ABSTRACT OF THESIS

LINE ASSOCIATIVE REGISTERS

As technological advances have improved processor speed, main memory speed has lagged behind. Even with advanced RAM technologies, it has not been possible to close the gap in speeds. Ideally, a CPU can deliver good performance when the right data is made available to it at the right time. Caches and Registers solved the problem to an extent. This thesis takes the approach of trying to create a new memory access model that is more efficient and simple instead of using various “add on” mechanisms to mask high memory latency. The Line Associative Registers have the functionality of a cache, scalar registers and vector registers built into them. This new model qualitatively changes how the processor accesses memory.

KEYWORDS: Cache, Register, CReg, Line Associative Register - LAR, SIMD Within A Register - SWAR

Krishna Melarkode

October 11, 2004.

LINE ASSOCIATIVE REGISTERS

By

Krishna Melarkode

Dr. Henry G. Dietz
(Director of Thesis)

Dr. Yu Ming Zhang
(Director of Graduate Studies)

October 11, 2004

THESIS

Krishna Melarkode

The Graduate School

University of Kentucky

2004

LINE ASSOCIATIVE REGISTERS

THESIS

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

By

Krishna Melarkode

Lexington, Kentucky

Director: Dr. Henry G. Dietz, Professor
Electrical Engineering, Lexington, Kentucky

2004

ACKNOWLEDGEMENTS

My sincere thanks and heartfelt gratitude are due to my academic advisor and thesis director, Dr. Hank Dietz for his guidance and support throughout this thesis. I am very thankful for his constant encouragement and support during the different phases of this thesis. I would like to extend my thanks to Dr. J. Robert Heath and Dr. William Dieter for serving on my thesis committee and providing me with invaluable comments and suggestions for improving this thesis.

I extend my deepest gratitude and thanks to my parents for their support and belief in me. Finally, no word of praise is high enough for my friends who have encouraged and supported me throughout the course of my thesis work and my study at Kentucky.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF FILES

1. INTRODUCTION

In modern computers, processors always are faster than memories. Ideally, the system performance peaks when data is supplied to the processor at the speed of data execution. Even newer RAM technologies like DDR and RAMBUS cannot match processor speeds. Hence it is necessary to introduce architectural mechanisms to prevent the RAM from being the bottleneck for overall system performance. The standard solution is the introduction of a memory hierarchy in which small, fast, memories are at the top and slower, larger, memories are at the bottom.

The bulk of the hierarchy consists of multiple levels of caches – L1, L2, and L3. The larger the L1 cache, the better the system performance. However, cost is a very important consideration as the L1 cache is made of the most expensive SRAM and each cache cell is made of 4-6 transistors [5].

Registers are on top of the memory hierarchy. When viewed from the compilers point of view, there are many advantages in using registers over caches. However, they cannot handle ambiguously aliased values while caches can. CRegs – the cache register hybrid structures on the other hand offer all the advantages of a cache and a register and are compiler friendly. At the time when CRegs were introduced, the concept of data parallelism within a processor was deemed impractical due to the relatively high circuit complexity it implies. Even vector processors of the day were really just heavily pipelined systems; they did not really operate on all elements of a vector simultaneously. However, data parallelism within a processor is a common concept today and the idea of extending the CRegs to support the data parallelism is promising. *SWAR – SIMD within a register* was good step taken to build a complete SIMD programming model. It is believed that the functionality of SWAR can be improved by integrating it with the CRegs.

From an architectural point of view, microprocessors like the Pentium III have normal registers to do the scalar operations and vector MMX and SSE registers to do the parallel integer operations. They also have two or three levels of cache – L1, L2 and sometimes L3. The concept of *LAR – Line Associative Registers*, a set of vector registers that hold data type information in them, replaces all levels of Cache and all types of Registers with a simpler structure, but is expected to give comparable performance. A simple LAR architecture and its instruction set are proposed in this thesis. Chapter 4 introduces the basic idea of the LAR structure and architecture, Chapter 5 deals with the instruction set, in detail, Chapter 6 discusses a possible hardware implementation, Chapter 7 theoretically compares the results of a matrix multiplication algorithm implementation with SWAR and LAR concepts and Chapter 8 holds the conclusion, results and the direction of future work.

1.1. Motivation

The goals of this thesis are to develop architecture consistent with the following goals –

1. Simplify the Cache and memory hierarchies. In modern computers, the memory system is a fairly complicated structure with registers at the top of the hierarchy, followed by the L1, L2 and L3 caches, and RAM. We suggest that the functional complexity of modern memory access mechanisms is not necessary; a logically simpler design with comparable (or lower) total circuit complexity can yield comparable (or better) performance.
2. Modify a conventional register to incorporate more functionality and compensate for the absence of caches. This is done by introducing a new structure called Line Associative Registers that has the features of a cache and vector registers. It also holds data type information for the data it holds.
3. Improve performance of the system in the presence of aliasing in a multi-block cache. The LAR handles aliases efficiently.
4. Find a solution to vector register word boundary problems. In this architecture, the contents of the same register can be treated as a byte vector or a same-size half-word vector by appropriately adjusting the width of the ALU.

2. HISTORY

The LAR concept and architecture is the result of collection of a number of solutions developed for a variety problems viewed through different perspectives. However, the motivation for this work is broadly categorized into 3 channels –

1. Compiler issues, namely Aliasing, Register allocation and solutions in hardware.
2. Vector Registers and SWAR – SIMD within a Register.
3. Tagged memory architectures.

2.1 Compiler Issues – Solutions to Aliasing, Register Allocation and solutions in hardware

Aliasing is the problem that arises when a location in memory is referenced by multiple names. In a conventional system, aliased variables cannot be kept in registers across possibly-aliased updates. When aliased values are loaded into registers, the compiler must treat them as distinct values, which results in register copies of a memory location potentially having multiple different values. Hardware structures were proposed to deal with this problem. Stacks and registers were integrated to form a hybrid structure called Rack [1] that acted like a stack cache. However, it failed to provide a solution to the aliased value referencing problem and was not associative. Next were the CRegs [2], to reduce processor-memory traffic and handle ambiguously aliased variables.

The LAR inherits its associative load and store features from the CRegs. These were cache – register hybrid structures that exploited the advantages of registers and caches. The associative property of a cache, in particular, makes it feasible to profitably handle ambiguously aliased values. Results [4, 9] proved that CRegs greatly aided in reducing the memory traffic.

During the time of the development of CRegs, a fully associative cache was a substantial piece of hardware. Thus, although fully-associative CRegs were discussed, the reference design had only 4 sets of 4 way associative CRegs. If the ambiguously aliased values were outside the CRegs associative set, they were spilled. This led to new schemes where ambiguously aliased values were not promoted to registers were proposed. Variable forwarding proposed by Ben Heggy et.al [3] was one of them.

To exploit instruction level parallelism, compilers often employ static code scheduling but the presence of ambiguous aliases greatly restricted it. A hardware mechanism called Memory conflict buffer was proposed by David M Gallagher et.al. [7] in 1994, to facilitate static code scheduling in the presence of memory Load / Store dependences. A Smart Register File mechanism was proposed by Matthew A. Postiff and Trevor Mudge [8], which eases some of the extremely conservative compiler assumptions by introducing an indirect register access that simplifies alias handling and other optimization. When data is aliased through Smart Register Files, data is kept in the original register file entry and augments that with data information as opposed to address information in the CRegs. The Smart Register File concept is perhaps best viewed as an alternative implementation of CRegs in which data copies are replaced by this indirect reference mechanism; although such an implementation is viable, we suggest that the extra pipe stage implied by the indirection makes this a less-desirable solution.

Conventional compiler analysis cannot perform register promotion proving that a value is free of aliases at all times. So yet another new hardware structure based on CRegs was proposed by Matthew Postiff et. al. [10] in 2000, called SLAT – Store Load address table. The SLAT watches all load and store instructions to see if the they already match with the SLAT entries by explicit software mapping. When certain values cannot be promoted to registers with conventional compiler analysis, they can with SLAT. When aliased loads and stores are caught, a fix-up operation is performed.

Overall, the LAR concept here is perhaps best viewed as CRegs gone wide. Although caches have long taken advantage of spatial locality by using multi-word line sizes, and even instruction CRegs as originally proposed had multiple words per register, the combination of associative data registers and multi-word handling is new.

2.2 Vector Registers and SWAR – SIMD within a Register

Most media and 3D applications require repetitive operation on arrays of data. A new version of SIMD, called SWAR [16] was aimed to provide a general and consistent programming model. It makes use of the fact that wide data path within the processor can also be treated as multiple thinner SIMD-parallel data paths.

The concept of treating a register as a fixed-bit-length vector of data, the base concept of SWAR, was first implemented by HP in the Hummingbird PA-RISC processor. The first commodity SWAR incorporating a relatively complete instruction set was Intel's Pentium with MMX technology, which introduced the 64 bit integer MMX registers [14]. AMD built upon MMX with their 3DNow! SWAR floating-point technology [15], which was a set of 24 instructions to provide more detailed and sharper 3-D imaging along with micro-architectural enhancements. The next major addition to the IA32 instruction set was the XMM floating point registers with Intel's Streaming SIMD Extension technology. These are 128 bit floating point vector registers that also support Scalar operations on the low portion of each XMM register. One would expect that the scalar operations so implemented would be useless because the core IA32 instruction set implements similar operations, but IA32 floating point is done in extended precision using a stack model, so SSE scalar operations are significantly simpler to implement and hence outperform ordinary IA32 floating-point implementations by Intel. The SSE1 and SSE2 extensions operate on single precision double precision floating point values respectively.

These multimedia extensions were targeted for specific applications. In contrast, our goal is for LAR operations to be very general-purpose in both SWAR and scalar support. The simple instruction set described in detail in this thesis is not intended to be the ultimate such design, but a minimal starting point sufficient for our preliminary investigation of LAR architecture.

2.3 Tagged Memory Architectures

In tagged memory architectures certain bits in the memory hold information about the data type stored in that location. SWARD architecture [17] that made use of naming and the protection concept of capability based addressing [18] was one of the early architectures developed to enhance program reliability. It made use of tagged or typed memories. For example, if a certain location in memory is declared to hold an `integer`, a `double` or `float` cannot be stored in that location.

There are number of advantages in using tagged architectures. They are:

1. Instructions can be generalized. There can be a single instruction like add or subtract that operates on different operands like an `integer` and `float`.
2. Type conversions are automatically implemented. The addition of an `integer` and a `float`, for example, yields a `float`.
3. Type checking is automatically done. In certain instruction sets, it is not possible to perform certain operations on certain data types. These operations can be avoided. Certain sensitive data can be protected by avoiding some meaningless operations on them. For example, the classic bug of reading or writing data past the end of an array can be trapped by hardware.

We do not propose to use a tagged memory format, but instead tag each line of data with type information at the time it is fetched: our LARs are type-tagged.

3. CONCEPTS

Before the concept of LAR is introduced, it is essential to review some basic concepts that are essential in understanding the working of the LAR.

3.1 CACHE – A brief overview

A simple cache has data and address fields. The address field is further divided into an index and a tag as shown in fig 3.1a & 3.1b. The cache makes use of the concept of locality – which can be classified into spatial and temporal locality. **Temporal locality** of reference is the concept that a resource that is referenced at one point in time will be referenced again sometime in the near future. **Spatial locality** of reference is the concept that likelihood of referencing a resource is higher if a resource near it was just referenced.

Address	Data

Fig 3.1.a. A simple cache

Tag	Index	Data

Fig. 3.1.b. Simple Cache

Caches can be classified into direct mapped, set associative and fully associative. Direct mapped caches are the simplest of the three. In a direct mapped cache, a block can be placed in exactly one location. In a set associative cache, a block can be placed in fixed number of locations. In a two way set associative cache, data can be placed in two locations and in a four way set associative cache; data can be placed in four locations. In a fully associative cache a block can be placed anywhere in the cache.

Caches greatly reduce memory access time if the data referenced are already present in the cache -- a cache hit. A cache miss on the other hand considerably affects performance. Optimizations can be made to reduce the number of cache misses. However, cache misses can never be completely eliminated because the associativity of the cache is in the same namespace used to identify a cache line; in other words, the compiler would have to resolve all potentially ambiguous aliases (a problem discussed in detail below) and know the tag hash function in order to predict when two different cache references may interfere with each other. As a result, predictable access time is rarely possible for a cache. A cache miss involves some memory traffic. One of the most interesting advantages of using a cache is that ambiguously aliased values can be profitably stored in caches. Aliases are discussed in detail in the following sections.

False sharing is a problem commonly associated with cache-based multiprocessor systems. In multiprocessor systems with caches with multi-word cache blocks, false sharing is the coherency overhead as a result of multiple processors accessing different words in the same block. Even if a processor re-uses a data item, the item may no longer be in the cache due to an intervening access by another processor to another word in the same cache line. Consider a multi-processor system involving four microprocessors – P1, P2, P3 and P4 and a cache line that has four words of data - A, B, C and D as shown in fig 3.3. Processor P1 tries to accesses the value C, P2 tries to accesses B, P3 tries to accesses D and P4 tries to accesses A. When processor P1 tries to access C, it might not be in the cache line as P2 is trying to access B. The same argument also applies to other processors and memory words.

