University of Kentucky

UKnowledge

University of Kentucky Master's Theses

Graduate School

2010

DESIGN AND IMPLEMENTATION OF THE INSTRUCTION SET ARCHITECTURE FOR DATA LARS

Kalyan Ponnala
University of Kentucky, ponnala.kalyan@gmail.com

Right click to open a feedback form in a new tab to let us know how this document benefits you.

ABSTRACT OF THE THESIS

DESIGN AND IMPLEMENTATION OF THE INSTRUCTION SET ARCHITECTURE
FOR DATA LARS

The ideal memory system assumed by most programmers is one which has high capacity, yet allows any word to be accessed instantaneously. To make the hardware approximate this performance, an increasingly complex memory hierarchy, using caches and techniques like automatic prefetch, has evolved. However, as the gap between processor and memory speeds continues to widen, these programmer-visible mechanisms are becoming inadequate.

Part of the recent increase in processor performance has been due to the introduction of programmer/compiler-visible SWAR (SIMD Within A Register) parallel processing on increasingly wide DATA LARs (Line Associative Registers) as a way to both improve data access speed and increase efficiency of SWAR processing. Although the base concept of DATA LARs predates this thesis, this thesis presents the first instruction set architecture specification complete enough to allow construction of a detailed prototype hardware design. This design was implemented and tested using a hardware simulator.

KALYAN PONNALA

(03/09/10)

DESIGN AND IMPLEMENTATION OF THE INSTRUCTION SET ARCHITECTURE
FOR DATA LARS

By

Kalyan Ponnala

Dr. Hank Dietz

Director of Thesis

Dr. Stephen D. Gedney

Director of Graduate Studies

(03/09/2010)

RULES FOR THE USE OF THESES

Unpublished theses submitted for the Master's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with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

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

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

Name   Date

_____

_____

_____

_____

_____

_____

THESIS

Kalyan Ponnala

The Graduate School

University of Kentucky

2010

DESIGN AND IMPLEMENTATION OF THE INSTRUCTION SET ARCHITECTURE
FOR DATA LARS

_____

THESIS
_____

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

By

Kalyan Ponnala

Lexington, Kentucky

Director: Dr. Hank Dietz, Professor of Electrical Engineering

Lexington, Kentucky

2010

DEDICATION

To my extraordinary parents

And

All my friends at the University of Kentucky

ACKNOWLEDGEMENTS

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF FILES

CHAPTER 1

1. INTRODUCTION

The volume of data processed by the processors these days is increasing exponentially whereas the methods to suppress the high memory latency through programmer invisible mechanisms like caches or multithreading are falling short to keep up with these huge complexities. Even the speed of the conventional Von Neumann organization is highly dependent on the technology. This chapter provides a background the problem of the inefficiency of present memory hierarchies in bridging the gap between the high performance processors and the main memory and the motivation to opt for the Line Associative Registers (LARs) design.

1.1 MOTIVATION

The usual memory hierarchy goes from Registers (smaller) to RAMs (bigger). The reason for using different levels of memory from the cost issues of fabricating these memory units; registers being expensive because of the fabrication technology they use and the number of flip-flops they employ for storing a single bit of data. Layers of Cache are introduced to compensate for this gap in a less expensive way at the cost of relatively low bandwidth and high memory latency. The increase of the processor speeds these days has left a larger memory footprint on the whole processor-memory performance system.

As one of the solutions to this problem, parallel processing is considered where similar processing elements work simultaneously on a single big problem. SWAR (Single Instruction Multiple Data (SIMD) Within A Register) introduces a good programming model for data level parallelism by adding the ability to perform SIMD-like operations on fields within a register or datapath. SWAR operations replace a series of memory accesses and field extraction/insertion operations with a single access for a word's worth of fields. CRegs (Cache Registers) provide a hybrid hardware structure for registers with both the properties of caches and registers and have the advantages of both. The idea of extending the concepts of CRegs to support data level parallelism leads to the motivation of this thesis. Line Associative Registers (LARs) are evolved with the properties of

CRegs and SWAR. These registers along with type tagging lead to a whole lot of advantages. The next section gives the background of LARs.

1.2 BACKGROUNG

1.2.1 SWAR

SIMD parallel processing over multiple data fields within each processor registers was intended to speed-up multimedia algorithms by supporting high-level parallel programming models. SWAR is such a hardware model which was implemented using only minor modifications to existing datapaths and function units. The PA-RISC [16][17] and the IA32 MMX [18][19] extensions both operate on the SWAR concept of having wide registers with the ability to operate on short integer operands on fields within 64-bits datapaths, although these mechanisms are add-ons to the existing architecture designs.

AltiVec and SSE have 128-bit data paths. One problem with AltiVec is that data cannot be moved directly between the vector and general-purpose integer registers. Thus, array indices generated in the vector registers must be moved via memory to the integer registers for use in a load or store instruction. 3DNow! [20] is a good first step toward adding floating-point SWAR capabilities to MMX and improving its coverage. There is still room for improvement which is addressed by the Athlon extensions to 3DNow!

The most salient aspect of these architectures is their vector SIMD nature. This has several implications for the design of a programming model including the expression of data parallelism and the execution of multiple control paths. In order to ease adoption of SWAR, many of the implementations provide both parallel and scalar instructions. The oxymoronic scalar SWAR instructions simply act on a single field, generally the one in the lowest bit field [21].

1.2.2 CREGS

The CRegs (C-Reg, Cache Registers) [9][10] are designed to reduce the number of memory accesses required by spotting and updating the ambiguous aliasing automatically and thereby letting these values stay in registers much longer. This idea would be really

important for SWAR-like processing where a single fetch cycle would mean fetching a whole line of data.

A CReg is a memory module which replaces the registers in a computer architecture design. It combines the structures of cache and registers as shown in the figure below. These special registers allow the variable values to stay in registers a while longer by controlling the way ambiguously aliased values are detected as well as updated.

Name	Datum	Object Address	status
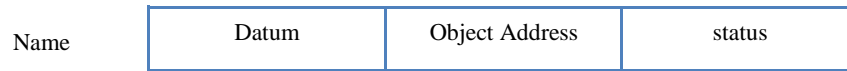
Figure 1: The hardware structure of CRegs

Ambiguous aliasing is a concept that occurs when the processor is dealing with arrays or pointers. Considering an array "a" with "n" elements,

```
function (a[i], a[j])
    {
        a[i] = a[j] + a[i];
        a[j] = a[j] ^ a[i];
    }
```

Listing 1: compile-time Unresolvable Aliasing problem

In the above C program, the compiler does not know if "i" and "j' have the same values till runtime, so the compiler takes these values to be ambiguously aliased and assigns two separate registers to a[i] and a a[j]. The compiler can't tell if a change of the value in the register holding a[i] should also change the value in the register holding a[j]. Thus, the compiler is forced to generate code that flushes the new value of a[i] to memory and then re-loads the value of a[j] from memory. The compiler is forced not only to allocate a separate register for each, but also to flush and reload *all potentially aliased* registers each time the value in *any one* of those registers is modified. This problem is common in code working on arrays and/or pointers.

CRegs handle this problem efficiently with less memory access traffic. Before writing any values to these registers the processor checks for any register with same address field. In case it finds one, the hardware associatively updates both, the current register

3

which is going to be written with this new value and the register with same address field. By doing so the hardware maintains coherence of these entries in a way similar to the implementing the Associativity of a cache. The difference between the CReg hardware and the Cache is that, unlike a cache, CRegs avoid making memory references on an aliased `Load` operation by using duplicating entries in the CReg array.

The important point to note here is that a cache can handle the ambiguously aliased values since it has the address field of the variables on it, but the problem with it is that it cannot be controlled by a compiler.

Despite the idea being twenty years old, the strangeness of requiring a new instruction set design has prevented CRegs from being widely applied. The only commercial implementation to date is the IA64 Advanced Load mechanism [11], which does not achieve the full benefit because it uses its CReg-like mechanism as a filter for memory references rather than as a replacement for conventional registers and cache. The idea of STM (Short Term Memory cell) [12] and Rack [13] is similar to CRegs but without the Associativity of CRegs. Trace cache and loop recognition systems employed by Intel in their P4 and i7 [14] [15] also use the CReg mechanisms but they just use it as a storage buffer rather than replacing the register and cache with these cache-registers in which case they could have improved the efficiency of the system by the help of the compiler.

1.2.3 CACHE

The several levels of Cache that were mentioned before are introduced to bridge the semantic gap between the processor and main memory speeds and act like a buffer which maintain quickly accessible copies of the data and instructions which are most likely to be needed by the processor. As long as the cache holds right data the processor effectively sees the cache access time with the large memory address. The Hardware cell of a cache is given below:

Index (Line Address)	Data	Tag (Line Address)	Status

Figure 2: The hardware structure of cache

Since 1969 when the first cache based computer was developed by IBM as IBM 360/85 [1], having a cache in the processor-memory system design is proving to be inefficient. The following points would make this clear.

➢ Having a cache only helps when there is a cache hit. It's the only case where the processor sees the actual intended speed called the cache access time rather that the memory access time. Misses are more costly here when compared to a cache-less memory hierarchy.

➢ A majority of data elements which are reference in a program are referenced so infrequently that other cache traffic is certain to evict these elements from cache before they are referenced again. In such cases, there is no benefit in placing the item in cache, but there is the excess overhead involved in evicting some other item out of cache to make room for this useless cache entry. This would prove more inappropriate in cases where a cache line is larger than a processor word in which case it has an additional penalty of loading an entire line from memory into cache. [2] [25]

➢ Since processors these days use SIMD registers, it is not wise to have same data in both cache and the long register lines. [Intel's MMX, SSE, Itanium, GPUs etc].

➢ Caches nowadays take up a lot of space on chip which could be used for better purposes [3].

Re-evaluation of many computer design concepts like compiler methods for optimization and parallelization [4][5], architecture concepts of RISC and CISC [6][7][8], should be opted and inspired since the traditional memory hierarchy which includes the cache designs is falling short of the required expectations of bridging the semantic gap between the processor and main memory.

1.3 THESIS ORGANIZATION

The thesis documentation is divided into 6 chapters. The first chapter presented the introduction along with motivation and background of LARs. The second chapter describes the idea of LARs along with its hardware structure. The third chapter describes the instruction set architecture of DATA LARs. The fourth chapter gives a detailed description of the DATA LARs hardware simulator of the straw-man model along with

the discussions about its tradeoffs. The fifth chapter gives the results where comparison of DATA LARs design with other architectures is explained along with some simple program examples run on the simulator. The sixth chapter concludes and gives the future work for this topic.

CHAPTER 2

2. LARS

Line Associative Registers (LARs) concept is derived mainly from Associativity of CREGs and SWAR operations on wide lines of data. Compared to a normal CPU register, LAR can also hold the address of the starting object similar to a CREG and are wide SIMD registers similar to SWAR. LARs are assigned to both instructions and data when the processor's architecture design is considered. An Instruction LAR has the address of the starting object and is wide as discussed above whereas the DATA LARs are type tagged in addition to having the address field and the width. The address field could be used as a typed pointer to point to any object within the given LAR line. This makes it much easier for scalar operations on objects within a LAR to have great flexibility to access any field, not just the one in the lowest bits. Registers like MMX and SSE extensions [18][19] which utilize the SWAR concepts, are just wide registers which are not that convenient to be used for scalar operations as LARs could be used as discussed above.

Since the CPU registers are compiler friendly, LARs could be handled efficiently when compared to a Cache (because compiler doesn't know the address of the cache line). Moreover, the ambiguous aliasing flush/reload problem is handled by updating aliased objects in registers as in CREGs with the difference that this updating is extended to work with wider lines rather than single objects in registers.

This helps improve the memory bandwidth. Type tagging Line Associative Register Lines also bring a lot of advantages to the design. Since data LARs are type tagged at the load time this would decrease the instruction set considerably by removing the type conversion instructions thereby increasing the code density and increasing architectural regularity and simplifying the instruction set. Type conversions occur automatically in this design in the hardware itself according the conditions depending on the data types and signs that would be discussed later. The stores are lazy in this design. This would reduce the memory cycles at hand and above that this makes store instructions to be used for different purpose which in our case would be to do changes to the type information of

the LAR under use.   Intel's iAPx432 [22] had type-tagged main memory when compared to LARs which has type-tagged registers. Type-tagging objects in memory worsened the dependence on memory performance.

Research on the concept of LARs began with the thesis of Krishna Melarkode [23] However, the initial ideas have proven to have many more complex implications than were originally realized. The prototype designed for this thesis is a straw man implementation with the parameters such as line width of the LAR scaled down to fit the standard FPGA. The main aim of the design is to present sufficient details so as to allow the reader to fully appreciate the complexity and logic involved in the LARs concept compared to a conventional processor.

