A Syntax Directed Imperative Language Microprocessor for Reduced Power Consumption and Improved Performance

Glenn Coates

PhD

University of York

Computer Science

August 2018

Abstract

This thesis investigates high-level *Instruction Set Architectures* (ISAs) and supporting processor architectures. A *Syntax Directed Imperative Language Processor* (SDLP) and associated ISA have been defined with the ultimate aim of reducing power consumption and improving performance.

The findings of this thesis suggest that there may be a number of benefits of the SDLP over traditional ISAs and architectures. Initial results suggest that the SDLP ISA places less burden on the memory system by reducing the number of instructions executed for a given program. It also appears that the SDLP could reduce the number of interactions with the memory system for data. These results are significant since a large portion of the total power for a system is consumed by the memory system. It is illustrated how the SDLP requires fewer cycle counts for the equivalent throughput of traditional microprocessor architectures. The implication is that further performance improvements could be obtained with uniprocessors, before considering multiprocessors.

The main contributions of this thesis include:

- The design of a hybrid control flow and data flow architecture with a supporting Instruction Set Architecture;
- Implementation of an assembler and software-based cycle accurate simulator for the SDLP processor;
- Comparisons of the SDLP architecture with traditional CISC and RISC processors;
- It has been shown that high-level ISAs and supporting processor architectures can reduce the burden on the memory system for both instructions and data; and can reduce the cycle count of programs.

Contents

1 Introduction **13**

1.1 Microprocessor Trends 13

1.2 Microprocessor Limits 14

1.3 Research Questions 16

1.4 Thesis Structure 17

2 Literature Review **18**

2.1 Processor Architectures 18

2.1.1 Complex Instruction Set Computers (CISC) 19

2.1.2 Reduced Instruction Set Computers (RISC) 21

2.1.3 Reduced Operand Set Computers (ROSC) 23

2.1.4 Very Long Instruction Word Computers (VLIW) 25

2.1.5 Dataflow Computers 25

2.1.6 Language Specific Processors (LSPs) and High-Level Language Computer Architectures (HLLCAs) 35

2.1.7 Syntax Directed LSPs (SDLSPs) 35

2.2 Multiprocessors 37

2.2.1 Theoretical and practical Speedup 37

2.3 Memory Hierarchy 38

2.3.1 Caches 38

2.3.2 Hardware Stack 44

2.3.3 Scratch Pad Memory 46

2.3.4 Multiprocessor Memory Hierarchy 47

2.4 Summary 52

3 Problem Analysis **55**

3.1 The Motivation for Multiprocessor Systems . . . 55

3.2 The Challenges for Multiprocessor Systems . . . 58

3.2.1 Computational Complexity . . . 58

3.2.2 Multiprocessor Scaling . . . 60

3.3 Understanding what CPUs Spend Most Time Doing . . . 60

3.3.1 Benchmarks . . . 62

3.3.2 Energy and Power Measurements . . . 65

3.4 Where Can Improvements be Made? . . . 72

3.5 Opportunities for Abstract Instructions . . . 74

3.6 Opportunities for Abstract Expressions . . . 77

3.7 Summary . . . 79

4 Expression Engine **81**

4.1 Background . . . 81

4.2 Expression Engine Design . . . 84

4.3 Expression Encoding . . . 88

4.4 Example Encodings . . . 92

4.5 Comparison of Expression Encoding . . . 103

4.6 Summary . . . 108

5 Abstract Instructions **110**

5.1 Background . . . 110

5.2 Instruction Set Architectures . . . 111

5.3 Instruction Set Design . . . 115

5.4 Instruction Encoding . . . 118

5.5 Example Instruction Encoding . . . 119

5.6 Summary . . . 126

6 An SDLP Architecture and Simulator **128**

6.1 Initial Comparisons . . . 128

6.2 SDLP Architecture and Behavioural Description . . . 136

6.2.1 SDLP Execution Model . . . 137

6.3 SDLP Software Simulator . . . 143

6.4 Comparison of Simulation Measurements . . . 147
6.5 Summary . . . 174

7 Conclusions **176**
7.1 High-Level ISAs and Supporting Processor Architectures can Reduce the Burden on the Memory System for Both Instructions and Data . . . 176
7.1.1 The Burden that RISC Processors Place on the Memory System and the Projected Energy Costs . . . 177
7.1.2 Comparing Assembled Expressions and ARM Thumb . . . 178
7.1.3 Comparing Loads, Stores and Instructions for Assembled SDLP and ARM Thumb . . . 178
7.2 High-Level ISAs and Supporting Processor Architectures can Reduce the Cycle Count of Programs . . . 179
7.3 Research Contributions . . . 180
7.4 Closing Remarks . . . 181

8 Future Work **182**
8.1 Expression Engine . . . 182
8.2 Practicalities . . . 184
8.3 ASIC Implementation . . . 184

A Function Calls and Returns **186**

B Assembler User Guide **191**
B.1 Introduction . . . 191
B.2 Memory Model . . . 191
B.3 Data Segment and BSS Segment Declaration and Definition . . . 193
B.3.1 Expressions . . . 194
B.3.2 While Instructions . . . 197
B.3.3 If Instructions . . . 197
B.3.4 If-else Instructions . . . 198
B.3.5 Example SDLP Program . . . 198

List of Tables

2.1 Common Addressing Modes, adapted from Silc et al. [1, p.7] 20
2.2 Example C code with corresponding CISC and RISC based assembly code 22
2.3 Comparative access times, adapted from Hennessy and Patterson [2, p.79, Fig. A] 38
2.4 Summary of differences between CISC, RISC and ROSC 52

3.1 Raw Simulation Results 64
3.2 Verma et. al. Energy Access Results, taken from [3, p.22] 67
3.3 System Stack 73

4.1 SDLP Operators 89
4.2 RP0 - RP5 Meanings 89
4.3 D Meanings 90
4.4 C Expressions and Equivalent SDLP Assembly 92
4.5 Disassembly of expr_assign_ptr_addr_of_a0 94
4.6 Disassembly of expr_count_less_than_numElement 95
4.7 Disassembly of expr_inc_count 97
4.8 Disassembly of expr_element_greater_than_biggest 99
4.9 Disassembly of expr_assign_element_to_biggest 100
4.10 Disassembly of expr_inc_ptr 102

5.1 Programming Paradigms 111
5.2 ISAs 112
5.3 Types of High Level Language Systems, taken from [4] 113
5.4 C keywords mapping onto SDLP Instructions 116
5.5 Rewriting a for loop using while 116
5.6 C Language Constructs and SDLP Support 117

5.7 SDLP While Loop Disassembly - Memory Dump 125
5.8 SDLP While Loop Disassembly - Segment Offset Table 125
5.9 SDLP While Loop Disassembly - Text Segment 125

6.1 Comparison Results for Linear Search 132
6.2 Cycle Times for MicroBlaze Soft-core Processor, adapted from [5] 145
6.3 Cycle Times for ARM7TDMI, taken from [6, p.8] 146
6.4 Difference between ARM and SDLP for the Linear Search Program 149
6.5 Benchmark Categories 150
6.6 Benchmark Clock Cycle Counts for ARM and SDLP 172

A.1 SDLP Function Call and Return Instruction Sequences derived from 8086 190

B.1 SDLP Operators 195
B.2 RP0-RP5 Meanings 196
B.3 D Meanings 196

List of Figures

1.1 Example Heat Sink for a Modern Microprocessor, taken from [7] 14

2.1 Typical CISC Instruction Encoding, adapted from Hennessy and Patterson [8, p.A-22] 19
2.2 Superscalar Pipeline 21
2.3 RISC Instruction Encoding, adapted from Hennessy and Patterson [8, p.A-22] . . . 23
2.4 Dataflow Computation from Sharp [9, p.55] 27
2.5 Dataflow Computation Snapshot (a) from Sharp [9, p.56] 28
2.6 Dataflow Computation Snapshot (b) from Sharp [9, p.56] 29
2.7 Dataflow Computation Snapshot (c) from Sharp [9, p.56] 29
2.8 Dataflow Computation Snapshot (d) from Sharp [9, p.56] 30
2.9 While loop Dataflow Graph based on Sharpe [9, p.32] 31
2.10 Manchester Dataflow Block Diagram, taken from Gurd et al. [10, p.40] 33
2.11 TINY Architecture, from Audsley and Ward [11] 36
2.12 TINY Expression Tree, based on Audsley and Ward [11] 37
2.13 Four way set associative cache, taken from Sloss et al. [12, p.413] 40
2.14 UMA/SMP Memory Hierarchy with Private Caches, adapted from Baer [13, 264] . 48
2.15 NUMA, Distributed Memory Hierarchy, adapted from Baer [13, 264] 48
2.16 Basic MESI Cache Coherence Finite State machine - local cache controller, taken from [13, 274] 50
2.17 Basic MESI Cache Coherence Finite State machine - remote cache controller, adapted from [13, 274] 51

3.1 40 Years of Microprocessor Trend Data, taken from National Research Council [14, p.55]. Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten Dotted line extrapolations by C. Moore. 56

3.2 Speed Up Possibilities for 50%, 75%, 90% and 95% parallelism, taken from [15] . . 59
3.3 Simulation Results showing number of Loads, Stores and Simulated Instructions . 63
3.4 basicmath large Power Projections showing number of Writebacks, Misses and Hits 67
3.5 bitcnt large Power Projections showing number of Writebacks, Misses and Hits . . 68
3.6 qsort large Power Projections showing number of Writebacks, Misses and Hits . . . 68
3.7 susan large Power Projections showing number of Writebacks, Misses and Hits . . 69
3.8 dijkstra large Power Projections showing number of Writebacks, Misses and Hits . 69
3.9 patricia large Power Projections showing number of Writebacks, Misses and Hits . 70
3.10 blowfish large Power Projections showing number of Writebacks, Misses and Hits . 70
3.11 adpcm large Power Projections showing number of Writebacks, Misses and Hits . . 71
3.12 crc32 large Power Projections showing number of Writebacks, Misses and Hits . . . 71
3.13 fft large Power Projections showing number of Writebacks, Misses and Hits 72

4.1 Quantifying the Opportunity for Parallelism, Encoding Length and Programmability for Application-Specific ASIC, Uni-core, Multicore and the SDLP, in terms of High, Medium and Low 83
4.2 Expression Engine 87
4.3 expr_assign_ptr_addr_of_a0 93
4.4 expr_count_less_than_numElement 93
4.5 expr_inc_count 96
4.6 expr_element_greater_than_biggest 98
4.7 expr_assign_element_to_biggest 98
4.8 expr_inc_ptr 101

5.1 SDLP Encoding - while 118
5.2 SDLP Encoding - if 119
5.3 SDLP Encoding - if else 119

6.1 SDLP Block Diagram 136
6.2 Pipeline 138

A.1 Function Stack Frame 187

B.1 SDLP Memory Model 192
B.2 Expression Engine 194

Acknowledgements

I would like to extend my sincere thanks and gratitude to Professor Neil Audsley. You have been a great mentor, guide and inspiration. Thank you for keeping me motivated and on track especially during difficult times.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other University. All sources are acknowledged as references.

Chapter 1

Introduction

1.1 Microprocessor Trends

Microprocessors are ubiquitous and used in almost every aspect of modern life. Significant portions of the population use multiple computers on a regular daily basis, in the form of smart-phones, desktop computers, tablet computers, gaming consoles and television and video broadcasting systems. The rapid increase in the performance of microprocessors has led to a huge increase in demand over the years. This demand motivates processor vendors to increase performance further, in what appears to be a cyclical feedback loop.

Microprocessor speeds have been increasing exponentially since their invention circa 1971, following Moore's Law [16]. Moore's Law states that the number of transistors laid out on a silicon chip doubles approximately every two years. This trend began in the early 1970s and is continuing to the present day. It follows that transistors have effectively been shrinking in size approximately every two years. The clock speeds of microprocessors have also been increasing exponentially during this time; clock frequencies have increased by three orders of magnitude since the early 1970s.

However these trends cannot carry on indefinitely and are coming to an end; clock speeds have remained largely unchanged since circa 2005. There are two reasons why this level of performance improvement cannot continue for practical computer systems.

Firstly, there is a practical limit on the Instruction Level Parallelism (ILP) that can be achieved on a uniprocessor system and is reaching its limits for single-core processors. Designers have exhausted most avenues for any further meaningful performance improvements. For the past 50

years, processor designers have mainly focused on *optimising* existing designs; fundamentally, there have been very few *revolutions*.

Secondly, as processor clock speeds increase, so does power consumption. Condensing the number of transistors on a silicon chip works to a certain extent. As the transistors are scaled down, the switching speed increases making the processor faster. However, as transistors get smaller, *power density* increases because power consumption does *not* decrease linearly with size. This means that heat dissipation becomes so problematic that it can damage the silicon, National Research Council [14]. Processor power consumption is exceeding the few hundred watts that can be dissipated in a practical computer system [14]. Figure 1.1 illustrates a typical heat sink for managing heat dissipation for a modern microprocessor.

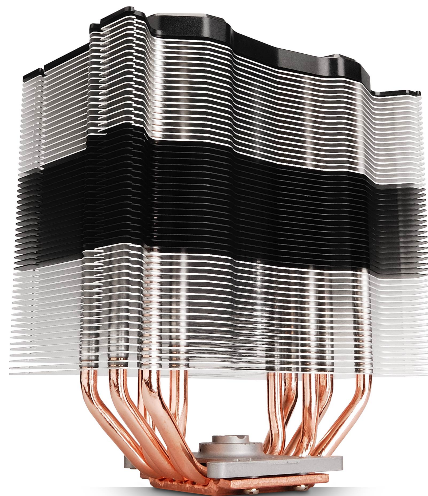

Figure 1.1: Example Heat Sink for a Modern Microprocessor, taken from [7]

1.2 Microprocessor Limits

Even though huge performance gains have been made during the past 50 years, there was always an inevitable limit on how long such gains could be made. An alternative to increasing the speed of a single processor to achieve higher performance is to employ *multiprocessors* which offer hardware parallelism coordinated by a single operating system. Instead of relying on a single processor, multiple microprocessors and the necessary associated communication busses are laid out on the silicon. Performance can be improved since there are more processors, and heat dissipation can be

managed by distributing the power consumption over a larger area of silicon. It appears processor vendors have been cornered into a situation where they have no option but to employ multiprocessors in an attempt to continue delivering improvements. Unfortunately, this strategy is severely limited and any performance gains will appear insignificant compared with those achieved over the past 50 years. This is because of the inherent limitations of multiprocessors and the difficulty in programming them.

Most RISC ISAs are based on the work described by Katavenis [17]. His thesis describes the benchmarking of C programs to understand the instructions executed most frequently in general computing domains. Whilst noting that such exercises are compiler dependent (since the compiler will have been designed to select a constrained set of instruction combinations), this allowed processor designers to focus on the *core set* of instructions that are selected the most often by the compiler. This enabled the fundamental data paths to be optimised by removing unnecessary circuitry and reducing the *critical path*. Much of this work was done in the early 1980s when the dominant issue was performance. At that time the use of microprocessors was more limited and analogue computers were certainly more prevalent than they are now.

The fundamental ISA and architecture of a processor and surrounding subsystems dictate to some extent the basic performance ultimately achievable. In order to optimise performance further, designers employ techniques to improve the *common case*. These approaches are aimed at reducing the *average* execution time. This inadvertently means that temporal non-determinism is introduced into the fundamental processor design.

There have been many alternative approaches to processor design over the past decades, although many have failed in the commercial world. Examples include Stack and Dataflow architectures. The reasons for this may be technical, commercial or a combination of the two. Some have been unfairly referred to as *anti-patterns* or *dead-ends*. An example cited by Nurmi [18] is the *Inmos Transputer*. It is claimed that a significant reason for its demise was *Occam*. Occam is a niche language used by the Transputer, which failed to compete with mainstream languages. However, it does not necessarily follow that because an approach fails to become popular, that it was a bad idea. There are many environmental factors that contribute to the success of a given system: technical, academic, commercial and momentum. There are many examples of where technical ideas fail to become adopted by the mainstream to begin with, but later become widespread. An example of this is the Java Virtual Machine (JVM) and Java Bytecode (JBC) which were greatly influenced by earlier work on Forth [19].

1.3 Research Questions

Having a narrow view towards research may result in limiting processor design and evolution since it discourages *revolutionary* thinking. Previous *old* processor designs may now be more viable faced with the modern day issues of power consumption and the associated limitation of performance improvement. In order for processor technology to progress, it is necessary to reconsider alternatives with a view to solving new and emerging challenges. This may include reconsidering the design of the ISA and aspects of the architecture. *Revolution* over *evolution* should not be discounted, at least from a research perspective, since there is a limit to how far a mature design approach can be optimised. Commercial trends should not limit technical research.

There appears to be an implicit assumption that ISA design has reached a *sweet spot* and Reduced Instruction Set Computers (RISC) and Complex Instruction Set Computers (CISC) are already at an optimum in terms of performance and power consumption. A prevalent industry opinion also appears to accept that RISC processors are best for reduced power consumption. It has been suggested by Hennessy and Patterson that the only way that performance can be increased and power consumption reduced further is via domain specific architectures [2, ch.7]. The general idea is that processors implementing domain specific algorithms and tasks can be used in conjunction with general purpose processors. However, there may still be opportunities for improving performance and reducing power consumption by considering the general purpose ISA.

This thesis proposes that *high-level ISAs and supporting processor architectures can reduce the burden on the memory system for both instructions and data; and can reduce the cycle count of programs.*

By reducing the burden on the memory system it may be possible to reduce cache sizes. This may have the effect of reducing power consumption. Reducing the cycle count for programs whilst at the same time limiting the power consumption per clock cycle, may reduce overall power consumption further.

Whilst reducing the overall power consumption is a central motivation for the thesis, the results can only be determined after a full ASIC implementation of a processor and compiler have been developed. This is out of scope for this thesis and is noted for future work. Programmability of the processor is not a focal point for the thesis, again this is left as future work when developing an appropriate compiler.

1.4 Thesis Structure

The thesis will begin with a Literature Review (Chapter 2) describing the different types of processor architectures with a discussion of the advantages and disadvantages of each approach. Multiprocessors are discussed as a way to improve processor performance. Next the Memory Hierarchy is discussed since this has a significant impact on power consumption and performance.

The Problem Analysis (Chapter 3) discusses the motivation and challenges for multiprocessors. To gain an independent opinion, benchmark results are presented to determine what processors spend most of their time doing. Two potential ideas for both increasing performance and reducing power consumption are then introduced. These ideas are explored further in Chapter 4 (Expression Engine) and 5 (Abstract Instructions). Chapter 6 describes an SDLP Architecture and Simulator supporting an *Expression Engine* and *Abstract Instructions*. The results are then explored, drawing conclusions and suggesting future work for the continued development of the SDLP.

Chapter 2

Literature Review

The following is a literature review and is intended to provide a foundation for the material in later chapters. The software interface to a processor is via the *Instruction Set Architecture (ISA)* and processor design approaches including different ISAs are discussed (Section 2.1). This includes the most common processor architectures and other less common varieties which may have been more popular in earlier decades. A discussion is then given illustrating how processor designers have increased performance and throughput for uniprocessors. *Dataflow* architectures aimed at increasing the granularity of parallelism are introduced (Section 2.1.5). This is followed by *Language Specific Processors (LSPs)* and *High-Level Language Computer Architectures* (HLLCAs) (Section 2.1.6) which provide explicit support for a particular language. A relatively new type of processor architecture called *Syntax Directed LSP (SDLSP)* is finally introduced (Section 2.1.7). This is a type of LSP which has an architecture defined by the language that it executes. The motives for the shift towards multiprocessors are then explored (Section 2.2). Since accessing external memory is orders of magnitude slower to access than on-chip memory, a discussion of how memory systems can be organised and managed is covered (Section 2.3).

2.1 Processor Architectures

This section discusses the most common microprocessor architectures. These are Complex Instruction Set Computer (CISC), Reduced Instruction Set Computer (RISC), Reduced Operand Set Computer (ROSC) and Very Large Instruction Word (VLIW) computer. Less common architectures including Dataflow, LSPs and HLLCAs are also discussed.

2.1.1 Complex Instruction Set Computers (CISC)

CISC architecture was the conventional design for microprocessors in the 1960s and 1970s and was typified by a complex Instruction Set Architecture (ISA). An example of an early CISC microprocessor is the VAX-11/780 minicomputer which provided over 300 instructions, 16 addressing modes and more than 10 different instruction lengths. An example of a modern CISC microprocessor is the 80x6-based Intel Pentium. One of its complex instructions is the *MOVSB* which copies a memory byte pointed to by *DS:SI* to *ES:DI*. Then, depending on the direction flag, increments or decrements SI and DI. If *MOVSB* is qualified with *REP*, then the operation is repeated until the value of CX register is equal to zero. After setting up the required registers, *REP MOVSB* can be used to copy blocks of data including strings. The instruction *REP MOVSB* is input data dependent and terminates only when the value of *CX* reaches zero. Many CISC instructions are multi-cycle, in that they execute over more than one clock cycle. This may be because an instruction has input data dependencies (as with *REP MOVSB*) or simply because the instruction is sufficiently complex to require multiple instruction cycles.

2.1.1.1 Encoding

CISC computers are characterised by densely encoded instructions. There are three primary advantages with this. Firstly, the program image is compact making good utilisation of a limited, potentially slow and expensive main memory. Secondly, since most early CISC-based computers were programmed using assembly language, this made the programmers task much easier since the programmer's model of the processor is much closer to the ISA. Thirdly, the complexity of the instruction set reduces the semantic gap between the ISA and a high-level language. Since the assembly language instructions are designed to closely match the constructs of high-level languages it means that a compiler can be simplified; transferring complexity from software to hardware is a fundamental design philosophy of CISC. Figure 2.1 illustrates the typical encoding of a CISC ISA.

Opcode and Number of Operands	Address Specifier 1	Address Field 1	...	Address Specifier n	Address Field n

Figure 2.1: Typical CISC Instruction Encoding, adapted from Hennessy and Patterson [8, p.A-22]

The complexity of the instruction format has a direct impact on the complexity of the functional units, for example, the decoder, pipeline, execution unit and memory architecture. How-

ever, greater semantic meaning is packed into individual instructions making assembly language programming or compiler code generation simpler.

2.1.1.2 Addressing Modes

CISC instructions generally provide complex addressing modes for source and destination operands. Operands can specify register-to-register, register-to-memory and memory-to-register locations. Furthermore CISC ISAs provide helpful ways of accessing memory which satisfies the needs of the programmer and compiler. Table 2.1 is based on Silc et al.[1] and illustrates the addressing modes found on most CISC-based microprocessors. Register Transfer Language (RTL) is used to show the dataflow.

Addressing Mode	**RTL**	**Description**
Register	reg1 $\leftarrow reg2$	Simple register transfer
Immediate/literal	reg1 $\leftarrow const$	Literal is encoded in the instruction
Direct/absolute	reg1 $\leftarrow mem[const]$	Address of operand is stored in the instruction
Register indirect	reg1 $\leftarrow mem[reg2]$	Address of operand stored in reg2
Auto-increment	reg1 $\leftarrow mem[reg2++]$	Like register indirect, but reg2 is post-incremented
Auto-decrement	reg1 $\leftarrow mem[--reg2]$	Like register indirect, but reg2 is pre-decremented
Displacement	reg1 $\leftarrow mem[d + reg2]$	Typically used for arrays. d is encoded in the instruction, and would typically be the address of an array (lvalue). reg2 would contain the offset for the array element
Indexed and scaled index	reg1 $\leftarrow mem[reg2 * scale]$	Can be used for arrays and pointer dereferencing
Indirect and scaled index	reg1 $\leftarrow mem[reg2 + reg3 * scale]$	Can be used for arrays and pointer dereferencing
PC-relative	pc $\leftarrow pc + displ$	The address is an offset from the current pc. displ is encoded in the instruction

Table 2.1: Common Addressing Modes, adapted from Silc et al. [1, p.7]

2.1.1.3 Superscalar

Since a philosophy of CISC is to improve performance by adding more sophisticated hardware, many machines are superscalar (super: beyond; scalar: one dimensional). A superscalar processor executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to multiple functional units on the processor. Hence the term instruction-level parallelism (ILP) is often used to describe this feature. This form of physical parallelism increases throughput for a given clock rate. Figure 2.2 illustrates how a simplified pipeline might process instructions on a super-scalar architecture; the processor has two fetch units, two decoders and two execution units. It is therefore able to achieve physical parallelism at each pipeline stage.

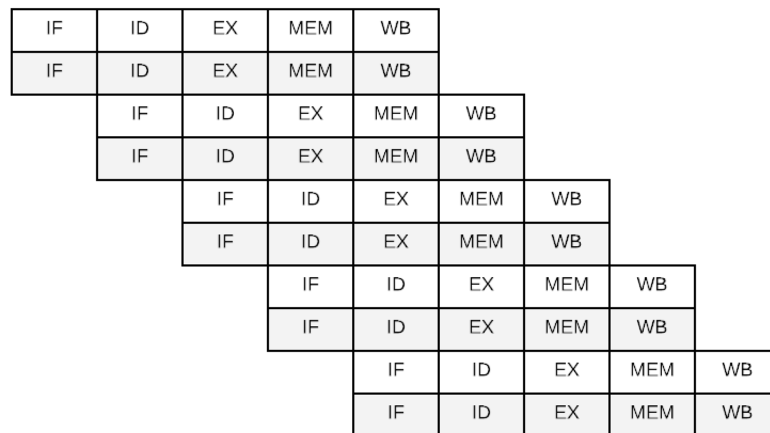

Figure 2.2: Superscalar Pipeline

2.1.2 Reduced Instruction Set Computers (RISC)

RISC can be considered a minimal design philosophy for processors. Advocates of RISC architectures claim that adding more sophisticated logic to a processor to improve non- critical instructions negatively affects other aspects of the design. This degradation in overall performance may include longer critical paths, increased propagation delays and increased power consumption. The increase in physical complexity may only be required for satisfying instructions that rarely execute in real-world programs.

RISC architecture has become the conventional microprocessor design for modern day mobile devices and embedded systems. For example, the ARM series of microprocessor is used in the

majority of mobile phones and portable computing devices. ARM, the PowerPC and MIPS are commonly used in embedded real-time systems.

2.1.2.1 Encoding

RISC instructions are typified by a simple unified encoding scheme; as such they are considered primitive when compared with CISC instructions. The motivation for simple instructions is described by Katevenis [17]. Profiling was conducted during design exploration of the Berkley RISC project. Several C and Pascal benchmark programs were used to profile the types of instructions executed and addressing modes. The studies indicated that for the benchmark programs executed, complex instructions are used much less often than simple instructions. The premise was that removing the infrequently used and unnecessary instructions meant that the processor could be made simpler and more efficient.

The unfortunate consequence is that RISC-based programs suffer from *code bloat*. This is the reason that the ARM ISA includes an additional 16-bit ISA called the ARM Thumb instruction set [12]. The differences between example CISC and RISC code can be seen in Table 2.2.

C	CISC (8086)	RISC (ARM)

a = b + c;	movl −24(%bp),%edi
	addl −28(%bp),%edi
	movl %edi,−20(%bp)
		ldr [r0],r5
		ldr [r1],r6
		add r7,r5,r6
		str [r2],r7

Table 2.2: Example C code with corresponding CISC and RISC based assembly code

2.1.2.2 Addressing Modes

RISC architectures provide primitive load/store addressing modes. For example, to increment an operand held in memory, the operand must first be loaded into a register. Then the value in the register can be incremented. Finally, the value of the register can be stored back to memory.

A restricted load/store addressing allows constant instruction cycle times. Constant instruction cycles simplify pipeline stages, instruction decoding and execution. The CISC philosophy of providing multiple addressing modes leads to a number of design complications. For example, if an opcode allows one or more operands to be addressed in memory (as opposed to only registers), this requires variable length instructions because the memory address must be specified along with the opcode itself. Figure 2.3 illustrates the typical format of a RISC instruction.

Opcode	Address Field 1	Address Field 2	Address Field 3

Figure 2.3: RISC Instruction Encoding, adapted from Hennessy and Patterson [8, p.A-22]

RISC computers have a large number of registers which complement the load/store addressing mode. Most of the registers can be used in any context; they are not tied to particular instructions. This reduces the number of memory accesses and so reduces the probability of cache misses and memory access delays. RISC computers are also likely to implement a Harvard memory model, where the instructions and data are held in separate caches. This is in contrast to CISC architectures which usually implement a Von-Neumann memory model where instructions and data share a unified cache, Hennessy and Patterson [8].

2.1.3 Reduced Operand Set Computers (ROSC)

ROSC processors are more commonly referred to as *stack-based processors*. A stack data structure is used instead of a *register set*. An example of a stack based computer is a *reverse polish* interpreter. Operands are held on top of the stack. These are popped off the stack, an operation is performed, and the result is pushed back. Stack architectures were more prevalent in the 1960s and 1970s. LaForest describes three generations of stack-based computers [20].

Modern day stack architectures are more commonly used for abstract machines, in other words, interpreters and runtime systems for intermediate code. Notable examples include the Pascal P-Code, Forth and the Java Virtual Machine. Koopman [21] describes many stack-based computers ranging from 8-bit to 32-bit machines. Notable examples of hardware stack machines include the Transputer, Pascal Micro Engine, and various Forth implementations.

2.1.3.1 General Architecture

Stack computers use one or more last-in-first-out (LIFO) stacks instead of random access registers to organise operands. The top elements of the stack are used implicitly by the ISA. Whilst this may appear restrictive, Koopman [21] argues that many register-based processors spend a significant amount of time emulating stack-based processors. This observation is probably based on how modern languages implement function calls and how compilers evaluate expressions. Koopman categorises stack architectures in three dimensions; stack size (small or large), number of operands

(0, 1 or 2) and number of stacks (single or multiple). Small on-chip stacks limit cost but may require operands to be spilled to and restored from main memory. Most practical stack machines allow 2 operands to be specified as part of the opcode. Single stack architectures are more common, though some use multiple stacks for example, separate data and return stacks, as with Coates [22]. Multiple stacks are usually employed to separate out related data which can be difficult to manipulate using a single stack. Whilst there are a number of benefits of stack-based architectures there are also a number of disadvantages; the most common and significant being the restrictive access of operands held within the stack. Stack machines been developed further in recent years, as described by Crispin-Bailey [23] and Coates [22].

2.1.3.2 Encoding

ROSC architectures are typified by a complex ISA but with extremely simple encoding. For example, the Java Virtual Machine (JVM) [24] includes type information with the opcode (otherwise known as *bytecode*) to aid verification during dynamic loading. For example *iadd* is used to add two integers whereas *fadd* is is used to add two floating point types.

There are a number of complex bytecodes which provide direct support for the Java language. These include support for language features including array management, exception handling and synchronisation. An example JVM instruction is *multinewarray*, Venners [25]. This instruction has three operands. Two bytes are combined to provide an index into a class constant pool which contains further information for creating the array. A third operand byte is used to specify the number of dimensions for the array. This is used so that the correct number of sizes for each dimension (assuming row major ordering) can be popped off the expression stack. Whilst this instruction is considered very complicated it can be represented in the same memory space as a primitive 32-bit RISC instruction (the type information is contained in the associated class files constant pool). Practical Java processors, however, follow a RISC philosophy of implementing complex instructions in the software runtime system, an approach first used with Forth.

2.1.3.3 Addressing Modes

The restrictive nature of a LIFO stack means that opcodes must have their associated operands on top of the expression stack. This is what is meant by *Reduced Operand Set Computer*. The location of the operands, the source and the destination are implied; they do not need to be explicitly stated. For example *iadd* pops two integers from the expression stack, adds them, then pushes the result.

2.1.4 Very Long Instruction Word Computers (VLIW)

An approach to improving performance in CISC designs includes superscalar execution; this requires considerably complex hardware. The objective of *VLIW* is to replace the superscalar approach to parallel processing by moving complexity away from the hardware and into the compiler. This is analogous to the RISC design philosophy of moving complexity from the processor and into the compiler. Hence, VLIW processors do not perform any dynamic scheduling or reordering of operations. All operations specified within a VLIW instruction must be independent of each other, in order that they can execute in parallel [1]. VLIW can be considered a static, compile-time approach to superscalar architecture.

2.1.4.1 Encoding and Addressing Modes

The ISA is normally based on RISC ISA and hence the encoding and addressing modes are very similar, Hennessy and Patterson [8].

Instructions that can be executed in parallel without interfering with one another are combined to form a long instruction. This *super instruction*, consisting of several instructions, can be executed in parallel by the VLIW processor. For example, if a super instruction consists of four instructions, the VLIW processor will have four execution units capable of executing all four instructions at the same time. Hence, the compiler groups independent instructions capable of being executed in parallel so as to keep multiple execution units busy. In cases where this is not possible, the compiler must insert *noops* in the large instruction word. However, this can increase code bloat found with RISC ISAs. This static, ahead-of-time encoding of parallel operations results in much simpler hardware compared to CISC and RISC designs.

VLIW can be regarded as a specialist architecture. It has become relatively popular in the area of signal and image processing where parallelism is reasonably easy to detect at compile time, Hennessy and Patterson [8]. In contrast, parallelism in general purpose applications is more difficult to detect at compile time.

2.1.5 Dataflow Computers

Dataflow Computers adopt a architecture very different from the classic Von-Neumann model of computing. To understand Dataflow Computers, it is helpful to first recap on the traditional Von-Neumann model of computing. The Von-Neumann model is based on *control flow*. “The control flow ordering is based on the idea of a temporal sequence of operations”, Sharpe [9, p.18]. The

classic Von-Neumann architecture executes the program using a variation of the fetch, decode, execute cycle. An instruction is fetched from the memory system and loaded into the *control unit*. The instruction then drives the control unit which loads operands into an internal register bank, invokes the ALU and writes register results back to the memory system. This is the fundamental model for software-based computers since The University of Manchester's Baby (circa 1948), recognised as the world's first digital computer which executed a stored program [26].

Dataflow computers directly contrast the traditional von Neumann architecture or *control flow* architecture. "A dataflow program is one in which the ordering of operations is not specified by the programmer, but that is implied by the data dependencies" Burger et al. [9, p.8]. Dataflow computers were a prominent topic of research the 1970s and early 1980s. Jack Dennis of MIT pioneered the field of static dataflow architectures, Dennis [27], Rumbaugh [28] while the Manchester Dataflow Machine, Gurd et al. [10] and MIT Tagged Token architecture were major projects in dynamic dataflow.

Conceptually, the processor does not have a program counter, register bank and single ALU. The CPU executes the computations in the order of the data interdependencies and the availability of hardware resources; it effectively executes a *dataflow* representation of the program. A dataflow CPU employs *direct instruction communication*. The processor delivers a *producer* computation output directly as an input to a *consumer* computation input, instead of using a register set to store the intermediate result.

Since program execution is determined by the data interdependencies of instructions and the availability of resources, dataflow processors employ a form of *out-of-order execution*. Out-of-order execution has become the dominant feature of superscalar processors since the 1990s in order to increase *Instructions Per Cycle*. "Modern out-of-order issue RISC and CISC designs require many inefficient and power-hungry structures, such as per-instruction register renaming, associative issue window searches, complex dynamic schedulers, high-bandwidth branch predictors, large multi-ported register files, and complex bypass networks" Burger et al. [29, p.46].

To understand conceptually how a dataflow computation executes, an example is helpful. Figure 2.4 illustrates the dataflow graph for a program taken from Sharp [9, p.55] that calculates the difference between the sum and product of two numbers. The variable a is copied using demultiplexers to the + and * nodes. Similarly, the variable b is copied to nodes + and *. The respective nodes then take the inputs, perform the appropriate calculation and output the results. These outputs become the inputs to the - node. Once the final calculation is executed, the result is output from the - node.

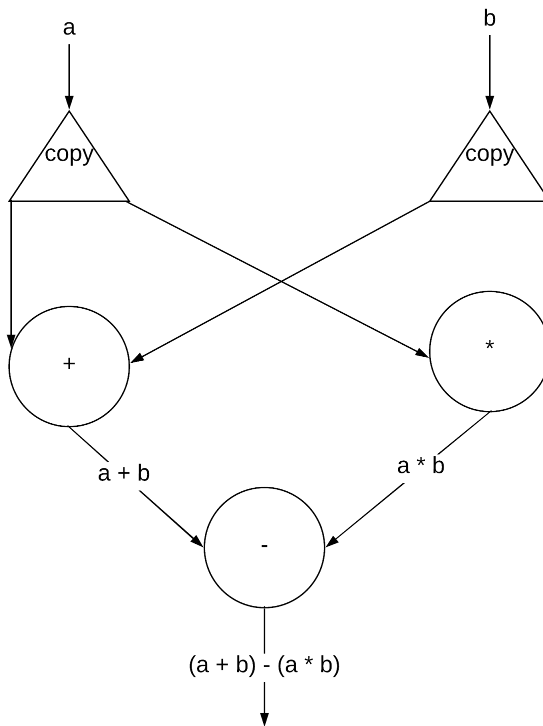

Figure 2.4: Dataflow Computation from Sharp [9, p.55]

Figures 2.5, 2.6, 2.7 and 2.8 illustrate the execution *steps*. The small circles over the arcs illustrate the intermediate results during program execution. Data flowing along arcs between nodes is represented by a *token*. Tokens can be constants, variables or control data. A node is *enabled* when all of its input tokens are available[1].

First the variables a and b are presented as *tokens* for the copy nodes as shown in Figure 2.5. The processor detects that the required input tokens to the copy nodes are available, which causes them to be *enabled*. This means that the nodes can be *fired*. In other words, the copy nodes execute and output the results as *input tokens* to the + and * nodes, shown in Figure 2.6.

The + and * nodes are *enabled* when their *input tokens* are available and therefore *fire* to produce their *tokens* to the - node (Figure 2.7). The - node detects this and becomes *enabled*, which causes it to *fire*. Its *output token* is the final result and shown in Figure 2.8.

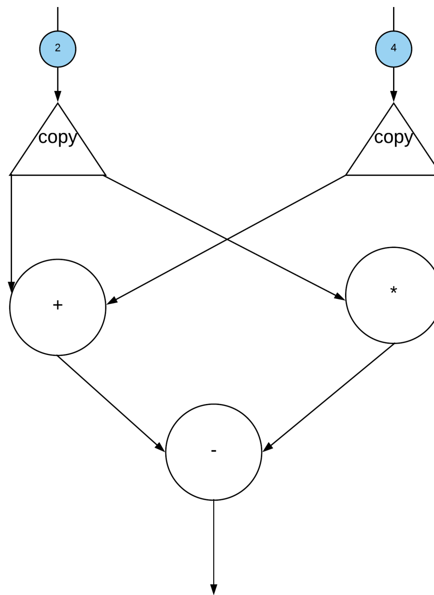

Figure 2.5: Dataflow Computation Snapshot (a) from Sharp [9, p.56]

[1] There are exceptions to this rule, for example the *merge* node. These cases are discussed by Sharp [9].

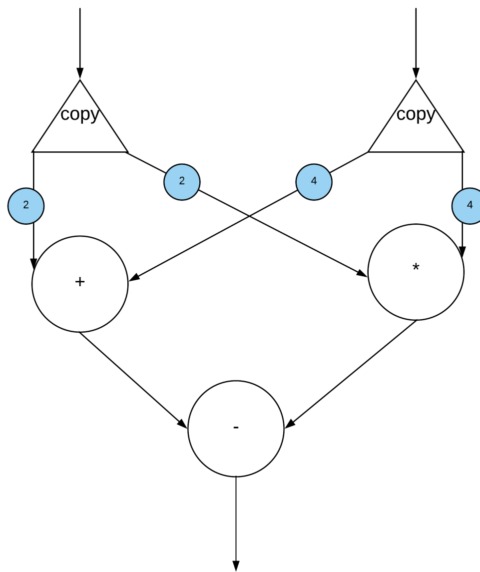

Figure 2.6: Dataflow Computation Snapshot (b) from Sharp [9, p.56]

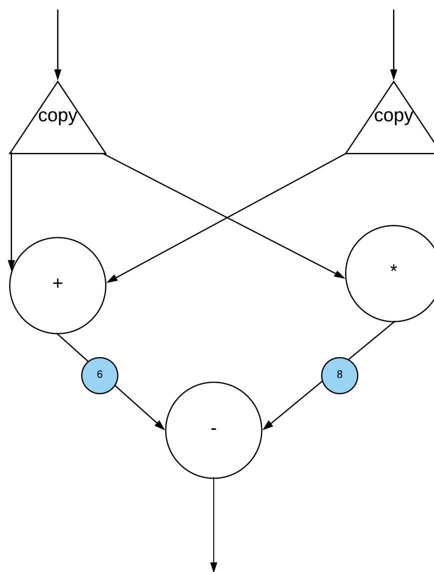

Figure 2.7: Dataflow Computation Snapshot (c) from Sharp [9, p.56]

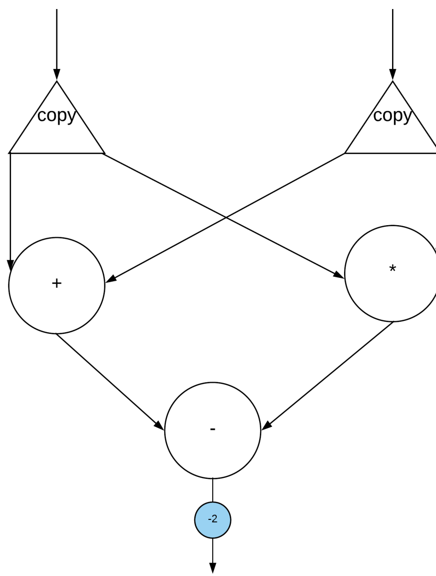

Figure 2.8: Dataflow Computation Snapshot (d) from Sharp [9, p.56]

Conditional statements can also be represented using a dataflow graph. Figure 2.9 illustrates how a *while loop* can be implemented using a *switch* and a *merge* node. The switch node places the input token on the appropriate output arc depending on the control input. The merge node places whichever input token is available on the output arc.

