Research project performed at TIMA

Development of an Online Redundant MAC for integration in a RISC-V SoC

Fici Alessandro

Under the supervision of:

Mr. Edgar Ernesto Sanchez Sanchez, POLITO
Mr. Benabdenbi Mounir, TIMA
Mr. Ait Said Noureddine, TIMA

Abstract

This report is a summary of part of my internship work at TIMA laboratory, Grenoble, France. Energy management is becoming more and more important in today's IT field, especially when dealing with battery-powered devices. Internet of Things devices in particular, require an extreme power consumption optimization and surface area reduction, while retaining the performances acceptable. One way to achieve this target is the sacrifice of computing precision thus leveraging the approximate computing paradigm. This work is therefore based on a new Multiply and Accumulation unit, designed by TIMA laboratory members, that aims to reach the above mentioned target by exploiting redundant arithmetic and on-line operators. It makes possible to adapt the precision depending on the application. Since this unit is using a different numbering system than pure binary, it was needed an input/output conversion wrapper which allows to communicate with the external binary world and to be compared with existing binary solutions. Then a testing phase in a real environment to explore all the pros and cons of the unit had to be carried on, to know where to focus the possible improvements.

Key-words: RISC-V, Rocket Chip, SoC, ORMAC, Approximate Computing

Contents

List of Figures

Chapter 1

TIMA laboratory

TIMA[1](Technics of Informatics and microelectronics for integrated systems Architecture) is a French public research laboratory situated in Grenoble under the aegis of CNRS (Centre National de la Recherche Scientifique), Grenoble-INP (Institut Polytechnique de Grenoble), and UGA (Université Grenoble Alpes).
The laboratory is composed of five research teams which work on microelectronics:

- **AMfoRS**: Architectures and Methods for Resilient Systems
- **CDSI** : Circuits, Devices and System Integration.
- **RIS** : Robust Integrated Systems.
- **RMS** : Reliable Mixed-signal Systems.
- **SLS** : System Level Synthesis

[1]TIMA laboratory website: `http://tima.univ-grenoble-alpes.fr`

Chapter 2

Problem Definition

2.1 Project Introduction

Hardware Approximate Computing is one of the main research path of the **AMfoRS** team in TIMA laboratory. My internship lies exactly in this field, aiming to exploit the possibilities given by a new HW unit developed by TIMA laboratory members and integrate it inside **RISC-V Rocket Chip SoC**, allowing future research to expand and explore further the subject. The internship was supervised by **Dr. Mounir Benabdenbi**, Associate Professor in Grenoble INP and **Edgar Ernesto Sanchez Sanchez**, Associate Professor in Politecnico di Torino.

2.2 Project Problematic

Approximate computing is an active research field, with constant improvements and new ideas emerging. However the hardware based Approximate Computing is still a relatively new field. The hardware unit my project is focused on lacked of proper testing, especially in terms of realistic application. So the idea was to exploit the great level of customization that Rocket Chip allows, integrating the unit inside the SoC as a peripheral, and finally using software application to finalize the testing process. It is worth noting that the whole unit was written in VHDL but Rocket Chip accepts peripherals only written in Verilog; so a complete translation was needed.

2.3 Project Objectives

- Getting familiar with the RISC-V and Rocket Chip environment
- Understanding the Approximate Computing state of art, with its different techniques, advantages, drawbacks and future challenges;
- Understanding the ORMAC (On-Line Redundant Multiply and Accumulate) unit and its VHDL version;
- Improving the VHDL source code, developing an input/output conversion wrapper, testing it against a classical MAC unit and translating it to Verilog;
- Integrating the ORMAC unit, now written in Verilog, in Rocket Chip, testing it using a software compiled for our target environment.

Chapter 3

RISC-V and Rocket Chip overview

This chapter will introduce some information useful to understand the target platform and the reason behind the choice of using it. The first section will briefly present the RISC-V ISA [1], pointing out its design principles and what makes it different from the other existing ISA, describing also the instruction format and the extension mechanism. The second section will analyze the Rocket Chip SoC Generator[5]

3.1 RISC-V

RISC-V is an open-source hardware instruction set architecture (ISA) based on reduced instruction set computer(RISC) principles. As written inside the official Berkeley technical report [1], RISC-V is structured as a small ISA with a variety of optional extension. The base ISA is very simple, making it a good choice for research and education, but also complete enough to be used in inexpensive low power embedded devices. The various optional extensions can form a more powerful ISA for general purpose and high performance computing.
A standard base integer ISA is defined, on 32 ("RV32I") and 64 ("RV64I") bit. The base integer instruction set has been designed in order to include a small number of instructions and reduce the cost in terms of hardware and complexity for a minimal implementation. To this base ISA it is possible to add several standard extension, ranging from the multiplication and division units to the vector-based operations unit. It is also possible to define its own non-standard extension, to fit almost any kind of requirement.
All standard extensions are supported by GCC("GNU C Compiler") and a RISC-V Linux kernel version is officially supported by the Linux foundation. The RISC-V capability to run Linux has definitely incremented the interest in it in the industrial field. [6]

3.1.1 Register set

RISC-V has 32 integer registers, and, if the floating-point extension ("F") is included, 32 floating-point registers. Except for memory access instructions, only registers are addressed by instruction (load-store architecture). The first integer register ("x0" or "Zero") is a always zero register and the remainder are general purpose registers. A read from the zero register always provides 0, a write has no effect. Complete register set is showed in 3.1.

Register name	Symbolic name	Description
32 integer registers		
x0	Zero	Always Zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary
x6-7	t1-t2	Temporary
x8	s0/fp	Saved register Frame pointer
x9	s1	Saved register
x10-11	a0-1	Function argument/ return value
x12-17	a2-7	Function argument
x18-27	s2-11	Saved register
x28-31	t3-6	Temporary
32 floating-point extension registers		
f0-7	ft0-7	Floating-point temporaries
f8-9	fs0-1	Floating-point saved registers
f10-11	fa0-1	Floating-point arguments/ return values
f12-17	fa2-7	Floating-point arguments
f18-27	fs2-11	Floating-point saved registers
f28-31	ft8-11	Floating point temporaries

Table 3.1: Register sets

3.1.2 Instructions format

RISC-V base integer ISA is a simple instruction set, comprising just 47 instruction, but it is complete enough to form a compiler target and satisfy the basic requirements of modern operating systems and runtimes.
Four basic instruction format exist: *R-type*, *I-type*, *S-type*, *U-type*. In order to simplify the decoding hardware both the sources (*rs1* and *rs2*) and the destination (*rd*) register fields are kept in the same position for all the formats. For the same reason the immediates are always placed starting from the leftmost significant bit and the sign position is always the bit 31 of the instruction.

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper Immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

Figure 3.1: RISC-V Instructions format

3.1.3 ISA extensions

One of the design goals of RISC-V is to keep the integer base ISA as simple as possible providing however full support for several standard extensions. Following the list of standard and frozen(the instructions will not change in the future) extensions:

- **M** : Standard Extension for Integer Multiplication and Division
- **A** : Standard Extension for Atomic Instructions
- **F** : Standard Extension for Single-Precision Floating-Point
- **D** : Standard Extension for Double-Precision Floating-Point
- **Q** : Standard Extension for Quad-Precision Floating-Point
- **C** : Standard Extension for Compressed Instructions

Often to indicate the "IMAFD" set the letter "G" is used, so the resulting ISAs are called *RV32G* for the 32-bit version and *RV64G* for the 64-bit one.
As mentioned before is natively possible to extend the ISA adding custom extension not included in the original ISA. The simplest method to add new instruction is to leverage two of the four custom-reserved opcodes, *custom-0* and *custom-1*, which are guaranteed to not be used in future language official extensions, while *custom-2* and *custom-3* will be probably used for the 128 bit ISA extension.

3.2 Rocket Chip

Rocket Chip is an open-source System-on-Chip design generator based on the RISC-V ISA designed by the Berkeley Architecture Research (BAR) group of the university of California Berkeley (UCB). It

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80b

Figure 3.2: RISC-V opcode map showing the custom fields [1]

emits synthesizable RTL leveraging the *Chisel* hardware construction language (a dialect of the Scala programming language) to interconnect cores, caches and peripherals for creating a full integrated SoC, generating then Verilog code compatible with FPGA and ASIC design tools. Rocket Chip generates general purpose processor cores, providing both a five-stage pipeline in-order core generator (Rocket Core) and an out-of-order core generator(BOOM). Moreover, Rocket Chip supports the integration of custom accelerators leveraging the instruction set extension provided by RISC-V and custom peripherals, written in Verilog and embedded using the Blackbox feature of Chisel.
By using **Verilator** [1], from Chisel it is also possible to generate a cycle-accurate RTL simulator implemented in C++, that converts the Verilog code produced by the Chisel (Scala) compiler to C++. This emulator is functionally equivalent to Verilog simulator but definitely faster and it can be used to simulate an entire Rocket Chip instance.

3.2.1 Rocket Chip Generator

The Rocket Chip generator consists of a collection of parameterized chip-building libraries which can be used to generate different SoC variants. The plug-and-play environment allows to swap-in and out design components simply by changing configuration files, without touching the hardware source code. 3.3 shows an instance of Rocket Chip. Two tiles are attached to a 4-bank L2 cache that is itself connected to the external I/O and memory system with an AXI interconnect.
Tile 1 is composed by an out-of-order BOOM core with an FPU, L1 data cache and L1 instruction cache and a RoCC accelerator.
Tile 2 features an in-order Rocket Core with FPU, L1 instruction and data cache with different parameters with respect to Tile 1 and a different RoCC accelerator.
Following a summary of the components and their capabilities:

- **Core**: The actual CPU core generator. The generated CPU can be an in-order Rocket Core or an out-of-order BOOM superscalar core. Both of which can include an optional FPU, configurable functional unit pipelines, and customizable branch predictors.
- **Caches**: Cache and TLB (Translation Lookaside Buffer) generators with configurable sizes, associativities, and replacements policies.
- **RoCC**: The Rocket Custom Coprocessor interface, allows to build an accelerator with its own instructions directly integrated inside the main CPU pipeline.
- **Tile**: A tile-generator template for cache-coherent tiles. The number and type of cores and accelerators are configurable.
- **TileLink**: A generator for cache coherency networks and cache controllers. The number of tiles, the coherence policy, the presence of shared backing storage can be configured.

[1] Verilator website: https://www.veripool.org/wiki/verilator

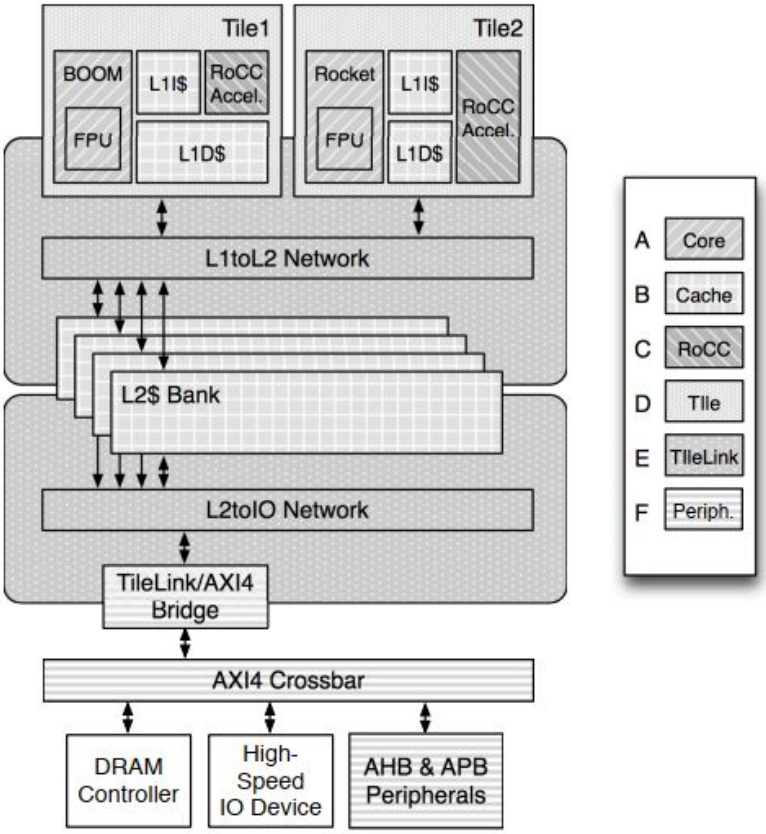


Figure 3.3: Rocket Chip

- **Peripherals**: Generators for converters and controllers as well as peripherals needed to implement the SoC on a FPGA or an ASIC. Our ORMAC unit will be integrated as a peripheral.

