NTNU – Trondheim
Norwegian University of
Science and Technology

Implementing a Heterogeneous Multi-Core Prototype in an FPGA

Leif Tore Rusten
Gunnar Inge Sortland

Master of Science in Computer Science

Submission date:	July 2012
Supervisor:	Magnus Jahre, IDI
Co-supervisor:	Gunnar Tufte, IDI

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

Problem Description

Implementing a Heterogeneous Multi-Core Prototype in an FPGA

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. In this thesis, the students should develop a framework for realizing heterogeneous multi-core processors in an FPGA. The minimum requirements of this thesis are two simple processor core implementations, a scalable interconnect and a memory interface. If time permits, a simple Task Based Parallelism (TBP) runtime engine can be implemented for the prototype architecture.

Supervisor: Magnus Jahre, IDI

Sammendrag
(Norwegian Abstract)

Siden midten av 1980-tallet har ytelsesveksten for prosessorer vært formidabel, med en årlig vekst på om lag 52%. Denne veksten er gjort mulig blant annet grunnet arkitekturer som utnytter ILP og økende klokkefrekvenser. Disse fremgangsmåtene for ytelseøkning er derimot ikke lenger like effektive siden vi har nådd den praktiske grensen for effektforbruket til prosessorer. Ved å bruke heterogene prosessorer kan vi forbedre energieffektiviten, men dette krever forskning på åpne spørsmål vedrørende heterogen prosessering. Mangelen på heterogen maskinvare gjør det derimot vanskelig å drive effektiv forskning. I denne oppgaven har vi laget et svært modulært og konfigurerbart rammeverk for prototyperealiseringer av heterogene flerkjerneprosessorer. Rammeverket, som er planlagt brukt hos datamaskingruppen ved NTNU, består av to forskjellige prosessorfliser i tillegg til en rekke funksjonelle fliser. Disse flisene kan flislegges til forskjellige heterogene prosessorprototyper. Ved å bruke rammeverket vårt opp mot en Xilinx Virtex 6 fikk vi implementert prosessorer med opp til 40 heltallskjerner eller 16 flyttallskjerner.

Abstract

Since the mid-1980s processor performance growth has been remarkable, with an annual growth of about 52 %. Methods such as architectural enhancements exploiting ILP and frequency scaling have been effective at increasing performance, but are now limited by its diminishing returns and the power wall. Heterogeneous processors as an alternative source for continued growth looks promising, but research on heterogeneous software is made difficult as heterogeneous hardware is in low supply. This thesis cover the design and implementation of a heterogeneous processor called SHMAC and its framework. Flexibility of the delivered system allows rapid exploration of both hardware and software sides of heterogeneous processor research questions. The system is intended for research at CARD at NTNU. Two processor tiles and a set of additional tiles for extended functionality are provided, yielding a wide range of possible hardware setups in the delivered framework. Using a Xilinx Virtex 6 we were able to implement 40 integer cores or 16 floating-point cores.

Preface

This thesis is submitted to the Norwegian University of Science and Technology for partial fulfillment of the requirements for a master's degree.

This work has been performed at the Department of Computer and Information Science, NTNU, Trondheim, with Magnus Jahre as supervisor and Gunnar Tufte as co-supervisor.

Acknowledgment

We would like to thank our supervisors for good discussions and useful feedback. Without their continued support, eagerness and guidance this thesis would not have been possible. The weekly follow-up meetings have been a source for many great discussions guiding the progression of this thesis in a worthwhile direction.

We would also like to thank Nordic Semiconductor for providing us with a modern FPGA development board capable of large designs and with an easy host interface. We would not have done as well without it.

Gunnar Inge would also like to thank his wife Synne Agnete for all her extensive support.

The "Heterogeneous Multicorn" logo is built upon the work of Dee Dreslough with post-processing by Karl Johan Heimark.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

ABI Application Binary Interface. 48

ASIC Application-Specific Integrated Circuit. 72

CARD The Computer Architecture and Design Group. 2, 4, 69, 70

CMP Chip Multiprocessor. 16

CPI Cycles Per Instruction. 59

CPOP Critical-Path-on-a-Processor. 17

CPU Central Processing Unit. 14, 34

CUDA Compute Unified Device Architecture. 16

DAG Directed Acyclic Graph. 17, 18, 105

DLS Dynamic Level Scheduling. 17

FCP Fastest Critical Path. 17

FPGA Field-Programmable Gate Array. i, vii, 3–5, 43, 46, 47, 55, 56, 61, 62, 65, 69, 72, 73, 95, 105

FPU Floating-Point Unit. 24, 61, 62

GCC GNU Compiler Collection. 5–7, 24, 34, 35

GDB GNU Debugger. 53, 75

GPU Graphical Processing Unit. 15, 16

HEFT Heterogeneous Earliest-Finish-Time. 16, 17

HMP Heterogenous Multicore Processor. 8

ILP Instruction Level Parallelism. 1, 2, 5, 13, 14, 17

ISA Instruction Set Architecture. 4, 6, 16, 24, 33, 34, 51, 85

LED Light-Emitting Diode. 25, 29, 38, 39, 75

LL Load-Linked. 30, 31, 34, 35, 41, 47, 49–51, 55, 56, 65, 70, 78, 85, 101

LMT Levelized-Min Time. 17

LUT Lookup Table. 62

NoC Network-on-Chip. 43

NTNU Norwegian University of Science and Technology. 2

NUMA Non-Uniform Memory Access. 23, 26, 33

PC Program Counter. 40

RAM Random Access Memory. 25, 29, 31, 32, 38, 40, 41, 61, 65, 67, 70, 71, 73, 105

RISC Reduced Instruction Set Computer. 24

RMW Read-Modify-Write. 41, 51, 85

SC Store-Conditional. 30, 31, 34, 35, 41, 47, 49–51, 55, 56, 65, 70, 78, 85, 101

SHMAC Single-ISA Heterogeneous Multicore Architecture Computer. 3, 4, 6, 8, 23, 24, 26, 29, 31–33, 39–43, 45, 46, 48, 51, 52, 55, 56, 60, 65, 66, 69, 70, 72–76, 101, 105

SIMD Single Instruction, Multiple Data. 16

SoC System-on-Chip. 18

TBP Task Based Parallelism. xi, 4, 17, 48, 53, 66, 76, 77

TLP Task Level Parallelism. 14, 15

WLIW Very Long Instruction Word. 14

1 Introduction

It is roughly 60 years since the first general-purpose electronic computer was created. In this period, the progress of the field of processor design has been incredible. Figure 1.1 shows that from 1986 to 2002 processors had a performance growth of about 52 % annually. The performance growth for single core processors has however declined the latest years due to the limitations of power, available Instruction Level Parallelism (ILP) and memory latency [1]. According to Horowitz and Dally [2] the power of today's chip is now limited by the cost of cooling.

As the growth in single-core processors have been high, computer software has traditionally been written for serial computation. Until recently, performance growth have focused on increasing instruction throughput through pipelining for allowing frequency scaling and exploiting all forms of ILP [3]. To enable higher frequencies and thus performance, processors became increasingly complex with deep pipelines, advanced branch

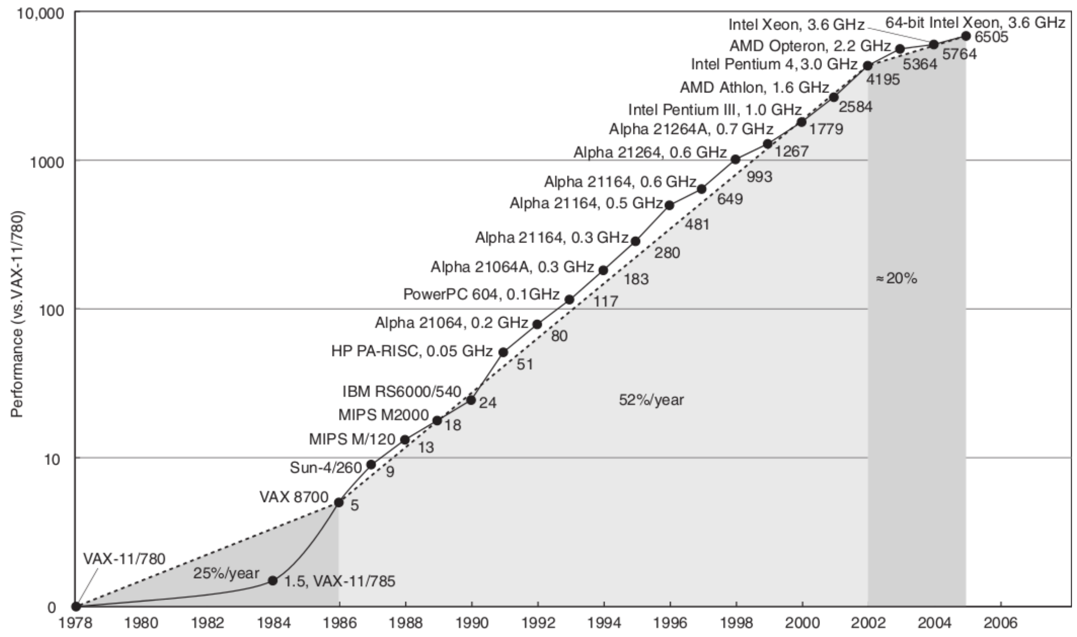


Figure 1.1: History of Processor Performance from 1978 to 2006. (Reproduced from [1, p. 3])

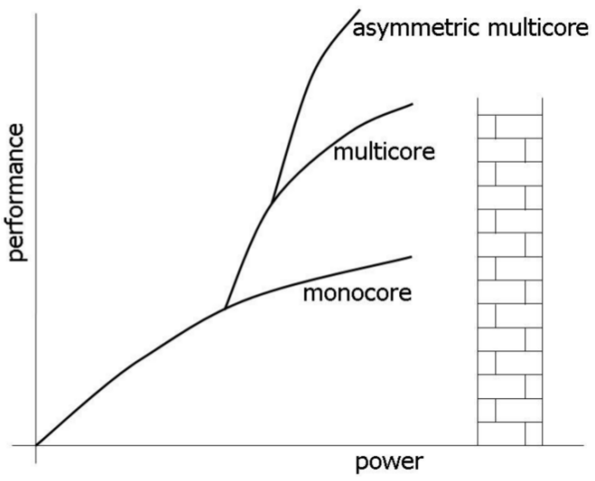


Figure 1.2: The Power Wall. (Reproduced from [4])

prediction to avoid the cost of flushing these pipelines and reordering of instructions to exploit ILP.

The increased complexity of processor designs have led to an increased power usage as frequency is increased and more transistors are utilized. Meenderinck and Juurlink [4] argues that we have reached the power wall for single-core processors (illustrated in Figure 1.2) i.e. we can no longer easily get more performance growth from adding complex hardware modules. This is due to power constraints which limit the complexity at a given clock frequency, but designs are also influenced by its economics [5]. Data centers consumed 1.5 % of all electricity in the U.S. in 2006 with an annual growth of 12 % [6]. This has created an increased focus on energy efficiency.

Borkar and Chien [7] predicts that operation frequency will increase slowly, with energy as the key limiter, during the next two decades. This forces a change in our approach to exploiting parallelism for increasing performance. One possibility is to focus on heterogeneous cores and accelerators to achieve good performance and energy efficiency. As given by Amdahl's law, total speedup for parallel execution of a program is limited by its sequential parts. Performance will therefore require research in exploiting parallelism without sacrificing single thread performance [8].

In nearly every program there will be a substantial amount of work which cannot easily be implicitly parallelized by hardware. This is due both to true dependencies within the program in addition to that there may be to expensive to exploit all available ILP within a program. In a multicore setting, the total execution time and/or power consumption would benefit from assigning unparallelizable work to a faster CPU core or accelerator. This is exactly what heterogeneous processors enables if one assigns unparallelizable or hard work to higher performance cores or accelerators, while still using simple cores for normal parallelizable work. Some early research suggests that a task-based programming model may be fitting to utilize a system using heterogeneous cores, given that the scheduler is aware of its architecture [9, 10].

Heterogeneous computing is becoming one of the main focuses of research at The Computer Architecture and Design Group (CARD) at Norwegian University of Science and Tech-

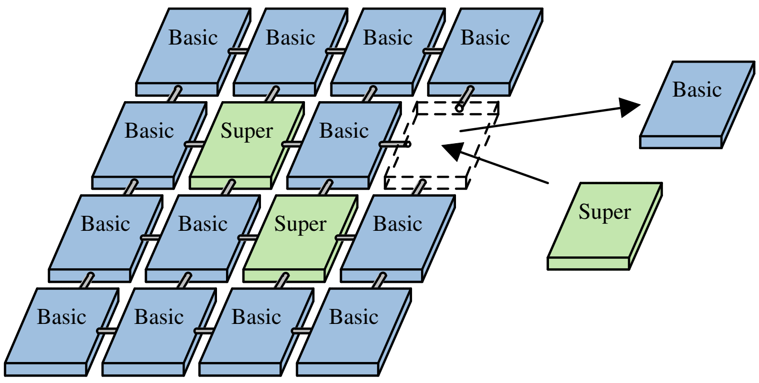


Figure 1.3: A first SHMAC overview

nology (NTNU). This computer architecture approach looks promising both for reducing energy consumption and increasing performance. To obtain as large a potential gain as possible it is necessary to understand how to utilize the heterogeneous computational units in smart ways. This requires research on the topic of heterogeneous processors, but heterogeneous processor implementations are not readily obtainable.

1.1 SHMAC Overview

This thesis' purpose is to implement a heterogeneous processor using an Field-Programmable Gate Ar ray (FPGA) in an attempt to circumvent the current lack of heterogeneous hardware available. A hardware framework is requested rather than a simulator framework because software approaches becomes limited due to simulation overhead as the core count increases. Simulators are on the other hand very flexible. Our thesis' outcome should allow for the specification of multiple different heterogeneous designs. This customization shall compensate for the lack of currently commercial solutions to build research on. The customization also tries to mitigate the loss of flexibility incurred by moving away from simulators. Based on the hardware space attainable from this customizable framework, research on characteristics software solutions need to fully exploit heterogeneous processing can be done. In case the hardware space made available in our delivery becomes to restrictive, the system is designed in a way that allows for easy modification and additions of new modules and concepts to the framework.

We have created a processor design called Single-ISA Heterogeneous Multicore Architecture Computer (SHMAC) which together with its associated framework promises to fulfill the previously outlined purpose. Figure 1.3 shows one possible hardware configuration of the SHMAC processor consisting of sixteen cores. The cores are encapsulated in a construct called a tile. The tile concept is known for its ease of scalability and how its reusability leverage initial design cost and subsequent maintenance efforts. Furthermore, as previous work by Zhang and Asanovic [11] shows, tile-based architectures allows for a modular approach. The details of the different interiors of tiles are kept contained within each tile. By encapsulating the different cores in uniform tiles we allow for the complete separation of the interconnection module from the core modules. This makes it easy to modify the organization of tiles and the implementations of the modules. New hardware concepts can easily be made into tiles and be integrated with the existing framework.

In addition to the processor framework we have created a small run-time support system creating the abstractions needed for communication between cores and for synchronization. If cores are to cooperate, then these building blocks are of importance since reasoning about shared memory is easily error-prone if not supported by some sort of software abstractions. The SHMAC framework is intended to be CARD's initial prototype of a heterogeneous framework. It is going to aid in the exploration of existing solutions (e.g. task-based parallelism) and discovery of novel software approaches. The delivered software abstractions is considered helpful for a swift start-up after project handover.

1.2 Assignment Interpretation

We have split the assignment into the following precise tasks:

T1 (Mandatory) Create a framework for heterogeneous multi-core processors in an FPGA. Different processor cores must be included in the delivered framework.

T2 (Mandatory) Include a scalable interconnect for processors and memory.

T3 (Mandatory) Add a memory interface for external access.

T4 (Optional) Implement a Task Based Parallelism (TBP) run-time engine on top of the hardware.

T3 has been made more precise than what the assignment text specifies as merely requiring any memory interface is trivially fulfilled in any processor design. Making precise the notion of external memory accesses as for e.g. examining memory from a lab setup, introduces functionality and possibilities not already inherent. This is assumed to be the true intent of this part of the assignment.

1.2.1 Requirements

A set of developer requirements were determined based on a combination of the tasks found, input from CARD and our own self-interests in such a framework.

Functional Requirements

1. The framework must ship with a basic integer based core
2. The framework must ship with a floating-point accelerated core
3. The framework shall utilize the modular tile architecture
4. The framework shall utilize a mesh interconnect
5. The framework must have an external memory interface
6. The framework need only support a single Instruction Set Architecture (ISA)

Table 1.1: Key design decisions

Design question	Choices considered (**bold** marks decision)	Remarks
Architectural class	Classic, **Tile**	By Classic we mean the de facto industry standard with cores connected in a multi-level cache hierarchy.
Interconnect type	**Mesh**, Tree, All-to-all, Bus	All-to-all and bus interconnects were only briefly considered as these were considered to not meet the scalability requirement of T2.
Core diversity by	**Accelerators**, ILP	Starting from a simple core it seemed less time-consuming to integrate an accelerator then to add ILP techniques.
Multi-ISA	Yes, **No**	Single-ISA suffice when researching the impact of software techniques on different heterogeneous processor layouts, while simplifying aspects of both processor tile design and run-time system.

Non-Functional Requirements

1. A FPGA must be utilized
2. The GNU Compiler Collection (GCC) tool-chain should be supported

These requirements fixes the more loosely defined tasks. As the tasks left us with a great deal of freedom of choice, some overall factors were decided upon to guide in the decision making. Table 1.1 succinctly summarize the key decisions made.

From the outset we decided to opt for choices which were time-saving to implement, but still generic enough to be considered a suitable first architecture for design space explorations in both hardware and software. Also, to enable easy future extensions of the framework, modularity were considered a crucial factor when finalizing the requirements. For the same reasons the novelty of our architecture has been given a low priority so novel ideas were only brought to completion whenever it was considered the more time-efficient alternative. E.g. when implementing atomic hardware operations, in the choice between adding cache-coherence (the conventional way) and creating our own solution, the former option was discarded as it was considered too time-consuming.

Task T1 asks for multiple processor cores. We choose to fulfill this aspect by using a simple core as one of the two core designs required and then modifying this with new features to provide the second design. Modifying the core to exploit more ILP was considered to daunting for our given time budget. For instance, adding support for speculative execution would require substantial redesign of an already functioning processor pipeline, incurring regression test costs in addition to the already high costs associated with complex logic. Instead we choose to add a floating point accelerator to the design whose complexity is contained within itself.

Seeing as how the assignment tasks us to make a framework and not a complete processor design from scratch, we opted for the simple core design to be a third-party component. This was in part a consequence of our wish for supporting the GCC tool-chain for easy software development. Since supporting GCC meant we had to implement a GCC supported ISA fully we would rather use a third-party design and use the time saved to focus on the overall framework.

Additionally, the core design should allow for modifications so adding an accelerator is possible. This meant we would not be using Xilinx IP from our development environment. MIPS Technologies provides complete cores, but since these are not free this was not considered an option. Instead we focused on finding suitable designs on OpenCores[12].

To evaluate third-party modules we assume that we can find suitable and working designs without too much effort wasted. Also, while using freely available modules for the more complex parts such as the cores and accelerator should allow us to focus on the integration and the overall architecture, there is a trade-off. Time and effort have to be redirected to learning these modules, possibly extending them to meet our requirements and tailor them to fit in the design. Even so this was considered a less time-consuming approach than to write and test similar modules from scratch ourselves.

An added benefit from using open sourced third-party modules is that its maintenance becomes a collective endeavor of the original authors and its users. A drawback from this is that any modifications done on the module by our part which are for internal use only and not patched upstream must be regression maintained when updating from the third-party. This effort is considered small in comparison to full maintenance, so all in all this should free resources for the users and maintainers of our framework.

When considering the requirement for a scalable interconnect, two designs were found suitable. One is a mesh based approach where each tile is connected to a neighbor on each side of itself. This is a very conventional and tried-and-true approach. The second approach is tree-based and works by dividing memory space according to the relative height of leaf nodes (i.e. tiles) in the tree. The tree-based approach, albeit interesting, was discarded in favor of a known to be reliable and seemingly simpler mesh-based design. An especially compelling reason for settling on the mesh approach is that it is deadlock free if using well-known routing strategies.

Even though the possibility for a multitude of different designs is available with a modular tile approach, for this thesis some constraints were enforced to aid and simplify the total system. The most considerable is the decision to focus on a single ISA. Even though the SHMAC architecture does not restrict the addition of cores using different ISAs, it would require inter-ISA adaptations not currently implemented. For instance, mechanisms to keep the memory space consistent with regards to endianness would be needed. The complexity introduced with these adaptions, combined with the run-time system challenges, were considered too large a work burden. Support for such a system were not needed by the research group and by constraining on the number of ISAs supported, requirements such as supporting the GCC tool-chain also becomes more feasible.

1.3 Main Contributions

This thesis gives a flexible and highly modular framework for heterogeneous multi-core processors, focused on being easy to expand and improve so it can be used for rapid prototyping. The resulting framework consists of the following contributions.

Hardware

- **C1** A mesh interconnect with a uniform router interface both internal and external to tiles. Enables encapsulation and modularity
- **C2** An integer tile as a basic core
- **C3** A floating-point accelerated tile as a advanced core
- **C4** An implementation of atomic locking in processor hardware without the use of a cache subsystem
- **C5** An atomic lock tile to support C4
- **C6** A technique called the jump tile to correctly bootstrap the different tiles in different parts of the memory space

Configuration/debug system

- **C7** A configuration system to setup different processor layouts and coordinate the hardware and software components accordingly
- **C8** Support for GCC using C7 with custom linker script and mechanisms to support shared symbols between cores
- **C9** A communication tile for external memory access and to control execution
- **C10** Support tiles such as clock tile for timing, led tile for debugging and ram tile for extending memory is also provided

Run-time system

- **C11** A mutex library utilizing the atomic locking enabled by C4 and C5

C1 answers task T2 as both processors and memory is contained within tiles and C1 constitutes an interconnect between tiles. Contributions C2 and C3 fulfills the requirement for multiple cores as required by task T1. Task T3 is handled by contribution C9. The rest of the listed contributions are functionality and aspects of the framework considered by us to have significant worth for users.

Throughout the thesis, multiple design opportunities and improvements outside our scope have been located and since documented in this thesis. We mention this briefly here even though it should not be considered a main contribution. On the other hand, as heterogeneous processing is an ongoing research topic, these remarks and observations are of interest, especially when considering that the delivered framework will be put to use in active research.

In addition we have created microbenchmarks which tests the memory access performance. Benchmark showing parallel performance and relative speedup is also included.

1.4 Report Outline

The rest of the report is organized as follows:

Chapter 2, Background, introduces the theoretical basis for the implemented architecture, in addition to a broader basis needed for understanding the different challenges arising when designing an Heterogenous Multicore Processor (HMP). This chapter will also include theory about programming models useful for these architectures. Chapter 3, The SHMAC, gives an overview of the implemented architecture. Here we present a top-down view of the system, the tile concept, its interconnect and memory system. Finally the programming model, configuration and run-time system is also described.

Implementation details for hardware is given in Chapter 4, Detailed Design. This presents the different tiles and their inner workings, including the different cores' implementations and basic units such as the interconnect switch and routing. Software implementation is given in Chapter 5, SHMAC Software. Chapter 6, Verification, gives methodology and test strategy for verification of the design.

Chapter 7, Results, lists the results from this work. This includes microbenchmarks showing memory access performance, parallel performance for the processor and synthesis time for the design. Chapter 8, Discussion, compares the obtained results with the requirements of this thesis with regard to programmability, scalability and performance and examines the merit of our contributions, as well as pinpointing possible improvements to shortcomings identified during our implementation and discusses alternative designs. Concluding remarks and outlined future work is given in Chapter 9, Conclusion.

2 Background

This chapter gives the theoretical background needed for this thesis. This includes a brief introduction to analytical models, design of multi-core processors, scheduling algorithms and routing algorithms. Our main focus is on heterogeneous multi-core processors, but as a natural part of this discourse we will also consider homogeneous multi-cores.

2.1 Analytical Models of Computer Systems

2.1.1 Amdahl's law

Amdahl's law states that the upper bound on the parallel speedup for any application is given by $1/(1-p)$, where p is the fraction of the program that can be run in parallel [13].

This assumes that the fraction is independent of the size of the problem that is solved, while this is not necessarily so. By relaxing the assumption, Gustafson comes up with a more optimistic result regarding maximal speedup. The two can be combined as equation 2.1 [4], when assuming homogeneous cores:

$$S = \frac{N}{\frac{s(N-1)}{s+\sqrt[n]{N(1-s)}+1}} \tag{2.1}$$

This equation gives speedup, S, with regard to serial fraction, s. $n = 1$ is equal to Gustafson's law, while $n = \infty$ is equal to Amdahl's law. A plot showing this is given in Figure 2.1.

In reality, maximum speedup for a program will be somewhere in between Amdahl's and Gustafson's law and the value of n will be application specific. Both Amdahl and Gustafson tries to express how the critical path of a program limits maximal potential speedup.

