NTNU – Trondheim
Norwegian University of
Science and Technology

Turbo Amber

A high-performance processor core for
SHMAC

Anders Tvetmarken Akre
Sebastian Bøe

Master of Science in Computer Science
Submission date: June 2014
Supervisor: Magnus Jahre, IDI
Co-supervisor: Yaman Umuroglu, IDI
Nikita Nikitin, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

The performance increase of state of the art processors has stagnated due to power and thermal constraints. Heterogeneous computing has lately attracted interest and may be the key for improving the performance and energy-efficiency of computing systems under their tight physical budgets. The Single-ISA Heterogeneous MAny-core Computer (SHMAC) project is a research project that seeks to explore the challenges of heterogeneous computing. Therefore, the SHMAC project requires a diversity of processor cores of different size and performance, to explore heterogeneous architectures.

Currently, there is only one processor core, Amber, available for SHMAC. This thesis presents the development of Turbo Amber (TA), a high-performance processor core for SHMAC. We present an overview of high-performance processor architecture techniques and extend the existing Amber core with a branch predictor, a return address stack, an instruction queue and a fast multiplier. The front-end of TA is capable of fetching and performing branch prediction on up to four instructions each clock cycle and can be extended with an appropriate back-end for superscalar execution.

Our results show that TA has a performance increase of 49% over the original Amber core, with a moderate increase in FPGA resource utilization: slice utilization from 4.7% to 7.9%, block RAM utilization from 1.0% to 3.3%, and no DSP slices used on a Xilinx Virtex-5 FPGA. TA has also been integrated into the existing SHMAC-infrastructure and verified by running applications on top of Linux and freeRTOS.

Sammendrag

Ytelsesøkning til avanserte prosessorer har stagnert på grunn av strøm og termiske begrensninger. Heterogene datamaskiner har i det siste vekket interesse og kan være nøkkelen til å oppnå bedre ytelse mens man samtidig unngår overoppheting. SHMAC er et forskningsprosjekt som utforsker utfordringene relatert til heterogene datamaskiner. SHMAC krever prosessorkjerner i varierende størrelse og ytelse, men dessverre er det per dags dato kun én prosessorkjerne, Amber, tilgjengelig for SHMAC arkitekturen.

Denne oppgaven omhandler utvikling av TA, en høy-ytelses prosessorkjerne for SHMAC. Vi gir en oversikt over høy-ytelse prosessoarkitekturteknikker og utvider den eksisterende Amber-kjernen med en branch predictor, en Return Address Stack, en instruksjonskø og en rask multiplikator. Turbo Ambers front-end er i stand til å hente og gjøre prediksjon på inntil fire instruksjoner hver klokkesykel, og kan kobles til en passende back-end for superskalar eksekvering.

Våre resultater viser at Turbo Amber har en ytelsesøkning på 49% over den opprinnelige Amber-kjernen, med en moderat økning i slice-forbruk fra 4,7 % til 5,6 %, blokk RAM forbruk fra 1,0 % til 3,3 %, og ingen DSP-slices på en Xilinx Virtex - 5 FPGA. TA har også blitt integrert inn i SHMAC-infrastrukturen og verifisert ved å kjøre applikasjoner på Linux og freeRTOS.

Problem description

Current multi-core processors are constrained by energy. Consequently, it is not possible to improve performance further without increasing energy efficiency. A promising option for making increasingly energy efficient CMPs is to include processors with different capabilities. This improvement in energy efficiency can then be used to increase performance or lower energy consumption.

Currently, it is unclear how system software should be developed for heterogeneous multi-core processors. A main challenge is that little heterogeneous hardware exists. It is possible to use simulators, but their performance overhead is a significant limitation. An alternative strategy that offers to achieve the best of both worlds is to leverage reconfigurable logic to instantiate various heterogeneous computer architectures. These architectures are fast enough to be useful for investigating systems software implementation strategies. At the same time, the reconfigurable logic offers the flexibility to exploit a large part of the heterogeneous processor design space.

The Single-ISA Heterogeneous MAny-core Computer (SHMAC) project aims to develop an infrastructure for instantiating diverse heterogeneous architectures on FPGAs. A prototype has already been developed, but currently the only processor core available is a 5-stage in-order scalar pipeline. A variety of microarchitectural optimizations, including speculative execution or techniques to exploit the Instruction Level Parallelism (ILP) inherent to applications can be employed to further increase the core performance and/or energy efficiency.

The task in this assignment is to design and implement a high-performance processor tile for SHMAC. The first step is to survey the literature to identify the architectural techniques necessary to improve on the performance of the existing core. The proposed design has to be implemented in a hardware description language, then integrated into and evaluated using the SHMAC prototype. If time permits, the students should suggest extensions to the initial design, implement them and investigate their implications.

Preface

This thesis is submitted to the Norwegian University of Science and Technology.

Acknowledgments

We would like to thank our supervisors Yaman Umuroglu, Nikita Niktin and Magnus Jahre for their excellent guidance. Thank you for assisting us during design decisions and for the Sundays spent reading our thesis.

We would also like to thank our friends working alongside us on different parts of the SHMAC projects for giving us support in their particular expertise. Especially thanks to Joakim Andersson, Håkon Øye Amundsen, Magnus Walstad, Benjamin Bjørnseth, Stian Hvatum, Terje Runde and Håkon Wikene.

Contents

List of Figures

List of Tables

List of Acronyms

ASIC Application-Specific Integrated Circuit.

BHT Branch History Table.

BTAC Branch Target Address Cache.

BTFN Backwards Taken, Forward Not taken.

CISC Complex Instruction Set Computer.

CPI Cycles per Instruction.

FIFO First In, First Out.

FPGA Field-Programmable Gate Array.

FPU Floating-Point Unit.

FSM Finite-State Machine.

HDL Hardware Description Language.

ILP Instruction-Level Parallelism.

IPC Instructions Per Cycle.

ISA Instruction Set Architecture.

PC Program Counter.

RAS Return Address Stack.

RISC Reduced Instruction Set Computer.

SHMAC Single-ISA Heterogeneous MAny-core Computer.

SIMD Single Instruction Multiple Data.

TA Turbo Amber.

UART Universal Asynchronous Receiver/Transmitter.

Chapter 1
Introduction

Modern high-performance processors have emerged out of several decades of scientific research and development. Transistor scaling facilitated a consistent exponential single-threaded performance increase lasting from the 1970s until the early 2000s, as depicted in Figure 1.1. Moore's law [Moo06] and Dennard scaling [DGnY+74] allowed transistors to decrease in size and at the same time consume less power. Performance increased by shortening the critical path and increasing the clock frequency. Engineers also introduced highly complex techniques, like superscalarity and out-of-order execution, for exploiting Instruction-Level Parallelism (ILP). The exponential increase in single-threaded performance eventually discontinued when Dennard scaling came to an end: reducing voltage any further would cause high static power dissipation. As a consequence, utilizing all of the available transistors, or increasing the clock frequency would result in overheating.

Table 1.1 illustrates scaling after the end of Dennard scaling. The table is an excerpt from *The GreenDroid mobile application processor: An architecture for silicon's dark future* [GHSV+11]. The classical scaling column shows how different transistor

Transistor property	**Classical scaling**	**Leakage-limited scaling**
ΔV_t (threshold voltage)	1/S	1
ΔV_{DD} (supply voltage)	1/S	1
Δ quantity	S^2	S^2
Δ frequency	S	S
Δ capacitance	1/S	1/S
Δ power ($\Delta(QFCV_{DD}^2)$)	1	S^2
Δ utilization (1/power)	1	$1/S^2$

Table 1.1: The consequences of the end of Dennard scaling. S is the transistor feature size scaling factor. Table taken from the GreenDroid article [GHSV+11].

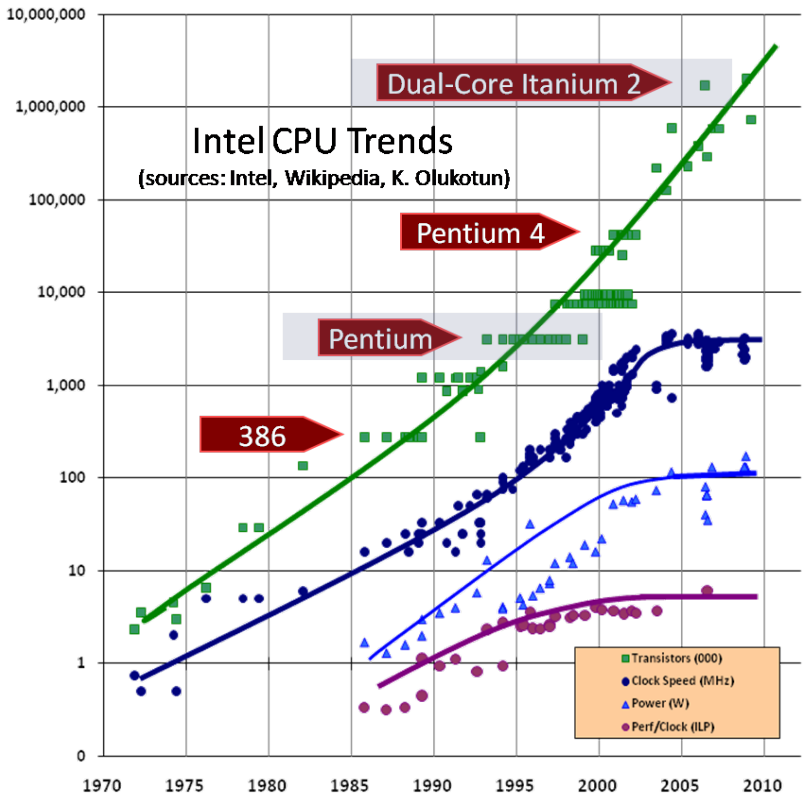

Figure 1.1: Performance history chart over Intel processors, chart taken from [Sut05].

properties were affected by scaling before the end of Dennard scaling. The decrease in threshold and supply voltage compensated for the quadratic increase in transistor count and kept the total power consumption constant. The leakage-limited scaling column shows how scaling affected power consumption after the end of Dennard scaling. Since the voltages no longer can be downscaled in line with transistor feature size the power consumption increases quadratically.

1.1 The Dark Silicon Effect

To overcome the challenge of thermal overheating, the focus shifted from single-core to multi-core architectures, utilizing the increased number of transistors for more cores instead of more complex cores. This improved the overall processor throughput, without increasing the frequency of a single core. It was critical to keep the clock frequency constant since this also kept the dynamic power dissipation constant

($P_{dynamic} \propto capacitance \times frequency \times Voltage^2$) and the processors remained within the power budget. At the same time Moore's law continued and the transistor count per chip grew. We have now reached a point where the transistor density is too high to simultaneously utilize all the transistors without experiencing thermal overheating that may damage the chip. As a consequence, only parts of the chip can be active at once while others have to be powered off in order to stay within the power budget. This is known as the dark silicon effect [EBSA+11].

1.2 The SHMAC Research Project

Today engineers and scientists are exploring the domain of heterogeneous multi-core computers to increase the chip energy efficiency and overcome the dark silicon effect. Heterogeneous systems contain a variety of specialized cores that could achieve higher performance and consume less power when executing specific tasks than general-purpose processor cores. Many argue for heterogeneous computers [BC11] [HM08] [KTR+04] and heterogeneous cores can already be seen in the consumer market, for example ARMs big.LITTLE core.

The EECS group at NTNU contributes to heterogeneous architecture development through an ongoing research project named SHMAC [shm]. SHMAC is an infrastructure for heterogeneous computing systems which assists the study of both hardware and software issues related to heterogeneous architectures. The hardware aspect of SHMAC is to evaluate and compare different hardware architectures to gain insight into which combination of components maximize energy efficiency. The software aspect aims at investigating different programming models for heterogeneous architectures, and how software should be developed for them. SHMAC is instantiated using Field-Programmable Gate Array (FPGA) technology. FPGAs have the benefit of being reconfigurable and faster than software simulations.

The SHMAC architecture can be seen in Figure 1.2. It is a tile-based infrastructure connected by a two dimensional mesh network. Each tile can communicate directly with its four neighbors: north, south, east and west. There are no restrictions on the content of a tile. For instance, a tile may contain a processor core, an accelerator, or on-chip memory. The only requirement is that the tile implements the SHMAC's router interface. There are no dependencies between the tiles, which makes the infrastructure modular.

In the first prototype of the SHMAC infrastructure, there was only one type of processing tile available, named Amber [San], obviously limiting the heterogeneity of the SHMAC project. Amber is a 5-stage in-order scalar processor core which supports the ARMv2a instruction set. The core has been modified to support the SHMAC tile interface. The Amber architecture is hard to extend with features to

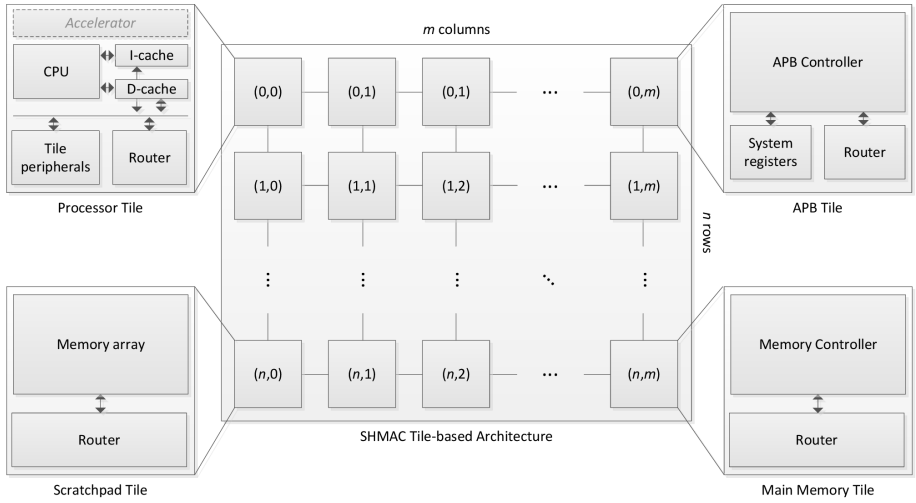

Figure 1.2: The SHMAC architecture

improve its performance without significant modifications. However, the SHMAC project needs a processor tile which can be used as a basis for other tiles with different specializations to research and carry out experiments with a higher degree of heterogeneity. These requirements have motivated the design of a high-performance and extendable processor for SHMAC.

Turbo Amber is a 32-bit high performance ARM processor which extends the original Amber. It has a 5-stage pipeline which has been extended with a front-end capable of fetching up to four instructions per cycle, a branch predictor, a Return Address Stack (RAS), an instruction queue and a fast multiply unit. We have also merged the work of Andersson and Amundsen [AA14] into TA, which upgraded the original Amber to the ARMv4T instruction set.

1.3 Requirements

Requirements are important for systematically evaluating the success of the Turbo Amber project. In this section we present the requirements interpreted from the assignment text and elaborate on why they were included. Some requirements were taken directly from the text, while others were not mentioned in the text but after discussion with EECS researchers were deemed important.

Fast	TA should be faster than Amber.
Verified	TA should be extensively verified to be functionally correct.
Integrated	TA should be integrated into SHMAC as a separate tile.
Area efficient	TA should be area-efficient.
Resource balanced	TA should not excessively drain any one FPGA-resource.

Figure 1.3: The five requirements that TA must satisfy.

1.3.1 Fast

The assignment text specifies that TA should be a high-performance processor. What high-performance means is context dependent. In the context of SHMAC high-performance is relative to the other processors in SHMAC, at the moment only Amber. We therefore rephrase the "high-performance" requirement to be the more testable requirement "faster than Amber". This requirement is important because if the processors have negligible performance differences we will not be able to exploit heterogeneity and the processor will be useless for heterogeneity research purposes.

