2009

SEPARATING INSTRUCTION FETCHES FROM MEMORY ACCESSES : ILAR (INSTRUCTION LINE ASSOCIATIVE REGISTERS)

Nien Yi Lim

University of Kentucky, nien.yi@uky.edu

Abstract of thesis

SEPARATING INSTRUCTION FETCHES FROM MEMORY ACCESSES : ILAR (INSTRUCTION LINE ASSOCIATIVE REGISTERS)

Due to the growing mismatch between processor performance and memory latency, many dynamic mechanisms which are “invisible” to the user have been proposed: for example, trace caches and automatic pre-fetch units. However, these dynamic mechanisms have become inadequate due to implicit memory accesses that have become so expensive. On the other hand, compiler-visible mechanisms like SWAR (SIMD Within A Register) and LARs (Line Associative Registers) are potentially more effective at improving data access performance. This thesis investigates applying the same ideas to improve instruction access.

ILAR (Instruction LARs) store instructions in wide registers. Instruction blocks are explicitly loaded into ILAR, using block compression to enhance memory bandwidth. The control flow of the program then refers to instructions directly by their position within an ILAR, rather than by lengthy memory addresses. Because instructions are accessed directly from within registers, there is no implicit instruction fetch from memory. This thesis proposes an instruction set architecture for ILAR, investigates a mechanism to load ILAR using the best available block compression algorithm and also develop hardware descriptions for both ILAR and a conventional memory cache model so that performance comparisons could be made on the instruction fetch stage.

KEYWORDS: Memory latency, CRegs (Cache Registers), SWAR (SIMD Within a Register), LARs (Line Associative Registers), Searching block compression algorithm using a GA (Genetic algorithm).

Nien Yi Lim

11/23/2009

SEPARATING INSTRUCTION FETCHES FROM MEMORY ACCESSES : ILAR (INSTRUCTION LINE ASSOCIATIVE REGISTERS)

By

Nien Yi Lim

Dr. Hank Dietz

(Director of thesis)

Dr. Stephen Gedney

(Director of Graduate Studies)

11/23/2009

(Date)

RULES FOR THR USE OF THESIS

Unpublished theses submitted for the Master's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgments.

Extensive copying or publication of the thesis in whole or in part also requires the consent of the Dean of the Graduate School of the University of Kentucky.

A library that borrows this thesis for use by its patrons is expected to secure the signature of each user.

Name	Date

THESIS

Nien Yi Lim

The Graduate School
University of Kentucky
2009

SEPARATING INSTRUCTION FETCHES FROM MEMORY ACCESSES : ILAR (INSTRUCTION LINE ASSOCIATIVE REGISTERS)

THESIS

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science in the
College of Engineering
at the University of Kentucky

By

Nien Yi Lim

Director: Dr. Hank Dietz, James F. Hardymon Chair in Networking
Professor of Electrical and Computer Engineering
Lexington, Kentucky

2009

This thesis is dedicated to Ching Joo Khor, my mom and dad

ACKNOWLEDGEMENTS

This thesis would not be possible without the support and guidance from people around me. First of all, I want to thank my adviser Dr. Hank Dietz for his timely and insightful advises. I would also like to extend my gratitude to Dr. Robert Heath for words of wisdom and support throughout my graduate school curriculum. In addition to that, I would also like to acknowledge Dr. Meikang Qiu for his willingness to serve on my defense committee. Without the advises from my thesis committee, I would not have been able to overcome technical barriers and keep myself on track for graduation.

Secondly, I would like to thank Ching Joo Khor for her understanding and unconditional support while working on my thesis. Next, I would also like to thank my parents: Piang Lim and Shan Shan Ng for their unconditional guidance, financial support and for giving me the opportunity to pursue my MS degree at the University of Kentucky. I really appreciate the emotional support and space given from them to concentrate on my thesis.

Thirdly, I would also like to thanks my friends in Kentucky who I shared many happy memories together during the semester. Furthermore, I would also like to thank the staffs at University of Kentucky Department of Electrical and Computer Engineering who provided a good working environment and who helped in scheduling of my thesis defense.

Without the guidance and encouragement from all of the parties above, I would not have gotten this far.

Table of Contents

List of Tables

List of Figures

List of Files

MS_Thesis_Nien_Yi_Lim.pdf .....640 KB
Sofware_Simulator.zip .....36 KB
Hardware_Model.zip ..... 308 KB

1. Introduction

Over many years, the amount of computation that can be executed by a processor in a fixed period of time has steadily increased. This steady increase has been driven by both speed and density improvements in circuits fabricated using silicon. The density improvements are more significant, and they also are the trend predicted by Moore's law. In 2008, Intel launched Hafnium-based 45nm high-k metal gate silicon technology in it's Core architectures [1] . In 2009, Intel now is launching a new 32nm logic technology. There is no doubt that tomorrow's chips will hold many more, yet smaller, transistors.

However, the performance of a computing system is not directly a function of how many transistors are available to build it, nor even of how quickly those transistors can switch. The key to steady improvement of performance is balance. Not only must the processor be fast, but the rest of the system must be able to support it running at that speed. For example, buses, interconnection networks, and especially memory must be able to meet the processor's demands. Achieving this balance is the focus of modern computer architecture and the concern addressed in this thesis.

Specifically, improvements in memory performance have not been as rapid as improvement in the logic circuitry. This growing deficit is visible in two main ways. First, access to data stored in main memory has become prohibitively expensive, so it has become necessary to invent mechanisms that can reduce the frequency and impact of such access. Second, frequent fetching of instructions from main memory has become impractical. Whereas most recent architectural proposals focus on the first problem, this thesis attacks the second: literally, this thesis is attempting to provide an alternative to the conventional Von Neumann instruction fetch model.

A conventional Von Neumann instruction fetch model consists of mechanisms that enable sequential fetching of instructions. Since the 70s, computer scientists have realized that main memory accesses are slow and therefore created a second level of memory containing buffer-like structures called instruction caches. Instruction caches are intended to store frequently used instructions so that future instruction references can be accessed directly from the caches. Nevertheless, although memory systems have become

larger in capacity, the relative bandwidth versus latency improvement of processors has outperformed memory systems and this trend continues to be true until today. In fact, the exponential increase in the performance of processors have produced a wider and wider gap between them.

As the speed of processors continues to increase, one big problem with the Von Neumann model is that memory latency is not reduced if instructions are not found in instruction caches (a cache "miss"). Also, addresses in caches are memory addresses and control transfer instructions, like branch instructions will thus require accesses to main memory addresses. These two main problems form the motivation for this thesis, where a new instruction fetch mechanism needs to be develop to meet future needs of computing performance.

1.1 Motivation

As the improvement in latencies of memory continue to lag behind the bandwidth of processor chips, computer designers faced a "memory wall" in designing a balanced system. To hide the growing mismatch between processor and memory, a variety of processor and memory architectures have been proposed. All of these approaches so far tend to be "invisible" to users and system software, implemented dynamically at run time by hardware. For example, multilevel caches, automatic prefetches, multithreading and different bus interfaces and protocols are all intended to keep the programmer's model intact. However, it has become inherently apparent that classical sequential execution of the Von Neumann architecture has become not suitable in a world where random access to memory is extremely expensive.

Improvements to the Von Neumann architecture of having a processing unit and a single storage of data and instructions have been slow and painful. So, although this problem has been around for a while, changes in architecture and the programming model haven been slow to catch on. Despite that, the 1990s saw the introduction of the SWAR (SIMD Within a Register) concept. [3][4][5] Other architecture models that are designed to reduce fetches from memory for example, CRegs (Cache Registers) did not catch on as fast as the SWAR concepts. Nevertheless, like SWAR, it also tries to minimize the number of memory accesses by combining the functionality of caches and registers. [6]

Combining the benefits of both of these concepts, this thesis proposes an instruction fetch mechanism that is able to reduce memory latency by limiting memory accesses. The following sections provide a survey of the current memory models and the background on the concepts behind this new instruction fetch mechanism: SWAR and CRegs.

1.1.1 Survey of current memory models

Many researchers have acknowledged that memory latency is hindering the overall improvements in processor speed. As a result, many journal and conference publications have been generated to address the issue mainly based on improvements in the Von Neumann architecture. This section discusses a survey carried out on instruction fetch models that attempts to minimize latency.

Eyerman and Eeckhout [7] proposes a smart fetch policy to exploit MLP (Memory-Level Parallelism) in SMT (Simultaneous Multi-threading) processors. This fetch policy makes decisions on whether the threads should be allocated memory resources during long latency loads to make use of MLP: No additional resources are allocated to long latency loads which have no MLP, in the case where MLP is present, just-enough resources are allocated. The MLP-aware fetch policy enables other threads to use the spare resources and improve performance.

Other research focuses on developing novel fetch policies to control performances of threads in SMT processors. The proposed fetch policy tries to minimize the effects of L2 caches by introducing multiple fetch priorities. [8] Threads are assigned priorities based on their behavior in cache. In a later paper, the authors proposed another fetch policy that adjusts fetch priorities by comparing threads, where time critical threads get higher priority. [9] At the same time, this fetch policy tries to maximize the throughput of the non-critical threads by implementing predictable performances for critical threads.

Other proposed solutions to improve performance on SMT processor architectures include introducing a dynamically allocated “ready thread buffer”. [10] The solution is divided into two stages: the first stage estimates the confidence level for each of the possible branches and marks the threshold for each branch. The second stage applies a

fetch mechanism based on the marked threshold and the “ready thread buffer” is then used to manage instructions from threads with different confidence levels.

In addition to that, researchers have proposed instruction fetch schemes to run on superscalar processors, where multiple instruction could be processed in a single cycle. This proposed scheme contain a “flag-in-cache” where a flag contained in the instruction cache is used together with a instruction branch prediction scheme to increase fetch efficiency. [11] This fetching scheme not only decreases the time required for parallel execution checks, it also helps increase the accuracy of instruction pre-fetches in superscalar processors.

Other investigations on out-of-order instruction fetch in superscalar processors include trying to increase the instruction fetch bandwidth, efficient use of available ILP (Instruction Level Parallelism) and accurate branch predictions. [12] In order to do so, empirical models of super-scalar processors were made and to double the performance, the conclusion obtained was to double the fetch rate and decrease by four-fold the mis-predicted branches.

A group of researchers also propose an extension to the classical Von Neumann architecture by using “Instruction Fetch Registers” to improve access to frequently occurring instructions. [13] Instruction Fetch Registers are used as a complementary technique with instruction caches to minimize bottlenecks and to provide additional fetch bandwidth. Compiler technology is used to pack an application's instructions resulting in decreased code size, better execution time and a smaller memory footprint in instruction caches. Another approach is to subdivide the instruction cache into categories based on execution frequency. [14][15] Frequently executed sections of code are placed into smaller and lower powered cache to handle energy requirements. The splitting for the lookup of the different categories in cache reduces the miss rate.

Besides these work, researchers have look to improve existing instruction pre-fetching techniques by attacking the flaws in the design. One research group look to improve the branch prediction bandwidth so that accurate pre-fetching of instructions could take place. [16] A mechanism called “Temporal Instruction Fetch Streaming” is used to pre-fetch temporally-correlated instruction streams from lower-level caches.

Rather than exploring a program's control flow graph, this mechanism predicts future instruction cache misses directly by recording and replaying recurring L1 instruction miss sequences.

1.1.2 Background on SWAR and CRegs

The SWAR model as described in the thesis of Fisher [4] uses SIMD's (Single Instruction stream, Multiple Data stream) concept of data parallelism in a single CPU register. SIMD is a processing model that exploits data parallelism by executing one instruction across as many data points as possible.

SWAR allows “micro-parallelism” to be executed within multiple data stored in a register where all of these data are manipulated by a single instruction stream. This creates a general-purpose programming model of registers, which allows sub-word processing whenever a data consists of bits that are less than a full machine word. SWAR are largely driven by memory performances and many data objects, especially those associated with multimedia processing are much smaller than the natural word size and datapath widths used in modern processors. By adding the ability to perform SIMD-like operations on fields within a register or datapath, SWAR operations replaces a series of memory accesses and field extraction/insertion operations with a single access for a word's worth of fields. SWAR concept provides so many benefits that most modern processors include some form of SWAR instructions.

Another concept worth mentioning is CRegs. CRegs have been introduced in the late 1980's and it combines the hardware of both a conventional cache and a register to create a new memory structure. [6] CRegs are used to replace cache hardware and this allows ambiguously aliased names to be grouped together. Therefore, this results in a more efficient execution of instructions than even the combination of conventional caches and registers.

Ambiguously aliased variables cannot be placed efficiently in registers because of the limited number and the need of constant flushing of registers. One might think that this can be resolved by placing them in caches. However, caches can be ineffective too when dealing with ambiguously aliased variables. This is because references to other

objects might have addresses hashing to the same cache line, thus there is a possibility of overwriting the desired object from cache. Therefore, the combination of the functionality of caches and registers in CRegs hardware allows value to be buffered, ambiguously aliased or not. Besides maintaining the full benefits of a register, it also allows short name for addresses, therefore reducing the instruction-fetch bandwidth latency. The ambiguously aliased problem does not apply to CRegs because CRegs associatively updates names of fields that match.

Despite the obvious benefits of CRegs, the adoption of CRegs were impeded by the need of a specialized CRegs Instruction Set Architecture (ISA). Furthermore, unlike caches, it does not utilize spatial locality. Therefore, this call for a new hardware model that addresses these issues. By adopting SWAR concepts within CRegs, a new model has been proposed, named as LARs (Line Associative Registers) The following section describes the background behind this new model.

1.1.3 Background on LARs

LARs is a new memory access model proposed in Melarkode's thesis in 2004. [7] It combines the concepts of SWAR operations on long lines into the cache-like associativity of CRegs hardware. Due to it's similarity with conventional caches, only minor ISA modifications need to be made to an architecture for it to fully utilize the benefits of LARs.

LARs hardware inherits all the benefits from CRegs and also reap spatial locality benefits by utilizing SWAR concepts of having long lines. Besides that, by having a wide width, the number of references to memory is reduced, which in turn improves the memory bandwidth. LARs is separated similar to the Harvard memory architecture, where it is divided into Data LARs (DLAR) and Instruction LARs (ILAR). Conveniently, it separates the storage for data and instructions.

