OpenCL Implementation of Exposed Data Path Architectures as General Purpose Accelerators

Julius Roob
julius@juliusroob.de

University of Kaiserslautern, Embedded Systems Group
Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Kaiserslautern, _______________ ________________

Date Signature
Acknowledgements

I want to use this opportunity to thank all the kind and supporting people who helped me through developing and writing this master thesis.

My supervisors, Tripti Jain and Prof. Klaus Schneider for their patient encouragement, guidance and faith in my abilities.

The kind and welcoming people working at the Service Center Informatik (SCI), who provided me with a powerful virtual machine without which nothing would have worked.

My friends, family and partner who have shown nothing but passionate care and understanding when I was frustrated by slow progress or difficult situations.

Thank you all.
## Contents

1 Introduction ........................................... 1
   1.1 Motivation ........................................ 1
   1.2 Main Goals ........................................ 2
   1.3 Structure .......................................... 3

2 Related Work .......................................... 5
   2.1 SCAD Machine ...................................... 5
   2.2 Exposed Datapath Architectures ..................... 5
   2.3 FPGAs ............................................. 6
      2.3.1 OpenCL on FPGA ............................... 6
   2.4 Radix-Based Interconnection Networks ................ 6

3 The SCAD Machine Architecture ....................... 7
   3.1 Assembly Language ................................... 9
      3.1.1 Assembly Language Example ..................... 9
      3.1.2 List of Instructions ........................... 9
   3.2 Functional Units .................................... 10
      3.2.1 Control Unit .................................. 10
      3.2.2 Memory: Load-Store Unit ....................... 10
      3.2.3 Reorder Unit .................................. 10
      3.2.4 General Purpose Processing Unit ............... 11
   3.3 Example Programs .................................. 11
   3.4 Input and Output Buffers ........................... 15
   3.5 Data Transport Network (DTN) ....................... 15
   3.6 Move Instruction Bus ................................ 16
   3.7 Reconfigurability .................................... 16

4 OpenCL and Intel HARP .................................. 17
   4.1 Programming Model ................................ 17
      4.1.1 C++ API ...................................... 18
   4.2 Motivation for OpenCL on FPGAs ..................... 18
   4.3 Intel FPGA SDK for OpenCL ........................... 18
      4.3.1 Available Documentation ....................... 19
      4.3.2 Concurrent Kernel Execution .................... 19
      4.3.3 Pipes and Channels ............................ 19
      4.3.4 Autorun Kernels ................................ 21
   4.4 Intel Harp ........................................ 22
      4.4.1 Intel Quickpath Interconnect ................. 22
## Architecture and Implementation

5.1 Motivation for Use as an Accelerator .............................................. 26
5.2 Configuration ................................................................. 26
5.3 Kernels .............................................................................. 29
5.4 Channels ............................................................................. 29
5.5 Fundamental Data Types ......................................................... 31
5.6 Data Transport Network .......................................................... 31
5.7 Buffers ................................................................................. 32
  5.7.1 Input Buffers ................................................................. 32
  5.7.2 Output Buffers ............................................................... 33
  5.7.3 Usage ............................................................................ 33
  5.7.4 Message Acknowledgement .............................................. 34
  5.7.5 Sync ............................................................................. 36
5.8 Control Unit .......................................................................... 36
  5.8.1 Sending Sync Markers ...................................................... 37
  5.8.2 Hardware Instruction Set .................................................. 38
  5.8.3 Second Iteration Control Unit .......................................... 38
5.9 Load-Store Unit ....................................................................... 39
5.10 Host Application API .............................................................. 40
  5.10.1 Description .................................................................... 40
  5.10.2 Assembly ...................................................................... 40
  5.10.3 Machine ........................................................................ 40
5.11 Host Applications ................................................................. 43
5.12 Assembler ............................................................................ 43

## SCAD Machine Configurations

6 SCAD Machine Configurations .......................................................... 45

## Hardware Synthesis

7 Hardware Synthesis ................................................................. 47
  7.1 Problems, Contrast to Emulation ........................................... 47
  7.2 Hardware Requirements ....................................................... 48
  7.3 Experiments ....................................................................... 48

## Results

8 Results ...................................................................................... 51
  8.1 Successful Synthesis ............................................................. 51
  8.2 Unexpected Behaviour .......................................................... 53

## Summary and Conclusion

9 Summary and Conclusion .............................................................. 55

## Future Work

10 Future Work .............................................................................. 57
  10.1 Asynchronous MIB ............................................................... 57
  10.2 Alternate Instruction Set ......................................................... 58
  10.3 Shared Memory LSU ............................................................ 58
  10.4 Interconnects ...................................................................... 58
  10.5 Partial reconfiguration .......................................................... 58
  10.6 Avalon Interface .................................................................. 58
  10.7 If-Then-Else Functional Unit ................................................ 59

Bibliography .................................................................................... 61
Abstract

The Synchronous Control Asynchronous Data (SCAD) architecture is an exposed datapath processor architecture in development at the Embedded Systems Group at the University of Kaiserslautern. It aims at maximising Instruction-Level Parallelism (ILP) by replacing the register file of traditional superscalar processor architectures by distributed buffers closely located to each functional unit. SCAD machine processors are programmed using move instructions, which transfer values from functional unit outputs to new inputs through a scalable interconnect network.

To increase ILP, these moves happen asynchronously, i.e., when data is produced or arrives, instead of waiting for each transfer to happen before executing the next one.

The aim of this thesis is to provide a first FPGA-based implementation of a parameterised SCAD machine. It will be tested as an accelerator, a special co-processor to which computational problems are delegated, both for performance and to keep load on the application processor low. Also, this SCAD machine is intended as a test environment for different interconnecting Data Transport Networks (DTNs).

As a platform for this, the Intel FPGA SDK for OpenCL generating hardware for the Intel HARP hardware has been chosen. There are open questions regarding the code generation and hardware synthesis from OpenCL that this thesis intends to answer through experimentation.

Zusammenfassung


Um ILP zu verbessern, finden diese Verschiebevorgänge erst statt, wenn die Daten produziert oder angekommen sind. So lange genug Platz in den entsprechenden Puffern vorhanden ist, wird nicht auf die Fertigstellung gewartet. Stattdessen wird das Programm asynchron weiter abgearbeitet.

Ziel dieser Masterarbeit ist es, eine erste FPGA-basierte Implementierung der SCAD Maschine zu entwickeln. Sie wird als Beschleuniger getestet, d.h. als spezieller Ko-prozessor an den Rechenaufgaben delegiert werden, um die Berechnung zu beschleunigen und den Anwendungsprozessor zu entlasten. Außerdem soll diese Implementierung der SCAD Maschine als Testumgebung für verschiedene Data Transport Networks dienen.

Als Platform wurde das Intel FPGA SDK for OpenCL zur Erzeugung von Hardware für die Intel HARP Hardware gewählt. Offene Fragen zur Code Erzeugung und Hardwaresynthese von OpenCL soll diese Masterarbeit beantworten.
1 Introduction

1.1 Motivation

To further increase compute power according to Moore’s Law, processor designers have not been able to rely on advancements in semiconductor and integrated circuit technology for years (Source: \[22\]).

The only solution to still increase the power of modern processors is to have multiple execution units that work in parallel. Where vector processors, vector extensions or graphics cards require a programmer to explicitly write parallel code, modern general-purpose processors transparently execute sequential programs in a parallel manner.

Superscalar processors of the current generation analyse the data dependencies of programs at run-time. Unlike pipelining, where all instructions were still performed in order in a single execution unit, this allows the processor to distribute instructions to multiple execution units or to reorder them to achieve higher throughput. This concept is called out-of-order execution or Instruction-Level Parallelism (ILP). State of the art superscalar processor architectures require a central set of registers all operations are referring to. While techniques like register renaming provide more scalability by introducing virtual registers, all of these still need to be kept in, and passed through, a central component.

This thesis implements a Synchronous Control, Asynchronous Data (SCAD) machine architecture. SCAD machines are a family of exposed-datapath architectures. Exposed-datapath architectures allow the compiler - or assembly programmer - to address functional units instead of performing operations that are scheduled by the processor. SCAD machines have no central register set. Instead, temporary data is stored in input and output buffers close to each functional unit. Programs are written as a sequence of move instructions of the form \texttt{lsu@out -> pu0@in0}. As the name implies, these transfer a data value from an output buffer to an input buffer. A self-routing interconnection network between all functional units called the Data Transport Network (DTN) is used for the data transfer.

From a software perspective, these moves are executed in a synchronous fashion. To achieve asynchronous execution and data transport, input and output buffers only store tags for issued instructions. When a data value is produced - or arrives via the interconnection network - it is matched to a tag. Values arriving in the input buffers are reordered and afterwards processed. The values appearing in output buffers are matched to the first available destination tag and sent to the corresponding unit. With a highly optimised and scalable DTN, this distributed storage is expected to achieve a higher scalability than register-based machines and maximise ILP.

For explicit data-parallel programming, a current trend is to supplement the software-programmable General Purpose Computation on Graphics Processing Unit (GPGPU) by Accelerators based on Field Programmable Gate Arrays (FPGAs). The parallel computations and decisions performed on FPGAs are only constrained by data dependencies and available logic gates \[40\]. With this, they are able to perform better in terms of delay and throughput than processors or accelerators relying on sequential software. In this thesis, the platform we build on will be OpenCL on the Intel Heterogeneous Architecture Research Platform (HARP) system. Intel HARP \[39\] is a platform...
for accelerated computing in data centres. FPGAs are connected to the servers’ Xeon processors via the QuickPath Interconnect (QPI) and take part in the servers’ cache coherence domain. This gives Accelerator Function Units (AFUs) running on the FPGA fast access to memory shared with a host application.

There are two different approaches to program the Intel HARP System. One is to use a Hardware Description Language (HDL) such as Verilog or VHDL to define the hardware circuits running on the FPGA. The other one is focused on the Intel FPGA SDK for OpenCL to generate Verilog from kernels written in OpenCL. The Open Compute Language (OpenCL) is a C-based language specifically developed to run on accelerators such as the general purpose GPUs of modern graphics cards.

A motivation to run OpenCL on FPGAs is the steep learning curve of directly designing hardware circuits compared to sequential programs. The Intel FPGA SDK for OpenCL intends to make the power of highly optimised parallel computation offered by FPGAs available to software programmers. This is achieved by translating sequential programs into deeply pipelined special execution units. Just like OpenCL kernels are generally executed in parallel, these units may have several parallel instances to increase throughput.

1.2 Main Goals

The goal of this thesis is to develop an implementation of the SCAD machine that runs as a co-processor to the main application processor and speeds up computation-intensive tasks. It is implemented in OpenCL for the Intel HARP platform, and serve as a case study to explore the implementation of programmable accelerators in OpenCL.

There are several open questions regarding the Intel FPGA SDK for OpenCL that this thesis intends to answer:

- Is the implementation of a programmable accelerator in OpenCL possible?
- How are the asynchronous components of the SCAD machine applicable to OpenCL?
- Are there limitations to consider such as a reduced efficiency regarding chip area?
- Will a SCAD machine implemented with the Intel FPGA SDK for OpenCL be portable to other OpenCL platforms such as NVIDIA graphics cards or Intel CPUs?