1.3.2 Area Efficient

Blindly optimizing for performance with no regard to the area usage can lead to an area-inefficient design. To prevent this we add the requirement that TA's performance increase should be proportional to the area increase. We use Pollack's rule as a reference for proportionality between performance increase and area increase. Pollack's rule is a historical observation, akin to Moore's law, that processor performance increases by the square root of the area increase [Pol99]. This gives us the testable requirement formulated in Equation 1.1. The degree to which one can increase performance without excessively increasing area is a testament to the design's quality.

$$\frac{TA\_performance}{A\_performance} \geq \sqrt{\frac{TA\_area}{A\_area}} \tag{1.1}$$

1.3.3 Resource Balanced

TA should not excessively drain any of the available FPGA-resources. Being resource balanced is about not consuming all the specialized FPGA-resources, but instead leaving room for other tiles in what is to be a Many-Core Computer. An FPGA is a heterogeneous computing substrate and has a diverse set of resources available. The three most important resources are generic (logic) slices, DSP slices, and block RAM. The FPGA-resource usage should be balanced and none of the resource types should be excessively drained by instantiating TA. If TA were to use excessive amounts of

DSP slices or block RAM one may not be able to instantiate other specialized tiles such as vector processors or on-chip RAM tiles. These specialized tiles may prove to have a greater need of the specialized resources than TA does. And if TA were to be instantiated many times in a SHMAC layout one might be limited by DSP slices before all the generic slices were utilized. This would leave the FPGA in an underutilized state.

1.3.4 Integrated

To make TA easy to use for research in instantiating diverse heterogeneous layouts it should be integrated into SHMACs infrastructure as a separate tile. This requirement calls for some engineering effort unrelated to the core design, but is necessary for TA to be usable in SHMAC.

1.3.5 Verified

The final and most challenging requirement for a complex digital design project is verification. Although not specified by the assignment text, it is implied, and of utmost importance, that TA should work as expected without any major bugs. Significant effort should be made to show functional correctness. Applying a few test vectors in a testbench is insufficient if the goal is to run real-life programs and operating systems. The success of this requirement is critical for TA's reusability in the SHMAC project.

1.4 Contributions

An overview of the work done and which requirement they relate to is listed in Table 1.2.

The main contributions were made to satisfy the performance requirement. The fetch efficiency was increased by introducing an instruction queue, a branch history table, and a branch target address cache. The multiply performance was increased by introducing a fast multiplier. To satisfy the SHMAC integration requirement a Turbo Amber tile was created, allowing multiple heterogeneous cores to be instantiated. To verify Turbo Amber's correctness a regression test strategy was used during development and large ARM binaries were run for final verification.

Since we kept in mind the resource usage while designing and analyzed the resource usage reports, no additional optimization efforts were necessary to satisfy the area efficiency or resource balance requirement.

Requirement	Section	Contribution
Fast	3.3	An instruction queue to buffer instructions between the fetch and decode stage.
Fast	3.4	A branch history table to predict branch directions.
Fast	3.5	A branch target address cache to predict branch targets.
Fast	2.2.3	A return address stack to improve function call performance.
Fast	3.7	A fast multiply unit.
Fast	4.4	Benchmarks have been ported to SHMAC and performance has been measured on Amber and Turbo Amber.
Fast	4.3	Microbenchmarks have been designed, run, and analyzed to verify that the performance improvements behave as intended.
Integrated	5.1.4	A Turbo Amber tile has been created.
Verified	4.2	79 ARM assembly programs have been run on TA.
Verified	4.2	Linux has been successfully booted and used to run applications.
Verified	4.2	freeRTOS has been successfully booted and an MPI application has successfully been run.
Verified	N/A	The Amber test framework has been improved to support a much faster simulator.

Table 1.2: An overview of the contributions made with references to where they are detailed and what requirement they satisfy.

Chapter 2
Background

In this chapter we give an overview of architectural techniques for processor design with an extra emphasis on the techniques implemented in this thesis. Amber is presented due to it being the basis from which TA was developed. And finally an ARM Cortex-A8 is reviewed to show how a high-performance ARM processor can be designed.

2.1 Fundamentals Of Processor Design

In this section we present the concepts necessary to be able to reason about performance. We also present classic micro-architectural techniques used in high-performance processors.

2.1.1 Defining Performance

To be able to discuss what a fast processor is, one must first understand what fast means. In the simplest of terms, a processor is fast if it can run a given program in a short amount of time. This execution time can be split into the three factors *instruction count*, *clock cycles per instruction* and *clock cycle time*, as seen in Equation 2.1. In practice the three factors are often interdependent, and a technique that improves one factor may adversely affect another. And when a processor architect designs a micro-architecture he must take into account all three to achieve high performance.[1] The following subsections define each factor and presents them from a processor architect's viewpoint.

$$time = instruction\_count \times clock\_cycles\_per\_instruction \times clock\_cycle\_time \tag{2.1}$$

[1] "He" should be read as "he or she" throughout the thesis.

Instruction Count

For a given program the instruction count is primarily determined by the compiler, but the architect can also have an impact if he can extend the Instruction Set Architecture (ISA). Extensions such as floating point and Single Instruction Multiple Data (SIMD) instructions can significantly reduce the number of instructions needed, but require re-compilation and will therefore not improve binary blobs[2].

CPI

The average number of Cycles per Instruction (CPI) is determined by the ISA and the micro-architecture, with a Reduced Instruction Set Computer (RISC) typically requiring fewer cycles per instruction than a Complex Instruction Set Computer (CISC). This is due to the fact that each CISC instruction does more work than a typical RISC instruction would. To give an example; doing arithmetic operations on values in memory can be one instruction in CISC, while requiring a load-, a register-operation-, and a store-instruction, in a RISC instruction set. This alone does not imply an increase in CPI, they may be single-cycle machines, giving RISC and CISC both 1 in CPI. But in practice high-performance CISC processors decode CISC instructions into RISC-like instructions internally, this decoding implies that each CISC instruction will require several cycles to compute the several RISC operations. The architect can reduce the cycle count by increasing chip area and design complexity with techniques such as superpipelining, superscalarity, branch prediction, trace caches, register renaming, forwarding, high-associative caches, prefetching and many others [SL13].

Clock Cycle Time

The clock cycle time is a property of the micro-architecture and the underlying computing substrate or process technology. The design's longest delay is known as the critical path and determines the cycle time. Changing the technology node will often be outside the scope of what the architect can affect, but reducing the design's longest delay is feasible. To reduce the clock cycle time and thereby improve performance the architect can split the work into multiple pipeline stages or spread the work more evenly between the existing stages.

2.1.2 Pipelining

Modern processor micro-architectures have a high degree of complexity as architects have been given more and more transistors to work with. The following sections present RISC architectures of increasing complexity and performance.

[2]From wikipedia: In the context of open source software, a binary blob is a closed source binary-only driver without publicly available source code.

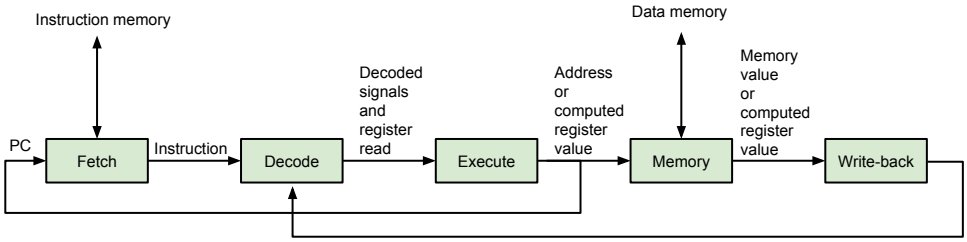

Figure 2.1: The classic 5 stage RISC pipeline.

Figure 2.1 shows how the execution of an instruction can be divided into 5 pipeline stages. The five stages are:

Fetch:	An instruction is read from cache or memory.
Decode:	An instruction is decoded and the register bank is read.
Execute:	A register value or memory address is computed.
Memory:	The data memory is read or written.
Write-back:	The register bank is written.

Splitting the processor into five stages can reduce the clock cycle time by up to a factor of five. But in practice pipelining a processor will not yield perfect frequency speedup due to it being difficult to perfectly balance the pipeline, the added work needed to be done to resolve new hazards, and the overhead from pipeline registers [HP11]. As it has been presented up until now, pipelining seems like the simple act of carefully adding registers at strategic positions in the processor. The true challenge of pipelining comes from hazards.

Hazards

Hazards are systematically divided into control hazards, data hazards and structural hazards. They occur because of very different conditions, but common to all is that they occur because of dependencies between instructions and that to resolve them extra communication between the stages than what is depicted in the simplified Figure 2.1 is required.

Control hazards occur because of conditional branches. If a branch is conditional then the instructions following it are dependent upon the if the branch is going to be computed taken or not taken. Data hazards are common and can occur when an instruction uses the result of a preceeding instruction before the result has been commited to the register bank. Structural hazards occur when two instructions

```
1 beq label
2 mov r1, r2
3 mov r3, r4
```

Figure 2.2: A short assembly snippet that contains a control hazard that needs to be resolved.

attempt to use the same hardware resource, this can happen for example if the L1 data and instruction caches are unified and a load instruction attempts to read the cache at the same time as an instruction is attempted read from the cache.

Consider the program shown in Figure 2.2. The beq label instruction is a conditional branch that may or may not branch to the address aliased to label. After fetching the branch instruction the fetch unit should fetch a new instruction, but the branch instruction has not been computed yet so the next instruction address is unknown. This dependency of the branch instruction result that could potentially disrupt the flow of instructions through the pipeline is known as a hazard. To resolve the hazard the architect can choose between speculatively fetching a new instruction, even though it is not known with certainty what the next instruction is, or stop fetching instructions until the branch has reached the execute stage and the next instruction address is known. If instructions are speculatively fetched and decoded then care must be taken to nullify or flush the instructions if the speculation was incorrect. If the fetch stage stops fetching instructions it may pass along to decode the equivalent of a NOP instruction, this NOP instruction is known as a bubble. These two techniques, bubbling, and flushing are used to resolve other types of hazards as well.

2.1.3 Beyond Pipelining

Simple 5-stage pipelines can give enough performance for the embedded domain, but to achieve truly high performance more advanced techniques are necessary. In the following sections we show how superpipelining can improve the clock cycle time, and how superscalarity and out-of-order execution can improve the CPI.

Superpipelining

Superpipelining is a technique which increases the pipeline depth by dividing each stage into smaller steps. This yields a reduced clock cycle time since the critical path of the design is shortened. Figure 2.3 shows an example of superpipelining where each stage is divided into three sub-stages. This could increase the frequency by up to a factor of three.

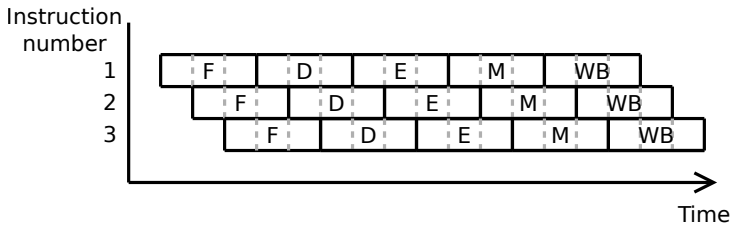

Figure 2.3: A timing diagram of a superpipelined design. Each stage is divided into three sub-stages.

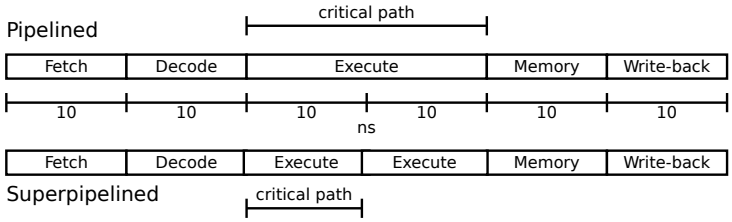

Figure 2.4: How superpipelining can even out the amount of work.

Superpipelining can also even out imbalances in the pipeline. Consider Figure 2.4. The critical path of the execute stage in the upper pipeline is twice the length as the critical paths in the other stages. By dividing the execute stage into two sub-stages, the clock frequency can be doubled.

The architect must take care that the cycle time decrease is not cancelled out by a CPI increase. Increasing the number of stages may hurt the CPI because there will be more hazard possibilities that need to be resolved and because the already existing hazards may require longer pipeline bubbles to resolve. As an example, the branch misprediction penalty will increase with a deeper pipeline.

Superscalarity

While superpipelining improves the clock cycle time to increase performance, superscalar pipelines improve CPI. Superscalar designs are able to reduce the CPI below 1. A superscalar design duplicates hardware at each pipeline stage to allow processing of multiple instructions in parallel. The width, or the way, of the pipeline tells how many instructions each stage is able to process in parallel. For instance, a pipeline of width 2, or a 2-way pipeline, is capable of processing two instructions per cycle, as illustrated in Figure 2.5. Superscalar pipelines must handle dependencies between the instructions issued in parallel. For example, two instructions that write

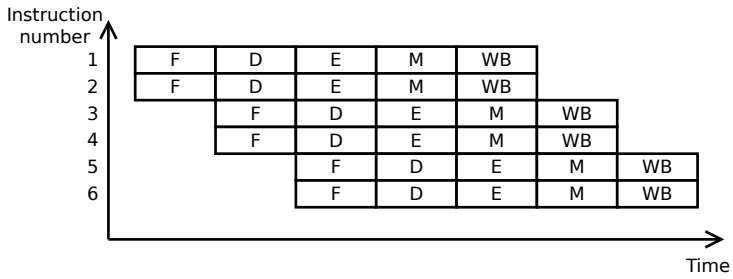

Figure 2.5: A timing diagram of a 2-way superscalar pipeline.

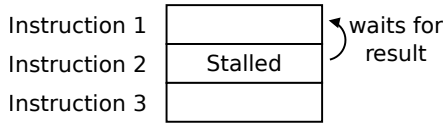

Figure 2.6: An example where OoO execution achieves better performance than in-order execution

to the same register (i.e. WAW hazard) must be completed in the same order as they appear in the instruction stream.

Superscalar pipelines impose a large increase in area due to the duplication. An N-way pipeline needs N copies of each pipeline stage. Also, the result of any of the N units in one pipeline stage may be piped to any of the N units in the next stage, which makes the interconnect between the pipeline stages complicated.

Out-Of-Order Execution

Out-of-Order execution tries to increase the amount of instructions that can be executed in parallel by executing instructions in a different order than they appear in the instruction stream. This way, cycles spent on stalling can instead be used on processing other trailing instructions that are independent of the instruction causing the stall. Consider the example in Figure 2.6. In an in-order execution scheme, the processor will stall until the result of instruction 1 is ready, since the second instruction depends on its result. In an out-of-order processor, the third instruction can be issued while the second instruction waits for its operands.