LARs in general are useful because they are fully associative where the hardware is explicitly told to load which specific entry. Although the uses of DLAR and ILAR are totally different, the hardware layout differences between DLAR and ILAR are minimal, with the former hardware having two additional fields of “Type” and “Status”. One

benefit of DLAR is that they are organized in “lines” where vector operations can be performed easily with minimum memory references. [17] Also, DLAR record the current data position within it's data field. Therefore, the “Type” field in each DLAR contains the current object data type. Since data are type tagged in DLAR, the benefit is that the type information does not need to be encoded in scalar arithmetic operations.

On the other hand, ILAR is derived directly from SWAR and CRegs by applying the data concepts to instruction fetches. Figure 1 shows how the ILAR hardware differs from conventional registers, caches and CRegs. Melarkode's thesis provides an introduction to ILAR hardware, but no mechanisms for fetching instructions using them were proposed. The motivation behind having an ILAR hardware is that it totally removes the implicit fetch from memory that normally is associated with each instruction being processed, instead using an explicit fetch instruction that is able to load a specific number of ILAR.

The other motivation of ILAR is that it uses instruction positions within ILAR as addresses. Although the layout of ILAR as shown in Figure 1 is similar to instruction CRegs, instruction CRegs are designed in the namespace of instruction addresses in memory, whereas ILAR contains “Obj. Addr.” that are local addresses within that ILAR. Furthermore, in an ILAR, “Obj. Addr.” are relatively short compared to “Line Addr.” of a Cache. Short “Obj. Addr.” provides ILAR the means to contain local offsets to instructions. Hence, if a block that is already in another ILAR is being requested, the

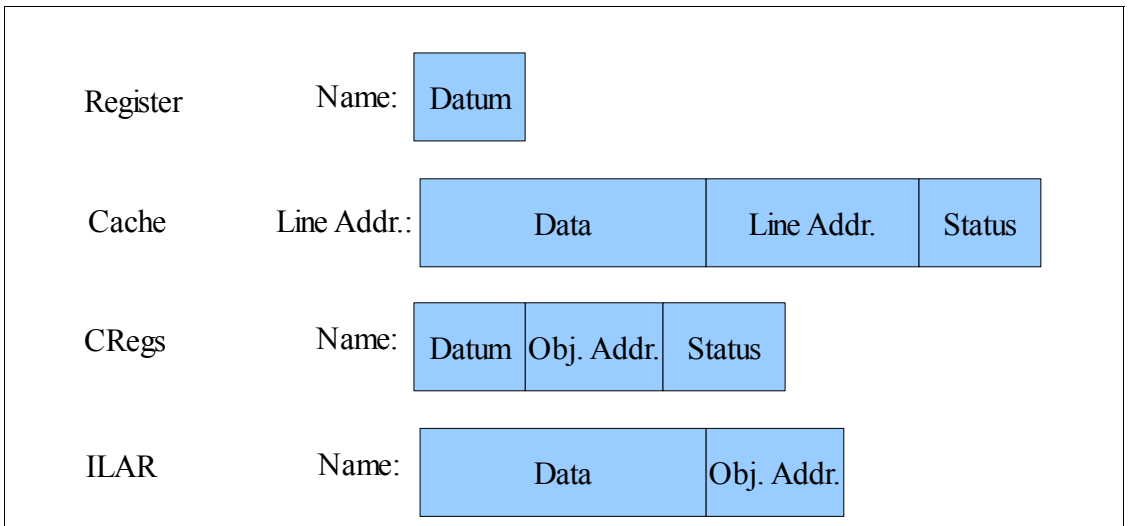

Figure 1: Comparison with ILAR hardware

block is logically copied without any additional memory activity. The motivation of having ILAR hardware also can be seen when there is control flow instructions, for example, branch instructions. Branch targets are specified as local offsets within an ILAR. Thus, by doing so, it reduces the overall memory footprint, improves the utilization of memory bandwidth and also completely remove the latency of "misses" during instruction processing.

Having discussed the motivation behind ILAR, how do one efficiently fetch instructions into ILAR? Since ILAR has wide widths, it will take a long time for us to fetch one instruction at a time. Instead of fetching single instructions, an efficient way is to fetch blocks of instructions that can populate the ILAR. Nevertheless, the block size is directly proportional to the number of instructions in ILAR and can become very large. Therefore, there needs to be a compression scheme in place where instructions are decompressed from main memory before being fetched into the ILAR hardware.

1.1.4 Compression and prior work

Code density in computing is an area of research that has been looked into seriously, especially by embedded application developers. For example, a smaller code size translates to less physical storage and a more portable device. [18] Logically, as the architecture becomes more and more complex, the code size is also increased. Thus, in order to achieve better code density, many developers have employed some form of compression in their designs.

In recent years, especially in the "memory wall" era, it is essential to have a small code size with efficient memory accesses because memory accesses are so expensive. Therefore, many developers for HPC systems have started to take code density seriously by using compression techniques to help reduce the instruction space occupied and improve the memory fetch bandwidth. Compression for HPC are generally divided into two large categories: instruction set independent compression methods and instruction set specific compression methods. Instruction set independent compression methods, for example PPM (Prediction by Partial Match) technique uses previous symbols in the uncompressed stream to predict the next symbol.

Possible instruction set independent compression schemes can be derived from well known lossless compression techniques. There are many forms of lossless data (instruction) compression techniques and are usually dictionary or/and frequency based. For example, a well known compression algorithm that is dictionary based is the Lempel-Ziv (LZ) compression algorithm. The LZ77 and LZ78 are universal algorithms that do not require priori knowledge of the input source characteristics because the words are compressed by building dictionaries based on the input source bits. [19] LZ77 build the dictionary to encode future codewords based on previous outputs. Whereas LZ78 forward scans the input buffer by adding new words into the dictionary. Popular variations of the LZ algorithm like the Lempel-Ziv-Welch (LZW) also uses similar concepts where words or part of the words are replaced by longest entries in the dictionary. The dictionary is then grown by adding in partly compressed words. Therefore, larger chunks of string could be encoded in future replacements. All of these LZ algorithms provide efficient and universal methods that could be applied for a wide range of applications that are suitable for source bit based compressions.

Another popular source coding technique is Huffman encoding. Huffman encoding is an entropy or frequency coding based lossless data compression method. It is independent of the characteristics of the source and utilizes a variable length code table to encode the source symbols. These encodings are also known as “prefix codes” where the encoding represents common symbols with shorter bit strings than less common source symbols. The common implementation of Huffman encoding is by building a binary tree using a bottom up approach. [20] A sorted queue is created with nodes and internal nodes which have ascending probabilities. The two nodes which has the least probability are used to form a new internal node which has the sum of both of the child's probabilities. This process is repeated until the probability of unity is achieved. Subsequently, parent nodes and child nodes are assign encoding bits. The common encoding notation used is “0” for the left child and “1” for the right child. Huffman encoding could be implemented in linear time where the the time is proportional to the n size of the input, O(n) or could be implemented as logarithmic time, O(log n). Since Huffman encoding uses variable bit length encoding, decoding blocks of compressed texts will require a frequency look up table that is stored efficiently with the text.

On the other hand, some compression methods are developed to target certain instruction sets. For example the Thumb instruction set in the ARM7TDMI processors balances between code density and performance by extracting only the most commonly used instructions from the ARM instruction set. By doing so, Thumb compresses the original 32-bit instructions down to 16 bits. Besides that, it also provides the interoperability between the compressed and original instruction sets to retain full functionality. Nevertheless, each instructions are compressed individually and are limited only to instructions from the ARM instruction set. [21] Besides that, not all ARM instructions have Thumb equivalents, so some ARM instruction needs to be called before returning to the Thumb code. [24]

Other examples of instruction set dependent compression algorithm includes the compression for Intel's iAPX432 instruction set. This instruction sets has compression to encode all of its 200 over variable length, four field instructions. [22] The first two field which are the class and format field specifies the number of operands in each instruction and how they should be accessed. The third field is the reference field which specifies the logical addresses for its operands, if any. The last field is a optional opcode which is Huffman encoded to determine the instruction's operator. These variable length instructions are read in from memory as 32 bit length streams for decoding and one of the drawbacks is the complexity of the decoding unit for the compressed instruction set. Due to the decoding complexity, the iAPX processor has 3 separate chips, one for instruction fetch and decode, one for execution and one for interface processing. Having 3 separate chips make the iAPX design very hardware intensive for fetching and decoding instructions.

Due to the benefits of compression, many researchers have employed compression methods in the instruction fetch model itself when dealing with architectures that require a substantial amount of memory accesses. Taking the popular memory cache model as an example, compressed instructions could either be stored in main memory or in caches.

An example of using compression in the instruction fetch model is the Compressed Code RISC Processor (CCRP) proposed by Wolfe and Chanin. [23] This

processor model keeps all the benefits of a RISC processor including pipelining and also provides denser instruction storage. This processor model keeps the programmer's model intact by utilizing traditional RISC compiler and linker to generate the object code. The object code is then compressed and stored in the instruction memory. At run time, the compressed instructions are decompressed to fill an empty cache line or during a cache miss. The main benefit of the CCRP model is that it keeps the programmer's model intact so that the original optimizing compilers could still be used and it also improves the instruction fetch bandwidth.

On the other hand, another method to reduce the instruction fetch bandwidth is by utilizing a specialized cache called a trace cache proposed by Rotenburg, Bennett and Smith. [25] Trace caches increases the instruction fetch bandwidth of processors thus has been incorporated into the Pentium 4 architecture and newer architectures like the Pentium Itanium 2. A trace cache stores traces of decoded instructions which include taken branches, therefore allowing fetching of multiple blocks without considering branches in the execution flow. Trace caches make use of temporal locality to predict branch behavior and relies on dynamic sequences of code to be reused. Therefore, in this model the cache stores compressed traces of instructions.

Nevertheless, since compression methods described for the instruction fetch models are used in conventional memory-cache models, fetch instructions are implicit and additional memory latency would be introduced on cache “misses”. Therefore, there needs to be a new instruction fetch mechanism that minimizes memory latency and still maintains code density. My thesis proposes that lLAR is used and also describes a mechanism to decompress blocks of compressed instructions needed to load ILAR. Using ILAR, all instruction fetches are explicit, which allows fetch instructions to be rescheduled if needed. Furthermore, instructions are accessed in ILAR namespace, not by memory addresses, therefore this minimizes memory activity. The following section describes the work carried out in this thesis.

1.1.5 My thesis and block compression

My thesis introduces ILAR and defines an instruction set for the hardware model. In addition to that, an instruction fetch mechanism that can utilize the instruction sets

defined is also proposed. A block compression algorithm is used to compress instructions in main memory and decompression takes place when a fetch instruction is received to load the ILAR. After loading the ILAR, instructions are referenced using the ILAR number and position of the instructions within the ILAR.

Block compression is not a new method of compressing information. Mobile communication systems uses block coding schemes to maintain a suitable information rate within the channel capacity. Unlike source coding schemes like Huffman encoding, block compression is a fixed length encoding scheme which encodes a fixed set of messages with a fixed set of bits.

Therefore, by having a finite number of encoding bits, a simple decoding hardware for block compression schemes can be developed. Besides that, by knowing the bit boundaries, the decoding process will be fast as many decoding blocks could be processed at the same time. These concepts form some of the pre-requisites for the block compression algorithm developed in this thesis. The first requirement is that instructions should be vertically encoded to safe instruction bits in the compressed block. Vertical encoding encodes decompressing information in the compressed blocks. Although it requires extra decoding logic to decode this information, the benefits of having a better compression outweighs the cons.

The second requirement is that the compression scheme developed has to be implemented on a simple decompression hardware to allow decompression to be performed near to constant time. Decompression near constant time does not depends on the number of instructions in the compressed blocks. Therefore, to be able to do so is vital because extra latency will be introduced for decompression if it is done otherwise.

The third requirement is that there needs to be constant block size for the compression so that a constant compression rate could be achieved. Choosing this constant block size is important because very large block sizes causes redundancy and thus decreases the compression rate.

As block compression is a fixed length compression scheme, the compression algorithm used depends on how well the algorithm is able to compress a given sequence of input instructions. Therefore, to obtain a good compression rate, there needs to be a

way for us to determine the best compression algorithm for a given set of input instructions. My thesis here developed a search algorithm to determine the best available compression algorithm for a given input instruction sequence.

Having determined the best compression algorithm, the next step is to determine whether it is easily decompress-able. A easy way to prototype this is to build a FPGA hardware description of the decompression hardware. This hardware model is also useful for comparing with a conventional memory-cache model hierarchy to clearly showcase the benefits of the new instruction fetch mechanism using ILAR. Here is a summary of the objectives of my thesis:

1. Propose a new instruction fetch mechanism using ILAR hardware and define instruction sets needed for executing using this model.
2. Develop a set of block compression algorithms that could potentially be used to compress a given input instruction sequence.
3. Develop a search method that could determine the best compression algorithm given an input instruction sequence.
4. Select one of the best compression algorithm to build a hardware description of the decompression hardware.
5. Build hardware descriptions for ILAR and a conventional memory-cache model so that performance comparisons could be made.

1.2 My thesis organization

There are two major pieces to my thesis, the first piece discusses the block compression algorithm design and development. The remaining piece describes hardware prototypes for the models used and also the results obtained. Based on these two pieces, my thesis is divided into seven chapters.

Chapter one gives an introduction of the background related to the proposal of ILAR. Also, it provides the motivation behind ILAR and also the objectives of this thesis. Chapter two describes the benefits and issues of ILAR. The instruction sets defined for ILAR are also described in this chapter. Subsequently in chapter three, the block

compression algorithms design and customizations available are discussed. Continuing in chapter four, a software simulator written for developing the block compression algorithms is described and the best block compression method is determined by using a genetic algorithm.

Chapter five describes the hardware description developed for ILAR, a memory cache model and the block decompression module. Chapter six then discusses the results obtained from the hardware modules and also analyses the performance of ILAR. Chapter seven concludes and present future work in this area.

2. Overview and benefits of ILAR

As described in the introduction chapter, LARs extends the concepts of SWAR and CRegs to allow the direct manipulation of data and instruction objects within wide registers. Following Harvard memory architecture of separating memory into data and instruction memory , LARs can be divided into DLAR and ILAR. Both of these proposed memory models minimizes memory references to reap the benefits of having wide registers.

