



# Instituto Politécnico Nacional

---

Centro de Investigación en Computación  
Laboratorio de Microtecnología y Sistemas Embebidos

Design of a Load/Store Queue with Out-of-Order  
Execution

## TESIS

Que para obtener el grado de:

Maestría en Ciencias en Ingeniería de Cómputo con opción en  
Sistemas Digitales

P R E S E N T A :

Ing. Abraham Josafat Ruiz Ramírez

Directores de Tesis:

Dr. Marco Antonio Ramírez Salinas

Dr. Adrián Cristal Kestelman



Centro de Investigación  
en Computación

Enero 2016



# INSTITUTO POLITÉCNICO NACIONAL SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

## ACTA DE REVISIÓN DE TESIS

En la Ciudad de México, D.F. siendo las 14:00 horas del día 8 del mes de enero de 2016 se reunieron los miembros de la Comisión Revisora de la Tesis, designada por el Colegio de Profesores de Estudios de Posgrado e Investigación del:

**Centro de Investigación en Computación**

para examinar la tesis titulada:

**“Design of a Load/Store Queue with Out-of-Order Execution”**

Presentada por el alumno(a):

<b>Ruiz</b> Apellido paterno	<b>Ramírez</b> Apellido materno	<b>Abraham Josafat</b> Nombre(s)
		Con registro: <b>B 1 3 0 0 9 1</b>

aspirante de: **MAESTRÍA EN CIENCIAS EN INGENIERÍA DE CÓMPUTO CON OPCIÓN EN SISTEMAS DIGITALES**

Después de intercambiar opiniones los miembros de la Comisión manifestaron **APROBAR LA TESIS**, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

### LA COMISIÓN REVISORA

Directores de Tesis

Dr. Marco Antonio Ramírez Salinas

Dr. Adrián Cristal Kestelman

Dr. Luis Alfonso Villa Vargas

Dr. Herón Molina Lozano

M. en C. Osvaldo Espinosa Sosa

Dr. Víctor Hugo Ponce Ponce

INSTITUTO POLITÉCNICO NACIONAL  
CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN  
DIRECCIÓN

Dr. Luis Alfonso Villa Vargas




**INSTITUTO POLITÉCNICO NACIONAL**  
**SECRETARÍA DE INVESTIGACIÓN Y POSGRADO**

*CARTA CESIÓN DE DERECHOS*

En la Ciudad de México, D.F. el día 8 del mes Enero del año 2016, el (la) que suscribe Abraham Josafat Ruiz Ramírez alumno (a) del Programa de Maestría en Ciencias en Ingeniería de Cómputo con opción en Sistemas Digitales con número de registro B130091, adscrito a Centro de Investigación en Computación, manifiesta que es autor (a) intelectual del presente trabajo de Tesis bajo la dirección de Marco Antonio Ramírez Salinas y Adrián Cristal Kestelman y cede los derechos del trabajo intitulado Design of a Load/Store Queue with Out-of-Order Execution, al Instituto Politécnico Nacional para su difusión, con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y/o director del trabajo. Este puede ser obtenido escribiendo a la siguiente dirección abraham.ruiz1990@yahoo.com. Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

  
Abraham Josafat Ruiz Ramírez

Nombre y firma

# Resumen

*El procesador súper-escalar trata de explotar el paralelismo a nivel de instrucción (ILP) presente en el código; la clave es ejecutar la mayor cantidad de instrucciones por cada ciclo de reloj (IPC). Para alcanzar esta meta es necesario implementar algunas técnicas de planificación dinámica para identificar instrucciones en vuelo que no tengan dependencia de datos entre ellas y que puedan ser ejecutadas en paralelo, técnicas como Predicción de Saltos, Renombrado de registros, Ejecución fuera de orden, así como la implementación de colas de instrucciones clasificadas por tipo de instrucción (enteros, punto flotante, acceso a memoria, vectoriales, etc.). Las instrucciones de acceso a memoria están divididas en dos categorías, “Load” y “Store”, y tienen acceso directo a la memoria cache de datos de nivel uno; a causa de que las direcciones de memoria son calculadas hasta la etapa de ejecución (una vez emitida la instrucción a la unidad de generación de direcciones), es necesario tener un mayor control sobre las dependencias entre las instrucciones de acceso a memoria en vuelo, de otra forma la ejecución del programa producirá errores.*

*El objetivo de esta tesis es el diseño e implementación de la Cola de instrucciones de acceso a memoria para un procesador RISC súper-escalar con ejecución fuera de orden, así como implementar un diseño eficiente para la técnica de desambiguación de direcciones de memoria.*



# Abstract

*Superscalar processors exploit the instruction level parallelism (ILP) present in the code; the key is to execute the maximum amount of instructions per clock cycle (IPC). In order to reach this goal are needed to implement several schedule techniques to discover in-Flight instructions without data dependencies that can be executed in parallel inside of the execution window, techniques such as Branch prediction, Register rename, and out-of-order execution as well as to implement special execution engines classified by their kind of instruction (integer, floating point, memory access, vector, etc.). Memory access instructions are split into two categories, Load, and Store instructions and they have direct access with the L1 Cache Memory; because the memory addresses are computed in the execution stage, it is needed to have a greater control with the dependencies between the In-Flight instructions, otherwise the executed program will have failures.*

*The objective of this thesis is the design and implementation of the Load Store Queue for a Superscalar RISC processor with an out-of-order execution and to design an efficient memory addresses disambiguation technique.*

# Acknowledgments

I'm glad to say that this thesis wouldn't be achieved without the help and motivation of several people, friends and family. First of all, I'd like to thank my advisors Marco Antonio Ramírez Salinas and Adrián Cristal Kestelman, who guided me through the achievement of this thesis project, also thanking their motivation and patience.

For all my friends that provided me so much support in this stage of my life, I'm really happy to have you in my life... Cesar Cervantes Cazales, you're like that brother who I can always rely on, thank you for all your faith in me, for always telling me that I'm able to do whatever I aim to do in my life, I really hope you stay as my best friend for a really long time. Elsa Dorantes Merino, thank you for all those nights which I needed to have an ear for my heart and concerns, as one of my best friends I'm glad to have you by my side whenever something shows up. Lidia Rico García, although we don't have too much time since we know each other, I'm so glad that our life-paths crossed each other, you have made me to trust in myself again, something that I lost a few years ago, and that has helped me in such a grateful way with the realization of my thesis, thank you so much for entering my life, you're amazing. Laura Fabila, you're one of that friends who tells the very truth of things, even though it hurts sometimes, it is so helpful and meaningful, everytime I talk with you, I discover something else of myself, please don't leave. Shadia Hernandez Andrade, you moved so much things in my life that I'm so thankful of, even

though we have moved apart from each other's life, you'll always be part of this life event of mine, I'll always remind you with dearness. I want to say thank you to all my partners who made of this life-stage of mine easier to handle, Cristobal Ramírez Lazo, thank you for your company it that year in Barcelona. Cesar Hernandez, you have been my teacher, my friend and now my brother, thank you for all you advise and friendship through the years, I'm happy to think that we'll be working as teammates for a long time. Also, I'd like to say sorry for my friends that I forgot to name in this Acknowledgments page, but that doesn't mean that I'm not thankful with all of you, without you I may be lost sleeping beneath a bridge by now, thank you so much.

To my confident, my best friend, my life partner and sister Cintia Elisa Ruiz Ramírez, you're my favourite person in the world, thank you for your company and love, I'm really proud of you and so happy that I have been a good brother/father/friend for you, I really love you. To my precious mom, María de Jesús Ramírez Huerta, thank you so so much for your patience and love, I'm the product of your hardworking as an excellent mother, you really inspired me in so many levels that I can't imagine what I should be without you, thank you for always taking care of us, you're a beautiful person-model to follow, I love you. To my dad... You made of me a good person, my whole will is dedicated to you...

# Contents

<b>List of Figures .....</b>	<b>vii</b>
<b>List of Tables .....</b>	<b>x</b>
<b>Chapter 1 – Introduction .....</b>	<b>1</b>
1.1    Problematic Approach.....	2
1.2    Justification .....	3
1.3    Considerations.....	4
1.4    Objectives.....	4
1.4.1    Main objective.....	4
1.4.2    Specific objectives .....	4
1.5    Scope of this thesis project and contributions .....	5
1.6    Organization of this thesis.....	5
<b>Chapter 2 – Background .....</b>	<b>6</b>
2.1    RISC and CISC architectures.....	7
2.2    General concepts of a memory access instruction.....	9
2.3    Superscalar Processor .....	11
2.3.1    Dependencies .....	13
2.3.2    Main pipeline stages of a superscalar processor.....	14
2.3.3    Address, Load and Store Queues.....	26
2.4    Cache Memory and Memory Hierarchy .....	28
2.4.1    Cache Memory Parameters.....	31
2.5    Tag and Data Array Accesses.....	36
2.5.1    Parallel Tag and Data Array Access.....	37
2.5.2    Serial Tag and Data Array Access.....	38
<b>Chapter 3 – State of the Art .....</b>	<b>39</b>
3.1    Research Proposals for LSQ & Memory Disambiguation .....	41

3.1.1	TRIPS.....	41
3.1.2	TRIPS (Unordered, Late-Binding LSQ Design) .....	44
3.1.3	Store Sets.....	46
3.1.4	Store Vectors .....	47
3.1.5	Store-to-Load Forwarding via Store Queue index Prediction.....	49
3.1.6	Address Indexed Memory Disambiguation and Store-to-Load Forwarding .....	50
<b>Chapter 4 – Proposed Load/Store Queue Design .....</b>		<b>52</b>
4.1	Load/Store Unit with In-Order Execution.....	52
4.1.1	In-Order Address Computation Pipeline .....	53
4.1.2	In-Order Memory Access Pipeline .....	55
4.2	Load/Store Unit with Out-of-Order Execution .....	58
4.2.1	Unknown/Known Memory Address, Block Mapping Table & Address Generation .	59
4.2.2	Out-of-Order Select Logic.....	63
4.2.3	Memory Disambiguation & Store-to-Load Data Forwarding Mechanism .....	65
4.2.5	Store-to-Load Forwarding Mechanism and Delayed Source Data Forwarding .....	70
<b>Chapter 5 – Results.....</b>		<b>72</b>
<b>Chapter 6 – Conclusions and Future Work .....</b>		<b>82</b>
6.1	Conclusions.....	82
6.2	Future Work .....	83
<b>Appendix-A .....</b>		<b>84</b>
<b>Appendix-B .....</b>		<b>90</b>
<b>References .....</b>		<b>a</b>

# List of Figures

<i>Figure 2-1. Core and its interfaces with caches</i> .....	7
<i>Figure 2-2. x86 Instruction fields</i> .....	8
<i>Figure 2-3. CISC architecture</i> .....	8
<i>Figure 2-4. RISC Instruction fields</i> .....	9
<i>Figure 2-5. General memory access instruction data path</i> .....	10
<i>Figure 2-6. Scalar vs Superscalar Execution in Pipeline Processors</i> .....	12
<i>Figure 2-7. Superscalar Processor Architecture</i> .....	13
<i>Figure 2-8. True and False Data Dependencies</i> .....	14
<i>Figure 2-9. Fetch stage</i> .....	15
<i>Figure 2-10. Decoder ROMs</i> .....	16
<i>Figure 2-11. Typical RISC decode stage</i> .....	17
<i>Figure 2-12. Register renaming example</i> .....	18
<i>Figure 2-13. Register Renaming building blocks</i> .....	19
<i>Figure 2-14. Reorder Buffer</i> .....	20
<i>Figure 2-15. Instruction's tracking inside the ROB</i> .....	21
<i>Figure 2-16. Recovery Logic in Front-End and Back-End</i> .....	21
<i>Figure 2-17. Dispatch Logic</i> .....	22
<i>Figure 2-18. INT and FP Queues with Mapper</i> .....	22
<i>Figure 2-19. Organization of the CAM Section in the Instruction Queue</i> .....	23
<i>Figure 2-20. Two-Issue INT Queue's Payload RAM Logic</i> .....	24
<i>Figure 2-21. Timing of the wake-up signal to support back-to-back execution</i> .....	25
<i>Figure 2-22. An execution engine with two functional units, without (left) and with (right) value bypassing</i> .....	26
<i>Figure 2-23. Recycling Physical Registers at Commit Time</i> .....	26
<i>Figure 2-24. Conventional LSQ design</i> .....	28
<i>Figure 2-25. SRAM cell</i> .....	29
<i>Figure 2-26. DRAM cell</i> .....	29
<i>Figure 2-27. The Cache Memory concept</i> .....	30
<i>Figure 2-28. Memory hierarchy</i> .....	31
<i>Figure 2-29. Fully-Associative Cache Organization</i> .....	34
<i>Figure 2-30. Direct-Mapped Cache Organization</i> .....	35
<i>Figure 2-31. Two-Way Set-Associative Cache Organization</i> .....	36
<i>Figure 2-32. Parallel Tag and Data Array Access</i> .....	38
<i>Figure 2-33. Serial Tag and Data Array Access</i> .....	38
<i>Figure 3-1. TRIPS prototype microarchitecture</i> .....	42
<i>Figure 3-2. TRIPS Processor's Data Tile</i> .....	43
<i>Figure 3-3. The Age-Indexed LSQ</i> .....	44

Figure 3-4. The ULB-LSQ Microarchitecture .....	45
Figure 3-5. BFP Search Filtering.....	46
Figure 3-6. Implementation of Store Sets Memory Dependence Prediction.....	47
Figure 3-7. (a) Store-address tracking of dependencies, and (b) Age-tracking of dependencies .....	48
Figure 3-8. Store vectors data structures and interaction with a conventional load queue.....	48
Figure 3-9. Example store vectors operation.....	49
Figure 3-10. Store queues: (a) associative, (b) indexed .....	50
Figure 3-11. Processor pipeline, store forwarding cache (SFC), memory disambiguation table (MDT) and Store FIFO .....	51
Figure 4-1. Lagarto II Microarchitecture .....	53
Figure 4-2. Ready Bit Assignment Example .....	54
Figure 4-3. Head Pointer and Tail Pointer .....	55
Figure 4-4. Select Logic and Address Generation Unit.....	55
Figure 4-5. Forward Logic Example .....	56
Figure 4-6. Memory Access (Load) .....	57
Figure 4-7. Memory Access (Store).....	57
Figure 4-8. Proposed LSQ General Pipeline .....	59
Figure 4-9. Block Mapping Table Functionality Example.....	60
Figure 4-10. Address Queue (inside the L/S Unit) & the Block Mapping Table.....	62
Figure 4-11. Queue's Ready Bit Logic .....	63
Figure 4-12. Leading Zero Counter as a Priority Selector (oldest-first) – 32bits .....	64
Figure 4-13. Non/Possible dependencies vector generation .....	65
Figure 4-14. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (a).....	66
Figure 4-15. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (b).....	66
Figure 4-16. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (c) .....	67
Figure 4-17. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (d).....	67
Figure 4-18. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (e).....	67
Figure 4-19. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (f).....	68
Figure 4-20. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (g).....	68
Figure 4-21. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (h).....	68
Figure 4-22. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (i).....	69
Figure 4-23. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (j).....	69
Figure 4-24. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (k).....	69
Figure 4-25. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (l).....	70
Figure 5-1. Maximum Overall Frequency .....	72
Figure 5-2. Decoder .....	73
Figure 5-3. Integer Ready Bit Register - Test 1 .....	74

<i>Figure 5-4. Integer Ready Bit Register - Test 2</i> .....	75
<i>Figure 5-5. Floating Point Ready Bit Register - Test</i> .....	76
<i>Figure 5-6. Pointer Control Logic - Test 1</i> .....	77
<i>Figure 5-7. Recovery Mechanism</i> .....	77
<i>Figure 5-8. Selection Logic - Test 1</i> .....	77
<i>Figure 5-9. Selection Logic - Test 2</i> .....	78
<i>Figure 5-10. Selection Logic - Test 3</i> .....	78
<i>Figure 5-11. Address Queue's Write/Read Test</i> .....	78
<i>Figure 5-12. Block Mapping Table Test</i> .....	79
<i>Figure 5-13. Possible dependencies vector - Desk evaluation 1</i> .....	79
<i>Figure 5-14. Possible Dependencies Vector - Test 1</i> .....	80
<i>Figure 5-15. Possible dependencies vector - Desk evaluation 2</i> .....	80
<i>Figure 5-16. Possible Dependencies Vector - Test 2</i> .....	80
<i>Figure 5-17. Store-to-Load Data Forwarding - Test</i> .....	81
<i>Figure 7-1. Front-End</i> .....	85
<i>Figure 7-2. Store-to-Load Forwarding Logic including the Delayed Source Data</i> .....	86
<i>Figure 7-3. Load/Store Queue with In-Order Execution</i> .....	87
<i>Figure 7-4. Incorporation of the proposed Load/Store Unit to the processor</i> .....	88
<i>Figure 7-5. Select Logic, Address Generation &amp; PDVs pipeline</i> .....	89



# List of Tables

<i>Table 1. Memory disambiguation schemes</i>	27
<i>Table 2. Cache memory parameters</i>	32
<i>Table 3. Memory Address Fields</i>	37
<i>Table 4. Load Execution Scenarios (X=Don't care)</i>	43
<i>Table 5. Entries Organization inside the Queue</i>	61
<i>Table 6. 4 Blocks-RAM/CAM write enables and multiplexers between two input instructions</i>	61

# “Design of a Load/Store Queue with Out-of-Order Execution”

*by*

Abraham Josafat Ruiz Ramírez

A thesis submitted in partial fulfillment of the requirements for the degree of

Maestría en Ciencias en Ingeniería de Cómputo

*and*

Master in Innovation and Research in Informatics

*at the*

“Instituto Politécnico Nacional”

Centro de Investigación en Computación – Mexico

*and*

“Universitat Politècnica de Catalunya”

Facultat d’Informàtica de Barcelona – Spain

January 2016

*under the supervision of*

PhD. Marco Antonio Ramírez Salinas

PhD. Adrián Cristal Kestelman

---

# Chapter 1

---

## INTRODUCTION

---

Lagarto II, is a Processor architecture still in development by some students, researchers, and lecturers from the Microtechnology and Embedded Systems research group (MICROSE) which belongs to the *Centro de Investigación en Computación* of the *Instituto Politécnico Nacional* of Mexico. The objective of this MICROSE's project is to design a 64-bits Superscalar RISC processor with a dynamic scheduling and out-of-order execution. The design is modeled at RTL level in HDL code using some EDA tools, such as Mentor Graphics ModelSim and Altera Quartus II.

Though most of the Lagarto II processor's design is already finished, it still does not have a working Load Store Queue, so this thesis project will be the first design of its corresponding memory access building block.

The memory access engine in a superscalar processor is commonly separated into three functional blocks (address queue, load queue, and store queue), which houses some specific logic for memory access instructions. The Address Queue: the issuing and computation of memory addresses, the Load Queue: the logic to read data from the L1 Cache Memory and

to write this data to the Register Files, and finally, the Store Queue: the logic to read from the Register Files and to write this data to the L1 Cache Memory.

Because a superscalar processor executes the instructions in an out-of-order fashion, the identification of data hazards created by the dependencies between the in-flight LOAD and STORE instructions becomes a necessity. So, a memory disambiguation technique is used, which checks the ordering age and memory address of each of the issued instructions inside both the Load and Store queues, it also does some data forwarding for those instructions that have the same memory address (taking care of their age ordering inside the queue).

The out-of-order execution takes advantage of the speculation technique, permitting an early execution of instructions that do not have a dependency with younger instructions. Forwarding some data, such as a Load that (speculatively) has the same memory address of an earlier Store, the data that would be written to the L1 Cache Memory by that Store instruction can be forwarded to the dependent Load instruction, thus, completing it beforehand.

In order to do this speculative execution, a predictor must be used. There are two major kinds of predictors, those that use a “naive” or “blind” prediction which always says that the Load instructions don't have any dependency with earlier Stores, and those that are “dynamic” which update their dependencies tables as long as there are wrong speculations, so, later with these dependencies tables updated a better decision would be made.

---

## 1.1 PROBLEMATIC APPROACH

The problematic appointed in this thesis is to design a Load-Store queue able to execute the memory access instructions in an out-of-order fashion. The design of this complete queue will be simulated in Mentor Graphics ModelSim software and evaluated in an FPGA board. For a first approach, the design will be only a simple Load Store Queue, without a predictor but with an age ordering memory disambiguation, the design will be able to support the store-to-load data forwarding logic as well as some energy saving techniques.

This store-to-load data forwarding can be done whenever there is a Load instruction with the same memory address as an earlier Store instruction in the queue (the memory address

isn't known until execution stage) but in order to achieve a high performance IPC, these memory access instructions must be executed in an out-of-order manner, thus, the memory disambiguation must be included in the design in order to ensure their correct execution.

---

## 1.2 JUSTIFICATION

Nowadays in Mexico there is a huge foreign technological dependence, this dependence is reflected in obsolete solutions purchased at high costs in areas as health, food, education, energy and security. But most important is the knowledge dependence, most of the intellectual property belong to EEUU and this carries big security issues as well as low-end given technology.

Because of that, some countries such as China have broken this barrier and fabricated their own trustworthy processors, they also promoted the foreign technology investment in their country and nowadays they have acquired a great amount of IT fabs as well as "Research and Development" (R&D) centres.

The first Chinese processor version was called "Loongson 1", a 32bit MIPS compatible processor and it was working at 266MHz, later on, they achieved a second processor called "Loongson 2", it was a 64bit superscalar processor with out-of-order execution working at 1GHz, and it also included a graphic accelerator, after this approach, they continued with multicore technology achieving in 2009 an 8-core multiprocessor working at 1GHz, subsequently, and in 2011 they built a 6 and 8-core multiprocessors working at 1.2GHz. With this effort they created their own low-cost PCs, servers, High-performance PCs, industrial control equipment, and most important, their national security applications.

Taking into account this important fact, the "Lagarto II" will be the first Mexican superscalar processor, and it will carry big efforts in the national security area, as well as trying to diminish this actual technology dependence; it is cleared that the goal for this big project is inspired by the mentioned Chinese success and the design of this processor's block, the Load Store Queue, will enforce the national technological advance.

---

## 1.3 CONSIDERATIONS

In a high-performance superscalar processor, the energy consumption is a really important challenge to solve, thus, there are a lot of energy saving techniques included in vendor's processors, but these techniques are kept in patents so that they can't be used by another processor vendor. A sort of important processor's building blocks that should be using these power saving techniques are the Register Renaming stage, the Wake-Up logic in the Instruction Queues (integer, floating point, memory access) and the memory disambiguation for the memory accesses executed in an out-of-order fashion. As I have already stated, the energy saving in a high-performance superscalar processor is a very important fact to achieve, thus, this thesis is also focused in these energy saving techniques, such as disabling some unnecessary comparisons done by every entry on each of the queues included in the LS Queue (address queue, load queue, and store queue).

For research and educational purposes, the Lagarto II architecture uses the MIPS64 Instruction Set Architecture, thus, to evaluate this design the MIPS64 ISA's memory access instructions will be used.

---

## 1.4 OBJECTIVES

### 1.4.1 MAIN OBJECTIVE

---

To design and implement a Load-Store Queue for a superscalar processor with out-of-order execution and to design its memory disambiguation method using power saving techniques.

### 1.4.2 SPECIFIC OBJECTIVES

---

- I. To define the requirements for the memory access instructions execution.
- II. To identify the different kinds of memory access instructions, to design a digital logic to decode those instructions and generate the necessary control signals.

- III. To design the logic for the different blocks included in the Load Store engine (Address Queue, Load Queue, and Store Queue) and the selection logic in order to execute the memory access instructions in an out-of-order fashion.
- IV. To design a memory disambiguation scheme and to do a desk evaluation for every combination between three consecutive memory access instructions (Load or Store), results must be presented.
- V. To use microarchitectural energy saving techniques, trying not to decrease the overall performance.

---

## 1.5 SCOPE OF THIS THESIS PROJECT AND CONTRIBUTIONS

At the end of this thesis, the Lagarto II processor will feature a Load-Store queue able to receive two instructions per cycle and to issue one instruction per cycle in an out-of-order fashion, this will be approachable with the help of the memory disambiguation scheme designed, also, it will have a reduced power consumption taking into account a previous Instruction queue design [1], this novel design will be adapted to the Address queue, and it motivated the design for both the Load queue and Store queue.

---

## 1.6 ORGANIZATION OF THIS THESIS

This thesis is organized as follows, in the second chapter, all the background information needed to explain this thesis approach is described, as well as how the processor works and the functionality of its main building blocks, it is also explained some terminology. In the third chapter, several papers and research proposals in the state of the art for the Load/Store Queue and Memory Disambiguation are described. Then, the proposed design for the Load/Store Queue is explained in the fourth chapter presenting some results in the fifth chapter. Conclusions are stated in the last chapter.

---

# Chapter 2

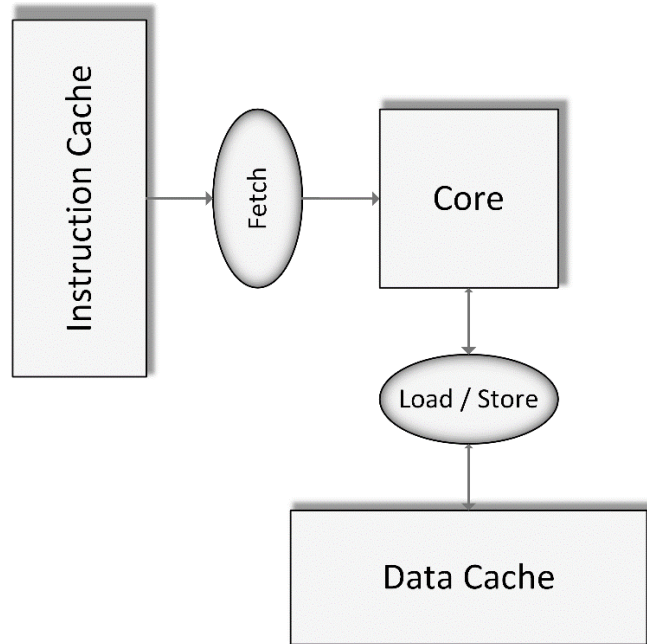
---

## BACKGROUND

---

Programs are normally written in a high-level programming language, such as C and C++, so these programs need to be transferred to a specific machine language using their corresponding compiler tool (such as ICC or GCC) configured for that specific microarchitecture. This code that is already in machine language represents the instructions that the processor can understand [2] [3], these instructions are read from instruction cache memory (via the Fetch mechanism) [4] and executed by that processor (making use of the data cache memory) (*Figure 2-1*). It is clear that every architecture (such as x86, MIPS, Alpha, etc.) has its own Instruction Set Architecture (ISA) which represents the way that the instructions are decoded and executed by that processor, but there are two major kinds of ISA, the Complex Instruction Set Computing (CISC) and the Reduced Instruction Set Computing (RISC), both of them with their own advantages and disadvantages.





*Figure 2-1. Core and its interfaces with caches*

## 2.1 RISC AND CISC ARCHITECTURES

Instructions of RISC ISA are smaller and simpler than CISC ISA as well as easier to decode because the length of their instructions is the same, this feature also requires a bigger amount of instructions in order to do the same job as a fewer CISC instructions, this is because the complex instructions do not have the same length for every instruction (normally vary from 1 byte to 15 bytes) nor have the same structure. The CISC ISA instructions are more specialized (*Figure 2-2*) [5]. Normally, the CISC processors have a specialized Decoder which transforms each instruction into several simpler RISC instructions with the help of a “Micro-Code ROM” (*Figure 2-3*) [6].

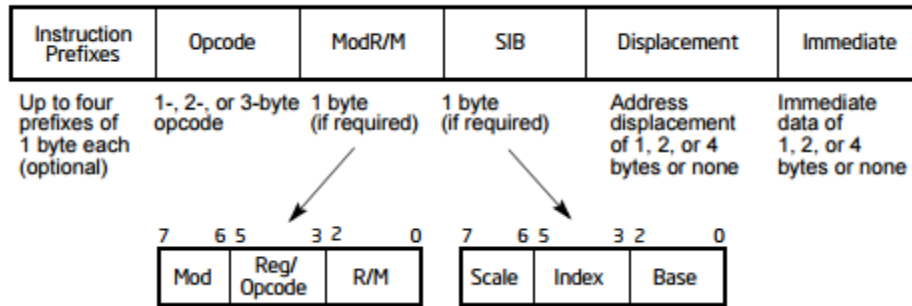


Figure 2-2. x86 Instruction fields

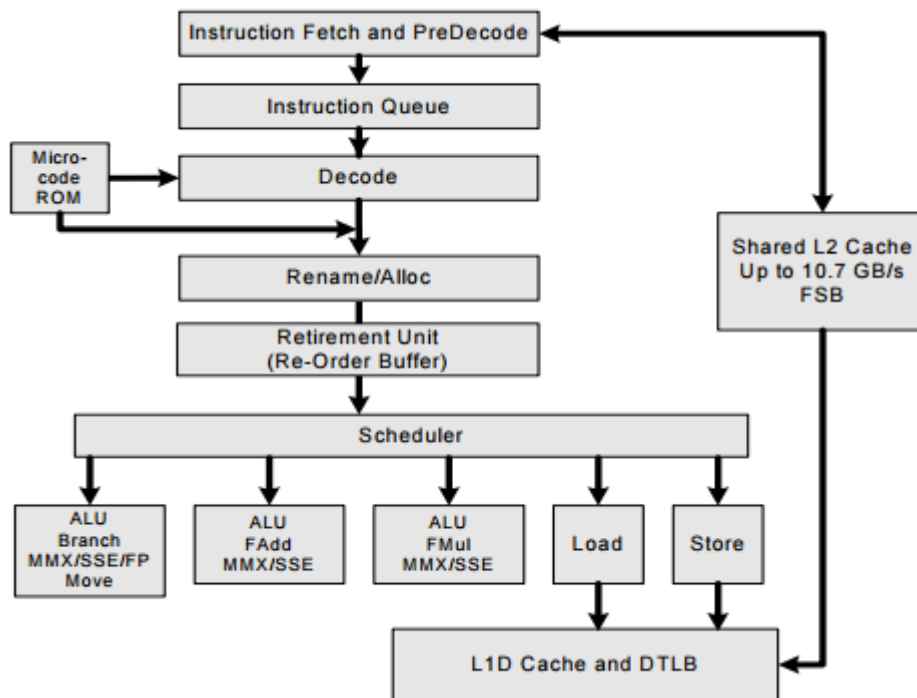


Figure 2-3. CISC architecture

In another hand, the RISC architectures have a few instruction formats, support a few addressing memory modes, and lack of instructions that operate directly on memory operands. The most common fields in the RISC architectures instructions are the “base register”, “destination register”, “source register”, “immediate value”, “function”, “opcode” and “index (in some architectures)”. The three main instruction encodings in the RISC ISA are: The “Register Encoding (R-Instruction)”, the “Immediate Encoding (I-Instruction)” which has a 16-bit (18-bit with 2-bit left shift) immediate signed value, and the “Jump Encoding (J-Instruction)” which has a 26-bit immediate value used to calculate the branch target address in order to jump unconditionally. In *Figure 2-4* there are shown these RISC instruction encodings.

<i>R-Instruction</i>					
OPCODE	BASE REGISTER	SOURCE REGISTER	DESTINATION REGISTER	SHIFT AMOUNT	FUNCTION
31-26	25-21	20-16	15-11	10-6	5-0

<i>I-Instruction</i>			
OPCODE	BASE REGISTER	DESTINATION REGISTER	IMMEDIATE VALUE
31-26	25-21	20-16	15-0

<i>J-Instruction</i>	
OPCODE	OFFSET
31-26	25-0

*Figure 2-4. RISC Instruction fields*

Memory access instructions (Load or Store) use the I-Instruction encoding, [20:26] bits encode the destination register for Load Instructions or the source register for Store Instructions, [25:21] bits encode the base register, and [15:0] encode a 16-bit offset.

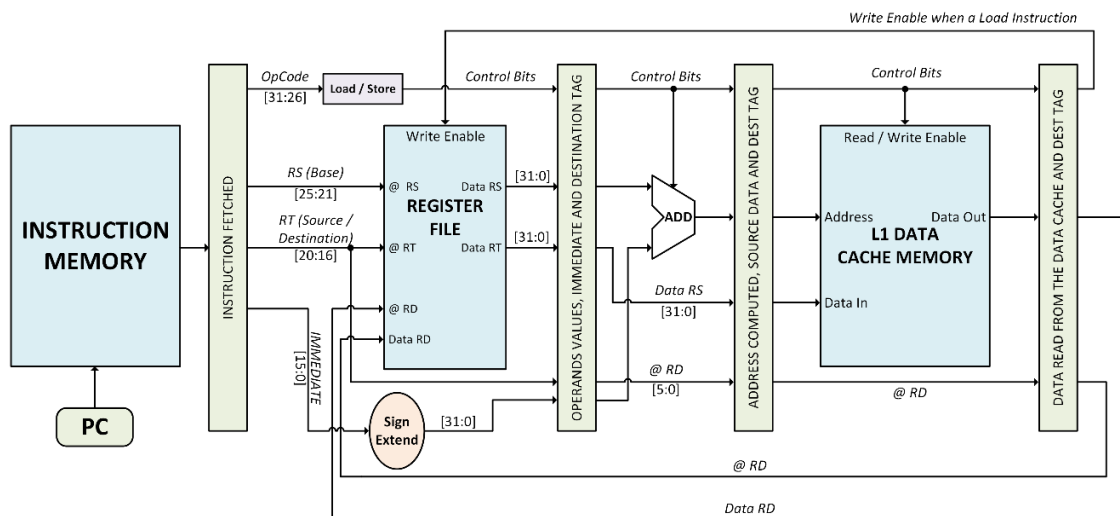
```
LW $t0, 0x0200($gp)  # Load Word Instruction
                       - Load word from $gp + 0x0200 to $t0

SW $t1, 0x0204($gp)  # Store Word Instruction
                       - Store word from $t1 to $gp + 0x0204
```

## 2.2 GENERAL CONCEPTS OF A MEMORY ACCESS INSTRUCTION

This subsection describes a general execution process of a memory access instruction in a scalar processor in order to understand how the memory address is generated and either a read or write operation to the L1 Data Cache Memory is accomplished. Then, next subsection describes same instruction execution in a superscalar processor model and the utility of the Load Store Queue.

*Figure 2-5* shows the basic data path of a memory access instruction in a scalar processor. In the first stage, the Load / Store instruction is read from the Instruction Memory addressed by the Program Counter, later on, this instruction is decoded and identified as either a Load or a Store so that the control bits can be generated, in each of the memory access instructions, there are two register tag fields, the Base Register (RS) and the Source or Destination Register (RT), in both cases (Load and Store) the memory address is generated by adding the Data from the Base Register and the Immediate value sign extended (this memory address is either for Reading or Writing the L1 Data Cache Memory) and the RT register tag is used to read the source data that will be written to the L1 Cache Memory (in a store instruction) or to write the data read from the L1 Cache Memory to the Register File (in a load instruction), the second stage is for reading the Register File with these register tags and for sign extending the immediate value, the third stage is the execution stage, where the memory address is computed by adding the base register and the immediate value sign extended, the fourth stage is the Memory Access stage, here the L1 Data Cache Memory is either read or written, and the last stage, the Write Back stage, is only used where there is a Load instruction executed, here the Data read from the L1 Data Cache Memory is written to the Register File addressed by the Destination Register Tag, which had flowed through the stages as the instruction was being executed.



*Figure 2-5. General memory access instruction data path*

---

## 2.3 SUPERSCALAR PROCESSOR

The main purpose or goal of a superscalar processor is to exploit the parallelism within the instructions stream, it means that whenever there are independent instructions, they should be executed in parallel, and of course, there must be extra careful with those instructions that are dependent.