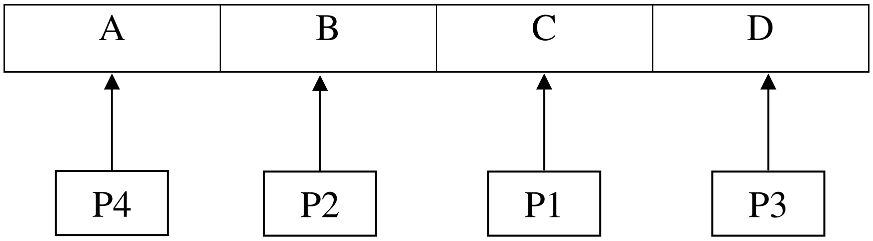


Fig 3.3. A multiprocessor system with 4 processors trying to access a cache line

3.2 REGISTER – a brief overview

A simple register has a name and a data field. It is on top of the memory hierarchy chart. A simple register is shown in fig 3.4.

Name :	5204

Fig 3.4. A simple Register

A value stored in a register can be accessed fast in turn reducing latency. A value in a register can be referenced with no interference with the memory processor path. As a result, the usable memory bandwidth increases. Register access time is predictable as there are no hit-miss issues. Once data is loaded into a register, the probability of finding it there is a 100 percent which greatly aids in compile-time code optimizations. In a system, there are always a small number of registers. Once a value is in a register, it is easier to reference it with a short register name than a long memory address. Consider a simple memory with data `5204` in address `15129714` as in fig 3.5. Consider a simple example of adding the value 10 with contents of memory location `15129714` and storing it back in the same location. It can be done as shown below

Address	data
15129714	5204

Fig 3.5. A simple data memory cell

```
Memory [15129714] = Memory [15129714] + 10
```

However, when the value at `15129714` is loaded into a register, `r2`, the assembly code is -

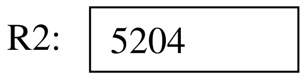


Fig 3.6. A simple register

```
R2 = Memory [15129714]
R2 = R2 + 10
```

It is observed that the second piece of code looks clean and has only one memory reference. There are some issues when using registers. A value cannot be stored in a register permanently as there are only a limited number of registers in a system. Not all values can be stored in registers. Consider the same example discussed above. The value at location `15129714` is referenced with arrays of different indices – a[i] and a[j]. It is possible that i and j point to the same location – for example, when i = j = 2. Such values should be loaded into a register only when the compiler ascertains that a[i] and a[j] definitely point to different or same locations and if not, the two array references are said to be *ambiguously aliased*. If such values are loaded into two different registers, it would result in an ugly situation where a single memory location has two different values. The primary drawback of using a register is that it cannot handle ambiguously aliased values.

3.3 Basic Compiler Problems - Register Allocation and Aliasing

The placement of data items into registers is called register allocation. It is a very important optimization as registers are the fastest storage devices in the entire computer system. In any processor architecture there are only a limited number of registers and it is necessary to decide which variables go into registers. It is not profitable to hold a variable in a register permanently. The variables are promoted to registers for the regions in their lifetimes when there are no aliases [11, 12, 13]. A value is promoted to a register for that region by a load instruction at the top of the region just once. It is demoted back to the memory by a store at the bottom of the region just once. In between the promotion and demotion, all the accesses are through the register. The process of register allocation is greatly affected by *aliasing*.

Aliasing is the problem that arises when the value in a variable is referenced by more than one name.

```
E.g.1  int a;          (1)
       int *p;         (2)
       p=&a;           (3)
```

In example 1, variable ‘a’ and pointer ‘p’ refer to the same memory location.

```
E.g.2  int a[i];       (1)
       int a[j];       (2)
```

In example 2 both statements may or may not point to the same memory location. If they point to the same location it is called ambiguous alias. Consider a piece of code as in example 3.

```
E.g.3  int b;
       int *c;
       c=&b;
       …..
       *c=*c-10;
       print b;
       b=14;
       …...
       print *c
```

The memory location that holds the value of variable `b` is referenced by more than one name. The variable `b` cannot be allocated to a register as its value can me modified with the pointer `c`. The subtraction of `*c` cannot be completed before initializing the value of `b`. Thus, such values cannot be stored in a register that has only one name.

Aliasing happens in languages like C, FORTRAN and Java. Data at a memory location can be kept in a register only when it can be assured that the data in a memory location can be accessed by accessing a register instead. It is impossible to predict the memory location to which an instruction points as instructions compute the address at the time of execution. It is possible for two or more registers to hold different values of the same variable since it is referenced by different names. From example 1, it is incorrect to allocate statements 1 and 2 to two different registers.

The compiler examines the program before execution and cannot determine if there is going to be address aliasing when the program runs. So it has to make conservative assumptions, which means that data cannot be placed in registers for at least a part of their life time. From example 2, it is incorrect to allocate statements 1 and 2 to different registers unless they are disjoint memory locations.

From the above discussions, it is essential to ensure that variables are allocated to registers in their non-aliased region. But such allocations do not last long as a change in memory value with a different name will not reflect a change in value in register. So every time the value in a memory is updated, the register should be reloaded from memory.

3.4 IMPORTANCE OFALIAS ANALYSIS

Alias analysis is important because of the following reasons –

1. It enables optimizations to the program to be applied correctly.
2. It is used to determine potential data dependencies.
3. It determines if a variable can be allocated to a register or not.
4. It determines if code transformation is legal or not.

Aggressive alias analysis aids in good optimizations.

3.5 The SIMD/SWAR Models

SIMD stands for Single Instruction Multiple Data. There are some applications where the same operations have to be performed on different sets of data. A classic example is the inversion of a RGB picture to produce its negative, where the same operations have to be performed over an array of uniform integer values. A SIMD model is handy in such cases. Microprocessors were originally designed to be SISD – Single Instruction Single Data models (fig 3.7), but have adapted and incorporated a variant of SIMD called SWAR (SIMD Within A Register), as shown in figure 3.8.

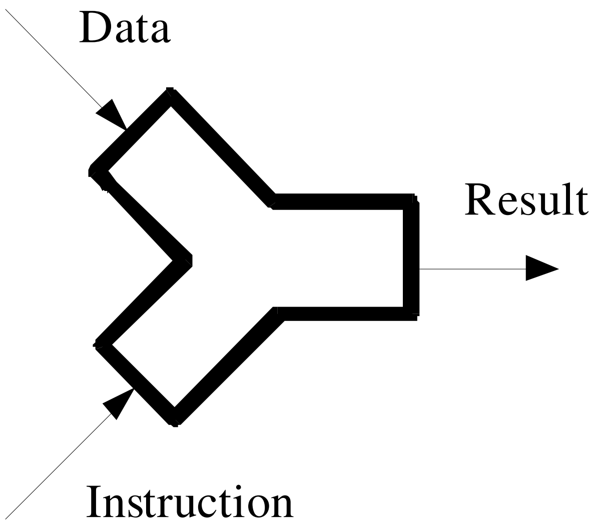


Fig 3.7: SISD Model

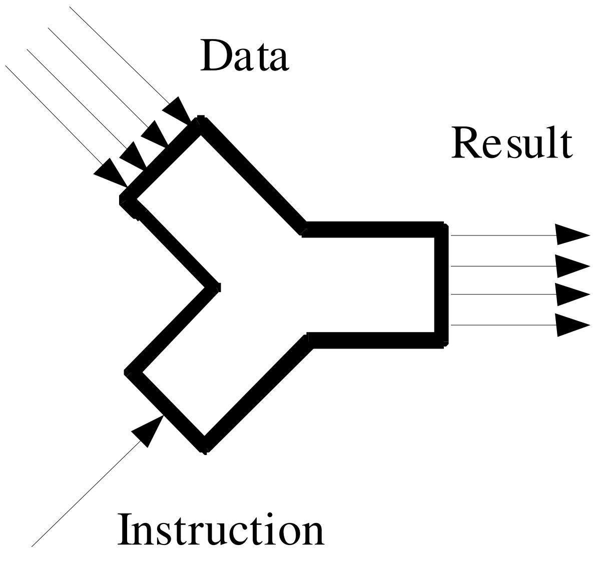


Fig 3.8: SIMD/SWAR Model

The SIMD model makes use of data parallelism. To implement the SWAR model, various special hardware and instructions have been developed – such as the MMX, 3D-Now, SSE and SSE2 instruction set extensions.

3.6 Special Purpose MMX Registers

MMX was the first technology introduced to implement the SWAR model as an extension to IA32. It was designed to accelerate the performance of advanced media and communication applications. There were eight 64-bit integer MMX registers that held integers of different sizes and performed a variety of operations on these registers. The MMX

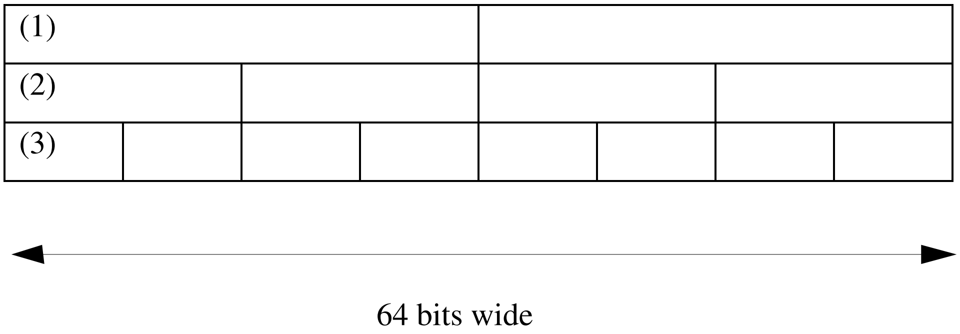


Fig 3.9: MMX Registers – supported data types

Each MMX register could hold either 8 packed bytes as in line (1), or 4 packed words as in line (2), or 2 packed double words as in line (1) as shown in fig 3.9.

3.7 Special Purpose XMM Registers

The XMM Registers were introduced with the SSE extension to the IA32 architecture. The MMX extension allowed SWAR operations to be performed on packed integers. SSE extends the functionality of the SWAR implementation by adding facilities to work on larger vectors of packed and scalar single precision floating point values. Eight 128-bit registers, called XMM registers. were introduced. These registers performed SWAR operations on four single precision floating point values stored in the XMM registers. The SSE extension greatly helped computation-intensive repetitive operations for media and communication applications. The XMM registers cannot be used to address memory. Scalar operations are performed on individual single precision floating point values at the lower double-word of XMM register.

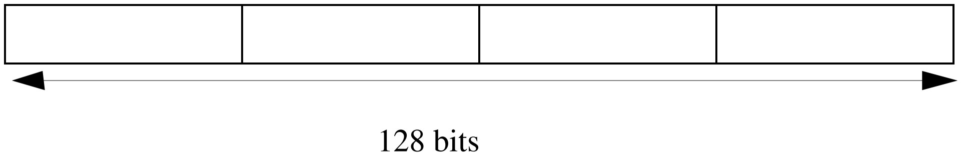


Fig 3.10: Floating point XMM register that holds 4 floating point values

The SSE 2 extension further extended the functionality of SSE by enabling it to operate on double precision floating point values. This capability enhanced scientific and engineering applications and applications that involved complex 3-D geometry techniques. Additional flexibility is provided to scalar operations.

3.8 CREGs

The CReg is a Cache-Register hybrid structure that exploits the advantages of both caches and a registers. It has all the advantages a register has and it is associative like the cache and can efficiently handle ambiguous aliases. A CReg is shown in fig 3.11.

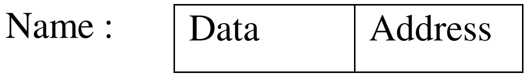


Fig 3.11. A CReg Structure

The CReg array is very similar to a simple 1 block cache. It has a name, a data and an address field. When a CReg is referenced, an associative search is performed with the neighboring CRegs, just as in a cache, for a matching address. A match found as a result of the search is an alias of the original CReg. However, CRegs avoid making memory references on aliased loads by using duplicate entries in the CReg array.

A simple example where a memory location is accessed through multiple forms in C is considered. Variables ‘a’ and pointer variable ‘p’ point to the same memory location.

```
int a, *p;
p = &a;
a = a +10;
```

Address	data
4124	25

3.12a Contents of memory location 4124

	CReg	Address	Data	Dirty
p:	R0	4124	25	1
a:	R1	4124	25	1

Fig 3.12.b. CReg Contents before the Add

	CReg	Address	Data	Dirty
p:	R0	4124	35	1
a:	R1	4124	35	1

Fig 3.12.c. CReg Contents after the Add

These two values are aliases. A conservative compiler does not allow the loading of these values into ordinary registers as these values are aliases. However, they can be loaded into CRegs and handled efficiently. Variable a is loaded into R0 and p is loaded into R1 as shown in fig 3.12. Whenever the content of p is referenced or changed, the content of a also is updated. A match in address sets the dirty bit of the matching CReg to ‘1’.

4. LAR

LAR stands for Line Associative Registers. This is a new hardware structure that inherits its features from other hardware structures previously discussed. A data LAR is a CReg with a multiple data fields that also performs scalar operations. It has the features of SWAR and also holds data size and type information in it. Its structure is shown in fig 4.1.

Data	Address		Wd-Sz	Type	Dirty
	Base-Pt	Offset			

Fig 4.1. LAR structure

It is more efficient than the cache as the hardware does not have to decide which line to replace with schemes like the LRU. The hardware is explicitly told where to make each entry. It is fully associative, unlike SWAR designs such as MMX, XMM and SWAR. Moreover, each register holds data type information about the data contained in it. It can handle vector operations and there is a lot of flexibility when it comes to scalar operations. Unlike the XMM registers, scalar operations can be performed at any location in a LAR.

