ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Αρχιτεκτονική και ασφάλεια υπολογιστών: Σχεδίαση νέας αρχιτεκτονικής VLIW με κωδικοποίηση των εξαρτήσεων μεταξύ των εντολών της ίδια λέξης εντολής

ΜΥΡΤΖΑΚΗΣ ΚΩΝΣΤΑΝΤΙΝΟΣ ΝΙΚΟΛΑΟΣ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΥΠΕΥΘΥΝΟΣ

Δημητρίου Γεώργιος
Επίκουρος Καθηγητής

Λαμία Παρασκευή 15/03 έτος 2024

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Αρχιτεκτονική και ασφάλεια υπολογιστών: Σχεδίαση νέας αρχιτεκτονικής VLIW με κωδικοποίηση των εξαρτήσεων μεταξύ των εντολών της ίδια λέξης εντολής

ΜΥΡΤΖΑΚΗΣ ΚΩΝΣΤΑΝΤΙΝΟΣ ΝΙΚΟΛΑΟΣ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΥΠΕΥΘΥΝΟΣ

Δημητρίου Γεώργιος
Επίκουρος Καθηγητής

Λαμία Παρασκευή 15/03 έτος 2024

UNIVERSITY OF
THESSALY

SCHOOL OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE & TELECOMMUNICATIONS

Computer architecture and security: The design of a new VLIW architecture that encodes instruction dependencies within the same instruction word

KONSTANTINOS NIKOLAOS MYRTZAKIS

FINAL THESIS

ADVISOR

Georgios Dimitriou
Assistant Professor

Lamia Friday 15/03 year 2024

«Με ατομική μου ευθύνη και γνωρίζοντας τις κυρώσεις [(1)], που προβλέπονται από της διατάξεις της παρ. 6 του άρθρου 22 του Ν. 1599/1986, δηλώνω ότι:

*1. Δεν παραθέτω κομμάτια βιβλίων ή άρθρων ή εργασιών άλλων αυτολεξεί **χωρίς να τα περικλείω σε εισαγωγικά** και χωρίς να αναφέρω το συγγραφέα, τη χρονολογία, τη σελίδα. Η αυτολεξεί παράθεση χωρίς εισαγωγικά χωρίς αναφορά στην πηγή, είναι λογοκλοπή. Πέραν της αυτολεξεί παράθεσης, λογοκλοπή θεωρείται και η παράφραση εδαφίων από έργα άλλων, συμπεριλαμβανομένων και έργων συμφοιτητών μου, καθώς και η παράθεση στοιχείων που άλλοι συνέλεξαν ή επεξεργάσθηκαν, χωρίς αναφορά στην πηγή. Αναφέρω πάντοτε με πληρότητα την πηγή κάτω από τον πίνακα ή σχέδιο, όπως στα παραθέματα.*

*2. Δέχομαι ότι η αυτολεξεί **παράθεση χωρίς εισαγωγικά**, ακόμα κι αν συνοδεύεται από αναφορά στην πηγή σε κάποιο άλλο σημείο του κειμένου ή στο τέλος του, είναι αντιγραφή. Η αναφορά στην πηγή στο τέλος π.χ. μιας παραγράφου ή μιας σελίδας, δεν δικαιολογεί συρραφή εδαφίων έργου άλλου συγγραφέα, έστω και παραφρασμένων, και παρουσίασή τους ως δική μου εργασία.*

3. Δέχομαι ότι υπάρχει επίσης περιορισμός στο μέγεθος και στη συχνότητα των παραθεμάτων που μπορώ να εντάξω στην εργασία μου εντός εισαγωγικών. Κάθε μεγάλο παράθεμα (π.χ. σε πίνακα ή πλαίσιο, κλπ), προϋποθέτει ειδικές ρυθμίσεις, και όταν δημοσιεύεται προϋποθέτει την άδεια του συγγραφέα ή του εκδότη. Το ίδιο και οι πίνακες και τα σχέδια

4. Δέχομαι όλες τις συνέπειες σε περίπτωση λογοκλοπής ή αντιγραφής.

Ημερομηνία: 15/03/2024

Ο - Η Δηλ.

(1) «Όποιος εν γνώσει του δηλώνει ψευδή γεγονότα ή αρνείται ή αποκρύπτει τα αληθινά με έγγραφη υπεύθυνη δήλωση
του άρθρου 8 παρ. 4 Ν. 1599/1986 τιμωρείται με φυλάκιση τουλάχιστον τριών μηνών. Εάν ο υπαίτιος αυτών των πράξεων
σκόπευε να προσπορίσει στον εαυτόν του ή σε άλλον περιουσιακό όφελος βλάπτοντας τρίτον ή σκόπευε να βλάψει άλλον, τιμωρείται με κάθειρξη μέχρι 10 ετών.»

ΠΕΡΙΛΗΨΗ

Το να σχεδιάσει κανείς, υπό συνθήκες ακριβείας, μία αρχιτεκτονική συνόλου εντολών Very Long Instruction Word(VLIW), δεν είναι μικρό ζήτημα. Σε αυτήν εδώ την εργασία, στόχος μας είναι να σχεδιάσουμε μία αρχιτεκτονική VLIW, η εντολή της οποίας να χωρίζεται σε οκτώ συλλαβές. Κάθε μία από τις πρώτες επτά συλλαβές, είναι και μία εντολή εκτέλεσης. Με την τελευταία συλλαβή να αποτελεί ορισμός εξαρτήσεων μεταξύ αυτών των επτά εντολών.

Η ερευνά μας για αυτήν την εργασία, ξεκίνησε με την μελέτη αναγνωρισμένων βιβλίων πάνω στα θέματα της αρχιτεκτονική, οργάνωσης, και σχεδίασης υπολογιστών. Αυτό όμως δεν ήταν αρκετό ώστε να μάθουμε περισσότερα για την ιστορία των επεξεργαστών VLIW. Για να μάθουμε περισσότερα για την προέλευση, καθώς και τους πρωτοπόρους των VLIW επεξεργαστών, καταφύγαμε σε έρευνες που αφορούσαν μεταγλωττιστές, έως και σε εγχειρίδια συγκεκριμένων υλοποιήσεων επεξεργαστών VLIW. Η αρχιτεκτονική του Itanium, είναι μία τέτοια αρχιτεκτονική VLIW. Η αποκαλούμενη σήμερα Explicitly Paraller Instruction Computing(EPIC) προσέγγιση, αναφέρεται σε πολλά επιστημονικά βιβλία του κλάδου, ως ο δρόμος προς την δημιουργία αποτελεσματικών VLIW επεξεργαστών. Έτσι, μελετήσαμε το παράδειγμα του EPIC προσεκτικά. Για να σχεδιάσουμε το δικό μας σύνολο εντολών, πήραμε έμπνευση από υπάρχουσες αρχιτεκτονικές, όπως των: Mips, RISC-V, και του Itanium. Πλέον προετοιμασμένοι για την δημιουργία του δικού μας σχεδίου, ξεκινήσαμε την δημιουργία του, με στόχο επίσης την δημιουργία μίας προσομοίωσης του σχεδίου που να μπορεί να τρέξει ενδεικτικό κώδικά παραγμένου δια χειρός, ώστε να αναδείξουμε τον παραλληλισμό επιπέδου εντολή(ILP) που διακατέχει τους VLIW.

Σε αυτό το σημείο, θεωρήσαμε σωστό, την αναφορά σε διάφορα παραδείγματα VLIW που κυκλοφόρησαν στην αγορά, ώστε να αναδείξουμε την ιστορία τους. Ανά τα χρόνια, επεξεργαστές διάφορων ικανοτήτων δημιουργήθηκαν, βασισμένοι σε διάφορα σχέδια συνόλων εντολών VLIW. Από τους "γίγαντες" της Multiflow, έως και τους ενσωματωμένους επεξεργαστές της οικογένειας ST200, παραγωγής STMicroelectronics και Hewllet-Packard(hp). Παρόλα αυτά, λίγοι από αυτούς βρήκαν επιτυχία.

Τα συμπεράσματά μας, δώθηκαν στο τέλος της εργασίας. Οι επεξεργαστές VLIW είναι πολύ ικανοί πάνω σε επιστημονικές εφαρμογές, και έχουν μικρότερη κατανάλωση ενέργειας από ότι οι υπερβαθμωτοί επεξεργαστές. Παρόλα αυτά, η απαιτητική στην σχεδίαση μεταγλωττιστών φύση τους, τους κάνει δύσκολο να χρησιμοποιηθούν ως επεξεργαστές γενικού σκοπού. Ένα πιο απλό σχέδιο, το οποίο θα ακολουθούσε τις βασικές αρχές των VLIW, και που θα δανείζεται τα λιγότερο περίπλοκα "κομμάτια" των υπερβαθμωτών επεξεργαστών(Όχι πολύ διαφορετικό, από αυτο που κάνει το πρότυπο EPIC), θα αναδείκνυε την φιλοσοφία των VLIW, και θα ήταν περιζήτητο στην αγορά ως επεξεργαστής.

ABSTRACT

To design a Very Long Instruction Word(VLIW), Instruction Set Architecture(ISA), under certain precise parameters is not a small task. In this paper, our goal was to design a VLIW architecture, whose instruction were to be split into eight syllables. Each of the first seven syllables an operation, with the final one functioning as a way to define operation interdependencies between the operations issued by the rest syllables.

We begun out research by studying well credited books on the matter of computer architecture, design and organization. This however wasn't enough to learn about the history of the VLIW philosophy. To know how this philosophy first appeared, and who were it's pioneers we had to read research papers on the subject of compilers, as well as manuals on specific implementations of such VLIW designs. Itanium's, architecture is one such VLIW architecture, which added it's own twist on the VLIW paradigm. The now called Explicitly Parallel Instruction Computing(EPIC) approach, is mentioned in many scientific books as the way to move forward when it comes to implementing effective modern VLIW architectures. As such, we studied the paradigm thoroughly. In order to design our own ISA, we took inspiration from existing architectures including: Mips, Risc-V, as well as Itanium's. Having now prepared for the creation of our own design, we set fourth, aiming to also create a simulation of this design, capable of running examples of "handwritten" code, to showcase the instruction level parallelism(ILP) for which VLIW processors are known.

Reaching this point, we saw fit, a presentation of commercial examples, of such VLIW processing units, allowing us a viewpoint to their history. Processing units of various capabilities were created based on several different VLIW ISAs. From Multiflow's line of "giant" "Trace -/" computers, to todays ST200 family of embedded processing units developed by STMicroelectronics and Hewlett-Packard(hp). However, not many of these processing units found commercial success.

To conclude our paper, we draw our conclusions. VLIW processors, are very capable on scientific applications, and less power-consumption demanding, than their more architecturally complex superscalar relatives. However, their demanding on the compiler design nature, makes them difficult to work with as general purpose computers. A simpler design, following the main principles of the VLIW philosophy, and borrowing the least complex parts of superscalar computer architecture(Not very dissimilar to what the EPIC paradigm does), both exemplifies better the original VLIW philosophy, and makes for a more sought after processing unit, when implemented.

Table of Contents

ΠΕΡΙΛΗΨΗ.....I
ABSTRACT.....III

CHAPTER 1 INTRODUCTION.....1

(Introduction to parallel execution 1.1).....1
(Historic attempts in the field 1.2).....2
(Our interest in the architecture 1.3).....3
(Characteristics of a VLIW architecture 1.4).....4
(EPIC in relation to VLIW 1.5).....6
(The Project's Aim 1.6).....7

CHAPTER 2 LITERATURE REVIEW.....8

(The Resources we used 2.1).....8
(More books and other literature 2.2).....9

CHAPTER 3 EXAMPLES OF COMMERCIAL VLIW PROCESSORS.....10

(Chapter introduction 3.1).....10
(i860 3.2).....11
(FR-V 3.3).....12
(Super Harvard Architecture Single-chip Computer(SHARC) 3.4).....13
(Multiflow's Trace -/ 3.5).....14
(ST240 3.6).....15

CHAPTER 4 THE NEW ARCHITECTURE.....17

(How is it similar to others? 4.1).....17
(How is it different to others? 4.2).....18
(Getting Started 4.3).....19
(Getting things in line 4.4).....20
(Instruction Fetch, the beginning 4.5).....21
(Instruction Decode 4.6).....22
(Execution Units 4.7).....23
(Memory Access 4.8).....25
(WriteBack 4.9).....27
(Distribute 4.10).....28
(Distribute Memory Access Points 4.11).....30
(Other Execution Units 4.12).....31
(Integer Units 4.12.a).....31
(Floating-point Units 4.12.b).....33
(The New Instruction Set 4.13).....36
(The Instruction Format 4.14).....37
(The Immediate and Other Fields 4.15).....40

(Interdependencies in a Syllable 4.16)....41
(Future Expansions and Security 4.17)....42
(Concluding the Chapter 4.18)....43

CHAPTER 5 SIMULATING THE ARCHITECTURE....44

(Chapter introduction 5.1)....44
(The Simulation's Structure of the Files 5.2)....45
(Simulating the Registers and the System's Memory 5.3)....46
(Definition 5.3.a)....46
(Usage 5.3.b)....47
(Operation Latency 5.4)....48
(Instruction Decode and Change of Cycle 5.5)....49
(Instruction Execution 5.6)....50
(What is each C-file for? 5.6.a)....50
(The rest of our C-files, and a flowchart of the code 5.6.b)....51
(Executable's Output 5.7)....53
(Terminal's Output 5.7.a)....53
(System State Output 5.7.b)....54
(Executable Input 5.8)....55
(Instruction Memory 5.8.a)....55
(List of Operation Opcodes 5.8.b)....56
(Simulated Examples of Execution 5.9)....57
(Concluding the Chapter 5.10)....61

CHAPTER 6 CONCLUSIONS....62

BIBLIOGRAPHY....63

Chapter 1 Introduction

(Introduction to parallel execution 1.1)

Since the conceivement of the processor, there have been many attempts to increase it's processing power. Higher clock rates, pipelining, and super scalar architecture, have all been ways in which processing unit designers have tried to achieve this end result. However, in the early 2000's there were already steps in achieving parallel execution within the processing unit, in order for it to meet, the rapidly, growing market demands. Parallel execution of instructions, became the way to move forward.