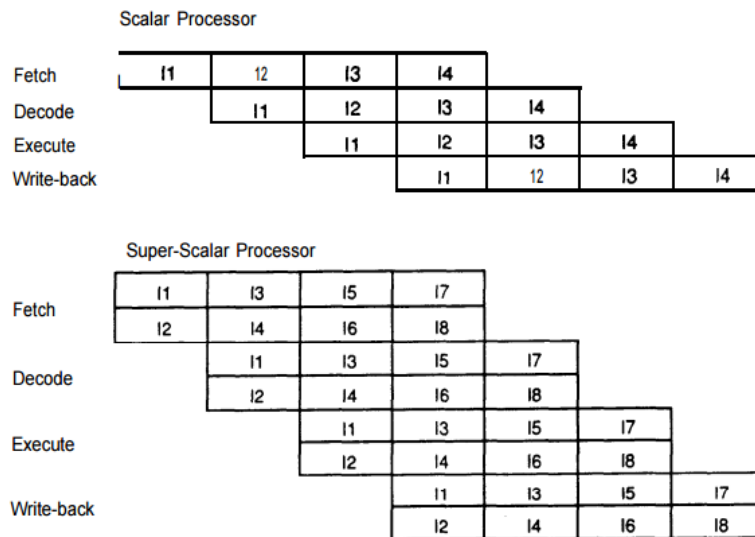
There exist two implementations of superscalar processors regarding the way they execute their instructions, the one with an in-order execution and the other one with an out-of-order execution, due to the scope of this thesis, the in-order scheme won't be explained but the Superscalar Processor with out-of-order execution. It is divided into two main blocks, called the Back-End and the Front-End, the main difference between these blocks is that in the Front-End the instructions are flowing in order (sequentially) through its stages (Fetch, Decode, Rename, and Dispatch), whereas in the Back-End, the instructions can flow in an out-of-order fashion (Issue, Execution, and Write Back), except for the Commit stage, which is always executed in order ensuring correctness in the program execution. In the Front-End, the instructions are flowing sequentially through the stages, but after they are dispatched (entering the Back-End) and sent to its corresponding queue, the execution of each instruction can be made out-of-order and in parallel whenever there are sufficient structural resources (adders, bus, register file ports, memory ports, etc.).

There are seven main stages along this superscalar processor, which are:

- Fetch
- Decode
- Rename and Dispatch
- Issue
- Execution
- Write Back
- Commit

Also, there are some main structures involved in the superscalar processor core (Reorder Buffer, Integer Queue, Floating Point Queue, Address Queue, Load Queue, Store Queue, Branch Predictor, Bypass, Register File, Free Register List, Register Alias Table, etc.). In order to understand how the parallelism is exploited, in *Figure 2-6* [7] is shown the comparison

between the amount of in-flight instructions in a scalar processor and a superscalar processor, here is shown that the amount of in-flight instructions inside a superscalar processor (for this example) is doubled as well as its execution is twice the number of executed instructions in the scalar processor in the same number of clock cycles. A dynamic scheduling [8] must be integrated along the data path in order to exploit the instruction level parallelism existent in the code in execution, this dynamic scheduling ensures correctness within the dependencies and takes advantage of parallel execution techniques.



*Figure 2-6. Scalar vs Superscalar Execution in Pipeline Processors*

*Figure 2-7* illustrates the building blocks inside a superscalar processor.

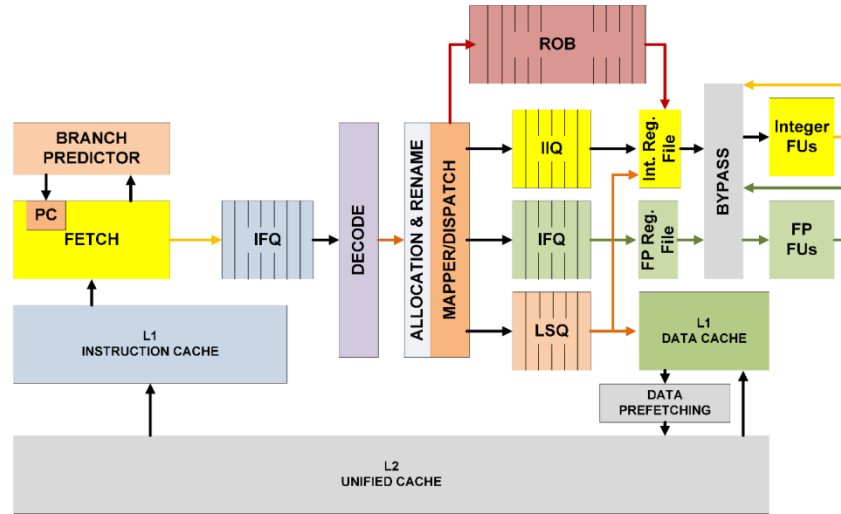


Figure 2-7. Superscalar Processor Architecture

PC = Program Counter; IFQ = Instruction Fetch Queue; ROB = Reorder Buffer; IIQ = Instruction Integer Queue; IFQ = Instruction Floating Point Queue; LSQ = Load Store Queue; Int. Reg. File = Integer Register File; FP Reg. File = Floating Point Register File; Integer FUs = Integer Functional Units; FP FUs = Floating Point Functional Units

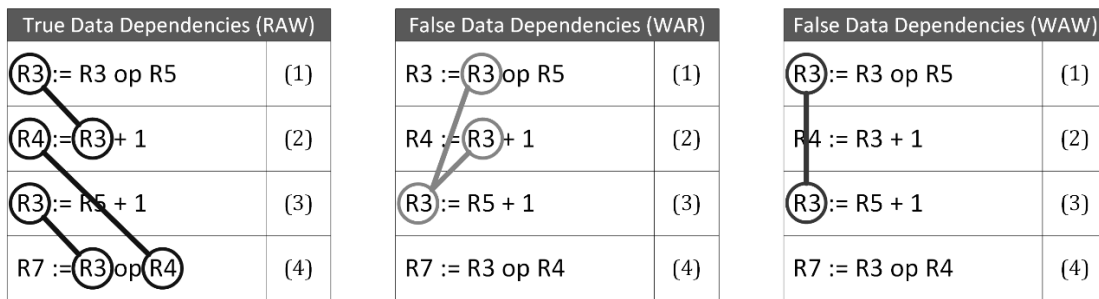
### 2.3.1 DEPENDENCIES

There are three different categories for dependencies: “*Data dependencies*”, “*Control dependencies*” and “*Structural dependencies*”. Because the main objective of a superscalar processor is to exploit the instruction level parallelism and to execute the maximum amount of instructions in parallel, most of its logic is dedicated in identifying and solving these dependencies.

Inside the *Data Dependencies* category, there are two kind of data dependencies and the dependencies can be either within memory locations or CPU registers, the “*True data dependency*”, which can be defined as a *RAW* (read-after-write) dependency, happens when the result of one instruction is needed as an input operand in another younger instruction, in order to do the correct execution for this sequence, the younger instruction must wait to the first instruction to be executed, after the first instruction has generated the result, the younger instruction can proceed, otherwise, it will have erroneous values.

The other data dependencies are called the “*False data dependencies*” (also known as “artificial dependencies” or “name dependencies”), although these dependencies are cleared

with the Renaming logic, for memory access instructions these dependencies are not resolved until their memory addresses are known, these are the *WAR* (write-after-read) and the *WAW* (write-after-write) dependencies, the first one happens when an older instruction is trying to read an operand from the Register file or from data from memory location, and it is going to be updated by another younger instruction. In order to ensure correctness, the older instruction must read this location before the younger instruction writes in it. The other dependency, *WAW*, happens when two instructions are trying to write to the same register or memory location, in order to keep the most actual value, they have to update this value in order, it means that the younger instruction must wait to the older instruction to write in this location (*Figure 2-8*).



*Figure 2-8. True and False Data Dependencies*

The *Control Dependencies* are present in conditional branch instructions inside the program in execution, it is because the conditional value is not known until the execution stage, then, there must be decided either to take the branch or not, in advance to this stage.

The last dependencies, the *Structural Dependencies* occur when two or more instructions are requiring the same architectural resource, it can be an adder, multiplier, register port, etc., if there are no available resources, the instructions have to be executed sequentially.

### 2.3.2 MAIN PIPELINE STAGES OF A SUPERSCALAR PROCESSOR

Fetch is the stage (*Figure 2-9*) where the instructions are brought from the L1 Instruction Cache Memory and stored in a structure called Instruction fetch queue, in this structure are housed the instructions already read in advance from the Instruction Cache (in order to preserve good performance and low memory access traffic, these instructions have to be read



faster than the time they take to execute). The Instruction Cache is read with the Fetch address, which can be the Program Counter value (next sequential value), Branch target value, Return-address value read from the stack, etc., this fetch address is computed every cycle, and thus, there must be extra considerations with the conditional branches, jumps and any instruction that goes to a specific instruction memory address. Furthermore, because in conditional branch the next fetch address isn't known until the execution stage, A *Branch Predictor* is needed working in parallel with the fetch engine.

This Branch Predictor is often build of two primary blocks, the Branch Target Buffer (BTB) which gives the last address where the branch has jumped, and the Branch History Table (BHT) where the branch condition is predicted with the help of its "taken" or "not taken" history records, owing to the fact that branch mispredictions can occur, this branch predictor is updated with any wrong or correct prediction. A mechanism for recovery must be launched whenever this happens, instructions-stream of the wrong path needs to be flushed of the pipeline and the correct path has to be re-established by fetching the correct execution path. There must be a recovery logic inside the processor for this correction.

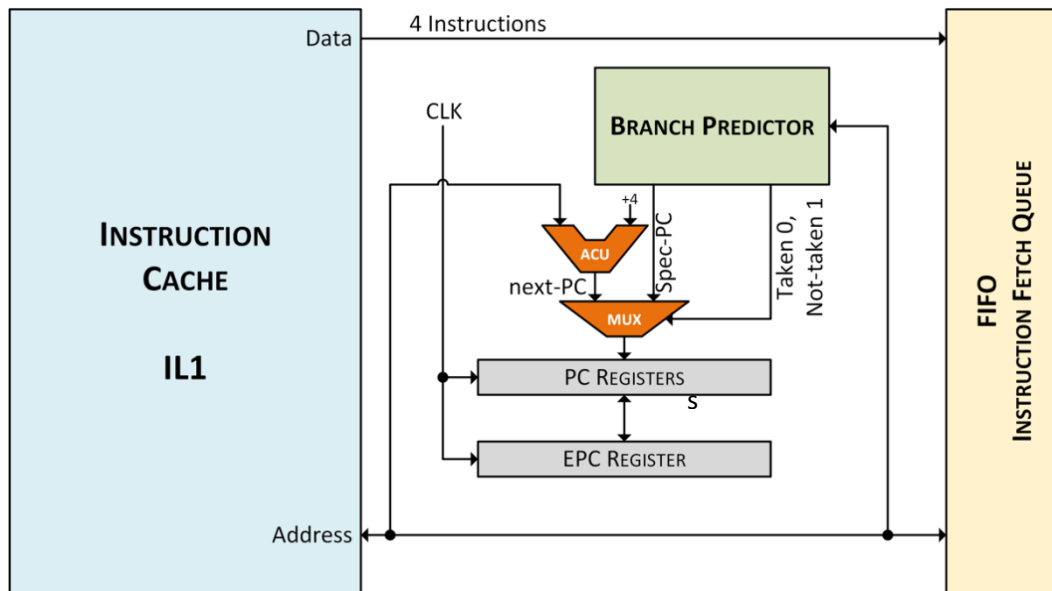
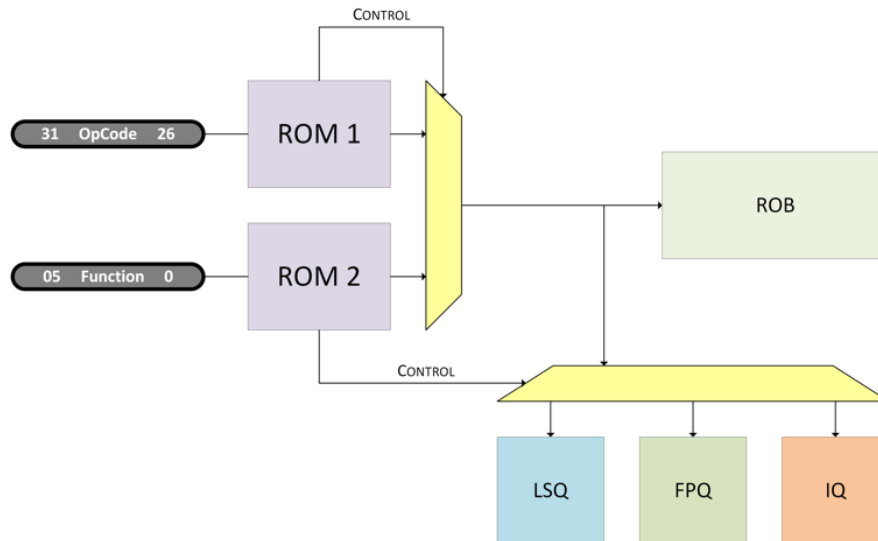


Figure 2-9. Fetch stage

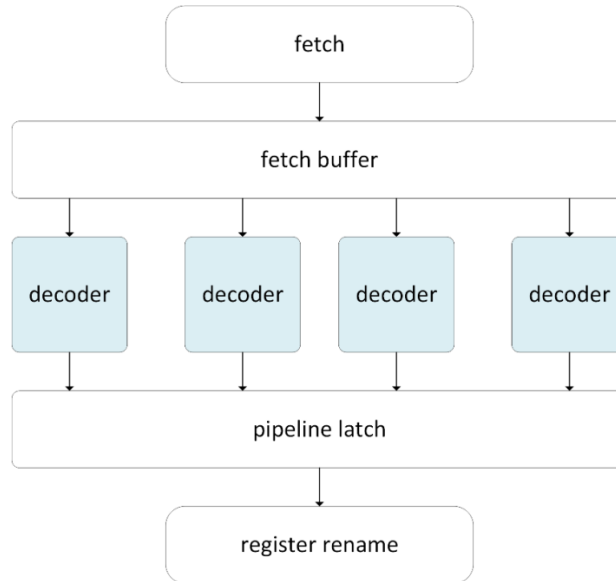
EPC Registers = Exception Program Counter Registers; ACU = Address Computation Unit; Spec-PC: Speculative PC

Decode is the stage where instructions are read from the Instruction fetch queue and are identified by their instruction opcode and function field, both opcode and function address the ROMs (*Figure 2-10*) which bring the operation control bits as well as flags indicating if it is a branch instruction, a jump, and link instruction, etc., this control bit packet is called “resource vectors”. The register tags and immediate value for every instruction are read in this stage.



*Figure 2-10. Decoder ROMs*

It is needed a decoder for each instruction read from the Instruction Fetch Queue in the same clock cycle (*Figure 2-11*).



*Figure 2-11. Typical RISC decode stage*

Rename is the stage where the logic registers are linked to physical registers in order to solve false dependencies (also known as “artificial dependencies” or “name dependencies”), this allows the instructions (not memory access instructions) to be executed freely without any data hazards whenever they have their source operands ready to be read from the register files. If two or more instructions do not have dependencies, then the ILP can be exploited by executing them in parallel and out-of-order.

In order to illustrate how the renaming process takes place, a simple renaming operation is shown in *Figure 2-12* [7] (L=Logical Registers, P=Physical Registers), here the L3 is renamed to P6 and in the second version of L3 is renamed to P9, because it was sharing the same location with another instruction that should not be dependent (name dependency), it has to be renamed in both the first and second instructions in order to have the last updated value for this register. In this same cycle, the first version of L3(P6) becomes old with the second version of L3(P9), then the value of P6 isn’t needed anymore in the program execution, thus, it can be freed. Instructions 3 and 4 must preserve the true dependencies with the last version link of the register L3(P9). It has to be considered that the logical registers L3, L4, and L5 were previously renamed to the physical register P3, P4 and P5 respectively, and that the new physical registers P6, P7, P8 and P9 are available for renaming the logical destination registers tags.

UN-RENAMED REGISTER TAGS		RENAMED REGISTER TAGS	
L3 := L3 op L5	(1)	P8 := P3 op P5	(1)
L4 := L3 + 1	(2)	P4 := P8 + 1	(2)
L3 := L5 + 1	(3)	P9 := P5 + 1	(3)
L7 := L3 op L4	(4)	P7 := P9 op P4	(4)

Figure 2-12. Register renaming example

Regardless of register renaming, the true dependencies must be kept, for example, within instructions 2 and 4, P7 must be kept with its true dependency. The same happens between the instructions 1 and 2 with L3 (renamed to P6). This is accomplished by the dependency check logic.

This register renaming algorithm can be stated with these steps:

- 1) Look for all instruction inputs (Src) and outputs (Dest).
- 2) Check for true dependencies ( $Dest_i \cap Src_j$ ).
- 3) Detect early old destinations ( $Dest_i \cap Dest_j$ ).
- 4) Rename new destination (PDest).
- 5) Save the actual registers assignation (Context saving).

Four structures are necessary to implement this register renaming logic (*Figure 2-13*):

- 1) Register Alias Table (RAT): It has the record of all the renamed registers assignation.
- 2) Dependency Check Logic (DCL): Identifies all the dependencies inside a group of instructions being renamed.
- 3) Free List Register (FLR): It has the record of all the unassigned physical registers.
- 4) Shadow Register Map (SRM): It saves the state of all the assignations in order to perform a precise context recovery whenever it is necessary.

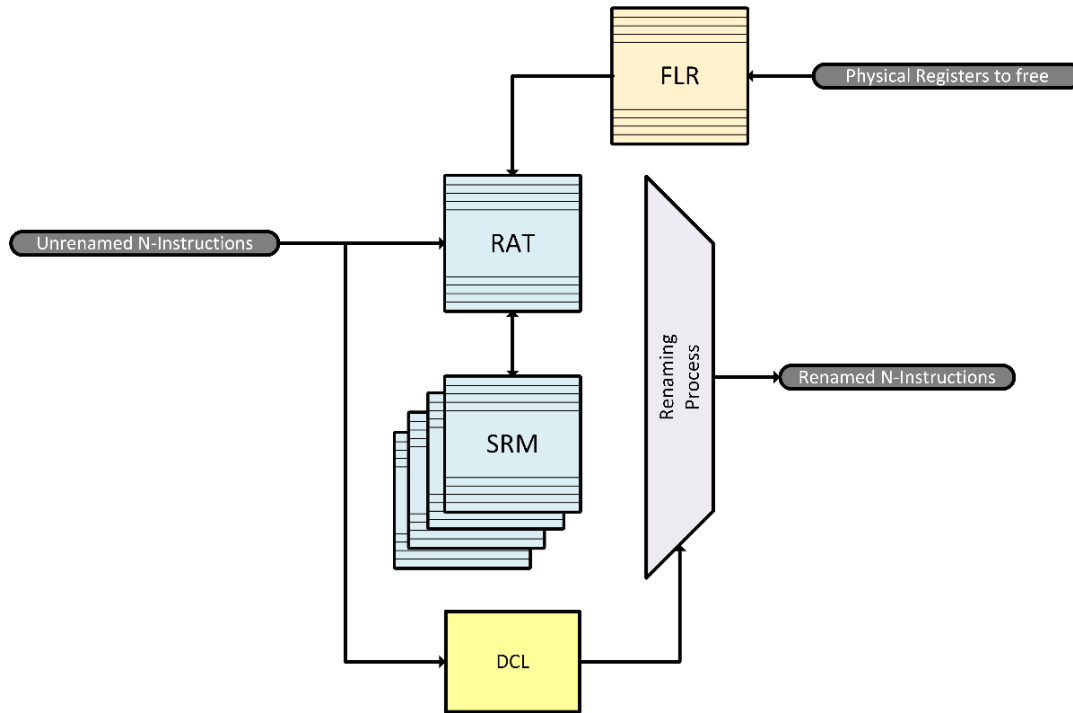


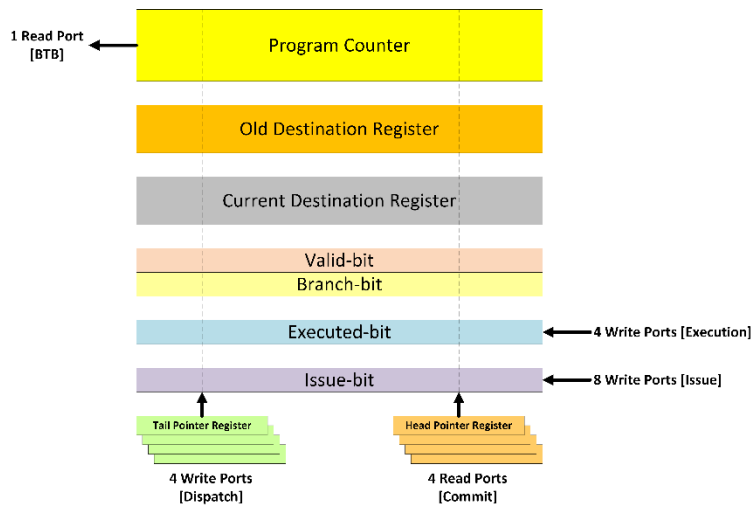
Figure 2-13. Register Renaming building blocks

There are three main renaming schemes used in contemporary processors [9], the first one is the “*Renaming through the Reorder Buffer*”, where the non-committed instructions results are stored in a Reorder Buffer, and only when the instruction is committed this result is written to the Register File changing the architectural state of the processor. The second one is the “*Renaming through a Rename Buffer*”, it is very similar to the first scheme, but it has a separate buffer for only those in-flight instructions that generates a result value, it saves storage and energy in comparison with the first scheme. And the third scheme is the “*Merged Register File*” scheme, here, the Register File stores both speculative and committed values, its main advantage is that there is no data transfer from the Reorder Buffer to the Register File.

Also in this stage, each instruction is assigned to an entry in the Reorder Buffer (ROB), if there isn't any empty entry in the ROB, the instruction fetch has to be stalled and wait for an empty entry. In order to identify which physical register can be freed (it means that its stored value isn't needed anymore and can be reused by another un-renamed register), the Reorder Buffer has two special entries for that purpose, the current destination register tag and the old destination register tag, so that, when the instruction in that ROB entry is committed, its old destination register is freed to the free register list (at this point) and the most actual value for that logical destination is the current destination register. The ROB (also known as the “active

list”) keeps track of every dispatched instruction inside the Back-End block of the processor, it is used to save the ordering of the instructions, even if they are executed out-of-order, at commit time, every instruction must be resolved in order (*Figure 2-14*). Whenever there is an exception or branch misprediction, the recovery logic read the modified variables using the In-Flight Tag given by the ROB and bring these variables back, then the execution of the program can continue. *Figure 2-15* shows the organization of the ROB (as a circular FIFO buffer) [10].

The Context Recovery is different between the Front-End and the Back-End blocks of the processor, for the Front-End, all the in-flight instructions have to be flushed and the fetch mechanism must take the right path immediately (the flush logic is simpler), whether in the Back-End (taking more clock cycles than in the Front-End), only the in-flight instructions that were dispatched after that recovery point have to be flushed, also, the renaming tables have to be modified with the correct entries stored at that recovery point, and the queues entries that were modified by the wrong path have to be recovered (*Figure 2-16*) [9].



*Figure 2-14. Reorder Buffer*

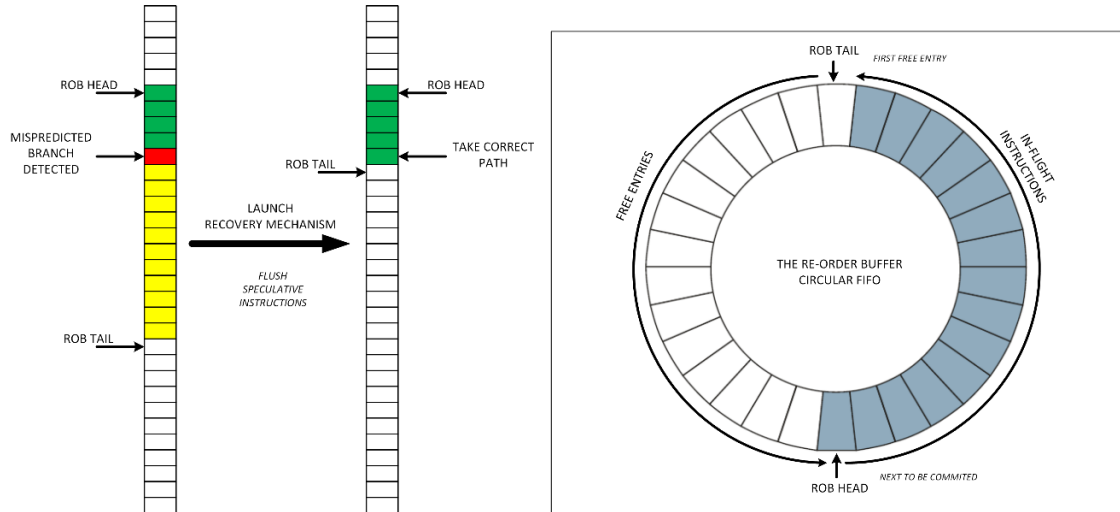


Figure 2-15. Instruction's tracking inside the ROB

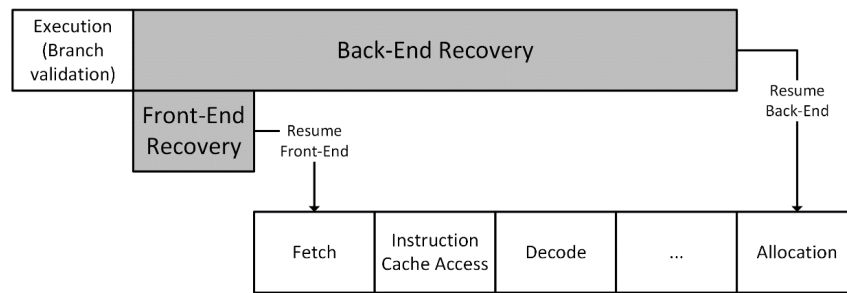


Figure 2-16. Recovery Logic in Front-End and Back-End

Dispatch is the stage where each instruction is sent to its corresponding queue (Integer Instruction queue, Floating point queue, Load store queue, etc.) (Figure 2-17). The Mapper is in charge of selecting the instructions by their kind of operation (integer, floating point, memory access, etc.), the selected instruction information is written in its assigned entry in the RAM-CAM queue structure, in the payload RAM are written the complete operation bits of the instruction, its source operands, destination register tags, and its resource vector, whether in the CAM are written the source operands tag to perform the dependent instructions wake-up by effectuating comparisons between results tags and source operands tags of instructions waiting in the queue (stored in the reservation stations) (Figure 2-18).

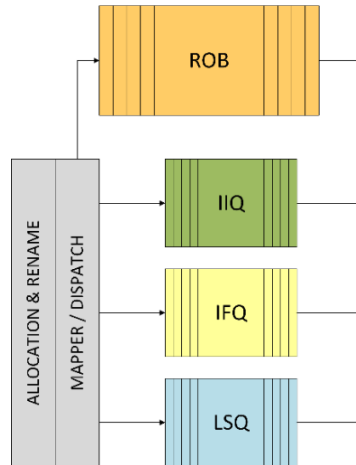


Figure 2-17. Dispatch Logic

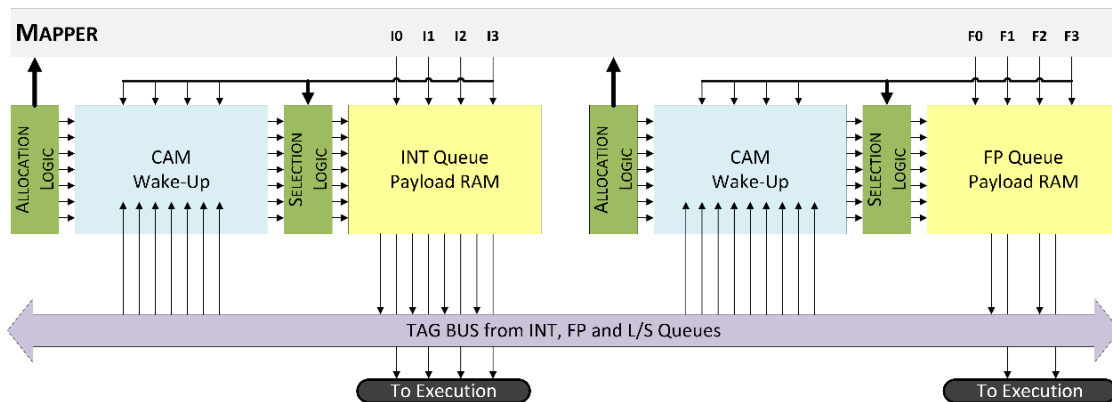
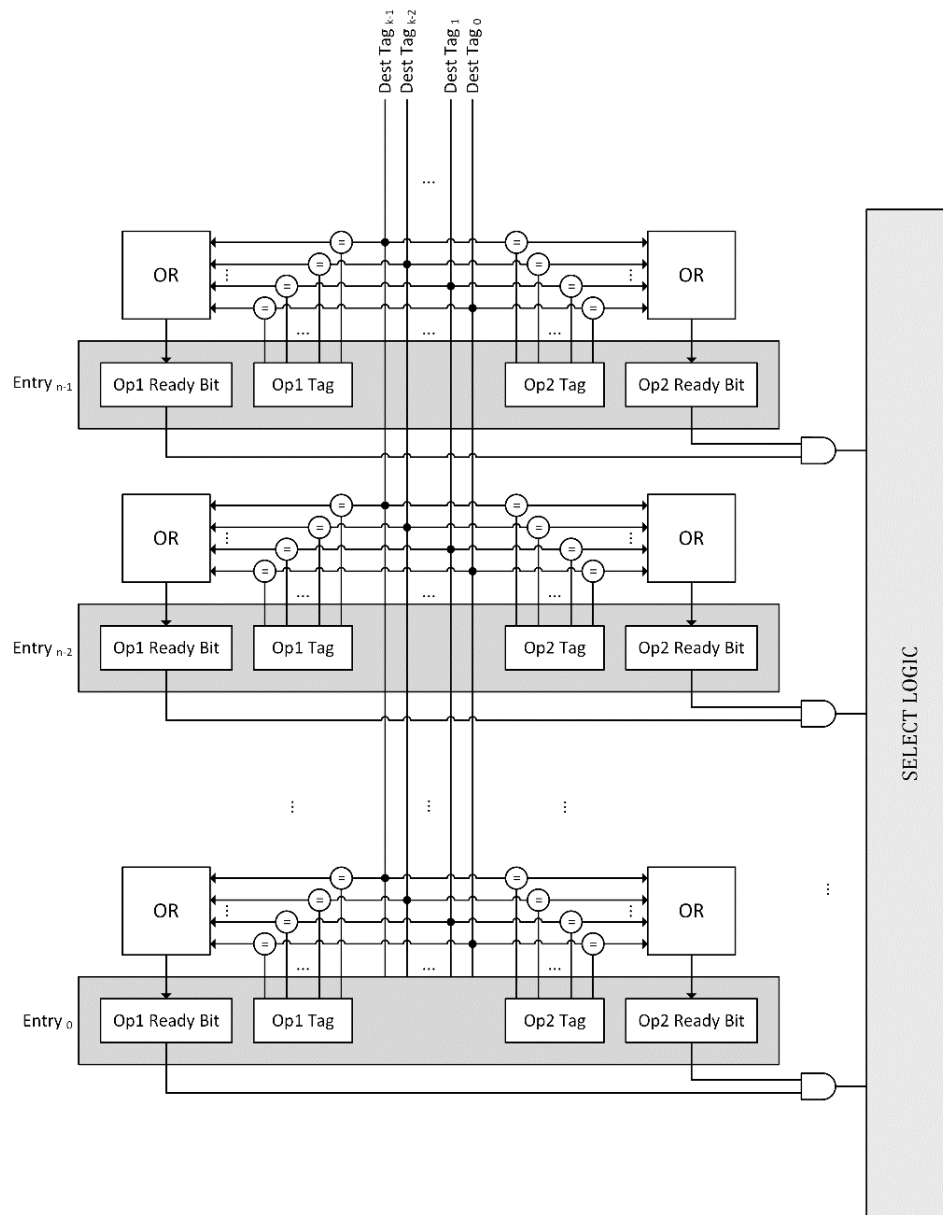


Figure 2-18. INT and FP Queues with Mapper

Issue stage is where the instructions that are ready to be selected (that is to say, that values of their source operands have been generated by older instructions and these values can be read from the Register Files or bypassed across the value forwarding network), can be issued in an out-of-order fashion, but there must be extra care with the memory access instructions (because the memory address is still uncertain). In this logic are involved the Wake-up logic, Ready logic, Selection logic (with a priority order logic) and Context recovery logic. The Wake-up logic is in charge of “waking up” these instructions by doing comparisons between the source operands tags stored in the CAM and the destination registers tags sent through the tag bus by the functional units, whenever there is a match, the Ready bit of the source operand is set to “1” because its value has been produced, then, when all the source operands are ready, the Select logic indicates which instructions can be issued to the next stage. *Figure 2-19* illustrates a CAM organization inside an Instruction queue, with “k” Destination Tags and “n” entries in the queue [1].





*Figure 2-19. Organization of the CAM Section in the Instruction Queue*

The destination register tags, the source operands register tags and the resource vector of the selected instruction are read from the payload RAM and sent to the next stage (*Figure 2-20*). The cycles required by an instruction to do its operation depend on which resource is requesting (i.e. an adder will take fewer cycles than a multiplier or a divisor), thus, the wake-up signal for those instructions that take one cycle to complete has to be sent at the same time it is issued (by the select logic) in order to efficiently do the bypassing with consecutive consumer instructions, with the instructions that take longer number of cycles to complete, the wake-up signal needs to be scheduled to be sent a few cycles before the end of the ALU's

computation. In *Figure 2-21* [9] is shown a comparison between two pipelines, one pipeline with this wake-up signal is sent at the end of Execution stage and another pipeline sends this signal at Select time, it is illustrated that without sending this signal at Select stage, it generates (for this example) three bubble cycles for the next consumer instruction, thus, it decreases the overall IPC, whether in the second pipeline the data bypassing is performed.

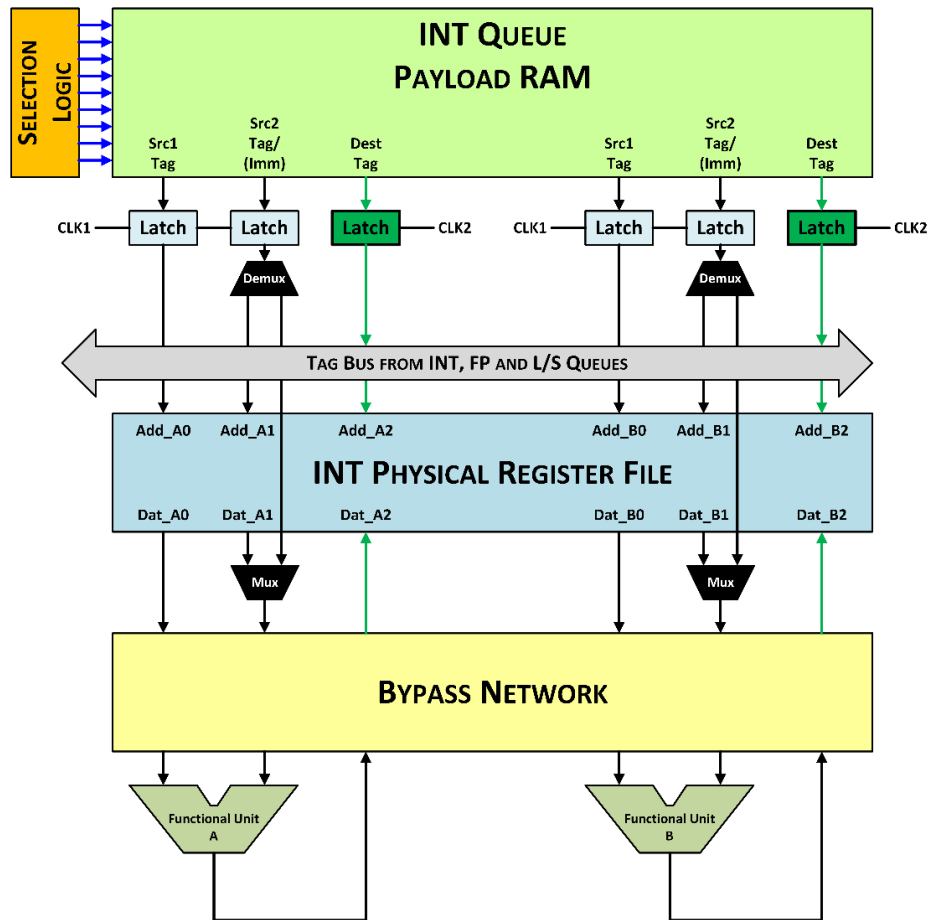
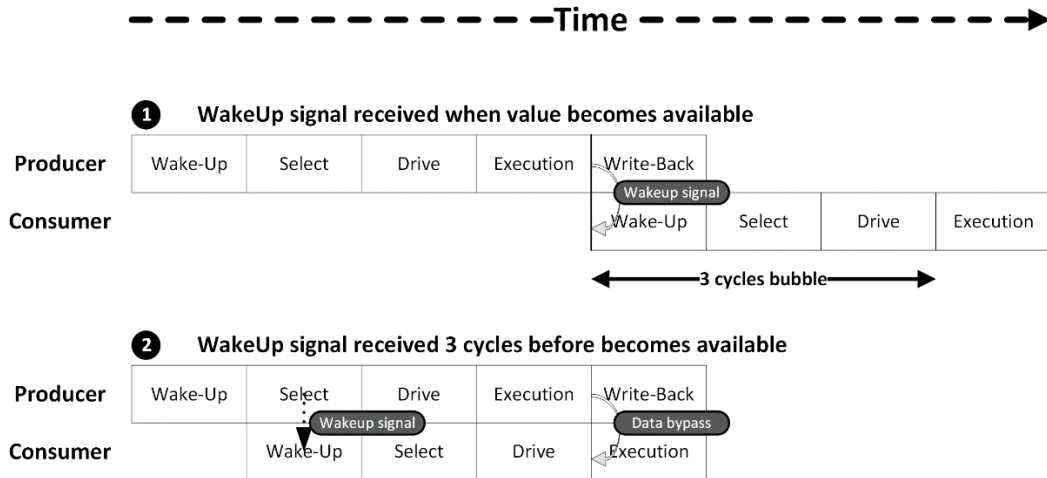


Figure 2-20. Two-Issue INT Queue's Payload RAM Logic



*Figure 2-21. Timing of the wake-up signal to support back-to-back execution*

The Execution stage is where the issued instructions read their source operands from their corresponding Register Files or get the data bypassed from the Destination Bus and are sent to their requested ALU resources (adder, multiplier, divider, etc.), here, any arithmetic or logic operation is computed as well as memory addresses are calculated. The Integer functional units consist of adders, logical functional units, shift functional units, conditional branch solvers, multipliers, and dividers. The Floating Point functional units are more complex than the Integer ones, at first, the values in the FP Register File have to be encoded as the IEEE 754 Standard format for single and double precision, the operands go through a decompression operation after being read from the FP Register File and get compressed before writing in it (encoded according to the single or double precision format), except for FP load/store operations, the FP functional units consist of an adder, a multiplier, and divider/square root logic unit, the adders can execute several operations such as additions, subtractions, comparisons and format conversions.