The design implemented here is a 32-bit microprocessor with 6-stage pipeline to achieve higher throughput and shorter clock. There are 8 data LARs 64-bits wide (CPU registers which handle data), although this number could be varied anytime and the number of instruction LARs (CPU registers which handle code) is not fixed either. The actual design was intended to contain 32-data LARs 256-bits.

2.1 DATA TYPES

This chapter introduces data types defined for Line Associative Registers (LARs) architecture.

2.1.1 FUNDAMENTAL DATA TYPES

The fundamental data types are bytes, half-words, words and double-words. A byte is of 8-bits, a half-word is of 2 bytes (16-bits), a word is of 4 bytes (32-bits), and a double-word is of 8 bytes (64-bits) as shown in figure 3. Byte order of each of these data types when referenced as operands in memory is shown in the figure 4. The low byte (bits 0 through 7) of each data type occupies the lowest address in memory and that address is also the address of the operand. This is a "little endian" machine which means that bytes of a word are numbered starting from the least significant byte.
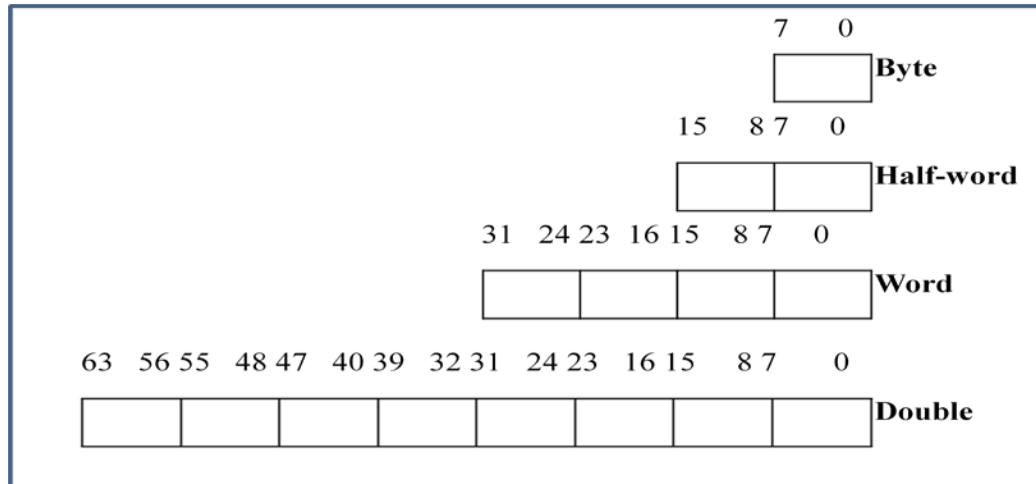
Figure 3: Fundamental Data Types
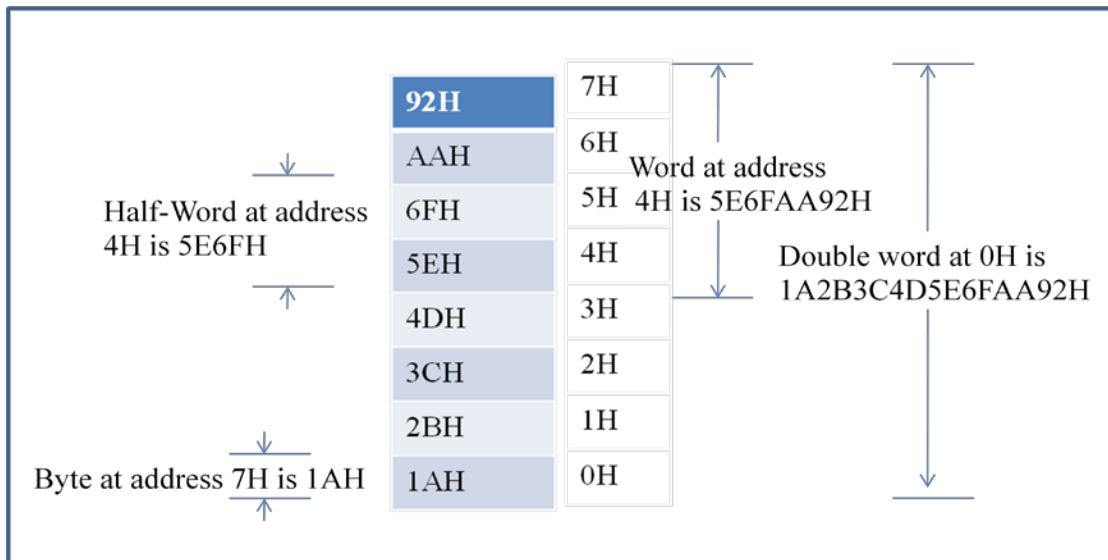
Figure 4: Bytes, Half words, Words and Double Words in memory

2.1.2 PACKED SIMD DATA TYPES

The designed LARs architecture operates on 64-bits of packed SIMD data. The fundamental packed SIMD data types are defined as packed bytes, packed half-words, packed words and packed double-words. At the time of processing, the numeric SIMD operations on the CPU registers interpret these packed data types to contain packed or scalar byte, half-word, word, or double-word integer values.

9
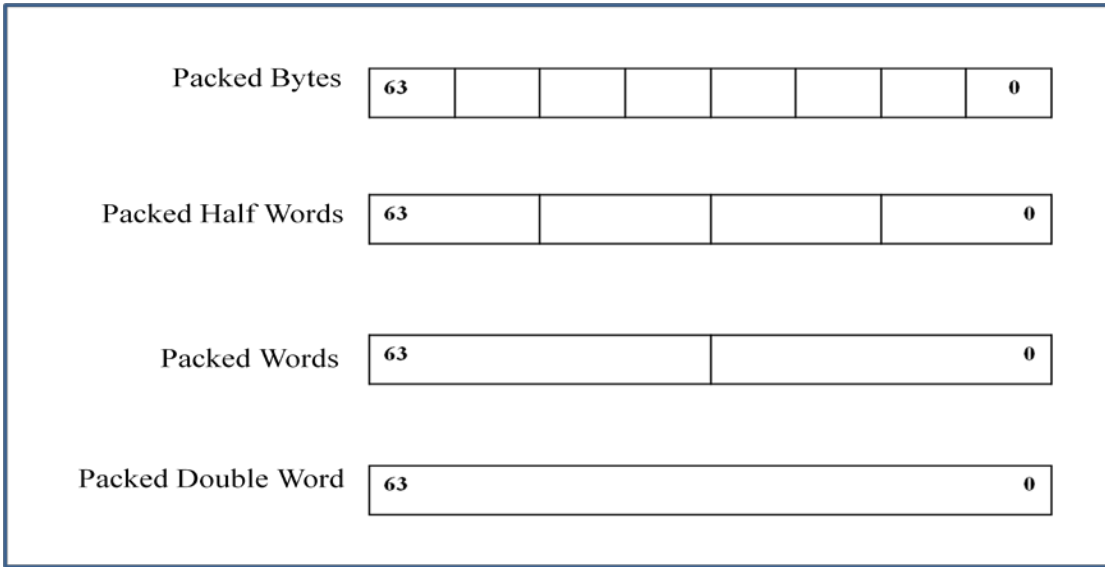
Figure 5: Packed SIMD data types

2.2 SIGNED AND UNSIGNED INTEGERS

Unsigned integers are ordinary binary values ranging from 0 to the maximum positive number that can be encoded in the selected operand size. Signed integers are two's complement binary values that can be used to represent both positive and negative integer values.

Unsigned integers are unsigned binary numbers contained in a byte, half-word, word, and double-word. Their values range from 0 to 255 for an unsigned byte integer, from 0 to 65,535 for an unsigned half-word integer, from 0 to 2^{32} -1 for an unsigned word integer and from 0 to 2^{64}-1 for an unsigned double word integer.

Signed integers are represented in two's complement form throughout this design. The sign bit is located in bit 7 in a byte integer, bit 15 in a half-word integer, bit 31 in a word integer, and bit 63 in a double word integer. Saturation arithmetic uses the following representations to fix the overflow or underflow values to some boundary. For an unsigned, 8-bit byte, the largest and the smallest represent-able values are FFh and 0x00; for a signed byte the largest and the smallest represent-able values are 7Fh and 0x80. This is important for pixel calculations where this would prevent a wrap-around add from

10

causing a black pixel to suddenly turn white while, for example, doing a 3D graphics Gouraud shading loop.

2.3 MEMORY ALIGNMENT

Data structures are always aligned in this design. So words, double words, and quad words are all aligned in memory on natural boundaries. The natural boundaries for words, double words, and quad words are even-numbered addresses, addresses evenly divisible by four, and addresses evenly divisible by eight, respectively. The main advantage with this limitation is that it will improve the performance of programs. This is because the processor would require only one memory cycle to fetch an aligned data compared to two memory cycles required for an unaligned data.

2.4 DATA LARS REGISTER SET

Data LARs are the CPU registers which handle data part of the processor. All the data LARs are assigned the address field, the type field and the dirty field as shown in the figure 6.

Data LAR	Data 64-bit								Address 64-bits		WDSZ 2-bits	TYP 1-bit	Dirty 1-bit
									Tag 61-bits	Word offset 3-bits			
d0	7	6	5	4	3	2	1	0		Bytes	2'b00	1'b0	1'b0
d1	3		2		1		0		Half	Word	2'b01	1'b0	1'b0
d2													
d3	1				0					Word	2'b10	1'b0	1'b0
d4													
d5	0								Double	Word	2'b11	1'b0	1'b0
d6													
d7													

Figure 6: The hardware structure of DATA LARs

There are 8 data LARs represented as "d" followed by "0 – 7" (0,1,2..7) 64-bits wide each which can hold one of the packed SIMD data types (8 packed bytes or 4 packed

half-words or 2 packed words or 1 packed double word). Each of these data LARs from d0 to d7 has an Address field, WDSZ field, TYP field and a Dirty field. The address field (64-bits in our case) is further classified into a TAG (61-bits) and a Word-Offset (3-bits). The whole address field acts as a typed pointer where TAG field points to the starting location of the LAR line and Word-Offset field is used to pick one of the objects within this line of data. So address operations can be scaled by object size much as the C programming language does for pointer arithmetic and moreover, scalar operations on objects within a LAR have full flexibility to access any field, not just the one in the lowest bits.

The WDSZ and TYP field are used to represent the type tagging information of the data that gets loaded from the main memory. When compared architectures with type tagging or object in main memory, LARs type tags the objects or say lines after they get loaded from the main memory. Type tagging the memory makes it expensive since it increases the memory bandwidth and space requirement. Overriding these tags would also be difficult.

WDSZ field is 2-bits wide this specifies if the data loaded into the data LARs is one of the packed data types. The table 1 below shows the different bit field settings for different data types.

TYP field is 1-bit wide and this specifies if the data loaded is Signed or Unsigned. The table 2 below shows the bit field variations for signed and unsigned data.

Table 1: Bit field variations of WDSZ

WDSZ 2-bits	PACKED SIMD DATA TYPE
2'b00	Packed Bytes
2'b01	Packed Half Words
2'b10	Packed Words
2'b11	Packed Double Words

Table 2: Bit Field variation of TYP

TYP 1-bit	SIGN
1'b0	Unsigned Data
1'b1	Signed Data

LARs architecture tries to reduce memory cycles as far as possible. It uses Lazy Store mechanism to store back the data from CPU registers to main memory. It has a Write-Buffer, whose size could be varied as necessary, which stores the dirty information spilled to the main memory. Dirty field (1-bit field) is used to know if the data in a certain data LAR has been changed due to some operation. If the processor tries to load some data into a dirty location, the dirty data is spilled to the Write-Buffer. For the current design, this data is sent to the main memory is a case where the buffer itself is full. In that situation, the whole pipeline is stalled and the buffer spills the excess data to main memory.

There would not be any need for store instructions in this case. So the store instructions are used for type castings, or say, to change the address, WDSZ and TYP fields of the current data LAR so that it could be used for a different purpose without actually loading the data again since the processor already knows if the required address is present in one of the data LARs or not. This would be further explained with an example in the third chapter.

CHAPTER 3

3. INSTRUCTION SET ARCHITECTURE

3.1 DATA TRANSFER INSTRUCTIONS

There are eight different LOAD instructions in this architecture whose opcodes are used for type tagging the DATA LARs. These instructions transfer data from the main memory to the LARs and type tag the registers by filling in the required information about the type and size of the data that is loaded. The difference between these instructions and the other architectures instructions like MIPS is the way they get executed; there would be an associative search of DATA LARs internally to check if the data at the calculated effective address has been already loaded by previous load instructions. This would save the cost of fetching the data (line of data) all the way from the main memory in case of a hit at the data LARs.

