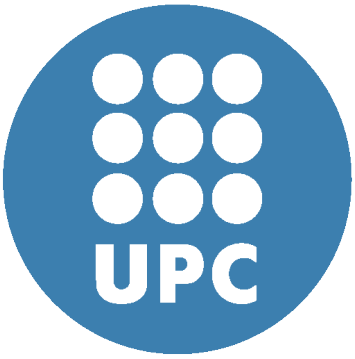
Design and implementation of a Multimedia Extension for a RISC Processor



Eduardo Jonathan Martínez Montes

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya - BarcelonaTech

A thesis submitted for the degree of

Master in Innovation and Research in Informatics (MIRI-HPC)

2 July 2015

Advisor: Oscar Palomar, CAPP Junior Researcher at BSC-CNS
Advisor: Marco Antonio Ramírez Salinas, Researcher at CIC-IPN
Tutor: Adrian Cristal Kestelman, CAPP Senior Researcher Professor at BSC-CNS

To my parents and my sister for their support during all this years

To Oscar, my advisor, for his patience and his support.

Acknowledgements

I want to thank to all people and friends that helping me to do this possible. I also want to thank to my colleagues at MICROSE Lab and its leader Marco Ramírez. Also, it would not have been possible without the support of the BSC and its leader Mateo Valero.

This work has been supported by the CONACYT and its project SIP: 20150957 "Desarrollo de Procesadores de Alto Desempeño para Sistemas en Chips Programables"

Special thanks

Adrián Cristal Kestelman

Cesar Hernández Calderón

Luis Villa Vargas

Osman Sabri Ünsal

Patricia Cantú Velasco

Ramón Canal Corretger

Santiago Reyes

Ulises Cortes

Víctor Hugo Ponce Ponce

Abstract

Nowadays, multimedia content is everywhere. Many applications use audio, images, and video. Over the last 20 years significant advances have been made in compilation and microarchitecture technologies to improve the instruction-level parallelism (ILP). As more demanding applications have appeared, a number of different microarchitectures have emerged to deal with them. A number of microarchitectural techniques have emerged to deal with them: pipelining, superscalar, out-of-order, multithreading, etc.

All these techniques increase the ILP at core level. An alternative to increase the performance is to exploit data level parallelism. This technique improves performance in applications where highly repetitive operations need to be performed. It performs the same operation on multiple pieces of data. The first approach was vector processors and most modern implementations use short fixed size vectors in what is known as Single Instruction Multiple Data (SIMD) extensions.

Present and next generation of mobile and embedded devices require having multimedia support to be competitive. The most popular technique at the microarchitecture level to deal with these requirements is to use SIMD extensions to the Instruction Set Architecture. It improves performance by processing vector operations in parallel. Their high compute power and hardware simplicity improve overall performance in an energy efficient manner. Furthermore, their replicated functional units and simple control mechanisms make them manageable to scaling to higher vector lengths.

MIPS born as an academic research and also it has been the favorite architecture used to teach Computer Architecture Design. Other architectures as x86 or ARM are quite complex to understand for university students.

Most popular microarchitectures today have at least a SIMD unit implementation, x86-64 (MMX, SSE, AVX), PowerPC (AltiVec), ARM (Neon), MIPS (MDMX, MSA), and so on. This work focuses on the MIPS microarchitecture because it follows the RISC (Reduced Instruction Set Computer) philosophy quite closely, which is desirable since it simplifies implementation and it is easy to understand by students.

First SIMD implementation for MIPS was MIPS Digital Media Extension (MDMX) that supported video, audio and graphics pixel processing by introducing two vectors formats of small integers. These vectors have a width of 64-bits, in the form of signed 16-bit or 8 unsigned 8-bit integers. MDMX is quite old, and it never reached production. Latest SIMD implementation on MIPS appeared in 2014 as an add-on of MIPS32/64 Release 5. It is called MIPS SIMD Architecture (MSA). It is designed to support 128-bit vectors of 8, 16, 32 and 64-bit integer vectors; 16 and 32-bit fixed-point, or 32 and 64-bit floating-point elements.

We have decided to implement the MSA ISA on an FPGA together with a MIPS-like implementation (soft-core). Since it is not intended for general purpose computing we take

a MIPS32 core from opencores.org, upgraded it with the needed MIPS Release 6 instructions, microarchitecture and control to use MSA coprocessor. We have created a MSA coprocessor, a testing bench and adapted some embedded micro benchmarks to test the MSA coprocessor.

Contents

List of Figures iv
List of Tables viii
1 Introduction 1
1.1 Motivation 1
1.2 Context of the project 4
1.3 Objectives 5
2 State of the Art 6
2.1 Origin 6
2.2 Parallelism key of performance 6
2.3 Vector Architectures 8
2.4 SIMD Multimedia extension 8
2.5 Graphics Processing Units 10
2.6 SIMD History 11
3 MIPS Architecture 13
3.1 Brief History of MIPS Company 13
3.2 History of the MIPS ISA 13
3.3 Current MIPS ISA 14
3.4 Optional Components 16
3.5 Brief description of the optional components available on Release 6 17
4 The MIPS® SIMD Architecture module 19
4.1 Instruction Decoding and Formats 20
4.2 GCC support 23
5 Tools and resources 26
5.1 Verilog 26
5.2 Quartus II 26
5.3 DE2-115 board 26
6 Basic components 27
6.1 Adder 27
6.1.1 Ripple Carry Adder 28

6.1.2 Carry Look-ahead Adder 29
6.1.3 Kogge-Stone Adder 30
6.1.4 Adder evaluation 31
6.2 Multiplier 32
6.3 Divider 34
6.4 Multiplexer 35
6.5 Saturated Arithmetic 36
7 The MIPS SIMD Architecture Instruction Set 38
7.1.1 Data Transfer 38
7.1.2 Arithmetic 40
7.1.3 Comparison 42
7.1.4 Logical and Shift 43
7.1.5 Insertion and Extraction 45
8 Architecture Implementation 45
8.1 Overview 45
8.2 MIPS32 core 46
8.3 Fetch 48
8.4 Decode 50
8.5 Register File 50
9 SIMD execution stage 53
9.1 Vector Processing Unit 53
9.2 Multipurpose adder lane 55
9.3 Multiplier lane 56
9.4 Divider circuit 57
9.5 Special unit 1 61
9.6 Special unit 2 61
9.7 Special unit 3 62
10 Memory 63
10.1 Instruction Memory 64
10.2 Data Memory 65
11 Software Tools 69
11.1 GCC 69
11.2 QEMU 71

11.3 ModelSim 71
12 Evaluation 75
12.1 FPGA utilization 75
12.2 Benchmarks 75
12.2.1 FDCT 76
12.2.2 Matmult 76
12.3 Summary 78
13 Future work 79
14 Conclusions 80
15 References 81
16 Annexes 85
16.1 Joining results from 3R lanes 85
16.2 Detail Implementation of Special 1 unit 86
16.3 Detail implementation of Special 2 unit 89
16.4 Detail implementation of Special 3 unit 91
16.5 VSHF unit 93
16.6 SRLR unit 98
16.7 SRAR unit 103
16.8 SLD unit 108
16.9 SPLAT unit 115
16.10 PCKEV unit 116
16.11 PCKOD unit 117
16.12 ILVL unit 118
16.13 ILVR unit 119
16.14 ILVEV unit 120
16.15 ILVOD unit 121
16.16 Insert unit 122
16.17 INSVE unit 123
16.18 Dot Product unit 124
16.19 Population count unit 128
16.20 Leading Ones/Zeros unit 131
16.21 Vector Operations unit 132
16.22 SHF unit 134

16.23 SAT unit 136
16.24 CEQ unit 139
16.25 CLT unit 141
16.26 CLE unit 142
16.27 MAX unit 144
16.28 MIN unit 146
16.29 MAX MN unit 148
16.30 SLL unit 151
16.31 SRA unit 153
16.32 SRL unit 155
16.33 BIT unit 157
16.34 BINSL unit 159
16.35 BINSR unit 160
16.36 Join results 161
16.37 Branch unit 162

List of Figures

Figure 1: 35 years of microprocessor trend data (6) 3
Figure 2: Example of a SIMD parallel addition 9
Figure 3: Dynamic Power 10
Figure 4: MIPS32/64 Releases and optional modules (39) 16
Figure 5: Optional components supported by MIPS32 and MIP64 (40) 17
Figure 6: MSA formats 20
Figure 7: Data format and element index field encoding 23
Figure 8: Data format and bit index field encoding 23
Figure 9: GCC compilation using MSA 24
Figure 10: Linker script example. 24
Figure 11: Example for addition of two integer arrays 25
Figure 12: One-bit full adder implementation 27
Figure 13: One-bit full adder block 28
Figure 14: 4-bit ripple carry adder 29
Figure 15: 4-bit carry look-ahead adder 30
Figure 16: Boolean equations used in 4-bit carry look-ahead adder 30
Figure 17: 4-bit Kogge-Stone Adder 31
Figure 18: Carry operator 31

Figure 19: Boolean equations involved in Kogge-Stone adder 31
Figure 20: formula of Karatsuba algorithm 32
Figure 21: The circuit to perform Karatsuba multiplication 33
Figure 22: The overlap circuit for the 8-bit Karatsuba multiplier 33
Figure 23: Numerical example of Karatsuba multiplier using base 16 34
Figure 24: Alternative division algorithm 35
Figure 25: Multiplexer 4-to 1 of 128-bit width 35
Figure 26: Boolean equation for a 4-to 1 multiplexer 35
Figure 27: 8-bit signed integer subtraction using wraparound and saturated arithmetic 37
Figure 28: 8-bit unsigned integer addition using wraparound and saturated arithmetic 37
Figure 29: Interconnection between MIPS32 Core and MSA unit (control signals omitted) 40
Figure 30: Modifications made to the main core to attach SIMD unit 47
Figure 31: Interconnection between core and coprocessor 49
Figure 32: Signals generated by SIMD decode 50
Figure 33: Representation of the MSA register file 51
Figure 34: Implementation of the MSA register file 52
Figure 35: SIMD execution stage 53
Figure 36: Four lanes (left) vs two lanes (right) 54
Figure 37: Multipurpose adder lane 55
Figure 38: Multiplication circuit and fused multiplication 57
Figure 39: Divider circuit 58
Figure 40: 3R Lane, multiplier, divider, adder and substrate multipurpose 60
Figure 41: Special unit 1 61
Figure 42: Special unit 2 62
Figure 43: Special unit 3 62
Figure 44: Instruction and data memory 64
Figure 45: Example of row and memory cell calculation 65
Figure 46: Multiplexors used to rearrange the data to load and store it into the data memory 66
Figure 47: Logical circuit that creates the 16 write enable signals and multiplexor used to rearrange the write enable mask. 67
Figure 48: Calculation of rows for each memory cell 68
Figure 49: Compiler flags to enable SIMD 69
Figure 50: File script.ld 69
Figure 51: File startup.S 70
Figure 52: Makefile 70
Figure 53: Fragment of a decompile file using CodeBench 71
Figure 54: Example of waveforms generated by ModelSim 72
Figure 55: Memory visualization in ModelSim 72
Figure 56: Example of instrumentation code 73
Figure 57: A segment of a trace execution from the VPU 74
Figure 58: FDCT Benchmark results 76
Figure 59: Matmult Benchmark results 77
Figure 60: Superscalar processor 79
Figure 61: Selector of vector result format 85

Figure 62: Implementation of special unit 1 86
Figure 63: Implementation of special unit 1 (cont.) 87
Figure 64: Joining result of special unit 1 88
Figure 65: Implementation of the special unit 2 89
Figure 66: Result of the special unit 2 90
Figure 67: Implementation of special unit 3 91
Figure 68: Result of special unit 3 92
Figure 69: VSHF unit 93
Figure 70: Implementation of VSHF unit Byte format 94
Figure 71: Implementation of the VSHF unit Halfword format 95
Figure 72: Implementation of the VSHF unit Word format 96
Figure 73: Implementation of the VSHF unit Doubleword format 97
Figure 74: SRLR unit 98
Figure 75: Implementation of the SRLR unit byte format 99
Figure 76: Implementation of the SRLR unit halfword format 100
Figure 77: Implementation of the SRLR unit word format 101
Figure 78: Implementation of the SRLR unit doubleword format 102
Figure 79: SRAR unit 103
Figure 80: Implementation of the SRAR unit byte format 104
Figure 81: Implementation of the SRAR unit halfword format 105
Figure 82: Implementation of SRAR unit word format 106
Figure 83: Implementation of the SRAR unit doubleword format 107
Figure 84: Implementation of SLD unit, input signal generation and “n” signal generation 109
Figure 85: Implementation of the SLD unit byte format 110
Figure 86: Implementation of the SLD unit byte format (cont.) 111
Figure 87: Implementation of the SLD unit halfword format 112
Figure 88: Implementation of the SLD unit word format 113
Figure 89: Implementation of the SLD unit doubleword format 114
Figure 90: SPLAT unit implementation 115
Figure 91: PCKEV implementation 116
Figure 92: PCKOD implementation 117
Figure 93: ILVL unit implementation 118
Figure 94: ILVR unit implementation 119
Figure 95: ILVEV unit implementation 120
Figure 96: Implementation of ILVOD unit 121
Figure 97: Implementation of Insert unit 122
Figure 98: Implementation of Insve unit 123
Figure 99: Implementation of Dotproduct unit for halword format 124
Figure 100: Implementation of Dotproduct for word format 125
Figure 101: Implementation of Dotproduct for doubleword format 126
Figure 102: Selection of results from Dotproduct 127
Figure 103: Population counter for a byte. 128
Figure 104: Population counter for a halfword 129
Figure 105: Population counter unit 130

Figure 106: Leading byte counting 131
Figure 107: Leading counting unit 132
Figure 108: Vector operations 133
Figure 109: SHF implementation 136
Figure 110: SAT unit implementation for byte format 136
Figure 111: SAT implementation for halfword format 137
Figure 112: SAT implementation for word format 137
Figure 113: SAT implementation for Doubleword format 138
Figure 114: Implementation of CEQ unit 140
Figure 115: Implementation of CLT unit 142
Figure 116: Implementation of CLE unit 143
Figure 117: Implementation of MAX unit 145
Figure 118: Implementation of MIN unit 147
Figure 119: Implementation of MAX MIN unit 148
Figure 120: Implementation of MX MIN unit (cont.) 149
Figure 121: MAX MIN unit implementation (cont.) 150
Figure 122: Implementation of SLL unit 152
Figure 123: Implementation of SRA unit 154
Figure 124: Implementation of SRL unit 156
Figure 125: Implementation of BIT unit 157
Figure 126: Implementation of BIT unit (cont.) 158
Figure 127: Implementation of BINSL unit 159
Figure 128: Implementation of BINSR unit 160
Figure 129: Result format selection 161
Figure 130: Layout of branch instructions 162
Figure 131: Branch detection of format Doubleword 163
Figure 132: Branch detection of format Word 163
Figure 133: Branch detection of format Halfword 164
Figure 134: Branch detection of format Byte 164
Figure 135: Format selector of branches 165

List of Tables

Table 1: Summary of SIMD implementations on PC 12
Table 2: Supported formats by MSA 19
Table 3: Decode of instructions 21
Table 4: Adder performance evaluation 32
Table 5: Range Limits for Saturated Arithmetic 37
Table 6: Data transfer instructions in MSA 39
Table 7: Arithmetic instructions in MSA 42
Table 8: Comparison instructions in MSA 43
Table 9: Logical and shift instructions in MSA 44
Table 10: Insertion and extraction instructions in MSA 45
Table 11: Instructions introduced to MIPS32 core to support SIMD unit 47
Table 12: Instructions that uses multipurpose adder lane. 56
Table 13: Instructions that use Multiplication circuit 57
Table 14: Instructions that uses divided circuit 59
Table 15: Resources of the implementation 75
Table 16: FDCT Benchmark results, total speedup of using SIMD is 4.05x 76
Table 17: Matmult Benchmark results, total speedup of using SIMD is 2.68x 77
Table 18: Vector instructions 133
Table 19: SIMD branch instructions 162

1 Introduction

1.1 Motivation

One of the main objectives of computer architecture is to keep improving performance. Many different microarchitectural techniques have appeared to try to extract and exploit parallelism from sequential applications at run time. From the beginning of the age of circuit integration the number in each generation of transistors available per die has doubled approximately every two years (1). It means that computer architectures have more resources available on the design for the new microarchitectural techniques.

The first approach to extract parallelism was trying to execute more than one instruction at the same time. This concept is call instruction-level parallelism (ILP). ILP started by overlapping execution of instructions in time-sliced fashion (pipeline). It allows to have ideally as many instructions in execution as total pipeline stages. Implementation examples are Intel 80486 and MIPS R2000.

Next approach was focused on executing multiple instruction at the same time. It was achieved by incrementing the number of functional units and dispatching multiple instructions per cycle. There are two techniques based on this concept. Superscalar and Very Long Instruction Word (VLIW). The main difference between them is the complexity of their control unit. Superscalar decides at run time which instructions are dispatched together meanwhile VLIW requires explicitly encoded dispatch, controlled by the programmer. Early, superscalar implementation examples are Intel i960 and Motorola MC88100. Early VLIW examples are Intel i860 and Elbrus 2000.

Important academic research in the 80s on restricted form of data flow (2), and other techniques like register renaming (that breaks false dependencies) and introducing a dynamic executing scheduler unit, facilitated out-of-order (OoO) execution. Main benefit of OoO comes from the possibility of avoiding idle processor due to instructions that are waiting for source operands to be calculated by older instructions. On the late 90s out-of-order (OoO) microarchitectures became popular due to the huge performance improvement they provide. Examples of OoO designs are Intel Pentium Pro, MIPS R1000 and DEC Alpha 21264.

Pipelining, superscalar and OoO techniques do not require any programmer intervention to run. Nevertheless, they improve performance when they are applied. Another example of techniques that are hidden from the programmer or not expose to the Instruction Set Architecture (ISA) are branch prediction and cache hierarchy. This characteristic allows backward and forward application compatibility. On the other hand, VLIW exposes many

microarchitectural features to the ISA than makes almost impossible (without recompiling) to use applications from one generation to another one.

One important challenge of OoO is finding enough parallelism. Performance results and trends are expressed in terms of issue width and windows size. Increasing the size of the instruction window is an straightforward solution to find parallelism at the instruction stream. A larger window is required for finding more independent instructions to take advantage of wider issue. The issue window logic is one of the primary contributors of complexity in typical out-of-order microarchitectures. The complexity and size of the structures of the microarchitecture necessary to implement OoO execution grows quadratically respect to the size of the issue window. Furthermore, each of the components shows a linear increase respect to the issue width (3).

Designers decided to allow the execution of more than one thread (typically two) to increase parallelism. Instructions from different threads are independent by definition. This is called Multi-threading. It requires minimal changes at the architectural level to be implemented but the programmer must provide at least two threads to exploit this feature. Examples of this design are Intel Pentium IV HT and IBM Power5.

At mid-2000, maximum single thread performance was achieved. Energy consumption and heat dissipation became very hard to manage because of power density (4). This is known as the power wall. Figure 1 shows several trends associated with Moore's law evolution. However the number of transistors is still increasing. Designers started to put more than one core in the same die and Multicore processors appeared. One of the main benefits is that multicore allow higher performance at lower energy than a single core processors with the same performance. An important drawback of multicores is programmability. It is very difficult to create applications that can keep all cores busy. This problem is described by Amdahl's Law (5). Examples of multicore architecture are AMD Athlon 64, Intel Core Duo and IBM Power5.

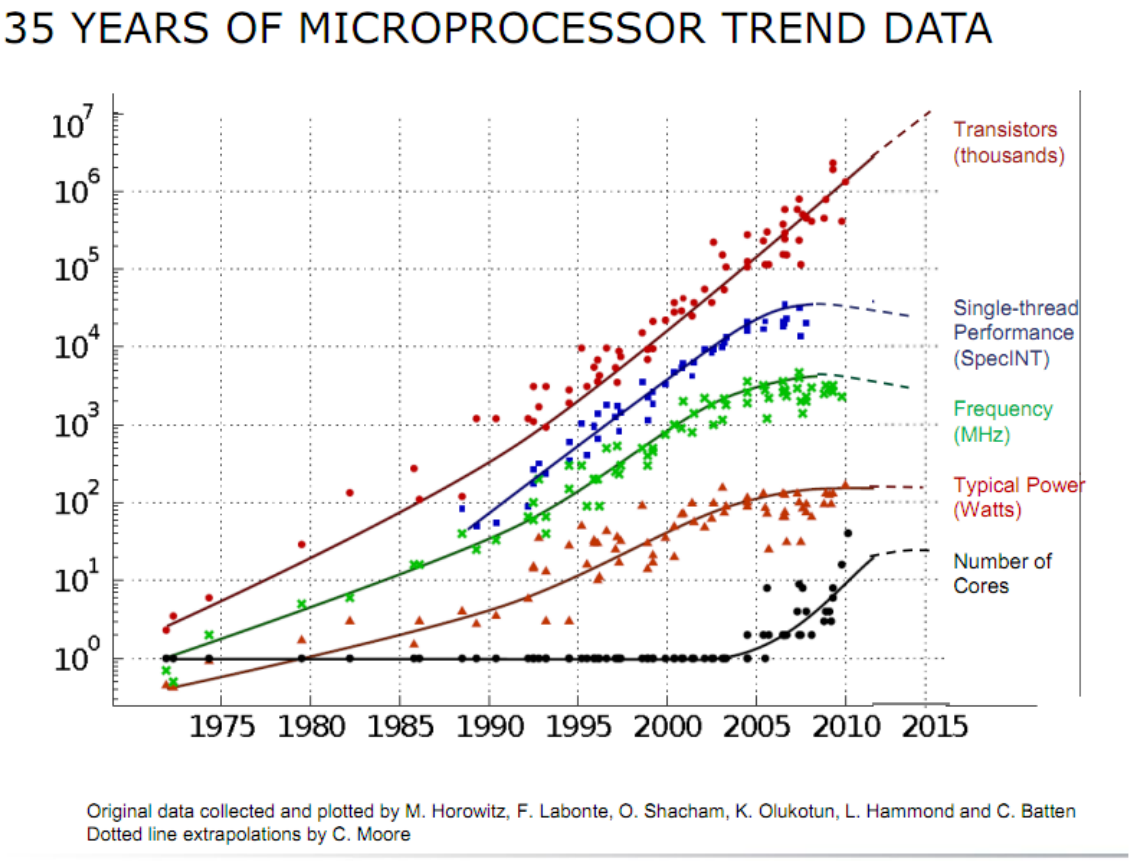


Figure 1: 35 years of microprocessor trend data (6)

The number of cores has not stopped to increase. Nowadays, there are processors that have a huge number of cores. They are called many-cores and examples of this design are Calvium's Thunder X (48 cores) (7) and Intel Xeon Phi (61 cores) (8).

Pipeline, superscalar, and OoO microarchitectures techniques are focused on exploiting Instruction Level Parallelism (ILP). Another different approach to exploit parallelism is called Data-Level Parallelism (DLP). The idea behind DLP is using a single instruction to launch many data operations. Examples of vector architectures techniques that exploit DLP are vector architectures and what is known as Single Instruction Multiple Data (SIMD) extensions.

As multimedia applications became popular, data to be processed increased and real-time processing graphics were needed as well. Computer architectures incorporated special hardware units to deal with these new multimedia issues. The designers introduced to personal computers (9) SIMD extensions to the ISA, an alternative to vector architectures for exploiting DLP. The main difference between them is that vector architectures process the elements of the vector operands in a pipelined fashion, one or a few per cycle, while SIMD instructions process all the elements of the operands at once. Moreover, vector

architecture supports a variable number of data operands per operation meanwhile SIMD has a fixed number of data operands.

The integration of SIMD units on the processors has exposed a new challenge. The programmer (compiler) has to be capable to detect DLP and generate SIMD operations. Code optimizations play an important role in performance.

Early SIMD implementations could perform integer instructions and elements were small (8-bit and 16-bit). SIMD extensions have been increasing its performance and functionality from generation to generation. SIMD width vector has increased from 64-bit to 512-bit. SIMD floating-point data types, single and double precision appeared (10).

Nowadays, all processors have a multimedia extension (or SIMD unit), and since it can perform floating-point operations computer architectures avoid the implementation of FPU and these operations are performed by the SIMD unit as special case in which the data is considered as a one-dimensional vector. Additionally, it has to be considered that the x86-64 ISA has more SIMD instructions than general-purpose instructions (11).

1.2 Context of the project

Most popular architectures[1] currently have at least a SIMD unit implementation, x86-64 (MMX, SSE, AVX), PowerPC (AltiVec), ARM (Neon), MIPS (MDMX, MSA), and so on. MIPS was born as an academic research and it is used to teach Computer Architecture Design. Other architectures as x86 or ARM are quite complex to understand for university students and also, their internal architecture has not been disclosed, thus many important implementation details are unknown.

MIPS architecture follows RISC (Reduced Instruction Set Computer) philosophy. RISC means regular ISA, as well. A regular ISA allows to have simple instruction decode. Since no-complex instructions are used by RISC, the microarchitecture[2] implementation becomes simple by avoiding to use complex circuits to solve complex instructions or splitting complex instructions into microcode. MIPS original design was published in "Computer Architecture: A Quantitative Approach". Because of that, there are many MIPS-like implementations done by academic circles. This thesis is focused on the MIPS architecture because it follows the RISC philosophy to a greater extent than others architectures.

The last SIMD implementation is called *MIPS SIMD Architecture* (MSA) and was launched the last year on April. At the moment the present work started there was not any public

[1] Architecture describes the capabilities and programming model but not a particular implementation, sometimes refers as the ISA.
[2] Microarchitecture is the way a given instruction set architecture (ISA) is implemented on a processor.

soft-core implementation. HPCA (12) decided to implement *MIPS SIMD Architecture* on a FPGA.

These statements were stablished because this SIMD unit is supposed to be part of a bigger project at IP-CIC is composed of many smaller projects like this (13). The idea is provide a System on Chip (SoC) designed for education. It should be used to help in processor architecture lectures, develop university projects, etc.

This project is based on an ISA compatible with MIPS32/64 Release 6. It is developed using Verilog which is a popular Hardware Description Language (HDL).

To validate the implementation we used a soft-core MIP32 implementation from opencores.org and some micro benchmarks for embedded architectures compiled with GCC.

1.3 Objectives

The main goal of this project is to create a SIMD unit based on three statements:

- To provide a SIMD unit as a MIPS coprocessor.
- To implement it as a soft-core optimize it for FPGAs.
- To have clean code and well documented, in order to be easy to understand and modify.

Secondary objectives are:

- To provide a test bench tool to test and debug.
- To provide some benchmarks to validate executions and measure performance.

2 State of the Art

2.1 Origin

From the beginning of microprocessors computer architects and designers have been fighting to keep improving performance from generation to generation. This has been possible by doubling the number of transistors approximately every two years. This continuous improvement is called Moore´s Law (1). The typical increase in transistor density enables designers to introduce new microarchitectural techniques to achieve higher performance levels. An example of this continuous improvement is the Tick-Tock model from Intel (14).

2.2 Parallelism key of performance

Nowadays, parallelism is used at multiple levels from the microarchitecture to the nodes, with energy and cost being the primary constrains. Parallelism can be classified in three types (15):

- Instruction-Level Parallelism (ILP), is focused on executing more than one instruction in parallel or overlapping instruction execution. Some microarchitectural techniques that exploit ILP are pipelining, superscalar and out-of-order. A modest level of data-level parallelism is achieved.
- Data-Level Parallelism (DLP) focuses on operating multiple data items at the same time. Some microarchitectural techniques that exploit DLP are Vector Architectures, SIMD extensions and Graphic Processor Units (GPUs).
- Thread-Level Parallelism (TLP) focuses on operating tasks of work (like threads[3]) in parallel that are independent. Some microarchitectural techniques that exploit TLP are hyper-threading, multicore and many-core.

ILP exploits implicit parallel operations within a loop or straight-line code segment. TLP is focused on splitting program into independent tasks. Philosophies like divide-and conquer are used. TLP explicitly represented by the use of multiple threads of execution that are inherently parallel. TLP could be more cost-effective to exploit than ILP. DLP energy cost grows linear with respect to ILP. GPUs have a large energy efficiency advantage with respect to ILP or DLP. GPUs are designed to exploit high levels of data and thread level parallelism for performance rather than extracting ILP from a small number of threads (16).

[3] Thread: process with own instructions, data and PC. It may be a subpart of a parallel program or it may be and independent program.

These previous schemes for hardware to provide the support to data-level parallelism and task-level parallelism are not at all new. Michael Flynn classified all computers in four groups (17):

- Single instruction stream, single data stream (SISD). This classification is the classic uniprocessor. From the point of view of the programmer, instructions are executed sequentially. Although ILP can be exploited, for example with superscalar and speculative execution.
- Single instruction stream, multiple data streams (SIMD). This classification describes computers that execute the same instruction on multiple processors simultaneously (but not necessarily concurrently) using different data streams. SIMD exploits DLP by applying the same operation to multiple items of data in parallel. There are three different architectures that exploit DLP: vector architectures, multimedia extensions and GPUs[4].
- Multiple instructions streams, single data stream (MISD). This is rarely used. Some computers implement it for reliability. Systems that require high fault tolerance like aircraft use heterogeneous systems that operate on the same data stream and results must match or alternately, the value result is chosen by a "two of three" vote. If all copies of the output are not identical, then an error has occurred (18).
- Multiple instruction streams, multiple data streams (MIMD). This classification describes multicore computers. Each general processor (or core if they are in the same die) fetches its own instructions and operates on its own data. MIMD exploits mainly TLP. Examples are distributed systems and multicore processors.

[4] Although GPUs like to call their model Single Instruction, Multiple Threads (SIMT) for Multiple Thread.

2.3 Vector Architectures

Vector architectures take streams of data operands from memory (load), place them into large, register files, operate on them sequentially in those register files, and then send the result streams back to memory (store). It is done pretty much in the RISC style. Also, vector architectures implement a pipeline that stretched from memory, through the processor, and back to memory is very long and takes many clock cycles to fill, but once it is filled, the throughput was tremendous. Vector-length is variable and it is set using a vector-length register (VLR)[5].

The first two vector supercomputers appeared in the early 1970s. One was the Texas Instrument-ASC and the other was the STAR-100 developed by Control Data Corporation (CDC) (19). With the introduction of the CRAY-1 in 1976 vector supercomputing became successful. CRAY-1 was followed by the CRAY-2 and then by CRAY X-MP. The STAR-100 was followed by the Cyber 200 series and then ETA-10.

Vector processors can greatly improve performance on certain workloads, notably numerical simulations and similar task. Vector processor do not benefit from executing scalar operations. Bandwidth and probably register space could be wasted. Nevertheless, performance should be similar to a scalar processor[6].

In the early 90s by the continuous increasing of the number of transistors that could be fit in a single die. Improvements in CMOS VLSI technology that allowed to break the 100MHz barrier. Microprocessors like DEC Alpha surpassed the cycle times of the fastest supercomputers of that age (19). The introduction of fast microprocessors substantially changed the supercomputer market. Due to their much higher volumes, microprocessors offer very low prices per processor.

The idea of building supercomputers using many of these processors spread swiftly as a consequence of their cheaper cost and powerful. Nowadays supercomputers are made mainly using commodity parts (20).

2.4 SIMD Multimedia extension

Multimedia applications uses narrow data-types, typical widths of data are 8-bit and 16-bit, for instance, for graphic representation of each of the three primary colors plus 8 bits for transparency; or audio samples are also typically represented with 8 or 16 bits. SIMD extensions allow performing the same operation on a group of elements in parallel.

Both, SIMD extensions and Vector processors operate using vectors. Vector processors can deal with vectors of variable size and SIMD extensions operates always in vectors of fixed

[5] The exception is the STAR-100, which operated directly from memory. It did not have a vector register file.
[6] CRAY-I (1976) was capable of much higher scalar performance than any of its contemporaries (21).

size, it is usually called packed operand. The main difference between them is the execution model, SIMD extensions execute the whole packed operand (vector) in parallel (or mostly in parallel)[7] whereas vector processors uses a pipe line execution that process element by element (or a few of them) in a RISC style .

This difference in philosophy stems from the fact that the original reason for modern SIMD instruction sets was to speed up multimedia applications and games rather than scientific computing. Because of this implementation (and philosophical) difference, the term "vector" usually refers to vectors of many elements, while "SIMD" usually refers to vectors of a few elements (21).

Figure 2 shows an example of an instruction for a SIMD operation, it is shown schematically how a sum of two vectors is done $\vec{C} = \vec{A} + \vec{B}$. The sum is performed element to element and is executed as a vector sum as well. In contrast, for vector architectures, which offers elegant instruction set that is intended to be the target of a vectorizing compiler, SIMD extensions have three major omissions:

- Multimedia SIMD extensions fix the number of data operands in the opcode.
- Multimedia SIMD usually does not support the sophisticated addressing modes of vector architectures, like stride accesses and gather-scatter accesses[8].
- Multimedia SIMD usually does not offer mask registers to support conditional execution of elements as in vector architectures[9].

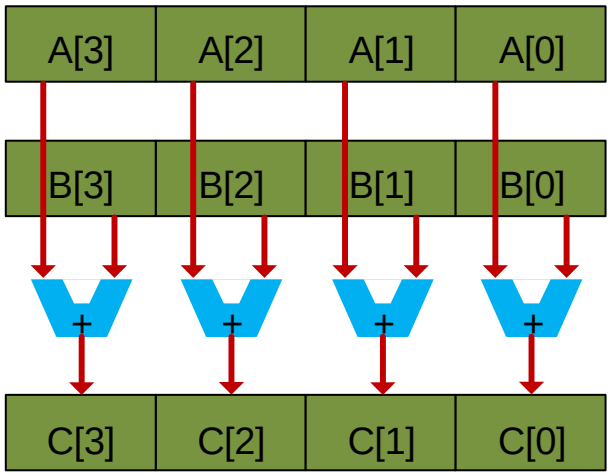


Figure 2: Example of a SIMD parallel addition

[7] It depend on the number and symmetric of lanes implemented.
[8] This limitation is being addressed with the newest SIMD extensions from Intel: AVX2 added support for gather and AVX-512 will include support for gather and scatter.
[9] AVX-512 instructions support 8 opmask registers. Seven of them provide conditional execution and efficient merging of data elements (58).

Usually the SIMD unit is composed of individual sub-units: a float-point unit, an integer/logical unit and a shuffle unit. Moreover, each unit is further divided into lanes. A lane is the minimum building block of a vector. A SIMD unit can be built just by putting next to each other multiple copies of the same lane. Integer lanes are almost identical to the ALU hardware and floating-point lanes are very similar to the FPU hardware. It is useful because is possible to reuse components. The drawback is that lanes have to be designed to support operating with different size, typically 8, 16, 32 and 64-bit.

One advantage of SIMD is that typically the latency of each SIMD instruction is the same as the corresponding scalar operation. It is due to SIMD have an ALU (Lane) for each element. Moreover power consumption on SIMD grows linearly because frequency and voltage do not need to change. Figure 3 shows the Dynamic power equation. By increasing the number of ALU or Lanes, capacitance increases too.

$$P_{dynamic} = \propto C_L V^2 f$$

Where

P=power

∝=activity factor

C=capacitance

V=voltage

f=frequency

Figure 3: Dynamic Power

2.5 Graphics Processing Units

The GPU is a multiprocessor composed of multithreaded *SIMD processors*[10]. Multithreaded *SIMD processors* are similar to a Vector Processors, but they have many parallel functional units instead of a few that are deeply pipelined, as vector processors have. *SIMD processors* are full processors with separate PCs and are programmed using threads. Each *SIMD processor* is composed of a set of lanes. These lanes are quite similar to the lanes used by SIMD multimedia extensions.

The multiple *SIMD processors* in a GPU act as independent MIMD cores, just as many vector computers have multiple vector processors. The main difference between GPU and vector processors is multithreading, which is fundamental to GPUs and is missing from most vector processors. Nevertheless, this is not a rule. There are Vector processors that exploit multithreading by merging ILP and DLP (22). Multithreading in GPUs is used to hide DRAM latency (15).

[10] SIMD processor: a processor focused on perform SIMD operations, also called stream processors.

GPUs introduced a new parallel execution model called Single Instruction, Multiple Thread (SIMT) (23). Here multithreading is simulated by SIMD processors. Each processor have multiple threads (called work-items), which execute in lock-step, and are analogous to SIMD lanes. The main benefit from SIMT is to reduce instruction fetching overhead (24).

2.6 SIMD History

Nowadays almost every processor includes a SIMD unit, common examples of this are computers, tablets, smartphones, and videogames. Before the mid-90s personal computers (PCs) became popular, where Intel was one of the pioneers to introduce SIMD technology to PCs with this, when Intel tried to increase the performance of multimedia applications. Table 1 shows a brief summary (not exhaustive) of the last 20 years of SIMD implementations.

Year	Description
1994	Hewlett-Packard introduced the "Multimedia Acceleration eXtensions" (MAX). It was 64-bit wide (25).
1995	Sun Microsystems introduced the "Visual Instruction Set" (VIS). It was used by SPARC processors. It was 64-bit wide (26).
1996	Intel launched the MMX. This unit was not so popular due to its technical limitations. The original name was supposed to be "Sub-word Parallelism" but marketing team decided to change it. MMX shared registers with the FPU. It was 64-bit wide. (9).
1996	MIPS Technologies developed its own SIMD implementation. It was called "MIPS Digital Media eXtension" (MDMX). It was pretended to be launched as coprocessor of MIPS-V instruction set. Unfortunately MIPS-V was never launched neither MDMX. It was 64-bit wide and share registers with the FPU (27).
1997	AIM (Apple, IBM and Motorola) introduced the AltiVec instruction set on its G3 processors. It was 128-bit wide and supported SIMD floating-point operations (28).
1999	Intel developed a new SIMD implementation called "Streaming SIMD Extension" (SSE). It doubled the size of the register respect MMX from 64-bit to 128-bit wide. Also SSE introduced support to perform floating-point operations of single-precision. SSE were more popular than MMX (10).
2000	AMD launched its own SIMD implementation called 3DNow! that was similar to SSE. Because AMD market was relatively small and 3DNow! was implemented only in AMD processors, 3DNow! was not popular. 3DNow! shared registers with the FPU.
2002	Intel released the next generation of SSE (SSE2). It fixed many problems with previous implementation and introduced support to perform double-precision floating-point operations. It was launched with the Pentium IV and became very popular (29).

Year	Description
2004	Intel launched SSE3. It introduced 13 new instructions over SSE2 which are primarily designed to improve thread synchronization and specific application areas such as media and gaming (29).
2006	Intel launched SSE4. It introduced 54 new instructions over SSE3. It introduced support to perform floating-point dot products (30).
2008	Intel announced a new SIMD set instruction for x86-64 architecture. It was called "Advanced Vector Extension" (AVX). One of the main improvements was to double the size of the SIMD register from 128-bits to 256-bits, support up to four operand instruction and fused operations. (31).
2011	Intel launched Sandy Bridge microarchitecture under the Core brand. Sandy Bridge was the first microarchitecture that implements AVX (32).
2013	Intel launched Haswell microarchitecture which includes the second generation of AVX (AVX2). (14)
2015	This year AVX-512 (AVX3) will be launched with Knights Landing Xeon Phi processor. AVX3 doubles the size of the registers from 256-bits to 512-bits (33).

Table 1: Summary of SIMD implementations on PC

3 MIPS Architecture

3.1 Brief History of MIPS Company

A group of researchers from Stanford University led by John L. Hennessy in the early 1980s created the first MIPS (Microprocessor without Interlocked Pipeline Stages) processor (34). They believed that using Reduced Instruction Set Computing (RISC), combined with excellent compilers and hardware that exploit pipelining to execute these instructions, could produce a faster processor using less die area. These were such success that MIPS Computer Systems, Inc. was formed in 1984 to commercialize the MIPS architecture.

On 1992 Silicon Graphics Inc. (SGI) acquired the company and rename it as MIPS Technologies, Inc. (35). SGI spun MIPS out on June 20th, 2000 by distributing all its interest as stock dividend to the stockholders (36).

On 8 February 2013 MIPS Technologies, Inc. was acquired by Imagination Technologies (37). Over the years MIPS has been focused on embedded markets such as Windows CE devices, routers, video game consoles (Nintendo 64, PlayStation).

3.2 History of the MIPS ISA

From the first proposal ISA presented by John L. Hennessy research team. MIPS has been evolving introducing new features generation by generation. A brief summary of the evolution of MIPS processors is presented below. Each new MIPS generation is a superset revision from the previous one. Additionally, newer versions are fully backward compatible.

MIPS I

The first MIPS design was introduced in 1985. It was designed with 32 general purpose registers of 32 bits each. It had a 5 stages pipeline. Floating point on MIPS was originally done in a separate chip called coprocessor 1 also called the Floating Point Accelerator (FPA). It had 32 single precision (32-bit) floating-point registers. Double precision was implemented by using pairs of single precision registers to hold operands. All MIPS chips use the IEEE 754 floating-point standard, both the 32-bit and the 64-bit versions. (38).

MIPS II

It was introduced in 1990. It is a superset revision from the previous one.

MIPS III

It was introduced in 1992. It increases the width of the registers and integer units up to 64-bits. It introduced square root floating-point instruction. It is a superset revision from the previous one.

MIPS IV

It was introduced in 1994. It introduced floating-point fused operations.

MIPS V

It was announced in 1996 but it was never launched to the market. A major improvement was the MIPS Digital Media Extension (MDMX) which was a multimedia extension designed to improve the performance of 3D graphics applications. MDMX was one of the first SIMD implementations by MIPS.

3.3 Current MIPS ISA

After departure of MIPS Technologies from Silicon Graphics, architecture definition was changed to refocus on the embedded market. All releases from 1 to 6 have both 32 and 64-bit versions. MIPS64 versions are supersets of the corresponding 32-bit versions. It means that MIPS32 ISA is part of the MIPS64 ISA. MIPS32 instructions are sign extended to work in MIPS64 registers. Additionally, newer versions are fully backwards compatible. For instance, a processor that implements MIPS64 Release 6 ISA can execute instructions from MIPS32 Release 6 ISA, but not vice versa. In fact it can execute any previous MIPS32 and MIPS64 release.

MIPS32 and MIPS64 release 1

Introduced in 1999. It is mostly based on MIPS II but it borrows some features from MIPS III, MIPS IV and MIPS V.

MIPS32 and MIPS3264 release 2

Introduced in 2002. It is a superset from the previous one and several improvements were made. Some of them are support for 64-bit coprocessor for 32-bit and 64-bits CPU, support for Virtual and Physical Memory, support for larger TLB pages, and support for external interrupts controller.

MIPS32 and MIPS64 release 3

Introduced in 2010. It also introduces microMIPS32 and microMIPS64 instruction sets which instructions of 16 and 32-bits respectively. The main idea of microMIPS32 and microMIPS64 is to have all the functionality of MIPS32 and MIPS64 with smaller code sizes. It introduced support for non-executable and write-only virtual pages and for certain IEEE-754-2008 FPU behaviors.

MIPS32 and MIPS64 release 4

This name was skipped for commercial issues. Number 4 in Asia is considered unlucky.

MIPS32 and MIPS64 release 5

Introduced in 2013. Here optional components appeared, e.g. virtualization, multithreading, multimedia and DSP.

MIPS32 and MIPS64 release 6

Introduced in 2014. One of the most important changes is that the instruction set has been simplified by removing infrequently used instructions and rearranging instruction encoding. For example, unaligned memory accesses are now directly supported, without requiring special instructions. Figure 4 shows the evolution of MIPS architecture from the original MIPS I to MIPS32/64 Release 6. Also optional modules are shown.

Instructions that were removed from previous versions than Releases 6, still available and they can be implemented to allow backward compatibility. Nevertheless, they are clearly marked as obsolete and programmers should avoid using these instructions.

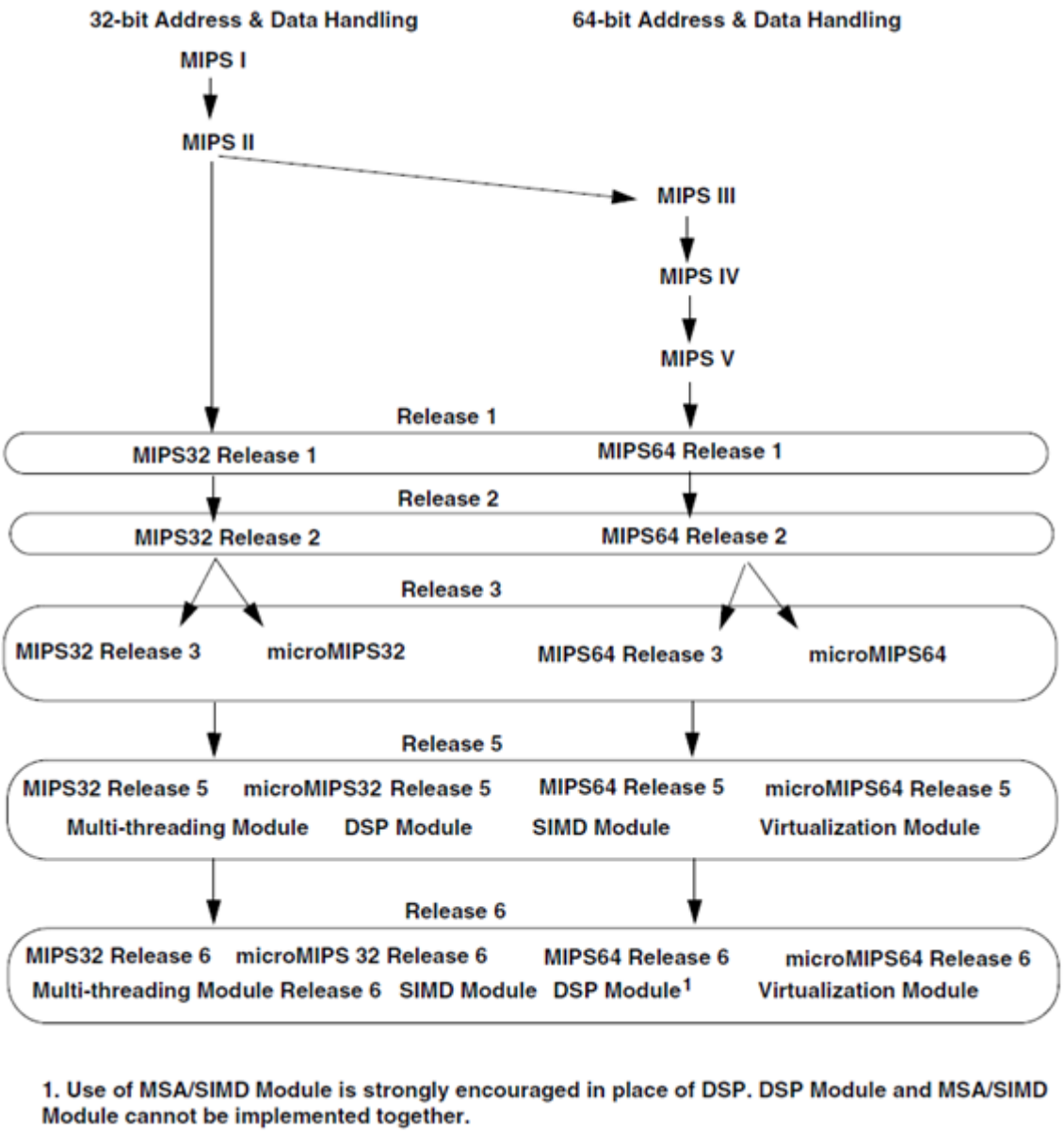


Figure 4: MIPS32/64 Releases and optional modules (39)

3.4 Optional Components

MIPS architecture is targeted to plenty of markets, in order to fill all possible requirements. Imagination Technologies© provides some optional components fully compatible with MIPS32, MIPS64, microMIPS32 and microMIPS64 ISA from release 5 and higher. Figure 4 shows the MIPS evolution from the original MIPS I to the MIPS32/64 Release 6 Figure 5 shows optional ISA modules compatible with MIPS32/64 Release 6. They are used to improve some specific applications.

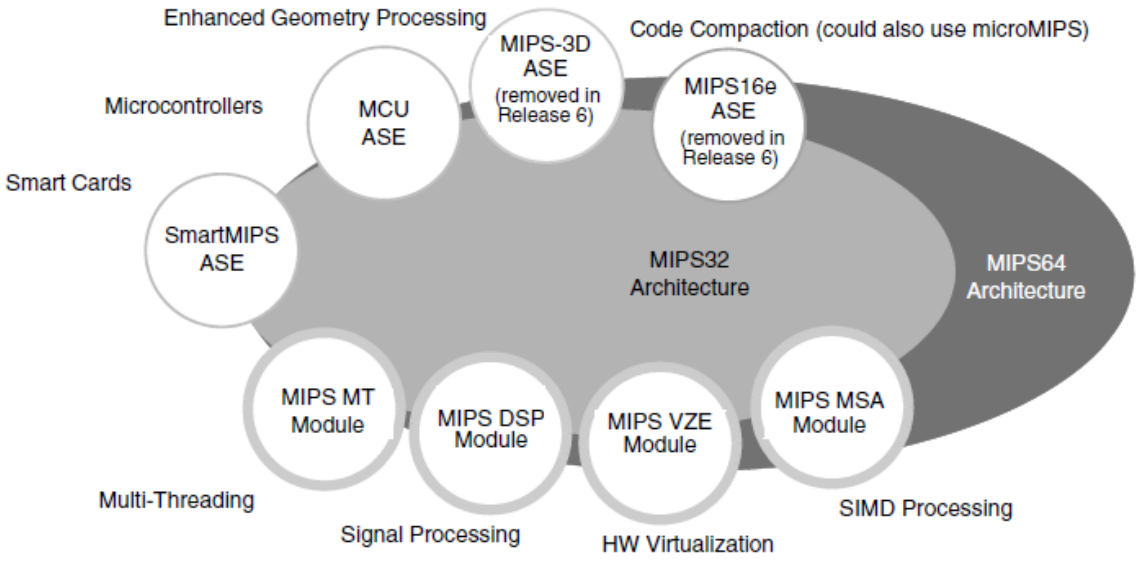


Figure 5: Optional components supported by MIPS32 and MIP64 (40)

3.5 Brief description of the optional components available on Release 6

The following briefly describes the modules compatible with MIPS Release 6 (41).

MCU

Provides enhanced handling of memory-mapped I/O registers and lower interrupt latencies. This is intended to extend the interrupt controller support, typically required in microcontroller system designs.

SmartMIPS

It is an instruction set extension designed to improve the performance and reduce memory consumption of MIPS-based smart card or smart objects systems. These are very lower-power CPUs whose biggest task is encryption/decryption.

MIPS MT

The MIPS MT provides the micro-architectural support needed to perform multithreading. This includes support up to two virtual processors and lightweight contexts.

MIPS DSP[11]

The MIPS DSP provides enhanced performance of signal-processing applications. Roughly it provides DSP functionality in MIPS processor cores.

MIPS VZE

The MIPS Virtualization Module provides hardware acceleration of virtualization of Operating Systems.

MIPS MSA

The MIPS SIMD Architecture provides high performance parallel processing of vector operations through the use of 128-bit wide vector registers. MIPS MSA is described in chapter 4. It substitutes MIPS Digital Media Extension (MDMX).

[11] MSA is recommended substitute. DPS is not allowed if MSA is implemented. USE of DPS Module is strongly discouraged from Release 6 onwards.

4 The MIPS® SIMD Architecture module

Common operations that are used in multimedia processing which can be vectorized as SIMD operations include:

- Addition and subtraction
- Multiply
- Logical and arithmetical shift operations
- Logical operations (AND, OR, Nor, XOR, etc.)
- Load and Store
- Fused operations like Multiply-Add, dot product

MIPS SIMD Architecture (MSA) module was implemented with strict adherence to the RISC design principles pioneered by MIPS. It is a simple, yet very efficient instruction set carefully selected with a hardware implementation that is efficient in terms of speed, area and power consumption.

The MSA introduces 186 new instructions that operate on vector registers of 128-bit wide. It supports four formats:

Data Format	Characteristics	Format Abbreviation
Byte	16 elements of 8-bit wide	.B
Halfword	8 elements of 16-bit wide	.H
Word	4 elements of 32-bit wide	.W
Doubleword	2 elements of 64-bit wide	.D
Vector	Whole 128-bit wide vector	.V

Table 2: Supported formats by MSA

Figure 6 shows the distribution and layout representation of elements for all four data formats. MSA vectors are stored in memory starting from the least significant bit at the lower byte address. The byte order can follow big or little endian conventions.

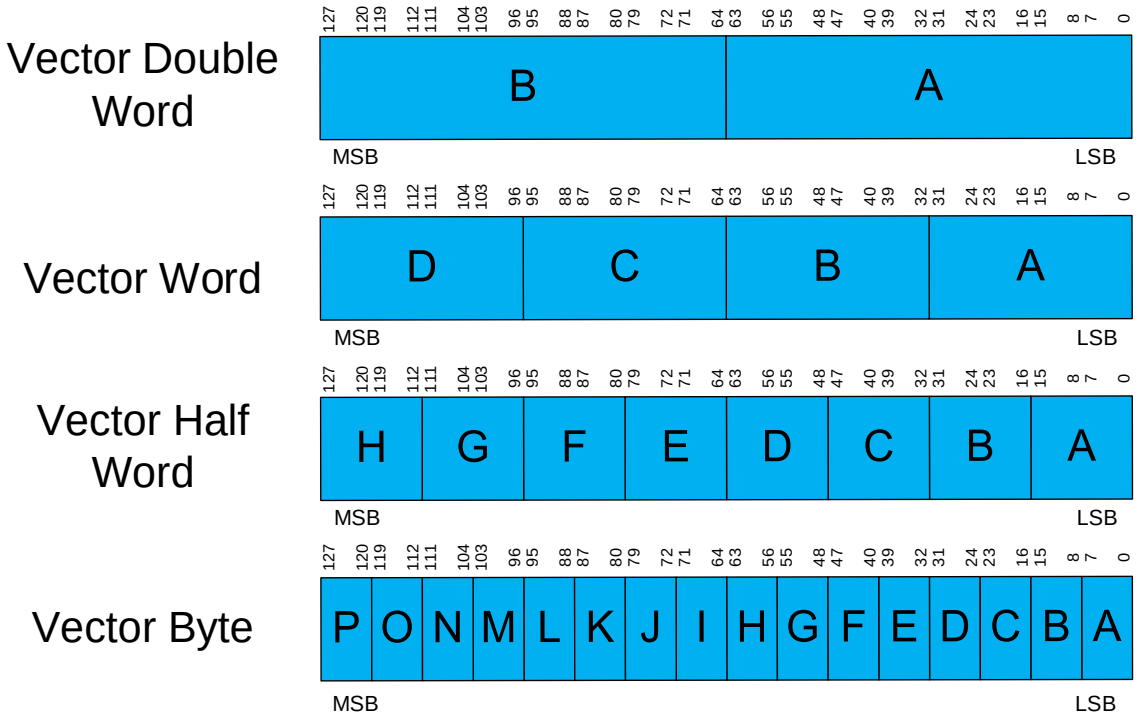


Figure 6: MSA formats

The 64 least significant bits of MSA are shared with the floating-point unit (FPU). MSA and FPU cannot be present, at the same time, unless the FPU has 64-bit floating-point registers. Since MSA floating-point implementation is compliant with the IEEE Standard for Floating-Point Arithmetic 754™ 2008 (42) and supports single and double precision. Moreover, MIPS ISA indicates that FPU and MSA registers are mapped together. It means that MSA data is destroyed when FPU instructions are executed and vice versa.

Programmer is responsible of saving registers if FPU and MSA instructions are mixed (kind of context switch). An option could be avoid to use FPU instructions and use MSA FPU instructions using the start address of the FPU data as a start address of the vector. Remaining elements will have trash-data. A better option is to vectorize your code (see GCC section).

4.1 Instruction Decoding and Formats

The MSA instructions are encoded in 32 bits following RISC principles. These instructions are well formed according to RISC. Many bit-ranges are common. For example, MSA

identifier, MSA opcode and register addresses (WS, WD, and WT). Table 3 shows all decoding formats. This regularity makes easier to decode it. Instructions are distributed in 12 groups according to the number of operands, operation and behavior. All MSA instructions except branches use 40 minor opcodes in MSA major opcode 0x1E. MSA branch instructions are encoded in the COP1 opcode 0x11.

- I8 8-bit immediate value. 10 instructions.
- I5 5-bit immediate value. 11 instructions.
- BIT Immediate bit index. 12 instructions.
- I10 10-bit immediate value. 1 instruction.
- 3R 3-register operations. 63 instructions.
- ELM Immediate element index. 9 instructions.
- 3RF 3-register floating-point or fixed-point operations. 41 instructions.
- VEC Bit wise operations over whole vector. 7 instructions.
- MI10 Memory operations, immediate offset 10-bit. 2 instructions.
- 2R 2-register operations. 4 instructions.
- 2RF 2-register floating-point or fixed-point operations. 16 instructions.
- Branch Opcode is shared with COP1 instructions. Branches are taken at element level, immediate offset 16-bit. 10 instructions.

Mnemonic	Instruction layout								Type
INST.df[12]	MSA OP	operation	df	wt	ws	wd	opcode		3R
INST.df	MSA OP	operation	df	wt	ws	wd	opcode		3RF
INST.df	MSA OP	df	i8		ws	wd	opcode		I8
INST.df	MSA OP	operation	df	u5	ws	wd	opcode		I5
INST.df	MSA OP	operation	df	s10		wd	opcode		I10
INST.df	MSA OP	operation	df/m		ws	wd	opcode		BIT
INST.df	MSA OP	operation	df/n		ws	wd	opcode		ELM
INST.V[13]	MSA OP	operation	wt		ws	wd	opcode		VEC
INST.df	MSA OP	s10			rs	wd	opcode		MI10
INST.df	MSA OP	operation	df		ws	wd	opcode		2R
INST.df	MSA OP	operation	df		ws	wd	opcode		2RF
INST.V	COP1	operation	df	wt	s16				COP1

Table 3: Decode of instructions

[12] df - supported data format abbreviation, see Table 2

[13] V – vector variable of type V

Where:

MSA OP: Major Opcode space, this field has a constant binary value of 011110 and identify MSA instructions.