Another goal of this thesis is to provide a framework to test different DTNs under realistic conditions. The Embedded Systems group at the University of Kaiserslautern is working on different radix-based interconnection topologies that should achieve a better scalability than classical tile or tree-based solutions. A generic interface will be used to allow a quick exchange of DTN implementations for experiments.
1.3 Structure

This thesis is structured as follows: After the Introduction, Chapter 2 establishes the scientific context for this thesis by providing an overview of related publications. Chapter 3 will provide an in-depth explanation of the SCAD machine architecture. It will focus on the SCAD machine in general, and avoid the implementation-specific deliberations which are given in Chapters 5.

Information about the development platform will be conveyed in Chapter 4, which will discuss OpenCL in general and on FPGAs, and Section 4.4, where details on the Intel HARP platform will be explained.

Following these introductory chapters, the steps taken in design and implementation for this thesis are presented. Chapter 5 starts with an overview of the system architecture, choices made for the implementation and includes detailed descriptions of the architecture and low-level details of components.

After an overview of the synthesis process in Chapter 7, Chapter 8 presents the hardware generation results. Chapter 9 summarises and concludes the work of this thesis and is only followed by the future work in Chapter 10.
2 Related Work

This section provides an overview of the publications related to this thesis. As an introduction, the SCAD machine related work published by the Embedded Systems group at the University of Kaiserslautern is discussed. Afterwards, some alternative exposed datapath architectures will be introduced. Subsequently, OpenCL on FPGA will be discussed and followed by the further related topic of radix-based interconnection networks.

2.1 SCAD Machine

The SCAD machine architecture is a topic of ongoing research at the Embedded Systems group. Publications on the SCAD machine can be grouped loosely into the categories hardware architecture, code generation and interconnection networks.

In addition to translation from queue machine programs to SCAD machine programs, [36] is a first introduction to the SCAD machine architecture. [43] analysed the benefits of out-of-order execution within functional units.

Since development is still ongoing and not all design decisions are fixed, [11] compares the classical SCAD machine to two variants called the Statically Ordered SCAD (SO-SCAD) and Dynamically Ordered SCAD (DO-SCAD). SO-SCAD discards the reordering behaviour of the functional unit input buffers. This results in a latency-sensitive architecture and requires the compiler to know about timing behaviour and scheduling of the involved components and order instructions accordingly. A step in the opposite direction, DO-SCAD assigns each operation a tag that is used by functional units to match input values and find the destination and new tag for their output results. The conclusion of [11] states that the classical SCAD machine is the best choice for maximal ILP and scalability. "The SCAD machine", as it is found in this thesis, references the parameterised architecture described in Chapter 3.

[7] proposes special chaining semantics for buffers where loop kernels are set to run until a certain stop condition occurs. These chained operations are intended to run without further instruction decoding and thus reduce the delays caused by waiting for branch conditions. The programs and concepts in [7] were written and tested using an online simulator.

2.2 Exposed Datapath Architectures

The SCAD machine architecture was not developed in isolation. It draws inspiration from the works on other exposed datapath architectures. Some that have had an influence are the Mill architecture, the Transport Triggered Architecture (TTA) [21] and FlexCore [16].

The TTA presented in [21] allows the programmer to explicitly move data from functional unit outputs to registers and from there to functional unit inputs. The main difference to the SCAD machine is the use of registers where the SCAD machine skips
that intermediate step. Compiler techniques for TTA have been presented in [30]. An implementation at the Embedded Systems group in Kaiserslautern is presented in [38].

The Mill [19] utilises a bounded FIFO, called ”the belt”, as a replacement for a register file. The Belt implements Least Recently Used (LRU) semantics. All values produced are inserted at the head and old values are dropped off the tail end. In the Mill architecture, all operations refer to values addressed by their relative position on the Belt.

2.3 FPGAs

A prominent example for the large-scale deployment of FPGAs in datacenters is the Bing search engine by Microsoft. In the corresponding paper [37], the deployment of FPGAs for latency reduction in their ranking algorithm is described. The description ranges from the overall architecture, i.e., the partition of the FPGA into shell and application, the topology of groups of networked FPGAs to details of the ranking algorithm applied.

2.3.1 OpenCL on FPGA

An introductory work on OpenCL for FPGAs is the whitepaper [40] by Altera (now Intel). It encourages the use of FPGAs in general, namely their massive yet highly flexible parallelism and the ability to deeply pipeline operations. Furthermore, OpenCL as a language for FPGA programming is introduced as a high-level alternative to classical Hardware Description Languages (HDLs).

The compiler infrastructure and extensions used to generate Verilog from VHDL are introduced in [14]. It includes a description of the transformation performed on the LLVM Intermediate Representation (LLVM-IR) to produce pipelined circuits.

A variety of publications have been made that describe the use of the Intel FPGA SDK for OpenCL in special applications. [42] introduces an implementation of convolutional neural networks. In [41], the performance of the Smith-Waterman algorithm implemented in OpenCL for Xilinx FPGA is compared to that on CPUs as well as GPGPUs.

2.4 Radix-Based Interconnection Networks

An efficient interconnect for data transport is one of the core components of the SCAD machine. As such, the development of scalable, self-routing DTNs is a closely coupled topic of research for the Embedded Systems group.

Inspired by radix-based sorting networks and the results of [13] and [8], the self-routing Selector Tree [28] Network has been developed. This approach has been refined further in [15], [29] and [?].
3 The SCAD Machine Architecture

This section will introduce the SCAD machine architecture from a high-level perspective. Most of the details explained here are defined - or part of - previous works on the architecture. This is especially true for the SCAD machine simulator [5], compatibility with which is a basis for this thesis. After an overview of the SCAD machine as a whole, there will be a top to bottom explanation of the aspects involved. That explanation will start with the assembly language in Section 3.1 followed by a list of the basic functional units in Section 3.2. These two sections give the background information required for the example programs listed in Section 3.3. Afterwards, the more technical aspects of input and output buffers, the DTN and the MIB will be explained in Sections 3.4, 3.5 and 3.6. Finally, Section 3.7 discusses reconfigurability of the SCAD machine architecture.

Figure 3.1: Program View of a SCAD Machine.

The SCAD machine is an architectural concept for explicit data path programming. It consists of an array of functional units that send an receive data to and from each other through a Data Transport Network (DTN). An example for a SCAD machine from a programmers point of view is given in Figure 3.1.

Depending on the operations they perform, the functional units have specific inputs and outputs. For example, the universal processing unit (PU) takes a triple of opc, i.e., a number describing the operation to perform, and two operands as an input and returns the result of the computation at its output. The inputs and outputs of each functional unit are realised as buffers of a defined depth that temporarily store as well as send and receive data.

In assembly language, the SCAD machine is programmed using move instructions. Generally, each of these instructions moves a value from one output of one functional unit to an input of a functional unit. As an extension to the move from buffer to buffer, there are special instructions to send immediate values to an input. A detailed list of the assembly language instructions is given in section 3.1.2.

The following paragraphs are a description of the semantics of asynchronous be-
haviour as they are defined, used and implemented in this thesis. For a comparison of different SCAD machine architecture variants, see [11].

"Synchronous Control" in SCAD describes that, from a software point of view, all instructions are executed in order. "Asynchronous Data" is realised by the asynchronous hardware implementation of the input buffers and the DTN. While move instructions are issued in-order by a control unit, data is transferred only when available. This allows the SCAD machine to perform out-of-order parallel execution while observing the same semantics as a strictly sequential execution.

To achieve the asynchronous behaviour, input buffers reorder incoming messages from the DTN so they are handled in program order. A system diagram showing multiple functional units with separate connections for move instructions and data is given in Figure 3.2.

In contrast to register-based architecture, data in a SCAD machine is either stored in a buffer close to the component it was produced in or will be consumed by or it is spilled to memory. There is no central set of registers through which all processed data must flow. Instead, a self-routing DTN is employed to transfer the operands and results of all operations directly from their source to the functional unit they are needed at. With the improvements in radix-based interconnection networks as DTNs, this distributed storage and transfer is expected to perform significantly better in larger-scale.
3.1 Assembly Language

The assembly language used to program the SCAD machine is based on that of the SCAD machine simulator \[5\] on the website of the Embedded Systems group. A feature missing in the assembler language and buffer implementation of this thesis are the chained buffers introduced in \[11\]. Instead, for convenience, the linking to labels was added to the assembler syntax.

3.1.1 Assembly Language Example

```markdown
<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>// memory[0] = 77</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$0 -&gt; lsu@in0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$77 -&gt; lsu@in1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>st -&gt; lsu@opc</td>
<td></td>
</tr>
</tbody>
</table>
```

3.1.2 List of Instructions

Statements understood by the assembler:

- `pu0@out -> pu1@in0`: Move from output buffer "out" of functional unit "pu0" to input "in0" of "pu1".
- `$1 -> lsu@in0`: Immediate value to input buffer of functional unit.
- `$0 -> pc`: Moves the immediate value 0 to the program counter. This is an absolute jump to address 0.
- `st -> lsu@opc`: For some buffers, for example the LSU unit, the assembler translates operation mnemonics. This tells the LSU to perform a "store" operation with the address and value at the two other inputs.
- `(ld, 5) -> lsu@opc`: Instead of requiring a special "copy" or "duplication" unit, some data producing operations have a parameter that specifies the number output copies to produce. In this instance, the LSU is instructed to load a value and place 5 copies in its output.

Labels are linked by the assembler. After translation, label moves `<label> -> X` are translated to immediate moves `$<label address> -> X`.

- `pu0@out -> null`: Moves to null destroy values. This happens without transmission through the DTN and is used in loop teardown or after branches.
- `// Comment`: Everything between a `//` and the end of the line is ignored.

[https://es.cs.uni-kl.de]
3.2 Functional Units

This section introduces a basic set of functional units of which a SCAD machine is built. They are also described and implemented in the SCAD machine simulator available at [5].

While the functional units developed are compatible to those of the existing SCAD machine simulator, the semantics of some operations were changed to fit the system architecture (introduction of Chapter 5) or test aspects of the hardware synthesis.

3.2.1 Control Unit

The control unit is the one functional unit required for any operation of the SCAD machine. It is responsible for decoding the program and sending move instructions to the functional units involved. To this end, a Program Counter (PC) and memory access for reading the program are part of this unit. The control unit is connected to the DTN for sending the values of immediate moves and receiving branch conditions and targets. The two inputs are expected to be written in order by the program, first the target address, then the condition. When the move to branch condition is interpreted by the control unit, it sends the move instruction to the source functional unit and waits for both condition and target to arrive.

If the branch condition is \( \neq 0 \), the branch is taken, i.e., the pc is set to the branch target. Otherwise, execution continues with the following instruction.

3.2.2 Memory: Load-Store Unit

As the name implies, the load-store unit handles load and store operations to memory. Memory in the SCAD machine is organised as a continuous array of words starting at index 0.

Each operation takes a triplet \((\text{opcode}, \text{address}, \text{value})\) where \(\text{opcode}\) is either \(\text{st}\) or \((\text{ld}, <\text{copies}>\)). For a \(\text{st}\), i.e., store operation, \(\text{value}\) is written to memory at index \(\text{address}\). \((\text{ld}, 10)\) instructs the LSU to place 10 copies of the the value at memory position \(\text{address}\) in the output buffer.

3.2.3 Reorder Unit

The reorder unit is a places every input it receives into its output buffer. In essence, it implements a FIFO bounded by the depth of the buffer. As the name implies, this unit is an essential for cases when the order of output values differs from the order these values need to arrive at an input. In those cases, values needing to wait may be temporarily stored in the reorder unit. Another use, demonstrated in Listings 3.1 and 3.3, is the use of the reorder buffer as a temporary register that stores a single value.
3.2.4 General Purpose Processing Unit

The general purpose Processing Unit (PU) is an Arithmetic Logic Unit (ALU) that performs all computations in the current SCAD machine design. It consumes a triplet \((\text{opcode}, \text{left operand}, \text{right operand})\) and performs the operation defined by opcode. To produce multiple copies of the same result, the operand may be given in the form \((\text{mnemonic}, \text{copies})\). There is a single output buffer in which the results are placed.

