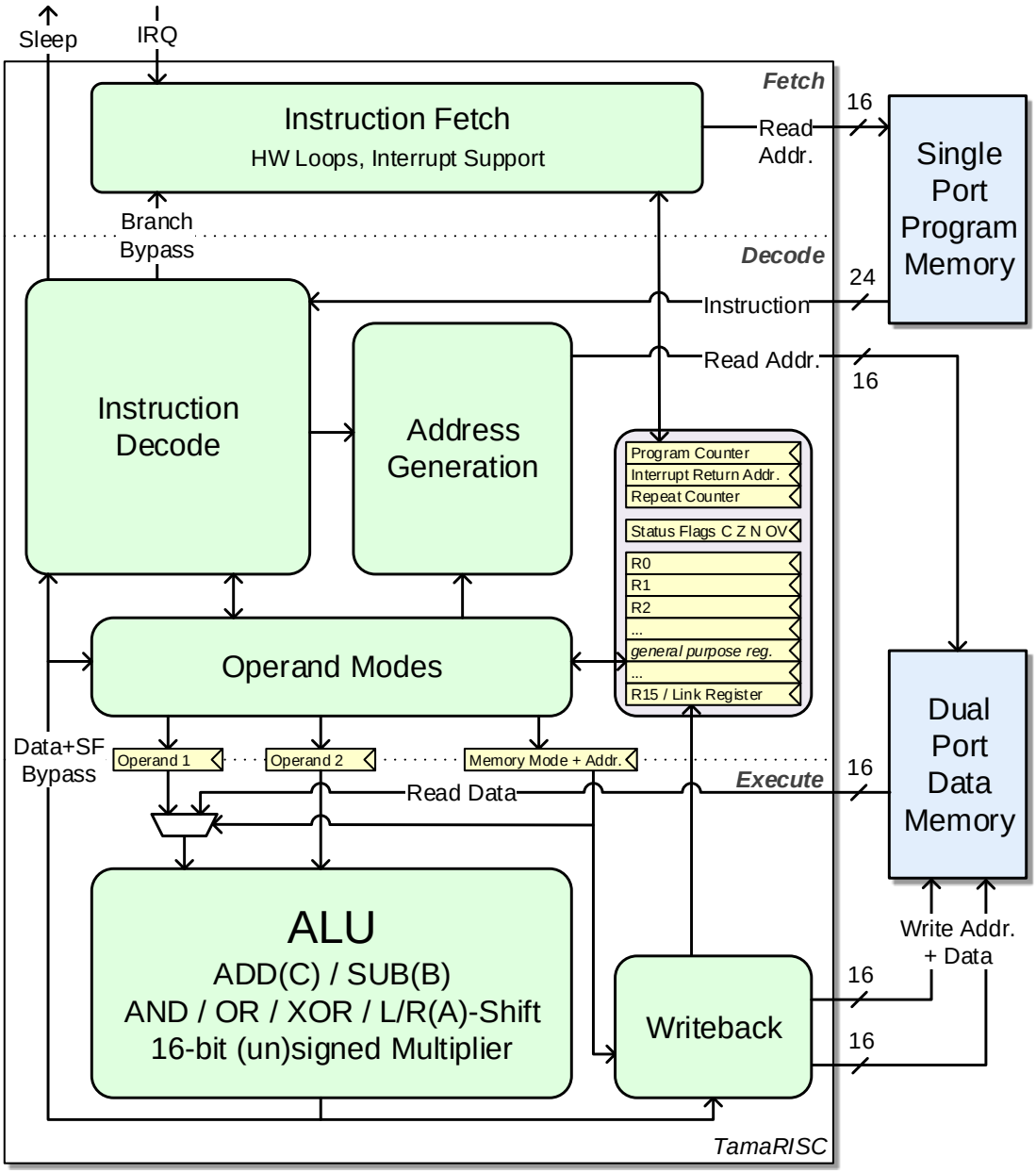


Figure 3.1 – TamaRISC microarchitecture with a 3-stage pipeline, comprising a fetch, decode, and execute stage. The instruction memory and data memory are tightly coupled to the core.

data read and one for data write, all with single-cycle access latency. The tightly coupled data memory (TCDM) hence has to support simultaneous read and write accesses within the same cycle, and can for example be implemented as a dual-port SRAM. The memory ports have single-word width, meaning 24-bit for the program memory, and 16-bit for the data memory. The physical address widths depend on the amount of memory that a specific system implementation integrates, with both the instruction and data address spaces each limited to the range of 16-bit addresses.

The register file has 3 read ports and 4 write ports and provides 32-bit double word writeback support, employed for the multiplication instructions. Three parallel read ports are used to support the reading of two operands and one destination memory address, when using register indirect addressing for the destination. Two write ports are dedicated to the advanced addressing modes with increment/decrement, which modify the register holding a memory address, while the remaining two provide the mentioned double word writeback support.

All instructions generally execute in one cycle, which is guaranteed by the use of complete data bypassing inside the core for register as well as memory writeback data. Additionally, status flag results of ALU operations are forwarded to the decode stage, to allow immediate branching in all cases. This is especially relevant for common instruction pairs where a conditional branch is directly preceded by an arithmetic instruction.

The microarchitecture implements basic hardware loops through a dedicated repeat counter, which can be loaded by the `REP` instruction and is decremented by one in each cycle. The fetch stage effectively freezes the program counter as long as the repeat counter has not reached zero.

The sleep mode is implemented by a core output signal, which is asserted by the `SLP` instruction, (de)activating a core-external clock gate, which controls the clock for the entire TamaRISC core. Wakeup from sleep mode is performed over the interrupt request (IRQ) core input. An incoming IRQ in general initiates execution at a hardcoded IRQ handler address, with the fetch stage saving the next program counter to the dedicated interrupt return address register, which allows to resume the normal program flow at the end of the interrupt handler.

3.1.3 Implementation & Evaluation Flow

This section describes the implementation flow and the evaluation method used for assessment of the power and performance numbers of the presented TamaRISC core. Figure 3.2 illustrates the design flow, highlighting the key electronic design automation (EDA) tools that are employed.

At the center of the flow stands the LISA model of the TamaRISC core, which includes the processor description written in LISA 2.0 [Syn12], simulator enhancements written in C++ to facilitate peripheral simulation in the ISS, and a set of ACE CoSy compiler generator [ACE16] rules. The model is processed by Synopsys Processor Designer (PD) [Syn12] to generate the

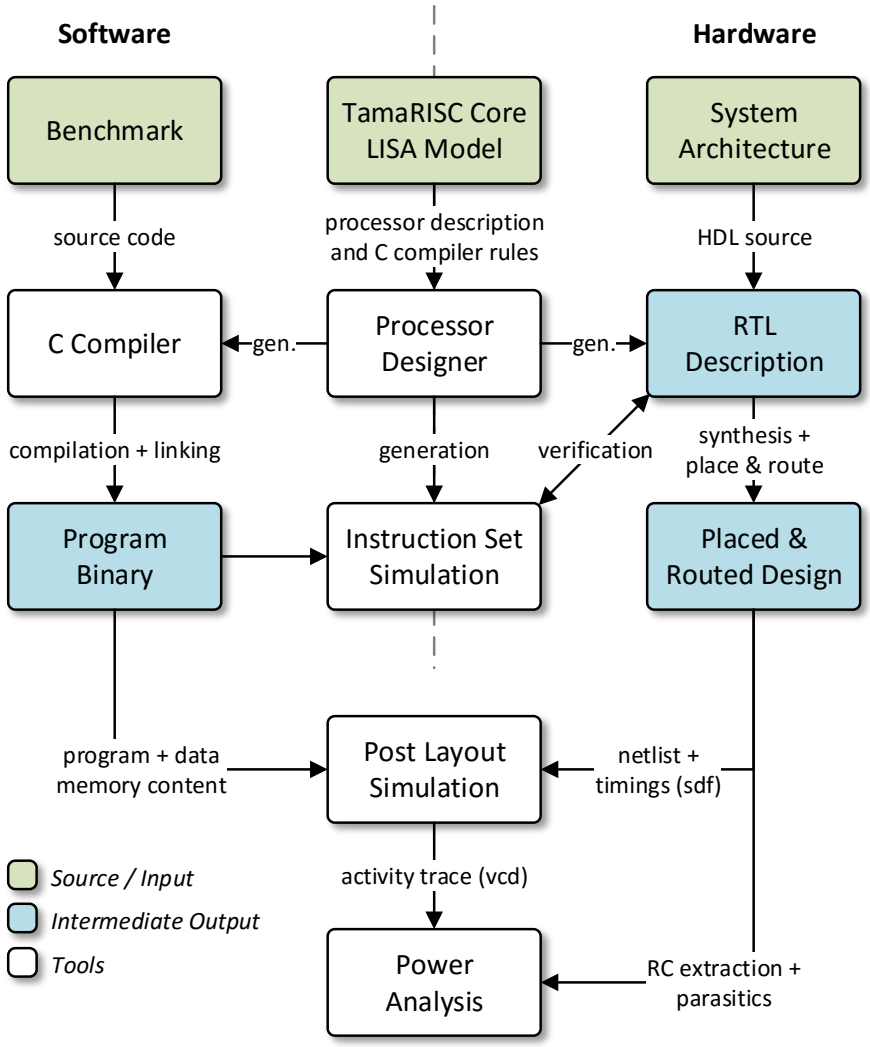


Figure 3.2 – Implementation and evaluation flow for the TamaRISC architecture.

RTL description of the TamaRISC hardware core, the C compiler, and the cycle-accurate ISS, which includes behavioral models for memory-mapped core-external peripherals, namely in our case an ADC, an interrupt controller, and a timer. On the software side, benchmarks written in C, such as sensing applications, are compiled and linked to produce program binaries in the common ELF file format. The ELF files can be directly loaded by the ISS and executed, to verify functional correctness of the application on the architecture, as well as to assess first performance numbers in terms of cycle counts. For a more detailed evaluation using RTL or post layout gate level simulation the content of an ELF file is transformed into two files (separately, for the program and data memory) with a simple ASCII format, which can be read by the employed HDL testbench. On the hardware side the full system architecture, which includes the memory system (e.g., a data memory constructed of multiple banks of SRAM macros), instantiates the TamaRISC core, and is combined in form of VHDL sources

with the VHDL files generated by PD. This RTL description is then verified via RTL simulation against the behavior of the ISS to assure no functional mismatches were introduced during the HDL generation step, transforming LISA to VHDL. The design is then synthesized using Synopsis Design Compiler, and thereafter placed and routed (including clock tree generation) using Cadence Encounter. Static timing analysis is performed on the final post layout netlist, to determine the delay of the critical path, and therefore the maximum achievable clock frequency. The final netlist is then exported together with accurate timings, including gate and wire delays, to perform a post layout simulation of the processor. The simulation incorporates the memory contents for the considered benchmark, and produces a value change dump (VCD) which captures the exact activities of each node in the netlist. Moreover, the simulation output in form of final data memory contents is again verified against the golden result from the ISS, to assure functional correctness. Finally, the VCD trace is used to perform an accurate power analysis of the core executing the given benchmark, based on the post layout netlist and its derived parasitics.

3.1.4 Software Toolchain and Real-Time OS Support

The software tool chain for the TamaRISC processor consists of a standard C compiler, a TamaRISC ISA assembler, and a linker that produces ELF files compatible with the memory address space of the Harvard architecture.

The assembler is directly generated from the LISA processor description, which incorporates the assembly syntax of the defined instructions and operand modes within the model itself. Moreover, the assembler supports basic macros and other directives, which allow the development of hand-written TamaRISC assembly programs with relative ease.

Unfortunately, the C compiler cannot automatically be generated from a LISA model description of the processor architecture, since the full semantics of the defined instructions cannot be derived automatically by the PD tool. It is hence necessary to formulate a set of rules, which instruct the compiler generator (which is part of PD) how to build the C compiler for the target architecture. The main purpose of these rules is to describe transformations from parts of the internal graph representations of the program to sequences of assembly code. The compiler derives an internal representation of the C program that is currently compiled, which has the form of a graph consisting of nodes representing operations (e.g., arithmetic), and edges representing resources (e.g., registers). The task is then to match sub-graphs of such representations onto as optimal sequences of native TamaRISC instructions as possible. A relatively large set of these rules has to be specified for an ISA to be able to generate a basic functional C compiler. Besides all arithmetic operations and different possibilities for data memory accesses this also includes handling of function calls (calling conventions/ABI), handling of the stack, and especially of variable spills (when dynamic register allocation fails). Moreover, rules to provide support for longer data types than the native 16-bit words are essential to support a wider range of C applications and/or libraries, which require 32-bit or even 64-bit data types. The

TamaRISC C compiler, which has been developed as part of this design flow, is a basic compiler which is built upon the CoSy compiler framework embedded in PD, already providing standard optimization techniques/options available in state-of-the-art compilers which can be applied independent of the detailed ISA specifics. However, for example the specific TamaRISC ISA feature that allows pre-/post-modifiers in register indirect addressing modes are so far not included in the current version of the compiler, which leaves room for further improvements in the efficiency of the emitted code for the TamaRISC ISA. Consequently, in cases where every cycle counts, e.g., for small kernel loops that perform the majority of the computations in an algorithm, it is advised so far to rely on hand-written assembly implementations, which can be combined with the full C application via the in-line assembly capabilities of the compiler.

Many embedded systems rely on light-weight real-time operating systems (RTOS) for the efficient management of their tasks that are executed on the system. One of the main tasks of an RTOS is the scheduling of the different tasks on the system, which can be based on different types of scheduling algorithms. A common approach is preemptive scheduling, with fixed time slices, based on a hardware timer, which provides the OS with so-called tick events. Typically, a periodical interrupt gives control to the RTOS, which then preempts the currently running task and resumes the next task on its list (according to some criterion, such as priority, or other). For a processor core to support such a scheduling mechanism, it needs to support both interrupts and the ability to freely configure the interrupt return address and save the full processor state on OS level, to allow preemption of tasks. This functionality, which is necessary for task/context switching, is provided by the TamaRISC ISA via the `MOVS` instruction (see Table 3.1). Furthermore, a core-external tick timer is available in the TamaRISC system, both in the form of a simulation model for use with the ISS, as well as in form of HDL for system integration. These core features are used to support FreeRTOS [Fre16], a popular real-time operating system kernel for embedded devices. FreeRTOS[4] has been ported to the TamaRISC architecture, including all device-specific assembly sources, and is compiled from the original C sources, using the presented C compiler. It has been successfully employed for system simulations running multiple tasks scheduled via the RTOS, on the ISS enhanced by the necessary peripherals.

[4] The specific version of FreeRTOS is v7.0.0.

3.2 TamaRISC-CS: A Sub-Threshold ULP Processor for Compressed Sensing

Digital signal processing traditionally relies on the Nyquist sampling theorem, which states that a faithful reconstruction of a signal, limited to a bandwidth B in the frequency spectrum, can be ensured with a sampling rate of $f_s \geq 2 * B$. Unfortunately, when sampled data needs to be stored or needs to be transmitted over a wireless link, the storage or transmission costs of the raw samples can often limit the energy-autonomous lifetime of the system. In this case, it is advisable to first compress the data. However, in this case the power consumption of the compression process must also be kept low to ensure an overall energy-efficiency advantage.

Compressed sensing (CS) [Don06] is a universal, low-complexity data compression technique to compress sparse signals. CS has been widely used in environmental monitoring systems and in WBSNs [MKAV11], where portable and autonomous devices are expected to operate for long periods of time with limited energy resources. Hence, an ultra-low-power (ULP) CS implementation is crucial for these systems.

On the architectural level, supply voltage scaling, potentially all the way to the sub-threshold (sub-V_T) regime, can reduce both dynamic and leakage power consumption. Therefore, many sensing platforms exploit sub-V_T computing. The state-of-the-art processors for sensing platforms have been reported to consume as little as a few pJ/cycle while operating in the sub-V_T regime [JBW+09, HSL+09, KRV+08]. Sub-V_T computing can also be used to perform CS data compression (in the digital domain). However, most established CS implementations either require a large memory footprint (memory-based implementation) or still require considerable computational effort (despite the inherent complexity advantage of CS). Leakage power consumption becomes a very important challenge in the sub-V_T regime with reduced active power. A considerable amount of leakage in sensing platforms is due to the integrated memories [DCA+12, Dog13]. Moreover, many sensing platforms cannot be power gated completely, to retain their memory content [HSL+09] and hence leakage power is always dissipated. Therefore, implementations with large memory requirements are not desirable in the sub-V_T regime. On the other hand, high computational effort requirements can limit the degree of voltage scaling because of performance degradation issues in the sub-V_T regime [HZS+08, ZNO+06, DWB+10]. These issues ultimately limit the benefits of CS based data compression in ULP sensor nodes, so far.

Application-specific instruction set processors (ASIPs) can compensate for the performance degradation issue, since they are optimized for a specific application domain, providing increased efficiency and performance for the core algorithms of the domain's target applications. For instance, an ASIP optimized for stereo image processing can achieve up to 130x speedup compared to a conventional processor [BDCB11]. These performance optimizations also lead to energy savings as in [KC11], where a processing core with few accelerators dedicated to biomedical applications, can achieve up to 11.5x energy savings compared to the processing core-only implementation. However, despite their efficiency in some specialized application

domains, to our knowledge no ASIP core has been reported so far for ULP CS compression.

We therefore propose to synergistically exploit sub-V_T computing in conjunction with an ASIP core for CS compression (based on the ULP TamaRISC core presented in Section 3.1) to provide an ULP solution for compression of sparse signals for sensing applications. To this end, we extend the instruction set of the low-power TamaRISC processor to better support the specific operations of the CS compression algorithm. Our ASIP core does not require high clock frequencies to run the CS application, and therefore enables more aggressive sub-V_T voltage scaling for a given throughput requirement. The very low memory requirements additionally allow for a major reduction in leakage power, which becomes critical to achieve overall system-wide energy-efficiency in the sub-V_T regime. However, it is important that the introduced ISEs of the ASIP do not significantly impact the voltage-scalability of the core, such that gains from reduced cycle counts are not offset by a potentially smaller range for voltage scaling. Another reason for keeping the introduced ISEs light-weight is the large proportion of leakage power in the overall system power consumption at sub-V_T voltages. The goal is to introduce ISEs which are not area-intensive, i.e., which do not counter-balance the leakage savings achieved through reduction of the system memory requirements.

In the following sections, a custom designed application-specific processor solution to this problem is introduced in detail. We refer to this design by the name TamaRISC-CS in the following. All required parts of the computing system are described (which includes the processing core with the ISE, as well as the embedded memories) that are necessary to enable ULP CS compression on a software-programmable architecture in the sub 100 nW regime. For a typical case study of electrocardiogram (ECG) signal compression in WBSNs (see Section 3.2.4.3), the processor consumes only 30.6 nW for an ECG sampling rate of 125 Hz. Moreover, we show that the proposed processing system achieves 62x speedup and 11.6x lower power consumption with respect to the established computation-based CS implementation running on the baseline low-power TamaRISC processor.

To this end, Section 3.2.1 first explains in detail the CS-based data compression and its features, together with a discussion of the computational bottleneck of the algorithm (generation of a pseudo-random sequence of indices) on the considered architecture. Next, Section 3.2.2 describes the CS-specific ISE for the TamaRISC architecture, and introduces the corresponding adjustments to the microarchitecture. Section 3.2.3 provides an overview of the employed custom embedded memories, enabling sub-V_T operation of the system. Finally, Section 3.2.4 describes the sub-V_T synthesis flow and presents the power and performance results of our TamaRISC-CS ASIP processor for a case study of CS-based ECG signal compression.

3.2.1 Compressed Sensing

Signal compression based on compressed sensing (CS) [Don06] is performed by computing the matrix-vector multiplication

$$y = \Phi x, \tag{3.1}$$

where the random sensing matrix $\Phi \in \mathbb{R}^{k \times n}$ with $k < n$ maps an input data vector $x \in \mathbb{R}^n$ holding n samples to a compressed data vector $y \in \mathbb{R}^k$ with k entries, for a compression ratio of $\frac{k}{n}$.

A general key issue in transform based data compression is how to choose a proper sensing matrix Φ with k rows and n columns. A key advantage of CS compared to conventional transform based compression schemes is that sensing matrices with near optimal properties can be constructed with high probability by choosing the entries of Φ by independent and identically distributed (i.i.d.) random sampling from a uniform distribution [Don06].

3.2.1.1 Reduced Complexity Compression Algorithm

Besides influencing the quality of the compression, the structure and values of the entries of Φ determine the computational complexity of the matrix-vector multiplication. Mamaghanian et al. [MKAV11] showed (for WBSNs) that in fact choosing Φ as a sparse matrix that contains only a few non-zero entries per column at random positions still yields good compression results with a high probability, while it significantly reduces complexity. The non-zero elements can furthermore be chosen as 1, and the number of ones per column (namely I) can be fixed. These constraints on Φ lead to a very efficient algorithm (Algorithm 1) for performing CS data compression. As a result, the computational complexity of the CS algorithm is reduced from $n \times k$ multiplications and $(n-1) \times k$ additions for a dense sensing matrix of arbitrary values, to only $I \times n$ additions. The sensing matrix can therefore be represented in a compact form by a sequence of $I \times n$ random indices $\in \{0, 1, 2, ..., k-1\}$ which describe for each column in Φ the rows with non-zero entries.[5]

Algorithm 1 Pseudocode of Compressed Sensing Algorithm with Reduced Complexity

```
1: buffer[0..k−1] := 0
2: for i := 1 → n do
3:     sample := getSample()
4:     for j := 1 → I do
5:         index := getRandomIndex(0..k−1)
6:         buffer[index] := buffer[index] + sample
7:     end for
8: end for
```

[5] Note that strictly speaking such a representation requires unique row-indices per column. However, this requirement can often be relaxed without a significant impact.

On a resource constrained system, the key challenge of Algorithm 1 is the generation of the (pseudo)random indices. The optimized reference implementation [MKAV11] uses a single arbitrary sensing matrix realized as a fixed sequence of indices stored in memory (for a specific value of k, i.e., a fixed compression ratio). Since multiple such matrices are desirable in practical systems while large memory footprints are undesirable, especially in the context of ULP sensor nodes and sub-V_T operation, we first discuss the generation of the required random indices at runtime.

3.2.1.2 Pseudorandom Number Generation

A pseudorandom number generator (PRNG) can be employed for the generation of the random indices. A common implementation of such a PRNG is a linear feedback shift register (LFSR). The random sequence generated by an LSFR is defined by the sequence of its internal states. The initial state of an LFSR is referred to as its seed. For each state transition (LFSR step) the current internal state bits are combined with the binary coefficients of a polynomial, which defines the pseudorandom sequence of the LFSR. The bits selected by the polynomial are summed to produce one new bit (parity bit). The next state of an LFSR is calculated by shifting out the least significant bit of the state and shifting the generated bit in as the new most significant bit.

Maximum-length LFSRs provide a cycle length of the generated random number sequence that is equal to the number of maximum possible states (excluding zero). Note that although maximum-length LFSRs can provide good sequences of random numbers, the correlation between two subsequent LFSR states, i.e., subsequent indices, i_1 and i_2 is high, since either $i_2 = \lfloor i_1/2 \rfloor$ or $i_2 = \lfloor i_1/2 \rfloor + k/2$. When the state is used directly in the presented Algorithm 1 for CS, this correlation of the generated indices has a negative effect on the compression/reconstruction quality of the data. Hence, we propose to use an LFSR that advances multiple steps per generated index. The number of steps should be equal to the number of required index bits. For example, for $k = 256$ the LFSR has to advance 8 states to generate the next index, which yields no direct correlation to its predecessor since the two subsequent index values share no bits in their representation. The quality of our generated random indices for CS is assessed in the case study presented in Section 3.2.4.3. The drawback of this approach is the increased computational effort for the PRNG, which can be compensated for by custom hardware support.

The proposed generation of the sensing matrix Φ can hence be described with four main parameters: the LFSR polynomial, the LFSR seed, the number of index bits (given by $\log_2(k)$), and the number of non-zero elements per column (I). These four configuration parameters enable the generation of a large set of different sensing matrices. At the same time, the compact representation of the PRNG configuration keeps the memory overhead very limited compared to the case where all indices for multiple matrices would need to be stored.

Consequently, by choosing from a preconstructed pool of favorable values for the PRNG

configuration, it is possible to achieve good sensing performance for a variety of different signal conditions, potentially even by dynamically changing the PRNG configuration at runtime. This capability supports the previously mentioned strength of the compressed sensing method, which lies in the fact that even a randomly chosen sensing matrix Φ ensures a good mapping for the sample data of a signal source, which has sparsity in a specific (potentially) unknown base with high probability. On the contrary, any CS implementation using only a single or a very small number of pre-stored arbitrary sensing matrices loses its generality to perform well independent of the signal source. To alleviate this issue, our approach therefore tries to minimize all related storage and memory costs to support a large number of different random sensing matrices, which can be dynamically changed at runtime.

3.2.1.3 Index Sequence Implementations

This subsection discusses the two common implementation options for the index sequence used as the buffer indices in Algorithm 1 to implement (3.1). The first approach employs precomputation and storage of all required indices in form of a large array in data memory, while the second approach performs the computation of the index sequence at runtime based on a PRNG, as introduced in Section 3.2.1.2.

Precomputation The storage of a preconstructed sequence effectively trades computational effort for memory consumption. For example, the requirement for a single sensing matrix (with 12 non-zero entries per column), used for the compression of a set of 512 samples by 50%, is 6 kbyte of memory. However, as has been discussed a relatively large memory footprint is especially undesirable in an ULP embedded system, for reasons of die area and power consumption. Since sub-V_T memories are large and consume most of the total system power through leakage for low voltages, the storage of tens of kbyte of data for sensing matrices further aggravates the problem, especially when different matrices are to be supported.

Computation at Runtime The generation of suitable random indices can also be performed by using a PRNG, as proposed in Section 3.2.1.2. This approach only requires the data memory to comprise the sensing buffer, which for compression of a set of 512 samples by 50% equals 256 data words (e.g., 512 byte with a sample precision of 12 (up to 16) bit). As shown in Section 3.2.1.2, for each generated index the PRNG has to perform the same number of LFSR steps as the number of bits per index. A typical implementation (on a RISC-like ISA) in software can perform one 16-bit LFSR step in about 10 operations/instructions. For the example of a sensing buffer size of 256 and 12 ones per column in the sensing matrix, this results in 12×8 steps per sample, i.e., a minimum computational requirement of 960 operations per sample, dedicated to the task of random number generation alone. This requirement becomes problematic, since downscaling of the supply voltage considerably limits the maximum core clock frequency (see Figure 3.7). Due to the relatively large computational overhead, achievable sampling rates for sub-V_T operation — i.e., how many samples can be processed per

second by the CS algorithm — are consequently reduced for a purely software-based PRNG approach to the range of tens of Hz, which is often insufficient.

To combine the benefits of instant random number access of the storage approach, with the memory savings of the computational approach, we propose in the following an instruction set extension for TamaRISC, which performs the task of pseudorandom index generation efficiently in hardware.

3.2.2 Instruction Set Extension for CS

Analysis of the CS kernel loop shows that the extension of memory operand addressing with efficient randomization can result in significant performance gains. We hence introduce an extension to the TamaRISC instruction set architecture, adding a new instruction that performs an accumulation of sample data on randomized memory addresses within a defined buffer. Essentially, lines 3-6 of Algorithm 1 are combined into a single instruction, named Compressed Sensing Accumulation (CSA). The assembly semantic of CSA is:

```
CSA *Rb, Rs
```

As shown in Figure 3.3, the CSA instruction takes two general purpose registers as arguments: the first (`Rb`) holds the data memory base address b of the sensing buffer, the second (`Rs`) contains the sample data s. The CSA instruction addresses a random element i within the referenced buffer and adds the provided sample onto the existing value in the memory. This operation is repeated for a configured number of iterations, by the use of a counter register dedicated to the instruction. With each repetition a new pseudorandom element of the buffer is addressed.

The CSA instruction is generally used in a small loop in conjunction with the sleep mode of the processor, which puts the core in a dormant state (clock-gated) to significantly reduce power consumption until new sample data is available. On wakeup by an interrupt request, the sample data is fetched from the ADC and the CSA instruction is executed, after which the core can immediately return to sleep again.

Configurability To enable the construction of many different sensing matrices, the custom instruction is based on four parameters, accessible through dedicated configuration registers. The custom instruction supports software reconfigurability regarding the employed 16-bit LFSR polynomial, the LFSR seed, and the required index width used for memory addressing. Additionally, the number of non-zero entries per matrix column can be configured, which equals the number of times a sample is added to different pseudorandom locations of the sensing buffer. This configurability amounts to storage requirements of at most three 16-bit values per sensing matrix. Since the LFSR state of the *address randomizer* can be directly accessed through the register file, the LFSR hardware can also be used for efficient pseudorandom number generation, independently of the CS-specific memory addressing and accumulation.

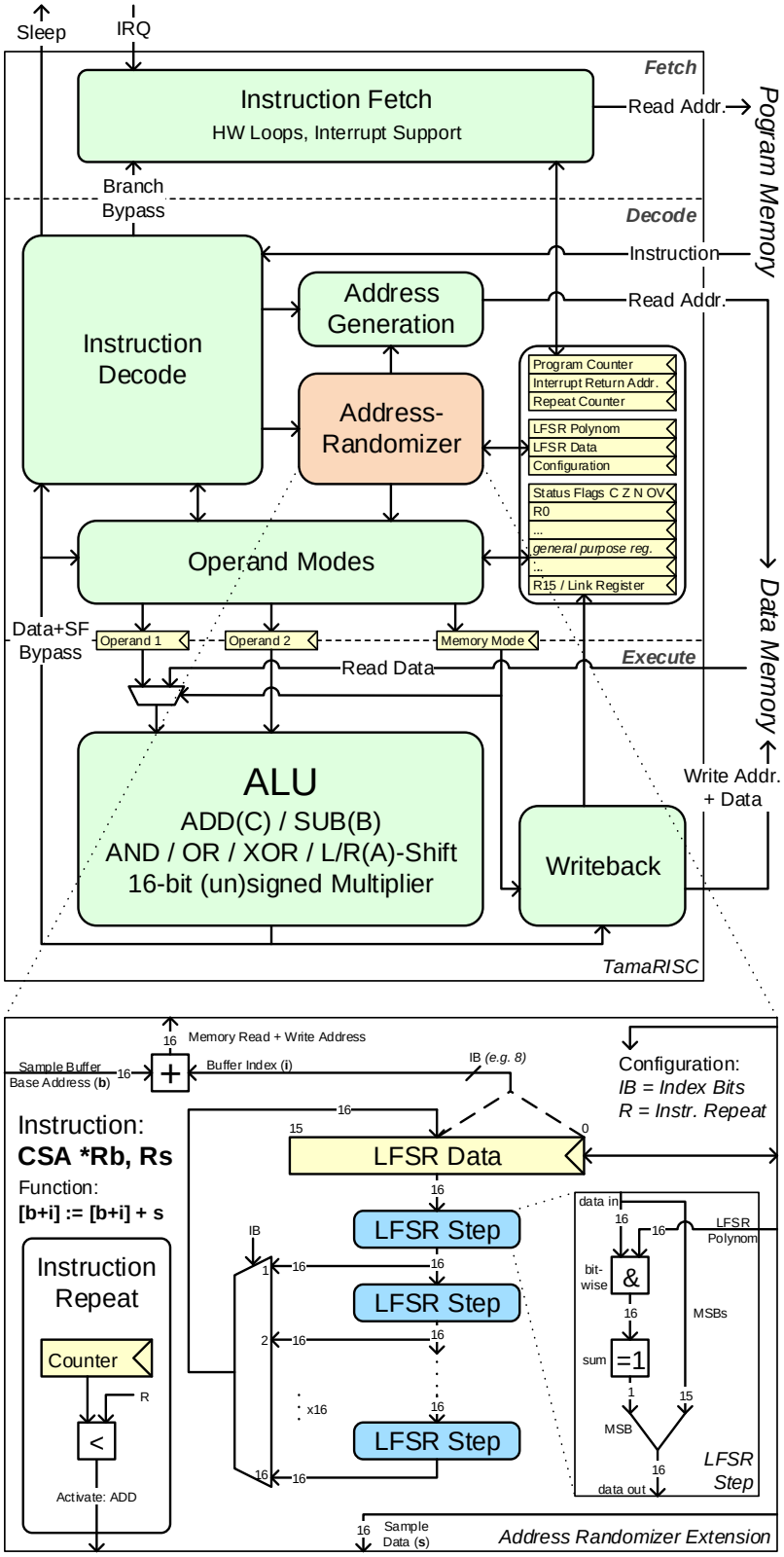


Figure 3.3 – TamaRISC-CS sub-V_T microprocessor architecture including address-randomizer extension for compressed sensing.

Hardware Implementation The internal hardware structure of the *address randomizer* extension to the TamaRISC microarchitecture is presented in Figure 3.3. The custom instruction employs for the sample accumulation the existing 16-bit adder unit in the ALU and does not introduce any new units to the data path of the execution stage of the processor. The decode stage holds the extended address generation logic, which enables addressing of a random word inside the sensing buffer by combining a buffer base address b with index bits (i) taken from the least significant bits of the current LFSR state. The number of index bits depends on the value set in the configuration register (1-16). In one cycle, the LFSR state is updated by the same number of LFSR steps as index bits used. Additionally, the instruction set extension is realized as a multi-cycle instruction, which allows handling of one sample in a number of cycles equal to the configured number of non-zero entries I per matrix column for the CS application.

3.2.3 Sub-V_T Memories

While the core logic of the sub-V_T CS processor works reliably at low supply voltages in the sub-V_T regime, conventional data and instruction memories based on 6-transistor (6T) SRAM bitcells fail to operate reliably at low voltages [QSC11]. Therefore, such conventional, embedded 6T SRAM macrocells prohibitively limit the overall reliability and the manufacturing yield of the proposed sub-V_T CS processor. More precisely, under gradual supply voltage down-scaling, read and write access failures start to appear first, before the occurrence of data retention failures at even lower voltages [MMR05]. Specially designed SRAM macrocells based on 8- or 10-transistor (8-10T) bitcells are typically used to enable reliable data storage in the sub-V_T regime. For example, a typical 8T SRAM cell contains a read buffer to avoid the direct access of the bit lines to the internal storage nodes and consequently to avoid the risk of switching the bitcell during a read access [CFH+05], thereby improving read-ability. Moreover, a popular 10T SRAM bitcell contains, in addition to the read buffer, a tri-state inverter in the cross-coupled latch; this tri-state inverter is disabled during a write access in order to avoid write contention [JKY+12], thereby improving write-ability. All these 8T or 10T SRAM macrocells, specifically optimized for robust sub-V_T operation, need to be custom-designed due to the lack of good, commercially available low-voltage memory compilers. Such custom design is associated with a high engineering effort and bares high risk, unless each macrocell is first manufactured and silicon-proven independently, before its integration into a larger VLSI system.

As opposed to such custom-designed sub-V_T 8T/10T SRAM macrocells, we employ a fully automated standard-cell based memory (SCM) compilation flow [MSBR11]. The use of SCMs considerably simplifies the design process, and the resulting latch or flip-flop arrays directly avoid the aforementioned reliability concerns of conventional 6T SRAM. In particular, standard-cell latches already contain a read buffer to avoid read failures and a cell-internal keeper which is disabled during write, i.e., during the transparent phase of the latch, to avoid the risk of write failures. Consequently, the proposed SCMs work reliably in the sub-V_T regime without the

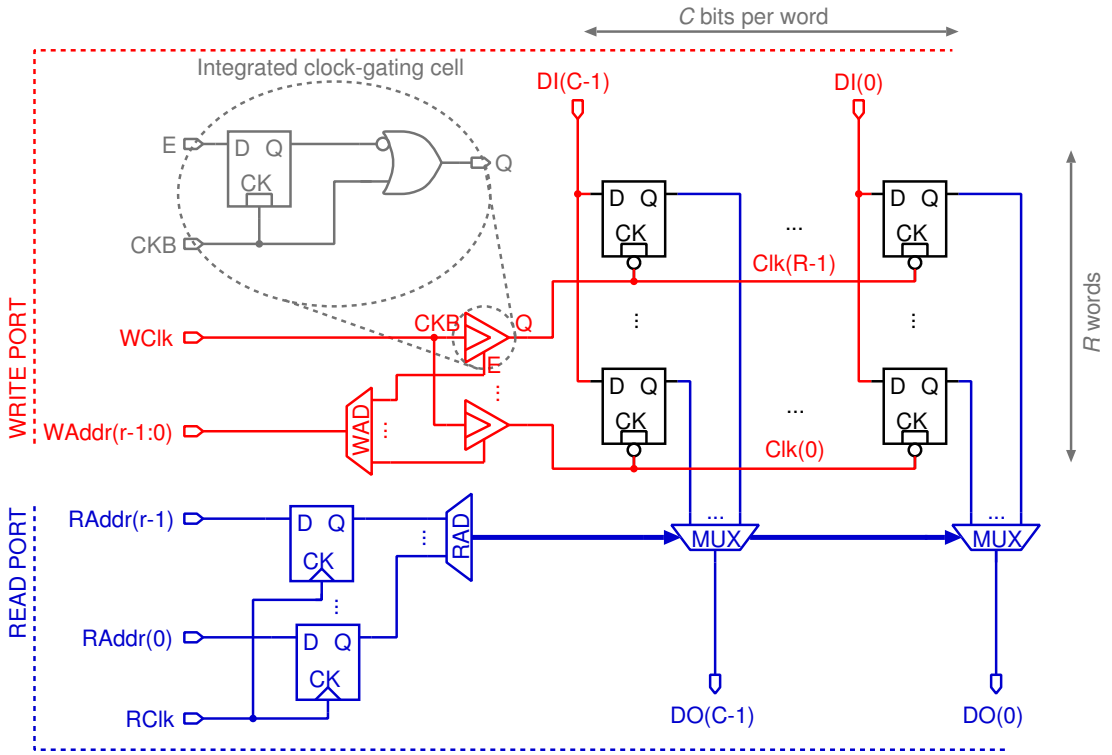


Figure 3.4 – Schematic of the latch array used for sub-V_T operation of TamaRISC-CS, with clock-gates for the generation of write select signals and static CMOS readout multiplexers. The write port is highlighted in red, while the read port is highlighted in blue.

need for any extra engineering effort and allow the complete system to operate at aggressively scaled voltages.

Among many architectural variants, summarized in [MSBR11], this work adopts the latch array architecture shown in Figure 3.4. This architecture consists of a write address decoder (WAD) and clock gates for the generation of the one-hot encoded write select pulses (row-wise gated clock signals). Moreover, static CMOS multiplexers are used to read out the desired address (word). While a read logic based on tri-state buffers exhibits lower leakage current, the chosen CMOS multiplexers are faster and more robust for sub-V_T operation. This latch array architecture can be synthesized from commercially available standard-cell libraries.

In addition to the baseline array employed in our experiments, it is possible to customize one or several standard-cells to meet a specific design goal, such as ultra-low leakage power. For example, the leakage power and access energy can be reduced by approximately 50% by using a single custom-designed standard-cell, namely an ultra-low leakage latch using stack forcing and channel length stretching, as well as a tri-state enabled output buffer to implement the read logic [MAM+12].

Even though these latch-based memories are optimized for low-voltage and low-power op-

eration, they still consume considerable leakage power. In our system example, memories account for 70–95% of the architecture's total power consumption, depending on the mode of operation. Furthermore, the sub-V_T memories consume a considerable area share. In our implementation with moderate memory sizes of 256 instructions (6 kbit) and 512 data words (8 kbit) the processing core only consumes 16% of the total area, with the rest occupied by memory.

3.2.4 Power and Performance Results

Due to the need to retain their memory content, many sensing platforms cannot be power gated completely [HSL+09], and consequently leakage power is always dissipated. Therefore, our sub-V_T CS processor always operates at a clock frequency that barely accomplishes the task (of fixed computational effort) on time, while lowering the supply voltage to the corresponding minimum possible level that avoids timing violations at the lowest required target frequency. Note that the objective is to minimize total power for a given workload in contrast to operation at the energy minimum voltage, where maximizing energy efficiency often requires a higher operating voltage to balance leakage and active energy. As a result, the CS processor operates at a higher energy per clock cycle than the achievable minimum. However, it does so with a duty cycle of 100%, which makes the approach from a system perspective more power and energy efficient than operation at the EMV, with a duty cycle of less than 100%, where higher leakage power is still dissipated during the inactive phases in which the memory cannot be power gated to assure data retention.

Yet, it has to be noted that reductions in total power below the EMV are unfortunately not as substantial per scaled mV of supply voltage as they are above the EMV, since leakage power begins to fully dominate and as a consequence the still strong further reduction in active power has less and less of an impact on the total power. Figure 3.7 also illustrates this fact. This issue can make the choice of the operating point also subject to a valid trade-off between increased variability issues that occur with reduced voltage and potentially non-substantial additional power gains. This means that around and below the EMV it is often possible to trade a slightly higher power consumption and supply voltage, for a circuit that exhibits considerably less performance variation [DWB+10]. In the following, we investigate only the achievable minimum power consumption, since the considered circuit structures are relatively small and simple, and hence are not necessarily exposed to the same issues found due to strong variations in larger designs.

3.2.4.1 Synthesis Strategy and Sub-V_T Energy Profiling

The design is synthesized above threshold at nominal supply voltage of 1.0 V with a low-power high threshold-voltage ($V_T \approx 700$ mV) 65-nm CMOS technology. Toggling information is obtained by simulating a fully placed & routed design (including clock tree) with back-annotated timing information. The design is characterized by employing the sub-V_T energy charac-

Table 3.3 – Circuit properties of TamaRISC-CS for sub-V_T modeling after [ARLO12].

Design property	Value
Capacitance scaling factor (k_{cap})[a]	254000
Critical path delay (k_{crit})[b]	434
Average leakage scaling factor (k_{leak})[c]	194000
Mean switching activity (μ_e)	0.0675

[a] k_{cap} is normalized to the input capacitance of a single inverter.
[b] k_{crit} is normalized to a single inverter delay.
[c] k_{leak} is normalized to the leakage current I_0 of a single inverter.

terization model that has been derived in [ARLO12]. With this model, parameters retrieved from critical path information as well as a traditional value change dump (VCD) based power simulation are used to compute maximum operational speed, energy and power dissipation in the sub-V_T region. The sub-V_T energy profiling model is validated by measurements [ARLO12] and accuracy is within a 10% error rate at the measured temperatures 0° C, 27° C and 37° C. In Table 3.3 the design properties of TamaRISC-CS are shown in terms of leakage, capacitance, and critical path normalized to a single inverter, as well as switching activity. For more details, the reader is referred to [ARLO12].

In our implementation, the post-layout critical path delay at nominal supply voltage is 5.2 ns, according to the gate-level static timing analysis. Optimization for maximum frequency and thus a larger slack on the critical path allows for more aggressive voltage scaling. However, leakage and active power increase considerably with hard timing-constrained designs. Tight constraints will force the tool to infer nets with high fan-out as well as stronger buffers, which increases capacitance on the critical path and consequently yield a slower operation in the sub-V_T region. Following the strategy proposed in [MAS+11, MSBR11], we relax the timing constraint to achieve a design with low area and low leakage cost.

Figure 3.5 shows the design exploration for this implementation strategy, where the timing constraint is relaxed to achieve lower power. The figure details the relative corresponding power consumptions in the sub-V_T regime for the CS processor, optimized with different clock constraints above threshold. The power values are normalized to the design optimized with the maximum clock frequency (5.2 ns), for each operating point (frequency/throughput) separately. For this experiment, each design is always supplied by the minimum voltage level required for the respective throughput. As seen from the figure, the design optimized with a 9 ns clock constraint above threshold achieves better energy efficiency compared to the same design with other target clock frequencies at nominal voltage. To obtain the respective minimum power solution in the sub-V_T domain, we therefore optimize the design with a relaxed clock constraint of 9 ns above threshold voltage.

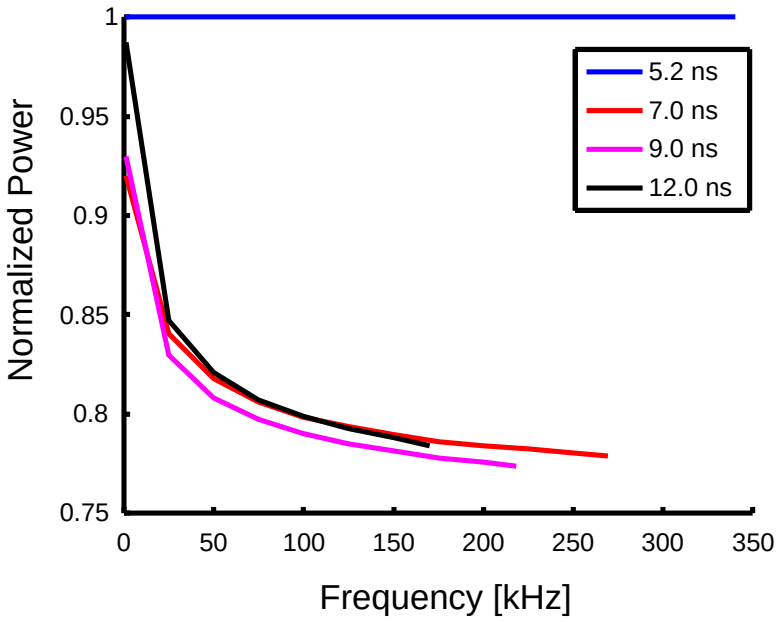


Figure 3.5 – Relative power comparison vs fixed operating frequency in the sub-V_T regime for the TamaRISC-CS processor, when implemented with different clock constraints. The power is normalized for each operating frequency separately to the consumption of the hard-constrained design (5.2 ns) at that specific frequency, operating at the minimum possible voltage.

3.2.4.2 Implementation and Simulation Results

In the following, we present a general performance characterization of the TamaRISC-CS core including our SCMs, for the sub-V_T regime. The characterization data is obtained via simulations of the placed & routed circuit (layout depicted in Figure 3.6), in conjunction with sub-V_T energy profiling, as described and summarized in Section 3.2.4.1.

