Application-specific Configuration of Exposed Datapath Architectures

MASTER THESIS

Author: Sireesha Rudraiah Basavaraju
Submission Date: June 8, 2017

A thesis submitted in fulfillment of the requirements for the degree of Master of Science in
Electrical and Computer Engineering

First Reviewer: Prof. Dr. Wolfgang Kunz
Chair of Electronic Design Automation

Second Reviewer: Prof. Dr. Klaus Schneider
Chair of Embedded Systems Group

Supervisor: Prof. Dr. Klaus Schneider
Declaration

Hereby, I declare that the present thesis work titled *Application-specific Configuration of Exposed Datapath Architectures* was drawn up after MPO "Elektrotechnik und Informationstechnik" by myself, without the help of third parties but the support of my supervisor, and that all the used sources and tools including the internet are completely and exactly mentioned, and that everything is marked which is taken unchanged, shortened or analogous from other literature.

__________________________  Kaiserslautern,___________
(Sireesha Rudraiah Basavaraju)
Acknowledgement

Firstly, I would like to express my heartfelt gratitude to my supervisor and advisor Prof. Dr. Klaus Schneider for giving me an opportunity to carry out my thesis work in his research group, and for continuously supporting and motivating me during my thesis and my studies. His priceless advice and guidance has always helped me to arrive at the right decisions during my entire journey of Masters.

I express my sincere thanks to Prof. Dr. Wolfgang Kunz for mentoring me throughout my studies and for giving me the permission to carry out this thesis.

I would also like to thank Mr. Anoop Bhagyanath and the other members of Embedded Systems Group, Department of Computer Science for their support during my thesis work.

Finally, I would like to express my very profound gratitude to my parents, grandparents and friends for providing me with unfailing support and continuous encouragement throughout my years of studies. This accomplishment would not have been possible without them.

Thank you very much.

Sireesha R Basavaraju
Application-specific Configuration of exposed datapath architectures

Abstract

Application-specific computing is widely used to meet the cost, performance and power requirements of embedded systems. Application-specific instruction-set processors (ASIPs) have emerged as a promising solution in this domain. In the last few decades, a number of techniques have been developed for designing ASIPs using the classic Reduced Instruction-Set Computer (RISC) architecture. Recent studies have show that these RISC-based processors are restricted in exploiting instruction-level parallelism, and this has led to the design of new architectures that offer a high level of scalability and parallelism. One example of such an architecture are the exposed datapath architectures which consist of an array of concurrently executing functional units, and they expose the details of the datapath to the compiler allowing it a fine-grained control over the datapath. This eliminates redundant data movements and results in high performance and energy efficient processors. Moreover, they are very good candidates for application-specific computing. The synchronous control asynchronous dataflow (SCAD) processor belongs to the family of exposed datapath architectures. SCAD consists of an array of buffered functional units interconnected via special networks and it is programmed using special instructions called move instructions.

In this work, we present a novel idea called chaining of functional units which can be implemented in the SCAD to make it application-specific. The most frequently occurring expressions in the application program will be implemented with chaining in the SCAD architecture which results in a fine-grained application-specific configuration of the SCAD architecture. Two different methods of chaining are proposed and evaluated qualitatively to find the most beneficial implementation. In addition, this thesis also presents a simple approach used in identifying the potential candidates to be implemented with chaining in SCAD. Finally, we conduct some experiments by extending the existing SAT solver based SCAD code generator that helps us to understand that when chaining is used in SCAD, the number of processing units required for computation increases. This increase is quantifiable, and hence we can still claim that chaining is a beneficial idea.

5.4 Results of Expression Generation . . . . . . . . . . . . . . . . 60

6 Experiments ........................................... 63

7 Conclusion ............................................. 68

Bibliography ............................................ 73
List of Figures

1.1 Comparision of Flexibility vs. Efficiency for Different Architectural Options [7] .......... 2

3.1 Frontend of the MiniC compiler ......................... 14
3.2 Architecture of SCAD processor[28] ................... 17
3.3 Firing in a FU when tgt address is available in output buffer . 22
3.4 Firing in a FU when tgt address is not available in output buffer 22
3.5 Control unit in SCAD processor ....................... 25
3.6 Load-Store unit in SCAD processor .................... 25
3.7 Reorder buffer unit in SCAD processor ................. 27
3.8 Universal processing unit in SCAD processor .......... 27

4.1 A simple expression in an application program and its chaining implementation .................. 35
4.2 Different scenarios in an output buffer of a PU .......... 38
4.3 Different scenarios in an input buffer of a PU ........... 39
4.4 DFG corresponding to Listing 4.1 ...................... 40
4.5 DUP unit in SCAD ....................................... 41
4.6 Using DUP to produce multiple copies of a result .......... 41
4.7 The expression tree in Heron’s iteration ............... 44
4.8 Input buffer example with chaining .................... 45

5.1 Program call graph for the expression generation .......... 55
5.2 Expression Generation ................................. 59
5.3 Expression trees in different sorting algorithms .......... 61
5.4 Expression trees occurring in DCT, FIR and matrix multiplication programs .................... 62

6.1 Variation in the number of processing units used in a SCAD machine with half, full and no chaining in basic blocks .... 65
6.2 Variation in the number of processing units in SCAD machine for different number of chained nodes . . . . . . . . . . . . . . . 66
6.3 Variation in the SAT solver execution time with and without chaining in the basic block . . . . . . . . . . . . . . . . . . . . 67
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Different buffered ports at the input of the control unit</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Move instructions used with the control unit</td>
<td>24</td>
</tr>
<tr>
<td>3.3</td>
<td>Different buffered ports at the input of the load-store unit</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>Move instructions used with the load-store unit</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>Move instructions used with the Reorder buffer</td>
<td>26</td>
</tr>
<tr>
<td>3.6</td>
<td>Different buffered ports at the input of a universal processing unit</td>
<td>27</td>
</tr>
<tr>
<td>3.7</td>
<td>Move instructions used with the universal processing units</td>
<td>28</td>
</tr>
<tr>
<td>3.8</td>
<td>Different variables used in the SAT encoding for SCAD code generation</td>
<td>31</td>
</tr>
<tr>
<td>4.1</td>
<td>Syntax of Type 1 Chaining Instructions</td>
<td>49</td>
</tr>
<tr>
<td>4.2</td>
<td>Implementation of Chaining Instructions in F#</td>
<td>50</td>
</tr>
<tr>
<td>4.3</td>
<td>Syntax of Type 1 Unchaining Instructions</td>
<td>50</td>
</tr>
<tr>
<td>4.4</td>
<td>Implementation of Unchain Instructions in F#</td>
<td>50</td>
</tr>
</tbody>
</table>
List of Abbreviations

ADL  architecture description language
ASIC application specific integrated circuit
ASIP application specific instruction processor
CPU  central processing unit
DTN  data transport network
GPP  general purpose processor
HDL  hardware-description language
ILP  instruction level parallelism
ISA  instruction set architecture
MIB  move instruction bus
PC   program counter
PU   processing unit
RISC reduced instruction-set computer
SCAD synchronous control asynchronous dataflow
1 Introduction

1.1 Motivation

Embedded processors are commonly designed to implement target application functionality under strict design requirements such as power, performance, area and efficiency. Different processor architectures that are used to implement the target functionalities can be categorized into two types based on their degree of flexibility: general purpose computing and application-specific computing [9, 21].

In general purpose computing, processors are designed based on the Von Neumann computation model [9]. Here, the functionality is implemented using pre-designed off-the-shelf general purpose processors (GPPs). These processors implement a generic instruction set which is applicable to a wide variety of applications. Though they provide complete flexibility in terms of programming, they are usually too slow in speed and consume too much power to be a feasible solution for many embedded applications.

In application-specific computing, the earliest designs widely known as application specific integrated circuits (ASICs), implemented the entire target functionality purely in hardware. ASICs provide high performance for a given application because of the highly optimized hardware. ASICs can fulfill the specific constraints and reduce the energy consumption for a given application by using an appropriate architecture designed for that application [21]. However, designing ASICs is a time-consuming and expensive task [10]. They also lack flexibility in terms of programmability, i.e., they cannot be used for applications other than the one they were initially designed for.
application specific instruction processors (ASIPs) have emerged as a solid solution in application-specific computing, since here the specialization does not result in loss of flexibility and this flexibility does not compromise on the performance and power requirements. They are often also referred to as customizable processors [24]. ASIPs allow longer and broader use of design in the market since the complexity is shifted from silicon into software. The attribute ‘application-specific’ no longer refers to just a software application. Instead, it refers to the application of a processor in a specific system context, performing specific functions with distinct design goals [19]. Figure 1.1 shows traditional computing approaches used in embedded systems compared in terms of flexibility vs. efficiency. We can observe that ASIPs provide higher flexibility than ASICs in conjunction with better efficiency in terms of performance per area and performance per power-consumption when compared to GPPs.

![Figure 1.1: Comparision of Flexibility vs. Efficiency for Different Architectural Options [7]](image)

Over the last few decades, there has been extensive research involved in the design of customizable processors [19, 12, 17]. All these methods are focused on implementing ASIPs on top of the classic reduced instruction-set
computer (RISC) architecture. A key technique used to improve the performance of RISC processors is the instruction level parallelism (ILP) [1]. For instance, the different methods that are widely used to increase ILP are: instruction pipelining, superscalar execution, out-of-order execution, speculative execution and register renaming. By exploiting the ILP in the program, the performance of the processors can be increased.

While there has been extensive work carried out in implementing high performance RISC processors, recent studies in [6] have shown that these processors do not scale well because of the following reasons: (1) power- and pipeline-limits impede clock rate growth, (2) wire delays cause overheads for concurrency and (3) complexity of large monolithic architectures.

Recent developments in processor architecture try to augment the classic program counter driven von Neumann model of computation with compiler technology designed for limited dataflow-like execution. This has led to new class of processor architectures called exposed datapath architectures. Prominent examples for these architectures include TRIPS [6], RAW [3], Wave-Scalar [5], TTA [11] and SCAD [28] architectures. The goal of these architectures is to use more ILP to provide more single-threaded performance. The main idea of these architectures is to expose all the functional units as well as the datapath information to the compiler so that the compiler is able to schedule the instructions, and also to move the data directly between the functional units without writing it to a common space like registers.

The key features of these architectures include:

- They provide a rich interface between the compiler and the microarchitecture.
- To increase the concurrency and operational level parallelism, these architectures include an array of concurrently executing functional units.
- ISA of these architectures conveys the data dependence graph. Hence there is no need for complex hardware structures, such as reservation stations [2], to detect these dependencies.
- The hardware implementation conveys direct data transfer between instructions, i.e., in the hardware, the output of producer functional...
unit is directly delivered to the consumer functional unit, rather than transferring it through registers.

A qualitative analysis of these architectures indicates the following advantages over superscalar processors [11]:

- A clear separation of functional units and data transport results in design simplicity.
- These architectures do not enforce any limit on type, number and implementation of functional units, and on data transport capacity, making them very flexible.
- Since the compiler can schedule data transports, it can exploit high degree of scheduling freedom. This allows the compiler designer to use different types of compile-time optimizations.
- Available functional units and data transport network are used efficiently, resulting in high performance potential.

In addition to design flexibility, design simplicity and efficient resource utilization, direct data transfer between the instructions replaces register writes and this results in energy-efficient processors. Also, since the data-dependency information is analysed during the compilation time and encoded into the compiled instructions, there is no need for complex hardware structures to detect these dependencies thus reducing the area and power requirements. Due to the simplicity, efficiency and scalability features, exposed datapath architectures can be used to implement highly efficient application-specific processors.

This thesis presents a new concept that allows the use of exposed datapath architectures, in particular the Synchronous Control Asynchronous Dataflow (SCAD) architecture developed at the Embedded systems group in the University of Kaiserslautern, in application-specific computing. SCAD is a modular and easily customizable processor architecture, thus fits well for designing efficient parallel processors to implement target applications.

The SCAD processor has an array of simple functional units (also referred to as processing units) connected to each other by special inter-connection networks. Since the ISA of these architectures conveys the data-dependence graph, in each cycle of operation, the result of one functional unit will be
sent to another functional unit by using special instructions called *move* instructions. But if these functional units are connected to each other based on the expressions found in the application program, then the results will be transported directly between the functional units without the need for explicit move instructions. The concept of *chaining* is used to implement this idea in SCAD. When the functional units are chained automatically depending on the different expressions found in the program, the overall performance of the application can be improved. Hence with chaining of functional units, the SCAD architecture can be configured to be application-specific. In this thesis, we study this concept of chaining and its effects on the SCAD architecture.