Opcode [31:27]	DST [26:22]	SRC1 [21:17]	SRC2 [16:12]	IMMEDIATE [11:0]

Figure 7: Format of LOAD instruction

The DEST LAR (destination DATA LAR), SRC1 (Source DATA LAR 1), and SRC2 (Source DATA LAR 2) are all five bit fields since these fields are aimed to address 32 DATA LARs instead of 8. Hence the architecture utilizes only 3 bits out of 5 to encode the register set and the two most significant bits are left in place and ignored in this prototype. And there is a 12-bit immediate field as shown.

The calculation of the effective address and the whole procedure of a load instruction could be better understood by an example. Let the instruction be

```
LOADUB d1 d2 d3 5
```

The instruction says that the data to be loaded from memory is an unsigned byte. The opcode for LOADUB is 5'b00001, this opcode is sent to the DATA LARs along with the

data from main memory so that the TYP and WDSZ fields of the destination LAR can be filled simultaneously to 1'b0 for unsigned and 2'b0 for byte data respectively.

The TAG and WORDOFFSET fields together constitute the effective address calculated by ALU. The way the effective address is calculated is:

```
Effective addr        = SRC1.Addr + SRC2.Data + immediate
(Destination LAR)
```

Let us assume that the data stored at DATA LARs d2 and d3 are as shown below:

Data LAR	Data 64-bit								Address 64-bits		WDSZ 2-bits	TYP 1-bit	Dirty 1-bit
									Tag 61-bits	Word offset 3-bits			
d1	7	6	5	4	3	2	1	0		Bytes	2'b00	1'b0	1'b0
d2									61'b0	3'b000	2'b10	1'b0	1'b0
d3	16'b0		16'b0		16'b0		{10'b0,6'b1}		{61'b0, 1'b1}	3'b111	2'b01	1'b0	1'b0

Figure 8: DATA LARs before the LOAD instruction

```
Destination DATA LAR   = d2.address + d3.data + 5;
d1's effective address

                       = 0 + {58'b0, 6'b1} + 5;

                       = 000000 + 111111 + 000101;

    Effective Address = 1000100;

              =>Tag = 61'b {57'b0, 4'b1000} = (8)₁₀;

        =>Word Offset = 3'b100 = (4)₁₀;
```

The 64 bit address (TAG+WORDOFFSET) of d2 (source 1) added with 64 bit data of d3 (source 2) and 5 (the 12 bit immediate field) would be the effective address. The obtained 64 bit data from the main memory using this effective address, along with the opcode, will be sent to the DATA LAR in the last stage. Even the effective address would be

utilized here in the form of TAG and WORDOFFSET for the destination DATA LAR. The word offset is 3 bits wide since the DATA LAR is 64 bits wide and is byte accessible (eight bytes accessed by three offset bits).

After the Load the data LARs will look like this

Data LAR	Data 64-bit				Address 64-bits		WDSZ 2-bits	TYP 1-bit	Dirty 1-bit
					Tag 61-bits	Word offset 3-bits			
d1	Data from data memory from the calculated effective address location				61'b1000	3'b100	2'b00	1'b0	1'b0
d2					61'b0	3'b000	2'b10	1'b0	1'b0
d3	16'b0	16'b0	16'b0	{10'b0,6'b1}	{61'b0, 1'b1}	3'b111	2'b01	1'b0	1'b0

Figure 9: DATA LARs after the LOAD instruction

As explained before, the LARs architecture would not go to the main memory blindly. It would stall the whole pipeline for one clock cycle and perform an associative search for every LOAD instruction to check if the calculated effective address is already present in one of the DATA LARs. In the above case, if 64'b1000100 address is already present in one of the LARs then the fetch cycle is canceled and a copy of the already present data would be placed at the DATA LAR d1.

d1	Data from some other Data LAR with same address shown in the address column	61'b1000	3'b100	2'b00	1'b0	1'b0

Figure 10: DATA LARs d1 in the case of cancel LOAD

Therefore, in order to save the cost of fetching the data all the way from the data memory this architecture uses one clock cycle to associatively search for the calculated address in the DATA LARs. This cost would be more visible when we are dealing with SIMD data (multiple data elements in one line). This approach is advantageous because caches are not employed to store vector data elements [2] [3] [25] as it would be a waste of space to have long lines of data in both the registers and the cache.

16

Table 3: Variations of LOAD instruction

Opcode		Instruction	Description
Binary	Hex		
5'b00001	01	LOADUB	Check for the required effective address in the data LARs and then Load unsigned byte from memory in case you can't find that address.
5'b00010	02	LOADUHW	Check for the required effective address in the data LARs and then Load unsigned half-word from memory in case you can't find that address.
5'b00011	03	LOADUW	Check for the required effective address in the data LARs and then Load unsigned word from memory in case you can't find that address.
5'b00100	04	LOADUDW	Check for the required effective address in the data LARs and then Load unsigned double from memory in case you can't find that address.
5'b00101	05	LOADSB	Check for the required effective address in the data LARs and then Load signed byte from memory in case you can't find that address.
5'b00110	06	LOADSHW	Check for the required effective address in the data LARs and then Load signed half-word from memory in case you can't find that address.
5'b00111	07	LOADSW	Check for the required effective address in the data LARs and then Load signed word from memory in case you can't find that address.
5'b01000	08	LOADSDW	Check for the required effective address in the data LARs and then Load signed double from memory in case you can't find that address.

3.2 TYPE CASTING INSTRUCTIONS

Stores are unnecessary for this architecture as it supports lazy store mechanism through which it updates the memory with the changes made to the DATA LARs. Hence STORE instructions are used for type casting in this architecture. As explained above, DATA LARs are all type tagged and therefore the type tags are updated on the new destination LAR. Whenever there is overwrite of data into one of the DATA LARs, the current data is sent into the write buffer from where it will be sent to data memory when the processor finds a free bus cycle. The format of the store instruction is:

Opcode [31:27]	DST [26:22]	SRC1 [21:17]	SRC2 [16:12]	IMMEDIATE [11:0]

Figure 11: Format of STORE instruction

All the fields remain the same for store instructions as they were for load instructions. The way the type castings happen when a store instruction is encountered is better explained with an example. Consider that these two instructions occur in the order shown.

```
LOADUB d1, d2, d3, 5

        ...

STORESHW d1, d4, d5, 10
```

DATA LAR	Data 64 bit								Address		WDSZ 2 bit	TYP 1 bit	D 1 bit
									TAG 61 bit	OFFSET 3bit			
d1	0	5	0	1	0	3	0	4	8	4	2'b00	1'b0	1'b0
d2	32'b0				{28'b0,4'b1}				61'b0	3'b000	2'b10	1'b0	1'b0
d3	16'b0		16'b0		16'b0		{10'b0,6'b1}		1	7	2'b01	1'b0	1'b0
d4	0	0	0	0	0	0	0	4	0	2	2'b00	0	0
d5	0				16				3	4	2'b10	0	0

Figure 12: DATA LARs before the STORE instruction

The effective address of the STORE instruction will be calculated in the same way as for the LOAD instructions. So the calculated value will be:

```
Effective address = 2'b10 + 5'b10000 + 10;

                  = 6'b101100;

           =>TAG = 3'b101 = 5;

        =>Offset = 3'b100 = 4;
```

The content of the DATA LAR d1 will not be changed after the store instruction. Their type fields will be changed. So the data values will look as if they are words now instead of bytes as they were before the store.

DATA LAR	Data 64 bit				Address TAG 61 bit	Address OFFSET 3bit	WDSZ 2 bit	TYP 1 bit	D 1 bit
d1	5	1	3	4	5	4	2'b01	1'b1	1'b0
d2	32'b0		{28'b0,4'b1}		61'b0	3'b000	2'b10	1'b0	1'b0
d3	16'b0	16'b0	16'b0	{10'b0,6'b1}	1	7	2'b01	1'b0	1'b0
d4	0 0 0 0	0 0	0	4	0	2	2'b00	0	0
d5	0		16		3	4	2'b10	0	0

Figure 13: DATA LARs after the STORE instruction

The first LOAD instruction loads d1 with some TAG and WORDOFFSET fields by calculating the effective address as shown above and fills the WDSZ and TYP fields with 2'b00 (byte) and 1'b0 (unsigned) bits. When the STORE instruction gets executed, it will replace all the fields of d1 leaving the 64-bit data associated with it unchanged. WDSZ field will change to 2'b10 (word) and TYP field will change to 1'b1 (signed). So, we can see that STORE instruction does not store the data from d1 to the effective address calculated as we might expect from conventional architectures. It simply changes the address, type and word size fields of d1.

Table 4: Variations of the STORE instruction

Opcode Binary	Opcode Hex	Instruction	Description
5'b01001	09	STOREUB	Change destination LAR's data type to unsigned byte.
5'b01010	0A	STOREUHW	Change destination LAR's data type to unsigned half-word.
5'b01011	0B	STOREUW	Change destination LAR's data type to unsigned word.
5'b01100	0C	STOREUDW	Change destination LAR's data type to unsigned double.
5'b01101	0D	STORESB	Change destination LAR's data type to signed byte.
5'b01110	0E	STORESHW	Change destination LAR's data type to signed half-word.
5'b01111	0F	STORESW	Change destination LAR's data type to signed word.
5'b10000	10	STORESDW	Change destination LAR's data type to signed double.

3.3 ARITHMETIC AND LOGICAL INSTURCTIONS

The arithmetic and logical instructions are designed to support the different packed and unpacked SIMD data types. The DATA LARs architecture supports ADD, SUB, AND, OR, and EXOR operations. Unlike MMX [18][19] or SSE designs, where the architecture has 57 or more opcodes to actually distinguish the type conversion operations, LARs design has only five different opcodes for these five instructions and the rest of the bits of the 32-bit instruction are used as control bits to help resolve the boundaries of the type conversions.

Therefore, all these type conversions take place during the flow of data from one pipeline stage to another and not as separate instructions in which case we would have to wait for this converted data to be placed back in the registers. Hence this design would reduce the instruction fetch bandwidth and would be faster compared to the present architectures which employ separate registers for SIMD operations.

The basic arithmetic instruction format would be

Opcode [31:27]	Dest [26:22]	SRC1 [21:17]	SRC2 [16:12]	SV [11]	Offset1 [10:8]	Offset2 [7:5]	Dest offset [4:2]	NO USE [1:0]

Figure 14: Format of Arithmetic instructions

The ADD and SUB instructions have 4 variations in them. The 11th bit of the 32-bit instruction is used to specify if it is a vector or a scalar instruction. The offsets from bit 1 to bit 10 are all used for the type conversions for packing and unpacking the data to required ALU boundaries. Vector operations are made on the whole array of data inside a DATA LAR, and scalar operations are made on the data pointed to by the WORDOFFSET of that particular data LAR. All calculations are made by taking WDSZ and TYP bits into consideration which are filled by the instructions executed earlier in that pipeline process. In case of data hazards, forwarding of these required bits (WDSZ and TYP) would be done and for cases where this is not possible, pipeline bubbles would be introduced to stall until the required data is ready to be forwarded.

Table 5: Arithmetic instructions and their variations

Opcode		Instruction	Description
Binary	Hex		
5'b10010	12	ADD	This is classified into ADDV and ADDS namely vector or scalar addition depending on the 12th bit of the instruction. And at the end of this instruction, associatively update the data LARs with same address field.
5'b10011	13	SUB	This is classified into SUBV and SUBS, vector or scalar subtraction respectively depending on the 12th bit of the instruction. And at the end of this instruction, associatively update the data LARs with same address field.
5'b10100	14	MUL	(Reserved) for multiplication.
5'b10101	15	AND	Perform AND operation on the two source LAR lines. And at the end of this instruction, associatively update the data LARs with same address field. Classified into ANDS and ANDV.
5'b10110	16	OR	Perform OR operation on the two source LAR lines. And at the end of this instruction, associatively update the data LARs with same address field. Classified into ORS and ORV.
5'b10111	17	EXOR	Perform EXOR operation on the two source LAR lines. And at the end of this instruction, associatively update the data LARs with same address field. Classified into EXORS and EXORV.

