**POLITECNICO DI TORINO**
**Department of Electronic Engineering**


**Master degree in Electronic Engineering - Embedded systems**

---

# Speed-up of RISC-V core using Logic-in-Memory operations

**Supervisors**:
Marco Vacca - Politecnico di Torino
Marco Ottavi - Università degli Studi di Roma Tor Vergata

**Candidate:**
Antonia Ieva - s253237

---

**Academic year**
**2019 - 2020**

## Abstract

The memory-wall is a known issue in modern computing systems, that states the difference in terms of speed between memory and processor in a typical Von-Neumann architecture. The memory low speed masks the actual processor speed, becoming the bottleneck of the communication between these units. To overcome this problem, many research works started moving towards new memory architectures that allow to increase the communication speed or to distribute part of the arithmetic computations in the memory itself.
The Logic-in-Memory (LiM) is a new memory architecture that offers the possibility to have a unit that merges storage and computational capabilities. The literature offers many applications of the Logic-in-Memory concept, but this work aims to integrate the Logic-in-Memory architecture in a real-world computing system such as RI5CY. The RI5CY processor, part of the RISC-V family, has been chosen because it offers the possibility to customise the available Instruction Set Architecture (ISA) and then to add new instructions that support the new memory potentials.
The intention of this work is to show that the new memory architecture improves the average programs execution time, because it reduces the number of memory accesses by performing some logical operations or entire algorithms directly in memory.
The Logic-in-Memory proposed, is able to perform bitwise operations between some memory locations and a given input mask, and it can also compute the maximum and minimum value of the data stored.
To integrate this new memory into the RI5CY core, two solutions have been explored: 'Same Interface' and 'New Interface'. In the 'Same Interface' solution, the RI5CY introduces new instructions to coordinate the new memory operations, but it maintains the original interface with the memory. Instead, in the 'New Interface' solution, the interface is changed to maximise the efficiency of the memory operations, therefore the new instructions are adapted accordingly.
Both implementations show an important reduction of the execution time in the tested algorithms, so the Logic-in-Memory can be considered a valid approach to overcome the memory-wall problem and in general to speed-up the programs execution.

*Keywords*: Logic-in-Memory, RISC-V, memory-wall

# Contents


**1 Problem statement** **1**

**2 RISC-V ISA** **3**
1 RISC-V ISA overview . . . 3
1.1 Extensions . . . 3
1.2 Instruction Encoding . . . 4
2 RV32I Base Integer Instruction Set . . . 5
2.1 User visible registers . . . 5
2.2 Base Instruction Formats . . . 6
2.3 Integer Computational Instructions . . . 6
2.4 Control Transfer Instructions . . . 8
2.5 Load and Store Instructions . . . 9
2.6 Control and Status Register Instructions . . . 10
3 RV32M Base Integer Instruction Set . . . 11
3.1 Multiplication Instructions . . . 12
3.2 Division Instructions . . . 12
4 RV32F Single-Precision Floating-Point Instruction Set . . . 12
4.1 User visible registers . . . 12
4.2 Floating-Point Control and Status Register Instructions . . . 12
4.3 Single-Precision Load and Store Instructions . . . 14
4.4 Single-Precision Floating-Point Computational Instructions . . . 14
4.5 Single-Precision Floating-Point Conversion Instructions . . . 15
4.6 Single-Precision Floating-Point Move Instructions . . . 16
4.7 Single-Precision Floating-Point Compare Instructions . . . 16
4.8 Single-Precision Floating-Point Classify Instruction . . . 17
5 RV32C Compressed Instructions . . . 17
5.1 Load and Store Instructions . . . 18
5.2 Control Transfer Instructions . . . 19
5.3 Integer Computational Instructions . . . 20

**3 RIC5Y microprocessor** **24**
1 Introduction . . . 24
2 Supported ISA . . . 24
3 RTL top view . . . 26
3.1 Block diagram . . . 26
3.2 Interfaces . . . 26
4 Architecture description . . . 29
4.1 Instruction Fetch stage . . . 29
4.2 Decode stage . . . 33
4.3 Execution stage . . . 37
4.4 Load and Store Unit stage - Write Back stage . . . 39

4.5 Peripherals and Memory model . . . . . 40

**4 Logic-in-Memory in RI5CY Framework 42**
1 Logic-in-Memory State of Art . . . . . 42
2 Logic-in-Memory architecture . . . . . 44
2.1 Bitwise operations - Logic-in-Memory cell . . . . . 45
2.2 Maximum and minimum computation - logic around array . . . . . 46
2.3 Range operations . . . . . 47
3 Logic-in-Memory ISA extension . . . . . 48
3.1 Same interface Memory-Processor ISA extension . . . . . 48
3.2 New interface Memory-Processor ISA extension . . . . . 51
4 Architectural changes in RISC-V project . . . . . 53
4.1 Same interface Memory-Processor RI5CY change . . . . . 53
4.2 Same interface Memory-Processor RI5CY changes . . . . . 54
4.3 Differences between the Logic-in-Memory implementations . . . . . 55

**5 Simulations and Synthesis 57**
1 Tools . . . . . 57
2 Simulation with custom programs . . . . . 57
2.1 Bitwise . . . . . 58
2.2 Max-Min . . . . . 61
3 Simulation with standard programs . . . . . 64
3.1 Database search with Bitmap Indexes algorithm . . . . . 64
3.2 AES Addroundkey algorithm . . . . . 68
3.3 Transport problem - Least Cost Method algorithm . . . . . 71
4 Simulation Results Analysis . . . . . 75
5 Synthesis . . . . . 76

**6 Conclusions and Future Work 78**

**A System Verilog basics 80**
1 Introduction . . . . . 80
2 Data objects and data types . . . . . 80
2.1 Data types . . . . . 80
2.2 Data objects . . . . . 82
3 Literal Values . . . . . 83
4 Operators . . . . . 84
5 Signals and Constants . . . . . 84
6 Continuous assignments . . . . . 85
7 Procedural assignments . . . . . 86
7.1 Procedural blocks . . . . . 86
7.2 Procedural statements . . . . . 89
8 Design elements . . . . . 90
8.1 Module . . . . . 90
8.2 Interface . . . . . 93
8.3 Package . . . . . 94
8.4 Program . . . . . 94
9 Assertion . . . . . 95
9.1 Immediate assertions . . . . . 96
9.2 Concurrent assertions . . . . . 97
9.3 Binding assertion . . . . . 100
10 System tasks and system functions . . . . . 101

10.1 Simulation time functions . . . . . 102
10.2 Math functions . . . . . 102
10.3 Severity tasks . . . . . 102
10.4 Assertion tasks . . . . . 103

**Appendices** **80**

# List of Figures


1.1 Memory-wall problem . . . . . 1

2.1 RISC-V instruction length encoding. . . . . 5
2.2 Integer registers . . . . . 5
2.3 Instruction formats . . . . . 6
2.4 Integer Register-Immediate instructions with I-type format . . . . . 7
2.5 Integer Register-Immediate instructions with special I-type format . . . . . 7
2.6 Integer Register-Immediate instructions with U-type format . . . . . 7
2.7 Integer Register-Register instructions with R-type format . . . . . 8
2.8 Control Transfer instruction with J-type format . . . . . 8
2.9 Control Transfer instruction with I-type format . . . . . 9
2.10 Control Transfer instructions with B-type format . . . . . 9
2.11 Load instructions with I-type format . . . . . 10
2.12 Store instructions with S-type format . . . . . 10
2.13 CSR instructions with I-type format . . . . . 11
2.14 Timer and Counter Instructions with I-type format . . . . . 11
2.15 Multiplication instructions with R-type format . . . . . 12
2.16 Division instructions with R-type format . . . . . 12
2.17 Floating-Point Register File . . . . . 13
2.18 Floating-Point Status and Control Register . . . . . 13
2.19 Single-Precision Load & Store Instructions with S and B format . . . . . 14
2.20 Single-Precision Register to Register Instructions with R-type format . . . . 14
2.21 Single-Precision Register to Register Fused Instructions with specific type format . . . . . 15
2.22 Single-Precision to Integer and Integer to Single-Precision Instructions with R-type format . . . . . 16
2.23 Single-Precision Move Instructions with R-type format . . . . . 16
2.24 Single-Precision Move Instructions with R-type format . . . . . 16
2.25 Single-Precision Comparison Instructions with R-type format . . . . . 17
2.26 Single-Precision Classify Instruction with R-type format . . . . . 17
2.27 Compressed 16-bit RVC instruction formats. . . . . 18
2.28 Three-bit registers rs1', rs2', and rd' . . . . . 18
2.29 Stack-pointer-based Load and Stores with CI-type format. . . . . 18
2.30 Stack-pointer-based Load and Stores with the CSS-type format . . . . . 19
2.31 Register-based Load and Stores with CL-type format . . . . . 19
2.32 Register-based Load and Stores with CS-type format . . . . . 19
2.33 Compressed Control Transfer instructions with CJ-type format . . . . . 20
2.34 Compressed Control Transfer instructions with CR-type format . . . . . 20
2.35 Compressed Control Transfer instructions with CB-type format . . . . . 20
2.36 Compressed Integer Constant-Generation instructions with CI-type format . 21
2.37 Compressed Integer Register-Immediate instructions with CI-type format . 21

2.38 Compressed Integer Register-Immediate instructions with CIW-type format 21
2.39 Compressed Integer Register-Immediate instructions with CI-type format . 22
2.40 Compressed Integer Register-Immediate instructions with CB-type format . 22
2.41 Compressed Integer Register-Immediate instructions with CB-type format . 22
2.42 Compressed Integer Register-Register instructions . . . . . 23
2.43 Compressed Integer Register-Register instructions . . . . . 23
2.44 Compressed NOP instruction . . . . . 23
2.45 Compressed breakpoint instruction with CR-type format . . . . . 23

3.1 RI5CY Hardware loops mapping in CSR address space . . . . . 25
3.2 RI5CY block diagram . . . . . 26
3.3 Timing diagram Instruction memory/cache communication protocol . . . . 27
3.4 Timing diagram Data memory/cache communication protocol . . . . . 28
3.5 RI5CY pipeline . . . . . 29
3.6 IF stage block diagram . . . . . 30
3.7 ID stage block diagram . . . . . 34
3.8 EX stage block diagram . . . . . 38
3.9 LSU stage block diagram . . . . . 39
3.10 Peripherals and Memory model organisation . . . . . 41

4.1 Logic-in-Memory typologies . . . . . 43
4.2 Dual port Logic-in-Memory high level architecture . . . . . 45
4.3 Logic-in-Memory bit-cell . . . . . 45
4.4 Around-array logic for max-min computation . . . . . 46
4.5 Range decoder . . . . . 48
4.6 RISC-V-Logic-in-Memory interface in the 'Same interface' implementation . 49
4.7 New ISA for 'Same interface' implementation . . . . . 49
4.8 Waveforms for 'Same interface' implementation . . . . . 50
4.9 RISC-V-Logic-in-Memory interface in the 'New interface' implementation . 51
4.10 New ISA for 'New interface' implementation . . . . . 51
4.11 Waveforms for 'New interface' implementation . . . . . 52
4.12 ID stage architectural change for 'Same interface' implementation . . . . . 53
4.13 ID stage architectural change for 'New interface' implementation . . . . . 54
4.14 LSU stage architectural change for 'New interface' implementation . . . . . 55
4.15 Logic-in-Memory implementation differences . . . . . 55

5.1 Estimation execution time bitwise.c in old_ISA, newIF_ISA and sameIF_ISA 59
5.2 Estimation execution time max_min.c in old_ISA, newIF_ISA and sameIF_ISA 63
5.3 Bitmap indexes example: query result . . . . . 65
5.4 Estimation execution time bitmap_search.c in old_ISA, newIF_ISA and sameIF_ISA . . . . . 67
5.5 AES encryption algorithm . . . . . 69
5.6 Estimation execution time transport_min_cost.c in old_ISA, newIF_ISA and sameIF_ISA . . . . . 74

# List of Tables


2.1 Floating-point format encoding . . . . . 15
2.2 Domains of float-to-integer conversions and behavior for invalid inputs . . . 15
2.3 Format of result of FCLASS instruction. . . . . 17

3.1 Instruction memory/cache communication protocol . . . . . 27
3.2 Data memory/cache communication protocol . . . . . 28

5.1 Bitmap indexes example: students age ranges mapped with bits . . . . . 65
5.2 Transport problem example . . . . . 72
5.3 Simulation results comparison . . . . . 76
5.4 Synthesis results comparison . . . . . 77

A.1 SystemVerilog operators . . . . . 84

# Chapter 1

# Problem statement

Nowadays, every computing system typically includes a microprocessor, that performs arithmetic operations, and a memory, used to read instructions and store data. A computing system that has a shared memory for instructions and data is classified as Von-Neumann architecture.
Over the last decades the technology progress led CPUs to become faster and faster. Advanced design techniques, such as pipelining, contributed to increase even more the clock frequency of microprocessors. Nevertheless, the improvements adopted to increase the processor speed are masked by the much slower improvement in memory speed. As a matter of fact, the *memory-wall* problem is defined as the gap in terms of speed between the processor and the memory (see Figure 1.1) [1].

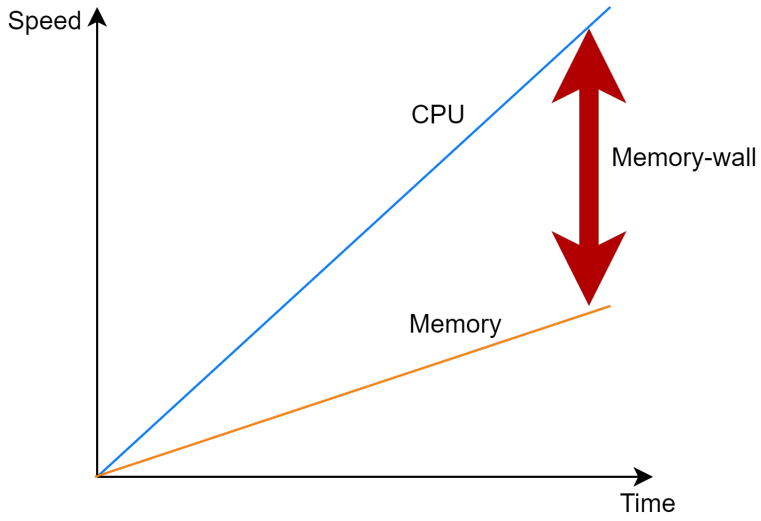



Figure 1.1: Memory-wall problem

Many solutions have been adopted to overcome this issue. The most common solution, available in all computing systems, is the memory hierarchy. In fact, the memory hierarchy classifies the memories according to speed, complexity and capacity. Typically smaller and faster memories are placed really close the microprocessor (e.g. cache memories), while bigger and slower memories are placed far from the microprocessor and do not interact directly with it (DRAM). The most frequently-accessed data or instructions are copied into the high-speed memories, therefore the access to less-frequent instructions and data stored into the low-speed memories is done less often.

This thesis aims to introduce the Logic-in-Memory (LiM) concept as an alternative solution to the stated problem.
The Logic-in-Memory mixes logic and memory in the same device, so that the workload of the arithmetic computations is not concentrated only on the CPU, but is distributed between CPU and memory. The central issue of the *memory-wall* problem is the constant communication between CPU and memory. As a matter of fact, the LiM is a valid solution because it would guarantee a less number of accesses in memory. In fact, a partial data manipulation can be performed directly in the memory itself. As a consequence, there would be a reduction of the number of operations performed with the memory frequency.

The work done in this research will focus on the architectural implementation of a Logic-in-memory model, and the related speed advantage it would guarantee in a computing system.

# Chapter 2

# RISC-V ISA

## 1 RISC-V ISA overview

The computing system chosen for this thesis is a RISC-V core. RISC-V is an Instruction Set Architecture (ISA) that was originally designed to support computer architecture research and education. However, it has now become a standard free with an open architecture for industry implementations. The flexibility of this computing system was the main reason why it was chosen for this Logic-in-Memory study.

However, the main features of the RISC-V ISA are summarised below:

- Freely available to academia and industry.
- Suitable for hardware implementation and suitable for any implementation technology: e.g. ASIC or FPGA.
- Extensive user-level ISA: the RISC-V has a base integer ISA, which must be present in any implementations, plus optional extensions to the base ISA.

### 1.1 Extensions

The base integer RISC-V provides a restricted set of instructions, sufficient to provide a reasonable target for compilers, assemblers, linkers, and operating systems, so exhaustive enough to build a software toolchain skeleton.
Around the base integer RISC-V is possible to build a more customized processor ISAs. As a matter of fact, the extensions to the basic ISA introduce instructions that provide new capabilities for architecture in order to improve code density and performance. While the base integer RISC-V is mandatory for any extensions, any customised RISC-V architecture can be expanded according one or more extensions.
Between all the possible RISC-V extensions, it is possible distinguish standard and non-standard extensions:

1. *Standard extensions* should be generally useful and should not conflict with other standard extensions.
2. *Non-standard extensions* may be highly specialized, or may conflict with other standard or non-standard extensions.

Most common standard extensions:

- **Base I**. The base integer ISA is named "I" (prefixed by RV32 or RV64 depending on the architecture parallelism), and contains integer computational instructions,

integer loads and stores in memory, and control-flow instructions, and is mandatory for all RISC-V implementations.

- **Extension C**. The standard compressed instruction set extension, named "C", aims to reduce static and dynamic code size by adding short 16-bit instruction encodings for common operations. The C extension can be added to any of the base ISAs.
- **Extension M**. The standard integer multiplication and division extension named "M", adds instructions to multiply and divide integer values.
- **Extension A**. The standard atomic instruction extension, denoted by "A", adds instructions that atomically read, modify, and write memory for synchronization purposes.
- **Extension F**. The standard single-precision floating-point extension, denoted by "F", adds floating-point registers, single-precision computational instructions, and single-precision loads and stores.
- **Extension D**. The standard double-precision floating-point extension, denoted by "D", expands the floating-point registers, and adds double-precision computational instructions, loads, and stores.

Usually the most common unions between different extensions is indicated with an abbreviation. For example, the group "IMAFD" has the abbreviation "G" and provides a general-purpose scalar instruction set, then the implementation is called RV32G or RV64G according to the parallelism. While, the case of study of this thesis will be the RI5CY that represents the group "IMFC", so in the next sections all the extensions related to this implementation will be analysed.

While standard extensions exploit the most common processor operations, non-standard extensions instead, can be created for any scopes and for any specialised operations that the processor might perform. Therefore, the power and advantage of RISC-V ISA is the possibility to be customised by anyone in order to meet the requirements of any specific applications.

## 1.2 Instruction Encoding

The base integer ISA has a fixed instructions lenght equal to 32 bits. However, the standard RISC-V encoding scheme is designed to support ISA extensions with variable-length instructions, where each instruction can have a length equal to a multiple of 16 bits.

- All the 32-bit instructions in the base ISA have their lowest two bits set to 11.
- The optional compressed 16-bit instruction-set extensions have their lowest two bits equal to 00, 01, or 10.
- Standard instruction-set extensions encoded with more than 32 bits have additional low-order bits set to 1, with the conventions for 48-bit and 64-bit lengths shown in Figure 2.1.
- Instruction lengths between 80 bits and 176 bits are encoded using a 3-bit field in bits [14:12] giving the number of 16-bit words in addition to the first 5x16-bit words. The encoding with bits [14:12] set to 111 is reserved for future longer instruction encodings.

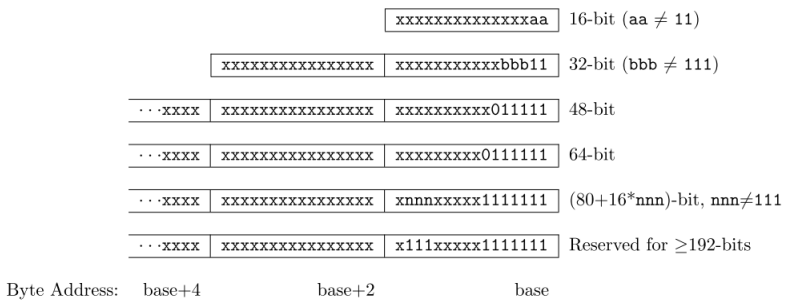



Figure 2.1: RISC-V instruction length encoding.

# 2 RV32I Base Integer Instruction Set

This section shows the details about the base integer RISC-V ISA.

## 2.1 User visible registers

RISC-V ISA guarantees the presence of 31 general-purpose registers x1–x31 in the RISC-V core which hold integer values, as shown in Figure 2.2. Register x0 is hardwired to the constant 0. The term XLEN is used to refer to the width of an x register in bits (either 32 in RV32 or 64 in RV64). However, even most of the instructions are independent on the length of the registers, in this thesis will be considered only instructions supported for a 32-bit core.

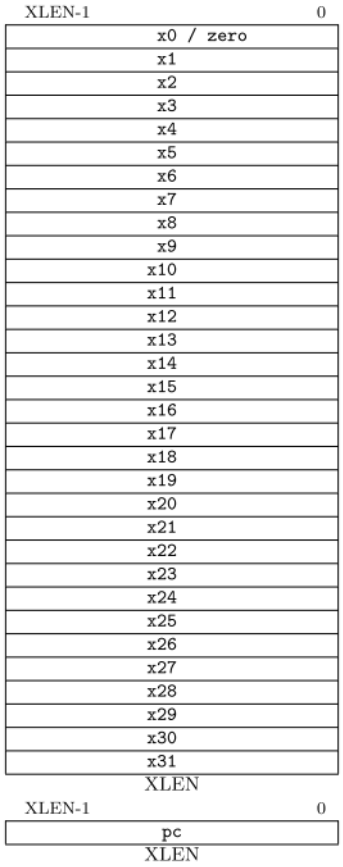



Figure 2.2: Integer registers

## 2.2 Base Instruction Formats

In the base ISA, there are six instruction formats (R/I/S/U/B/J) as shown in Figure 2.3. All formats have a fixed 32 bits in length and must be aligned on a four-byte boundary in memory.
RISC-V ISA keeps the source (rs1 and rs2) and destination (rd) registers at the same position in all formats to simplify decoding.

- **R format**: it is very straightforward. In fact, R type instructions have both sources that are registers.
- **I format**: the sources are one immediate and one register.
- **S format**: it manages one immediate and two source registers.
- **B format**: as the S type, it has one immediate and two source registers. The 12-bit immediate field is used to encode branch offsets in multiples of 2.
- **U format**: it has an immediate as the only source. The 20-bit immediate is shifted left by 12 bits.
- **J format**: as the U type, it has one immediate as source. The 20-bit immediate is shifted left by 1 bit in this case.

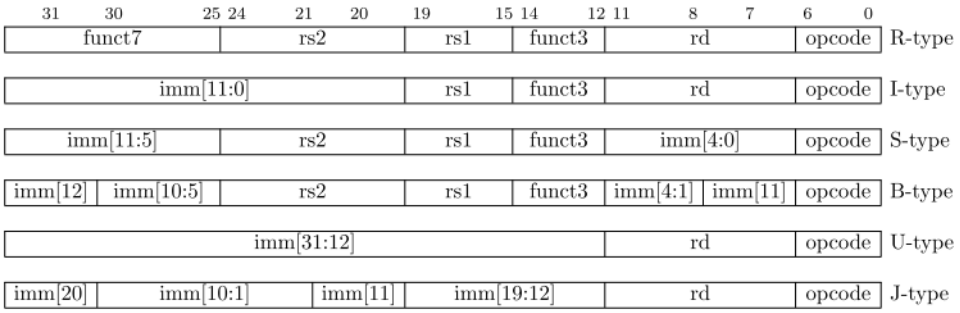



Figure 2.3: Instruction formats

## 2.3 Integer Computational Instructions

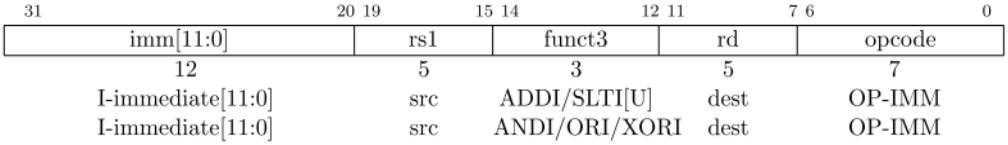
Integer computational instructions are either encoded as register-immediate operations using the I-type format or as register-register operations using the R-type format. The destination is register rd for both register-immediate and register-register instructions. Integer computational instructions do not cause arithmetic exceptions.

### 2.3.1 Integer Register-Immediate Instructions

Instructions with I-type format in Figure 2.4:

- **ADDI**. It adds the sign-extended 12-bit immediate to register rs1. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result.
- **NOP**. This instruction does not change any user-visible states, except for advancing the Program Counter (PC). NOP is encoded as ADDI x0, x0, 0.
- **SLTI**. It means "set less than immediate" and it places the value 1 in register rd if register rs1 is less than the sign-extended immediate, else 0 is written to rd. rs1 is also treated as a signed-number.

- **SLTIU**. It is similar the previous one, but compares the values as unsigned numbers.
- **ANDI, ORI, XORI**. They are logical operations that perform bitwise AND, OR, and XOR on register rs1 and the sign-extended 12-bit immediate and place the result in rd.

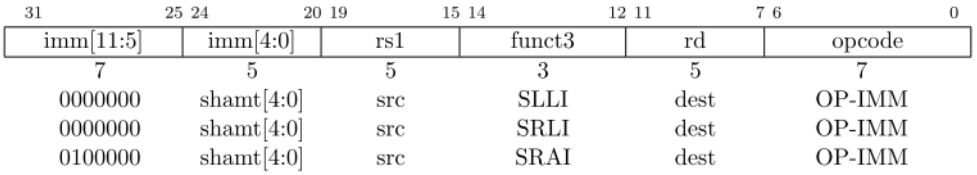


| 31 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|
| imm[11:0] | rs1 | funct3 | rd | opcode |
| 12 | 5 | 3 | 5 | 7 |
| I-immediate[11:0] | src | ADDI/SLTI[U] | dest | OP-IMM |
| I-immediate[11:0] | src | ANDI/ORI/XORI | dest | OP-IMM |

Figure 2.4: Integer Register-Immediate instructions with I-type format

Figure 2.5 lists instructions that use the I-type format in a specialized way. The immediate number is encoded in such a way a shift by a constant operation is performed. The operand to be shifted is in rs1, and the shift amount is encoded in the lower 5 bits of the I-immediate field. The right/left shift type is encoded in a high bit of the I-immediate.

- **SLLI**. It is a logical left shift (zeros are shifted into the lower bits).
- **SRLI**. It is a logical right shift (zeros are shifted into the upper bits).
- **SRAI**. It is an arithmetic right shift (the original sign bit is copied into the vacated upper bits).

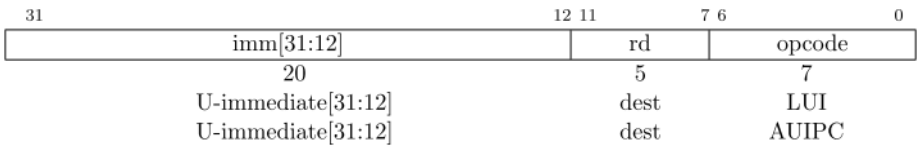


| 31 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| imm[11:5] | imm[4:0] | rs1 | funct3 | rd | opcode |
| 7 | 5 | 5 | 3 | 5 | 7 |
| 0000000 | shamt[4:0] | src | SLLI | dest | OP-IMM |
| 0000000 | shamt[4:0] | src | SRLI | dest | OP-IMM |
| 0100000 | shamt[4:0] | src | SRAI | dest | OP-IMM |

Figure 2.5: Integer Register-Immediate instructions with special I-type format

Figure 2.6 lists instructions that use the U-type format:

- **LUI**. It means "load upper immediate", it is used to build 32-bit constants and uses the U-type format. It places the U-immediate value in the top 20 bits of the destination register rd, filling in the lowest 12 bits with zeros.
- **AUIPC**. It means "add upper immediate to PC", it is used to build PC-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the PC, then places the result in register rd.



| 31 12 | 11 7 | 6 0 |
|---|---|---|
| imm[31:12] | rd | opcode |
| 20 | 5 | 7 |
| U-immediate[31:12] | dest | LUI |
| U-immediate[31:12] | dest | AUIPC |

Figure 2.6: Integer Register-Immediate instructions with U-type format

### 2.3.2 Integer Register-Register Operations

This set of instructions use the R-type format, where rs1 and rs2 are source registers and the result of an operation is written into the destination register rd. The funct7 and funct3

fields select the type of operation.
Figure 2.7 shows all the instructions of this type.

- **ADD, SUB**. They perform addition and subtraction respectively. Overflows are ignored and the low 32 bits of results are written to the destination.
- **SLT, SLTU**. They perform signed and unsigned compares respectively, writing 1 to rd if rs1 < rs2, 0 otherwise.
- **SLL, SRL, SRA**. They perform logical left, logical right, and arithmetic right shifts on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2.

| 31 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| funct7 | rs2 | rs1 | funct3 | rd | opcode |
| 7 | 5 | 5 | 3 | 5 | 7 |
| 0000000 | src2 | src1 | ADD/SLT/SLTU | dest | OP |
| 0000000 | src2 | src1 | AND/OR/XOR | dest | OP |
| 0000000 | src2 | src1 | SLL/SRL | dest | OP |
| 0100000 | src2 | src1 | SUB/SRA | dest | OP |

Figure 2.7: Integer Register-Register instructions with R-type format

## 2.4 Control Transfer Instructions

RV32I provides two types of control transfer instructions: unconditional jumps and conditional branches.

### 2.4.1 Unconditional Jumps

- **JAL**. It uses the J-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the PC to form the jump target address. JAL stores the address of the instruction following the jump (PC+4) into register rd. See Figure 2.8. It is possible to fulfill unconditional jumps encoding JAL with rd=x0.
- **JALR**. It uses the I-type encoding. The target address is obtained by adding the 12-bit signed I-immediate to the register rs1, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (PC+4) is written to register rd. Register x0 can be used as the destination if the result is not required. See Figure 2.9

The JAL and JALR instructions will generate a misaligned instruction fetch exception if the target address is not aligned to a four-byte boundary. According to the execution environment, the exception should cause the execution of a trap handler.

| 31 | 30 21 | 20 | 19 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode |
| 1 | 10 | 1 | 8 | 5 | 7 |
| | offset[20:1] | | | dest | JAL |

Figure 2.8: Control Transfer instruction with J-type format

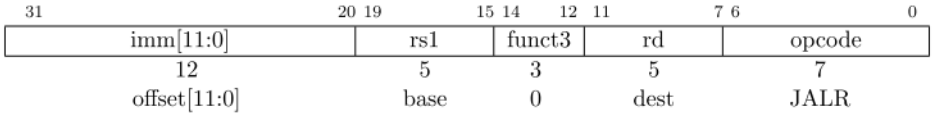



Figure 2.9: Control Transfer instruction with I-type format

### 2.4.2 Conditional Branches

All branch instructions use the B-type instruction format. The 12-bit B-immediate encodes a signed number, that corresponds to the offset to be added to the current PC to compute the target address.
Figure 2.10 lists the B-type instructions:

- **BQE, BNE**. They take the branch if registers rs1 and rs2 are equal or unequal respectively.
- **BLT, BLTU**. They take the branch if rs1 is less than rs2, using signed and unsigned comparison respectively.
- **BGE, BGEU**. They take the branch if rs1 is greater than or equal to rs2, using signed and unsigned comparison respectively.
- **BGT, BGTU, BGT, BGTU**. They can be synthesized by reversing the operands to BLT, BLTU, BGE, and BGEU, respectively.

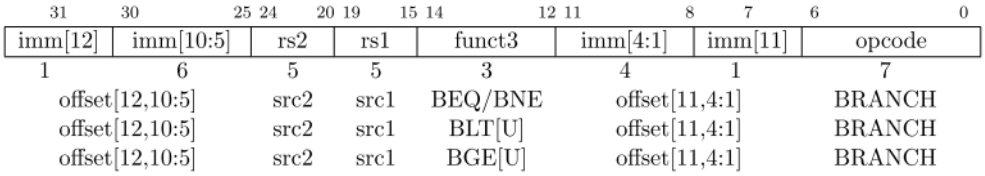



Figure 2.10: Control Transfer instructions with B-type format

## 2.5 Load and Store Instructions

RV32I is a load-store architecture, where only load and store instructions access the memory and arithmetic instructions only operate on CPU registers. RV32I provides a 32-bit user address space that is byte-addressed and little-endian. The execution environment will define what portions of the address space are legal to access. Loads with a destination of x0 must raise any exceptions and action any other side effects even though the load value is discarded.

Load and store instructions transfer a value between the registers and memory. Loads are encoded in the I-type format and stores are encoded in S-type. The effective byte address is obtained by adding register rs1 to the sign-extended 12-bit offset. Loads copy a value from memory to register rd (See Figure 2.11). Stores copy the value in register rs2 to memory (See Figure 2.12).

- **LW, LH, LHU, LB, LBU**. LW loads a 32-bit value from memory into rd. LHU loads a 16-bit value from memory, then sign-extends to 32-bits before storing in rd, while LHU do the same but performing a zero extension to 32 bits. LB and LBU are defined analogously to LH and LHU but for 8-bit values.

- **SW, SH, SB**. They store respectively 32-bit, 16-bit, and 8-bit values from the low bits of register rs2 to memory.

The base ISA supports misaligned accesses for data, but these might be very inefficient and slow, depending on the implementation. For this reason, only aligned loads and stores are guaranteed to execute atomically.

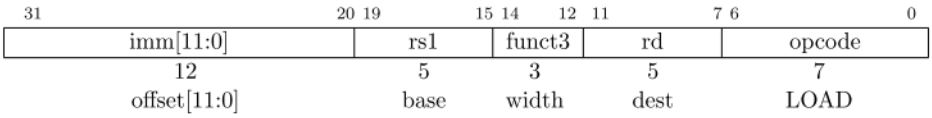



Figure 2.11: Load instructions with I-type format

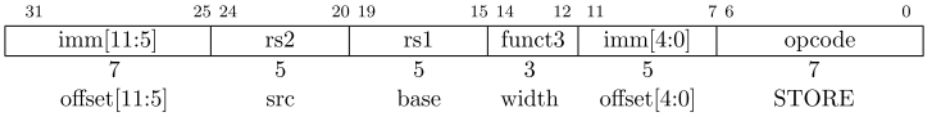



Figure 2.12: Store instructions with S-type format

## 2.6 Control and Status Register Instructions

System information is stored in special registers called Control Status Registers (CSRs). Those registers usually store information about the previous instruction executed and the operating mode. RV32I allows to access those registers with I-type instruction format. According to the implementation, these instructions can require privileged access to be executed.

### 2.6.1 CSR Instructions

In Figure 2.13 below, the full set of CSR instructions is defined:

- **CSRRW**. It means "Atomic Read/Write CSR", this instruction atomically swaps values in the CSRs and integer registers. CSRRW reads the old value of the CSR, zero-extends the value to 32 bits, then writes it to integer register rd. The initial value in rs1 is written to the CSR. If rd=x0, then the instruction should not read the CSR.
- **CSRRS**. It means "Atomic Read and Set Bits in CSR", this instruction reads the value of the CSR, zero-extends the value to 32 bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be set in the CSR. Any bit that is high in rs1 will cause the corresponding bit to be set in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected. As for the previous instruction, if rs1=x0, then the instruction will not write to the CSR at all.
- **CSRRC**. it means "Atomic Read and Clear Bits in CSR", this instruction reads the value of the CSR, zero-extends the value to 32 bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be cleared in the CSR. Any bit that is high in rs1 will cause the corresponding bit to be cleared in the CSR, if that CSR bit is writable. Other bits in the CSR are not changed. If rs1=x0, then the instruction will not write to the CSR at all.

- **CSRRWI, CSRRSI, CSRRCI**. They are variants of the previous ones and they are similar except they update the CSR using an 32-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field instead of a value from an integer register.
  For CSRRSI and CSRRCI, if the uimm[4:0] field is zero, then these instructions will not write to the CSR. For CSRRWI, if rd=x0, then the instruction should not read the CSR.

| 31 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|
| csr | rs1 | funct3 | rd | opcode |
| 12 | 5 | 3 | 5 | 7 |
| source/dest | source | CSRRW | dest | SYSTEM |
| source/dest | source | CSRRS | dest | SYSTEM |
| source/dest | source | CSRRC | dest | SYSTEM |
| source/dest | uimm[4:0] | CSRRWI | dest | SYSTEM |
| source/dest | uimm[4:0] | CSRRSI | dest | SYSTEM |
| source/dest | uimm[4:0] | CSRRCI | dest | SYSTEM |

Figure 2.13: CSR instructions with I-type format

### 2.6.2 Timers and Counters

RV32I provides a number of 64-bit read-only user-level counters, which are mapped into the 12-bit CSR address space and accessed in 32-bit pieces using CSRRS instructions.

- **RDCYCLE[H]**. The RDCYCLE pseudo-instruction reads the low 32 bits of the *cycle CSR* which holds a count of the number of clock cycles executed by the processor core which is running from an arbitrary start time in the past. RDCYCLEH is an RV32I-only instruction that reads bits 63–32 of the same cycle counter. Details like the cycle rate (cycles/second) of the counter will depend on the implementation.
- **RDTIME[H]**. The RDTIME pseudo-instruction reads the low 32 bits of the *time CSR*, which counts the real time that has passed from an arbitrary start time in the past. RDTIMEH is an RV32I-only instruction that reads bits 63–32 of the same real-time counter. The execution environment should provide means of determining the period of the real-time counter (seconds/tick).
- **RDINSTRET[H]**. This pseudo-instruction reads the low 32 bits of the *instret CSR*, which counts the number of instructions retired from some arbitrary start point in the past. RDINSTRETH is an RV32I-only instruction that reads bits 63–32 of the same instruction counter.

| 31 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|
| csr | rs1 | funct3 | rd | opcode |
| 12 | 5 | 3 | 5 | 7 |
| RDCYCLE[H] | 0 | CSRRS | dest | SYSTEM |
| RDTIME[H] | 0 | CSRRS | dest | SYSTEM |
| RDINSTRET[H] | 0 | CSRRS | dest | SYSTEM |

Figure 2.14: Timer and Counter Instructions with I-type format

# 3 RV32M Base Integer Instruction Set

The extension named "M" introduces the standard integer multiplication and division instructions, that multiply or divide values held in two integer registers.

### 3.1 Multiplication Instructions

- **MUL, MULH, MULHU, MULHSU**. MUL performs a 32-bit multiplication and places the lower 32 bits in the destination register. MULH, MULHU, and MULHSU perform the same multiplication but return the upper 32 bits of the full 32-bit product, for signed X signed, unsigned X unsigned, and signed X unsigned multiplication respectively.

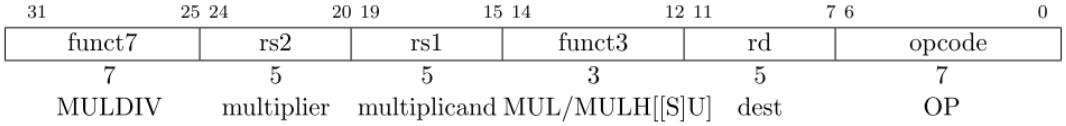



Figure 2.15: Multiplication instructions with R-type format

### 3.2 Division Instructions

- **DIV, DIVU**. They perform signed and unsigned integer division of 32 bits by 32 bits.
- **REM, REMU**. They provide the remainder of the corresponding division operation.

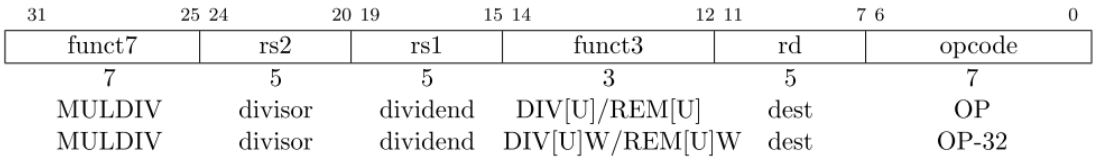



Figure 2.16: Division instructions with R-type format

## 4 RV32F Single-Precision Floating-Point Instruction Set

The "F" extension describes the standard instruction-set for single-precision floating-point numbers.

### 4.1 User visible registers

The F extension requires 32 floating-point registers, f0–f31, and a Floating-point Control and Status Register (FCSR), which contains the operating mode and exception status of the floating-point unit. Figure 2.17 shows the floating-point register file and the FCSR. The term FLEN describes the width of the floating-point registers in the RISC-V ISA. FLEN=32 corresponds to length for single-precision floating-point extension.