COP1: Major Opcode space, this field has a constant binary value of 010001 and identify coprocessor 1 instructions. Some of them are overridden by MSA.

ws: 5-bit MSA register address of source operand 1, e.g. $w0, $w1, …, $w31

wt: 5-bit MSA register address of source operand 2, e.g. $w0, $w1, …, $w31

wd: 5-bit MSA register address of source operand 3 and MSA register address destination, e.g. $w0, $w1, …, $w31

rs: 5-bit general purpose register (GPRs) address, e.g. $0, $1, …, $31

opcode: Minor opcode space, this field identify instructions by type like 2R, 3R, ELM and so on.

df: destination data format, which could be a *byte*, *halfword*, *word*, *doubleword* or vector. See Table 2.

df/n: vector register element of index *n*, where *n* is a valid index value for elements of data format df. See Figure 7.

df/m: Immediate value valid as a bit index for the data format df. See Figure 8.

u5: Immediate unsigned value of 5-bit

i8: Immediate value of 8-bit

s10: Immediate signed value of 10-bit

s16: Immediate signed value of 16-bit

operation: Instruction name.

df/n[1]	*Bits 5…0*			
	00nnnn	100nnn	1100nn	11100n
	Byte	Halfword	Word	Doubleword

df/n	*Bits 5…0*			
	01nnnn	101nnn	1101nn	11101n
	Reserved			

1. Bits marked as *n* give the element index value.

Figure 7: Data format and element index field encoding

df/m[1]	*Bits 6…0*			
	0mmmmmm	10mmmmm	110mmmm	1110mmm
	Doubleword	Word	Halfword	Byte

1. Bits marked as *m* give the bit index value.

Figure 8: Data format and bit index field encoding

4.2 GCC support

Imagination Technologies (which is the owner of MIPS Technologies), provides some developer tools. One of this tools is called Codescape (43) which includes GCC that supports all MIPS configurations including a library to support MSA instructions. Simple C programming allows portable codes in a short development time.

MSA toolchain includes

- Built-in intrinsic and data-types for all vector formats and instructions available from C/C++ programming
- Support from common operators (+,-,*) that can be used on vector data-types.
- Complete replacement for hand-coded assembly
- Compiler optimization by auto vectorization

The MIPS SIMD Architecture is designed to meet multimedia requirements and other compute-intensive applications. Video compression has a typical pixel depth of 8-bits or 10-bits; further mathematical operations can take intermediate results to 16-bits or 32-bits. Therefore using an implementation of a typical 128-bit vector register size SIMD processor,

it is possible to make a four to eight time reduction to mathematical and data load and store operations (44).

The use of built-in data types and intrinsics makes C code quickly portable across all MSA core implementations.[14] Moreover, it also indirectly instructs the compiler to make the best use of the SIMD instructions and architecture. For each instruction, the compiler provides a C-style built-in to be used in conjunction with vector-type data structures in order to target MSA vector registers. The preferred coding style is to use regular C arithmetic operators on vector data types and fall back to built-ins for complex MSA instructions which the compiler is unable to relate to vectorized C code (45).

To enable MSA features when compiling, the command line option –mmsa has to be used. The command line option –mfp64 must be used in conjunction with –mmsa. Moreover, to enable auto vectorization the command line option –O2 could be used.

```
$ mips-mti-elf-gcc –O2 –o test test.c –mmsa –mfp64 –T script
```

Figure 9: GCC compilation using MSA

It is required to specify a linker script (.ld files) with the –T option to build applications for bare-metal[15] targets, the -T option is required when linking to avoid references to undefined symbols (46).

```
ENTRY(main)
PROVIDE (__stack = 0);
SECTIONS
{
    . = 0x1fc00000;
    .text : {
            *(Inicio)
            *(.text);
      }
    . = 0x1fc80000;
    .rodata : { *(.rodata) }
    .data : { *(.data) }
    .bss : { *(.bss) }
}
```

Figure 10: Linker script example.

Figure 11 shows a C code example that perform the addition of two integer arrays. The arrays are added in groups of 4 elements. Vector type variables are declared using the vector extension provided by GCC.

[14] MIPS32, MIPS64, microMIPS32 and microMIPS64 for Release 5 and higher.
[15] Applications that run in computers without operating system

```
typedef int v4i32 __attribute__ ((vector_size(16)));
#define N 16
/* Pointers to the input arrays */
int *input1, *input2 ;
/* Pointer to the output array to store addition */
int *output;
/* Vectors of type word */
v4i32 a, b, c;
int i;
/* Loop unrolled 4x */
for (i = 0; i < N; i += 4)
{
      /* Load 4 elements of array input1 */
      a = *((v4i32 *)(input1 + i));
      /* Load 4 elements of array input2 */
      b = *((v4i32 *)(input2 + i));
      /* Vector addition */
      c = a + b;
      /* Store addition of 4 elements */
      *((v4i32 *)output + i) = c;
}
```

Figure 11: Example for addition of two integer arrays

5 Tools and resources

5.1 Verilog

Verilog is a hardware description language (HDL). The standard includes support for modeling hardware at the behavioral, register transfer level (RTL), and gate-level abstraction levels, and for writing test benches. Verilog syntax is similar to the C programming language and Verilog codes are shorter than VHDL codes (47).

5.2 Quartus II

The Altera Quartus II is a design software produced and provided by Altera. It is multiplatform and provides analysis and synthesis of Verilog and VHDL hardware description languages. The Quartus II software includes solutions for all phases of FPGA design. We used 14.1 web edition version. It is a free version that only provides compilation and programing for a limited number of Altera devices. Cyclone FPGA family is fully supported. Some advanced features like LogicLock Region and Power Analysis are only available in the Subscription Edition that requires a subscription license.

Web edition version of Quartus II also includes a ModelSim starter edition version which is free. ModelSim is a source-level multi-language HDL simulation environment developed by Mentor Graphics. ModelSim supports VHDL, Verilog and SystemC HDL. We used ModelSim version 10.3c to simulate and run test benches.

5.3 DE2-115 board

Since this project is oriented to generate a soft-core implementation, we selected a popular educational FPGA board. The DE2-115 board is equipped with a Cyclone EP4CE115 FPGA with 114,480 logical elements (48). These are a summary of the main features of the DE2-115 board:

- 114,480 logic elements (LEs)
- 266 Embedded 18x18 multipliers
- 128 MB SDRAM
- 2 MB SRAM
- 8 MB Flash
- 528 User I/Os
- Serial port
- Three 50MHz oscillators

6 Basic components

There are basic circuits that are used in the implementation like adders, multipliers and multiplexors. We are going to describe how they were selected.

6.1 Adder

An adder is a digital circuit that produces the arithmetic sum of two binary numbers. The majority of the adders use the full-adder as minimum component. A full adder is a combinational circuit that performs the addition of three bits. Figure 12 shows the implementation of a full adder. Figure 13 shows the block representation of a full adder. We implemented three types of adders and evaluated their performance and resources used in our FPGA. The implementation of 4-bit adders is explained next.

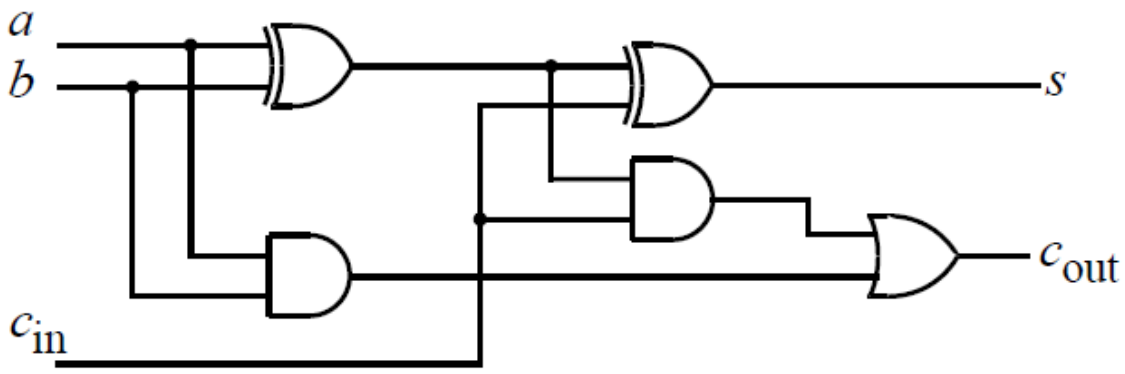


Figure 12: One-bit full adder implementation

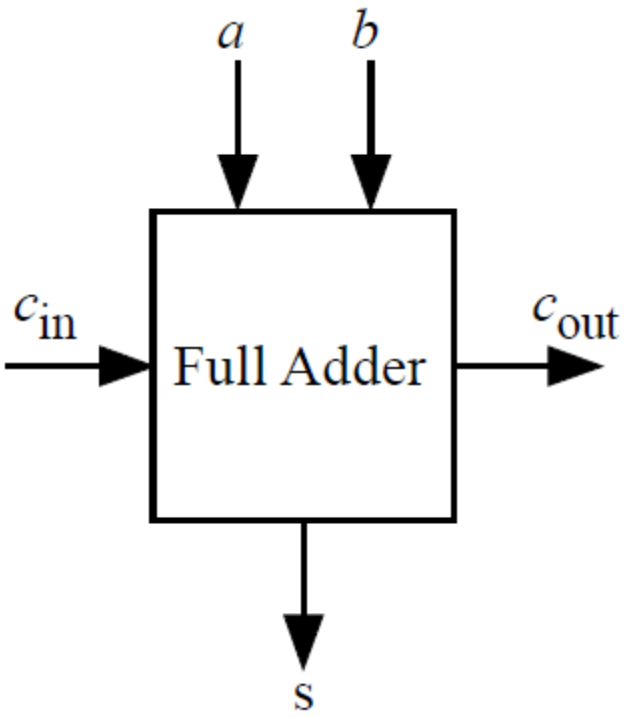


Figure 13: One-bit full adder block

6.1.1 Ripple Carry Adder

The ripple carry adder consists of N full adders to add N-bit numbers. Full adders are connected in a cascade. It means that from the second full adder, carry input of every full adder is the carry output of its previous full adder. In the ripple carry adder, the result is known after the carry signal has rippled through the whole adder. As a result the sum will be valid after a considerable delay time. Figure 14 shows an implementation of a 4-bit ripple carry adder.

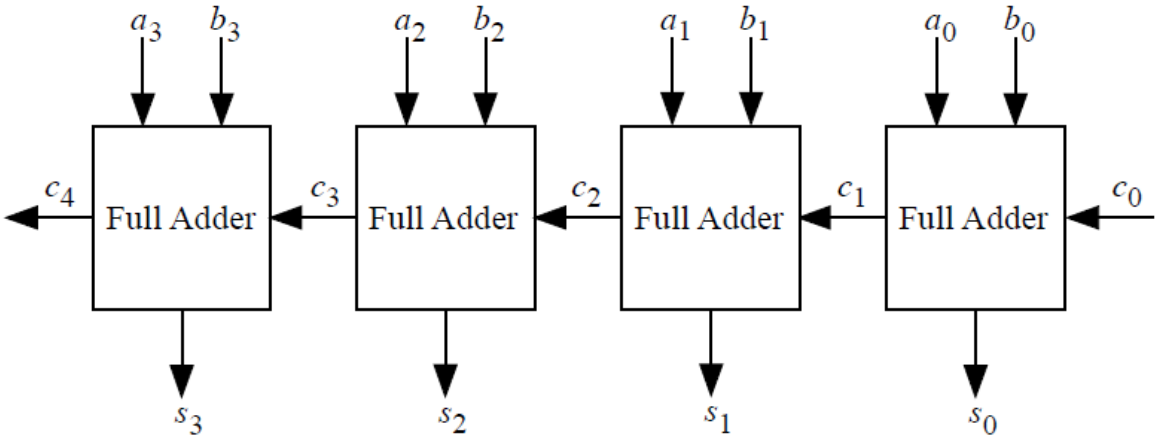


Figure 14: 4-bit ripple carry adder

6.1.2 Carry Look-ahead Adder

The carry look-ahead adder reduces the carry delay time by calculating the carry signals in advance, based on the input signals. Figure 15 shows a 4-bit carry look-ahead adder implementation. The carry logic block is implemented using boolean equations shown in Figure 16. The disadvantage of the carry look-ahead adder is that the carry logic block becomes complicated as the size of the adder grows. The carry look-ahead adder usually is implemented as 4-bit or 8-bit blocks. These adder blocks are connected like full adders in ripple carry adder are connected the same fashion.

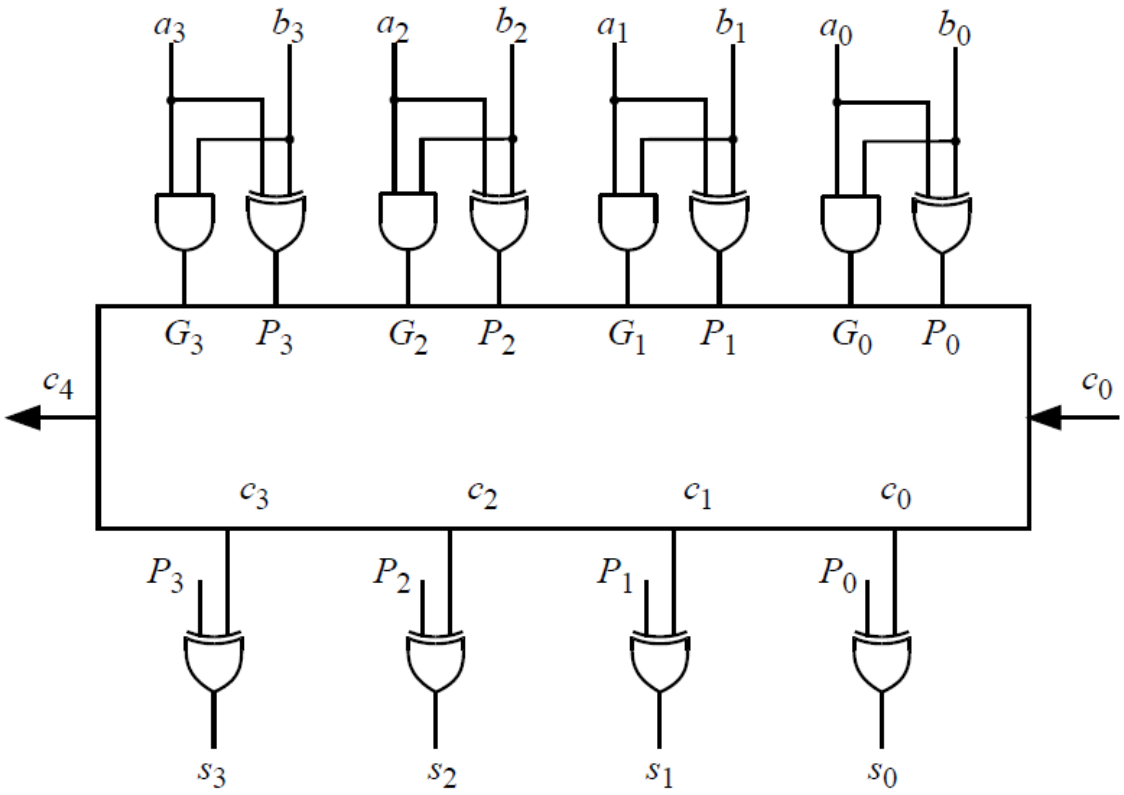


Figure 15: 4-bit carry look-ahead adder

$$
\begin{aligned}
c_1 &= G_0 + P_0.c_0 \\
c_2 &= G_1 + P_1.G_0 + P_1.P_0.c_0 \\
c_3 &= G_2 + P_2.G_1 + P_2.P_1.G_0 + P_2.P_1.P_0.c_0 \\
c_4 &= G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3.P_2.P_1.G_0 + P_3.P_2.P_1.P_0.c_0
\end{aligned}
$$

Figure 16: Boolean equations used in 4-bit carry look-ahead adder

6.1.3 Kogge-Stone Adder

The Kogge-stone adder is a parallel prefix form of carry look-ahead adder. It is the fastest adder with focus on design time and is the common choice for high performance adders in industry (49). The drawback of the Kogge-Stone adder is that it occupies a large silicon area. Figure 17 shows a 4-bit Kogge-stone adder. It is composed by carry operators as shown in Figure 18. Boolean equations that are used to implement the carry operators and calculate the final result “S” are show in Figure 19.

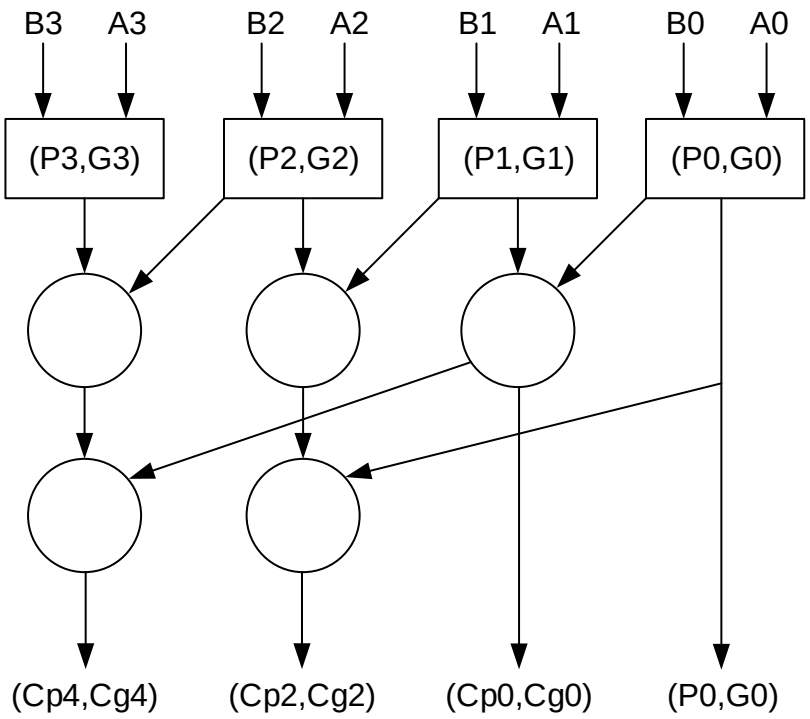


Figure 17: 4-bit Kogge-Stone Adder

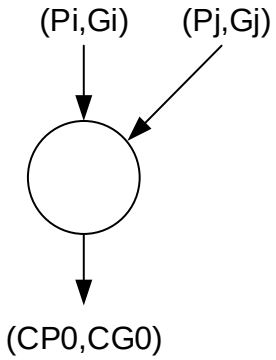


Figure 18: Carry operator

$$CP_0 = P_i and P_j$$

$$CG_0 = (P_j and G_j) or G_i$$

$$C_{i-1} = (P_i and C_{in}) or G_i$$

$$S_i = P_i xor C_{i-1}$$

Figure 19: Boolean equations involved in Kogge-Stone adder

6.1.4 Adder evaluation

We implemented in the Cyclone IV FPGA the adders using a 16-bit width. Then these 16-bit adders were used as basic adder blocks and 32-bit of the lower and faster 16-bit version adders were implemented. Finally the faster adder, the Kogge-Stone adder was expanded

up to 64-bit. An interesting observation is that the adder used by Quartus II which is called Quartus II adder is even faster than the Kogge-Stone implementation.

This might happen because Kogge-Stone adder is losing performance by the 16-bit block distribution. Also Quartus II selects the best adder algorithm depending on the width. Examples of other adder algorithms are Carry Select Adders and combinational adders (50). Finally, it has been decided to use the adder provider by Quartus II because it has similar performance and requires less Logical Elements (LEs) than the Kogge-Stone adder. Future implementations should have Kogge-Stone adder implemented with the appropriated width and not using small blocks.

Adder	Width	Number of LEs	Fmax (MHz)
Ripple Carry Adder	16	56	223
Carry Look-Ahead Adder	16	54	215
Kogge-Stone Adder	16	70	247
Ripple Carry Adder	32	109	109
Kogge-Stone Adder	32	212	211
Kogge-Stone Adder	64	423	159
Quartus II Adder	64	192	167

Table 4: Adder performance evaluation

6.2 Multiplier

As it has been said above, at the beginning this multimedia extension is oriented to be implemented in FPGA. Therefore, we are using the DSP blocks included in the Cyclone IV FPGA. Small width values like *byte* (8-bit) and *halfword* (16-bit) are executed directly on one DSP block. DSP blocks in Cyclone IV FPGA are 18-bit width (51). For large integers like word (32-bit) and *doubleword* (64-bit) we use the Karatsuba Algorithm.

The Karatsuba multiplication algorithm is an efficient way to build high bit width integer multiplication, suitable for conserving DSP blocks in return for additional latency and cell area. Karatsuba algorithm is based on a formula for multiplying two linear polynomials which uses only 3 multiplications and 4 additions (52). The formula of the Karatsuba algorithm is:

$$(f_1 x^m + f_0)(g_1 x^m + g_0) = h_2 x^{2m} + h_1 x^m + h_0$$

Figure 20: formula of Karatsuba algorithm

f_0 , f_1 , g_0 and g_1 are m-bit polynomials. The polynomials h_0, h_1 and h_2 are computed by applying the Karatsuba algorithm to the polynomials f_0 , f_1 , g_0 and g_1 as single coefficients and adding coefficients of common powers of x together. The circuit to perform Karatsuba Algorithm is shown in Figure 21.

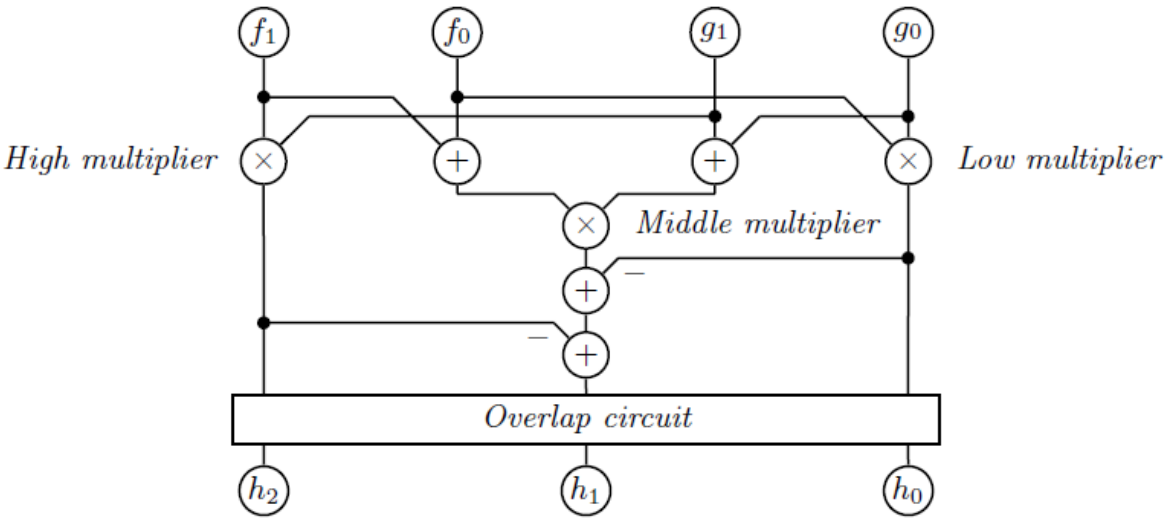


Figure 21: The circuit to perform Karatsuba multiplication

The "Overlap circuit" adds common powers of x in the three generated products. For instance if n=8, then the input polynomials have a degree at most 7, for each of the polynomials f_0 , f_1 , g_0 and g_1. Figure 22 shows the effect of the overlap module. Coefficients to be added together are shown in the same columns.

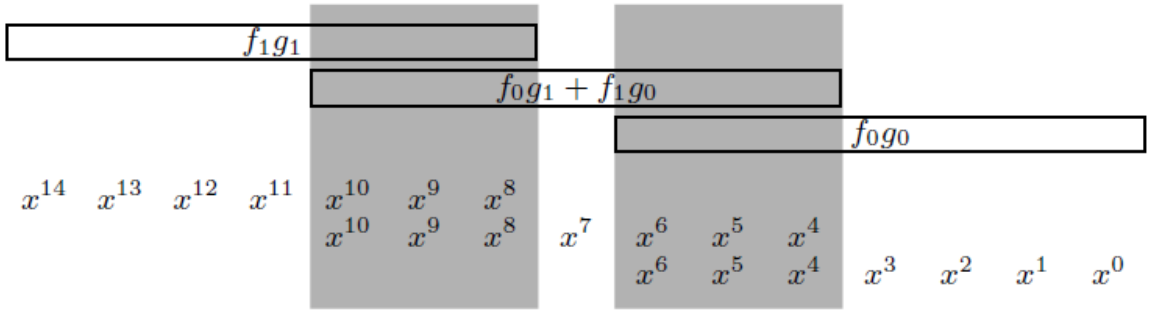


Figure 22: The overlap circuit for the 8-bit Karatsuba multiplier

Example

Let A=197 and B=114 values of 8-bit width. We can split these numbers into their higher and lower 4-bits using 16 as a base. Figure 23 shows a numerical example of the multiplication of 197 by 144 based on the data flow and intermediate results given by the circuit shown in Figure 21. Overlap row is two parallel adders of a few bits each one, a show in Figure 22.

	$A = 197$		$B = 84$	
Operands	$AB = (f_1x + f_0)(g_1x + g_0)$			
	$A = (f_1x + f_0)$		$B = (g_1x + g_0)$	
Split (x=16)	$f_1 = 12$	$f_0 = 5$	$g_1 = 7$	$g_0 = 2$
Operations	12x7=84	12x5=17 17x9=153 153-10=143 143-84=59	7x2=9	5x2=10
	$h_2 = 84$	$h_1 = 59$		$h_0 = 10$
Overlap	$AB = h_2x^2 + h_1x + h_0$ $AB = 84(16^2) + 59(16) + 10$			
Result	$AB = 22458$			

Figure 23: Numerical example of Karatsuba multiplier using base 16

6.3 Divider

Divider is one of the most expensive resources unit to implement. It uses substantial area (50). Divider implementation is not part of this thesis and has been left for future implementation. In order to save area we will use the Floating Point lanes to perform integer division (21). Current implementation does not provide Floating Point SIMD operations but MSA ISA does. As temporary solution the *lpm_divide megafunction* provided by Altera has been used considering a 4-stage pipeline.

A low area, low performance alternative to the *lpm_divide megafunction* is using the elementary school algorithm of processing the number from the most significant bit to the less significant bit. When difference is negative, the next quotient bit is 0 and workspace is untouched. When the difference is zero or positive, the next quotient bit is set to one and workspace is overwritten with the difference value. The quotient bits accumulate in the numerator register, and the remainder accumulates in the workspace as the clock progressed. The ready signal indicates completion. Figure 24 shows this algorithm.

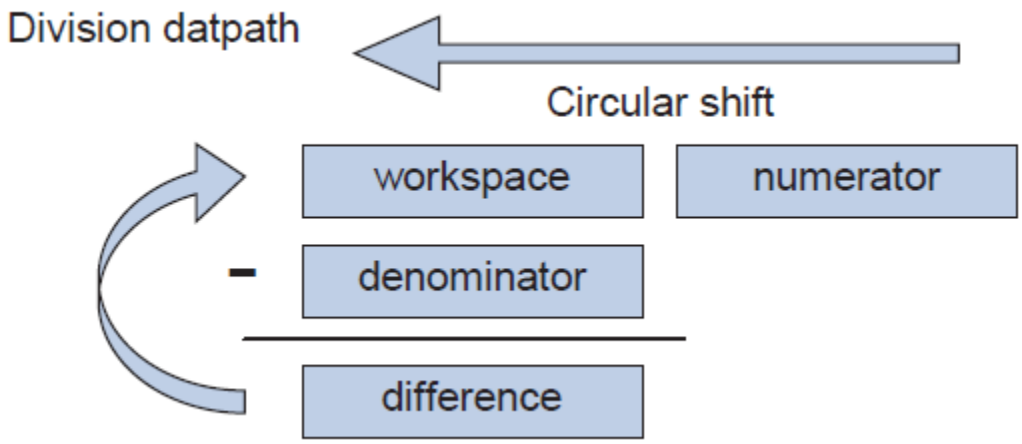


Figure 24: Alternative division algorithm

This alternative divider implementation requires as many clock ticks as width-bits source operands. In other words, a division of 64-bits requires 64 clock ticks. Dividers can be chosen by modifying parameters at configuration file of project.

6.4 Multiplexer

A multiplexer is a logical circuit used to select one of some input signals and forward the selected one to an output signal. Figure 25 shows a block representation of a 4-to 1 multiplexer.

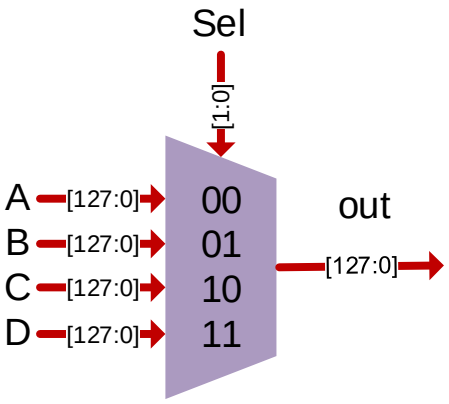


Figure 25: Multiplexer 4-to 1 of 128-bit width

$$out = \left(A\overline{Sel_1}\overline{Sel_0}\right) + \left(B\overline{Sel_1}Sel_0\right) + \left(CSel_1\overline{Sel_0}\right) + \left(DSel_1Sel_0\right)$$

Figure 26: Boolean equation for a 4-to 1 multiplexer

We have decided to code big multiplexers (128-bit width) using the boolean equation shown in Figure 26 because Quartus does not implement always the shortest circuit if a case statement is used.

6.5 Saturated Arithmetic

One extremely useful feature of MIPS SIMD Architecture technology is its support for saturated integer arithmetic. In saturated integer arithmetic, computational results are automatically clipped by the processor to prevent overflow and underflow conditions. This differs from normal wraparound integer arithmetic where an overflow or underflow result is retained. Saturated arithmetic is handy when working with pixel values since it eliminates the need to explicitly check is the result of each pixel calculation for an overflow or underflow condition. MIPS SIMD Architecture technology includes instructions that perform saturated arithmetic using 8-bit, 16-bit, 32-bit and 64-bit integers, all signed and unsigned.

Figure 27 shows an example of 8-bit signed integer subtraction using wraparound and saturated arithmetic. An overflow condition occurs if the two 8-bit signed integers are subtracted using wraparound arithmetic. With saturated arithmetic, however, the result is clipped to the lowest possible 8-bit signed integer value. MIPS SIMD Architecture also supports saturated integer addition, as shown in Figure 28. Table 5 summarizes the saturated arithmetic range limits for all possible integer sizes and sign types.

Integer Type	Lower Limit	Upper Limit
8-bit signed	-128 (0x80)	+127 (0x7F)
8-bit unsigned	0	+255 (0xFF)
16-bit signed	-32768 (0x8000)	+32767 (0x7FFF)
16-bit unsigned	0	+65535 (0xFFFF)
32-bit signed	-21447483648 (0x80000000)	+2147483647 (0x7FFFFFFF)
32-bit unsigned	0	+4294967295 (0xFFFFFFFF)
64-bit signed	-9.2233E+18 (0x8000000000000000)[16]	+9.2233E+18 (0x7FFFFFFFFFFFFFFF)[17]
64-bit unsigned	0	+1.8446E+19 (0xFFFFFFFFFFFFFFFF)[18]

Table 5: Range Limits for Saturated Arithmetic

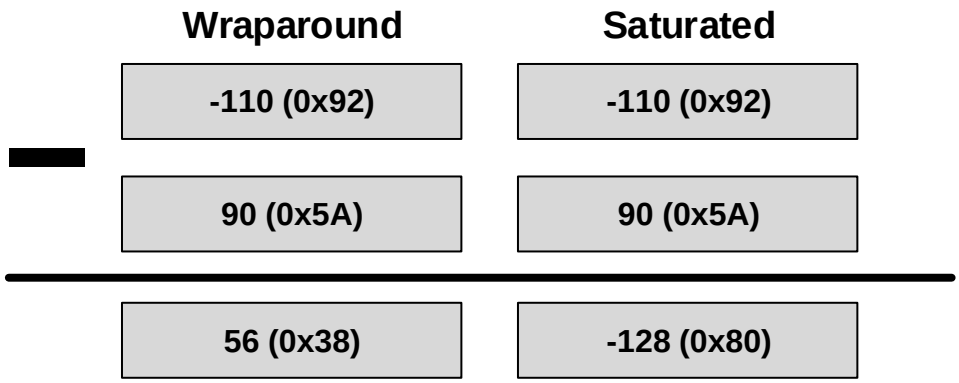


Figure 27: 8-bit signed integer subtraction using wraparound and saturated arithmetic

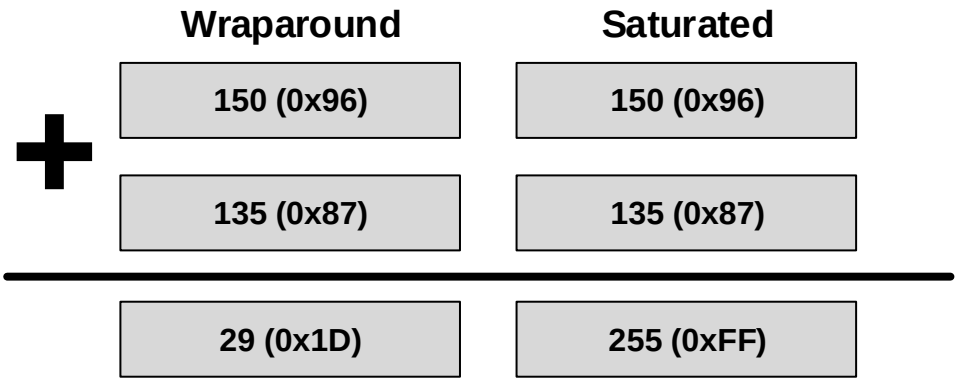


Figure 28: 8-bit unsigned integer addition using wraparound and saturated arithmetic

[16] It is –9,223,372,036,854,775,808
[17] It is 9,223,372,036,854,775,807
[18] It is 18,446,744,073,709,551,615

7 The MIPS SIMD Architecture Instruction Set

The MIPS SIMD Architecture (MSA) consists of integer, fixed-point, and floating-point instructions, all encoded in the MSA major opcode space (31-26 bit). The semantics of most MSA instructions are defined at the granularity of vector elements. A few instructions have semantics that consider the whole SIMD operand as a bit-vector, e.g. bitwise logical operations.

For certain instructions, the source operand could be an immediate value or a specific vector element selected by an immediate index. The immediate or vector element is being used as a fixed operand across all destination vector elements.

The MSA integer instruction set can be partitioned into the following functional groups:

- Data transfer
- Arithmetic
- Comparison
- Conversion
- Logical and Shift
- Unpack and Shuffle
- Insertion and Extraction

7.1.1 Data Transfer

The data transfer group contains instructions that copy packed integer data values from one MSA register to another, and also between general-purpose registers and control registers. Table 6 shows the MSA data transfer instructions. These instructions are executed in the MSA unit and in the MIPS core.

Mnemonic	Type	Description	Implementation
FILL.df	2R	Vector Fill from GPR	Figure 29
CFCMSA	ELM	GPR Copy from MSA Control Register	Figure 29
COPY_S.df	ELM	Copy from MSA to GPR Signed	Figure 29
COPY_U.df	ELM	Copy from MSA to GPR Unsigned	Figure 29
CTCMSA	ELM	GPR Copy to MSA Control Register	Figure 29
INSERT.df	ELM	GPR Insert Element	Figure 29
MOVE.V	ELM	Vector Move	Figure 29
LDI.df	I10	Immediate Load	Figure 29
LD.df	MI10	Vector Load	Figure 29
ST.df	MI10	Vector Store	Figure 29

Table 6: Data transfer instructions in MSA

Figure 29 shows data-paths of the MIPS32 core and the SIMD unit. The figure shows that stages are aligned and there are only four paths required to share data. These paths are:

- A: Path A is used to copy one element from MSA register to GPR. The block called DFN chooses the element selected by the field “df/n” and depending on the instruction sign extension is performed. Also data from MSA control register is copied using this data path.
- B: Path B is used to copy data from GPR to MSA registers. Special unit 3 is used to transform the scalar data into vector data.
- C: Path C is used to send data from MSA register to memory. The MIPS32 core calculates the memory address like in integer store instructions using the “rs” instruction field to read the address base from GPR and the “s10” instruction field as address offset.
- D: Path D is used to receive data from memory to MSA registers. The MIPS32 core calculates the memory address like in integer read instruction.

MOVE instruction is executed only in the SIMD unit. It copies values from MSA register to MSA register. The MSA Control block in the MIPS32 core represents the “Decode extension” implemented into the original MIPS32 core to support fundamental Release 6 features needed by the MIPS SIMD unit.

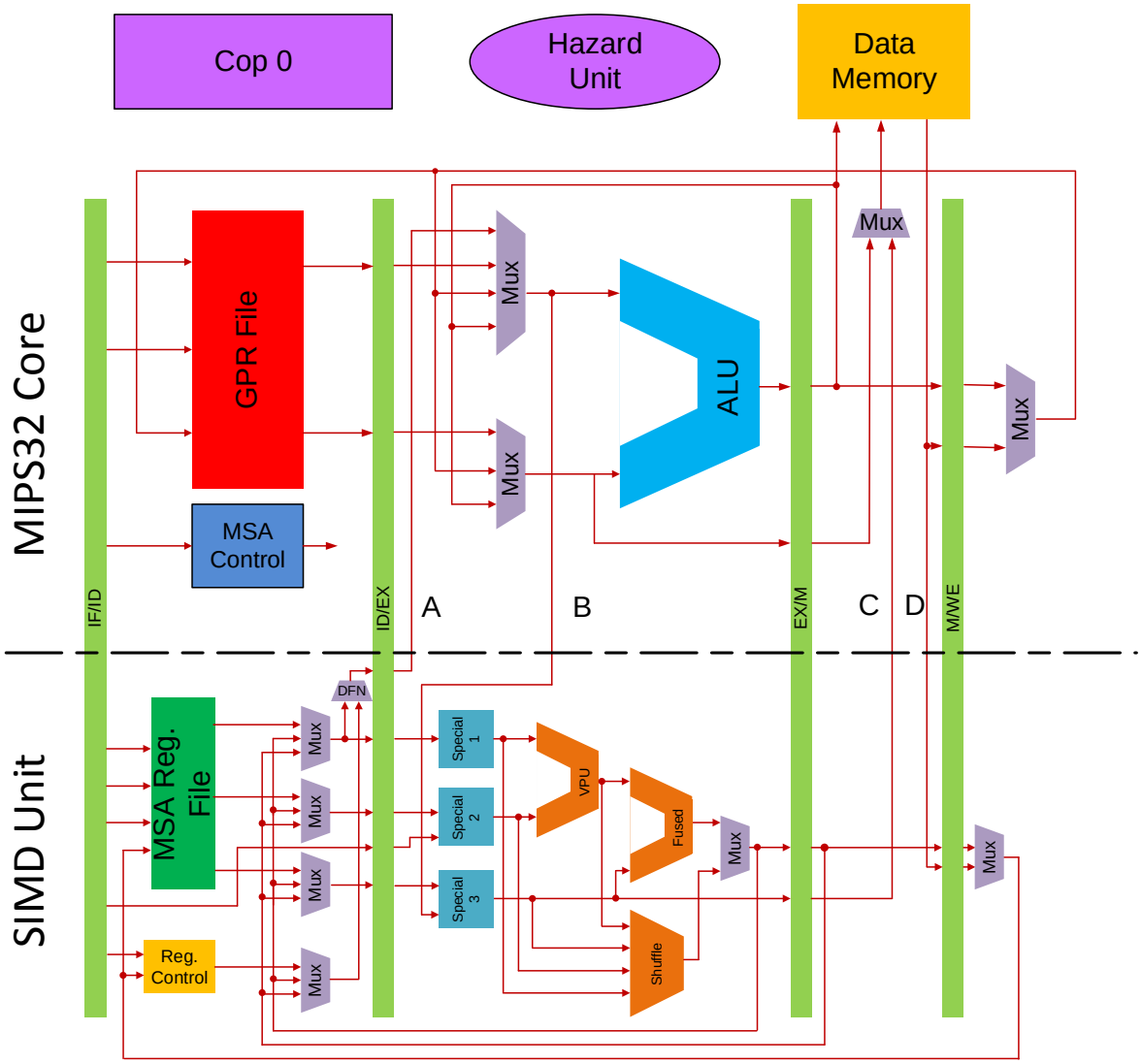


Figure 29: Interconnection between MIPS32 Core and MSA unit (control signals omitted)

7.1.2 Arithmetic

The arithmetic group contains instructions that perform basic arithmetic (additions, subtractions, and multiplications) on packed operands. This group also includes instructions that are used to perform high-level operations such as min/max, averaging, absolute values, and integer sign changes. All the arithmetic instructions support signed and unsigned integers unless otherwise noted.

Mnemonic	Type	Description	Implementation
ADD_A.df	3R	Vector Add Absolute Values	Figure 40
ADDS_A.df	3R	Vector Saturated Add of Absolute Values	Figure 40
ADDS_S.df	3R	Vector Signed Saturated Add of Signed Values	Figure 40
ADDS_U.df	3R	Vector Unsigned Saturated Add of Unsigned Values	Figure 40
ADDV.df	3R	Vector Add	Figure 40
ASUB_S.df	3R	Vector Absolute Values of Signed Subtract	Figure 40
ASUB_U.df	3R	Vector Absolute Values of Unsigned Subtract	Figure 40
AVE_S.df	3R	Vector Signed Average	Figure 40
AVE_U.df	3R	Vector Unsigned Average	Figure 40
AVER_S.df	3R	Vector Signed Average Rounded	Figure 40
AVER_U.df	3R	Vector Unsigned Average Rounded	Figure 40
DIV_S.df	3R	Vector Signed Divide	Figure 40
DIV_U.df	3R	Vector Unsigned Divide	Figure 40
DOTP_S.df	3R	Vector Signed Dot Product	Dot Product
DOTP_U.df	3R	Vector Unsigned Dot Product	Dot Product
DPADD_S.df	3R	Vector Signed Dot Product and Add	Dot Product
DPADD_U.df	3R	Vector Unsigned Dot Product and Add	Dot Product
DPSUB_S.df	3R	Vector Signed Dot Product and Subtract	Dot Product
DPSUB_U.df	3R	Vector Unsigned Dot Product and Subtract	Dot Product
HADD_S.df	3R	Vector Signed Horizontal Add	Figure 40
HADD_U.df	3R	Vector Unsigned Horizontal Add	Figure 40
HSUB_S.df	3R	Vector Signed Horizontal Subtract	Figure 40
HSUB_U.df	3R	Vector Unsigned Horizontal Subtract	Figure 40
MADDV.df	3R	Vector Multiply and Add	Figure 40
MAX_A.df	3R	Vector Maximum Based on Absolute Values	MAX MN unit
MAX_S.df	3R	Vector Signed Maximum	MAX unit
MAX_U.df	3R	Vector Unsigned Maximum	MAX unit
MIN_A.df	3R	Vector Minimum Based on Absolute Value	MAX MN unit
MIN_S.df	3R	Vector Signed Minimum	MIN unit
MIN_U.df	3R	Vector Unsigned Minimum	MIN unit
MOD_S.df	3R	Vector Signed Module	Figure 40

Mnemonic	Type	Description	Implementation
MOD_U.df	3R	Vector Unsigned Module	Figure 40
MSUBV.df	3R	Vector Multiply and Subtract	Figure 40
MULV.df	3R	Vector Multiply	Figure 40
SUBS_S.df	3R	Vector Signed Saturated Subtract of Signed Values	Figure 40
SUBS_U.df	3R	Vector Unsigned Saturated Subtract of Unsigned Values	Figure 40
SUBSUS_U.df	3R	Vector Unsigned Saturated Subtract of Signed from Unsigned	Figure 40
SUBSUU_S.df	3R	Vector Signed Saturated Subtract of Unsigned Values	Figure 40
SUBV.df	3R	Vector Subtract	Figure 40
SAT_S.df	BIT	Immediate Signed Saturate	SAT unit
SAT_U.df	BIT	Immediate Unsigned Saturate	SAT unit
ADDVI.df	I5	Immediate Add	Figure 40
MAXI_S.df	I5	Immediate Signed Maximum	MAX unit
MAXI_U.df	I5	Immediate Unsigned Maximum	MAX unit
MINI_S.df	I5	Immediate Signed Minimum	MIN unit
MINI_U.df	I5	Immediate Unsigned Minimum	MIN unit
SUBVI.df	I5	Immediate Subtract	Figure 40

Table 7: Arithmetic instructions in MSA

7.1.3 Comparison

The comparison group contains instructions that compare two packed operands element-by-element. The result of each comparison is saved to the corresponding position in the destination operand.

Mnemonic	Type	Description	Implementation
CEQ.df	3R	Vector Compare Equal	CEQ unit
CLE_S.df	3R	Vector Compare Signed Less Than or Equal	CLE unit
CLE_U.df	3R	Vector Compare Unsigned Less Than or Equal	CLE unit
CLT_S.df	3R	Vector Compare Signed Less Than	CLT unit
CLT_U.df	3R	Vector Compare Unsigned Less Than	CLT unit
BNZ.df	COP1	Immediate Branch If All Elements Are Not Zero	Branch unit

Mnemonic	Type	Description	Implementation
BNZ.V	COP1	Immediate Branch If Not Zero (At Least One Element of Any Format Is Not Zero)	Branch unit
BZ.df	COP1	Immediate Branch If At Least One Element Is Zero	Branch unit
BZ.V	COP1	Immediate Branch If Zero (All Elements of Any Format Are Zero)	Branch unit
CEQI.df	I5	Immediate Compare Equal	CEQ unit
CLEI_S.df	I5	Immediate Compare Signed Less Than or Equal	CLE unit
CLEI_U.df	I5	Immediate Compare Unsigned Less Than or Equal	CLE unit
CLTI_S.df	I5	Immediate Compare Signed Less Than	CLT unit
CLTI_U.df	I5	Immediate Compare Unsigned Less Than	CLT unit

Table 8: Comparison instructions in MSA

7.1.4 Logical and Shift

The logical and shift group contains instructions that perform bitwise logical operations. It also includes instructions that perform logical and arithmetic shift using the individual data elements of a packed operand.

Mnemonic	Type	Description	Implementation
NLOC.df	2R	Vector Leading Ones Count	Leading Ones/Zeros unit
NLZC.df	2R	Vector Leading Zeros Count	Leading Ones/Zeros unit
PCNT.df	2R	Vector Population Count	Population count
BCLR.df	3R	Vector Bit Clear	BIT unit
BINSL.df	3R	Vector Bit Insert Left	BINSL unit
BINSR.df	3R	Vector Bit Insert Right	BINSR unit
BNEG.df	3R	Vector Bit Negate	BIT unit
BSET.df	3R	Vector Bit Set	BIT unit
SLL.df	3R	Vector Shift Left	SLL unit
SRA.df	3R	Vector Shift Right Arithmetic	SRA unit
SRAR.df	3R	Vector Shift Right Arithmetic Rounded	SRAR
SRL.df	3R	Vector Shift Right Logical	SRL unit
SRLR.df	3R	Vector Shift Right Logical Rounded	SRLR unit
BCLRI.df	BIT	Immediate Bit Clear	BIT unit

Mnemonic	Type	Description	Implementation
BINSLI.df	BIT	Immediate Bit Insert Left	BINSL unit
BINSRI.df	BIT	Immediate Bit Insert Right	BINSR unit
BNEGI.df	BIT	Immediate Bit Negate	BIT unit
BSETI.df	BIT	Immediate Bit Set	BIT unit
SLLI.df	BIT	Immediate Shift Left	SRL unit
SRAI.df	BIT	Immediate Shift Right Arithmetic	SRA unit
SRARI.df	BIT	Immediate Shift Right Arithmetic	SRAR
SRLI.df	BIT	Immediate Shift Right Logical	SRL unit
SRLRI.df	BIT	Immediate Shift Right Logical Rounded	SRLR unit
ANDI.B	I8	Immediate Logical And	Vector Operations
BMNZI.B	I8	Immediate Bit Move If Not Zero	Vector Operations
BMZI.B	I8	Immediate Bit Move If Zero	Vector Operations
BSELI.B	I8	Immediate Bit Select	Vector Operations
NORI.B	I8	Immediate Logical Negated Or	Vector Operations
ORI.B	I8	Immediate Logical Or	Vector Operations
XORI.B	I8	Immediate Logical Exclusive Or	Vector Operations
AND.V	VEC	Vector Logical And	Vector Operations
BMNZ.V	VEC	Vector Bit Move If Not Zero	Vector Operations
BMZ.V	VEC	Vector Bit Move If Zero	Vector Operations
BSEL.V	VEC	Vector Bit Select	Vector Operations
NOR.V	VEC	Vector Logical Negated Or	Vector Operations
OR.V	VEC	Vector Logical Or	Vector Operations
XOR.V	VEC	Vector Logical Exclusive Or	Vector Operations

Table 9: Logical and shift instructions in MSA

Unpack and Shuffle

The unpack and shuffle group contains instructions that interleave (unpack) the data elements of a packed operand. It also contains instructions that can be used to reorder (shuffle) the data elements of a packed operand.

Mnemonic	Type	Description	Implementation
ILVEV.df	3R	Vector Interleave Even	ILVEV unit
ILVL.df	3R	Vector Interleave Left	ILVL unit
ILVOD.df	3R	Vector Interleave Odd	ILVOD unit

Mnemonic	Type	Description	Implementation
ILVR.df	3R	Vector Interleave Right	ILVR unit
PCKEV.df	3R	Vector Pack Even	PCKEV unit
PCKOD.df	3R	Vector Pack Odd	PCKOD unit
SLD.df	3R	GPR Columns Slide	SLD unit
VSHF.df	3R	Vector Data Preserving Shuffle	VSHF unit
SLDI.df	ELM	Immediate Columns Slide	SLD unit
SHF.df	I8	Immediate Set Shuffle Elements	SHF unit

7.1.5 Insertion and Extraction

The insertion and extraction group contains instructions that are used to insert or extract elements in a MSA register.

Mnemonic	Type	Description	Implementation
SPLAT.df	3R	GPR Element Splat	SPLAT unit
INSVE.df	ELM	Element Insert Element	INSVE unit
SPLATI.df	ELM	Immediate Element Splat	SPLAT unit

Table 10: Insertion and extraction instructions in MSA

8 Architecture Implementation

8.1 Overview

Based on the objective this implementation should be as simple as possible. For example, we have implemented an in-order scalar microarchitecture because it is simpler than implementing superscalar or OoO. Pipelining is a well understand technique to improve performance. It has been part of computer architecture lectures for years, even from basic levels (15). Since SIMD unit cannot work with a processor we needed to found one.

This SIMD unit was developed to work as a coprocessor, to easyly change in the future the main core. Unfortunately we could not find a MIPS32 or MIPS64 Release 5 core implementation. So, we decided to use a MIPS32 soft-core processor called mips32r1_xum from opencores.org (53). It is an open source repository focused on hardware Intellectual property (IP).

We upgraded the MIPS32 core with the minimal features from Release 6 needed by the MIPS SIMD Architecture.

8.2 MIPS32 core

MIPS32 core mips32r1_xum is a 5-stage pipeline single-issue in-order processor. Unfortunately it only supports MIPS32 Release 1 ISA while the SMID unit has been developed based on MIPS64 Release 6[19], so is necessary at least a core running MIPS32

Release 5. In consequence, it has been necessary to make some modifications to the core. We have been added the minimum MIPS32 Release 6 requirements to use the SIMD unit. Furthermore, the Exception Handler (Cop 0) was upgraded to support MIPS SIMD Architecture ISA. Finally, we have has been added support for unaligned memory operations as required by the SIMD unit at the memory stage. Figure 30 shows the new paths created and the MSA Control unit that control them.

Table 11 shows the 11 instructions added to the MIPS32 core. Decode and execution of these instructions is done on both the MIPS32 core and the SIMD unit. Besides, execution is synchronized at the stage level, thus whenever is needed the core and SIMD side exchange data or control signals in both directions. Even though MIPS32 core and SIMD unit are separated designs for these 11 instructions both units cooperate as if they were one. Figure 29 showed both as big one unit.

Finally mips32r1_xum core does not have a branch predictor since this design does not need one. It is because it has a sort pipeline of 5-stages and branch instructions have a branch delay slot (54). This means that meanwhile a branch is resolved another instruction is executed avoiding to stall the pipeline.

Both the main core and the SIMD unit have a 5-stage pipeline and their execution is aligned. Control signals are shared between control units. When any of the pipelines needs to stall at any stage the other pipeline stalls too, keeping instructions execution between pipelines.

As a summary of the mips32r1_xum characteristics, we can list:

- In-order single issue
- Five-stage pipelining
- MIPS32 Release1 (partial upgrade to Release 6)

[19] In fact supports microMIPS32, microMIPS64, MIPS32 and MIPS64 for Releases 5 and 6

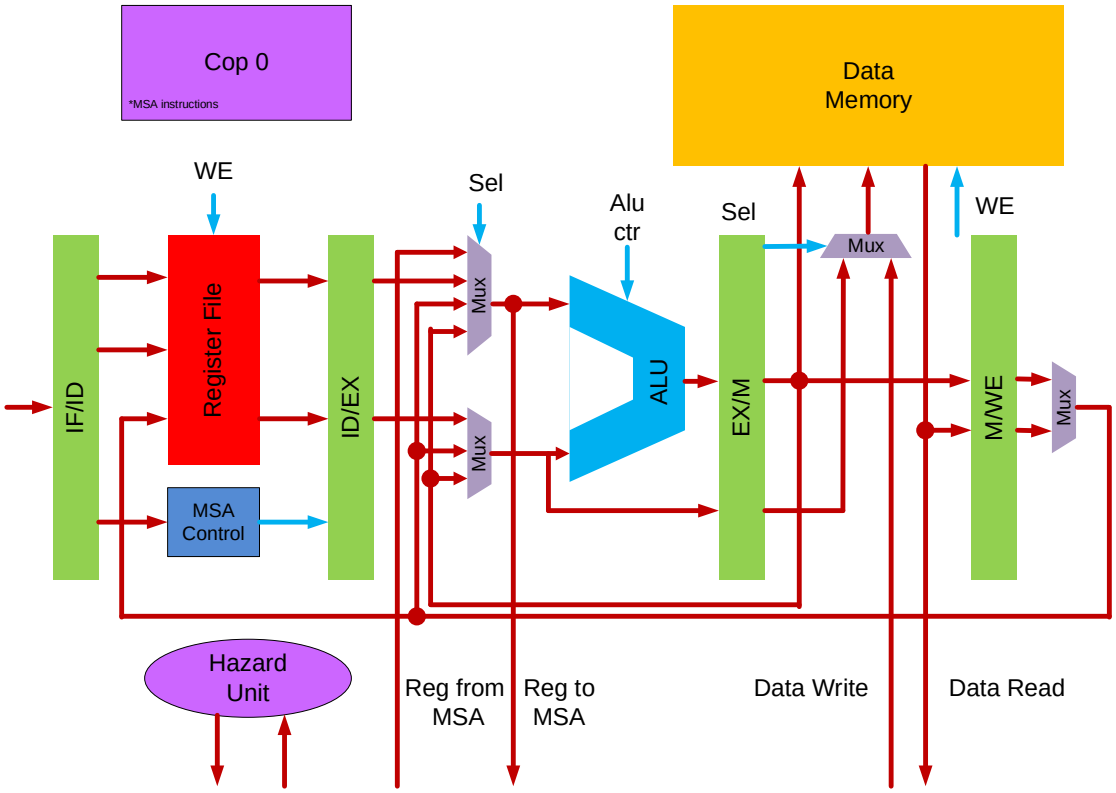


Figure 30: Modifications made to the main core to attach SIMD unit

Instruction	Type	Purpose
LD.df	MI10	Vector Load
ST.df	MI10	Vector Store
CTCMSA	ELM	GPR Copy to MSA Control Register
CFCMSA	ELM	GPR Copy from MSA Control Register
COPY_S.df	ELM	Copy from MSA to GPR Signed
COPY_U.df	ELM	Copy from MSA to GPR Unsigned
FILL.df	2R	Vector Fill from GPR
BNZ.V	COP1	Immediate Branch If Not Zero (At Least One Element of Any Format Is Not Zero)
BNZ.df	COP1	Immediate Branch If All Elements Are Not Zero
BZ.df	COP1	Immediate Branch If At Least One Element Is Zero
BZ.V	COP1	Immediate Branch If Zero (All Elements of Any Format Are Zero)

Table 11: Instructions introduced to MIPS32 core to support SIMD unit.

8.3 Fetch

Figure 31 shows the interconnection between the SIMD coprocessor and the main core. The main core executes all instructions that are not SIMD. It also manages the SIMD unit, performing fetch and decode of the instructions and accessing memory.

SIMD unit has a decoder. It is connected in parallel with the decoder of the MIPS core and snoops fetched instructions from the main core pipeline. When it detects MSA instructions it decodes them and SIMD unit starts to work. Otherwise SIMD unit just executes NOP operations. On the other hand, the main core also decodes SIMD instructions when these are issued to the pipeline and is able to determine whether they should be kept in the main pipeline in order to execute a specific task. These are instructions[20] that require some intervention of the main core such memory operations, which involve address calculation, branches and instructions that access the general purpose register file. Oher SIMD operations are interpreted as NOP operations by the main core. Figure 31 shows the interconnection between the MIPS32 core and the SIMD unit at IF/DC stage.