Besides reaping the benefits of having a wide width to work with, data in DLAR are type-tagged to absorb the type conversion latencies found in most architectures. Furthermore, DLAR have an address field that increases the data fetch bandwidth by eliminating ambiguous aliasing and the need to associatively update other DLAR. The discussion of DLAR here is used as an introduction to the fields contained in data instructions in the instruction set. Since DLAR manipulates data in the proposed architecture, it will not be discussed further in this thesis because the focus here is primarily on the instruction fetch/decode stages.

2.1 Benefits of ILAR

ILAR can be considered as a group of register files used to store instructions. However, ILAR differs from traditional registers or caches because it eliminates the traditional instruction fetches, making the process of getting instructions from main memory independently controllable by the compiler.

Table 1 shows the overall structure of an arbitrary sized, thirty two (I0 - I31) 1024 bits wide ILAR. Each line of an ILAR has an immediate address field which corresponds to the address in main memory. Instructions in an ILAR can be accessed by local offsets from the immediate address. In addition to that, control transfer instructions, for example branch instructions, have target addresses which uses these local offsets. The overall benefit is less memory accesses during program execution and an efficient instruction fetch process where instructions are read using local ILAR addresses.

Table 1: ILAR Structure

Address	Instructions				
I0					
I1					
I2					
I3			**32 bit Instr.**		
I...					
I...					
I31					

Loading instructions into ILAR is also a simple process. Comparing to caches, instead of loading or replacing one cache line at a time, ILAR has the ability to pre-load several ILAR worth of instructions. As this allows the instruction fetch process to proceed without additional memory accesses, it completely removes the latency of "misses" from the instruction fetch process. Besides that, ILAR takes advantage of having instructions in "registers", as this allows immediate access to instructions, improving upon existing pre-fetch designs.

As addresses in ILAR are local offsets, lengthy memory addresses that are present in existing caches could be avoided. Furthermore, if a load request a block that is already in another ILAR, the decoded instruction block is logically copied without any memory activity. This also contributes to less memory latency.

2.2 Issues in ILAR

ILAR is new low memory latency model that can be use to replace the existing instruction fetch process. However, there are certain issues with ILAR that needs to be overcome for it to reach it's full potential.

The first issue with ILAR is that a new ISA needs to be defined. In general, whenever a new architecture is defined, it requires a new ISA. Fortunately for ILAR, since only a new instruction fetch process is being proposed, most of the instruction sets stay the same. A new instruction set that is needed to load the ILAR has to be defined. Also, branch instructions need to be modified to be able to contain local addresses of

ILAR.

ILAR requires a new instruction fetch mechanism to be defined. In order for the instruction fetch to work, ILAR needs to be loaded. Loading the ILAR with instructions can be a slow process given that ILAR has wide widths. Therefore, this issue could be resolved by employing a compression scheme able to compress instructions in main memory at compile time. During run time, these compressed block are decompressed to load the required ILAR with instructions. Therefore, an efficient loading of ILAR could take place.

2.3 ISA (Instruction Set Architecture) for ILAR

An instruction set is defined to be executed on the described LARs hardware. ISA presented in this section contains instruction sets for fetching to and branching within an ILAR. The summary of the instructions sets for ILAR are shown in Table 2 .

The ISA for LARs are divided into five large categories. *Data Transfer, Type Conversion, Arithmetic and Logical* instruction sets are used to manipulate data in DLAR. These instruction sets are presented at the end of this chapter to give an idea of the instructions available to be loaded into the ILAR. Whereas, *Control Transfer* and *Fetch* instruction sets are used for branching in execution and loading the ILAR respectively These two categories of instruction sets have unique opcodes associated with them. The *Control Transfer* and *Fetch* instruction sets will be discussed in detail in the following sections.

Table 2: Summary of the ISA for ILAR.

Category	Mnemonics	No. of different opcodes	Description
Control Transfer	*SELECT*	1	Selects one of the two ILAR and branches execution to the given offset
Fetch	*FETCH*	1	Fetches blocks of instructions at the given offset to populate the ILAR.

2.3.1 Instruction fetch

Many researchers try to minimize the latency of fetching instructions by dynamically keeping track of instruction sequences so that fewer fetches from main memory are made. Logically, a wider register or cache will provide better instruction throughput but will introduce additional latency.

However, with the support of compiler technology, compressed instruction blocks can be fetched from main memory to populate a line of cache registers. Therefore, the proposed design provides a good throughput and reduces memory latency without introducing additional latency to the fetch cycle. The proposed solution is described in the following sections.

Opcode [31:27]	DEST LAR [26:22]	SRC1 [21:17]	SRC2 [16:12]	NUM [11:10]	IMMEDIATE [9:0]

Figure 2: Fetch Instruction

Figure 2 shows the format of a `Fetch` instruction. The `Fetch` instruction loads the ILAR after decompressing the instructions stored in the instruction memory. The destination LAR field specifies which ILAR to load the instructions to. The SRC1 field specifies the data pointed to by data LAR and SRC2 refers to the address pointed to by the instruction LAR. The NUM field specifies the number of contiguous ILARs to load. IMMEDIATE is a 10 bit immediate field that can be assigned an integer value between 0 and 1024.

The first step after receiving an instruction fetch is to calculate the effective address. Next, the calculated effective address of the destination ILAR is compared with all other ILARs. If there is an address match, the processor cancels the load from memory, if not, it will load instructions from instruction memory. The effective address of the ILAR will always be multiples of 32 since there will be 32 instructions in each 1024 bit wide ILAR. (32 x instruction length of 32 bits). This instruction can be better understood with the help of the following example.

```
Fetch i2, d3, i5, 990
```

The effective address is calculated in the same way as the load and store

instructions:

Effective address of ILAR = SRC1. Address + SRC2.Data + Immediate value

990 = 1111011110, which is the combination of the NUM and IMMEDIATE field.

Therefore, the 10 bit Immediate field will be (11011110)2 = 222 with the two most significant bits being the NUM field which tells the processor to load 4 contiguous ILAR lines starting from the effective address calculated. Since the effective address to load from memory needs to be multiple of 32, the address to fetch from memory is rounded down to 192. The number of ILARs to be loaded is decoded as described in Table 3.

Table 3: Number of bits field decoding

Bits [11:10]	# of ILAR loaded
00	0
01	1
10	2
11	3

In the above example, 4 ILARs will be loaded. The effective address eff, eff+32, eff+64 and eff+96 will be compared to all the existing ILARs to determine if a load from memory has already occurred. Instructions are copied locally for effective addresses that are found in existing ILARs. Otherwise, a fetch from Instruction memory will be initiated. The operation is similar to how a conventional cache hits and misses is handled. The table below shows an example of the ILARs after an instruction fetch.

Table 4: Example of instruction fetch

ILAR	Effective Address	Instructions
...	...	...
I2:	992	Instruction 992 to 1023
I3:	1024	Instruction 1024 to 1055
I4:	1056	Instruction 1056 to 1087
I5:	1088	Instruction 1088 to 1119
...	...	...
I31:	...	...

2.3.2 Control transfer instruction

The following figure shows the format of the `Select` instruction. This instruction handles all the branch requests in a program.

Opcode [31:27]	DEST LAR [26:22]	SRC1 LAR [21:17]	SRC2 LAR [16:12]	OFFSET [11:0]

Figure 3: Select Instruction

The field DEST LAR is the destination address port of the data LAR. SRC1 and SRC2 are the instruction LAR pointers which point to the starting addresses of one of the instruction LARs (ILARs). Offset field specifies the offset of the instruction in that particular ILAR. Therefore, a `Select` instruction could let the program execution jump to any location inside the ILARs. The DEST LAR field points to one of the DATA LARs and the data of this particular LAR is checked for zero value. If the data in the Data LAR equals to zero, the instructions in ILAR1 will be executed. Otherwise, the instructions in ILAR2 will be executed instead.

```
Select d0, i2, i3, 30, 25
```

The example describes that if the data in the Data LAR equals zero, instructions in ILAR i2 starting from the 30th instruction (bits 960) will be executed. Otherwise, instructions in ILAR i3 starting from the 25th instruction (bits 800) will be executed.

2.4 Remaining instruction sets

Although the basic instruction set for data manipulation using DLAR is somewhat

separable from instruction fetch using ILAR, the compression methods and their effectiveness is in part a function of the DLAR instruction encoding. Table 5 shows the remaining instructions sets defined for DLAR and gives an insight of the fields needed to be manipulated in the compression method.

For *Data Transfer* and *Type Conversion* instruction sets, each of the opcodes represent operations on byte (B), half-word (HW), word (W) and double-word (D) respectively. For *Arithmetic and Logical* instruction sets, there are only one opcode for each operation as additional bits in in instructions are allocated for decoding purposes. One bit is used to differentiate between vector or scalar operation, whereas another bit is used to differentiate between modular or saturation arithmetic.

Table 5: Summary of ISA for DLAR

Category	**Mnemonics**	**No. of different opcodes**	**Description**
Data Transfer	*LOADU[B,HW,W,D]*	4	Load unsigned [byte, half-word, word, double]
	LOADS[B,HW,W,D]	4	Load signed [byte, half-word, word, double]
Type Conversion	*STOREU[B,HW,W,D]*	4	Change the Type, Size and Address information to unsigned [byte, half-word, word, double]
	STORES[B,HW,W,D]	4	Change the Type, Size and Address information to signed [byte, half-word, word, double]
Arithmetic and Logical	*ADDVM, ADDVS, ADDSM, ADDSS*	1	Add vector or scalar data with Modular or Saturation arithmetic
	SUBVM, SUBVS, SUBSM, SUBSS	1	Subtract vector or scalar data with Modular or Saturation arithmetic
	ANDS, ANDV	1	And vector or scalar data
	ORS, ORV	1	Or vector or scalar data
	EXORS, EXORV	1	Exor vector or scalar data

3. Block compression algorithm and design

Block compression is a fixed length compression method that could be used in our proposed instruction fetch mechanism using ILAR. Since ILAR has wide widths, loading them would be slow and inefficient. Therefore, block compression is proposed as the method to compress instructions in main memory and during a `Fetch` instruction, the compressed instruction blocks are decompressed to load the specified ILAR.

In this section, the block algorithms proposed and the design customization of each of them is described in detail.

3.1 Compression algorithms format

The compression algorithms are developed based on the concept of compressing instructions in blocks. Each block compresses instructions that could potentially be used to load one ILAR. Since block compression is a fixed length compression scheme, a constant compression rate will be achieved. So, by determining what compression rate is desired, the block size of the compressed blocks can be known. Therefore, if there are thirty two 32-bit instructions in an ILAR, each compressed block's size have to be 512 bits to have a 2x compression. Similarly to have a 4x compression rate, each block has to be 256 bits wide.

Each compressed block consist of two parts, namely the header block and the instruction block. There is an exception to this where some compressed blocks consist only of the instruction blocks without a header block. These special blocks will be distinguished from the conventional header-instruction format blocks. The header block acts as a dictionary for instructions compressed in the block. It is further divided into the 3 sectors, namely the opcode reference, register reference and the immediate reference. Figure 4 shows the general layout of a compressed block. The references contained in the header block determines how many instructions can be compressed in the block and how good the compression algorithm is.

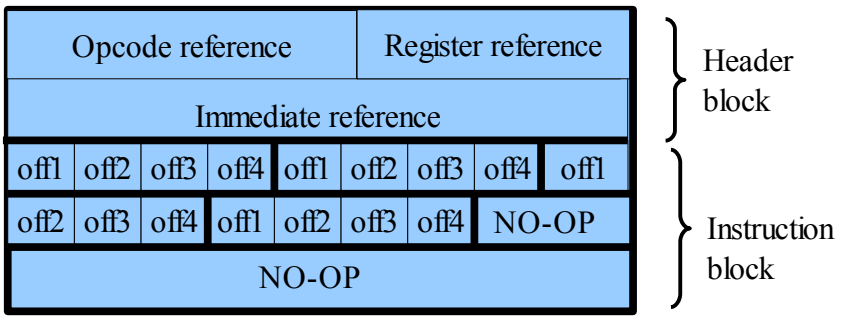

Figure 4: Typical header-instruction compressed block format

The instruction block typically consists of offsets referenced from the block's header. It could also contain local offsets between similar fields, for example SRC1 field is compressed as the offset from DREG. Compressed instructions are concatenated together to form the instruction block.

It will be ideal to load an entire ILAR in one compressed block. Fundamentally, this could be achieved by a smart compiler generating instructions that could be compressed well in a block. However, at this point in time the instructions are hand assembled and are therefore not optimized in any way. In order to maintain a fixed block length and a fixed compression rate, NO-OPs are padded in any unfilled instruction blocks. Subsequently, instructions that do not fit in the block are compressed using a new block header.

To make the compression concept clear, a MIPS32 equivalent example using block compression is shown in Table 6 and Table 7. Firstly, the MIPS32 assembly instructions are converted into bit patterns. Next, these instructions are compressed into a block by extracting similarities in the opcode, register and immediate fields. These similarities form the references in the header block. Subsequently, instructions in the instruction block references the field by a simple offset.

Table 6: MIPS32 assembly instructions

```
  LW $t1, j($sp)
  LW $t2, 0($t1)
  LW $t3, k($sp)
  LW $t4, 0($t3)
  LW $t5, k($sp)
  LW $t6, 0($t4)
ADD $t6, $t2, $t4
  SW $t6, 0($t5)
  LW $t2, 0($t1)
  LW $t4, 0($t3)
AND $t4, $t2, $t4
  SW $t4, 0($t3)
```

Table 7: MIPS32 equivalent compressed block

MIPS32 bit patterns	MIPS32 compressed block
100011 01001 11101 0000000000001010	100011 000000 101011 ◄— Opcode Ref.
100011 01010 01001 0000000000000000	01001 01010 01011 01100 11101 01101
100011 01011 11101 0000000000001011	01110 ◄———————— Register Ref.
100011 01100 01011 0000000000000000	0000000000000000 0000000000001010
100011 01101 11101 0000000000001011	0000000000001011 0110000000100000
100011 01110 01100 0000000000000000	0110000000100100 ◄— Immediate Ref.
000000 01110 01010 01100 00000 100000	00 000 100 001 00 001 000 000
101011 01110 01101 0000000000000000	00 010 100 010 00 011 010 000
100011 01010 01001 0000000000000000	00 101 100 010 00 110 011 000
100011 01100 01011 0000000000000000	01 110 001 011 10 110 101 000 ◄— Offset
000000 01100 01010 01100 00000 100100	00 001 000 000 00 011 010 000
101011 01100 01011 0000000000000000	10 011 001 100 10 011 010 000

The compression algorithms developed follow the general idea described above. However, a good compression rate could not be achieved by simply grouping instructions into blocks. Therefore, a divide-and-conquer approach is used where intermediate blocks go through similar iterations to obtain the final compressed instruction block. Each compression algorithm will need to go through three main iterations. These iterations have to be done in order for the algorithm to work. The following section describes each iteration of the compression algorithms.