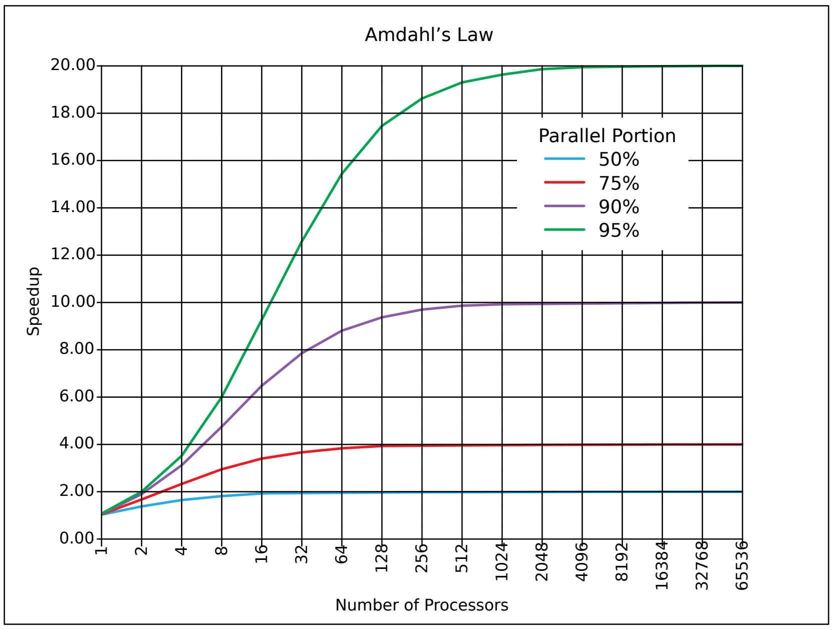


Figure 2.1: Plot showing maximal speedup for an application given its parallel portion. (Reproduced from [14])

Hill and Marty [8] describes maximal parallel speedup with respect to the software fraction that is parallelizable (f), number of cores (n) and the performance of it's cores. The equation assumes that one core is used to execute the sequential part with a performance $perf(r)$.

$$Speedup_{symmetric}(f,n,r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f\times r}{perf(r)\times n}} \tag{2.2}$$

$$Speedup_{asymmetric}(f,n,r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{perf(r)+n-r}} \tag{2.3}$$

With heterogeneous processors it may be possible to use high performance cores to increase efficiency on sequential tasks while still being able to exploit parallelism. This can be done by reducing execution time of the critical path of the program. Optimal execution of a program therefore utilize all processors in such a way that all resources are dispatched in the order to make the program's critical path as short as possible. If all cores are equal (homogeneous cores), a work intensive task will not be given more resources than any other task. If the other cores must idle during such work intensive tasks then the resources could have been spent more efficiently.

2.1.2 Pollack's Rule

Pollack's Rule states that microprocessor performance increases due to micro-architecture advances is roughly proportional to the square root of the increase in complexity [15].

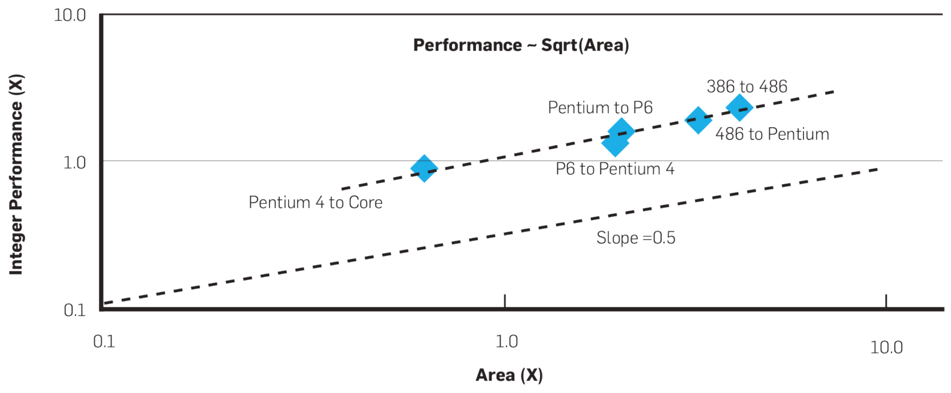


Figure 2.2: Integer performance with regard to area. (Reproduced from [7])

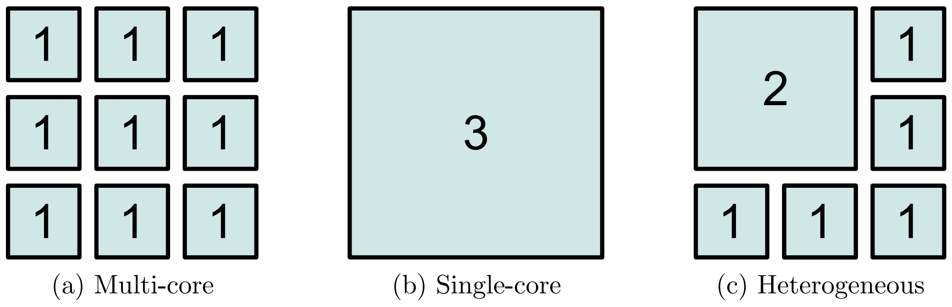


Figure 2.3: Examples of architectures with different complexity, showing performance with regard to complexity.

Figure 2.2 exemplifies this with a plot of integer performance versus area for a selection of processors.

An advanced core using nine times the resources as a simple core, would according to this rule give three times the performance as illustrated by Figure 2.3. Figure 2.3a shows a multi-core processor with simple cores having a complexity and performance of 1, while Figure 2.3b shows an advanced core with a complexity of 9, which according to Pollack's rule obtains a performance of $\sqrt{9} = 3$. Figure 2.3c shows a heterogeneous architecture with one core with complexity of 4 and performance of 2 in addition to simple cores. The exact distribution of resources used for the advanced core versus the simple cores is set arbitrary for illustration purposes, as the optimal distribution would depend on the type of applications this processor is expected to run.

Table 2.1 shows execution time (number of cycles) for one single-threaded and one perfectly parallel program doing the same amount of work. From these extremes the generalized view is that assuming the program contains enough parallelism it will be ideal to let it run on many small cores instead of fewer advanced cores. For the sequential single-thread example, i.e. there is not enough parallelism within the program, most cores would be idle and an advanced super-core would utilize resources better. As can be seen from the table, if the program's parallel portion is e.g. 80 % then a heterogeneous approach is preferable.

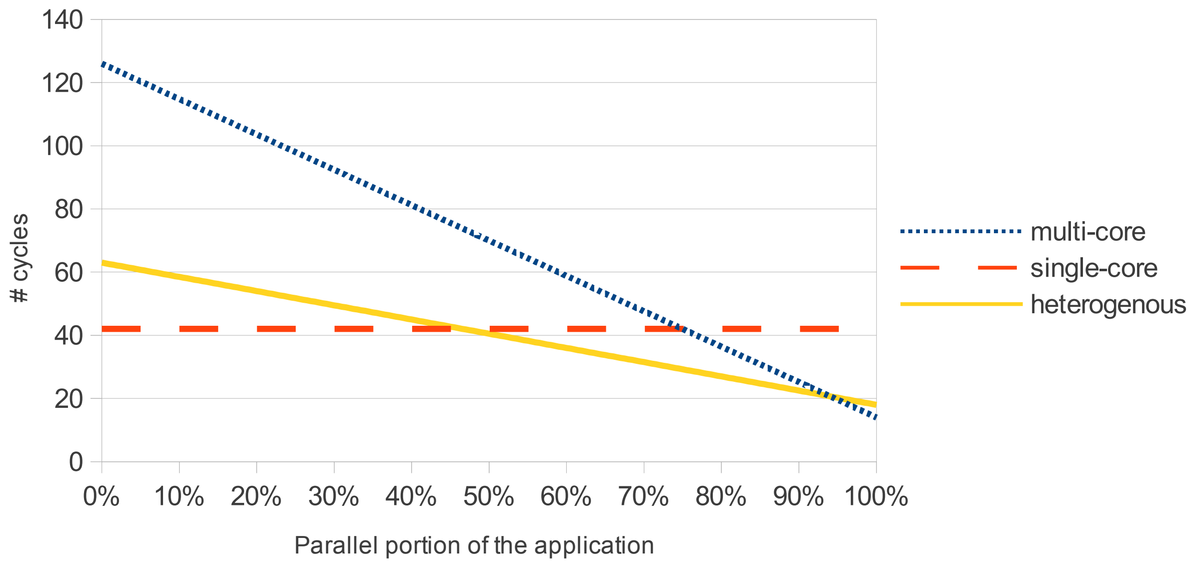


Figure 2.4: Run-time example for various processor architectures showing execution time versus parallel portion of the application

Heterogeneous processors is a combination of the two extremes outlined above. It allows sequential parts such as execution start-up to be run on a super-core. Keeping Pollack's rule in mind, the parallel parts would be run on a multitude of simpler cores. Execution time is given by sequential run-time $\times (1 - f) +$ parallel run-time $\times f$. These calculations are only a rough estimate and omits important aspects such as memory interfaces, caching and more.

Table 2.1: Run time example for different processor architectures, where the parallel portion of the program is given as f.

Program	Work	Multi Core	Single Core	Heterogeneous
sequential	126	$\frac{126}{1} =$ 126.0	$\frac{126}{3} =$ 42.0	$\frac{126}{2} =$ 63.0
parallel	126	$\frac{126}{9} =$ 14.0	$\frac{126}{3} =$ 42.0	$\frac{126}{7} =$ 18.0
f = 80 %	126	36.4	42.0	27.0

From this simple example it can be observed that the sequential parts of a program can significantly slow down overall performance, as shown by Amdahl [8]. For our example the heterogeneous core will outperform the single core processor even if when only half the program can be parallelised. In the same example the multicore will only be faster when the program is close to absolute parallel, as shown in Figure 2.4.

Also, the optimal distribution of advanced versus simple cores and the amount of resources allocated to each advanced core will be application specific. This is dependent on the amount of sequential work in the applications [8]. It is important to keep in mind that there may be parts of the program with different degrees of parallelization such that one could benefit from having more than one super-core. For example if some parts of the program only have a few threads while the rest is embarrassing parallel.

2.2 Chip Multiprocessors

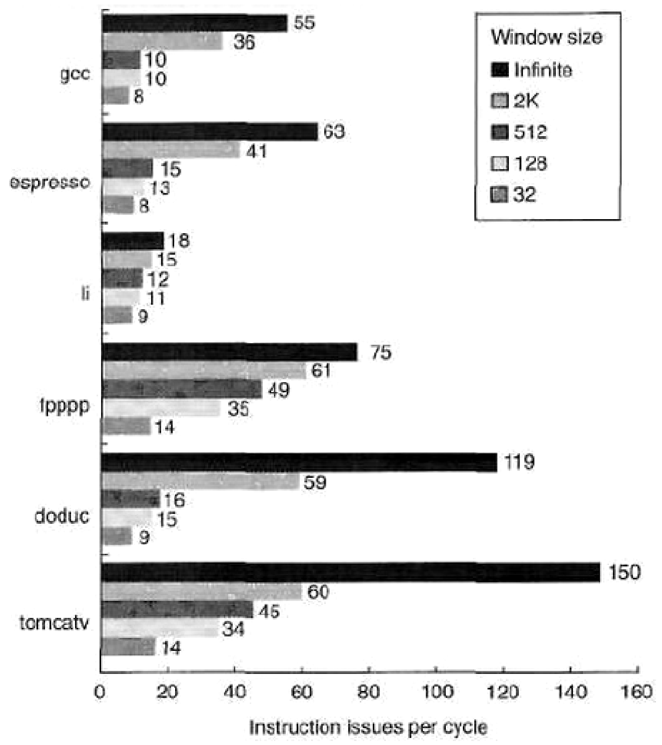


Figure 2.5: ILP available in a perfect processor for size of the SPEC92 benchmarks. (Reproduced from [1, p. 157])

Lately there have been a shift from focusing on increased single-thread performance, mainly driven by exploiting ILP and increasing clock frequency, to maximizing throughput by using multi-core processors.

There are three primary factors that have led to this shift:

The ILP wall To keep increasing frequency, processors needs to find enough parallelism in a instruction stream demands reordering instructions and advanced techniques to keep the processor busy. This requires speculative executions with complicated, dynamically-scheduled processors [16]. Even with these advanced techniques, finding enough parallelism to keep a high-performance processor occupied is hard. Figure 2.5 lists the available ILP for six of the SPEC92 benchmarks, where the last three benchmarks are loop-intensive floating-point programs. This figure shows that even if there is a lot of available ILP, not all of this can be exploited due to limitations in window size (as shown in the figure) for finding ILP, imperfect branch and jump predictions and finite number of virtual registers [1, p. 154].

The power wall Power consumption is proportional with frequency, and depending on the cooling system, power can become the limiting factor for the maximum obtainable single core performance [4].

The memory wall There has been a growing disparity of the processors performance and the main memory latency [17].

One of the greatest challenges this shift have brought is increased complexity for application development as exploiting all available resources requires the software to be written with parallelism in mind.

2.2.1 ILP focused CMP

ILP techniques exploit the inherent parallelism between instructions. Several instructions can be executed simultaneously (super-scalar) or multiple instructions' can be in-flight at different parts of the pipeline at once. One example of a processor utilizing these techniques is the Pentium 4, which includes massive out-of-order execution, deep pipelines and advanced branch prediction [18].

Hennessy and Patterson [1, p. 67] lists loop-level parallelism as the simplest and most common way to exploit ILP, that is to exploit parallelism among iterations of a loop. One example of code inheriting this parallelism is given in Listing 1. Each `N` iteration is independent, allowing them to run in parallel. This is typically exploited by either allowing the compiler to create vector instructions utilizing the Central Processing Unit (CPU)'s SIMD unit, by loop unrolling or in more advanced processors let the out-of-order unit handle loop-level parallelism automatically. Since it may be costly to flush a deep pipeline, many processors reorder instructions to avoid running instructions with dependencies at the same time.

Listing 1 Loop-level Parallelism

```
for (int i=0; i<N; ++i)
  x[i] = x[i] + y[i];
```

Very Long Instruction Word (WLIW) processors are designed to exploit ILP without adding much complexity, with the motivation that single instructions does not use all of the processors resources. In these processors, instructions are bundled. A bundle is a fixed number of instructions which can be executed simultaneously. For instance that one multiplication instruction and three addition instructions can be done in parallel and can therefore be added to the same bundle. These families of architectures is popular in embedded systems.

2.2.2 TLP focused CMP

Sun Microsystem's UltraSPARC T1 (Niagara 8 Core) is an processor which focuses on maximizing Task Level Parallelism (TLP) instead of ILP. This design opts out on techniques such as multiple instruction issue, out-of-order issue and aggressive branch prediction. This allows for keeping each core simple which increases the number of cores that may fit into a chip [19].

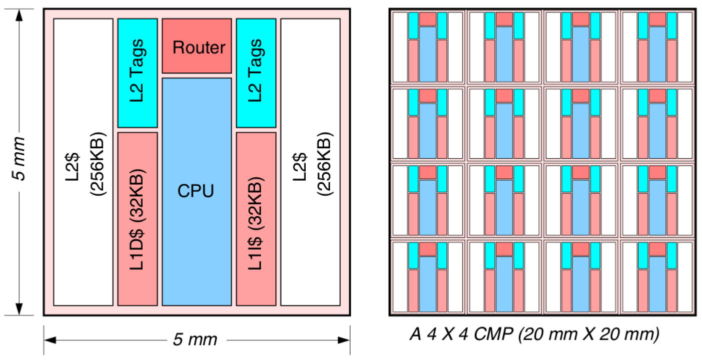


Figure 2.6: Tile concept with CPU, cache and router. (Reproduced from [11])

It is a barrel processor support 32 threads and switches thread each clock. If a thread is waiting for memory it is simply skipped. With this architecture the processor may operate efficient even when threads are waiting for memory, reducing the need for advanced caching.

Another processor type focused on TLP is Graphical Processing Units (GPUs). This is however a separate unit giving rise to significant latency when copying data to and from the unit.

2.2.2.1 Tiled Chip Multiprocessors

The term Tiled Chip multiprocessor was introduced in 2005 by Zhang and Asanovic [11]. The main idea introduced is to create a regular structure of modular processing cores named tiles. These tiles are replicated throughout the chip and connected using an interconnect. Figure 2.6 shows, that by using replicated cores, one may create a less error prone architecture, as each core may be simpler and tested independently.

Tilera is currently shipping processors with up to 100 cores. The TILE-Gx100 processor family has an operating frequency of 1.5 GHz and typical power usage of 10 to 55W. I.e. both power and frequency lower than for typical state-of-the-art server processors. This processor includes in addition on-chip hardware accelerators for encryption and compression.

A paper by Facebook and Tilera shows that for a tuned version of Memcached *"a TILEPro64-based S2Q server with 8 processors handles at least three times as many transactions per second per Watt as the x86-based servers with the same memory footprint"*, and had a performance advantage many times higher than Intel XEON and AMD Opteron processors [20].

Figure 2.7 shows that this family of processors uses multiple copies of the same core interconnected with a mesh using packet switching. Each tile consists of a processor with L1 and L2 cache, in addition to a non-blocking switch. Each processor is connected with a two-dimensional on-chip mesh network.

To provide more deterministic throughput several parallel meshes are used with different transaction types. To keep communication latency low and scalable bandwidth, more

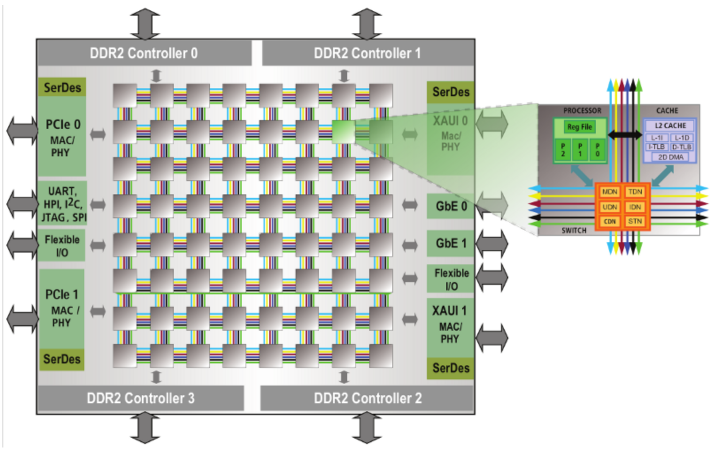


Figure 2.7: Overview of the TILEPro64^{TM} processor. (Reproduced from [21])

parallel networks could be added as the number of tiles increases. In order to keep power efficiency high, idle tiles can be set to a low-power sleep mode [21].

2.2.3 Heterogeneous Chip Multiprocessors

A heterogeneous multi-core architecture is a Chip Multiprocessor (CMP) composed of cores with different properties regarding size, performance and complexity and different functionality such as cryptographic accelerators or video decoding.

As the heterogeneous processor consists of both simple cores and more powerful cores, it combines the power of efficient parallel performance of multi-core processors without neglecting sequential performance.

Kumar et al. [22] shows through simulation that their heterogeneous multi-core design has the potential to increase energy efficiency by a factor of three. This work shows that most energy efficiency can be obtained even when using as few as two cores. The simulated processors are reusing four different Alpha cores, with different complexity and power usage. Minor differences between the cores are handled by using either the least common denominator of the supported ISAs or by using software traps.

2.3 Software for Heterogeneous CMPs

Frameworks such as Compute Unified Device Architecture (CUDA) and OpenCL are specialized for Single Instruction, Multiple Data (SIMD) operations by utilizing GPUs. These are languages which requires that the programmer reason about the architecture. They may therefore be best suited when the parallelism is regular and typically the same instruction should be executed on multiple data.

Topcuoglu et al. [23] suggests that the existing scheduling algorithms for heterogeneous architectures is not generally efficient because of their high complexity and/or insufficient results. Two new algorithms is presented: Heterogeneous Earliest-Finish-Time

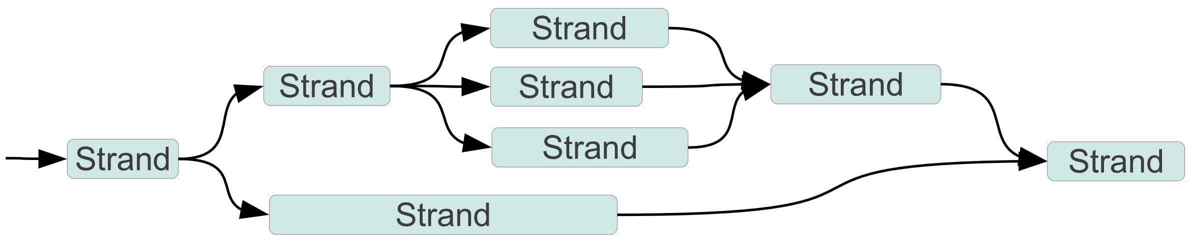


Figure 2.8: Execution model using strands. (Reproduced from [25])

(HEFT) and Critical-Path-on-a-Processor (CPOP).

Directed Acyclic Graph (DAG) scheduling is used in many applications. It uses partial ordering, which can be represented as a DAG. General DAG scheduling is NP-complete, and there have been much research in finding heuristics for this problem. Most of these are, however, targeting homogeneous processors.

The HEFT algorithm selects the task with the highest upward rank at each step and assigns this to the most suitable processor that minimizes the earliest finish time. CPOP schedules critical-path nodes onto a single processor in order to minimizes the critical path length. HEFT is claimed to outperform all other tested algorithms, but also CPOP seems to perform better or equal to existing algorithms.

HEFT has been observed to be significantly affected by how weights are assigned to the nodes and edges of the DAG [24]. This study focuses on five heuristics for heterogeneous DAG-scheduling; HEFT, CPOP, Dynamic Level Scheduling (DLS), Fastest Critical Path (FCP) and Levelized-Min Time (LMT). Sakellariou and Zhao [24] suggests using an hybrid approach for DAG scheduling by reducing the problem to smaller sub-problems for scheduling independent tasks, leading to their Balanced Minimum Completion Time heuristic.

TBP is a form of parallelization, focusing on parallelising tasks in contrast to ILP and data parallelism which is much used in most modern processors using SIMD-instructions and automatic reordering by the processor.

Using the definitions from *Intel© Cilk^{TM} Plus*, a strand is a sequence of instructions without any spawn or sync points. A spawn occurs where one strand ends and begins two new ones, and a sync ends one or more strands and starts a new strand. All new strands can be run in parallel with each other.

All strands of a programs execution can be seen as a DAG, where spawns are viewed as outgoing edges and syncs as incoming edges. This is shown in Figure 2.8.

2.3.1 Work Stealing

Optimal DAG scheduling is known to be NP-hard [26]. As the execution of a program may be viewed as a DAG, finding a good online algorithm for scheduling with minimal overhead is important.

Well known implementations of TBP-libraries such as Wool [27] (C) and Intel© Cilk$^{\text{TM}}$

Plus [28] (C/C++) use an algorithm called work stealing for scheduling work to each processor. This algorithm is designed to minimize the number of times work is moved between the processors [29].

The original paper on Cilk [28] proves that the upper bounds of resources used is:

Space The memory used by a P-processor execution follows $S_P \leq S_1 \times P$, where S_1 is the memory usage of a serial execution of the Cilk program.

Time With P processors, the expected execution time is bounded by $T_P = O(T1/(P + T_\infty))$, where $T1$ is the execution time by its serial elision.

Communication The expected number of bytes communicated during execution is $O(T_\infty \times P \times S_{max})$, where S_{max} is the largest size of any closure.

Even though work scheduling is known to be NP-complete, all these bounds are proved to be within a constant factor of the optimal [28].

2.3.2 Architecture-aware Task-scheduling

The execution time of a program will be bound by the critical path of the application's execution DAG. Both Wool and Intel© Cilk™ Plus assumes a homogeneous architecture. An heterogeneous architecture can shorten the critical path by allocating task on this path to higher performance cores.

There has been some research on architecture-aware task-scheduling with different objectives, such as minimizing the processors temperature [30], minimize energy usage [31] and maximizing performance [23]. Task-scheduling for heterogeneous processors is a complex problem as the scheduler needs to be able to find a matching between the tasks and cores without significant cost in performance and/or power usage. This still remains an active field of research.

2.4 Interconnect

Interconnects is an essential part of multi-core System-on-Chips (SoCs) design since typical workloads utilize communication patterns such as map-reduce and all-to-all. Using a single communication interface between all parts of the system also allows for easy modularization and abstraction. Such networks can utilize both packet and circuit switching, but only packet switching will be covered in this report. The mesh is well-known and is thoroughly analyzed elsewhere.

A mesh arranged in n-dimensions with k nodes in each dimension contains a total of $N = k^n$ nodes. The number of dimensions chosen have an impact on obtained throughput and latency, but to keep wires short and to minimize serialization latency the lowest dimension number that still allows for maximum throughput should be chosen [32]. The number of dimensions may be increased as the number of cores becomes sufficiently large, to allow the interconnect to scale well with regard to bandwidth.

2.4.1 Routing Algorithms

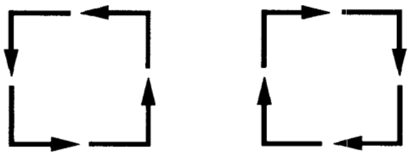


Figure 2.9: The possible abstract cycles and turns in a 2D mesh. (Reproduced from [33])

For n-dimensional mesh topologies, dimension order routing algorithms are proven to create deadlock-free routing. These algorithms are very popular, and includes the routing algorithms Turn Model and XY routing [32]. Figure 2.9 shows the eight possible turns for a two dimensional mesh. Dimensional order routing algorithms imposes restrictions on which of these turns that should be allowed to avoid possible deadlocks.

Using a torus instead of a mesh gives the network a smaller diameter, which can improve latency and efficiency in the network. Making a routing algorithm deadlock-free in a torus will however require two virtual channels, in contrast to a mesh where one channel suffices. Utilizing a torus network would require other algorithms than for a mesh and is left as future work [34].

2.4.1.1 The Turn Model

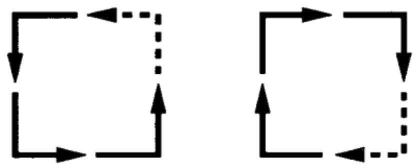
The Turn Model is an deadlock and livelock free routing algorithm, which is adaptive and restricts packets to the shortest possible path [33]. The path between two nodes is determined based on the network load. By disallowing some turns, such that no cycles can be formed, the routing can be done deadlock free [32, p. 268]. Figure 2.10a shows the six turns allowed by the west-first algorithm, which is one of the possible routing algorithms within the turn model. Figure 2.10b gives an example of this algorithm for an 8x8 mesh.