[20] Data transfer instructions

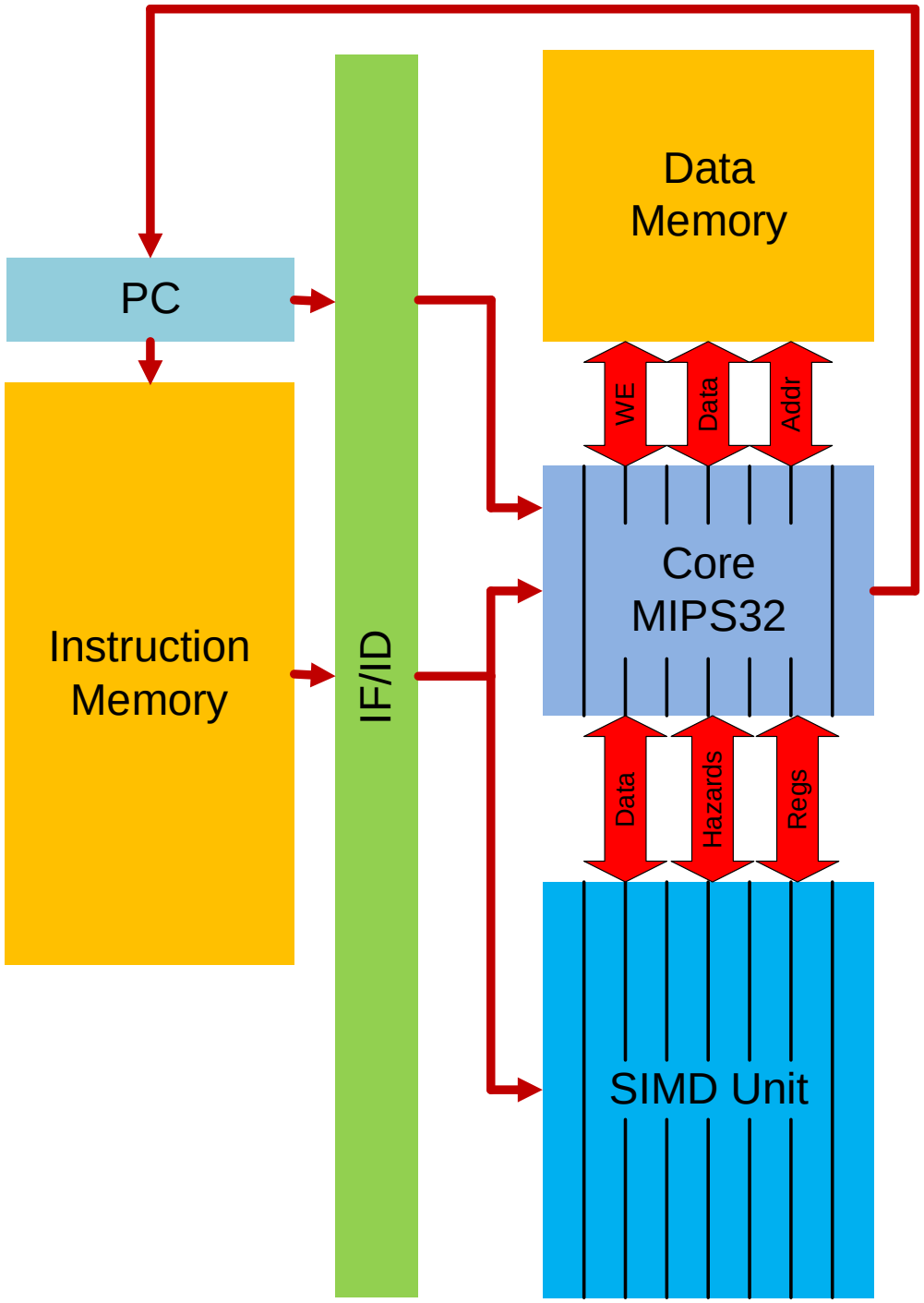


Figure 31: Interconnection between core and coprocessor

8.4 Decode

SIMD unit has its own decode logic, and is able to generate all the necessary signals to activate and control all the lanes. Figure 32 shows the main signals generated by SIMD decode. Instruction recodification is done using the layout shown in Table 3.

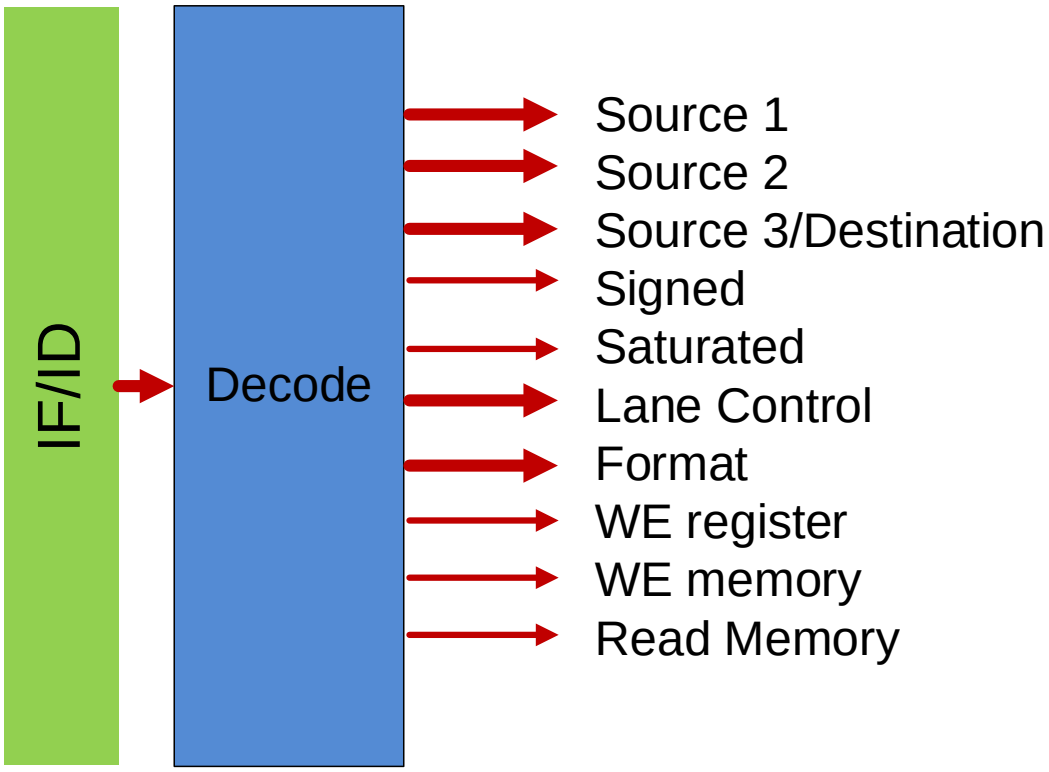


Figure 32: Signals generated by SIMD decode

8.5 Register File

The SIMD unit has a private register file. Each register stores a vector of 128-bit wide. There are 32 registers. Every vector register can be interpreted according to four formats: *byte* (8-bit), *halfword* (16-bit), *word* (32-bit), *doubleword* (64-bit). It depends of the instruction field *df*. There is no way to know the data format of a vector unless associated to a specific instruction. Programmer is responsible of keeping the semantic of the application.

Corresponding to the associated data format, a vector register consists of a number of elements indexed from 0 to n-1. Figure 6 shows the vector register layout for elements of all four data formats. Element 0 is always in the less significant part of the vector register.

MSA vectors are stored in memory starting from the 0th element at the lowest byte address. The byte order of each element can be little-endian or big-endian depending on the implementation. The scalar floating-point unit (FPU) registers are mapped to the MSA vector registers.

MSA has operations that uses up to 3 source operands and only writes up to one register. In order to implement a register file with these features a memory with 3 read ports and 1 write port is required. Figure 33 shows how the MSA register file should be.

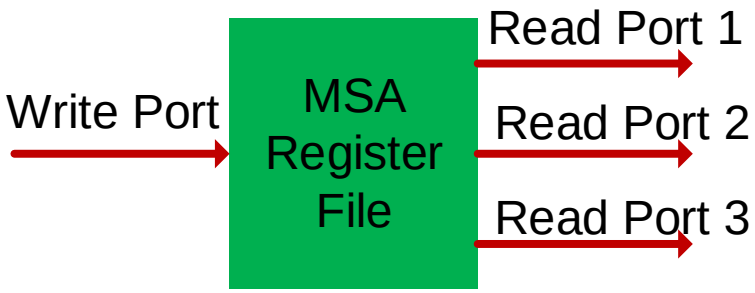


Figure 33: Representation of the MSA register file

Considering that the implementations is intended to run on FPGA, instead of building a multiported memory using logical elements, we have used the memory elements provided by the FPGA. These memories have two ports that can be used as read or write ports. The design contemplates one as a read port and the one as a write port and we implement three copies of the register file, in order to provide the 3 required read ports. Figure 34 shows how the memories ports are connected. The write port all memory are connected in parallel, so all memories will have a copy of all values written.

Using this three memory block in parallel we avoid to use LEs. An implementation of a register file 3R1W[21] built as a memory declaration statement in Verilog, will generates a costly circuit that requires a lot of LEs. Moreover, FPGA memory elements have some interesting features such as initialization at compile time and visualization at run time using Quartus II.

[21] 3R1W: 3 read ports, 1 write port.

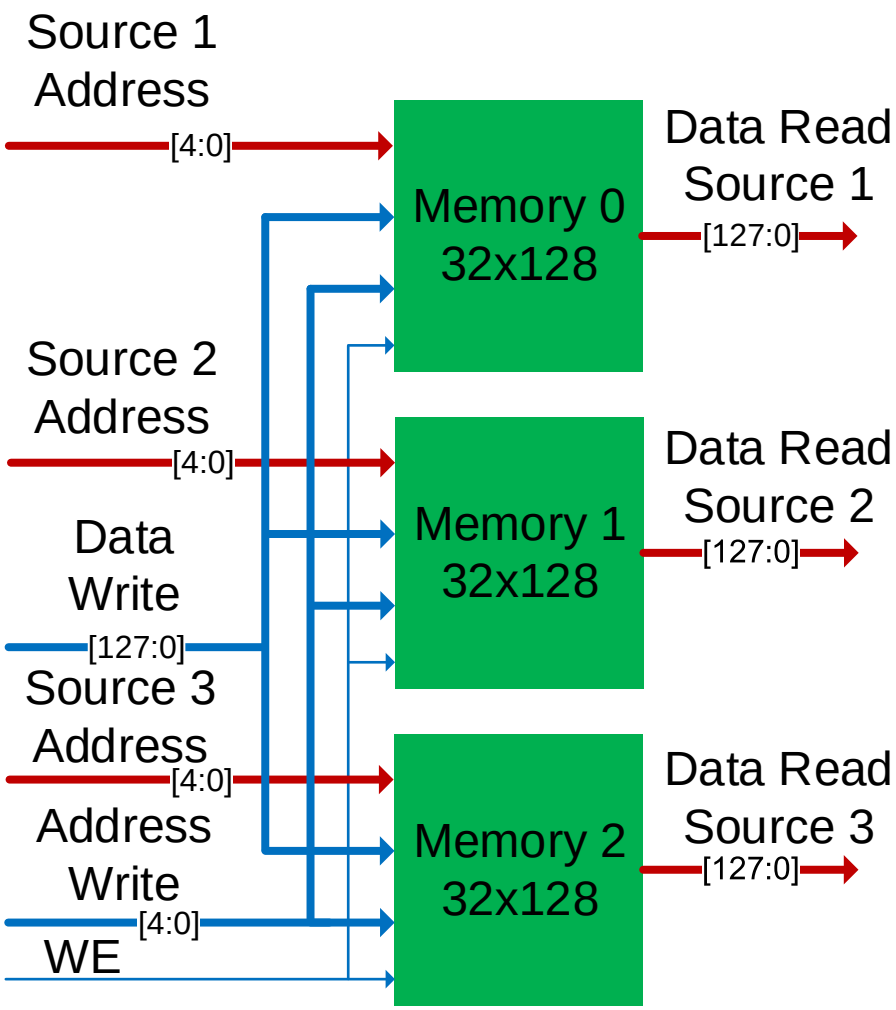


Figure 34: Implementation of the MSA register file

9 SIMD execution stage

The SIMD unit is composed of several individual sub-units. Figure 35 shows all sub-units of the SIMD unit. The Vector Processing Unit (VPU) performs integer and bit-wise logical operations, There are some lanes that have three source operands to support fused operations. The Shuffle unit performs element permutation operations. There are three special units that are used to transform or adapt some values into vector representation and also to transform from one specific vector format to another one. In this chapter we describe these units.

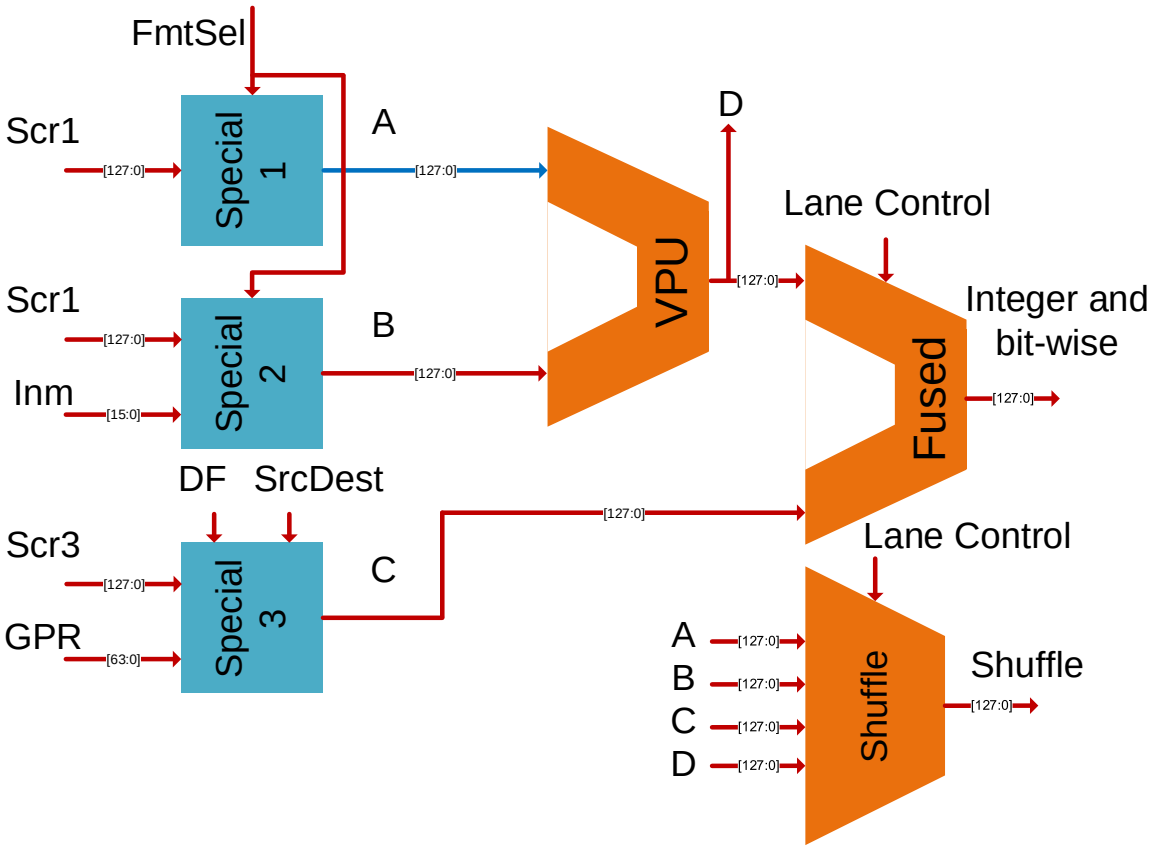


Figure 35: SIMD execution stage

9.1 Vector Processing Unit

A Vector Processing Unit (VPU) executes vector operations over vector registers. This operations can be integer, bit-wise logical or even Floating Point operations. Each VPU is divided into identical elements called lanes. A lane is a basic building block unit of a VPU. A lane is similar to an ALU included in a scalar processor, with the important difference that all lanes work in a lock-stepped fashion.

To achieve the maximum parallel performance there must be a lane for each element. For instance, to perform a SIMD add over two vector registers of 4 elements in one single cycle there must have 4 lanes, one to perform each operation. But it is not, to save area we could implement the SIMD unit by utilizing from 1 up to 4 lanes. For instance, if we wanted to perform again the 4 element SIMD operation but using 2 lanes, it would require twice the time. Figure 36 shows an implementation using the same number of lanes than elements to process (left) and other that has half the lanes than elements to process.

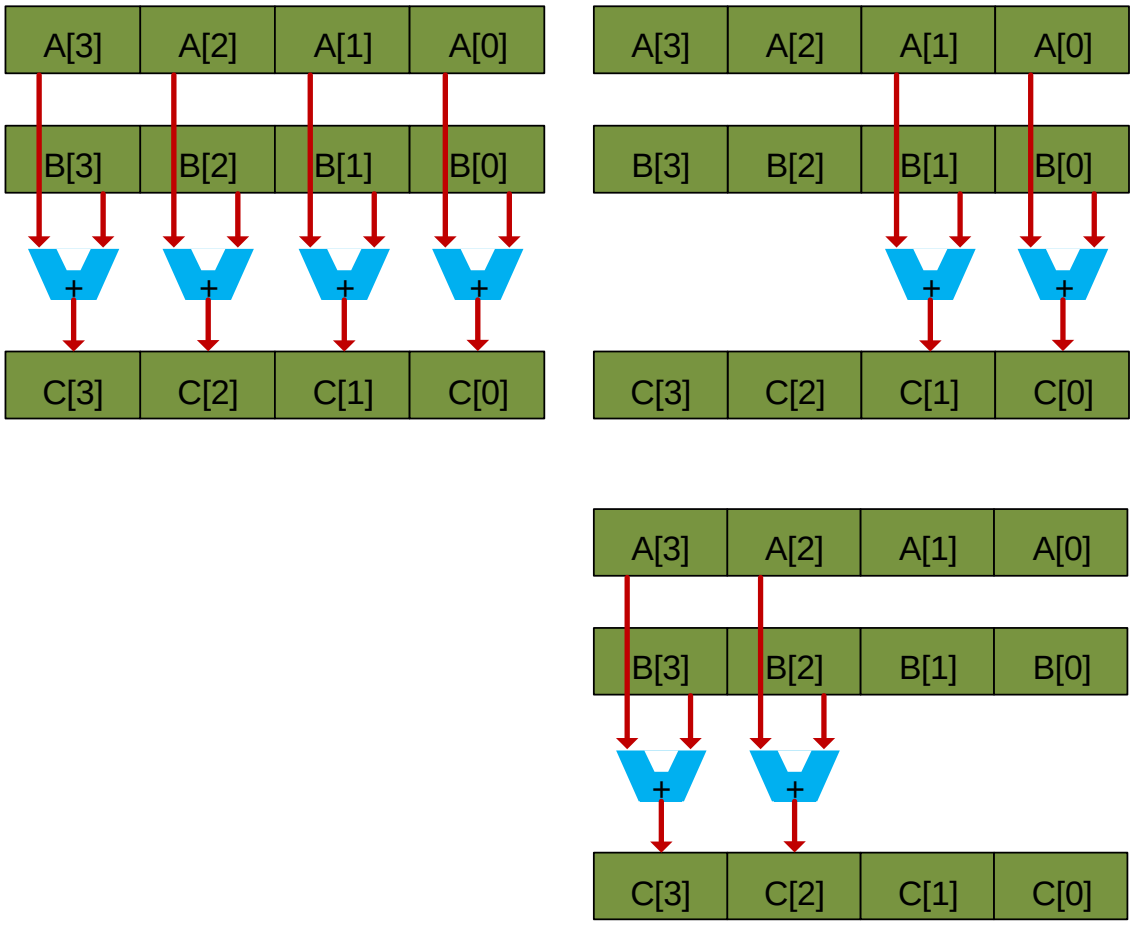


Figure 36: Four lanes (left) vs two lanes (right)

The VPU has to deal with four data-formats, it means that we have four width of elements and therefore lanes. There are two options here, design lanes for each of the four width or design lanes that can joint (sub-lanes) to perform bigger operations. For instance, joining two 8-bit lanes to perform 16-bit operations or four 8-bit lane to perform 32-bit operation. Using sub-lanes reduces area consumption but increases the critical path and the complexity of the design and coding, because extra control logic is needed. On the other hand, using individual lanes only requires to generate one parametrized lane, and the compiler will generate all four formats. Future work should be find a middle point.

9.2 Multipurpose adder lane

The circuit shown in Figure 37 allows the SIMD unit to calculate 22 different operations. These operations are shown in Table 12 and all of them are variations of addition and subtraction operations. Some operations use a secondary unit, that can be unit special 1, 2 or 3 to transform data before computation. For instance, to fill a vector B with as many copies of an immediate value as elements in the vector.

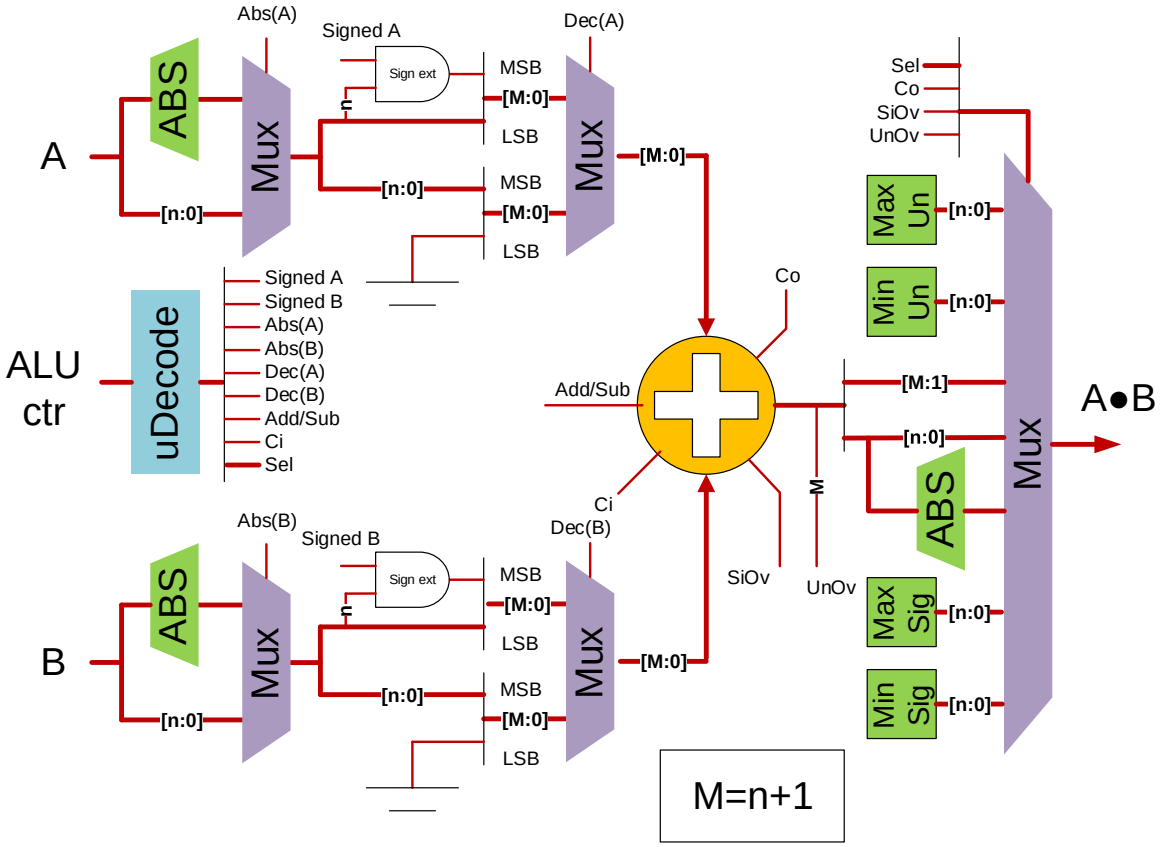


Figure 37: Multipurpose adder lane

Input Signals			Control Signals									
Signed	Saturated	Operation	Signed A	Signed B	Add/Sub	Ci	Abs(A)	Dec(A)	Abs(B)	Dec(B)	Description	Instruction
0	0	0001	0	0	1	0	0	0	0	0	A+u5	ADDVI.df
0	0	0010	0	0	0	0	0	0	0	0	A-u5	SUBVI.df
0	0	0001	0	0	1	0	0	0	0	0	A+B	ADDV.df
0	0	0010	0	0	0	0	0	0	0	0	A-B	SUBV.df
0	0	0110	0	0	1	0	1	0	1	0	abs(A)+abs(B)	ADD_A.df

1	1	0110	1	1	1	0	1	0	1	0	sat(abs(A)+abs(B))	ADDS_A.df
1	1	0001	1	1	1	0	0	0	0	0	sat(sig(A)+sig(B))	ADDS_S.df
0	1	0001	0	0	1	0	0	1	0	1	sat(A+B)	ADDS_U.df
1	0	0111	1	1	1	0	0	1	0	1	(sig(A)+sig(B))/2	AVE_S.df
0	0	0111	0	0	1	0	0	1	0	1	(A+B)/2	AVE_U.df
1	0	1000	1	1	1	1	0	1	0	1	(sig(A)+sig(B) +1)/2	AVER_S.df
0	0	1000	0	0	1	1	0	1	0	1	(A+B+1)/2	AVER_U.df
1	0	0001	1	1	1	0	0	0	0	0	sig(A)+sig(B)	HADD_S.df
0	0	0001	0	0	1	0	0	1	0	1	A+B	HADD_U.df
1	0	0010	1	1	0	0	0	0	0	0	sig(A)-sig(B)	HSUB_S.df
0	0	0010	0	0	0	0	0	1	0	1	A-B	HSUB_U.df
1	1	0010	1	1	0	0	0	0	0	0	sat(sig(A)-sig(B))	SUBS_S.df
0	1	0010	0	0	0	0	0	1	0	1	sat(A-B)	SUBS_U.df
0	0	1011	0	1	0	0	0	1	0	1	sat(A-sig(B))	SUBSUS_U.df
0	0	1100	0	0	0	0	0	1	0	1	sat(sig(A-B))	SUBSUU_S.df
1	0	1101	1	1	0	0	0	1	0	1	abs(sig(A)-sig(B))	ASUB_S.df
0	0	1101	0	0	0	0	0	1	0	1	abs(A-B)	ASUB_U.df

Table 12: Instructions that uses multipurpose adder lane.

9.3 Multiplier lane

Circuit show in Figure 38 performs a multiplication between two elements. The most significant half of the multiplication result is discarded. The multiplier is implemented using dedicated hardware from the FPGA instead of building them using only logical elements, leveraging that dedicated hardware multipliers are faster than those implemented with only logic elements. The implementation details from the multiplier are discussed in section 6.2. The multiplication result can be added or subtracted from a third operand to execute fused operations. Table 13 shows operations that can be executed using this circuit.

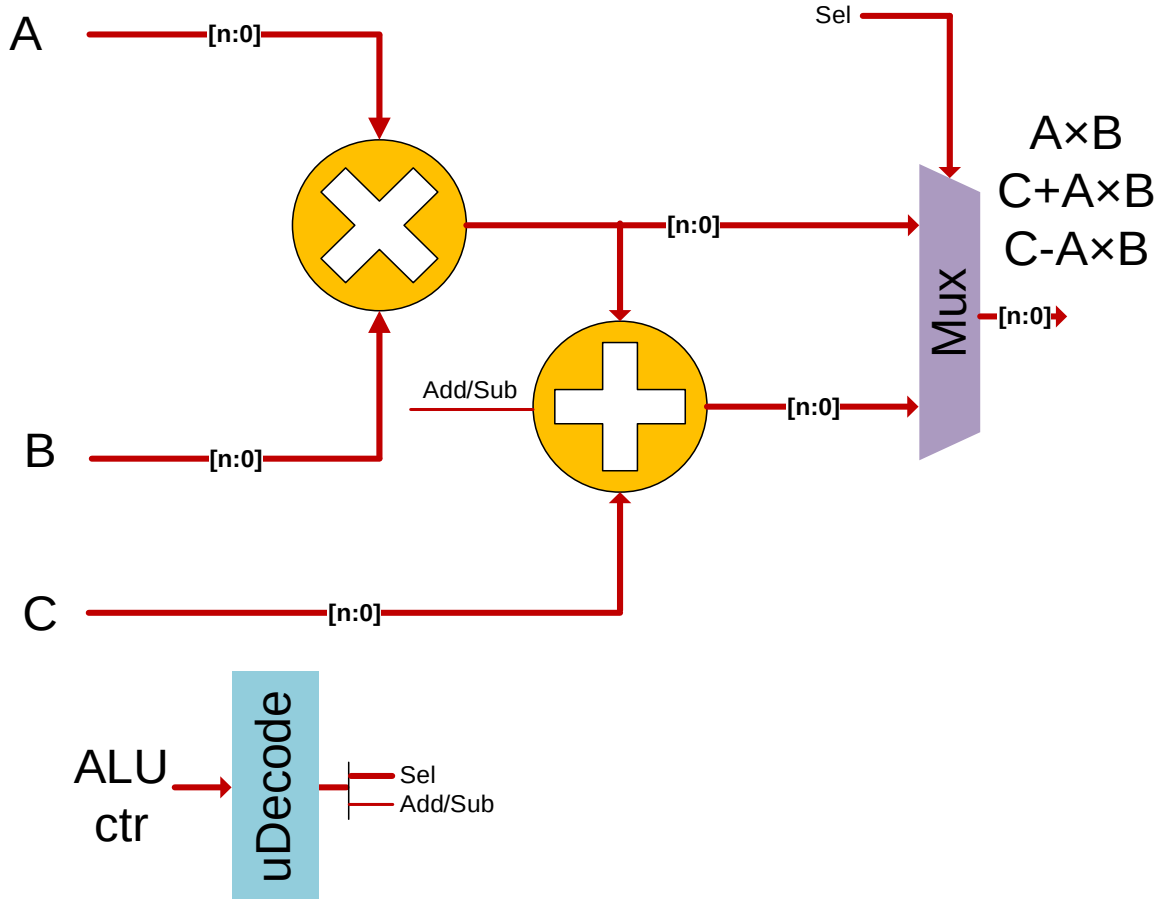


Figure 38: Multiplication circuit and fused multiplication

Input Signals			Control				
Signed	Saturated	Op	Add/Sub	Sel	Description	Instruction	Purpose
0	0	0011	0	0	A *B	MULV.df	Vector Multiply
0	0	1001	1	1	C+A*B	MADDV.df	Vector Multiply and Add
0	0	1010	0	1	C-A*B	MSUBV.df	Vector Multiply and Subtract

Table 13: Instructions that use Multiplication circuit

9.4 Divider circuit

Figure 39 shows the divider circuit and Table 14 shows the instructions that are executed on this circuit. Divider circuit is implemented using the LPM modules provided by Altera. LPM-divider is a pipelined operation in which each operation (division and module) requires 4 cycles.

Therefore, division is the operation that requires the highest execution time. Moreover, the division module is the most expensive in terms of area, due to the logic elements necessaries to its implementation. For instance a 64-bit divider uses about 6k logical elements. That is approximately the same amount of elements that uses the entire MIPS32 core. Implementation details are discussed in the section 9.4.

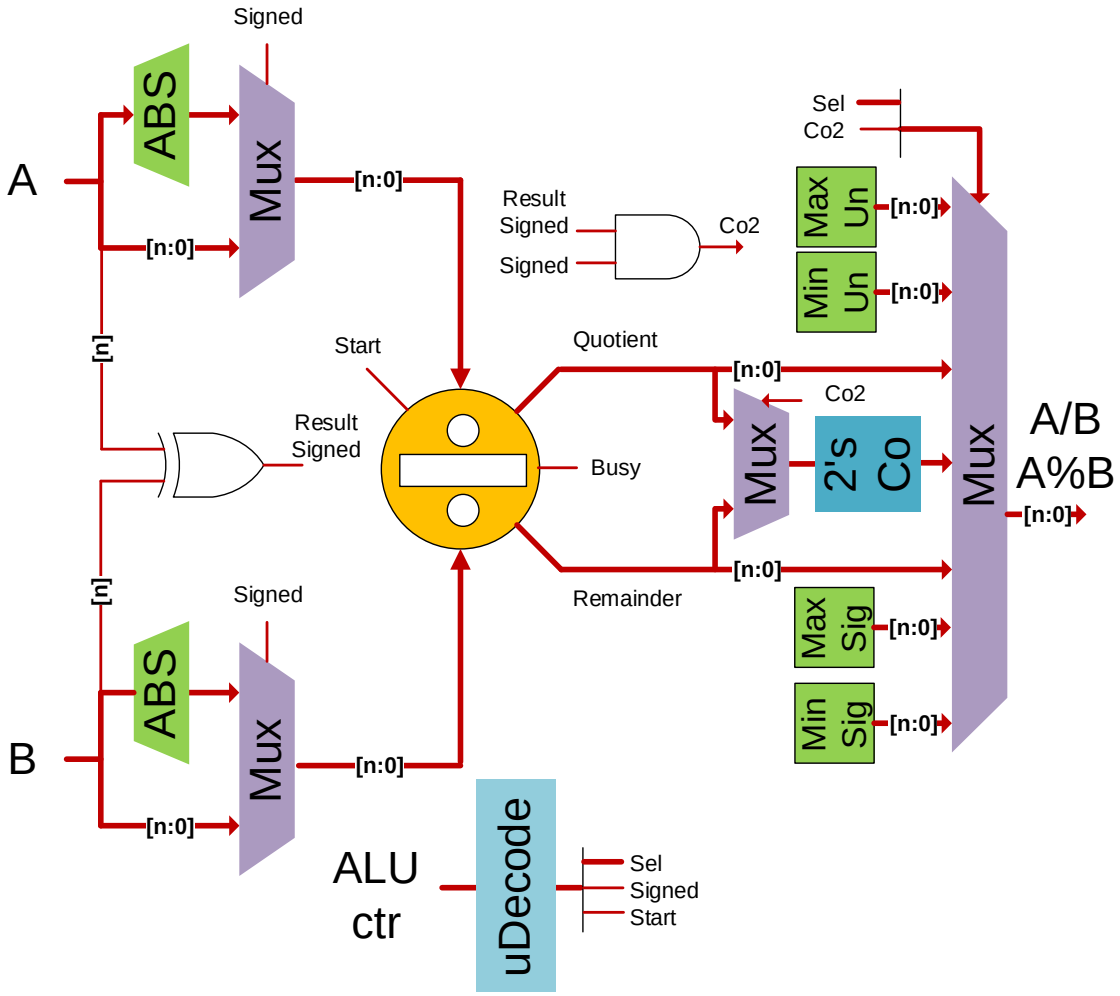


Figure 39: Divider circuit

Input Signals					
Signed	Saturated	Op	Description	Instruction	Purpose
1	0	1100	sig(A)/sig(B)	DIV_S.df	Vector Signed Divide
0	0	1100	A/B	DIV_U.df	Vector Unsigned Divide
1	0	0101	sig(A)%sig(B)	MOD_S.df	Vector Signed Module
0	0	0101	A%B	MOD_U.df	Vector Unsigned Module

Table 14: Instructions that uses divided circuit

Figure 40 shows the 3R lane that is composed by the circuits described above, multipurpose adder, multiplier and divider. These circuits shares some resources: Two input multiplexors, used to calculate absolute values; and the output multiplexor, used to choose the final result. To achieve the maximum data level parallelism we decided to implement one lane per element to process. By doing this we can process all data formats using the same time.

There are 30 of these 3R lanes inside the VPU.

- 16 lanes of 8-bit wide to calculate *Byte* format operations
- 8 lanes of 16-bit wide to calculate *Halfword* format operations
- 4 lanes of 32-bit wide to calculate *Word* format operations
- 2 lanes of 64-bit wide to calculate *Doubleword* operations

Figure 61 (in the Annexes) shows how the vector result is reconstructed in the different available formats: *byte*, *halfword*, *word* and *doubleword* using the results of 3R lanes.

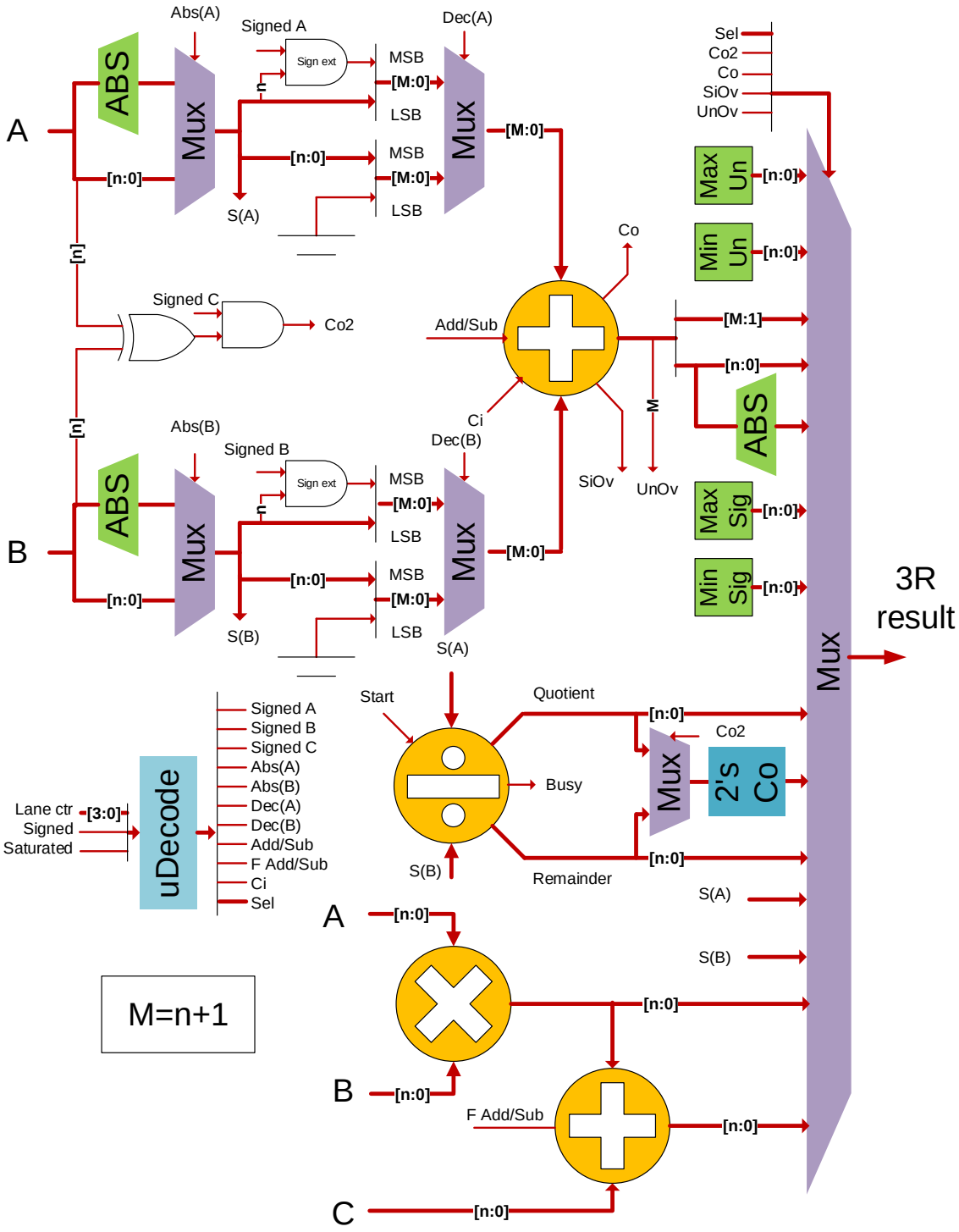


Figure 40: 3R Lane, multiplier, divider, adder and substrate multipurpose

9.5 Special unit 1

Figure 41 shows the special unit 1 schematically. This unit transforms between some vectors formats, including:

- *Byte* to *Halfword*
- *Halfword* to *Word*
- *Word* to *Doubleword*

Each transformation only takes odd or even values from the original format and depending on the flag *signed* a sign extend operation is performed. Figure 62 and Figure 63 shows the implementation of the special unit 1. Figure 64 shows the selector for the result in the desired format. The result of the special unit 1 is sent to the *VPU* or the *Shuffle* unit.

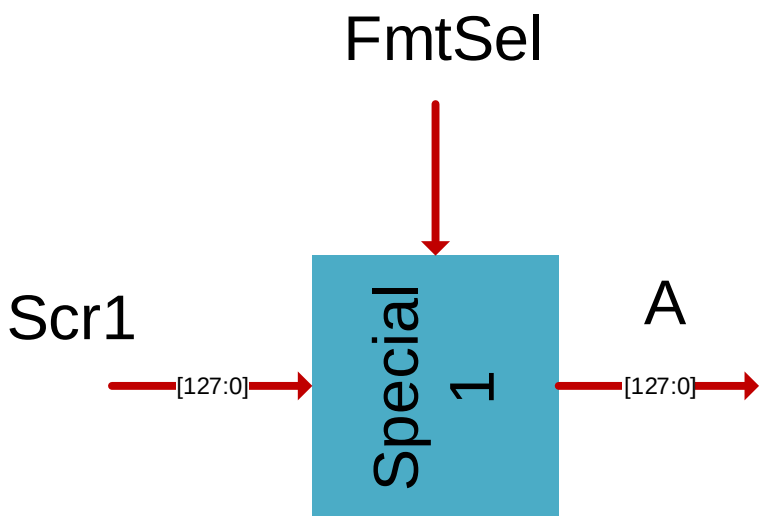


Figure 41: Special unit 1

9.6 Special unit 2

Figure 42 shows the special unit 2 schematically. This unit is similar to special unit 1. Circuits shown in Figure 62, Figure 63 and Figure 65 compose the special unit 2. One extra feature is that here Immediate vectors are created from the immediate value taken from the instruction. Figure 66 shows the multiplexor that chooses between of the possible results. Note that Even/Odd transformations are mismatch between the special unit 1 and the special unit 2.

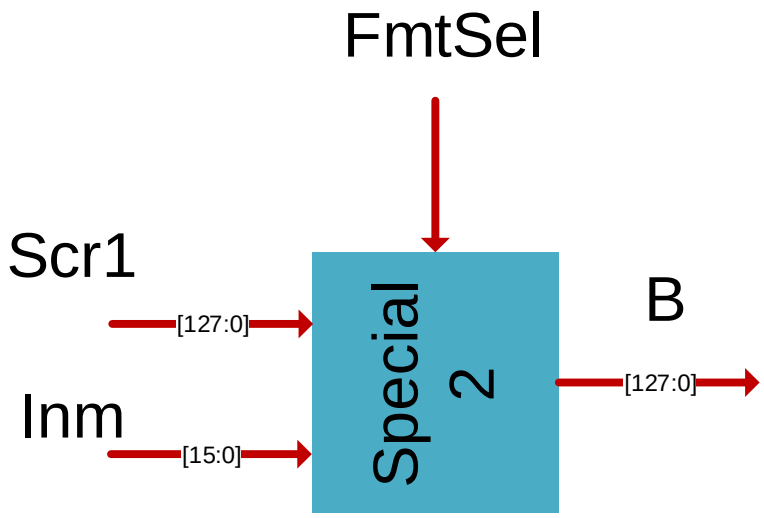


Figure 42: Special unit 2

9.7 Special unit 3

Figure 43 shows the special unit 3. The main purpose of this unit is to choose between source operand 3 from the vector register file or the value read from the general purpose register. If the later is selected, these value is truncated and replicated to create all 4 vector formats. Figure 67 shows the implementation of the special unit 3, while Figure 68 shows the multiplexor that chooses the final result of this unit.

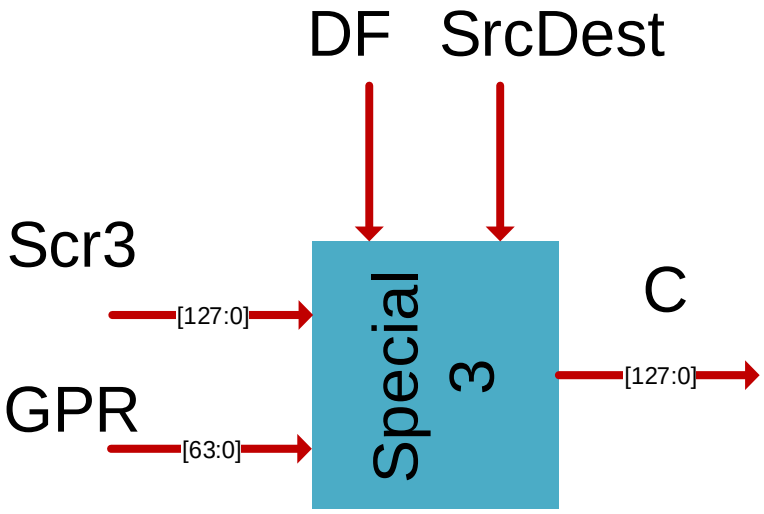


Figure 43: Special unit 3

10 Memory

MIPS processors are designed under Harvard architecture philosophy, this means that instructions and data are stored in separate memories. As a result, the pipeline design has to consider an instruction cache and a data cache. The minimum storage capacity (granularity) is a cache line, which can have just a few bytes, typically 64-bytes. MIPS release prior to Release 5 support only aligned memory accesses. Release 5 of the architecture supports 128-bit memory accesses without natural alignment.

Considering what has been described above, as well as the fact that MIPS SIMD Architecture only runs on Release 5 or higher, we have implemented these memories according with Release 5 requirements, such as load/store of 128-bit size and unaligned memory accesses. Moreover, these memories are designed to use memory elements from the FPGA, so they are not caches strictly. Figure 44 shows schematically the data memory and the instruction memory. The size of the memories depends on the width of the address (“n” and “m”). Data memory reads 16 bytes and can write 1, 2, 4, 8 or 16 bytes at the same time.

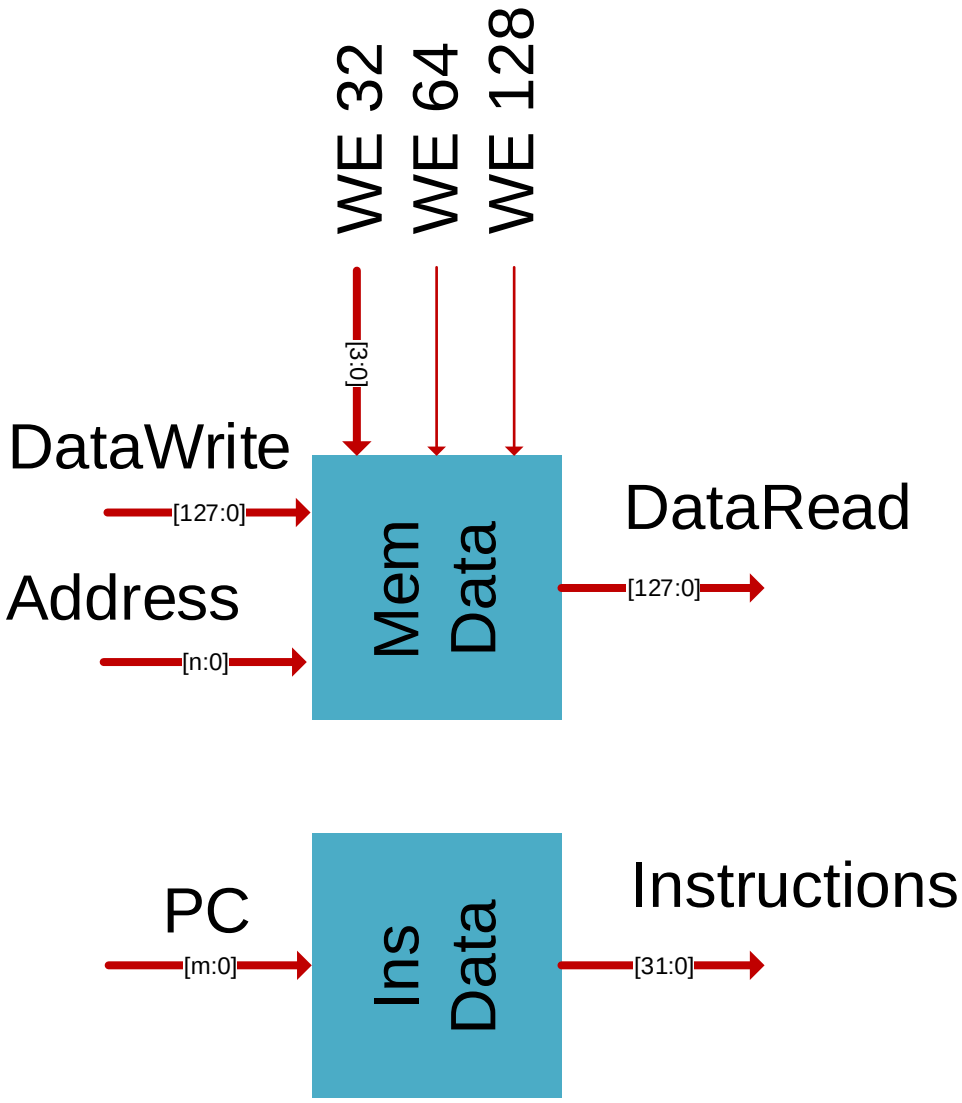


Figure 44: Instruction and data memory

10.1 Instruction Memory

This memory stores all the instructions that are going to be executed by the processor including those that should be executed by the SIMD unit. Since each instruction is composed by 4 bytes (32-bits) each memory row stores a whole instruction. This memory has 128k rows of 4-byte wide, with a total capacity of 512 kB. Instruction memory is implemented using FPGA's memory elements in order to save logic elements and get better operational frequency. Finally, this memory is initialized using hexadecimal files specified at compile time.

10.2 Data Memory

This memory stores all the data used by the processor and the SIMD unit. The data memory reads 16 bytes (128-bits) at the same time to provide data to read operations of all sizes. Additionally, this memory can write 1, 2, 3, 4, 8 or 16 bytes at the same time. It is used by nibble-store operations, half, word and double word store operations. One important feature is that all read and write operations do not require natural alignment at the address level.

To provide unaligned support it has been decided to use 16 individual memories (cells) where each store a byte. Based on the address that is set by the processor to read or write it is necessary to reorder the cell's contents. Figure 46 and Figure 47 show the mechanism to rearrange memory access.

Figure 48 shows the data path to the memory cells. The Write Enable (WE) signals of each memory cell are also rearranged. The 4 less significant bits from the address are used to calculate the positions of a given byte to a given cell. The remainder bits are used to select the row of each memory cell. The constant 4-bits adders are used to allow read or write operations from segments of two adjacent memory rows to create a 128-bits row.

Figure 45 shows an example of row and memory calculation. Suppose that the processor needs to read address 3226. It is divided into the lower 4 bits (10) and the remaining upper bits (201). Then using the IDs of each memory cell given by its position (15 to 0) a 4 bit offset is calculated. This offset allows to jump to the next row when it is needed.

Address	[n:4]	[n:0]
3226	201	10

Address Rows	Memory Cells															
	15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
200	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
201	F	E	D	C	B	A	X	X	X	X	X	X	X	X	X	X
202	X	X	X	X	X	X	P	O	N	M	L	K	J	I	H	G
203	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Data	F	E	D	C	B	A	P	O	N	M	L	K	J	I	H	G
Cell ID	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Offset 0-Addr[3:0]	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6
Result Addr+Offset	3231	3230	3229	3228	3227	3226	3241	3240	3239	3238	3237	3236	3235	3234	3233	3232
Row Result[n:4]	201	201	201	201	201	201	202	202	202	202	202	202	202	202	202	202

Figure 45: Example of row and memory cell calculation

Once all the 16 bytes of a row (or rows) have been read, we need to rearrange the data using the 4 lower bits from the address and the left multiplexor shown in Figure 46. Finally, we send the data back to the processor. Depending on address, data could be fit in one memory row if it matches with the natural alignment or it could be splitted in two rows (unaligned).

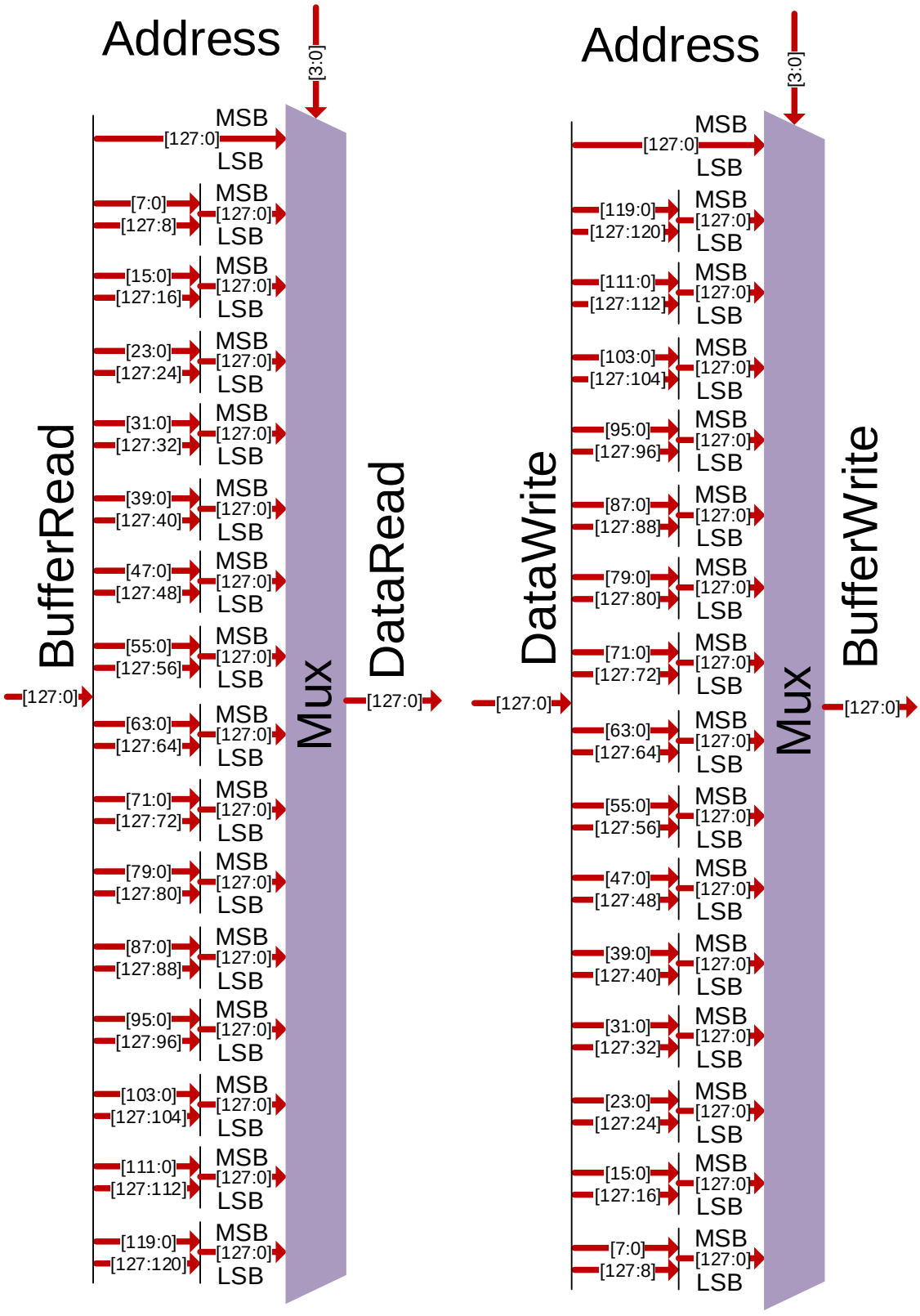


Figure 46: Multiplexors used to rearrange the data to load and store it into the data memory

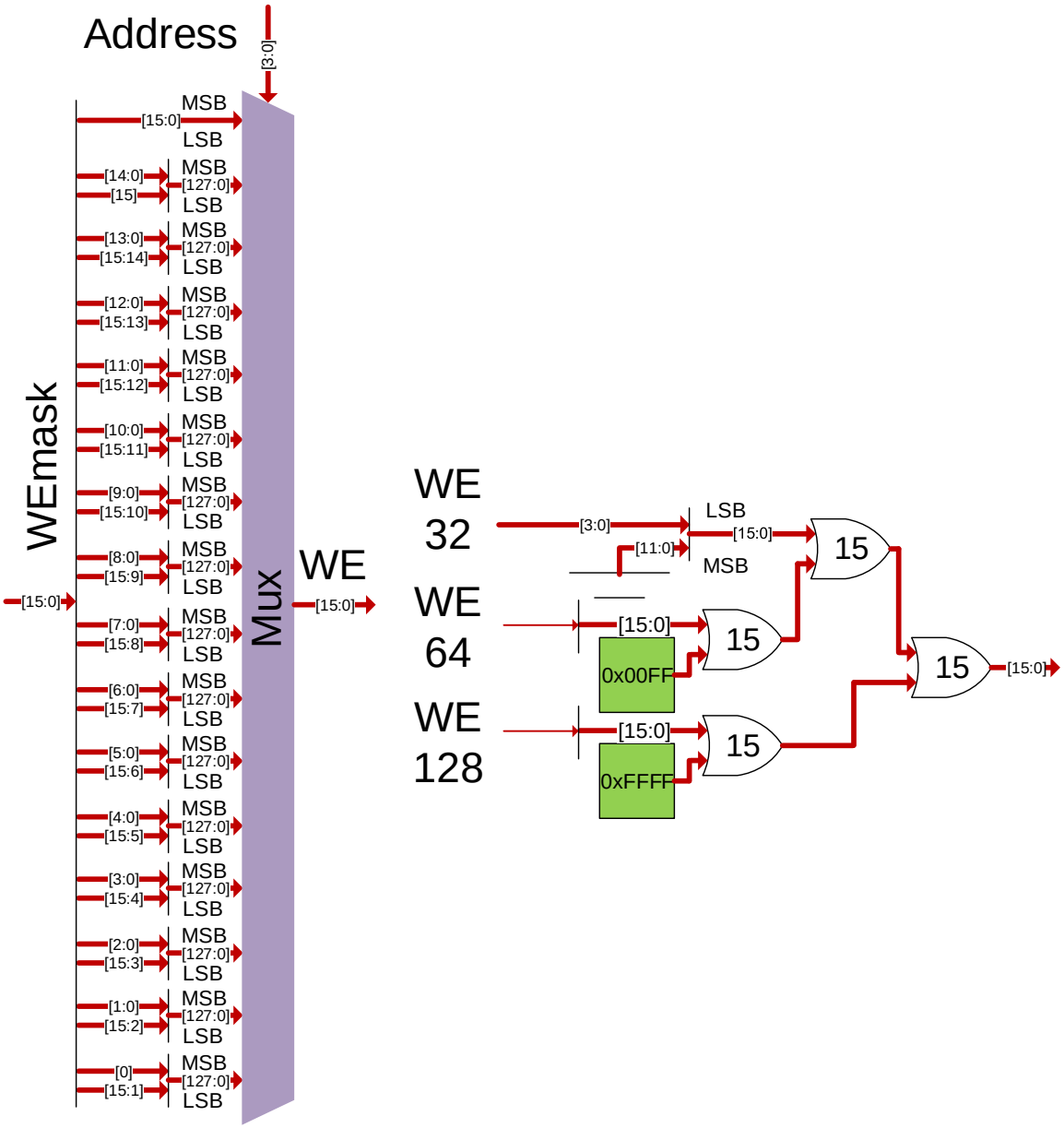


Figure 47: Logical circuit that creates the 16 write enable signals and multiplexor used to rearrange the write enable mask.