After finishing execution phase, in the Write-Back stage, the result values from the functional units are written back to the Register files (integer or floating point) as well as sent to the Bypass Network in order to perform a data bypassing to younger issued instructions (*Figure 2-22*) [9], also, the data read from the L1 Data Cache Memory (in a load instruction) is written to the Register files, the data in store instructions remains in the Store queue until commit stage (then the Data cache can be updated).

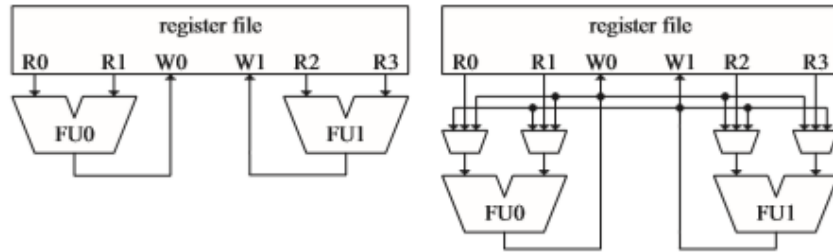


Figure 2-22. An execution engine with two functional units, without (left) and with (right) value bypassing

Commit stage, also known as Retire stage or Completion Stage, is in charge to modify the speculative state of the processor to an architectural state, the structure in charge of committing the instructions is the ROB (it has the instructions order stored in its entries), in order for an instruction to commit it must have all its pipeline flags set to “1” (valid-bit, issue-bit, execution-bit, no-speculative-bit), also, the previous older instructions must have committed in order to guarantee the ordering correctness of the program. When an instruction commits, its old destination register is freed so that it can be used again in the register renaming process (*Figure 2-23*), the current destination register changes its status to no speculative value and the Branch predictor tables are updated if it is the case.

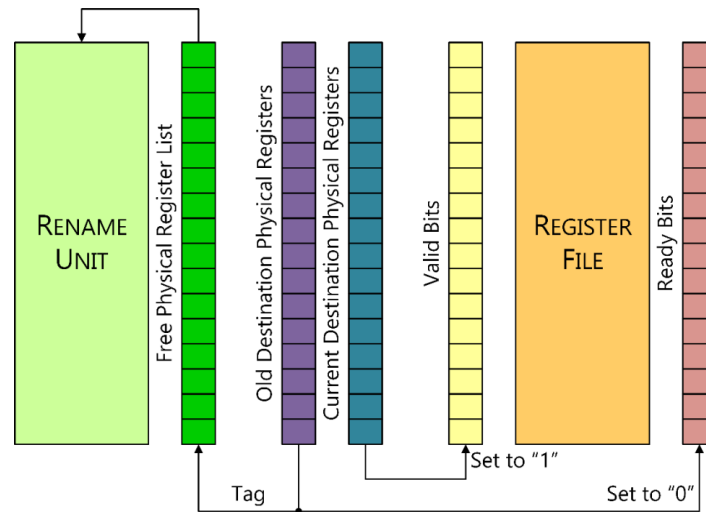


Figure 2-23. Recycling Physical Registers at Commit Time

### 2.3.3 ADDRESS, LOAD AND STORE QUEUES

Conventionally, the Load/Store queue is divided into three different building blocks, the Address queue (AQ), Load queue (LQ) and the Store queue (SQ). The LQ houses load instructions with their memory address computed and ready to read the Data Cache, similarly

the SQ houses store instructions that have already computed their memory address but waiting to commit in order to update the Data cache with the source data read from the Register file, whether the AQ has the memory access instructions mixed (loads and stores) and without their memory address computed, here the memory access instructions are waiting for their source operand to be ready in order to be sent to Execution stage in order to compute their memory address ( $\text{Address} = \text{Base register} + \text{Offset}$ ).

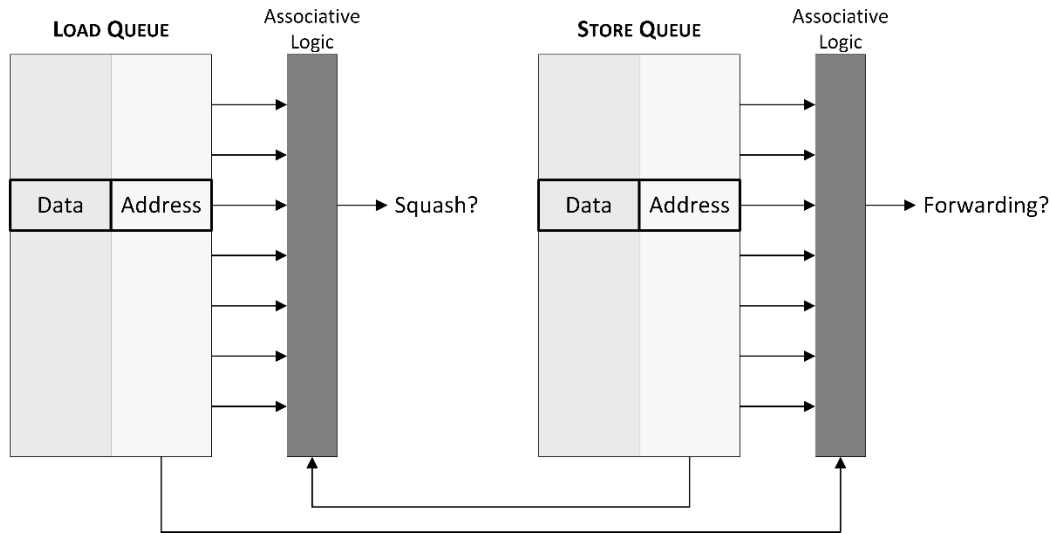
Because the memory addresses aren't known until execution stage, the issuing logic for memory access instructions is a little more complex, this AQ remains organized as a FIFO buffer in order to maintain the program order so that the dependencies can be computed easily, also, these instructions are kept in the queues until they commit. Due to the fact that the rename mechanism used for integer and floating point instruction is not feasible for memory addresses (there are a lot more memory locations than physical registers) [11], there must be a mechanism in charge of ensuring that the data hazards are properly resolved in order to permit an out-of-order execution, this mechanism in charge of detecting all dependence violations is called the "Memory Disambiguation Policy". There are two kinds of memory disambiguation schemes, the non-speculative scheme, and the speculative scheme, the non-speculative scheme practically issues the memory access instructions in order, whether the speculative uses a dependency predictor to issue those instructions that are predicted as co-dependent of an older memory access instruction.

Although the non-speculative issues in order, there are three kinds of non-speculative issuing, some with an out-of-order execution, in Table 1 [9] are shown these different kinds of memory disambiguation schemes.

*Table 1. Memory disambiguation schemes*

<b>Name</b>	<b>Speculative</b>	<b>Description</b>
Total Ordering	No	All memory accesses are processed in order.
Partial Ordering	No	All stores are processed in order, but loads execute out-of-order as long as all previous stores have computed their addresses.
Load Ordering Store Ordering	No	Execution between loads and stores is out of order, but all loads execute in-order among them, and all stores execute in-order among them.
Store Ordering	Yes	Stores execute in-order, but loads execute completely out-of-order

*Figure 2-24* [11] shows a conventional Load Store queue design separated for loads and stores in two different queues, in this scheme, a load associatively searches the Store queue in order to forward the data from an older in-flight store, whether a store searches for loads that have executed speculatively (wrong) in order to squash them by its re-execution.

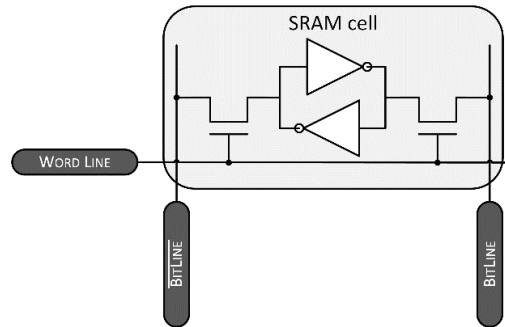


*Figure 2-24. Conventional LSQ design*

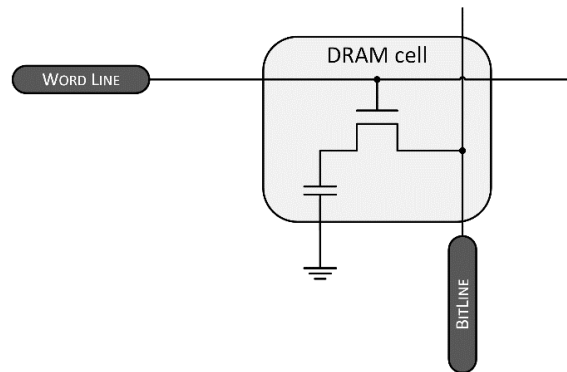
## 2.4 CACHE MEMORY AND MEMORY HIERARCHY

Due to the variation with the memory access latency depending on the capacity of the memory, the cache memory subsystem is hierarchically divided into cache levels (normally up to three levels), being the L1 the closer one to the processor and with the smaller access latency. Also, there are two kinds of cache organizations, the inclusive caches, and the exclusive caches, in the inclusive caches the lower level caches have portion of data (trying to be the portions currently in use) copied from the upper level caches, whether in the exclusive caches it is to say that there is only one copy of that portion of data within all the cache levels (is more complex to handle). The cache hierarchical levels use different memory technologies, for the internal cache memory L1, L2, and L3 are always implemented using an SRAM (Static Random Access Memory) because of its small access latency, whether for the rest of the memory subsystem levels are normally implemented with DRAMs (Dynamic Random Access Memory).

An SRAM cell (it stores one bit) is normally made of six CMOS transistors (6T SRAM cell), these transistors are composed of two access transistors and a pair of inverters (using two transistors each), whether the DRAM cell is a lot simpler within its structure, it is made of one or three transistors and a capacitor (needing to be refreshed every certain period of time in order to maintain its stored value). In *Figure 2-25* and *Figure 2-26* are shown these different memory technologies.



*Figure 2-25. SRAM cell*



*Figure 2-26. DRAM cell*

Though the SRAM access is a lot faster than the DRAM, using only SRAMs for the whole memory is unfeasible because of cost, also, this memory technology wastes more energy, occupies more hardware space and its latency is long. In order to concealment the overall memory access latency, the data that is being accessed constantly is stored in the high-speed L1, L2, and L3 cache memory and whenever there is a cache miss (the requested data is not found in the L1 cache memory), there is an exchange of data within upper cache levels. It is normally for the L1 caches to be separated into Instruction Cache and Data Cache, whether, for the upper cache levels, they are usually composed of mixed instruction and data information (unified cache).

The cache memory controller [4] is in charge of the tracking of all this stored information in the different cache levels, whenever there is a request from the processor, this cache memory controller checks if the data is stored in the high-speed cache, if effectively it is in L1, this location can be accessed immediately and there won't be any waiting cycles, but, if it is not, it results in a miss cache and the requested data is taken from upper memory levels L2, L3 or Main Memory (DRAMs) leading to wait cycles (Figure 2-27) [4].

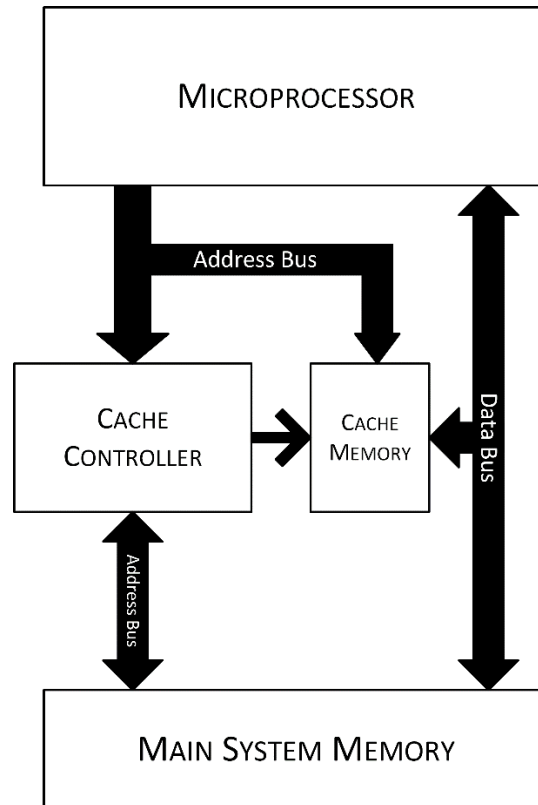
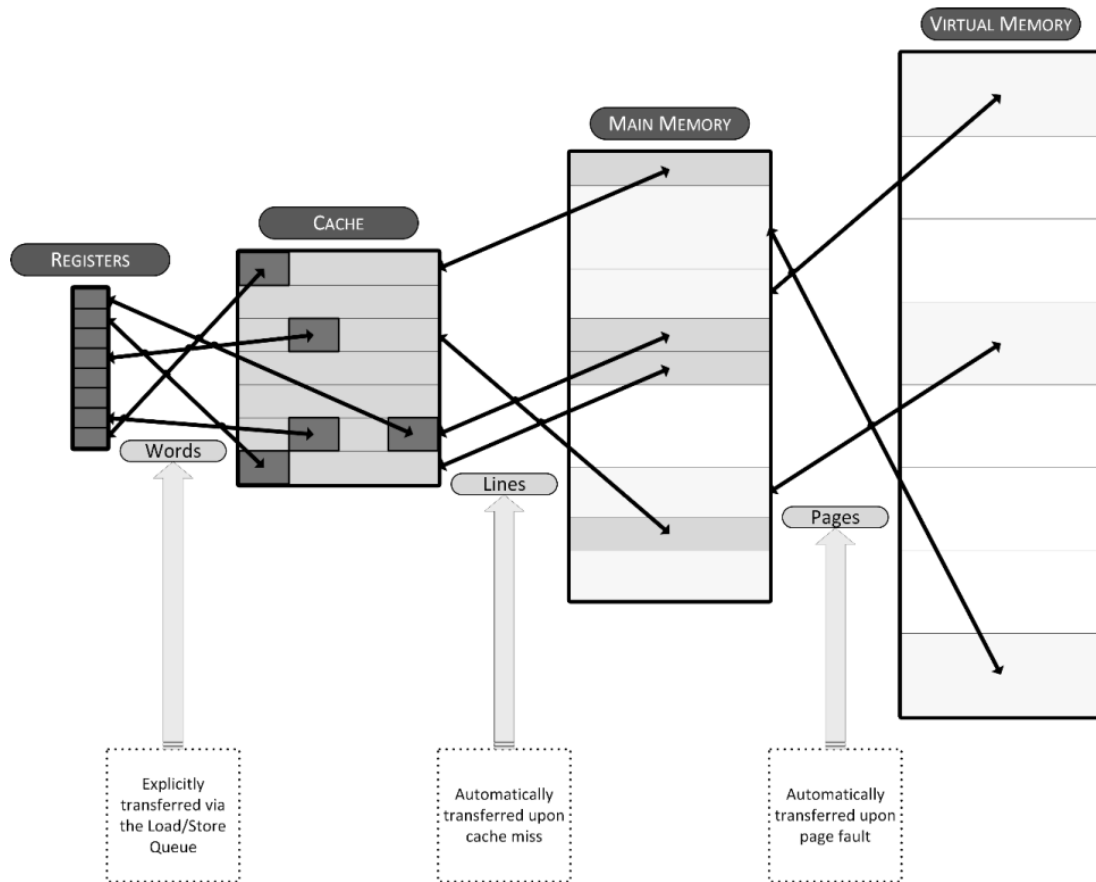


Figure 2-27. The Cache Memory concept

Memory hierarchies only work if the levels closer to the processor have stored data that the program will be reusing, if not, there will be a lot of wait cycles provoked by the long latency memories. Memory hierarchies work by taking into account the two aspects of the locality property inside an address space, the first aspect is the Temporal Locality which says that “*The information that is already in use is likely to be used in the near future*”, and the second aspect is the Spatial Locality, it says that “*The portions of the address space which are in use, generally consist of a fairly small number of individually contiguous segments of that address space*” [12].



Normally, the instruction cache memory addresses as well as the data cache memory addresses, are virtual addresses, there must be performed a prior translation in order to obtain the physical addresses. This physical address is the range of memory locations that can be generated and driven by the address bus, whether the virtual memory address is the range of memory locations that a program can make use of. The whole address space is divided into pages, and the operating system is in charge of mapping them with the help of its page table data structure stored in main memory (*Figure 2-28*).



*Figure 2-28. Memory hierarchy*

### 2.4.1 CACHE MEMORY PARAMETERS

The cache memory can be classified by some parameters within its architecture and by its write policy, these parameters are noted in *Table 2*.

*Table 2. Cache memory parameters*

BUS UTILIZATION	ASSOCIATIVITY	LINE SIZE	CACHE SIZE	WRITE POLICY
Look-through Look-aside	Fully-associative Direct-mapped N-Way set-associative	(Also known as block size), these are the number of bytes per line, normally 32bytes or 64bytes	KB of data in the Memory, normally these are 32KB for L1 cache and 512KB for L2 cache	Write-through Buffered write-through Write-back

### Bus Utilization

The cache can be organized and implemented as a Look-through cache or a Look-aside cache, each one with its own advantages and disadvantages. Whenever the Look-through architecture searches for the requested data in the L1 cache it uses the local bus, and if it wasn't found, then it goes to an upper-level cache to search that data, thus, it will be using the system bus at this point, it has to be noted that the bus utilization for the upper-level caches is only required whenever there is a cache miss, by this way it saves unnecessary searches to these long-latency, it also saves power-wasting upper-level cache memories as well as unnecessary system bus utilization. In the other hand, the Look-aside searches in the L1 cache memory as well as in the upper-level caches, that is to say, it does not isolate the local bus searches from the system bus searches, this wastes more power and does more utilization of the bus (this can drive to memory access overhead), by the way, an advantage is that whenever there is a cache miss it gets the requested data faster than with another scheme, while the Look-through first searches in the closer level cache and then to the other level caches, the Look-aside searches all at the same time, thus it does not waste time in waiting for a cache miss.

### Write Policy

There are two concepts for outdated data problems within the cache consistency, the first one happens when some memory location has been modified or updated (it is called to be "dirty" data) in a level cache but the outdated data in the another level cache hasn't (it is called

to be “stall” data), thus, an updating operation must be driven in order to ensure cache consistency. The most common case is when the cache has been updated (now is a dirty memory location) with a store operation, while the copy in main memory is outdated (has stall data for that memory location). The way the processor solves the cache consistency problem is called the “Write Policy”, the easiest implementation is the Write-Through policy, and this is because, whenever a store operation has been selected to update the architectural state of the processor’s data cache, it also updates the copy of that memory location in its upper-level cache, it is simpler to implement but it results in lower overall performance because it has to access a long-latency main memory. A modification of this scheme is the Buffered Write-Through policy, this is one of the most used, it has a buffer where the store operations are saved, and simulates that there was a cache hit, but actually the operation will take place a little later, this policy is an improvement of the write-through policy. The Write-Back policy only updates the upper-level cache whenever it is necessary (just the locations that are marked to be dirty), the upper level is updated only when the cache location that contains “dirty” data is about to be overwritten by another memory location, then the upper-level cache has to be updated with this modified data.

### Associativity

Remembering that the main memory is organized by pages and that the cache is divided by cache lines (also known as blocks or sets), there are three main concepts within the organization and addressing inside the cache, this organization can be a Fully-Associative, Direct-Mapped or an  $N$ -Way Set-Associative cache. With the Fully-Associative cache organization, the main memory is seen as a huge only “page”, so that it is only divided by blocks, in this organization every block in main memory can be located (or not) in any of the L1 cache blocks, thus, when a memory location is accessed, this memory address is searched in each of the cache directory entries in order to know if this memory location copy is actually in the L1 cache or if it may need to be taken from upper-level caches, this drives to long-latency waiting cycles in the case of a big L1 cache (more than 4KB), it also consumes more power energy because of all the comparisons inside this CAM directory. This model increases the cache hit rate as well as overall performance for those processors that use small L1 caches. In *Figure 2-29* is shown a simple diagram of this cache organization.

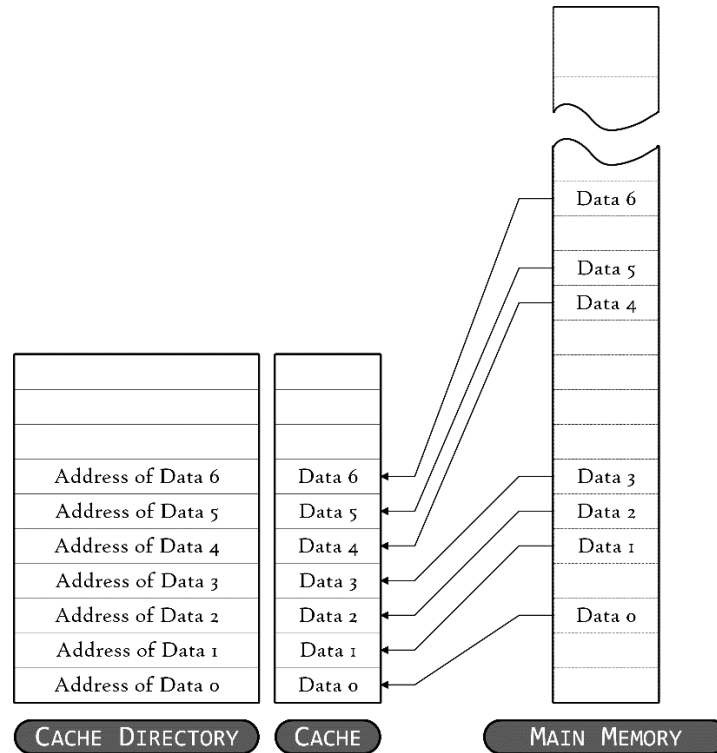


Figure 2-29. Fully-Associative Cache Organization

Contrary to the fully-associative cache is the Direct-Mapped cache organization, only one entry has to be checked in the cache directory in order to indicate if the memory location copy is in cache or in main memory (cache hit or cache miss), in this organization the main memory is divided into pages, so that every set in the cache can map to that set of any of the pages in main memory, thus, it is simpler to implement and the latency in checking the cache directory is smaller than with the other schemes, the disadvantage with this organization is that there is only one permitted entry in the cache per set in the main memory, then it forces that whenever it has to be accessed the same set but from another page, that set has to be overwritten (in the hypothetical case where there are various consecutive accesses to set 0 but from different pages, every access will incur in page miss) (*Figure 2-30*).

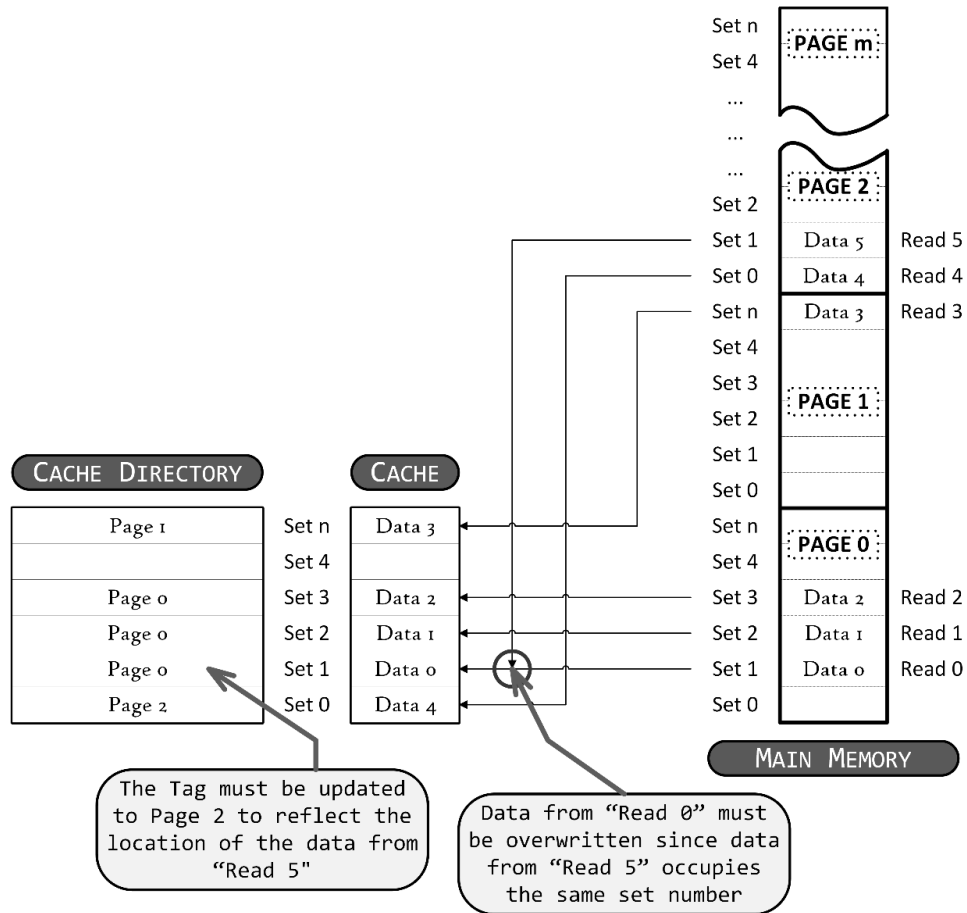


Figure 2-30. Direct-Mapped Cache Organization

In order to make it possible to have in cache more sets of the same set in main memory but from different pages, an improvement of the last cache organization, is the  $N$ -way set-associative cache, it can be 2-Way Set-Associative, 4-Way Set-Associative, etc., as it increases the number of "ways", its associativity increases. Each "way" is one more portion of cache memory of the size of a single page (4KB, 8KB, etc.), which can store the same set but from a different page in main memory, that is to say, if there is already a set stored in "Way A" and an access to the same set but from a different page occurs, it isn't necessary to be overwritten the already stored set in "Way A", it is stored in another way (i.e. "Way B"). This scheme increases the cache hit rate compared to the Direct-Mapped cache organization.

Figure 2-31 illustrates a simple diagram for this cache organization (a two-way set-associative cache organization).

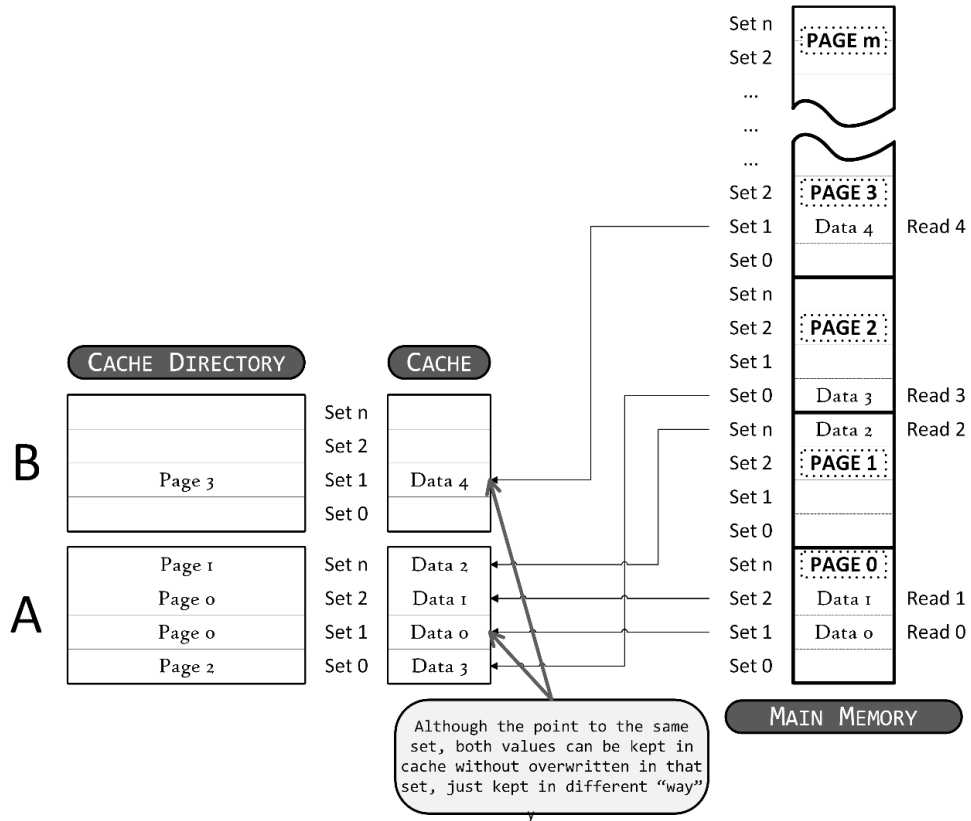


Figure 2-31. Two-Way Set-Associative Cache Organization

## 2.5 TAG AND DATA ARRAY ACCESSES

Each memory address has three fields of information, the  $K$  least significant bits are the offset bits, with these bits a specific byte along the cache line (set) can be accessed, the next field is the index field, it determines which set in the page is being addressed, and the last field (in the most significant bits) is the tag, which is used to identify the page inside the memory subsystem. The size of these fields is noted in Table 3.

Table 3. Memory Address Fields

<i>Taking into account...</i>	<b>Block Size (in bytes)</b> = $Q$
	<b>Number of sets</b> = $S$
	<b>Tag bits</b> = $T$
<b>Offset bits (K)</b> = $\log_2(Q)$	
<b>Index bits (M)</b> = $\log_2(S)$	
<b>Page Size (in bytes)</b> = $S * Q$	
<b>Memory address length</b> = $T + M + K$	

Because different addresses can map to the same set inside the cache (but it belongs to a different page, though), there is a Tag Array that serves as a mechanism to “reverse-map” these indexes to addresses. It has the same organization as the Data Array, and for each block inside the data array, the tag array has stored both the Tag bits and the state bits of that block (whether it is valid, dirty, cached, uncached, etc.). There are two kinds of tag and data array accesses, the parallel access and serial access.

### 2.5.1 PARALLEL TAG AND DATA ARRAY ACCESS

In this scheme, the tag and data arrays are accessed in a parallel manner, the access process is the following: The tag bits from the memory address are compared with the tag bits found in all the “ways” of the tag array addressed by the index field of the memory address, if there is a hit, then it says that the memory location copy is in cache (it indicates in which “way” of the data cache blocks can be found), owing to the fact that all the “ways” of the data cache are read, the correct “way” has to be selected with the help of a multiplexor (the selected “way” is indicated by the tag array hit). If the tag is not found in the tag array, then it is said that the memory location copy isn’t in the L1 cache and it has to be brought from upper-level memory. This scheme is fast by the fact that it read in parallel both the tag array and the data array, but it unnecessary wastes power energy in reading all the “ways” of the data array (*Figure 2-32*) [9].

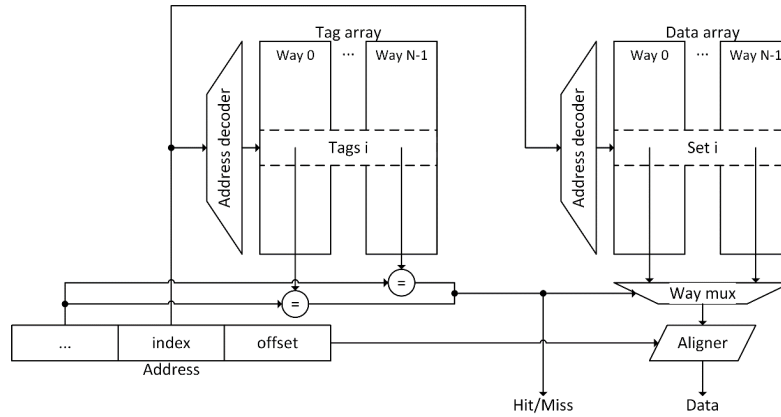


Figure 2-32. Parallel Tag and Data Array Access

## 2.5.2 SERIAL TAG AND DATA ARRAY ACCESS

In the other hand, with the serial tag and data array access, an extra cycle is introduced but some energy is saved, also, the access frequency increases because the “Way-Mux” is not necessary anymore. The Tag array is accessed firstly indicating if there is a cache hit or miss (if there is a cache hit, in this phase it is already known in which “way” of the Data array the data can be found). If a hit, the data array is accessed directly, if not, then the data is brought from the upper-level memory (*Figure 2-33*) [9].

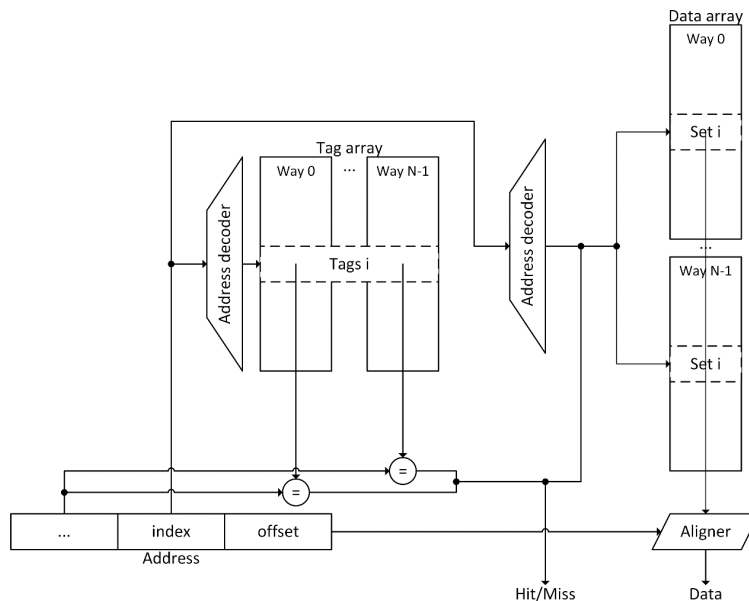


Figure 2-33. Serial Tag and Data Array Access



# Chapter 3

---

## STATE OF THE ART

---

In contemporary superscalar processors almost all designs include techniques such as store-to-load data forwarding [13], memory dependence prediction, speculated execution, etc., the conventional design is the one where the Load queue and the Store queue are two separated CAM queues searching associatively each other in order to perform either a data forwarding or a squashing in those load instructions executed prematurely (*Figure 2-24*), leading to unnecessary wasted energy, thus, there has been some Load Store queue design proposals which offer a low latency and a reduced power consumption [13] [14] [15] [16] [17] [18]. Due to the necessity in identifying any data dependency between a younger and an older in-flight memory access instruction, these research proposals use filtering techniques to reduce the number of associative searches.