### 1.2 Contribution

This thesis presents novel ideas which enables the use of exposed datapath architectures, in particular, the SCAD architecture, in the field of application-specific computing. The main contributions are listed below:

- The first part of this thesis presents the concept of *chaining* in SCAD. Two different methods are proposed and compared qualitatively. The architectural changes that are required to implement chaining are discussed and compared to find the most suitable implementation. We present the set of new instructions that are required for implementing chaining in SCAD.

- The second part of this thesis concentrates on identifying the most suitable candidate expressions in the target-application program to be implemented with chaining in SCAD. The idea of this technique is to visualize the SCAD program as a set of expressions and identifying all possible expression trees in the program. Then, by calculating their number of occurrences in the program, one can obtain the most suitable candidates for the implementation with chaining of functional units in SCAD. For this purpose, we develop an automatic application-specific expression identification technique that starts with an application de-
scribed in a high-level programming language and produces a set of potential candidates that can be implemented using chaining in SCAD.

- In the last part of this thesis, we evaluate the effects of implementing chaining in SCAD. Specifically, we want to measure the increase in the number of processing units required when the large expressions found in a basic block is implemented with chaining instructions. An experimental study based on the existing SAT-based SCAD code generator shows this drawback that comes with the use of chaining in SCAD architecture.

1.3 Organization

The structure of this thesis is divided into the following chapters: Chapter 2 lists some of the prominent customizable processor frameworks used in industry and also the instruction-set extension techniques used in the ASIP design process. Chapter 3 gives an introduction to the MiniC compiler and presents the SCAD architecture in detail. The concept of chaining is explained the Chapter 4. Chapter 5 presents a simple technique that can be used for identifying application-specific expressions. In Chapter 6, we discuss the effects of using chaining in SCAD by experimenting with the existing SAT-based code generator. Finally, Chapter 7 concludes with a discussion on possible future work.
2 Related Work

Section 2.1 presents some of the commercially available customizable platforms and ASIP design methodologies used in today’s embedded systems. Followed by this, Section 2.2 briefly describes some of the techniques that are widely used in ASIP design.

2.1 Customizable Processors for Embedded Systems

Customizable processors or ASIPs have been widely used in multiprocessor system-on-chip designs. The design range available today spans from pure ASIP design with extensible functional units to processors equipped with embedded reconfigurable arrays. The granularity of customization that is offered in today’s ASIPs ranges from fine-grained customization to coarse-grained customization. In fine-grained customization, the processor’s base instruction set is extended with application-specific instructions to obtain ASIPs. In coarse-grained customization, the implementation of the architecture is also augmented with custom architectural components such as specialized register files, memory interfaces, etc. Today, the commercially available ASIP support customization at various levels of granularity, and the user can choose the different features to efficiently implement their applications. Several ASIP frameworks have been developed over the years in the embedded processor industry. They are designed to meet short time-to-design and time-to-market requirements of ASIPs de-
sign, by lowering the design and verification efforts. These frameworks can be categorised as follows [8]:

- **Base processor based framework**: These frameworks allow designers to develop an ASIP from a pre-designed and pre-verified configurable base processor. Extensions can be added in the form of functional units, custom instructions and parametrised hardware blocks. Examples include Xtensa by Tensilica Inc. [12], AR Ctangent from ARC International [14], NIOS I and II from Altera Corp. [13]. Xtensa and NIOS II processors are further described in this section.

- **Specification based frameworks**: These frameworks are based on architecture description language (ADL) which are languages used to design ASIPs from the scratch. The ISA is specified using ADL and the framework automatically generates both the hardware model of the ASIP in HDL and the software tools like the compiler, simulator, linker, and assembler. A widely used example is the LISATek framework from CoWare [15]. This framework is further described in this section.

Tensilica’s Xtensa processor [12] is a widely used commercial ASIP platform. Xtensa is an extensible and configurable processor core which can be used by system designers in embedded application by sizing and selecting features and adding new instructions. The Xtensa processor core is based on a RISC architecture and it can be configured in two dimensions: First, ”fine-grained” in form of instruction extensions. Second, ”coarse-grained” in form of structural extensions. Xtensa ISA serves as the foundation for the processor extensibility. This ISA is extended with application-specific instructions that are described using a special high-level language called Tensilica Instruction Extension or TIE. The Xtensa processor generator (XPG) uses this description to automatically add new instructions to the RTL description of the hardware resulting in seamless integration of the new hardware into the processor pipeline, and also extends the complete toolset including the compiler, assembler, debugger and simulator for the extended processor. All new instructions are executed in a single-cycle like the base instructions (at least in the programmer’s view). Hence the maximum frequency of the extended processor may depend on the TIE compiler-generated hardware extensions.
The NIOS II processor from Altera Corp. [13] is another commercially available customizable processor. NIOS II is a general-purpose RISC processor core implemented as soft IP core i.e, the core is not fixed in silicon and can be implemented by any FPGA family. One way to obtain a customized processor using NIOS II is by the addition of custom instructions. The custom instruction logic helps to realize the application-specific operations in hardware. The custom instruction logic is integrated into the NIOS II processors arithmetic logic unit. These custom instructions are similar to native RISC instructions. From the software perspective, custom instructions appear as assembly macros, so the programmers can use custom instructions during software development. Altera offers Qsys tools to fully automate the process of configuring processor features and generating the hardware design that is implemented on FPGA.

The LISATek tool suite developed at Aachen University is the most popular example for an ADL-based ASIP development framework. It uses the Language for Instruction Set Architectures (LISA) [20] ADL for processor modeling. A LISA model captures the machine’s ISA as well as the processor resources, such as register, memories and instruction pipeline. The tool flow is a stepwise refinement procedure: It starts with a LISA model of the target architecture which drives the generation of both software tools as well as the hardware synthesis models. The software tools allow the mapping of application to ASIP architecture and simulation on a virtual prototype. A feedback loop exists that adapts the ADL model with custom instructions. Another feedback loop annotates the results from hardware synthesis. In this way, LISATek’s iterative architecture exploration methodology can be used to produce ASIPs optimized for performance.

Recently, ASIPs have been coupled with programmable logic to develop what is known as reconfigurable ASIPs [18]. The main idea here was to incorporate hardware flexibility in ASIPs by augmenting them with reconfigurable logic. The reconfigurable fabric can be implemented as a stand-alone processing unit or, as a co-processor or, it could even be added as a functional unit within the processor’s datapath. Some of the existing examples are described next.
Razdan and Smith propose PRogrammable Instruction Set Computer or in short PRISC [16] wherein the conventional set of RISC instructions were augmented with application-specific instructions which were purely implemented in hardware using programmable functional units (PFUs). These PFUs can be dynamically reconfigured for each application. In their first implementation called PRISC-I, PFUs were added in parallel with the existing functional units, augmenting the existing CPU datapath. These PFUs were added to the microarchitecture so that the benefits of high-performance RISC techniques were maintained (e.g., fixed instruction formats) and at the same time impact on the processor’s cycle time was also kept minimal. This method typically uses a standard FPGA platform which were not designed specifically to optimize the needs of an integrated FPGA-processor system. FPGA-like approaches can be very efficient if it is not necessary to reconfigure the FPGA too often.

Similar to the above method, Saponara et al. propose an ASIP-based reconfigurable architecture called as the reconfigurable-ASIP in [17]. The idea here again is to combine the ASIP paradigm with reconfigurable hardware with a goal to provide both flexibility and performance. In one of the implementations of r-ASIP, they integrated the reconfigurable logic as a functional unit into the datapath of a deeply pipelined ASIP architecture. This improved the architecture flexibility and efficiency by providing post-fabrication customization. The r-ASIP architectures have a circuit complexity in the order of 100s of equivalent ASIC gates.

In general, designing such reconfigurable ASIPs is a challenging task, because the design space is huge. The flexibility and performance of the reconfigurable ASIPs relies strongly on the organization of the architecture. The initialization overhead of reconfigurable fabrics also limit their use with ASIPs.
Chapter 2. Related Work

2.2 Instruction Set Extension in ASIP Design

The second part of this thesis presents a technique to identify the potential candidate expressions in the application program which will be implemented by chaining in the SCAD processor. Identifying recurring clusters of operations (arithmetic and logical) is similar to a popular technique used in ASIP design, known as the Instruction-Set Extension (ISE). ISE extends the base instruction set with application-specific or domain-specific instructions. Automatic identification of most profitable ISEs for a given application is very challenging task and a number of algorithms have been proposed in this context during the past few decades.

The ISE identification process is divided into two phases: custom-instruction generation and selection. The instruction generation process is a design space exploration problem and it consists of clustering of basic operations (such as add, sub, etc.) into larger complex operations to be implemented as ISEs. The instruction selection process selects a subset of most profitable ones obtained from the instruction generation phase. The ISE identification process is subject to different constraints such as area, cycle count, code size, etc. This is a well-specified topic of research and the problems involved are known to be computationally complex. Since the program is visualized as a control-/ data-flow graph (CDFG) by the compiler, the results and concepts from graph theory are extensively used to identify ISEs. The problem of identifying clusters of operation can be mapped to problem of identifying sub-graphs from a given graph in this case.

In [21], Galuzzi and Bertels present a detailed overview of all aspects involved in the customization of an instruction-set and compare different methodologies in ISE problem. Different methods of ISE generation have been proposed by many researchers based on different criteria such as: connected instructions (having only inter-dependent nodes in the sub-graphs) or disconnected instructions (having independent sequence of nodes that could be implemented by parallel operation in hardware); single-output instructions.
or multiple-output instructions. The number of outputs which can be implemented in a custom instruction is limited by the architectural organization. Usually since register files implement only one write-port, many methodologies consider only single-output custom instructions. In case of SCAD, since there is no use of register files, we could easily implement multiple-output expressions provided the functional units have multiple output ports.

In [22], the authors propose an all-pair common slack identification tool for the identification of clusters of operation which can be implemented as parallel templates. These parallel templates are disconnected instructions which consist of parallel independent operations and they correspond to nodes in the CDFG which can be scheduled for simultaneous execution in the hardware, i.e, it has clusters of operation which are free of data dependencies.

In [23], the authors identify ISE generation as a task of a hardware-software partitioning approach applied at instruction-level granularity. They propose their ISEGEN algorithm which is an iterative improvement methodology following the basic principles of Kernigham-Lin min-cut heuristic, which is a well known graph-partitioning heuristic. This method generates solutions close to those obtained manually by expert designers.

All the methods that are available in ISE generation could be used to obtain potential expressions for implementation with chaining. However, these methods are exhaustive and are known to be computationally complex and hence the technique implemented in this thesis considers a simple approach based on a pre-order traversal algorithm. The application is simulated to obtain a profiling information which is used to select the beneficial candidate expressions. The technique proposed in this thesis considers only connected instructions, although SCAD allows the implementation of disconnected instructions since it has a large number of functional units operating concurrently. Since the current implementation of SCAD considers functional units with only a single output, the expression generation also produces only single output expressions.
3 Preliminaries

The first part of this chapter presents various tools that are used by the algorithms in the expression generation algorithm presented in Chapter 5. The benchmark programs for experimental purposes are written in MiniC and the algorithms used in the expression generation phase will use Cmd programs which is the intermediate code generated by the compiler for MiniC programs as the input. The second part starts with a detailed description of the SCAD architecture followed by the description of the universal SCAD machine which has been developed for experimental purposes. Finally, we present some insights on SAT-based SCAD code generation for basic blocks. This method will be further useful in understanding the experiments conducted for this thesis.

3.1 MiniC: Language and Compiler

MiniC is a small programming language developed at the Embedded Systems group in the University of Kaiserslautern [36]. It mimics a subset of the C language. MiniC has a minimal set of data types, namely boolean (bool), unsigned or signed integers (nat and int), arrays, and tuples. The MiniC compiler is used to compile these applications to target machine dependent code. The MiniC compiler structure is organized into the frontend and the backend. Figure 3.1 shows the frontend of the MiniC compiler. The frontend translates the MiniC program into an intermediate representation (IR) format.
IR is the compiler’s general presentation format of the program which is easier to analyze, optimize, and transform from one form to another [26]. It is usually target and source language independent. Most commonly used forms of the IR are data-flow graphs and three-address code. In the MiniC compiler, three-address code representation is used. One can say, three-address code representation is the textual form of a data-flow graph representation.

