Institutionen för systemteknik
Department of Electrical Engineering

Examensarbete

Modeling and algorithm adaptation for a novel parallel DSP processor

Examensarbete utfört i Datateknik
vid Tekniska högskolan i Linköping
av

Johan Olsson, Olof Kraigher

LiTH-ISY-EX--09/4269--SE

Linköping 2009

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings tekniska högskola
Linköpings universitet
581 83 Linköping

Modeling and algorithm adaptation for a novel parallel DSP processor

Examensarbete utfört i Datateknik
vid Tekniska högskolan i Linköping
av

Johan Olsson, Olof Kraigher

LiTH-ISY-EX--09/4269--SE

Handledare: **Dake Liu**
ISY, Linköping university

Examinator: **Dake Liu**
ISY, Linköping university

Linköping, 10 June, 2009

Avdelning, Institution
Division, Department

Department of Electrical Engineering
Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Datum
Date

2009-06-10

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN
__

ISRN
LiTH-ISY-EX--09/4269--SE

Serietitel och serienummer **ISSN**
Title of series, numbering __

URL för elektronisk version
http://www.da.isy.liu.se/
http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-ZZZZ

Titel
Title
Modellering och algoritm-anpassning för en ny parallell DSP-processor
Modeling and algorithm adaptation for a novel parallel DSP processor

Författare
Author
Johan Olsson, Olof Kraigher

Sammanfattning
Abstract

The P3RMA (**P**rogrammable, **P**arallel, and **P**redictable **R**andom **M**emory **A**ccess) processor, currently being developed at Linköping University Sweden, is an attempt to solve the problems of parallel computing by utilizing a parallel memory subsystem and splitting the complexity of address computations with the complexity of data computations. It is targeted at embedded low power low cost computing for mobile phones, handsets and basestations among many others.

By studying the radix-2 FFT using the P3RMA concept we have shown that even algorithms with a complex addressing pattern can be adapted to fully utilize a parallel datapath while only requiring additional simple addressing hardware. By supporting this algorithm with a SIMT instruction almost 100% utilization of the datapath can be achieved.

A simulator framework for this processor has been proposed and implemented. This simulator has a very flexible structure featuring modular addition of new instructions and configurable hardware parameters. The simulator might be used by hardware developers and firmware developers in the future.

Nyckelord
Keywords DSP, FFT, Parallell, Simulator, P3RMA, SIMD

Abstract

The P3RMA (**P**rogrammable, **P**arallel, and **P**redictable **R**andom **M**emory **A**ccess) processor, currently being developed at Linköping University Sweden, is an attempt to solve the problems of parallel computing by utilizing a parallel memory subsystem and splitting the complexity of address computations with the complexity of data computations. It is targeted at embedded low power low cost computing for mobile phones, handsets and basestations among many others.

By studying the radix-2 FFT using the P3RMA concept we have shown that even algorithms with a complex addressing pattern can be adapted to fully utilize a parallel datapath while only requiring additional simple addressing hardware. By supporting this algorithm with a SIMT instruction almost 100% utilization of the datapath can be achieved.

A simulator framework for this processor has been proposed and implemented. This simulator has a very flexible structure featuring modular addition of new instructions and configurable hardware parameters. The simulator might be used by hardware developers and firmware developers in the future.

Sammanfattning

P3RMA (**P**rogrammable, **P**arallel, and **P**redictable **R**andom **M**emory **A**ccess) är en processor som för tillfället är under utveckling vid Linköpings universitet. Dess syfte är att försöka lösa problem vid parallella datorberäkningar. Detta löses genom att implementera en parallell minnesarkitektur och dela upp komplexiteten för adress- och databeräkningar. Processorn är ämnad för inbyggda system såsom mobiltelefoner, handdatorer och basstationer med låg effektförbrukning och beräkningskostnad.

Genom att studera radix-2 FFT tillsammans med P3RMA-konceptet har vi visat att även algoritmer med komplexa adressmönster kan anpassas för att fullständigt utnyttja en parallell dataväg med bara simpla hårdvarutillägg. En SIMT-instruktion med stöd för denna algoritm kan nästintill nå ett 100-procentigt utnyttjande av datavägen.

För denna processor har ett simulator-ramverk föreslagits och implementerat. Detta har en mycket flexibel struktur och stödjer tillägg av nya instruktioner på ett modulärt sätt samt konfigurerbara hårdvaruparametrar. Simulatorn kan användas av hårdvaru- och mjukvaru-utvecklare.

Acknowledgments

We would like to thank our examiner and mentor Professor Dake Liu for the opportunity to do our master thesis as a part of his research on parallel computing. We would also like to thank the research team for interesting discussions and Ph.D. student Jian Wang for his contributing work on the simulator. It has been a rewarding and interesting experience working with you all.

Contents

Chapter 1

Introduction

1.1 Background

The market today is demanding faster and smaller chips with lower power consumption than their predecessors. Moore's law states that the number of transistors on a chip doubles every two years. In the past the clock frequency has been able to scale along with the shrinking of the feature size, but not any more. Increases in clock frequency has hit a wall because of its negative impact on power consumption. A higher power consumption raises the cost of cooling for stationary devices and reduces the battery life of mobile devices. This raises the question of how to develop faster chips for applications where low power consumption is of most importance. How can chip designers utilize higher transistor density instead of a higher clock frequency? The answer is parallel architectures.

With a parallel architecture the single core performance is of course bounded by the clock frequency. High performance is instead achieved by utilizing multiple parallel cores each running at the lower frequency. Because of the increasing transistor density the number of feasible parallel processing elements on a chip will be able to scale with Moore's law. Parallel architectures are very attractive because of their potentially higher performance at a much lower power consumption.

The switch from single core to multi core architectures will confront the software industry with a big problem. Most software are written to maximally utilize the performance of single core processors. There is no easy translation to multi core processors. New software and software development tools will have to be created to assist parallel programming.

Increases in raw computational power with parallel cores poses great demands on the memory subsystem. The memory subsystem also has to be parallelized which requires the study of parallel access patterns of algorithms for conflict free parallel access. These parallel access patterns raises the cost of the already expensive address computations. The memory subsystem is often a big bottle neck in parallel systems leading to starvation of the datapath.

The P3RMA (**P**rogrammable, **P**arallel, and **P**redictable **R**andom **M**emory **A**ccess) processor, currently being developed at Linköping University Sweden, is

an attempt to solve these issues by utilizing a parallel memory subsystem and splitting the complexity of adress computations with the complexity of data computations. It is targeted at embedded low power low cost computing for mobile phones, handsets and basestations among many others.

1.2 Purpose

The purpose of this thesis is to create a simulator for the hardware model of the SIMD co-processors in the P3RMA processor. An existing simulation model of the memory subsystem should also be integrated.

The simulator should ultimately be used by firmware developers but also by hardware designers to profile the instruction set and other design parameters of the P3RMA processor.

Another thesis goal is to provide a case study of an algorithm using the P3RMA concept. The algorithm chosen for study is the FFT (**F**ast **F**ourier **T**ransform). This will show the benefits of the P3RMA concept and give useful knowledge about the requirements on the address generating hardware.

Since the project itself is in an early development stage one must take into account that the system parameters might change several times and in the end might not bear any similarities to the early specifications. Hence the simulator has to be flexible and make few assumptions while being easy to manipulate. The idea is that, as more and more design decisions are made the simulator will be more fixed function but this will not be the scope of this thesis.

1.3 Thesis outline

For the readers consideration follows a brief overview of the outline of the thesis. This is for easier understanding the structure of the thesis but also to get the idea of what is found in each chapter.

- **Chapter 1: Introduction** Gives the reader a background about the situation today on the DSP market and to what problems are ment to be solved in this thesis.
- **Chapter 2: P3RMA Concept** Describing the P3RMA concept.
- **Chapter 3: Hardware model** The hardware model used for the P3RMA processor is described.
- **Chapter 4: Pre-study** Describes the implementation philosophy and problems faced during the development of the simulator. Also tools used for developing will be given a short introduction.
- **Chapter 5: Simulator framework** This chapter describes the resulting framework, how the simulator was actually implemented in C++.

- **Chapter 6: User manual** A manual on how to use and develop new features for the simulator is included.

- **Chapter 7: Case study: FFT** The FFT algorithm is implemented using the P3RMA concept.

- **Chapter 8: Discussion** Conclusions can be found in the last chapter along with recommendations for future work to be done.

- **Appendices** The instruction set used during the development can be found here along with class definitions for the hardware modules of the simulator.

1.4 Abbreviations

AGU	Address Generation Unit
ALU	Arithmetic Logic Unit
API	Application Programming Interface
DM	Data Memory
DMA	Direct Memory Access
DSC	Digital Signal micro-Controller
DSP	Digital Signal Processor
FFT	Fast Fourier Transformation
FIFO	First In First Out
FSM	Finit State Machine
I/O	Input / Output
LM	Local Memory
MAC	Multiply And aCcumulate
MCU	Micro-Controller Unit
MIMO	Multiple Input Multiple Output
MISO	Multiple Input Single Output
MMX	MultiMedia eXtension
P3RMA	Programmable, Parallel, and Predictable Random Memory Access
PC	Program Counter
PE	Processing Element
PM	Program Memory
RF	Register File
RISC	Reduced Instruction Set Computer
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SWIG	Simplified Wrapper and Interface Generator
XML	Extensible Markup Language

1.5 Glossary

Assembly file parser	A program that translates assembly code into simulator specific data.
Benchmark	Measure the performance of a software.
Digital signal microcontroller	A DSP processor with basic MCU features.
Framework	A software that includes support programs, libraries, scripting languages, or other software to help tie different components of a software project together.
Hardware module	A single hardware entity eg. ALU, Register File
OpenCL	A framework based on C for writing programs that execute across heterogeneous platforms.
Python	High-level general-purpose programming language.

Chapter 2

P3RMA Concept

2.1 Introduction

P3RMA stands for **P**rogrammable **P**arallel memory architecture for **P**redictable **R**andom **M**emory **A**ccess. P3RMA is a memory solution to supply parallel data to computing engines not relying on insufficient and unpredictable caches or ultra large register files with high cost and high power consumption. This also includes separating data and addressing paths to decrease the computing latency to that of only the latency of arithmetic computing. [2]

The P3RMA concept uses parallel scratchpad memories to achieve a high theoretical bandwidth. Let us consider an eight way parallel scratchpad memory; each bank carrying 16-bit words.

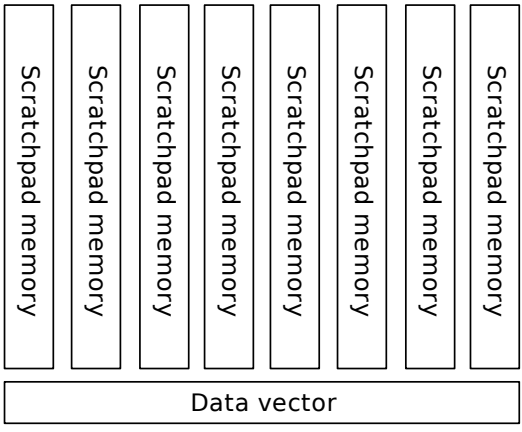

Figure 2.1. Eight way parallel scratchpad memories

For a given algorithm, each 16-bit data is distributed in the eight parallel scratchpad memories. The eight memories should then be able to deliver eight 16-bit

data in parallel for a total bandwidth of 128-bits. This is great but it makes the assumption that data words needed by the computing is located in different banks of the parallel scratchpad memory. When data that needs to be accessed in parallel is not located in different banks there is a conflict and the computing has to be stalled. To identify and analyse the access patters of algorithms is therefore of vital importance to allow conflict free parallel memory access.

2.2 Conflict free parallel memory access

To describe the allocation of one data vector into the parallel scratchpad memory lets first map its address space.

Bank 0	Bank 1	Bank 2	Bank 3	Bank 4	Bank 5	Bank 6	Bank 7
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
$8i+0$	$8i+1$	$8i+2$	$8i+3$	$8i+4$	$8i+5$	$8i+6$	$8i+7$
$N-8$	$N-7$	$N-6$	$N-5$	$N-4$	$N-3$	$N-2$	$N-1$

Table 2.1. Address space of an eight way parallel scratchpad memory

Then a set of addresses $a_i, i = 0 \dots 7$ can be accessed in parallel without conflict if:

$$i \neq j \Rightarrow a_i \not\equiv a_j \mod 8 \tag{2.1}$$

Which basically means that each access is mapped to a different bank.

Every conflict free access into the parallel scratchpad memory can be viewed as feeding one address to each bank and reordering the output to form the final data vector. This can be viewed as addressing the vector memory with a vector address. Each element in the vector address contains the bank address,annotated a_i, and the bank selection, annotated S_i, following to the notation of [3].

S_0, a_0	S_1, a_1	S_2, a_2	S_3, a_3	S_4, a_4	S_5, a_5	S_6, a_6	S_7, a_7

Table 2.2. Vector address (S, a) for an eight way scratchpad memory

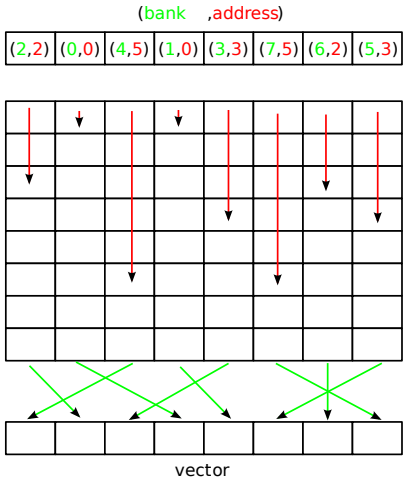

Figure 2.2. Vector address access into an 8-way parallel scratchpad memory

2.3 Permutations for conflict free parallel access

Imagine a set of addresses a_i, $i = 0 \dots 7$ that needs to be accessed in parallel but unfortunately (2.1) does not hold. That means they cannot be accessed in parallel. To be able to access them in parallel the data needs to be allocated in a different way so that (2.1) does hold. Such an allocation is called a permutation. A permutation is a bijective function $P(i) \rightarrow j$, $i = 0 \dots N$ which defines a new data allocation.

Bank 0	Bank 1	Bank 2	Bank 3	Bank 4	Bank 5	Bank 6	Bank 7
$P^{-1}(0)$	$P^{-1}(1)$	$P^{-1}(2)$	$P^{-1}(3)$	$P^{-1}(4)$	$P^{-1}(5)$	$P^{-1}(6)$	$P^{-1}(7)$
$P^{-1}(8)$	$P^{-1}(9)$	$P^{-1}(10)$	$P^{-1}(11)$	$P^{-1}(12)$	$P^{-1}(13)$	$P^{-1}(14)$	$P^{-1}(15)$
$P^{-1}(8i+0)$	$P^{-1}(8i+1)$	$P^{-1}(8i+2)$	$P^{-1}(8i+3)$	$P^{-1}(8i+4)$	$P^{-1}(8i+5)$	$P^{-1}(8i+6)$	$P^{-1}(8i+7)$
$P^{-1}(N-8)$	$P^{-1}(N-7)$	$P^{-1}(N-6)$	$P^{-1}(N-5)$	$P^{-1}(N-4)$	$P^{-1}(N-3)$	$P^{-1}(N-2)$	$P^{-1}(N-1)$

Table 2.3. Address space of a eight way parallel scratchpad memory after permutation

The set of addresses a_i, $i = 0 \dots 7$ can then be accessed in parallel without conflict if and only if:

$$i \neq j \Rightarrow P(a_i) \not\equiv P(a_j) \mod 8 \tag{2.2}$$

2.3.1 Example

Lets study an example algorithm that operates on a square matrix $A_{64\times64}$ which has 64 rows and 64 columns. The algorithm needs access to eight consecutive elements in a row but also to eight consecutive elements in a column.