Many are the techniques that allow the parallel execution of operations(or instructions), in the central processing unit. However, the majority of these techniques result in highly complex central processing unit(CPU) architectures, that come with disadvantages. One technique which aims to move in the exact opposite direction is the Very Long Instruction Word Architecture(VLIW), which aims to simplify the CPU architecture, at the cost of more complex compiler design. In this architectural technique, the heavy lifting is being delivered by the software side of the computer, as opposed to the hardware side, as mentioned earlier. We call this approach ***static*** instruction level parallelism, as opposed to the ***dynamic*** solution of non-VLIW techniques.

The parallelism of this nature, is achieved by having the compiler predetermine which instructions are capable of being executed concurrently, and feeding the architecture with the proper compilation output. This minimizes the need for complex architectures, and instead we get multiple units capable of performing similar simple instructions concurrently, thus achieving the design's intended function, and goal.

(Historic attempts in the field 1.2)

The first commercial VLIW processor was produced by a company named Multiflow. One of the founding members of Multiflow was a computer scientist named Joseph A. Fisher, the person who originally proposed the benefits of a theoretical VLIW processor.

He originally worked on a technique for compilers which was able to take advantage of a processor's Instruction Level Parallelism(ILP), which was why he advocated for the creation of a VLIW processor. His Trace Scheduling compiler algorithm is a worthy subject on it's own. However, it is beyond the scope of our current paper. The theoretical VLIW computer, that he had originally conceived, was never built commercially. However, Multiflow's VLIW computers found relative success in scientific applications.

Perhaps the most notorious attempt of commercializing the VLIW Architecture, was the processor known as Itanium, by Intel.
Intel, in collaboration with HP, moved forward in creating a processing unit based on the then only thought of as a theoretical expansion of what was already known as VLIW, and the term they used to describe their implementation based on this, up until then theoretical approach, came to be known as Explicitly Parallel Instruction Computing(EPIC).

Unfortunately, although an architectural format which makes use of multiple execution units processing information based on a single instruction, similar to VLIW, known as Single Instruction Multiple Data(SIMD), has found use since the need to move on independent Graphical Processing Units(GPU) arose, the technique aimed on CPUs never really took off, and was mostly abandoned. The benefits of the architecture seemed minor, compared to the high complexity of the compilers that had to be developed with it.

(Our interest in the architecture 1.3)

Although not widely used in commercial processing units, computer scientists may find interest in this technique, because we can always learn from the past, and improve for the future.

With this specific example, we take a closer look on the topic of instruction level parallelism, on a computer organization level, and the interdependencies of operations executed in parallel.

(Characteristics of a VLIW architecture 1.4)

A VLIW architecture, is usually comprised of a series of fetch units, which feed an equal numbered series of Execution Units, with parts of an instruction acquired from within the instruction memory. The Instruction Memory is usually located within a cache memory closer to the processing unit than the main memory. During the fetching process, part of the instruction acquired defines the way in which the individual parts of the instruction will be delivered to the execution units, and thus “decoding” is required before issuing these instruction parts, to the execution units.

The Execution Units can be comprised by Arithmetic Logical Units(ALU), Floating Point Arithmetic Units(FPU), both of the above, and, or other similar units.

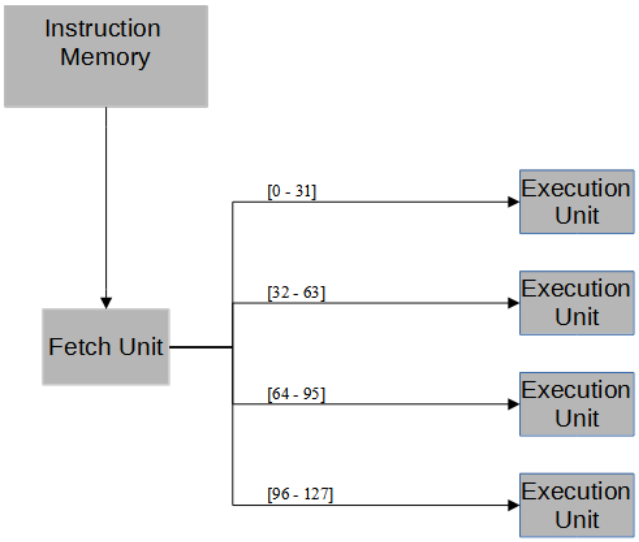

Image 1.1

In the Image 1.1 above, we can see a simplified example, of the design mentioned earlier. In this visual example, we see four individual 32-bit “words” being fetched from the instruction memory, to the execution units, through a mechanism of an unspecified exact nature.

Since the resolution of the order of execution takes place during the compilation process, statically, we have a single **very long instruction word,** from which these individual operations are fetched from, to the total of the observed execution units. In the above visualization, we do not see a mechanism tasked with the decoding being implied, which is though, a subject that will be of interest in our project.

Given limitless resources, certain operations of an uncountable quantity would be capable of being executed in parallel, within a single processing unit cycle. Within the context of the EPIC implementation of a VLIW architecture, we have a limited number of resources, and thus a group of operations capable to be executed in parallel would find itself within two markers which would imply a change of cycle. The designers of Itanium, when creating their work decided that grouping the operations in groups of a specific size, would be of a benefit.

Each operation belonged to a category, and each category of operations could be executed within certain units. A table was then defined, with the possible combinations of execution units in concurrent use by a group of operations. This table included options of markers, that would imply a change of cycle, within or at the end of such a group. Furthermore, one of the groups operations, could be a placeholder operation, that would be there to fill a gap, in case there was no need for all the slots of the group, to be utilized. The table would then be used to statically describe the way in which the operations within a single, but also in relation to the following, instruction were to be executed by the processing unit.

(EPIC in relation to VLIW 1.5)

It is worth noting that the aforementioned Itanium processor, which follows the Explicitly Parallel Instruction Computing(EPIC) approach developed by Intel and HP, differs to the usual theoretical VLIW processor. Within the EPIC paradigm, the VLIW processor that is Itanium, as seen in the appendix H of the 6th edition's Computer Architecture: A Quantitative Approach book adopts many characteristics of the superscalar computers, as presented in the 4th chapter of 4th edition's Computer Organization and Design book, of the same authors.

The difference between the two approaches is based on a basic principle. The "pure" VLIW approach handles dependencies during the compilation process, while the EPIC approach uses specialized hardware mechanisms to handle a number of the same dependencies. Intel, HP, and distinguished academics, believe this second approach has it's merits. However, in this paper, we will not present an architecture which adopts as many similar "superscalar" mechanisms, even though it has inspired, and influenced our Instruction Set Architecture(ISA).

(The Project's Aim 1.6)

Our goal, is to design a new VLIW architecture, which will include, within it's instruction set paradigm, a "sub-word", from now on referred to as a syllable, instructing the computer organization on how to deal with interdependencies that appear within the same instruction, and in relation to the following instruction. To specify the architecture a bit more, we will require it to have an instruction set, within which 256-bit instructions will appear. This will be a total of 8, 32-bit syllables in length, and the 8^{th} syllable will be reserved for the dependency deputy. After the design is complete, we will set up a cycle-level simulation using c, and we will make observations on it.

In their book, Computer Architecture: A Quantitative Approach, John L. Hennessy, and David A. Patterson wrote about what is to be expected of a modern Reduced Instruction Set Computer(RISC). Since our project's aim is to create a RISC, VLIW architecture, it seems to be a sound choice to try, and follow their suggestions. As such, we want our Instruction Set Architecture, to have:

- a variety of options for operations, without unnecessary changes of format between the various available instruction types, improving the performance,
- a wide enough "immediate-field" for our Branch, and Jump formats, for as we will see later, it allows for a better range to the changes of control within the instruction memory,
- these same operations should also, support the displacement, and immediate methods of calculating the target address,
- a forethought, regarding any future expansions of the instruction set, allowing for the option to expand without making the operations harder to read, or processed,
- and finally, the use of a suitable number of registers.

What we will **not** cover, in this endeavor, include the construction of a suitable compiler backend, that would target our architecture, and the design of advanced "hazard control" mechanisms.

We will produce suitable pieces of code, to test the simulation of our design, non of which will challenge it in an unforeseeable way. There won't be a need of such a test since, with VLIW computers, the compiler is the one typically handling those "hazards".

Chapter 2 Literature Review

(The Resources we used 2.1)

For our project, we relied on the following scientific books: Computer Organization and Design, as well as, Computer Architecture a Quantitative approach. Both of these books, are authored by the same pair, John L. Hennessy, and David A. Patterson. Within these two books, the implementation of a central processing unit(CPU) is explored, and made clear.

The Authors do not just present a single simple example of a hypothetical CPU. Instead, they go above and beyond, presenting the design of processors that have been both tested, as well as, implemented in the real world. And even though advanced ideas are presented, they are made clear through simple, and gradual examples, to the reader.

An example, which answers the question of "what is a VLIW architecture?", is given through a hypothetical example, in the chapter related to the processor, as well as through the very tangible Itanium, of the EPIC philosophy, within appendix H of the 6th edition Computer Architecture book, which dives deeper into VLIW, and EPIC.

In order to avoid being narrow-sighted to other sources of information, that could add or possibly even refute the very credible source that we already have, we also delved into the computer organization book of the C. Hamacher, Z. Vranesic, S. Zaky. However, although a refreshing reading, what we cover in this paper, for our project, can be solely based on the books of David A. Patterson, and John L. Hennessy.

It should be noted, that we studied the American edition of the books, as translated in Greek. This means that since the project paper will be written in English, we won't be able to directly quote anything from the book. We will in such a case however, make our best effort to interpret back into English.

(More books and other literature 2.2)

Our research also included books, which allowed us to demystify the aspects of computer architecture which border other fields of computer science. These books included Operating System Concepts by Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne, as well as the Compilers book written by Alfred V. Aho, Jeffrey D. Ullman, Ravi Sethi, and Monica S. Lam.

Beyond books, we learned more on the topic of computer architectures, by visiting the STMicroelectronics website, where we were able to study reference manuals, as well as programming manuals on RISC(Cortex-M0) and VLIW(ST240) architectures. Further research lead us to the "i860 64-bit Microprocessor Programmer's Reference Manual", which we also read.

In order to learn more about the origin of the Very Long Instruction Word paradigm, we also visited wikipedia, which lead us to learning about Joseph A. Fisher and his work on compilers, and VLIW. Through wikipedia's VLIW page, we were also able to locate interesting examples of commercial VLIW architectures, which we were able to include in the corresponding chapter. The examples we located included:

- the Fujitsu FR-V processor
- the SHARC architecture
- and Multiflow.

This prompted us to visit the websites of Fujitsu, as well as of Analog Devices. We also read the book written by Elizabeth Fischer, "Multiflow Computer: A Start-up Odyssey", through which we learned more about the history of the company Multiflow.

Finally, we also read two of Joseph A. Fischer's published papers. Both of those papers would prove a fascinating read to anyone wanting to learn more about VLIW architectures, as well as the compilation techniques that surround them.

Now that we are familiar with what our sources, and literature are, we should be ready to proceed into the subject of the matter!

Chapter 3
Examples of commercial VLIW processors

(Chapter introduction 3.1)

There are examples of VLIW processors that had a commercial impact on the processor market. A trait which many of those processors share, is that they are a great example of a low-power, embedded processing unit(pu). One such processor was the FR-V.

An example of a more “general purpose” VLIW pu, prior the development of Itanium was i860, which was also developed by intel. Although i860 was mostly succesful within the embedded processing unit market as well, it had many characteristics of a general purpose pu.

A mention should also be made, to what we see fitting to call the “grandfather” of VLIW processing units, Multiflow’s Trace line of processors. We call this line of VLIW processors the grandfather of VLIW processing units, for various reasons. Most important of these reasons, is the fact that Multiflow, was not only the first company to introduce a commercial VLIW processor, but it was also a company co-founded by Joseph A. Fischer, which was the person to first propose the benefits of a theoretical VLIW processor.

Finally, a VLIW processing unit which sold a great number of units, and is still available for purchase today, is the ST200 family of processing units. This processor comes with many features that are nowadays considered important to exist on a processing unit, and it a great example of what a modern day VLIW processor is.

(i860 3.2)

i860 was designed by intel. It was a RISC, VLIW instruction set architecture which was introduced commercially in 1989. I860's VLIW design, allowed it to be able to "run" two different operations at the same time. The second operation was always one of the "floating-point" type.

An interesting point of i860's floating-point instruction set architecture, was that there was the possibility to execute a combination of floating -point operations, as long as the two operations were on of addition, and one of multiplication. This meant, that in theory, the VLIW mode of i860, could execute up to three operations per clock cycle.

We previously used the word "mode" to make notice of the fact that, i860, could be run in a single instruction, non-VLIW mode. In this paper, we are going to focus on the VLIW mode, which can fetch and decode 64 bits as a single VLIW instruction. 32 bits for the integer, and logical operations, and 32 bits for the floating-point operations.