### 4.2 Floating-Point Control and Status Register Instructions

As for the integer CSRs, the Floating-point Control and Status Register (FCSR), is a 32-bit register than can be written or read. In fact, it allows to select the dynamic rounding mode for floating-point arithmetic operations and holds the arised exception flags, as shown in Figure 2.18.

- **FRCSR**. It reads FCSR by copying it into integer register rd.
- **FSCSR**. It swaps the value in FCSR by copying the original value into integer register rd, and then writing a new value obtained from integer register rs1 into FCSR.

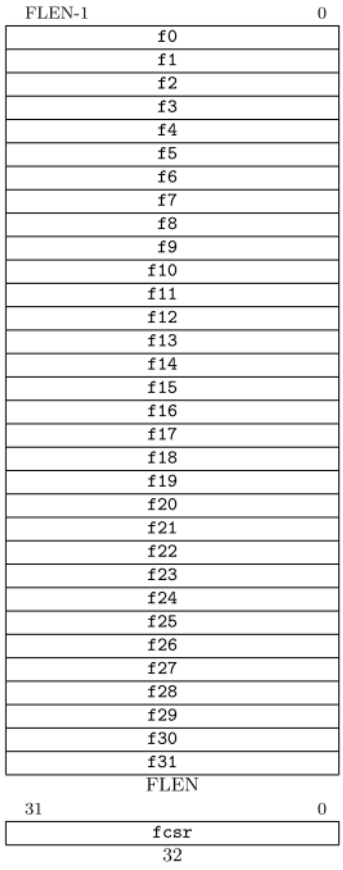



Figure 2.17: Floating-Point Register File

- **FRRM**. It reads the Rounding Mode field *frm* and copies it into the least-significant three bits of integer register rd, with zero in all other bits.
- **FSRM**. It swaps the value in *frm* by copying the original value into integer register rd and then, writing a new value obtained from the three least-significant bits of integer register rs1 into *frm*.
- **FRFLAGS, FSFLAGS**. Instructions defined for the Accrued Exception Flags *fflags*, then they respectively copy the field in rd and swap the values with register rs1.
- **FSRMI, FSFLAGSI**. Instructions defined to swap *frm* or *fflags* with an immediate value instead of a register rs1.

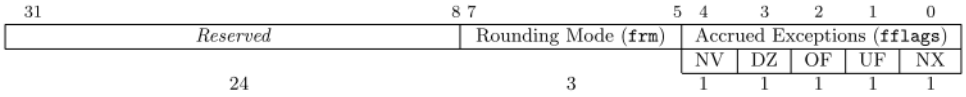



Figure 2.18: Floating-Point Status and Control Register

More details about the Accrued Exception Flags (*fflags*) or the Rounding Mode (*frm*) fields are not discussed because out of the scope of this work.

### 4.3 Single-Precision Load and Store Instructions

Floating-point loads and stores use the same base+offset addressing mode as the integer base ISA, with a base address in register rs1 and a 12-bit signed byte offset.

- **FLW**. The FLW instruction loads a single-precision floating-point value from memory into floating-point register rd.
- **FSW**. It stores a single-precision value from floating-point register rs2 to memory.

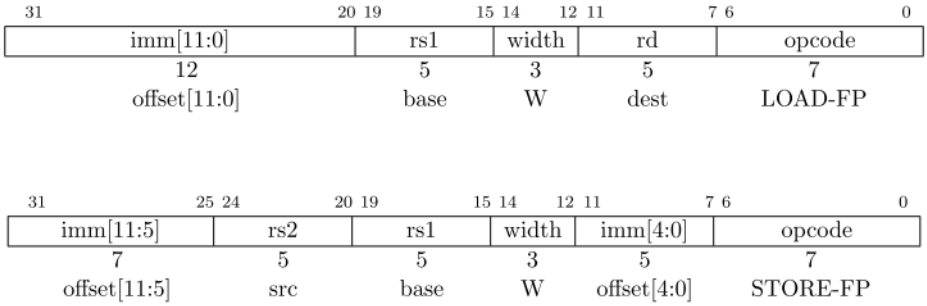



Figure 2.19: Single-Precision Load & Store Instructions with S and B format

### 4.4 Single-Precision Floating-Point Computational Instructions

Floating-point arithmetic instructions with one or two source operands use the R-type format with the OP-FP opcode. The following instructions perform single-precision floating-point operations between rs1 and rs2, writing the result to rd.

- **FADD.S**. It performs the addition.
- **FSUB.S**. It performs the subtraction.
- **FMUL.S**. It performs the multiplication.
- **FDIV.S**. It performs the division.
- **FMIN.S**. It writes the smaller between rs1 and rs2 to rd.
- **FMAX.S**. It writes the larger between rs1 and rs2 to rd.
- **FSQRT.S**. It computes the square root of rs1 and writes the result to rd.

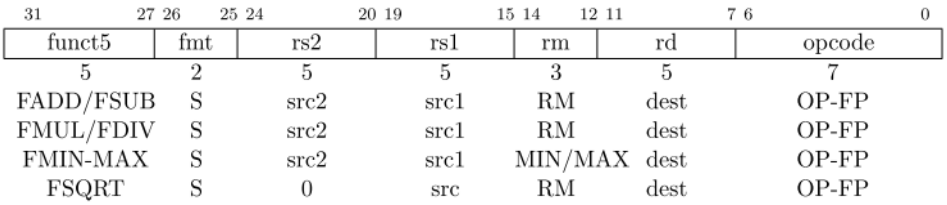



Figure 2.20: Single-Precision Register to Register Instructions with R-type format

The 2-bit floating-point format field *fmt* is encoded as shown in Table 2.1. It is set to S (00) for all instructions in the F extension.
Floating-point ISA combine multiply and add instructions using three source registers (rs1, rs2, and rs3) and a destination register (rd). Fused multiply-add instructions multiply the values in rs1 and rs2, optionally negate the product, then add or subtract the value in rs3, writing the final result to rd. The fused multiply-add instructions must raise the invalid

| fmt field | Mnemonic | Meaning |
|---|---|---|
| 00 | S | 32-bit single precision |
| 01 | D | 64-bit double-precision |
| 10 | - | reserved |
| 11 | Q | 128-bit quad-precision |

Table 2.1: Floating-point format encoding

operation exception when the multiplicands are $\infty$ and zero, even when the addend is a quiet NaN.

- **FMADD.S**. It computes rs1×rs2+rs3.
- **FMSUB.S**. It computes rs1×rs2-rs3.
- **FNMSUB.S**. It computes -rs1×rs2+rs3.
- **FNMADD.S**. It computes -rs1×rs2-rs3.

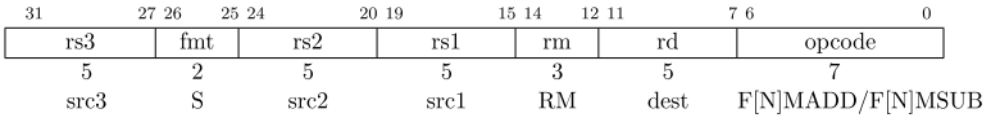



Figure 2.21: Single-Precision Register to Register Fused Instructions with specific type format

## 4.5 Single-Precision Floating-Point Conversion Instructions

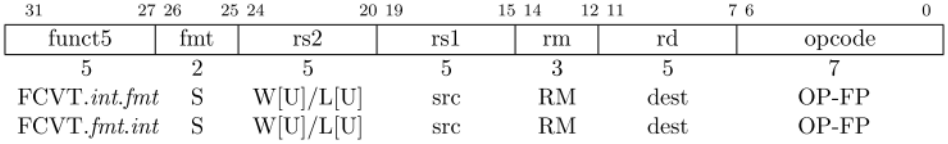
Floating-point-to-integer and integer-to-floating-point conversion instructions are encoded in the OP-FP opcode space.

- **FCVT.W.S**. It converts a floating-point number in floating-point register rs1 to a signed 32-bit, and writes it integer register rd.
- **FCVT.S.W**. It converts a 32-bit signed integer in integer register rs1 to a floating-point number in floating-point register rd.
- **FCVT.WU.S, FCVT.S.WU**. They are variants of the previous ones and convert to or from unsigned integer values.

If the rounded result does not fit in the destination format, it is clipped to the nearest value and the invalid flag is set. Table 2.2 gives the range of valid inputs for FCVT.int.S and the behavior for invalid inputs.

| | FCVT.W.S | FCVT.WU.S |
|---|---|---|
| Min valid input (after rounding) | $-2^{31}$ | 0 |
| Max valid input (after rounding) | $2^{31}-1$ | $2^{32}-1$ |
| Output for -$\infty$ | $-2^{31}$ | 0 |
| Output for +$\infty$ or NaN | $2^{31}-1$ | $2^{32}-1$ |

Table 2.2: Domains of float-to-integer conversions and behavior for invalid inputs



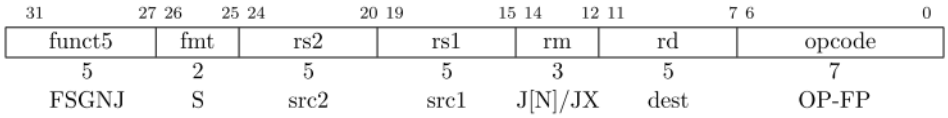
| 31 27 | 26 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|---|
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |
| 5 | 2 | 5 | 5 | 3 | 5 | 7 |
| FCVT.*int.fmt* | S | W[U]/L[U] | src | RM | dest | OP-FP |
| FCVT.*fmt.int* | S | W[U]/L[U] | src | RM | dest | OP-FP |

Figure 2.22: Single-Precision to Integer and Integer to Single-Precision Instructions with R-type format

## 4.6 Single-Precision Floating-Point Move Instructions

Floating-point to floating-point sign-injection instructions. Sign-injection instructions do not set floating-point exception flags (Figure 2.23):

- **FSGNJ.S**. It produces a result that takes all bits except the sign bit from rs1. The result's sign bit is rs2's sign bit.
- **FSGNJN.S**. It produces a result that takes all bits except the sign bit from rs1. The result's sign bit is the opposite of rs2's sign bit.
- **FSGNJX.S**. It produces a result that takes all bits except the sign bit from rs1. The sign bit is the XOR of the sign bits of rs1 and rs2.

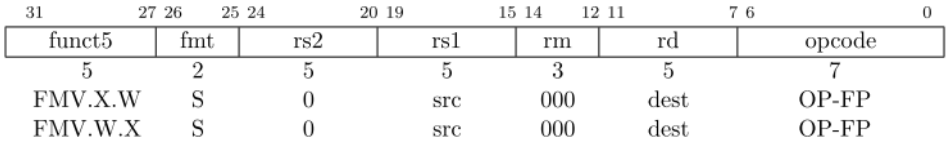


| 31 27 | 26 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|---|
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |
| 5 | 2 | 5 | 5 | 3 | 5 | 7 |
| FSGNJ | S | src2 | src1 | J[N]/JX | dest | OP-FP |

Figure 2.23: Single-Precision Move Instructions with R-type format

Other instructions are provided to move bit patterns between the floating-point and integer registers.

- **FMV.X.W**. It moves the single-precision value in floating-point register rs1 to the lower 32 bits of integer register rd.
- **FMV.W.X**. it moves the single-precision value from the lower 32 bits of integer register rs1 to the floating-point register rd. The bits are not modified in the transfer.



| 31 27 | 26 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|---|
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |
| 5 | 2 | 5 | 5 | 3 | 5 | 7 |
| FMV.X.W | S | 0 | src | 000 | dest | OP-FP |
| FMV.W.X | S | 0 | src | 000 | dest | OP-FP |

Figure 2.24: Single-Precision Move Instructions with R-type format

## 4.7 Single-Precision Floating-Point Compare Instructions

Floating-point compare instructions perform the specified comparison (equal, less than, or less than or equal) between floating-point registers rs1 and rs2 and record the boolean result in integer register rd.

- **FLT.S, FLE.S**. They perform the *signaling* comparison: an Invalid Operation exception is raised if either input is NaN.
- **FEQ.S**. It performs a *quiet* comparison: it does not cause an Invalid Operation exception.

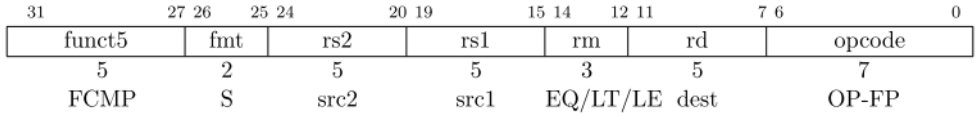



Figure 2.25: Single-Precision Comparison Instructions with R-type format

### 4.8 Single-Precision Floating-Point Classify Instruction

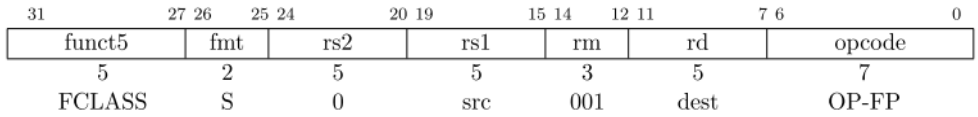



Figure 2.26: Single-Precision Classify Instruction with R-type format

- **FCLASS.S**. This instruction examines the value in floating-point register rs1 and writes to integer register rd a 10-bit mask that indicates the class of the floating-point number. The format of the mask is described in Table 2.3. The corresponding bit in rd will be set if the property is true and clear otherwise. All other bits in rd are cleared. Note that exactly one bit in rd will be set.

| rd bit | Meaning |
|---|---|
| 0 | rs1 is $-\infty$. |
| 1 | rs1 is a negative normal number. |
| 2 | rs1 is a negative subnormal number. |
| 3 | rs1 is $-0$. |
| 4 | rs1 is $+0$. |
| 5 | rs1 is a positive subnormal number. |
| 6 | rs1 is a positive normal number. |
| 7 | rs1 is $+\infty$. |
| 8 | rs1 is a signaling NaN. |
| 9 | rs1 is a quiet NaN. |

Table 2.3: Format of result of FCLASS instruction.

## 5 RV32C Compressed Instructions

This standard describes the extension named "C", that contains the RISC-V standard compressed instructions. This extension reduces static and dynamic code size by adding short 16-bit instruction encodings for common instructions. The C extension can be added to any of the base ISAs (RV32, RV64, RV128), and the generic term “RVC” covers any of these. Typically, 50%–60% of the RISC-V instructions in a program can be replaced with RVC instructions, resulting in a 25%–30% code-size reduction.
RVC uses a simple compression scheme that offers shorter 16-bit versions of common 32-bit RISC-V instructions when:

- the immediate or address offset is small;
- one of the registers is the zero register (x0), the ABI link register (x1), or the ABI stack pointer (x2);
- the destination register and the first source register are identical;
- the registers used belong to a subset of 8 registers.

The C extension allows 16-bit instructions to be freely intermixed with 32-bit instructions, with the latter now able to start on any 16-bit boundaries. With the addition of the C extension, JAL and JALR instructions will no longer raise an instruction misaligned exception.
Figure 2.27 shows the eight compressed instruction formats. CR, CI, and CSS can use any of the 32 RVI registers, but CIW, CL, CS, and CB are limited to just 8 of them. Figure 2.28 lists these subset of registers, which correspond to registers range from x8 to x15.

| Format | Meaning | 15 14 13 | 12 | 11 10 | 9 8 7 | 6 5 4 3 2 | 1 0 |
|---|---|---|---|---|---|---|---|
| CR | Register | funct4 | | rd/rs1 | | rs2 | op |
| CI | Immediate | funct3 | imm | rd/rs1 | | imm | op |
| CSS | Stack-relative Store | funct3 | imm | | | rs2 | op |
| CIW | Wide Immediate | funct3 | imm | | | rd′ | op |
| CL | Load | funct3 | imm | rs1′ | imm | rd′ | op |
| CS | Store | funct3 | imm | rs1′ | imm | rs2′ | op |
| CB | Branch | funct3 | offset | rs1′ | offset | | op |
| CJ | Jump | funct3 | jump target | | | | op |

Figure 2.27: Compressed 16-bit RVC instruction formats.

| RVC Register Number | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| Integer Register Number | x8 | x9 | x10 | x11 | x12 | x13 | x14 | x15 |
| Integer Register ABI Name | s0 | s1 | a0 | a1 | a2 | a3 | a4 | a5 |
| Floating-Point Register Number | f8 | f9 | f10 | f11 | f12 | f13 | f14 | f15 |
| Floating-Point Register ABI Name | fs0 | fs1 | fa0 | fa1 | fa2 | fa3 | fa4 | fa5 |

Figure 2.28: Three-bit registers rs1', rs2', and rd'

## 5.1 Load and Store Instructions

RVC provides two variants of loads and stores. One uses the ABI stack pointer x2, as the base address and can target any data register. The other can reference one of 8 base address registers and one of 8 data registers.

#### 5.1.0.1 Stack-Pointer-Based Loads and Stores

This first type of instructions use the CI format (Figure 2.29):

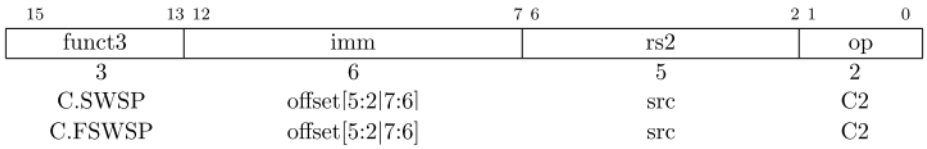
- **C.LWSP**. It loads a 32-bit value from memory into register rd. It computes an effective address by adding the zero-extended offset, scaled by 4, to the stack pointer x2.
- **C.FLWSP**. It is an RV32FC-only instruction that loads a single-precision floating-point value from memory into floating-point register rd. It computes its effective address by adding the zero-extended offset, scaled by 4, to the stack pointer x2.

| 15 13 | 12 | 11 7 | 6 2 | 1 0 |
|---|---|---|---|---|
| funct3 | imm | rd | imm | op |
| 3 | 1 | 5 | 5 | 2 |
| C.LWSP | offset[5] | dest≠0 | offset[4:2\|7:6] | C2 |
| C.FLWSP | offset[5] | dest | offset[4:2\|7:6] | C2 |

Figure 2.29: Stack-pointer-based Load and Stores with CI-type format.

These instructions use the CSS format (Figure 2.30):

- **C.SWSP**. It stores a 32-bit value in register rs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 4, to the stack pointer x2.
- **C.FSWSP**. It is an RV32FC-only instruction that stores a single-precision floating-point value in floating-point register rs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 4, to the stack pointer x2.



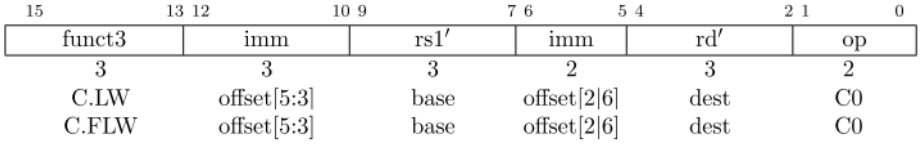
| 15 | 13 12 | 7 6 | 2 1 | 0 |
|---|---|---|---|---|
| funct3 | imm | rs2 | op | |
| 3 | 6 | 5 | 2 | |
| C.SWSP | offset[5:2\|7:6] | src | C2 | |
| C.FSWSP | offset[5:2\|7:6] | src | C2 | |

Figure 2.30: Stack-pointer-based Load and Stores with the CSS-type format

### 5.1.1 Register-based Loads and Stores

Load instructions use the CL format (Figure 2.31):

- **C.LW**. It loads a 32-bit value from memory into register rd'. It computes an effective address by adding the zero-extended offset, scaled by 4, to the base address in register rs1'.
- **C.FLW**. It is an RV32FC-only instruction that loads a single-precision floating-point value from memory into floating-point register rd'. It computes an effective address by adding the zero-extended offset, scaled by 4, to the base address in register rs1'.

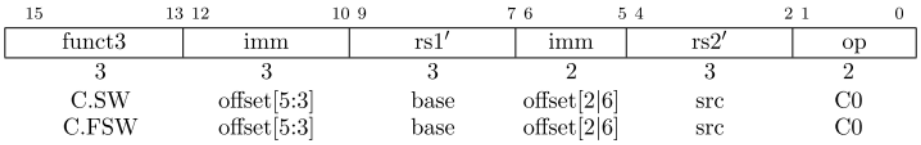


| 15 | 13 12 | 10 9 | 7 6 | 5 4 | 2 1 | 0 |
|---|---|---|---|---|---|---|
| funct3 | imm | rs1′ | imm | rd′ | op | |
| 3 | 3 | 3 | 2 | 3 | 2 | |
| C.LW | offset[5:3] | base | offset[2\|6] | dest | C0 | |
| C.FLW | offset[5:3] | base | offset[2\|6] | dest | C0 | |

Figure 2.31: Register-based Load and Stores with CL-type format

Store instructions use the CS format (Figure 2.32):

- **C.SW**. It stores a 32-bit value in register rs2' to memory. It computes an effective address by adding the zero-extended offset, scaled by 4, to the base address in register rs1'.
- **C.FSW**. It is an RV32FC-only instruction that stores a single-precision floating-point value in floating-point register rs2' to memory. It computes an effective address by adding the zero-extended offset, scaled by 4, to the base address in register rs1'.



| 15 | 13 12 | 10 9 | 7 6 | 5 4 | 2 1 | 0 |
|---|---|---|---|---|---|---|
| funct3 | imm | rs1′ | imm | rs2′ | op | |
| 3 | 3 | 3 | 2 | 3 | 2 | |
| C.SW | offset[5:3] | base | offset[2\|6] | src | C0 | |
| C.FSW | offset[5:3] | base | offset[2\|6] | src | C0 | |

Figure 2.32: Register-based Load and Stores with CS-type format

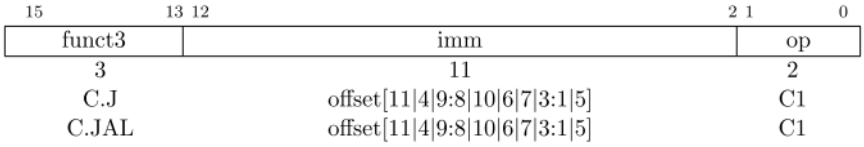
## 5.2 Control Transfer Instructions

RVC provides unconditional jump instructions and conditional branch instructions. The offsets of all RVC control transfer instruction are in multiples of 2 bytes.

### 5.2.1 Unconditional jumps

A group of unconditional jumps instructions use the CJ format (Figure 2.33):

- **C.J**. It performs an unconditional control transfer. The offset is sign-extended and added to the PC to form the jump target address. C.J can therefore target a ±2 KiB range.
- **C.JAL**. It is an RV32C-only instruction that performs the same operation as C.J, but additionally writes the address of the instruction following the jump (PC+2) to the link register x1.

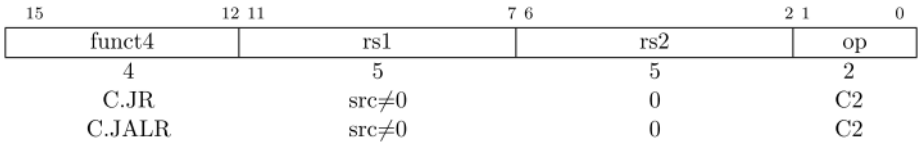


| 15 | 13 12 | 2 1 | 0 |
|---|---|---|---|
| funct3 | imm | op | |
| 3 | 11 | 2 | |
| C.J | offset[11\|4\|9:8\|10\|6\|7\|3:1\|5] | C1 | |
| C.JAL | offset[11\|4\|9:8\|10\|6\|7\|3:1\|5] | C1 | |

Figure 2.33: Compressed Control Transfer instructions with CJ-type format

Another group of instructions use the CR format (Figure 2.34):

- **C.JR**. This instruction (jump register) performs an unconditional control transfer to the address in register rs1.
- **C.JALR**. This instruction (jump and link register) performs the same operation as C.JR, but additionally writes the address of the instruction following the jump (PC+2) to the link register x1.



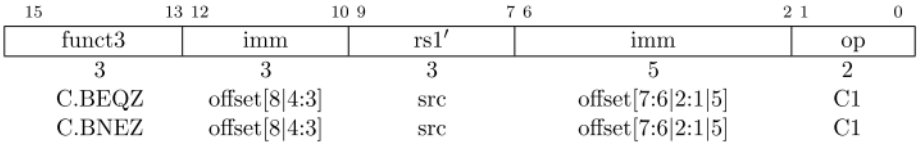
| 15 | 12 11 | 7 6 | 2 1 | 0 |
|---|---|---|---|---|
| funct4 | rs1 | rs2 | op | |
| 4 | 5 | 5 | 2 | |
| C.JR | src≠0 | 0 | C2 | |
| C.JALR | src≠0 | 0 | C2 | |

Figure 2.34: Compressed Control Transfer instructions with CR-type format

### 5.2.2 Conditional jumps

These instructions use the CB format (Figure 2.35):

- **C.BEQZ**. It performs conditional control transfers. The offset is sign-extended and added to the PC to form the branch target address. It can therefore target a ±256 B range. C.BEQZ takes the branch if the value in register rs1' is zero.
- **C.BNEZ**. It is defined analogously, but it takes the branch if rs1' contains a nonzero value.



| 15 | 13 12 | 10 9 | 7 6 | 2 1 | 0 |
|---|---|---|---|---|---|
| funct3 | imm | $rs1'$ | imm | op | |
| 3 | 3 | 3 | 5 | 2 | |
| C.BEQZ | offset[8\|4:3] | src | offset[7:6\|2:1\|5] | C1 | |
| C.BNEZ | offset[8\|4:3] | src | offset[7:6\|2:1\|5] | C1 | |

Figure 2.35: Compressed Control Transfer instructions with CB-type format

## 5.3 Integer Computational Instructions

RVC provides several instructions for integer arithmetic and constant generation.

### 5.3.1 Integer Constant-Generation Instructions

The two constant-generation instructions both use the CI instruction format and can target any integer registers:

- **C.LI**. It loads the sign-extended 6-bit immediate, into register rd. C.LI is only valid when rd6=x0.
- **C.LUI**. It loads the non-zero 6-bit immediate field into bits 17–12 of the destination register, clears the bottom 12 bits, and sign-extends bit 17 into all higher bits of the destination. C.LUI is only valid when rd6=x0, x2, and when the immediate is not equal to zero.

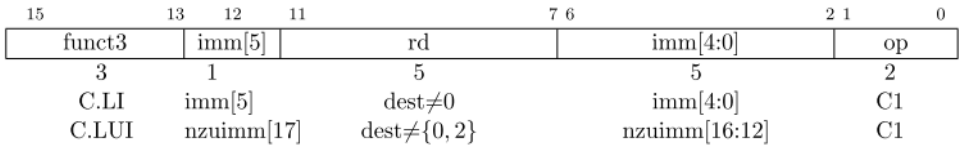



Figure 2.36: Compressed Integer Constant-Generation instructions with CI-type format

### 5.3.2 Integer Register-Immediate Operations

These integer register-immediate operations are encoded in the CI format and perform operations on any non-x0 integer registers and a 6-bit immediate. The immediate cannot be zero:

- **C.ADDI**. It adds the non-zero sign-extended 6-bit immediate to the value in register rd then writes the result to rd.
- **C.ADDI16SP**. It shares the opcode with C.LUI, but has a destination field of x2. C.ADDI16SP adds the non-zero sign-extended 6-bit immediate to the value in the stack pointer (sp=x2), where the immediate is scaled to represent multiples of 16 in the range (-512,496).

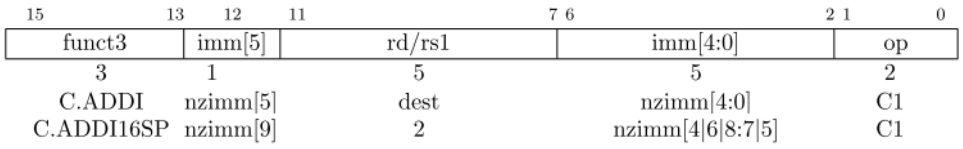



Figure 2.37: Compressed Integer Register-Immediate instructions with CI-type format

- **C.ADDI4SPN**. It is a CIW-format RV32C instruction that adds a zero-extended non-zero immediate, scaled by 4, to the stack pointer, x2, and writes the result to rd'.

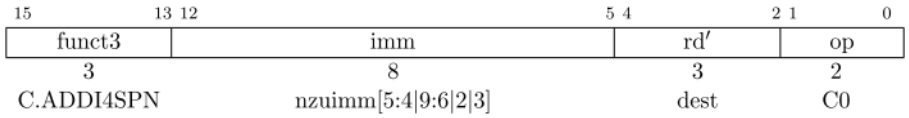



Figure 2.38: Compressed Integer Register-Immediate instructions with CIW-type format

- **C.SLLI**. It is a CI-format instruction that performs a logical left shift of the value in register rd then writes the result to rd. The shift amount is encoded in the shamt field, where shamt[5] must be zero for RV32C, but the whole shift amount must be non-zero.

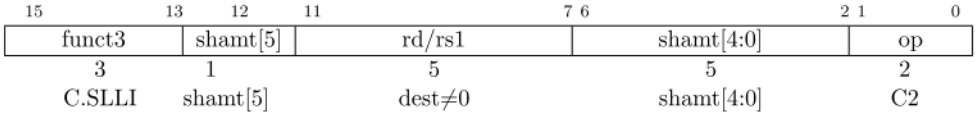



Figure 2.39: Compressed Integer Register-Immediate instructions with CI-type format

- **C.SRLI**. It is a CB-format instruction that performs a logical right shift of the value rd' then writes the result to rd'. The shift amount is encoded in the shamt field, where shamt[5] must be zero for RV32C, but the overall shamt must be non-zero.
- **C.SRAI**. It is defined analogously to C.SRLI, but instead performs an arithmetic right shift.

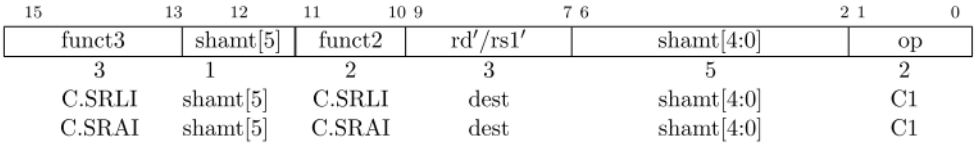



Figure 2.40: Compressed Integer Register-Immediate instructions with CB-type format

- **C.ANDI**. It is a CB-format instruction that computes the bitwise AND of the value in register rd' and the sign-extended 6-bit immediate, then writes the result to rd'.

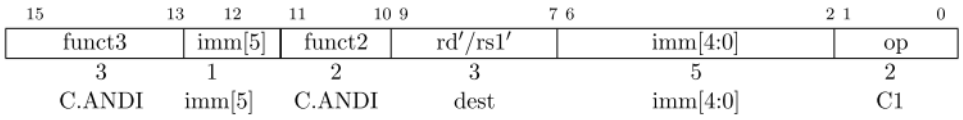



Figure 2.41: Compressed Integer Register-Immediate instructions with CB-type format

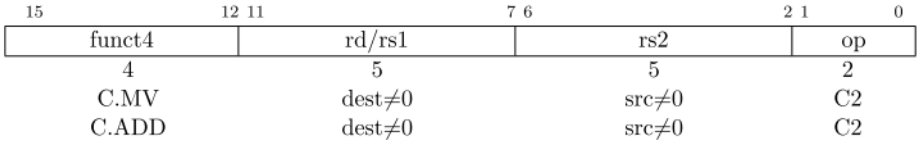
### 5.3.3 Integer Register-Register Operations

These first group of instructions uses the CR format (Figure 2.42):

- **C.MV**. It copies the value in register rs2 into register rd. C.MV expands into add rd, x0, rs2.
- **C.ADD**. It adds the values in registers rd and rs2 and writes the result to register rd.

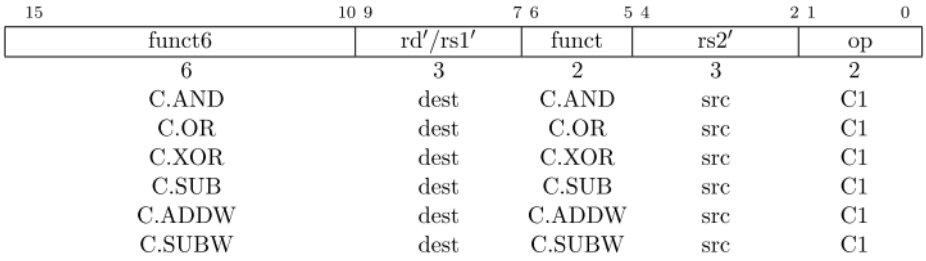
This other group of instructions instead, uses the CS format (Figure 2.43):

- **C.AND**. It computes the bitwise AND of the values in registers rd' and rs2' then writes the result to register rd'.
- **C.OR**. It computes the bitwise OR of the values in registers rd' and rs2', then writes the result to register rd'.
- **C.XOR**. It computes the bitwise XOR of the values in registers rd' and rs2', then writes the result to register rd'.
- **C.SUB**. It subtracts the value in register rd' from the value in register rs2', then writes the result to register rd'.



| 15 12 | 11 7 | 6 2 | 1 0 |
|---|---|---|---|
| funct4 | rd/rs1 | rs2 | op |
| 4 | 5 | 5 | 2 |
| C.MV | dest≠0 | src≠0 | C2 |
| C.ADD | dest≠0 | src≠0 | C2 |

Figure 2.42: Compressed Integer Register-Register instructions



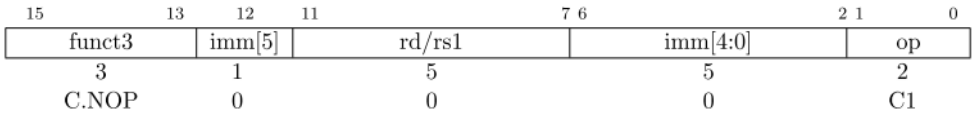
| 15 10 | 9 7 | 6 5 | 4 2 | 1 0 |
|---|---|---|---|---|
| funct6 | rd′/rs1′ | funct | rs2′ | op |
| 6 | 3 | 2 | 3 | 2 |
| C.AND | dest | C.AND | src | C1 |
| C.OR | dest | C.OR | src | C1 |
| C.XOR | dest | C.XOR | src | C1 |
| C.SUB | dest | C.SUB | src | C1 |
| C.ADDW | dest | C.ADDW | src | C1 |
| C.SUBW | dest | C.SUBW | src | C1 |

Figure 2.43: Compressed Integer Register-Register instructions

### 5.3.4 NOP Instruction

The NOP instruction uses the CS format (Figure 2.44):

- **C.NOP**. It is a CI-format instruction that does not change any user-visible state, except for advancing the PC.



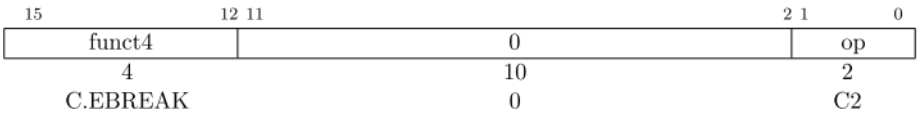
| 15 13 | 12 | 11 7 | 6 2 | 1 0 |
|---|---|---|---|---|
| funct3 | imm[5] | rd/rs1 | imm[4:0] | op |
| 3 | 1 | 5 | 5 | 2 |
| C.NOP | 0 | 0 | 0 | C1 |

Figure 2.44: Compressed NOP instruction

### 5.3.5 Breakpoint Instruction

This instruction uses the CR format (Figure 2.45):

- **C.EBREAK**. Debuggers can use the C.EBREAK instruction, to cause control to be transferred back to the debugging environment. C.EBREAK shares the opcode with the C.ADD instruction, but with rd and rs2 both zero.



| 15 12 | 11 2 | 1 0 |
|---|---|---|
| funct4 | 0 | op |
| 4 | 10 | 2 |
| C.EBREAK | 0 | C2 |

Figure 2.45: Compressed breakpoint instruction with CR-type format

# Chapter 3

# RIC5Y microprocessor

## 1 Introduction

The case of study of this thesis is the RI5CY or CV32E40P. This work started from the analysis of an already existing implementation developed and maintained by the PULP Platform (https://www.pulp-platform.org/), that offers many open source projects.
As already stated, the RIC5Y core has been chosen because of its flexibility that characterises the RISC-V standard family. In fact, as in all the RISC-V implementations it is possible to extend the ISA, in order to make the core able to support new custom functionalities. Therefore, the RI5CY core is a good candidate for the integration of the Logic-in-Memory architecture.
This chapter describes the features and the architecture of the initial version of the core, that has then been modified to support the new memory operations (https://github.com/openhwgroup/cv32e40p) [3].

## 2 Supported ISA

The RI5CY core supports standard and non-standard extensions. While the standard extensions have been widely discussed in the previous chapter, the non-standard extensions will be described at a very high level, because they are not the objective of this work.

- **Standard - RV32I Base Integer extension**. More details about the ISA in Section [ch. 2, 2].
- **Standard - RV32C Compressed extension**. Details about the ISA in Section [ch. 2, 5].
- **Standard - RV32M Integer Multiplication and Division extension**. Details about the ISA in Section [ch. 2, 3].
- **Standard - RV32F Single Precision Floating Point extension**. Details about the ISA in Section [ch. 2, 4]. This extension is optional. The hardware that manages the floating-point instructions can be enabled or disabled through the parameter "FPU" in the top-level file.
- **Non-Standard - Post-Incrementing load and store extension**. Post-incrementing load and store instructions perform a load/store operation from/to the data memory while at the same time increasing the base address by the specified offset. The memory access uses the base address without offset. Post-incrementing load and stores reduce the number of required instructions to execute code with regular data access

patterns, which can typically be found in loops. These post-incrementing load/store instructions allow the address increment to be embedded in the memory access instructions without the need of separate instructions. Coupled with the hardware loop extension, these instructions allow to reduce the loop overhead significantly.

- **Non-Standard - Multiply-Accumulate extension**. RI5CY supports non-standard extensions for multiply-accumulate and half-word multiplications with an optional post-multiplication shift.
- **Non-Standard - PULP ALU extension**. RI5CY supports advanced ALU operations that allow to perform multiple instructions that are specified in the base instruction set in one single instruction and then to increases efficiency of the core. For example, those instructions include zero-/sign-extension instructions for 8-bit and 16-bit operands, simple bit manipulation/counting instructions and min/max/avg instructions. The ALU does also support saturating, clipping, and normalising instructions which make fixed-point arithmetic more efficient.
- **Non-Standard - PULP Hardware Loops extension**. The following feature in RI5CY allows to use in a more efficient way the instructions for a loop.
  As a matter of fact, the hardware loops feature makes it possible to execute a piece of code multiple times, without the overhead of branches or updating a register counter. Hardware loops involve zero stall cycles for jumping to the first instruction of a loop. A hardware loop is implemented with three registers, that respectively represent the *start address* (pointing to the first instruction in the loop), the *end address* (pointing to the instruction that will be executed last in the loop) and a *counter* (decremented every time the loop body is executed).
  RI5CY contains two hardware loops (six registers, see Figure 3.1) to support nested hardware loops. If the *end address* registers of the two hardware loops are identical, loop 0 has higher priority and only the *counter* for hardware loop 0 is decremented. As soon as the *counter* of loop 0 reaches 1 when executing the instruction corresponding to the *end address*, loop 1 gets active too, because it means that counter of the loop 0 will be decremented to 0. In this case, both *counter* registers will be decremented and the core jumps to the start of loop 1.
  In order to use hardware loops, the loop instructions need to be preceded by the hardware loop instructions for the setup of the CSRs. The minimum loop size is two instructions and the last instruction cannot be any jump or branch instructions.
  For debugging and context switches, the hardware loop registers are mapped into the CSR address space and so it is possible to read and write them via CSR-like instructions (Section [ch. 2, 2.6]):

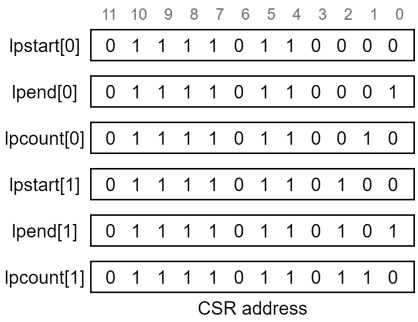



Figure 3.1: RI5CY Hardware loops mapping in CSR address space

Since hardware loop registers could be overwritten when processing interrupts, the

registers have to be saved during the interrupt routine together with the general purpose registers.