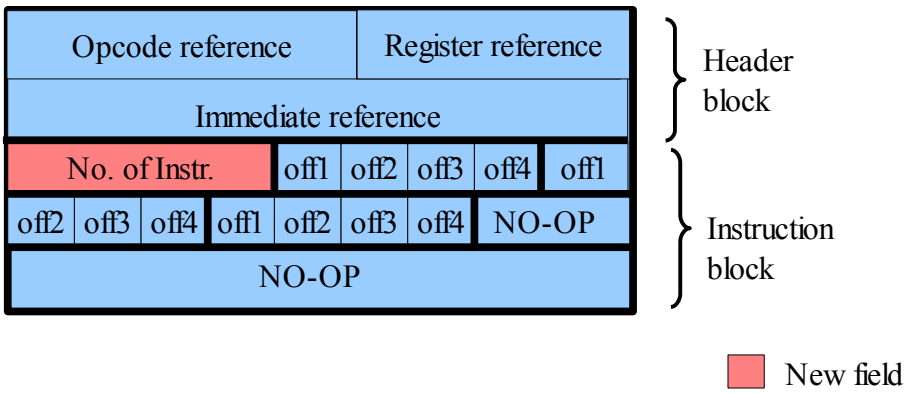

Figure 5: Intermediate compressed block format – Iteration one

3.1.1 Compression stages - iteration one

The first stage of the compression groups instructions into intermediate blocks following the compiled order. Intermediate blocks have the header-instruction block layout as shown in Figure 4. In addition to that, an additional field named *No. of Instr* is included in the intermediate block. This field is a fixed 5 bit field which sums the total number of instructions compressed. Figure 5 Shows how an intermediate block would look after iteration one. The first iteration is complete once the number of instructions being compressed within an intermediate block is known. The compression algorithm continues with iteration two.

3.1.2 Compression stages – iteration two

Just by looking at the block format layouts, the block header takes up a large portion of the block as a dictionary. Besides that, not all entries that have the same lookup are referenced in the same block. Therefore, to reduce this overhead, the intermediate blocks from the first iteration is processed further.

In the second iteration, the intermediate blocks are numbered in ascending order. The numbered intermediate blocks are searched exhaustively to find blocks instances that have the same header block. Then, the instruction blocks of these instances are concatenated with the intermediate block that has the lowest block number. In other words, repetitive header blocks are removed by recombining instructions out-of-order. Out-of-order compression could be performed on the instructions by adding in decoding

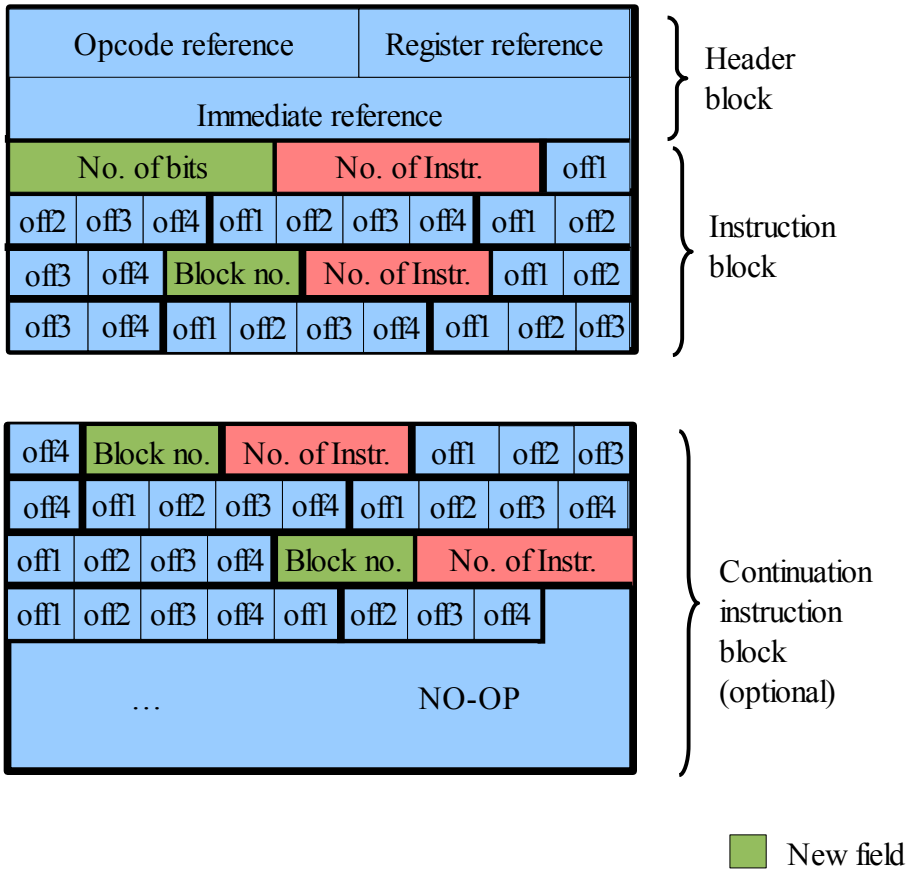

Figure 6: Intermediate compressed block format – Iteration two

information to the compressed block. Figure 6 shows the layout of the intermediate block after iteration two.

Since intermediate blocks that have the same header block are combined, the first section of the compressed block in this stage consist of the similar header block used in all instances. As can be seen from Figure 6, a new variable length *number of bits* and *block number* field is added to the compressed block. The *block number* field is the number associated with the compressed block in this iteration. The width of this field could be determined by the number of blocks being processed. For example if after iteration one there are 65 blocks, 7 bits have to be used for the *block number* field. On the other hand, the *number of bits* field is the summation of the total number of bits after this field onwards until the end of the block. This field is also inclusive of the bits in optional continuation block(s) that follow. Therefore, the optional continuation block(s) is/are

distinguished from a new header-instruction block by this field.

The layout of the intermediate block after iteration two is as follow. After the *number of bits* field, the instruction block that has the lowest block number is copied. This is followed by the concatenation of the instruction blocks that have similar header block instances. Each of the concatenation is separated by the *block number* field, which identifies where the combined instruction blocks come from. In the example shown in Figure 6, three instruction blocks which have similar header blocks are combined with the intermediate block which has the lowest *block number*.

Iteration two is continued until all blocks which have similar header blocks are combined. Subsequently, iteration three of the compression algorithm is executed.

3.1.3 Compression stages – iteration three

In this iteration, the intermediate blocks are further processed to achieve a better density of instructions compressed. Intermediate blocks that have the same opcode references and register references in the header block are combined.

Similar to iteration two, the intermediate blocks are numbered in ascending order. Then, these immediate blocks are searched exhaustively for matches in the header blocks. After that, the instruction blocks are concatenated with the immediate block that has the lowest block number. Similar decoding information are added to the blocks with an additional *local header* field, where the difference in the header blocks among similar instances (the immediate reference) is specified. Figure 7 shows the layout of the compressed block after iteration three.

Among the similar instances, the intermediate block that has the lowest block is used as a base where all similar instances are concatenated. Subsequently, a new *number of bits* field is added. This field sums the total number of bits for the combinations in this iteration after this field onwards until the end of the block. Following this field is a *local header* field which contains the immediate reference(s) of the block instance not contained in the header block. Then, the block number field is added which specifies which intermediate block the instruction blocks come from. Subsequently, the instruction block is concatenated which includes the *number of bits* and *number of instruction* field

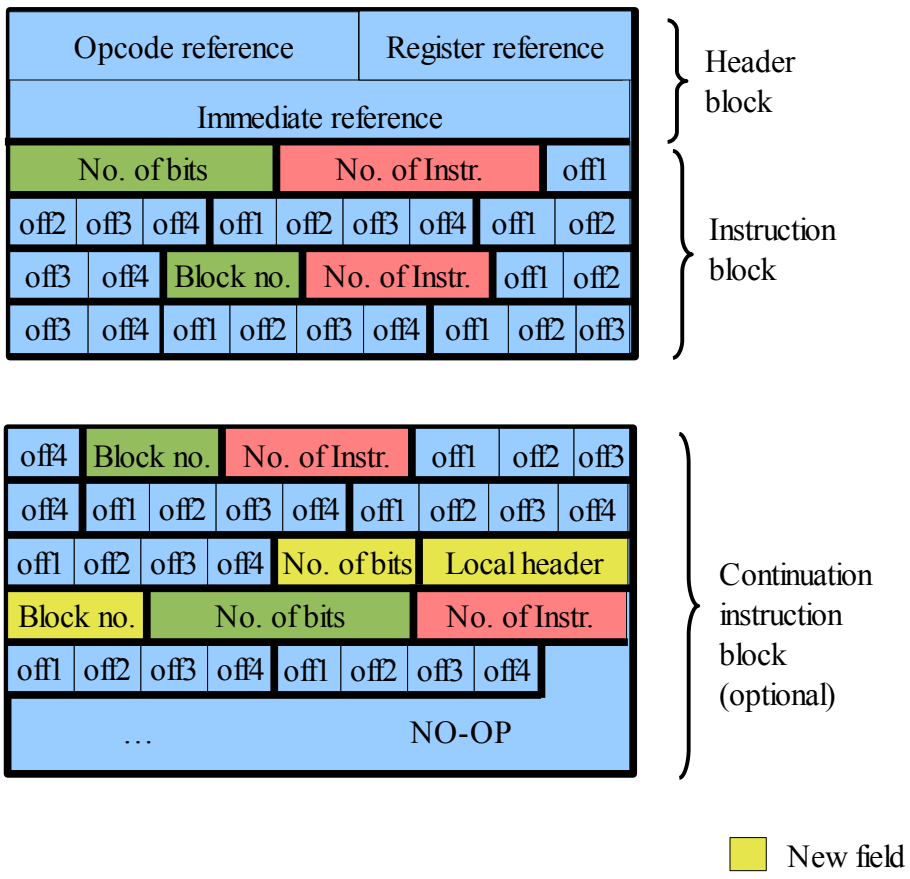

Figure 7: Intermediate compressed block format – Iteration three

from previous iterations.

This iteration is repeated until all intermediate blocks which have similar opcode and register references in the header block are combined.

3.2 Compression algorithm customization

The compression algorithms proposed in this thesis are block based. Instructions are compressed in blocks of fixed size and each of these compressed blocks has a header block that determines how good the compression algorithm will be. Since only a weak correlation exist among the set of inputs, it is almost impossible for us to manually determine which is the best header selection for a set of inputs. Besides that, one algorithm could yield a great compression rate for a particular set of inputs but not for another set of inputs. So, there needs to be method for us to customize and search for the compression algorithm that yields a good compression rate for a particular set of inputs.

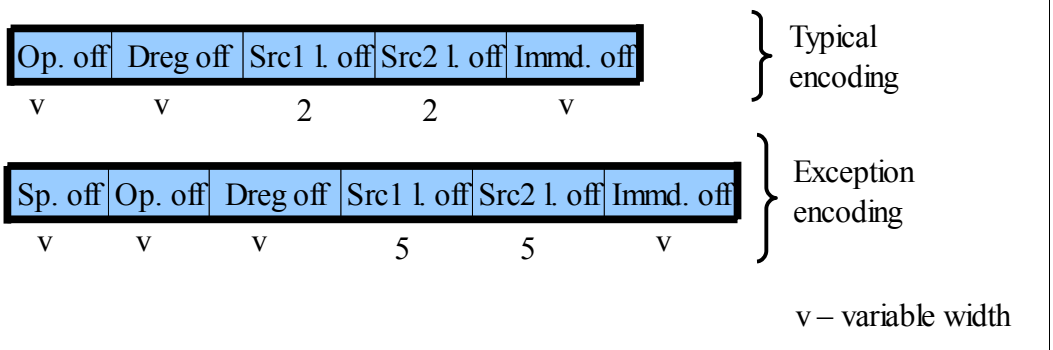

Figure 8: Typical and exception instruction encoding format

Next, the different customization of the blocked based compression algorithms are investigated. All the customization described in this section still uses the three iterations of the compression algorithm described in the previous section. The following section describes the customizations that have been developed.

3.2.1 Number of opcode references

Let's say the *LOADSW*, *ADDS* and *ANDS* opcodes get executed frequently in a group, it will be beneficial to be able to compressed all similar group instances in one instruction block. In order to get a good compression rate, the number of opcode references in the header block needs to be varied to suit the number of opcodes in this frequent group input sequence. Since header references are referenced in the instruction block using offsets, the number of opcodes that could be used as a reference will change with the power of two. A compression algorithm could have either 2, 4, 8 or 16 opcode references in the block's header. The corresponding *Op. off* field in the instruction block will be 1, 2, 3 or 4 bits wide.

Not making any assumptions about the inputs and to ensure that all the twenty three opcodes from Table 2 are available for selection, *header sets* are generated randomly at compile time. A header set contains opcodes that are used as opcode references. The number of *header sets* generated depends on the number of opcode references used in the compressed blocks. The total number of *header sets* is calculated by dividing the total number of opcodes (23 + 1 special opcode) with the number of opcode references used. The special opcode is needed for exception encoding and is

shown in Figure 8 in the next subsection. For example, if 8 opcode references are used, 3 unique *header sets* will be generated. When 16 opcode references are used, 2 *header sets* will be generated where one will contain 8 repetitive opcodes. The *header sets* are used as opcode references for all header blocks.

In iteration one, *header sets* are selected as opcode references in the header block when there is a match in the input's opcode. Subsequent inputs that have the same opcodes in the *header set* will be encoded in the same block. If there is no match in the *header set*, a new block will be created with the *header set* that contains the opcode match, provided the register and immediate references are valid. Otherwise, changes to the two latter references in the header block are also made in the newly created block. As described in the previous section, iteration two and three will try to combine blocks which have similarities in the header block.