Operations supported by the PU are:

- **Bitwise:** \(\text{andB}, \text{orB}, \text{eqqB}, \text{neqB}\)
- **Signed Integer:** \(\text{addZ}, \text{subZ}, \text{mulZ}, \text{divZ}, \text{modZ}, \text{lesZ}, \text{leqZ}, \text{eqqZ}, \text{neqZ}\)
- **Unsigned Integer:** \(\text{addN}, \text{subN}, \text{mulN}, \text{divN}, \text{modN}, \text{lesN}, \text{leqN}, \text{eqqN}, \text{neqN}\)

3.3 Example Programs

The following are move instruction programs used to test the SCAD machine implementation of this thesis. While the fibonacci sequence was written as part of this thesis, the Heron iteration is taken from [7], where it was developed using the existing SCAD machine simulator.

**Fibonacci** Listing 3.1 shows the first example built as a part of this thesis to test the correctness of move instruction execution. The idea to use the fibonacci sequence came from [38], where it was used to test the correctness of a TTA implementation.

An important part of this example is the explicitly noted loop invariant. Invariants are essential in the development of looping move instruction programs, because parts of the code expect very clearly defined buffer contents. The fibonacci program also serves as an example for the linking of symbols implemented in the assembler.

**Heron Iteration** The Heron iteration (Listing 3.2) is an algorithm based on newton iteration to calculate the square root of a number. The more readable MiniC implementation this was built from as well as an in-depth explanation of the steps performed is part of [7].

**Squares** Listing 3.3 is a quick test to check the input and output of memory through the LSU. The program iterates over a number of memory positions (given in line 2), and replaces each value by its square. This example also shows the flexible order of move instructions.
Listing 3.1: Fibonacci Example

setup:
1
2 // upper bound for sequence -> 2 copies of n
3 $255 -> pu0@in0
4 $0 -> pu0@in1
5 (orB, 2) -> pu0@opc
6 // loop counter $0 -> 3 copies of i
7 $0 -> pu1@in0
8 $0 -> pu1@in1
9 (orB, 3) -> pu1@opc
10 // 3 copies of fib(i)
11 $1 -> pu2@in0
12 $1 -> pu2@in1
13 (orB, 3) -> pu2@opc
14
15 // fib(0)
16 $0 -> rob@in0
17
18 loop:
19 // Loop invariant:
20 // pu0: [n, n]
21 // pu1: [i, i]
22 // pu2: [fib(i), fib(i)]
23 // rob: [fib(i - 1)]
24
25 // fib(i) -> output[i]
26 pu1@out -> lsu@in0 // addr: i
27 pu2@out -> lsu@in1 // value: fib(i)
28 st -> lsu@opc
29 // i = i + 1
30 $1 -> pu0@in0 // 1
31 pu1@out -> pu1@in1 // i
32 (addN, 3) -> pu0@opc // 3 copies of i + 1
33 // n == i implemented as (n - i)
34 pu0@out -> pu0@in0 // n
35 pu1@out -> pu1@in1 // i
36 (subN, 1) -> pu0@opc // branch condition: (n - i)
37
38 $0 -> pu0@in0 // 0
39 pu0@out -> pu0@in1 // n
40 (orB, 2) -> pu0@opc // n for next iteration
41
42 rob@out -> pu2@in0 // fib(i - 1)
43 pu2@out -> pu2@in1 // fib(i)
44 (addN, 3) -> pu2@opc // fib(i + 1) = fib(i) + fib(i - 1)
45
46 pu2@out -> rob@in0 // fib(i) -> fib(i - 1)
47
48 // Loop condition/branch
49 loop -> cu@in1
50 pu0@out -> cu@in0
51
52 cleanup:
53 pu0@out -> null pu0@out -> null pu0@out -> null
54 pu1@out -> null pu1@out -> null pu1@out -> null
55 pu2@out -> null pu2@out -> null rob@out -> null
3.3 Example Programs

Listing 3.2: Heron Example taken from [7]

```c
// Heron iteration adapted from
// Which is the master thesis
// "Application-specific Configuration
// of Exposed Datapath Architectures"
// by Sireesha Rudratah Basavaraju

// * Changed pu3 to pu1 to allow execution on
// scad machine with 3 PUs. (Lines 12 to 16)
// The operations of pu1 and pu3 had no overlap and
// were data-dependent.
// * Changed number to 812731^2 to test 64bit configuration,
// was 121.

//------- initialization -------
0: $660531678361 -> pu0@in0
1: $0 -> pu0@in1
2: (addN, 3) -> pu0@opc // zold = xnew

//-------- loop begins ----------
3: $660531678361 -> pu2@in0 // a for a / zold
4: pu0@out -> pu2@in1 // zold for a / zold
5: (divN, 1) -> pu2@opc // div is i on opcode
6: pu2@out -> pu1@in1 // a / zold for zold + a / zold
7: pu0@out -> pu1@in0 // zold for zold + a / zold
8: (addN, 1) -> pu1@opc // addition
9: $2 -> pu0@in1 // 2 for ( zold + a / zold ) / 2
10: pu1@out -> pu0@in0 // zold + a / zold for
11: // (zold + a / zold ) / 2
12: (divN, 4) -> pu0@opc // calculate znew
13: pu0@out -> pu1@in1 // zold for znew < zold
14: pu0@out -> pu1@in0 // znew for znew < zold
15: (lesN, 1) -> pu1@opc // calculate the condition
16: $3 -> cu@in1 // jump address
17: pu1@out -> cu@in0 // branch instr

//-------- loop ends -----------
18: st -> lsu@opc // store the result
19: $0 -> lsu@in0 // memory address is 0
20: pu0@out -> lsu@in1
```
Listing 3.3: Squares Example

```
// Number of elements
mov $256 -> rob@in0

loop:
    rob@out -> pu0@in0
    mov $1 -> pu0@in1
    mov (subN, 4) -> pu0@opc
    pu0@out -> rob@in0
    pu0@out -> rob@in0
    $0 -> lsu@in1 // value discarded by load
    (ld, 2) -> lsu@opc
    st -> lsu@opc
    pu0@out -> lsu@in0 // addr load
    pu0@out -> lsu@in0 // addr store

// mem[i] * mem[i]
    lsu@out -> pu0@in0
    lsu@out -> pu0@in1
    mov (mulN, 1) -> pu0@opc
    pu0@out -> lsu@in1 // addr store

// Loop condition/branch
    loop -> cu@in1
    rob@out -> cu@in0
    rob@out -> null
```
3.4 Input and Output Buffers

Reordering input buffers are a fundamental for the asynchronous data flow of the scad machine. Each buffer has a depth $N$, which gives the number of slots data may be stored in temporarily.

When a move instruction is issued, tags are placed in slots of the relevant input and output buffers to keep asynchronously arriving and produced data in order. An output buffer stores tags and data values in two independent FIFOs and, when there is a tag and a data value available, transmits them as a message through the DTN explained in the following section.

Data and tags in an input buffer are more connected. The tags are stored in a FIFO in the order they are issued, but incoming data messages need to be matched to the tag they belong to.

Once a pair of $(tag, value)$ is at the head of the input buffer, it is available for consumption from the functional unit.

Figure 3.3 shows how an incoming message is placed into the slot corresponding to its tag. An invariant of the SCAD machine is that every message arriving at an input buffer is expected, i.e., there is a slot reserved for that data value. This allows all arriving data messages to be handled in a non blocking fashion.

3.5 Data Transport Network (DTN)

The DTN is a self-routing network that transports data packets. In a $N \times N$ Network, there are $N$ input ports and $N$ output ports. Packets submitted to any input port will be routed to the output port they are addressed to. As stated in Section 2.4, the research in interconnection networks is still ongoing.

The DTN connects the buffers of all functional units with each other. When a move instruction is issued and an output buffer has a value to send, a message consisting of $(sender, recipient, data)$ is given to the DTN. $sender$ and $recipient$ both are tuples $(unit, buffer)$ that exactly identify the buffers between which data is transferred. While the $recipient$ is required for the network to route packets, $sender$ is a necessary information for the input buffer to ensure ordering.

A requirement of the DTN is that messages to the same input buffer that translate to the same tag, i.e., from the same unit’s output, must arrive in order.
3.6 Move Instruction Bus

The Move Instruction Bus (MIB) is an abstract component that transfers move instructions from the control unit to the relevant functional units. One task of the MIB is to ensure that move instructions are acknowledged by the sender before the corresponding data packet arrives.

Different approaches to the architecture of the MIB have been considered. One of these, implemented in this thesis, is a point-to-point connection between the control and each other functional unit.

3.7 Reconfigurability

To conclude this chapter: The SCAD machine architecture is developed to be easily adapted to different applications.

One aspect of this is that new functional units have a fixed and powerful interface. All that is required to add a special purpose unit to the SCAD machine architecture is the consumption of input values and production of outputs. These inputs and outputs are then connected to an existing SCAD machine by the generic input and output buffers.

As stated in [12], this also requires the compiler that produces move instruction programs to know how many output values are produced for each input. At the point of writing, a compiler that produces assembly code for a specific configuration is in development. (See [9], [10] and [3])

The generation of efficient DTNs for an arbitrary number of endpoints is another problem for which solutions are being developed [28, 27, 15, 29]. Combining these techniques may allow future implementations to generate efficient and powerful processors while remaining easy to configure to fit various applications.
OpenCL and Intel HARP

The SCAD machine implemented in this thesis is written in OpenCL for the Intel FPGA SDK for OpenCL on the Intel HARP platform.

This chapter will give a short introduction to OpenCL and the programming model involved. Following a short comparison of the C and C++ APIs, the use of OpenCL on FPGAs will be motivated and explained in detail for the Intel FPGA SDK for OpenCL.

OpenCL is a C-based language to write software, and more recently hardware, for parallel computations with a focus on acceleration hardware. It is a standard managed by the Khronos Group and supported by various hardware and software vendors [31]. Platforms for OpenCL are not limited to GPUs. Available SDKs include ones for Intel processors and Xilinx as well as Intel FPGAs [3].

4.1 Programming Model

This section introduces the programming model of OpenCL. It does so in a bottom-up manner, starting with kernels, the accelerated functions called by a host application, and continues on the hierarchy until the topmost entity, the platform, is reached.

A kernel is the fundamental interface between an application and the OpenCL code on an accelerator. Being part of the OpenCL program, but callable from the host application, it defines the entry point for any OpenCL execution. Kernels may be started as a task of several work-groups each consisting of multiple kernel threads running in parallel, or as a single work-item.

Data is exchanged between host and device by pointers to buffers, specially allocated device-accessible memory regions, as parameters to a kernel execution. The declaration of kernel parameters and the allocation on the host allows a developer to restrict kernel access to a buffer to read-only, write-only or allow full read-write access. This is a mechanism to give the OpenCL compiler more information on the use of memory and allows it to produce more efficient programs or hardware.

Executions of a kernel are managed through command queues. An application can create one or more command queues to execute kernels on a device. OpenCL Command queues are designed with support for asynchronous operations. Most commands are either non-blocking per default or have an option to perform in a non-blocking manner. To support this, there are event markers that may be inserted into the stream of commands and allow the host application to wait for a specific position in the queue to be reached.