In three-address code representation, all high level control-flow constructs and complex expressions are decomposed into simple sequences of three-operand assignments and gotos. There is no standard format for three-address code, but all the statements will have the following form:

\[ y := x \ op \ z \]

Here \( op \) refers to an arithmetic or logical operator like addition, subtraction, less-than, etc., and \( x \) and \( z \) can be constants or variables, and \( y \) refers to a variable. The goto statement will have the following format: \( \text{goto } l \), which indicates a jump to line number given by \( l \) in the IR code. The IR generator inserts \textit{temporary variables} to store the intermediate results of computations [24]. Static Single-Assignment (SSA) form is a variation of three-address code with the further restriction that each temporary variable can be assigned only once in SSA[26]. Both forms are exploited in compiler optimization, and they may be used in examining different flow analyses of the program.
Chapter 3. Preliminaries

Listing 3.1: A simple MiniC program

```c
// MiniC program
function func1(nat b, nat c, nat d, nat e) : nat {
    nat y;
    y = b * c + c * d;
    return y;
}

thread Func1 {
    nat z;
    z = func1(2,3,4,5);
}
```

Listing 3.2: Cmd code generated by MiniC compiler

```cmd
0 :  _t2 := 2
1 :  _t1 :=  _t2 * 3
2 :  _t4 := 3
3 :  _t3 :=  _t4 * 4
4 :  y.FC1 :=  _t1 +  _t3
5 :  _t0 :=  y.FC1
6 :  goto 7
7 :  z :=  _t0
8 :  sync
```

The IR generated by the MiniC compiler is called Cmd code. The Cmd code looks like a virtual machine code with infinitely many temporary registers. Listing 3.1 shows sample MiniC source code of a simple function and Listing 3.2 shows the corresponding Cmd code.

In the Listing 3.1, the function `func1` contains an expression: \( y = b \times c + d \times e \), which is broken into a sequence of three-operand assignments in the Cmd code as seen on Lines 0 to 4 in Listing 3.2.

The backend of the compiler generates machine dependent instructions from the Cmd code. The backend consists of a code generator to produce assem-
bly code for the Abacus processor. Abacus processor is a simple RISC-based processor designed at the Embedded Systems Group [36]. The Abacus instruction set simulator [27] which simulates the assembler program of the given application step-by-step, is also used in the expression generation algorithm to obtain the dynamic occurrences of the expressions in the program.

3.2 SCAD Architecture

The synchronous control asynchronous dataflow (SCAD) architecture [28] is an exposed datapath architecture developed at the Embedded Systems group in the University of Kaiserslautern. The SCAD processor consists of an array of buffered functional units (also referred to as processing units), each with limited control, connected to each other by a dedicated communication network for passing the data. Conceptually, each functional unit may have an arbitrary number of input and output ports, with each port consisting of queues or first-in first-out (FIFO) buffers. Each functional unit can implement any arbitrary function. The only restriction for these implemented functions is that the number of input values consumed and the number of output values produced must be known during the compilation time. The compiler has the knowledge about both the datapath and the number and types of functional units. Therefore, it has a fine-grained control over the datapath compared to a traditional RISC processor. A program compiled for SCAD architecture directly encodes the data-flow information between the instructions.

3.2.1 SCAD Structure

Figure 3.2 shows the operational model of SCAD processor, wherein each functional unit contains two input ports and one output port. A FIFO buffer is connected to each port of the functional unit and the depth of this buffer (i.e., the number of entries the buffer holds) is at least equal to one. Each buffer has a unique address assigned to it. Each entry in the buffer is of the form \((adr, val)\). For an input buffer, the \(adr\) field corresponds to the address
Chapter 3. Preliminaries

Figure 3.2: Architecture of SCAD processor[28]

of the output buffer which computes and sends the value. This value will be stored in val field. For an output buffer, the adr field holds the address of the input buffer to which it will send the computed value.

A special value ”⊥” is used to indicate the non-availability of the actual value in each entry of these buffers. If the input buffer contains an entry (adr,⊥), it means the required value is not yet available and it will be sent later on by the output buffer with address adr. If the output buffer of a functional unit contains an entry (adr,⊥), it means that the value to be sent to the input buffer adr is yet to be computed by this functional unit. If the output buffer
of a functional unit contains the entry \((\perp, \text{val})\), it means that the value is already computed by the functional unit but the target address is yet to be received, which will be sent later on by the control unit.

The input and output buffers of the functional units are connected to each other by a special interconnection network, called the data transport network (DTN). The DTN is used by the buffers to *asynchronously* send the data values to each other when they are ready. The DTN can be implemented by a simple set of buses and sockets, or by complex parallel interconnection networks like Banyan networks [31] or Benes networks [32].

Apart from the functional units that compute arithmetic and logical operations, SCAD also requires at least one load-store unit (LSU) to perform data memory access operations. The LSU has two input buffers and one output buffer. In case of a load operation, only one input buffer is used for firing and it holds the address of the memory location. The second input buffer holds a dummy value for a load operation. In case of a store operation both the input buffers are used for firing; the first one holds the address of the memory location and the second one holds the value to be stored to the memory location. The store operation does not use the output buffer. The output buffer holds the value loaded from the memory (for a load operation).

Similar to the buffers of other functional units, all the three buffers of LSU also hold entries of the form \((\text{adr}, \text{val})\).

The SCAD machine is programmed by a sequence of special instructions called the *move instructions*. The effect of a move instruction is to move a value to an input buffer of a functional unit. Move instructions have the following form:

- \((\text{src}, \text{tgt})\): defines the move of a value from the head of output buffer with address \(\text{src}\) to the tail of input buffer with address \(\text{tgt}\).
- \((c, \text{tgt})\): defines the move of a constant value \(c\) to the tail of input buffer with address \(\text{tgt}\).

The control unit (CU) is a special functional unit in the SCAD machine. It mainly fetches the move instructions from the instruction memory using the address given by a special register, the program counter (PC). These instructions are sent *synchronously* from the control unit to the functional
units via a special interconnect network called as the move instruction bus (MIB). Like the DTN, the MIB can also be implemented as a simple set of buses and sockets, or by complex parallel interconnection networks like Banyan or Benes networks.

The details of the move instructions and the different functional units in the SCAD machine developed for experimental purposes are explained in Section 3.2.4.

3.2.2 Execution of move Programs on SCAD

SCAD program execution is described as follows: In each cycle of execution, the control unit fetches the next move instruction, \((src, tgt)\) or \((c, tgt)\), from the instruction memory using the address given by the PC and broadcasts it on the MIB to all the functional units. In each cycle, all the functional units continuously snoop both the MIB and DTN. When a move instruction of the form \((c, tgt)\) arrives on the MIB, the input buffer with the address \(tgt\) will add a new entry \((cu, c)\) to its tail. Here, the source functional unit for the value \(c\) is the control unit \(cu\). When a move instruction of the form \((src, tgt)\) arrives on the MIB, the input buffer with address \(tgt\) will add a new entry \((src, \perp)\) to its tail. Since it is possible that the result value has already been computed before a corresponding move instruction has been issued by the control unit, the output buffer with address \(src\) will first check if it contains entries of the form \((\perp, val)\) and then replaces the one that is closest to the head of the queue by \((tgt, val)\). If no such entry exists, then it adds a new entry \((tgt, \perp)\) at the tail of the buffer.

Each functional unit uses a special signal \(fullBuffer\) to indicate "buffer is full" condition to the control unit. If either the \(src\) buffer or the \(tgt\) buffer is full, then the control unit is stalled by signaling on \(fullBuffer\) signal. The control unit then aborts the writes to the buffer(s) and resends the same move instruction in the next cycle.

All the functional units operate concurrently. Each functional unit fires when sufficient data is available at their input ports and sufficient space is available at their own output buffer. This means it consumes the input values and
produces a specified number of copies of output values in its output buffer. The results are written at the corresponding output buffers. The output buffers are responsible for the final transport of the values to the input buffers of the functional units. Since the space is already allocated in the target buffers, the data transfer can be done \emph{asynchronously} by the output buffers via the DTN. For every completed entry \((tgt, val)\) present at the head of the output buffer \(src\), a message of the form \((src, tgt, val)\) is created, wherein \(tgt\) corresponds to the target input buffer to which the value \(val\) must be sent. This message is transported via the DTN to the input buffer \(tgt\). This input buffer replaces the entry \((src, \perp)\) that is closest to the head of the queue with \((src, val)\).

Conditional branch instructions in SCAD programs are also of the from \((src, tgt)\) and are handled by the control unit itself. The control unit has one buffer at its input port and this is associated with the PC of the processor. If the control unit finds the address of its input buffer in the \(tgt\) field of the current move instruction, then the control unit realizes that this move instruction corresponds to a branch instruction and stalls immediately. The input buffer holds the branch-target address as well as the next program counter value i.e., PC+1. The control unit then waits till the branch condition value arrives at its input buffer. If the branch condition is ”true”, it starts fetching from the branch-target address, else it starts fetching from the next PC address.

Jump instructions (unconditional branch) are of the form \((c, tgt)\) and are also handled by the control unit. Unlike conditional branch instructions which result in a stall, the jump instruction does not cause any stalls in the program because the instruction itself encodes the branch target address. There is no need to wait for the calculation of any condition, since a branch is always taken in case of a jump instruction. The control unit recognizes the jump instruction by observing the address of its input buffer in the \(tgt\) field of the move instruction \((c, tgt)\). It then replaces the current PC with the value in the instruction. In the next cycle, the control unit fetches the instruction using the PC value.
As observed, the move instructions are registered into the buffers in the order they were sent by the control unit. Hence the instruction’s transport happens synchronously and this ensures that the program-flow order is maintained. In contrast, the data transport related to move instruction is deferred to a later point of time when the data becomes available. Hence, the data transport happens asynchronously and in the data-flow order.

3.2.3 Firing in SCAD Functional Units

It is necessary to understand the firing process of the functional units in SCAD. Later on, this firing process is modified to incorporate the chaining implementation in SCAD. The firing process is explained with functional units containing only two inputs and one output. However, the process can be generalized to any number of inputs and outputs.

Each functional unit can fire when all the input operands required for the execution are available. The firing decision in each functional unit works as follows: In every cycle, it scans the head of the input buffer for valid values, i.e., if entries \((adr_1; x_1)\) and \((adr_2; x_2)\) with \(x_i\) not equal to \(\bot\) are available, and if there is free space in its output buffer to store \(nc\) copies of the result, then the functional unit can fire and it will consume entries \((adr_1; x_1)\) and \((adr_2; x_2)\) to produce the new result value \(y := f_1(x_1; x_m)\) and \(nc\) copies of \(y\) are stored in the output buffer. Here, we can distinguish between two cases:

1. If the output buffer has one or more entries of the form \((tgt, \bot)\), then the output value \(y\) is stored in that entry of the output buffer which is closest to the head of the output, i.e., that entry is replaced with \((tgt, y)\). An example for this scenario is in Figure 3.3.

2. If there should be no such entry in the output buffer, then a new entry \((\bot, y)\) is placed at the tail of output buffer, and the next target address for this output buffer will be stored in this entry. This means, the result value is computed before a corresponding move instruction has been issued by the control unit. An example for this scenario is in Figure 3.4.
Chapter 3. Preliminaries

3.2.4 Universal SCAD Machine

The SCAD processor developed for experimental purpose is called the universal SCAD machine. In general, a universal SCAD machine can consist of an arbitrary number of universal processing units whose input and output ports are buffered. An universal processing unit can perform any arithmetic or logical operation. Hence, in addition to moving the two operands to these units, it is necessary to move the opcode for each operation. Therefore, apart
from the normal input ports which hold the operand values, for each pro-
cessing unit an additional input port is required to hold the opcode and the
number of copies to be produced for that operation. Each processing unit
has a unique address \( \text{adrU} \), and each one of its buffers extends this address to
a unique buffer address like \( \text{adrU}@\text{in0} \), \( \text{adrU}@\text{in1} \), \( \text{adrU}@\text{out} \), or \( \text{adrU}@\text{opc} \).
The different processing units used in the SCAD processor are assigned ad-
dresses as shown:

- \( \text{Address 0} \) is used for the control unit \( \text{cu} \).
- \( \text{Address 1} \) is used for the load/store unit \( \text{lsu} \).
- \( \text{Address 2} \) is used for the reordering unit \( \text{rob} \).
- \( \text{Address 3..N + 2} \) are used for the \( N \) universal processing units \( \text{pu}(0) \)
  upto \( \text{pu}(N - 1) \).