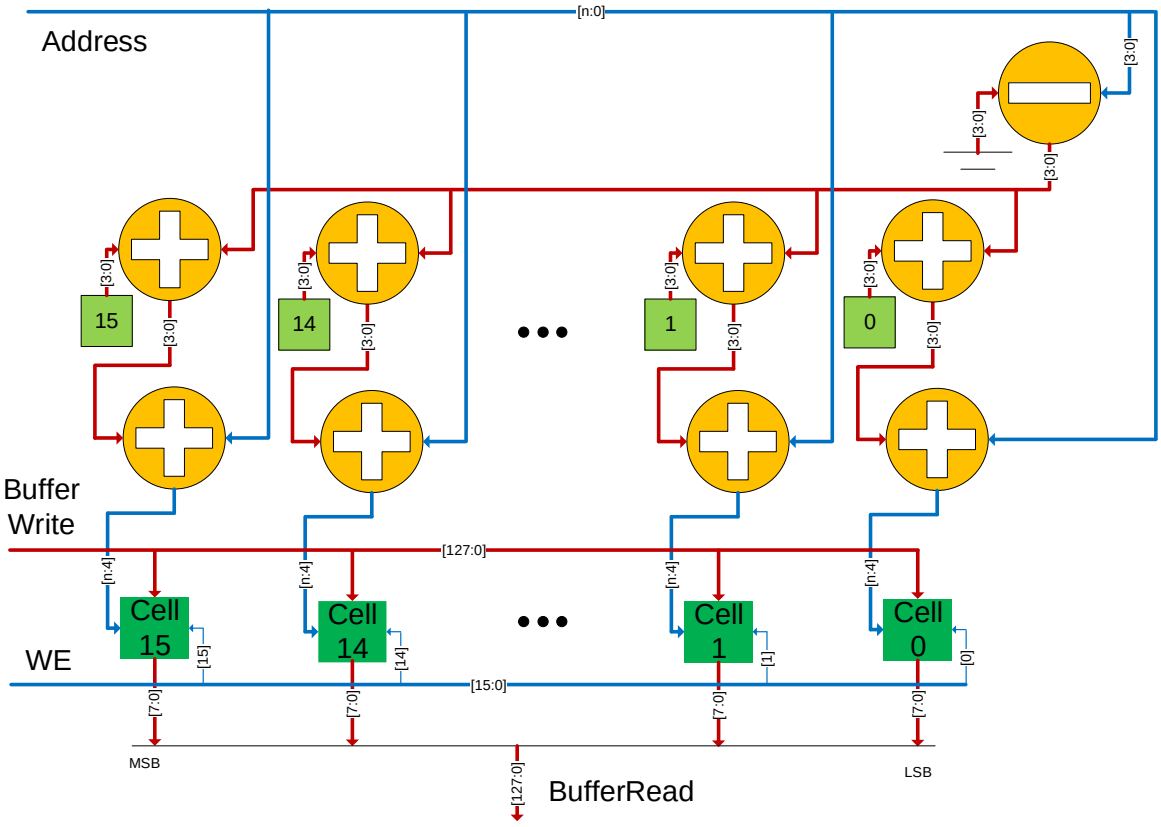


Figure 48: Calculation of rows for each memory cell

11 Software Tools

In this chapter we describe the software tools used to develop and test this project. Moreover, we describe a small application that we created to load programs into the FPGA for simulations.

11.1 GCC

In order to compile C programs to generate MIPS binaries we used a version of the GCC compiler provided by Imagination Community. This compiler is distributed together with Codescape MIPS SDK. The version (4.9.0) includes support for MIPS SIMD Architecture (55). Figure 49 shows the compilations flags needed to enable the SIMD ISA. As a requirement, MSA can be used only with MIPS Release 5 or higher, it can be 32 or 64-bit, but FPU must to be 64-bits wide. Using optimization flag –O2 GCC tries to auto-vectorize loops.

```
# Bare Metal Compiler SIMD
CC= mips-mti-elf-gcc
CFLAGS= -EL -mmsa -mfp64 -mips32r5 -ftree-vectorize -ftree-vectorizer-verbose=2 -O2
```

Figure 49: Compiler flags to enable SIMD

After the compilation there are some points to consider. First, the executable files generated by GCC use the *Executable and Linkable Format* (ELF). Second, it is desirable to generate Bare-Metal applications [ref to bare]. In consequence, it is indispensable to provide some extra info to GCC linker to avoid or at least to override *syscalls* to the Operating System. To fulfill this, the file "scrip.ld" has been provided to GCC the linker. Figure 50 shows this script, that indicates the start address of ".text" section, the end of data section and the other sections that will appear in the executable file.

```
ENTRY(main)
PROVIDE (__stack = 0);
SECTIONS
{
    . = 0x1fc00000;
    .text : {
                *(Inicio)
                *(.text);
        }
    . = 0x1fc80000;
    .rodata : { *(.rodata) }
    .data : { *(.data) }
    .bss : { *(.bss) }
}
```

Figure 50: File script.ld

Figure 51 shows the file “startup.S”, that initializes the global pointer ($gp) and the stack pointer ($sp), and then calls the main function from the program that is going to be executed. When the program ends, the *exit* function is called. The function just keeps the processor spinning at a certain address. When this address is reached the emulator (ModelSim) knows that the simulation has finished.

```
        .section "Inicio"

reset:
        lui $gp,0x8005
        addiu $gp,$gp,-20128
        lui $sp,0x1FC8
        addiu $sp,$sp,0x1FFC
        jal main
        nop
        jal exit
        nop

exit:
        j exit
```

Figure 51: File startup.S

Figure 52 shows the *makefile* used to compile and link bare-metal MIPS applications. First, the object files are created from the source code of the target application and “startup.S” file. Then they are linked using “script.ld” file. Once the executable application has been built is possible to run it using QEMU but to run it on ModelSim some additional steps are required.

```
jfdctint-bare:
        $(CC) $(CFLAGS) -EL -S -c $(src)jfdctint-auto.c
        $(CC) $(CFLAGS) -c $(src)jfdctint-auto.c -o jfdctint.o
        $(CC) $(CFLAGS) -c $(src)startup.S -o startup.o
        $(CC) -EL -o jfdctint jfdctint.o startup.o -T $(src)script.ld
        mips-sde-elf-objdump -D jfdctint > $(out)jfdctint.S
        ./translate ./ASM/jfdctint.S
```

Figure 52: Makefile

Once the application is compiled the next step is disassemble it. We are using CodeBench from MentorGraphics (56), its most recent version is 2.24.51.20140217. Figure 53 shows an example of disassembled code. CodeBench provides the memory address for each instruction and data. With this information it is possible to use an application that we have

developed call “translate” to create all *.hex files to fill memory cells required by ModelSim. These additional steps are essential because every memory cell requires an individual *.hex file and each memory cell stores only a byte. Finally, “translate” application fills with zeros the memory space that is not used by the application to avoid warning messages.

```
1fc001ec <Multiply>:
1fc001ec:       3c0e1fc8        lui     t6,0x1fc8
1fc001f0:       3c0d1fc8        lui     t5,0x1fc8
1fc001f4:       25ce065c        addiu   t6,t6,1628
1fc001f8:       3c0c1fc8        lui     t4,0x1fc8
1fc001fc:       25ad0cd4        addiu   t5,t5,3284
1fc00200:       3c091fc8        lui     t1,0x1fc8
1fc00204:       258c001c        addiu   t4,t4,28
1fc00208:       00005821        move    t3,zero
1fc0020c:       2529006c        addiu   t1,t1,108
1fc00210:       240f0640        li      t7,1600
1fc00214:       01ab4021        addu    t0,t5,t3
1fc00218:       01803821        move    a3,t4
1fc0021c:       01cb5021        addu    t2,t6,t3
```

Figure 53: Fragment of a decompile file using CodeBench

11.2 QEMU

Is possible to run MIPS applications on QEMU (57) which is a generic and open source machine emulator. It supports several architectures. Version 2.0 and higher supports MIPS Release 5 and SIMD instructions. Codescape MIPS SDK also includes QEMU. QEMU can be used to test applications and debug them using for instance GDB that is supported by QEMU.

Unfortunately QEMU only gives us information at software level[22], to obtain more accurate information at circuit level like information in each register, memory, buses, control signals, stages of the pipeline and so on we need to use a different tool.

11.3 ModelSim

ModelSim is a simulation tool developed by Mentor Graphics for simulation of hardware description languages (HDL). It supports different languages as VHDL, Verilog and SystemC. Quartus II suite includes a free version of ModelSim and it is linked to run simulation on ModelSim from Quartus II with one click. Once everything is configured, ModelSim generates waveforms allowing to observe the behavior of each component even at the gate

[22] QEMU allows to read data at the register level but not at the microarchitectural level.

level. Figure 54 shows an example of waveforms. Once a simulation has been executed it becomes possible to traverse across all signals.

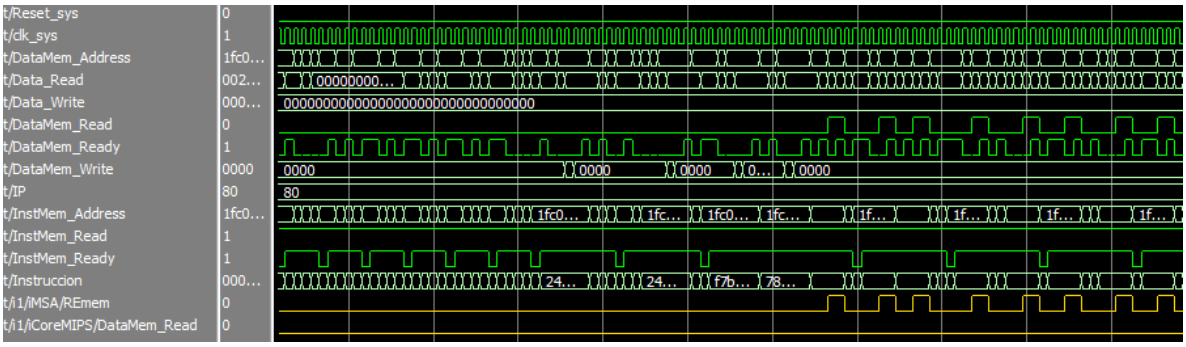


Figure 54: Example of waveforms generated by ModelSim

Analyzing waveforms implies much toil because there are a lot of them. ModelSim includes a memory visualizer; Figure 55 shows a data portion example. Despite of the benefits that are provided by this tool, sometimes is not enough to debug. Memories are too big to keep track of changes cycle by cycle.

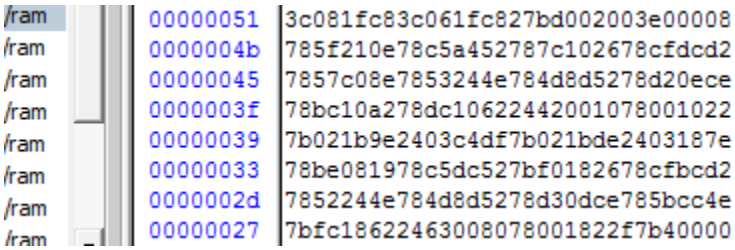


Figure 55: Memory visualization in ModelSim

The term instrumentation refers to the ability to monitor or measure the level of a product's performance and to diagnose errors. In programming, this means the ability of an application to incorporate (58):

- Code tracing: Receiving informative messages about the execution of an application at run time.
- Debugging: Tracking down and fixing programming errors in an application under development.
- Profiling: Tracking the performance of the application and events of the system via performance counters.
- Event logs: Tracking major events in the execution of the application.

These instrumentations concepts can be applied to hardware development since modern hardware development is similar to software development since it uses programing languages as well. Moreover, in the hardware world Debugging process is formally known verification (47).

It is often desired to keep verification code separated from the design code. We decided to use compiler directives to enable or disable instrumentation.

This has been implemented across the project in key points like write and read ports of memories, VPU, Control unit and so on. We have also created a test bench. Figure 56 shows a piece of code used to keep track of read and write operations. Figure 57 shows an example of a piece of flow data on the VPU. Each cycle VPU generates a result

```
`ifdef TraceDebug
initial
begin
        f = $fopen("output.txt","w");
        g = $fopen("trace.txt","w");
        $fwrite(f,"WE\tAddress\t\tData\n");
        $fwrite(g,"PC\t\tInstruction\n");
        @(negedge Reset_sys); //Wait for reset to be released
        @(posedge clk_sys);   //Wait for fisrt clock out of reset

         i = 0;
         while(InstMem_Address!=32'h1fc00028)
         begin
                @(posedge clk_sys);
                i=i+1;
                if(InstMem_Ready)
                begin
                        $fwrite(g,"%h\t%h\n",InstMem_Address,
                        Instruccion);
                end
                if(|DataMem_Write)
                begin
                        $fwrite(f,"%h\t%h\t%h\n",DataMem_Write,
                        DataMem_Address,Data_Write);
                end
                if( DataMem_Read & DataMem_Ready )
                begin
                        $fwrite(f,"Read\t%h\t%h\n",
                        DataMem_Address,Data_Read);
                end
         end


         $fwrite(f,"\nClock ticks: %d\n",i);
         $fclose(f);
         $fclose(g);
         $display("\nClock ticks: %d\n",i);

         $finish;
end
`endif
```

Figure 56: Example of instrumentation code

PC	Instruction	Src1	ReadA	Src2	ReadB	Dest	Result
1fc00318	78ccf912	31	0000227cffffc7d1ffffeb3f00001006	12	0000098e0000098e0000098e0000098e	4	09461e2e075ac955fe7d25d6fec19941
1fc00318	78ccf912	31	0000227cffffc7d1ffffeb3f00001006	12	0000098e0000098e0000098e0000098e	4	09461e2e075ac955fe7d25d6fec19941
1fc0031c	7be01ce6	3	06894f1f0194c1ddf53a239bced111fe	0	0001713cfffe9f10000268acfffcdf34	19	06774f1f016c3f230b3a2365322f1102
1fc00320	78c9a092	20	ffff9686ffffe42c00001b4d00000059	9	0000300b0000300b0000300b0000300b	2	f4313928043aa84b04630e33fe3933c0
1fc00324	78be0cd9	1	08c971170217dc92f8835ba1f74a4f4d	30	00000000000000000000000000000000	19	08c971170217dc92f8835ba1f74a4f4d
1fc00328	78be8e59	17	ee9842f708d6b3f706a7bbee17f61db5	30	00000000000000000000000000000000	25	ee9842f708d6b3f706a7bbee17f61db5
1fc00328	78be8e59	17	ee9842f708d6b3f706a7bbee17f61db5	30	00000000000000000000000000000000	25	ee9842f708d6b3f706a7bbee17f61db5
1fc0032c	78beac59	21	e71b56c5de21c55610fe1b7b1184744d	30	00000000000000000000000000000000	17	e71b56c5de21c55610fe1b7b1184744d
1fc00330	78cfbcd2	23	00008e4affffacb5ffff9016fffef731	15	0000187e0000187e0000187e0000187e	19	16666983fa1fd5a8edce5675ddf4916b
1fc00334	7bf01826	3	06894f1f0194c1ddf53a239bced111fe	16	00001151000011510000115100001151	0	06774f1f016c3f230b3a2365322f1102
1fc00338	78c5dc52	27	ffff752fffffc7c30000497200009175	5	fffff384fffff384fffff384fffff384	17	ede06001e0dfdee20d6930430a6c83a1
1fc00338	78c5dc52	27	ffff752fffffc7c30000497200009175	5	fffff384fffff384fffff384fffff384	17	ede06001e0dfdee20d6930430a6c83a1
1fc0033c	78be0819	1	08c971170217dc92f8835ba1f74a4f4d	30	00000000000000000000000000000000	0	08c971170217dc92f8835ba1f74a4f4d
1fc00340	7859210e	4	09461e2e075ac955fe7d25d6fec19941	25	ee9842f708d6b3f706a7bbee17f61db5	4	f7de612510317d4c0524e1c416b7b6f6
1fc00344	78ceb012	22	fffff39d0000723d0000013b0000880c	14	ffffc4dfffffc4dfffffc4dfffffc4df	0	0ba5dadae7b513b5f83a9a06d7de01c1
1fc00348	785918ce	3	06894f1f0194c1ddf53a239bced111fe	25	ee9842f708d6b3f706a7bbee17f61db5	3	f52192160a6b75d4fbe1df89e6c72fb3
1fc00348	785918ce	3	06894f1f0194c1ddf53a239bced111fe	25	ee9842f708d6b3f706a7bbee17f61db5	3	f52192160a6b75d4fbe1df89e6c72fb3
1fc0034c	794b210a	4	f7de612510317d4c0524e1c416b7b6f6	11	0000000b0000000b0000000b0000000b	4	fffefbcc000206300000a49c0002d6f7
1fc00350	794b18ca	3	f52192160a6b75d4fbe1df89e6c72fb3	11	0000000b0000000b0000000b0000000b	3	fffea43200014d6fffff7c3cfffcd8e6
1fc00354	7851904e	18	16f48a42279c2ecae8e793b2044e622a	17	ede06001e0dfdee20d6930430a6c83a1	1	04d4ea43087c0dacf650c3f50ebae5cb
1fc00358	7851108e	2	f4313928043aa84b04630e33fe3933c0	17	ede06001e0dfdee20d6930430a6c83a1	2	e2119929e51a872d11cc3e7608a5b761
1fc00358	7851108e	2	f4313928043aa84b04630e33fe3933c0	17	ede06001e0dfdee20d6930430a6c83a1	2	e2119929e51a872d11cc3e7608a5b761
1fc0035c	794b000a	0	0ba5dadae7b513b5f83a9a06d7de01c1	11	0000000b0000000b0000000b0000000b	0	000174bbfffcf6a2ffff0753fffafbc0
1fc00360	794b9c4a	19	16666983fa1fd5a8edce5675ddf4916b	11	0000000b0000000b0000000b0000000b	17	0002cccdffff43fbfffdb9cbfffbbe92
1fc00364	794b084a	1	04d4ea43087c0dacf650c3f50ebae5cb	11	0000000b0000000b0000000b0000000b	1	00009a9d00010f82fffeca180001d75d
1fc00368	794b108a	2	e2119929e51a872d11cc3e7608a5b761	11	0000000b0000000b0000000b0000000b	2	fffc4233fffca35100023988000114b7
1fc00368	794b108a	2	e2119929e51a872d11cc3e7608a5b761	11	0000000b0000000b0000000b0000000b	2	fffc4233fffca35100023988000114b7

Figure 57: A segment of a trace execution from the VPU

The final step to validate the implementation is comparing the output of the simulation and the execution of the same application running on QEMU. The results must match.

12 Evaluation

12.1 FPGA utilization

Table 15 shows a summary of hardware resources used by the implementation. We can see that the SIMD unit uses a huge amount of resources (75.55%) of the FPGA. It is not a fair comparison between the MIPS32 core because SIMD unit supports operations up to 64-bit and MIPS32 only 32-bit Some operations increase quadratically the amount of resources needed with respect the width of the operation. Moreover SIMD unit supports more instructions than the MIPS32 core.

Unit	LEs	Utilization
MIPS32 Core	6,937	6.06%
VPU and Shuffle	81,341	71.05%
SIMD miscellanies	5,153	4.50%
Memory	3,128	2.73%
Total	96,559	84.35%

Table 15: Resources of the implementation

MIPS32r1_xum core only can run up to 30 MHz so entire maximum frequency of project is limited by the MIPS32 core. To achieve higher a frequency pipeline size should increase too. But MIPS32 processor must be modified as well to keep stage synchronization. Nevertheless, the maximum frequency achieve by the SIMD unit is about 60 MHz, but also will benefit of extra pipeline.

12.2 Benchmarks

To evaluate the performance of the SIMD unit a couple of micro benchmarks have been compared considering three scenarios.

- Compile from “C” code using the maximum optimization flag supported by Codescape but avoiding to use SIMD instructions too (-O2 flag).
- Compile from “C” code using the maximum optimization flag supported by Codescape and enable SIMD instructions and auto-SIMD by GCC (-mmsa -O2 flag).
- Rewrite the benchmark kernel in assembler using all possible SIMD instructions and for the remaining “C” code use the maximum optimization flag.

There are some restrictions to run benchmarks, the FPU is not included on the main core, neither floating point SIMD. Moreover, memory size is limited up to 512 KB and there is not any operating system that can be executed without a full implementation of the MIPS ISA. These are the reasons why it has been decided to use micro benchmarks, the

implementation of the remaining elements that are needed to integrate the complete system are out of the scope of this thesis. We have used two microbenchmarks: FDCT and Matmul.

12.2.1 FDCT

An example of the use of the Fast Discrete Cosine Transform (FDCT) is as a kernel on JPEG codification. FDCT requires a lot of computation based on integer array elements. Some characteristics are that it operates with arrays and bit operations. As input it has a Matrix of 16x16 32-bit elements that in our experiments is initialized with random data.

Table 16 shows the results of executing this benchmark under the constraints of each of the three scenarios proposed. The speedup is calculated using the number of cycles required to execute the benchmark. The difference between using SIMD instructions or not is up to 4.05x over the code generated by GCC using optimization flag "-O2".

Optimization	Instructions	Cycles	Step Speedup	Cumulative Speedup
-O2	451	10299	1.0x	1.0x
-O2 mmsa	642	5694	1.81x	1.81x
Hand written assembly	387	2546	2.24x	4.05x

Table 16: FDCT Benchmark results, total speedup of using SIMD is 4.05x

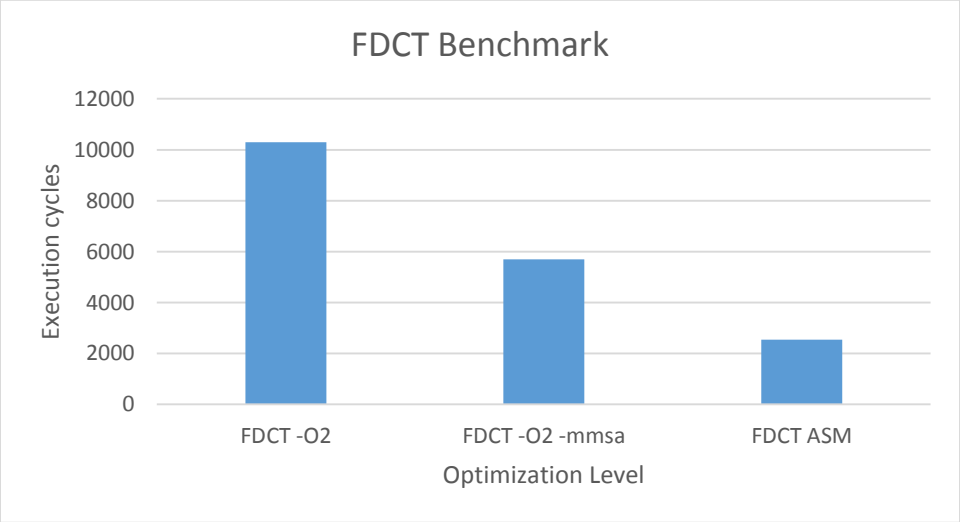


Figure 58: FDCT Benchmark results

12.2.2 Matmult

Matmult is a matrix multiplication of two 20x20 matrices. Both matrices were filled with random data. Some features are loops, nested loops and arrays.

Table 17 shows the results of executing this benchmark in the three scenarios proposed. The speedup is calculated using the number of cycles required to execute the benchmark. The speedup of using SIMD instructions with hand written assembly is 2.68x over the code generated by GCC using optimization flag "-O2".

Optimization	Instructions	Cycles	Step Speedup	Cumulative Speedup
-O2	195	85825	1.0x	1.0x
-O2 -mmsa	194	49803	1.72x	1.72x
Hand written assembly	194	31903	1.56x	2.68x

Table 17: Matmult Benchmark results, total speedup of using SIMD is 2.68x

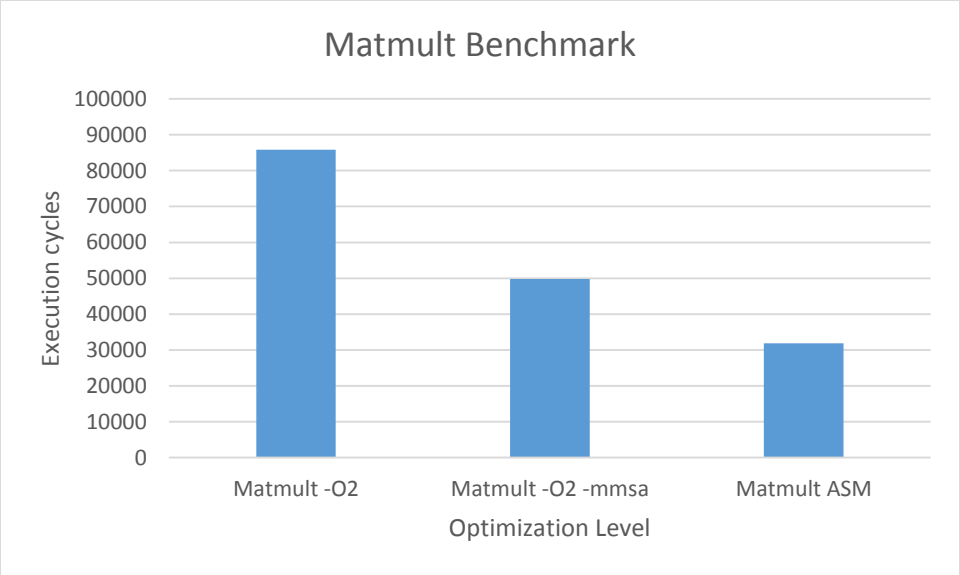


Figure 59: Matmult Benchmark results

12.3 Summary

It has been implemented 123 SIMD instructions, as well as it has been built and tested, using an entire test bench to take code C code, compile it with GCC and test in Simulink. Two micro benchmarks have been executed with and without SIMD instructions to study the impact in performance of adding SIMD support, as well as the effectiveness of GCC auto vectorization.

13 Future work

Current implementation of the MIPS SIMD Architecture is still under development. It can be expanded with the incorporation of Floating-Point SIMD operation lanes. This project is part of a bigger project, that has as main goal the design and implementation of a superscalar processor with out-of-order support, based on MIPS64 Release 6 (13).

Figure 60 shows a block diagram of the superscalar processor. That project is divided in smaller projects, some of them are SIMD, FPU and Load/Store Queue. The FPU will be transformed into FPU lanes and integrated into the SIMD unit. The SIMD Functional Units (VPU, Shuffle, FP) will be used in the superscalar processor. An alternative to increase the performance the SIMD unit is to support a wider vector register set of 256 bits. MSA scales with the vector register width.

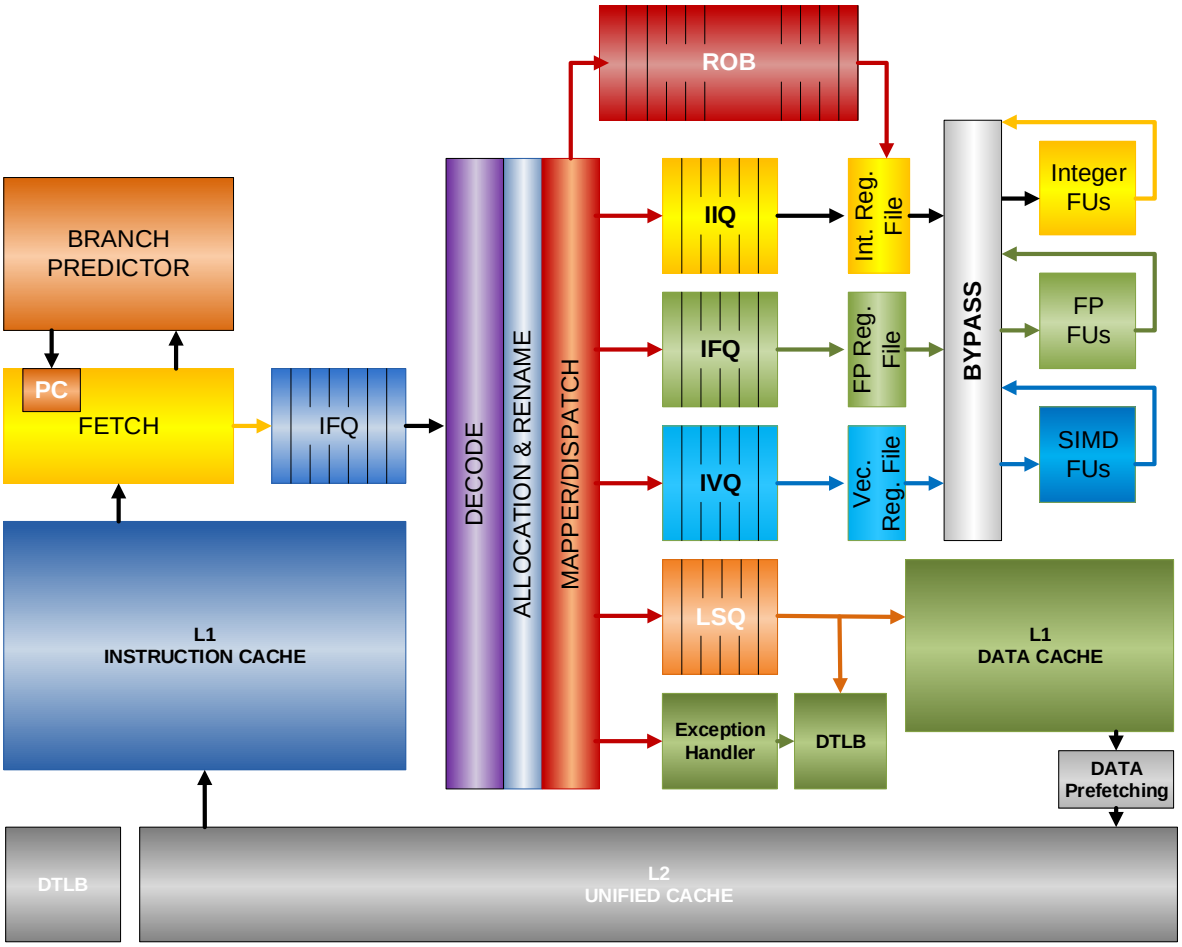


Figure 60: Superscalar processor

14 Conclusions

We realized that designing a microarchitecture requires significant effort and expertise. It is full of challenges, because there is always a tradeoff between performance and power. Computer architecture has been improving designs generation by generation and always trying to keep easy the programmer life. We are in an inflection point, because now the programmer have to take care about de architecture in order to achieve performance. Parallel programming models such as OpenMP are helping. It is just the beginning since ILP, DLP and TLP are starting to mix and new architectures are appearing. Also new execution models that exploits them are appearing too, such as SIMT.

We have to keep in mind that classic models and techniques are also evolving such as, new SIMD extensions with wider registers and more features will continue appearing. For instance from MMX, SSE, AVX and AVX-512 Intel has move from 64-bit, 128-bit, 256-bit and 512-bit wide. We can expect 1024-bit wide SIMD extensions to appear in the mid-term future. We have to start to think in parallel from the basic “computer education”, programmers and computer architects. That is the main goal of this big project. Encourage new generations to thing in parallel.

15 References

1. *Cramming More Components onto Integrated Circuits.* **MOORE, GORDON E.** 1, 1998, PROCEEDINGS OF THE IEEE, Vol. 86.

2. *HPSm, a high performance restricted data flow architecture having minimal functionality.* **Hwu, Wen-reel and Patt, Yale N.** 1986. ACM SIGARCH Computer Architecture News. Vol. 14, pp. 297-306.

3. *Complexity-Effective Superscalar Processors.* **Jouppi, Norman P and Smith, JE.** 1997. Intl. Symposium on Computer Architecture. Vol. 145.

4. *Design challenges of technology scaling.* **Borkar, Shekhar.** 4, s.l. : IEEE, 1999, Micro, IEEE, Vol. 19, pp. 23-29.

5. *Amdahl's law in the multicore era.* **Hill, Mark D and Marty, Michael R.** 7, s.l. : IEEE, 2008, Computer, pp. 33-38.

6. *Data processing in exascale-class computer systems.* **Moore, Chuck.** 2011. The Salishan Conference on High Speed Computing.

7. **Cavium.** ThunderX™ ARM Processors. [Online] [Cited: Jun 1, 205.] http://www.cavium.com/ThunderX_ARM_Processors.html.

8. **Intel Corporation.** Familia de productos Intel Xeon Phi. [Online] [Cited: Jun 1, 2015.] http://www.intel.es/content/www/es/es/processors/xeon/xeon-phi-detail.html.

9. *MMX technology extension to the Intel architecture.* **Peleg, Alex and Weiser, Uri.** 4, s.l. : IEEE, 1996, Micro, IEEE, Vol. 16, pp. 42-50.

10. **Kusswurm, Daniel.** Streaming SIMD Extensions. *Modern X86 Assembly Language Programming.* s.l. : Springer, 2014, pp. 179-206.

11. *Intel© 64 and IA-32 Architectures Software Developer's Manual.* **Intel Corporation.** April 2015, Intel Corporation, pp. 109-138.

12. **Instituto Politécnico Nacional.** Arquitectura de Computadoras Embebidas de Alto Desempeño. [Online] [Cited: July 1, 2015.] http://www.microse.cic.ipn.mx/emb.hpca.

13. **Microse.** Microtechnology and Embedded System Lab. *Microtechnology and Embedded System Lab.* [Online] June 24, 2015. http://www.microse.cic.ipn.mx/lagarto.

14. *The Haswell microarchitecture--4th generation processor.* **Jain, Tarush and Agrawal, Tanmay.** 3, s.l. : Citeseer, 2013, International Journal of Computer Science and Information Technologies, Vol. 4, pp. 477-480.

15. **Hennessy, John L and Patterson, David A.** *Computer architecture: a quantitative approach.* s.l. : Elsevier, 2011.

16. *Scaling the power wall: a path to exascale.* **Villa, Oreste, et al.** 2014. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 830-841.

17. *Some computer organizations and their effectiveness.* **Flynn, Michael J.** 9, s.l. : IEEE, 1972, Computers, IEEE Transactions on, Vol. 100, pp. 948-960.

18. *SIFT: Design and analysis of a fault-tolerant computer for aircraft control.* **Wensley, John H, et al.** 10, s.l. : IEEE, 1978, Proceedings of the IEEE, Vol. 66, pp. 1240-1255.

19. *Vector architectures: past, present and future.* **Espasa, Roger, Valero, Mateo and Smith, James E.** 1998. Proceedings of the 12th international conference on Supercomputing. pp. 425-432.

20. *Supercomputing with commodity CPUs: are mobile SoCs ready for HPC?* **Rajovic, Nikola, et al.** 2013. High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for. pp. 1-12.

21. *Processor microarchitecture: An implementation perspective.* **González, Antonio, Latorre, Fernando and Magklis, Grigorios.** 1, s.l. : Morgan \& Claypool Publishers, 2010, Synthesis Lectures on Computer Architecture, Vol. 5, p. 81.

22. *Merging ILP and DLP for High Performance.* **Espasa, Roger.**

23. *NVIDIA´s Next Generation CUDA Compute Architecture: Fermi.* **NVIDIA Corporation.** 2009.

24. **McCool, Michael D, Robison, Arch D and Reinders, James.** *Structured parallel programming: patterns for efficient computation.* s.l. : Elsevier, 2012.

25. *64-bit and Multimedia Extensions in the PA-RISC 2.0 Architecture.* **Lee, Ruby and Huck, Jerry.** 1996. Compcon'96.'Technologies for the Information Superhighway'Digest of Papers. pp. 152-160.

26. *VIS speeds new media processing.* **Tremblay, Marc, et al.** 4, s.l. : IEEE, 1996, Micro, IEEE, Vol. 16, pp. 10-20.

27. *Digital, MIPS add multimedia extensions.* **Gwennap, Linley.** 15, 1996, Microprocessor Report, Vol. 10, pp. 24-28.

28. *Altivec extension to PowerPC accelerates media processing.* **Diefendorff, Keith, et al.** 2, s.l. : IEEE, 2000, Micro, IEEE, Vol. 20, pp. 85-95.

29. **Intel Corporation.** Processors - Define SSE2, SSE3 and SSE4. [Online] July 26, 2015. http://www.intel.com/support/processors/sb/cs-030123.htm.

30. *Intel SSE4 Programming Reference.* **Intel Corporation.** 2007, pp. 7-8.

31. *Intel avx: New frontiers in performance improvements and energy efficiency.* **Firasta, Nadeem, et al.** 2008, Intel white paper.

32. *A fully integrated multi-CPU, GPU and memory controller 32nm processor.* **Yuffe, Marcelo, et al.** 2011. Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International. pp. 264-266.

33. **Reinders, James.** Knights Corner: Your Path to Knights Landing. [Online] July 26, 2015. https://software.intel.com/sites/default/files/managed/e9/b5/Knights-Corner-is-your-path-to-Knights-Landing.pdf.

34. **Sweetman, Dominic.** *See MIPS run.* s.l. : Morgan Kaufmann, 2010.

35. **The New York Times.** Silicon Graphics to Buy MIPS for $406.1 Million. [Online] July 26, 2015. http://www.nytimes.com/1992/03/13/business/silicon-graphics-to-buy-mips-for-406.1-million.html.

36. —. Silicon Graphics to Sell Its Stake in MIPS. [Online] July 26, 2015. http://www.nytimes.com/1999/01/15/business/silicon-graphics-to-sell-its-stake-in-mips.html?ref=topics.

37. **Imagination Technologies LTD.** Acquisition of MIPS Technologies completed. [Online] July 26, 2015. http://www.imgtec.com/news/detail.asp?ID=724.

38. **Kjell, Bradley.** Programmed Introduction to MIPS Assembly Language. [Online] July 26, 2015. https://chortle.ccsu.edu/AssemblyTutorial/Chapter-31/ass31_2.html.

39. *MIPS Architecture For Programmers Volume I-A: Introduction to the MIPS64 Architecture.* **Imagination Technologies LTD.** 2014, p. 24.

40. —.**Imagination Technologies LTD.** 2014, p. 36.

41. —.**Imagination Technologies LTD.** 2014, pp. 36-38.

42. *MIPS SIMD Architecture.* **Imagination Technologies LTD.** 2014, Whitepaper, p. 4.

43. **Imagination Technologies LTD.** Codescape MIPS SDK. [Online] July 27, 2015. http://community.imgtec.com/developers/mips/tools/codescape-mips-sdk/.

44. *MIPS SIMD programming Optimizing multimedia codecs.* **Imagination Technologies LTD.** 2015, p. 2.

45. *MIPS Toolchain MSA Programmers Guide.* **Imagination Technologies LTD.** 2014, p. 3.

46. **Free Software Foundation, Inc.** Options for Linking. [Online] July 27, 2015. https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html.

47. **Association, IEEE Standards and others.** *IEEE Standard for SystemVerilog--unified Hardware Design, Specification, and Verification Language.* s.l. : IEEE, 2010.

48. *DE2-115 User Manual.* **Terasic Technologies Inc.** 2012, Terasic Technologies Inc, pp. 11-13.

49. *Design of High Speed Kogge-Stone Based Carry Select Adder.* **Chakali, Pakkiraiah and Patnala, Madhu Kumar.** 2013, International Journal of Emerging Science and Engineering, Vol. 1.

50. *Advanced Synthesis Cookbook.* **Altera Corporation.** 2011, Altera complete design suit, pp. 16-17.

51. *Cyclone IV Device Handbook Volume 1.* **Altera Corporation.** 2010, Altera Corporation, pp. 15-16.

52. *Efficient FPGA-Based Karatsuba Multipliers for Polynomials over F2.* **von zur Gathen, Joachim and Shokrollahi, Jamshid.** 2006. Selected Areas in Cryptography. pp. 359-369.

53. **Ayers, Grant.** mips32r1_xum. [Online] July 28, 2015. https://github.com/grantae/mips32r1_xum.

54. *MIPS Architecture For Programmers Volume I-A: Introduction to the MPIS64 Architecture.* **Imagination Technologies LTD.** 2014, p. 16.

55. **Imagination Technologies LTD.** Imagination's popular Codescape tools now provide MIPS CPU support and extended Linux/RTOS capabilities. [Online] July 28, 2015. http://www.imgtec.com/news/detail.asp?ID=840.

56. **Mentor Graphics.** Sourcery CodeBench. [Online] July 28, 2015. http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/overview/.

57. **Fabrice Bellard.** QEMU. [Online] July 02, 2015. http://wiki.qemu.org/Main_Page.

58. **Microsoft.** Introduction to Instrumentation and Tracing. [Online] July 29, 2015. https://msdn.microsoft.com/en-us/library/aa983649(VS.71).aspx.

59. *New hardware architecture for bit-counting.* **Dalalah, Ahmed, Baba, Sami and Tubaishat, Abdallah.** 2006. Proceedings of the 5th WSEAS international conference on Applied computer science. pp. 118-128.

60. *MIPS SIMD Architecture.* **MIPS Technologies, Inc.** April 8, 2014, MIPS Technologies, Inc., pp. 1-28.

61. **Tiassou, Kossi, et al.** Modeling aircraft operational reliability. *Computer Safety, Reliability, and Security.* s.l. : Springer, 2011, pp. 157-170.

62. *Intel Architecture Instruction Set Extensions Programming Reference.* **Intel Corporation.** October 2014, p. 18.

16 Annexes

16.1 Joining results from 3R lanes

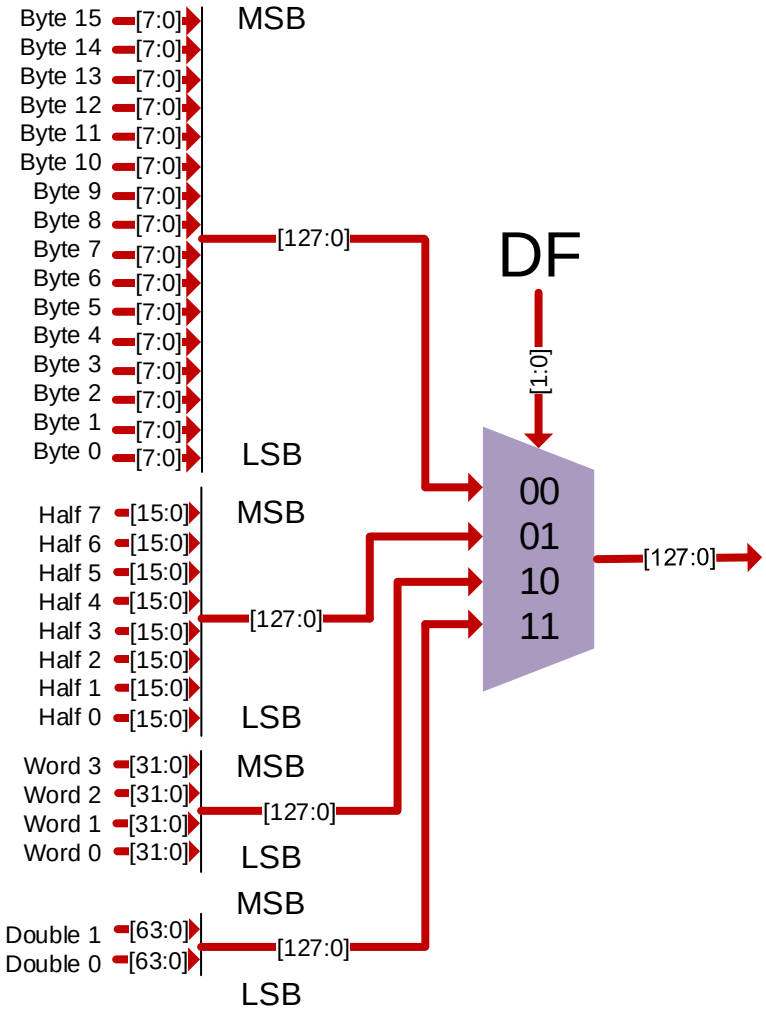


Figure 61: Selector of vector result format

16.2 Detail Implementation of Special 1 unit

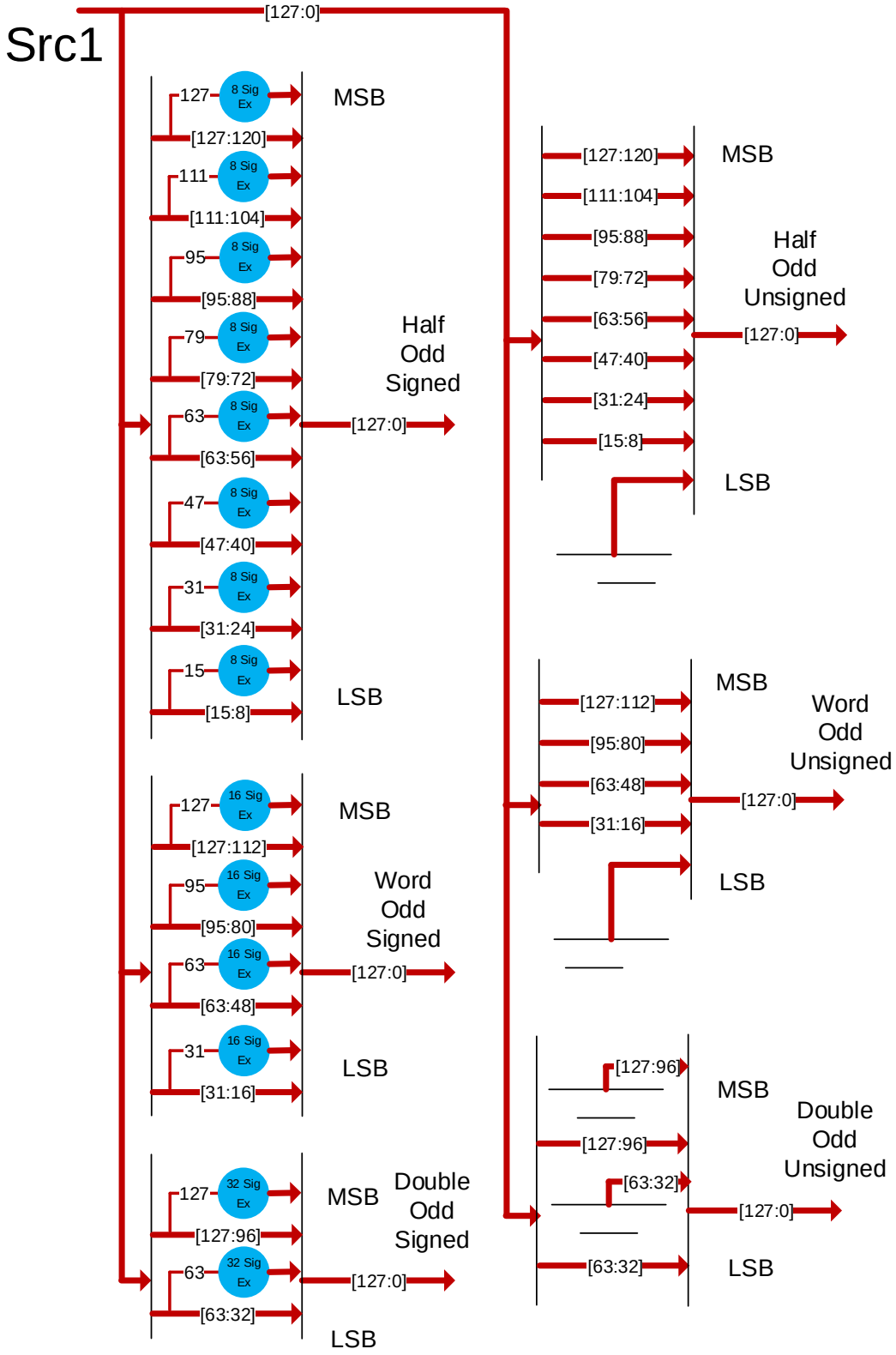


Figure 62: Implementation of special unit 1

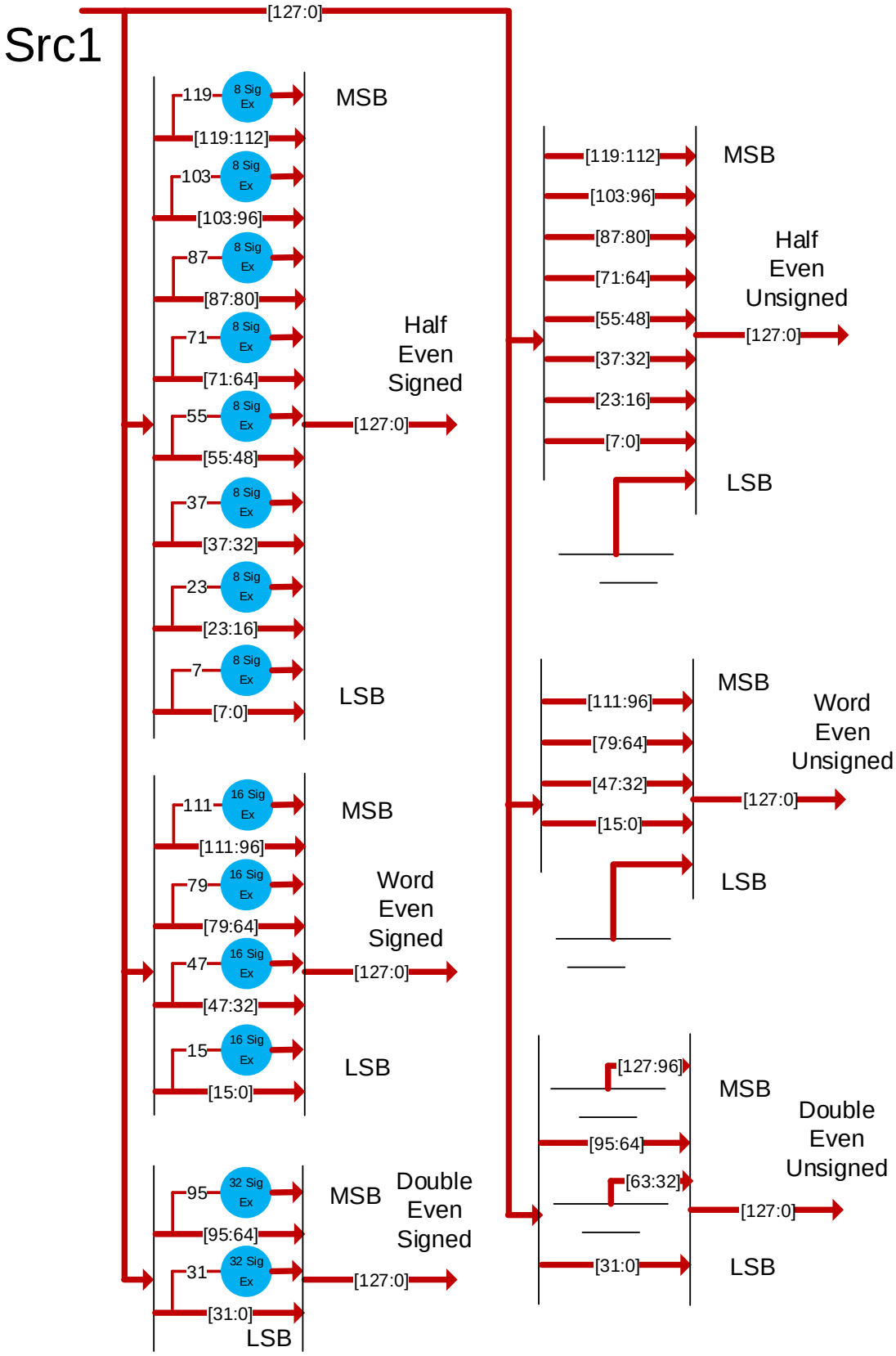


Figure 63: Implementation of special unit 1 (cont.)