I860 had:

- thirty-two 32-bit "general purpose registers"
- and an equivalent amount in bits dedicated as a "floating-point register file" that could be acted upon, as if it was "split" in a variety of many different bit sizes(which corresponded to data sizes).
  For instance:
  - thirty-two 32-bit,
  - sixteen 64-bit,
  - or eight 128-bit registers.

This notable feature of i860, the ability to act upon a single bunch of denominations of a single register file, was meant for the faster determination of the value, in operations related to pixels. A very interesting feature for it's time, even though similar operations of the SIMD nature are more commonplace nowadays.

(FR-V 3.3)

FR – V was a product of it's time. In the late 90s, and early 2000s, there was a drive on the pu market, to have processors with vector/media capabilities. "People" who didn't adapt, risked commercial failure.

Able to start eight operations per clock cycle, FR-V was a processing unit that that for it's time, it was very hard to compare it's performance to that of other processing units. In "Computer Architecture, a Quantitative approach", it is shown how the number of instructions(operations) per clock cycle(IPC) is an important performance metric, when it comes to VLIW processing units. Based on this performance metric alone, it would rank quite high.

(Super Harvard Architecture Single-chip Computer(SHARC) 3.4)

First introduced back in 1994 by Analog Devices, this processor was designed as a digital signal processor. A Harvard architecture processor, as in it's name, SHARC has a separate memory storage for it's instructions and data. SHARC is a VLIW processor which can perform operations on:

- 32-bit integers,
- and single precision(32-bit) floating-point operands.

SHARC has two sets of registers. This feature gives to a processor the ability to handle a context switch well.

SHARC instructions, have a size of 48-bits. Instructions which use an immediate field(to obtain an opperand), are usually a single operation. While non-immediate field instructions, can perform multiple operations at once. This means, that SHARC doesn't act as a VLIW processor, for operations with an immediate operand.

Also, specific SHARC instructions have the ability to be ignored/discarded, under certain conditions.

(Multiflow's Trace -/ 3.5)

Multiflow was a company founded by scientists, and as such their products were oriented to solve scientific problems. As we have seen in the example of other commercial VLIW processors, we may notice a trend which suggests, that VLIW processors, are "naturally" performing their best when used for scientific applications.

The Trace computers come from an age when computers were comprised of big boards, each of which were meant to implement a function of the computer's processing. Even though very different to the modern day computer physically, it's goals and execution were not so different at all.

In a sense the "grandfather" of VLIW processing units, the "7/" model of the "-/" line of Multiflow's Trace computers, was also a 256-bit instruction VLIW, similar to our design in this paper. Unlike our own design though, which also borrow some minor elements from the Epic paradigm of Itanium, the Trace 7/ was a VLIW processor at it's purest form.

The seven operations each instruction could perform were each of a strictly defined type:

- Four arithmetic and logical operations,
- two floating-point operations,
- and a single branch operation.

Similarly, the Trace 14/ would have a 512-bit instruction, and could perform double the amount of each type of operations.

Because of how long ago this line of computers was first introduced, it is no wonder the only remaining physical examples, are held in museums. As such, the best way to learn more about it's inner workings, is to study the compilation techniques, of the compiler, whose name is given to the computers, "Trace". For it's day and age(late 1980s), having the ability to start as many operations as the Trace line of computers could, was no small feat. However, their production was stopped, and only the compiler survived the test of time, being used as an inspiration or outright in other VLIW projects.

(ST240 3.6)

The ST240, is a processor in the line of ST200 family of VLIW embedded processor units developed by STMicroelectronics in cooperation with Hewlett-Packard(hp). Interestingly enough, Joseph A. Fisher was involved in the production of this family of VLIW processing units as well. Still produced and "evolving" today, ST240 is a VLIW processing unit with access to many modern features that a processor's Instruction Set Architecture(ISA) should have. Such features include:

- the access to "predicated" operations which allows the processor to handle branch predictions a bit more gracefully than previous VLIW design attempts.
- and "multiprocessing" support with the existance of operations that provide "atomic" functionality.

ST240 uses a 6-stage pipeline which includes the following stages:

- Fetch(F),
- Decode(D),
- Read(R),
- Execute 1(E1),
- Execute 2(E2),
- and Writeback(WB)

ST240 was the first processor of it's family, of units, with access to floating point operations. However, the example of IEEE 754 standard is not followed faithfully by these floating-point operations of 32-bit operands, as mentioned in the reference manual. The processor can access sixty-four individual 32-bit integer registers. The processor can "read" from up to nine registers, and "write" to up to five registers. Each of those register accesses, must not share a register. The numbered "0" register of this ISA, has similar properties to Mips' "R0" register, as we have seen in the Computer Organization and Design book. By that we mean that, it is always read as if it has the integer value of '0', and whenever it is the destination register of an operation, it's value is never altered. Additionally, usage of register numbered "63" is usually best to be avoided, since ST240's design uses it as a special "linking" register. Floating-point operations, seem to use this very same register file.

The Instruction Set Architecture's "operation grouping" rules(were multiple operations are grouped into a single VLIW instruction), are not implementation specific. This means that they can work with newer implementations, and they are also subject to the current implementation's limitations. In the case of an invalid instruction, the ST240 design handles it as an exception. Whether there is a specific rule on how long can an instruction be in the ST200 family of processing units is unclear. However, each implementation has a "natural" limit derived from the number of available execution units, as well as from the "grouping" rules. Currently, ST240 seems to be able to initiate four different operation syllables per clock cycle.

The execution units available to ST240 are as following:

- Four ALU units,
- two multiplication(integer) units,
- one division(integer) unit,
- one branching unit,
- the two newly introduced floating-point addition units,
- and the unit responsible for the loading and storing from the registers.

Chapter 4 The New Architecture

(How is it similar to others? 4.1)

It would be very difficult to create a completely new architecture, that would also perform as expected, from the moment of it's conceivement. For this reason, we will use positive elements from all the architectures that have inspired us during the research for this project. A good starting example, would be the instruction formatting of the RISC-V Instruction Set Architecture, and the datapath elements of a typical RISC pipeline.

As mentioned in the 6th edition, of the appendix A, of the Computer Architecture a Quantitative Approach book, RISC-V, has many positive traits that any new Reduced Instruction Set Computer Architecture should try, and emulate. By simple having a similar format, our design inherits many of those positive elements. Similarly, explained thoroughly as it is, the datapath of the theoretical implementation of MIPS, as seen in the computer Organization and Design book, has many elements that have passed the test of time. Adopting similar hypothetical elements or even replicas of the existing theoretical elements, would give our new design a more realistic final state.

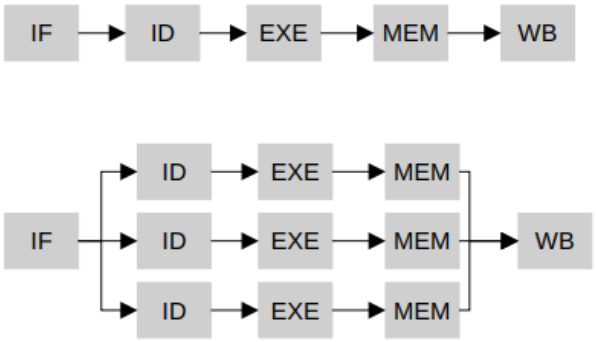

Image 4.1

There are also similarities with other VLIW architectures. The similarities are based around the fact, that while the datapath starts with a single Very Long Instruction Word, it seems to be, in a way, "forking" down it's length, creating multiple paths parallel to each other! Image 4.1 above, depicts this comparison, using a five stage datapath, similar to the one presented in the example pipeline of appendix C.1 in the CAQA book. In the same image, the hypothetical VLIW design implied, seems to have it's multiple paths meet again at one point. This is what we will also see happening in our own design later.
Similarly, the way Itanium schedules operations, within the same instruction, based on their interdependencies, could inspire a newer VLIW architecture.

(How is it different to others? 4.2)

While Itanium had more bits assigned for each individual sub-operation of it's instruction, compered to our design, it also has a far less amount of total bits in each individual instruction. With our instruction length of 256 bits in total, and a substantial 32 bits assigned on interdependencies, and similar functions, we have far more flexibility when it comes to this part of our design.

(Getting Started 4.3)

For this project, we assume that the reader has read, and is familiar enough, with the processor design of MIPS[COD, Chapter 4], as well as, with the RISC-V design[CAQA, Appendix C] presented in the books mentioned earlier. This will allow us to present something new to the reader, based however on something they have a good understanding of!

We will begin the presentation of the additions, on the very start of the data-path, the Instruction Fetch(IF) stage, and we will end with it on the very end, the WriteBack(WB) stage.

(Getting things in line 4.4)

This processor's datapath, shares the typical stages present in a typical five-stage RISC datapath[COD, chapter 4]:

- Instruction Fetch(IF),
- Instruction Decode(ID),
- Execution(EXE),
- Memory Access(MEM),
- and, WriteBack(WB).

As seen however, in appendix A of the CAQA book, a modern pipeline's data-path can have more stages suitable to it's implementation needs. Similarly, our processor's design would either, have a much slower clock cycle(cc) for each stage to complete it's function in, or we would need to add a few stages in order to re-duce the extra cc overhead.

The reason as to why, the clock cycle of our processor, would probably, be slower than that of the typical RISC processor, is that as with it's superscalar cousins, it would probably have a sophisticated system which distributes, the "data" in each parallel path of the "fork", which was seen depicted on image 3.1 earlier. Similarly, it could share the superscalar trait of having a commit unit. Unlike in the case of Out of Order execution though[COD, Chapter 4], this unit's purpose would be to simply accept and properly submit the result produced by each execution unit, at the end of the previous clock cycle.

As such, our processor design, could in theory have two extra datapath stages. This would allow for the clock cycle to stay lower, as well as providing a way, to make these two new additions distinct, from the rest. Whether this would be the case in an actual real world implementation of this processor, as with any processor, remains to be seen. Since in reality, five stages, seven stages, or a different number of stages all together, could be the one providing the most performance to our design.

(Instruction Fetch, the beginning 4.5)

Our datapath begins with the Instruction Fetch stage. During this first stage, an instruction is fetched from the instruction memory. Then, while the appropriate data is passed on their proper destination(towards the next stage), there is an increment of the Program Counter(PC) register, which points to the instruction to be read during the next clock cycle.

What makes it different to the RISC datapath we are familiar with, is the fact that not only has it got to fetch a longer instruction(256 bits vs 32bits or 64bits), but it also has each part of this longer instruction been sent to a different, yet similar, destination. Seven 32-bit portions are sent for the ordinary operation decoding, while the 8th and final one is sent to "explain" how the dependencies are to be handled. This is also the reason why, the amount by which the PC is incremented, at the end of each cycle, is equal to 32. As many as the bytes of data within each individual instruction! This is a lot compared to the 4 bytes of data an instruction of a typical 32 bit implementation has.

Another addition to the Instruction Fetch mechanism, is calculating which operations are actually meant to be performed, and which should be replaced with a no-operation, in a way that will be seen later.

(Instruction Decode 4.6)

During decoding, the seven 32-bit operations are individually handled very similarly to the typical RISC process. The register fields are sent to the register files as inputs, in order for the appropriate registers to be read, the immediate fields are extended and passed on to the next stage, and the function and opcode fields are decoded in order for the appropriate control signals to be produced. As such, there needs to be multiple copies of this decoding unit, as well as a register file which can support multiple concurrent inputs, and selective reading depending on how many signals are active.

Since our processor will be able to support floating point arithmetic, there will be the need for a separate Floating Point Register File(FPR). Both of these register files, will be read concurrently, and the data, that will be selected through a multiplexer, will be appropriate to the "nature" of the unit that will use them. As such, it is important to note, one of the first functions of our 8th syllable, the one of showing whether the execution unit about to be used, makes use of integer, or floating point registers! Of the 32 bits in the syllable, each of the seven operations will have 3 bits assigned to them. The first of these three bits, will signify whether we choose the data read from the integer register file, or from the FPR. The other two bits, will in-fact specify which specific execution unit type from a list of four different options(2 bits available), will be the one in use by this operation.

(Execution Units 4.7)

For now, we shall skip the newly added Distribution stage of the datapath, and proceed with the familiar Execution Stage.

For our processor, we will have 4 different types of execution units. Two are meant to use floating point registers, and two are meant to use integer registers. Based on what we talked about earlier, this leaves four extra unit type slots available to be filled, for our processor. There is one integer unit, the so called "mainUnit", that will in fact be based on the typical integer execution stage of the typical RISC datapath.

The "mainUnit" will make use of a very similar Arithmetic Logic Unit(ALU), as presented in the computer organization and design book, in both appendix A, as well as Chapter 4. This ALU, will allow our execution unit access to a variety of operations it provides. In order for the ALU to function, two of the signals produced during decoding, in conjunction with another part of the operation, will be used as an input on a ALU Control Unit(ALUOp1, ALUOp0). The resulting signals produced by the ALU Control Unit, are then used as an input by the ALU itself. Based on this input, the ALU will provide a specific functionality. In order for our "mainUnit" to make the most out of this ALU, we will use appropriate multiplexers, that based on a signal(ALUSrc) shall be able to choose between the extended immediate, or the secondary register(rs2) read during the Instruction Decode stage. This will allow our architecture access to I–type instructions, which are instructions using the extended value from the immediate field of the operation syllable.

Our "mainUnit"'s ALU, produces two outputs. The output proper, showing us the result of the operation, and a single bit with the value of one when the output's value is equal to zero. This singular bit output, from now on referred to as signal zero, will allow us to have a new group of operations presented later. This group of operations, will be the branch-type instructions!

Instructions affecting the control flow of the program, without a condition, also have their target address calculated during the execution stage of the "mainUnit" and passed on forward, on later stages. A single bit control signal, is responsible for the activation of this function, and the operations are named "jumps".