Figure 3.7 shows the power consumption versus the corresponding supply voltage of the sub-V_T CS processor at the maximum achievable clock frequency in the sub-V_T domain. More specifically, a clock frequency of 100 kHz for the CS processor is achieved at a supply voltage of 0.37 V. As a result, a total power of 288 nW is dissipated, where 27% of the power consumption is due to leakage power. When the required clock frequency is reduced to 1 kHz, the model indicates that the sub-V_T CS processor consumes 22.5 nW in total, where the leakage dissipation now has a share of 98%. As demonstrated in Figure 3.7, the leakage power dominates the overall power for clock frequencies lower than 1.5 kHz, corresponding to 0.2 V supply voltage. However, we note that in this particular technology, operation below 0.25 V is not recommended due to higher rates of functional failures from larger process variations according to [ARLO12].

The energy profile of the sub-V_T TamaRISC-CS processor is shown in Figure 3.8. Energy dissipation per cycle at maximum operational frequency is shown together with fixed clock

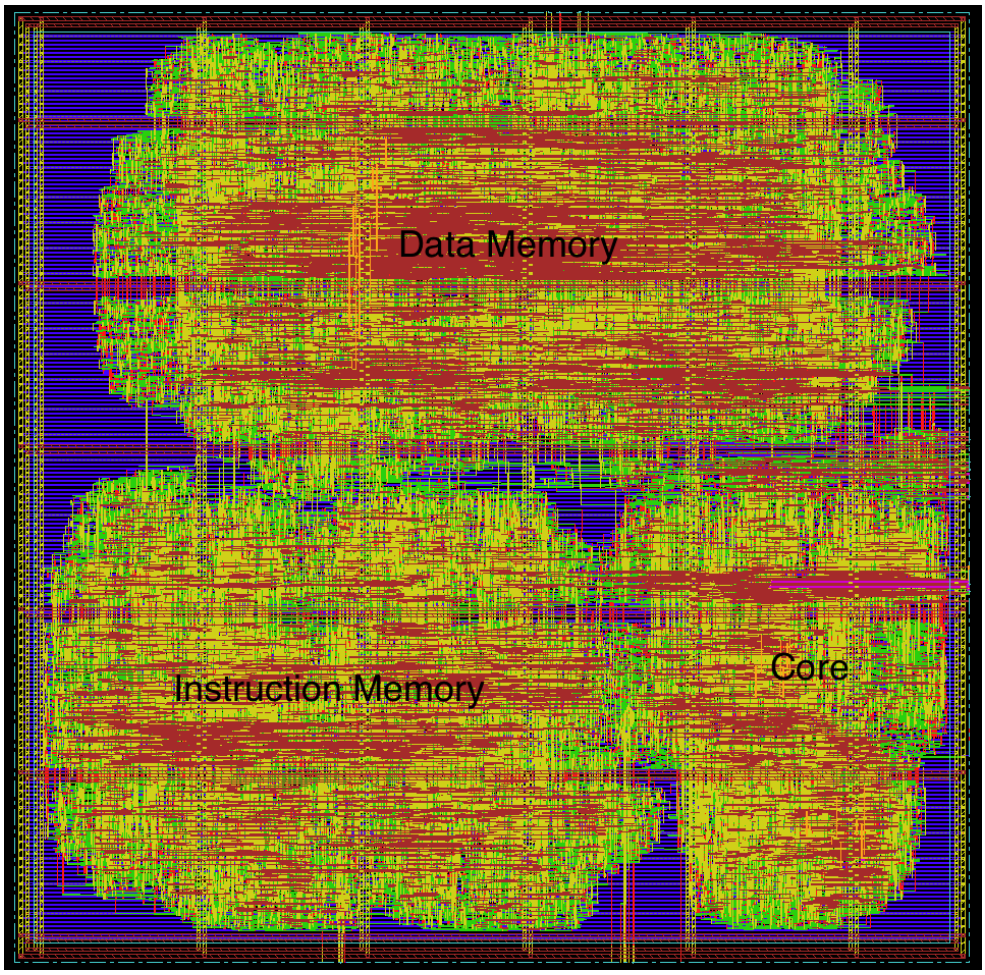


Figure 3.6 – Layout of placed & routed TamaRISC-CS processor in 65 nm, including instruction and data memories (6 kbit + 8 kbit) implemented in the form of ultra-low-voltage SCMs.

frequencies of 2.1 kHz, 16.5 kHz, and 100 kHz. Note that the points of the three fixed frequency traces in the plot which are below the maximum frequency line are only theoretical extrapolations, for better comparison of the energy profiles. In general, it can be observed that operating at a lower frequency than dictated by the supply voltage results in higher energy dissipation for equal work. Thus the implementation goal is to use a supply voltage that is just barely sufficient to support the necessary clock frequency.

The total area of the sub-V_T TamaRISC-CS processor is 84.7 kGE, where 1 GE corresponds to the area of a NAND-2 minimum drive strength gate. The instruction and data memory in the processor have a size of 768 byte and 1024 byte, respectively. The memories occupy 84% of the overall area, whereas the core occupies the rest. The area overhead of the proposed instruction set extension for CS accounts for less than 3% of the overall area. Moreover, the introduced CS ISE does not affect the critical path of the design, and hence does not impact the maximum clock frequency or voltage-scalability of the core, in comparison to the baseline design.

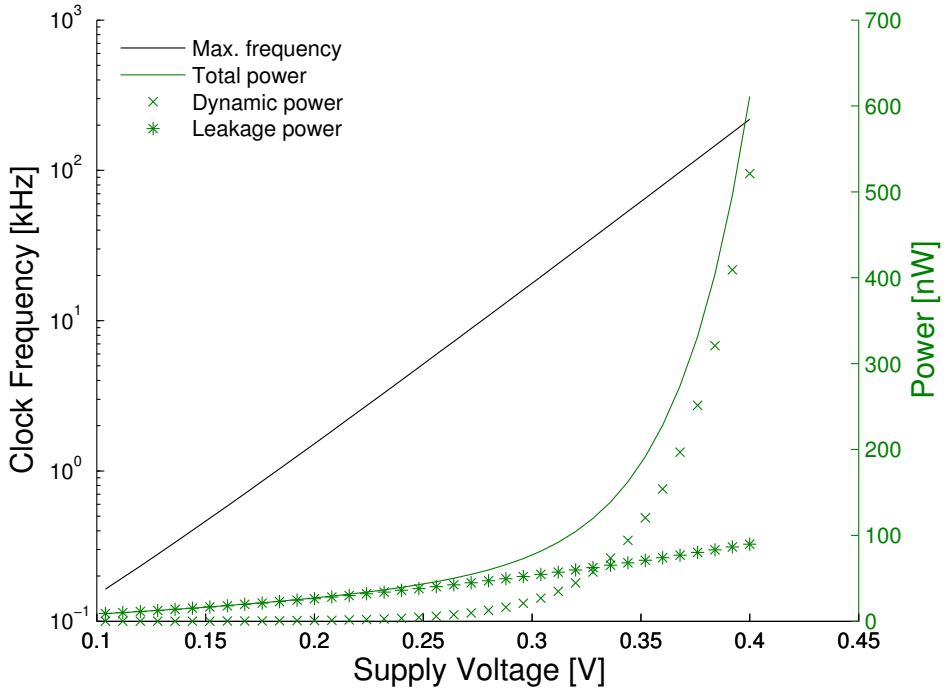


Figure 3.7 – Power and performance exploration of TamaRISC-CS. The power values are for operation at the indicated maximum achievable clock frequencies for a given voltage.

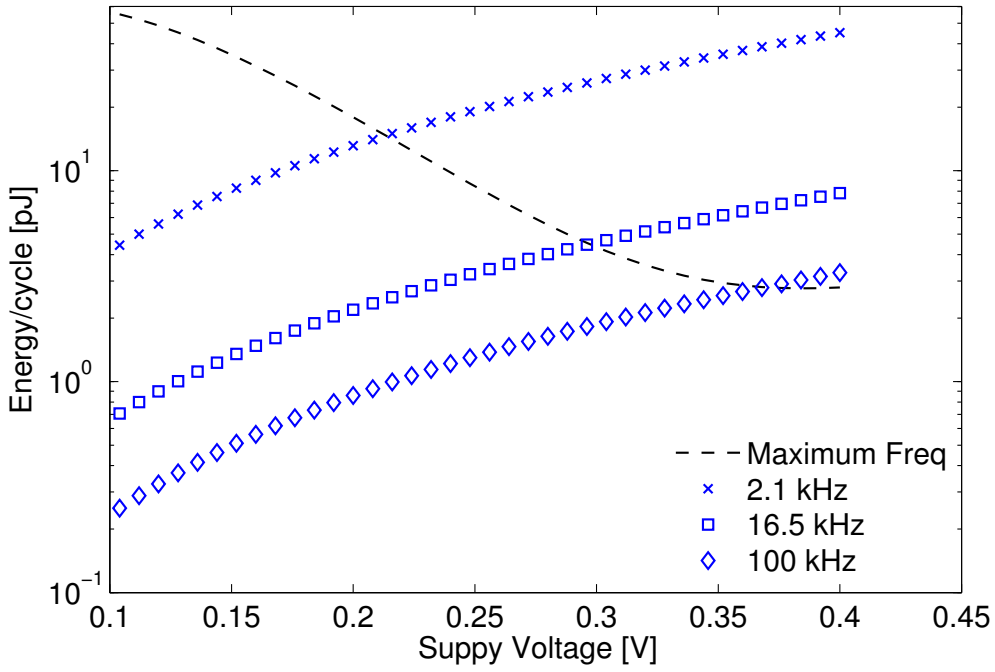


Figure 3.8 – Energy efficiency profile for operation of TamaRISC-CS at the maximum achievable frequency over the sub-V_T range (black) and for fixed frequency operation for three different frequencies (blue).

3.2.4.3 Case Study: CS-Based ECG Signal Compression

As a case study, we apply the CS algorithm for the compression of electrocardiogram (ECG) signals, as proposed in [MKAV11]. ECG monitoring is one of the key applications for healthcare-oriented wearable WSNs, utilizing deeply embedded processing cores. The ECG test case performs data compression on blocks of 512 samples, recorded at different sampling rates.

Quality of Produced Sensing Matrices Mamaghanian et al. [MKAV11] have shown that 12 non-zero elements in each column of the sensing matrix are sufficient to maintain satisfactory quality of reconstructed ECG signals for diagnostic purposes. Based on the study in [MKAV11], we group random indices into groups of 12, where each group determines the non-zero elements of the corresponding column in the sensing matrix. Assuming that there are no repeated indices in a group, the corresponding column of the sensing matrix will have only ones and zeros. However, in case of repetition of indices, the corresponding data values will accumulate, which, according to our experiments, does not lead to any quality degradation in the reconstructed signal as shown in Figure 3.9.

To ensure a good quality of diagnostic analysis on the reconstructed ECG signal, the compression performance is quantified according to the percentage root-mean-square difference (PRD) for different compression ratios [MKAV11]. PRD quantifies the percent error between the original and the reconstructed signal where a PRD value less than 9 is classified as "very good" or "good" quality for ECG diagnosis. Thanks to our configurable CS-extension, many sensing matrices with different combinations of primitive polynomials and seeds can be constructed. These sensing matrices are analyzed by quantifying their corresponding PRD values for various compression ratios. More specifically, Figure 3.9 shows as an example the PRD values with respect to various compression ratios for one of the constructed sensing matrices with a polynomial

$$p = x^{13} + x^{12} + x^{11} + x^{10} + x^9 + x^7 + x^3 + x^2 + x^1 + 1$$

and the seed “0x6218” in hexadecimal notation. As seen from Figure 3.9, a PRD value just below 10 is retained for compression ratios up to almost 60%. Moreover, 50% compression is achieved with a PRD of 7.7. Similar to the state-of-the-art CS sensing matrices [MKAV11], the sensing matrices that are generated by our multi-step LFSR mechanism, accomplish a "good" or "very good" quality of the reconstructed signals for compression ratios less than 53%.

Power vs. Performance Analysis We consider the example of 50% data compression of ECG signals, using the ECG database in [oHSC00] for stimuli generation, to analyze the power and performance of our sub-V_T CS processor. The required operating frequency to support a given sampling rate to compress ECG signals in real-time is given by: $f \geq N * f_s$ where f_s and N stand for the given sampling rate and the required average number of clock cycles to process one sample. The clock frequency of the sub-V_T CS processor is always adjusted, to have the

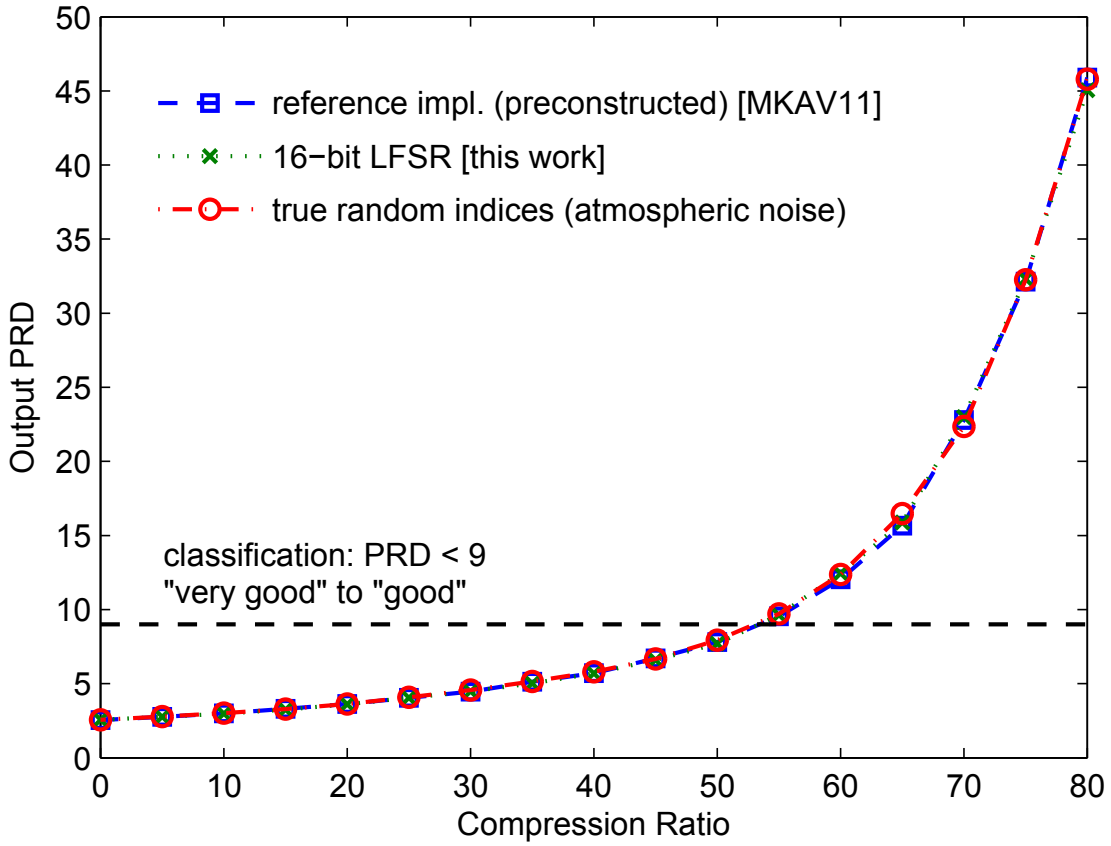


Figure 3.9 – PRD values at various compression ratios for three index sequences (sensing matrices Φ), each using different methods of construction.

minimum required clock frequency, according to the given ECG sampling rate. The supply voltage of the processor is then lowered accordingly.

The presented sub-V_T TamaRISC-CS processor requires 8460 clock cycles to apply 50% compression to 512 samples of ECG data when the sensing matrix is constructed by 12 random indices per column ($I = 12$). This corresponds to an average of $N = 16.5$ cycles processing time for each sample (16 cycles per sample + setup overhead per sample set). As a result, the sub-V_T TamaRISC-CS processor must operate with a clock frequency of 2.1 kHz and 16.5 kHz for 125 Hz and 1 kHz sampling rates, respectively. Figure 3.10 shows the power consumption of the sub-V_T CS processor for various ECG sampling rates. More specifically, for a 125 Hz sampling rate the sub-V_T CS processor consumes only 30.6 nW in total with 95% of the power due to leakage. For a sampling rate of 1 kHz, the total power consumption is only 74 nW , where 70.7% is because of leakage dissipation.

To compare our ISE-enhanced CS processor with the baseline processor, we consider the construction of the CS sensing matrix by computing random sequences of indices based on a PRNG algorithm (see Section 3.2.1.2) running on the baseline ISA of TamaRISC. Our results

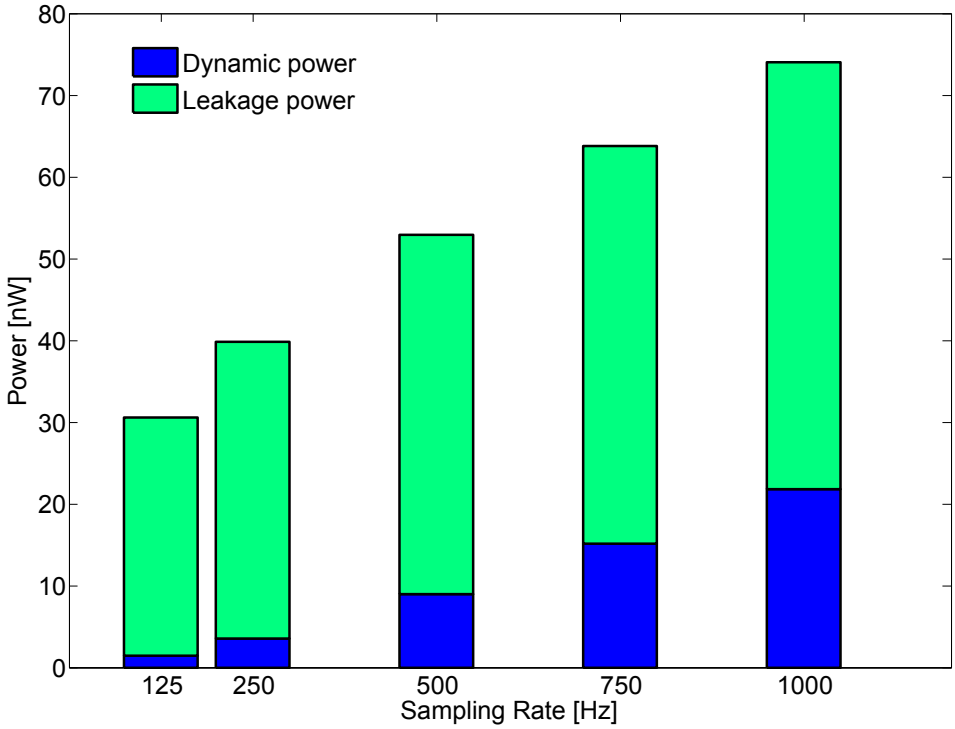


Figure 3.10 – Total power consumption of TamaRISC-CS for CS-based ECG signal compression, with various ECG sampling rates.

show that the optimized implementation for the baseline core requires a significantly higher computational effort. Specifically, the increased total computational effort per sample in terms of cycles amounts to $(10log_2(k)+5)I+5$, compared to our implementation based on the proposed CSA instruction with an effort of $I+4$ cycles. Hence, in the case of LFSR emulation by software, code optimized to the baseline ISA processes one ECG sample, including the sensing matrix construction, on average in $N = 1025.5$ cycles, which translates into a speedup of 62x for our ISE-supported implementation on TamaRISC-CS. Therefore, a sampling rate of $f_s = 125$ Hz requires a clock frequency of 128 kHz, using the pure software approach. In combination with the necessary higher supply voltage to still meet the sampling frequency target, this results in a total power consumption for the design of 355 nW (compare Figure 3.7), which is 11.6x higher than the sub-V_T CS processor with ISE, where the random indices are produced with the help of the embedded LFSR. Looking at scenarios with higher ECG sampling rate requirements than only 125 Hz, we see that our proposed TamaRISC-CS processor is able to stay in the sub-V_T operating regime, while the baseline design is not. Already for a sampling rate of 250 Hz, a clock frequency of $f \geq 256\,kHz$ is required without ISEs to construct the sensing matrix via software on the fly. This high performance cannot be achieved anymore in the sub-V_T regime, yet in the near-threshold regime.

Moreover, Mamaghanian et al. [MKAV11] report a code execution time of 25 ms on a different

16-bit architecture (MSP430-based Shimmer platform) with a clock frequency of 8 MHz, for applying 51% compression on a set of 512 ECG samples, where pre-computed random indices are stored in the memory. This results in $N = 390.5$ cycles per sample, a 23.6x higher performance requirement than our ISE-supported TamaRISC implementation (TamaRISC-CS), in terms of cycle count alone.

The presented power results highlight one of the main challenges of deep sub-threshold design, which has been discussed earlier, the strong impact of leakage power for ultra-low-voltage operation. Even while keeping the processor design relatively small, with very limited memory, leakage power quickly dominates the total power consumption, as soon as the clock cycle requirements for the application can be significantly reduced, and voltage is scaled accordingly. Effectively, the introduced CS ISE pushes the microprocessor in the deep sub-V_T region, which does not allow to take full advantage of the strong reductions in active energy, because of the high leakage that does not reduce with the same rate with scaled voltage, as illustrated in Figure 3.10. This effect can also clearly be seen by the fact that a reduction in processing cycle requirements by 62x does not result in a power reduction of at least 62x, but only 11.6x. If the circuit could be efficiently operated in a regime where the leakage power only makes up a very small part of the total power consumption, the expected power reduction through frequency reduction alone would be close or equal to the performance speedup, not even considering additional power gains due to lowering of the supply using the created voltage headroom.

3.3 Comparison with State of the Art

The aim of this section is to give a brief overview of state-of-the-art embedded low-power processors and MCUs, including both latest research and commercial designs. This overview includes discussion on the different aspects of the implementations that make a direct comparison, e.g., in terms of raw power or energy numbers, challenging.

Table 3.4 lists different processor-based general-purpose[6] low-power ICs for (deeply) embedded applications, together with their main architectural features, as well as their power and performance numbers. The table includes five popular, commercial MCU products from five different vendors targeted at low-power and ultra-low-power applications, in their latest product versions. Moreover, the comparison includes two of the most advanced fabricated research ICs, aimed at pushing the limits of energy efficiency for deeply embedded computing on stand-alone WSNs. The two designs are chosen out of the many designs reported in the literature, since they achieve not only extremely low energy consumption for both active and standby modes, but do so for a full SoC design, integrating basic peripherals, as well as on-chip voltage regulation, which makes them closest in terms of necessary hardware overhead to a device used in real-world applications. Although there are a number of other interesting processor designs reported in the literature which have similar (sometimes lower) active energy consumption [HSL+09, AHK+11, ISP+11, FKHO11, ISU+14, ACB+15], they do not integrate all the necessary circuitry for fully stand-alone operation, e.g., off a battery cell. For comparison, the table also includes two designs based on the contributions of this thesis, specifically the TamaRISC-CS processor presented in Section 3.2, using the introduced TamaRISC architecture, and the DynOR processor presented in Chapter 5, based on the OpenRISC architecture. While the TamaRISC-CS design is targeted at ultra-low-power operation in the sub-threshold regime, the DynOR design probably fits the least within the list of other processors, since it is targeted more towards embedded computing with higher processing requirements (hundreds of million/mega operations per second (MOPS)). DynOR however does support operation at fairly low voltages, which make the measurements provided in Table 3.4 interesting for comparison, as they turn out to be still competitive with other low-power designs.[7] Note that due to the nature of the research goals that the DynOR test-IC addresses, the design does not have any circuit provisions for dedicated sleep or standby modes, which would allow retention of CPU and memory states with low power consumption.

When comparing different MCUs, one of the key aspects that makes direct comparison challenging is the difference in the employed microprocessor architectures. From Table 3.4 we see that the CPU architectures of course differ in the data path width they employ, which can have a major performance impact, strongly depending on the targeted application.

[6] The term *general-purpose* is here used to differentiate between specialized processor-based designs which are fully or mainly application specific, such as DSPs, vector/array processors, etc.

[7] As indicated in Table 3.4, DynOR does not comprise on-chip voltage regulators or other more advanced features of typical MCU SoCs with extensive peripherals.

Table 3.4 – Overview of low-power microprocessors and MCUs, comparing state-of-the-art commercial products, fabricated research-ICs, and contributions of this work.

	P&R Circuit	Fabricated Research-ICs			Commercial Products				
	TamaRISC-CS [this work] *Chapter 3*	**DynOR [this work]** *Chapter 5*	**Cricket [MSG+16]**	**Sleepwalker [BVH+13]**	**STM32L433xx (STMicro.) [STM16]**	**EFM32ZG110 (Silicon Labs) [Sil15]**	**ATSAML21x (Atmel) [Atm16]**	**MSP430FR573x (Texas Instr.) [Tex16]**	**PIC24FJxxx (Microchip) [Mic11]**
CPU	TamaRISC	OpenRISC	ARM Cortex-M0+	MSP430	ARM Cortex-M4	ARM Cortex-M0+	ARM Cortex-M0+	MSP430	PIC24
Data Path	16-bit	32-bit	32-bit	16-bit	32-bit	32-bit	32-bit	16-bit	16-bit
Features	ultra-low-voltage SCMs	dynamic clock adjustment	10T-SRAM for sub-V_T operation	adaptive voltage scaling (on-chip)	FPU, ART Acc.	AES	AES	FRAM	-
Full SoC incl. Peripherals	no	no	yes	yes	yes	yes	yes	yes	yes
On-Chip Regulation	no	no	LDO + SCVR	SCVR	yes	yes	yes	yes	yes
Technology	65 nm LP	28 nm FD-SOI	65 nm LL	65 nm LP/GP	-	-	-	-	-
Max. Clock Freq. [MHz]	192	1306	66	71	80	24	48	24	32
Cycles per Instr. (CPI)	1	≈1	≈1	1-6 (avg. ≈3-4)	≈1	≈1	≈1	1-6 (avg. ≈3-4)	2
Work Mem. (SRAM) [kB]	1 (+ 0.75 IM)	8 (+ 16 IM)	16 (6T / HV) + 8 (10T / ULV)	2 (+ 16 IM)	64	4	32 + 8 (LP)	1	8
Non-Volatile Memory [kB]	-	-	-	-	up to 256 (flash)	up to 32 (flash)	up to 256 (flash)	up to 16 (FRAM)	up to 128 (flash)
Min. Supply Voltage [V]	0.20 - 0.25 (estimated)	0.26 (SRAMs: 0.40)	0.20	0.27	1.71	1.98	1.62 (core: 0.9)	2.00	2.00 (core: 1.2)
Active Power [µW/MHz]	2.8*	13.0	11.7	7	143	226	84.6	163	150
Standby Power [µW]	0.03*	-	0.08	1.7	0.48	0.99	2.7	3.0	0.66
Best Energy [pJ/Op.]	2.8* @ 0.108 MOPS	13.0 @ 24 MOPS	11.7 @ 0.688 MOPS	24.5 @ 6.57 MOPS	143 @ 26 MOPS	226 @ 24 MOPS	84.6 @ 12 MOPS	570 @ 6.86 MOPS	300 @ 16 MOPS

* The power and energy values for TamaRISC-CS are derived via the sub-V_T profiling model of [ARLO12], and comprise the core with its integrated memories.

While more control oriented applications will generally not see much benefit from larger word widths, signal processing applications might take advantage of a 32-bit architecture, if the targeted algorithms require arithmetic on larger data words. There exist many embedded signal processing applications however, which can efficiently be implemented on 16-bit architectures [Dog13], often due to the limited precision of the collected sensor data, for example from a low-power ADC providing only 12 bit. Consequently, the "computational value" of each operation/instruction is almost impossible to assess if no specific application is considered.

When considering power consumption and energy efficiency, this issue of assessing computation "value" is further exacerbated by the fact that different microarchitectures exhibit different instruction throughputs in terms of how many clock cycles per instruction (CPI) are required. This is sometimes even true when comparing different microarchitecture implementations of the same ISA. However, power is mainly reported in terms of µW/MHz (or often as consumed active current in µA/MHz) and energy efficiency is often reported as pJ/cycle. How much computation can be performed for one MHz of clock frequency can however vary wildly, not only due to data width and other architecture characteristics, but especially due to differences in the CPI performance. The popular MSP430 architecture is a prime example for this, since its CPI varies over the full ISA from 1 to 6, with a typical average CPI of 3-4. For this reason we compare energy efficiency in Table 3.4 in terms of pJ per operation instead of per cycle. Furthermore, it has to be noted that providing a typical energy consumption per operation can still be challenging since this average value also depends heavily on the considered application, i.e., the instruction sequences that are executed by the processor. Arithmetic heavy benchmarks for example can often consume high power, since they cause more internal switching. As an example, in the case of an ARM Cortex M0+ core utilized in [Atm16], the increase in power is 24% when running the CoreMark [EEM16] benchmark, in comparison to a while-1 loop (which effectively only executes branches and possibly NOPs). Consequently, it is important to compare energy efficiency for similar or ideally identical workload types, executed on the different cores.

Finally, standby power can be an important measure for deeply embedded devices, since although on-node processing is becoming increasingly important, there are still applications where the device spends more than 99% or even 99.9% of the time in some form of sleep or standby mode. This is especially true for circuits which are not able to operate at sub-threshold voltages (e.g., due to the chosen memory system), where the aim is to operate as efficiently and typically fast as possible during the active phase, which is then followed by a long sleep phase. It is furthermore often not only important that extremely low-power standby modes exist, but also what types of functionality are still provided by the MCU during operation in such modes, most importantly if state retention of work memories and CPU are possible with low consumption. Other aspects that can influence the overall power requirements depending on the application are energy costs for read and write access to the embedded non-volatile memories, which offer the possibility of full power gating for increased battery lifetimes in applications with very short duty cycles.

3.4 Multi-Core Processing

Multi-core processing is typically associated with systems of relatively high performance, such as general-purpose laptop, desktop, or server CPUs, and in recent years application processors used in smartphones and tablets. Nevertheless, it is possible to successfully apply multi-core processing to low-power WSN and IoT type systems as well, especially if the targeted signal processing applications contain significant parallelism. This is for example particularly the case for biomedical signal analysis, which often operates on multiple data sources, i.e., parallel channels of sampled data such as for example multiple leads of an ECG or electroencephalogram (EEG).

This section discusses multi-core processing in the context of low-power sensing platforms, with non-negligible on-node processing requirements (>1 MOPS). Specifically, we investigate the energy benefits that can be obtained from efficient memory organization in such multi-core architectures, on the microarchitectural level. This concerns both extensions to the employed processing cores as well as some core-external improvements of the memory architecture. In detail, extensions and microarchitectural support for configurable data memory mapping, instruction broadcast, and synchronized code execution are proposed in the following subsections.

3.4.1 Comparison of Single-Core and Multi-Core Processing

Before comparing the power and performance characteristics of single-core and multi-core processing, let us first define two reference single- and multi-core architectures that employed TamaRISC as the processing core [DCA+12].

Figure 3.11a shows a single-core architecture, which is comprised of a processing unit holding the core and an instruction memory, and a multi-bank data memory connected over a selection logic. The core requires a dual-port data memory, since it needs to be able to perform a read and a write access within the same clock cycle. In order to reduce memory access power, the data memory is built out of multiple banks of dual-port SRAMs. The simple selection logic connects read and write ports separately to the core, generating appropriate chip select signals out of the higher address bits, for the SRAM banks.

Figure 3.11b details a multiple instruction, multiple data (MIMD) multi-core architecture, based on the same processing units. N processing units are connected via a crossbar interconnect to M data memory banks, consisting of dual-port SRAMs. The crossbar is a mesh-of-trees interconnection network [RLKB11] to support high-performance communication between the cores and the shared data memory. In the case where two or more cores request access to the same memory port (read or write) on the same SRAM bank, an access collision occurs, and an arbiter stalls all but one of the cores. To reduce possible memory access conflicts, the number of banks M is chosen as $M = 2N$. For example, using 4 kB SRAM banks, a multi-core architecture with 8 processing units comprises 64 kB of tightly-coupled, shared data memory.

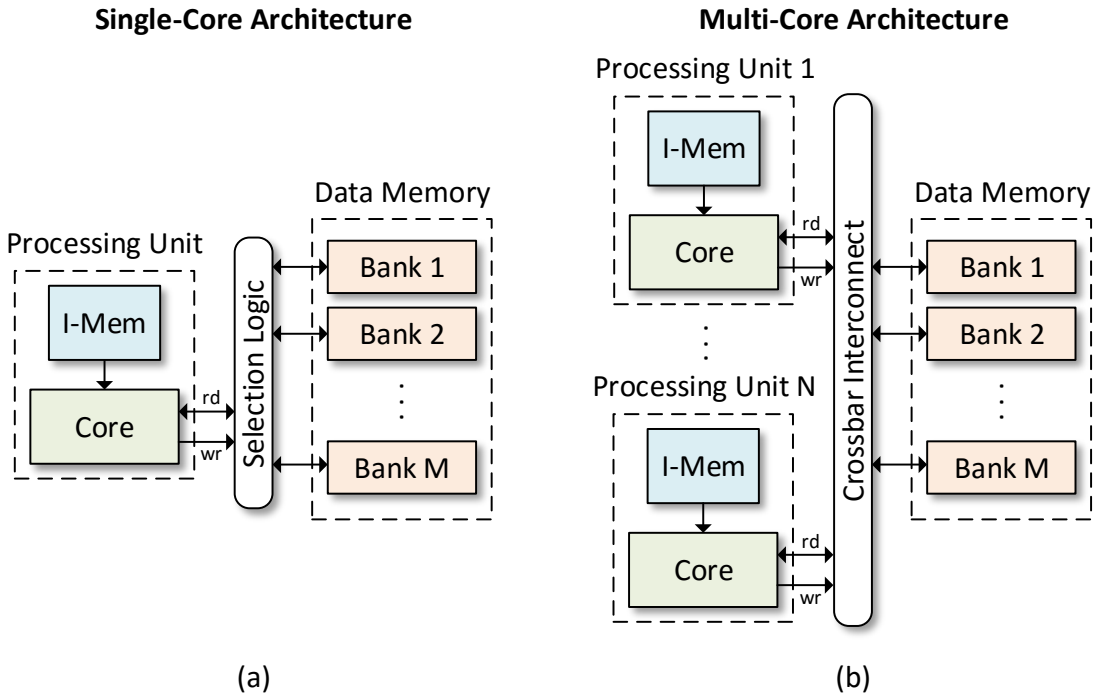


Figure 3.11 – (a) Single-core architecture and (b) multi-core architecture, both utilizing a multi-banked data memory system.

Considering embarrassingly parallel applications, such as multi-lead biosignal analysis, i.e., applications which have a number of parallel threads or tasks that can largely work independently from each other, multi-core processing can be exploited to significantly increase energy efficiency. The key observation when comparing single-core and multi-core architectures in this context is that N cores allow for processing of an N-fold higher workload compared to a single-core architecture at the same frequency. Consequently, for an equal workload the multi-core architecture allows for voltage-frequency down-scaling, providing lower power consumption. As long as the circuits are still operated in a regime where leakage power is insignificant (i.e., above- or near-threshold, and for reasonable high clock frequencies), the lower active power consumption is not offset by the increased leakage of the multi-core architecture and we obtain an overall better energy efficiency.

As we show in [DCA+12], based on the extensive PhD work of Ahmed Dogan [Dog13], multi-core architectures can outperform single-core architectures in terms of energy efficiency for workloads starting in the range of 1-2 MOPS, while at the same time supporting higher maximum workloads of hundreds of MOPS. Figure 3.12 details this result, with a comparison of single-core and multi-core behavior in terms of power consumption for given workloads. The multi-core architecture here comprises 8 identical cores, with the same amount of shared data memory as in the single-core configuration (64 kB). The point where single-core and multi-core power consumption is equal is reached with 0.03 mW at a throughput of 1.7 MOPS. Note that both architectures, implemented in a low-leakage 90 nm CMOS technology, operate

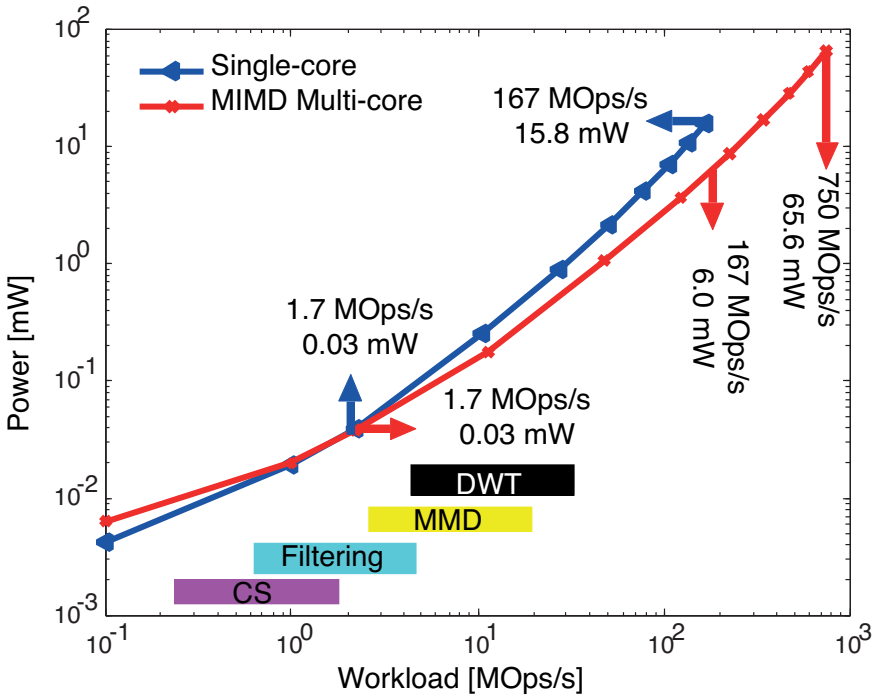


Figure 3.12 – Comparison of single-core and multi-core power consumption at given workload requirements [DCA+12], utilizing TamaRISCs as the processing cores. The workload operating ranges are indicated for the following biosignal analysis applications: compressed sensing (CS), morphological filtering, multiscale morphological derivatives (MMD), and discrete wavelet transform (DWT).

here at their minimum supply voltage that was chosen in the near-threshold regime at 0.5 V. At the peak performance of 167 MOPS of the single-core, the multi-core provides a reduction in power consumption of 62%. Moreover, the multi-core architecture is able to provide a significantly higher peak performance of 750 MOPS, supporting a wider range of applications.

3.4.2 Efficient Memory Organization

As shown for example in [DCA+12] and as summarized in Section 3.4.1, multi-core processing can provide significant energy benefits due to the ability to provide comparable throughput at scaled voltage with reduced clock frequency. Multi-core processing however provides additional opportunities for energy reductions, since the memory architecture can be optimized with the targeted embarrassingly parallel applications in mind.

Considering a multi-core architecture as depicted in Figure 3.11b, we observe that embarrassingly parallel applications lend themselves to be optimized regarding their memory access patterns, both for the instruction memory and data memory. The following subsections introduce a number of optimization techniques regarding efficient memory organization, which improve performance and/or provide energy savings, as developed together with my colleague

Ahmed Dogan. Complementary to [Dog13], the focus of the following lies more on the general architectural concepts, and the extensions that are involved on the microprocessor core side.

3.4.2.1 Configurable Data Memory Mapping

In a software implementation of a parallel signal processing algorithm on an embedded multi-core architecture, we differentiate two different types of data memory that are used by the multiple cores. On the one hand, we have private data, which typically comprises some form of working buffer and for example the data on the stack. This private data is hence typically only accessed by the corresponding core. On the other hand, we have shared data, which is used by all cores. This can include either tokens or complete buffers for inter-core communication, but also globally shared data that is required by each core, executing the same algorithm on different input data sets. An example for such shared data is a code book used in compression algorithms, which can consume a considerable amount of the limited total data memory. To conserve precious memory space, such data should not be replicated carelessly in the form of private copies (if even possible), but should rather exist as one copy that is accessible from all cores at the same time.

As a result, the aim is to partition the data memory address space such that one region is reserved and optimized in terms of access bandwidth for shared data, while N other regions are each private to the respective cores. While this approach seems straightforward to implement, problems arise due to possible access conflicts that can occur depending on how the memory address space is mapped to the set of parallel SRAM banks that constitute the data memory. For purely private access, the optimal organization is to address each SRAM bank linearly, from its physical address 0 to its full capacity, before addressing the next SRAM bank. Consecutive subsequent data words in the logical continuous private address space allocated to a core always map to subsequent data words in the same dedicated bank of the physical address space (except for crossings from one bank to the next), which avoids access collisions as long as the individual cores confine their accesses to their private memory regions. In an architecture with $M = 2N$ banks, each of the N cores hence would receive a private memory region of the size of 2 SRAM banks, which can always be accessed contention free. For shared memory access however, the optimal organization is word-by-word interleaving of the logical address space over all M banks of the corresponding physical address space. As a result, consecutive data words always map to different banks (modulo M), which reduces access conflicts of multiple cores that access the same shared data.

There is hence a conflict regarding which mapping to choose when both types of data must be supported. Opting for a non-interleaved/linear approach can significantly reduce performance, if the code contains many accesses to shared data, which for example all target a large array that is mapped to a single bank. All cores will then compete for access to this bank, resulting in many stall cycles. A fully interleaved mapping alleviates this problem, however it makes private access suboptimal, since even though other cores do not access the same data, they might access the same bank in the same cycle. The performance that is lost due to these

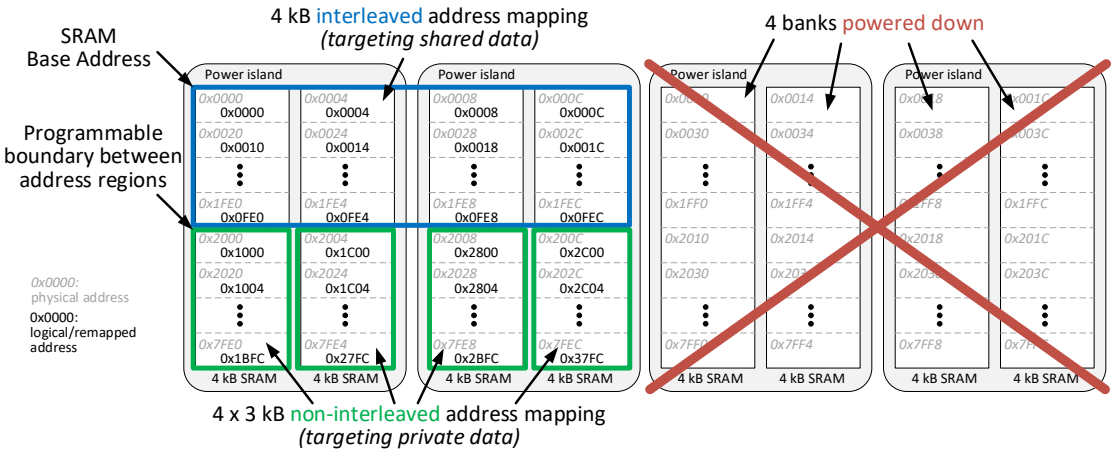


Figure 3.13 – Illustration of the configurable data memory mapping mechanism, providing improved access for a combination of shared and private data in a multi-core architecture.

conflicts can be significant, since private memory access is typically the most frequent type.

To address this inefficiency, we propose a data memory mapping mechanism that allows to combine both interleaved and non-interleaved mappings within the same memory architecture. To this end, extremely light-weight memory management units (MMUs)[8] are introduced to the processor, which allow an address translation from a logical address (used by the program code), to a physical address that is used on the memory bus to access the data memory banks over the crossbar interconnect. This dynamic translation of the memory addresses allows to define a shared section of freely configurable size at the beginning of the address space, with the remaining space divided up into private sections for the individual cores. In our memory architecture the default mapping is chosen to be interleaved banks, which results in a physical address format as illustrated in the top right of Figure 3.14. Moreover, we address the issue of address space holes that would occur, when a subset of the SRAM banks are powered down (power gated) in order to reduce the power consumption for applications with reduced memory requirements.

Figure 3.13 details an example configuration for the proposed dynamic address mapping, on a 4-core system with an 8-bank data memory. Here, 4 of the 8 SRAM banks, each of size 4 kB, are powered down via control of the corresponding power islands. The remaining 4 banks are divided into 5 sections, one shared section of size 4 kB interleaved over 4 banks for data that is accessed by all cores, and 4 separate core-private sections of size 3 kB each, mapped onto separate banks. The boundary between the shared section/region and the remaining address space can be dynamically configured[9], which allows for the memory partitioning to

[8] The term MMU is chosen here, since it is the most fitting, regarding the performed translation of logical/virtual to physical addresses. However, it has to be noted that the introduced units that perform this translation are not comparable in terms of complexity and hardware overhead with a traditional MMU of a general-purpose CPU supporting classical address space virtualization, with the help of page tables, TLBs, and so forth.

[9] The boundary must lie on an address that is a power of 2.

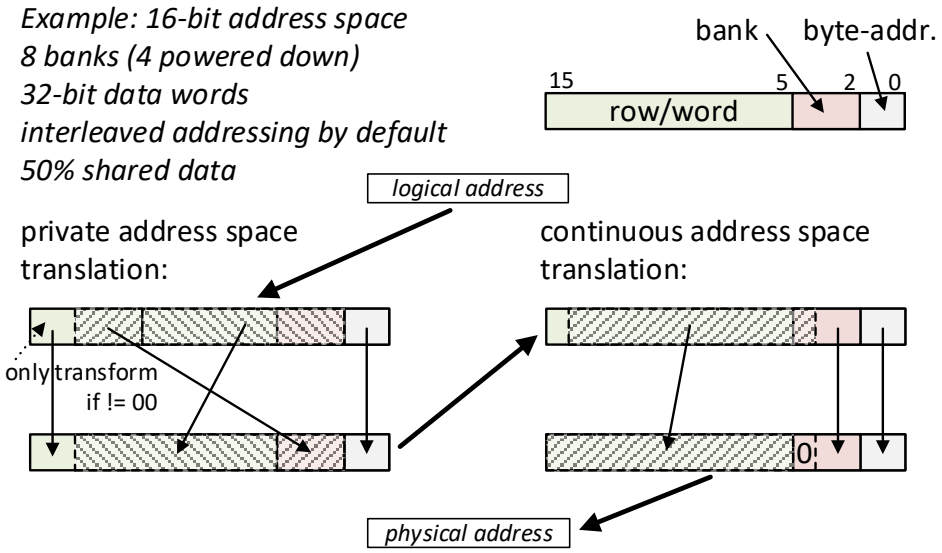


Figure 3.14 – Translation of virtual to physical addresses using private & continuous address space translation within the MMU.

be optimized for the currently executing application.