3.2.2 Number of register references

Other than allowing variation to the number of opcode references, the number of destination register references in the header block are also allowed to be varied to suit the register usage in the inputs. To use efficient encoding, the number of register references available also varies with the power of 2. The specific numbers are 1, 2, 4, 8, or 16 register references. The corresponding *Dreg off* field in the instruction block will be 3, 1, 2, 3 or 4 bit(s) wide. It may seems counter intuitive when only one register reference is used, 3 bits offset is needed. This will be explained later in this section.

The algorithms are developed to be able to compress instructions in two encoding formats. Instructions are primarily compressed using the typical encoding format, but when an exception occurs, the exception encoding format is used. In a typical instruction encoding format, *Src1 l. off* and *Src2 l. off* fields are 2 bits local offsets from the destination register. However, if the local offsets are not sufficient, an exception occurs and the instructions are encoded differently. For example, *ANDS R5, R0, R1* will cause an exception to occur. In the exception encoding format, an extra special opcode referenced by the *Sp.off* field from the header block is used to distinguish it from the typical format. Since the local offsets for SRC1/2 are not sufficient in the exception case, the SRC1/2 5 bit register values are used in the exception instruction encoding format. The two

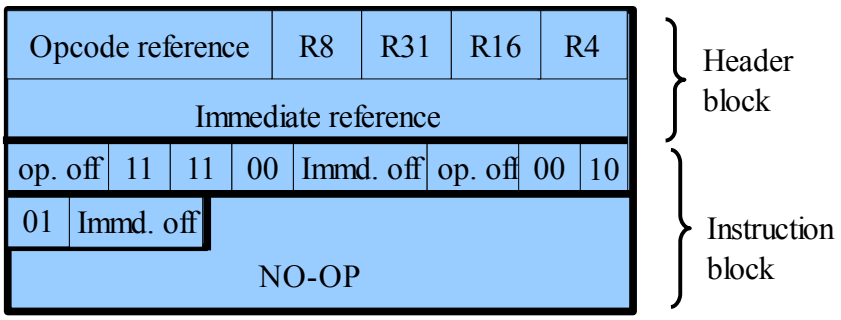

Figure 9: Example of register offsets with typical instruction encoding format

different encoding format are shown in Figure 8. Field are labeled – v to indicate variable width, since these fields are customizable for a compression algorithm.

When 2, 4, 8 and 16 registers are used as references in the header block, the *Dreg. off* field is used to referenced each of these available registers. The register references are registers that have been used recently. The SRC1 l. off and SRC2 l. off are local offsets Figure 9 shows an example of 4 registers being used as reference and a typical instruction encoding format after iteration one. The first instruction has offset 11, 11, 10. So, by looking at the header block, the destination register is R4, whereas the SRC1 and SRC2 register are R1 and R4 respectively.

When 1 register reference is used in the header block, the *Dreg. off* field is used as a 3 bit offset from this register. Since the offset used is 3 bits, the only four valid reference registers are R0, R8, R16 and R24. For example, if RO is the register reference in the header block, an offset of 101 will indicate the destination register is R5. *SRC1 l. off* and *SRC2 l. off* fields are still 2 bit offsets from the destination register.

3.2.3 Number of immediate references

Similar to the number of opcode and register references, the immediate references could also be customized in the block header. There could be 1, 2, 4, 8 or 16 immediate references. The *Immd off* field as shown in Figure 8 is scaled accordingly to the number of references.

When 2, 4, 8 and 16 immediate references are used, the *Immd off* field will be

used to reference these references in the header block. The immediate reference header block is formed by the most recently used immediate values. The scenario is similar to Figure 9, only that immediate references and offsets are involved. As shown in Figure 7, the local header inserted in iteration three is going to be number of immediate reference(s) used in the compression algorithm.

When only 1 immediate reference is used in the header block, *Immd off* field is 3 bits wide, which allows LLS (<<) of the base immediate reference up to seven times. Three randomly chosen values are used as the base immediate reference, which are 8, 24 and 160 respectively. These three base references allow the most frequently used immediate values to be obtained after LLS. Nevertheless, if these references could not be used even after shifting, the immediate value of the input is used as the reference of that particular compressed block.

3.2.4 Number of bits

The algorithm customizations described up until now has the potential of using variable width fields. Thus, there is a dynamic nature to these fields which do not allow field boundaries to be determined at compile time. Owing to that, once the field boundaries are known, extra processing could be performed to further improve the density of the compressed blocks.

Another customization is to allow the *number of bits* field as shown in Figure 7. to be optimized to use the least number of bits required. The default width for the field is fifteen bits. The largest number of bits for all the intermediate blocks after iteration two could be used to determine the optimum width of this field. Using this information, an extra iteration is performed on the intermediate blocks to obtain the optimum bit width for this field. The same procedure is also applied after iteration three. Therefore, this yields the optimum *number of bits* field for the compression block.

Using the minimum possible number of bits for each field allows more instructions to be compressed into a single bock. For example, if the value of the *number of bits* field is 1,579 after iteration three, this field could be stored in a few as 11 bits because 2^11 allows a range from 1,024 - 2,047.

3.2.5 Block number

The b*lock number* field from iteration two and three could also be further optimized to use the least number of bits. The default number of bits for the *block number* field is eight bits.

Similar to the customization of the *number of bits* field in the previous section, the extra iterations after iteration two and three is performed to obtain the least number of bits for the *block number* field. For example, if after iteration two, 61 blocks are required to compress the instructions, the *block number* field will be optimized to have only 6 bits because 2^6 allows a range from 32 - 63.

3.2.6 Block size

The default block size for the compression algorithm is 512 bits which yield a 2x compression when loading a 1,024-bit-wide ILAR. Nevertheless, It is possible to vary the block size field to investigate the feasibility of having a higher compression rate. To obtain a 4x compression, the block size of 256 bits is used. All of the algorithm customization discussed up until now, apply also to a 4x compression algorithm.

4. Software simulator and genetic algorithm

Based on the block compression algorithm discussed in the previous chapter, a software simulator is developed to simulate the compression process. This simulator also generates the final compressed blocks of instructions depending on the algorithm customization discussed. Therefore, this simulator is useful for us to get a good estimate of the effectiveness of the block compression algorithms developed.

Furthermore, which set of block compression algorithm customization is suitable for a given input instruction sequence needs to be determined. This is done by searching through the outcomes of the software simulator using a genetic algorithm (GA). The following sections describes the software simulator in detail and the search process using a GA.

4.1 Software simulator

The software simulator model is used to test the feasibility of the compression methods proposed. All the codes are written in C and are compiled using the gnu gcc compiler. The input to the software model is a .csv file containing test datasets of LARs assembly instructions defined in the ISA. The assembly instructions are decoded to binary code by an interpreter. The binary code of the instructions are then compressed by the simulator using a random compression method. The results of the software simulator is then searched using a GA for the best compression method among a population of compression methods. The GA provides us with a framework of determining the best compression algorithm among a defined population.

The output of the compression algorithm is the compressed instruction blocks generated by a random compression method. These compressed blocks are outputted to a .txt file. The block diagram in Figure 10 shows how the software simulator is laid out. The code for the software simulator is included in the list of files attached with this thesis.

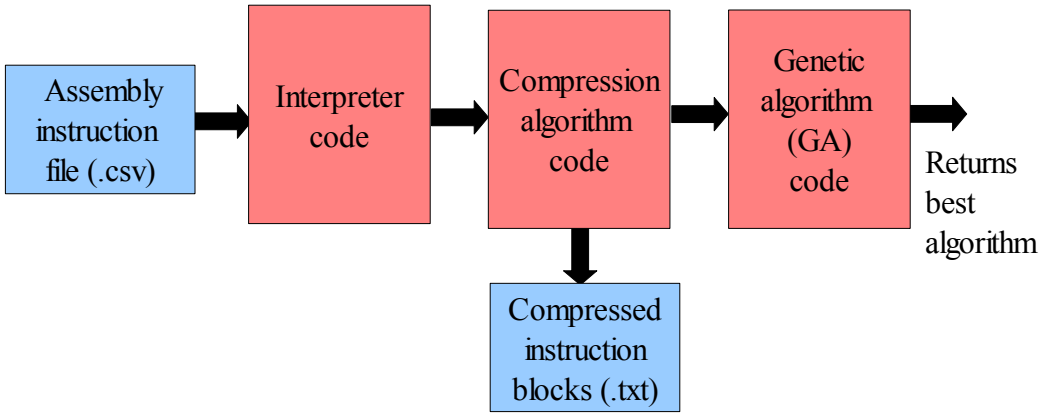

Figure 10: Software simulator model

4.1.1 Generating the assembly instruction input file

The assembly instruction inputs are assembly instructions that are to be fetched and decoded by the processor pipeline. In order for the algorithms and performance of the LARs model to be determined the input sequence of assembly instructions need to be defined.

By comparing the ISA of LARs with the well-known MIPS32 architecture, there are many similarities between them. First and foremost, MIPS pipelined uses two source operands and hold the outcome in a destination register. The LARs pipelined architecture is also based upon this. Secondly, many instructions are similar to MIPS32. For example, the MIPS ALU instructions are also defined in the LARs architecture and a BEQ instruction could be translated to a SELECT instruction in LARs.

Therefore, the test input file for LARs are defined from the assembly output of a MIPS based processor. Two well know algorithms, namely the bubble sort and binary search algorithms are compiled with gcc using the "-a" command in a MIPS32 cross compiler toolchain for Linux. The generated MIPS32 assembly instructions are then manually converted to the equivalent LARs instructions which is saved in a .csv file extension format.

4.1.2 Interpreting the assembly instructions

The LARs assembly instructions saved in .csv format needs to be interpreted

based on the definition in the ISA. As shown in Figure 10, C code is written to do the conversions.

The C source code and header file, *intp.c* and *intp.h* are compiled using the "*make intp"* command. It performs the following command in gcc.

gcc intp.c -o intp.out

To perform conversions, the .csv assembly instruction input file are inputted in the following manner. Bubble.csv is an example assembly instruction input file and bubble.txt is the generated input file of the compression algorithm.

./intp.out < bubble.csv > bubble.txt

4.1.3 Executing the block compression algorithm

The block compression algorithm could be compiled using *"make comp"* command which performs the following in gcc.

g*cc block_comp.c -o a -lm*

Then, the compression algorithm could be executed using the following command line and the compressed blocks are outputted to an example file input.txt.

./a > input.txt

4.2 Genetic algorithm (GA)

The previous section described many different customization that can generate different algorithms. By customizing the algorithm differently, the compression rate achieved could be different even for a fixed set of test inputs. In order words, for two distinctly different input sets A and B, one particular compression algorithm could be optimum for test input sets A but might yield a terrible compression rate for test inputs B. Therefore, different algorithms need to be run on the test inputs to find the best compression algorithm available.

There are 400 parametrically different algorithms that could be generated from the customization described and an exhaustive search could be performed to search for the best algorithm. Nevertheless, given that each algorithm simulated takes up processing

time of a CPU, an exhaustive search which involves testing each and every possible algorithm will be slow. Therefore, a GA is developed to take a sample of algorithms and search for the best among them.

GA is a type of evolutionary computing technique where one tries to evolve to the best solution of a problem. Firstly, the genetic algorithm creates random set of chromosomes, named as the *population*. Then, a solution to the problem is found for this initial population and a fitness score is assigned for each solutions. Subsequently, solutions to this problem are taken to form a new population. The populations created depends on the number of *generations*. From the initial populations, solutions that are less fit are either recombined or mutated based on the respective crossover and mutation rates. Less fit solutions that have their chromosome bits recombined are named *crossovers*. On the other hand, solutions whose chromosome bits are mutated are named *mutants*. Solutions to This is driven by hope that the new population will be better than the old one. These steps are repeated for N generations.

By taking a significant amount of samples in a population, a good estimate on the best algorithm could be achieved. Besides that, a GA also reduces the search time significantly by having many *generations* that replaces badly performing algorithms with *crossovers* and/or *mutants*.

4.2.1 Searching for the best block compression algorithm

In this thesis, the big problem is how to obtain the best algorithm for a set of test inputs? The best algorithm is one that has a good compression rate which translates to the least number of compression blocks. Therefore, the number of blocks compressed produced by an algorithm is used as a fitness measure to evaluate the effectiveness of the compression scheme in the search process.

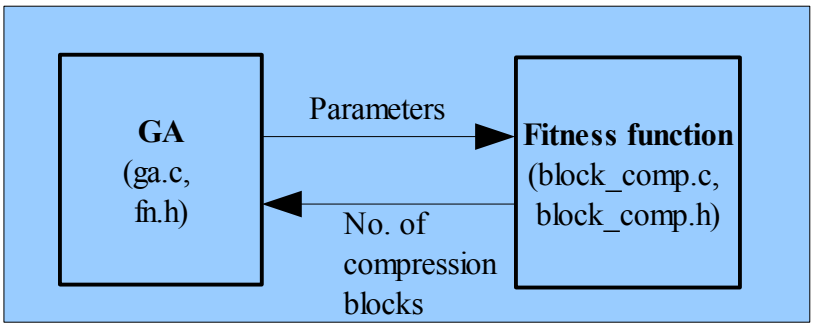

Figure 11: Layout of the genetic algorithm for searching the best compression method

Figure 11 summarizes how the GA is set up. The GA is created such that it uses the block compression algorithm as the fitness function. Firstly, parameters shown in Table 8, that represent a unique compression customization is randomly generated to create the initial population. Then, each of the compression algorithm is evaluated and the solutions are sorted in ascending order based on the number of compression blocks. Depending on the number of *generations*, new populations are created based on the previous population of compression algorithms. Compression algorithms that have large block sizes are discarded and could be replaced by *crossovers* or *mutants*. In Table 8, shorten field represents an optimized number of bits and block number fields.

Table 8: Parameters of the genetic algorithm

Encoding\Parameters	**No. of Opcode References**	**No. of Register References**	**No. of Immediate References**
000	Two	One	One
001	Four	Two	Two
010	Eight	Four	Four
011	Sixteen	Eight	Eight
100		Sixteen	Sixteen
101		One (Shorten field)	One (Block size = 256)
110		Four (Shorten field)	Two (Block size = 256)
111		Sixteen (Shorten field)	Four (Block size = 256)

The GA is compiled using the following using *"make ga"* command which is the following command line in gcc.

gcc ga.c block_comp.c -o ga -O2 -lm

The GA is then executed using the following format in the command line. 10 specifies the number of initial population to be created and searched. 5 is the number of *crossovers* to use. 3 is the number of *mutants* to use and 2 specifies the number of *generations* to make. Alternatively, *"make all"* could be used to run the GA with predefined values.