- **Non-Standard - PULP Vectorial extension**. Vectorial instructions perform operations in a Single Instruction-Multiple Data (SIMD) manner on multiple sub-word elements at the same time. This is done by segmenting the data path into smaller parts when 8 or 16-bit operations should be performed.
- **Non-Standard - PULP specific extension**. PULP Platform offers some single-core and multi-core micro-controllers based on the RI5CY. For this reason PULP-specific instructions are supported by the the RI5CY in order to interact with the System-on-Chip (SoC) components.

# 3 RTL top view

## 3.1 Block diagram

RI5CY is a 4-stage in-order 32-bit core. Figure 3.2 shows an overview of the RI5CY architecture. More details on the architecture will be covered in Section 4.

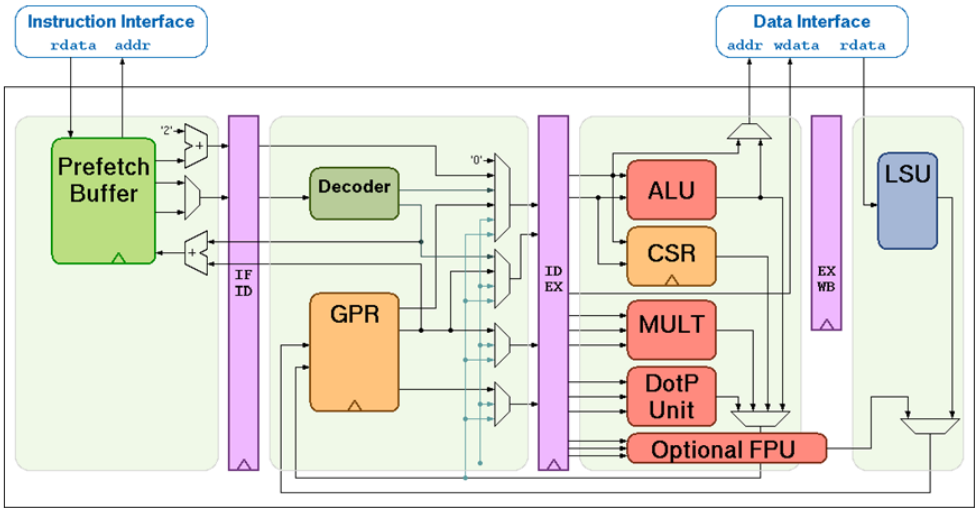



Figure 3.2: RI5CY block diagram

## 3.2 Interfaces

The RI5CY core as any microprocessors, must communicate with the memory to read the instructions and for exchange data information. The core can also communicate with external peripherals through interrupts and with the Auxiliary Processing Units (APU). Anyway, for the scope of this research, the focus will be on the interfaces related to the memory.

### 3.2.1 Instruction memory interface

The instruction fetcher of the core is able to supply one instruction per cycle if the instruction cache or the instruction memory is able to serve one instruction per cycle. The instruction address can be half-word-aligned due to the support of compressed instructions (RVC).

The protocol and the handshake signals used for this communication are explained below. The core can only perform reads on the instruction memory:

- **Read from memory**. The protocol requires that the core initiates the communication with the memory, when an instruction is needed. The request is performed with *instr_req_o* set and the *instr_addr_o* pointing to the needed instruction. These signals should remain stable until the memory sees the core request, by setting the signal *instr_gnt_i* for only one clock cycle. The address and the request signals are expected to change after that. Whenever the memory finishes to process the request, sets the signal *instr_rvalid_i* to let the core sample the instruction on the *instr_rdata_i* bus.

The Table 3.1 and the Timing diagram 3.3 summarise the communication protocol.

| Signal | Direction | Description |
|---|---|---|
| instr_req_o | output | RI5CY core uses the request signal to start the communication with the memory. This signal must stay high until *instr_gnt_i* is high for one cycle. |
| instr_gnt_i | input | This pulse signal is high when the memory accepted the request: *instr_addr_o* may change in the next cycle. |
| instr_addr_o[31:0] | output | Instruction address to read: sampled by the memory when *instr_gnt_i* and *instr_req_o* are high. |
| instr_rvalid_i | input | *instr_rdata_i* is considered valid when *instr_rvalid_i* is high, for exactly one cycle. |
| instr_rdata_i[31:0] | input | Instruction data read from memory: sampled by the prefetcher in RI5CY core when *instr_rvalid_i* is high. |

Table 3.1: Instruction memory/cache communication protocol

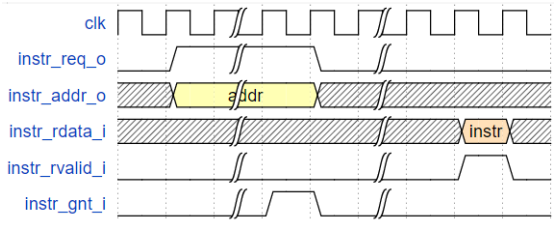



Figure 3.3: Timing diagram Instruction memory/cache communication protocol

### 3.2.2 Data memory interface

The core takes also care of accessing the data memory. Load and stores on words (32 bits), half words (16 bits) and bytes (8 bits) are supported.
The RI5CY core is able to perform misaligned accesses, so accesses that are not aligned on natural word boundaries. However, if a misaligned access is really needed, the core will need to perform two separate word-aligned accesses. This means that at least two cycles are needed for misaligned loads and stores.
The core communicates with the memory according to the same protocol used for the instructions. Anyway, the data interface can execute both reads and writes, so more signals are needed for the handshake. Details below:

- **Read from memory**. The protocol requires that the core notifies the memory when it wants to access, providing a valid address in *data_addr_o* and setting *data_req_o* high at the same time. The memory then answers with a *data_gnt_i* set high as

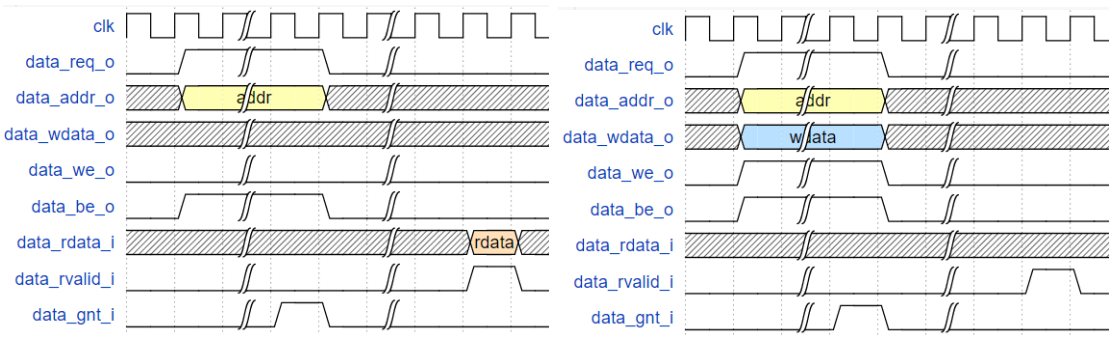
soon as it is ready to serve the request. This may happen in the same cycle as the request was sent or any number of cycles later. After a grant is received, the core may change the address in the next cycle. After receiving a grant, the memory answers with a *data_rvalid_i* set high if *data_rdata_i* is valid. This may happen one or more cycles after the grant has been received. The read access in the data memory is then exactly the same as the instruction memory.

- **Write in memory**. The protocol is very similar also for a write. In fact, the core always requires a memory access with *data_addr_o* and *data_req_o*, but at this time it provides also the signals *data_wdata_o*, *data_we_o* and *data_be_o*. The memory answers with a *data_gnt_i* set to high, whenever it is ready to process the request. After a grant is received, the address, data to be written and other control signals may be changed in the next cycle because it is assumed that the memory has already processed and stored that information. Note that the memory must answer with *data_rvalid_i* set even when a write is performed, although the *data_rdata_i* has no meaning in this case.

Table 3.2 and Figure 3.4 summarise this protocol:

| Signal | Direction | Description |
|---|---|---|
| data_req_o | output | Request signal, must stay high until *data_gnt_i* is high for one cycle. |
| data_gnt_i | input | The signal is high when the other side accepted the request: *data_addr_o* may change in the next cycle. |
| data_addr_o[31:0] | output | Address to read the data: sampled by other side when *data_gnt_i* and *data_req_o* are high. |
| data_we_o | output | Write Enable, high for writes, low for reads, sent together with *data_req_o*. |
| data_be_o[3:0] | output | Byte Enable. Is set for the bytes to write/read, sent together with *data_req_o*. |
| data_wdata_o[31:0] | output | Data to be written to memory, sent together with *data_req_o*. |
| data_rdata_i[31:0] | input | Data read from memory: sampled by the core when *data_rvalid_i* is high. |
| data_rvalid_i | input | *data_rdata_i* is considered valid when *data_rvalid_i* is high. This signal behaves like a pulse. |

Table 3.2: Data memory/cache communication protocol




(a) Read transaction

(b) Write transaction

Figure 3.4: Timing diagram Data memory/cache communication protocol

# 4 Architecture description

As already mentioned, this RI5CY implementation, is a 4-stage pipeline in-order 32-bit RISC-V processor core. In this section the microprocessor architecture is described with more details.
Anyway, before entering into the details of the RTL blocks, it is worth to analyse how the pipeline works in RI5CY. A representation of the pipeline is showed in Figure 3.5. The pipeline stages are:

- Instruction Fetch stage (IF stage)
- Instruction Decode stage (ID stage)
- Execution stage (EX stage)
- Load and Store stage (LSU stage)

Each pipeline stage has two control inputs: an *enable* and a *clear*: the *enable* activates the pipeline stage and the core moves forward by one instruction, while the *clear* removes the instruction from the pipeline stage as it is completed and there is not a new instruction to process.
Every pipeline stage is cleared if the *ready* coming from the stage to the right is high, and the *valid* signal of the same stage is low. If the *valid* signal is high, the stage is enabled.
Every pipeline stage is independent from its left neighbour, meaning that it can finish its execution no matter if a stage to its left is currently stalled or not. On the other hand, an instruction can only propagate to the next stage if the stage to its right is ready to receive a new instruction. This means that in order to process an instruction in a stage, its own stage needs to be ready (*valid* signal high) and so does its right neighbour (*ready* signal high).

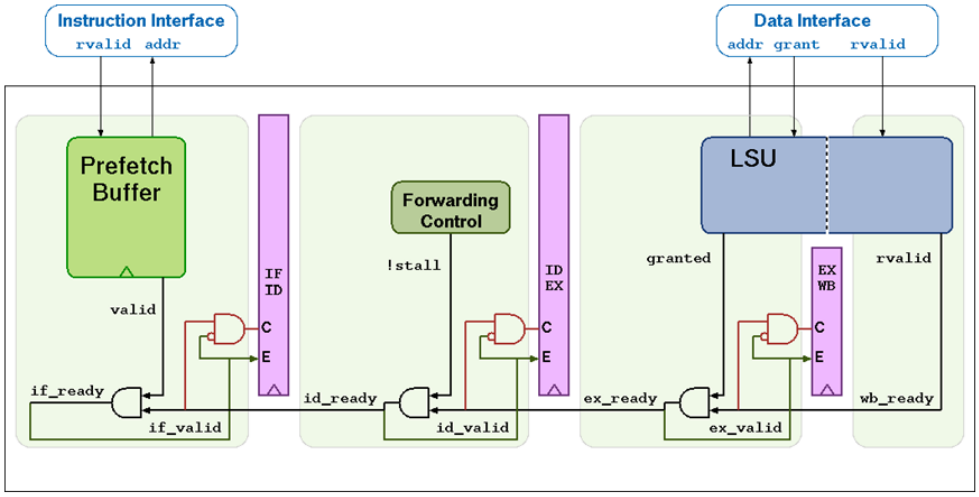



Figure 3.5: RI5CY pipeline

In the following sections, each stage of the pipeline is fully expanded. The description will not cover all the details of the architecture, but it is intended to provide a general background of the RI5CY core.

## 4.1 Instruction Fetch stage

This first stage allows to read instructions from memory and do some preliminary evaluations on the instruction read. The main components are described below, while a repre-

sentation of the full stage is available in the Figure 3.6.

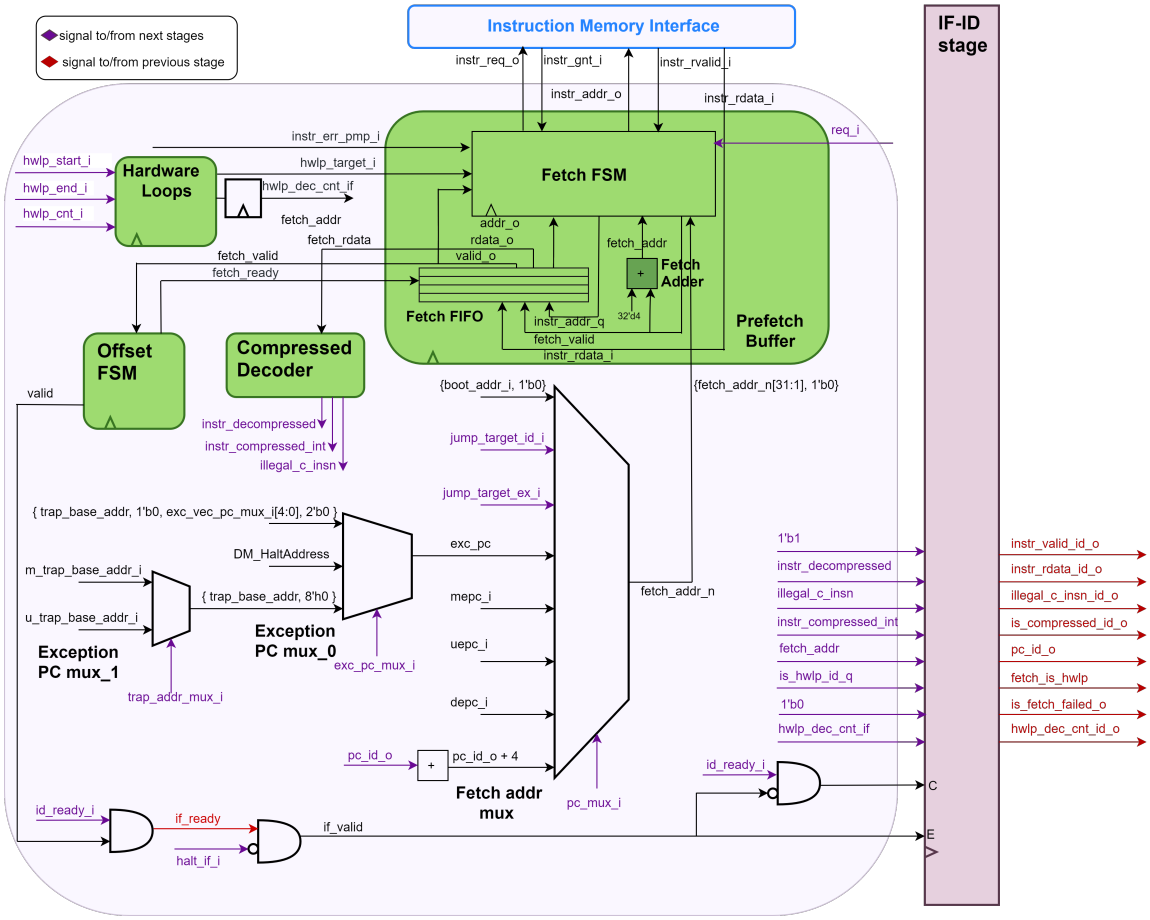



Figure 3.6: IF stage block diagram

### 4.1.1 PC multiplexers

Three multiplexers are used to select the address of the instruction to fecth from the memory. The control signals used to make the selection come from the ID stage:

- **Fetch_addr_mux**: this multiplexer selects the 32 bits address signal *fetch_addr_n* from a set of available addresses like the immediate next Program Counter (PC), the boot address, the address related to an exception handler, the address of the of the PC restored after returning an exception or the address of the target address of a jump instruction.
- **Exception_PC_mux_0**: this multiplexer selects the exception address between an interrupt address (from the ID stage), a trap base address or *DM_HaltAddress* (that is a constant value).
- **Exception_PC_mux_1**: this multiplexer selects the trap handler base address between the user trap base address and the machine trap base address. Both base addresses are computed by the CSRs block, not showed here because out of the scope of this work.

### 4.1.2 Prefetch Buffer

A prefetch buffer is the module that directly communicates with the instruction memory, or instruction cache. It helps to cut overly long critical paths to the instruction memory. There are two prefetch buffer flavors available, according to the width of the instructions to read:

1. **32-Bit word prefetcher**. It stores the fetched words in a FIFO with three entries (instruction address, read data and valid bit). Generally, the basic functionality of the prefetcher consists in computing the address to send over the memory and then storing this address into a FIFO for future usage.

   - **Fetch FIFO**. The logic outside the FIFO computes the *instr_addr_q*, and the *in_valid_i*, while the *instr_rdata* comes directly from the memory. Those three data are written into the fifo according to the valid-ready protocol. In particular, the input interface of the fifo always accepts data if there is space inside the fifo. In fact, its *in_ready_o* goes low only if the fifo is full.
     The output interface instead, has *out_valid_o*\|*fetch_valid* always high if the fifo has data to be processed. If the interface outside the fifo is ready, the data is available to the output of the fifo immediately, then within the same clock cycle. The fifo outputs are then *fetch_addr*, *fetch_rdata* and *fetch_valid* signal.
     Moreover, the fetch fifo has logic inside to manipulate the address in case of compressed instructions and hardware loops instructions. As a matter of fact, it is expected that the instruction memory is always accessed with aligned addresses to read 4 bytes, that can correspond to a normal instruction (32 bits) or to at least one compressed instruction (16 bits). According to the RISCV ISA, all the 32 bits instructions are coded with the first two bits equal to 2'b11, while the 16 bits instructions do not show this equality. The logic inside the fifo evaluates these two bits to distinguish between the two cases. In case of an hardware loop instruction, the input *in_is_hwlp_i* is used to detect it.

   - **Fetch adder**. Defining as current instruction address *instr_addr_q*, that is the input of the fifo, the Fetch adder computes the next possible instruction address (*fetch_addr*) by increasing it by 4.

   - **Fetch FSM**. The finite state machine inside the the prefetcher keeps track on the state to perform the communication with the instruction memory according to the protocol described in Section 3.2.1. As a matter of fact, this component can be in one of the following states:

     – IDLE, it is the state that the FSM assumes after the reset signal. In this state if no request to memory is done, the input valid of the fifo is 0 and the input of the fifo *instr_addr_q* is equal to 0 as well, because corresponds to the reset value. The *instr_addr_q* is the output of a register, whose enable is the signal *addr_valid*, set to 0 in this case. If there is no need to rise a request for an instruction to the memory, the output *instr_addr_o* can assume any values.
     Otherwise, if a request is raised from the ID stage with the signal *req_i*, the request *instr_req_o* is sent to the memory with a valid value in *instr_addr_o*. The *instr_addr_o* signal can assume the value of the *fetch_addr* signal in a normal case (previous instruction address value incremented by 4), the value of the signal *fetch_addr_n* computed outside the prefetcher in case of a branch/jump or the value of the signal *hwloop_target_i*. Moreover, in case of a request the register *instr_addr_q* is enabled as well.

Since a valid address is available and sent to the memory, this address can be recorded by the register in the next cycle.
A state change occurs only if the request to the memory is sent. In fact, in the next clock cycle, the state will be WAIT_RVALID if the signal *instr_gnt_i* from memory is immediately set to 1, that means that the memory has read the request of an instruction from the microprocessor. If this signal is not granted the state changes in WAIT_GNT.

- WAIT_GNT, is the state in which the IF stage waits for the grant from the instruction memory. As a matter of fact, according to the protocol the request signal *instr_req_o* is kept high and the *instr_addr_o* is still stable, because the memory did not accept the request yet.
  The state will change in the next cycle only if the *instr_gnt_i* is received, into the WAIT_RVALID state.

- WAIT_RVALID, is the state in which the processor waits for the instruction content from the memory. In fact, as soon as the *instr_rvalid_i* is set, the fifo is enabled so that it can read the *instr_rdata_i* (containing a valid value) and the *instr_addr_q* (that contains the corresponding memory address).
  In this cycle another request can be raised again from the ID stage. If the *instr_rvalid_i* is still not received when the new request is set by the core, it means that the previous request is aborted so next state is WAIT_ABORTED. Otherwise, in case the request signal is concurrent to the *instr_rvalid_i*, the previous request is still valid, so fifo will be enabled and the FSM can start processing the new request starting from the next cycle: according to the grant signal value the next state will be WAIT_GNT or WAIT_RVALID.
  In case of no concurrent request and valid signals received the next cycle the state will be IDLE.

- WAIT_ABORTED, state in which the FSM goes if in the WAIT_RVALID state a new *req_i* is received before *instr_rvalid_i*. This means that the previous request has been aborted, and the new one needs to be processed. Nevertheless, it is expected that the memory will reply with a valid signal for the old request, in fact as soon as this happens, the read instruction is not registered into the fifo and the FSM can process the response for the valid request in the usual way. In other words, this state is needed to manage two overlapping responses from memory.

- WAIT_JUMP, this state is reachable from every state if the signal *instr_err_pmp* is raised by the PUMP UNIT. So, this state allows to wait for the *branch_req* signal, and then to start the processing of a instruction that follows a jump/branch instruction.

2. **128-Bit cache line prefetcher**. The available hardware allows to store one 128-bit wide cache line plus 32-bit for cross-cache line misaligned instructions. Anyway, a detailed description is not needed because this version has not been used for this research work.

### 4.1.3 Offset FSM

A finite state machine in the IF stage is used to manage the pipeline stage. Two possible states are possible here:

- IDLE, state in which there is no request for a new instruction. As a matter of fact, the *fetch_ ready* signal to be sent at the output of the fifo is set to zero, because no instruction will be read, and the *valid* signal of the IF stage is set to zero as well. In case the request signal *req_ i* is set from the ID stage, the state will change into WAIT.
- WAIT, in this state, if the fifo in the prefetcher contains an instruction and so the signal *fetch_ valid* is high, then the *valid* signal is set to high too. In this case, if the ID stage is ready, the instruction from the fifo is read by setting the signal *fetch_ ready*.

### 4.1.4 Hardware Loops

The hardware loop controller unit is responsible for handling hardware loops. This can be split into two sub-tasks: first to compare PC to all stored end addresses and then to jump to the right start address if counter is equal to 0.
The ID stage contains the hardware loop registers, whose values is given to IF stage with the signals: *hwlp_ start_ i*, *hwlp_ end_ i* and *hwlp_ cnt_ i*.
This module compares through comparators the current fetched address with the value of the hardware loop end addresses. There are as many comparators as the the hardware loops supported. If the fetched address has the value of one of the end addresses, the value of the corresponding hardware loop counter is checked. The signal *hwlp_ dec_ cnt_ if* will go high at the next cycle to decrease the value of the counter. As soon as the end the loop is reached, the *hwlp_ jump_ o* is set to 1.

### 4.1.5 Compressed Decoder

The Compressed Decoder is a fully combinational module, that converts the RISC-V compressed instructions (RV32C) into their extended version (RV32I). The *instr_ decompressed* is passed to the next stage with the *instr_ compressed_ int* that is a flag that indicates that the instruction read has been decompressed.

## 4.2 Decode stage

This stage decodes the instructions and hosts the Register File. The main components are described below, while a representation of the full stage is available in the Figure 3.7.

### 4.2.1 Decoder

The ID stage receives as input from the the previous stage the instruction read from the memory. The instruction needs to be interpreted in order to proceed with the specific actions described by the instruction itself.
The decoder is a combinational logic block that is used for this purpose. This block sets up the processor control lines as required by the current instruction *instr_ rdata_ i*. In particular, only the fields related to the opcode and more general control bits are evaluated. It is not worth to describe all the control lines managed by the decoder, but it may be useful to have an idea on the typologies of control signals that are used into the RI5CY:

- *Operands used*. Control signals that notify when an operand between *Operand_ a*, *Operand_ b* and *Operand_ c* is being used. This will enable the forwarding logic to check the dependency of the current operands from the result of previous non-completed instructions.

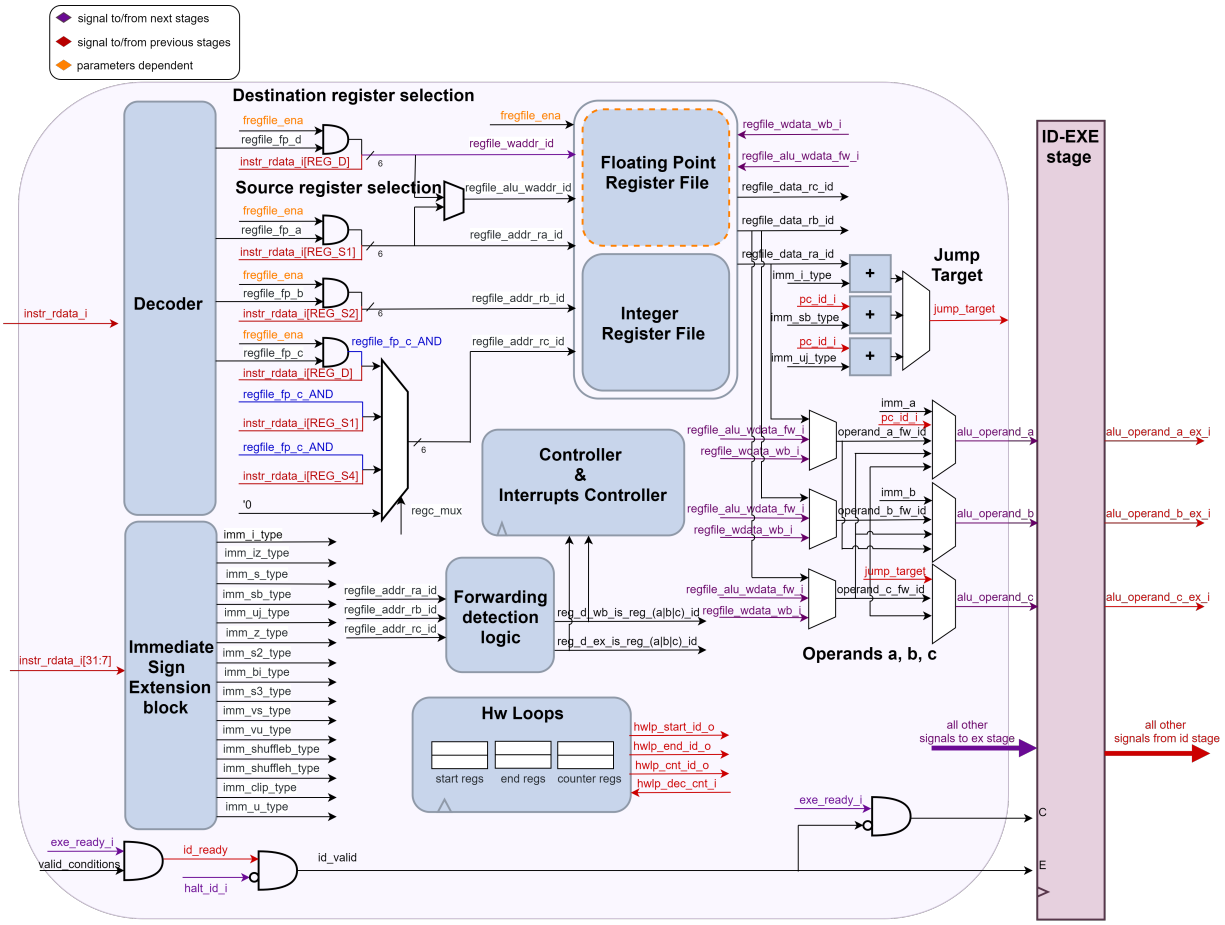



Figure 3.7: ID stage block diagram

- *Floating-Point Register File used.* In order to distinguish an integer operation from a floating-point operation, the decoder sets some control signals in order to select integer or floating-point operands.
- *ALU/MUL.* ALU operands selection, that can be registers, immediate values, PC or a jump target. The ALU operation is also determined by the decoder. The Multiplication Unit needs also an operation type to be selected and other control signal, for example signals that select the operand types (integer or floating-point).
- *CSR manipulation.* It includes signals that enable the access to the Control and Status Registers and select the operation type to perform on this registers.
- *Load and Store Unit.* Load and Store instructions determine the value of signals that will kick off the communication with the memory once the instruction is the next stage.
- *Hardware Loops.* Signals that include the enable for the hardware loops registers and control signals that select the target values to assign to them.
- *Jumps.* Signals to perform a jump during the execution, in order to select a given Program Counter (PC).

### 4.2.2 Immediate sign-extension block

The Immediate sign-extension block is a combinational module that prepares the immediate field to fit 32 bits, according to all the supported instructions that use an immediate.

### 4.2.3 Source and Destination registers selection

An operation in the RISC-V can require up to 3 operators. The operators can be read from the register files, that need to be addressed. In fact, in there is a section in the decode stage that manages the address signals for two register files: an Integer Register File and a Floating-point Register File. Both register files contain 32 registers, but they are wrapped together so the address signals width is 6 bits.
In order to distinguish a floating point register address from a integer register address, each address has the most significant bit (MSB) equal to 1 for a FP operation and equal to 0 for an integer one.
There are four address signals, three for source registers selection and two for destination registers selection:

- *regfile_addr_ra_id*, *regfile_addr_rb_id*, *regfile_addr_rc_id*: source registers address.
- *regfile_waddr_id*, *regfile_alu_waddr_id*: destination registers address.

For each source\destination address, the MSB is determined by the *regfile_fp_a*, *regfile_fp_b*, *regfile_fp_c*, *regfile_fp_d* signals coming from the decoder unit, that are set when a floating-point operation is decoded.

### 4.2.4 Forwarding detection logic

Forwarding is an optimisation in pipelined CPUs to reduce the pipeline stalls due data dependencies. In particular, a stall can occur when the current operation has to wait for the results of an earlier operation which has not yet finished.
The RI5CY microprocessor offers the forwarding mechanism, that consists in simple combinational logic that compares the write address of the previous instructions (that are being processed in a later stage: Execution or Load-Store), with the three source registers address of the current instruction in the ID stage. The outputs of the forwarding logic are control signals that will notify if there is a dependency with the result of a previous instruction, that is still not written in the Register Files.

### 4.2.5 Controller

The Controller is the logic unit that coordinates all the operations during the instructions execution. With respect to the decoder, the control signals that this unit produces, come from an evaluation of external requests (e.g. interrupts) or general pipeline status.
The basic functionalities of this unit are summarised below:

- *FSM Core Controller.* The key component of the Controller is the Finite-State Machine (FSM). The FSM is used whenever there is the need to take track of some subsequent events. In fact, the FSM in the Controller is used to flush the pipeline in case of branches taken, to manage the core at the boot time, by setting the initial memory address to fetch, to take action after an interrupt reception and many other operations that will be not covered into details.
- *Forwarding Control Logic.* Forwarding logic is finalised into the controller, because the decoding of the Forwarding detection logic is not enough to guarantee that the potential dependency is a real one. As a matter of fact, the controller needs to check if the involved write back registers are really being update or not. As a result, the controller drives the signal *operand_a_fw_mux_sel_o*, *operand_b_fw_mux_sel_o* and *operand_c_fw_mux_sel_o* that will be used for the operands selection.

- *Stall Control.* Nevertheless the forwarding mechanism is in place, there are cases in which is unavoidable to stall the pipeline. For example, the result that comes from a load instruction cannot be retrieved before the load instruction is actually complete, so the pipeline might be stalled if the next instruction needs the load result. Same applies for the jump with link instructions or in case of misaligned access in memory.

### 4.2.6 Hardware loop

This block is the place where the hardware loops registers live. Whenever an hardware loop instruction is decoded, the hardware loop registers are initialised. The hardware loop logic in RI5CY allows the management of two nested hardware loops. Once the registers are enabled, the counter register is decremented every time the signal *hwlp_dec_cnt_i[1:0]* is received from the IF stage. In fact, when this signal is set it means that the corresponding hardware loop has completed one loop, then the program counter matches the end address. The registers values *hwlp_start_id_o[1:0]*, *hwlp_end_id_o[1:0]* and *hwlp_cnt_id_o[1:0]* are sent to the IF stage in order to perform the evaluation on the fecthed address.

### 4.2.7 Jump Target

Whenever a jump instruction is encountered, the RI5CY architecture allows to immediately compute the target address, without waiting for the instruction to reach the EX stage. The target address *jump_target_o* will be sent to the ID stage to correctly update the Program Counter in the next cycle. Its value is different according to the type of jump decoded.

### 4.2.8 Operand a, Operand b, Operand c

The computational unit in the EX stage can perform an operation with up to three operands. These operands can assume many different values according to the need of the instruction being executed:

- *alu_operand_a.* This operand can assume the value of an immediate for a certain set of instructions. As a matter of fact, *imm_a* corresponds only to a subset of the possible immediate values. Moreover, this operand allows to select the Program Counter (PC) to perform the operation in case of jump instructions. Lastly, the operand can also assume the value of one of the registers read through the ports *regfile_data_ra_id*, *regfile_data_rb_id* or *regfile_data_rc_id* of the Register File. If the value of these registers is expected to change due to previous instructions, the forwarding logic will select the expected value of the register instead of the current one.

- *alu_operand_b.* The immediate values *imm_b* correspond to a different subset of immediate values. Same as before, this operand can assume the value of Register File ports *regfile_data_ra_id*, *regfile_data_rb_id* or *regfile_data_rc_id* and the forwarding logic will care to give to the operand the expected value in case of conflicts with previous instructions.

- *alu_operand_c.* This operand cannot assume any immediate values. It can instead select the *jump_target* value and a the content between only two ports of the Register File: *regfile_data_rb_id* or *regfile_data_rc_id.*

### 4.2.9 Register Files

The role of the Register File in a microprocessor is to store some useful and most used variables to perform the computation. RI5CY Register File contains:

- *Integer Register File.* It consists in 32 registers each of 32 bits used to store integer numbers ($x0 - x31$). The register $x0$ (the first one) is not writable, and contains the value 0.
- *Floating-Point Register File.* It also consists in 32 registers each of 32 bits used to store floating-point numbers ($f0 - f31$).

The two register arrays are accessible through 3 read ports and 2 write ports.

- Read port a: *regfile_ data_ ra_ id*, *regfile_ addr_ ra_ id*.
- Read port b: *regfile_ data_ rb_ id*, *regfile_ addr_ rb_ id*.
- Read port c: *regfile_ data_ rc_ id*, *regfile_ addr_ rc_ id*.
- Write port a: *regfile_ wdata_ wb_ i*, *regfile_ waddr_ wb_ i*.
- Write port b: *regfile_ alu_ wdata_ fw_ i*, *regfile_ alu_ waddr_ fw_ i*.

The listed ports are shared between the two Register Files. In fact, accesses to the Integer Register File and the Floating-Point Register Files are differentiated through the address ports, that shows the MSB equal to 0 in case of an Integer Register File access, while 1 in case of a Floating-Point Register File access.
Moreover, the Floating-Point Register File can be disabled through the parameter "FPU". In fact, this will save power consumption in contexts in which the floating-point ISA extension is not needed.

## 4.3 Execution stage

The execution stage is the place where the actual computations are performed between the operands. Figure 3.8 shows an high-level representation of the full stage.

### 4.3.1 Arithmetic Logic Unit

The Arithmetic Logic Unit is the computational unit that manages various type of operations with a maximum of three operands *alu_ operand_ a*, *alu_ operand_ b* and *alu_ operand_ c*:

- *Bit manipulation.* ALU can work on single bits or groups of bits within a word, performing negation, shifts, bit-counting operations.
- *Arithmetic operations.* General operations such as addition, subtraction, comparisons and divisions.
- *Floating-Point classification.*

All these operations can be performed within one clock cycle except for the division and reminder that take between 2 and 32 cycles. The number of cycles depends on the operand values.

### 4.3.2 Multiplier

The Multiplier module performs:

- *Integer Multiplications.* Multiplication generally requires only one clock cycle for 32-bit results. The multiplications with upper-word result of 32-bit x 32-bit multiplication, take 4 cycles to compute.
- *Multiply-Accumulate operations.* Those operations are multi-cycle.

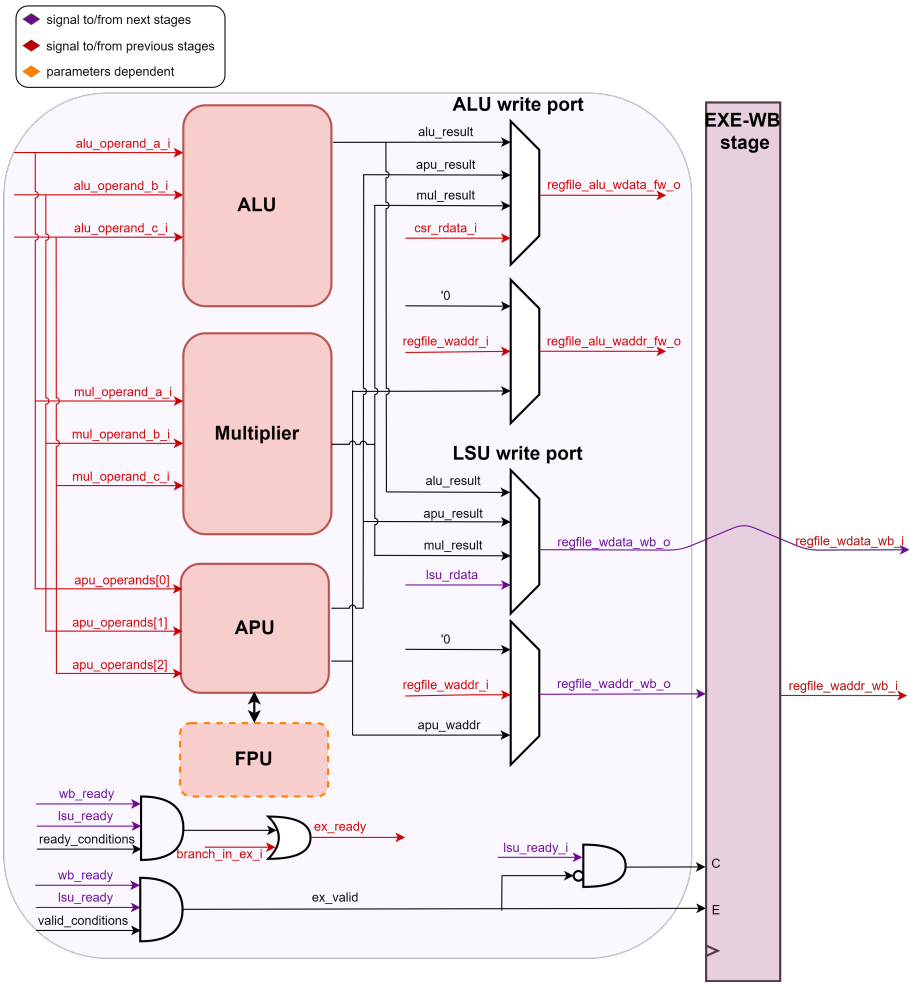



Figure 3.8: EX stage block diagram

### 4.3.3 APU and Floating point unit

The available version of the RISC-V does not contain a FPU. Anyway, the structure allows to easily extend the core with a private FPU, which is capable of performing all RISC-V floating-point operations that are defined in the RV32F ISA extension. The FP extensions can be enabled by setting the parameter of the top-level module to 1. To access to the extended FPU unit, the Auxiliary Processing Unit (APU) is used. In fact, the APU has a standard valid-ready interface to be connected to an external processing unit, such as the FPU. Any processing units that have a compatible interface can be connected to the APU and therefore can be used for introducing new operations in the RI5CY core.

### 4.3.4 ALU write port

The ALU write port allows to write the result of the execution stage directly into the Register File to accommodate the forwarding mechanism. The result is selected between the units presented above or from one control-status register: *alu_result*, *mul_result*, *apu_result* or *csr_rdata*.

### 4.3.5 LSU write port

The LSU write port is used instead for results of the execution stage that takes two clock cycles (*alu_result*, *mul_result* or *apu_result*) or for writing back the result read from the

memory. This port is placed into the Execution stage module, but it actually belongs to the LSU stage in terms of timing.