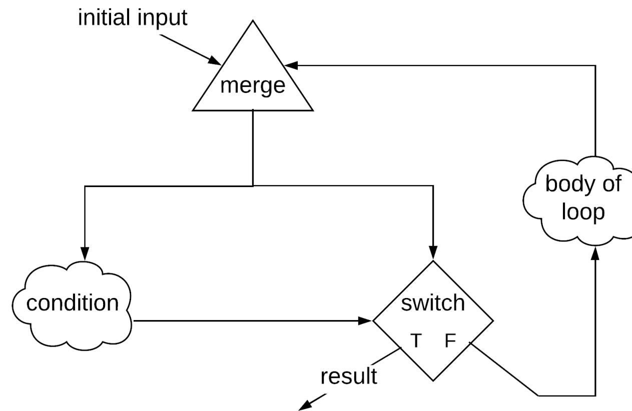

Figure 2.9: While loop Dataflow Graph based on Sharpe [9, p.32]

Early dataflow computers were programmed using graphical notation. However, there are various problems with this approach. Cyclic graphs such as the one in Figure 2.9 can suffer from *deadly embraces* and *race conditions*. These problems can be addressed by constructing *acyclic* graphs instead. Although more restrictive, a simple tree structure can be used to represent program structures whilst avoiding some of these problems.

The dataflow program illustrated in Figure 2.4 is capable of calculating the difference between the sum and product of two numbers, but not much more than this. Hard-coding the links between the node and the arcs like this is referred to as a *static* architecture and may not be very useful for executing arbitrary programs. To be flexible and practical, dataflow architectures can be designed for different types of configuration:

- Static
- Reconfigurable Static
- Runtime Dynamic

A static architecture, as shown, means that the graph structure is hardwired by the processor and cannot be changed. Whilst such an architecture may appear very restrictive, it may be possible to add flexibility by allowing each node to support multiple operations. Whilst the connections

of the arcs cannot be modified, the node operations can selected via the software instructions. A reconfigurable static architecture is one that allows the node operators and arc connections to be set during *program loading*. In other words, the graph can be configured on a per-program basis. The information to perform such configuration is provided by the compiler or other parts of the software tool-chain. A runtime dynamic configuration is the most flexible and allows the nodes and the arc connections to be configured by the instructions, as the program executes.

Figure 2.10 illustrates the processor block diagram for the Manchester Dataflow Computer taken from Gurd et al. [10, p.40]. This is referred to as a *ring* architecture where each of the hardware components sit in a ring configuration connected to the host computer system via an *I/O Switch*. The modules operate independently of one another. The modules are independently clocked and communicate asynchronously via the ring.

Tokens flow around the ring in a clockwise fashion. Tokens destined for the same instruction are paired together by the *Matching Unit*. The *Overflow Unit* is used when the limited storage of the Matching Unit is exhausted, for example if the program has a large data set. Tokens then obtain their associated instruction from the *Instruction Store*. The instruction and the input tokens are then forwarded onto the *Processing Unit* for execution. The output tokens circulate back to the Matching Unit via the *Token Queue*. This is used to support uneven token flows. The I/O Switch module allows programs and data to be loaded and results to be uploaded via the host computer.

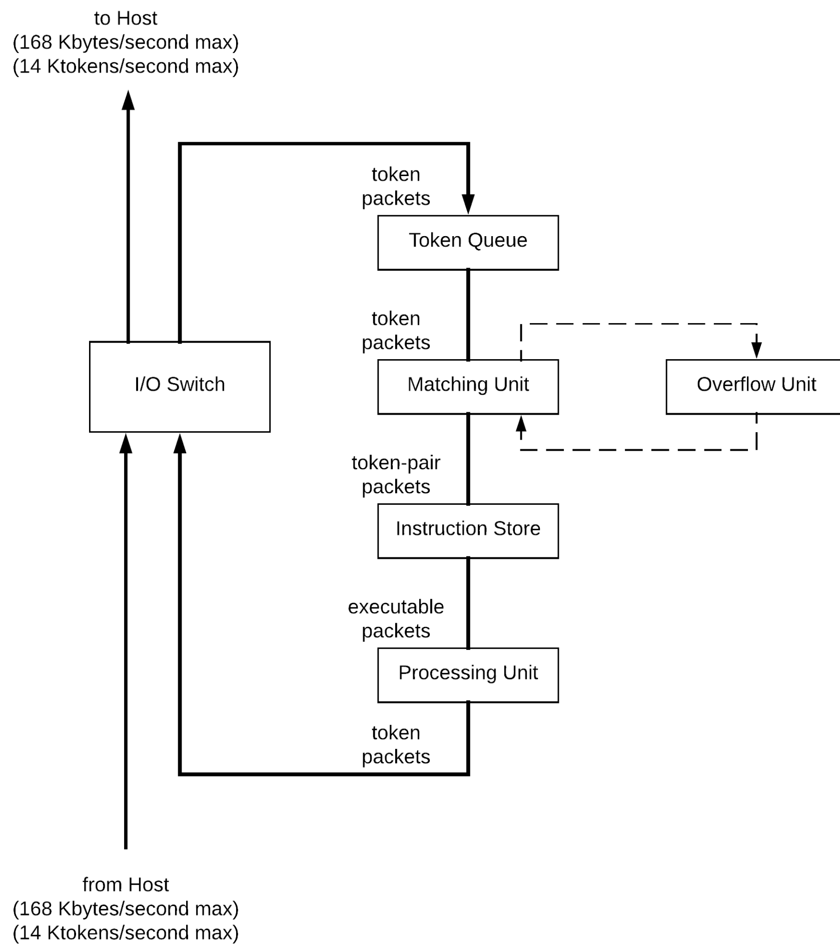

Figure 2.10: Manchester Dataflow Block Diagram, taken from Gurd et al. [10, p.40]

An example of more recent work on dataflow computing is by the University of Texas at Austin, developing the *TRIPS* processor which executes an *EDGE* ISA as described by Burger et al. [29]. *EDGE* is defined as *Explicit Data Graph Execution*. It is essentially an ISA specifically designed for dataflow processors, supporting *direct instruction communication*. The ISA directly expresses the dataflow graph generated by the compiler instead of relying on the processor to discover this information during execution. This means the hardware does not need to re-discover this information at runtime, resulting in significant savings in complexity and power.

The *TRIPS* processor [29] contains two cores. Each core consists of 16 *Execution Units* organised as a 4 * 4 matrix using a lightweight network. Each Execution Unit is capable of executing 8 instructions, therefore a total of 128 instructions can be supported. These are referred to as *hyperblocks*. Instructions with an Execution Unit are EDGE instructions and hence, they are executed as dataflow graphs. "The TRIPS microarchitecture behaves like a conventional processor with sequential semantics at the block level, with each block behaving as a *mega-instruction*. Inside the executing blocks, however, the hardware uses a fine-grained dataflow model with direct instruction communication. The processor can achieve power-efficient, out-of-order execution across an extremely large instruction window because it eliminates many of the power-hungry structures found in traditional RISC implementations", Burger et al. [29, p.47].

An evaluation of the TRIPS Computer System by Gebhart et al. [30] concludes that the performance of the TRIPS processor executing SPEC CPU200 benchmarks is outperformed by the Intel Core 2 processor. However, TRIPS did match the Pentium 4. On simple benchmarks TRIPS did outperform the Intel Core 2 by 10% and hand-optimised code outperforms it by a factor of 3. These comparisons were for cycle counts, not power consumption.

Pure dataflow programs appear to be better expressed using functional programming languages rather than imperative, control flow-based languages. These issues are discussed further by Sharp [9]. It is interesting that the TRIPS architecture uses a hybrid approach where a dataflow model is used at the intra-block level and a control-flow model is used at the inter-block level. This should make the approach more realistic for executing the vast majority of software programmed using imperative languages. However, it still employs a dataflow execution approach at the coarse-grained block level, comprising of upto 128 instructions. Having a large instruction window will undoubtedly introduce pressure on the compiler and toolchain when attempting to calculate an optimal dataflow solution. Some of the issues for compiler development are introduced by Smith et al. [31].

2.1.6 Language Specific Processors (LSPs) and High-Level Language Computer Architectures (HLLCAs)

Language Specific Processors (LSPs) and *High-Level Language Computer Architectures (HLLCAs)* are architectures with the goal of directly supporting the execution of a software programming language. The term LSP encompasses HLLCA and is the most general form of language processor. LSP can be used to describe any software-based interpreter or hardware processor which offers support for a particular programming language. Many LSPs are interpreters or virtual machines, Smith and Nair [32]. They are often implemented as stack machines executing a ROSC-based ISA as discussed in Chapter 2.1.3. Some LSPs are implemented in hardware, for example, Coates [22] defines a hardware Java Virtual Machine (JVM) for use in hard real-time systems. The Java Virtual Machine is a classic example of an LSP. Another well known LSP is the Transputer [33] designed to execute Occam.

An HLLCA may be regarded as a more specialised form of LSP. A definition is given by Chu, "a high-level language computer system is one that can accept and execute a high-level language program"[34]. In other words, a HLLCA supports the execution of a programming language at the source level or a direct representation of the language constructs.

HLLCAs were popular for the Lisp programming language in the 1970s and 1980s. They were given particular consideration for functional and logic languages (e.g. Lisp and Prolog) which had to be executed on processors designed primarily for executing imperative language-based programs.

The motives for HLLCAs include the following:

- Reduce the semantic gap between programming and machine languages;
- Simplify the compiler by reducing the semantic gap between the processor and compiler;
- Increase code density, thus reducing memory costs;
- Eliminate or drastically reduce system software;
- Increase throughput and efficiency;
- Ease debugging.

However, HLLCAs have been heavily criticised in the past, for example by Ditzel and Patterson [35].

2.1.7 Syntax Directed LSPs (SDLSPs)

A *Syntax Directed Language Specific Processor (SDLSP)* is an LSP which has an architecture defined by the grammar rules of the language itself, Audsley and Ward[11]. SDLSPs are generally

smaller than traditional general purpose CPUs and programs can be expressed in less space than a program compiled for a traditional CPU. An important aspect is the parallel evaluation of expressions, Audsley and Ward [11].

An SDLSP has the following characteristics according to Audsley and Ward [11]:

- The architecture follows the grammar rules of the language that it executes or interprets;
- Instructions are simple encodings of the source language;
- Executes instructions in a non-atomic nested manner;
- Permits independent parts of statements, constructs and expressions to be executed in parallel.

An example SDLSP is also presented by Audsley and Ward [11] which executes a language called *TINY*. TINY is a small language comprising of basic imperative language constructs, for example, *read*, *write*, *assign*, *if* and *repeat*. The type system is limited to supporting only integers and there is no support for functions or procedures. Figure 2.11 illustrates the architecture of the SDLSP. It can be seen that the architecture comprises of modules resembling the programming constructs *read*, *write*, *assign*, *if* and *repeat*.

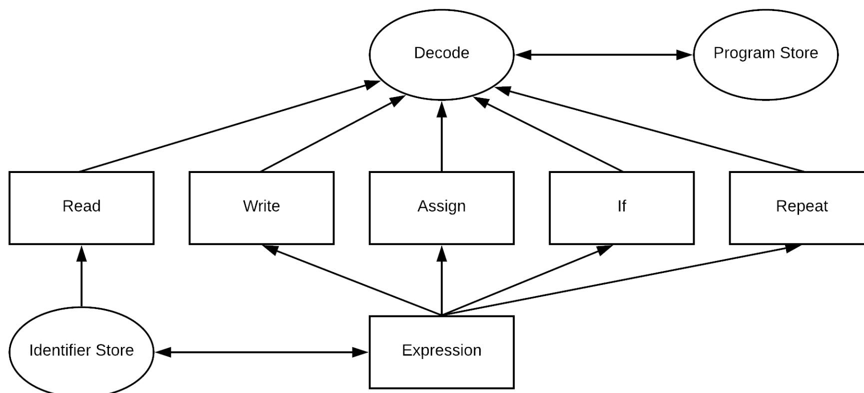

Figure 2.11: TINY Architecture, from Audsley and Ward [11]

It can be argued that the only part of an imperative language which can easily be executed using a dataflow paradigm are the individual expressions and this is the approach that the SDLSP takes. Individual TINY expressions are evaluated using an expression tree like the one illustrated in Figure 2.12. Constants (C1, C2, C3, C4) or variables (V1, V2, V3, V4) enter the leaves of the tree and flow to the root at the top of the tree. Nodes within the tree can be programmed to

use various operators. It can be seen that the structural properties of trees naturally facilitate parallelism. For example, it can be seen how all the nodes at a particular level could be executed together in a single *step* or clock cycle. By including multiple copies of operator nodes, higher levels of parallelism can be achieved.

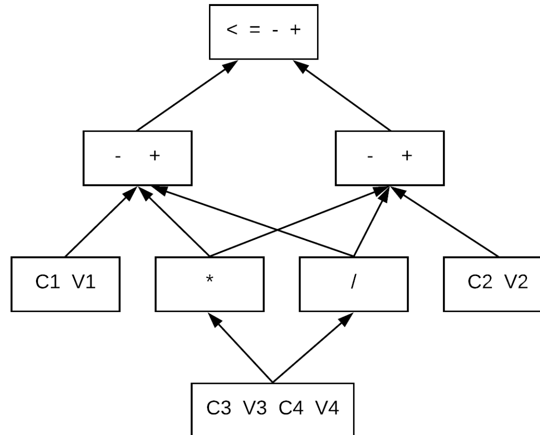

Figure 2.12: TINY Expression Tree, based on Audsley and Ward [11]

2.2 Multiprocessors

Until recently, the aim of processor designers has been to increase performance by focusing on the optimisation of the single processor. There are two fundamental ways in which this is done:

- Increasing the number of instructions that complete execution on each clock cycle; in other words by increasing throughput;
- Decreasing the clock period; in other words, increasing the clock frequency.

However, designers of multiprocessors increase processing power by increasing the number of processor cores on a silicon chip. This allows the execution of a program to be distributed over the processors, achieving physical multiprocessing.

2.2.1 Theoretical and practical Speedup

The potential performance increases offered by multiprocessors can be described using *Amdahl's Law* [36]. Amdahl's Law illustrates the potential speed up by considering the portions of a program

that can be made to run in parallel:

$$speedup = \frac{1}{(1-p) + \frac{p}{n}}$$

Where p is the portion of a program that can be executed in parallel, and n is the number of additional processors. $(1-p)$ is the portion of the program that must be executed sequentially.

2.3 Memory Hierarchy

The memory hierarchy includes all the components that the CPU interacts with when reading and writing to main memory. This includes memory controller, caches and stack memory managers. These will be discussed next.

2.3.1 Caches

Caches are used to increase the throughput of a processor by reducing the gap between the access time of data held in the processor and the main memory. As illustrated in Table 2.3, the time taken to access data within the CPU can be quantified in *pico* seconds whereas accessing data in main memory can be quantified in *nano* seconds. This difference of more than two orders of magnitude means that most processors have become reliant upon high-speed on-chip cache memory. This high-speed memory is managed by a hardware *cache controller*.

CPU	**L1 Cache**	**L2 Cache**	**Main Memory**
300 pico seconds	1 nano second	5 - 10 nano seconds	50 - 100 nano seconds

Table 2.3: Comparative access times, adapted from Hennessy and Patterson [2, p.79, Fig. A]

Caches are commonly organised within a memory hierarchy where the fastest cache is the closest to the CPU. However, since the fastest memory is the most expensive it is also the most limited in size. It is common for systems to have two caches, known as a *L1* (level 1) and *L2* cache. If data is not found in the small high-speed L1 cache, the relatively slower L2 cache is checked. If the data is not located in either, then the data is obtained from the much slower main memory. Caches are integrated with a processor such that the software is largely unaware of the presence of a cache. For some cache designs, even the operating system may be unaware of a cache.

2.3.1.1 Unified versus Split Caches

A cache can be used to store both instructions (opcodes) and data (operands). This is called a *unified cache* and follows the Von-Numan memory architecture. Alternatively, a memory system can be organised so that one cache stores instructions and a separate cache stores data. These are commonly called *I-Cache* (instruction cache) and *D-Cache* (data cache) respectively and follow the *Harvard* memory architecture.

The advantage of a unified cache is reduced gate count. Separate instruction and data caches increase the gate count since two distinct cache controllers are required. In some scenarios they can introduce memory inconsistencies. However, the use of separate caches doubles the cache bandwidth since it allows the processor to fetch instructions from the instruction cache while simultaneously reading or writing data to the data cache.

2.3.1.2 Data Structures

Figure 2.13 illustrates the data structures which form a commonly used cache known as a *four-way set associative cache*. This design can be used for instruction, data or unified caches. The cache is 4KB in total (2^{10} from bits 0-9, multiplied by 4 *ways*). Each *way* has 64 *lines* (2^6 from bits 4-9). Each cache line contains four words (2^4 from bits 0-3 divided by 4 bytes per word)[2], Sloss et al. [12].

Part of the address (bits 4 to 9) is the *set index*. This is used to map the memory address space to the cache address space. The *tag* is the remaining bits of the address and is used as a unique key for the data stored in the cache. Hence, the tag is used to decide whether a *hit* or *miss* is the result of a lookup. The *data index* is used to select a specific word in the *cache line*.

[2]Memory for holding *cache-tags* and *status* bits are not included in the size.

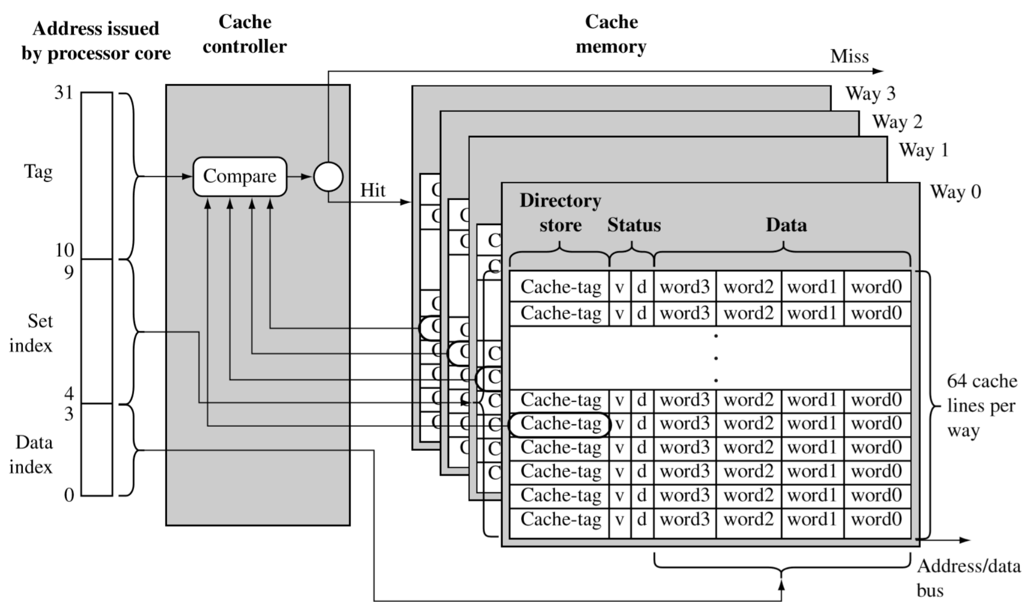

Figure 2.13: Four way set associative cache, taken from Sloss et al. [12, p.413]

A hardware cache can be viewed abstractly, similar to a software *hash table* data structure:

- array indexed by the *hash key* is synonymous with the *set*
- *way* is synonymous with the list of entries for an array element for a given hash key

A phenomenon known as thrashing can occur where data share a set index. For example, where bits 4 to 9 of the address are $xxxxxxxxxxxxxxxxxxxxxx111111xxxx$. Thrashing occurs when data in a cache location is repeatedly replaced with data for different physical memory locations. An example is in a unified one way set associative cache, with two functions called in a loop and the start address of both functions have the same set index. To prevent thrashing, a *set* can increase the number of *ways*. For example, a cache with a single way is referred to as *direct*. A cache with 2-ways is a *two way* cache, and a cache with 4-ways is a *four way* cache etc. A cache that can map any memory address to any cache line is a *fully associative* cache. Each subsequent way can be used to store additional data with the same set index. As associativity increases, the probability of thrashing decreases. However, as the associativity increases, correspondingly, so does the hardware logic of the cache controller.

2.3.1.3 Cache Tagging

Cache tagging can be performed using a number of schemes, each with various advantages and disadvantages. Popular cache organisations include the following:

- Virtual
- Virtual with Physical Tags
- Virtual with Virtual Tags and Process Identifier
- Physical

In systems incorporating a memory management unit (MMU) a *virtual cache* can be used. A virtual cache is tagged and indexed with the virtual address of the data being cached. The advantage of this design is that the virtual address of the accessed data does not need to be translated first by the MMU for every read or write operation. A virtual cache can therefore be seen as closer to the processor. The virtual address only needs converting to a physical address when a cache miss occurs. However, the translation can be done in parallel alongside cache lookup.

Whilst virtual caches are efficient by saving the address translation step, they are the most difficult type of cache to manage for an operating system. Since processes share virtual address space, problems known as *ambiguities* and *aliases* can occur. The operating system must manage

and prevent these problems. An ambiguity occurs when different data have the same tag and index in the cache. Data cached by one process could be mistaken for data belonging to another process. The consequence of an ambiguity is that the program will get the wrong data. An alias occurs when more than one virtual address is used to refer to the same physical address. This can occur if a process has the same shared memory region attached at two different virtual addresses or when two different processes use the same shared memory at different addresses in their respective address spaces.

To prevent ambiguity and alias errors in virtual caches, the operating system must ensure that data in the virtual cache is written back to main memory before another process uses it. This essentially means that a virtual cache must be *flushed* and invalidated on a context switch. This can be time consuming and the time taken is proportional to the size of the cache and number of modified lines. The new process will then miss on all its memory accesses since the cache has been completely invalidated. This means that spatial and temporal locality are under utilised.

There are alternative approaches to using a purely virtual cache which retains the spatial and temporal locality. One approach is to augment a process identifier with the virtual address tag. This effectively makes the virtual address unique. Whilst a virtual address tag can be common to multiple processes, the process identifier is unique. An alternative way of making the tag unique is to use a virtual cache, but use the physical address for the tag. The set index is still derived from the virtual address. The drawback of a virtual cache with a physical tag is that cache lookup is dependent on virtual-to-physical address translation by the MMU. Typically, the virtual address is sent to the cache and the MMU. This way the address translation and cache access are overlapped. While the MMU is translating the address, the cache begins its look-up by hashing the virtual address to obtain the index.

A physical cache uses the physical address for both the set index and the cache tag. With a physical cache the virtual address must first be translated to the physical address by the MMU. This improves the spatial and temporal locality of the cache since the cache does not need to be flushed when a context switch occurs. Physical caches are also able to provide better distribution of data throughout the cache since unique physical addresses are used for each process, rather than the same virtual addresses being used by multiple processes. The cache tag also contains state information for the cache line. This includes a *valid bit* and a *modified bit.*

2.3.1.4 Cache Read

When a program reads a variable, and assuming that the variable is held in main memory, the processor will execute a move or load instruction for the variable. The memory system will intervene and, rather than read the variable from main memory, it will first check to see if it is present in the cache. The index is obtained from the variable address, which is then used to index the set. The tag is obtained from the variable address and compared with the ways for the set. This comparison is performed in parallel rather than sequentially.

The cache tag also contains state information for the cache line. This includes a *valid bit*. When the system is booted the valid bit for all cache lines is set to 0 (which denotes false). This is so that invalid and uninitialised values are not returned from the cache. When a valid cache line is stored in the cache the valid bit is set to 1 (denoting true) meaning that the cache line contains usable data. This is termed a *hit*.

The cache controller checks the valid bit in parallel with the tag of each way. If a tag matches and the valid bit is set, then the variable has been successfully located in the cache. The line index is then used to extract the specific word in the line for the variable. If a tag match with a valid status bit cannot be found then a miss is the result. A cache miss will result in the memory system using the value from main memory instead. In this scenario the processor has to wait a number of processor cycles for the variable to be returned from main memory. The processor cycles are known as *wait states*. When the variable is returned from main memory, it is written to the cache by the cache controller and the valid bit is set to 1. This is done in parallel whist being used by the processor.

2.3.1.5 Cache Write Policy

At some point the program will write a variable by executing a move or store instruction. The cache is first searched in the same manner for a cache read. If the search results in a hit, then the data in the cache line is replaced. The way in which the data is written to the cache is known as the write policy. The write policy can either be *write back* or *write through*.

With write back policy the data is only initially written to the cache. This has the advantage of speed and simplicity in the sense that once the memory system has written the data to the cache, then the write is complete. However, the data must be written back to main memory when the cache line is evicted (see replacement policy below), or when the operating system explicitly flushes the line back to main memory. With write back, a memory system is susceptible to inconsistencies

and the task of ensuring memory coherence becomes complicated and subtle. With write through, data is written back to the cache and through to main memory. These two operations are done in parallel and the processor is able to resume execution whilst the write through is being performed. However, writing through to main memory consumes memory cycles. The advantage of write through is that the memory system is more coherent in that there are fewer memory visibility inconsistencies.

2.3.1.6 Replacement Policy

After some time the cache will fill to capacity and subsequent reads of a new variable will result in a miss. Such a cache miss is termed a *capacity miss*, because it is caused by the caches limited capacity. This will require a cache line to be evicted to make room for the new data.

The choice of which line to replace can be based on various replacement algorithms including *least recently used* (LRU). As the name suggests the least active cache line is replaced. The LRU is expensive to implement in terms of gate count. Therefore, *pseudo-LRU* is commonly used in caches with greater than four ways. Pseudo-LRU employs a binary tree to reduce the amount of state information that requires storing. An even simpler and more hardware efficient replacement policy is random. A replacement called *Most Recently Used* (MRU) does the opposite to LRU. MRU is advantageous when the oldest data in the cache is likely to be accessed more often.

2.3.2 Hardware Stack

Stack architectures use one or more *last-in-first-out* (LIFO) stacks instead of random-access registers to organise operands. If the stack overflows, elements must be spilled to main memory. If the stack underflows, stack elements must be restored from main memory. This is analogous to pushing and popping registers to and from memory during function calls on CISC-based architectures.

Koopman [21] monitored data stack spilling and restoring for benchmarks *Life, Hanoi, Frac, Math* and *Queens.* The spilling algorithm spilled one stack element each time an instruction attempted to push to a full stack and restored an element each time a pop operation was performed on an empty stack. The results showed that stack spilling and restoration tapered off at an exponential rate for these programs. As a practical matter, a stack size of 32 will eliminate stack spilling for almost all programs, Koopman [21]. According to Waldron and Harrison[37], programs including *atom, fire, jas, jjt and jcc* rarely have a data stack size of more than ten words.

2.3.2.1 Overflow and Underflow

There are a number of strategies for managing stack overflows and underflows. The following discusses some of these.

2.3.2.2 Size Stack for Worst Case Depth

An on-chip stack large enough for the worst case stack depth requirements of the program can be used. This approach completely removes the possibility of overflows and underflows. As control logic is not required for managing these scenarios, this is the simplest approach that can be taken. Sizing the stack for the worst case scenario also has a significant advantage for real time systems. Since stack spilling and restoration is input data dependent, it can be very challenging to predict at exactly which points during program execution this needs to be done. Since static analysis is not required this simplifies *Worst Case Execution Time* (WCET) analysis. Because spilling and restoration are not required, this greatly simplifies the processor design. However, static analysis is still required to determine the maximum stack depth for the program. Stack depth analysis is considered easier than spilling and restoration analysis, Koopman [21]. Given that a stack size of 32 will eliminate stack spilling for almost all programs, this may be the simplest and most cost effective strategy.

The major disadvantage to this approach is that it may not be possible to modify the size of the stack; this is only practical for FPGA-based processors (soft cores). A compromising strategy could be taken where offline analysis could be used to confirm where in a program stack spilling is likely to occur. At these points a software-based runtime library could be used to perform spilling and restoration at the appropriate boundaries. Where spilling and restoration are required, the programmer would modify the program to call the runtime library functions. Alternatively, the compiler could be modified so that the stack depth analysis and runtime support library functions are called automatically at the required program points.

2.3.2.3 Demand Fed

This approach spills and restores single elements at a time when required. A stack element is only transferred if necessary, hence minimising transfers between the on-chip stack and main memory. Very good use is made of the on-chip stack, making this approach suitable for use where chip space is at a premium. The disadvantages include the fact that complex control logic is required. The control logic is likely to be the most complex part of a stack-based processor. The points at

which spilling and restoration occur introduce temporal non-determinism and this is problematic for real- time systems. However, if the points in a program at which spilling and restoration will occur can be determined by static analysis then this can be accounted for during WCET analysis. Demand-fed single element spilling is a very common approach. A number of processors employing this approach are described in Koopman [21].

2.3.2.4 Paging

Instead of spilling and restoring single elements one at a time as and when required, a paging approach spills and restores fixed sized pages. The premise is that as soon as a single element is spilled or restored then more elements are likely to require spilling and restoring too. The work required to perform the spilling and restoration is typically done by the software runtime system. For example, the processor generates an interrupt when a stack overflows or underflows. A corresponding interrupt service routine (ISR) then performs the necessary data transfers for example, using direct memory access (DMA). This was the approach taken for *Moon2* by Vulcan Machines (company now dissolved). The transfer of stack pages is done by the software runtime system, allowing the processor hardware to remain simple. Paging requires a larger on-chip stack than the demand-fed approach, to reduce the execution frequency of the relatively slow ISR and runtime software. The cost of paging is about twice that of the demand-fed approach for memory cycles spent copying stack elements, Koopman [21]. If a program can execute within the constraints of the stack size most of the time, then paging provides an inexpensive method for graceful program degradation.

2.3.2.5 Cache

A data cache can be used by mapping it to the memory space used for the data stack. This is the usual approach used by register-based architectures. It involves complex hardware control but does not provide any advantage over alternative approaches. Caches are used by mainstream processor designers since they improve average case execution times. They are advantageous when variable length data structures such as strings and C structures are accessed.

2.3.3 Scratch Pad Memory

Scratch pad memory is a more predictable alternative to using instruction, data and unified caches in register-based architectures. Scratch pad memory is high-speed, on-chip RAM just like cache

memory. However, instead of the memory being speculatively managed by a cache controller, it is managed explicitly by software. Programs have to be modified in source or binary form to explicitly load and restore the scratch pad memory. This modification process is called *partitioning*. The process of partitioning splits the program into regions. Each region is small enough to be loaded in its entirety into the scratch pad memory. Scratch pad memory is temporally deterministic since the access time is independent of the preceding sequence of memory accesses, Whitham and Audsley [38]. This is not true for associative caches. Scratch pad memory can be managed and used within a scheduling system such as Carousel, Whitham and Audsley [39], to reduce context switching inter-task interference to zero. The cost of pre-emption is incurred by the pre-empting task rather than the pre-empted task, which removes interference from WCET calculations.

2.3.4 Multiprocessor Memory Hierarchy

The memory hierarchy of a multiprocessor system can be organised in a variety of ways, each with its own advantages and disadvantages. The most common type of multiprocessor architecture is shared memory architecture. A type of shared memory architecture is *Uniform Memory Access* (UMA), otherwise referred to as *Symmetric Multiprocessors* (SMP). Figure 2.14 illustrates a UMA/SMP system. It can be seen that each processor *(P)* typically has its own private level 1 cache *(C)*. However, all the processors access a shared main memory *(M)*. Access to the shared memory is symmetric. The terminology appears ambiguous, but simply means that all processors have equal access times to the memory, so all suffer the same latencies. Whilst the private caches may delay or prevent non-shared data traffic from competing for bus access, it can be noted that the shared bus and main memory are obvious bottlenecks. As a result, UMA/SMP-based systems are considered unscalable.

An alternative architecture for a UMA/SMP system uses a single shared cache instead of multiple private caches. Whilst this architecture may reduce the bus and memory bottleneck, a bottleneck is created at the shared cache.

Figure 2.15 illustrates a Non-uniform Memory Architecture (NUMA). This could also be termed Asymmetric Multiprocessor (AMP). A NUMA system is split into a set of segments or *nodes* where a node consists of a block of memory *(M)*, caches *(C)* and processors *(P)* which share a common bus. The nodes communicate by a distributed *interconnect*. An interconnect can be a *crossbar* or a *mesh*. Non-uniform memory access means that it will take longer to access some regions of memory than others. This is because a memory location may be on a physically different node. Communication between processors at different nodes is typically performed by *message passing*.

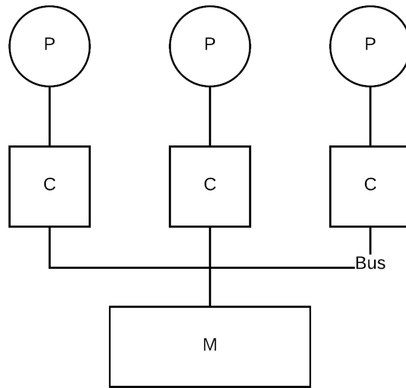

Figure 2.14: UMA/SMP Memory Hierarchy with Private Caches, adapted from Baer [13, 264]

As previously discussed, a disadvantage of UMP/SMP systems is that they are inherently unscalable. The NUMA architecture was designed to surpass the scalability limits of the SMP architecture by distributing memory. Distributing the memory among the nodes increases bandwidth and reduces the latency to local memory, Hennessy and Patterson [8]. Since the bottlenecks can be distributed throughout the node structure, scalability is improved. The main disadvantage of a NUMA system is that inter-node communication can be complex and place additional burden on the software engineering effort.

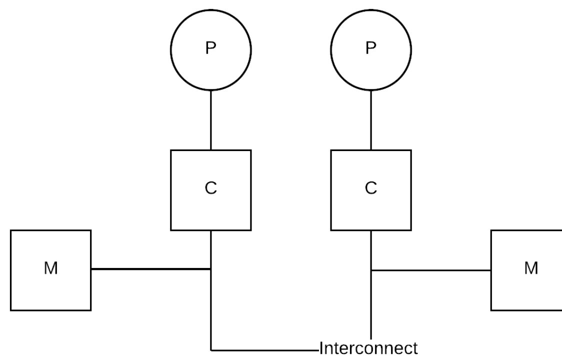

Figure 2.15: NUMA, Distributed Memory Hierarchy, adapted from Baer [13, 264]

2.3.4.1 Cache Coherency

A multiprocessor system may hold multiple copies of the same data in the caches and memory and this is the case for both UMA and NUMA systems. The shared data values must all match each other so that the processors have a coherent view of memory. This is known as *cache coherency.* The definition of cache coherency is subtle and complex, Hennessy and Patterson[8], and includes *visibility and ordering.* Informally, a memory system is coherent if any read of a data item returns the most recently written value of that data item. The two common approaches to maintaining cache coherency are *cache snooping* and *directory-based coherency.*

2.3.4.2 Snooping Based Cache Coherency

A snooping-based cache coherency protocol is used in UMA/SMP-based systems. The caches connected to the shared bus listen to messages placed on the bus by all the other caches. The cache controllers then respond in the appropriate manner to implement the coherence protocol. There are a number of common cache coherency protocols, of which a common one is the *Modified, Exclusive, Shared, Invalid (MESI) protocol.* Figures 2.16 and 2.17 illustrate the basic finite state machine for the MESI protocol, for both the local and remote cache controllers. Each cache line is tagged with the states *M, E, S, I* and the states are updated by the cache controller. It should be noted that, in practical applications, the state machine would be much more complicated due to additional architectural features such as *split transaction busses.* Figure 2.16 illustrates the behaviour of a write back cache controller in master mode, i.e. when being driven by the local processor. Figure 2.17 illustrates the behaviour of the same cache controller when acting in slave mode, i.e. when snooping and responding to remote cache requests.

At some point during execution, the processor will attempt to read a variable. Since the cache line containing the variable is not currently in the cache it is *Invalid* and a *read miss* occurs. The read miss causes the cache controller to issue a *read request* on the bus; the data may either come from memory or from another cache if one holds it. When the local cache controller places a read request on the bus, a remote cache controller snooping on the bus may recognise that it holds the data *Exclusively.* If so, the remote cache controller writes the data on the bus and sets the status of its cache line to *Shared.* The local cache controller then receives the cache line, stores it in its cache and also sets the status of the copy to *Shared.* At this point there are two caches holding the data for the same cache line and the status of both is *Shared.* In the case where the remote cache does not hold the cache line, no remote response is given. In this case the data is read from

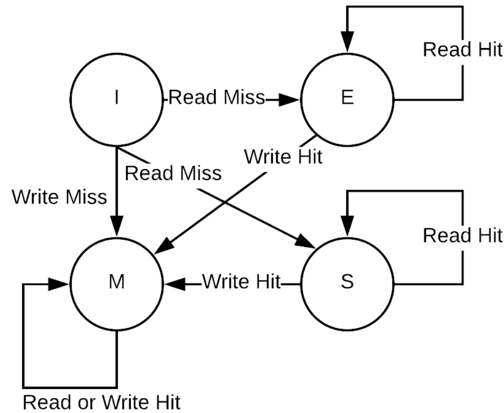

Figure 2.16: Basic MESI Cache Coherence Finite State machine - local cache controller, taken from [13, 274]

memory and the state for the local cache line is set to *Exclusive.*

If the processor writes to a variable which is stored in its local cache and a remote cache (hence both cache lines are set to *Shared*), the write results in a *write hit.* The local processor performing the write will change the state of the cache line to *Modified.* The local cache line is now the only valid cached version of the data. The remote cache controllers snooping on the bus see the write for the cache line on the bus and set the state of its version to *Invalid.* This means that the remote cache line will effectively be discarded. If the local processor writes to a cache line which is in the *Exclusive* state, the state of the locally held cache line transitions to *Modified.* However, the write is not broadcast on the bus, since there is no remote copy of it.

2.3.4.3 Directory Based Cache Coherency

Snooping-based cache coherency protocols are used for UMA/SMP-based systems. However, snooping is inappropriate for NUMA systems since the act of broadcasting on all nodes causes coupling of the busses which would reduce the improved scalability of the NUMA architecture. Effectively, the separate physical busses become a single logical bus causing a bottleneck. Since a NUMA system is inherently distributed, the cache coherency protocol also needs to be distributed. Just as main memory is physically distributed throughout the machine to improve aggregate memory bandwidth, so too is the directory. This eliminates the bottleneck that would be caused by using a single monolithic directory.

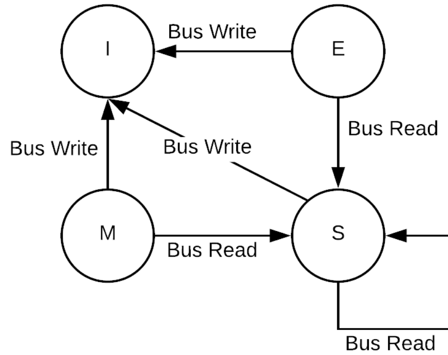

Figure 2.17: Basic MESI Cache Coherence Finite State machine - remote cache controller, adapted from [13, 274]

In a simple NUMA architecture, a node may consist of a single processor, a single cache, a single main memory and a directory. The main memory for a node is divided into cache-line sized blocks. The directory then consists of state information for each main memory block including the MESI state and the home node of the memory line.

The following describes what may happen when a read miss occurs. When a remote processor attempts to read a variable not in the cache or local memory, the cache controller must send a request to the appropriate home node (the home node is the node whose memory contains the initial value of a memory line). The home node receives the request, and may respond in a number of ways.

If the home directory indicates that that the memory line is uncached or is cached but unmodified, the memory line will be sent to the remote node. The home directory changes the state of the memory line to *Shared* and notes that a copy of the memory line is held by the remote node. If the home directory indicates that the memory line has been modified by the home cache, the data is written back from the home node cache to the home node memory. The data is then sent to the remote node.

If the home directory indicates that the memory line has been modified by a second remote cache, then the home node requests the data. When the data is received at the home node, it is written to the home node memory and forwarded to the remote node.

Other state transitions are described by Baer [13] and Hennessy and Patterson [8]. However, the above simplification demonstrates the complexity inherent in a distributed cache directory.

2.4 Summary

CISC architectures have evolved since the 1970s. They have a compact ISA; there is a high semantic mapping between CISC instructions and C programming constructs. For this reason executables tend to be very compact. A driver for CISC is to encapsulate complexity at the processor level. CISC architectures often support advanced features to improve ILP for example superscalar execution units.

The philosophy of RISC is to have simpler processors; this can be observed by the primitive *load-store* ISA. The premise is that removing the complex and infrequently used instructions means that the processor can be made more efficient. However, a portion of this complexity is transferred to the compiler and software runtime system.

ROSC architectures, otherwise known as *stack-based* architectures employ one or more on chip LIFO stacks instead of a random access register set. Stack machines tend to have simpler hardware and attributes advantageous for embedded real-time systems, Koopman [21]. Stack machines have been implemented in silicon and as abstract machines, for example p-code [40], Forth [19] and the Java Virtual Machine (JVM) [24]. Table 2.4 illustrates the main differences between CISC, RISC and ROSC architectures.

CISC	**RISC**	**ROSC**
Sophisticated instructions	Primitive instructions	Sophisticated instructions
Deep pipelines	Deep pipelines	Shallow pipelines
Compact code	Verbose code	Compact code
ISA close to programmer's model	ISA far from programmer's model	ISA far from programmer's model
Easy to program using ASM	More difficult to program using ASM	Even more difficult to program using ASM
Multi-cycle instructions	Mostly single-cycle instructions	Multi-cycle instructions
Specialised resisters	Many registers general purpose	Restrictive on-chip stack

Table 2.4: Summary of differences between CISC, RISC and ROSC

Increasing ILP and clock frequency is becoming very challenging for processor designers. This is mainly due to maturing designs, unacceptable power consumption and heat dissipation. It has been suggested that improving performance can be achieved by employing multiprocessors for explicit parallelism. Amdahl's Law illustrates the speedup potential of a given program [36]. However, the practicalities of parallel computation mean that Amdahl's Law is optimistic.

Dataflow architectures are used to increase the granularity of parallelism to the instruction level. A dataflow program is one in which the ordering of operations is implied by the data dependencies rather than explicit control flow. A dataflow architecture employs *direct instruction communication*. Programs can be expressed using a graphical notation or more simply using trees. The Manchester Dataflow Computer and the more recent TRIPS processor have been introduced.

HLLCAs directly support the execution of a high level software programming languages. HLL-CAs can be regarded a subset of LSPs, since they are usually designed to execute a *specific* high-level language. However, HLLCAs have been heavily criticised by Ditzel and Patterson [35].

LSPs may be regarded as a more general form of HLLCA. LSPs usually refer to a software interpreter and runtime system or JIT compiler targeted at a specific language. Many LSPss are interpreters or virtual machines, Smith and Nair [32]. They are often implemented as stack machines.

A SDLSP is an LSP and has an architecture which follows the grammar rules of the language. The instructions are simple encodings of the source language.