3.2.2 Rocket Core

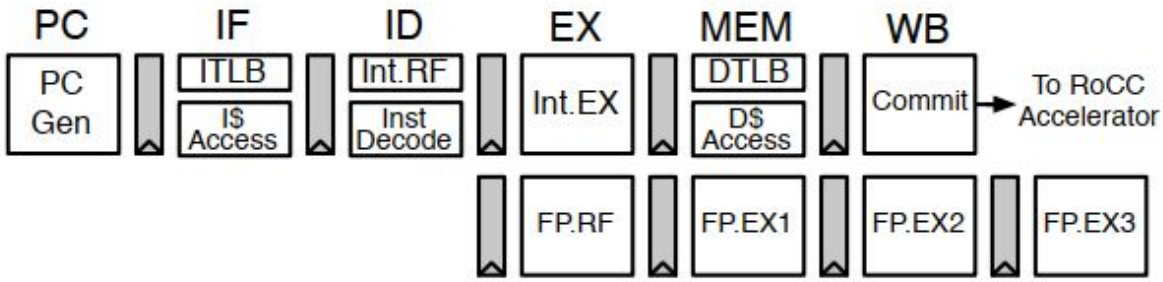


Figure 3.4: Rocket core pipeline

Rocket is a 5-stage in-order scalar core generator that implements the *RV32G* and *RV64G* ISAs [2]. It has an MMU supporting page-based virtual memory, non-blocking data cache and branch prediction unit. Branch prediction is configurable and it is provided by a branch target buffer (BTB),

[2]Rocket Core Github repository: `https://github.com/chipsalliance/rocket-chip`

Rocket Core	Parameters
Architecture	32 or 64 bits
ISA	Customizable with Standard/non-Standard extensions
Pipeline	5 stages: Fetch, Decode, Execute, Memory, Commit
FPU	Add/Remove or Modify
Multiplication and Division Unit	Add/Remove or Modify
Co-Processor (RoCC)	Add/Remove or Modify
L1 instruction cache	Parametrizable size and architecture
L1 data cache	Parametrizable size and architecture
Traps / Interrupts	Synchronous and asynchronous interrupts, parametrizable interrupt vector size
Virtual Memory	Parametrizable TLB Block
Modes	User, Supervisor and Machine modes are implemented

Table 3.2: Main characteristics of Rocket-Core in a default Rocket Chip configuration.

branch history table (BHT), and a return address stack (RAS). The floating point unit makes use of the Berkeley's Chisel FPU implementation [3].
The *Rocket Custom Coprocessor Interface (RoCC)* makes the communication between the Rocket CPU and the attached coprocessors easier. Through the RoCC unit various coprocessors have been implementend including crypto units (e.g., SHA3) and vector processing units. The RoCC interface accepts coprocessor commands generated by committed instructions executed by the Rocket core. The RoCC interface also allows the attached coprocessor to share the Rocket core's data cache and page table walker, and provides a facility for the coprocessor to interrupt the core, allowing the coprocessor to participate in a page-based virtual memory system.

[3]Hardfloat repository: `https://github.com/ucb-bar/berkeley-hardfloat`

Chapter 4

Approximate Computing

This chapter gives a brief and non-exhaustive overview of the various Approximate Computing techniques on different hierarchical levels based on Moons[7], Mittal[8], Xu[9] survey works and Noureddine Ait Said[10] report.
Each technique is evaluated and one or more existing research paper are reported, commented and analyzed while trying to enhance their main features and contributions to the research field.

4.1 Introduction

Even if the significant advances in semiconductor technologies, processor architectures, and low-power design techniques have led to huge improvements in terms of computational power and energy consumption, the global demand for power and storage is still increasing.
New rising techniques such as Data Mining, IoT, Machine Learning commonly known as **RMS** (Recognition, Mining, Synthesis) as well as Social Networking, require growing quantity of raw data and power to extract information.
This increase in computational and storage demands can come at high economic and environmental impact costs. Table 4.1 shows a summary of computation and data storage related global power consumption values[11].

The solutions proposed in the past to overcome these issue such as technology scaling and architecture improvements are not sufficient anymore. Fortunately many of the **RMS** applications, that currently account for a significant portion of computational resources around the world, are based on Neural Networks which allow a certain degree of error-tolerance. For instance in image or speech recognition, analog signals are converted into strings and digital images. Since this translation is often based on clustering and estimations, small differences or deviation w.r.t. a more precise computation usually do not affect the final result. Another example could be found in multimedia applications such as image compression or video encoding, where a slightly lower quality in terms of resolution could hardly be seen by a human eye but can save a lot of energy.
So the main idea behind the approximate computing paradigm is to exploit the inner error tolerance of such applications by admitting a certain degree of "acceptable errors" in the computation, in order to grant significant gains in terms of power consumption. Also, reduction of circuits surface area could be achieved, especially if the target are embedded or IoT devices. **Approximate computing** tries to leverage the gap between the accuracy required by the users or applications and the one provided by the computing system, for achieving an optimized result under different constraints.
It is worth noting that even though the applications listed above are error tolerant, the computation

EU consumption	
Consumption (TWh)	**Reporting Year**
18.3	2000
41.3	2005
56	2007
72.5	2010
104	2020
US consumption	
Consumption (TWh)	**Reporting Year**
91	2013
140	2020
Global consumption	
Consumption (TWh)	**Reporting Year**
216	2007
269	2012

Table 4.1: Energy Consumption estimation and projection in TWh from a European, American and Global Perspective

Approximate technique		
Software		Loop perforation, Thread/Task fusion, Memoization, Approximate programming language/compilers
Hardware	**Architecture**	Approximate storage, ISA extentions, Accelerators
	Circuit	Inexact arithmetic circuits, Voltage overscaling, Precision Scaling

Table 4.2: Approximate computing techinques divded by kernel type. Some techniques will be further covered.

and the output quality of an approximate computing technique should be **dynamic** and **tunable** in order to be able to lower accuracy only if it is needed and allowed, for avoiding unreasonable quality loss or catastrophic errors.

4.2 Approximate Computing Techniques

4.2.1 Approximate Kernels

In the approximate computing field the term kernel denotes the main support that handles the approximate application as well as those techniques used to realize approximation. It can be either a pure software component (e.g. a program or part of it, a thread, a process) or a pure hardware component (e.g. an approximate adder/multiplier circuit). It can be even a combination of both hardware and software efficiently communicating through specific protocols. Table 4.2 shows a summary of some approximate computing techniques.

4.2.2 Error Resilience Identification and Quality Management

Even for error-tolerant applications there exist error-sensitive parts where applying approximation techniques can lead to fatal errors such as segmentation faults due to wrong memory accesses or

completely wrong results. It is then fundamental to formally identify parts of a program or a system where an approximation is feasible and worthy. These parts where approximation can be profitably applied are identified as **Resilient**. Therefore this property is called **Error Resilience** and can be generally defined as the characteristic of an application either hardware or software to produce "acceptable" output (output that can be considered correct under certain constraints and bounds) despite its input data have a certain degree of noise and/or its constituent computation being performed with errors[12].

Resilience identification and Characterization

Even though an automated and unified method to identify application resilience does not exist yet several approaches have been studied. Generally resilience identification is application-dependent and it is performed offline i.e. during the design phase, when the program is not running. Various computational approximation are monitored, to verify their impact on a specific application. However it exists the possibility to identify the resilience online i.e. at run time through dynamic quality management.
Chippa et al.[13] propose an application resilience characterization framework called **ARC** (Application Resilience Characterization) that can be used to quantitatively evaluate the resilience of applications through two major steps: identification of potentially resilient computations and characterization of these computations by using approximation models. The ARC framework inputs are the application program to be tested, a representative data set and a quality evaluation function. The quality evaluation function is application specific and has to be provided by the user. It processes the output of the application program by eventually providing a numerical value as quality evaluation. The general approach taken in both steps of the ARC framework is to inject random errors or controlled approximations into specific computations during the application execution, and check the resulting application behavior.

- **Resilience Identification**: As stated above, even the RMS applications, usually considered the most resilient, contain both resilient and sensitive computations. Of course the approximate computing techniques should be applied to resilient computations only while avoiding the sensitive ones. Thus potentially resilient kernels are identified. Then the program is run over the input data set and the ARC framework adds random errors to the program variables of the probably resilient parts. If the application program crashes, hangs or provides an output not meeting the quality criterion the kernel under analysis is marked as sensitive, otherwise it is marked as potentially resilient.

- **Resilience Characterization**: Once the potentially resilient kernels are identified, the second step of the ARC framework is to characterize their resilience to analyze whether an approximate computing technique is profitably applicable or not. The resilience is then quantified using generic approximation attributes such as error probability, magnitude and predictability of the introduced errors, and the output quality impact of one or more approximate techniques. Finally the quality evaluation function provided by the user is applied to generate a profile that characterizes the application output depending on the approximation model used (e.g. approximation of arithmetic operations, of data representation or algorithmic level approximations).

Quality management

Through this method the intermediate computation quality is regularly evaluated at run time and it is decided whether certain kernels can be approximated or not.

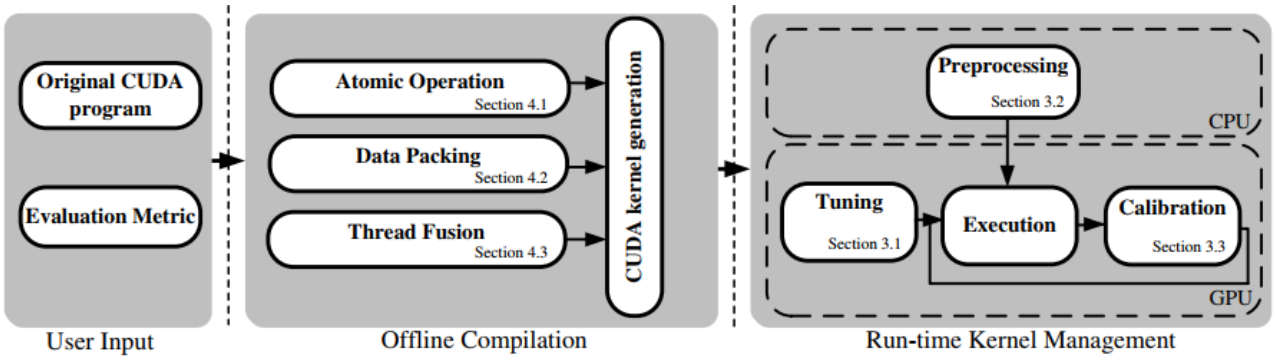


Figure 4.1: An overview of the SAGE framework

Samadi et al.[14] propose **SAGE** framework, an automated technique targeting GPUs which combines a static compiler that automatically generates a set of CUDA kernels with varying levels of approximation with a run time system which selects among the available kernels to achieve better performances while complying with a target output quality (TOQ) set by the user. SAGE has two phases: **offline compilation** and **run time kernel quality management**. During offline compilation, SAGE performs approximation optimization on each kernel by creating multiple versions with varying degrees of accuracy. At run time a greedy algorithm is used to tune the parameters of the approximate kernels in order to identify the best configuration with the highest performances and a result quality satisfying the TOQ. Calibrations and kernel updates are performed by SAGE and the kernel configuration is updated accordingly.
SAGE utilizes three optimization techniques in order to create approximate CUDA kernels.

- **Atomic Operation Optimization**: Atomic operations are commonly used in multi-threading applications such as the ones usually run on GPUs in order to make writes to a common variable sequential. This optimization selectively skips atomic operations that generate frequent collisions thus reducing performances since threads are serialized.

- **Data Packing**: The number of bits needed to represent a variable or an array of variables is reduces, lowering precision while improving latency of memory operations.

- **Thread Fusion**: This optimization eliminates some GPU threads by combining similar threads into a single one and replicating their output.

4.2.3 Approximate Circuits

In this section several circuit level approaches will be analyzed.