4.1 Assumptions

In the discussion to follow, the following assumptions are made

1. There are 32 instruction LARs.
2. There are 32 data LARs.
3. The data LARs handle only on integers.
4. Instructions are fixed length – 32 bits each.
5. Memory word size is a byte.

6. The address field size depends on memory size. Memory is byte addressed and the size is left open to the hardware design. Irrespective of the size, the last five bits of the address is always the line offset.

4.2 The Instruction LAR

The instructions from the instruction memory are loaded into the instruction LAR with the fetch instruction and start getting executed serially. The fetch instruction is analogous to the pre-fetch of a conventional processor. However, in a fetch, the block of instructions to be fetched can be explicitly specified.

Instructions (1024 bits wide)	Address

Fig 4.2. Instruction LAR structure

The instruction LARs are represented by an `i` followed by a number (`i0`, `i1`, `i2`, … `i31`). The structure is shown in fig 4.2. There are 32 instruction LARs, each 1024 bits wide, that accommodates 32 instructions. Each instruction LAR also has an address field. Instruction blocks are loaded into the instruction LARs from the instruction memory. The Instruction field of 32^{nd} instruction LAR acts as the instruction register and its address field acts as the program counter (Fig 4.3). There are two bits that specify the number of lines to be loaded, thus, a maximum of four lines of instructions (32 x 4) can be loaded with a single fetch. It is sufficient if the address of the first LAR is specified. If the address of the destination LAR matches any of the already-loaded LARs, the load from memory is cancelled and the instructions are copied from the matching LAR.

Instructions (1024 bits wide)	Address
I0	400
I1	800
I2	432
...	...
I10	640
...	...
I31 IR	PC

Fig 4.3. Instruction LAR file

Once an instruction LAR is loaded into i31, the instructions start getting executed serially. The program counter increments itself after every instruction. From fig 4.3, assume that instructions in I0 are getting executed. PC points to 400 at the first instruction and 431 at the last instruction of I0. If I2 has instructions starting from memory location 432, I2 gets copied next into the IR and it starts getting executed. The sequential execution of instructions breaks only when a jump is encountered.

The *select* is the instruction to handle the jumps. With this instruction, the PC can select between two LARs based on a condition. For example, a select instruction could cause either I1 or I10 to be executed next based on the value in a data LAR. An important point to note is that a select can only jump to the start of a different line. It is not possible to jump to an instruction that is in the middle of a line.

4.3 The Data LAR

As mentioned before, this architecture does not have a data cache or special vector registers. The data LARs perform the functions of both. The loads handle most of the memory traffic in this architecture. The stores are performed automatically – the function of a store instruction in this architecture is different altogether. They are used to duplicate the contents of a data LAR line or can be used to perform data type conversions.

DATA								ADDRESS		WD SZ	TYP	DTY
D[7]	D[6]	D[5]	D[4]	D[3]	D[2]	D[1]	D[0]	BASE PT	OFFSET			
256							0	400	3	2	1	1

Fig 4.4. Data LAR structure

A data LAR is represented by a “d” followed by 0 -31 (d0, d1, d2 …..d31). There are 32 data LARs. The structure of a data LAR is shown in fig 4.4. It has a data, an address, an offset, a word size, a type and a dirty field. The data field is 256 bits wide that can store eight memory words, 32 bits each. The address field is subdivided into a base pointer and an offset field. The last 3 bits of the address field is the offset. This is like a multi-word cache line that has a tag, index and offset to locate a word in the middle of a line. The word size field, 2 bits wide, stores the size information of the data in that line. It can be either a byte (8 bits), a half-word (16 bits), a word (32 bits) or a double-word (64 bits). The type is a 1 bit field that stores the sign information of the contents of the line. The type bit is ‘1’ when the contents of the line are signed and ‘0’ when contents are unsigned. The word size field bit setting for different word sizes are shown in table 4.1.

BIT SETTING	WORD SIZE
00	Byte
01	Half word
10	Word
11	Double Word

Table 4.1. Word size bit settings for different word sizes

The data LAR ***d0*** is different from the others and it requires a special note. It is analogous to R0 of the MIPS architecture. It is permanently set to ‘0’, i.e., all the fields of d0 are set to ‘0’. All the 8 data fields of d0, the base pointer and the offset fields are set to ‘0’, to be more specific. There is never a word size field for d0 and it is always a source.

It can never be a destination, i.e., it cannot be loaded with data like the other LARs. While implementing the hardware, a write to d0 can be used to implement a no-op.

The data LARs are loaded with the load instruction. Every load instruction calculates an effective address from where data from the data memory is loaded. The calculated effective address is compared with the address fields of the other LARs. If there is a match, the load from memory is cancelled and the data is copied from the matching LAR. Every time the data in a data LAR changes as a result of an ALU operation, it gets updated with the memory to maintain consistency. The updating process can occur whenever there is a free bus cycle. This is the reason why stores are not required. The dirty bit is a 1 bit field which is usually ‘0’ but gets set to ‘1’ when the contents of an already loaded data LAR changes after an ALU operation. For example, in the instruction below, when d1 is already loaded with a load operation, sets the dirty bit if d1 to ‘1’.

```
Loaduw d1, d0, d0, 804
Adds d1, d1, d2
```

After main memory has been made consistent with LAR contents, the dirty bit gets reset to ‘0’. The Line Associative Register file is shown in fig 4.5.

Data	Addr	Wd-Sz	Type	Dirty
d0				
d1				
d2				
….				
d31				

Fig 4.5. Line Associative Register file

4.4 Highlights

The LAR concept has many advantages when compared to previous structures -

1. They can efficiently handle aliases, as they have been derived from CRegs. They handle aliases the same way as CRegs.
2. Scalar or Vector ALU operations are performed on words, parts of words, and multiple words with the same set of LARs.
3. Scalar operation can be performed on aligned data residing in any location in the LAR. It is possible to work on any word in a line with any word in another line without affecting the other words, as shown in fig 4.6. This flexibility is not available in any other vector register. Although the XMM registers allow scalar operation, they are restricted to always access the lower bits. The restriction to access only the low object in a register not only requires shifting, but also would destroy the surrounding data if applied to a LAR because of the association of an address with the LAR. Thus, the ability to access anywhere within a LAR is critical not only for performance, but also for correctness.

	Data	Address	W-Sz	Type	Dirty
d4	***42*** ………………. 24 34 32 56	400	8	0	0
d18	………… 424 ***345*** ………… 532 109	1200	16	0	0

Fig 4.6. A scalar operation can be performed on the italicized words

4. They store type information in them. Attempts were made to store data type information only in the memory till now. They offer atomicity in handling data types. This is explained with an example. Consider an example where the data from a location, 400, is

	Data	Address	W-Sz	Type	Dirty
d4:	00100101 11110000 10001010 10111000	400	8	0	0
d18:	0010010111110000 1000101010111000	400	16	0	0
d28:	00100101111100001000101010111000	400	32	0	0

Fig 4.7 LAR treating data of a memory location as different word sizes

required at different locations of a code segment as different word sizes. When required as a byte, the whole line is loaded into d4 and word size is set to a byte. Any reference to d4 will refer to the contents of location 400 as bytes. When the same contents are required as half-word in a different segment of the code, the same contents are copied into another LAR, d18, and the word size is set as half-word as shown in fig. Any reference to d18 will refer to the contents of location 400 as half-words. Similarly, by copying the same value to d28 and setting the word size of d28 to a word, the contents of memory location 400 can be accessed as a word. Whenever the contents either one - d4, d18 or d28 change, it also updates the value of the others.

5. They also perform type conversions (or at least type relabeling or casting). Two LARs with data size of bytes can take participate in an ALU operation and the result can be converted to a word with a *Store* instruction.
6. It solves the problem of false sharing (discussed in the previous chapter) because different references within the same line, whether to the same or different type of object, always can have separate LARs allocated without performance penalties.

5. INSTRUCTION SET

The instruction set has 32 instructions, each 32 bits wide. There are 32 instruction LARs and 32 data LARs, thus a 5-bit field can encode a LAR number. The opcode also is encoded in a 5-bit field. There are basically 4 types of instructions

1. Data Transfer
2. Fetch
3. ALU
   a. Scalar
   b. Vector
4. Select

The data transfer and the fetch instructions have a similar format and have similar functionality. The ALU instructions have the same format as the select, but differ considerably in functionality.

The Data transfer operations transfer data between the memory and the data LARs. The load instruction is the primary data transfer operations. The load instruction loads values from the main memory into a data LAR. The store operation has a different functionality than the conventional store. It performs type and size conversions in the data LARs.

The fetch has functionality similar to that of a conventional pre-fetch, but it is generally a mandatory operation, not a performance-enhancing suggestion as pre-fetch is commonly used. Fetch is the only instruction with which instructions can be loaded into instruction LARs. With a single fetch instruction, a maximum of four consecutive instruction LAR lines can be loaded.

The ALU operations have two forms – scalar and vector. They include the basic arithmetic operations -add, subtract, multiply and logic operations like AND, OR and EXOR in both scalar and vector forms.

The Select operation is used for branches. It selects between two instruction LARs based on the value present in a data LAR. All these instructions are explained in the following sections.

5.1. Data Transfer Operations

As mentioned in the previous section, the data transfer operations move data between the memory and the data LARs. Load is the primary data transfer instruction. A load operation loads an entire LAR (256 bits of data or 8 memory words, 32 bits each) from the memory at a time.

OPCODE	DST	SRC1	SRC2	IMMEDIATE
5	5	5	5	12

Fig 5.1. Data transfer instruction format

The Load / Store instruction has the format as shown in fig 5.1. The opcode is a 5 bit field which selects 1 of the 32 instructions. The destination field (DST) selects 1 of 32 data LARs as the destination. The SRC1 and the SRC2 point to two other data LARs and the immediate is a 12 bit field. It can be any value between 0 – 4095.

5.1.1. The Load Instruction

The load instruction operates on a variety of data types -

1. Load unsigned byte
2. Load signed byte
3. Load unsigned half-word
4. Load signed half-word
5. Load unsigned word
6. Load signed word
7. Load unsigned double word

8. Load signed double word

All these types are basically two's-complement integer bit patterns, the only difference being the word size and sign tag that they assign to the data LARs. Floating-point types also can be supported in the same way, but have been omitted from the preliminary design in order to facilitate prototype implementation.

The load operation has 3 steps –

1. Calculate the effective address
2. Compare the effective address with the address field of the other LARS
3. Transfer data to the destination from the matching LAR or the data memory.

The effective address of the destination can be calculated by adding

Ea. destination = d[source1]. Address + d[source2]. Data + Immediate

This is explained with the following example.

```
Loadub d4, d0, d0, 122
```

Every data LAR points to an address and data in that address. The above instruction can be interpreted as

`d4`. Destination = `d0`. Address + `d0`. Data + Immediate

The address and data fields of d0 are always '0'. So the effective address is

`d4`. Destination = 0 + 0 + 122

Once the effective address is calculated this is rounded to the closest multiple of 32 words before it. The remainder is the offset. For example, effective address 122 is split into a base pointer with the value 96 and the offset 26. This is more meaningful from the hardware's point of view. The last 5 digits in the binary form are the offset and the remaining digits form the base pointer.

$$(122)_{10} = (11\ \mathbf{11010})_2$$

Base pointer: $(1100\ 000)_2 = (96)_{10}$

Offset: $+ (11010)_2 = (26)_{10}$

DATA								BASE PTR	OFFSET	WD SZ	TYPE	D
127	…	122	…	…	98	97	96	96	26	8	0	0

Fig 5.2. LAR `d4` after a Load unsigned byte operation

Since the load is of unsigned type, the type is set to 0. The data LAR `d4` after the load is shown in fig 5.2.The base pointer of the destination is compared with the base pointer of the other data LARs. If there is a match, the load from memory is cancelled and the data from the matching data LAR is copied into the destination. Neither the size nor the type of the destination is considered while performing an associative load. It is sufficient if there is a match in address as shown. Figure 5.3 shows this case where the matching LAR has a data of size 16 while the destination is of size 8. If there is no match in address, data is loaded from the main memory. A load from memory results in loading thirty two bytes from contiguous memory locations starting from the base pointer.

DATA								BASE PTR	OFFSET	WD SZ	TYPE	D
152	151	150	…	…	122	121	120	120	7	8	0	0

Destination data LAR

⇧ ⇧

Matching data LAR

DATA								BASE PTR	OFFSET	WD SZ	TYPE	D
152	151	150	…	…	122	121	120	120	10	16	0	0

Fig 5.3. Associative load from an address matching LAR

The size and type information for a data LAR is set by the load instruction. The load byte, load half-word, load word and load double-word set the word sizes to 8, 16, 32 and 64 respectively. The type can be signed or unsigned. A load unsigned word sets the type as unsigned and a load word sets the type to be signed. The dirty bit is normally at ‘0’. The dirty bit is set to ‘1’ when the contents of a data LAR change after they are loaded. Once memory has been made consistent with the LAR contents, the dirty bit is reset to‘0’.

5.1.2 The Store Instruction

The store instruction operates on a variety of data types -

1. Store unsigned byte
2. Store signed byte
3. Store unsigned half-word
4. Store signed half-word
5. Store unsigned word
6. Store signed word
7. Store unsigned double word
8. Store signed double word

The LAR architecture is designed in such a way that stores are unnecessary. Every time there is a change in the contents of a data LAR, data is automatically updated (or scheduled for lazy update) with the memory and other matching LARs to keep data consistent. The Store instruction has more functionality than a regular store. It is performs type casting – it changes the address, size and type information of a data LAR.