The memory hierarchy includes main memory, caches and stack management components. Caches are used to reduce the adverse effects of disparity in speed between CPU and main memory by taking advantage of spatial and temporal locality, Hennessy and Patterson [8]. If the ways of a set are full for a given address, then an appropriate cache line must be evicted to make space; the cache line to evict depends on the replacement policy. Cache modelling can be used to predict cache behaviour, though it is generally regarded as NP-hard. An alternative to speculative caches is scratch pad memory and this may also help reduce non-determinism. The memory architecture of a multiprocessor system can be organised in a variety of ways including UMA/SMP and NUMA (distributed memory). Multiple copies of the same data may be held in multiple caches. So that all processors see the same value, cache coherency protocols must be implemented by the caches. Cache coherency for UMA/SMP systems uses *snooping* whereas cache coherency for NUMA uses *distributed directory-based* approaches, Hennessy and Patterson [8]. Practical implementations tend to be very complicated. Stack spilling and restoration is an aspect of memory hierarchy for stack-based processors. A number of approaches can be used for managing this, including sizing the stack for the worst case depth, demand-fed, paging, caching, function stack, block stack and scratch pad memory, Koopman [21].

The fundamental ISA and architecture of a processor and surrounding subsystems dictate to some extent the basic performance ultimately achievable. In order to optimise performance, designers employ techniques to improve the common case, for example, pipelining, super-scaler

and caching.

Microprocessor vendors are naturally reluctant to change a processor ISA for commercial reasons; doing so could undermine the entire product base. Overhauling or even simply modifying it would have massive repercussions for the processor's eco-system. Modifying the ISA would impact the compiler back-end, including machine specific optimisers and code generators. Other toolchain components such as linkers, loaders, romisers, assemblers, debugger back-ends, profilers, memory checking tools would all be significantly impacted. As a result, vendors may have been hesitant in addressing the problems and improving ISAs.

Chapter 3

Problem Analysis

This chapter outlines the problems that the thesis aims to address. Firstly, the motives for multiprocessor systems are discussed. The motives are driven by the current problems faced by uniprocessor systems. The challenges faced by multiprocessor systems are then presented. In order to address the challenges it is necessary to understand where improvements can be made in CPU design. These improvements are focused on considering fundamental changes rather than optimising the *status quo*. In order to understand where fundamental improvements can be made, it is first necessary to understand what CPUs spend most of their time and energy doing. This understanding can be gained by reviewing previous literature and using benchmarks to project energy estimates on the results. After reviewing the initial findings, this chapter finishes by suggesting a possible alternative CPU ISA to address help the challenges.

3.1 The Motivation for Multiprocessor Systems

In 1965, Gordon Moore observed that the number of transistors that could be integrated on silicon chips was doubling about every two years. This trend has become known as Moore's Law [16]. Figure 3.1 illustrates the transistor counts and other design variables for microprocessors from 1970 to 2020 (projected).

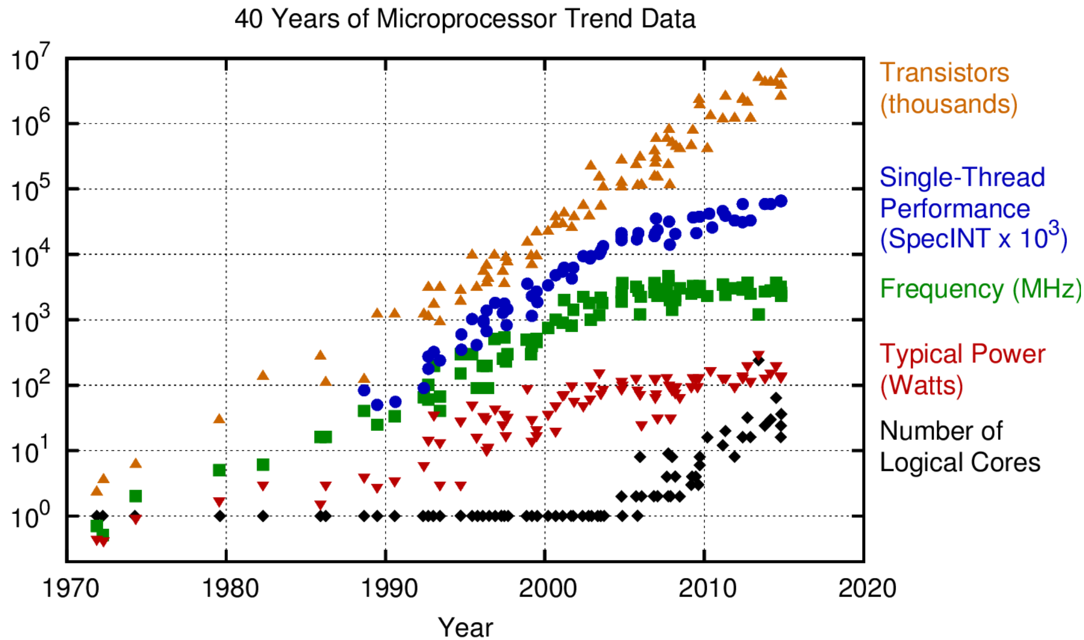

Figure 3.1: 40 Years of Microprocessor Trend Data, taken from National Research Council [14, p.55]. Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten Dotted line extrapolations by C. Moore.

Coinciding with Moore's Law is the exponential increase in the performance of microprocessors. Until 2004 there was a rise in the switching speed of transistors. Clock frequencies have increased by three orders of magnitude since 1971, McCool et al. [41]. The improvement of instruction level parallelism (ILP) and the increase in transistor switching capability supported the increase in clock speeds. It can be seen from Figure 3.1 that clock frequencies and, correspondingly, power consumption have flat-lined around 2005. Transistor counts have increased by about six orders of magnitude since the early 1970s and the trend is continuing to do so. This is because designers are increasing the number of cores on a silicon chip in order to compensate with the flat-lining of clock frequencies. Performance can no longer be improved by increasing clock speeds. It can only be improved by adding more processor cores to a silicon chip. This approach is taken as there are seemingly no alternatives.

There are three converging factors limiting the growth in performance of single-core processors. These are known as the 3 Walls, McCool et al. [41]:

- Power Wall
- ILP Wall

- Memory Wall

Power consumption and dissipation has become a limiting constraint for increasing clock speeds. The power consumption of a processor is proportional to $clockspeed * supplyvoltage^2$. Processor power consumption is exceeding the few hundred watts that can be dissipated in a practical computer system, National Research Council [14]. This is why modern desktop microprocessors require such large heat-sinks. The power wall exists because power consumption and dissipation increases non-linearly with clock frequency. Dennard [42] originally observed that voltage and current should be proportional to the linear dimensions of a transistor. Therefore, as transistors shrink, so should the necessary voltage and current. The supposition was that power is proportional to the area of the transistor. However, Dennard did not consider the *leakage current* and *threshold voltage*, which establish a baseline of power per transistor. As transistors get smaller, power density increases because these do not scale with size. This has resulted in the power wall that has limited practical processor frequency to around 4-5 GHz since about 2005.

Instruction Level Parallelism is reaching limits for single-core processors, for example pipelining has hit a practical limit at around 20 stages. The number of instructions which can be executed in parallel via superscalar approaches has peaked at four instructions per clock cycle, Olukotun et al. [43]. This is because the logic required to identify the opportunity for parallel instructions is proportional to the square of the number of instructions that can be issued simultaneously; it is quadratic.

There is still a disparity between processor speed and memory access times. There are several reasons for this, but a significant issue is power consumption and heat dissipation, McCool et al. [41]. It is because of this disparity that microprocessors are so heavily reliant upon large caches.

In order for computing power to increase it appears that an alternative approach to increasing clock frequencies and ILP of single core processors is required. An obvious alternative is to increase computing power by increasing explicit parallelism via multiprocessors. Multiprocessors consist of two or more fully functioning processors on a single piece of silicon. This organisation has the effect of distributing computation and, therefore, power and heat dissipation across a much larger area of silicon rather than concentrating these to the area of a single processor. Typically, the processors are coordinated and synchronised by a single operating system to permit hardware concurrency.

3.2 The Challenges for Multiprocessor Systems

Although processor designers are relying on multiprocessors to continue the performance improvement trends of previous decades, there are many challenges to overcome. Some of these problems are fundamental computational problems and are unsolvable. Many of the issues are also pertinent to uniprocessor design. The most significant challenge for multiprocessor systems is the inherent computational complexity of parallel programs.

3.2.1 Computational Complexity

As discussed in Chapter 2, the potential performance improvements offered by multiprocessors can be described using Amdahl's Law [36] which considers the portions of a program that can be made to run in parallel, as:

$$speedup = \frac{1}{(1-p) + \frac{p}{n}}$$

The graph in Figure 3.2 shows potential speedup according to Amdahl Law given the number of processors and the percentage of a program that can be made to run in parallel.

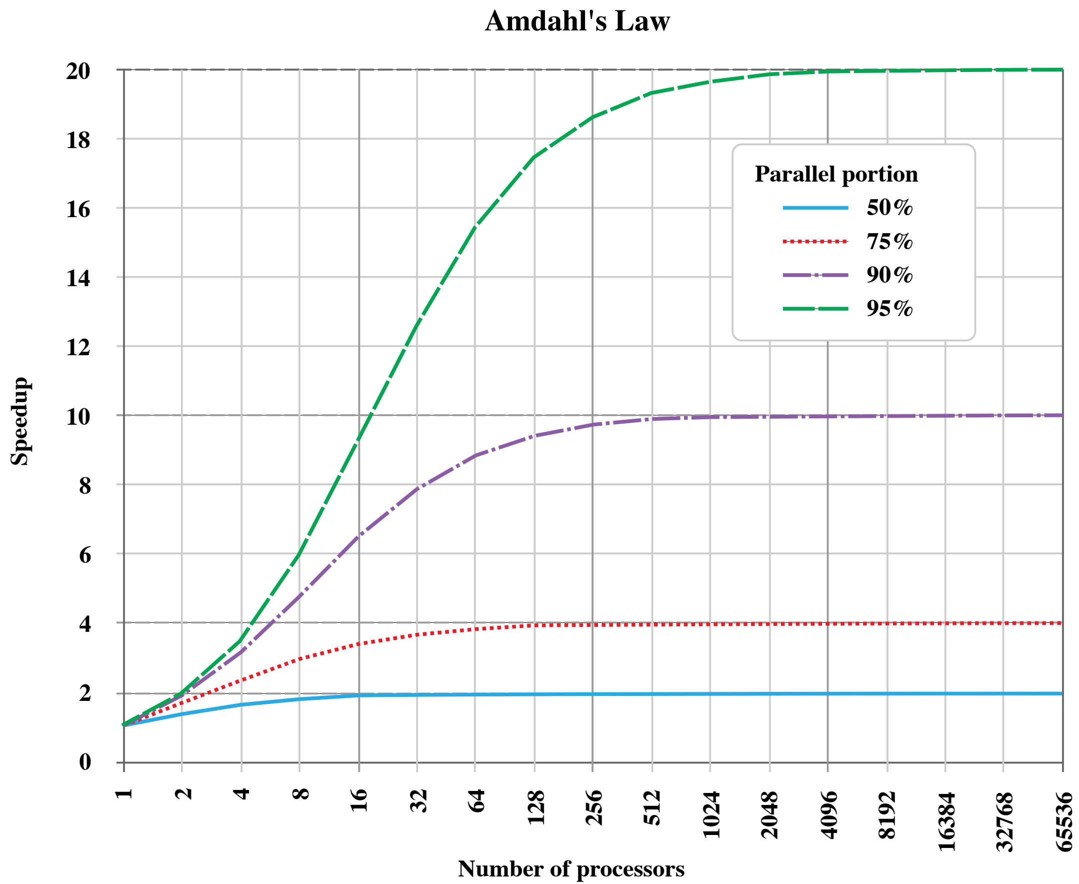

Figure 3.2: Speed Up Possibilities for 50%, 75%, 90% and 95% parallelism, taken from [15]

As an example, Amdahl's Law states that, if 75% of a program can be executed in parallel with 128 processors, the program will execute only four times faster than it would with a single microprocessor. Increasing the number of processors beyond this point yields no benefit, in fact, performance may degrade. Whilst Amdahl's Law may appear sobering, it must be stressed that it gives optimistic speedup factors. Amdahl's Law does not account for implementation overheads and complexities, for example communication overhead. The maximum number of full duplex paths of communication between a set of n microprocessors is $n * (n - 1)/2$ which is obviously $n * (n - 1)$ half duplex paths. If a system has 100 processors, this means there are potentially 10,000 half duplex paths of communication. This is $O(N^2)$, which is quadratic. The ways in which the processor cores can communicate and share data is unbounded. The challenge is structuring the communication without causing bottlenecks. Multiprocessor communication is inherently a computational problem rather than a technical one. The challenges that they bring may outweigh the advantages in some cases.

3.2.2 Multiprocessor Scaling

A study into dark silicon by Esmaeilzadeh et al. [44] suggests that, regardless of chip organisations and topology, multicore scaling is power limited to a degree not widely appreciated by the computing community. "Even at 22 nm, 21% of a fixed-size chip must be powered off, and at 8nm, this number grows to more than 50%. Through 2024, only a 7.9x average speedup is possible across commonly used parallel workloads, leaving a nearly 24-fold gap from a target of doubled performance per generation"[44, p.1].

3.3 Understanding what CPUs Spend Most Time Doing

In 1980, the RISC project was started at Berkley, with the goal of investigating an alternative to the trend of increasing architecture complexity. Katevenis [17] received the ACM Doctoral Dissertation Award in 1983 for his thesis documenting the rationale for the RISC approach. The choice of the instruction set was a key to this philosophy. Firstly, the most necessary and frequent operations in programs were identified. Then, the data path and timing required for their execution were found. Finally, the other frequent operations which could also fit into that data path and timing were also included in the instruction set. During the definition of the RISC architecture, its implementation was kept in mind at all times. Katevenis goes on to say "It is very difficult for such a study to be made abstractly - not in connection with a particular model of computers and

computations, because real programs and programming languages are written and defined with a particular model in mind"[17, p.10]. The properties of benchmark programs are then discussed, in particular and of specific interest, operations (instructions) are described which are to be counted. These include *test, compare, add, subtract, multiply, divide* and so on. Execution sequencing is considered to determine the control and pipeline organisation. It appears that the general ISA and CPU design may have been already decided prior to the benchmarking analysis. By prematurely assuming a CPU model based on traditional techniques at the outset, the opportunities for new and different architectures may have been somewhat restricted.

Koopmans [21] viewpoint is that stack machines are much simpler than CISC and RISC machines. "They do this without requiring complicated compilers or cache control hardware for good performance"[21, p.15]. Koopman goes on to argue that RISC processors require huge caches and deep pipelines because of their verbose instruction encoding. Stack-based computers require smaller caches or can better utilise caches because of the density of the ROSC ISA. Stack-based processors require shallower pipelines than RISC machines and are better suited to interrupt processing, Koopman [21]. It is argued that RISC computers spend much of their processing effort emulating stack machines.

Verma et al. demonstrated that 50-70% of the total power for a system is consumed by the memory system [3]. Further studies show that the memory subsystem consumes 65.2% and 45.9% of the total energy budget for uni-processor ARM and multi-processor ARM systems respectively [3]. Thumb instructions resulting in high code density leads to around 30% reduction in energy by instruction fetches, Verma et al. [3]. This indicates that the choice of ISA is important when considering instruction memory usage.

Before becoming completely reliant on multiprocessors for improving performance of microprocessors, it would be prudent to ensure that the performance in uni-processors cannot be significantly improved upon. Many of the approaches to processor design are decades old and the general consensus from the literature is that these approaches have reached an optimum design or *sweet spot* and cannot be improved. However, previously discounted or overlooked approaches could yield benefits to the current problems faced by processor designers.

3.3.1 Benchmarks

The premise from previous research is that CPUs spend significant time accessing the memory system and that this is a significant problem. In order to reduce the number of memory accesses, it is necessary to understand the interaction between the microprocessor core, the cache and external memory. To do this, a set of benchmarks called *mibench* (Guthaus et al. [45]) were compiled with *GCC* and executed under the *gem5* simulator (Binkert et al. [46]). Optimisation options were omitted from the *GCC* command line. The simulation is of an ARMv7 Cortex A15 uniprocessor, with an clock frequency of 1GHz. The following statistics were obtained for the simulations:

- Total number of instructions simulated
- Frequency of load and store instructions
- Number of hits for instruction and data caches
- Number of misses for instruction and data caches
- Number of write-backs for the data cache

From Figure 3.3 it can be observed that the number of *load* and *stores* are significant. For example, the proportion of combined load and store instructions for *crc32_large* is 50% of the total instructions simulated. Drilling down further, the raw data in Table 3.1 shows *bitcnt_large* has high hit rates. *adpcm_large* has a relatively high miss rate for both data and instruction caches resulting in a higher number of memory accesses. Some benchmarks show that the data cache misses are more prominent than instruction cache misses, for example *crc32_large*. Others show that the instruction cache misses are more prominent, for example, *patricia_large*.

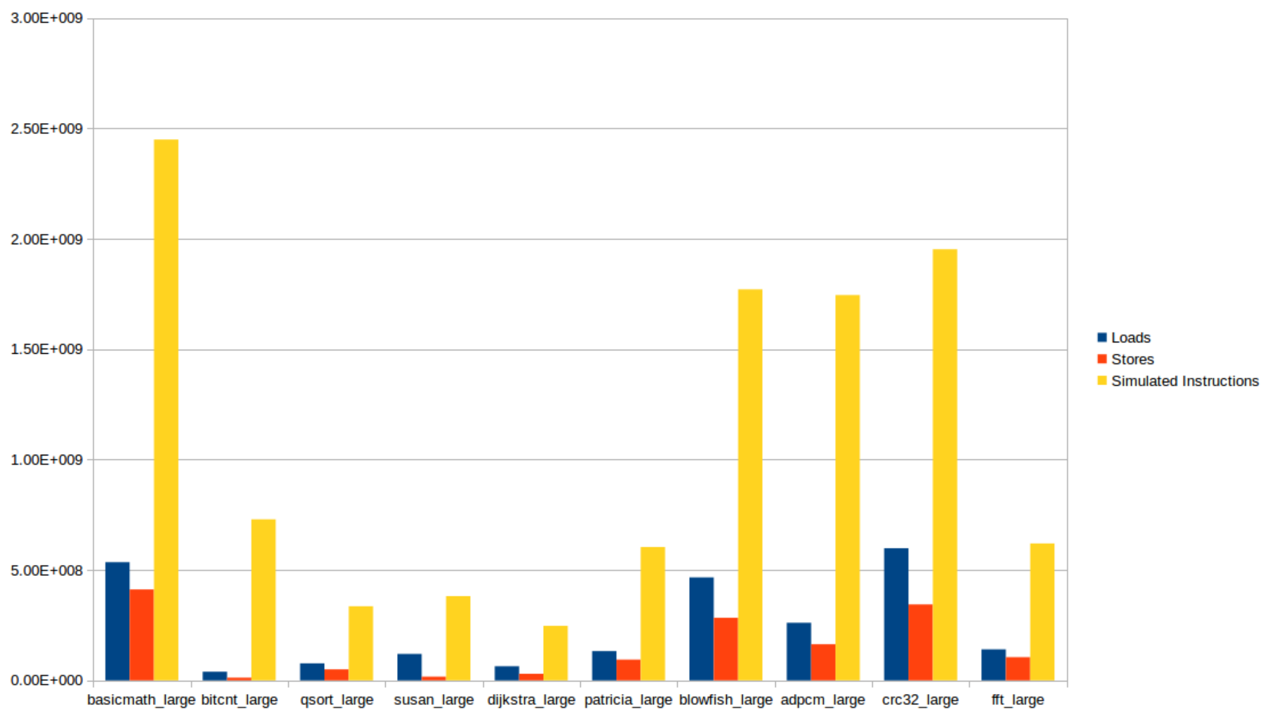

Figure 3.3: Simulation Results showing number of Loads, Stores and Simulated Instructions

	Loads	Stores	Simulated instructions	I-cache hits	D-cache hits	I-cache misses	D-cache misses	D-cache write-backs
basicmath_large	535188128	412105579	2449405763	2871418107	907534341	9353491	1328152	604493
bitcnt_large	39417048	12144952	729145905	816684325	51380277	22530	28321	16367
qsort_large	77339374	50027846	335787280	408007439	123456082	513188	620033	326071
susan_large	119814925	16890897	381480085	504204148	135843952	69473	149881	83501
dijkstra_large	64069400	29871172	247199700	260424786	92731781	134226	235342	51160
patricia_large	132885973	93821463	603702121	696924044	218545446	5143457	405701	200517
blowfish_large	466199227	283583550	1771055668	2049092545	662705274	2196801	596405	267059
adpcm_large	260785097	163961467	1745071316	1919245178	384103860	7538463	2653518	1160273
crc32_large	598364442	343895250	1952702221	2406926939	819334515	638850	1023756	525044
fft_large	140516556	105017578	620108528	736816376	238150755	1443497	493716	398843

Table 3.1: Raw Simulation Results

3.3.2 Energy and Power Measurements

Energy is defined as the capacity for doing work and in electronics is measured in units of *joules*. One joule is the energy needed to move one ampere through one ohm of resistance for one second. Power is measured in *watts* and is the amount of energy, or joules for a given period of time, one second. Therefore, one joule is the the equivalent of one *watt* of power dissipated for one second.

The energy consumed for the execution of an instruction is the integral of the power dissipated over the time interval T, Verma et al. [3]. This can be described more formally by:

$$E = \int_0^T P(t)\, dt = \int_0^T V \ * I(t)\, dt$$

Each instruction requires a specific interval of time to complete and the the energy required to perform an operation decreases if the time T decreases and/or the power dissipation *P(T)* decreases. Energy for a given operation can be obtained by measuring the average current drawn by a processor and memory system providing that the measured current does not show a high variance over the the time interval T, Verma et al. [3].

The Micron DDR2 technical paper [47] provides a model for estimating the power consumption (note, not energy consumption) for a DDR2 memory system under various system usage conditions. However, this is a complex process involving a number of variables. The following summarises the main variables used when estimating power consumption. Background power includes pre-charge and activate states. Pre-charge includes power down PRE_PDN and pre-charge standby PRE_STBY. Pre-charge power down is the lowest power state in which the device can operate. Activate power includes active power down ACT_PDN and activate standby ACT_STBY. These power states are used to select a bank and row for accessing the DDR2. The above variables can be expressed for clock-high and clock-low states; hence there are a total of eight variables for calculating background power. Power for each of these is the product of the average device current and the input voltage. However, the DDR2 does not spend all of its operating time in all of the above states. Ratios need to be applied to each of the variables; hence it is necessary to estimate the proportion of time that the DDR2 spends in each of the above states. This is also dependent upon the clock period (hence DDR2 clock frequency). Write power *WR* is obtained by subtracting the activation background current (calculated above) from the write current. This is then multiplied by the input voltage. This power consumption value must then be multiplied by the ratio of the write bandwidth. The read *RD* power is calculated in a similar manner. The I/O and termination power must be calculated and includes the following: output driver power when driving the bus

(DQ); power when terminating a write to the DRAM $termW$; power when terminating a read from another DRAM $termR$; power when terminating a write to another DRAM. The final variable is the refresh REF power. DDR memory cells store data in small capacitors that require refreshing. Calculating the power consumption for DDR2 reads and writes is a complex and tedious statistical calculation and is dependent upon the device, input voltage and bandwidth estimates. Accurately determining the energy for a single read or write is almost impossible because these operations cannot be expressed in terms of a single isolated activity. Many variables must be considered including various background quantities. Furthermore, attempting to isolate the dynamic energy for a single operation is counter productive, since operations are often performed in bursts. From an architectural aspect, isolating the energy consumption for a single instruction is desired. However, from an electronics aspect, average power for a given bandwidth is required. These are opposing and conflicting requirements.

There has been limited research investigating energy consumption at the ISA level. However, Verma et al. [3] employ a model originally used by Tiwari et al. [48] to measure instruction-level energy (static and dynamic) for an ARM7TDMI Atmel evaluation board AT91EB01. Physical measurements of the average current drawn by the processor and memory system on the ARM evaluation board were obtained. The base energy cost is the energy consumed by an instruction when it is executed in isolation. To obtain this, a given instruction is executed multiple times and the average current drawn is calculated. The inter-instruction cost is the energy consumed when the processor switches from one instruction to another. The research provides values for load and store instructions where the opcode and operands are in combinations of on-chip scratch pad memory and external main. The equation $E \approx V * I_{avg} * T$ is used to calculate the energy E for the ARM uniprocessor model. This approach is valid provided that the measured current does not show a high degree of variance over the time T and that the voltage is kept constant, Verma et al. [3]. Table 3.2 summarises the pertinent results for reading and writing four bytes to external main memory and on-chip scratchpad memory.

The benchmark simulations provide frequencies for interactions between the microprocessor, the cache and external memory. However, in their current form they do not provide any indication for *energy* consumption. The frequencies for hit, miss and write-backs can be scaled in order to indicate energy based on the work from Verma et al. [3].

It can be assumed that a hit consumes the least energy and so is simply multiplied by $1.2nJ$ observed by [3] when accessing four bytes of scratchpad memory. Cache misses are multiplied by $49.3nJ$ observed by [3] when reading four bytes from main memory. Write-backs are multiplied by

41.1 nJ observed by [3] when writing four bytes to main memory. Figures 3.4 to 3.13 labelled A illustrate the results using the energy results from [3].

Memory	Energy per access (nJ)
Main Memory Read	49.3
Main Memory Write	41.1
Scratchpad Read	1.2
Scratchpad Memory Write	1.2

Table 3.2: Verma et. al. Energy Access Results, taken from [3, p.22]

In order to provide worst case values for energy costs, approximations of the *time* differences between accessing L1 cache and main memory is applied to the data using the comparative access times for registers, L1 cache, L2 cache and main memory in Figure 2.3. 1ns is used for a hit and 100ns is used for misses and write-backs. This is using the upper bound for the main memory access time rule of thumb. Figures 3.4 to 3.13 labelled B illustrate the results. Even though these results are scaled using *time* rather than *energy*, there is a relationship between the two. Figures 3.4 to 3.13 labelled C show very pessimistic projections for energy costs. These use a factor of 1,000 for cache misses and writebacks.

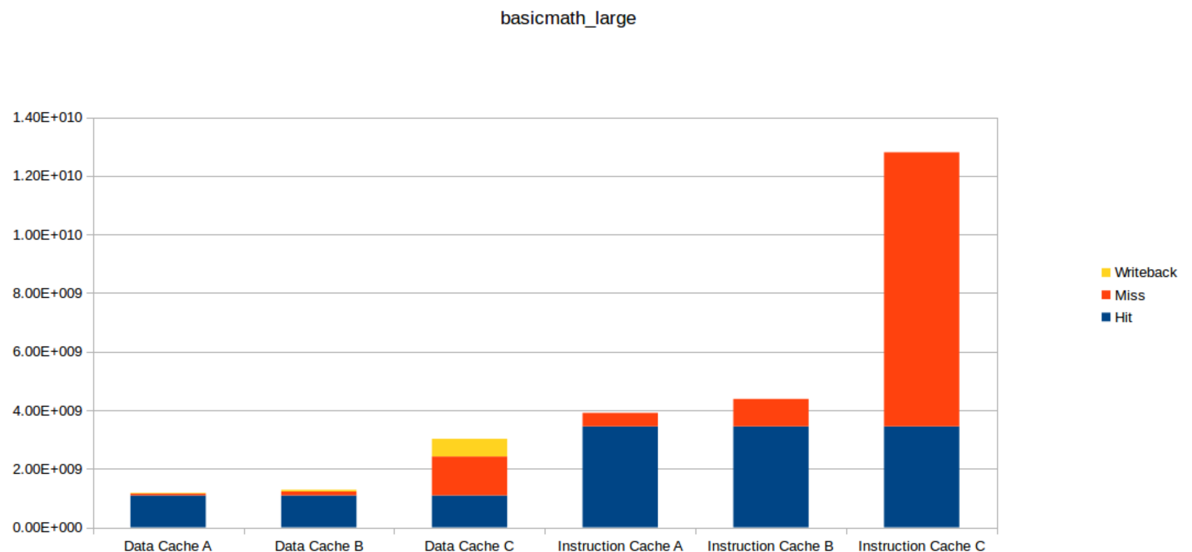

Figure 3.4: basicmath large Power Projections showing number of Writebacks, Misses and Hits

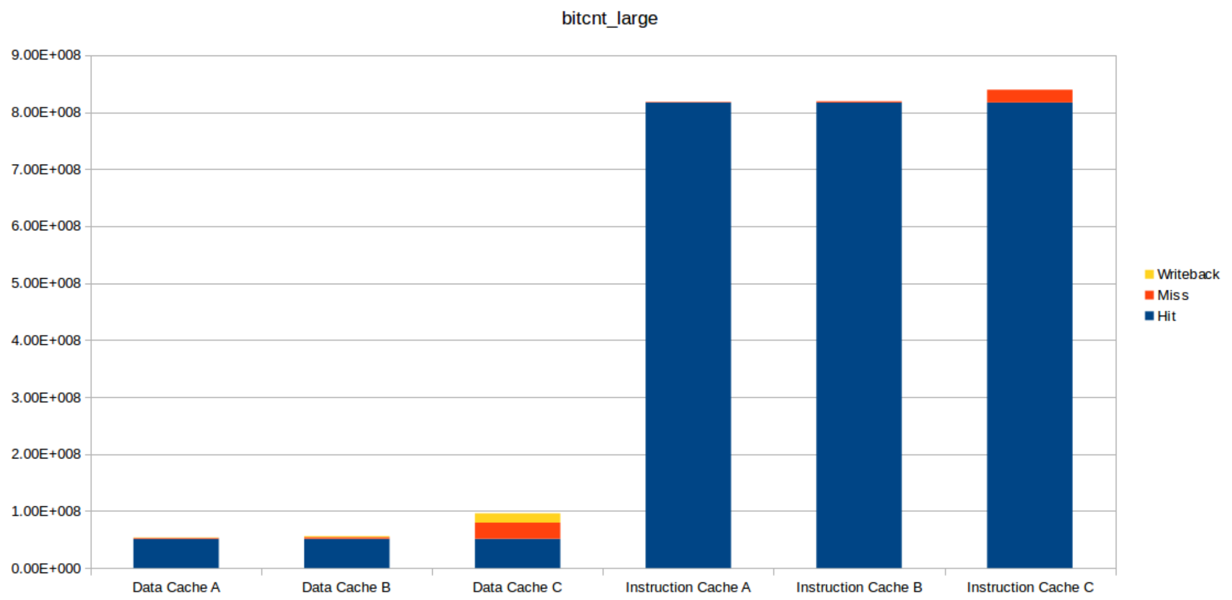

Figure 3.5: bitcnt large Power Projections showing number of Writebacks, Misses and Hits

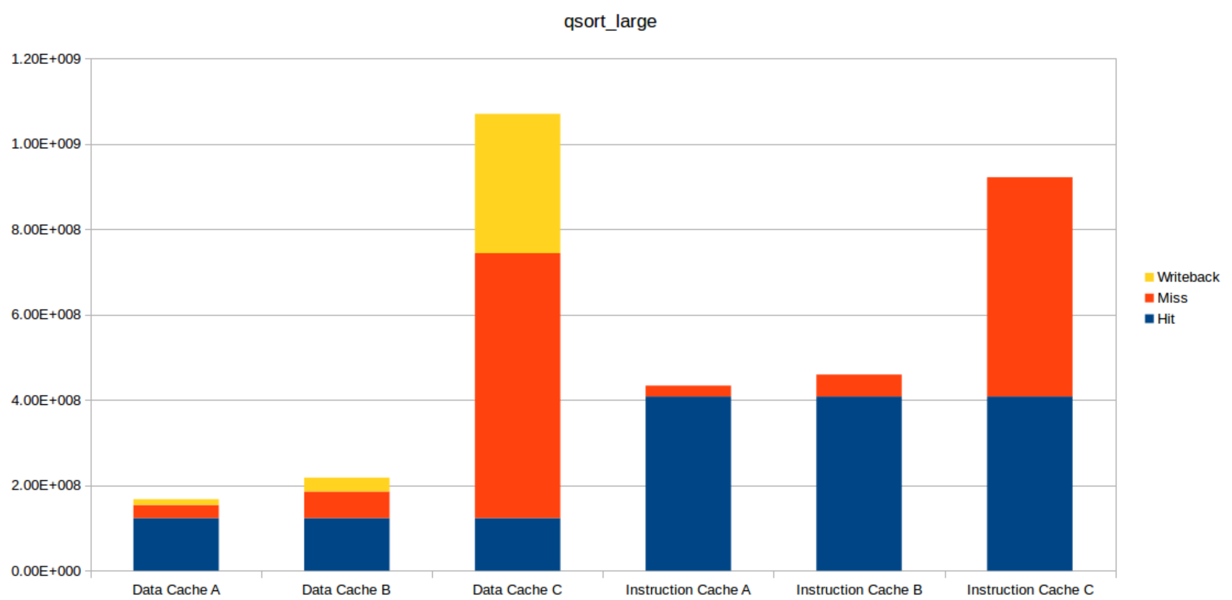

Figure 3.6: qsort large Power Projections showing number of Writebacks, Misses and Hits

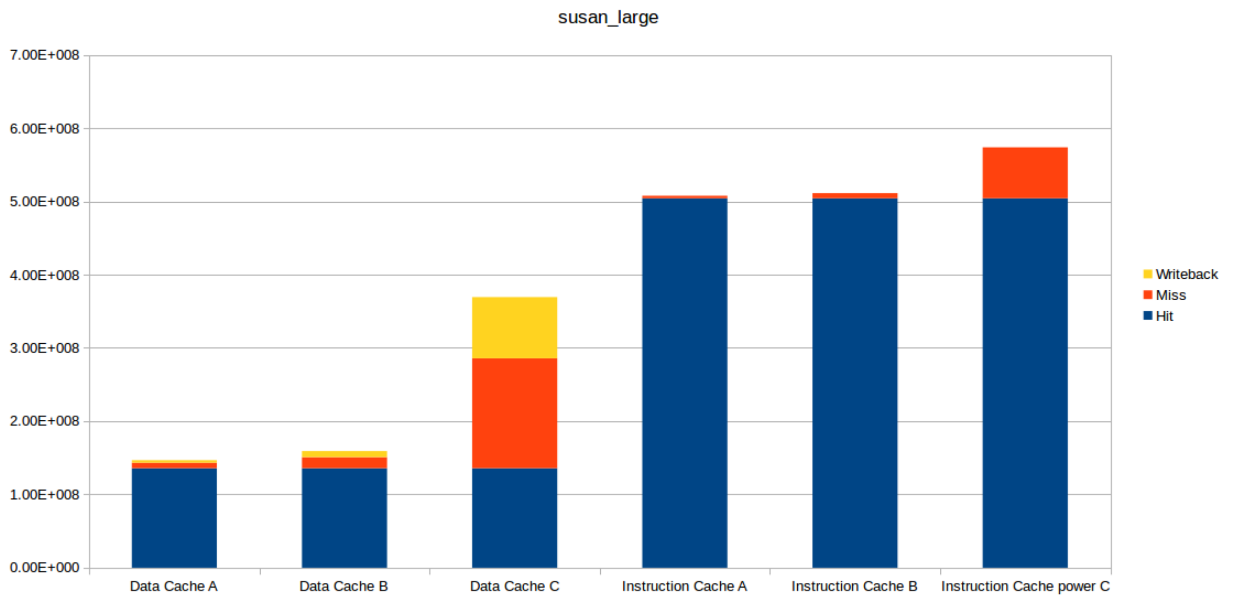

Figure 3.7: susan large Power Projections showing number of Writebacks, Misses and Hits

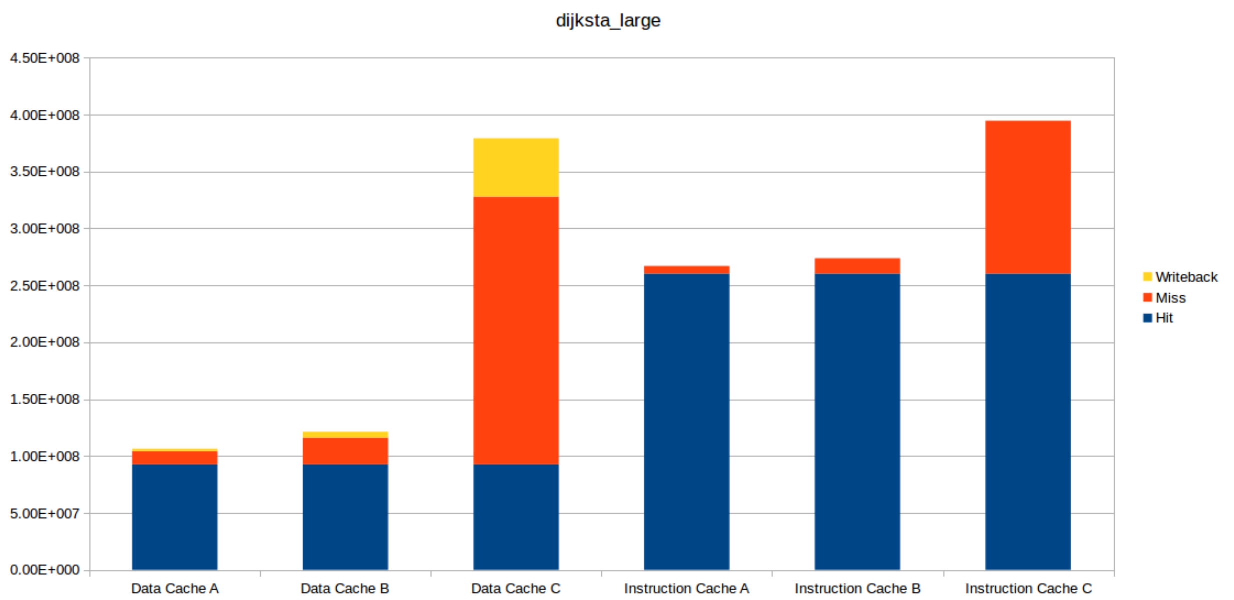

Figure 3.8: dijkstra large Power Projections showing number of Writebacks, Misses and Hits

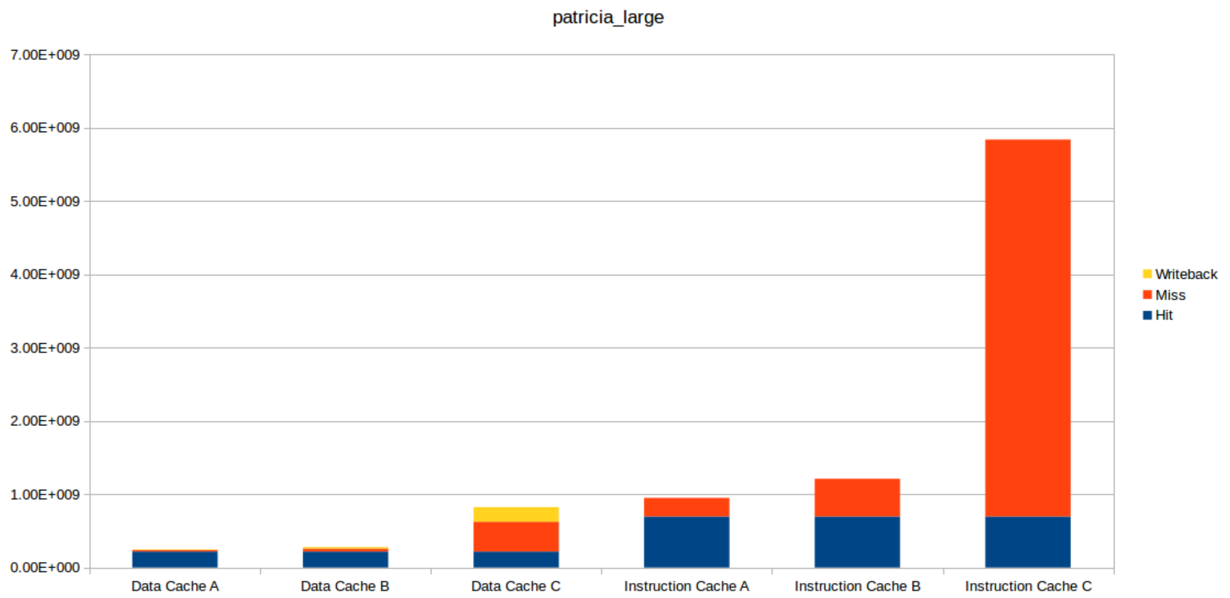

Figure 3.9: patricia large Power Projections showing number of Writebacks, Misses and Hits

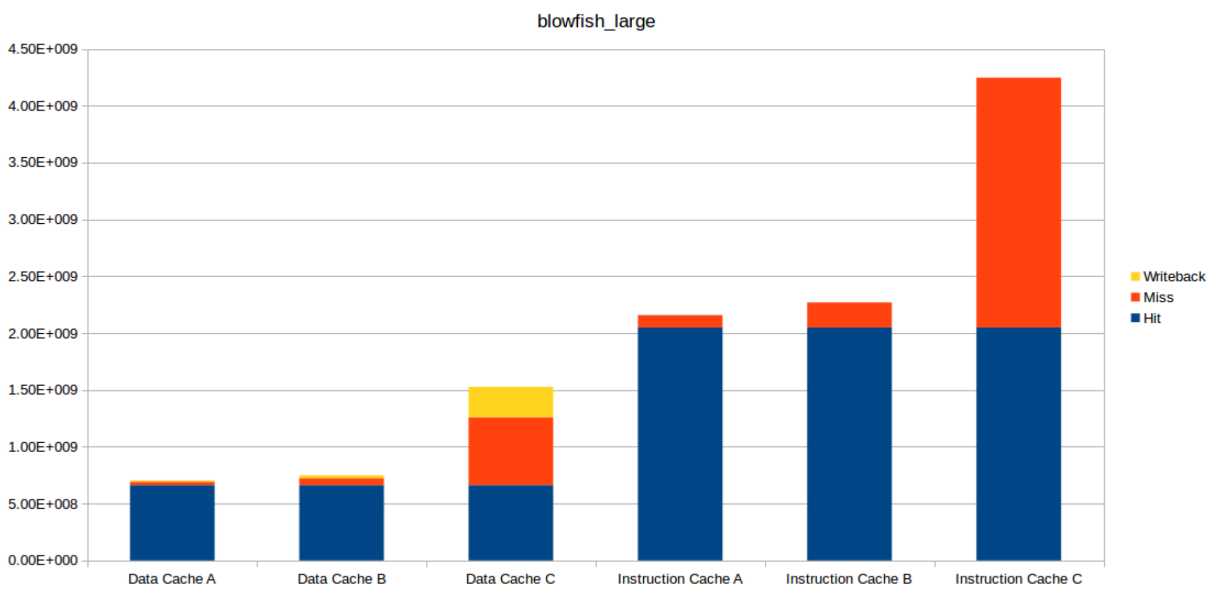

Figure 3.10: blowfish large Power Projections showing number of Writebacks, Misses and Hits