All the ALU operations follow modular arithmetic, also known as wrap-around arithmetic. It is the normal computer arithmetic in which the stored result is the low n bits of the actual result, where n is the size of the space in which the result is to be stored. This is equivalent to taking the actual result modulo the maximum value storable in the available space. Most existing instruction sets include some form of modular addition except for MVI, which does not, and the extensions to MMX, which use the MMX instructions for this purpose. When the source operand's word size is bigger than the destination operand's word size, the bits that are in excess when compared to the destination operand's bits will be discarded. Figure 15 will explain this procedure.
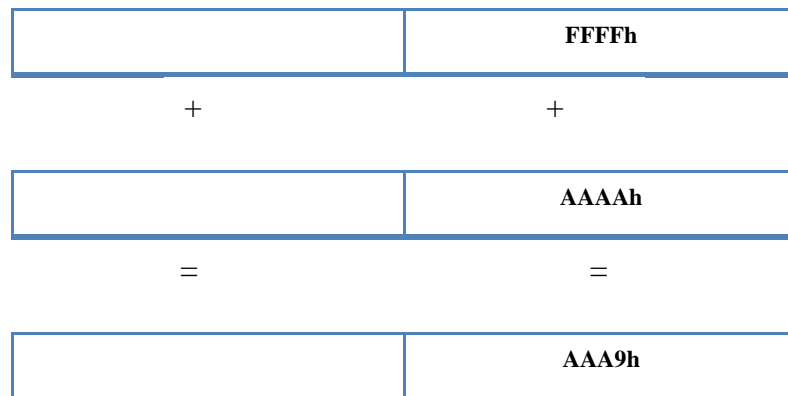
	FFFFh
+	+
	AAAAh
=	=
	AAA9h

Figure 15: modular arithmetic followed by the ALU

In this case, the right-most result exceeds the maximum value representable in 16-bits thus it wraps around. This is the way regular computer arithmetic behaves. FFFFh + AAAAh would be a 17 bit result. The 17th bit is lost because of wrap around, so the result is AAA9h.

3.3.1 TYPE CONVERSIONS

Since LARs design is type tagged, the data type information is used by the hardware to process the SIMD data by keeping track of the data type fields. There will not be any explicit data conversions generated by the compiler as in the Intel's iAPX432 [22], so considerable compiler overhead is saved.

The third pipeline stage is used for these type conversions. The two operands are sent to the Sign Extension-Truncation Unit and in a case where the operand is of different word size when compared to the destination register's word size field, depending on the sign bit of the operand, sign extension or truncation of the operand's data to the destination operand's word size takes place. If the operand is unsigned, sign extension would be appending zeros and if the operand is signed, sign extension takes place by appending ones. In case of truncation, the conversion follows saturation arithmetic rules. It is best explained by considering what happens in computing an assignment like:

C=A+B

3.3.1.1 SATURATION ARITHMETIC

Saturation arithmetic is a form of computer conversion in which the result is set to the maximum storable value of the same sign when an overflow occurs. This form of addition is used primarily for multimedia applications in which the data value represents some physical parameter whose value should not wrap with incremental changes. For example, adjusting the volume level of a sound signal can result in overflow, and saturation causes significantly less distortion to the sound than wrap-around (wrap around may result in a sudden drop from high to 0). LARs architecture follows saturation

arithmetic rules to truncate the operand's data in case it does not fit in the destination LAR.

Consider a case where A is byte (8 bits), B is a half word (16-bits) and C is word (32-bits). Now the boundaries of the ALU will be set according to the destination register's word size which is a word (32-bits). A will be converted from 64-bit data with bytes in it to 64-bit data with words in it. We take a temporary register D which acts as a pipeline register for the later stage and do the following operation.
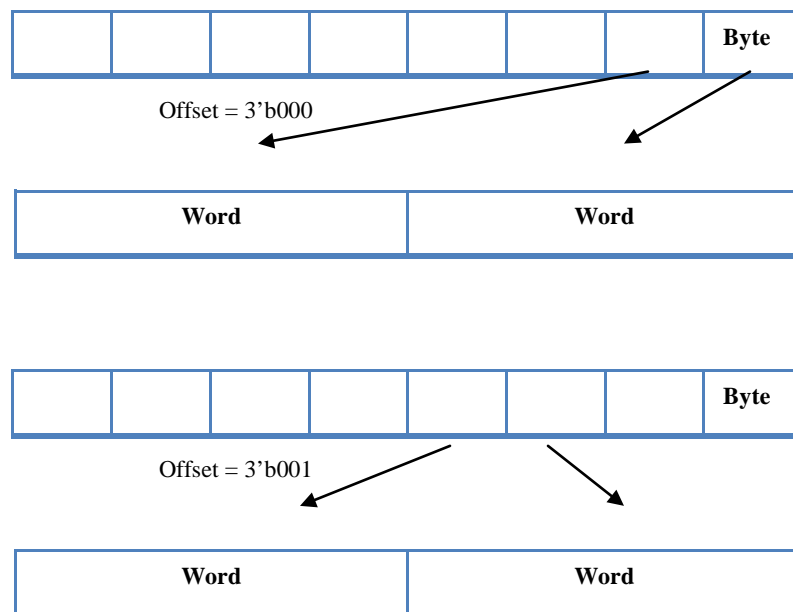
Figure 16: Type conversion- Rules for sign extension in the DATA LARs architecture

The two cases with offset as 3'b000 and 3'b001 are derived from the offset field of the 32-bit instruction. If the offset of the source register A is given to be 3'b000 then the type conversion will be applied on the first two eight bits of the DATA LAR A. If the offset is 3'b001 then the next two 8-bits of A[23:16] and A[31:24] will be considered and so on. There are 3-bits for offset because the maximum possibility of selecting a pair of data elements occurs in case if A is a byte and C is a double where there are 8 possibilities.

```
D[7:0] = A[7:0]; D[31:8] = 24 zero's
D[39:32] = A[15:8]; D[63:40] = 24 zero's.
```

23

If A is signed, then sign extension is performed:

```
D[7:0] = A[7:0];
If (A[7] == 1'b1) C[31:8] = 24 1's
else C[31:8] = 24 0's;

D[39:32] = A[15:8];
If (A[15] == 1'b1) C[63:40] = 24 1's
else C[63:40] = 24 0's;
```

Now consider a case where we have to truncate the operand A to fit in the destination register. Let A be a word (32-bits), B be a half word (16-bits) and C be a byte. The ALU boundaries will be set to byte. If A is an unsigned word data type we compare the magnitudes of A[31:0] and A[7:0] and if A[31:0] is greater than or equal to A[7:0] we place 8 1's in D[7:0] which is greatest possible number for unsigned numbers or else we place A[7:0] in D[7:0]. We do the same to the next 32 bits of A and place the truncated result in D[15:8]. The rest of D[63:16] is filled with zeros.
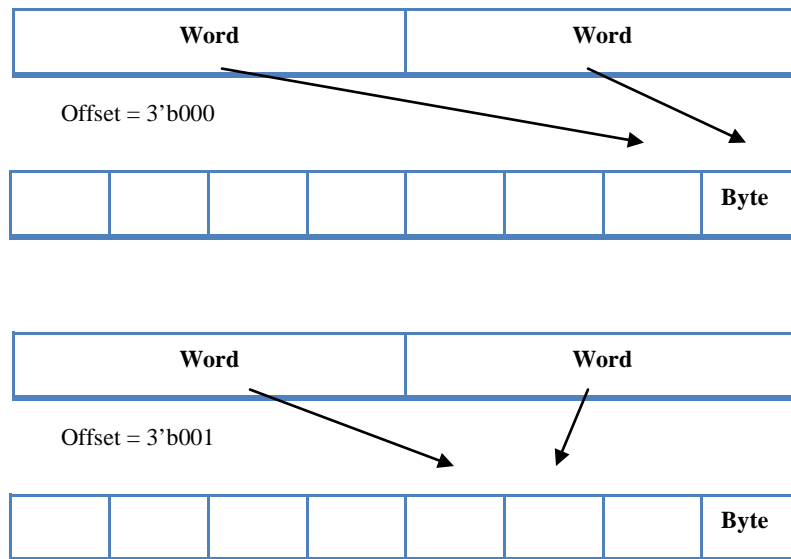
Figure 17: Type conversion- saturation arithmetic rules for packing data

24

In case of signed data type we see if the most significant bit of A is 1 or not. If it is a 1 then we replace the `C[7:0]` with `8'b10000000` which is the least negative number. If it is not we check for the magnitudes again and replace `D[7:0]` with `8'b01111111` which is greatest positive number for signed numbers in the case `A[31:0] > A[7:0]`. If it is not greater then we just copy the value `A[7:0]` into `D[7:0]`. The rest is again filled with zeros.

3.3.2 SCALAR ALU OPERATIONS

Unlike SSE-1 or SSE-2 extensions, LARs can perform flexible scalar operations [24]. When the SV bit (11th bit of the instruction) is high the processor considers the instruction to be a scalar instruction. In this case we allow only the data pointed to by the word offset to pass through to the pipeline register which supplies the operands to the ALU. This isolated data is sign extended or truncated according to the destination LAR's WDSZ and TYP fields. Finally, after the data is processed at the ALU, we place the data at the position pointed to by the destination LAR's word offset field. The following example explains the way in by which a scalar `ADD` instruction is executed:

ADDS D1, D4, D5;

DATA LAR	Data 64 bit								Address		WDSZ 2 bit	TYP 1 bit	D 1 bit
									TAG 61 bit	OFFSET 3bit			
d1	0	5h	0	1h	0	3h	0	4h	8	4	2'b00	1'b0	1'b0
d2	0				FFh				0	0	2'b10	1'b0	1'b0
d3	0		0		0	3Fh			1	6	2'b01	1'b0	1'b0
d4	0	0	0	0	0	5h	0	4h	2	2	2'b00	0	0
d5	FFFFh				Fh				3	4	2'b10	0	0

Figure 18: DATA LARs before the ADDs Instruction

The address of d4 points to the third location (byte) from the right. That particular byte is taken out and sent to the conversion unit where it will be aligned with the boundaries of the ALU as discussed before. Here the boundaries of ALU are set to byte since the destination LAR's WDSZ is byte, so there will be no need to convert the contents of d4. Since d5 has words in it, its contents will be truncated to bytes as explained before. Only the second word which is FFFFh is sent out to get converted since the word offset is pointing to it.

After masking and shifting the data pointed to by the offset is extracted out and truncated. Only after these operations does the processor add the two operands. The destination LAR d1's contents will be brought along through the pipeline registers, and at the end of the ALU operation the result will be shifted to the position pointed to by the destination LAR's word offset and placed at the required position in that pipeline register. After the whole scalar arithmetic operation, the shifted sum replaces the data pointed to by the word offset of the d1 DATA LAR. So, 1h is replaced by 4h. The following figure gives us the end result.

DATA LAR	Data 64 bit								Address		WDSZ 2 bit	TYP 1 bit	D 1 bit
									TAG 61 bit	OFFSET 3bit			
d1	0	5h	0	4h	0	3h	0	4h	8	4	2'b00	1'b0	1'b0
d2	0				FFh				0	0	2'b10	1'b0	1'b0
d3	0		0		0	3Fh			1	6	2'b01	1'b0	1'b0
d4	0	0	0	0	0	5h	0	4h	2	2	2'b00	0	0
d5	FFFFh				Fh				3	4	2'b10	0	0

Figure 19: DATA LARs after the ADDs instruction. d1's value pointed by the word offset is changed from 1h to 4h

3.4 NO-OP

This is used by the hardware internally to introduce pipeline bubbles in case the forwarding unit does not meet the data hazards requirements. This is recognized by its opcode field which is all zeroes- 5'b00000. When this opcode is seen at the second stage (Instruction Decode Stage), the hardware makes all the control signals required by the pipeline hardware for particular ID stage to point to do harmless work, thereby acting like a pipeline bubble.

3.5 LOADDUMMY

This instruction is used before an ALU instruction when the destination DATA LAR's address and type fields are not set by a previous instruction. This is necessary because of the way this architecture is designed. The type tags are placed on the DATA LARs along with the fetched data from main memory by the LOAD instructions. So for cases where the DATA LAR has to used before a LOAD or STORE instruction has filled its contents, this instruction is sent through the pipe just before the main instruction. The hardware recognizes this instruction and lets it pass through the pipe like a STORE instruction. So this instruction has the same format as the STORE or a LOAD with same effective address calculation procedure. Its opcode is 5'b11111.