2.4.1.2 XY routing

Another, more restrictive, dimensional order routing algorithm applicable for 2D meshes is the XY routing algorithm. Figure 2.11a shows the allowed turns for this algorithm, with an example given in Figure 2.11. This algorithm will route packets in the X direction until it is on the correct column before routing the packet in the Y direction.

This algorithm gives a deterministic routing, where two packets with the same source and destination will cross the same route without considering traffic. There will therefore be no reordering of memory accesses due to routing.

XY routing also guarantees that a packet will, assuming use of a mesh topology, take the shortest possible path in the network, i.e. the Manhattan distance. It does however seem like there is a tendency of accumulation of packets around the center of the network when there is significant traffic [35].


(a) The six turns allowed by the west-first algorithm. (Reproduced from [33])

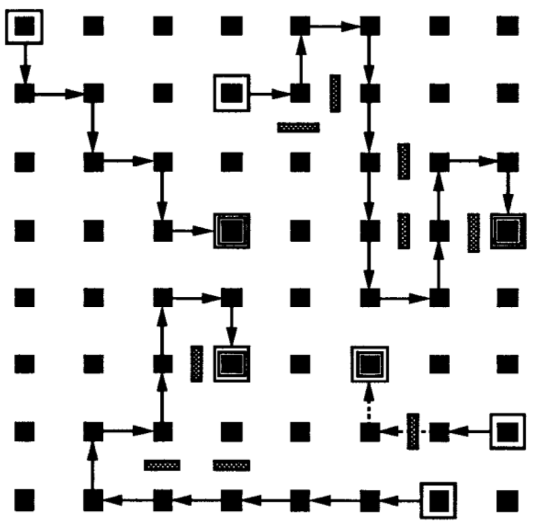
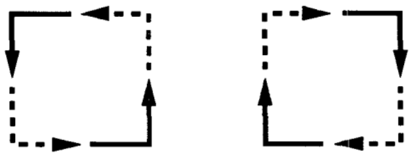

(b) Examples of the west-first algorithm. (Reproduced from [33])

Figure 2.10: The west-first algorithm.

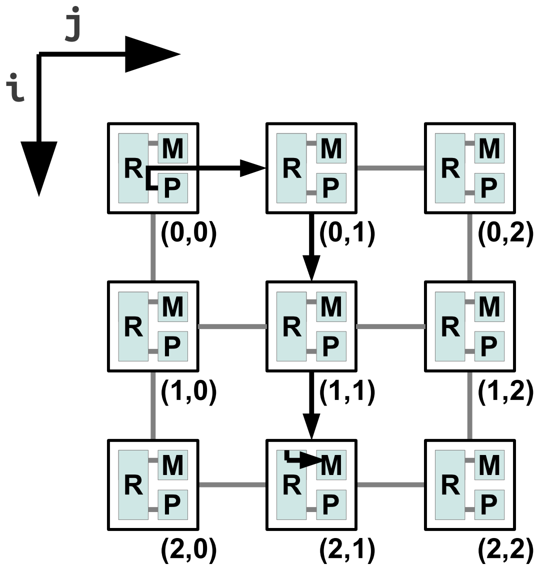
2.4.2 Wormhole Flow Control

Cut-through switching is a method for packet switching networks, where the switch start forwarding a packet before the whole frame is received. Wormhole flow control is a form for flit-buffer flow control, which operates like cut-through, but the channel and buffers are allocated to flits rather than packets [32].

Each network packet is broken into a number of smaller packets named flits. When the head flit arrives at a node it acquires a virtual channel, one flit buffer and one flit of the channels bandwidth. Each following flit is routed using this virtual channel until the arrival of the tail flit. The tail flit releases the virtual channel as it passes. Using wormhole flow control the number of lines between each tiles could be decreased, but will lead to higher latency than sending full packets.



(a) The four turns allowed by the XY routing algorithm. (Reproduced from [33])



(b) Example of XY routing

Figure 2.11: The XY routing algorithm

3 The SHMAC

This chapter presents an overview of the processor architecture while the details about the implementation are deferred until Chapter 4.

Figure 3.1 gives an overview of the implemented architecture, SHMAC, which consists of a number of independent tiles interconnected using a mesh. Each tile is allocated a subset of the global memory's address space, referred to as its local memory. This makes SHMAC a Non-Uniform Memory Access (NUMA) processor as the memory is distributed and the mesh interconnect access times are dependent on the distance between tiles.

The interconnection network is organized as a mesh with full-duplex channels between neighboring tiles. This interconnect is hardwired for a two dimensional network with fixed bandwidth. The network transports memory requests and responses in a packet format. A master unit sends a request to a slave unit over the mesh, which the slave unit then must handle appropriately. Figure 3.2 shows how tiles in a mesh structure can be mapped to coordinates in a regular fashion. Based on such a regular coordinate structure, routing can be carried out in a simple manner.

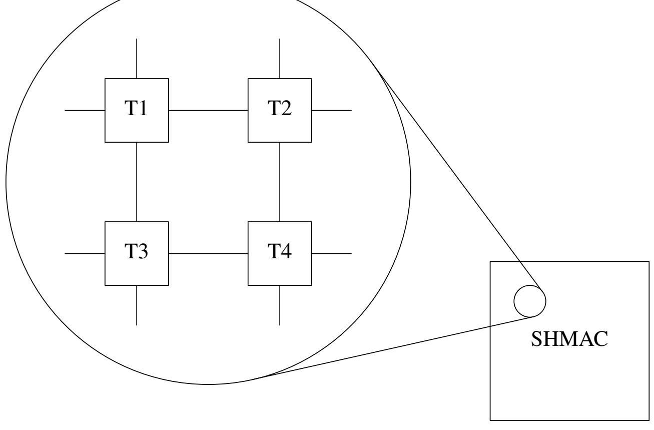


Figure 3.1: Top-level overview of the SHMAC processor showing organization of tiles and interconnection network

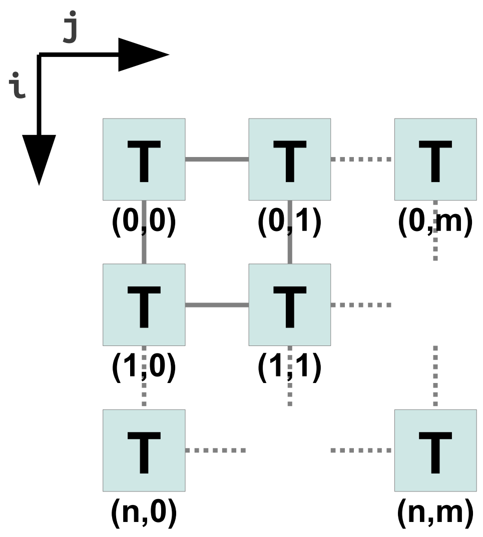


Figure 3.2: Mesh organization of tiles with given coordinates.

The concept of using replicated tiles throughout the design does not require the use of a mesh network. If a new router module is created, SHMAC could use alternate topologies such as discussed in the Piranha project [36].

3.1 Instruction Set Architecture

The SHMAC is a single-ISA architecture, similar to the architecture given in [22]. All of the processor cores should therefore be able to execute the same instructions, i.e. same program binaries, even though the different cores have different capabilities. Single-ISA allows a scheduling run-time system to schedule tasks to arbitrary cores. That is all processor cores should be able to execute the same code, but the performance of this execution may suffer if scheduling does not take the individual cores' capabilities into account.

The current ISA supported is the MIPS-I, which is chosen on the basis of it being a relatively small Reduced Instruction Set Computer (RISC) architecture supported by GCC. MIPS also facilitate the use of coprocessors, which makes it simpler to expand the processor, e.g. by adding a floating-point unit. Its 32 bit-architecture helps to keep the logic footprint small. Also, multiple third-party open-source implementations are available for the MIPS architecture, which lowers the risk for the project as one of its main components is easily replaced in case of serious bugs or other deficiencies.

The MIPS-ISA allows coprocessors to throw exceptions if operations is not implemented. It also allows a processor to set an exception if coprocessor 1, which is reserved for Floating-Point Unit (FPU), is not present. This can be used for full floating point support in both cores with and without an FPU module by implementing floating-point support in software. Hardware traps can also allow for the FPU to support only a subset of the floating-point operations. Jumping to interrupt routines which implement the missing FPU operations while operating on the data stored in the FPU register file is readily possible with CPU-to-FPU register instructions in the MIPS-ISA.

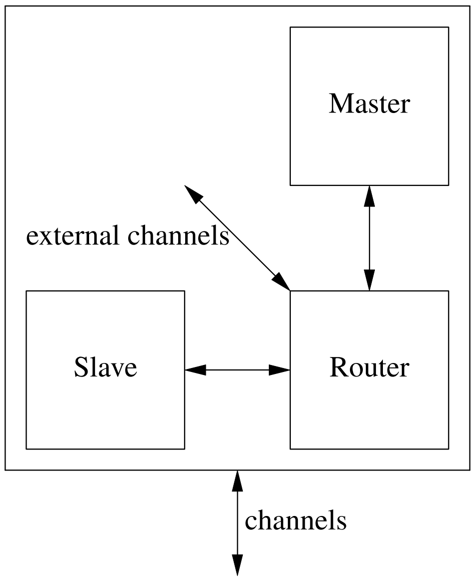


Figure 3.3: An overview of a generic SHMAC-tile

3.2 Tile Organization

Figure 3.3 gives an overview of the content of a generic tile. Each tile consists of an optional master unit, a slave unit and a router unit.

Master Unit sending memory requests

Slave Unit handling memory requests – must send reply on memory read operations

Router Forward packets to other tiles or internal if tile is destination

Master units will typically be processor cores, while slave units typically will be on-chip block Random Access Memory (RAM). Slave units may also be control of Light-Emitting Diodes (LEDs), off-chip memory and other units that can be controlled by memory accesses.

Any request to a tile's memory range will be handled by this tiles slave unit, and will handle this request according to if it is a read or write operation. For a read operation it must return an reply to the master unit. The slave unit is not required to function as RAM, such that how the unit handles request to its address range is specific for each tile; The only requirement is that if this is a read request the unit must send a response. If the local address is not valid, this response should include a flag indicating this.

If a tile is not making use of its associated local memory space the architecture still requires that external memory accesses are handled as according to the specifications, i.e. returning a reply to any read requests. This is the tile's responsibility associated with being allocated a subset of the global memory space. A dummy slave unit which simply replies to any read request with zeroes is provided to uphold this responsibility for tiles not utilizing a slave. Such a tile will in the report be referred to as a pure master tile, with a pure slave tile being a tile without a master.

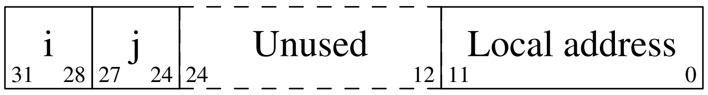


Figure 3.4: Global address layout

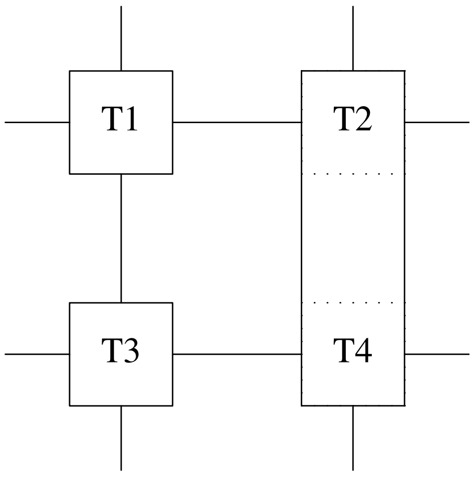


Figure 3.5: Virtual tiles merged

3.3 Memory Layout

SHMAC uses a NUMA layout. All tiles shares the same memory space wherein each tile have a local memory, but are still able to access all other memory locations in the processor. As the processing cores are without cache, this makes the tiles very sensitive to memory access patterns.

Figure 3.4 shows the global memory address layout. Each tile is given a subset of the memory map, where the most significant bits of the memory address maps to the tile responsible for it. Specifically the tile's coordinates (i,j) is used as seen in the figure to partition the global memory space. The remaining bits gives the local memory byte address space, wherein only a subset of the addresses is used. The SHMAC defines the global memory space as little-endian. This means that all data shared with other tiles is assumed to use this convention. A big-endian tile's data should therefore be converted when crossing this tile's interface.

Currently the SHMAC is configured to use eight bit for coordinates, allowing $2^8 = 256$ cores with local memory limited to 2^{24} B (16 MB). A research goal of reaching 1024 cores would give a local memory of only 2^{22} B (4 MB). Further still, most tiles will have an even smaller memory than this. This according to how much memory resources are available per tile when reaching a high number of cores, but also by what is required for the tile's functionality. In other words, the number of unused bits in the local memory address will vary between tiles. The SHMAC also does not require that the size of the mesh is a power of two. This allows that not all the values of the coordinate fields of Figure 3.4 will map to tiles and therefore will not map to memory addresses.

Multiple tiles can be combined virtually so that a physical tile can have a larger address space. The need for such a solution becomes more apparent the larger the maximum

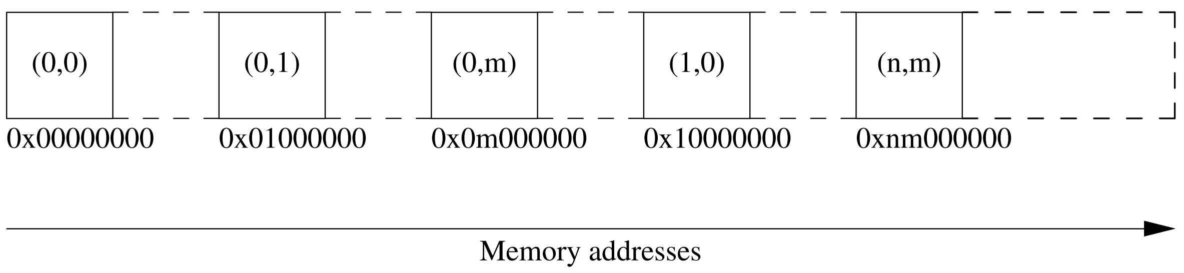


Figure 3.6: Global Memory Space Layout

count of tiles supported becomes, as the architecture divides the global memory space equally according to the maximum number of tiles possible. When tiles require a larger address range to function properly, then one possibility is to use as many tile slots as needed. This will require adapter logic, for instance a tile made to consume a given number of tile slots to produce a single tile slot. Figure 3.5 shows an example of where all tiles at the right border would be connected to the same physical tile, i.e. where tile T2 and T4 are virtual.

For tiles which include a processing core, the start of its local memory contains the program to be executed. The start of a tile's local memory is on the other hand not necessary the memory address the processing core asks to fetch first. Instead of fixing this start address in the cores, we fix it at the memory location. If a processing core which is included in a specific SHMAC layout starts by requesting instruction address a, then we solve this by placing what we call a Jump Tile at the coordinate responsible for the global address a. The Jump Tile's responsibility will then be to make sure that the different cores will start executing at the its associated tile's local memory instead. Typically the first address fetched for processors will be address 0 so a Jump Tile is normally positioned at $(0, 0)$ in the layout.

$$Address_{start} = i << (32 - 4) + j << (32 - 8) + local \tag{3.1}$$

The first memory address associated with each tile is given by Equation 3.1, assuming an 16×16 mesh. Figure 3.6 exemplifies how the global memory space is partitioned for such interconnect parameters, where n and m are hexadecimal digits.

4 Detailed Design

This chapter presents the details of the SHMAC design. Some overall tile concepts are further elaborated before the different tiles are presented. Both implementation and functionality are examined. Finally the interconnect is detailed.

Table 4.1 gives an overview of all implemented tiles. Some points are worth remarking. The LED Tile does no address checking, as all of its memory locations are aliases to the same set of LEDs. Some tiles are very restricted in the resources available on the development board. Such tile types are marked with the current maximum number of tiles possible. More details is given are given in the respective sections for each tile.

Table 4.1: Tile Details

Glyph	Reference	Description	Master	Slave	Address range
`P`	`PLASMA`	Integer Processor	`mlite`	`RAM`	12 bit : 0 – 4095
`F`	`FPU`	Floating-point Processor	`mlite+FPU`	`RAM`	12 bit : 0 – 4095
`R`	`RAM`	RAM	–	`RAM`	12 bit : 0 – 4095
`S`	`LED`	LED [Max one]	–	`LED`	Full range
`C`	`CLOCK`	Clock	–	`clock`	0 – 3
`J`	`JUMP`	Jump tile	–	`jump`	0 – 3
`L`	`LOCK`	Lock tile	–	`RAM+lock`	12 bit : 0 – 4095
`U`	`UART`	UART [Max one]	`UART`	`dummy`	N/A

As can be seen from the table, all processor tiles delivered with the current SHMAC framework uses either mlite or a mlite derivative as its master unit. A variety of different processor cores can however be fitted into a processor tile, as long as they implement the interface required to operate correctly with the router unit.

4.1 Network Packets

All communication throughout the SHMAC is done using packet switching. The layout of a packet is shown in Figure 4.1, assuming that 2×4 bits are used for tile addressing.

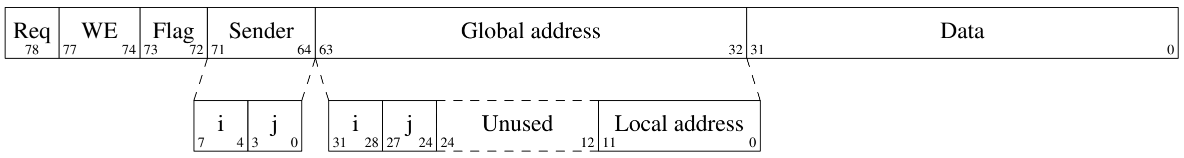


Figure 4.1: Network packet layout, assuming 2 × 4 bits are used for tile addressing.

As the sender's coordinate is included, total packet size depends on the interconnect's size. The same layout is currently used both for memory requests and responses.

Each packet contains a `req` flag, which is used to decide whether this packet is a memory response or a memory request. If the `req` flag is set the packet's destination coordinate is given by the upper bits of `Address`, that is bit 56 to 63. If this flag is not set then the packet's destination is the `sender` coordinate.

As master units are the ones making memory requests, the packets from these units will have the `req` flag set. If a packet has arrived at its destination tile it will be routed using one of its internal ports. This flag will then be used for distinguishing between these internal ports, such that if `req` is set this is a memory request and the packet should be routed to `intern_mem`. If this is not set, it will be routed to `intern_proc`.

Accesses to memory locations outside the tile's scope of responsibility will on the other hand be routed using one of the router's external channels according to the implemented routing algorithm. The router will therefore need to read the `req` field, and in according to this read either the sender's or destination's address.

The `write_enable` field (referred to as `WE` in the figure) sets the write mode of this request. Read requests are given as `0000`, while each set bit indicate a write of this byte as shown in Table 4.2. It is not required that slave units support unaligned memory accesses. Any slave unit receiving a read request must send a reply.

The packet includes a `flag` field. This field supports the enums `none`, `sync` and `interrupt`. The `sync` flag is used for Load-Linked (LL)/Store-Conditional (SC). An memory response with the `interrupt` flag set indicate that the read was unsuccessful, typically occurring from reading an invalid memory address.

An request with the codeinterrupt flag set is used for explicitly send an interrupt to an

Table 4.2: Setting memory request mode

WE	Type	Bits
`0000`	Read 4 byte	0 – 31
`0001`	Write 1 byte	0 – 7
`0010`	Write 1 byte	8 – 15
`0100`	Write 1 byte	16 – 23
`1000`	Write 1 byte	24 – 31
`0011`	Write 2 byte	0 – 15
`1100`	Write 2 byte	16 – 31
`1111`	Write 4 byte	0 – 31

slave unit. The only tile utilizing this is the LL/SC, where this performs an explicit invalidation of the requested memory location.

4.2 Basic Tile Units

Some units are used in many different tiles and are considered to be the basic building blocks when composing tiles. The most reused units in the delivered framework is the on-chip block RAM, the dummy slave and the mesh router. These will be further examined in this section. The other units are more specific for their intended tiles and are elaborated in conjunction with them.

4.2.1 On-chip Block RAM

All of the implemented processor tiles, in addition to locking tile and RAM tile, uses block memory. This unit is a minimal wrapper utilizing a 1 kB third-party memory module in a manner such that the memory can be accessed with our implemented interconnection interface.

4.2.2 Dummy Slave

As each tile owns a subset of the global memory space, all slave units are required to give an reply to any read request in order to avoid requesting master units to stall forever waiting for a response. Some tiles does not utilize its local memory space and should never receive either read or write requests. If a processor still tries to read a memory location belonging to this tile, this error should be handled correctly without leading to a stall.

The slave unit is therefore simply a unit which replies to any read request by sending a packet with the `interrupt` flag set. The `data` field of this packet is all zeros, which is a `nop` instruction in the MIPS architecture. Write request to invalid memory locations are simply dropped.

4.2.3 Mesh Router

This router is the only one implemented in the SHMAC and is used in all of the tiles. It assumes the use of a regular mesh interconnect, such that it does not utilize the SHMAC's torus.

Each router have up to six ports, including two internal ports for the master and slave units. All ports have the same interface. Each port is named after their geographical direction, i.e. `east`, `north`, `west` and `south` in addition to `intern_proc` and `intern_mem`. The two latter are internal channels, while the rests is connected to other routers.

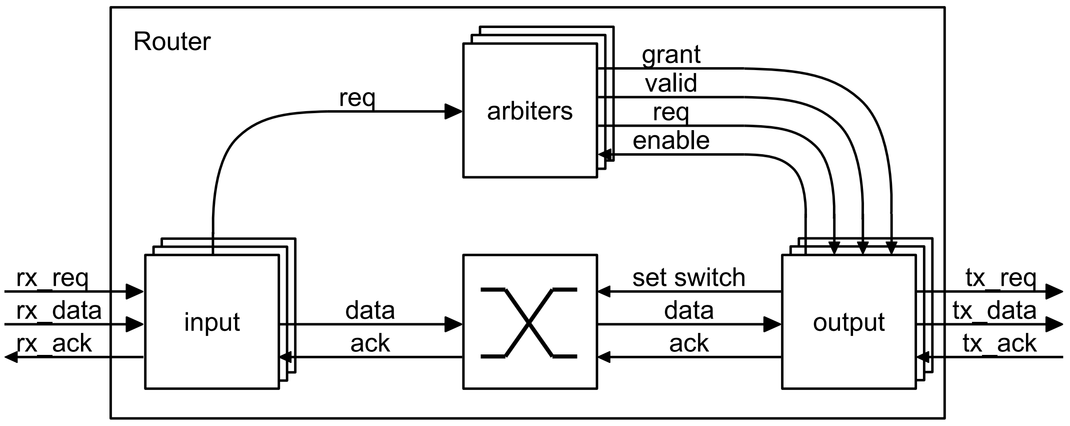


Figure 4.2: Overview of the router module

Each router consist of four sub-modules; `input ports`, `output ports`, `switch` and `arbiters` as shown in Figure 4.2. Each of these, except the switch, consist of one independent sub-module for each routing direction. That is all modules have a four sub-modules for each outgoing channel in addition to the two internal channels.

An incoming packet will be buffered in `input port`, setting `ack` to allow the sender to continue. This module then sets up an request to the correct `output port`, given by it's routing algorithm. Each `output port` is connected with an `arbiter` which decides which of the requesting `input port` that is to be handled. This arbitrator sets priority in a round-robin manner. The `output port` is then setting up it the `switch`, holding this channel open until `input port` no longer have `req` set.

This router may be done more efficient by dropping this strict handshake between `input port` and `output port`, but this implementation makes communication between modules straight forward and easy to understand.

The router uses a handshaking mechanism for interaction which all connected units must match. When `tx_req` is set, it signals that there is valid data on the `tx_data` bus. The receiving router sets the `tx_ack` when it have buffered the data, allowing the sending router to remove the data from the bus and unset the `tx_req`.

The routing is done using XY routing, by first routing in the i-direction and then in the j-direction. The routing is simply done by comparing the packets destination with the current tiles coordinate. If the packet's destination is the current tile it will be routed to either `intern_proc` or `intern_mem` as described in Section 4.1

4.3 Integer Processor Tile

The Integer Processor Tile is our basic processor tile. Figure 4.3 shows this tile's organization. We have wrapped the third-party mlite core into a master unit while using the basic RAM unit as the slave unit. The wrapper takes care of fitting the mlite's memory stalling mechanism to SHMAC's interconnect interface.

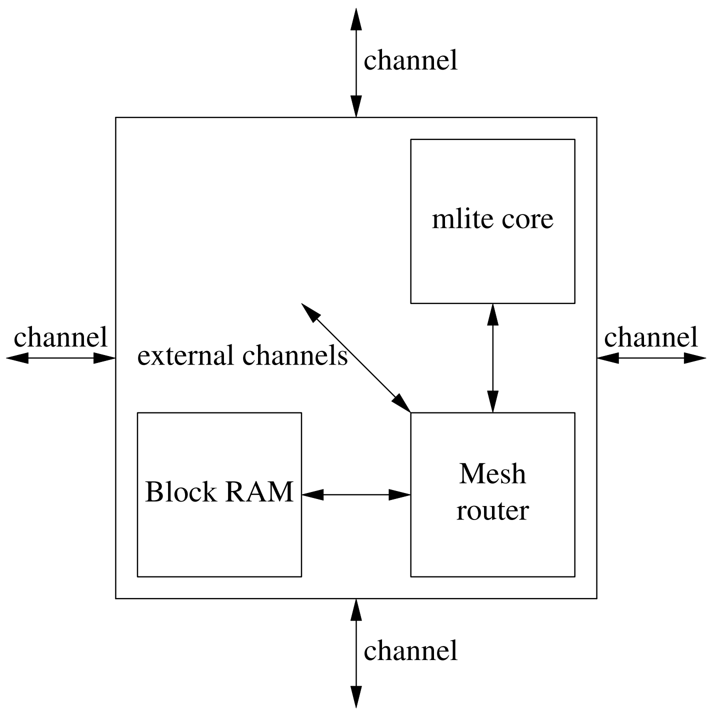


Figure 4.3: Processor tile

4.3.1 mlite

The mlite is the CPU core of the Plasma MCU by Steve Rhoads [37] found on OpenCores.org. This is a a 32-bit processor supporting most of the MIPS-I ISA, configurable to either a two or three-stage pipeline. In the Integer Processor Tile this is setup as a two stage pipeline. Figure 4.4 gives a block diagram of the core.