./ga 10 5 3 2

Search results of the GA are returned in the following format. The generation number is specified followed by the best and worst number of compressed blocks produced in the generation. Next, it specifies the parameter of the best performing compression algorithm, following by the number of compressed blocks used.

Generation 1: 6.0..999.0: 192

best is 192 → 6.0

A test case is carried out to search for the best compression algorithm among these algorithm. The genetic algorithm with *./ga 10 5 3 2* is executed thirty times and the best number of compressed blocks used are recorded. Table 9 shows the summarized results of the genetic algorithm.

As can be seen, the best compressed block numbers are produced when eight opcode references, one register reference, and either one or two immediate references are used. Also, the same number of compressed blocks are obtained when four opcode references, one register reference and one immediate reference are used. Besides that, shortening the *number of bits* field also helps give us better compressed blocks.

Table 9: Results of executing the genetic algorithm

Num. of Opcode Ref.	Num. of Register Ref.	Num. of Immediate Ref.	Average Num. of Compressed Blocks
16	1 (Shorten field)	1 or 2	3.33
16	1	2	3.5
16	Others	Others	8.75
8	1 (Shorten field)	1 or 2	3
8	1	1 or 2	4
8	Others	Others	7
4	1 (Shorten field)	1	3
4	1	1	4
4	1 (Shorten field)	4	7

Based on this result of the GA, a compression algorithm is picked to be implemented on hardware. The following chapter describes the design of the decompression logic needed for the selected algorithm.

5. Hardware descriptions of models

The results of the GA selects the best block compression available. For this algorithm to be used in loading the ILAR, a decompression process needs to take place Since the decompression process takes place during runtime, it needs to be performed close to constant time.

Before investigating the decompression hardware, how the ILAR hardware is being laid out needs to be understood. Also, since the newly proposed memory model is compared to the conventional MIPS IF model, a hardware description of both of these models are built so that comparisons on the performance could be made. The conventional 5 stage MIPS pipelined is shown in Figure 12; the changes to the MIPS architecture introduced by ILAR is shown in Figure 13 . Note that the pipeline still has 5 stages and the obvious substitution is made in the first stage, but the ILAR version also loses the two adders that normally are involved in producing instruction addresses. It is also significant that memory access happens only in the MEM stage, not in IF and MEM.

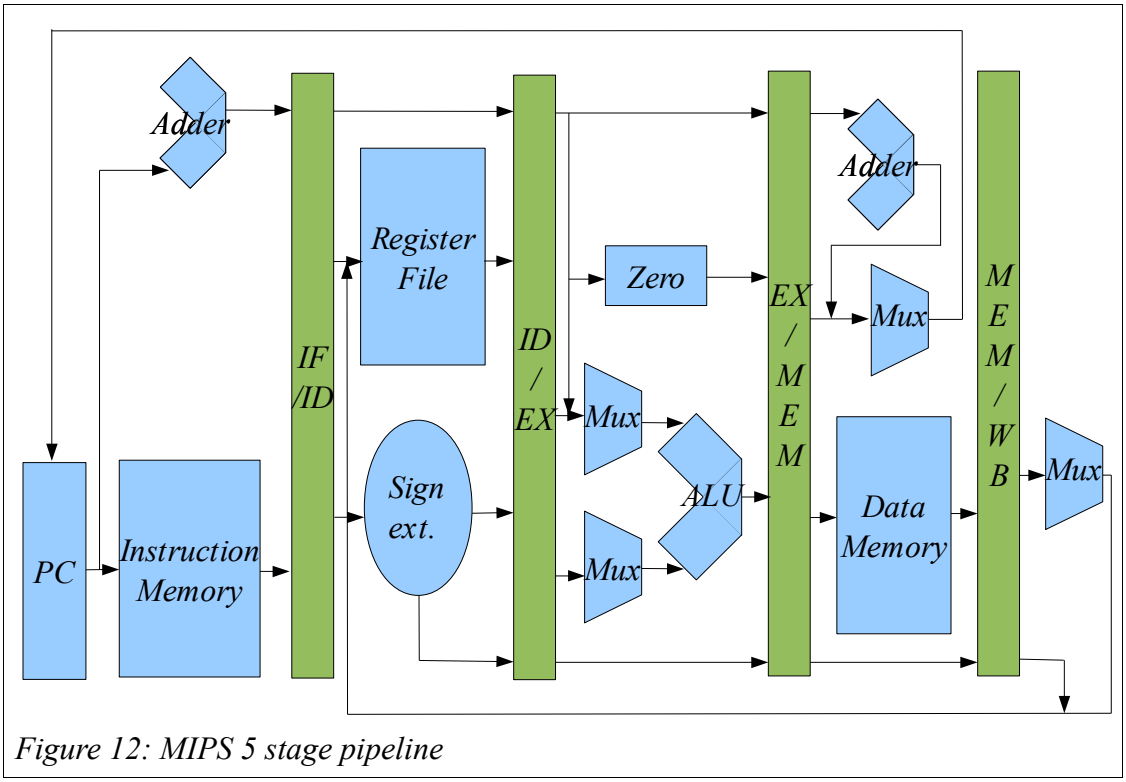

Figure 12: MIPS 5 stage pipeline

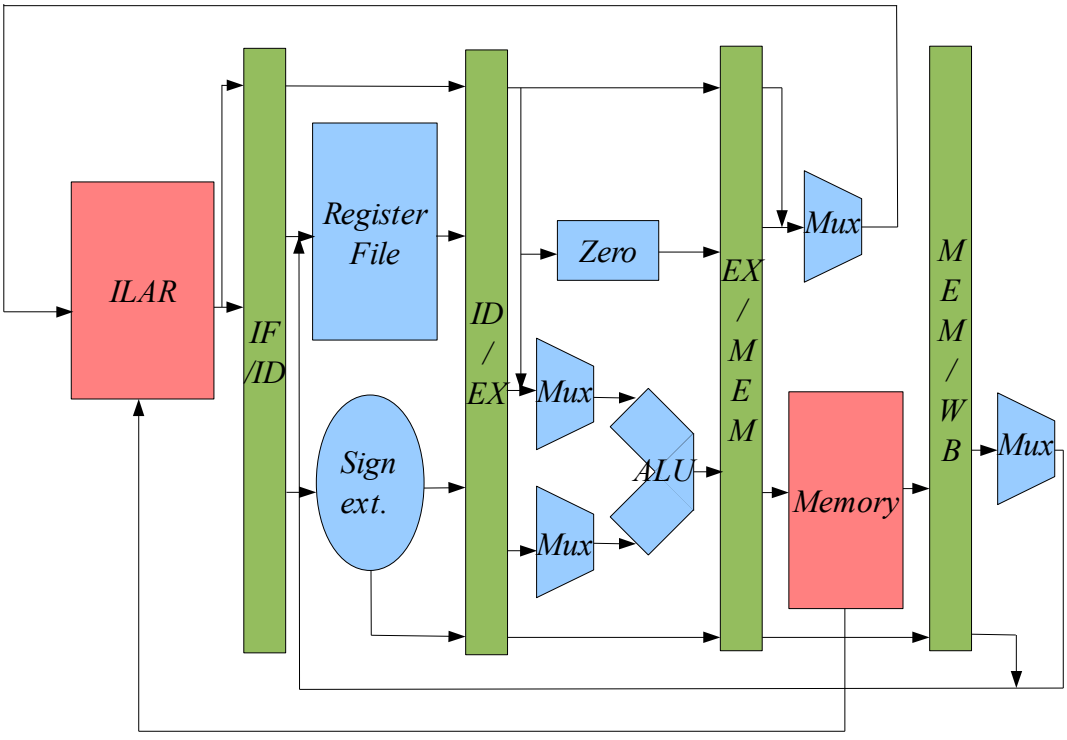

Figure 13: ILAR changes to the MIPS IF-stage

All the hardware description models are coded in Verilog and developed in Xilinx ISE 10.1.03. The target technology is Xilinx Virtex II xc2v8000 chip. Models are also post implementation simulated using Modelsim SE 6.4a. The HDL files are included in the list of files together with this thesis.

5.1 ILAR hardware

A hardware description of the ILAR hardware is developed as shown in Figure 14. *ILAR_MODULE_P* consist of 6 ILAR modules where each of them are 32 x 32 bits or 128 B wide. The operation of ILAR is similar to a normal ram memory where *read/write* signals can be inserted to read from/write to addresses specified by *addr.*

Instead of having a program counter to specify which address to read from, a *instrsel* (instruction select) and *rowsel* (instruction select) pointer are defined where *rowsel* selects the ILAR number and *instrsel* selects which instruction position is being referred to in the ILAR.

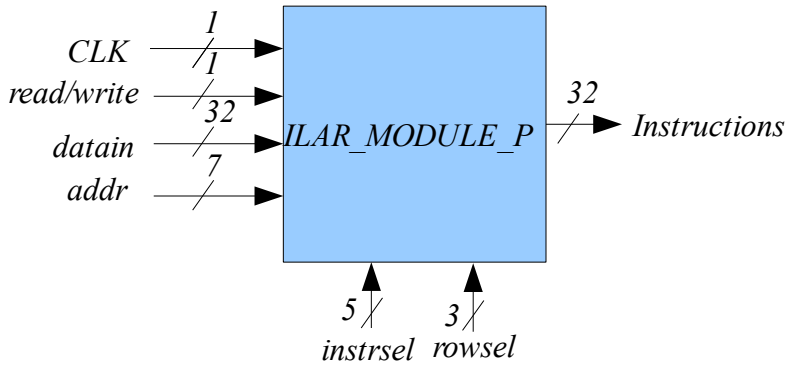

Figure 14: Block Diagram of the ILAR_MODULE_P module

Compressed instruction blocks stored in main memory are decompressed to be loaded into the specified ILAR whenever a `Fetch` instruction command is received. The size of this memory or specifically the *RAM_COMP_MODULE* as shown in Figure 18 has been developed to be 256 x 1024 bits or 64 GB.

5.2 MIPS IF memory-cache hardware

Besides developing the ILAR hardware described in the previous section, a MIPS-like IF stage is developed in Verilog to simulate a typical instruction fetch cycle. This model consists of a memory and a direct mapped cache model which are used to store and fetch instructions in an instruction fetch stage. A block diagram of this module is shown in Figure 15.

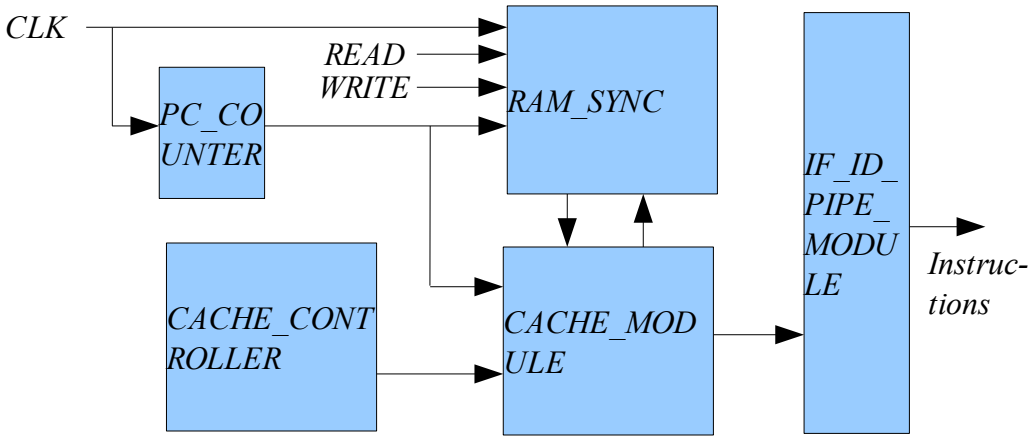

Figure 15: Block Diagram of the MIPS IF Memory-Cache module

The *PC_COUNTER* module counts off the clock and provides a count for both the *RAM_SYNC* and *CACHE_MODULE*. This count is used by the *CACHE_MODULE* to load the cache so that future references to the same instructions do not need to be loaded directly from *RAM_SYNC*. The *CACHE_CONTROLLER* module provides the necessary control signal to the cache and the *IF_ID_PIPE_MODULE* is a pipelined register to hold the instructions fetched.

In order to provide a direct comparison with an ILAR module described in the previous section, the RAM size is also developed to have 32 x 8192 bits or 64 GB. The Cache module will have 1024 bits or 128 B in size. This is the size of one ILAR.

5.3 ILAR decompression hardware

A hardware description of the decompression algorithm is performed to establish a proof of concept. The purpose of this hardware is to decompress compressed instruction blocks in main memory to load the ILAR hardware as shown in Figure 14.

This model uses the compressed blocks generated by the software simulator and the results of the GA from previous section. A block digram which shows how the results from the software simulator is integrated with the decompression hardware is shown in Figure 16.

Compressed instructions blocks generated by the simulator are inputted to a Xilinx memory model through a binary text file using the *$readmemb* command. The blocks are decompressed and loaded into a ILAR hardware descibed. The decompression hardware model describes the logic building blocks needed in the decompression process. A testbench waveform is generated and simulated using Modelsim SE 6.4a to validate the compression algorithm.

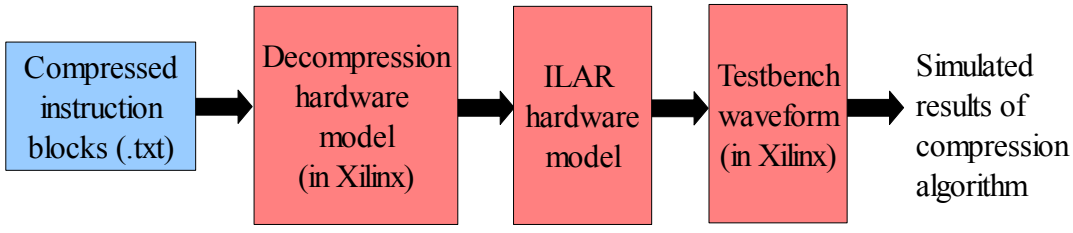

Figure 16: Hardware model

A structured behavioral coding style is used to decompressed the instruction blocks. By using this coding method, the decoding process could be pipelined so that similar modules could be instantiated to decode different parts of the instruction blocks simultaneously.