The "mainUnit" takes a single clock cycle, to produce an output. This makes it the fastest of the units that will be present on this project. There are three more units:

- the MulDiv unit, responsible for the multiplication, and division of integers,
- the FloatingPoint Unit, responsible for floating point addition, as well as multiplication,

- and the PackedFP unit, which is providing the architecture with Single Instruction Multiple Data(SIMD) operation, performed on floating point registers.

We will examine the rest of the execution units later, in order to first give a presentation of the datapath stages. The Main Unit, for now, is a good representation of the execution stage.

(Memory Access 4.8)

Having just received the results of the execution stage, we then move to the distribution of Memory Access. This part could also be a stage of it's own, and since it is a newer addition to the datapath, we will explore it after first visiting the five typical stages. As such, we move on with the assumption that the output of a single execution unit, has been passed on to a memory access point.

During the Memory Access stage, the result produced earlier by the Main Unit's ALU, is used to calculate the target address from which we either read, or write to the System Memory. For this purpose, there will be two signals. A MemWrite signal, and a MemRead signal. Both of which, when set to their active value perform their function. In the case of writing to memory, the contents of the appropriate register are transferred to the selected memory address. While in the case of reading, the contents of the memory from the specific address, are transferred to the next stage, in order to be submitted to the appropriate register.

During this stage, Branch-type operations are also reaching an important checkpoint. Not only is there an appropriate signal active when a branch operation is active, in order to activate their function(the branch signal), but there is also a functional element which provides the choice between four different Branch-type operations.

This functional element, receives a 2-bit signal in order to decide which of the four branch operation comparisons is performed(CondOp1, CondOp0). If the condition of the selected comparison is met, then in conjunction with the branch signal, through an AND logical gate, the pcsource signal activates. This signal signifies a change in the value of the Program Counter, thus affecting the control flow of the program.

The 2-bit CondOp signal allows up to four different options. Those four operations are:

- beq, checking for equality between the two registers(rs1,rs2),
- bne, checking for inequality between the two registers,
- bgt, checking whether the first register(rs1) is greater than the secondary(rs2),
- blt, checking whether the first register is lesser than the secondary.

For beq, and bne, the comparison is performed by subtracting rs2 from rs1(rs1 - rs2). When the two numbers are equal, the resulting signal zero has the value of '1'! As such, for beq the value of '1' is simple passed on. While the value of '0', in the case of bne, passes through a NOT logical gate first, creating the resulting condition signal.

For bgt, and blt, there is also the same subtraction of the registers. The difference is seen on the how the result is handled. Since a negative number's most significant bit, is always equal to '1', when the first register is greater than

the secondary, the value will always be equal to ‘0’. While when the opposite is true, the value of the most significant bit is equal to ‘1’! As such, for the condition of bgt, the most significant bit, is simply passed through a NOT Logical gate, while in the case of blt, the value of the most significant bit, is directly the one to be checked.

(WriteBack 4.9)

Entering the WriteBack stage, with all the previous stages having been performed, we are finally on the final stage of our datapath! Within this stage of the datapath, three signals are of importance, and not all three should ever be active for a single operation. There is a signal responsible for writing on the integer register file(RegWrite), a signal responsible for writing on the floating point register file(FPRegWrite), and finally a signal responsible for deciding whether the input to be received by the appropriate register file, has it's origin within the System Memory, or if it is the result of the Arithmetic Logic Unit(MemtoReg). A single operation, does not currently target both register files, so not all signals can be active at the same time.

It is worth noting, that the Memory, as well as the WriteBack stage, will be a common ending point for all operations regardless of the type. This means that even operations that don't actively use one or both of these stages will have to pass through them. This also means, that since we will have multiple instances of WriteBack at the same time, the register files should be able to accept as many concurrent input, as well as signaling for each individual input. This is a thing, that would increase the digital circuit density of the register files.

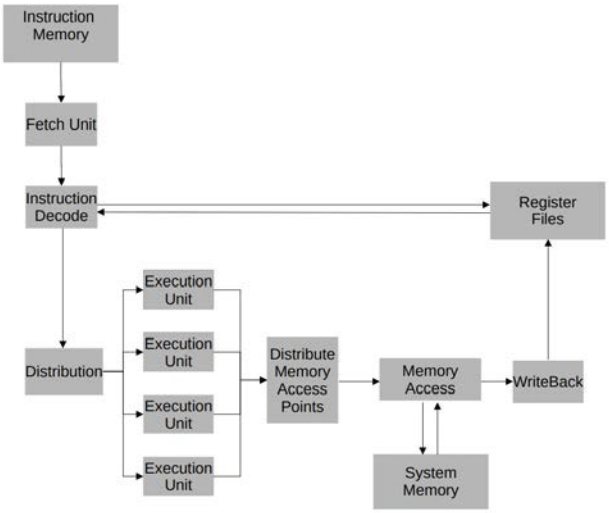

Image 4.2

An example of the architecture's diagram would look something like the image 4.2 above. However, we haven't yet explained what exactly would the nature of the Distribution stage, as well as the Distribute Memory Access Points stage, be.

(Distribute 4.10)

The Distribution stage, is a new addition to the typical datapath as we know it. Because each implementation of an existing Instruction Set Architecture could be different to one another, the way a VLIW architecture distributes each operation to an available execution unit, would possibly differ from one implementation to another. As such, we will give our own hypothetical example of this mechanism, whose duration we will theorize would be long enough, for the mechanism to be it's own stage.

There were two approaches for the Instruction Set Architecture, to handle this mechanism. The first one, would be to explicitly define which execution unit is assigned each operation. However, this would require a great amount of bits, of the 8^{th} 32-bit syllable, to be assigned on this task, which would reduce the ISA's flexibility, as well as it's expandability. By flexibility, we mean the different possible combinations, and options of Execution Unit types, while with expandability we mean the number of those Execution Units. Even if we were to give the architecture a satisfactory combination of Execution Unit options explicitly described on the 8^{th} syllable, future expansion of the Instruction Set Architecture would suffer, since the ISA would already be highly specialized on the one original implementation.

As such, we proceeded with the second option of allowing the hardware, to dynamically distribute the available execution units of the type assigned to each operation, through the 8^{th} syllable, and as such, gain in both flexibility, and expandability. This mechanism was named Distribution stage, and it's functionality is invisible to the designer of the binary code meant for the architecture, whether that is a programmer, or a compiler.

In order for the distribution to happen on schedule during this stage, preparation needs to be done during the Instruction Decode stage preceding it. During the instruction decode stage, the $8^{th,}$ 32 bit, syllable is also sent for decoding on a special decoding unit named "Unit Distribution Control". This unit has two input sources: the 8^{th} syllable, and a special purpose register file containing a special purpose Unit State Register for each available unit type(four in our case). Each Unit State Register, represents an execution unit type available to the architecture. Each bit, represents a specific unit's availability, with a value of '1' showing that it is unavailable for use. The output of this Unit Distribution Control(UDC), is a n-bit number for each of the operations that are about to proceed to execution, which identifies the execution unit to receive the operation. As such, when the time of the Distribution stage arrives, the Assign Unit Control(AUC) receives each operation's identification number, and signals appropriate values to the multiplexers responsible for assigning operations to each execution unit. In the case in which an execution unit wouldn't be in use, by an operation, an appropriate no-operation(An operation without an actual result) would be assigned to it instead.

Two operations assigned on the same unit would be impossible, and as such responsibility for the proper timing of operations, so that there are available units, falls to the compiler.

What some may notice is, that with the current design, there would always have to be a cycle change between the final operation of the very long instruction, to the first operation of the following instruction. A solution to this would be reading two instructions during the instruction fetch process, and making appropriate modifications to the units responsible for the distribution in order to only allow the operations meant to be executed to proceed. The changes wouldn't have to be tremendous. Simply adding a few extra bits, which would be as many numbered, as the number of operations in two instructions(14), would allow the architecture to signal with the use of '1' which operations are meant to be executed during the current cycle. Operations marked with a '0', will instead be replaced by a no-operation. This means, that this series of bits would also have to be passed on the next stage, in order to be an additional input to the Unit Distribution Control. An extra register would also be needed, keeping track of which was the last "marker", in the 8th syllable, signifying a change of cycle, to be activated.

(Distribute Memory Access Points 4.11)

A similar mechanism will reassign the output, from each independent execution unit, to the appropriate Memory Access Point. We will call this stage Distribute Memory Access Point(DM), and similarly to before, any Access Point that isn't explicitly assigned a valid execution unit output, will be assigned with signals representing a no-operation. There are as many Access Points, as there were decode units.

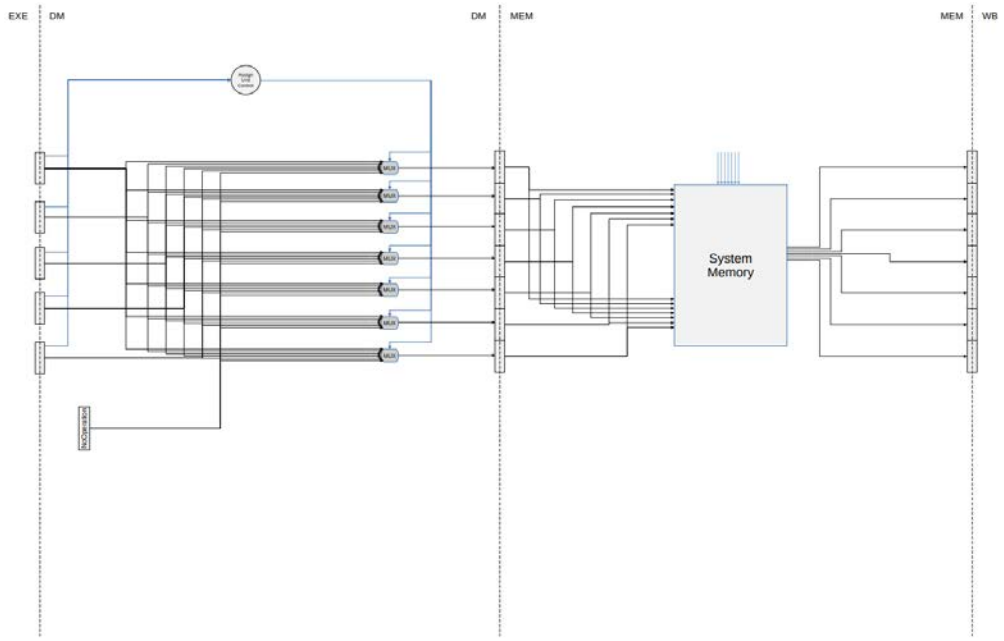

Image 4.3

This mechanism will use an Assign Unit Control(AUC) unit, which will receive from the execution units, the identification of their operation, produced earlier during the Distribution stage. Execution units operating on no-operations, will have appropriate identifications, and the AUC will be able to produce a new collection of bit strings, which it will send to the multiplexers of each Memory Access Point. This idea is presented visually, in an approximation, in the image 4.3 above.

(Other Execution Units 4.12)

(Integer Units 4.12.a)

The time has come to talk about the rest of the execution units. The MulDiv unit, uses a mechanism similar to the one presented for MIPS in the Organization and Design book, chapter 3. There are two 32 bit registers involved, named Hi, and Lo. This two registers have the result of both the multiplication, as well as, the division placed within them.

When multiplication is concerned, the result of a 32-bit number multiplied with another 32-bit number, can produce a number greater in length than that of 32 bits. Since our integer registers will have 32 bits in length, any resulting number longer than 32 bits will be considered overflowed. We can check if a result is overflowed by examining the value of the Hi register. Since the rightmost bit of the result will be placed in the rightmost bit of the Lo register, any value within the limits of the Lo register will be within acceptable limits, as long as there isn't a noticeable overflow passing onward to the Hi register. Similarly, the results of the division shall be placed within those two, available to the unit, registers. For division, there are two output results. The first one is the result of the division(quotient), and the second one is the remainder of the division. The quotient will be placed within the Lo register, while the remainder will be placed within the Hi register, giving us a way to get the result of both the "div", as well as, the "mod" operations concerning two 32-bit numbers.

In order to transfer the contents of Hi, and Lo, into a target register, our architecture will use an extra signal called "RegSrc". When RegSrc has the value of '1', the result typically expected to land within the target register, is replaced with one of the two(Hi, and Lo). This creates the need for another single bit signal "HiSwitchLo", with the value of '1' meaning the Hi register.

The MulDiv unit, responsible for the division, and multiplication of integers, will share resources. As such, a single copy of the unit, won't be able to perform both of these operations at the same time. It should also be noted, that typically division takes a lot more time to be performed, compared to multiplication, which may result, in computer architects, adding an additional unit dedicated to the task. Luckily, the unit can still be designed in a way that multiplication, and division take a different number of cc to be performed, thus not slowing down multiplication.

When the question is, "How are these operations actually performed?", we respond with the following:

During the multiplication of integers, the MulDiv Unit, does a series of operations to produce a result. An important part of multiplication is figuring out the result's "sign bit". To do this we have a functional unit which compares the leftmost bit of the two integer operands. Since the two operands are encoded in a two's complement, binary, form, we have a positive result when those two bits

have the same value. Another functional unit, within the MulDiv unit, also checks the leftmost bit of the original operands. When it recognizes a negative number, it reverses all the number's bits, and adds the number one to that new value. This process gives us the positive(or in the reverse case negative) of the number we originally had, in the two's complement format. The step that follows, is performing addition multiple times(thirty-two in the case, of 32-bit numbers), in order to produce the results of a multiplication of the two numbers. The additions are performed in the following manner:

The resulting positive integers, from the previous step, are placed within two 32 bit registers, which we will call A, and B. We begin by adding those two registers together. Then, register A is shifted left by one, while register B is shifted right by one. The process is then repeated for another thirty-one times, thirty-two in total. The result can be more than thirty-two bits in total, and as such, both the register Hi, and Lo are used to keep the result, as mentioned earlier.