Lets first map the elements of the matrix to the linear address space $0 \dots 4095$ by the following mapping:

$$A_{rc} \rightarrow r + 64c \tag{2.3}$$

Then the a consecutive row access pattern starting at A_{rc} is:

$$a_i = 64r + c + i \equiv c + i \mod 8,\ i = 0 \dots 7 \tag{2.4}$$

and a consecutive column access pattern starting at A_{rc} is:

$$a_i = 64(r + i) + c \equiv c \mod 8,\ i = 0 \dots 7 \tag{2.5}$$

It is easy to verify that the row access pattern satisfies 2.1 but that the column access pattern does not.

Fortunately the following permutation will make both access patterns conflict free:

$$8 \left\lfloor \frac{i}{8} \right\rfloor + \left(\left(\left\lfloor \frac{i}{64} \right\rfloor + i \right) \mod 8 \right) \tag{2.6}$$

The permutation in 2.6 actually corresponds to a cyclic rotation of the banks used to store a row depending on its equivalence class mod 8. The following table illustrates this:

	Bank 0	Bank 1	Bank 2	Bank 3	Bank 4	Bank 5	Bank 6	Bank 7
Row $8i+0$	0	1	2	3	4	5	6	7
Row $8i+1$	1	2	3	4	5	6	7	0
Row $8i+2$	2	3	4	5	6	7	0	1
Row $8i+3$	3	4	5	6	7	0	1	2
Row $8i+4$	4	5	6	7	0	1	2	3
Row $8i+5$	5	6	7	0	1	2	3	4
Row $8i+6$	6	7	0	1	2	3	4	5
Row $8i+7$	7	0	1	2	3	4	5	6

Table 2.4. Data allocation for row and column based conflict free access

By carefully analyzing the table above one can see that the parallel accesses used can be formulated as sixteen base vector addresses, eight for row accesses and eight for column accesses, plus an offset address. This enables this particular permutation to be efficiently used with the AGU described in section **??**.

The form of the sixteen base vector addresses used are listed by the following tables:

$r+0,0$	$r+1,0$	$r+2,0$	$r+3,0$	$r+4,0$	$r+5,0$	$r+6,0$	$r+7,0$

Table 2.5. Eight base vector addresses (S, a) for row access, $r = 0 \dots 7$ (The $+$ uses wrap-around arithmetic)

$c-0,8(c+0)$	$c-1,8(c+1)$	$c-2,8(c+2)$	$c-3,8(c+3)$	$c-4,8(c+4)$	$c-5,8(c+5)$	$c-6,8(c+6)$	$c-7,8(c+7)$

Table 2.6. Base vector address (S, a) for column access, $c = 0 \dots 7$ (The $-, +$ uses wrap-around arithmetic)

2.4 Kernels

Before starting to analyze the access patterns of algorithms their inner most loops must be identified. The core of their computations called kernels. A kernels is an inner most loop containing some regular computations. Data might be accumulated in the loop in the case of an FIR-filter or directly written back to memory in the case of FFT.

```
for (int i=0; i<N; i++) {
    /* 1. Address computations */

    /* 2. Memory access */

    /* 3. Data computation */

    /* 4. Accumulation */
}
```

Listing 2.1. Accumulating kernel, example: FIR-filter

```
for (int i=0; i<N; i++) {
    /* 1. Load address computations */

    /* 2. Memory access */

    /* 3. Data computation */

    /* 4. Store address computations */

    /* 5. Memory access */
}
```

Listing 2.2. Load-Store kernel, example: FFT

When looking at pseudo-code for kernels the cost of addressing appears, something often overlooked when looking at the pure mathematical definition of an algorithm. Often the number of multiplications or additions required in the data computations are well studied but addressing computations neglected.

In their initial form these kernels are often formulated in a non-parallel way since traditionally most machines are not parallel or do not expose parallelism to the programmer in the case of superscalar. Also when looking at algorithms on a mathematical level, parallelism has no relevance. The only interesting thing is the mathematical relationships. Of course to someone implementing the algorithm on a parallel machine parallelism is of the most importance.

To accelerate these kernels on a parallel machine one would run many iterations of the loop in parallel.

```
for (int i=0; i<N/P; i++) {
    /* 1. P address computations in parallel */

    /* 2. P memory accesses in parallel */

    /* 3. P data computation in parallel */
```

```
    /* 4. P address computations in parallel */

    /* 5. P memory accesses in parallel */
}
```

Listing 2.3. Parallel kernel

Unfortunately this trivial rewriting might not be valid since in the case of a load-store type of kernel there might be data dependencies between the loop iterations. Also the addresses might not be in different banks and therefore not accessible in parallel, this requires the use of a memory permutation which increases the cost of addressing even more.

```
for (int i=0; i<N/P; i++) {
    /* 1. P address computations in parallel */
    /* 2. P address permutations in parallel */
    /* 3. P memory accesses in parallel */

    /* 4. P data computation in parallel */

    /* 5. P address computations in parallel */
    /* 6. P address permutations in parallel */
    /* 7. P memory accesses in parallel */
}
```

Listing 2.4. Parallel kernel with permutation

An important observation is that the datapath cannot simply be scaled up for parallel computations without scaling up the AGU (Address generating unit) and the memory subsystem with it. A non-parallel memory subsystem can never support a parallel datapath it will just be a waste. Even when pairing a parallel datapath with a parallel scratchpad memory there is much extra overhead for address permutation computations. To design the AGU for parallel addressing with addressing permutations is therefore very critical for achieving high utilization of the datapath.

2.5 AGU for conflict free parallel access

To directly supply a vector address to the parallel scratchpad memory from the instruction code would greatly increase cost of the code memory and is not a desirable solution. Only a limited amount of address bits can be supplied from the instruction code and they have to be the seed from which the vector address is computed. There might also be other values which need to influence the address generation such as the hardware loop counter. The AGU also has to perform the computations associated with the memory permutation used to support conflict free access.

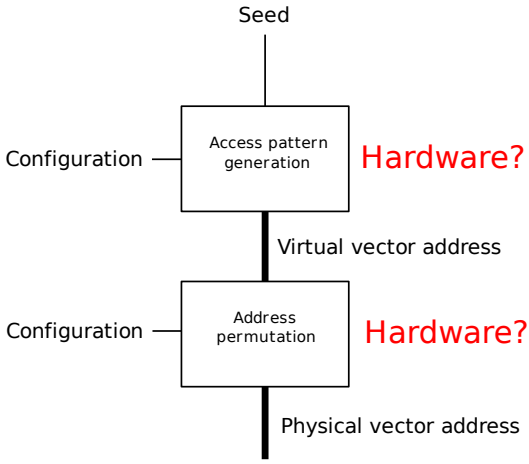

Figure 2.3. Functional overview of AGU for a parallel scratchpad memory

The hardware associated with all of these computations cannot be known without exhaustively studying the access patterns of algorithms and their associated memory permutations. In this thesis the access patterns of the FFT algorithm will be studied and their implications on the AGU will be analyzed.

Chapter 3

Hardware model

3.1 Introduction

The hardware model of the processor is roughly based on the architecture of IBM Cell. It is adapted to fit the specifications of the P3RMA concept described in the previous chapter. This model is used as a blueprint when designing the simulator framework.

The following hardware description has been given and is considered as fact in this thesis. This due to the fact that the processor is still a work in progress and the specifications are still unpublished. Therefore no references can be made at this point. There will be publications with a more detailed description coming in the near future.

3.2 Host

The host processor is a scalar DSC responsible for the entire system. Here controls for both scalar and parallel computing are handled as-well as task scheduling for scalar and parallel hardware resources. Required by the host is flexibility, scalability and the ability to handle parallel tasks efficiently. In the proposed architecture it has 8 SIMD units, used for parallel computations, connected via two independent ring-busses. Regular scalar computations are performed in the host processor.

Tasks and data to the SIMD units are transferred over a bus. Transactions are triggered by the host but are handled by the DMA connected to the bus chosen for the transfer. The following figure gives a basic overview of the important parts in the architecture and how these are connected.

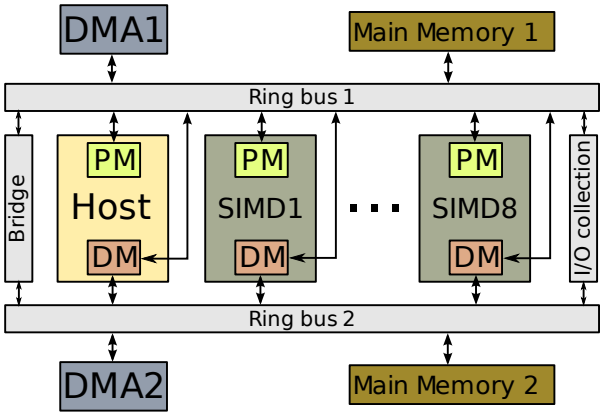

Figure 3.1. Overview of the system architecture.

The result is collected by the host from the accelerating SIMD units forming a unified result.

To improve performance the complexity of control is partitioned according to a scheme where multiple FSMs are used. The suggested partition is as follows:

- System level specific tasks such as hardware and asynchronous event handling, inter-task and inter-processor communication and OS specific tasks.
- Preparation of parallel tasks, load balancing and inter-SIMD synchronization and communications.
- Execution of top level code.

The partitioning of the control complexities gives the possibility to localize them. This is a necessity for realising the P3RMA concept.

Multiple hosts can be connected together sharing a task over multiple master-SIMD systems. One such master-SIMD system is called a cluster. Clusters are connected together via a switch forming a cluster network. A cluster network is controlled by a simple MCU.

3.3 SIMD

The SIMD unit is a multiple data processing unit being able to perform vector operations. Therefore the storage area of a SIMD unit are all designed to store vector values. The data path and computational parts of the SIMD unit is also designed to perform such operations. Operations both consists of simple RISC-like instructions as-well as function solver accelerators eg. convolution and butterfly operations.

3.3.1 Vector register

The term vector is defined as the ability of handling data and computations in parallel which is the purpose of the SIMD unit. Preliminary specification of the vector representation states that the vectors should be 8 words long where one word is 16 bits. This gives register with the total width of $8 * 16 = 128$ bits. Since this is a research project these parameters are subjects that might be changed.

The vector registers are mostly found in the vector register file but there are also some internal operand vector registers used in the ALU. The different element sizes of the vector register supported are byte, half word, word and double word.

Figure 3.2. Representations of vector registers using various vector element widths.

3.3.2 Data path

The SIMD unit supports two types of data paths, MIMO and MISO. When operating on a register value the same operation is either performed on all elements of a vector or performed in a triangular matter resulting in a single value result. This behaviour can be seen in the following figure:

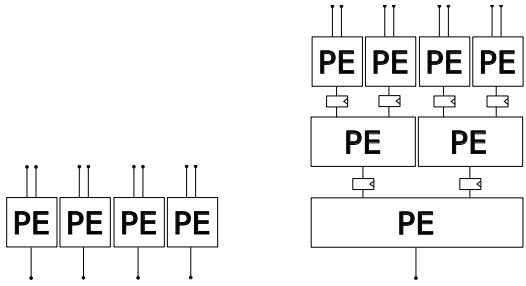

Figure 3.3. MIMO and MISO data paths.

The selection of data path is performed according to control signals generated from the instruction implementation.

Supported instructions involve basic arithmetic ones that can be found in any common processor eg. `add`, `sub`, shift operations. By default, the MIMO data path will be used. For using the MISO data path specific instructions will be used eg.

`tadd`, `tsub`. The prefix `t` stands for *triangular*. Various data reorganizing schemes are also supported eg. shuffling, packing and unpacking. These schemes are based on the MMX instruction set. Vector load and store are essential functions used to transfer data between the local store and vector register file.

The SIMD unit has different levels of memory. The only one exposed outside of the SIMD unit is the local store. It supports both SIMD and SIMT operations. Depending on the operation the pipeline depth might vary. The execution of conditional operations are still a research task and is not described here.

3.4 Ring-bus

There are two ring-busses in the system. Connected to the busses are main memories, one on each bus, the host and the SIMD units. Connected to each ring-bus is also a DMA.

The DMAs conducts the transactions over the bus it is connected to. The ring-busses can propagate data independent of each-other but have a bridge connection for exchanging data between the two main memories. Ring-bus 1 is controlled by the host and is used for data transaction since it is connected to the data memory of the host and the SIMD units. Ring-bus 2 is connected to the program memories of each connected unit. This bus can also be used for data transactions between SIMD units when cache coherence is needed.

3.5 DMA

There are two DMAs in the processor, one on each ring-bus. It is the manager of its given bus and the main memory connected to it. DMA tasks can be initialized by either the host or a SIMD unit. The host mainly distributes tasks and exchanges data via main memory and the local memory of a SIMD unit, whereas SIMD units most often initializes tasks when cache coherence is needed. Any DMA task is terminated by the DMA itself and will issue an interrupt when finished.

The outline of the DMA is as follows:

- 16 communication channels
- Two AGUs, one for data source and one for data destination
- Two clock generators, one for source memory and one for destination memory

For loading and storing data in an intelligent way, the AGU is able to generate permutation tables and send these to SIMD units. The DMA can use the permutation tables itself when loading and storing data between the main memory and the local store of a SIMD unit. For generating the correct permutation table and configuring various parameters, the host sends a transaction control table to the DMA when the transaction is issued. When a transaction is is executed all parameters according to the transaction control are forwarded to the given hardware. Eg. addresses and clock signals are supplied to the memories and the channel is

reserved for the transaction. A FIFO buffer in the DMA is used for adapting data format between the source and destination memory.

When using linking table and permutation table for data transferring using DMA, data blocks from different addresses are collected and packed into one DMA package and is sent over the bus. When reaching its destination the data is unpacked and stored into data blocks according to the permutation achieving the conflict free data access. It is for example highly desired to read and store to different memory blocks of a memory.

3.6 Memory subsystem

The memory subsystem is based on the specifications of OpenCL memory subsystem. It consists of three parts each situated on different levels in the memory hierarchy. Both hardware and software are involved when conducting the data flow. The main memories are situated in the top of the memory hierarchy. It is accessed by the host processor and the DMA connected to the same bus as the memory. The hardware responsible for the transaction holds the data scheduled to move between the main memory and data paths of given SIMD units. The software consists of instructions how data is moved to and from the hardware. It also provides control signals to the DMA and configurations for the bus. The next figure illustrates this structure in a more clear way.

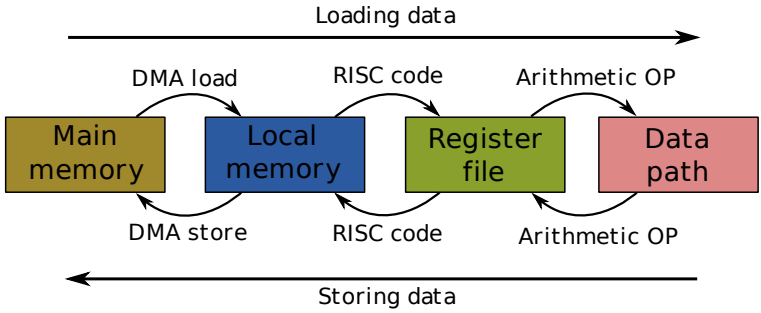

Figure 3.4. Data access flow in the memory subsystem.

The RISC code in figure 3.4 is limited to the operations used for loading and storing between the local memory and register file. Arithmetic OP is here defined as the execution of ALU operations or instructions eg. `add`. Hardware used for these transactions is the data path of the SIMD unit.