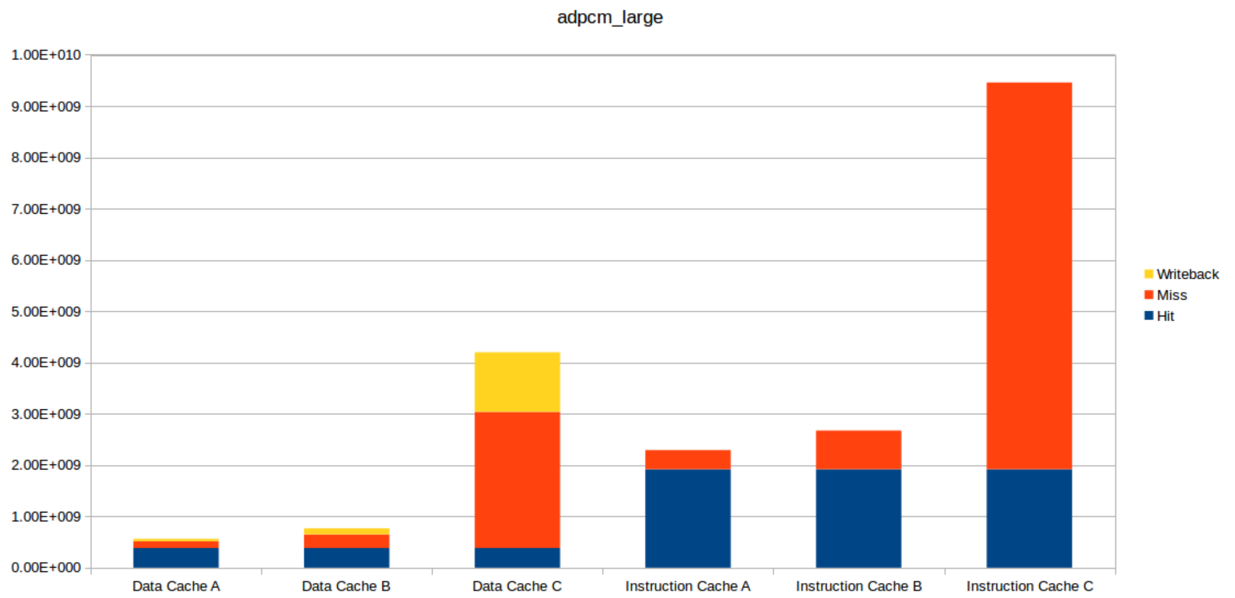

Figure 3.11: adpcm large Power Projections showing number of Writebacks, Misses and Hits

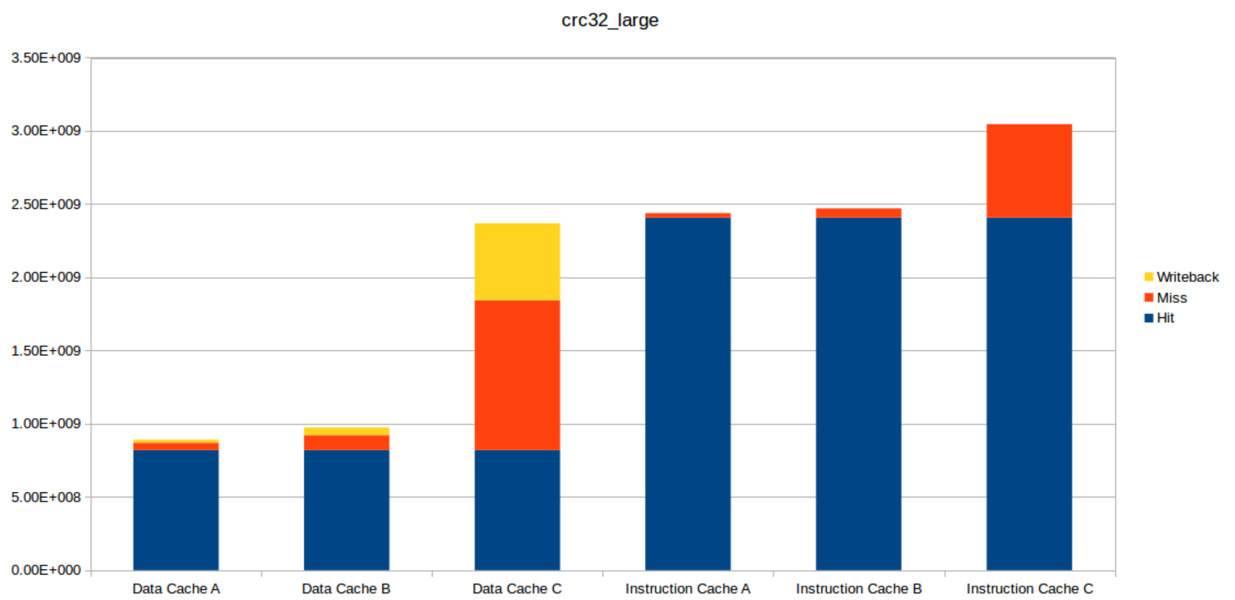

Figure 3.12: crc32 large Power Projections showing number of Writebacks, Misses and Hits

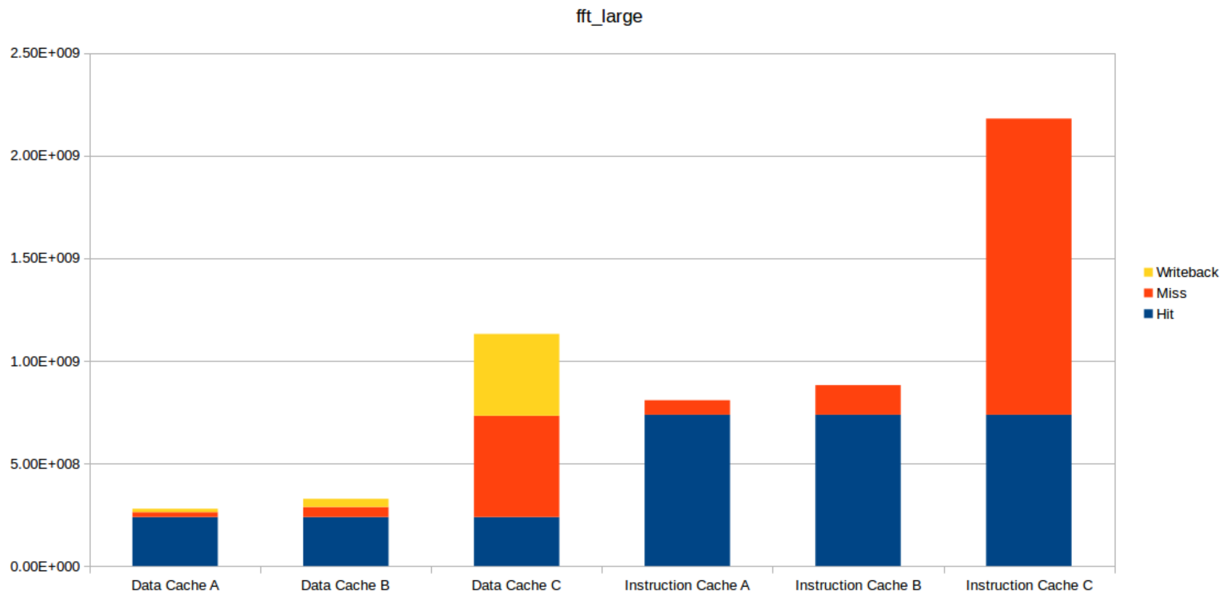

Figure 3.13: fft large Power Projections showing number of Writebacks, Misses and Hits

The results indicate that for the energy estimates from [3] A, the impact of accessing external memory (due to cache misses and write-backs) is not very significant provided that an appropriate sized cache is used. Even assuming worst case values B, the additional energy values do not appear too troublesome. The pessimistic values C are very significant. However, this is expected since the overhead is a order of magnitude greater than even the worst case assumption and two orders of magnitude away from the more realistic values obtained by Verma et al. [3]. It can be assumed therefore, that time wasted due to accessing external memory is not a huge concern provided that appropriate caches are used. The benchmark energy projections illustrate that there is an underlying trend that most applications are completely reliant on complex caching hardware in order to reduce the number of memory accesses. Even then, 50-70% of the total power for a system is still consumed by the memory system, Verma et al. [3]. Further studies by Verma et al. [3] show that the memory subsystem consumes 65.2% and 45.9% of the total energy budget for uni-processor ARM and multi-processor ARM systems respectively.

3.4 Where Can Improvements be Made?

Since multiprocessors alone cannot solve the 3 Walls, it is necessary to reconsider where improvements can be made by reducing overall main memory accesses. Table 3.3 illustrates the layers in

the system stack which can be used as a discussion point.

Application
Programming Paradigm/Language
Compiler
ISA and Processor Architecture

Table 3.3: System Stack

The application layer can employ design techniques which reduce memory accesses. Software pre-fetching can be used to take advantage of algorithmic locality, Jacob [49]. Additionally, cache memory can be explicitly managed by locking down frequently accessed code blocks and data. Alternatively, the cache can be statically managed by the compiler in the form of *scratch pad memory*.

The programming paradigm[1] is the way in which the structure and elements of a program are defined. Particular programming paradigms may exhibit favourable memory access patterns which reduce memory accesses. However, it may not be possible or practical to switch paradigms to reduce memory transactions.

The compiler may employ algorithms which reduce memory accesses. For example, specific code generation and register allocation algorithms may result in reduced memory accesses. The compiler can also be used to manage scratch pad memory, as discussed previously.

Microprocessor architecture and ISA trends follow commercial interests and it is inevitable that the industry resists evolving to changing needs. For example, the influence of Hennessy and Patterson [8] on modern processor design is significant, LaForest [20]. ISAs have changed very little since the invention of the microprocessor, although the ISA may have a significant impact on memory accesses. For example, it seems intuitive that a program executing on a RISC architecture will require more memory accesses than a CISC-based architecture due to the load/store design philosophy. Instruction memory accesses for a given program could potentially be reduced by:

- Changing the encoding of the instructions
- Employing more abstract instructions

Changing the encoding of the instructions would involve keeping the same ISA, but compacting the instructions in some way, for example using Huffman Encoding. Alternatively, the instructions could be made context sensitive. Whilst, in theory, this approach could be used to reduce the

[1]The four main programming paradigms are: procedural (imperative), object oriented (imperative), functional (declarative) and logic (declarative).

instruction stream, there are likely to be practical problems. The microprocessor would need to decompress the instructions prior to execution. This would complicate the decoding and pipeline units which may inadvertently increase power consumption. The power consumption problem may be simply shifted from one part of the microprocessor to another.

3.5 Opportunities for Abstract Instructions

In order to further investigate the opportunities for an abstract ISA, a program implementing the bubble sort algorithm was compiled for the ARM Coretx M3 processor and then disassembled. Listing 3.1 illustrates the swap function used in many bubble sort implementations. Note that the Cortex M3 processor uses the Thumb2 instruction set. The assembly language code is non-optimised and has been commented to aid understanding. The code in bold highlights a sample sequence of instructions generated for an array element load and store. For the array load, the index is loaded first, then scaled to the size of an integer for the platform (4 bytes). The address of the array is then loaded and added to the scaled index to form the address of the array element in memory. This location can then be loaded as required.

In order to reduce the number of instructions required for the array load and store, more abstract instructions could be employed. For example, the Java Virtual Machine Specification (inspired by Forth) uses a set of instructions for an array load and array store. Specific instructions are used depending on the size of the array element. In other words, the instruction is used to specify the index scaling. For example an *iaload* is used for arrays of int, whereas *caload* is used for arrays of char. Listing 3.2 illustrates how the array load and store operations could be performed using a more abstract ISA similar to that used by the Java Virtual Machine (JVM). In this case, the abstract instructions map onto the language semantics as opposed to the language keywords.

The ratio of instruction counts for the array load operation is 5:3. This is a 40% reduction in instruction stream length with potential for reducing instruction memory accesses. The ratio of instruction counts for the array store operation is 6:4. This is a 33% reduction in instruction stream length. To summarise, the reduction in the instruction stream length is achieved by the following:

- The scaling of the array index is implied and specified by the instructions
- The addition of the scaled index and the array address is implied by the instruction

```
static void swap(int array[], int ndx1, int ndx2) {
; must save r4 and above.
; No need to save lr as this is a leaf function.
00 : b480        push {r7}
; Reserve space for transferring args in regs
; to params in stack.
02 : b087        sub sp, #28
; make r7 stack frame pointer
04 : af00        add r7, sp, #0
06 : 60f8        str r0, [r7, #12]; array
08 : 60b9        str r1, [r7, #8] ; ndx1
0a : 607a        str  r2, [r7, #4] ; ndx2
int temp = array[ndx1];
0c : 68bb        ldr r3, [r7, #8]; ndx1
0e : 009b        lsls r3, r3, #2; ndx1 * 4
10: 68fa         ldr r2, [r7, #12]; array
12: 18d3         adds r3, r2, r3; array + (ndx1 * 4)
14: 681b         ldr r3, [r3, #0]; array + (ndx1 * 4)
; temp = [array + (ndx1 * 4)]
16: 617b         lstr r3, [r7, 20]
array[ndx1] = array[ndx2];
18: 68bb         ldr r3, [r7, 8] ; ndx1
1a: 009b         lsls r3, r3, 2 ; ndx * 4
1c: 68fa         ldr r2, [r7, 12] ; array
1e: 18d3         adds r3, r2, r3 ; array + (ndx1 * 4)
20: 687a         ldr r2, [r7, #4] ; ndx2
22: 0092         lsls r2, r2, #2 ; ndx2 * 4
24: 68f9         ldr r1, [r7, #12] ; array
26: 188a         adds r2, r1, r2 ; array + (ndx2 * 4)
28: 6812         ldr r2, [r2, #0] ; array + (ndx2 * 4)
; [array + (ndx1 * 4)] = [array + (ndx2 * 4)]
2a: 601a         str r2, [r3, 0]
array[ndx2] = temp;
2c: 687b         ldr r3, [r7, #4] ; ndx2
2e: 009b         lsls r3, r3, #2 ; ndx2 * 4
30: 68fa         ldr r2, [r7, #12] ; array
32: 18d3         adds r3, r2, r3 ; array + (ndx2 * 4)
34: 697a         ldr r2, [r7, #20] ; temp
; temp = [array + (ndx2 * 4)]
36: 601a          str r2, [r3, #0]
38: f107 071c    add.w r7, r7, #28 ;
3c: 46bd         mov  sp, r7 ; restore sp
3e: bc80         pop {r7} ; restore r7
40: 4770         bx lr ; ret
42: bf00         nop ; delayed branch pipeline
```

Listing 3.1: Swap ARM Assembly

```
int temp = array[ndx1];
        ldr r2, [r7, #8]          ; ndx1
        ldr r3, [r7, #12]         ; array
        iaload r2, r3, r3

array[ndx2] = temp;
        ldr r1, [r7, #20]         ; temp
        ldr r2, [r7, #4]          ; ndx2
        ldr r3, [r7, #12]         ; array
        iastore r1, r2, r3
```

Listing 3.2: Load and Store using a hypothetical abstract ISA

The Java Byte Code (JBC) carries more semantic meaning with a closer mapping to the source language. This approach is possible because all Java objects (including arrays) are constructed on a JVM heap. Java arrays do not contain objects as such. Instead they hold references; essentially immutable pointers. In C, arrays can contain true aggregate data for example structures, or arrays. This means that instructions specifying all possible array element sizes could not be used all of the time. If an array of a non-primitive type is used, an alternative load instruction would be required which reads the element size in addition to the array address and the index. Depending on the length of the array and the alignment rules of the processor, it may even be possible to encode the scaling parameter in the instruction or index.

Changing the encoding of an instruction by reducing the instruction width results in fewer bits required for storing it in memory, shortening the overall instruction stream. However, if insufficient bits are available for encoding, then fewer instructions will be available for describing a given program, resulting in a longer instruction stream. There is clearly a tradeoff to be made when reducing the encoding scheme for an instruction set. If the encoding is too verbose or overly concise, this will result in larger programs and increased memory accesses.

Employing a more abstract ISA or augmenting an existing ISA with a set of abstract instructions could be used to shorten the instruction stream. If an instruction has a closer semantic meaning with the source program, then fewer instructions will be required to represent it. For example, RISC executable images are two to three times larger than reduced operand set computers (ROSC). One of the reasons for this lies with the motives of RISC; to simplify the hardware (and not necessarily reduce memory interaction). To address this problem, ARM have designed a 16-bit Thumb ISA as an alternative to the regular 32-bit ISA. Koopman argues that RISC computers spend much of their processing effort emulating stack machines [21]. Whether this is true or not, there is no doubt that processors do spend their time emulating programming language constructs and semantics.

Using abstract instructions is by no means a novel idea. Application Specific Instruction Processors (ASIPs) use abstract instructions to represent algorithm constructs. However, these are so application specific that they cannot be widely used in a general purpose manner. It appears that ISA design has been either too general, resulting in a many-to-one mapping between processor instructions and programming constructs, or too specific, which drastically limits the reusability of a particular instruction. The middle ground is to compose a set of abstract instructions representing language constructs. In order to map and use an abstract ISA with a high-level programming language, a number of decisions must be made regarding the language. For example:

- Should the ISA support a particular programming paradigm or language?
- Should the ISA have a direct mapping to the language keywords?

The approach taken by Audsley and Ward [11] is to provide direct support for keywords of a simplified imperative language called *Tiny*. This is known as a *Syntax Directed Language Specific Processor* (SDLP). It is different from Java Byte Code (JBC) which is also defined as a Language Specific Processor. The ISA for Tiny is a *top-down* design to provide a one-to-one mapping between language keyword/construct and instruction. JBC on the other hand is a *bottom-up design* where instructions map onto the requirements of a stack-based virtual machine.

An alternative approach based on Tiny would be to base the ISA on a subset of the C programming language. The benefit of this approach is that C is the most prevalent language in use and is considered a subset or a basis of other popular languages, for example C++ and Java. An additional benefit is that many compilers are available, and the translations from the language constructs to existing ISAs can be easily studied. Furthermore, C compiler front ends can be reused, requiring only the back ends to be written for a new ISA.

3.6 Opportunities for Abstract Expressions

A traditional microprocessor evaluates expressions on a step-by-step basis. Each (potentially large) expression, is decomposed into a set of simple sub expressions each consisting of a single operator and the necessary operands. Each simple sub expression is then processed in turn by the ALU. This decomposition is performed by the compiler, whilst adhering to the precedence and associativity rules of the language. An example of a complex expression being decomposed ready for the ALU is:

$$x = a * b + 100 - 1/c;$$

Next, the brackets indicate the evaluation order of the sub expressions, again following the precedence and associativity rules of the C programming language:

$$x = ((a * b) + 100) - (1/c);$$

A C language compiler might evaluate the expression as follows:

```
x = a*b
x += 100;
temp = 1 / c;
x -= temp;
```

Listing 3.3: C Compiler Evaluation

It can be seen that the complex expression has been decomposed into a set of four subexpressions. This decomposition is necessary since a traditional ALU is only capable of evaluating a single operator at a time. There is a one-to-many mapping between the expression described by the high-level language and the ISA. For example, for the ARM RISC processor, the following instruction stream could be generated by the compiler:

```
ldr r3, [r7, #12]
ldr r2, [r7, 8]
mul r3, r2, r3
add r2, r3, #100
movs r1, #1
ldr r3, [r7, #4]
sdiv r3, r1, r3
subs r3, r2, r3
str r3, [r7, #20]
```

Listing 3.4: ARM assembly for the expression

The processor must store the set of instructions in non-volatile memory, cache and instruction stream memory. It must also interpret each of the individual instructions which make up the expression. An alternative method is to employ a syntax-directed approach to expression evaluation. At the language level, this means processing expressions as opposed to individual arithmetic and logic operations. This requires a more complex expression unit capable of evaluating an expression tree. Ausdley [11] describes an *Expression Engine* which is used instead of an ALU. Such an approach could be used in order to reduce the encoding of expressions and therefore reduce the load on the memory system and core processor. Fewer bits are required for storing it in memory, shortening the overall instruction stream.

3.7 Summary

Microprocessor performance increased exponentially up to around 2004 and this coincided with Moore's law. However, performance has flat-lined due to the 3 Walls: power, ILP and memory, McCool et al. [41].

The clock frequency of microprocessors has effectively flat-lined since around 2005. This is because, as transistors get smaller, power density increases since these do not scale with size. Processor power consumption is exceeding the few hundred watts that can feasibly be dissipated in a practical computer system. ILP has hit a practical limit and there is still a disparity between processor speed and external memory access speed. Therefore, processor designers have been forced to adopt multiprocessor designs in order to improve throughput. However, parallelising software is challenging and Amdahl's law states that if 75% of a program can be executed in parallel with 128 processors, the program will execute only four times faster than it would with a single processor. This is an optimistic view, since such a system would have an upper-bound of quadratic communication paths between the cores. Organising these into a hierarchy creates bottlenecks: this is inevitable. There is also a risk of *dark silicon* which means areas of silicon may need to be powered down to control power consumption and silicon damage.

Verma et. al [3], demonstrated that 50-70% of the total power for a system is consumed by the memory system. As a result, processors are completely dependent on large and complex caches. Benchmark simulations indicate that, provided appropriate caches are used, the effects of external memory access can be tolerated.

Designers have overlooked the possibility of modifying the ISA in order to address the 3 walls. For example, if the length of the instruction stream for a program could be reduced this would reduce the pressure on the external memory. This may mean that smaller, less complex caches could be used. It may also be possible to lower the processor clock frequency whilst maintaining the same throughput. Since power is related to time and switching, this may reduce power consumption. The instruction stream length for a program could be reduced by employing abstract instructions and abstract expressions.

An approach based on Audsley and Ward [11] could be used to provide direct support for keywords of modern imperative languages based on C. This is known as a *Syntax Directed Language Specific Processor (SDLP).*

In order to design an alternative ISA and processor to address the problems discussed, the following instruction types must be considered:

- Instructions for ALU operations
- Instructions for Control flow

This thesis considers the equivalent of the above instruction types. Chapter 4 discusses the Expression Engine, which is the equivalent of the traditional processor ALU. Chapter 5 covers Abstract Instructions, which are the equivalent of Control Flow instructions. Finally, the thesis presents an Architecture in Chapter 6 illustrating how the expression engine and abstract instructions can be combined into a working software simulator.

Chapter 4

Expression Engine

This chapter proposes an *expression engine* that can be used in the architecture presented in Chapter 6.

4.1 Background

As discussed in Chapter 3, an alternative to a traditional ALU is an *Expression Engine* described by Audsley and Ward [11]. This approach to expression evaluation is by no means novel; it is based on Dataflow Computing which is described by Sharp [9]. Most programming languages are designed for execution or interpretation on processors employing the Von Neumann model. This is the classic architecture with a control unit, instruction register, program counter and a unified memory system which stores both instructions and data. Program execution is based on the *control flow* model where a sequence of operations is specified by the programmer. In dataflow computing, operations are executed in an order determined by the data interdependencies and the availability of the required hardware resources. The motivation for dataflow computers is to achieve higher throughput. Sharp [9] describes various dataflow topologies. The simplest is structured as a tree, where each node is used for a specific operation.

An ideal expression engine is capable of evaluating the majority of expressions in a single pass through the tree; this is the advantage over an ALU. To achieve this, an adequate number of levels are required with the appropriate operators. However, there is a trade-off to be made; increasing the number of levels and operators results in an increased number of bits required to encode the expression. The maximum possible number of bits required to encode an arbitrary expression is

exponential (due to the mathematical properties of a binary tree structure). Assuming that an expression engine is structured as a binary tree, with each input node taking 2 operands and a single output node emitting 1 operand, the number of bits required for configuration is[1]:

$$(nodes\ *\ operator\ bits)\ +\ (leaf\ nodes\ *\ input\ operands\ bits\ *\ 2)\ +\ output\ operand\ bits$$

which is:

$$(2^{levels}-1) * log2(number\ of\ operators) + (2^{levels}/2) * log2(address\ space) * 2 + log2(address\ space)$$

For example, if the expression unit has a height of 2 and each node has 8 operators, and an address space of 256 bytes, the number of bits required for encoding an expression is:

$$7 * 3 + 4 * 8 * 2 + 8 = 93$$

in the above example, 93 bits are required to support the evaluation of 7 sub-expressions; since there are $2^{levels} - 1$ nodes in a binary tree. In terms of parallelism, the 7 sub-expressions can be evaluated (or executed) in *levels* or *height + 1* steps.

It appears that the encoding length of an expression is dominated by the bits used to address the operand. The encoding length could be reduced if a register set was used instead of memory. For example, to address 16 registers, $log2(16)$ bits would be required. This would reduce the number of bits for an expression to:

$$7 * 3 + 4 * 4 * 2 + 4 = 57$$

Whilst it may be tempting to discount the approach of addressing memory directly in an expression encoding scheme for space saving reasons, it should be considered that using register-based addressing would still incur additional instruction stream length overhead. This overhead would be all of the additional *load* and *store* instructions required to manage the register file as with a RISC-based processor.

An ALU-based system provides opcodes for constructing an arbitrary expression from simple sub-expressions which means that all of the instructions contribute in a meaningful way to the final outcome. However, using an expression engine may mean that not all processing nodes are

[1]Note that the number of *levels* in a binary tree is *height* + 1, and the number of *leaf* nodes = (n+1)/2 = $((2^{levels} - 1) + 1) / 2$.

used effectively for the final outcome. For example, an expression such as $a = b * c$ consists of a multiplication and an assignment and therefore only utilises one or two nodes of a multi-node expression engine. However, all of the nodes still need to be configured, so the expression encoding would be the same length regardless. Even if nodes could be made *pass-through* to indicate they are not utilised, this would still need encoding in the expression. This would also maintain a uniform expression size, which would control complexity during decoding.

When considering the use of an expression engine instead of a traditional ALU, it appears that there are a number of trade-offs to consider. These include opportunity for parallelism, encoding length and programmability. Figure 4.1 illustrates how each of these attributes for application-specific ASIC, Uni-core, Multicore and the SDLP can be quantified.

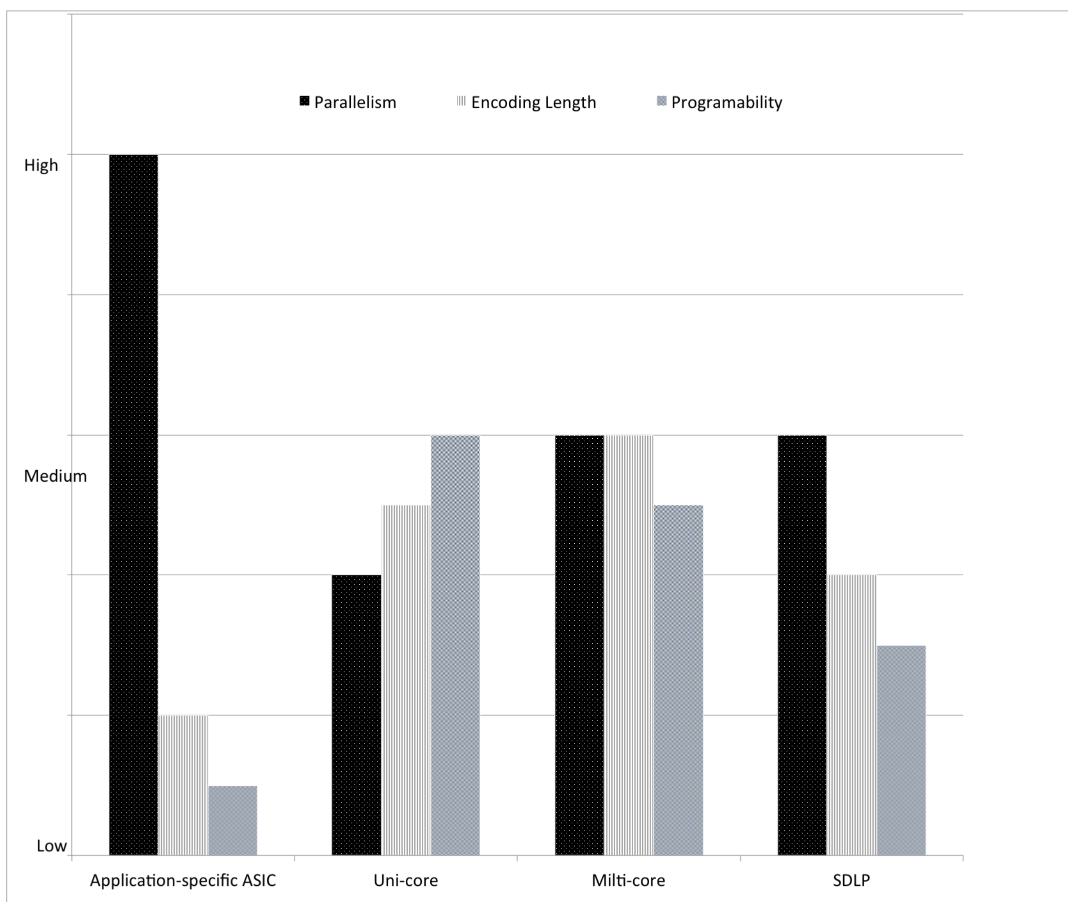

Figure 4.1: Quantifying the Opportunity for Parallelism, Encoding Length and Programmability for Application-Specific ASIC, Uni-core, Multicore and the SDLP, in terms of High, Medium and Low

It can be seen that an application-specific ASIC can be highly parallel with a minimum encoding length; this is due to the fact that ASICs are highly specialised circuits for solving specific problems. Consequently, they are the least flexible and hence the least programmable. Uni-core processors are very flexible computing devices and as such are highly programmable. However, this is at the cost of having a bigger encoding length and not being as parallelisable as ASICs. Multicore processors are similar to uni-core processors but offer limited improvement in parallelisation due to Amdahl's Law and an upper bound of quadratic communication paths between the cores. Programmability decreases due to the challenges of developing parallel software, McCool et al. [41]. The SDLP processor using an expression engine would be more difficult to program and would require a more complex code generator for the compiler back-end. However, the tradeoff is that many expressions can be encoded in less space than a traditional ALU. Expressions would be specified only once for a compilation unit[2] and then invoked via references to them. Multiple expression engines could be employed to increase parallelism further.

Note that computation and optimisation issues are not discussed within this thesis until future work is discussed in Chapter 8

4.2 Expression Engine Design

When considering the design of an expression unit, a number of design decisions must be considered as discussed in Section 2.1.5. The choice of a static or dynamic configuration can be considered first. A static configuration is defined at compilation time of the processor. A number of static configurations could be defined for *expression shapes*. A dynamic configuration is defined at software compilation time. For example, the compiler would emit instructions for a basic expression unit configuration for a set of common/basic expressions. However, it may then change the configuration of the expression unit to benefit the execution of more tailored expressions. These are likely to be domain or application specific. A dynamic configuration would naturally incur a higher hardware cost. This thesis limits considerations to a statically configured tree. More advanced approaches should be considered as additional work.

Another important design decision is the choice of operators to include and how they should be grouped together. Any arbitrary source base contains many expressions consisting of many shapes. However, when designing a static expression engine the aim is to find a configuration that is able to process as many expressions as possible in a single pass through the tree. Another conflicting

[2]A compilation unit is defined as separate unit of linkage, for example an object file in Linux

aim is to find a configuration that minimises *node wastage* in order to fully utilise the expression encoding and instruction stream. The choice of expression engine design is unlimited and finding an optimal configuration is NP-hard. The design could be done *ad-hoc* or it could be influenced empirically after collecting expression information for representative software that is likely to be executed on the processor. The middle ground is to analyse a source base to assist with an initial design.

To understand the operators needed in an expression tree, a compiler such as GCC could be instrumented. The output from the GCC parser is a tree-based intermediate representation called *GENERIC*. The purpose of GENERIC is to provide a language-independent way of representing an entire function [50]. This includes *declarations*, *types*, *statements* and *expressions*. If the output is appropriately instrumented, it will realise information about how expressions are composed; in particular, how operators are *grouped*. However, it is likely that for real-world applications the results obtained would contain many operator combinations, with a large range of frequencies. Analysis of expression usage therefore requires more structured future research. This should consider developing an appropriate analysis tool to extrapolate the operator usage at the *source language level*. In this thesis, an *ad-hoc* approach was taken for the expression tree design.

From a simplistic viewpoint, an expression tree can be constructed using logical grouping of arithmetic and logical operators. For example, addition and subtraction, multiplication and division. The nodes of the expression unit can be organised to support the precedence and associativity of C-based languages. For example, in C-based languages, the multiplication operator has higher precedence than the addition operator. It would therefore be sensible to have the multiplication operator nearer the leaf nodes and the addition operator closer to the root node so that the multiplication is completed before the addition.

If the expression engine is designed as a binary tree, it certainly makes it easy to calculate various properties of the tree. With this scheme, a tree with a height of 2 would have 7 processing nodes. However, each leaf node is required to take 2 inputs *rvalues* and produce a single output result; this cannot be classified as a *complete* or *full* binary tree. The design of the Expression Engine is illustrated in Figure 4.2. It can be observed in Figure 4.2 that it can be more accurately described as an *n-array* tree and it requires fewer bits than the binary tree. This is because each input node has 2 input *rvalues* and some nodes only require a single configuration bit. The expression tree is influenced by Audsley and Ward's expression tree illustrated in Figure 2.12, which is based on a dataflow model.

Two *rvalues* enter the leaf nodes at the bottom of the tree and propagate to the root node.

Since each node is capable of multiple operations, control bits are used to configure each node. The expression engine uses 8 bits for addressing an *rvalue/lvalue* in the memory subsystem. Therefore the expression engine imposes a *memory-memory* architecture as described by Hennessey and Paterson [8]. There are no explicit registers available, all operations are performed directly on storage elements.

It should be noted that an 8-bit address width for operands (*rvalues* and *lvalues*) places severe limitations on the addressing capabilities of the expression engine. This limitation means that currently, only 256 bytes can be accessed. Future work will need to focus on reducing this limitation whilst keeping the encoding length of an expression to a minimum.

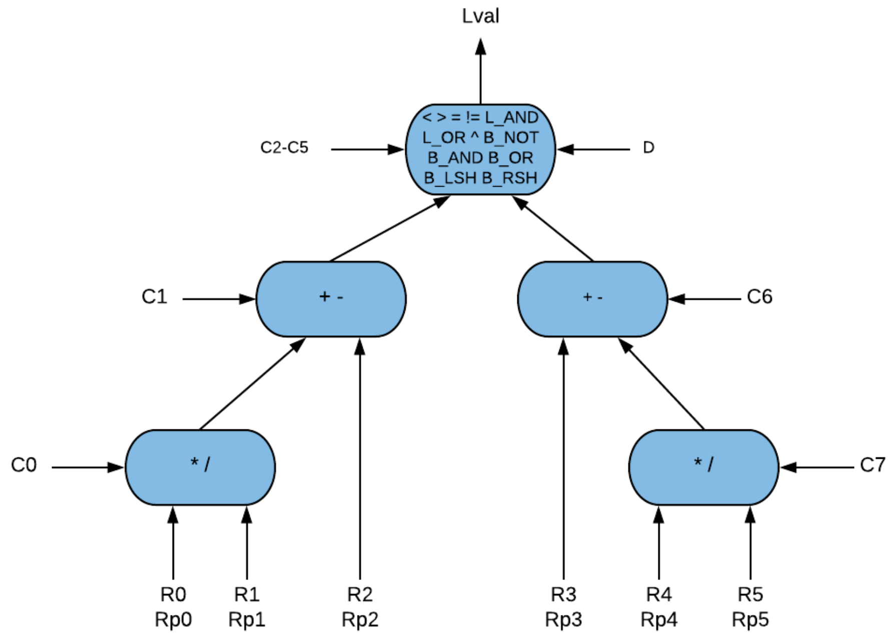

Figure 4.2: Expression Engine

In order to evaluate an expression, the following information is required to configure each node in layers 1 - 2 (see Figure 4.2):

- The operators (C0 , C1, C6, C7)
- The input operands (R0 - R5). These refer to *rvalues*.
- How the rvalues should be addressed (Rp0 - Rp5)

For the level 3 output node:

- The operator (C2 - C5)
- The output operand (lval)
- How the lvalue should be addressed (D)

4.3 Expression Encoding

Listing 4.1 illustrates the Backus-Naur form for an expression and how it is described for the Assembler.

```
<expression> := <expression name> :
                <C0> <C1> <C2–C5> <C6> <C7>
                <RP0> <RP1> <RP2> <RP3> <RP4> <RP5>
                <R0> <R0> <R2> <R3> <R4> <R5>
                <D>
                <lval>
```

Listing 4.1: BNF for an Expression

R0 - R5 are *rvalues* and are the equivalent of variables or constants on the right hand side of a C expression. The output operand is referred to as an *lvalue* (left of assignment). The size of each *lvaue* and the *rvalue* is 8 bits and can therefore address 256 memory locations. For each node the operator must be selected, for example multiplication or division. Nodes in layers 1-2 (the bottom two layers) contain basic arithmetic operators, whereas the node in layer 3 (the top layer) contains logical and bitwise operators. The reason for this arrangement is simplicity in configuring the expression tree for general expressions.

Listing 4.2 illustrates the possible operators.

```
<C2–C5> := '<' | '>' | '=' | '!=' | L_AND
           | L_OR | ^ | B_NOT | B_AND | B_OR | B_LSH | B_RSH
<C0> := '*' | '/'
<C7> := '*' | '/'
<C1> := '+' | '-'
<C6> := '+' | '-'
```

Listing 4.2: BNF for Operators

Table 4.1 illustrates the values for each of the operators.

Node	Operator	Value
C0 and C7	* (arithmetic multiply)	0
	/ (arithmetic divide)	1
C1 and C6	+ (arithmetic add)	0
	- (arithmetic subtract)	1
C2 - C5	< (relational less than)	0
	> (relational greater than)	1
	= (relational equals)	2
	! = (relational not equals)	3
	L_AND (logical and)	4
	L_OR (logical or)	5
	ˆ(bitwise xor)	6
	B_NOT (bitwise not)	7
	B_AND (bitwise and)	8
	B_OR (bitwise and)	9
	B_LSH (bitwise left shift)	10
	B_RSH (bitwise right shift)	11

Table 4.1: SDLP Operators

RP0 - RP5 are used to specify how each of the corresponding *rvalues* should be interpreted. The possible values for RP0 - RP5 in layers 1-2 are illustrated in Table 4.2:

Description	Value
rvalue (offset from .data)	0
Address of the *rvalue* (& in C) (offset from .data)	1
Value of the location pointed to by the rvalue (* in C) (offset from .data)	2
Not used	3
rvalue (offset from stack base)	4
Address of the *rvalue* (& in C) (offset from stack base)	5
Value of the location pointed to by the *rvalue* (* in C) (offset from stack base)	6
Literal (value is treated as a literal)	7

Table 4.2: RP0 - RP5 Meanings

It can be observed that *rvalues* can refer to variables in a chosen segment, pointers to variables in a segment and the offset addresses of variables in a segment. An *rvalue* can also be an 8-bit literal value. Whilst this may seem restrictive, it would be possible to have variable length literals (for example, 8, 16, and 32 bit). However, this in turn would require expressions to be variable lengths. The possible values for D in layer 3 are illustrated in Table 4.3:

Similar to *rvalues*, *lvalues* can simply be variables, pointers and address of variables. It is also possible to skip writing the *lvalue*. This may be done when the result of the expression is used as

Description	**Value**
lvalue (offset from from .data)	0
Location pointed to by the *lvalue* (* in C) (offset from .data)	1
lvalue (offset from stack base)	2
Location pointed to by the *lvalue* (* in C) (offset from stack base)	3
Not used	4
Not used	5
Not used	6
Ignore (do not write *lvalue*, however internal flag still set to indicate result of expression result)	7

Table 4.3: D Meanings

a condition used by an instruction such as an *if* or a *while*. In this case an internal condition flag is set if the *lvalue* is non-zero and unset otherwise, i.e. $flag = lvalue$.

Listing 4.3 illustrates how the Assembler formats an expression for execution; this is the C structure used by the Assembler. It can be noted that the size of this structure is 85 bits which is rounded up to 11 bytes.

```c
typedef union {
  struct  {
    uint8_t c0   : 1;
    uint8_t c1   : 1;
    uint8_t c2c5 : 4;
    uint8_t c6   : 1;
    uint8_t c7   : 1;

    uint8_t r0p  : 3;
    uint8_t r1p  : 3;
    uint8_t r2p  : 3;
    uint8_t r3p  : 3;
    uint8_t r4p  : 3;
    uint8_t r5p  : 3;
    uint8_t r0r5[6];

    uint8_t d    : 3;
    uint8_t lval;
  } __attribute__((packed)) expr;

  uint8_t bytes[sizeof(expr)];
} __attribute__((packed)) expression_t;
```

Listing 4.3: C Structure used by the Assembler Representing an Expression

4.4 Example Encodings

It is useful to illustrate how an expression is programmed for the assembler, how the expression is represented as a tree and how it is represented as a byte stream for execution by the processor. To achieve this, a set of expressions are taken from a *linear search* program which searches for the biggest value in an integer array. This can be seen in Listing 6.3.

Table 4.4 illustrates how each of the C expressions are specified for the SDLP assembler. For each SDLP expression, a reference to the corresponding tree diagram and disassembly is given, which are also illustrated.