3.5 SUMMARY OF INSTRUCTION SET ARCHITECTURE

Table 6: Summary of Instruction Set Architecture

Opcode		Instruction	Description
Binary	Hex		
5'b00000	00	NO-OP	This is no operation instruction which is used by the hardware to introduce a pipeline bubble in case of data hazards.
5'b00001	01	LOADUB	Check for the required effective address in the data LARs and then Load unsigned byte from memory in case you can't find that address.
5'b00010	02	LOADUHW	Check for the required effective address in the data LARs and then Load unsigned half-word from memory in case you can't find that address.
5'b00011	03	LOADUW	Check for the required effective address in the data LARs and then Load unsigned word from memory in case you can't find that address.
5'b00100	04	LOADUDW	Check for the required effective address in the data LARs and then Load unsigned double from memory in case you can't find that address.
5'b00101	05	LOADSB	Check for the required effective address in the data LARs and then Load signed byte from memory in case you can't find that address.
5'b00110	06	LOADSHW	Check for the required effective address in the data LARs and then Load signed half-word from memory in case you can't find that address.
5'b00111	07	LOADSW	Check for the required effective address in the data LARs and then Load signed word from memory in case you can't find that address.
5'b01000	08	LOADSDW	Check for the required effective address in the data LARs and then Load signed double from memory in case you can't find that address.
5'b01001	09	STOREUB	Change destination LAR's data type to unsigned byte.
5'b01010	0A	STOREUHW	Change destination LAR's data type to unsigned half-word.
5'b01011	0B	STOREUW	Change destination LAR's data type to unsigned word.
5'b01100	0C	STOREUDW	Change destination LAR's data type to unsigned double.
5'b01101	0D	STORESB	Change destination LAR's data type to signed byte.
5'b01110	0E	STORESHW	Change destination LAR's data type to signed half-word.
5'b01111	0F	STORESW	Change destination LAR's data type to signed word.
5'b10000	10	STORESDW	Change destination LAR's data type to signed double.
5'b10010	12	ADD	This is classified into ADDV and ADDS namely vector or scalar addition depending on the 12th bit of the instruction. And at the end of this instruction, associatively update the data LARs with same address field.
5'b10011	13	SUB	This is classified into SUBV and SUBS, vector or scalar subtraction respectively depending on the 12th bit of the instruction. And at the end of this instruction, associatively update the data LARs with same address field.

Table 7: Summary of Instruction Set Architecture (contd...)

5'b10100	14	MUL	(Reserved) for multiplication.
5'b10101	15	AND	This is classified into ANDS and ANDV namely scalar or vector And operation depending on the 12th bit of the instruction. And at the end of this instruction, associatively update the data LARs with same address field.
5'b10110	16	OR	Perform ORS or ORV operation on the two source LAR lines. And at the end of this instruction, associatively update the data LARs with same address field.
5'b10111	17	EXOR	Perform EXORS or EXORV operation on the two source LAR lines. And at the end of this instruction, associatively update the data LARs with same address field.
5'b11111	1f	LOADDUMMY	This is just a dummy instruction sent before an ALU instruction in a case where the destination DATA LAR does not have any of the fields set.

CHAPTER 4
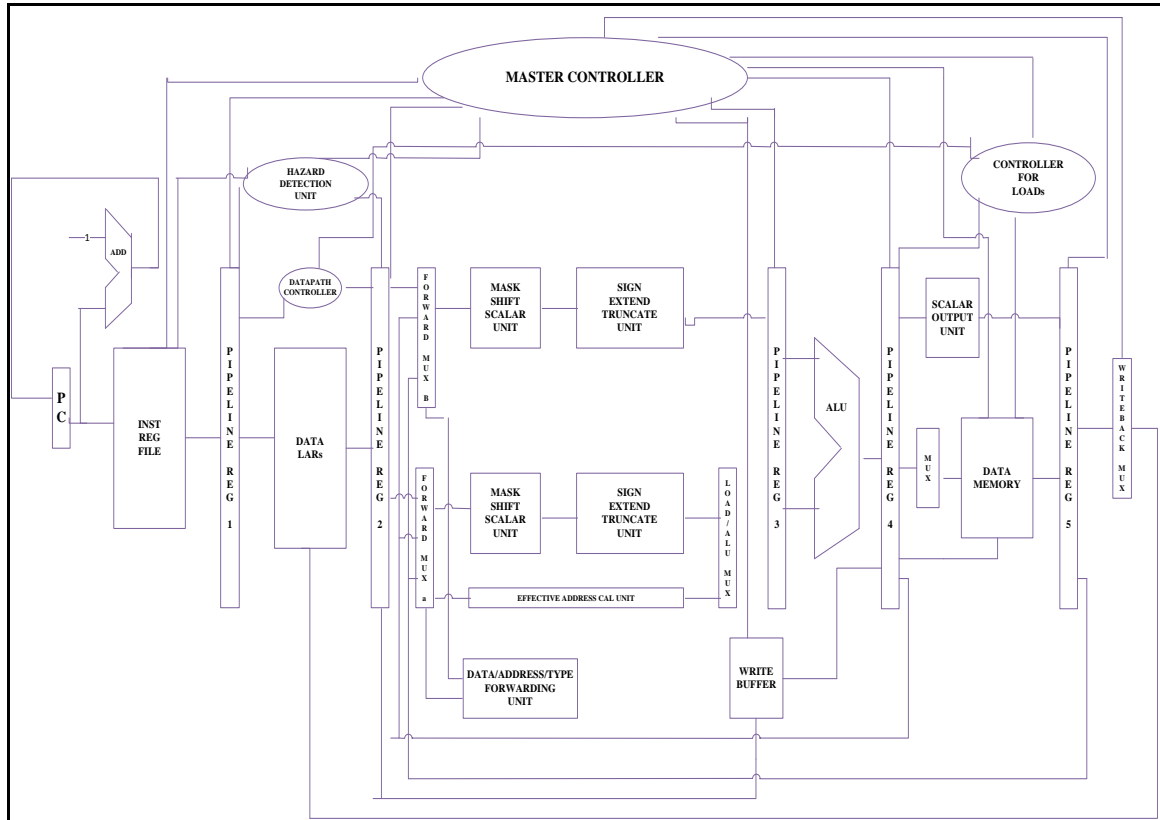
4. DATA LARS ARCHITECTURE

Figure 20: Block diagram of DATA LARs architecture design

The DATA LARs Architecture design is a six-stage pipelined architecture. The stages are named as follow:

Stage 1: Instruction Fetch

Stage 2: Instruction Decode

Stage 3: Conversion

Stage 4: Execution

Stage 5: Memory

Stage 6: Write Back

The Hardware units designed and used for this architecture are as follow:

- ➢ Instruction Register File with 32 32-bit registers to give instructions every cycle to the data path. Since this design being a simulator for testing the DATA LARs unit in an ordinary architecture (replacing the normal CPU registers), there is no instruction memory that is actually being used to fetch code from. This register file just acts as a continuous supplier of instructions to test the actual concept of DATA LARs.

- ➢ The DATA MEMORY is designed to supply the 64-bit data required by the 64-bit wide DATA LARs. So it's designed to contain 16 64-bit wide registers. This design is this small because of the chip constraints that it is being design on. The main aim of this design is to fit as many registers as possible to actually test the validity of the concepts that it is based on and in this process add some more ideas to make it work and be efficient.

- ➢ The DATA LARs unit which is the replacement of the normal CPU registers when compared to a normal architecture. These registers are 132-bits wide each as they were described in the section 2. The actual data is of 64-bits wide SIMD data and the rest of the bits are used to store the address and type information of the same. There are about 8 of these registers employed in the current design.

- ➢ Five pipeline registers named as IF/ID, ID/CONV, CONV/EX, EX/MEM, MEM/WB.

- ➢ Write Buffer with two 125-bit registers to lazy-store the evicted data and address from the DATA LARs to main memory.

- ➢ There are 3 ALUs in this design. The first ALU is fixed to add only and it is for incrementing the program counter to point to the next instruction in the Instruction Fetch stage. The second ALU is also fixed to add only in the conversion stage where it is used as an initial adder for calculating the main effective address for `LOAD` and `STORE` instructions. And the third ALU is a carry select adder which is used for calculating `ADD`, `SUB`, `AND`, `OR` and `EXOR` instructions and also for the final effective address calculation in case of `LOAD` and `STORE` instructions.

- The MASK SHIFT SCALAR UNIT is activated only for scalar operations. This unit allows only the required data pointed by the WORD OFFSET field of the particular DATA LAR in question. It masks the rest of the data and shifts it to the least significant bit position.

- The SIGN EXTEND/TRUNCATE UNIT is the main hardware block where all the sign extensions and compressions required by the operand's data to level according to the ALU's boundaries are made. The ALU's boundaries are set according to the type of the destination DATA LAR.

- The EFFECTIVE ADDRESS CALULATION UNIT is used to add the initial two operands: The 64-bit address and the sign extended 12-bit immediate field.

- The SCALAR OUTPUT UNIT shifts the ALU output back to its original position to which the destination DATA LAR's WORD OFFSET field was pointing to. And this shifted output is placed back in the old destination DATA LAR so that only the pointed data is changed.

- Since the DATA LARs architecture is type tagged and also lets the registers contain the address of the data, forwarding information to prevent data hazards has to be done for all the three fields of the DATA LAR. Therefore, the DATA/ADDRESS/TYPE FORWARDING UNIT does all the three forwards.

- The ALU at the Execution stage comprises of a 64-bit carry select adder with a provision to add byte sized 64-bit packed SIMD data. It could be viewed as a collection of eight 8-bit adders. The carries from these adders are monitored and managed by the hardware block called the ANDING UNIT.

- The HAZARD DETECTION UNIT detects the data hazards which cannot be cured by the forwarding unit. It sends the main controller signals to stall the pipeline process by introducing a no-op instruction.

- There are three controllers in this design. The first is DATAPATH CONTROLLER. This generates signals to hardware in the later stages which solely depend on the opcode. The second controller is the CONTROLLER FOR LOADs. This is really important as it is responsible to send the interrupt signal to main controller to stall the whole pipeline process for one clock cycle in case of a LOAD instruction. This controller actually incorporates the main concept of

LARs which is to check for a possible match of effective address in the DATA LARs itself due to the previous LOAD instructions. In case of a match of the calculated effective address in the DATA LARs, the actual fetch cycle for data from main memory is cancelled and a copy of the found data at this address location inside the DATA LAR is sent to the MEM/WB pipeline register, after which this data would be copied into the destination DATA LAR. This is a structural hazard and this controller deals with the same issue.

➢ The Third controller is the MAIN CONTROLLER whose task is to keep the other two controllers in sync and take care of stalls by considering the interrupts due to data and structural hazards.

4.1 INTERRUPTS

There are 4 interrupts in this current design.

1. Interrupt from the Hazard Detection Unit.
2. Interrupt from the CONTROLLER for LOADs.
3. Interrupt from the WRITE BUFFER.
4. Interrupt from the DATA MEMORY.

All these interrupts are handled by the main controller appropriately.

4.2 TRADE OFFs OF DATA LARs

The DATA LARs architecture is six stage pipelined design. The conversion stage is the difference between the five stage MIPs design and the current DATA LARs design. This is introduced here because of the conversions that are needed between the packed data types to align the data to a proper boundary. MIPs architecture does not follow the SIMD execution pipeline, so it doesn't need a conversion stage. So the additional stage being a necessity for having to deal with the vector registers in the DATA LARs design, there isn't a lot of difference between a MIPs five stage pipelined design and our DATA LARs six stage pipelined design. Therefore MIPs is a good architecture for the current design to compare and point out the advantages and disadvantages of having DATA LARs in an ordinary architecture. The following section describes the main tradeoffs of LARs

architecture by comparing it with MIPs architecture. Consider the figure below as a reference for a normal pipelined execution where forwarding helps prevent a pipeline bubble (stall). This figure would be changed according to the problem in question. Hence understanding how this works is essential in order to understand the rest of the chapter. The dependencies between the pipeline registers move forward in time, so it is possible to supply the inputs to the ALU needed by the AND instruction by forwarding the results found in the pipeline registers. ADD instruction is first issued and then comes the AND instruction. Register R3's value is available only at the end of 3rd clock cycle and the AND instruction needs this value at the start of the 4th clock cycle. Since the data dependency goes forward in time, there are no data hazards as we can just route back the value of the ALU from the pipeline register to the input of the ALU.
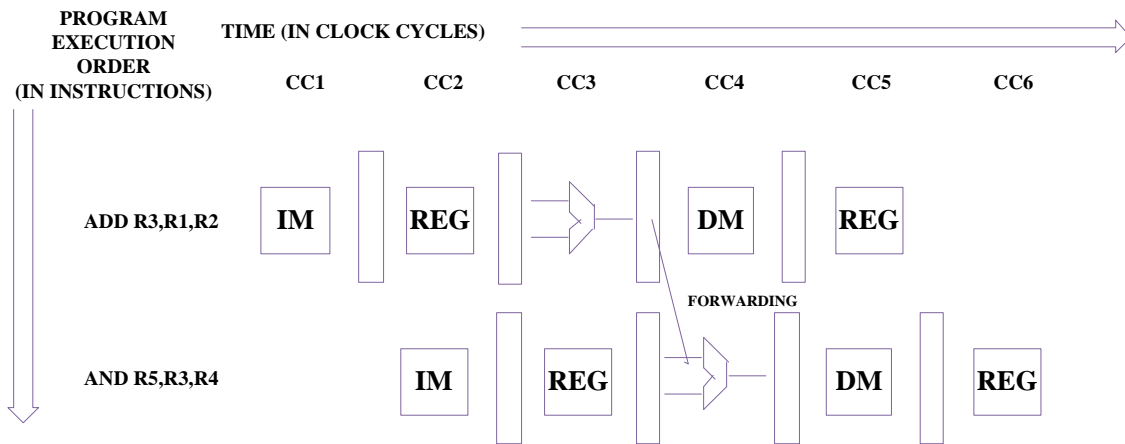
Figure 21: Forwarding the pipeline saves the cost of a stall cycle

4.2.1 DATA HAZARDS

In MIPs, one case where forwarding cannot save the day is when an instruction tries to read a register following a load instruction that writes the same register. The figure 22 below shows this case. The hazard forces the AND instruction to repeat in clock cycle 4 what it did in clock   cycle 3. This would just delay the fetch of the next instruction and therefore like an air bubble in a water pipe, a stall bubble delays everything behind it and proceeds down the instruction pipe until it exits at the end.