This core is basically wrapped into a state machine so it can be used with the SHMAC's network interface. When the state machine receives a memory request it pauses the core while waiting for the response packet. The state machine must also distinguish between data read and writes, allowing the core to immediately continue execution if it is a write operation. Read request will however force it to pause until it gets the response over the interconnect.

The mlite itself utilize the same memory port for both instructions and data fetches. This is a possible improvement opportunity, but since the shared port behavior is well-defined, i.e. the order of data versus instruction fetching is consistent, it is easy to wrap correctly. Since this is a third-party core the possibility of the order to change is present, making it possible for the state machine to break when updating the mlite design from upstream.

The order of fetches is strictly enforced by our state machine. In other words, for the current pipeline both instruction and data fetches occurring at the same time from the different pipeline stages are ordered so one must wait until the other is completed. The state machine introduces this stall to make sure there is no possibility of reordering the two fetches. If we did not make sure to wait, then given the NUMA architecture of our design a data fetch response could return before the instruction fetch which was issued

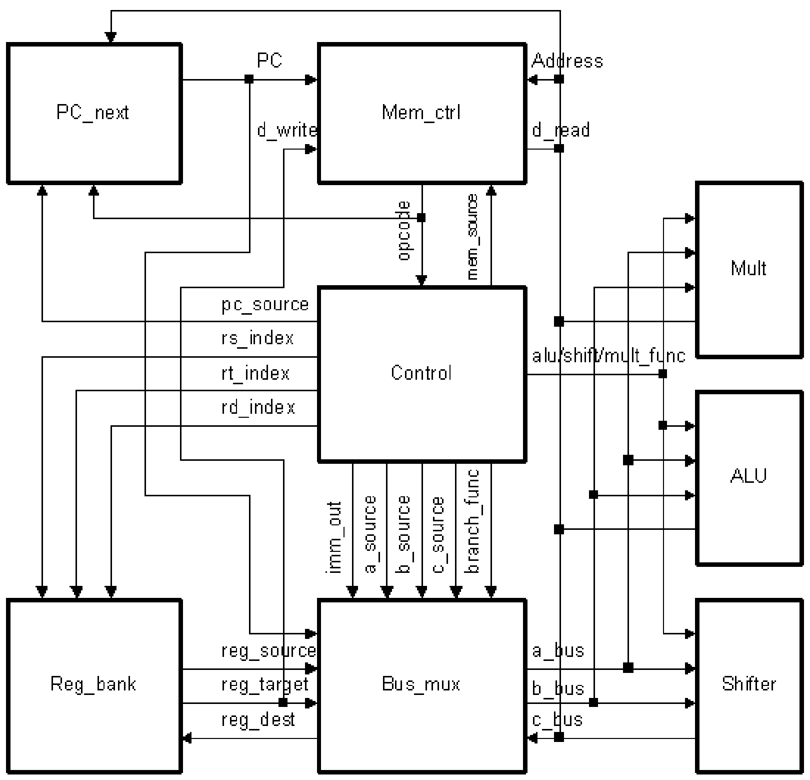


Figure 4.4: Block diagram for mlite. (Reproduced from [37])

prior. This would result in the data response being interpreted as the next instruction. On the other hand, a simple reordering station in conjunction with the state machine would rectify this. This could be a simple alternative to the two ports solution alluded to above, but would still give the mlite module an improved performance according to the degree of data fetches in the workload.

In order to support atomic locking operations, we have expanded the CPU core to support the instructions LL and SC defined in the MIPS-II ISA. These are the only two MIPS-II ISA instructions our processor tiles support, so an application must not try to use any other MIPS-II instructions. A provided locking library utilizing LL and SC may be used so that other applications do not need to enable MIPS-II support in the GCC tool-chain.

In order to support LL/SC the control unit of mlite is expanded to properly decode these instructions. The control unit was modified to decode and assert control signals in such a way that the previously available memory mechanisms could be reused with only minor modifications. The memory controller was adjusted to interpret two new control modes referring to whether a Load-Linked or Store-Conditional was being executed. The most important modification of the memory controller stems from the Store-Conditional being in our solution both a store and a load at the same pipeline step. This means that for the beginning of a Store-Conditional the execution is much like a regular store. Their differences stems from the fact that a regular store operation does not stall the processor core, while our SC does. This since the conditional part of our SC can not be determined until a response to whether or not the SC operation was successful has arrived. This is similar to a load and is performed at the end of the pipeline step for the execution of SC. Reusing the already present load and store memory control modes allowed for a cheap realization in terms of hardware and design time.

Both LL and SC requires special attention over the interconnect interface since they must be differentiated by regular requests and responses. To handle these special memory accesses a `sync`-flag was introduced to the packet format. The units capable of handling packets with this property may do so, while other units should issue an exception. Currently no exceptions are raised as the support for interrupts are somewhat lacking in the mlite core.

4.3.1.1 Known Limitations

As previously mentioned the mlite core lacks a few instructions from the MIPS-I ISA (unaligned load and store) and some very minor restrictions on the use of others (break and syscall) which should normally not be problematic if using the GCC tool-chain. These are further documented on the plasma homepage[37].

What is not documented as thoroughly is the way exceptions are handled. Currently interrupts are hard-coded in the sense that on the assertion of an interrupt request line the processor jumps to a hard-wired address, i.e. interrupt vector. The address fits neatly into the software solution provided with the Plasma MCU, but not necessarily so for other solutions, e.g. our run-time system. A solution would be to provide a substitution mechanism in our configuration system which at synthesis time would rewire the interrupt vector to the address of a user-provided function. This would on the other hand tie the intended run-time software and hardware tightly together, making it necessary to re-synthesize for changes to the software. A better solution would be to make it possible to specify the interrupt vector programmatically.

Furthermore, no hardware traps exists for exceptions regarding other coprocessors as currently all non-implemented coprocessor instructions are treated as NOPs. This means among other things that for floating-point programs to run correctly on the integer tile, they must be compiled with the soft-float option enabled. This means that we currently need two sets of the programs to be run in experiments, one set compiled with soft-float for the integer tile and one set with coprocessor 1 (i.e. FPU) for the accelerated tile. We recommend that hardware traps be implemented in the mlite instead and emulate an FPU in the interrupt system of the run-time system. This makes it easier to change tile types and layout without requiring the experiments' program binaries to change. This leads to a better separation of hardware issues from software issues.

To summarize, what we see is a coprocessor 0 that is only partly implemented in the current version of the mlite core and a fault-tolerance for coprocessor 1 that is lacking more or less completely. The improvement of the mlite's coprocessor support is considered to be a prime candidate for future work.

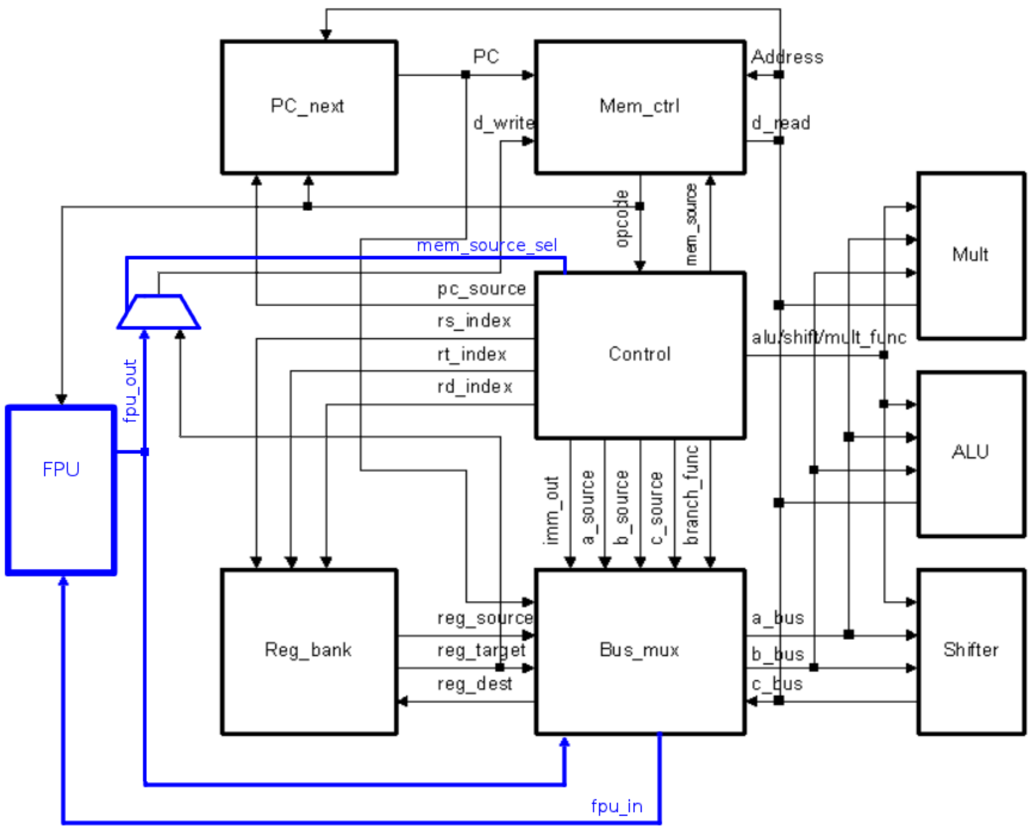


Figure 4.5: Block diagram for floating-point mlite. (Based on figure from [37])

4.4 Floating Point Processor Tile

For the floating-point accelerated tile we based the design on the integer core and added a third-party floating-point unit to it as is shown in Figure 4.5. Furthermore the integer core's control units were modified to support coprocessor 1-instructions and to stall the processor when waiting for FPU computation to complete. In conjunction with this stall logic a state machine were created to coordinating stalls according to the different coprocessor 1-instructions. These has not been added to the figure, similar to the original figure where all stall lanes were left out for clarity.

Seen from the figure is an FPU module and a multiplexer. The FPU module is further elaborated in Subsection 4.4.1. One of the main responsibilities for the mlite core regarding our FPU support is to present data and receive data from the FPU module. As seen from the figure, some new lanes was needed and modifications to some existing modules. Beyond this, processor register to co-processor 1 register operations was fairly straightforward to implement by controlling the pipeline in subtle ways. For memory-register operations on the other hand was a bit more tricky. The multiplexer has been added because a memory store from coprocessor 1 (SWC1 instruction) uses the FPU's register file and retrofitting this into the regular store operation required that all values needed for the store must be ready for the next cycle. To make this timing requirement the data could not reuse the regular pipeline and was shortcut by using a multiplexer and some glue logic. Memory loads to the coprocessor 1 register file on the other hand happens over multiple pipeline stages which makes it possible to reuse parts of the pipeline associated with the regular loads. This enables us to avoid adding a multiplexer for this case.

Only some of the floating-point functionality is currently implemented. This is the aforementioned load and store together with CPU-FPU move operations and the basic floating-point arithmetic for double precision. Other COP1 instructions (e.g. compares and square root) causes hardware exceptions. The interrupt request lines are currently left unconnected to the mlite core as an implementation and integration of a new co-processor 0 module to support this was considered unfeasible within our remaining time budget. Therefore special care must be taken when utilizing this version of the accelerated core.

4.4.1 FPU Module

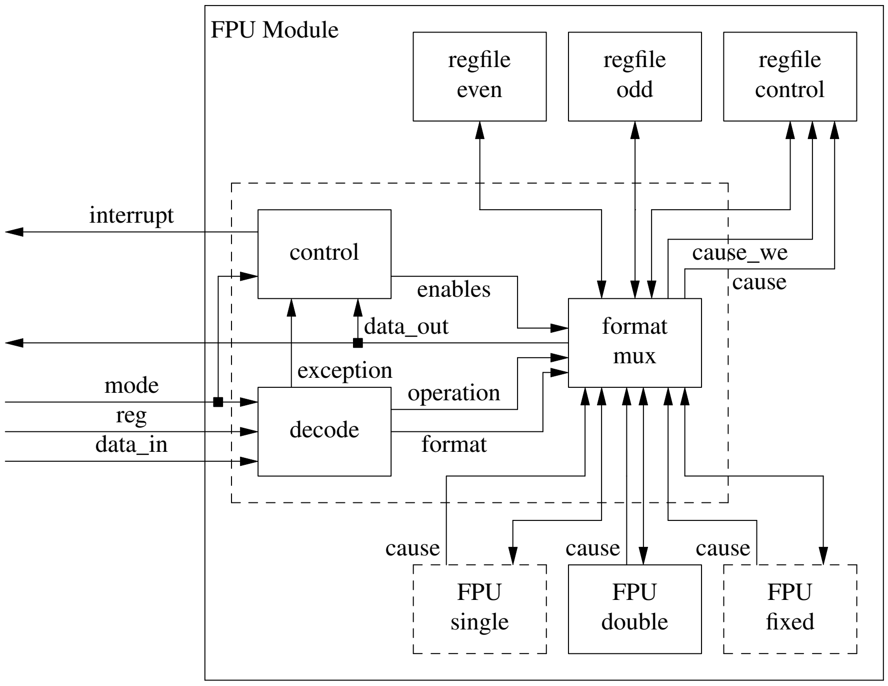


Figure 4.6: Block diagram of FPU module

Figure 4.6 shows a overview of the FPU module implemented for integration into the mlite for the creation of an accelerated core. The detail level of this figure is kept somewhat abstract for clarity. Some lines are not shown and some are combined. Most notable is the handshake mechanism between the FPU Module and the mlite core and the unlabeled arrows from the sub-module labeled "format mux". These arrows represent data and control lines in both direction. The dashed lines are modules not currently implemented, but which were considered and designed for. The sub-module "FPU double" provides our support for double-precision operations and is a third-party module created by Lundgren [38].

When the mlite core decodes a floating-point instruction it sorts it into one of different categories or modes as defined by us. The `mode` is one of the categories regular load, regular store, control load, control store or arithmetic. `mode` is input to the FPU Module as seen in the figure. If it is an arithmetic operation then the floating-point instruction is given as the input data `data_in` for further decoding at this level. If the mode is one of the other categories then the decoder unit in the mlite will present to the FPU Module which register `reg` to operate on, and as mentioned earlier will present data on `data_in` or expect data on `data_out` for loads and stores respectively.

The FPU Module's decoder interpret the `mode` and any arithmetic instruction present on `data_in` and decides if this is an operation that the module is capable of doing. If not then it raises an exception to the control module which in turn shall assert an interrupt. If no exception is raised then the decode module presents to the "format mux" what operation to perform and which registers/data to perform on. It also select which data format the operation is to use as shown by the line `format`. This works somewhat like a selector for a multiplexer, but the "format mux" is a multiplexer by name only.

The sub-module "format mux" started as a multiplexer early in the design, but expanded its functionality to encompass aspects of control regarding the different FPU format execute units. Also, the different ways to use the register files according to which format is in use for the current operation made it logical to place this part of the control within the multiplexer. The different FPU execution units are all envisaged to have the same interface and thus control and status to and from these units would be beneficial to multiplex. The register files indexing scheme is also different according to which `format` the operation is for, and instead of merging parts of the decoder and the control module to be able to handle this, the task of choosing registers from the multiple banks was left for the "format mux". As such the "format mux" is one part multiplexer and one part control/decoder. This somewhat overlap of responsibility is symbolized in the figure by the dashed box surrounding the control, decode and the "format mux" sub-module.

4.5 RAM Tile

The RAM tile is a pure slave tile containing a RAM slave unit. The intentions for this tile is to increase the available amount of accessible memory locations in a SHMAC layout. This is useful for layouts for data heavy applications and experiments requiring large data sets to produce interesting results. Also since this tile is one of the simplest functional tiles it is helpful in regression testing of overall tile changes.

4.6 LED Tile

The LED tile is used to set the LEDs of the development board and has mainly been used for debugging purposes. It is a pure slave tile where the slave unit is mapped to external pins which routes to a LED array on-board. It provided us with a simple feedback mechanism during development and debugging, but is now mostly superseded by the UART memory interface for such cases.

XXXX XXXX XXXX XXXX XXXX XXXX	LED
31 8	7 0

Figure 4.7: Complete memory layout of the LED Tile, with a total of 8 bit on-chip RAM

The unit holds just eight bits of data which is directly mapped to eight LEDs on the development board (ML605). The LEDs will show the eight lower bits of any data written to any memory location within this tile's range as shown in Figure 4.7. Reading from the tile will return this data. Bit 8 to 31 is regarded don't care, and will return zero on read. It can be noted that this tile takes no different action depending on the address requested, making all addresses within the tile's address space aliases to the same memory. As the slave unit is connected directly to the LED pins in the SHMAC's toplevel, only one such tile can exist.

4.7 Clock Tile

This tile is a pure slave tile which allows a program to get tick counts, which may be used for benchmarking programs. Using this tile can also make possible implementations of other timed events such as sleeps.

The function provided is a tick count, i.e. a counter which is incremented on each rising clock edge. This counter can be read and reset by memory operations. It also allows "silent counting" which is a mode where the counter value is frozen when viewed externally, but is still actively counting in the background. As this functionality is implemented as a tile, accuracy will be influenced by external factors such as network congestion. Given no network congestion the latency for starting and stopping the clock from a given tile will be the same. Thus benchmarking will be accurate if the same core initiate the clock prior to execution and freeze the clock at the end.

Memory layout for this tile is given in Table 4.3 gives the memory layout for this tile, where all control signals are set by writing to address zero. Table 4.3b gives details for the control register.

The tick count register is 64 bit, so an application would need two reads to retrieve the complete tick count. A correct reading would therefore require that the memory locations are not updated between those reads, as Tick Count$_0$ overflows every $2^{32}/(33 \times 10^6 \text{ Hz})$ sec $\approx$ 130 sec, assuming a 33 MHz clock. By setting the `update` field to 0, the memory is frozen and reads are safe.

An example of the writes needed to the control register in a benchmark is shown below. If the total execution time is wanted, there is no need in resetting the clock as this is done when the execution starts. In our benchmarks, the measured application simply disables the clock as its last instruction. The host polls the clock control register over UART until the clock is disabled and continues to read the result.

1. Set ctrl reg to `0b111`: Resets and starts counter. Memory is updated.

Table 4.3: Memory layout of the clock tile

(a) Complete memory layout

Address	Function	Description	Read/Write
0	Control Register		R/W
1	Tick Count$_1$	Most significant bits	R
2	Tick Count$_0$	Least significant bits	R

(b) Control Register for Clock Tile

Fields				
Name	Bit	Description	R/W	Reset State
Enable	0	Enables counting	R/W	1
Update	1	To update memory or not when counting	R/W	1
Reset	2	Resets count on every write of 1	W	0

2. Wait for timed event to finish
3. Write `0b001` to ctrl register: Memory is frozen, and can be safely read.

4.8 Jump Tile

This tile is a pure slave tile which returns MIPS-I instructions specially tailored for performing a jump to the start of the local memory of the requesting processor tile. This is necessary since the different processor tiles' local memories start at different global memory addresses and must in some way be directed to its associated program instructions. The Jump Tile contains a simple slave unit with no RAM and no master unit.

By utilizing this tile we avoid the need for either hard-wiring the start addresses for the different processor cores or requiring the cores to implement a common interface to support for an external method of setting the Program Counter (PC) at reset. As all the processing tiles implemented in the SHMAC starts execution at the global address 0, the Jump Tile is placed at coordinate `(0,0)` which is responsible for this segment. At reset when all processor tiles access address 0 each will receive replies tailored to it directing them to jump to the start of their associated local memory.

The memory layout for the tile is given in Table 4.4 on the facing page. The reply of an access to address 0 will be set using the requester's coordinate. The memory at address 1 and 2 are hardwired constant, and accesses to other addresses will give `nop` (all zeroes). The slave unit does not contain any RAM.

Instruction one and two will set register 12 to $coord(0)||coord(1)||000000000000000000000000$, where *coord* is the requester's coordinate assuming 4 bit coordinates. Instruction three gives a jump using the address in register 12.

Table 4.4: Memory layout of the jump tile

Address	Function	Description	R/W
0	`LUI`	Load jump address (upper)	R
1	`ORI`	Load jump address (lower)	R
2	`JR`	Jump instruction	R
3	`NOP`	NOP for branch delay slot	R

It would also be possible to implement this functionality by making the tile virtual, i.e. in a distributed fashion in each processor tile's router unit, or as done in our current implementation as a physical/actual tile. The virtual tile approach would favor start-up speed at the cost of increased hardware resource usage. As all processors will send requests to this tile at start-up it will give massive initial congestion and latencies in large SHMAC layouts when using the physical tile version. An implementation of a virtual Jump Tile will typically intercept all requests to tile `(0,0)` and reply directly to the processor instead of routing.

4.9 LL/SC Tile

The LL/SC Tile is used for implementing lock-free atomic Read-Modify-Write (RMW). Normally implementations of the LL and SC instructions utilizes caching mechanics, but as the SHMAC currently have no caching this tile is implemented to allow for a cache-less implementation of these instructions. This tile is a superset of the RAM tile containing additional registers for book-keeping whether a loaded word is still valid or not. This tile is a pure slave tile.

The bookkeeping of the atomicity of a LL/SC-pair is implemented as a table indexed by processor ID. Table 4.5 gives the layout of this validation table.

For each LL the valid bit for the processor will be set and the target address is saved. Data will be returned just as a normal load. Normal load operations function as before, while store operations will invalidate entries in the same manner as a successful SC.

A SC will be successful if the valid bit is set and the saved address equals the target address. A successful SC will return a packet with the `sync`-flag set, and allow data to be written to RAM. All entries in the valid-table with the same address will have their valid bit unset. An unsuccessful SC will return a packet without the `sync`-flag, and will not be allowed to modify the memory in accordance to the specifications given in [39].

The MIPS specification requires that an SC fails if an exception occurs on the processor execution the LL/SC. For supporting this the tile includes a option for explicit invalidating entries. If the LL/SC Tile receives a packet from a processor tile with the `interrupt` flag set, then it will invalidate this processor's entry.

Table 4.5: LL-tile's synchronization bookkeeping example

(a) Example state

Processor ID	Address	Valid
0	a	1
1	a	1
⋮	⋮	⋮
i	a_i	v_i
⋮	⋮	⋮

(b) After processor 1's SC to $a \neq a_i$

Processor ID	Address	Valid
0	a	0
1	a	0
⋮	⋮	⋮
i	a_i	v_i
⋮	⋮	⋮

(c) Processor 0's SC fails, then retries with a LL to a

Processor ID	Address	Valid
0	a	1
1	a	0
⋮	⋮	⋮
i	a_i	v_i
⋮	⋮	⋮

4.10 UART Tile

The UART Tile is a pure master tile that provides the functionality needed to control the SHMAC processor. The host system is typically the laboratory computer running experiments. The delivered version supports commands for reading and writing to memory and toggling resets. Table 4.6 lists the supported commands.

This hardware module is somewhat specific to the development board used since an on-board UART-controller chip is utilized. As only the basic UART functionality presented from the controller was used this module should be easy to port to other development boards.

Table 4.6: UART commands

Full Command Syntax	Command Byte Encoding	Function
`RESET_ON`	`11110000`	Resets interconnect and state machines throughout the design, reset and freezes the computational cores.
`RESET_OFF`	`00001111`	Unfreeze the computational cores.
`READ <addr>`	`00000000`	Returns over UART the content of `<addr>`
`WRITE <addr> <data>`	`11111111`	Writes `<data>` to `<addr>`

The UART Tile adds flexibility as run-time configuration of the system is possible. The assignment text require a memory interface for the developed system. Prior to adding the UART tile this consisted of writing initial memory at FPGA synthesis/implementation time and reading the final memory state through ChipScope, LEDs or using an external scope on output pins. Although possible it is a tedious and cumbersome form of running experiments. The addition of an UART memory interface is considered a good solution.

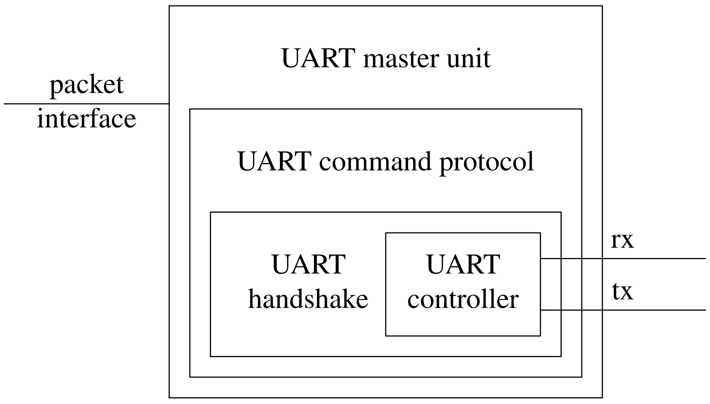


Figure 4.8: UART master unit design

The master unit of the UART Tile is shown in Figure 4.8. The digital design of the UART solution consists of the UART controller from the Plasma project, a corresponding facade/wrapper responsible for handshaking and a state machine utilizing this in our command protocol. In the version delivered, the UART is clocked at 1 MHz which with the current transfer protocol achieves an actual transfer rate of 1.1 kBps. The reason for such a relative slow actual transfer rate is the intensive handshaking during the protocol. The handshakes are added to avoid buffer overflows on the SHMAC side of the communication channel. The current UART controller only has a two byte buffer and given the possibility for delays/congestion in the Network-on-Chip (NoC) some type of throttling for overflow fault tolerance was needed. This results in handshakes for every byte transferred. The speed achieved was more than sufficient for our needs during development. Therefore we decided to not modify the third-party modules to avoid incurring more validation and verification time costs than strictly needed. If future needs dictate it the UART speed should be fairly easy to increase by enlarging the controller's buffer size to $n + 1$ and handshake for every n-th byte instead.