The details of the address translation mechanism are illustrated in Figure 3.14. The MMU first decides if a private address space translation is necessary, by checking if the accessed virtual address is within the shared section or not. Only if the boundary for the start of the private region is reached, the MMU performs the private address space translation, which places the upper most address bits (of the relevant part of the address) onto the bits that select the physical memory bank, to achieve a non-interleaved memory mapping. The remaining lower bits of the address are consequently shifted up, to form the new address, while the two lowest bits are always kept in place, since they address the individual bytes within the 32-bit data words considered for this example. The MMU additionally then performs a continuous address space translation, to assure a fully continuous address space regarding the virtual addresses that the application uses, as illustrated in Figure 3.13. Note that no continuous address space translation is necessary, if all memory banks are active.

The integration of the MMUs into a processing core is shown in Figure 3.15. There are two separate MMUs required per core, one for the read address translation and one for the write address translation. Since both read and write access need to be supported within the same cycle for the considered core, the MMU hardware cannot be shared. Both MMUs are configurable via a single configuration register, which holds information on the size of the shared data section (region boundary), and the number of active memory banks. The section size information is utilized for the conditional private address space translation, while the active bank count controls the way the continuous address space translation is performed. Note that these configuration values need to be kept in-sync across all cores of the multi-core architecture, which can be assured either via software (e.g., during task initialization), or via a

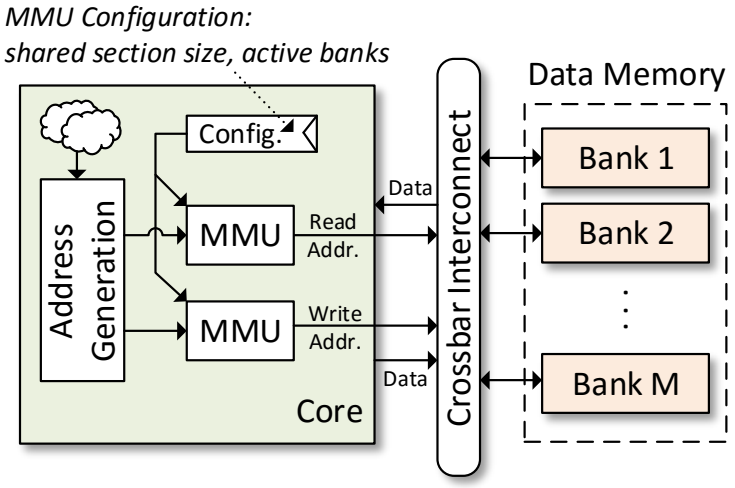


Figure 3.15 – Integration of MMUs into the core microarchitecture, to enable configurable data memory mapping.

hardware mechanism that implements a single memory mapped peripheral register within the full multi-core architecture, providing the same configuration value to all cores at once.

The presented configurable data memory mapping mechanism has been successfully employed in an 8-core 16-bit architecture based on TamaRISC and aimed at ultra-low-power wearable health monitoring systems [DCR+12], as well as in an energy efficient 32-bit near-sensor computing platform [RPL+16] (named PULPv2). PULPv2 is a full SoC implementation in 28 nm FD-SOI of an ultra-low-power parallel computing platform, which utilizes a tightly coupled data memory architecture for its cluster of four cores. This data memory architecture is in general similar to the presented banking mechanism (with some further extensions), and employs the discussed light-weight MMUs, enabling not only performance improvements due to reduced access conflicts, but also enabling power gating and in turn leakage reduction for memory banks that are currently not utilized.

3.4.2.2 Data and Instruction Broadcast

Let us consider the ideal case of an embarrassingly parallel application, where N identical programs/tasks operating on different input data, are executed in parallel on N cores. If any potential private memory access conflicts are eliminated utilizing the techniques presented in Section 3.4.2.1, the programs on the N cores run in-sync, i.e., each core is executing the same instruction in parallel, until they reach a memory access to shared data.[10] Since all cores access the same memory bank at the same time, all but one of them have to be stalled. However, since all cores do not only access the same bank, but also the same word within the same bank, this problem can be addressed. In the case of a read access, the data word

[10] This includes the assumption for now that the application code does not perform branches which depend on the input data (e.g., sensor data). This limitation is addressed in Section 3.4.2.3.

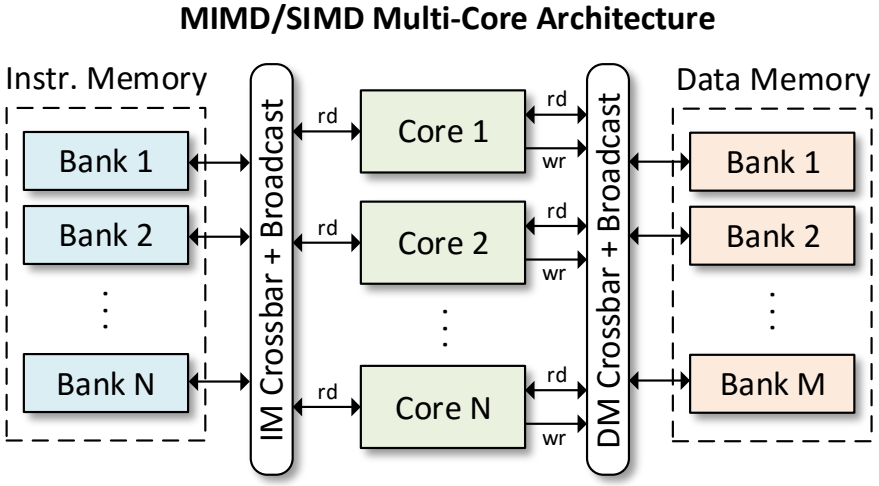


Figure 3.16 – Multi-core architecture with shared instruction and data memories, each connected over a crossbar with broadcast support. The architecture is capable of operating both in standard MIMD, as well as in SIMD mode.

can be read from the memory bank, and then broadcasted to all cores. In practice this means that the read access is accepted by the arbiter, and the read result is put on the data bus as normal. Any cores reading this same data word are furthermore not stalled, but can continue operation uninterrupted. In the case of a write access to the same word by multiple cores (which should not occur, if the system does not include any provisions for atomic memory operations, such as read-modify-write), any one of the provided values can be written, and the others discarded.

The data memory broadcast feature can be implemented via comparators on the address buses of the crossbar interconnect. When matching addresses are detected for accesses of the same bank, the accesses can be granted for all matching requests in the same cycle. Note that this broadcasting also reduces the number of accesses to the data memory, and hence saves energy. More importantly, this allows now in theory for the application to be executed completely in lockstep over all cores.

To further extend the concept of broadcasting of concurrent memory accesses, we enhance the multi-core architecture presented in Figure 3.11b with an additional crossbar interconnect on the instruction memory side. Figure 3.16 depicts the enhanced multi-core architecture, which removes the private instruction memory of each core, and replaces it with a multi-bank instruction memory shared between all cores. The instruction memory can now be operated either in interleaved or banked (non-interleaved) mode, similarly to the data memory, as discussed in Section 3.4.2.1. The architecture can be used in classic MIMD mode as before, with the instruction memory partitioned in a banked fashion, guaranteeing one private bank of instruction memory per core. However, the strength of the architecture is that it can also be operated in single instruction, multiple data (SIMD) mode, where all (or a subset of) cores are executing the same task, operating on different private data. When operating in SIMD

mode, the instruction memory organization can also be chosen as banked, as in the MIMD case, together with instruction broadcasting that works analogous to the presented data broadcasting. This again allows to power gate instruction memory banks that are not utilized for the current application. Moreover, note that the potential maximum program size for SIMD applications that utilize instruction broadcasting is increased by a factor of N, in comparison to MIMD operation with private instruction banks. Note that an interleaved organization of the instruction memory can also be a useful organization for both MIMD and SIMD operation, where all cores still run the same program (which now can exceed the size of a single bank), but lockstep operation cannot be guaranteed.

Besides providing more possibilities for applications with larger program memory requirements, the main issue that is addressed by this enhanced MIMD/SIMD architecture with shared instruction memory, is the overall energy consumption of instruction memory accesses. As was shown in [DCR+12], in the multi-core architecture depicted in Figure 3.11b the instruction memories have a 56% share of the total power consumption of the architecture. This can be reduced down to a share of only 13% of the total consumption with the use of the enhanced architecture operating in SIMD mode with broadcasting and banked partitioning of the instruction memory. Moreover, the total power consumption is reduced by 40%, which can be mainly attributed to the significant savings in instruction memory energy consumption.

One aspect that has not been discussed yet, regarding the proposed SIMD mode, is how private data memory accesses can be handled correctly. Since all cores are now executing not only the same program, but the exact same binary code, there is no possibility any more to access core-private data structures (such as the stack, or a sensing buffer) via differences in the compiled software. While on the standard MIMD architecture, each core can have its own version of the binary code which has the correct memory addresses embedded during linking, for private data memory accesses of the respective core, in SIMD mode a single binary must be able to provide the same functionality for all cores. This problem is solved via an extension to the MMU mechanism presented in Section 3.4.2.1. Both MMUs depicted in Figure 3.15 are extended such that they can provide core-private data memory access during SIMD operation.

To this end, a unique id from 0 to $N-1$, which can be read internally, is assigned to each of the N cores. This id is used during private address space translation, and inserted as hardcoded bank bits (compare Figure 3.14). The single program binary which is used for SIMD operation is then compiled as if it would be executed on the core with id 0 (or any other core id for that matter). This means that for this binary the higher (relevant) address bits are always 0, if equally sized private memory segments are used per core (which is required for SIMD operation). These zero bits are then replaced during address translation with the core id. In essence, the SIMD mode with broadcasting can hence be supported from a microarchitectural level by a very minor addition to the already present MMU inside the processor core.

3.4.2.3 Synchronized Code Execution

Since SIMD operation as presented in Section 3.4.2.2 can provide significant energy gains mainly through coordinated instruction memory access with broadcasting, the aim is to maximize the amount of code that can be executed in lockstep. To this end, we propose in [DBC+13] a synchronization mechanism that allows to keep the cores in-sync as much as possible, in the general case.

Issues arise, when conditional branches are taken that depend on the input data of the algorithm. This is the case for signal processing applications that go beyond basic, fixed transformations of the signal data, but include other operations such as sorting, or threshold-based iterations. As shown in [DBC+13] it is however often possible to re-synchronize applications at predefined checkpoints in the application, where lockstep operation can be resumed for all cores.

The idea is to annotate the source code of the application with such checkpoints, specifically a combination of check-in and check-out markers, which enclose the data-dependent sections of the code. When a core passes a check-in point it marks its identity and increases a counter in a specific shared data structure that is assigned to this section. Once a core reaches the check-out point for this section, it decreases the counter and goes automatically to sleep, until all the cores that are part of this section have reached the corresponding checkpoint. When the last core reaches the check-out point, all cores that have traversed this section of code are woken up to resume execution of the subsequent code in lockstep. The core identities are stored via single bits that are set within a byte for an 8-core architecture. A second byte is utilized to hold the counter for the check-in/check-out pair. These two bytes build the data structure that is required per annotated data-dependent code section. An array with these 16-bit values is stored at a fixed address in data memory, namely the base address of the synchronization markers.

In theory, such a mechanism could be purely implemented in software[11], with the application on each core modifying the described synchronization marker data structures as necessary. There are however two issues that need to be addressed to make this mechanism efficient. The first is the required overhead in terms of instructions and cycles for passing a check-in or check-out point. The second is the issue of multiple cores reaching the same checkpoint at the same time. To address both these aspects, we propose the use of two dedicated ISE, together with a read-modify-write extension for the data memory system, here implemented for the TamaRISC architecture:

- `SINC #index`: This instruction is used for a check-in point in the code. It marks the entry for the predefined code section with the id `#index`. The ISE introduces a new dedicated special register, which holds the base address of the synchronization markers. The correct data structure is hence automatically addressed by combining the id with

[11] In conjunction with a dedicated sleep instruction and interrupts.

the base address. The instruction fetches the 16-bit marker, and forwards it to a core-external synchronizer unit, which sets the appropriate bit to identify the core, and increments the counter embedded in the marker. If multiple cores execute the same `SINC` instruction in the same cycle, the synchronizer unit is capable of merging the separate increment requests correctly. The modified marker is then written to the data memory. This is achieved in an atomic fashion via a lock signal that the data crossbar receives from the core. This bus lock signal is asserted for two clock cycles, while the instruction resides in the decode and execute stage.

- `SDEC #index`: The instruction works analogous to the `SINC` instruction, but marks the exit (check-out point) for the predefined code section with the id `#index`. The ISE consequently decrements the counter of the respective synchronization marker. Furthermore, the core automatically enters the sleep mode with execution of the `SDEC` instruction. If execution of an `SDEC` instruction causes the synchronizer unit to decrement the counter down to zero, a wakeup signal is asserted by the synchronizer unit for all cores that are indicated in the marker.

As a result of the presented hybrid hardware/software code synchronization technique, biosignal analysis applications that exhibit considerable data-dependent code sections, can be accelerated by 1.85x - 2.38x, depending on the application, over an architecture running in SIMD mode without synchronization. This reduction in memory conflicts and consequently number of stalls, can be leveraged to further reduce power consumption by 50-64%, for equal workload requirements [DBC+13, Dog13].

4 Dynamic Timing Margins in Embedded Microprocessors

As has been illustrated in Section 1.2, Chapter 2, and Chapter 3, processing performance and energy consumption of a digital circuit go hand in hand. This means in particular that any increases in potential processing performance can typically be traded off for energy savings through voltage/frequency scaling. Power benefits therefore arise from circuits that are optimized towards the most efficient use of their processing cycles.

When operating a synchronous circuit, its maximum fixed clock frequency — i.e., its performance — is determined by the maximum path delay that can occur between any two sequential elements in the circuit, for a given set of operating conditions. This maximum delay depends on many factors, and is also subject to variations from a host of different sources. While classically during the design of a sequential circuit, the main focus from a timing perspective is put on the worst possible path/delay, we note that a dynamic or statistical view of the circuit timing can open up many potential opportunities. Starting from the main observation that maximum delay paths are not excited in every clock cycle, we investigate in this chapter the dynamic changes in the timing of synchronous circuits, and in particular of embedded microprocessors. In contrast to the extensive recent work in the literature that has been done regarding mitigation of process, voltage, and temperature (PVT) variation effects on timing, we focus here on the dynamic effects on timing due to the information that is processed by a circuit, i.e., its changes in state from one cycle to the next.

This chapter starts with a discussion on timing margins in synchronous circuits in Section 4.1. The discussion provides an overview of variation effects and related mitigation techniques from the literature, and then continues to introduce state-dependent dynamic timing margins, and our dynamic timing analysis (DTA) method. Section 4.2 afterwards focuses on DTA specifically for microprocessors. In Section 4.3 DTA is applied to enable detailed characterization of application behavior under frequency/voltage over-scaling, with a case study for an OpenRISC microprocessor.

In the subsequent Chapter 5 we then demonstrate a way to exploit uncovered dynamic timing margins for both performance gains and energy savings in a microprocessor.

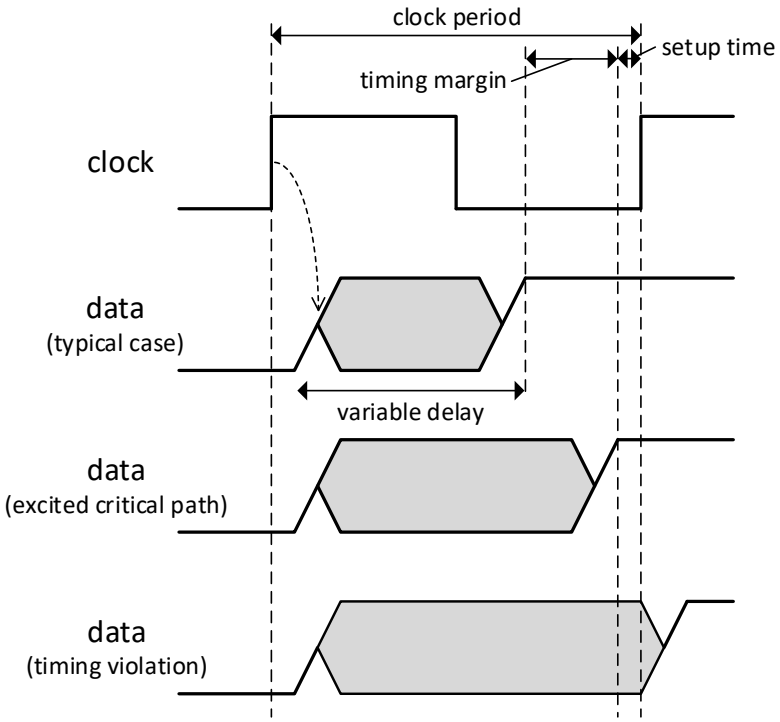


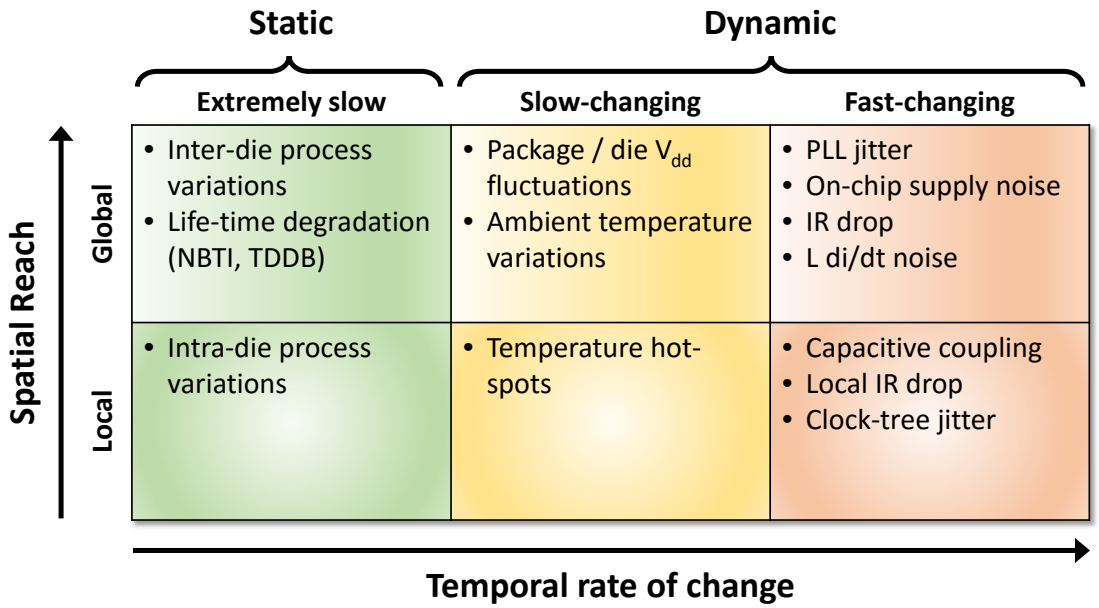
Figure 4.1 – Timing diagram, illustrating a timing margin, critical path delay, and timing violation.

4.1 Timing Margins in Synchronous Circuits

We define the *timing margin*[1] of a signal within a clock cycle to be the difference between the latest time at which the signal at the input of a sequential element is allowed to change without causing a timing violation and the time at which the signal actually settles to a stable value. This margin is illustrated in the timing diagram shown in Figure 4.1. Within a clock cycle of a positive-edge-triggered circuit, the latest time at which the data signal is allowed to change is the clock period minus the setup time of the input of the sequential element (e.g., a register or memory) that samples the data signal. Let us define the input of any sequential element of the circuit as an endpoint. The timing margin at a specific endpoint in a clock cycle hence depends on the delay of the longest *excited* path to that endpoint, which can vary from cycle to cycle. The critical path of the circuit when excited exhibits the longest possible delay, bringing the timing margin for that endpoint down to zero, and therefore determines the minimum clock period. If a path encounters a degradation in its delay (or the clock period is chosen too short/aggressive), a timing violation can occur, due to the data signal settling too late, and thereby violating the setup time. Such a timing violation can result in functional failure, since either a wrong value is sampled at the endpoint, or the sequential element becomes metastable, i.e., its output value does not settle to a stable logic level of 0 or 1 within the defined maximum propagation delay of the element, causing potentially unbounded delays

[1] This timing margin is often also referred to as *slack*.

Table 4.1 – Classification of variations, regarding their temporal rate of change and spatial reach (table cited with minor adaptations from [BDS+11, DBW15]).



for the subsequent paths.

The following Section 4.1.1 gives an overview of static and dynamic variations, which cause degradation in the path delays of a circuit and, in turn, can lead to timing errors/violations. Moreover, classical and emerging approaches to cope with these variations are discussed. In Section 4.1.2 we then change perspective and look at dynamic timing variations that emerge due to the different states that a sequential circuit traverses. This is followed in Section 4.1.3 by the introduction of an analysis method to characterize these dynamic timing margins.

4.1.1 Variations, Guardbanding, and Mitigation Techniques

Variations in integrated circuits and their underlying CMOS devices are manifold, affecting directly both power consumption and circuit timing. However, with degraded circuit timing, power efficiency further suffers also indirectly as a consequence. Since the 65 nm technology node, the major contributor to variations is parametric process variations, causing a strong shift in design practices towards more variation-aware design, to meet power, speed, and reliability constraints [GR10, ARA11, KKT13].

Besides process parameters, two other main factors contribute to performance variations, namely voltage and temperature. An overview and classification of PVT variations regarding their temporal rate of change and spatial reach is provided in Table 4.1 (cited with minor adaptations from [BDS+11, DBW15]). The two main categories are static and dynamic variations. Static variations are variations that are either fixed for a given fabricated die, i.e., process

variations, or only change extremely slowly, such as lifetime degradation due to negative-bias temperature instability (NBTI) or time-dependent dielectric breakdown (TDDB), and are hence considered static during operation. Dynamic variations on the other hand affect the circuit at runtime, and can be grouped into slow-changing and fast-changing variations. Slow-changing variations cause changes in transistor behavior over many cycles (ms to µs range), while fast-changing variations affect transistor performance within a single cycle (ns to ps range). Temperature effects are generally categorized as slow-changing variations, together with die-external voltage supply fluctuations. Fast-changing variations include noise effects on the on-chip power delivery network, e.g., due to supply ripple of integrated DC-DC converters, as well as IR drop and L di/dt noise effects [Zhu04]. Clock distribution related jitter, and capacitive coupling effects are additional very fast-changing variations, which can cause significant timing degradation. Variations can furthermore be distinguished regarding their local or global effects. Local variations only affect groups of transistors in a local region of the die in the same way, while global variations generally affect all transistors similarly. Examples for global variations are changes in ambient temperature, supply voltage noise, or inter-die variations, due to fluctuations in transistor channel dimensions and oxide thickness [GR10]. Dynamic local variations are mainly caused by (instantaneous) strong local switching activity, which gives rise to local IR drop, and temperature hot-spots. Static local variations are caused by intra-die process variations, induced by line edge roughness and random dopant fluctuations [GR10].

Guardbanding

The classic approach to handling the discussed variations and uncertainties is the introduction of guard bands, which enforce additional timing margins to ensure functionally correct behavior of the circuit under all possible conditions. However, minimum performance requirements in terms of circuit speed still need to be fulfilled, which typically requires the acceleration of path timings, e.g., through gate up-sizing or by adding supply voltage margins, to meet the corresponding tougher timing requirements/constraints. Additional timing margins are typically enforced through so-called derating factors, in their most basic form with one factor each for statistical process, voltage, and temperature variations (K_P, K_V, K_θ) [Kae08].[2] The derating factors K_P, K_V, and K_θ describe by how much the nominal timing $t_{nominal}$ is affected, due to differences in process corner, supply voltage, or junction temperature, respectively. The derated timing $t_{derated}$ is then calculated by multiplying all derating factors together [Kae08]:

$$t_{derated} = t_{nominal} \cdot K_P K_V K_\theta \tag{4.1}$$

As an example, in a 130 nm CMOS technology (which has still a large enough feature size to not be considered to be significantly affected by variation issues), best case PVT conditions (fast corner at 1.32 V and -25° C) result in a 59% faster timing, while worst case conditions (slow

[2] Derating factors are also used to capture other variations, such as clock distribution uncertainties, related to clock generation and the clock tree.

corner at 1.08 V and 125° C) result in a 83% slower timing, compared to nominal conditions (typical corner at 1.20 V and 25° C) [Kae08]. Operation with worst case conditions is hence 2.9x slower than operation in the best case.

This basic derating approach however assumes all devices are equally affected by variations, and hence does not capture local variations adequately, requiring to add additional guard bands to also account for on-chip (intra-die) variations in process, as well as degradation caused by localized switching activity. With increasing variations in deeply-scaled nm technology nodes, guardbanding hence leads unfortunately to excessive over-design and over-margining of circuits, incurring high power costs and sub-optimal design.

Mitigation Techniques for PVT Variations

Up until the 65 nm node, static timing analysis (STA) based on worst-case path delays for given operating conditions, was the main tool of choice to determine circuit timings. While basic STA still provides the foundation for most circuit timing analysis, a strong increase in on-chip variations in newer nodes gave rise to a statistical approach to STA, so-called statistical static timing analysis (SSTA) [OK02, VRK+06]. SSTA captures on-chip variations, by representing gate and wire delays not as fixed (worst case) values, but as statistical distributions of delays. These distributions, which are provided by SSTA-enabled standard cell libraries, and statistical information on metal interconnects of the process, are then combined along a timing path to determine a probability density function for the total path delay, which is dependent on process variations. As a result of the more accurate analysis, which is not any more based on pure worst-case (but highly unlikely) fabrication outcomes, better design constraints can be derived, and the important paths which require further optimization to meet the overall design goals can be identified.

Moreover, there exist novel modeling techniques which try to capture the effects of parameter variations on timing errors from a microarchitectural perspective, to assist the designer early in the design process. Examples are VARIUS [SGT+08], and VARIUS-NTV [KKKT12] for assessment of increased variation effects at near-threshold operation.

Furthermore, there are a wide range of different circuit techniques that have been devised in the last decade to address the problem of increasing PVT variations, especially also targeting dynamic variations. The aim of such techniques is to include circuit enhancements which allow the dynamic adjustment of the circuit at runtime, typically in terms of clock frequency and/or voltage (supply or body biasing). This adjustment at runtime aims at tuning/calibrating the circuit towards its optimal operating point, in terms of performance or energy efficiency, effectively reducing the over-margining effects introduced by guard bands at design time to a minimum.

One of the key innovations in this area, especially for microprocessor circuits, is the approach to detect and correct timing errors in situ, i.e., within the circuit where they occur during

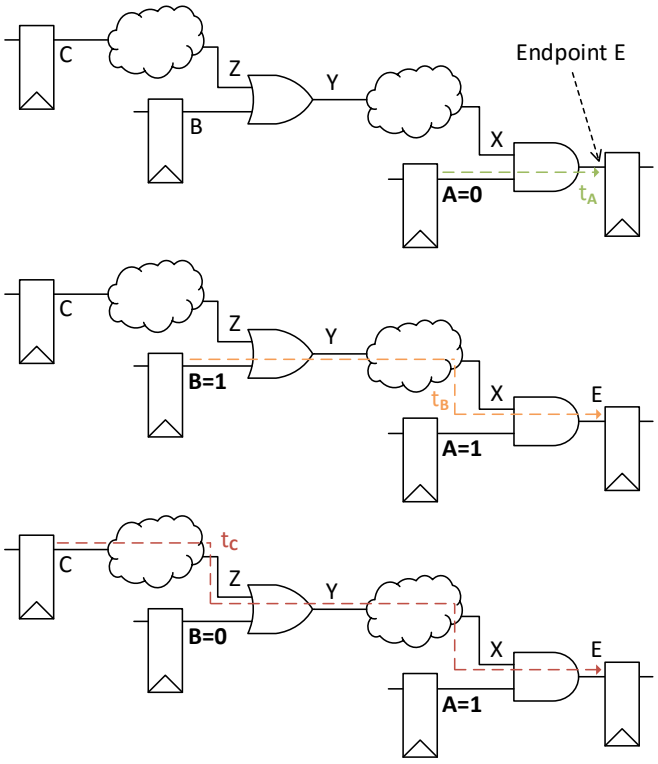
operation. Timing errors are detected by the use of special error-detection flip-flops, and either a microarchitectural or circuit mechanism thereafter allows the correction of the timing errors to ensure 100% correct operation of the circuit. The rate of the timing errors is then utilized to tune the clock frequency and/or supply voltage, using DVFS, to increase the performance of the circuit, or reduce its power consumption through voltage scaling. This approach was pioneered in 2003 by the Razor flip-flop [EKD+03], and is followed by a long line of research on Razor-like techniques and their continuous improvements. The Razor family includes:

- the original Razor [EKD+03, EDL+04] flip-flop with integrated error correction;
- Razor II [DTP+09, BDS+11], improving upon Razor through error correction via microarchitectural replay;
- Bubble Razor [FFK+13], introducing an automated insertion flow, allowing architecture independent error detection and correction;
- Razor-Lite [KKF+14], reducing error detection hardware overhead via a supply rail based solution requiring only 8 additional transistors per flip-flop;
- iRazor [ZKY+16], further reducing overhead to only 3 transistors, via a current-based design.

Other Razor-like designs targeted at in situ error detection and correction have been proposed in the literature, such as: error-detection sequentials (EDS) [BTK+09, BTL+11], which improve on metastability issues, and self-checking flip-flops [EHG+07] for use with a dual-V_{dd} architecture. Moreover, [SKS15, SKLS16] introduce a modified version of the Razor flip-flop for one-cycle error correction using pulsed latches. Finally, HEPP [ZLL+13] presents a timing error prediction and prevention technique.

Another approach to circuit-level timing error mitigation techniques is the use of replica paths with timing monitors. Here, typically the critical path of the design is replicated on-chip and monitored to allow correction of frequency and/or voltage in case that timing margins on this replica path are used up, due to variations. Extensive work in this area has been done in the context of IBM POWER CPUs, for example in the form of a critical-path timing monitor (CPM) for a high-performance POWER6 multi-core microprocessor [DSD+07]. In this work, the CPM is also used to measure intra-die process variations, and is hence inserted at 8 different positions within one POWER6 core. This work on CPMs was further developed for POWER7 [LDF+11] and POWER7+ [LDF+13, DFW+13] server CPUs, achieving accurate active guardband management with clock frequency adjustments within several cycles. Tunable replica circuits can be either employed together with time-to-digital converters [BTT+11] to keep constant, but safe margins, or used in conjunction with EDS and instruction replay to provide an efficient solution for variation tolerance in microprocessor pipelines [TBW+09]. Additionally, [RTB+11] introduces the concept of tunable replica bits, to provide read and write error detection for the microprocessor cache made from 8T SRAM.

Finally, due to the effectiveness of adaptive biasing techniques [TKN+02], thermal sensors and voltage droop detectors are employed in [TKD+07] to dynamically control the body bias



Previous state	A	1	-	-	-
	B	-	0	-	-
	C	-	-	0	1
Current state	A	0	1	1	1
	B	-	1	0	0
	C	-	-	1	0
Max. excited path delay		t_A	t_B	t_C	
Min. dynamic timing margin		$t_{clk} - t_{su} - t_A$	$t_{clk} - t_{su} - t_B$	$t_{clk} - t_{su} - t_C$	

Figure 4.2 – Illustration of state-dependent dynamic timing margins for an example circuit.

voltage of the circuit, to counter dynamic variations as well as aging effects.

4.1.2 State-Dependent Dynamic Timing Margins

The discussed variations and mitigation techniques all focus on PVT variations, i.e., either static variations that are fixed for a specific fabricated die, or dynamic variations which are caused due to operating conditions/environmental effects on the circuit.[3] However, for a given endpoint in the circuit, the timing margin that is available in any given cycle dynamically changes not only due to above-mentioned dynamic variation effects, but also strongly dependent on the state[4] of the sequential circuit. In the following, we define margins that fluctuate depending on changes in state alone, as *dynamic timing margins*.

As an example of a dynamic timing margin at an endpoint E, consider the sub-circuit depicted in Figure 4.2. If the circuit transitions from any state where A=1 to a state in the subsequent cycle where A=0, the maximum path delay that needs to be considered to derive the dynamic timing margin is the delay t_A from the flip-flop output A to the endpoint E through the AND gate. The node X can still toggle at any time later in the cycle, but since A=0 this will have

[3] Strictly speaking, dynamic variations that are for example caused by local IR drops, also exhibit data dependencies/correlations, however typically to a lesser extent than state-dependent dynamic timing margins.

[4] The word *state* here also comprises the momentary values of any primary inputs a circuit or module has.

no effect (i.e., will not lead to any activity) on E. The minimum dynamic timing margin in this case is hence $t_{clk} - t_{su} - t_A$, where t_{clk} is the clock period, and t_{su} is the setup time of the endpoint E. This calculated dynamic timing margin represents only the lower bound, i.e., the margin can be even larger, since for example the node X could have settled already to 0 in the previous state, and hence E would already be at its final value at the beginning of the current cycle, and will never switch, if it is guaranteed that the transition from 1 to 0 on A propagates to the AND gate prior to any switching that might occur on X during the cycle. In this case E will never toggle during the whole cycle, and the dynamic timing margin would be the maximum possible ($t_{clk} - t_{su}$). If we consider now A to be fixed to 1, and look at a state change where B transitions from 0 to 1, we can similarly derive the maximum excited path to go from B to E, with path delay t_B. Here, Y is held at 1 due to B=1, even if Z toggles at a later point in the cycle. Since A=1, any changes in Y that might cause changes in X, continue to propagate to E. Similarly to the previous case, the dynamic timing margin $t_{clk} - t_{su} - t_B$ represents only the lower bound, since Z might be 1 during the start of the cycle, causing Y to potentially see no transition. Finally, if the circuit transitions into a state where A=1 and B=0, and where C toggles its value, the dynamic timing margin of the endpoint E is now determined by the path delay t_C, reaching from C all the way to E. From this example we can observe that the dynamic timing margin at an endpoint can be highly dependent on the performed state transitions, i.e., on the previous as well as current state of the circuit for a given cycle.

Dynamic timing margins are interesting, since they can provide a statistical view of the circuit timing during runtime, in contrast to SSTA, which provides a static view in terms of different possible circuit fabrication outcomes. To illustrate these differences in modeling, three different approaches towards modeling of timing uncertainties are compared in Figure 4.3 in the three columns of the figure. The figure consists of illustrative probability density functions (PDFs), representing probabilities that are assumed in relation to the relative path delay impact for different variation sources. The different impact categories/sources that are distinguished for single paths leading to one endpoint are: static physical variations (local, intra-die), dynamic physical variations, and state/data-dependent variations. The PDFs of these three categories are then combined (assuming independence of the variation sources) to derive the PDF representing the total variations for a specific fabricated die and endpoint in the circuit. Note that while the first two variation categories apply to single paths (of which multiple lead towards the same endpoint), the state-dependent variations are better represented by talking about single endpoints, since they are mainly characterized by which of the many paths towards one endpoint are activated in a given cycle.

On the left hand side of Figure 4.3 we see the classical circuit timing approach based on STA, which conservatively considers worst-case delays regarding all aspects contributing to variations in timing. Going from top to bottom of the figure, we see that first, the worst-case gate compositions and wire delays are selected along timing paths. Worst-case effects are then assumed for dynamic physical variations, e.g., by taking worst-case values from the IR-drop analysis of the power grid. Regarding state-dependent variations, for each endpoint only the longest possible path to the endpoint is then considered. This includes the worst possible

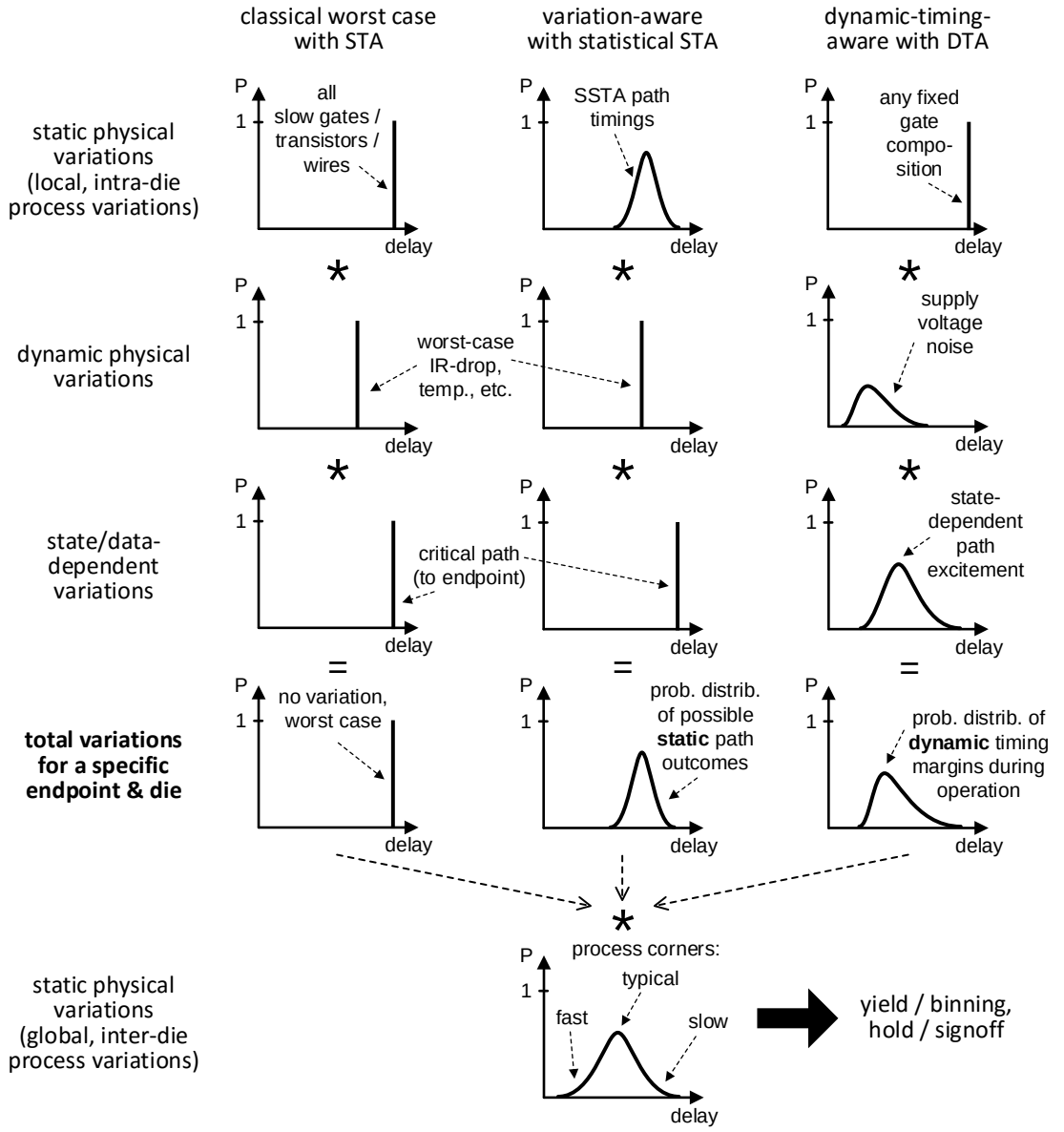


Figure 4.3 – Static and dynamic views of delay variation modeling, using proper combination of probability density functions of different effects.

combination of switching transitions (rising and falling edges) along the analyzed paths, to derive the highest possible delay, i.e., the smallest timing margin for that endpoint. Since all involved PDFs in this case are shifted Dirac delta functions, the resulting PDF is a shifted Dirac delta function as well, representing the worst-case delay.

In addition, independent of the variation modeling view, global, inter-die process variations are then always considered for the full circuit, to derive expectations for yield and speed/power binning. Moreover, the fastest process corners are used to ensure hold constraints are met

under all possible conditions and achieve successful timing signoff.

A variation-aware static view utilizing SSTA is depicted in the middle column of Figure 4.3. It differs from the classical worst-case view with STA, by considering delay distributions for the single gates and wires along a path. SSTA provides the means to efficiently combine these distributions, resulting in a delay distribution that represents the static physical variations of a path, due to intra-die process variations. The resulting PDF for the total variation consequently describes the probability distribution of all possible static fabrication outcomes for a path.

In order to not mix fabrication related static variation issues, with the dynamic circuit characteristics we want to study, the presented view on the right hand side of Figure 4.3, which we will adopt in the following, focuses only on dynamic variation effects. To this end, we assume a fixed gate composition with a deterministic delay for any given path (e.g., a worst-case composition, or a mean path composition as it would be derived from SSTA), and then include dynamic physical variations, as well as state-dependent variations. Dynamic physical variations can also be assumed as worst cases, which still allows to give insight to the dynamic timing behavior. However, the inclusion of models for dynamic delay degradation effects (such as supply voltage noise) can be helpful to understand the relative importance of the dynamic physical and state-dependent variations. The state-dependent variations, as they have been illustrated in Figure 4.2, are derived via our dynamic timing analysis (DTA) method, which is introduced in Section 4.1.3. As a result, the dynamic-timing-aware view on the circuit captures the probability distribution of the dynamic timing margins occurring during the operation of the circuit.

Note that the variation-aware and dynamic-timing-aware approaches in no way exclude each other, especially since the dynamic-timing-aware view does not aim at solving the same problem that SSTA does. Consequently, a combination of the approaches can potentially be beneficial, as it can provide even better modeling accuracy.

4.1.3 Dynamic Timing Analysis

In order to derive statistics from a given realization of a sequential circuit that describe (state-dependent) dynamic timing margins, we introduce an analysis method based on gate-level simulation, namely dynamic timing analysis (DTA). The purpose of DTA is to capture via a simulation of the circuit the dynamically occurring timing margins in each cycle at each endpoint. These captured timing margins are then aggregated into individual statistics for each endpoint. For accurate results, DTA should be performed on a placed & routed netlist of the design, which incorporates a real clock tree and all relevant wire delays. To derive useful statistics, the input vectors utilized for the gate-level simulation should be of sufficient length to cover enough cycles, and correspond to representative inputs for the targeted mode of operation of the circuit.

Figure 4.4 details the concept of DTA with a timing diagram, showing the global clock of the

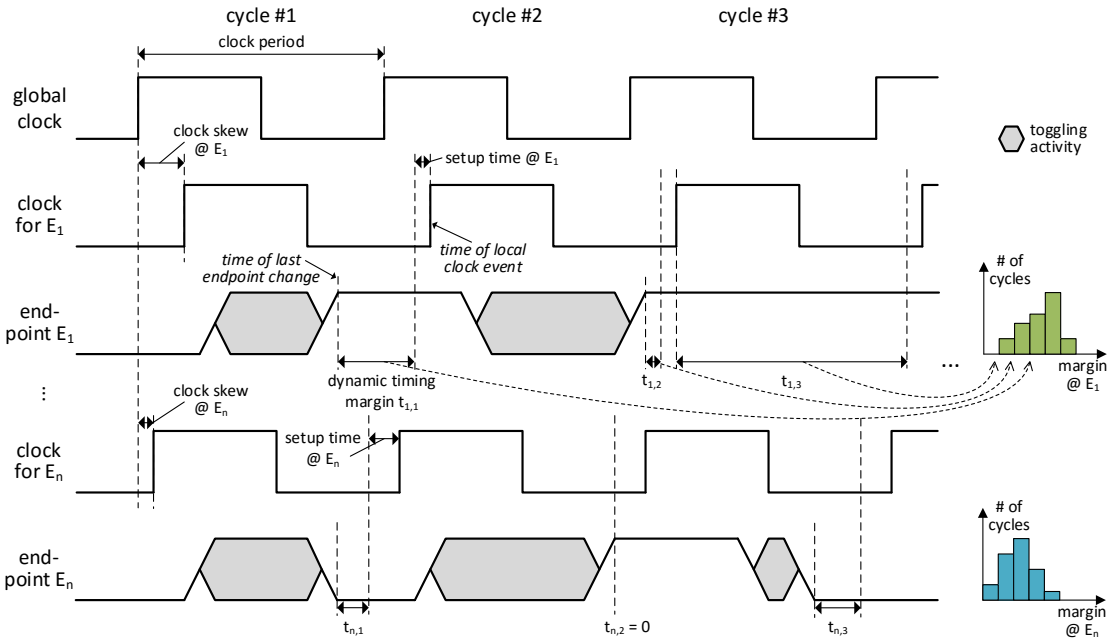


Figure 4.4 – Timing diagram detailing the concept of dynamic timing analysis.

circuit, and for each endpoint E_k the local clock arriving at the sequential element to which the considered endpoint belongs, together with the data input of the endpoint. The local clock is delayed compared to the global clock (at the clock tree root) due to the insertion delay of the clock tree. Due to imbalances in the clock tree causing clock skew between nodes, different endpoints have differences in this delay.[5] When performing DTA it is important to account for this skew, and to not use the rising edges of the global clock as the reference points regarding when cycles end or begin for a specific endpoint. The dynamic timing margin $t_{k,c}$ for an endpoint E_k in cycle c, is derived by capturing the time of the last change of E_k and the time of the next local clock event (i.e., the rising clock edge). $t_{k,c}$ is then calculated by subtracting both the setup time required for E_k and the time of last change of E_k from the time of the next rising clock edge. As is illustrated in the figure, the setup time can vary between endpoints, sometimes significantly, especially for different types of endpoints, e.g., between standard flip-flops and SRAM data inputs. In cycle #2 we can observe that the endpoint E_n has a dynamic timing margin of 0, since the critical path of the circuit leads to E_n and is excited for that cycle. E_1 on the other hand sees a maximum possible dynamic timing margin in cycle #3, since the endpoint does not toggle in that cycle.