34
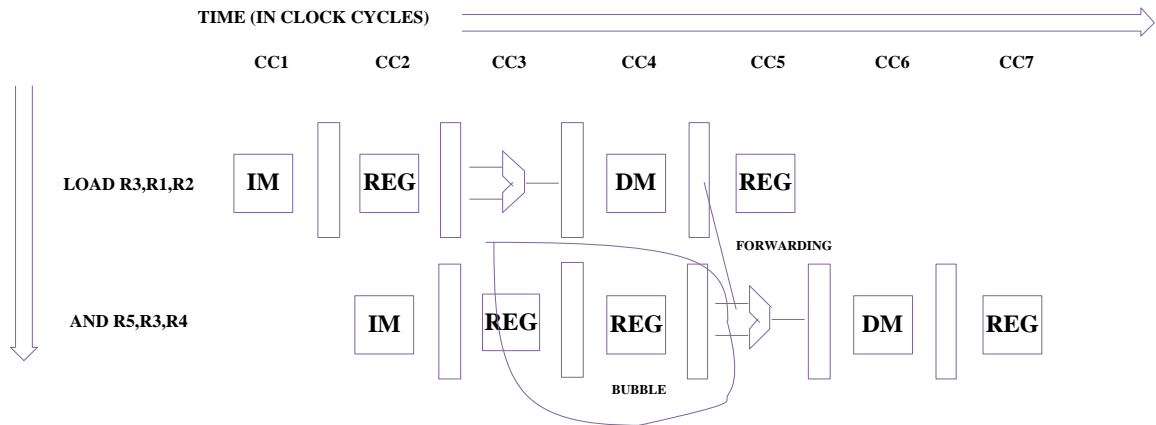
Figure 22: This shows what really happens in the MIPs hardware when the data dependency goes backward in time.

DATA LARs architecture introduces 2 stall cycles, compared to one stall as in the case of MIPs, for same kind of problem.
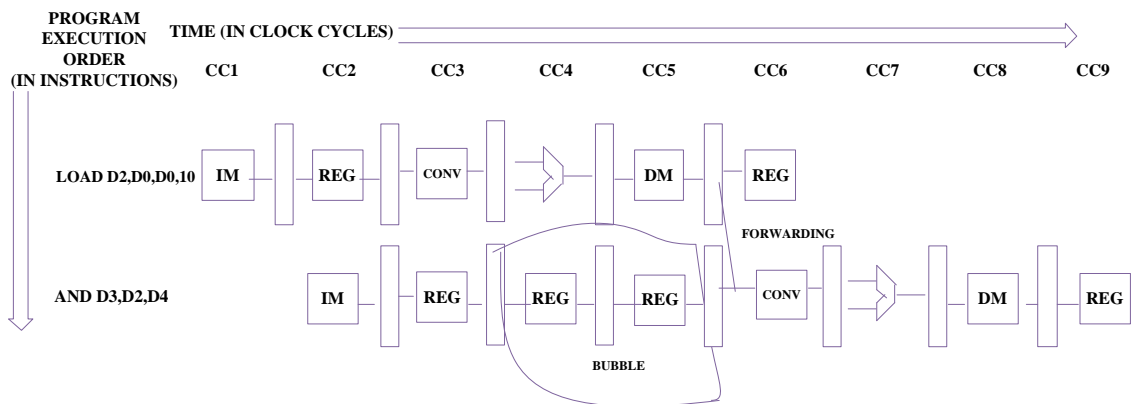
Figure 23: For same problem the hazard forces the AND instruction to repeat in the clock   cycles 4 and 5 what it did in clock cycle 3.

This happens because of the conversion stage. Since the architecture deals with packed SIMD data, it needs to convert the forwarded before actually sending the aligned data to the ALU. Hence the pipeline is stalled until the data dependency is forward in time for the conversion stage and this needs two clock cycles stall. Again, there is a data hazard involved when an instruction tries to read a register following an arithmetic instruction that writes the same register. This is due to the same problem explained above, because of the conversion stage.
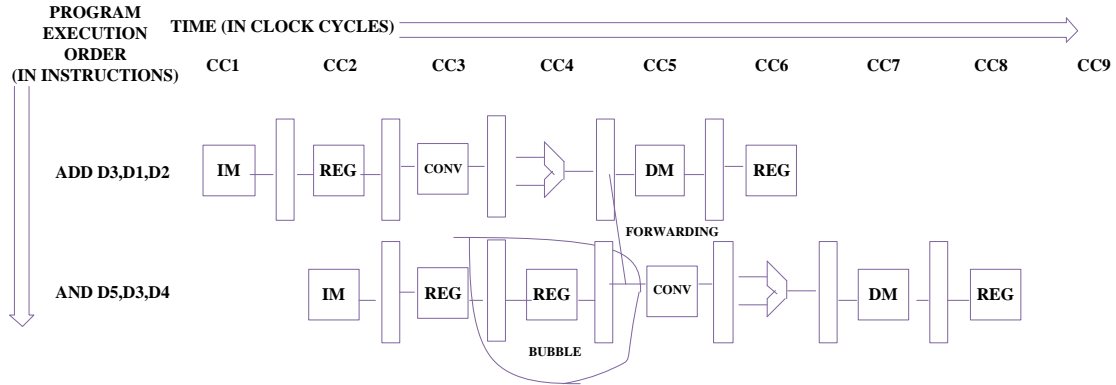
Figure 24: Clock cycle 4 repeats the operations done during clock cycle 3 to introduce a pipeline bubble.

This architecture does not allow ALU's output to be routed back to one of the inputs of the ALU as in the case of MIPs architecture. This is because of the data is needed to be aligned properly before it is sent to the ALU and this process is done in the conversion stage, hence a stall cycle.

4.2.1.1 ARGUMENT

Although this looks like an additional stall cycle compared to MIPs, this is just a tradeoff for dealing with large amounts of data. Having SWAR like architecture helps in reducing the number of instructions needed to actually perform the basic operations such as ADD. Consider the case of MMX of instructions. This architecture helps the Intel's x86 designs to deal with SIMD data. Now for a normal arithmetic operation like an ADD, it needs to issue type conversion instructions before actually issuing the main instruction. That is the reason why the designs with MMX-like register sets need to have large instruction set (Intel's MMX has 57 different instructions to deal with these wide registers). The cost of having these additional stall cycles is negligible when compared to the advantages that the architecture has due to the wide type-tagged registers.

4.2.2 STRUCTURAL HAZARD

The LARs architecture does an associative search of the DATA LARs before every LOAD instruction to check for the presence of the calculated effective address in the DATA LARs. Since this is a pipelined architecture, each hardware unit could be used

36

only once in a clock cycle. Now this associative search needs the DATA LARs registers during the 5th stage (Memory stage). At the same time another instruction might need the DATA LARs to write back the executed data during the 6th stage. Since the design lets DATA LARs to be read at any point of time, there would be a collision of usage of these registers at the 5th and 6th stages when the hardware encounters a LOAD instruction at the 5th stage. This is a Structural Hazard.
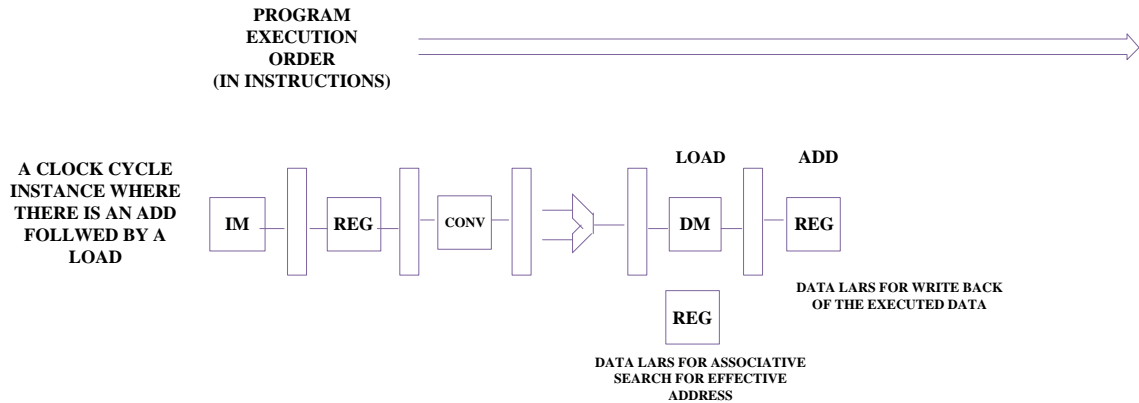
Figure 25: The structural hazard problem with DATA LARs design for LOADs.

At an instance where there is a LOAD after an ADD instruction, the load wants to search the DATA LARs at the same time the ADD instruction is writing the executed data into the DATA LARs. The figure 26 below explains the problem in detail. DATA LARs deals with this problem in this way; the CONTROLLER FOR LOADs sends an interrupt signal to the MAIN CONTROLLER and the main controller stalls all the 6 pipeline registers for one clock cycle CC1. During this stall, the associative search for the effective address in the DATA LARs takes place and this is shown in the figure by highlighting only the REG block. At the end of the stall cycle CC1 another interrupt is sent to the MAIN CONTROLLER telling it if there was a hit or a miss. In case of a hit, the controller cancels the fetch cycle for the LOAD instruction and in case of a miss the controller continues its usual process of fetching the packed SIMD data from the main memory. This is shown in the CC2 where all the hardware blocks are active and this time the DATA LARs are used by the Write Back stage to write back the executed ADD instruction's data.
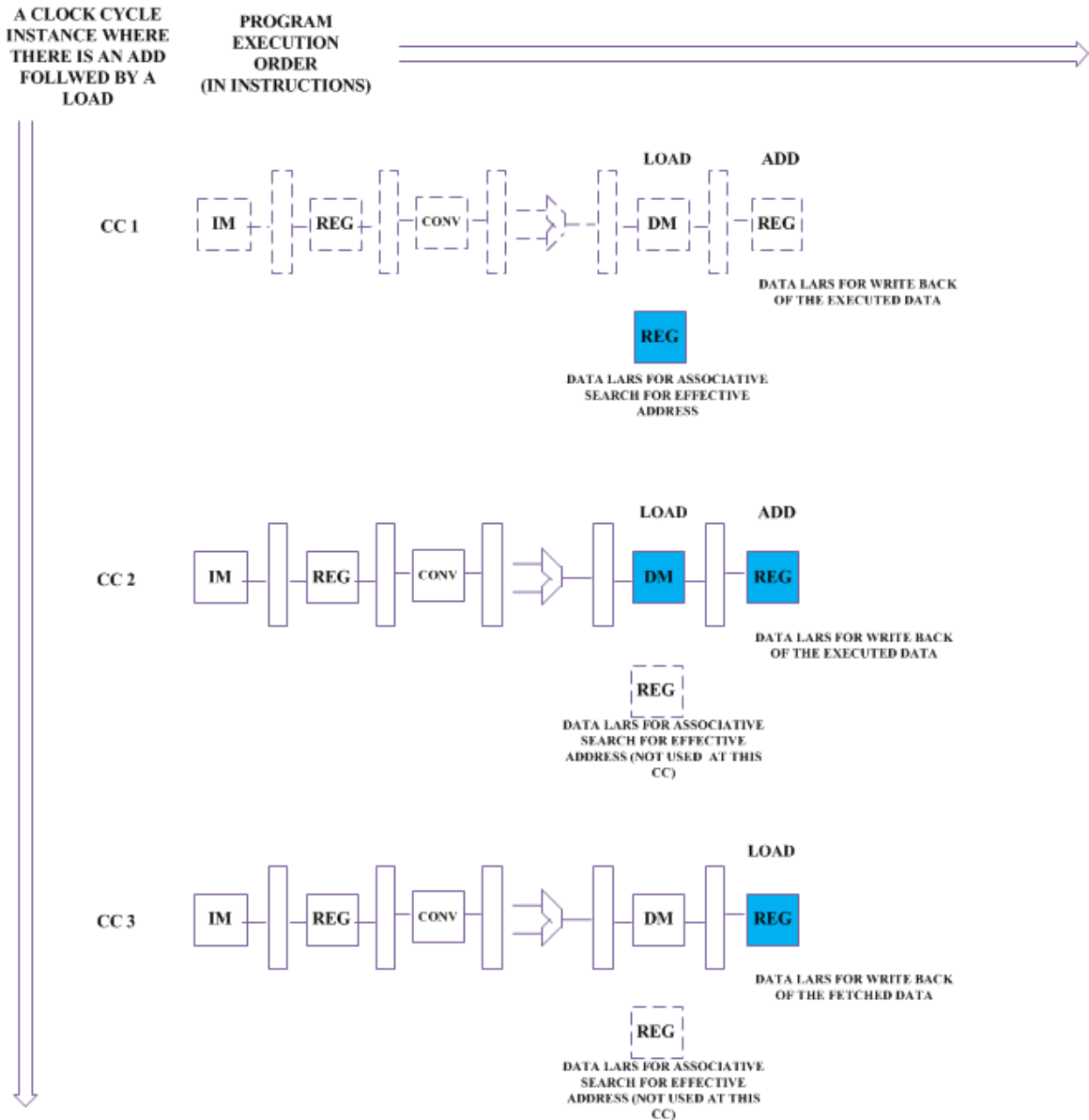
Figure 26: The flow of instructions through the pipeline when there is a load instruction.