Figure 2.7 shows the stages of a typical Out-of-Order processor. The fetch and decode stages are in charge of fetching and decoding the instructions. The rename stage renames the destination registers of the instructions by assigning a logical name to

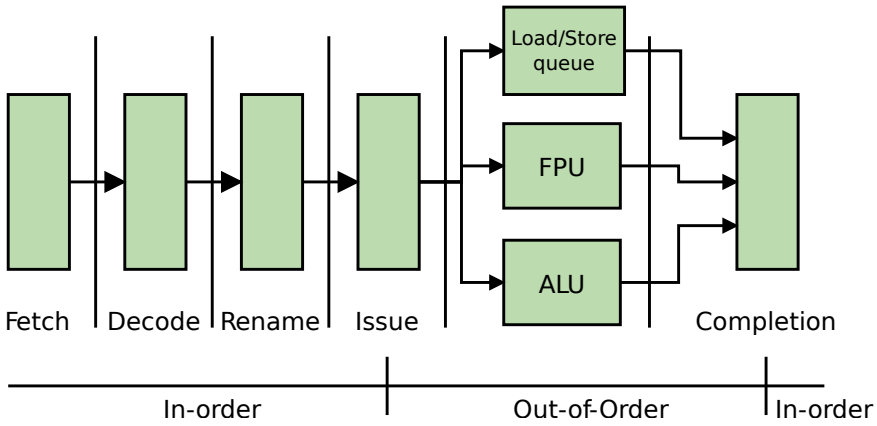

Figure 2.7: Typical architecture of an Out-of-Order processor.

each of them. The mapping between the logical name and the real register is stored in a table. Register renaming eliminates false dependencies (i.e. WAR and WAW hazards) between instructions and allows the processor to exploit a higher degree of ILP. The issue stage queues the instructions until they are ready to be executed. This stage is complex as there are many conditions that need to be fulfilled before an instruction can execute. For example, the operands must be available, and there must be a functional unit ready for a new instruction. The issue queue also decides when and where the instruction is dispatched. The instructions may be dispatched out of order. The execution stage contains various functional units which are specialized for certain operations. For instance, there may be ALUs, multipliers, floating-point units and memory load/store queues. The functional units may use various number of cycles to execute an instruction. When the instructions finish execution, which may happen out of order, they enter the completion stage. The completion stage is a buffer where the result of an instruction is temporarily held until all its preceding instructions are completed and their results written to the register bank. This way, the instructions are written to the register bank in the same order as they appear in the instruction stream.

The out-of-order scheme fetches instructions in-order, issues them out-of-order, and writes them to the register bank in-order. Therefore, by just looking at the fetched instruction stream and the register bank, one would get the impression of in-order execution.

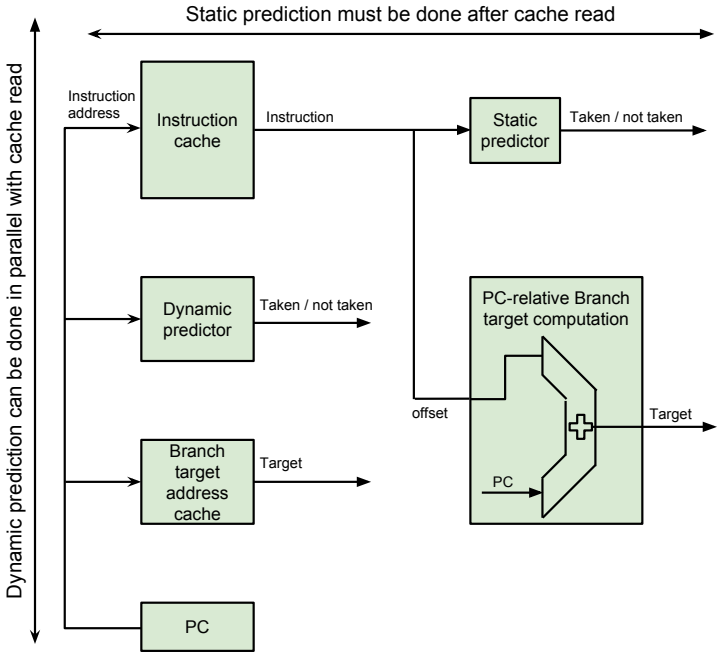

Figure 2.8: A figure demonstrating that Dynamic prediction can be done in parallel with cache read, while static prediction must be done sequentially after cache read.

2.2 Instruction Flow Techniques

A program and its inputs define a stream of instructions. In the eyes of the fetch unit each instruction in the stream computes the next instruction address and causes some side-effect to the internal registers or the external memory. The goal of the fetch unit then is to produce this instruction stream for consumption by the decode and execute units. The challenge faced by the fetch unit is that the next instruction address is not known until the current instruction has computed it. The fetch unit is forced to guess or predict the next instruction address, and recover if the speculatively produced instruction stream does not match the program instruction stream.

2.2.1 Branch Prediction

To speculatively fetch instructions one needs to do branch prediction and target prediction. Branch prediction is predicting whether the fetched instruction will result in a taken branch, while target prediction is predicting where the branch will jump to. Both are needed, and we begin with a discussion on branch prediction.

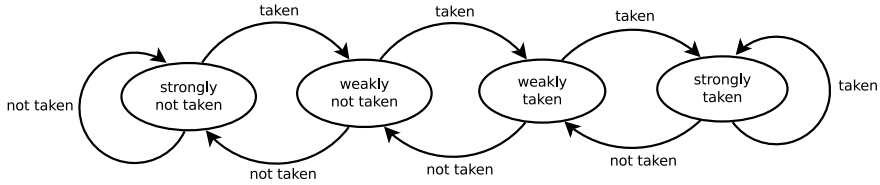

Figure 2.9: State diagram of a 2-bit saturating up/down counter.

Static Branch Prediction

Static branch prediction is done purely based on what kind of branch it is, e.g. whether it branches backwards or forwards, and does not take into account the previous history of the branch. They are useful because they are simple and do not require expensive memories. They can also be fairly accurate, it has been shown that static prediction can achieve an accuracy of 70 % to 80 % over all SPEC benchmarks [CGJ+97].

But static techniques have a disadvantage over dynamic techniques aside from accuracy. As Figure 2.8 illustrates, dynamic prediction can be overlapped with instruction cache read, while static prediction naturally has to come after cache read. This is because dynamic prediction is done based on the instruction address, while static prediction requires the instruction itself. The consequence is that even if you are able to correctly predict a taken branch, you still get a one cycle branch penalty. This one cycle branch penalty can be quite expensive in a non-superpipelined processor. Despite this they are still useful as a fall through in case the dynamic predictor does not remember the branch or as a part of a hybrid predictor.

The simplest static technique is simply predicting all branches taken or all branches not taken. This technique is very simple, but it is not very accurate. More effective techniques are possible when examining the opcode of the instruction. Backwards Taken, Forward Not taken (BTFN) exploits the fact that backward branches are taken around 90% of the time [USS97]. BTFN examines the opcode and the offset and predicts that backward branches are taken, while forward branches are not taken. This technique is able to achieve an accuracy of 65% [USS97].

Dynamic Branch Prediction

To achieve a fetch efficiency that can satisfy a superscalar processor one can do dynamic branch prediction. Common to all dynamic predictors is that they exploit some property of the past to predict the future. Branch prediction is a field in its own right and a thorough survey can not be presented here. Instead we focus on the simple implementations that were feasible to implement in this thesis's time frame.

Smith's algorithm [Smi81] is a branch prediction scheme that is easy to to understand and implement. In Smith's algorithm a hardware table, known as the Branch History Table (BHT), is maintained in a cache-like structure. Each element in the table contains some tag bits to identify the branch, and a counter which represents the recent history of the branch. The counter is implemented as an up/down saturating counter, meaning it behaves as seen in Figure 2.9[3]. When the counter is in the weakly or strongly taken state the branch is predicted to be taken, and otherwise predicted to be not taken. With a counter size of 1 bit Smith's algorithm simply becomes a table of whether a branch was most recently taken or not taken. With 2 bits the accuracy is increased since it correctly handles the apparently common occurrence of one-off changes in branch direction, e.g. TTNTTTNTNTTTNTTTTN.

In theory any Finite-State Machine (FSM), and any mapping from FSM state to predict taken or not taken could be used. It just so happens to be that a saturating counter works very well. For state machines of 2 bits an exhaustive search of the optimal machine is possible, and it was done in 1992 at IBM using the SPEC benchmarks [Nai92]. They found some unintuitive optimal state machines for some benchmarks, but interestingly enough of the 2^{20} possible machines a 2 bit saturating counter was shown to be the optimal strategy for the GCC benchmark and not far from optimal for most benchmarks.

2.2.2 Target Prediction

Predicting taken/not taken correctly is not enough, one also needs to predict the branch target as seen in Figure 2.8. High-performance processors are required to predict branch taken/not taken and what the branch target will be. Target prediction is typically implemented by writing recently computed branch targets to a hardware table indexed by the branch address. It should be noted that this hardware table has two competing names, Branch Target Buffer (BTB) and Branch Target Address Cache (BTAC), we will be using the name BTAC since cache better represents its function than buffer.

A diagram of a simple BTAC implementation can be seen in Figure 2.10. Just like an instruction cache it is indexed into by the program counter and it stores some tag bits to verify a cache hit. It can be implemented either as a direct-mapped cache, set-associative cache, or a fully-associative cache, the function remains the same. When implemented, the table can be merged together with the table for branch prediction or even the instruction cache. With merged tables the indexing hardware, tag space, and tag comparison hardware can be reused.

[3] "Branch prediction 2bit saturating counter" by Afog is licensed under CC BY-SA 3.0

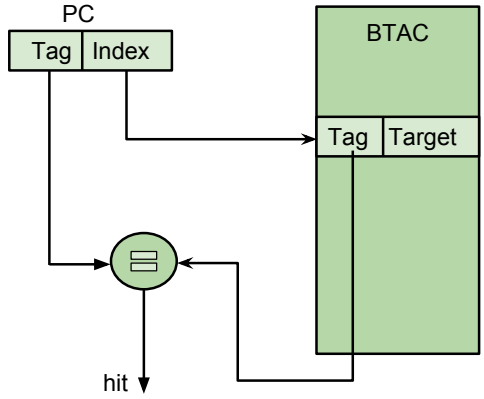

Figure 2.10: A simple view of a Branch Target Address Cache, implemented as a direct-mapped cache.

An interesting optimization can be done when choosing the tag bits. An instruction or data cache typically chooses the least significant bits of an address for the index and the remaining 20 bits or so as tag bits. But as opposed to an instruction or data cache, the BTAC does not have to set the hit flag correctly every single time for the program to be semantically correct. The BTAC can instead store a much smaller subset of the tag bits and accept an occasional false positive. This technique is known as Partial resolution, and it has been shown that a 99.9% accuracy is obtainable with only 2 tag bits for a 512 entry BTAC [Fag97].

2.2.3 Return Address Stack

The BHT and Branch Target Address Cache (BTAC) are good at predicting regular branches. However, there is a particular type of branch instruction, subroutine return instructions, that they tend to mispredict very often. Subroutine return instructions occur frequently in programs written in high-level languages. For example, more than 15% of the branches in the SPEC95 benchmarks are caused by returns [HP11].

Consider the example in Figure 2.11. The BHT would correctly predict the `printf()` functions as taken. When the first `printf` function returns, the BTAC notes the address on line 6 as the predicted address for the return instruction in the `printf` subroutine, and execution continues on line 6. When the return instruction of the second `printf` subroutine call is fetched, the BTAC will predict the address on line 6 as the return address, which is wrong (the correct address is the next address after the `printf` on line 6). The pipeline will be flushed and the BTAC will update the predicted return address to line 7. This will again lead to a mispredict when returning from the first `printf` subroutine during the next iteration of the loop.

```
1 int main()
2 {
3     for (int i = 0; i < 10; ++i)
4     {
5         printf("i   is: \%d\n", i   );
6         printf("i+1 is: \%d\n", i+1);
7     }
8 }
```

Figure 2.11: Example where the BTAC will achieve poor performance

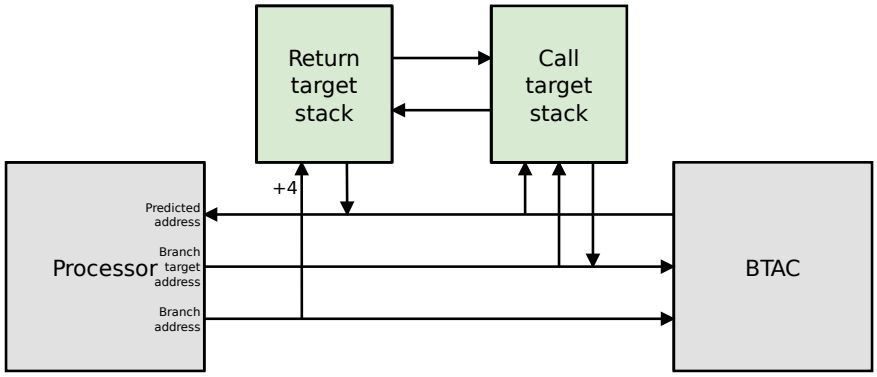

Figure 2.12: Structure of a RAS

Continuously updating the BTAC leads to mispredicts on every `printf` return, every iteration of the loop. Clearly, there is a potential for improvement.

A RAS is a branch predictor that is designed specifically to predict subroutine return instructions. The original scheme suggested by Kaeli and Emma [KE91] uses two stacks, as depicted in Figure 2.12. The call target stack holds the target address of a subroutine call instruction, i.e. the address of the first instruction of the subroutine called. The return target stack stores the return address of the called subroutine. In addition to regular stack operations, e.g. pushing and popping, the stacks can also be searched for a particular value. The stacks are tightly coupled, so a match in one of the stacks will make the other stack output the value at the same index as the match occurred. In addition, the BTAC contains a bit for each index which indicates that it is a entry for a return instruction. Normally, the value at an index in the BTAC contains a branch target. When the special bit is set, the value represents the address of the first instruction in the subroutine, which is the same as the call target stack.

Before we explain the scheme in detail, we will outline its main principles. When a subroutine call is executed, the call target address and return target address are pushed onto the stacks and the BTAC updated. When a subroutine return is executed, the BTAC is searched through after a match. On a miss, the BTAC is written and the special bit set. The value written, which originally is the branch target of the return instruction, is replaced with the call target address of the subroutine (i.e. address of the first instruction in the subroutine) which the return instruction belongs to. If there is a match in the BTAC for a return instruction, the value in the BTAC is used to look up an entry with the same value in the call target stack. If it exists, the corresponding index in the return target stack is read and replaces the value on the predicted address bus.

To see how it works we will do one iteration of the loop in Figure 2.11. When the call to `printf` at address 5 is executed, it is written into the BTAC. At the same time the call address is written to the call target stack, and the return address written into the return target stack. Assuming the `printf` subroutine starts at address 100, the values written are 100 and 6 respectively. When the `printf` return instruction at address 150 is executed, the branch target address (6) is searched for in the return target stack. On a match, the same entry is indexed in the call target stack and the value, in this case 100, replaces the branch target address arriving from the processor. The value gets written into the BTAC. The special bit in the BTAC is also set to indicate that this is a special entry. At this point, the BTAC entry at index 150 contains the value 100. On the next call to `printf` at address 6, the same process is repeated and the call target and return target stacks pushes the values 100 and 7. When the `printf` return instruction is fetched, the BTAC looks up the entry at the instruction address (150) and finds the special bit set. This makes the call target stack search for an entry with the same value as the BTAC entry (100). The call target stack reports a hit, and makes the return target stack replace the predicted address with the value (7) at the same index as the call target stack matched, and the core correctly continues to fetch instructions from address 7.

The RAS only require a few number of entries per stack (4 - 16). Kaeli and Emma showed only a small increase in performance doubling the stack size from 5 to 10 [KE91]. The few number of entries makes the RAS cheap in terms of hardware. The low cost combined with the high accuracy and speedup makes them very common in high performance processor designs, e.g. Alpha 21264 [alp99] or ARM Cortex-A15 [cor11].

2.3 The Amber Project And Amber25

The Amber25 core is the base which TA is built upon. To help the reader better understand the internals of TA, we give an introduction to the original Amber core.