The main goal of the memory subsystem is to reduce the total number of data accesses reducing the latency cost. The ambition is to make the latency to only consist of arithmetic computing which is not far from realizable since data and addressing computations are separated. The data access also has to be conflict free and predictable. When accessing multiple data from a memory conflicts emerges when data that needs to be fetched is stored in the same memory block in the

same physical chip. In order to achieve this data has to be read and written to the memory at the same time. Also data has to be structured in such a way that multiple data access can be performed without conflicts. This is done by designing the local memory as a ping pong memory.

3.6.1 Ping pong memory

A ping pong memory is defined as a memory seen by the system as only one physical memory but in reality consists of two or more. These two or three memories is exposed to different parts of the systems via exclusive ports. The connections between port and memory can be swapped to expose a memory to a different part of the system without the system to notice any difference. Additionally, each memory consists of multiple data blocks to make storing according to a permutation table possible. For this hardware the memory consists of 8 scratchpad memory blocks. Only one block can be written at a time.

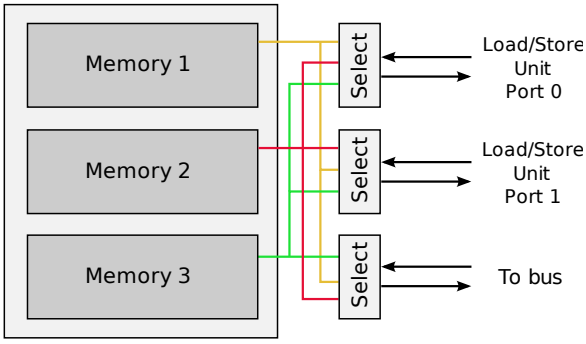

Figure 3.5. Ping pong memory model.

The ping pong memory used consists of three physical memories where one is exposed to the bus and the other two to the SIMD unit. The system itself only sees one memory but this realisation makes it possible to read and write concurrently to the memory. For certain operations it might even be possible for the SIMD unit to access all three memories.

When the memories are swapped, changing the unit it is exposed to, it neglects time spent waiting for a memory to load new values as the SIMD unit can perform computations all the time.

3.6.2 SIMD vs. SIMT execution

The memory subsystem supports two kinds of executions, SIMD and SIMT. The difference between the two is the source and destination of operands.

SIMD instructions together with DMA transactions can be run i parallel if the local vector memory is not used by the instruction. Usually SIMD instructions

only uses the vector register file as its source and destination (exceptions `load` and `store`).

SIMT instructions are iterative instructions normally executing two instructions per clock cycle. One instruction loads the data from the local vector memory and the other instruction performs the computations. The results are usually written back to the local vector memory.

Both approaches will be prepared for since both ways will be evaluated during the research of the project.

Chapter 4

Pre-study

4.1 Design philosophy

The idea is to model the hardware using a hierarchical structure of modules each representing a piece of hardware. The top level would be considered as the master holding all top level hardware controlling and allocating these dynamically. Each single hardware module could then hold its own set of lower level resources only visible to that module. This will be achieved by modeling the various resources using C++ classes to make use of the languages object oriented features. This also means that the modules would be independent and therefore more flexible to work with.

The control signals should not be a part of the hardware but rather parameters influencing these.

Instructions should be stateless and rather be given the current cycle as a parameter.

4.2 Modelling hardware with C++

Modelling hardware is not a simple task, especially in a programming language like C++ which is not designed to simulate hardware behaviour. This calls for a design strategy where typical hardware constraints and problems are taken into account. In hardware there can not exist combinatorial loops simply because timing cannot be guaranteed. Combinatorial logic is typically separated with synchronous registers holding a value until a certain point when data is propagated into the next set of combinatorial logic. For these reasons a type of register for the simulator had to be designed that can mimic the hardware behaviour in software.

The resulting register consists of two storage locations, one at input and one at output, and a clock signal.

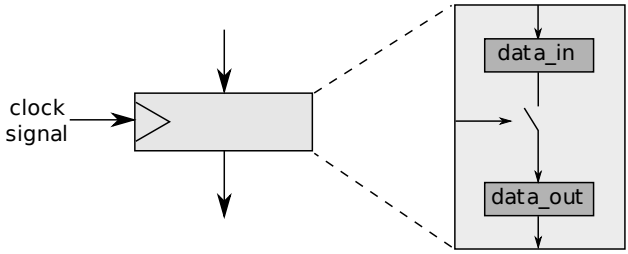

Figure 4.1. The modelled register in C++.

As can be seen in the figure the data from the input is only fed to the output when the clock signal arrives. The picture is only a software representation of the register to mimic the behaviour of a hardware one. Therefore after a value has been clocked the previous value is still stored in the `data_in` variable. If no new data is written to the register the same data is propagated yet again when the next clock signal arrives. Using this structure a synchronous data path using a homogenous clock signal can be achieved in software.

4.3 SWIG

SWIG is an tool to wrap C/C++ code exposing it to a higher level language[1]. It it distributed under BSD licences and may therefore be freely used, distributed and modified for commercial and non-commercial use. It enables developers to write low level parts demanding high performance in C/C++ and non critical parts in flexible scripting language like Python. For example one could write a high performance image processing library in C, use SWIG to expose it as a Python module and then use Python write a dynamic graphical desktop application using it.

A SWIG interface can be created independently of the C/C++ code requiring no modifications to the code base. Therefore a complete C/C++ program could be written without any dependencies on a scripting language but then later be orthogonally extended with one at a minimal cost. The executable Python script is also smaller in size compared to the equivalent code in C++. That is what the following figure shows as-well as the internal connections between the modules. The size of the boxes reflects somewhat the size of the code base of the different main-loops.

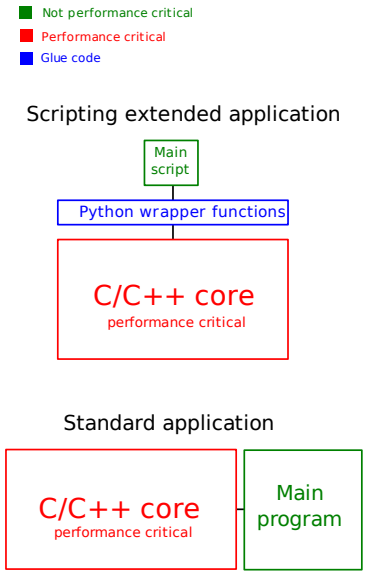

Figure 4.2. C++ main program vs. Python script using SWIG.

The benefit of exposing a C/C++ program as a module is that its behaviour can be easily customized and extended. It enables a rapid development process and debugging.

The `main()` function is replaced by a script which loads and uses the module. Other C/C++ modules could also be loaded by the scripting interpreter effectively combining several C/C++ programs into one.

Chapter 5

Simulator framework

5.1 Usages

The simulator framework will be used to model and benchmark a novel multi-core SIMD-processor system similar to IBM Cell. Instruction set, memory size and other system parameters will be profiled according to many applications specific needs. The framework should have a modular architecture facilitating easy addition of new instructions and hardware modules. There is a need to monitor the hardware state in a customizable way to enable the user to filter out the interesting information specific to their task. In FFT calculations it is desired to present the results in a complete different manner than for example matrix inversion. The effort to integrate with a graphical user interface will be greatly reduced.

The inputs to the simulator will be, a library of instruction types, assembly code using these instructions and hardware parameters configurations. Here an instruction type is defined as an assembler instruction eg. `add`. Users should be able to load an instruction type by specifying its syntax and implementation. They should also be able to define the hardware parameters configuration used during run-time.

5.2 Requirements on modelling

The simulation time is incremented in discrete steps equal to one clock cycle. In each increment all modelled hardware components will be requested to simulate their cycle. During each cycle the simulator need to read inputs, change state and finally write outputs. For any given cycle the simulation of all components shall be independent. That means all inputs have already been written in the preceding cycle. Unfortunately this makes simulating combinatorial interdependencies between components harder. An advantage of this is the easy integration of different components which is desirable in this project.

Components are modelled with the object oriented features of C++. These components are hierarchically arranged to form a complete executable hardware

model of the system.

5.3 Concurrency

A disadvantage of using C++ for hardware modelling is the lack of support for concurrency. Components must be able to execute their cycles independently of others. In this thesis concurrency functions are implemented based on synchronization of parallel hardware tasks. These functions mimics the behaviour of a clock signal.

5.4 Modelling of the SIMD processor

The SIMD processor model is divided into two parts; the modelling of the hardware state and the modelling of the instruction flow. The hardware state is represented by register contents, memory contents, program counter and other hardware capable of storing intermediate data of some kind. An instruction flow is the flow of data between hardware components which alters of the state of these components. In A.1.1 the definition of the SIMD unit class can be examined. It is stated within the header file which declarations are subject to which part of the SIMD processor model.

5.4.1 Hardware state

The hardware is modelled as a hierarchical set of C++ objects. They will all provide a method to advance their state one clock cycle. This method should be independent of the actions taken by other hardware components during that cycle. If this condition is invalid, concurrency between components cannot be guaranteed.

5.4.2 Instruction flow

The instruction flow is modeled as a state-less program manipulating the hardware state. Different instructions are gathered into an instruction library which provides an implementation of the flow of each instruction. To understand the structure of the instruction library and the instruction implementation one can study their class descriptions in sections A.1.6 and A.1.8 respectively.

5.4.3 Active instructions

Instructions currently propagating through the pipeline are active instructions. They are executed according to instructions from the instruction library. During execution of the current cycle a instruction flow originating from the instruction library is used to change the state of the hardware. The active instructions are stored in a list paired with a variable storing information about the instructions current cycle. After each clock cycle the variable is incremented by one. Active

instructions are stored in the pipeline list until their execution is complete, then they are removed.

5.5 Instruction data representation

The assembly instructions are not transformed into op-codes since no such scheme has been defined yet. But to use their textual representation in the simulator would be inefficient. It would require repeated parsing operations in the simulator main loop. To solve this the textual representation of the assembly instructions are parsed into fields of a C++ struct, specified in section A.1.7. Stored in the struct is sufficient information needed for executing the instruction. Parameters stored can be any number or all of the following: instruction type, operand types, operand values, element width and mode (signed or unsigned).

5.6 Instruction modules

Two important requirement of the project are flexibility and scalability. Therefore the need to add new instructions in a modular way was investigated early. Instruction modules are compiled separately, requiring only the SIMD unit library and header file as inputs.

The instruction modules are dynamically loaded into the simulator at run-time and can be omitted or included depending on configuration. The developer of a new instruction module only has to familiarize himself with the hardware model of the SIMD unit without knowing the intricate parts of the simulator main loop.

It is in the instruction implementation that the instruction pipeline is outlined. For convenience and to achieve a more homogenous pipeline, structure templates can be defined in header files common for all instruction implementations. As an example a basic common header file can be examined in section A.1.13 where a template pipeline for typical ALU operations is defined.

Instruction implementation binaries can be compiled on one computer and then be redistributed to other computers where simulations may be performed. There is no need to recompile the instruction implementations as long as the same SIMD unit library and header file are used. Any changes to the simulator will have an impact on instruction execution and thus recompilation is needed.

The instruction modules are stored in a dedicated instructions folder. Within this folder each instruction is represented by a folder named according to a particular naming scheme. The folder should use the same name as the assembler syntax with an additional `.instr` suffix. An instruction module has to provide two files to the simulator.

- Assembly instruction parser format used by the assembly file parser.
- Assembly instruction flow implementation used by the SIMD unit.

The format is specified in a regular text file named `format`. The implementation of the instruction flow should be an object file named `implementation`. These

two files must be present in a instruction directory with the `.instr` suffix in order for the simulator to recognize the instruction.

The `format` file is a one line plain text which must contain the assembler syntax used for identifying the instruction within assembly files. Optional are various tokens used for parsing additional parameters used by the instructions when executing eg. immediate values and register indexes. The object file `implementation` is created when compiling the source code file `implementation.cpp` which normally also resides within the instruction folder.

5.7 Vector registers

The vector registers are used for storing and loading intermediate values as they are propagated through the pipeline. Hence they are a building block defining the hardware state of the SIMD unit. Further it is the sole building block of the register file. To cope with the concurrency problem, the vector register had to be modeled in a specific manner.

The vector register was implemented using two variables, one representing the value on the input and the other on the output. This simulates the behaviour of hardware since the value of the vector register is not changed when the register is written to by C++ methods.

During the whole clock cycle the previous value stored within the register can always be accessed from the vector register. It is not until the end of the cycle, when the clock method is executed that the data stored in the output variable is overwritten by the value in the input variable. Hence mimicking the behaviour of a hardware register. The vector register class definition is found in section A.1.4.

Like the name suggests, the vector registers consists of multiple values stored as vectors in the registers. The vector elements can be of various length like figure 3.2 shows. For this reason, not only methods for loading complete vectors is implemented but methods that loads slices of the memory as-well. The slices can be retrieved with any offset and any width between one and 64 bits.

5.8 Assembly file parser

The assembly file parser does exactly what its name suggests, it transforms a textual representation of the assembly instructions into a vector of instruction data. It is implemented in C++ but is not a part of the SIMD unit hardware. Instead it is a part of the simulator top level structure with the simulator communicating with the SIMD unit hardware via the instruction library. The assembly file parser is specified in section A.1.10.

5.8.1 Assembly file syntax elements

An assembly file consists of the following elements: preprocessor directives, comments, label declarations and actual assembly instruction lines. The elements uses various tokens for identifications specified as follows.

- Preprocessor directives are preceded by the <#> character which is familiar from the C preprocessor.
- Comments are preceded by the <;> character and ends with a new line.
- Labels are preceded by the <.> character followed by the label name.

A main label (`.main`) must always be present in the assembler file to indicate the entry point of the assembly program.

Before parsing the textual representation of each assembly line the assembly file parser first assigned them addresses according to the position of the labels. This makes it possible to use label names as arguments in the assembly instructions. When parsed the label name given as arguments in the assembler file is translated into a proper address or relative offset. This is useful when using branch and sub-routine instructions.

An assembly instruction is simply the name of the instruction followed by its arguments. To parse an assembly instruction the assembly instruction parser first reads the name of the instruction. Using this name it collects the assembly format for this instruction from the instruction library and uses it to parse the rest of the line.

```
Format:
add <width> <mode> <rt> <op> <op>

Example assembler instruction:
add 16 signed r1 r2 $5
```

The format of an assembly instruction is a chain of tokens. Each token consumes a part of the line while filling out the fields of the instruction data. Each assembler file will result in a vector of instruction data which later can be loaded into the program memory of a given SIMD unit.

5.9 Configurations

The configuration specifies the outline of the SIMD unit. A large variety of parameters can be set, such as word length, register file size, register width etc. This allows the user to change the structure of the simulator fast without much effort. For the sake of the research this is valuable for testing and evaluation to help decide on future parameters. The configuration is represented by a set of variables residing in the SIMD unit and is used when creating the hardware model of the system.

5.10 Resulting model of a SIMD processor

5.10.1 Standalone SIMD unit

Taking all above parts into account one SIMD unit will result in a model pictured in figure 5.1.