One possibility would have been to use a protocol with a checksum and then resend when this were incorrect. We went with a simpler approach where we assume the host has the necessary buffering resources required and that only host-initiated communication needs to be stalled. This is accomplished by a contract between host and SHMAC that makes both writes and reads into blocking operations. For reads this is trivial, just wait for the answer before proceeding in the host's program execution. For writes, the UART tile will echo back the written data after having processed it. If the host waits until hearing its echo then no buffer overflows can occur. Still, the possibility of some line errors would make the checksum method more robust, but even then the possibility of errors is present, for instance by multiple line errors in the same UART packet.

The UART has been tested thoroughly and a correct functionality has been empirically verified by overnight (12-hour or more) stress tests. These tests consists of continuous tests running at the maximal sustainable rate over UART and tests with varying time between consecutive communications for testing different usage patterns.

5 SHMAC Software

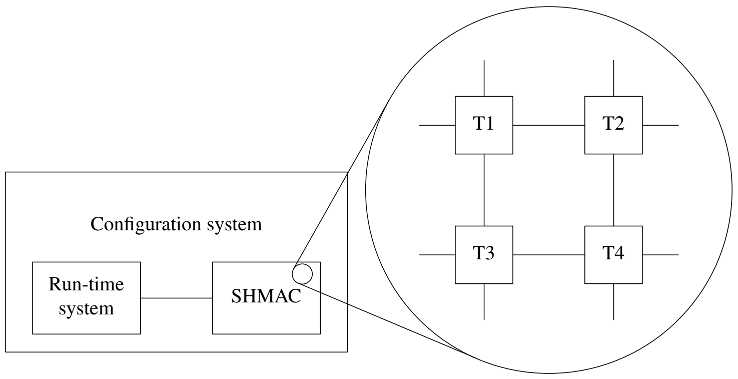
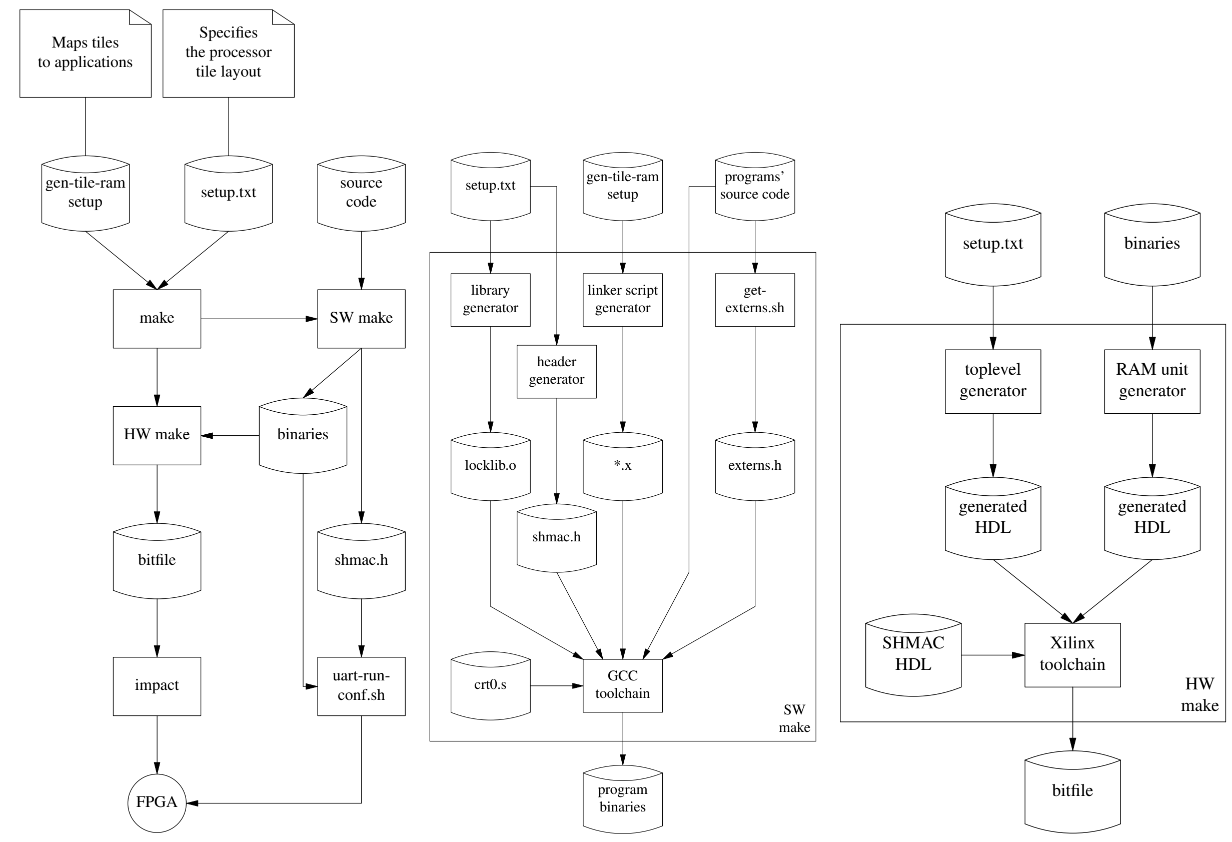


Figure 5.1: Overview of the SHMAC framework. Both the hardware and software are configured by the configuration system.

Figure 5.1 shows the created framework. It contains two software parts: The configuration system and the run-time system. The configuration system sets up the hardware and software build systems with the correct layout of tiles according to specifications from a configuration file. It also configures different run-time system components to work with this architecture layout. As software written for the SHMAC is compiled according to a specific configuration, a change in hardware layout may require that software be rebuilt.

In addition to the framework components, host programs have been written which are used to setup and administer the SHMAC to run different experiments on the currently implemented hardware configuration. These tools load new software in an easier fashion than if required to go through the complete configuration system build process each time.



(a) Overview (b) Software build steps (c) Hardware build steps

Figure 5.2: Configuration system build steps

5.1 Configuration System

The SHMAC Configuration System is a collection of user space tools which chained together implement a processor with a specified layout onto an FPGA and tailors our software framework to work in this layout. Figure 5.2 shows the steps taken by the system to move from specification to FPGA implementation.

```
JPPPP
PFPFP
PPPPP
```

Figure 5.3: Example processor layout configuration file

Table 5.1: Tile types. Details given in Chapter 4.

Glyph	Reference	Name	Limitations
`P`	`PLASMA`	Integer Processor Tile	
`F`	`FPU`	Floating Point Processor Tile	
`R`	`RAM`	RAM Tile	
`S`	`LED`	LED Tile	Maximum one
`C`	`CLOCK`	Clock Tile	
`J`	`JUMP`	Jump Tile	Should be placed in `(0,0)`
`L`	`LLSC`	LL/SC Tile	
`U`	`UART`	UART Tile	Maximum one

The processor layout is input to the configuration system in form of a text file schematic. The intended layout is drawn with different ASCII characters representing the different types of tiles available as given in Table 5.1. For instance the regular processor core is denoted P in the schematic and the floating-point accelerated processor core is denoted F. An example of such an schematic is shown in Figure 5.3 on the preceding page.

Based on the layout specification in `setup.txt` the configuration system generates a toplevel HDL source file with the tiles interconnected in a mesh topology. It also creates a header file describing the layout by the use of a `TILE` macro which may be used programmatically or as part of configuration files. The toplevel is then processed by the regular Xilinx toolchain creating an FPGA bit-file ready for FPGA implementation using the Xilinx flash writer program `impact`.

Figure 5.2b shows the steps taken for software to be built correctly for a given hardware layout. A mapping of applications to tile slots are input to the build system as a setup directory `gen-tile-rams`. In this directory, symbolic links of the form i_j points should point to the ELF binary destined for the local memory of the tile at position (i, j). An example of this is given in Appendix E. Furthermore, if the application needs atomic locking, the configuration system can create a lock library, `locklib.o`, which presents applications with a convenient C API for atomic locking. A more fundamental responsibility of the directory setup is to instruct the linker script generator to create linker scripts for the correct mapping of the different binaries to the correct local memories of tiles. Without correct linker scripts the execution is bound to crash as instructions using absolute memory addressing mode will be erroneous. In case any program is to be in memory immediately after the FPGA initialization then this `gen-tile-ram` setup must be provided at the hardware build time. If provided then a RAM unit generator makes sure to initialize the block RAM within the tiles with program binaries associated with it.

Makefiles are provided to aid in the different parts of this configuration process. Using the dependency tracking mechanisms of `make` allows changes made to the different hardware or software steps of incurring the least amount of rebuilding necessary. In addition to a Makefile for building the complete system, Makefiles for building software for the SHMAC, i.e. setup to use the correct cross-compilation tool-chain, is also provided. All Makefiles are considered part of the configuration system.

Not shown in the figure is the generation of a user constraints file. Different tiles requires different user constraints which means that dependent on which tiles are included in the SHMAC layout the `ucf`-file must match accordingly. Using the included Makefile takes care of these details.

5.2 Run-time System

The SHMAC framework provides support for a C run-time environment and provides an atomic lock mechanism. This goes a long way to support TBP run-time engines, but unfortunately time did not permit the implementation of such a run-time scheduling solution. Some deliberations about TBP run-time systems for our architecture is given under Future Work.

The supported MIPS ABI is O32 and gives developers access to the C programming model as opposed to writing in assembly. To support the C run-time environment, an assembly snippet, `crt0.s`, sets up the required hardware state at boot while the linker scripts makes sure symbols referring to stack and heap is available for use. The stack is heavily used to support function calls. Currently the heap support of the MIPS Application Binary Interface (ABI) is left for the users of the framework to implement. Figure 5.2b shows where in the build process the provided C run-time setup is used.

To run a simple multi-programmed workload on the SHMAC requires only to configure the memory space in such a way that each tile run their own program and does not access other processor tiles' memory and only read from other type of tiles. Multi-programmed workloads are on the other hand not sufficient for the implementation of TBP.

To support proper co-operative parallel run-time systems, the SHMAC framework provides a lock library and header files which may be used to create more interesting run-time system. This may be a simple map-reduce system or it may be a TBP. For the remainder of the thesis the approach taken by us was to explicitly code synchronization statements where needed. This was a viable possibility for our small benchmarks, but for larger software systems a proper parallelization engine is much preferred.

5.3 Writing SHMAC Software

5.3.1 Accessing External Symbols

Each tile is compiled with a binary each and the binaries will not have access to each others symbols out of the box. To allow the SHMAC to support programming multiple threads of execution with shared memory a binary can access shared symbols by including an header file containing this information.

The binary of the core holding the shared symbol must be compiled before those which depend on it. The symbol is retrieved using a script which access this symbol using `nm`

Listing 2 Listing dependencies between binaries in Makefile

```
<uses-shared-variable>.elf: externs.h

externs.h: <declares-shared-variable>.elf
```

Listing 3 Make variable accessible for other binaries

```
#pragma shmac <shared>
volatile unsigned long <shared> = 1
```

and builds a header file, `externs.h`, containing this information. To ensure that binaries are compiled correctly, circular dependencies between binaries should be avoided, and the Makefile should include dependencies as shown in Listing 2.

To make a symbol accessible for other binaries, the variable must be global and be listed using `#pragma shmac <name>` as shown in the example given by Listing 3. Shared variables will be renamed to `ext_<name>` to avoid problems with redeclaration of names. The prefix `ext_` should therefore not be used for any user variables. An example showing this is given in Listing 4 on the next page.

5.3.2 Access Memory from other Tiles

The start of a tile's memory address is defined in the header `shmac.h`, so it can be used at compile-time. This header also gives how many of each tile type the configured hardware contains. This is used for instance in the provided locking framework, as this automatically uses the first available LL/SC Tile. Each tile is numbered using left-right, top-bottom, such that `TILE(<type>,1)` is placed either down or right of `TILE(<type>,0)`. The header-file can also be used to retrieve an array containing all tiles of a specific type which may be used to set this address at runtime. Listing 5 shows a example of this syntax.

Listing 5 Retrieving Memory Addresses using shmac.h

```
#include <shmac.h>

// Will not compile without a Clock Tile in layout
unsigned int * start_address_of_clock_tile = (void*) TILE(CLOCK, 0);

// Will compile without a Clock Tile
unsigned int clock_tiles = TILE(CLOCK, ARRAY); // Array of start addresses
if( TILE(CLOCK, NUM) )
  unsigned int * start_address_of_clock_tile = clock_tiles[0];
else
  unsigned int * start_address_of_clock_tile = NULL;
```

Listing 4 Accessing a shared variable

```
#include <externs.h>

volatile unsigned long * <shared> = (volatile unsigned long*) ext_<shared>;
```

Listing 6 Using the locking framework

```
#include <locklib.h>

#pragma shmac wait
volatile int wait = 1;

#pragma shmac lock_address
unsigned int * lock_address;

int main()
{
  // Initializes the locking library
  init_ll();

  // Allocate lock
  int lock_error;
  alloc_lock(&lock_address, &lock_error);

  // Allows other binaries to proceed
  wait = 0;

  lock(lock_address);
  (...)
  unlock(lock_address);
}
```

5.3.3 Locking Library

The locking library is set to use LL/SC Tile zero, that is `TILE(LOCK,0)`. The address to this tile is set statically, so it must be compiled specifically for the given configuration. It does not currently support using more than one LL/SC Tile.

Listing 6 gives an example of usage of the locking library. The locking library needs to be initialized before use. This must only be once. Locking addresses is not set until after `alloc_lock`, so it is important that other binaries that is to use the lock does not proceed until after this point. This is typically assured by letting these binaries run a loop checking if the value of the shared variable `wait` is zero.

By setting `wait = 1` we assure that the variable is not placed in `.BSS`, which would allowed it to be uninitialized at program start. In worst case this may allow other binaries to run `lock()` before initialization.

In a shared memory multiprocessor environment the requirement for synchronization is paramount for enabling any possible cooperation and work sharing. There exists pure software implementations such as Peterson's algorithm [40] for mutual exclusion. They are however assumed to be inferior to hardware solutions utilizing atomic operations. We have therefore implemented a subset of the MIPS-II ISA to support the LL and SC instructions.

The locking library provides functions for handling all synchronization. Applications should still be compiled for MIPS-I as the full MIPS-II ISA is not supported, and invoke LL and SC through this library.

To provide atomic RMW among multiple processors, all accesses to the LL/SC tiles must be done with a cache coherent system [39]. This will also be true in the implemented cache free system. With the use of a x-y routing, no accesses will be reordered, which can guarantee that exceptions in the processors will not create unpredictable behavior.

5.4 Host Software

5.4.1 UART Utility Programs

On the host a suite of programs running on Linux was developed. These uses the typical tty/terminal library to communicate over UART. These programs where written in accordance with the handshake protocol contract described given in Section 4.10. The program suite consists of the UART stress test programs, the program loader with all its tool-chain components, and a program to dump memory content. With the program loader and the memory dump program it is possible to create a system for conducting experiments.

5.4.1.1 Program Loader

`uart-run-conf.sh` is a script that takes an experiment configuration which maps binaries to tiles. An example of such a configuration is given in Listing 7. The utility writes the binaries to the SHMAC as directed and sends a reset signal. It is invoked using `uart-run-conf.sh <configuration file>`

Listing 7 Example of UART configuration file

```
    TILE(PLASMA, 0) <binary>
    TILE(PLASMA, 1) <binary>
```

The steps taken by the utility is given in Table 5.2.

A remark about `uart-load-with-readback-bin` is in order. This tool-chain component writes to memory, but since writes are non-blocking this behavior alone is not enough to ensure a correct behavior. This because the following release of reset might happen before all in-flight packets truely have been written to memories. Therefore this component verifies that data has been written by issuing a read to the same address

Table 5.2: `uart-run-conf.sh` tool-chain steps

Step	Tool Component	Description
1	`uart-reset-on`	Holds reset on the SHMAC
2	`uart-load-with-readback-bin`	For each configuration line the utility writes a binary to the s
3	`uart-reset-off`	Releases reset

range which are written. This is possible based on the fact that we use the deterministic XY-routing which makes sure that no reordering occurs in the mesh, making the readback guaranteed to return after the write has finished.

5.4.1.2 Read Memory over UART

This program is used for retrieving the memory content of a specified address range in the SHMAC. It takes a byte addressed memory range as input. The output is written to standard out.

```
uart-tools/uart-memdump <from> <to>
```

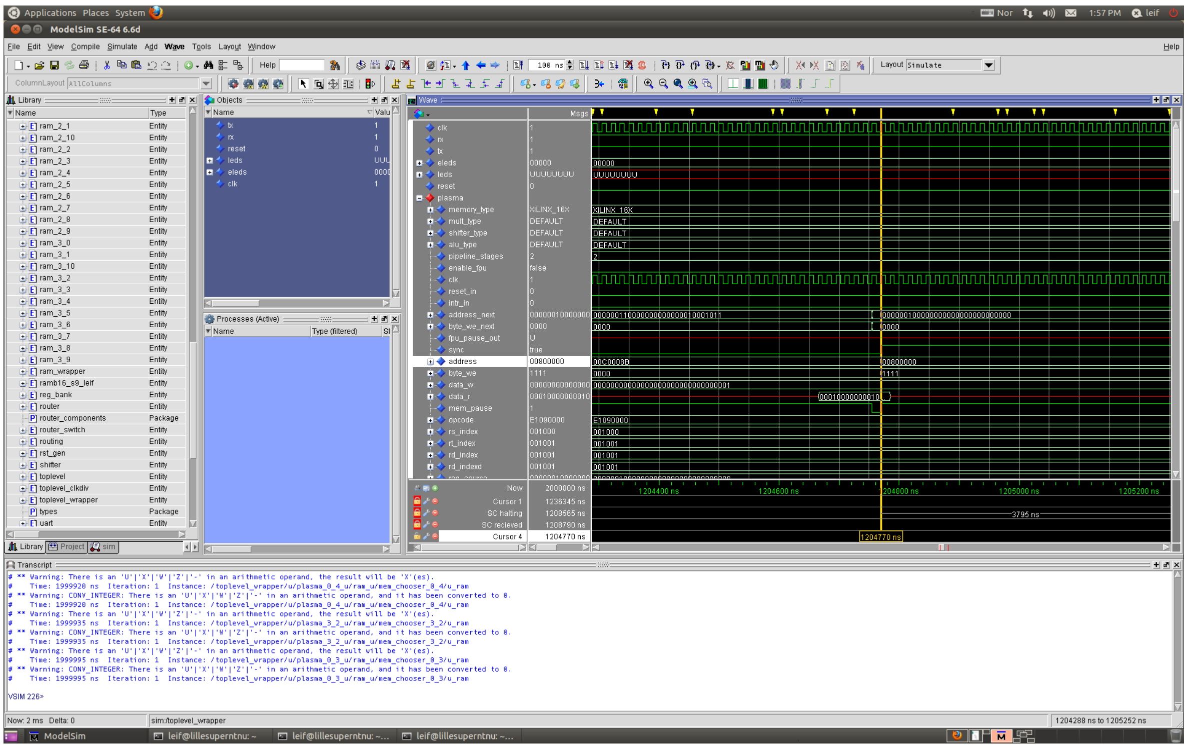
6 Verification and Reproducibility

The verification phase of our thesis coincided with the implementation phase, realized as test-driven development by unit testing. Also larger integration test-benches were created, but a planed full system test-suite were not completed. This system test was to be used during optimizations (speed or logic size) of modules and would allow for rapid feedback on introduced breakage. Due to time constraints, simulations to support the verification of the full hardware system were given a lower priority than verifying the correct functionality of the complete system. In other words, the hardware is considered correct as long as the software running on it is observed to function correctly.

This assumption does however tie the hardware tightly to software troubleshooting since no full system test can be used to eliminate the hardware from the software error observed. As such, faulty behavior which is not immediately evident to origin from software may or may not originate from hardware. It may be that a previously undiscovered hardware bug is accidentally exercised by a new piece software. This uncertainty regarding the source of errors may cause time to be wasted on efforts to track down bugs in the wrong part of the design. During development, the debug mantra of "always assume the problem to be software related first" seemed accurate, but more complete test suits are still encouraged for future work since the uncertainty will be reduced.

Regarding Software Verification the approach taken is a pragmatic one. Software is executed and its correct behavior asserted from a correct end-result, usually determined by examining the memory state of selected processor tiles. Any faults discovered prompts a manual insertion of debug statements in the code with the purpose of creating debug memory state to examine the faulty behavior closer. This makes it possible to examine the memory and execution state at specific points in the program by setting a variable to unique values all according to which part of the program is currently executing, or make use of control statements which simulate a break points in the execution until poking the control statement's associated test value over UART to continue execution. This rudimentary approach have treated us well for the verification of our software, but for larger software systems it would be considered cumbersome and we recommend adding support for GNU Debugger (GDB) or another sophisticated debugging system before implementing the TBP run-time system.

6.1 Hardware Verification


```
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 1999920 ns  Iteration: 1  Instance: /toplevel_wrapper/u/plasma_0_4_u/ram_u/mem_chooser_0_4/u_ram
# ** Warning: CONV_INTEGER: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, and it has been converted to 0.
#    Time: 1999920 ns  Iteration: 1  Instance: /toplevel_wrapper/u/plasma_0_4_u/ram_u/mem_chooser_0_4/u_ram
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 1999935 ns  Iteration: 1  Instance: /toplevel_wrapper/u/plasma_3_2_u/ram_u/mem_chooser_3_2/u_ram
# ** Warning: CONV_INTEGER: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, and it has been converted to 0.
#    Time: 1999935 ns  Iteration: 1  Instance: /toplevel_wrapper/u/plasma_3_2_u/ram_u/mem_chooser_3_2/u_ram
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 1999995 ns  Iteration: 1  Instance: /toplevel_wrapper/u/plasma_0_3_u/ram_u/mem_chooser_0_3/u_ram
# ** Warning: CONV_INTEGER: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, and it has been converted to 0.
#    Time: 1999995 ns  Iteration: 1  Instance: /toplevel_wrapper/u/plasma_0_3_u/ram_u/mem_chooser_0_3/u_ram

VSIM 226>
```

Figure 6.1: Modelsim window

For this thesis the main test strategy for hardware consisted of writing VHDL test-benches capable of running in a simulator. Typically use cases for the different modules and use cases for collections of modules were tested. This was implemented by writing non-synthesizable HDL containing sequences of test vectors with associated precomputed result vectors. The digital design is then driven by this sequence and its output compared to the precomputed result vectors.

The simulators available to us were the integrated simulator ISim as part of Xilinx' development tools and the standalone simulator Modelsim by Mentor Graphics. Since we have prior experience with Modelsim this was chosen to be our main test tool. The version utilized was Modelsim SE-64 6.6d. Using Modelsim both functional and timing simulations could be performed on the test-benches.

In addition to writing test-benches to be used for test-driven development, i.e. tests written based on intended functionality and specifications, another debugging strategy was used when bugs were found or tests were lacking. Using Modelsim's possibility to drive lines (including clock generation) from command line, a method focusing on rapidly examining the waveforms in a similar setup, i.e. hardware layout and software, was developed. Using functional simulation the design was simulated until the fault observed in the field was reproduced in waveform. An iterative approach of backtracking the wave and forcing signals in conjunction with reasoning about the signals behavior helped in tracking down which module part was faulty and suggested what cases was erroneous. After such a session, portions of the Modelsim command log could also serve

Table 6.1: System-level Software Tests

Experiment	Thoroughly exercises	Notable remarks
Tutorial	Processor, RAM and LL/SC Tile, interconnect	
μbenchmarks	Processor Tile	
Parallel sum	Processor Tile	Also verifies that the lock mecha

as the prototype of new test-benches. Figure 6.1 shows a typical debug session with a functional simulation showing our design's waveforms.

For board-under-test verification, a setup consisting of a computer, with remote access over the network, was connected to the JTAG interface of the ML605 board. We could then flash the FPGA remotely from our work places. When combined with X tunneling the possibility of a remote scope became apparently beneficial. Utilizing Xilinx' ChipScope Pro Core Inserter 12.4 and ChipScope Pro Analyzer 12.4 M.81d the interior signals and state of the true realization on the FPGA could be examined. This allows for a very thorough verification of the final product.

To examine the contents of FPGA memory primitives which were directly instantiated in third-party modules, Xilinx' unisim/simprim VHDL source codes for these primitives were modified to become synthesizable. Using these versions the memory state signals could explicitly be kept after synthesis by using constraints during the FPGA implementation process. If done in this way then verification of the memory content becomes considerably less cumbersome throughout all the verification steps, from functional through timing simulation to even board-under-test since bus signals then became available for ChipScope to use. An unsatisfactory consequence of this approach is that such modifications may introduce inconsistencies between the simulated and the actual since this method require substituting the simulation primitives for modified ones. For instance in the board-under-test verification an inconsistency is introduced when using the modified simulation primitives. Block RAM primitives are changed to distributed RAM, i.e. flip-flops. Although this should be functionally equivalent, it constitutes a difference between the verified system and the actual.

An example of a positive effect of development on verification is how the development of the UART memory interface simplifies system verification and rectify the inconsistency introduced in the approach outlined above. Also with this new functionality, when doing system-level verification new experiments could be continuously programmed onto the currently implemented SHMAC layout and a correct result verified from a memory dump over UART. By using this approach a serious bug in the LL/SC Tile missed in previous unit and integration tests was found and corrected. Automating such an approach would allow for continuous randomized test vector explorations, or other test methodologies exploring coverage and exercising subsets of all possible input combinations in an automatic test system.

6.2 System Verification

The complete SHMAC system is considered verified as it has been running comprehensive experiments which stress the different portions of our design. The ones which are included in our final delivery is listed in Table 6.1. They are briefly examined as follows.