C Expression	**SDLP Assembly**	**Tree (Figure)**	**Disassembly (Table)**
int *ptr = &a0[0];	expr_assign_ptr_addr_of_a0: *, +, B_AND, +, *, 1, 7, 7, 7, 7, 7, a0, 1, 0, 255, 1, 0, 0, ptr;	4.3	4.5
if(count < numElement)	expr_count_less_than_numElement: *, +, <, +, *, 0, 7, 7, 0, 7, 7, count, 1, 0, numElements, 1, 0, 7, ignore	4.4	4.6
count++;	expr_inc_count: *, +, B_AND, +, *, 0, 7, 7, 7, 7, 7, count, 1, 1, 255, 1, 0, 0, count;	4.5	4.7
if(*ptr > biggest)	expr_element_greater_than_biggest: *, +, >, +, *, 2, 7, 7, 0, 7, 7, ptr, 1, 0, biggest, 1, 0, 7, ignore;	4.6	4.8
biggest = *ptr;	expr_assign_element_to_biggest: *, +, B_AND, +, *, 2, 7, 7, 7, 7, 7, ptr, 1, 0, 255, 1, 0, 0, biggest;	4.7	4.9
ptr++;	expr_inc_ptr: *, +, B_AND, +, *, 0, 7, 7, 7, 7, 7, ptr, 1, 4, 255, 1, 0, 0, ptr;	4.8	4.10

Table 4.4: C Expressions and Equivalent SDLP Assembly

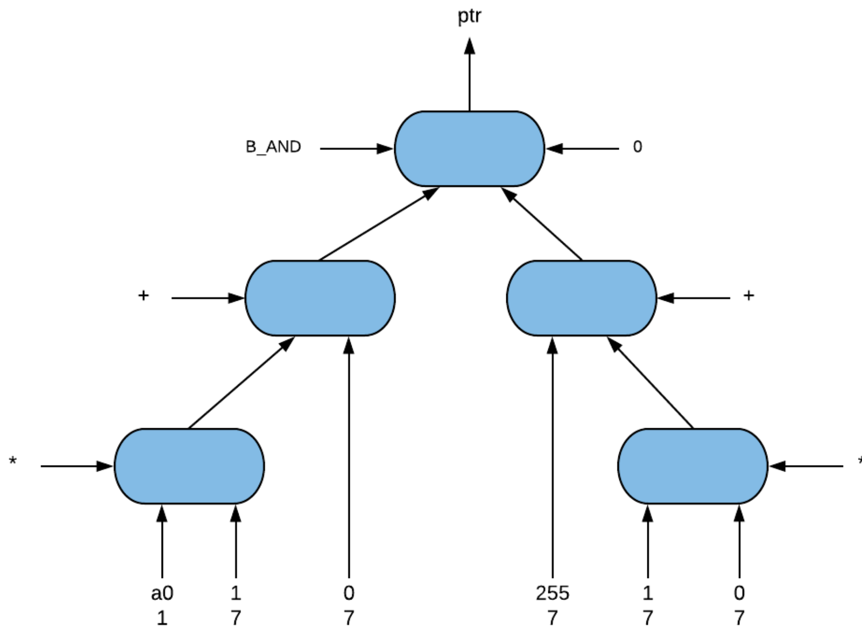

Figure 4.3: expr_assign_ptr_addr_of_a0

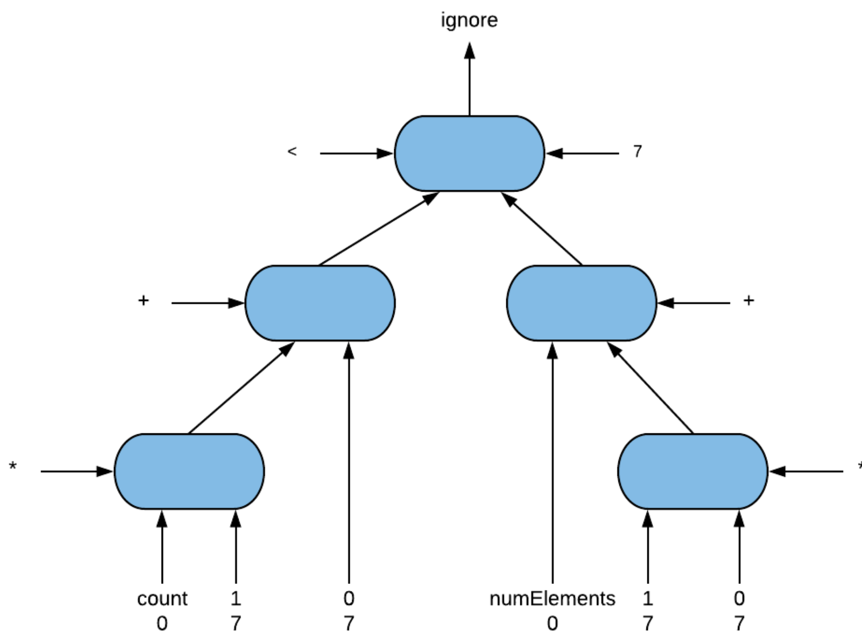

Figure 4.4: expr_count_less_than_numElement

Hex:	20 F9 FF 3F 00 01 00 FF 01 00 40 30	
expression_t element	**Size in bits**	**Value**
c0	1	0x0
c1	1	0x0
c2c5	4	0x8
c6	1	0x0
c7	1	0x0
r0p	3	0x1
rlp	3	0x7
r2p	3	0x7
r3p	3	0x7
r4p	3	0x7
r5p	3	0x7
r0	8	0x0
r1	8	0x1
r2	8	0x0
r3	8	0xFF
r4	8	0x1
r5	8	0x0
d	3	0x0
ival	8	0x30

Table 4.5: Disassembly of expr_assign_ptr_addr_of_a0

Hex:	00 F8 C7 3F 28 01 00 2C 01 00 47 38	
expression_t element	**Size in bits**	**Value**
c0	1	0x0
c1	1	0x0
c2c5	4	0x0
c6	1	0x0
c7	1	0x0
r0p	3	0x0
rlp	3	0x7
r2p	3	0x7
r3p	3	0x0
r4p	3	0x7
r5p	3	0x7
r0	8	0x28
r1	8	0x1
r2	8	0x0
r3	8	0x2C
r4	8	0x1
r5	8	0x0
d	3	0x7
ival	8	0x38

Table 4.6: Disassembly of expr_count_less_than_numElement

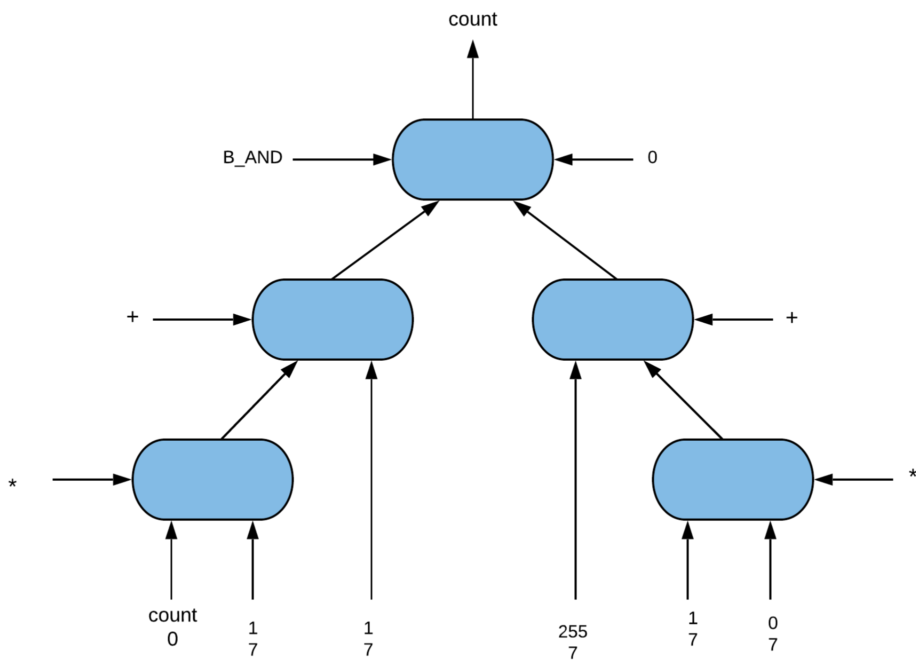

Figure 4.5: expr_inc_count

Hex:	20 F8 FF 3F 28 01 01 FF 01 00 40 28	
expression_t element	**Size in bits**	**Value**
c0	1	0x0
c1	1	0x0
c2c5	4	0x8
c6	1	0x0
c7	1	0x0
r0p	3	0x0
r1p	3	0x7
r2p	3	0x7
r3p	3	0x7
r4p	3	0x7
r5p	3	0x7
r0	8	0x28
r1	8	0x1
r2	8	0x1
r3	8	0xFF
r4	8	0x1
r5	8	0x0
d	3	0x0
ival	8	0x28

Table 4.7: Disassembly of expr_inc_count

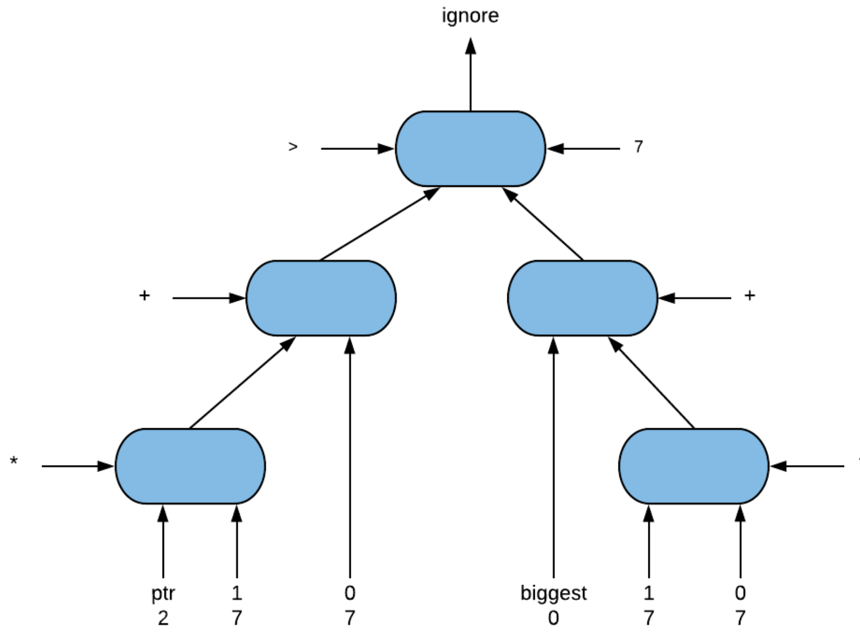

Figure 4.6: expr_element_greater_than_biggest

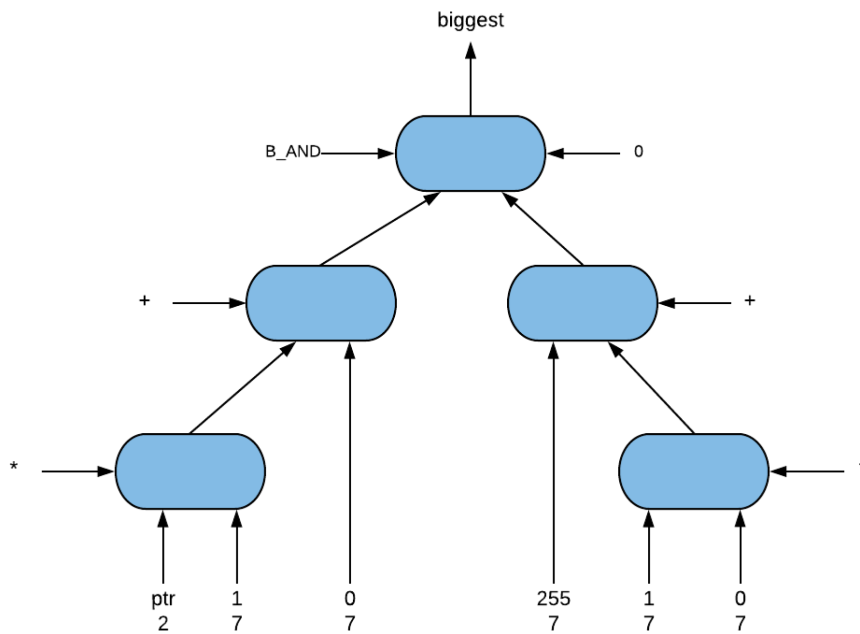

Figure 4.7: expr_assign_element_to_biggest

Hex:	04 FA C7 3F 30 01 00 34 01 00 47 38	
expression_t element	**Size in bits**	**Value**
c0	1	0x0
c1	1	0x0
c2c5	4	0x1
c6	1	0x0
c7	1	0x0
r0p	3	0x2
r1p	3	0x7
r2p	3	0x7
r3p	3	0x0
r4p	3	0x7
r5p	3	0x7
r0	8	0x30
r1	8	0x1
r2	8	0x0
r3	8	0x34
r4	8	0x1
r5	8	0x0
d	3	0x7
ival	8	0x38

Table 4.8: Disassembly of expr_element_greater_than_biggest

Hex:	20 FA FF 3F 30 01 00 FF 01 00 40 34	
expression_t element	**Size in bits**	**Value**
c0	1	0x0
c1	1	0x0
c2c5	4	0x8
c6	1	0x0
c7	1	0x0
r0p	3	0x2
r1p	3	0x7
r2p	3	0x7
r3p	3	0x7
r4p	3	0x7
r5p	3	0x7
r0	8	0x30
r1	8	0x1
r2	8	0x0
r3	8	0xFF
r4	8	0x1
r5	8	0x0
d	3	0x0
ival	8	0x34

Table 4.9: Disassembly of expr_assign_element_to_biggest

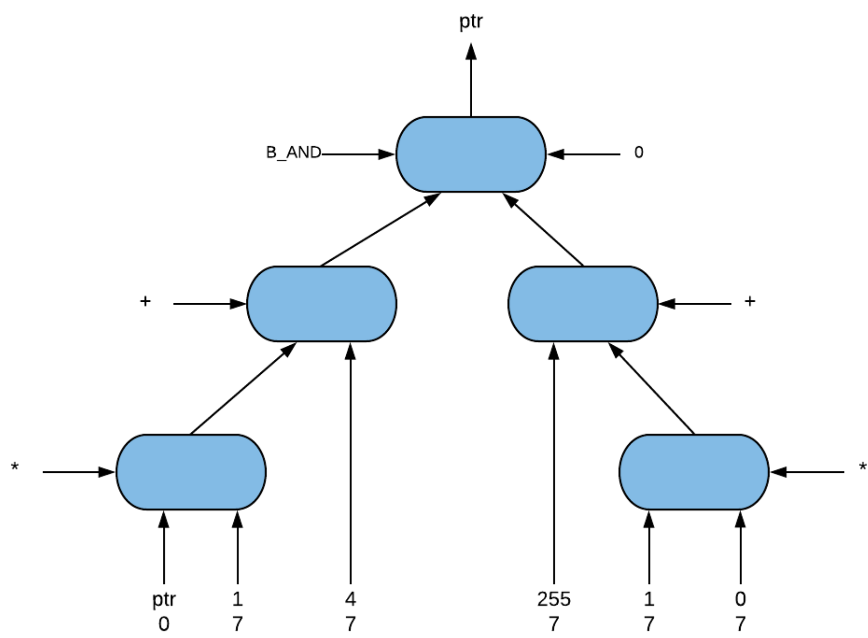

Figure 4.8: expr_inc_ptr

Hex:	20 F8 FF 3F 30 01 04 FF 01 00 40 30	
expression_t element	**Size in bits**	**Value**
c0	1	0x0
c1	1	0x0
c2c5	4	0x8
c6	1	0x0
c7	1	0x0
r0p	3	0x0
r1p	3	0x7
r2p	3	0x7
r3p	3	0x7
r4p	3	0x7
r5p	3	0x7
r0	8	0x30
r1	8	0x1
r2	8	0x4
r3	8	0xFF
r4	8	0x1
r5	8	0x0
d	3	0x0
ival	8	0x30

Table 4.10: Disassembly of expr_inc_ptr

4.5 Comparison of Expression Encoding

The primary motives for the expression engine instead of a traditional ALU are to:

- Reduce the length of the instruction stream, thus reduce the dependency on the memory system;
- Increase the level of parallelism executing an expression.

To determine the potential improvements, the encoding scheme for the expression engine can be compared with an instruction stream of another architecture, e.g. ARM Thumb. Since ARM Thumb is used to reduce the overall instruction stream length, and hence reduce the dependency of the memory system of a RISC-based architecture, this comparison should be of significant interest. To achieve this, the ARM Thumb assembly generated by the *GCC* compiler (unoptimised) was obtained for the expressions taken from the *linear search* program. The C expressions are illustrated in Listing 4.4. Note, these are the same C expressions shown in Table 4.4.

```
int *ptr = &a0[0];
if(count < numElement)
count++;
if(*ptr > biggest)
biggest = *ptr;
ptr++;
```

Listing 4.4: Linear Search Expressions and Corresponding C Expressions

Listing 4.5 illustrates the ARM Thumb disassembly for the expressions. Since this is Thumb code, it is half the size of normal code. A Thumb instruction size is 16 bits as opposed to a normal ARM instruction of 32 bits.

```
// int *ptr = &a0[0];
8428:    1c3b            adds    r3, r7, #0
842a:    3310            adds    r3, #16
842c:    607b            str     r3, [r7, #4] // ptr

// if(count < numElement)
8436:    683a            ldr     r2, [r7, #0] // count
8438:    68bb            ldr     r3, [r7, #8] // numElement
843a:    429a            cmp     r2, r3

// count++;
8442:    683b            ldr     r3, [r7, #0] // count
8444:    3301            adds    r3, #1
8446:    603b            str     r3, [r7, #0]

// if(*ptr > biggest)
844c:    687b            ldr     r3, [r7, #4] // ptr
844e:    681a            ldr     r2, [r3, #0] // dereference ptr
8450:    68fb            ldr     r3, [r7, #12] // biggest
8452:    429a            cmp     r2, r3

// biggest = *ptr;
845a:    687b            ldr     r3, [r7, #4] // ptr
845c:    681b            ldr     r3, [r3, #0] // dereference ptr
845e:    60fb            str     r3, [r7, #12] // biggest

// ptr++;
8460:    687b            ldr     r3, [r7, #4] //ptr
8462:    3304            adds    r3, #4
8464:    607b            str     r3, [r7, #4]
```

Listing 4.5: ARM Thumb Disassembly for the expressions

The instruction stream length for the above is 38 bytes.

The C code was written using *local variables*; hence it can be observed that the code is accessing variables from the stack. An alternative implementation was written using *global variables.* Listing 4.6 illustrates the alternative ARM Thumb disassembly for the expressions.

```
// int *ptr = &a0[0];
8410:   4b18        ldr    r3, [pc, #96]    // ptr
8412:   4a19        ldr    r2, [pc, #100] // address of a0
// store address of a0 into ptr
8414:   601a        str    r2, [r3, #0]

// if(count < numElement)
8422:   4b16        ldr    r3, [pc, #88]
8424:   681a        ldr    r2, [r3, #0]
8426:   4b16        ldr    r3, [pc, #88]
8428:   681b        ldr    r3, [r3, #0]
842a:   429a        cmp    r2, r3

// count++;
8434:   4b11        ldr    r3, [pc, #68]
8436:   681b        ldr    r3, [r3, #0]
8438:   1c5a        adds   r2, r3, #1
843a:   4b10        ldr    r3, [pc, #64]
843c:   601a        str    r2, [r3, #0]

// if(*ptr > biggest)
8444:   4b0b        ldr    r3, [pc, #44]
8446:   681b        ldr    r3, [r3, #0]
8448:   681a        ldr    r2, [r3, #0]
844a:   4b0e        ldr    r3, [pc, #56]
844c:   681b        ldr    r3, [r3, #0]
844e:   429a        cmp    r2, r3

// biggest = *ptr;
8458:   4b06        ldr    r3, [pc, #24]
845a:   681b        ldr    r3, [r3, #0]
845c:   681a        ldr    r2, [r3, #0]
845e:   4b09        ldr    r3, [pc, #36]
8460:   601a        str    r2, [r3, #0]

// ptr++;
8462:   4b04        ldr    r3, [pc, #16]
8464:   681b        ldr    r3, [r3, #0]
8466:   1d1a        adds   r2, r3, #4
8468:   4b02        ldr    r3, [pc, #8]
846a:   601a        str    r2, [r3, #0]
```

Listing 4.6: Alternative ARM Thumb Disassembly for the expressions

The instruction stream length for the above is 58 bytes.

It can be seen that *pc-relative* addressing is used to obtain the variables. In the case of the expression *if(count < numElement)* the number of loads required is almost double. This is because the compiler has generated instructions to first load the address of a literal pool into a register. The literal pool is essentially an array of values. It then uses register indirect addressing to load the

contents of the memory pointed to by the register into a second register. This happens regardless of optimisation flags used during compilation. The reason that ARM ISA requires this is because of the uniform size of instructions. They must be either 16 bits for Thumb or 32 bits for normal ARM mode; literals cannot be interjected in the byte stream.

Figure 4.7 illustrates the expressions encoded using the expression engine.

```
expr_assign_ptr_addr_of_a0:
*, +, B_AND, +, *,
1, 7, 7, 7, 7, 7,
a0, 1, 0, 255, 1, 0,
0, ptr;

expr_count_less_than_numElement:
*, +, <, +, *,
0, 7, 7, 0, 7, 7,
count, 1, 0, numElements, 1, 0,
7, ignore

expr_inc_count:
*, +, B_AND, +, *,
0, 7, 7, 7, 7, 7,
count, 1, 1, 255, 1, 0,
0, count;

expr_element_greater_than_biggest:
*, +, >, +, *,
2, 7, 7, 0, 7, 7,
ptr, 1, 0, biggest, 1, 0,
7, ignore;

expr_assign_element_to_biggest:
*, +, B_AND, +, *,
2, 7, 7, 7, 7, 7,
ptr, 1, 0, 255, 1, 0,
0, biggest;

expr_inc_ptr:
*, +, B_AND, +, *,
0, 7, 7, 7, 7, 7,
ptr, 1, 4, 255, 1, 0,
0, ptr;
```

Listing 4.7: Equivalent expressions using expression engine

Each expression for the Expression Engine is 11 bytes, therefore a total of 66 bytes are required for all of the expression definitions. However, in order to execute an expression, a reference is required in the instruction stream. The size of each expression reference is 1 byte. Therefore in order to define and *invoke* an expression reference, 12 bytes are required. Therefore, a total of 72

bytes are required for all of the expression definitions and invocations. This is 14 bytes more than the PC-relative ARM Thumb assembly and 34 bytes more than the ARM Thumb code which uses local variables.

However, this is not a fair comparison. If the size of an expression is adjusted to assume 16 registers rather than 256 bytes of memory, 8 fewer bytes are needed per expression definition. This is $11\ bytes\ =\ 88\ bits,\ then\ 88\ bits - (7*4)\ bits = 60\ bits, or \approx 8\ bytes$. This would be a total of 48 bytes for all of the expression definitions and a total of $48 + 6 = 54\ bytes$ for the definitions and invocation references.

This is 16 bytes more than the ARM Thumb code but 4 bytes fewer than the ARM Thumb code using global variables (i.e. literal pool). It is clear that opcode addressing dominates the expression definition space requirements. It should be noted at this point that a maximum of 256 expressions can be invoked using a 1-byte reference.

To estimate the clock cycles or throughput for both the expression engine and ARM, it is simpler to remove the effect of pipelining and hence intrinsic and extrinsic stalls. This is reasonable, as pipelining is implementation specific and is an optimisation that the expression engine could potentially utilise. With this in mind, instead of counting the number of clock cycles, the number of *instruction steps* can be counted; this is simply the instruction count for the ARM assembly. Whilst this approach does not account for the loading and storing of operands and the addressing modes required for this, it does provide a rough estimate. For the expression engine, the number of steps would equate to the number of levels in the tree, i.e. 3. The total number of clock cycles for the expression engine is 18. The total number of steps required for ARM Thumb using pc-relative addressing is is 29. The total number of steps for ARM Thumb using local variables is 19. Whilst the expression engine appears only marginally better than ARM Thumb, the approach taken here gives optimistic counts for ARM and pessimistic counts for the SDLP. The ARM Thumb is likely to suffer from pipelining stalls, whereas the SDLP is able to use other implementation optimisations. For example, it may be possible for the expression engine to be implemented using asynchronous logic in order to reduce the number of clock cycles.

When considering a *one size fits all* expression engine there appears to be wastage, where nodes may not contributing to an expression outcome. For example, *expr_inc_count* does not make use of the level-one nodes which perform multiplication and division. This can be seen where the left level-one node multiples count by one in order to pass-through count to the level-two node. The right level-one node simply multiplies 1 by 0 and the right level-two node adds 0 to 255. This is so that the value from the left hand side of the expression tree can be bitwise *anded* with the large

number (255) so that it can be outputted as the *lval*. It can be observed that 4 out of 6 of the expressions under utilise the expression engine in this way; half of the nodes are effectively wasted.

It is clear that the the design of the expression engine requires more consideration. Various techniques for saving dynamic power should be considered for future research. For example, *operand forwarding*, similar to that used for minimising data hazards in pipelining, Hennessy and Patterson [2, C-14] could be used to bypass unnecessary nodes. *Clock gating* described by Shinde and Salankar [51], would allow bypassed or unused nodes to be disabled during a clock cycle. The expression engine could be designed to be *dynamic*. For example nodes such as (*/ and +-) could swap positions or *levels*. The number of node operations could be increased and nodes could even be dynamically *linked*. However, dynamic configuration would require more bits for expression encoding, which ultimately may increase the dependency on the memory system.

4.6 Summary

An *Expression Engine* influenced by dataflow computing is suggested as a means to to achieve higher throughput and reduced instruction stream length. The expression engine is structured as an n-array tree. Programming the SDLP using assembly language is significantly more difficult than traditional microprocessors. A compiler back end for the expression engine code generator is likely to be significantly more complex.

To aid progress in determining the potential advantages of an expression engine, an *ad-hoc* approach was taken. A static n-array tree structure with 3 levels has been suggested. Each expression can be encoded in 11 bytes. Example expressions were used to compare the SDLP with ARM and X86-based processors. ARM Thumb disassembly using local variables requires an instruction stream length of 38 bytes. The expression engine instruction stream requires 66 bytes and 1 byte per invocation; this is clearly more than ARM. However, the expressions are only defined once in the instruction stream[3]. To *invoke* them requires 1 byte only. Further usage of an expression requires a single byte rather than having to be repeated at each use in the instruction stream. If the size of an expression is adjusted to assume 16 registers rather than 256 bytes of memory, 8 fewer bytes are needed per expression definition. This would be a total of 48 bytes for all of the expression definitions and a total of 48 + 6 = 54 bytes including invocation references. This is still 16 bytes more than the ARM Thumb code but 4 bytes fewer than the ARM Thumb code using global variables (i.e. literal pool). However, up to half of the expression engine nodes are

[3]It is likely that expression would be defined at the unit of linkage, e.g. at the object file level in Linux

not utilised in many cases. All of the nodes must be configured, regardless of the simplicity of the expression. In addition to this, the expression engine does not support displacement addressing. This means arrays must be emulated using pointers. This is discussed in further detail in Chapter 6.1. More research is required on designing an expression tree structure that reduces wastage. Various optimisations can be considered as well as utilising a dynamic expression tree design.

The total number of steps for the expressions using the expression engine is 18. The corresponding number of steps for ARM Thumb using PC-relative addressing is is 29. The total number of steps for ARM Thumb using local variables is 19. However, if an expression tree can be designed for higher node utilisation, this is likely to improve the results. Currently, it is assumed that each level in the expression tree requires a *step* to execute.

Chapter 5

Abstract Instructions

This chapter proposes *abstract instructions* that can be used in the architecture presented in Chapter 6, where the results will be evaluated.

5.1 Background

The C programming language by Kernighan and Ritchie [52] is used for systems and application programming and can be considered the most widely *deployed* programming language in existence. It is used to implement most modern operating systems such as Linux. It was originally designed for and implemented on the UNIX operating system on the DEC PDP-11 by Denis Ritchie [52]. It is used as the implementation language for most compilers and embedded systems. It has also been the influence of many other programming languages. Wikipedia [53] lists over 60 languages which have been influenced by it. C has also been used as an intermediate language by implementations of other languages [54] for example, C– and the early versions of C++. Other notable characteristics of C include:

- Facilitates modular and functional decomposition;
- Highly portable suitable for low level programming;
- Incorporates the C standard library which provides various application programming interfaces for common programming tasks including string manipulation, mathematics, input and output processing, memory management and operating system calls;
- Supports small and fast executables;
- Used as a subset or heavily influences many other languages.

The C language supports the imperative programming paradigm. Imperative programs specify a sequence of of statements which each change the state of the program. This is in contrast to the *declarative* programming paradigm which describes the desired results without explicitly specifying the sequence of statements that need to be performed. Procedural languages are built on top of imperative languages and add support for module and functional decomposition. Object-oriented languages build on top of procedural languages and provide support to object orientation, for example, functional inheritance, generics, objects and exception handling. It is common for these features to be provided in the form of a software *runtime* system. As such, compilers for object-oriented languages are *bootstrapped* using procedural languages. Table 5.1 illustrates this.

Object Oriented
Procedural
Imperative

Table 5.1: Programming Paradigms

It seems sensible that a processor design would in some capacity support the C programming language. One of the most important decisions in any processor design is to decide where the boundary between the software and hardware lies, Silc et al. [1]. In other words, the software instructions that the hardware decodes and interprets need to be considered. A high-level interface would place the boundary closer to the source language, in terms of its syntax and semantics. A low-level interface would place the boundary closer to a general purpose processor architecture, e.g. RISC or CISC. There are advantages and disadvantages to both extremes. The placement of this hardware/software boundary dictates the *Instruction Set Architecture* of the processor.

5.2 Instruction Set Architectures

The term *Instruction Set Architecture* includes various constraints and information about the processor. The compiler and the assembler must generate machine code that is compatible with the ISA in order for the program to execute as intended. *ISA* is an umbrella term and usually includes information about the following:

- Instruction format
- Register set
- Memory model

- Addressing modes

ISAs can be classified in a number of ways. For example, Hennessy and Patterson [8] consider characteristics such as the type of internal storage of the processor (e.g. registers or stack). There are two classes of register architectures which are *register-memory* and *load-store*. Hennessy and Patterson also describe a third class which keeps all operands in memory; this is called a *memory-memory* architecture. An example of a *register-memory* architecture is the 8086 CISC and an example of the *load-store* architecture is the ARM RISC. An example of a *memory-memory* architecture is Tiny, Audsley and Ward [11].

Another common way of classifying the ISA is the level of abstraction of the instructions. Table 5.2 illustrates the common types of ISAs organised as an abstraction hierarchy.

ISA	**Example**
Application Specific	
High Level Language Specific	Tiny
Language Specific Virtual Machine	JBC, Forth, p-code
CPU Architecture	8086, ARM

Table 5.2: ISAs

At the highest level of abstraction the ISA supports specific applications or algorithms. The benefit of this ISA is that the instruction stream can be very compact for a given application. The processor is not burdened with the overhead of interpreting general purpose instructions which implement application or algorithmic behaviour. However, a severe disadvantage is that the processor is not general enough for widespread use. A common compromise is to allow a general purpose processor to be augmented with abstract, application specific instructions. This type of processor is called an *Application Specific Instruction Processor (ASIP)*. The new instructions can be implemented by way of FPGA or *microcode*. An example is Xtensa [55]

The next type of ISA is High Level Language Specific. This type of ISA means that the processor can interpret a direct representation of the source language. In other words, there is a one-to-one mapping between the source language constructs and the processor instructions. This means it is not necessary to interpret low-level or fine-grained instructions which implement high-level programming abstractions and notations. If an instruction has a closer semantic meaning with the source program, then fewer instructions will be required to represent it. It is also likely to shorten the instruction stream for a given program which should reduce the burden on the memory system when executing a program. However, the disadvantage is that the processor is coupled to a single source language. This is likely to limit its commercial adoption and applicability.

It is even possible for the processor to interpret the source language directly rather than an equivalent representation of it. Machines that provide direct support for high-level languages in this way were proposed in the 1960s and are by no means novel. Chu and Cannot [4] illustrate a taxonomy of *High Level Language Systems* for directly executing high-level languages. This is summarised in Table 5.3 and will be briefly discussed.

High Level Language System Type	**Subtype**	**Description**
Interactive Compilation systems	1(a)	Editing, compiling, executing the entire source code
	1(b)	Editing, syntax checking, compiling, executing the entire source code
	1(c)	Editing, syntax checking each line, compiling and executing the entire source code
Interactive Interpretation Systems	2(a)	Editing, syntax checking and interpreting the entire source code
	2(b)	Editing, syntax checking and interpreting each line of source code
Interactive direct execution Systems	3	Editing, syntax checking and executing each symbol of source code

Table 5.3: Types of High Level Language Systems, taken from [4]

Type 1(a) systems are similar to traditional compiler tool chains and processors. For example, a set of C files is compiled and linked to produce a single binary which is executed on the processor. A type 1(b) system would differ in that the source code could be syntax checked during development, then only fully compiled when an executable is required. This may appear an outdated approach to development, but a modern equivalent would be the use of syntax checking in an *Integrated Development Environment (IDE))* such as *eclipse*. A Type 1(c) system would interactively syntax check each line (e.g. as with BASH shell), however, once programming is complete, the program is then compiled and executed on the processor. Type 2 systems are both unconventional and uncommon. In Type 2(a) the program is created and syntax checked as normal. However, the *source text* is then directly interpreted by the processor. Type 2(b) differs in that the processor allows the program source to be inputted interactively line by line. An analogy of this would be a BASH shell implemented in hardware. Type 3 systems process symbols. For example, these can be strings of reverse polish notation or Forth dictionaries [19].

It can be argued that Type 1(b) systems are outdated due to the availability of processor time during the development life cycle. Type 1(c) systems are not of interest, since these imply an interactive shell development environment. Type 2, whilst interesting, have had little or no

adoption in the commercial world. Type 3 systems may be of interest since their use may yield some benefits for general processor design.

Continuing on from High Level Language Specific ISAs, the next level of ISA abstraction is the Language Specific Virtual Machine ISA. This type of ISA provides an abstraction layer for a *virtual machine* for a specific language. The virtual machine is usually software based and interprets *bytecode*. Bytecode is higher level than a CPU-based ISA (discussed below), but lower level than a language ISA. It may interpret or compile the bytecode in a manner of ways discussed by Smith and Nair [32]. The motivation for this additional layer in the system stack is portability. Examples of a bytecode ISAs include Forth [19], p-code [40], and Java bytecode [24]. Ironically, there have been many attempts at implementing parts of a virtual machine in hardware in order to improve performance. The bytecode-based ISAs were developed to support the porting onto CPU-based ISAs (discussed below). This is why these ISAs resemble either a *stack*- or *register*-based design. This type of ISA is a *bottom-up design* where instructions map onto the requirements of an abstract processor model intended for implementation in software.

The most common type of ISA is the CPU Architecture ISA. These are general purpose ISAs. The processor interprets a number of instructions that implement the source level constructs. In other words, there is a one-to-many mapping between the source language constructs and the processor instructions. This has the advantage that the processor is applicable to a wide range of source languages making it more commercially viable. Example ISAs include RISC and CISC. The design philosophy of these ISAs has remained largely unchanged for the past 40 years or so.

It appears that ISA design has been either too general (e.g. RISC- and CISC-based ISAs), resulting in a many-to-one mapping between processor instructions and programming constructs, or too specific (e.g. ASIPs), which may limit the reusability of a particular instruction. A possible *compromise* is to support a set of abstract instructions representing fundamental imperative language constructs. In turn this would provide support for the C programming language and the many languages based on it and influenced by it. The ISA could be supported by an SDLP as suggested by Audsley and Ward [11]. This would be an SDLP supporting fundamental imperative programming constructs; in other words, an *SDLP for imperative languages*.

The reduction in the semantic gap between the source language and the ISA for such a processor may yield a number of benefits whilst not limiting its support for most practical languages. The possible benefits may include:

- Reduce the dependency on the memory system. If the instruction stream is more compact, the processor will require fewer instruction memory interactions for a given program.

- Reduce the frequency of external memory accesses. If there is less dependency on the memory system, there should be fewer accesses to external memory.
- Reduce power consumption. Since according to Verma et al., the memory system consumes the most power [3], reducing the dependency should reduce overall power consumption.

5.3 Instruction Set Design

The C programming language and other languages based on it specify the rules of the syntax in the form of a *grammar*. The grammar describes the set of legal *tokens* or keywords of the language and how these can be legally structured. The actual *meaning* of the tokens and they way in which they are structured is referred to as *semantics*. The syntax of the language is processed by the *compiler front end*, the main parts being the lexical analyser and parser. The semantics of the language are derived by the *compiler back end* including the optimiser and the code generator. As with any practical programming language, the semantics of C are complex. For example, C expressions specify a number of complicated concepts such as *lvalue* and *rvalue* semantics, sequence points, operator precedence, associativity and integral type promotions. Expressions also control how operands are accessed, for example as simple variables, array elements or pointers. Such semantics must be implemented by the compiler back end when generating target assembly language or machine code for the particular target processor. It is normal for the ISA to offer support for the semantics of common programming language features. However, this support is low level. For example, a traditional ISA may provide *indirect addressing* which supports pointer semantics and *displacement addressing* which supports array semantics. The Expression Engine discussed in Chapter 4 provides support for indirect addressing and for obtaining the address of a variable, which supports pointer semantics.

The Expression Engine provides semantic support for expressions, however there is an opportunity for the SDLP to provide better support for programming control constructs such as loops and conditional statements. Any support provided for C, should also be applicable for *all* languages derived from and based on C. SDLP instructions can be used to directly support C control constructs and are illustrated in Table 5.4.

It can be seen in Table 5.4, that some SDLP instructions are marked as *Not currently supported.* These are considered out of scope for this thesis. The supported SDLP instructions are *while*, *if* and *if-else*. The remaining SDLP instructions should be considered for future work.

Some common C constructs, for example *do* and *for*, have been omitted from Table 5.4. This is

C Control Construct	SDLP
if	if
if else	ifelse
while	while
function call and return	Not currently supported
switch, case, default	Not currently supported
break, continue	Not currently supported
goto, label	Not currently supported
sizeof	Not currently supported

Table 5.4: C keywords mapping onto SDLP Instructions

because some C constructs can be simply re-written or transformed using the supported keywords. Table 5.5 shows how a *for* loop can be rewritten as a *while* loop, obviating the need for an SDLP *for* instruction.

For loop	**Equivalent while loop**

for(**int** i=0; i < 10; i++)
{
  <statements>
}
	int i = 0;
	while(i++ < 10)
	{
	  <statements>
	}

Table 5.5: Rewriting a for loop using while

Table 5.6 illustrates various other C programming constructs and whether or not support is provided by the SDLP.

C Language Construct	**SDLP Support**
Scalar types, e.g. char, int	Expression engine (control bits RP0-RP5, D) provides direct addressing which supports scalar type access
Aggregate types, e.g. struct, union, enum	Expression engine (control bits RP0-RP5, D) provides indirect addressing which supports accessing aggregate types
Function calls and return	Not currently supported
Arrays	Not currently supported. Would require assembler modification so that expression engine supports displacement addressing.
Pointers	Expression engine (control bits RP0-RP5, D) provides indirect addressing and address of operator which supports pointer semantics
static, volatile, register, const, extern, typedef	Compiler constructs, not supported directly by SDLP

Table 5.6: C Language Constructs and SDLP Support

Listings 5.1, 5.2, and 5.3 illustrate the Backus-Naur form for the supported SDLP instructions and how they are described for the Assembler. For all of these, *expression_id* refers to an expression as defined in the previous chapter. A *statement* is defined as either an instruction or an *expression_id*.

```
while <label> <expression_id> {, <expression_id>}
  <statement> {<statement>}
<label>:
```

Listing 5.1: BNF for while instruction

In Listing 5.1, the *label* defines the point in the program following the *while*. In C, this would be the next statement after the closing brace of a *while* block. *expression_id* refers to an expression defined elsewhere in the binary. Multiple expression ids can be specified for complex expressions. Expressions can be chained by specifying the *lvalue* output of a preceding expression as an *rvalue* input to the next expression. If the result of the last expression evaluation is zero, then the program control branches to *label*. Otherwise, control passes to the first statement in the *while* block.

Listing 5.2 specifies an *if* statement that works in a similar way to the *while*.

```
if <label> <expression_id> {, <expression_id>}
  <statement> {<statement>}
<label>:
```

Listing 5.2: BNF for if instruction

Listing 5.3 specifies an *if-else* statement. The *label_0* defines the start of the *else* block. In C, this would be the statement after the opening brace of the *else* block. *label_1* defines the point in the program after the *else* block. In C, this would be the statement after the closing brace of the *else* block. If the result of the last expression evaluation is zero, then the program control branches to *label_0*, which is the *else* block. Otherwise, control passes to the first statement in the *if* block. The *if* block and the *else* block must both contain at least one statement each. When the last statement of the *if* block has been executed, program control jumps to *label_1*.

```
ifelse <label_0> <label_1> <expression_id> {, <expression_id>}
  <statement> {<statement>}
<label_0>:
  <statement> {<statement>}
<label_1>:
```

Listing 5.3: BNF for ifelse instruction

5.4 Instruction Encoding

Figures 5.1, 5.2 and 5.3 illustrate the encoding for the SDLP abstract instructions. The field of each instruction and the range of legal values is shown[1].

Field	opcode	label	1 .. n expression_id's	1 .. n statements
Value	0xFF	0x00 - 0xFF	0x00 - 0xFF	0x00 - 0xFF

Figure 5.1: SDLP Encoding - while

[1]Currently the assembler supports 4 expression Ids for each instruction. However, this could be increased for *while* and *if*.

Field	opcode	label	1 .. n expression_id's	1 .. n statements
Value	0xFE	0x00 - 0xFF	0x00 - 0xFF	0x00 - 0xFF

Figure 5.2: SDLP Encoding - if

Field	opcode	label_0	label_1	1 .. n expression_id's	1 .. n statements
Value	0xFD	0x00 - 0xFF	0x00 - 0xFF	0x00 - 0xFF	0x00 - 0xFF

Figure 5.3: SDLP Encoding - if else

5.5 Example Instruction Encoding

Where an ISA may provide low-level support for programming language semantics as discussed, they typically offer very primitive support for control flow. Listing 5.4 illustrates a portion of an 8086 hand-written assembly language program implementing a simple linear search. Only the constructs implementing the *while* loop are shown. It can be noted that the 8086 ISA support for implementing a C *while* loop is basic. It is constructed using a conditional an unconditional *jump* with an explicitly managed loop counter. Listing 5.5 illustrates the corresponding disassembly[2]. It can be seen that the 8086 disassembly is a faithful representation of the 8086 source code. A total of 21 bytes are required to represent the *while* loop; this can be otherwise referred to as the instruction stream length.