The behaviour of these processing units is described in more detail in the
following subsections.

**Control Unit**

The control unit \( \text{cu} \) maintains a local program counter \( \text{pc} \) that is used to
address the program memory. Whenever \( \text{cu} \) fires, it reads an instruction
from \( \text{ProgMem}[\text{pc}] \) and sends it on the MIB. The control unit also handles the
conditional branch and unconditional branch (or jump) instructions. Jump
instructions are encoded as move instructions to the \( \text{pc} \), represented by
\( \text{Stl} \rightarrow \text{pc} \). As long as a \( \text{pc} \) is locally available in the \( \text{cu} \), the corresponding
move instruction is fetched and issued, and the local \( \text{pc} \) is incremented. If \( \text{cu} \)
observer a jump instruction, it sets \( \text{pc} \) to \( l \).

<table>
<thead>
<tr>
<th>\text{cu}@\text{in0}</th>
<th>branch condition (0 or 1) port</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{cu}@\text{in1}</td>
<td>&quot;Then&quot; branch target address port</td>
</tr>
<tr>
<td>\text{cu}@\text{in2}</td>
<td>&quot;Else&quot; branch target address port</td>
</tr>
</tbody>
</table>

Table 3.1: Different buffered ports at the input of the control unit

The \( \text{cu} \) contains three input ports as shown in Figure 3.5, which are used to
handle the conditional branch instructions. They are explained in Table 3.1.
Conditional branch instructions are implemented by the program as follows:
Chapter 3. Preliminaries

1. First we have to move the branch target address \( pc_{Then} \) to \( cu@in1 \) by using: \( \$pc_{Then} -> cu@in1 \).

2. Next, issue move instructions to compute the branch condition.

3. Finally, move the result of the branch condition evaluation found in \( adrU@out \) to \( cu@in0 \) by using: \( adrU@out -> cu@in0 \). If this instruction is issued by the \( cu \), then it will invalidate its local \( pc \), and will automatically move \( pc+1 \) to \( cu@in2 \).

4. The \( cu \) can fire if it has a local \( pc \) or if the three input ports contain values. In the latter case, a new local \( pc \) is defined, otherwise, instructions are fetched using the existing local \( pc \). Note that the \( cu \) stops issuing further instructions as soon as the \( pc \) is outside the allowed addresses of the program memory. The move instructions concerning the \( cu \) are summarized in Table 3.2.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l -&gt; pc )</td>
<td>set the ( pc ) in the control unit to the address ( l )</td>
</tr>
<tr>
<td>( $pc_{Then} -&gt; cu@in1 )</td>
<td>move the address ( $pc_{Then} ) to ( cu@in1 )</td>
</tr>
<tr>
<td>( adrU@out -&gt; cu@in0 )</td>
<td>move the result of branch condition calculated by functional unit with address ( adrU ) to ( cu@in0 )</td>
</tr>
</tbody>
</table>

Table 3.2: Move instructions used with the control unit
Chapter 3. Preliminaries

Figure 3.5: Control unit in SCAD processor

Figure 3.6: Load-Store unit in SCAD processor

**Load/Store Unit**

The load/store unit (LSU) has the structure as shown in Figure 3.6. The different ports at the input of the LSU are given in Table 3.3. The output side has only one port that can be accessed using `lsu@out`. It stores the values produced by load operations. The move instructions concerning the `lsu` and their function are summarized in Table 3.4.

| **lsu@in0** | holds the memory address to which a load or store is made |
| **lsu@in1** | holds the values to be stored in case of store |
| **lsu@opc** | holds a pair `(ls, nc)` where `ls` is the opcode and `nc` is number of copies (used in case of load); the opcode `ls` is `true` for load and `false` for store instructions |

Table 3.3: Different buffered ports at the input of the load-store unit
### Chapter 3. Preliminaries

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$adr -&gt; lsu@in0</td>
<td>move the address value $adr to lsu@in0</td>
</tr>
<tr>
<td>$val -&gt; lsu@in1</td>
<td>move the data value $val to lsu@in1</td>
</tr>
<tr>
<td>st -&gt; lsu@opc</td>
<td>move the opcode for Store operation to lsu@opc</td>
</tr>
<tr>
<td>$(ld, nc) -&gt; lsu@opc</td>
<td>move the opcode and the number of copies $nc for Load operation into lsu@out</td>
</tr>
<tr>
<td>lsu@out -&gt; adrU@bf</td>
<td>move the loaded value to some input buffer $adrU@bf</td>
</tr>
</tbody>
</table>

Table 3.4: Move instructions used with the load-store unit

### Reorder Unit

If result values are not found in the desired order in the output buffers, one can make use of the reorder unit $rob$. The reorder unit $rob$ has one one input buffered port $rob@in0$ and one output buffered port $rob@out$. It does not contain any special port for opcode value, since it performs only one operation. It fires whenever there is a value in the input lane, and simply copies that value to the output lane. The move instructions concerning the $rob$ and their function are summarized in Table 3.5.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$val -&gt; rob@in0</td>
<td>move a direct operand $val to the input buffer of $rob rob@in0</td>
</tr>
<tr>
<td>$adrU@out -&gt; puI@in0</td>
<td>move a value from the output buffer $adrU@out to the input buffer of $rob rob@in0</td>
</tr>
<tr>
<td>rob@out -&gt; adrU@inI</td>
<td>move a result value from the output buffer of $rob rob@out to some other input buffer $adrU@inI</td>
</tr>
</tbody>
</table>

Table 3.5: Move instructions used with the Reorder buffer

### Universal Processing Units

The SCAD machine has $N$ universal processing units: $pu0, \ldots, puN$. Each processing unit $puI$ has three input ports and one output port as shown
in Figure 3.8. The ports are described in Table 3.6. Appropriate move instructions concerning processing units are are summarized in Table 3.7.

<table>
<thead>
<tr>
<th>Port</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>puI@in0</td>
<td>left operand of a binary operation</td>
</tr>
<tr>
<td>puI@in1</td>
<td>right operand of a binary operation</td>
</tr>
<tr>
<td>puI@opc</td>
<td>((opc, nc)) where (opc) is the opcode of a binary operation and (nc) is number of copies to be written to puI@out</td>
</tr>
<tr>
<td>puI@out</td>
<td>result values</td>
</tr>
</tbody>
</table>

Table 3.6: Different buffered ports an the input of a universal processing unit

### 3.2.5 Example SCAD move Program

Listing 3.4 shows a move program implemented for Heron’s iteration given in Listing 3.3. The SCAD machine here consists of four processing units \(pu0\) to \(pu3\). Since for each operation, we have to move two operands and an opcode
Table 3.7: Move instructions used with the universal processing units

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$val -&gt; puI@in0</td>
<td>move a direct operand $val to the left operand buffer of the input puI@in0</td>
</tr>
<tr>
<td>$val -&gt; puI@in1</td>
<td>move a direct operand $val to the right operand buffer of the input puI@in1</td>
</tr>
<tr>
<td>$adrU@out -&gt; puI@in0</td>
<td>move a value from the buffer $adrU@out to the left operand buffer of the input puI@in0</td>
</tr>
<tr>
<td>$adrU@out -&gt; puI@in1</td>
<td>move a value from the buffer $adrU@out to the right operand buffer of the input puI@in1</td>
</tr>
<tr>
<td>(opc,nc) -&gt; puI@opc</td>
<td>move the opcode $opc and the number of copies $nc to the puI@opc</td>
</tr>
<tr>
<td>puI@out -&gt; $adrU@inI</td>
<td>moves a result value from the output buffer puI@out to some other input buffer $adrU@inI</td>
</tr>
</tbody>
</table>

value, each RISC-like instruction will be translated to three move operations in the move program.