Inexact Arithmetic Circuits

Probably the inexact arithmetic circuits have been one of the most active field of research in approximate computing. Arithmetic basic building blocks such as adders and multipliers are simplified making them inexact i.e. having a non-zero possibility to produce inexact output, but also smaller, faster and less consuming. A naive approach is to modify the basic full adder block design in order to reduce or avoid the carry chain. Various different designs have been proposed and sometimes implemented in silicon prototypes.
Wang et al[2] propose a **GDA** (gracefully-degrading-accuracy-configurable adder). Their GDA adder

consists of some basic adder units, where each unit is a k-bit adder which can be implemented using any adder design scheme as shown in fig 4.2. An **N-bit** GDA adder is considered. Given two **N-bit** addends **A** and **B**, they are partitioned into segments (e.g. A = (A_3, A_2, A_1, A_0) and B = (B_3, B_2, B_1, B_0)) with k bits in each and an adder unit is used to compute the segmented partial sum and carry (e.g. $S_n + C_n = A_n + B_n + C_{n-1}$). Adder units are connected using multiplexers, which select the carry-in either from the lesser significant adder unit or from a carry-in prediction component each unit is equipped with. If all the multiplexers select the carry-in from the prediction unit the delay to execute the addition is minimum (almost equal to delay of the single Adder Unit) but the approximation error can be the maximum one. Instead, if the selected carry-in is the one directly produced by the lesser significant adder unit for each unit, the delay is maximum but the precision is also maximum. Thus it can be said that using the multiplexers control signals it is possible to tune the precision of the final computed result. It is worth noting that in order to get a full precise result up to the N-bit all the units producing a lesser significant result segment that the N-unit have to provide a precise result i.e. the paired multiplexer has to select the previous unit carry-in. Also, it should be noted that the error rate is dependent on the carry-in prediction unit, that will not analyzed in this essay.Figure 4.2 shows the GDA schematic.
There exist some other circuit approximation techniques focused on synthesis tools that generate approximate circuitry given an accuracy constraint instead of designing the basic arithmetic blocks by hand.
Venkataramani et al.[12] propose **SALSA**, a Systematic methodology for Automatic Logic Synthesis of Approximate circuits. Given a RTL specification of a circuit and a quality constraint that basically defines the amount of error or uncertainty that may be introduced in the hardware implementation, SALSA synthesizes and approximate version of the circuit adhering to the quality bound that have been specified. Moreover the approximate synthesis problem is mapped into an equivalent traditional logic synthesis procedure, thus allowing the existing synthesis tools to be utilized for approximate logic synthesis. In order to implement this new methodology SALSA leverages a feature called *Approximation Don't Cares*, that allows the circuit simplification using traditional *don't cares* based optimization technique.
Unfortunately since the energy-accuracy trade off is performed at design time, usually these methods cannot guarantee the best efficiency and are outperformed by some others techniques. Still, especially the SALSA method which automatically synthesizes approximate circuits for a given error constraints can achieve very good results.

Voltage over-scaling (VOS) techniques

The main idea behind this technique is to let circuits operate at a higher frequency than the one allowed by the supply voltage. The timing margins to ensure a correct result(e.g. setup time) are no more respected, so timing errors in the computation might appear. However, since digital circuits power consumption scales quadratically with supply voltage ($P \propto CV_{dd}^2 f_{clk}$) through this technique it is possible to obtain important energy gains. Also, frequency and voltage can be easily modulated dynamically through techniques such as **DVFS** (Dynamic Voltage and Frequency Scaling) allowing a fine energy-accuracy trade-off. However, any over-scaling modification has to be finely tuned, to avoid catastrophic and unacceptable errors.

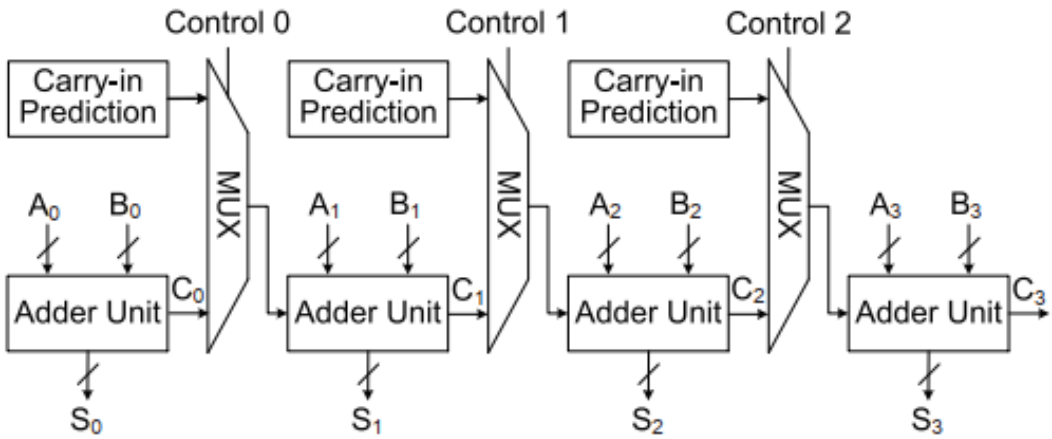


Figure 4.2: The proposed gracefully-degrading accuracy-configurable adder[2]

Precision Scaling Techniques

Precision scaling are probably the most powerful, general and easily available way to implement approximate computing paradigm. Even our proposed **ORMAC** falls in this category and will be further analyzed in the following chapter. Through precision scaling the bit width is reduced or extended at run-time accordingly to the required output accuracy. In the literature there exist several implementations.
Yeh et al[15] propose dynamic precision scaling for improving efficiency of physics-based animation. Real-time physics shows a certain degree of resilience in floating point (FP) operations. So in this paper they describe an architecture with a hierarchical FPU leveraging dynamic precision reduction to allow an efficient FPU sharing among multiple cores. The area required by these cores is then reduced, thus allowing more cores to be integrated. Their technique finds out the minimum precision required by an application by performing design time profiling. At run-time, the energy difference between consecutive application steps is measured and checked against a threshold to detect whether the simulation is becoming unstable. If the simulation becomes unstable the full precision is restored and then progressively reduced again until the minimum stable value is found. Reducing the precision in a floating point based application can lead to three main additional optimization opportunities:

- A FP operation may become trivial, such as multiplication by one or a power of two, operations which would not require the usage of the FPU at all.
- Similar values can be combined into a single value which improves the coverage of cache techniques and can allow using a look table for performing FP multiply and addition operations.
- Precision scaling can allow using smaller FPU resulting in the improvements already described above.

Based on these opportunities, a hierarchical FPU architecture is proposed. A simple core-local (meaning that it can be used just by the core it belongs) is used at L1 level and full precision FPUs are share at the L2 level in order to save surface area for allowing more cores to be integrated. Hence, an operation where precision reduction is possible is executed on the core-local L1 FPU. A more complicated operation where full precision is required is instead executed on the L2 shared FPU.

4.2.4 Approximate Architectures

The best results both in terms of power consumption and performances are obtained if hardware and software are strictly optimized. The approximate computing paradigm is no exception and adapting either the processor or the SOC on which the approximate application will run can provide better results.

Processor Instruction Set Architectures (ISA)

The **ISA** is the main interface between the processor hardware and the software that will run on it. It can be optimized for either fine-grained or coarse-grained approximate computing. For **fine-grained** technique a set of special instructions allows the compiler to decide whether something (in terms of single instruction, group of instructions or an entire part of a program) can be approximated or not, mapping it to approximate or exact hardware. Specifically the arithmetic, logic and FP assembly and machine instructions should be doubled, in order to have one precise and one approximate instruction version. Unfortunately this technique can not guarantee considerable improvements in terms of power consumption, since in most processor architectures the most of the energy is consumed in control, data transfers and clock distribution and none of these latter blocks can be approximated. In **coarse-grained** approximate computing specific code segments are directly mapped to dedicated approximate accelerators or full different cores outside of the processor pipeline.
Esmaeilzadeh et al[3] propose **Truffle**, a new processor architecture supporting new ISA extensions, which aim to exploit what they call **Disciplined Approximate Programming**. Such programming paradigm lets programmers declare which parts of a program can be approximated and consequently lower the energy request. Then, a proper compiler proves statically that all the approximate computation is properly isolated from precise computation and generates the target machine code, letting the actual hardware to decide how to approximate such signaled parts of code. In this way the hardware is lightened from the complexity of correctness checks. The two main contributes achieved from the authors of this paper are the Truffle ISA and its processor architecture description.
With Disciplined Approximate Programming, a program is divided into two components: one running *precisely* i.e. like a conventional computer, and a second one running *approximately* offering no guarantees in terms of correctness but instead an expectation of best effort computation. Of course in the latter subset fall the resilient parts of the code such as FP computations in error-tolerant applications.
Their ISA design follows two basic principles: *approximation abstraction* and *unsafety*. With the former a guaranteed results are replaced by informal expectations without specifying which technique will be used to approximate, with the latter the hardware executing the ISA blindly trust the compiler to enforce the separation between data that must be precise and data that can be approximated. This ISA extension consists of new instruction variants that leave certain aspects of their behavior undefined. Balance must be guaranteed though, to avoid catastrophic results. Indeed, control flow, exception handling and memory access have to be maintained predictable. The proposed approximate-aware ISA exhibits the following properties:

- *fine-grained granularity* to interleave approximate computation with precise ones. For example, a loop variable increment has to be precise while an arithmetic, logic or floating point operation may be approximate.

- *approximate storage* support. The compiler should be able to instruct the ISA to store data approximately or precisely in registers and caches.

ISA extensions for disciplined approximate programming. These instructions are based on the Alpha instruction set.

Group	Approximate Instruction
Integer load/store	LDx.a, STx.a
Integer arithmetic	ADD.a, CMPEQ.a, CMPLT.a, CMPLE.a, MUL.a, SUB.a
Logical and shift	AND.a, NAND.a, OR.a, XNOR.a NOR.a, XOR, CMOV.a, SLL.a, SRA.a, SRL.a
Floating point load/store	LDF.a, STF.a
Floating point operation	ADDF.a, CMPF.x, DIVF.a, MULF.a, SQRTF.a, SUBF.a, MOV.a, CMOV.a, MOVFI.a, MOVIF.a

Figure 4.3: The proposed ISA extension[3]

- Address computations are always precise in order to avoid the writing of arbitrary memory locations.

The extended ISA presents approximate versions of all integers arithmetic, floating-point arithmetic and bitwise operation instructions provided by the original ISA. The extended ISA instruction present the same form as their precise version but give no guarantees about their output values, instead providing some sort of "expected value". For instance the **ADD.a** (add approximate) instruction takes two argument and produces one output that has no guarantee to be the sum of the two operands. The instruction is expected to perform an addition but neither the compiler nor the programmer should rely on the output.
Register modes are not set explicitly by the compiler. Each register can be, at any time, in either *approximate mode* or *precise mode* depending on the precision of the last instruction that has written to it. Basically a precision operation makes the destination register precise, while an approximate operation makes it approximate and then unreliable.
Finally quick glance to the proposed Truffle processor architecture represented in figure 4.4. Such an architecture must carefully distinguish between resilient and non-resilient structures i.e. structures where completely reliable operations are always required. Instruction *fetch* and *decode* have to be precise and their target and source register have to be identified without errors as well. However data content of those registers may be approximate as well as the operation that will work on them. Similarly, memory addresses have to be computed in a error-free way but the data gotten from the memory can present approximation.
Thus the micro-architecture is divided into two distinct planes: *data movement/processing plane* and *instruction control plane*. Register file, data caches, load/store queue, functional units and bypass network belong to the former group which can be approximate. Fetch, decode and control flow hardware belong to the latter group which should be kept precise.
We know that at each frequency level f_{max} is associated a minimum supply voltage V_{min} and lowering the voltage below that minimum can cause timing error, while allowing a significant power consumption reduction though. This processor architecture exploit voltage reduction as a technique to reduce energy consumption as well as applying the approximate computing paradigm. The main idea is to run critical non-resilient structures always at a safe voltage i.e. a voltage which guarantees a correct functioning if the maximum frequency constraints are respected, while non-critical structures are allowed to work at a lower voltage. Hence two different voltage lines exist: one for precise operations and one for approximate operations.