[2]The disassembly was generated using *objdump -D -S*. The version of GCC was 4.8.4. The version of Binutils was 2.24.

```
whileLoop:
        // Check if we have compared all elements.
        mov     count, \%eax
        cmp     \%eax, numElements
        je      label1

        // Body of loop goes here.

        incl    count
        jmp     whileLoop
label1:
```

Listing 5.4: 8086-32 While Loop

```
0804816f <_start>:
804816f:        a1 28 a0 04 08          mov 0x804a028, \%eax
8048174:        39 05 2c a0 04 08       cmp \%eax, 0x804a02c
804817a:        74 21                   je     804819d <label1>

// Body of loop goes here.

08048195 <label2>:
8048195:        ff 05 28 a0 04 08       incl   0x804a028
804819b:        eb d2                   jmp  804816f <_start>
0804819d <label1>:
```

Listing 5.5: 8086-32 While Loop Disassembly

Listing 5.6 illustrates the equivalent of Listing 5.4 but hand written for ARM. This has been assembled for ARM Thumb and ARM-32 respectively. As with the 8086 assembly, it is constructed using a conditional and an unconditional branch with an explicitly managed loop counter. However, there is additional overhead due to the load/store architecture and PC-relative addressing for managing the global variables *count* and *numElements*. Listings 5.7 and 5.8 illustrate the corresponding disassemblies. For ARM Thumb the instruction stream length is 20 bytes. Note that the instruction stream is only 1 less than the 8086 disassembly, even though the ARM instructions are 16 bits as opposed to 32 bits for 8086. For ARM-32, the instruction stream length is 40 bytes. This is 19 bytes more than the equivalent 8086-32 byte stream. This means that the ARM 32-bit ISA requires

$$(40 - 21)/21 * 100 = 90\%$$

more instructions to represent the same program written in 32-bit 8086 code.

```
_start:
        ldr     r0, =numElements
        ldr     r0, [r0]
        ldr     r1, =count
        ldr     r4, [r1]
        cmp     r0, r4
        ble     label1

        // Body of loop goes here.

label2:
        ldr     r0, =count
        add     r4, r4, #1
        str     r4, [r0]
        b       _start
label1:
```

Listing 5.6: ARM Thumb while loop

```
8178 <_start>:
8178:   480a        ldr     r0, [pc, #40]   ; (81a4 <label1+0x8>)
817a:   6800        ldr     r0, [r0, #0]
817c:   490a        ldr     r1, [pc, #40]   ; (81a8 <label1+0xc>)
817e:   680c        ldr     r4, [r1, #0]
8180:   42a0        cmp     r0, r4
8182:   dd0b        ble.n   819c <label1>

// Body of loop goes here.

8194:   4804        ldr     r0, [pc, #16]   ; (81a8 <label1+0xc>)
8196:   3401        adds    r4, #1
8198:   6004        str     r4, [r0, #0]
819a:   e7ed        b.n     8178 <_start>
819c <label1>:
```

Listing 5.7: ARM Thumb While Loop Disassembly

```
8178 <_start>:
8178:    e59f004c            ldr     r0, [pc, #76]   ; 81cc <label1+0xc>
817c:    e5900000            ldr     r0, [r0]
8180:    e59f1048            ldr     r1, [pc, #72]   ; 81d0 <label1+0x10>
8184:    e5914000            ldr     r4, [r1]
8188:    e1500004            cmp     r0, r4
818c:    da00000b            ble     81c0 <label1>

// Body of loop goes here.

81b0 <label2>:
81b0:    e59f0018            ldr     r0, [pc, #24]   ; 81d0 <label1+0x10>
81b4:    e2844001            add     r4, r4, #1
81b8:    e5804000            str     r4, [r0]
81bc:    eaffffed            b       8178 <_start>
81c0 <label1>:
```

Listing 5.8: ARM-32 While Loop Disassembly

The above code is for hand-written assembly language. To satisfy curiosity, a small C program was written to implement the equivalent *while* loop and compiled (unoptimised) for ARM Thumb. Listing 5.9 illustrates the interesting portions of the disassembly. The GCC compiler has transformed the code in a way that no longer resembles the original C program. Therefore, the original C statements have been removed from the output for the purposes of clarity. The instruction stream length is 48 bytes as opposed to 20 bytes for the hand-written equivalent.

```
81a2:    e00a          b.n                81ba <main+0x22>
81a4:    f241 0310     movw    r3, #4112   ; 0x1010 // count
81a8:    f2c0 0301     movt               r3, #1
81ac:    681b          ldr                r3, [r3, #0]
81ae:    1c5a          adds               r2, r3, #1
81b0:    f241 0310     movw    r3, #4112   ; 0x1010 // count
81b4:    f2c0 0301     movt               r3, #1
81b8:    601a          str                r2, [r3, #0]
81ba:    f241 0310     movw    r3, #4112   ; 0x1010 // count
81be:    f2c0 0301     movt               r3, #1
81c2:    681a          ldr                r2, [r3, #0]
81c4:    f241 030c     movw    r3, #4108   ; 0x100c // numElements
81c8:    f2c0 0301     movt               r3, #1
81cc:    681b          ldr                r3, [r3, #0]
81ce:    429a          cmp                r2, r3
81d0:    dbe8          blt.n   81a4 <main+0xc>
```

Listing 5.9: C to ARM Thumb While Loop Disassembly

Listing 5.10 illustrates the disassembly for the same program, but compiled with *-Os* for space saving optimisation. The result is not dissimilar to the hand-written assembly; the instruction stream length is 20 bytes. This confirms that the hand-written assembly language program is a

reasonable implementation for comparison.

```
while(count < numElements)
8198:    4b05    ldr    r3, [pc, #20]   ; (81b0 <main+0x18>)      // count
819a:    4806    ldr    r0, [pc, #24]   ; (81b4 <main+0x1c>)      //
    numElements
819c:    6819    ldr    r1, [r3, #0]
819e:    6802    ldr    r2, [r0, #0]
81a0:    4291    cmp    r1, r2
81a2:    da03    bge.n  81ac <main+0x14>
{
count++;
81a4:    681a    ldr     r2, [r3, #0]      // count
81a6:    3201    adds  r2, #1
81a8:    601a    str     r2, [r3, #0]
81aa:    e7f7     b.n    819c <main+0x4>
}
81ac:
```

Listing 5.10: C to ARM Thumb While Loop Disassembly (optimised for size -Os)

Listing 5.11 illustrates the equivalent SDLP *while* loop program. This was disassembled and the results are shown in Tables 5.7, 5.8 and 5.9.

```
.data
struct data
{
  uint32_t count = 0;
  uint32_t numElements = 10;
};

.bss
struct bss
{
  uint32_t ignore;
};

.tree
expr_UNUSED: *, +, B_AND, +, *, 7, 7, 7, 7, 7, 7, 1, 1, 0, 255, 1, 0, 7,
    ignore

expr_count_less_than_numElement: *, +, <, +, *, 0, 7, 7, 0, 7, 7, count, 1,
    0, numElements, 1, 0, 7, ignore

expr_inc_count: *, +, B_AND, +, *, 0, 7, 7, 7, 7, 7, count, 1, 1, 255, 1, 0,
     0, count

.text
while label1 expr_count_less_than_numElement
        expr_inc_count
label1:
```

Listing 5.11: SDLP While Loop

Offset	**Value (hex)**
0x0000	20 00 00 00 28 00 00 00 2C 00 00 00 50 00 00 00
0x0010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0020	00 00 00 00 0A 00 00 00 00 00 00 20 FF FF C3
0x0030	01 01 00 FF 01 00 47 08 00 F8 F1 C3 00 01 00 04
0x0040	01 00 47 08 20 F8 FF C3 00 01 01 FF 01 00 40 00
0x0050	FF 05 01 00 02 00

Table 5.7: SDLP While Loop Disassembly - Memory Dump

Segment Offset Table	**Offset**	**Value**
Data Segment Offset	0x0	0x20
BSS Segment Offset	0x4	0x28
Tree Segment Offset	0x8	0x2C
Text Segment Offset	0xC	0x50

Table 5.8: SDLP While Loop Disassembly - Segment Offset Table

Text Segment	**Offset**	**Value**
while instruction	0x50	0xFF
while label	0x51	0x5
expr_ref_1	0x52	0x1
while expression terminator	0x53	0x0
expr_ref_2	0x54	0x2
while instruction terminator	0x55	0x0

Table 5.9: SDLP While Loop Disassembly - Text Segment

It can be seen that the instruction stream length for the *while* loop is 6 bytes. However, the expressions for *expr_count_less_than_num_element* and *expr_inc_count* are not counted in these results. In ARM and 8086 assembly language, these expressions are *inlined* with instruction stream. In the SDLP, they are defined *outside* of the instruction stream, in a dedicated segment. They are then *invoked* when necessary. The size of an expression is 11 bytes. Therefore, if the calculations are modified to assume that the expressions are inlined with the instruction stream, the instruction stream length becomes 28 bytes. This is 8 bytes more than the ARM Thumb code. The reason for this is that the SDLP needs to configure 5 nodes simply to compare 2 variables and another 5 nodes to increment a variable; the expression engine is being significantly under utilised in its current configuration. In addition, the the SDLP Expression Tree uses 8 bits for operand addressing, whereas ARM uses 4 bits.

5.6 Summary

An ISA specifies the interface between a processor and the instructions it interprets. It includes details of the instruction format, internal registers, memory model and how memory is addressed. This chapter has focused mainly on the instruction format of the ISA. The ISA can be placed at various levels of abstraction, for example at the application level, domain level, language level or lower. Some early ISAs have even been placed at the source language level, however, these are regarded as research ISAs.

A High Level Language Specific ISA are uncommon and provide direct support for source level constructs. Language Specific Virtual Machine ISAs provide support for constructs of a specific language but are regarded *bottom-up*. As such there is still a one-to-many mapping between source language constructs and machine instructions. Whilst the instructions may be different from the source language constructs, they are still designed to support the general constructs and semantics of the language. CPU Architecture ISAs can be considered low-level and are the most common. The benefit of having a low-level ISA in the traditional sense is that the processor can be used for many domains. Typically, there is a one-to-many mapping between the source language constructs and the processor instructions. Syntax Directed Language Specific Processors may be considered both Language Specific Processors (LSP) and High-Level Language Computer Architectures (HLLCA), however, the architecture is defined based on the keywords of the language. Processors and ISAs at this level appear to have been overlooked.

It has been suggested that the only way that performance can be increased and power consump-

tion reduced is via *domain specific architectures*, Hennessy and Patterson [2, ch.7]. The general idea is that processors implementing domain specific algorithms and tasks would be used in conjunction with general purpose processors. However, there may still be opportunities for improving performance and reducing power consumption by considering the general purpose ISA. Current CISC and RISC ISAs have remained relatively unchanged for the past 40 years and there is a general misconception that they are already optimum and cannot be improved further.

C is the most widely deployed programming language in existence and is used as a basis for many other languages. It is therefore reasonable to support the most common C language constructs. By supporting the fundamental C programming constructs, the instructions should be applicable to most, if not all imperative languages. Constructs that are not currently supported can be rewritten and therefore expressed in alternative ways. The various addressing modes are supported by the expression engine and more abstract constructs are supported directly by the compiler.

This chapter presented an ISA for the SDLP that supports C-based imperative languages. Whilst not all C constructs are directly supported by the ISA, they can be added as further work is carried out developing the processor. Although the ISA can support various programming constructs, some constructs are covered by the expression engine, compiler and runtime system.

Chapter 6

An SDLP Architecture and Simulator

Chapters 4 and 5 described an Expression Engine and a set of Abstract Instructions which are fundamental features of the SDLP. Before describing an overarching SDLP architecture, some initial comparisons will be considered focusing on the memory system counts, e.g the number of *loads* and *stores*.

Hennessy and Patterson [8] classify ISAs using a number of characteristics, e.g. the type of internal storage such as registers or stack. There are two classes of register architectures which are *register-memory* and *load-store*. Examples of these architectures include 8086 and ARM respectively. Hennessy and Patterson also describe a third class which keeps all operands in memory; this is called a *memory-memory* architecture. This chapter describes such a *memory-memory* architecture and software simulator for the SDLP.

Finally, the results of various benchmarks are presented and compared against equivalent ARM code.

6.1 Initial Comparisons

After adding support to the SDLP for expressions and abstract instructions, it is possible to compare various attributes with RISC and CISC architectures. A benchmark suite is typically used for this purpose, but since benchmarks are usually written in C to enable cross platform comparisons, this would require a C compiler for the SDLP. At such an early stage in the development of the

SDLP, a simple assembly language benchmark can be used to assess the initial potential of the processor. Assembly language programs were written (unoptimised) for ARM, 8086 and the SDLP which implement a *linear search* for the largest integer in an array. Listing 6.1, 6.2 and 6.3 are the listings for each of the programs.

```
        .data
a0:
        .long 6,5,2,7,8,1,9,0,4,3
count:
        .long 0
numElements:
        .long 10
biggest:
        .long 0
        .globl _start
        .text
_start:

whileLoop:
        // Check if we have compared all elements.
        mov     count, %eax
        cmp     %eax, numElements
        je      label1

        // Multiply the 'count' by 4 to obtain the offset into the
        // array.
        imull   $4, %eax

        mov     biggest, %ebx
        mov     a0(%eax), %ecx
        cmp     %ebx, %ecx
        jl      label2
        mov     %ecx, biggest

label2:
        incl    count
        jmp     whileLoop

label1:
        // Since we have compiled without the C runtime, we can't just
        // return from main, as there is no runtime for us to return back
        // to.  We have to call the Linux kernel's exit() syscall with
        // our exit code stored on the stack.
        push    $0
        mov     $1, %eax
        int     $0x80
```

Listing 6.1: 8086 Linear Search

```
        .data
a0:
        .long 6,5,2,7,8,1,9,0,4,3
count:
        .long 0
numElements:
        .long 10
biggest:
        .long 0
        .globl _start
        .text
_start:
        // Check if we have compared all elements.
        ldr     r0, =numElements
        ldr     r0, [r0]
        ldr     r1, =count
        ldr     r4, [r1]
        cmp     r0, r4
        ble     label1

        // Multiply the 'count' by 4 to obtain the offset into the
        // array.
        lsl     r3, r4, #2

        ldr     r0, =biggest
        ldr     r1, [r0]
        ldr     r2, =a0
        ldr     r2, [r2, r3]
        cmp     r2, r1
        blt     label2
        str     r2, [r0]
label2:
        ldr     r0, =count
        add     r4, r4, #1
        str     r4, [r0]
        b       _start

label1:
        // Since we have compiled without the C runtime, we can't just
        // return from main, as there is no runtime for us to return back
        // to.  We have to call the Linux kernel's exit() syscall with
        // our exit code stored on the stack.
        mov     r0, #0
        mov     r7, #1
        svc     0
```

Listing 6.2: ARM Linear Search

```
.data
struct data
{
  // Assembler does not currently support arrays, so the following is used
      to represent uint32_t array[10];
  uint32_t a0 = 6;
  uint32_t a1 = 5;
  uint32_t a2 = 2;
  uint32_t a3 = 7;
  uint32_t a4 = 8;
  uint32_t a5 = 1;
  uint32_t a6 = 9;
  uint32_t a7 = 0;
  uint32_t a8 = 4;
  uint32_t a9 = 3;

  uint32_t count = 0;
  uint32_t numElements = 10;
  uint32_t ptr = 0;
  uint32_t biggest = 0;
};

.bss
struct bss
{
  uint32_t ignore;
};

.tree
expr_UNUSED: *, +, B_AND, +, *, 7, 7, 7, 7, 7, 7, 1, 1, 0, 255, 1, 0, 7,
    ignore

expr_assign_ptr_addr_of_a0: *, +, B_AND, +, *, 1, 7, 7, 7, 7, 7, a0, 1, 0,
    255, 1, 0, 0, ptr

expr_count_less_than_numElement: *, +, <, +, *, 0, 7, 7, 0, 7, 7, count, 1,
    0, numElements, 1, 0, 7, ignore

expr_inc_count: *, +, B_AND, +, *, 0, 7, 7, 7, 7, 7, count, 1, 1, 255, 1, 0,
     0, count

expr_element_greater_than_biggest: *, +, >, +, *, 2, 7, 7, 0, 7, 7, ptr, 1,
    0, biggest, 1, 0, 7, ignore

expr_assign_element_to_biggest: *, +, B_AND, +, *, 2, 7, 7, 7, 7, 7, ptr, 1,
     0, 255, 1, 0, 0, biggest

expr_inc_ptr: *, +, B_AND, +, *, 0, 7, 7, 7, 7, 7, ptr, 1, 4, 255, 1, 0, 0,
    ptr

.text
expr_assign_ptr_addr_of_a0
```

```
while label1 expr_count_less_than_numElement
  if label2 expr_element_greater_than_biggest
    expr_assign_element_to_biggest
  label2:
  expr_inc_ptr
  expr_inc_count
label1:
```

Listing 6.3: SDLP Linear Search

Information was obtained via static analysis of the programs. This was simply a case of manually calculating the frequency of various aspects such as load, stores and the number of instructions (not bytes required to represent them). Additionally, dynamic analysis was carried out for ARM and 8086 programs using *GDB* scripts[1]. Dynamic analysis was completed for the SDLP using the simulator. The results of the static and dynamic analysis were compared to ensure that there were no discrepancies. This was done to provide confidence when simulating more complex benchmarks. Table 6.1 illustrates the information that was gathered and the results. Some of this information was specific to the SDLP because of the way in which it differs from traditional processor design.

	ARM	**8086**	**SDLP**
Loads	94	52	80
Stores	14	14	25
Instructions	180	107	11
Instruction Memory Reads	n/a	n/a	89
Literals	n/a	n/a	209
Expression Id's	n/a	n/a	46
Expression Terminators	n/a	n/a	36
Program Bytes	44	33	85

Table 6.1: Comparison Results for Linear Search

It can be observed that ARM incurs the highest dependency on the memory subsystem. The main culprit is the heavy data load dependency; in order to calculate, compare or modify a variable, it must first be loaded. 8086 requires the fewest loads. It can be observed that the number of stores for both ARM and 8086 are equal. This makes sense, as the number of stores can be considered a function of the algorithm.

[1]The version of GCC for 8086 was 4.8.4. The version of Binutils for 8086 was 2.24. The version of GCC for ARM was 4.7.3. The version of Binutils for ARM was 2.24.

When comparing the architectures, the number of data loads is of interest. The number of data loads for the SDLP is fewer than ARM but greater than 8086. There are a number of reasons for this. One of the reasons why ARM requires more data loads is because it uses PC-relative addressing in order to read global variables. The compiler must generate instructions to first load the address of a literal pool into a register. This is relative to the current PC which points to the address of the next instruction but one. It then uses register indirect addressing to load the contents of the memory pointed to by the register into a second register[2]. This happens regardless of optimisation flags used during compilation. The reason that the ARM ISA requires this is because of the uniform size of instructions. They must be either 16 bits for Thumb or 32 bits for normal ARM code; literals cannot be contained within the instructions. PC-relative addressing can have a significant impact on the number of accesses to the memory system. However, it does not increase the number of instructions executed.

The SDLP requires 3 loads each time *expr_element_greater_than_biggest* is executed, compared to the equivalent code for 8086 which requires 2 loads each time it is executed.
expr_element_greater_than_biggest requires 1 load for *biggest*, 1 load for the address of *element* and 1 load to dereference *element*. The equivalent code for 8086 requires 1 load for *biggest* and 1 load to obtain the current array element using displacement addressing via mov a0(%eax), %ecx. 8086 is able to reduce the number of loads by supporting arrays via displacement addressing where the offset of the array is a literal in the *mov* instruction. The SDLP currently has no support for array processing; they must be simulated using pointers.

The SDLP requires 2 loads for *expr_assign_element_to_biggest*. The 8086 does not require these loads, since the element value is already stored in a register. The SDLP requires an additional load as part of the *expr_inc_ptr*. The 8086 does not require this since it supports displacement addressing for array processing and hence does not need to simulate arrays using pointers.

Another metric of specific interest is the number of stores across the architectures. The SDLP has a higher number of stores than both ARM and 8086 respectively; it requires 25 stores whereas the ARM and 8086 each require 14. The reasons for this are explained below.

The SDLP requires a store for *expr_assign_ptr_address_a0*. It is executed once prior to the *while* loop to assign the address of the array to the pointer. This is necessary since the SDLP does not

[2] An example of PC-relative addressing is:
ldr ro, =0xDEADBEEF
bx lr
which translates into:
0: ldr ro[pc, #4]
4: bx lr
8: .word 0xDEADBEEF

currently support arrays. The 8086 does not require this since *a0* is an immediate/literal offset within the *mov* instruction.

The SDLP requires an additional store as part of *expr_inc_ptr* each time it is executed. It is executed 10 times inside the *while* loop. Both 8086 and ARM support displacement addressing for array processing. They do not need to maintain a pointer for mimicking arrays, so they do not incur this overhead.

The number of overall bytes (program bytes) required for representing the programs is of interest. ARM requires more than 8086, but the SDLP requires double that of ARM. The culprit of this is the verbosity of the SDLP expressions.

The number of instructions for the three architectures can also be compared. There are large variances in the number of instructions required to implement the algorithm. ARM requires the most instructions; 60% more than 8086. The SDLP appears to require a fraction. The main reason for these differences is the varying levels of abstraction between the ISAs. The ARM ISA has the greatest semantic gap between the machine code and the algorithm. The 8086 has a smaller gap. The SDLP being a syntax-directed, empirical language processor has a smaller gap still. However, the results are not as conclusive as they first appear. The instruction counts for the SDLP are for the opcodes only; they do not include the additional overhead that the instructions incur. Each processor may incur architecture specific overheads. For example, on 8086 a *mov* opcode utilising displacement addressing can be up to 7 bytes wide if a 32-bit register and a 32-bit displacement are specified. For the SDLP, a program will incur overheads such as:

1. Labels (immediate values for relative jumps)
2. Expression Ids for control constructs and for other expressions
3. Null terminators for all expression Ids and instructions

Whilst all three architectures are likely to require immediate values for relative jumps, only the SDLP requires expression Ids and null terminators; since these are a unique feature of the architecture. The equivalent of these for ARM and 8086 are the instructions that implement conditional expressions and the instructions to implement expressions within code blocks. The total overhead for the SDLP for comparing with instruction counts for 8086 and ARM is:

$$instructions + expressionsIds + nullterminators = 93$$

This is less than the instruction count for 8086. If the null terminators can be encoded in the expression Ids, this would reduce further to:

$$instructions + expressionIds = 57$$

Comparing *loads*, *stores* and *instructions* for three differing architectures is challenging because of the subjective nature of the comparisons. It is always possible to argue exceptions and reasons why comparisons may be unfair or biased. However, in order that improvements to processor architecture can be made, it is necessary to make such comparisons whilst taking into account possible reasons. It is surprising that the 8086 processor has such low demands on the memory system compared to ARM since many assume that ARM is designed with low power demands in mind. There may be many reasons for this, however power savings may be due to factors other than the ISA.

The comparisons made are based on abstract notions such as *data loads*, *stores* and *instructions* as opposed to the byte counts for each. Nevertheless it is possible to see that the SDLP may have potential in reducing demands on the memory subsystem. It is important that future generations of the processor support arrays in the form of immediate offsets and displacement addressing. It is also necessary to understand the implications of implementing the processor logic at the hardware level. Currently the analysis has been limited to simulations at the basic block level.

An architecture for the SDLP will now follow which will enable benchmarking comparisons to be made focusing on the cycle counts for a set of benchmarks.

6.2 SDLP Architecture and Behavioural Description

Figure 6.1 illustrates the block diagram for the SDLP.

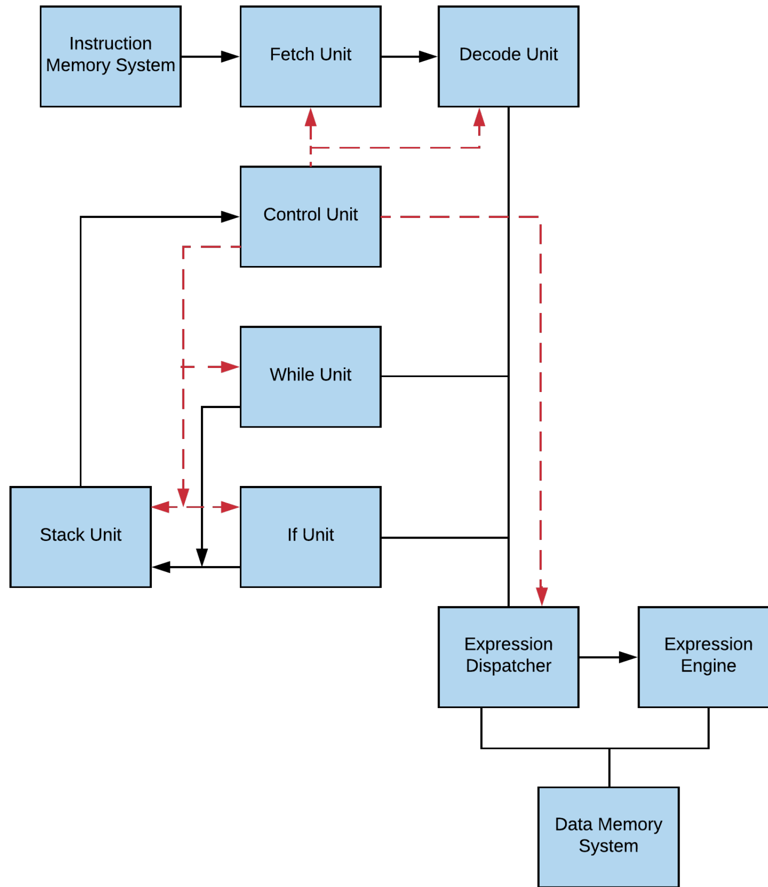

Figure 6.1: SDLP Block Diagram

As with a traditional microprocessor, the *Control Unit* is responsible for managing and coordinating the rest of the processor. The dashed lines show connections between the Control Unit and various units that it drives, i.e. the control bus. The solid lines indicate instruction and data flow between the various units, i.e. the instruction and data busses. The *Stack Unit* is required for nesting of *while* loops and control flow of *if-else* statements.

The *Instruction Memory System* represents the interface to the instruction stream, which would typically be an instruction cache.

The *Fetch Unit* reads the next instruction sequence or expression sequence. It fetches a byte

stream for the longest instruction, allowing for the maximum number of supported expression references for the condition evaluation. This is an *if-else* instruction since it requires 1 byte for the opcode, 1 byte for the *else* block start offset, 1 byte for the *else* block end offset and a null byte to terminate the expression references for the condition sequence. A maximum of 4 expression references are supported. Therefore, the Fetch Unit fetches a total of eight bytes. This is enough bytes to represent all of the SDLP instructions or a continuous sequence of up to 8 expression references.

It is assumed that the Instruction Memory System has an Instruction Cache and the Data Memory System has a Data Cache. The memory system therefore assumes a Harvard architecture. Each cache is assumed to have an 8-byte line width, which can be supported by standard DDR[3]. This means that each SDLP expression will require two memory transfers to fetch (an SDLP expression is 11 bytes in length). The six *rvalues* for an expression will require three transfers to load their values (there are six rvalues, four bytes each in length). An *if-else* is the longest SDLP instruction and requires a total of eight bytes, therefore can be fetched in a single memory transfer. However, the affects of memory transfers are not considered any further in this thesis.

6.2.1 SDLP Execution Model

Most modern processors employ a pipeline execution model in order to improve throughput. Figure 6.2 illustrates how the fetch, decode and execute phases can, in principle, overlap. However, because the execution phases of instructions and expressions may require different numbers of clock cycles, this means that the overlapping is not perfectly aligned. In these cases, the pipeline would *stall*. These are more accurately referred to as *intrinsic stalls* since they are due to internal processor resource conflicts. An example is a single memory unit that is accessed in the fetch stage where an instruction is retrieved from memory, and the memory stage where data is written and/or read from memory.

The current execution model assumes no overlapping of instructions. Therefore, the execution model is *sequential*. The benefits at this stage of development mean that the simulator is significantly easier to develop. It also means that any benchmark comparisons with other architectures are simpler since the affect and modelling of pipelining on other architectures does not need to be considered. Pipeline development is an iterative and evolutionary engineering process. Modern architectures will have had many thousands of man-years of effort dedicated to pipelining alone.

[3]Double Data Rate, Synchronous Dynamic RAM

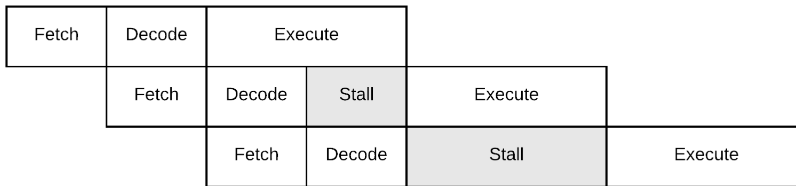

Figure 6.2: Pipeline

Listing 6.4 illustrates the pseudocode for the fetch unit. The *par* statement from Occam [56] (and more recently Handel-C [57]) is used to denote parallelism. All statements within a *par* block are executed in parallel within the same clock cycle. Any statement outside of a *par* statement requires one clock cycle to execute.

```
uint8_t *statement = readFromMemorySystemInBytes(8); // 1 clock cycle
```

Listing 6.4: Fetch Unit Pseudocode

Listing 6.5 illustrates the pseudocode for the *Decode Unit*. The *Decode Unit* takes the instruction stream or expression reference sequence from the Fetch Unit and prepares them for execution. For a *While* instruction, this consists of preparing the *While Unit* for execution. The offset of the first instruction of the *while* block is calculated. The offset of the instruction following the end of the while block is extracted as well as up to 4 expression references representing the condition sequence. A null byte denotes the end of the sequence.

For an *if* instruction, the *If Unit* is prepared for execution. The offset of the first instruction of the *if* block is calculated. The byte offset of the instruction following the end of the *if* block is extracted as well as up to 4 expression references representing the condition sequence.

Decoding an *if-else* instruction is slightly more complicated. The offset of the first instruction of the *if* block is calculated. The byte offset of the start of the *else* block is extracted. The byte offset of the instruction following the end of the *else* block is also extracted. Up to 4 expression references representing the condition sequence are extracted.

For all instructions, the decoder prepares the expression references ready for the *Expression Dispatcher*. Once decoding is complete, the Control Unit passes control to either the *While Unit*, *If Unit* or directly to the *Expression Dispatcher*.

It should be noted that the decoder cannot yet determine the instruction type. It therefore prepares for the next statement to be any instruction or expression sequence in parallel (i.e. in

a single clock cycle). This is simple for the *while* and *if* instructions, since they are uniform; they are the same length and shape. However, an *if-else* instruction is different since it has an additional *label* field. This means that the gathering of expression references starts a byte offset later. The same applies when finding the offset of the first instruction in the *if* block of an *if-else* instruction. The decoder must also attempt to gather a sequence of expression references in place of an instruction. These will be interpreted as such by the *Control Unit* if the instruction variable is not recognised. The problems associated with a non-uniform instruction shape is evident here, hence the complication during decoding.

```
par // 1 clock cycle.
{
    uint8_t instr = statement[0];

    // Needed for While instruction so we can loop back.
    uint8_t instrOffset = pc;

    // Get byte offset to next instruction after if block (for an If),
    // or start of else block (for an If-else)
    uint8_t label0 = statement[1];

    // Get byte offset of instruction following end of else block
    // (for an If-else)
    uint8_t label1 = statement[2];

    // Copies expression refs until NULL is reached.
    // Sets 'count' to number of expression refs parsed.
    uint8_t *exprRefs = parseExpressionRefs(statements[2]);
    uint8_t *ifElseExprRefs = parseExpressionRefs(statements[3]);
    uint8_t *sequenceExprRefs = parseExpressionRefs(statements[0]);

    // Search for next byte after the NULL terminating
    // condition sequence

    // Start of While or If block.
    uint8_t startOffset = findStartOffset(statements[3]);

    // Start of If block for an If-else.
    uint8_t ifElseStartOffset = findStartOffset(statements[4]);
}
```

Listing 6.5: Decode Unit Pseudocode

The following describes the common execution behaviour common to all instructions. First, the *Expression Dispatcher* iterates through the expression references representing the condition sequence. Each expression reference is an offset from the *.tree segment* and is used to obtain the expression for execution. The *Expression Dispatcher* obtains the expression from the *Data Memory*

System. Listings 6.6 and 6.7 illustrate this. The expression is then passed to the *Expression Engine* for execution. This sequential process is repeated for each expression in the condition sequence. The result of the last expression is used to determine the *decision*.

```
switch (instr)
{
    case While:
    case If:
        dispatchExpressions(exprRefs, count); // 1 clock cycle.
    break

    case If-else:
        dispatchExpressions(ifElseExprRefs, count); // 1 clock cycle.
    break

    default:
        dispatchExpressions(sequenceExprRefs, count); // 1 clock cycle.
    break;
}
```

Listing 6.6: Expression Dispatcher Psudocode

```
void dispatchExpressions(uint8_t *expressionRefs, int count)
{
    // 3 * clock cycles per expression reference.
    for(int i=0 i < count; i++)
    {
        exprEngine.evaluate(exprRefs[i]); // 3 clock cycles.
    }
}
```

Listing 6.7: Expression Dispatch Loop Pseudocode

The following describes the specific behaviour when executing each instruction. For a *while* instruction [4], if the decision is *false*, the *While Unit* will pass control to the relative address specified by the the *While label* offset. This is the first instruction following the *while* block. If the result is *true*, the offset of the *while* statement is passed to the *Stack Unit* for pushing. Execution continues from the first instruction of the *while* block and can consist of expression references and instructions. Listing 6.8 illustrates the pseudocode for the *While Unit*.

[4] The BNF for a *while* loop is *while* $< label > < expr_id > \{, < expr_id >\} < statement > \{< statement >\} < label >$:

```
// Check flags register for Zero.
// Zero equates to false, meaning that the While block is not entered.
// Non-zero equates to true, meaning the While block is entered.
if (flags.zero == 0) // 1 clock cycle.
{
    // Branch to jump over While block.
    pc = label0;  1 clock cycle.
}
else
{
    par  // 1 clock cycle.
    {
        stack.push(instrOffset);
        pc = startOffset;
    }
}
```

Listing 6.8: While Pseudocode

Eventually, a *null* byte will be hit in the instruction stream instead of an instruction or expression reference. This null byte signifies the end of the *while* block. When this happens, the relative offset is popped off the stack, and execution continues from this point. This is the address of the beginning of the *while* loop so that its condition can be reevaluated. Listing 6.9 illustrates the pseudocode.

```
pc = stack.pop(); \\ 1 clock cycle.
```

Listing 6.9: Null Pseudocode

It should be pointed out that the *Stack Unit* is a *processor stack* as opposed to a *system stack*. This means that its only purpose is to support the execution of processor instructions and not stack frames during function call, return and interrupt handling. However, since it stores information for currently executing instructions, it will require saving prior to any context switch due to an interrupt, process or thread context switch. It would obviously require restoration upon return. If the SDLP instruction set is extended to support the *break* or *continue* instructions, careful attention will be required. For example, in the case of a *break*, the stack will need to be popped without modification to the program counter. The processor will need to keep track of the nested instruction in order to correctly manage the stack.

For an *if* instruction [5], if the *decision* is *false*, the *If Unit* will pass control to the to the relative address specified by the label offset; this is the first instruction following the end of the *if* block.

[5] The BNF for an If is $if < label >< expr_id > \{, < expr_id >\} < statement > \{< statement >\} < label >$:

If the *decision* is *true*, execution continues from the first instruction of the *if* block. Listing 6.10 illustrates the pseudocode.

```
// Check flags register for Zero.
// Zero equates to false, meaning that the If block is not entered.
// Non-zero equates to true, meaning the If block is entered.
if (flags.zero == 0) // 1 clock cycle.
{
    // Jump over If block.
    pc = label0;   // 1 clock cycle.
}
else
{
    pc = startOffset;   // 1 clock cycle.
}
```

Listing 6.10: If Unit Pseudocode

For an *if-else* instruction [6], if the *decision* result is *false*, program control is passed to the relative address specified by *label_0*; this is the start offset of the *else* block. If the *decision* result is *true*, the relative address specified by *label_1* is passed to the *Stack Unit* for pushing; this is the location of the first instruction following the end of the *else* block. Program control then resumes from the start of the *if* block. Eventually, a null byte will be encountered in the instruction stream instead of an instruction or expression reference. When this happens, the relative offset is popped off the stack, and execution continues from this point. This is the offset of the instruction following the end of the *else* block. Listing 6.11 illustrates this.

```
// Check flags register for Zero.
// Zero equates to false, meaning that the Else block is entered.
// Non-zero equates to true, meaning the If block is entered.
if (flags.zero == 0) // 1 clock cycle.
{
    // Continue execution from start of Else block.
    pc = label0;   // 1 clock cycle.
}
else
{
    par // 1 clock cycle.
    {
        stack.push(label1);
        pc = ifElseStartOffset;
    }
}
```

Listing 6.11: If-else Pseudocode

[6]The BNF for an *if-else* is *ifelse* < *label_0* > < *label_1* > < *expr_id* > {, < *expr_id* >} < *statement* > {< *statement* >} < *label_0* >: < *statement* > {, < *statement* >} < *label_1* >:

In order to support the development of the SDLP, support will be required for other processor features such as *function call* and *return*. This will ultimately facilitate more complex benchmarks for architecture exploration. Whilst function calls and return are outside the scope of this thesis, support for such features would be very similar to 8086. The only difference between SDLP and 8086 function calls and return is that the SDLP could provide more abstract instructions, representing the *caller* and *callee prologues* and *epilogues*. Appendix A describes how such features could be implemented as future work for the SDLP.

6.3 SDLP Software Simulator

A significant contribution is an assembler and a software simulator for the SDLP. The purpose of these tools is to:

1. Verify that the concepts described are practical with regards to implementation;
2. Provide a reference model for further architectural exploration and experimentation;
3. Provide a reference model to aid the development of an FPGA implementation;
4. Allow various statistics to be gathered when executing benchmark code.

The assembler is written in simple object-based C++. The simulator is written accordingly in C++ as opposed to using a simulation framework such as SystemC [58]. This means that most software and hardware engineers will be able to quickly understand and modify the simulator without specialist knowledge and experience of frameworks. The execution statistics that can be gathered include the following:

- Instruction Memory Reads;
- Data Memory Reads;
- Data Memory Writes;
- Number of byte literals (constant values used in an expression);
- Number of null terminators (in instructions and condition sequences);
- Clock cycles.

It is not possible to be be completely accurate regarding clock cycles, as this will ultimately depend on the *RTL (Register Transfer Language)* implementation details. This work must be done in successive refinements and include the skills of a digital electronics engineer working alongside the processor designer. However, at this stage of development, sensible estimates for clock cycles are adequate for the purpose of architecture exploration and ascertaining the viability of the SDLP.

As discussed previously, the simulator does not currently model the pipeline behaviour. Instead the fetch, decode and execute cycles are considered sequential; there is no overlapping. Modifying the simulator for dynamic pipeline modelling should be considered for future work.

For the current sequential execution model, the pseudocode listed above is used to determine the number of clock cycles required for each fetch, decode and instruction execution. In particular, the execute phases of each instruction can be defined.

A *while* instruction requires the following number of clock cycles to execute:

- If the condition is false, 2;
- If the condition is true, 2;
- If the condition is true, a further cycle is required for the *null* processing (see Listing 6.9).

An *if* instruction requires 2 clock cycles to execute regardless of the condition outcome.

An *if-else* instruction requires the following number of clock cycles to execute:

- If the condition is false, 2;
- If the condition is true, 2;
- If the condition is true, a further cycle is required for the *null* processing (for jumping over the *else* block when the end of the *if* block has been reached, see Listing 6.9).

Instructions are just one consideration for clock cycle simulation; another is the execution of expressions. The clock cycles for expressions must account for the following:

- Cycles for *rvalue* addressing;
- Cycles for *lvalue* addressing;
- Cycles for node processing.

rvalue addressing can be:

- Literals - part of the instruction stream and already decoded;
- Address of - part of the instruction stream and already decoded;
- Variable - a read from the Data Memory System is required;
- Pointer Dereference - 2 reads from the Data Memory System are required.

lvalue addressing can be:

- Ignore - no value is written to the Data Memory System, only internal processor flags are updated;

- Variable - a write to the Data Memory System is required;
- Pointer Dereference - 1 read and 1 write to the Data Memory System are required.

Arithmetic and logical operations performed by nodes within the Expression Engine are likely to be implemented using similar techniques that traditional ALUs employ. Therefore, the clock cycle values for these operations can use the values taken from the data sheet of an existing processor. The cycles times for node processing could be taken from the data sheet for the MicroBlaze Processor Reference Guide [5]. The MicroBlaze is a *soft-core* processor, intended for use on platforms with an FPGA. Since the next step in the development of the SDLP may be an FPGA rather than an ASIC (Application Specific Integrated Circuit) implementation, the MicroBlaze appears appropriate.

Table 6.2 illustrates the cycle values for an appropriate selection of MicroBlaze instructions. These are the figures for when *area optimisation* is enabled.

Instruction	**Number of clock cycles**
ALU	
and, or, xor	1
add	1
cmp	1
bs (barrel shift)	2
mul	3
Load/Store	
imm (load immediate)	2
lw (load word)	2
Branch	
br	3
beq	3

Table 6.2: Cycle Times for MicroBlaze Soft-core Processor, adapted from [5]

If the clock cycle values for the MicroBlaze are to be used to derive values for the SDLP simulator it is important that they are reasonable and within range of what can be considered typical. To ensure this, clock cycle values for the ARM7TDMI processor were also considered alongside the MicroBlaze values. The ARM7TDMI core is a popular 32-bit embedded RISC processor for embedded systems requiring low power consumption, small size and high performance. The processor is based on the *Von Neumann* architecture and has a three-stage pipeline comprising of fetch, decode and execute. The data sheet for the ARM7TDMI [6] details the number of cycles required for different types of instructions, and these are shown in Table 6.3. However, these must

Instruction	Cycle Count	Additional
Data Processing	1S	+ 1l for SHIFT(Rs) + 1S+1N if R15 written
MSR, MRS	1S	-
LDR	1S+1N+1l	+ 1S+1N if R15 loaded
STR	2N	-
LDM	nS+1N+1l	+ 1S+1N if R15 loaded
STM	(n-1)S+2N	-
SWP	1S+2N+1l	-
B, BL	2S+1N	-
SWI	2S+1N	-
MUL, MLA	1S+ml	-
MUL	1S+ml	-
MLA	1S+(m+1)l	-
MULL	1S+(m+1)l	-
MLAL	1S+(m+2)l	-
CDP	1S+bl	-
LDC, STC	(n-1)S+2N+bl	-
MCR	1N+bl+1C	-
MRC	1S+(b+1)l+1C	-

Table 6.3: Cycle Times for ARM7TDMI, taken from [6, p.8]

be interpreted with some caution. The data sheet states that these are *the incremental number of cycles required by an instruction, rather than the total number of cycles for which the instruction uses part of the processor* [6, p.7]. Therefore, it may be assumed that the table illustrates the number of execute cycles only.