## 4.4 Load and Store Unit stage - Write Back stage

The Load and Store unit (LSU) or Write Back (WB) stage manages the communication with the data memory. For timing purposes the memory communication starts during the Execution stage and it is completed during the actual Load and Store stage. In fact, Figure 3.9 shows that some Load and Store Logic actually performs during the previous stage even if it is inserted into this stage.

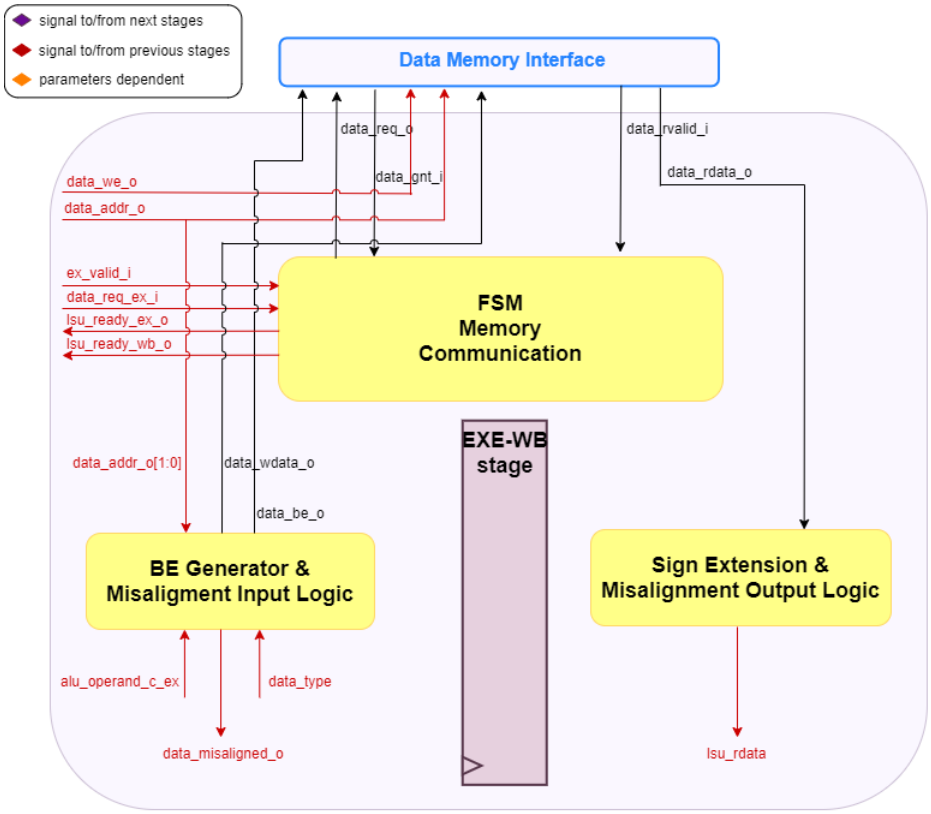



Figure 3.9: LSU stage block diagram

### 4.4.1 FSM Memory Communication

The FSM in the LSU stage takes care of the communication with the Data Memory, sending and evaluating the control signals according to the communication protocol already described in 3.2.2. The states the FSM can assume are described below:

- IDLE. This state indicates that in the previous cycle there was not any active transactions with the memory. The FSM in this state evaluates the request signal *data_req_ex_i* coming from the ID stage. If the signal is set, it means that a Load/Store instruction is being processed in the Execution stage. The request output to the memory *data_req_o* is raised at the same time. Moreover, the protocol allows the memory to send the grant signal *data_gnt_i* immediately, therefore the grant signal is evaluated in this state too. The next state can be WAIT_RVALID, in case the signal or WAIT_RVALID_EX_STALL in case of signal equal to 0. This signal indicates that the Execution stage is going to be stalled.

- WAIT_RVALID. In this state, the FSM waits for the rvalid signal in the LSU stage. If the *data_rvalid_i* is received from the Memory, the transaction can be considered finished. In fact, in the same cycle, the Execution stage might process another Load/-Store instruction, so another transaction can be started setting the signal *data_req_o* and after that the reception of the *data_gnt_i* is immediately performed. It means that the next state can be WAIT_RVALID or WAIT_RVALID_EX_STALL according to the same conditions stated before. On the contrary, if the Execution stage is processing a different instruction, the *data_req_o* will be 0 and the next state will be IDLE again.
- WAIT_RVALID_EX_STALL. The FSM assumes this state when the Execution stage has been stalled for some reasons and the Load/Store instruction waits for the rvalid signal while is in the Execution stage. In this state it is not possible to initiate a new memory request. As soon as *data_rvalid_i* is received, if the stall is deasserted (*ex_valid_i* equal to 1) the transaction is completed and the next state will be IDLE, otherwise the next state will be IDLE_EX_STALL.
- IDLE_EX_STALL. This state is needed only to wait for stall deassertion. As soon as *ex_valid_i* is equal to 1, the FSM state will change into IDLE again.

### 4.4.2 Bytes-Enable Generator & Misalignment Input Logic

The Memory is organised in bytes and it is supposed to accept only aligned accesses, so within the 4-bytes address ranges. The memory will evaluate only the offset address *{data_addr_o[31:2],2'b00}*, so the microprocessor needs to communicate which set of bytes are actually accessed using another signal *data_be_o*. This is a 4-bits signal, each bit corresponds to a byte position starting from the given memory offset address:

- *data_be_o[0]* corresponds to the first byte with offset address *{data_addr_o[31:2],2'b00}*.
- *data_be_o[1]* corresponds to the second byte with offset address *{data_addr_o[31:2],2'b00}*.
- *data_be_o[2]* corresponds to the third byte with offset address *{data_addr_o[31:2],2'b00}*.
- *data_be_o[3]* corresponds to the fourth byte with offset address *{data_addr_o[31:2],2'b00}*.

Those bits can be set at the same time in order to access to many bytes at the same time (e.g. 4'1111b corresponds to a word memory access). This address manipulation is done to process misalignment accesses. In fact, the RI5CY core splits the misaligned access into aligned accesses.
In case of stores, the write data should be adapted to have meaningful data corresponding to the position indicated by the bits set in *data_be_o*.

### 4.4.3 Data Sign Extension & Misalignment Output Logic

Data read after a load operation is manipulated in order to perform a sign-extension in case of bytes and half-words and in case of misaligned accesses. The result is then written into the Register File through the LSU write port.

## 4.5 Peripherals and Memory model

For the scope of this thesis, it is worth to briefly describe the memory model adopted by the RI5CY core.
Figure 3.10 represents the structure available for the simulation of the core. As a matter of fact, the RI5CY testbench wraps together the RI5CY microprocessor with a memory model. This model consists of:

- The actual dual-port ram with the logic that implements the communication protocol.
- Non-synthesisable structures that belong to the testbench. Those are used to emulate other peripherals and for simulation specific requirements.

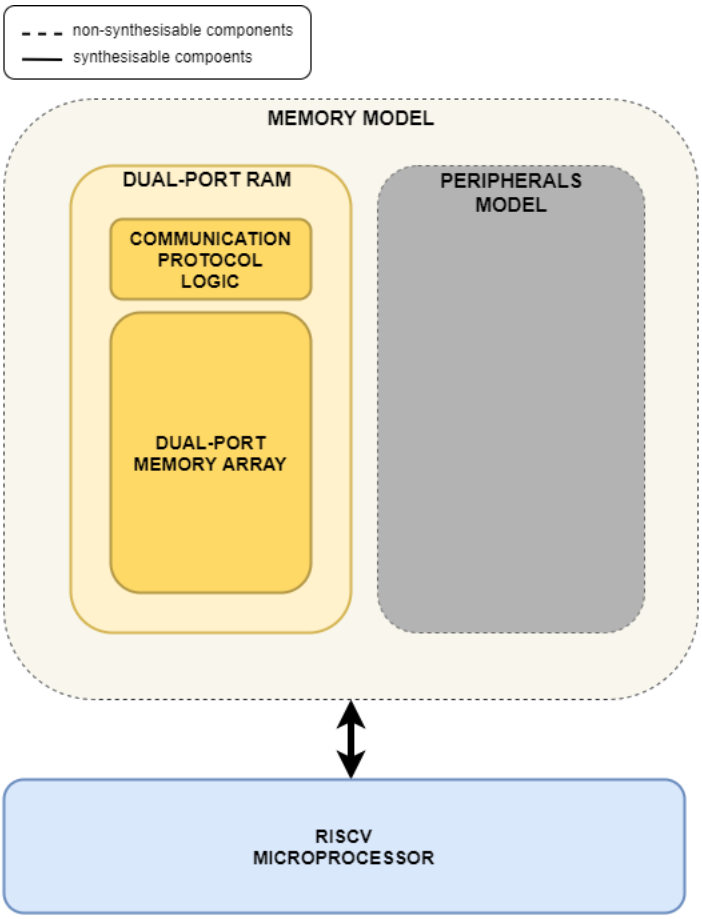



Figure 3.10: Peripherals and Memory model organisation

# Chapter 4

# Logic-in-Memory in RI5CY Framework

## 1 Logic-in-Memory State of Art

To overcome the communication speed issue between CPU and Memory, research institutes and industry are moving towards to Processing-in-Memory architectures.
The goal of the Processing-in-Memory concept is to reduce load and store operations in memory by distributing the computational part in the memory to execute some basic operations. The classic data movement proposed by the Von-Neumann architecture is revised, so that data will be not moved back and forward into the memory.
As a matter of fact, memory will not only be the storage center of a computing systems, but will offer the possibility to manipulate the data stored bypassing the CPU. With this new system architecture, the CPU will not have the need to read the data, compute the operation and then store the data back in the memory, because the data processing will be performed in the memory itself. Therefore, the CPU is left to only coordinate the operation.

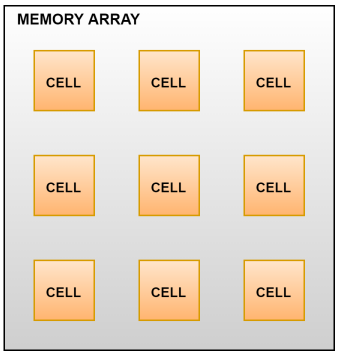
The advantage in terms of speed is immediate, reason why this new approach is very promising for the future and the scientific community is pushing towards this direction.
Literature offers a wide set of Processing-in-Memory (PiM) implementations and definitions. In particular, [7] classifies the four main typologies of PiM according to the role that memory has for the computation:

- *Computation-near-Memory (CnM).* According to this definition, memory and computational logic are kept separated. Anyway, thanks to the new 3D-integration technology, those two entities can be really close, so the length of the interconnections is extremely reduced. An example of this typology is WIDE-IO2, a 3D stacked DRAM memory [8] that has a logical layer placed at the bottom of the stack.
- *Computation-with-Memory (CwM).* Under this definition, are the memories that store pre-computed results. Often, a combination of Look Up Table (LUT) and Content Address Memory (CAM) is used. In fact, the LUT indicates the truth table of a certain operation, while the CAM stores the results. The "computation" is then performed into two steps. Firstly, the inputs are used to access the LUT, that in turn access to the CAM, retrieving an address. The second step consists in using the obtained address to read the result stored in the CAM.
- *Computation-in-Memory (CiM).* This typology does not change the memory array. Data computation is instead performed in the peripheral circuitry. For example,

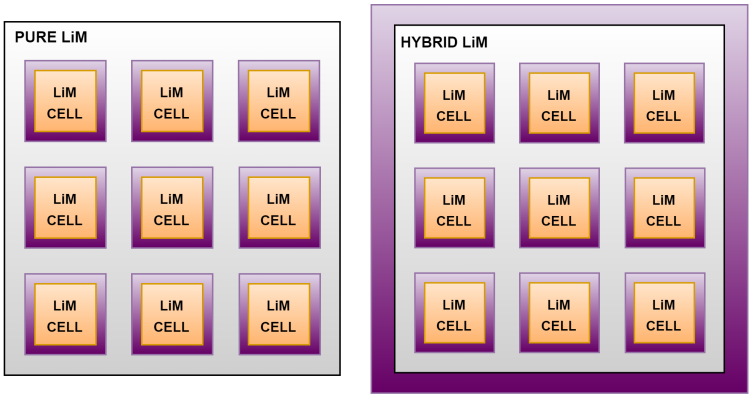
sense amplifiers (SAs) can be slightly changed to perform simple bitwise operations, or particular decoders are also adopted to perform operations between many memory locations.
Configurable Logic-in-Memory Architecture (CLiMA) [9] is an heterogeneous architecture composed of an in-memory (LiM and/or CiM) computing unit that offers much flexibility.

- *Logic-in-Memory (LiM).* Data computation is performed directly inside the memory array, by adding some logic in each memory cell.
  Example of LiM is [10], an hardware implementation of a the Binary Neural Network, in particular of the XNOR-Net model, that exploits the usage of XNOR gates.

For the scope of this thesis, it is worth to focus on the Logic-In-Memory. Figure 4.1 shows a further classification of the Logic-in-Memory, at a very high-level:




(a) Memory array




(b) Pure Logic-In-Memory (c) Hybrid Logic-In-Memory

Figure 4.1: Logic-in-Memory typologies

- *Pure Logic-in-Memory.* This definition sticks to the LiM definition given previously. The introduction of new gates within the single memory cell is a very simple and cost-effective LiM solution. According to the technology used to implement this kind of memory, very easy bitwise operations can be obtained almost for free from the array. The new cell will compute some combinational operations in addition to the classic storage functionality. (See part *b*).
- *Hybrid Logic-in-Memory.* A more broaden definition of the LiM can include some additional control logic around the memory array. The logic within each memory cell will still be present, but in order to execute more complex functions on the data stored, some control logic can be added around the array. This solution guarantees

much more flexibility and will allow to perform more elaborated operations and algorithms. This solution can still be classified as LiM and not as CiM. In fact, the memory cells still requires the additional gates, while the logic around the memory is only needed to coordinate the gates within the memory array. (See part *c*).

The Logic-in-Memory considered for this thesis, belongs to the second typology. Of course, this is a more elaborated and expensive solution, but it allows to consider a various number of operations and algorithms that the memory could do, without the need of the CPU.
The number and the types of operations that the LiM would perform depend on the actual implementation and from the needs of the system in which the memory will be integrated. Those operations can vary from bitwise operations between memory cells to more complex algorithms.

Logic-in-Memory opens a new world for the data processing design and it is clearly a desirable feature that must be explored. Nevertheless, it is important to consider that the LiM will introduce an additional level of complexity and it could have an impact in terms of area (definitely bigger than a normal memory), power and timing. All these considerations should be performed on a case by case basis, because depend on the actual implementation.

## 2 Logic-in-Memory architecture

The RI5CY core supposes to have only one memory for both instructions and data. As a matter of fact, the instructions and data parts of the memory, not only share the same physical memory but there is not any specific address that divides the two parts. The management of the division between them is left to the compiler according to the size of the program that will be run.
Therefore, the available memory model is a dual port memory, such that fetch and load-store operations can occur in the same clock cycle and two decoders corresponding to the two ports, give the possibility to address all memory locations.

The implemented Logic-in-Memory adds some logic around the memory array and within the memory array itself. The result is a memory capable of some basic bitwise operations and of few simple algorithms.
In more details the proposed memory architecture can perform the following operations:

- *Normal load and store operations*: the new memory can still behave as a memory, so it will perform a read or a write in any memory locations in just 1 clock cycle. Data port of the memory supports load and store for 8-bit, 16-bit or 32-bit data.
- *Bitwise operations*: Only the data port of the memory can enable the logic for this kind of operations. Load and store can be performed together with a bitwise operation, using an input mask, in just 1 clock cycle. The available bitwise operations are AND, OR and XOR. Only in case of a store, the bitwise operation is supported also on a range of memory locations, assuming that the input mask to use is the same for all the locations selected.
  Bitwise operations are supported only on 32-bit data, on aligned addresses.
- *Maximum and minimum*: a special load operation can give the maximum or minimum value on certain range of memory locations. The duration of this memory operation is 33 clock cycles for any range selected.
  Maximum and minimum are computed considering 32-bit data values.

Figure 4.2 shows the high-level Logic-in-Memory structure. As mentioned before, due to the non-static distinction between instructions and data memory, the logic introduced

for the above listed operations, is distributed over all the memory locations, even if is effectively used with data and not with instructions.

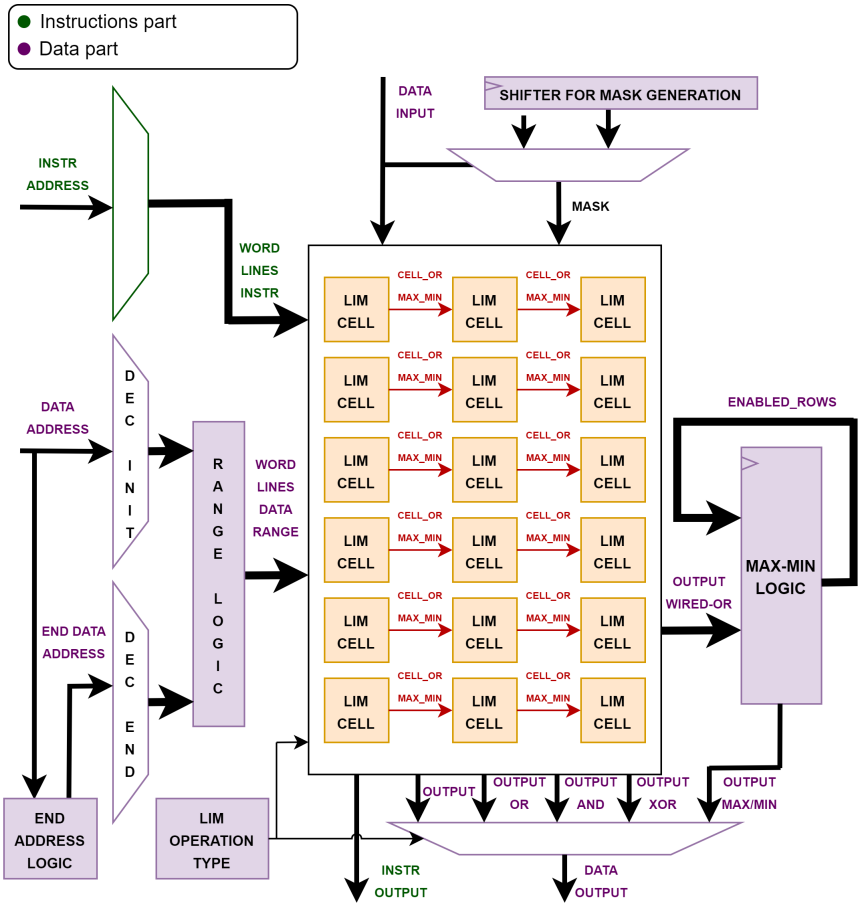



Figure 4.2: Dual port Logic-in-Memory high level architecture

## 2.1 Bitwise operations - Logic-in-Memory cell

The heart of the bitwise operations is the memory bit-cell as shown in Figure 4.3. The memory cell has been enlarged in order to compute the bitwise operation between the content of the cell and an input mask bit.

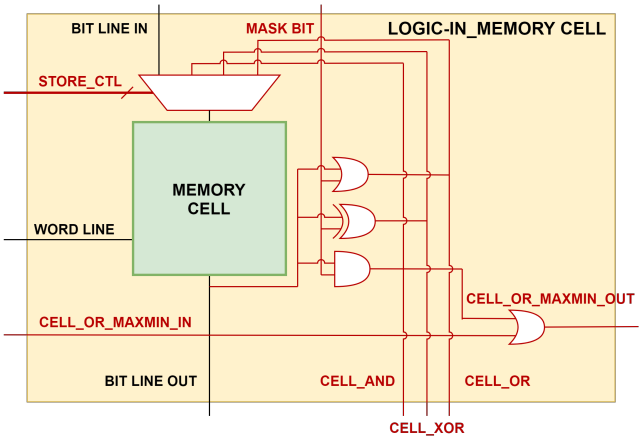



Figure 4.3: Logic-in-Memory bit-cell

Defining as *load logic* the load operation that perform a logic operation, the load logic does not compromise the cell memory content: logic gates such AND, OR and XOR are placed at the output of the memory cell, then on the output bit line. The additional OR port is needed for max and min computation (more details in Section 2.2). In a *store logic* instead, the result of the bitwise operation is fed back as input of the cell.
Both load and store logic requires then just 1 clock cycle to execute.

## 2.2 Maximum and minimum computation - logic around array

The Logic-in-Memory computes maximum and minimum with a very straightforward algorithm, based on an easy procedure people usually use to perform this computation.
In case of maximum search, the algorithm starts evaluating the MSB of a set of memory words. If at least one MSB is equal to 1, the words that have MSB equal to 0 are excluded. The same operation is repeated N times as the number of bits of the words considered. The last operation is performed on the LSB.
The minimum computation works at the same way, with the only difference that the exclusion is done on words that show the considered bit equal to 1, instead of 0.
The implementation of this algorithm is based on the research work [6] and requires some additional logic within the memory cells (Figure 4.3) and around the memory array (Figure 4.4):

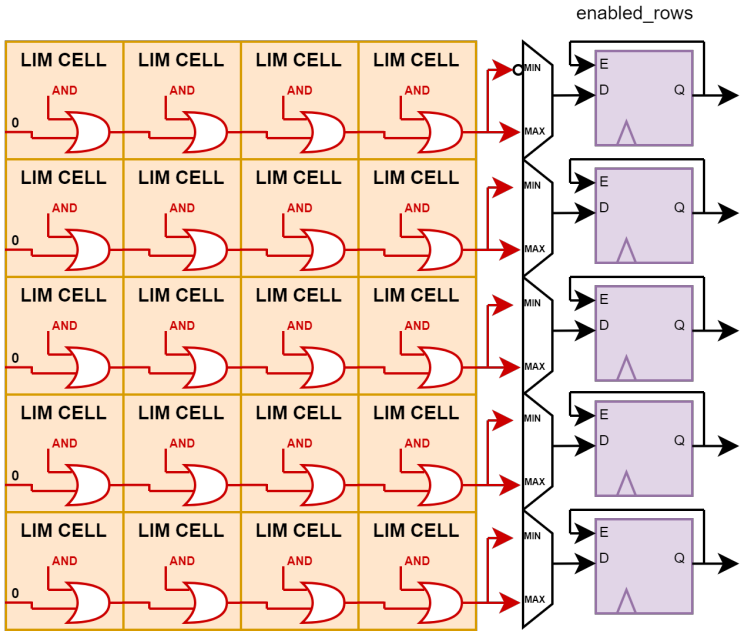



Figure 4.4: Around-array logic for max-min computation

- Some logic around the memory produces a 32-bit mask at each clock cycle, that has only one bit set per clock cycle. The initial mask sets the MSB, while the last mask produced sets the LSB.
- The 32-bit mask is distributed to groups of 4 bytes, therefore the evaluation is done in parallel in all the memory locations involved. The AND gate inside the new memory cell that has as inputs the bit cell content and the input mask, gives as result the bit cell content only in the position of the mask bit set to 1. All the other bits are masked with a 0. A 32 bits wired-or between the AND gate outputs is performed by the additional OR gate in the memory cell. The wired-or net gives the value of the bit to evaluate for each 4-byte group.

- Another piece of logic around memory completes the one clock cycle step of the algorithm. In fact, this part of the logic evaluates the wired-or net corresponding to the words of interest. In case of max computation, words with the wired-or bit equal to 1 will be considered in the next cycle, otherwise those words will be excluded from the comparison. Opposite procedure is done in case of min computation. The information of the enabled words is stored inside registers *enabled_ rows* and updated cycle by cycle. The initial value of these registers is given by the range address decoder, according to the range of words that are of interest. During the execution, only the registers of the *enabled_ rows* equal to 1, can be updated by the wired-or result i each cycle. At the end of the algorithm, the enabled words registers that show a 1, correspond to the memory words with the maximum/minimum value.

The full algorithm requires 33 clock cycles to run: one cycle to initialise the enabled words information and 32 cycles to evaluate 32-bit words. The algorithm requires 33 clock cycles despite of the number of words to evaluate, because all the steps are performed in parallel between words.

## 2.3 Range operations

As indicated previously, the achieved memory architecture is able to perform operations on a certain range of memory locations. The range operations are:

- Allowed in case of a store logic with the same input mask;
- Mandatory in case of min/max computation.

The range operations suppose to have a starting address and the range of 32-bit words to involve, both given by the processor. The logic adopted to enable the required word lines is showed in Figure 4.5. The range decoder logic requires two normal address decoders, one for the starting address and one for the end address (end address is computed by adding the range to the starting address). The two intermediate nets of word lines are combined through some bitwise operations such that only the lines in between are effectively activated. In case of a non-range operation only the starting address is considered and the range logic is bypassed.

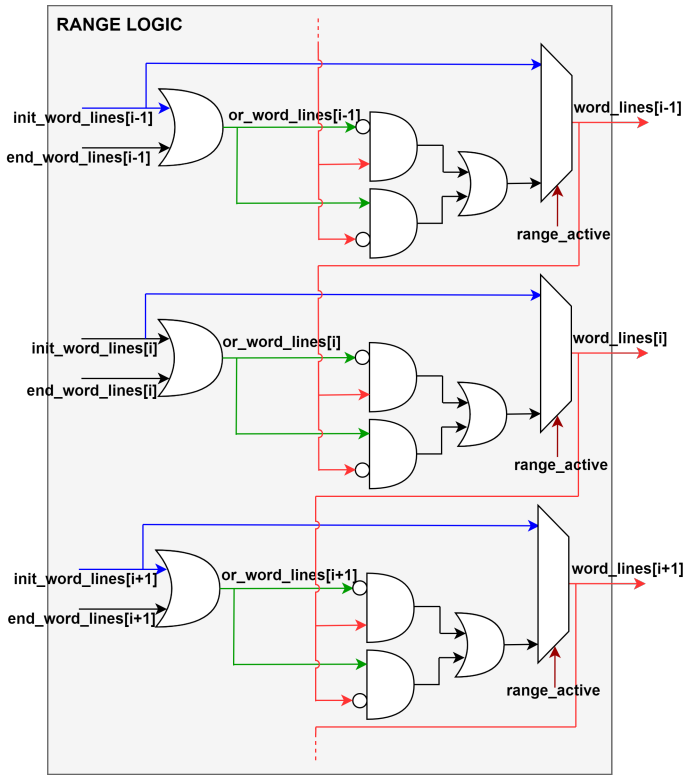



Figure 4.5: Range decoder

# 3 Logic-in-Memory ISA extension

In order to replace the simple RAM model of the RI5CY with the Logic-in-Memory architecture described, the ISA has been expanded introducing new instructions that manage the new memory capabilities. The new Logic-in-Memory extension has been created following the rules of the RISC-V ISA (described in section [ch. 2, 1.2]).
This thesis focused on two different extensions of the ISA in order to reflect two different hardware implementations of the interface Memory-Processor:

- *Same interface Memory-Processor*: this implementation supports the new Logic-in-Memory by keeping the same memory interface of the RISC-V core. This is done to prioritise the flexibility and re-usability of the core in other existing platforms.
- *New interface Memory-Processor*: this implementation instead, supports the new Logic-in-Memory operations by changing the RISC-V memory interface to maximise the efficiency of the communication between processor and memory.

## 3.1 Same interface Memory-Processor ISA extension

The solution proposed in Figure 4.6, keeps the same Memory-Processor interface. Therefore, the Logic-in-Memory functionality cannot be set by any inputs but should be stored somewhere.
This project version proposes:

- The Logic-in-Memory settings are stored inside one specific memory address. The functionality of the memory will depends on what is written in this memory location, that will act as control signal for the entire memory. The information needed are the logic operation type and the range size for the operation.

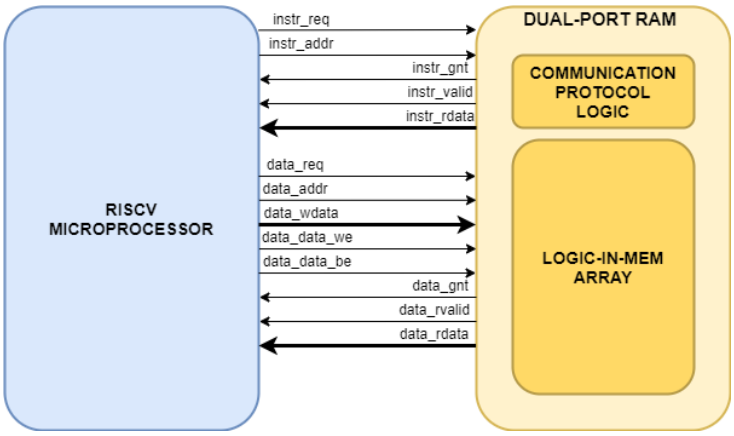



Figure 4.6: RISC-V-Logic-in-Memory interface in the 'Same interface' implementation

- The processor will program the memory simply by performing a store to this memory address. The processor must provide the information needed using the write-data bus already available from the interface.

To support this new functionality the RISC-V ISA introduces new instructions (Figure 4.7):

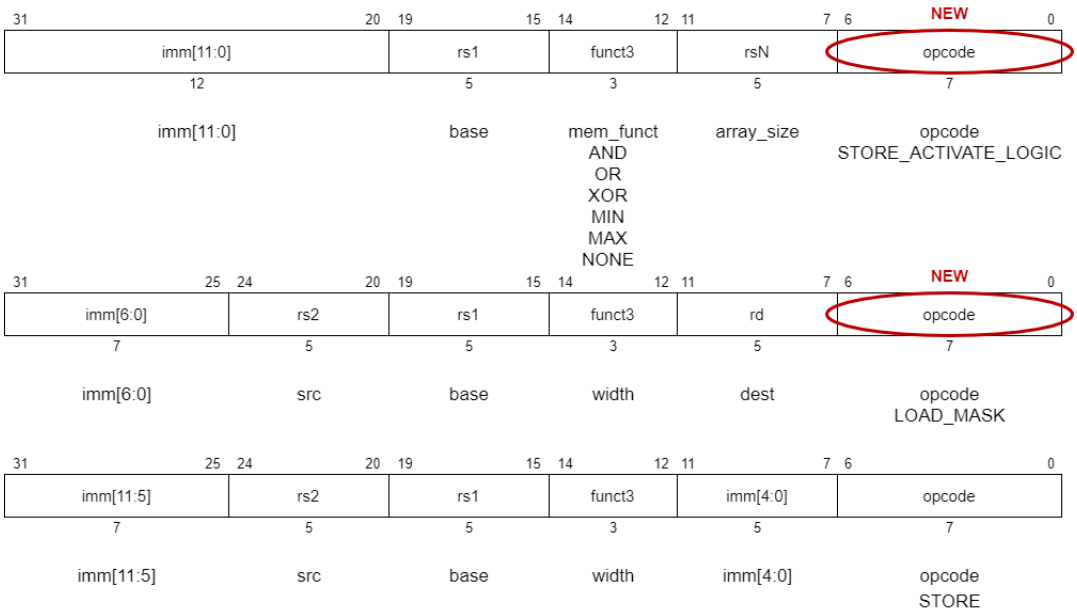


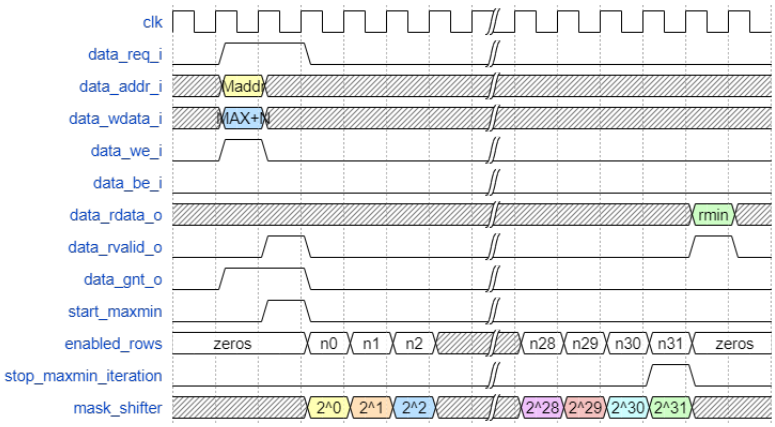

Figure 4.7: New ISA for 'Same interface' implementation

- **STORE_ACTIVATE_LOGIC**. This new instruction allows to program the memory to operate in a certain mode, by writing in the specific memory location. The I instruction format gives the possibility to decode the information about the operation type and the range size. Those two information are packetised to be sent over the 32-bit write data bus. The available operation types are NONE, AND, OR, XOR, MAX, MIN. The range field should be used accordingly when allowed, otherwise it should be set to 0.
- **LOAD_MASK**. The already available LOAD instruction was not sufficient to perform a logic operation in memory. In fact, the RVI load instruction does not give

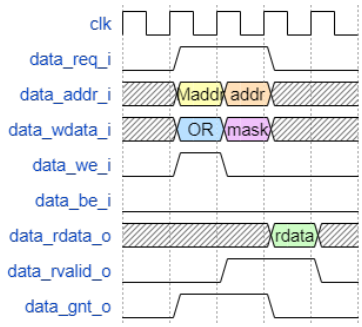
the possibility to send to the memory the input mask. In fact, the LOAD_MASK instruction is introduced only to read the input mask from the Register File through the rs2 field. The value read will be sent to the memory using the write data bus. The compiler should always place this instruction after the logic operations in memory are activated.

- **STORE**. This is not a new instruction for the RISC-V core. A normal store instruction is interpreted by the memory as a logic store instruction if the memory has been programmed accordingly. The value read by the source register rs2 corresponds to the input mask in case of a logic store or to the data to effectively store in case of a normal store.

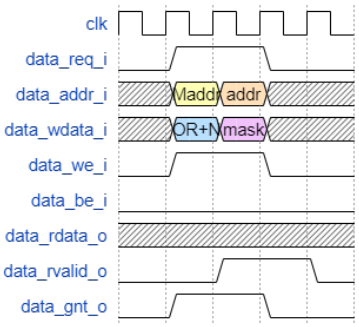
The expected behaviour of memory interface is illustrated in Figure 4.8. In all the cases each actual Logic-in-Memory operation is preceded by the STORE_ACTIVE_LOGIC that passes the type of operation and the range of the operation through the *write_data* bus. The next load/store instruction in memory will be interpreted by the memory according to what is stored in the special address. In case of max/min computation some other internal signals are showed: *start_maxmin* is a pulse signal that kicks off the max/min specific hardware, in fact in the following cycle the *enabled_rows* will assume an initial value (computed according to the starting address and the given range size) and the *mask_shifter* is updated starting showing the value $2^{31}$ to isolate the MSB in the computation; when the mask assumes the value $2^0$ it means that the LSB is being evaluated, so *stop_maxmin_computation* is set to stop the algorithm.




(a) Load Max/Min




(b) Load OR/AND/XOR




(c) Store OR/AND/XOR

Figure 4.8: Waveforms for 'Same interface' implementation

## 3.2 New interface Memory-Processor ISA extension

As already mentioned, another version of the processor-memory system has been explored to reduce as much as possible the memory accesses. Figure 4.9 shows the new signals needed to drive the LIM:

- *logic_memory* to differentiate a normal load/store from a logic load/store.
- *opcode_mem* that specifies the type of the logic operation.
- *asize_mem* to give the range width of the operation.

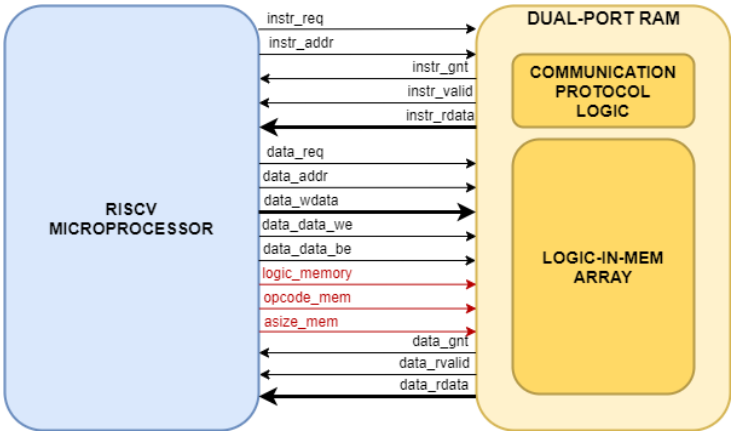



Figure 4.9: RISC-V-Logic-in-Memory interface in the 'New interface' implementation

The new ISA extension includes the following instructions (Figure 4.10):

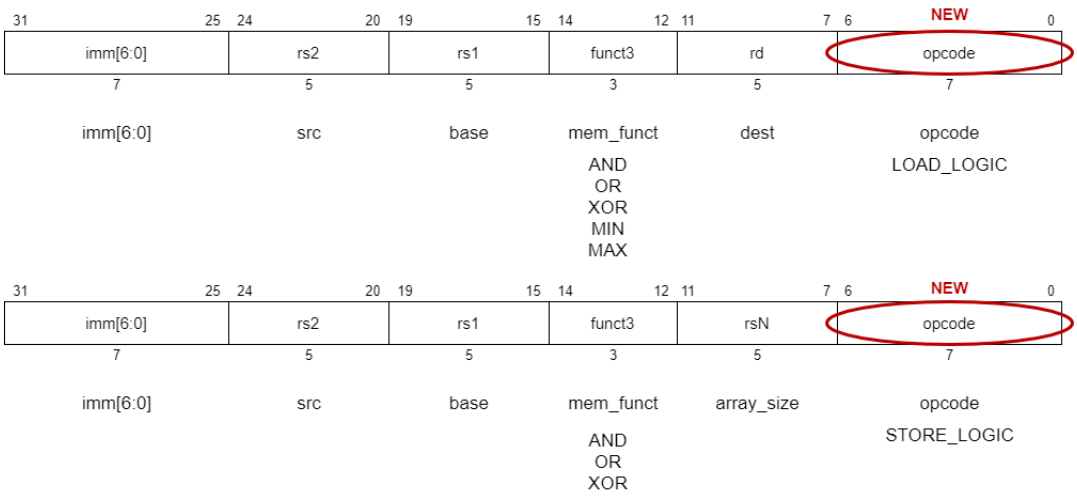
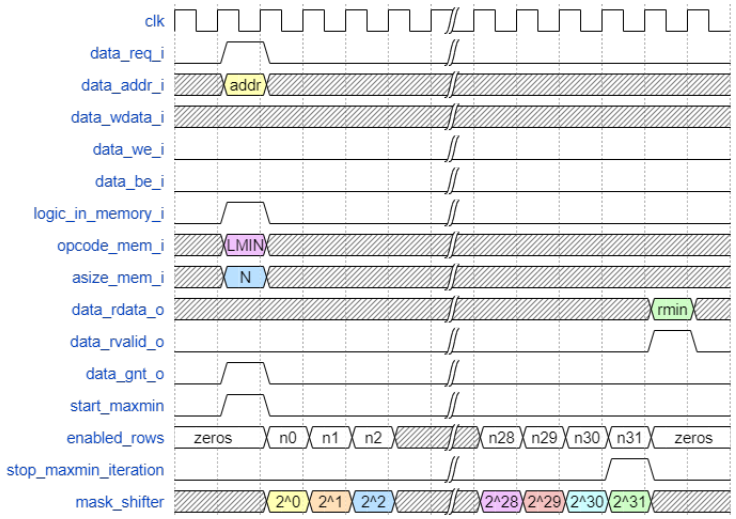


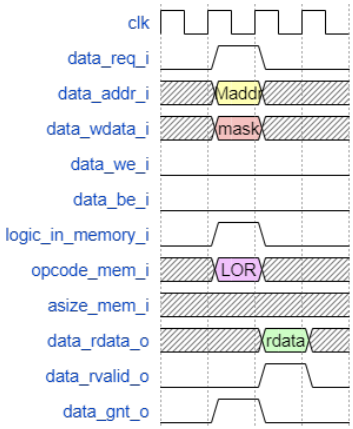

Figure 4.10: New ISA for 'New interface' implementation

- **STORE_LOGIC**. This instruction performs store memory operations on many memory locations at a time, as many as indicated in the register rsN: the value 0 and 1 will both allow the operation on a single integer memory location. The field *mem_funct* indicates the type of the logic operation.
- **LOAD_LOGIC**. The instruction only allows single location load logic operations. Max/min computation is performed on a fixed size array equal to 10. If the comparison should be done among less than 10 integers, the compiler should take care of filling all the memory location involved in order to not compromise the result.