There are 2 steps in a store operation –

1. Calculate the effective address.
2. Replace the address, size and type pointed by the data LAR.

This is explained with the following example.

```
Loadub d4, d0, d0, 120
Storeuw d4, d0, d0, 200
```

The first instruction loads `d4` with data from memory location 120 or from some other matching data LAR and sets the base pointer, offset, word size and type fields to 96, 24, 8 and 1 (signed) respectively as shown in fig 5.4. The second instruction does nothing to the data contents of the LAR; the type conversion is simply a relabeling of the same bit pattern as having a different type. It changes the address, size and type information of d4.

d4:

DATA								BASE PTR	OFFSET	WD SZ	TYPE	D
127 10	…	124 30	…	104 80	…	97 50	96 60	96	26	8	0	0

Fig 5.4. LAR d4 after the Load instruction

The Store works in the same way as a load till a certain point. The effective address is calculated in the same way as in the load.

Effective address d4 = d0. Address + d0. Data + Immediate

It would be worthwhile to note that d0.addr and d0.data are always ‘0’. So the effective address would be

Effective address d4 = 0 + 0 + 200

The calculated effective address at the destination is compared with all the other data LARs, if there is a matching LAR, the address field of the matching LAR are also updated before the value is written into memory.

d4:

DATA								BASE PTR	OFFSET	WD SZ	TYPE	D
223 10	…	220 30	…	200 80	…	193 50	192 60	192	8	32	0	0

Fig 5.5. LAR d4 after the Store Instruction

After executing the store instruction, d4 appears as in Fig 5.5. The base pointer, offset, size and type fields have new values. The word size is now 32 and the type is unsigned.

5.2. Fetch

The fetch operation loads the instruction LARs with instructions from the instruction memory. A fetch operation loads an entire instruction LAR with instructions. For our example design, an instruction LAR is 1024 bits wide and it can accommodate 32 instructions. With each fetch operation, 1, 2, 3 or 4 instruction LARs can be loaded.

The Fetch instruction has a format similar to a load.

OPCODE	DST	SRC1	SRC2	IMMEDIATE	
				NUM	IMM
5	5	5	5	2	10

Fig 5.6. Fetch Instruction Format

The opcode is a 5 bit field and has a particular bit setting for a fetch instruction. The DST refers to an instruction LAR to be the destination. The SRC1 refers to the data pointed by a data LAR and SRC2 refers to the address pointed by an instruction LAR. The Immediate field is split into 2 – a 2 bit field to denote the number of contiguous instruction LARs to be loaded and a 10 bit regular immediate field that can be assigned an integer value between 0 – 1024.

The fetch instruction has 3 steps –

1. Calculate the effective address for the instruction memory
2. Determine the number of contiguous instruction LARs to be loaded
3. Compare the effective address of the destination with the other instruction LARs. If there is a match, cancel the load from memory and copy the instructions from the matching LAR. If there are no matches, load instructions from the instruction memory into the destination

The instruction LAR has 1024 bits to store instructions plus an address field. Every address in the instruction LAR is multiple of 32 words. If the effective address is calculated to be 6235 the corresponding value in the address field would be 6208 i.e., (6235 / 32) * 32. The instruction LAR holds 32 instructions starting from location 6208. The fetch instruction is explained in detail with the following examples.

```
Fetch i4, d0, i6, 900
```

Before calculating the effective address it is important to separate the immediate field into the number and immediate fields.

$$(900)_{10} = (\mathbf{00}1\ 110\ 000\ 100)_2$$

In this case, the binary equivalent of 900 has just 10 digits. The number of instruction LARS to be loaded depends on the 11th and 12th bits of the immediate field. As a result the immediate field is padded with zeros till the 12th digit. Since the most significant 2 bits of 900 are '00' only 1 instruction LAR line is loaded. The number of instruction LARs loaded for different 11th and 12th bit settings are listed in the table 5.1.

12th and 11th bits	# of Inst LARS Loaded
00	1
01	2
10	3
11	4

Table 5.1. Bit settings and the number of instruction LARs loaded

The effective address is first calculated using

Eff Addr[DST]= d [SRC1] .data + i [SRC2].addr + immediate

Assuming i6 has an address field set to 96, the effective address would be 996 and the address of the instruction LAR i4 is 992. The first instruction at I4 would be the instruction at instruction memory location 992. There would be 32 contiguous instructions starting from contiguous memory locations 992.

	INSTRUCTIONS (1024 bits wide)	ADDR
I4:	I@1024 - I@992	992
	…	…
I31:	INSTRUCTION REGISTER	PRG CTR

Fig 5.7. Instruction LARS

Consider the instruction below. Proceeding in the same way as before

```
Fetch i4, d0, i6, 2085
```

$$(2085)_{10} = (\mathbf{10}0\ 000\ 100\ 101)_2$$

In this case the 11th and 12th bits are 0 and 1 respectively. From the table 5.1, it can be inferred that 3 instruction LARs are to be loaded. For this case the effective address is 128, adding $(100\ 101)_2$ with $(96)_{10}$. The instruction LARs after loading are shown in fig 5.8.

	INSTRUCTIONS (1024 bits wide)	ADDR
I4:	I@128 - I@159	128
I5:	I@160 - I@191	160
I6:	I@192 - I@224	192
	…	…
I31:	I31: INSTRUCTION REGISTER	PRG CTR

Fig 5.8. Instruction LARs after executing the Fetch

5.3. ALU Operations

The general format of the ALU operations is shown in fig 5.9. The format is very similar to a Load / Store; however there is no immediate field. The instruction is still 32 bits and the immediate field is padded with zeros (or could be used to provide extended opcode bits, much as is done in the MIPS instruction set encoding). The basic ALU operations are

1. Add
2. Subtract
3. Multiply
4. AND
5. OR
6. EXOR

Each one of the above has a Scalar and a vector form. Both the scalar and vector ALU operations have the same format.

OPCODE	DST	SRC1	SRC2
5	5	5	5

Fig 5.9. Format of ALU operation

5.3.1. Scalar ALU Operations

The Scalar ALU operations operate on selected data fields of the data LAR. They operate only on the data pointed by the address field (base pointer and offset together) of the data LAR. The Scalar ALU operations have 3 steps –

1. Determine and isolate the data pointed to by both source data LARs, taking the word size into account
2. Operate **only** on the isolated data from both the sources
3. Replace **only** the data pointed by the address of the destination data LAR with the sum. Word size of destination plays an important part in determining the result as it sets the ALU field boundaries

The scalar ALU operation ADD is explained in detail with an example.

```
Loadub d1, d0, d0, 68
Loaduw d2, d0, d0, 130
Adds d2, d1, d2
```

	DATA								BASE PTR	OFFSET	WD SZ	TYPE	D
d1:	95 55	.. 	.. 	69 130	**68** **120**	.. 	65 32	64 24	64	4	8	0	0
d2:	159 126	..	**133 132 131 130** **800**				129 120	128 119	128	2	32	0	0

Fig 5.10. LAR d1 & d2 before executing the Add instruction

In the above example d1 is one source and d2 is the other. The data LAR d1, points to base address 64 and d2 points to 128 as shown in figure 5.10. Both d1 and d2 hold unsigned values. The destination is d2. The address pointed by d1 is 68 and the address pointed by d2 is 130. It is assumed that the data at location 68 is $(120)_{10}$ and four bytes starting from 130-133 together store $(800)_{10}$. The word size of d1 is 8 and the word size of d2 is 32. The Scalar ADD adds only the two values - $(120)_{10}$ and $(800)_{10}$. The remaining values in d1 and d2 remain untouched.

The destination d2 has a size of 32. The ALU boundary is set to 32 bits. So marked 8 bits of d1 are added with the marked 32 bits of d2 and the result replaces the originally marked contents of d2. From this example 920 replaces 800 as shown in figure 5.11.

<table>
<tr><th></th><th colspan="8">DATA</th><th>BASE
PTR</th><th>OFFSET</th><th>WD SZ</th><th>TYPE</th><th>D</th></tr>
<tr><td>d1:</td><td>95
55</td><td>..</td><td>..</td><td>69
130</td><td><b>68</b>
<b>120</b></td><td>..</td><td>65
32</td><td>64
24</td><td>64</td><td>4</td><td>8</td><td>0</td><td>0</td></tr>
<tr><td>d2:</td><td>159
126</td><td>..</td><td colspan="4"><b>133 132 131 130</b>
<b>920</b></td><td>129
120</td><td>128
119</td><td>128</td><td>2</td><td>32</td><td>0</td><td>0</td></tr>
</table>

Fig 5.11. LAR d1 and d2 after executing the Add instruction

The data LARs d1 and d2 were previously loaded with some values and after the ALU operation, the value in d1 has changed. To make a note of this the dirty bit of d1 is set to 1. The address of the destination is also checked with other data LARs and if there is a match, the new value of d1 is also copied to the matching LARs. This dirty bit becomes 0 when the contents of d1 are written back into memory.

This tagged-type field extraction may seem complex, but actually is no different from the extraction done using a conventional level-one cache when a processor makes a byte or other non-native-word-size reference. Further, although the promotion semantics may seem complex because they allow, for example, addition of two 16-bit values to store a legitimate17-bit result in a 32-bit destination, this is precisely the same behavior obtained when a similar operation is performed in a convention processor with word-size registers.

5.3.2. Vector Parallel ALU Operations

The Vector ALU operations operate on **all** the data within a data LAR. The Vector ALU operations have 2 steps –

1. Determine the word sizes of both the source data LARs and perform the ALU operation on all the data fields of the two sources, aligning the least significant bits of both sources
2. Determine the word size of the destination LAR and set the word size markers to the ALU to set word boundaries and the result appropriately to point to the right data size.

The Vector / Parallel ALU instruction Add is explained with an example

```
Loaduw d1, d0, d0, 72
Loaduw d2, d0, d0, 328
     Addp d1, d1, d2
```

In this example, the first instruction loads d1 with 8 words of data starting from memory locations 64. The second instruction loads d2 with 8 words of data starting from memory locations 320. The data-LAR d1 is a source and d2 is the second source and also the destination. The sources d1 and d2 point to four bytes of data starting from 72 and 328 respectively as shown in fig 5.12.

	DATA								BASE PTR	OFFSET	WD SZ	TYPE	D
d1:	92 127	88 126	84 125	80 124	76 123	**72** **122**	68 121	64 120	64	8	32	0	0
d2:	348 726	344 725	340 724	336 723	332 722	**328** **721**	324 720	320 719	320	8	32	0	0

Fig 5.12. LARs d1 and d2 before the vector add instruction

The address field – base pointer and offset are not important to the Vector ALU operations as they operate on all the data fields of the source data LARS. However, it is important to take the word size of the destination into consideration as it decides word boundaries in the ALU for the ADD operation. The sum of d1 and d2 replace the contents of the destination d1 as shown in figure 5.13.

	DATA								BASE PTR	OFFSET	WD SZ	TYPE	D
d1:	92 853	88 851	84 849	80 847	76 845	**72** **843**	68 841	64 839	64	8	32	0	0
d2:	348 726	344 725	340 724	336 723	332 722	**328** **721**	324 720	320 719	320	8	32	0	0

Fig 5.13. LARs d1 and d2 after the add instruction

The data LARs `d1` and `d2` were previously loaded with some values and after the ALU operation, the value in `d1` has changed. However, the memory location still holds the old values. To make a note of this the dirty bit of `d1` is set to 1. The address of the destination is also checked with other data LARs and if there is a match, the new value of `d1` is also copied to the matching LARs. This dirty bit becomes 0 when the contents of `d1` are written back into memory.

In both the scalar and vector cases discussed, the destination data LAR is one of the sources. It is also possible that the destination is not one of the sources and it is never loaded before with a load. As a result the address pointers and word size fields will be empty. It would not be possible to complete a Scalar or Vector ALU instruction. To overcome this shortcoming, the data portion of all the data LARS are initialized to zero and the base pointer, offset and word size fields are set to some predetermined value. A good choice for the initial value might be an offset of 0 and a native-word-size integer as the default tagging.

5.4. SELECT

The Select instruction is the only instruction that works with the instructions of the instruction LAR. It selects between two instructions based on the value in a data LAR. It has the format as shown in Fig 5.14.

OPCODE	D-LAR	I-LAR1	I-LAR2
5	5	5	5

Fig 5.14 Select instruction format

The D-LAR refers to a data LAR and I-LAR1 and I-LAR2 refer to 2 instruction LARs. If the data LAR referred by D-LAR points to a value ‘0’ then the instructions in instruction LAR I-LAR1 get executed, otherwise the instructions in instruction LAR I-LAR2 get executed. This is explained with an example.

```
          Select d0, i12, i14
If (d0.data = 0)
     Execute i12 (really i31=i12)
Else
     Execute i14 (really i31=i14)
```

Instructions (1024 bits wide)	Address
I0	400
...	...
I12 I@671 … I@640	640
I14 I@831 … I@800	800
...	...
I31	PC

Fig 5.15. Select Instruction

When i12 is selected, from fig 5.15, instructions starting from instruction memory location 640 start getting executed. When i14 is selected, instructions from instruction memory 800 start getting executed.

6. A HARDWARE MODEL OF LAR ARCHITECTURE

The basic LAR model has a data and an instruction memory. There are no L1, L2 or L3 caches. There are two arrays of associative registers – the data LAR and the Instruction LAR. The instruction LARs are loaded with a *Fetch* instruction, equivalent to a pre-fetch. An instruction LAR can hold 32 instructions, each 32 bits wide. Once the instructions are loaded, they start getting executed. One of the instruction LARs acts as the instruction register and its address part acts as the program counter. After executing each instruction, the program counter increments itself by 1.