Despite of this restriction, there are a lot of models which offer speculated execution for these load instructions freeing them from waiting for the younger store instruction to have their memory address computed, this is done via a memory address dependence predictor, and it can be a “naive” predictor or a more sophisticated “dynamic” predictor, this last one can actually offer a very accurate prediction ( $\pm 96\%$ ), it achieves this accuracy because it goes learning the memory address dependency patterns as long as the program is executing, when

the program starts its execution, the predictor will take “naive” decisions and fail occasionally but after some cycles the accuracy increase considerably.

Normally, it is necessary for a load instruction to wait for any older store instruction to have already computed its memory address in order to ensure any memory dependency with these older store instructions. If a load has a data dependency with an earlier store, the load either has to wait for the store instruction to commit or a store-to-load forwarding can be done in order to forward the speculative data from the uncommitted store to the load. Store-to-load data forwarding occurs when the load virtual address matches with a store virtual address and the store size is greater than or equal to the load size [19]. Normally, a load instruction is more urgent to be executed than a store [20], it is because the data read from cache memory is more probable to be needed by a close younger instruction, because of that, load instructions may have preference over store instructions, but there must be extra careful with their un-computed memory addresses, executing out-of-order dependent memory access instructions may lead to incorrect results.

The memory disambiguation is the mechanism that identifies any dependencies between memory access instructions, allowing these instructions to be executed in parallel. Any processor should have an efficient memory disambiguation scheme if the Instruction Level Parallelism is wanted to be achieved with memory access instructions. There is no dependency conflict when load instructions execute out-of-order, the conflict is generated when a load instruction executes before of a dependent older store, also, in order to maintain the memory semantically, it is necessary an in-order execution with those store instructions addressing to the same memory location, that is why modern out-of-order processors execute stores at commit time. In first memory disambiguation implementations this was made at compile-time [21] (called “static”) but it is more suitable if it is made via a combined hardware/software implementation or an only hardware implementation, that is at run-time (“dynamic” memory disambiguation).

Because the architectural state of the processor (talking about the data cache) is only changed at commit time, a store instruction has its source data stored in the store queue or written in a store buffer, allowing a store-to-load data forwarding by detecting whenever there is a load dependency with an older store.

Some early implementations of dynamic hardware disambiguation [20] are:

- The IBM System / 360 Model [22] – It has a store queue which can detect store-load dependencies. If it is detected a dependency between a load and an earlier store, the data is forwarded from the store to the load instruction.
- The HPS Model [23] - It uses a dependency matrix which blocks all the younger loads ongoing an unresolved store.
- The Address Resolution Buffer (ARB) [24] – It allows speculative execution for load instructions and it detects if this execution was successful (without any dependence) or not. Stores do not update data cache memory until commit time, this implementation is more complex than the IBM System 360 model and the HPS model.

---

## 3.1 RESEARCH PROPOSALS FOR LSQ & MEMORY DISAMBIGUATION

### 3.1.1 TRIPS

---

The TRIPS (Tera-op, Reliable, Intelligently adaptive Processing System) microarchitecture was designed and implemented by a research group in the Department of Computer Sciences at the University of Texas at Austin. This distributed, tiled microarchitecture prototype (*Figure 3-1*) offers higher instruction-level concurrency than current industrial processors, each core can execute up to 16 out-of-order operations per cycle and it is composed of multiple copies of five different types of tiles interconnected via microarchitectural networks [25].

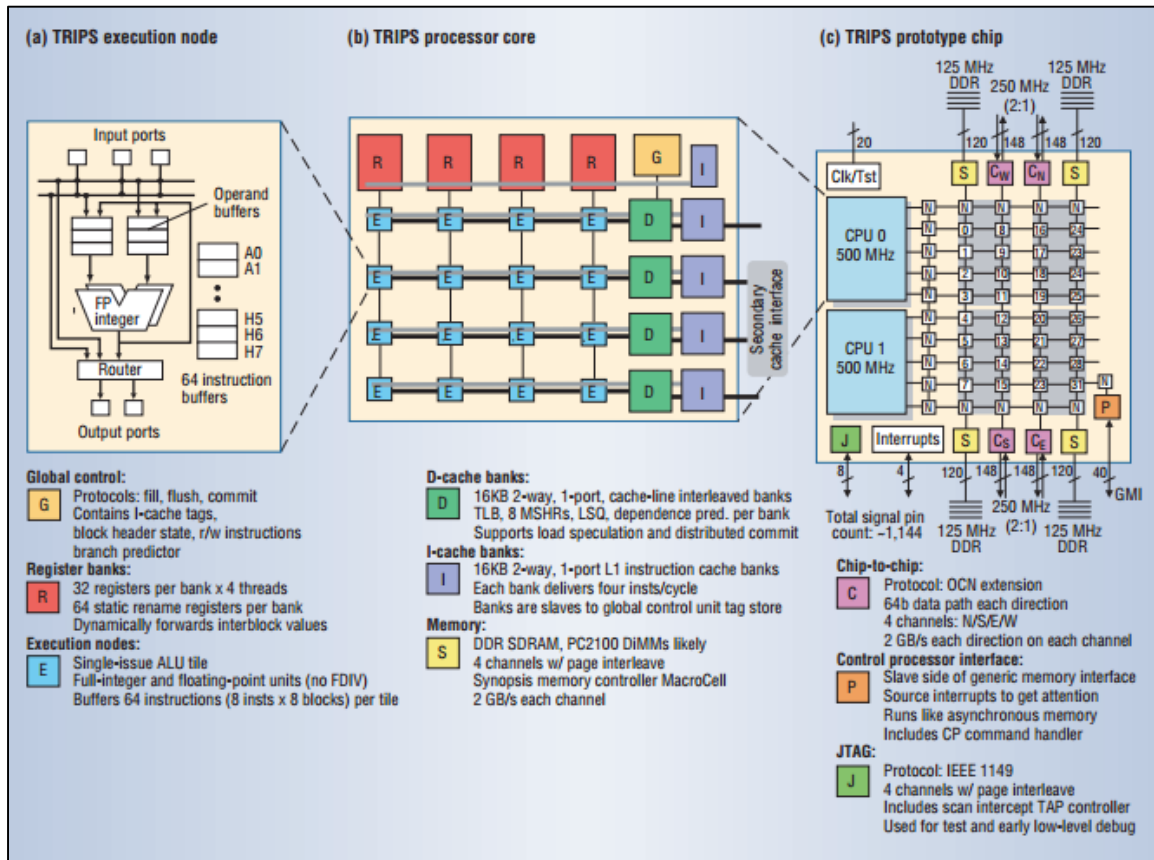


Figure 3-1. TRIPS prototype microarchitecture

Each data tile (DT) partition manages memory (load and store) instructions, performs address translation and protection with the help of its Data TLB, has a Miss-Handling Unit which can handle up to 64 cache misses, tracks and resolves memory dependencies between load and store instructions using its Load Store queues (*Figure 3-2*), it uses load/store dependency prediction in order to permit an out-of-order execution, and updates the data cache when the results become non-speculative.

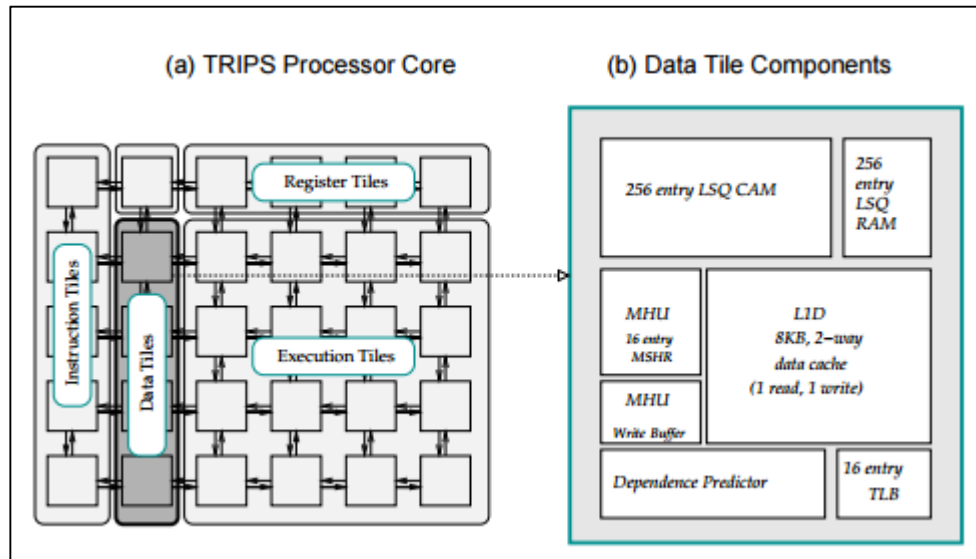


Figure 3-2. TRIPS Processor's Data Tile

Each load/store instruction can be mapped into any of the 16 execution units, it issues when all its source operands are available and then it is sent to the DT through the network. [26] When a load enters the data tile, it accesses the TLB to do an address translation, checks protection attributes, checks the Dependence Predictor (DPR) for a store dependency, identifies older matching uncommitted stores to perform a store-to-load data forwarding and it also checks the tag array for a cache hit/miss. There are four possible scenarios depending on the hit/miss responses as shown in *Table 4*, and thus, an operation to be performed in every scenario.

Table 4. Load Execution Scenarios (X=Don't care)

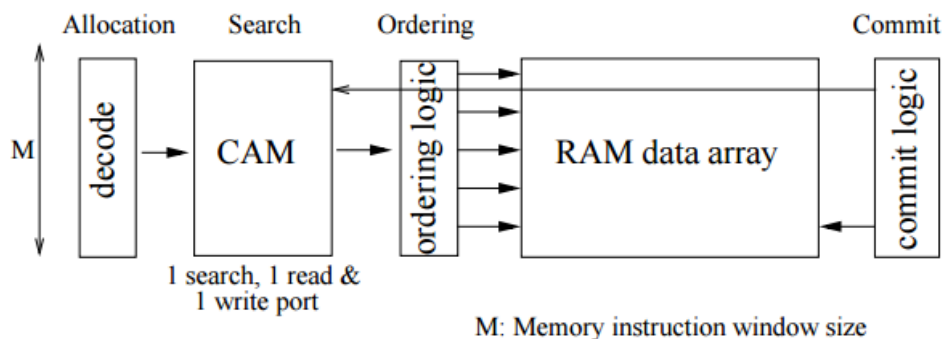
TLB	DPR	Cache	LSQ	Action
Miss	X	X	X	Report TLB Exception
Hit	Hit	X	X	Defer load until all prior stores are received
Hit	Miss	Hit	Miss	Forward data from cache
Hit	Miss	Miss	X	Forward data from L2 cache, issue cache fill request
Hit	Miss	Hit	Hit	Forward data from LSQ and cache

This architecture supports different sizes for store-to-load data forwarding (8, 16, 32 and 64 bits), thus, whenever there is a “hit” in the LSQ indicating that a store-load data forwarding is possible, the matching bytes are taken from the LSQ whether the remaining bytes are taken from the data cache.

When a store is sent to a data tile it is buffered in the LSQ and notifies to the other DTs that it has been issued, it checks for any dependence violation inside the DTs and if a younger load with the same memory address is found in the queue, then a recovery process is initiated. The dependence predictor is updated with this dependence misprediction. In order for a store to update the architectural state of the processor, it has to commit, then its store data is taken from the LSQ, the cache tags are checked in TLB and tag array (it is also updated the LRU algorithm mechanism), and if there is a hit, the cache/memory system is updated with the store data, then the corresponding cache line is marked as dirty.

### 3.1.2 TRIPS (UNORDERED, LATE-BINDING LSQ DESIGN)

[15] [27] Unordered, Late-Binding LSQs (ULB-LSQs) sizes are smaller than traditional LSQs, this is because it allocates the memory instructions at issue time, and thus, it requires a different allocation mechanism. The allocation entry is taken from a pool of free LSQ slots in comparison with the traditional LSQs which is age-indexed, this un-links the age of the instruction from its occupied slot. The age of every L/S instruction is stored in a separate special CAM which output can indicate the result of a “greater, lesser or equal” operation instead of a “match” operation. *Figure 3-3* and *Figure 3-4* show the contrast between the traditional LSQ design and the ULB-LSQ design.



*Figure 3-3. The Age-Indexed LSQ*

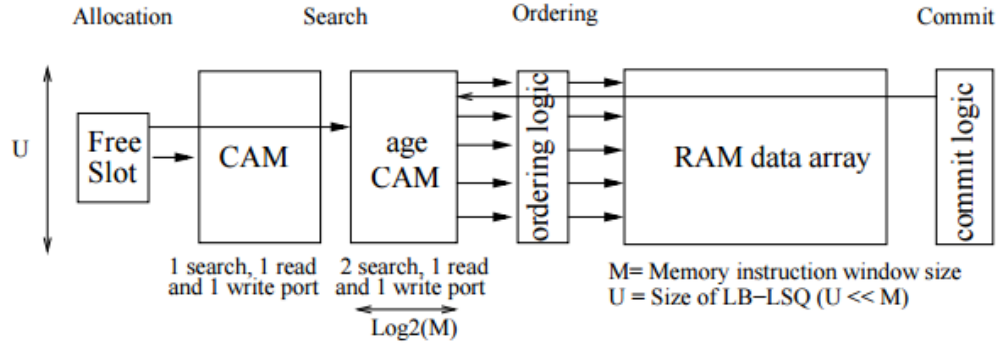


Figure 3-4. The ULB-LSQ Microarchitecture

When a store instruction arrives, it checks the address CAM for matching loads, and the special age CAM for younger loads, then using an OR operation between these two results, a violation is detected. With the store-load data forwarding mechanism, there is no conflict when there is only one match, but when there are multiple matches, in this case, the age of every match are read from the ULB-LSQ (one per cycle) and encoded into a per-byte bit vector (taking multiple cycles to generate this vector), these bits indicate which bytes should be forwarded to the load, multiple store forwarding is uncommon in many benchmarks, though.

This proposal also includes LSQ filtering optimizations [28], in conventional LSQs, as the number of in-flight instructions increases, the number of entries that have to be searched increases as well, so the number of searches can be reduced with the help of hash tables and bloom filters [29] [30], every load/store address is hashed to a single bit (with the help of these bloom filters), it is written (incremented) in a hash table, later, when a filtered load/store address checks this hash table and it is already set that bit, then there is likely to be a match in the LSQ, if it isn't set, then it is certain that there is not any possible match inside the LSQ, this search elimination technique is called Bloom filter predictor (BFP) (Figure 3-5). When a load/store instruction retires, then the BFP entry is reduced by one.

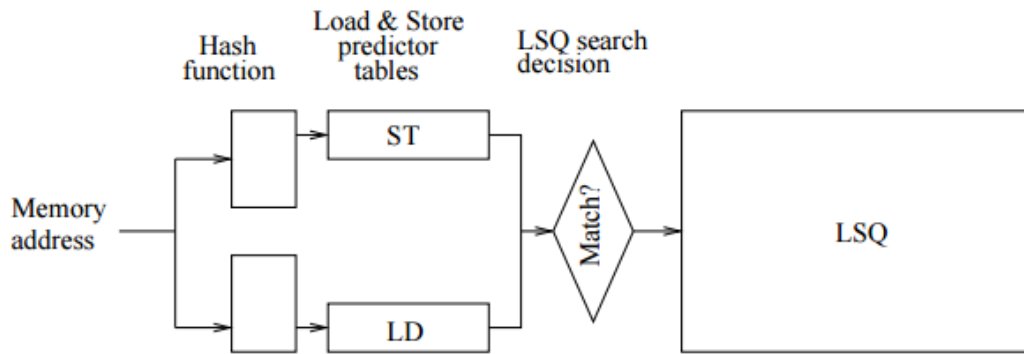


Figure 3-5. BFP Search Filtering

### 3.1.3 STORE SETS

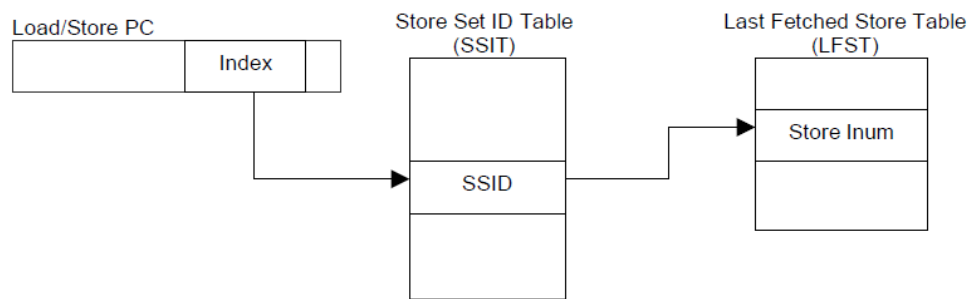
[31] In order to achieve a maximum performance in out-of-order processors, the load instructions must issue as soon as possible, this can be achieved with the help of memory dependence predictors which indicate whether a load instruction may have memory dependence with some of the in-flight store instructions or it should be issued freely, if a dependence violation occurs, then a recovery process must be taken and the memory dependence predictor is updated (dynamic predictors), the simplest memory dependence predictor is the “naïve” one (also known as “static” or “blind” predictor), this predictor indicates either that any load instruction may be dependent of any earlier store forcing it to wait for all the memory addresses of earlier stores to be known, or that non load instructions are dependent of any uncommitted store and they should issue as soon as possible.

In this research proposal, they achieved an improved memory dependence predictor which uses “store sets”, a store set is the set of the stores (identified by their PC) upon which a load has ever been dependent. They take as a baseline the assumption that historic behavior of memory-order violations servers is a good prediction for future memory dependencies.

The algorithm is the following: whenever a program begins, the store sets of all the load are cleared and a naïve prediction is temporally used, every time a dependence violation occurs, the store PC is saved in the store set of the implicated load, if another dependence violation occurs with the same load, that store PC is saved in that store set as well, then, the next time the load has to wait until those store instructions included in its store set issue. If the load has no occupied entries in its store set, it should issue as soon as possible.



Because the implementation for this store sets mechanism is hardware expensive, the modified implementation is achieved using two tables (*Figure 3-6*), a Store Set ID Table (SSIT) and a Last Fetched Store Table (LFST). Both load and store instructions access the SSIT based on their PC and get a store set identifier (SSID), when the SSID is valid it means that the load/store has a valid store set and with this SSID the LFST is accessed, an “inum” (hardware pointer which identifies an in-flight instruction) of the most recently fetch store instruction belonging to that store set is taken, then the memory access instruction must wait for that store instruction to issue. When a store accesses an entry in the LFST, it updates that entry with its own “inum”, later on, when it issues, it accesses again the LFST and if it is still its own “inum” then, it clears that entry.



*Figure 3-6. Implementation of Store Sets Memory Dependence Prediction*

When a recovery process (branch misprediction, jump misprediction or memory order violation) is taken, the aborted stores are simply marked as done in the LFST.

### 3.1.4 STORE VECTORS

[18] It is proposed an improved algorithm for memory dependence prediction (8.1% better performance than store-sets) based on store vectors, rather than tracking the program counters of dependent stores, the load-store dependencies are tracked based on the relative age of a store. Each store vector for any load has the relative positions of all the stores that enforced memory ordering violations in the past (*Figure 3-7*). There are three steps: “lookup/prediction”, “scheduling” and “update due to ordering violations”. The load-store dependencies are recorded in a data structure called “Store Vector Table”, it is indexed with the least significant bits of the load’s PC, then the store vector taken from the table is rotated and copied into the load scheduling matrix (LSM). The store vector is rotated such that the

least significant bits are aligned to the most recent store (resolved stores are cleared in order to prevent deadlocks), then, it is written into the LSM.

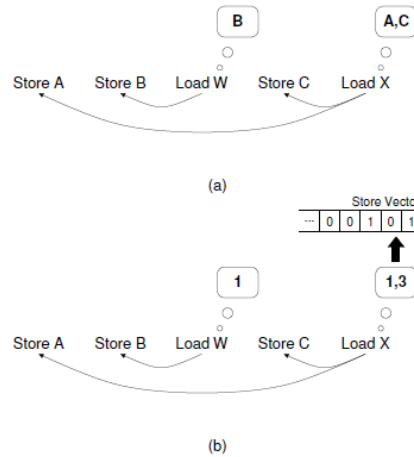


Figure 3-7. (a) Store-address tracking of dependencies, and (b) Age-tracking of dependencies

The position of the bits in the store vectors indicate which stores are predicted to have a memory dependency with each load, thus, every load has to wait for their predicted dependent stores to be resolved. When a store is resolved its bit is cleared from the store vectors (its corresponding column), and once a load has all its store vector's bits cleared, it is considered as ready, then it can be issued (*Figure 3-8*). Initially, the vectors are full of zeros and a naïve prediction is used, the memory ordering violations will fill the corresponding bits in these vectors, the contents of the SVT are periodically reset in order to clear possible inexistent predicted dependencies due to changes in program phases, dynamic data values, etc. In *Figure 3-9* is shown an example of the store vectors operation steps.

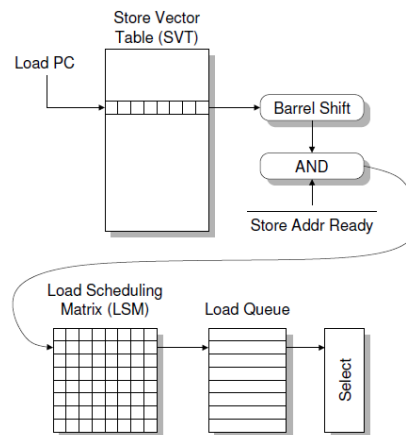


Figure 3-8. Store vectors data structures and interaction with a conventional load queue

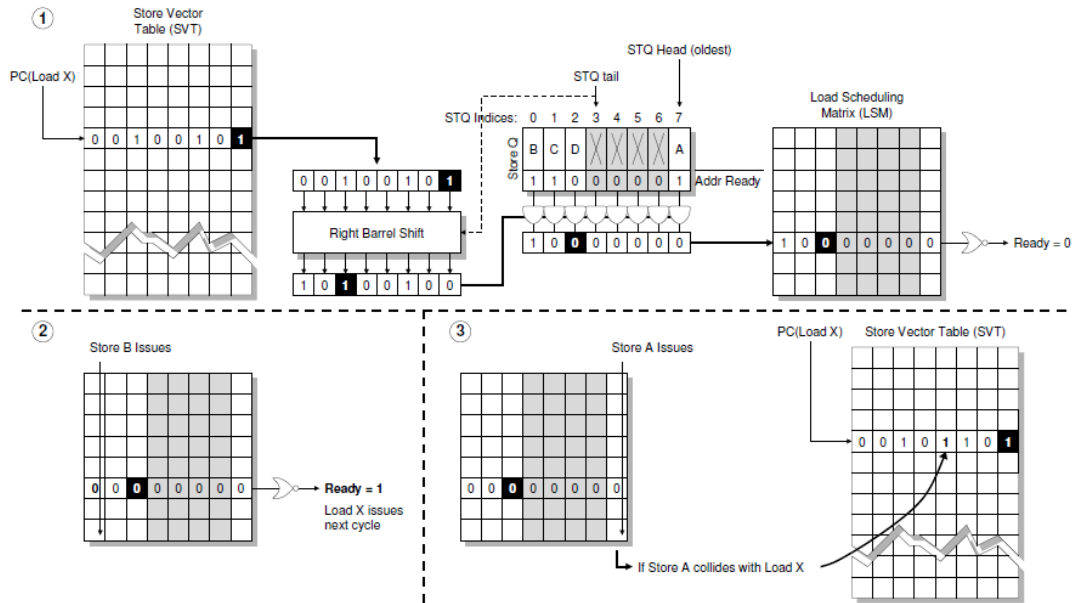


Figure 3-9. Example store vectors operation

### 3.1.5 STORE-TO-LOAD FORWARDING VIA STORE QUEUE INDEX PREDICTION

[16] In this store-load data forwarding proposal, rather than doing associative searches within the store queue, they incorporate a speculative indexed access. The address CAM is replaced by a simple decoder (*Figure 3-10*), in order to avoid this associative search (a power saving technique), they use a forwarding index predictor (based on store sets) to predict the most likely store index to be dependent with a load, then forward data from this store entry. For those loads that had a memory ordering violation, a delay index predictor is used, this delays the execution of a load until all non-predicted stores that may have a memory dependency have committed. The predicted SQ index is generated during the decode/rename pipeline stages, in this stage is either identified a dependency between a store or predicted a non-dependency, load issues only when its input registers are ready and when the store corresponding to its forwarding index has executed. In order to ensure memory ordering correctness, a load re-execution prior to committing may be made [14] (a violation is detected when a load's re-executed value is not equal as the first executed value) for those loads that executed with older unresolved stores.

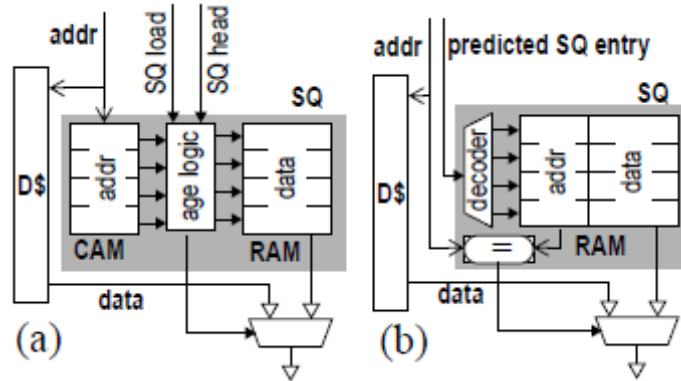
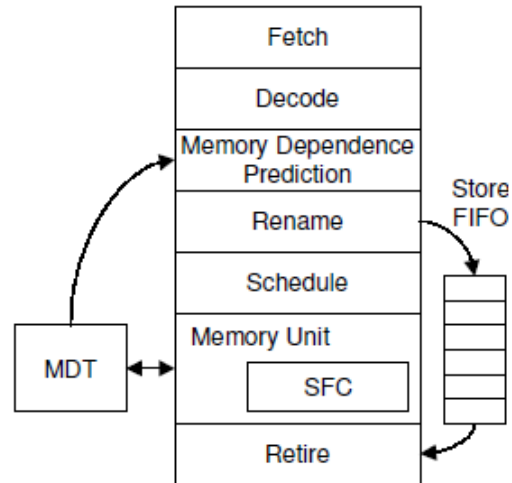


Figure 3-10. Store queues: (a) associative, (b) indexed

### 3.1.6 ADDRESS INDEXED MEMORY DISAMBIGUATION AND STORE-TO-LOAD FORWARDING

[17] Because store-to-load data forwarding and memory disambiguation require fully-associative, age-ordered searches, this yields to high latency as well as high dynamic power consumption. In this proposal, the store-to-load forwarding is done with help of a store-forwarding cache (SFC) and the memory disambiguation is resolved with a memory disambiguation table (MDT), both of them are not CAM structures, the SFC is accessed as a cache, speculatively and out-of-order, and the MDT requires sequence numbers in order to identify and recover from memory dependence violations.

The SFC stores the values of the store instructions as they complete, the loads may forward their values from this SFC (it is accessed in parallel with the L1 data cache) because SFC is accessed speculatively and out-of-order, memory ordering violations can occur, thus, MDT identifies all these memory ordering violations, if exist a memory ordering violations, the memory unit initiates a recovery mechanism. There is also a store FIFO which retires store instructions in order when they are dispatched, their data is written to the FIFO, as they execute, their address is written to their address field in this FIFO, and when they commit, its entry is cleared (*Figure 3-11*).



*Figure 3-11. Processor pipeline, store forwarding cache (SFC), memory disambiguation table (MDT) and Store FIFO*

Because there can be memory ordering violations due to the out-of-order accesses in the SFC, the MDT tracks the sequence numbers (in-flight tags) of loads and stores to each in-flight address, whenever it is identified a violation, a recovery mechanism is triggered. In this proposal, the store set predictor [31] is modified in order to reduce the pipeline flushed caused by memory ordering violations, the store set id table is replaced by a producer table and a consumer table, and the last-fetched store table is replaced by a last-fetched producer table. The updating process of this predictor table is: “When the MDT notifies the producer set predictor of a dependence violation, the predictor inserts a dependence between the earlier instruction (the producer) and the later instruction (the consumer) by placing the two instructions in the same producer set”. Merging producer sets is very similar to merging store sets. The MDT is address-indexed, if a load/store’s sequence number is later than the one found in the MDT entry, or if there is no valid load sequence number, then the entry takes the sequence number from this load/store. Otherwise, if it is earlier, then a memory ordering violation is identified.

# Chapter 4

---

## PROPOSED LOAD/STORE QUEUE DESIGN

---

The main contribution of the thesis is the design of a LSQ for the Lagarto II processor which fetches two instructions per clock cycle (*Figure 4-1*), a conventional Load/Store Queue with in-order execution design was used as a base model and is described in detail in the first subsection of this chapter, then a modified design with an out-of-order execution and the energy saving techniques implemented is described in the second subsection.

---

### 4.1 LOAD/STORE UNIT WITH IN-ORDER EXECUTION

This first design of a LSQ with in-order execution was taken as a base model in order to be modified to the out-of-order design, both designs get from Dispatch stage up to two instructions per cycle and can issue up to one instruction per cycle. The main building blocks in this first design are the “Ready Bit Register”, “WakeUp Logic”, “Select Logic”, “Register Files”, “Address Computation Unit”, “Tag Bus”, “Destination Bus”, “Bypass Logic”, “Forward Logic”, “Reorder Buffer (Dispatch, Issue & Execution flags)” and the “Memory Access mechanism” (*Figure 7-3*).

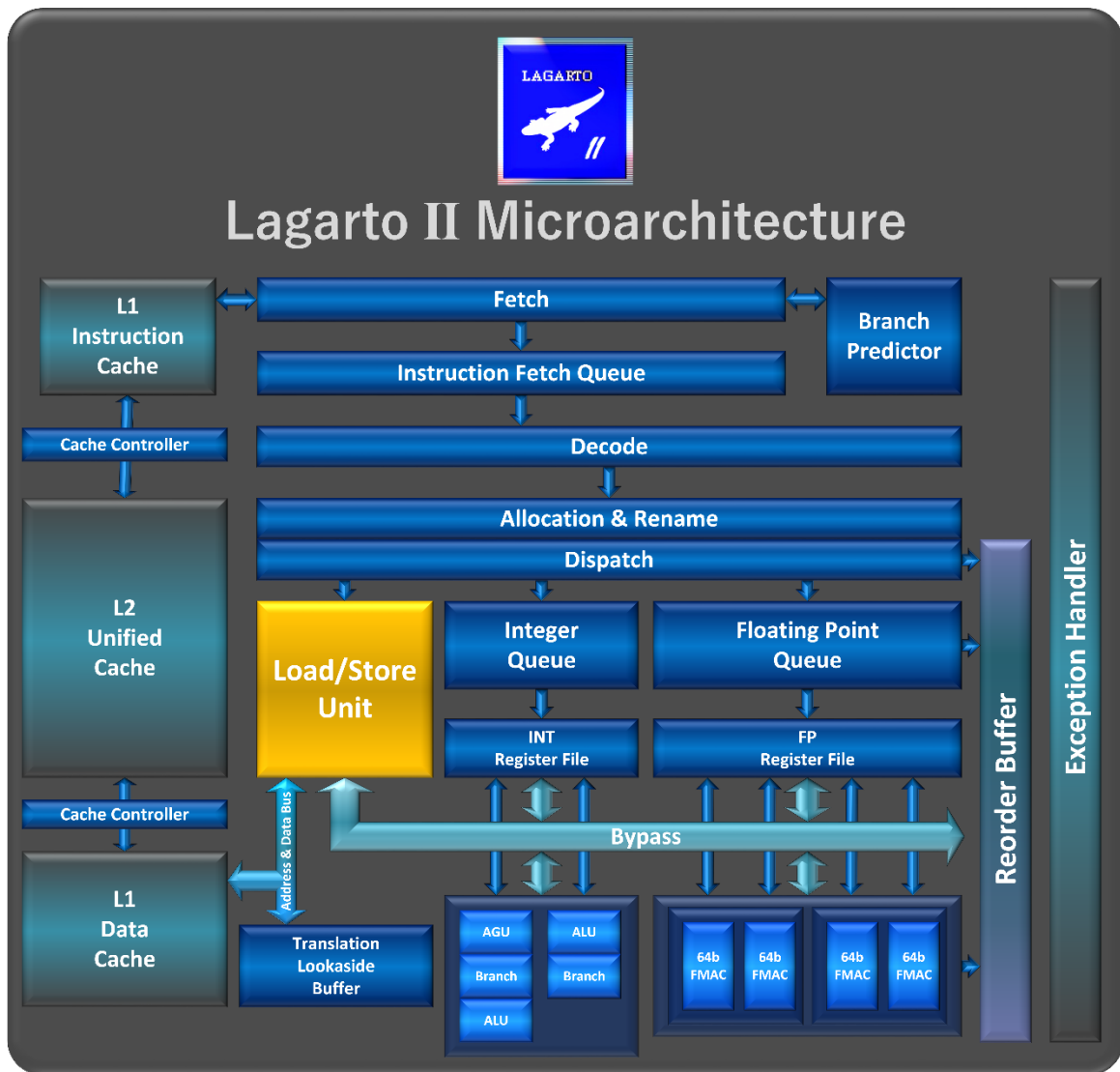
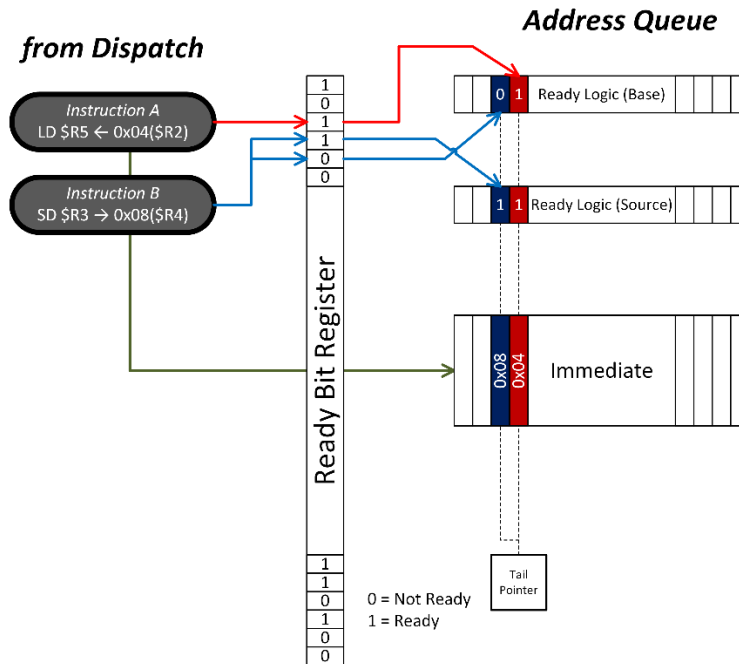


Figure 4-1. Lagarto II Microarchitecture

#### 4.1.1 IN-ORDER ADDRESS COMPUTATION PIPELINE

The whole pipeline of this Load/Store Unit is divided into Address Computation (also known as the Address Queue) and Memory Access (also known as the Load/Store Queue), the Address Computation section is where the instructions wait for their source operands to be ready in order to either bypass their source operands values from the Bypass network or read them from the Register Files, then the Address Computation Unit can calculate their memory addresses adding a base operand and a signed immediate value .

An instruction is written into the LSQ when its entry has been assigned in the queue (the Tail Pointer assigns this entry in the ROB) and is dispatched (its flag in the Dispatch-ROB entry is then set to “1”) from the Front-End, its source registers check the Ready bit vector register (a building block which keeps track of the value generation for every register) in order to know if their values have been already generated by earlier instructions, the Ready Bits for the ready registers are set to “1” and written in its corresponding instruction queue entry, if their values aren’t generated yet, then a “0” is written (*Figure 4-2*). Also, the Immediate value, the control bits and the physical register tags are stored in the queue (the source and destination tags are written in the Memory Access section).