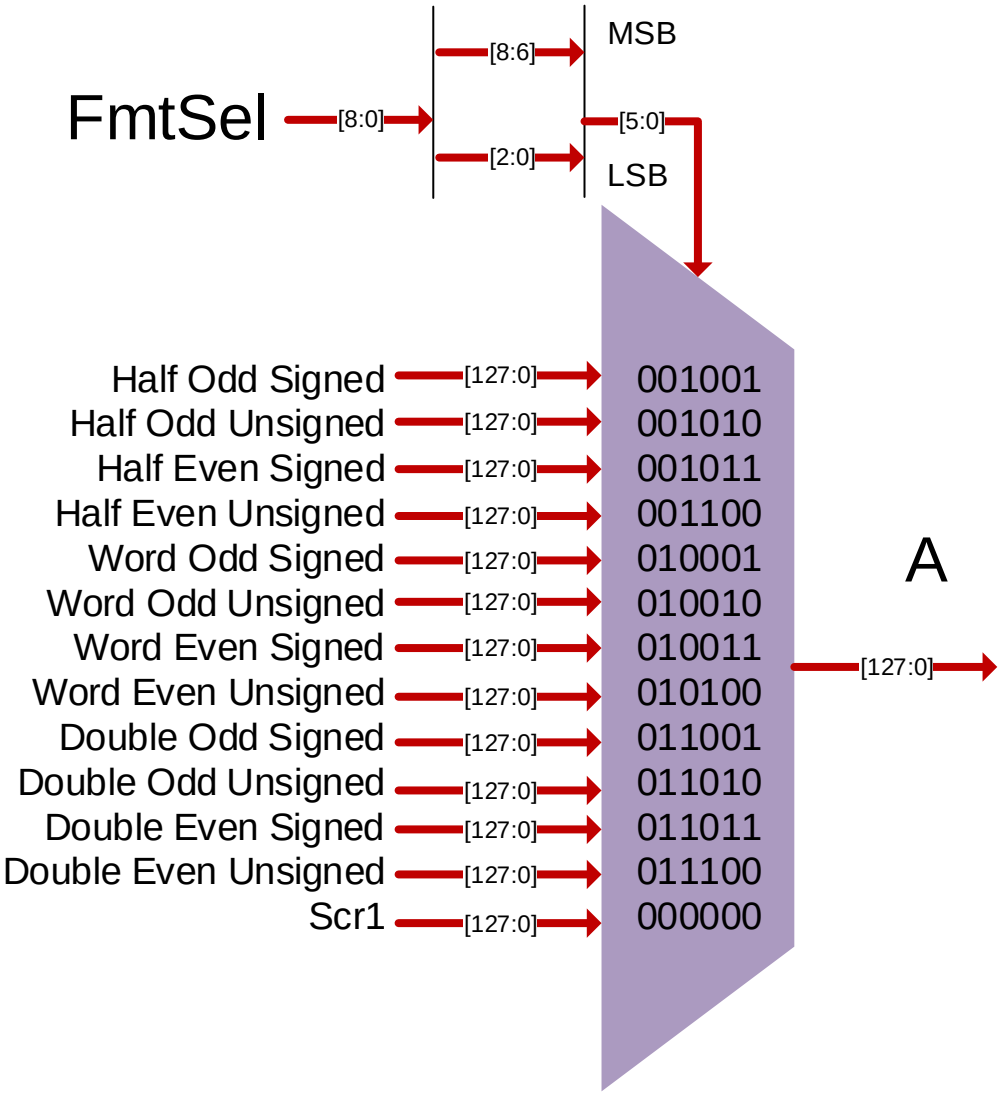


Figure 64: Joining result of special unit 1

16.3 Detail implementation of Special 2 unit

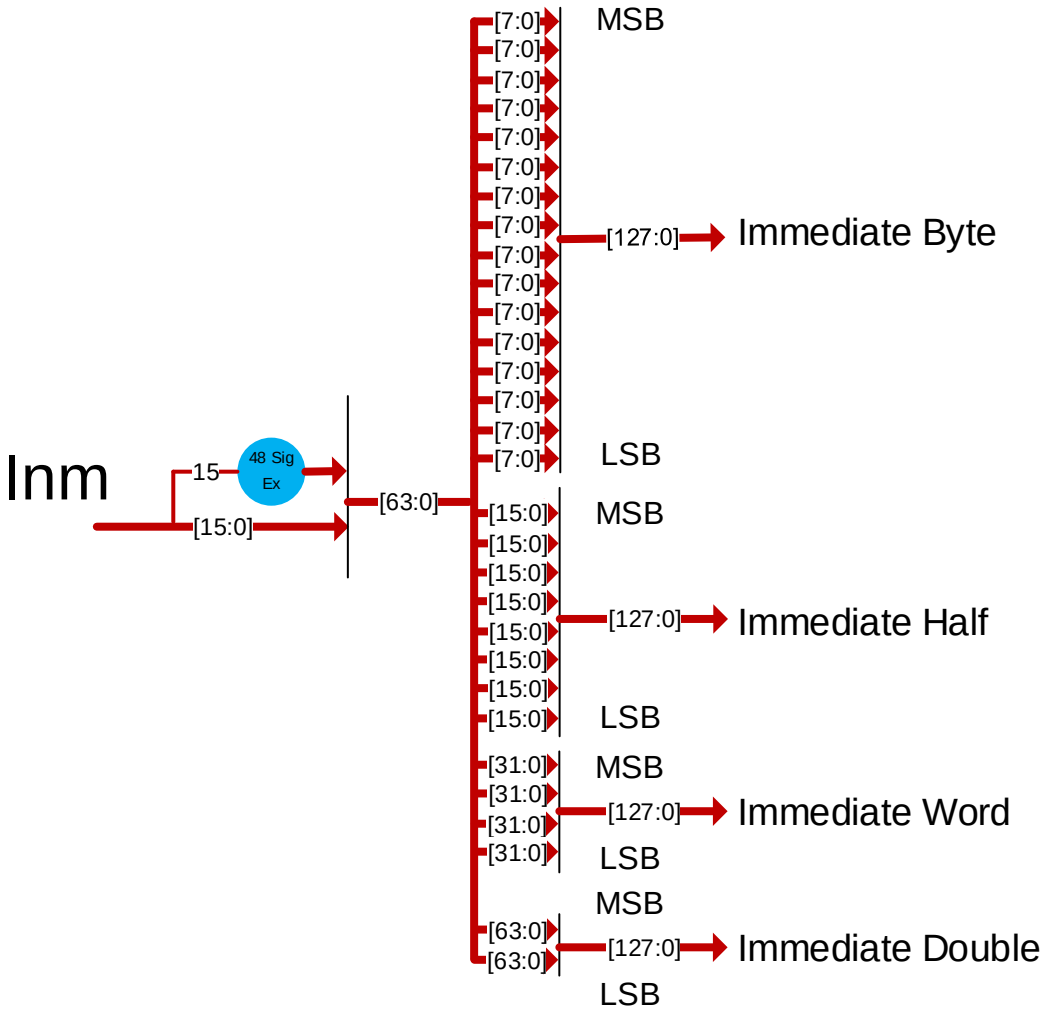


Figure 65: Implementation of the special unit 2

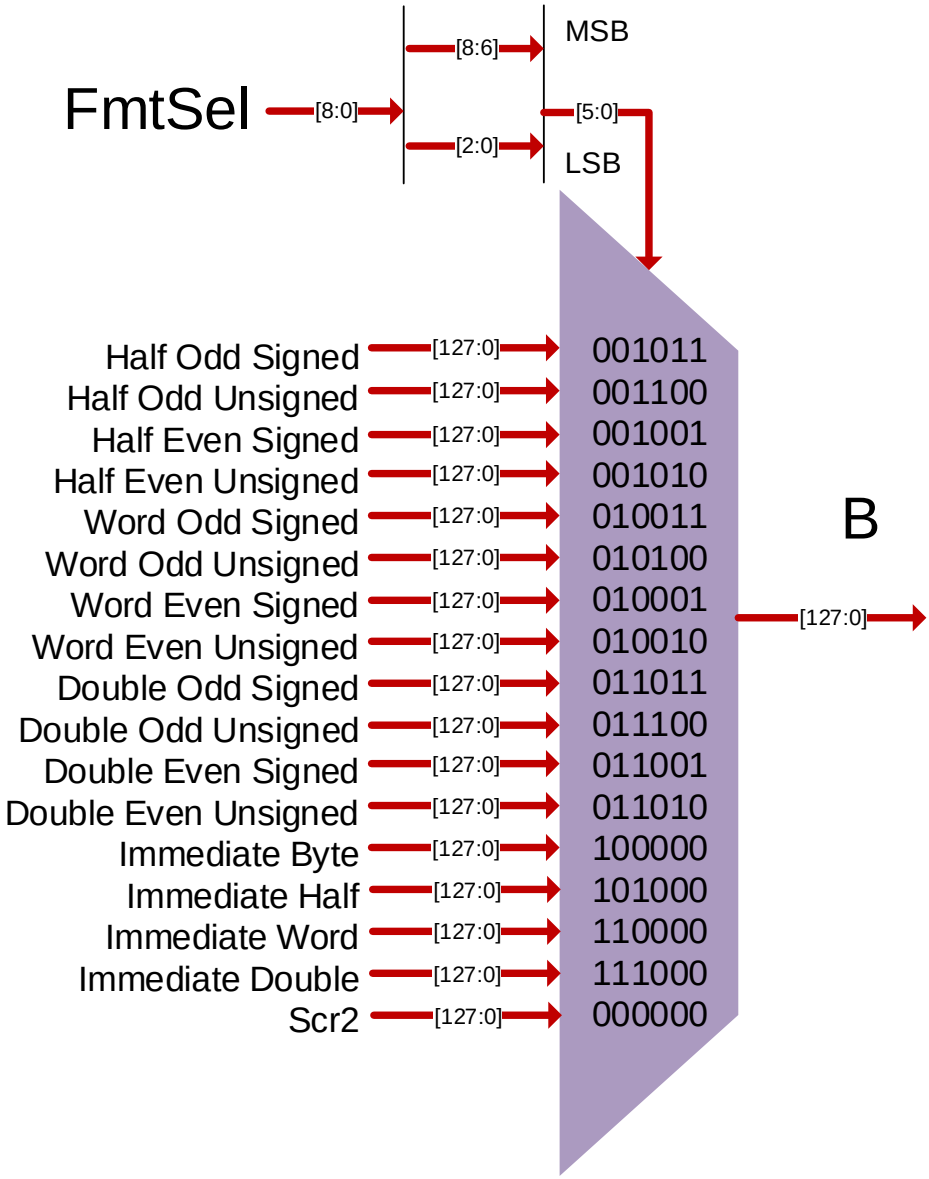


Figure 66: Result of the special unit 2

16.4 Detail implementation of Special 3 unit

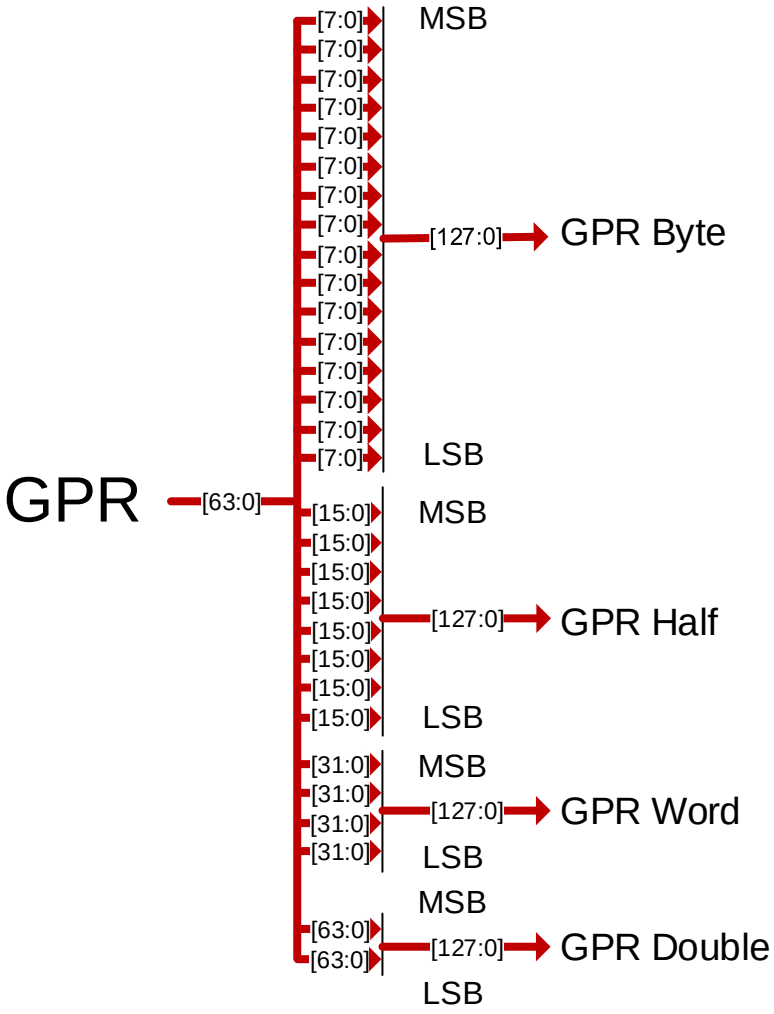


Figure 67: Implementation of special unit 3

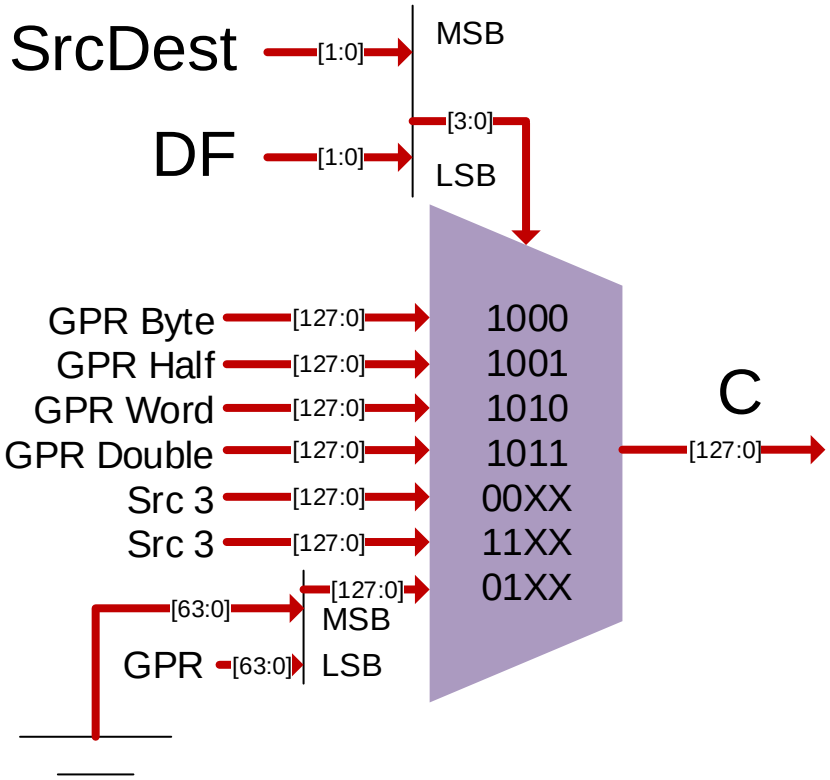


Figure 68: Result of special unit 3

16.5 VSHF unit

Figure 69 shows the VSHF unit. This unit is used to process Vector Data Preserving Shuffle instruction in 4 vector formats. For the correct implementation of this unit is required a specific circuit responsible for the operation over each vector format. This circuits are shown in Figure 70, Figure 71, Figure 72 and Figure 74.

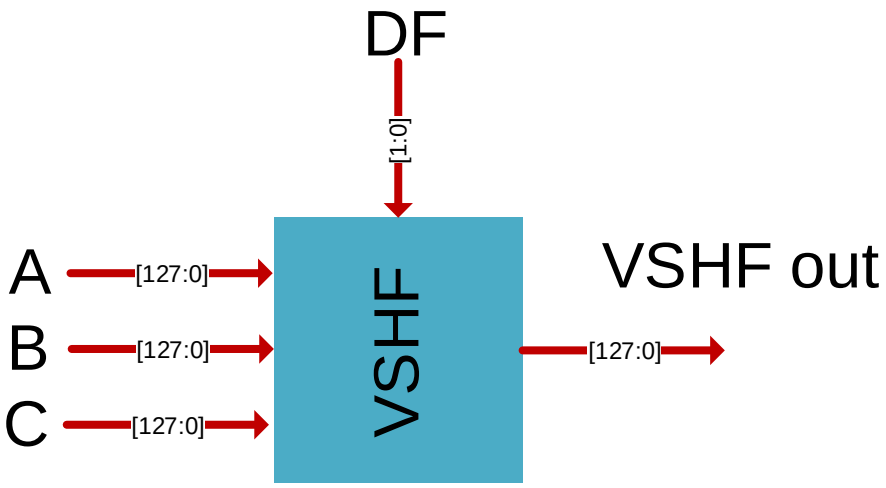


Figure 69: VSHF unit

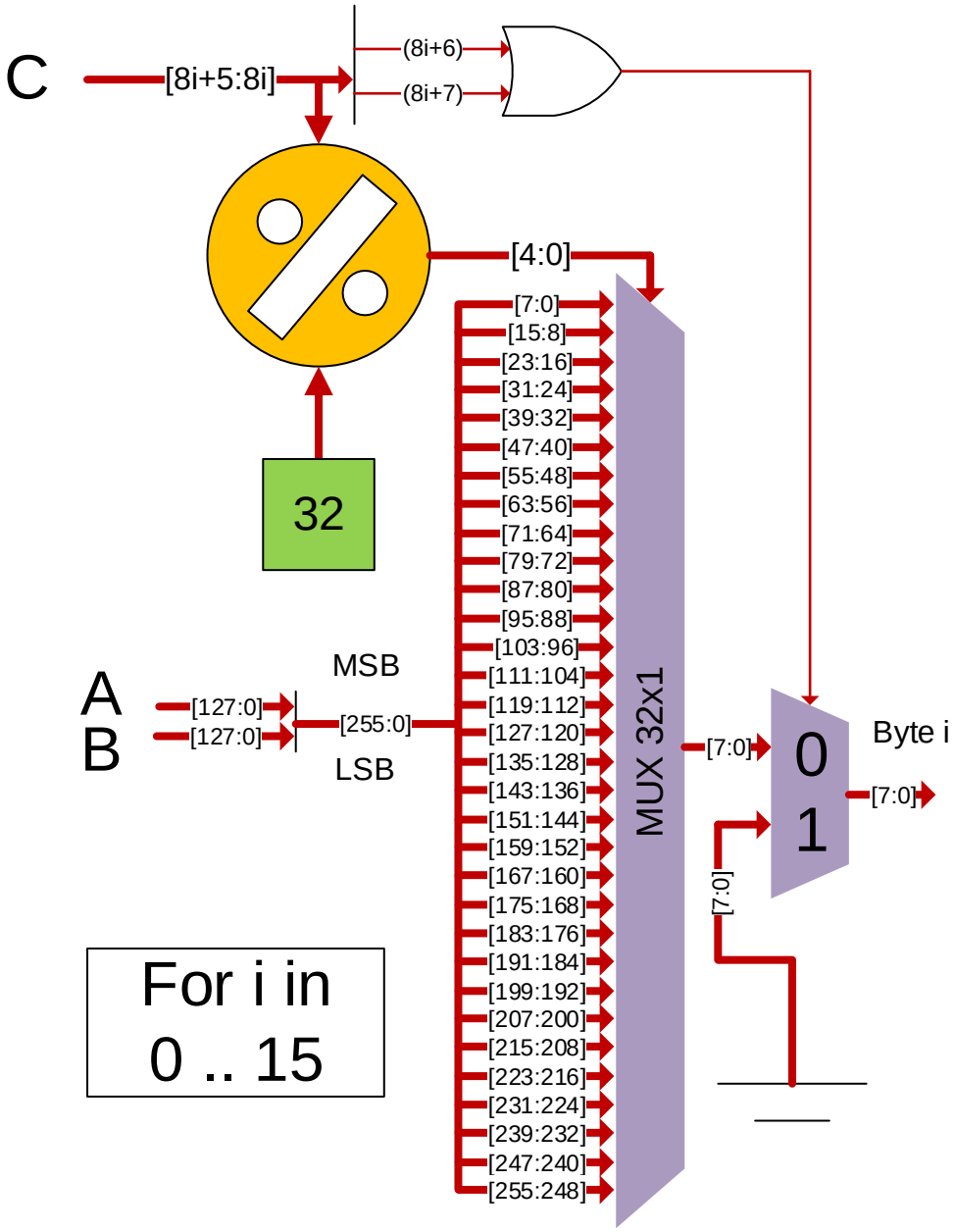


Figure 70: Implementation of VSHF unit Byte format

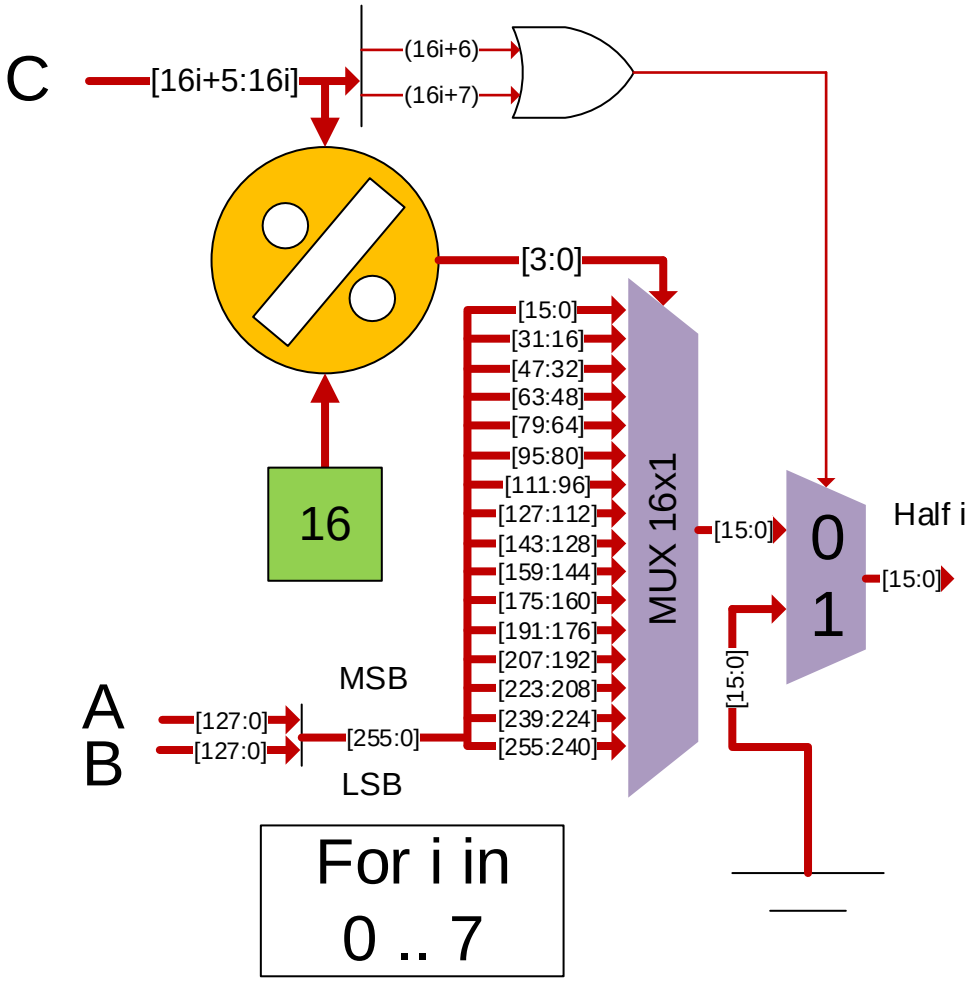


Figure 71: Implementation of the VSHF unit Halfword format

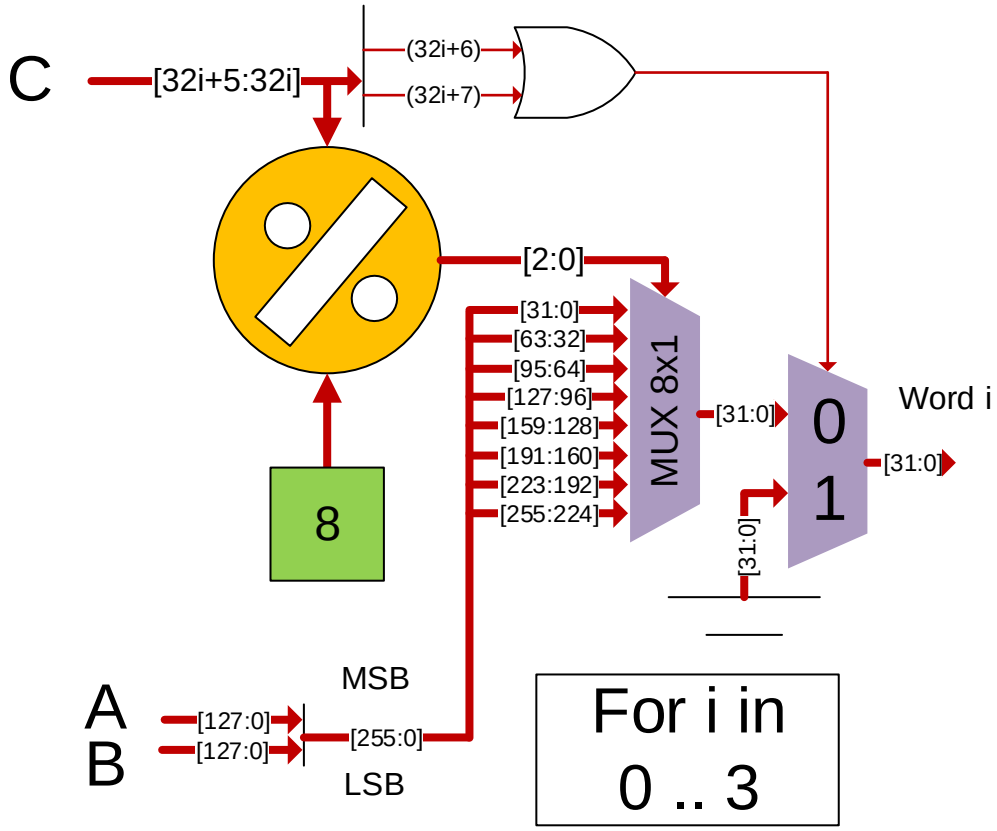


Figure 72: Implementation of the VSHF unit Word format

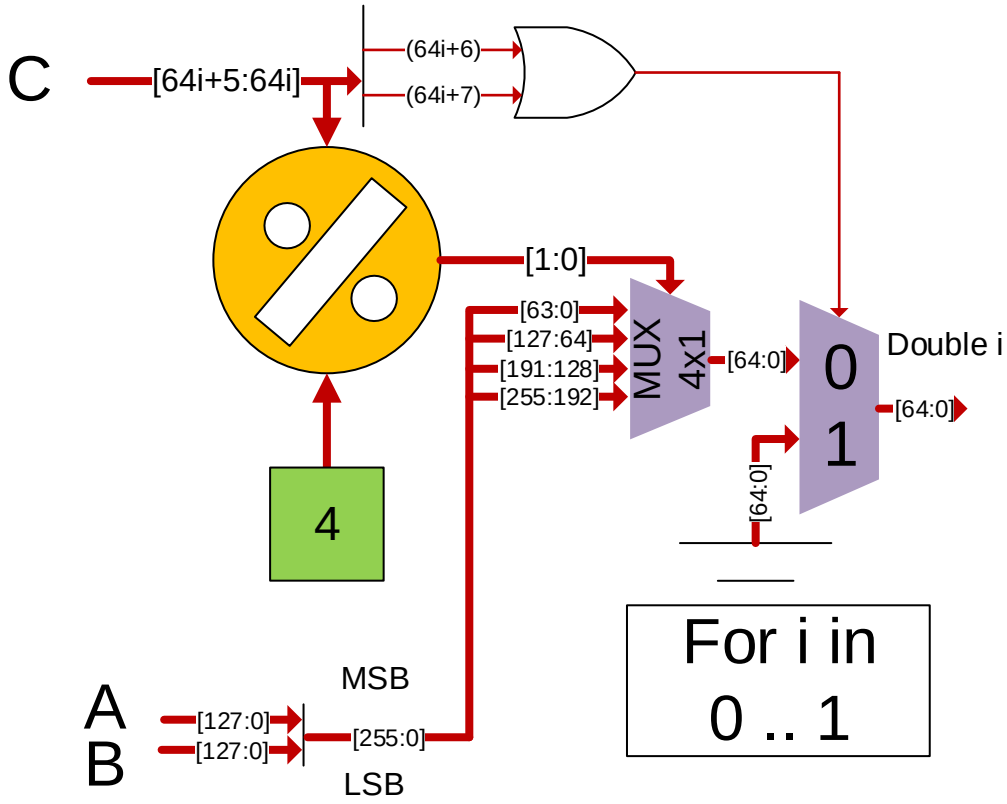


Figure 73: Implementation of the VSHF unit Doubleword format

16.6 SRLR unit

Figure 74 shows the SRLR unit. This unit perform vector shift right logical rounded vector bit count shift right logical with rounding. This unit is used to calculate instructions SRLR and SRLRI. To calculate SRLRI Special unit 2 is used. Figure 75, Figure 76, Figure 77 and Figure 78 shows the implementation of SRLR unit for each vector format.

The elements in vector A are shifted right logical by the number of bits the elements in vector B specify modulo the size of the element in bits. The most significant discarded bit is added to the shifted value (for rounding). The operands and results are values in integer data format specified by *df* field.

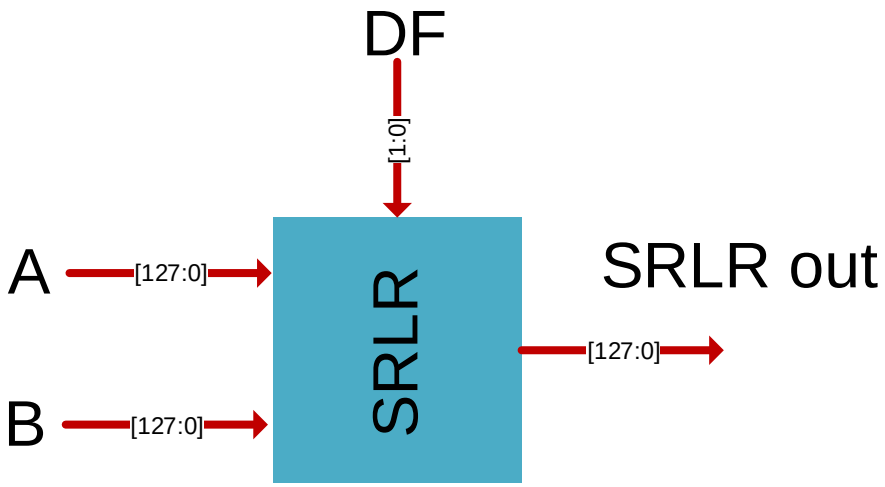


Figure 74: SRLR unit

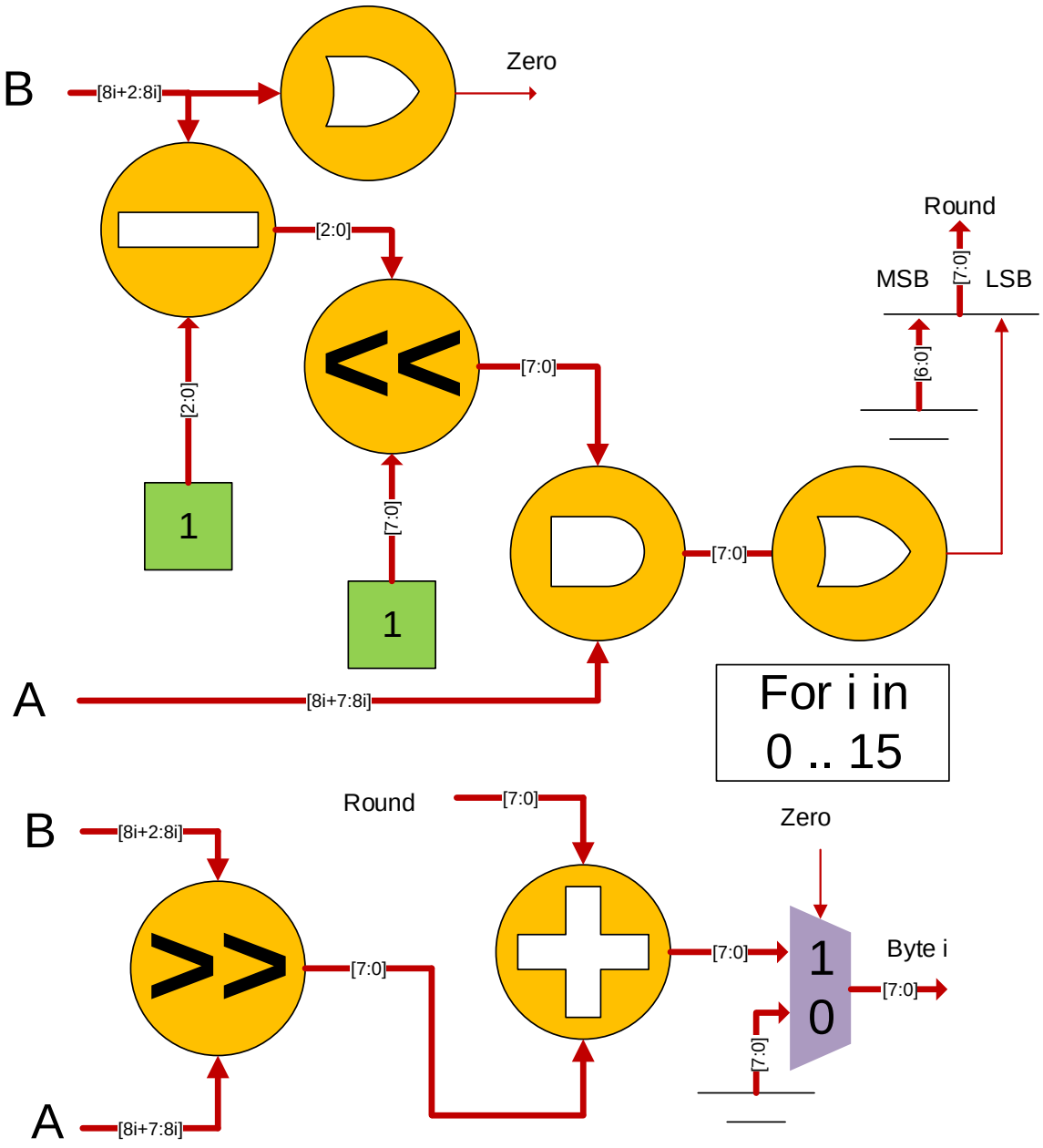


Figure 75: Implementation of the SRLR unit byte format

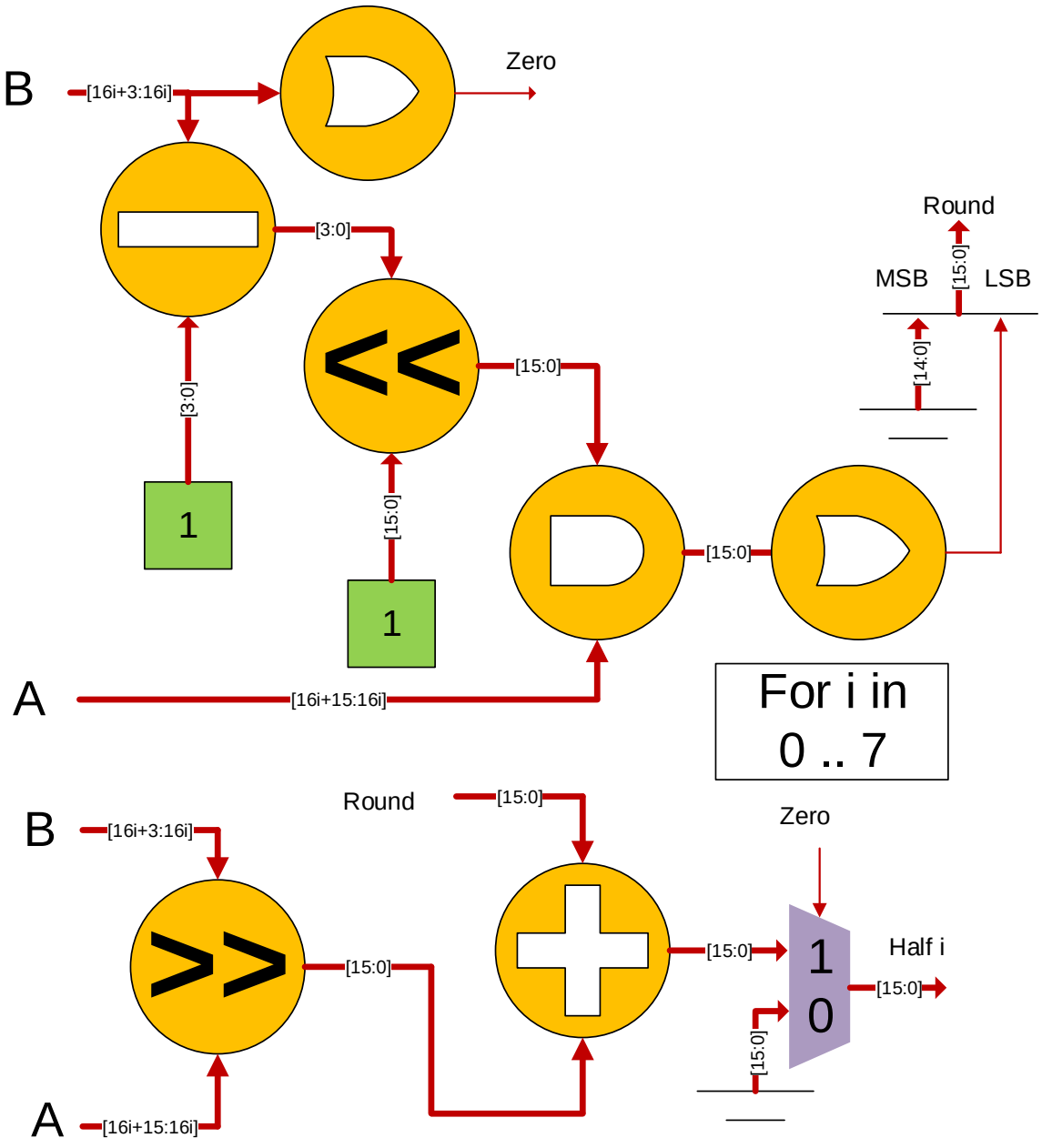


Figure 76: Implementation of the SRLR unit halfword format

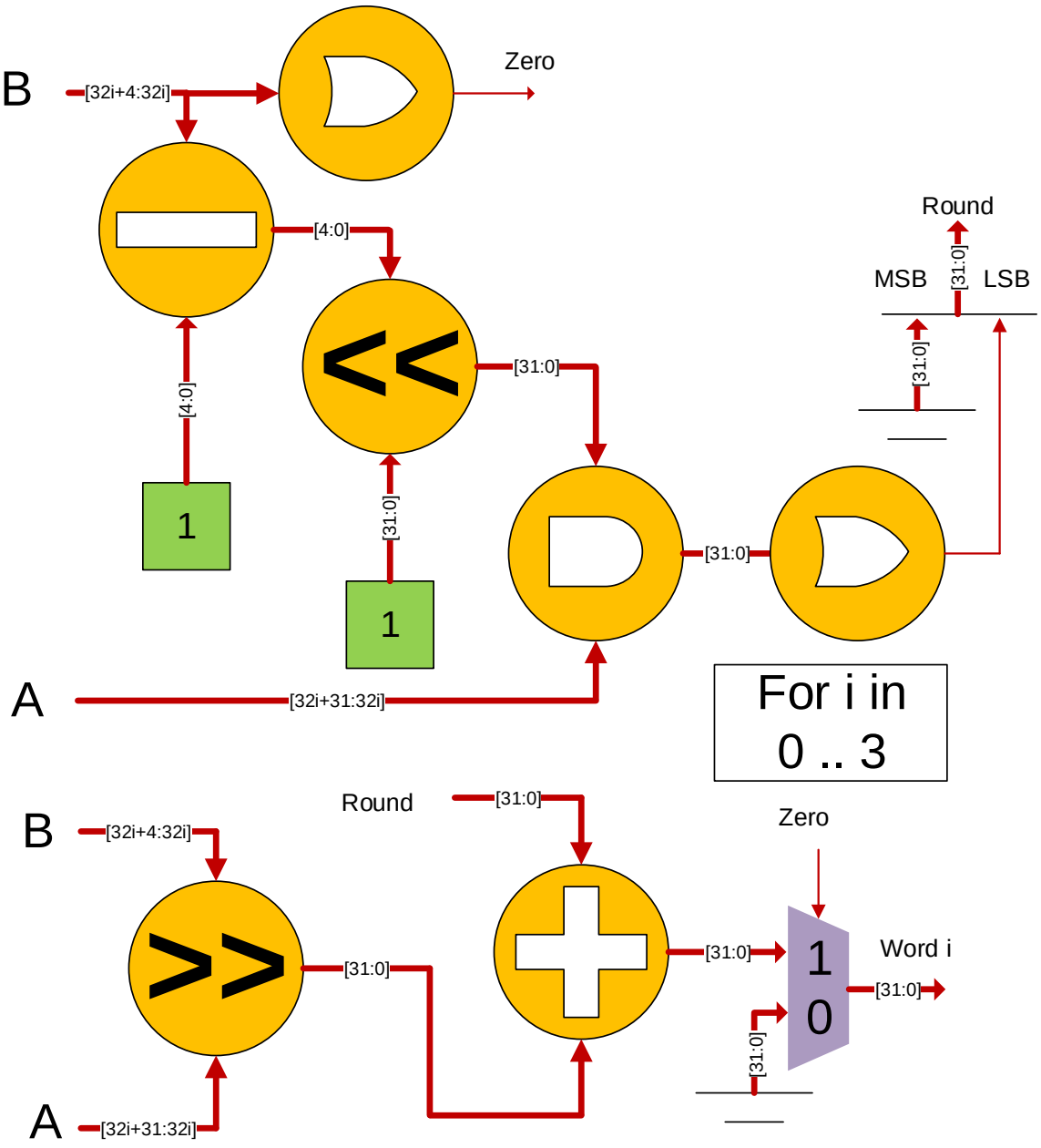


Figure 77: Implementation of the SRLR unit word format

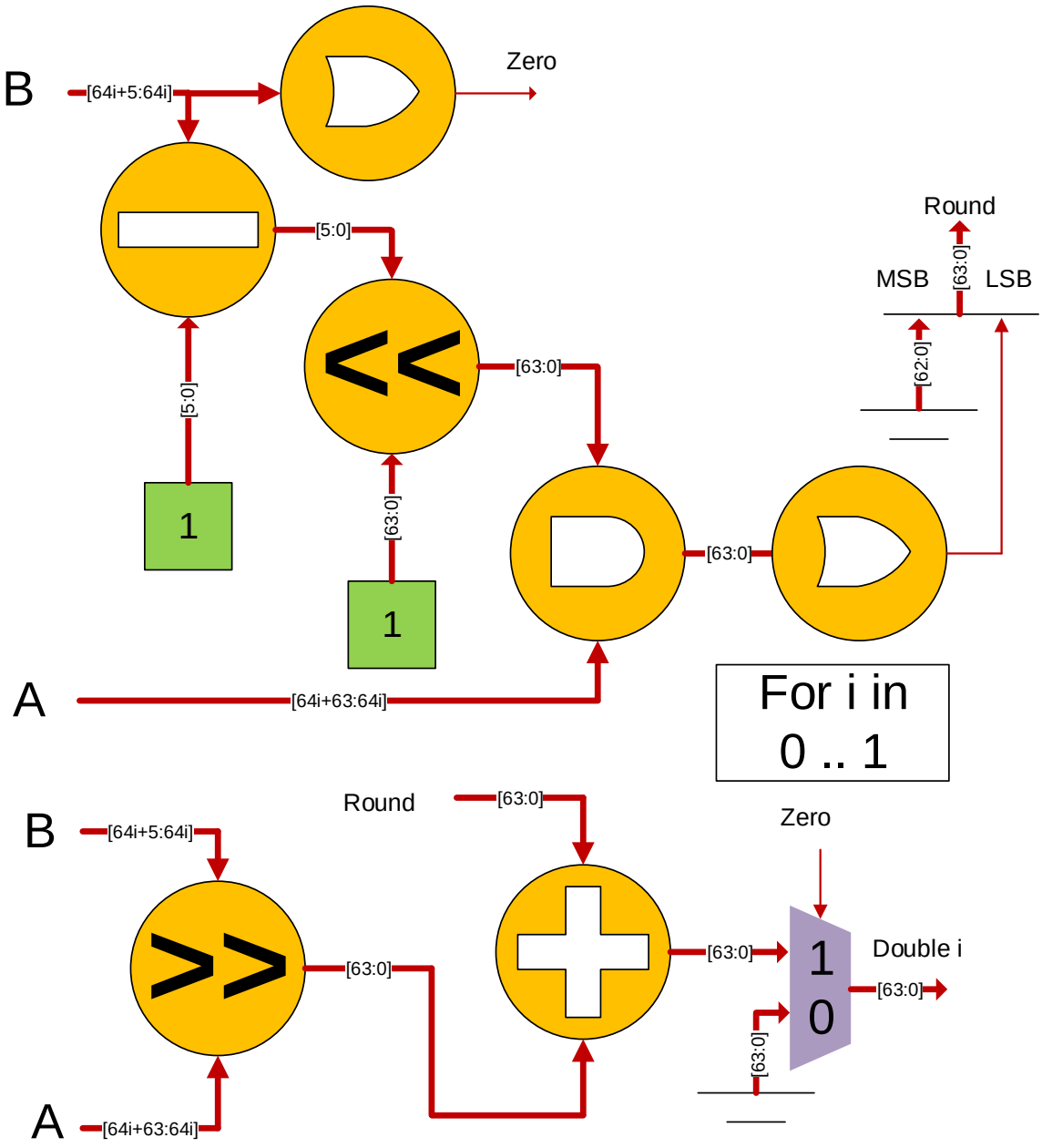


Figure 78: Implementation of the SRLR unit doubleword format

16.7 SRAR unit

Figure 79 shows the SRAR unit. This unit is used to calculate instructions SRAR and SRARI. To calculate SRARI Special unit 2 is used to. Figure 80, Figure 81, Figure 82 and Figure 83 shows the implementation of SRLR unit for each vector format.

The elements in vector A are shifted right arithmetic by the number of bits the elements in vector B specify modulo the size of the element in bits. The most significant discarded bit is added to the shifted value (for rounding). The operands and results are values in integer data format *df* field.

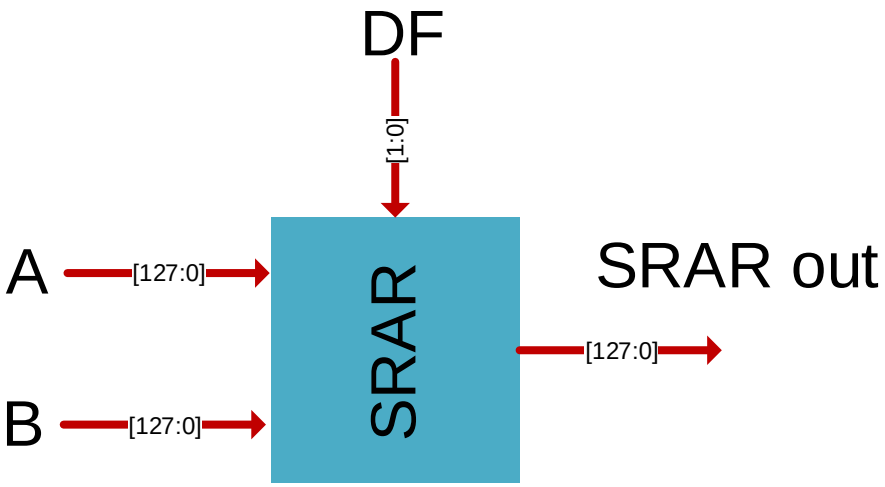


Figure 79: SRAR unit

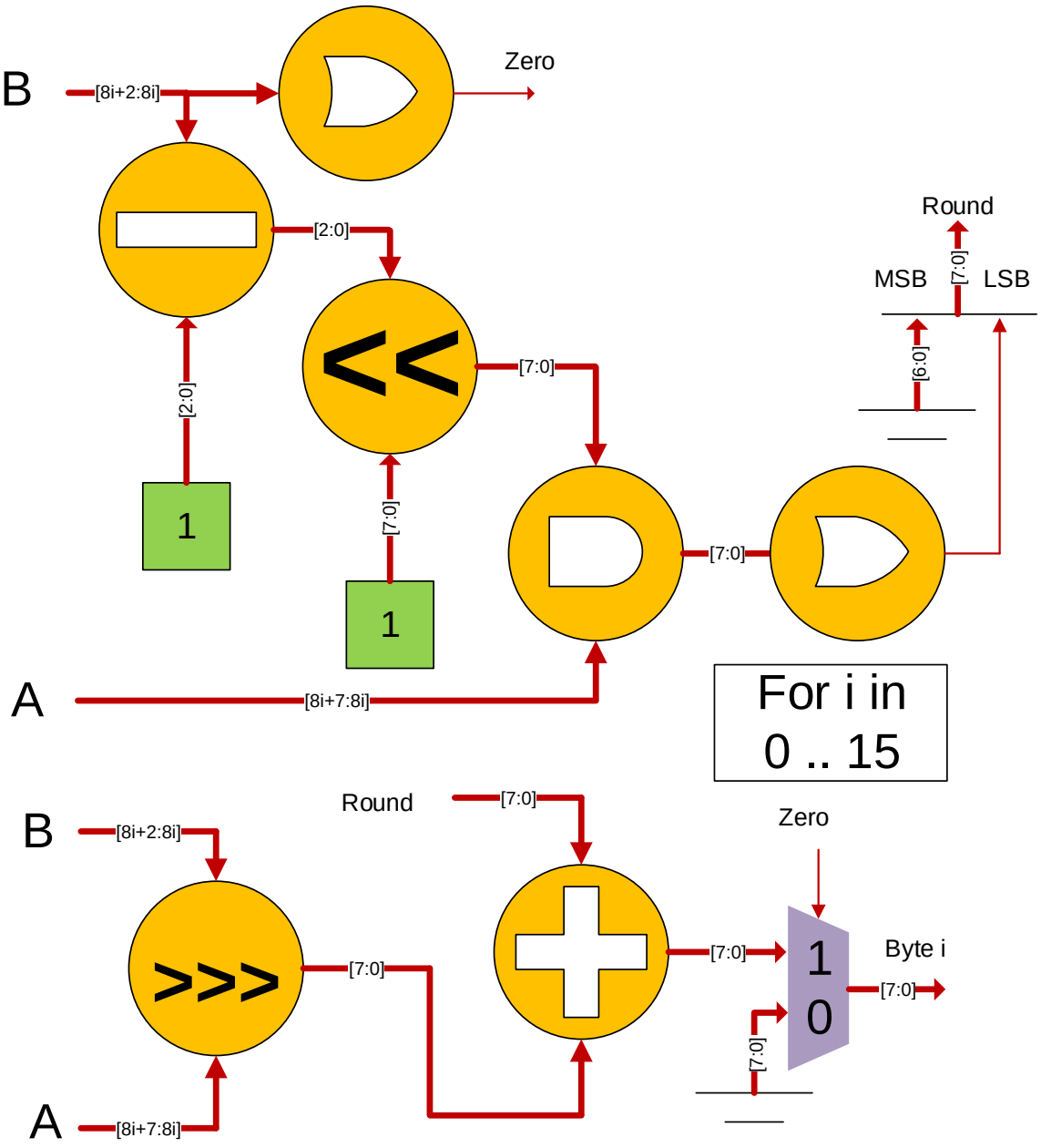


Figure 80: Implementation of the SRAR unit byte format

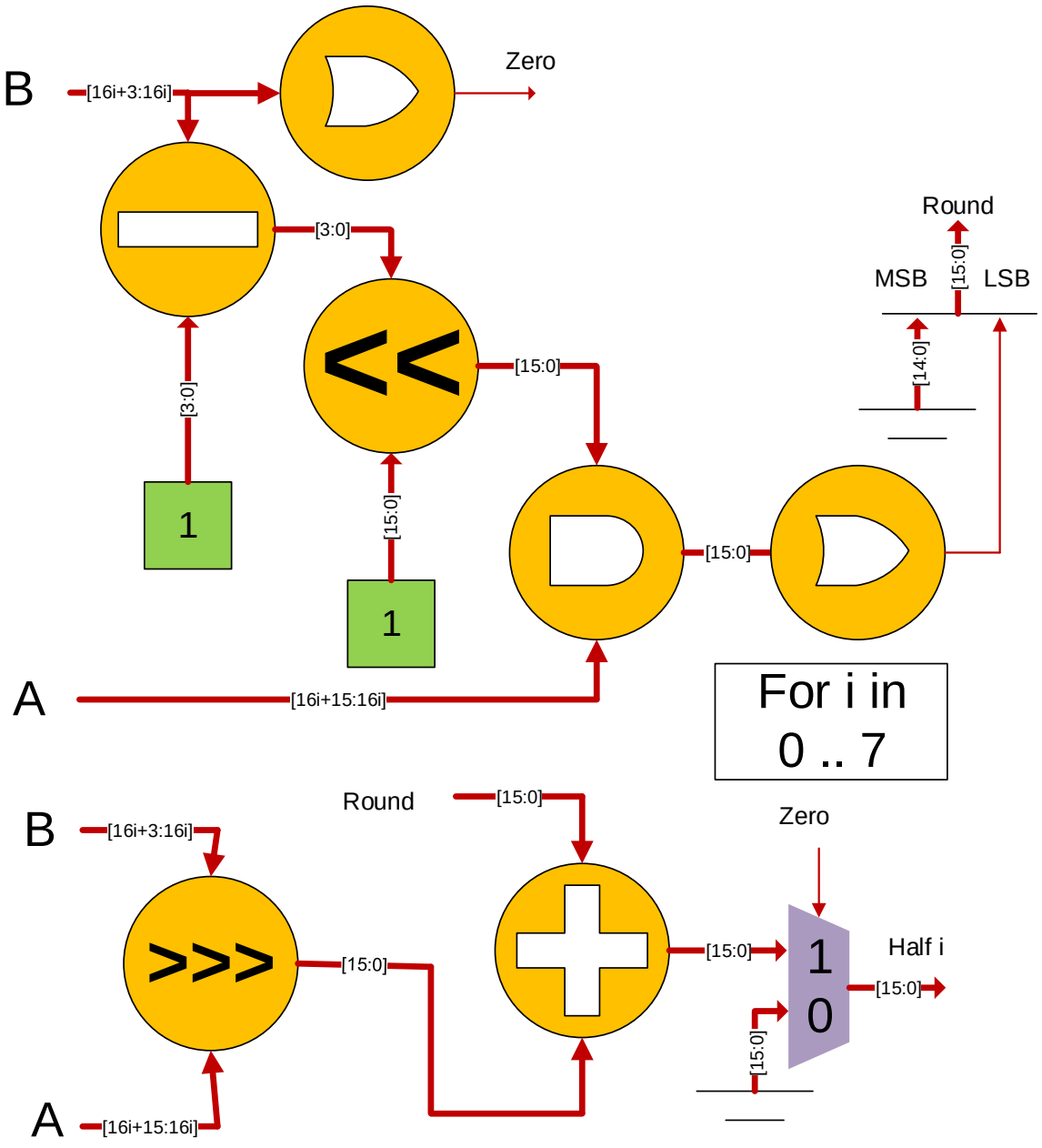


Figure 81: Implementation of the SRAR unit halfword format

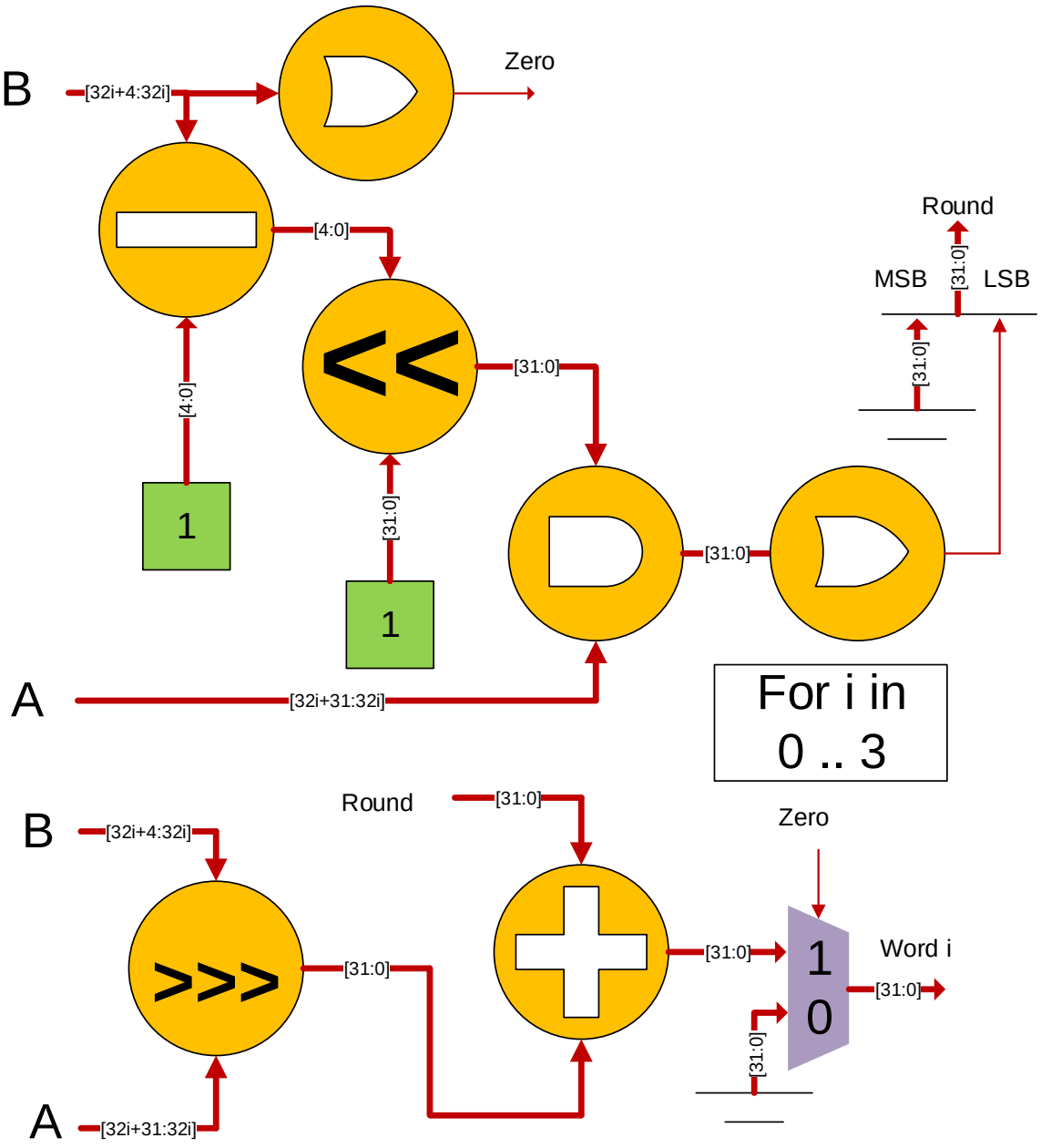


Figure 82: Implementation of SRAR unit word format

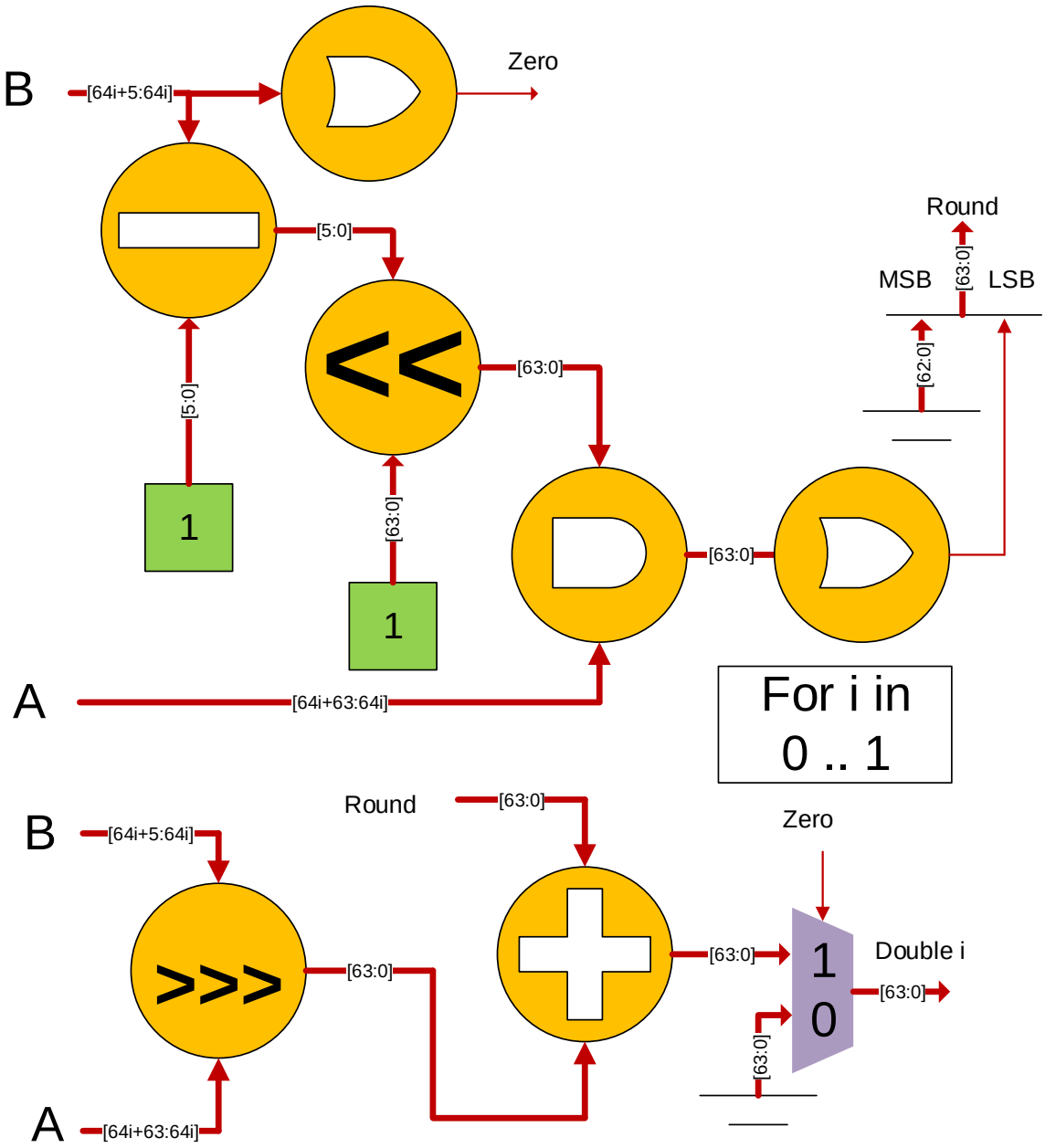


Figure 83: Implementation of the SRAR unit doubleword format

16.8 SLD unit

Figure 84 shows the interconnection of the SLD unit to one of the 64-bit wide 3R lane. This is done to use its 64-bit divider. So, we can calculate module between 64-bit general purpose value and one constant 16, 8, 4 or 2. Figure 85, Figure 86, Figure 87, Figure 88 and Figure 89 shows the SLD implementation for each vector format. Also using special unit 2 instruction SLDI I executed.