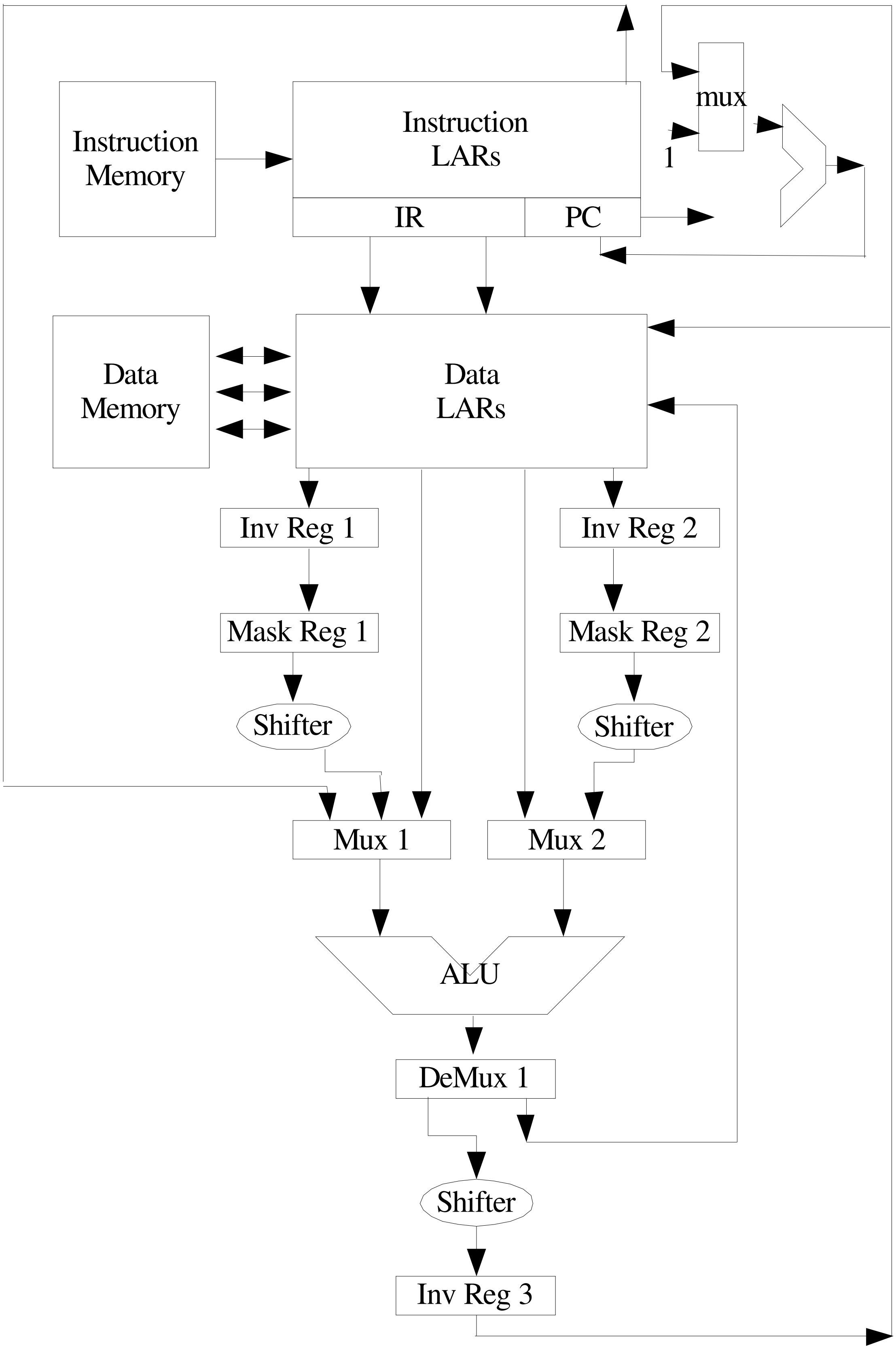
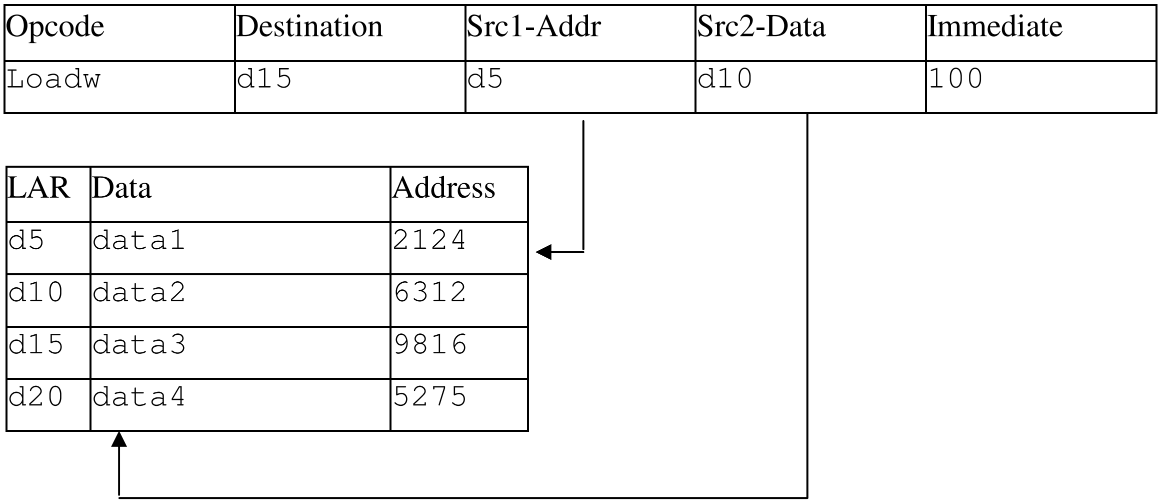


Fig 6.1. Data Path

Instructions are loaded into the instruction LARs with the fetch instruction. The data LARs have to be loaded next. The load points to three data LARs and has an immediate field. Consider an example

```
Loadw d15, d5, d20, 100
```

From the figure below, data1, address 6312 are added with 100. This is the effective address.



Opcode	Destination	Src1-Addr	Src2-Data	Immediate
Loadw	d15	d5	d10	100

LAR	Data	Address
d5	data1	2124
d10	data2	6312
d15	data3	9816
d20	data4	5275

Fig 6.2. Decoding a load instruction

Effective address = data1 + 6312 +100

The effective address calculation is implemented with a 3 input adder that can function either as a two or three input adder, based on whether the instruction is an ALU or load/store instruction. Alternately, there can be two 2-input adderss. The contents of the two source LARs form the 2 inputs of the first adder. The output of the first adder is the sum of the contents of the two source data LARs. The output of this adder is fed into another 2-input adder. The other input of the second adder is the immediate field. The output of the second adder is the effective address.

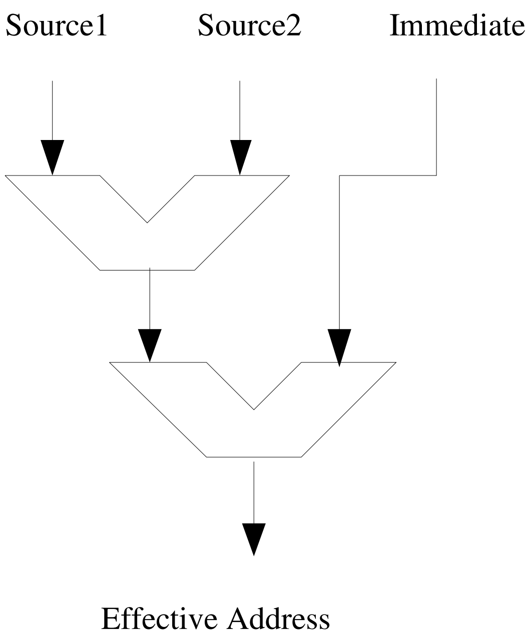


Fig 6.3. A couple of 2 input adders to calculate effective address

This effective address is then compared with the address fields of all the other data LARs. This can be done with a subtract or exclusive-or operation and if there is a hit the zero flag of the ALU gets set. If there is a match, the data values are loaded from the matching LAR. If there are no matches, the contents are loaded from the memory. The word size bits are appropriately set based on the type of load instruction. A load-byte, for example, sets the word size to 8 and a load half-word sets the word size to 16. There are 2 bits in each data LAR that are used to specify the size of data in each data LAR. The bit setting for different word sizes were shown in the table 4.1.

Initially at start-up, all the data fields of the data LARs are initialized to ‘0’. In a case where an ALU operation has to be performed with the destination LAR never loaded before with a load, it is not possible to determine the data type of the result computed. Thus, the data types of the data LARs must be initialized before such use.

A LAR is a vector register that can hold multiple memory words and perform SWAR operations. The data LAR is 256 bits wide and can hold eight memory words. Parallel ALU operations can be performed with a wide ALU. An interesting point to note here is that the ALU unit adjusts its internal partitioning in accordance with the data type of the LAR. As a result, the contents of a LAR can be treated as different size datum in different instructions. For example, if the data size in a LAR is 16, an ALU operation adjusts the width of the ALU inputs to 16.

To perform a Parallel / vector ADD operation with two LARs, the size of the destination is determined. The size of the destination sets the ALU field partitioning. All the contents of the two sources get added and the sum replaces the contents of the destination. This is better explained with an example:

```
Loaduw d1, d0, d0, 80
Loaduw d2, d0, d0, 188
   Addp d1, d1, d2
```

In the above example, `d1` is loaded with unsigned-words and the second LAR `d2`, is also loaded with unsigned words. A vector-add operation is performed between `d1` and `d2` and the result is stored in `d2`. The size of the destination is 32. So the ALU performs an add operation of 8 words (32 bits each) and the sum replaces the contents of the destination. If the two LARs have words of unequal sizes, the destination size again decides the ALU width for the operation.

The scalar operations are slightly different from the vector operations. A scalar operation selects a small portion of the entire LAR line, neglecting the others. It extracts either a byte, half word or a word from the LAR line to operate on. To implement them in hardware a set of invisible and masking registers are required. The required data to be isolated is masked and copied into an invisible register, shifted to right justify, operated on, and then sent back to its original position and then stored. It is better explained with the following example.

Figure 6.4 shows two data LARs. Both have a word size of 8. It is assumed that the first LAR is d1 and the second is d2. d1 has an offset of 2 and d2 has an offset of 5. The scalar add instruction extracts the bits pointed by the instruction and performs addition on the extracted bits.

```
Adds d1, d1, d2
```

Data								B-Ptr	Offset	W-Sz
..	..	..	135	125	**100**	95	85	xxx	2	8
..	..	130	**120**	110	100	90	80	xxx	5	8

Fig 6.4. LARs d1 and d2 before the scalar add operation

The data LAR d1 points to 100 and d2 points to 120. A mask register allows only the word pointed by that LAR to pass through. The contents are then shifted for right justification. The sequence of operations for d1 is shown in the figure 6.5. The offset is field retains the data position information in the LAR line.

..	..	..	135	125	**100**	95	85

&

0	0	0	0	0	**-1**	0	0

=

0	0	0	0	0	**100**	0	0

>>2

0	0	0	0	0	0	0	**100**

Fig 6.5. Scalar Add: Extracting the effective data pointed by d1

The word size of d1 is a byte – so the last 8 binary digits of 100 are taken into consideration for the ALU operation.

The sequence of operations for LAR d2 is shown in fig 6.6.

701	702	130	**120**	110	100	90	80

&

0	0	0	**-1**	0	0	0	0

=

0	0	0	**120**	0	0	0	0

>>5

0	0	0	0	0	0	0	**120**

Fig 6.6. Scalar Add: Extracting effective data pointed by d2

$$(100)_{10} = (0110\ 0100)_2$$

$$(120)_{10} = (0111\ 1000)_2$$

$$+ \qquad (220)_{10} = (1101\ 1100)_2$$

On addition, the 100 and 120 alone get added resulting in 220.. The size of d1 is 8. So the result is also a 8 bit value, 220. The result is also stored in an invisible register. It is then shifted to its original position by shifting it by the number of times specified in the offset as shown in fig 6.7. The sum replaces the data pointed at the destination LAR.

0	0	0	0	0	0	0	**220**

2<<

0	0	0	0	0	**220**	0	0

..	..	..	125	135	**220**	95	85

Fig 6.7. Putting back the sum into its appropriate place

7. RESULTS

Given the large scale of the differences between the proposed LAR architecture and conventional designs, it was not possible to create a highly-accurate simulation from which timing information could be obtained. However, a functional simulator was constructed using C and PCCTS. This simulator accepts assembly language code, translates it into bit patterns, and simulates the execution of the code at the instruction level. Complete source code for the simulator is freely available from the author.

Although the concepts and preliminary design of LAR architecture itself are considered to be the primary contributions of this thesis, qualitative evaluation of the resulting architecture also has been done.