Devices in OpenCL correspond to a physical device such as graphics cards or FPGAs. All Devices have a set of parameters, such as available memory, maximum work-group size, endianess and availability of an online compiler. In general, all devices of a vendor are grouped into one platform.

Platforms are the entry points to a specific vendors OpenCL implementation. Each installed OpenCL driver registers a shared library with the Khronos Installable Client Driver (ICD) framework. To use OpenCL, applications perform a search of the registered platforms and choose one or more with devices fitting the requirements.

OpenCL programs are either loaded as source code and compiled by the OpenCL
implementation, were compiled offline and are loaded as bytecode that needs translation or direct binary code such as the bitstream for an FPGA.

4.1.1 C++ API

For this thesis, C++ was chosen as the language for the implementation of all host-side executables and libraries. The primary reason for this was the verbosity of the C library when compared to the C++ API.

Listing 4.1: Printing the platform name in C

```c
char buffer[1024];
c1GetPlatformInfo(platforms[i], CL_PLATFORM_NAME, 1024, buffer, NULL)
printf("Platform name: \%s\n");
```

Listing 4.2: Printing the platform name in C++

```cpp
std::cout << platforms[i].getInfo<CL_PLATFORM_NAME>() << std::endl;
```

Another reason for using C++ was the added robustness when using smart pointers and following the Resource Acquisition Is Initialisation (RAII) principle.

4.2 Motivation for OpenCL on FPGAs

There are two main reasons that motivate the use of OpenCL as a hardware description language:

**Ease of Learning** In contrast to pure hardware description languages such as VHDL or Verilog, OpenCL programs are easier to learn. The main reason for this is that OpenCL is based on C and OpenCL programs are very similar to multi-threaded programs written in other imperative languages, while VHDL and Verilog require the developer to be aware of and make use of the signal flow in combinatorial circuits.

**Reusability** A fundamental concept of OpenCL is that programs may be compiled for different hardware architectures. This allows applications to choose from available acceleration devices without a developer having to prepare special code for each platform.

Also, being based on C, algorithms written in the C language have a chance to be usable in OpenCL as well. There are some exceptions to this portability, e.g. special semantics of pipes and channels on the Intel FPGA platform.

4.3 Intel FPGA SDK for OpenCL

The platform chosen for this thesis is the Intel FPGA SDK for OpenCL.

To run OpenCL programs on an FPGA, the SDK translates them to pipelined execution units in the hardware description language Verilog. Together with interfaces for the host’s command queues and memory interfaces, these are then compiled into FPGA bitstreams. Bitstreams are loaded through the standard OpenCL API like other precompiled binaries.

The different memory models of OpenCL are translated to registers, block RAM and DRAM. Hardware synthesis may decide to place private arrays into block RAM to save LUTs, and may even ignore the `__attribute__((register))` directive, so it is advised to check if the output meets the requirements.
4.3 Intel FPGA SDK for OpenCL

4.3.1 Available Documentation

In addition to the standard documents on OpenCL [33] and the C++ wrapper for OpenCL [18], there are several manuals available for the Intel FPGA SDK for OpenCL.

**Getting Started Guide:** [25] gives instructions for the setup of a development environment, the synthesis server and hardware device.

**Programming Guide:** [26] is an extensive development guide for the Intel FPGA SDK for OpenCL. It describes the overall development process, non-standard extensions to the language and their use, gives advice on how to structure kernels, emulate them, compile them and use advanced features such as the Avalon Streaming (Avalon-ST) Interface.

**Best Practices Guide:** [24] is intended for developers that want to optimise their OpenCL code. To this end, it gives a detailed overview of the reports generated during synthesis, how to read them and solve common problems. There are several chapters that describe general strategies to avoid inefficient code or hardware generation as well as an introduction to the profiling feature of the SDK.

**Design Examples:** Some design examples are given on [2]. These are built to show general implementation details and special solutions for problems using the SDK.

The complexity of these examples ranges from vector addition to multi-kernel FFT and JPEG decoder applications that demonstrate the use of channels to connect multiple kernels.

**Altera Forums:** The OpenCL section of the Altera Forum [1] has an active community that answered several questions regarding features and results for this thesis within less than a day each.

4.3.2 Concurrent Kernel Execution

For the deployment on FPGAs, the Intel FPGA SDK for OpenCL explicitly allows running multiple kernels in parallel. The mechanism used for this is the creation of more than one command queue, each of which starts one kernel.

This is a fully asynchronous interface, the host is free to enqueue buffer writes and kernel executions to multiple command queues in parallel and have the OpenCL runtime schedule their execution.

4.3.3 Pipes and Channels

Pipes and Channels are the main means of data transport between kernels used in this thesis. From a programmer perspective, they may be described as stalling FIFOs.

Pipes are a standard OpenCL 2.2 [31] feature that allows using a section of memory as a FIFO for communication between kernels. Channels are an Intel OpenCL extension for the OpenCL SDK [26].

While the standard definition of pipes describes them as memory operations, the hardware synthesis employed in Intel FPGA SDK for OpenCL translates both pipes and channels to direct, on-chip interfaces without needing memory. To achieve this, the compiler needs to know the endpoints of each pipes connecting during translation. It does this by matching the names of pipes across multiple kernels.
Channels are FPGA-specific FIFOs between different kernels. They have a depth, i.e., a number of elements that are buffered before stalling. The basic operations performed on a channel are blocking read and write as well as non-blocking read and write. An example for a non-blocking read:

```c
bool has_read;
scad_data input = read_channel_nb_altera(input_channel, &has_read);
if (has_read) {
    /* Handle packet */
}
```

There are two restrictions for the use of channels in OpenCL owed to implementation details of the code generation process:

- **Single call-site**: Writes to, or reads from, one channel may only be done at one place in code. One strategy used to reduce the number of writes to one is the split of existing code into separate sections for calculation and output.

- **No dynamic indexing into array of channels**: When channels are grouped into an array, e.g., all inputs and outputs of an interconnect, access to these may not be dynamic. The programming guide suggests using a switch/case construct instead:

  ```c
  switch (gid)
  {
    case 0: value = read_channel_intel(ch[0]); break;
    case 1: value = read_channel_intel(ch[1]); break;
    case 2: value = read_channel_intel(ch[2]); break;
    // ...
  }
  ```

  Decision for this thesis fell on a `#pragma unroll` `while` loop, which allows a fully dynamic number of elements in the array without use of macros.

Listing 4.4: Workaround for dynamic indexing into channel arrays as used in this thesis.
This approach has the benefit that it is more compact to write and requires no changes to the code should the number of different channels \( (N) \) change. However, the product of synthesising these two approaches has not been compared. If there are special transformations applied to the switch/case statement during code optimisation and hardware generation, the results may differ.

### 4.3.4 Autorun Kernels

Concurrent kernels and channels allow the organisation of system components as a data-flow process network (DPN). A comfort feature that saves the labour of having to explicitly start every single component of such a DPN are autorun kernels. Kernels that do not need parameters by the host, e.g. an interconnect, an intermediate computation or reorder unit, are declared as autorun.

```c
__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
__kernel void interconnect() {
    # ifdef EMULATOR
    printf("interconnect\nstarting.\n");
    # endif
}
```

Unlike host-started kernels, autorun kernels require no logic for interaction with an OpenCL command queue. This reduces the complexity of these components, allows the hardware synthesis to further optimise parts, and saves FPGA space.
4.4 Intel Harp

The HARP is a hardware platform with a software framework for the development of tightly coupled CPU/FPGA interaction. It uses the Intel Quickassist Architecture and the Accelerator Abstraction Layer (AAL) as well as FPGAs connected to the host via the QuickPath Interconnect (QPI) for fast and coherent shared memory access.

Intel HARP is targeted at acceleration of applications running in data centres. The direct connection from the FPGA to the application processor’s memory controller and cache coherency via QPI allows fast and coherent access to shared memory. An analysis of the performance for pointer-chasing in a linked list, suggests that tightly-coupled processor-FPGA combinations can provide a significant performance boost for some applications.

[39] is a presentation that shows additional technical details and presents the concept of a cloud-based store on which third-party developers market their Intellectual Property (IP). Another presentation, gives the use in deep learning frameworks, accelerating Open vSwitch and low-latency high frequency trading as examples of applications for the Intel HARP.

4.4.1 Intel Quickpath Interconnect

Intel QuickPath Interconnect (QPI) is an interconnect for processors, IO hubs and recently FPGAs, that is used in multi-processor Xeon Servers. QPI allows a fast routed exchange of messages between components and is primarily used for a shared access to each components memory. It consists of multiple bidirectional point-to-point links between these components and connects the different memory controllers into one shared cache coherency domain. QPI extends the MESI coherence protocol by an additional Forward (F) state that allows caches to relay clean cache lines to each other. In extension to that, there are multiple layers including physical distortion correction, routing, link fail-over and CRC.
4.4 Intel Harp

For the development of custom AFUs, the desired functionality has to be defined in a HDL or, by using the Intel FPGA SDK for OpenCL introduced in Section 4.3, in OpenCL. For shared memory access, the Cache Coherent Interface (CCI) is included in the hardware synthesis process. It is implemented as an IP core that is linked against the AFU and is responsible for DRAM access and QPI communication with CPUs or other FPGAs.

Intel QuickAssist and Accelerator Abstraction Layer (AAL) [35] form a framework for the hardware acceleration of computation-expensive tasks in servers. They define a set of generic interfaces and drivers that allow applications to pass descriptors for tasks to special acceleration devices that then process it and signal task completion to the host. [23] lists encryption and zlib compression as example applications for Intel QuickAssist.
This chapter will describe the high-level architecture of the SCAD machine architecture as it is implemented in this thesis. Unlike Chapter 3, this includes technical and implementation-specific aspects with a focus on solutions found in this thesis. After the overview of the whole system, and the motivation for the development of an accelerator, the process used to configure various different SCAD machines is presented. Then the fundamental building blocks, i.e., kernels, channels and the fundamental data types, are explained. The following sections demonstrate technical details important for the implementation of various components. Finally, the host API and test applications are introduced.

An overview of the fundamental system layout is shown in Figure 5.1. For this thesis the SCAD machine is modelled as a network of asynchronous components, or threads, that are connected to each other via bounded FIFOs. All separate components are implemented as OpenCL kernels, which are allowed to run in parallel in the Intel FPGA SDK for OpenCL. Communication between components is realised through channels. (See Section 4.3.3 for details on channels.) These are used similar to signals in VHDL, i.e., wires connecting input and output ports of different modules.

The control unit and LSU are started by the host application or library while all other kernels are permanently running. These two components are passed pointers to, and
lengths of, memory buffers for program and SCAD machine memory contents. Once
the control unit is started, it will start execution of the program it was passed. The
LSU will be started with input data and terminate to return the results to the host
when the program terminates. The mechanism used to transmit the termination signal
is explained in Section 5.7.4.

The MIB is no separate component but an integral part of the control unit. It
has point to point channels to every input and output buffer through which move
instructions are sent. All buffers are also connected to the DTN via channels that
transport the data packets. Section 5.5 will show the common data types used in all
exchanges between functional units as well as the host. Functional units, including the
control unit at address 0, are addressed by a global index. Names will be translated to
indices by the Assembler (Section 5.12).