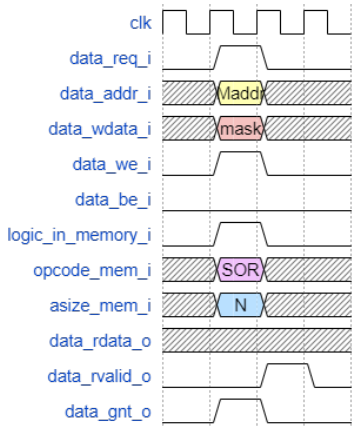
Figure 4.11 shows the expected waveforms for this second solution. It is immediately possible to observe the speed enhancement reached by this architecture. Thanks to the new Logic-in-Memory instructions, the RISC-V produces all the information needed by the new memory, so the memory operation takes only one instruction to complete.




(a) Load Max/Min




(b) Load OR/AND/XOR




(c) Store OR/AND/XOR

Figure 4.11: Waveforms for 'New interface' implementation

# 4 Architectural changes in RISC-V project

To support the new ISA extension, the available RI5CY architecture has been slightly expanded.

## 4.1 Same interface Memory-Processor RI5CY change

The implementation that keeps the same interface with the memory required some changes only into the ID stage (see Figure 4.12):

- The Decoder has been expanded to be able to recognise the new ISA extension with the new instructions STORE_ACTIVE_LOGIC and LOAD_MASK and then to set the appropriate control signals.
- The LOAD_MASK instruction requires a new Immediate format. For this reason the *imm_logmem_type* has been introduced. It is signed extended version of the last 7 MSBs of the instruction to decode *instr_rdata[31:25]*.
- The STORE_ACTIVE_LOGIC instruction needs to pass to the memory the value of the range and the operation type. The range is obtained by a reading of the Register File through the rs2 field, while the operation type is obtained directly by the instruction *instr_rdata[14:12]*. Those two values are merged in a 32-bit value and the result is sent to the memory through the *alu_operand_c*.

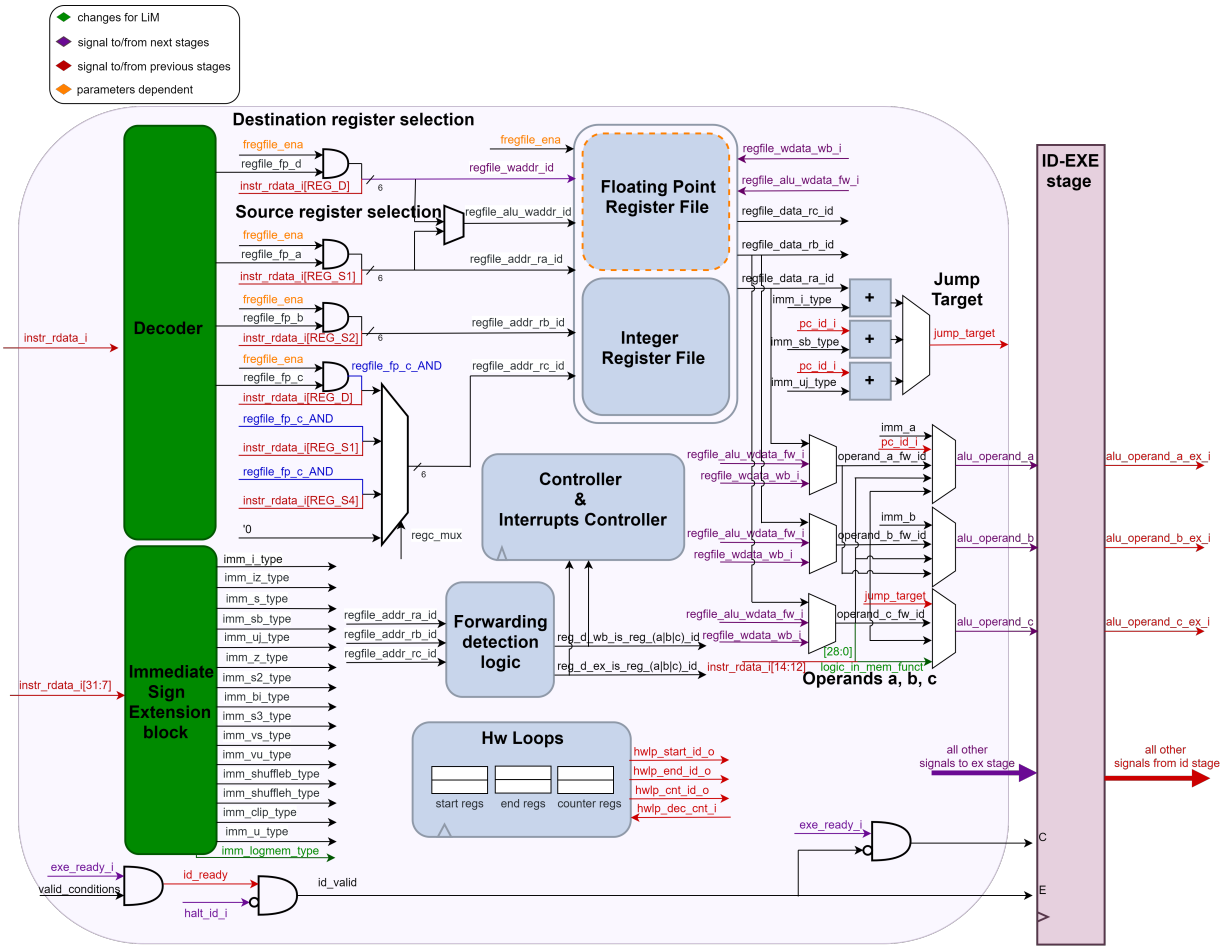



Figure 4.12: ID stage architectural change for 'Same interface' implementation

## 4.2 Same interface Memory-Processor RI5CY changes

This implementation requires changes in the ID and LSU stages because of the new signals introduced in the interface between microprocessor and memory (see Figures 4.13 and 4.14):

- The Decoder has been expanded as well to support the new instructions *LOAD_LOGIC* and *STORE_LOGIC*.
- The same Immediate format *imm_logmem_type* has been added also in this case because it is needed in both instructions.
- The new memory related signals *data_opcode_mem_o* and *data_logic_in_memory_o* are available in the interface with the memory in the LSU stage. Anyway, those signals are driven in the ID stage by the decoder, that assigns to them 0, or *instr_rdata[14:12]* and 1 respectively.

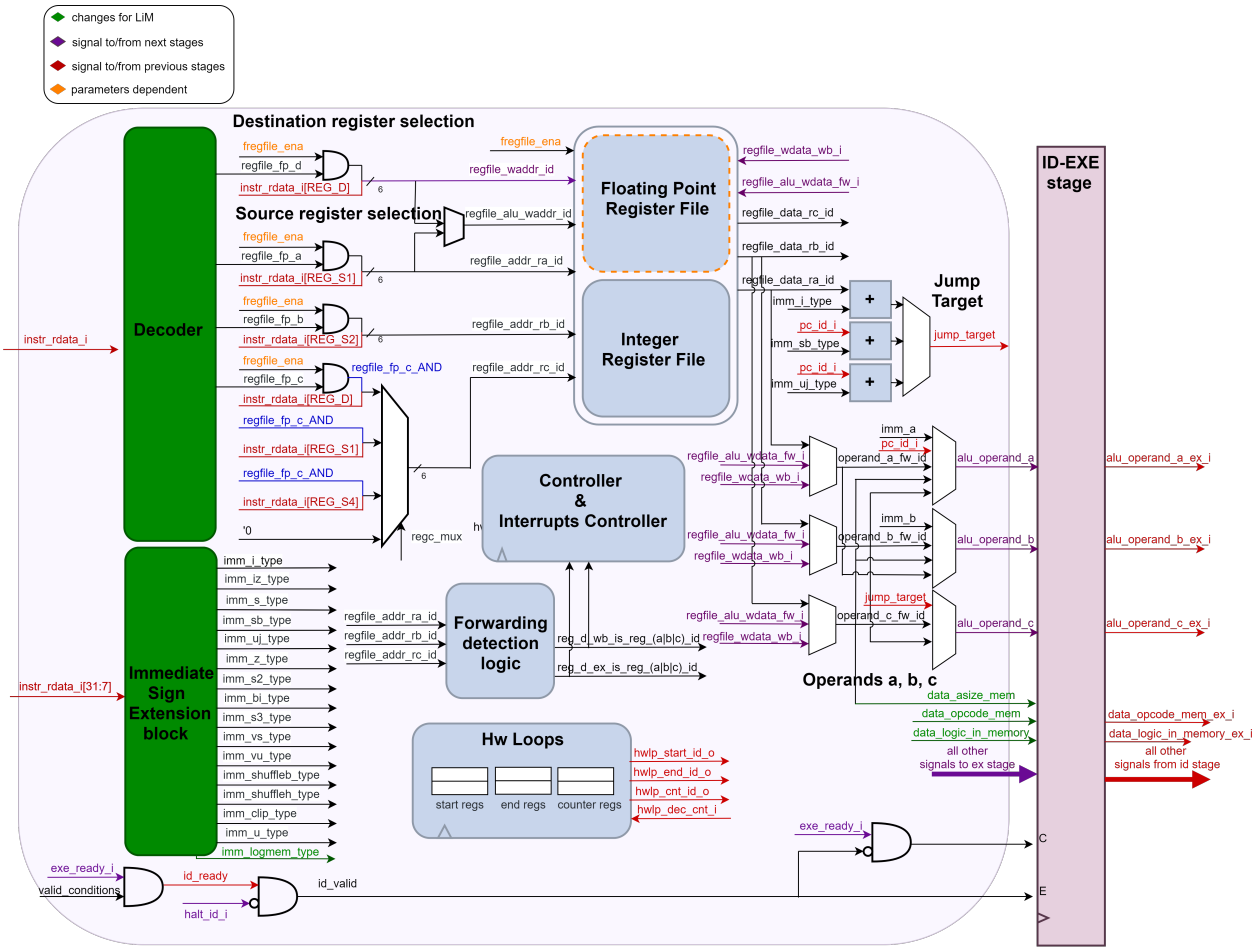



Figure 4.13: ID stage architectural change for 'New interface' implementation

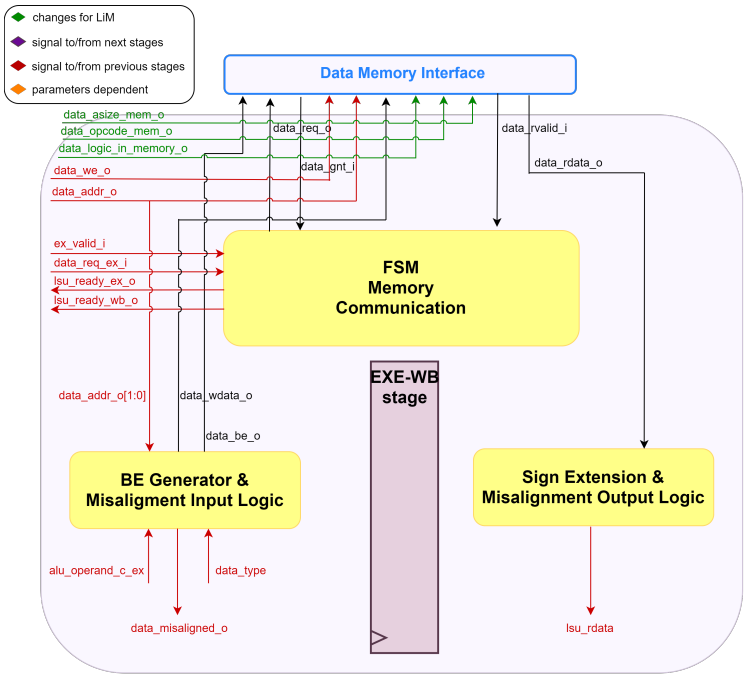
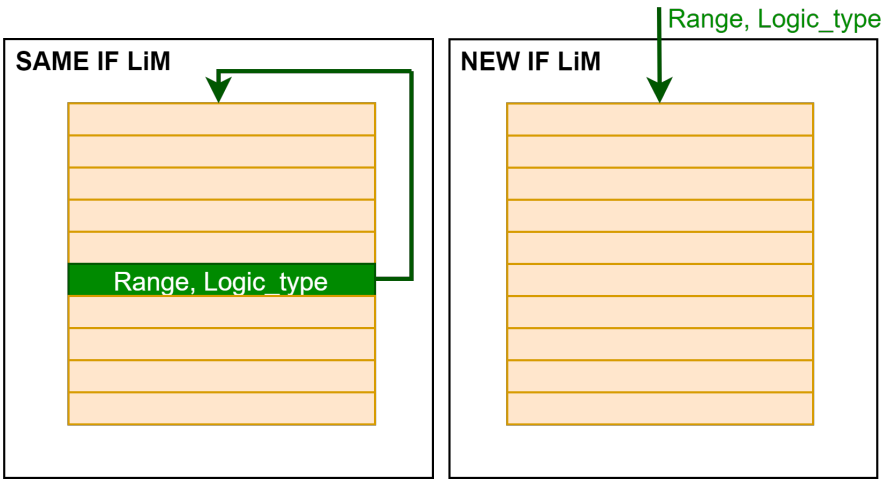



Figure 4.14: LSU stage architectural change for 'New interface' implementation

## 4.3 Differences between the Logic-in-Memory implementations

The Logic-in-Memory used in the two versions of the project are very similar and both reflect the architectural description done in Section 2 of this chapter.
This section instead, aims to highlight the small differences between the two LiM projects. Obviously those are caused by the need to have different interfaces. Figure 4.15 emphasises the variation adopted for the two LiM versions:




(a) 'Same interface' Logic-in-Memory (b) 'New interface' Logic-in-Memory

Figure 4.15: Logic-in-Memory implementation differences

- 'Same interface' Logic-in-Memory keeps the original interface, then the RI5CY can only communicate with the memory using the standard signals. In fact, information regarding the logic operation to perform and the range of locations involved in the operation are stored inside a memory location. The RI5CY performs a store in a

specific memory location and all the control signals needed to coordinate the memory operations are driven directly by that memory location.

- 'New interface' Logic-in-Memory allows a new interface with the new signals. Therefore, the needed signals for the coordination of the new memory operations are driven directly by the RI5CY during a load or a store in memory.

# Chapter 5

# Simulations and Synthesis

## 1 Tools

The two different versions of the RISC-V and LiM were designed using SystemVerilog hardware description language (more details in Appendix A). The simulations were made using Modelsim-Altera Starter Edition 10.6c, while for their synthesis the adopted tool was Synopsys 2018.06.

## 2 Simulation with custom programs

Many times in this thesis has been stated that new ISA extension can generally improve the speed of the RISC-V in terms of execution time. In order to prove that, a set of programs have been written and simulated.

All the programs have been written in C language and then compiled with available RISC-V compiler (https://github.com/riscv/riscv-gnu-toolchain). However, the LiM ISA extension introduced in the hardware with this thesis, is not supported by the compiler. In fact, before any simulations all the programs have been compiled using the available ISA and then the new instructions from the LiM ISA extensions have been manually added to replace some parts of the code. The mixed software and manual compilation is not ideal, because takes a lot of time and does not guarantee any smart optimisations that the compiler alone would perform. For this reason, a limited amount of test programs have been compiled and simulated.

The methodology to determine the correctness of the results, consists of:

1. Writing the C program and compiling it with the available RISC-V compiler. The result is the <program>.hex file containing the instructions in machine language. The extensions used are the standard RVI (Base Integer extension) and RVM (Integer Multiplication and Division extension).
2. Kicking-off the simulation on the RISC-V system with the LiM. The LiM is used as a normal memory in this case, because the compiler do not use the new implemented instructions to translate the program. The simulation results will be the baseline for the comparison with the LiM-specific instructions.
3. Manually converting the instructions file <program>.hex in <program>_lim.hex, that contains the LiM instructions replacing a subset of compiled instructions.

4. Kicking-off the simulation on the RISC-V system with the LiM. This time, the simulation results contain the new ISA extension.

5. Comparing the two simulation outcomes.

This procedure is executed on the same programs for the both versions of the RISC-V: Same Interface and New Interface. Obviously, steps 1 and 2 bring to the same <program>.hex in both project versions. This is because the ISA used is the same and the changes done to introduce the two LiM ISA extensions do not alter the execution of the existing supported instructions.
For the sake of simplicity, the ISA without LiM extension, ISA with LiM extension for the new interface and ISA with LiM extension for the same interface will be referred respectively as old_ISA, newIF_ISA and sameIF_ISA.

To verify in the first place the functionality of the two project flavours, a couple of programs have been written and tested. Details about the programs and simulation results are described in the next sections.

## 2.1 Bitwise

The aim of this first test is to demonstrate the correct functionality of the two RISC-V projects with the Logic-in-Memory. In particular, this test program aims to show all the LiM bitwise operations performed on single memory words or on range of memory words.

Broadly speaking, the program *bitwise.c* performs the bitwise operations on an integer vector of 10 elements and on a standalone integer element. The bitwise operations performed are in order OR, AND and XOR. The final result is stored in another memory location and it is obtained with a combination of the value assumed by an element of the vector and the standalone variable.

### 2.1.1 C program

In the below figure is shown the C code used for the compilation. To facilitate the replacement of some instructions with the LiM instructions, the program explicitly declares the memory address of most variables.

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

int main(int argc, char *argv[])
{
    /* variable declaration */
    int N = 5, i, mask_or, mask_and, mask_xor;
    int *vector = 0x030000, *stand_alone = 0x30040, *final_result = 0x30080;

    /* fill vector */
    for(i=0; i<N; i++){
      vector[i] = i*13467;
    }
    *stand_alone = vector[1]+0x768;

    /* OR operation */
    mask_or = 0xF1;
    for(i=0; i<N; i++){
      vector[i] = vector[i] | mask_or;
    }
    *stand_alone = *stand_alone | mask_or;

    /* AND operation */
    mask_and = vector[N-1] & 0x8F;
    for(i=0; i<N; i++){
      vector[i] = vector[i] & mask_and;
    }
    *stand_alone = *stand_alone & mask_and;

    /* XOR operation */
    mask_xor = vector[N-2] ^ 0xF0;
    for(i=0; i<N; i++){
      vector[i] = vector[i] ^ mask_xor;
```

```
34      }
35      *stand_alone = *stand_alone ^ mask_xor;
36
37      *final_result = ~vector[N-3] + ~(*stand_alone);
38
39      return EXIT_SUCCESS;
40  }
```

Listing 5.1: Custom program *bitwise.c*

### 2.1.2 Execution time estimation

According to the structure of the program, it is possible to perform a rough estimation of the executed time in terms of number of clock cycles (cc) in the three cases old_ISA, newIF_ISA and sameIF_ISA. This computation only takes into account the meaningful part of the code and assumes N as the size of the vector.

$$Execution\_time_{old\_ISA}(bitwise.c) \approx 3(3N(vector\_element) + 3(single\_element)) + 6(final\_result) \tag{5.1}$$

$$Execution\_time_{newIF\_ISA}(bitwise.c) \approx 3(1(vector\_element) + 1(single\_element)) + 4(final\_result) \tag{5.2}$$

$$Execution\_time_{sameIF\_ISA}(bitwise.c) \approx 3(1(mem\_active) + 1(vector\_element) + 1(single\_element)) + 4(final\_result) + 1(mem\_active) \tag{5.3}$$

In 5.1 the factor 3 comes from the consideration that each vector element stored in

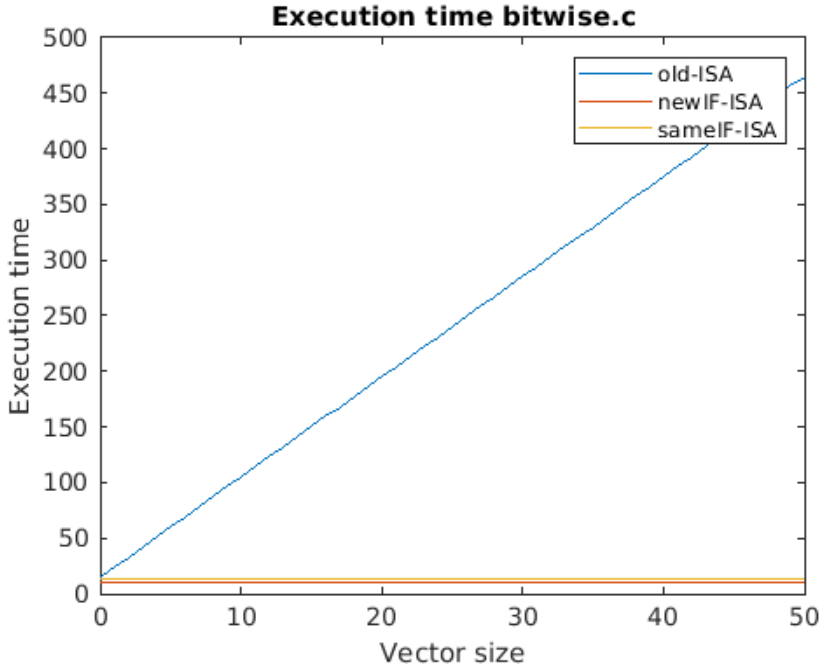



Figure 5.1: Estimation execution time bitwise.c in old_ISA, newIF_ISA and sameIF_ISA

memory, needs to be loaded into the microprocessor, combined with the mask value and then stored back in the memory. Same applies to the standalone element. Those operations

are repeated 3 times as the number of bitwise types (OR, AND, XOR). For the final result instead, the RISC-V needs to load two variables, perform the negation on both, combine the values with addition and then store the result.
In 5.2 and 5.3 each memory element and standalone element can be manipulated directly in memory, so factor equal to 1. The final result is obtained performing the negation directly during the load from memory, then the addition and the the result is normally stored back in memory. The only difference between the two is the need to set up the memory operation in the latter case.
As clearly highlighted from Figure 5.1, the newIF_ISA and the sameIF_ISA execution times do not depend on the size of vector, because all the elements of the vector perform the same operation with the same mask. The gap in terms of execution time becomes very important in case of a large vector. Simulation results are collected with N = 5.

### 2.1.3 Simulation results

The RISC-V has a debug unit, that is used only during the simulation, which prints a file containing the simulation results and showing all the meaningful details of the executed instructions. This debug unit has been very useful to compare the results in the three cases.
Only meaningful parts of the simulation are reported as follows. A white space between the rows shows a discontinuity between the parts of the program, always associated to the execution of loops.
As stated before, the main differences between the old_ISA program and the other ones are the missing loops replaced by range operations in memory, and bitwise operations are performed directly in memory whenever possible.
For a better understanding of the simulation results reported below, it would be helpful to refer to the variables memory addresses.

```
Time     Cycles PC    Instr    Mnemonic

2016ns   197 00000218 0007a023 sw              x0, 0(x15)          x15:00030000  PA:00030000
2026ns   198 0000021c 00d7a823 sw              x13, 16(x15)        x13:0000d26c x15:00030000  PA:00030010
2036ns   199 00000220 00030737 lui             x14, 0x30000        x14=00030000
2046ns   200 00000224 01478613 addi            x12, x15, 20        x12=00030014 x15:00030000
2056ns   201 00000228 0007a683 lw              x13, 0(x15)         x13=00000000 x15:00030000  PA:00030000
2066ns   202 0000022c 00478793 addi            x15, x15, 4         x15=00030004 x15:00030000
2076ns   203 00000230 0f16e693 ori             x13, x13, 241       x13=000000f1 x13:00000000
2086ns   204 00000234 fed7ae23 sw              x13, -4(x15)        x13:000000f1 x15:00030004  PA:00030000
2096ns   205 00000238 fec798e3 bne             x15, x12, -16       x15:00030004 x12:00030014

2376ns   233 00000238 fec798e3 bne             x15, x12, -16       x15:00030014 x12:00030014
2386ns   234 0000023c 04072783 lw              x15, 64(x14)        x15=00003c03 x14:00030000  PA:00030040
2396ns   235 00000240 000306b7 lui             x13, 0x30000        x13=00030000
2406ns   236 00000244 0f17e793 ori             x15, x15, 241       x15=00003cf3 x15:00003c03
2416ns   237 00000248 04f72023 sw              x15, 64(x14)        x15:00003cf3 x14:00030000  PA:00030040
2426ns   238 0000024c 01072703 lw              x14, 16(x14)        x14=0000d2fd x14:00030000  PA:00030010
2436ns   239 00000250 000307b7 lui             x15, 0x30000        x15=00030000
2446ns   240 00000254 01478593 addi            x11, x15, 20        x11=00030014 x15:00030000
2456ns   241 00000258 08f77713 andi            x14, x14, 143       x14=0000008d x14:0000d2fd
2466ns   242 0000025c 0007a603 lw              x12, 0(x15)         x12=000000f1 x15:00030000  PA:00030000
2476ns   243 00000260 00478793 addi            x15, x15, 4         x15=00030004 x15:00030000
2486ns   244 00000264 00e67633 and             x12, x12, x14       x12=00000081 x12:000000f1 x14:0000008d
2496ns   245 00000268 fec7ae23 sw              x12, -4(x15)        x12:00000081 x15:00030004  PA:00030000
2506ns   246 0000026c feb798e3 bne             x15, x11, -16       x15:00030004 x11:00030014

2776ns   273 00000268 fec7ae23 sw              x12, -4(x15)        x12:0000008d x15:00030014  PA:00030010
2786ns   274 0000026c feb798e3 bne             x15, x11, -16       x15:00030014 x11:00030014
2796ns   275 00000270 0406a783 lw              x15, 64(x13)        x15=00003cf3 x13:00030000  PA:00030040
2816ns   277 00000274 00e7f733 and             x14, x15, x14       x14=00000081 x15:00003cf3 x14:0000008d
2826ns   278 00000278 00c6a783 lw              x15, 12(x13)        x15=00000081 x13:00030000  PA:0003000c
2836ns   279 0000027c 04e6a023 sw              x14, 64(x13)        x14:00000081 x13:00030000  PA:00030040
2846ns   280 00000280 00030737 lui             x14, 0x30000        x14=00030000
2856ns   281 00000284 0f07c793 xori            x15, x15, 240       x15=00000071 x15:00000081
2866ns   282 00000288 000306b7 lui             x13, 0x30000        x13=00030000
2876ns   283 0000028c 01470593 addi            x11, x14, 20        x11=00030014 x14:00030000
2886ns   284 00000290 00072603 lw              x12, 0(x14)         x12=00000081 x14:00030000  PA:00030000
2896ns   285 00000294 00470713 addi            x14, x14, 4         x14=00030004 x14:00030000
2906ns   286 00000298 00f64633 xor             x12, x12, x15       x12=000000f0 x12:00000081 x15:00000071
2916ns   287 0000029c fec72e23 sw              x12, -4(x14)        x12:000000f0 x14:00030004  PA:00030000
2926ns   288 000002a0 feb718e3 bne             x14, x11, -16       x14:00030004 x11:00030014

3206ns   316 000002a0 feb718e3 bne             x14, x11, -16       x14:00030014 x11:00030014
3216ns   317 000002a4 0406a703 lw              x14, 64(x13)        x14=00000081 x13:00030000  PA:00030040
3226ns   318 000002a8 00000513 addi            x10, x0, 0          x10=00000000
3236ns   319 000002ac 00e7c7b3 xor             x15, x15, x14       x15=000000f0 x15:00000071 x14:00000081
3246ns   320 000002b0 0086a703 lw              x14, 8(x13)         x14=000000f4 x13:00030000  PA:00030008
```

```
3256ns   321 000002b4 04f6a023 sw              x15, 64(x13)        x15:000000f0 x13:00030000  PA:00030040
3266ns   322 000002b8 fff7c793 xori            x15, x15, -1        x15=ffffff0f x15:000000f0
3276ns   323 000002bc fff74713 xori            x14, x14, -1        x14=ffffff0b x14:000000f4
3286ns   324 000002c0 00f707b3 add             x15, x14, x15       x15=fffffe1a x14:ffffff0b x15:ffffff0f
3296ns   325 000002c4 08f6a023 sw              x15, 128(x13)       x15:fffffe1a x13:00030000  PA:00030080
3306ns   326 000002c8 00008067 jalr            x0, x1, 0            x1:000001d8
```

Listing 5.2: Extract of simulation result of *bitwise.c* - old_ISA

```
Time     Cycles PC    Instr    Mnemonic

2016ns   197 00000218 0007a023 sw              x0, 0(x15)          x15:00030000  PA:00030000
2026ns   198 0000021c 00d7a823 sw              x13, 16(x15)        x13:0000d26c x15:00030000  PA:00030010
2036ns   199 00000220 0f100713 addi            x14, x0, 241        x14=000000f1
2046ns   200 00000224 00500693 addi            x13, x0, 5          x13=00000005
2056ns   201 00000228 00e7b6bb sw_or           N-x13 x14, 0(x15)   x14:000000f1 x15:00030000 x13:00000005  PA:00030000
2066ns   202 0000022c 04078593 addi            x11, x15, 64        x11=00030040 x15:00030000
2076ns   203 00000230 00e5b03b sw_or           N-x0 x14, 0(x11)    x14:000000f1 x11:00030040  PA:00030040
2086ns   204 00000234 08f00713 addi            x14, x0, 143        x14=0000008f
2096ns   205 00000238 20e7a71b lw_and          x14, x14, 16(x15)   x14=0000008d x14:0000008f x15:00030000  PA:00030010
2116ns   207 0000023c 00e7a6bb sw_and          N-x13 x14, 0(x15)   x14:0000008d x15:00030000 x13:00000005  PA:00030000
2126ns   208 00000240 00e5a03b sw_and          N-x0 x14, 0(x11)    x14:0000008d x11:00030040  PA:00030040
2136ns   209 00000244 0f000713 addi            x14, x0, 240        x14=000000f0
2146ns   210 00000248 18e7971b lw_xor          x14, x14, 12(x15)   x14=00000071 x14:000000f0 x15:00030000  PA:0003000c
2166ns   212 0000024c 00e796bb sw_xor          N-x13 x14, 0(x15)   x14:00000071 x15:00030000 x13:00000005  PA:00030000
2176ns   213 00000250 00e5903b sw_xor          N-x0 x14, 0(x11)    x14:00000071 x11:00030040  PA:00030040
2186ns   214 00000254 fff00693 addi            x13, x0, -1         x13=ffffffff
2196ns   215 00000258 10d7971b lw_xor          x14, x13, 8(x15)    x14=ffffff0b x13:ffffffff x15:00030000  PA:00030008
2206ns   216 0000025c 00d5961b lw_xor          x12, x13, 0(x11)    x12=ffffff0f x13:ffffffff x11:00030040  PA:00030040
2226ns   218 00000260 00c70633 add             x12, x14, x12       x12=fffffe1a x14:ffffff0b x12:ffffff0f
2236ns   219 00000264 04c5a023 sw              x12, 64(x11)        x12:fffffe1a x11:00030040  PA:00030080
2246ns   220 00000268 00000513 addi            x10, x0, 0          x10=00000000
2256ns   221 0000026c 00008067 jalr            x0, x1, 0            x1:000001d8
```

Listing 5.3: Extract of simulation result of *bitwise.c* - newIF_ISA

```
Time     Cycles PC    Instr    Mnemonic

2016ns   197 00000218 0007a023 sw              x0, 0(x15)          x15:00030000  PA:00030000
2026ns   198 0000021c 00d7a823 sw              x13, 16(x15)        x13:0000d26c x15:00030000  PA:00030010
2036ns   199 00000220 00020637 lui             x12, 0x20000        x12=00020000
2046ns   200 00000224 0f100713 addi            x14, x0, 241        x14=000000f1
2056ns   201 00000228 00500693 addi            x13, x0, 5          x13=00000005
2066ns   202 0000022c ffc636bb sw_active_or    Nx13 -4(x12)        x12:00020000 x13:00000005  PA:0001fffc
2076ns   203 00000230 00e7a023 sw              x14, 0(x15)         x14:000000f1 x15:00030000  PA:00030000
2086ns   204 00000234 ffc6303b sw_active_or    Nx0 -4(x12)         x12:00020000  PA:0001fffc
2096ns   205 00000238 04e7a023 sw              x14, 64(x15)        x14:000000f1 x15:00030000  PA:00030040
2106ns   206 0000023c ffc6203b sw_active_and   Nx0 -4(x12)         x12:00020000  PA:0001fffc
2116ns   207 00000240 08f00713 addi            x14, x0, 143        x14=0000008f
2126ns   208 00000244 20e7a71b lw_mask         x14, x14, 16(x15)   x14=0000008d x14:0000008f x15:00030000  PA:00030010
2146ns   210 00000248 04e7a023 sw              x14, 64(x15)        x14:0000008d x15:00030000  PA:00030040
2156ns   211 0000024c ffc626bb sw_active_and   Nx13 -4(x12)        x12:00020000 x13:00000005  PA:0001fffc
2166ns   212 00000250 00e7a023 sw              x14, 0(x15)         x14:0000008d x15:00030000  PA:00030000
2176ns   213 00000254 ffc6103b sw_active_xor   Nx0 -4(x12)         x12:00020000  PA:0001fffc
2186ns   214 00000258 0f000713 addi            x14, x0, 240        x14=000000f0
2196ns   215 0000025c 18e7a71b lw_mask         x14, x14, 12(x15)   x14=00000071 x14:000000f0 x15:00030000  PA:0003000c
2216ns   217 00000260 04e7a023 sw              x14, 64(x15)        x14:00000071 x15:00030000  PA:00030040
2226ns   218 00000264 ffc616bb sw_active_xor   Nx13 -4(x12)        x12:00020000 x13:00000005  PA:0001fffc
2236ns   219 00000268 00e7a023 sw              x14, 0(x15)         x14:00000071 x15:00030000  PA:00030000
2246ns   220 0000026c fff00693 addi            x13, x0, -1         x13=ffffffff
2256ns   221 00000270 ffc6103b sw_active_xor   Nx0 -4(x12)         x12:00020000  PA:0001fffc
2266ns   222 00000274 10d7a71b lw_mask         x14, x13, 8(x15)    x14=ffffff0b x13:ffffffff x15:00030000  PA:00030008
2276ns   223 00000278 04078793 addi            x15, x15, 64        x15=00030040 x15:00030000
2286ns   224 0000027c 00d7a59b lw_mask         x11, x13, 0(x15)    x11=ffffff0f x13:ffffffff x15:00030040  PA:00030040
2306ns   226 00000280 00b705b3 add             x11, x14, x11       x11=fffffe1a x14:ffffff0b x11:ffffff0f
2316ns   227 00000284 ffc6003b sw_active_none  Nx0 -4(x12)         x12:00020000  PA:0001fffc
2326ns   228 00000288 04b7a023 sw              x11, 64(x15)        x11:fffffe1a x15:00030040  PA:00030080
2336ns   229 0000028c 00000513 addi            x10, x0, 0          x10=00000000
2346ns   230 00000290 00008067 jalr            x0, x1, 0            x1:000001d8
```

Listing 5.4: Extract of simulation result of *bitwise.c* - sameIF_ISA

## 2.2 Max-Min

The program *max_ min.c* simply performs the maximum and the minimum computation within a given vector. The used algorithm is the classical one, where max and min variable are initialised to the first element of the array. The max and min values are computed by a comparison of the max/min variable with all the vector elements. This algorithm has been useful to test the correct functionality of the max/min operations available in the LiM.

### 2.2.1 C program

The implemented C code for this algorithm also specifies the memory address for the most meaningful variables.

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    /* variable declaration */
    int N = 10;
    int i;
    int *vector = 0x30000;
    int *max = 0x300F0;
    int *min = 0x300F4;

    /* fill vector */
    for(i=0; i<N; i++){
      vector[i] = i*13467;
    }

    /* MAX operation */
    *max = vector[0];

    for(i=1; i<N; i++){
      if(vector[i]>*max){
    *max = vector[i];
        }
    }

   /* MIN operation */
    *min = vector[0];

    for(i=1; i<N; i++){
      if(vector[i]<*min){
    *min = vector[i];
        }
    }

    return EXIT_SUCCESS;
}
```

Listing 5.5: Custom program *max_ min.c*

### 2.2.2 Execution time estimation

The execution time estimation is showed below, assuming that N is the size of the vector:

$$Execution\_time_{old\_ISA}(max\_min.c) \approx 4N(max) + 4N(min) \tag{5.4}$$

$$Execution\_time_{newIF\_ISA}(max\_min.c) \approx 33(max) + 33(min) \tag{5.5}$$

$$Execution\_time_{sameIF\_ISA}(max\_min.c) \approx 1(mem\_active) + 33(max) + 1(mem\_active) + 33(min) + 1(mem\_active) \tag{5.6}$$

The execution time in the case of the old_ISA 5.4, has been estimated taking into account that the max/min variable will be computed within a for-cycle that evaluates all the element of the vector. Each cycle consists of loading the max/min and the vector element, then performing the comparison and finally updating the max/min value. The algorithm actually initialises the max/min to the first memory element, so theoretically the four described operations should be repeated N-1 times. However, N is considered for this rough estimation to take in to account the overhead due to the max/min variable initialisation.
The new implemented ISAs, also for this second program give a big advantage with a big N. The reason is again that the LiM computes the max/min in parallel on all the vector elements, by considering one bit at a time. See 5.5, 5.6 and Figure 5.2. Similarly to the previous case, also in the *max_ min.c* the only difference between the two LiM implementations is the need to set up the memory to perform the desired operation. Anyway, for a big N, this difference is negligible.
The chosen N for the simulation is 10. In fact, according to the approximate estimation, N=10 should already show an improvement in terms of execution time, Figure 5.2. The

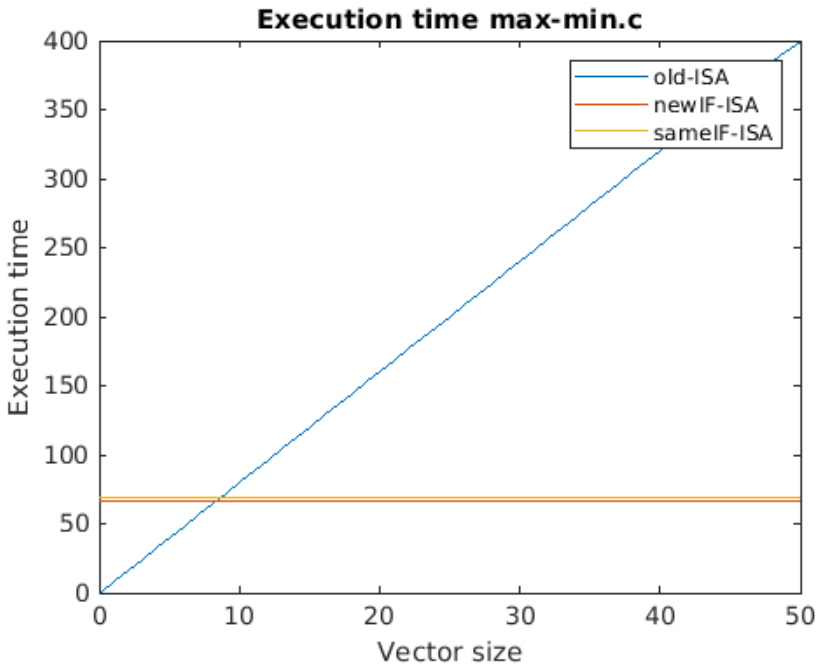



Figure 5.2: Estimation execution time max_min.c in old_ISA, newIF_ISA and sameIF_ISA

number N has been chosen to minimise the simulation time but at the same time, big enough to show the LiM benefits.