The final step is a simple one. We check what is meant to be the results new sign bit. If it was meant to be positive, we leave the result as it is, ending the process. If however the result was meant to be negative, we once again perform the process of reversing a two's complement binary number, in order to get our final result.

Similarly, during the division of integers, the MulDiv unit, uses a functional unit to find the value of the sign bit the result will have. The difference here, is that the Hi, and Lo registers actually hold two different "results" of the division, as seen earlier. Within the two registers A, and B, we have the dividend, and the divisor. If the leftmost bit of A, is equal to the leftmost bit of B, the result placed within Lo will be positive. While if the leftmost bit of A is equal to zero, then the result of the Hi register is positive. The process also makes sure for the two operands to have their absolute integer value, by giving them their two's complement positive value. The following step is having the divisor be subtracted from the dividend. When the resulting value is higher than zero, the Lo register(which originally has the value of zero), is shifted left by one, and the new least significant bit(LSB) is equal to one. On the other hand, if the resulting value is equal or less than zero, the register Lo is once more shifted, but this time the value of the LSB is equal to zero. In this case, the dividend is also returned to it's value before the subtraction, by adding the divisor. No matter which of the two possible outcomes came to be, the divisor is then shifted right by a single bit.

This process, for 32-bit integers, is repeated thirty-three times. Having been repeated as many times as it was needed, the process is then complete.

(Floating-point Units 4.12.b)

For the floating point units, we have registers of 128 bits each. The reason as to why, is because we want SIMD floating point operations to use the same registers as the simple floating point operations. Our hypothetical floating point units, will also have the characteristic of sharing resources between the functionalities of addition, and multiplication. This means that a single individual copy of the floating point unit, will be able to perform addition, subtraction or multiplication, but not more than one type, of the available operations, at the same time.

The modern day standard, in regards to the representation of floating point numbers in binary, is the use of the IEEE 754 standard. While it was first established almost four decades ago, it is the "golden" standard, and it has been receiving updates which aim to keep it up to the times. The way floating point numbers, using the standard, are stored in the computer memory is by splitting the available bits in three parts:

- the sign of the number,
- the exponent, which depicts the power of '2' with which the third part is multiplied,
- and the fraction, depicting a sum of the negative powers of two.

All these three parts coming together, comprise a way to depict floating point numbers, that has been found effective for many years.

The typical way to handle floating point addition, within the IEEE 754 standard, begins with comparing the exponent numbers, and shifting, the fraction of the one with the lowest value, to the right, until both exponent numbers have the same value. This is followed by adding those fractions, and determining whether the result is a positive or a negative number. The next step, is checking if the final number still obeys the rules of being a normally accepted number by the standard, and if not, turning the number back into an accepted form. Finally, we check for overflow or underflow, and there is also the rounding of the result. In case the result after the rounding isn't an acceptable one, the normalization step, and the ones following it, are repeated as necessary. For Multiplication, the exponent values are added, and then multiple additions are followed. Normalization, checking for overflow, underflow, and figuring out the value of the resulting number's sign, are also part of this process.

This is a very simplified presentation of the way floating point arithmetic works within the floating point units. It is also worth noting, that besides normal numbers, there are other numbers included within the IEEE 754 standard. These special case numbers, include the number zero('0'), as well as the infinities. Special case numbers, have set results when they appear(Multiplying by zero, results into zero).

It should also be noted, that when encoded into actual memory, an extra '1' is implied beyond the leftmost bit of the explicitly encoded bits. This implicit bit,

provides extra precision for the encoded number! During the operation performed, intermediate results are stored in registers with a number of bits exceeding the one of the precision. This is important for increasing precision during the rounding phase.

Since our floating point arithmetic units will have access to 128-bit registers, our architecture will support floating point arithmetic of various sizes. Those sizes will be the four most common, half precision (16-bit), single precision(32-bit), double precision (64bit), and quadruple precision (128-bit). In order to discard, and update the specific part of the 128-bit register the operation is performed on, each time, there will be two additional signals(FMTOp1, FMTOp0) which inform the appropriate mechanism.

For the Packed Floating Point operations, the floating point arithmetic used is very similar, if not exactly the same. The only difference is that the 128-bit register is meant to be loaded with multiple n-bit floating point numbers. This allows for operations related to linear algebra. Operations performed, for example, between two different arrays. The signals mentioned earlier, defining the precision of the number operated on, are used again, this time also defining the number of numbers operated on, and appropriately named for the "PackedFP" unit(VFMTOp1, VFMTOp0). For these operations to be performed in parallel, multiple copies of the floating point unit responsible for the operation are required. As many copies, as the number of n-bit numbers operated on, are needed.

From the above, it is noticeable, that the floating point operations have more "stages" to pass through, than that of the integer addition, and as such will need multiple cycles of execution to be performed. More cycles are also expected for the integer multiplication, and division. The MulDiv process we described earlier, for both multiplication, and division, could have taken more than thirty-three cycles in total to be completed. However, for the sake of the simulation we will create in this very paper, we will hypothesize that the number of cycles needed, for each of the execution units, and different operations available in them, are the same as those presented for RISC V, and MIPS R4000 in the appendix C, of the computer architecture a quantitative approach(6th edition) book. In this example, the different execution unit types, took the following number of clock cycles in order to complete their task:

- The typical integer execution unit, would take a single cycle,
- The integer, as well as the floating point, multiplication, would take seven cycles,
- The floating point addition, and subtraction would take four clock cycles,
- And finally, the integer division, would take a total of 24 cycles to complete their task.

Since the SIMD floating point operations, are practically the same to the ordinary floating point, they will also take the appropriate amount of cycles to be completed.

Having now presented the changes to what we would typically see in a RISC datapath, as well as the execution units available to our architecture, we should be able to proceed into the specific choices made regarding the Instruction Set Architecture. We should also note, that there won't be a focus to the output product of those units meant to help with distribution of execution units, and memory access points. This is because, those mechanisms are implementation specific, and invisible to anyone using the product architecture. We will aim to showcase any signals related to the actual Instruction Set.

(The New Instruction Set 4.13)

For our new instruction set architecture, we have decided to use traits from three different sources:

- The MIPS Instruction Set Architecture, as seen in the computer organization and design book,
- The RISC V Instruction Set Architecture, as seen in the Computer Architecture a Quantitative Approach book,
- and finally, the Itanium Instruction Set Architecture traits, as seen in Appendix H of the Computer Architecture book.

From MIPS, we are getting influenced by the way it's signals are produced, as well as used. A decoding unit will be used for each syllable's operation, in order to decode a collection of bits(opcode, and function bits), producing a series of bits which act as signals to the architecture. This signals, are the key to make the architecture do as it must, during each clock cycle.

RISC V has a very well made instruction format, as seen in Appendix A of the Computer Architecture a Quantitative Approach 6th edition book. This format will allow us to both make the best out of our 32 bits allocated to each operation, as well as provide us a proven, performance boosting, instruction format. Having a standard way to allocate the bits of each syllable, as well as a standard decoding process, provides us with a stable performance, across all the options of operations that will be available to our architecture set, or that could be available in the future.

Finally, the Itanium Instruction Set Architecture, brings the explicit nature, of the EPIC paradigm, on the table. With 32 bits dedicated to interdependencies, as well as execution unit management, our Instruction Set Architecture, might actually have more bits assigned to it, than Itanium ever actually did. This allows us to move away from the list of predetermined options Itanium had for this task, and instead opt for a collection of "switches" of sort. Activating these switches, by giving them the value of '1', means that a multitude of combinations is available to be chosen. These combinations, are only limited, by the actual limits of the real world implementation of the architecture using this new instruction set.

Since it is vital to any further discussion about the Instruction Set's "nature", we will begin presenting the formatting of the instructions, and we will explain why we allocated certain portions of instruction bits, to specific functions.

(The Instruction Format 4.14)

Our 32-bit syllables responsible for the seven operations executed within a single instruction, have a format very similar to the one RISC-V has in it's instruction set. In fact, the main difference is the sizes of the individual fields, since a VLIW architecture has some needs different to that of a typical RISC architecture. We will start by noting the difference on top of the R – type instruction format. This format, is one used by a substantial amount of operations including integer, and floating point operations. Usually the operations using the R – type format, have both of their operands read from the register file, and have their output also written to a suitable register. The architects the RISC-V Instruction Set Architecture, decided to use the same field of bits between the different instruction formats, in order to reduce complexity of reading different types of format, and improving performance. This means, that any changes on the R – type format, will in fact be very similar, if not exactly the same, on the rest of the formatting options.

Each bit of the thirty-two, within an instruction, is numbered. In RISC-V, starting from the rightmost bit, we could say that the first field of bits we meet is the "opcode" field. Taking up seven bits, this field a common sight in all the different formats, being the only one present in each single one. The opcode field is providing the 32-bit instruction with an identity, showing which format is used on this specific occasion. This means, the way that the decoding process interprets the rest of the thirty-two bits, is based on the value held with these first seven bits. In certain cases, they also play a substantial role in identifying a specific operation using the format identified. Having 7 bits available, the opcode has 128 different values it could take(2 ^ 7). This allows for a variety of options, and the Instruction Set Architecture can be expanded with many different operations. The other two field types, are the function fields, and the register fields. The function fields, which are two(funct7, and funct3), help distinguish between each individual operation together with the opcode. The register fields on the other hand are three in number(rs2, rs1, and rd), and are responsible for distinguishing between the different individual registers available to the register file. In RISC-V, the registers are 32 in total, thus making the size of the register fields equal to five(2 ^ 5 = 32). The function fields, with 10 bits, have 1024 possible different values. This makes for a total of 32 bits. What are then the differences with our new VLIW design? Thirty-two bits are enough for a RISC architecture, and expandability an important factor for this "general purpose" architecture. However, a VLIW architecture, with multiple individual execution units, working on a register file with just thirty-two different registers, could perhaps run in trouble. A single bit given to each register field, three in total, can make the register file feel less "crowded". Six bits instead of five, brings us to the total of 64 different registers, within each register file. However, these three bits need to come from somewhere. As such, we reduce the size of the function fields by two(funct7 –> funct5), and the size of the opcode field by one. With function fields of 8 bits, and an opcode of 5 bits, we have much less in terms of expandability. However, a VLIW processor, won't necessarily need many of the expansions available for the RISC-V Instruction Set Architecture. Many of these ex-

pansions are aimed for embedded systems, or to provide functionality that a VLIW architecture can provide in a different way. Comparing these field sizes with another architecture, the one of MIPS, also makes for an interesting comparison. In fact, MIPS which has had different expansions, in the form of co-processors, as well as a pretty successful commercial release in it's time, has a lesser amount of expandability options to RISC-V.

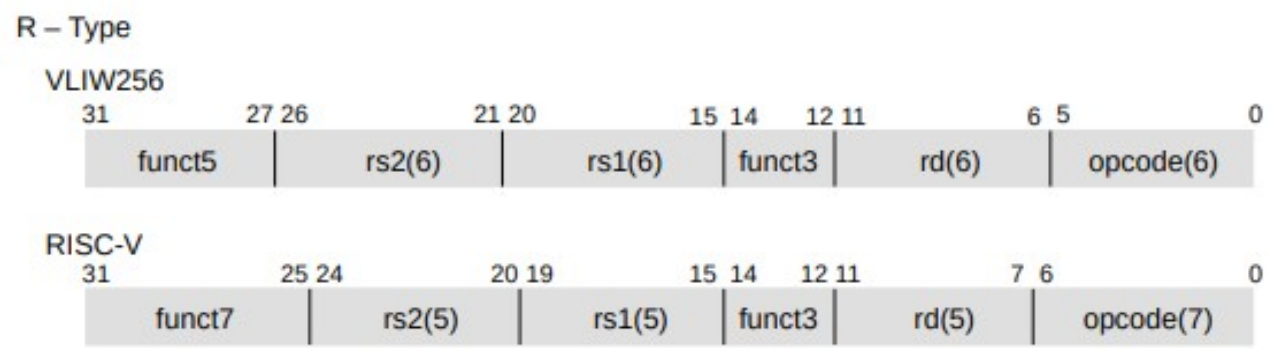

Image 4.4

The image 4.4 above, shows the comparison between RISC-V, and what we have in our instruction set for the R-type format.

What is made obvious from the image above, is the split of the function bits, in two different fields. The reasoning is simple. In the rest of the format options available, certain fields are replaced by the immediate field. The immediate field provides it's value as an operand to certain operations. In order to avoid repositioning the same individual field, to a different assortment of bits, between different instruction, the function bits were split in two. For example, in the I – type Instructions, the funct5, and rs2 fields are interpreted as the immediate field. And in U – type instructions, every single field instead of the rd, and opcode field, are interpreted as the immediate field.