As the result of DTA, we construct a histogram for each endpoint E_k, which describes the statistical distribution of the dynamic timing margins for that endpoint. Section 4.3 presents a possible application for such statistical distributions derived by DTA, utilizing detailed endpoint statistics for impact-evaluation of timing errors on application behavior of a micro-

[5] Note that clock skew is often even introduced on purpose as a rebalancing/optimization technique, in the form of so-called *useful skew*.

processor. Details on a specific tool flow, which allows to perform DTA with common EDA tools, is presented in the following Section 4.2, which introduces an extended/specialized version of DTA for microprocessors.

4.2 Dynamic Timing Analysis for Microprocessors

This section enhances the general dynamic timing analysis method presented in Section 4.1.3, to specifically target the analysis of microprocessor circuits. The goal of DTA for microprocessors is to analyze the state-dependent dynamic timing margins in order to uncover correlations between certain states and favorable timings of the processor pipeline. In that regard, one of the most promising state variables[6] to consider are state variables which reflect or are dependent on the instruction types that are processed by all pipeline stages of the processor. The idea is to derive distributions of the dynamic timing margins at all endpoints in the processor pipeline, each additionally depending on the instruction type that is currently residing in the respective stage with which an endpoint is associated.

To illustrate this idea more clearly, Figure 4.5 shows a timing diagram of DTA for a microprocessor pipeline. The employed processor has an arbitrary number of pipeline stages, of which only two subsequent stages (decode and execute) are depicted in the figure for simplicity. For each of the two stages, the timings of two endpoints are shown, together with the respective local clock signals. In this example, three different instructions (addition, multiplication, and no-operation) are used in cycles #1-#4. As a result, for each endpoint there are separate statistics collected per instruction type, i.e., the figure shows for each endpoint a set of three histograms. As an example for differences in the timing distributions, we can see how the multiplication instruction causes the longest delays, i.e., has the smallest timing margins, while residing in the execute stage, where arithmetic operations are performed. On the other hand, the no-operation (NOP) instruction only causes extremely short delays, i.e., provides large dynamic margins, both in the decode and execute stage. The timing distributions of the addition instruction for the endpoints in the execute stage indicate that larger margins than for the multiplication instruction are available, due to the lower complexity of the arithmetic operation.

While individual timing distributions that are derived on a per endpoint basis are useful (e.g., as they are employed in Section 4.3), analysis of instruction type timings in relation to whole pipeline stages can give a more comprehensive picture. In order to aggregate the timing data according to pipeline stages, we calculate in a cycle c the minimum stage-dependent dynamic timing margin $t_{min}(s,c)$ for a pipeline stage s which comprises k endpoints:

$$t_{min}(s,c) = \min_{i=1...k} t_{s,i,c}, \tag{4.2}$$

where $t_{s,i,c}$ is the dynamic timing margin in cycle c at endpoint i belonging to stage s (notation as depicted in Figure 4.5). The timing distribution for a specific instruction type T and a pipeline stage s is then constructed from the subset of all $t_{min}(s,c)$, where for cycle c, T resides in stage s. With the introduction of instruction-based dynamic clock adjustment in Section 5.1, we will have a closer look at how this stage-dependent analysis view, which aggregates groups of endpoints, can be exploited.

[6] A *state variable* here denotes any subset of the full processor state, e.g., a set of registers.

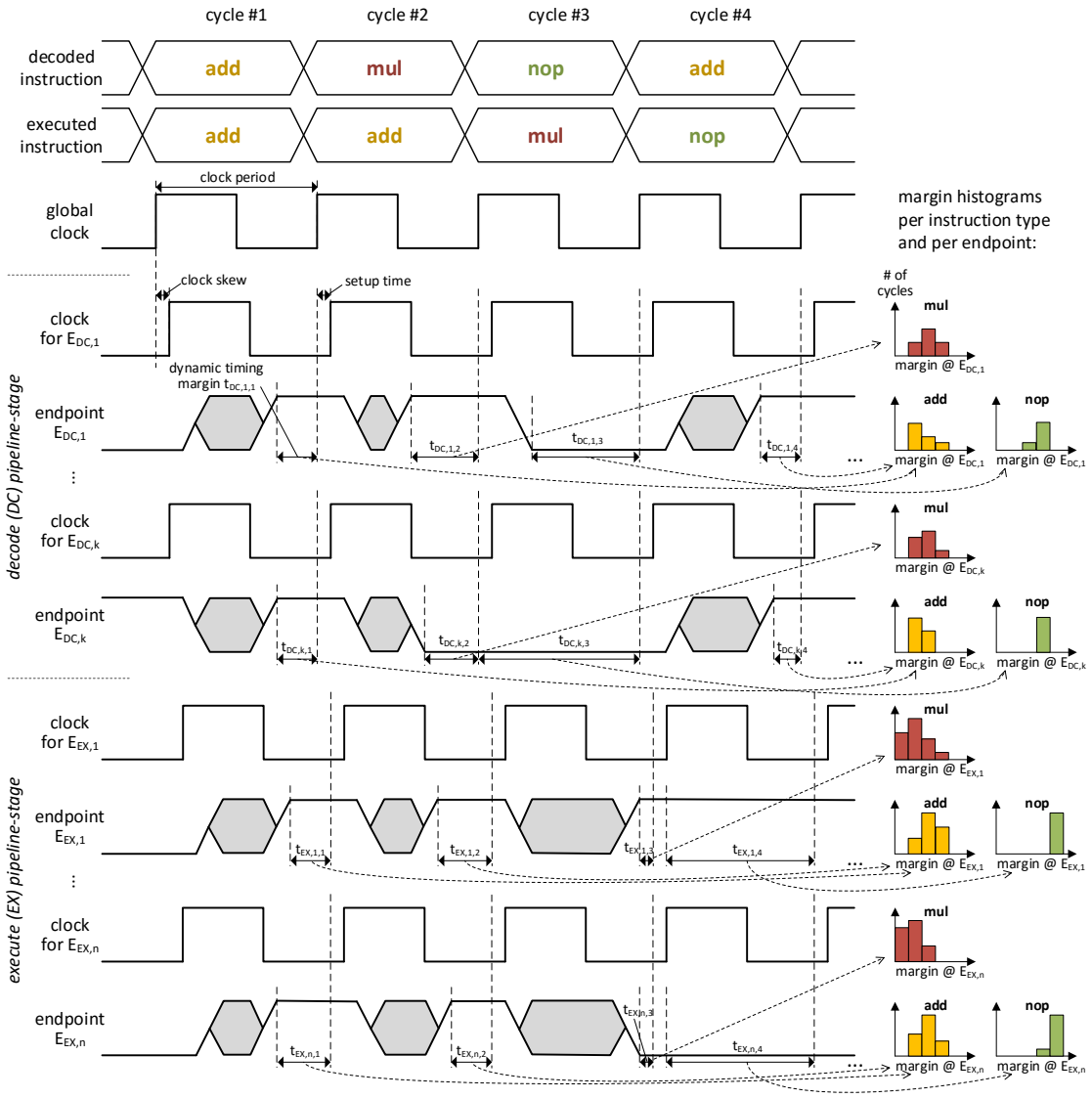


Figure 4.5 – Timing diagram illustrating the application of dynamic timing analysis to a microprocessor pipeline. The different instruction types passing through the pipeline stages cause state-dependent dynamic timing margins of particular distributions at the different endpoints of the pipeline.

In Figure 4.6 we present our tool flow using a mix of commercial EDA tools and custom developed scripts, to perform DTA for a microprocessor architecture. The flow comprises a hardware implementation part (depicted on the left-hand side of the figure), and a characterization part including DTA (right-hand side). On the implementation side, the processor description is synthesized with Synopsys Design Compiler and the resulting netlist is then processed by a backend flow using Cadence Encounter, to produce a fully placed and routed netlist, including the clock tree. This netlist is then used with extracted, accurate timing information in SDF form as the basis for the gate-level simulation. The simulation moreover requires

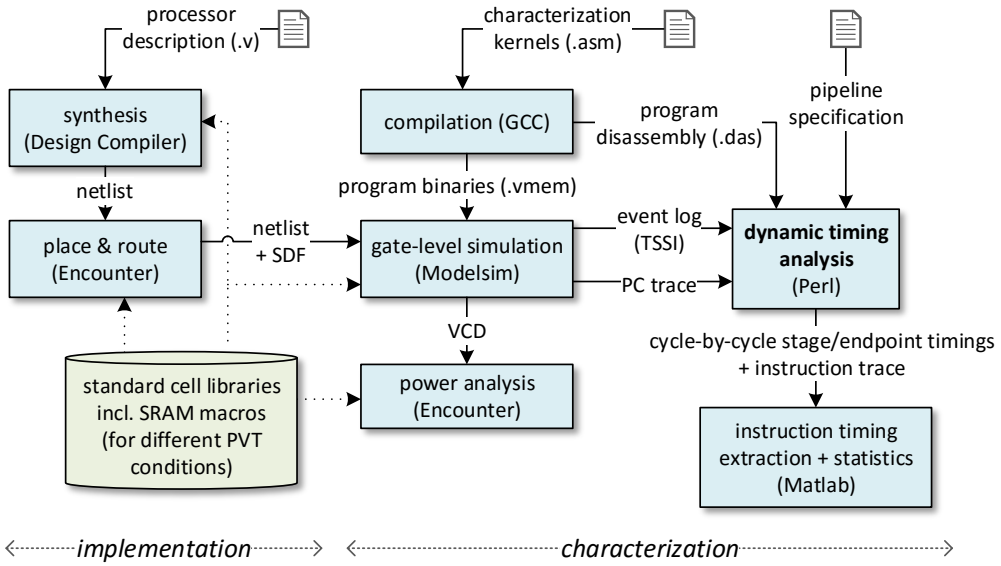


Figure 4.6 – Proposed dynamic timing analysis tool flow for microprocessors.

the program binaries that the processor should run, which are generated via compilation of characterization kernels. These kernels cover in their instruction flows all instruction types that should be analyzed, while employing these instructions with an appropriate (wide) range of different operands. The gate-level simulation is then performed using Mentor Graphics Modelsim, to generate three outputs: First, an event log is created, containing all timestamps of any toggles of endpoints and clock inputs of all sequential elements in the circuit, occurring during execution of the characterization kernels. This log is generated in TSSI[7] format, by exporting a ModelSim list window that is logging all endpoints of the processor (i.e., all data inputs of any flip-flop/latch standard cell type, or any SRAM inputs), as well as all clock inputs corresponding to these endpoints. Second, a trace of the program counter (PC) evolution is logged during the simulation, and also forwarded to the custom dynamic timing analysis tool. Third, a VCD file is recorded and used with the post-layout netlist in Encounter to provide accurate power analysis.

Our custom dynamic timing analysis tool (written in Perl) is then utilized to produce cycle-by-cycle timings of all endpoints and of the predefined pipeline stages. The information regarding the predefined pipeline stages is supplied to the tool via a pipeline specification file, which contains descriptions of the group of endpoints belonging to each pipeline stage. These descriptions are written in the form of regular expressions matching the netlist names of the (standard) cells corresponding to the endpoints of a stage. The cycle-by-cycle slack timings are derived from the event log and stored in a simple format, which provides endpoint- and stage-timings via one dynamic timing margin value per line for each cycle. Moreover,

[7] TSSI is a simple ASCII format for test vectors by Test Systems Strategies, Inc., supported by ModelSim. While by far not as compact as the VCD format, TSSI is a more convenient format for further processing in the dynamic timing analysis tool.

the DTA tool combines the program disassembly with the PC trace to generate a trace of instructions that were executed during the simulation, which then allows to directly associate cycle numbers with instruction types. Finally, the output of the DTA tool is then fed into a set of custom Matlab scripts, which process the cycle-by-cycle slack data and instruction trace to derive the instruction-dependent timing distributions and histograms.

4.3 Characterization of Application Behavior under Timing Errors due to Frequency- and/or Voltage-Over-Scaling

This section proposes a novel approach to modeling of gate level timing errors during high-level instruction set simulation, based on statistical timing information derived by dynamic timing analysis. In contrast to conventional, purely random fault injection, our physically motivated approach directly relates to the underlying circuit structure, hence allowing for a significantly more detailed characterization of application performance under scaled frequency/voltage (including supply noise). The model uses gate level timing statistics extracted by dynamic timing analysis (as presented in Section 4.2) from the post place & route netlist of a general-purpose processor to perform instruction-aware fault injections. The goal is to employ statistical fault injection to provide a more accurate and realistic analysis of power versus error performance, especially in the transition region between 100% reliable operation and almost guaranteed complete failure of particular realizations of a die.

As has been discussed in more detail in Section 4.1.1, shrinking transistor sizes and increased static and dynamic parametric variations, as well as the need to reduce pessimistic design margins renders circuits more prone to timing errors, caused for example by supply voltage noise. Such timing errors may permeate various parts of the microarchitecture, propagate to the system software layer and eventually lead to catastrophic program failure. The severe impact of such errors on system functionality has led to new design paradigms and mitigation techniques that either try to predict the potential errors and apply voltage/timing guardbands at design time or try to detect the incurred timing errors at run time and take corrective actions at the microarchitecture level [DTP+09, BTL+11].

To circumvent the large power and performance penalties of such approaches, the approximate computing paradigm has emerged as an alternative, where output-quality is traded off against power by exploiting the error-resilient nature of various applications [CVC+13]. However, the efficiency of this design approach largely depends on the identification of the real impact of timing errors on system operation, which is usually evaluated through models and system simulators at design time. Inaccurate prediction/simulation of the errors at design time may not only lead to the design of inefficient techniques that waste resources and power, but may also lead to complete failure, if the impact of errors is underestimated. Therefore, an essential step in coping with variations and the resulting timing errors is the development of a detailed understanding and accurate characterization approaches that consider the statistical nature and real impact of such errors at the microarchitecture level.

Several high-level timing models and simulators exist for injecting errors and studying their impact on system performance [STB97, SIHM06, KGA+09, CMC+13]. Although emulation of a faulty environment at the gate level or on real hardware may be more accurate in capturing the impact of faults, such approaches are not widely used due to the prohibitively long simulation time and high setup cost [CP95]. Instead, fault injection (FI) at the microarchitecture level by flipping register bits in a cycle-accurate simulator or at the software layer by altering

memory states have prevailed, due to the reduced simulation time [NV03]. Unfortunately, this methodology only has a reasonable link to the physical reality for the case where errors are assumed to originate from single-event-upsets (SEUs) [Nor96] due to radiation. However, when considering timing violations and PVT variations, such approaches suffer from low accuracy since errors at various registers are either injected randomly without any view of the actual gate level implementation or timing [HLR+09, dKNS10, WTLP14, RHLA14] making the fault injection rather unrealistic. A compromise between speed and accuracy for better understanding and simulating the impact of timing violations lies in modeling the gate level timing behavior of the underlying circuits carefully in an instruction set simulator.

In the following, we propose a novel approach to model gate level timing errors during high-level instruction set simulation based on accurate characterization of the statistical nature of the timing of an open-source processor. The proposed approach involves a number of key contributions that are described as follows.

- In contrast to conventional, almost purely random fault injection, the proposed approach directly relates to the underlying circuit, since characterization of timing error statistics is performed at gate level on a post place & route netlist.
- The characterization accuracy of timing errors during high-level simulation is improved by conditioning the error statistics on the instruction type using gate level DTA.
- The initially fixed characterization of the DTA for different operating conditions is extended to also model the dynamic impact of (high frequency) supply voltage noise, which is one of the most critical timing uncertainties since it is difficult to compensate for with more conventional process compensation techniques.
- The characterized statistical instruction-based timing behavior of the underlying processor is used to inject faults in a cycle-accurate simulator. This allows accurate evaluation of the impact of faults at the application layer. The impact is quantified in terms of output quality, as well as energy and performance.
- The proposed approach is applied to a 32-bit 6-stage OpenRISC core in 28 nm CMOS and the impact of timing errors on various application kernels with different characteristics (computation, control) in terms of output quality and point of first failure (PoFF), is assessed.

Overall, the proposed approach does not only provide an alternative and more accurate approach for rapid evaluation of the impact of timing errors on system performance, but can also prove as an essential tool to identify and mitigate reliability bottlenecks in hardware implementations (e.g., by pointing out structures that lead to timing walls that cause frequent fatal errors) as well as to determine the timing margins required to achieve a desired quality metric.

The rest of this section is organized as follows. Section 4.3.1 presents the case study used for evaluation of the proposed approach. This includes a description of the employed hardware

processor core, the instruction set simulator with fault injection, and the software benchmarks. Section 4.3.2 then discusses different models for timing errors, including our novel statistical approach. In Section 4.3.3 the statistical fault-injection approach is applied to the case study, providing analysis of the power, performance and output quality results obtained by our proposed model.

4.3.1 Case Study

Before describing in detail our modeling approach, in the following we first present the targeted hardware and software environment, as well as the simulator employed in our case study for evaluation of the performance of various benchmarks under frequency- and/or voltage-over-scaling.

4.3.1.1 Hardware Processor Core

The microprocessor used in our case study is comprised of a modified 32-bit OpenRISC [Ope16, Ope14] general purpose embedded processor core, which includes tightly-coupled instruction and data memories. The microarchitecture of the core has a 6-stage pipeline and achieves close to one instructions per cycle, including single-cycle 32-bit multiplications. Both memories are realized in form of single-cycle latency SRAM macros. For more specifics regarding the microarchitecture of the core and its implementation, the reader is referred to Chapter 5 (specifically Section 5.2.1), which discusses our custom OpenRISC microarchitecture in detail, and presents a fabricated silicon prototype of the core used also for this case study.

The core is implemented with a timing constraint strategy which avoids a timing wall in the path delay distribution of the circuit to ensure that important control paths are not immediately affected by frequency-over-scaling (also see Section 5.1.3.2, for more discussion on this concept). For such a core we find that this smoothing of the timing wall can be achieved with limited area and power overhead[8], enabling a graceful performance degradation beyond the static timing analysis (STA) limit. In our implementation, this optimization ensures that only the ALU endpoints of the execution stage data path limit the maximum clock frequency (here 707 MHz at 0.7 V), while the paths in all other stages are short enough to still be safe even when operating below a certain much higher threshold frequency (here 1.15 GHz at 0.7 V). Hence, for this case study, we can limit our modeling to timing errors that are induced in these 32 ALU-endpoint flip-flops. Furthermore, we assume the use of metastable-hardened flip-flops [LRC+11] for these 32 ALU pipeline registers, and use the fact that non-ALU instructions (e.g., branch, load, store, etc.) are always safe from timing errors (below the given threshold frequency).

For the timing characterization required by our proposed model we use the DTA as presented

[8] Both the area overhead and the increased power consumption due to acceleration of some paths in the circuit are limited to 5–13%, depending on the targeted PVT operating conditions in the employed 28 nm CMOS technology.

in Section 4.2 and apply it to a fully placed and routed final netlist of a test-chip design including the processor, which has been fabricated in a 28 nm FD-SOI CMOS technology, and is presented in detail in Section 5.3 and Section 5.4. Such statistics derived from timing-annotated gate level simulations are as close as possible to data from silicon measurements (in a typical semi-custom digital design flow) and can be extracted for different process corners, supply voltages, or temperatures. Alternatively, a silicon-based characterization can be performed from a prototype, instrumented with error detection flip-flops, but the effort is considerable and limited to the corner of the available samples.

4.3.1.2 Instruction Set Simulator with Fault Injection

Simulation of the processor is performed using a cycle-accurate instruction set simulator (ISS) that is generated from a custom LISA model of our OpenRISC implementation. The ISS is enhanced by the FI framework developed by the authors of [WCC13], which allows the injection of faults on the level of the microarchitecture (e.g., into pipeline registers).

While a benchmark is executed on the ISS, FI is only performed for the algorithmic kernel part of that benchmark (typically accounting for >99% of the runtime cycles), excluding any initialization and setup code around the kernel from FI. This exclusion allows us to focus on the effects of the induced timing errors on the characteristic parts of the code of the application. Furthermore, since FIs can frequently cause wrong branching behavior, we include a basic infinite loop detection in the ISS to abort the execution in case of obvious fatal errors.

In more detail, the developed statistical simulation framework employs the Synopsys Processor Designer tool suite [Syn12] for generation of the cycle-accurate ISS from our custom LISA model, which has been developed from scratch to accurately model the microarchitecture of the employed OpenRISC processor core. The generated ISS source code (in C++) is then linked together with an enhanced version of the FI framework [WCC13], which incorporates the support for FI using endpoint timing error statistics in form of cumulative distribution functions (CDFs), as required by our novel analysis approach (compare Section 4.3.2.4). Moreover, support for the modeling of supply voltage noise effects is added to the FI framework, and all newly added features are optimized and tightly integrated with the existing framework, to minimize the impact on additional runtime requirements for the ISS.

The developed ISS with statistical fault injection supports the following list of run time configuration options[9]:

- Clock period [ps]
- Cycle count limit (used as a timeout criterion, if an endless-loop situation cannot be automatically detected)

[9] Please also refer to the model description in Section 4.3.2.4 for more details and context regarding the ISS configuration options.

- Set of timing error statistics derived via DTA (allows for configuration for any arbitrary operating condition, with different sets of CDFs, to reflect: supply voltage, process corner, temperature, age)
- Reference supply voltage level (interpolation base point) [V]
- Voltage noise level (sigma) [mV]
- Start label or cycle number for FI (to isolate relevant code)
- End label or cycle number for FI (to isolate relevant code)
- Threshold for maximum number of jumps in a row (freeze/crash detection)
- Random number seed for Monte Carlo simulations
- ELF binary of the program to be executed on the ISS

The developed statistical ISS for timing error impact analysis is then used in conjunction with custom developed Perl and shell scripts for distributed Monte Carlo simulation, to allow for an efficient characterization of the probabilistic behavior of applications under timing errors. Our Monte Carlo simulation environment uses the HTCondor software framework [Uni16] at its core for the purpose of job distribution and result collection. Moreover, the developed scripts provide a means to launch different sets of simulations via simplified configuration files, which provide an effective way to evaluate different frequency windows in the interesting region between fully error free operation and complete circuit failure. Windows of specific interest can then be simulated with increased granularity for closer study of how the application behaves during the transition phase. Our scripting environment moreover automates most parts of the post-processing, where the single results of the separate ISS invocations are aggregated and sorted according to the various swept parameters, with minimal manual interaction.

4.3.1.3 Software Benchmarks

We characterize the application performance for a set of four common kernels used in signal processing and other areas, each focusing on different types of operations. As the benchmark properties in Table 4.2 show, some kernels are more computation (data path) heavy, while others are more control oriented. The execution length of a single benchmark is in the range of 60 k to 1 M cycles, depending on the kernel. This shows that a high-level ISS is required to be able to assess the application behavior effectively, since a gate-level simulation with fault injection is clearly not feasible. This is especially true, when considering that due to the statistical nature of the model, a large amount of Monte Carlo trials is required. The performance metrics reported in this study are assessed by Monte Carlo simulation with at least 100 simulations per parameter configuration (data point).

In the following, we briefly introduce the characteristics and algorithm parameters of the four considered benchmarks.

Table 4.2 – Overview of benchmark properties.

bench-mark	**median**	**matrix mult. (8- & 16-bit)**	**k-means clustering**	**Dijkstra**
type	sorting	arithmetic	data mining	graph search
compute	-	++	+	-
control	+	-	+	++
size	129 values	16x16 matr.	8 points (2D)	10 nodes
cycles	216 k	60 k	351 k	984 k
output error	relative difference	mean squared error (MSE)	cluster membership	mismatch in min. distance

Median The median benchmark sorts a vector of 129 uniformly random 16-bit unsigned numbers, and then selects the middle value of the vector as its output. The data operations only involve movement of data, while the control flow is based on value comparisons (effectively subtractions). The output error is reported relative to the correct median value, where the absolute value of the difference is used to obtain a positive ratio. The length of the benchmark is 216 k cycles. The kernel is chosen as the illustrative benchmark in this study, since it can be significantly frequency-over-scaled and shows interesting behavior under all parameter configurations (supply voltages and noise levels). It should be noted however that any of the other benchmarks can equally be used to demonstrate the significant improvements in characterization detail between the different timing error models discussed in Section 4.3.2.

Matrix Multiplication The matrix multiplication benchmark performs a multiplication of two 16×16 matrices of signed numbers, and is implemented in two versions: 8-bit and 16-bit. The bit-width concerns the input value range, as well as the normalization that is performed on the result matrix entries. The kernel is mainly dominated by data operations, which consist of multiplications and additions, while the control flow is not based on the computed data. The output error is reported in form of the mean squared error (MSE) of the result matrix. For both versions, the length of the benchmark is 60 k cycles.

K-Means Clustering A popular kernel of data mining applications is k-means clustering. This benchmark groups a number of n-dimensional points (objects) into clusters. The data operations include multiplications and additions (Euclidean distance), while the control flow is heavily based on the computed data, to converge to a clustering solution. To allow for a feasible runtime of our experiments, we perform clustering on 8 objects with n=2 dimensions into 4 separate clusters. The output error is reported as the percentage of points which differ in their cluster memberships, compared to the fault-free reference run. Due to the heuristic nature of the algorithm, we have to account for structural equivalence, i.e. the possibility of resulting clusters to have identical structure, while having different internal naming (cluster ids), and do not consider these cases as output errors. The length of the benchmark is 351 k cycles.

Dijkstra One of the most common graph-algorithms is Dijkstra's algorithm, which finds the shortest path between two nodes in a graph. The benchmark computes the minimum distance between all possible pairs of nodes in a connected graph of 10 nodes. The data operations consist mainly of simple additions (path length sums), while control flow is based on comparison of the computed data (maintenance of sorted node list). The output error is reported as the percentage of node pairs, which have any mismatch in their calculated minimum distance. The length of the benchmark is 984 k cycles.

4.3.2 Modeling of Timing Errors

Modeling of timing errors in high-level instruction set simulators can be performed with different levels of detail to match the underlying circuit-level implementation.

Table 4.3 provides an overview of different modeling approaches, and their features. We begin our analysis with the widespread purely random fault injection model A, to show its limitations. Next, model B follows more closely the actual circuit behavior by considering the results of the static timing analysis of the circuit under a given set of operating conditions. We then further refine this model by considering the impact of supply voltage noise on the timing behavior and we refer to this refined model B as model B+. Finally we introduce our novel model C, which further improves the accuracy of the characterization of application behavior with an even more detailed fault injection that accounts also for critical path activation statistics conditioned on individual instruction types.

As a general note regarding the term *fault injection*: In the context of this work, we define a fault injection by the injection of a random value (i.e., a 50% bit-flip probability), since a timing error typically means that the new value is sampled during the settling phase of the data (D) input of a flip-flop, and hence can be 0 or 1 with 50/50 probability.

4.3.2.1 Fixed Probability FI

Model A is based on the introduction of random bit flips into all or a limited subset of logical or physical registers within the processor core (and potentially the memories). This approach

Table 4.3 – Overview of timing error models and their features.

model	*fault injection technique*	*timing data*	*multi-V_{dd}*	*V_{dd} noise*	*gate-level aware*	*instruction aware*
A	fixed probability	none	no	no	no	no
B	fixed period violation	STA	yes	no	partially	no
B+	modulated period violation	STA	yes	yes	partially	no
C	probabilistic period violation (using CDFs)	DTA	yes	yes	yes	yes

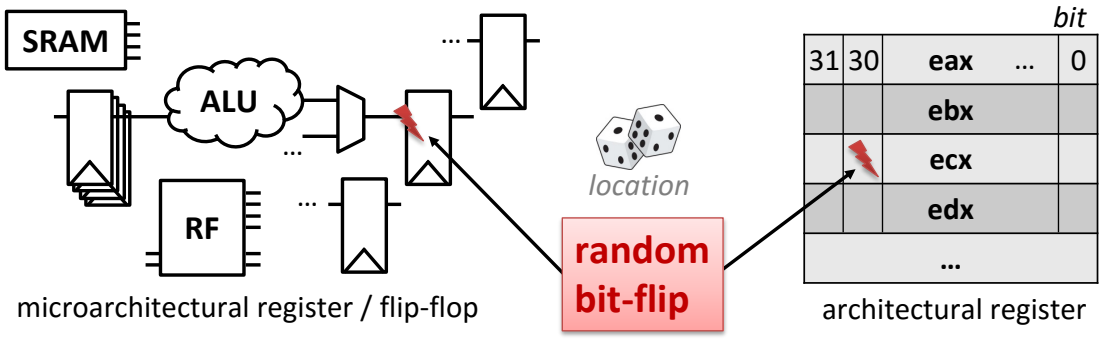


Figure 4.7 – Random fault injection in microarchitectural or architectural registers, based on fixed probability FI (Model A).

of random FI is illustrated in Figure 4.7. Each bit flip occurs with a fixed FI probability. This simple model has originally been motivated by the analysis of SEUs, for example caused by particle radiation, which affect all resources independently of each other and independent of the processor state or timing properties, but even for that it has been shown that accurate modeling of the underlying hardware is essential [CMC+13].

Nevertheless, this random FI is also frequently used to model the impact of variations which actually manifest in the form of timing errors. Unfortunately, this straightforward approach is obviously highly inaccurate and lacks any physical motivation: The model neglects the fact that timing errors appear selectively only on the endpoints of critical or near-critical paths and only if these paths are actually excited, however then with rather high probability. Moreover, the FI rate has no direct link to the activity of the hardware or to the operating conditions. This is especially problematic, since we aim to characterize the impact of frequency- and/or voltage-over-scaling and supply voltage noise on the program behavior and application output error.

Furthermore, it is important to note that architectural registers which are defined in the ISA of a processor (as depicted on the right-hand side of Figure 4.7), do very often not correspond to one specific single physical register in the microarchitecture. This can be for example due to the use of shadow register banks, or register renaming mechanisms employed in the processor pipeline, which makes the random modification of single bits of architectural registers not appropriate for the study of timing error induced failures.

4.3.2.2 Static Timing Based FI

To relate the FI for individual endpoints within the processor core more closely to the underlying hardware, [WCC13] proposes to consider the worst case path delays to each endpoint. These delays can be obtained for any PVT operating condition that is available from the design kit through STA of the gate level netlist of the placed and routed design. The individual endpoints are then related to their location in the execution pipeline of the processor and

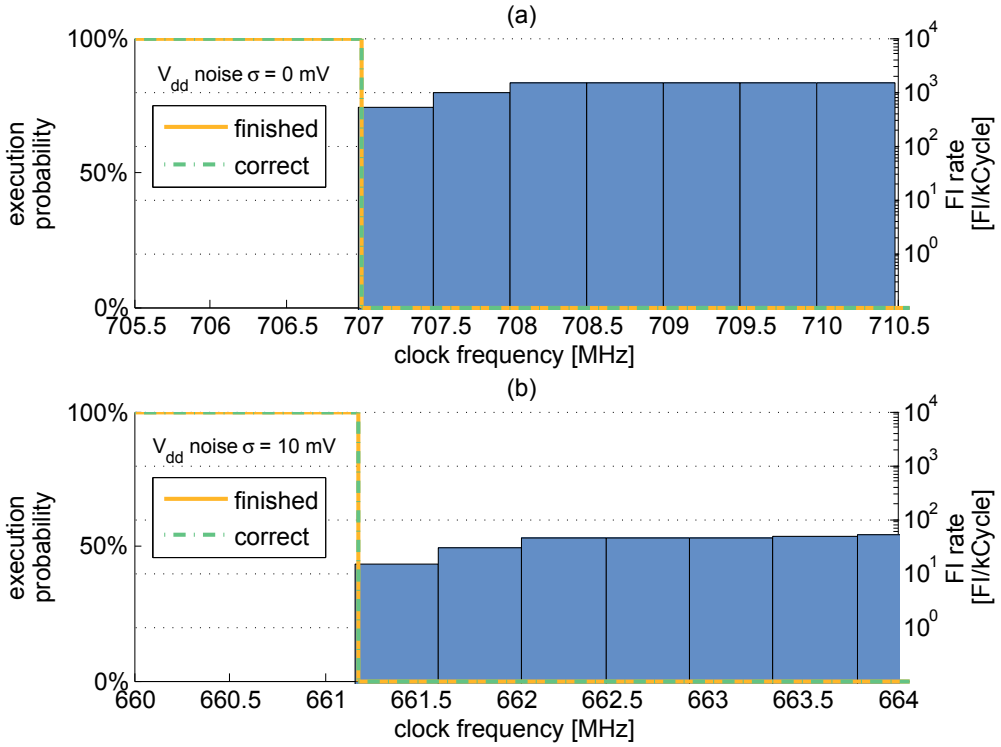


Figure 4.8 – Execution probability for the median benchmark application to finish (with and without correct result) and fault injection rate of the median benchmark versus clock frequency of the processor core, for (a) model B based on STA @ 0.7 V, and (b) model B+ with supply voltage noise (σ = 10 mV).

to the set of instructions that has an effect on this pipeline stage. A fault is always injected into a register whenever the pipeline stage is activated (e.g., only ALU instructions can trigger the FI in the execution stage which shows no activity for other instructions) and when the longest path delay to that endpoint exceeds the clock period. Concerning our case study, each of the 32 ALU endpoints under model B separately experiences a fault injection, when an ALU instruction is executed and the STA-derived clock period limit for this particular endpoint is violated by the clock frequency set in the ISS.

The issue with model B is that it is overly pessimistic, since details of the CPU state and current or previous data values which have an influence on the path excitation are not taken into account. Furthermore, no distinction is made between potentially very different path delays to the same register, depending on the type of instructions that all trigger the same pipeline stage. Finally, factors such as high-frequency supply voltage noise are not considered. Nevertheless, such factors critically determine the behavior of a circuit on the boundary between 100% reliable operation and complete failure and are therefore essential.

To illustrate the pessimism of this model, we apply it to our case study and show the FI rates and program behavior for the median benchmark with different frequencies in Figure 4.8a. It

can be seen from the plot that the FI rate immediately rises significantly as soon as the clock frequency just slightly exceeds the static timing limit, since any executed ALU instruction, independent of its type, leads to timing errors in the execution stage. As a result of this high FI rate, the probability for a program to execute correctly and even to finish drops abruptly from 100% down to 0% with almost no transition region that could be exploited. Repeated execution of the same program will not change this behavior since for a given program, there is no randomness in the model. Note that we limit the illustration here to the results of only one benchmark, since we observe the exact same behavior also with all other benchmarks.

4.3.2.3 Supply Voltage Noise

To recover the link to uncertainties (randomness) in the underlying circuit behavior, we extend model B to model B+ by accounting for the influence of supply voltage noise as a primary source of dynamic physical variation of gate delays and timing behavior of a specific instance of the chip. Supply voltage noise can have many sources: it is inherently caused by DC-DC converters and depends strongly on the off-chip and on-chip power delivery network and the circuit switching activity (V_{dd}-droop). In this study, we model this supply voltage noise by a normal distribution, with a mean of 0 V and a standard deviation σ, but other distributions are also possible. The maximum noise level is clipped/saturated at 2σ, to avoid the occurrence of large, physically unrealistic, spikes due to the tails of the distribution before clipping.

During each cycle in the simulation a new independent random value for the supply voltage noise is drawn and translated into a factor which modulates the timing of each path of the circuit for that cycle. The relation between a small supply voltage change and the corresponding effect on delay is extracted from a fitted V_{dd}-delay curve, which is interpolated from the delay of the worst path characterized at 5 different supply voltages (0.6 V to 1.0 V in 100 mV steps).[10] These modified delays are then used to determine the injection of faults in the same way as for model B (Section 4.3.2.2).

The FI behavior under model B+ is shown for σ = 10 mV (maximum V_{dd} noise of ±20 mV) in Figure 4.8b. Compared to the no-noise scenario of model B (Figure 4.8a), the clock frequency at which first faults start to get injected is now significantly lower. The higher the noise σ, the further away from the static timing limit of 707 MHz lies the first point of fault injection (at 661 MHz and 588 MHz for σ = 10 mV and σ = 25 mV, respectively). However, the observed fault injection rate at the first point of fault injection is significantly lower at only around 10 faults per 1000 cycles compared to model B, due to the modulated (instead of fixed) path delays, caused by the random characteristic of the supply voltage noise.

Unfortunately, we still observe the same hard threshold in the application behavior as for model B (for the shown and all other benchmarks). The reason for this behavior is that even

[10] Although not all paths scale equally in terms of delay (especially over a wide voltage range), due to varying gate compositions of the paths, this is nevertheless a valid approximation for capturing small delay changes around an accurately characterized operating point.

model B+ does not capture the significant instruction and data dependencies of path delays.

4.3.2.4 Proposed Dynamic Timing Statistical FI

To improve the accuracy and link to the physical circuit of models B and B+, we further refine the resolution of the fault injection with respect to different instructions and introduce a better statistical model C to cope with data dependent and microarchitectural or circuit implementation related delay uncertainties.

To this end, we employ dynamic timing analysis to extract the statistics of the data arrival times (i.e., the dynamic timing slack) on all relevant endpoints inside the processor, as introduced in Section 4.2. This characterization is performed independently for different instructions, even if they affect the same pipeline stage. For our experiments, we use a gate level characterization kernel (here with 8 kCycles), covering all ALU instructions with randomized operands. We further verify that all non-ALU instructions have a sufficient timing margin to always be non-critical.

The extracted dynamic timing margin statistics are then used to determine the probabilities $P_{E,V,I}(f)$ of an endpoint E to have a timing error at a given frequency f with supply voltage V, while the instruction I is executed (resides in the pipeline stage associated with E). We calculate

$$P_{E,V,I}(f) = v_f / n_I \tag{4.3}$$

where n_I is the total number of cycles in which DTA encounters instruction I, and v_f is the number of these cycles for which the dynamic path delay to E (including the setup time) is larger than the clock period $1/f$, i.e., cycles in which the endpoint timing is violated by instruction I. Sweeping f provides us with the cumulative distribution functions (CDFs) for the dynamic timing error probabilities, as shown in Figure 4.9 for two instructions, two endpoints, and two supply voltages.

As can be seen from Figure 4.9, the multiplication instruction starts to fail at a lower frequency than the less complex addition instruction for the same supply voltage and ALU endpoint. Moreover, we observe that bits with higher significance tend to fail earlier than bits with lower significance and a higher supply voltage shifts the CDF to the right. However, Figure 4.9 also shows that for example the relative effect of supply voltage scaling compared to the effect of the bit position on the timing error probability can be significantly different, depending on the instruction type. This can be seen from the plots by observing how in Figure 4.9a the CDFs for bit position 3 and 24 at equal voltage exhibit a frequency gap of ≈400 MHz, while in Figure 4.9b they are only ≈100-200 MHz apart.

As illustrated in Figure 4.10, the proposed model C integrates the instruction-aware statistical dynamic timing information from the DTA in form of CDFs, and combines it with the supply voltage noise model presented in Section 4.3.2.3. Specifically, model C performs the following

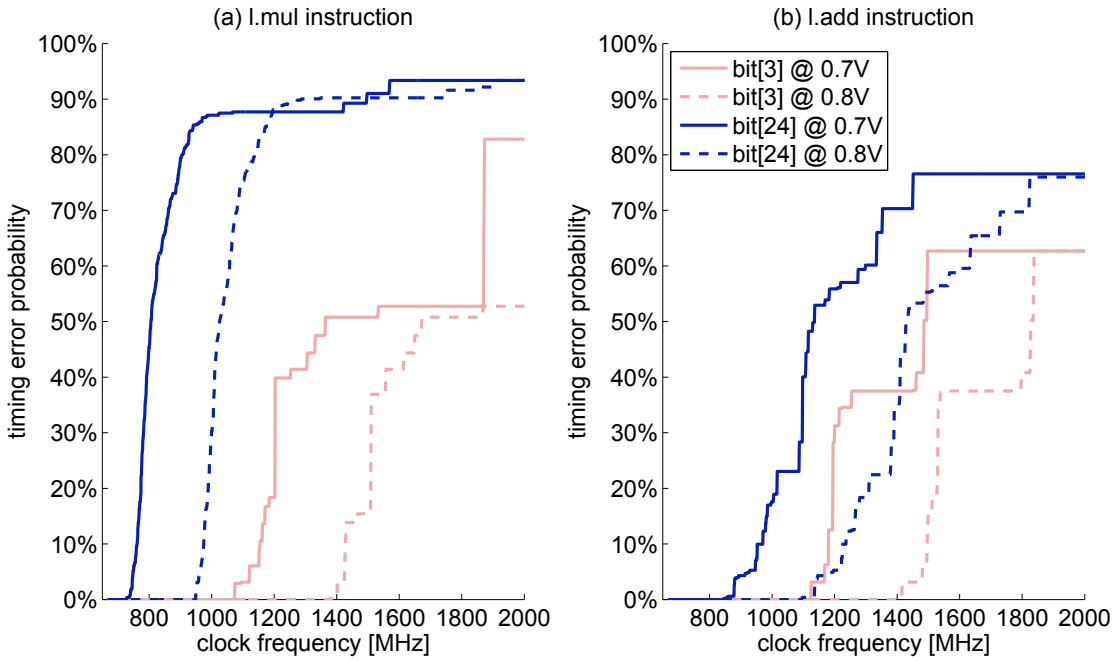


Figure 4.9 – Cumulative distribution functions of timing error probabilities extracted by DTA, for different ALU endpoints (bit 3 (lower significance) & bit 24 (higher significance)) and supply voltages, for (a) the signed multiplication instruction (l.mul) and (b) the addition instruction (l.add).

steps in each cycle of the simulation:

1. A CDF scaling-factor is derived from the defined simulator clock frequency together with the randomly distributed supply voltage noise, which allows for the dynamic adjustment of the CDFs.

2. The timing error probability $P_{E,V,I}(f)$ is determined for all of the relevant endpoints E, by evaluating at f the corresponding scaled CDF with matching instruction I, and supply voltage V (without noise).

3. FI is performed on each endpoint E with the respective probability $P_{E,V,I}(f)$.

Our proposed approach is also able to account for parameters that are constant or vary only slowly for a specific die, such as process variations, temperature, and aging. These effects can be modeled accurately by performing DTA on a netlist that is timed with libraries provided by the foundry that are characterized for the desired process corner, temperature, and age. Different sets of CDFs can then be used within the simulation environment to model the effects on the application.

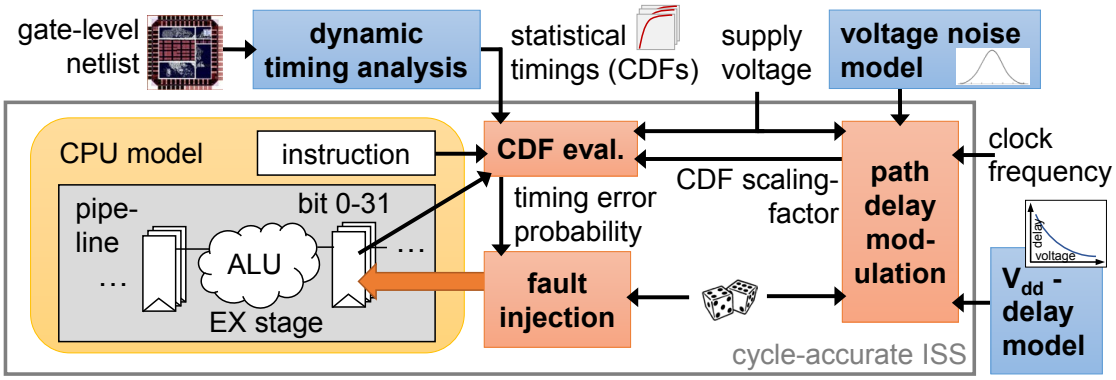


Figure 4.10 – High-level instruction set simulation with statistical fault injection (model C), incorporating effects of supply voltage noise.

4.3.3 Application of Statistical FI

In the following we apply the proposed statistical FI (model C) to the considered case study to illustrate how this detailed model can provide interesting insights into the operation and application behavior under operating conditions that may lead to timing errors.

4.3.3.1 Instruction Characterization

We first study the behavior of addition (l.add) and signed multiplication (l.mul) instructions across different frequencies. Addition is evaluated in two forms, using input operands with a 16-bit value range and a 16-bit result, and with 32-bit operands giving a 32-bit result. Multiplication is performed with input operands that cover a 16-bit value range and a 32-bit result. All operands are chosen uniformly random and the operating point for the error analysis is a supply voltage of 0.7 V with σ = 10 mV voltage noise. The mean squared error (MSE) due to timing errors versus the clock frequency is shown in Figure 4.11.

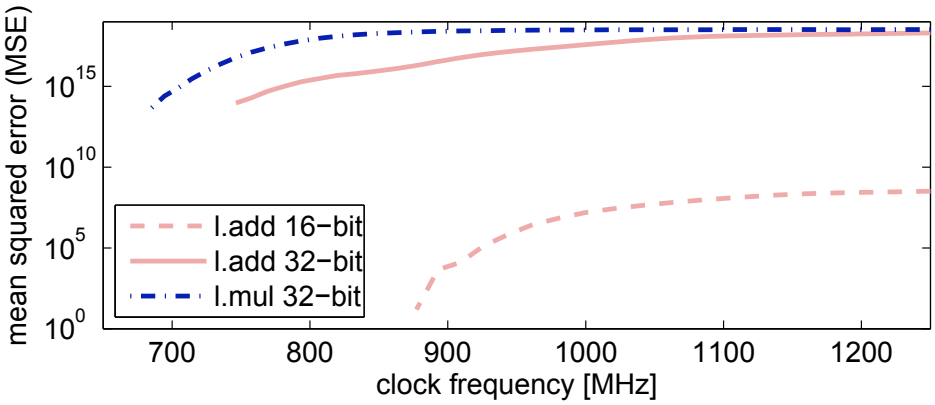


Figure 4.11 – Mean squared error vs. clock frequency for addition (with 16-bit and 32-bit operands) and multiplication instructions at V_{dd} = 0.7 V with σ = 10 mV (model C).