The following attempts to explain each of the variable in Table 6.3:

- n is the number of machine words transferred;
- m is 1 if bits [32:8] of the multiplier operand are all zero or all one;
- m is 2 if bits [32:16] of the multiplier operand are all zero or all one;
- m id 3 if bits [31:24] of the multiplier operand are all zero or all one;
- b is the number of cycles spent in the coprocessor busy-wait loop;
- S is a sequential memory cycle. During this cycle, the processor requests a transfer to or from an address that is either one word or one half word greater than the address used in the preceding cycle;
- N is a non-sequential memory cycle. During this cycle, the processor requests a transfer to or from an address that is unrelated to the address used in the preceding cycle;
- I is an internal memory cycle. During this cycle, the processor does not require a transfer

because it is performing an internal function and no useful prefetching can be performed at the same time;

- C is a coprocessor register transfer memory cycle.

For simplicity, constant values for S, N, I and m can be assumed. A value of 1 is used for S, N and I. A value of 3 is used for m.

With this in mind, it can be seen that a MicroBlaze takes 2 cycles to execute a load whereas the ARM7TDMI would require 3. To execute an add for the MicroBlaze requires 1 cycle which is the same as for the ARM7TDMI. The MicroBlaze requires 3 cycles to execute a multiply. The ARM7TDMI requires between 2 -5 cycles depending on the value of the multiplier. Even though the MicroBlaze is a soft-core RISC processor and the ARM7TDMI is an ASIC processor, these figures are not wildly different.

It was decided *not* to use either of the clock cycle values in Table 6.2 or Table 6.3 for reasons discussed in Section 6.4

6.4 Comparison of Simulation Measurements

The simulator is programmed to accumulate the number of clock cycles as it interprets each instruction and expression. Whilst the clock cycle values for the execution phase for the MicroBlaze and the ARM7TDMI are available in table form, these can be confusing. For example, the ARM7TDMI requires detailed knowledge of the memory cycles during execution. Even if it is possible to use a cycle accurate simulator, the results may show large differences due to factors other than the ISA for example, pipelining.

Instruction selection between architectures is also likely to introduce differences. For example, the Linear Search illustrated previously in Listing 6.2, an *lsl* (left shift) may be performed in order to multiply by a power of two; this can be executed in a single clock cycle. However, the SDLP may achieve the same result by using a multiply instruction, requiring 3 clock cycles to execute.

Compiler optimisations can also have a significant impact on clock cycle counts. For example, GCC has 7 optimisation options (-O0, -O1, -O2, -O3, -Os, -Ofast and -Og) [59]. Each selectively include a multitude of optimisation flags. An experiment was conducted to count the number of instructions executed in the GDB debugger for a matrix multiplication program compiled for ARM Thumb. This was done using a GDB user-defined function which counts the number of instructions up to a given *program counter* address. The results showed that the unoptimised program (-O0 or omitted optimisation option) took 333 instructions to execute, whereas an -O3

optimised program only required 29. GCC is clearly capable of aggressive optimisation, for example SIMD (Single Instruction Multiple Data) transformations. In this example, it impressively reduced the number of instructions executed by an order of magnitude. However, comparing aggressively optimised ARM programs with unoptimised SDLP programs is clearly unfair since for whatever optimisations are possible for ARM, there may be the equivalent and additional optimisations for the SDLP. Assuming this statement is valid, it is necessary at this stage of development to compare *like for like* by discounting optimisations. A program running on ARM should be similar in structure to the equivalent program running on the SDLP.

Such differences are inevitable when comparing implementations using different ISAs. At this stage of exploration, the interest is not in understanding the number of cycles a processor takes to perform primitive ALU-based operations. Understanding how the *ISA* can impact the clock cycle counts is of the greatest interest. Comparing metrics between architectures is fraught with potential problems since subtle assumptions or oversights can impact the results significantly. It is therefore necessary to simplify the method used for measurement, removing as many variables as possible.

The following assumptions have been made for ARM:

- A sequential execution model;
- Fetch, decode and execute phases are assumed to require 1 clock cycle each - therefore, every ARM instruction requires 3 clock cycles to complete[7];
- No compiler optimisations.

The following assumptions have been made for the SDLP:

- A sequential execution model;
- Fetch and decode phases are assumed to require 1 clock cycle each (based on the above pseudocode);
- Execute phase for instructions uses the clock cycle values taken from the above pseudocode;
- Execute phase for expression operations require one 1 clock cycle (the same as ARM ALU operations);
- Data memory reads and writes each require 1 clock cycle[8];
- Pointer dereference requires 2 clock cycles (equivalent to two loads in ARM);
- Address of operation requires 0 clock cycles (this information is already available internally);

[7]It is assumed that the execute phase for all ARM instructions requires a single clock cycle. This is to ensure that the clock cycle counts for ARM are optimistic. More pessimistic assumptions are used for the SDLP.

[8]Note that in ARM, memory accesses are done via explicit *load/store* instructions.

- Literals require 0 clock cycles (this information is already available internally);
- Up to 4 condition expressions can be fetched and decoded alongside their associated instruction since they are processed as part of the instruction;
- Stand-alone expressions require separate fetch and decode cycles [9];
- No optimisations.

In general, the total number of clock cycles required for an expression e is $r + n + l$ where r is the number of clock cycles for *rvalue* addressing, n is the number of clock cycles for node execution and l is the number of clock cycles for *lvalue* addressing.

r can be defined as $\sum_{n=1}^{6} 0 \leq r_n \leq 2$

e is defined as 3 (representing the number of *levels* in the tree)

l can be defined as $0 \leq r \leq 2$

Based on the above assumptions, the number of clock cycles for the ARM implementation of the Linear Search Algorithm in Listing 6.2, can be calculated and augmented with the initial comparison results in Table 6.1.

The clock cycle count for the ARM code is $180 * 3 = 540$, since the ARM7TDMI has a 3-stage pipeline and so this needs to be factored into the results. The clock cycle count for the SDLP obtained via the simulator is 413. This includes the clock cycle counts for the stand-alone instructions. The difference is given in Table 6.4, which summarises the differences in clock cycles, loads and stores between ARM and the SDLP for the Linear Search Program.

	ARM	**SDLP**	**Difference**	**% Improvement**
Cycles	540	413	127	23
Loads	94	81	13	14
Stores	14	25	-11	-78
Instruction Memory Reads	n/a	89	n/a	n/a
Program Bytes	44	85	-41	-51

Table 6.4: Difference between ARM and SDLP for the Linear Search Program

The SDLP is able to execute its version of the Linear Search Program using 23% fewer clock cycles than ARM. Given that the SDLP is being compared with an architecture known for low power, these initial results appear significant.

The SDLP is also able to reduce the number of interactions with the Data Memory System. Since the instruction stream length of an SDLP program is significantly shorter than the equivalent

[9] This is a pessimistic view, since groups of 8 stand-alone expressions could be fetched and decoded together. However, this would be more complicated to model in the simulator.

RISC instructions, the number of interactions with the Instruction Memory System is significantly reduced. Both these aspects are beneficial in reducing power consumption. The reasons for the increase in stores is discussed in Section 6.1.

To further validate the comparison results of the Linear Search Algorithm, a number of additional benchmarks were written. For ARM, these were written in C. The benchmarks attempt to look at the potential benefits between abstract instructions and expressions; in other words, which of these two features provide the most significant improvement. An additional benchmark was written to understand the implications of PC-relative addressing for ARM. The benchmark categories are summarised in Table 6.5.

Benchmark	**Category**
Linear Search	Expressions
Loop 1	Instructions
Loop 6	PC-relative addressing (ARM only)
Loop 2	Instructions
Checksum 1	Instructions
Checksum 2	Expressions
Checksum 3	Expressions
Matrix	Expressions

Table 6.5: Benchmark Categories

- *Linear Search* is the benchmark previously illustrated in Listings 6.2 and 6.3;
- *Loop 1* is a *while* loop which iterates 10 times. The loop counter is a local variable;
- *Loop 6* is the same *while* loop, however the the loop counter is a global variable. For ARM this means it is accessed using PC-relative addressing. There is no corresponding listing for the SDLP;
- *Loop 2* is a *while* loop which iterates 100 times. The loop counter is a local variable;
- *Checksum 1* is taken from the ARM System Developer's Guide [12];
- *Checksum 2* is a loop unrolled version, with the loop body containing 4 expressions rather than 1;
- *Checksum 3* is also unrolled but with 8 expressions in the loop body;
- *Matrix* calculates the products of a matrix pair.

The loop-based benchmarks contain minimal expressions. As such any saving would be weighted more towards abstract instructions. The loop benchmarks vary the number of iterations, to confirm that savings should not increase with the number of iterations. The affect of PC-relative addressing is shown with Loop 6. The Checksum benchmarks gradually increases the number of expressions into a program via *loop unrolling*. Again, this is to understand whether abstract instructions or expressions provide the most significant improvement. Matrix further increases the ratio of number of expressions to abstract instructions.

The following listings are for the ARM Thumb disassemblies and the associated SDLP assembly language for the benchmarks.

```
int loop(int n)
{
    8248:       b480            push    {r7}
    824a:       b085            sub     sp, #20
    824c:       af00            add     r7, sp, #0
    824e:       6078            str     r0, [r7, #4]
  int i;
  while(i<n)
    8250:       e002            b.n     8258 <loop+0x10>
  {
    i++;
    8252:       68fb            ldr     r3, [r7, #12]
    8254:       3301            adds    r3, #1
    8256:       60fb            str     r3, [r7, #12]
}

int loop(int n)
{
  int i;
  while(i<n)
    8258:       68fa            ldr     r2, [r7, #12]
    825a:       687b            ldr     r3, [r7, #4]
    825c:       429a            cmp     r2, r3
    825e:       dbf8            blt.n   8252 <loop+0xa>
  {
    i++;
  }
```

Listing 6.12: Loop 1 - ARM

```
.data
struct data
{
  uint32_t i = 0;
  uint32_t n = 10;
};

.bss

struct bss
{
  uint32_t ignore;
};

.tree
expr_UNUSED: *, +, B_AND, +, *, 7, 7, 7, 7, 7, 7, 1, 1, 0, 255, 1, 0, 7,
    ignore
expr_i_less_than_n: *, +, <, +, *, 0, 7, 7, 0, 7, 7, i, 1, 0, n, 1, 0, 7,
    ignore
expr_inc_i_by_1: *, +, B_AND, +, *, 0, 7, 7, 7, 7, 7, i, 1, 1, 255, 1, 0, 0,
     i

.text
while label1 expr_i_less_than_n
  expr_inc_i_by_1
label1:
```

Listing 6.13: Loop 1 - SDLP

```
  while(i<n)
    81a2:       e00a          b.n     81ba <main+0x22>
  {
    i++;
    81a4:       f241 0310     movw    r3, #4112       ; 0x1010
    81a8:       f2c0 0301     movt    r3, #1
    81ac:       681b          ldr     r3, [r3, #0]
    81ae:       1c5a          adds    r2, r3, #1
    81b0:       f241 0310     movw    r3, #4112       ; 0x1010
    81b4:       f2c0 0301     movt    r3, #1
    81b8:       601a          str     r2, [r3, #0]
static int i = 0;
static int n = 10;

int main(char *argv[], int argc)
{
  while(i<n)
    81ba:       f241 0310     movw    r3, #4112       ; 0x1010
    81be:       f2c0 0301     movt    r3, #1
    81c2:       681a          ldr     r2, [r3, #0]
    81c4:       f241 030c     movw    r3, #4108       ; 0x100c
    81c8:       f2c0 0301     movt    r3, #1
    81cc:       681b          ldr     r3, [r3, #0]
    81ce:       429a          cmp     r2, r3
    81d0:       dbe8          blt.n   81a4 <main+0xc>
  {
    i++;
  }
```
Listing 6.14: Loop 6 - ARM

```
  int i;
  while(i<n)
    8250:       e002          b.n     8258 <loop+0x10>
  {
    i++;
    8252:       68fb          ldr     r3, [r7, #12]
    8254:       3301          adds    r3, #1
    8256:       60fb          str     r3, [r7, #12]

int loop(int n)
{
  int i;
  while(i<n)
    8258:       68fa          ldr     r2, [r7, #12]
    825a:       687b          ldr     r3, [r7, #4]
    825c:       429a          cmp     r2, r3
    825e:       dbf8          blt.n   8252 <loop+0xa>
  {
    i++;
  }
```
Listing 6.15: Loop 2 - ARM

```
.data
struct data
{
  uint32_t i = 0;
  uint32_t n = 100;
};

.bss

struct bss
{
  uint32_t ignore;
};

.tree
expr_UNUSED: *, +, B_AND, +, *, 7, 7, 7, 7, 7, 7, 1, 1, 0, 255, 1, 0, 7,
    ignore
expr_i_less_than_n: *, +, <, +, *, 0, 7, 7, 0, 7, 7, i, 1, 0, n, 1, 0, 7,
    ignore
expr_inc_i_by_1: *, +, B_AND, +, *, 0, 7, 7, 7, 7, 7, i, 1, 1, 255, 1, 0, 0,
     i

.text
while label1 expr_i_less_than_n
  expr_inc_i_by_1
label1:
```

Listing 6.16: Loop 2 - SDLP

```
int sum = 0;
  827a:        2300           movs    r3, #0
  827c:        60fb           str     r3, [r7, #12]
do
{
  sum += *data++;
  827e:        687b           ldr     r3, [r7, #4]
  8280:        1d1a           adds    r2, r3, #4
  8282:        607a           str     r2, [r7, #4]
  8284:        681b           ldr     r3, [r3, #0]
  8286:        68fa           ldr     r2, [r7, #12]
  8288:        4413           add     r3, r2
  828a:        60fb           str     r3, [r7, #12]
  n--;
  828c:        683b           ldr     r3, [r7, #0]
  828e:        3b01           subs    r3, #1
  8290:        603b           str     r3, [r7, #0]
} while (n != 0);
  8292:        683b           ldr     r3, [r7, #0]
  8294:        2b00           cmp     r3, #0
  8296:        d1f2           bne.n   827e <checksum+0xe>
```

Listing 6.17: Checksum 1 - ARM

```
.data
struct data
{
  // Assembler does not currently support arrays, so the following is used
      to represent uint32_t array[64];
  uint32_t a0 = 0;
  uint32_t a1 = 1;
  uint32_t a2 = 2;
  uint32_t a3 = 3;
  uint32_t a4 = 4;
  uint32_t a5 = 5;
  uint32_t a6 = 6;
  uint32_t a7 = 7;
  uint32_t a8 = 8;
  uint32_t a9 = 9;
  uint32_t a10 = 10;
  uint32_t a11 = 11;
  uint32_t a12 = 12;
  uint32_t a13 = 13;
  uint32_t a14 = 14;
  uint32_t a15 = 15;
  uint32_t count = 0;
  uint32_t numElements = 16;
  uint32_t ptr = 0;
  uint32_t sum = 0;
};

.bss

struct bss
{
  uint32_t ignore;
};

.tree
expr_UNUSED: *, +, B_AND, +, *, 7, 7, 7, 7, 7, 7, 1, 1, 0, 255, 1, 0, 7,
    ignore
expr_assign_ptr_addr_of_data: *, +, B_AND, +, *, 1, 7, 7, 7, 7, 7, a0, 1, 0,
     255, 1, 0, 0, ptr
expr_count_less_than_numElements: *, +, <, +, *, 0, 7, 7, 0, 7, 7, count, 1,
     0, numElements, 1, 0, 7, ignore
expr_inc_count: *, +, B_AND, +, *, 0, 7, 7, 7, 7, 7, count, 1, 1, 255, 1, 0,
     0, count
expr_inc_ptr: *, +, B_AND, +, *, 0, 7, 7, 7, 7, 7, ptr, 1, 4, 255, 1, 0, 0,
    ptr
expr_add_element_to_sum: *, +, B_AND, +, *, 2, 7, 0, 7, 7, 7, ptr, 1, sum,
    255, 1, 0, 0, sum

.text
expr_assign_ptr_addr_of_data

while label1 expr_count_less_than_numElements
  expr_add_element_to_sum
```

```
  expr_inc_ptr
  expr_inc_count
label1:
```

Listing 6.18: Checksum 1 - SDLP

```
int sum = 0;
  827a:        2300          movs    r3, #0
  827c:        60fb          str     r3, [r7, #12]
do
{
  sum += *data++;
  827e:        687b          ldr     r3, [r7, #4]
  8280:        1d1a          adds    r2, r3, #4
  8282:        607a          str     r2, [r7, #4]
  8284:        681b          ldr     r3, [r3, #0]
  8286:        68fa          ldr     r2, [r7, #12]
  8288:        4413          add     r3, r2
  828a:        60fb          str     r3, [r7, #12]
  sum += *data++;
  828c:        687b          ldr     r3, [r7, #4]
  828e:        1d1a          adds    r2, r3, #4
  8290:        607a          str     r2, [r7, #4]
  8292:        681b          ldr     r3, [r3, #0]
  8294:        68fa          ldr     r2, [r7, #12]
  8296:        4413          add     r3, r2
  8298:        60fb          str     r3, [r7, #12]
  sum += *data++;
  829a:        687b          ldr     r3, [r7, #4]
  829c:        1d1a          adds    r2, r3, #4
  829e:        607a          str     r2, [r7, #4]
  82a0:        681b          ldr     r3, [r3, #0]
  82a2:        68fa          ldr     r2, [r7, #12]
  82a4:        4413          add     r3, r2
  82a6:        60fb          str     r3, [r7, #12]
  sum += *data++;
  82a8:        687b          ldr     r3, [r7, #4]
  82aa:        1d1a          adds    r2, r3, #4
  82ac:        607a          str     r2, [r7, #4]
  82ae:        681b          ldr     r3, [r3, #0]
  82b0:        68fa          ldr     r2, [r7, #12]
  82b2:        4413          add     r3, r2
  82b4:        60fb          str     r3, [r7, #12]
  n -= 4;
  82b6:        683b          ldr     r3, [r7, #0]
  82b8:        3b04          subs    r3, #4
  82ba:        603b          str     r3, [r7, #0]
} while (n != 0);
  82bc:        683b          ldr     r3, [r7, #0]
  82be:        2b00          cmp     r3, #0
  82c0:        d1dd          bne.n   827e <checksum+0xe>
```

Listing 6.19: Checksum 2 - ARM

```
.data
struct data
{
  // Assembler does not currently support arrays, so the following is used
     to represent uint32_t array[64];
  uint32_t a0 = 0;
  uint32_t a1 = 1;
  uint32_t a2 = 2;
  uint32_t a3 = 3;
  uint32_t a4 = 4;
  uint32_t a5 = 5;
  uint32_t a6 = 6;
  uint32_t a7 = 7;
  uint32_t a8 = 8;
  uint32_t a9 = 9;
  uint32_t a10 = 10;
  uint32_t a11 = 11;
  uint32_t a12 = 12;
  uint32_t a13 = 13;
  uint32_t a14 = 14;
  uint32_t a15 = 15;
  uint32_t count = 0;
  uint32_t numElements = 16;
  uint32_t ptr = 0;
  uint32_t sum = 0;
};

.bss

struct bss
{
  uint32_t ignore;
};

.tree
expr_UNUSED: *, +, B_AND, +, *, 7, 7, 7, 7, 7, 7, 1, 1, 0, 255, 1, 0, 7,
    ignore
expr_assign_ptr_addr_of_data: *, +, B_AND, +, *, 1, 7, 7, 7, 7, 7, a0, 1, 0,
     255, 1, 0, 0, ptr
expr_count_less_than_numElements: *, +, <, +, *, 0, 7, 7, 0, 7, 7, count, 1,
     0, numElements, 1, 0, 7, ignore
expr_inc_count_by_4: *, +, B_AND, +, *, 0, 7, 7, 7, 7, 7, count, 1, 4, 255,
    1, 0, 0, count
expr_inc_ptr: *, +, B_AND, +, *, 0, 7, 7, 7, 7, 7, ptr, 1, 4, 255, 1, 0, 0,
    ptr
expr_add_element_to_sum: *, +, B_AND, +, *, 2, 7, 0, 7, 7, 7, ptr, 1, sum,
    255, 1, 0, 0, sum

.text
expr_assign_ptr_addr_of_data

while label1 expr_count_less_than_numElements
  expr_add_element_to_sum
```

```
  expr_inc_ptr
  expr_add_element_to_sum
  expr_inc_ptr
  expr_add_element_to_sum
  expr_inc_ptr
  expr_add_element_to_sum
  expr_inc_ptr
  expr_inc_count_by_4
label1:
```

Listing 6.20: Checksum 2 - SDLP

```
int sum = 0;
  827a:        2300            movs    r3, #0
  827c:        60fb            str     r3, [r7, #12]
do
{
  sum += *data++;
  827e:        687b            ldr     r3, [r7, #4]
  8280:        1d1a            adds    r2, r3, #4
  8282:        607a            str     r2, [r7, #4]
  8284:        681b            ldr     r3, [r3, #0]
  8286:        68fa            ldr     r2, [r7, #12]
  8288:        4413            add     r3, r2
  828a:        60fb            str     r3, [r7, #12]
  sum += *data++;
  828c:        687b            ldr     r3, [r7, #4]
  828e:        1d1a            adds    r2, r3, #4
  8290:        607a            str     r2, [r7, #4]
  8292:        681b            ldr     r3, [r3, #0]
  8294:        68fa            ldr     r2, [r7, #12]
  8296:        4413            add     r3, r2
  8298:        60fb            str     r3, [r7, #12]
  sum += *data++;
  829a:        687b            ldr     r3, [r7, #4]
  829c:        1d1a            adds    r2, r3, #4
  829e:        607a            str     r2, [r7, #4]
  82a0:        681b            ldr     r3, [r3, #0]
  82a2:        68fa            ldr     r2, [r7, #12]
  82a4:        4413            add     r3, r2
  82a6:        60fb            str     r3, [r7, #12]
  sum += *data++;
  82a8:        687b            ldr     r3, [r7, #4]
  82aa:        1d1a            adds    r2, r3, #4
  82ac:        607a            str     r2, [r7, #4]
  82ae:        681b            ldr     r3, [r3, #0]
  82b0:        68fa            ldr     r2, [r7, #12]
  82b2:        4413            add     r3, r2
  82b4:        60fb            str     r3, [r7, #12]
  sum += *data++;
  82b6:        687b            ldr     r3, [r7, #4]
  82b8:        1d1a            adds    r2, r3, #4
  82ba:        607a            str     r2, [r7, #4]
  82bc:        681b            ldr     r3, [r3, #0]
  82be:        68fa            ldr     r2, [r7, #12]
  82c0:        4413            add     r3, r2
  82c2:        60fb            str     r3, [r7, #12]
  sum += *data++;
  82c4:        687b            ldr     r3, [r7, #4]
  82c6:        1d1a            adds    r2, r3, #4
  82c8:        607a            str     r2, [r7, #4]
  82ca:        681b            ldr     r3, [r3, #0]
  82cc:        68fa            ldr     r2, [r7, #12]
  82ce:        4413            add     r3, r2
  82d0:        60fb            str     r3, [r7, #12]
```

```
    sum += *data++;
    82d2:       687b        ldr     r3, [r7, #4]
    82d4:       1d1a        adds    r2, r3, #4
    82d6:       607a        str     r2, [r7, #4]
    82d8:       681b        ldr     r3, [r3, #0]
    82da:       68fa        ldr     r2, [r7, #12]
    82dc:       4413        add     r3, r2
    82de:       60fb        str     r3, [r7, #12]
    sum += *data++;
    82e0:       687b        ldr     r3, [r7, #4]
    82e2:       1d1a        adds    r2, r3, #4
    82e4:       607a        str     r2, [r7, #4]
    82e6:       681b        ldr     r3, [r3, #0]
    82e8:       68fa        ldr     r2, [r7, #12]
    82ea:       4413        add     r3, r2
    82ec:       60fb        str     r3, [r7, #12]
    n -= 8;
    82ee:       683b        ldr     r3, [r7, #0]
    82f0:       3b08        subs    r3, #8
    82f2:       603b        str     r3, [r7, #0]
  } while (n != 0);
    82f4:       683b        ldr     r3, [r7, #0]
    82f6:       2b00        cmp     r3, #0
    82f8:       d1c1        bne.n   827e <checksum+0xe>
```

Listing 6.21: Checksum 3 - ARM

```
.data
struct data
{
  // Assembler does not currently support arrays, so the following is used
     to represent uint32_t array[64];
  uint32_t a0 = 0;
  uint32_t a1 = 1;
  uint32_t a2 = 2;
  uint32_t a3 = 3;
  uint32_t a4 = 4;
  uint32_t a5 = 5;
  uint32_t a6 = 6;
  uint32_t a7 = 7;
  uint32_t a8 = 8;
  uint32_t a9 = 9;
  uint32_t a10 = 10;
  uint32_t a11 = 11;
  uint32_t a12 = 12;
  uint32_t a13 = 13;
  uint32_t a14 = 14;
  uint32_t a15 = 15;
  uint32_t count = 0;
  uint32_t numElements = 16;
  uint32_t ptr = 0;
  uint32_t sum = 0;
};

.bss

struct bss
{
  uint32_t ignore;
};

.tree
expr_UNUSED: *, +, B_AND, +, *, 7, 7, 7, 7, 7, 7, 1, 1, 0, 255, 1, 0, 7,
    ignore
expr_assign_ptr_addr_of_data: *, +, B_AND, +, *, 1, 7, 7, 7, 7, 7, a0, 1, 0,
     255, 1, 0, 0, ptr
expr_count_less_than_numElements: *, +, <, +, *, 0, 7, 7, 0, 7, 7, count, 1,
     0, numElements, 1, 0, 7, ignore
expr_inc_count_by_8: *, +, B_AND, +, *, 0, 7, 7, 7, 7, 7, count, 1, 8, 255,
    1, 0, 0, count
expr_inc_ptr: *, +, B_AND, +, *, 0, 7, 7, 7, 7, 7, ptr, 1, 4, 255, 1, 0, 0,
    ptr
expr_add_element_to_sum: *, +, B_AND, +, *, 2, 7, 0, 7, 7, 7, ptr, 1, sum,
    255, 1, 0, 0, sum

.text
expr_assign_ptr_addr_of_data

while label1 expr_count_less_than_numElements
  expr_add_element_to_sum
```

```
  expr_inc_ptr
  expr_add_element_to_sum
  expr_inc_ptr
  expr_add_element_to_sum
  expr_inc_ptr
  expr_add_element_to_sum
  expr_inc_ptr
  expr_add_element_to_sum
  expr_inc_ptr
  expr_add_element_to_sum
  expr_inc_ptr
  expr_add_element_to_sum
  expr_inc_ptr
  expr_add_element_to_sum
  expr_inc_ptr
  expr_inc_count_by_8
label1:
```

Listing 6.22: Checksum 3 - SDLP

```
82a8:   e034        b.n     8314 <matrix_mul+0x7c>
82aa:   2300        movs    r3, #0
82ac:   617b        str     r3, [r7, #20]
82ae:   e02b        b.n     8308 <matrix_mul+0x70>
82b0:   2300        movs    r3, #0
82b2:   61fb        str     r3, [r7, #28]
82b4:   2300        movs    r3, #0
82b6:   61bb        str     r3, [r7, #24]
82b8:   e017        b.n     82ea <matrix_mul+0x52>
82ba:   693b        ldr     r3, [r7, #16]
82bc:   005a        lsls    r2, r3, #1
82be:   69bb        ldr     r3, [r7, #24]
82c0:   4413        add     r3, r2
82c2:   009b        lsls    r3, r3, #2
82c4:   68ba        ldr     r2, [r7, #8]
82c6:   4413        add     r3, r2
82c8:   681b        ldr     r3, [r3, #0]
82ca:   69ba        ldr     r2, [r7, #24]
82cc:   0051        lsls    r1, r2, #1
82ce:   697a        ldr     r2, [r7, #20]
82d0:   440a        add     r2, r1
82d2:   0092        lsls    r2, r2, #2
82d4:   6879        ldr     r1, [r7, #4]
82d6:   440a        add     r2, r1
82d8:   6812        ldr     r2, [r2, #0]
82da:   fb02 f303   mul.w   r3, r2, r3
82de:   69fa        ldr     r2, [r7, #28]
82e0:   4413        add     r3, r2
82e2:   61fb        str     r3, [r7, #28]
82e4:   69bb        ldr     r3, [r7, #24]
82e6:   3301        adds    r3, #1
82e8:   61bb        str     r3, [r7, #24]
82ea:   69bb        ldr     r3, [r7, #24]
82ec:   2b01        cmp     r3, #1
82ee:   d9e4        bls.n   82ba <matrix_mul+0x22>
82f0:   693b        ldr     r3, [r7, #16]
82f2:   005a        lsls    r2, r3, #1
82f4:   697b        ldr     r3, [r7, #20]
82f6:   4413        add     r3, r2
82f8:   009b        lsls    r3, r3, #2
82fa:   68fa        ldr     r2, [r7, #12]
82fc:   4413        add     r3, r2
82fe:   69fa        ldr     r2, [r7, #28]
8300:   601a        str     r2, [r3, #0]
8302:   697b        ldr     r3, [r7, #20]
8304:   3301        adds    r3, #1
8306:   617b        str     r3, [r7, #20]
8308:   697b        ldr     r3, [r7, #20]
830a:   2b01        cmp     r3, #1
830c:   d9d0        bls.n   82b0 <matrix_mul+0x18>
830e:   693b        ldr     r3, [r7, #16]
8310:   3301        adds    r3, #1
8312:   613b        str     r3, [r7, #16]
```

```
8314:      693b       ldr    r3, [r7, #16]
8316:      2b01       cmp    r3, #1
8318:      d9c7       bls.n  82aa <matrix_mul+0x12>
```

Listing 6.23: Matrix - ARM

```
.data
struct data
{
  // Assembler does not currently support arrays, so the following is used
      to represent uint32_t array[10];
  uint32_t a0 = 0;
  uint32_t a1 = 0;
  uint32_t a2 = 0;
  uint32_t a3 = 0;
  uint32_t b0 = 1;
  uint32_t b1 = 2;
  uint32_t b2 = 3;
  uint32_t b3 = 4;
  uint32_t c0 = 2;
  uint32_t c1 = 0;
  uint32_t c2 = 1;
  uint32_t c3 = 2;
  uint32_t ptr_a = 0;
  uint32_t ptr_b = 0;
  uint32_t ptr_c = 0;
  uint32_t i = 0;
  uint32_t j = 0;
  uint32_t k = 0;
  uint32_t elem_a_addr = 0;
  uint32_t elem_b_addr = 0;
  uint32_t elem_c_addr = 0;
  uint32_t elem_b = 0;
  uint32_t elem_c = 0;
  uint32_t sum = 0;
};

.bss

struct bss
{
  uint32_t ignore;
};

.tree
expr_UNUSED: *, +, B_AND, +, *, 7, 7, 7, 7, 7, 7, 1, 1, 0, 255, 1, 0, 7,
    ignore
expr_assign_ptr_a_addr_of_a0: *, +, B_AND, +, *, 1, 7, 7, 7, 7, 7, a0, 1, 0,
     255, 1, 0, 0, ptr_a
expr_assign_ptr_b_addr_of_b0: *, +, B_AND, +, *, 1, 7, 7, 7, 7, 7, b0, 1, 0,
     255, 1, 0, 0, ptr_b
expr_assign_ptr_c_addr_of_c0: *, +, B_AND, +, *, 1, 7, 7, 7, 7, 7, c0, 1, 0,
     255, 1, 0, 0, ptr_c
expr_set_sum_to_0: *, +, B_AND, +, *, 7, 7, 7, 7, 7, 7, 0, 1, 0, 255, 1, 0,
    0, sum
expr_i_less_than_2: *, +, <, +, *, 0, 7, 7, 7, 7, 7, i, 1, 0, 2, 1, 0, 7,
    ignore
expr_j_less_than_2: *, +, <, +, *, 0, 7, 7, 7, 7, 7, j, 1, 0, 2, 1, 0, 7,
    ignore
```

```
expr_k_less_than_2: *, +, <, +, *, 0, 7, 7, 7, 7, 7, k, 1, 0, 2, 1, 0, 7,
    ignore
expr_inc_i: *, +, B_AND, +, *, 0, 7, 7, 7, 7, 7, i, 1, 1, 255, 1, 0, 0, i
expr_inc_j: *, +, B_AND, +, *, 0, 7, 7, 7, 7, 7, j, 1, 1, 255, 1, 0, 0, j
expr_inc_k: *, +, B_AND, +, *, 0, 7, 7, 7, 7, 7, k, 1, 1, 255, 1, 0, 0, k
expr_set_i_to_0: *, +, B_AND, +, *, 7, 7, 7, 7, 7, 7, 0, 1, 0, 255, 1, 0, 0,
     i
expr_set_j_to_0: *, +, B_AND, +, *, 7, 7, 7, 7, 7, 7, 0, 1, 0, 255, 1, 0, 0,
     j
expr_set_k_to_0: *, +, B_AND, +, *, 7, 7, 7, 7, 7, 7, 0, 1, 0, 255, 1, 0, 0,
     k

// Calc ndx for b
expr_calc_addr_elem_b_step_1: *, +, B_AND, +, *, 0, 7, 0, 7, 7, 7, i, 2, k,
    255, 1, 0, 0, elem_b_addr

// Mul index for size of uint32_t
expr_calc_addr_elem_b_step_2: *, +, B_AND, +, *, 0, 7, 7, 7, 7, 7,
    elem_b_addr, 4, 0, 255, 1, 0, 0, elem_b_addr

// Add to ptr_b
expr_calc_addr_elem_b_step_3: *, +, B_AND, +, *, 0, 7, 0, 7, 7, 7,
    elem_b_addr, 1, ptr_b, 255, 1, 0, 0, elem_b_addr
expr_get_elem_b: *, +, B_AND, +, *, 2, 7, 7, 7, 7, 7, elem_b_addr, 1, 0,
    255, 1, 0, 0, elem_b

// Calc ndx for c
expr_calc_addr_elem_c_step_1: *, +, B_AND, +, *, 0, 7, 0, 7, 7, 7, k, 2, j,
    255, 1, 0, 0, elem_c_addr

// Mul index for size of uint32_t
expr_calc_addr_elem_c_step_2: *, +, B_AND, +, *, 0, 7, 7, 7, 7, 7,
    elem_c_addr, 4, 0, 255, 1, 0, 0, elem_c_addr

// Add to ptr_c
expr_calc_addr_elem_c_step_3: *, +, B_AND, +, *, 0, 7, 0, 7, 7, 7,
    elem_c_addr, 1, ptr_c, 255, 1, 0, 0, elem_c_addr
expr_get_elem_c: *, +, B_AND, +, *, 2, 7, 7, 7, 7, 7, elem_c_addr, 1, 0,
    255, 1, 0, 0, elem_c

// Mul elem_b and elem_c and add to sum
expr_calc_sum: *, +, B_AND, +, *, 0, 0, 0, 7, 7, 7, elem_b, elem_c, sum,
    255, 1, 0, 0, sum

// Calc ndx for a
expr_calc_addr_elem_a_step_1: *, +, B_AND, +, *, 0, 7, 0, 7, 7, 7, i, 2, j,
    255, 1, 0, 0, elem_a_addr

// Mul index by for size of uint32_t
expr_calc_addr_elem_a_step_2: *, +, B_AND, +, *, 0, 7, 7, 7, 7, 7,
    elem_a_addr, 4, 0, 255, 1, 0, 0, elem_a_addr

// Add to ptr_a
```

```
expr_calc_addr_elem_a_step_3: *, +, B_AND, +, *, 0, 7, 0, 7, 7, 7,
    elem_a_addr, 1, ptr_a, 255, 1, 0, 0, elem_a_addr
expr_set_elem_a *, +, B_AND, +, *, 0, 7, 7, 7, 7, 7, sum, 1, 0, 255, 1, 0,
    1, elem_a_addr

.text
expr_assign_ptr_a_addr_of_a0
expr_assign_ptr_b_addr_of_b0
expr_assign_ptr_c_addr_of_c0

while label1 expr_i_less_than_2
    expr_set_j_to_0

    while label2 expr_j_less_than_2
        expr_set_sum_to_0
        expr_set_k_to_0

        while label3 expr_k_less_than_2
            expr_calc_addr_elem_b_step_1
            expr_calc_addr_elem_b_step_2
            expr_calc_addr_elem_b_step_3
            expr_get_elem_b
            expr_calc_addr_elem_c_step_1
            expr_calc_addr_elem_c_step_2
            expr_calc_addr_elem_c_step_3
            expr_get_elem_c
            expr_calc_sum
            expr_inc_k
        label3:
        expr_calc_addr_elem_a_step_1
        expr_calc_addr_elem_a_step_2
        expr_calc_addr_elem_a_step_3
        expr_set_elem_a
        expr_inc_j
    label2:
    expr_inc_i
label1:
```

Listing 6.24: Matrix - SDLP

Table 6.6 illustrates the benchmark clock cycle counts for ARM and the SDLP.

	ARM	**SDLP**	**Difference**	**% Improvement**
Linear Search	540	413	127	23
Loop 1	267	180	87	32
Loop 6	493	180	313	63
Loop 2	2517	1710	807	32
Checksum 1	752	577	175	23
Checksum 2	500	373	127	25
Checksum 3	458	339	119	26
Matrix	*990*	*844*	*146*	*14*

Table 6.6: Benchmark Clock Cycle Counts for ARM and SDLP

It can be seen that the most significant improvement in clock cycles is by not using PC-relative addressing for managing global variables. However, PC-relative addressing is necessary in order to have a uniform instruction size. Since there are only 2 bytes available for ARM Thumb instructions, there is limited space for encoding literals. There are benefits in having a uniform instruction size and shape. For example, this helps to keep the decoder as simple as possible. The side effect of this is that the power required by the decoder circuitry is minimised. However, PC-relative addressing significantly increases the workload of the processor, therefore increasing power consumption. The aim would therefore be to minimise such overheads by limiting the usage of global variables on ARM-based architectures. It is interesting that, whilst certain design decisions may be motivated by reducing complexity in one domain, they may inadvertently introduce a negative, more significant impact in another. Design decisions made decades ago may no longer be valid.

It is clear from comparing Loop 1 and Loop 2 that the ratios of improvement are constant with respect to the number of iterations. This makes sense because the instructions themselves are static; they do not change as they are executing.

The differences in improvement for the Checksum benchmarks are interesting. Each Checksum benchmark increases the number of expressions executed in a loop, whilst reducing the number of iterations of the loop. It shows that, as the number of loop iterations decreases by a factor 4 whilst still processing the same number of expressions, the improvement decreases only slightly. This implies that the execution overhead of the abstract instructions is low. The ratio of clock cycles between Checksum 1 and Checksum 3 for ARM is 1.6. For the SDLP it is 1.7. The reason why the improvement between Checksum 1 and Checksum 3 for the SDLP has not changed much, even though the loop has been unrolled, is likely to be because of the under utilisation of the expression

engine. The expressions clearly dominate the execution cycles.

The clock cycle counts for Matrix were obtained slightly differently from the other benchmarks (hence the results are shown in italics). Since the program has 3 nested loops with multiple expressions, the machine code generated by *GCC* was structurally more complex. This is because of the way that the compiler translates *while* loops. Manually *dry running* the program in order to calculate the number of *data memory reads* and *data writes* was difficult and error prone. A *GDB* user-defined function was written to count the instructions up to a given *program counter* address. The simpler benchmarks assumed that each data memory access consumed one clock cycle. Since the *GDB* user-defined function is only capable of counting instructions, the memory access counts were *not* included in the Matrix results.

The Matrix benchmark shows interesting results. Even though the improvement is only 14%, this is a pessimistic result, due to the way that the simulator accumulates clock cycles. It can be observed in Listing 6.24 that the program has various *groups* of expression references in the *.text* section. There are groups of 3, 2, 10 and 5. These groups of expression references are within up to 3 levels of loop nesting. It is assumed that the SDLP would be capable of fetching and decoding a *contiguous group* of (up to) 8 expression references together. However, the simulator is not capable of identifying such groupings. The simulator accumulates a separate fetch and decode cycle for each *individual* expression reference instead of 2 cycles for each *group* of expressions. As an example, for the group of 10 expression references (which are executed 8 times), this means that $10 * 2 * 8 = 160$ fetch/decode cycles are counted instead of $2*8 = 16$ for the first group of 8, and then $2*8 = 16$ for the remaining group of 2. This is a total difference of 128 cycles. Accounting for this means that the Matrix benchmark would show a 27% improvement. It appears that the ability to fetch and decode contiguous expression references yields significant improvements.

From writing the SDLP benchmarks it is apparent that the number of expressions could be reduced by having a better shaped expression engine. The tree may have a reasonable shape for comparisons of subexpressions on either side of a logical operator, e.g.

$$a * b + c < d/e - f$$

However, it can be noted from Listing 6.24 and other benchmarks that the right hand side of the tree is often ineffective. For the common case the right hand side is not utilised for any constructive calculation. It simply *bitwise ands* the value of the left hand side with ~ 0 in order to allow it to *pass-through*.

6.5 Summary

An architecture for the SDLP has been defined. An assembler[10] and simulator have been developed and are both regarded significant contributions. The purpose of these is to allow further architectural exploration of the SDLP.

The SDLP requires 14% fewer *loads* for the *Linear Search program* than ARM, but more than 8086. The reasons for this have been explained. With appropriate support for arrays in the form of *immediate offsets* and *displacement addressing*, it is possible that the number of loads could be reduced significantly to match that of 8086.

The instruction count (including *instructions*, *expressions Ids* and *null terminators*) for the SDLP is significantly less than both 8086 and ARM, provided that expressions are pre-loaded during start-up. This could be further reduced if *null terminators* could be encoded within the *expression Ids*.

The above provide a positive indication to the thesis claim that *high-level ISAs and supporting processor architectures can reduce the burden on the memory system for both instructions and data.* However, this is for a single benchmark only, so these results must be viewed very cautiously.

The approach used for calculating ARM Thumb clock cycles is processor model agnostic and no tools other than *objdump* and *GDB* were used for this purpose. In most cases calculations were done manually using the disassemblies. The approach used for the SDLP was to instrument the simulator with the clock cycle counts derived from pseudocode. Comparing metrics such as the number of clock cycles from one ISA to another is challenging and comparisons can often be less than meaningful. However, comparisons must be made if improvements to processor architecture are to be possible. To this end, a careful balance has been made when calculating the clock cycle counts. The calculations for ARM Thumb code have been deliberately optimistic, whereas the calculations for the SDLP have been rather pessimistic. Furthermore, factors such as pipelining, optimisation and primitive ALU-based operations have purposely been factored out from the figures. This means that the focus is biased more towards the effects of the ISA rather than feature optimisations.