And during the CC3 the fetched data from either the Main Memory (in case of a miss) or the DATA LARs (in case of a Hit) is sent to DATA LARs to be written onto the destination LAR pointed by the write address of the LOAD instruction.

4.2.2.1 ARGUMENT

The cost of a stall cycle before every LOAD instruction is negligible when compared to the cost going all the way back to the Main Memory for fetching data. This is also strongly supported as the width of the registers is not just one object wide in this design.

Having a cache does not help a lot either. In traditional cache-based computers, all the memory references are made through cache and most of the items that are referenced in a program are referenced infrequently which leads to bumping the fetched data from the cache even before they are referenced again. In such cases, there is no benefit in placing the item in the cache. Moreover, there is an additional overhead of bumping some other items out of the cache to make room for these useless entries into cache [2] [25]. And it is wastage of space in our case to have the same item in both cache and the DATA LARs as they are wide data elements.

4.2.3 ASSOCIATIVE SEARCH OF LOAD INSTRUCTIONS

4.2.3.1 ARGUMENT

People might be concerned that the LARs (and CRegs before them) would be stretching the cycle time or making the register access take multiple cycles; however, the Verilog module of the LARs system design does NOT evidence this (nor did the earlier CRegs circuits designs). The reason that LARs do not take longer to access than conventional registers is that, for reads and writes, they truly ARE wide, but conventional, registers. The associativity affects only the associative processing of Loads, which is still faster than L1 cache's associativity because the LARs are on the processor side of any TLBs.

CHAPTER 5

5. RESULTS

Like a conventional register, DATA LARs can hold the values of the resulting operations. Like a cache line, LARs can contain a number of spatially local scalar values. As in SWAR, a LAR can hold a vector of values to be operated on in parallel. Like their progenitor CRegs, LARs can transparently resolve ambiguous aliases in hardware. The following examples explain the operations of DATA LARs with the help of snapshots taken from the simulations run on the DATA LARs simulator designed using Verilog Hardware Descriptive Language (HDL).

5.1 TRIVIAL EXAMPLE

To demonstrate some of these properties, a trivial example, compiled to both LARs- and MIPS-like assembly.

```
nasty(int* i, int* j, int* k)
{
    *i=*j+*k;
    *k=*i&*k;
}
```

Listing 2: Example with pointers for alias analysis

The same C code would be compiled into the following instructions and get executed likewise. As we can see in the table below the RISC processor needs around five more instructions to execute the same program and uses 9 memory cycles over all, whereas the DATA LARs processor takes only a maximum of 5 memory cycles. The table below describes the execution of the above C code in terms of MIPs and DATA LARs processor designs.

Table 8: Assembly language code of the C program "nasty" for DATA LARs and MIPs

Conventional RISC	DATA LARs
LW $t1, j(0)	LOADUDW d1, d0, d0, i
LW $t2, 0($t1)	LOADUDW d2, d0, d0, j
LW $t3, k(0)	LOADUDW d3, d0, d0, k
LW $t4, 0($t3)	LOADDUMMY d6, d0, d1, 0
LW $t5, i(0)	LOADSW d4,d0,d2, 0
ADD $t6, $t2, $t4	LOADSW d5,d0,d3, 0
SW $t6, 0($t5)	ADDS d6, d4,d5
LW $t7, 0($t5)	AND d5, d6, d5
LW $t8, 0($t3)	
AND $t9, $t7, $t8	
SW $t9, 0($t3)	

Table 9: Static Comparison of DATA LARs and MIPs

	LARs	MIPs
Memory Accesses	0-5	9
Reads	0-5	7
Writes	0	2
Total Instruction Count	8	11

The reason there are ranges on the LARs memory access counts is because if the used locations are aliased to *any* "live" value, the memory access is replaced with a simple associative update. Even if we allow that the MIPS version may have passed its parameters in registers, it would only reduce the number of instructions to parity at eight, and there would still be eight assured memory accesses for the MIPS version. While it is true that some or all of these memory accesses may be satisfied from cache, this would still require traffic across the memory interface. An associative update in the LARs version is entirely internal to the processor, and does not incur any bus traffic. It is also

worth noting that the LARs version could operate on entire vectors, each the length of the data field, by only changing the ADD and AND operations to their vector forms. There would be **no** additional memory references or additional cycles in processing the operations.

5.1.1 EXECUTION OF ALIAS ANALISIS EXAMPLE ON DATA LARs SIMULATOR

The program in listing 2 is modified at the assembly language level a bit so that it could be fed to the DATA LARs simulator and shown in the table 9 below. This section explains the execution of alias analysis program by sending it through the simulator and the results obtained. It assumes the initial values of the memory as shown throughout the section and explains the obtained results according to these values.

Table 10: the assembly language instructions modified for DATA LARs to test the alias analysis program of listing 2.

DATA LARs INSTRUCTION	OPERATION
LOADUDW d1, d0, d0, 0	Loads data = 28 from memory location 0; 28 is the address location of i => addr(i)
LOADUDW d2, d0, d0, 8	Loads data = 18 from memory location 1; 18 is the address location of j => addr(j)
LOADUDW d3, d0, d0, 10	Loads data =20 from memory location 2; 20 is the address location of k => addr(k)
LOADDUMMY d6, d0, d1, 0	Not an actual load. This will just calculate the effective address of variable i and place this at the data LAR d6 with type information so that the added value would be placed with proper attributes. No fetch cycles required here when compared to MIPs or other RISC architectures. =>val(i)
LOADSW d4,d0,d2, 0	Loads the data of variable j using the address fetched by the instruction into d2. =>val(j)
LOADSW d5,d0,d3, 0	Loads the data of variable k using the address fetched by the instruction into d3. =>val(k)
ADDV d6, d4,d5	Adds the fetched integer values of j and k and places the result in d6, which has the address of integer i
ANDV d5, d6, d5	Performs ANDV operation on the result of previous addition and k and places result in k- d5

42

The integer values of j and k are

J = 32'h dddd dddd;

K = 32'h bbbb bbbb;

Since the simulator loads 64-bit data into the DATA LARs, it fetches the data beside the j and k also. The DATA LARs d4 and d5 are filled in this way:

d4 = 64'h dddd dddd dddd dddd;

d5 = 64'h bbbb bbbb bbbb bbbb;

Now consider the following cases:

Table 11: Register assignment in case of ambiguous alias of j and k in above example

ALIAS ANALYSIS	REGISTER ASSIGNMENT	
Compiler knows j == k	Share one register	
Compiler knows j != k	Two separate registers	
Compiler doesn't know??	**DATA LARs**	**OTHER RISCs**
	Doesn't happen. Since compiler has the address of j and k, only the above 2 cases occur in DATA LARs	Which of the above?

Therefore for the first case we just have a two fetch cycles for both j and k; one for fetching the address of the integer j and the next for fetching the data of j since we are dealing with pointers here. When the hardware sees the same effective address for j and k it will cancel the k's fetch cycle and feed the data fetched for j to k. The snapshots obtained by executing the instructions in table 9 are given below in figures 27 and 28 for the above two cases for vector add instruction.

For case II:

j + k = 64'h dddd dddd dddd dddd (packed word--- integer j)
   +   64'h bbbb bbbb bbbb bbbb   (packed word--- integer k)

   =   64'h 9999 9998 9999 9998 = d6 = i (packed word—integer)
----
i & k = 64'h 9999 9998 9999 9998 (packed word--- integer i)
   &   64'h bbbb bbbb bbbb bbbb (packed word--- integer k)

   =   64'h 9999 9998 9999 9998 = d5 = k (packed word integer)
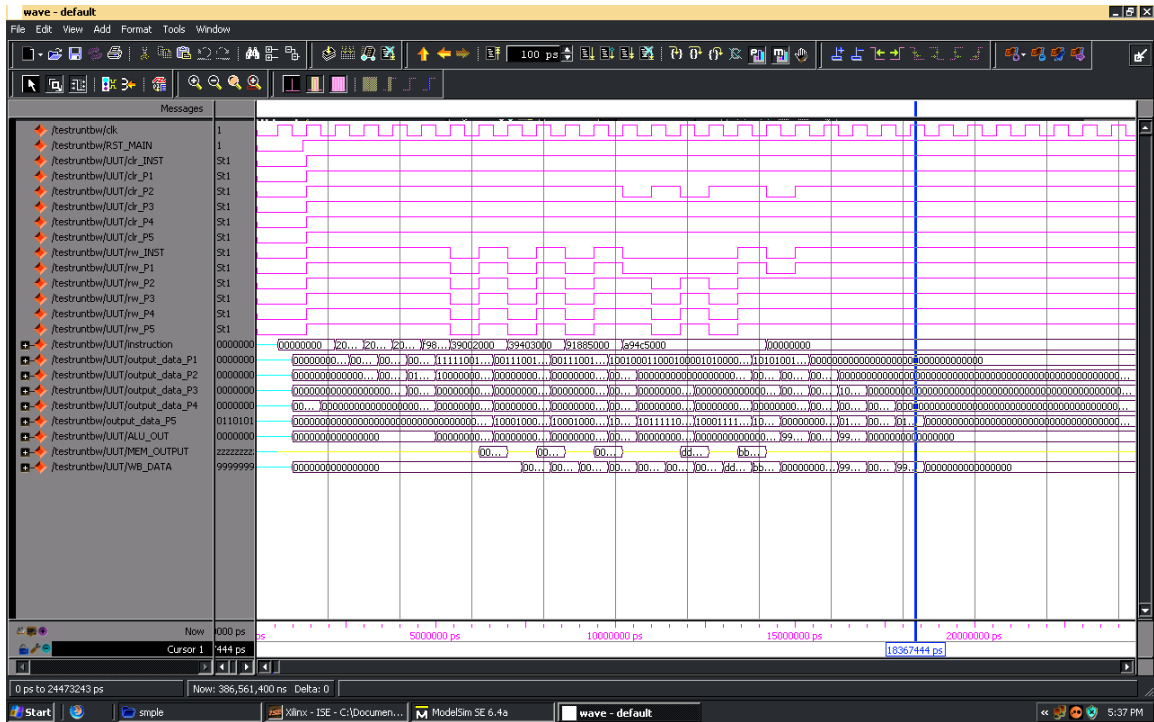---

Figure 27: the snapshot of alias analysis program run on the DATA LARs simulator with j != k

The waveform in the snapshot of figure 27 is a Modelsim wave output which shows the execution output of the instructions during each clock cycle. All the internal signals could be viewed at a certain moment during the execution of the program. ALU_OUT gives the output of the ALU during the 4th stage. MEM_OUT gives the memory output in case of a fetch cycle. As we can see in the snapshot, there are only 5 fetch cycles involved in the whole execution. WB_DATA gives the data to be written back into the DATA LARs after the whole execution process of an instruction.