The plot shows that first calculation errors (MSE > 0) occur at 877 MHz, 746 MHz, and 685 MHz for the 16-bit addition, 32-bit addition, and multiplication, respectively. This spread illustrates the relatively large difference between the points of first failure (PoFF) for different arithmetic instructions and also highlights the importance of timing error modeling on single-bit granularity, which can be seen by the significant difference in PoFF when comparing 16-bit with 32-bit addition. Moreover, we observe that the MSE has moderate magnitude for low frequencies and saturates close to maximum values (corresponding to the used operand bit-widths) after about 15% of further frequency increase beyond the PoFF.

4.3.3.2 Impact of Frequency, Voltage, and Noise

A key feature of the proposed statistical fault injection model is that it captures many details of the gate level implementation that are required to study the transition region in which timing errors start to appear due to frequency- and/or voltage-over-scaling or insufficient margins to protect against supply noise. On the application level, the impact of these effects is often characterized by four different metrics:

- the probability for the program/application to finish execution;
- the probability for the execution to be correct, i.e., to provide a fully correct result;
- the rate of injected faults (in FIs per 1000 cycles of kernel execution);
- and the error of the program output, which corresponds to the output quality of the application.

Figure 4.12 shows these metrics for the median benchmark running at different operating frequencies on the hardware, with two different supply voltages and three different levels of voltage noise (sub-figures a-f), averaged over 200 Monte-Carlo trials per data point. The graphs only show the interesting transition regions between reliable and unreliable operation, while the low-frequency ranges where no errors occur (i.e., where no faults are injected) are grayed out and marked with *n/a*. The high-frequency ranges where it is not possible anymore for any program run to finish execution, are similarly left out and marked with *n/a* as well.

A first interesting observation (without considering voltage noise) is that the simulations reveal that the PoFF where the application first does not finish with a 100% correct result is displaced from the pessimistic STA limit. The corresponding possible gain in operating speed from frequency-over-scaling is indicated in the sub-figures. Note that it is possible that below the PoFF (in terms of clock frequency) faults might already injected with a very low rate, if the application is able to tolerate these errors and hence still produce a fully correct result.

A second observation relates to the impact of voltage noise. From Figure 4.12a-c/d-f we can clearly see the impact of the amount of noise on the transition ranges, with higher noise causing a shift for all four metrics down to lower frequencies. This can also clearly be seen by

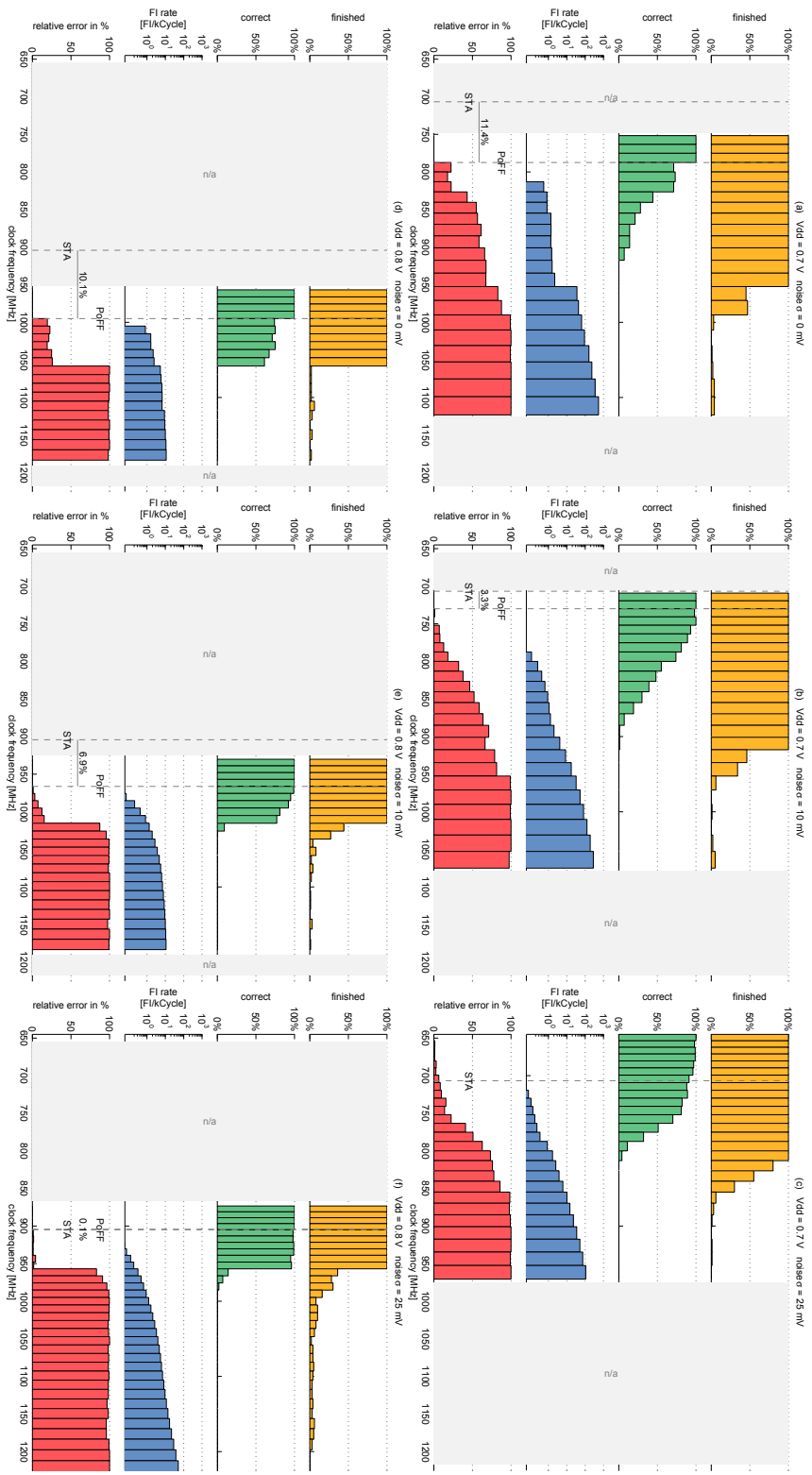


Figure 4.12 – Program performance depending on clock frequency, in terms of finish and correctness probabilities of the program, fault injection rate, and output error, for the median benchmark for different levels of V_{dd} and V_{dd}-noise (model C).

the impact of the voltage noise on the gain over the STA limit at the PoFF, which already disappears at a noise level of σ = 25 mV. It can furthermore be noted that increased voltage noise causes the transition regions to be smoothened out, especially for the output error metric, mainly caused by the more gradual increase in the FI rate.

As can be seen in all configurations, as soon as the probability of the program to finish reaches low values, the output error of the remaining successful runs quickly saturates.

Looking at the effect of supply voltage, one can observe that a higher supply voltage results in sharper changes in the transition regions, which also means that the application error explodes more rapidly after the PoFF. Hence our analysis indicates that lower supply voltage seems to favor a gradual failure behavior, which is often desired in approximate computing applications.

It has to be noted that the presented trends regarding impact analysis of supply voltage and noise on application performance cannot easily be generalized, and can highly depend on the specific kernels employed by the applications, hence requiring dedicated simulation and analysis using our proposed statistical approach.

4.3.3.3 Performance Comparison of Benchmarks

A key aspect of our model C is the distinction between different operations. This distinction is especially important when considering different types of kernels which rely on different instruction types and sequences. We demonstrate how this behavior is exposed by comparing different benchmarks in Figure 4.13 at an operating point of 0.7 V with a supply noise of σ = 10 mV. For comparison, the corresponding sub-figure with equal parameters for the median benchmark is Figure 4.12b. Please note that due to the voltage noise level being set at non-zero, errors can occur already at and below the frequency determined by STA.

Comparing 8-bit and 16-bit matrix multiplication (Figure 4.13a & Figure 4.13b), we see very similar application behavior. The lower bit-width helps the 8-bit benchmark in the region below the STA frequency to have a significantly higher rate of fully correct benchmark executions where timing errors are still induced due to the supply voltage noise. The MSE develops similarly for both bit-widths, with a factor of about 10^3 between them, due to the different operand and result ranges.

Compared to the matrix multiplication benchmark, the k-means benchmark (Figure 4.13c) experiences a fault injection rate that is almost one order of magnitude lower at the same operating frequency, which can be explained by the significantly lower number of more timing critical multiplications. Nevertheless, the kernel shows a considerable performance degradation (30-40%) of the quality metric, even though the code is still able to finish execution for a similar frequency range above the STA limit.

Finally, the Dijkstra benchmark (Figure 4.13d) is characterized by only a very narrow transition

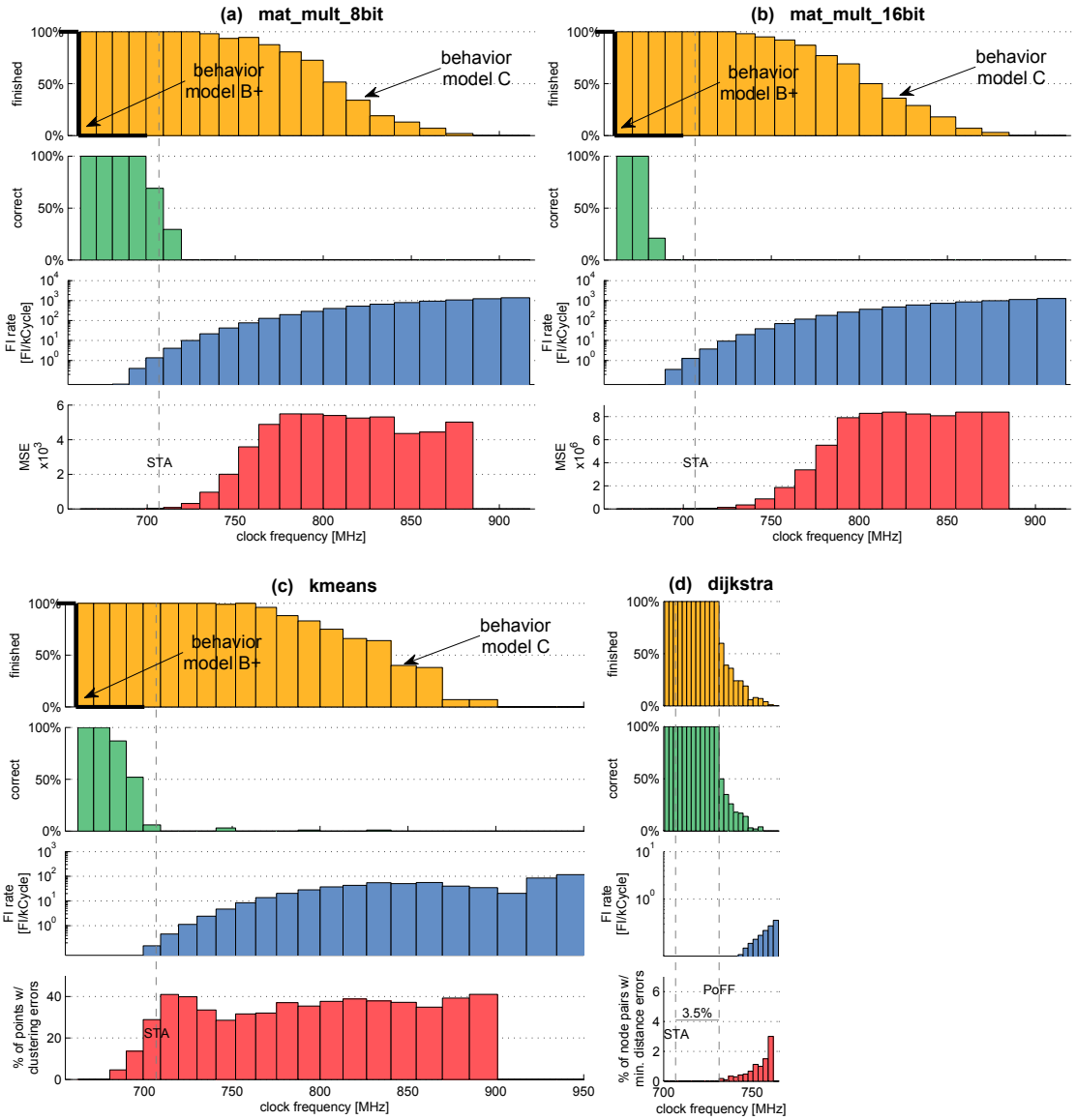


Figure 4.13 – Program performances for the matrix multiplication (8-bit & 16-bit), k-means clustering, and Dijkstra benchmarks, at V_{dd} = 0.7 V with V_{dd}-noise σ = 10 mV (model C).

region (hence it is simulated with a higher resolution). We can observe that although the kernel shows a frequency gain at the PoFF (which the others do not, apart from the median benchmark), 4% of further frequency increase beyond the PoFF already causes the application to fail completely, while still having a very low FI rate (below 1 FI per kCycle).

Figure 4.13 furthermore contrasts the high characterization detail of our proposed model C with the observed behavior under model B+. As can be seen from the plot, the hard failure threshold at 661 MHz (see also Section 4.3.2.3) applies to all benchmarks equally, which means that model B+ does not allow for any of the provided analysis in the relevant transition region.

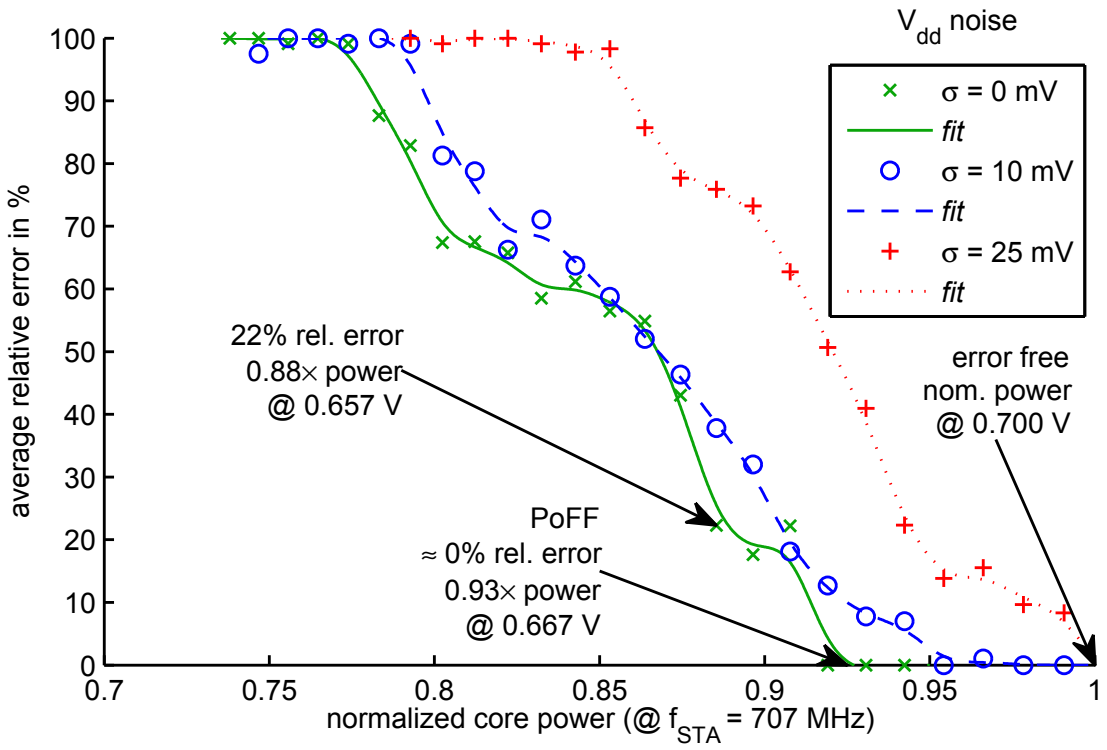


Figure 4.14 – Relative error vs. core power consumption trade-off for the median benchmark, through voltage over-scaling at fixed iso-frequency (nominal: 707 MHz @ 0.7 V) with effect of varying supply voltage noise (model C).

4.3.3.4 Error vs. Power Consumption Trade-Off

An important motivation for avoiding unnecessary voltage margins at the risk of errors or quality degradation are power savings. The proposed simulation model C improves the corresponding trade-off analysis by accounting for the underlying gate level hardware structure.

To illustrate this analysis for the interesting transition region between nominal/safe voltage levels and complete system failure at severely reduced voltage levels and the same nominal frequency, we characterize achievable power savings in relation to the output quality for the median benchmark. The microprocessor operates at a fixed nominal frequency of 707 MHz (the STA limit at 0.7 V). The quality metric is computed by using our simulation model C, while the power savings are obtained by translating potential frequency-over-scaling performance gains into an equivalent reduction of the supply voltage. More specifically, the power savings from this voltage-over-scaling are calculated by quadratic scaling of the consumed active core power between two reference points obtained through VCD-based gate level post layout simulations of the core. The reference points are 10.9 µW/MHz @ 0.6 V and 15.0 µW/MHz @ 0.7 V, while leakage (of the core) only consumes 2% and 3%, respectively.

Figure 4.14 details the trade-off analysis. The bottom right of the figure indicates the nominal

operating point at a normalized core power of 1, and fully error free behavior. At a relative core power of 0.93 (due to a V_{dd} reduction of 33 mV) we reach the PoFF if no supply voltage noise is present. Scaling the voltage further shows a relative power of 0.88 with an average relative output error of 22%.

Considering also the impact of supply voltage noise, we observe that at σ = 10 mV the error-vs-power curve still relatively closely follows the no-noise configuration, with some higher power costs for equal quality. At σ = 25 mV however, we observe a significantly earlier rise in output error, indicating that only marginal power gains are possible for a reasonable output quality.

5 A Microprocessor with Cycle-By-Cycle Dynamic Clock Adjustment

Today, digital synchronous circuits, and in particular microprocessors, all adhere to the basic assumption that the clock period must be set according to a global worst-case timing path. Many innovative techniques on circuit and microarchitectural level that have been developed in recent years [EKD+03, DTP+09, BTK+09, TBW+09] to counter PVT variation effects[1], especially dynamic variations, still adhere to this basic assumption on a fundamental level. While all these approaches apply dynamic methods (i.e., DVFS) to adapt their operation to the varying dynamic circuit conditions, from a cycle-by-cycle perspective the circuit is still operating in a worst-case mode. This is because clock frequency and supply voltage are calibrated such that an overall worst-case path delay (under current temperature and voltage conditions) is not exceeded with very high probability. In other words, a microprocessor under the Razor and other similar approaches typically operates only slightly beyond or at the point of first failure (PoFF) dictated by the (current) critical path of the circuit, in order to keep architectural replay (or other correction) costs due to timing errors down, and to maximize energy efficiency while keeping the impact on throughput minimal. Consequently, the clock frequency and supply voltage of the processor are set such that for any sequence of instructions that the processor executes, timing errors only occur extremely rarely, e.g., with a rate of $\approx 10^{-5}$ at the suggested operating point (PoFF) for a RazorII prototype design [DTP+09]. Timing errors hence only occur in such an operating scenario with a tuned but fixed clock period, when a critical or near-critical path is excited in rare unfavorable conditions due to dynamic variations and consequently violates the applied period. However, as has been shown in Chapter 4, different instructions can exhibit significantly different state-dependent timing margins in the same stage of a processor pipeline since in many cycles critical or near-critical paths are not excited. This leaves a large potential for unexploited timing margins on the table, even for DVFS-capable processors with error detection and correction.

In this chapter we show that by applying a nonconventional synthesis strategy to a standard microprocessor core, the longest relevant path for a given pipeline state can vary significantly, depending on the executed instruction types. Therefore, by departing from the conventional

[1] As has been discussed in some more detail in Section 4.1.1.

constant frequency paradigm and by adjusting the clock frequency according to the current pipeline state, the corresponding dynamic timing margins can be reduced and average throughput and/or energy-efficiency can be improved.

The rest of the chapter is organized as follows: Section 5.1 first discusses related work, and then formally introduces our instruction-based dynamic clock adjustment (DCA) approach. We then present a design flow and a characterization & evaluation environment for DCA-enabled microprocessor design. In Section 5.2 the DCA approach is then evaluated in a case study using instruction set simulation for an OpenRISC core. To this end, a custom OpenRISC microarchitecture is presented, as well as its characterization results in terms of the DTA of the implemented processor, and a simulation based performance and power analysis demonstrates the potential gains achievable by DCA. In order to show the feasibility of the DCA idea beyond a simulation environment, we then target the implementation of a silicon prototype utilizing DCA. To this end, we extend our OpenRISC microarchitecture in Section 5.3 by the necessary hardware modules to enable DCA on a real microprocessor IC, named *DynOR*. The section details the full chip architecture of DynOR, including the clock generation unit that enables cycle-by-cycle dynamic clock adjustment. Section 5.4 discusses the implementation details of the test chip in 28 nm FD-SOI CMOS, and presents the employed measurement setup. Finally, in Section 5.5 detailed measurement and characterization results of the fabricated DynOR samples are provided, regarding the achieved speed improvements and energy savings.

5.1 Instruction-Based Dynamic Clock Adjustment

The idea of instruction-based dynamic clock adjustment is to clock a microprocessor circuit with a cycle-by-cycle varying clock period, which is derived for each cycle from the instruction type(s) that are in flight in the processor pipeline. The goal of this technique is to leverage the differences in timing requirements for the different instruction types, in order to reduce dynamic timing margins and thereby increase instruction throughput beyond what is feasible with a constant worst-case clock period.

5.1.1 Related Work

A SIMD linear array processor with dynamic frequency clocking has been proposed in [RVB98]. This application-specific array processor for image processing applications is constructed from 8-bit processing elements (PEs), chained in a 1D array that is operated in a SIMD fashion. Each PE comprises the functionality of a simple 8-bit ALU (adder, multiplier, logic, etc.) together with a small scratch pad memory (SRAM). The PEs are then clocked according to the operation/instruction that they execute. The generation of the different clock periods is very coarse grained, and large timing margins are given up to achieve a simple clocking scheme which consists of four clock periods, where each period is always doubled in length (frequencies: 50, 100, 200, and 400 MHz). Results are simulation-based for a design in a 1.0 µm

CMOS technology. In [KRV99] a more generalized approach to datapath synthesis using dynamic frequency clocking and multiple voltages is discussed. However, the proposed/assumed processor model is still an array processor architecture composed of many small non-pipelined functional units working in parallel. Moreover, results are purely model-based and no hardware implementations showing the feasibility of the approach are provided.

For field-programmable gate arrays (FPGAs), work on processor systems operated by a dynamic clock can be found in the literature. [BES06] proposes a variable speed processor with a variable period clock synthesizer based on period multipliers. A prototype on an Altera Stratix FPGA is demonstrated, targeted at a basic configuration of the Nios soft-core (no pipeline stall avoidance, no forwarding, and no caches implemented). The clock speed adjustment in the prototype is limited to two distinct levels: fast instructions (147 MHz) and slow instructions (110 MHz). The authors additionally showed post-synthesis results for an implementation of the clock synthesizer (supporting more frequencies) in a 180 nm CMOS technology, without considering any backend related timing effects, which are however crucial, e.g., for accurate assessment of the clock distribution.

More recently, [PCC13] presented an FPGA based SIMD vector processor architecture with 128 PEs, employing 2 effective instruction classes (multiply and non-multiply) to dynamically adjust the clock frequency (to 117 Mhz and 160 MHz, respectively). This work is further extended in [GPC15] by refining the dynamic clock generation on the FPGA via a hybrid approach employing both clock stretching and clock multiplexing.

A general, related dynamic clocking approach is presented in [GMKR10], where adaptive clocking using occasional two cycle operations is applied to arithmetic units (i.e., not a full processor). The approach differs from instruction-based dynamic clock adjustment, since the two cycle clock stretching (effectively implemented with clock gating) is activated by a pre-decoder for critical path excitation detection, looking at the incoming operand data.

A method providing application-adaptive guardbanding to mitigate static and dynamic variability based on a switchable clock is proposed in [RBG14]. The instruction window/history of the first three pipeline stages (before the critical execution stage) of a LEON3 processor is monitored, discerning instruction sequences of two different instruction classes (ALU and non-ALU), derived through a systematic design-time characterization method. At run-time the clock frequency of the core is then switched between two different values corresponding to the two classes, to guarantee correctness when operating in non-probabilistic/intolerant application mode. Voltage and temperature effects are additionally considered for correct adjustment of the clock, as in other CPM-based systems with DVFS. The presented timing results are based on accurate post place & route data of an implementation in 45 nm CMOS of the LEON3 core including the dynamic period lookup table, however excluding the also required circuitry for the clock generation, which enables the adaptive clocking approach. Besides potential issues regarding the immediate adjustment of the clock period, the integration of the clock generation with the employed LEON3 core microarchitecture, especially regarding

timing closure of the clock-adjustment control loop, are not further addressed.

5.1.2 DCA Concept

The aim of our approach to instruction-based dynamic clock adjustment is to significantly improve its granularity in two ways in contrast to the presented related work. To reduce the unexploited remaining dynamic timing margins by as much as possible, we aim to differentiate between a larger set of different instruction types, and for each of these types provide a clock period that is as close to the required period as possible, i.e., employ a very fine-grained clock adjustment.

To this end, let us begin by categorizing the different path delays present in a pipelined processor, and define instruction-based DCA formally.

Definition of DCA for Pipelined Processors

As almost any digital circuit, a pipelined processor typically consists of a set of N unique combinational paths $\mathcal{P} = \{p_1, ..., p_N\}$, which are characterized by their delays $D(p_n)$ for $n = 1, ..., N$ (each including the setup time of the endpoint). In a pipeline with S stages, each path can be attributed to exactly one stage $s = 1, ..., S$, based on its endpoint, and we can define corresponding exclusive per-stage path groups $\mathcal{P}^s$ such that $\bigcup_{s=1}^{S} \mathcal{P}^s = \mathcal{P}$ and $\mathcal{P}^s \cap \mathcal{P}^{s'} = \emptyset$ for $s \neq s'$. In a synchronous design, the longest path (in terms of delay) across all pipeline stages determines the clock period such that

$$T_{\text{clk}}^{static} = \max_{s=1,...,S} \left\{ \max_{p \in \mathcal{P}^s} \{D(p)\} \right\} = \max_{p \in \mathcal{P}} \{D(p)\}. \tag{5.1}$$

However, the bound put on the clock period in (5.1) is overly pessimistic since it ignores the fact that in each pipeline stage, the respective longest path $\max_{p \in \mathcal{P}^s} \{D(p)\}$ is not always excited. Instead, at time t, both operands and especially the instruction $I_s[t]$ that is currently evaluated in stage s determine a subset of relevant paths $\mathcal{P}^s_{I_s[t]} \subset \mathcal{P}^s$ that is sufficient to determine the position of the next clock edge for that pipeline stage. Since all registers are connected to the same clock, we can now determine a safe, but less pessimistic clock period for the time instant t as follows

$$T_{\text{clk}}^{dyn}[t] = \max_{s=1,...,S} \left\{ \max_{p \in \mathcal{P}^s_{I_s[t]}} \{D(p)\} \right\} \leq \max_{p \in \mathcal{P}} \{D(p)\}. \tag{5.2}$$

The corresponding relevant (active) instructions $I_s[t] = L[t + 1 - s], s = 1, ..., S$ are thereby determined by the S subsequent instructions in the program trace $L[t], t = 1, ..., M$ which

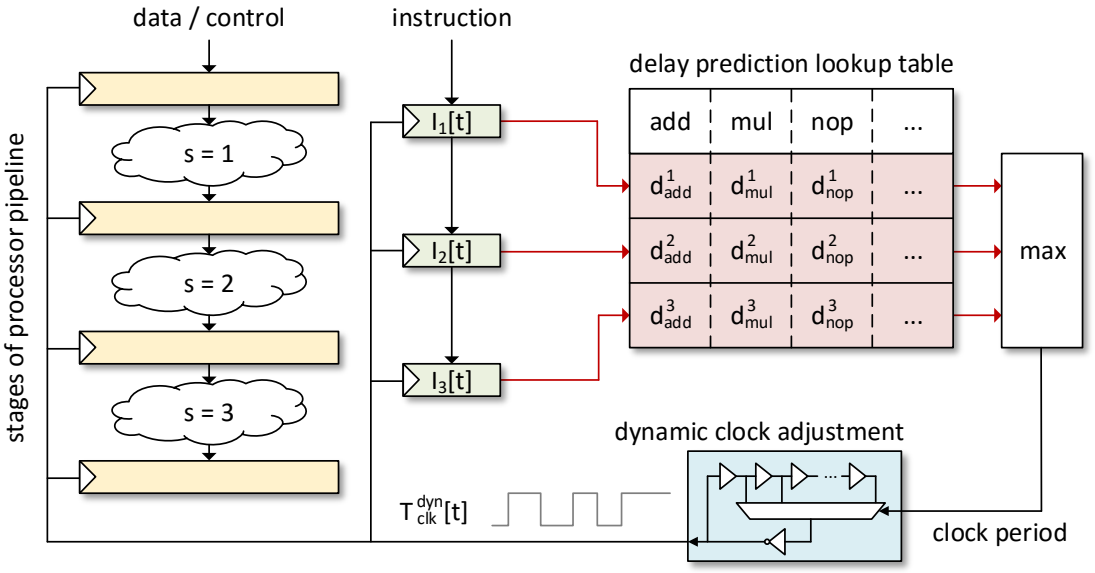


Figure 5.1 – Proposed instruction-based dynamic clock adjustment (DCA) technique in a S=3-stage pipelined processor, with lookup table for worst-case instruction-dependent stage delays.

influence the average cycle time T_{clk}^{avg} taken over the entire program execution:[2]

$$T_{\text{clk}}^{avg} = \frac{1}{M} \sum_{t=1}^{M} T_{\text{clk}}^{dyn}[t]. \tag{5.3}$$

Less formally, we can say that in each cycle the *relevant path* for each pipeline stage depends on the instruction[3] that is currently in that stage. The longest relevant path across all stages then determines the clock period for that cycle that is necessary to avoid timing violations.

The main idea behind our approach is to exploit different timing requirements of the various instruction sequences in the program trace to opportunistically over-scale the average frequency (i.e., to operate effectively at $T_{\text{clk}}^{avg} < T_{\text{clk}}^{static}$). To this end, we propose to adjust the clock period on a cycle-by-cycle basis, ideally according to (5.2). The corresponding architecture, as shown in Figure 5.1 monitors the instructions $I_s[t]$ that are in-flight in all processor pipeline stages $s = 1, ..., S$ and uses a lookup table (LUT) for each stage that contains for each instruction type the maximum delay of all relevant paths $d_I^s = \max_{p \in \mathcal{P}_I^s} \{D(p)\}$ in that stage. These per-stage maximum delays (for the respective current instructions $I_s[t]$, $s = 1, ..., S$) are then combined to yield $T_{\text{clk}}^{dyn}[t]$ and to adjust a tunable clock generator (CG) accordingly in each cycle.

[2] For simplicity, we assume long programs with limited impact of initialization effects.

[3] The delay of the *relevant path* also depends on other conditions, such as the operand values, state of the forwarding logic (influenced by the instructions in other stages), as well as the dynamic physical variations (such as temperature), which are all accounted for by considering their worst case effects on the path delay.

Such a CG can for example be realized in form of a tunable delay-locked loop (DLL) with a multiplexed clock output [KKK+06], or via a multi-PLL clocking unit such as the one proposed in [TKD+07]. We note that the design of an appropriate CG can have a significant influence on the system power consumption, and requires special care. Moreover, fine granularity and a high number of different available clock period settings that can be switched between instantaneously is desired for effective DCA. In Section 5.3 we present our hardware solution for clock generation with DCA, in the form of a tunable ring-oscillator with cycle-by-cycle period adjustment over a wide range with fine granularity.

5.1.3 Design Flow and Evaluation Environment

The design flow for the implementation and performance evaluation of the proposed cycle-by-cycle DCA technique is shown in Figure 5.2.

5.1.3.1 Implementation

The first step in the design process is the RTL implementation, optimization, synthesis, and physical design (full place & route) of a suitable microprocessor[4]. While we keep the objective of achieving a high clock frequency, as determined by static timing analysis, we also note that this worst-case timing is not the only objective to obtain a high *average* clock frequency

[4]The illustration in Figure 5.2 is based on an OpenRISC core that is employed for our case study in Section 5.2.

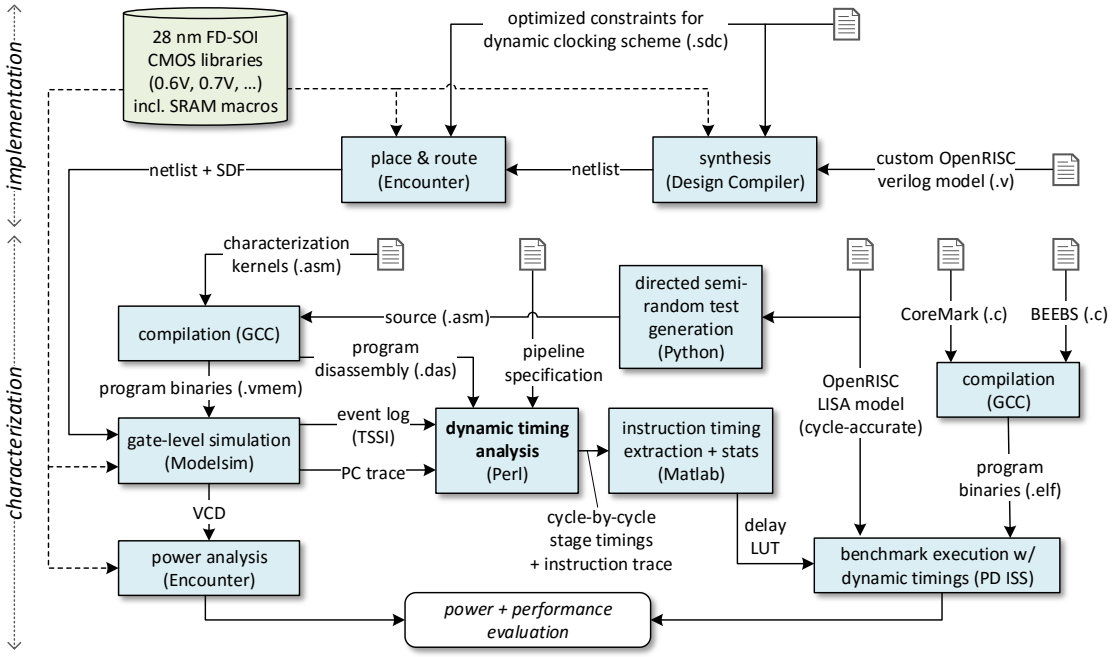


Figure 5.2 – DCA design flow including dynamic timing analysis, instruction timing extraction and power & performance evaluation.

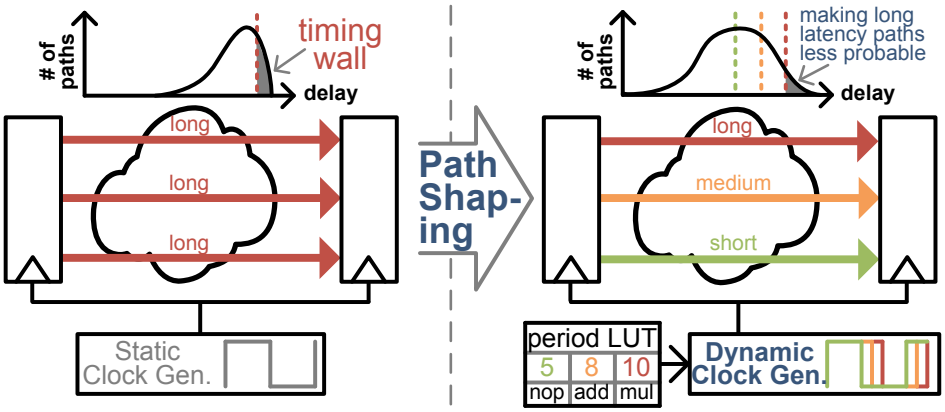


Figure 5.3 – Shaping of path delay profile for the DCA approach.

with DCA. Instead, it is important to not only optimize the critical path of the design, but also to reduce the number of near-critical paths and to keep the remaining paths short to minimize $\max_{p \in \mathcal{P}_I^s} \{D(p)\}$, $s = 1, ..., S$ for all (or at least all frequent) instructions I. Unfortunately, conventional implementation strategies and well balanced pipelines tend to produce a so-called timing wall as shown on the left-hand side of Figure 5.3, since they focus on the critical path only and allow other paths to become near-critical to recover area or power [KKKS10]. While this timing wall has no negative impact on the static timing limit, it affects the possible gains available from the proposed DCA technique. We therefore propose to apply suitable synthesis and implementation constraints (and possibly dedicated tools as in [KKKS10]) that also optimize sub-critical paths to keep them short, as illustrated on the right-hand side of Figure 5.3. While this objective usually involves some overhead (in area, speed, and/or power), we will later show that this overhead can be kept small, when discussing in Section 5.2 the specific processor core chosen for our DCA case study. Additionally, to avoid the timing wall in the synthesis and place & route step, optimizations at the RTL can be used to remove long, but functionally irrelevant paths from $\mathcal{P}_I^s$, for example through shielding or silencing. Section 5.3 provides some more specific examples of such optimizations for our silicon prototype implementation of an OpenRISC processor with DCA.

5.1.3.2 Characterization

The second step in the process is the characterization of the dynamic timing margins per instruction and per stage. This information is used to predict the potential performance (speed and power) improvements of the approach, to identify possible bottlenecks in the architecture, and eventually to populate the lookup-table for the clock period adjustment of the design. To this end, we start from the final netlist and the standard delay format (SDF) post-layout timing information of the design and perform gate-level simulations. We run different test programs, which include small hand-written kernels as well as semi-random test-cases that are generated by a code generation tool. The simulation outputs value change dumps (VCDs) that are used for power analysis based on the switching activity of the core. Moreover, the

evolution of the program counter is recorded to be able to generate a program trace $L[t]$ from the disassembled binaries. The next step is the dynamic timing analysis, as introduced in Section 4.2. In contrast to conventional static timing analysis, we are interested in the delay after which each path endpoint settles in each cycle during execution of actual programs. As has been discussed earlier, dynamic timing analysis aims to uncover the corresponding unused timing margins of the processor that are available at runtime, which cannot accurately be characterized through static timing analysis, due to the missing notion of path activation probabilities. DTA provides the available dynamic slack for all endpoints within the processor pipeline for each clock cycle. The time-average over the worst-case slack among all endpoints in each cycle then allows to derive an optimistic lower bound T_{clk}^{min} for the average clock period T_{clk}^{avg} of the processor, including all data and instruction dependencies:

$$T_{\text{clk}}^{min} = \frac{1}{M} \sum_{t=1}^{M} \max_{p \in \mathcal{P}_e(t)} \{D(p)\}, \tag{5.4}$$

where $\mathcal{P}_e(t) \subset \mathcal{P}$ is the subset of all excited paths of the circuit at time t. We furthermore define $\mathcal{P}_e^s(t) \subset \mathcal{P}_e(t)$ as the subset of excited paths at time t which go to an endpoint in stage s.

In a next step, the DTA tool partitions path endpoints into path-groups $\mathcal{P}^s$ according to a provided pipeline specification of the processor and determines the longest delay per pipeline stage and per cycle

$$d^s[t] = \max_{p \in \mathcal{P}_e^s(t)} \{D(p)\}, \tag{5.5}$$

with $s = 1, ..., S$ and $t = 1, ..., M$. The maximum delays d_I^s per instruction and per pipeline stage are then extracted by taking the maximum across all occurrences of that instruction in the program trace L as

$$d_I^s = \max_{\{t:L[t]=I\}} \{d^s[t+1-s]\}. \tag{5.6}$$

The worst-case delays d_I^s for $s = 1, ..., S$ and all instructions I are then exported and used to populate the delay prediction LUT (compare Figure 5.1) that can later be used for the DCA procedure. Note that the maximization in (5.6) essentially extracts the worst case across all occurrences of I in the stage s. Hence, (5.6) provides a safe limit on the cycle time for each pipeline stage s, if only the corresponding instruction type I is known. From this limit, we can later compute a realistic prediction of the speedup if only information on I is used for DCA, according to the dynamically set clock period $T_{\text{clk}}^{dca}(t)$ at time t:

$$T_{\text{clk}}^{dca}(t) = \max_{s=1,...,S} \{d_{L[t+1-s]}^s\}. \tag{5.7}$$

Moreover, the average clock period under instruction-based DCA which is lower-bounded by

T_{clk}^{min} (as defined in (5.4)) is given by

$$T_{\text{clk}}^{dca_{avg}} = \frac{1}{M}\sum_{t=1}^{M} T_{\text{clk}}^{dca}(t) \geq T_{\text{clk}}^{min}. \tag{5.8}$$

5.1.3.3 DCA Evaluation

We evaluate the potential performance gains for an arbitrary program with the help of a cycle-accurate instruction set simulator of the processor core, which is enhanced to be aware of the dynamic clock adjustment technique. Consequently, performance evaluation can be performed on full benchmark suites, since the ISS typically executes programs with some orders of magnitude higher throughput compared to a fully timing-annotate gate-level simulation. While still operating on a cycle-by-cycle basis, the ISS accounts additionally for varying real-time instruction delays according to the delay table generated by our dynamic timing analysis tool flow. To this end, the ISS provides next to a cycle counter also an execution time counter, which is used to accumulate the clock period duration for each simulated cycle (5.7). It is hence not only required that the ISS operates cycle-accurate, but also that it has a notion of the processor pipeline state, such that in each cycle the maximum clock period can be calculated according to the instructions residing in the pipeline and their corresponding delay entries in the LUT, as illustrated in Figure 5.1.

The speedup S_{DCA} due to DCA compared to standard static clocking is then simply calculated by the ratio

$$S_{\text{DCA}} = \frac{n_{cyc} \cdot t_{period}}{t_{exec}} \tag{5.9}$$

where n_{cyc} is the total number of cycles executed by the ISS at the end of a program/benchmark, t_{period} is the static clock period according to the worst path delay indicated by STA, and t_{exec} is the total execution time as accumulated by the DCA-aware ISS (according to (5.7)).

5.2 DCA Case Study Based on Instruction Set Simulation

The proposed DCA approach is applied to a general purpose open-source processor core, the OpenRISC [Ope16, Ope14]. Section 5.2.1 discusses the microarchitecture of that core, including our microarchitectural modifications, as well as the applied constraints for reducing (i.e., smoothing) the timing wall and making the original core more suitable for our technique. In Section 5.2.2 we describe the specifics of the employed performance evaluation environment, based on a custom OpenRISC instruction set simulator. In Section 5.2.3 we provide the characterization and evaluation results of the OpenRISC with DCA.

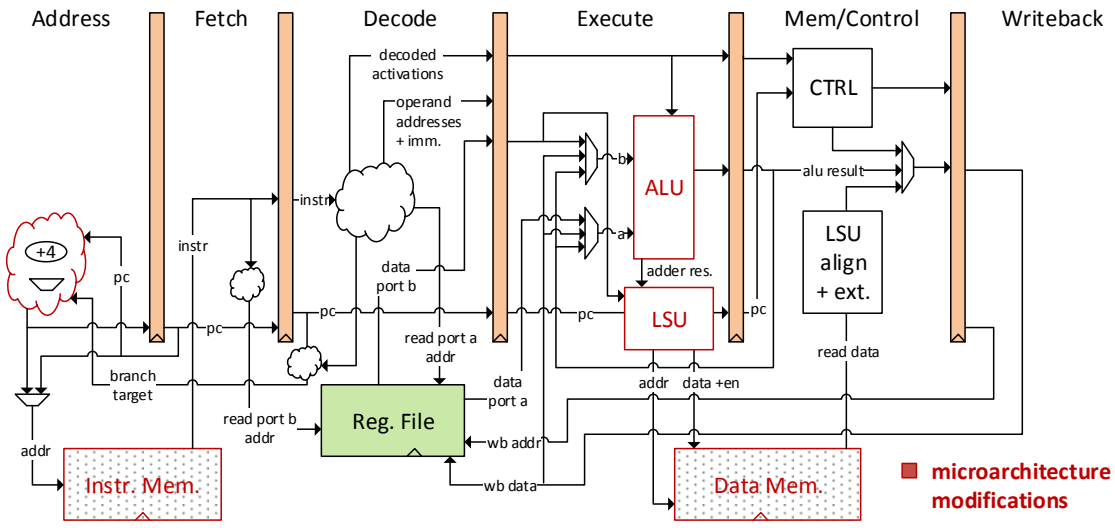


Figure 5.4 – Customized mor1kx microarchitecture of OpenRISC.

5.2.1 OpenRISC Microarchitecture, Optimization, and Implementation

The mor1kx cappuccino microarchitecture [KB+14] implementation of the OR1000 OpenRISC architecture [Ope14], which is an in-order 32-bit RISC pipeline consisting of six stages, is used as a basis for our case study. The schematic of our adapted core microarchitecture is shown in Figure 5.4. The fetch unit and load store unit (LSU), as well as parts of the arithmetic logic unit (e.g., the multiplier) have been optimized to achieve close to one instruction per cycle (IPC) execution throughput. Furthermore, the microarchitecture of the pipeline has been modified in order to support a tightly coupled memory interface, for the instruction as well as the data memory.[5] Since both memories are implemented as fast SRAMs, access latency for instructions and data are both single-cycle. Other modifications include also the integration of the data memory requests and parts of the load store unit in the execute stage. Most modifications of the base microarchitecture are aimed at increasing the IPC metric, such that it is as close to one (1) as possible, i.e., avoidance of any stall cycles or pipeline bubbles. Besides the throughput increase, the motivation behind these optimizations is to achieve a more predictable flow of instructions through the pipeline, which allows the DCA to be performed in a simpler fashion.

The microarchitecture moreover employs a single 32-entry register file with two read ports and one write port. The ALU is comprised of an adder and a single-cycle multiplier (with 32-bit output), along with a shifter, and supports moreover standard logical operations (AND, OR, XOR, etc.).