An algorithm is picked to be implemented on hardware based on the search result returned by the GA shown in Table 9. This algorithm has eight opcode references, one register reference and one immediate reference. Besides that, this algorithm has an optimized *number of bits* field where the software simulator chose the *number of bits* field from iteration two and three to be 8 bits and 11 bits respectively. As for the *block number* field, it is chosen to be 6 and 5 bits for iteration two and three respectively.

As the implementation of the decompression hardware can be deeply pipelined, there will be different hierarchies of modules. There are three different hierarchy to the pipelined decompression hardware. The modules that make up the hierarchies are described, starting from the lowest hierarchy.

5.3.1 Lowest hierarchy

This is the lowest functional unit in the pipeline named *DEC_BLOCK* which decompresses instructions contained in a single block. A single compressed block (512 bits) stored in the instruction memory is broken down into two 256 bits blocks and one of

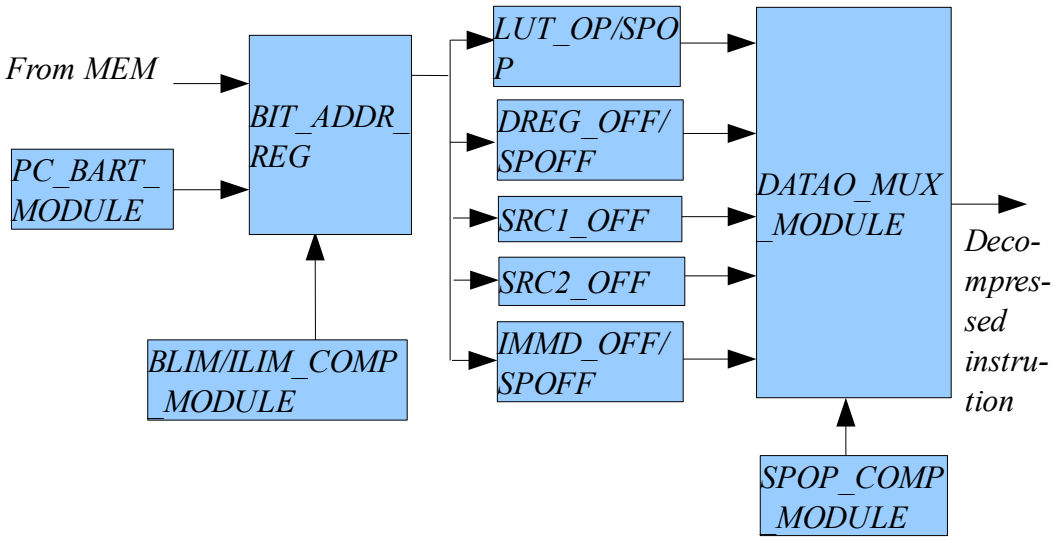

Figure 17: Block Diagram of the DEC_BLOCK0/1/2/3/4 module

these blocks is loaded into a local register labeled as the *BIT_ADDR_REG* as shown in Figure 17. This register read chunks of the compressed bit sequences at the boundaries to be decoded. The register operates off a program counter labeled as *PC_BART_MODULE* which increments a count associated with the bit boundaries in the 256 bit instruction block.

To differentiate the decompression of special instruction sequences from regular sequences, a comparator labeled as *SPOP_COMP_MODULE* is used to check for the special opcode. Using this information, decompression of the instruction fields are performed using look-up tables (LUT) and adders. Instruction opcodes from the block header are loaded into a LUT labeled as *LUT_OP* and *LUT_SPOP*. The instruction bit sequences are used to lookup the corresponding instruction opcodes. The remaining instruction fields, namely the register and immediate fields are offsets from reference field in the header. Therefore, adders are used to decompress these fields. These adders adds the offsets to the references in the block header and are labeled as follows: *DREG_OFF*, *DREGSP_OFF*, *SRC1_OFF*, *SRC2_OFF*, *IMMD_OFF* and *IMMDSP_OFF*.

As described in the compression algorithm section in Chapter 4, compressed instruction blocks may contain local immediate references. In hardware, this translates to a 2 to 1 multiplexer labeled as *IMMD_MUX_MODULE* to select between immediate header and local references.

A 2 to 1 multiplexer labeled as *DATAO_MUX_MODULE* is used to select the correct decompressed instruction. As the instruction compression blocks could span several blocks, two comparator models labeled as *BLIM_COMP_MODULE* and *ILIM_COMP_MODULE* are used to load the next 256 bit blocks into the local register for decompression.

5.3.2 Intermediate hierarchy

The second level module named *LARS_DECOMP* uses a structural coding style to instantiate instances of *DEC_BLOCK* to perform the decompression within a compressed instruction block.

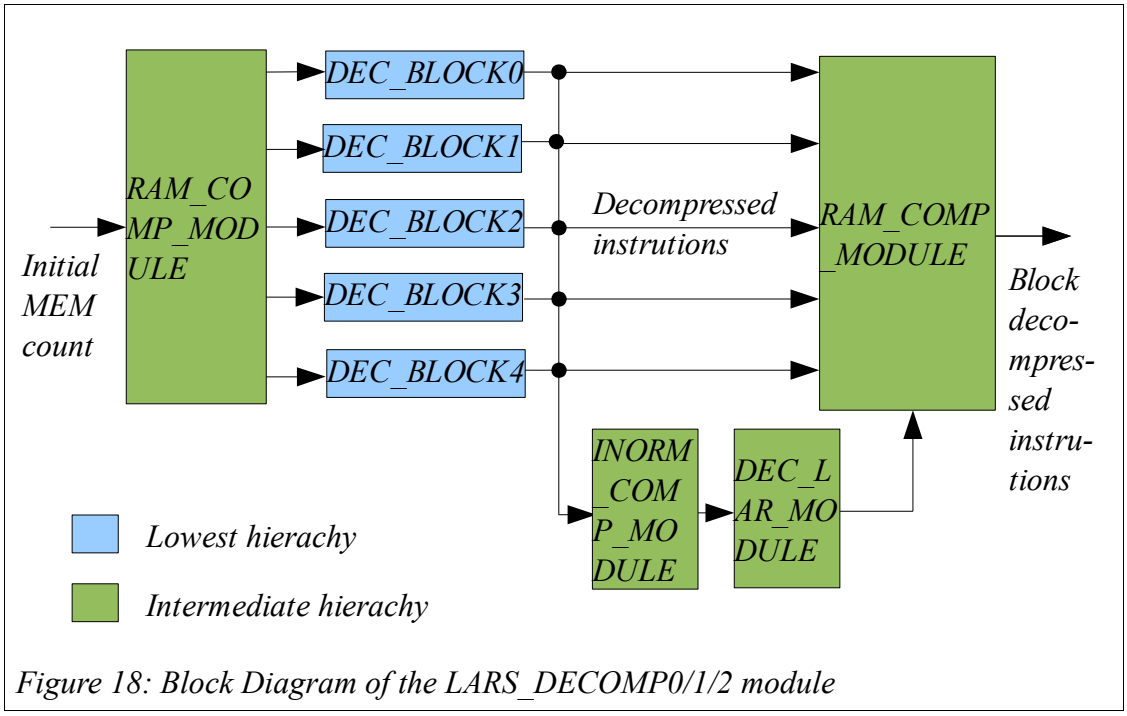

Figure 18: Block Diagram of the LARS_DECOMP0/1/2 module

The first block is a RAM where the compressed instructions are stored and is labeled *RAM_COMP_MODULE* as shown in Figure 18. In this block, compressed instructions are read out in blocks of 256 bis using a program counter labeled as *PC_RAM_COMP_MODULE*.

The *DEC_BLOCK* module is then instantiated five times. Each *DEC_BLOCK* modules enables the next *DEC_BLOCK* module when the bit boundaries are known. A 'done' signal is flagged when the *DEC_BLOCK* is finished decoding its part. If there are more sub-blocks to be decompressed, operation from the *DEC_BLOCK5* circulates back to *DEC_BLOCK0* until all the sub-blocks in the compressed instruction block are decompressed.

A buffer labeled *BUF_ADDR_DATA_MODULE* will buffer all the decompressed instructions from the five *DEC_BLOCK* modules. Decompressed instructions are placed into ILAR based on the information decoded from the instruction blocks. Information is decoded by the *DEC_LAR_MODULE* and the *INORM_COMP_MODULE*.

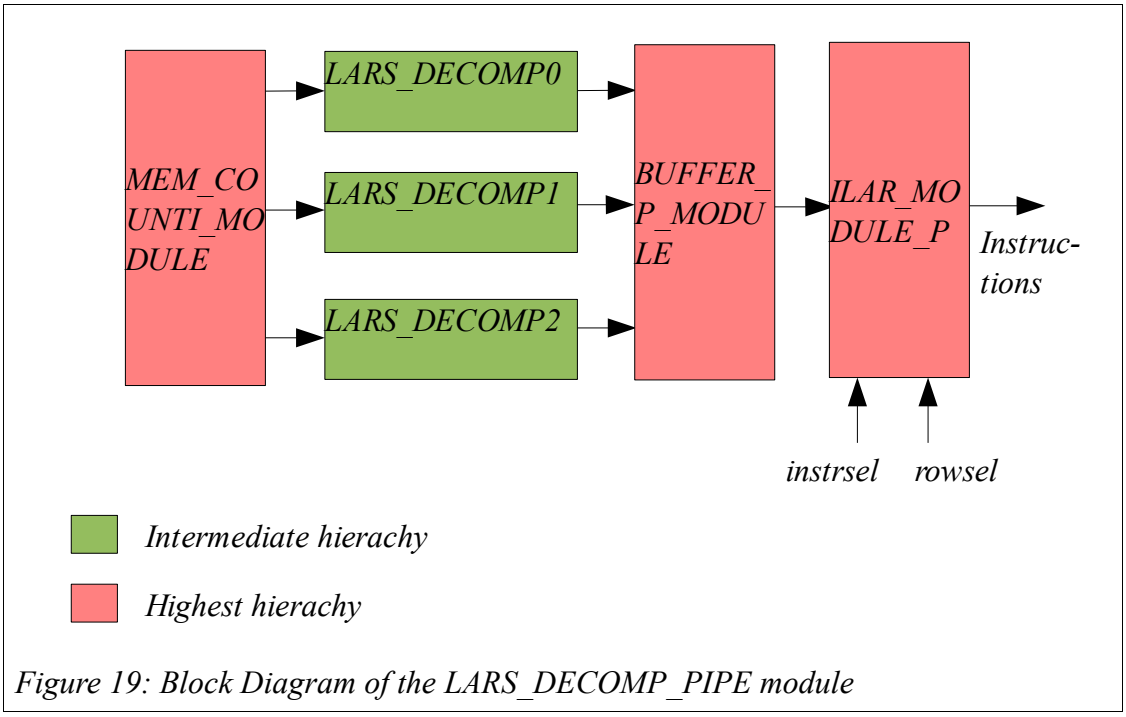

Figure 19: Block Diagram of the LARS_DECOMP_PIPE module

5.3.3 Highest hierarchy

The highest hierarchy of the hardware description ties all the blocks together to decompressed blocks of compressed instructions. The highest hierarchy is labeled *LARS_DECOMP_PIPE* as shown in Figure 19. Besides that, it also includes the connection with the ILAR hardware model described which is named *ILAR_MODULE_P* in the description.

Three of the intermediate module engines labeled *LARS_DECOMP0/1/2* are instantiated structurally in the hardware description. *MEM_COUNTI_MODULE* loads the initial memory count to load the compressed block of instructions from main memory. The decompressed instructions are buffered according in a buffer labeled as BUFFER_P_MODULE.

Decompressed instructions in the buffer are loaded into the ILAR hardware in order. After loading the ILAR, the instruction fetch process will be fast without requiring additional memory accesses. During an instruction fetch, normal execution of a program could be performed by using the *instrsel* and *rowsel* pointer.

6. Results

In this section, some of the results of the decompression hardware will be discussed. Due to the complexity and drastic changes to the architecture model, comparison to existing architecture will not be straightforward. Furthermore, the compression algorithms developed here do not guarantee to be the best solution for every instruction sequence. In addition to that, the "straw man" model of the decompression hardware could only be used as a stepping stone reference for future work in the ILAR area.

Despite that, considerable time and effort have been spend to prove that ILAR changes to the architecture is beneficial to reduce the memory latency in a typical instruction fetch cycle. In order to perform comparison to existing architectures, a hardware prototype of the MIPS IF memory-cache model has been developed besides the ILAR and decompression hardware model as described in the previous chapter. In this chapter, the results obtained from the hardware prototypes developed are discussed.

6.1 Memory latency

In a typical MIPS architecture, the memory latency is the summation of the time associated with reading and also loading the caches. During this time, the processor is idle and this could potentially waste precious processor time.

In the MIPS IF model developed, it takes 6 clock cycles to fetch instructions from main memory into a cache line. In other words, the cost of a cache miss is 6 clock cycles. On the other hand, a cache hit is 3 clock cycles. Figure 20 Shows the waveform simulation captured from Modelsim. In the example shown in Figure 20, *pcsel=1* increments the program counter, whereas when *pcsel=2* the address from *count* is loaded into the program counter. This example shows the scenario of reading instructions stored in the cache line. On the second *read* pulse, it is a cache hit and the instruction is retrieved from cache line 0. The third read pulse tries to read instruction in *address 32*, this caused a cache miss and the instruction retrieved from main memory is overwrite into cache line 0. Subsequent reads form *address 0* of main memory will cause cache misses again.