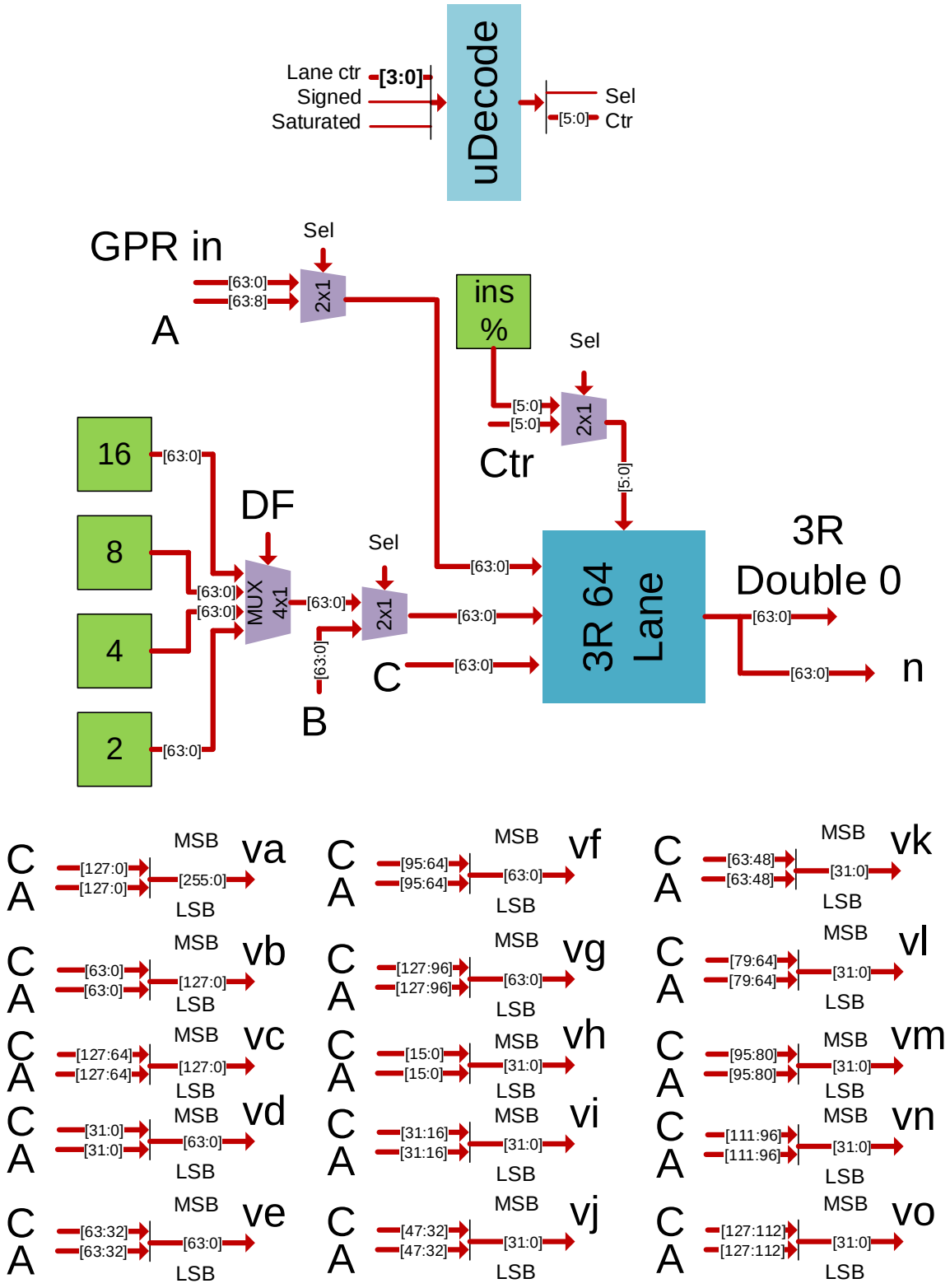


Figure 84: Implementation of SLD unit, input signal generation and "n" signal generation

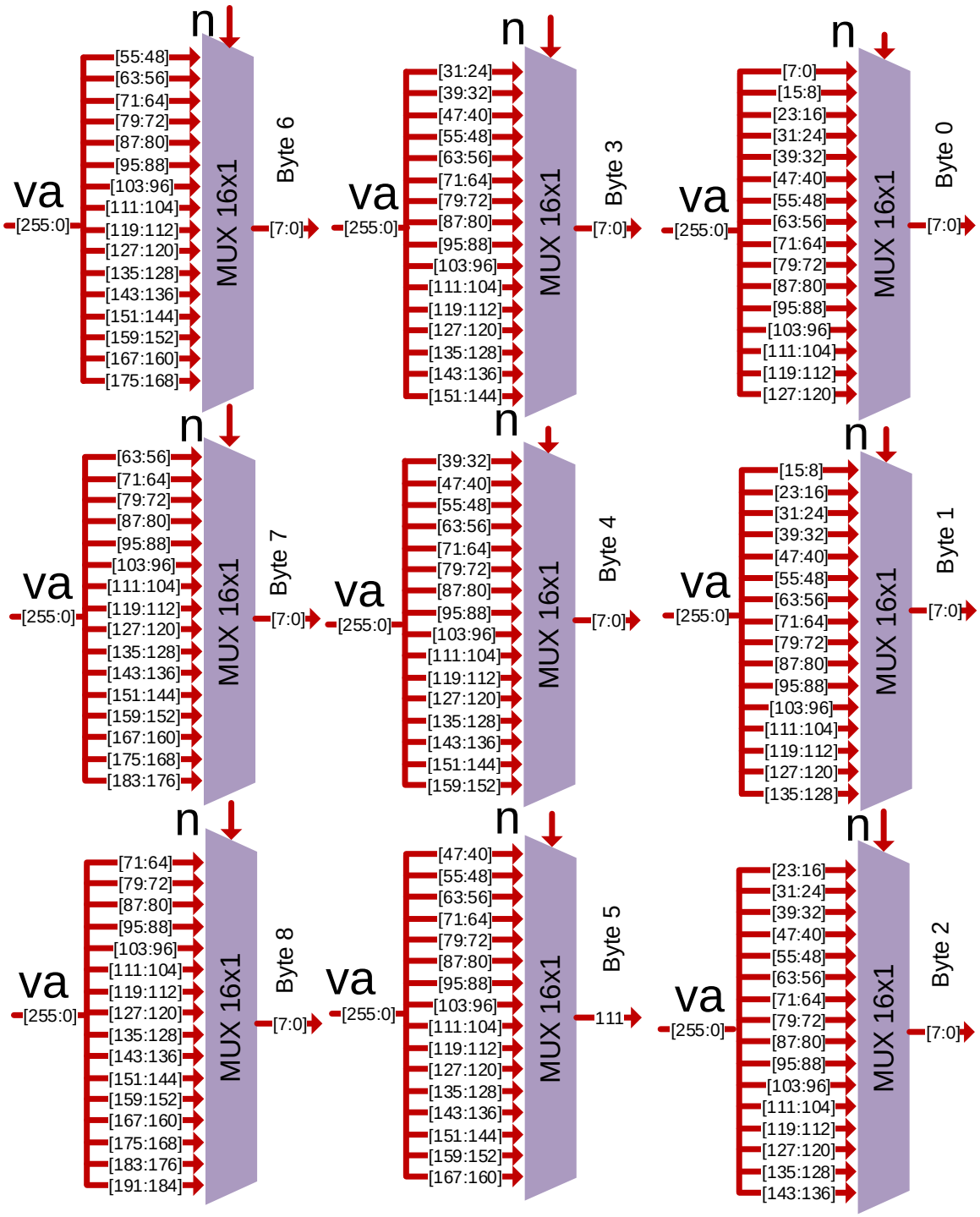


Figure 85: Implementation of the SLD unit byte format

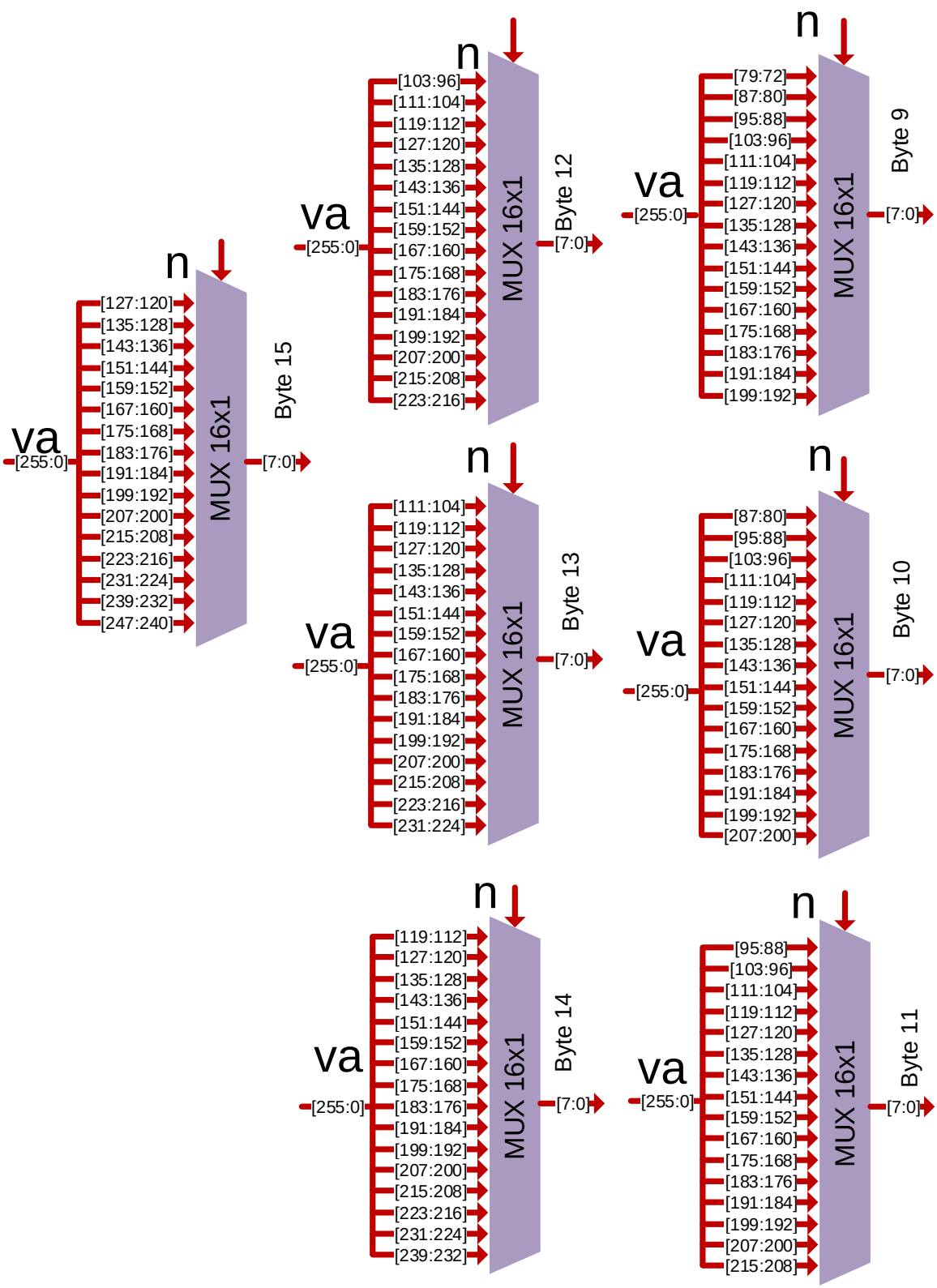


Figure 86: Implementation of the SLD unit byte format (cont.)

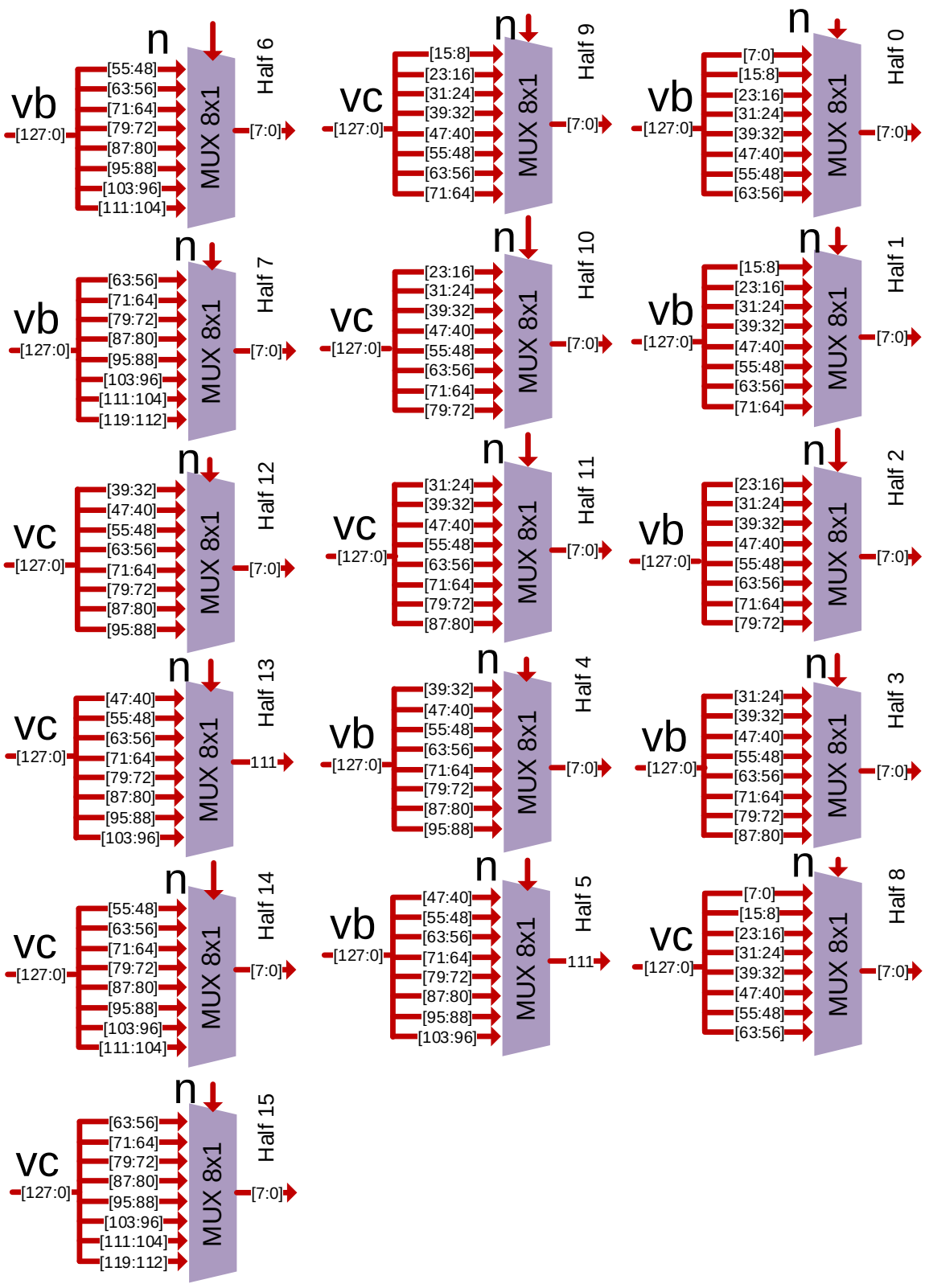


Figure 87: Implementation of the SLD unit halfword format

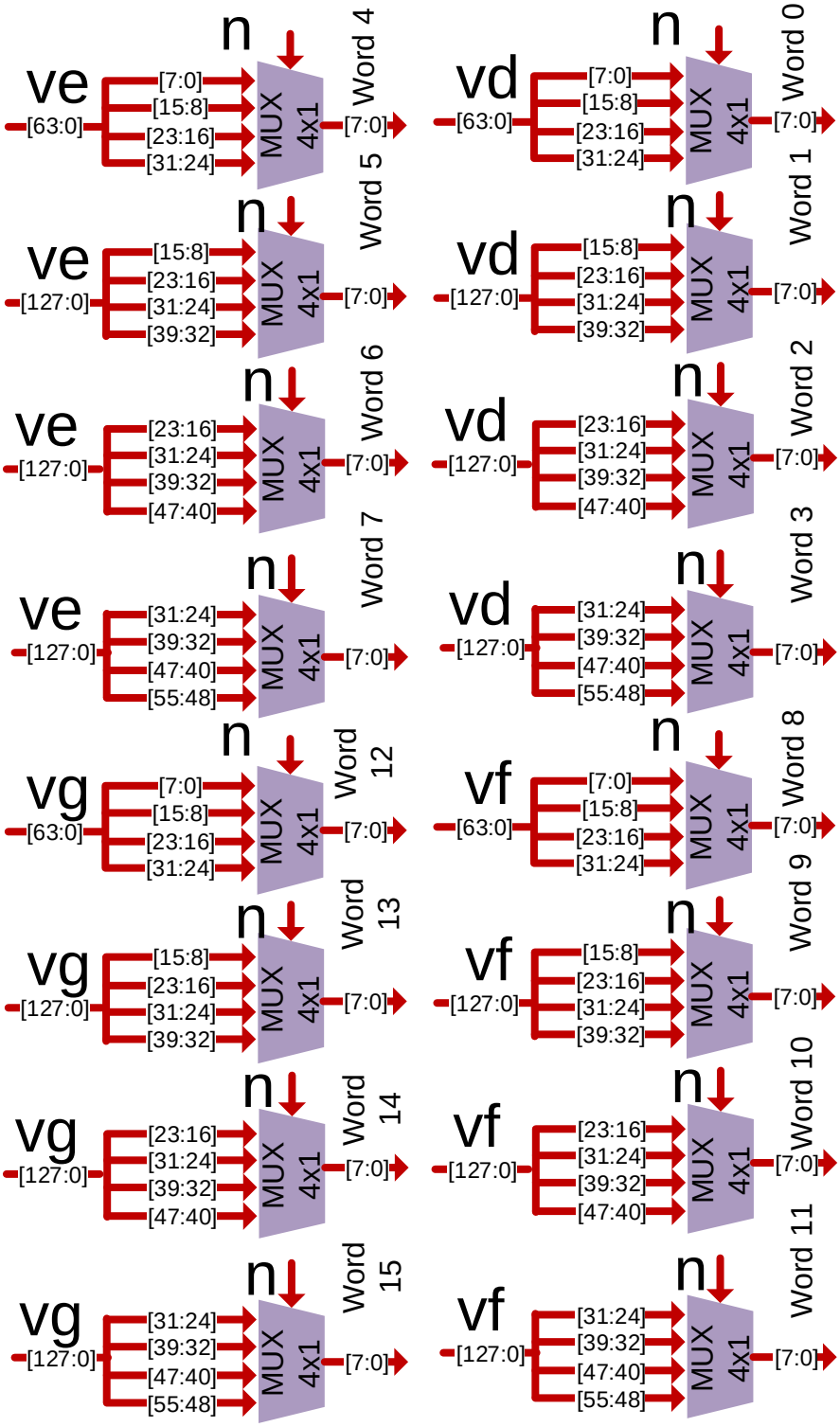


Figure 88: Implementation of the SLD unit word format

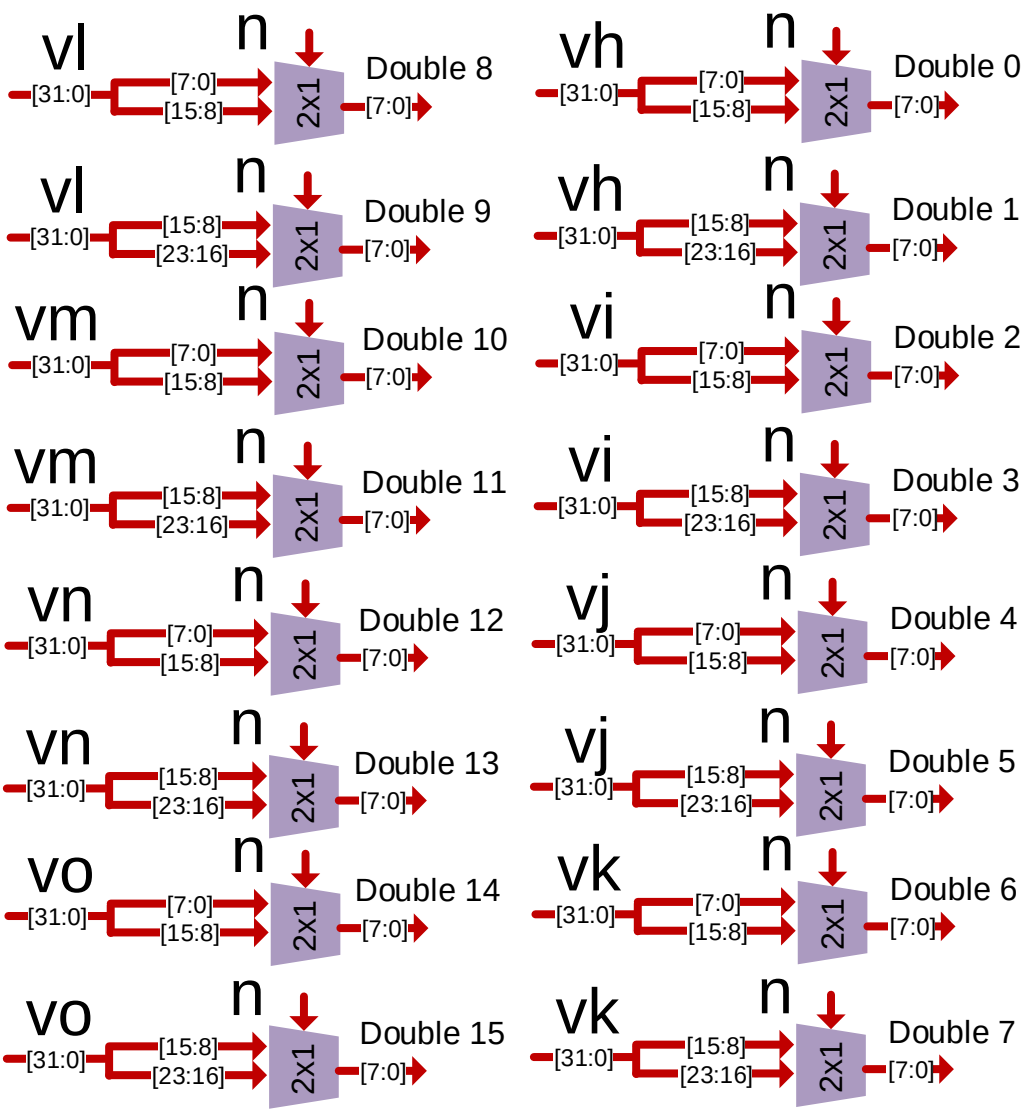


Figure 89: Implementation of the SLD unit doubleword format

16.9 SPLAT unit

Figure 90 show the implementation of SPLAT unit. It uses module calculated using the circuit show in Figure 84. Also SPLATI instruction is calculated using Special unit 2. These instruction replicates vector A element with index given by n (GPR module) to all elements in vector C. GPR value is interpreted modulo the number of data format *df* elements in the destination vector. The operands and results are values in data format *df*.

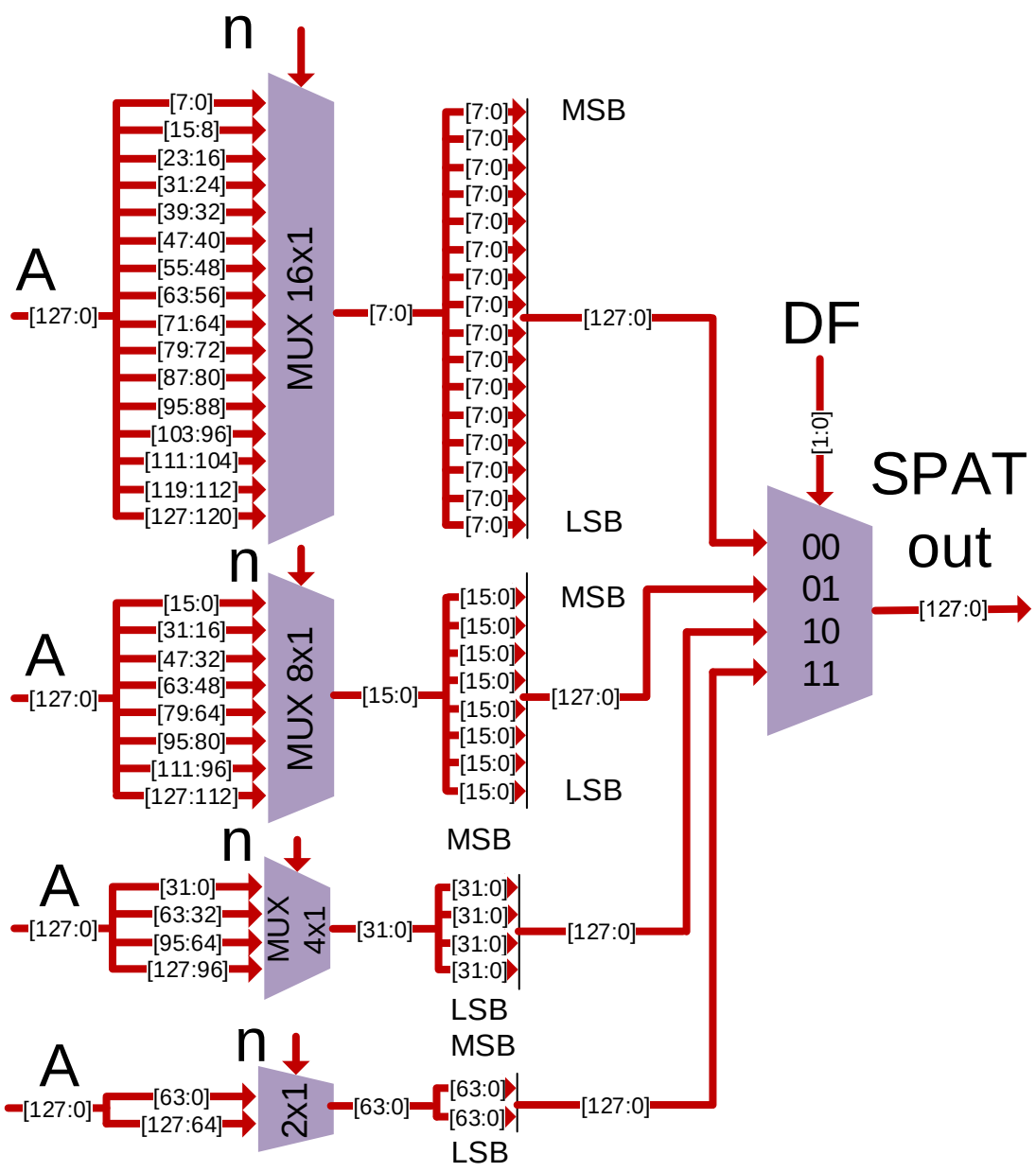


Figure 90: SPLAT unit implementation

16.10 PCKEV unit

Figure 91 shows the implementation of PCKEV unit for all 4 vector formats. This unit calculates the PCKEV instruction. Even elements in vector A are copied to the left half of vector result and even elements in vector B are copied to the right half of vector result. The operands and results are values in integer data format *df* field.

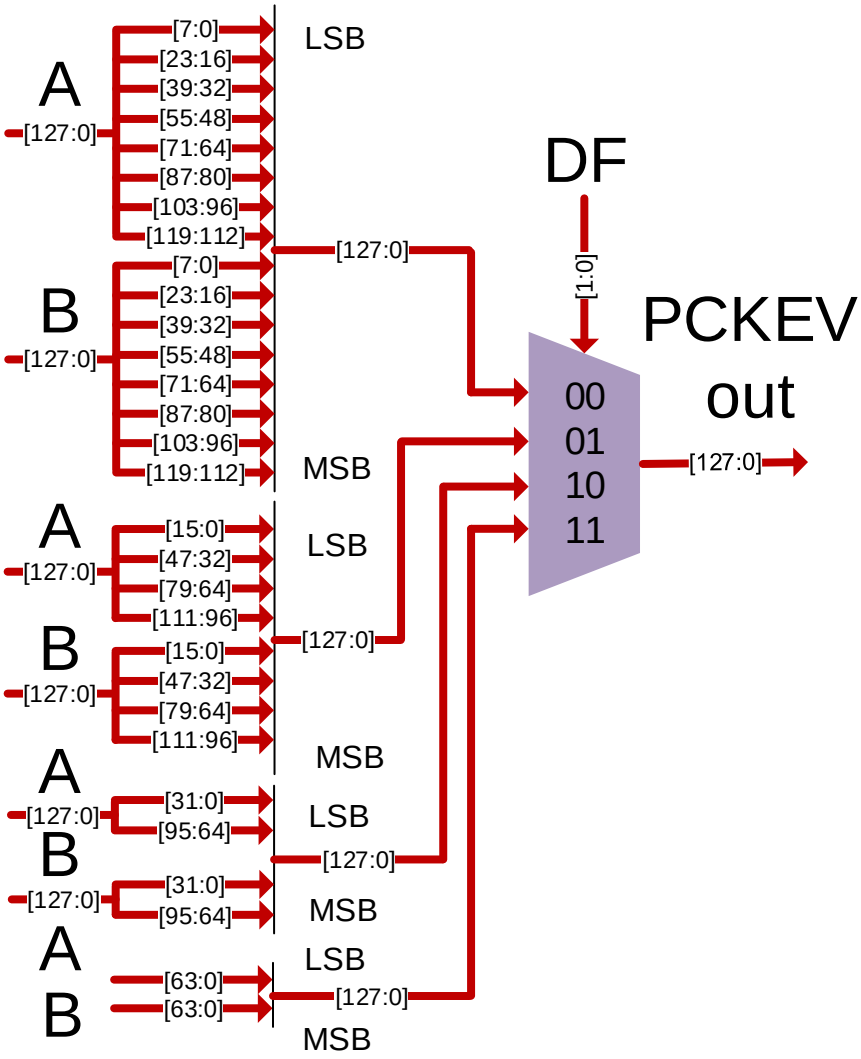


Figure 91: PCKEV implementation

16.11 PCKOD unit

Figure 92 shows the implementation of PCKOD unit for all 4 vector formats. This unit calculates the PCKOD instruction. Odd elements in vector A are copied to the left half of vector result and odd elements in vector B are copied to the right half of vector result. The operands and results are values in integer data format *df* field.

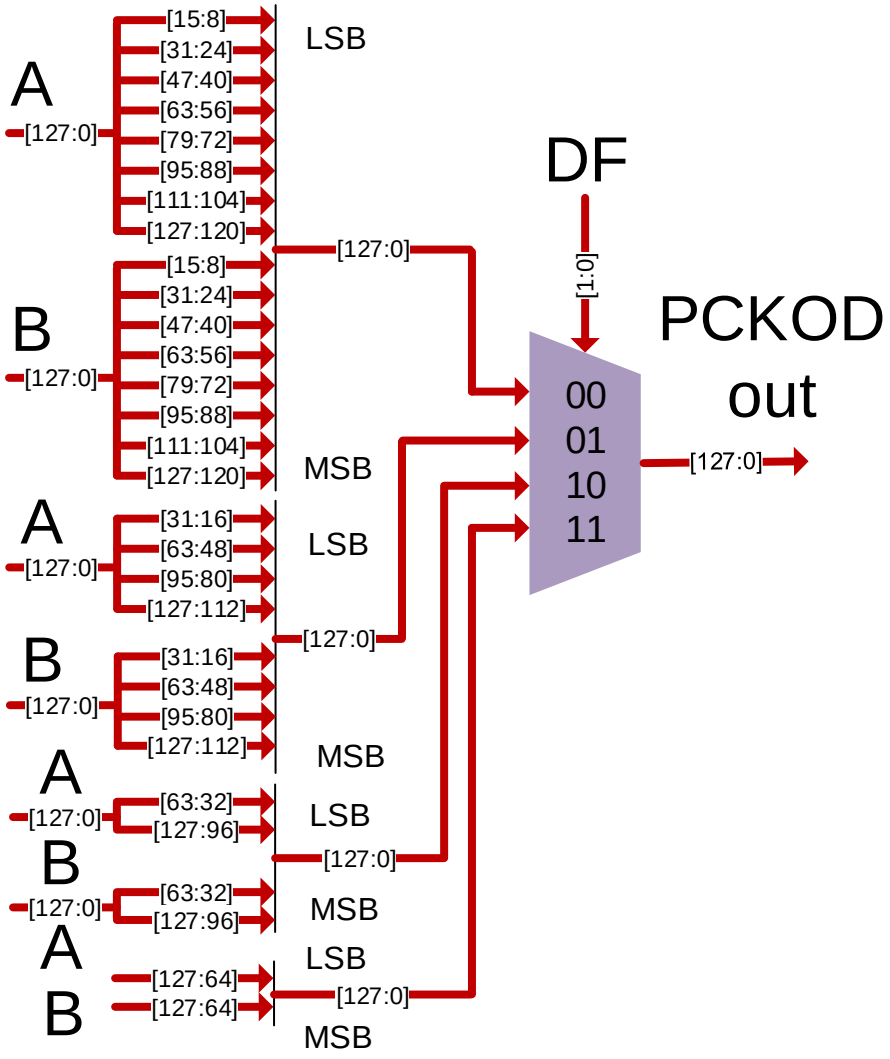


Figure 92: PCKOD implementation

16.12 ILVL unit

Figure 93 shows the implementation of ILVL unit for all 4 vector formats. This unit calculates the ILVL instruction. The left half elements in vectors A and B are copied to vector result alternating one element from A with one element from B. The operands and results are values in integer data format *df* field.

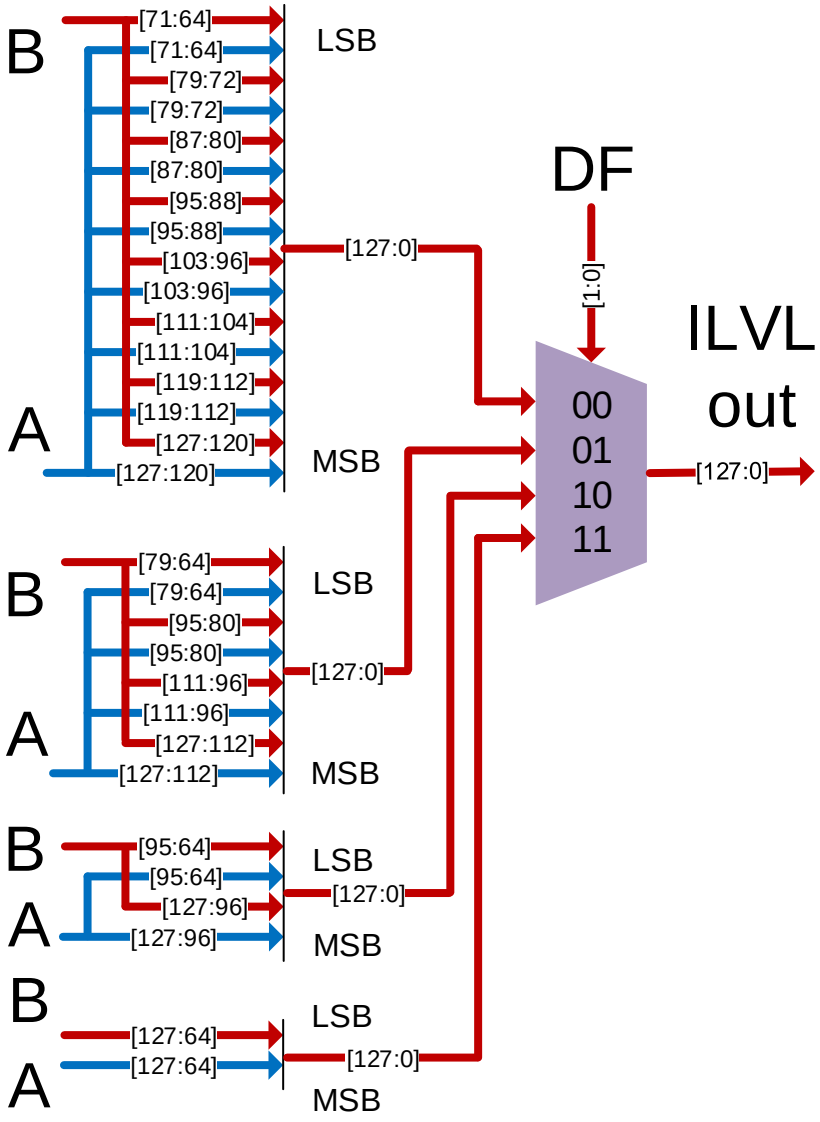


Figure 93: ILVL unit implementation

16.13 ILVR unit

Figure 94 shows the implementation of the ILVR unit for all 4 vector formats. This unit calculates the ILVR instruction. The right half elements in vectors A and B are copied to vector result alternating one element from A with one element from B. The operands and results are values in integer data format *df* field.

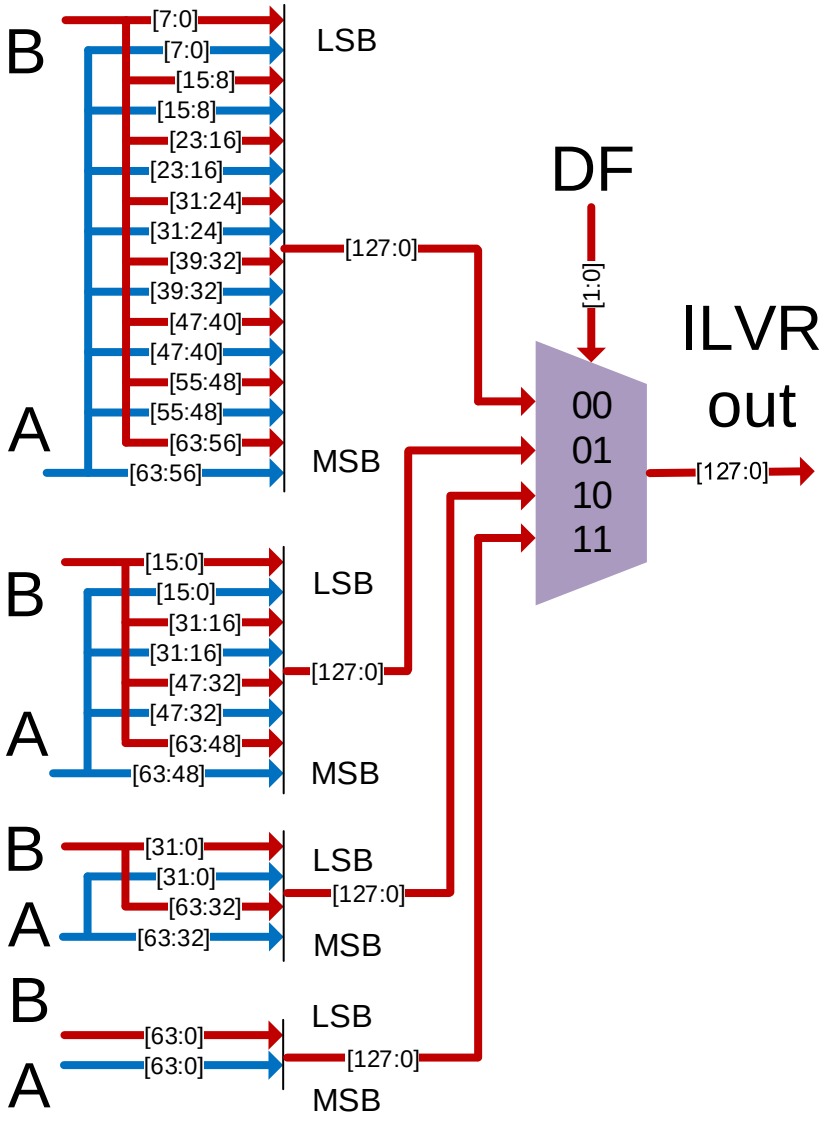


Figure 94: ILVR unit implementation

16.14 ILVEV unit

Figure 95 shows the implementation of ILVEV unit for all 4 vector formats. This unit calculates the ILVEV instruction. Even elements in vectors A and B are copied to vector result alternating one element from A with one element from B. The operands and results are values in integer data format *df* field.

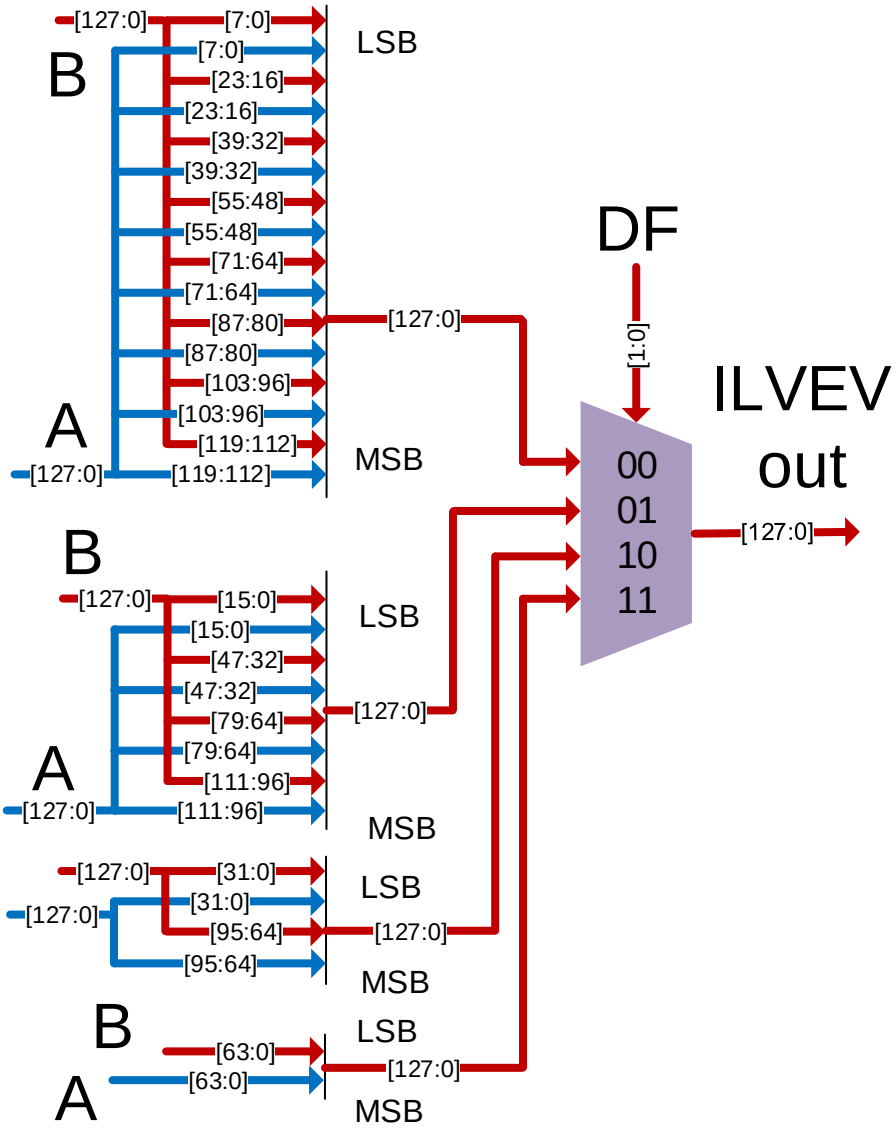


Figure 95: ILVEV unit implementation

16.15 ILVOD unit

Figure 96 show the implementation of the ILVOD unit for all 4 vector formats. This unit executes the instruction ILVOD. Odd elements in vectors A and B are copied to vector result alternating one element from A with one element from B. The operands and results are values in integer data format *df* field.

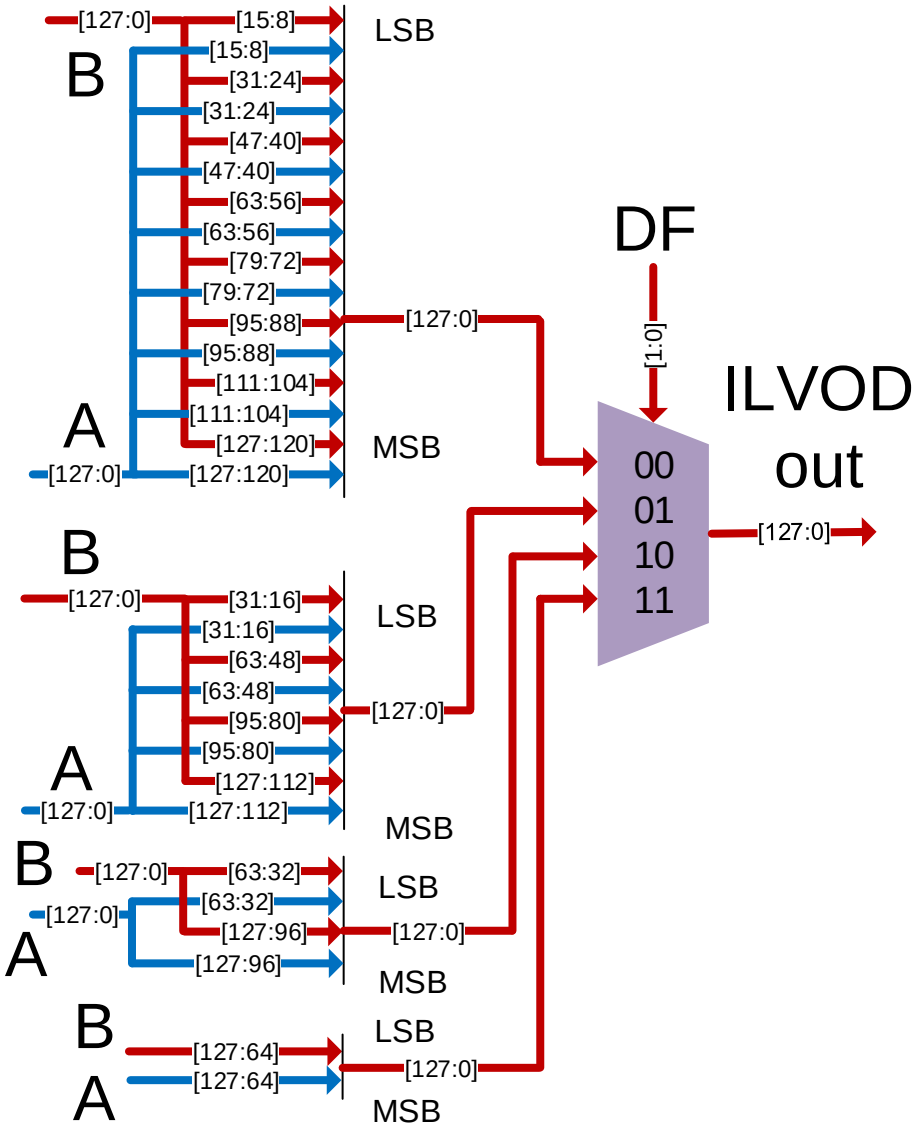


Figure 96: Implementation of ILVOD unit

16.16 Insert unit

Figure 97 shows the implementation of unit insert for all 4 formats. This unit is used to execute the instruction Insert. This instruction takes a value from the general purpose register and insert this value in one element of vector read form port C of the vector register file.

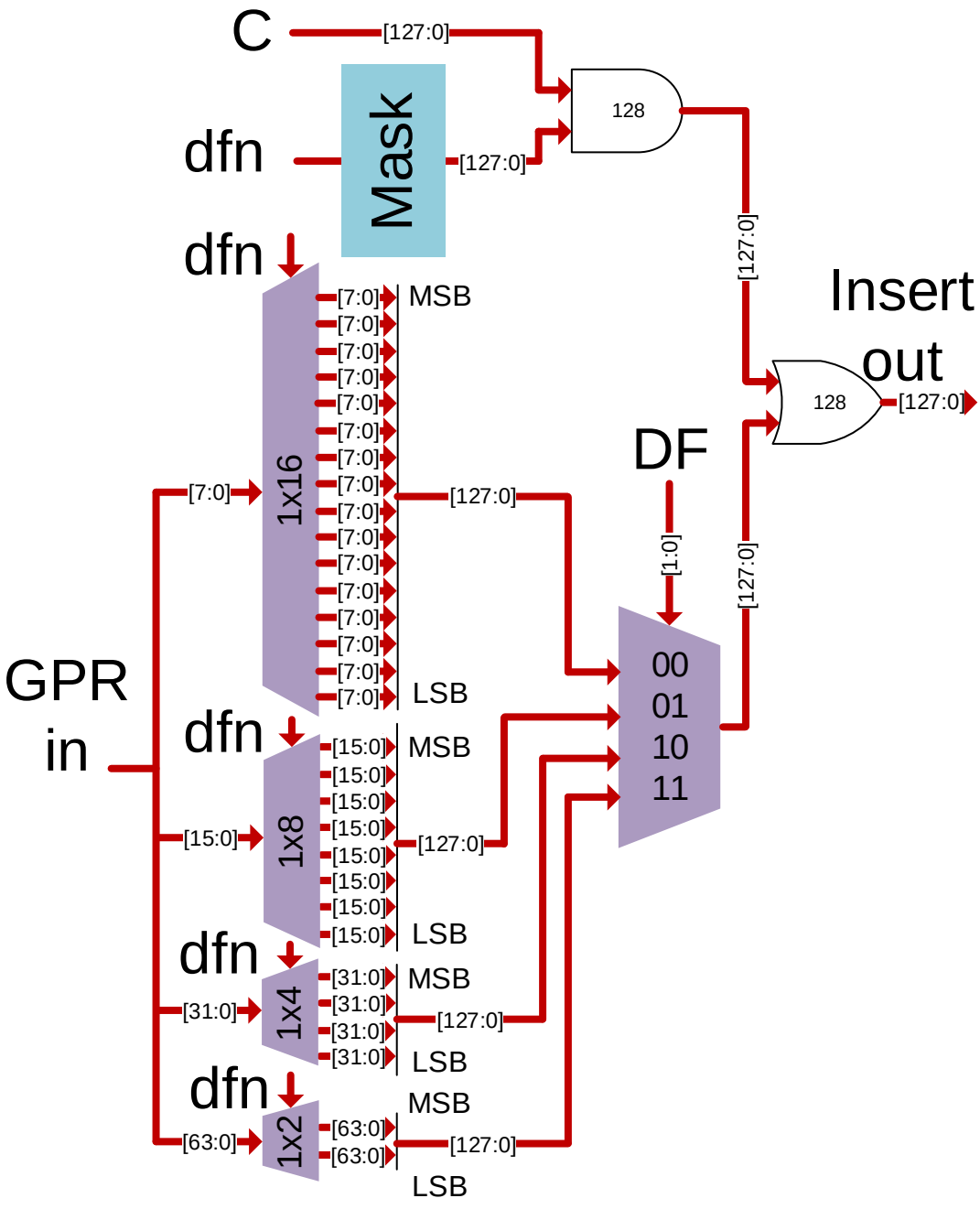


Figure 97: Implementation of Insert unit

16.17 INSVE unit

Figure 98 shows the implementation of Insve unit. This is similar to insert unit but this one takes an element from vector A and insert it on vector C.

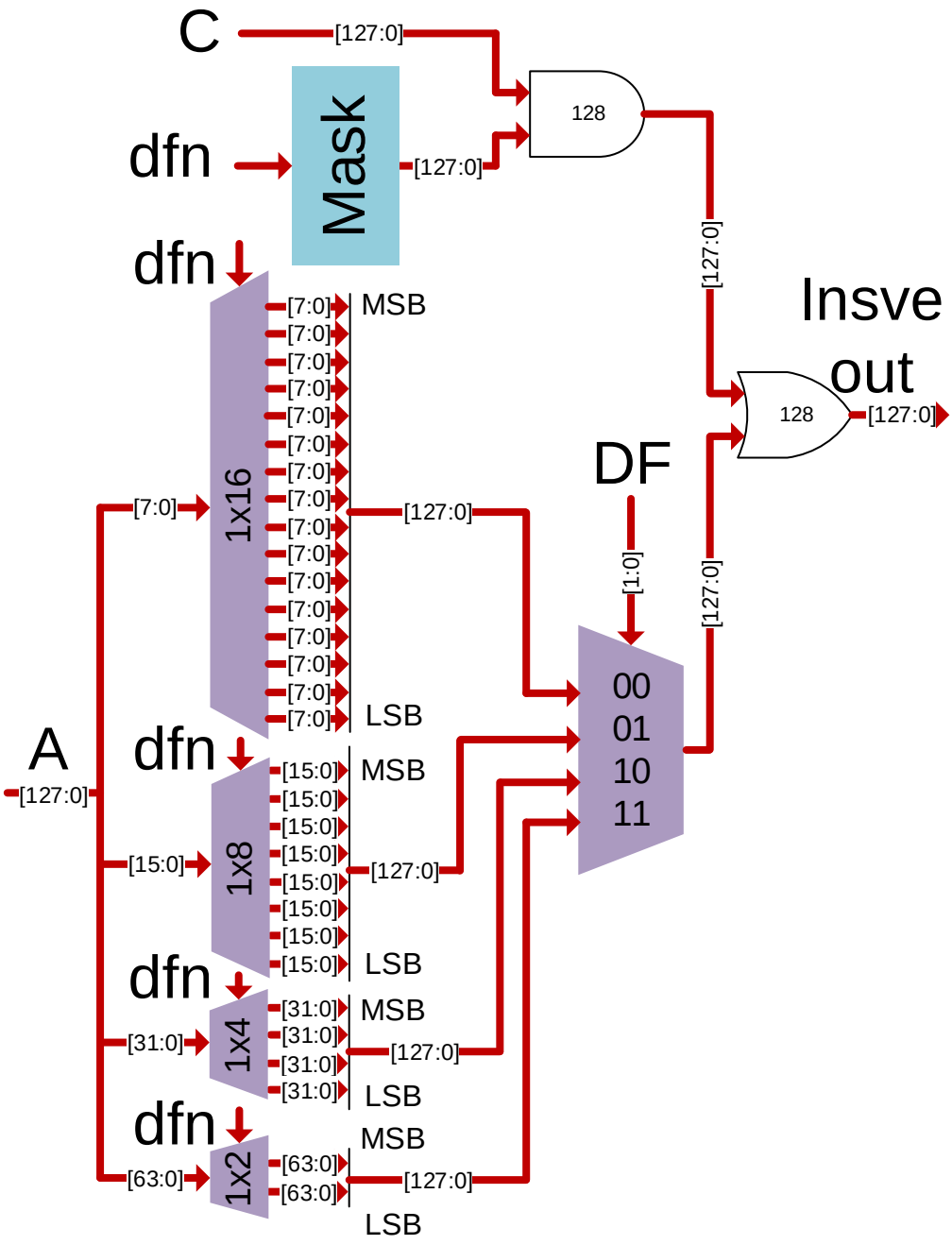


Figure 98: Implementation of Insve unit

16.18 Dot Product unit

Figure 99, Figure 100 and Figure 101 shows the implementation of dot product unit for formats *halfword*, *word* and *doubleword*. Figure 102 shows the multiplexor that choose one format result. This unit executes instructions DOTP_S, DOTP_U. DPADD_S, DPADD_U, DPSUB_S and DPSUB_U.