To increase the opportunities for shorter clock cycles, the multiplier has been shielded from the inputs of the other units of the ALU by separate, parallel registers that are only loaded

[5] Note that the default cache support of the reference mor1kx cappuccino implementation has been removed in favor of a tightly coupled memory architecture.

for multiplications. This measure avoids unnecessary "parasitic" activity in the multiplier for operations other than multiplications, which not only reduces power but also avoids excitation of the long paths through the multiplier during other operations.

To reduce the issue of a timing wall and to increase the number of short paths as described in Section 5.1.3.1, we utilize the *critical range* optimization feature of Synopsys Design Compiler (DC) along with path over-constraining during synthesis in topographical mode. The *critical range* feature of DC directs the synthesis optimization process to further work on paths that are near-critical or not critical. For example if a synthesis optimization of a design ends up with a critical path delay of d_{crit} for the longest path of the design, other mechanisms such as area recovery will try to relax most other paths, such that they end up having a delay that can be close to d_{crit} to save area and power, e.g., through gate down-sizing, consequently creating the discussed timing wall. However, with a *critical range* setting of c_{range} applied to all endpoints of the design, DC now considers during synthesis all paths that exceed the delay $d_{\text{crit}} - c_{\text{range}}$ also as critical, and tries to work on them, as well. This allows to push back all paths which actually can be made short, such that they eventually have a decreased delay, even when the overall critical path is not shortened by this optimization.

Clearly, this optimization does not come for free in terms of area and power. However, a comparison to a core without the critical range optimization shows that both the area overhead and the increase in power consumption can be limited to around 5-13% in a 28 nm FD-SOI CMOS technology, depending on the target operating voltage, process corner, and temperature (i.e., depending on the target library). It has to be noted that together with the general inherent uncertainty of a few percent of synthesis results, we did not see any specific trends regarding the targeted operating conditions and the amount of induced overhead in area or power, due to *critical range* optimization for DCA. However, we did not encounter any conditions where the penalties exceeded 13%.

The positive effects of the applied design step for our DCA technique are shown in Figure 5.5, which reports the changes for the dynamic maximum instruction delays, when applying the critical range optimization, compared to a standard implementation of the same design. Interestingly, for this specific design the overall minimum clock period derived from static timing analysis increases by 10% with the critical range constraints, due to unwanted side effects of the synthesis tool.[6] However, the worst-case delay excited by most instructions reduces significantly compared to the conventional design, and the increase in delay is only visible for the multiplication instruction.

As indicated in the design flow description in Section 5.1.3, after synthesis the OpenRISC core is always fully placed & routed, together with the required memories for instruction and data, before it is simulated and analyzed regarding its dynamic timings. The layout of the

[6] Note that this worsening of the critical path delay is not a general/fundamental issue of the applied path reshaping concept, but rather a specific artifact of the applied synthesis flow and tools, which unfortunately could not be avoided at the time, for this specific implementation of the OpenRISC that is employed for the case study in this section.

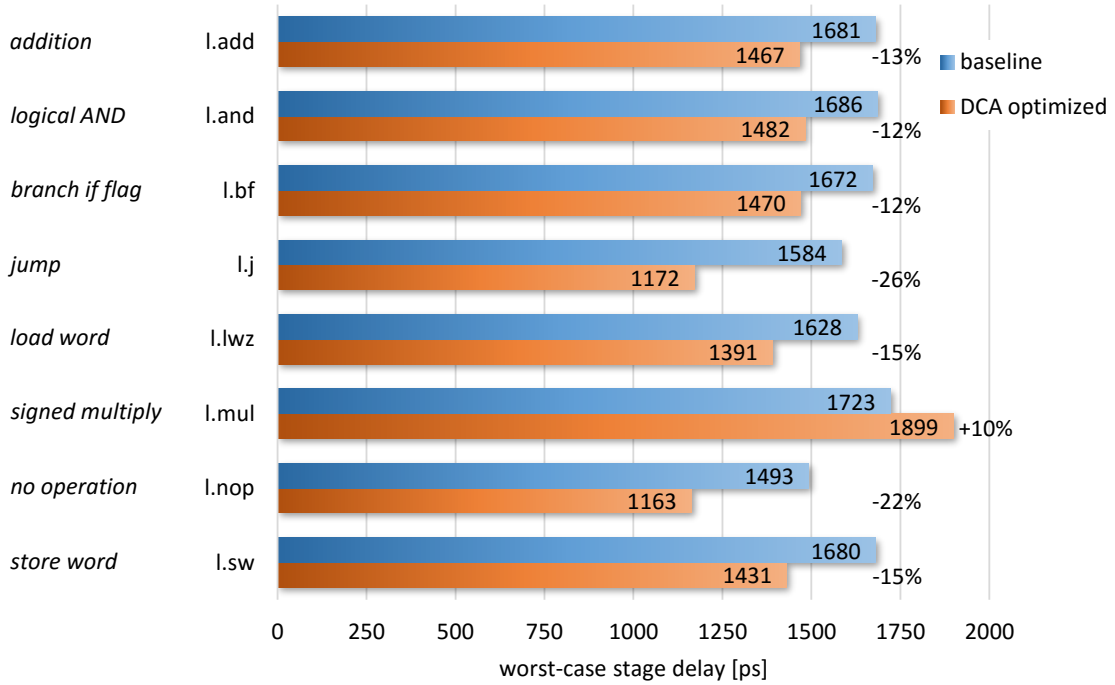


Figure 5.5 – Effects of critical range optimization for DCA on the worst-case delays for some exemplary OpenRISC instructions. The worst-case delays for each instruction occur in the execute stage, apart from l.j, where the longest delay is found in the address stage.

placed & routed OpenRISC core in 28 nm FD-SOI CMOS is shown in Figure 5.6. On the top and bottom there is one 16 kB (4096×32 bit) high-density high-performance SRAM macro each, while the CPU core is placed in the middle of the layout, occupying only part of the middle of the floorplan, due to the very wide aspect ratio of the SRAM cuts. The DTA and DCA performance results presented in the following Section 5.2.3 are based on this initial implementation. Moreover, please note that this initial implementation does not comprise any circuitry for performing actual DCA in hardware, i.e., with a dynamic clock. At this stage the design only encompasses the processor core with a shaped timing profile as discussed, including the SRAMs for accurate timing behavior.

5.2.2 Performance Evaluation Environment

For this case study, we base our instruction set simulator on a custom developed LISA model of the presented OpenRISC core microarchitecture, and employ the Synopsys Processor Designer tool suite [Syn12] to generate the custom simulator from this model. This model is then enhanced to be DCA-aware, including the necessary provisions to track the execution time, as has been outlined in Section 5.1.3.3. To this end, we incorporate header files into the generated C++ source code of the ISS. These header files contain the instruction delay tables, generated by the dynamic timing analysis tools.

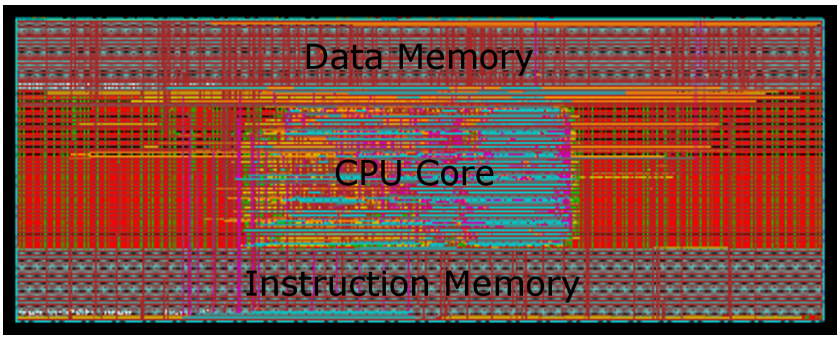


Figure 5.6 – Layout of placed & routed OpenRISC core optimized for DCA in 28 nm FD-SOI CMOS with instruction and data memories, realized each as 16 kB high-density high-performance SRAM macros.

Performance benchmarks are then executed, using the popular embedded benchmark suites CoreMark [EEM16] and BEEBS [PHB13]. Their C sources are compiled with the standard OpenRISC GNU toolchain (GCC).

The execution time and speedup results of the simulator are finally combined with the extracted power consumption values to evaluate the energy efficiency gains from voltage-frequency scaling for a given iso execution time as typically encountered in embedded applications with real-time constraints. The performed voltage-frequency scaling is based on repeated VCD-based gate-level simulations followed by power analysis in Encounter, using fully characterized cell libraries for the different operating points.

5.2.3 Characterization and Performance Results

We present the results of our proposed DCA design method in two parts: The first part focuses on the characterization of the dynamic timing margins per instruction and per stage for the OpenRISC core by applying the developed flow as described in Section 5.1.3 and also evaluates a lower bound on $T_{\text{clk}}^{dca_{avg}}$ according to (5.4). The second part leverages these margins and presents the achievable performance and power gains, when executing popular benchmark suites on our enhanced core with instruction-based DCA.

5.2.3.1 Dynamic Timing Analysis of OpenRISC

Before characterizing the core timing on an instruction level, we study an upper bound on speed improvements from dynamic clocking with a genie-aided clock adjustment according to (5.4). To this end, we perform dynamic timing analysis, but initially we do not enforce a worst-case clock period for all occurrences of the same instruction. Instead, we assume that the duration of each cycle can be adjusted according to the a-posteriori measured worst-case dynamic timing slack over all endpoints as described in Section 5.1.3.2. Figure 5.7 shows that considering all endpoints of the core (including the SRAMs), on average we only require a

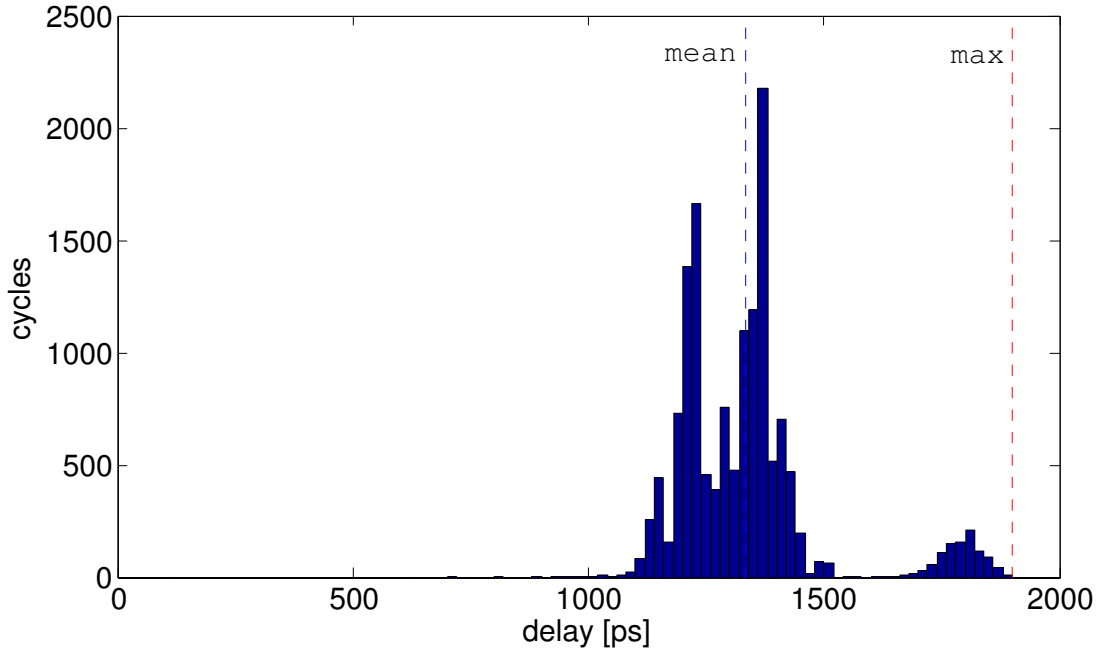


Figure 5.7 – Histogram of dynamic maximum delays per cycle over all pipeline stages and endpoints of the OpenRISC core, derived via DTA.

delay of 1334 ps for the correct program execution, in contrast to the conservative limit of 2026 ps given by the pessimistic static timing analysis. With correct program execution we mean that all timing requirements of all excited paths in any given cycle are always met. The histogram shows the distribution of the required clock period available in the full core on a cycle-by-cycle basis. Exploiting these dynamic timing limits, the theoretical average speedup is 1.52x, for a arithmetic-focused characterization kernel, containing a significant amount of multiplication cycles (≈15%).

When introducing endpoint groupings according to the pipeline stages, our analysis shows that the execution stage is by far the dominating stage, regarding worst-case dynamic paths,

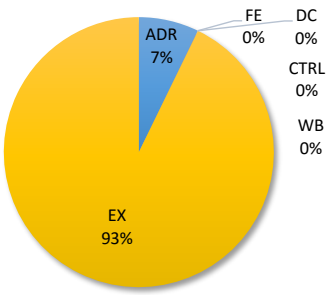


Figure 5.8 – Percentage of a pipeline stage containing the limiting path, which determines the minimum cycle time when employing dynamic clocking.

as shown in Figure 5.8. In particular, for 93% of the cycles the overall maximum delay, defining the length of a cycle in Figure 5.7 is due to an endpoint that is located at the end of the execute stage (i.e., between the execute and control stage), which can be a pipeline register or the data memory SRAM macro cell. In 7% of the cases, endpoints attributed to the address stage are the limiting factor, which corresponds to the instruction memory endpoints. Rarely, the fetch or decode stage can also be the limiting stage, however these cases appear in less than 1% of the cycles, and the limiting delays are in these cases very short.

These results indicate that clock adjustment can be performed in the case of our OpenRISC core by considering only the delay of an instruction in the execute stage, as long as it is guaranteed that the instruction memory address timings (address stage) are always respected, as well as the few cases where other stages are limiting, which however have a short delay in general for these cases. This fact can significantly simplify the clock adjustment control module, since its pipeline monitoring can be simplified, as we will see in more detail for the implementation of the DynOR microarchitecture in Section 5.3.

Table 5.1 presents a selective overview of extracted maximum instruction timings based on a characterization benchmark with a gate-level simulation of 14 k cycles. Such a table (with 6 stage-entries per instruction) is employed in our instruction set simulator to accurately model the instruction delays. Instructions where no accurate maximum delay characterization could be performed (due to limited number of occurrences in the benchmark) are represented in the table with the worst-case clock period timings from static timing analysis.

In Figure 5.9 we illustrate the dynamic timing distributions for the 6 pipeline stages for the l.mul instruction (signed 32-bit multiplication), as derived from our DTA tool. It can be

Table 5.1 – Instruction delay worst-cases including the limiting stages where they occur, for a representative set of different OpenRISC instructions.

Instruction	**Function**	**Max. delay [ps]**	**Limiting stage**
l.add(i)	addition (with immediate)	1467	EX
l.and(i)	logical AND (with immediate)	1482	EX
l.bf	branch if flag	1470	EX
l.j	jump	1172	ADR
l.lwz	load word & zero extend	1391	EX
l.mul	signed multiplication	1899	EX
l.nop	no operation	1163	EX
l.sll(i)	left shift (with immediate)	1270	EX
l.sw	store word	1431	EX
l.xor	logical XOR	1514	EX

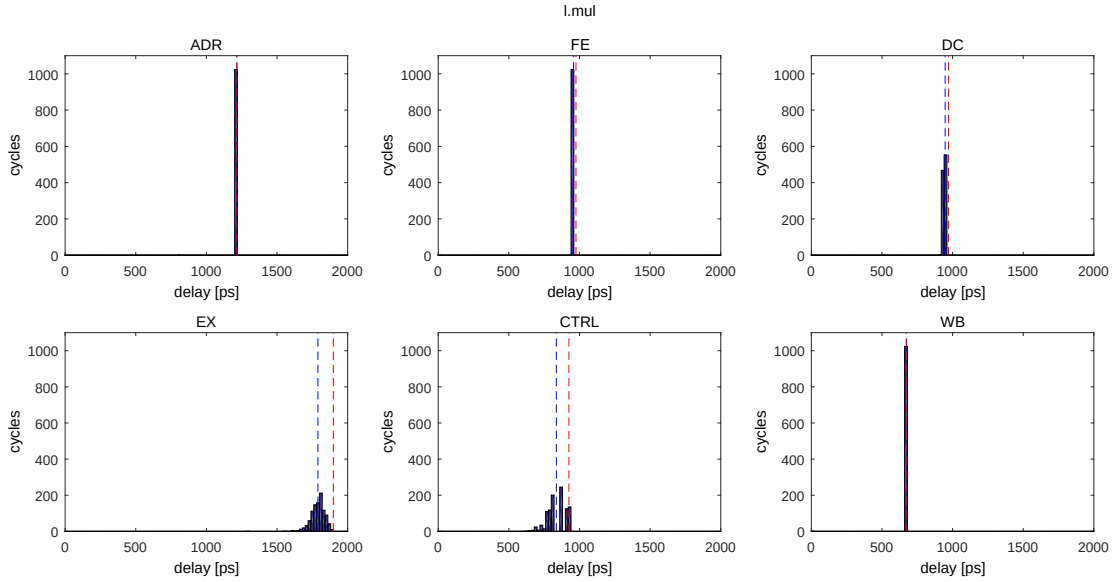


Figure 5.9 – Histograms of dynamic maximum delays per pipeline stage, for the l.mul instruction (signed multiplication). The red lines indicate the maximum delay, while the blue lines indicate the mean delay.

observed that while the delay of the instruction in the execute stage is in general high (and close to the static maximum), delays for the other stages are significantly lower, which can be exploited when other, faster instructions are currently in the execution stage. The distribution of delays with a spread of about 300 ps for the execute stage stems from the varying data dependent activations of worst-case paths. This data dependent timing variation could be further leveraged by approximate computing techniques [CVC+13], which would introduce the notion of using shorter clock periods to enhance performance or save energy, while actually allowing a violation of the timing requirements of certain paths in the design. As we have shown in detail in Section 4.3, these timing violations would then produce approximate results, for example for the output of our multiplication instruction, due to paths in the execution stage of the multiplier circuit being violated under certain conditions (e.g., critical operands exciting the worst paths).

5.2.3.2 Performance and Power

We evaluate the performance gains of the presented dynamic clock adjustment method by executing the popular CoreMark [EEM16] and BEEBS [PHB13] embedded benchmark suites on our custom instruction set simulator with integrated delay tables. The benchmark-dependent speedups are reported in Figure 5.10. On average the effective clock frequency can be increased by 38%, from 494 MHz (static timing limit) to 680 MHz at a fixed supply voltage of 0.70 V. We focus our evaluation in 28 nm FD-SOI CMOS on a target operating condition with relatively low supply voltage for reasons of better energy efficiency.

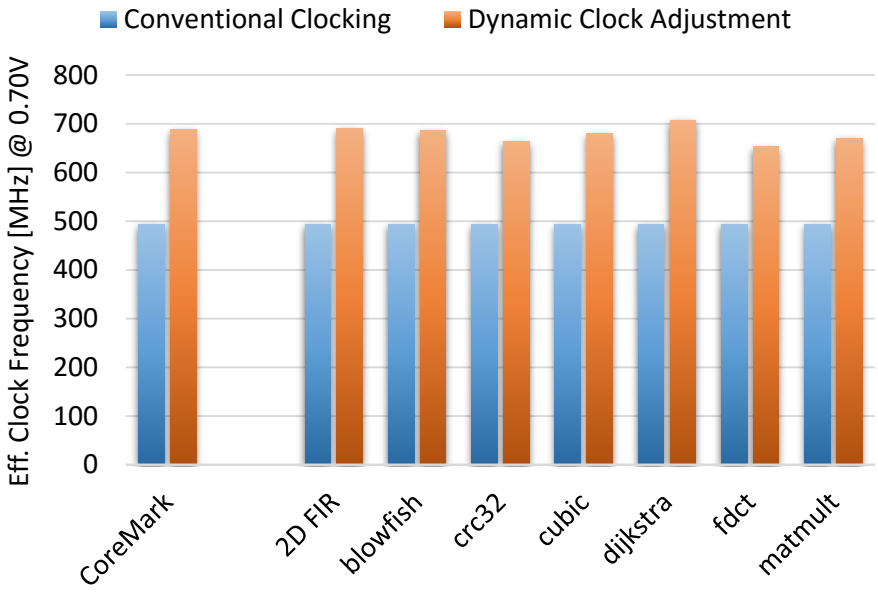


Figure 5.10 – Performance gains on a 32-bit OpenRISC core with dynamic clock adjustment over conventional static clocking, for the CoreMark and BEEBS benchmark suites.

The results show that in terms of clock speed on average we only loose 14% by performing clock adjustment based on the instruction types alone. This is compared to the theoretically achievable maximum speedup of 52% (1.52x), when adjusting the clock period perfectly each cycle, i.e. when considering all data and instruction dependencies, to fully consume all available dynamic timing margins.

The increase in performance can also be traded off for reduced power consumption through voltage scaling of the core, while operating at iso-throughput. The available speedup allows on average to operate the core with a supply voltage that is 70 mV lower, which translates into an improvement in energy efficiency by 25%. Under scaled voltage, but with dynamic clock adjustment, the core consumes only 11.0 μW/MHz while providing the same throughput, compared to 13.7 μW/MHz for the conventional clocking scheme.

5.3 DynOR Hardware Architecture

The final three sections of this chapter present a fully operational hardware prototype, named DynOR[7], demonstrating the feasibility of the introduced DCA approach. In this section we discuss the hardware architecture of DynOR, including all required core-external modules that enable the DCA technique on a real system, which were so far not considered in detail in the previous sections.

5.3.1 Chip Architecture

The DynOR architecture is depicted in Figure 5.11. At the core of the chip lies a 32-bit, 6-stage, in-order OpenRISC microprocessor with one integer ALU, supporting single-cycle 32-bit multiplications. The employed microarchitecture of the OpenRISC core is identical to the one presented in Section 5.2.1. Note that this includes especially RTL modifications that prevent unnecessary excitations of long paths in the execution stage, e.g., through shielding of multiplier paths in the ALU, via operand register replication.

The processor comprises 16 kB of tightly coupled instruction memory and 8 kB of data memory, both realized using single-cycle latency 6T SRAM macros. Specifically, the used macros, both for the instruction and data memory, are single-port high-speed SRAMs (of type *register file*) of size 1024×32 bit. The memory ports of the SRAMs are multiplexed, such that they can also be accessed through a bus interface which enables off-chip access.

Instructions are fetched into the processor pipeline while they are simultaneously analyzed by the DCA module, in order to determine the required next clock period. The clock generation unit supplies the dynamic clock within the core clock domain, which comprises the CPU, the memories, and the DCA, while the peripheral/interface components of the chip are externally clocked. All modules of the core clock domain are connected to those peripherals over a 32-bit bus interface. The CPU has also indirect access to this bus over a multi-cycle memory-mapped register interface (that only the CPU sees), which allows to send read or write requests to the bus. Note that the bus and the memory interface of the CPU need to be decoupled since they operate asynchronously in two different clock domains (dynamic core clock & external clock).

A custom serial interface provides communication for DynOR with the external world, while the central configuration and control module gives access to all clocks, resets, and different mode and calibration settings of the modules of the chip. Debug signals for the operation of the dynamic clock adjustment are observable via a monitor interface. A multiplexed general-purpose input/output (GPIO) interface also provides access to these debug signals, as well as to a scan chain.

The chip comprises two separate voltage islands (with level-shifted boundaries): one for

[7] The name *DynOR* stands for *dynamic OpenRISC*, i.e., an OpenRISC microprocessor with dynamic clock adjustment capabilities.

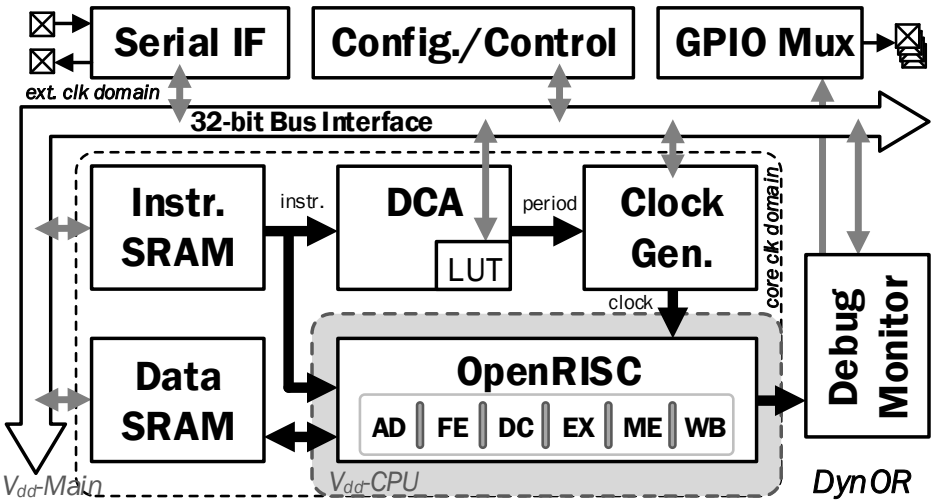


Figure 5.11 – Architecture of DynOR test-chip, including a 32-bit 6-stage OpenRISC core with cycle-by-cycle instruction-based dynamic clock adjustment.

the OpenRISC core (V_{dd}-CPU, marked in grey in Figure 5.11) and one for the rest of the chip (V_{dd}-Main). This allows for the operation of the CPU at low voltages, below the point where the SRAMs are functional, and enables accurate power measurement of the CPU core.

5.3.2 Dynamic Clock Adjustment Module

To limit the hardware complexity induced by the DCA technique, we ensure during implementation that all paths which do not reside in the execution stage (EX) of the pipeline are non-critical, or can be sufficiently accelerated through voltage scaling. This is possible with low additional overhead, as indicated by the findings we presented in Section 5.2.3 (specifically Figure 5.8), which show that even in the baseline DCA-design (prior to further optimization) 93% of all cycles are already limited by the execution stage. The acceleration of specific paths through voltage scaling is possible for the paths connected to the SRAMs. The access times of the SRAMs can be decreased, since they are placed in the global voltage island V_{dd}-Main which can be operated independently at a higher voltage, while the voltage of the core can be kept at a lower level in the separate V_{dd}-CPU island. This allows for example the sufficient acceleration of the instruction fetch mechanism, such that it is no longer possible that the address stage can become the limiting stage of the core, for any cycle.

The DCA module provides the functionality for dynamically determining the next clock period, based on the type of the currently fetched instruction. The schematic of the DCA module, together with the clock generation, is shown in Figure 5.12. During every cycle in which an instruction is fetched into the FE stage of the CPU, the DCA also receives the same 32-bit instruction from the memory bus. This instruction is then partially decoded (using a subset of the more complex decoding logic of the OpenRISC) to determine its type, such that it can

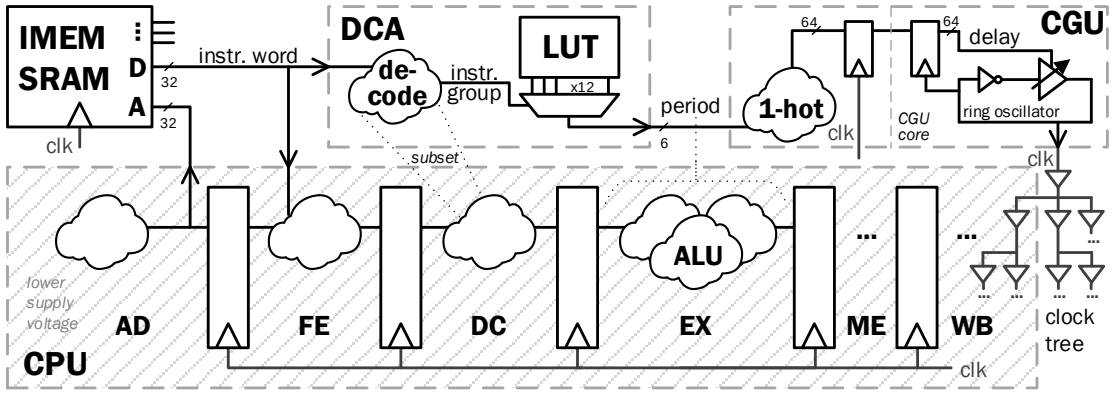


Figure 5.12 – Schematic of the dynamic clock adjustment and generation, within the DynOR architecture.

be classified into an instruction group. The chosen groups are as follows: branch, branch-condition, load, store, logic, xor, shift, add, multiply, nop, register-move, and other. The group is then used to address a 12-entry 6-bit lookup table, which provides the critical period, associated with the respective instruction type/group. As can be seen from Figure 5.12, this period setting only needs to come into effect for the cycle when the fetched instruction will reside in the execution (EX) stage of the CPU, which contains the critical paths. This latency allows the period setting to take two cycles to propagate to the delay configuration flip-flops which control the period of the ring oscillator in the CGU.

The lookup table of the DCA is populated through calibration of a die for the current operating condition (i.e., supply voltage and temperature). The minimum achievable period value of each entry is determined through binary search, where each run of the search is evaluated by testing the outcome of the program for correctness (including data memory contents), i.e., by checking if the current clock periods cause a timing error for one of the instruction groups, resulting either in an erroneous result or a processor crash. Note that the lookup table produced by a single calibration point is inherently tuned for a specific application (type). To generate a “worst case” table, which provides DCA capabilities independent of the application that is executed, multiple tables from subsequent calibration runs using different applications can be merged, where for each entry the worst observed period per instruction type is chosen. More details on the exact DCA LUT calibration process employed for DynOR are given in Section 5.4.3.

5.3.3 Clock Generation Module

The clock generation unit (CGU), shown on the right hand side of Figure 5.12, is based on a digitally controlled ring oscillator that produces a clock signal whose period can be changed in each clock cycle, as required by the proposed DCA technique. The schematic of the CGU core supporting 64 different period settings is shown in Figure 5.13.

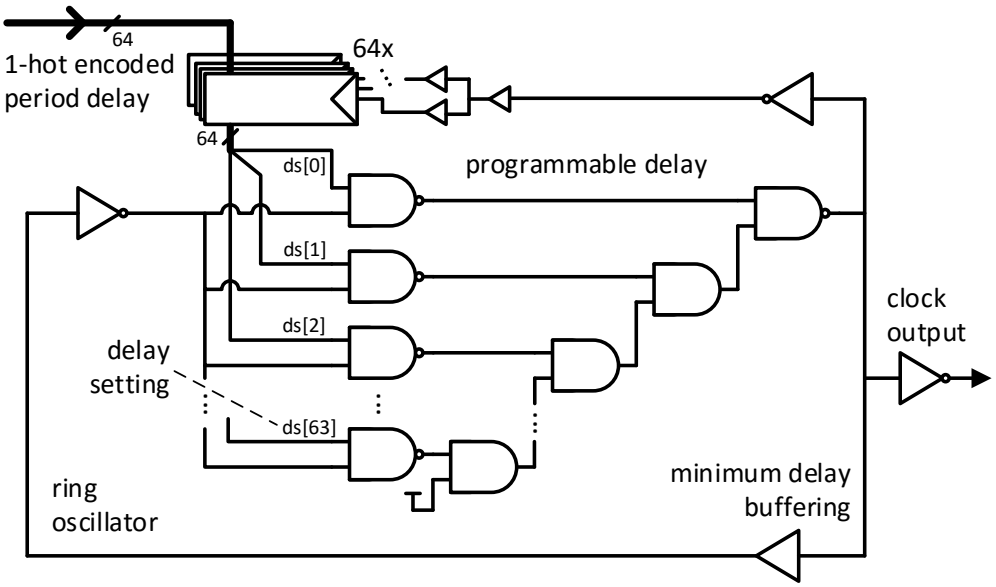


Figure 5.13 – Schematic of clock generation unit (CGU), allowing cycle-by-cycle clock period adjustment, via a 1-hot controlled programmable delay.

To modify the clock period, the ring oscillator includes a programmable delay unit implemented with a cascade of AND gates that have been laid out with controlled placement to produce an accurate range of clock periods. The cycle-by-cycle operation is ensured by using a 1-hot encoded period setting for the programmable delay unit, and by sampling this delay setting with a set of additional registers that have been placed close to the programmable delay unit inside the CGU core. This controlled placement of the registers in close proximity to the NAND gates performing the selection of the programmable delay path, is done to minimize both the propagation delay of the 1-hot encoded select signals, as well as to minimize the delay of the clock signal they receive from the ring oscillator. These registers are clocked as soon as the rising edge of the clock has propagated through the programmable delay block. This ensures that the delay setting is stable at the input of the programmable unit as soon as the next falling clock edge arrives at the input of the unit, thereby avoiding both glitches on the clock signal as well as any contamination of the selected clock period duration.

Moreover, note that the full CGU has been implemented using only standard cells of the available 28 nm FD-SOI design kit. However, by employing controlled placement and extensive circuit simulations of the post placed & routed CGU, it was possible to achieve a cycle-by-cycle dynamic clock period generation with very fine granularity, and sufficient accuracy.

5.4 DynOR Test-Chip

This section discusses various aspects regarding the fabricated DynOR test-chip. Section 5.4.1 gives a brief overview of the physical implementation of DynOR. The validation setup that has been created for the measurement and testing of the DynOR chips is described in Section 5.4.2. Finally, Section 5.4.3 details the employed calibration method for the DCA LUT, and then outlines an integrated/on-chip calibration technique using timing monitoring flip-flops.

5.4.1 Implementation

DynOR is designed in the 28 nm FD-SOI CMOS technology of ST [MFC13] in the regular-V_T process option, using a semi-custom digital design flow with a 12 track standard cell library. The front end design has been carried out with Synopsys Design Compiler in topographical mode to perform synthesis of the RTL code using *compile_ultra* with retiming optimization. Additionally, extensive *critical range* optimization is performed to achieve a timing profile of the CPU core which allows for effective DCA, as discussed in Section 5.2.1. Moreover, the design hierarchy is manually flattened (apart from the top-level modules) before synthesis. Two different voltage regions are set up for synthesis, according to the two islands V_{dd}-CPU and V_{dd}-Main. The chosen synthesis corner of the CPU island is slow-slow (SS) at 0.70 V and 125° C, while the main island (including the SRAM cells) uses the SS, 1.10 V, 125° C corner.

The back end design has been carried out with Cadence Encounter, to perform a standard physical implementation flow consisting of floor-planning, placement, clock-tree synthesis & insertion, routing, and hold fixing. The flow employs the same set of operating corners for the two voltage islands as for synthesis, with an additional fast operating corner (fast-fast (FF) at 1.10 V and -40° C) to perform hold fixing. The final layout of the designed test-chip is shown in Figure 5.14. Special care has been taken at the floor-planning stage to help with timing closure of the important path going from the instruction memory output through the DCA to the CGU (see top half of Figure 5.12). This path[8] requires special care, since even for the shortest possible clock period (i.e., when the fastest possible instruction is executed in the OpenRISC pipeline) this path has to derive the new period setting for the upcoming cycle from the fetched instruction word. Consequently, during floor-planing DCA and CGU are placed tightly together on the right-hand side of the chip. Both modules are then placed exactly next to the border where CPU and instruction memory SRAM meet. This then also allows for a good starting point of the clock tree/distribution, origination from the CGU. Additionally, double-registering of the 1-hot encoded period setting, together with the controlled placement of the CGU internal flip-flops allows this important path constraint to be met. Moreover, since this DCA path is fully part of the V_{dd}-Main domain, it can have a relative voltage boost compared to the paths of the CPU residing in the V_{dd}-CPU domain, which in turn results in enough margin on this path for all considered operating scenarios, according to STA results.

[8] Technically, this is a whole set of paths going from the multiple (data) output ports of the instruction SRAM to the inputs of the flip-flops holding the 1-hot encoded period setting.

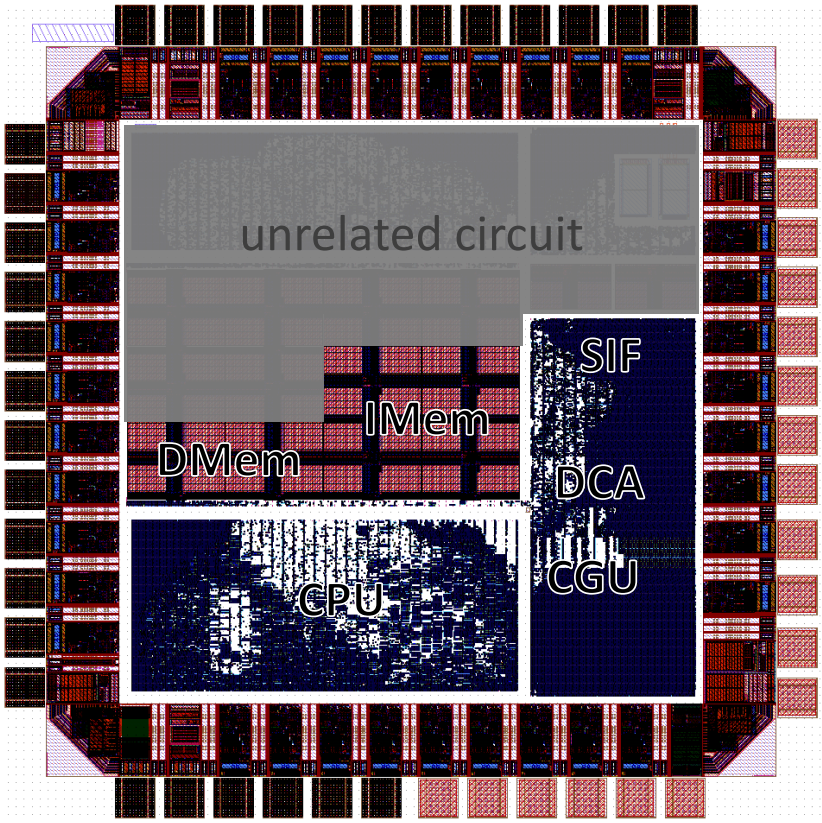


Figure 5.14 – Layout of complete 1.2 mm^2 test-chip design in 28 nm FD-SOI CMOS including pad ring with 36 I/O and 12 power supply pads. The bottom half of the chip contains the DynOR architecture, and the total core area occupied by DynOR-related circuitry, including the SRAM macros, amounts to only 0.24 mm^2.

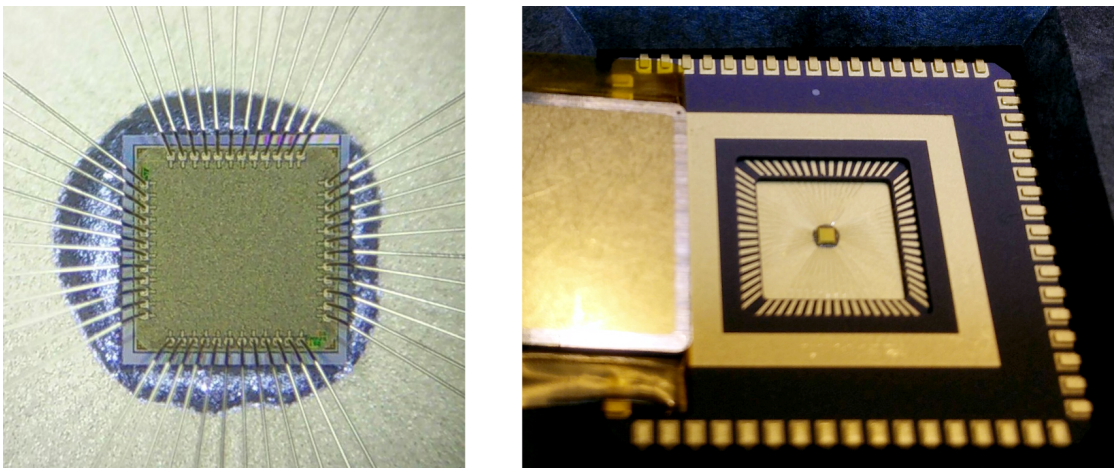


Figure 5.15 – Left-hand side: wire-bonded DynOR die (1.2 mm^2) with 48 pads in 28 nm FD-SOI CMOS; right-hand side: photograph of the die sitting in the cavity of an opened JLCC-68 package (2.4 cm × 2.4 cm).

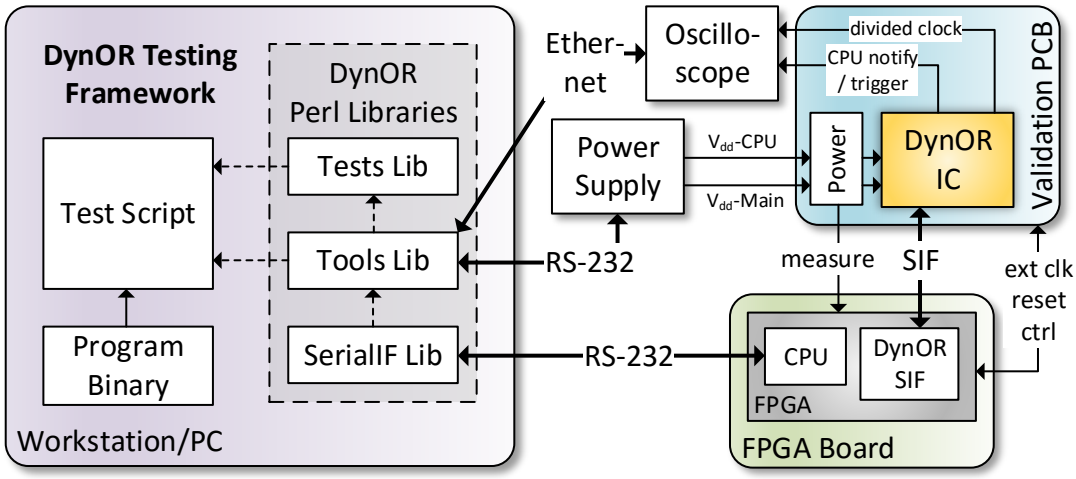


Figure 5.16 – Overview schematic of automated measurement and validation setup for DynOR, including a workstation.

5.4.2 Measurement Setup

An overview of the hardware and software measurement infrastructure for DynOR is given in Figure 5.16. Photographs of a wire-bonded die and JLCC-68 package that are used for testing are depicted in Figure 5.15, while Figure 5.17 shows a photograph of the hardware setup. The hardware measurement setup consists of three main components, a custom designed PCB hosting the DynOR IC, an FPGA board for pin control and bridging of the custom serial interface with an RS-232 interface, and a workstation/PC for control of the automated or interactive tests.

The custom PCB provides the appropriate I/O capabilities for supplying the external clock & reset, the bi-directional serial interface, the scan chains, and access to all remaining debug GPIOs, which for example allow the accurate measurement of the internally generated clocks from outside the IC. The PCB furthermore has extended support for sourcing and control of the different voltage domains required for the chip and provides automated power measurment capabilities on the PCB itself via dedicated power ICs. For serial communication and supply of the external clock & reset, the PCB is plugged onto an FPGA development board (Xilinx XUPV5-LX110T), hosting a Virtex 5 FPGA and providing an RS-232 connection. The FPGA hosts a small custom developed embedded system, which is mainly comprised of an embedded Microblaze processor, bridging in software the communication between the light-weight, custom designed DynOR serial interface (SIF) and the RS-232 port, which is connected to a workstation. The workstation communicates with DynOR over this FPGA interface, using a custom developed set of perl libraries targeted at easing the development of the various test scripts required for the wide range of functional verification and performance characterization tasks. The libraries on the host-PC furthermore provide extensive automation functionality regarding the regulation of supply voltages and capturing of external signals via the oscilliscope

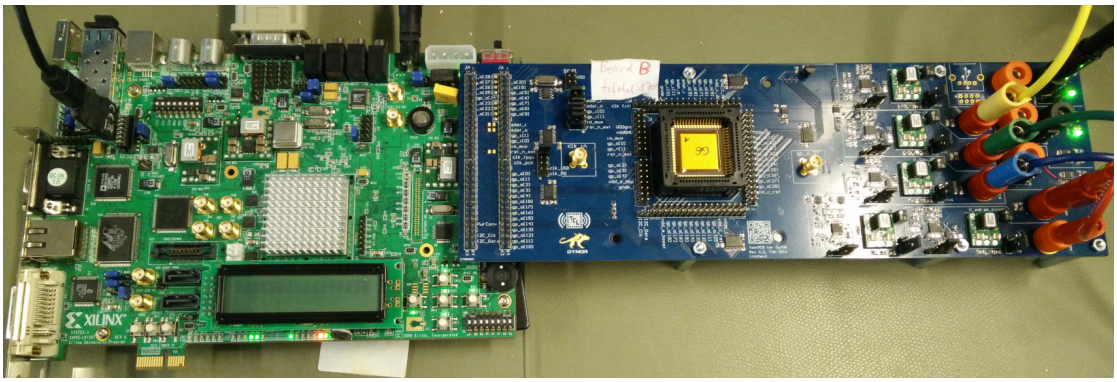


Figure 5.17 – DynOR measurement setup, comprised of a custom validation PCB on the right-hand side hosting the DynOR IC, connected to a Xilinx XUPV5-LX110T FPGA board.

that is integrated within the test setup. More specifically, convenient application programming interface (API) access for the following functionality is provided by the developed host-PC tools library of the DynOR testing framework, in order to ease the creation of automated tests:

- Direct read and write access for the DynOR internal memory interface (32-bit peripheral bus);
- Wrapper/helper functions for use of the DynOR configuration register map, which controls all functionality on chip (clocks, resets, modes, etc.);
- CGU and DCA (including LUT) setup and control;
- Functions for automated loading and execution of program binaries on the CPU, including hooks for custom extensions and data integrity checks;
- External chip reset;
- Frequency configuration for the external chip clock (speed of custom SIF);
- Configuration of GPIO multiplexer unit (via dedicated serial pin);
- Configuration/setup of all supply voltages (including current limits);
- Voltage measurement for all supply voltages;
- Current and power measurement (with multiple precision ranges), separately for each supply;
- Ambient temperature measurement;
- Chip package temperature measurement (via contacting sensor);
- Automatic capture of oscilloscope signals (including data dump) and configured statistics (e.g., frequency).