First, it is clear that the proposed LAR mechanism has all the same benefits claimed for CRegs, since ignoring the parallel operations reduces LAR architecture to a mode of operation that is indistinguishable from CRegs. It is unfortunate that the CRegs simulators constructed years ago were not available to be modified to become a LARs simulator. In any case, the CRegs benefits are primarily:

1. The ability to place ambiguously aliased objects in registers, allowing predictable and efficient compiler management of all data references
2. The ability to replace most store operations with implicit, lazy, stores of values that are marked as dirty
3. Efficient handling of instruction blocks rather than individual instructions
4. The ability to explicitly prefetch instruction blocks, which eliminates the need for brach prediction logic while simultaneously allowing the compiler to manage instruction fetch (for example, hoisting instruction fetches out of loops)

However, the original CRegs concepts were somewhat coupled to the architectural ideas and constraints of the late 1980s, and these issues make CRegs awkward to implement in a modern processor and less efficient than they would have been. As a modern adaptation of CRegs, LARs provide solutions to problems that were not important when CRegs were invented. Thus, in addition to the CRegs benefits, LARs provides:

1. A cleaner, more appropriate, implementation directly using the wide memory paths which modern systems require for good performance; data CRegs were only one word wide
2. SWAR parallelism, with the ability to handle greatly increased parallelism widths if appropriate
3. Type-tagging that allows objects of various sizes and types to be directly manipulated with a simple and very regular instruction set
4. Scalar operations that, using type-tagging and addresses within LARS (offsets), allow field extraction and insertion within a LAR operation on any type of data; CRegs required different instructions for operating on different data types and required other CRegs to be used as temporaries (e.g., a byte operation could not be done in-place without affecting all bytes in a word)
5. The ability to fetch multiple instruction blocks with a single fetch instruction

8. CONCLUSION AND FUTURE WORK

In this thesis, a new concept called LAR was introduced and a simple instruction set, suitable for creation of a first prototype, was proposed. An instruction-level functional simulator was built to demonstrate LAR processor operation.

LARs offer a very different way to efficiently handle memory access, both for data and instructions, in a modern processor. Instead of adding complexity and layers to the memory and cache structures, it handles memory traffic through a set of typed, associative registers that can replace multiple levels of caches, vector registers, and scalar registers. It also is compiler-friendly and deals with aliases efficiently.

LARs offer greater parallelism and hence better throughput for parallel operations when compared to the MMX and XMM registers. Unlike the SSE-1 or SSE-2 extensions, it performs flexible scalar operations. It also offers atomicity in handling different data types and sizes. The LARs instruction set proposed here is deliberately kept simple; it does not take full advantage of lessons learned from these and other SWAR models, but it easily can be extended to do so.

Future work should be directed toward designing a specific hardware organization, building a cycle-accurate simulator, and extending the instruction set to be more complete. The type system also should be extended to support floating point data, providing serial and parallel operations on both 32-bit and 64-bit floats. Only after these steps have been taken can quantitative comparisons between LARs and other memory access architectures be made.

APPENDIX A: LINE ASSOCIATIVE REGISTERS SIMULATOR

The following files are for the LAR based simulator developed using PCCTS – Purdue Compiler Construction Tool Set. The simulator developed is a functional simulation of the LAR concept. The source code is available at the “swdev.ece.engr.uky.edu” server.

A.1. Source listing of “simple.g”

```
#header     <<#include "simple.h">>

#lexclass START
#token "[,\ \t\r]+"      << zzskip(); >>

#token "\n"              << ++zzline; >>

#token "/\*"             <<    zzreplstr("");
                               zzmode(LEXCOM);
                               zzmore();
                         >>

#token "\-\-"            <<    zzreplstr("");
                               zzmode(LEXCOM2);
                               zzmore();
                         >>

#lexclass LEXCOM
#token "\n"              <<    ++zzline;
                               zzreplstr("");
                               zzmore();
                         >>

#token "\*/"             <<    zzmode(START);
                               zzskip();
                         >>

#token "~[]"             <<    zzreplstr("");
                               zzmore();
                         >>

#lexclass LEXCOM2
#token "\n"              <<    zzmode(START);
                               zzskip();
                         >>

#token "~[]"             <<    zzreplstr("");
                               zzmore();
                         >>

#lexclass START

input:

(sep |("loadub"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}imm{sep}
<<
      immed=atoi($14.num);dst=atoi($5.num);
      srca=atoi($8.num);srcd=atoi($11.num);
      d[dst].ws=8;d[dst].s=0;//if dst.s=0 ->>unsigned number
      calc(dst,srca,srcd,immed);
      d[dst].edu8 = dm[efa];
      assoc(dst,basep);
      efd = d[dst].du8[offset];
      d[dst].ed = efd;
      dirty1[dst]=1;
      printstuff(dst);
>>

)|("loadb"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}imm{sep}

<<
      immed=atoi($14.num);dst=atoi($5.num);
      srca=atoi($8.num);srcd=atoi($11.num);
      d[dst].ws=8;d[dst].s=1;//if dst.s=1 ->>signed number
      calc(dst,srca,srcd,immed);
      d[dst].ed8 = dm[efa];
      assoc(dst,basep);
      efd = d[dst].d8[offset];
      d[dst].ed = efd;
      dirty1[dst]=1;
      printstuff(dst);

>>

)|("loaduh"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}imm{sep}

<<
      immed=atoi($14.num);dst=atoi($5.num);
      srca=atoi($8.num);srcd=atoi($11.num);
      d[dst].ws=16;d[dst].s=0;//if dst.s=0 ->>unsigned number
      calc(dst,srca,srcd,immed);
      d[dst].edu16 = dm[efa];
      assoc(dst,basep);
      efd = d[dst].du16[offset];
      d[dst].ed = efd;
      dirty1[dst]=1;
      printstuff(dst);
>>

)|("loadh"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}imm{sep}

<<
      immed=atoi($14.num);dst=atoi($5.num);
      srca=atoi($8.num);srcd=atoi($11.num);
      d[dst].ws=16;d[dst].s=1;//if dst.s=1 ->>signed number
      calc(dst,srca,srcd,immed);
      d[dst].ed16 = dm[efa];
      assoc(dst,basep);
      efd = d[dst].d16[offset];
      d[dst].ed = efd;
      dirty1[dst]=1;
      printstuff(dst);
>>

)|("loaduw"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}imm{sep}
```

```
<<
      immed=atoi($14.num);dst=atoi($5.num);
      srca=atoi($8.num);srcd=atoi($11.num);
      d[dst].ws=32;d[dst].s=0; //d[dst].s=0 ->> unsigned number
      calc(dst,srca,srcd,immed);
      d[dst].edu32 = dm[efa];
      assoc(dst,basep);
      efd = d[dst].du32[offset];
      d[dst].ed = efd;
      dirty1[dst]=1;
      printstuff(dst);
>>

)|("loadw"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}imm{sep}

<<
      immed=atoi($14.num);dst=atoi($5.num);
      srca=atoi($8.num);srcd=atoi($11.num);
      d[dst].ws=32;d[dst].s=1;
      calc(dst,srca,srcd,immed);
      d[dst].ed32 = dm[efa];
      assoc(dst,basep);
      efd = d[dst].d32[offset];
      d[dst].ed = efd;
      dirty1[dst]=1;
      printstuff(dst);
>>

)|("loadud"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}imm{sep}

<<
      immed=atoi($14.num);dst=atoi($5.num);
      srca=atoi($8.num);srcd=atoi($11.num);
      d[dst].ws=64;d[dst].s=0; //d[dst].s ->>0 unsigned number
      calc(dst,srca,srcd,immed);
      d[dst].ed64 = dm[efa];
      assoc(dst,basep);
      efd = d[dst].du32[offset];
      d[dst].ed = efd;
      dirty1[dst]=1;
      printstuff(dst);
>>


)|("loadd"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}imm{sep}

<<
      immed=atoi($14.num);dst=atoi($5.num);
      srca=atoi($8.num);srcd=atoi($11.num);
      d[dst].ws=64;d[dst].s=1; //d[dst].s ->>1 signed number
      calc(dst,srca,srcd,immed);
      d[dst].ed64 = dm[efa];
      assoc(dst,basep);
      efd = d[dst].du32[offset];
      d[dst].ed = efd;
      dirty1[dst]=1;
      printstuff(dst);
>>

)|("storeub"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}imm{sep}
```

```
<<
      immed=atoi($14.num);dst=atoi($5.num);
      srca=atoi($8.num);srcd=atoi($11.num);
      d[dst].ws=8;d[dst].s=0;//if dst.s=0 ->>unsigned number
      calc(dst,srca,srcd,immed);
      upgrade(dst);
      printstuff(dst);
>>

)|("storeb"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}imm{sep}

<<
      immed=atoi($14.num);dst=atoi($5.num);
      srca=atoi($8.num);srcd=atoi($11.num);
      d[dst].ws=8;d[dst].s=1;//if dst.s=0 ->>unsigned number
      calc(dst,srca,srcd,immed);
      upgrade(dst);
      printstuff(dst);
>>

)|("storeuh"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}imm{sep}

<<
      immed=atoi($14.num);dst=atoi($5.num);
      srca=atoi($8.num);srcd=atoi($11.num);
      d[dst].ws=16;d[dst].s=0;//if dst.s=0 ->>unsigned number
      calc(dst,srca,srcd,immed);
      upgrade(dst);
      printstuff(dst);
>>

)|("storeh"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}imm{sep}

<<
      immed=atoi($14.num);dst=atoi($5.num);
      srca=atoi($8.num);srcd=atoi($11.num);
      d[dst].ws=16;d[dst].s=1;//if dst.s=0 ->>unsigned number
      calc(dst,srca,srcd,immed);
      upgrade(dst);
      printstuff(dst);
>>

)|("storeuw"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}imm{sep}

<<
      immed=atoi($14.num);dst=atoi($5.num);
      srca=atoi($8.num);srcd=atoi($11.num);
      d[dst].ws=32;d[dst].s=1;//if dst.s=0 ->>unsigned number
      calc(dst,srca,srcd,immed);
      upgrade(dst);
      printstuff(dst);
>>

)|("storew"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}imm{sep}

<<
      immed=atoi($14.num);dst=atoi($5.num);
      srca=atoi($8.num);srcd=atoi($11.num);
      d[dst].ws=32;d[dst].s=0;//if dst.s=0 ->>unsigned number
      calc(dst,srca,srcd,immed);
      upgrade(dst);
      printstuff(dst);
>>

)|("storeud"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}imm{sep}

<<
      immed=atoi($14.num);dst=atoi($5.num);
      srca=atoi($8.num);srcd=atoi($11.num);
      d[dst].ws=64;d[dst].s=0;//if dst.s=0 ->>unsigned number
      calc(dst,srca,srcd,immed);
      upgrade(dst);
      printstuff(dst);
>>

)|("stored"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}imm{sep}

<<
      immed=atoi($14.num);dst=atoi($5.num);
      srca=atoi($8.num);srcd=atoi($11.num);
      d[dst].ws=64;d[dst].s=1;//if dst.s=0 ->>unsigned number
      calc(dst,srca,srcd,immed);
      upgrade(dst);
      printstuff(dst);
>>

)|("addp"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}

<<
      dst=atoi($5.num);src1=atoi($8.num);
      src2=atoi($11.num);
      operation=1;siplu=2;
      ALU(dst,src1,src2,operation,siplu);
      upgrade(dst);
      d[dst].dty[dst]=0;
>>

)|("adds"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}

<<
      dst=atoi($5.num);src1=atoi($8.num);
      src2=atoi($11.num);
      operation=1;siplu=1;
      ALU(dst,src1,src2,operation,siplu);
      upgrade(dst);
        d[dst].dty[dst]=0;
>>

)|("subp"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}

<<
      dst=atoi($5.num);src1=atoi($8.num);
      src2=atoi($11.num);
      operation=2;siplu=2;
      ALU(dst,src1,src2,operation,siplu);
      upgrade(dst);
      d[dst].dty[dst]=0;
>>

)|("subs"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}
```

```
<<
      dst=atoi($5.num);src1=atoi($8.num);
      src2=atoi($11.num);
      operation=2;siplu=1;
      ALU(dst,src1,src2,operation,siplu);
      upgrade(dst);
      d[dst].dty[dst]=0;
>>

)|("mulp"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}

<<
      dst=atoi($5.num);src1=atoi($8.num);
      src2=atoi($11.num);
      operation=3;siplu=2;
      ALU(dst,src1,src2,operation,siplu);
      upgrade(dst);
      d[dst].dty[dst]=0;
>>

)|("muls"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}

<<
      dst=atoi($5.num);src1=atoi($8.num);
      src2=atoi($11.num);
      operation=3;siplu=1;
      ALU(dst,src1,src2,operation,siplu);
      upgrade(dst);
      d[dst].dty[dst]=0;
>>

)|("andp"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}

<<
      dst=atoi($5.num);src1=atoi($8.num);
      src2=atoi($11.num);
      operation=4;siplu=2;
      ALU(dst,src1,src2,operation,siplu);
      upgrade(dst);
      d[dst].dty[dst]=0;
>>

)|("ands"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}

<<
      dst=atoi($5.num);src1=atoi($8.num);
      src2=atoi($11.num);
      operation=4;siplu=1;
      ALU(dst,src1,src2,operation,siplu);
      upgrade(dst);
      d[dst].dty[dst]=0;
>>

)|("orp"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}

<<
      dst=atoi($5.num);src1=atoi($8.num);
      src2=atoi($11.num);
      operation=5;siplu=2;
      ALU(dst,src1,src2,operation,siplu);
      upgrade(dst);
      d[dst].dty[dst]=0;
>>

)|("ors"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}

<<
      dst=atoi($5.num);src1=atoi($8.num);
      src2=atoi($11.num);
      operation=5;siplu=1;
      ALU(dst,src1,src2,operation,siplu);
      upgrade(dst);
      d[dst].dty[dst]=0;
>>

)|("exorp"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}

<<
      dst=atoi($5.num);src1=atoi($8.num);
      src2=atoi($11.num);
      operation=6;siplu=2;
      ALU(dst,src1,src2,operation,siplu);
      upgrade(dst);
      d[dst].dty[dst]=0;
>>

)|("exors"{sep}"d"dcr{sep}"d"dcr{sep}"d"dcr{sep}

<<
      dst=atoi($5.num);src1=atoi($8.num);
      src2=atoi($11.num);
      operation=6;siplu=1;
      ALU(dst,src1,src2,operation,siplu);
      upgrade(dst);
      d[dst].dty[dst]=0;
>>

)|("fetch"{sep}"i"dcr{sep}"d"dcr{sep}"i"dcr{sep}imm{sep}

<<
      immed=atoi($14.num);dst=atoi($5.num);
      srca=atoi($8.num);srcd=atoi($11.num);

      icroutlook();

      int i,j;
      top2=((immed & 3072)\>>10);
      bot10=(immed & 1023);
      icr[dst].addr = ((d[srca].ed32 + icr[srcd].addr + bot10)/8 )*8;
      effadd=icr[dst].addr;
      temp2=icr[dst].addr;

      if (top2==0)  //load 1 ICREG
            {
            icrop(dst);
            }
      else
      if (top2==1)
            {
            icrop(dst);
            icrop(dst+1);
            }
      else
      if (top2==2)
                {
            icrop(dst);
            icrop(dst+1);
            icrop(dst+2);
                }
      else
        if (top2==3)
                {
            icrop(dst);
            icrop(dst+1);
            icrop(dst+2);
            icrop(dst+3);
                }
>>

)|("select"{sep}"d"dcr{sep}"i"dcr{sep}"i"dcr{sep}

<<
      dst=atoi($5.num);src1=atoi($8.num);
      src2=atoi($11.num);
      icroutlook();int i;
      if (d[dst].edu64 == 0)
            {
            printf("i%d ->\t",src1);
            for (i=0;i<8;i++)
                  printf("%d\t",icr[src1].inst[i]);
            printf("%d",icr[src1].addr);
            printf("\n");
            }
      else
            {
                printf("i%d ->\t",src2);
                for (i=0;i<8;i++)
                        printf("%d\t",icr[src2].inst[i]);
                printf("%d",icr[src2].addr);
                printf("\n");
            }
>>
))+
;

dcr:
      "[0-9]" | "[12][0-9]" | "30" | "31"
;

imm:
      ("[0-9]"| "[12][0-9]" | "30" | "31"| "32" | "33" | "34" | "35" | "36" |
"37" | "38" | "39" |"4[0-9]" |"[5-9][0-9]" |"[1-9][0-9][0-9]" | "[1-3][0-9][0-
9][0-9]" | "4000" | "400[1-9]"|"40[1-9][0-5]")
;

sep:
      ("\n" | "," | "\ ")+
;
```