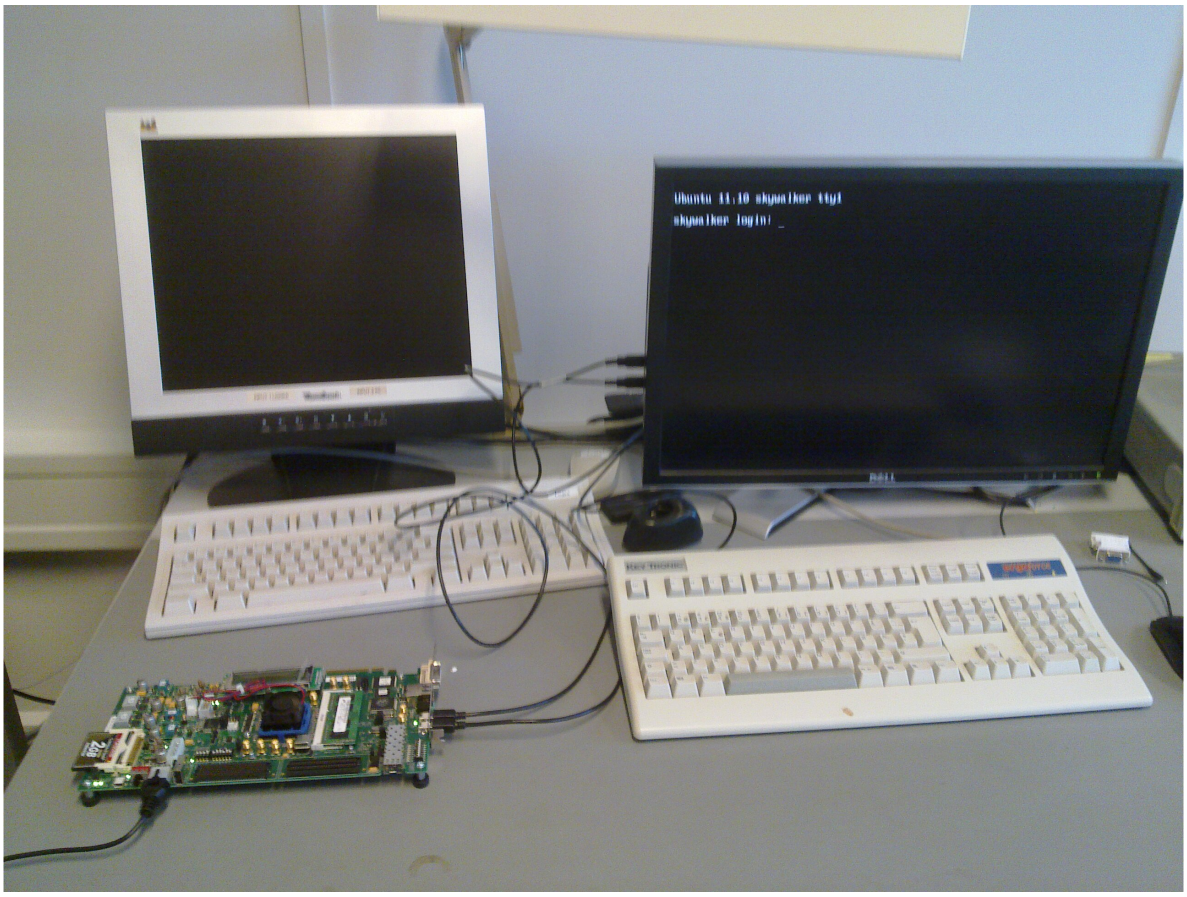
The tutorial (Chapter E) gives an implementation of the textbook showcase for atomic locking, i.e. the bank account example. This program's correct behavior is a good indicator for the total system's correctness. To maximally stress the components a modified version with maxed out number of processor tiles was utilized. This stresses the interconnect so that congestion arises since the number of processor tiles relative to the number of LL/SC Tiles becomes large and multiple processor cores is actively trying to gain the lock ownership of the single lock. Also since each tile's program's account is stored in the same RAM Tile this tests the ability of the interconnect and RAM Tile to handle diversity and quantity of packets and memory accesses.

The parallel sum experiment mainly exercise the processor tiles, but as the program's correctness is dependent on the functionality of the LL/SC Tile this tile is also implicitly verified. As the time measurements made from the different experiment runs looks as expected, the correctness of the Clock Tile is given.

6.3 Our Setup

In this thesis we have made use of a development kit from Xilinx, namely the Virtex-6 FPGA ML605 Evaluation Kit. This board contains a fairly large FPGA chip, enabling us to implement processors with a large number of cores. In addition it contains I/O such as UART over USB aiding in the interfacing of the final design and the surrounding environment. The development tools used from synthesis to implementation is Xilinx' ISE software version 12.4 M.81d. Figure 6.2 shows the laboratory setup used during the thesis. As shown the development board is connected to a host system which besides controlling and implementing our design onto the FPGA also provided us with remote access to our development system over the network.

For the software part, since the delivery do not contain any operating system, the C compiler suite and tool-chain needs to be configured for this case. This can be accomplished by utilizing a C standard library implementation which do not use operating system syscalls. A handy shell script for setting up such a bare metal tool-chain which utilize the C standard library implementation called newlib was found on the OpenCores homepage for the Plasma project. After some minor modifications of this script the tool-chain builds automatically. The most important tools built from this script for our implementation is binutils 2.21.1, GCC 4.5.2 and newlib 1.19.0.



```
Ubuntu 11.10 skywalker tty1
skywalker login: _
```

Figure 6.2: Development board setup

7 Results

7.1 Micro Benchmarks

Figure 7.1 gives Cycles Per Instruction (CPI) for three microbenchmarks, *Local Memory Read*, *Neighboring Tile Read* and *Neighboring Tile Write*. As the figure shows, all these benchmarks includes reading instructions from local memory. Implementation details for the benchmarks is given in Appendix B.1 on page 87.

The *Local Memory Read* benchmarks 2000 register-to-register instructions. This is done in 58038 cycles, giving $\frac{58038 \text{ cycles}}{2000 \text{ instr}} \approx 29CPI$.

Reading 2000 addresses from a neighboring tile takes 156038 cycles, as measured by the *Neighboring Tile Read* microbenchmark. This benchmark includes equally many instruction reads from local memory. The processor can have maximum one outstanding memory read request, such that the external read request takes a total of 98000 cycles.

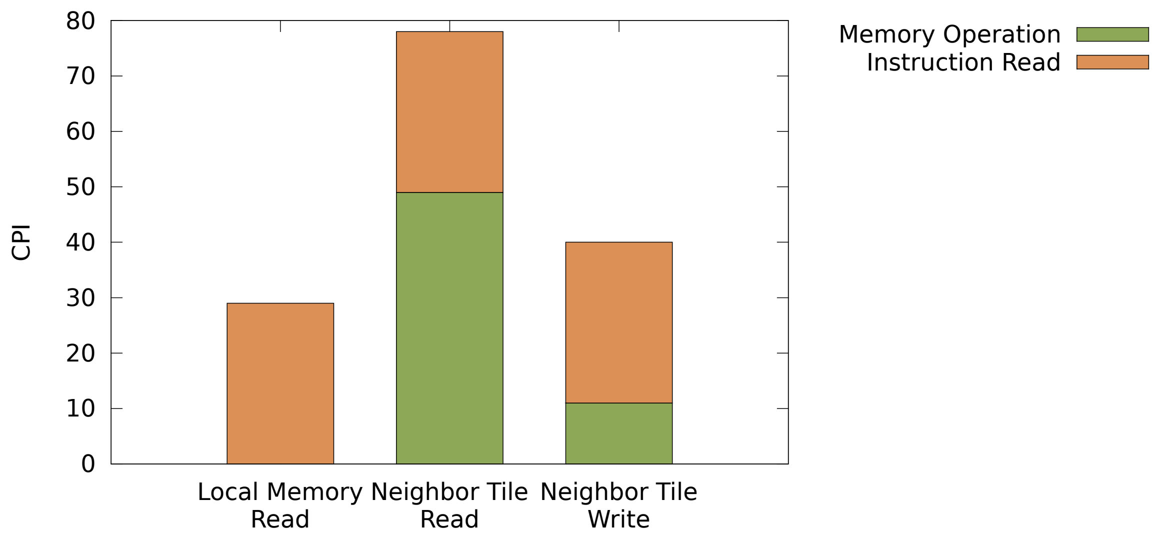


Figure 7.1: CPI for the implemented microbenchmark, showing both cycles used for reading instruction from local memory and the memory access to neighboring tile

Read performance to neighboring tile is then approximately 49 CPI in addition to cycles used for fetching program instruction.

Neighboring Tile Write issues 2000 memory write requests to a neighboring tile using 80038 cycles. That is 22000 cycles in addition to instruction fetches. This gives a performance $\frac{22000\text{ cycles}}{2000\text{ instr}} \approx 11CPI$ in addition to instruction fetches. As the router have received a write request, the processor is allowed to continue and may fetch the next program instruction.

7.2 Parallel Sum

This benchmark runs a summation using a loop with 360000 iterations using 1949596 cycles with 36 Integer Processor Tiles. The benchmarks source code is given in Appendix B.2 on page 89

All of the benchmarks are done using the same bit file, with the master program executing on the same tile in all runs. The tile closest to this tile executes the slave program, while redundant tiles is set to halt.

Table 7.1 gives execution time in cycles for this application given number of workers and problem size. Figure 7.2 shows the speedup as the numbers of worker increases for the different problem sizes. It can be observed that for very small problem sizes, such that 288 iterations, the overhead of parallelization makes the sequential version faster. Using four workers does however give a lower execution time than two workers.

As expected, it can be observed that if the problem size is not sufficient large, adding more workers may decrease overall performance.

7.3 Time Required for building Prototypes

The processor tiles implemented in the SHMAC can easily be altered to allow use of different processor cores. The only requirement is that they implement the interface

Table 7.1: Execution time in cycles given number of workers and problem size

	Problem Size					
Workers	288	2 304	12 960	51 840	180 000	360 000
36	210 988	220 732	272 236	460 156	1 079 596	1 949 596
32	201 392	212 354	270 296	481 706	1 178 576	2 157 326
16	152 564	174 488	290 372	713 192	2 106 932	4 064 432
8	100 219	144 067	375 835	1 221 475	4 008 955	7 923 955
4	72 912	160 608	624 144	2 315 424	7 890 384	15 720 384
2	75 097	250 489	1 177 561	4 560 121	15 710 041	31 370 041
1	50 440	401 224	2 255 368	9 020 488	31 320 328	62 640 328

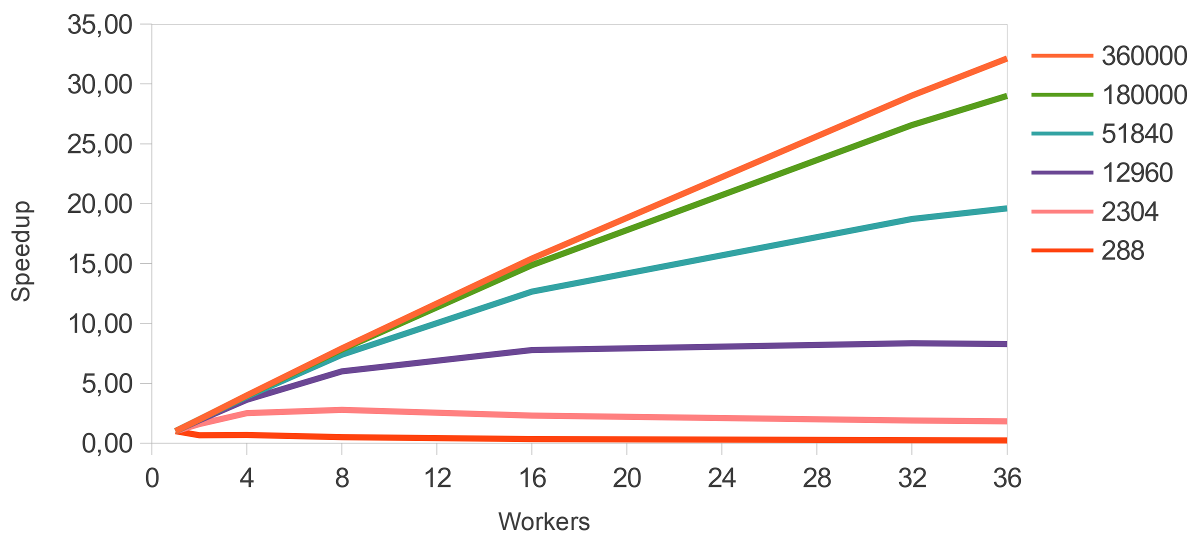


Figure 7.2: Speedup given as a function of workers for different problem sizes

required to operate against the router. This is typically done by adding a simple wrapper or adapter between the processor core and the router, with a simple state machine to convert the processors memory operations to network packets.

This modular approach is helpful in reducing the time investment required to develop different heterogeneous processors, as many of the modules can be partly or completely reused.

7.3.1 Synthesis Time

The bit files have been synthesized using an eight core 2.50 GHz Intel(R) Xeon(R) processor with 8 GB RAM using Xilinx ISE 12.4 with multi-threading allowing usage of maximum four cores are activated for `map` and `par`.

Synthesis time for different dimensions is shown in Table 7.2 for a setup with Integer Processor Tiles and in Table 7.3 for Floating Point Processor Tiles. Each of this setups consists of one Jump Tile and one UART Tile with the rest as Integer Processor Tiles. The Jump Tile and UART Tile is added to create a typical fully programmable example.

A synthesizing of 42 Integer Processor Tiles uses approximately 2 hours and 45 minutes in total. It can be observed that synthesis time is increasing as we reach the resource limit for the FPGA as `map` (placer and map) must do heavier optimizations. For balanced dimensions, that is keeping the length of the axis relatively equal, this is the largest design we have been able to synthesize. The synthesis time is shown in Figure 7.3. Larger design have been tested, but will not fit the used FPGA.

The Floating Point Processor Tiles requires significant more logic than their integer counterparts. This is both due to that floating-point operations in general requires more logic, but also due to that our FPU implementation introduces a significant longer critical path which complicates mapping. Our largest synthesized Floating Point Pro-

Table 7.2: Synthesis Time for Integer Processor Tiles in second

								Total	
Dimensions	Tiles	Cores	xst	bld	placer	map	par	In seconds	In hours
2x2	4	2	140	15	166	178	165	664	≈ 0.2 h
2x4	8	6	257	31	469	496	265	1518	≈ 0.4 h
4x4	16	14	514	64	615	674	462	2329	≈ 0.6 h
4x8	32	30	1080	174	1275	1416	863	4808	≈ 1.3 h
4x9	36	34	1354	149	1517	1686	978	5684	≈ 1.6 h
5x8	40	38	1549	166	1748	1952	1160	6575	≈ 1.6 h
4x10	40	38	1550	165	1787	1989	1088	6579	≈ 1.8 h
6x7	42	40	1658	175	1853	2069	1300	7055	≈ 2.0 h

Table 7.3: Synthesis Time for Floating Point Processor Tiles in second

								Total	
Dimensions	Tiles	Cores	xst	bld	placer	map	par	In seconds	In hours
3x3	9	7	845	113	1178	1263	603	4002	≈ 1.1 h
4x4	16	14	1633	222	3724	3926	1115	10620	≈ 2.9 h
3x6	18	16	1945	257	4600	4842	1313	12957	≈ 3.6 h

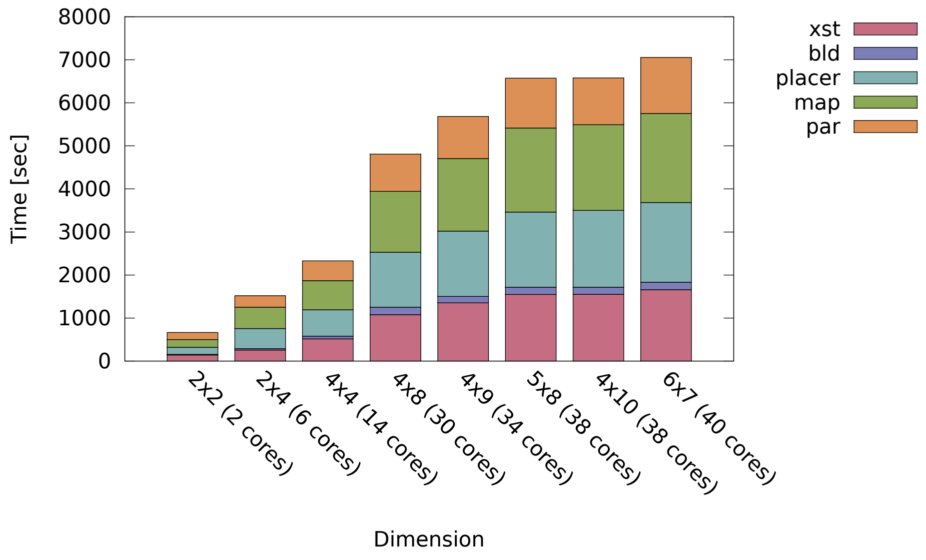
cessor Tile design is a 3x6 mesh containing 16 processor cores. Designs containing 18 cores is tested, but does not fit in the FPGA. Synthesis time is shown in Figure 7.4.

Table 7.4: Synthesis Time for mixed Processor Tiles in second

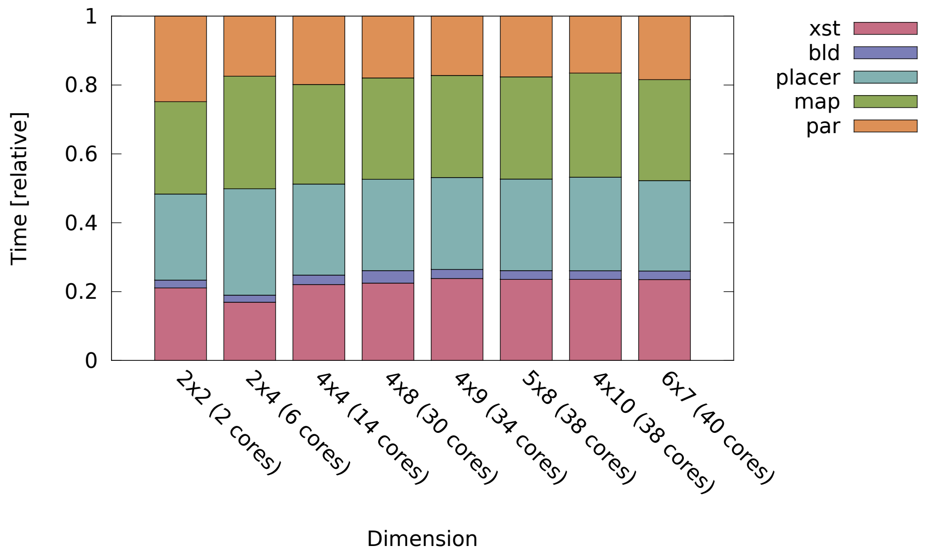
								Total		
Dimensions	Tiles	Int	FP	xst	bld	placer	map	par	In secs	In hours
4x5	20	12	6	1436	147	1934	2071	851	6439	≈ 1.8 h
5x5	25	18	6	1724	168	3832	4002	944	10670	≈ 3.0 h
5x6	30	20	8	2175	213	5200	5433	1423	14444	≈ 4.0 h

Table 7.4 gives synthesis time for a configuration including both Integer Processor Tiles and Floating Point Processor Tiles, respectively labeled Integer and Floating-point. The distribution of integer and floating-point processors is set arbitrarily to show a combination of those.

Synthesis reports are given in Appendix C with place-and-route report for a 6x7 mesh with 40 Integer Processor Tiles given in Listing 16. This synthesis uses 99 % of all available slices, where 1 % of those have an unused Lookup Table (LUT). Synthesis of a configuration with 20 Integer Processor Tiles and 8 Floating Point Processor Tiles gives the same result as shown in Listing 18. The largest FPU design synthesized, that is 16 cores, occupies 97 % of all slices, from which 1 % of these have unused LUT as shown in Listing 17.

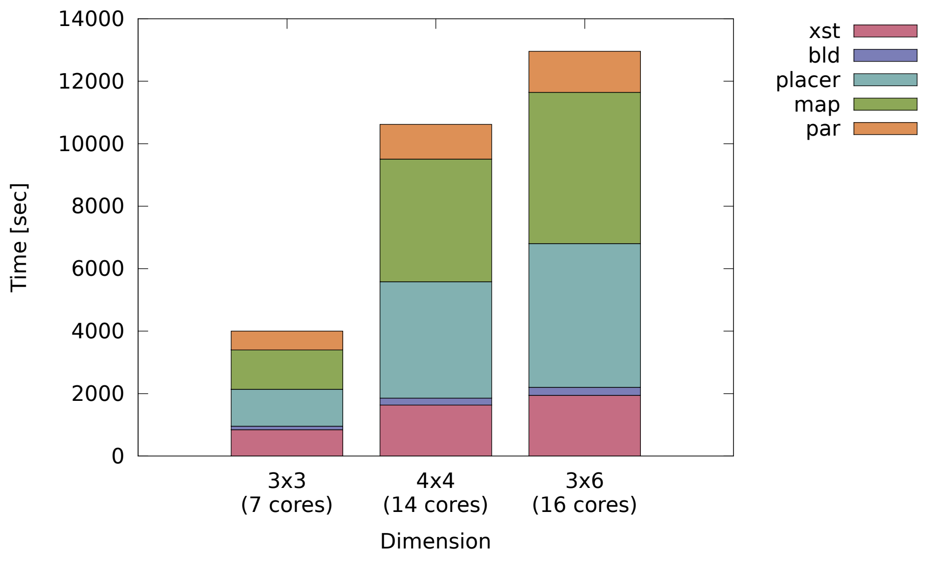


(a) Absolute

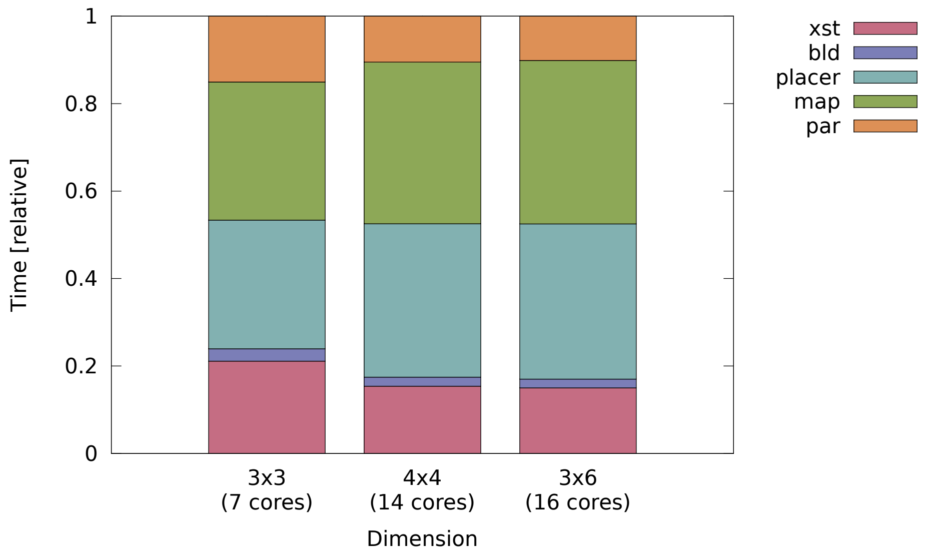


(b) Relative

Figure 7.3: Synthesis time for Integer Processor Tiles



(a) Absolute



(b) Relative

Figure 7.4: Synthesis time for Floating-Point Processor Tiles

8 Discussion

8.1 Processor Performance

The *Local Memory Read* microbenchmark measures the performance of reading local memory. As this microbenchmark only performs memory read requests to local memory and there is no other traffic on the interconnect, all latencies are from state machines in the master unit wrapping the processor, slave unit controlling the RAM and the local router.

Instruction fetches constitutes a large part of the cycles used within the *Neighboring Tile Read* and *Neighboring Tile Write* benchmark, mostly due to a slow router implementation. The program executed by a processor should be placed in its local memory to obtain good performance, which makes overall system performance highly dependent on local memory access latency. Improving local memory performance may therefore be crucial for the SHMAC's performance. The utilized RAM slave unit may be expanded to obtain this goal by assigning a designated port for the tile's master unit to the RAM, as the FPGA have dual-port block-ram RAM.

For a sufficient large problem size, that is when the parallelizable work is sufficient large with respect to communication overhead, we obtains close to linear speedup. For small problem sizes increasing core count may even decrease performance due to communication overhead. All the current benchmarks have one point of synchronization, namely the LL/SC Tile. This may lead to a high load for the LL/SC Tile as the work is equally partitioned between cores and all locking is invoked within a short time span.

8.2 Synthesis Time and Rapid Prototyping

One of the goals for this project is to allow rapid prototyping of heterogeneous processors. In comparison with using simulators, compiling a bit-file gives a significant overhead. Software may however be executed without the overhead imposed by simulation.

All the different benchmarks may be executed using the same hardware, that is there is no need for compiling more than one bit-file. By reusing bit-files this may give better performance than using simulators. Using techniques such as floorplanning and smart guide may be used for decreasing synthesis time.

8.3 Design Vulnerabilities

If an master unit issues a read operation to an address not handled by any tiles or to an tile without slave unit, the master unit may halt as it will never receive a response packet. If the interconnect is a mesh the request will simply be dropped, while the request will be routed eternally in the network as it will never reach its destination for a torus.

As the memory system provides no restrictions for memory addressing, a processor with a pipeline or using prefetching may therefore halt even if the program does not explicitly reads an invalid memory location.

If the size of the implemented mesh does not perfectly fit the allocated memory space, e.g. it is not a power two, the border of the mesh should include dummy slave units which would avoid this hazard. As there is no restrictions in addressing, the binary of a tile may write into another tile's binary which may cause undefined behavior.