The results for clock cycle counts also look positive, with a median 25.5% improvement for the set of eight benchmarks. These results offer cautiously positive answers to the thesis claim of *reducing the cycle count of programs.*

The SDLP appears capable of executing programs with fewer loads and cycles than ARM. This has been made possible with an ISA that is applicable to modern general purpose imperative

[10] A guide to using the assembler can be found in Appendix B.

languages. Since the SDLP can execute general-purpose programs in fewer cycles than ARM, this means:

- It may be clocked at a lower rate whilst achieving the same throughput;
- The reduced clock rate may have the potential of reducing power consumption;
- Alternatively, it may be clocked at the same rate as other processors with the potential of achieving higher throughput.

Whilst there is *potential* for the SDLP to either reduce power consumption or increase throughput compared to other processors, it should be noted that any power savings cannot be determined until the SDLP is developed in ASIC. This is because power consumption may actually *increase* if the SDLP datapaths are more complex. Whilst the clock cycles for a *simulated* program may be lower, the processor circuitry of the SDLP in ASIC may increase. It is therefore possible that any increase in circuitry may negate any power savings due to the SDLP ISA.

Chapter 7

Conclusions

This chapter offers conclusions and reconsiders the research hypothesis that was proposed in the Introduction (Chapter 1), *that high-level ISAs and supporting processor architectures can reduce the burden on the memory system for both instructions and data; and can reduce the cycle count of programs*. A summary contributions is presented, followed by closing remarks.

7.1 High-Level ISAs and Supporting Processor Architectures can Reduce the Burden on the Memory System for Both Instructions and Data

In order to determine whether the burden on the memory system can be reduced for both instructions and data, a number of benchmarking exercises were conducted. An initial benchmarking exercise was conducted to better understand the actual burden that a RISC processor places on the memory system. After the expression engine had been designed and assembler developed, comparisons of assembled expressions and ARM Thumb programs were made. Finally, after the ISA and SDLP architecture had been defined and a simulator developed, further benchmarking experiments were conducted. These compared loads, stores and instructions for assembled SDLP and ARM Thumb programs.

7.1.1 The Burden that RISC Processors Place on the Memory System and the Projected Energy Costs

Verma et al. [3] demonstrated that 50-70% of the total power for a system is consumed by the memory system. In order to obtain an independent opinion on the frequency of various memory system interactions, a benchmarking exercise was conducted. The benchmarks employed were *mibench* [45] and they were run via the *gem5* simulator [46], which simulated an ARMv7 Cortex A15 uniprocessor. Various results were gathered including:

- Number of instructions executed;
- Number of loads and stores;
- Number of hits and misses for both instructions and data cache;
- Number of write-backs to the data cache.

The findings suggest that the number of loads and stores are significant. For example, the proportion of load and store instructions for *crc32_large* is 50% of the total instructions simulated.

The next step was to determine how much *energy* was spent processing instruction and data cache misses and data cache write-backs. To achieve this, *energy* estimates from Verma et al. [3] were used. The energy estimates for *scratchpad* memory read and writes were used to represent cache read and writes, since these are both high-speed *on-chip* memory and should be similar. The energy estimates for *main memory* reads were used to represent cache misses and *main memory* writes were used to represent data cache *write-backs*. These energy estimates were multiplied with the corresponding metrics for the benchmark results. However, the results were not a source of concern.

Next, a more pessimistic approach was taken using the *comparative access times* for register access, *L1* cache, *L2* cache and main memory, illustrated in Figure 2.3. Although these approximations are based on *time* rather than *energy*, there is still, nonetheless, a relationship between time and energy. Again the results were of no immediate concern. It can be concluded that *time* wasted accessing external memory due to cache misses and write-backs is not a concern.

Cache designers have done a very good job managing the disparity in both energy and time when interacting with external memory. However, from another perspective, it is clear that most applications are completely reliant on caching hardware in order to manage these disparities. It can be argued that, whilst the RISC design philosophy aims to simplify the core data path by providing very simple instructions, the architecture requires significant circuitry in the form of one or more caches: problems may have been simply *moved* from one area of the processor to another.

7.1.2 Comparing Assembled Expressions and ARM Thumb

After designing the expression engine and developing the assembler, expressions taken from a *Linear Search* program were compared with the equivalent assembled ARM Thumb code. The *number of bytes* required for the expressions were of interest.

If it is assumed that the expressions are fetched each time they are used, the SDLP requires 10 bytes more than the ARM Thumb code using global variables (hence PC-relative addressing) and 34 bytes more than ARM Thumb using local variables.

It is apparent that if the expressions are fetched each time they are referenced, then the SDLP increases the dependency on the memory system for instructions. The reason for this is that ARM Thumb code only requires 4 bits for addressing 16 registers, whereas the SDLP can address 256 bytes of memory and so requires 8 bits for operand encoding. It should be emphasised that the ARM programs were compiled for Thumb (16-bit instructions) as opposed to regular ARM instructions which are 32 bits. It is likely that regular ARM instructions would increase the byte count significantly.

The current overhead of the SDLP expressions can be mitigated by *pre-loading* the expressions for a given program during start-up.

7.1.3 Comparing Loads, Stores and Instructions for Assembled SDLP and ARM Thumb

Once the ISA for the SDLP had been defined and simulator developed, further benchmarking experiments were conducted using the *Linear Search Program*. The gathered metrics included the number of loads, stores and number of instructions required to express the program for the given architecture.

The results showed that ARM required the greatest number of loads (94) following the SDLP (81) and then 8086 (52). Both ARM and 8086 required 14 stores and the SDLP required 25. The reason for the SDLP load and store results being so high is that the SDLP does not currently support arrays; they must be simulated using pointers.

ARM required 180 instructions, 8086 required 107, and the SDLP 11. Again, it should be emphasised that the ARM programs were compiled for Thumb (16-bit instructions) as opposed to regular ARM instructions which are 32 bits. It is clear that ARM incurs the highest dependency on the memory system for this benchmark. This is because of the load/store architecture and the verbosity of the ISA. Another reason for ARM loads being so high is that it employs PC-relative

addressing for global variables, which is necessary because of the uniform instruction size; literals cannot be encoded within an instruction.

It is surprising to see that ARM requires 60% more instructions than 8086; it appears that a RISC ISA imposes a significant overhead on the instruction memory system than 8086 for this benchmark.

The reason that the SDLP requires an order of magnitude fewer is because other necessary information, such as labels (immediate values for relative jumps), expression Ids used to *invoke* expressions and *null terminators* for terminating the end of a list of expression Ids and instructions, are not counted. The total overhead for the SDLP, including instructions, expressions Ids and null terminators is 92. Whist this is still a significant improvement, it should be noted that expressions are not counted in these results since they are pre-loaded; only their *invocation* is counted.

However, it does appear that high-level ISAs and supporting processor architectures can reduce the burden on the memory system for both instructions and data.

7.2 High-Level ISAs and Supporting Processor Architectures can Reduce the Cycle Count of Programs

Revisiting the *Linear Search* program, and assuming the the expressions are pre-loaded at program start-up, the SDLP can execute the program with 23% fewer clock cycles than ARM.

A further seven assembly language benchmarks were developed for ARM Thumb and the SDLP.

The SDLP results showed an improvement in clock cycle counts over ARM; the biggest was 63% and the smallest was 14%. The *mean* improvement was 29.75% with a *standard deviation* of 14.56. The *median* improvement was 25.5%.

The SDLP appears to require fewer clock cycles over an equivalent ARM-based processor for a set of simple benchmarks. However, these results should be considered with caution. The benchmarks used consist of a small set of simplistic programs as opposed to large real-world applications; they serve as a potential indicator only. The benchmarks for ARM have all been compiled without optimisations. This is because SDLP currently lacks the support of an optimising compiler. It is assumed that whatever optimisations are available for traditional architectures, there will be competing optimisations for the SDLP.

The real results can only be determined after a full ASIC implementation of the processor and compiler have been developed. It is important that the *critical-path* and *power per-cycle* does not

increase above other comparative architectures. It is also necessary to resolve the SDLP issues of expression verbosity, expression engine design, node underutilisation and optimisation.

7.3 Research Contributions

The main contributions of this thesis include:

- The design of a hybrid control flow and data flow architecture with a supporting Instruction Set Architecture (Chapters 4 and 5);
- Implementation of an assembler and software-based cycle accurate simulator for the processor (Chapter 6);
- Comparisons of the new architecture with traditional CISC and RISC processors (Chapters 6 and 7);
- It has been shown that high-level ISAs and supporting processor architectures can reduce the burden on the memory system for both instructions and data; and can reduce the cycle count of programs.

Other minor contributions include a clearer understanding of the topic:

- Microprocessors vendors are cornered into using multiprocessors to increase performance. However, this approach is severely limited compared to the performance gains made in past decades;
- Whilst RISC processors have achieved simpler data-paths, the ISA significantly burdens the memory system, for both instruction and data;
- The 8086 ISA appears significantly better than a RISC ISA for reducing the burden on the memory system;
- Amdahl's Law gives an optimistic view of speed-up via parallelism. The practicalities of NOCs and management of bottlenecks must also be considered;
- Despite dataflow computing not becoming mainstream to date, most superscalar architectures adopt this approach in hardware. This is expensive and unscalable;
- Previous dataflow approaches have adopted a coarse-grained or *wide instruction window* approach to execution, e.g. the TRIPS processor discussed in Charper 2.1.5. It appears that dataflow execution for imperative language-based programs is most natural at the *expression boundary*;

- The biggest improvements of the SDLP are due to the expression engine (and thus dataflow execution) over abstract instructions.

7.4 Closing Remarks

Current CISC and RISC ISAs have remained relatively unchanged for the past 50 years and there is a general misconception that they cannot be improved upon. However, these designs are clearly reaching their limits with current process technology. In order for processor technology to progress, it is necessary to reconsider alternative (and sometimes *old*) approaches with a view to solving current and emerging challenges.

Chapter 8

Future Work

This thesis documents the development of the SDLP which shows potential for reducing power consumption or increasing throughput at equivalent clock frequencies of traditional processors. This is achieved by placing fewer demands on the memory system and reducing the number of cycles required for program execution. Whilst the work indicates that there is significant potential for such a processor, many avenues of further research have become apparent along the way.

8.1 Expression Engine

Firstly, there is further work on the research and development of the expression engine. These suggestions should be considered first, as they impact the immediate issues and viability of the SDLP approach.

- Support for arrays. This can be achieved by implementing *displacement addressing* as discussed in Section 6. An example of this in 8086 assembly language is *mov array(%eax), %ebx*. The *displacement* is the value held in the *eax* register which gets added to the literal *array* which is an address in memory.
- Improve expression tree utilisation. Currently the expression tree is under-utilised by as much as 50%. It is suggested that static analysis of various domain code bases should be conducted to statistically determine the *most common* expression *shapes* and operands. This work could motivate the development of *domain specific expression engines*.
- As suggested in Chapter 4.5, various techniques should be considered for expression engine optimisation. These include *operand forwarding* and *clock gating* techniques. A *dynamic*

expression engine design should also be considered.

- Research how to *fit* arbitrary C expressions onto a given expression engine tree (or other suitable data structure). This would be required for a C compiler back-end code generator.
- Development of a C compiler *back-end* to facilitate the execution of existing benchmarking suites such as *mibench*. If a *GNU Compiler Collection* back-end is developed, it should be compatible with C and C++. Full compatibility with the GNU toolchain should be considered, for example by re-implementing the assembler using *GNU Assembler* GAS. This would encourage the open source development of the SDLP toolchain.
- Research compiler optimisations possible with the SDLP and how these compare with ARM compiler optimisations.
- Increase the number of operands that can be addressed. Currently 256 bytes can be addressed since *lvalues* and *rvalues* are 8 bits wide. An investigation needs to made to increase the number of operands that can be addressed whilst acknowledging that operand addressing dominates the expression definition space requirements as discussed in Section 4.5.
- Decrease the operand address width. This is in opposition to increasing the number of operands that can be addressed (above). Section 4.5 explained how opcode addressing dominates the expression definition space requirements. One possible approach is to use an internal register set in order to reduce the size of the operand address width from 8 bits to 4 bits, the same as ARM. Alternatively a register set could be *memory mapped*. However, if registers are used, then *load* and *store* or *mov* instructions would needed adding to the ISA. This would change the SDLP from a *memory-memory* architecture to a *register-memory* or *load-store* architecture.
- Single expression engine or multiple? Currently, the SDLP needs to configure *all* of the nodes of the expression engine simply to compare two variables or to simply increment a variable, which is wasteful. A number of expression engines could be supported to accommodate different expressions of varying complexities or *shapes*. A simple ALU could be used for supporting dual operand expressions.
- Multiple expression engines or ILP for increasing parallelism? Parallelism could be achieved using ILP or by increasing the number of expression engines. Each approach will yield various advantages and disadvantages, which need to be understood.

8.2 Practicalities

Secondly, a number of other items should be considered to complete the SDLP ISA and architecture in order to make it a practical processor.

- Decide whether to continue with the use the abstract instructions or use alternative CISC or RISC-based instructions for flow control.
- Further development of the abstract instructions. The SDLP ISA can be augmented to provide support for other C control constructs, e.g. *for*, *dowhile*, *goto*, *setjump*, *longjump* and *switch*. With regards to switch statements, approaches using *lookup tables* and j*ump tables* for efficiency should be considered. Support for function call and return also needs adding. Since this work is outside of the scope of the current thesis, suggestions have been written up in Appendix A.
- Floating Point support needs to be added to the SDLP. There are various approaches to this, for example by modifying the expression engine to support floating point calculations, or by using a seperate floating point expression engine. Floating point support will enable more complex benchmarks to be executed on the SDLP.
- Support for peripheral devices and interrupts. In order to be capable of controlling external devices, the SDLP must provide support for external device control and interrupts. For example, this could be done using hardware control registers or memory mapped registers. The integration or development of an interrupt controller also needs to be considered. However, this work is not necessary for supporting most benchmarking suites.

8.3 ASIC Implementation

Finally, the ultimate aim for the SDLP is to move from a research concept into a customised *integrated circuit* for use in a commercial environment. This stage requires the expertise of a multi-functional team including the SDLP architect, runtime software engineers, tool-chain engineers (including compiler experts) and digital design engineers (in particular FPGA and ASIC experts). This phase of development is likely to require significant investment.

- Development of a *Register Transfer Level* (RTL) *Field Programmable Gate Array* (FPGA) based implementation of the SDLP. The FPGA is an integrated circuit which can be configured by the designer after manufacturing. This is usually one of the first steps

towards implementing a processor as an *Application Specific Integrated Ciruit* (ASIC), which is a fully customised integrated circuit. When implementing designs on FPGAs, either a *behavioural* model can be applied or an RTL. An RTL approach is a clocked synchronous implementation which models the flow of digital signals between the various hardware components. This approach means that the implementation is easier to transition to an ASIC implementation later, whereas a behavioural model relies on the FPGA toolchain to generate a *state machine*. Hence, this resembles a simulator running on FPGA. This work will require the expertise of a digital hardware engineer working closely with the SDLP architect.

- Determine how the non-uniform instruction length impacts decoder circuit, in particular transistor count and critical path. Currently, the instruction length for an *if-else* instruction is 8 bytes compared to 7 bytes for *while* and *if*. This does introduce some complexity to the *decoder* as described in Section 6.2.1. This information will feed into the development of the ISA for other C constructs (discussed above).
- Determine potential savings due to the SDLP having fewer dependencies on caches. This may reduce overall power consumption.
- Finally, an ASIC implementation of the processor will allow a fully integrated circuit to be benchmarked. Only at this point will the true potential of the SDLP be understood.

Appendix A

Function Calls and Returns

Functional decomposition refers broadly to the process of resolving a functional relationship into its constituent parts in such a way that the original function can be reconstructed (i.e. recomposed) from those parts by function composition. The *function* is one of the fundamental abstractions in modern programming languages and it is a feature provided by all practical imperative languages. The rules for describing the function call, return and parameter passing are specified in a *Procedure Call Standard*, for example [60]. Having such a standard means that modules compiled by different compilers, are compatible and can be linked into a single binary.

On 8086-based architectures, the *__cdecl* is the default calling convention for C and C++ programs. One of the main advantages of this calling convention is that can support *variable parameters*. This is because the stack is cleaned up by the *caller*.

The context for a function call is stored in an *Activation Record*. The activation record contains the following information (not necessarily in this order):

- Parameters
- Return address
- Storage space for local variables
- Platform-specific context information

Activation records are created on a stack, since this data structure naturally mimics the control sequence of function calls and returns. Figure A.1 illustrates the activation records for a *caller* and *callee* based on the *__cdecl* procedure call standard. It assumes that the stack grows downwards and, for the sake of simplicity, no platform-specific context information (e.g.

registers) need to be saved. The top 5 elements are for the *caller* and bottom 6 elements represent the *callee*. *BP 1* is the base pointer; this is used to provide a stable reference point for accessing the function parameters and local variables. *SP 1* is the stack pointer; this is used to track the last element in the stack.

When a function is called, a new activation record is created on the stack. First, *caller* arguments are pushed onto the stack; these become *callee* parameters. This is seen as *SP 2*. Next, a *call* instruction pushes the return address onto the stack, prior to jumping to the start of the *callee* function. The 8086-based assembly to achieve the pushing of the arguments and the call is referred to as the *caller_prologue*. The *callee* then preserves the *bp* by pushing it; it then sets the *bp* to the current *sp*. This is seen as *BP 2*. This creates an anchor point for accessing parameters and local variables. Finally, *sp* is adjusted to allocate local variable storage. This is seen as *SP 3*. The 8086-based assembly to achieve this is referred to as the *callee_prologue*. To return from the function, the *caller bp* must be restored, the local variables deallocated, and a *ret* instruction executed to return to the point after the function call in the *caller*. This is done by the *callee_epiloue*. Finally the *caller* arguments must be deallocated from the *caller* activation record. This is done by the *caller_epilogue*.

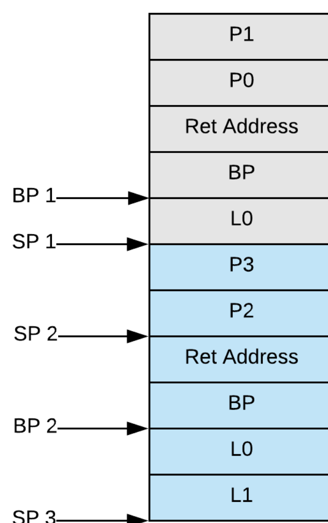

Figure A.1: Function Stack Frame

8086 assembly language can be used as pseudocode for describing the semantics that the SDLP would be required to implement for supporting function calls. Listing A.1 illustrates

the caller creating the first part of the activation record by pushing the arguments onto the stack. This is seen as *SP 2* in Figure A.1. These arguments will then become the callee's parameters. The caller then invokes the *call* instruction. This pushes the return address onto the stack (this is the address of the next instruction after the *call*), before jumping to the address of the function. At this point the stack pointer (*sp*) will point to the next stack element below the return address.

```
push p3
push p2
call callee
```

Listing A.1: Caller prologue

When control is passed to the *callee* via the jump, the *callee prologue* is the next code to execute. Listing A.2 illustrates how the construction of the activation record is completed by the *callee*. First the *caller's bp* is preserved by pushing it onto the stack. *bp* is then set to the stack pointer. This is shown in Figure A.1 as *BP 2*. The purpose of this is to create a *stable reference point* for accessing function parameters and local variables using *bp* for relative addressing. Next, space for local variables is allocated in the activation record. This is done by subtracting the required bytes from the stack pointer. This is shown in Figure A.1 as *SP 3*. Once the function is finished, control can be returned back to the *caller*. This is done by the *callee epilogue*.

```
push bp
mov bp, sp
sub sp, 8
```

Listing A.2: Callee prologue

Listing A.3 shows how control is passed back to the *caller*. First the stack pointer is set to the base pointer. Since *bp* points to the stack element after the *callers bp*, this has the effect of deallocating the space originally allocated for local variables. This is shown as *BP 2* in Figure A.1. The caller's base pointer is then restored by popping the stack. The state of *bp* is now back to where it was prior to the invocation of the *call*. This is shown as *BP 1*. Finally, the *ret* is executed. This pops the caller's return address from the stack and writes it to the program counter; this is shown as *SP 2* in Figure A.1. However, the stack still contains the arguments that were passed by the caller. The final cleanup is done by the *caller epilogue*.

```
mov sp, bp
pop bp
ret
```

Listing A.3: Callee epilogue

Listing A.4 shows how the arguments are removed from the stack. This is done by adding the appropriate number of bytes to the stack pointer. This is shown as *SP 1* in Figure A.1. The reason that the function arguments are deallocated by the *caller* rather than the *callee* is so that variable parameters can be supported during function calls; only the *caller* knows the number of arguments passed.

```
add sp, 8
```

Listing A.4: Caller epilogue

SDLP instructions can be derived from the above 8086 instruction sequences and these are illustrated in Table A.1.

Instruction Sequence	**8086 Code Sequence**	**Equivalent SDLP Code**
Caller Prologue	push p3 push p2 call callee	push <val> | <literal> call <function_name>

Callee Prologue	push bp
	mov bp, sp
	sub sp, 8
		salloc <num_bytes>

Callee Epilogue	mov sp, bp pop bp ret	ret
Caller Epilogue	add sp, 8	sdealloc <num_bytes>

Table A.1: SDLP Function Call and Return Instruction Sequences derived from 8086

Appendix B

Assembler User Guide

B.1 Introduction

The assembler is used to translate a human readable SDLP program into a binary executable for interpretation by the simulator.

The input to the assembler is a single text file and the output is a binary executable. The format of the output file is a proprietary executable, following a simple structure. To invoke the assembler the following command is entered: $asm < filename > < outputfilename >$.

There is no separate linker, and information that would be normally provided in a *linker map file* is defined internally by the assembler. This includes the memory start address and lengths of the various *memory segments*. For the sake of simplicity, the executable binary is loaded directly into memory by the associated simulator, ready for execution. To this extent, the assembler is also a *romizer*. The syntax of an SDLP assembly language program is loosely based on the GNU Assembler (GAS).

B.2 Memory Model

Figure B.1 illustrates a memory model used for SDLP executables. Memory address 0 is at the top of the diagram and the highest address is at the bottom. Byte order is *little endian* format for ease of debugging. Bit order is *least significant-bit first*. Memory is split into a number of sections, inspired by the Executable and Linkable Format (ELF) [61]. The *Segment Offset Table* contains the offsets of all the *segments* in the memory space. Each entry in this

table is 4 bytes in size. The purpose of the offset table is to allow the processor to locate all of the segments it requires for execution. For example, during boot-up, the processor must start executing code in the *text* segment. This segment consists of control instructions and expression references. The processor must also know the offset of the *data* segment when processing operands. The *tree* segment contains the expressions which configure the expression engine; each are 11 bytes in size.

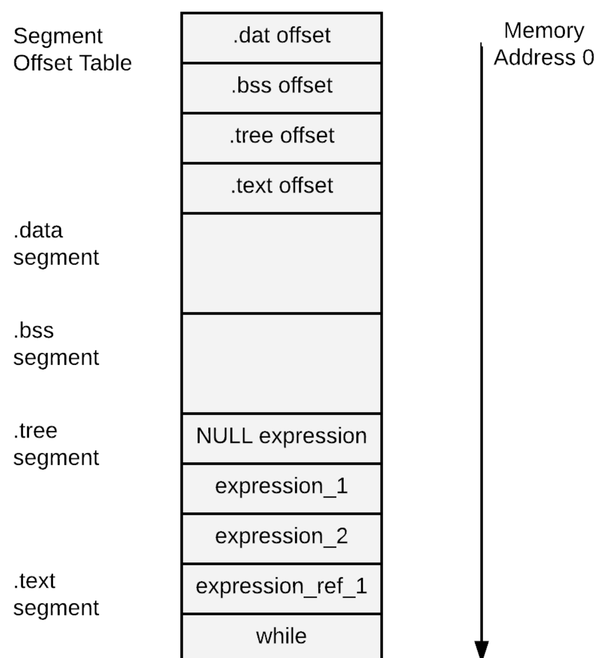

Figure B.1: SDLP Memory Model

An SDLP assembly language program must consist of the following sections in the specified order:

- *.data* - this contains initialised global variable definitions. Each global variable in this section *must* be initialised.
- *.bss* - this contains uninitialised global variables.
- *.tree* - this contains all of the program expressions, each uniquely identified by an *expression Id*.
- *.text* - this contains the program logic, comprising of instructions and *expression Ids*.

B.3 Data Segment and BSS Segment Declaration and Definition

The assembler supports the most basic primitive types including bytes, words and double words. These are declared and defined using a C-like syntax using types found in stdint.h of the standard C library. Listing B.1 illustrates how to declare various types in the *data* and *bss* segments[1]. Note, arrays are currently not supported.

```
.data
struct data {
  uint8_t foo1 = 0x0;
  uint32_t foo2 = 0xffffffff;
};
.bss
struct bss {
  uint8_t bar1;
  uint32_t bar2;
};
```

Listing B.1: Data and BSS Segments

[1]The assembler requires at least one variable in the *.data* segment and one variable in the .bss segment. If one is not required, simply insert a *dummy* one.

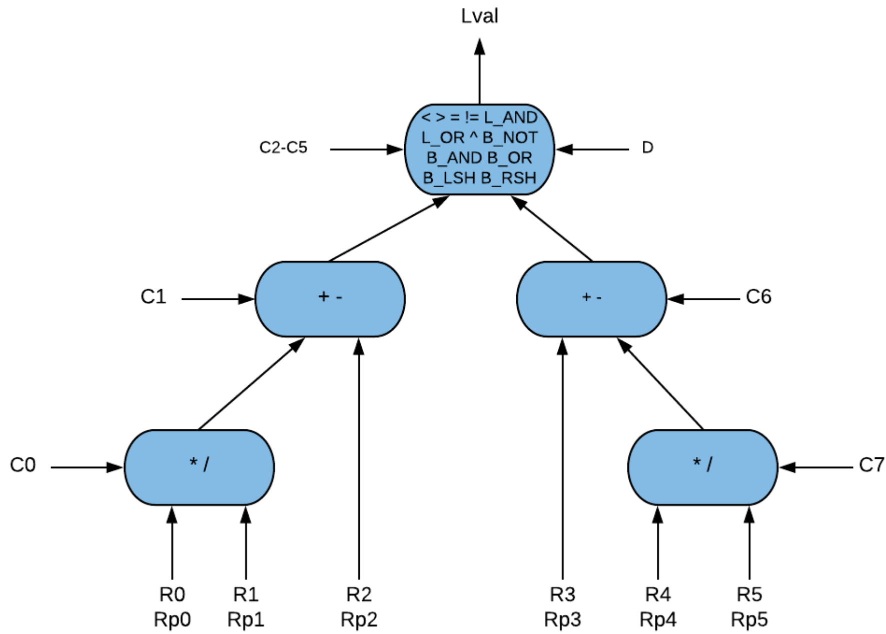

Figure B.2: Expression Engine

B.3.1 Expressions

Instead of a traditional ALU, the SDLP uses an *Expression Engine*. Figure B.2 illustrates the n-array tree structure of the expression engine.

Listing B.2 is the *Bacus Naur Form* (BNF) illustrating the syntax of an expression. The *terminals* refer to the node inputs in Figure B.2 above.

```
<expression> := <expression name> :
                <C0> <C1> <C2–C5> <C6> <C7>
                <RP0> <RP1> <RP2> <RP3> <RP4> <RP5>
                <R0> <R0> <R2> <R3> <R4> <R5>
                <D>
                <lval>
```

Listing B.2: Expression Encoding

R0-R5 are *rvalues* and are the equivalent of variables or constants on the right hand side of a C expression. The output operand is referred to as an *lvalue* (left of assignment). An operator must be selected for each node *C0-C7*; Table B.1 illustrates the possible operators for each of these.

Node	**Operator**
C0 and C7	* (arithmetic multiply) / (arithmetic divide)
C1 and C6	+ (arithmetic add) - (arithmetic subtract)
C2 - C5	< (relational less than) > (relational greater than) = (relational equals) ! = (relational not equals) L_AND (logical and) L_OR (logical or) ^(bitwise xor) B_NOT (bitwise not) B_AND (bitwise and) B_OR (bitwise or) B_LSH (bitwise left shift) B_RSH (bitwise right shift)

Table B.1: SDLP Operators

RP0-RP5 are used to specify how each of the corresponding *rvalues* should be *addressed*. The possible values for RP0-RP5 are illustrated in Table B.2.

Value	**Description**
0	*rvalue* (offset from.data)
1	Address of the *rvalue* (& in C) (from .data)
2	Value of the location pointed to by the *rvalue* (* in C) (offset from .data)
3	Not used
4	*rvalue* (offset from stack base)
5	Address of the *rvalue* (& in C) (from stack base)
6	Value of the location pointed to by the *rvalue* (* in C) (offset from stack base)
7	Literal (value is treated as an 8-bit literal)

Table B.2: RP0-RP5 Meanings

The possible values for D in layer 3 are illustrated in Table B.3.

Value	**Description**
0	*lvalue* (offset from .data)
1	Location pointed to by the *lvalue* (* in C) (offset from .data)
2	*lvalue* (offset from .stack)
3	Location pointed to by the *lvalue* (* in C) (offset from stack base)
4	Not used
5	Not used
6	Not used
7	Ignore (do not write *lvalue*, however internal *zero flag* still set to indicate result of expression)

Table B.3: D Meanings

Similar to *rvalues*, *lvalues* can be plain old variables, pointers and address of variables. It is also possible to skip writing the *lvalue*. This can be done when the result of an expression is used as a condition by an instruction, such as an *if* or a *while*. In this case an internal boolean *flag* is set *true* if the *lvalue* is non-zero and *false* otherwise, i.e. $bool\ flag = lvalue$. This flag is then used by an instruction to determine the result of its associated condition.

Listing B.3 illustrates an example expression which increments a variable *count*.

```
.tree
expr_inc_count:
*, +, B_AND, +, *,
0, 7, 7, 7, 7, 7,
count, 1, 1, 255, 1, 0,
0, count;
```

Listing B.3: Example Expression

B.3.2 While Instructions

Listing B.4 illustrates the syntax for a *while* and Listing B.5 illustrates a corresponding example.

label specifies the where program control jumps if the *while condition* equates to *zero*. In C, this would be the next statement after the closing brace of a *while* block.

expr_id refers to an expression defined in the *.tree* segment of the assembly language program. Multiple expression Ids can be specified for complex expressions. Expressions can be *chained* by specifying the *lvalue* output of a preceding expression as an *rvalue* input to the next expression. If the result of the last expression evaluation is zero, then the program control branches to *label*. Otherwise, control passes to the first statement in the *while* block.

The *while* must contain at least one *statement*. This can be another instruction or an expression.

```
while <label> <expr_id> [, <expr_id >...]
    <statement> [<statement >...]
<label >:
```

Listing B.4: While Encoding

```
while label1 expr_count_less_than_numElement
        expr_inc_count
label1:
```

Listing B.5: Example While

B.3.3 If Instructions

Listing B.6 illustrates the syntax for an *if* and Listing B.7 illustrates a corresponding example. The syntax is very similar to the *while*.

```
if <label> <expression_id> {, <expression_id>}
  <statement> {<statement>}
<label>:
```

Listing B.6: If Encoding

```
if label1 expr_count_less_than_numElement
        expr_inc_count
label1:
```

Listing B.7: If Encoding

B.3.4 If-else Instructions

Listing B.8 illustrates the syntax for an *if-else* and Listing B.9 illustrates a corresponding example.

label_0 defines the start of the *else* block. In C, this would be the statement after the opening brace of the *else* block. *label_1* defines the point in the program after the *else* block. In C, this would be the statement after the closing brace of the *else* block. If the result of the last expression evaluation is *zero*, then the program control branches to *label_0*, which is the *else* block. Otherwise, control passes to the first statement in the *if* block. The *if* block and the *else* block must both contain at least one statement each. When the last statement of the *if* block has been executed, program control jumps to *label_1*.

```
ifelse <label_0> <label_1> <expression_id> {, <expression_id>}
  <statement> {<statement>}
<label_0>:
  <statement> {<statement>}
<label_1>:
```

Listing B.8: If-else Encoding

```
if label_0 label_1 expr_a_less_than_b
        expr_mark_less_than
label_0:
        expr_not_less_than
label_1:
```

Listing B.9: If-else Encoding

B.3.5 Example SDLP Program

Listing B.10 illustrates a complete SDLP assembly language program. The program simply iterates *while* the value of a variable *count* is less than the value of a variable *numElements*.

```
.data
struct data
{
  uint32_t count = 0;
  uint32_t numElements = 10;
};

.bss
struct bss
{
  uint32_t ignore;
};

.tree
expr_UNUSED: *, +, B_AND, +, *, 7, 7, 7, 7, 7, 7, 1, 1, 0, 255, 1, 0, 7,
     ignore

expr_count_less_than_numElement: *, +, <, +, *, 0, 7, 7, 0, 7, 7, count,
     1, 0, numElements, 1, 0, 7, ignore

expr_inc_count: *, +, B_AND, +, *, 0, 7, 7, 7, 7, 7, count, 1, 1, 255,
    1, 0, 0, count

.text
while label1 expr_count_less_than_numElement
        expr_inc_count
label1:
```

Listing B.10: Example SDLP Assembly Language Program

List of References

[1] J. Silc, B. Robic, and T. Ungerer, *Processor Architecture: From Dataflow to Superscalar and Beyond.* Springer Science & Business Media, 1999.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.* Elsevier, 2019.

[3] M. Verma and P. Marwedel, *Advanced Memory Optimization Techniques for Low Power Embedded Processors.* Springer, 2010, vol. 1.

[4] Y. Chu and E. R. Cannont, "Interactive High-Level Language Direct-Execution Microprocessor System," *IEEE Trans. Soft. Eng.*, vol. SE-2, no. 2, pp. 126–134, 1976.

[5] Xilinx. *MicroBlaze Processor Reference Guide.* (Accessed: Dec. 24, 2018). [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf

[6] ARM. *ARM7TDMI (Rev 3) Core Processor Product Overview.* (Accessed: Dec. 24, 2018). [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.dvi0027b/DVI_0027A_ARM7TDMI_PO.pdf

[7] quietpc.com. *Zalman CNPS10X FLEX High Performance CPU Heatsink.* (Accessed: Dec. 23, 2018). [Online]. Available: https://www.quietpc.com/cnps10x-flex

[8] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.* Elsevier, 2012.

[9] J. A. Sharp, *Data Flow Computing.* Ellis Horwood, 1985.

[10] J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester Prototype Dataflow Computer," *Communications of the ACM*, vol. 28, no. 1, pp. 34–52, 1985.

[11] N. C. Audsley and M. Ward, "Syntax-Driven Implementation Of Software Programming Language Control Constructs and Expressions on FPGAs," in *Proceedings of the*

2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems. ACM, 2006, pp. 253–260.

[12] A. Sloss, D. Symes, and C. Wright, *ARM System Developer's Guide: Designing and Optimizing System Software.* Morgan Kaufmann, 2004.

[13] J. L. Baer, *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors.* Cambridge University Press, 2010.

[14] N. R. Council, *The Future of Computing Performance: Game Over or Next Level?*, S. H. Fuller and L. I. Millett, Eds. National Academies Press, 2011. [Online]. Available: https://www.nap.edu/catalog/12980/the-future-of-computing-performance-game-over-or-next-level

[15] Wikipedia. *Amdahl's Law.* (Accessed: Dec. 23, 2018). [Online]. Available: https://en.wikipedia.org/wiki/Amdahl%27s_law

[16] G. E. Moore *et al.*, "Progress In Digital Integrated Electronics," in *Electron Devices Meeting*, vol. 21, 1975, pp. 11–13.

[17] E. Katevenis, "Reduced Instruction Set Computer Architectures for VLSI, PhD Thesis," 1983.

[18] J. E. Nurmi, *Processor Design, System-On-Chip Computing for ASICs and FPGAs*, 2007.

[19] E. D. Rather and E. K. Conklin, *Forth Programmer's Handbook.* BookSurge Publishing, 2007.

[20] C. E. LaForest, "Second-Generation Stack Computer Architecture," B.S. Thesis, University of Waterloo, 2007, (Accessed: Jan. 30, 2019). [Online]. Available: https://en.wikipedia.org/wiki/P-code

[21] P. Koopman, *Stack Computers: The New Wave.* Ellis Horwood, 1989.

[22] G. Coates, "A Hardware Java Virtual Machine for Hard Real Time Systems," MPhil Thesis, University of Manchester Institute of Science and Technology, 2005.

[23] C. Crispin-Bailey. *Implicitly Coded Machines Research Page.* (Accessed: Jan. 18, 2019). [Online]. Available: https://www-users.cs.york.ac.uk/chrisb/main-pages/stack-machines/stack-machines-research-page.html

[24] Oracle. *The Java Virtual Machine Specification.* (Accessed: Dec. 24, 2018). [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se7/html/

[25] B. Venners, *Inside the Java 2 Virtual Machine.* Computing McGraw-Hill, 2000.

[26] Wikipedia. *Manchester Baby.* (Accessed: Jan. 18, 2019). [Online]. Available: https://en.wikipedia.org/wiki/Manchester_Baby

[27] J. B. Dennis, "First Version of a Data Flow Procedure Language," in *Programming Symposium.* Springer, 1974, pp. 362–376.

[28] J. Rumbaugh, "A Data Flow Multiprocessor," *IEEE Transactions on Computers*, vol. 100, no. 2, pp. 138–146, 1977.

[29] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, and W. Yoder, "Scaling to the End of Silicon with EDGE Architectures," *Computer*, vol. 37, no. 7, pp. 44–55, 2004. [Online]. Available: http://www.cs.utexas.edu/users/cart/trips/publications/computer04.pdf

[30] M. Gebhart, B. A. Maher, K. E. Coons, J. Diamond, P. Gratz, M. Marino, N. Ranganathan, B. Robatmili, A. Smith, J. Burrill *et al.*, "An Evaluation of the TRIPS Computer System," in *ACM Sigplan Notices*, vol. 44, no. 3. ACM, 2009, pp. 1–12.

[31] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, and K. S. McKinley, "Compiling for EDGE Architectures," in *Code Generation and Optimization, 2006. CGO 2006. International Symposium on.* IEEE, 2006, pp. 11–pp.

[32] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes.* Elsevier, 2005.

[33] U. De Carlini and U. Villano, *Transputers and Parallel Architectures: Message-Passing Distributed Systems.* Horwood, 1991.

[34] Y. Chu, *High-Level Language Computer Architecture.* Academic Press, 1975.

[35] D. R. Ditzel and D. A. Patterson, "Retrospective on High-Level Language Computer Architecture," in *Proceedings of the 7th annual symposium on Computer Architecture.* ACM, 1980, pp. 97–104.

[36] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference.* ACM, 1967, pp. 483–485.

[37] J. Waldron and O. Harrison, "Analysis of Virtual Machine Stack Frame Usage by Java Methods," in *IMSA*, 1999, pp. 271–274.

[38] J. Whitham and N. Audsley, "Optimal Program Partitioning for Predictable Performance," in *Real-Time Systems (ECRTS), 2012, 24th Euromicro Conference on.* IEEE, 2012, pp. 122–131.

[39] J. Whitham and N. C. Audsley, "Explicit Reservation of Local Memory in a Predictable, Preemptive Multitasking Real-time System," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th.* IEEE, 2012, pp. 3–12.

[40] Wikipedia. *P-Code.* (Accessed: Dec. 24, 2018). [Online]. Available: https://en.wikipedia.org/wiki/P-code

[41] M. D. McCool, A. D. Robison, and J. Reinders, *Structured Parallel Programming: Patterns for Efficient Computation.* Elsevier, 2012.

[42] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's With Very Small Physical Dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[43] K. Olukotun, L. Hammond, and J. Laudon, "Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency," *Synthesis Lectures on Computer Architecture*, vol. 2, no. 1, pp. 1–145, 2007.

[44] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *ACM SIGARCH Computer Architecture News*, vol. 39. ACM, 2011, pp. 365–376.

[45] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on.* IEEE, 2001, pp. 3–14.

[46] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The Gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[47] Micron. *TN-47-04: Calculating Memory System Power for DDR2.* (Accessed: Dec. 24, 2018). [Online]. Available: www.micron.com

[48] V. Tiwari, S. Malik, A. Wolfe, and M. T.-C. Lee, "Instruction Level Power Analysis and Optimization of Software," in *Technologies for Wireless Computing.* Springer, 1996, pp. 139–154.

[49] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2010.

[50] Wikipedia. *GNU Compiler Collection*. (Accessed: Dec. 26, 2018). [Online]. Available: https://en.wikipedia.org/wiki/GNU_Compiler_Collection

[51] J. Shinde and S. S. Salankar, "Clock Gating - A Power Optimizing Technique for VLSI Circuits," in *2011 Annual IEEE India Conference*, Dec 2011, pp. 1–4.

[52] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Prentice Hall, 1988.

[53] Wikipedia. *List of C-Family Programming Languages*. (Accessed: Dec. 24, 2018). [Online]. Available: https://en.wikipedia.org/wiki/List_of_C-family_programming_languages

[54] ——. *C Programming Language*. (Accessed: Dec. 24, 2018). [Online]. Available: https://en.wikipedia.org/wiki/C_(programming_language)

[55] S. Leibson, *Designing SOCs with Configured Cores: Unleashing the Tensilica Xtensa and Diamond Cores (Systems on Silicon)*. Elsevier, 2006.

[56] D. Daniel C. *Introduction to the Programming Language Occam*. (Accessed: Dec. 24, 2018). [Online]. Available: http://www.eg.bucknell.edu/~cs366/occam.pdf

[57] Wikipedia. *Handel-C*. (Accessed: Dec. 24, 2018). [Online]. Available: https://en.wikipedia.org/wiki/Wiki/Handel-C

[58] Accellera. *SystemC*. (Accessed: Dec. 24, 2018). [Online]. Available: http://www.accellera.org/downloads/standards/systemc

[59] GNU. *Options That Control Optimization*. (Accessed: Dec. 29, 2018). [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[60] ARM. *Procedure Call Standard for the ARM Architecture*. (Accessed: Dec. 24, 2018). [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042f/IHI0042F_aapcs.pdf

[61] Wikipedia. *Executable and Linkable Format (ELF)*. (Accessed: Dec. 24, 2018). [Online]. Available: https://elinux.org/Executable_and_Linkable_Format_(ELF)