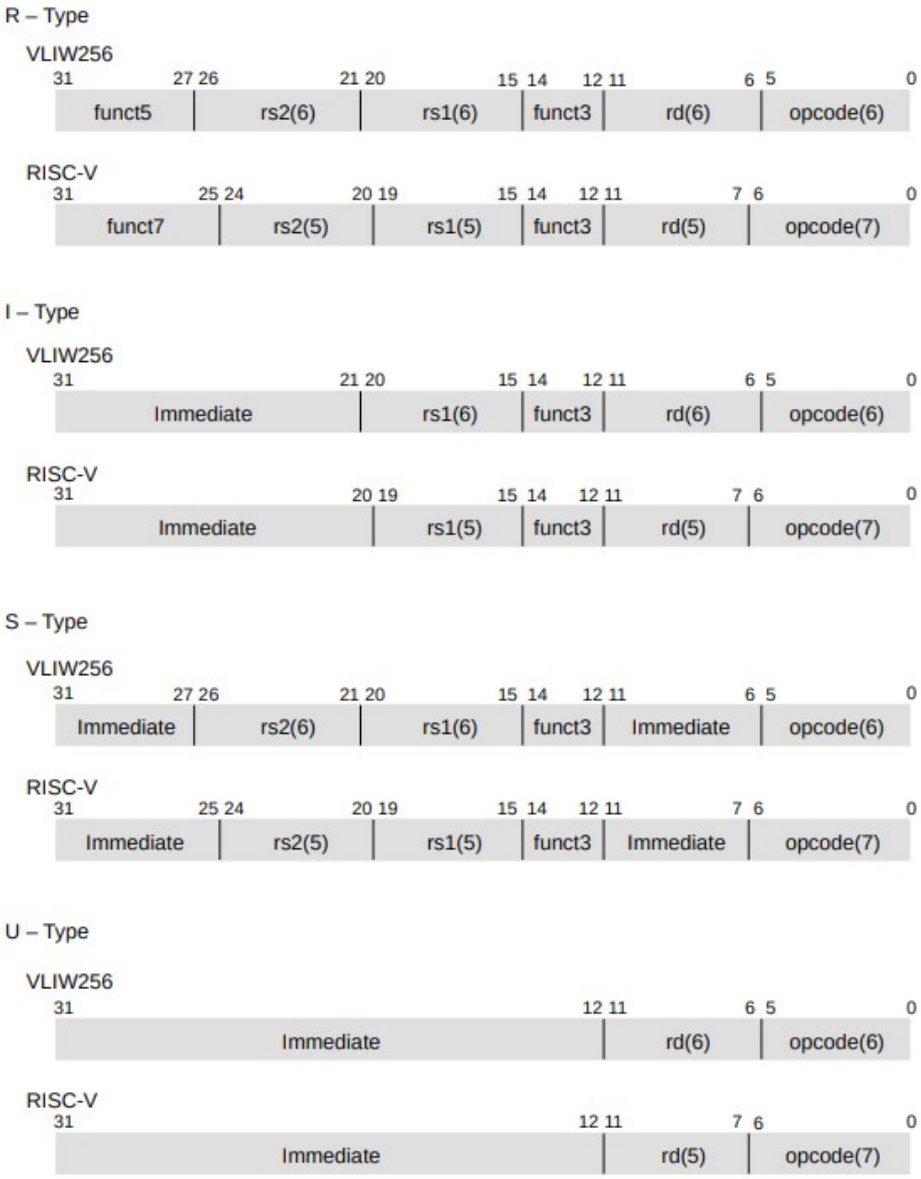

Image 4.5

In the image 4.5 above, we can see how the different fields are interpreted as an immediate field, depending on the operation's needs. On the S – type format, the immediate bits are even split in two different fields. With one, taking the place of the rd field, since usually the rs2 register field is used as the target of the specific operation type, and sometimes there isn't actually a target register(in the case of branch type instructions).

(The Immediate and Other Fields 4.15)

Another important use of the immediate fields, is the one present in the load operations(I – type), branch operations(S – type), as well as the jump operations(U – type). Once again the immediate field's value is used as an operand. This time however, the result of the operation is meant for a specific purpose. After the operation in which the immediate takes part is complete, it's result is shifted left by a value of five. This is because, we need the resulting value of the operation to align to memory addresses, and because our instructions are 32 bit long, we need the shift to result into a multiple of 32. This is also happening in branch operations, which use the S – type format.

RISC-V, and as a result our own architecture, allows for two ways in which memory addresses can be "calculated" through an operation. The Immediate, and the Displacement method.

It is worth noting, that while the instruction memory is indeed aligned in 32 byte segments, the way system memory is handled is not. When referencing system memory, the alignment is based on the FMT signals. For instance, when reading a 32 bit(4 bytes) memory segment, the shifting will be made by a value of two. While when trying to read a 16 bit memory segment(2 bytes), we would be shifting by a value of one. This is in contrast to what RISC-V does, where it can read segments out of alignment.

While in both instruction sets, the U – type operations provide 20 bits to the immediate, in our own VLIW architecture, we have two less immediate bits in every other instruction format. However, because our system memory, when referenced, is aligned, we have a few bits extra memory reference range, depending on the size of the segment read. In the case of instruction memory reference specifically, we have an even greater advantage. Instruction memory reference in our architecture is Program Counter Relative(PC Relative). This means that any changes in control flow, are only possible within a certain distance from the current program counter. Both in U – type instructions which have the same number of immediate bits, as well as in S – type, for branching, where we lack two bits worth of immediate, we are referencing an instruction memory segmented in 32 bytes versus the 4 bytes of the RISC-V instruction set.

A final thing worth mentioning, is in relation to the function fields. Specifically, in many types of operation, during decoding, we use the funct3 field(3 bits) in order to figure out the value of the FMT signals. This is for instance happening in the floating point instructions, using the R – type format. In which case, a floating point instruction would:

- Be recognized, as a floating point, by it's opcode,
- Be distinguished between the available floating point instructions(add.f, mul.f, etc) by it's funct5 field,
- and finally, it would operate on a certain data size, based on the funct3 field.

(Interdependencies in a Syllable 4.16)

As we briefly mentioned earlier, our instruction's 8th syllable, has the distinct purpose of managing interdependencies between the operations of the other syllables, as well as defining which unit type, each of this operations requires to use. With 32 bits allocated to this purpose, we have plenty to use, even leaving a few unused bits to spare, and perhaps use in future expansions!

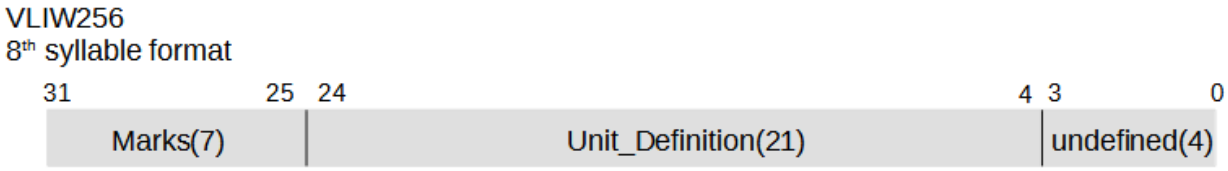

Image 4.6

In the image 4.6 right above, we can see that the rightmost 4 bits(0 – 3) are reserved for future use, while the rest find some use. More specifically, the "Unit_Definition" bits(4 - 24), which are 21 in numbers, are responsible for defining the unit type which the operations will use, and the "Marks" segment(25 - 31) is comprised of 7 bits, showing after which which operation we will have a cycle change. A single bit is a single mark, while for each syllable's operation, the unit definition needs 3 bits. While we number the bits of the instruction from right to left, the syllables are numbered from left to right. This means that the leftmost syllable, is the first one. Similarly, the leftmost mark, as well as the leftmost unit_definition, are the first of their type.

A mark is activated on a value of one. Each mark is in a way "following" it's corresponding syllable. By following we mean that when a mark is activated, all the syllable operations prior to it, and following the previous mark(when not the beginning of the program), are executed in parallel, during the same clock cycle.

The three bits corresponding to each of the seven operation syllables, function as follows:

- The leftmost bit of the three, functions as a way to distinguish between floating point, and integer units. When this bit is set on a '1', the unit type is floating point. The unit type is integer on a '0'.
- The following two bits, provide for up to four unique options of execution units of the type defined by the first bit. By default, each of the two types has two units.

(Future Expansions and Security 4.17)

We mentioned the existing undefined bits of the 8th syllable, as well as the possibility for, the addition of, more execution units, in our architecture. While we could present a substantial amount of possible expansions, suitable for our architectural design, we thought it would be better to "turn the spotlight on" a single expansion, that could also take advantage of existing functional units. This expansion, would be one aimed at increasing security through the usage of encryption and decryption operations as well as, through the use of "***levels of security***".

With the term "***levels of security***", we speak of a way to help the operating system, running on our computer architecture, to be able to mark certain operations, as capable of handling sensitive information. This is an important subject, which can heavily affect a computer's security, and through the years many are the ways with which computer architects, and security specialists, have contributed in this endeavor. Whether or not an operation is meant to act upon sensitive information, that is the job of the operating system(or the user) to figure out. Whatever the exact mechanism, handling this "sensitive information", might end up being, allowing some of our undefined bits(of the 8th syllable) to be used in order to provide a functionality like this one, and as such ***levels of security***, is an excellent usage.

Additionally, the functionality of the floating-point execution units, especially the one provided by the PackedFP unit, is suitable for helping us achieve the encryption/decryption functionality that is desired, in security related tasks. With the usage of a "keyword", and a series of arithmetic and logical operations performed on a certain string of binary information, we can have a piece of information encrypted(as well as decrypted), in no time. The keyword, would provide a way of decrypting, the using this keyword, string of information. Only using the keyword, with which it was encrypted, a string of encrypted information would be capable of returning to it's original "value". This functionality could then be made into an operation, of our instruction set, and thus provide a convenient way of encrypting/decrypting binary information, loaded on a register.

(Concluding the Chapter 4.18)

Through all of the above observations, and modules of our Instruction Set Architecture, we conclude this chapter, confident that the instruction set, has successfully handled what we originally wanted it to, back when we introduced our project's aim!

All that remains, is proceeding in the simulation of this instruction set architecture. We will set the parameters we want it to follow, and we shall then see it in action.

Chapter 5 Simulating the Architecture

(Chapter introduction 5.1)

The final step of our project, is the simulation of the new VLIW architecture, the one, which we designed within the previous chapter. To code this simulation, we used the "C programming language", and in order to better simulate the architecture, there are certain parameters that we aimed followed.

Certain of these parameters, have already been set within the previous chapter. These include:

- the number of stages,
- the cycles of execution required for certain operations,
- the undefined bits of our instruction,
- and others.

All of the above are things to take into consideration, for the needs of the simulation.

We will also assume, that the implementation simulated, will be fetching a single new instruction, having a cycle change, after each operation within a 256-bit instruction has been issued.

After the simulation is complete, we will be able to run examples of "binary" code, which will then produce appropriate results. We will provide a few guidelines on how to structure such code, and during execution, the results will be produced on a terminal.

(The Simulation's Structure of the Files 5.2)

A number of "C-code" files(.c), make up our simulation's code:

- Main
- NewPipeline
- BinaryEmulation
- DecodeUnit
- FloatingPoint
- FullAdder_and_FullSubtractor
- InstructionDecode
- InstructionFieldIDFunction
- MulDiv
- PackedSIMDFloatingPoint
- PrintSubwordInstructions

All of the above, perform a certain function(or even multiple functions) to achieve our goal.

A few ".txt" files were also used to help us achieve our goal. A file named "CurrentSystemState.txt" helps during execution, and a file named "Instruction-Memory.txt" acts as the system's instruction memory in our simulation's architecture. Finally, a set of thirthy-three text files(.txt), named "RegAndMem.txt"(example: RegAndMem0.txt, and RegAndMem32.txt) are used to help simulate the cycle latency, of the various operations. These files are all used by our simulation, and a person wishing to compile and run, our simulation, should have the files mentioned above within a single folder.

To compile properly, the code should also have access to the following files:

- "stdio.h",
- "math.h",
- and "stdlib.h".

(Simulating the Registers and the System's Memory 5.3)

(Definition 5.3.a)

To simulate the registers of our architecture we used the following technique. We defined our registers as arrays of shorter length integers, so that they take up less of our system's memory. We defined the two register files as:

- "short int $[NumberfRegisters][registerSize]", for integer registers,
- and "short int FPR[NumberofFPR][FPRSize]", for floating-point registers.

We also defined the System Memory as "static int MEM[RAMSize]". Given a "RAMSize", with a value of '1', provides our simulation with four bytes of system memory. Consequently, a "RAMSize" with a value of "268435456", provides our simulation with one gigabyte of system memory.

As mentioned earlier, we designed the architecture to have two sets of sixty-four registers. However, even when defined as "short integers", the register files and system memory can take up more space than what may be reserved by default for the stack(during the program's execution). To avoid this problem, one can set the values of certain variables in their environment of execution. We recommend this solution. However, we didn't want to use this solution in our paper, because it would be specific to our environment of execution, and of no use to a different environment.

As such, for someone wanting to run simple examples of "binary code" with our simulation, we suggest, before compiling, to either:

- make the appropriate "environment variable" changes or,
- to set the number of registers, and memory size, to a lower, permissible amount.

The second solution is very easy to perform, since all the person executing the code, has to do is find the "#define" statements, on the top(within the first 4 to 40 lines) of our code files. This change will have to be made in each ".c" file, to match the same value.

(Usage 5.3.b)

We talked about defining our registers and system memory, let's now talk about using them. We used two different formats for them, with our reasoning being that it would be simpler for the end user of the simulation to execute "binary" code examples. The two format's we used are as follows:

- The registers, are stored in a "binary" form of zeros('0'), and ones('1'),
- while the system memory, is stored as a decimal integer.

Since, we can fill the system's memory with integers(typed in decimal form), for the registers to be loaded with, our life is made easier when simulating operations of integers. However, it requires a bit more finesse to use when loading floating point operations directly from memory. This is especially true, because floating point operations are designed to be performed with variable precision(as seen earlier), and with operations utilizing the "PackedFP" unit.