5.1 Motivation for Use as an Accelerator

Implementing a general-purpose CPU from scratch is a complex task. Modern operat-
ing systems (OSes) require privilege separation, virtual memory and interrupts as well
firmware such as the BIOS or EFI to function. Using the SCAD machine as an accel-
erator alleviates the need to support an OS. Instead, the OS runs on an application
processor (e.g. x86, ARM) and uses existing drivers to first load the SCAD machine
onto an FPGA and then passes programs and memory buffers to that. This allows us
to design and study the SCAD machine architecture as a pure computational platform
first and perform our experiments in isolation.

Another reason for the use as an accelerator is that the platform we chose for this
implementation, OpenCL on the Intel HARP system (See Section 4.4), is specifically
built for the acceleration of datacenter applications.

5.2 Configuration

To produce multiple SCAD machine configurations without difficult the need to main-
tain different instances of the same code, a configuration tool was developed. As shown
in Figure 5.2, this tool takes a platform configuration file (Example: Listing 5.1) and a
folder with implemented component templates and emits a folder of configured compo-
nents that are all included in a <configuration name>.cl. To configure the templates,
simple string replacements such as $\{NAME\}$ to pu0 are used.

As an alternative, C preprocessor macros were also considered. The decision to
not attempt configuration based on macros was made to avoid more complicated and
difficult to debug code.
Figure 5.2: Configuration Process of this Framework.
Listing 5.1: Source of “multimem” configuration

```xml
<processor name="basic_multimem" buffersize="5">
  <interconnect>
    <name>interconnect</name> <size>8</size>
    <implementation>interconnect_trivial</implementation>
  </interconnect>

  <unit>
    <name>cu</name><number>0</number>
    <type>cu</type><implementation>control_hardware</implementation>
    <!-- Comma-separate list of unit numbers and buffers
to send sync markers to when program is done.
FORMAT: {unit, buffer}
Disable like this:
  <parameter><key>SYNC_TO</key>
    <value>{0,0}</value></parameter>
--><parameter><key>SYNC_TO</key>
    <value>{1,1},{2,2},{3,2},{0,0}</value></parameter>
  </unit>

  <unit><name>lsu</name><number>1</number>
    <type>lsu</type><implementation>lsu_scratch</implementation>
    <parameter><key>MEMORY_SIZE</key>
      <value>512</value>
      <!--512*scad_data--></parameter></unit>
  <unit><name>lsu_input</name><number>2</number>
    <type>lsu</type><implementation>lsu_input</implementation> </unit>
  <unit><name>lsu_output</name><number>3</number>
    <type>lsu</type><implementation>lsu_output</implementation> </unit>
  <unit><name>rob</name><number>4</number>
    <type>rob</type><implementation>reorder</implementation> </unit>
  <unit><name>pu0</name><number>5</number>
    <type>pu</type><implementation>processing_basic</implementation> </unit>
  <unit><name>pu1</name><number>6</number>
    <type>pu</type><implementation>processing_basic</implementation> </unit>
  <unit><name>pu2</name><number>7</number>
    <type>pu</type><implementation>processing_basic</implementation> </unit>
</processor>
```
5.3 Kernels

The OpenCL kernels are used as an asynchronous thread or process each. Kernels are either started automatically (See Section 4.3.4), or by the host application via command queues. All kernels that have no run-time parameters, i.e., every component but the control unit and LSU, is declared as an autorun kernel to save space and increase performance. An added benefit is that this removes the need for the host application or library to start all kernels explicitly.

Most autorun kernels have a structure similar to the following:

```
__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
__kernel void component() {
    while(true) {
        // * Non-blocking reads from channels.
        // * Operation on data.
        // * Non-blocking writes to channels.
    }
}
```

For functional units there are two possible configurations of the input and output buffers. One, having input and output buffers integrated into the kernel of the functional unit, is shown in Figure 5.3. The second variant also shown is to have input buffers and output buffers as a separate kernel each. They are then connected to the functional unit via extra channels, one for each input and output. Code for these two configurations will be shown in Section 5.7.3.

5.4 Channels

Channels are the basic mechanism for components of the SCAD machine to exchange data. Each channel is a bounded FIFO that is generally written to in one kernel and read from by another kernel.
5 Architecture and Implementation

To connect all functional units, the control unit and the DTN, global arrays of channels have been defined. Each functional unit, including the control unit, uses the channels at the index equal to its address for communication.

```c
// MIB
channel struct scad_instruction
    channel_move_instructions_to [UNIT_COUNT];

channel bool
    channel_move_instructions_to_ack[UNIT_COUNT];

channel struct scad_instruction
    channel_move_instructions_from [UNIT_COUNT];

// DTN
channel struct scad_data_packet
    channel_to_interconnect [UNIT_COUNT];

channel struct scad_data_packet
    channel_from_interconnect[UNIT_COUNT];
```

Input and output buffers and the DTN have a highly asynchronous and timing-dependent behaviour. For this, they rely on non-blocking reads and writes introduced in Section 4.3.3. With external buffers, functional units may be implemented with only blocking channel reads or writes as shown in Figure 5.4. Further details will be given in Section 5.7.3.
5.5 Fundamental Data Types

All data structures used between the host and components of the SCAD machine have been defined in one common header file.

The most basic data type is \texttt{scad\_data}. It is a union of an integer type (\texttt{cl\_ulong} or \texttt{cl\_uint}), a floating-point type (\texttt{cl\_float} or \texttt{cl\_double}), and a tuple with two elements used for (\texttt{<opcode>, <copies>}). The three different possible types have an equal length to maximise the use of buffers and signal busses. Only the \texttt{scad\_data} and \texttt{scad\_address} types and the assembler as well as the PU need to be changed to switch the SCAD machine between 64 bit and 32 bit word width.

All data types need to be declared so that both host and device order and align all parts the same. This is required to access binary data from a common memory, e.g. the program for the control unit. As stated in Sections 5.9.1 and 5.9.2 of [26], all structs declared in the common header file are declared with \texttt{__attribute__((packed))}. This instructs the compiler to not apply any padding to the members of each struct which may make the binary layout differ between host and device. Also, the compiled bitstream uses little endian binary representation for data types larger than one byte, which matches the default behaviour on x86 processors.

5.6 Data Transport Network

The DTN receives packets from each endpoint through the channels in \texttt{channel\_to\_interconnect[]} and routes them, according to their destination address, to the correct endpoint in \texttt{channel\_from\_interconnect[]}. Since channels in the Intel FPGA SDK for OpenCL are bounded and writes are blocking when no space is available, DTNs written for the framework of this thesis have to support stalling unless timing guarantees of input buffers are assumed. Only a trivial DTN was implemented for this thesis.
5.7 Buffers

The input and output buffers of each functional unit receive move instructions from the control unit and send as well as receive data packets from the interconnect. In the input buffer, the matching of incoming data packets to previously received move instructions allows the functional units to perform all operations in program order. The output buffers are less complex, requiring only two independent FIFOs to produce data packets out of move instructions and data values.

Example implementations using the input and output buffers are given in Section 5.7.3, Listings 5.3 and 5.2.

5.7.1 Input Buffers

Most functional units have more than one input buffer. All of these share one MIB and DTN connection and reception of move instructions or data packets is handled in a common loop.

Each input buffer consists of one ring buffer for the move instruction tags and a linked array of data values. When a move instruction arrives and there is a free slot at the end of the ring buffer, the destination address is stored as a tag to match against the sender of incoming data packets. Then the end pointer is incremented with wrap-around and - if it equals the start pointer, the full flag of the buffer is set to 1.

Once the corresponding data packet arrives, it is placed in the data buffer with the index equal to the first matching tag. When the tag and data at the start of the input buffer are valid, buffer_input_has_data() returns true and the data may be read via a buffer_input_peek() operation or read and removed via buffer_input_pop(). An exception to this is the reception of sync markers described in Section 5.8.1. When a move instruction with a source address of RESERVED_ADDRESS, which is set to (-1, -1), arrives, the input buffers automatically mark the corresponding data slot as valid. Functional units or components that react to sync markers use buffer_input_has_marker(...) to check for a sync marker at the start of the buffer before reading values.

Figure 5.7: Input Buffer Structure - Ring Buffer and Data Slots.
5.7.2 Output Buffers

The output buffers consist of two ring buffers with a common start. Since the data produced by the functional unit does not need to be reordered, these ring buffers are only used in combination when sending data packets. When a move instruction arrives and there is at least one space left in the destination ring buffer, it is stored at the end and the pointer is advanced. Data is handled in the same manner. The output handling routine checks if the start of both ring buffers has a destination address and data available and sends out a data packet if that is the case.

5.7.3 Usage

The buffers were implemented as a set of C functions. A design criteria was to keep the duplicate code required for their use in functional units to a minimum. Listing 5.2 shows how the buffers are integrated into the same kernel the function is implemented in. This approach is suitable for functional units without host interaction. A load-store unit or the control unit however may need to be restarted by the host several times while the SCAD machine is running. To keep the state of the input and output buffers, the approach outlined in Listing 5.3 was chosen. Furthermore, having the input and output buffers in separate kernels has the benefit of reducing the complexity of each part of a functional unit. The main kernel of the functional unit, which, in this case, is started by the host, only performs blocking reads and writes on channels.
Listing 5.2: Buffer Integrated into Functional Unit.

```c
void kernel unit_f() {
    // Ringbuffers with indices and "full" flag.
    input[1];
    output[1];

    // Address, number of buffers and pending move.
    input_management = buffer_input_init(ADDRESS, 1, input);
    output_management = buffer_output_init(ADDRESS, 1, output);

    while (true) {
        // Nonblocking receive of moves and data
        scad_input_handle(ADDRESS, &input_manage, input);

        if (!buffer_output_to_full(&output[0]) // Room in output?
            && buffer_input_has_data(&input[0])) { // Message to handle?
            scad_data data = buffer_input_pop(&input[0]);
            scad_data result = f(data); // INSERT FUNCTION HERE
            buffer_output_push_data(&output[0], result);
        }
        scad_output_handle(ADDRESS, &output_manage, output);
    }
}
```

5.7.4 Message Acknowledgement

During implementation and when debugging in emulation it became clear that the first iteration of the input buffers are susceptible to a race condition. These occur when the input buffers receive a data packet for a move instruction that has not been registered yet. In that case, the input buffer would fail to handle the message.

A first solution is to have the control unit wait for the recipient of each message to acknowledge the move instruction. That way, there is an invariant that each message sent through the interconnect is expected by its recipient. The second solution considered is to stall data reception when a message with an unknown sender is received. This was not done because the resulting stalls of all data to that input buffer and the possibility of stalling unrelated messages on the interconnect was considered too expensive. Having separate channels to acknowledge every incoming move was chosen for this implementation.
Listing 5.3: Separate Buffer in Extra Kernel.

```c
channel scad_data channel_unit_input;
channel scad_data channel_unit_output;

void kernel unit_input() {
    input[1];
    input_management =
        buffer_input_init(ADDRESS, 1, input);

    while (true) {
        scad_input_handle(ADDRESS, &input_manage, input);
        if (buffer_input_has_data(&input[0])) {
            if (write_channel_nb_altera(channel_unit_input,
                buffer_input_peek(&input[0]))) {
                buffer_input_pop(&input[0]);
            }
        }
    }
}

void kernel decrement() { // functional unit kernel
    while (true) {
        scad_data in = read_channel_altera(channel_unit_input);
        scad_data out;

        // scad_data is a union of integer, float and (opcode, copies)
        out.integer = in.integer - 1;

        write_channel_altera(channel_unit_output, out);
    }
}

void kernel unit_output() {
    output[1];
    output_management =
        buffer_output_init(ADDRESS, 1, output);

    while (true) {
        if (!buffer_output_to_full(&output[0])) {
            // Check for data to output
            bool data_read;
            scad_data data = read_channel_nb_altera(channel_unit_output,
                &data_read);

            if (data_read)
                buffer_output_push_data(&output[0], data);
        }

        scad_output_handle(ADDRESS, &output_manage, output);
    }
}
```
5.7.5 Sync

A mechanism developed to deploy the SCAD machine on hardware are special sync markers. All components of the SCAD machine run in parallel threads. Some, in all configurations of this thesis only control unit and LSU, are started by the host to allow an application to pass parameters to those. While the control unit has a fixed condition for termination, i.e., the end of the program, the LSU should terminate with the program as well.