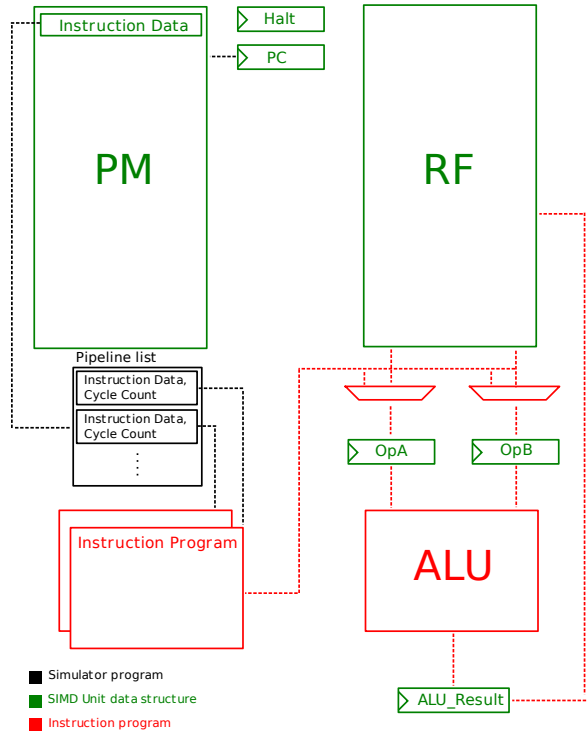

Figure 5.1. A model of the SIMD unit.

Green parts indicates a hardware module and is described above in section 5.4.1 as modules where changes in the hardware state takes place. The instruction flow described earlier in section 5.4.2 is represented as the red parts. The black box is a simulator object keeping track of active instructions which are currently in the processor pipeline. Within resides instruction data of active instructions currently in the pipeline, which are fetched from the program memory. The register `halt` is used to signal that the SIMD unit has completed its execution. Register `PC` informs which program is next in line to start its execution.

5.10.2 SIMD unit connected to a memory subsystem

To provide the SIMD unit with a local memory, additional hardware was attached to the SIMD unit. The memory subsystem is a 3rd part module developed by Jian Wang[1]. This testing the flexibility of the simulator, which determines that the design philosophy holds. It also adds the important features of inter-SIMD communication using the ring-busses. This obviously allows the simulator to consist of multiple SIMD units enabling the parallel features. The modified model can be seen in figure 5.2.

[1] jianw@isy.liu.se

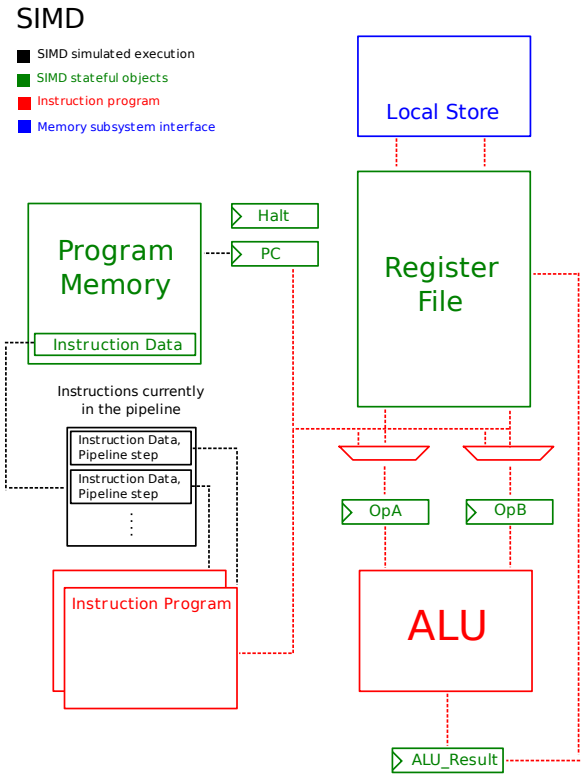

Figure 5.2. SIMD unit model connected to a memory subsystem.

From the previous figure 3.1 other important hardware can be seen connected to the ring-bus. These additional resources are all integrated in the memory subsystem. Contents of the memory subsystem includes local store, DMA, AGU, main memory and the bus itself. The memory subsystem handled all communication with resources surrounding the SIMU unit creating a more formal model of the master-SIMD architecture.

5.11 Instruction set

This section describes the instruction set used during the development of the simulator. Below is a table with the most useful instructions in the set. Detailed descriptions of each instruction can be found in appendix C. The purpose of the instruction set used is to verify the functionality of the modules added to the simulator. For validating the different registers within the system, especially those within the register file, the instructions `set` and `sete` were used. The arithmetic functions `add` and `sub` verifies the instruction flow and changes in the hardware states. Also correctness of the vector computations performed is verified together

`add`	Vector addition.
`sub`	Vector subtraction.
`nop`	Dummy instruction. Performs no operation.
`halt`	Set the halt register to true. Indicator to the simulator to stop simulation.
`set`	Sets one value to **all** elements of a vector register in the register file.
`sete`	Sets one value to **one** element of a vector register in the register file.
`load`	Load a vector value from the local store to the register file.
`store`	Store a vector value from the register file to the local store.
`inX`	Read the content on the given address from ring bus X to the register file.
`outX`	Send a value from the register file on the given address on ring bus X.
`portswap`	Swap the memory ports exposed to the SIMD unit.

Table 5.1. Basic instruction set used for validation of the simulator.

with pipeline, instruction specific parts such as the pipeline list and instruction library.

Since handling of data hazards such as read-after-write is not taken cared of in the simulator the `nop` instruction was greatly used. It often served as a delay to postpone the start of the execution for neighbouring instructions where hazards could occur.

For validating the memory subsystem multiple tests had to be performed. Primarily used for the validation of the local vector memory were the instructions `load` and `store`. To test the bus functionality an inter-SIMD communication was issued using two instructions, `inX` and `outX`.

All instructions defined are none-destructive meaning that no used operand register have to be overwritten when computation is finished, unless it is desirable.

5.12 SWIG wrapper

The SWIG wrapper is created using a Python setup script and SWIG. The source code for the setup script can be found in section A.2.1. Using specifications from the interface file, found in A.2.2, SWIG creates a wrapper around the simulator which functions as a Python-C++ interface. When executing the setup script, the newly created wrapper is used to create a loadable Python specific library containing methods for communicating with the C++ modules from Python.

The functions interfaced to Python can be seen in section A.1.11 and these are described in more detail in the user manual table 6.1.

Chapter 6

User manual

There will only be a detailed view on how to compile and execute the simulator using a Unix-like system, eg. Linux, Unix and Mac OSX.

6.1 Preparing the simulator

6.1.1 Prerequisites

Before one is able to use the simulator some things needs to be in place. Make sure all following requirements are met before you continue.

- A C++ compiler, eg. g++
- SWIG[1]
- The simulator source code.
- Python 2.X

Since the hardware modules or cores of the simulator are written in C++ a compiler for that programming language is obviously needed. SWIG is used for creating an API between the modules and, in this case, the scripting language Python. Since the simulator modules are accessed via Python scripts a program for executing these scripts is also needed[1].

6.1.2 Compiling the source code

When all prerequisites are met the simulator can be compiled. The whole simulator is divided into three parts; the hardware modules, the set of instructions and a Python script for executing the simulator. The simulator also provides an executable programmed in C++. If this executable is chosen, the Python/SWIG specific parts can be ignored.

[1] Python's official homepage, http://www.python.org/

The order of compilation is important due to dependencies between the different parts. The hardware modules has to be compiled first to create the fundamental libraries used by the other parts. This is done preferably from a terminal window. To start the compilation make sure you're in the simulator root folder and then execute the following command:

```
make
```

After a successful compile it is possible to compile the remaining parts of the simulator. The instruction set is compiled by executing the following command:

```
make -C instructions/
```

Now the simulator is ready to run using the C++ binary. If one wants to run the simulator using Python scripts, the necessary Python library has to be built. Run the following command to create the library:

```
make pymodule
```

When successfully reaching this point the simulator has completed its compilation steps. All that remains is an assembly program for the simulator to run. Provided are several test programs used for validating the simulator behaviour is as expected. These assembler files are located in the `asm_test_cases` folder. Provided along with the assembler files are correct results for the programs used to validate the simulator behaviour.

6.1.3 Loadable instruction modules

The simulator program only consists of the hardware modules and a reference to an instruction library. The instructions are dynamically loaded into the common library by the assembly file parser during run-time. The instruction directory must be specified before the assembler program is parsed. It is possible to have different instruction directories for different test purposes and these instructions are loaded when the simulator is executed.

In the instruction set directory each instruction definition is stored in individual directories which should use the following naming scheme to be recognized by the parser: `<instr_name>.instr` eg. `add.instr`. Within each folder there are two required files, `format` and `implementation.cpp`. The `format` provides the parser with tokens on how assembly lines should be parsed and interpreted. The `implementation.cpp` contains the implementation of the instruction. It describes the task to be performed in each and every cycle of the instruction.

Example

This example demonstrates how to include an instruction that performs a typical add operation. For convenience this instruction will merely be called `add`. First of all the `format` file will be defined. In this example the following format has been proposed:

```
add <width> <mode> <rt> <op> <op>
```

First segment should be the name of the instruction. This name will be used when the instruction is called upon in an assembly file. The name should be the same as the name of the folder without the suffix `.instr`. The other five segments indicates what kind of parameter to expect at the corresponding positions. These tokens are used when parsing an instruction and each have specific instructions on how to parse the value occurring on the token positions. The token instructions are predefined in the assembly file parser. Implemented tokens and their parsing schemes are listed below:

- <*width*> Element width in bits.
- <*mode*> Interpret operands as unsigned or signed.
- <*rt*> Register id. *rX*
- <*imed*> Immediate. *$X*
- <*op*> Register or immediate.
- <*rel_addr*> Immediate relative address or label to jump to.

Secondly the implementation itself needs to be defined. The implementations is a functions that inherits its properties from a template seen in section A.1.8 whereas a template of an instruction implementation can bee seen in section A.1.9. The instruction tasks should be defined in the `switch` statement in the `execute` functions. Each `case` statement represents one cycle and the numbering indicates the order of execution of stage in the pipeline. In the class constructor the depth of the pipeline is defined which must be set to the amount of stages the instruction contains.

6.1.4 The configuration

The simulator has a number of configurable parameters which can be specified by the user. The parameters abel to be configured are found in the `config.h` file where also a set of default values are defined. These values can be reset during runtime right before simulation starts. The configurable parameters are the following:

- `WORD_SIZE` Size of a word in bits.
- `VECTOR_SIZE` Size of vector in words.
- `GUARD_SIZE` Number of guard bits per word.
- `RF_SIZE` Size of the register file in vectors.
- `PM_SIZE` Size of the program memory in instructions.
- `LM_SIZE` Size of the local memory in vectors.
- `SIMD_UNIT_COUNT` Amount of SIMD units in the simulator.

6.2 Running the simulator

The simulator can be executed in two ways; via native binary executable or via a Python script.

Execute via native binary executable:

```
./simulator
```

Execute via Python script:

```
python simulator.py
```

The difference between these two are that the binary executable is implemented in C++ and therefore has to be recompiled when changed. The Python script does not require any recompilation when changed and is therefore meant for various debugging purposes.

6.2.1 Example

Now a simple program will demonstrate how an executable might look like. The same example will be expressed in both C++ and Python scripting language.

Python script:

```
# Declare the simulator
sim = Simulator()

# Set configuration and update
sim.config.WORD_SIZE        = 16
sim.config.VECTOR_SIZE      = 8
sim.config.GUARD_SIZE       = 0
sim.config.RF_SIZE          = 16
sim.config.PM_SIZE          = 1024
sim.config.LM_SIZE          = 1024
sim.config.SIMD_UNIT_COUNT  = 1
sim.config.update()

# Build the hardware module
sim.build()

# Load the instructions to the instruction library
sim.loadInstructions("instructions/")

# Load program and assign it to a SIMD unit
sim.loadProgram("asm_files/simple.asm", "simple")
sim.setProgram(0, "simple")

# Start simulation, runs until SIMD unit 0 is halted.
sim.simUntilHalt(0)

# Get register values from the register file and print them
xmlStr = sim.getXMLrepr()
printRegisterFile(0, xmlStr)
```

C++ main-loop:

```
int main(int argc, char** argv) {
  // Declare the simulator
  Simulator sim;

  // Set configuration and update
  sim.config.WORD_SIZE        = 16;
  sim.config.VECTOR_SIZE      = 8;
  sim.config.GUARD_SIZE       = 0;
  sim.config.RF_SIZE          = 16;
  sim.config.PM_SIZE          = 1024;
  sim.config.LM_SIZE          = 1024;
  sim.config.SIMD_UNIT_COUNT  = 1;
  sim.update();

  // Build the hardware module
  sim.build();

  try {
    // Load the instructions to the instruction library
    sim.loadInstructions("instructions/");

    // Load program and assign it to a SIMD unit
    sim.loadProgram("asm_files/simple.asm", "simple");
    sim.setProgram(0, "simple");

    // Start simulation, runs until SIMD unit 0 is halted.
    sim.simUntilHalt(0);

    // Get register values from the register file and print them
    printf("%s\n", sim.getXMLrepr().c_str());
  }

  // Error handling
  catch (std::string& errorMsg) {
    printf("Error: %s\n", errorMsg.c_str());
  }
  return (EXIT_SUCCESS);
}
```

The two representations are very similar but the simulator specific functions used are in fact the same.

This very simple executable first loads the configurations and builds the hardware using those parameters. It then loads all instructions residing within the directory `instructions/` into the instruction library. Afterwards the assembler files can be parsed using the recently loaded instruction library to arrange and store the assembler file into an vector of instruction data.

In this example the file `asm_files/simple.asm` is parsed and stored into the internal structure which is given the name `simple`. It is possible to parse many assembler files and giving them different names making it easy to switch programs in-between simulations. Parsed assembler programs can then be loaded into the program memory of SIMD units declared in the simulator. In this example the program `simple` is loaded into SIMD unit 0. Finally simulation can start and when completed, the register file of SIMD unit 0 is printed.

There are numbers of different methods provided which can manipulate the SIMD unit further. A list of existing methods are described in the following table: Complete definitions of these methods can be found in section A.1.11. To further

build	Builds the hardware model according to the configuration parameters.
loadInstructions	Loads all instruciton from within a directory.
loadInstruction	Load one instruction.
loadProgram	Loads an asm program into the simulator and assigns it a name.
loadInlineProgram	Loads an inline asm program.
loadLocalStore	Load values to the local vector memory of a given SIMD unit and memory port.
readLocalStore	Read the content of the local memory with address, SIMD unit and memory port as parameters.
loadRegisterFile	Load values to a register in the register file.
listPrograms	Get a list of available programs.
setProgram	Sets the program of a SIMD unit.
getXMLrepr	Get an XML representation of the hardware model.
simUntilHalt	Simulate until a particular SIMD unit halts.
simCycles	Simulate a given number of cycles.

Table 6.1. List of simulator specific methods for accessing the simulator hardware.

extend this list of methods, functions can be manually added in the `simulator.cpp` and `simulator.h` files. How this is done is not shown in this manual.

During simulation various log messages will be printed. Instructions found in the instruction directory specified and information about the assembler files parsed will be printed as default. If used, information about the DMA will also be printed.

After simulation is complete the number of cycles simulated as-well as the register file will be printed. All of the log messages and various print outs can be removed, added, rearranged etc. by the user in any way that fits the application. This is fairy easy when simulating using Python scripts.

6.2.2 Creating assembler files

All assembler files must contain a main label (`.main`) which denotes the starting point of the execution. Preferable in each assembly program is the `halt` instruction if it is desired to at any point stop the simulation. The simulator supports registers and immediate values as operands. Immediate values can be either decimal or hexadecimal.

Example

Now follows a short example that illustrates how the simulator is executed using a simple assembler program.

```
.main
; This is a comment
set 16 r0 $10
nop
nop
add 16 signed r1 r0 $0xf
nop
nop
halt
```

This short program will first set register 0 in the register file with the value 10 to all register elements. Here the elements are defined to have a width of 16. If the configuration from previous examples is used then register 0 will now store a vector with 8 elements, all being 10.