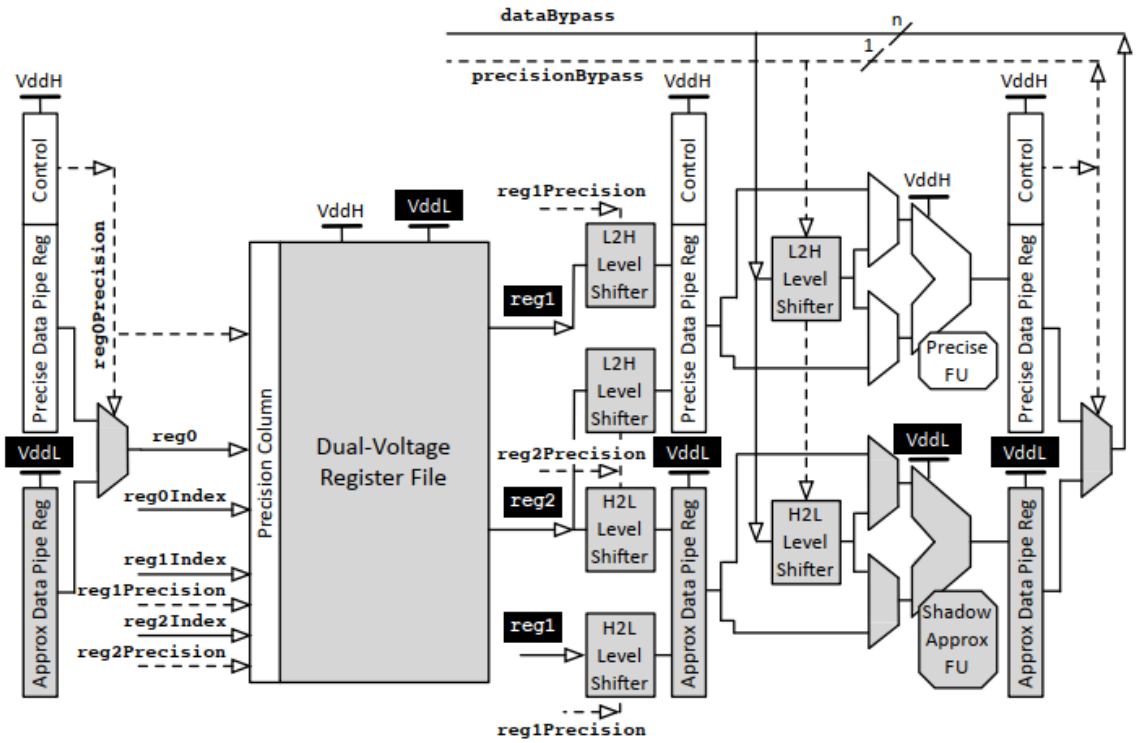


Figure 4.4: The proposed **dual-voltage** datapath[3]

SOC Architectures

Circuit and architectures techniques such te ones proposed above, often require changes on the SOC level, especially when aggressive voltage-scaling techniques are used (e.g. in Esmaeilzadeh [3] as discussed earlier). The whole SOC, including volatile memories and peripherals has to be organized in specific voltage domains, which can influence the full SOC layout, setting new constraints on frequency generators and voltage regulators.

4.2.5 Approximate Software

We can group techniques belonging to the approximate software paradigm into two main groups: application level and programming languages/compilers.

Programming Languages and Compilers

Some language have been proposed to properly fit a program which exploits the approximate computing paradigm. **EnerJ**[4] and **Rely**[16] are programming languages that provide approximation abstraction through their syntax.

- **EnerJ** is an extension to Java that adds approximate data types. By using these types, the system automatically maps variables tagged as approximate to approximate storage, uses approximate operations and, if provided by the programmer, applies energy efficient algorithms. Isolation of precise variables and operations from the approximate components is guaranteed statically, eliminating the need of dynamic checks, further improving energy savings.

```
@Approx int a = 10;
int p;            // precise by default
p = endorse(a);   // explicit casting from approximate to precise
```

Construct	Purpose
@Approx, @Precise, @Top	Type annotations: qualify any type in the program. (Default is @Precise.)
endorse(e)	Cast an approximate value to its precise equivalent.
@Approximable	Class annotation: allow a class to have both precise and approximate instances.
@Context	Type annotation: in approximable class definitions, the precision of the type depends on the precision of the enclosing object.
_APPROX	Method naming convention: this implementation of the method may be invoked when the receiver has approximate type.

Figure 4.5: Summary of EnerJ's language extensions[4]

- **Rely** delegates the task of defining data flows to the compiler. It is defined as an imperative language that enables developers to specify and verify quantitative reliability specifications for programs that allocate data in unreliable memory regions and use unreliable arithmetic or logical operations[16].

Along with completely new programming language like Rely or extensions to existing ones like EnerJ there exist libraries exposing abstractions that can model approximate data types and operations such as **Uncertain<T>**[17]. The main target of Uncertain<T> is the representation of those data that are uncertain by nature such as sensor data, probabilistic models, machine learning, big data, human biometric data and basically all data coming from a measure since there is always a difference (*uncertainty*) between the "true" value and its estimate. This uncertainty is represented by a probability distribution while the computations and conditional expression are supported by a Bayesian network. Uncertain<T> is available for *C++*, *C#* and *Python*.

Application Level

A lot of approximate computing techniques are focused on the application level while the underlying hardware is unaware of the approximation. These techniques can be run on any type of pre-existing hardware yielding good performance improvements while on the other side their gains in term of power consumption are negligible.
Following some examples:

- **Loop perforation**: A rather easy technique based on the idea of skipping some iterations of a loop to reduce computation overhead. Hence, accuracy is traded for performance by transforming loops to execute just a subset of their iterations[18]. Sidiroglou et al [18] propose a first phase filtering out *critical loops* to identify *tunable loops* and a second phase for finding Pareto-optimal perforation policies.
  They also identify several global computation patterns that works well with the loop perforation technique such as the Monte Carlo simulation, which is frequently used in finance and engineering to model outcomes of highly unpredictable processes.

- **Task skipping**: A technique where memory references, tasks or input samples are skipped to achieve better efficiency while reducing the output precision. Samadi et al.[19] present Paraprox, a SW only approximate computing technique which identifies common patters in data-parallel program and uses a specific approximation strategy for each pattern, in some cases implementing the task skipping paradigm.

- **Memoization**: This is basically a caching technique. It is based on the idea of storing previously computed results for future reuse in similar contexts such as functions having similar inputs. Values are stored in an accurate way but when they are reused as results for other functions they can be considered as an approximate result of what should have been the actual output of the function. Thus, instead of calling a computationally hard function a memorized value is fetched reducing the energy and execution time required. In Rahimi et al[20], authors applied this technique in SIMD (Single Instruction Multiple Data) architectures leveraging the high temporal and spacial data locality that characterizes this kind of architecture.

4.3 Final comments

This concludes the survey on approximate computing. As we can see, lots of different techniques have been proposed in recent years. While software based techniques are easier to implement and evaluate, hardware approaches are rather difficult to be properly implemented and tested, since the costs of building an entire new system on silicon is still quite high. In most of the cases when dealing with hardware approaches, researchers report their results based on software simulations or FPGA implementations. While this method can apply well in the prototyping phase, it is not enough to have a complete perspective and allow approximate computing to become a new actual design standard. Though, the research in this field is still at the beginning and, especially when dealing with RMS applications, approximate computing can have a promising future. However, to be extended to the general purpose market, it is fundamental to improve the flexibility provided by the approximate systems. If the target are mobile and IoT systems self-adaptability techniques should be introduced, in order to let the system modify the computing precision depending on external factors, such as battery state of charge or input data quality. With this in mind a closer relationship between hardware and software should be targeted in order to optimize the programs for their platform and reach better performances and lower power consumption.

Chapter 5

Redundant arithmetic and ORMAC Unit

5.1 Introduction to the ORMAC Unit

The **Online Redundant Multiply and Accumulate** (ORMAC) unit is a new type of MAC unit theorized by TIMA's lab members Ali Skaf et al. in the paper "On-Line Adjustable Precision Computing" [21]. It uses **SBD** (**Binary Signed Digit**) as the basic computing unit, allowing to obtain the result starting from the **Most Significant SBD** (**MSD**), one digit at each clock cycle, since there is no carry propagation. It also allows to choose the precision at will, stopping the computation with a variable number of digits after the MSD.
This unit aims to settle a new way of thinking about hardware approximate computing, where speed performance is not an issue and low power consumption is the main constraint. It makes it possible to choose the precision depending on the context or the application provided.

5.1.1 On-line arithmetic

First a brief overview on what is called On-line arithmetic. On-line arithmetic principles were introduced by Ercegovac and Trivedi in 1977 [22][23]. The **On-line** property implies that to generate the jth digit of the result, it is necessary and sufficient to have the operands available up to the $(j+\delta)$th digit, where the difference δ is a small positive constant. In order to produce the first digit of the result it is necessary to provide δ initial digits of the operands. Then, one digit of the result is produced upon receiving one digit of each of the operands. Thus δ is defined as the *on-line delay*. Algorithms based on this principle can be used to speed up arithmetic units thanks to their potential to perform a sequence of operations in an overlapped fashion. Another application of great interest to the approximate computing field, is in performing variable precision arithmetic. The on-line technique implies a left-to-right digit-by-digit (hence starting from the most significant digit) algorithm. The use of redundant number representation is required for on-line algorithms. If a non-redundant numeric system is used, even for basic operation such as addition and subtraction, the on-line delay is $\delta = m$ where m is the number of digits due to carry propagation [22].
In our case the chosen redundant representation is Signed Binary Digit (SBD), further described in the next section. It is worth noting that the On-line arithmetic coupled with a redundant representation such as SBD can have a remarkable impact on power consumption for three main reason:

- The redundant numeric system allows to get rid of the carry chain, since there is no carry propagation. This simplify significantly the unit design.
- The On-line method implies a reduced number of components (as we will se in the next section).

Thus the implementation scales better with the bit size w.r.t. a classic implementation.

- Since the result is obtained starting from the MSD we can stop the computation earlier with a reduced precision.

It appears clearer why this paradigm can have a remarkable impact on the approximate computing field, especially in therms of flexibility and power consumption reduction.

5.2 Signed Binary Digit (SBD) Arithmetic Principles

SBD arithmetic theory has been theorized by Avizienis in 1961[24]. This arithmetic is defined as redundant. Thus every SBD is represented on two bits $\mathbf{a}^+$ and $\mathbf{a}^-$ such that $\mathbf{a} = \mathbf{a}^+\mathbf{a}^- = \mathbf{a}^+ - \mathbf{a}^-$. Three values are possible for an SBD $\{\bar{1}, 0, 1\} = \{01, 00 \text{ or } 11{,}10\}$ and two different codings exist for $0 : \{00, 11\}$
Example on 4 SBDs:

$$-5 = \bar{1}011 = 0\bar{1}\bar{1}1 = 0\bar{1}0\bar{1} \tag{5.1}$$

$$-5 = -8 + 2 + 1 = -4 - 2 + 1 = -4 - 1 \tag{5.2}$$

5.2.1 Addition-Subtraction

Basic Blocks

Two basic blocks exist to perform an addition or a subtraction. These blocks are called **PPM** for Plus-Plus-Minus and **MMP** for Minus-Minus-Plus (w.r.t the single Full Adder Block that exists in the classic binary arithmetics). These two blocks allow to add or subtract an SBD $\mathbf{a} = \mathbf{a}^+\mathbf{a}^-$ with a single bit b.

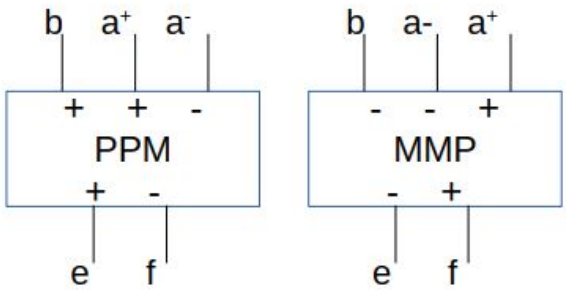


Figure 5.1: The PPM and MMP blocks

$$\mathbf{PPM} : \mathbf{e} = \mathbf{Maj}(b, a^+, \bar{a}^-) \tag{5.3}$$

$$\mathbf{MMP} : \mathbf{e} = \mathbf{Maj}(b, a^-, \bar{a}^+) \tag{5.4}$$

$$\mathbf{f} = b \oplus a^+ \oplus a^- \tag{5.5}$$

Parallel Addition-Subtraction

Combining more PPMs and MMPs makes possible to build multiple SBD addition. To add two SBDs $\mathbf{a} = \mathbf{a}^{+}\mathbf{a}^{-}$ and $\mathbf{b} = \mathbf{b}^{+}\mathbf{b}^{-}$ we note that $\mathbf{a} + \mathbf{b} = ((\mathbf{a} + \mathbf{b}^{+}) - \mathbf{b}^{-})$ which is done by a PPM followed by an MMP. Adding two SBD numbers $\mathbf{A} + \mathbf{B} = ((\mathbf{A} + \mathbf{B}^{+}) - \mathbf{B}^{-})$ is hence done by a hybrid adder followed by a hybrid subtractor, i.e. a row of PPMs followed by a row of MMPs.
It is worth noting that there is no global carry (or borrow) propagation thanks to redundancy. The execution of parallel addition/subtraction is done in a constant time, independently of the operand size.