**sync** markers are move instructions with a source of \((-1, -1)\), which is an invalid address, that allow input buffers to detect and treat them in a special manner. Unlike transfer through a separate channel, a move instruction with special semantics adheres to program order. Shown in Figure 5.9, a LSU performs all operations scheduled ahead of a sync (purple) before it terminates, and leaves the following instructions (yellow) in the input buffer for the next execution.

5.8 Control Unit

The first iteration of the control unit was developed without any regard for hardware synthesis. It is started with a pointer to the buffer containing the program and the
length of the program as parameters. The main loop executing the program consists of a while loop that terminates when the PC runs out of bounds and a switch/case handling the different operations.

```c
__kernel void ${NAME}(read_only __global
    struct scad_instruction program[],
    read_only cl_uint program_length) {

    // ...

    // Run program.
    while(pc < program_length) {
        struct scad_instruction instr = program[pc];
        switch(instr.op) {
            case SCAD_MOVE_PC:
                pc = instr.immediate.integer;
                break;
            case SCAD_MOVE:
                // ...
                break;
            case SCAD_MOVE_IMMEDIATE:
                // ...
                break;
            default:
                // Handle error.
                break;
        }
    }
    // Send sync instructions to ${SYNC_TO}.

    // ...
}
```

### 5.8.1 Sending Sync Markers

As described in Section 5.7.5, functional units that are started by the host and have no intrinsic termination condition require sync instructions to be sent to them when program execution terminates. For this thesis, the sync instructions were hard-coded into the control unit, i.e., there is an array of sync instructions to send to each unit that the control unit iterates over when the program has terminated. An example for this array is shown in Listing 5.1 of Section 5.2.

```xml
<parameter><key>SYNC_TO</key>
    <value>{1,2},{2,2},{3,2},{0,0}</value></parameter>
```
5.8.2 Hardware Instruction Set

Most assembly instructions introduced in Section 3.1 are syntactic sugar that is translated to immediate moves. For example, processing unit or load-store unit opcodes are assumed to be translated to their numeric representation that are all handled by the same basic immediate move instruction. Thus, the implemented binary instruction set format supports only the following basic operations:

- **Move to pc**: A special immediate move. Replaces PC with immediate value.

- **Immediate move**: Places an immediate value in the output buffer of the control unit and issue a move instruction to the corresponding recipient.

- **Move**: General move instruction, has a source (unit, buffer) and a destination (unit, buffer).

Unlike the definition of other publications or the SCAD machine simulator, this variant of the control unit does not send immediate values through the MIB but instead relies on the DTN for all data transport. The decision for this has been motivated by several factors: One is that the signal lanes and registers required to transport the data to each functional unit would put a non-negligible burden on our available FPGA space. The second is that the handling of move instructions that include data would have introduced additional logic into the input buffers. This was considered a problem to avoid, since these are responsible for a large part of the complexity in implementing the SCAD machine and should remain as easy to understand and maintain as possible. Also, having two separate transport mechanisms for data is redundant and, unless it is strictly necessary to reach performance goals, wastes space on the FPGA usable for other features.

5.8.3 Second Iteration Control Unit

The first iteration of the Control unit is a direct sequential implementation of the required semantics.

It proved to be unusable in hardware synthesis. The main reason for this is that the first iteration was developed and tested in emulation. As will be explained in Section 7.1, there are significantly different constraints for kernels in emulation and in hardware synthesis. One of these is that writes to and reads from channels must only have one call site, i.e., only happen at one point in code.

To build a control unit usable for hardware synthesis, a second iteration was written that splits the evaluation of each instruction into different steps:

- Instruction is read at PC.
- Determine move instruction to send.
- Send move instruction to relevant functional units.
- Pass immediate value to output if necessary.
- Calculate new PC.
- If PC reaches end of program, switch state.
5.9 Load-Store Unit

Besides the control unit program, the load-store unit (LSU) is the only mechanism to transfer data from host to device and device to host. Also, all temporary data that does not fit into buffers or the reorder unit is spilled into memory.

Care was taken to explore the different possibilities to realise memory within the SCAD machine. There are three basic concepts of memory access through a load-store unit that were devised and implemented within the SCAD machine.

The first is a single LSU that handles access to one Region of DRAM allocated by the host application (Figure 5.11).

As mentioned in Chapter 4, OpenCL allows the description of memory buffers as write-only, read-only and read-write, which restrict the OpenCL kernel in its access. The implementation of this thesis serves as a test if memory access which is restricted to read- or write-only leads to less complex and thus more efficient hardware. To test whether this will improve performance, separate LSUs for input and output were implemented.

These are accompanied by a scratchpad LSU that offers access to a region of device-only memory for temporary storage that requires reads and writes. Depending on the synthesis, this scratchpad memory may reside in slower DRAM or fast on-chip Block RAM.

Because the SCAD machine developed in this thesis will be tested as an accelerator which performs computation on a stream of data, streaming input- and output memory units were implemented as well. The streaming input unit iterates over the input memory and places all data words into its output buffer. The output unit successively writes all values received through its input buffer into the output memory. Both will terminate when they reach the end of their allocated memory, which allows the host to read back results and restart them with new data while the SCAD machine program
continues running.

5.10 Host Application API

The host application API developed for the SCAD machine consists of three basic components. The description, which contains information used by the assembly library to link move instructions to the correct addresses, and the machine library, which performs most OpenCL operations.

5.10.1 Description

The machine description contains the names and addresses of all functional units in the current configuration. Outside of the configure tool presented in Section 5.2, it is primarily used to give the assembly library a map of addresses for functional units and buffers. Line 44 of Listing 5.4 shows that the library loads all information from a configuration file without exposing the internal format used.

44    processor_description proc(description_filename);

5.10.2 Assembly

The assembly library performs all operations required to turn assembly language source code into a binary representation suited for the running control unit. This includes the linking of labels to fixed addresses and the unit and buffer name to address translation via the configuration description. An example for the use of the assembly library is shown in lines 52 to 55 of Listing 5.4:

52    scad::assembly assembly(proc);
53    assembly.parse(assembly_src);
54    std::vector<struct scad_instruction> prog = assembly.build();

5.10.3 Machine

The machine library provides a wrapper around the OpenCL calls required to start a SCAD machine.

81    scad::machine machine(platform, devices[0], aocx_filename);

Its main feature is the transparent handling of separate command queues for each kernel which is the documented method to run them in parallel. This is achieved by implicitly creating a command queue for each kernel started.

95    auto data_buff = machine.buffer_for(CL_MEM_READ_WRITE, data);
96    auto lsu = machine.get_component("lsu");
97    lsu->write_buffer(data_buff, data);
98    lsu->start(data_buff, (cl_uint) data.size());

Most OpenCL functions required to use kernels of the SCAD machine implementation with parameters have been wrapped in a more compact interface. This includes waiting for a kernel and buffer transfer to finish:

103    lsu->read_buffer(data_buff, data); // Read back data.
104    lsu->wait(); // Wait for all operations on lsu to finish.
Listing 5.4: "run" Test Tool. (1/2)

```cpp
#include "util.hpp"
#include "machine.hpp"
#include "assembly.hpp"
#include "common/instructions.h"

using namespace scad;

int main (int argc, char *argv[]) {
    setbuf(stdout, NULL); // Have OpenCL kernels not buffer debug messages.
    std::vector<std::string> args(argv+1, argv+argc);
    std::string description_filename = "", aocx_filename = "",
        assembly_filename = "";
    switch(args.size()) {
    case 3:
        description_filename = args[0];
        aocx_filename = args[1];
        assembly_filename = args[2];
        break;
    default:
        std::cerr << "usage: run <processor_description>
        <processor_aocx> <assembly_program>
        " << std::endl;
        exit(1);
    }

    // Used by assembler to map unit names to addresses.
    processor_description proc(description_filename);

    // Read assembly program into string.
    std::ifstream assembly_stream(assembly_filename);
    std::string assembly_src((std::istreambuf_iterator<char>(assembly_stream)),
        std::istreambuf_iterator<char>());

    // Parse and link assembly source into vector of scad instructions.
    scad::assembly assembly(proc);
    assembly.parse(assembly_src);
    std::vector<struct scad_instruction>
        prog = assembly.build();

    // Align program for transfer to buffer.
    std::vector<struct scad_instruction>,
        AlignedAllocator<struct scad_instruction> prog_aligned;

    // Copy unaligned to aligned memory.
    std::copy(prog.begin(), prog.end(),
        std::back_inserter(prog_aligned));
```
Listing 5.5: "run" Test Tool. (2/2)

```cpp
// Only run on FPGA platform.
#ifndef AOC_VERSION == 17
  cl::Platform platform =
  cl_find_platform("Intel(R) FPGA SDK for OpenCL(TM)");
#else
  cl::Platform platform =
  cl_find_platform("Altera SDK for OpenCL");
#endif

std::vector<cl::Device> devices;
platform.getDevices(CL_DEVICE_TYPE_ALL, &devices);
std::cout << "Using first device: " << devices[0].getInfo<CL_DEVICE_NAME>() << std::endl;

// Custom host API root object.
scad::machine machine(platform, devices[0], aocx_filename);

// INPUT/OUTPUT MEMORY
std::vector<scad_data, AlignedAllocator<scad_data>> data(256, (scad_data){.integer = 0});

// Initialize with 0, 1, 2, 3, 4, 5, ...
// for(size_t i = 0; i < data.size(); i++)
// data[i] = (scad_data){.integer = static_cast<cl_uint>(i)};
std::cout << "input: " ; print_scad_vector(data);
std::cout << std::endl;

// Write input data to buffer and pass to LSU kernel.
auto data_buff = machine.buffer_for(CL_MEM_READ_WRITE, data);
auto lsu = machine.get_component("lsu");
lsu->write_buffer(data_buff, data);
lsu->start(data_buff, (cl_uint) data.size());

// Execute control unit with program.
auto control = machine.get_component("cu");
auto prog_buff = machine.buffer_for(prog_aligned);
control->write_buffer(prog_buff, prog_aligned);
control->start(prog_buff, (cl_uint) prog_aligned.size());
lsu->read_buffer(data_buff, data); // Read back data.
lsu->wait(); // Wait for all operations on lsu to finish.
std::cout << "output: " ; print_scad_vector(data);
std::cout << std::endl;
control->wait(); // Wait for control to finish
```
5.11 Host Applications

There are some basic tools that have been developed to configure, test and run various examples.