A.2. Source listing of “simple.h”

```
/*    simple.h

      general header file....
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#ifndef     SIMPLE_H
#define     SIMPLE_H
#define mem 99999

void error(char *s);
void warn(char *s);
void syntaxerror(char *s);
void icroutlook();
void calc(int p,int q,int r,int s);
void simplify(int p,int q,int s,int t,int y);
void assoc(int p,int b);
void upgrade(int p);
void ALU(int p,int q,int r,int s,int t);
void vectoropnorm(int p,int q,int r,int s);
void vectoroplong(int p,int q,int r,int s);
void scalarop(int p,int q,int r,int s);
long int todo(long int a,long int b,long int c);
void getsize(int t);
void printstuff(int z);
void dirtymech(int y);
void icrop(int j);

struct dlar {
            int s,ed,ea,bp,off,ws,dty[32];
            union {
                  unsigned char edu8;
                  char ed8;
                  unsigned short edu16;
                  short ed16;
                  unsigned int edu32;
                  int ed32;
                  unsigned long long edu64;// <<-BASE
                  long long ed64;
                  };
            union {
                  unsigned char du8[32];
                  char d8[32];
                  unsigned short du16[16];
                  short d16[16];
                  unsigned int du32[8];
                  int d32[8];
                  unsigned long long du64[4];// <- BASE
                  long long d64[4];
                  };
            }d[32];

struct ilar {
            int effec,inst[8],addr,dirtyicr[32],officr;
            }icr[32];

int efa,efd,basep,offset,size,size1,size2,temp3,temp1,temp2;
int dst,srcd,srca,immed;
int dm[mem];
int im[mem];
int dirty1[32],dirty2[32];
int src1,src2;
int limit,incr;
long int temdoub[4];

int operation,siplu;

int top2,bot10,effadd;

/*    PCCTS stuff...
*/
typedef union {
      char  *text;            /* lexeme text */
      int   num;        /* number value */
} Attrib;

#define zzdef0(a) { /* nothing */ }
#define zzd_attr(a)     { /* nothing */ }

extern void
zzcr_attr(Attrib *a, int tok, char *s);

#endif
```