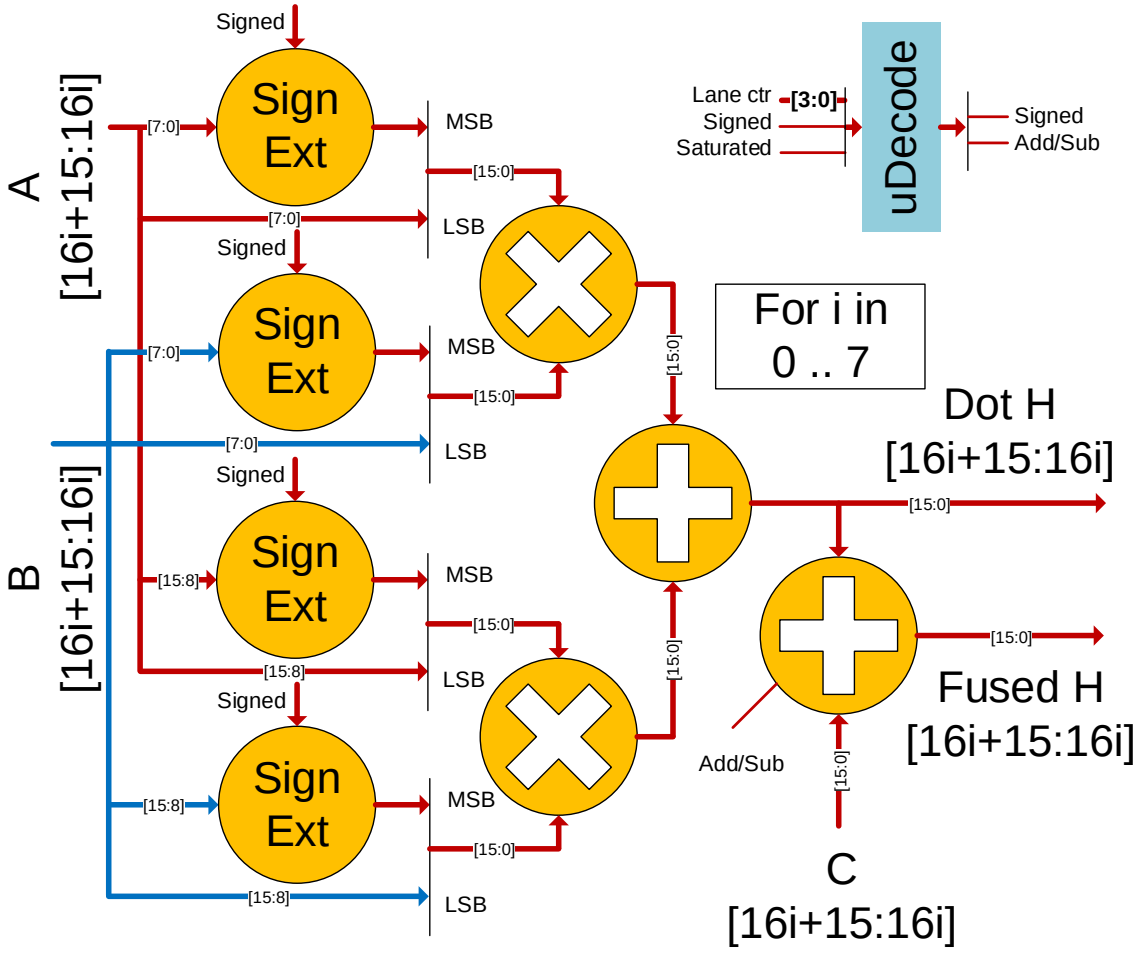


Figure 99: Implementation of Dotproduct unit for halword format

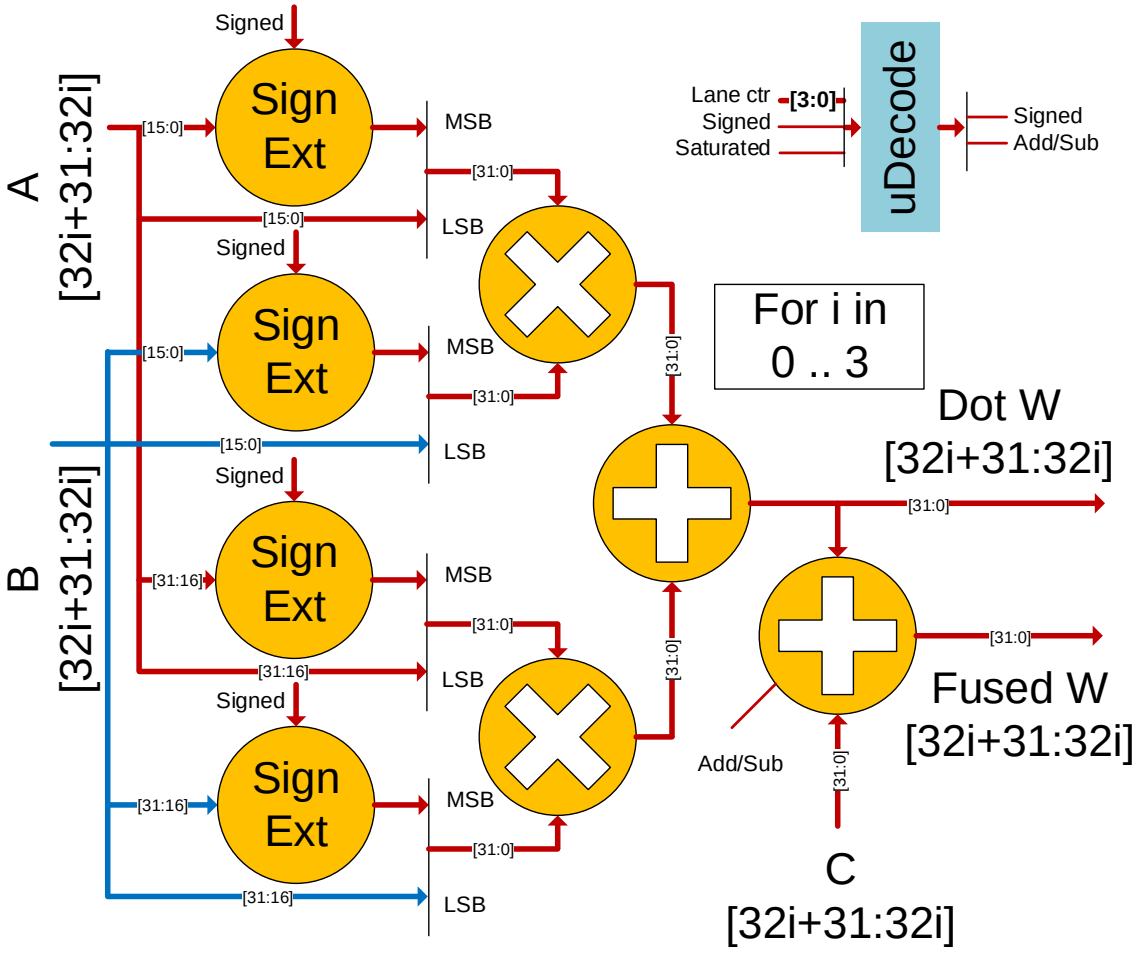


Figure 100: Implementation of Dotproduct for word format

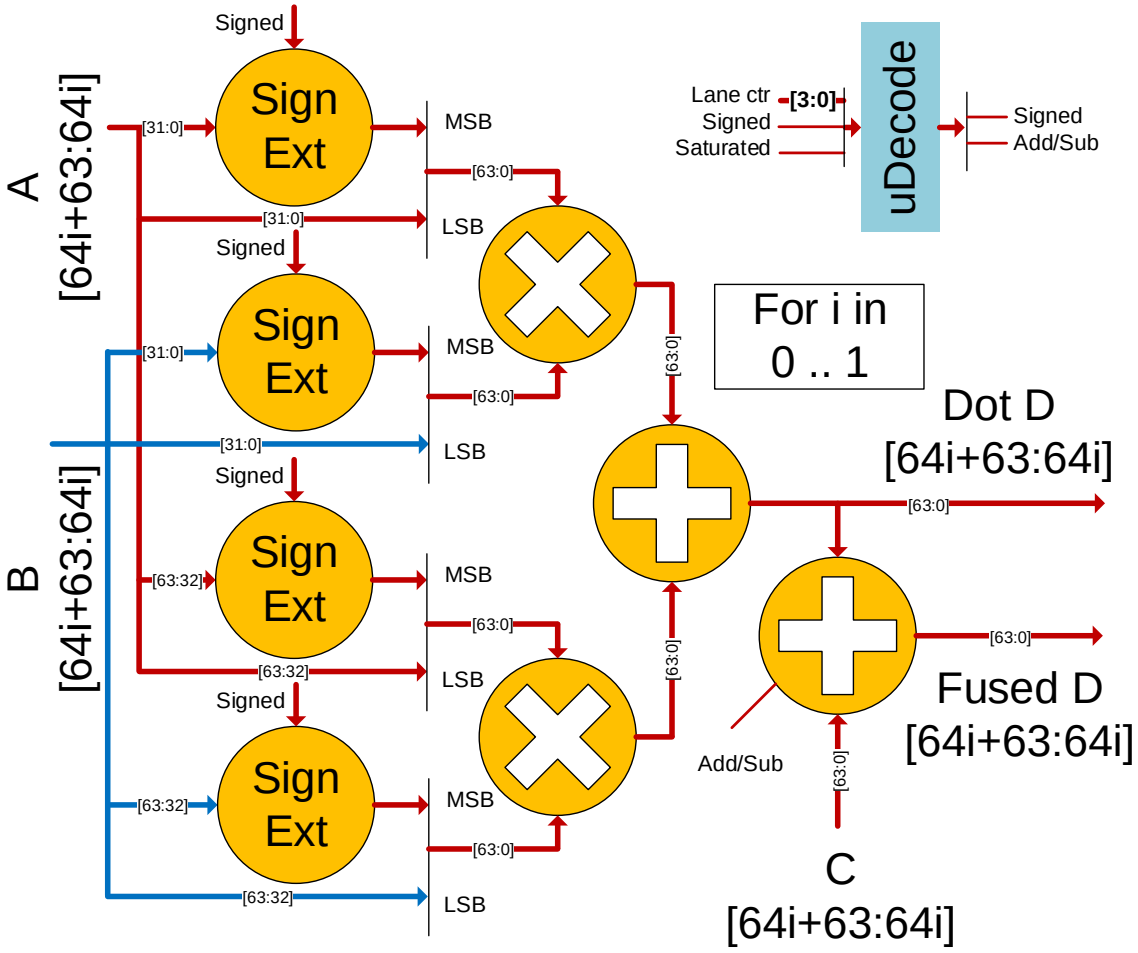


Figure 101: Implementation of Dotproduct for doubleword format

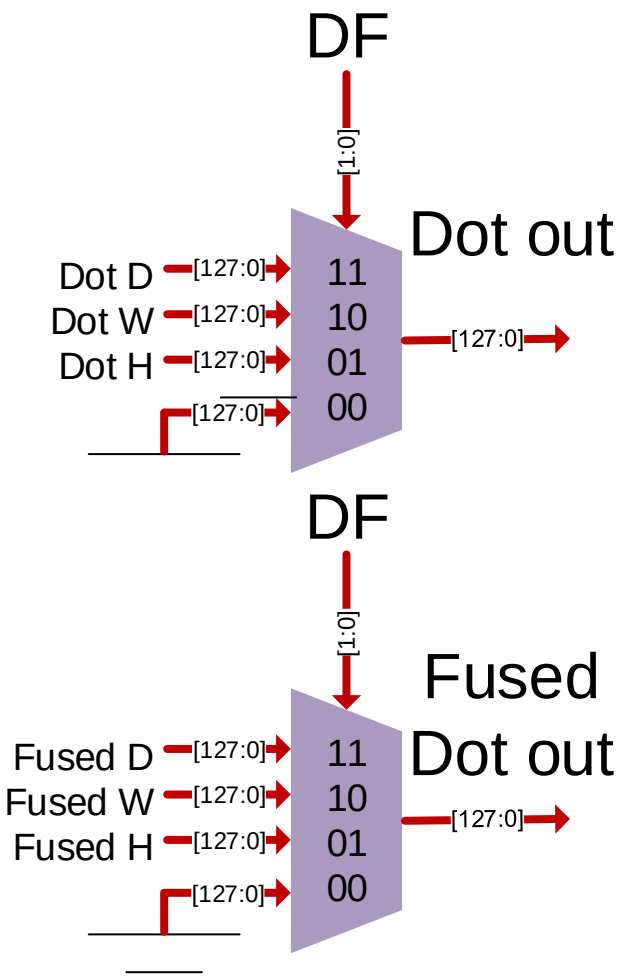


Figure 102: Selection of results from Dotproduct

16.19 Population count unit

Figure 105 shows the population count unit for all 4 vector formats. This unit executes the PNCT instruction. This unit is composed for 16-byte population counter. Figure 103 shows how each one is made. Adding some full and half adder we can obtain 16-bit, 32-bit and 64-bit population counts. An example of 16-bit population count base on 8-bit population count is show in Figure 104.

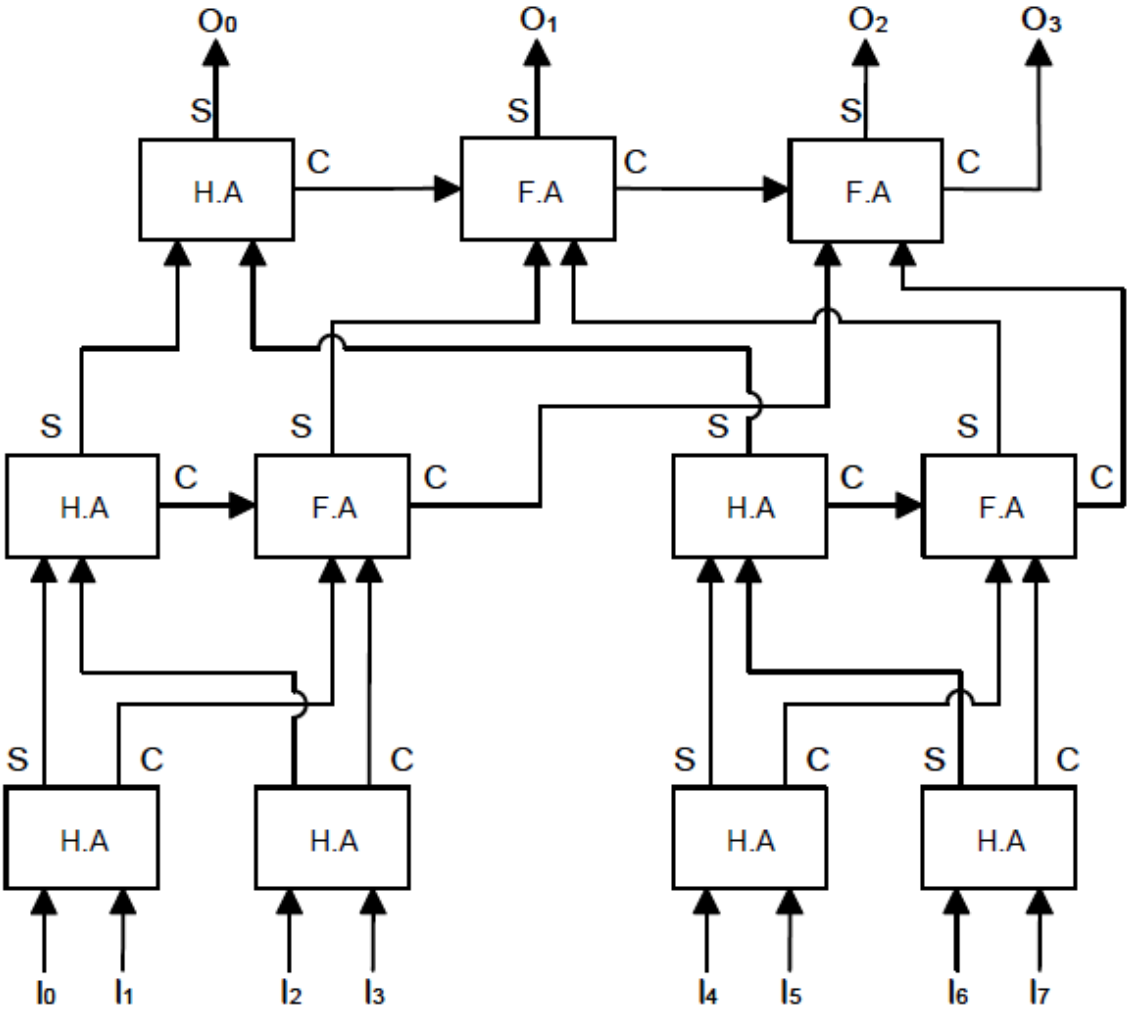


Figure 103: Population counter for a byte.

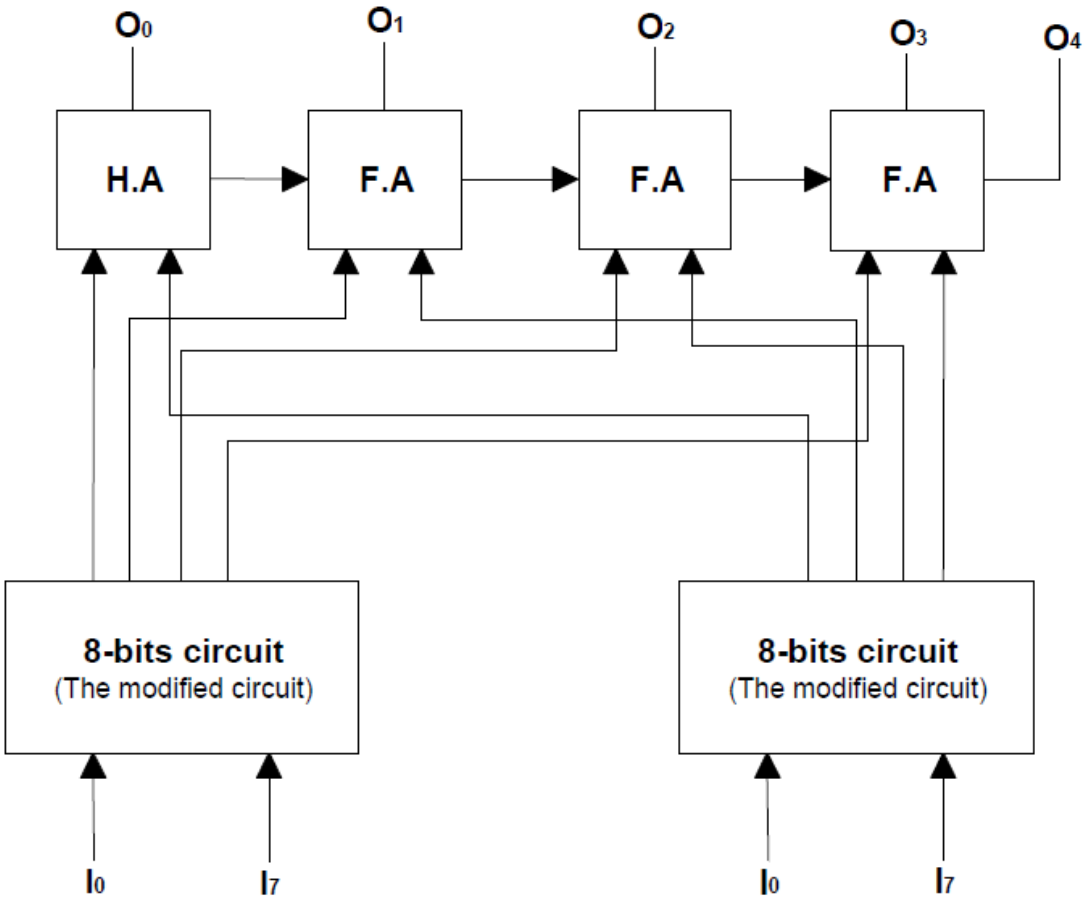


Figure 104: Population counter for a halfword.

The total number of logic gates for Figure 103 is as follows: 5 * 2 FA + 2 * 9 HA + 2 OR = 30 logic gates. The design of Figure 103 is for an input of 8-bit. However, we can duplicate the number of input bits (i.e., 16-bit) by duplicating the circuit above and adding another layer which consists of three FAs and one HA to get an output of 5 bits, as shown below in Figure 104.

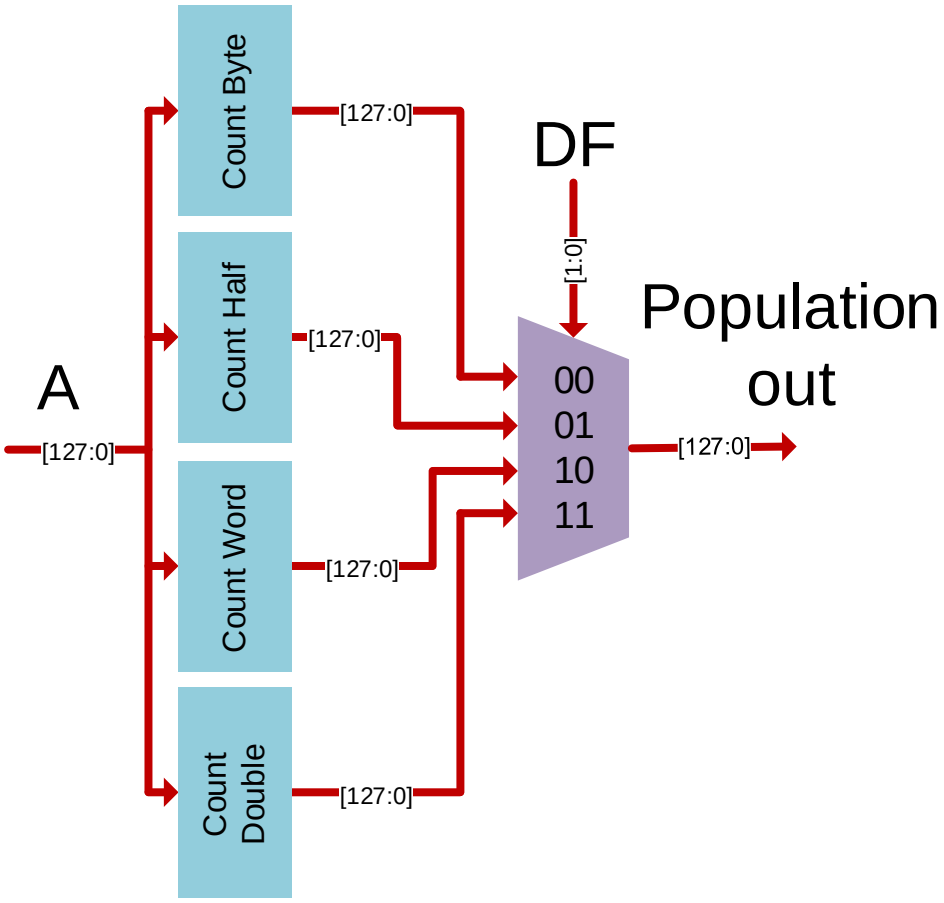


Figure 105: Population counter unit

16.20 Leading Ones/Zeros unit

Figure 107 shows the Leading counting unit for all 4 vector formats. This unit executes instructions NLOC and NLZC. The leading counting unit is made of leading one counters of a byte. One of them is shown in Figure 106. Negating the input we can count zeros instead ones.

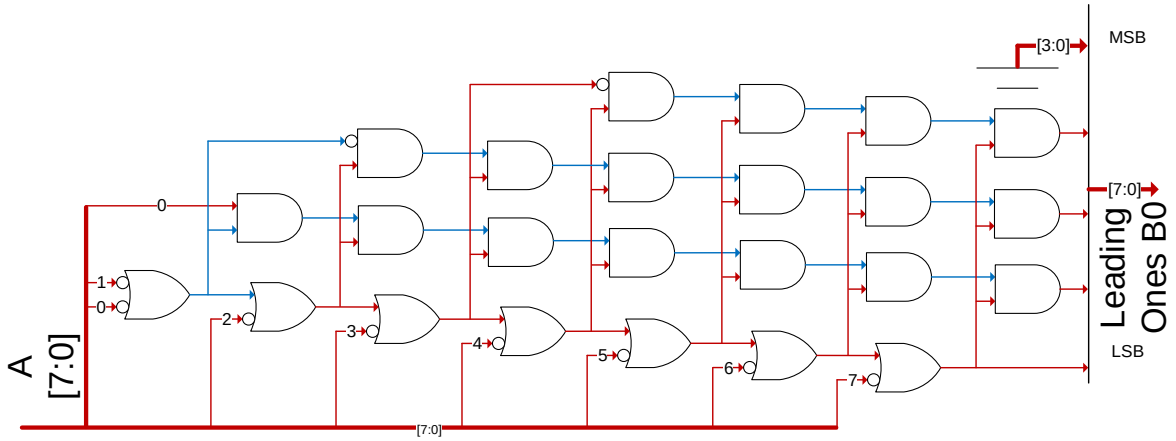


Figure 106: Leading byte counting

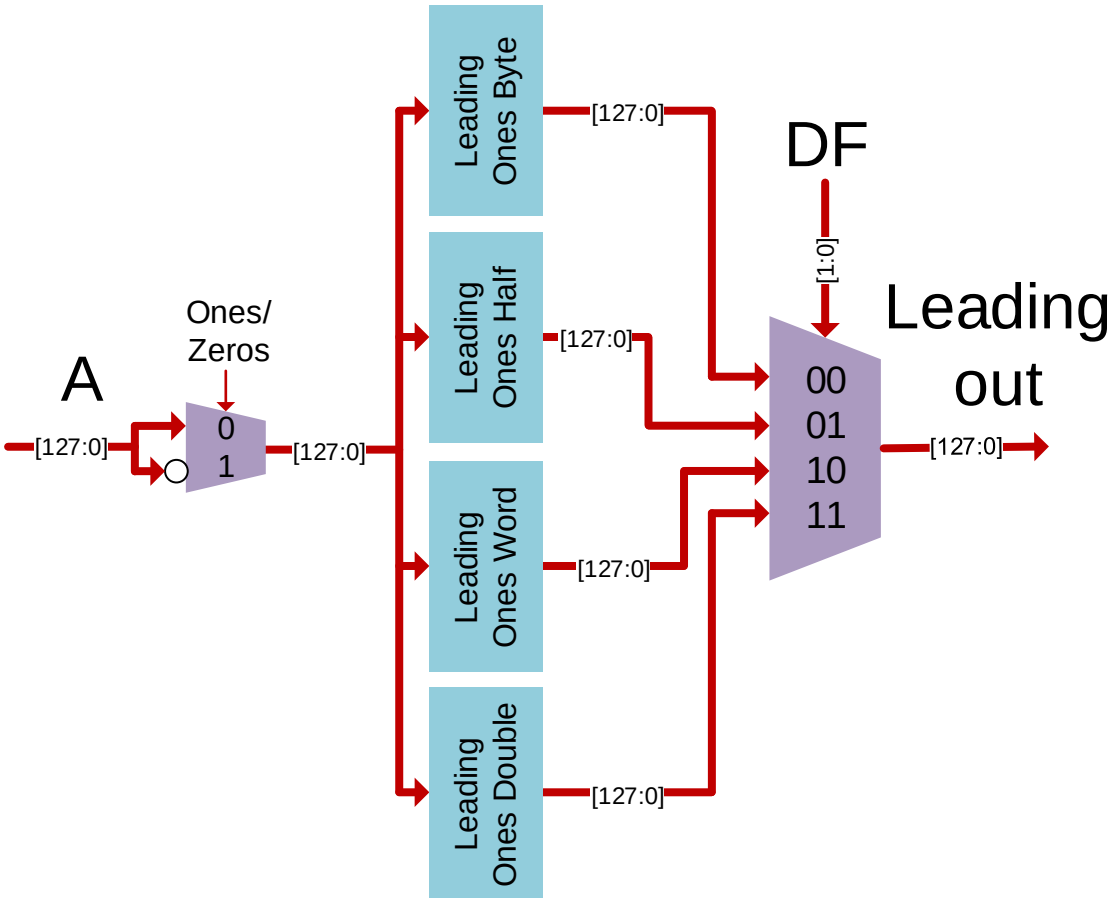


Figure 107: Leading counting unit

16.21 Vector Operations unit

There are 7 instructions (Table 18) that are independent of the vector format. They are executed at bit level over the whole vector. Figure 108 shows the circuits used to perform these vector operations. These instructions are AND, OR, NOR, XOR, BMNZ, BMZ and BSEL. Also using the special unit 2 immediate version of these instructions are executed too (Table 18).

Mnemonic	Type	Description
AND.V	VEC	Vector Logical And
BMNZ.V	VEC	Vector Bit Move If Not Zero
BMZ.V	VEC	Vector Bit Move If Zero
BSEL.V	VEC	Vector Bit Select
NOR.V	VEC	Vector Logical Negated Or
OR.V	VEC	Vector Logical Or
XOR.V	VEC	Vector Logical Exclusive Or

Mnemonic	Type	Description
ANDI.B	I8	Immediate Logical And
BMNZI.B	I8	Immediate Bit Move If Not Zero
BMZI.B	I8	Immediate Bit Move If Zero
BSELI.B	I8	Immediate Bit Select
NORI.B	I8	Immediate Logical Negated Or
ORI.B	I8	Immediate Logical Or
XORI.B	I8	Immediate Logical Exclusive Or

Table 18: Vector instructions

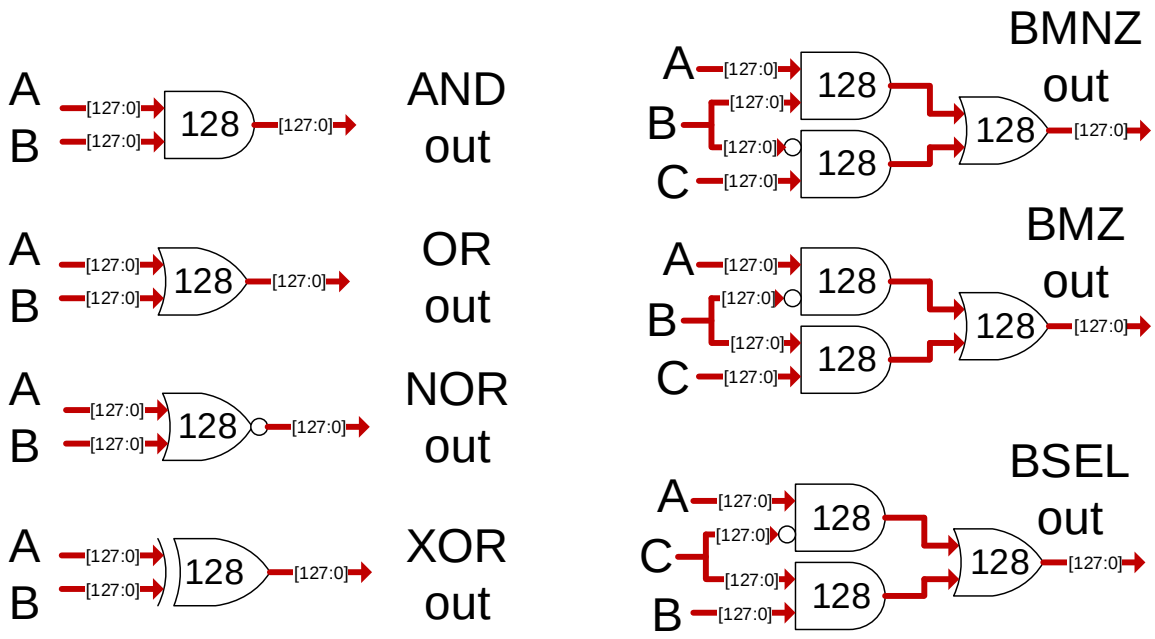


Figure 108: Vector operations

16.22 SHF unit

Figure 109 shows the implementation of SHF unit for *byte*, *halfword* and *word* vector formats. This unit executes instruction SHF. The set shuffle instruction works on 4-element sets in df data format. All sets are shuffled in the same way: the element i82i+1..2i in A is copied over the element i in C, where i is 0, 1, 2, 3. The operands and results are values in *byte* data format.

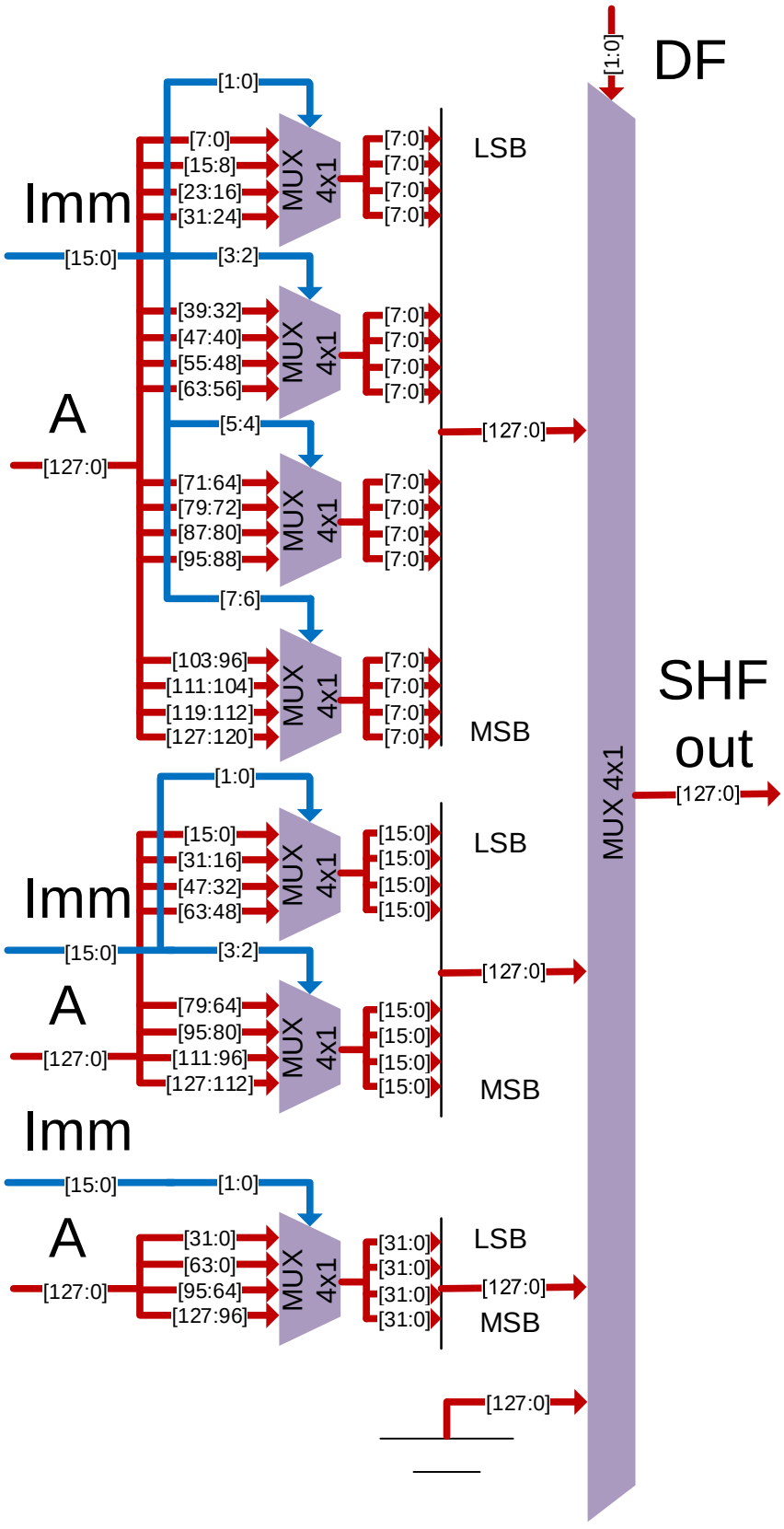

DF
[1:0]
Imm
[15:0]
A
[127:0]
Imm
[15:0]
A
[127:0]
Imm
[15:0]
A
[127:0]
MUX 4x1
SHF
out
[127:0]
LSB
MSB
[1:0]
[3:2]
[5:4]
[7:6]
[7:0]
[15:8]
[23:16]
[31:24]
[39:32]
[47:40]
[55:48]
[63:56]
[71:64]
[79:72]
[87:80]
[95:88]
[103:96]
[111:104]
[119:112]
[127:120]
[15:0]
[31:16]
[47:32]
[63:48]
[79:64]
[95:80]
[111:96]
[127:112]
[31:0]
[63:0]
[95:64]
[127:96]

Figure 109: SHF implementation

16.23 SAT unit

Figure 110, Figure 111, Figure 112 and Figure 113 shows the implementation of SAT unit for all 4 vector formats. This unit executes SAT_S and SAT_U instructions. See Saturated Arithmetic for more details.

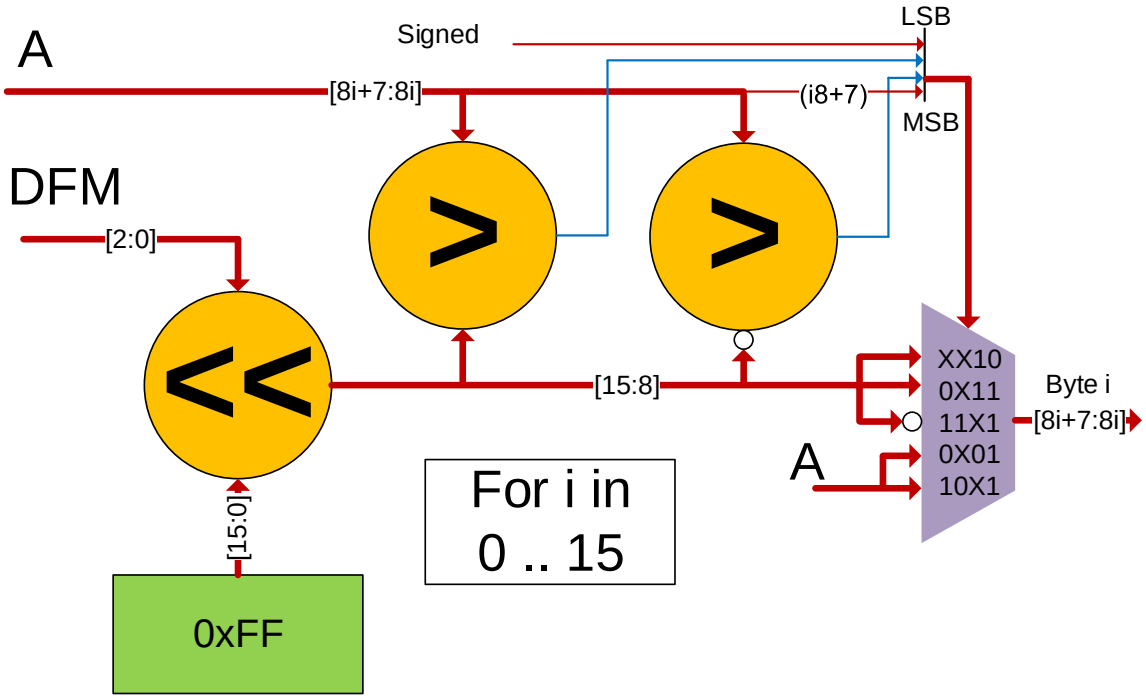


Figure 110: SAT unit implementation for byte format

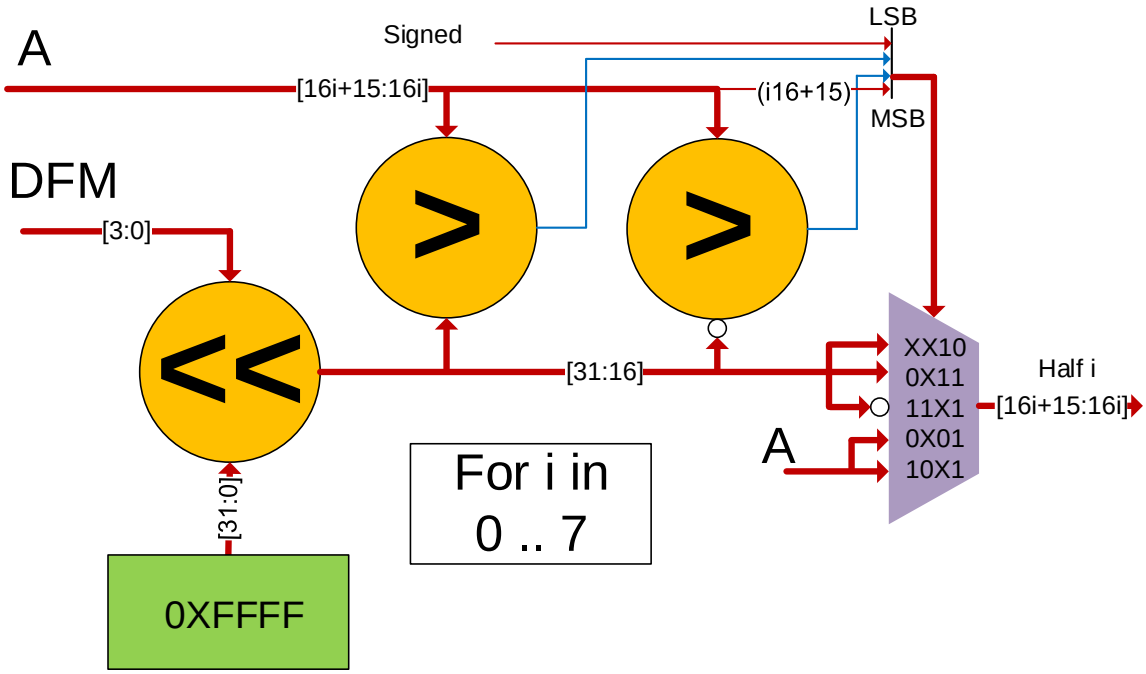


Figure 111: SAT implementation for halfword format

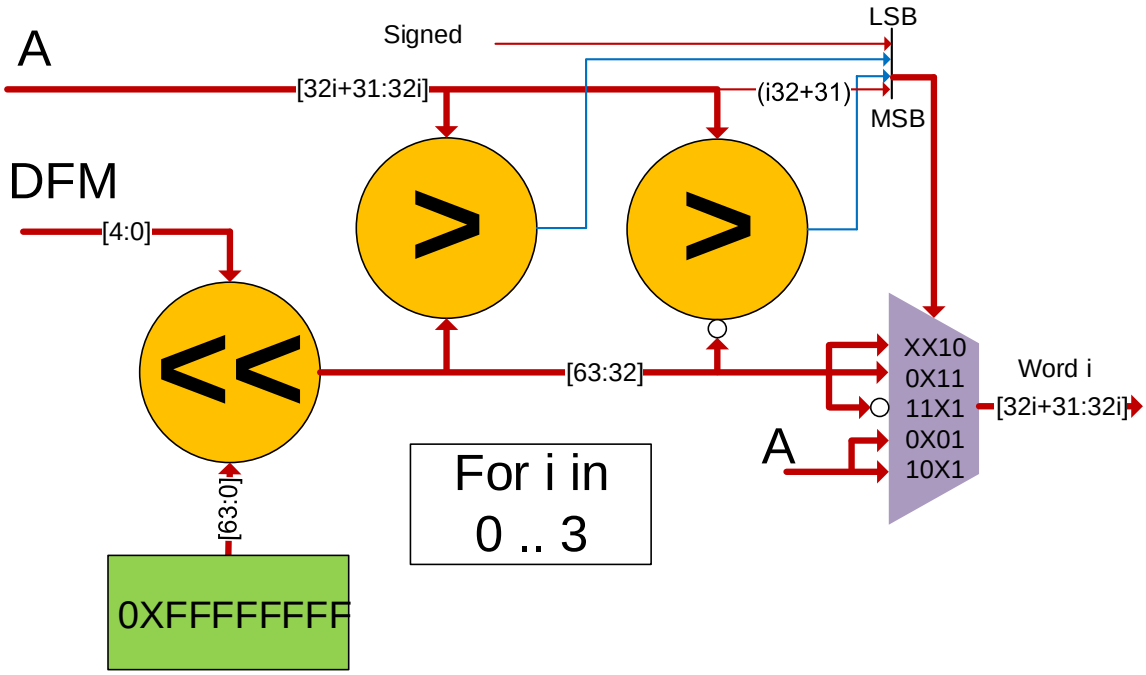


Figure 112: SAT implementation for word format

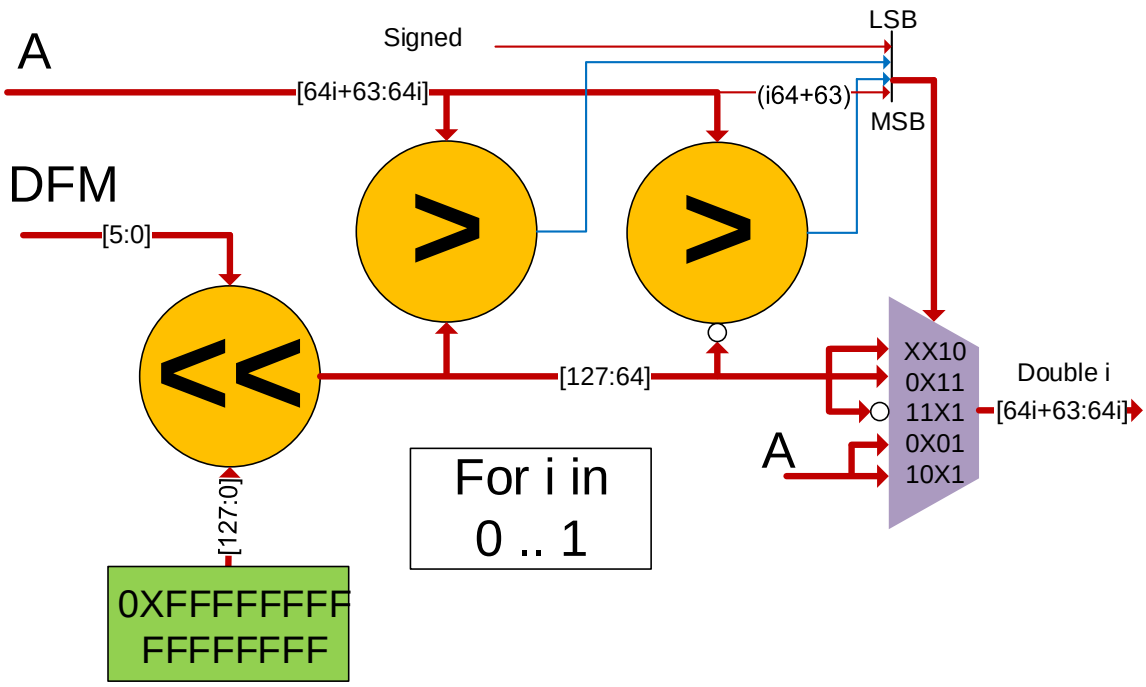


Figure 113: SAT implementation for Doubleword format

16.24 CEQ unit

Figure 114 shows the implementation of CEQ unit for all 4 vector formats. This unit executes CEQ and CEQI instructions using the special unit 2. CEQ unit set all bits to 1 in vector result elements if the corresponding A and B elements are equal, otherwise set all bits to 0. The operands and results are values in integer data format *df* field.

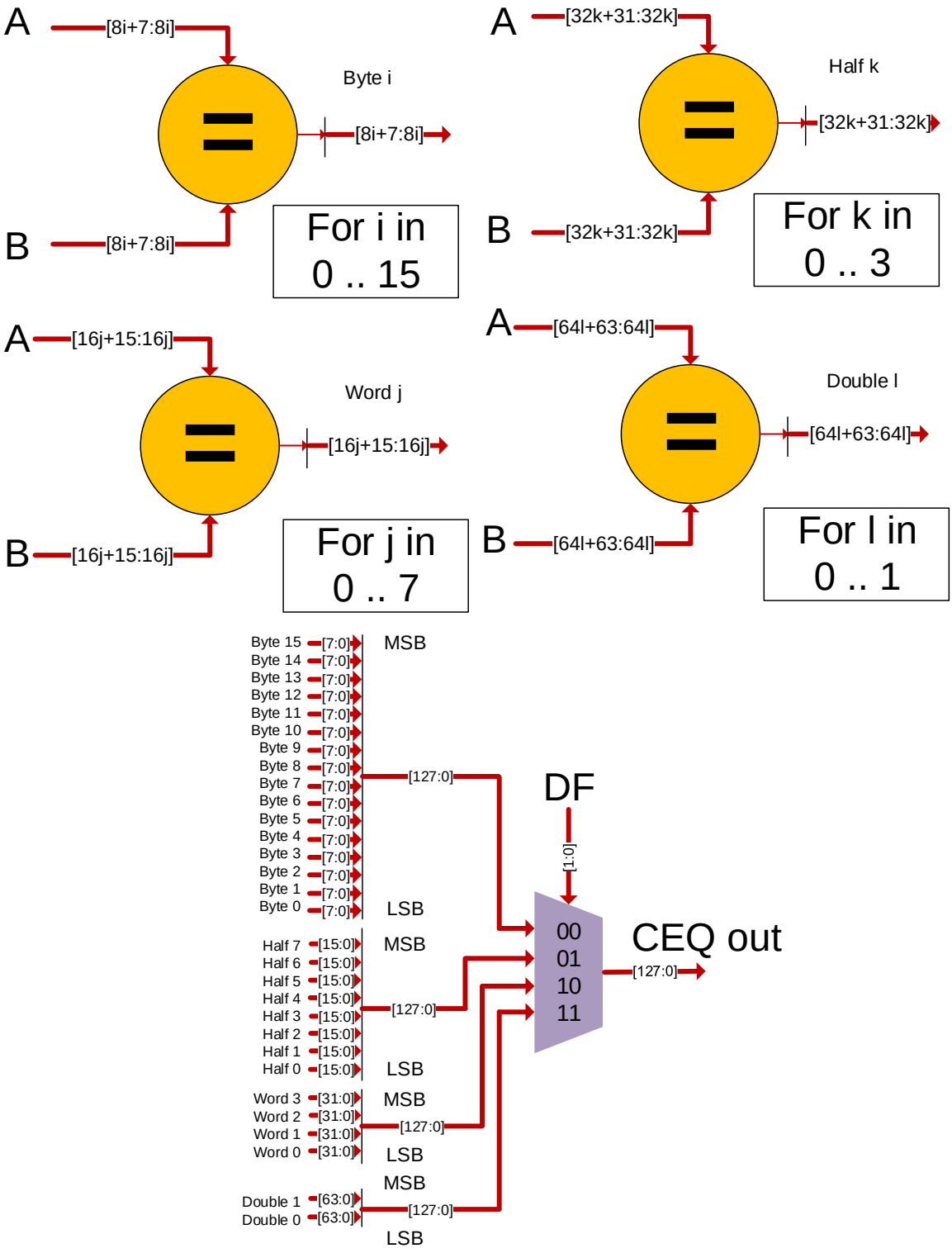


Figure 114: Implementation of CEQ unit

16.25 CLT unit

Figure 115 shows the implementation of the CLT unit for all 4 vector formats. Using the special unit 2, this unit can execute 4 instructions. They are CLT_S, CLT_U, CLTI_S and CLTI_U. CLT unit set all bits to 1 in vector result elements if the corresponding A elements are signed/unsigned (depending on signed flag) less than B elements, otherwise set all bits to 0. The operands and results are values in integer data format *df* field.

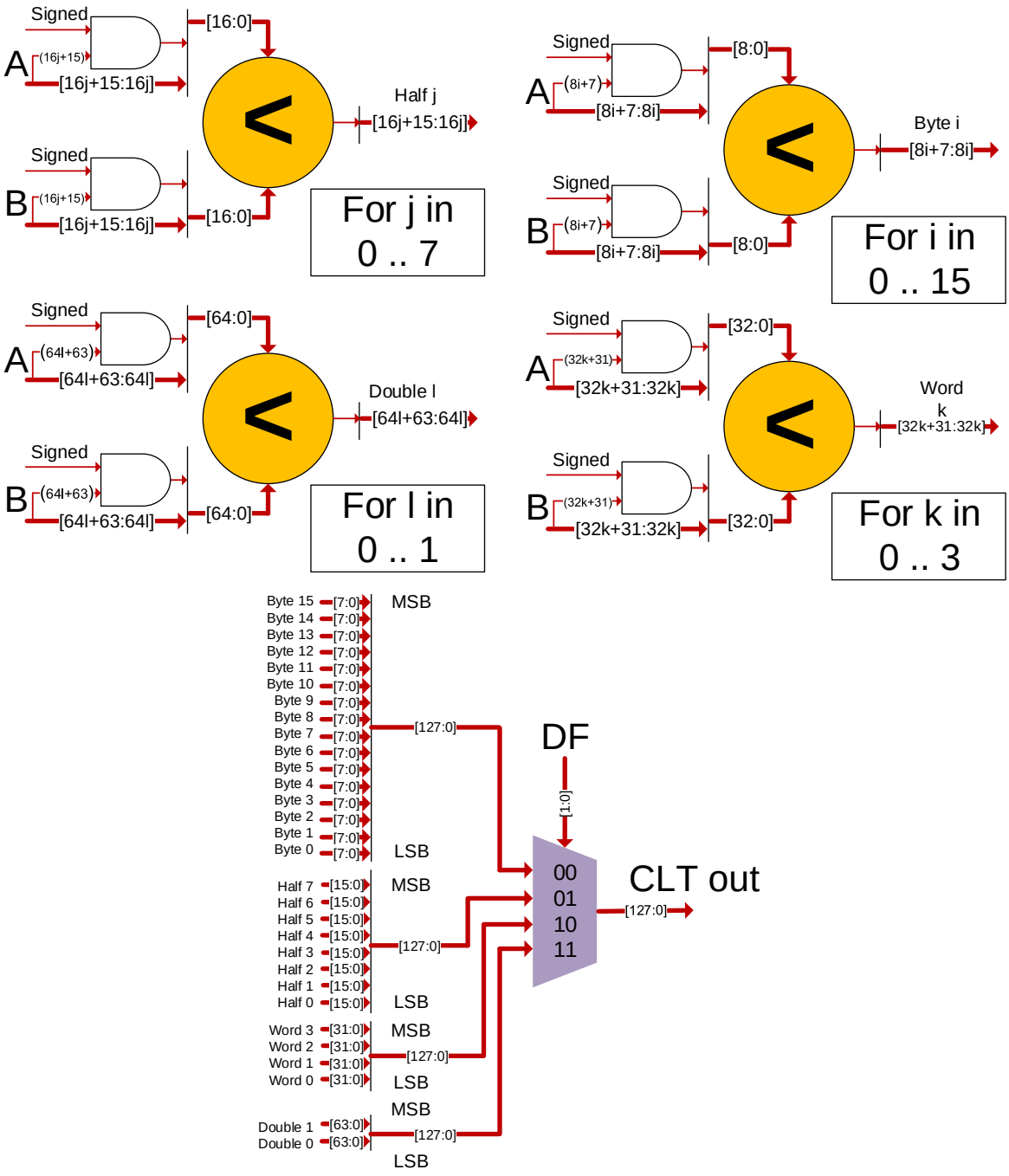


Figure 115: Implementation of CLT unit

16.26 CLE unit

Figure 116 shows the implementation of the CLR unit for all 4 vector formats. Using the special unit 2, this unit can execute 4 instructions. They are CLE_S, CLE_U, CLEI_S and CLEI_U. CLE unit set all bits to 1 in vector result elements if the corresponding A elements

are signed/unsigned (depending on signed flag) less than or equal to B elements, otherwise set all bits to 0. The operands and results are values in integer data format *df* field.

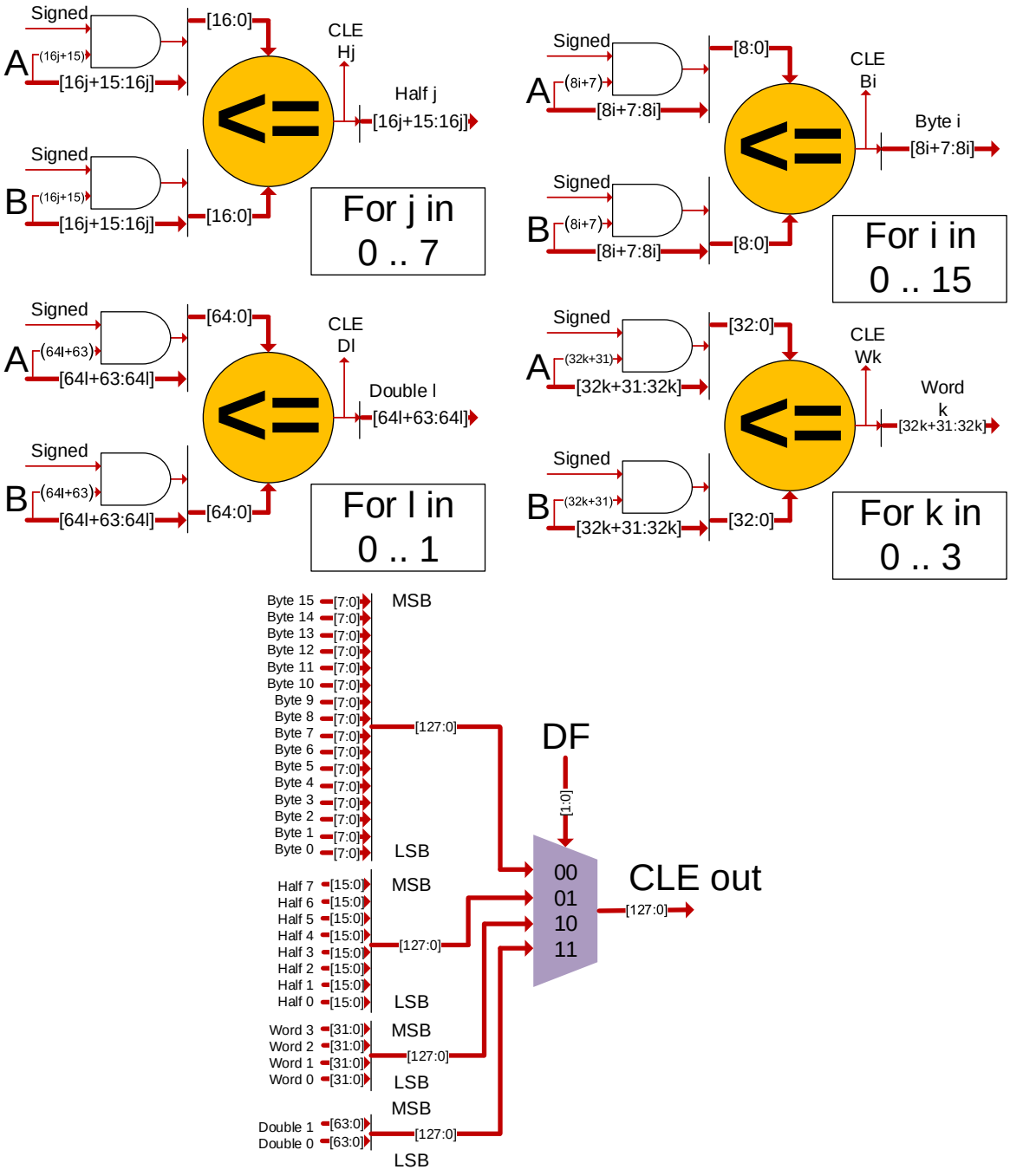


Figure 116: Implementation of CLE unit

16.27 MAX unit

Figure 117 shows the implementation of the MAX unit for all 4 vector formats. Using the special unit 2, this unit can execute 4 instructions. They are MAX_S, MAX_U, MAXI_S and MAXI_U. Maximum values between signed elements in vector B and signed/unsigned elements in vector A are written to vector result. The operands and results are values in integer data format *df* field.