*Figure 4-2. Ready Bit Assignment Example*

The associative WakeUp is done by comparing the un-ready base register tag from each entry in the queue with the register tags sent through the Tag Bus every cycle, if there are any matches, the corresponding ready bits for those registers are then set to “1” (*Figure 2-19*). The Select Logic waits until the queue entry pointed by the Head Pointer (*Figure 4-3*) has its base register “ready” in order to issue that instruction (the Head Pointer is incremented in order to point to the next instruction and its flag in the Issue-ROB entry is set to “1”), the Integer Register File is read and the Bypass network is accessed, then the memory address is computed by adding the base register data with the sign-extended immediate value (*Figure 4-4*). The generated memory address is then stored in its corresponding entry in the Memory Access section of the LSQ.



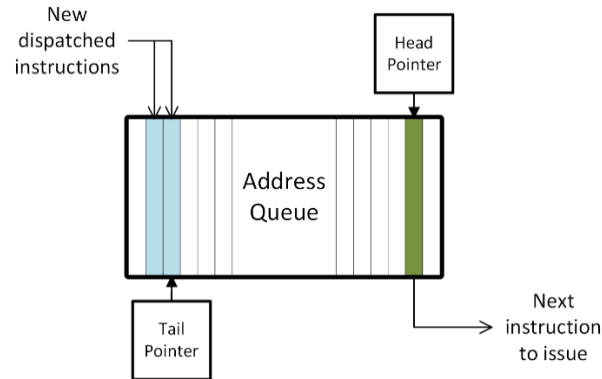


Figure 4-3. Head Pointer and Tail Pointer

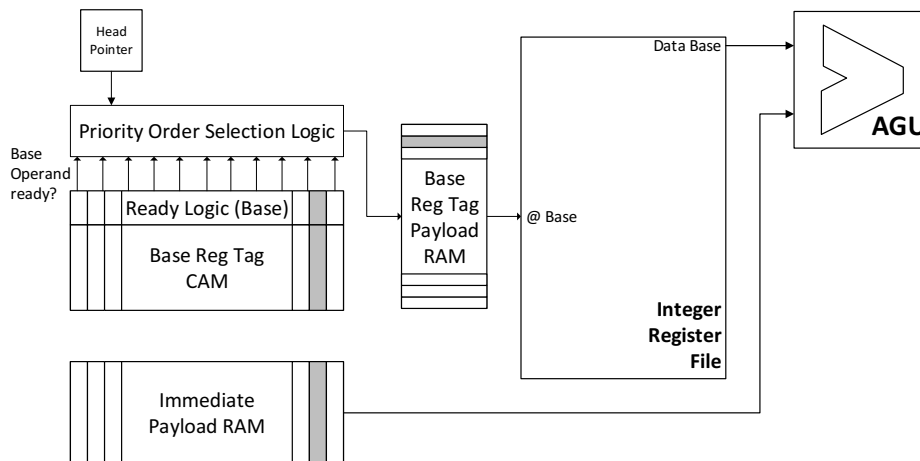


Figure 4-4. Select Logic and Address Generation Unit

#### 4.1.2 IN-ORDER MEMORY ACCESS PIPELINE

In this section of the LSQ are stored the memory address, the Load/Store bit (it serves as a read/write enable for the Cache Memory), the Source data (the data to be written to cache memory by a store instruction), the Destination register tag (the register to be updated with the data read from cache memory by a load instruction) and the Destination data (the data read from the cache memory by a load instruction) for every dispatched memory access instruction; also, the cache memory is accessed by the memory access instructions selected to be executed. When a store instruction is dispatched and enters the LSQ, its source register tag (it points to the physical register containing the data that will be written to cache memory) accesses the Register File, the Bypass Network and the Ready bit Register in order to know if

it has been already generated, if it has, the data taken from the Register File (Integer or Floating Point) or the Bypass network has the newest value for that physical register, then it is written to its corresponding entry in the Memory Access section of the queue, if it hasn't been generated, a "0" is written in the ready bit field of that source register and the source tag is stored in the Forward Logic CAM, this CAM is similar to the WakeUp CAM, the un-ready register tags in the Forward Logic CAM are compared with the tags sent through the Destination Bus, and, when there is a match, the source data is forwarded to its corresponding data field entry of the queue (*Figure 4-5*).

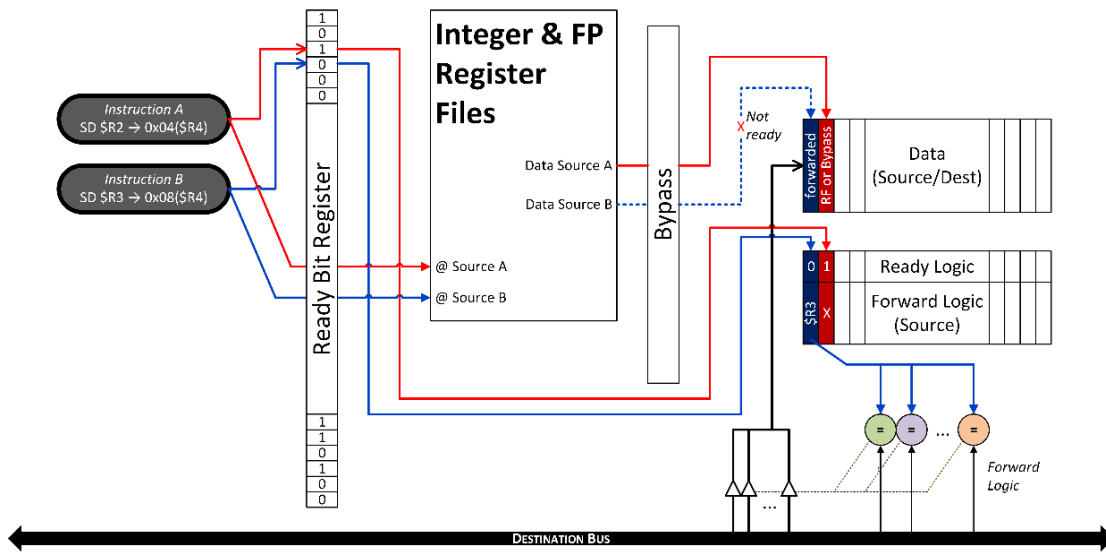


Figure 4-5. Forward Logic Example

The memory addresses generated in the Address Computation section of the queue are stored in the "memory address" field of this Memory Access Queue section, the select logic checks the entry pointed by the Head Pointer and sends to execution a load if it has its memory address already computed, the stores are sent to execution only at commit cycle and it is also needed its source data to be already forwarded, taken from the Register Files or bypassed. In order to access the cache memory, the L/S Bit serves as a Read/Write enable for the cache memory, in the case of a load, the data loaded is sent through the Destination Bus and written in the corresponding Register File (accessed by its Destination Register tag), and for a store, the Source Data is stored in cache memory accessed by its memory address (it is written first in the store buffer in order to don't generate waiting cycles due to cache hit misses). *Figure 4-6* and *Figure 4-7* show the memory access for both Load and Store instructions.

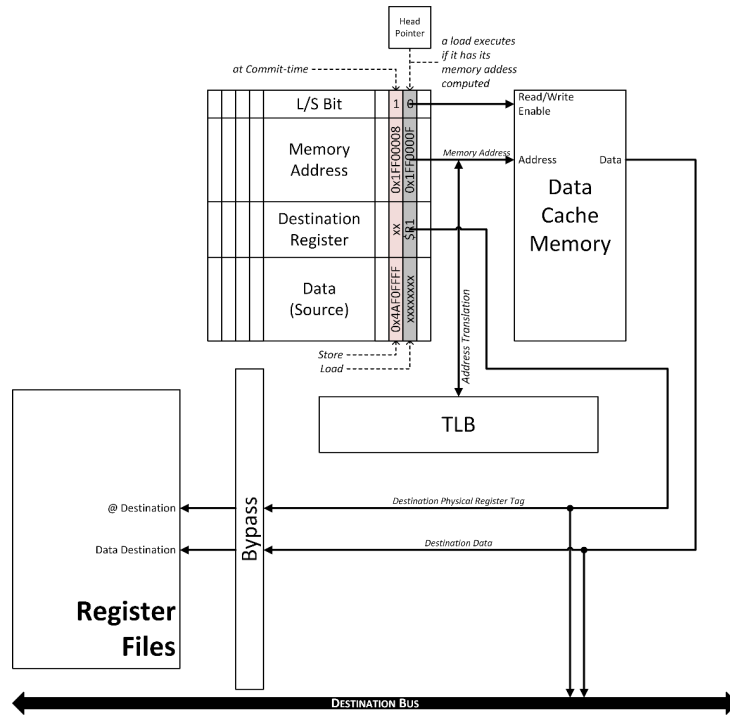


Figure 4-6. Memory Access (Load)

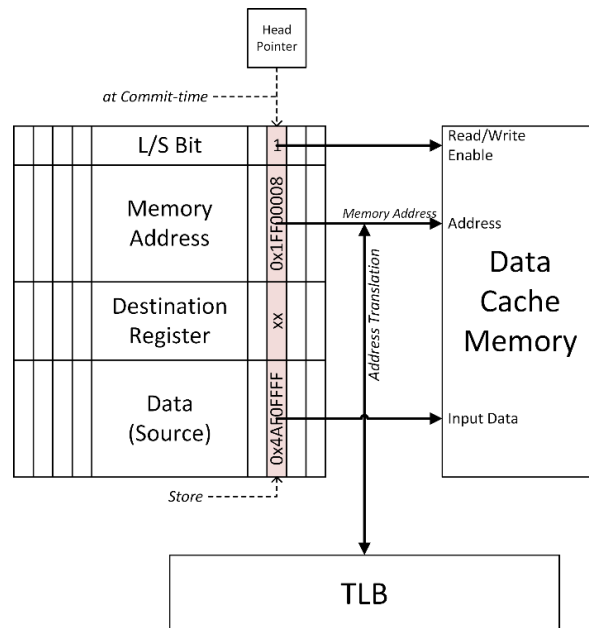


Figure 4-7. Memory Access (Store)

---

## 4.2 LOAD/STORE UNIT WITH OUT-OF-ORDER EXECUTION

In-order LSQ designs requires less complex logic due to the in-order select mechanism which only checks the entry pointed by the Head Pointer, and since there is no out-of-order execution, memory dependency violations can't occur (a memory disambiguation scheme isn't necessary), but, the drawback is that ready non-dependent instructions can't issue nor execute if there are earlier instructions that haven't issued/executed yet, leading to an overall low performance.

This proposed LSQ design comprises a power-efficient memory disambiguation scheme using a Hash table and Bloom filters [30] as well as a novel low dynamic power consumption store-to-load data forwarding and memory address disambiguation using "Non/Possible Dependencies Vectors", the LSQ is divided into three logical queues, the Address Queue, the Load Queue and the Store Queue (*Figure 4-8*), being the Address Queue the section (Address Computation section) where the instructions remains in their reservation units until their base registers are ready in order to compute its memory addresses in the Address Computation Unit (ACU) (also known as the Address Generation Unit), whether the Load Queue and Store Queue belong to the Memory Access section, in the Load and Store Queues are carried the memory accesses, the store-to-load data forwarding and the memory disambiguation mechanism. The Instruction Queue using a Block Mapping Table [1] was adapted to the Address Queue in order to reduce the number of comparisons carried by the WakeUp Logic, it also inspired the idea of the Non/Possible Dependencies Vectors design reducing the amount of comparisons carried by the Memory Disambiguation Scheme and the Store-to-Load Forwarding Logic.

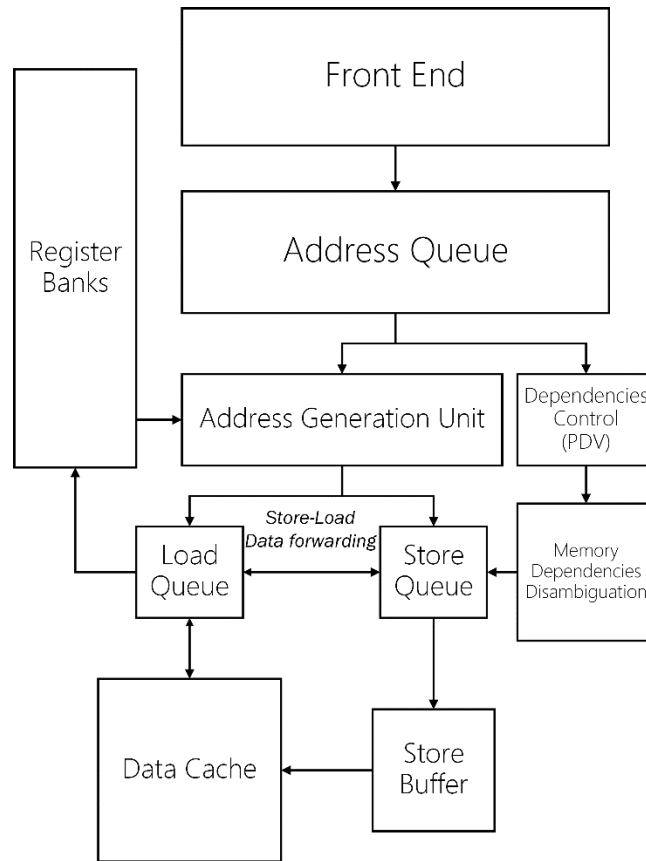


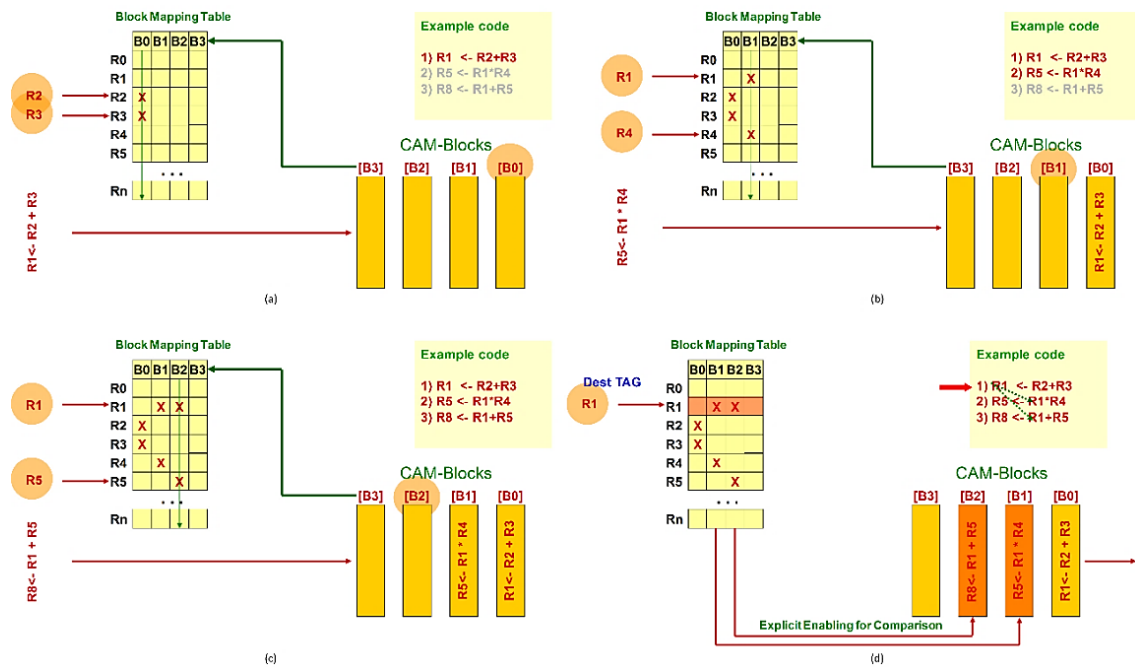
Figure 4-8. Proposed LSQ General Pipeline

#### 4.2.1 UNKNOWN/KNOWN MEMORY ADDRESS, BLOCK MAPPING TABLE & ADDRESS GENERATION

Every memory access instruction is classified into any of these two memory address-status categories, a *known address* or an *unknown address*, being the Address Queue the logical structure with the unknown address instructions stored in it, whether the instructions are stored in either the Load or the Store Queue. Since a memory address isn't known until being computed by the Address Generation Unit (each instruction waits in its reservation stations until it has its base register "ready" to be read from the Integer Register File or to be bypassed from the Destination Bus), there must be extra careful with the memory access instructions executed out-of-order, thus, a memory disambiguation scheme must be used in out-of-order designs.

The Address Queue implemented in this thesis project uses an adaption of the “Low-Energy Instruction Wakeup Mechanism” [1], the whole queue is divided into four blocks and the comparisons carried out between every base register tag of the “unready” entries and the register tags sent through the tag bus, are directly enabled or disabled by a Block Mapping Table (BMT). *Figure 4-9* shows an example of the Block Mapping Table functionality, the “unready” source operands for the three instructions are marked in the BMT (*a, b, c*) in the moment they are stored in the CAM-Blocks and the value for their source operands are “unready”, this BMT indicates which blocks require the physical registers sent through the Tag Bus, then, whenever a register tag is received, the BMT enables only the required comparisons and disables the remaining blocks (*d*).

*Figure 4-10* illustrates how the Address Queue’s CAMs & RAMs are organized along the BMT.



*Figure 4-9. Block Mapping Table Functionality Example*

When an instruction first enters the Load/Store Unit (Address Queue, Load Queue & Store Queue), it still does not have its memory address calculated, thus, it is sent to the Address Queue, belonging to the unknown address status, it is assigned an entry in the Load or Store Queue as well. In order to implement (for both CAM & RAM) four single-port memories, the entries along the queue are organized as shown in *Table 5*, needed to be included some write enabling control and multiplexers intended for supporting the logical two-input ports for the two input instructions (*Table 6*). This logic control makes use of the Tail Pointer in order to

determine in which block the instructions are stored, avoiding to implement multi-port memories.

*Table 5. Entries Organization inside the Queue*

LSB	MSB	DECIMAL	# BLOCK
00	000	0	Block 0
	001	4	
	010	8	
	011	12	
	100	16	
	101	20	
	110	24	
	111	28	
01	000	1	Block 1
	001	5	
	010	9	
	011	13	
	100	17	
	101	21	
	110	25	
	111	29	
10	000	2	Block 2
	001	6	
	010	10	
	011	14	
	100	18	
	101	22	
	110	26	
	111	30	
11	000	3	Block 3
	001	7	
	010	11	
	011	15	
	100	19	
	101	23	
	110	27	
	111	31	

*Table 6. 4 Blocks-RAM/CAM write enables and multiplexers between two input instructions*

Tail Pointer	Enable Block 0	Enable Block 1	Enable Block 2	Enable Block 3	Instruction 0/1 Block 0	Instruction 0/1 Block 1	Instruction 0/1 Block 2	Instruction 0/1 Block 3
00	1	1	0	0	0	1	0	0
01	0	1	1	0	0	0	1	0
10	0	0	1	1	0	0	0	1
11	1	0	0	1	1	0	0	0
BASE POINTER	ENABLES				MUX			

In the BMT are marked only the “unready” operand tags needed by every block, and are cleared every time the operand tags are load received from the Tag Bus.

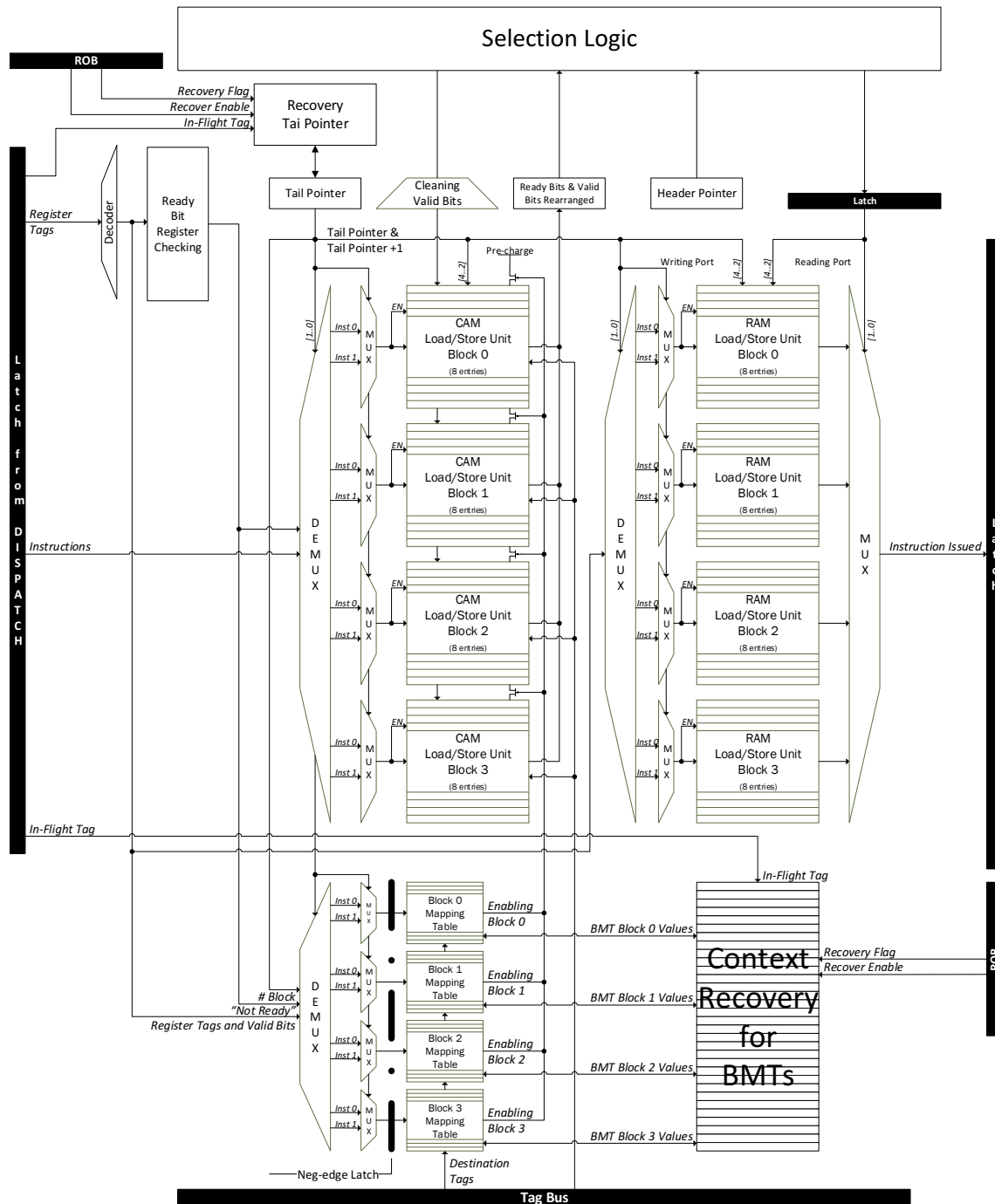


Figure 4-10. Address Queue (inside the L/S Unit) & the Block Mapping Table



The implemented design for the Queue's Ready Bit Logic is shown in *Figure 4-11*. It reduces the dynamic power by setting the unnecessary comparison's register tags to "0" so that it won't change until the ready bit is updated with a new dispatched load/store instruction.

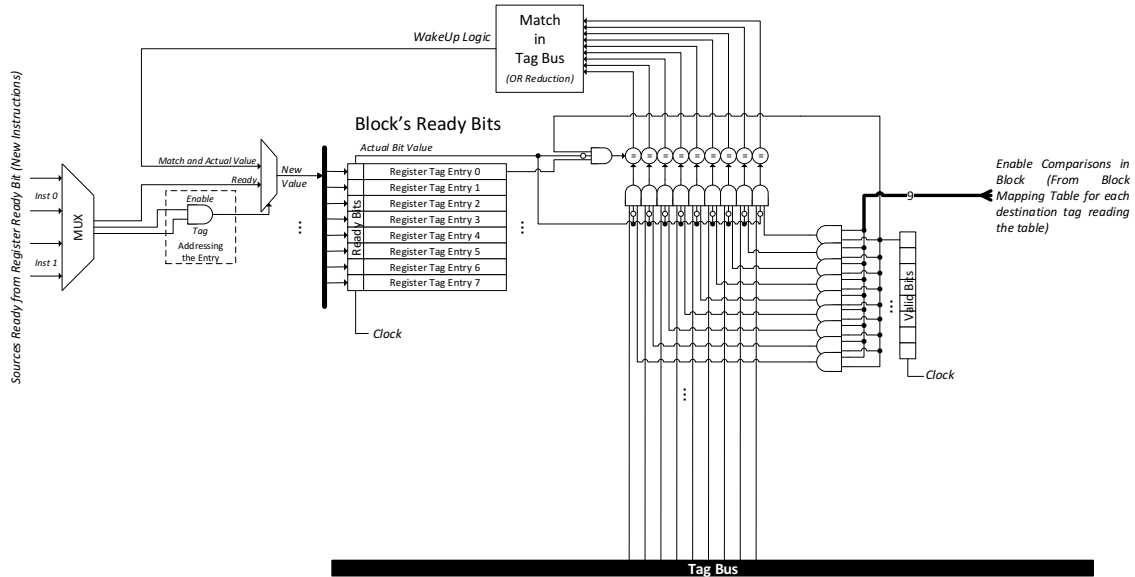


Figure 4-11. Queue's Ready Bit Logic

#### 4.2.2 OUT-OF-ORDER SELECT LOGIC

In order to prevent deadlocks and to support an out-of-order selection, it was adapted a Leading Zero Counter [32], achieving a fast low power priority order selector (*Figure 4-12*). This circuit is implemented for both "oldest first" and "youngest first" required priority order selectors, the "oldest first" mechanism is used in the Select Logic for the Address Queue and Load Queue (the Select Logic for the Store Queue is carried by the Commit mechanism), whether the "youngest-first" mechanism is used in the Store-to-Load Data Forwarding, this is because it is needed to forward the most updated data value (load instruction-relative) for every memory location whenever this mechanism is used. The circuit used in order to perform this out-of-order selection is illustrated in *Figure 4-12*, and the whole selection pipeline (incorporating the Address Queue, the Address Generation Unit, and the Possible Dependencies generation) is shown in *Figure 7-5 in Appendix-A*.

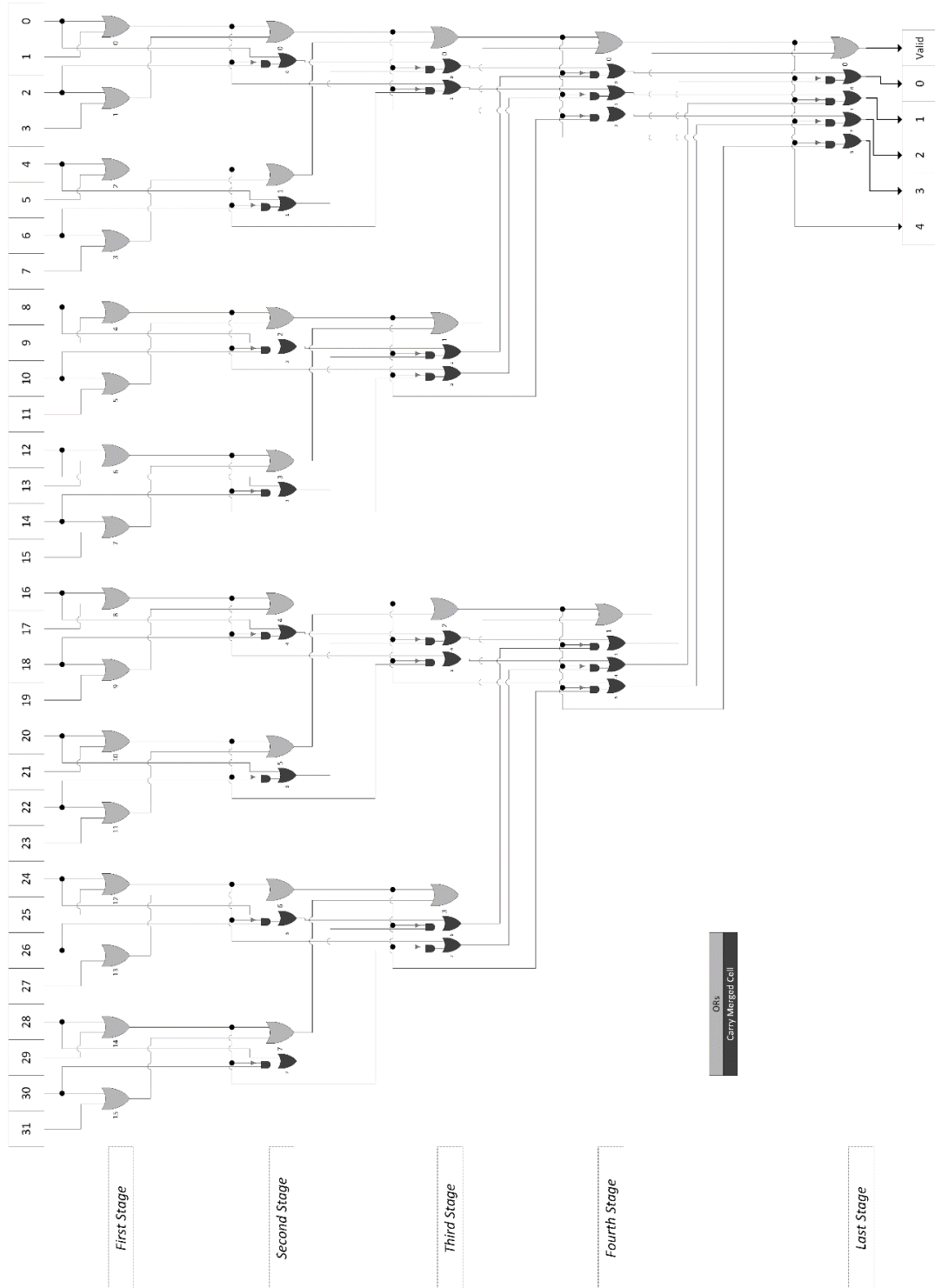


Figure 4-12. Leading Zero Counter as a Priority Selector (oldest-first) – 32bits

4.2.3 MEMORY DISAMBIGUATION & STORE-TO-LOAD DATA FORWARDING MECHANISM

The memory disambiguation and store-to-load data forwarding mechanism proposed in this thesis are constituted by “non/possible dependencies vectors”, “hash table and bloom filters” [29] [30].

A) Non/Possible Dependencies Vectors

The non/possible dependencies vectors are dynamic vectors used to enable/disable several memory address comparisons carried between the issued loads and stores, the “un-rotated” selection bits given by the Select Logic in the Address Queue are used to index a Possible Dependencies Memory, this memory’s output is a vector mask which after a combinational logic operation between the valid/busy bits and the load/store bits, generates the Non/Possible Dependencies Vector. *Figure 4-13* shows the generation of this vector.

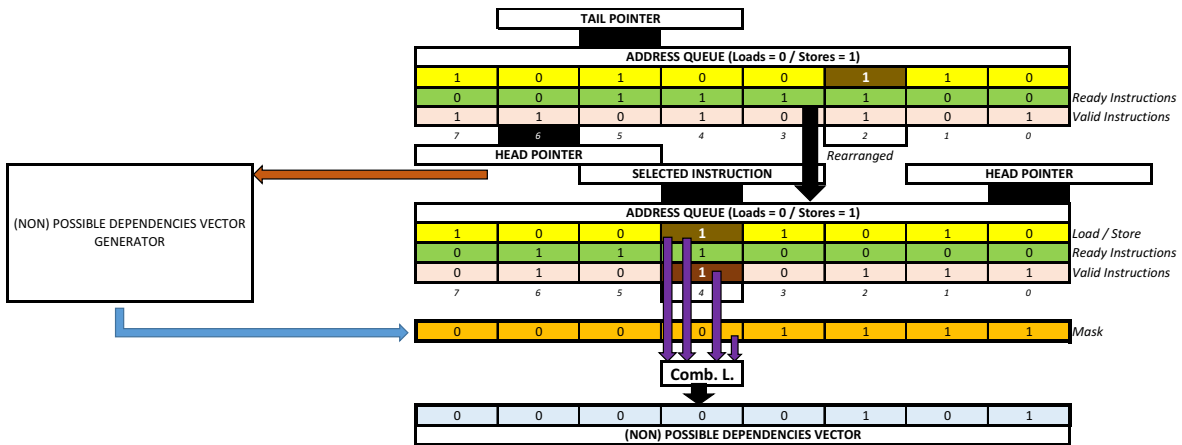


Figure 4-13. Non/Possible dependencies vector generation

A table test is illustrated in *Figure 4-14..25*, it shows how the non/possible dependencies vectors along with the hash table mechanism identifies memory order violations while it reduces the amount of memory addresses comparisons carried between the loads and stores achieving a store-to-load data forwarding effectively. For every memory order violation, the memory dependence predictor has to be updated. When a store issues, its memory address is written (incremented the counter in “1” after the bloom filter reduction) in the hash table, the un-issued load instructions found from its entry position in the queue to the head pointer are marked as “1” in its non-dependencies vector, these marked loads disables the

comparisons between this issued store. In the other hand, when a load issues, it searches the hash table for a possible memory address dependency, if it matches, then the store queue is accessed, if not, it is written directly to the load queue without executing any comparison, the un-issued store instructions found in the queue from the load's entry and the head pointer are marked as "1" in its possible dependencies vector", these marked stores enables the comparisons between that load and the un-issued marked stores. When a store issues, if a match has occurred with any younger load, the load's possible dependencies vector is updated and all the bits from that store's position are cleared to "0", if there isn't any match, just the entry of the store is cleared in the vectors. A load is solved (it means that it has no more memory dependencies left) when the bits in its possible dependencies vectors are all set to "0".