The figure 28 explains the case where j is equal to k, i.e., they point to the same memory location. Here this case arises if the address of variable k is the same as variable j which is 18. This is obtained by changing the data at memory location 2 to 18 instead of 20 which could be seen in the table 10. From the snapshot it can be seen that cancelload_LAR is the signal that tells the controller for loads that it found the address it was searching for and there is no need to go to data memory to fetch this data. And the signal LOADCANCEL gives the data from the DATA LAR which has the effective address that the load instruction was going to use to fetch data. To support the LARs

44

concept here, it could be seen that there are only 4 memory cycles in this case which get the job done correctly.
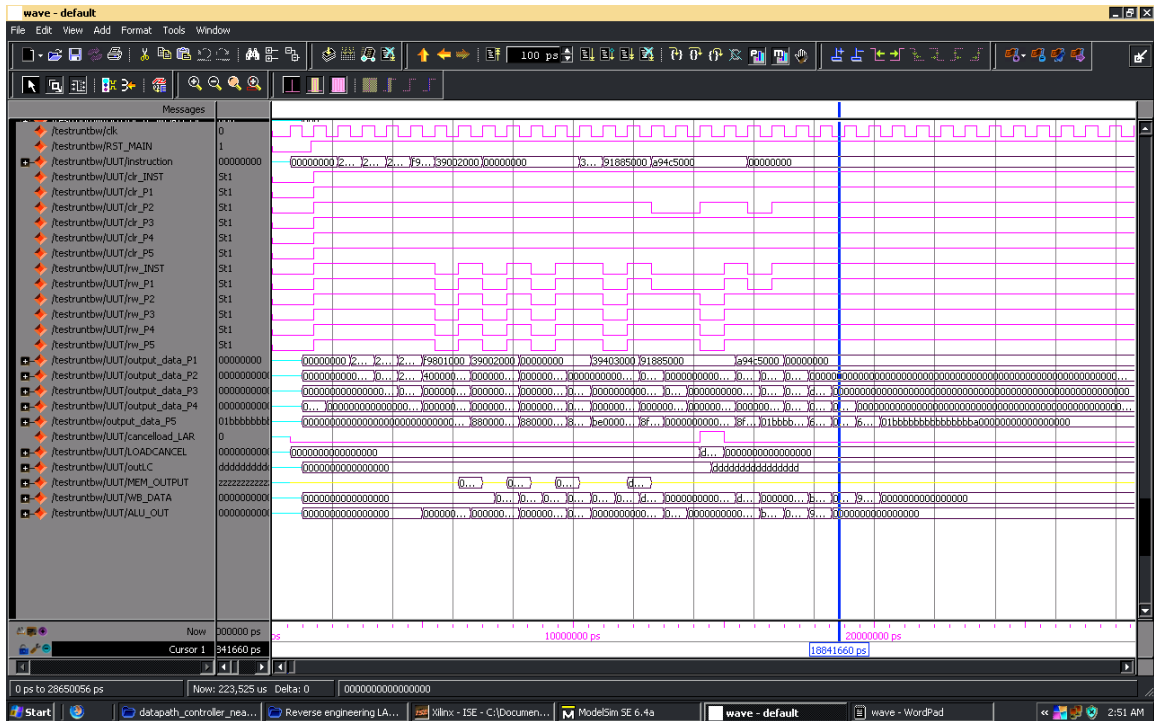
Figure 28: the snapshot of alias analysis program run on the DATA LARs simulator with j == k

By adding the ability to perform SIMD-like operations on fields within a register or datapath, DATA LARs operations replace a series of memory accesses and field extraction/insertion operations with a single access for a word's worth of fields. Compiler optimization methods like loop unrolling could be rewarding for architecture like DATA LARs which employs the SWAR like registers. A single instruction implies a lot of work here and each result is independent of previous result because compiler ensures that there are no dependencies. And there is high interleaved memory. The DATA LARs instruction set provides structured register accesses when compared to a single-issue scalar design where arbitrary register accesses adds to area and power.

45

5.1.2 EXECUTION OF LAZY STORE EXAMPLE ON DATA LARs SIMULATOR

The DATA LARs simulator designed using Verilog on the Xilinx platform gives the following results for the simple program shown in the table 11 below. The program loads the operands into d2, d3, d4 and d5 and performs ADDv operation on them. And then it loads other data from a different memory location into the same destination so that the previous data which is marked dirty would be evicted to the write buffer. It performs ADDv operation again with different operands and stores the result into the same destination which makes the DATA LAR d6 dirty again. Then another load into d6 would evict the data again to the write buffer.

Table 12: program to show lazy store mechanism of DATA LARs

INSTRUCTION	DISCRIPTION
LOADUB d2,d0,d0,10h	Data at memory location 2 which is 64'hdddddddddddddddddd would be fetched into DATA LAR d2 and its contents would be type tagged to unsigned bytes
LOADUB d3,d0,d0,20h	Data at memory location 4 which is 64'hbbbbbbbbbbbbbbbb would be fetched into DATA LAR d3 and type tagged to unsigned bytes
LOADUB d4,d0,d0,40h	Data at memory location 8 which is 64'h3333333333333333 would be fetched into DATA LAR d4 and type tagged to unsigned bytes
LOADUB d5,d0,d0,18h	Data at memory location 3 which is 64'hcccccccccccccccc would be fetched into DATA LAR d5 and type tagged to unsigned bytes
LOADDUMMY d6,d0,d0,78h	This is just a dummy load instruction to fill in the contents of address and type information to bytes
ADDv d6,d3,d2	This add vector instruction makes the 64-bit contents of d3 and d2 to pass through the conversion stage into the ALU where the packed bytes would be added to give a result of 64'h9898989898989898 and the dirty bit of d6 would be made high
LOADUB d6,d0,d0,28h	Data at memory location 5 which is 64'haaaaaaaaaaaaaaaa would be fetched into DATA LAR d6. But before writing this back at the write back stage of the processor the previous data which is marked dirty by the ADD instruction would be evicted to write buffer.

Table 13: program to show lazy store mechanism of DATA LARs (Contd...)

ADDv d6,d4,d3	The data at DATA LARs d4 and d3 would be added to give 64'heeeeeeeeeeeeeeeee which would replace 64'haaaaaaaaaaaaaaaaa and in the process make dirty bit of d6 high
LOADUB d6,d0,d0,30h	This would load 64'h1111111111111111 into d6 and tag it to unsigned bytes thereby evicting the previous contents again.

The MEM_OUTPUT and WB_DATA are the main outputs in the screenshot which show that correct values are been used in and out of the system. The MEM_OUTPUT gives the output from the main memory and the WB_DATA shows the executed data in the Write Back stage of the pipeline architecture before it is actually written back to the DATA LAR.

Figure 29: screen shot of DATA LARs simulator output for the program given above

5.2 DEVICE UTILIZATION SUMMARY

Table 14: Summary of the device utilization of the straw man's model of DATA LARs architecture

Number of BUFGMUXs	1
Number of External IOBs	215
Number of LOCed IOBs	0
Number of RAMB16s	1
Number of SLICEs	4251
Number of 4 input LUTs	6772

The current reduced-size synthesizable test core, which uses 8 64bit DATA LARs, synthesizes to an FPGA in approximately 7000 4-input LUTs, with a maximum clock frequency of 25.732MHz, prior to optimization.

CHAPTER 6

6. CONCLUSION AND FUTURE WORK

In order to find a perfect hardware structure which does not have the deficiencies of the present day memory hierarchies but all of their advantages Line Associative Registers (LARs) are derived from the concepts of SWAR, CRegs and type tagged designs. It is known that the present day cache technologies and levels are not sufficient to bridge the gap between the processors and the main memory speeds. It is found that the performance of the whole system can be improved by a certain amount by employing the concepts of SWAR and having wide lines of registers. The CRegs reduce the number of memory accesses required by spotting and updating the ambiguous aliasing automatically and thereby letting these values stay in registers much longer. This idea would be much more effective when added to SWAR-like processing where a single fetch cycle would mean fetching a whole line of data.

This thesis is about using the concepts of LARs and applying them to the CPU registers. These special registers are called DATA LARs since they target the data part of the processer, although they could be used on the instruction side of the processor also. An instruction set architecture suitable for the prototype was developed. A 6-stage pipelined architecture was designed using these DATA LARs using the hardware descriptive language Verilog. A qualitative evaluation of the designed architecture for different scenarios where it could reduce the number of fetch cycles has also been done. The results section of this thesis compares and elaborates the uses of DATA LARs design. The simulator accepts assemble language instructions from the instruction set of the DATA LARs and executes them. The outputs could be viewed comfortably with proper timing information.

The design lets the programmer operate on 64-bit data. So a floating point unit should be designed and patched to the existing design so that the simulator could target floating point operations. The instruction set should be tweaked accordingly. The performance evaluation results obtained by targeting the floating point data could be used to actually

compare this hardware simulator to the existing architectures out there with SWAR concepts integrated in them.

REFERENCES

[1] Liptay, j., "structural aspects of the system/360 model 85: part II: The Cache," IBM systems journal, 1968, pp. 15-21.

[2] C-H. Chi and H. G. Dietz, "Improving Cache Performance by Selective Cache Bypass," IEEE Proceedings of the *22nd Hawaii International Conference on Systems Sciences*, Architecture Track, vol. 1, pp. 256-265, January 1989.

[3] Hill, M.D., "Evaluation of on-chip cache," M.S. Thesis, University of California, Berkeley, December, 1983.

[4] Dietz, H.G., "The Refined-Language approach to compiling for parallel supercomputers," Ph.D. Thesis, Polytechnic University, June 1987.

[5] http://www.aggregate.org/TechPub/TARIQ/thesis.html

[6] Patterson, D.A., "Reduced Instruction Set Computers," Communications of the ACM, Volume 28, Number 1, January 1985, pp.8-21.

[7] Daniel Tabak, "Reduced Instruction Set Computer" RISC Architecture.

[8] Steve Heath, "Microprocessor Architectures" RISC, CISC and DSP.

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

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

[11] C.McNairy, D.Soltis, "Itanium 2 Processor Microarchitecture", IEEE Micro Vol. 23 Issue 2, pp.44-55, March 2003.

[12] Sites, R., "How to use 1000 Registers" Proceeding of the Caltech Conference on VLSI, January 1979, pp. 527-536.

[13] Steele, G.L. Jr., Sussman, G.J., "The Dream of a Lifetime: A Lazy Variable Extent Mechanism," ACM SIGPLAN, 1985, pp. 163-172.

[14] Eric Rotenberg and Steve Bennett and Jim Smith, Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching, In Proceedings of the 29th International Symposium on Microarchitecture, 1996, 24—34.

[15] G. Hinton, D. Sagar, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the Pentium 4 processor." Intel Technology J. Q1 2001.

[16] Ruby B. Lee. Subword Parallelism with MAX-2. IEEE Micro, 16(4):51-59, August 1996.

[17] Ruby Lee and Jerry Huck. 64-bit and multimedia extensions for the PA-RISC 2.0 architecture. In Proceedings of Compcon '96, Technologies for the Information Superhighway. Digest of Papers, 152-160, Los Alamitos, California, 1996. IEEE Computer Society Press.

[18] Intel Corporation. MMX technology overview. Technical report, Intel Corporation, February 1997.
Formally at http://developer.intel.com/drg/mmx/.

[19] Intel Corporation. Intel Architecture MMX technology: Programmer's reference manual. Technical report, Intel Corporation,
http://developer.intel.com/drg/mmx/Manuals/prm/prm_covr.htm, March 1996.

[20] Advanced Micro Devices, Inc. "AMD Extensions to the 3DNow! and MMX Instruction Sets Manual"
http://www.amd.com/us-en/Processors/DevelopWithAMD/0,,30_2252_739_1102%5E1144,00.html

[21] Bottlenecks in Multimedia Processing with SIMD Style Extensions and Architectural Enhancements, Deepu Talla, Member, IEEE, Lizy Kurian John, Senior Member, IEEE, and Doug Burger, Member, IEEE.

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

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

[24] Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3A: System Programming Guide, Part 1. Order Number: 253668-030US, March 2009.

[25] Weatherford, James R., Kimmel, Arthur T., Wallach, Steven J., "Cache store bypass for computer", U. S. patent 4,942,518. 17 July, 1990.

VITA

Kalyan Ponnala

Born: August 15th, 1985

Hyderabad, Andhra Pradesh, India

Education

B.S in Electrical and Electronics Engineering, Muffakham Jah College of Engineering and Technology, Osmania University, 2006