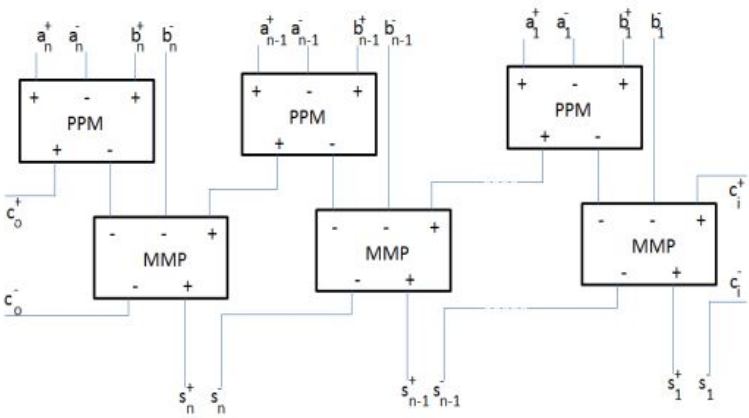


Figure 5.2: Two-input SBD adder/subtractor

Serial Addition-Subtraction (On-line)

The SBD arithmetics also allows to build operators working in a serial way instead of parallel. This means computing the result digit by digit, one for each clock cycle, using just two basic blocks, hence greatly reducing circuit area.
D blocks represent *D-type flip-flops* and are used to keep the different weight of operands correct, contributing to what is called on-line delay. We can choose to compute the addition starting with the LSD (Least Significant Digit) or perform an on-line operation by starting with the MSD (Most Significant Digit).

5.2.2 Multiplication

To multiply two SBD $\mathbf{a} = \mathbf{a}^{+}\mathbf{a}^{-}$ and $\mathbf{b} = \mathbf{b}^{+}\mathbf{b}^{-}$ we use the formulas

$$s^{+} = a^{+}.b^{+} + a^{-}.b^{-} \tag{5.6}$$

$$s^{-} = a^{+}.b^{-} + a^{-}.b^{+} \tag{5.7}$$

5.2.3 Online Multipication

To perform an On-line multiplication of two SBD numbers $P = A \cdot X$ the algorithm proposed by Trivedi & Ercegovac [22] is used.

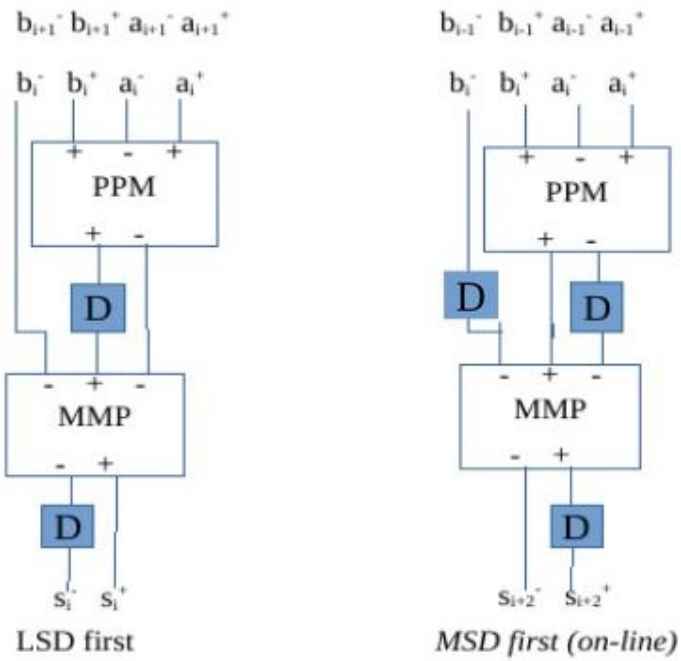


Figure 5.3: SBD serial adder

Let

$$A = \sum_{i=1}^{m} a_i \cdot r^{-i} \tag{5.8}$$

$$X = \sum_{i=1}^{m} x_i \cdot r^{-i} \tag{5.9}$$

be the radix r (in our case it will be $r = 2$ since we are dealing with binary arithmetic) representations of the positive multiplicand and multiplier, respectively. Then define

$$A_j = \sum_{i=1}^{j} a_i \cdot r^{-i} = A_{j-1} + a_j \cdot r^{-j} \tag{5.10}$$

$$X_j = \sum_{i=1}^{j} x_i \cdot r^{-i} = X_{j-1} + x_j \cdot r^{-j} \tag{5.11}$$

to be the j digit representations of the operands A and X available at the jth step by definition of an on-line algorithm. The corresponding partial product then is:

$$A_j \cdot X_j = A_{j-1} \cdot X_{j-1} + (A_j \cdot x_j + X_{j-1} \cdot a_j) \cdot r^{-j} \tag{5.12}$$

Let P_j be the scaled partial product:

$$P_j = A_j \cdot X_j \cdot r^j \tag{5.13}$$

in this way the recursion of the multiplication algorithm can be expressed as:

$$P_j = rP_{j-1} + A_j \cdot x_j + X_{j-1} \cdot a_j \tag{5.14}$$

By defining $P_0 = 0$, the scaled product $P_m = A \cdot X \cdot r^m$ can be generated in m steps. If a redundant numeric system is used, the result can be obtained starting from the most significant digit. The recursive step can be performed in a time independent from the operand precision by exploiting the carry free addition property of redundant numeric systems.

5.2.4 ORMAC Unit

By using the blocks and techniques presented previously we can build the On-line Redundant MAC unit. With slight modifications to the adder block we can also introduce a way to choose the precision at will i.e. choose how many digits will form the result starting from the MSD. SBDs data ($\mathbf{a_k}$

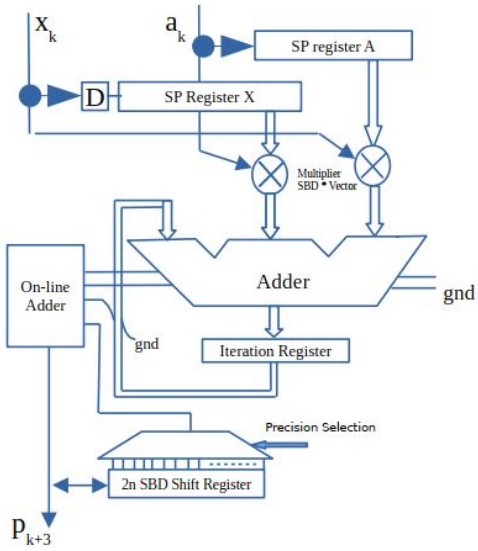


Figure 5.4: On-line redundant MAC

and $\mathbf{x_k}$) are inserted one by one, one each clock cycle, starting with the MSD. Partial product are generated and then added to the previously accumulated value. One of the most interesting feature of this unit is that both input and output are provided starting from the Most Significant Digit, the exact opposite with respect to the classical units. This can open up new possibilities, especially when more ORMAC units are used together. If a unit exploits the result of the previous one for its computations there is no need to wait until the first computation is over, but results obtained from the MSD can be directly used by the second one.

These features are made possible by the choice of the redundant numbering system. Using this kind of notation system, instead of the classical binary one allows to get rid of carry propagation as seen in the Parallel Addition paragraph.
However one issue can arise from the cost of passing from redundant notation system to classical binary.

5.2.5 Conversion from Binary to SBD

The conversion from binary to SBD is straightforward. Let $\mathbf{A}$ be a 2's complement binary number on N bits, $\mathbf{S} = \mathbf{S_p}\mathbf{S_m}$ a SBD number on N digits where $\mathbf{S_p}$ is the SBD positive part (on N bits) and $\mathbf{S_m}$ the SBD negative part(on N bits). The conversion can be done in the following way:

$\mathbf{S_p}$(N-1) <= '0';
$\mathbf{S_p}$(N-2 downto 0) <= $\mathbf{A}$(N-2 downto 0);
$\mathbf{S_m}$(N-1) <= $\mathbf{A}$(N-1);
$\mathbf{S_m}$(N-2 downto 0) <= (others => '0');
The impact of this conversion is small since it can be done with

5.2.6 Conversion from SBD to Binary

An MMP adder will be used to subtract every bit. The carry will be propagated through all the MMP blocks. The carry at the last MMP block will decide the sign bit of the 2's complement number. The delay of this circuit is completely dependent on the implementation of the MMP blocks.

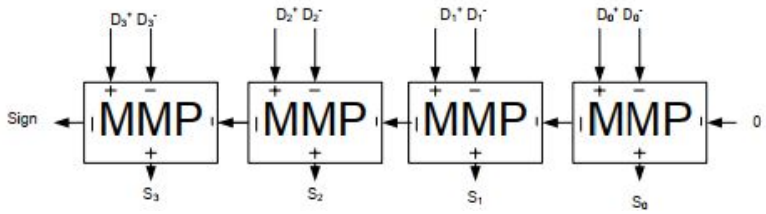


Figure 5.5: Block diagram to convert from a 4 digit SBD number to a 5 bit binary number

The delay of this unit is O(n). Better architectures exist based on the sparse tree adder.

5.3 ORMAC Implementation

The original VHDL code for the ORMAC unit has been developed by Mona Ezzadeen, another TIMA laboratory member. Since the original unit gets input data and provides result in a serial on-line way, in order to use it in a more complex environment such as Rocket Chip and to test it against normal MAC units, I made some modifications to the original unit and then I embedded the modified unit in a wrapper. The wrapper allows to load and retrieve binary data in a parallel way while the whole computation is done by the inner ORMAC unit using BSDs.

5.3.1 ORMAC Wrapper

Datapath

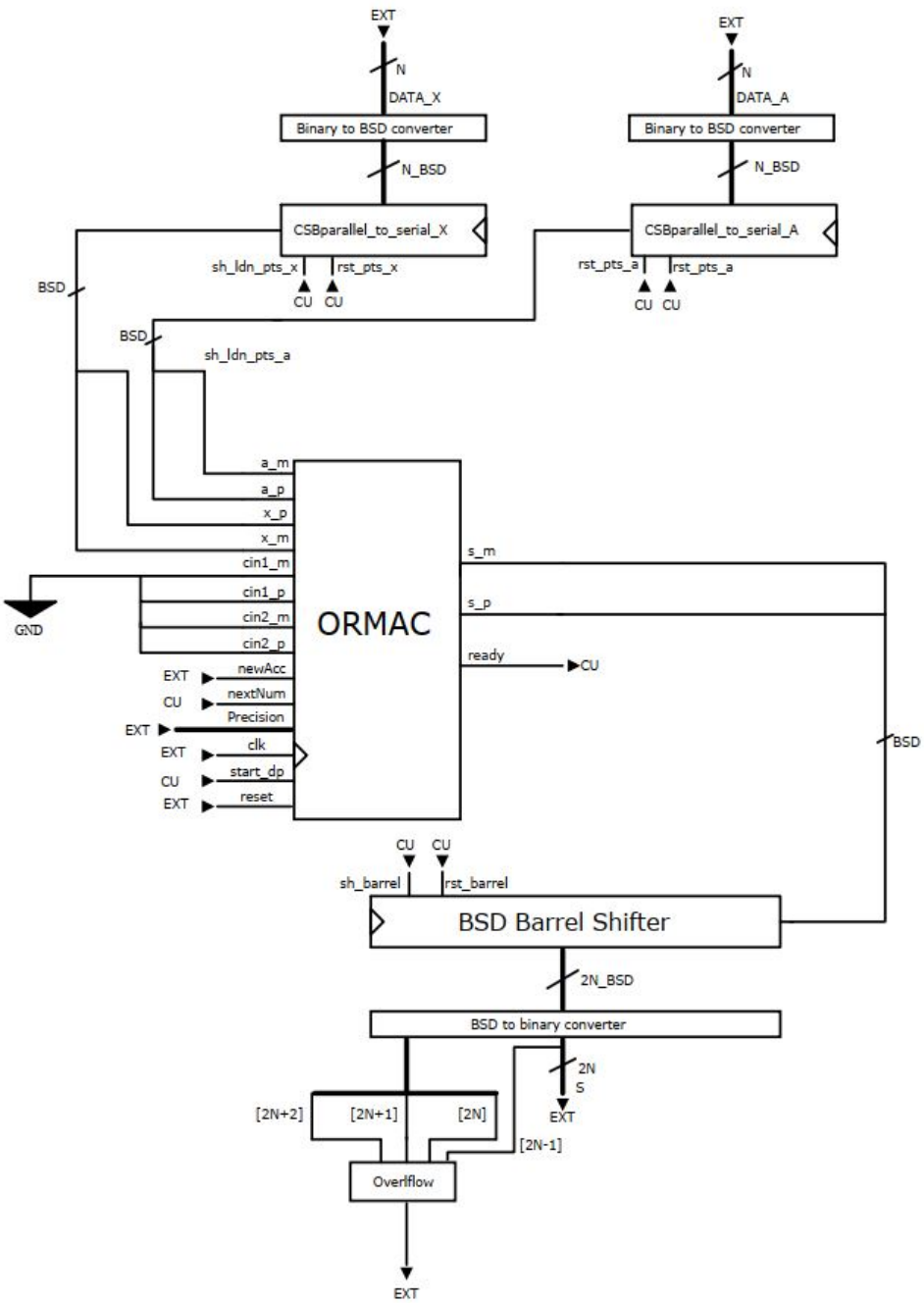


Figure 5.6: Wrapper Datapath

- **Binary to SBD converter**: Input converters, one for each input datum
- **SBD Parallel to Serial**: Shift registers needed to feed the ORMAC unit with a single SBD each clock cycle
- **ORMAC**: Actual ORMAC unit. Inside it there are a datapath and a control unit

- **SBD Barrel Shifter**: An Incremental barrel shifter. The first SBD after reset will be placed in the MSD position, the second one in the MSD-1 position etc.
- **SBD to binary converter**: The last component, required to convert back the number. It is worth noting that the ORMAC unit provides an output on 2N+3 SBDs because of the so called *delay SBD*. These SBDs are obtained as most significant SBDs and they are caused by the internal flip-flops. If the result is correct (so if there is no overflow) when the value is converted back to binary the three most significant bits are equal to the sign bit; if this condition is not respected so we have overflow. Then we can use these three bits to compute the unit overflow.
- **Overflow computation unit**: it is nothing more than three logic gates, one *and* gate and two *nor* gates as shown in the figure 5.7. It makes possible to know when we are trying to compute a value greater than the maximum possible value that can be represented on 2N bits.