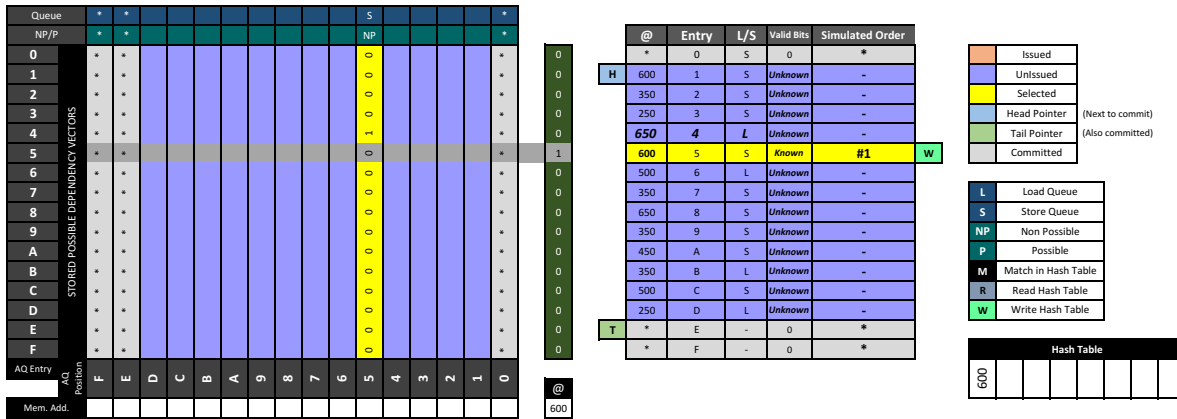


Figure 4-14. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (a)

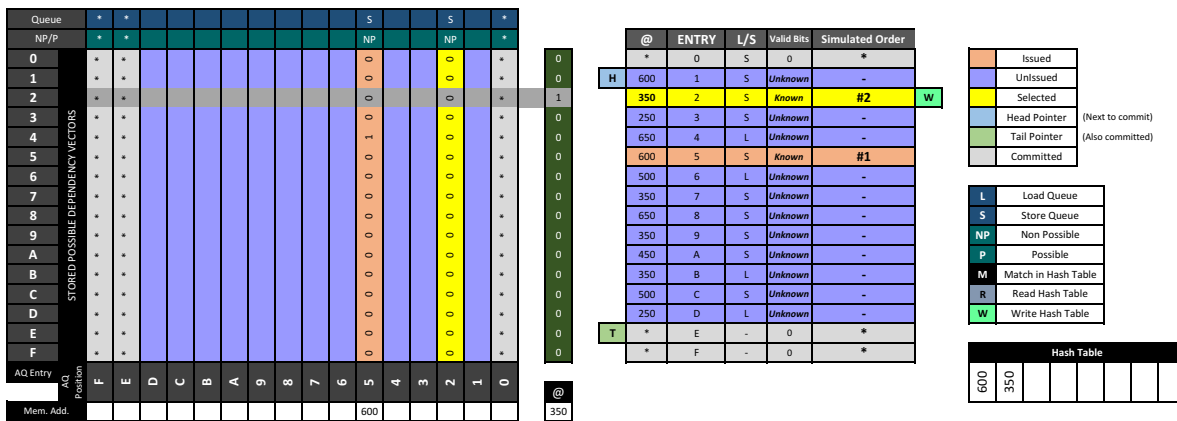


Figure 4-15. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (b)

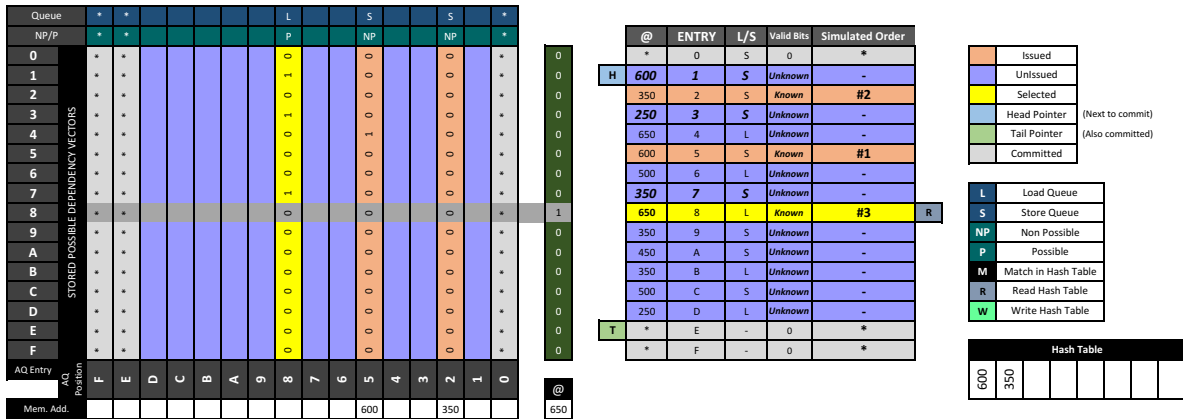


Figure 4-16. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (c)

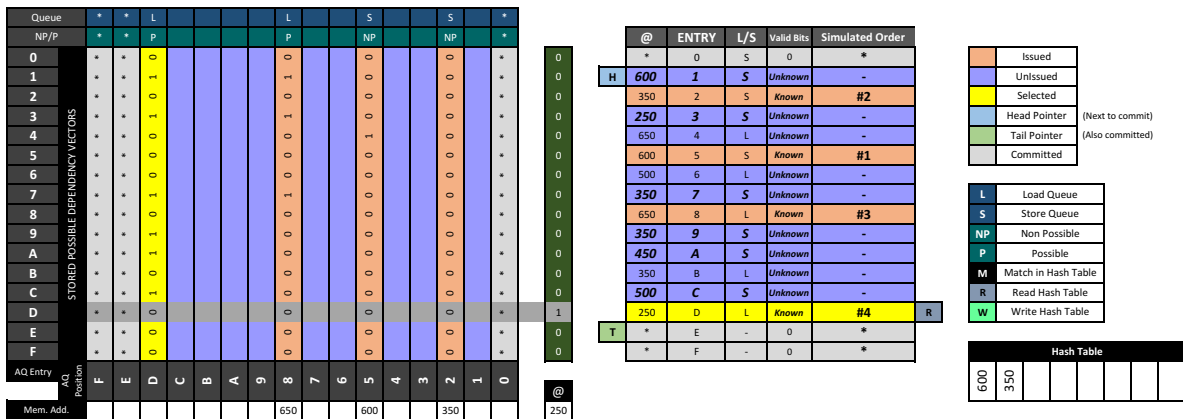


Figure 4-17. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (d)

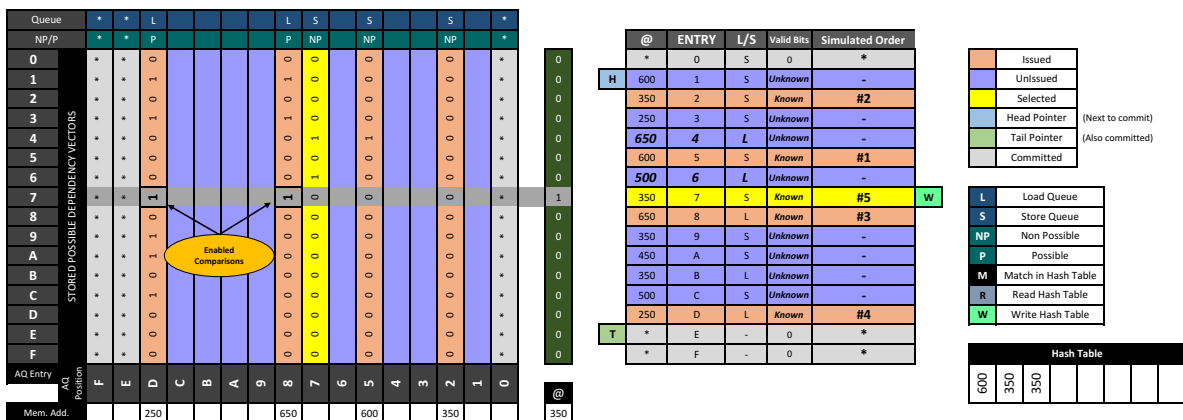


Figure 4-18. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (e)

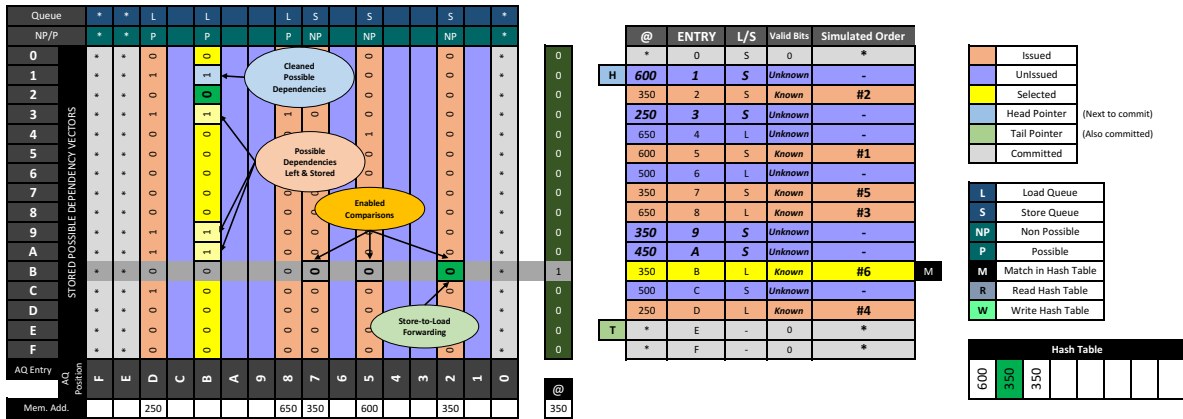


Figure 4-19. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (f)

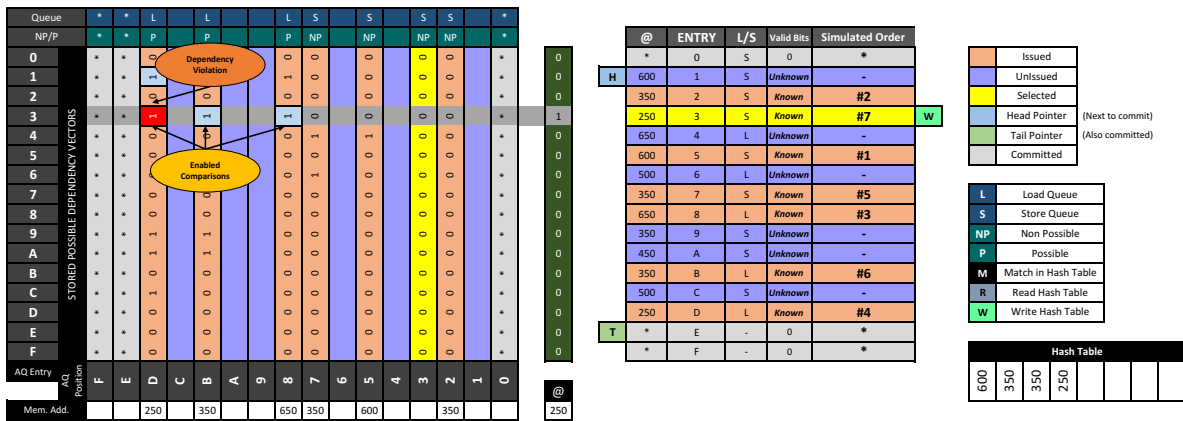


Figure 4-20. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (g)

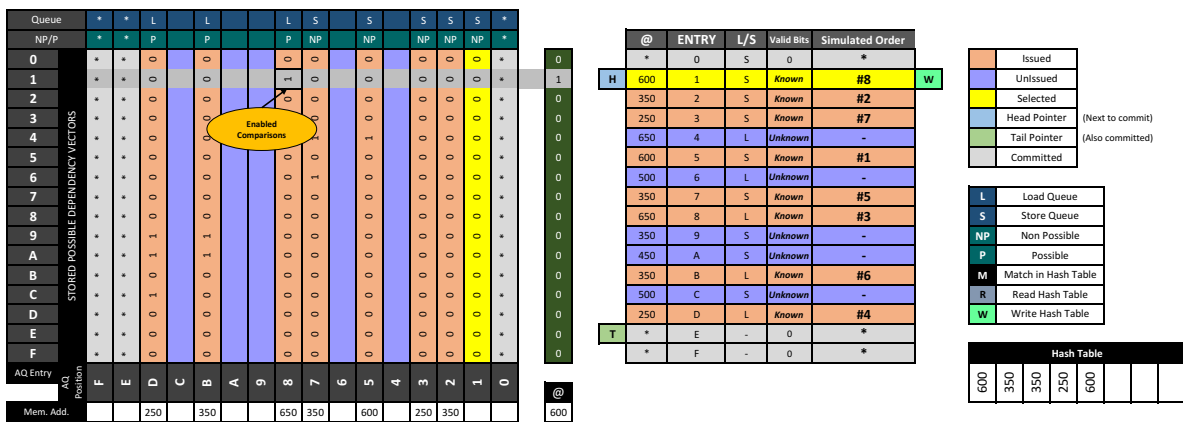


Figure 4-21. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (h)

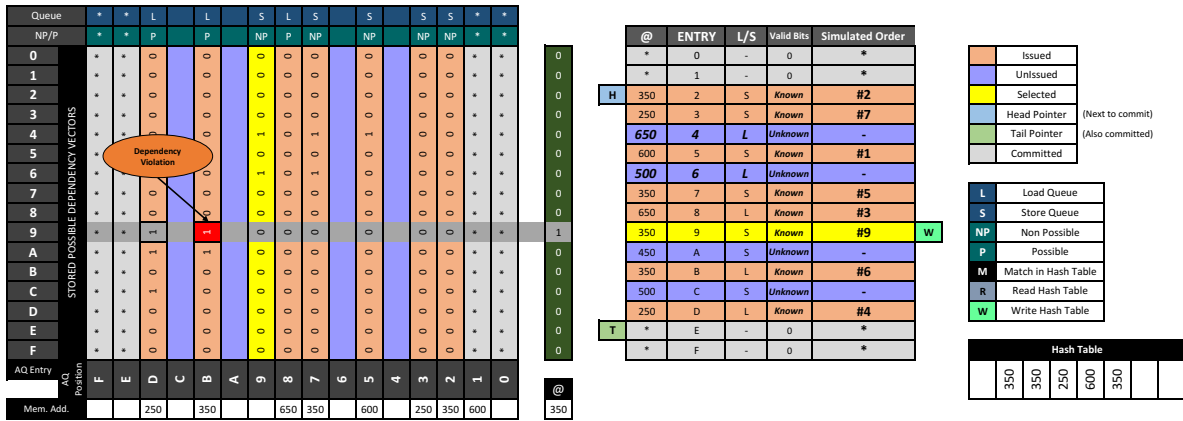


Figure 4-22. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (i)

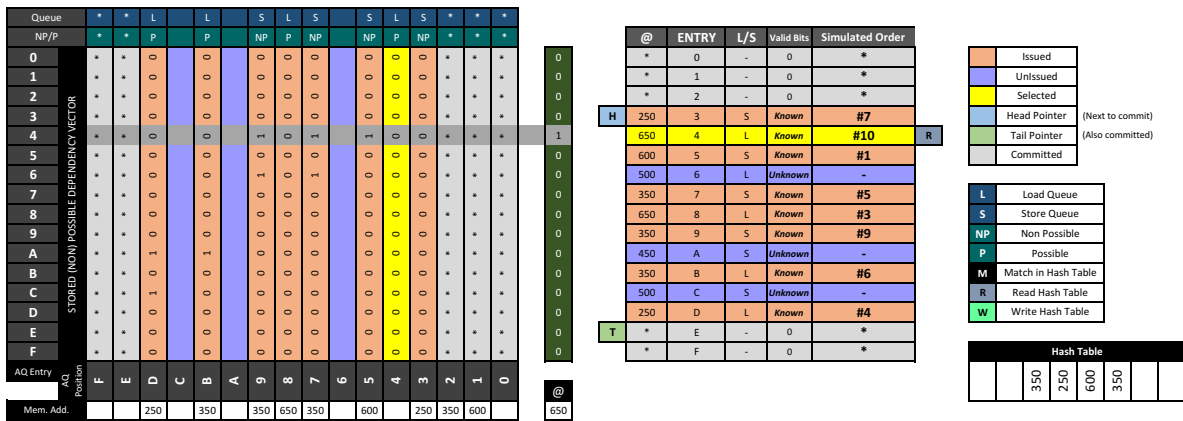


Figure 4-23. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (j)

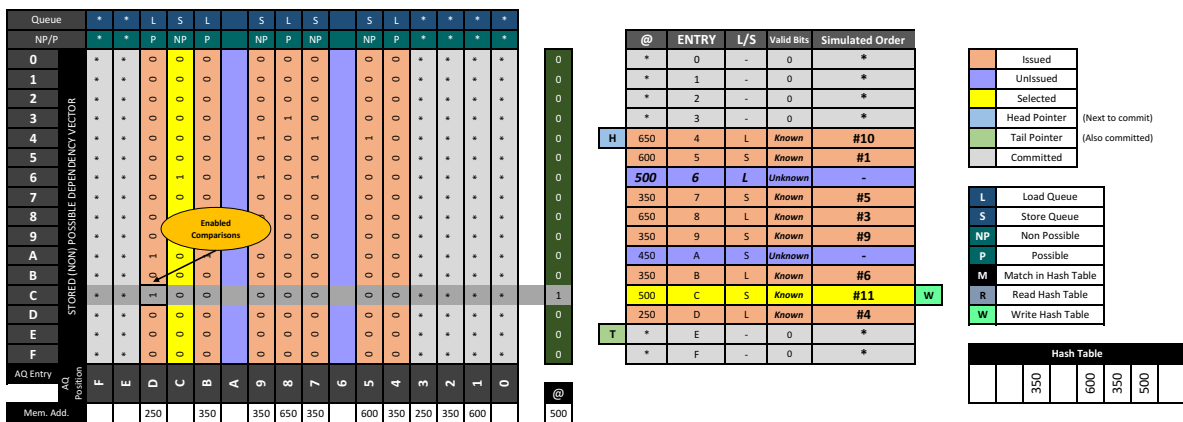


Figure 4-24. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (k)

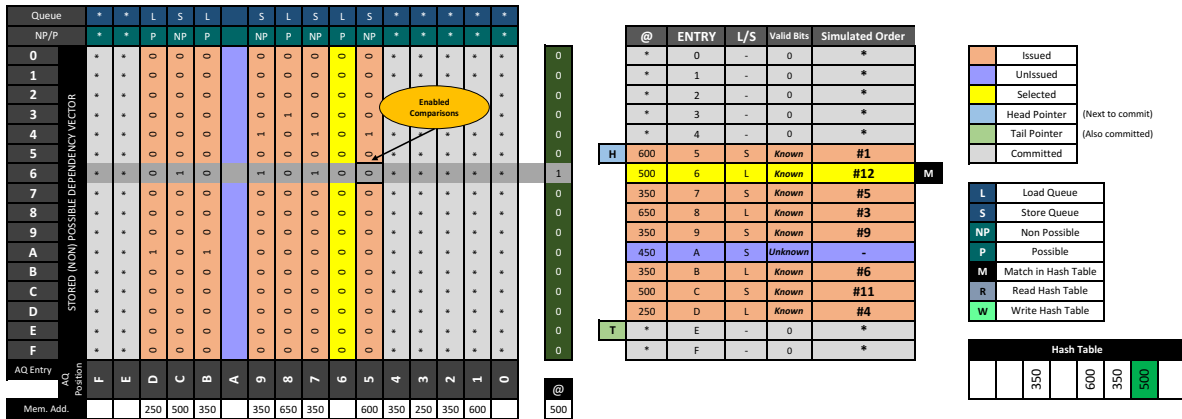


Figure 4-25. Example for the Memory Disambiguation & Store-to-Load data forwarding using the proposed Non/Possible Dependencies Vectors & Hash Table (I)

A desk evaluation for every Load/Store combination between three consecutive load/store instructions and its memory disambiguation scheme is provided in Appendix-B.

### Hash Table & Bloom Filters

The Bloom filters along with the hash table function as a reduction filter operation, taking the 40 LSB bits of each load/store’s memory address, it is applied this reduction in order to turn it into only 5 bits incrementing one of the 32 counters found in the hash table. When a memory address is known just after the issuing stage, it passes through a bloom filter which uses a three-phase XOR operation within its 40 less significant bits  $\rightarrow 40 \div 2 \div 2 \div 2 = 5$  bits, then with these bits the hash table is accessed (32 locations), for every location there is a 5 bit counter (counting from 0 to 32, one for every queue entry), if a store accesses these counters a “+1” operation is executed in the accessed counter, whether a load only reads these counters and if a “[1-32]” is read, then it means that it is possible to be the complete 40 bits memory address in the store queue, thus it enters to the Store-to-Load Forwarding Logic Mechanism.

### 4.2.5 STORE-TO-LOAD FORWARDING MECHANISM AND DELAYED SOURCE DATA FORWARDING

Due to the fact that a store can compute its memory address before its data value is generated by an earlier instruction, a “waiting cam table” containing the waiting store queue’s



entries is included in the load queue, when a Forwarding mechanism is executed but this source data is not ready, the entry of the required store's source data is stored in this waiting CAM, then a forwarding is performed whenever the Store Queue's Source Data Forward Logic (*Figure 4-5*) sends the forwarded data through the "Delayed Source Data Local Bus", the design is shown in *Figure 7-2*.

# Chapter 5

---

## RESULTS

---

This design has been implemented and simulated using the Altera's Quartus II EDA software getting an overall working frequency of 88.79MHz (*Figure 5-1*) making use of the Altera's DE2-115 FPGA board, the Front-End data path also is included in order to perform these simulations. The results are presented as follows, the simulation of the Front-end, the generation of the Possible Dependencies Vectors, the selection logic, the tests within the Block Mapping Table and the whole RTL diagram for the implementation.



The image shows a screenshot of the Quartus II software interface. At the top, there are two window titles: 'Compilation Report - AddressQueue' with a close button, and 'CAM\_RegTag.v'. Below the titles is a blue header bar with the text 'Slow 1200mV 85C Model Fmax Summary'. Underneath is a table with five columns: an index column, 'Fmax', 'Restricted Fmax', 'Clock Name', and 'Note'. The first row of the table contains the values '1', '88.79 MHz', '88.79 MHz', 'Clock', and an empty cell.

	Fmax	Restricted Fmax	Clock Name	Note
1	88.79 MHz	88.79 MHz	Clock	

*Figure 5-1. Maximum Overall Frequency*

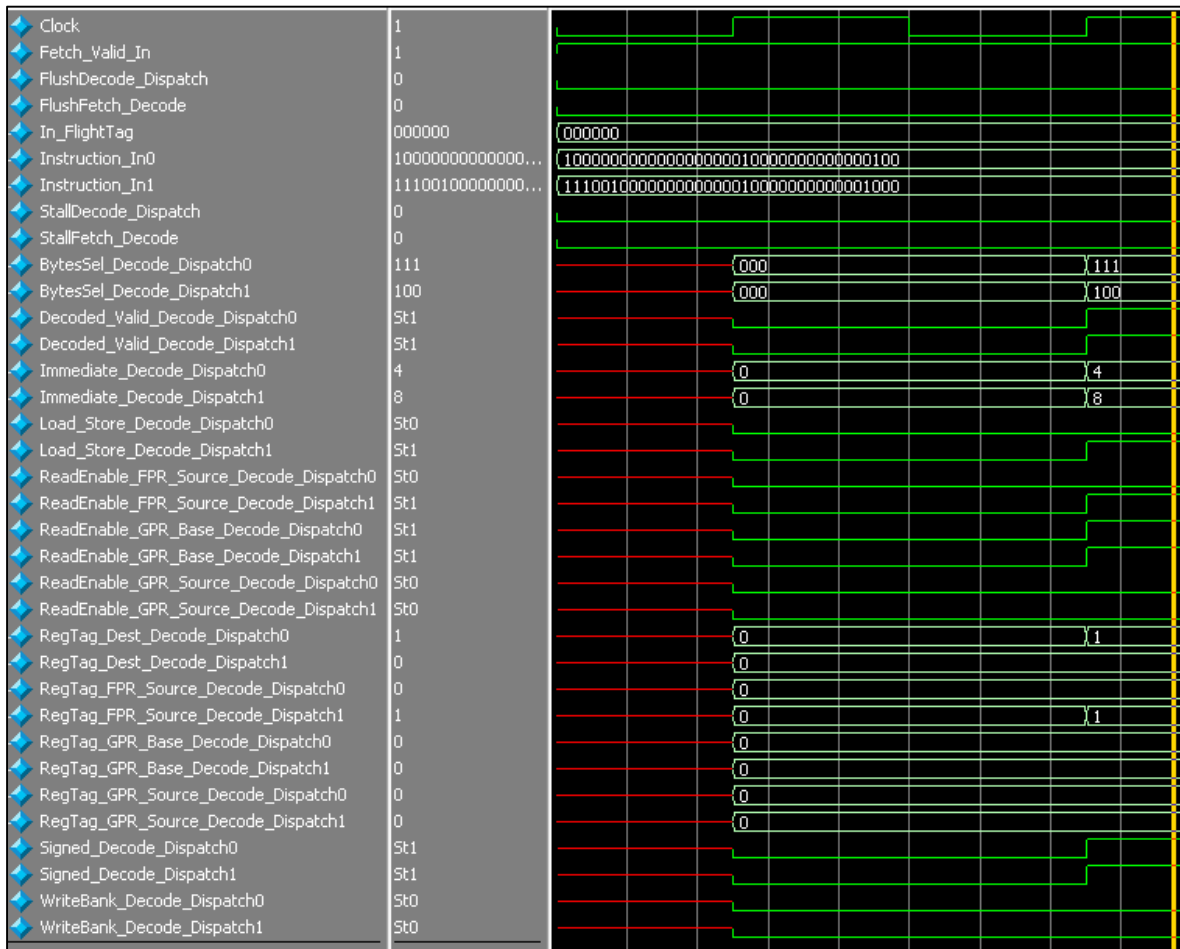


Figure 5-2. Decoder

The Decoding circuit reads the base, source and destination registers, it also generates the control bits-vector used to execute the load/store instruction through the whole datapath. In this simulation two instructions (*Instruction\_In0* and *Instruction\_In1*) are dispatched, they correspond to: *LB \$R1<-4(\$R0)* and *SWC1 \$R1->8(\$R0)* respectively.

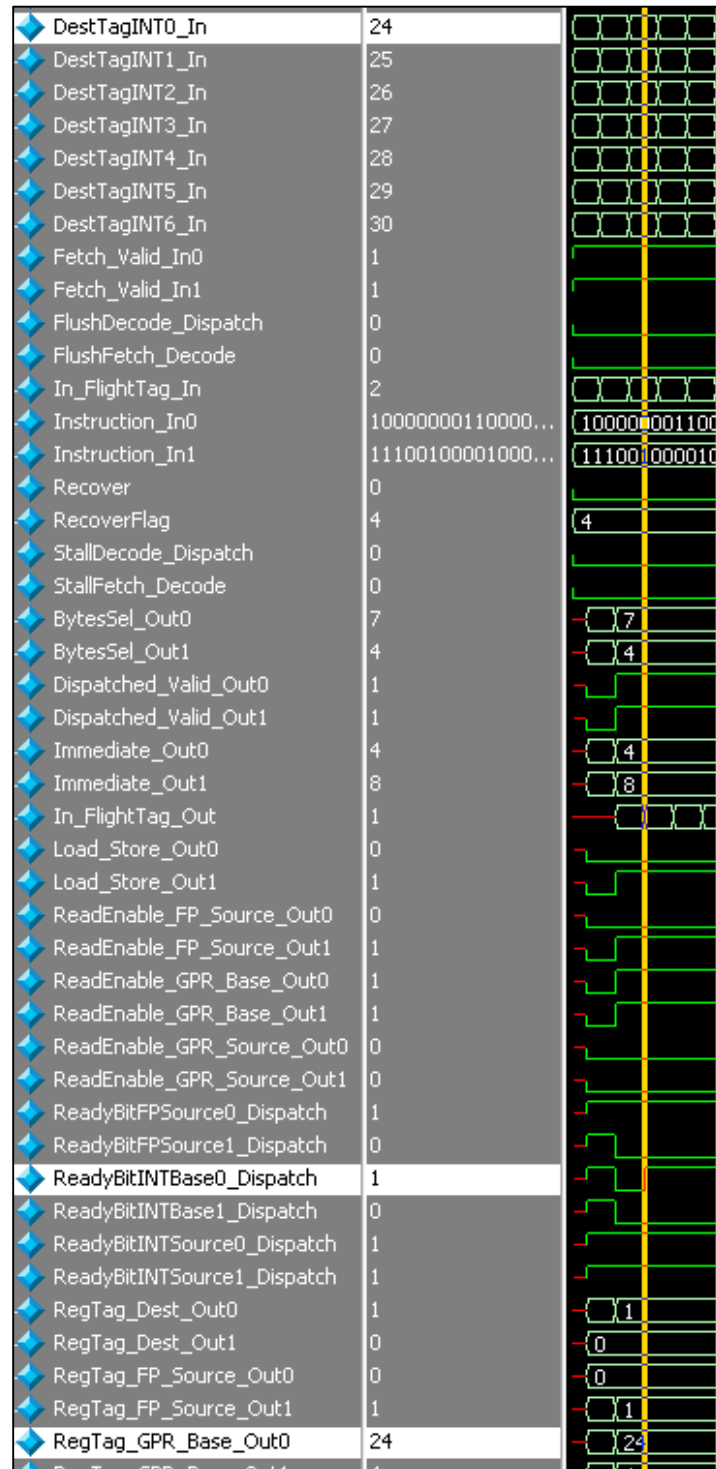


Figure 5-3. Integer Ready Bit Register - Test 1

An implementation and simulation of the Ready Bit Registers (INT & FP) is done in order to check at dispatch time if a source physical register is ready to be read from the Register Files or to be bypassed from the Bypass network, in the test 1 (*Figure 5-3*) a physical register

24 is checked and it is found in the Destination Bus, so a ready bit set to “1” is sent to the Address Queue. In the test 2 (*Figure 5-4*), a 4 is required and it is also found in the Destination Bus.

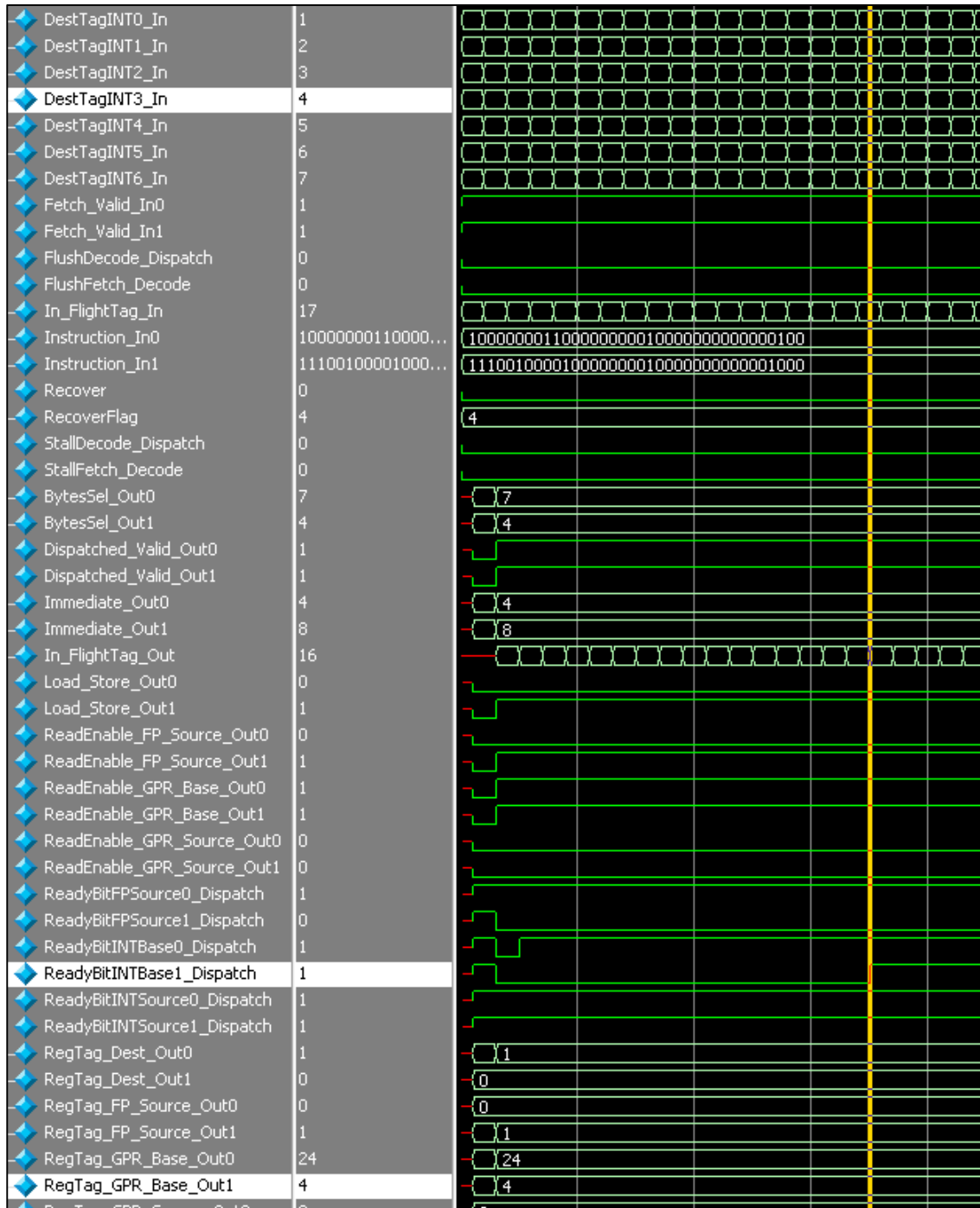


Figure 5-4. Integer Ready Bit Register - Test 2

Some simulations are made for the FP Ready Bit Register (*Figure 5-5*).

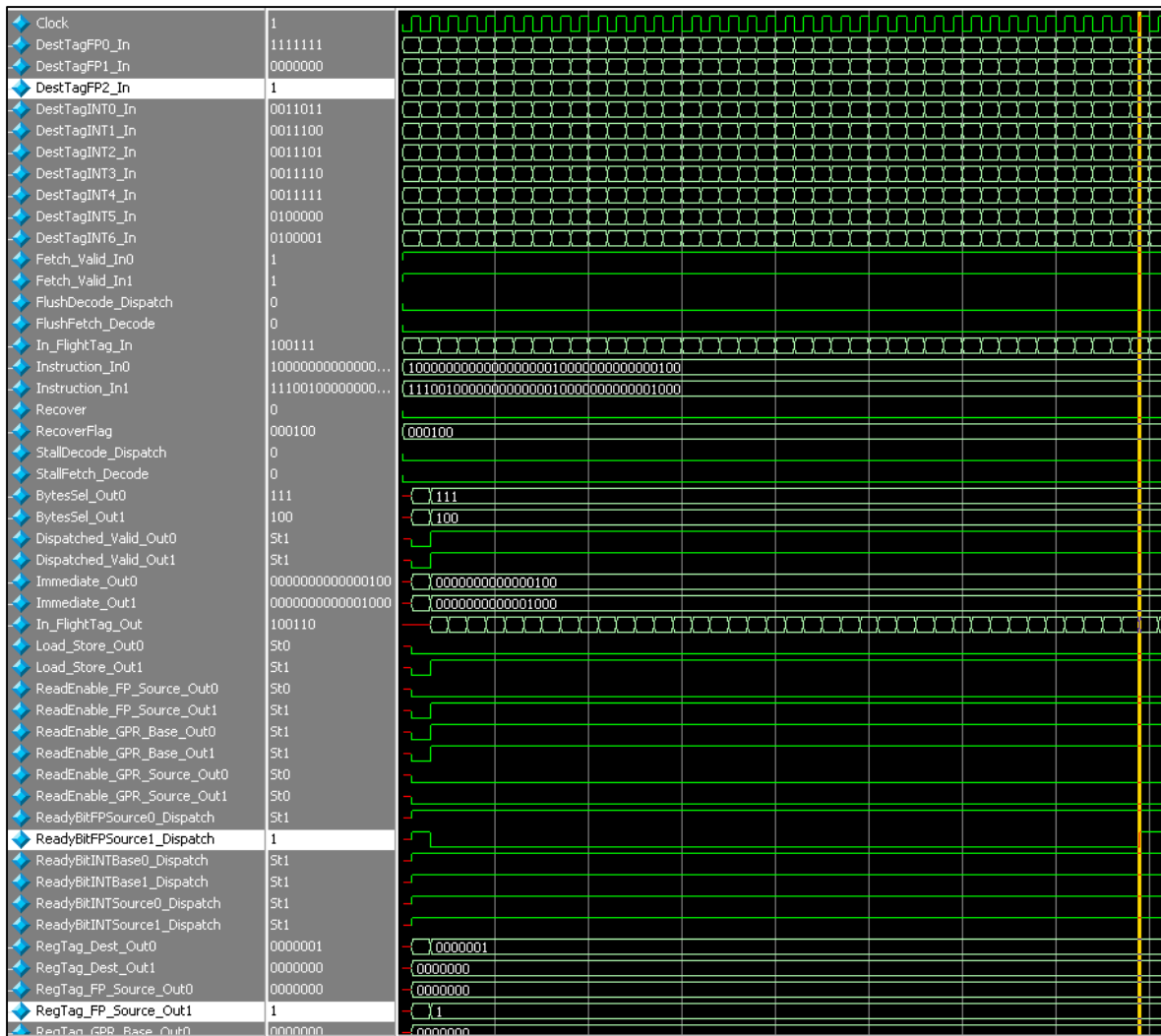


Figure 5-5. Floating Point Ready Bit Register - Test

The control logic of the Head Pointer and Tail Pointer is implemented and simulated, in test 1 (*Figure 5-6*), it was expected to get a “Queue Full” signal in order to stall the pipeline and in the context recovery mechanism test (*Figure 5-7*), the Recovered Tail Pointer is being taken from a Recovery Memory and it updates the actual Tail Pointer after the recovery mechanism.











the entry 2 of the Store Queue (the value is not generated yet, thus, the entry of the dependent store is written in the waiting table), at the same time this store has a memory-dependency with the entry 0 of the Load Queue but the memory-dependency for this entry found in the store-to-load data forwarding belonging to an associative access from a previous issued load (memory-dependent with the entry 4 of the Store Queue) has the most updated value for that memory-dependency, thus, the memory-dependency with the entry 2 is ignored and the store-to-load data forwarding with the entry 4 is executed (forwarding the data “654”). The Store Queue’s entry 2 is sent through the bus in the following clock cycles letting the Load Queue’s entry 5 to forward this data (“102”). The loads (entries 5 and 0) are sent to execution in order to ensure that their data values have been forwarded (in “Data\_Sent”).