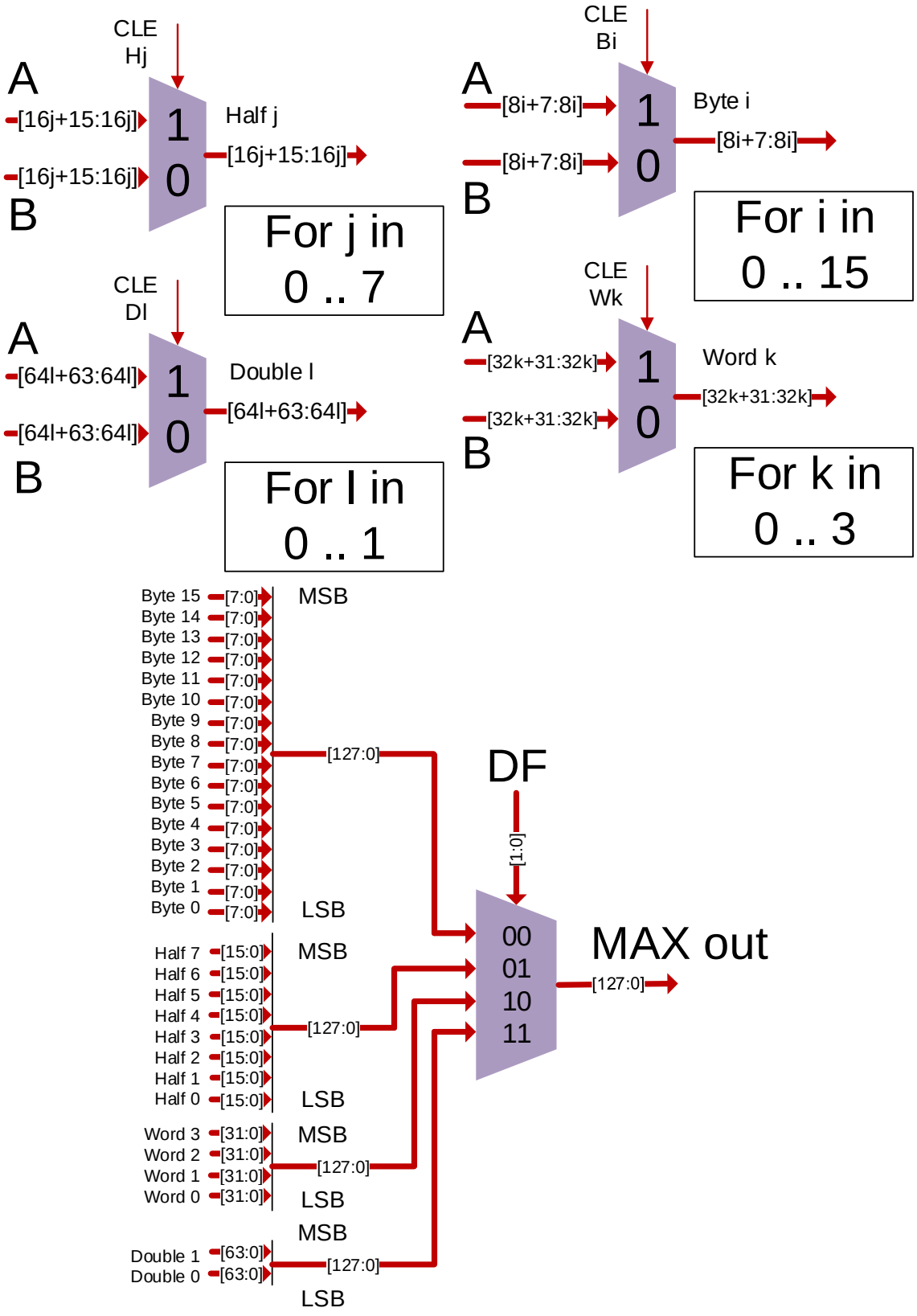


Figure 117: Implementation of MAX unit

16.28 MIN unit

Figure 118 shows the implementation of the MIN unit for all 4 vector formats. Using the special unit 2, this unit can execute 4 instructions. They are MIN_S, MIN_U, MINI_S and MINI_U. Minimum values between signed elements in vector B and signed elements in vector A are written to vector result. The operands and results are values in integer data format *df* field.

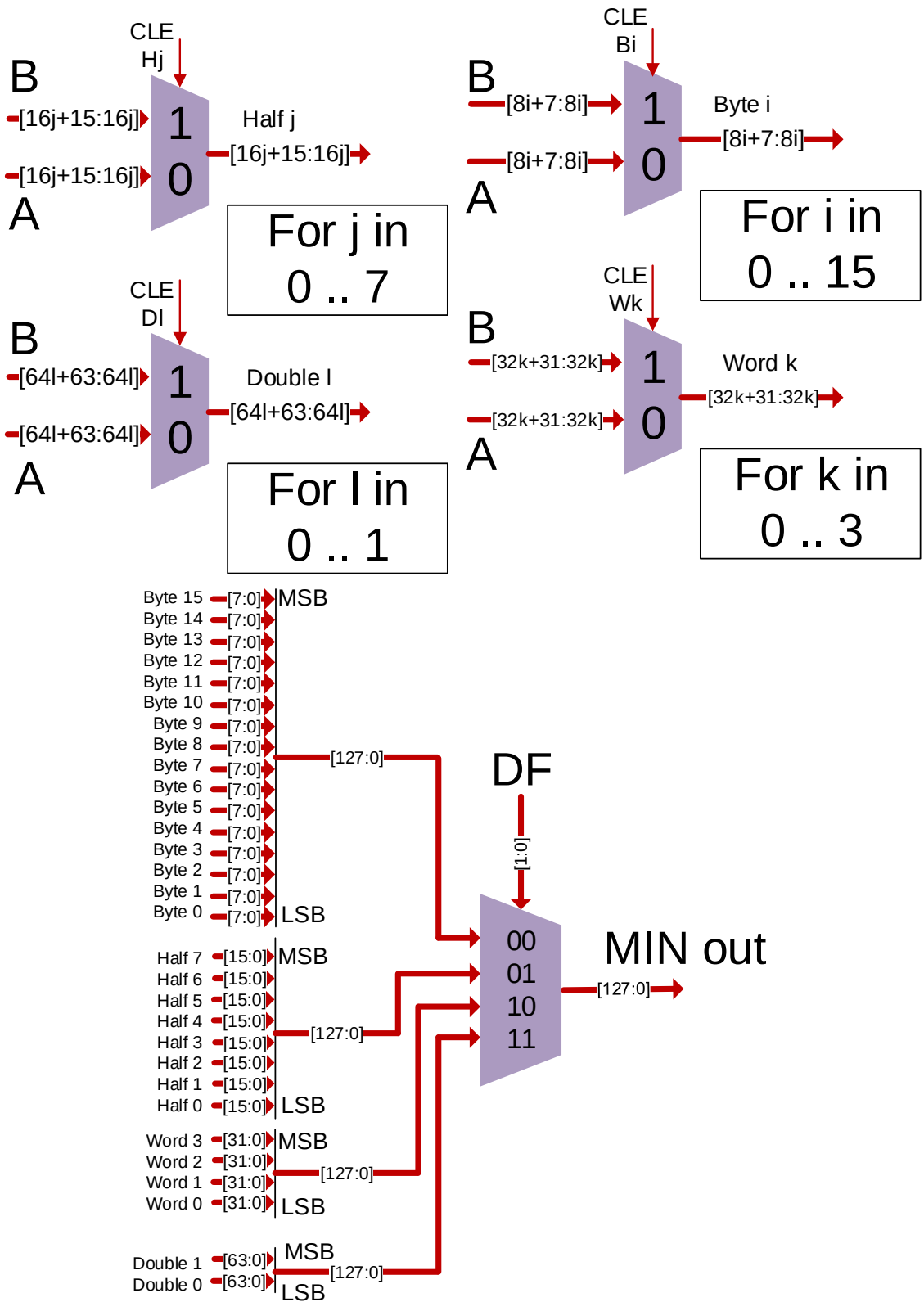


Figure 118: Implementation of MIN unit

16.29 MAX MN unit

Figure 119, Figure 120 and Figure 121 shows the implementation of the MAX MIN unit for all 2 vector formats. This unit uses the absolute values generated in Figure 40. This unit calculates instructions MAX_A and MIN_A. The value with the largest magnitude, i.e. absolute value, between corresponding signed/unsigned elements in vector A and vector B are written to vector result. The minimum negative value representable has the largest absolute value. The operands and results are values in integer data format *df* field.

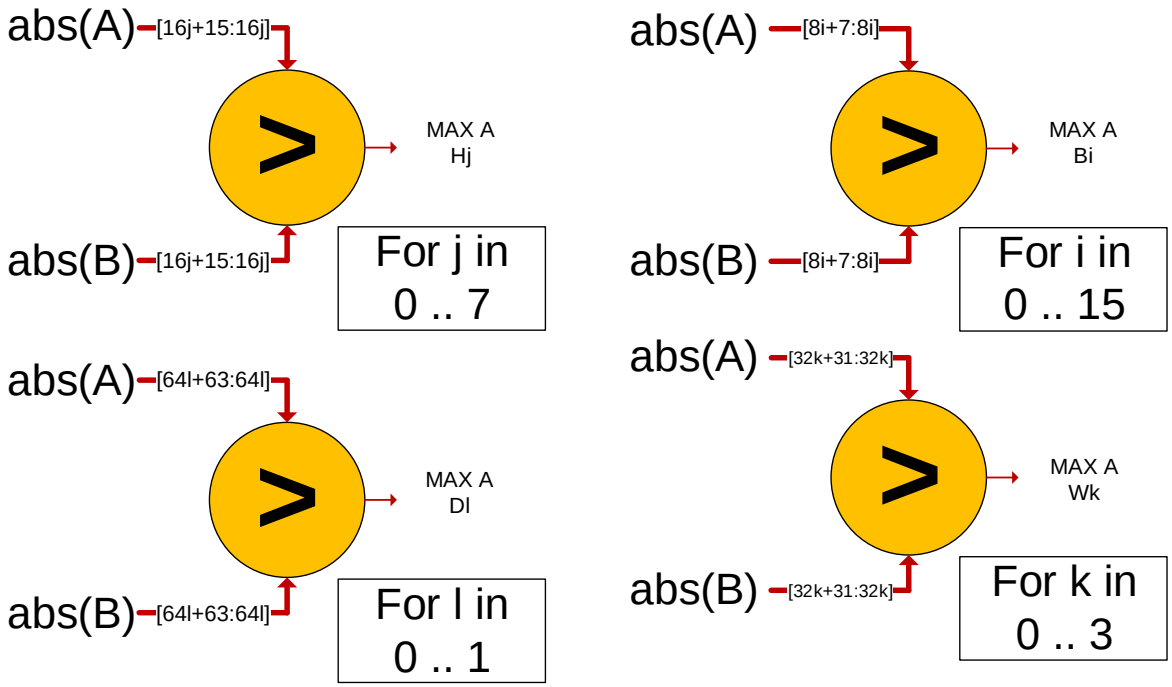


Figure 119: Implementation of MAX MIN unit

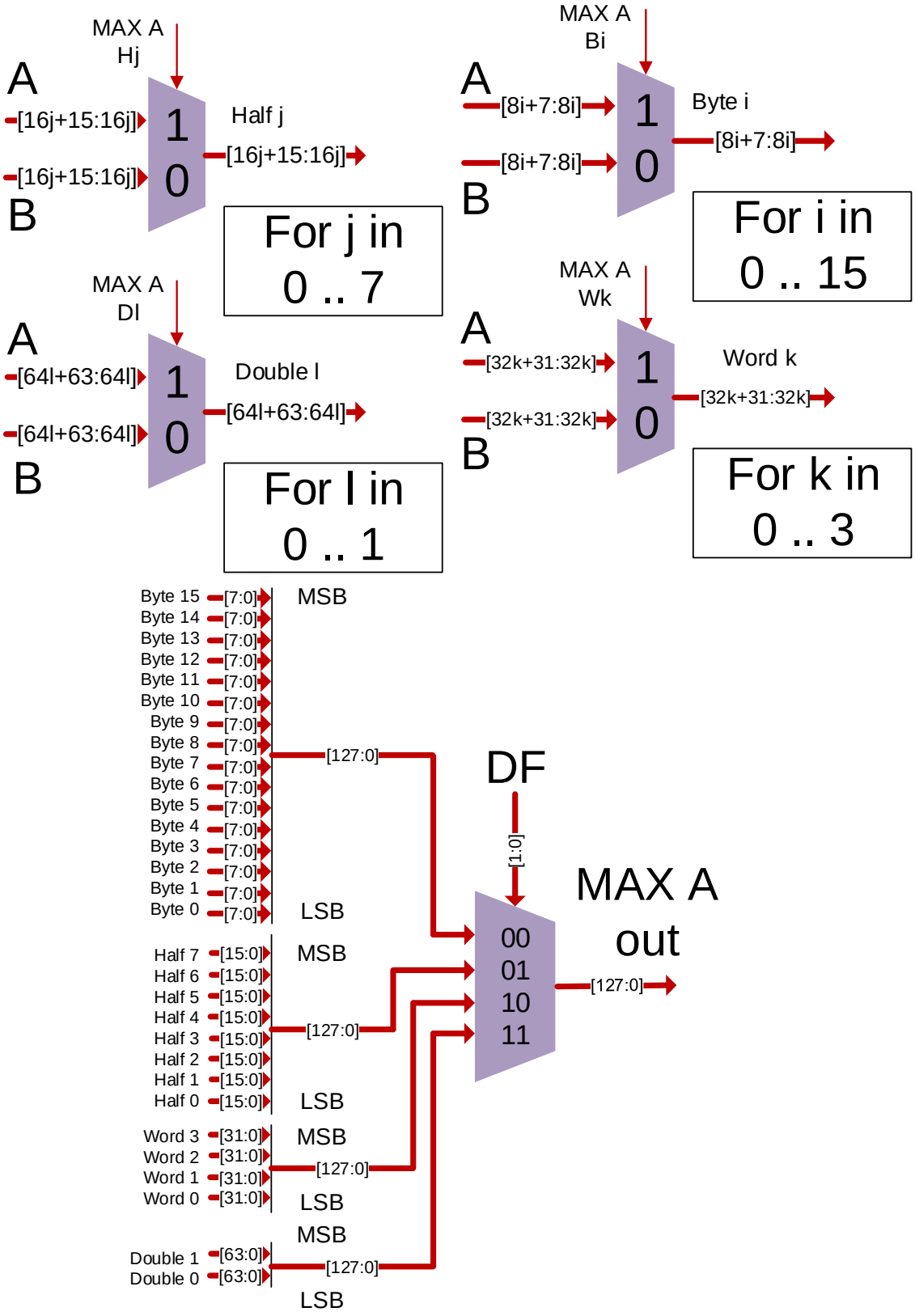


Figure 120: Implementation of MX MIN unit (cont.)

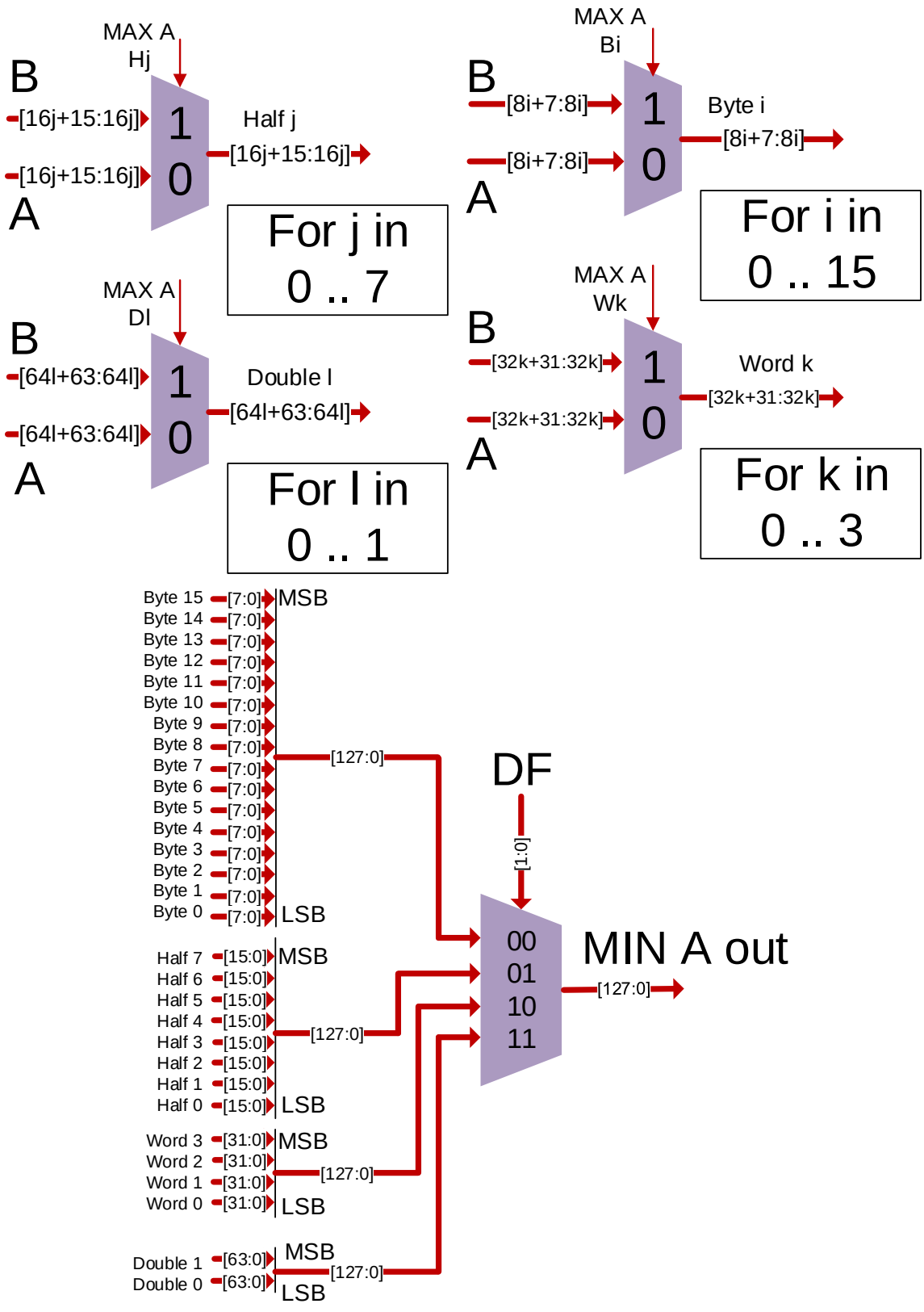


Figure 121: MAX MIN unit implementation (cont.)

16.30 SLL unit

Figure 122 shows the implementation of the SLL unit for all 2 vector formats. Using the special unit 2, this unit can execute 4 instructions. They are SLL and SLLI instructions. The elements in vector A are shifted left by the number of bits the elements in vector B specify modulo the size of the element in bits. The result is written to vector result. The operands and results are values in integer data format *df* field.

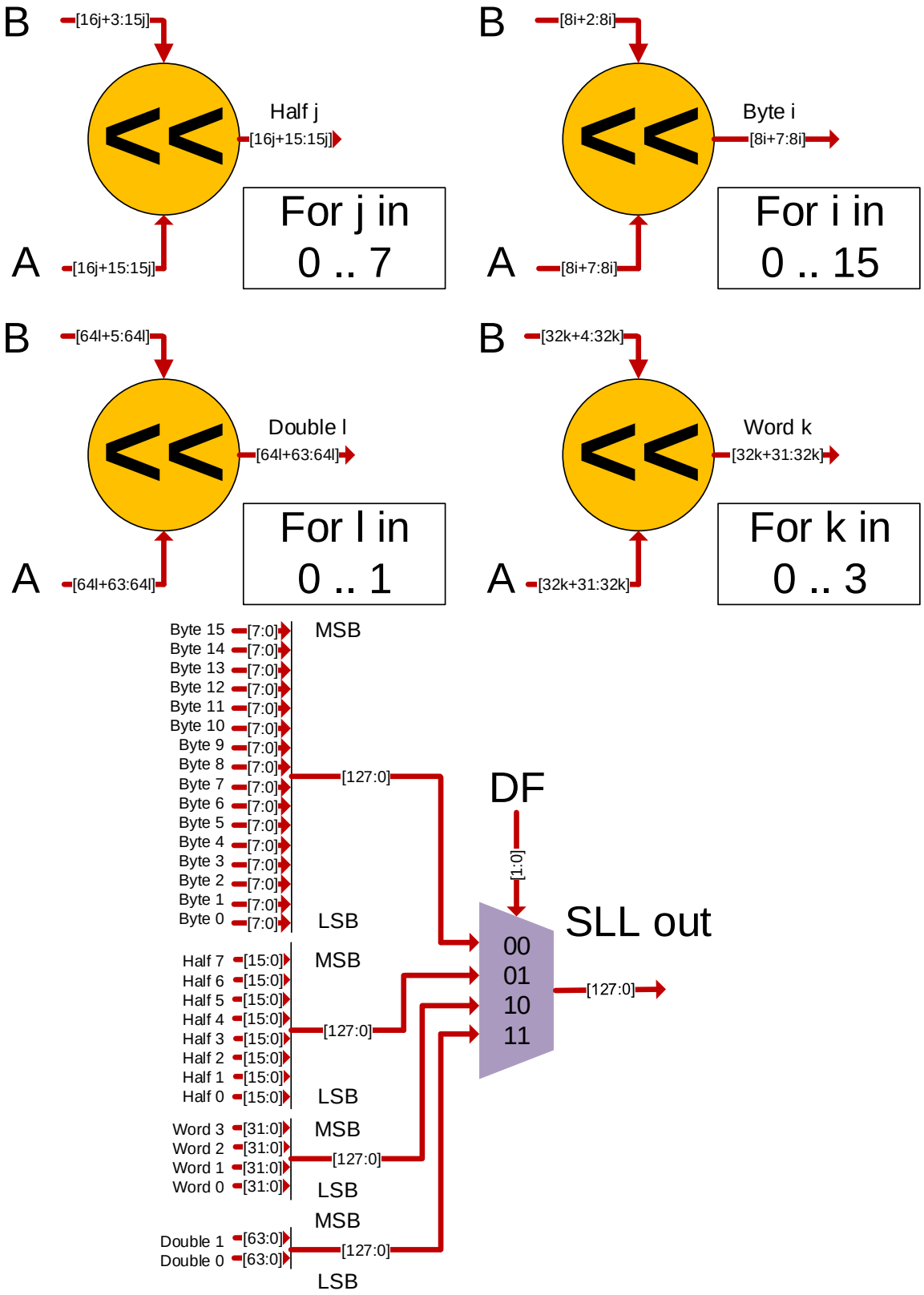


Figure 122: Implementation of SLL unit

16.31 SRA unit

Figure 123 shows the implementation of the SRA unit for all 4 vector formats. Using the special unit 2, this unit can execute 2 instructions. They are SRA and SRAI instructions. The elements in vector A are shifted right arithmetic by the number of bits the elements in vector B specify modulo the size of the element in bits. The result is written to vector result. The operands and results are values in integer data format *df* field.

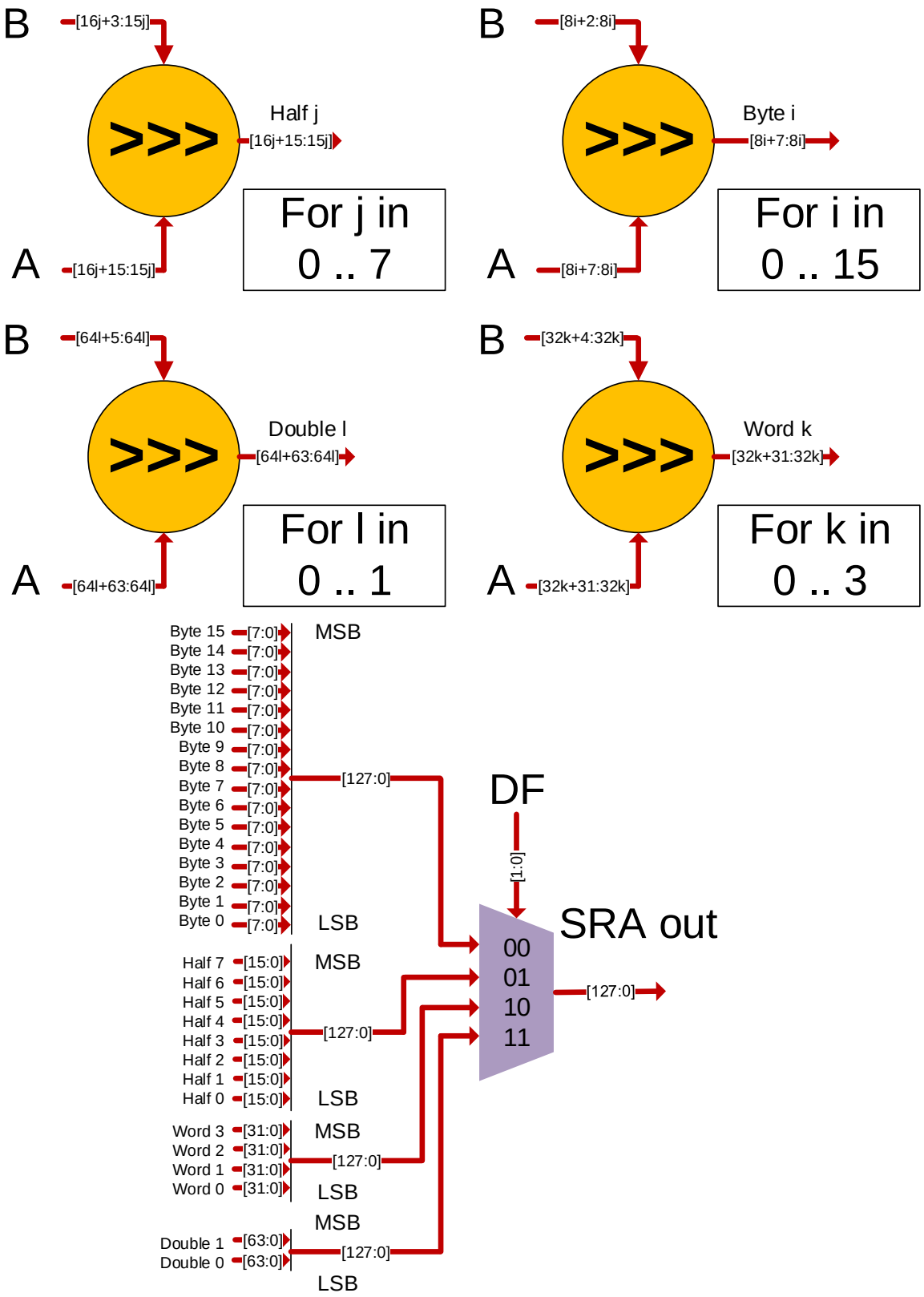


Figure 123: Implementation of SRA unit

16.32 SRL unit

Figure 124 shows the implementation of the SRL unit for all 4 vector formats. Using the special unit 2, this unit can execute 2 instructions. They are SRL and SRLI instructions. The elements in vector A are shifted right logical by the number of bits the elements in vector B specify modulo the size of the element in bits. The result is written to vector result. The operands and results are values in integer data format *df* field.

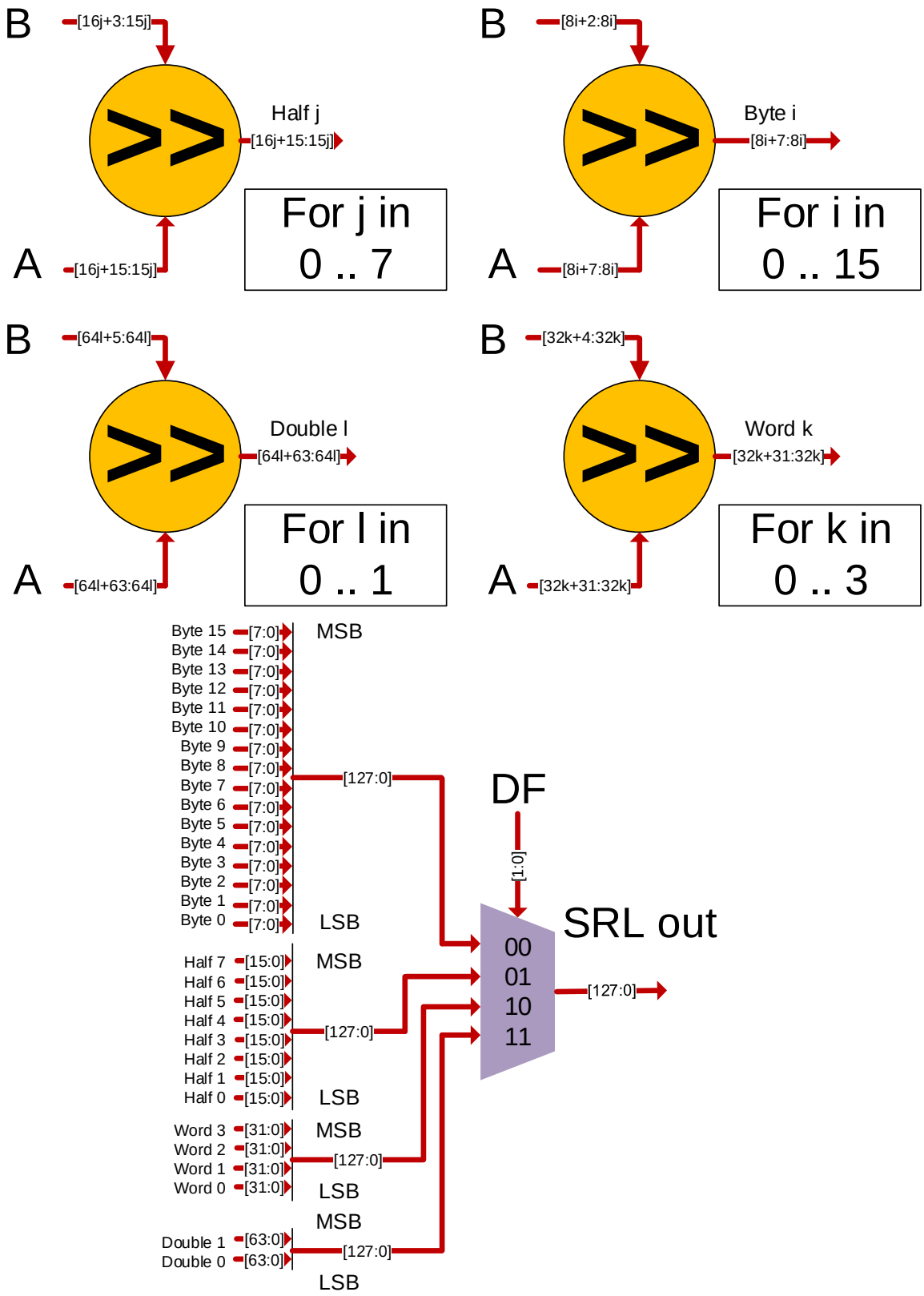


Figure 124: Implementation of SRL unit

16.33 BIT unit

Figure 125 and Figure 126 shows the implementation of the BIT unit for all 4 vector formats. Using the special unit 2, this unit can execute 6 instructions. They are BCLR, BSET, BNEG, BCLRI, BSETI and BNEGI.

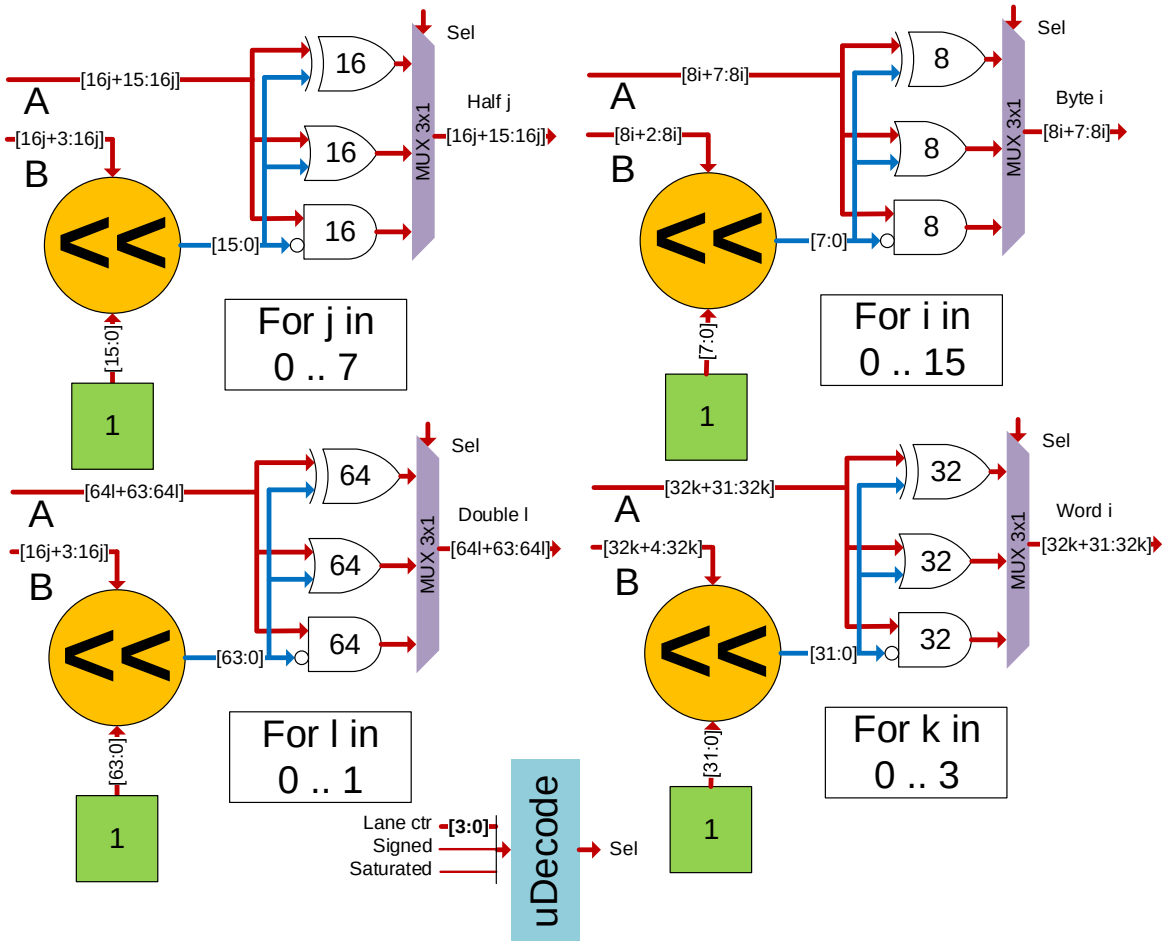


Figure 125: Implementation of BIT unit

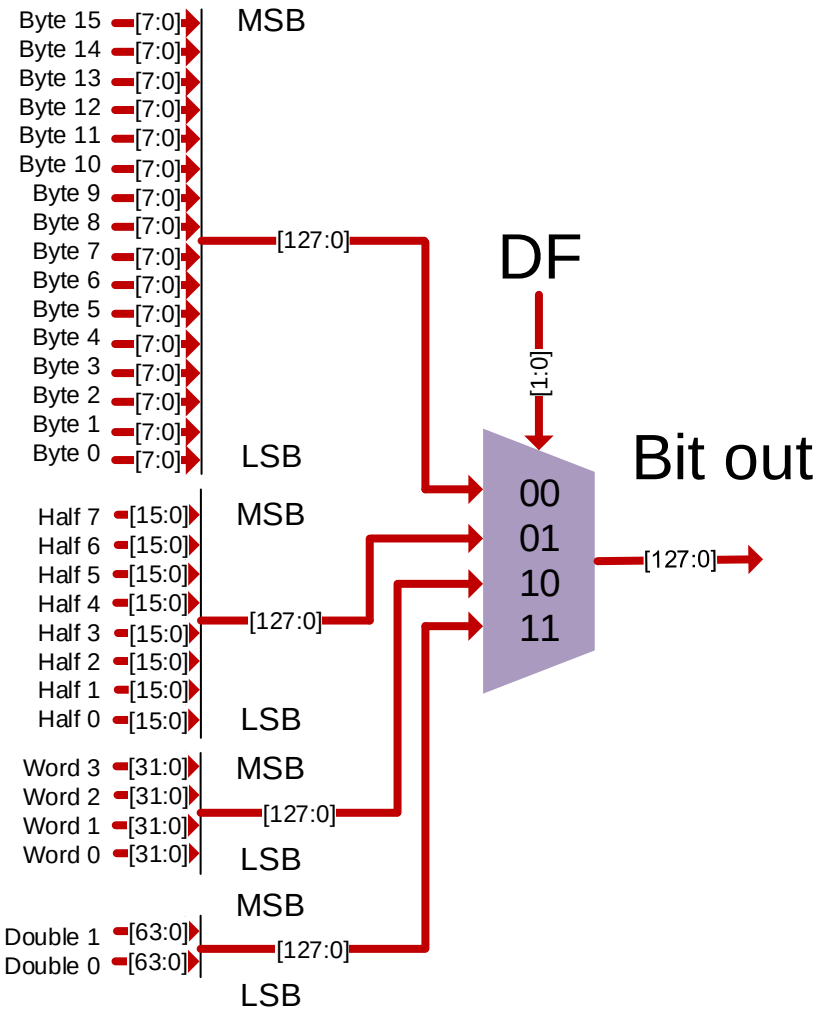


Figure 126: Implementation of BIT unit (cont.)

16.34 BINSL unit

Figure 127 shows the implementation of BINSL unit for all 4 vector formats. This unit using the special unit 2 executes instructions BINSL and BINSLI. To select the output format circuit shows in Figure 129 is used. Copy most significant (left) bits in each element of vector A to elements in vector C while preserving the least significant (right) bits. The number of bits to copy is given by the elements in vector B modulo the size of the element in bits plus 1. The operands and results are values in integer data format *df* field.

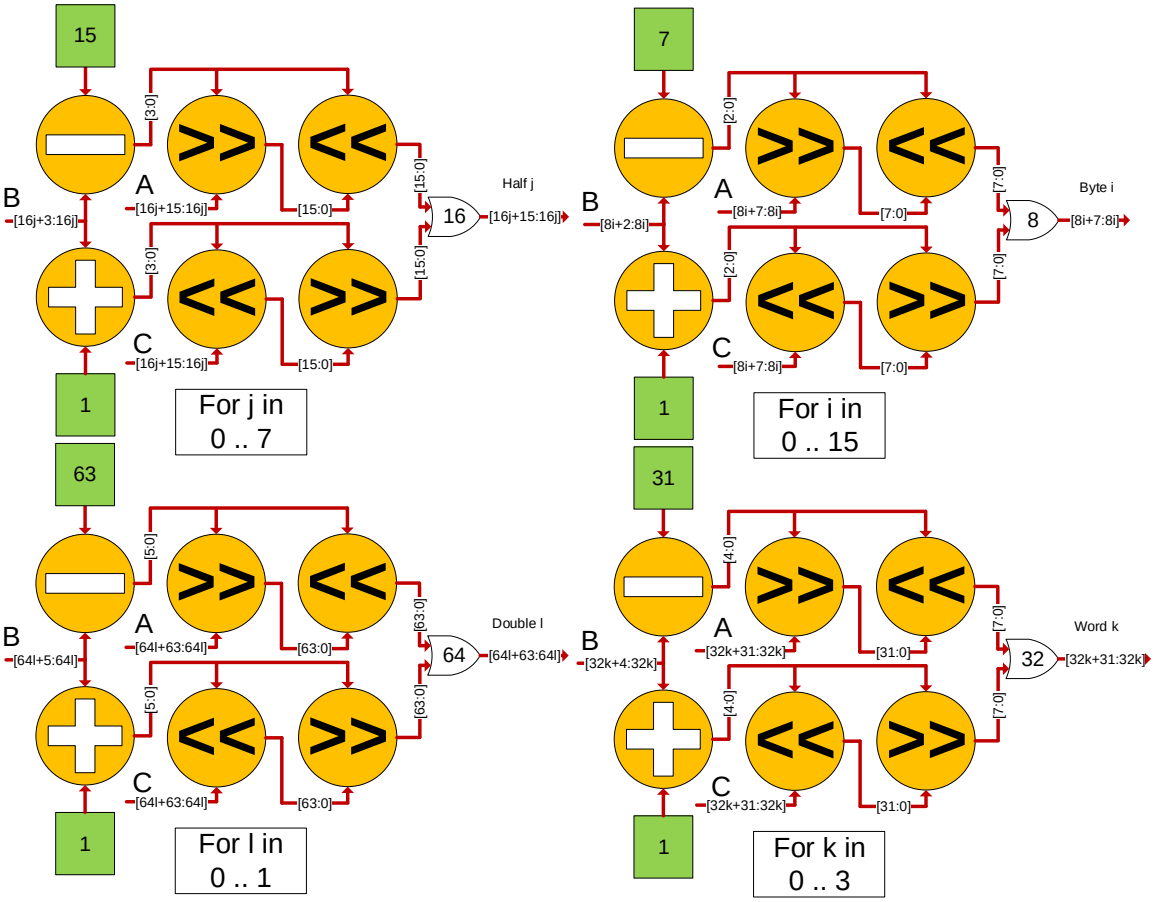


Figure 127: Implementation of BINSL unit

16.35 BINSR unit

Figure 128 shows the implementation of BINSR unit. This unit using the special unit 2 can executes instructions BINSR and BINSRI. To select the output format circuit shows in Figure 129 is used. Copy least significant (right) bits in each element of vector A to elements in vector C while preserving the most significant (left) bits. The number of bits to copy is given by the elements in vector B modulo the size of the element in bits plus 1. The operands and results are values in integer data format *df* field.

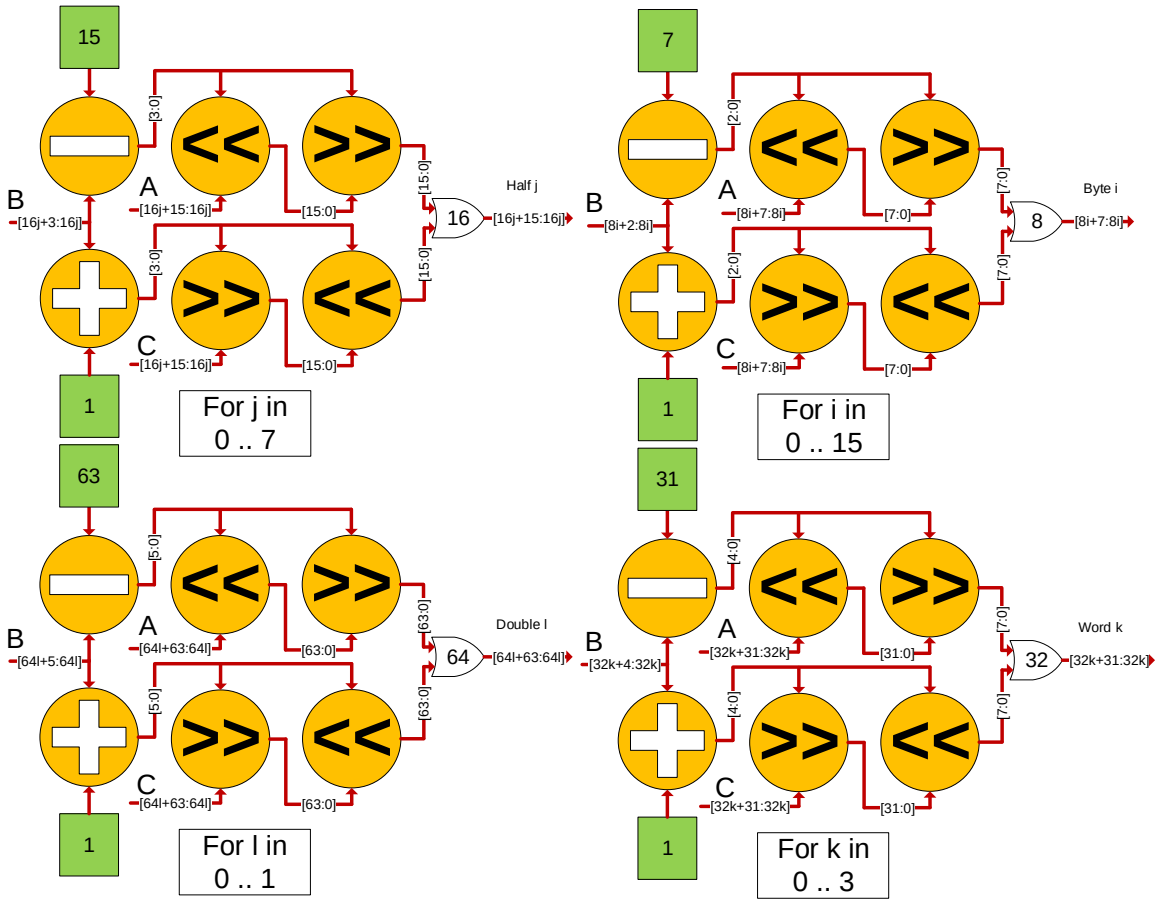


Figure 128: Implementation of BINSR unit

16.36 Join results

Figure 129 shows the circuit used by almost every unit that chose the output result format. Al possible formats are always executes but just only one of them has sense.

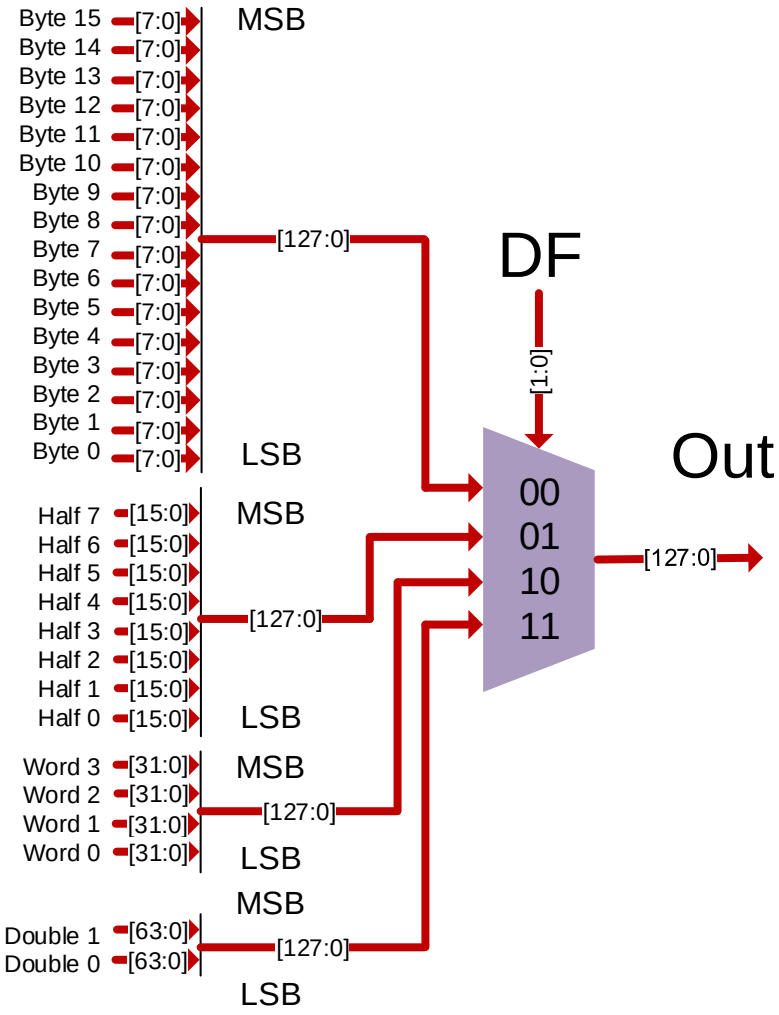


Figure 129: Result format selection

16.37 Branch unit

For branch, MSA uses no global condition flags: compare instructions write the results per vector element as all zero or all one bit values. Branch instructions test for zero or not zero elements or vector value. Figure 130 shows the layout of branch instructions. Field COP1 is decoded by MSA and MIPS32 decoders. Field s16 is used by the MIPS32 core to calculate the branch address. Fields OP/DF and wt are used by the SIMD unit to evaluate the branch.

The branch instruction has a delay slot. s16 is a PC word offset, i.e. signed count of 32-bit or 64-bit[23] instructions, from the PC of the delay slot. Finally the MSA Jump signal is send to the MIPS32 control unit to take or not take the branch.

Mnemonic	Type	Description
BNZ.V	COP1	Immediate Branch If Not Zero (At Least One Element of Any Format Is Not Zero)
BNZ.df	COP1	Immediate Branch If All Elements Are Not Zero
BZ.df	COP1	Immediate Branch If At Least One Element Is Zero
BZ.V	COP1	Immediate Branch If Zero (All Elements of Any Format Are Zero)

Table 19: SIMD branch instructions

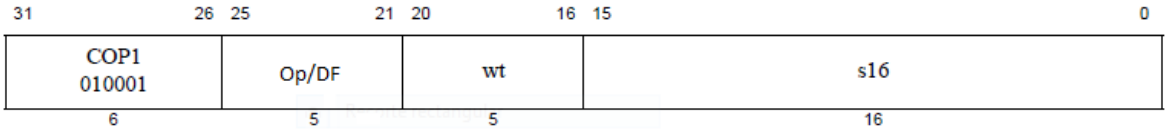


Figure 130: Layout of branch instructions

[23] Depending on the MIPS32 or MIPS64 implementaion runninf MSA

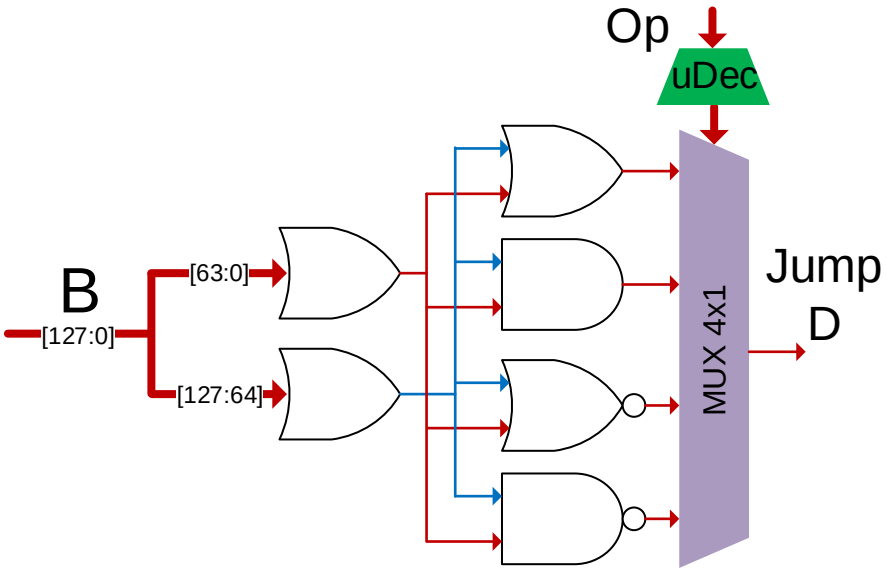


Figure 131: Branch detection of format Doubleword

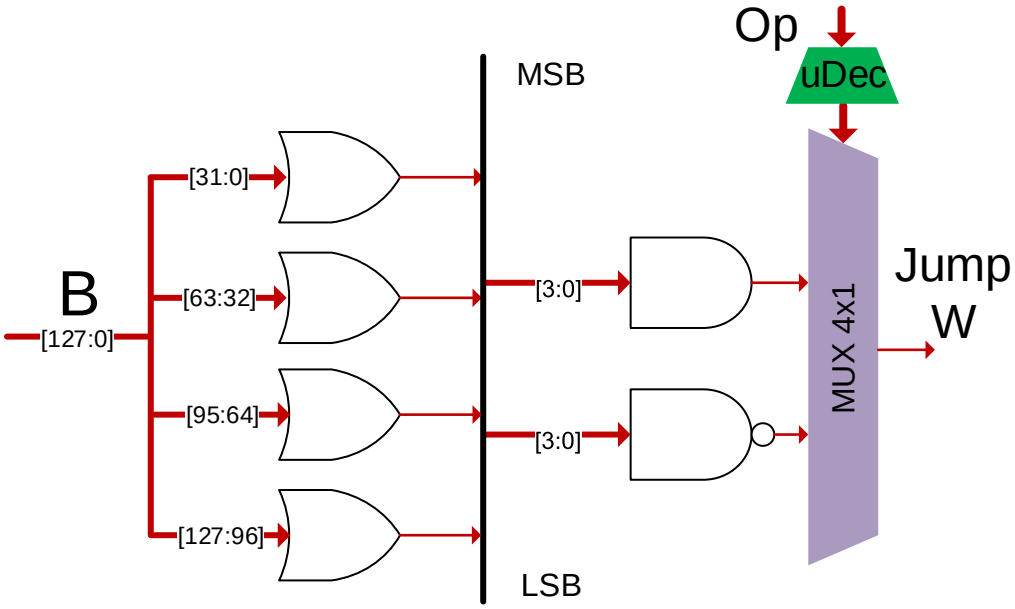


Figure 132: Branch detection of format Word

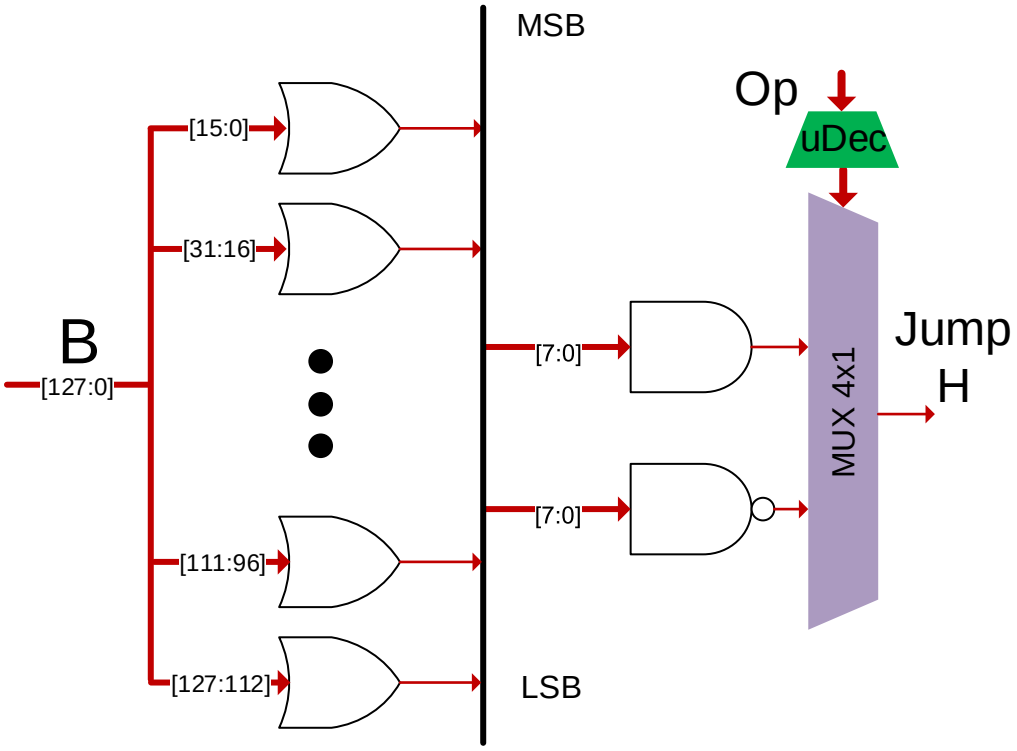


Figure 133: Branch detection of format Halfword

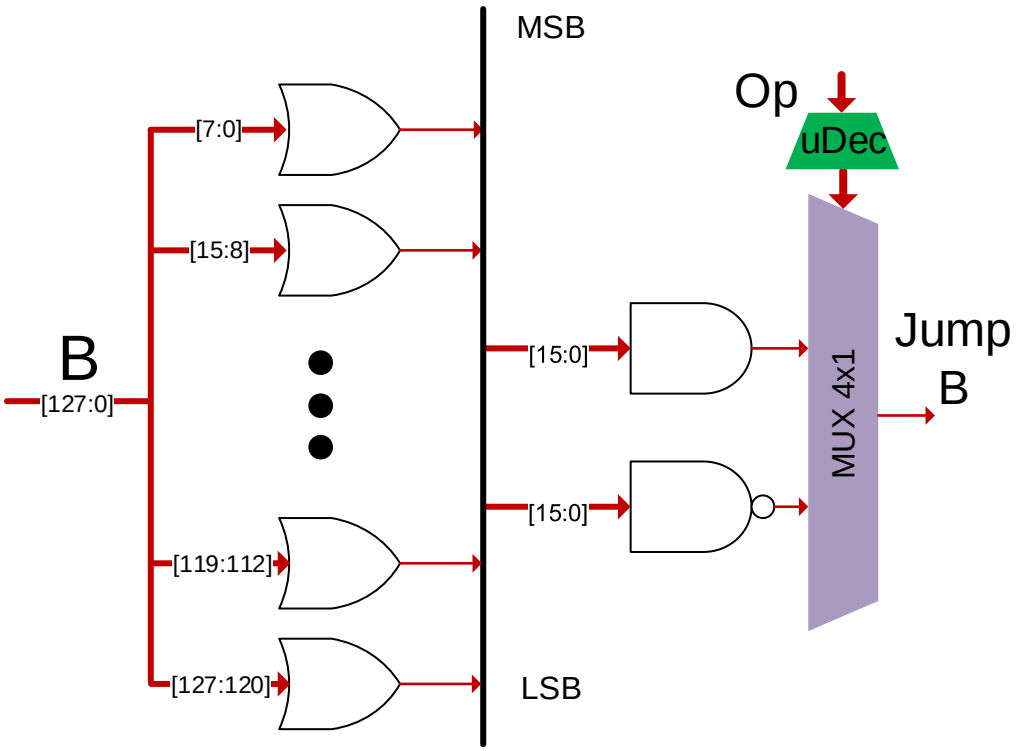


Figure 134: Branch detection of format Byte

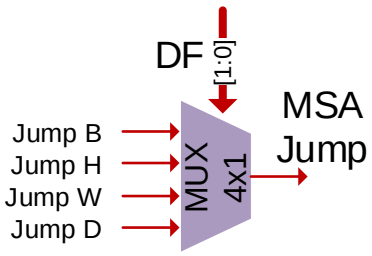


Figure 135: Format selector of branches