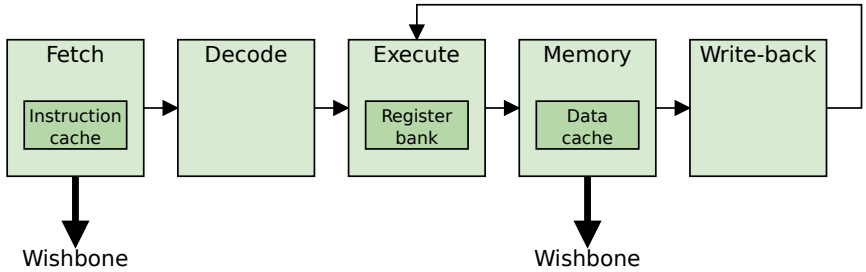

Figure 2.13: Amber25 pipeline overview

This way, the required modifications applied to Amber will become more evident when we introduce the micro-architecture of TA.

The Amber project is a complete open source embedded system written in Verilog by Conor Santifort [San]. The system contains an ARM processor core and various peripherals like an interrupt controller and a timer, together with peripherals for external communication like Ethernet and Universal Asynchronous Receiver/-Transmitter (UART). All units are compatible with the wishbone interface which is an open source interconnect architecture. The Amber processor core is a 32-bit ARM RISC processor which supports the ARMv2a instruction set. It is an in-order scalar processor, meaning that all instructions are executed in order and the peak throughput is 1 Instructions Per Cycle (IPC). The core exists in a 3-stage and 5-stage version, named Amber23 and Amber25. We will call attention to Amber25 since it is the version that TA builds upon. The Amber project was chosen for the SHMAC project because it was considered to be the most stable and mature open source ARM processor core, having been verified by booting Linux and passing a test suite of 59 assembly programs.

The five stages of Amber25 are fetch, decode, execute, memory and write-back. Figure 2.13 visualizes the dataflow between the pipeline stages. There are separate caches for instructions and data. The former is placed in the fetch stage and the latter in the memory stage. The first stage fetches instructions from the memory through the wishbone interface. The Wishbone interface is capable of transferring an entire cache line (16 bytes, four words) of data per memory request. The decode stage takes as input an instruction, decodes it, and generates control signals which are applied to the execute stage in the next cycle. It also contains a state machine for handling multi-cycle instructions. The Execute stage reads, computes a result, and writes the result back to the register bank based on the control signals from the decode stage. This is where Amber25 differs from classical 5-stage RISC pipeline designs. Amber25 writes the result of register operand instructions to the register bank in the execute stage, whereas regular 5-stage pipelines writes them in the write-back stage. Register

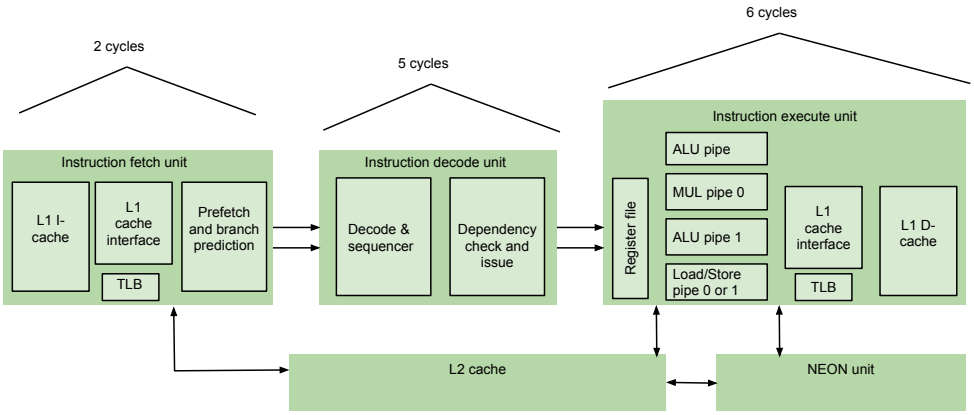

Figure 2.14: Cortex-A8 architectural overview. Adapted from the figure in [Sta13] on page 618.

operand instructions can be executed in a single cycle in the common case. Load instructions can also be executed without stalling if the data is in the cache and no RAW hazards occur. Amber25 implements a cache write-through policy. That is, all store instructions are immediately written to main memory.

2.4 ARM Cortex-A8

The ARM Cortex-A8 is an ARMv7-A processor designed by ARM Holdings, it can be found in well-known products such as the iPad [ipa], iPhone 4 [iph], and Apple TV [app]. Being ARMv7-A it is considerably more complex than Amber which implements the old ARMv2a. We will be focusing on the general micro-architectural techniques relevant for integer performance and be glossing over the details of the less relevant features to ARMv2a such as double-precision floating point arithmetic, SIMD instructions, Thumb instructions, and memory management. A review of the Cortex-A8 will give insight into the challenges a superscalar ARM processor faces and how the ARM engineers solved them.

2.4.1 Architectural Overview

Being targeted for the mobile platform, Cortex-A8 strives for energy efficiency and stays away from power hungry out-of-order techniques. Instead it exploits ILP through superpipelining and two-way superscalar execution. Figure 2.14 is an overview of the architecture and shows that the Cortex-A8 is superpipelined with 13 cycles, and that it is two-way superscalar at every stage. The architecture can be

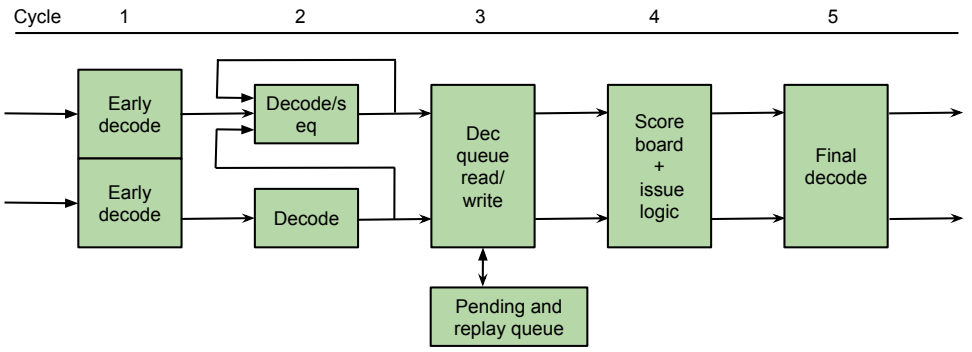

Figure 2.15: Cortex-A8 decode unit overview. Adapted from the figure in [Sta13] on page 619.

divided into three logical parts; instruction fetch, instruction decode, and instruction execute. These three parts will be discussed in the following sections.

2.4.2 Instruction Fetch Unit

The instruction fetch unit contains the instruction cache, TLB, and the branch predictor, as seen in Figure 2.14.

Superpipelining with a 13-cycle pipeline allows very high frequency performance, but mispredicted branches now incur a penalty of 13-cycles. To mitigate this, an advanced branch predictor is used which ARM claims achieves an accuracy of >95% [a8o]. The branch predictor uses a global history buffer, a BTAC, and a RAS.

An instruction spends two cycles going through this unit. In the first cycle the instruction is read from the instruction cache. As the instruction cache is read, the address is also sent to the branch predictor which attempts to predict the next instruction to be executed. In the second cycle the instruction is written to an instruction queue which feeds the decode unit. The instruction fetch unit can read a total of four instructions each cycle and queue a total of 12 instructions in the instruction queue.

2.4.3 Instruction Decode Unit

The decode unit reads instructions in pairs from the First In, First Out (FIFO)-buffer and issues them in pairs to the execute unit. Stallings describes the decode unit as a 5-stage two-way pipeline as depicted in Figure 2.15 [Sta13].

In the first two stages the instruction type and register operands and destination are decoded before being buffered in a new FIFO-queue at stage three. If a multiple transfer type instruction is found it will be decoded into a series of micro-coded transfer instructions by a state machine. The state machine is visualized in Figure 2.15 by a block that feeds back into itself.

It is in stage 4 that hazards are detected and instructions are issued. WAR hazards do not exist since instructions are issued in-order and retire in-order, and WAW hazards are trivially prevented by not dual-issuing instructions that write to the same register, which leaves only RAW hazards. RAW hazards are detected by the scoreboard in stage 4. The scoreboard issues zero, one, or two instructions from the queue in stage three each cycle. Without the queue it would not be able to issue two instructions after issuing one in the previous cycle.

When using a scoreboard the A8 predicts that a dependency will be met in time and issues an instruction accordingly. But sometimes the prediction fails and the value is not ready for forwarding in time. This can happen on L1 data cache misses. On missing in the L1 cache, the entire execute unit pipeline is flushed. To prevent losing the instructions that were flushed a replay queue is used, as seen in Figure 2.15. The replay queue contains instructions that have been issued, but have not retired yet (written to the register bank / cache). When the L1 cache miss has been resolved the instructions are replayed from the replay queue.

2.4.4 Instruction Execute Unit

John and Rubi describe the execute unit as depicted in Figure 2.16 [JR07]. Central to the design is that there are two logical execution pipelines named pipe 0 and pipe 1, while there are in fact four physical pipelines with different capabilities. The scheduler enforces a set of rules that must hold for the logical pipelines.

- A logical pipeline can have one or zero instructions issued to it each cycle.
- Pipe 1 cannot be issued an instruction unless pipe 0 is also issued an instruction.
- You cannot issue the instructions A, B to the logical pipelines 0, and 1 respectively where B precedes A in the program flow.

The execution pipelines are deeply pipelined with the barrel shifter and the ALU working in their own cycles, whereas Amber executes shifting and arithmetic in the same cycle. There are two ALU pipelines, but only one of MUL and Load/store pipelines, allowing the most common instructions to be executed two-way. The stages are heavily bypassed and the scheduler will issue instructions if it knows that the value can be calculated and forwarded in time.

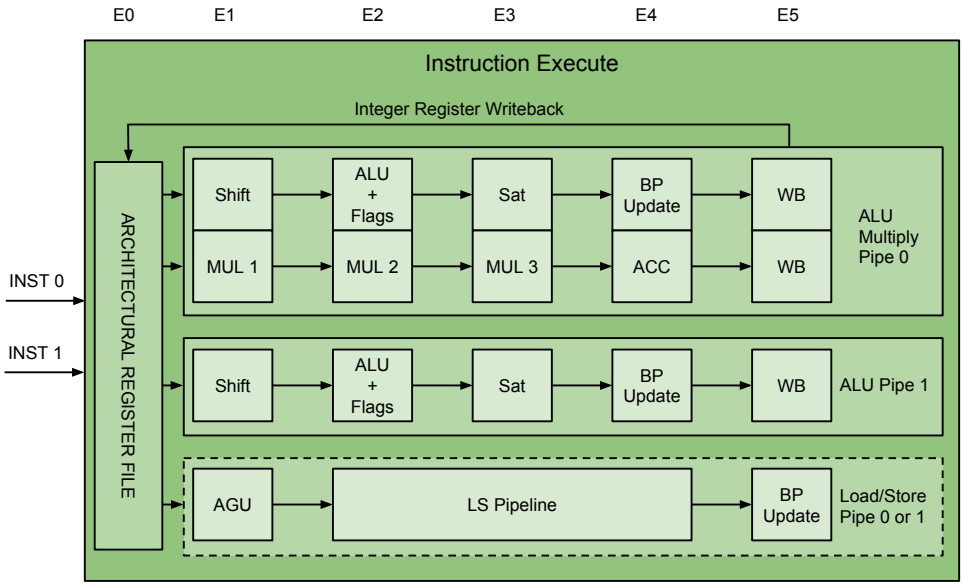

Figure 2.16: Cortex-A8 execute unit overview. Adapted from the figure in [JR07] on page 94.

Chapter 3
Turbo Amber

In this chapter we introduce Turbo Amber. We begin by discussing which performance improvements were considered and why we chose the implemented improvements. Following the design decisions we give a high-level overview of the new fetch stage. And finally we dive deeper into the details of each component in the high-performance fetch stage. Upon reading this chapter an understanding of how the performance has been improved should be obtained.

3.1 Design Decisions

At the beginning of our project we had to decide between extending Amber or starting from scratch. The design possibilities considered can be seen in Figure 3.1. If we had started from scratch, we could have chosen between implementing a scalar superpipelined architecture, a superscalar architecture (like the Cortex-A8), or an

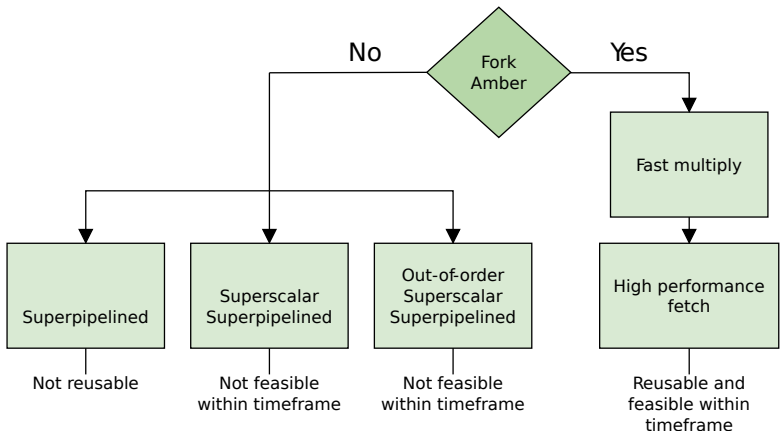

Figure 3.1: The design options considered

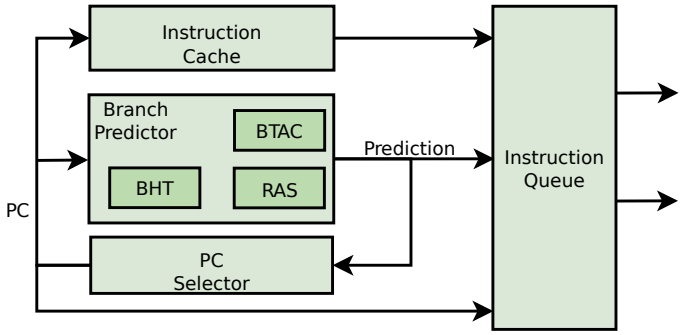

Figure 3.2: A diagram of the fetch stage showing the relationship between the instruction queue, the predictors and how the next PC is determined.

out-of-order architecture. The ideal solution would have been a superscalar out-of-order architecture, but this would have taken too much time and effort to implement and fully verify. Instead we chose to prioritize a well verified solution over a highly complex solution. Through discussion with EECS researchers it was decided that by focusing on the front-end, the final product would be reusable as well as provide an increase in performance. The goal for the SHMAC project is to have a diverse range of high performance cores, and common for high performance cores is that they require a high performance front-end. Since the front-end can be relatively independent from the back-end, future SHMAC projects can reuse our front-end in other processor designs.

As a side note, the first performance improvement attacked was suggested by Håkon Wikene [Wik13]. His benchmarking showed that the multiplier was hurting performance. Even though it was not a part of the front-end, there was a small amount of effort associated, and a big improvement in performance by replacing the multiplier.

3.2 Architectural Overview

TA's fetch stage can be seen in Figure 3.2. We moved the Program Counter (PC) from the register bank and into the fetch stage. This allows the fetch stage to be decoupled from the rest of the pipeline, because it can update the PC without coordinating with the rest of the pipeline. The decoupling allows the fetch stage to do work when the other stages are stalling and vice versa. If a branch mispredict occurs the execute stage will notify the fetch stage and pass along the correct PC value. The fetch stage then updates its local PC and continues execution.