Listing 3.3: Heron’s iteration as MiniC program

```c
function heron(nat a) : nat {
    nat xold, xnew;
    xnew = a;
    do {
        xold = xnew;
        xnew = (xold + a/xold)/2;
    } while (xnew < xold);
    return xnew;
}
	hread Heron {
    nat z;
    z = heron(121); // compute the square root of 121
}
Listing 3.4: SCAD program for Heron’s iteration

//初始化
0: $121 → pu0@in0
1: $0 → pu0@in1
2: (addN,3) → pu0@opc // xold = xnew

//循环开始
3: $121 → pu2@in0 // a for a/xold
4: pu0@out → pu2@in1 // xold for a/xold
5: (divN,1) → pu2@opc // division opcode
6: pu2@out → pu1@in1 // a/xold for xold+a/xold
7: pu0@out → pu1@in0 // xold for xold+a/xold
8: (addN,1) → pu1@opc // addition
9: $2 → pu0@in1 // 2 for (xold+a/xold)/2
10: pu1@out → pu0@in0 // xold+a/xold for // (xold+a/xold)/2
11: (divN,4) → pu0@opc // calculate xnew
12: pu0@out → pu3@in1 // xold for xnew < xold
13: pu0@out → pu3@in0 // xnew for xnew < xold
14: (lesN,1) → pu3@opc // calculate the condition
15: $3 → cu@in1 // jump address
16: pu3@out → cu@in0 // branch instr

//循环结束
17: st → lsu@opc // store the result
18: $0 → lsu@in0 // memory address is 0
19: pu0@out → lsu@in1

3.2.6 Optimal Code Generation for SCAD using SAT Solvers

In [28], the authors suggest that the classic code generation based on depth-first traversal of syntax trees is inadequate for exposed datapath architectures
and propose that the breadth-first traversal based approach which has been formerly used in classic queue machines to exploit ILP, is very appropriate for exposed datapath architectures. The breadth-first traversal based approach ensures that the operands are found in the correct order in the buffers and thus eliminates the need for additional memories like registers. However, as explained in [28], the simple code generation used for queue machines does not produce optimal code for SCAD machine, and hence, the authors propose in [29] to map the code generation problem to a boolean satisfiability (SAT) problem and use a SAT solver to determine the minimal number of processing units required to execute the programs without any computational overhead, i.e., without using any duplication and swap operations. The SAT encoding of optimal SCAD code generation is explained further in detail in this section. In general, this method can be used for any exposed datapath architecture with buffered processing units. In this thesis, the same experimental set up will be extended with new boolean constraints as described in Chapter 6, in order to evaluate the increase in the number of processing units that results from the use of chaining in SCAD architecture.

Methodology

The SCAD code generation problem can be defined as follows: Given a SCAD machine with one load-store unit and \( p \) universal processing units as explained before and a basic block in SSA form, i.e.,

\[
\begin{align*}
  x_{tgt}(0) &= x_{srcL}(0) \odot_0 x_{srcR}(0) \\
  &\vdots \\
  x_{tgt}(l1) &= x_{srcL}(l1) \odot_{l1} x_{srcR}(l1)
\end{align*}
\]

for some variables \( V := \{x_0, \ldots, x_{n1}\} \), where \( \odot_i \) denotes some binary operation, we need to determine if the basic block can be executed on SCAD without any computational overhead. If so, then the output contains the number of processing units required with a schedule for the basic block.
Chapter 3. Preliminaries

<table>
<thead>
<tr>
<th>$V_{tgt}$</th>
<th>set of target variables i.e., the variables that occur as left hand sides in SSA form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_{src}$</td>
<td>set of source variables i.e., the variables that occur on the right hand sides in SSA form</td>
</tr>
<tr>
<td>$V_{ld}$</td>
<td>set of load variables i.e., the variables that occur only on the right hand sides in SSA form (note that $V_{ld} := V_{src} \setminus V_{tgt}$)</td>
</tr>
</tbody>
</table>

Table 3.8: Different variables used in the SAT encoding for SCAD code generation

In SSA form, every variable $x_i$ is assigned a value only once and its value can be used any number of times after the assignment. Table 3.8 defines three different kinds of variables used in the SAT encoding. The following assumptions also hold:

- If a variable is in the set $V_{src} \cap V_{tgt}$, then all its read operations occur after its unique write operation.
- Processing unit 0 is the load-store unit.
- Processing units $\{1, \ldots, p\}$ are the p universal processing units.

Two different relations are defined to obtain the allowed schedules for the basic block on the SCAD machine as follows [29]:

- processing unit assignment $\alpha_{i,j}$ for $x_i \in V$ and $j \in \{0, \ldots, p\}$
  - $\alpha_{i,j}$ means that $x_i \in V$ is produced by processing unit $j$
  - we may fix that all $x_i \in V_{ld}$ are produced by load/store unit
  - this determines the instructions of the basic block executed by processing unit $j$
- variable order $x_i \prec x_j$ for $x_i, x_j \in V$ which means that $x_i$ and $x_j$ occurs in some buffer in that order, which includes:
  1. production order: $x_i$ and $x_j$ are target variables produced by the same PU. Therefore they both occur in the output buffer of that PU.
  2. consumption order: $x_i$ and $x_j$ are source variables (same argument) of instructions assigned to the same processing unit. Therefore, they both occur in the corresponding input buffer of that processing unit.
Next, a set of boolean constraints are formulated as follows:

**Strict order relation constraint** \(\prec\): The constraints given by Equations 3.1 and 3.2 ensure that the variable ordering \(\prec\) will be transitive and irreflexive, respectively. Together they imply that \(\prec\) is acyclic as given by Equation 3.3.

\[
\bigwedge_{x_i, x_j, x_k \in V} x_i \prec x_j \land x_j \prec x_k \rightarrow x_i \prec x_k \tag{3.1}
\]

\[
\bigwedge_{x_i \in V} \neg x_i \prec x_i \tag{3.2}
\]

\[
\bigwedge_{x_i, x_j \in V} x_i \prec x_j \rightarrow \neg x_j \prec x_i \tag{3.3}
\]

**Data dependency**: For every instruction \(x_{tgt}(i) = x_{srcL}(i) \odot x_{srcR}(i)\) in the basic block, this constraint ensures that operands \(x_{srcL}(i)\) and \(x_{srcR}(i)\) will be produced before producing \(x_{tgt}(i)\):

\[
\bigwedge_{i=0}^{t-1} x_{srcL}(i) \prec x_{tgt}(i) \land x_{srcR}(i) \prec x_{tgt}(i) \tag{3.4}
\]

**Unique PU assignment**: The constraint 3.5 demands that every variable is produced by one and only one processing unit, since in SSA form, a variable can be assigned only once.

\[
\left( \bigwedge_{x_i \in V} \bigvee_{k=0}^{p} \alpha_{i,k} \right) \land \left( \bigwedge_{x_i \in V} \bigwedge_{k=0}^{p} \alpha_{i,k} \rightarrow \bigwedge_{j=0 \land j \neq k}^{p} \neg \alpha_{i,j} \right) \tag{3.5}
\]

**Buffer constraints**: The final constraint ensures that a total variable ordering will exist for those variables \(x_i\) that are at some time in the same buffer. If two instructions \(x_{tgt}(i) = x_{srcL}(i) \odot x_{srcR}(i)\) and \(x_{tgt}(j) = x_{srcL}(j) \odot x_{srcR}(j)\) of the basic block are executed on different PUs, then it is possible to move their operands to the corresponding input buffers irrespective of the ordering of operand values in some buffers. However, if the instructions
are executed on the same processing unit $k$ in the order $x_{tgt(i)}$ followed by $x_{tgt(j)}$, we should be able to move the operand $x_{srcL(i)}$ before the operand $x_{srcL(j)}$ to the left input buffer of PU $k$. When the left operands $x_{srcL(i)}$ and $x_{srcL(j)}$ are produced by different processing units, then this is not a problem, but when both operands are produced by the same PU, then the ordering $x_{srcL(i)} \preceq x_{srcL(j)}$ has to be enforced by the constraint. Similar constraints apply for the right operands. Equation 3.6 gives the constraint that ensures that the variables will occur in the right order in the input and output buffers, when two instructions are executed in the same PU:

$$\bigwedge_{i,j=0}^{l-1} \beta_{tgt(i),tgt(j)} \rightarrow \begin{cases} 
x_{tgt(i)} \prec x_{tgt(j)} \\
\land \\
\beta_{srcL(i),srcL(j)} \rightarrow x_{srcL(i)} \preceq x_{srcL(j)} \\
\land \\
\beta_{srcR(i),srcR(j)} \rightarrow x_{srcR(i)} \preceq x_{srcR(j)} \\
\lor \\
x_{tgt(j)} \prec x_{tgt(i)} \\
\land \\
\beta_{srcL(i),srcL(j)} \rightarrow x_{srcL(j)} \preceq x_{srcL(i)} \\
\land \\
\beta_{srcR(i),srcR(j)} \rightarrow x_{srcR(j)} \preceq x_{srcR(i)} 
\end{cases}$$  

(3.6)

Here, the new relation $\beta_{i,j}$ as given by Equation 3.7 will be true when the variables $x_i$ and $x_j$ are produced by the same processing unit and false otherwise.

$$\beta_{i,j} = \bigvee_{k=0}^{p} \alpha_{i,k} \land \alpha_{j,k}$$  

(3.7)

The conjunction of these constraints (Equations 3.1 to 3.6) provides a boolean propositional formula for the given basic block. Every satisfying assignment of that formula will be a valid schedule for executing that basic block on the SCAD machine. Also, this schedule will give the minimum number of processing units that are required to run the basic block on the SCAD machine.
4 Chaining in SCAD

This chapter presents a detailed study of the concept of chaining in SCAD processors. The first section gives the motivation for introducing chaining in SCAD followed by the different types of chaining. Section 4.2 gives a detailed description of two different methods of chaining and their semantics. After discussing the drawbacks of each method, one of the two chaining methods is chosen for implementation in SCAD and an example SCAD program with this chaining is also given. In the last section, we present the disadvantages of chaining.

4.1 Concept of Chaining

A SCAD program is nothing but a set of expressions that are executed to produce a final result. A SCAD machine consists of an array of processing units (PUs) connected to each other by an interconnection network, and the program is executed in data-flow order. An expression in a program is nothing but a combination of different ALU operations. For example, consider the expression given below and its corresponding data flow graph (DFG) in Figure 4.1a:

\[ t0 = (b * c) + (d * e) \]

This expression is broken into three intermediate expressions during compilation which can be evaluated with three different PUs in SCAD processor, i.e., two multiplier units and one adder unit, and this computation requires two move instructions in order to send the results from the two multiplier units to an adder unit to compute the final result. If this expression occurs frequently
in the program, for example, if this expression is a part of a loop which runs for multiple times, then the intermediate results have to be moved explicitly for every iteration. With increasing number of iterations, the number of intermediate move instructions increases, resulting in increased contention on the MIB. If the PUs could be temporarily connected to each other in the form of this recurrent expression, then the number of moves required reduces. This is the idea of chaining in the SCAD machine. **Chaining** can be defined as: Linking output buffers and input buffers of functional units via software to implement a large expression more or less directly in hardware. The above example expression can be implemented by chaining two MUL units and one ADD unit as shown in Figure 4.1b.

![Figure 4.1: A simple expression in an application program and its chaining implementation](image)

Since we know that most of the execution time of a program is spent on loops; using chaining for expressions within the loop can be very beneficial for executing programs on the SCAD machine. Much of our considerations that follow in the next sections will also use chaining within loops of a program.
4.1.1 Types of Chaining in SCAD

There are three different types of chaining required in SCAD processor as follows:

- Chaining two PUs: This type of chaining would be useful for computing big expressions. Here, say if output buffer of PU0 is chained to one of the input buffers in PU1, then output of PU0 is always moved to that input port of the PU1 directly without any need for explicit move instructions.

- Chaining of a constant value to the input port of a PU: This type of chaining would be useful when any of the input operands is a fixed constant in a big expression, or while implementing operations like increment by 1, or multiply by a constant $v$ etc. In this case, the constant value is chained to the input port of the PU fixing that operand for further computations.

- Chaining an opcode to a PU: Since the SCAD machine has universal PUs, which can perform any basic ALU operation, we can fix the opcode of the PU by using this type of chaining. In this case, the PU will perform the specified ALU operation and produce the given number of copies of the result, and we can avoid moving opcode to the PU for every single computation.

4.2 Semantics of Chaining

This section explains the two possible ways of implementing chaining in SCAD. It also presents the effect of the implementation on SCAD.

4.2.1 Type 1: Chaining without Multiple Copies

In the first type of implementation of chaining, the following points are considered:
There is only one type of chaining instruction which can be used to chain any two PUs through their buffers or to chain a constant to an input buffer of PU.

There is one type of unchain instruction, which when used breaks the chain between two PUs, or the PU’s input buffer and the constant.

If the output buffer of a PU which produces multiple copies of a single operation is chained to another input buffer, then all the values are sent to that input buffer, i.e., it cannot be chained to multiple input buffers.

This is the simplest form of implementing chaining in the SCAD machine. Here, we associate a boolean flag with each FIFO buffer of the PU in the SCAD machine, which when set to \textit{true} conveys that the corresponding buffer of the PU is a part of larger chained expression. Apart from this, we need to introduce a source address holder and a value holder for each input buffer, and a target address holder for each output buffer.

When a chaining instruction is sent on the MIB to a buffer, the buffer might still not be empty because of the previously issued move instructions. Therefore, it is important to understand the different scenarios that might exist in the buffers when chaining occurs and how these scenarios should be handled by the implementation.

\textbf{Output buffer:} Consider the three different situations shown in Figure 4.2 occurring in the output buffer. The control unit sends a chaining instruction to this PU on the MIB and the PU sets the chain flag corresponding to the output buffer to \textit{true} and saves the target address in the target address holder. In the scenario of Figure 4.2a, the result is produced by the PU even before the move instruction which corresponds to the target address was issued by the control unit. Now, if the chaining flag is set, then this chaining information provides the target address and this must be used to complete the entry or else the buffer will enter a deadlock situation. If the chaining flag for this buffer is not set, then it means an appropriate move instruction will arrive sooner or later in the execution. In the scenario in Figure 4.2b, even if the chaining flag is set for the output buffer, the result produced by the PU
will be used to complete the entry \((tgt, \perp)\) at the head of the output buffer. Remember that the addresses are sent synchronously by the control unit. Therefore, the address \(tgt\) corresponds to a previously moved instruction and this should use the result produced to maintain the dataflow order of computation. The empty buffer scenario in Figure 4.2c is the expected scenario, so if the chaining flag is set for this buffer, then the result will be added to the head of the output buffer and the target address is obtained from the chaining information.