The next useful instruction will perform a vector addition adding the immediate constant $0xf$ (15) to all 8 vector elements in register 0. The result is afterwards stored into register 1. The nop instructions which are present in this example is needed due to data hazards which are not considered. Finally the simulator is halted with the instruction halt. When the simulator is halted the content of the register file is printed:

```
SIMD 0
R00 000a000a000a000a000a000a000a000a
R01 00190019001900190019001900190019
```

To summarize what can be seen in the example program:

- rX denotes register number X in the register file.
- $X denotes a immediate constant with value X in the given base (default is decimal, $0x$ is hexadecimal).
- The character ; signal that the line is a comment.
- The character . denotes a label used for branching. The label `.main` must always be present in any program.
- Use nop operations when you suspect read-after-write or write-after-write hazards.
- Stop simulation using the halt instruction.

6.2.3 Example on simulator customization using Python

Because the simulator API also is written in Python it can easily be modified to fit the users needs. For example, the register file and the local vector memory can be printed in a way suitable for a particular application. When working with FFT, it might be desirable to examine result, intermediate data etc. in a more

readable way than examine vectors stored in the register file directly. Also the way of viewing these values might be completely different depending on the application being evaluated in the simulator. This can easily be changed using Python without recompiling neither the simulator modules nor the instruction set.

When changing the appearance of the simulator the provided `simulator.py` file would be worth looking into. Another way is to create a new Python script file and start from the simple example in section 6.2.1 extending it using functions from table 6.1. Using the various functions provided the user is able to read and store values from and to registers and local vector memory arbitrary in a way that suits the application. This can be performed both before and after the simulation.

Example

To demonstrate the benefits of using this approach is an example that calculates an 8-point FFT using values which are pre-stored into the local vector memory and prints the result in a more readable fashion. This example will show very limited source code of the Python script used since the focus is on what can be achieved easily from a customization point-of-view when using Python for simulating. It will also not describe how the FFT was performed in detail.

First lets study the input values used:

```
# Decimal representation
x(0): 0.0462 + 0.0971i
x(1): 0.8235 + 0.6948i
x(2): 0.3171 + 0.9502i
x(3): 0.0344 + 0.4387i
x(4): 0.3816 + 0.7655i
x(5): 0.7952 + 0.1869i
x(6): 0.4898 + 0.4456i
x(7): 0.6463 + 0.7094i

# Vector Representation
39093eb279a0289761fc30d80c6e05ea
5ace52ba3827046717ec65c958ef6968
```

The values in the decimal representation is randomly generated with Python. The values in the vector representation is a transformed version of the decimal one using various Python functions developed for this purpose to be stored directly in the local vector memory of a SIMD unit.

```
// sim.loadLocalStore(SIMDUnit, MemoryAddress, Data, MemoryPort)

sim.loadLocalStore(0, 0, '39093eb279a0289761fc30d80c6e05ea', 0)
sim.loadLocalStore(0, 1, '5ace52ba3827046717ec65c958ef6968', 0)
```

Using the commands above the data is stored in the local vector memory and is ready to be used when simulation starts. The procedure is the same for loading other prerequisite data, like the twiddle factors.

After simulation it might be desired to study the results. From this particular simulation, these were the resulting vectors:

```
Local Vector Memory
0x0000 f056ee00e876f5ac077a0ca2449a3889
0x0001 f4c0f71e067afe32e8a8f8c803a8eef7
```

As can be seen this kind of representation is not very suitable for debugging since it is hard to see what the values are. Instead the reversed operations performed when transforming the input values can now be made to get a more clear representation.

```
// sim.readLocalStore(SIMDUnit, MemoryAddress, MemoryPort), returns
     a std::string

result = sim.readLocalStore(0, 0, 0)
result = result + sim.readLocalStore(0, 1, 0)
```

Using the command above the result is fetched from the local vector memory and is now stored in the variable `result`. Finally the resulting vector values are converted into decimal representation and printed in the following way:

```
FFT result:
X(0): 3.533447265625  + 4.28759765625i
X(1): 0.78955078125   + 0.46728515625i
X(2): -0.6455078125   - 1.47119140625i
X(3): -1.125          - 0.97900390625i
X(4): -1.064697265625 + 0.228515625i
X(5): -0.451171875    - 1.458984375i
X(6): -0.11279296875  + 0.40478515625i
X(7): -0.55517578125  - 0.703125i
```

This is, as stated before, all done without recompiling the simulator. It makes the simulator very flexible as a team of developers can use the same simulator modules and set of instructions while they have individual Python scripts that preloads the simulator, runs different assembler files and prints the result differently.

Chapter 7

Case study: FFT

7.1 Radix-2 FFT algorithm

The radix-2 FFT (Fast Fourier Transform) algorithm is a faster way of computing the DFT (Discrete Fourier Transform) based on the divide and conquer principle. It divides an N-point DFT into two $\frac{N}{2}$-point DFTs. One for the the odd and one for the even data. The data points of the odd DFT are multiplied with the so called twiddle factors $W_N^k = e^{\frac{2\pi ik}{N}}$ and added to the data points of the even DFT. This yields the following recursive formula.

$$DFT_N[x(n)](k) = DFT_{\frac{N}{2}}[x(2n)](k) + W_N^k * DFT_{\frac{N}{2}}[2n+1](k)$$

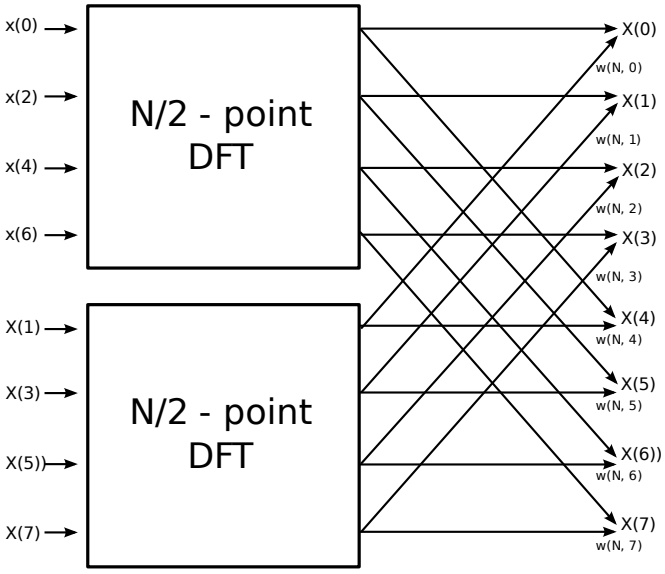

Figure 7.1. Radix-2 divide and conquer

7.1.1 Radix-2 butterfly

The operation which combines the odd and even DFT's are most commonly called butterflies due to their appearance. A butterfly is a three input - two output operation taking two complex data X_0, X_1, one complex twiddle factor W_N^k yielding two complex data Y_0, Y_1. Its function is as follows:

$$\begin{aligned} Y_0 &= X_0 + W_N^k X_1 \\ Y_1 &= X_0 + W_N^{\frac{N}{2}+k} X_1 \end{aligned}$$

But we have $W_N^{\frac{N}{2}+k} = -W_N^k$ which simplifies the equations.

$$\begin{aligned} Y_0 &= X_0 + W_N^k X_1 \\ Y_1 &= X_0 - W_N^k X_1 \end{aligned}$$

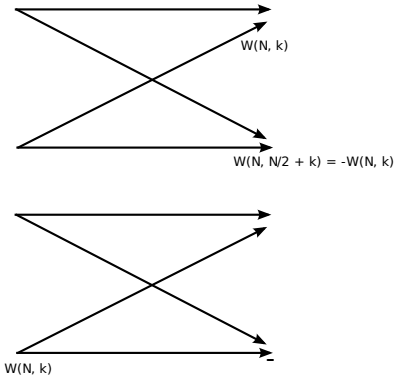

Figure 7.2. Butterfly equivalence

The computational cost as well as the memory accesses of a radix-2 butterfly can be easily tabulated.

Complex operations
Loading two complex numbers
Loading one complex twiddle factor
Computing one complex multiplication
Computing two complex additions
Storing two complex numbers

Table 7.1. Radix-2 butterfly operation counts

It is assumed that one complex number is represented as two 16-bit words, one each for the real and imaginary parts. The memory accesses then amounts to 6 load word operations and 4 store word operations. A complex addition consists of two real additions and a complex multiplication consists of four real multiplications. The computational cost is therefore 6 word additions and 4 word multiplications.

Load word	6
Store word	4
Multiply words	4
Add words	6

Table 7.2. Radix-2 butterfly word operations

7.1.2 Radix-2 signal flow graph

By recursively applying the split presented in Figure 7.1 one can observe a structure where the input data appears in bit reversed order and pass through $log_2(N)$ butterfly stages. The space between the inputs of the butterflies are doubled each stage.

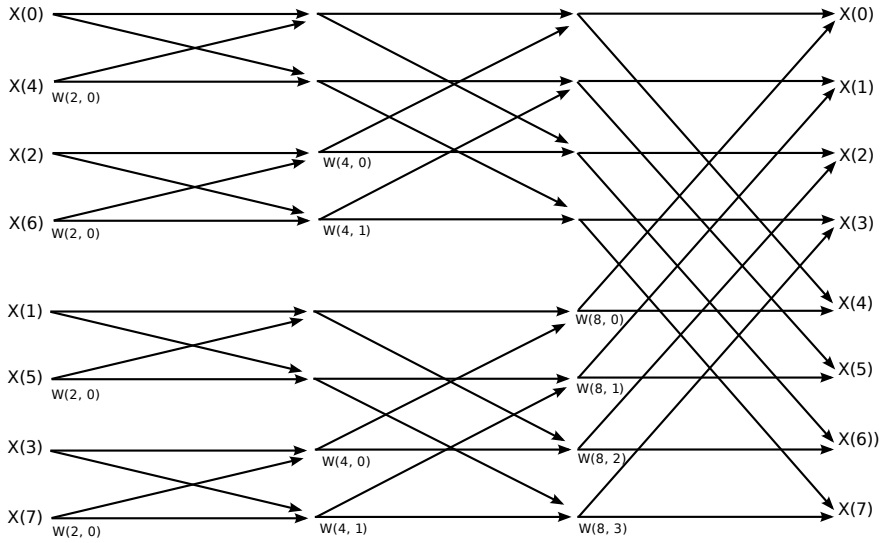

Figure 7.3. Radix-2 FFT structure

7.1.3 Memory access pattern

Data

As seen in figure 7.1.2 the memory access pattern is dependent on the FFT-layer. The data points accessed are always of the form: (where l is the FFT-layer)

$$\begin{pmatrix} i \\ \\ i + 2^l \end{pmatrix}$$

Enumerating the butterflies in each FFT-layer, l, as $i = 0 \ldots \frac{N}{2}$ the data points to be access in parallel are

$$\begin{pmatrix} i + 2^l \left\lfloor \frac{i}{2^l} \right\rfloor \\ \\ i + 2^l \left\lfloor \frac{i}{2^l} \right\rfloor + 2^l \end{pmatrix}$$

Twiddle factors

Enumerating the butterflies in each FFT-layer, l, as $i = 0 \dots \frac{N}{2}$ the twiddle factor to be accessed is:

$$W(i) = W_N^k = W_{2^l}^{i-2^l \lfloor \frac{i}{2^l} \rfloor}$$

Using the fact that

$$W_{\frac{N}{2}}^k = W_N^{2k}$$

we arrive at

$$W(i) = W_{2^l}^{i-2^l \lfloor \frac{i}{2^l} \rfloor} = W_N^{\frac{N}{2^l}(i-2^l \lfloor \frac{i}{2^l} \rfloor)}$$

7.2 Hardware constraints

7.2.1 Datapath

The datapath consists of 8 multipliers followed by 12 adders. The adders can be configured in parallell or in different kind of triagonal topologies.

7.2.2 Memory

The memory subsystem has three available ports. Each memory port is an eight-way memory with each bank being 16 bits wide. One memory port provides simultaneous access to vectors of 16 * 8 = 128 bits. The memory can either be accessed directly with the same adress to each bank or via a permutation table which provides a local offset for each bank address to the global adress. The permutation also reorder to outputs / inputs to each bank. This is illustrated in Figure 7.4

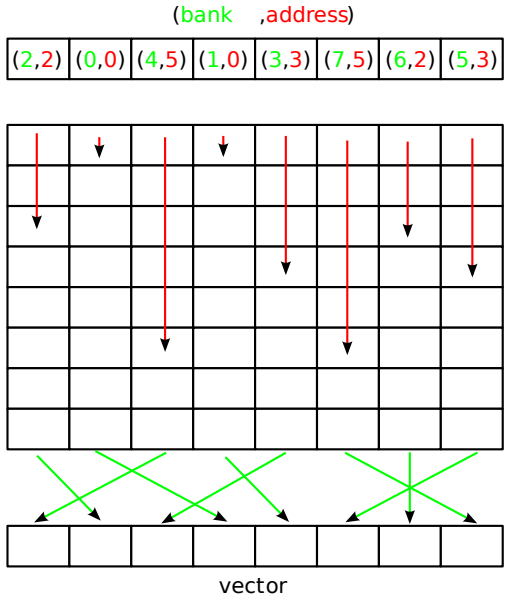

Figure 7.4. Vector memory with permutation enabled

The three memory ports can support the loading of 16 16-bit words and the storing of 8 16-bit complex data. This is enough to support two radix-2 butterflies every cycle as seen in table 7.1.1 on page 45.

7.3 Solution

The previous section about hardware constraints exposed the opportunity to schedule an instruction performing two radix-2 butterflies with an execution time of one clock cycle.

This instruction fetches one vector with four complex data and one half vector with two complex twiddle factors, computes two radix-2 butterflies and stores back the result.

One memory port is used to load data, one memory port is used to store data and one is used to load twiddle factors. The load and store port are switched between each layer of the FFT.

The complexities of addressing are offloaded to the AGU and the permutation table.