8.4 Run-time System

The SHMAC does not provide a TBP framework. As the MIPS standard makes it possible to perform floating-point operations in software if the processor lacks hardware support. It may be benefittal to assign a task using few floating-point operations to an Integer Processor Tile if all Floating Point Processor Tile are busy.

8.5 Alternative Topologies and Addressing

There is huge differences between how much memory each slave unit utilizes. An alternative interconnect for dividing memory between tiles more flexible could be to use a tree-based approach. This can be done by partitioning the memory space according to a tree with interior nodes being switches routing on a subset of bits (fan-out bit-set) which size determines this interior node's fan-out. Bit-sets can be chosen such that a traversal downward the tree fixes an increasing prefix of the addresses.

The leaf nodes of the tree is the tile with master and slave units. The leaf nodes (i.e. tiles) handle all external communication by sending and receiving packets from its parent node. Router nodes compare packets' destination address with its own address where the fan-out bit-set is set as don't cares. If it doesn't match, that is the destination and the router's fixed prefix differ, the router node routes the packet to its parent. For a match it uses an ordering on its fan-out bit-set to determine which of its child nodes should receive the packet.

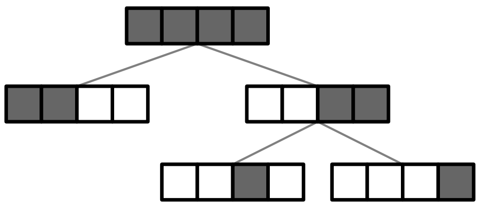


Figure 8.1: Tree Interconnect

To avoid deadlocks and keeping performance higher this may be implemented as a fat-tree with an increasing number of buffers toward the root. This may give a more flexible way of allocating different memory ranges to tiles, but may introduce limitations to where tiles can be placed as a specific tile must be placed somewhere in the tree that have enough allocatable memory range.

Figure 8.1 shows how addresses could be divided between three tiles, leaving the left-most tile the largest memory address range. In this example for example external RAM can use 16 bit, leaving 8 bit to each of the other tiles.

9 Conclusion

The main goal for this thesis has been to implement a heterogeneous processor on an FPGA. We have created a fully functional processor framework, implemented layouts with forty processor cores complete with interconnect, support for atomic locking and a memory interface connection to a host system. It consists of a number of implemented tiles which provides the needed functionality for running multiple threads of execution with the use of synchronization between threads.

SHMAC is highly modular and allows rapid prototyping of different architectural layouts by simple configuration. It supports a simple interface between the tiles such that new tiles can be easily implemented for specific needs. We have also provided a programming framework which creates a layer of abstraction for accessing the different tile's local memory from other tiles and abstractions for utilizing atomic locking in applications.

Execution of several microbenchmarks has shown that the SHMAC have a large potential for improvements. This framework is however able to get close to linear speedup for large parallel programs.

Besides the delivered prototype we have also given a specific list of future expansion possibilities which may be implemented in future projects at CARD to improve the framework. As this prototype is to be used in research on the topic of heterogeneous multi-core processor, we have focused on modularity and keeping things simple to allow for maximal flexibility. These qualities are also evident in the following listed suggestions of future expansions and enhancements.

9.1 Future Work

As this is a first prototype, a part of this thesis objective has been to give an overview of the main design challenges within the field of heterogeneous processing. The most important part of this is the realization of the SHMAC, but almost equally important is how this prototype can be further developed. During development some choices were made because of time constraints leading to and some of the design outcomes gave rise to new alternatives, or ideas for optimizations. This is knowledge acquired during development that would be lost if not documented since the maintainers of the framework is going to be carried out by CARD. The following moments therefore reflect how we see that the SHMAC framework best could be improved.

9.1.1 Processor Tile Improvements

9.1.1.1 Exception Handling

The mlite core should be expanded to allow correct exception handling, both from interrupts over the interconnect and for unsupported operations. This may then be used for setting up handlers when accessing illegal memory addresses or to allow software execution of floating-points operations if this is not enabled in hardware (e.g. when using our integer core). Since the current mlite core reserves specific regions of the global memory space for interrupt state and configuration the simplest solution would be to create a virtual IRQ Tile for these memory regions. Piggy-backing such a virtual tile to all processor tiles would make the current interrupt mechanism in the mlite-core integrate nicely. Otherwise, modifications of the core is called for. Currently there is no support for doing floating-point operations in software for the processors lacking coprocessor 1 because of the limited interrupt support available.

9.1.1.2 Caching

As the SHMAC currently is not using any caching, performance is extremely sensitive of how instructions and data is distributed in memory. Including caching would allow the processor to be more flexible and efficient, but would require a cache coherency mechanism. If implementing a cache coherency protocol then the LL/SC Tile may be retired in favor for a more standard implementation of LL/SC. This would allow arbitrary memory locations to be used in conjunction with LL/SC.

9.1.1.3 Improved FPU Performance

The floating-point enhanced processor tile have a significant complexity and long critical path. Rodolfo et al. [41] have implemented floating-point support with the Plasma CPU with good results. This work may be used to create a more efficient Floating Point Processor Tile.

9.1.1.4 Dual-Port RAM Slave Unit

The on-chip block RAM for Virtex-6 may be used as a true dual-port RAM, i.e. each port can read or write independently of each other [42]. One of those ports should connect the RAM to the router, while the other port can be made freely available for the tile for any use. Figure 9.1 gives an overview of the extra ports available for the tile, when expanding the basic on-chip block RAM unit. Simultaneous writes to the same address would result in data uncertainty and should be avoided [42].

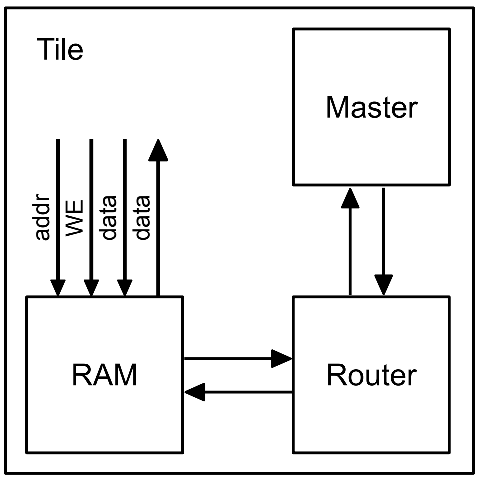


Figure 9.1: Overview of tile with dual-port block RAM

A possibility is for both ports to be connected to the router, such that the router can perform two memory operation concurrently. This would complicate the router, but does not require any further changes to the different slave and master unit interfaces.

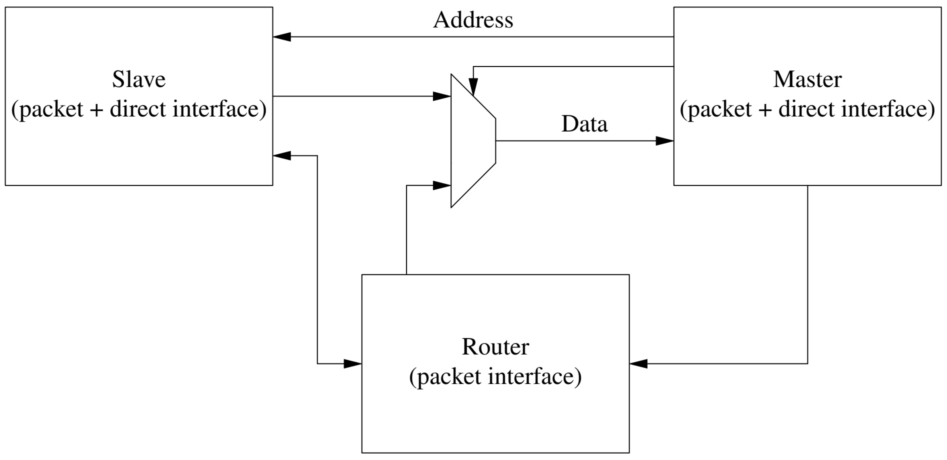


Figure 9.2: Improved processor tile

9.1.1.5 Improved Local Memory Performance

By using the suggested dual-port RAM unit, the processor core master units can be connected directly to their associated local memory. Figure 9.2 shows a suggestion for the new interconnection between these two units. The address can be hardwired from the outgoing packet of the master unit, with the data source select signal set to correspond either to data received from the local slave unit or the router dependent on the packet's destination address field. If a dual-port RAM unit is implemented then this modification is a natural next step. Only a small expansion of the master unit's state machine combined with the modifications outlined in the figure is needed to circumvent the router unit's latency. We postulate that this modification alone would increase the processor's performance ten- to thirtyfold compared to the current version based on observed versus theoretical IPC performance of the processor core's pipeline.

9.1.2 Interconnect Improvements

9.1.2.1 Configurable Memory Properties

Currently, SHMAC is defined as a 16x16 mesh, i.e. two fields of 4 bits each of the memory addresses is used for coordinates (Figure 3.4). Parametrization of these fields would give SHMAC users extra flexibility in their hardware design explorations. For instance exploring the extremes of these parameters could provide challenging environments giving rise to novel solutions. For instance, using the complete address layout for coordinates would give each tile ownership of only a single byte.

Another modification to increase the flexibility of the current memory solution is to parametrize the number of block RAMs to use in each tile. The current hardwired amount is a remnant from early design decisions where careful spending of hardware resources was considered to enable larger core counts. Still, this should be up to the user to decide, but for a first prototype it is an acceptable simplification.

9.1.2.2 Buffering at Router Ports

To increase efficiency in routing, the input and output ports may convert to larger FIFO buffers. This can decrease congestion in the mesh, but will consume more logic resources. If reordering of memory accesses is allowed then the first ready, i.e. non-blocked, packet in the buffers could be routed also possible increasing the network performance. To allow memory access reordering on the other hand would require implementing support for different memory consistency models which is a large task to undertake.

9.1.2.3 Flit-buffer Flow Control

The interconnect is using channels where each data channel is as wide as a packet. By using a flit-buffer flow control, such as wormhole routing, the interconnection and buffers would require less resources. Using less resources could improve scaling at the cost of decreased bandwidth. This trade-off should be investigated further.

9.1.2.4 Routing Resources

A FPGA, in contrast to Application-Specific Integrated Circuits (ASICs), have limited interconnection resources. For saving routing resources it is possible to have channels support only one direction between each neighboring tile. This would require the current interconnect to be setup as a torus instead of a mesh so packets can move along the complete row or column by wrapping around at the borders. The routing algorithm would become a simplified form of XY routing, where a packet is routed along the X axis's only direction until it reach the correct column, and then be routed along the Y axis until it reaches its destination. This approach will on the other hand increase communication latency. Since the synthesis reports suggest that logic is the limited resource at the moment, this would not help in increasing the core count, but a simpler network may reduce the synthesis time usage.

9.1.2.5 Routing Algorithms

The current mesh interconnect is simply converted to a torus, but this would require modifications to the routing algorithm. Even so, with the network changed to a torus the tiles could communicate using shorter paths, e.g. tiles placed at the border of the layout could be able to communicate directly. Introducing new routing algorithms could therefore reduce the average hop count of the network significantly.

Adaptive routing algorithms could also be implemented to overcome network congestion, but this may cause reordering of memory accesses and would therefore entail considerable processor tile adaptations.

9.1.3 SHMAC Enhancements

The SHMAC could easily be expanded to suit different needs by adding different application specific tiles. Some different examples is listed in this section.

9.1.3.1 DRAM Tile

The DRAM Tile makes any external DRAM on the development board available for use internally on the processor, thereby expanding the SHMAC's amount RAM as it is limited by the FPGA.

If only eight bits are used for coordinates, each tile's local address space consists of 16 MB. To exploit more of the DRAM's resources, the DRAM Tile could occupy several tile slots as discussed in Section 3.3.

9.1.3.2 Random Tile

The Random Tile is used to generate random values. It is a pure slave tile with a random generator slave unit. It could be seeded and pseudo-random or better still connected to an external true random generating unit. This may be used for randomized algorithms and for use in cryptography.

9.1.3.3 Crypto Tile

Accelerators for cryptography is common in processors today, as this is typically work that can be done efficient in hardware. If an application makes heavy use of cryptography the SHMAC could easily be expanded to include such a tile either as a coprocessor in a processor tile or as a stand-alone function tile.

9.1.3.4 Collection Tile

Many slave units utilize only a small part of their allocated memory addresses. This is true for both Clock Tile and Jump Tile for instance. If memory addresses becomes a scarce resource then such wasteful overhead should be avoided. A tile design can be created which collate a number of slave units or master units from other pure slave or master tiles. Figure 9.3 shows how the Collection Tile increases the utilization of the address space by making use of previously unused address bits.

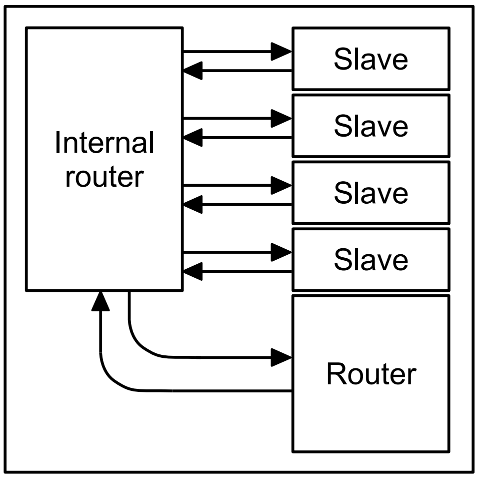


Figure 9.3: Collection Tile with slave units

The end-result is that the Collection Tile's local memory space is evenly shared by all its slave units. Figure 9.4 gives an example of how the address format could be, using two bit for internal routing where `x` and `y` is for internal address. For a tile only containing master units the internal router can be even simpler.

i	j	y	x	Unused	Address
31 28	27 24	23	22	21 12	11 0

Figure 9.4: Collection Tile's internal address representation

9.1.3.5 CA Tile

Another tile concept that shows of the possibilities with the flexibility and ease of modification enabled by the SHMAC is the Cellular Automata Tile. A sub-mesh of CA Tiles in the interconnect could be reserved for the computation of a cellular automata. The slave unit would consist of memory containing automata rules, cell state and tile control, while the master unit is responsible of following the automata rules presented by its slave. Typically the automata master unit reads its own and neighbors state and updates its own state based on this. Other tiles in the SHMAC can interface with this automata through the memory interface as per usual, e.g. a processor tile could run the genetic algorithm to decide which rules should advance to the next generation and since update the different CA Tile's rule set.

9.1.4 External Communication

9.1.4.1 UART Debug Support

For debug purposes we envision the extension of the current UART solution with the new commands HALT, STEP and SCAN. The HALT command is envisioned as a clock enable/disable for the tiles. The memory is then readily available through the UART memory interface, but registers needs a way to be accessed. By implementing a register scan line in the system the state of all computations and routing becomes accessible from the outside environment. STEP enables the clock for a single cycle, thereby stepping through execution one step at a time. This gives the possibility of inspecting the execution in great detail and would make it possible to add GDB support for the SHMAC.

9.1.4.2 Automatic Result Gathering

Another UART extension which could aid in running experiments is the addition of a data structure on the SHMAC that the host system could access after execution has reached the end on the SHMAC. If such a structure was defined then the program running experiments on the host could be guided in dumping memory to files after a run. E.g. a linked-list of memory segments' start, length and which file to store the memory dump in could simplify tedious bookkeeping work needed to document an experiment.

Another beneficial gain from this solution is the possibility of programmatically creating the structure, helpful for experiments where the result data is not necessarily known in advance where it will reside in memory, for instance if memory allocations on heap are performed. If well-known memory addresses in the UART Tile's slave unit is used to store pointers to such defined structures then UART commands could be used to perform such a memory dump without the need for the researcher to track down and dump the data manually. This addition is the last piece remaining to be able to automate the task of running large numbers of experiments.

9.1.4.3 SHMAC to Host Communication

The delivered version of SHMAC has no proper way to communicate that execution is finished since the current UART Tile is host side initiated, i.e. the host side is currently the only side capable to initiate communication as per the strict handshake contract utilized. During the thesis, LEDs have been used as indicators to whether execution has finished or not, but this is an unsatisfactory way to support for instance time measurements, because it does not enable the automation of running batches of test benches or rapidly exploring design spaces. What is currently done to circumvent this one-sidedness is to use the UART Tile in combination with the Clock Tile. Using this time keeping module the UART can know when the execution has finished by polling the Clock Tile's control register, i.e. by stopping the clock the execution can signal its end to the host side. The polling from the host side does however create unnecessary network load, and could be better solved by a control register embedded in the UART Tile's local memory space which would avoid the polling to traverse the interconnect. Better still would be to enable two-way communication over UART.

9.1.4.4 Robust UART Reset

A shortcoming with the current UART Tile is that the reset command is part of the regular state machine. If for some reason a malevolent packet is introduced, for instance because of a bug in another unit, and this is addressed to the UART's master unit then the state machine could hang. The same is true for incomplete or wrong UART communication from the host side. Reset in our current design is a command equal in priority to any other command making the command protocol's state machine fragile. A more robust design should make reset self-contained so it would always be possible to reset SHMAC to an initial state over UART.

9.1.5 Software Improvements

9.1.5.1 Work Stealing Algorithm

A run-time system based on TBP scheduling is a logical continuation of this thesis. Such scheduling systems will require efficient algorithms. Cilk and Wool is examples of contemporary homogeneous multi-core scheduling systems which make use of what is known as work stealing to distribute the global workload fairly and efficiently.

The work stealing approach may well be a good fit as a scheduling algorithm for heterogeneous multi-core processors also. Work stealing is however non-deterministic and would require that a tile can access random memory locations without large memory access latencies which would require a cache system to be implemented.

For heterogeneous architectures such as the SHMAC it should be considered hard to program parallel workloads explicitly, especially for the SHMAC case as the exploration of new layouts could introduce inconsistencies between program and hardware. A work stealing enabled run-time system which abstracts away the heterogeneous nature of the hardware for the programmer would be helpful in this regard.

9.1.5.2 TBP Run-time Engine

As the SHMAC currently does not utilize any form of cache, accessing arbitrary memory locations may introduce significant overhead. TBP would require that it is possible to map available tasks to idle processors efficiently. This will require that a core can run arbitrary tasks as they are ready for execution. As the execution time for each task may be unknown at compile-time, the mapping must be done in run-time. This is true also for scheduling algorithm utilizing work stealing, but these algorithms is dynamic so it would also be hard to predict which core will be assigned each task. Getting good performance in a architecture without cache would require that initial placement of tasks in memory is done such that each core should access memory as close at possible to the core.

Since execution time for the tasks may be unknown it would be impossible to create a optimal placement for tasks in memory that minimizes each core's access delay. One possibility would be to duplicate the most heavily used tasks throughout memory, using a heuristic for finding a good placement. Efficient scheduling within TBP without use of cache would therefore require extensive analysis of the application. Memory accesses for run-time systems utilizing work stealing scheduling would be unpredictable, which would make performance of the system equally unpredictable.

9.1.5.3 SHMAC Layout Introspection

Currently the details of the SHMAC layout is conveyed to the run-time by using auto-generated header files included at compile time. Integrating this configuration in the hardware itself using a `ROM` would allow the applications to fetch the current layout's information at run-time instead. This would make programs more portable between different FPGA implemented SHMAC layouts. For instance could it simplify any run-time system's responsibility of abstracting away differences between heterogeneous layouts as the run-time system then would not need to be recompiled together with each synthesis of new layouts.

9.1.5.4 Per-Experiment Configurations

The delivered configuration system separates the hardware and software steps of the build process. This is a remnant from our legacy configuration system were we considered matching the configuration to the assumed typical usage pattern. This would be to specify a hardware layout, generate and implement this before running multiple benchmarks or other experiments. From results gathered in this round, hardware would be modified and a new iteration could be done. Unfortunately tailoring the configuration system directly to this approach has made any other usage pattern cumbersome and makes reproduction of results dependent on a researcher dutifully document every different combinations of hardware layout and software used. This should be improved upon before using our prototype for research.

Specifically this means that the layout configuration must change. The layout schematics, setup.txt should be moved together with the experiment's software source code or possibly a new folder structure for experiments which links the two aspects together in some other way. The gen-tile-ram file system structure should be discarded in favor of reusing the experiment configuration for this step also. In addition to encapsulating the whole SHMAC setup in self-contained experiments this would assure the correct mapping of the same programs after both synthesis and experiment runs.

9.1.5.5 Alternative Linker Script

When multiple threads of execution is to share a variable belonging to one of the cores, the binary of this core must be compiled before cores depending on this variable. This restricts the order of compilation and cause problems in cases of circular dependencies.

This could be solved by creating a single binary containing the complete program memory of the SHMAC, instead of binaries per processor tile as is currently done. A linker script would need to be generated which ensures that the program sections are mapped to the different processor tiles' local memory spaces correctly. Each tile's binary could be extracted from the large binary blob to allow for selective reprogramming of individual tiles.

9.1.5.6 Configurable Topology

Currently the interconnect topology is defined to be a mesh and the configuration system assumes this when creating the toplevel description, but allowing a more flexible interconnect setup could make interesting hardware solutions possible. This would require new types of router units and routing algorithms. Moreover the configuration system must provide the means for specifying arbitrary topologies and parsing these into synthesizable hardware descriptions.

9.1.5.7 Fallback Lock Library

The current lock library generation requires a LL/SC Tile and causes builds to fail if such hardware is missing. This breaks applications which require locking on such layouts. A solution is to let the lock library generator generate a pure software locking scheme if no lock hardware is available. Generating a `locklib.o` in all cases allows for applications to build on more layouts and will reduce breakage.

9.1.5.8 Interrupt Routines

Given the possibility of both hardware and software interrupts, routines should be created to handle faults for application. A set of default interrupt handlers could be created and bundled with the framework after proper interrupt support is implemented in hardware. Some interrupt handlers are time consuming to implement, for instance the interrupt mechanism for FPU emulation to handle exceptions caused by coprocessor 1 missing from our integer tiles. As such this is worthwhile to provide for new users of the system.

9.1.5.9 Heap Support

The supported C run-time environment supports the heap concept, but due to time shortage no malloc implementation is included in the delivery. Such an implementation may be as simple as the standard textbook implementation [43] or it may take advantage of the distributed nature of the global memory space. If the textbook implementation of malloc runs out of local memory it returns an error code, but a distributed malloc could search outwards from its local tile until finding the closest neighbor with available memory to allocate.

Bibliography

[1] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 4th Edition.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. ISBN 978-0123704900.

[2] M. Horowitz and W. Dally. How scaling will change processor architecture. In *Solid-State Circuits Conference, 2004. Digest of Technical Papers. ISSCC. 2004 IEEE International*, pages 132 – 133 Vol.1, feb. 2004. doi: 10.1109/ISSCC.2004.1332629.

[3] S. Yehia and O. Temam. From sequences of dependent instructions to functions: an approach for improving performance without ilp or speculation. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 238 – 249, june 2004. doi: 10.1109/ISCA.2004.1310778.

[4] Cor Meenderinck and Ben Juurlink. Euro-Par 2008 Workshops - Parallel Processing. chapter (When) Will CMPs Hit the Power Wall?, pages 184–193. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-00954-9. doi: http://dx.doi.org/10.1007/978-3-642-00955-6_23. URL `http://dx.doi.org/10.1007/978-3-642-00955-6_23`.

[5] Susanne Albers. Energy-efficient algorithms. *Commun. ACM*, 53(5):86–96, May 2010. ISSN 0001-0782. doi: 10.1145/1735223.1735245. URL `http://doi.acm.org/10.1145/1735223.1735245`.

[6] Patrick Kurp. Green computing. *Commun. ACM*, 51(10):11–13, October 2008. ISSN 0001-0782. doi: 10.1145/1400181.1400186. URL `http://doi.acm.org/10.1145/1400181.1400186`.

[7] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011. ISSN 0001-0782. doi: 10.1145/1941487.1941507. URL `http://doi.acm.org/10.1145/1941487.1941507`.

[8] M.D. Hill and M.R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7): 33 –38, july 2008. ISSN 0018-9162. doi: 10.1109/MC.2008.209.

[9] 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. ISSN 0163-5964. doi: http://dx.doi.org/10.1145/1028176.1006707. URL `http://doi.acm.org/http://dx.doi.org/10.1145/1028176.1006707`.

[10] Nagesh Lakshminarayana, Sushma Rao, and Hyesoon Kim. Asymmetry Aware Scheduling Algorithms for Asymmetric Multiprocessors. URL `http://www.cc.gatech.edu/~hyesoon/amps_wiosca08.pdf`.

[11] Michael Zhang and Krste Asanovic. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. *SIGARCH Comput. Archit. News*, 33(2):336–345, May 2005. ISSN 0163-5964. doi: 10.1145/1080695.1069998. URL `http://doi.acm.org/10.1145/1080695.1069998`.

[12] OpenCores. OpenCores, apr 2012. URL `http://opencores.org/`. Retrived 2012-05-19.

[13] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. doi: 10.1145/1465482.1465560. URL `http://doi.acm.org/10.1145/1465482.1465560`.

[14] Daniel. SVG Graph Illustrating Amdahl's law, apr 2008. URL `http://en.wikipedia.org/wiki/File:AmdahlsLaw.svg`. Retrived 2012-05-09.

[15] Fred J. Pollack. New microarchitecture challenges in the coming generations of CMOS process technologies. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 32, pages 2–, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0437-X. URL `http://dl.acm.org/citation.cfm?id=320080.320082`.

[16] Michael D. Smith, Mark Horowitz, and Monica S. Lam. Efficient superscalar performance through boosting. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, ASPLOS-V, pages 248–259, New York, NY, USA, 1992. ACM. ISBN 0-89791-534-8. doi: 10.1145/143365.143534. URL `http://doi.acm.org/10.1145/143365.143534`.

[17] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995. ISSN 0163-5964. doi: 10.1145/216585.216588. URL `http://doi.acm.org/10.1145/216585.216588`.