In order to fill the memory with floating-point appropriate "binary", one would have to think of the number he would wish to see stored in the system's memory, the way it would be, in it's binary form, and then convert it to the form of a decimal integer.

(Operation Latency 5.4)

We originally designed a simulation that would produce a result, for every operation, within a single cycle. So, in order to simulate the number of cycles between issuing an operation, and the operation's execution finally producing a result, we used an interesting method.

We created a series of "RegAndMem"(numbered '0' to "33"), ".txt" files, which are as many as the number of cycles(+1) it would take for the most time consuming operation to produce a result. This way, when an operation is issued, the result is directly saved on the "RegAndMem" file, whose number is equal to the number of cycles it will take for the operation to produce a visible to the architecture result.

To perform this technique, before executing each operation producing a result(certain "flow-control" operations are excluded) during this cycle, our simulation first saves the current state of our simulated architecture, in a ".txt" file named "CurrentSystemState", and then reads, from a "RegAndMem" file, the state which it will have at the moment the operations result should be produced. It then produces the result of the operation, and saves this updated state on the same "RegAndMem" file. It then reloads the current state from "CurrentSystemState", and moves to the next operation, of the current cycle.

(Instruction Decode and Change of Cycle 5.5)

Once the executable of our code has started running, it defines certain variables within the main function, and it then calls the "NewPipeline" function in order to start simulating the architecture. Within this function, a variable representing the instruction register is created(IR), and the decoding process starts.

Decoding of the 8th syllable is done early, and the type of unit used by each operation, of the instruction, is printed on the terminal. Along with the decoding of the 8th syllable, we will have an early "decoding"(of all the syllables), showing as the architectural signals that will be produced, during decoding, for each operation of this 256-bit instruction. The same exact signals, will also be shown, when it is time, for each operation to be executed(unfortunately, not during the execution). It would be even more interesting to have, in display, the current value of each signal, in each component of the architecture. However, the output of the terminal would then be bloated with information, and it would be hard to keep track of the execution.

Finally, the "marks" pinpointing the change of cycle, is something we check after each operation of the instruction is performed. Because our program only imitates the existence of parallel execution, and in reality executes each operation(of a cycle) sequentially, we a have a "for()-loop" of seven repetitions whenever a single 256-bit instruction is "fetched" for execution from within the "InstructionMemory.txt".

(Instruction Execution 5.6)

(What is each C-file for? 5.6.a)

We have already described within the previous chapter, the design of each execution unit type. As such, writing in detail, how the result for each specific kind of operation is produced by our code, would be a repetition of the logic we have already seen earlier. However, we will indicate the path(in our code), that each operation has to follow, in order to produce a result. Likewise, anything of interest in our code, will also be mentioned.

In the "InstructionMemory.txt", the instructions are stored in "binary" form. For this reason, a function from "InstructionFieldIDFunction.c" is called, for each syllable of the instruction, in order to receive the "binary" fields, denoting registers, and transforming them to decimal integers. In order to perform this transformation(From binary to decimal), a function from "BinaryEmulation.c" is called.

Following this, another function is called from the "InstructionDecode.c" file, which is meant to recognize the operation's type and either:

- perform the operation in it's body(the function's), if the operation is of a small length(in lines of code),
- or call another function which is designed to execute the operations, which take up more lines of code.

These functions, which provide the code responsible for executing the lengthier operations are included within the following files:

- "FloatingPoint.c", for floating point operations of the "FloatingPoint" unit,
- "MulDiv.c", for the multiplication and division of integers of the "MulDiv" unit,
- and "PackedSIMDFloatingPoint.c", for the floating-point operations of the "PackedFP" unit

For the final steps of the execution, we return within the "NewPipeline" function, which updates the system's memory, and registers(In the way we showcased earlier).

(The rest of our C-files, and a flowchart of the code 5.6.b)

We also have other ".c" files included in our code. These files include functions which perform useful roles. An example of this is the "FullAdder_and_FullSubtractor.c" file which includes functions with the ability to produce results, that the logical circuits of the same name would produce(when given a binary input). And the "PrintSubwordInstructions.c" file which includes a small piece of code, useful for bug fixing.

We think a visual example, would do much better at demonstrating the code's "flow". For this reason, in the image 5.1, just below, a simple flowchart, which is a graph, is given. The flowchart, will visually showcase the way with which the most important bodies, of our code, are connected. Certain terms present within the flowchart, such as "system state", might not have been used extensively within this paper. However, we will explain them, in this chapter.

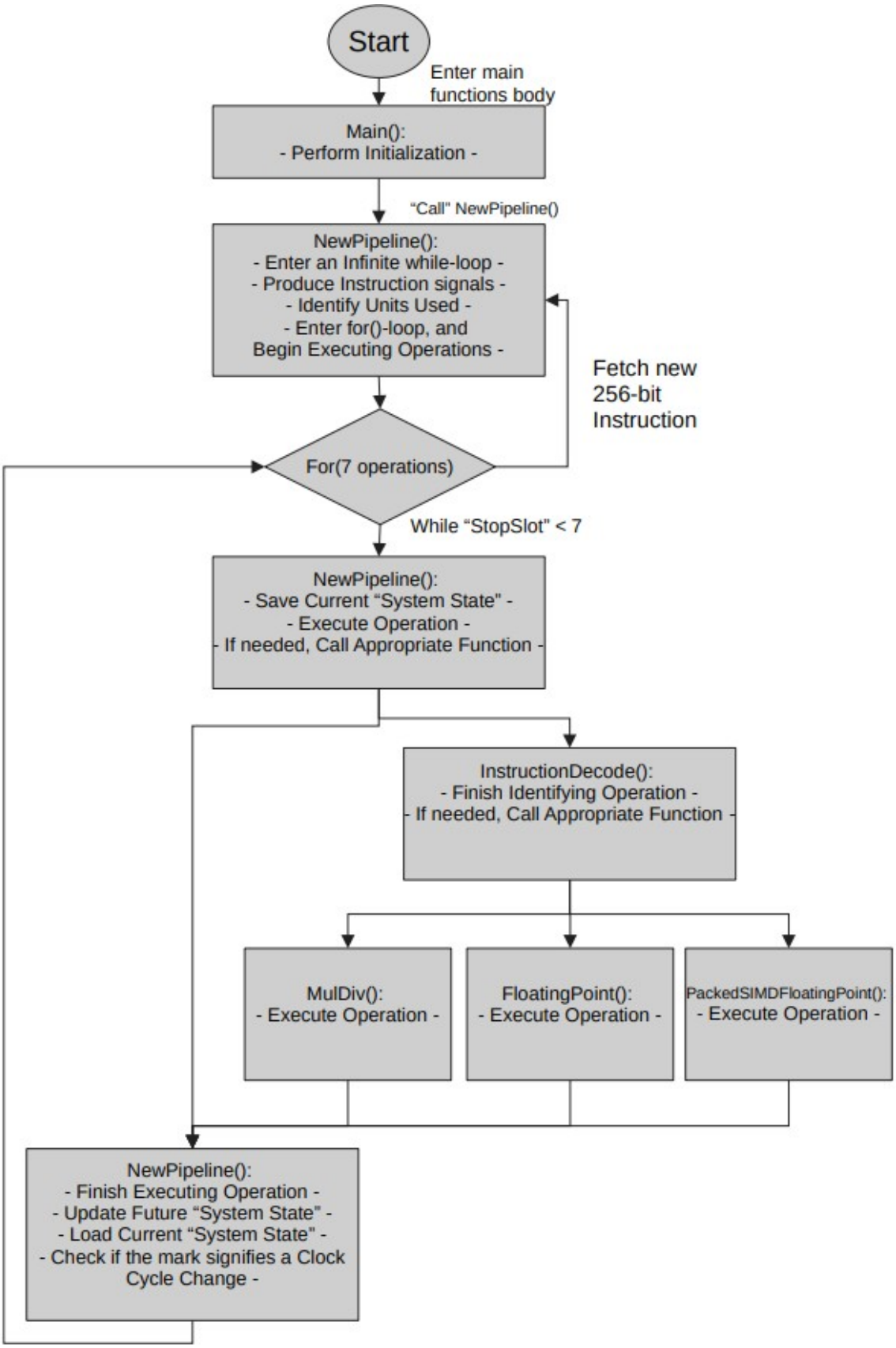

Image 5.1

(Executable's Output 5.7)

(Terminal's Output 5.7.a)

Earlier, we mentioned the terminal. The terminal is an important part of our simulation, since it will keep us updated, on several matters. The information provided by the terminal will include:

- The number of the current 256-bit instruction,
- the number of clock cycles, that have passed, since our simulation begun,
- The signals produced by each syllable, during the operation's decoding,
- The unit type, used by the operation of each syllable,
- as well as, reaching a mark, and with it, a change of clock cycle.

When changing a clock cycle, which includes fetching a new 256-bit instruction(in the currently simulated implementation, and not in every implementation), the terminal also informs us, and requests of us, to "produce" a button input("enter") in order for the simulation to continue. We coded our simulation this way, in order to be able to take our time and read the new information produced with every new clock cycle.

A secondary aim of ours, was to produce a ".txt" file with everything outputted in our terminal. This would allow us to read the terminal's output in our leisure, after the simulation's execution, as well as creating a graphical user interface, which could use this ".txt" file as an input, visually simulating the architecture. Unfortunately, this was never implemented. However, it could prove a useful expansion, or even something to add in a future project.

(System State Output 5.7.b)

We define the “system state” of our simulation as: the collection of the values that are stored within the registers, and the memory of our simulated system. In order to avoid outputting these values, by printing them, on the terminal(which would create an information bloat), our solution is a rather simple one.

At the very end of each cycle, the moment that is requested of us to press a button(“enter”), in order to continue, if we instead open the “RegAndMem0.txt” file, we get the system state, as it will be at the start of the next cycle!

(Executable Input 5.8)

(Instruction Memory 5.8.a)

In order to input specific values on the system state, before we run the executable, we would have to make changes on the variables of the system state, within the main function, of the “main.c” file. These additions, would have to be added, before the “NewPipeline” function is called.

In addition to the above, we can change the binary code, which we would like to have executed, by simply going to our “InstructionMemory.txt” file, and changing the values present there. A person aiming to do that should remember that each line, of the “.txt” file, will represent a single instruction(256 characters), and that each instruction is represented in “binary”. This means, that the acceptable input for each line is a string of zeros(‘0’), and ones(‘1’).

(List of Operation Opcodes 5.8.b)

These are, an example of, the opcodes in the hypothetical implementation of our simulation(table 5.1).

operation	opcode	func5	func3
add	000000(0)	00000(0)	000(0)
sub	000000(0)	00000(0)	001(1)
load(32-bit)	000001(1)	--	011(3)
store(32-bit)	111101(61)	--	011(3)
addi	000011(3)	--	000(0)
beq	111011(59)	--	000(0)
jumpAndLink	111111(63)	--	--
and(logical)	000000(0)	00000(0)	010(2)
or(logical)	000000(0)	00000(0)	011(3)

Table 5.1

These opcodes, can otherwise be found(and changed) by delving into our simulation's code, within the "NewPipeline.c", the "InstructionDecode.c", and the "DecodeUnit.c" files.

(Simulated Examples of Execution 5.9)

Having presented our simulator's functionality, and having given guidelines, for it's operation, we are ready to demonstrate our simulator's capabilities. In order to do that, we have a small example of a few 256-bit instructions.

For this example, and in order to have some non-zero values, we will initialize a few "cells" of the System's Memory(MEM[]).

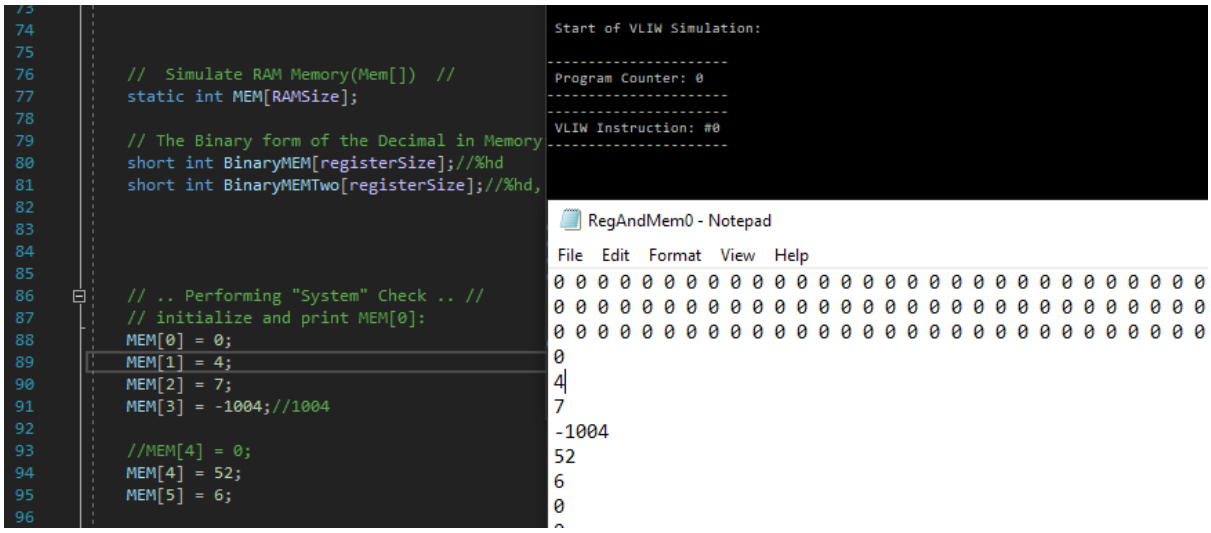

Image 5.2

As you can see in the image 5.2, right above, we initialize the memory, by giving it certain integer values. Afterwards, when we run the executable of our code, we can see that these values, have appeared, from the very beginning of the simulation, in the system's memory. Having now initialized our system's memory, let's review the code that we will run on it.