7.3.1 Data permutation for conflict free parallel access

From the 7.1.3 section we learned that the data access pattern of the radix-2 FFT has the following form:

$$\begin{pmatrix} i + 2^l \left\lfloor \frac{i}{2^l} \right\rfloor \\ i + 2^l \left\lfloor \frac{i}{2^l} \right\rfloor + 2^l \end{pmatrix}$$

From the P3RMA concept chapter we learned that in order to access these in parallel the memory space has to be permutated with a bijective permutation function $P(i) \rightarrow j$ so that

$$P\left(i + 2^l \left\lfloor \frac{i}{2^l} \right\rfloor\right) \not\equiv P\left(i + 2^l \left\lfloor \frac{i}{2^l} \right\rfloor + 2^l\right) \mod 4 \tag{7.1}$$

The mapping $P_l(x) \rightarrow x >> l$ would satisfy (7.1) for layer l since

$$\begin{aligned} P_l\left(i + 2^l \left\lfloor \frac{i}{2^l} \right\rfloor + 2^l\right) = \\ P_l\left(i + 2^l \left\lfloor \frac{i}{2^l} \right\rfloor\right) + P_l\left(2^l\right) = \\ P_l\left(i + 2^l \left\lfloor \frac{i}{2^l} \right\rfloor\right) + 1 \not\equiv P_l\left(i + 2^l \left\lfloor \frac{i}{2^l} \right\rfloor\right) \mod 4 \end{aligned} \tag{7.2}$$

The problem is that the function not bijective and therefore cannot be used. Fortunately this permutation can be modified to achieve bijectivity while still keeping the property of equation 7.1 by replacing the logical right shift by cyclic right shift. The permutation for each layer l then becomes:

$$P_l(x) = x >>_{cyc} l \tag{7.3}$$

A very interesting property of the permutations P_l is their recursive property ($A \circ B(x) = A(B(x))$):

$$P_{l+1}(x) = P_l(x) >>_{cyc} 1 = P_1 \circ P_l(x) \tag{7.4}$$

This very nice property means that there is only need for one permutation

$$P_\Delta(x) = P_1(x) = x >>_{cyc} 1 \tag{7.5}$$

Using this differential permutation data can be loaded from consecutive addresses:

$$\begin{pmatrix} 2i \\ 2i+1 \end{pmatrix} \tag{7.6}$$

and stored to

$$\begin{pmatrix} P_\Delta(2i) \\ P_\Delta(2i+1) \end{pmatrix} = \begin{pmatrix} i \\ i + \frac{N}{2} \end{pmatrix} \tag{7.7}$$

The observant reader probably noticed that the storage pattern found in (7.7) is not conflict free since:

$$i + \frac{N}{2} \equiv i \mod 4 \tag{7.8}$$

Fortunately this pattern can be made conflict free by applying a simple rotation to the data vectors in the lower one half of the parallel scratchpad memory. This permutation is denoted $P_{\frac{N}{2}}$ This is illustrated by the following table:

	Bank 0	Bank 1	Bank 2	Bank 3
$i < \frac{N}{2}$	$4i+0$	$4i+1$	$4i+2$	$4i+3$
$i \geq \frac{N}{2}$	$4i+2$	$4i+3$	$4i+0$	$4i+1$

Table 7.3. Contents of the high part of the parallel vector memory has a simple permutation

As mentioned initially in this section the goal was to do two radix-2 butterflies simultaneously. The following load pattern would be used for the i-th double radix-2 butterfly:

$$L = P_{\frac{N}{2}} \begin{pmatrix} 4i \\ 4i+1 \\ 4i+2 \\ 4i+3 \end{pmatrix} \tag{7.9}$$

And the corresponding store pattern:

$$\begin{aligned} S = P_{\frac{N}{2}} \circ P_{\Delta} \circ P_{\frac{N}{2}}^{-1}(L) = \\ P_{\frac{N}{2}} \circ P_{\Delta} \circ P_{\frac{N}{2}}^{-1} \circ P_{\frac{N}{2}} \begin{pmatrix} 4i \\ 4i+1 \\ 4i+2 \\ 4i+3 \end{pmatrix} = \\ P_{\frac{N}{2}} \circ P_{\Delta} \begin{pmatrix} 4i \\ 4i+1 \\ 4i+2 \\ 4i+3 \end{pmatrix} = \\ P_{\frac{N}{2}} \begin{pmatrix} 2i \\ 2i+\frac{N}{2} \\ 2i+1 \\ 2i+1+\frac{N}{2} \end{pmatrix} \end{aligned} \tag{7.10}$$

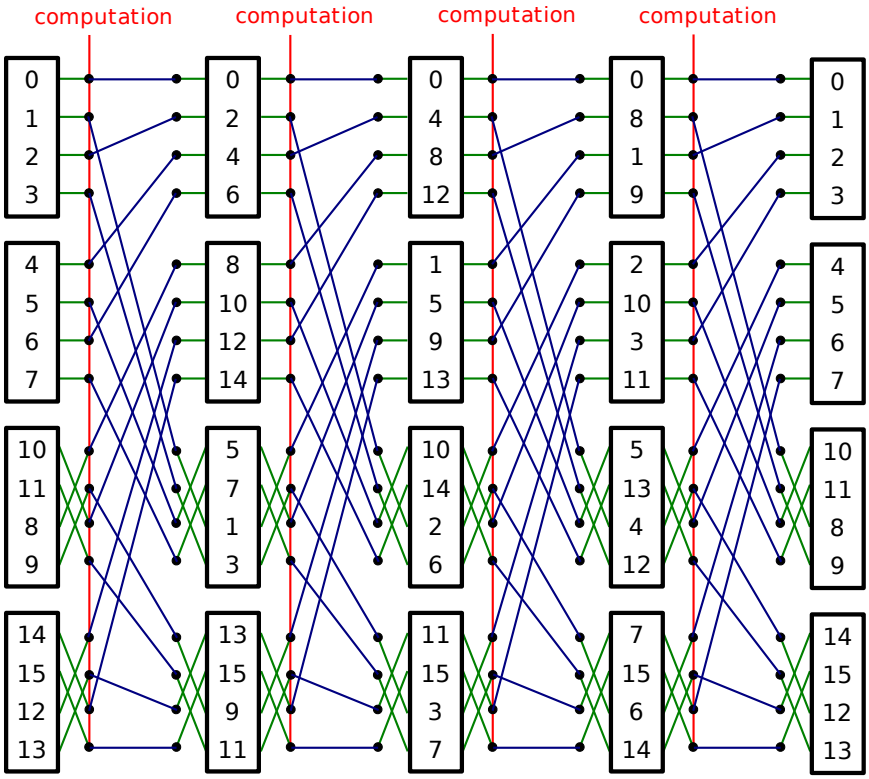

Figure 7.5. Load store patterns. $P_{\frac{N}{2}}$ is drawn with green color, P_{Δ} i drawn with blue color

By studying L and S carefully one can observe that they have two variants due to the influence of $P_{\frac{N}{2}}$.

	L		S
$i < \frac{N}{8}$	$\begin{pmatrix} 4i \\ 4i+1 \\ 4i+2 \\ 4i+3 \end{pmatrix}$	$i \equiv 0 \mod 2$	$\begin{pmatrix} 2i \\ 2i+\frac{N}{2}+2 \\ 2i+1 \\ 2i+1+\frac{N}{2}+2 \end{pmatrix}$
$i \geq \frac{N}{8}$	$\begin{pmatrix} 4i+2 \\ 4i+3 \\ 4i \\ 4i+1 \end{pmatrix}$	$i \equiv 1 \mod 2$	$\begin{pmatrix} 2i \\ 2i+\frac{N}{2}-2 \\ 2i+1 \\ 2i+1+\frac{N}{2}-2 \end{pmatrix}$

Table 7.4. Load and store patterns

As seen from table 7.3.1 there are four ways to load and store data for a double radix-2 butterfly. There are two ways to load data depending on if the index of the radix-2 butterfly belongs to the first or second half and there are two ways to store data depending on if the index is odd or even.

7.3.2 Twiddle factor pattern

In the previous section the data access patterns were discussed but the twiddle factors were ignored. Data is loaded like this:

$$L = P_{\frac{N}{2}} \begin{pmatrix} 4i \\ 4i+1 \\ 4i+2 \\ 4i+3 \end{pmatrix}$$

Two radix-2 butterflies are performed in parallel on this data, one on the first two complex values and one on the second two complex values. Which twiddle factor should be fetched for these butterflies?

Lets look at two data used for a radix-2 butterfly in layer l. They are fetched from $2i, 2i+1$. As shown in the previous section data is permutated by $P_{\frac{N}{2}} \circ P_l$ in layer l. Lets ignore $P_{\frac{N}{2}}$ for now since it wont change the relation between values of the form $2i, 2i+1$ and assume that data is permutated by just $P_l(x) = x >>_{cyc} l$. By applying the inverse $P_l^{-1}(x) = x <<_{cyc} l$ it is possible to find the indexes mapped to $2i, 2i+1$ and also their associated twiddle factor:

$$P_l^{-1} \begin{pmatrix} 2i \\ 2i+1 \end{pmatrix} \sim W_N^{2^{L-l-1}\lfloor \frac{i}{2^{L-l-1}} \rfloor} \tag{7.11}$$

Its also noted that

$$W_N^{k+\frac{N}{4}} = e^{-2\pi i \frac{k+\frac{N}{4}}{N}} = e^{-2\pi i \frac{k}{N}} e^{-\frac{\pi}{2} i} = -iW_N^k$$

A multiplication by $-i$ can easily be done by supplying different inputs and control signals to the adders in the datapath. This reduces the storage requirement for the twiddle factors by 50%.

Butterfly index	Layer 0	Layer 1	Layer 2	Layer 3
0	W^0	W^0	W^0	W^0
1	W^0	W^0	W^0	W^1
2	W^0	W^0	W^2	W^2
3	W^0	W^0	W^2	W^3
4	W^0	$-iW^0$	$-iW^0$	$-iW^0$
5	W^0	$-iW^0$	$-iW^0$	$-iW^1$
6	W^0	$-iW^0$	$-iW^2$	$-iW^2$
7	W^0	$-iW^0$	$-iW^2$	$-iW^3$

Table 7.5. Twiddle factors used for the layers of an $N = 16$ butterfly

7.4 AGU for FFT addressing

In table (7.3.1) from before four different kinds of accesses where characterized. Lets write them in the (S, a) form and analyze their implications on the AGU.

	L		S
$i < \frac{N}{8}$	$(0,i)(1,i)(2,i)(3,i)$	$i \equiv 0 \mod 2$	$(0,\frac{i}{2}),(2,\frac{i}{2}),(1,\frac{i}{2}+\frac{N}{8}),(3,\frac{i}{2}+\frac{N}{8})$
$i \geq \frac{N}{8}$	$(2,i)(3,i)(0,i)(1,i)$	$i \equiv 1 \mod 2$	$(1,\frac{i}{2}+\frac{N}{8}),(3,\frac{i}{2}+\frac{N}{8}),(0,\frac{i}{2}),(2,\frac{i}{2}),$

Table 7.6. Load and store patterns

The first observation is that the load operation does not really require any special address pattern. The same address should be supplied to each bank. The reordering of the data when loading is not necessary since it does not matter if the first two data goes to the first butterfly or the second as long as the right coefficient is supplied. The output of the two butterflies can be permutated while storing instead.

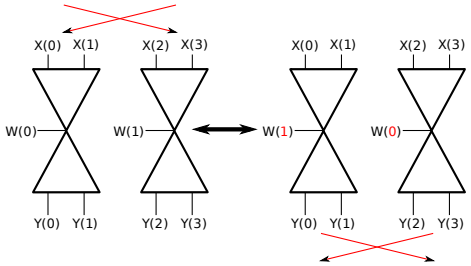

Figure 7.6. Load reordering can be done when storing instead

The store patterns required are shown in the following table:

		S	W
$i < \frac{N}{8}$	$i \equiv 0 \mod 2$	$(0,\frac{i}{2}),(2,\frac{i}{2}),(1,\frac{i}{2}+\frac{N}{8}),(3,\frac{i}{2}+\frac{N}{8})$	Normal
$i < \frac{N}{8}$	$i \equiv 1 \mod 2$	$(1,\frac{i}{2}+\frac{N}{8}),(3,\frac{i}{2}+\frac{N}{8}),(0,\frac{i}{2}),(2,\frac{i}{2})$	Normal
$i \geq \frac{N}{8}$	$i \equiv 0 \mod 2$	$(0,\frac{i}{2}),(2,\frac{i}{2}),(1,\frac{i}{2}+\frac{N}{8}),(3,\frac{i}{2}+\frac{N}{8})$	Flipped
$i \geq \frac{N}{8}$	$i \equiv 1 \mod 2$	$(1,\frac{i}{2}+\frac{N}{8}),(3,\frac{i}{2}+\frac{N}{8}),(0,\frac{i}{2}),(2,\frac{i}{2})$	Flipped

Table 7.7. Store patterns

The AGU connected to the memory port of the memory used for loading data does not need hardware to calculate a different address for each bank, they all use the same one. The same goes for the AGU used to load the twiddle factors.

The most complex AGU required is the one connected to the memory used for storing data. Fortunately the bank addresses can be written as a base address plus a different offset to each bank. The base address is a scalar value which can be computed by accumulation with stride. The offset for each bank and the bank selection can be configured in a register in the AGU. Each bank then has one associated adder for computing its absolute address.

7.4.1 Addressing modes

The syntax of an addressing mode is:

M<portid><width>(ar<regid>++<stride>&<mask>)

Name	Function
portid	Memory port to be used
width	Full width, half width upper part, half width lower part
regid	Index of the address register index to be used
stride	Stride to be added to the address register
mask	Used for the addressing needed by the coefficients

Table 7.8. Addressing mode parameters

7.5 Double radix-2 SIMT-instruction

The syntax for the double radix-2 instruction is:

d_r2_bfly <param> <load_addr> <coefficient_addr> <store_addr>

param consists of three individual parameters that can be either set or not set, **w_duplicate**, **flip** and **w_imag**. **w_duplicate** is used in every layer but the last to use the same twiddle factor in both butterflies. **flip** is used to compensate for the load permutation not being carried out in the memory subsystem. **w_imag** is used for the higher part of the twiddle factors which need to be multiplied by $-i$.

Name	Function
w_duplicate	Use the same coeffcient for both butterflies
flip	Flip the coeffcients and the output data
w_imag	Use $-iW$ instead of W

Table 7.9. Double radix-2 instruction parameters

7.6 Assembly program for radix-2 FFT

```
assign AGU0 to M0
assign AGU1 to M1
assign AGU2 to M2

set R0 [(0,0), (2,0), (1,N/8), (3,N/8)] ;; Two permutation tables
set R1 [(1,N/8), (3,N/8), (0,0), (2,0)] ;;

;; Layer 0 ------------------
set AR0 of AGU0 to 0
set AR0 of AGU1 to 0
set PTABLE of AGU1 to R0

set AR0 of AGU2 to 0
repeat N/16
    d_r2_bfly w_duplicate,       M0(AR0++2), M2Low(0), M1(AR0++)

set AR0 of AGU2 to 0
repeat N/16
    d_r2_bfly w_duplicate, flip, M0(AR0++2), M2Low(0), M1(AR0++)

set AR0 of AGU0 to 1
set AR0 of AGU1 to 1
set PTABLE of AGU1 to R1

set AR0 of AGU2 to 0
repeat N/16
    d_r2_bfly w_duplicate,       M0(AR0++2), M2Low(0), M1(AR0++)

set AR0 of AGU2 to 0
repeat N/16
    d_r2_bfly w_duplicate, flip, M0(AR0++2), M2Low(0), M1(AR0++)

flip memory port M0 with M1

;; --------------------------


;; Layer 1 to L-3  ----------
for (layer=0; layer<=layers-3; layer++) {

    coeff_mask = (0xFFFF << (layers - layer - 3))

    set AR0 of AGU0 to 0
    set AR0 of AGU1 to 0
    set PTABLE of AGU1 to R0

    set AR0 of AGU2 to 0
    repeat N/16
        d_r2_bfly w_duplicate,               M0(AR0++2), M2Low(AR0++&coeff_mask), M1(AR0++)

    set AR0 of AGU2 to 0
    repeat N/16
        d_r2_bfly w_duplicate, flip, w_imag, M0(AR0++2), M2Low(AR0++&coeff_mask), M1(AR0++)

    set AR0 of AGU0 to 1
    set AR0 of AGU1 to 1
    set PTABLE of AGU1 to R1

    set AR0 of AGU2 to 0
    repeat N/16
        d_r2_bfly w_duplicate,               M0(AR0++2), M2Low(AR0++&coeff_mask), M1(AR0++)

    set AR0 of AGU2 to 0
    repeat N/16
        d_r2_bfly w_duplicate, flip, w_imag, M0(AR0++2), M2Low(AR0++&coeff_mask), M1(AR0++)

    flip memory port M0 with M1
}
;; --------------------------

;; Layer L-2 ----------------
set AR0 of AGU0 to 0
set AR0 of AGU1 to 0
set PTABLE of AGU1 to R0

set AR0 of AGU2 to 0
repeat N/16
    d_r2_bfly w_duplicate,               M0(AR0++2), M2Low(AR0++), M1(AR0++)

set AR0 of AGU2 to 0
repeat N/16
    d_r2_bfly w_duplicate, flip, w_imag, M0(AR0++2), M2Low(AR0++), M1(AR0++)

set AR0 of AGU0 to 0
set AR0 of AGU1 to 0
set PTABLE of AGU1 to R1

set AR0 of AGU2 to 0
```

```
repeat N/16
    d_r2_bfly w_duplicate,               M0(AR0++2), M2High(AR0++), M1(AR0++)

set AR0 of AGU2 to 0
repeat N/16
    d_r2_bfly w_duplicate, flip, w_imag, M0(AR0++2), M2High(AR0++), M1(AR0++)

flip memory port M0 with M1
;; --------------------------

;; Layer L-1 ----------------
set AR0 of AGU0 to 0
set AR0 of AGU1 to 0
set PTABLE of AGU1 to R0

flip memory port M0 with M1

set AR0 of AGU2 to 0
repeat N/16
    d_r2_bfly               M0(AR0++2), M2Low(AR0++), M1(AR0++)

set AR0 of AGU2 to 0
repeat N/16
    d_r2_bfly flip, w_imag, M0(AR0++2), M2Low(AR0++), M1(AR0++)

set AR0 of AGU0 to 1
set AR0 of AGU1 to 1
set PTABLE of AGU1 to R1

set AR0 of AGU2 to 0
repeat N/16
    d_r2_bfly               M0(AR0++2), M2High(AR0++), M1(AR0++)

set AR0 of AGU2 to 0
repeat N/16
    d_r2_bfly flip, w_imag, M0(AR0++2), M2High(AR0++), M1(AR0++)

flip memory port M0 with M1
;; --------------------------
```

Listing 7.1. Pseudo assembler code for radix-2 FFT on bit reversed data

Chapter 8

Discussion

8.1 Conclusions

The inherently flexible structure of the simulator was necessary in the early development stage of the project. As the project evolves and more design decitions are made the simulator will loose much of its flexibility and become more fixed function. This can be achieved within the existing simulator framework by adding more constraints on the instruction implementations so that they conform to only a fixed set of pipeline templates.