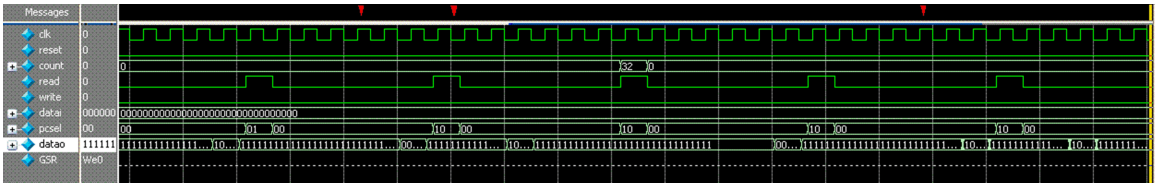

Figure 20: Waveform of reading from cache in the MIPS IF Model

On the other hand, the ILAR model does not introduce additional memory latency to the overall processor time since instructions are read directly from the ILAR hardware. Compressed instructions blocks are read from main memory and loaded into ILAR hardware after decompression. Program execution will then read instructions in sequence from ILAR. Since branches is performed locally in the ILAR, there will not be any additional latency caused by misses in program execution. Figure 21 shows a waveform of the decompression hardware in the ILAR model.

Instructions are decoded in parallel in the pipelined decompression hardware model. Signal *dbgbart0/1/2/3/4* are the decoded bit boundaries from the compressed instruction block. The *deco* signal is the decoded sequence number of the instructions. These decoded informations provide the necessary details for loading the appropriate ILAR. The decompressed instructions are contained in the *bufout* signal.

After successful decompression of the compressed blocks, instructions would be loaded into ILAR. This completes the instruction fetch stage. All future program execution could read from ILAR by using the *instrsel* and *rowsel* signals. Figure 22

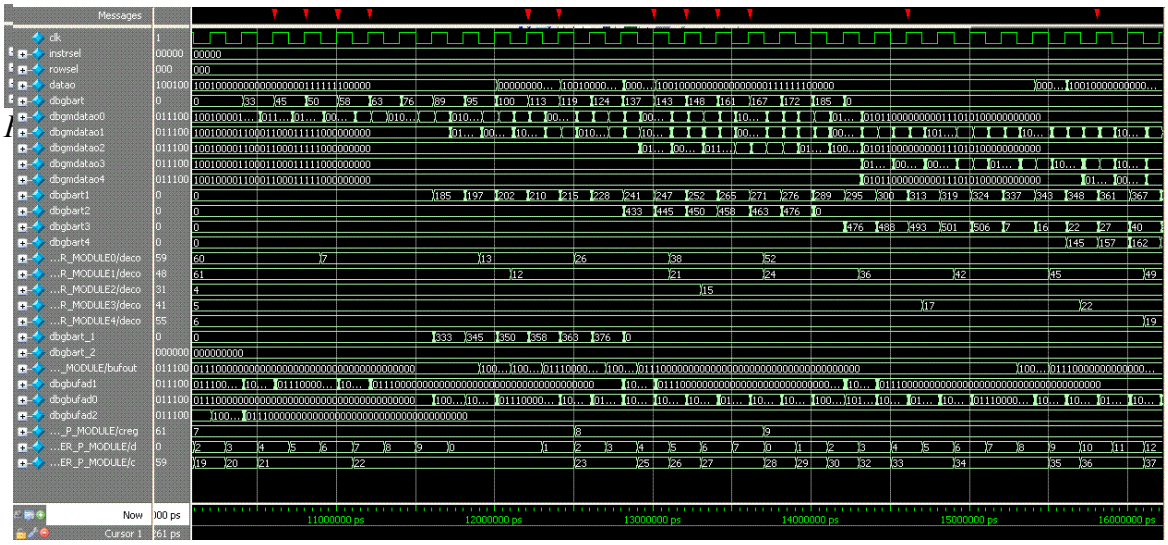

Figure 21: Waveform of the decompression process in the ILAR model

shows the waveform of reading from ILAR. The *rowsel* signal is used to select the specific ILAR to read from while *instrsel* selects which instruction to read from the ILAR.

The execution time taken for both models to process a similar input sequence are taken and compared in Table 10. As there will be no misses in the ILAR hardware, no extra latency is introduced by needing to fetch instructions from main memory again. The execution time for the ILAR model is inclusive of the time required to decompress the compressed blocks. In actual fact, the execution time is shorter because program execution could continue even during a `Fetch` instruction.

The MIPS IF Model requires longer execution given that there will be cache misses occurring. The latency associated with misses will be larger given a longer instruction sequence or a more complex code with many branch instructions. All of these has no effects on the ILAR model.

Table 10: Comparison of the execution time between the hardware for both models

	MIPS IF Model	**ILAR Model (no cache misses)**
	Execution time (us)	
60% cache miss	66	27
50% cache miss	62	
40% cache miss	58	
30% cache miss	54	

6.2 Compression performance

The compression algorithm determines how good instructions are compressed in a block. For the algorithm that is prototyped, a block size of 512 bits is used. Therefore, it yields a constant 2x compression for every block.

For the input instruction sequence that is used, instructions are compressed into six compressed blocks of 512 bits. These blocks are decoded by the decompression hardware and are used to load 6 ILAR with instructions. Table 11 summarizes the parameters obtained from the compressed blocks. All the compressed blocks are only about 75 % filled with compressed instructions. Also, the algorithm developed in

hardware results in more than half the ILAR being empty.

Since the input sequence is hand assembled, smart compiler techniques could not be utilized to obtain an easily compressed assembly instructions sequence. Also, not all the compressed instruction blocks are utilized. Having a larger input sequence will also help increase the compression performance.

Table 11: Compression performance of algorithm implemented

Parameters	Percentage (%)
Average % of no-ops in ILAR	64.06
Average % of blocks filled	76.24

6.3 Hardware utilization

Both of the hardware prototypes are synthesized using Xilinx's XST simulator. Then, the designs are translated and implemented using logic resources available on the FPGA chip. All signals are successfully placed and routed using the IDE and the summary of the device utilization is shown in Table 12. The gate count shown is an estimate of the number of ASIC gates for the two prototypes using the ratio 1:5 for slices/flip-flops/LUTs.

Table 12: Summary of hardware utilization of the two models

	MIPS IF Model	ILAR Model
Gate count	30780	679145
Slice registers	1488	7561
Flip - flops	132	7030
Latches	1356	531
Occupied slices	1173	42061
4 input LUT used as logic	2007	78646
Bonded	83	680
RAMB16s	16	22
BUFGMUXs	4	8

Due to the fact of having more decompression logic in the ILAR model, there is a

5x increase in the slices of registers used. Furthermore, more RAMB16 (synchronous block RAMs) of 16 bits width are used in the ILAR model. Even though the size of the cache in the MIPS-IF model and an ILAR are developed to be of equal size on purpose, the ILAR model has six ILAR developed in the hardware model to showcase the benefits. Therefore, it results in more block RAMs used after synthesis. This also explains why there are more BUFGMUX (multiplexed global clock buffer) for the ILAR model, because it is used to select the desired clock within the FPGA chip. Besides that, the decompression hardware is pipelined to decompress instructions in parallel. All of these contributes to larger occupied slices for the ILAR model in the synthesis result.

In addition to that, more flips-flops are used for the state machines associated with pipelining in the ILAR model. However, ILAR model provides savings in the number of latches used since most of the modules are fully pipelined.

The ILAR model is developed to have logic to look up fields in the compressed instruction blocks. For example, adders are instantiated for calculating offsets from references in the header blocks. These instantiations get translated to LUTs in the synthesis process and therefore contributes to almost 40x more 4 input LUTs in the ILAR model.

7. Conclusions and future work

As a conclusion, the work that has been done for this thesis and the results obtained from the experiments conducted are summarized.

In this thesis, the main research topic is LAR, specifically ILAR and how it separates memory access from instruction access. ILAR is a wide register that utilizes SWAR concepts, allowing pre-fetching of instructions into registers and also cache like spatial locality benefits. It also removes conventional instruction fetches from main memory and prevents cache misses in conventional memory-cache architectures. Despite the obvious benefits, an efficient mechanism is still needed to fetch instructions into ILAR. Therefore, the concepts of block compression is proposed as a method to populate ILAR. The first part of this thesis focuses on proposing block compression algorithms by developing a software simulator to compress a given input sequence of assembly instructions. The algorithms are discussed in detail in Chapter 4.

However, the effectiveness of the compression algorithm depends heavily on the input instruction sequence. Therefore, a search method is needed to determine the best algorithm for a given input sequence. A GA is used to to search for the best algorithm. This GA is general solution to determining the best algorithm as the fitness function could be modified easily for new algorithms developed. In the experiment conducted using the GA, an algorithm is chosen to be investigated further.

Part two of the thesis involves building a virtual hardware prototype of the decompression hardware needed by this algorithm. Chapter 5 describes the design of the decompression hardware prototype in detailed. This virtual hardware prototype is compared with a MIPS IF direct mapped cache model to showcase the benefits of ILAR.

The results yielded by the experiments conducted are very encouraging. The reduction in memory latency is definitely a big win for ILAR as cache misses are eliminated and it brings an improvement in the use of memory bandwidth. Nevertheless, the compression algorithm proposed and chosen to be implemented did not fully utilize the ILAR hardware structure. The compressed instruction blocks are not fully utilized and this may be a reason why the ILAR are filled with a large amount of no-ops. Other block

compression algorithms may also be investigated to yield better compression performances.

As expected, more hardware is required for the ILAR model for decompression of the instruction blocks. However, even though more hardware is needed, the latency is not significantly affected. A large improvement in memory latency is still observed in the models developed. Future work could further improve the "straw man" model of decompression hardware and propose improvements to the compression algorithms once the framework of an optimizing compiler has been defined.

Bibliography

[1] Intel, http://www.intel.com/technology/architecture-silicon/silicon.htm

[2] Computer Architecture A Quantitative Approach, John L. Hennessy and David A. Patterson

[3] Hank Dietz, Technical summary: SWAR technology. Technical report, School of Electrical and Computer Engineering, Purdue University , February 1997

[4] Randall J. Fisher. General-purpose SIMD Within A Register: Parallel processing on consumer engineering. Purdue University, PhD Thesis proposal, November 1997

[5] Randall J. Fisher and Henry G. Dietz. The Scc Compiler: SWARing at MMX and 3DNOW! In Larry Carter and Jeanne Ferrante, editors, Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing, La Jolla, California, August 1999. Springer – Verlag

[6] H. Dietz, C. H. Chi, "CRegs: a new kind of memory for referencing arrays and pointers", Supercomputing'88, pp.360-367, Jan 1988

[7] Eyerman, S; Eeckhout, L; A Memory-Level Parallelism Aware Fetch Policy for SMT processors, HPCA 2007, IEEE 13th International Symposium on 10-14 Feb. 2007 Pages(s):240-249

[8] Caixia Sun, Hongwei Tang and Minxuan Zhang, "An Instruction Fetch Policy Handling L2 Cache Misses in SMT Processors", Proceedings of the Eight International Conference on High-Performance Computing in Asia-Pacific Region 2005 (HPCASIA'05)

[9] Caixia Sun, Hongwei Tang and Minxuan Zhang, "Controlling Performance of a Time-Critical Thread in SMT Processors by Instruction Fetch Policy", Proceedings of the Seventh International Conference on Parallel and Distributed Computing Applications and Technologies 2006 (PDCAT'06)

[10] Jing Wang, Shengbing Zhang, Meng Zhang, Xiaoping Huang, Pan Yongfeng, "A Modified Instruction Fetch Control Mechanism for SMT Architecture", IEEE Region 10 Annual International Conference, Proceedings/TENCON 2007

[11] Shoji Yoshida, Shigeya Tanaka, Kotaro Matsuo, Takashi Hotta, Hideo Sawamoto, Teruhisa Shimizu, “Instruction Fetch and Dispatch Schame with Flag-in-Cache/in-IBR, Systems and Computer in Japan, Vol. 29, No.4,1998

[12] Pierre Michaud, Andre Seznec, Stepehn Jourdan, “An Exploration of Instruction Fetch Requirement in Out-of-Order Superscalar Processors”, International Journal of Parallel Programming, Vol. 29, No.1 2001

[13] Stephen Hines, Gary Tyson, David Whalley, “Addressing Instruction Fetch Bottlenecks by Using an Instruction Register File”, LCTES'07 ACM, June 13-15, 2007

[14] Bellas, N., Haji, I., Polychronopoulos, C., and Stamoulis, G., “Energy and performance improvements in a microprocessor design using a loop cache, In Proceedings of the 1999 International Conference on Computer Design (October 1999), pp. 378-383.

[15] Bellas, N., Haji, I., Polychronopoulos, C, Using dynamic cache management techniques to reduce energy in general purpose processors, IEEE Transactions on Very Large Scale Integrated Systems 8, 6 (2000), 693-708.

[16] Michael Fredman, Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, Andreas Moshovos, “Temporal Instruction Fetch Streaming”, IEEE 2008

[17] Krishna Melarkode, Line Associative Registers. Master's Thesis, University of Kentucky, October 2004, http://lib.uky.edu/ETD/ukyelen2004t00195/Krishna.pdf

[18] Lekatsas, Haris and Wolf, Wayne, Code Compression for Embedded Systems, Proceedings of the 1998 35th Design Automation Conference, June 15, 1998 - June 19, 1998

[19] Jacob Ziv and Abraham Lempel; A Universal Algorithm for Sequential Data Compression, IEEE Transactions on Information Theory, 23(3), pp. 337–343, May 1977

[20] Huffman's original article: D.A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", Proceedings of the I.R.E., September 1952, pp 1098-1102

[21] ARM7TDMI Technical Reference Manual, Rev. r4p1, 2004

[22] Intel iAPX432 General Data Processor Architecture Reference Manual, 171860-001

[23] A. Wolfe and A. Chanin, Executing compressed programs on an embedded RISC architecure, In Proc. Int'l Symp. Of Microarchitectures, 1992

[24] Richard Phelan, Improving ARM Code Density and Performance, New Thumb Extensions to the ARM Architecture, June 2003

[25]Eric Rotenburg, Steven Bennett, Jim Smith, Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching, April 11, 1996

Vita

Nien Yi Lim was born in Ipoh, Malaysia on 8th September 1983. Previously, he attended Prime College in Subang Jaya and was a second year direct entry student at the University of Bristol, United Kingdom. He graduate with first class honors in Electronics and Communications Engineering in 2004. After graduation, he joined Motorola Inc. as a software engineer in November 2004. He worked at Motorola until July 2007 when he decided to pursue his MS degree in electrical engineering at the University of Kentucky.

He has published papers in conferences where the most recent one is at the International Conference on Parallel and Distributed Computing and Communication Systems in with a paper titled, "A New Reconfigurable Network Node Processor Architecture for Distributed Implementation of Ephemeral State Processing".