A.3. Source listing of “main.c”

```
/*    main.c

      Main program...
      process the command line, invoke the parser, etc.

      Fall 2002 by Hank Dietz
*/


#include "stdpccts.h"
#include "simple.h"

main(void)
{
      int i,j=0;
      for (i=0;i<100000;i++)
            {
            dm[i]=j;
            im[i]=j;
            j=j+1;
            }

      for (i=0;i<32;i++)
            {
            dirty1[i]=0;
            dirty2[i]=0;
            d[i].dty[i]=0;
            }

      d[0].bp=0;d[0].off=0;d[0].ws=8;d[0].ea=0;d[0].edu64=0;
      d[1].bp=0;d[1].off=0;d[1].ws=8;d[1].s=0;
      d[2].bp=0;d[2].off=1;d[2].ws=8;d[2].s=0;
      d[3].bp=0;d[3].off=2;d[3].ws=8;d[3].s=0;
      d[4].bp=0;d[4].off=3;d[4].ws=8;d[4].s=0;

      d[5].bp=0;d[5].off=0;d[5].ws=16;d[5].s=0;
      d[6].bp=0;d[6].off=1;d[6].ws=16;d[6].s=0;
      d[7].bp=0;d[7].off=2;d[7].ws=16;d[7].s=0;
      d[8].bp=0;d[8].off=3;d[8].ws=16;d[8].s=0;

      d[9].bp=0;d[9].off=4;d[9].ws=32;d[9].s=1;
      d[10].bp=0;d[10].off=5;d[10].ws=32;d[10].s=1;
      d[11].bp=0;d[11].off=6;d[11].ws=32;d[11].s=1;
      d[12].bp=0;d[12].off=7;d[12].ws=32;d[12].s=1;

      d[13].bp=0;d[13].off=0;d[13].ws=64;d[13].s=1;
      d[14].bp=0;d[14].off=1;d[14].ws=64;d[14].s=1;
      d[15].bp=0;d[15].off=2;d[15].ws=64;d[15].s=1;
      d[16].bp=0;d[16].off=3;d[16].ws=64;d[16].s=1;

      d[17].bp=0;d[17].off=0;d[17].ws=32;d[17].s=1;
      d[18].bp=0;d[18].off=1;d[18].ws=32;d[18].s=1;
      d[19].bp=0;d[19].off=2;d[19].ws=32;d[19].s=1;
      d[20].bp=0;d[20].off=3;d[20].ws=32;d[20].s=1;

      d[21].bp=0;d[21].off=3;d[21].ws=16;d[21].s=1;
      d[22].bp=0;d[22].off=2;d[22].ws=16;d[22].s=1;
      d[23].bp=0;d[23].off=1;d[23].ws=16;d[23].s=1;
      d[24].bp=0;d[24].off=0;d[24].ws=16;d[24].s=1;

      d[25].bp=0;d[25].off=7;d[25].ws=32;d[25].s=0;
      d[26].bp=0;d[26].off=6;d[26].ws=32;d[26].s=0;
      d[27].bp=0;d[27].off=5;d[27].ws=32;d[27].s=0;
      d[28].bp=0;d[28].off=4;d[28].ws=32;d[28].s=0;

      d[29].bp=0;d[29].off=3;d[29].ws=16;d[29].s=0;
      d[30].bp=0;d[30].off=2;d[30].ws=16;d[30].s=0;
      d[31].bp=0;d[31].off=1;d[31].ws=16;d[31].s=0;

      ANTLR(input(), stdin);
      exit(0);

      d[0].bp=0;d[0].off=0;d[0].ws=8;d[0].ea=0;d[0].edu64=0;

}

void icroutlook()
{
      int i;
      printf("\n");
        for (i=0;i<38;i++)
                printf("--");
        printf("\n\tADDR\tI[0]\tI[1]\tI[2]\tI[3]\tI[4]\tI[5]\tI[6]\tI[7]");
        printf("\n");
        for (i=0;i<38;i++)
                printf("--");
        printf("\n");
}

void calc(int p,int q,int r,int s)
```

```
{
      //p=dst  q=srca  r=srcd  s=immed
      efa = d[q].ea + d[r].ed + s;
      d[p].ea = efa;
      basep = (efa/8)*8;
      d[p].bp = basep;
      if (d[p].ws ==64)
            {
            offset=efa-basep;
            if ((offset==0) || (offset==1))
                  offset=0;
            else if ((offset==2) || (offset==3))
                  offset=1;
            else if ((offset==4) || (offset==5))
                  offset=2;
            else if ((offset==6) || (offset==7))
                  offset=3;
            d[p].off=offset;
            }
      else
      {
      offset = efa - basep;
        d[p].off=offset;
      return;
      }

}

void assoc(int p,int b)
{
//p=dst b=basep
      int i;
      temp1=0;temp3=0;
      for (i=1;i<32;i++)
            {
            if ((d[i].bp == b) && (i !=p))
                  {
                  temp3=i;temp1=1;
                  printf("\nAssociative Load - Matching LAR d%d!",temp3);
                  }
            }
      for (i=0;i<8;i=i+1)
            {
            if (temp1 == 1)
                  {
                  d[p].du32[i]=d[temp3].du32[i];
                        }
            else if (temp1 == 0)
                  {
                  d[p].du32[i]=dm[b];
                  }
            b++;
            }
}

void printstuff(int z)
{
      printf("\n");
      printf("\n                      32 Bytes of DATA in the data LAR
\n");
```

```
      printf("        MSB       ---------------------------------------------
LSB   \n\n");
      int i,j=0;

      if (d[z].ws == 8)
            {
            printf("d%d ->\t",z);
            for (i=31;i>-1;i--)
                  {
                  if (d[z].s ==0)
                        {
                        printf("%d ",d[z].du8[i]);
                        j++;
                        if ((j%4)==0)
                              printf("   ");
                        }
                  else if (d[z].s ==1)
                        {
                        printf("%d ",d[z].d8[i]);
                        j++;
                        if ((j%4)==0)
                              printf("   ");
                        }
                  }
                  printf("\n");
                  printf("\nBase Ptr      Offset            WordSize    Type
      Dirty\n");
                  printf("%d\t\t %d\t\t %d\t\t %d\t\t %d
",d[z].bp,d[z].off,d[z].ws,d[z].s,d[z].dty[z]);
                  printf("\n");
            }
      else

      if (d[z].ws == 16)
                  {
                  printf("d%d ->\t",dst);
                  for (i=15;i>-1;i--)
                        {
                  if (d[z].s ==0)
                        {
                        printf("%d ",d[z].du16[i]);
                        j++;
                        if ((j%2)==0)
                              printf("   ");
                        }
                  else if (d[z].s ==1)
                        {
                        printf("%d ",d[z].d16[i]);
                        j++;
                        if ((j%2)==0)
                              printf("   ");
                        }
                        }
                  printf("\n");
                  printf("\nBase Ptr      Offset            WordSize    Type
      Dirty\n");
                  printf("%d\t\t %d\t\t %d\t\t %d\t\t %d
",d[z].bp,d[z].off,d[z].ws,d[z].s,d[z].dty[z]);
                  printf("\n");
            }
      else
      if (d[z].ws == 32)
                {
                printf("d%d ->\t",dst);
                for (i=7;i>-1;i--)
                        {
                  if (d[z].s ==0)
                        {
                              printf("%d ",d[z].du32[i]);
                              printf("   ");
                        }
                  else if (d[z].s ==1)
                        {

                        printf("%d ",d[z].d32[i]);
                              printf("   ");
                        }
                        }
                  printf("\n");
                  printf("\nBase Ptr      Offset            WordSize    Type
      Dirty\n");
                  printf("%d\t\t %d\t\t %d\t\t %d\t\t %d
",d[z].bp,d[z].off,d[z].ws,d[z].s,d[z].dty[z]);
                        printf("\n");
                }
      else

      if (d[z].ws == 64)
                  {
                  printf("d%d ->\t",dst);
                  for (i=3;i>-1;i--)
                        {
                        if (d[z].s ==0)
                              {
                              printf("%lg ",d[z].du32[i]);
                              j++;
                              if ((j%2)==0)
                                    printf("   ");
                              }
                        else if (d[z].s ==1)
                              {
                              printf("%lg ",d[z].d32[i]);
                              j++;
                        if ((j%2)==0)
                              printf("   ");
                              }
                        }
                  printf("\n");
                  printf("\nBase Ptr      Offset            WordSize    Type
      Dirty\n");
                  printf("%d\t\t %d\t\t %d\t\t %d\t\t %d
",d[z].bp,d[z].off,d[z].ws,d[z].s,d[z].dty[z]);
                  printf("\n");
                }
}

void vectoropnorm(int p,int q,int r,int s)
{
        int i;
        for(i=0;i<8;i++)
                {
```

```
                if (d[p].s==0)
                        {
                        if ((d[q].s==0) && (d[r].s==0))
                                d[p].du32[i] =
todo(d[q].du32[i],d[r].du32[i],s);
                        else if((d[q].s==0) && (d[r].s==1))
                                d[p].du32[i] =
todo(d[q].du32[i],d[r].d32[i],s);
                        else if((d[q].s==1) && (d[r].s==0))
                                d[p].du32[i] =
todo(d[q].d32[i],d[r].du32[i],s);
                        else if ((d[q].s==1) && (d[r].s==1))
                                d[p].du32[i] =
todo(d[q].d32[i],d[r].d32[i],s);
                        }
                else
                if (d[p].s==1)
                        {
                        if ((d[q].s==0) && (d[r].s==0))
                                d[p].d32[i] =
todo(d[q].du32[i],d[r].du32[i],s);
                        else if((d[q].s==0) && (d[r].s==1))
                                d[p].d32[i] =
todo(d[q].du32[i],d[r].d32[i],s);
                        else if((d[q].s==1) && (d[r].s==0))
                                d[p].d32[i] =
todo(d[q].d32[i],d[r].du32[i],s);
                        else if ((d[q].s==1) && (d[r].s==1))
                                d[p].d32[i] = todo(d[q].d32[i],d[r].d32[i],s);
                        }

                }
}

void vectoroplong(int p,int q,int r,int s)
{
        int i;
        for(i=0;i<4;i++)
                {
                if (d[p].s==0)
                        {
                        if ((d[q].s==0) && (d[r].s==0))
                                d[p].du64[i] =
todo(d[q].du64[i],d[r].du64[i],s);
                        else if((d[q].s==0) && (d[r].s==1))
                                d[p].du64[i] =
todo(d[q].du64[i],d[r].d64[i],s);
                        else if((d[q].s==1) && (d[r].s==0))
                                d[p].du64[i] =
todo(d[q].d64[i],d[r].du64[i],s);
                        else if ((d[q].s==1) && (d[r].s==1))
                                d[p].du64[i] =
todo(d[q].d64[i],d[r].d64[i],s);
                        }
                else if (d[p].s==1)
                        {
                        if ((d[q].s==0) && (d[r].s==0))
                                d[p].d64[i] =
todo(d[q].du64[i],d[r].du64[i],s);
                        else if((d[q].s==0) && (d[r].s==1))
                                d[p].d64[i] = (d[q].du64[i],d[r].d64[i],s);
                        else if((d[q].s==1) && (d[r].s==0))
                                d[p].d64[i] = (d[q].d64[i],d[r].du64[i],s);
                        else if ((d[q].s==1) && (d[r].s==1))
                                d[p].d64[i] = (d[q].d64[i],d[r].d64[i],s);
                        }
                }
}

void scalarop(int p,int q,int r,int s)
{
//p=dst  q=src1  r=src2 s=operation t=(if t=1 scalar) (if t=2 vector)
                if (d[p].s==0)
                {
                if ((d[q].s==0) && (d[r].s==0))
                        d[p].du32[d[dst].off] = todo(d[q].edu32,d[r].edu32,s);
                else if((d[q].s==0) && (d[r].s==1))
                        d[p].du32[d[dst].off] = todo(d[q].edu32,d[r].ed32,s);
                else if((d[q].s==1) && (d[r].s==0))
                        d[p].du32[d[dst].off] = todo(d[q].ed32,d[r].edu32,s);
                else if ((d[q].s==1) && (d[r].s==1))
                        d[p].du32[d[dst].off] = todo(d[q].ed32,d[r].ed32,s);
                }
                else
                if (d[p].s==1)
                        {
                        if ((d[q].s==0) && (d[r].s==0))
                                d[p].d32[d[dst].off] =
todo(d[q].edu32,d[r].edu32,s);
                        else if((d[q].s==0) && (d[r].s==1))
                                d[p].d32[d[dst].off] =
todo(d[q].edu32,d[r].ed32,s);
                        else if((d[q].s==1) && (d[r].s==0))
                                d[p].d32[d[dst].off] =
todo(d[q].ed32,d[r].edu32,s);
                        else if ((d[q].s==1) && (d[r].s==1))
                                d[p].d32[d[dst].off] =
todo(d[q].ed32,d[r].ed32,s);
                        }
}

void ALU(int p,int q,int r,int s,int t)
{
//p=dst  q=src1  r=src2  s=operation t=(if t=1 scalar) (if t=2 vector)
      if (t==2)         //Parallel operation
            {
            if ((d[src1].ws==64) && (d[src2].ws==64) && (d[dst].ws==64))
                  vectoroplong(p,q,r,s);
            else if ((d[src1].ws == 64) && (d[src2].ws == 64) || (d[dst].ws ==
64))
                  goto oops;
            else if ((d[src1].ws == 64) || (d[src2].ws == 64) && (d[dst].ws ==
64))
                  goto oops;
            else if ((d[src2].ws == 64) && (d[dst].ws==64) && (d[src1].ws ==
64))
                  goto oops;
            else if ((d[src1].ws == 64) || (d[src2].ws == 64) || (d[dst].ws ==
64))
                        {
            oops: printf("\nDouble words are special cases !\n");
                  printf("This Operation cannot be performed !!\n");
                  goto end;
                  }
            else if ((d[src1].ws != 64) && (d[src2].ws != 64) && (d[dst].ws !=
64))
                  vectoropnorm(p,q,r,s);

            dirtymech(p);
            printstuff(p);
            end:;
            }
      else if (t==1)
            {
            if ((d[src1].ws==64) && (d[src2].ws==64) && (d[dst].ws==64))
                        goto ooh;
                else if ((d[src1].ws == 64) && (d[src2].ws == 64) ||
(d[dst].ws == 64))
                        goto ooh;
                else if ((d[src1].ws == 64) || (d[src2].ws == 64) &&
(d[dst].ws == 64))
                        goto ooh;
                else if ((d[src2].ws == 64) && (d[dst].ws==64) && (d[src1].ws
== 64))
                        goto ooh;
                else if ((d[src1].ws == 64) || (d[src2].ws == 64) ||
(d[dst].ws == 64))
                        {
                ooh:   printf("\nDouble words are special cases !\n");

                        printf("This Operation cannot be performed !!\n");
                        goto theend;
                        }
                else if ((d[src1].ws != 64) && (d[src2].ws != 64) &&
(d[dst].ws != 64))
                        scalarop(p,q,r,s);

            dirtymech(p);
            printstuff(p);
            theend:;
}

void scalarop(int p,int q,int r,int s)
{
//p=dst  q=src1  r=src2  s=operation
        if (d[p].s==0)
            {
                if ((d[q].s==0) && (d[r].s==0))
                  d[p].du32[d[dst].off] = todo(d[q].edu32,d[r].edu32,s);
                else if((d[q].s==0) && (d[r].s==1))
                  d[p].du32[d[dst].off] = todo(d[q].edu32,d[r].ed32,s);
                else if((d[q].s==1) && (d[r].s==0))
                  d[p].du32[d[dst].off] = todo(d[q].ed32,d[r].edu32,s);
                else if ((d[q].s==1) && (d[r].s==1))
                  d[p].du32[d[dst].off] = todo(d[q].ed32,d[r].ed32,s);
                        }
                else
                if (d[p].s==1)
                        {
                        if ((d[q].s==0) && (d[r].s==0))
                        d[p].d32[d[dst].off] = todo(d[q].edu32,d[r].edu32,s);
                        else if((d[q].s==0) && (d[r].s==1))
                        d[p].d32[d[dst].off] = todo(d[q].edu32,d[r].ed32,s);
                        else if((d[q].s==1) && (d[r].s==0))
                        d[p].d32[d[dst].off] = todo(d[q].ed32,d[r].edu32,s);
                        else if ((d[q].s==1) && (d[r].s==1))
                        d[p].d32[d[dst].off] = todo(d[q].ed32,d[r].ed32,s);
                        }

                }

}
long int todo(long int a,long int b,long int c)
{
      if (c==1)
            return (a+b);
      else if (c==2)
            return (a-b);
      else if (c==3)
                return (a*b);
      else if (c==4)
                return (a&b);
      else if (c==5)
                return (a|b);
      else if (c==6)
                return (a^b);
}

void upgrade(int p)
{
      int i,j,k;
      k=d[p].bp;
      for (i=1;i<32;i++)
            {
            if (d[p].bp == d[i].bp)
                  {
                  for (j=0;j<8;j++)
                        d[i].d32[j] = d[p].d32[j];
                  }
            }
      for (j=0;j<8;j++)
            {
            dm[k] = d[p].d32[j];
                k++;
            }
}

void dirtymech(int y)
{
            int i;
            dirty2[y]=1;
                if ((dirty1[y]) && (dirty2[y]) ==1)
                        d[y].dty[y]=1;
            if (d[y].dty[y]==1)
                        {
                        for (i=d[y].bp;i<(d[y].bp+8);i++)
                                {
                                dm[i] = d[y].du64[i];
                                }
                        }
}

void icrop(int j)
{
```

```
            int i;
            printf("i%d ->\t",j);
            printf("%d\t",temp2);
            temp2=temp2+8;
            for (i=0;i<8;i++)
                        {
                        icr[j].inst[i]=im[effadd];
                        printf("%d\t",icr[j].inst[i]);
                        effadd++;
                        }
            printf("\n\n");
}

void error(char *s)
{
      printf("error(%s)\n", s);
      exit(0);
}

void warn(char *s)
{
      printf("warn(%s)\n", s);
      exit(0);
}

void
syntaxerror(char *s)
{
      printf("syntaxerror(%s) in line %d\n",
            s,
            zzline);
}

void
zzcr_attr(Attrib *a, int tok, char *s)
{
      a->text = malloc(strlen(s) + 1);
      if (a->text == 0) {
            syntaxerror("out of memory while reading input");
            a->text = "";
      } else {
            strcpy(a->text, s);
      }
}
```

REFERENCE

1. Guy Lewis Steele, Jr. , Gerald Jay Sussman, "The dream of a lifetime: A lazy variable extent mechanism", Proceedings of the 1980 ACM conference on LISP and functional programming, pp.163-172, August 1980.

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

3. Ben Heggy, Mary Lou Soffa, "Architectural support for register allocation in the presence of aliasing", Proc., Supercomputing '90, pp. 730-739, Feb 1990.

4. Peter Dahl, Matthew O'Keefe, "Reducing memory traffic with CRegs", Proceedings of the 27th annual international symposium on Microarchitecture, pp 100-104, Nov 1994.

5. John Stokes, "Understanding CPU caching and Performance," http://arstechnica.com/paedia/c/caching/caching-1.html

6. David. A. Patterson, John L. Henessey, "Computer Organization and Design – The Hardware / Software Interface".

7. David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, Wen-mei W. Hwu, "Dynamic memory disambiguation using the memory conflict buffer", ACM SIGPLAN Notices, Vol 29 No. 11, pp.183-183, Nov, 1994.

8. Matthew A. Postiff, Trevor Mudge, "Smart Register Files for High Performance Microprocessors", Technical Report CSETR -403-99, June 1999.

9. David R. Engebretsen, Peter E. Bergner, Matthew T. O'Keefe, "Register Allocation and Code Scheduling for CRegs using SUIF", The First SUIF Compiler Workshop, Stanford University, January 11-13, 1996.

10. Matthew Postiff, David Greene, Trevor Mudge, "The store-load address table and speculative register promotion", Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture, pp.235-244, December 2000.

11. John Lu, Keith D. Cooper, "Register promotion in C programs", Proceedings ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI-97), pp. 308-319, June 1997.

12. A. V. S. Sastry, Roy D. C. Ju, "A new algorithm for scalar register promotion based on SSA form", Proceedings ACM SIGPLAN '98 Conf. Programming Language Design and Implementation (PLDI), pp. 15-25, May 1998.

13. Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, Peng Tu," Register promotion by sparse partial redundancy elimination of loads and stores", Proceedings ACM SIGPLAN '98 Conf. Programming Language Design and Implementation (PLDI), pp. 26-37, May 1998.

14. IA32 Intel Architecture Software Developers Manual, Volume 1: Basic Architecture.

15. AMD Extensions to the 3DNow! And MMX Instruction Sets Manual.

16. Randall J. Fisher, Henry G. Dietz, "Compiling For SIMD Within A Register**",** Proc. 11th International workshop on Languages and Compilers for Parallel Computing, pp. 290 – 304, 1998.

17. G. J. Myers, B. R. S. Buckingham, "A Hardware implementation of capability based addressing", ACM SIGOPS Operating Systems Review, Volume 14 Issue 4, pp. 13-25, October 1980.

18. R. S. Fabry, "Capability-based addressing", Communications of the ACM, Volume 17 Issue 7, pp. 403 – 412, July 1974.

VITA

This thesis was done by Krishna Melarkode. He was born on the 25th of April, 1980 at Chennai, INDIA. He got his Bachelors of Engineering degree in Electronics and Communication Engineering from the University of Madras, INDIA in 2001. He worked as a Teaching assistant in the Department of Physics, University of Kentucky for a year. He also worked as a Research Assistant for Dr. Hank Dietz in the department of Electrical and Computer Engineering, University of Kentucky.