By studying the radix-2 FFT we have shown that even algorithms with a complex addressing pattern can be adapted to fully utilize the parallel datapath while only requiring additional simple addressing hardware. By supporting this algorithm with a SIMT instruction almost 100% utilization of the datapath can be achieved. This is very promising but more algorithms need to be studied.

8.2 Future work

Since the simulator is based on a design philosophy and hardware design which still is in development there will be various of things to change, add and remove in the future. The research project will continue to proceed and so will the functionalities of this simulator. Limitations that needs to be taken care of are suggested as follows:

- Extend the simulator with a GUI.
- Add functionality to take care of structural hazards.
- Specify the different pipeline templates.
- Add simulator variables to identify hardware collisions.

There are also many different algorithms that need to be analyzed so that the true requirements on the address generating hardware and memory subsystem can be known.

Bibliography

[1] David M. Beazley. *Using SWIG to Control, Prototype, and Debug C Programs with Python*. http://www.swig.org/papers/Py96/python96.html.

[2] Dake Liu. *Embedded DSP processor design*. Morgan Kaufmann Publishers, 1 edition, 2008. ISBN 978-0-12-374123-3.

[3] Burghard Rebel & Reiner Creutzburg Michael Gössel. *Memory Architecture & Parallel Access*. Elsevier Science B.V., 1 edition, 1994. ISBN 0-444-82104-X.

Appendix A

Class definitions

A.1 C++ classes

A.1.1 SIMD Unit

```
#ifndef SIMD_UNIT_H
#define SIMD_UNIT_H

#include "../config.h"
#include "./register.h"
#include "./register_file.h"
#include "./program_memory.h"
#include "./mac_unit.h"
#include "./alu.h"
#include "../memory_subsystem/LocalStore.h"

#include "../instruction_library/instruction_library.h"
#include "./instruction_implementation.h"

#include <utility>


typedef std::pair<InstructionData, unsigned int> PipelineItem;

struct BusCmd {
  bool pending;

  int readWrite; // 0 read, 1 write
  int busIdx;    // 1, 2

  uint32_t  addr;
  uint16_t *data;
  int       size;

  uint16_t result; // stores the result of the bus cmd
};


class SIMDUnit
{
 public:
  Config config;
  bool performBusCmd();

  SIMDUnit(){};
  SIMDUnit(Config& config);
  ~SIMDUnit();

  // Clock all registers within the SIMD unit including those in the register file
  void clock();

  // Creates a XML structured variable containing the hardware state of components in the SIMD unit
  std::string getXMLrepr();

  /* Instruction flow  */

  // Keeps track of instruction in the pipeline.
  std::vector<PipelineItem> pipelineItems;
```

```
  // Reference to the instruction library
  InstructionLibrary* instructionLibrary;


  /* Hardware state */

  // Signals that the SIMD unit want to halt.
  bool halt;

  // Signals that the SIMD unit is in saturation mode.
  bool saturation;

  // Program counters
  int pc, nextpc;

  ProgramMemory pm;
  LocalStore* ls;
  RegisterFile rf;

  MACUnit mac;
  ALU alu;

  BusCmd busCmd;
};

#endif
```

A.1.2 ALU

```
#ifndef _ALU_H
#define _ALU_H

#include "../config.h"
#include "register.h"

class ALU {
 public:
  Config config;

  // Registers available
  Register opA;
  Register opB;
  Register result;

  // Build hardware
  void build(Config& config);
  void destroy();

  // Clock registers
  void clock();

};


#endif
```

A.1.3 Register File

```
#ifndef REGISTER_FILE_H
#define REGISTER_FILE_H

#include "register.h"

class RegisterFile
{
 public:
  // Width of registers in bits
  int size;
  Register *registers;

  void alloc(int rf_size, int reg_width);
  void dealloc();

  // Copies content of registers[index] to dstReg.
  void readRegister(int index, Register& dstReg);

  // Copies content of srcReg to registers[index].
  void writeRegister(int index, Register& srcReg);

  // Get a reference to register.
  Register& getRegister(int index);

  // Clocks the register file feeding data on the input to the output of all registers
  void clock();
};

#endif
```

A.1.4 Register

```
#ifndef REGISTER_H
#define REGISTER_H

#include <stdint.h>


class Register
{

 private:
  uint64_t getSliceNoOverlap(int offset, int size);
  void setSliceNoOverlap(uint64_t bits, int offset, int size);

 public:
  // Width of register in bits
  int size;
  uint64_t *data_in, *data_out;

  void alloc(int size);
  void dealloc();

  // Gets a slice of bits of <size> from the start of <offset>
  // Maximally 64 bits can be retrieved.
  uint64_t getSlice(int offset, int size);

  // Sets a slice of bits of <size> from the start of <offset>
  // Maximally 64 bits can be set.
  void    setSlice(uint64_t bits, int offset, int size);

  // Copy the contents from srcReg to this register.
  void copyFrom(Register& srcReg);

  // Clocks the register and feeds data on the input to the output
  void clock();
};

#endif
```

A.1.5 Program Memory

```
#ifndef PROGRAM_MEMORY_H
#define PROGRAM_MEMORY_H

#include "instruction_data.h"
#include <vector>

class ProgramMemory
{
 public:
  // The list of instructions in the memory
  std::vector<InstructionData> idata;

  void alloc(int size);
  void dealloc();

  // Store an instruction on address <index>
  void setInstructionData(InstructionData idata, int index);

  // Retrieve an instruction from address <index>
  InstructionData& getInstructionData(int index);
};

#endif
```

A.1.6 Instruction Library

```
#ifndef _INSTRUCTION_LIBRARY_H
#define _INSTRUCTION_LIBRARY_H

#include <utility>
#include <map>
#include <string>
#include <vector>

class Object {
 public:
  void* handle;
  virtual ~Object(){};
};

typedef Object* (*objectCreatorPtr)();
Object* loadObject(std::string filePath);
```

```
struct InstructionType {
  int id;
  std::string name; // Instruction syntax
  std::vector<std::string> format; // List of tokes needed to be parsed
  Object* implementation;
};

typedef std::map<std::string, int> NameToIdMap_t;

class InstructionLibrary {
 public:
  NameToIdMap_t instrIdMap;

  // List of available instructions
  std::vector<InstructionType> instrTypes;

  // Load an instruction
  void loadInstructionsFromFolder(std::string folderName);

  // Load the instruction set
  void loadInstruction(std::string instrDirName);

  InstructionType getInstructionTypeById(int id);
  InstructionType getInstructionTypeByName(std::string name);

};

#endif
```

A.1.7 Instruction Data

```
#ifndef INSTRUCTION_DATA_H
#define INSTRUCTION_DATA_H

#include <stdint.h>
#include <vector>
#include <utility>

// Representing types of operands and operations provided with an instruction.
// Two types of operand, register or immediate
// Two types of operations, signed or unsigned
enum {
  OPTYPE_UNSET    = 0,
  OPTYPE_REG      = 1,
  OPTYPE_IMED     = 2,
  OPMODE_UNSIGNED = 3,
  OPMODE_SIGNED   = 4,
};

typedef std::pair<int, int> Operand; // type, value

struct InstructionData
{
  // ID-number which represents the type of operation, eg. add, sub, nop
  int8_t instructionType;

  // Contains the operands
  std::vector<Operand> operands;

  // Configuration flags like width, signed or unsigned etc. defined by instruction parser and program.
  int configFlags;

  void addOperand(Operand operand) {
    operands.push_back(operand);
  }

  // Get width from configFlags
  int getWidth() {
    return (configFlags & 0xFF);
  }

  // Set width to configFlags
  void setWidth(int width) {
    width = width & 0xFF;
    configFlags = width;
  }

  // Get signed or unsigned from configFlags
  int getMode() {
    return ((configFlags & 0xF00) >> 8);
  }

  // Set signed or unsigned to configFlags
  void setMode(int mode) {
    configFlags = (configFlags | (mode << 8));
  }
};
```

```
#endif
```

A.1.8 Instruction Implementation

```
#ifndef INSTRUCTION_IMPLEMENTATION
#define INSTRUCTION_IMPLEMENTATION

#include "../instruction_library/instruction_library.h"
#include "simd_unit.h"

class SIMDUnit;

class InstructionImplementation : public Object {
 public:
  unsigned int pipelineDepth;
  virtual void execute(SIMDUnit& simd, InstructionData& idata, unsigned int pipelineStage) = 0;
  virtual ~InstructionImplementation(){};
};

#endif
```

A.1.9 Instruction template

```
#include "../../common.h"

class Implementation : public InstructionImplementation {
public:
  Implementation();
  void  execute(SIMDUnit& simd, InstructionData& idata, unsigned int pipelineStage);
};

Implementation::Implementation() {
  // Depth of pipeline
  pipelineDepth = 1;
}


void Implementation::execute(SIMDUnit& simd, InstructionData& idata, unsigned int pipelineStage) {
  switch (pipelineStage) {
  case 0:
    /* Tasks to be performed in cycle 1 */
    return;

  case 1:
    /* Tasks to be performed in cycle 2 */
    return;
  };
}

Object* createObject() {
  return static_cast<Object*>(new Implementation());
}
```

A.1.10 Assembly File Parser

```
#ifndef _ASSEMBLYFILEPARSER_H
#define _ASSEMBLYFILEPARSER_H

#include <string>
#include <vector>
#include <utility>
#include <map>

#include "../instruction_library/instruction_library.h"
#include "../simd_unit/instruction_data.h"


typedef std::pair<std::string, std::string> AsmLinePair_t; // <location str, line>
typedef std::map<std::string, int> LabelMap_t;            // <name, addr>

typedef std::string (*TokenParser_t) (InstructionData&, std::string, LabelMap_t&, int);
typedef std::map<std::string, TokenParser_t> TokenMap_t;

class AssemblyFileParser {
public:
    AssemblyFileParser(InstructionLibrary* _library);

    virtual ~AssemblyFileParser();

    static const std::string whitespace;
    static const char commentToken;
    static const char preprocessorToken;
    static const char stringToken;
    static const char labelToken;
```

```
    static const std::string includeString;

    // Parse .asm file
    std::vector<InstructionData>* parse(std::string fileName);

    // Parse one line of a .asm file
    std::vector<InstructionData>* parseString(std::string inlineAsm);

private:
    InstructionLibrary* library;
    TokenMap_t tokenMap;

    std::vector<AsmLinePair_t> lines;
    LabelMap_t labelMap;

    void load(std::string fileName); // Load .asm file
    void loadString(std::string inlineAsm); // Load one line of a .asm file

    // Move .main segment to the top of the file
    void reorder();

    // Parse one assembler line
    void parseLine(InstructionData& idata, std::string line, int lineAddr);
};

#endif  /* _ASSEMBLYFILEPARSER_H */
```

A.1.11 Simulator

```
#ifndef _SIMULATOR_H
#define _SIMULATOR_H

#include "config.h"
#include "assembly_file_parser/assemblyfileparser.h"
#include "simd_unit/simd_unit.h"
#include "memory_subsystem/MemorySubsystem.h"
#include "hardware_model.h"

#include <string>
#include <map>
#include <utility>

class Simulator {
private:
  typedef std::map<std::string, std::vector<InstructionData>*> ProgramMap_t;
  ProgramMap_t programMap;

public:
  Config config;
  InstructionLibrary library;
  HardwareModel model;

  // Build the hardware model.
  void build();

  // Loads all instruciton from within a directory
  void loadInstructions(std::string directoryName);

  // Load one instruction
  void loadInstruction(std::string instrDirectoryName);

  // Loads an asm program into the simulator and assigns it a name.
  void loadProgram(std::string fileName, std::string programName);

  // Loads an inline asm program.
  void loadInlineProgram(std::string program, std::string programName);

  // Loads the local store with values
  void loadLocalStore(int simdIndex, uint32_t addr, std::string data, int portID);

  // Loads the permutation table
  void loadPermTable(int simdIndex, uint32_t addr, std::string data);

  // Read the content on a given address of the local memory
  std::string readLocalStore(int simdIndex, uint32_t addr, int portID);

  // Loads the register file with values
  void loadRegisterFile(int simdIndex, int registerIndex, std::string registerContent);

  // Get a list of available programs.
  std::vector<std::string> listPrograms();

  // Sets the program of a simd unit.
  void setProgram(int simdIndex, std::string programName);

  // Get an xml representation of the hardware model.
  std::string getXMLrepr();

  // Simulate until simd_unit[simdIndex] halts.
```

```
  void simUntilHalt(int simdIndex);

  // Simulate a given number of cycles.
  void simCycles(unsigned int cycles);
};

#endif
```

A.1.12 Configuration

```
#ifndef CONFIG_H
#define CONFIG_H

#include <string>

struct Config {
public:
  int WORD_SIZE;       // Number of bits in a word
  int VECTOR_SIZE;     // Number of words in a vector
  int GUARD_SIZE;      // Number of guard bits for a double word
  int RF_SIZE;         // Size of the register file in vector registers
  int PM_SIZE;         // Size of the program memory
  int LM_SIZE;         // Size of the local memory in vectors
  int SIMD_UNIT_COUNT; // Number of SIMD units.

  int VECTOR_BIT_SIZE; // Number of bits in a vector
  int VECTOR_BIT_SIZE_WO_GUARDS; // Number of bits in a vector without guards

  // Predefined configuration
  Config() {
    WORD_SIZE   = 16;
    VECTOR_SIZE = 8;
    GUARD_SIZE  = 0;
    RF_SIZE = 16;
    PM_SIZE = 1024;
    LM_SIZE = 1024;
    SIMD_UNIT_COUNT = 8;

    update();
  }

  void update() {
    VECTOR_BIT_SIZE = WORD_SIZE*VECTOR_SIZE + (VECTOR_SIZE*GUARD_SIZE)/2;
    VECTOR_BIT_SIZE_WO_GUARDS = WORD_SIZE*VECTOR_SIZE;
  }

};

#endif
```