**configure**  As outlined in Figure 5.2 on page 27, configure applies a platform description to the set of template files and emits the OpenCL source code for the specified processor.

```
> host/configure
usage: configure <platform_description> [<implementations location>]
```

**run**  As the name implies, this tool is used to load and run SCAD machines and move instruction programs. It is given as an example for the SCAD machine library host API in Section 5.10 and as full source code in Listings 5.4 and 5.5.

```
> host/run
usage: run <processor_description> <processor_aocx> <assembly program>
```

**assembler**  This tool is used for debugging the assembly library. It translates source code to the binary instructions used by the control unit which it then displays as text. The example usage in Listing 5.8 also shows that the immediate move instruction contains no information on which data type it conveys.

```
> host/assembler
usage: assembler <platform_description> <assembly file>

This tool is meant to test the assembly library.
In production, assembly is meant to be done by the 'run' tool.

> more examples/branch_hang.asm
loop:
  $1 -> rob@in0
  loop -> cu@in1
  rob@out -> cu@in0

> host/assembler device/basic_minimal.xml examples/branch_hang.asm
moveImmediate $1 or float $4.94066e-324 or $(1, 0) -> 100
moveImmediate $0 or float $0 or $(0, 0) -> 001
move 100 -> 000
```

5.12 Assembler

The assembler is implemented as a library that transforms the assembly source code input string into a C++ vector of `struct scad_instruction`. Assembly is done in what is called 1.5 passes.

First, the input source code is traversed in order. A single regular expression is used to iterate over the instructions which are translated into a vector of C/C++ structs.
immediately. During this iteration, all labels found are stored into a symbol table that maps from `string` to `long int`. If an instruction is encountered that refers to a label, i.e., a jump or branch, it will be resolved through the symbol table immediately if it is defined before the instruction. All label references that are not resolved in this step are stored in a list of symbol uses which the second pass will link.

The assembler requires the processor description introduced in section 5.2 to look which functional units are available, their names, addresses and the unit type which defines the buffers available.
6 SCAD Machine Configurations

This chapter introduces some SCAD machine configurations used both as examples and tests for the configurability of the SCAD machine architecture. To test the limits of the implementation and hardware synthesis, these range from a minimal case with only control unit and reorder buffer to configurations large enough to allow for complex applications and realistic interconnect requirements.

Programs written in the SCAD machine assembly language have a minimal number of functional units of different types that are required to run them. For example, the Heron iteration of Section 3.3 requires a control unit \( cu \), PUs \( pu_0 \) to \( pu_2 \) and a LSU \( lsu \) for the result. Not all of the programs given in this chapter are sufficient for every program shown in this thesis.

There are no application-specific functional units, e.g., Fast Fourier Transformation (FFT) or increment/decrement for loop handling developed in this thesis. All computations will be performed by the general-purpose processing unit as documented for the simulator \([5]\). There are, however, different variants of the LSU implemented to test memory performance of the Intel FPGA SDK for OpenCL and Intel HARP systems. Apart from the monolithic LSU, there are separate LSUs for input and output, a LSU accessing the on-chip block RAM as a scratchpad, and streaming memory interfaces. All configurations share the control unit (\( cu \)) and reorder buffer (\( rob \)), as these are considered to be an absolute minimal requirement for meaningful program execution.

This is a SCAD machine only usable for testing the basic functionality of control unit, reorder buffer and fundamental message handling in buffers and the DTN.

Apart from simulation or profiling data, the only means of gathering output data from this are "program terminates" and "program hangs". Two programs that transfer a branch condition through the reorder unit were written, one is expected to terminate and the other to hang.

Listing 6.1: Branch "terminate" Test. Listing 6.2: Branch "hang" Test.
"Basic" is the smallest configuration that includes a LSU and PU. It is intended to run programs with loops over data to extend test coverage of control unit, buffers, LSU, PU and the interconnect. The "squares" example was written to test this configuration. To run programs like "Fibonacci" or "Heron", three or more PUs are required. For these, the "Basic 3" configuration provides a working environment.

This is an extension of the "Basic" configuration. A scratchpad LSU was added as a platform for testing compilation techniques and performance with on-chip memory.

Testing the concept of separate input and output LSUs, these configurations also feature the scratchpad LSU to be able to write and read temporary data that exceed the buffer capacity.
7 Hardware Synthesis

This chapter presents an overview of the synthesis process and the difficulties involved in the development for this thesis. After an introductory explanation, some of the differences between the emulation environment and the full hardware synthesis will be described. Afterwards, some general problems with the hardware synthesis will be discussed.

Three different systems were used in the development for this thesis. Writing the software was done on a local laptop. While testing in emulation is possible even on mobile devices, hardware synthesis needed to be run on a dedicated server because of hardware requirements. Finally, the compiled software and bitstreams were run on a specialised cluster with job scheduling located at the Paderborn Center for Parallel Computing (P\textsuperscript{2}C) at the University of Paderborn.

7.1 Problems, Contrast to Emulation

Developing large parts of the SCAD machine with only the emulation environment to test in has resulted in some components (e.g., Section 5.8) that are not suitable for hardware synthesis. This is due to several steps and constraints of the hardware generation not being applied by the compiler used for emulation:

**Single call site for channels:** Emulated kernels may contain multiple writes to one channel although hardware synthesis forbids these. Transforming kernels to work in hardware synthesis may require restructuring.
Dynamic indexing into channel arrays: The emulator accepts dynamic indexing into channel arrays, the hardware synthesis does not. Two workarounds for this issue are mentioned in Section 4.3.3, both of which may be wrapped in a function that leaves the original code structure intact.

Open-ended channels: Unlike emulation, hardware synthesis will not proceed when a channel is not read and written to at some location in the system. Together with the automatic removal of dead code performed by the optimisation step prior to hardware synthesis, this poses a problem for the generic DTN implementation. Since the current implementation globally defines the channels for each endpoint, functional units with only inputs will leave the output channel open-ended and thus prevent a successful synthesis.

The first two of the points mentioned here are part of Section 4.3 and documented in the manuals of the Intel FPGA SDK for OpenCL [26]. An approach to solve the open-ended channel problem is the use of unused buffers with static initial content that can not be optimised away.

Arrays of registers An attempt to implement an on-chip interconnect by performing index operations on one large array failed. To define an interconnect with a configurable size, all permutations and switching were done with index calculation on an array that represents all buffering registers of the network. The problem with this approach was that the synthesis places arrays of a certain size into block ram despite an explicit __attribute__((register)) annotation. Since the interconnect is designed as a distributed structure with high throughput and local routing, storing all temporary data into a central block RAM makes the whole structure obsolete.

7.2 Hardware Requirements

For this thesis, all hardware synthesis was performed on a virtual machine in our local data centre (Figure 7.1 middle column). While the initial configuration was 8 processor cores and 16 GiB of RAM, there were multiple instances where hardware could not be synthesised without additional RAM. After multiple iterations, 96 GiB of RAM was found to be sufficient for all designs that fit onto the FPGA.

7.3 Experiments

Since the hardware synthesis of a full SCAD machine developed in emulation proved difficult, smaller experiments were written that test aspects of the synthesis process as well as the functionality of the development environment in general.

These experiments are designed to rely only on basic features and progressively extend on those to test whether assumptions such as the transparent handling of various C data types are correct.

Memcopy Since memory transfer between host and device is a fundamental feature of OpenCL, tests were written that copy all data from an input buffer into an output buffer. Data types tested this way were char, long and scad_data, the basic union data type used in the SCAD machine.

No discrepancies between the expected behaviour and the result of the synthesis were found.
7.3 Experiments

**Channels**  Channel behaviour was tested by extending the memory copy to two kernels that asynchronously handle input and output. These tests confirmed that blocking writes and reads on channels perform as expected, even in the case where multiple executions of one kernel are performed while another keeps running.

**Matrix Additions**  Matrix additions were implemented as a tool to measure the relation between code complexity and synthesis time as well as RAM requirements and FPGA utilisation.

All of these experiments confirmed the assumptions made in the implementation of the SCAD machine. Also, the enormous time and memory complexity of the SCAD machine synthesis, when compared to that of these experiments, was found to be within extrapolated expectations and not the result of implementation errors.
8 Results

This section reports results from the synthesis performed in Chapter 7. To convey some of the unexpected aspects of the results, these are listed after the following analysis.

8.1 Successful Synthesis

This Section will present the results of all successful attempts made to synthesise the SCAD machine configurations listed in Chapter 6. There are many configurations missing in this chapter because they either could not be synthesised or there was not enough time left to do so. Many of these configurations failed synthesis because they could not be fitted into the available FPGA space. The first successfully synthesised and tested configurations are listed in Table 8.1. These values are not accurate comparisons because implementation details were changed between synthesis runs.

The table header acronyms shorten the following terms:

- **LE**: Logic Element
- **FF**: Flip Flop
- **RAM**: Block RAM Elements
- **DSPs**: Digital Signal Processing blocks

One irregularity is that \texttt{basic} is larger than \texttt{basic 3} in terms of Block RAM elements. This is due to buffer calculation effects fixed in later synthesis runs and explained in Section 8.2.

After the limits of the 64 bit implementation were reached, the low number of possible functional units motivated the port to a 32 bit architecture. Consistent use of the data types described in Section 5.5 meant that only these types, the assembly library and parts of the PU had to be changed. Cutting the width of most signals and variables in half made the synthesis of configurations with more PUs possible. The two which were successfully synthesised to hardware are shown in Table 8.2. "scratchpad 6" is considered the maximal configuration for the 32 bit architecture since adding an additional PU or more scratchpad memory prevents successful synthesis. Finally, Table 8.3 shows the relative sizes of the different kernels containing the components of the 32 bit "scratchpad 6" configuration.

All of the configurations listed in this thesis have been tested for basic correctness using the sample programs shown.
### 64 Bit Configurations

<table>
<thead>
<tr>
<th>Name</th>
<th>cu</th>
<th>rob</th>
<th>lsu</th>
<th>scratchpad</th>
<th>pu</th>
<th>LEs</th>
<th>FFs</th>
<th>RAMs</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>minimal</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>32174</td>
<td>527545</td>
<td>1482</td>
<td>183</td>
</tr>
<tr>
<td>basic</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
<td>614039</td>
<td>1030266</td>
<td>3071</td>
<td>191</td>
</tr>
<tr>
<td>basic_3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>753020</td>
<td>1261137</td>
<td>1816</td>
<td>207</td>
</tr>
<tr>
<td>scratchpad</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1×64</td>
<td>1</td>
<td>574349</td>
<td>988282</td>
<td>1233</td>
<td>191</td>
</tr>
</tbody>
</table>

Table 8.1: 64 Bit SCAD Machine FPGA Requirements

### 32 Bit Configurations

<table>
<thead>
<tr>
<th>Name</th>
<th>cu</th>
<th>rob</th>
<th>lsu</th>
<th>scratchpad</th>
<th>pu</th>
<th>LEs</th>
<th>FFs</th>
<th>RAMs</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>scratchpad 3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1×128</td>
<td>4</td>
<td>654903</td>
<td>1147609</td>
<td>1415</td>
<td>227</td>
</tr>
<tr>
<td>scratchpad 6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1×512</td>
<td>6</td>
<td>793046</td>
<td>1397846</td>
<td>1735</td>
<td>249</td>
</tr>
</tbody>
</table>

Table 8.2: 32 Bit SCAD Machine FPGA Requirements

### Hardware Generation Results

<table>
<thead>
<tr>
<th>Component</th>
<th>LEs</th>
<th>FFs</th>
<th>RAMs</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>cu</td>
<td>18066</td>
<td>24996</td>
<td>118</td>
<td>0</td>
</tr>
<tr>
<td>cu input</td>
<td>28025</td>
<td>50305</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>interconnect</td>
<td>7760</td>
<td>13044</td>
<td>44</td>
<td>0</td>
</tr>
<tr>
<td>lsu</td>
<td>5686</td>
<td>11441</td>
<td>41</td>
<td>0</td>
</tr>
<tr>
<td>lsu input</td>
<td>43684</td>
<td>77396</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>lsu output</td>
<td>14242</td>
<td>31561</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>pu 0</td>
<td>66196</td>
<td>119960</td>
<td>131</td>
<td>11</td>
</tr>
<tr>
<td>pu 1</td>
<td>66196</td>
<td>119960</td>
<td>131</td>
<td>11</td>
</tr>
<tr>
<td>pu 2</td>
<td>66196</td>
<td>119959</td>
<td>131</td>
<td>11</td>
</tr>
<tr>
<td>pu 4</td>
<td>66196</td>
<td>119958</td>
<td>131</td>
<td>11</td>
</tr>
<tr>
<td>pu 3</td>
<td>66196</td>
<td>119959</td>
<td>131</td>
<td>11</td>
</tr>
<tr>
<td>pu 5</td>
<td>66196</td>
<td>119960</td>
<td>131</td>
<td>11</td>
</tr>
<tr>
<td>rob</td>
<td>26342</td>
<td>51253</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>scratch</td>
<td>2157</td>
<td>4590</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>scratch input</td>
<td>41053</td>
<td>73028</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>scratch output</td>
<td>14242</td>
<td>31561</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 8.3: FPGA Requirements for Individual Components of 32 Bit Configuration "Scratchpad 6"
8.2 Unexpected Behaviours

This section lists unexpected behaviour encountered during synthesis.