\[
\begin{array}{|c|c|}
\hline
\text{adr} & \text{val} \\
\hline
\hline
\perp & x \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
\text{adr} & \text{val} \\
\hline
\hline
tgt & \perp \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
\text{adr} & \text{val} \\
\hline
\hline
\hline
\end{array}
\]

(a) Output buffer with entry \((\perp, val)\)  
(b) Output buffer with entry \((adr, \perp)\)  
(c) Output buffer is empty

Figure 4.2: Different scenarios in an output buffer of a PU

- **Input buffer**: Consider the three different situations shown in Figure 4.3 occurring in the input buffer. For a given input port of a PU, the buffer might still have some values at the head when the chaining flag is set as shown in Figure 4.3a, then the firing process for that PU should first use all the values at the head of the input buffers in order to maintain the dataflow computation order. Once they are consumed, from the next firing onwards, the chaining information is used for computing values. Figure 4.3b shows a scenario where the source address is available and the values have not yet arrived. Now if the chaining flag is set for this buffer for a chain to buffer, then the value which arrive from the DTN will be first use to complete the two entries to preserve the dataflow order. When the input buffer is empty as shown in Figure 4.3c, and the chaining flag is set, it uses this information for firing or else it simply waits for the next move instruction for the buffer.
Chapter 4. Chaining in SCAD

Each PU has a firing process which consumes the values and produces the results in the output buffer, and a sending process which removes the results from the output buffer and sends it on the DTN. In order to meet all the conditions in different input and output buffer scenarios that were explained, the following changes are considered for implementing type 1 chaining in SCAD:

**Firing Process**: Note that the PU can fire only when all the input buffers have valid entries of the form $(adr, val)$ at their head. The firing decision in each PU is modified as follows:

1. Check all the buffers at the input ports for completed entries $(adr, val)$.
2. If an input buffer has $(adr, val)$, then the PU can use this to fire irrespective of the value of the chaining flag else go to 3.
3. Check if chaining flag is set for this input buffer. If yes, then go to 3(a) else go to 4.
   
   (a) If chained to constant, obtain the value and complete the entry, else go to 3(b).
   
   (b) If chained to another PU’s output buffer, wait for the value from that PU.
4. If the chain flag is not set, wait for next move instruction.

**Sending process**: To handle the output buffer scenarios, the sending process in each PU is also modified as follows:

- If the head of the output buffer has value $(\bot, val)$, check if the chaining flag is set for this output buffer.
1. If chaining flag = \texttt{true}, then use the target address information from chaining to send the value on DTN.

2. If chaining flag = \texttt{false}, then wait for the appropriate move instruction on the MIB.

   - If the head of the output buffer has value \((adr, val)\), then send \(val\) to \(adr\) irrespective of the chaining condition.

\section*{Drawback of Type 1 Chaining}

It is often the case that a node in DFG feeds more than one node in the next step, i.e., the result of an operation is used by more than one node in the DFG. For example, consider the set of expressions given in Listing 4.1 and its DFG representation as shown in Figure 4.4. In this example, the variable \(t0\) is used by two other expressions. Since there are no registers in the SCAD machine, a PU computing \(t0\) will produce multiple copies (two in this example) of the same value.

\begin{verbatim}
Listing 4.1: A set of expressions
\begin{verbatim}
t0 = (b*c) + (d*e) 
t1 = 4 + t0 
t2 = t0 + 20 
\end{verbatim}
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{dfg_example.png}
\caption{DFG corresponding to Listing 4.1}
\end{figure}

Assume that \(PU0\), \(PU1\) and \(PU2\) correspond to variables \(t0\), \(t1\) and \(t2\) respectively, and this combined expression has several occurrences and hence will be a good candidate for chaining. In this case when \(PU0\) is chained to \(PU1\), then both values of \(t0\) will be forwarded to \(PU1\). Now, if we chain \(PU0\) and \(PU2\), this will overwrite the chain of \(PU0-PU1\), since it will overwrite the target address holder. Therefore, this expression cannot be implemented by using type 1 chaining.
Since chaining is implemented with one bit flag for each buffer, this problem could easily arise during the execution. To avoid this problem, there are three possible solutions:

1. Use a new PU called as duplicate (DUP) as shown in Figure 4.5 which has one input port and two output ports, all containing FIFO buffers. The function of DUP would be to simply copy the value from the head of the input buffer to the tail of the two output buffers. A chaining flag and target address holder is added to each one of the output buffers of the DUP unit and this way the output ports can be chained to two different PUs. Using DUP operators, we could implement the expression in Figure 4.4 as shown in DFG given by Figure 4.6. If we need more number of copies, then we could use a sequence of DUPs. In general, we would require a minimum of $\log_2(N)$ number of DUPs (using a binary tree wherein each node is a DUP), to produce $N$ copies of a result (with $N$ being a power of 2).

![DUP unit in SCAD](image)

![Using DUP to produce multiple copies of a result](image)

The main disadvantage of this method is that we need to add DUPs to the SCAD machine which consumes chip area. Also, the result will be available at different points of time since it has to move through the DUPs, and this can increase the computation time of the overall result. The movement of values from one DUP to next DUP happens on the DTN, resulting in increased traffic on the DTN.
2. We could compute the same value by using multiple concurrent sets of chained PUs and this method would be feasible if the SCAD machine has enough number of PUs and if the expression tree is small, i.e, it has a small number of nodes. Otherwise, this method becomes quickly inefficient when compared to normal SCAD computations.

3. The simplest solution is to avoid chaining of an output buffer connected to multiple nodes and use normal move instructions. This maintains the simplicity of the SCAD machine, and we could still obtain some performance gain by using chaining for the rest of the expression.

4.2.2 Type 2: Chaining with Multiple Copies

Chaining implementation type 2 is a more general implementation of chaining in the SCAD machine. With this type of chaining, an input buffer can be connected to more than one constant value or output buffers and the output buffer can feed more than just one input buffer. Unlike type 1, this kind of implementation requires more than just a flag to convey the chaining information. One possible solution is to use an integer value with the following meaning:

- For the output buffers, this integer value is the number of copies produced and the number of PUs to which the result must be sent to.
- For the input buffers, this integer value represents the number of constants or the output buffers to which it is chained to.

One has to note that the number of nodes to which a buffer can be chained with is limited only by the depth of the buffer and it is compile-time analyzable. The basic rules of firing and producing the output are still the same as in the normal SCAD machine. The chaining information will be enclosed between two new instructions: begin and end in the produced SCAD program. This implies that all the instructions that appear between begin and end are considered to be chaining instructions and there is no new syntax which is required to implement type 2 chaining. The different nodes to which the buffer is connected to will be defined between these begin and end instructions. This implies that in a given basic block of the program, all
the chaining information must be identified and collected between these two instructions. The unchaining of PUs is obtained by using a single unchain instruction. Once an unchain instruction is observed, all the existing chained connections will be removed.

Using type 2, the expression in Figure 4.4 can be implemented without any need for a DUP unit. Say if $t_0$, $t_1$ and $t_2$ are calculated in $PU_0$, $PU_1$ and $PU_2$, respectively, then the output buffer of $PU_0$ is chained to the input buffers of both $PU_1$ and $PU_2$ using type 2 chaining as shown in Listing 4.2. The chaining information is enclosed between \textit{begin} and \textit{end} instructions. The two copies of the value $t_0$ is produced in the $PU_0$ is now moved to both $PU_1$ and $PU_2$ for every firing till an unchain instruction is observed.

\begin{verbatim}
begin  // begin chaining
pu0@out -> pu2@in0 // chained instruction
pu0@out -> pu1@in1 // chained instruction
end    // end chaining
\end{verbatim}

Using type 2 chaining for the Heron’s iteration given by Listing 3.3, it is possible to implement the complete loop of the program by a set of chained PUs as seen in Figure 4.7. The output buffer of $PU_0$ is now chained to four different buffers, and each time when four values are produced, the values are distributed to those four nodes.
The following changes are considered for the implementation of type 2 chaining in SCAD:

- **Firing Process and Sending Process:** The firing rule doesn’t change for the PUs. They can fire when the head of the buffers on all input ports have valid values. After firing, the result is appended to the tail of the output buffer just like in the normal SCAD machine. The sending process simply sends the completed entries from the head of the output buffer.

- **Buffer Implementation:** The buffer implementation is modified to handle chaining with multiple copies. Firstly, each buffer entry is now associated with a flag to indicate chained when `true` and unchained condition when `false`. Secondly, the buffer management is modified. Without chaining, the buffers were simple FIFO queues. Whenever a value is read from the head of the buffer, these values are destroyed and the next values are shifted towards the head of the buffer. With this chaining under consideration, the buffer management becomes more related to a rotating buffer. It is explained as follows:
The buffer checks the flag associated with the entry at the head of the buffer.

- If the flag is *false*, then the *adr* is destroyed. The address-value entries are shifted towards the head of the buffer.
- If the flag is *true*, then the *adr* (and also *val* in case of chaining input buffer with multiple constant) is removed from the head and added to the tail of the buffer by using the integer number as the offset value. This is now a rotating buffer with length equal to the integer number. This way chaining to multiple copies in the buffer can be handled.

**MIB and the Control Unit:** The MIB now has a unique bit to indicate the difference between a chained and a normal move instruction, since the only way to differentiate between them is by observing the `begin` - `end` instructions. When the control unit observes a `begin`, this bit on the MIB will be set to high. From now on, every move instruction sent on the MIB is considered to be a chaining instruction by all the buffers. Now the buffers will not only write the addresses into their buffers but they will also set the special flag for those entries. When an `end` is observed by the control unit, then this bit on MIB is reset. The move instructions sent thereafter are considered to be normal move instructions.

As an example to show the buffer management, consider Figure 4.8. In the first firing, since the flag = *false*, the *adr* is destroyed, but from the second firing onward, the *adr* values are rotated around. Hence, after the first firing, the buffer management changes to rotating actions because the entries have chained flag = ”true”. If the input buffer was chained to a set of constants,
then after every firing the \textit{val} field is also rotated along with the \textit{adr}. The rotating action of \textit{adr} field holds even for sending the values from a chained output buffer.

Note that this implementation of the buffer is possible because even in the normal SCAD machine, the buffers can be read and written at the same time. So, it is easy to implement the rotation action. The only modification that is required will be, when the flag is set, the \textit{adr} is read out from the read pointer and written at the write pointer in the same buffer to produce a rotating buffer action.

**Restrictions to be Checked by the Compiler**

For the auto-generated move programs, the following restrictions have to be considered by the compiler during the code generation. Even for the manually written code, the same restrictions have to be ensured by other tools:

1. For every \textit{begin} instruction, there exists an \textit{end} instruction. This problem is similar to checking for balanced parentheses in a string and hence can be easily solved.
2. \textit{begin} and \textit{end} pairs are not overlapping on all the paths of the control-dataflow graph of the program. This is because only one unchain instruction is used to break the chained connections. So if there are overlapping \textit{begin-end} pairs, then it becomes difficult to identify which connections are removed and which connections are retained for further execution.

**Unchaining Implementation for Type 2**

For type 2 implementation, there is a single unchain instruction. Whenever the control unit sends an unchain instruction on the MIB, all the buffers which have been chained will flush their values and reset the flags associated with the entries. This implementation holds as long as we do not allow overlapping \textit{begin-end} pairs. But one problem exits with this implementation: If the computation of expression takes longer time, then the values in the buffers
Chapter 4. Chaining in SCAD

will be legal copies and they will get flushed when an unchain instruction is observed. So, implementing type 2 Chaining is more complicated.

Drawbacks of Type 2 Chaining

1. It is often the case that a result obtained in one iteration of a loop is used in the next iteration as well as in the condition checked before entering the next iteration. When an output buffer which evaluates this result is chained to multiple input buffers, depending on whether all the values are sent simultaneously on the DTN to those input buffers or one after the other, there are different problems:
   - If the values are sent one after the other, then the order of chaining affects the time at which the result is evaluated. If we send the value sooner to the PU evaluating the branch condition of a loop, the condition is evaluated sooner, or else by the time the condition is evaluated, may be the next iteration could have begun and even completed if the expression is smaller.
   - If all values are sent at the same time, then still a problem exists because, by the time the condition is evaluated, the other expression might have started the execution and this can sometimes destroy the result of the previous iteration, assuming the expression is not very large and the delay is close to the delay of the condition evaluation.

Listing 4.3: A simple loop in a C program