### 2.2.3 Simulation results

Here below the details on the simulations. Note that the vector has been initialised with increasing numbers, so the maximum is in the last position, while the minimum corresponds to the first element. In the old_ISA case, white spaces are used to indicate the presence of loops.

```
Time     Cycles PC    Instr    Mnemonic

2086ns   204 00000254 0007a023 sw              x0, 0(x15)          x15:00030000  PA:00030000
2096ns   205 00000258 0e07a823 sw              x0, 240(x15)        x15:00030000  PA:000300f0
2106ns   206 0000025c 02870613 addi            x12, x14, 40        x12=00030028 x14:00030000
2116ns   207 00000260 00478793 addi            x15, x15, 4         x15=00030004 x15:00030000
2126ns   208 00000264 0007a683 lw              x13, 0(x15)         x13=0000349b x15:00030004  PA:00030004
2136ns   209 00000268 0f072583 lw              x11, 240(x14)       x11=00000000 x14:00030000  PA:000300f0
2156ns   211 0000026c 00d5d463 bge             x11, x13, 8         x11:00000000 x13:0000349b
2166ns   212 00000270 0ed72823 sw              x13, 240(x14)       x13:0000349b x14:00030000  PA:000300f0
2176ns   213 00000274 00478793 addi            x15, x15, 4         x15=00030008 x15:00030004
2186ns   214 00000278 fec796e3 bne             x15, x12, -20       x15:00030008 x12:00030028
2216ns   217 00000264 0007a683 lw              x13, 0(x15)         x13=00006936 x15:00030008  PA:00030008

2886ns   284 00000270 0ed72823 sw              x13, 240(x14)       x13:0001d973 x14:00030000  PA:000300f0
2896ns   285 00000274 00478793 addi            x15, x15, 4         x15=00030028 x15:00030024
2906ns   286 00000278 fec796e3 bne             x15, x12, -20       x15:00030028 x12:00030028
2916ns   287 0000027c 00072783 lw              x15, 0(x14)         x15=00000000 x14:00030000  PA:00030000
2926ns   288 00000280 00470713 addi            x14, x14, 4         x14=00030004 x14:00030000
2936ns   289 00000284 0ef72823 sw              x15, 240(x14)       x15:00000000 x14:00030004  PA:000300f4
2946ns   290 00000288 000307b7 lui             x15, 0x30000        x15=00030000
2956ns   291 0000028c 02878613 addi            x12, x15, 40        x12=00030028 x15:00030000
2966ns   292 00000290 00072683 lw              x13, 0(x14)         x13=0000349b x14:00030004  PA:00030004
2976ns   293 00000294 0f47a583 lw              x11, 244(x15)       x11=00000000 x15:00030000  PA:000300f4
2996ns   295 00000298 00b6d463 bge             x13, x11, 8         x13:0000349b x11:00000000
3026ns   298 000002a0 00470713 addi            x14, x14, 4         x14=00030008 x14:00030004
3036ns   299 000002a4 fec716e3 bne             x14, x12, -20       x14:00030008 x12:00030028
3066ns   302 00000290 00072683 lw              x13, 0(x14)         x13=00006936 x14:00030008  PA:00030008

3766ns   372 00000290 00072683 lw              x13, 0(x14)         x13=0001d973 x14:00030024  PA:00030024
3776ns   373 00000294 0f47a583 lw              x11, 244(x15)       x11=00000000 x15:00030000  PA:000300f4
3796ns   375 00000298 00b6d463 bge             x13, x11, 8         x13:0001d973 x11:00000000
3826ns   378 000002a0 00470713 addi            x14, x14, 4         x14=00030028 x14:00030024
3836ns   379 000002a4 fec716e3 bne             x14, x12, -20       x14:00030028 x12:00030028
3846ns   380 000002a8 00000513 addi            x10, x0, 0          x10=00000000
3856ns   381 000002ac 00008067 jalr            x0, x1, 0            x1:000001d8
```

Listing 5.6: Extract of simulation result of *max_min.c* - old_ISA

```
Time     Cycles PC    Instr    Mnemonic

2086ns   204 00000254 0007a023 sw              x0, 0(x15)          x15:00030000  PA:00030000
2096ns   205 00000258 0007e69b lw_max          N10 x13 0(x15)      x13=0001d973 x15:00030000  PA:00030000
2436ns   239 0000025c 0ed7a823 sw              x13, 240(x15)       x13:0001d973 x15:00030000  PA:000300f0
2446ns   240 00000260 0007d59b lw_min          N10 x11 0(x15)      x11=00000000 x15:00030000  PA:00030000
2786ns   274 00000264 0eb7aa23 sw              x11, 244(x15)       x11:00000000 x15:00030000  PA:000300f4
2796ns   275 00000268 00000513 addi            x10, x0, 0          x10=00000000
2806ns   276 0000026c 00008067 jalr            x0, x1, 0           x1:000001d8
```

Listing 5.7: Extract of simulation result of *max_min.c* - newIF_ISA

```
Time     Cycles PC    Instr    Mnemonic

2086ns   204 00000254 0007a023 sw              x0, 0(x15)          x15:00030000  PA:00030000
2096ns   205 00000258 00020637 lui             x12, 0x20000        x12=00020000
2106ns   206 0000025c 00a00593 addi            x11, x0, 10         x11=0000000a
2116ns   207 00000260 ffc665bb sw_active_max   Nx11 -4(x12)       x12:00020000 x11:0000000a  PA:0001fffc
2126ns   208 00000264 0007069b lw_mask         x13, x0, 0(x14)     x13=0001d973 x14:00030000  PA:00030000
2136ns   209 00000268 ffc655bb sw_active_min   Nx11 -4(x12)       x12:00020000 x11:0000000a  PA:0001fffc
2466ns   242 0000026c 0007059b lw_mask         x11, x0, 0(x14)     x11=00000000 x14:00030000  PA:00030000
2476ns   243 00000270 ffc6003b sw_active_none  Nx0 -4(x12)        x12:00020000  PA:0001fffc
2806ns   276 00000274 0ed7a823 sw              x13, 240(x15)       x13:0001d973 x15:00030000  PA:000300f0
2816ns   277 00000278 0eb7aa23 sw              x11, 244(x15)       x11:00000000 x15:00030000  PA:000300f4
2826ns   278 0000027c 00000513 addi            x10, x0, 0          x10=00000000
2836ns   279 00000280 00008067 jalr            x0, x1, 0           x1:000001d8
```

Listing 5.8: Extract of simulation result of *max_min.c* - sameIF_ISA

# 3 Simulation with standard programs

The following sections instead are intended to show the usage of the LiM in real world algorithms, so called standard algorithms.
In order to fully exploit the advantages of the Logic-in-Memory, the selection of the algorithms has been done according to the following criteria:

- *High data demand.* Applications with high memory content are suitable because the LiM can reduce the data movement to/from memory.
- *Data manipulation with supported LiM operations.* The implemented LiM can perform a limited amount of operations without the need of the microprocessor. To exploit its functionality it is needed to select algorithms that would use these operations.
- *Simple algorithms.* Not too complex algorithms are needed at this stage because the compiler does not support the new LiM ISAs. Simple algorithms avoid the risk to introduce errors during the partial manual compilation.

## 3.1 Database search with Bitmap Indexes algorithm

A bitmap index [11] is a special kind of data structure that uses bitmaps to speed-up the processing of stored data. In fact, it is usually used for search operations in large data warehouses.
The basic idea is to use bits (0 or 1) to indicate whether a feature is satisfied or not. The index is mapped with 1 if the feature is satisfied, or with 0 if not.
For example, considering the Table 5.1, the first column contains the student ID (the index) and the remaining ones the age brackets (the features) that can be satisfied or not. Bitmap indexes can efficiently process any queries, using bitwise operations [12]. The search algorithm is therefore reduced to simple bitwise operations, resulting extremely fast. This is the reason why this kind of algorithms are really used in real world applications.

| Student ID | Age 17 | Age 18 | Age 19 |
|---|---|---|---|
| 00 | 1 | 0 | 0 |
| 01 | 0 | 1 | 0 |
| 10 | 0 | 0 | 1 |
| 11 | 1 | 0 | 0 |

Table 5.1: Bitmap indexes example: students age ranges mapped with bits

According to the students example, a possible query can be: Which students are over 18? The answer is showed in Figure 5.3.

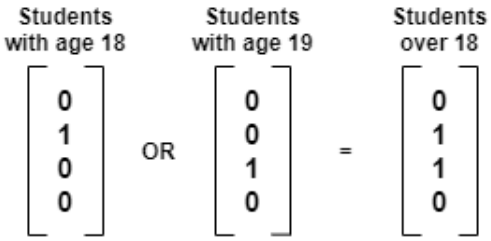



Figure 5.3: Bitmap indexes example: query result

### 3.1.1 C program

The Logic-in-Memory could further speed-up the data search algorithm, performing some bitwise operations in memory.
The C program developed for this thesis is very similar to the example showed above. The dataset includes a set of features on some high school students. Features considered are gender and range ages, while the queries are:

1. Which students are male and older than 19?
2. Which students are older than 18?

Note that 160 students are considered in the dataset. The features associated to the students are distributed over 5 integer vectors, because Logic-in-Memory operations can only be performed on 32-bit data.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char* argv[])
5  {
6      /* variable declaration */
7      int i;
8      int *result_M_over19=0x300B0;
9      int *result_over18  =0x300D0;
10
11     /* Initialize bitmap */
12     int *v_age16   = 0x30000;
13     int *v_age17   = 0x30018;
14     int *v_age18   = 0x30030;
15     int *v_age19   = 0x30048;
16     int *v_age20   = 0x30060;
17     int *v_genderM = 0x30078;
18     int *v_genderF = 0x30090;
19
20     v_genderM[0]=0x00000000; v_genderM[1]=0x00000000; v_genderM[2]=0x00000000; v_genderM[3]=0xFFFFFFFF; v_genderM[4]=0
         xFFFFFFFF; v_genderM[5]=0xFFFFFFFF;
21     v_genderF[0]=0xFFFFFFFF; v_genderF[1]=0xFFFFFFFF; v_genderF[2]=0xFFFFFFFF; v_genderF[3]=0x00000000; v_genderF[4]=0
         x00000000; v_genderF[5]=0x00000000;
22     v_age16[0]  =0x00000000; v_age16[1]  =0x00000000; v_age16[2]  =0x0000FFFF; v_age16[3]  =0x00000000; v_age16[4]  =0
         x00000000; v_age16[5]  =0x0000FFFF;
23     v_age17[0]  =0x00000000; v_age17[1]  =0x00000000; v_age17[2]  =0xFFFF0000; v_age17[3]  =0x00000000; v_age17[4]  =0
         x00000000; v_age17[5]  =0xFFFF0000;
24     v_age18[0]  =0x00000000; v_age18[1]  =0x0000FFFF; v_age18[2]  =0x00000000; v_age18[3]  =0x00000000; v_age18[4]  =0
         x0000FFFF; v_age18[5]  =0x00000000;
25     v_age19[0]  =0x00000000; v_age19[1]  =0xFFFF0000; v_age19[2]  =0x00000000; v_age19[3]  =0x00000000; v_age19[4]  =0
         xFFFF0000; v_age19[5]  =0x00000000;
26     v_age20[0]  =0xFFFFFFFF; v_age20[1]  =0x00000000; v_age20[2]  =0x00000000; v_age20[3]  =0xFFFFFFFF; v_age20[4]  =0
         x00000000; v_age20[5]  =0x00000000;
```

```
27
28      /* Initialise results to 0 */
29      for(i=0; i<6; i++) {
30          result_M_over19[i] = 0;
31          result_over18[i]   = 0;
32      }
33
34      /* Query: identify male people that are 19 or 20 */
35      for(i=0; i<6; i++) {
36          result_M_over19[i] = v_genderM[i] & (v_age19[i] | v_age20[i]);
37      }
38
39      /* Query: identify people that are older than 18 */
40      for(i=0; i<6; i++) {
41          result_over18[i] = ~v_age16[i] & ~v_age17[i] ;
42      }
43
44      return EXIT_SUCCESS;
45  }
```

Listing 5.9: Standard program *bitmap_search.c*

### 3.1.2 Execution time estimation

The approximate time estimation consider $N = \frac{N_{indexes}}{32}$, because 32 corresponds to the parallelism of data memory operations.

$$Execution\_time_{old\_ISA}(bitmap\_search.c) \approx 6N(result\_M\_over19) + 5N(result\_over18) \tag{5.7}$$

$$Execution\_time_{newIF\_ISA}(bitmap\_search.c) \approx 4N(result\_M\_over19) + 4N(result\_over18) \tag{5.8}$$

$$Execution\_time_{sameIF\_ISA}(bitmap\_search.c) \approx 1(mem\_active) + 5N(result\_M\_over19) + 4N(result\_over18) + 1(mem\_active) \tag{5.9}$$

In case of the old_ISA, the execution time for the first query considers a factor equal to 6, because of the need to load three different memory variables, perform two bitwise operations and then store the result in the required memory location. The same query can be computed for newIF_ISA by loading the first variable, performing a load-OR for the second variable, then proceeding with a load-AND for the third variable and finally store the result in memory. The second query is computed similarly taking into account the compiler will recognise the optimisation $(\sim A)\&(\sim B) = \sim (A|B)$, according to the De Morgan's Law.
For this application a special note is needed for the same_ISA system. As a matter of fact, this system needs to assume that the memory location where to store the queries results needs to be initialised to zero. The reason behind this assumption is that, to keep the efficiency of the Logic-in-Memory instructions the memory should be set on one memory operation for the entire algorithm execution. In this case, the most convenient logic operation to choose is the OR operation. Therefore, assuming the results memory locations initialised to zero would avoid to change the memory configuration, because the store-OR operation between a value and zero corresponds to a normal store operation.
For this reason, in the case of sameIF_ISA the execution time will include the activation and deactivation of the OR operation in memory, all the neutral operations will be performed using x0 as the input mask. The first query will require a load of the first variable, then a load OR of the second variable using the first variable as input mask, afterwards the third variable is loaded and a normal AND is performed. The result is finally stored in memory. The second query instead, requires the loading of the first variable, the load-OR of the second variable, the NOT operation on the partial result and the final result can be stored in memory.

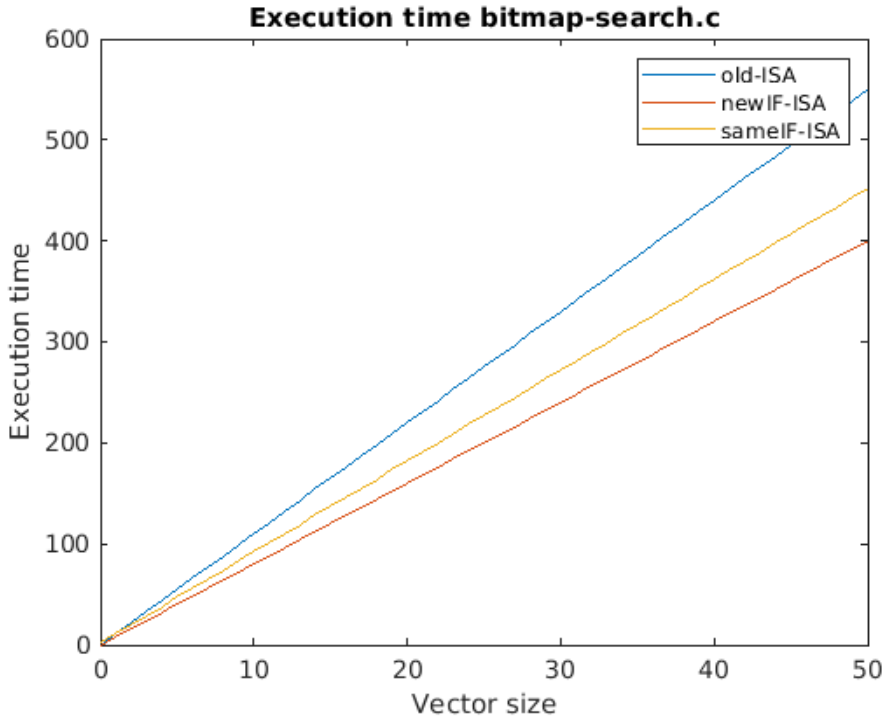



Figure 5.4: Estimation execution time bitmap_search.c in old_ISA, newIF_ISA and sameIF_ISA

### 3.1.3 Simulation results

The simulation shows a really slight improvement for $N = 5$ ($N_{indexes} = 160$). The improvement is not much as expected because of the stall of the RISC-V pipeline when performing an operation that needs the result of a load operation. The stall is caused because of the forwarding mechanism not possible with a data read from memory. The instructions saved with the Logic-in-Memory operations are partially wasted with the introduction of these stalls, resulting in a smaller improvement than expected.

```
Time     Cycles PC    Instr    Mnemonic

2276ns   223 000002a0 09078793 addi            x15, x15, 144       x15=00030090 x15:00030000
2286ns   224 000002a4 fe872603 lw              x12, -24(x14)       x12=ffffffff x14:00030078  PA:00030060
2296ns   225 000002a8 fd072683 lw              x13, -48(x14)       x13=00000000 x14:00030078  PA:00030048
2306ns   226 000002ac 00470713 addi            x14, x14, 4         x14=0003007c x14:00030078
2316ns   227 000002b0 00c6e6b3 or              x13, x13, x12       x13=ffffffff x13:00000000 x12:ffffffff
2326ns   228 000002b4 ffc72603 lw              x12, -4(x14)        x12=00000000 x14:0003007c  PA:00030078
2346ns   230 000002b8 00c6f6b3 and             x13, x13, x12       x13=00000000 x13:ffffffff x12:00000000
2356ns   231 000002bc 02d72a23 sw              x13, 52(x14)        x13:00000000 x14:0003007c  PA:000300b0
2366ns   232 000002c0 fef712e3 bne             x14, x15, -28       x14:0003007c x15:00030090

2836ns   279 000002a4 fe872603 lw              x12, -24(x14)       x12=00000000 x14:0003008c  PA:00030074
2846ns   280 000002a8 fd072683 lw              x13, -48(x14)       x13=00000000 x14:0003008c  PA:0003005c
2856ns   281 000002ac 00470713 addi            x14, x14, 4         x14=00030090 x14:0003008c
2866ns   282 000002b0 00c6e6b3 or              x13, x13, x12       x13=00000000 x13:00000000 x12:00000000
2876ns   283 000002b4 ffc72603 lw              x12, -4(x14)        x12=ffffffff x14:00030090  PA:0003008c
2896ns   285 000002b8 00c6f6b3 and             x13, x13, x12       x13=00000000 x13:00000000 x12:ffffffff
2906ns   286 000002bc 02d72a23 sw              x13, 52(x14)        x13:00000000 x14:00030090  PA:000300c4
2916ns   287 000002c0 fef712e3 bne             x14, x15, -28       x14:00030090 x15:00030090
2926ns   288 000002c4 000307b7 lui             x15, 0x30000        x15=00030000
2936ns   289 000002c8 01878693 addi            x13, x15, 24        x13=00030018 x15:00030000
2946ns   290 000002cc 0187a703 lw              x14, 24(x15)        x14=00000000 x15:00030000  PA:00030018
2956ns   291 000002d0 0007a603 lw              x12, 0(x15)         x12=00000000 x15:00030000  PA:00030000
2966ns   292 000002d4 00478793 addi            x15, x15, 4         x15=00030004 x15:00030000
2976ns   293 000002d8 00c76733 or              x14, x14, x12       x14=00000000 x14:00000000 x12:00000000
2986ns   294 000002dc fff74713 xori            x14, x14, -1        x14=ffffffff x14:00000000
2996ns   295 000002e0 0ce7a623 sw              x14, 204(x15)       x14:ffffffff x15:00030004  PA:000300d0
3006ns   296 000002e4 fed794e3 bne             x15, x13, -24       x15:00030004 x13:00030018

3396ns   335 000002cc 0187a703 lw              x14, 24(x15)        x14=ffff0000 x15:00030014  PA:0003002c
3406ns   336 000002d0 0007a603 lw              x12, 0(x15)         x12=0000ffff x15:00030014  PA:00030014
3416ns   337 000002d4 00478793 addi            x15, x15, 4         x15=00030018 x15:00030014
3426ns   338 000002d8 00c76733 or              x14, x14, x12       x14=ffffffff x14:ffff0000 x12:0000ffff
3436ns   339 000002dc fff74713 xori            x14, x14, -1        x14=00000000 x14:ffffffff
3446ns   340 000002e0 0ce7a623 sw              x14, 204(x15)       x14:00000000 x15:00030018  PA:000300e4
```

```
3456ns   341 000002e4 fed794e3 bne             x15, x13, -24       x15:00030018 x13:00030018
3466ns   342 000002e8 00000513 addi            x10, x0, 0          x10=00000000
3476ns   343 000002ec 00008067 jalr            x0, x1, 0            x1:000001d8
```

Listing 5.10: Extract of simulation result of *bitmap_search.c* - old_ISA

```
Time     Cycles PC    Instr    Mnemonic

2276ns   223 000002a0 09078793 addi            x15, x15, 144       x15=00030090 x15:00030000
2286ns   224 000002a4 fd072683 lw              x13, -48(x14)       x13=00000000 x14:00030078  PA:00030048
2306ns   226 000002a8 d0d7361b lw_or           x12, x13, -24(x14)  x12=ffffffff x13:00000000 x14:00030078  PA:00030060
2316ns   227 000002ac 00470713 addi            x14, x14, 4         x14=0003007c x14:00030078
2326ns   228 000002b0 f8c7269b lw_and          x13, x12, -4(x14)   x13=00000000 x12:ffffffff x14:0003007c  PA:00030078
2346ns   230 000002b4 02d72a23 sw              x13, 52(x14)        x13:00000000 x14:0003007c  PA:000300b0
2356ns   231 000002b8 fef716e3 bne             x14, x15, -20       x14:0003007c x15:00030090

2786ns   274 000002a4 fd072683 lw              x13, -48(x14)       x13=00000000 x14:0003008c  PA:0003005c
2806ns   276 000002a8 d0d7361b lw_or           x12, x13, -24(x14)  x12=00000000 x13:00000000 x14:0003008c  PA:00030074
2816ns   277 000002ac 00470713 addi            x14, x14, 4         x14=00030090 x14:0003008c
2826ns   278 000002b0 f8c7269b lw_and          x13, x12, -4(x14)   x13=00000000 x12:00000000 x14:00030090  PA:0003008c
2846ns   280 000002b4 02d72a23 sw              x13, 52(x14)        x13:00000000 x14:00030090  PA:000300c4
2856ns   281 000002b8 fef716e3 bne             x14, x15, -20       x14:00030090 x15:00030090
2866ns   282 000002bc 000307b7 lui             x15, 0x30000        x15=00030000
2876ns   283 000002c0 01878693 addi            x13, x15, 24        x13=00030018 x15:00030000
2886ns   284 000002c4 0187a603 lw              x12, 24(x15)        x12=00000000 x15:00030000  PA:00030018
2906ns   286 000002c8 00c7b71b lw_or           x14, x12, 0(x15)    x14=00000000 x12:00000000 x15:00030000  PA:00030000
2916ns   287 000002cc 00478793 addi            x15, x15, 4         x15=00030004 x15:00030000
2926ns   288 000002d0 fff74713 xori            x14, x14, -1        x14=ffffffff x14:00000000
2936ns   289 000002d4 0ce7a623 sw              x14, 204(x15)       x14:ffffffff x15:00030004  PA:000300d0
2946ns   290 000002d8 fed796e3 bne             x15, x13, -20       x15:00030004 x13:00030018

3336ns   329 000002c4 0187a603 lw              x12, 24(x15)        x12=ffff0000 x15:00030014  PA:0003002c
3356ns   331 000002c8 00c7b71b lw_or           x14, x12, 0(x15)    x14=ffffffff x12:ffff0000 x15:00030014  PA:00030014
3366ns   332 000002cc 00478793 addi            x15, x15, 4         x15=00030018 x15:00030014
3376ns   333 000002d0 fff74713 xori            x14, x14, -1        x14=00000000 x14:ffffffff
3386ns   334 000002d4 0ce7a623 sw              x14, 204(x15)       x14:00000000 x15:00030018  PA:000300e4
3396ns   335 000002d8 fed796e3 bne             x15, x13, -20       x15:00030018 x13:00030018
3406ns   336 000002dc 00000513 addi            x10, x0, 0          x10=00000000
3416ns   337 000002e0 00008067 jalr            x0, x1, 0            x1:000001d8
```

Listing 5.11: Extract of simulation result of *bitmap_search.c* - newIF_ISA

```
Time     Cycles PC    Instr    Mnemonic

2396ns   235 000002d0 09078793 addi            x15, x15, 144       x15=00030090 x15:00030000
2406ns   236 000002d4 00020837 lui             x16, 0x20000        x16=00020000
2416ns   237 000002d8 ffc8303b sw_active_or    Nx0 -4(x16)         x16:00020000  PA:0001fffc
2426ns   238 000002dc a007261b lw_mask         x12, x0, -48(x14)   x12=00000000 x14:00030078  PA:00030048
2436ns   239 000002e0 00470713 addi            x14, x14, 4         x14=0003007c x14:00030078
2446ns   240 000002e4 c8c7269b lw_mask         x13, x12, -28(x14)  x13=ffffffff x12:00000000 x14:0003007c  PA:00030060
2456ns   241 000002e8 f807261b lw_mask         x12, x0, -4(x14)    x12=00000000 x14:0003007c  PA:00030078
2476ns   243 000002ec 00c6f6b3 and             x13, x13, x12       x13=00000000 x13:ffffffff x12:00000000
2486ns   244 000002f0 02d72a23 sw              x13, 52(x14)        x13:00000000 x14:0003007c  PA:000300b0
2496ns   245 000002f4 fef714e3 bne             x14, x15, -24       x14:0003007c x15:00030090

2926ns   288 000002dc a007261b lw_mask         x12, x0, -48(x14)   x12=00000000 x14:0003008c  PA:0003005c
2936ns   289 000002e0 00470713 addi            x14, x14, 4         x14=00030090 x14:0003008c
2946ns   290 000002e4 c8c7269b lw_mask         x13, x12, -28(x14)  x13=00000000 x12:00000000 x14:00030090  PA:00030074
2956ns   291 000002e8 f807261b lw_mask         x12, x0, -4(x14)    x12=ffffffff x14:00030090  PA:0003008c
2976ns   293 000002ec 00c6f6b3 and             x13, x13, x12       x13=00000000 x13:00000000 x12:ffffffff
2986ns   294 000002f0 02d72a23 sw              x13, 52(x14)        x13:00000000 x14:00030090  PA:000300c4
2996ns   295 000002f4 fef714e3 bne             x14, x15, -24       x14:00030090 x15:00030090
3006ns   296 000002f8 000307b7 lui             x15, 0x30000        x15=00030000
3016ns   297 000002fc 01878693 addi            x13, x15, 24        x13=00030018 x15:00030000
3026ns   298 00000300 3007a71b lw_mask         x14, x0, 24(x15)    x14=00000000 x15:00030000  PA:00030018
3036ns   299 00000304 00478793 addi            x15, x15, 4         x15=00030004 x15:00030000
3046ns   300 00000308 f8e7a61b lw_mask         x12, x14, -4(x15)   x12=00000000 x14:00000000 x15:00030004  PA:00030000
3066ns   302 0000030c fff64613 xori            x12, x12, -1        x12=ffffffff x12:00000000
3076ns   303 00000310 0cc7a623 sw              x12, 204(x15)       x12:ffffffff x15:00030004  PA:000300d0
3086ns   304 00000314 fed796e3 bne             x15, x13, -20       x15:00030004 x13:00030018

3476ns  343 00000300 3007a71b lw_mask          x14, x0, 24(x15)    x14=ffff0000 x15:00030014  PA:0003002c
3486ns  344 00000304 00478793 addi             x15, x15, 4         x15=00030018 x15:00030014
3496ns  345 00000308 f8e7a61b lw_mask          x12, x14, -4(x15)   x12=ffffffff x14:ffff0000 x15:00030018  PA:00030014
3516ns  347 0000030c fff64613 xori             x12, x12, -1        x12=00000000 x12:ffffffff
3526ns  348 00000310 0cc7a623 sw               x12, 204(x15)       x12:00000000 x15:00030018  PA:000300e4
3536ns  349 00000314 fed796e3 bne              x15, x13, -20       x15:00030018 x13:00030018
3546ns  350 00000318 ffc8003b sw_active_none   Nx0 -4(x16)         x16:00020000  PA:0001fffc
3556ns  351 0000031c 00000513 addi             x10, x0, 0          x10=00000000
3566ns  352 00000320 00008067 jalr             x0, x1, 0            x1:000001d8
```

Listing 5.12: Extract of simulation result of *bitmap_search.c* - same_ISA

## 3.2 AES Addroundkey algorithm

The Advanced Encryption Standard (AES) plays an important role in the security of data transmission [13] [14] [15]. The algorithm has been already implemented in hardware or

software in many different ways, because of its broad range of applications.
The cryptography algorithm encrypts chunks of 128-bit data (also called plaintext) organised in a 4x4 matrix defined as *states*. In the AES-128 case, the data is transformed by means of a 128-bit *key* also organised in a 4x4 matrix. Figure 5.5 describes at a very high level the 10 rounds that perform the data encryption and all the steps needed in each round.

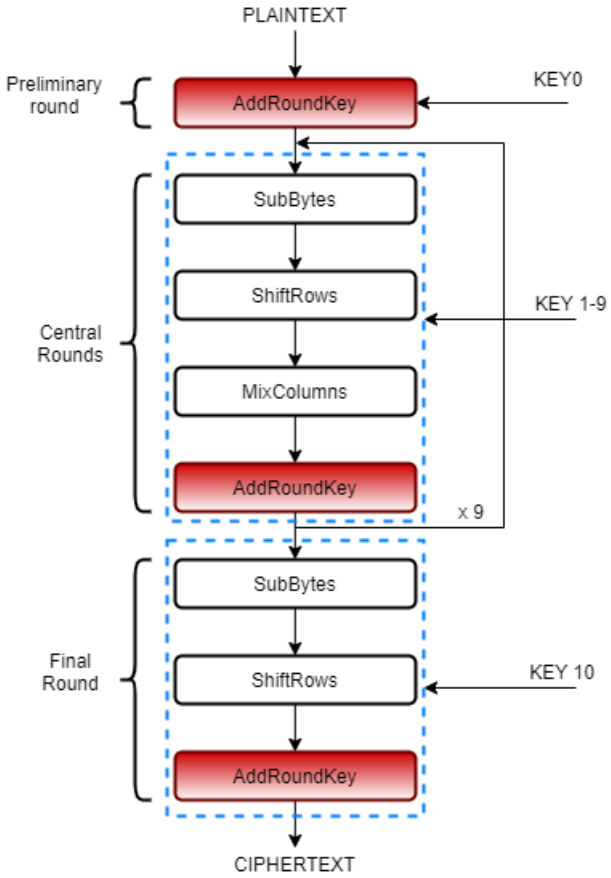



Figure 5.5: AES encryption algorithm

The work done in this thesis for the AES, aims to speed-up the software implementation of the *AddRoundKey* step, that is performed 11 times in the whole encryption process.
The *AddRoundKey* consists in XOR operations between the bytes of the states matrix and the key matrix, element by element. The result is the next states matrix, which is used for the subsequent steps that lead to the encryption.

### 3.2.1 C program

The C code implemented, initialises the states and the key into predefined memory locations. The execution of the algorithm itself is very straightforward.

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

int main(int argc, char* argv[])
{
    /* input variables declaration */
    int (*states)[4][4] = 0x30000;
    (*states)[0][0]=0x32; (*states)[0][1]=0x88; (*states)[0][2]=0x31; (*states)[0][3]=0xE0;
    (*states)[1][0]=0x43; (*states)[1][1]=0x54; (*states)[1][2]=0x31; (*states)[1][3]=0x37;
    (*states)[2][0]=0xF6; (*states)[2][1]=0x30; (*states)[2][2]=0x98; (*states)[2][3]=0x07;
    (*states)[3][0]=0xA8; (*states)[3][1]=0x8D; (*states)[3][2]=0xA2; (*states)[3][3]=0x34;

    int (*key)[4][4] = 0x30200;
    (*key)[0][0]=0x00; (*key)[0][1]=0xA5; (*key)[0][2]=0xA8; (*key)[0][3]=0xA0;
    (*key)[1][0]=0xE9; (*key)[1][1]=0x09; (*key)[1][2]=0xBB; (*key)[1][3]=0x2A;
```

```
16      (*key)[2][0]=0xC9; (*key)[2][1]=0xD4; (*key)[2][2]=0xB7; (*key)[2][3]=0xAB;
17      (*key)[3][0]=0xF2; (*key)[3][1]=0xE8; (*key)[3][2]=0x60; (*key)[3][3]=0x08;
18
19      /* Others */
20      int i, j;
21
22      /* Add around key */
23      for (i=0; i<4; i++) {
24          for (j=0; j<4; j++) {
25              (*states)[i][j] = (*states)[i][j] ^ (*key)[i][j];
26          }
27      }
28
29      return EXIT_SUCCESS;
30  }
```

Listing 5.13: Standard program *aes128_addroundkey.c*

### 3.2.2 Execution time estimation

The estimation of the number of clock cycles needed by the *addroundkey.c* program does not depend on any variables, because the size of the input data is fixed. In fact, the number of elements N is 16.

$$Execution\_time_{old\_ISA}(aes128\_addroundkey.c) \approx 4N = 64cc \tag{5.10}$$

$$Execution\_time_{newIF\_ISA}(aes128\_addroundkey.c) \approx 2N = 32cc \tag{5.11}$$

$$Execution\_time_{sameIF\_ISA}(aes128\_addroundkey.c) \approx 1(mem\_active) + 2N + 1(mem\_active) = 34cc \tag{5.12}$$

For each element of the array, the operations to consider with the old_ISA are, the load of the states and key elements, the XOR operation between them and the store back in memory of the result. The newIF_ISA saves two clock cycles per each element, because the operation is performed directly in memory. As usual, the sameIF_ISA introduces two additional clock cycles because of the operations needed for the memory configurations.

### 3.2.3 Simulation results

Simulation results show an important improvement in terms of execution time in both cases. Considering that in the full AES algorithm, this piece of program would be executed 11 times for each 128-bit chunk of input data, the overall speed-up introduced by the LiM is remarkable.

```
Time     Cycles PC    Instr    Mnemonic

2516ns   247 000002e0 000308b7 lui             x17, 0x30000        x17=00030000
2526ns   248 000002e4 00400313 addi            x6, x0, 4            x6=00000004
2536ns   249 000002e8 00000693 addi            x13, x0, 0          x13=00000000
2546ns   250 000002ec 00261813 slli            x16, x12, 0x2       x16=00000000 x12:00000000
2556ns   251 000002f0 00d80733 add             x14, x16, x13       x14=00000000 x16:00000000 x13:00000000
2566ns   252 000002f4 00271713 slli            x14, x14, 0x2       x14=00000000 x14:00000000
2576ns   253 000002f8 00e88533 add             x10, x17, x14       x10=00030000 x17:00030000 x14:00000000
2586ns   254 000002fc 00e78733 add             x14, x15, x14       x14=00030200 x15:00030200 x14:00000000
2596ns   255 00000300 00052583 lw              x11, 0(x10)         x11=00000032 x10:00030000  PA:00030000
2606ns   256 00000304 00072703 lw              x14, 0(x14)         x14=00000000 x14:00030200  PA:00030200
2616ns   257 00000308 00168693 addi            x13, x13, 1         x13=00000001 x13:00000000
2626ns   258 0000030c 00e5c733 xor             x14, x11, x14       x14=00000032 x11:00000032 x14:00000000
2636ns   259 00000310 00e52023 sw              x14, 0(x10)         x14:00000032 x10:00030000  PA:00030000
2646ns   260 00000314 fc669ee3 bne             x13, x6, -36        x13:00000001  x6:00000004

4516ns   447 00000300 00052583 lw              x11, 0(x10)         x11=00000034 x10:0003003c  PA:0003003c
4526ns   448 00000304 00072703 lw              x14, 0(x14)         x14=00000008 x14:0003023c  PA:0003023c
4536ns   449 00000308 00168693 addi            x13, x13, 1         x13=00000004 x13:00000003
4546ns   450 0000030c 00e5c733 xor             x14, x11, x14       x14=0000003c x11:00000034 x14:00000008
4556ns   451 00000310 00e52023 sw              x14, 0(x10)         x14:0000003c x10:0003003c  PA:0003003c
4566ns   452 00000314 fc669ee3 bne             x13, x6, -36        x13:00000004  x6:00000004
4576ns   453 00000318 00160613 addi            x12, x12, 1         x12=00000004 x12:00000003
4586ns   454 0000031c fcd616e3 bne             x12, x13, -52       x12:00000004 x13:00000004
4596ns   455 00000320 00000513 addi            x10, x0, 0          x10=00000000
4606ns   456 00000324 00008067 jalr            x0, x1, 0            x1:000001d8
```

Listing 5.14: Extract of simulation result of *aes128_addroundkey.c* - old_ISA

```
Time     Cycles PC    Instr    Mnemonic

2516ns   247 000002e0 000308b7 lui             x17, 0x30000        x17=00030000
2526ns   248 000002e4 00400313 addi            x6, x0, 4            x6=00000004
2536ns   249 000002e8 00000693 addi            x13, x0, 0          x13=00000000
2546ns   250 000002ec 00261813 slli            x16, x12, 0x2       x16=00000000 x12:00000000
2556ns   251 000002f0 00d80733 add             x14, x16, x13       x14=00000000 x16:00000000 x13:00000000
2566ns   252 000002f4 00271713 slli            x14, x14, 0x2       x14=00000000 x14:00000000
2576ns   253 000002f8 00e88533 add             x10, x17, x14       x10=00030000 x17:00030000 x14:00000000
2586ns   254 000002fc 00e78733 add             x14, x15, x14       x14=00030200 x15:00030200 x14:00000000
2596ns   255 00000300 00072703 lw              x14, 0(x14)         x14=00000000 x14:00030200  PA:00030200
2606ns   256 00000304 00168693 addi            x13, x13, 1         x13=00000001 x13:00000000
2616ns   257 00000308 00e5103b sw_xor          N-x0 x14, 0(x10)    x14:00000000 x10:00030000  PA:00030000
2626ns   258 0000030c fe6692e3 bne             x13, x6, -28        x13:00000001  x6:00000004

4176ns  413 000002f0 00d80733 add             x14, x16, x13       x14=0000000f x16:0000000c x13:00000003
4186ns  414 000002f4 00271713 slli            x14, x14, 0x2       x14=0000003c x14:0000000f
4196ns  415 000002f8 00e88533 add             x10, x17, x14       x10=0003003c x17:00030000 x14:0000003c
4206ns  416 000002fc 00e78733 add             x14, x15, x14       x14=0003023c x15:00030200 x14:0000003c
4216ns  417 00000300 00072703 lw              x14, 0(x14)         x14=00000008 x14:0003023c  PA:0003023c
4226ns  418 00000304 00168693 addi            x13, x13, 1         x13=00000004 x13:00000003
4236ns  419 00000308 00e5103b sw_xor          N-x0 x14, 0(x10)    x14:00000008 x10:0003003c  PA:0003003c
4246ns  420 0000030c fe6692e3 bne             x13, x6, -28        x13:00000004  x6:00000004
4256ns  421 00000310 00160613 addi            x12, x12, 1         x12=00000004 x12:00000003
4266ns  422 00000314 fcd61ae3 bne             x12, x13, -44       x12:00000004 x13:00000004
4276ns  423 00000318 00000513 addi            x10, x0, 0          x10=00000000
4286ns  424 0000031c 00008067 jalr            x0, x1, 0            x1:000001d8
```

Listing 5.15: Extract of simulation result of *aes128_addroundkey.c* - new_ISA

```
Time     Cycles PC    Instr    Mnemonic

2516ns   247 000002e0 000308b7 lui             x17, 0x30000        x17=00030000
2526ns   248 000002e4 00400313 addi            x6, x0, 4            x6=00000004
2536ns   249 000002e8 000204b7 lui             x9, 0x20000          x9=00020000
2546ns   250 000002ec ffc4903b sw_active_xor   Nx0 -4(x9)           x9:00020000  PA:0001fffc
2556ns   251 000002f0 00000693 addi            x13, x0, 0          x13=00000000
2566ns   252 000002f4 00261813 slli            x16, x12, 0x2       x16=00000000 x12:00000000
2576ns   253 000002f8 00d80733 add             x14, x16, x13       x14=00000000 x16:00000000 x13:00000000
2586ns   254 000002fc 00271713 slli            x14, x14, 0x2       x14=00000000 x14:00000000
2596ns   255 00000300 00e88533 add             x10, x17, x14       x10=00030000 x17:00030000 x14:00000000
2606ns   256 00000304 00e78733 add             x14, x15, x14       x14=00030200 x15:00030200 x14:00000000
2616ns   257 00000308 0007271b lw_mask         x14, x0, 0(x14)     x14=00000000 x14:00030200  PA:00030200
2626ns   258 0000030c 00168693 addi            x13, x13, 1         x13=00000001 x13:00000000
2636ns   259 00000310 00e52023 sw              x14, 0(x10)         x14:00000000 x10:00030000  PA:00030000
2646ns   260 00000314 fe6692e3 bne             x13, x6, -28        x13:00000001  x6:00000004

4196ns   415 000002f8 00d80733 add             x14, x16, x13       x14=0000000f x16:0000000c x13:00000003
4206ns   416 000002fc 00271713 slli            x14, x14, 0x2       x14=0000003c x14:0000000f
4216ns   417 00000300 00e88533 add             x10, x17, x14       x10=0003003c x17:00030000 x14:0000003c
4226ns   418 00000304 00e78733 add             x14, x15, x14       x14=0003023c x15:00030200 x14:0000003c
4236ns   419 00000308 0007271b lw_mask         x14, x0, 0(x14)     x14=00000008 x14:0003023c  PA:0003023c
4246ns   420 0000030c 00168693 addi            x13, x13, 1         x13=00000004 x13:00000003
4256ns   421 00000310 00e52023 sw              x14, 0(x10)         x14:00000008 x10:0003003c  PA:0003003c
4266ns   422 00000314 fe6692e3 bne             x13, x6, -28        x13:00000004  x6:00000004
4276ns   423 00000318 00160613 addi            x12, x12, 1         x12=00000004 x12:00000003
4286ns   424 0000031c fcd61ae3 bne             x12, x13, -44       x12:00000004 x13:00000004
4296ns   425 00000320 ffc4803b sw_active_none  Nx0 -4(x9)           x9:00020000  PA:0001fffc
4306ns   426 00000324 00000513 addi            x10, x0, 0          x10=00000000
```

Listing 5.16: Extract of simulation result of *aes128_addroundkey.c* - same_ISA

## 3.3 Transport problem - Least Cost Method algorithm

The transport problem is an algorithm that minimises the cost of distributing a product from a number of sources or origins to a number of destinations [16]. Applications of this algorithm can be found in the logistic and shipment activities, where the aim is to minimise the cost of the shipments.
The origin of a transport problem is the location from which shipments start, while the destination of a transport problem is the location to which shipments are transported. The unit transport cost is the shipment cost related to a unique path between an origin and a destination.
Assuming to organise the transport problem data in a matrix, the matrix would follow the this structure:

- In the rows there are the M suppliers ($s_1$, $s_2$, ..., $s_M$).
- In the columns there are the M buyers ($b_1$,$b_2$, ..., $b_M$).