**Channel scheduling:** In all working synthesis results, the code generation decided to extend the depth of the channels in `channel_from_interconnect[]` to 4 instead of using the defined depth of 1.

Channel array with 5 elements. Each channel is implemented 96 bits wide by 4 deep. Requested depth was 1.

Channel depth was changed for the following reason:

- instruction scheduling requirements

The report also lists depths for other channels that differ from those explicitly specified in the source code. This is an unexpected behaviour and it is unclear if the transformations applied change the semantics of any components in a significant way.

**Caching** There are no hints on the implicit creation of caches for memory access which would speed up LSU or control unit operations. Without further testing, the results can not confirm the presence or absence of caches.

**Ring Buffer Modulus:** Table [8.1](#) shows a significant block RAM usage for the basic configuration despite it being smaller than the following configurations. The working theory at the point of writing is that this was caused by a modulus operation for the ring buffer increment produced at least the parts of a division unit required for remainder calculation. A for this theory is that the area report for synthesis reported 71 RAM elements to be used by the following line of code:

```c
buffer->end = (buffer->end + 1) % INPUT_BUFFER_DEPTH;
```

Replacing this expression with a more verbose but explicit check against the maximal value removed all block RAM requirements for this part of the code.

```c
buffer->end = (buffer->end + 1);
if (buffer->end == INPUT_BUFFERDEPTH) {
    buffer->end = 0;
}
```

**Use of Digital Signal Processing Blocks:** When the processing unit was synthesised for the 64 bit SCAD machine, only the *signed long* multiplication was reported to utilise DSP blocks. For 32 bit, the *unsigned* multiplication and division as well as the *signed* division reported DSP block utilisation.

A major part of the LUT requirements of the synthesised SCAD machine appear to be the division and modulo operations of the PUs. This is contrary to the initial expectation that such operations would be translated to use the DSP blocks that are present in most FPGAs and which are more efficient than LUT implementations.
9 Summary and Conclusion

In this thesis, a configurable framework for SCAD machine configuration and generation as well as supporting infrastructure was developed. From one file describing the parts, layout and parameters it produces the OpenCL sources and compiles them to an FPGA bitstream containing the SCAD machine. The templates used for all components are written in OpenCL and built for future modification and experimentation. With the high-level interface for buffers and connections between components, writing new functional units to test ideas is a matter of minutes to hours. Disregarding the synthesis time of 20 hours or more, this allows developers to create SCAD machines for their application with minimal effort.

To support future deployment of the hardware synthesised using this framework, a set of high-level libraries was written. Using the configuration description, an assembly class produces a vector of binary instructions with the correct address mapping understood by the SCAD machine’s control unit. Wrapping parts of the OpenCL C++ API, the machine library loads a previously synthesised FPGA bitstream and exposes an interface to start the included kernels with a minimal amount of code.

This system was tested using both new and existing programs and found to produce the correct results in hardware as well as emulation. Although there are no performance benchmarks, this proves the basic viability of general-purpose programmable accelerators implemented in OpenCL. With the current implementation, the largest SCAD machine configurations still fitting on the FPGA are:

- **64 bit**: 1 control unit, 1 reorder unit, 1 LSU for main memory and 3 PUs.
- **32 bit**: 1 control unit, 1 reorder unit, 1 LSU for main memory, 1 LSU for a 512-word scratchpad and 6 PUs.

There were severe issues with initial delays and technical problems I could take no influence on, including several months of wait for legal processes. Due to these, this presentation of a SCAD machine implementation is severely lacking in several aspects.

One of these is the measurement of performance. Creating benchmarks and comparing those across several SCAD machine configurations and other hardware architectures is a goal that could not be reached. Another lacking feature is the in-depth analysis and optimisation of the generated hardware. Some attempts were made to simplify code that had an unexpected high logic utilisation, but three was no time for a systematic or thorough approach. The most prominent example for these is the PU, which was expected to use more of the DSP Blocks. The main feature of the Intel HARP platform, cache coherent access to shared memory, has also not been used.
To conclude, this implementation of SCAD machines in OpenCL does not meet expectations. The number of functional units fitting on the FPGA is below what prior experiments in VHDL suggested. Hardware generation appears to be less than optimal for components with very data-dependent behaviour such as the Input and output buffers. A reason for this may be that the strength of the Intel FPGA SDK for OpenCL appears to be the construction of deeply pipelined execution unit, which is not suited for components with a complex internal state such as the input buffers. Time and resource requirements of the hardware synthesis process rule out fast development cycles with a tightly coupled feedback loop as well as run-time reconfiguration of the SCAD machine.

Finally, while it may be useful for comparative analysis of features or techniques, the current state as presented in this thesis is not yet suited for practical application.
10 Future Work

10.1 Asynchronous MIB

The MIB is one of the few remaining synchronous and blocking components of the SCAD machine architecture. With the current implementation requiring each recipient to acknowledge their move instructions, the potentially blocking wait for one round-trip may cause serious delays. Also, in contrast to the DTN, the MIB is not optimised for non-blocking behaviour despite being a structure potentially spanning many functional units.

A major obstacle for a non-blocking MIB is the fact that receiving buffers should know about all incoming messages to be able to reorder them. Another problem is that even a non-blocking MIB will need to deliver move instructions in the order they were issued in. This is especially problematic when considering the architecture outlined in Figure 10.1, because the re-transmission of moves from the recipient to the sender is timing-dependent. While this problem may be solved by introducing additional tags, building an efficient asynchronous MIB poses an interesting challenge.
10.2 Alternate Instruction Set

The following is a short list of SCAD machine instructions I consider equally powerful to the ones used currently. A benefit of these is the more efficient use of space in a fixed-width instruction encoding.

```
imm 42  Load an immediate value into the “imm” output buffer of the control unit.
```

```
loop:
  // ...
branch loop
```

```
branch consumes a value from the input cu.branch. If it is 0, the instruction does nothing. If it is 1, PC is set to the address given.
```

```
cu.imm -> lsu.addr
```

```
lsu.out -> cu.branch
```

Move instructions between buffers remain as they are.

10.3 Shared Memory LSU

Shared memory is an OpenCL feature that is especially suited for the Intel HARP hardware. To use it from within the SCAD machine, an LSU is required that uses a shared virtual memory buffer for communication with the host. Shared Virtual Memory (SVM) is pinned to the same memory addresses on both host and device, and allows the passing of structures that include pointers. Also, some configurations of SVM support concurrent coherent access by host and device.

Section A.1.6 (“Atomic Functions”) of [26] states that the Intel FPGA SDK for OpenCL features the standard 32 bit atomic functions, but has no support for the 64 bit atomic functions.

10.4 Interconnects

Because of time constraints and the limited number of functional units fitted onto the FPGA, no non-trivial DTN has been developed. For larger implementations, and to test their resource utilisation in hardware generation, there are several efficient interconnects to choose from. Examples for these are the Omega [34] network or the Selector Tree Network [28].

10.5 Partial reconfiguration

Partial reconfiguration is a technology supported by the Stratix V FPGA series produced by Intel. It is used for the deployment of OpenCL to FPGAs and allows a system designer to make areas of the FPGA separately reconfigurable.

If this could be applied to the functional units of a SCAD machine, it may make a fully run-time configurable accelerator possible. The main downside of this approach is that the cost of the infrastructure and work required to support it may outweigh the benefit, especially with the limited number of different functional units and limited combinations of those used so far.

10.6 Avalon Interface

The Intel OpenCL SDK for OpenCL relies on, and supports using components written in a VHDL or Verilog from OpenCL. Future analysis of the implementation produced
may indicate congestion hot-spots where a replacement written in Verilog or VHDL may improve performance. This is especially interesting for DTNs already implemented in VHDL or more efficient input buffers.

10.7 If-Then-Else Functional Unit

Instead of explicit branching, a new functional unit may choose between the different results of two basic blocks by evaluating a condition.

I propose a functional unit that implements the ternary operator

\[ <\text{condition}> \, ? \, <\text{result if true}> : <\text{result if false}>. \]

In essence, this is a multiplexer that allows the compiler to replace branches with calculation of both parts and a subsequent choice of the correct result.
Bibliography


# List of Figures

3.1 Program View of a SCAD Machine .................................................. 7
3.2 SCAD machine overview given in [9] .................................................. 8
3.3 High Level Functionality of Reordering Input Buffers .............................. 15

4.1 Concurrent Kernel Execution ............................................................... 20
4.2 Functional Overview of an Example Configuration of Intel HARP System .... 22

5.1 System Architecture ........................................................................ 25
5.2 Configuration Process of this Framework .............................................. 27
5.3 Different Buffer Configurations .......................................................... 29
5.4 Blocking/Non-blocking Components .................................................. 30
5.5 Structure of Fundamental Data Types .................................................. 31
5.6 Interface of the Data Transport Network .............................................. 31
5.7 Input Buffer Structure - Ring Buffer and Data Slots ............................. 32
5.8 Output Buffer - Two Ring Buffers With Common Start ......................... 33
5.9 Termination of a LSU via sync .......................................................... 36
5.10 Control Unit Overview .................................................................. 36
5.11 Monolithic LSU ............................................................................. 39
5.12 Separate LSUs for Input, Output and Scratchpad .................................. 39

7.1 Workflow of Synthesis Process ............................................................ 47

10.1 Conceptual draft of asynchronous MIB ................................................. 57
Listings

3.1 Fibonacci Example ........................................... 12
3.2 Heron Example taken from [7] .............................. 13
3.3 Squares Example ............................................ 14
4.1 Printing the platform name in C .......................... 18
4.2 Printing the platform name in C++ ..................... 18
4.3 Workaround for dynamic indexing into channel arrays, Excerpt from [26] 20
4.4 Workaround for dynamic indexing into channel arrays as used in this thesis .......................... 20
5.1 Source of "multimem" configuration ....................... 28
5.2 Buffer Integrated into Functional Unit .................. 34
5.3 Separate Buffer in Extra Kernel ......................... 35
5.6 Configure Tool Usage ...................................... 43
5.7 Run Tool Usage ............................................ 43
5.8 Assembler Tool Usage ...................................... 43
6.1 Branch "terminate" Test ...................................... 45
6.2 Branch "hang" Test ........................................ 45