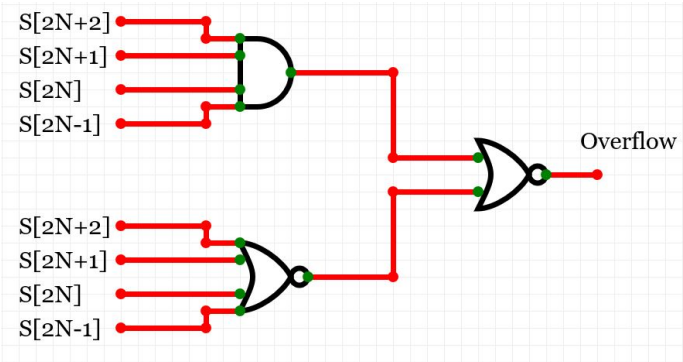


Figure 5.7: Overflow Circuit

Control Unit

A simple Control Unit, it feeds the datapath with signals needed to load, enable and reset registers. Due to design constraints it is a mealy machine. This choice was made to enforce the synchronization with the ORMAC internal CU.

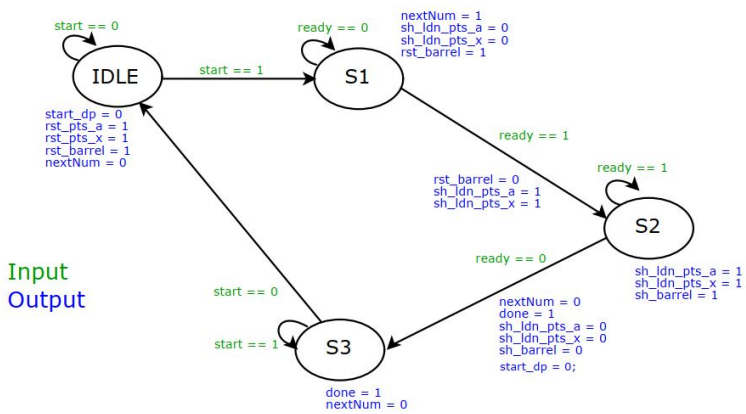
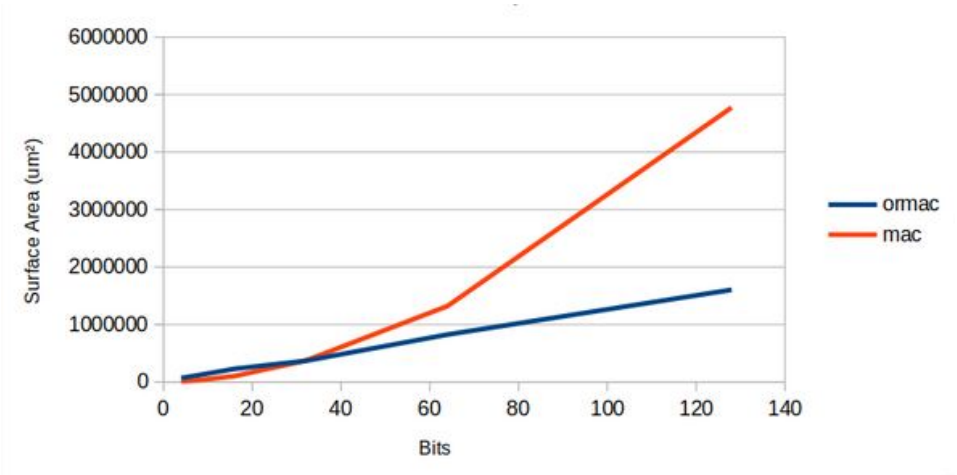


Figure 5.8: Wrapper Control Unit

5.4 Development, Testing and Results

The entire development and debugging has been done using Xilinx Vivado Design Suite[1] and its integrated simulator.
For the area analysis computation we used Synopsys Design Compiler [2] ,while for the power consumption analysis we used Synopys PrimeTime[3] The ORMAC unit aims to reduce power consumption and area sacrificing speed performances. Following the main results we obtained in the comparison with a classic Multiply and Accumulate unit, coded in a behavioral way.
All the result showed are obtained using the input/output wrapper around the basic ORMAC circuit.

Surface Area per input bit length - Synopsys Design Vision



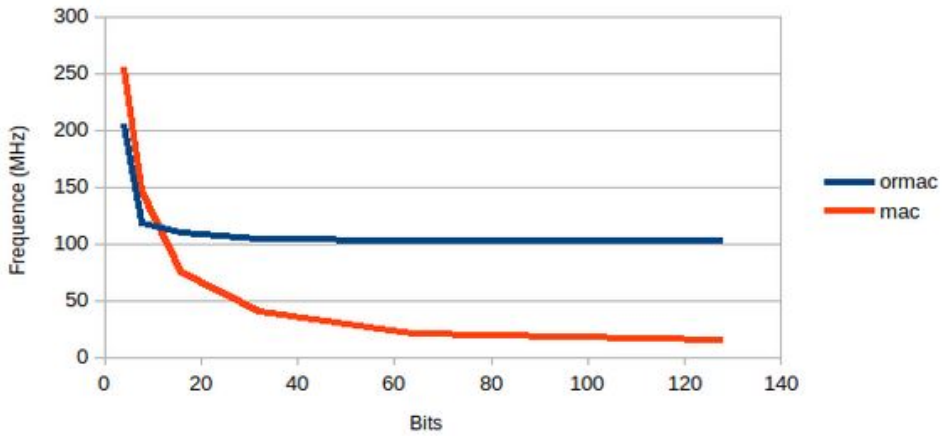
The ORMAC uses much less surface than a normal MAC if we use more than 32 bit otherwise their area is comparable. This behavior is probably due to the inner circuital complexity of the ORMAC w.r.t. the classic mac. Even for small size inputs the ORMAC presents many more components than the classic version. However if the bit size grows, the complexity of the multiplier and adder inside the classic mac becomes higher, and as consequence the circuit area grows. The impact of the bit size on the ORMAC is definitely lower since the on-line adder, which is one of the components, has a fixed size independently of the input bit length.

[1]Vivado Website: `https://www.xilinx.com/products/design-tools/vivado.html`

[2]Design Compiler Website:`https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-graphical.html`

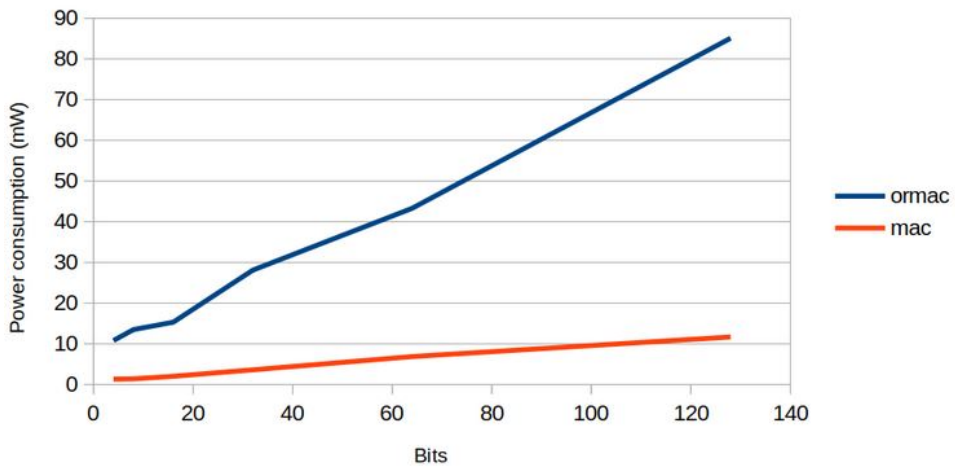
[3]`https://www.synopsys.com/support/training/signoff/primetime1-fcd.html`

Maximum Frequency per input bit length - Synopsys Design Vision



The frequency stabilizes around 102 MHz. The result is definitely better than the classic mac. By increasing the bit size the carry chain of the classic mac becomes longer, heavily impacting on the circuit longest path. This reduces the frequency significantly. The ORMAC frequency dependency on the input bit length is definitely lower, not having carry propagation. Hence the ORMAC maximum frequency is kept constant after an initial drop.

Power Consumption per input bit length - Synopsys Prime Time



In terms of power consumption the ORMAC is not as good as expected. This is probably due to the ORMAC datapath design that could be definitely improved. This will be something to be investigated in a future work.

Chapter 6

32 bit ORMAC Unit Integration inside Rocket Chip

6.1 Introduction

As widely explained in the chapter related to Rocket Chip, this platform is extremely customizable and it fits perfectly our needs. However it is not straightforward getting used to it. It lacks in terms of documentation and since it is a relatively new platform, not so much support exists on the internet.
To integrate our ORMAC unit inside Rocket Chip two main paths exist. Either embedding it as an accelerator exploiting the RoCC, strictly coupling the unit with the CPU, or adding it as an external peripheral, allowing the processor to write and read data through memory mapped registers. Even if the former is the most indicated one it is more difficult and it is actually needed only in case of performance constraints. Thus the latter was my choice.

6.2 BlackBox Structure

In general the BlackBox structure is used when there is some IP written in Verilog that is likely to be included in Chisel design (our case) or when it is not possible to express some module because of Chisel semantics and so it is useful to code and include the module in Verilog. Chisel is a dialect of Scala which itself derives from Java. Then it is no surprise that Chisel is an Object Oriented programming language. Hence to add peripherals to the main SoC it is necessary to extend a basic class through inheritance. If the new class, that represents the new peripheral, extends the basic class *Module*, the circuit behavior needs to be described in Chisel, if instead the new class extends *BlackBox* it is possible to create just the interface without implementing the actual internal logic. Thus the class that extends *BlackBox* is nothing more than a container, a top level view written in Chisel whose instantiation can be written in Verilog and then referenced in the Makefile.
The name of the blackboxed module needs to match the Verilog module one so that the Verilog compiler can properly resolve the instantiation. The BlackBox Verilog code is reported in Appendix.1.1. It features a top level **module** including a datapath, which is the actual ORMAC, and a further level of control unit, which is needed to synchronize the BlackBox unit with the system it is attached to.
This new unit will be compiled together with all the other files to produce the C++ emulator, a fundamental component for Rocket Chip based development since it can also generate circuit's waveforms.

6.3 Integration Procedure

6.3.1 Code Translation

As I previously wrote the whole ORMAC code that I modified, developed and tested the wrapper for, was coded in VHDL. But Chisel accepts BlackBox peripheral written in Verilog, since Verilator, the program that generates the emulator, supports only Verilog. Hence after designing, testing and debugging the wrapped VHDL unit (process that took me a while to be completed), I needed to fully convert the code from VHDL to Verilog. I did this partially using a software and then I refined the translation manually. The software I used is **vhd2vl** (https://github.com/ldoolitt/vhd2vl). Unfortunately not every construct is supported and not everything is always well translated. So a manual review of the entire code was needed. In particular the *generics* VHDL construct was not supported at all by the conversion program.
After completing the conversion the Verilog code was tested and debugged again, to be sure that no bugs where introduced by the translation procedure and to ensure the same behavior as the VHDL original code.