The fetch stage can be viewed as a state machine, where the PC is the state and every cycle the next PC is predicted and updated. In parallel to this prediction,

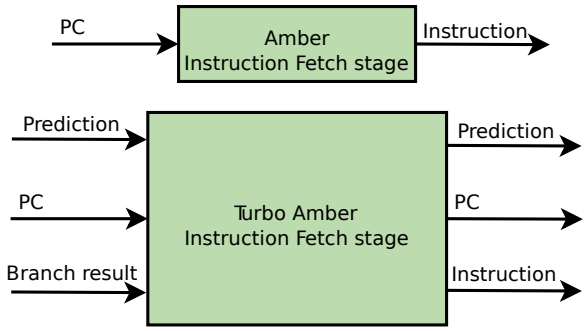

Figure 3.3: The differences between Amber and TA when viewed as a black box

instructions are read from the cache and written to the instruction queue. The prediction information is also used to discard instructions that are predicted to be not executed due to taken branches.

From Figure 3.3 we can see that TA passes the prediction made and the PC for each instruction into the pipeline. The PC value is passed because some instructions use it as an operand. This was not necessary before because the PC was placed in the register bank inside the execute stage. The prediction data is passed because all branch predictions must be compared to the actual branch outcome which are computed in the execute stage. This comparison is used to detect and handle branch mispredictions, and is also used to update the branch predictor. By sending the prediction data with each instruction, the instruction fetch stage does not have to remember the recent branch predictions made. This simplifies the design of the instruction fetch stage.

The fetch stage mostly works on a cache block granularity: It receives cache blocks from memory, it is capable of writing an entire cache block into the instruction queue, and it is also capable of writing and reading a cache block into/from the cache. The level of granularity makes the fetch stage superscalar by nature. It is capable of providing up to four instructions (one cache block) per cycle.

3.3 Instruction Queue

The instruction queue is a temporary storage for instructions located between the fetch and decode stage. This is where the instructions are held before they are issued. It enables a producer-consumer pattern between the fetch and decode stage. The fetch stage “produces” instructions by fetching them from memory, while the decode stage “consumes” instructions by decoding and issuing them.

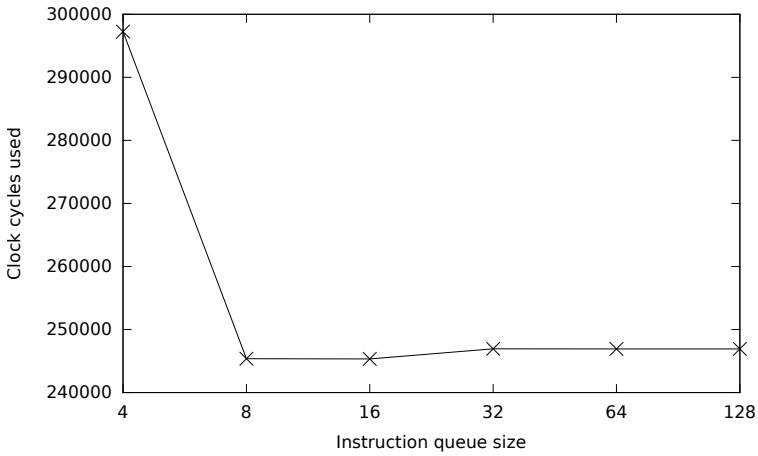

Figure 3.4: Impact of instruction queue size on performance

Instructions are produced at an uneven rate because of cache behavior and consumed at an uneven rate because of multi-cycle instructions. A cache hit can provide an instruction on the next cycle, while a cache miss can make the access time increase significantly. Some instructions also take several cycles to execute. Any of these scenarios would make either the fetch unit stall because the back-end is overloaded, or the back-end stall because of starvation. The instruction queue can increase instruction throughput by providing a smooth stream of instructions to the back-end. The fetch stage can avoid stalling when the back-end is busy, because the instructions can be placed in the instruction queue. It also allows the fetch stage to prefetch instructions which can reduce the amount of stalling required, or even remove it completely, in the case of a cache miss. The instruction queue also simplifies the fetch stage interface and makes it more modular. This makes it easier to integrate different processor back-ends.

The instruction queue is a FIFO hardware structure implemented as a circular buffer. It is capable of receiving up to four writes and providing two reads per clock cycle. The number of instructions pushed will vary, for example the target of a branch may point to the last instruction in a cache block. Hence, the three foremost instructions should be discarded and only the last one written to the instruction queue. The instruction queue must also handle a varying number of read instructions, ranging from none, if the decode stage is busy decoding a multi-cycle instruction, to two, supporting a 2-way superscalar back-end. The ideal number of entries in the instruction queue depends on several factors, e.g. the particular benchmark, type of back-end, or memory latency. To facilitate this, the number of entries in the instruction queue are made configurable, and currently set to 8. The decision was

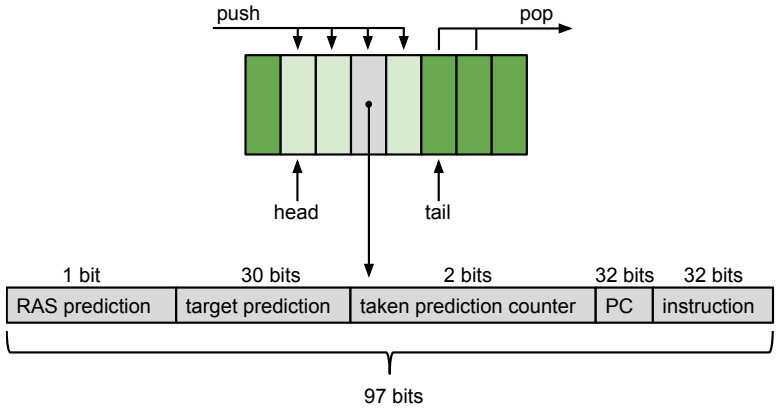

Figure 3.5: A figure demonstrating an instruction queue with room for 8 instructions and showing what each FIFO entry contains. The figure shows that four entries can be pushed simultaneously (one for each instruction in a cache block), and that two entries can be popped simultaneously. It also shows that each instruction brings with it a lot of metadata. In addition to the actual instruction one needs to pass along it's address and the prediction made about the instruction. This prediction information is necessary to detect branch mispredictions.

made empirically by running the Dhrystone benchmark multiple times with varying instruction queue size. The result can be seen in Figure 3.4. It shows that, in our case, increasing the instruction queue size to 16 entries has no effect on performance, and increasing it beyond 16 decreases performance. This is because the front-end prefetches more aggressively, which creates more bus contention.

An entry in the instruction queue contains three types of information. The instruction itself is stored in the lowest part of the entry, as visualized in Figure 3.5. The next 32 bits are occupied by the instruction address, or PC. An instruction entry also contains prediction data which is used to check if a branch was correctly predicted.

3.4 Branch History Table

With an instruction queue we can prefetch several instructions before they are needed. But if we always predict branches to be not taken, as Amber does, many of these instructions will be mispredictions. This is why we have implemented a branch prediction scheme that remembers the past history of branches. This section presents the design that was implemented.

The branch predictor has an architecture as seen in Figure 3.6. The BHT predicts the branch direction of each of the 4 instructions being read from the instruction

cache. And it does so solely on the address of the instructions because waiting for the cache to be read would take too long for a single-cycle fetch unit. It is essentially a direct-mapped cache of 2-bit saturating counters that encode the branch prediction state. The cache consists of 256 cache blocks containing 4 counters and tag bits, allowing a total of 1024 different branches to be remembered at a time. It could have been organized with any set-associativity and still functioned correctly but a direct-mapped organization was chosen because in FPGAs large simple memories are cheaper than smaller memories with complex logic and because a direct-mapped cache is easier to implement [WBR11].

To understand the branch history table and branch target address cache, one must first understand how an instruction address is partitioned. See Figure 3.7 for a diagram showing the different fields. Since ARMv4T ISA requires that instructions and data are word-aligned in memory we know that the byte offset has to be `00` for all valid addresses. This saves us some space when identifying an instruction. Since each cache block in Amber can contain 4 instructions the block offset consists of 2 bits that identify where in a cache block the instruction is located. The remaining 28 bits identify cache blocks.

3.5 Branch Target Address Cache

The BTAC is architecturally very similar to the BHT, they are both in essence direct-mapped caches. Except that the BHT stores branch direction prediction information, and the BTAC instead stores branch target addresses. They both make four predictions each cycle, one for each instruction being read from a cache block. But where the BHT has a reasonable default for when no prediction information can be found (predict not taken), the BTAC has no reasonable default branch target and must instead signal a failure to predict on BTAC miss.

The BTAC stores tags and prediction data in blocks of four just like the BHT, except that the BHT needs only store 2 bits of prediction data per branch, while the BTAC needs to store an entire 30 bit address. A naive solution would require that a BTAC that predicts the same amount of branches as a BHT be more than twice as large. To keep the block RAM usage down we use partial resolution as described in the background material in Section 2.2.2. With this scheme we reduce the number of tag bits stored from 20 bits to 2 bits, giving us a memory usage of 32 bits per branch instead of 50 bits per branch. The downside is that occasionally we will get a false positive where the 2 least significant bits of the tag will match but the branch target in fact belongs to another address.

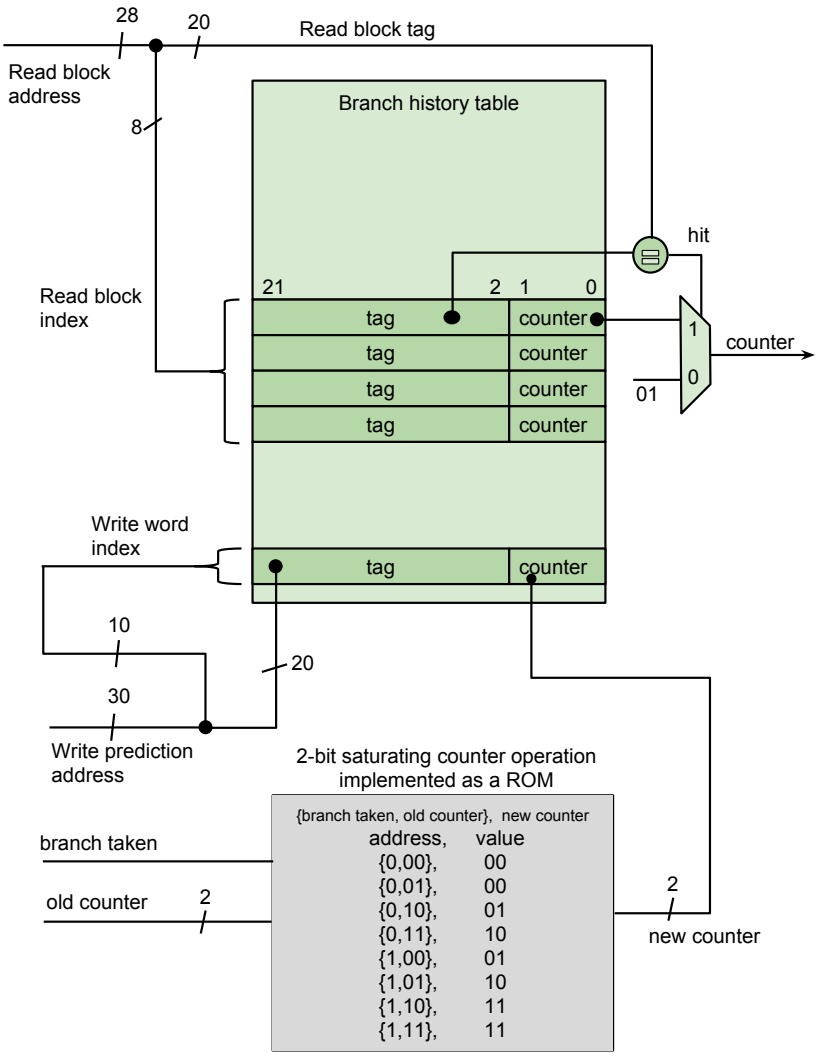

Figure 3.6: A superscalar branch history table capable of making a prediction for each of the four instructions read from the instruction cache and updating itself with one new branch each cycle. Only one tag-comparison is depicted here, but there are in fact 4 tag comparators, one for each tag in a block.

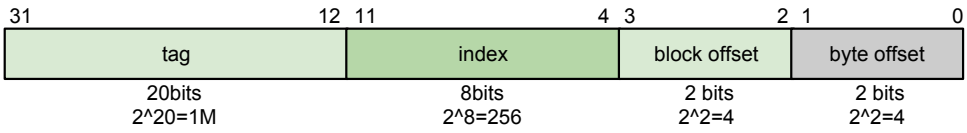

Figure 3.7: An instruction address as seen by the BHT and BTAC.

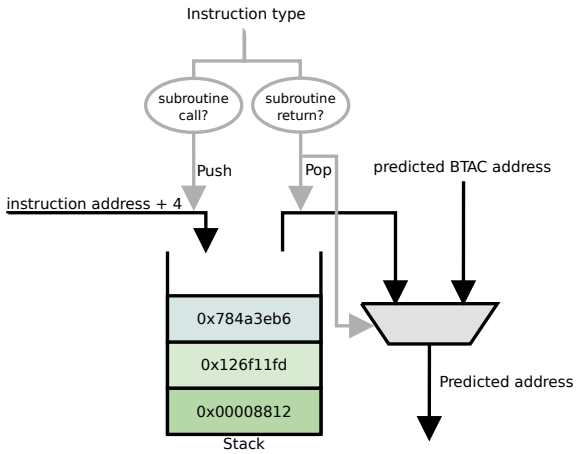

Figure 3.8: Structure of Turbo Amber's RAS

3.6 Return Address Stack

The RAS is a type of branch predictor that is specially made for predicting subroutine returns. The implementation in subsection 2.2.3 used two searchable stacks together with the BHT. TA uses a simpler RAS implementation.

A simplified overview of TA's RAS is depicted in Figure 3.8. It consists of an instruction stream monitor and a stack. The instruction stream monitor is simple and the storage size is small, which makes it cheap in terms of hardware. If it encounters a subroutine call instruction, it pushes the address of the subsequent instruction (i.e. subroutine call instruction address + 4) onto the stack. If the instruction stream monitor encounters a return from subroutine instruction, it predicts next PC to be the address of the top stack element, and pops it off the stack.

The RAS handles nested function calls as well, thanks to the stack structure used as storage. If a subroutine is called within a subroutine, the RAS pushes the return address of the inner function call to the top of the stack, and pops it off when it returns. This will make the return address of the outer function call the top stack

element which will be popped correctly when the function returns. The RAS only supports a finite number of nested function calls, bounded by the size of the stack. Stack overflow may occur if the function call nesting is deeper than the number of entries in the stack. If this happens, the oldest entry in the stack is truncated and the new entry pushed onto the top of the stack. This will later result in a misprediction when the instruction monitor encounters the return instruction, which belongs to the subroutine, whose return address was truncated.

To see how the RAS of TA works we return to the example in Figure 2.11 The RAS will push the address of line 6 onto the stack when it encounters the `printf` on line 5, and correctly continue execution on line 6 when it returns. This will happen for the `printf` on line 6 as well. This shows how the RAS can improve performance when a function is called from multiple places in a program.

In addition to improving the performance of function calls, the RAS also indirectly improves the performance of the other branches. Since the RAS handles prediction of all function calls and returns we can omit writing them to the BHT and BTAC tables. This saves a lot of space that can be used to make predictions about a larger window of branches.