5.4.3 DCA LUT Calibration

While in the simulation case (Section 5.2) the period LUT entries are simply derived via DTA, on a real hardware system it is required that the LUT entries are calibrated according to the speed of the die (static process variations) and the current dynamic physical conditions (e.g., temperature) of the operating environment. Section 5.3.2 outlined our calibration approach, which is based on verification of program execution correctness. The exact calibration procedure that has been employed during measurements is presented in the flow chart shown in Figure 5.18. For a given set of applications, individual/application-specific LUTs are derived via binary search for all LUT entries, which represent the period values for the 12 different instruction types. The decision criterion for the binary search algorithm is, if a program could be executed 100% correct with the current LUT configuration. In case the execution failed (or the calculated results have errors), the LUT is adjusted, and execution is restarted. The final LUT then eventually contains the minimum periods for all instruction types, while executing this specific application. The process is repeated for other applications, and all resulting LUTs are eventually merged to form a worst-case LUT, by taking the maximum entry over all applications for each instruction type separately. The purpose of the worst-case LUT is to have a stable DCA configuration, which allows speedup over static clocking independent of the application.

On-chip Calibration

The DynOR architecture has been instrumented to allow automatic on-chip calibration of the DCA LUT, based on timing error detection. Unfortunately, due to unforeseen problems in the specific circuit implementation, this calibration mechanism is not functional in the tested chips. The suspected reason for the failure of this mechanism are buffering issues on very fast paths (i.e., the lack of buffering), which create a type of hold violation (for the employed detection mechanism only, not the actual flip-flop data inputs). Since the monitoring of internal timing behavior of an IC is very challenging, due to extremely limited visibility and controllably, unfortunately no further detailed analysis of this issue or characterization of the mechanism as a whole can be provided at the time.

Nevertheless, we present in the following an overview of the developed concepts which allow in principal on-chip calibration of a DCA LUT, without the need for a trial and error procedure (Figure 5.18) as employed instead in this work to obtain the results reported in Section 5.5. The basic idea of the on-chip calibration technique is to determine/monitor the available timing slack, without causing actual timing violations in the circuit. To this end, we use a flip-flop that can flag any transition on its data input D during the low phase of the clock. We can now use a safe, long calibration clock period, and vary the high phase of this clock to find the point in time when the last transition on D occurs. This idea is illustrated in the timing diagram shown in Figure 5.19. We see on the left-hand side that the worst-case delay for the instruction that is to be calibrated exceeds the high phase of the clock, and hence triggers the transition detection flag.

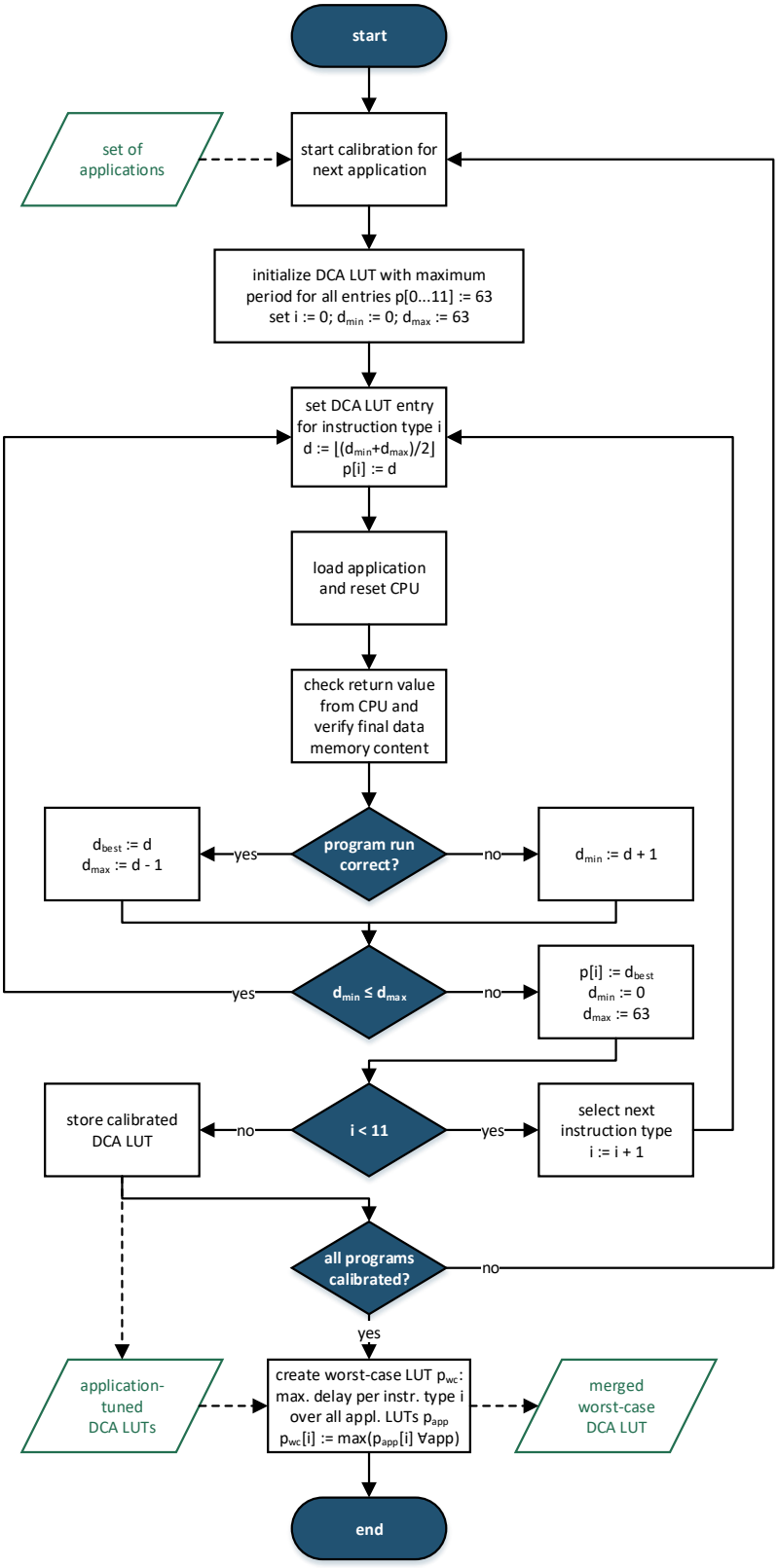


Figure 5.18 – Flow chart of the DCA LUT calibration procedure used for the measurements.

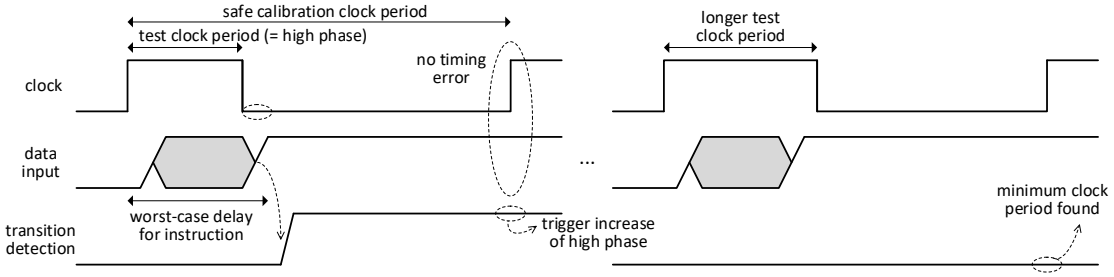


Figure 5.19 – Timing diagram, illustrating DCA LUT calibration using timing monitoring flip-flops.

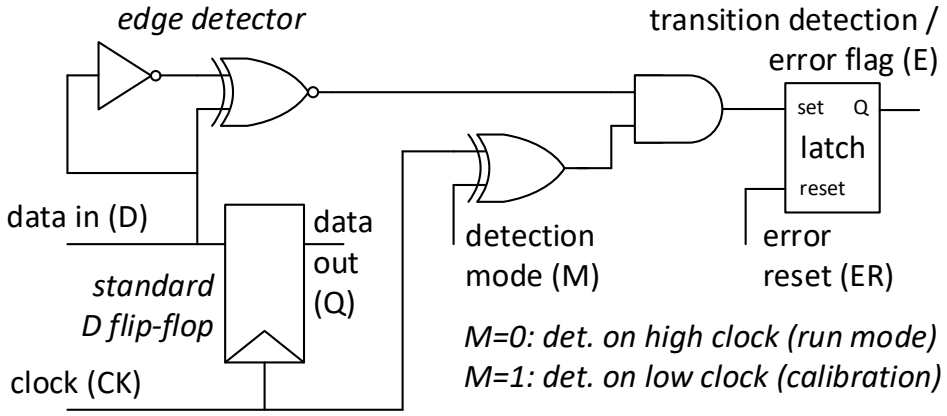


Figure 5.20 – Schematic of dual-mode timing monitoring flip-flop.

Moreover, we observe that even if a transition is flagged, no timing error occurs, since the actual total clock period is still long enough, which means that the circuit can be operated as normal during calibration mode, i.e., the target application can be run functionally 100% correct, with a slightly reduced clock frequency. Since the transition detection was asserted during the cycle, an increase of the high phase is triggered. On the right-hand side we see that for the next setting (increased length of the high phase), no transition detection is flagged, since the data signal settles before the low phase of the clock signal. As a consequence, the minimum clock period for this instruction type has been determined, it is equal to the shortest high phase of the employed calibration clock period that does not flag a transition detection (during the low phase). In order to make this approach work in hardware, a clock generation unit can be designed that provides very good matching between the high phase duration it generates for a certain delay setting when operating in calibration mode, and the corresponding clock period it generates for the same setting, when operating in normal run mode.

The schematic of a timing monitoring flip-flop as it is required for the presented on-chip calibration approach is shown in Figure 5.20. The design is based on a standard D flip-flop, and extended by additional circuitry to provide input transition detection in two different modes. Besides providing transition detection on the clock low phase (for M=1), the flip-flop

can also be operated in a second mode, which provides detection on the clock high phase (for M=0). This mode can be employed for classical timing error detection, since it monitors the window after the active clock edge of the flip-flop. Please note that the presented schematic of the flip-flop is only conceptual to illustrate its functionality, since it is possible to reduce the transistor overhead induced by the additional gates for an actual circuit implementation.

5.5 Measurement Results

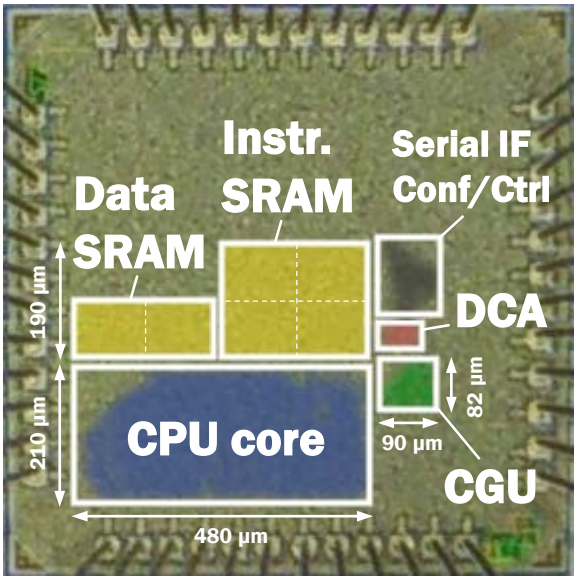
The DynOR system architecture, described in Section 5.3, was fabricated in a 28 nm FD-SOI regular-V_T CMOS technology, using 0.24 mm^2 of the complete 1.2 mm^2 die, with a complexity of about 316 k gate equivalents (GEs[9]). The chip micrograph together with the main features are shown in Figure 5.21.

Figure 5.22 details the distribution of all fabricated dies, regarding their maximum static CPU frequency (F_{max}). At a CPU supply voltage (V_{dd}-CPU) of 0.8 V slow dies operate at an F_{max} which is up to 14% lower than a typical die, while fast dies are up to 9% faster.

For a typical die at 1.0 V (V_{dd}-Main) the CGU provides a clock frequency between 365 MHz and 1906 MHz, with a period step granularity of 35 ps (with an average step-to-step accuracy of ±1.8 ps) over 64 settings. A normal operation range for the CPU at a corresponding supply voltage of 0.8 V (V_{dd}-CPU) employing DCA is a dynamic frequency range between 509 MHz and 1189 MHz, when executing the slowest and fastest instruction types, respectively.

In the following, we report the measured speedups and power savings, while varying two main parameters: V_{dd}-CPU and the application executed on the CPU. For each reported V_{dd}-CPU, the corresponding V_{dd}-Main is set to a voltage level that is 200 mV higher (exceptions are 0.6 V and 0.4 V for V_{dd}-CPU, where V_{dd}-Main is only 0.73 V and 0.45 V, respectively). An overview of the employed voltage pairs for the operation of DynOR is given in Table 5.2. This higher V_{dd}-Main is mainly to allow the SRAMs to provide enough access speed, and in some cases to enable the DCA to operate fast enough, as well.

[9] One GE is here defined as one minimum drive strength 2-input NAND gate (area: 0.49 μm^2).



Technology	UTBB FD-SOI 28 nm RVT
CPU	32b, 6-stage OpenRISC with dynamic clock
Die area	0.24 mm^2 (chip: 1.2 mm^2)
Memory	16 KB (IM) + 8 KB (DM)
Gate Equiv.	172 k (logic) + 144 k (SRAM)
V_{dd} range	0.40 V – 1.10 V (cpu min: 0.26 V)
Freq. range	24 – 1306 MHz
Power range	0.31 – 112 mW
Energy range	13 – 86 pJ/Op

Figure 5.21 – Die micrograph and chip features of DynOR.

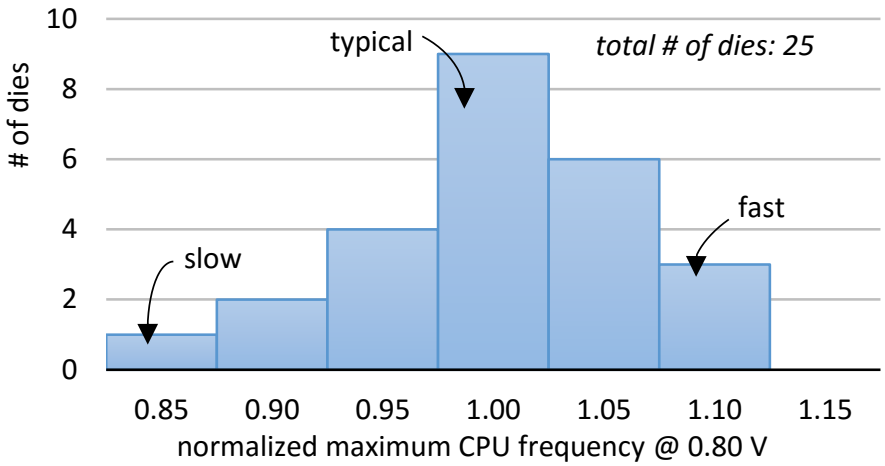


Figure 5.22 – Speed distribution of the 25 fabricated dies.

The two chosen applications are matrix multiplication (MM) and median calculation (MC), each with an execution length of about 10^{10} instructions. Our experiments showed that a program execution length of more than 10^9 instructions, i.e., a program execution time of more than 1 second at a clock frequency of 1 GHz, is necessary to achieve stable measurement results. If the number of executed instructions is too low, benchmarks will from time to time (at random) pass the final correctness test, at operating frequencies that are clear fail frequencies, when the number of benchmark iterations (execution length) is increased. Increasing the execution length beyond 10^{10} instructions, e.g., to 10^{11}–10^{12} instructions, however does not result in any further changes in behavior for our measurement setup.

Detailed measurements have been performed focusing on the two chosen applications (MM & MC), since they differ significantly in the flow and composition of instruction types they employ. The MM (using full-range 32-bit values) can be seen as a worst case stress test, since it is heavy in multiplication instructions, and hence excites the worst paths of the CPU with the highest rate. The MC is based on sorting, and in general, performs a more balanced set of instructions, regarding their path excitations.

Table 5.2 – Overview of supply voltage settings utilized for operation of the two voltage islands of DynOR, during measurements.

Voltage island	V_{dd}-CPU	V_{dd}-Main
Operating points	1.10 V	1.30 V
(voltage pairs)	0.80 V	1.00 V
	0.60 V	0.73 V
	0.40 V	0.45 V
Modules	CPU core	I+D SRAM, DCA, CGU

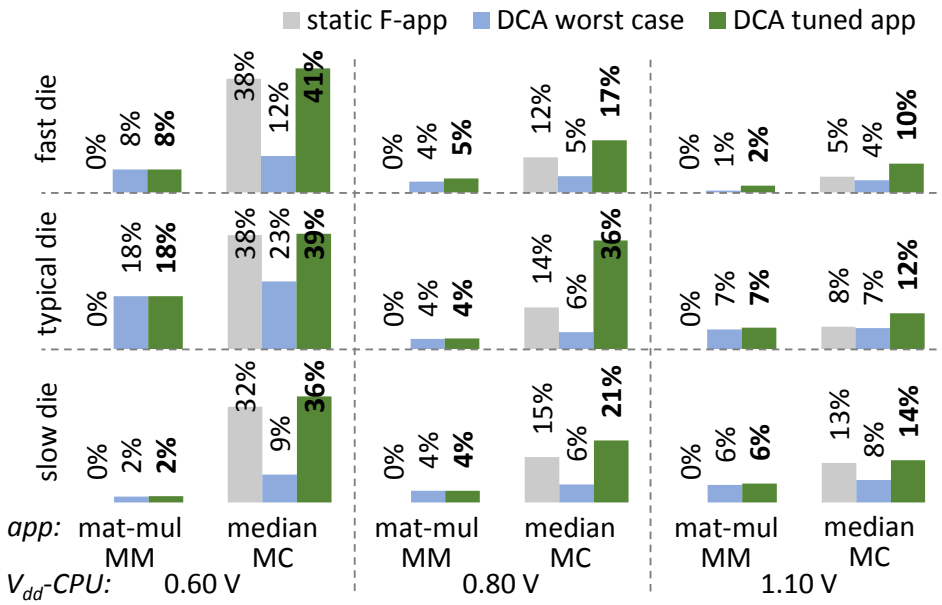


Figure 5.23 – Measured application speedup with respect to conventional static clocking with F_{max}, for different supply voltages, die types, and applications.

5.5.1 Speedup

Figure 5.23 reports the speedups (individually for a fast, typical, and slow die), which are all calculated relative to the baseline speed provided by F_{max} for a given V_{dd}-CPU and die at room temperature (25° C). For each parameter configuration, we report the three different speedups, achieved through three different modes of operation:

1. Over-clocking the CPU with a higher static clock frequency F_{app}, which the application allows. A speedup over F_{max} is possible, since we implement a path distribution that avoids the timing wall present in classical designs.[10] Note that although F_{app} is a static clock frequency, and does not require our DCA technique or any dynamic clock adjustment, for a classical design that is optimized towards a timing wall, the over-clocking gains provided by F_{app} are not available and the operating frequency is always limited to F_{max}, for any application. Moreover, since the MM always excites the worst paths, F_{app} equals F_{max} for this specific case (indicated as 0% speedup for MM in Figure 5.23).[11]

2. Utilizing DCA with a merged "worst case" period table (as described in Section 5.4.3), which shows the application-independent DCA gains.

3. Utilizing DCA with a period table that is tuned for the specific application, showing maximum achievable DCA gains.

[10] The path shaping approach that allows this application-dependent over-clocking is detailed in Section 5.1.3.1 and Section 5.2.1.

[11] Effectively, in our test setup we actually define F_{max} by the maximum frequency at which MM operates reliably. Hence, equality between F_{app} and F_{max} for MM is by definition.

For the MC on a typical die at 0.8 V, we observe that the implemented path distribution allows for an application-specific speedup of 14% with an increased static clock F_{app}. The proposed DCA approach allows to raise this speedup by an additional 22% (compared to F_{app}) to a total of 36% through the means of dynamically adjusting the clock frequency on an instruction basis. Even with a conservative "worst case" DCA table, which operates application-independent, the total speedup still amounts to 6% over F_{max}. The application specific speedup for MC enabled by DCA over all die types and different V_{dd}-CPU values is on average 25%, and can reach up to 41%. Furthermore, we see that for the MM a speedup is only enabled through DCA, and even though the application frequently excites the critical paths of the design, which works against our DCA approach, an average speedup of 6% can still be measured, which can help amortize any potential design overhead due to the applied path profile shaping. The average speedup provided by DCA over all operating conditions and applications is 19% for a typical die, and 14% for both the fast and the slow dies.

In general, we see that lower supply voltages allow for higher speedups, and show the strength of DCA, especially for critical/demanding applications such as MM, while at normal and higher supply voltages, the strength of DCA is shown for more balanced (MC type) applications, where significant additional speedups over F_{app} can be achieved.

5.5.2 Power Reduction

The presented application speedups can be traded off against power consumption, by means of supply voltage scaling. In Figure 5.24, we report the measured reductions in power consumption enabled by DCA at fixed throughput rates. To this end, we lower the CGU frequencies until the CPU performs the same number of Op/s as it does at fixed F_{max}. We then use the created voltage headroom to lower V_{dd}-CPU accordingly.

The power density of DynOR ranges between 13 µW/Mhz (at 0.4 V) and 87 µW/Mhz (at 1.1 V). Note that since the CPU processes very close to one (1) instruction per cycle, 1 µW/MHz here is equivalent to 1 pJ/Op. At a V_{dd}-CPU of 0.8 V we measure power savings of 15% (by decreasing V_{dd}-CPU by 50 mV) for the MC at 521 MOp/s, reducing the chip energy consumption from 50 pJ/Op down to 42 pJ/Op.

All power numbers include the total current drawn by the chip for both voltage islands (V_{dd}-CPU and V_{dd}-Main). The power consumption of the pad-ring (operating at 1.5 V) is not included/measured, however since DynOR has inherently low I/O activity (only for loading of programs over the serial interface) during its operation, and since it is internally clocked, this power can be neglected.

The relative distribution of power consumption for the different parts of the architecture stays surprisingly constant (changes of ±1–2%) over the full measured voltage range. As indicated in Figure 5.24, the dynamically clocked OpenRISC core consumes the majority of the chip's power with 69%, while the SRAM memories consume 25% and the CGU accounts for the

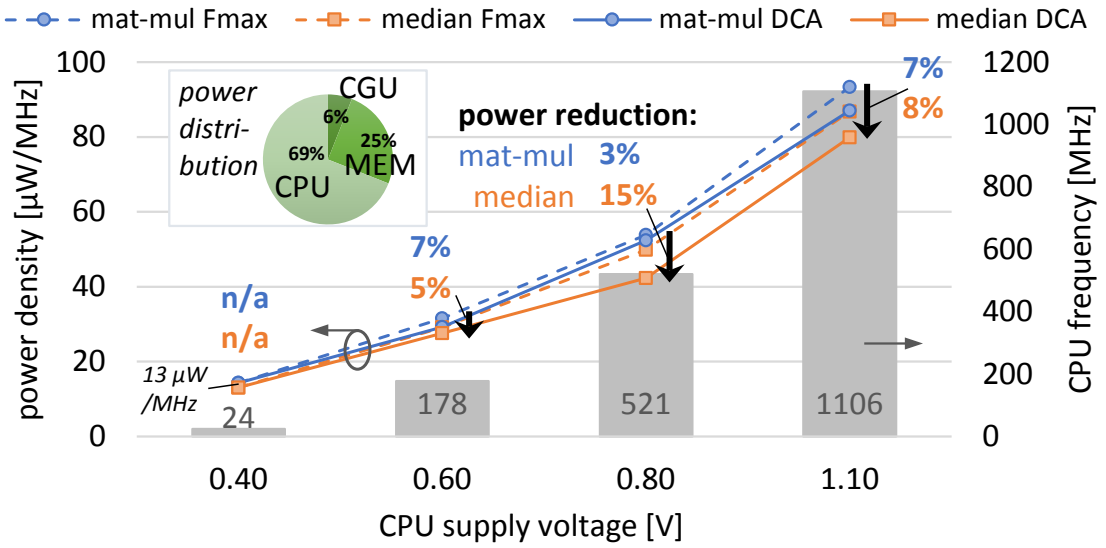


Figure 5.24 – Measured power reduction at iso-throughput for a typical die, due to supply voltage scaling enabled by DCA

remaining 6%. The power consumption of the DCA module could not be measured separately (it is accounted for with the SRAM consumption), but is expected to be negligible.

5.5.3 Comparison

Since, to our best knowledge, there are no other fabricated designs which employ a dynamic clocking scheme, a direct comparison with other recent works regarding the effectiveness of the microarchitectural design techniques proves to be difficult. Nevertheless, to put our silicon implementation into a broader context of embedded processing, regarding the overall performance numbers, Table 5.3 compares this work with other recent CPU implementations.

Table 5.3 – General comparison with recent state-of-the-art embedded CPU implementations from the literature.

	This work	[RPL+16]	[ACB+15]	[JKY+12]
Domain	**Embedded Processing**	Near-Sensor Processing	IoT / MCU Applications	General Purpose Processing
Technology	**FD-SOI 28nm RVT**	FD-SOI 28nm LVT	FD-SOI 28nm	32nm
CPU	**32b OpenRISC**	32b OpenRISC	32b Cortex-M4	32b IA-32 Pentium
# of cores	**1**	4	1	1
I$/D$/L2	**16K/8K/-**	4K/48K/64K	8K/8K/-	8K/8K/-
Voltage range	**0.40 – 1.10 V**	0.32 V – 1.15 V	0.34 – 1.10 V	0.28 – 1.20 V
Max. frequency	**1306 MHz**	825 MHz	834 MHz	915 MHz
Best performance	**1.3 GOPS**	3.3 GOPS	?	1.8 GOPS
Best power density	**13.0 µW/MHz @ 24 MOPS**	20.7 µW/MHz @ 4x40.5 MOPS	8.9 µW/MHz @ 45 MHz	170 µW/MHz @ 100 MHz
Features	**Dynamic Clock Adjustment**	Body Biasing, Shared TCDM	10-T SRAM	2x Superscalar, FPU, Bran. Pred.

6 Conclusions & Outlook

Low power consumption and high energy efficiency are the key requirements for embedded microprocessor designs of today, which typically operate as part of severely energy-constrained, battery-powered devices. In this thesis, a set of different microarchitectural design techniques for different processor architectures were presented that aim at supporting processor designers to meet these low-power requirements. We have seen that one of the main techniques which provide significant energy savings is supply voltage scaling, which however is accompanied by performance degradation of the processor core. To counter this performance degradation, a range of different low-overhead microarchitectural techniques have been presented, which effectively allow increased processing performance that can be traded for energy savings.

Microcontroller Architecture Enhancements for Cryptographic Applications

Especially in the context of cryptographic algorithms, it is well known that instruction set extensions (ISEs) can increase the performance of an application on a given microprocessor architecture through addition of datapath units. Considering algorithm-specific performance bottlenecks regarding memory consumption and execution time on resource constrained systems, we proposed dedicated ISEs based on lookup table integration and microcoded instructions using finite state machines for operand and address generation, which do not significantly add to the complexity of the processor datapath.

Looking at the application-specific case of cryptographic hash functions on a 16-bit MCU (PIC24), we assessed the potentials of these ISEs for all five SHA-3 final round candidates, representing a variety of different hash algorithms. We showed that our proposed classes of ISEs can provide significant speedup, hence enabling throughput increase and reduced energy consumption, in addition to substantial memory footprint reduction which can help to reduce leakage power. It was shown that most improvements were made by instructions that provided efficient generation of memory address patterns. Rather than adding datapath units to calculate complete functions, we find that additional multi-cycle instructions that handled complex (but regular) memory accesses as a result of constant permutation templates

are especially beneficial. Furthermore, moving constant lookup tables from data memory into the datapath of the processor turned out to be highly effective in reducing memory footprints at negligible core-area overhead. Minor extensions of existing computational units, combined with state driven microcoded instruction execution, to support rotation operations on larger bit-widths (64-bit) within the limits of the native 16-bit architecture, proved also to be especially helpful for significant speedup of most of the SHA-3 candidate algorithms.

In three out of five cases, the ISEs had no measurable impact on the core area of the microcontroller, in the two other cases, the overhead was limited to 10%, whereas the execution speed in terms of cycles improved by 2.72x on average over five candidates. In particular, we were able to improve the execution speed of Grøstl by more than a factor of 8x, and reduce its memory consumption by more than 70%. This has moved Grøstl from being one of the slowest implementations (about 3x slower than the other candidates) to the fastest implementation by some margin (1.75x faster than the next algorithm).

Microprocessor Design for Deeply Embedded Ultra-Low-Power Processing

A custom designed 16-bit processor core, named TamaRISC, for deeply embedded signal processing applications has been presented. We show with the TamaRISC architecture, which comprises a custom clean-slate ISA, as well as a corresponding microarchitecture with a 3-stage pipeline, that simplicity of the base ISA and core structure is key to low power consumption. At the same time, we observed that the integration of advanced operand addressing modes is beneficial for signal processing applications, with little costs in terms of hardware overhead.

The TamaRISC architecture in combination with an instruction set extension has been successfully applied to the application-specific use case of compressed sensing (CS). CS is a well-known universal data compression technique applied to sparse signals, used widely for sensing environment applications. Autonomous and portable devices, such as sensing platforms, however enforce ultra-low-power CS implementations, due to their limited energy resources. Therefore, we have proposed a subthreshold processing platform specifically optimized for CS, while still maintaining the flexibility and configurability of a processor based system. To this end, we have customized the instruction set architecture of a low-power baseline processor to exploit the specific operations of the CS algorithm. Specifically, we proposed a Compressed Sensing Accumulation (CSA) instruction that efficiently performs accumulation of sample data on randomized memory addresses within a defined sampling buffer. Moreover, our processing platform embeds the required data and instruction memories in the form of sub-V_T-capable standard-cell memories, which are essential for ULP operation. We show that our processing platform requires neither high computational effort nor excessive memory sizes compared to straight-forward implementations. Therefore, the platform is well suited to exploit subthreshold computing at low voltages and with very low leakage. Our system consumes only 30.6 nW for a case study of CS-based electrocardiogram (ECG) signal compression at an ECG sampling rate of 125 Hz. Our results show that the proposed processing

platform achieves 62x speed-up and 11.6x power savings with respect to an established CS implementation running on the baseline low-power processor.

Moreover, TamaRISC was employed in a new hybrid SIMD/MIMD multi-core architecture to significantly improve energy efficiency for on-node processing with processing requirements exceeding 1 MOPS. This thesis in particular presented multiple optimizations regarding the organization of the memory architecture of such multi-core systems, which allowed to reduce power consumption. The three key contributions that enable these improvements are as follows. First, we proposed a configurable data memory mapping mechanism allowing the partitioning of the address space into private and shared data sections which lowers memory access contention, thereby improving instruction throughput. Additionally, the mechanism allows for the efficient use of memory banks due to address remapping which enables power gating of parts of the physical memory that are currently not used, in turn reducing standby power. The mechanism retains a continuous logical address space and has been realized through the addition of a light-weight MMU to the processing cores. Second, in order to reduce the power spent on instruction fetches, which was found to be the main contributor in our multi-core architecture, instruction broadcast is introduced with the aim to operate the architecture in a SIMD mode, where multiple cores execute the same instructions in lockstep. Utilizing these two architecture enhancements, the total power consumption of the multi-core system could be reduced by 40%. Third, in order to improve the percentage of code that can be executed efficiently in SIMD mode, a code synchronization mechanism based on a hybrid hardware/software solution involving instruction set extensions was introduced. As a result of this improved SIMD operation, a further reduction by 50-64% in power consumption was achieved.

Dynamic Timing Margins in Embedded Microprocessors

To take a more application-independent view, we then studied an inherent suboptimality due to the worst-case design principle in synchronous circuits, and introduced the concept of dynamic timing margins. We showed that dynamic timing margins exist in microprocessor circuits, and that these margins are to a large extend state-dependent and that they are correlated to the sequences of instruction types which are executed within the processor pipeline. With the help of our proposed dynamic timing analysis method (and the corresponding developed tools), we accurately characterized these dynamic timing margins for all endpoints within a general-purpose embedded processor core.

This characterization data was then employed to perform high-level instruction set simulations in order to assess the impact of timing errors on application behavior under non-nominal operating conditions (i.e., over-scaled voltage or frequency). Modeling of timing errors on high-level simulators enables the rapid evaluation of application performance in terms of output quality, while over-scaling frequency and/or supply voltage. However, the evaluation of the impact and identification of the reliability bottlenecks heavily depends on the accuracy of the employed characterization model. We showed that in contrast to conventional methods,

based on purely random FI or static timing based FI, our proposed statistical fault injection approach provides a significant improvement in modeling accuracy for the important transition regions between fully error-free operation and total circuit failure. The proposed approach achieves these improvements through the use of instruction-based timing statistics, obtained by dynamic timing analysis at the gate level of a placed & routed processor. Moreover, our model provides the capability to incorporate dynamic physical variation effects, such as supply voltage noise, to further enhance the accuracy of the characterization of dynamic timing effects. As a result, trade-offs targeted by approximate computing applications could be analyzed for a general-purpose OpenRISC core, in particular the characterization of application output error versus achievable energy savings.

A Microprocessor with Cycle-By-Cycle Dynamic Clock Adjustment

One way of utilizing the uncovered dynamic timing margins in microprocessors is the presented over-scaling of frequency/voltage beyond nominal levels. However, our results have shown that the induced detrimental effects on output quality can be often significant, up to levels where not much energy gains can be achieved, if only a small degradation in output quality can be tolerated by an application. We hence devised a second approach to exploiting dynamic timing margins, which allows to operate the circuit fully error-free, while still exploiting the available dynamic margins to boost performance or improve energy efficiency.

To this end, we presented a fine-grained dynamic clock adjustment technique to trim dynamic timing margins by exploiting the different timing requirements of individual instructions in a microprocessor. To ensure considerable performance gains, the proposed approach starts with the design of the processor with a suitable timing profile which avoids a classical timing wall and is characterized by many short paths. The developed design flow for DCA based on dynamic timing analysis was used to characterize the improved timing upper bounds of each instruction. The flow illustrates in detail the steps that should be followed for the application of our approach to any other design. The application of the proposed approach to a RISC processor (6-stage OpenRISC) demonstrates for a custom developed instruction set simulator based environment that for popular embedded benchmarks on average the speed of the design could be increased by 38%, which could be translated to power savings of 24%.

Finally, we demonstrated the feasibility of the proposed DCA technique for a real hardware system with a fabricated silicon prototype of a full microprocessor system. Specifically, we introduced the first silicon implementation and microarchitecture of a 32-bit microprocessor which uses a dynamic clocking scheme to vary its clock period on a cycle-by-cycle basis, according to the executed instruction types. The effectiveness of this DCA approach is enabled through careful path reshaping of the logic in the processor pipeline. Our measurements in 28 nm FD-SOI show that by employing DCA, the throughput of the processor can be increased on average by 19% (up to 41%), while the power reduction reaches up to 15% for a typical die. Moreover, we demonstrated that consistent speedup is possible over varying die speeds, supply voltages, and applications, which can be differently suited for the proposed technique.

Outlook

Particularly the concepts of dynamic timing margins and dynamic clock adjustment, introduced in Chapters 4 and 5, pose a number of interesting open research questions.

DCA requires accurate calibration of the dynamic clock period values that are applied during operation of the processor. Moreover, these values need to be recalibrated when environmental conditions (e.g., temperature) change, and if large safety margins for these values should be avoided in order to minimize energy consumption. Consequently, the development of efficient on-chip calibration methods is essential for the implementation of DCA in a real hardware system. While a first approach to this problem based on timing monitoring flip-flops is presented in Section 5.4.3, further investigation into different circuit techniques that can provide a calibration mechanism for DCA and especially the evaluation of the incurred hardware overheads are of great interest.

A possible further improvement for DCA is enhancement of the types and number of criteria that are employed for adjustment of the clock period. While we show that DCA based on instruction types alone is already a promising approach, capturing most of the available dynamic timing margins, including aspects such as data dependencies or consideration of the state of the forwarding logic of a processor can potentially further enhance the efficiency of the DCA mechanism. DCA which is additionally based on data dependencies can especially be interesting for more signal processing heavy applications, which can exhibit very specific input data distributions. Ultimately, the imposed hardware complexity for implementation of these enhancements in decision criteria has to be carefully examined and compared to the potential gains in further speedup and evaluated in terms of energy efficiency of the complete system.

In Chapter 5 we showed the applicability and feasibility of DCA for a general purpose 32-bit embedded processor core. Further exploration of DCA can moreover investigate its applicability for structurally different processing architectures, such as SIMD architectures or DSPs, especially with different pipelining structures. Challenges arise here from both data path intensive pipeline stages (i.e., stages which mainly exhibit data dependent dynamic timing behavior) and deep pipelines, which both might limit the applicability of DCA and require new techniques to cope with these challenges. In the context of multi-core architectures, the synchronization and communication between different cores which reside in separate DCA domains, is another interesting open question requiring efficient solutions.

The DCA approach also provides new ways for optimization of applications on the level of the ISA and above. Specifically on compiler level, the optimization of instruction sequences for a DCA capable architecture is of great interest. During code emission the compiler has often a wide range of choices for expressing a certain high level construct in assembly language. The careful selection and potentially ordering of the emitted machine instructions could be performed with new and enhanced cost functions, which take into account the execution speed of the instruction sequence on the DCA architecture. This optimization would allow to exploit

the specific DCA characteristics of an ISA on a particular architecture implementation and tune the code emission for this architecture to improve throughput and/or energy efficiency.

In Chapter 4 we introduce the notion of exploiting dynamic timing margins in the context of approximate computing, with the goal to achieve a scaling of application output quality over a wide operating range by allowing timing errors to manifest in the processor architecture in a controlled way. While the concept leverages the dynamic timing margins of different instruction types, the applied clocking scheme is using a traditional static clock period. An open question is how to effectively combine DCA with approximation, for example by selectively overclocking certain instruction types beyond their failure limits, in order to further increase energy efficiency beyond the current limits of DCA. This combined approach would potentially allow to more accurately control the rate of failure for specific instructions and in turn the overall application or task that is being executed in an approximate context. Additionally, this would provide more degrees of freedom for tuning the operating point of the system, in order to achieve the desired graceful degradation of quality in operating scenarios where highest energy efficiency is required. A major step towards realizing such a combination of approximation with DCA on a classical RISC ISA would be to extend the ISA by introducing a new set of approximate instructions, specifically approximate versions of the standard ALU instructions.[1] These approximate instructions would allow the differentiation in the machine code between instructions which are required to be fully correct, since they handle the control part of the application (such as loop counters, memory addresses, etc.), and instructions which are allowed to be approximated, since they operate on the data of the application.[2] With this distinction it would now be possible to selectively apply the even more aggressive overclocking via DCA only to instructions which can tolerate timing errors and which allow approximation of their results. Consequently, with such an extended, approximate ISA and a corresponding DCA-enabled microarchitecture, it would be in principal possible to achieve significantly better control of the desired output quality (by systematically avoiding complete program failures and processor crashes), while still operating in the regime of timing errors.

[1] These approximate versions do not necessarily require separate hardware implementation (although this is an option to provide better failure behavior), but can be simply seen as identifiers or flags, which allow the programmer to communicate to the hardware via the ISA the intention regarding approximation of the result.

[2] Note that during compile time, at which more high level information of the code is still available, this required separation of control and data instructions can be performed with relative ease.

Bibliography

[ACB+15] F. Abouzeid, S. Clerc, C. Bottoni, B. Coeffic, J. M. Daveau, D. Croain, G. Gasiot, D. Soussan, and P. Roche. 28nm FD-SOI technology and design platform for sub-10pJ/cycle and SER-immune 32bits processors. In *European Solid-State Circuits Conference (ESSCIRC), ESSCIRC 2015 - 41st*, pages 108–111, Sept 2015.

[ACE16] ACE. CoSy compiler development system, 2016. http://www.ace.nl/compiler/cosy.html.

[ACH15] I. Andrea, C. Chrysostomou, and G. Hadjichristofi. Internet of Things: Security vulnerabilities and challenges. In *2015 IEEE Symposium on Computers and Communication (ISCC)*, pages 180–187, July 2015.

[AHK+11] M. Ashouei, J. Hulzink, M. Konijnenburg, J. Zhou, F. Duarte, A. Breeschoten, J. Huisken, J. Stuyt, H. de Groot, F. Barat, J. David, and J. Van Ginderdeuren. A voltage-scalable biomedical signal processor running ECG using 13pJ/cycle at 1MHz and 0.4V. In *2011 IEEE International Solid-State Circuits Conference*, pages 332–334, Feb 2011.

[AHMP10] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. SHA-3 proposal BLAKE. Submission to NIST (Round 3), 2010.

[AOD+08] K. Atasu, C. Ozturan, G. Dundar, O. Mencer, and W. Luk. CHIPS: Custom hardware instruction processor synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):528–541, March 2008.

[ARA11] M. A. Alam, K. Roy, and C. Augustine. Reliability- and process-variation aware design of integrated circuits - a broader perspective. In *Reliability Physics Symposium (IRPS), 2011 IEEE International*, pages 4A.1.1–4A.1.11, April 2011.

[ARLO12] O. C. Akgun, J. N. Rodrigues, Y. Leblebici, and V. Owall. High-level energy estimation in the sub-VT domain: Simulation and measurement of a cardiac event detector. *IEEE Transactions on Biomedical Circuits and Systems*, 6(1):15–27, Feb 2012.

[ARM16] ARM Limited. IoT subsystem for Cortex-M processors system diagram, 2016. http://www.arm.com/products/internet-of-things-solutions/iot-subsystem-for-cortex-m.php.

[Atm16] Atmel. SAM L21x Data Sheet, 2016. http://www.atmel.com/Images/Atmel-42385-SAM-L21-Datasheet.pdf.

[BD06] Eli Biham and Orr Dunkelman. A framework for iterative hash functions: HAIFA. In *In Proceedings of Second NIST Cryptographic Hash Workshop*, 2006.

[BDCB11] C. Banz, C. Dolar, F. Cholewa, and H. Blume. Instruction set extension for high throughput disparity estimation in stereo image processing. In *ASAP 2011 - 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 169–175, Sept 2011.

[BDPA11a] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Cryptographic sponge functions, 2011. http://sponge.noekeon.org.

[BDPA11b] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak reference. Submission to NIST (Round 3), 2011.

[BDS+11] D. Bull, S. Das, K. Shivashankar, G.S. Dasika, K. Flautner, and D. Blaauw. A power-efficient 32 bit ARM processor using timing-error detection and correction for transient-error tolerance and adaptation to PVT variation. *Solid-State Circuits, IEEE Journal of*, 46(1):18–31, Jan 2011.

[Be11] Daniel J. Bernstein and Tanja Lange (editors). eBACS: ECRYPT benchmarking of cryptographic systems, 2011. http://bench.cr.yp.to.

[BES06] F. R. Boyer, H. G. Epassa, and Y. Savaria. Embedded power-aware cycle by cycle variable speed processor. *IEE Proceedings - Computers and Digital Techniques*, 153(4):283–290, July 2006.

[BTK+09] K. A. Bowman, J. W. Tschanz, N. S. Kim, J. C. Lee, C. B. Wilkerson, S. L. L. Lu, T. Karnik, and V. K. De. Energy-efficient and metastability-immune resilient circuits for dynamic variation tolerance. *IEEE Journal of Solid-State Circuits*, 44(1):49–63, Jan 2009.

[BTL+11] K. A. Bowman, J. W. Tschanz, S. L. L. Lu, P. A. Aseron, M. M. Khellah, A. Raychowdhury, B. M. Geuskens, C. Tokunaga, C. B. Wilkerson, T. Karnik, and V. K. De. A 45 nm resilient microprocessor core for dynamic variation tolerance. *IEEE Journal of Solid-State Circuits*, 46(1):194–208, Jan 2011.

[BTT+11] K. A. Bowman, C. Tokunaga, J. W. Tschanz, A. Raychowdhury, M. M. Khellah, B. M. Geuskens, S. L. L. Lu, P. A. Aseron, T. Karnik, and V. K. De. All-digital circuit-level dynamic variation monitor for silicon debug and adaptive clock control. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 58(9):2017–2025, Sept 2011.

[BVH+13] D. Bol, J. De Vos, C. Hocquet, F. Botman, F. Durvaux, S. Boyd, D. Flandre, and J. D. Legat. SleepWalker: A 25-MHz 0.4-V sub-mm^2 7-uW/MHz microcontroller in 65-nm LP/GP CMOS for low-carbon wireless sensor nodes. *IEEE Journal of Solid-State Circuits*, 48(1):20–32, Jan 2013.

[CC06] B. H. Calhoun and A. P. Chandrakasan. Static noise margin variation for sub-threshold SRAM in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 41(7):1673–1679, July 2006.

[CCHL12] Chiung-An Chen, Shih-Lun Chen, Hong-Yi Huang, and Ching-Hsing Luo. An efficient micro control unit with a reconfigurable filter design for wireless body sensor networks (WBSNs). *Sensors*, 12(12):16211, 2012.

[CFH+05] L. Chang, D. M. Fried, J. Hergenrother, J. W. Sleight, R. H. Dennard, R. K. Montoye, L. Sekaric, S. J. McNab, A. W. Topol, C. D. Adams, K. W. Guarini, and W. Haensch. Stable SRAM cell design for the 32 nm node and beyond. In *Digest of Technical Papers. 2005 Symposium on VLSI Technology, 2005.*, pages 128–129, June 2005.

[CLC+09] S. L. Chen, H. Y. Lee, C. A. Chen, H. Y. Huang, and C. H. Luo. Wireless body sensor network with adaptive low-power design for biometrics and healthcare applications. *IEEE Systems Journal*, 3(4):398–409, Dec 2009.

[CMC+13] H. Cho, S. Mirkhani, C. Y. Cher, J. A. Abraham, and S. Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–10, May 2013.

[CP95] J. A. Clark and D. K. Pradhan. Fault injection: a method for validating computer-system dependability. *Computer*, 28(6):47–56, Jun 1995.

[CVC+13] V. K. Chippa, S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan. Approximate computing: An integrated hardware approach. In *2013 Asilomar Conference on Signals, Systems and Computers*, pages 111–117, Nov 2013.

[Dam90] Ivan Bjerre Damgård. *Advances in Cryptology — CRYPTO' 89 Proceedings*, chapter A Design Principle for Hash Functions, pages 416–427. Springer New York, New York, NY, 1990.