6.3.2 Adding the hardware module to Rocket Chip

Custom Peripherals code with the BlacBox unit attached can be found in Appendix.1.2. It is a *.scala* file, written in Chisel and provided by Rocket Chip. It has to be modified and expanded every time a new unit has to be added to the SoC.

- **Adding the BlackBox**: Writing to the file dedicated to the custom peripherals our BlackBox unit, that represents the top level view of our ORMAC, setting also the input/output bit length. It is worth noting that even if our unit takes 32 bit input and provides 64 bit (+1 overflow bit) output, the BlackBox unit is instantiated using 64 bit input/output, to make simpler the register mapping.

- **Adding the configuration and data registers**: After configuring the base address for all the custom peripherals we need to map the input/output registers used by our unit inside the peripherals address map, paying attention to maintain the addresses aligned, otherwise the code will not compile. Usually custom peripherals addresses start at `0x2000`, so to maintain them aligned, knowing that Rocket Chip addresses bytes, we have to use 8 bytes multiples starting from the base address (so `0x2008`, `0x2010`, `0x2018` and so on...).

- **Modifying the configuration**: to include the peripherals in the future generated emulator we need to modify the configuration file adding to our configuration the information that new peripherals, memory mapped at the addresses we defined, exist. Then we can move the Verilog files describing the wrapped ORMAC unit inside the folder dedicated to Verilog peripherals. Finally we have to modify the Makefile, adding the Verilog files to the list of files to be compiled.

6.3.3 Developing the HAL

To interface the higher level software to be run on the CPU with the new added peripheral we need to develop a Hardware Abstraction Layer which takes into account the register writing/reading, exposing a simpler API. I did this using the C programming language and my choice was to create an .h file called *mac.h* where I put the registers mapping of the ORMAC unit and the functions declaration to interact with them, and a *mac.c* where I put the functions definition. Using these files

it is possible to write a *main.c* file that can easily interact with the ORMAC unit, providing inputs and getting outputs.
Following the two main function exposed in the HAL:

- `char mac(int32_t a, int32_t b, int64_t*s, uint64_t precision);` it allows to set up the `a, b` data input registers and the `p` precision register, the `s` pointer to the variable that will contain the result of the operation. It returns 1 if there is overflow, 0 otherwise.

- `void reset_mac(void);` it allows to reset to 0 the mac accumulator register. To be executed before the `mac` function every time we want to start a new accumulation.

It is worth noting that before embedding the wrapped ORMAC inside Rocket Chip, a small three-states synchronization control unit was added. This control unit stops the mac computation every time a new result is ready. In this way it is possible for the CPU executing the machine code of the main file that we provided to read the result and write it in a variable. However this external control unit can be easily integrated inside the wrapper's one. The synchronization unit is available in the *Blackbox.v* code in Appendix.1.1.
The CPU can access results in two ways: polling and interrupts. Both ways are supported by Rocket Chip. The former was my choice. Before reading the results the CPU "polls" i.e. continuously reads the *ready* bits until it becomes 1. At this point the result can be read from the peripheral register and written inside a variable. HAL code is available in Appendix.1.3 with both the .h and .c files.

6.3.4 Testing

The Verilog ORMAC unit has been extensively tested through an exhaustive test on 16 bits. So 2^{32} combinations (2^{16} combinations for a input and 2^{16} combinations for b input) were tested without rising any error. To test it in a coherent manner with the Rocket Chip environment, the Verilog code was translated in C++ using Verilator. The testbench was written in C++ using C++11 libraries for multithreading, in order to distribute the workload on four cores. The test took about 60 hours. I made this choice to speed up the testing phase. Indeed trying to test the entire Rocket Chip SoC using the C++ emulator was too slow. Full code of the testbench is available in Appendix.1.4.

Chapter 7

Future Work and Conclusion

Regarding the ORMAC unit, surface area and maximum frequency results are quite promising, meaning that the redundant arithmetic principles are worth to be further explored when applied to approximate computing based units. However, several improvements can be carried out. The ORMAC unit datapath design especially, can be improved a lot, reducing its power consumption and in general making more straightforward the understanding of the code itself.
Other improvements include the usage of interrupts instead of polling in the Rocket Chip integration part.
This work can set up the path for some interesting future extensions. The main challenge is to find a proper application that fits multiple ORMACs, in a configuration that can enhance its best characteristics such as variable precision and results obtained from the most significant digit. An idea could be to implement an array/matrix operation accelerator, particularly useful when dealing with machine learning or image processing algorithm, that are also two fields where approximate computing fits perfectly. This kind of unit can be also integrated as a directly coupled accelerator in Rocket Chip leveraging the RoCC unit, granting a considerable performance boost with respect to the peripheral integration that I did. Generally producing a single digit every clock cycle can be a slowdown for high performance computation, even if an approximate result is acceptable. So the main challenge is to find an architecture or an algorithmic application where this is not a slowdown but a feature that can enhance performances. However Reducing the unit power consumption is the main objective to be achieved. In this way this new unit can fit in application where speed is not a critical constraint such as IoT.
Finally, regarding the approximate computing field state of art, we have seen that a lot of studies, targeting different computing levels (hardware, architecture, software), have been carried out. However not all of them are likely to have a remarkable impact on future research, especially if their flexibility is limited. Being able to choose the computation precision whether a power consumption reduction is needed is the key feature that can really boost the adoption of such paradigm in future devices.

References

[1] Andrew Waterman. Design of the RISC-V instruction set architecture, 2016.

[2] Wang, Yuan, Kumar, and Xu. On reconfiguration-oriented approximate adder design and its application. 2013.

[3] Esmaeilzadeh, Sampson, Ceze, and Burger. Architecture support for disciplined approximate programming. 2012.

[4] Sampson, Dietl, Fortuna, Gnanapragasam, Ceze, and Grossman. Enerj: Approximate data types for safe and general low-power computation. 2011.

[5] Asanovic et Al. The rocket chip generator, 2016.

[6] Davide Pala and Massimo Poncino. Design and programming of a coprocessor for a RISC-V architecture, 2017.

[7] Bert Moons, Daniel Bankman, and Marian Verhels. *Embedded Deep Learning Algorithms, Architectures and Circuits for Always-on Neural Network Processing.* 2019.

[8] Mittal and Sparsh. A survey of techniques for approximate computing. 2016.

[9] Xu, Qiang, Mytkowicz, Todd, Kim, and Nam Sung. Approximate computing: A survey. 2016.

[10] Noureddine Ait Said. Sysax project: Self-adaptive approximate system-on-chip: Development of the software platform.

[11] Maria Avgerinou, Paolo Bertoldi, and Luca Castellazzi. Trends in data centre energy consumption under the european code of conduct for data centre energy efficiency. 2017.

[12] Venkataramani S, Sabne A, Kozhikkottu V, Roy K, and Raghunathan. SALSA: systematic logic synthesis of approximate circuits. 2012.

[13] Chippa et al. Analysis and characterization of inherent application resilience for approximate computing. 2013.

[14] Samadi, Lee, Jamshidi, Hormati, and Mahlke. SAGE: Self-tuning approximation for graphics engines. 2013.

[15] Ye, Faloutsosy, Ercegovac, Patelz, and Reinmany. The art of deception: Adaptive precision reduction for area efficient physics acceleration. 2007.

[16] Carbin and Misailovic and. Verifying quantitative reliability for programs that execute on unreliable hardware. 2013.

[17] Bornholt, Mytkowicz, and McKinley. UncertainT: A first-order type for uncertain data.

[18] Sidiroglou, Misailovic, and Hoffmann. Managing performance vs. accuracy trade-offs with loop perforation. 2011.

[19] Samadi, Jamshidi, Lee, and Mahlke. Paraprox: Pattern-based approximation for data parallel applications. 2014.

[20] Rahimi, Benini, and Gupta. Spatial memoization: Concurrent instruction reuse to correct timing errors in simd architectures. 2013.

[21] Ali Skaf, Mona Ezzadeen, Mounir Benabdenbi, and Laurent Fesquet. On-line adjustable precision computing. 2019.

[22] Trivedi and Ercegovac. On-line algorithms for division & multiplication. 1977.

[23] Ali Skaf and Alain Guyot. Vlsi design of on-line add/multiply algorithms. 1993.

[24] Avizienis and Algirdas. Signed-digit number representations for fast parallel arithmetic. 1961.

Appendices

.1 Source Codes

.1.1 BlackBox

Listing 1: BlackBox.v

```
module blackbox(
   input clock,
   input reset,
   input clear,
   input req,
   input [63:0] a,
   input [63:0] b,
   input newAcc,
   input [63:0] precision,
   output reg [63:0] s,
   output reg ready,
   output overflow
    );

wire [63:0] s_s;
wire rst_mac;
wire mac_done;
reg start_mac;
assign rst_mac = clear || reset;
TOPMAC #(.NSIZE(32), .PREC(7))
mac (
   .start(start_mac),
   .reset(rst_mac),
   .clk(clock),
   .DATA_A(a[31:0]),
   .DATA_X(b[31:0]),
   .newAcc(newAcc),
   .precision(precision[6:0]),
   .S(s_s),
   .done(mac_done),
   .overflow(overflow)
);
//internal variables
reg[1:0] state;
reg[1:0] next_state;
//internal constants
parameter IDLE = 2'b00, COMP = 2'b01, DONE = 2'b10;
//comb logic
always @ (state, req, mac_done, s_s)
begin: FSM_COMB
   next_state = state;
   start_mac = 0;
   ready = 0;
   case (state)
      IDLE : begin
            if (req == 0) begin
               next_state = IDLE;
            end else begin
               next_state = COMP;
            end
         end
```

```
        COMP : begin
                start_mac = 1;
                s = s_s[63:0];
                if (mac_done == 1) begin
                    next_state = DONE;
                end else begin
                    next_state = COMP;
                end
            end
        DONE : begin
                ready = 1;
                    if (req == 1) begin
                    next_state = DONE;
                end else begin
                    next_state = IDLE;
                end
            end
        default: next_state = IDLE;
    endcase
end
//seq logic
always @ (posedge clock)
begin : FSM_SEQ
    if (clear || reset) begin
        state <= IDLE;
    end else begin
        state <= next_state;
    end
end

endmodule
```

.1.2 Custom Peripherals

Listing 2: CustomPeripherals.scala

```
package freechips.rocketchip.devices.tilelink

import Chisel._
import freechips.rocketchip.subsystem._ //BaseSubsystem
import freechips.rocketchip.config.{Parameters, Field}
import freechips.rocketchip.diplomacy._
import freechips.rocketchip.regmapper.{HasRegMap, RegField}
import freechips.rocketchip.tilelink._
import freechips.rocketchip.util.UIntIsOneOf

/*******************************/
/*        BlackBoxModule       */
/*******************************/

/*describes a w-bit wide mac written in
 Verilog. This module uses an object instanciated
 from class blackbox, which extends the special
 BlackBox class*/