The fact that TA's RAS implementation is simplified does not come without consequences. It may suffer from stack misalignment if the RAS mistakenly pushes or pops an address from the stack. Then the subsequent predictions will be incorrect and hence incur a performance penalty. This will not happen in the scheme described in subsection 2.2.3 as the stack is searched rather than blindly popping the top stack element. However, our experiments show that this happens rarely and does not diminish the performance benefits from TA's RAS for the tested cases.

3.7 Fast Multiply

Håkon Wikene benchmarked Amber and reported multiplication as one of the areas where Amber had poor performance [Wik13]. Amber was using the space-efficient Booth's algorithm which only requires a 32-bit adder in resources, but requires a total of 33 cycles. We implemented a fast multiplier because it requires minimal changes to the existing pipeline and because of the large speedup possibility.

To create a fast multiplier we used the Xilinx CORE™Generator tool for generating commonly used components such as multipliers and memories using FPGA-native primitives. The generator could generate highly configurable multipliers with near-arbitrary operand lengths and result lengths. The generator could configure the number of cycles spent multiplying, allowing us to make the trade-off we needed between clock-frequency and the number of cycles required. One could also configure

whether to use LUTs or DSP slices to build the multiplier.

Multipliers with different pipeline lengths and clock frequencies were integrated into SHMAC and synthesized and the multiplier that required the fewest cycles while not hurting the clock cycle time was found to be a 2-cycle multiplier. Multipliers that were built of LUTs and multipliers that were built of DSP slices were generated and it was found, to our surprise, that the LUT-based 2-cylcle multiplier could uphold timing requirements while the DSP-based multiplier could not. Originally we believed that a DSP-based multiplier would always be faster than a LUT-based multiplier, upon synthesizing a 2-cycle DSP multiplier we discovered that an entire 50 % of the delay consisted of routing. This explains why a DSP based multiplier can be faster. If many DSP slices are used the signals must be routed far across the chip to reach each DSP, while a LUT-based multiplier can be contained to a concentrated group of LUTs. The 2-cycle multiplier is approximately 16 times faster than the original and uses 1156 LUTs.

Chapter 4
Evaluation

In this chapter we present the four different kinds of results we have gathered to assure that the assignment requirements are satisfied:

- We measured the FPGA-resource usage to be able to evaluate if TA is area efficient and resource balanced.
- We verified that the processor correctly implements the ISA.
- We verified that the processor performance was as expected under controlled environments using micro-benchmarks.
- We attempt to measure the "true" performance of the system by running large benchmarking programs.

4.1 Resource Usage

In this section we present the FPGA resource usage and compare Turbo Amber to Amber. The FPGA resources measured are logic slices and Block RAM[1]. We measured resources at the tile level since a tile is the unit that a programmer or layout architect can choose between. When listing resource usage we present the FPGA resource availability in the Xilinx Virtex-5 XC5VLX330-ff1760-1 FPGA, used by SHMAC, to give context to how scarce the resource is.

Measuring an FPGA-design's resource usage is not as straightforward as it is in an Application-Specific Integrated Circuit (ASIC) design, where one could simply measure the area required for a given process technology. An FPGA is a heterogeneous computing substrate and has many different resources available in different quantities. We chose to measure Block RAM, and total slices occupied. We believe this strikes a balance between being overly detailed about each resource category and omitting

[1]DSP slices are omitted because neither TA nor Amber uses them.

Resource	Amber	Turbo Amber	Increase
DSPs	0	0	N/A
Block RAM	108 KB	342 KB	217 %
Logic slices	2412	4092	70 %

Table 4.1: Increased FPGA resources relative to Amber.

Resource	Turbo Amber	Virtex5	Utilization
Block RAM	342 KB	10,368 KB	3.3 %
Logic slices	4092	51840	7.9 %

Table 4.2: Turbo Amber's FPGA resource utilization of a Virtex5

important resources. Note that we did not report LUT's or registers directly as they are contained within slices.

For reproducibility and transparency reasons we present the details of how the designs were synthesized. The synthesis was done by creating a Xilinx ISE 14.7 project with `amber_wrapper.v` and `turbo_amber_wrapper.v` as top level modules. Note that this excludes the router and network interface components. The entire tile was not synthesized because the router and network interface was unchanged. No synthesis flags were changed from their defaults, leaving the design goals to be balanced between area usage and speed optimization, and the optimization effort normal. For a detailed list of the flags used and fine-grained resource usage see Appendix A.

Table 4.1 shows that Turbo Amber uses a lot more Block RAM than Amber does. This was expected as the branch prediction mechanism is basically a large memory for remembering branches. The slice count is the closest heuristic we had to area usage and represents the overall registers + combinational logic increase. We considered the increase of 70 % to be within acceptable limits.

To know if using 342 KB of RAM is excessive or not one must compare the resource usage to how scarce the resource is. As we can see from the FPGA utilization in Table 4.2 the Block RAM usage is far from excessive, and if we had scaled up the number of Turbo Amber tiles in SHMAC we would run out of ordinary slices long before we were limited by RAM. In conclusion, the resource usage is moderate and the usage of heterogeneous resources is balanced.

4.2 Verification

In this section we present the work done to prove that Turbo Amber conforms to the ARMv4T instruction set. Since there is no publicly available test suite to prove conformance, as there is with other protocol standards, we proved conformance by running real-world applications. We also argue that the 79 assembly tests found in the Amber Test framework give confidence to its correctness.

We would like to quote the ARM verification engineers Nitin Sharma and Bryan Dickman from their article "Verifying the ARM Core" [SD01].

> At ARM, we develop test cases as self-checking assembler sequences. We then replay these code sequences on a simple simulation testbench consisting of the ARM CPU, a simple memory model, and some simple memory-mapped peripherals. [...] We believe that the coverage attained by this deterministic approach is very high, but not complete. Thus we need to apply other methodologies to fill the coverage holes. Often, running real application code examples or booting an OS on the simulation model will flush out cases missed by the above test generation approaches. [...] [B]ooting a real RTOS such as Windows CE on the simulation model is a great demonstration that the CPU core is actually working.

As a first line of defense against bugs we used the same assembly test simulation setup as the ARM engineers describe. The Amber test framework had a test suite of 79 assembly tests checking various instructions and hazard possibilities, ensuring ARMv4t conformance. The test framework is structured just as the ARM engineers describe it with a simulation testbench, some memory-mapped peripherals to support printf, and a simple memory model. This verification tool is relatively lightweight, with simulation of all assembly tests taking only a few minutes, thus it can be used as a part of daily regression testing.

Table B.1 in the appendix gives a description of each assembly test in the Amber Test framework. Most assembly tests and descriptions were inherited from the original Amber project. The exceptions are the tests added by the Linux team used to show ARMv4 conformance and a handful of assembly tests that produce bugs discovered by us during verification. The table gives an idea of how extensive this testing was.

For further verification we used the work of the Linux team and Magnus Walstad to run large applications on top of operating systems. This is the greatest proof of correctness as it tests more instruction combinations and hazard possibilities than one could possibly write hand-written assembly for. On the Linux side we successfully booted Linux and ran a small set of benchmarks on top of Linux as reported in

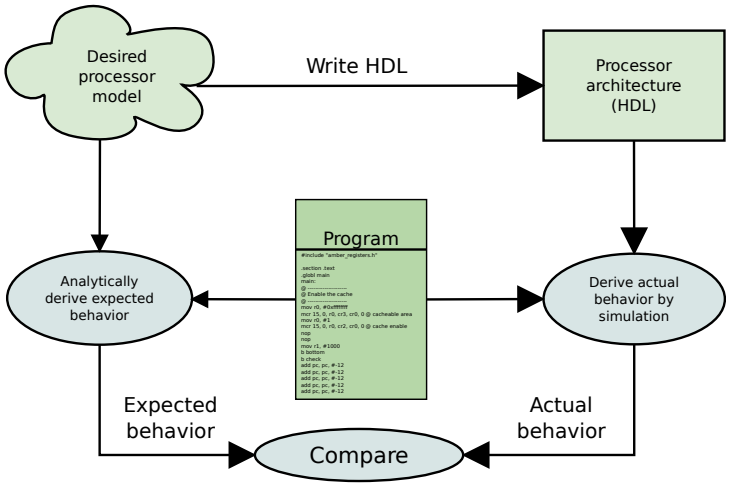

Figure 4.1: Analyzing and verifying processor behavior with micro-benchmarks

section 4.4. We also used Magnus Walstad's work on freeRTOS and OMPi to boot an RTOS and ran a large binary [Wal14].

Running applications on these operating systems is as the ARM engineers put it "a great demonstration that the CPU core is actually working.".

4.3 Micro-Benchmarks

In this section we analyzed and verified the processor behavior through running a suite of micro-benchmarks designed to utilize specific processor features. Verification of the processor, not just in terms of correctness, but also in that its performance is as expected, was an important task. The processor design can contain coding errors or there could be scenarios that could lead to undesired behavior. Some types of behavior can produce the correct result, but still use an unnecessary amount of cycles which would hurt performance.

Figure 4.1 illustrates the process of how we analyzed and verified the processor performance. Before we began implementation, we designed a processor model. This model describes the desired behavior of the processor. It is this model that is described in chapter 3. From this, we implemented the processor architecture in a Hardware Description Language (HDL). Since the model is detailed enough to analytically derive the expected behavior of short instruction streams on a cycle-to-cycle level, we could use it for performance verification by deriving expected behavior for a benchmark. Then we could simulate the processor architecture with the same

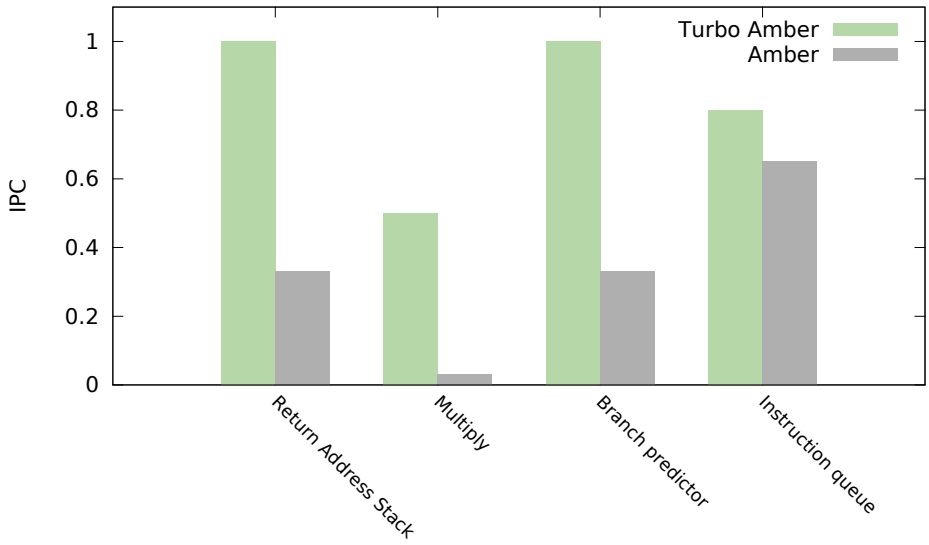

Figure 4.2: Summary of the performance of micro-benchmarks

benchmark and analyze its behavior. The results from the analytical derivation and the simulation could then be compared to ensure that the processor model and architecture comply.

Analytically deriving the expected behavior from the processor model for large programs could be hard and tedious, and is also prone to errors. For this reason we chose to use micro-benchmarks. Our micro-benchmarks are small synthetic programs made to utilize and isolate the performance of one specific feature of the processor. They usually contain a relatively small amount of instructions, and their control flow graphs are simple and easy to reason about.

To show that our processor behaved as expected we introduced several micro-benchmarks and analytically derived their expected behavior and performance in terms of IPC. We then ran the micro-benchmarks in simulation and showed that the processor model complies with the processor architecture. We were simulating with caches enabled and a 3 cycle memory latency. This process was repeated for the original Amber25 core to show that the feature utilized had a positive impact on performance.

A summary of the micro-benchmark results can be seen in Figure 4.2.

```
1       mov r1, #100     // 100 iterations
2   top:
3       bl func1         // 1 (Call the function func1)
4       bl func1         // 2
5       bl func1         // 3
6       ...
7       bl func1         // 199
8       bl func1         // 200
9       subs r1, r1, #1
10      bne top          // Loop 100 iterations
11      b testpass       // End the microbenchmark
12
13  func1:
14      mov pc, lr       // Function return
```

Figure 4.3: Return Address Stack micro-benchmark

4.3.1 Return Address Stack

The first micro-benchmark targets the RAS. The RAS is very good at predicting subroutine calls and returns, and hence, a micro-benchmark which contains many function calls can utilize the RAS to its utmost.

Figure 4.3 shows the RAS micro-benchmark. The `bl` (branch and link) instruction calls the subroutine `func1` which immediately returns. There is in total 200 `bl` instructions and each executes `func1` consisting of 1 instruction. The micro-benchmark loops 100 times to mitigate the impact of the compulsory cache miss penalty. This makes up for a total of about (200 bl + 200 ret) $\times$ 100 loops = 40 000 instructions executed when running the benchmark.

The RAS is able to correctly predict all the subroutine calls and returns, and since the predictions are done in the fetch stage the processor will immediately start execution from the correct address. Hence, there is no flushing or stalling associated with function calls or returns, and we should be able to achieve an IPC of 1. The simulation completed in 40 910 cycles which gives an IPC of 40 000 instructions / 40 910 cycles = 0.98. The reason for not achieving an IPC of 1.0 is due to the cache initialization time, the compulsory cache misses and the loop overhead.

We predicted that executing the same benchmark on Amber25 would not yield the same performance due to the lack of RAS. Since Amber25 always predicts branch not taken it would mispredict on every single bl and return instruction. Mispredicts incur a 3 cycle penalty since they are first discovered in the execute stage. It would therefore use 3 times the number of cycles as Turbo Amber used on the same amount

```
1       mov r1, #100          // 100 iterations
2   top:
3       mla r2, r3, r3, r3    // 1
4       mla r2, r3, r3, r3    // 2
5       ...
6       mla r2, r3, r3, r3    // 200
7       subs r1, r1, #1
8       bne top
9       b testpass
```

Figure 4.4: Multiply micro-benchmark

of instructions and hence get an IPC of 0.33. When simulating the benchmark on Amber25 it used 120 649 cycles, which indeed gave an IPC of 0.33. This shows that the RAS performs as expected and that it can give a performance improvement on function calls and returns.

4.3.2 Multiply

The multiply benchmark exploits the enhanced multiplier in Turbo Amber. The enhanced Turbo Amber multiplier decreases the number of cycles used on a multiply or multiply-accumulate instruction in Amber25, from 33 or 34 cycles respectively, down to 2 cycles. The micro-benchmark is shown in Figure 4.4. It consists of 200 multiply-accumulate instructions executed 100 times. This yields a total of 20 000 instructions executed when running this benchmark.

Using 2 cycles on each of the 20 000 multiply-accumulate instruction should take 40 000 cycles and hence we should measure an IPC of 0.5. The simulator spent 40 504 cycles executing the multiply microbenchmark and achieves a throughput of 0.49 IPC.

Amber25's multiply unit should use 34 cycles executing every multiply-accumulate instruction, which will yield an IPC of 0.03. The simulator used 720 596 cycles which indeed results in 0.03 IPC. This showed that the multiply unit performs as expected and that Turbo Amber multiplies significantly faster than Amber.