- In each cell of the matrix, are the associated transport cost from the supplier to the buyer. For instance the cost of transport from the supplier 1 to the buyer 1 will be $c_{11}$, while the cost to transport from supplier 1 t buyer 2 will be $c_{12}$ and so on.
- Two additional variables are needed: the supply capacity and buying demand for each of the suppliers and buyers.

Therefore, as shown in the Table 5.2 , it is possible to see all the related transport costs that will influence the final distribution. In fact, this problem aims at minimize the transport cost while meeting the various supply and demand. For the sake of simplicity the cost matrix and the supply and demand vectors are represented in the same table.

| **Supplier\Buyer** | $b_1$ | $b_2$ | $b_3$ | **Supply** |
|---|---|---|---|---|
| $s_1$ | 5 | 8 | 4 | 50 |
| $s_2$ | 6 | 6 | 3 | 40 |
| $s_3$ | 3 | 9 | 6 | 60 |
| **Demand** | 20 | 95 | 35 | |

Table 5.2: Transport problem example

The algorithm that computes the total minimum cost, works in the following way:

1. Find the minimum cost in the cost matrix.
2. In case of a tie, determine the path $s_i$-$b_j$ giving the priority to the path with the maximum demand.
3. Update the demand $demand_j$ of the buyer $b_j$ and the capacity $supply_i$ of the supplier $s_i$. Both quantities should subtract the $transfer$ amount, that in order to minimise the cost should be the maximum number of items that the supplier can give to the buyer, according to its demand.
4. Update the total cost: $Total\_cost = Total\_cost + transfer * c_{ij}$.
5. Repeat points 1-4 for all the elements of the cost matrix, then for all the possible paths, excluding already considered cost elements. At the end of the last iteration $Total\_cost$ contains the minimum cost for the overall transports.

### 3.3.1 C program

The C program used for the simulation reflects the example showed in 5.2. This algorithm has been chosen because of the minimum function that the LiM could perform on the cost matrix, for each iteration. Moreover, in order to avoid the evaluation of already considered costs, a mask is applied to the considered cost element. The masking operation can be performed by the LiM as well.

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

int main(int argc, char* argv[])
{
   /* input variables declaration */
   int (*costs)[3][3] = 0x30000;
   (*costs)[0][0]=5; (*costs)[0][1]=8; (*costs)[0][2]=4;
   (*costs)[1][0]=6; (*costs)[1][1]=6; (*costs)[1][2]=3;
   (*costs)[2][0]=3; (*costs)[2][1]=9; (*costs)[2][2]=6;

   int *demand = 0x30030;
   demand[0]=20; demand[1]=95; demand[2]=35;
   int *supply = 0x30040;
   supply[0]=50; supply[1]=40; supply[2]=60;
   int *total_cost = 0x30060;

   /* Others */
   int i, j, s, d, dest_i, src_i;
   int min_cost, max_demand, transfer;
```

```
21
22    *total_cost = 0;
23
24    for (i=0; i<3; i++) {
25        for (j=0; j<3; j++) {
26
27            /* Minimum value */
28            min_cost = 0x7FFFFFFF;
29            for(s=0; s<3; s++) {
30                for(d=0; d<3; d++) {
31                    if( (*costs)[s][d] < min_cost ) min_cost = (*costs)[s][d];
32                }
33            }
34
35            /* Path computation: source and demand */
36            max_demand = 0;
37            for(s=0; s<3; s++) {
38                for(d=0; d<3; d++) {
39                    if( ((*costs)[s][d] == min_cost) && (demand[d]>=max_demand) ) {
40                        max_demand = demand[d];
41                        dest_i     = d;
42                        src_i      = s;
43                    }
44                }
45            }
46
47            /* Total cost update */
48            if(supply[src_i] <= demand[dest_i])
49                transfer = supply[src_i];
50            else
51                transfer = demand[dest_i];
52            demand[dest_i] = demand[dest_i] - transfer;
53            supply[src_i]  = supply[src_i] - transfer;
54            (*costs)[src_i][dest_i] = (*costs)[src_i][dest_i] | 0x7FFFFFFF;
55            *total_cost = *total_cost + (transfer * min_cost);
56        }
57    }
58
59    return EXIT_SUCCESS;
60 }
```

Listing 5.17: Standard program *transport_min_cost.c*

### 3.3.2 Execution time estimation

The execution time estimation for *transport_min_cost.c* program, considers $N = M * M$ the total number of cost elements, corresponding to the number of possible paths between supplier and buyers. Only the parts of the code that could be executed with LiM instructions are considered in the computation. The code that remains unchanged between the three versions is considered as *fixed* and it is not included as offset in the Figure 5.6.

$$Execution\_time_{old\_ISA}(transport\_min\_cost.c) \approx N(4N(min) + fixed + 3(mask_min)) \tag{5.13}$$

$$Execution\_time_{newIF\_ISA}(transport\_min\_cost.c) \approx N(33(min) + fixed + 1(mask_min)) \tag{5.14}$$

$$\begin{aligned} Execution\_time_{sameIF\_ISA}(transport\_min\_cost.c) \approx N(&1(mem\_active) + 33(min) \\ &+ 1(mem\_active) + fixed \\ &+ 3(mask_m in)) \end{aligned} \tag{5.15}$$

From the execution with the old_ISA instructions, it is expected to see a quadratic dependency on the number $N$, as immediately visible from Figure 5.6. In fact, the total number of clock cycles needed is computed considering that the minimum needs 4 steps for each matrix element. In fact, in each iteration the RISC-V should read the matrix element and the minimum variable, compare the two and save the minimum between them. After all the updates are done with the new minimum, the considered cost element in the matrix is masked, and so a factor 3 is considered to load the data, mask with 1s, store the data back in memory. Those operations are repeated for all the possible $N$ paths in the matrix. The LiM instructions instead allow to change the dependency on $N$ from quadratic to

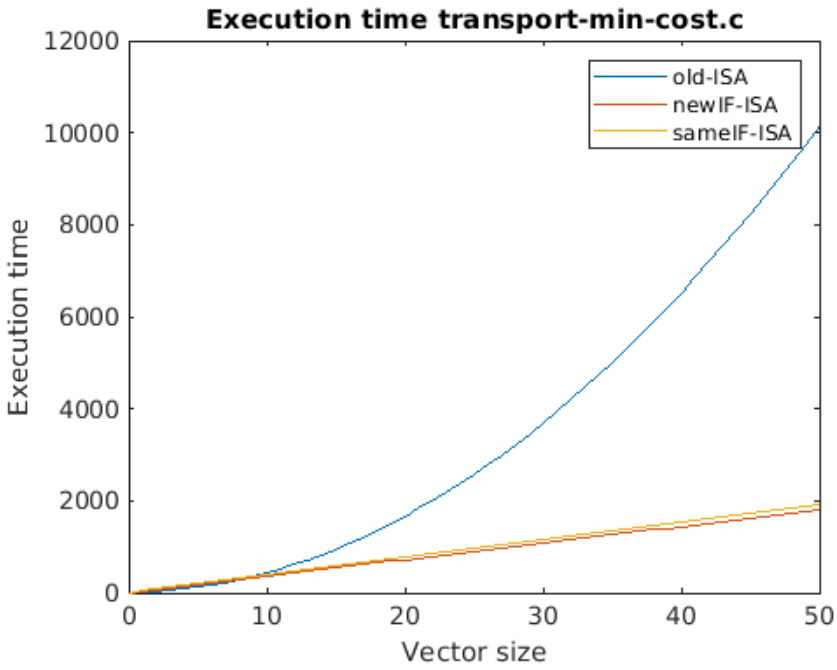



Figure 5.6: Estimation execution time transport_min_cost.c in old_ISA, newIF_ISA and sameIF_ISA

linear. As a matter of fact, for both newIF_ISA and sameIF_ISA, the minimum computation takes always 33 clock cycles so there is not a dependency on $N$ anymore. As always, the only difference between the two are the additional two clock cycles needed for the memory configuration. Regarding the mask computation, only the newIF_ISA introduced a change with respect to the initial version. In fact, while for newIF_ISA the masking operation can be done directly in the memory, for the sameIF_ISA doing it, would not have allowed to a reduction of the instructions required because of the needed configuration instructions.

### 3.3.3 Simulation results

Simulations show a really good improvement in terms of execution time in both cases for $N = 9$. By increasing the number $N$ the improvement is expected to be even better.

```
Time     Cycles PC    Instr    Mnemonic

2086ns   204 00000254 0607a023 sw              x0, 96(x15)         x15:00030000  PA:00030060
2096ns   205 00000258 00300f13 addi            x30, x0, 3          x30=00000003
2106ns   206 0000025c fffe4e13 xori            x28, x28, -1        x28=7fffffff x28:80000000
2116ns   207 00000260 00c00593 addi            x11, x0, 12         x11=0000000c
2126ns   208 00000264 00300f93 addi            x31, x0, 3          x31=00000003
2136ns   209 00000268 04078293 addi            x5, x15, 64          x5=00030040 x15:00030000
2146ns   210 0000026c 00300e93 addi            x29, x0, 3          x29=00000003
2156ns   211 00000270 1280006f jal             x0, 296
2176ns   213 00000398 000e0613 addi            x12, x28, 0         x12=7fffffff x28:7fffffff
2186ns   214 0000039c 00000693 addi            x13, x0, 0          x13=00000000
2196ns   215 000003a0 ed5ff06f jal             x0, -300
2216ns   217 00000274 02b68733 mul             x14, x13, x11       x14=00000000 x13:00000000 x11:0000000c
2226ns   218 00000278 00e78733 add             x14, x15, x14       x14=00030000 x15:00030000 x14:00000000
2236ns   219 0000027c 00472503 lw              x10, 4(x14)         x10=00000008 x14:00030000  PA:00030004
2246ns   220 00000280 00072703 lw              x14, 0(x14)         x14=00000005 x14:00030000  PA:00030000
2266ns   222 00000284 00e55463 bge             x10, x14, 8         x10:00000008 x14:00000005
2296ns   225 0000028c 02b68533 mul             x10, x13, x11       x10=00000000 x13:00000000 x11:0000000c
2306ns   226 00000290 00a78533 add             x10, x15, x10       x10=00030000 x15:00030000 x10:00000000
2316ns   227 00000294 00852503 lw              x10, 8(x10)         x10=00000004 x10:00030000  PA:00030008
2336ns   229 00000298 00e55463 bge             x10, x14, 8         x10:00000004 x14:00000005
2346ns   230 0000029c 00050713 addi            x14, x10, 0         x14=00000004 x10:00000004
2356ns   231 000002a0 00c75463 bge             x14, x12, 8         x14:00000004 x12:7fffffff
2366ns   232 000002a4 00070613 addi            x12, x14, 0         x12=00000004 x14:00000004
2376ns   233 000002a8 00168693 addi            x13, x13, 1         x13=00000001 x13:00000000
2386ns   234 000002ac fdf694e3 bne             x13, x31, -56       x13:00000001 x31:00000003

18036ns 1799 00000378 00e78733 add             x14, x15, x14       x14=0003001c x15:00030000 x14:0000001c
18046ns 1800 0000037c 00072503 lw              x10, 0(x14)         x10=00000009 x14:0003001c  PA:0003001c
```

```
18066ns 1802 00000380 01c56533 or              x10, x10, x28       x10=7fffffff x10:00000009 x28:7fffffff
18076ns 1803 00000384 00a72023 sw              x10, 0(x14)         x10:7fffffff x14:0003001c  PA:0003001c
18086ns 1804 00000388 0607a703 lw              x14, 96(x15)        x14=00000253 x15:00030000  PA:00030060
18106ns 1806 0000038c 00d706b3 add             x13, x14, x13       x13=000003bb x14:00000253 x13:00000168
18116ns 1807 00000390 06d7a023 sw              x13, 96(x15)        x13:000003bb x15:00030000  PA:00030060
18126ns 1808 00000394 000e8863 beq             x29, x0, 16         x29:00000000
18156ns 1811 000003a4 ffff0f13 addi            x30, x30, -1        x30=00000000 x30:00000001
18166ns 1812 000003a8 ec0f12e3 bne             x30, x0, -316       x30:00000000
18176ns 1813 000003ac 00000513 addi            x10, x0, 0          x10=00000000
18186ns 1814 000003b0 00008067 jalr            x0, x1, 0            x1:000001d8
```

Listing 5.18: Extract of simulation result of *transport_ min_ cost.c* - old_ISA

```
Time     Cycles PC    Instr    Mnemonic

2086ns   204 00000254 0607a023 sw              x0, 96(x15)         x15:00030000  PA:00030060
2096ns   205 00000258 00300f13 addi            x30, x0, 3          x30=00000003
2106ns   206 0000025c fffe4e13 xori            x28, x28, -1        x28=7fffffff x28:80000000
2116ns   207 00000260 00c00593 addi            x11, x0, 12         x11=0000000c
2126ns   208 00000264 00300f93 addi            x31, x0, 3          x31=00000003
2136ns   209 00000268 04078293 addi            x5, x15, 64          x5=00030040 x15:00030000
2146ns   210 0000026c 00300e93 addi            x29, x0, 3          x29=00000003
2156ns   211 00000270 0ec0006f jal             x0, 236
2176ns   213 0000035c 000e0613 addi            x12, x28, 0         x12=7fffffff x28:7fffffff
2186ns   214 00000360 00000693 addi            x13, x0, 0          x13=00000000
2196ns   215 00000364 f11ff06f jal             x0, -240
2216ns   217 00000274 02c7a223 sw              x12, 36(x15)        x12:7fffffff x15:00030000  PA:00030024
2226ns   218 00000278 0007d61b lw_min          N10 x12, 0(x15)     x12=00000003 x15:00030000  PA:00030000
2236ns   219 0000027c 00000693 addi            x13, x0, 0          x13=00000000

15426ns 1538 00000344 00e78733 add             x14, x15, x14       x14=0003001c x15:00030000 x14:0000001c
15436ns 1539 00000348 01c7303b sw_or           N-x0 x28, 0(x14)    x28:7fffffff x14:0003001c  PA:0003001c
15446ns 1540 0000034c 0607a703 lw              x14, 96(x15)        x14=00000253 x15:00030000  PA:00030060
15466ns 1542 00000350 00d706b3 add             x13, x14, x13       x13=000003bb x14:00000253 x13:00000168
15476ns 1543 00000354 06d7a023 sw              x13, 96(x15)        x13:000003bb x15:00030000  PA:00030060
15486ns 1544 00000358 000e8863 beq             x29, x0, 16         x29:00000000
15516ns 1547 00000368 ffff0f13 addi            x30, x30, -1        x30=00000000 x30:00000001
15526ns 1548 0000036c f10f10e3 bne             x30, x16, -256      x30:00000000 x16:00000000
15536ns 1549 00000370 00000513 addi            x10, x0, 0          x10=00000000
15546ns 1550 00000374 00008067 jalr            x0, x1, 0            x1:000001d8
```

Listing 5.19: Extract of simulation result of *transport_ min_ cost.c* - new_ISA

```
Time     Cycles PC    Instr    Mnemonic

2086ns   204 00000254 0607a023 sw              x0, 96(x15)         x15:00030000  PA:00030060
2096ns   205 00000258 00020cb7 lui             x25, 0x20000        x25=00020000
2106ns   206 0000025c 00300f13 addi            x30, x0, 3          x30=00000003
2116ns   207 00000260 fffe4e13 xori            x28, x28, -1        x28=7fffffff x28:80000000
2126ns   208 00000264 00c00593 addi            x11, x0, 12         x11=0000000c
2136ns   209 00000268 00300f93 addi            x31, x0, 3          x31=00000003
2146ns   210 0000026c 04078293 addi            x5, x15, 64          x5=00030040 x15:00030000
2156ns   211 00000270 00300e93 addi            x29, x0, 3          x29=00000003
2166ns   212 00000274 0fc0006f jal             x0, 252
2186ns   214 00000370 000e0613 addi            x12, x28, 0         x12=7fffffff x28:7fffffff
2196ns   215 00000374 00000693 addi            x13, x0, 0          x13=00000000
2206ns   216 00000378 f01ff06f jal             x0, -256
2226ns   218 00000278 00900613 addi            x12, x0, 9          x12=00000009
2236ns   219 0000027c ffccd63b sw_active_min   Nx12 -4(x25)        x25:00020000 x12:00000009  PA:0001fffc
2246ns   220 00000280 0007861b lw_mask         x12, x0, 0(x15)     x12=00000003 x15:00030000  PA:00030000
2256ns   221 00000284 ffcc803b sw_active_none  Nx0 -4(x25)         x25:00020000  PA:0001fffc
2586ns   254 00000288 00000693 addi            x13, x0, 0          x13=00000000

15856ns 1581 00000350 00e78733 add             x14, x15, x14       x14=0003001c x15:00030000 x14:0000001c
15866ns 1582 00000354 00072503 lw              x10, 0(x14)         x10=00000009 x14:0003001c  PA:0003001c
15886ns 1584 00000358 01c56533 or              x10, x10, x28       x10=7fffffff x10:00000009 x28:7fffffff
15896ns 1585 0000035c 00a72023 sw              x10, 0(x14)         x10:7fffffff x14:0003001c  PA:0003001c
15906ns 1586 00000360 0607a703 lw              x14, 96(x15)        x14=00000253 x15:00030000  PA:00030060
15926ns 1588 00000364 00d706b3 add             x13, x14, x13       x13=000003bb x14:00000253 x13:00000168
15936ns 1589 00000368 06d7a023 sw              x13, 96(x15)        x13:000003bb x15:00030000  PA:00030060
15946ns 1590 0000036c 000e8863 beq             x29, x0, 16         x29:00000000
15976ns 1593 0000037c ffff0f13 addi            x30, x30, -1        x30=00000000 x30:00000001
15986ns 1594 00000380 ee0f18e3 bne             x30, x0, -272       x30:00000000
15996ns 1595 00000384 00000513 addi            x10, x0, 0          x10=00000000
16006ns 1596 00000388 00008067 jalr            x0, x1, 0            x1:000001d8
```

Listing 5.20: Extract of simulation result of *transport_ min_ cost.c* - same_ISA

# 4 Simulation Results Analysis

All the simulation results are collected in the Table 5.3 in order to evaluate and compare the improvements introduced by the Logic-in-Memory.
Generally the estimations done in the previous sections were quite accurate. Even if the estimated execution time was a bit different from the actual execution time, the given

| | NO LiM | NEW IF LiM | SAME IF LiM |
|---|---|---|---|
| **Custom programs** | | | |
| **bitwise.c** | 129 cc | 24 cc (-82%) | 33 cc (-74%) |
| **max_min.c** | 177 cc | 71 cc (-59%) | 74 cc (-58%) |
| **Standard programs** | | | |
| **bitmap_search.c** | 119 cc | 113 cc (-5%) | *116 cc (-2%) |
| **aes128_addroundkey.c** | 208 cc | 176 cc (-15%) | 179 cc (-14%) |
| **transport_min_cost.c** | 1609 cc | 1345 (-16%) | 1391 cc (-14%) |

Table 5.3: Simulation results comparison

formulas revealed to be a good method to perform an approximation and then to evaluate if it is convenient or not use the new LiM instructions on a given algorithm.

Going into the details of the simulation results, the RI5CY processor with the LiM instructions showed an improvement of more than 70% in the case of the custom programs. The improvement is quite high because the programs were tailored to take advantage of the new LiM instructions. However, this is still a significant result since it demonstrate which is the upper bound of the improvement allowed by the Logic-in-Memory.
The simulation results of the standard programs instead were as expected different. The *bitmap_search.c* algorithm came out to be not really suitable for Logic-in-Memory operations. The main reason is the usage of the LOAD LiM instruction. In fact, the RI5CY processor cannot use the forwarding technique in case of a data coming from the memory. For this reason when the instruction gives a result needed by the immediate next instruction, the microprocessor stalls for one clock cycle. As a consequence, since this algorithm mostly relied on the LOAD LiM instruction to reduce the execution time, the result was not so good as expected. Anyway, a compiler can avoid as much as possible the dependencies and so fully exploit the Logic-in-Memory instructions.
A significant result is instead found for *aes128_addroundkey.c* and *transport_min_cost.c*, that showed around 15% of reduction in any LiM versions.

Anyway, for all the tested algorithms, as expected the RI5CY that supports the Logic-in-Memory keeping the original interface, always performs worse than the one with the new interface and in some cases it is even not convenient to use. For example, the *bitmap_search.c* program in the case of the sameIF_ISA (*) required an additional assumption that is not always true in every application of the algorithm. However, in most of the cases the improvement in terms of percentage is almost equivalent, so it could be more desirable to have a microprocessor that does not have a special interface.

Another last important consideration to make, is that the simulations have been done with a small amount of memory data, because of simulation time and manual compilation restrictions. Real world programs have to work with huge amount of data, so it possible to affirm that the new Logic-in-Memory ISA extensions would perform even better in that context.

# 5 Synthesis

The main goal of this thesis was to demonstrate the potentiality of the Logic-in-Memory in a system like the RISC-V microprocessor. The focus has always been to work on the architectural and behavioural point of view to reduce the execution time of the software running on the microprocessor.

However, the synthesis step has been performed just to have an idea on how the Logic-in-Memory can impact the characteristics of the entire system RISC-V and Memory. The gate library used for the synthesis is the 15nm Opengate Library.

| | NO LiM | NEW IF LiM | SAME IF LiM |
|---|---|---|---|
| **RI5CY** | | | |
| **Area** | 11188.518 $\mu m^2$ | 11259.691 $\mu m^2$ | 11193.385 $\mu m^2$ |
| **Minimum period** | 37.08 $ns$ | 38.01 $ns$ | 37.06 $ns$ |
| **Total power** | 112.5419 $mW$ | 114.0089 $mW$ | 112.5414 $mW$ |
| **RAM** | | | |
| **Area** | 19479.13 $\mu m^2$ | 40627.27 $\mu m^2$ | 40679.47 $\mu m^2$ |
| **Minimum period** | 10.87 $ns$ | 9 $ns$ | 18.14 $ns$ |
| **Total power** | 299.7810 $mW$ | 3.1012e+03 $mW$ | 1.4009e+03 $mW$ |

Table 5.4: Synthesis results comparison

Synthesis results of the two versions of the RI5CY that support the Logic-in-Memory ISA extension do not show any significant change (see Table 5.4). At basically the same cost in terms of timing, power and area, the RI5CY microprocessor can now support any kind of Logic-in-Memory operations. In fact, the new supported instructions are not LiM-specific, so a more complex Logic-in-Memory could be interfaced or even a Logic-in-Memory with a completely different structure but that keeps the same interface with the microprocessor.

The area and power are slightly bigger in the case of the new interface RI5CY. The reason is obviously related to the new interface ports, driven by the microprocessor. The result related to the minimum clock period does not have an explanation related to the new signals. As a matter of fact, the critical path does not change between the three designs. Therefore, it is possible to consider that the small difference is caused by the noise of the synthesis tool, because the algorithms that perform the synthesis can produce slightly different results with two slightly different initial conditions.

Regarding the memory synthesis results, it is necessary to state that the tool synthesised the memory as a set of flip-flops, so ignoring all the possible optimisations that can be applied on memory arrays. The synthesis memory compiler was not available, so the obtained data should not be considered valid in an absolute form, but can be used to to perform a relative comparison between the initial version of the memory and the two LiM versions.
During the simulation a memory of $2^{22}$ Bytes has been used, but the synthesis tool could not process such a large array, so data in the Table 5.4 are related to a memory of $2^{10}$ Bytes. Power consumption results are not considered reliable, because the introduction of the new ISA extensions should result in a different switching activity in the memory due to the less frequent memory operations, so resulting in a different total power. The synthesis tool did not have this information so the result is not considered valid to build a discussion.
The timing measurements, are not reliable as well because they do not corresponds to the real memories implementation and real memories size.
Area measurements can instead be considered more meaningful. In fact, even keeping in mind that the absolute area values do not correspond to the real implementation, it is possible to observe that the Logic-in-Memory has an area twice as big as a normal memory. This is the known cost to pay in order to have better performance in terms of execution time of programs.

# Chapter 6

# Conclusions and Future Work

The Logic-in-Memory integration in the RI5CY computing system allowed to achieve an improvement in the execution time in almost all the tested programs. However, in order to have much more flexibility, the Logic-in-Memory should support additional operations to balance the workload between memory and processor. In this way, a wider range of programs would benefit from the new memory.
The limit of the shifting of data-processing from processor to memory should be established by timing, area and power constraints, according to the actual physical implementation and the core involved. In fact, in case of the RI5CY core synthesis, the synthesised 1KB memory (typical cache size) does not limit the core in terms of timing, so if no additional constraints are required, it would be possible to apply many other LiM operations.

This Thesis aimed to introduce the Logic-in-memory concept in a microprocessor system and the validity of this approach has been demonstrated. Hence, future works can keep exploring this new path and new Logic-in-Memory models, introducing new operations and algorithms. Moreover, an important focus can be the actual physical implementation of the memory cell, that can exploit the available technologies to minimise the cell area.
Therefore, the Logic-in-Memory feature seems to be a promising option to overcome the microprocessor-memory communication bottleneck in Von-Neumann architectures.

This thesis opens many paths on possible future works. A path to explore is for sure the RISC-V compiler expansion that will include the new LiM extensions, supported now only by the hardware. Having the new ISA extension available with the compiler, not only would allow to consider a very wide set of benchmarks but it would also allow to perform all the optimisation at the machine language level that are not manually possible.

# Appendices

# Appendix A

# System Verilog basics

## 1 Introduction

SystemVerilog is a hardware description and hardware verification language used to model, design, simulate, test and implement electronic systems. SystemVerilog is built on top of the Verilog standard but improves the productivity, readability, and reusability of Verilog based code. The language enhancements in SystemVerilog provide more concise hardware descriptions. The enhancements also provide extensive support for directed and constrained-random testbench development, coverage driven verification, and assertion based verification.

## 2 Data objects and data types

### 2.1 Data types

SystemVerilog offers many improved data structures compared with Verilog. Some of these were created for designers but are also useful for testbenches.
Data types in SystemVerilog are used to describe how many states can be associate to a single bit of logic.

#### 2.1.1 Most common data-types

- **Logic.** It is a four-state data type, then each but can be 0,1,X or Z. Size is user-defined.
- **Bit.** It is a two-state data type, so each bit can assume value 0 or 1. Size is user-defined.
- **Byte.** It is a two-state data type with a size of 8 bits. By default is a signed number.
- **Byte unsigned.** The unsigned version of the bit data type.
- **Shortint.** It is a two-state data type with a size of 16 bits. By default is a signed number.
- **Shortint unsigned.** The unsigned version of the shortint data type.
- **Int.** It is a two-state data type with a size of 32 bits. By default is a signed number.
- **Int unsigned.** The unsigned version of the int data type.

- **Integer.** It is a four-state data type with a size of 32 bits. By default is a signed number.
- **Integer unsigned.** The unsigned version of the integer data type.
- **Longint.** It is a two-state data type with a size of 64 bits. By default is a signed number.
- **Longint unsigned.** The unsigned version of the longint data type.
- **Shortreal.** It represents a signed floating point number, with a size of 32 bits.
- **Real.** It represents a signed floating point number, with a size of 64 bits.
- **Time.** It is typically a 64-bit unsigned data type, used for the simulation.
- **String.** It is an ordered collection of characters. Its length is user-defined and each character is of type *byte*.
- **Enum.** It defines a set of named constants. A data-type can be associated to it, but in absense of a data-type declaration, the default data type should be *int*. The way to declare it is:

```
enum data_type {enum_list_declaration} enum_name_declaration;
```

Example:

```
enum { red, green, blue, yellow, white, black } Colors; // each color is an int data-type, ordered numbers
enum {a=3, b=7, c} alphabet;                            // c is automatically assigned the increment-value of 8
enum {a=0, b=7, c, d=8} alphabet;                       // Syntax error: c and d are both assigned 8
enum bit [3:0] {bronze='h3, silver, gold='h5} medal4;   // each constant is bit [3:0] data-type
```

- **Struct.** It is a way to group several data types. The entire group can be referenced as a whole, or the individual data type can be referenced by name. Struct can be defined as packet or unpacket.
  - A *packet* struct is treated as a single vector, and each data type in the structure is represented as a bit field. The entire structure is then packed together in memory without gaps. Only packed data types and integer data types are allowed in a packed struct. Because it is defined as a vector, the entire structure can also be used as a whole with arithmetic and logical operators. Packet structures can be followed by the signed or unsigned keywords, according to the desired arithmetic behavior.
  - An *unpacked* SystemVerilog struct, on the other hand, does not define a packing of the data types. It is tool-dependent how the structure is packed in memory. It is the default one, when not specified. Unpacked struct probably will not be synthesize by a synthesis tool, so to not use in RTL code.

In general struct data-type should be declared as:

```
struct {data_type_list} struct_object_identifier;
struct packet {data_type_list} struct_object_identifier;
```

Example:

```
// Declaration
struct { bit [7:0] opcode; bit [23:0] address;} IR; // anonymous structure, IR name of variable
// Usage od struct
IR.opcode = 1;                                      // set field opcode in IR.
IR.address = 1;                                     // set field address in IR.
```

```
// Declaration
struct packed signed { int a; shortint b; byte c; bit [7:0] d;} pack1; // signed, 2-state
struct packed unsigned { time a; integer b; logic [31:0] c; }   pack2; // unsigned, 4-state
```

- **Void.** It represents non-existent data. This type can be specified as the return type of functions, with no return value.
- **User-defined.** Using the existent data-types is possible to create a user-defined data-type with some specific characteristics. *Typedef* allows to create a new data-type (not a variable), to be used later in a variable declaration. The way to define it is:

```
typedef data_type type_name;
```

Example:

```
// Declare an alias for this long definition
typedef unsigned shortint                              u_shorti;    // name new shortint-based data-type, no variable
typedef bit [7:0]                                      ubyte;       // name new bit-based data-type, no variable
typedef enum {NO, YES}                                 boolean;     // name enum type, no variable
typedef struct { bit [7:0] opcode; bit [23:0] addr;}   instruction; // name structure type, no variable
// Creation of variables with the new data-types
u_shorti    my_data;
ubyte       my_byte;
boolean     myvar;
instruction IR;
```

#### 2.1.1.1 Arrays

All data types can be declared as arrays in SystemVerilog. Data-types like *logic*, *bit*, can have the vector-size associated to them. The dimension declared before the object name is referred to as the "vector width" dimension or "packet" dimension. The dimension declared after the object name is referred as the "array" dimension or "unpacket" dimension. To declare an array:

```
data_type [vector_dimension] name_variable [array_dimension]
```

Example:

```
bit [7:0] c1;           // packed array, one byte
bit [7:0] c1 [0:9];     // unpacked array of bytes
real u [7:0];           // unpacked array
```

## 2.2 Data objects

### 2.2.1 Net data objects

Nets represent structural connections between parts of design, such as gate primitives or module instances. Nets have values that depend on the drivers to which they are connected to. They can be used as scalar and vector wires to connect together the ports of design blocks . They can be treated as physical wires so no values get stored in them. They need to be driven by either continuous assign statement or from a port of a module. Assignments to nets can have a single driver or multiple ones and in the second case, the net will have a resulting value according to defined resolution function.

- **Wire.** It is probably the most common net data type. It is usually driven by one driver. In case many drivers are assigned to the wire net, the resolution function will give as result X with drivers with different value, and the value of the drivers if they show all the same value. By default, is associate to a 4-state data type with initial state equal to Z.
- **Wand.** It is used to be driven by multiple driver. In particular, the resolution function solves the conflict by giving as result of the net the value given by an AND gate that has as input the drivers of the net.

- **Wor.** It is used to be driven by multiple driver. In particular, the resolution function solves the conflict by giving as result of the net the value given by an OR gate that has as input the drivers of the net.

#### 2.2.1.1 Variable data objects

Variables generally represent a piece of storage. Variables are used to store combinational and sequential values. They retain their value till next value is assigned to them only using procedural assignments (this means inside an always block, an initial block, a task, a function). They can be synthesized as a flip-flop, latch or combinatorial circuit or they might not be synthesizable. They can be single or multi-bit.

- **Reg/Var.** It is associate by default to a 4-state data type with initial value equal to X.

## 3 Literal Values

This section explains how the numbers are interpreted and represented by SystemVerilog according to different categories of data-types.

- **Integer and Logical literals.** Numbers for this category can be represented using this format (<> is optional):

```
<size> ' <signed> <radix> value
```

  - *size* indicates the number of binary bits the number involved. Default is the lenght of the data-type.
  - ' is just a separator.
  - *signed* indicates if the value is signed, using s or S.
  - *radix* indicates the radix of the number: b or B (binary), o or O (octal), h or H (hexadecimal), d or D (decimal). Default is decimal.

  Example:

```
logic [11:0] var1 = 12'b1011;         //var1 = 0000_0000_1011
logic [11:0] var2 = 12'hB;            //var2 = 0000_0000_1011
logic [11:0] var3 = 4'shB;            //var3 = 1111_1111_1011, sign extention with MSB at position 3 (length-1)
logic [11:0] var4 = 'shB;             //var4 = 0000_0000_1011, sign extention with MSB at position 11 (length-1)
logic [11:0] var5 = 11;               //var5 = 0000_0000_1011
logic [11:0] var6 = 'X;               //var6 = XXXX_XXXX_XXXX
logic [11:0] var7 = '1;               //var7 = 1111_1111_1111
```

- **Real literals.** The default type is real for fixed-point format and exponent format.
  Example:

```
real var1 = 1.46;
real var2 = 1.46e0;
```

- **Time literals.** Time is written in integer or fixed point format, followed without a space by a time unit (fs ps ns us ms s).
  Example:

```
time delay = 5ns;
```

- **String literals.** The string literal is enclosed in quotes. The length of a string literal is not limited. A string literal can be assigned to an unpacked array of bytes.
  Example:

```
byte var1 [0:20] = "it␣is␣a␣string␣literal\n";
```

- **Array literals.** Array literals are syntactically similar to C initializers, but with the replicate operator ( {{}} ) allowed. Example:

```
int n [1:2][1:3] = '{'{0,1,2},'{3{4}}};  // n = '{'{0,1,2},'{4,4,4}}
int n [1:2][1:6] = '{2{'{3{4,5}}}};      // n = '{'{4,5,4,5,4,5},'{4,5,4,5,4,5}}
```

# 4 Operators

The SystemVerilog operators are a combination of Verilog and C operators. SystemVerilog, exactly as Verilog and C requires that the operands have the same data-type and the same size. The Table A.1 shows the most common operators.

| Operator Type | Operator Symbol | Operation Performed |
|---|---|---|
| **Arithmetic** | * | Multipy |
| | / | Division |
| | + | Addition |
| | - | Subtraction |
| | % | Modulus |
| | ++ | Increment |
| | - - | Decrement |
| **Logical** | ! | Logical negation |
| | && | Logical and |
| | \|\| | Logical or |
| **Relational** | > | Greater than |
| | < | Less than |
| | >= | Greater than or equal |
| | <= | Less than or equal |
| **Equality** | == | Equality |
| | != | Inequality |
| **Reduction** | ~ | Bitwise negation |
| | & | Bitwise and |
| | ~& | Bitwise nand |
| | \| | Bitwise or |
| | \|~ | Bitwise nor |
| | ^ | Bitwise xor |
| | ^ ~ | Bitwise xnor |
| | ~ ^ | Bitwise xnor |
| **Shift** | » | Right shift |
| | « | Left shift |
| **Concatenation** | {} | Concatenation = packed vector of bits |
| **Conditional** | ? | Conditional |

Table A.1: SystemVerilog operators

# 5 Signals and Constants

Signals are all the objects in the design that are not placed at the interface of a design module. A full declaration for them should have:

```
object_type data_type name;
```

Anyway, it is equally possible for signals to use the implicit declaration:

- The default data_object is *var/reg*;
- The default data_type is *logic*;

Here below an example:

```
wire my_wire;                       // implicitly means "wire logic my_wire"
wire logic my_wire;                 // explicit declaration
wire [7:0] my_wire_bus ;            // implicitly means "wire logic[15:0] my_wire_bus"
wire logic [7:0] my_wire_logic_bus; // explicit
struct_data_type my_struct;         // implicitly means "var struct_data_type my_struct"
wire struct_data_type my_struct;    // explicit
var[15:0] my_reg_bus;               // implicitly means "var logic[15:0] my_reg_bus"
logic my_var;                       // implicit means "var logic my_var"
```

Constants instead, are parameters that are used only within a design module and do not need to be declared in the interface. Declaration:

```
localparam data_type name;
```

The default data-type is integer signed.

## 6 Continuous assignments

The continuous assignment is the basic mechanism for placing values into nets and variables, in order to describe the functionality of the hardware, This assignment should occur whenever the value of the right-hand side changes.
There are two forms of continuous assignments:

- *Net declaration assignments*. It allows a continuous assignment to be placed on a net in the same statement that declares the net. Since the declaration of net is done one time, it is possible to use this assignment only once.
  Example:

```
/* Declaration and assignment */
wire logic mynet = enable;
```

- *Continuous assign statements*. It can be used with net and variables object types. Assignments on nets or variables are continuous and automatic. In other words, whenever an operand in the right-hand expression changes value, the whole right-hand side is evaluated.
  Nets can be driven by multiple continuous assignments or by a mixture of primitive outputs, module outputs, and continuous assignments. Variables can only be driven by one continuous assignment or by one primitive output or module output.

```
/* Declaration */
wire logic mynet;

/* Continous assignments */
assign mynet = enable;
```

```
module adder (sum_out, carry_out, carry_in, ina, inb);
    /* Ports declaration */
    output [3:0] sum_out;
    output carry_out;
    input [3:0] ina, inb;
    input carry_in;

    /* Signal declaration */
    wire carry_out, carry_in;       // net type is assigned to ports
    wire [3:0] sum_out, ina, inb;   // net type is assigned to ports

    /* Continous assignment */
    assign {carry_out, sum_out} = ina + inb + carry_in;

endmodule
```

# 7 Procedural assignments

Procedural assignments occur within procedures and can be thought of as “triggered” assignments. The trigger occurs during the simulation time and depends on the procedural block. The right-hand side of a procedural assignment can be any expression that evaluates a value, but the lefthand side should be a variable that receives the assignment from the right-hand side.
The most common procedural blocks are:

## 7.1 Procedural blocks

### 7.1.1 Initial block

This block, as the name suggests, is executed only once when simulation starts, then it can be useful for initialization in testbenches.