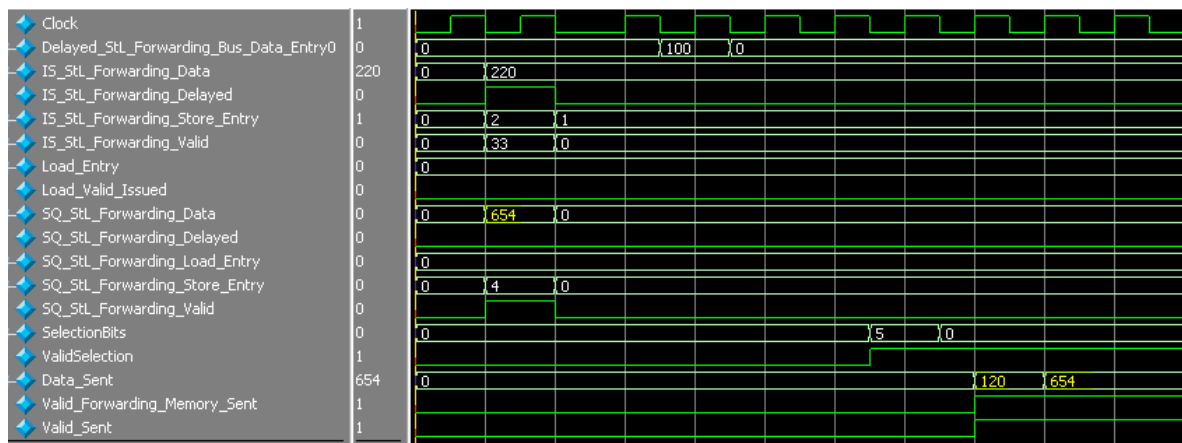


Figure 5-17. Store-to-Load Data Forwarding - Test

---

# Chapter 6

---

## CONCLUSIONS AND FUTURE WORK

---

---

### 6.1 CONCLUSIONS

The main contribution of the project thesis, which is an Out-of-Order Load/Store Queue, is successfully achieved supporting up to three accesses to the Data cache (one load and two stores), and it is shown that the proposed design of the Non/Possible Dependencies Vectors along with the bloom filters and hash table, resulted in an efficient way to reduce a big amount of unnecessary comparisons done by the Memory Disambiguation Mechanism and the Store-to-Load Data Forwarding Logic, reducing a lot of dynamic power consumption in their operation. The Load Queue is supporting up to four dispatched Load/Store instructions per cycle, but it is permitting only one instruction to be issued per cycle due to the complexity of the memory disambiguation process.

---

## 6.2 FUTURE WORK

For future work, it is planned to include a memory dependence predictor, this will increase the performance because each predicted-free dependent load won't wait to be solved (no possible stores left to be issued), thus, some load entries can be sent to execution (accessing the data cache or sending the forwarded data to the destination bus) in advance. Also, a replacement of the source data CAM is planned to be replaced for a RAM, this will decrease a little the performance of the design because the source data memory won't be updated whole every cycle, thus, two stores could get its source data every cycle, the advantage of this replacement is the reduction of the implementation because FPGA's logic elements will be replaced for memory bits. Another future work is to include miss-status handle registers in order to support in-flight data cache misses. After this work done, for future implementations, it will be added the support for multiprocessor technology, and more power saving techniques will be included to the design.

---

# Appendix-A

---

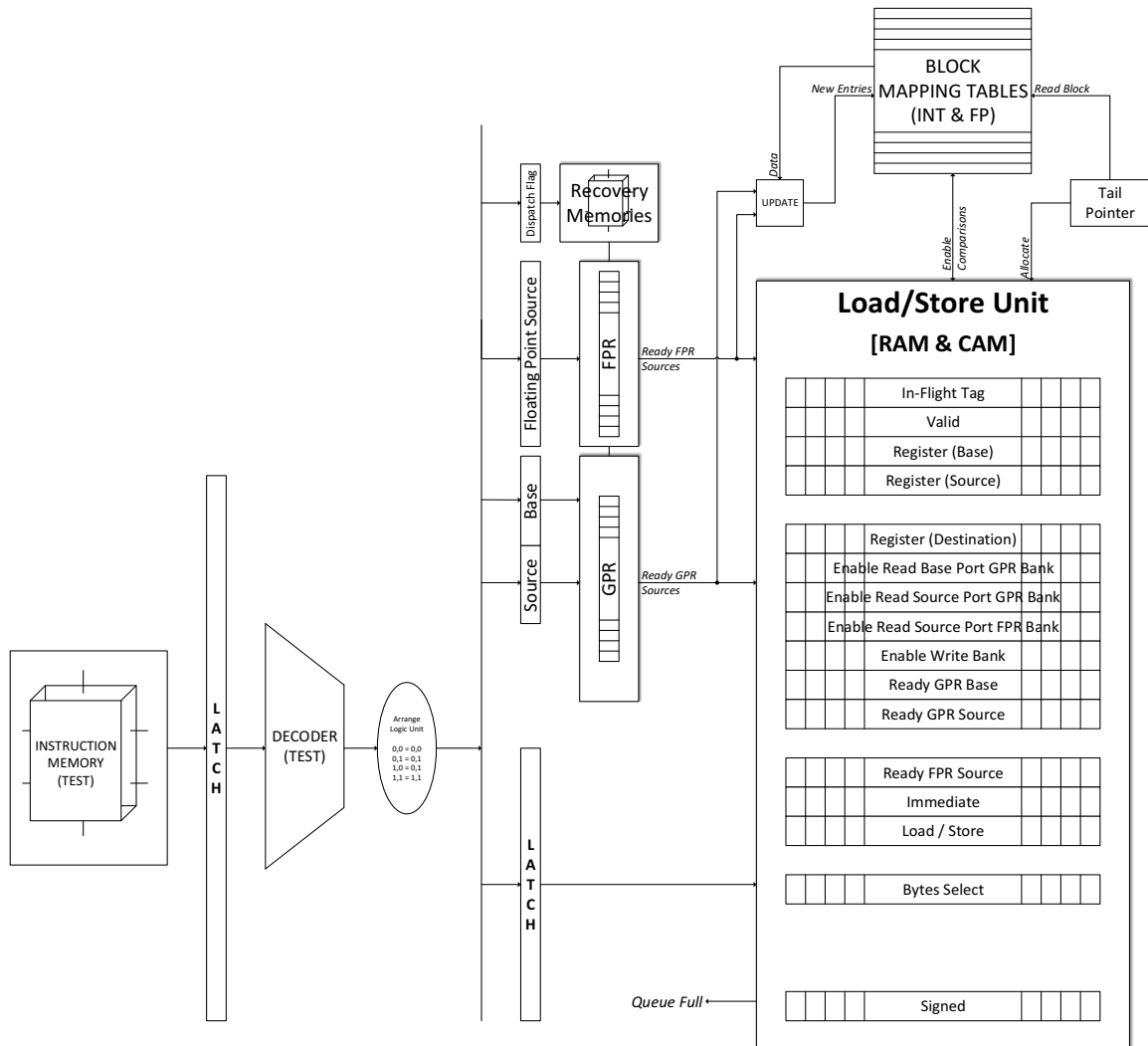


Figure 7-1. Front-End

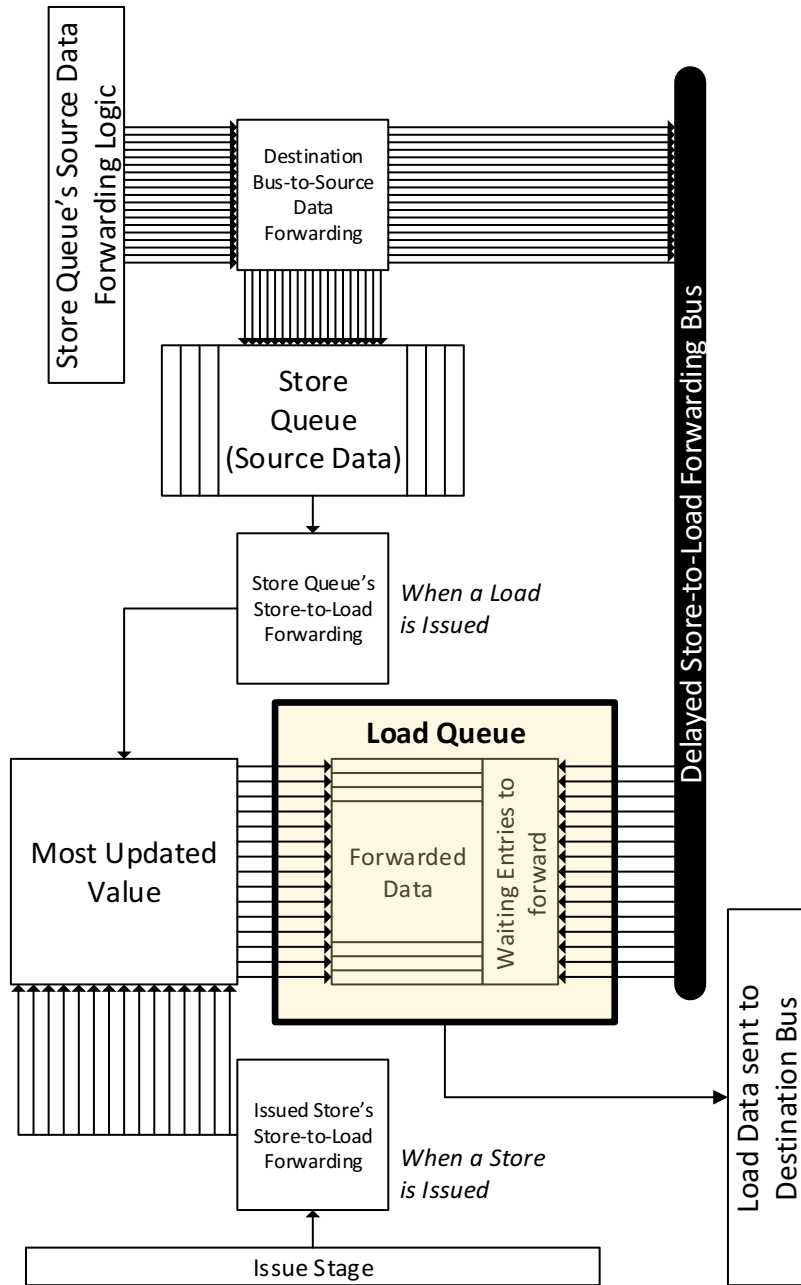


Figure 7-2. Store-to-Load Forwarding Logic including the Delayed Source Data



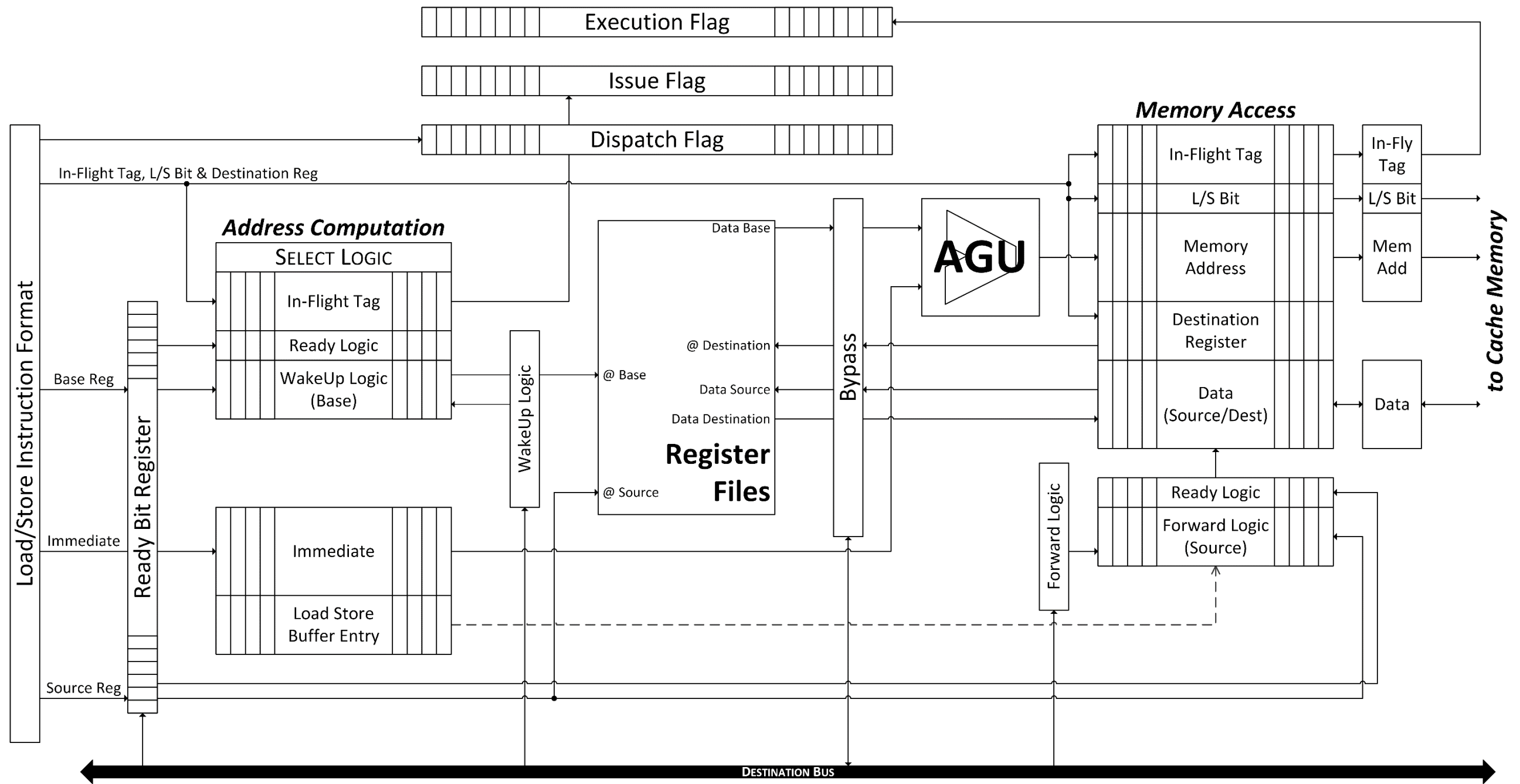


Figure 7-3. Load/Store Queue with In-Order Execution

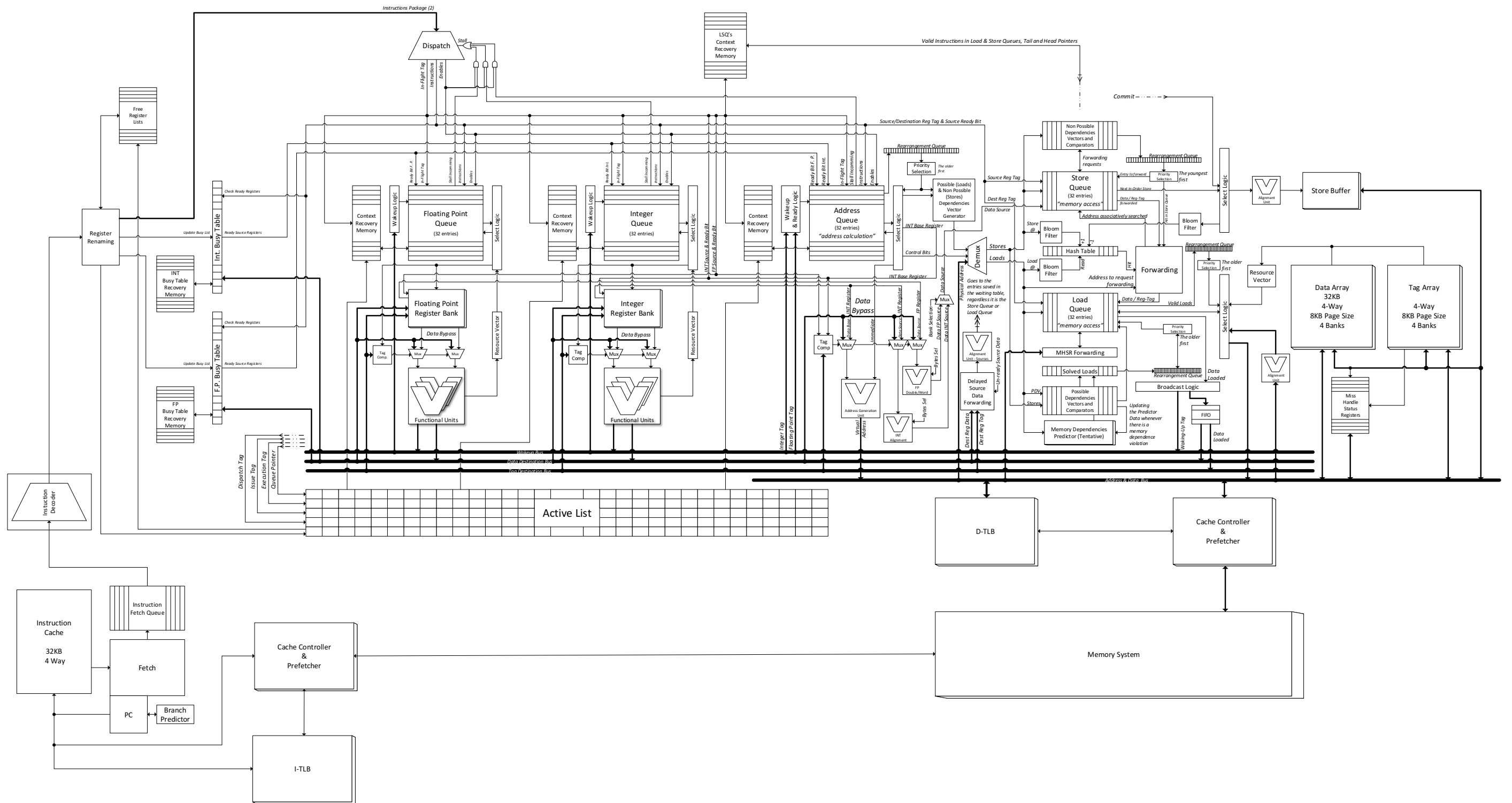


Figure 7-4. Incorporation of the proposed Load/Store Unit to the processor

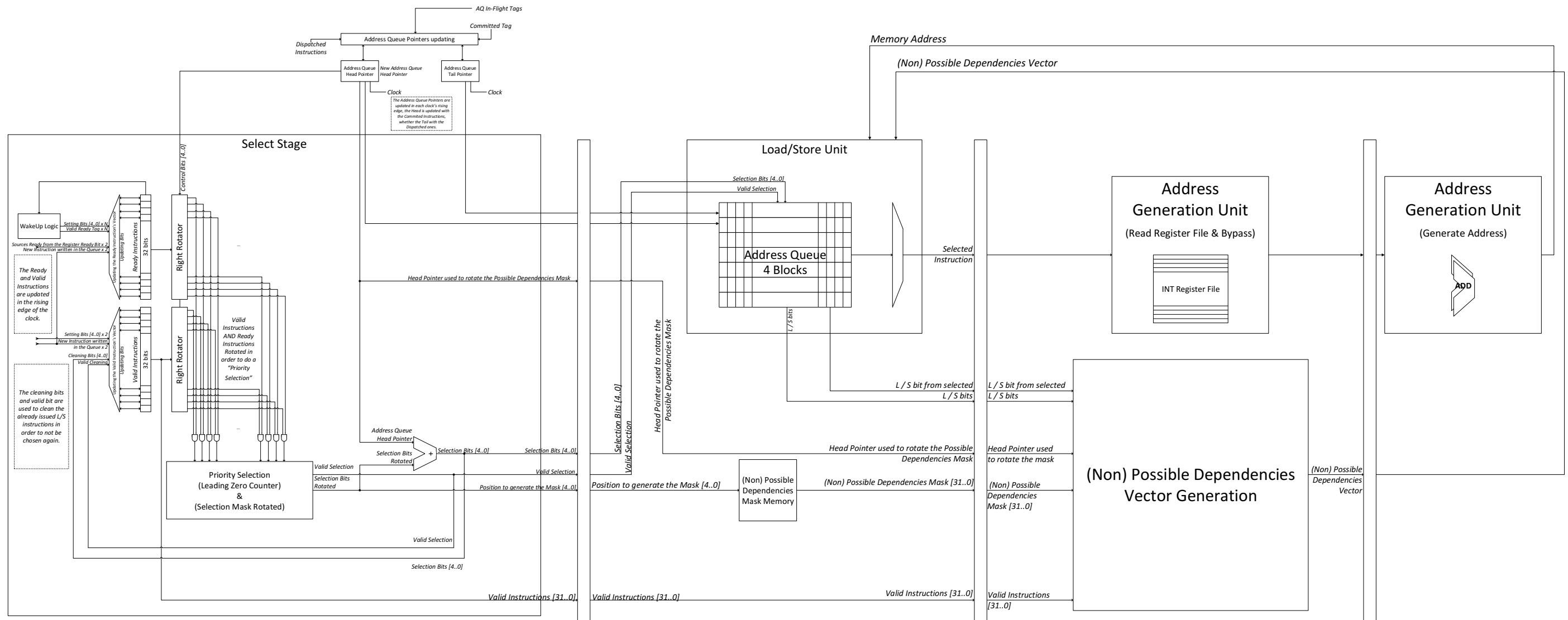


Figure 7-5. Select Logic, Address Generation & PDVs pipeline

---

# Appendix-B

---



State 1	Selected	Valid Instructions	New Head Pointer	(Non) Possible Dependencies Vector	Read / Write EBF	Hit in EBF	Forwarded	Dependence Detection	Load Queue Entries	Store Queue Entries	Solved Loads	State 2	Selected	Valid Instructions	New Head Pointer
*	o	0		000					0	0x00	0	*	o	0	
1	x	0	0x02	000	w	o	o	o	1	o	1	*	o	0	0x03
0	o	1		*				o	2	o	2	1	x	0	
0	o	1	0x01	010	r	o	o	o	1	0x01	1	1	x	0	0x03
1	x	0						o	2	o	2	*	o	0	
1	o	1	0x02	*	w	o	o	o	1	o	1	*	o	0	0x03
1	o	1						o	2	o	2	1	x	0	
1	x	0	0x02	000	r	x	o	o	0	0x00	0	*	o	0	0x03
*	o	0		001				o	1	o	1	*	o	0	
0	o	1		*				o	2	o	2	1	x	0	
0	o	1	0x00	000	r	x	x	o	0	o	0	1	x	0	0x03
*	o	0		001				o	1	o	1	*	o	0	
1	x	0		*				o	2	o	2	*	o	0	
1	x	0	0x02	000	r	x	o	o	0	0x00	0	*	o	0	0x03
*	o	0		001				o	1	o	1	*	o	0	
1	o	1		*				o	2	o	2	1	x	0	
1	x	0	0x01	000	r	o	o	o	0	0x00	0	*	o	0	0x03
0	o	1		*				o	1	0x01	1	1	x	0	0x03
*	o	0		010				o	2	o	2	*	o	0	
0	o	1	0x00	*	w	o	o	o	0	o	0	1	x	0	0x03
1	x	0		001				o	1	0x01	1	*	o	0	
*	o	0		010				o	2	o	2	*	o	0	
1	x	0	0x01	000	r	o	o	o	0	0x00	0	*	o	0	0x03
1	o	1		*				o	1	0x01	1	1	x	0	0x03
*	o	0		010				o	2	o	2	*	o	0	
1	o	0	0x02	000	w	o	o	o	0	0x00	0	*	o	0	0x03
1	x	0		000				o	1	o	1	*	o	0	
0	o	1		*				o	2	o	2	1	x	0	
1	x	0	0x02	000	w	o	o	o	0	0x00	0	*	o	0	0x03
0	o	1		*				o	1	o	1	*	o	0	
0	o	1	0x01	000	r	x	x	o	0	o	0	1	x	0	0x03
*	o	0		001				o	1	o	1	*	o	0	
1	x	0		000				o	2	o	2	*	o	0	
1	o	1	0x02	001	r	x	o	o	0	0x00	0	*	o	0	0x03
1	o	1		*				o	1	o	1	*	o	0	
1	o	1		*				o	2	o	2	1	x	0	
0	o	1	0x00	*	r	x	x	o	0	o	0	1	x	0	0x03
*	o	0		001				o	1	o	1	*	o	0	
1	x	0		000				o	2	o	2	*	o	0	
1	x	0	0x02	000	r	x	o	o	0	0x00	0	*	o	0	0x03
*	o	0		001				o	1	o	1	*	o	0	
1	o	1		*				o	2	o	2	1	x	0	
0	o	1	0x00	*	r	x	x	o	0	o	0	1	x	0	0x03
*	o	0		001				o	1	o	1	*	o	0	
1	x	0		000				o	2	o	2	*	o	0	
1	x	0	0x02	000	w	o	o	o	0	0x00	0	*	o	0	0x03
1	o	1		*				o	1	o	1	*	o	0	
1	o	1		*				o	2	o	2	1	x	0	



(Non) Possible Dependencies Vector	Read / Write EBF	Hit in EBF	Forwarded	Dependence Detection	Load Queue Entries	Store Queue Entries	Solved Loads	
000	r	x	x	o	0	0x00	0	
000					1	0x01	1	o
000					2	o	2	o
000	w	o	o	o	0	0x00	0	
000					1	0x01	1	o
010					2	o	2	o
000	r	x	x	o	0	0x00	0	
000					1	0x01	1	o
000					2	o	2	o
000	r	x	o	o	0	0x00	0	
001					1	0x01	1	o
000					2	o	2	o
000	r	x	o	o	0	0x00	0	
001					1	0x01	1	o
000					2	o	2	o
000	r	x	x	o	0	0x00	0	
001					1	0x01	1	o
000					2	o	2	o
000	w	o	o	o	0	0x00	0	
000					1	0x01	1	o
010					2	o	2	o
000	r	x	o	o	0	0x00	0	
001					1	0x01	1	o
010					2	o	2	o
000	w	o	o	o	0	0x00	0	
000					1	0x01	1	o
010					2	o	2	o
000	r	x	o	o	0	0x00	0	
000					1	0x01	1	o
000					2	o	2	o
000	r	x	x	o	0	0x00	0	
000					1	0x01	1	o
000					2	o	2	o
000	r	x	o	o	0	0x00	0	
001					1	0x01	1	o
000					2	o	2	o
000	w	o	o	x	0	0x00	0	
000					1	0x01	1	o
010					2	o	2	o
000	r	x	o	o	0	0x00	0	
000					1	0x01	1	o
000					2	o	2	o
000	r	x	x	o	0	0x00	0	
001					1	0x01	1	o
000					2	o	2	o
000	r	x	o	o	0	0x00	0	
000					1	0x01	1	o
000					2	o	2	o





Solved Loads	State 1	Selected	Valid Instructions	New Head Pointer	(Non) Possible Dependencies Vector	Read / Write EBF	Hit in EBF	Forwarded	Dependence Detection	Load Queue Entries	Store Queue Entries	Solved Loads	State 2	Selected	Valid Instructions
1 0 0	* 1 0	o x o	0 0 1	0x01	000 000 *	r	o	o	o	0 1 2	0x00 0x01 o	0 1 2	o	o	0 0 0
1 0 0	* 0 1	o o x	0 1 0	0x01	000 * 000	r	o	o	o	0 1 2	0x00 o 0x02	0 1 2	o	o	0 0 0
1 0 0	* 1 1	o x o	0 0 1	0x02	000 000 *	r	o	o	o	0 1 2	0x00 0x01 o	0 1 2	o	o	0 0 0
0 1 0	1 * 0	x o o	0 0 1	0x02	000 000 *	r	o	o	o	0 1 2	0x00 0x01 o	0 1 2	o	o	0 0 0
0 1 0	0 * 1	o o x	1 0 0	0x00	* 000 000	r	o	o	o	0 1 2	o 0x01 0x02	0 1 2	o	o	0 0 0
0 1 0	1 * 1	x o o	0 0 1	0x02	000 000 *	r	o	o	o	0 1 2	0x00 0x01 o	0 1 2	o	o	0 0 0
0 0 1	1 0 *	x o o	0 1 0	0x01	000 * 000	r	o	o	o	0 1 2	0x00 o 0x02	0 1 2	o	o	0 0 0
0 0 1	0 1 *	o x o	1 0 0	0x00	* 000 000	r	o	o	o	0 1 2	o 0x01 0x02	0 1 2	o	o	0 0 0
0 1 0	1 * 1	x o o	0 0 1	0x02	000 000 *	r	o	o	o	0 1 2	0x00 0x01 o	0 1 2	o	o	0 0 0
0 0 1	1 0 *	x o o	0 1 0	0x01	000 * 000	r	o	o	o	0 1 2	0x00 o 0x02	0 1 2	o	o	0 0 0
0 0 1	0 1 *	o x o	1 0 0	0x00	* 000 000	r	o	o	o	0 1 2	o 0x01 0x02	0 1 2	o	o	0 0 0
0 0 1	1 * 1	x o o	0 0 1	0x01	000 * 000	r	o	o	o	0 1 2	0x00 o 0x02	0 1 2	o	o	0 0 0
1 0 0	* 1 0	o x o	0 0 1	0x02	000 000 *	r	o	o	o	0 1 2	0x00 0x01 o	0 1 2	o	o	0 0 0
1 0 0	* 1 1	o x o	0 0 1	0x02	000 000 *	r	o	o	o	0 1 2	0x00 o o	0 1 2	o	o	0 0 0
0 1 0	0 * 1	o o x	1 0 0	0x00	* 000 000	r	o	o	o	0 1 2	o 0x01 0x02	0 1 2	o	o	0 0 0
0 1 1	1 * 0	x o o	0 0 1	0x01	000 * 000	r	o	o	o	0 1 2	0x00 o 0x02	0 1 2	o	o	0 0 0
1 0 0	* 1 1	o x o	0 0 1	0x02	000 000 *	r	o	o	o	0 1 2	0x00 0x01 o	0 1 2	o	o	0 0 0
0 1 0	0 * 1	o o x	1 0 0	0x00	* 000 000	r	o	o	o	0 1 2	o 0x01 0x02	0 1 2	o	o	0 0 0
0 1 1	1 * 0	x o o	0 0 1	0x01	000 * 000	r	o	o	o	0 1 2	0x00 o 0x02	0 1 2	o	o	0 0 0
1 0 0	* 1 1	o x o	0 0 1	0x02	000 000 *	r	o	o	o	0 1 2	0x00 0x01 o	0 1 2	o	o	0 0 0
1 0 0	* 1 1	o x o	0 0 1	0x02	000 000 *	r	o	o	o	0 1 2	0x00 o o	0 1 2	o	o	0 0 0
0 1 0	0 * 1	o o x	1 0 0	0x00	* 000 000	r	o	o	o	0 1 2	o 0x01 0x02	0 1 2	o	o	0 0 0
0 1 0	1 * 1	x o o	0 0 1	0x01	000 * 000	r	o	o	o	0 1 2	0x00 o o	0 1 2	o	o	0 0 0
1 0 0	* 1 1	o x o	0 0 1	0x02	000 000 *	r	o	o	o	0 1 2	0x00 0x01 o	0 1 2	o	o	0 0 0
1 0 0	* 1 1	o x o	0 0 1	0x02	000 000 *	r	o	o	o	0 1 2	0x00 o o	0 1 2	o	o	0 0 0

New Head Pointer	(Non) Possible Dependencies Vector	Read / Write EBF	Hit in EBF	Forwarded	Dependence Detection	Load Queue Entries	Store Queue Entries	Solved Loads	
0x03	000	r	o	o	o	0	0	1	
	000					0x00	0	o	1
	000					0x01	1	o	1
						0x02	2	o	1
0x03	000	r	o	o	o	0	0	1	
	000					0x00	0	o	1
	000					0x01	1	o	1
						0x02	2	o	1
0x03	000	r	o	o	o	0	0	1	
	000					0x00	0	o	1
	000					0x01	1	o	1
						0x02	2	o	1
0x03	000	r	o	o	o	0	0	1	
	000					0x00	0	o	1
	000					0x01	1	o	1
						0x02	2	o	1
0x03	000	r	o	o	o	0	0	1	
	000					0x00	0	o	1
	000					0x01	1	o	1
						0x02	2	o	1
0x03	000	r	o	o	o	0	0	1	
	000					0x00	0	o	1
	000					0x01	1	o	1
						0x02	2	o	1
0x03	000	r	o	o	o	0	0	1	
	000					0x00	0	o	1
	000					0x01	1	o	1
						0x02	2	o	1
0x03	000	r	o	o	o	0	0	1	
	000					0x00	0	o	1
	000					0x01	1	o	1
						0x02	2	o	1
0x03	000	r	o	o	o	0	0	1	
	000					0x00	0	o	1
	000					0x01	1	o	1
						0x02	2	o	1
0x03	000	r	o	o	o	0	0	1	
	000					0x00	0	o	1
	000					0x01	1	o	1
						0x02	2	o	1
0x03	000	r	o	o	o	0	0	1	
	000					0x00	0	o	1
	000					0x01	1	o	1
						0x02	2	o	1
0x03	000	r	o	o	o	0	0	1	
	000					0x00	0	o	1
	000					0x01	1	o	1
						0x02	2	o	1
0x03	000	r	o	o	o	0	0	1	
	000					0x00	0	o	1
	000					0x01	1	o	1
						0x02	2	o	1



Solved Loads	State 1	Selected	Valid Instructions	New Head Pointer	(Non) Possible Dependencies Vector	Read / Write EBF	Hit in EBF	Forwarded	Dependence Detection	Load Queue Entries	Store Queue Entries	Solved Loads	State 2	Selected	Valid Instructions		
0 0 0	* 1 0	o x o	0 0 1	0x02	000 000 *	r	o	o	o	0 1 2	0x00 0x01 o	0 1 2	o o o	1 1 0	* * 1	o o x	0 0 0
0 0 0	* 0 1	o o x	0 1 0	0x01	000 * 010	w	o	o	o	0 1 2	0x00 o o	0 1 2	0x00 o o	1 0 0	* 1 *	o x o	0 0 0
0 0 0	* 1 1	o x o	0 0 1	0x02	000 000 *	r	o	o	o	0 1 2	0x00 0x01 o	0 1 2	o o o	1 0 0	* * 1	o o x	0 0 0
0 1 0	1 * 0	x o o	0 0 1	0x02	000 000 *	r	o	o	o	0 1 2	0x00 0x01 o	0 1 2	o o o	1 1 0	* * 1	o o x	0 0 0
0 1 0	0 * 1	o o x	1 0 0	0x00	* 000 001	w	o	o	o	0 1 2	o 0x01 o	0 1 2	0x00 o o	0 1 0	1 * *	x o o	0 0 0
0 1 0	1 * 1	x o o	0 0 1	0x02	000 000 *	r	o	o	o	0 1 2	0x00 0x01 o	0 1 2	o o o	1 1 0	* * 1	o o x	0 0 0
0 0 0	1 0 *	x o o	0 1 0	0x01	000 * 011	r	x	o	o	0 1 2	0x00 o o	0 1 2	0x00 o o	1 0 0	* 1 *	o x o	0 0 0
0 0 0	0 1 *	o x o	1 0 0	0x00	* 000 011	r	x	o	o	0 1 2	o 0x01 o	0 1 2	0x00 o o	0 1 0	1 * *	x o o	0 0 0
0 0 0	1 1 *	x o o	0 1 0	0x01	000 * 011	r	x	o	o	0 1 2	0x00 o o	0 1 2	0x00 o o	1 0 0	* 1 *	o x o	0 0 0
1 0 0	* 1 0	o x o	0 0 1	0x02	000 000 *	r	o	o	o	0 1 2	0x00 0x01 o	0 1 2	o o o	1 1 0	* * 1	o o x	0 0 0
1 0 0	* 1 1	o x o	0 0 1	0x02	000 000 *	r	o	o	o	0 1 2	0x00 0x01 o	0 1 2	o o o	1 1 0	* * 1	o o x	0 0 0
0 1 0	0 1 1	o x o	1 0 1	0x00	* 000 001	w	o	o	o	0 1 2	o 0x01 o	0 1 2	0x00 o o	0 1 0	1 * *	x o o	0 0 0
0 0 0	1 1 *	x o o	0 1 0	0x02	000 000 *	r	o	o	o	0 1 2	0x00 0x01 o	0 1 2	o o o	1 1 0	* * 1	o o x	0 0 0
0 1 0	0 1 *	o x o	1 0 0	0x01	000 * 011	r	x	o	o	0 1 2	0x00 o o	0 1 2	0x00 o o	1 0 0	* 1 *	o x o	0 0 0
1 0 0	* 1 0	o x o	0 0 1	0x02	000 000 *	r	o	o	o	0 1 2	0x00 0x01 o	0 1 2	o o o	1 1 0	* * 1	o o x	0 0 0
1 0 0	* 1 1	o x o	0 0 1	0x02	000 000 *	r	o	o	o	0 1 2	0x00 0x01 o	0 1 2	o o o	1 1 0	* * 1	o o x	0 0 0
0 1 0	0 1 1	o x o	1 0 1	0x00	* 000 001	w	o	o	o	0 1 2	o 0x01 o	0 1 2	0x00 o o	0 1 0	1 * *	x o o	0 0 0
0 1 0	1 * 1	x o o	0 0 1	0x02	000 000 *	r	o	o	o	0 1 2	0x00 0x01 o	0 1 2	o o o	1 1 0	* * 1	o o x	0 0 0
1 0 0	* 1 1	o x o	0 0 1	0x02	000 000 *	r	o	o	o	0 1 2	0x00 0x01 o	0 1 2	o o o	1 1 0	* * 1	o o x	0 0 0