[DBC+13] A. Y. Dogan, R. Braojos, J. Constantin, G. Ansaloni, A. Burg, and D. Atienza. Synchronizing code execution on ultra-low-power embedded multi-channel signal analysis platforms. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 396–399, March 2013.

[DBW15] S. Das, D.M. Bull, and P.N. Whatmough. Error-resilient design techniques for reliable and dependable computing. *Device and Materials Reliability, IEEE Transactions on*, 15(1):24–34, March 2015.

[DCA+12] A. Y. Dogan, J. Constantin, D. Atienza, A. Burg, and L. Benini. Low-power processor architecture exploration for online biomedical signal analysis. *IET Circuits, Devices Systems*, 6(5):279–286, Sept 2012.

[DCR+12] A. Y. Dogan, J. Constantin, M. Ruggiero, A. Burg, and D. Atienza. Multi-core architecture design for ultra-low-power wearable health monitoring systems. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 988–993, March 2012.

[DFW+13] A. J. Drake, M. S. Floyd, R. L. Willaman, D. J. Hathaway, J. Hernandez, C. Soja, M. D. Tiner, G. D. Carpenter, and R. M. Senger. Single-cycle, pulse-shaped critical path monitor in the POWER7+ microprocessor. In *Low Power Electronics and Design (ISLPED), 2013 IEEE International Symposium on*, pages 193–198, Sept 2013.

[dKNS10] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 497–508, New York, NY, USA, 2010. ACM.

[Dog13] Ahmed Yasir Dogan. *Energy-Aware Processing Platform Exploration for Embedded Biosignal Analysis*. PhD thesis, EPFL-STI, Lausanne, 2013.

[Don06] D. L. Donoho. Compressed sensing. *IEEE Transactions on Information Theory*, 52(4):1289–1306, April 2006.

[DSD+07] A. Drake, R. Senger, H. Deogun, G. Carpenter, S. Ghiasi, T. Nguyen, N. James, M. Floyd, and V. Pokala. A distributed critical-path timing monitor for a 65nm high-performance microprocessor. In *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pages 398–399, Feb 2007.

[DTP+09] S. Das, C. Tokunaga, S. Pant, Wei-Hsiang Ma, S. Kalaiselvan, K. Lai, D.M. Bull, and D.T. Blaauw. RazorII: In situ error detection and correction for PVT and SER tolerance. *Solid-State Circuits, IEEE Journal of*, 44(1):32–48, Jan 2009.

[DWB+10] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge. Near-threshold computing: Reclaiming Moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, 98(2):253–266, Feb 2010.

[ECPS02] D. Estrin, D. Culler, K. Pister, and G. Sukhatme. Connecting the physical world with pervasive networks. *IEEE Pervasive Computing*, 1(1):59–69, Jan 2002.

[ECR11] ECRYPTII group. The SHA-3 Zoo, 2011. http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo.

[EDL+04] D. Ernst, S. Das, Seokwoo Lee, D. Blaauw, T. Austin, T. Mudge, Nam Sung Kim, and K. Flautner. Razor: circuit-level correction of timing errors for low-power operation. *Micro, IEEE*, 24(6):10–20, Nov 2004.

[EEM16] EEMBC. CoreMark - An EEMBC Benchmark, 2016. http://www.eembc.org/coremark/about.php.

[EHG+07] M. Eireiner, S. Henzler, G. Georgakos, J. Berthold, and D. Schmitt-Landsiedel. In-situ delay characterization and local supply voltage adjustment for compensation of local parametric variations. *Solid-State Circuits, IEEE Journal of*, 42(7):1583–1592, July 2007.

[EKD+03] D. Ernst, Nam Sung Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: a low-power pipeline based on circuit-level timing speculation. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 7–18, Dec 2003.

[FFK+13] M. Fojtik, D. Fick, Y. Kim, N. Pinckney, D.M. Harris, D. Blaauw, and D. Sylvester. Bubble Razor: Eliminating timing margins in an ARM Cortex-M3 processor in 45 nm CMOS using architecturally independent error detection and correction. *Solid-State Circuits, IEEE Journal of*, 48(1):66–81, Jan 2013.

[FKHO11] H. Fuketa, D. Kuroda, M. Hashimoto, and T. Onoye. An average-performance-oriented subthreshold processor self-timed by memory read completion. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 58(5):299–303, May 2011.

[FLS+10] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The Skein hash function family. Submission to NIST (Round 3), 2010.

[Fre16] FreeRTOS. The market leading, de-facto standard and cross platform real time operating system (RTOS)., 2016. http://www.freertos.org/.

[GGM+12] Frank Gürkaynak, Kris Gaj, Beat Muheim, Ekawat Homsirikamol, Christoph Keller, Marcin Rogawski, Hubert Kaeslin, and Jens-Peter Kaps. Lessons learned from designing a 65nm ASIC for evaluating third round SHA-3 candidates. In *Third SHA-3 Candidate Conference*, March 2012.

[GKM+11] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. Grøstl – a SHA-3 candidate. Submission to NIST (Round 3), 2011.

[GMKR10] S. Ghosh, D. Mohapatra, G. Karakonstantis, and K. Roy. Voltage scalable high-speed robust hybrid arithmetic units using adaptive clocking. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(9):1301–1309, Sept 2010.

[Gou10] Mourad Gouicem. Comparison of seven SHA-3 candidates software implementations on smart cards. Cryptology ePrint Archive, Report 2010/531, 2010. http://eprint.iacr.org/.

[GPC15] A. Gheolbănoiu, L. Petrică, and S. Coţofană. Hybrid adaptive clock management for FPGA processor acceleration. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1359–1364, March 2015.

[GR10] S. Ghosh and K. Roy. Parameter variation tolerance and error resiliency: New design paradigm for the nanoscale era. *Proceedings of the IEEE*, 98(10):1718–1751, Oct 2010.

[Gra15] A. Grau. Can you trust your fridge? *IEEE Spectrum*, 52(3):50–56, March 2015.

[GWZ13] M. Gorlatova, A. Wallwater, and G. Zussman. Networking low-power energy harvesting devices: Measurements and algorithms. *IEEE Transactions on Mobile Computing*, 12(9):1853–1865, Sept 2013.

[HB01] J. Hightower and G. Borriello. Location systems for ubiquitous computing. *Computer*, 34(8):57–66, Aug 2001.

[Hew15] Hewlett Packard. Internet of things research study: 2015 report, 2015. http://www8.hp.com/h20195/V2/GetPDF.aspx/4AA5-4759ENW.pdf.

[HGG+10] Luca Henzen, Pietro Gendotti, Patrice Guillet, Enrico Pargaetzi, Martin Zoller, and Frank K. Gürkaynak. *Cryptographic Hardware and Embedded Systems, CHES 2010: 12th International Workshop, Santa Barbara, USA, August 17-20, 2010. Proceedings*, chapter Developing a Hardware Evaluation Method for SHA-3 Candidates, pages 248–263. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[HKN+01] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr. A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language. In *Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol.20, no.11*, pages 1338–1354. IEEE, 2001.

[HLR+09] S. K. S. Hari, M. L. Li, P. Ramachandran, B. Choi, and S. V. Adve. mSWAT: Low-cost hardware fault detection and diagnosis for multicore systems. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 122–132, Dec 2009.

[HSL+09] S. Hanson, M. Seok, Y. S. Lin, Z. Foo, D. Kim, Y. Lee, N. Liu, D. Sylvester, and D. Blaauw. A low-voltage processor for sensing applications with picowatt standby mode. *IEEE Journal of Solid-State Circuits*, 44(4):1145–1155, April 2009.

[HZS+08] S. Hanson, B. Zhai, M. Seok, B. Cline, K. Zhou, M. Singhal, M. Minuth, J. Olson, L. Nazhandali, T. Austin, D. Sylvester, and D. Blaauw. Exploring variability and performance in a sub-200-mV processor. *IEEE Journal of Solid-State Circuits*, 43(4):881–891, April 2008.

[IBM14] IBM Security Systems. IBM X-Force threat intelligence quarterly, 4Q, 2014. http://www-01.ibm.com/common/ssi/cgi-bin/ssialias?infotype=SA&subtype=WH&htmlfid=WGL03062USEN.

[ISP+11] N. Ickes, Y. Sinangil, F. Pappalardo, E. Guidetti, and A. P. Chandrakasan. A 10 pJ/cycle ultra-low-voltage 32-bit microprocessor system-on-chip. In *ESSCIRC (ESSCIRC), 2011 Proceedings of the*, pages 159–162, Sept 2011.

[ISU+14] K. Ishibashi, N. Sugii, K. Usami, H. Amano, K. Kobayashi, Cong-Kha Pham, H. Makiyama, Y. Yamamoto, H. Shinohara, T. Iwamatsu, Y. Yamaguchi, H. Oda, T. Hasegawa, S. Okanishi, H. Yanagita, S. Kamohara, M. Kadoshima, K. Maekawa, T. Yamashita, Duc-Hung Le, T. Yomogita, M. Kudo, K. Kitamori, S. Kondo, and Y. Manzawa. A perpetuum mobile 32bit CPU with 13.4pJ/cycle, 0.14uA sleep current using reverse body bias assisted 65nm SOTB CMOS technology. In *COOL Chips XVII, 2014 IEEE*, pages 1–3, April 2014.

[JAB+09] C. H. Jan, M. Agostinelli, M. Buehler, Z. P. Chen, S. J. Choi, G. Curello, H. Deshpande, S. Gannavaram, W. Hafez, U. Jalan, M. Kang, P. Kolar, K. Komeyli, B. Landau, A. Lake, N. Lazo, S. H. Lee, T. Leo, J. Lin, N. Lindert, S. Ma, L. McGill, C. Meining, A. Paliwal, J. Park, K. Phoa, I. Post, N. Pradhan, M. Prince, A. Rahman, J. Rizk, L. Rockford, G. Sacks, A. Schmitz, H. Tashiro, C. Tsai, P. Vandervoorn, J. Xu, L. Yang, J. Y. Yeh, J. Yip, K. Zhang, Y. Zhang, and P. Bai. A 32nm SoC platform technology with 2nd generation high-k/metal gate transistors optimized for ultra low power, high performance, and high density product applications. In *2009 IEEE International Electron Devices Meeting (IEDM)*, pages 1–4, Dec 2009.

[JBW+09] S. C. Jocke, J. F. Bolus, S. N. Wooters, A. D. Jurik, A. C. Weaver, T. N. Blalock, and B. H. Calhoun. A 2.6-uW sub-threshold mixed-signal ECG SoC. In *2009 Symposium on VLSI Circuits*, pages 60–61, June 2009.

[JKY+12] S. Jain, S. Khare, S. Yada, V. Ambili, P. Salihundam, S. Ramani, S. Muthukumar, M. Srinivasan, A. Kumar, S. K. Gb, R. Ramanarayanan, V. Erraguntla, J. Howard, S. Vangal, S. Dighe, G. Ruhl, P. Aseron, H. Wilson, N. Borkar, V. De, and S. Borkar. A 280mV-to-1.2V wide-operating-range IA-32 processor in 32nm CMOS. In *2012 IEEE International Solid-State Circuits Conference*, pages 66–68, Feb 2012.

[Kae08] Hubert Kaeslin. *Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication.* Cambridge University Press, New York, NY, USA, 1st edition, 2008.

[KB+14] Stefan Kristiansson, Julius Baxter, et al. mor1kx - an OpenRISC processor IP core, 2014. https://github.com/openrisc/mor1kx.

[KC11] J. Kwong and A. P. Chandrakasan. An energy-efficient biomedical signal processing platform. *IEEE Journal of Solid-State Circuits*, 46(7):1742–1753, July 2011.

[KGA+09] D. Kammler, J. Guan, G. Ascheid, R. Leupers, and H. Meyr. A fast and flexible platform for fault injection and evaluation in Verilog-based simulations. In *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on*, pages 309–314, July 2009.

[KKF+14] Inyong Kwon, Seongjong Kim, D. Fick, Myungbo Kim, Yen-Po Chen, and D. Sylvester. Razor-Lite: A light-weight register for error detection by observing virtual supply rails. *Solid-State Circuits, IEEE Journal of*, 49(9):2054–2066, Sept 2014.

[KKK+06] J. H. Kim, Y. H. Kwak, M. Kim, S. W. Kim, and C. Kim. A 120-MHz-1.8-GHz CMOS DLL-based clock generator for dynamic frequency scaling. *IEEE Journal of Solid-State Circuits*, 41(9):2077–2082, Sept 2006.

[KKKS10] A.B. Kahng, Seokhyeong Kang, R. Kumar, and J. Sartori. Slack redistribution for graceful degradation under voltage overscaling. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 825–831, Jan 2010.

[KKKT12] U.R. Karpuzcu, K.B. Kolluru, Nam Sung Kim, and J. Torrellas. VARIUS-NTV: A microarchitectural model to capture the increased sensitivity of manycores to process variations at near-threshold voltages. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–11, June 2012.

[KKT13] U. R. Karpuzcu, N. S. Kim, and J. Torrellas. Coping with parametric variation at near-threshold voltages. *IEEE Micro*, 33(4):6–14, July 2013.

[KMKA11] K. Kanoun, H. Mamaghanian, N. Khaled, and D. Atienza. A real-time compressed sensing-based personal electrocardiogram monitoring system. In *2011 Design, Automation Test in Europe*, pages 1–6, March 2011.

[KRV99] V. Krishna, N. Ranganathan, and N. Vijaykrishnan. Energy efficient datapath synthesis using dynamic frequency clocking and multiple voltages. In *VLSI Design, 1999. Proceedings. Twelfth International Conference On*, pages 440–445, Jan 1999.

[KRV+08] J. Kwong, Y. Ramadass, N. Verma, M. Koesler, K. Huber, H. Moormann, and A. Chandrakasan. A 65nm sub-Vt microcontroller with integrated SRAM and switched-capacitor DC-DC converter. In *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, pages 318–616, Feb 2008.

[KSV+16] C. Kolias, A. Stavrou, J. Voas, I. Bojanova, and R. Kuhn. Learning Internet-of-Things security hands-on. *IEEE Security Privacy*, 14(1):37–46, Jan 2016.

[LDF+11] Charles R. Lefurgy, Alan J. Drake, Michael S. Floyd, Malcolm S. Allen-Ware, Bishop Brock, Jose A. Tierno, and John B. Carter. Active management of timing guardband to save energy in POWER7. In *Proceedings of the 44th Annual IEEE/ACM*

International Symposium on Microarchitecture, MICRO-44, pages 1–11, New York, NY, USA, 2011. ACM.

[LDF+13] C. R. Lefurgy, A. J. Drake, M. S. Floyd, M. S. Allen-Ware, B. Brock, J. A. Tierno, J. B. Carter, and R. W. Berry. Active guardband management in Power7+ to save energy and maintain reliability. *IEEE Micro*, 33(4):35–45, July 2013.

[LRC+11] D. Li, D. Rennie, P. Chuang, D. Nairn, and M. Sachdev. Design and analysis of metastable-hardened and soft-error tolerant high-performance, low-power flip-flops. In *Quality Electronic Design (ISQED), 2011 12th International Symposium on*, pages 1–8, March 2011.

[MAM+12] P. Meinerzhagen, O. Andersson, B. Mohammadi, Y. Sherazi, A. Burg, and J. N. Rodrigues. A 500 fW/bit 14 fJ/bit-access 4kb standard-cell based sub-VT memory in 65nm CMOS. In *ESSCIRC (ESSCIRC), 2012 Proceedings of the*, pages 321–324, Sept 2012.

[MAS+11] P. Meinerzhagen, O. Andersson, Y. Sherazi, A. Burg, and J. Rodrigues. Synthesis strategies for sub-vt systems. In *Circuit Theory and Design (ECCTD), 2011 20th European Conference on*, pages 552–555, Aug 2011.

[Mei14] Pascal Andreas Meinerzhagen. *Novel Approaches Toward Area- and Energy-Efficient Embedded Memories*. PhD thesis, EPFL-STI, Lausanne, 2014.

[Mer90] Ralph C. Merkle. *Advances in Cryptology — CRYPTO' 89 Proceedings*, chapter A Certified Digital Signature, pages 218–238. Springer New York, New York, NY, 1990.

[MFC13] P. Magarshack, P. Flatresse, and G. Cesana. UTBB FD-SOI: A process/design symbiosis for breakthrough energy-efficiency. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 952–957, March 2013.

[Mic09a] Microchip. 16-bit MCU and DSC programmer's reference manual, 2009. http://ww1.microchip.com/downloads/en/DeviceDoc/70157D.pdf.

[Mic09b] Microchip. PIC24HJXXXGPX06/X08/X10 Data Sheet, 2009. http://ww1.microchip.com/downloads/en/DeviceDoc/70175H.pdf.

[Mic11] Microchip. PIC24FJ128GA310 Data Sheet, 2011. http://ww1.microchip.com/downloads/en/DeviceDoc/39996f.pdf.

[Mic13] Microchip. Press Release: Microchip Technology Delivers 12 Billionth PIC Microcontroller to Leading Motor Manufacturer, Nidec Corporation, 2013. http://eecatalog.com/8bit/2013/05/16/microchip-technology-delivers-12-billionth-pic-microcontroller-to-leading-motor-manufacturer-nidec-corporation/.

[MKAV11] H. Mamaghanian, N. Khaled, D. Atienza, and P. Vandergheynst. Compressed sensing for real-time energy-efficient ECG compression on wireless body sensor nodes. *IEEE Transactions on Biomedical Engineering*, 58(9):2456–2466, Sept 2011.

[MMR05] S. Mukhopadhyay, H. Mahmoodi, and K. Roy. Modeling of failure probability and statistical design of SRAM array for yield enhancement in nanoscaled CMOS. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(12):1859–1880, Dec 2005.

[MSBR11] P. Meinerzhagen, S. M. Y. Sherazi, A. Burg, and J. N. Rodrigues. Benchmarking of standard-cell based memories in the sub-VT domain in 65-nm CMOS technology. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 1(2):173–182, June 2011.

[MSG+16] J. Myers, A. Savanth, R. Gaddh, D. Howard, P. Prabhat, and D. Flynn. A subthreshold ARM Cortex-M0+ subsystem in 65 nm CMOS for WSN applications with 14 power domains, 10T SRAM, and integrated voltage regulator. *IEEE Journal of Solid-State Circuits*, 51(1):31–44, Jan 2016.

[MYAZ15] R. Mahmoud, T. Yousuf, F. Aloul, and I. Zualkernan. Internet of things (IoT) security: Current status, challenges and prospective measures. In *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 336–341, Dec 2015.

[MYR+08] P. D. Mitcheson, E. M. Yeatman, G. K. Rao, A. S. Holmes, and T. C. Green. Energy harvesting from human and machine motion for wireless electronic devices. *Proceedings of the IEEE*, 96(9):1457–1486, Sept 2008.

[NAA+14] S. Natarajan, M. Agostinelli, S. Akbar, M. Bost, A. Bowonder, V. Chikarmane, S. Chouksey, A. Dasgupta, K. Fischer, Q. Fu, T. Ghani, M. Giles, S. Govindaraju, R. Grover, W. Han, D. Hanken, E. Haralson, M. Haran, M. Heckscher, R. Heussner, P. Jain, R. James, R. Jhaveri, I. Jin, H. Kam, E. Karl, C. Kenyon, M. Liu, Y. Luo, R. Mehandru, S. Morarka, L. Neiberg, P. Packan, A. Paliwal, C. Parker, P. Patel, R. Patel, C. Pelto, L. Pipes, P. Plekhanov, M. Prince, S. Rajamani, J. Sandford, B. Sell, S. Sivakumar, P. Smith, B. Song, K. Tone, T. Troeger, J. Wiedemer, M. Yang, and K. Zhang. A 14nm logic technology featuring 2nd-generation FinFET, air-gapped interconnects, self-aligned double patterning and a 0.0588 um2 SRAM cell size. In *2014 IEEE International Electron Devices Meeting*, pages 3.7.1–3.7.3, Dec 2014.

[NIS01] NIST. Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[NIS07] NIST. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. Federal Register, Vol.72, No.212, 2007. http://www.nist.gov/hash-competition.

[NIS15a] NIST. Secure hash standard (SHS). Federal Information Processing Standards Publication 180-4, 2015. http://dx.doi.org/10.6028/NIST.FIPS.180-4.

[NIS15b] NIST. SHA-3 standard: Permutation-based hash and extendable-output functions. Federal Information Processing Standards Publication 202, 2015. http://dx.doi.org/10.6028/NIST.FIPS.202.

[Nor96] E. Normand. Single event upset at ground level. *IEEE Transactions on Nuclear Science*, 43(6):2742–2750, Dec 1996.

[NV03] B. Nicolescu and R. Velazco. Detecting soft errors by a purely software approach: method, tools and experimental results. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 57–62 suppl., 2003.

[oHSC00] Harvard-MIT Division of Health Sciences and Technology Biomedical Engineering Center. MIT-BIH arrhythmia database directory, 2000. http://www.physionet.org/physiobank/database/mitdb.

[OK02] M. Orshansky and K. Keutzer. A general probabilistic framework for worst case timing analysis. In *Design Automation Conference, 2002. Proceedings. 39th*, pages 556–561, 2002.

[Ope14] OpenRISC Community. OpenRISC 1000 architecture manual (architecture version: 1.1), 2014. https://github.com/openrisc/doc/blob/master/openrisc-arch-1.1-rev0.pdf.

[Ope16] OpenRISC Community. OpenRISC Architecture, 2016. http://openrisc.io/architecture.

[PCC13] L. Petrica, V. Codreanu, and S. Cotofana. VASILE: A reconfigurable vector architecture for instruction level frequency scaling. In *Faible Tension Faible Consommation (FTFC), 2013 IEEE*, pages 1–4, June 2013.

[PH13] Peter Pessl and Michael Hutter. *Cryptographic Hardware and Embedded Systems - CHES 2013: 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*, chapter Pushing the Limits of SHA-3 Hardware Implementations to Fit on RFID, pages 126–141. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[PHB13] James Pallister, Simon J. Hollis, and Jeremy Bennett. BEEBS: open benchmarks for energy measurements on embedded platforms. *CoRR*, abs/1308.5174, 2013.

[Por11] Thomas Pornin. sphlib, update for the SHA-3 third round candidates, 2011. http://www.saphir2.com/sphlib.

[PPIM03] A. Peymandoust, L. Pozzi, P. Ienne, and G. De Micheli. Automatic instruction set extension and utilization for embedded processors. In *Application-Specific Systems, Architectures, and Processors, 2003. Proceedings. IEEE International Conference on*, pages 108–118, June 2003.

[PS05] J. A. Paradiso and T. Starner. Energy scavenging for mobile and wireless electronics. *IEEE Pervasive Computing*, 4(1):18–27, Jan 2005.

[QSC11] M. Qazi, M. Sinangil, and A. Chandrakasan. Challenges and directions for low-voltage SRAM. *IEEE Design Test of Computers*, 28(1):32–43, Jan 2011.

[RBG14] A. Rahimi, L. Benini, and R. K. Gupta. Application-adaptive guardbanding to mitigate static and dynamic variability. *IEEE Transactions on Computers*, 63(9):2160–2173, Sept 2014.

[RHLA14] Pradeep Ramachandran, Siva Kumar Sastry Hari, Manlap Li, and Sarita V. Adve. Hardware fault recovery for I/O intensive applications. *ACM Trans. Archit. Code Optim.*, 11(3):33:1–33:25, October 2014.

[RLKB11] A. Rahimi, I. Loi, M. R. Kakoee, and L. Benini. A fully-synthesizable single-cycle interconnection network for Shared-L1 processor clusters. In *2011 Design, Automation Test in Europe*, pages 1–6, March 2011.

[RMC05] Hongliang Ren, M. Q. H. Meng, and Xijun Chen. Physiological information acquisition through wireless biomedical sensor networks. In *2005 IEEE International Conference on Information Acquisition*, pages 6 pp.–, June 2005.

[RPL+16] D. Rossi, A. Pullini, I. Loi, M. Gautschi, F. K. Gurkaynak, A. Teman, J. Constantin, A. Burg, I. Miro-Panades, E. Beignè, F. Clermidy, F. Abouzeid, P. Flatresse, and L. Benini. 193 MOPS/mW @ 162 MOPS, 0.32V to 1.15V voltage range multi-core accelerator for energy efficient parallel and sequential digital processing. In *2016 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XIX)*, pages 1–3, April 2016.

[RTB+11] A. Raychowdhury, J. Tschanz, K. Bowman, S. L. Lu, P. Aseron, M. Khellah, B. Geuskens, C. Tokunaga, C. Wilkerson, T. Karnik, and V. De. Error detection and correction in microprocessor core and memory due to fast dynamic voltage droops. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 1(3):208–217, Sept 2011.

[RVB98] N. Ranganathan, N. Vijaykrishnan, and N. Bhavanishankar. A linear array processor with dynamic frequency clocking for image processing applications. *IEEE Transactions on Circuits and Systems for Video Technology*, 8(4):435–445, Aug 1998.

[SA14] Mohamed M. Sabry and David Atienza. Temperature-aware design and management for 3D multi-core architectures. *Foundations and Trends in Electronic Design Automation*, 8(2):117–197, 2014.

[Sat01] M. Satyanarayanan. Pervasive computing: vision and challenges. *IEEE Personal Communications*, 8(4):10–17, Aug 2001.

[SGT+08] S.R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas. VARIUS: A model of process variation and resulting timing errors for microarchitects. *Semiconductor Manufacturing, IEEE Transactions on*, 21(1):3–13, Feb 2008.

[SIHM06] M. Sugihara, T. Ishihara, K. Hashimoto, and M. Muroyama. A simulation-based soft error estimation methodology for computer systems. In *7th International Symposium on Quality Electronic Design (ISQED'06)*, pages 8 pp.–203, March 2006.

[Sil15] Silicon Labs. EFM32ZG110 Data Sheet, 2015. http://www.silabs.com/Support%20Documents/TechnicalDocs/EFM32ZG110.pdf.

[SKLS16] I. Shin, J. J. Kim, Y. S. Lin, and Y. Shin. One-cycle correction of timing errors in pipelines with standard clocked elements. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2):600–612, Feb 2016.

[SKS15] Insup Shin, Jae-Joon Kim, and Youngsoo Shin. Aggressive voltage scaling through fast correction of multiple errors with seamless pipeline operation. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 62(2):468–477, Feb 2015.

[SLM07] O. Schliebusch, R. Leupers, and H. Meyr. *Optimized ASIP Synthesis from Architecture Description Language Models*. Springer, 2007.

[STB97] V. Sieh, O. Tschache, and F. Balbach. VERIFY: evaluation of reliability using VHDL-models with embedded fault descriptions. In *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, pages 32–36, June 1997.

[STM16] STMicroelectronics. STM32L433xx Data Sheet, 2016. http://www.st.com/content/ccc/resource/technical/document/datasheet/f7/a0/fc/27/24/4e/4f/3f/DM00257192.pdf/files/DM00257192.pdf/jcr:content/translations/en.DM00257192.pdf.

[SVC15] O. Sahin, P. T. Varghese, and A. K. Coskun. Just enough is more: Achieving sustainable performance in mobile devices under thermal limitations. In *Computer-Aided Design (ICCAD), 2015 IEEE/ACM International Conference on*, pages 839–846, Nov 2015.

[Syn12] Synopsys. Automating the design and implementation of custom processors (Processor Designer, LISA 2.0), 2012. http://www.synopsys.com/Systems/BlockDesign/ProcessorDev/Pages/default.aspx.

[Syn16] Synopsys. DesignWare SHA-3 look aside core, 2016. https://www.synopsys.com/dw/ipdir.php?ds=security-crypto-sha-3-look-aside.

[Tar12] Target Compiler Technologies. IP designer, 2012. http://www.retarget.com/products/ipdesigner.php.

[TBW+09] J. Tschanz, K. Bowman, S. Walstra, M. Agostinelli, T. Karnik, and V. De. Tunable replica circuits and adaptive voltage-frequency techniques for dynamic voltage, temperature, and aging variation tolerance. In *2009 Symposium on VLSI Circuits*, pages 112–113, June 2009.

[Ten12] Tensilica. Xtensa customizable processors, 2012. http://www.tensilica.com/products/xtensa-customizable.htm.

[Tex16] Texas Instruments. MSP430FR573x Data Sheet, 2016. http://www.ti.com/lit/ds/slas639k/slas639k.pdf.

[TKD+07] J. Tschanz, N. S. Kim, S. Dighe, J. Howard, G. Ruhl, S. Vangal, S. Narendra, Y. Hoskote, H. Wilson, C. Lam, M. Shuman, C. Tokunaga, D. Somasekhar, S. Tang, D. Finan, T. Karnik, N. Borkar, N. Kurd, and V. De. Adaptive frequency and biasing techniques for tolerance to dynamic temperature-voltage variations and aging. In *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pages 292–604, Feb 2007.

[TKN+02] J.W. Tschanz, J.T. Kao, S.G. Narendra, R. Nair, D.A. Antoniadis, A.P. Chandrakasan, and V. De. Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage. *Solid-State Circuits, IEEE Journal of*, 37(11):1396–1402, Nov 2002.

[Uni16] University of Wisconsin-Madison. Computing with HTCondor, 2016. https://research.cs.wisc.edu/htcondor/.

[VC08] N. Verma and A. P. Chandrakasan. A 256 kb 65 nm 8T subthreshold SRAM employing sense-amplifier redundancy. *IEEE Journal of Solid-State Circuits*, 43(1):141–149, Jan 2008.

[VRK+06] C. Visweswariah, K. Ravindran, K. Kalafala, S. G. Walker, S. Narayan, D. K. Beece, J. Piaget, N. Venkateswaran, and J. G. Hemmett. First-order incremental block-based statistical timing analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(10):2170–2180, Oct 2006.

[WBG10] Christian Wenzel-Benner and Jens Gräf. XBX: eXternal Benchmarking eXtension for the SUPERCOP crypto benchmarking framework. In *Cryptographic Hardware and Embedded Systems - CHES*, volume 6225 of *LNCS*, pages 294–305. Springer, 2010.

[WBG11] Christian Wenzel-Benner and Jens Gräf. XBX: eXternal Benchmarking eXtension, 2011. http://xbx.das-labor.org/trac.

[WCC06] Alice Wang, Benton H. Calhoun, and Anantha P. Chandrakasan. *Sub-threshold Design for Ultra Low-Power Systems*. Springer, 2006.

[WCC13] Z. Wang, C. Chen, and A. Chattopadhyay. Fast reliability exploration for embedded processors via high-level fault injection. In *Quality Electronic Design (ISQED), 2013 14th International Symposium on*, pages 265–272, March 2013.

[Wei99] Mark Weiser. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):3–11, July 1999.

[WTLP14] J. Wei, A. Thomas, G. Li, and K. Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 375–382, June 2014.

[Wu11] Hongjun Wu. The hash function JH. Submission to NIST (Round 3), 2011.

[YY10] J. Yoo and H. J. Yoo. Emerging low energy Wearable Body Sensor Networks using patch sensors for continuous healthcare applications. In *2010 Annual International Conference of the IEEE Engineering in Medicine and Biology*, pages 6381–6384, Aug 2010.

[ZBSF04] Bo Zhai, D. Blaauw, D. Sylvester, and K. Flautner. Theoretical and practical limits of dynamic voltage scaling. In *Design Automation Conference, 2004. Proceedings. 41st*, pages 868–873, July 2004.

[ZG13] K. Zhao and L. Ge. A survey on the Internet of Things security. In *Computational Intelligence and Security (CIS), 2013 9th International Conference on*, pages 663–667, Dec 2013.

[ZHBS08] B. Zhai, S. Hanson, D. Blaauw, and D. Sylvester. A variation-tolerant sub-200 mV 6-T subthreshold SRAM. *IEEE Journal of Solid-State Circuits*, 43(10):2338–2348, Oct 2008.

[Zhu04] Q.K. Zhu. *Power Distribution Network Design for VLSI*. Wiley, 2004.

[ZKY+16] Y. Zhang, M. Khayatzadeh, K. Yang, M. Saligane, N. Pinckney, M. Alioto, D. Blaauw, and D. Sylvester. iRazor: 3-transistor current-based error detection and correction in an ARM Cortex-R4 processor. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 160–162, Jan 2016.

[ZLL+13] Jun Zhou, Xin Liu, Yat-Hei Lam, Chao Wang, Kah-Hyong Chang, Jingjing Lan, and Minkyu Je. HEPP: A new in-situ timing-error prediction and prevention technique for variation-tolerant ultra-low-voltage designs. In *Solid-State Circuits Conference (A-SSCC), 2013 IEEE Asian*, pages 129–132, Nov 2013.

[ZNO+06] B. Zhai, L. Nazhandali, J. Olson, A. Reeves, M. Minuth, R. Helfand, S. Pant, D. Blaauw, and T. Austin. A 2.60pJ/Inst subthreshold sensor processor for optimal energy efficiency. In *2006 Symposium on VLSI Circuits, 2006. Digest of Technical Papers.*, pages 154–155, 2006.

Glossary

#

6/8/10T (six/eight/ten)-transistor

A

ABI application binary interface
AD address
ADC analog-to-digital converter
AES Advanced Encryption Standard
ALU arithmetic logic unit
API application programming interface
ASCII American Standard Code for Information Interchange
ASIP application-specific instruction set processor

C

CAD computer-aided design
CDF cumulative distribution function
CG clock generator
CGU clock generation unit
CMOS complementary metal–oxide–semiconductor
CPI cycles per instruction
CPM critical-path timing monitor
CPU central processing unit
CS compressed sensing
CSA Compressed Sensing Accumulation
CTRL control

D

DC decode

DCA dynamic clock adjustment
DLL delay-locked loop
DM data memory
DMA direct memory access
DMEM data memory
DSP digital signal processor
DTA dynamic timing analysis
DVFS dynamic voltage frequency scaling
DWT discrete wavelet transform
DynOR dynamic OpenRISC

E

ECG electrocardiogram
EDA electronic design automation
EDS error-detection sequentials
EEG electroencephalogram
ELF executable and linkable format
EMV energy minimum voltage
ESD electrostatic discharge
EX execute

F

FD-SOI fully-depleted silicon-on-insulator
FE fetch
FFT fast Fourier transform
FI fault injection
FinFET fin field effect transistor
FIPS Federal Information Processing Standard
FPGA field-programmable gate array
FPU floating-point unit
FRAM ferroelectric random-access memory
FSM finite state machine

G

GE gate equivalent
GOPS billion/giga operations per second
GP general-purpose
GPIO general-purpose input/output

H

HDL hardware description language
HP high performance
HV high voltage
HW hardware

I

IC integrated circuit
IM instruction memory
IMEM instruction memory
I/O input/output
IoT internet of things
IP intellectual property
IPC instructions per cycle
IRQ interrupt request
ISA instruction set architecture
ISE instruction set extension
ISS instruction set simulator

L

LDO low-dropout regulator
LFSR linear feedback shift register
LISA Language for Instruction Set Architectures
LL low-leakage
LP low power
LSU load store unit
LUT lookup table
LVT low threshold voltage

M

Mbps megabit per second
MC median calculation
MCU microcontroller unit
ME memory
MEM memory
MIMD multiple instruction, multiple data

MM matrix multiplication
MMD multiscale morphological derivatives
MMU memory management unit
MOPS million/mega operations per second
MSB most significant bit
MSE mean squared error

N

NBTI negative-bias temperature instability
NIST National Institute of Standards and Technology
n-MOS n-type metal-oxide-semiconductor
NOP no-operation
NTV near-threshold voltage

O

OS operating system

P

PC program counter
PCB printed circuit board
PD Synopsys Processor Designer
PDF probability density function
PE processing element
PLL phase-locked loop
p-MOS p-type metal-oxide-semiconductor
PoFF point of first failure
P&R place and route
PRD percentage root-mean-square difference
PRNG pseudorandom number generator
PULP parallel ultra-low-power
PVT process, voltage, and temperature

R

RAM random-access memory
RAW read after write
RF register file
RFID radio-frequency identification

RISC reduced instruction set computing/computer
ROM read-only memory
RTL register transfer level
RTOS real-time operating system
RVT regular threshold voltage

S

SCM standard cell memory
SCVR switched capacitor voltage regulator
SDC Synopsys Design Constraints
SDF standard delay format
SEU single event upset
SF status flags
SHA-3 Secure Hash Algorithm-3
SIF serial interface
SIMD single instruction, multiple data
SoC system-on-chip
SP standard performance/power
SRAM static random-access memory
SSTA statistical static timing analysis
STA static timing analysis
STV super-threshold voltage
sub-V_T sub-threshold
SW software

T

TCDM tightly coupled data memory
TDDB time-dependent dielectric breakdown
TLB translation lookaside buffer
TSSI Test Systems Strategies, Inc.

U

ULP ultra-low-power
ULV ultra-low voltage

V

V_{dd} supply voltage

V_T	threshold voltage
VCD	value change dump
VHDL	VHSIC Hardware Description Language
VHSIC	very high speed integrated circuit
VLSI	very-large-scale integration

W

WAD	write address decoder
WB	write-back
WBSN	wireless body sensor network
WSN	wireless sensor node

List of Figures

List of Tables

List of Publications

Book Chapters

Jeremy Constantin, Ahmed Dogan, Oskar Andersson, Pascal Meinerzhagen, Joachim Rodrigues, David Atienza Alonso and Andreas Burg. *An Ultra-Low-Power Application-Specific Processor with Sub-V_T Memories for Compressed Sensing.* VLSI-SoC: From Algorithms to Circuits and System-on-Chip Design, pp. 88-106, IFIP Advances in Information and Communication Technology 418, Springer, 2013.

Conference Papers

Jeremy Constantin, Andrea Bonetti, Adam Teman, Christoph Müller, Lorenz Schmid and Andreas Burg. *DynOR: A 32-bit Microprocessor in 28 nm FD-SOI with Cycle-By-Cycle Dynamic Clock Adjustment.* 42nd IEEE European Solid-State Circuits Conference (ESSCIRC), Lausanne, Switzerland, September 12-15, 2016.

Jeremy Constantin, Zheng Wang, Georgios Karakonstantis, Anupam Chattopadhyay and Andreas Burg. *Statistical Fault Injection for Impact-Evaluation of Timing Errors on Application Performance.* 53rd ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, Texas, USA, June 5-9, 2016.

Reza Ghanaatian, Paul Whatmough, Jeremy Constantin, Adam Teman and Andreas Burg. *A Low-Power Correlator for Wakeup Receivers with Algorithm Pruning through Early Termination.* 2016 IEEE International Symposium on Circuits and Systems (ISCAS), Montreal, Canada, May 22-25, 2016. ‡

Davide Rossi, Antonio Pullini, Igor Loi, Michael Gautschi, Frank Gürkaynak, Adam Teman, Jeremy Constantin, Andreas Burg, Ivan Miro-Panades, Edith Beignè, Fabien Clermidy, Fady Abouzeid, Philippe Flatresse and Luca Benini. *193 MOPS/mW @ 162 MOPS, 0.32V to 1.15V Voltage Range Multi-Core Accelerator for Energy Efficient Parallel and Sequential Digital Processing.* 2016 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XIX), Yokohama, Japan, April 20-22, 2016.

Jeremy Constantin, Lai Wang, Georgios Karakonstantis, Anupam Chattopadhyay and Andreas Burg. *Exploiting Dynamic Timing Margins in Microprocessors for Frequency-Over-Scaling with Instruction-Based Clock Adjustment.* 2015 IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, March 9-13, 2015.

Ruben Braojos Lopez, Ivan Beretta, Jeremy Constantin, Andreas Burg and David Atienza Alonso. *A Wireless Body Sensor Network for Activity Monitoring with Low Transmission Overhead.* 12th IEEE International Conference on Embedded and Ubiquitous Computing (EUC), Milan, Italy, August 26-28, 2014. ‡

Ahmed Dogan, Ruben Braojos Lopez, Jeremy Constantin, Giovanni Ansaloni, Andreas Burg and David Atienza Alonso. *Synchronizing Code Execution on Ultra-Low-Power Embedded Multi-Channel Signal Analysis Platforms.* 2013 IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, March 18-22, 2013.

Nino Walenta, Andreas Burg, Jeremy Constantin, Nicolas Gisin, Olivier Guinnard, Raphael Houlmann, Charles Ci Wen Lim, Tommaso Lunghi and Hugo Zbinden. *1 Mbps Coherent One-Way QKD with Dense Wavelength Division Multiplexing and Hardware Key Distillation.* 2nd Annual Conference on Quantum Cryptography (QCRYPT), Singapore, September 10-14, 2012. ‡

Jeremy Constantin, Ahmed Dogan, Oskar Andersson, Pascal Meinerzhagen, Joachim Rodrigues, David Atienza Alonso and Andreas Burg. *TamaRISC-CS: An Ultra-Low-Power Application-Specific Processor for Compressed Sensing* [best paper nominee]. 20th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), Santa Cruz, California, USA, October 7-10, 2012.

Jeremy Constantin, Andreas Burg and Frank Gürkaynak. *Instruction Set Extensions for Cryptographic Hash Functions on a Microcontroller Architecture.* 23rd IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP), Delft, The Netherlands, July 9-11, 2012.

Ahmed Dogan, Jeremy Constantin, Martino Ruggiero, Andreas Burg and David Atienza Alonso. *Multi-Core Architecture Design for Ultra-Low-Power Wearable Health Monitoring Systems.* 2012 IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, Germany, March 12-16, 2012.

Journal Articles

Davide Rossi, Antonio Pullini, Igor Loi, Michael Gautschi, Frank Gürkaynak, Adam Teman, Jeremy Constantin, Andreas Burg, Ivan Miro-Panades, Edith Beignè, Fabien Clermidy, Fady Abouzeid, Philippe Flatresse and Luca Benini. *Energy Efficient Near-Sensor Computing: The PULPv2 Cluster* [submitted; invited article]. IEEE Micro, to appear.

Jeremy Constantin, Raphael Houlmann, Nicholas Preyss, Nino Walenta, Hugo Zbinden, Pascal Junod and Andreas Burg. *An FPGA-Based 4 Mbps Secret Key Distillation Engine for Quantum Key Distribution Systems.* Springer Journal of Signal Processing Systems (JSPS), November 18 (Online First), 2015. ‡

Nino Walenta, Andreas Burg, Dario Caselunghe, Jeremy Constantin, Nicolas Gisin, Olivier Guinnard, Raphael Houlmann, Pascal Junod, Boris Korzh, Natalia Kulesza, Matthieu Legré, Charles Ci Wen Lim, Tommaso Lunghi, Laurent Monat, Christopher Portmann, Mathilde Soucarros, Patrick Trinkler, Gregory Trolliet, Fabien Vannel and Hugo Zbinden. *A Fast and Versatile Quantum Key Distribution System with Hardware Key Distillation and Wavelength Multiplexing.* New Journal of Physics, vol. 16, January, 2014. ‡

Ahmed Dogan, Jeremy Constantin, David Atienza Alonso, Andreas Burg and Luca Benini. *Low-power Processor Architecture Exploration for Online Biomedical Signal Analysis.* IET Circuits, Devices & Systems, vol. 6, num. 5, pp. 279-286, September, 2012.

Technical Reports (E-Print)

Jeremy Constantin, Andreas Burg and Frank Gürkaynak. *Investigating the Potential of Custom Instruction Set Extensions for SHA-3 Candidates on a 16-bit Microcontroller Architecture.* Cryptology ePrint Archive, February, 2012.

Patent Applications

Paul Nicholas Whatmough and Jeremy Constantin (assignee: ARM Limited). *Correlation determination early termination.* WO2016012746A1, official filing date: June 3, 2015. ‡

Ahmed Yasir Dogan, Jeremy Constantin, Andreas Burg and David Atienza Alonso (assignee: EPFL). *Ultra-Low Power Multicore Architecture For Parallel Biomedical Signal Processing.* WO2013136259A3, official filing date: March 12, 2013.

PhD-Forums & Workshops

Jeremy Constantin, Andrea Bonetti, Adam Teman, Christoph Müller and Andreas Burg. *DynOR: A 6-stage OpenRISC Microprocessor with Dynamic Clock Adjustment in 28nm FD-SOI CMOS* [poster]. 3rd Workshop on Energy Efficient Electronics and Applications (WEEE), Turku, Finnland, September 10-12, 2015.

Andrea Bonetti, Jeremy Constantin, Adam Teman, and Andreas Burg. *Circuits and Techniques for Dynamic Timing Monitoring in Microprocessors* [poster]. Nano-Tera Annual Meeting, Bern, Switzerland, May 5, 2015. ‡

Jeremy Constantin and Andreas Burg. *Application-Specific Processor Design for Low-Complexity & Low-Power Embedded Systems* [poster]. Winter School on Design Technologies for Heterogeneous Embedded Systems (FETCH), Leysin, Switzerland, January 7-9, 2013.

‡ The contents of these publications do not form a part of this thesis.

Curriculum Vitae

Name: Jeremy Constantin

Address: EPFL, STI-IEL-TCL
Station 11
CH-1015 Lausanne
Switzerland

E-Mail: jeremy.constantin@epfl.ch
jeremy.constantin@web.de

Education

05/2012 – 10/2016 **Doctoral Program (PhD) in Electrical Engineering**
at the École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

09/2008 – 01/2011 **Master of Science in Electrical Engineering and Information Technology** (specialization: Microelectronics)
at the Swiss Federal Institute of Technology Zürich (ETH), Switzerland

09/2004 – 10/2007 **Bachelor of Science in Computational Engineering**
at the Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany

Professional Experience

04/2011 – 10/2016 **Scientific/Doctoral Assistant at Telecommunications Circuits Laboratory (TCL)**
at the École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
full-time position

09/2013 – 11/2013 **Summer Placement at ARM in Cambridge, UK**
full-time position in the R&D division of ARM headquarters
employment in two groups: IoT, low-power circuits

10/2008 – 01/2010	**Working Student at ST-Ericsson in Zürich, Switzerland** part-time position, group: mixed signal
04/2008 – 08/2008	**Post Graduate Internship at NXP Semiconductors / ST-NXP Wireless in Nürnberg, Germany** full-time position, group: hardware / validation
11/2007 – 03/2008	**Internship at NXP Semiconductors in Nürnberg, Germany** full-time position, group: hardware / validation