```c
for ( i =0; i <10; i++)
{
    xold= xnew;
    xnew = (xold / 2) + (xold * 8);
}
```

2. Considering a for-loop in a program, for example Listing 4.3. Implementing this with type 2 chaining leads to two independent chains: one that increments $i$ and another that evaluates $x_{new}$. Since evaluation
of \( x_{new} \) is a large expression, its delay is more than incrementing \( i \), so \( i \) becomes equal to 10 even before ten iterations are actually computed by the other chained functional units which calculate \( x_{new} \). This leads to a wrong result, because once \( i \) is 10, the loop exits and the PUs are unchained at the exit of the loop. This is a major drawback in type 2 chaining, because of the formation of independent chains, there is no more control over the number of iterations executed.

3. For implementing type 2, we need to store an integer value per buffer to chain with multiple copies. This integer value is required to implement the rotating buffer action in buffers that are chained as explained in Figure 4.8. In case of the output buffer, this can be determined using the number of copies of the result produced by the PU, i.e., from the instruction which chains the opcode and the number of copies to the PU. However, for the input buffer, there is no way to determine this and without this integer value, it is difficult to decide whether the PUs should be allowed to fire between \( \text{begin} - \text{end} \) instructions. The possible solutions and their drawbacks are as follows:

- The first solution is to introduce a new instruction which basically moves this value to each buffer that will be used in chaining. This is a feasible solution, but it does not align with the concept of having only one type of move instructions in SCAD processors, and in the implementation, the buffer which holds the integer value should be made accessible outside the PU which is not an elegant solution.

- The second solution is to restrict the PUs from firing even if they have valid entries at the heads of the buffers, until an \( \text{end} \) instruction has arrived. This solution changes the data-driven PUs to data- and control-driven PUs which is not aligned with the concept of SCAD since SCAD is meant to have an asynchronous dataflow.
4.2.3 Comparison between Type 1 and Type 2 Chaining

The drawbacks observed in type 2 chaining are really difficult to solve when compared to type 1. Specifically, there are no elegant solutions to solve the problems arising with input buffer implementation and the unchain instruction implementation. In type 1, we could easily obtain multiple copies using DUP operators or concurrent chains of the same expression, and then chain with multiple buffers. Other than this, all the necessary changes required for implementation are not very difficult when compared to type 2. Also in type 2 many of the solutions do not align well with the main concept of SCAD, i.e., have synchronous control and asynchronous dataflow. For example, we could prevent the PUs from firing until the end chain instruction is sent by the control unit but this means the PUs are no more asynchronous. Hence, we can conclude that type 1 chaining is the most suitable method for the SCAD machine.

4.3 F# Implementation of Chaining

| $puI@out => adrU@inI | chains the output buffer $puI@out to some other input buffer $adrU@inI |
| $val => $puI@in0 | chains value $val to the left operand buffer of the input $puI@in0. Similarly for $puI@in1 |
| $(opc,nc) => $puI@opc | chains the opcode $opc and the number of copies $nc to the $puI@opc |

Table 4.1: Syntax of Type 1 Chaining Instructions

Table 4.1 shows the syntax of chaining instructions implemented in SCAD. **LinkInstr** defines the type for these chaining instructions in F#. The three types of chaining instructions are given in Table 4.2. The type description\(^1\) for those chaining instructions are as follows:

- LinkOutBf2InpBf of $address * address

---

\(^1\)Refer to [33] to understand more about type description of different instructions implemented in MiniC
• LinkConst2InpBf of value * address
• LinkOpc2PU of Ops2 * int * addressPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinkOutBf2InpBf(src, tgt)</td>
<td>chains the output buffer src to the input buffer tgt</td>
</tr>
<tr>
<td>LinkConst2InpBf(v, tgt)</td>
<td>chains the constant v to the input buffer tgt</td>
</tr>
<tr>
<td>LinkOpc2PU((op,nc), adrU)</td>
<td>chains the opcode op and the number of copies nc to the option buffer of the PU adrU</td>
</tr>
</tbody>
</table>

Table 4.2: Implementation of Chaining Instructions in F#

<table>
<thead>
<tr>
<th>puI@out :&gt;adrU@inI</th>
<th>unchains the output buffer puI@out to some other input buffer adrU@inI</th>
</tr>
</thead>
<tbody>
<tr>
<td>$val :&gt;puI@in0</td>
<td>unchains val to the right operand buffer of the input puI@in0. Similarly for puI@in1</td>
</tr>
<tr>
<td>(opc,nc) :&gt;puI@opc</td>
<td>unchains the opcode opc and the number of copies nc to the puI@opc</td>
</tr>
</tbody>
</table>

Table 4.3: Syntax of Type 1 Unchaining Instructions

In order to remove the chaining information, we use the unchain instructions in SCAD given in Table 4.3. **UnlinkInstr** defines the type for unchain instructions. Similar to chaining, there are three types of unchain instructions described in the Table 4.4.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnlinkOutBf2InpBf(src, tgt)</td>
<td>unchains the output buffer src from the input buffer tgt</td>
</tr>
<tr>
<td>UnlinkConst2InpBf(v, tgt)</td>
<td>unchains the constant v from input buffer tgt</td>
</tr>
<tr>
<td>UnlinkOpc2PU((op,nc), adrU)</td>
<td>unchains the opcode-number of copies pair (op, nc) from the option lane of the PU adrU</td>
</tr>
</tbody>
</table>

Table 4.4: Implementation of Unchain Instructions in F#
4.4 Example of Chaining

Listing 4.4: SCAD Program for Heron’s Iteration with Type 1 Chaining Implementation