```
initial begin
...
end
```

### 7.1.2 Final block

This block, on the contrary is executed only once at the end of the simulation.

```
final begin
...
end
```

Example:

```
module blocking_assignment;
    //variables declaration
    int a,b;

    initial begin
        $display("_____________________________________________________________");
        //initializing a and b
        a = 10;
        b = 15;

        //displaying initial value of a and b
        $display("\tBefore_Assignment_::_Value_of_a_is_%0d",a);
        $display("\tBefore_Assignment_::_Value_of_b_is_%0d",b);

        a = b;
        b = 20;

        $display("\tAfter__Assignment_::_Value_of_a_is_%0d",a);
        $display("\tAfter__Assignment_::_Value_of_b_is_%0d",b);
        $display("_____________________________________________________________");
    end

    final begin  //final block will get executed at end of simulation
        $display("_____________________________________________________________");
        $display("\tEnd_of_Simulation_::_Value_of_a_is_%0d",a);
        $display("\tEnd_of_Simulation_::_Value_of_b_is_%0d",b);
        $display("_____________________________________________________________");
    end

endmodule
```

### 7.1.3 Always blocks

Blocks of this type execute a loop forever. An important note about always block is that they can not drive wire data type. There are few types of always blocks:

- **Always**. An always block should have a sensitive list or a delay associated with it. The sensitive list is the one which tells the always block when to execute the block of code. The @ symbol after reserved word ’ always’, indicates that the block will be triggered "at" the condition in parenthesis after symbol @.

```
always @ (<sensivity_list>)
    begin
    ...
end
```

Example:

```
always
    begin
    #5 clk = ~clk; //#5 in front of the statement delays its execution by 5 time units.
end
```

- **Always_comb**. SystemVerilog provides a special always procedure for modeling combinational logic behavior. always_comb automatically executes once at time zero, whereas always @ waits until a change occurs on a signal in the inferred sensitivity list. always_comb is sensitive to all changes within the contents described function.

```
always_comb
    begin
    ...
end
```

- **Always_latch**. SystemVerilog also provides a special always_latch procedure for modeling latched logic behaviour. The always_latch procedure determines its sensitivity and executes identically to the always_comb procedure. Software tools can perform additional checks to warn if the behaviour is equal to a latch.

```
always_latch
    begin
    ...
end
```

Example:

```
always_latch
    begin
    if(ck) q <= d;
end
```

- **Always_ff**. The SystemVerilog always_ff procedure can be used to model synthesisable sequential logic behaviour. This block imposes that it should contain one and only one event control and no blocking timing controls. Software tools will perform additional checks to warn if the behaviour within an always_ff procedure does not represent sequential logic behaviour.

```
always_ff @ (<sensivity_list>)
    begin
    ...
end
```

Example:

```
always_ff @(posedge clock iff reset == 0 or posedge reset)
    begin
    r1 <= reset ? 0 : r2 + 1;
    ...
end
```

### 7.1.4 Task

SystemVerilog allows a compact code by avoiding repetition, using tasks.

```
task <identifier> (<input_and_output>)
    ...
endtask
```

Example:

```
module sv_task;
    int x;

    //task to add two integer numbers.
    task sum(input int a,b,output int c);
        c = a+b;
    endtask

    initial begin
        sum(10,5,x);
        $display("\tValue of x = %0d",x);
    end
endmodule
```

### 7.1.5 Function

Functions work exactly as tasks, but they can return only one value.

```
function <data_type> <identifier> (<input_and_output>)
    ...
endtask
```

Example:

```
module sv_function;
    int x;
    //function to add two integer numbers.
    function int sum(input int a,b);
        sum = a+b;
    endfunction

    initial begin
        x=sum(10,5);
        $display("\tValue of x = %0d",x);
    end
endmodule
```

### 7.1.6 Generate

The generate construct allows to create multiple instances of an object, such as a module or an assign statements or to create or not objects according to a certain condition. The generate statement is not a run-time construct, as a matter of fact it is used to create hardware that cannot be removed at the simulation time. For this reason, the data-type *genvar* is an integer that exists only during elaboration time and is deleted before simulation time.

```
genvar i_var
generate
    for_loop_based_on_i_var
endgenerate
```

```
generate
    if_else_construct_for_parameter_evaluation
endgenerate
```

Example:

```
module mux_16(
    input  logic [0:15] [127:0] mux_in,
    input  logic [3:0] select,
    output logic [127:0] mux_out
    );

    logic [0:15] [127:0] temp;

    // The for-loop creates 16 assign statements
    genvar i;
    generate
        for (i=0; i < 16; i++) begin
            assign temp[i] = (select == i) ? mux_in[i] : 0;
        end
    endgenerate

    assign mux_out = temp[0] | temp[1] | temp[2] | temp[3] |
                     temp[4] | temp[5] | temp[6] | temp[7] |
                     temp[8] | temp[9] | temp[10] | temp[11] |
                     temp[12] | temp[13] | temp[14] | temp[15];
endmodule: mux_16
```

## 7.2 Procedural statements

Within the above-listed procedural blocks, it is possible to use many kind of statements.

### 7.2.1 Assignments

- **Blocking assignment**. It corresponds to '='. It is executed in series order within a procedural block, so it blocks the execution of the next statement until the completion of the current assignment execution.
- **Non-blocking assignment**. It corresponds to '<='. It is executed in parallel, at the same time in the simulation time.

### 7.2.2 If-else

If-else statements check a condition to decide whether or not to execute a portion of code. If a condition is satisfied, the code is executed. Else, it runs this other portion of code.

```
if (<condition>) begin
    ...
end
else begin
    ...
end
```

### 7.2.3 Case.

Case statements are used where there is one variable which needs to be checked for multiple values (e.g. an address decoder, where the input is an address and it needs to be checked for all the possible values.

```
case (<condition>) begin
    <value_1> : ... ;
    ...
    <value_n> : ... ;
    default   : ... ;
    ...
endcase
```

### 7.2.4 While

This loop-statement executes the code within it repeatedly if the condition assigned to check returns true.

```
while (<condition>) begin
    ...
end
```

Example:

```
module counter (clk,rst,enable,count);
    /* Port declaration */
    input clk, rst, enable;
    output [3:0] count;
    var [3:0] count;

    always @ (posedge clk or posedge rst)
        if (rst) begin
            count <= 0;
        end else begin : COUNT
            while (enable) begin
                count <= count + 1;
                disable COUNT;
            end
    end
endmodule
```

### 7.2.5 For loop

SystemVerilog for loop allows to declare a loop variable within the for loop, to use one or more initial declaration or assignment within the for loop one or more step assignment or modifier within the for loop.

```
for(initialization; condition; modifier) begin
    <statement_1>
    ...
    <statement_n>
end
```

# 8 Design elements

Design elements are the primary building blocks used to build a design and its verification environment. These building blocks contains the declarations, parallel and procedural code.

## 8.1 Module

The basic building block in SystemVerilog is the module. Modules are primarily used to represent design blocks, but can also serve as containers for verification code and interconnections between verification blocks and design blocks. A module can represent a simple digital components, such as a gates, or a complex digital system. Modules can instantiate other design elements, thereby creating a design hierarchy. Modules are defined using two keywords *module* and *endmodule*, identified with a unique name/identifier.

```
module();
...
endmodule
```

Usually a module contains many parts that allow to describe the hardware structure and functionality. These parts are described in the following paragraphs.

#### 8.1.0.1 Module header

The module header is the part that describes the interface of the module. It is placed right after the keyword *module* and the identifier (module name). The module header contains:

- **Parameter list**: the constant parameters associated to the module.

```
parameter data_type identifier;
```

- **Ports list**: each port should usually has **direction** (*input*, *output* or *inout*), **data object**, **data type** and **identifier**.

```
direction object_type data_type identifier;
```

  Anyway, it is possible to not use the full declaration but instead using the implicit ways to define the missing part:

  - The default direction is *inout*;
  - The default data_object is *wire* for input and inout ports. While the default data_object for output ports is *wire* if data_type is also omitted and *var* if data_type specified.
  - The default data_type is *logic*;

  Example:

```
module mh0 (wire x);                // inout wire logic x
module mh1 (integer x);             // inout wire integer x
module mh2 (inout integer x);       // inout wire integer x
module mh3 ([5:0] x);               // inout wire logic [5:0] x
module mh4 (var x);                 // ERROR: direction defaults to inout, which cannot be var
module mh5 (input x);               // input wire logic x
module mh6 (input var x);           // input var logic x
module mh7 (input var integer x);   // input var integer x
module mh8 (output x);              // output wire logic x
module mh9 (output var x);          // output var logic x
module mh10(output signed [5:0] x); // output wire logic signed [5:0] x
module mh11(output integer x);      // output var integer x
```

Modules header can be declared according to two different styles:

- Non-ANSI style

```
module module_name (port_identifier_list);       //module header
    parameter_declaration_list
    port_declarations_list

    Continous_assignments_or_procedural_blocks

endmodule: module_name
```

- ANSI style

```
module module_name #(parameter_declaration_list)
                    (port_declarations_list);    //module header

    Continous_assignments_or_procedural_blocks

endmodule: module_name
```

Examples:

```
module test(a,b,c,d,e,f,g,h);
    /* NON-ANSI */
    input logic [7:0]               a; // no explicit net declaration - net is unsigned
    input wire logic [7:0]          b;
    input wire logic signed [7:0]   c;
    input logic signed [7:0]        d; // no explicit net declaration - net is signed
    output logic [7:0]              e; // no explicit net declaration - net is unsigned
    output wire logic [7:0]         f;
    output wire logic signed [7:0]  g;
    output logic signed [7:0]       h; // no explicit net declaration - net is signed

    Continous_assignments_or_procedural_blocks

endmodule
```

```
module test
(   /* ANSI */
    input logic [7:0]               a, // no explicit net declaration - net is unsigned
    input wire logic [7:0]          b,
    input wire logic signed [7:0]   c,
    input logic signed [7:0]        d, // no explicit net declaration - net is signed
    output logic [7:0]              e, // no explicit net declaration - net is unsigned
    output wire logic [7:0]         f,
    output wire logic signed [7:0]  g,
    output logic signed [7:0]       h  // no explicit net declaration - net is signed
);

    Continous_assignments_or_procedural_blocks

endmodule
```

### 8.1.1 Module instance and hierarchy

There are two ways to instantiate a module: hierarchical or top level.

- **Top Level.** Top-level modules are implicitly instantiated. A design should contain at least one top-level module, whose instance name is the same as the module name. The name $root is used to unambiguously refer to a top-level instance. Since $root is the root of the instantiation tree, it allows to access through the tree.
  Example:

```
$root.A.B   // item B within top instance A
$root.A.B.C // item C within instance B within instance A
```

- **Hierarchical.** Hierarchical instantiation allows more than one instance of the same module. The module name can be a module previously declared or one declared later. Parameter assignments can be named or ordered. Port connections can be named, ordered, or implicitly connected. They can be nets, variables, or other kinds of interfaces, events, or expressions.

```
module_instance_name module_identifier ( ports_and_parameters_connection );
```

Connections can be made to module instances in the following four ways:

1. *Positional connections by port order.* With this method the port expressions listed for the module instance should be in the same order as the ports listed in the module declaration. A connection is done using simple net or variable identifiers, an expression, or a blank (no connection).
   Example:

```
module alu_accum1 (
    /* Module header */
    output [15:0] dataout,
    input [7:0] ain, bin,
    input [2:0] opcode,
    input clk, rst_n, rst );

    /* Signals */
    wire [7:0] alu_out;

    /* Sub-modules instances */
    alu alu_i (alu_out, , ain, bin, opcode);
    accum accum_i (dataout[7:0], alu_out, clk, rst_n);
    xtend xtend_i (dataout[15:8], alu_out[7], clk);

endmodule
```

2. *Connecting module instance ports by name.* The second way to connect module ports consists of explicitly linking the two names for each side of the connection: the port declaration name from the module declaration to the expression, i.e., the name used in the module declaration, followed by the name used in the instantiating module. The order is irrelevant.
   Example:

```
module alu_accum2 (
    /* Module header */
    output [15:0] dataout,
    input [7:0] ain, bin,
    input [2:0] opcode,
    input clk, rst_n, rst);

    /* Signal */
    wire [7:0] alu_out;

    /* Sub-modules instances */
    alu alu (.alu_out(alu_out), .zero(), .ain(ain), .bin(bin), .opcode(opcode));
    accum accum (.dataout(dataout[7:0]), .datain(alu_out), .clk(clk));
    xtend xtend (.dout(dataout[15:8]), .din(alu_out[7]), .clk(clk), .rst(rst) );

endmodule
```

3. *Connecting module instance using implicit named port connections.* SystemVerilog can implicitly instantiate ports using a .name syntax if the instance port name matches the connecting port name and their data types are equivalent. If a signal of the same name does not exist in the instantiating module, an error is issued.
   Example:

```
module alu_accum3 (
    /* Module header */
    output [15:0] dataout,
    input [7:0] ain, bin,
    input [2:0] opcode,
    input clk, rst_n, rst);

    /* Signal */
    wire [7:0] alu_out;

    /* Sub-modules instances */
```

```
12      alu alu (.alu_out, .zero(), .ain, .bin, .opcode);
13      accum accum (.dataout(dataout[7:0]), .datain(alu_out), .clk, .rst_n());
14      xtend xtend (.dout(dataout[15:8]), .din(alu_out[7]), .clk, .rst);
15
16  endmodule
```

4. *Connecting module instances using wildcard named port connections.* SystemVerilog allows to implicitly instantiate ports using a .* wildcard syntax for all ports where the instance port name matches the connecting port name and their data types are equivalent. This eliminates the requirement to list any port where the name and type of the connecting declaration match the name and equivalent type of the instance port. This implicit port connection style allows to specify the connection only for ports that do not match: a named port connection can be mixed with a .* connection to override a port connection to a different expression or to leave a port unconnected.

```
1   module alu_accum4 (
2       /* Module header */
3       output [15:0] dataout,
4       input [7:0] ain, bin,
5       input [2:0] opcode,
6       input clk, rst_n, rst);
7
8       /* Signal */
9       wire [7:0] alu_out;
10
11      /* Sub-modules instances */
12      alu alu (.*, .zero());
13      accum accum (.*, .dataout(dataout[7:0]), .datain(alu_out));
14      xtend xtend (.*, .dout(dataout[15:8]), .din(alu_out[7]));
15
16  endmodule
```

## 8.2 Interface

SystemVerilog uses the interface construct to describe the communication between blocks, as a matter of fact an interface is just a bundle of signals or nets. Interfaces can be used for the communication between a testbench and a design module or between design blocks. The interface allows:

1. to group together the number of signals as a single port: the single port handle is passed instead of multiple signal/ports.
2. to add and delete signals in an easy way.
3. the declare the interface once, and then just the handle is passed across the modules/components.

In other words, the interfaces aim is to encapsulate communication.
The declaration of an interface can include ports, parameters, constants, functions, and tasks. An interface cannot instantiate a module, but on the contrary, a module can instantiate an interface. The most common declaration of an interface has the declaration of the signals (the connections between modules) and then the declaration of modules with the ports that specify the direction associated to the connections. If the *modport* construct is not specified, then all the nets and variables in the interface are accessible with direction inout. The syntax is:

```
1   interface <interface_identifier> #( parameters )( ports );
2       signal_declaration
3
4       modport module_1 ( signal_direction ); //optional
5       modport module_2 ( signal_direction ); //optional
6
7   endinterface
```

The declaration of an interface inside a module is placed instead of the ports.

```
1   module module_identifier (interface_identifier.module_instance_int interface_instance_name)
```

Example:

```
//***************************************************
// Define the interface
//***************************************************
interface simple_bus(input logic clk); // Define the interface
    logic req, gnt;
    logic [7:0] addr, data;
    logic [1:0] mode;
    logic start, rdy;

    modport MEM(input address, inout data, input mode, input gnt, input start, output rdy, output req);
    modport CPU(output address, inout data, output mode, output gnt, output start, input rdy, input req);
endinterface: simple_bus

//***************************************************
//  Mem module with the interface
//***************************************************
module memMod(simple_bus.MEM a); // simple_bus interface port
    logic avail;
    // When memMod is instantiated in module top, a.req is the req
    // signal in the sb_intf instance of the 'simple_bus' interface
    always @(posedge clk) a.gnt <= a.req & avail;
endmodule

//***************************************************
//  Cpu module with the interface
//***************************************************
module cpuMod(simple_bus.CPU b); // simple_bus interface port
    ...
endmodule

//***************************************************
//  Top module with the interface
//***************************************************
module top;
    logic clk = 0;

    simple_bus.MEM sb_intf_mem(.clk(clk)); // Instantiate the mem interface
    simple_bus.CPU sb_intf_cpu(.clk(clk)); // Instantiate the cpu interface

    memMod mem(.a(sb_intf_mem)); // Connect interface to module instance
    cpuMod cpu(.b(sb_intf_cpu)); // Connect interface to module instance
endmodule
```

## 8.3 Package

SystemVerilog packages provide an additional mechanism for sharing parameters, data, type, task, function, sequence, and property declarations amongst multiple SystemVerilog modules, interfaces and programs. Packages are explicitly named in the source text (at the same level as top- level modules). All the shared objects in the design can be declared within a package.
Package definition:

```
package <package_name>;
...
endpackage : <package_name>
```

Package declaration:

```
import <name_package> ::*;
```

## 8.4 Program

For the testbench, the emphasis is not in the hardware-level details such as wires, structural hierarchy, and interconnects, but in modeling the complete environment in which a design is verified. The environment must be properly initialised and synchronised, automating the generation of input stimuli, and reusing existing models and other infrastructure.
In SystemVerilog the program construct contains full environment for testbench..
The program construct can be considered like a module with special execution semantics. Once declared, a program block can be instantiated in the required hierarchical location (typically at the top level), and its ports can be connected in the same manner as any other module (ports connection or interface).

```
program test (port_declaration_or_interface);
    initial begin
        ...
    end
```

```
5
6       ...
7   endprogram
```

Moreover, the main differences between a module and a program are:

1. Program block can not contain always block.
2. A module (design) can not call task/function inside a program block. But a program can call task/function inside module (design).

Example:

```
1   //************************************************
2   //  Program declaration
3   //************************************************
4   program simple(
5       input wire clk,
6       output logic reset, enable,
7       input logic [3:0] count);
8
9     //Initial block
10    initial begin
11      $monitor("@%0dns count = %0d",$time,count);
12      reset = 1;
13      enable = 0;
14      #20 reset = 0;
15      @ (posedge clk);
16      enable = 1;
17      repeat (5) @ (posedge clk);
18      enable = 0;
19      // Call task in module
20      simple_program.do_it();
21    end
22
23    //Task inside program
24    task do_it();
25      $display("%m I am inside program");
26    endtask
27
28  endprogram
29
30  //************************************************
31  //  Module declaration for program instance
32  //************************************************
33  module simple_program();
34      logic clk  = 0;
35      always #1 clk ++;
36      logic [3:0] count;
37      wire reset, enable;
38
39      //Counter
40      always @ (posedge clk) begin
41          if (reset) count <= 0;
42          else if (enable) count ++;
43      end
44
45      //Program instance
46      simple prg_simple(clk,reset,enable,count);
47
48      //Task inside module
49      task do_it();
50          $display("%m I am inside module");
51      endtask
52
53  endmodule
```

```
1   //************************************************
2   //  Simulation output
3   //************************************************
4    @0ns count = x
5    @1ns count = 0
6    @23ns count = 1
7    @25ns count = 2
8    @27ns count = 3
9    @29ns count = 4
10   I am inside module
11   @31ns count = 5
```

# 9 Assertion

One of the goals of SystemVerilog assertions is to provide a common semantic meaning for assertions so that they can be used to drive various design and verification tools.
Verification with assertions refers to the use of an assertion language to specify expected

behavior in a design. Assertions are primarily used to validate the behavior of a design. In addition, assertions can be used to provide functional coverage and to flag that input stimulus, which is used for validation, does not conform to assumed requirements.
Assertion statements appear with these keywords:

- **Assert**. Used to specify the property as an obligation for the design that is to be checked to verify that the property holds. Failure of an assert statement indicates a violation of the requirement and thus a potential error in the design.
- **Assume**. Used to specify the property as an assumption for the environment. Simulators check that the property holds, while formal tools use the information to generate input stimulus.
- **Cover**. Used to monitor the property evaluation for coverage. The cover statement specifies that successful evaluation of its expression is a coverage goal. Tools should collect coverage information and report the results at the end of simulation. The results of coverage for an immediate cover statement should contain the *number of times evaluated* and the *number of times succeeded*.
- **Restrict**. Used to specify the property as a constraint on formal verification computations. Simulators do not check the property.

If in the simulation a violation of an assertion occurs, the information about assertion failure can be printed using a severity system tasks in the action block (if available with the assertion type). Severity system tasks are explained in the Paragraph 10.3.

## 9.1 Immediate assertions

Immediate assertions are tests executed after simulation events, and are executed like a statement in a procedural block. Immediate assertions are primarily intended to be used with simulation. There is no immediate restrict assertion statement.
The expression is non-temporal and is interpreted the same way as an *if* procedural expression. If the expression evaluates to X, Z or 0, then it is interpreted as being false and the assertion is said to fail. Otherwise, the expression is interpreted as being true and the assertion is said to pass.
There are two types of immediate assertions:

- **Simple immediate assertions**. With this assertion, pass and fail actions take place immediately upon assertion evaluation.
  - **Assert**.

```
<label>: assert ( expression ) action_block
```

  - **Assume**.

```
<label>: assume ( expression ) action_block
```

  - **Cover**.

```
<label>: cover ( expression ) statement_or_null
```

  Example:

```
/* Assert */
assert_f: assert(f) $info("passed");
          else $error("failed");
/* Assume */
assume_inputs: assume (in_a || in_b) $info("assumption_holds");
               else $error("assumption_does_not_hold");

/* Cover */
```

```
cover_a_and_b: cover (in_a && in_b) $info("in_a_&&_in_b_==_1_covered");

/* Assertion repetitions */
time t;
always @(posedge clk)
    if (state == REQ)
        assert (req1 || req2)
    else begin
        t = $time;
        #5 $error("assert_failed_at_time_%0t",t); //If the immediate assert fails at time 10,
    end                                           //the error message should be printed at time 15
end
```

- **Deferred immediate assertions**. With this assertion, the actions are delayed until later in the time step, providing some level of protection against unintended multiple executions on transient or "glitch" values.
  When a deferred assertion passes or fails, the action block is not executed immediately. Instead, the action block subroutine call and the current values of its input arguments are placed in a *deferred assertion report queue*. Such a call is said to be a pending assertion report. If a *deferred assertion flush point* is reached in a process, its deferred assertion report queue is cleared and the pending assertion reports will not be executed. While pending assertion report that has not been flushed from its queue should mature, or be confirmed for reporting. Once a report matures, it may no longer be flushed. The deferred assertion flush point is reached if any of the following occur in the process:

  – The process, having been suspended earlier due to reaching an event control or wait statement, resumes execution.

  – The process was declared by an always_comb or always_latch, and its execution is resumed due to a transition on one of its dependent signals.

  – **Assert**.

  ```
  <label>: assert #0 ( expression ) action_block
  ```

  – **Assume**.

  ```
  <label>: assume #0 ( expression ) action_block
  ```

  – **Cover**.

  ```
  <label>: cover #0 ( expression ) statement_or_null
  ```

  Example:

```
assign not_a = !a;
always_comb begin : b1
    a1: assert (not_a != a);
    a2: assert #0 (not_a != a); // Should pass once values have settled
end
```

## 9.2 Concurrent assertions

Concurrent assertions are executed after simulation clock events, then evaluate sampled values of variables, ignoring any timing or event behavior between clock edges. The keyword *property* distinguishes a concurrent assertion from an immediate assertion.

### 9.2.1 Sequence layer

Concurrent assertions are often based on sequential behaviour. The sequence feature provides the capability to build and manipulate sequential behaviours. A sequence is a finite list of SystemVerilog *boolean expressions* in a linear order of increasing time. The sequence is said to match along a finite interval of consecutive clock ticks provided the first boolean

expression evaluates to true at the first clock tick, the second boolean expression evaluates to true at the second clock tick, and so forth, up to and including the last boolean expression evaluating to true at the last clock tick. A single boolean expression is an example of a simple linear sequence, and it matches at a single clock tick provided the boolean expression evaluates to true at that clock tick.

The construct to declare a sequence is:

```
sequence <sequence_identifier>
    @ ( clock_event )
    list_of_boolean_expressions
endsequence
```

A sequence can also not specify a clock. In this case, a clock would be inherited from some external source at a higher layer, such as a property or an assert statement:

```
sequence <sequence_identifier>
    list_of_boolean_expressions
endsequence
```

The sequences can use the following operators:

- ##. This operator followed by a number or a range specifies the delay from the current clock tick to the beginning of the sequence that follows. The delay ##1 indicates that the beginning of the sequence that follows is one clock tick later than the current clock tick. The delay ##0 indicates that the beginning of the sequence that follows is at the same clock tick as the current clock tick.
  Example:

```
//SEQUENCE 1: it means that gnt is expected to be asserted one clock cycle later req is asserted
sequence req_gnt_1clock_seq;
  req[0] ##1 gnt[0];
endsequence

//SEQUENCE 2: it means that gnt is expected to be asserted 0 to 3 clock cycles after req is asserted.
sequence req_gnt_3to5clock_seq;
  req[1] ##[3:5] gnt[1];
endsequence

//SEQUENCE 3: it means that the second sequence should occur once cycle later the first one occurs
sequence master_seq;
  req_gnt_1clock_seq ##1 req_gnt_3to5clock_seq;
endsequence
```

- **$**. This operator is used when something needs to be checked till end of simulation. It means that something will happen eventually before end of simulation.

- **[*<n>] Consecutive repetition**. operator is used to describe the repetition of a sequence. This operator specifies finitely many iterative matches of the operand sequence, with a delay of one clock tick from the end of one match to the beginning of the next. The overall repetition sequence matches at the end of the last iterative match of the operand.

- **[-><n>] Goto repetition**. It is used to describe the repetition of a sequence. This operator specifies finitely many iterative matches of the operand boolean expression, with a delay of one or more clock ticks from one match of the operand to the next successive match and no match of the operand strictly in between. The overall repetition sequence matches at the last iterative match of the operand.

- **[=<n>] Nonconsecutive repetition**. It is used to describe the repetition of a sequence. This operator specifies finitely many iterative matches of the operand boolean expression, with a delay of one or more clock ticks from one match of the operand to the next successive match and no match of the operand strictly in between. The overall repetition sequence

- **throughout**. It is a match operator. To check if some condition is valid over period of sequence.
- **within**. It is a match operator. To check containment of one sequence in another sequence.
- **intersect**. It is a binary operator. When both sequence are expected to match, and both the sequence start and end at same time.
- **and**. It is a binary operator. When both sequence are expected to match, and both the sequence start at same time, but are not expected to finish at same time.
- **or**. It is a binary operator. When one of the both sequence are expected to match.
- **|-> overlapped implication operator**. If there is a match for the antecedent sequence expression, then the first element of the consequent sequence expression is evaluated on the same clock tick.
- **|=> non-overlapped implication operator**. If there is a match for the antecedent sequence expression, then the first element of the consequent sequence expression is evaluated on the next clock tick.

### 9.2.2 Property layer

Property layer is built on top of sequence layer. It uses zero or more sequences to check a design assumption. Zero sequences because, property layer can contain boolean layer directly. So, generally a property defines a behavior of the design. A named property may be used for verification as an assumption, an obligation, or a coverage specification. In order to use the behavior for verification, an assert, assume, or cover statement must be used. A property declaration by itself does not produce any result.
To declare a property:

```
property <property_identifier>
    list_of_sequences_and_boolean_expressions
endproperty
```

Example:

```
/* Sequence declaration */
sequence seq;
  a ##2 b;
endsequence

/* Property declaration */
property p;
  @(posedge clk) seq;   //clock for sequence inherited by the property
endproperty
a_1 : assert property(p);
```

### 9.2.3 Assertions layer

- **Assert property**.

```
<label>: assert property ( property_identifier ) action_block
```

- **Assume property**.

```
<label>: assume property ( property_identifier ) action_block
```

- **Cover property**.

```
<label>: cover property ( property_identifier ) statement_or_null
```

- **Cover sequence**.

```
<label>: cover sequence ( [clocking_event ] [ disable iff ( expression_or_dist ) ] sequence_expr ) statement_or_null
```

- **Restrict property**.

```
<label>: restrict property ( property_identifier )
```

Example:

```
/* Assert */
property abc(a, b, c);
    disable iff (a==2) @(posedge clk) not (b ##1 c);
endproperty

env_prop: assert property (abc(rst, in1, in2))
$display("env_prop passed."); else $display("env_prop failed.");

/* Assume */
property abc(a, b, c);
    disable iff (c) @(posedge clk) a |=> b; //one cycle after a, b must be asserted
endproperty

env_prop:
assume property (abc(req, gnt, rst)) else $error("Assumption failed.");
```

## 9.3 Binding assertion

When RTL is already written and it becomes responsibilty of a verification engineer to add assertion, but an RTL designer does not want verification engineer to modify his RTL. For this reason, SystemVerilog provides a bind construct to keep verification code separated from the design code.
It is possible to write all the assertions in a separate file and using the bind statement, it is equally possible to bind the ports of his assertion file with the port/signals of the RTL in his testbench code. A bind feature can be used in modules or interfaces, so that the assertions can be instantiated in a target module or interface in a non-intrusive manner.
The syntax to bind all instances of a module is:

```
bind RTL_module_identifier Assertion_module_identifier Assertion_module_instance
```

The syntax to bind only one instance of a module is:

```
bind RTL_module_instance_path Assertion_module_identifier Assertion_module_instance
```

Example:

```
//************************************************
//   DUT With assertions
//************************************************
module RTL_module(
    input wire clk,req,reset,
    output reg gnt);

    //RTL description
    always @ (posedge clk)
        gnt <= req;
    end
endmodule

//************************************************
//   Assertion Verification IP
//************************************************
module assertion_ip(input wire clk_ip, req_ip,reset_ip,gnt_ip);

    // Sequence Layer
    sequence req_gnt_seq;
        (~req_ip & gnt_ip) ##1 (~req_ip & ~gnt_ip);
    endsequence

    // Property Specification Layer
    property req_gnt_prop;
        @ (posedge clk_ip)
            disable iff (reset_ip)
            req_ip |=> req_gnt_seq;
    endproperty

    // Assertion Directive Layer
    req_gnt_assert : assert property (req_gnt_prop)
                     else $display("@%0dns Assertion Failed", $time);
endmodule
```

```
35
36 //************************************************
37 //  Binding File version 1
38 //************************************************
39 module binding_module();
40
41     // Bind by Module name : This will bind all instance of DUT
42     bind RTL_module assertion_ip U_assert_ip (
43         .clk_ip   (clk),
44         .req_ip   (req),
45         .reset_ip (reset),
46         .gnt_ip   (gnt)
47     );
48 endmodule
49
50 //************************************************
51 //  Binding File version 2
52 //************************************************
53 module binding_module();
54
55     // Bind by instance name : This will bind only instance names in list
56     bind $root.bind_assertion_tb.dut assertion_ip U_assert_ip (
57          .clk_ip   (clk),
58          .req_ip   (req),
59          .reset_ip (reset),
60          .gnt_ip   (gnt)
61     );
62 endmodule
63
64 //************************************************
65 //  Testbench Code
66 //************************************************
67 `include "assertion_ip.sv"
68 `include "bind_assertion.sv"
69 `include "binding_module.sv"
70
71 module bind_assertion_tb();
72     reg clk = 0;
73     reg reset, req = 0;
74     wire gnt;
75
76     always #3 clk ++;
77
78     initial begin
79         reset <= 1;
80         #20 reset <= 0;
81         // Make the assertion pass
82         #100 @ (posedge clk) req  <= 1;
83         @ (posedge clk) req <= 0;
84         // Make the assertion fail
85         #100 @ (posedge clk) req  <= 1;
86         repeat (5) @ (posedge clk);
87         req <= 0;
88         #10 $finish;
89     end
90
91     RTL_module dut (clk,req,reset,gnt);
92 endmodule
```

```
1  //************************************************
2  //  Simulation output
3  //************************************************
4   "assertion_ip.sv", 22: bind_assertion_tb.dut.U_assert_ip.req_gnt_assert:
5    started at 237s failed at 243s
6           Offending '((~req_ip) & gnt_ip)'
7   @243ns Assertion Failed
8   "assertion_ip.sv", 22: bind_assertion_tb.dut.U_assert_ip.req_gnt_assert:
9    started at 243s failed at 249s
10          Offending '((~req_ip) & gnt_ip)'
11  @249ns Assertion Failed
12  "assertion_ip.sv", 22: bind_assertion_tb.dut.U_assert_ip.req_gnt_assert:
13   started at 249s failed at 255s
14          Offending '((~req_ip) & gnt_ip)'
15  @255ns Assertion Failed
16  "assertion_ip.sv", 22: bind_assertion_tb.dut.U_assert_ip.req_gnt_assert:
17   started at 255s failed at 261s
18          Offending '((~req_ip) & gnt_ip)'
19  @261ns Assertion Failed
20  $finish called from file "bind_assertion_tb.sv", line 26.
```

# 10 System tasks and system functions

### 10.0.1 Simulation control tasks

- **$stop()**. The $stop system task causes simulation to be suspended.
- **$finish()**. The $finish system task causes the simulator to exit and pass control back to the host operating system.

- **$exit()**. The $exit control task waits for all *program* blocks to complete, and then makes an implicit call to $finish. A program block may terminate the threads of all its initial procedures as well as all of their descendents explicitly by calling the $exit system task.

## 10.1 Simulation time functions

The following system functions provide access to current simulation time:

- **$time**. The $time system function returns an integer that is a 64-bit time, scaled to the time unit of the module that invoked it.
- **$stime**. The $stime system function returns an unsigned integer that is a 32-bit time, scaled to the time unit of the module that invoked it. If the actual simulation time does not fit in 32 bits, the low order 32 bits of the current simulation time are returned.
- **$realtime**. The $realtime system function returns a real number time that is scaled to the time unit of the module that invoked it.

## 10.2 Math functions

### 10.2.1 Integer math functions

- **$clog2**. The system function $clog2 should return the ceiling of the log base 2 of the argument (the log rounded up to an integer value). The argument can be an integer or an arbitrary sized vector value. The argument should be treated as an unsigned value, and an argument value of 0 should produce a result of 0.

### 10.2.2 Real math functions

Here below are listed only the most common functions:

- **$ln(x)**. Natural logarithm.
- **$log10(x)**. Decimal logarithm.
- **$exp(x)**. Exponential.
- **$sqrt(x)**. Square root.
- **$pow(x,y)**. x**y.

## 10.3 Severity tasks

SystemVerilog provides special text messaging system tasks that can be used to flag various exception conditions. The tasks are defined as follows:

- **$fatal**. It should generate a run-time fatal error, which terminates the simulation with an error code. Calling $fatal results in an implicit call to $finish.
- **$error**. It should be a run-time error.
- **$warning**. It should be a run-time warning.
- **$info**. It should indicate that the message carries no specific severity

## 10.4 Assertion tasks

### 10.4.1 Assertion control tasks

SystemVerilog provides the following three system tasks to control the evaluation of assertion statements:

- **$assertoff**. It should stop the checking of all specified assertions until a subsequent $asserton. An assertion that is already executing, including execution of the pass or fail statement, is not affected. In the case of a deferred assertion, currently queued reports are not flushed and may still mature, though further checking is prevented until the $asserton. In the case of a pending procedural assertion instance, currently queued instances are not flushed and may still mature, though no new instances may be queued until the $asserton.
- **$assertkill**. It should abort execution of any currently executing specified assertions and then stop the checking of all specified assertions until a subsequent $asserton. This also flushes any queued pending reports of deferred assertions or pending procedural assertion instances that have not yet matured.
- **$asserton**. It should reenable the execution of all specified assertions.

### 10.4.2 Assertion system functions

Assertions are commonly used to evaluate certain specific characteristics of a design implementation, such as whether a particular signal is "one-hot". The following system functions are included to facilitate such common assertion functionality:

- **$onehot**. It returns true if 1 and only 1 bit of expression is high.
- **$onehot0**. It returns true if at most 1 bit of expression is high.
- **$isunknown**. It returns true if any bit of the expression is X or Z.

# Bibliography


[1] Machanick P., n.d. *Approaches To Addressing The Memory Wall.* School of IT and Electrical Engineering, University of Queensland.

[2] Waterman A. and Asanović K., 2017. *The RISC-V Instruction Set Manual Volume I: User-Level ISA.*

[3] Traber A., Gautschi M. and Schiavone P., April 2019. *RI5CY: User Manual.* Micrel Lab and Multitherman Lab University of Bologna, Italy and Integrated Systems Lab ETH Zürich, Switzerland.

[4] Iis-projects.ee.ethz.ch. 2019. *PULP - iis-projects.* [online] Available at: http://iis-projects.ee.ethz.ch/index.php/PULP [Accessed 24 Nov. 2019].

[5] Suri M. et al, n.d. *Applications Of Emerging Memory Technology.*

[6] Vacca M., Tavva Y., Chattopadhyay A. and Calimera A. 2018. *Logic-In-Memory Architecture For Min/Max Search.* Department of Electronics and Telecommunications, Politecnico di Torino (Italy), School of Computer Science and Engineering, Nanyang Technological University (Singapore), Department of Control and Computer Engineering, Politecnico di Torino (Italy)

[7] Santoro G., Turvani G. and Graziano M., 2019. *New Logic-In-Memory Paradigms: An Architectural And Technological Perspective.* Politecnico di Torino (Italy).

[8] Akin B., Franchetti F., Hoe J.C., 2015. *Data reorganization in memory using 3D-stacked DRAM.* ACM SIGARCH Comput. Architect. News (USA).

[9] Santoro G., 2019. *Exploring New Computing Paradigms For Data-Intensive Applications.* Politecnico di Torino (Italy).

[10] Coluccio A., Vacca M. and Turvani G., 2020. *Logic-In-Memory Computation: Is It Worth It? A Binary Neural Network Case Study.* Department of Electronics and Telecommunications, Politecnico di Torino (Italy)

[11] Wu M. and Buchmann A., n.d. *Encoded Bitmap Indexing For Data Warehouses.* DVS1, Computer Science Department Technical University Darmstadt (Germany).

[12] Docs.oracle.com. 2020. *Database Data Warehousing Guide - Indexes.* [online] Available at: https://docs.oracle.com/cd/B28359_01/server.111/b28313/indexes.htm [Accessed 5 July 2020].

[13] Singh A., Agarwal P. and Chand, M., 2017. *Analysis Of Development Of Dynamic S-Box Generation.*

[14] Zhang X. and Parhi K., 2004. *High-Speed VLSI Architectures For The AES Algorithm.*

[15] Mahajan P. and Sachdeva A., 2013. *A Study Of Encryption Algorithms AES, DES And RSA For Security.* Global Journals Inc. (USA).

[16] Business Jargons, 2020. *What Is Least Cost Method? Definition And Meaning*. Business Jargons. [online] Available at: https://businessjargons.com/least-cost-method.html#:~:text=Definition%3A%20The%20Least%20Cost%20Method,the%20least%20cost%20of%20transportation. [Accessed 5 July 2020].

[17] IEEE Computer Society, 2009. *1800-2009 IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language*.

[18] Kumar Tala D., 2014. *SystemVerilog Tutorial*. [online] Asic-world.com. Available at: http://www.asic-world.com/systemverilog/tutorial.html [Accessed 25 Oct. 2019].

[19] Spear C. and Tumbush G., 2012. *SystemVerilog for Verification - A Guide to Learning the Testbench Language Features*. 3rd ed. Springer.

[20] Verificationguide.com, 2019. *SystemVerilog Tutorial*. [online] Available at: https://www.verificationguide.com/p/systemverilog-tutorial.html [Accessed 2 Nov. 2019].

[21] Chipverify.com, 2019. *SystemVerilog*. [online] Available at: https://www.chipverify.com/systemverilog/systemverilog-tutorial [Accessed 6 Nov. 2019].

[22] Systemverilog.io, 2019. *systemverilog.io*. [online] Available at: https://www.systemverilog.io/generate#overview [Accessed 11 Nov. 2019].