4.3.3 Branch Prediction

The improved branch predictor in Turbo Amber makes it possible to execute branches taken without any flushing or stalling. A micro-benchmark where the instructions are mostly branches would reveal if the branch predictor correctly predicted the branches.

```
1     mov r1, #1000      // 1000 iterations
2     b bottom
3     b check
4     add pc, pc, #-12 // 1
5     add pc, pc, #-12 // 2
6     ...
7     add pc, pc, #-12 // 199
8  bottom:
9     add pc, pc, #-12 // 200
10 check:
11    subs r1, r1, #1
12    bne bottom
13    b testpass
```

Figure 4.5: Branch prediction micro-benchmark

In a program without any branches the PC will advance as execution progresses. The branch prediction micro-benchmark does the exact opposite: it starts at the bottom and branches backwards 200 times. This whole process is then repeated 1000 times. Executing 200 instructions 1000 times gives a total of 200 000 executed instructions.

If we had implemented this correctly Turbo Amber's branch predictor would predict a newly discovered branch as not taken, but would correct itself on the next occurrence of the branch. Thereafter, it would correctly predict all the branches. Since predictions are done in the fetch stage it will immediately start execution from the predicted address, avoiding stalls caused by branching and Turbo Amber should achieve an IPC of 1. When simulated we measured that it did indeed spend 204609 cycles and reached an IPC of 0.98.

The static branch predictor in Amber25 always predicts branch not taken, so we believed it would mispredict the 200 000 branches and incur a 3 cycle branch mispredict penalty for each of them. We expected this would result in an IPC of 0.33. When simulated we measured that the simulation used 607252 cycles and yielded an IPC of 0.33. This shows that our branch predictor correctly predicted branches as designed and that when executing branches taken it was possible to achieve a performance improvement over Amber.

4.3.4 Instruction Queue

The instruction queue micro-benchmark demonstrates the prefetching effect caused by the instruction queue which reduces the performance penalty caused by compulsory misses. It simply consists of 1 000 consecutive `nop` instructions which will cause

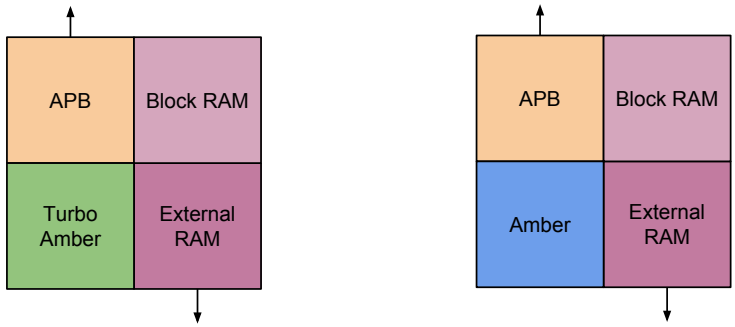

Figure 4.6: The layouts used when benchmarking Turbo Amber and Amber. Note that the APB tile and the External RAM tile are the only tiles that communicate with external peripherals.

compulsory misses.

Turbo Amber fetches instructions as long as the instruction queue has room for more. So it might seem like Turbo Amber should be able to achieve an IPC of 1 since it could prefetch the next set of instructions while the rest of the pipeline executes the instructions in the instruction queue. Unfortunately, the instruction cache has a turn-around time of two cycles when written to, and the fetch stage does not initiate a new memory transaction until the cache is ready. Spending two cycles waiting for the instruction cache and three cycles to fetch data from the memory means that it takes five cycles to fetch instructions from memory. If every instruction could be executed in one cycle, and Turbo Amber is able to fetch four instructions at a time, it would stall one cycle every five cycles. Hence, Turbo Amber achieves an IPC of 0.8. The simulator uses 1273 cycles executing the 1000 instructions and we get an IPC of 0.79.

Amber25 will fetch the next cache line of instructions when the last instruction in the current cache line is decoded. It must then stall two more cycles waiting for the memory transaction to complete. So Amber25 executes instructions 4 out of 6 cycles. This gives an IPC of 0.66. In simulation, the Amber25 core used 1543 cycles and achieved an IPC of 0.65.

4.4 Benchmarks

In this section we evaluated the performance of Turbo Amber relative to Amber by running benchmarks on SHMAC synthesized to FPGA.

We ran several benchmarks on Amber and Turbo Amber with the tile layout as seen in Figure 4.6. In addition to Turbo Amber the layout contained an on-chip RAM

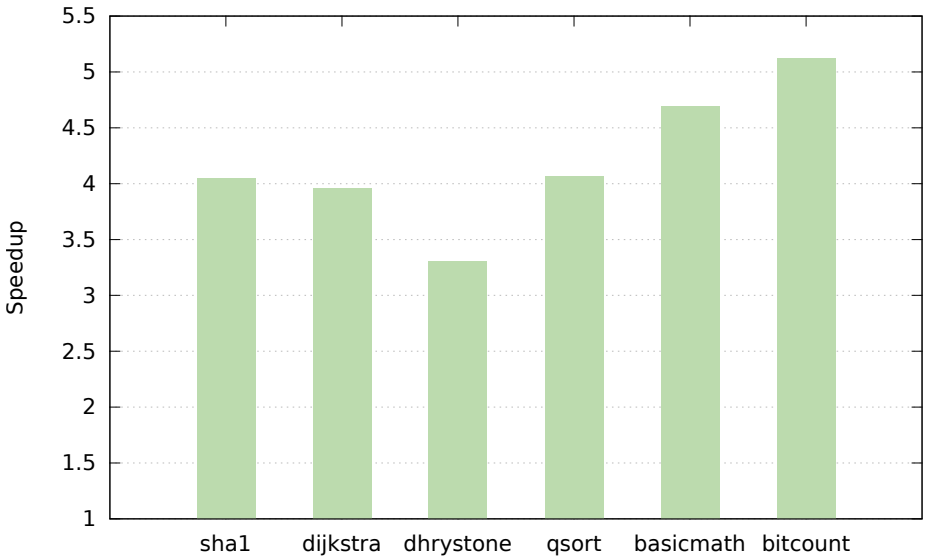

Figure 4.7: Speedups measured of Turbo Amber over Rav with caches disabled.

tile, an APB tile for I/O, and a tile for communicating with off-chip RAM. It should be noted that although we instantiated an on-chip RAM tile we did not store any data or instructions in its memory mapped area, so it was not actually in use. When benchmarking Amber we used the same layout, but replaced the Turbo Amber tile with a regular Amber tile.

Two challenges arose when attempting to benchmark Turbo Amber. Firstly, there is little to no benchmarking software that has been ported to SHMAC and that builds without significant effort. Secondly, even if you were able to port a benchmark to SHMAC you would probably have to run it with caches disabled. This is because SHMAC becomes unstable for most software when caches are enabled. The exact reason is unknown, but it is referred to as the "cache bug" and is a known bug reported by all users of SHMAC.

4.4.1 Uncached Benchmarks

Running benchmarks with caches disabled may not paint a realistic picture of Turbo Amber's micro-architectural improvement, but running software with caches disabled is the de facto standard for SHMAC. So showing the speedup results with disabled caches shows what programmers of SHMAC can get from Turbo Amber until the cache bug is fixed. The uncached speedup results have been gathered by the Linux

Benchmark		Amber	Turbo Amber	Speedup
Dhrystone	[Cycles per run]	752	507	1.48
SHA1	[microseconds per iteration]	269	179	1.50

Table 4.3: The performance measured when running benchmarks with caches enabled.

team of SHMAC [AA14]. They compared Turbo Amber to their processor Rav. Rav is another fork of Amber, it is an ARMv4 processor that has the same 2-cycle multiply unit as Turbo Amber. The speedup then comes only from the instruction queue and the branch predictor. As can be seen in Figure 4.7, Turbo Amber has a much larger speedup with caches disabled. This is due to the fact that the instruction queue causes a caching effect to occur.

4.4.2 Cached Benchmarks

We have been able to run two benchmarks with caches enabled, SHA1 and Dhrystone. SHA1 is a cryptographic hash function, whereas Dhrystone is a synthetic integer benchmark from the 80's. SHA1 could run on synthesized SHMAC, but Dhrystone suffered from the cache bug. Luckily the Amber simulation framework had ported a Dhrystone benchmark, enabling us to run Dhrystone in simulation, which does not suffer from the cache bug as it does not simulate a full SHMAC system. Interestingly their speedups over Amber are very similar, 48 % speedup on Dhrystone and 50 % speedup on SHA1 as can be seen from Table 4.3. From these benchmarks our best guess of the speedup of Turbo Amber over Amber is around 49 %.

Chapter 5
Conclusions And Further Work

In Chapter 1 we presented requirements that TA must satisfy. In this chapter we use the results obtained in the previous chapters to argue that the requirements were indeed satisfied. Also, we use our experience with Turbo Amber and the SHMAC infrastructure to give suggestions for how the project can be further developed.

5.1 Requirements

In this section we show that TA is fast, area-efficient, resource balanced, SHMAC-integrated and most importantly correct. Thus satisfying all requirements set in the introduction chapter. We begin with a short summary and then give a more in-depth evaluation.

Performance
TA's performance improvement was measured to be 49 % over Amber.

Area efficient
TA's logical slice increase is 70 % while TA's performance increase is 49 %.

Resource balanced
TA's most used resource are the logical slices and does not excessively drain any specialized FPGA-resources.

Integrated
TA has been instantiated into the SHMAC project as it's own tile and proved to be able to run programs as a part of SHMAC.

Verified
TA has been able to boot Linux and freeRTOS and run applications on top of them.

5.1.1 Fast

TA is required to be significantly faster than Amber. The faster TA is, compared to Amber, the more heterogeneous SHMAC will be. This requirement is therefore important for the main research goal of SHMAC.

To satisfy this requirement we have made significant changes to Amber by introducing instruction queueing between fetch and decode, a 4-way superscalar fetch stage, branch prediction, and improved multiply performance. Each technique has been verified to improve the performance as intended with micro-benchmarks and a 49 % overall performance increase has been measured with the SHA and Dhrystone benchmarks.

We conclude that the performance requirement is satisfied given that we have implemented improvements, shown that they behave as intended, and measured the average performance improvement to be 49 %.

5.1.2 Area Efficient

In the introduction we argued that the performance increase must be seen in context to the area increase. A reasonable test is if the performance increase is greater than what Pollack's rule would dictate. As can be seen from the equations below, we are well within the limits of a reasonable area increase. Note that Pollack's rule applies to area, and that we use number of occupied slices as a metric for area. But this metric does not take into account the increase in block RAM.

$$\begin{aligned}
\frac{TA_performance}{A_performance} &\geq \sqrt{\frac{TA_area}{A_area}} \\
1.49 &\geq \sqrt{\frac{4092}{2412}} \\
1.49 &\geq \sqrt{1.70} \\
1.49 &\geq 1.30
\end{aligned}$$

5.1.3 Resource Balanced

TA should not excessively drain any of the FPGA-resources. This is important because if there is an imbalance in the usage of FPGA-resources the FPGA will become underutilized. In section 4.1 Turbo Amber was shown to use no DSP slices, 3.3 % of the block RAM, and 7.9 % of the generic slices. This shows that if one scales up the number of Turbo Amber cores it is the primary computing fabric that

will limit us first. Neither DSP slices or block RAM are excessively drained when instantiating a Turbo Amber core.

5.1.4 Integrated

TA should be integrated into SHMAC as a separate tile. It is important for SHMAC contributors to make their work reusable and integrated into the existing SHMAC infrastructure. To this end we created a tile that can be instantiated along side the original Amber tile. Creating a new tile required changes to the source code that reads layouts and does HDL code generation. In conclusion, Turbo Amber is integrated into SHMAC's infrastructure and is ready to be reused by other projects.

5.1.5 Verified

TA should be extensively verified to be functionally correct. A processor can be revolutionary fast, but if it can't run programs correctly it would be useless. A significant engineering effort has been done to verify TA's functional correctness. The verification has been done with very similar techniques to what true ARM verification engineer's use. And the correctness is now evidenced by 79 self-checking assembly sequences and running applications on the Linux and freeRTOS operating systems. There is at the time of writing no known bugs in Turbo Amber.

5.2 Further Work

In this section we make suggestions for improving both TA and the SHMAC infrastructure.

5.2.1 Improve The SHMAC Simulator Framework Documentation

During the SHMAC integration phase we encountered bugs in Turbo Amber that would have been nearly impossible to fix without a simulator. We believe this will be so for future students working on SHMAC as well. A simulator could also be used as a tool for verifying the SHMAC platform more extensively. There exists a Modelsim script that allows one to simulate the entirety of SHMAC, but it is undocumented. Few know that it exists at all. The lack of documentation led to a great amount of time being spent on setting up the correct environment for the simulator. We believe the SHMAC community would benefit from documenting the simulator. For example, it should be easy to acquire the necessary libraries and tools, setting the correct environment variables etc.

5.2.2 Improving TA's Integer Performance

Turbo Amber has proven to be significantly faster than the original Amber. Still, there is potential for increasing Turbo Amber's performance. Although the fetch stage is superscalar, the back-end is still scalar. Creating a superscalar back-end would justify the performance of the fetch stage and definitely increase performance.

5.2.3 Improving TA's Floating-Point Performance

Turbo Amber's floating-point performance is poor due to the lack of a hardware Floating-Point Unit (FPU). An effort has already been made in integrating an FPU into the original Amber core [Knu14]. We believe the FPU could be reused in Turbo Amber with ease, and would increase the floating-point performance of Turbo Amber significantly.

References

[a8o] ARM Cortex-A8 processor. http://arm.com/products/processors/cortex-a/cortex-a8.php.

[AA14] Håkon Ø. Amundsen and Joakim E.C. Andersson. Linux for SHMAC. Master's thesis, Norwegian University of Science and Technology, 2014.

[alp99] *Alpha 21264 Microprocessor Hardware Reference Manual*, 1999.

[app] http://support.apple.com/kb/sp598.

[BC11] Shekhar Borkar and Andrew A. Chien. The Future of Microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[CGJ+97] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Evidence-based Static Branch Prediction Using Machine Learning. *ACM Trans. Program. Lang. Syst.*, 19(1):188–222, January 1997.

[cor11] *Cortex-A15 Technical Reference Manual*, 2011.

[DGnY+74] Robert H. Dennard, Fritz H. Gaensslen, Hwa nien Yu, V. Leo Rideout, Ernest Bassous, Andre, and R. Leblanc. Design of ion-implanted MOSFETs with very small physical dimensions. *IEEE J. Solid-State Circuits*, page 256, 1974.

[EBSA+11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. *SIGARCH Comput. Archit. News*, 39(3):365–376, June 2011.

[Fag97] Barry Fagin. Partial resolution in branch target buffers. *Computers, IEEE Transactions on*, 46(10):1142–1145, 1997.

[GHSV+11] Nathan Goulding-Hotta, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Po-Chao Huang, Manish Arora, Siddhartha Nath, Vikram Bhatt, Jonathan Babb, Steven Swanson, and Michael Bedford Taylor. The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future. *IEEE Micro*, 31(2):86–95, 2011.

[HM08] Mark D. Hill and Michael R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, July 2008.

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[ipa] http://www.gsmarena.com/apple_ipad_wi_fi-3828.php.

[iph] http://www.gsmarena.com/apple_iphone_4_cdma-3716.php.

[JR07] Eugene John and Juan Rubio. *Unique Chips and Systems*. CRC Press, 2007.

[KE91] David R. Kaeli and Philip G. Emma. Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns. *SIGARCH Comput. Archit. News*, 19(3):34–42, April 1991.

[Knu14] Jakob Dagsland Knutsen. Extending Amber with a Hardware FPU. 6 2014.

[KTR+04] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. *SIGARCH Comput. Archit. News*, 32(2):64– , March 2004.

[Moo06] Gordon E. Moore. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *Solid-State Circuits Society Newsletter, IEEE*, 11(5):33–35, 2006.