```plaintext
// initialization
0: $121 \rightarrow pu0@in0
1: $0 \rightarrow pu0@in1
2: (addN,3) \rightarrow pu0@opc  // xold = xnew

// loop begins
3: pu0@out \rightarrow pu2@in1
4: pu0@out \rightarrow pu1@in0
5: pu0@out \rightarrow pu4@in1
6: $121 \rightarrow pu2@in0  // chaining instructions
7: (divN,1) \rightarrow pu2@opc
8: pu2@out \rightarrow pu1@in1
9: (addN,1) \rightarrow pu1@opc
10: $2 \rightarrow pu0@in1
11: pu1@out \rightarrow pu0@in0
12: (divN,4) \rightarrow pu0@opc
13: pu0@out \rightarrow pu2@in1
14: pu0@out \rightarrow pu1@in0
15: pu0@out \rightarrow pu4@in0
16: pu0@out \rightarrow pu4@in1
17: (lesN,1) \rightarrow pu4@opc
18: $13 \rightarrow cu@in1
19: pu4@out \rightarrow cu@in0

// discard the extra values in these output buffers to the null buffer
20: $121 :> pu2@in0  // unchaining instructions
21: (divN,1) :> pu2@opc
22: pu2@out :> pu1@in1
```
23: (addN, 1) :> pu1@opc
24: $2 :> pu0@in1
25: pu1@out :> pu0@in0
26: (divN, 4) :> pu0@opc
27: st -> lsu@opc    // store the result
28: $0 -> lsu@in0
29: pu0@out -> lsu@in1

The SCAD program for Heron’s iteration is shown in Listing 3.4 and the program with chaining implementation is shown in Listing 4.4.

4.5 Disadvantages of Chaining

Even though chaining is advantageous in SCAD, it still has some drawbacks which are as follows:

- In general, an implementation of a program with chaining will use more number of processing units when compared with the program without chaining instructions. This can be observed by comparing Listing 3.4 and 4.4 for the Heron’s iteration example. Since, with chaining, the processing units is temporarily fixed to a node of an expression, it is not available for the calculation of other nodes in the basic block. In order to measure the amount of increase in the number of processing units required with chaining of variables, we set up some experiments by adding constraints to the existing based SCAD code generator and analyze results of these experiments as explained in Chapter 6.
- In order to handle the implementation of chaining instructions, new logic must be added to each processing unit. This results in increased chip area.
- The effect of chaining reduces the number of move instructions issued by the control unit. This results in reduced traffic only on the MIB, but there is no effect on the traffic on the DTN. The DTN will still transport the same number of values as in a normal SCAD implementation.
5 Expression Generation

In this chapter, we describe a simple expression generation algorithm that can be used to detect potential places in the program to use chaining instructions. Currently, a complete compilation flow that starts with a program in high-level language and that produces a corresponding move program for the SCAD machine is not available. So, the output of the expression generation phase can be used to manually insert the chaining information into the SCAD programs. We first describe the concept of expression generation followed by the algorithms that were developed during this thesis used in automatic expression generation. The last part gives a complete flow-diagram of the automatic expression generation process.

5.1 The Concept of Expression Generation

The objective is to design an automatic method which analyses the source code of the application in MiniC to obtain the most beneficial expressions that can be further implemented in the SCAD processor by chaining of the functional units. The automatic identification of most profitable expressions in the program is similar to the instruction set extension (ISE) problem commonly addressed in ASIP design. A few of the existing approaches were discussed in the Section 2.2.

The problem can be stated as follows: Identify large expressions in the program which basically consists of detecting clusters of basic operations that when implemented by using chaining in SCAD shall improve some metric. In particular, we want to reduce the number of move instructions, thereby
reducing the contention on the MIB. In traditional RISC-based ASIPs, the ISE generation process has restrictions like: (1) the new instruction should produce only one result and (2) the number of input operands should be limited to a small number, since the instruction can encode only few registers. In SCAD, since registers are not used and computation proceeds in dataflow order, the number of input operands in a large expression need not be limited. Only the number of outputs is currently limited to one since the processing units in the SCAD machine consists of only one output port. Hence, we will consider multiple-input single-output expressions in our method. However, if the SCAD machine has processing units with multiple output ports, then multiple-input multiple-output instructions can also be implemented with chaining.

The idea is to find most recurring clusters of operation in the application program. For example, consider an expression inside a body of the loop statement, if this expression is implemented as a single instruction in case of an ASIP, then the performance speed up would be much larger than implementing it as a sequence of basic instructions. Similarly, in a SCAD processor, we can implement this expression as a single fused operator which is achieved by chaining of processing units as described in Chapter 4. With the chaining of processing units according to the structure of the expression identified in the program, the number of move instructions that are issued by the control unit during the execution will be reduced which reduces the contention on the MIB. This is an important metric in case of SCAD.

5.2 Algorithms used in Expression Generation

The algorithms for the different functions used in the expression generation phase are summarized here. A call graph shown in Figure 5.1 provides an overview. The starting point is the intermediate code or Cmd code obtained from the application source code in MiniC as explained in Section 3.1. When the function \texttt{EXPRESSION GENERATION} terminates, it will return a dictionary
of expressions and their corresponding occurrence frequency, and also the corresponding line numbers in the Cmd code where the expression occurs, which can be used to insert the chaining instructions.

The first step in ExpressionGeneration given by Algorithm 1 is a for-loop which collects all the different expressions present in the application program in the list ExpressionTemplates. This list contains all the basic operators as well as large clusters of operators found in the program. In order to obtain the expression, we start at each line in Cmd code which has a three-operand format- \( y = x \ op \ z \), and we add the root \( y \) of the expression to the list nodes and call the procedure FindExpression given by Algorithm 2 to find the largest possible expression starting at this root node. If \( y \) is a local variable of the function, then a call to FindExpression will return a basic operation, and if it is a temporary variable, then it returns the largest possible expression rooted at that temporary variable. This is because, in the Cmd code a large expression tree is broken into a series of three-operand statements and the intermediate nodes are assigned to temporary variables. FindExpression uses a pre-order traversal of the expression tree [35] to obtain the tree rooted at a given node.

Once all the expression templates are collected in the list ExpressionTemplates, we would like to obtain the number of occurrences for each expression tree in order to find the most suitable candidates for chaining. This is obtained in the second for-loop of Algorithm 1. It uses the procedure MapExpression described by Algorithm 3. For each expression obtained in ExpressionTemplates, we search on each Cmd code line for a match. If a match exists, then we increment the count value and also record the line number where the match was found.
Algorithm 1 \textsc{ExpressionGeneration}: To generate all expressions with their occurrence frequency in the program

\begin{verbatim}
\begin{algorithm}
\caption{ExpressionGeneration\textsc{}}
\begin{algorithmic}[1]
\State \textbf{Input:}
\begin{itemize}
\item \textit{cmdp}: Cmd program of an application
\end{itemize}
\State \textbf{Output:}
\begin{itemize}
\item \textit{ExpressionDict}: Dictionary with key as expression and value as a tuple of count and Cmd code line number
\end{itemize}
\State \textbf{Local variables:}
\begin{itemize}
\item $l$: int
\item \textit{nodes}: List of string
\item \textit{ExpressionTemplates}: List of expressions
\item \textit{matchList}: List of List of int
\item \textit{lineList}: List of int
\item \textit{count}: int
\end{itemize}
\Procedure{ExpressionGeneration}{\textit{cmdp}}
\State \textbf{SomeInitialization}()
\For{each $l$ of form $x = y \ op \ z$ in \textit{cmdp}}
\State \textit{nodes}.Add(\textsc{WriteVariableOfCmd}($l$)) \Comment{add the variable written by Cmd code into \textit{nodes} which become the root node of the expression}
\State \textit{ExpressionTemplates}.Add(\textsc{FindExpression}(\textit{nodes},[])) \Comment{obtain the expression starting on line $l$ and add it to the list of expressions}
\EndFor
\State \textit{expression}.Clear() \Comment{clear the list \textit{expression}}
\State \textit{nodes}.Clear() \Comment{clear the list \textit{nodes}}
\For{each expression $e$ in \textit{ExpressionTemplates}}
\State \textit{matchList} \leftarrow []
\State \textit{count} \leftarrow 0
\For{each line $l$ in Cmd Program}
\State \textit{expression} \leftarrow $e$
\State \textit{nodes}.Add(\textsc{WriteVariableOfCmd}($l$)) \Comment{add the variable written by Cmd into \textit{nodes}}
\State \textit{lineList}.Clear() \Comment{Clear \textit{lineList}}
\State \textsc{MapExpression}(\textit{nodes}, \textit{expression}, \textit{lineList}) \Comment{Search if a match exists for $e$ on line $l$}
\EndFor
\State \textit{ExpressionDict}.Add($e$, (\textit{count}, \textit{matchList}))
\EndFor
\State \Return \textit{ExpressionDict} \Comment{return the collection of expressions}
\end{algorithmic}
\end{algorithm}
\end{verbatim}

\end{algorithm}

56
Finally, Algorithm 1 returns a dictionary $ExpressionDict$ which contains all the expressions with their static occurrence count and the line numbers where they are matched in the Cmd code. Static occurrences means the number of occurrences in the program which will be different to the number of actual executions because of the control-flow constructs used in the program. These results are further combined with simulation runs of the application using the Abacus simulator to obtain the dynamic occurrence count for each expression. Note that the dynamic occurrence of an expression depends on the input values given to an application program and hence can vary with different input sizes and the various control constructs in the program.

**Algorithm 2 FindExpression**: To find the largest expression from a given root node

**Input:**
- $nodes$: List of string
- $expression$: List of operator (Initially empty list)

**Output:**
- $expression$: List of operator

**Local variables:**
- $root$: string

1: procedure FindExpression($nodes, expression$)  
2: if $nodes \neq \emptyset$ then $\triangleright$ if the list $nodes$ is not empty  
3: root $\leftarrow$ nodes.popAt[0] $\triangleright$ pop the first node from $nodes$  
4: $expression$.append(GetOperator(root)) $\triangleright$ add operator corresponding to root node  
5: if isTempVariable(root) then $\triangleright$ only when root node is temporary variable, we consider its children  
6: nodes.Add(GetLeftChildNode(root))  
7: nodes.Add(GetRightChildNode(root))  
8: end if  
9: FindExpression($nodes, expression$) $\triangleright$ recursive call till there are no more nodes left to be explored in the tree  
10: else $\triangleright$ whole tree has been traversed  
11: return $expression$ $\triangleright$ return the expression tree  
12: end if  
13: end procedure
Algorithm 3 MapExpression: To enumerate different expressions obtained in the given application program

Input:
- nodes: List of string
- expression: List of operator (Initially empty list)
- lineList: List of int (Initially empty list)

Output:
- Increments count if a match has been found at line \( l \) in Cmd
- Modifies lineList if a match has been found at line \( l \) in Cmd else it is empty
- Adds lineList to matchList which holds all matches for the given expression

Local variables:
- root: string
- Oper: operator

1: procedure MapExpression(nodes, expression, lineList)
2:   if expression \( \neq \emptyset \) then
3:     root \( \leftarrow \) nodes.popAt[0] \( \triangleright \) pop the first node
4:     Oper \( \leftarrow \) expression.popAt[0] \( \triangleright \) pop the first operator
5:     if Oper == [ ] then \( \triangleright \) when operator is empty then there is no need to match
6:       lineList.Append(-1) \( \triangleright \) line number = -1 for leaf nodes
7:     else
8:       if Oper == GetOperator(root) then
9:         lineList.Append(GetLineNumber(root))
10:        nodes.Add(GetLeftChildNode(root))
11:        nodes.Add(GetRightChildNode(root))
12:        ExpressionMap(nodes, expression, lineList)
13:     end if
14:   end if
15: else \( \triangleright \) found a match for the given expression \( e \)
16:     count = count + 1
17:     matchList.Add(lineList)
18: end if
19: end procedure
5.3 A Complete Scheme for Expression Generation

The flow-chart in Figure 5.2 gives the complete scheme used for the generation of large expressions from given application program. It is described as follows:

1. The front-end of the MiniC compiler translates a MiniC program into Cmd program.
2. The Cmd program is used by the expression generation phase which implements EXPRESSIONGENERATION given in Algorithm 1. This collects all expressions and their static occurrence frequency in ExpressionDict.

3. From the Cmd program, we also obtain the Abacus assembler program.

4. A simulation of the Abacus program on the Abacus processor simulator obtains the number of executions of each assembly instruction.

5. A map from Cmd to Abacus instructions is generated in the expression generation phase. This map is combined with the Abacus simulator output to calculate the dynamic occurrence count of each line in the Cmd program.

6. By combining the number of static occurrences obtained in Step 2 with the number of actual executions of each line in the Cmd program, we obtain the total number of executions of each expression for a given input value.

7. The output is set of expressions with their dynamic and static occurrence frequencies, and their point of occurrences in the Cmd program. The set of expression trees obtained along with their number of executions can be used to manually add chaining information to the corresponding SCAD program.

5.4 Results of Expression Generation

We evaluated our expression generation algorithm for different application programs. The first set of application includes three different sorting algorithms: Bubble, insertion and selection sort. Figure 5.3 shows the different large expressions and their corresponding number of executions obtained. Note that the expressions are written by listing the operators in pre-order traversal form and "[]" refers to a leaf node in the tree representation of an expression. As observed from the figure, in sorting algorithms a subtraction(SubN) followed by compare(LeqN) operation occurs very frequently and this could be a suitable candidate to be implemented with chaining in SCAD. The next set of applications includes the discrete cosine transform(DCT), a
finite impulse response (FIR) filter and a 2x2 matrix multiplication. A plot of
different expressions trees and their occurrence frequencies are shown in Fig-
ure 5.4. Again, we obtain large expressions with high occurrence frequency
and these are also good candidate instructions for this set of application.

![Figure 5.3: Expression trees in different sorting algorithms](image)

Even though the results obtained refer to the dynamic number of occurrences
in the program and are dependent on the input size, we can easily see that
with the increase in the input size, the number of executions of these expres-
sions will also increase. Therefore, with large sizes of input, use of chaining
in SCAD becomes more and more advantageous.
Figure 5.4: Expression trees occurring in DCT, FIR and matrix multiplication programs
6 Experiments

Since a fully functional compiler is not available at present, the SAT-based code generator is used to set up the experiments for chaining. This code generator processes only basic blocks of different sizes and generates optimal SCAD move program. Our experiments will also only consider chaining of instructions within basic blocks and not across basic blocks.

In 3.2.6, we explained the SCAD code generation using SAT solvers, wherein the different boolean constraints were set up and used to obtain optimal code with minimum PUs of a SCAD machine. With chaining under consideration, we are aware that the number of PUs required will be obviously more than that without chaining. In order to evaluate the increase in the number of PUs, we have used an additional constraint while constructing the boolean propositional formula which is solved by the SAT solver as follows: When we say a variable is chained to a PU, then this PU will not be available for other executions except for that assigned variable, until that variable is unchained.

In order to enforce this for every chained variable, we define a new boolean constraint for dedicated PU assignment for chained variables: If $V_{ch}$ is the set of all variables that are chained, then the constraint is given by Equation 6.1 (note that $V_{ch} \subset V_{tgt}$):

$$\bigwedge_{x_i \in V_{ch}} \bigwedge_{j=0}^{p} \alpha_{i,j} \rightarrow \bigwedge_{x_k \in V_{j\neq k}} \neg \alpha_{k,j}$$

(6.1)

The existing random basic block generator that takes the number of nodes $n$ and number of levels $l$ as input was used to generate basic blocks for this experiment. Here, the basic block is generated by randomly choosing two
predecessors of every node ensuring that the DAG has \( l \) levels. For a basic block with \( n \) nodes, \( l = \{2, \ldots, n - 1\} \) levels are possible. A random subset of the target variables in the basic block will be selected as chain variables. For the cases \( l = 2 \) and \( l = 3 \), there are only one and two target variables, respectively, in the basic block. So chaining in this case is trivial and hence for the following experiments only basic blocks with expressions having a minimum of four levels were chosen. In each level \( l \), we can chain \( c = \{1, \ldots, l - 2\} \) variables. For every tuple \((n, l, c)\), 100 basic blocks are generated in this experiment. Next, the boolean constraints are generated as explained in Section 3.2.6. Apart from these constraints, for a SCAD machine with chaining, the additional constraints are obtained using Equation 6.1. A wrapper function around the custom SAT solver based on the DPLL algorithm [37] was used to determine the minimum number of processing units required in the SCAD machine with and without chaining under consideration. The results obtained are as follows. Figure 6.1 shows the increase in the average case and the worst case number of processing units required in SCAD with chaining half of the basic block and the entire basic block when compared to not chaining any variables in the basic block. We can infer to the following points from Figure 6.1: When the entire basic block is chained, the number of processing units required will be equal the number of lines in the basic block. For example, with \( n = 10 \), the maximum value of \( l \) is 9 and hence, there are 8 target variables which when chained require 8 processing units. The number of processing units required when half of the basic block is chained is approximately half the number of processing units required when the complete basic block is chained.
Figure 6.1: Variation in the number of processing units used in a SCAD machine with half, full and no chaining in basic blocks

Figure 6.2 shows the variation in the maximum number of processing units required with different numbers of chained variables in the basic block. We can infer that, as the number of chaining variables increases, the number of processing units required for the execution of the basic block also increases. After allocating the processing units to chained variables, the SAT solver tries to minimize the number of processing units required for the execution of the remaining basic block. In general, we observe that the following conclusion holds:
Maximum number of PUs with chaining =

\[ \text{number of chained variables} + \text{number of PUs required without chaining} \]

Figure 6.2: Variation in the number of processing units in SCAD machine for different number of chained nodes

As explained in [29], the average case and the worst case time taken by the SAT solver to produce the results grows with the number of nodes in the basic
block, since an increase in the number of nodes leads to an increase in the number of boolean constraints in the propositional logic formula. In addition to that, when we add additional constraints to assign dedicated processing units for chained variables, more boolean constraints will be added to this formula leading to an increased solving time for the SCAD machine with chaining. This can be easily inferred from Figure 6.3. The average case time for solving the boolean formula with chaining constraints is more than the worst case time taken by the solver with no chained variables, and this increases with \(n\). Even for basic blocks with \(n = 10\), the worst case time taken by the SAT solver increases exponentially when chaining is used.

Figure 6.3: Variation in the SAT solver execution time with and without chaining in the basic block
7 Conclusion

In this work, we presented a new concept called chaining of processing units which can be used to configure exposed datapath architectures, in particular, the SCAD machine to make it application-specific. Depending on whether we allow chaining of multiple copies or not, we can have different types of chaining in SCAD. After comparing them qualitatively, we decided to choose the chaining functionality which does not allow chaining of multiple copies to the buffers of the processing units. We then also presented a simple pre-order traversal based approach which can be used to identify large expressions in the program that can be implemented with the chaining instructions on a SCAD machine. By using chaining in SCAD, we could easily reduce the number of move instructions issued during runtime. Although the idea of chaining gives performance benefits and saves energy by reducing the number of instructions moved, it also has a major disadvantage that the number of processing units required with chaining is much higher than that without chaining. We tried to demonstrate this by using the existing SAT-based SCAD code generator for basic blocks. By analysing the results, we observed that increase was measurable and predictable. Therefore, we can conclude that, if the number of chained variables can be analysed and kept within a limit, then chaining can be used to improve the overall performance in terms of both energy and runtime for a given application on the SCAD machine. Of course, the real hardware implementation requires more logic to be added in order to implement this idea and the effect of this increased logic is yet to be explored.

That being said there is still much scope for future development. Once the code generator for the SCAD machine has been developed, it can be
integrated with the output of the expression generation phase to develop a completely automatic framework that can generate move programs with and without chaining instructions. Then we can run some benchmark programs to see the usefulness of chaining in real programs.

Also the currently used methodology for expression generation is not optimal, i.e., it tries to find candidates based only on the number of occurrences. We could instead use the existing algorithms that are widely used in ASIP design in order to find better candidates for implementation with chaining.

The problem of chaining multiple copies of a result to multiple buffers is yet to be solved in an efficient way without affecting the core principle of exposed datapath architectures. This would give higher performance for basic blocks and the overall program. In fact, if this type of chaining is made available then the entire basic block can be executed by issuing only chaining instructions and we could be able to emulate the concept of the TRIPS architecture [6] in this way.

We also did not explore the impact of chaining expressions occurring across basic blocks. If the expression is large, then a larger number of processing units will have to be chained for a long portion of the program, which in turn could result in worse performance due to an insufficient number of processing units. This needs still to be explored in order to decide whether chaining across basic blocks is beneficial or not.

So, all things considered, we can finally conclude by saying that the concept of chaining is a beneficial addition to the SCAD architecture to make it application-specific, with a lot of possibilities yet to be explored in the future.
Bibliography