The example we have prepared for this demonstration, is a small one, and it will make use of just four different operation opcodes. The opcodes used, are of the following operations:

- the add operation, which will be made to act as a no-operation,
- the addi operation, in order to have our simulator, perform some basic arithmetic,
- the load operation(of single precision), in order to use the initialized memory,
- and the store operation(of single precision), in order for it to act as the "finish line" of our demonstration.

```
Instruction: 0

Operation:
1: Load  00000000001(1) 000000(0 rs) 011(single-precision)  000100(4 rd)  000001(load)
2: Load  00000000010(2) 000000(0) 011  000101(5)  000001
3: Load  00000000011(3) 000000(0) 011  000110(6)  000001
4: no-op 00000000000    000000    000  000000     000000
5: no-op 00000000000    000000    000  000000     000000
6: no-op 00000000000    000000    000  000000     000000
7: no-op 00000000000    000000    000  000000     000000
Dependencies:
8:       0001111(Marks) 0[Integer]00[mainUnit] 000 000 000 000 000 000(Units) 0000(Undefined)


Instruction: 1

Operation:
1: no-op 00000000000    000000    000  000000     000000
2: no-op 00000000000    000000    000  000000     000000
3: no-op 00000000000    000000    000  000000     000000
4: no-op 00000000000    000000    000  000000     000000
5: no-op 00000000000    000000    000  000000     000000
6: no-op 00000000000    000000    000  000000     000000
7: no-op 00000000000    000000    000  000000     000000
Dependencies:
8:       1111101(Marks) 0[Integer]00[mainUnit] 000 000 000 000 000 000(Units) 0000(Undefined)
```

Image 5.3

We will first issue three “Load” operations:

- reg[4] <= Mem[1]
- reg[5] <= Mem[2]
- reg[6] <= Mem[3]

These three operations will be followed by a series of “no-operations”, in order to give them time, for their “execution” to be complete. You can see all this, with greater detail in the image 5.3 above.

The way our instructions are presented in the images given here, is not the way they are written within the “InstructionMemory.txt” file. The reason for that is readability(a single line of 256 bits, is hard for a human to comprehend).

Various “addi” operations are then used to imitate, the way a VLIW processor would be tasked, to perform, operations related to arrays. We can see this instruction, right below in the image 5.4.

```
Instruction: 2

Operation:
1: addi 00000000001(1) 000100(4 rs) 000(addi)  010000(16 rd)  000011(I-type)   A{}
2: addi 00000000010(2) 000100(4 rs) 000  010001(17 rd)  000011(addi)
3: addi 00000000011(3) 000100(4 rs) 000  010010(18 rd)  000011(addi)
4: addi 00000000001(1) 000101(5 rs) 000(addi)  010011(19 rd)  000011(I-type)  B{}
5: addi 00000000010(2) 000101(5 rs) 000  010100(20 rd)  000011(addi)
6: addi 00000000011(3) 000101(5 rs) 000  010101(21 rd)  000011(addi)
7: addi%Mark    00000000010(2) 000110(6 rs) 000(addi)  010111(23 rd)  000011(I-type) C{}
Dependencies:
8:         0000001(Marks) 0[Integer]00[mainUnit] 000 000 000 000 000 000(Units) 0000(Undefined)


Instruction: 3

Operation:
1: store 00000(Imm) 010101(rs2 == 21) 000000(rs1 == 0) 011(single-precision) 010000(Immediate == 16) 111101(store)
2: store 00000(Imm) 010110(rs2 == 22) 000000(rs1 == 0) 011(single-precision) 010001(Immediate == 17) 111101(store)
3: store 00000(Imm) 010111(rs2 == 23) 000000(rs1 == 0) 011(single-precision) 010010(Immediate == 18) 111101(store)
4: no-op 00000000000    000000    000  000000     000000
5: store 00000(Imm) 000100(rs2 == 4) 000000(rs1 == 0) 011(single-precision) 100000(Immediate == 32) 111101(store)
6: no-op 00000000000    000000    000  000000     000000
7: no-op 00000000000    000000    000  000000     000000
Dependencies:
8:         0011111(Marks) 0[Integer]00[mainUnit] 000 000 000 000 000 000(Units) 0000(Undefined)
```

Image 5.4

This example could continue further. We could then have a follow-up, 256-bit instruction, with operations taking the results from the 2nd instruction's operations, and re-using them to load the arrays "A[]", and "B[]"(from the system's memory), in order to add their contents together, and finally, having them stored back to the memory address given to array "C[]". However, instead of this, we will proceed with the third instruction(which we can see within image 5.4).

In this scenario, we have the contents originally loaded from the memory(at the very beginning of the program), being manipulated by the processor, and finally being stored in different memory cells. The effects of the 3rd instruction on it's operands, will be the following:

- Mem[16] <= reg[21]
- Mem[17] <= reg[22]
- Mem[18] <= reg[23]

This, "third" instruction, will actually be the fourth instruction issued from the binary program. The reason for this is that, because of operation latency, we wouldn't have the results of the second instruction's operations available within the registers 21 to 23 before a number of cycles had "passed". We will assume, for this example, that the real third 256-bit instruction's operations, do not affect this example's result in any way.

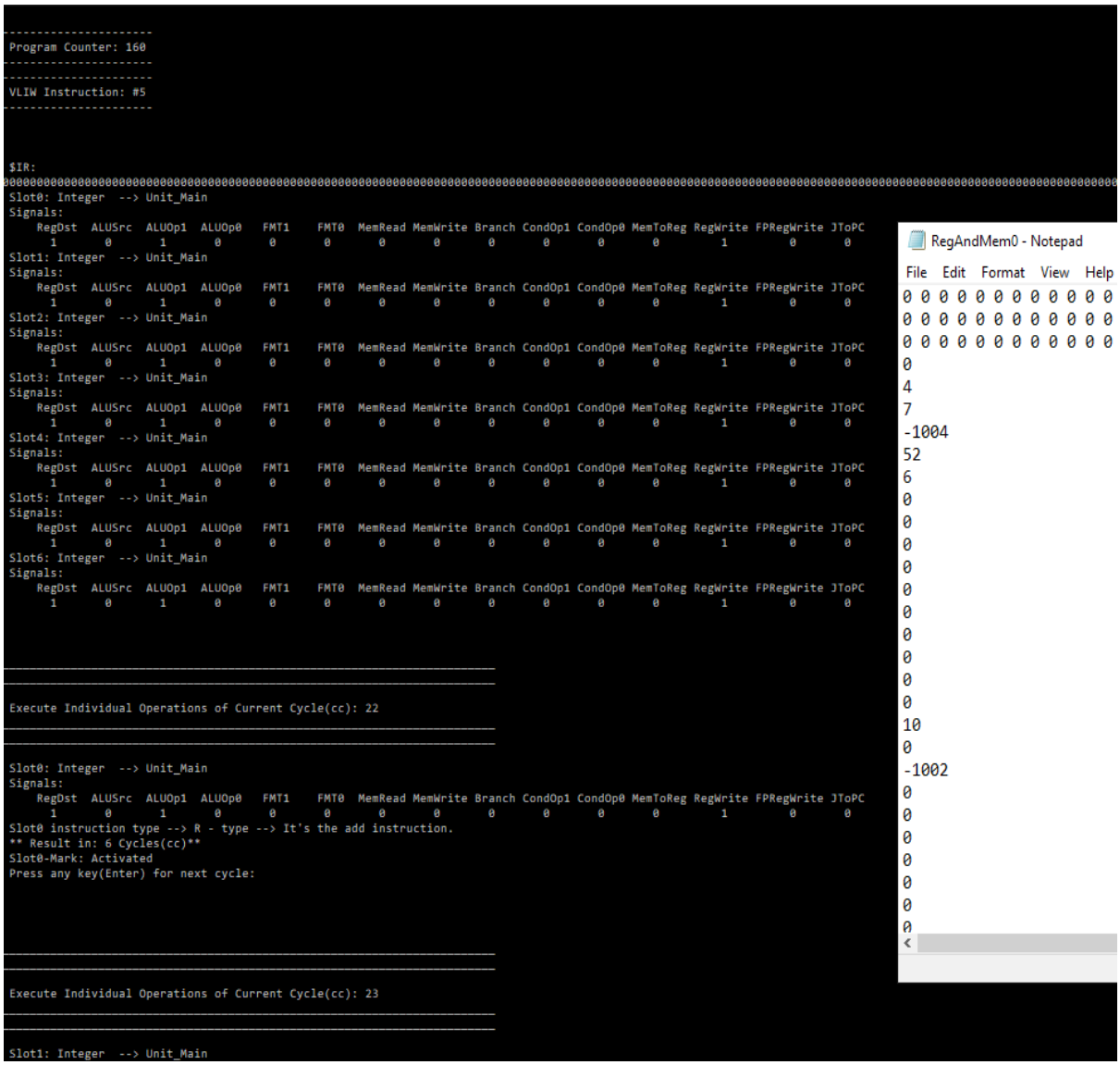

Image 5.5

As we can see in image 5.5 above, the results, of the program presented, will be visible within a few clock cycles(after the operations have been issued).

(Concluding the Chapter 5.10)

After setting the parameters of our simulation, we were able to see our, very own, new Very Long Instruction Word Architecture in action. We were also able to provide useful guidelines, which could help someone else, in using this simulation effectively. With the guidelines provided, one should have a good enough understanding of the code, to even expand the simulation further.

Simulating an architecture, proved itself an interesting endeavor to embark in. Even more so, if the said architecture, is one belonging to a VLIW computer. However, one must be ready to start from scratch, should the need arise, because as with designing the architecture, simulating it has it's own pitfalls, to be weary of.

Chapter 6 Conclusions

In conclusion, VLIW architectures seem to perform the best in scientific applications, as well as in other applications where the parameters of the problem to be "solved" are well known from the very beginning(or guaranteed to have a great amount of operations initiated, as in the case of graphical processing units). These favourable parameters, allow for the static initiation of the most operations possible, within a single clock cycle.

The usage of dynamic components, to enhance areas of otherwise natural weaknesses in the VLIW paradigm, is also highly suggested. The usage of predicated operations can be highly beneficial, especially as with the passage of time the available memory grows greater and greater per year(which has always been a limiter to predicated operations, in embedded systems).

It also seems wise, to have the VLIW architecture and implementation, as simple as possible in it's design. Designs with higher complexity, even when handled by dynamic components which don't require the programmer's input, have a history of failing commercially, and of being considered unreasonably complex to build a compiler for. Simpler designs on the other hand, even when not financially successful, have been praised for their effort in making VLIW architectures work.

Finally, being aware of historical examples in the field, of designing instruction set architectures(VLIW or otherwise), can help you avoid hurdles that other designers have met with in the past. Many ideas sound great in theory, but end up being an irreversible limitation to the architecture's design.

Bibliography

- Computer Architecture a Quantitative Approach, the 6th American Edition(In Greek).
  Authorship: John L. Hennessy, and David A. Patterson.
- Computer Organization and Design, the 4th edition(In Greek).
  Authorship: David A. Patterson and John L. Hennessy.
- Computer Organization and Architecture(In Greek).
  Authorship: Carl Hamacher, Zvonko Vranesic, Safwat Zaky.
- Operating System Concepts, the 9th Edition(In Greek).
  Authorship: Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne.
- Compilers: Principles, Techniques, and Tools, the 2nd Edition(In Greek).
  Authorship: Alfred V. Aho, Jeffrey D. Ullman, Ravi Sethi, and Monica S. Lam.
- Multiflow Computer: A Start-up Odyssey
  Authorship: Elizabeth Fisher
- Very Long Instruction Word architectures and the ELI-512
  Authorship: Joseph A. Fischer
- Trace Scheduling: A Technique for Global Microcode Compaction
  Authorship: Joseph A. Fischer
- Fujitsu: FR-V single-chip multicore processor: FR1000.
  Origin:
  https://www.fujitsu.com/global/documents/about/resources/publications/fstj/archives/vol42-2/paper03.pdf
- The First Million-Transistor Chip: the Engineers' Story.
  https://spectrum.ieee.org/intel-860
- i860 64-bit Microprocessor Programmer's Reference Manual.
  Published: Intel Corporation
- SHARC Processor Architectural Overview
  Origin: https://www.analog.com/en/lp/001/sharc-processor-architectural-overview.html
- SHARC Processors: Manuals
  Origin: https://www.analog.com/en/lp/001/sharc-manuals.html
- SHARC Processor Programming Reference
  Origin:
  https://www.analog.com/media/en/dsp-documentation/processor-manuals/adsp-2136x_2137x_214xx_pgr_rev2.4.pdf
- Index of SHARC Products
  Origin: www.analog.com/en/processors-dsp/sharc/products/index.html

- ST240 core and instruction set architecture Reference manual.
  Origin: https://www.st.com
- PM0215 Programming Manual(STM32F0xxx Cortex-M0 programming manual).
  Origin: https://www.st.com
- https://en.wikipedia.org/wiki/Josh_Fisher
  Origin: https://en.wikipedia.org/wiki/Main_Page, hosted by wikimedia.
- https://en.wikipedia.org/wiki/Multiflow
  Origin: https://en.wikipedia.org/wiki/Main_Page, hosted by wikimedia.
- https://en.wikipedia.org/wiki/Very_long_instruction_word
  Origin: https://en.wikipedia.org/wiki/Main_Page, hosted by wikimedia.