[Nai92] R. Nair. Branch behavior on the IBM RS/6000. In *Technical report*. IBM Computer Science, 1992.

[Pol99] Fred J. Pollack. New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies (Keynote Address)(Abstract Only). In *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pages 2–, Washington, DC, USA, 1999. IEEE Computer Society.

[San] Amber ARM-compatible core, OpenCores. http://opencores.org/project,amber.

[SD01] N Sharma and B Dickman. Verifying the ARM Core. *Integrated System Design*, 13:44–51, 2001.

[shm] http://www.ntnu.edu/ime/eecs/shmac.

[SL13] John Paul Shen and Mikko H Lipasti. *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[Smi81] James E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pages 135–148, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[Sta13] William Stallings. *Computer organization and architecture : designing for performance*. Pearson, Boston, 2013.

[Sut05] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), March 2005.

[USS97] A.K. Uht, V. Sindagi, and S. Somanathan. Branch effect reduction techniques. *Computer*, 30(5):71–81, May 1997.

[Wal14] Magnus Walstad. Task based programming on SHMAC. 5 2014.

[WBR11] Henry Wong, Vaughn Betz, and Jonathan Rose. Comparing FPGA vs. Custom Cmos and the Impact on Processor Microarchitecture. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 5–14, New York, NY, USA, 2011. ACM.

[Wik13] Håkon O. Wikene. Benchmarking SHMAC. 12 2013.

Appendix A
Synthesis Report

A.1 Turbo Amber Xilinx Mapping Report File

```
Release 14.7 Map P.20131013 (lin64)
Xilinx Mapping Report File for Design 'turbo_amber_wrapper'

Design Information
------------------
Command Line   : map -filter
/home/boe/SHMAC/XilinxProjects/shmacAmber/shmacAmber/iseconfig/filter.filter
-intstyle ise -p xc5vlx330-ff1760-1 -w -logic_opt on -ol std -t 1 -global_opt
speed -equivalent_register_removal on -mt 2 -cm speed -detail -ir off
-ignore_keep_hierarchy -pr b -c 1 -lc off -power off -o
turbo_amber_wrapper_map.ncd turbo_amber_wrapper.ngd turbo_amber_wrapper.pcf
Target Device  : xc5vlx330
Target Package : ff1760
Target Speed   : -1
Mapper Version : virtex5 -- $Revision: 1.55 $
Mapped Date    : Mon Jun  9 17:37:25 2014

Design Summary
--------------
Number of errors:      0
Number of warnings:    0
Slice Logic Utilization:
  Number of Slice Registers:                 4,092 out of 207,360    1%
    Number used as Flip Flops:               4,086
    Number used as Latch-thrus:                  6
  Number of Slice LUTs:                     11,849 out of 207,360    5%
    Number used as logic:                    9,765 out of 207,360    4%
      Number using O6 output only:           7,633
      Number using O5 output only:             209
```

```
      Number using O5 and O6:                1,923
    Number used as Memory:                   2,072 out of  54,720    3%
      Number used as Dual Port RAM:            252
        Number using O5 and O6:                252
      Number used as Single Port RAM:        1,820
        Number using O6 output only:         1,820
    Number used as exclusive route-thru:        12
  Number of route-thrus:                       243
    Number using O6 output only:               219
    Number using O5 output only:                22
    Number using O5 and O6:                      2

Slice Logic Distribution:
  Number of occupied Slices:                 3,248 out of  51,840    6%
  Number of LUT Flip Flop pairs used:       12,329
    Number with an unused Flip Flop:         8,237 out of  12,329   66%
    Number with an unused LUT:                 480 out of  12,329    3%
    Number of fully used LUT-FF pairs:       3,612 out of  12,329   29%
    Number of unique control sets:             123
    Number of slice register sites lost
      to control set restrictions:             146 out of 207,360    1%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  OVERMAPPING of BRAM resources should be ignored if the design is
  over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:
  Number of bonded IOBs:                       317 out of   1,200   26%
    IOB Flip Flops:                             82

Specific Feature Utilization:
  Number of BlockRAM/FIFO:                      10 out of     288    3%
    Number using BlockRAM only:                 10
    Total primitives used:
      Number of 36k BlockRAM used:               9
      Number of 18k BlockRAM used:               1
    Total Memory used (KB):                    342 out of  10,368    3%
  Number of BUFG/BUFGCTRLs:                      1 out of      32    3%
    Number used as BUFGs:                        1

Average Fanout of Non-Clock Nets:                5.20
```

A.2 Amber Xilinx Mapping Report File

```
Release 14.7 Map P.20131013 (lin64)
Xilinx Mapping Report File for Design 'amber_wrapper'

Design Information
------------------
Command Line   : map -filter
/home/boe/SHMAC/XilinxProjects/shmacAmber/shmacAmber/iseconfig/filter.filter
-intstyle ise -p xc5vlx330-ff1760-1 -w -logic_opt on -ol std -t 1 -global_opt
speed -equivalent_register_removal on -mt 2 -cm speed -detail -ir off
-ignore_keep_hierarchy -pr b -c 1 -lc off -power off -o amber_wrapper_map.ncd
amber_wrapper.ngd amber_wrapper.pcf
Target Device  : xc5vlx330
Target Package : ff1760
Target Speed   : -1
Mapper Version : virtex5 -- $Revision: 1.55 $
Mapped Date    : Mon May 12 10:14:20 2014

Design Summary
--------------
Number of errors:      0
Number of warnings:    0
Slice Logic Utilization:
  Number of Slice Registers:                3,551 out of 207,360    1%
    Number used as Flip Flops:              3,548
    Number used as Latch-thrus:                 3
  Number of Slice LUTs:                     8,879 out of 207,360    4%
    Number used as logic:                   7,100 out of 207,360    3%
      Number using O6 output only:          5,697
      Number using O5 output only:             67
      Number using O5 and O6:               1,336
    Number used as Memory:                  1,768 out of  54,720    3%
      Number used as Dual Port RAM:           232
        Number using O5 and O6:               232
      Number used as Single Port RAM:       1,536
        Number using O6 output only:        1,536
    Number used as exclusive route-thru:       11
  Number of route-thrus:                       82
    Number using O6 output only:               70
    Number using O5 output only:                4
    Number using O5 and O6:                     8

Slice Logic Distribution:
```

```
  Number of occupied Slices:                2,412 out of  51,840    4%
  Number of LUT Flip Flop pairs used:       9,147
    Number with an unused Flip Flop:        5,596 out of   9,147   61%
    Number with an unused LUT:                268 out of   9,147    2%
    Number of fully used LUT-FF pairs:      3,283 out of   9,147   35%
    Number of unique control sets:            149
    Number of slice register sites lost
      to control set restrictions:            220 out of 207,360    1%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  OVERMAPPING of BRAM resources should be ignored if the design is
  over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:
  Number of bonded IOBs:                    317 out of   1,200   26%
    IOB Flip Flops:                          82

Specific Feature Utilization:
  Number of BlockRAM/FIFO:                    3 out of     288    1%
    Number using BlockRAM only:               3
    Total primitives used:
      Number of 36k BlockRAM used:            3
    Total Memory used (KB):                 108 out of  10,368    1%
  Number of BUFG/BUFGCTRLs:                   2 out of      32    6%
    Number used as BUFGs:                     2

Average Fanout of Non-Clock Nets:             5.42
```

Appendix B

Self-Checking Assembly Tests

Test name	Test description
adc	Tests the adc instruction. Adds 3 32-bit numbers using adc and checks the result.
add	Tests the add instruction. Runs through a set of additions of positive and negative numbers, checking that the results are correct. Also tests that the 's' flag on the instruction correctly sets the condition flags.
barrel shift rs	Tests the barrel shift operation with a mov instruction, when the shift amount is a register value. Test that shift of 0 leaves Rm unchanged. Tests that a shift of > 32 sets Rm and carry out to 0.
barrel shift	Tests the barrel shift operation with a mov instruction when the shift amount is an immediate value.Tests lsl, lsr and ror.
bcc	Tests branch on carry clear.
bic bug	Test added to catch specific bug with the bic instruction.
bl	Test Branch and Link instruction. Checks that the correct return address is stored in the link register (r14).
bx	Test that it is possible to perform a function call return using the bx instruction.
cache1	Contains a long but simple code sequence. The entire sequence can fit in the cache. This sequence is executes 4 times, so three times it will execute from the cache. Test passes if sequence executes correctly.

Continued on next page

Table B.1 – continued from previous page

Test name	Test description
cache2	Tests simple interaction between cached data and uncached instruction accesses.
cache3	Tests that the cache can write to and read back multiple times from 2k words in sequence in memory - the size of the cache.
cacheable area	Tests the cacheable area co-processor function.
cache flush	Tests the cache flush function. Does a flush in the middle of a sequence of data reads. Checks that all the data reads are correct.
cache swap bug	Tests the interaction between a swap instruction and the cache. Runs through a main loop multiple times with different numbers of nop instructions before the swp instruction to test a range of timing interactions between the cache state machine and the swap instruction.
cache swap	Fills up the cache and then does a swap access to data in the cache. That data should be invalidated. Check by reading it again.
change mode	Tests teq, tst, cmp and cmn with the p flag set. Starts in supervisor mode, changes to Interrupt mode then Fast Interrupt mode, then supervisor mode again and finally User mode.
ddr31	Word accesses to random addresses in DDR3 memory.
ddr32	Tests byte read and write accesses to DDR3 memory.
ddr33	Test back to back write-read accesses to DDR3 memory.
ethmac mem	Tests wishbone access to the internal memory in the Ethernet MAC module.
ethmac tx	Tests ethernet MAC frame transmit and receive functions and Ethmac DMA access to hiboot mem. Ethmac is put in loopback mode and a packet is transmitted and received.
firq	Executes 20 FIRQs at random times while executing a small loop of code.
flow bug	The core was illegally skipping an instruction after a sequence of 3 conditional not-execute instructions and 1 conditional execute instruction.

Continued on next page

Table B.1 – continued from previous page

Test name	Test description
flow1	Tests instruction and data flow. Specifically tests that a stm writing to cached memory also writes all data through to main memory.
flow2	Tests that a stream of str instructions writing to cached memory works correctly.
flow3	Tests ldm where the pc is loaded which causes a jump. At the same time the mode is changed. This is repeated with the cache enabled.
hiboot mem	Tests wishbone read and write access to hi (non-cachable) boot SRAM. inflate bug A load store sequence was found to not execute correctly.
irq	Tests running a simple algorithm to add a bunch of numbers and check that the result is correct. This algorithm runs 80 times. During this, a whole bunch of IRQ interrupts are triggered using the random timer.
ldrh	Tests that the ldrh instruction correctly loads a 16 bit value to the correct memory location. It is possible to perform a single call to str and then fetch the entire word resulting from two ldrh calls.
ldm stm onetwo	Tests ldm and stm of single registers with cache enabled. Tests ldm and stm of 2 registers with cache enabled.
ldm1	Tests the standard form of ldm.
ldm2	Tests ldm where the user mode registers are loaded whilst in a privileged mode.
ldm3	Tests ldm where the status bits are also loaded.
ldm4	Tests the usage of ldm in User Mode where the status bits are loaded. The s bit should be ignored in User Mode.
ldr	Tests ldr and ldrb with all the different addressing modes.
ldr atr pc	Tests lrd and str of r15.
ldrsh	When loading a signed half word from memory using ldrsh, the value is correctly sign extended and stored in the target register.
mla	Tests the mla (multiply and accumulate) instruction.

Continued on next page

Table B.1 – continued from previous page

Test name	Test description
movs bug	Tests a movs followed by a sequence of ldr and str instructions with different condition fields.
mul	Tests the mul (multiply) instruction.
sbc	Tests the 'subtract with carry' instruction by doing 3 64-bit subtractions.
stm stream	Generates as dense a stream of writes as possible to check that the memory subsystem can cope with this worst case.
stm1	Tests the normal operation of the stm instruction.
stm2	Test jumps into user mode, loads some values into registers r8 - r14, then jumps to FIRQ and saves the user mode registers to memory.
strb	Tests str and strb with different indexing modes.
sub	Tests sub and subs.
swi	Tests the software interrupt – swi.
swp lock bug	Bug broke an instruction read immediately after a swp instruction.
swp	Tests swp and swpb.
uart reg	Tests wishbone read and write access to the Amber UART registers.
uart rxint	Tests the UART receive interrupt function. Some text is sent from the test uart to the uart and an interrupt generated.
uart rx	Tests the UART receive function.
uart tx	Uses the tb uart in loopback mode to verify the transmitted data.
undefined ins	Tests Undefined Instruction Interrupt. Fires a few unsupported floating point unit (FPU) instructions into the core. These cause undefined instruction interrupts when executed.
ldrsb	When loading a signed byte from memory using `ldrsh`, the value is correctly sign extended and stored in the target register.

Continued on next page

Table B.1 – continued from previous page

Test name	Test description
mla long	The `umlal` instruction correctly multiplies the two source registers, adds the sum of the two target registers and saves the result correctly. When the result of the multiplication is zero, but the value accumulated from the target registers is not zero, the zero flag is not set. When the result of the multiplication is zero, and the value accumulated from the target registers is zero, the zero flag is set.
mul long	The `umull` instruction correctly multiplies the two source registers and saves the result correctly in the two source registers. When the result is zero, the ZERO flag is set in CPSR.
mla long hazard	For each of the registers used as argument to the `umlal` instruction, if it is loaded from memory in the instruction preceding the `umlal` instruction, the processor will be stalled, and the correct value will be used.
mul long hazard	The same as for *mla long hazard* only with the `umull` instruction.
mrs	The `mrs` instruction correctly copies the value of the CPSR to the target register.
msr flags only i	`msr` with the `flag only` bit set only updates the flag bits of the CPSR register, and that it is possible to use an immediate value as operand.
msr flags only r	The same as the above-mentioned test, but uses a register as operand instead of an immediate value.
msr immediate	`msr` with an immediate value as argument correctly updates the entire CPSR register.
shift cpsr carry	The barrel shift unit is able to use the carry bit from the CPSR register.
smla long	Doing signed multiplication with `smlal` works correctly. If the result is negative, the NEGATIVE bit of the CPSR is set, otherwise it is not set. The value from the target registers is correctly added to the multiplication result, for both positive and negative values.

Continued on next page

Table B.1 – continued from previous page

Test name	Test description
smul long	Doing signed multiplication with `smull` works correctly. If the result is negative, the NEGATIVE bit of the CPSR is set, otherwise it is not set.
spsr	It is possible to write to and read from the SPSR register using the `msr` instruction.
spsr change mode	The CPSR register is correctly backed up in the SPSR register of the correct execution mode during context switches. When returning to user mode, the CPSR value is correctly restored.
spsr immediate	`msr` with an immediate value as argument correctly updates the entire SPSR register.
strh	Storing a half word using `strh` works correctly. It is possible to load a value written by two `strh` calls by using `ldr`.
system mode	It is possible to change the execution mode from system mode. The same registers are available from system mode and user mode.
halfword load-use conflict bug	Tests that a load-use hazard with halfword loads is handled correctly.
irq lr bug	Tests that the link register is handled correctly when an interrupt occurs.
load-use hazard mode check bug	When Amber25 does forwarding to resolve load-use hazards it forgets to check if the processor mode of the load-instruction and the use-instuction matches. This causes a false positive in forwarding and the wrong value to be used. This has been detected by this test and rectified in Turbo Amber.
mla long hazard2	This test demonstrates a load-use forwarding bug in UMLAL. The load-use hazard detector in execute was broken.
mul simple	A less comprehensive test of multiply than mul.