New Head Pointer	(Non) Possible Dependencies Vector	Read / Write EBF	Hit in EBF	Forwarded	Dependence Detection	Load Queue Entries	Store Queue Entries	Solved Loads		
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	000					1	0x01	1	o	1
	000					2	o	2	o	0
0x03	000	r	x	o	o	0	0x00	0	0x00	1
	000					1	0x01	1	o	1
	000					2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	000					1	0x01	1	o	1
	000					2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	000					1	0x01	1	o	1
	000					2	o	2	o	0
0x03	000	r	x	o	o	0	0x00	0	0x00	1
	001					1	0x01	1	o	1
	000					2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	000					1	0x01	1	o	1
	000					2	o	2	o	0
0x03	000	r	x	o	o	0	0x00	0	0x00	1
	000					1	0x01	1	o	1
	011					2	o	2	o	0
0x03	000	r	x	o	o	0	0x00	0	0x00	1
	000					1	0x01	1	o	1
	011					2	o	2	o	0
0x03	000	r	x	o	o	0	0x00	0	0x00	1
	000					1	0x01	1	o	1
	011					2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	000					1	0x01	1	o	1
	000					2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	000					1	0x01	1	o	1
	000					2	o	2	o	0
0x03	000	r	x	o	o	0	0x00	0	0x00	1
	001					1	0x01	1	o	1
	001					2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	000					1	0x01	1	o	1
	000					2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	000					1	0x01	1	o	1
	000					2	o	2	o	0



Solved Loads	State 1	Selected	Valid Instructions	New Head Pointer	(Non) Possible Dependencies Vector	Read / Write EBF	Hit in EBF	Forwarded	Dependence Detection	Load Queue Entries	Store Queue Entries	Solved Loads	State 2	Selected	Valid Instructions
0	*	o	0		000			o	o	0	0	0	*	o	0
0	1	x	0	0x01	000	w	o	o	o	1	1	0	*	o	0
0	0	o	1		*			o	o	2	2	0	1	x	0
0	*	o	0		000			o	o	0	0	0	*	o	0
0	0	o	1	0x01	*	w	o	o	o	1	1	0	1	x	0
0	1	x	0		000			o	o	2	2	0	*	o	0
0	*	o	0		000			o	o	0	0	0	*	o	0
0	1	x	0	0x02	000	w	o	o	o	1	1	0	*	o	0
0	1	o	1		*			o	o	2	2	0	1	x	0
0	1	x	0		000			o	o	0	0	0	*	o	0
0	*	o	0	0x02	000	w	o	o	o	1	1	0	*	o	0
0	0	o	1		*			o	o	2	2	0	1	x	0
0	0	o	1		*			o	o	0	0	0	1	x	0
0	*	o	0		000			o	o	1	1	0	*	o	0
0	1	x	0	0x00	000	w	o	o	o	2	2	0	*	o	0
0	1	x	0		000			o	o	0	0	0	*	o	0
0	*	o	0	0x02	000	w	o	o	o	1	1	0	*	o	0
0	1	o	1		*			o	o	2	2	0	1	x	0
0	1	x	0		000			o	o	0	0	0	*	o	0
0	0	o	1	0x01	*	w	o	o	o	1	1	0	1	x	0
0	*	o	0		000			o	o	2	2	0	*	o	0
0	0	o	1		*			o	o	0	0	0	1	x	0
0	1	x	0	0x00	000	w	o	o	o	1	1	0	*	o	0
0	*	o	0		000			o	o	2	2	0	*	o	0
0	1	x	0		000			o	o	0	0	0	*	o	0
0	1	x	0	0x02	000	w	o	o	o	1	1	0	*	o	0
0	1	o	1		*			o	o	2	2	0	1	x	0
0	1	x	0		000			o	o	0	0	0	*	o	0
0	0	o	1	0x01	*	w	o	o	o	1	1	0	1	x	0
0	*	o	0		000			o	o	2	2	0	*	o	0
0	0	o	1		*			o	o	0	0	0	1	x	0
0	1	x	0	0x00	000	w	o	o	o	1	1	0	*	o	0
0	*	o	0		000			o	o	2	2	0	*	o	0
0	1	x	0		000			o	o	0	0	0	*	o	0
0	1	x	0	0x02	000	w	o	o	o	1	1	0	*	o	0
0	1	o	1		*			o	o	2	2	0	1	x	0
0	1	x	0		000			o	o	0	0	0	*	o	0
0	0	o	1	0x01	*	w	o	o	o	1	1	0	1	x	0
0	*	o	0		000			o	o	2	2	0	*	o	0
0	0	o	1		*			o	o	0	0	0	1	x	0
0	1	x	0	0x00	000	w	o	o	o	1	1	0	*	o	0
0	*	o	0		000			o	o	2	2	0	*	o	0
0	1	x	0		000			o	o	0	0	0	*	o	0
0	1	x	0	0x02	000	w	o	o	o	1	1	0	*	o	0
0	1	o	1		*			o	o	2	2	0	1	x	0
0	1	x	0		000			o	o	0	0	0	*	o	0
0	0	o	1	0x01	*	w	o	o	o	1	1	0	1	x	0
0	*	o	0		000			o	o	2	2	0	*	o	0
0	0	o	1		*			o	o	0	0	0	1	x	0
0	1	x	0	0x00	000	w	o	o	o	1	1	0	*	o	0
0	*	o	0		000			o	o	2	2	0	*	o	0
0	1	x	0		000			o	o	0	0	0	*	o	0
0	1	x	0	0x02	000	w	o	o	o	1	1	0	*	o	0
0	1	o	1		*			o	o	2	2	0	1	x	0
0	1	x	0		000			o	o	0	0	0	*	o	0
0	0	o	1	0x01	*	w	o	o	o	1	1	0	1	x	0
0	*	o	0		000			o	o	2	2	0	*	o	0
0	0	o	1		*			o	o	0	0	0	1	x	0
0	1	x	0	0x00	000	w	o	o	o	1	1	0	*	o	0
0	*	o	0		000			o	o	2	2	0	*	o	0
0	1	x	0		000			o	o	0	0	0	*	o	0
0	1	x	0	0x02	000	w	o	o	o	1	1	0	*	o	0
0	1	o	1		*			o	o	2	2	0	1	x	0



New Head Pointer	(Non) Possible Dependencies Vector	Read / Write EBF	Hit in EBF	Forwarded	Dependence Detection	Load Queue Entries	Store Queue Entries	Solved Loads
0x03	000	w	o	o	o	0	0	0
	000					1	1	0
	000					2	2	0
0x03	000	w	o	o	o	0	0	0
	000					1	1	0
	000					2	2	0
0x03	000	w	o	o	o	0	0	0
	000					1	1	0
	000					2	2	0
0x03	000	w	o	o	o	0	0	0
	000					1	1	0
	000					2	2	0
0x03	000	w	o	o	o	0	0	0
	000					1	1	0
	000					2	2	0
0x03	000	w	o	o	o	0	0	0
	000					1	1	0
	000					2	2	0
0x03	000	w	o	o	o	0	0	0
	000					1	1	0
	000					2	2	0
0x03	000	w	o	o	o	0	0	0
	000					1	1	0
	000					2	2	0
0x03	000	w	o	o	o	0	0	0
	000					1	1	0
	000					2	2	0
0x03	000	w	o	o	o	0	0	0
	000					1	1	0
	000					2	2	0
0x03	000	w	o	o	o	0	0	0
	000					1	1	0
	000					2	2	0
0x03	000	w	o	o	o	0	0	0
	000					1	1	0
	000					2	2	0
0x03	000	w	o	o	o	0	0	0
	000					1	1	0
	000					2	2	0
0x03	000	w	o	o	o	0	0	0
	000					1	1	0
	000					2	2	0
0x03	000	w	o	o	o	0	0	0
	000					1	1	0
	000					2	2	0
0x03	000	w	o	o	o	0	0	0
	000					1	1	0
	000					2	2	0
0x03	000	w	o	o	o	0	0	0
	000					1	1	0
	000					2	2	0





Solved Loads	State 1	Selected	Valid Instructions	New Head Pointer	(Non) Possible Dependencies Vector	Read / Write EBF	Hit in EBF	Forwarded	Dependence Detection	Load Queue Entries	Store Queue Entries	Solved Loads	State 2	Selected	Valid Instructions		
1 0 0	* 1 0	o x o	0 0 1	0x02	000 000 *	w	o	o o o	o o o	0 1 2	0x00 o o	0 1 2	0x00 o o	1 0 0	* * 1	o o x	0 0 0
1 0 0	* 0 1	o o x	0 1 0	0x01	000 000 000	w	o	o o o	o o o	0 1 2	0x00 o o	0 1 2	o 0x01 o	1 0 0	* 1 *	o x o	0 0 0
1 0 0	* 1 1	o x o	0 0 1	0x02	000 000 *	w	o	o o o	o o o	0 1 2	0x00 o o	0 1 2	0x00 o o	1 0 0	* * 1	o o x	0 0 0
0 0 0	1 * 0	x o o	0 0 1	0x02	000 001 *	r	x	o o o	o o o	0 1 2	0x00 o o	0 1 2	0x00 o o	1 0 0	* * 1	o o x	0 0 0
0 0 0	0 * 1	o o x	1 0 0	0x00	* 001 001	w	o	o o o	o o o	0 1 2	o o o	0 1 2	0x00 0x01 o	0 0 0	1 * *	x o o	0 0 0
0 0 0	1 * 0	x o o	0 0 1	0x02	000 001 *	r	x	o o o	o o o	0 1 2	0x00 o o	0 1 2	0x00 o o	1 0 0	* * 1	o o x	0 0 0
0 0 0	1 0 *	x o o	0 1 0	0x01	000 * 001	r	x	o o o	o o o	0 1 2	0x00 o o	0 1 2	o 0x01 o	1 0 0	* 1 *	o x o	0 0 0
0 0 0	0 1 *	o x o	1 0 0	0x00	* 001 001	w	o	o o o	o o o	0 1 2	o o o	0 1 2	0x00 0x01 o	0 0 0	1 * *	x o o	0 0 0
0 0 0	1 * 0	x o o	0 0 1	0x02	000 001 *	r	x	o o o	o o o	0 1 2	0x00 o o	0 1 2	0x00 o o	1 0 0	* * 1	o o x	0 0 0
0 0 0	1 0 *	x o o	0 1 0	0x01	000 * 001	r	x	o o o	o o o	0 1 2	0x00 o o	0 1 2	o 0x01 o	1 0 0	* 1 *	o x o	0 0 0
0 0 0	0 1 *	o x o	1 0 0	0x00	* 001 001	w	o	o o o	o o o	0 1 2	o o o	0 1 2	0x00 0x01 o	0 0 0	1 * *	x o o	0 0 0
0 0 0	1 1 *	x o o	0 1 0	0x01	000 * 001	r	x	o o o	o o o	0 1 2	0x00 o o	0 1 2	o 0x01 o	1 0 0	* 1 *	o x o	0 0 0
1 0 0	* 1 0	o x o	0 0 1	0x02	000 000 *	w	o	o o o	o o o	0 1 2	0x00 o o	0 1 2	0x00 o o	1 0 0	* * 1	o o x	0 0 0
1 0 0	* 1 1	o x o	0 0 1	0x02	000 000 *	w	o	o o o	o o o	0 1 2	0x00 o o	0 1 2	0x00 o o	1 0 0	* * 1	o o x	0 0 0
o o o	0 * 1	o o x	1 0 0	0x00	* 001 001	w	o	o o o	o o o	0 1 2	o o o	0 1 2	0x00 0x01 o	0 0 0	1 * *	x o o	0 0 0
o o o	1 * 1	x o o	0 0 1	0x02	000 001 *	r	x	o o o	o o o	0 1 2	0x00 o o	0 1 2	0x00 o o	1 0 0	* * 1	o o x	0 0 0
1 0 0	* 1 1	o x o	0 0 1	0x02	000 000 *	w	o	o o o	o o o	0 1 2	0x00 o o	0 1 2	0x00 o o	1 0 0	* * 1	o o x	0 0 0

New Head Pointer	(Non) Possible Dependencies Vector	Read / Write EBF	Hit in EBF	Forwarded	Dependence Detection	Load Queue Entries	Store Queue Entries	Solved Loads		
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	000					1	o	1	0x01	0
	000					2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	000					1	o	1	0x01	0
	000					2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	000					1	o	1	0x01	0
	000					2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	001					1	o	1	0x01	0
	000					2	o	2	o	0
0x03	000	r	x	o	o	0	0x00	0	0x00	1
	001					1	o	1	0x01	0
	001					2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	001					1	o	1	0x01	0
	000					2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	000					1	o	1	0x01	0
	001					2	o	2	o	0
0x03	000	r	x	o	o	0	0x00	0	0x00	1
	001					1	o	1	0x01	0
	001					2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	000					1	o	1	0x01	0
	001					2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	000					1	o	1	0x01	0
	000					2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	000					1	o	1	0x01	0
	000					2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	000					1	o	1	0x01	0
	000					2	o	2	o	0
0x03	000	r	x	o	o	0	0x00	0	0x00	1
	001					1	o	1	0x01	0
	001					2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	000					1	o	1	0x01	0
	000					2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	000					1	o	1	0x01	0
	000					2	o	2	o	0
0x03	000	r	x	o	o	0	0x00	0	0x00	1
	001					1	o	1	0x01	0
	001					2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	001					1	o	1	0x01	0
	000					2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	000					1	o	1	0x01	0
	000					2	o	2	o	0



Solved Loads	State 1	Selected	Valid Instructions	New Head Pointer	(Non) Possible Dependencies Vector	Read / Write EBF	Hit in EBF	Forwarded	Dependence Detection	Load Queue Entries	Store Queue Entries	Solved Loads	State 2	Selected	Valid Instructions
0	*	o	0		000			x	o	0	0x00	0	0x00	1	* o 0
0	1	x	0	0x02	000	r	x	o	o	1	o	1	o	0	* o 0
0	0	o	1		*			o	o	2	o	2	o	0	1 x 0
0	*	o	0		000			x	o	0	o	0	0x00	0	* o 0
0	0	o	1	0x01	*	r	x	o	o	1	0x01	1	o	1	1 x 0
0	1	x	0		000			o	o	2	o	2	o	0	* o 0
0	*	o	0		000			x	o	0	0x00	0	0x00	1	* o 0
0	1	x	0	0x02	000	r	x	o	o	1	o	1	o	0	* o 0
0	1	o	1		*			o	o	2	o	2	o	0	1 x 0
0	1	x	0		000			o	o	0	0x00	0	0x00	1	* o 0
0	*	o	0	0x02	001	w	o	o	x	1	o	1	o	0	* o 0
0	0	o	1		*			o	o	2	o	2	o	0	1 x 0
0	0	o	1		*			o	o	0	0x00	0	o	0	1 x 0
0	*	o	0	0x00	001	r	o	o	o	1	0x01	1	o	0	* o 0
0	1	x	0		001			o	o	2	o	2	o	0	* o 0
0	1	x	0		000			o	o	0	0x00	0	0x00	1	* o 0
0	*	o	0	0x02	001	w	o	o	x	1	o	1	o	0	* o 0
0	1	o	1		*			o	o	2	o	2	o	0	1 x 0
0	1	x	0		000			o	o	0	o	0	0x00	0	* o 0
0	0	o	1	0x01	*	w	o	o	o	1	0x01	1	o	1	1 x 0
0	*	o	0		001			o	x	2	o	2	o	0	* o 0
0	0	o	1		*			o	o	0	0x00	0	o	0	1 x 0
0	1	x	0	0x00	001	r	o	o	o	1	0x01	1	o	0	* o 0
0	*	o	0		001			o	o	2	o	2	o	0	* o 0
0	1	x	0		000			o	o	0	o	0	0x00	0	* o 0
0	1	o	1	0x01	*	w	o	o	o	1	0x01	1	o	1	1 x 0
0	*	o	0		001			o	x	2	o	2	o	0	* o 0
0	*	o	0		000			x	o	0	0x00	0	0x00	1	* o 0
0	1	x	0	0x02	000	r	x	o	o	1	o	1	o	0	* o 0
0	1	o	1		*			o	o	2	o	2	o	0	1 x 0
0	0	o	1		*			o	o	0	0x00	0	o	0	1 x 0
0	1	x	0	0x02	000	r	x	o	o	1	o	1	o	0	* o 0
0	0	o	1		*			o	o	2	o	2	o	0	1 x 0
0	0	o	1		*			o	o	0	0x00	0	o	0	1 x 0
0	*	o	0	0x00	001	r	o	o	o	1	0x01	1	o	0	* o 0
0	1	x	0		001			o	o	2	o	2	o	0	* o 0
0	1	x	0		000			o	o	0	0x00	0	0x00	1	* o 0
0	*	o	0	0x02	001	w	o	o	x	1	o	1	o	0	* o 0
0	1	o	1		*			o	o	2	o	2	o	0	1 x 0
0	*	o	0		000			x	o	0	0x00	0	0x00	1	* o 0
0	1	x	0	0x02	000	r	x	o	o	1	o	1	o	0	* o 0
0	1	o	1		*			o	o	2	o	2	o	0	1 x 0

New Head Pointer	(Non) Possible Dependencies Vector	Read / Write EBF	Hit in EBF	Forwarded	Dependence Detection	Load Queue Entries	Store Queue Entries	Solved Loads		
0x03	000	r	x	x	o	0	0x00	0	0x00	1
	000			o	o	1	0x01	1	o	1
	000			o	o	2	o	2	o	0
0x03	000	r	x	x	o	0	0x00	0	0x00	1
	000			o	o	1	0x01	1	o	1
	000			o	o	2	o	2	o	0
0x03	000	r	x	x	o	0	0x00	0	0x00	1
	000			o	o	1	0x01	1	o	1
	000			o	o	2	o	2	o	0
0x03	000	r	x	x	o	0	0x00	0	0x00	1
	000			o	o	1	0x01	1	o	1
	000			o	o	2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	001			x	1	0x01	1	o	1	
	001			x	2	o	2	o	0	
0x03	000	r	x	x	o	0	0x00	0	0x00	1
	000			o	o	1	0x01	1	o	1
	000			o	o	2	o	2	o	0
0x03	000	r	x	x	o	0	0x00	0	0x00	1
	000			o	o	1	0x01	1	o	1
	000			o	o	2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	001			x	1	0x01	1	o	1	
	001			x	2	o	2	o	0	
0x03	000	r	x	x	o	0	0x00	0	0x00	1
	000			o	o	1	0x01	1	o	1
	000			o	o	2	o	2	o	0
0x03	000	r	x	x	o	0	0x00	0	0x00	1
	000			o	o	1	0x01	1	o	1
	000			o	o	2	o	2	o	0
0x03	000	w	o	o	o	0	0x00	0	0x00	1
	001			x	1	0x01	1	o	1	
	001			x	2	o	2	o	0	
0x03	000	r	x	x	o	0	0x00	0	0x00	1
	000			o	o	1	0x01	1	o	1
	000			o	o	2	o	2	o	0
0x03	000	r	x	x	o	0	0x00	0	0x00	1
	000			o	o	1	0x01	1	o	1
	000			o	o	2	o	2	o	0
0x03	000	r	x	x	o	0	0x00	0	0x00	1
	000			o	o	1	0x01	1	o	1
	000			o	o	2	o	2	o	0
0x03	000	r	x	x	o	0	0x00	0	0x00	1
	000			o	o	1	0x01	1	o	1
	000			o	o	2	o	2	o	0





Solved Loads	State 1	Selected	Valid Instructions	New Head Pointer	(Non) Possible Dependencies Vector	Read / Write EBF	Hit in EBF	Forwarded	Dependence Detection	Load Queue Entries	Store Queue Entries	Solved Loads	State 2	Selected	Valid Instructions
0	*	o	0		000			x	o	0	0x00	0	0x00	1	* o 0
0	1	x	0	0x02	000	r	x	o	o	1	o	1	o	0	* o 0
0	0	o	1		*			o	o	2	o	2	o	0	1 x 0
0	*	o	0		000			o	o	0	o	0	0x00	0	* o 0
0	0	o	1	0x01	*	w	o	o	o	1	o	1	0x01	0	1 x 0
0	1	x	0		010			o	o	2	o	2	o	0	* o 0
0	*	o	0		000			x	o	0	0x00	0	0x00	1	* o 0
0	1	x	0	0x02	000	r	x	o	o	1	o	1	o	0	* o 0
0	1	o	1		*			o	o	2	o	2	o	0	1 x 0
0	1	x	0		000			o	o	0	0x00	0	0x00	1	* o 0
0	*	o	0	0x02	001	w	o	o	x	1	o	1	o	0	* o 0
0	0	o	1		*			o	o	2	o	2	o	0	1 x 0
0	0	o	1		*			o	o	0	0x00	0	o	0	1 x 0
0	*	o	0	0x00	001	w	o	o	o	1	o	1	0x01	0	* o 0
0	1	x	0		000			o	o	2	o	2	o	0	* o 0
0	1	x	0		000			o	o	0	0x00	0	0x00	1	* o 0
0	*	o	0	0x02	001	w	o	o	x	1	o	1	o	0	* o 0
0	1	o	1		*			o	o	2	o	2	o	0	1 x 0
0	1	x	0		000			o	o	0	o	0	0x00	0	* o 0
0	0	o	1	0x01	*	w	o	o	o	1	o	1	0x01	0	1 x 0
0	*	o	0		010			o	o	2	o	2	o	0	* o 0
0	0	o	1		*			o	o	0	0x00	0	o	0	1 x 0
0	1	x	0	0x00	001	r	x	o	o	1	o	1	0x01	0	* o 0
0	*	o	0		010			o	o	2	o	2	o	0	* o 0
0	1	x	0		000			o	o	0	o	0	0x00	0	* o 0
0	1	o	1	0x01	*	w	o	o	o	1	o	1	0x01	0	1 x 0
0	*	o	0		010			o	o	2	o	2	o	0	* o 0
0	*	o	0		000			x	o	0	0x00	0	0x00	1	* o 0
0	1	x	0	0x02	000	r	x	o	o	1	o	1	o	0	* o 0
0	0	o	1		*			o	o	2	o	2	o	0	1 x 0
0	1	x	0		000			o	o	0	0x00	0	0x00	1	* o 0
0	*	o	0	0x02	010	w	o	o	x	1	o	1	o	0	* o 0
0	1	o	1		*			o	o	2	o	2	o	0	1 x 0
0	0	o	1		*			o	o	0	0x00	0	o	0	1 x 0
0	*	o	0	0x00	001	w	o	o	o	1	o	1	0x01	0	* o 0
0	1	x	0		000			o	o	2	o	2	o	0	* o 0
0	1	x	0		000			x	o	0	0x00	0	0x00	1	* o 0
0	1	o	1		*			o	o	1	o	1	o	0	* o 0
0	0	o	1		000			o	o	2	o	2	o	0	1 x 0
0	0	o	1		*			o	o	0	0x00	0	o	0	1 x 0
0	*	o	0	0x00	001	w	o	o	o	1	o	1	0x01	0	* o 0
0	1	x	0		000			o	o	2	o	2	o	0	* o 0
0	1	x	0		000			x	o	0	0x00	0	0x00	1	* o 0
0	1	o	1		*			o	o	1	o	1	o	0	* o 0
0	1	o	1		000			o	o	2	o	2	o	0	1 x 0
0	0	o	1		*			o	o	0	0x00	0	o	0	1 x 0
0	*	o	0	0x00	001	w	o	o	o	1	o	1	0x01	0	* o 0
0	1	x	0		000			o	o	2	o	2	o	0	* o 0
0	*	o	0		000			x	o	0	0x00	0	0x00	1	* o 0
0	1	x	0	0x02	000	r	x	o	o	1	o	1	o	0	* o 0
0	1	o	1		*			o	o	2	o	2	o	0	1 x 0



New Head Pointer	(Non) Possible Dependencies Vector	Read / Write EBF	Hit in EBF	Forwarded	Dependence Detection	Load Queue Entries	Store Queue Entries	Solved Loads
0x03	000	w	o	o	o	0	0	1
	000					0x00	0x00	0
	000					o	0x01	1
0x03	000	r	x	x	o	0	0	1
	000					0x00	0x00	0
	010					o	0x01	1
0x03	000	w	o	o	o	0	0	1
	000					0x00	0x00	0
	000					o	0x01	1
0x03	000	w	o	o	o	0	0	1
	000					0x00	0x00	0
	000					o	0x01	1
0x03	000	w	o	o	o	0	0	1
	000					0x00	0x00	0
	000					o	0x01	1
0x03	000	w	o	o	x	0	0	1
	001					0x00	0x00	0
	000					o	0x01	1
0x03	000	w	o	o	o	0	0	1
	000					0x00	0x00	0
	000					o	0x01	1
0x03	000	r	x	x	o	0	0	1
	000					0x00	0x00	0
	010					o	0x01	1
0x03	000	w	o	o	o	0	0	1
	001					0x00	0x00	0
	010					o	0x01	1
0x03	000	r	x	o	o	0	0	1
	000					0x00	0x00	0
	010					o	0x01	1
0x03	000	w	o	o	o	0	0	1
	000					0x00	0x00	0
	000					o	0x01	1
0x03	000	w	o	o	o	0	0	1
	000					0x00	0x00	0
	000					o	0x01	1
0x03	000	w	o	o	o	0	0	1
	000					0x00	0x00	0
	000					o	0x01	1
0x03	000	w	o	o	x	0	0	1
	001					0x00	0x00	0
	000					o	0x01	1
0x03	000	w	o	o	o	0	0	1
	000					0x00	0x00	0
	000					o	0x01	1
0x03	000	w	o	o	o	0	0	1
	000					0x00	0x00	0
	000					o	0x01	1
0x03	000	w	o	o	o	0	0	1
	000					0x00	0x00	0
	000					o	0x01	1
0x03	000	w	o	o	x	0	0	1
	001					0x00	0x00	0
	000					o	0x01	1
0x03	000	w	o	o	o	0	0	1
	000					0x00	0x00	0
	000					o	0x01	1



Solved Loads	State 1	Selected	Valid Instructions	New Head Pointer	(Non) Possible Dependencies Vector	Read / Write EBF	Hit in EBF	Forwarded	Dependence Detection	Load Queue Entries	Store Queue Entries	Solved Loads	State 2	Selected	Valid Instructions
0	*	o	0		000			o	o	0	0x00	0	*	o	0
0	1	x	0	0x02	000	w	o	o	o	1	0x01	0	*	o	0
0	0	o	1		*			o	o	2	o	0	1	x	0
0	*	o	0		000			x	o	0	0x00	0	*	o	0
0	0	o	1	0x01	*	r	x	o	o	1	o	0	1	x	0
0	1	x	0		010			o	o	2	o	0	*	o	0
0	*	o	0		000			o	o	0	0x00	0	*	o	0
0	1	x	0	0x02	000	w	o	o	o	1	0x01	0	*	o	0
0	1	o	1		*			o	o	2	o	0	1	x	0
0	1	x	0		000			o	o	0	0x00	0	*	o	0
0	*	o	0	0x02	000	w	o	o	o	1	0x01	0	*	o	0
0	0	o	1		*			o	o	2	o	0	1	x	0
0	0	o	1		*			o	o	0	0x00	0	1	x	0
0	*	o	0	0x00	000	r	x	x	o	1	0x01	0	*	o	0
0	1	x	0		001			o	o	2	o	0	*	o	0
0	1	x	0		000			o	o	0	0x00	0	*	o	0
0	*	o	0	0x02	000	w	o	o	o	1	0x01	0	*	o	0
0	1	o	1		*			o	o	2	o	0	1	x	0
0	1	x	0		000			o	o	0	0x00	0	*	o	0
0	0	o	1	0x01	*	w	o	o	o	1	o	0	1	x	0
0	*	o	0		011			o	x	2	o	0	*	o	0
0	0	o	1		*			o	o	0	0x00	0	1	x	0
0	1	x	0	0x00	000	w	o	o	o	1	0x01	0	*	o	0
0	*	o	0		011			o	x	2	o	0	*	o	0
0	1	x	0		000			o	o	0	0x00	0	*	o	0
0	1	o	1	0x01	*	w	o	o	o	1	o	0	1	x	0
0	*	o	0		011			o	x	2	o	0	*	o	0
0	*	o	0		000			o	o	0	0x00	0	*	o	0
0	1	x	0	0x02	000	w	o	o	o	1	0x01	0	*	o	0
0	0	o	1		*			o	o	2	o	0	1	x	0
0	*	o	0		000			o	o	0	0x00	0	*	o	0
0	1	x	0	0x02	000	w	o	o	o	1	0x01	0	*	o	0
0	1	o	1		*			o	o	2	o	0	1	x	0
0	0	o	1		*			o	o	0	0x00	0	1	x	0
0	*	o	0	0x00	000	r	x	x	o	1	0x01	0	*	o	0
0	1	x	0		001			o	o	2	o	0	*	o	0
0	1	x	0		000			o	o	0	0x00	0	*	o	0
0	*	o	0	0x02	000	w	o	o	o	1	0x01	0	*	o	0
0	1	o	1		*			o	o	2	o	0	1	x	0
0	0	o	1		*			o	o	0	0x00	0	1	x	0
0	*	o	0	0x00	000	r	x	x	o	1	0x01	0	*	o	0
0	1	x	0		001			o	o	2	o	0	*	o	0
0	1	x	0		000			o	o	0	0x00	0	*	o	0
0	*	o	0	0x02	000	w	o	o	o	1	0x01	0	*	o	0
0	1	o	1		*			o	o	2	o	0	1	x	0
0	*	o	0		000			o	o	0	0x00	0	*	o	0
0	1	x	0	0x02	000	w	o	o	o	1	0x01	0	*	o	0
0	1	o	1		*			o	o	2	o	0	1	x	0



# References

- [1] M. A. Ramírez, A. Cristal Kestelman, A. V. Veidenbaum, L. Villa and M. Valero, "A Simple Low-Energy Instruction Wakeup Mechanism," in *Proceedings of the 5th International Symposium, ISHPC*, Japan, 2003.
- [2] A. S. Tanenbaum, *Structured Computer Organization*, 5th ed., Pearson Prentice Hall, 2006.
- [3] R. L. Britton, *MIPS Assembly Language Programming*, Pearson Prentice Hall, 2004.
- [4] T. Shanley and D. Anderson, *ISA System Architecture*, MindShare, Inc., 1995.
- [5] I. Corporation, "Intel®64 and IA-32 Architectures Software Developer's Manual," 2011.
- [6] I. Corporation, "Intel®64 and IA-32 Architectures Software Developer's Manual," 2011.
- [7] W. M. Johnson, "Super-Scalar Processor Design," 1989.
- [8] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed., USA: Morgan Kauffmann, 2007.
- [9] A. González, F. Latorre and G. Magklis, *Processor Microarchitecture: An Implementation Perspective*, Synthesis Lectures on Computer Architecture, 2010.
- [10] J. R. G. Ordaz, *Diseño de un ROB-Distribuido para Procesadores Superescalares*, Mexico, 2011.
- [11] J. E. Smith and G. S. Sohi, "The Microarchitecture of Superscalar Processors," in *Proceedings of the IEEE*, 1995.
- [12] A. J. Smith, "Cache Memories," *ACM Computing Surveys*, vol. 14, no. 3, pp. 473-530, September 1982.
- [13] F. Castro, D. Chaver, L. Piñuel, M. Prieto and F. Tirado, "Memory Disambiguation Hardware: a Review," *Journal of Computer Science & Technology*, vol. 8, no. 3, 2008.
- [14] H. W. Cain and M. H. Lipasti, "Memory Ordering: A Value Based Definition," in *Proceedings of the 31st International Symposium on Computer Architecture*, 2004.

- [15] L. Sethumadhavan, B. E. and M. S., *Scalable Hardware Memory Disambiguation*, Texas, Austin, 2007.
- [16] T. Sha, M. M. K. Martin and A. Roth, "Scalable Store-Load Forwarding via Store Queue Index Prediction (MICRO 38)," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, Barcelona, 2005.
- [17] S. S. Stone, K. M. Woley and M. I. Frank, "Address-Indexed Memory Disambiguation and Store-to-Load Forwarding," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, Barcelona, 2005.
- [18] S. Subramaniam and G. H. Loh, "Store vectors for scalable memory dependence prediction and scheduling," in *The Twelfth International Symposium on High-Performance Computer Architecture, 2006*, Austin, 2006.
- [19] A. M. D. Inc., "Software Optimization Guide for AMD Family 15h Processors," 2014.
- [20] S. Bieschewski, *Design of a Distributed Memory Unit for Clustered Microarchitectures*, Spain, 2013.
- [21] J. A. Fisher, "Very Long Instruction Word Architectures and the ELI-512," in *Proceedings of the 10th Annual International Symposium on Computer Architecture, ISCA*, New York, USA, 1983.
- [22] L. J. Boland, G. D. Granito, A. U. Marcotte, B. U. Messina and J. W. Smith, "The IBM System/360 Model 91: Storage System," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 54-68, January 1967.
- [23] Y. N. Patt, S. W. Melvin, W. M. Hwu and M. C. Shebanow, "Critical issues regarding HPS, a high performance microarchitecture," in *MICRO 18 Proceedings of the 18th annual workshop on Microprogramming*, New York, 1985.
- [24] M. Franklin and G. S. Sohi, "ARB: A hardware mechanism for dynamic reordering of memory references," *IEEE Transactions on Computers*, vol. 45, no. 5, pp. 552-571, May 1996.
- [25] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler and D. Burger, "Distributed Microarchitectural Protocols in the TRIPS Prototype Processor," in *Proceeding of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Washington, 2006.
- [26] S. Sethumadhavan, R. Desikan, R. McDonald, D. C. Burger and S. W. Keckler, "Design and Implementation of the TRIPS Primary Memory System," in *ICCD 2006. International Conference on Computer Design*, 2006.

- [27] S. Sethumadhavan, F. Roesner, J. S. Emer, D. Burger and S. W. Keckler, "Late-Binding: Enabling Unordered Load-Store Queues," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, San Diego, 2007.
- [28] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore and S. W. Keckler, "Scalable Hardware Memory Disambiguation for High ILP Processors," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Washington, 2003.
- [29] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422-426, July 1970.
- [30] H. Song, S. Dharmapurikar, J. Turner and J. Lockwood, "Fast Hash Table Lookup Using Extended Bloom filter: An Aid to Network Processing," in *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Philadelphia, 2005.
- [31] G. Z. Chrysos and J. S. Emer, "Memory Dependence Prediction using Store Sets," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, Washington, 1998.
- [32] G. Dimitrakopoulos, K. Galanopoulos, C. Mavrokefalidis and D. Nikolos, "Low-Power Leading-Zero Counting and Anticipation Logic for High-Speed Floating Point Units," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 7, pp. 837-850, July 2008.
- [33] P. Mishra, N. Dutt and A. Nicolau, "A Study of Out-of-Order Completion for the MIPS R10K Superscalar Processor," Information and Computer Science, Irvine, California, 2001.
- [34] S. Palacharla, N. P. Jouppi and J. E. Smith, "Complexity-Effective Superscalar Processors," in *Proceedings of the 24th annual International Symposium on Computer Architecture, ISCA '97*, Denver, 1997.