```

```
class BlackBoxModule(w: Int) extends Module{ // p must be log2(w) + 1
  val io = new Bundle {
    val clock     = Clock(INPUT)
    val reset   = Bool(INPUT)
    val clear   = Bool(INPUT)
    val req   = Bool(INPUT)
    val a = UInt(INPUT, w)
    val b = UInt(INPUT, w)
    val newAcc = Bool(INPUT)
    val precision = UInt(INPUT, w)
    val s = UInt(OUTPUT, w)
    val ready = Bool(OUTPUT)
    val overflow = Bool(OUTPUT)
  }
  val blackbox = Module(new blackbox(64)).connect(io.clock,io.reset,io.clear,io.req,
                        io.a,io.b, io.newAcc, io.precision, io.s, io.ready,
                        io.overflow)
}

class blackbox(w: Int) extends BlackBox {
  val io = new Bundle {
    val clock = Clock(INPUT)
    val reset = Bool(INPUT)
    val clear   = Bool(INPUT)
    val req   = Bool(INPUT)
    val a = UInt(INPUT, w)
    val b = UInt(INPUT, w)
    val newAcc = Bool(INPUT)
    val precision = UInt(INPUT)
    val s = UInt(OUTPUT, w)
    val ready = Bool(OUTPUT)
    val overflow = Bool(OUTPUT)
  }

  def connect(c: Clock, r: Reset, clear: Bool, req:Bool, a: UInt, b: UInt,
              newAcc: Bool, precision: UInt, s: UInt, ready: Bool,
              overflow: Bool) = {
    io.clock := c
    io.reset := r
    io.clear := clear
    io.req := req
    io.a := a
    io.b := b
    io.newAcc := newAcc
    io.precision := precision
    s := io.s
    ready := io.ready
    overflow := io.overflow
  }
}
/*****************************/
/*                           */
/*                           */
/*          TopLevel         */
/*      CustomPeripherals    */
/*                           */
/*                           */
```

```
/******************************/
case class CustomPeripheralsParams(
  address   : BigInt,
  beatBytes: Int)

class CustomPeripheralsBase(w: Int) extends Module {
  val io = IO(new Bundle {
    /* BlackBoxModule Interface */
    val blackboxmod = new Bundle {
        val clock     = Clock(INPUT)
        val clear   = Bool(INPUT)
        val reset   = Bool(INPUT)
        val req   = Bool(INPUT)
        val a = UInt(INPUT, w)
        val b = UInt(INPUT, w)
        val newAcc = Bool(INPUT)
        val precision = UInt(INPUT)
        val s = UInt(OUTPUT, w)
        val ready = Bool(OUTPUT)
        val overflow = Bool(OUTPUT)
    }
    /* Another Module Interface */
  })
  /* BlackBoxMod */
  val blackboxmod = Module(new BlackBoxModule(64))
  blackboxmod.io.clock       := clock
  blackboxmod.io.reset       := reset
  blackboxmod.io.clear       := io.blackboxmod.clear
  blackboxmod.io.req         := io.blackboxmod.req
  blackboxmod.io.newAcc      := io.blackboxmod.newAcc
  blackboxmod.io.precision   := io.blackboxmod.precision

  io.blackboxmod.s           := blackboxmod.io.s
  io.blackboxmod.ready       := blackboxmod.io.ready
  io.blackboxmod.overflow    := blackboxmod.io.overflow

  blackboxmod.io.a           := io.blackboxmod.a
  blackboxmod.io.b           := io.blackboxmod.b

  /* Another Module */
}
/******************************/
/*                            */
/*                            */
/*          TopLevel          */
/*      Registers Mapping     */
/*                            */
/*                            */
/******************************/
trait CustomPeripheralsTLModule extends HasRegMap  {
  implicit val p: Parameters
  val io: CustomPeripheralsTLBundle
  def params: CustomPeripheralsParams

  /*************************/
  /* CustomPeripherals Base */
  /*************************/
```

```
  val base = Module(new CustomPeripheralsBase(64))
  /************************/
  /* BlackBloxMod Signals */
  /************************/
  val blackboxmod_a          = Reg(UInt(64.W))
  val blackboxmod_b          = Reg(UInt(64.W))
  val blackboxmod_s          = Reg(UInt(64.W))
  val blackboxmod_clear      = RegInit(false.B)
  val blackboxmod_req        = RegInit(false.B)
  val blackboxmod_ready      = Reg(UInt(1.W))
  val blackboxmod_newAcc     = RegInit(false.B)
  val blackboxmod_precision  = Reg(UInt(64.W))
  val blackboxmod_overflow   = Reg(UInt(1.W))

  // Outputs
  blackboxmod_s                  := base.io.blackboxmod.s
  blackboxmod_ready              := base.io.blackboxmod.ready
  blackboxmod_overflow           := base.io.blackboxmod.overflow
  // Inputs
  base.io.blackboxmod.a          := blackboxmod_a
  base.io.blackboxmod.b          := blackboxmod_b
  base.io.blackboxmod.clear      := blackboxmod_clear
  base.io.blackboxmod.req        := blackboxmod_req
  base.io.blackboxmod.newAcc     := blackboxmod_newAcc
  base.io.blackboxmod.precision  := blackboxmod_precision

  /*************************/
  /*      Register Map     */
  /*************************/
  regmap(
      /* BlackBoxMod */
      0x00 -> Seq(
        RegField(64, blackboxmod_a)),
      0x08 -> Seq(
        RegField(64, blackboxmod_b)),
      0x10-> Seq(
        RegField(64, blackboxmod_s)),
      0x18 -> Seq(
        RegField(64, blackboxmod_precision)),
      0x20 -> Seq(
        RegField(1, blackboxmod_clear)),
      0x28 -> Seq(
        RegField(1, blackboxmod_req)),
      0x30 -> Seq(
        RegField(1, blackboxmod_ready)),
      0x38 -> Seq(
        RegField(1, blackboxmod_newAcc)),
      0x40 ->  Seq(
        RegField(1, blackboxmod_overflow))
      )
}
/*********************************/
/* Encapsulation in TileLink     */
/*  * explicit use of Interruption */
/*********************************/
class CustomPeripheralsTL(c: CustomPeripheralsParams)(implicit p: Parameters)
  extends TLRegisterRouter(
```

```
    c.address, "customperipherals", Seq("ucbbar,customperipherals"),
    interrupts = 4, beatBytes = c.beatBytes)(
      new TLRegBundle(c, _) with CustomPeripheralsTLBundle)(
      new TLRegModule(c, _, _) with CustomPeripheralsTLModule)

trait HasPeripheryCustomPeripherals { this: BaseSubsystem =>
  implicit val p: Parameters

  private val address  = 0x2000
  private val portName = "customperipherals"
  // LazyModule
  //  * create and conects diferents nodes to make "requests"
  //  * others modules also make "requests" with their nodes
  //  * "requests" are resolved when LazyModule is realized with a .module
  val customperipherals = LazyModule(new CustomPeripheralsTL(
      CustomPeripheralsParams(address, pbus.beatBytes))(p))
  // TileLink node: pbus
  pbus.toVariableWidthSlave(Some(portName)) { customperipherals.node }
  // TileLink node: ibus (interrupt bus)
  // (remember set numbers of interrupts in CustomPeripheralsTLs)
  ibus.fromSync := customperipherals.intnode
}
// LazyModuleImp
//  * is approximately like Chisel Modules
//  * interface to transport signals for top-level (dut.pwm_pwmout)
trait HasPeripheryCustomPeripheralsModuleImp extends LazyModuleImp {
  val outer: HasPeripheryCustomPeripherals
  // Another Module
}
```

.1.3 ORMAC Hardware Abstraction Layer

Listing 3: mac.h

```
#ifndef __MAC_H
#define __MAC_H


// The base address of the hardware peripheral
#define MAC_BASE_ADDR 0x2000
#define MAC_A_REGISTER_ADDR          0x2000
#define MAC_B_REGISTER_ADDR          0x2008
#define MAC_S_REGISTER_ADDR          0x2010
#define MAC_PRECISION_ADDR           0x2018
#define MAC_RESET_ADDR               0x2020
#define MAC_REQ_ADDR                 0x2028
#define MAC_READY_ADDR               0x2030
#define MAC_NEWACC_ADDR              0x2038
#define MAC_OVERFLOW_ADDR            0x2040

#include "util.h"


/** @brief Sets the register A (INPUT)
 *
 *  @param a The new value on 32 bits.
```

```
 *  @return Void.
 */
void setA(int32_t a);

/** @brief Sets the register B (INPUT)
 *
 *  @param b The new value on 32 bits.
 *  @return Void.
 */
void setB(int32_t b);

/** @brief Gets the register S (INPUT)
 *
 *  @param Void.
 *  @return S The value of the result register.
 */
int64_t getS(void);

/** @brief Resets the unit accumulator
 *
 *
 */
void reset_mac(void);

void request(void);

/** @brief Function to use the mac
 *
 *  @param a The A register value
 *  @param b The B register value
 *  @param s The result, passed by reference
 *  @param precision the desired precision
 */
char mac(int a, int b, int64_t *s, uint64_t precision);


char get_overflow(void);

#endif
```

Listing 4: mac.c

```
#include "mac.h"

void setA(int32_t a)
{
    *(volatile uint64_t *)(MAC_A_REGISTER_ADDR) = (int64_t)a;
}

void setB(int32_t b)
{
    *(volatile uint64_t *)(MAC_B_REGISTER_ADDR) = (int64_t)b;
}

int64_t getS(void)
{
    while(*(volatile uint64_t *)(MAC_READY_ADDR) == 0); //polling
```

```
    int64_t result = *(volatile uint64_t *)(MAC_S_REGISTER_ADDR);
    *(volatile uint64_t *)(MAC_REQ_ADDR) = 0;
    return result;
}

void reset_mac(void)
{
    setA(0);
    setB(0);
    *(volatile uint64_t*)(MAC_REQ_ADDR) = 0;
    *(volatile uint64_t*)(MAC_RESET_ADDR) = 1;
    *(volatile uint64_t*)(MAC_RESET_ADDR) = 0;
}
void request(void){
    *(volatile uint64_t *)(MAC_REQ_ADDR) = 1;
}

char mac(int a, int b, int64_t *s, uint64_t precision){
    setA(a);
    setB(b);
    *(volatile uint64_t*)(MAC_PRECISION_ADDR) = precision + 1;
    *(volatile uint64_t*)(MAC_NEWACC_ADDR) = 0;
    request();
    *s = getS();
    return getOverflow();
}

char get_overflow(void){
    return *(volatile uint64_t *)(MAC_OVERFLOW_ADDR);
}
```

.1.4 Testbench

Listing 5: testbench

```
#include <stdlib.h>
#include "Vblackbox.h"
#include "verilated.h"
#include <iostream>
#include <thread>
#include <mutex>

#define N 65536

std::mutex mtx;

void delay(unsigned n, Vblackbox *tb) {
    for(unsigned i = 0; i < n; i++){
        tb->clock = 1;
        tb->eval();
        tb->clock = 0;
        tb->eval();
    }
}

void func(Vblackbox* tb, size_t tn, size_t begin, size_t end) {
```

```
    tb->clock       = 0;
    tb->reset       = 1;
    tb->clear       = 0x00;
    tb->req         = 0x00;
    tb->newAcc      = 0x00;
    tb->ready       = 0x00;
    tb->a           = 0;
    tb->b           = 0;
    tb->precision    = 65;
    tb->eval();
    delay(10, tb);
    long result = 0;
    long sum = 0;
    long prev_result = result;
    bool err = false;
    for (size_t i = begin; i < end; i++) {
        for (size_t j = begin; j < end; j++) {
            tb->clock       = 1;
            tb->reset       = 0;
            tb->clear       = 0;
            tb->req         = 1;
            tb->newAcc      = 0;
            tb->a           = i;
            tb->b           = j;
            tb->eval();
            tb->clock       = 0;
            tb->eval();
            delay(100, tb);
            result = tb->s;
            prev_result = result;
            sum += i*j;
            if ( result != sum){
                mtx.lock();
                std::cout << "thread number = "<< tn << "WRONG RESULT!" << std::endl;
                std::cout << "i = " << i << " j = " << j
                        << "previous result = " << prev_result
                        << " current result = " << result
                        << " correct value = "<< sum <<std:: endl;
                mtx.unlock();
                err = true;
                break;
            }
            tb->req = 0;
            tb->eval();
            delay(100, tb);
        }
        if (err)
            break;
        tb->clear = 1;
        tb->eval();
        delay(10, tb);
        mtx.lock();
        std::cout << "thread n " << tn << " i = " << i << std::endl;
        mtx.unlock();
        tb->clear = 0;
        delay(10, tb);
        result = 0;
```

```
        sum = 0;
    }
    if (err) {
        mtx.lock();
        std::cout <<"thread "<< tn << " COMPLETED WITH ERROR! " << std::endl;
        mtx.unlock();
        exit(1);
    }
    else {
        mtx.lock();
        std::cout <<"thread "<< tn << " Completed without errors." << std::endl;
        mtx.unlock();
    }
}

int main(int argc, char **argv) {
    // Initialize Verilators variables
    Verilated::commandArgs(argc, argv);
    // Create an instance of our module under test
    Vblackbox *tb0 = new Vblackbox;
    Vblackbox *tb1 = new Vblackbox;
    Vblackbox *tb2 = new Vblackbox;
    Vblackbox *tb3 = new Vblackbox;
    // Create threads
    std::thread t0(func, tb0, 0, 0, N/4);
    std::thread t1(func, tb1, 1, (N/4)+1, N/2);
    std::thread t2(func, tb2, 2, (N/2)+1, 3*(N/4));
    std::thread t3(func, tb3, 3, 3*(N/4)+1, N);
    // synchronize threads
    t0.join();
    t1.join();
    t2.join();
    t3.join();

    exit(EXIT_SUCCESS);
}
```