[18] Intel. Datasheet: Pentium® 4 Processor Extreme Edition.

[19] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded sparc processor. *Micro, IEEE*, 25(2):21 – 29, march-april 2005. ISSN 0272-1732. doi: 10.1109/MM.2005.35.

[20] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1 –8, july 2011. doi: 10.1109/IGCC.2011.6008565.

[21] Tilera. TILEPro64^{TM} Processor Product Brief. URL `http://www.tilera.com/sites/default/files/productbriefs/TILEPro64_Processor_PB019_v4.pdf`. Retrived 2012-05-10.

[22] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 81–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2043-X. URL `http://dl.acm.org/citation.cfm?id=956417.956569`.

[23] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Task Scheduling Algorithms for Heterogeneous Processors. In *Proceedings of the Eighth Heterogeneous Computing Workshop*, HCW '99, pages 3–, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0107-9. URL `http://dl.acm.org/citation.cfm?id=795690.797895`.

[24] R. Sakellariou and H. Zhao. A hybrid heuristic for DAG scheduling on heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 111, april 2004. doi: 10.1109/IPDPS.2004.1303065.

[25] Gunnar Inge G. Sortland. Evaluation of Intel Cilk Plus using the HPC Challenge Benchmark, dec 2011.

[26] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, Cambridge, MA, USA, 1989. ISBN 0262691302.

[27] Karl-Filip Faxén. Wool - fine grained independent task parallelism in C. URL `http://www.sics.se/~kff/wool/`. Retrived 2012-06-28.

[28] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30:207–216, August 1995. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/209937.209958. URL `http://doi.acm.org/10.1145/209937.209958`.

[29] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Not.*, 33:212–223, May 1998. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/277652.277725. URL `http://doi.acm.org/10.1145/277652.277725`.

[30] Artur Podobas and Mats Brorsson. Architecture-aware Task-scheduling: A thermal approach.

[31] L. Sawalha and R.D. Barnes. Energy-efficient phase-aware scheduling for heterogeneous multicore processors. In *Green Technologies Conference, 2012 IEEE*, pages 1 –6, april 2012. doi: 10.1109/GREEN.2012.6200965.

[32] William James Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. ISBN 0122007514.

[33] Christopher J. Glass and Lionel M. Ni. The turn model for adaptive routing. *J. ACM*, 41:874–902, September 1994. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/185675.185682. URL `http://doi.acm.org/10.1145/185675.185682`.

[34] Dara Rahmati, A E Kiasari, Hamid Sarbazi-Azad, and Shaahin Hessabi. *A Power-Efficient Routing Algorithm for Torus NoCs*. 2008.

[35] L.M. Ni and P.K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2):62 –76, feb. 1993. ISSN 0018-9162. doi: 10.1109/2.191995.

[36] Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. *SIGARCH Comput. Archit. News*, 28(2):282–293, May 2000. ISSN 0163-5964. doi: 10.1145/342001.339696. URL `http://doi.acm.org/10.1145/342001.339696`.

[37] Steve Rhoads. Plasma - most MIPS I^{TM} opcodes, apr 2012. URL `http://opencores.org/project,plasma`. Retrived 2012-05-19.

[38] David Lundgren. FPU Double VHDL, feb 2010. URL `http://opencores.org/project,fpu_double`. Retrived 2012-05-19.

[39] MIPS. MIPS32® Architecture for Programmers Volume II: The MIPS32® Instruction Set.

[40] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, pages 115–116, 1981.

[41] Taciano A. Rodolfo, Ney L. V. Calazans, and Fernando G. Moraes. Floating Point Hardware for Embedded Processors in FPGAs: Design Space Exploration for Performance and Area. *Reconfigurable Computing and FPGAs, International Conference on*, 0:24–29, 2009. doi: http://doi.ieeecomputersociety.org/10.1109/ReConFig.2009.26.

[42] Xilinx. Virtex-6 FPGA Memory Resources, April 2011. URL `http://www.xilinx.com/support/documentation/user_guides/ug363.pdf`. Retrived 2012-06-03.

[43] Brian W. Kernighan and Dennis M. Ritchie. *C Programming Language (2nd Edition)*. Prentice hall, 1988. ISBN 9870131103627.

A Glossary

LL/SC Tile Tile supporting LL/SC instructions for obtaining atomic RMW. 41, 47, 49, 50, 55, 56, 65, 70, 78, 101

Clock Tile Pure slave tile for counting ticks, used for benchmarking.. 47, 56, 75, 101

Floating Point Processor Tile Tile containing a MIPS-I processor with FPU.. 47, 61, 62, 66, 70, 95

Integer Processor Tile Tile containing a MIPS-I processor without FPU.. 32, 33, 47, 60–62, 66, 95, 101

Intel© Cilk^{TM} Plus An extension to the C and C++ programming languages, designed for multithreaded parallel computing.. 17, 18

Jump Tile Pure slave tile returning a jump instruction to the start of aprocessors local memory.. 27, 40, 41, 47, 61, 101

LED Tile Pure slave tile used for setting on-board LEDs.. 29, 47

Memcached A general-purpose distributed memory caching system used for keeping results of expensive queries.. 15

MIPS MIPS is a RISC ISA developed by MIPS Technologies.. 6, 24, 31, 33, 34, 40, 48, 51, 66, 85

mlite The processor core implemented in the Plasma project. . 29, 32–34

Plasma Processor implementing most of the MIPS-I ISA. The processor is written in VHDL by Steve Rhoads. . 85

RAM Tile Pure slave tile containing on-chip block RAM.. 47

UART Tile Pure master tile used for controlling the processor through UART.. 42, 47, 61, 75, 76

Wool Wool is an library for independent task parallelism developed by Karl-Filip Faxén, assistant professor at the Royal Inst. of Technology in Stockholm.. 17, 18

B Benchmarks

This chapter gives some details about the implemented benchmarks. All of the benchmark applications does some kind of calculation, which is controlled to be correct. In addition the assembly files are controlled to assure that the compiler does not do any unwanted optimizations.

Timing is done by simply deactivating the clock. The UART will pull the value of the clock's control register with regular intervals. If the clock is deactivated, the execution of the benchmark is done and the result can be read. This should give minimal interference with the benchmark as this interval is relative long, and the benchmark have no external communication.

B.1 Microbenchmarks

These benchmarks does not require any synchronization between the tiles, and there will be no additional traffic in the mesh. All initialization needed for the benchmarks are done before clock is started to avoid any overhead. Since there should be no traffic on the interconnect, the two writes to clock should take the same time. The timing should therefore be accurate.

B.1.1 Sustained Int

This is a benchmark for instruction reads, that is reads from local memory. No communication is done outside the tile except setting the Clock Tile's control register. This benchmark executes `addiu` 2000 times.

The benchmark have no memory operations making instruction reads the only memory requests measured. Code given in Listing 9.

Listing 8 Benchmark UART configuration file

```
#include ``../include/shmac.h''
TILE(PLASMA, 0) <bin-file>.bin
```

Listing 9 Sustained Int Benchmark

```
        .globl _start
_start:
        li      $4,0

        // Reset clock
        li      $2,TILE(CLOCK,0)         # 0x1000000
        li      $3,7
        sw      $3,0($2)

        // Add
        .rept 2000
        addiu   $4,$4,1
        .endr

        // Stop clock
        sw      $0,0($2)

        // Save result
        li      $2,TILE(RAM,0)
        sw      $4,0($2)

        // Halt
1:      beq     $0,$0,1b
```

B.1.2 Read to Neighbor

This benchmark measures read performance from neighboring tile by executing `lw` 2000 times. The execution times includes sending data through two routers in addition to the performance of the master and slave unit.

Code given in Listing 10.

B.1.3 Write to Neighbor

This benchmark measures write performance to neighboring tile by executing a `sw` 2000 times. This benchmark measures when the processor is finished, not when the data is actually written to memory such that the last `sw` will not be taken into account. As the number of request is so high this inaccuracy will however not be significant. Code is given in Listing 11.

Listing 10 Read from Neighbor

```
        .globl _start
_start:
        // Initialize read address
        li      $4,TILE(RAM,0)

        // Reset clock
        li      $2,TILE(CLOCK,0)
        li      $3,7
        sw      $3,0($2)

        // Load word to $5
        .rept 2000
        lw      $5,0($4)
        .endr

        // Stop clock
        sw      $0,0($2)

        // Halt
1:      beq     $0,$0,1b
```

```
JULCPPPPPP
PPPPPPPPPP
PPPPPPPPPP
PPPPPPPPPP
```

Figure B.1: Layout for hardware in the Parallel Sum Benchmark

B.2 Parallel Sum

This benchmark consists of a `for` loop which sums the 10000 $*$ 36 iterations using 36 Integer Processor Tiles. There is no loop unrolling, as this is does not affect the relative execution time for parallel versus sequential execution.

There is one master program, which will contain the final sum. This program is also responsible for all setup. The work is divided between all cores, and is written to the master after each worker is finished. When all workers are finished, the master stops the clock. Listing 12 gives the source for the master program, while Listing 13 gives the source for the slave program. Listing 14 gives the sequential version of this algorithm.

Figure B.1 gives the layout of the hardware used for the benchmark. This layout, i.e. the same bit file, is used for all runs of the benchmark. To reduce the number of workers, some of the cores halts by running the code given in 15. Figure B.1 shows the utilized hardware, with the master program located in tile `(2,4)`. All active cores are placed as close as possible to the master program to avoid latencies.

Listing 11 Write to Neighbor

```
        .globl _start
_start:
        // Initialize read address
        li      $4,TILE(RAM,0)

        // Data
        li      $5,0x11259375       #0xabcdef

        // Reset clock
        li      $2,TILE(CLOCK,0)
        li      $3,7
        sw      $3,0($2)

        // Write 0xabcdef to TILE(RAM,0)
        .rept 2000
        sw      $5,0($4)
        .endr

        // Stop clock
        sw      $0,0($2)

        // Halt
1:      beq     $0,$0,1b
```

Listing 12 Parallel Sum - Master

```c
#include <shmac.h>
#include <locklib.h>

#pragma shmac finished
volatile unsigned int finished = 1;
#pragma shmac wait
volatile unsigned int wait = 1;
#pragma shmac N
volatile unsigned int N = 25920;
#pragma shmac sum
volatile unsigned int sum = 0;
#pragma shmac lock_address
unsigned int * lock_address;

// Numbers of workers to wait for
const int workers = 2;

int main(void){
  // Reset Clock
  unsigned int * clock_ctrl_reg = (unsigned int *) TILE(CLOCK,0);
  *clock_ctrl_reg = 7;

  // Resets Lock Tile
  init_ll();

  // Allocate lock
  int lock_error;
  alloc_lock(&lock_address, &lock_error);

  // Local Variables
  int local_sum = 0;

  // Start run
  wait = 0;

  // Do summation
  for (int i=0; i<N; ++i)
    local_sum += 1;

  // Send reply
  lock(lock_address);
  finished += 1;
  sum      += local_sum;
  unlock(lock_address);

  // Wait until all slaves are finished
  while(finished!=workers){};

  // Stop Clock Tile
  *clock_ctrl_reg = 0;

  return 0;
}
```

Listing 13 Parallel Sum - Slave

```c
#include <shmac.h>
#include <locklib.h>
#include "externs.h"

 int main(void) {

  // Gives a local copy of external lock_address
  unsigned int * lock_address;

  // Local Variables
  int local_sum = 0;

  // Wait for ready signal
  while ( *((volatile int*) ext_wait) ){};

  lock_address = *ext_lock_address;

  // Do summation
  volatile int N = *((volatile unsigned long*) ext_N); // Create local copy of counter

  for (int i=0; i<N; ++i)
    local_sum += 1;

  // Send reply
  lock(lock_address);
  *((volatile unsigned long*) ext_finished) += 1;
  *((volatile unsigned long*) ext_sum)      += local_sum;
  unlock(lock_address);

  return 0;
}
```

Listing 14 Sequential Sum

```c
#include <shmac.h>

volatile unsigned int N = 12960;

int main(void){

  // Reset Clock
  unsigned int * clock_ctrl_reg = (unsigned int *) TILE(CLOCK,0);
  *clock_ctrl_reg = 7;

  unsigned int sum = 0;

  volatile double d = 1.1;
  d += 1.1;

  sum += (volatile int) d;

  // Do summation
  for (int i=0; i<N; ++i)
    sum += 1;

  // Stop Clock Tile
  *clock_ctrl_reg = 0;

  return 0;
}
```

Listing 15 Halt Program

```
        .globl _start
_start:
        // Halt
1:      beq     $0,$0,1b
```

C Synthesis Reports

This appendix contains the synthesis reports for the largest integer and FPU designs fitting the utilized FPGA as shown below. There is in addition synthesis report for a mixed integer/FPU design.

Listing 16 Design with 40 Integer Processor Tiles

Listing 17 Design with 16 Floating Point Processor Tiles

Listing 18 Design with 20 Integer Processor Tiles and 8 Floating Point Processor Tiles

Listing 16 Synthesis Report for 40 Integer Processor Tiles using a 6x7 mesh. Excerpt from `shmac.par`

```
Slice Logic Utilization:
  Number of Slice Registers:                55,213 out of 301,440   18%
    Number used as Flip Flops:              55,148
    Number used as Latches:                     65
    Number used as Latch-thrus:                  0
    Number used as AND/OR logics:                0
  Number of Slice LUTs:                    142,204 out of 150,720   94%
    Number used as logic:                  137,024 out of 150,720   90%
      Number using O6 output only:         129,922
      Number using O5 output only:              40
      Number using O5 and O6:                7,062
      Number used as ROM:                        0
    Number used as Memory:                   5,120 out of  58,400    8%
      Number used as Dual Port RAM:          5,120
        Number using O6 output only:             0
        Number using O5 output only:             0
        Number using O5 and O6:              5,120
      Number used as Single Port RAM:            0
      Number used as Shift Register:             0
    Number used exclusively as route-thrus:     60
      Number with same-slice register load:     58
      Number with same-slice carry load:         2
      Number with other load:                    0

Slice Logic Distribution:
  Number of occupied Slices:                37,564 out of  37,680   99%
  Number of LUT Flip Flop pairs used:      144,377
    Number with an unused Flip Flop:        91,427 out of 144,377   63%
    Number with an unused LUT:               2,173 out of 144,377    1%
    Number of fully used LUT-FF pairs:      50,777 out of 144,377   35%
    Number of slice register sites lost
      to control set restrictions:               0 out of 301,440    0%
```

Listing 17 Synthesis Report for 16 Floating Point Processor Tiles using a 3x6 mesh. Excerpt from `shmac.par`

```
Slice Logic Utilization:
  Number of Slice Registers:                94,369 out of 301,440   31%
    Number used as Flip Flops:              94,304
    Number used as Latches:                     65
    Number used as Latch-thrus:                  0
    Number used as AND/OR logics:                0
  Number of Slice LUTs:                    141,539 out of 150,720   93%
    Number used as logic:                  135,798 out of 150,720   90%
      Number using O6 output only:         111,077
      Number using O5 output only:           1,805
      Number using O5 and O6:               22,916
      Number used as ROM:                        0
    Number used as Memory:                   4,816 out of  58,400    8%
      Number used as Dual Port RAM:          4,800
        Number using O6 output only:            64
        Number using O5 output only:         1,344
        Number using O5 and O6:              3,392
      Number used as Single Port RAM:            0
      Number used as Shift Register:            16
        Number using O6 output only:            16
        Number using O5 output only:             0
        Number using O5 and O6:                  0
    Number used exclusively as route-thrus:    925
      Number with same-slice register load:    774
      Number with same-slice carry load:       150
      Number with other load:                    1

Slice Logic Distribution:
  Number of occupied Slices:                36,810 out of  37,680   97%
  Number of LUT Flip Flop pairs used:      143,552
    Number with an unused Flip Flop:        66,951 out of 143,552   46%
    Number with an unused LUT:               2,013 out of 143,552    1%
    Number of fully used LUT-FF pairs:      74,588 out of 143,552   51%
    Number of slice register sites lost
      to control set restrictions:               0 out of 301,440    0%
```

Listing 18 Synthesis Report for 20 Integer Processor Tiles and 8 Floating-point Processor Tiles using a 4x6 mesh.
Excerpt from `shmac.par`

```
Slice Logic Utilization:
  Number of Slice Registers:                    74,791 out of 301,440   24%
    Number used as Flip Flops:                  74,726
    Number used as Latches:                         65
    Number used as Latch-thrus:                      0
    Number used as AND/OR logics:                    0
  Number of Slice LUTs:                        142,166 out of 150,720   94%
    Number used as logic:                      136,969 out of 150,720   90%
      Number using O6 output only:             120,382
      Number using O5 output only:                 911
      Number using O5 and O6:                   15,676
      Number used as ROM:                            0
    Number used as Memory:                       4,968 out of  58,400    8%
      Number used as Dual Port RAM:              4,960
        Number using O6 output only:                32
        Number using O5 output only:               672
        Number using O5 and O6:                  4,256
      Number used as Single Port RAM:                0
      Number used as Shift Register:                 8
        Number using O6 output only:                 8
        Number using O5 output only:                 0
        Number using O5 and O6:                      0
    Number used exclusively as route-thrus:        229
      Number with same-slice register load:        139
      Number with same-slice carry load:            82
      Number with other load:                        8

Slice Logic Distribution:
  Number of occupied Slices:                    37,379 out of  37,680   99%
  Number of LUT Flip Flop pairs used:          144,616
    Number with an unused Flip Flop:            78,664 out of 144,616   54%
    Number with an unused LUT:                   2,450 out of 144,616    1%
    Number of fully used LUT-FF pairs:          63,502 out of 144,616   43%
    Number of slice register sites lost
      to control set restrictions:                   0 out of 301,440    0%
```

D Configuration

Configuration of toplevel tile layout is done by editing the file `toplevel/setup.txt` as descibed in 5.1 on page 46. By using the `Makefile`; this configuration builds `toplevel/toplevel.vhd` and `include/shmac.h`. The actual hardware is build using this toplevel. The header is used for easing access to each tile's memory locations.

```
JPPPP
PFPFP
PPPPP
```

Figure D.1: Example tile layout configuration file

Listing 19 Example of accessing a tile's memory

```c
/* Memory of the Clock Tile's registers */

// Set directly at compile time
int * clock_ctrl_register = TILE(CLOCK, 0) + 0;
int * clock_count0        = TILE(CLOCK, 0) + 1;
int * clock_count1        = TILE(CLOCK, 0) + 2;

// Create array at compile time
// Allows programmatic control of clock if present in the layout
int ** clock_tile_array = TILE(CLOCK, ARRAY);
if (TILE(CLOCK, NUM))
{
  int * clock = clock_tile_array[0];
  int * clock_ctrl_register = *(clock+0);
  int * clock_count0        = *(clock+1);
  int * clock_count1        = *(clock+2);
}
```

Listing 19 shows how memory addresses to the different tiles can be done.

E Tutorial

This tutorial assumes a clean checkout of the SHMAC repository, and that the ISE is installed.

During this tutorial we will create a simple “bank application”: Two workers start with £500000 each, and repeatedly deposits £1 to a shared account until their local account is empty. This tutorial demonstrates how to access variables from other tiles, usage of locking library and benchmarking.

Set up Configuration of SHMAC

The hardware to be used for this application contains two Integer Processor Tiles and a LL/SC Tile. In addition the processors would require a Jump Tile. Benchmarking will be done using the Clock Tile. To be able to program the system after synthesis an UART Tile is included. All of the bank data is placed in the RAM Tile at $(2, 0)$.

We have two workers, namely `east` and `west`. This application does not make use of any runtime-system, with one worker tied to each core. To create this mapping the following setup is used:

J	C	U
P	L	P
R	R	R

Listing 20 on the following page gives the generated headerfile for this setup. Each coordinate is set to use four bit.

Listing 20 Tutorial: shmac.h

```c
/* Header file for SHMAC
 * author: Leif Tore Rusten and Gunnar Inge G. Sortland
 *
 * Usage: TILE(NAME, NUMBER)
 * TILE(CLOCK, 0) gives address to first clock tile
 * TILE(CLOCK, NUM) gives number of clock tiles
 */

#ifndef SHMAC_H
#define SHMAC_H
#define TILE(tile_type, tile_num) TILE_ ## tile_type ## _ ## tile_num
#define TILE_JUMP_0 0
#define TILE_CLOCK_0 16777216
#define TILE_UART_0 33554432
#define TILE_PLASMA_0 268435456
#define TILE_LLSC_0 285212672
#define TILE_PLASMA_1 301989888
#define TILE_RAM_0 536870912
#define TILE_RAM_1 553648128
#define TILE_RAM_2 570425344
#define TILE_CLOCK_NUM 1
#define TILE_FPU_NUM 0
#define TILE_JUMP_NUM 1
#define TILE_LLSC_NUM 1
#define TILE_PLASMA_NUM 2
#define TILE_LED_NUM 0
#define TILE_RAM_NUM 3
#define TILE_UART_NUM 1
#define TILE_CLOCK_ARRAY {16777216}
#define TILE_FPU_ARRAY {}
#define TILE_JUMP_ARRAY {0}
#define TILE_LLSC_ARRAY {285212672}
#define TILE_PLASMA_ARRAY {268435456, 301989888}
#define TILE_LED_ARRAY {}
#define TILE_RAM_ARRAY {536870912, 553648128, 570425344}
#define TILE_UART_ARRAY {33554432}
#endif
```

The Bank Application

Listing 21 Tutorial: bank_west.c

```c
#include <shmac.h>
#include <locklib.h>
#include "externs.h"

 int main(void) {

  // Gives a local copy of external lock_address
  unsigned int * lock_address;

  // Accounts
  unsigned int * shared = (unsigned int*) TILE(RAM, 1) + 0;
  unsigned int * local  = (unsigned int*) TILE(RAM, 1) + 1;

  // Initialize accounts
  *local = 500000;

  // Wait for ready signal, since lock is then procured.
  while ( *((volatile int*) ext_wait) ){};

  lock_address = *ext_lock_address;

  // Add money to shared account as long as we have some
  while ( *local > 0 )
  {
    // Add £1 to shared, remove £1 from local
    *local  -= 1;
    lock(lock_address);     //
    *shared += 1;           // Critical section
    unlock(lock_address);   //
  }

  *((volatile int) ext_wait) = 1;

  return 0;
}
```

Listing 22 Tutorial: bank_east.c

```c
#include <shmac.h>
#include <locklib.h>

#pragma shmac wait
int wait = 1;
#pragma shmac lock_address
unsigned int * lock_address;

int main(void){

  // Resets Lock Tile
  init_ll();

  // Allocate lock
  int lock_error;
  alloc_lock(&lock_address, &lock_error);

  // Accounts
  unsigned int * shared = (unsigned int *) TILE(RAM, 1) + 0;
  unsigned int * local  = (unsigned int *) TILE(RAM, 1) + 2;

  // Clock
  unsigned int * clock_ctrl_reg = (unsigned int *) TILE(CLOCK,0);

  // Initialize accounts
  *local  = 500000; // Initial amount for east
  *shared =   0; // Initial amount in shared account

  // Reset Clock Tile
  *clock_ctrl_reg = 7;

  // Start run
  wait = 0;

  // Add money to shared account as long as we have some
  while ( *local > 0 )
  {
    // Add £1 to shared, remove £1 from local
    *local  -= 1;
    lock(lock_address);    //
    *shared += 1;          // Critical section
    unlock(lock_address);  //
  }
  while ( wait == 0 ){};

  // Stop Clock Tile
  *clock_ctrl_reg = 0;

  return 0;
}
```

The program running in `(1,0)` will be in charge of setting up all shared resources, such as lock and initialization of the bank's account. Listing 21 on page 103 and Listing 22 on the facing page gives the source code of these applications. `west` must wait until `east` have finished all initializations before it can start execution. This is done by `east` sharing a variable `wait = 1`, used as a spinlock by `west`. The same variable is used as a barrier for allowing `east` to ensure `west` has finished execution. This way the execution can be seen as a DAG where execution both starts and ends in `east`.

The following symbolic link are created to setup the linker scripts used for compiling:

```
ln -s plasma-prog/tutorial/bank_west gen-tile-rams/setup/1_0
ln -s plasma-prog/tutorial/bank_east gen-tile-rams/setup/1_2
```

`east` shares variables to other binaries. This are listed by using *#pragma shmac wait* so that the compiling framework will generate a pointer to this address in `externs.h`. Listing 23 shows an example of this generated headerfile.

Listing 23 Tutorial: externs.h

```
/* Header file containing external symbols
 * author: Leif Tore Rusten and Gunnar Inge G. Sortland
 *
 * Example:
 *    #pragma shmac <name>
 *    int <name> = 1
 */

#ifndef EXTERNS_H
#define EXTERNS_H
volatile unsigned long * const ext_wait = (void*) 0x120005cc;          // bank_east
volatile unsigned long * const ext_lock_address = (void*) 0x120005d0; // bank_east
#endif
```

Programming the SHMAC

The bit file is generated by invoking `make`. This will also generate `shmac.h` and linkerscripts for binaries with symbolic links present in `gen-tile-rams/setup/`. This will also invoke the application's `Makefile`.

`uart-run-conf.sh` located in the `uart-tools` folder will be used for loading the application to the FPGA's RAM. We will use the configuration file given below. Loading the application is done by invoking `./uart-run-conf.sh tutorial.conf`.

```
$ cat tutorial.conf
TILE(PLASMA, 0) bank_west.bin
TILE(PLASMA, 1) bank_east.bin
```

Validating output

The content of the accounts can be accessed using `uart-memdump <from address> <to address>` where addresses are given in decimal values. The memory location of the different addresses can be found by invoking `nmbank\_<west/east>.elf`.

The following values can be found after execution:

```
000f4240 // Account shared   1000000
00000000 // Account west           0
00000000 // Account east           0

00000000 // Ctrl register
00000000 // Ticks 0                0
37b69cff // Ticks 1        934714623
```

Execution time was equal to 934714623/33 MHz = 28 sec.