A.1.13 ALU Utilities

```
#ifndef ALU_UTILS_H
#define ALU_UTILS_H

uint64_t signExtend(uint64_t value, int width) {
  if(value >> (width-1) == 1) {
    return (-1LL << width) | value;
  }

  return value;
}

uint64_t saturate(uint64_t value, int width, int mode, bool satFlag) {
  uint64_t max, min;

  if(mode == OPMODE_SIGNED) {
    max = (1 << width-1) - 1;
    min = (-1 << width) | (1 << width-1);
  }
  else {
    max = (1 << width) - 1;
    min = 0;
  }
  if((int64_t)value > (int64_t)max && satFlag == true) {
    return max;
  }
  else if((int64_t)value < (int64_t)min && satFlag == true) {
    return min;
  }

  return value;
}

/* Template for an ALU pipeline */
```

```
class ALUBaseImplementation : public InstructionImplementation {
 public:

  ALUBaseImplementation() {
    pipelineDepth = 3;
  }

  virtual uint64_t operation(uint64_t a, uint64_t b) = 0;

  void  execute(SIMDUnit& simd, InstructionData& idata, unsigned int pipelineStage) {

    int width = idata.getWidth();
    int mode = idata.getMode();

    switch (pipelineStage) {
    case 0:

      switch(idata.operands[1].first) {
      case OPTYPE_REG:
  simd.rf.readRegister(idata.operands[1].second, simd.alu.opA);
  break;
      case OPTYPE_IMED:
  for (int i=0; i<simd.config.VECTOR_BIT_SIZE_WO_GUARDS; i+=width) {
    simd.alu.opA.setSlice(idata.operands[1].second, i, width);
  }
  break;
      };

      switch(idata.operands[2].first) {
      case OPTYPE_REG:
  simd.rf.readRegister(idata.operands[2].second, simd.alu.opB);
  break;
      case OPTYPE_IMED:
  for (int i=0; i<simd.config.VECTOR_BIT_SIZE_WO_GUARDS; i+=width) {
    simd.alu.opB.setSlice(idata.operands[2].second, i, width);
  }
  break;
      };
      return;



    case 1:
      for (int i=0; i<simd.config.VECTOR_BIT_SIZE_WO_GUARDS; i+=width) {
  uint64_t a, b;
  if(mode == OPMODE_SIGNED) {
    a = signExtend(simd.alu.opA.getSlice(i, width), width);
    b = signExtend(simd.alu.opB.getSlice(i, width), width);
  }
  else {
    a = simd.alu.opA.getSlice(i, width);
    b = simd.alu.opB.getSlice(i, width);
  }

  uint64_t r = saturate(operation(a,b), width, mode, simd.saturation);

  simd.alu.result.setSlice(r, i, width);
      }
      return;

    case 2:
      simd.rf.writeRegister(idata.operands[0].second, simd.alu.result);
      return;
    };
  };
};


#endif
```

A.2 SWIG extensions

A.2.1 Setup script

```
#!/usr/bin/env python

from distutils.core import setup, Extension

# Module definition, specifies SWIG wrapper to interface
p3rma_module = Extension('_p3rma',
                         sources=['p3rma_wrap.cpp'],
                         libraries = ['simulator',
                                      'dl'],
                         extra_compile_args = ['-fPIC'],
                         library_dirs = ['../'],
```

```
                          runtime_library_dirs = ['../'],
                          )

# Specifies which models to use when creating the Python library
setup (name = 'P3RMA',
       version = '0.3',
       author      = "",
       description = """P3RMA""",
       ext_modules = [p3rma_module],
       py_modules = ["p3rma"],
       )
```

A.2.2 Python interface

```
%module p3rma
%include "std_string.i"
%include "std_vector.i"

// Maps C++ data types to Python data types
%apply unsigned long long { uint64_t }
%apply unsigned long { uint32_t }

namespace std {
%template(StringVector) vector<string>;
}

// Include error handling
%include exception.i

%exception {
try {
    $action
  } catch( std::string& errorMsg) {
    PyErr_SetString(PyExc_RuntimeError, errorMsg.c_str());
    return NULL;
  }
}

// Header files to be interfaced
%include ../config.h


%include ../simulator.h
%{
#include "../simulator.h"
%}
```

Appendix B

Radix-2 FFT

```
import math
import cmath
import random
from numpy.fft import fft as fft_ref

class vector_memory():
    def __init__(self, width, size):
        self.width = width
        self.size = size

        self.m = [[0 for a in xrange(size/width)] for S in xrange(self.width)]

        self.ptable = [(i,0) for i in xrange(self.width)]

    def vload(self, va):
        assert(len(va) == self.width)
        raw_data = [self.m[i][va[i][1]] for i in xrange(self.width)]
        data = [raw_data[va[i][0]] for i in xrange(self.width)]
        return data

    def vstore(self, va, data):
        assert(len(va) == self.width)
        raw_data = [data[va[i][0]] for i in xrange(self.width)]
        for i in xrange(self.width):
            self.m[i][va[i][1]] = raw_data[i]

    def set_ptable(self, ptable):
        assert(len(ptable) == self.width)
        self.ptable = ptable

    def load_ptable(self, ga):
        va = [(self.ptable[i][0], self.ptable[i][1]+ga) for i in xrange(self.width)]
        return self.vload(va)

    def store_ptable(self, ga, data):
        va =[(self.ptable[i][0], self.ptable[i][1]+ga) for i in xrange(self.width)]
        self.vstore(va, data)


def randComplex():
    return random.random()*1.0 + random.random()*1.0j

def error(x, y):
    return sum([abs(x[i]-y[i]) for i in xrange(len(x))])/len(x)

def digit_reverse(x, N):
    S = log2(N)
    def reverse(i):
        return sum([((i>>j)%2)<<(S-j-1) for j in xrange(S)])

    return [x[reverse(i)] for i in xrange(len(x))]

def log2(N):
    x = 0
    while 2**x < N:
        x+=1
    return x

def twiddle(layer, k):
    return cmath.exp((-1.0j)*2.0*math.pi*k/(1 << (layer+1)))

def butterfly(z, w, updown=0, dup=0, flip=0, imag=0):
```

```
    if updown==1:
        w = [w[2],w[3], w[0], w[1]]

    if dup==1:
        w = [w[0], w[0]]

    if flip==1:
        w = [w[1], w[0]]

    if imag==1:
        w = [-1.0j*w[0], -1.0j*w[1]]

    t = [z[0],
         z[1] * w[0],
         z[2],
         z[3] * w[1]]

    z[0] = (t[0] + t[1])/2.0
    z[1] = (t[0] - t[1])/2.0
    z[2] = (t[2] + t[3])/2.0
    z[3] = (t[2] - t[3])/2.0

    if flip==1:
        z = [z[2], z[3], z[0], z[1]]

    return z

def fft(x, N):
    y = digit_reverse(x, N)
    layers = log2(N)

    m1 = vector_memory(4, N)
    m2 = vector_memory(4, N)
    c  = vector_memory(4, N/2)

    c.set_ptable([(0,0), (1,0), (2,0), (3,0)])
    for i in xrange(N/8):
        c.store_ptable(i, [twiddle(layers-1, 4*i + j) for j in xrange(4)])

    m1.set_ptable([(0,0), (1,0), (2,0), (3,0)])
    for i in xrange(N/8):
        m1.store_ptable(i, [y[4*i + j] for j in xrange(4)])

    m1.set_ptable([(2,0), (3,0), (0,0), (1,0)])
    for i in xrange(N/8):
        m1.store_ptable(i + (N/8), [y[4*i + (N/2) + j] for j in xrange(4)])

    for l in xrange(layers):
        coeff_mask = ~((1 << (layers-l-1))-1)
        coeff_mask = coeff_mask >> 2

        m1.set_ptable([(0,0), (1,0), (2,0), (3,0)])
        m2.set_ptable([(0,0), (2,0), (1,N/8), (3,N/8)])

        for k in xrange(N/16):
            data = m1.load_ptable(2*k)

            w = c.load_ptable(k & coeff_mask)

            new_data = butterfly(data, w,
                                 updown = 0,
                                 dup    = 1*(l<(layers-1)),
                                 flip   = 0,
                                 imag   = 0)

            m2.store_ptable(k, new_data)

        for k in xrange(N/16):
            data = m1.load_ptable(2*k + N/8)

            w = c.load_ptable(k & coeff_mask)

            new_data = butterfly(data, w,
                                 updown = 0,
                                 dup    = 1*(l<(layers-1)),
                                 flip   = 1,
                                 imag   = 1*(l!=0))

            m2.store_ptable(k + N/16, new_data)

        m2.set_ptable([(1,N/8), (3,N/8), (0,0), (2,0)])
        for k in xrange(N/16):
            data = m1.load_ptable(2*k+1)

            w = c.load_ptable(k & coeff_mask)

            new_data = butterfly(data, w,
                                 updown = 1*(l>=(layers-2)),
                                 dup    = 1*(l<(layers-1)),
                                 flip   = 0,
                                 imag   = 0)
```

```
            m2.store_ptable(k, new_data)

        for k in xrange(N/16):
            data = m1.load_ptable(2*k + N/8 + 1)

            w = c.load_ptable(k & coeff_mask)

            new_data = butterfly(data, w,
                                 updown = 1*(l>=(layers-2)),
                                 dup    = 1*(l<(layers-1)),
                                 flip   = 1,
                                 imag   = 1*(l!=0))

            m2.store_ptable(k + N/16, new_data)

        m1,m2 = m2,m1

    m1.set_ptable([(0,0), (1,0), (2,0), (3,0)])
    for i in xrange(N/8):
        data = m1.load_ptable(i)
        for j in xrange(4):
            y[4*i + j] = data[j]

    m1.set_ptable([(2,0), (3,0), (0,0), (1,0)])
    for i in xrange(N/8):
        data = m1.load_ptable(i + (N/8))
        for j in xrange(4):
            y[4*i + (N/2) + j] = data[j]

    return y


N = 2**10
x = [randComplex() for i in xrange(N)]
X1 = fft(x, N)
X2 = fft_ref(x)

print error([x*N for x in X1], X2)
print max([abs(x) for x in X1])
```

Appendix C

Assembly Instruction Set

portswap

Syntax

1 **portswap**

Operation

1 Swap ports

Description

1 Swap memories on port 0 and 1.

Example

```
portswap
```

out$X$

Syntax

1 **out$X$** K **r**t

$t \in [0..31], K \in [0..2^{LM\_SIZE} - 1], X \in 1, 2$

Operation

1 bus$X$_M(K) $\leftarrow$ **r**t[0..16]

Description

1 Stores the 16 first bits from **r**t on the given address in ring bus X.

Example

```
out1 $0xff14 r10
out2 $0xba34 r22
```

in*X*

Syntax

1 **in*X*** K **r***t*

$t \in [0..31], K \in [0..2^{LM\_SIZE} - 1], X \in 1, 2$

Operation

1 **r***t*$[0..16] \leftarrow$ bus$X$_M(K)

Description

1 Reads the content on the given address from ring bus X and stores it to the 16 first bits of register **r***t*.

Example

```
in1 $0xff14 r10
in2 $0xba34 r22
```

sat

Syntax

1 **sat**

Operation

1 Saturation operation

Description

1 Change the saturation register to true or false. Indicates if the SIMD unit should saturate the results generated from ALU operations.

Example

```
sat
```

halt

Syntax

1 **halt**

Operation

1 Halt operation

Description

1 Sets the halt register of the SIMD unit which signals that simulation should terminate.

Example

```
halt
```

sete

Syntax

1	**sete**	<width>	**r**t K **r**a
2	**sete**	<width>	**r**t K_1 K_2

$t \in [0..31], a \in [0..31], b \in [0..31], K \in [0..2^{width} - 1]$,
$K \in [-2^{width-1}..2^{width-1} - 1]$

width Byte, word or double word elements

Operation

1 **r**$t[K] \leftarrow$ **r**$a[K]$
2 **r**$t[K_1] \leftarrow K_2$

Description

1,2 Sets a value to a target element of a register.

Example

```
sete 8 r0 $0 r2
sete 32 r4 $1 $0x12340010
sete 16 r12 $0x04 $85
```

set

Syntax

1	**set**	<width>	$\mathbf{r}t$ $\mathbf{r}a$
2	**set**	<width>	$\mathbf{r}t$ K

$t \in [0..31], a \in [0..31], b \in [0..31], K \in [0..2^{width} - 1]$,
$K \in [-2^{width-1}..2^{width-1} - 1]$

width	Byte, word or double word elements

Operation

1 $\mathbf{r}t[i] \leftarrow \mathbf{r}a[i], i \in 1, 2..\#elements$
2 $\mathbf{r}t[i] \leftarrow K, i \in 1, 2..\#elements$

Description

1,2 Sets a value to all elements of a register.

Example

```
set 8 r0 r2
set 32 r4 $0x12340010
set 16 r12 $85
```

nop

Syntax

1 **nop**

Operation

1 No-operation

Description

1 Perform no operation.

Example

```
nop
```

store

Syntax

1 **store** K **r**a

$t \in [0..31], a \in [0..31], b \in [0..31], K \in [0..2^{LM\_SIZE} - 1]$

Operation

1 $\mathrm{M}(K) \leftarrow$ **r**a

Description

1 Stores a vector register to the local memory with different addressing modes.

Example

```
store r0 $0xff10
store r12 $0x5
```

load

Syntax

1 **load** **r**t K

$t \in [0..31], a \in [0..31], b \in [0..31], K \in [0..2^{LM\_SIZE} - 1]$

Operation

1 **r**$t \leftarrow \mathrm{M}(K)$

Description

1 Loads a vector from local memory to the vector register **r**t with different addressing modes.

Example

```
load r0 $0xff10
load r12 $0x5
```

sub

Syntax

1	**sub**	<width>	<mode>	**r**t **r**a **r**b
2	**sub**	<width>	<mode>	**r**t K **r**a
3	**sub**	<width>	<mode>	**r**t **r**a K

$t \in [0..31], a \in [0..31], b \in [0..31], K \in [0..2^{width} - 1]$,
$K \in [-2^{width-1}..2^{width-1} - 1]$

width	Byte, word or double word elements
mode	Signed or unsigned subtraction

Operation

1	**r**$t \leftarrow$ **r**$a -$ **r**b
2,3	**r**$t \leftarrow$ **r**$a - K$

Description

1	Subtract two vector registers.
2,3	Subtract all vector element in a register with a constant or vice versa.

The output can optionally be saturated as signed or unsigned numbers according to the saturation bit of `GV_MCONF`

Example

```
sub 16 signed r0 r1 r2
sub 32 unsigned r1 r2 $0x1234
sub 8 unsigned r8 $10 r1
```

add

Syntax

1	**add**	<width>	<mode>	**r**t **r**a **r**b
2	**add**	<width>	<mode>	**r**t K **r**a
3	**add**	<width>	<mode>	**r**t **r**a K

$t \in [0..31], a \in [0..31], b \in [0..31], K \in [0..2^{width} - 1]$,
$K \in [-2^{width-1}..2^{width-1} - 1]$

width	Byte, word or double word elements
mode	Signed or unsigned addition

Operation

1	**r**$t \leftarrow$ **r**a + **r**b
2,3	**r**$t \leftarrow$ **r**a + K

Description

1	Add two vector registers.
2,3	Add a constant to all vector element in a register.

The output can optionally be saturated as signed or unsigned numbers according to the saturation bit of `GV_MCONF`

Example

```
add 16 signed r0 r1 r2
add 32 unsigned r1 r2 $0xbeef
add 8 unsigned r8 $24 r1
```